



Selenium2 Python

自动化测试项目实战 (第二版)

作者：虫师



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 python 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。之前一位人民邮电出版社的编辑曾联系过我，并向我发送了一份编书的规范，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 编程入门的书。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。

2014. 1. 24

虫师

新前言

在刚做测试的一年多时间里，笔者对各种测试技术和工具有着强烈的兴趣和学习欲望，再加上工作较为空闲，所以有幸接触到 QTP、Selenium 等自动化测试工具，由于当时水平有限，学习也只停留在录制与回放的水平上。再次学习 Selenium 是时隔一年之后，笔者有幸跳槽到一家互联网公司继续做 web 软件测试，发现项目适合做自动化测试，于是再次捡起 Selenium。随着能力和眼界的开阔，发现单纯的使用工具的录制与回放并不能解决实际的问题。Selenium 本身支持多种语言编写脚本，这给我提供了选择的余地。之前一直有打算要学习一门脚本语言，在 Python 与 Ruby 之间犹豫不定。刚好项目组用 Python 开发项目，所以很自然的选择了 Python 和 Selenium 的组合。

从 2013 年开始用 Python 和 Selenium 进行自动化测试的实践，其间的过程颇为艰辛，除了官方文档，相关的资料并不算太多，尤其是中文资料；相比较而言 Java 和 Selenium 的资料要更多一些，这其中主要的原因是 Java 语言更为流行。

后来认识了乙醇和 MarkRabbit，前者的自动化测试文档给了我很大帮助，后者有丰富的 Python Selenium 自动化测试项目经验，而且不厌其烦的帮助后来者。在此对这两次前辈表示深深的敬意。

起初只是将 WebDriver API (Python 版) 对页面一些操作作为博客进行发表。后来积累了十几篇博客，为了方便阅读，将其整理成了一个文档供读免费下载。再后来就是不断的扩充文档的内容，文档命名为《Selenium WebDriver (python)》（其间更新了三个版本）。2013 年的下半年有了出书的念想，当时以为写书并算困难。于是，每天都活在兴奋之间，构思书的结构与内容。到 2013 年年底整理出了一个较为完整的自动化测试的知识体系，内容不再局限于 Selenium 上，加入了大量的 Python 技术的应用。所以重新命名为《Selenium2 Python 自动化测试实战》，我带着心虚在传播这份文档，因为有文档中有不少技术我并没真正“吃透”，自然讲解的不够透彻。

与此同时作者和乙醇的合讲的 Python 与 Selenium 网络课程也在同步进行中。《Selenium2 Python 自动化测试实战》很自然的成为了我的课程的教材。其间收到了大量读者的和学生的反馈，笔者从未停止对这份文档的更新，直到 2013 年 10 月份，我一直在对这份文档做加法，添加新的知识和技术。可是我却越加的不满意，文档中的大量案例已经过时，章节的安排也不够合理。这才有了整理（第二版）的念头。

本以为有（第一版）为基础进行修改应该很轻松的可以完成（第二版），但实际情况却超出了我的预期。为了整理（第二版）我花费了将近 4 个多月的空闲时间，其间又回顾和参考了大量资料。如果前一版在我看来只能叫“文档”的话，那么这一版已经可以上升到“电子书”的水准。

本文档做为自动化测试课程的教材，已经经历了六期的学员，其实用性是不容置疑的。同时也希望得到这本电子书的同学能够真正的学以致用，在自动化测试的道路上有长足的进步。

全书的结构：

全书共分16章，第1章为开始自动化测试之前的所要了解的基础知识。从第2章到第10章，虽然每一章所讲的知识点都不一样，但他们之间有非常强的连贯性和相互依赖性，只能弄懂了这几章的内容，才能开展自动化测试工作。从11章到15章，不管讲多线程技术也好，讲 Git 版本工具也好，有意在提升读者的综合开发的水平。

2015. 2. 25
虫师

声明

本电子书的著作权归虫师个人所有，禁止一切网络的传播和共享，请尊重作者的劳动成果。

目录

Selenium2 Python.....	1
序.....	2
前言.....	3
目录.....	6
第1章 自动化测试基础.....	8
1.1 软件测试分类.....	8
1.2 分层的自动化测试.....	13
1.3 什么样的项目适合自动化测试.....	14
1.4 自动化测试及工具简述.....	15
1.5 Selenium 工具介绍.....	15
1.6 前端技术介绍.....	18
1.7 前端工具介绍.....	20
1.8 开发语言的选择.....	22
第2章 测试环境搭建.....	23
2.1 window 下环境搭建.....	23
2.2 Ubuntu 下环境搭建.....	28
2.3 使用 IDLE 来编写 Python.....	29
2.4 编写第一个自动化脚本.....	32
2.5 安装浏览器驱动.....	33
2.5 不同编程语言下使用 WebDriver.....	34
第3章 Python 基础.....	38
3.1 输出与输入.....	38
3.2 分支与循环.....	41
3.3 数组与字典.....	43
3.4 函数与类、方法.....	44
3.5 模组.....	46
3.6 异常.....	50
第4章 WebDriver API.....	57
4.1 从定位元素开始.....	57
4.2 控制浏览器.....	70
4.3 简单元素操作.....	72
4.4 鼠标事件.....	75
4.5 键盘事件.....	78
4.6 获得验证信息.....	80
4.7 设置元素等待.....	82
4.8 定位一组元素.....	86
4.9 多表单切换.....	90
4.10 多窗口切换.....	92
4.11 警告框处理.....	93
4.12 上传文件.....	95
4.14 下载文件.....	101
4.15 操作 Cookie.....	102

4.16 调用 JavaScript.....	104
4.17 窗口截图.....	106
4.18 关闭窗口.....	107
4.19 验证码的处理.....	107
4.20 WebDriver 原理.....	110
第 5 章 自动化测试模型.....	114
5.1 自动化测试模型介绍.....	114
5.2 模块化实例.....	118
5.3 数据驱动实例.....	120
第 6 章 Selenium IDE.....	131
6.1 Selenium IDE 安装.....	131
6.2 Selenium IDE 界面介绍.....	133
6.3 创建测试用例.....	135
6.4 Selenium IDE 命令.....	139
6.5 断言与验证.....	142
6.6 等待与变量.....	145
第 7 章 unittest 单元测试框架.....	149
7.1 分析带 unittest 自动化测试脚本.....	149
7.2 unittest 单元测试框架解析.....	154
7.3 用 unittest 编写 web 自动化.....	171
7.4 用例执行的疑惑.....	174
第 8 章 自动化测试项目实战.....	179
8.1 自动化测试用例设计.....	179
8.2 126 邮箱项目实战.....	181
8.3 扩展自动化测试用例.....	191
第 9 章 自动化测试高级应用.....	200
9.1 使用 HTMLTestRunner 生成测试报告.....	200
9.2 创建定时任务.....	209
9.3 自动发邮件功能.....	220
第 10 章 Selenium Grid2.....	231
10.1 Selenium 工作原理.....	231
10.2 Selenium Server 环境配置.....	233
10.3 Selenium Grid 工作原理.....	237
10.4 Selenium Grid 应用.....	242
10.5 WebDriver 驱动.....	247
第 11 章 Python 多线程.....	251
11.1 单线程的时代.....	251
11.2 多线程技术.....	253
11.3 多进程技术.....	259
11.4 应用于自动化测试.....	264
第 12 章 Page Object 设计模式.....	271
12.1 认识 Page Object.....	271
12.2 Page Object 实例.....	272
第 13 章 BDD 框架之 lettuce 入门.....	280
13.1 安装 lettuce.....	281

13.2 认识 BDD(lettuce).....	281
13.3 添加测试场景.....	287
13.4 lettuce 目录结构与执行过程.....	289
13.5 lettuce WebDriver 自动化测试.....	291
第 14 章 Git 管理项目.....	296
14.1 Git/GitCafe 托管测试项目.....	296
14.2 Git/Git Server 搭建.....	309
第 15 章 持续集成 Jenkins 入门.....	317
15.1 环境搭建.....	318
15.2 创建任务.....	321
15.3 运行构建.....	325
附录.....	328
XPath 语法.....	328
CSS 选择器参考手册.....	330
Python 编辑器之 UliPad.....	332
Python 编辑器之 Sublime.....	333
Sublime 使用技巧.....	334
参考.....	340

第 1 章 自动化测试基础

在开始本书的学习之前，我们很有必要了解一些关于自动化测试相关的基础知识，这将为我们后面章节的学习打基础。

1.1 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

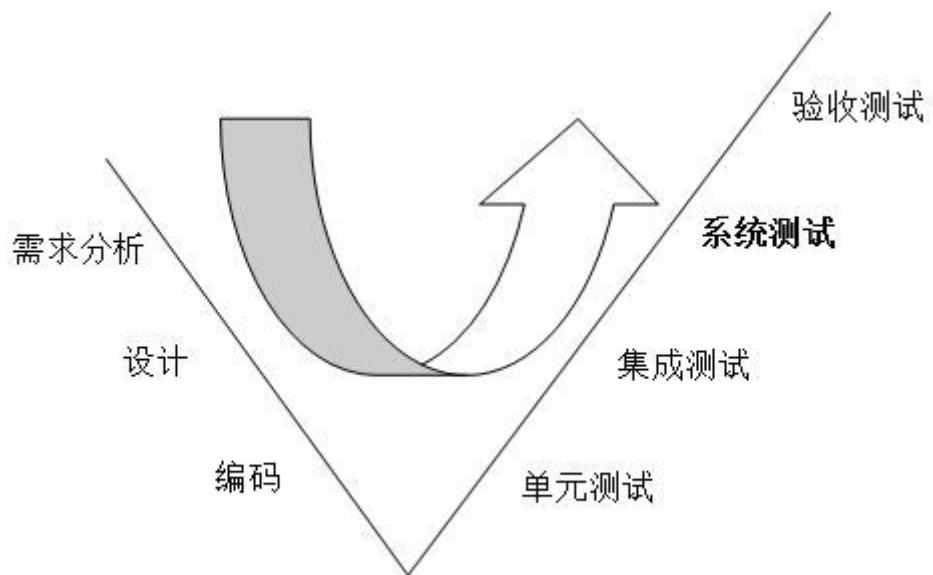


图 1.1 项目流程与对应的测试

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。

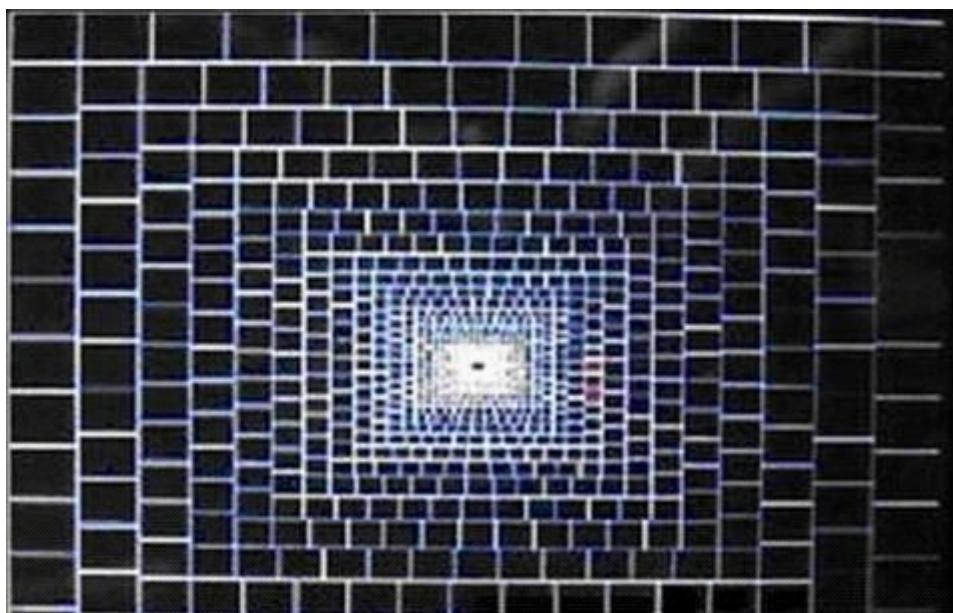


图 1.2 黑盒测试与白盒测试

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

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。

1.2 分层的自动化测试

传统的自动化测试更关注产品 UI 层的自动化测试，而分层的自动化测试倡导产品开发的不同阶段（层次）都需要自动化测试。



图 1.3 分层自动化测试

相信测试同学对上面的金字塔并不陌生，这个就是产品开发各个同阶段所对应的测试！虽然这个模型并不新鲜，或者经常被各种测试书中提到，但实际生产中，大多公司与研发团队其实是忽略了单元测试与集成测试阶段的自动化测试工作，所以，在分层的自动化测试中，我们有必要对这些定义重新理解和定义。

单元测试：我们需要规范的来做单元测试同样需要相应的单元测试框架，如 java 的 Junit、testNG，C# 的 NUnit，Python 的 unittest、pytest 等，几乎所有的主流语言，都会有其对应的单元测试框架。

集成、接口测试：对于不少测试新手来说不太容易理解，单元测试关注代码的实现逻辑，例如一个 if 分支或一个 for 循环的实现；那么集成、接口测试关注的是一是个函数、类（方法）所提供的接口是否可靠。例如，我定义一个 add() 函数用于计算两个参数的结果并返回，那么我需要调用 add() 并传参，并比较返回值是否两个参数相加。当然，接口测试也可以是 url 的形式进行传递。例如，我们通过 get 方式向服务器发送请求，那么我们发送的内容做为 URL 的一部分传递到服务器端。但比如 Web service 技术对外提供的一个公共接口，需要通过 soapUI 等工具对其进行测试。

UI 层的自动化测试：这个大家应该再熟悉不过了，大部分测试人员的大部分工作都是对 UI 层的功能进行测试。例如，我们不断重复的对一个表单提交，结果查询等功能进行测试，我们可以通过相应的自动化测试工具来模拟这些操作，从而解放重复的劳动。UI 层的自动化测试工具非常多，比较主流的是 QTP，Robot Framework、watir、Selenium 等。

为什么要画成一个金字塔形，则不是长方形 或倒三角形呢？这是为了表示不同阶段所投入自动化测试的比例。如果一个产品从没有做单元测试与接口测试，只做 UI 层的自动化测试是不科学的，很难从本质上保证产品的质量。如果你妄图实现全面的 UI 层的自动化测试，那更是一个劳民伤财的举动，投入了大量人力时间，最终获得的收益可能会远远低于所支付的成本。因为越往上层，其维护成本越高。尤其是 UI 层的元素会时常的发生改变。所以，我们应该把更多的自动化测试放在单元测试与接口测试阶段进行。

既然 UI 层的自动化测试这么劳民伤财，那我们只做单元测试与接口测试好了。NO！因为不管什么样的产品，最终呈现给用户的是 UI 层。所以，测试人员应该更多的精力放在 UI 层。那么也正是因为测试人员在 UI 层投入大量的精力，所以，我们有必要通过自动化的方式帮助我们“部分解放”重复的劳动。

在自动化测试中最怕的是变化，因为变化的直接结果就是导致测试用例的运行失败，那么就需要对自动化脚本进行维护；如何控制失败，降低维护成本对自动化的成败至关重要。反过来讲，一份永远都运行成功的自动化测试用例是没有任何价值。

至于在金字塔中三种测试的比例要根据实际的项目需求来划分。在《google 测试之道》一书，对于 google 产品，70%的投入为单元测试，20%为集成、接口测试，10% 为 UI 层的自动化测试。

1.3 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都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感到困难重重，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 条都具备的情况下才能开展测试工作。这里就需要读者做综合的权衡。在我们普遍的自动化测试经验中，一般满足三个条件就可以对项目开展自动化测试：

软件需求变动不频繁

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开发代码的过程，需要修改、调试，必要的时候还要修改自动化测试的框架，如果所花费的成本高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、脚本的开发与调试均需要时间来完成。这样的过程本身就是一个测试软件的开发过程。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。

1.4 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；广义上来说所有借助工具来进行软件测试都可以称为自动化测试；狭义上来说，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

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。

1.5 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。

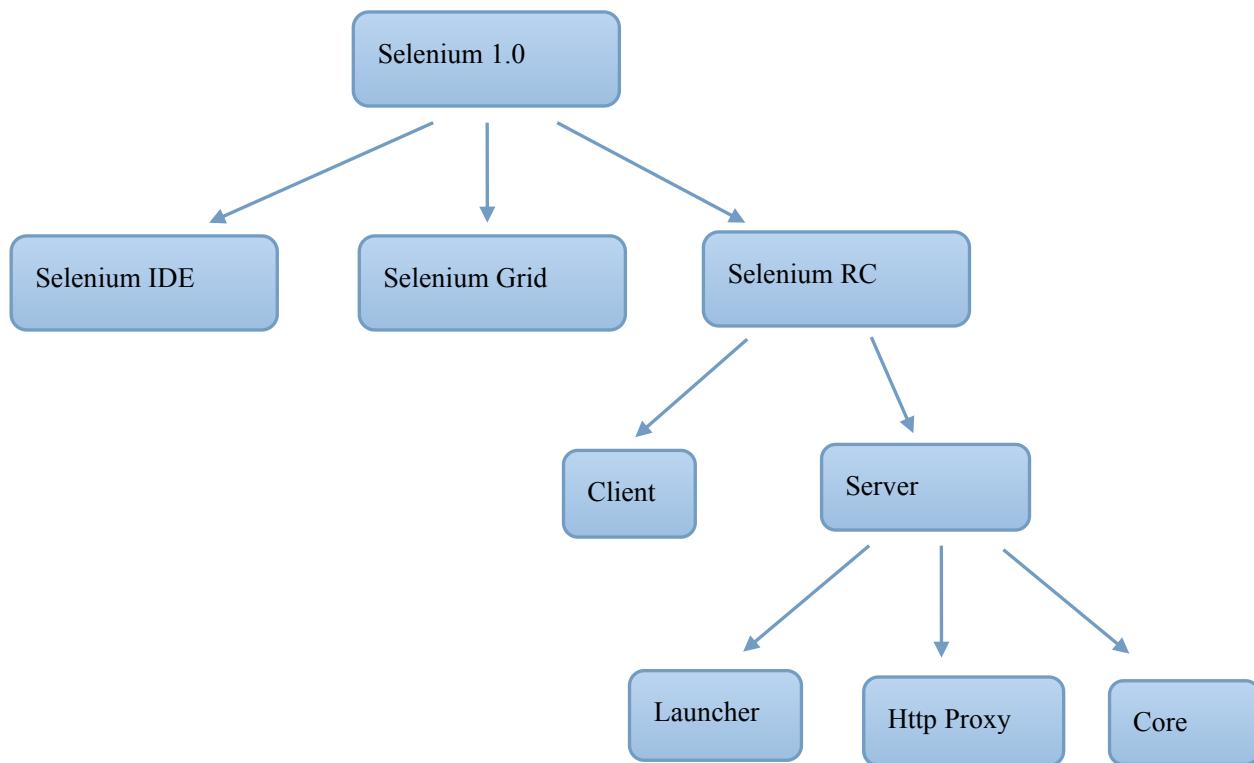


图 1.4 Selenium 家谱

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：

$$\text{Selenium 2.0} = \text{Selenium 1.0} + \text{WebDriver}$$

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。

Selenium 与 WebDriver 合并原因：

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。

1.6 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 web 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_page.html

```
<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（脚本语言是一个种轻量级的语言），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：

js_page.html

```
<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_file.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，才能传递、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到，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。

1.7 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。

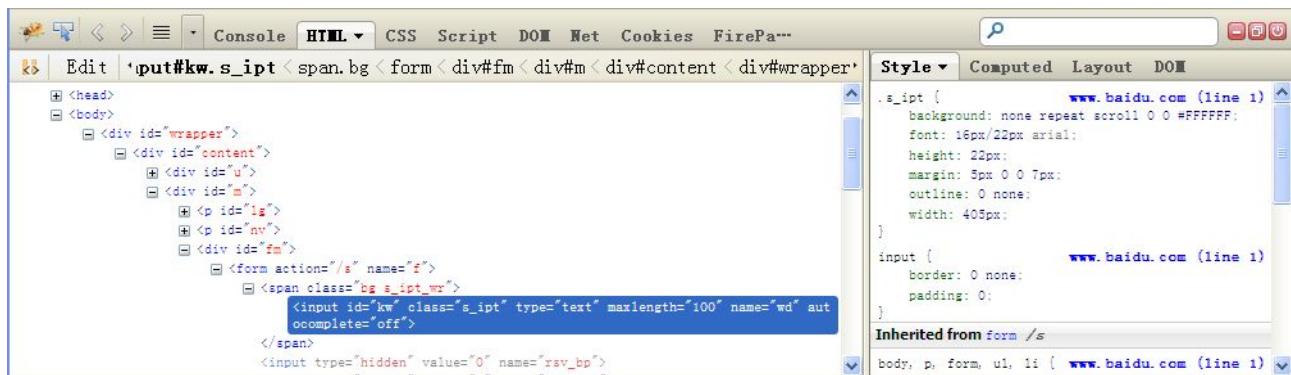


图 1.5 FireBug

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。

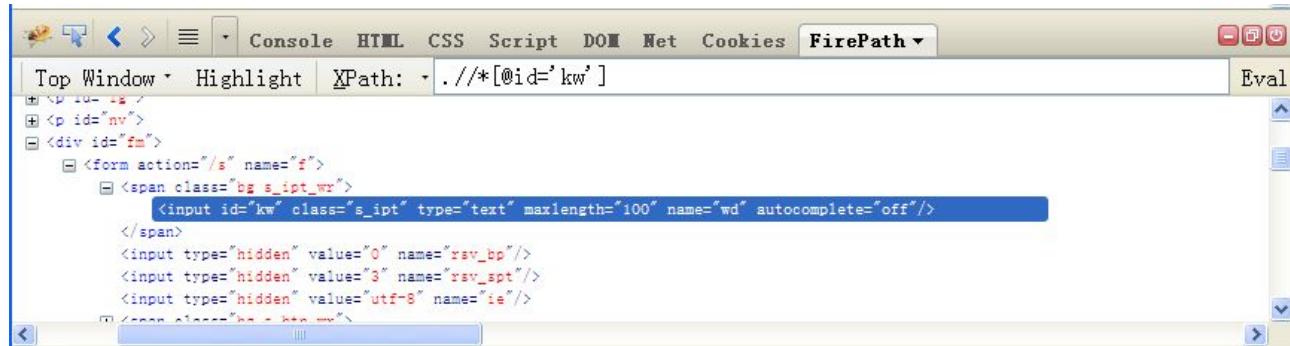


图 1.6 FirePath

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。

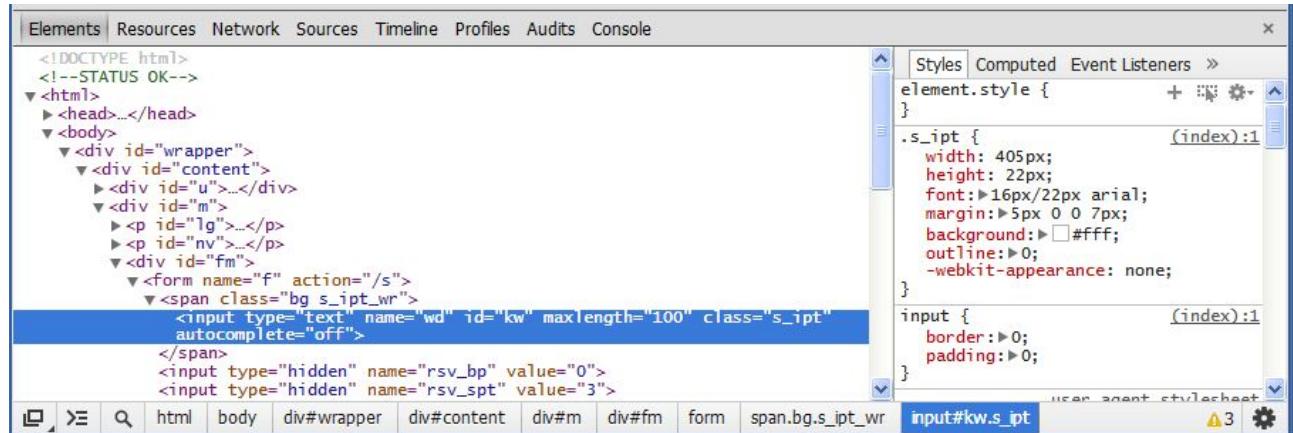


图 1.7 Chrome 开发者工具

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。

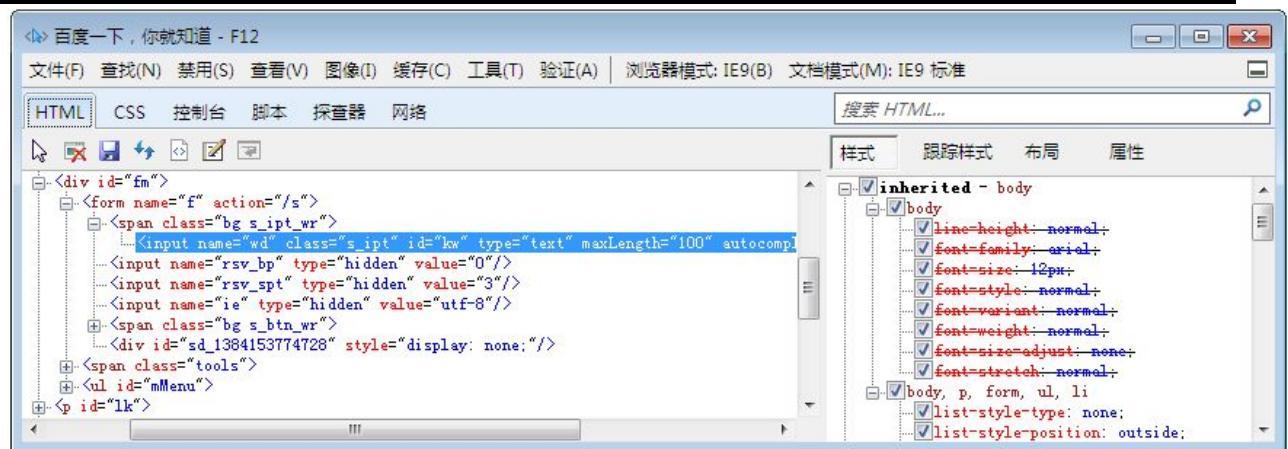


图 1.8 IE 开发者工具

1.8 开发语言的选择

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 2 章 测试环境搭建

也许你已经迫不及待的坐在了电脑面前，想要跟着我开始我们自动化测试之旅，不要着急，在此之前我们安装好测试环境，如果你选择了本书，那么你已经确定使用 Python 和 Selenium 来做自动化测试了。

2.1 window 下环境搭建

如果是想要学习一门编程语言，我们只用到官方网站上去下载最新版本安装就可以了，但对于想要学习 Python 的同学将会面临一个版本选择的问题，因为 Python 同时存在着两个版本（Python2 和 Python 3），而这两个版本目前处于并行更新状态。

之所以会有两个版本并存的情况，是因为随着近几年 Python 语言的逐渐流行起来，早期的 Python 为版本在基础的设计存在着一些不足之处，Python3 在设计的时候很好的解决了这些遗留问题，并且在性能上也有了很大的提升，但同时带来了新的问题就是不完全向后兼容，所以就造成了两个版本并存的情况。

就目前的情况来看，两个版本的更新与维护都在继续，因为目前大量的类库都是基于 Python2 开发的，过度 Python3 还需要些时间，当然 Python3 一定是未来的发展方向，但就就当前来看 Python2 依然是主流，所以本书的所有代码都以基于 Python2 版本编写。

2.1.1 安装 Python

访问 Python 官方网站：<https://www.Python.org/>

下载最新版本的 Python2，截止作者发稿，最新版本为 Python2.7.8 版本。读者根据自己的平台选择相应的版本进行下载；对于 Windows 用户来说，如果你的系统是 32 位的请选择 x86 版本，如果是 64 位系统请选择 64 版本进行下载。下载完成会得到一个以.msi 为后缀名的文件，双击进行安装。如图 2.1。



图 2.1 Python 安装界面

安装过程与其它 Windows 程序一样，安装完成在开始菜单中将看到安装好的 Python 目录：

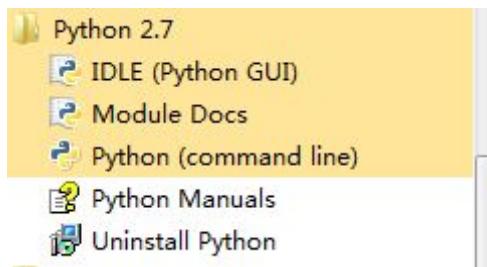


图 2.2 Python 菜单

打开 Python 自带的编辑器 IDLE 就可以编写 Python 程序了：

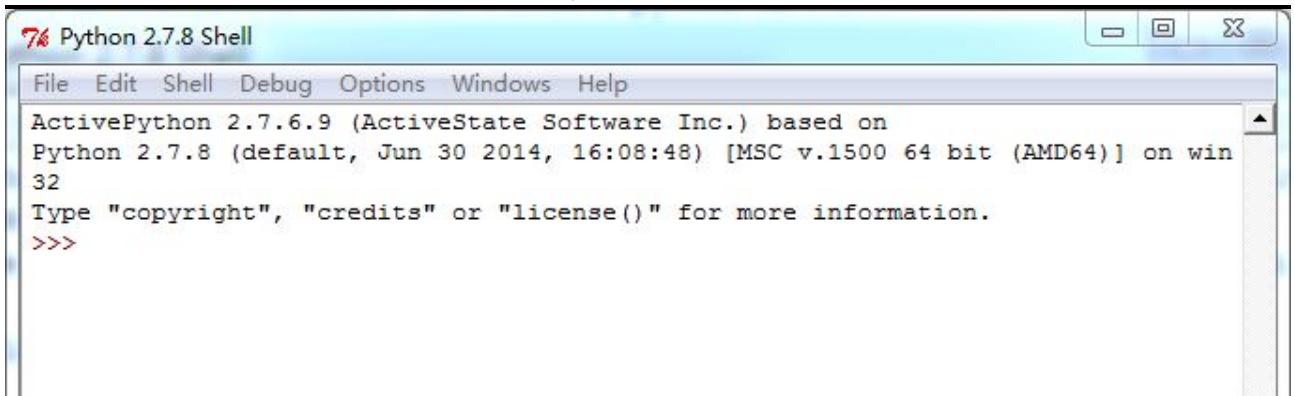


图 2.3 Python Shell 界面

或者通过在 Windows 命令提示符下输入“Python”命令，也可以进入 Python Shell 模式。如下：

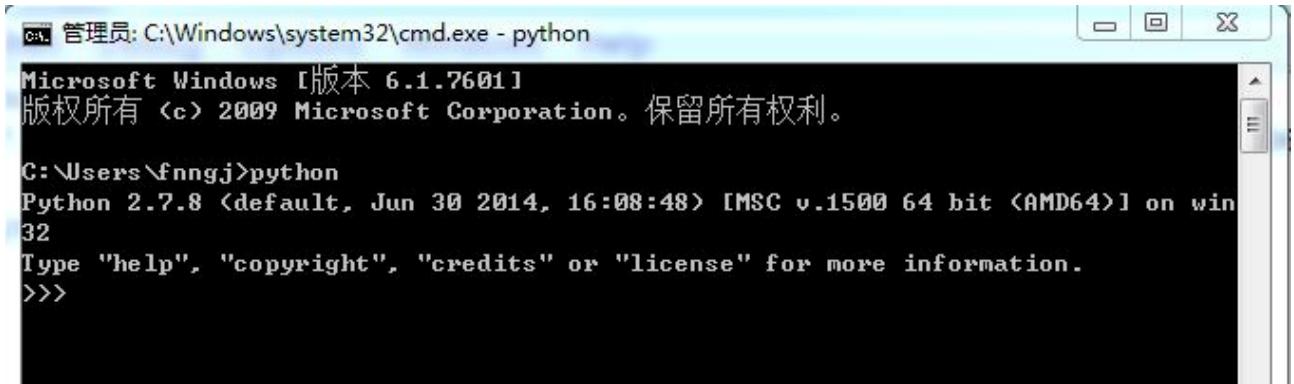


图 2.4 命令提示符下进入 Python Shell 界面

小提示：

如果提示 Python 不是内部或外部命令！别急，把 Python 的安装目录添加到系统环境变量的 Path 下面。桌面“我的电脑”右键菜单-->属性-->高级-->环境变量-->系统变量的 Path 中添加：

变量名：PATH

变量值：;C:\Python27

2.1.2 安装 steuptools 与 pip

setuptools 是 Python Enterprise Application Kit (PEAK) 的一个副项目，它是一组 Python 的 distutilsde 工具的增强工具可以让程序员更方便的创建和发布 Python 包，特别是那些对其他包具有依赖性的状况。

经常接触 Python 的同学可能会注意到，当需要安装第三方 Python 包时，可能会用到 easy_install 命令。easy_install 是由 PEAK 开发的 setuptools 包里带的一个命令，所以使用 easy_install 实际上是在调用 setuptools 来完成安装模块的工作。

pip 是一个安装和管理 Python 包的工具，通过 pip 去安装 Python 包将变得十分简单，我们将省去了搜索--查找版本--下载--安装等繁琐的过程。pip 的安装依赖于 setuptools，所以在安装 pip 之前需要先安装 setuptools。需要注意的是目前 Python3 并不支持 setuptools，需要使用 distribute。

setuptools 与 pip 下载地址：

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

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

通过上面的地址进行下载，将得到下面两个包（随着时间包的版本号会有变化）。

setuptools-7.0.zip

pip-1.5.6.tar.gz

通过解压工具进行解压将得到两个文件夹，在 Windows 命令提示符进入到文件解压目录，通过 Python 执行安装文件 setup.py 进行安装。安装 setuptools：

cmd.exe

```
C:\package\setuptools-7.0>Python setup.py install
```

安装 pip 的方法与 setuptools 相同，切换到 pip 解压目录，运行 setup.py 文件：

cmd.exe

```
C:\package\pip-1.5.6>Python setup.py install
```

安装完成，在 Windows 命令提示符下敲入 pip 命令：

cmd.exe

```
C:\package\pip-1.5.6>pip
Usage:
  pip <command> [options]

Commands:
  install           Install packages.
  uninstall        Uninstall packages.
  freeze           Output installed packages in requirements format.
  list              List installed packages.
  show              Show information about installed packages.
  search            Search PyPI for packages.
  wheel             Build wheels from your requirements.
  zip               DEPRECATED. Zip individual packages.
  unzip            DEPRECATED. Unzip individual packages.
  bundle            DEPRECATED. Create pybundles.
  help              Show help for commands.

General Options:
  -h, --help          Show help.
  -v, --verbose       Give more output. Option is additive, and can be
                      used up to 3 times.
  -V, --version       Show version and exit.
  -q, --quiet         Give less output.
  --log-file <path>   Path to a verbose non-appending log, that only
                      ....
```

如果出现 pip 命令的说明信息，则说明我们已经安装成功。如果提示 pip 不是内部或外部命令，pip 的

可以执行文件在 C:\Python27\Scripts\目录下面，根据前面的小提示，我们将目录添加到系统环境变量下的 Path 下面即可。

2.1.3 安装 Selenium

Selenium 这里就不再过多介绍，前面 pip 的安装是为了更方便的安装 Selenium 包，直接通过 pip 命令安装 Selenium 包：

cmd.exe

```
C:\Python27\Lib\site-packages>pip install Selenium
```

我们费尽周折安装 pip 好处是可以使用 pip 命令安装 Python 第三方库中的任何包，就像当前安装 Selenium 一样方便，如果只输入包名默认安装当前库中最新的版本，如果我们不想安装最新版本的包，可以在包名后面加版本号。

cmd.exe

```
C:\Python27\Lib\site-packages>pip install Selenium==2.42.1
.....
C:\Python27\Lib\site-packages>pip show Selenium
---
Name: Selenium
Version: 2.42.1
Location: c:\Python27\lib\site-packages
Requires:
```

pip 下面包含了很多命令，正如我们上面只输入一个有 pip 回车所得到的，show 就是其中一个，我们可以用 show 查看安装包的版本及路径。

2.1.4 ActivePython

ActivePython 是由 ActiveState 公司推出的专用的 Python 编程和调试工具。

ActivePython 包含了完整的 Python 内核，直接调用 Python 官方的开源内核，此外还有 Python 编程需要用到的 IDLE，并附加了一些 Python 的 Windows 扩展，同时还提供了全部的访问 Windows APIs 的服务。ActivePython 虽然不像纯 Python 那样是开源的，但是也可以免费下载使用。

使用 ActivePython 的好处是它集成了 pip 包管理工具，直接可以通过 pip 命令来安装 Python 第三方包。

访问 ActivePython 下载地址：

<http://www.activestate.com/activePython/downloads>

ActivePython 同样支持 Windows、Mac 和 Linux 等平台，请根据自己的平台下载相应的版本。

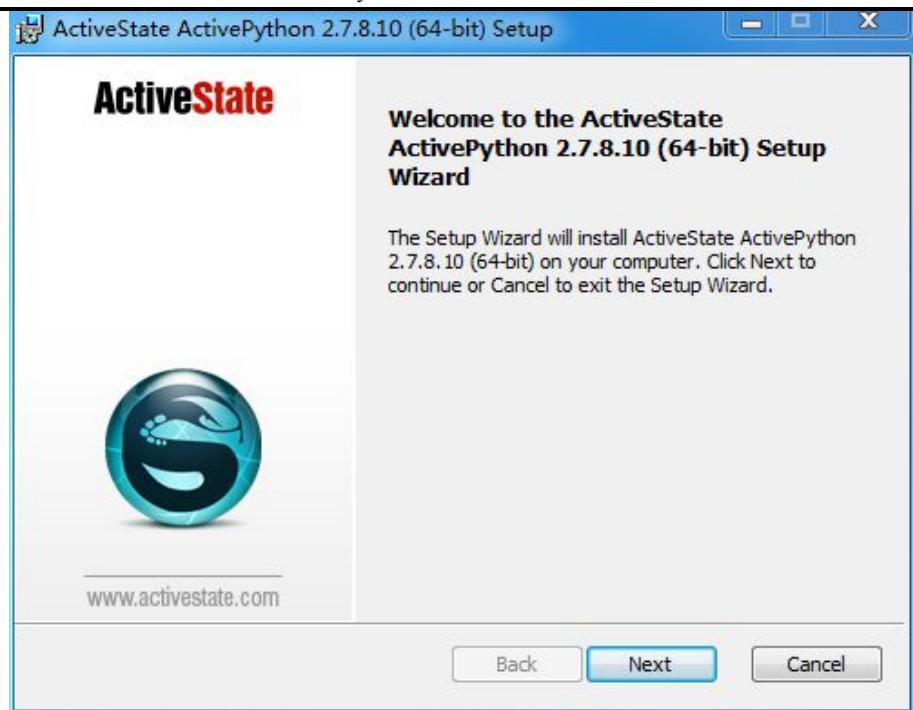


图 2.5 ActivePython 安装界面

ActivePython 的与 Python 的安装过程相同，安装完成同样会在 Windows 开始菜单中生成相应的菜单项。

安装 ActivePython 后可以直接使用 pip 命令安装 Selenium 包，过程同上，这里就不再详细阐述。对于新手来说 ActivePython 无疑会更加方便，所以作者推荐使用 ActivePython。

2.2 Ubuntu 下环境搭建

Linux 操作系统的版本很多，这里以流行的 Ubuntu 系统为例，介绍在其下面的安装过程。

因为 Ubuntu 系统本身对 Python 有很强的依赖，所以 Ubuntu 自带的就有 Python，笔者曾不小心卸载了 Ubuntu 系统自带的 Python 导致系统无法正常启动，这一点也说明了 Python 在各领域都有非常广泛的应用。

在 Ubuntu 中使用 Python 非常简单，打开终端下输入“Python”命令回车，就可以进入 Python Shell 模式了。

ubuntu 终端

```
fnngj@fnngj-PC:~$ Python
Python 2.7.3 (default, Sep 26 2012, 21:53:58)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

下面我们在 Ubuntu 下安装 setuptools 与 pip，因为 setuptools 已经存在于 Ubuntu 的软件仓库之中，所以可以使用 apt-get 命令进行安装。apt-get 是 debian, Ubuntu 发行版 Linux 系统的包管理工具。

安装 setuptools 方法如下：

ubuntu 终端

```
fnngj@fnngj-PC:~$ sudo apt-get install Python-setuptools
```

小提示：

apt-get 命令一般需要 root 权限执行，所以在只用 apt-get 命令之前需要先切换到 root 用户，如果不想要切换 root 用户可以在命令前加 sudo 命令。sudo 命令是允许系统管理员让普通用户执行一些或者全部的 root 命令的一个工具。

例：sudo apt-get xxxx

下面可以以同样的方法安装 pip

ubuntu 终端

```
fnngj@fnngj-PC:~$ sudo apt-get install Python-pip
```

如果通过 apt-get 命令无法安装，请参考 Windows 下面的安装方式，先到 Python 官方网站下载相应包，解压执行 setup.py 文件进行安装。

2.3 使用 IDLE 来编写 Python

通过上面繁琐的配置我们终于搭建好需要的自动化开发环境，那么你一定迫不及待要跟着作者一起写自动化脚本了，别急！在此之间我们需要先找到合适的编辑器，如果你是一位编程老手，那么你一定有自己趁手的编辑器，如果是个编程菜鸟，那么 Python 自带的 IDLE 是个不错的入门之选。

IDLE (Python GUI) 是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)，可以在其上面进行编程练习。

启动 IDLE 时，会显示“三个尖括号”提示符 (>>>)，可以在这里输入代码。在 Python Shell 输入代码回车后会立即执行，并直接在下面显示执行的结果。如图 2.5

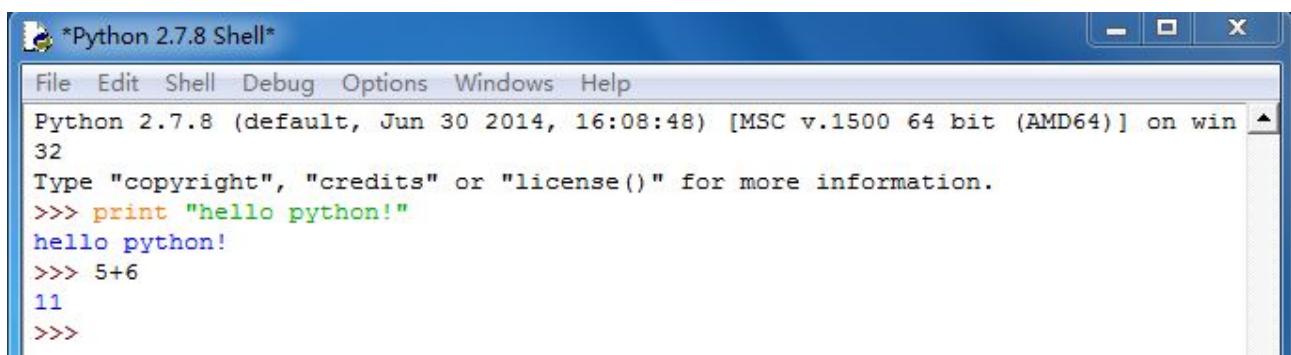


图 2.6 Python Shell 下输入代码

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

TAB 键自动补全：

先键入 Python 关键字的前面几个字母，，然后按下 TAB 键。IDLE 会提供与之相关的关键字，通过键盘上下键进行选择。从而提交代码输入速度以及避免输入错误。

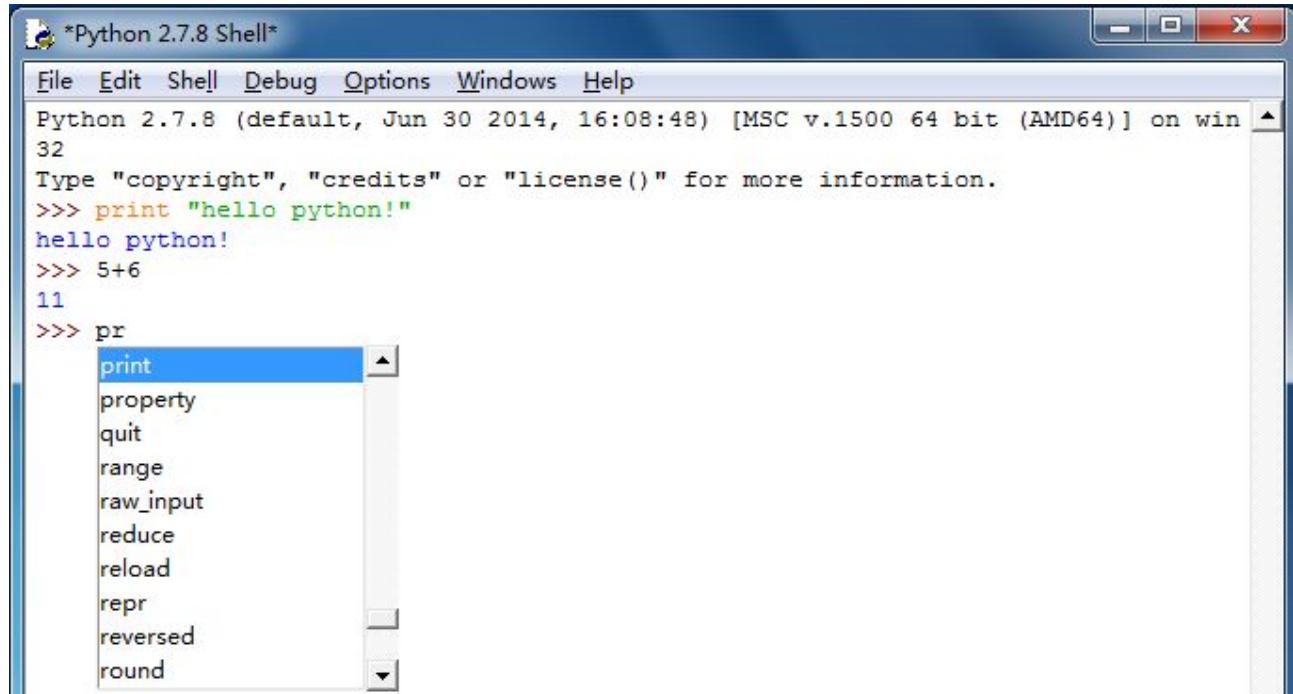


图 2.7 Tab 键自动补全

回退代码语句：

可以通过键盘快捷键 Alt+P 回退到上一次编辑的 Python 代码，Alt+ N 与之相反，可以前进至下一次编辑的代码。如果在 Python Shell 模式下代码不小心写错导致执行错误，那么通过回退修改要比重新输入一遍高效得多。如图 2.7。

```

*Python 2.7.8 Shell*
File Edit Shell Debug Options Windows Help
Python 2.7.8 (default, Jun 30 2014, 16:08:48) [MSC v.1500 64 bit (AMD64)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> a = "python"
>>> for i in a:
    print i
    ↑
p
y
t
h
o
n
>>> for i in a:
    print i

```

图 2.8 Alt+P 回退

在 Python Shell 模式下编写的代码只停留于内存当中，当关闭 Python Shell 后会自动消失，那么我们想把代码写到文件里保存起来，可以通过菜单栏 File--->New File 或通过快捷键 Ctrl+N 打开新的窗口，在此文件中编写的代码，完成后选择菜单栏 File--->Save 或通过快捷键 Ctrl+S 保存。

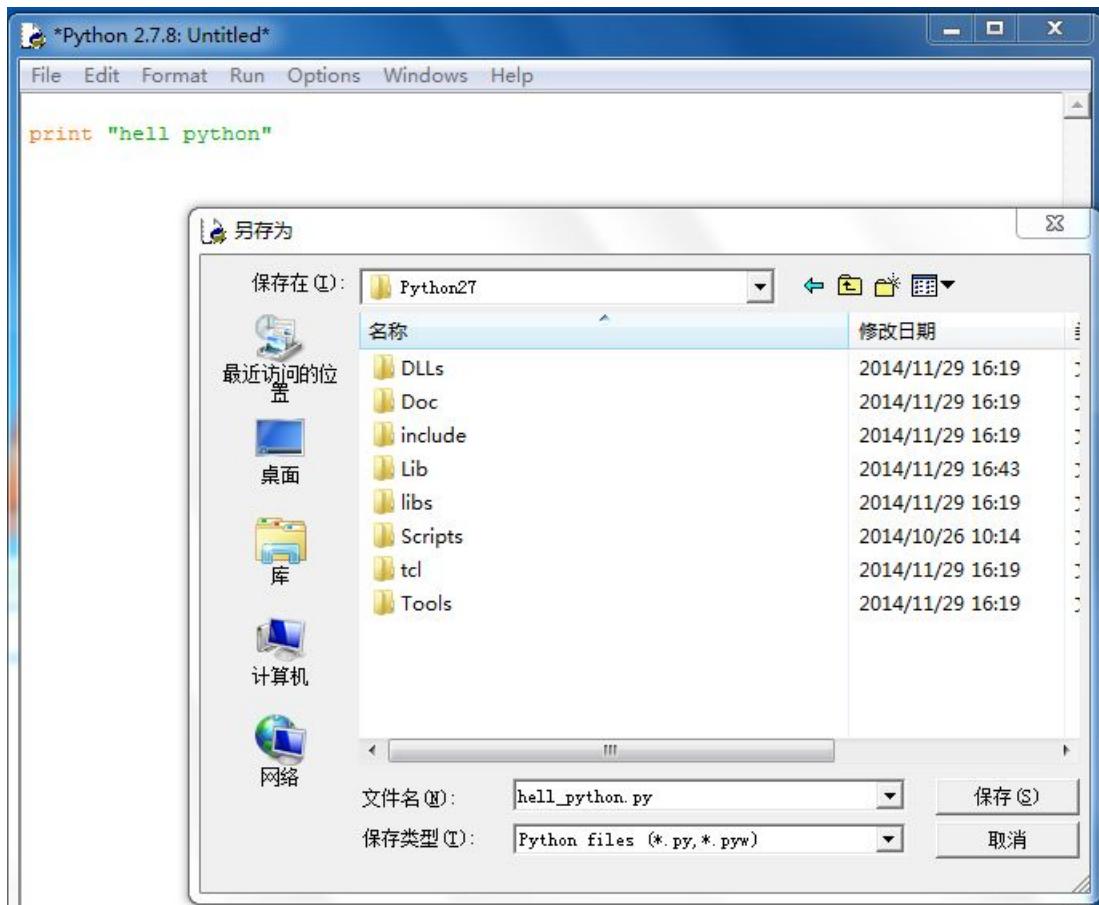


图 2.9 保存 Python 程序

需要注意的是在文件保存时，一定要加上文件后缀名.py；否则文件中代码的着色效果将消失。

2.4 编写第一个自动化脚本

掌握了 Python 编辑器的初步使用之后，我们就可以开始编写自动化脚本了。下面通过 IDLE 新建一个文件，跟着我来编写我们的第一个自动化脚本。

baidu.py

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

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

driver.find_element_by_id("kw").send_keys("Selenium2")
driver.find_element_by_id("su").click()
driver.quit()
```

在敲写这段代码的时候，你心里一定充满了疑问，这段代码到底做了什么事情，下面就来逐行的解释一下代码的含义。

#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。注意等号两边不要留空格，否则将不起作用。除此之外，下面的写法也可以达到相同的作用。

-*- coding: 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。在 Python 下面通过 from... Import ...或 import...引入包，更专业的叫法为：模组（modules）

driver=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。

driver.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（URL）。

```
driver.find_element_by_id("kw").send_keys("Selenium2")
```

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，并通过键盘输入方法 send_keys() 向百度输入框里输入 Selenium2 搜索关键字。

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

这一步通过 id=su 定位“百度一下”搜索按钮，并向搜索按钮发送单击事件 click()。

```
driver.quit()
```

退出并关闭窗口的每一个相关的驱动程序。

保存文件为 baidu.py，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“Selenium2”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），如图 2.9



图 2.10 自动化脚本启动浏览器

2.5 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。除此之外它还包括一个基于 HtmlUnit 的无界面实现，相关驱动为 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 文件放到系统环境变量 Path 下面，前面我们已经将 (C:\Python27) 添加到了系统环境变量 Path 下面，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.xx.zip, 将解压得到 IEDriverServer.exe，同样放置到 C:\Python27\ 目录下。

在 Linux 系统下，同样下载系统对应的浏览器驱动，并将浏览器驱动放置到环境变量 Path 所设置的路径下，不同的 Linux 环境变量的设置也会有所区别，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。

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

替换为：

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

或

```
driver = 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。

此外，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。

2.5 不同编程语言下使用 WebDriver

W3.org 对 WebDriver API 对做定义和规范。

WebDriver API 是独立于平台 (Windows\Linux\MAC) 和语言 (Java/c#/Ruby/Python) 的，它定义了接口和相关的通讯协议。允许脚本或程序通过本规范实现对 web 浏览器行为的控制。

该 WebDriver API 通过通讯协议和一组接口来发现页面上的 DOM 元素中定义的操作，包括控制浏览

器的行为。

我们可以这样来理解，比如，插板和插头，国标标准定义插板和插头标准，比如，插头与插版的规格，材质，大小等。那么所有的电器的厂商生产的插头与所有插板厂商生产的插板都按照这套标准来设计。所以，我们拿到任何一个合格的插头和插板都可以匹配得上。

WebDriver API 可以理解成对操作浏览器和页面元素的一套“国标”。那么不同的编程语言都可以按照这套标准实现自己的语言的 WebDriver 模块。

下面看一下不同编程语言下实现百度搜索的例子。

在 java 中引入 Selenium WebDriver 实现自动化测试：

baidu.java

```
//添加 Selenium(webdriver) 引用
import org.openqa.Selenium.By;
import org.openqa.Selenium.WebDriver;
import org.openqa.Selenium.WebElement;
import org.openqa.Selenium.firefoxB.*;

public class TestHelloWorld {

    public static void main(String[] args) {

        WebDriver driver = new FirefoxDriver();
        driver.get("http://www.baidu.com/");

        WebElement txtbox = driver.findElement(By.name("wd"));
        txtbox.sendKeys("Glen");

        WebElement btn = driver.findElement(By.id("su"));
        btn.click();

        driver.close();
    }
}
```

在 C#中引入 Selenium WebDriver 实现自动化测试：

baidu.cs

```
//添加 Selenium (webdriver) 的引用
using OpenQA.Selenium;
using OpenQA.Selenium.Firefox;
```

```

using OpenQA.Selenium.Support.UI;

namespace SeleniumTests
{
    class Baidu
    {
        static void Main(string[] args)
        {
            driver = new FirefoxDriver();
            url = "http://www.baidu.com/";
            driver.Navigate().GoToUrl(url)

            var searchBox = driver.FindElementById("kw");
            searchBox.SendKeys("Selenium");

            var btnClick = driver.FindElementById("su");
            btnClick.Click();

            driver.Quit();
        }
    }
}

```

在 Ruby 中引入 Selenium WebDriver 实现自动化测试:

baidu.rb

```

#导入 Selenium(webdriver) 包
require 'Selenium-webdriver'

driver = Selenium::WebDriver.for:chrome
driver.get "http://www.baidu.com"

driver.find_element(:id, 'kw').send_keys "Hello WebDriver!"
driver.find_element(:id, 'su').click
driver.quit

```

在不同的编程语言中语法会有一定差异，我们抛去语法的差异性，在不同的语言中实现百度搜索的自动化实例都完成了下面几个操作。

- 1、首先导入 Selenium (webdriver) 相关模块
- 2、调用 Selenium 的浏览器驱动，获取浏览器句柄 (driver) 并启动浏览器。
- 3、通过句柄访问百度 URL。

4、通过句柄操作页面元素（百度输入框和“百度一下”按钮）。

5、通过句柄关闭浏览器。

第3章 Python 基础

虽然本书是以自动化测试技术为主线的教程，但本书中所涉及到的所有代码都是以 Python 语言去实现的，所以，我们在这一章来学习和使用 Python，需要说明的是，本章默认你具备了一定编程的基础，我们不会花费时间来告诉你什么是变量，什么是运算符等基础的概念；你可以找一本 Python 基础教程来学习这些知识。

当然，如果你有一定编程基础，Python 对你来说将非常容易掌握。相信这一章的学习一定可以让你快速入门。如果你已经完全掌握了 Python 的使用，那么本章你可以直接跳过。

我们在上一章中提到 Python 自带编辑器 Python IDLE 的使用，那么我们后面的所有练习也将在这个编辑器下完成。

3.1 输出与输入

一般编程语言的教程都从打印“Hello World！”开始，我们这里也不免俗套，从打印开始。

3.1.1 print 打印

Python 提供 print 方法来打印信息，下面打开 Python shell 来打印一些信息。

Python shell

```
ActivePython 2.7.6.9 (ActiveState Software Inc.) based on
Python 2.7.8 (default, Jun 30 2014, 16:08:48) [MSC v.1500 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print "hello"
hello
```

调用 print 方法，用户双引号（" "）把需要打印输出的信息引起来就被输出了。可是，我给你什么信息，你就打印什么，这个很无聊！那我们就打印点有意思的。让打印信息向你问好。

Python shell

```
>>> name = "zhangsan"
>>> print "hello %s ,Nice to meet you!" %name
hello zhangsan ,Nice to meet you!
>>> name = "Lisi"
>>> print "hello %s ,Nice to meet you!" %name
hello Lisi ,Nice to meet you!
```

虽然，我们两次的打印语句一样，但由于定义的变量 name 两次赋值不同，那么打印出的结果也不完全相同。%s (string) 只能打印字符串，如果想打印数字，那么就要使用%d (data)。

Python shell

```
>>> age=27
>>> print "You are %d !" %age
```

You are 27 !

但是，有时候我们并不知道自己要打印的是什么类型的数据，那么可以用%r 来表示。

Python shell

```
>>> n = 100
>>> print "You print is %r ." %n
You print is 100 .
>>> n = "abc"
>>> print "You print is %r ." %n
You print is 'abc' .
```

3.1.2 input 输入

其实，上面的例子也没意思，打印的变量信息还不是我们事先定义好的。比如 name= “zhangsan” 。那我希望打印的信息在程序运行的过程中由用户输入来。Python 提供了 input 方法来接收用户输入的信息。创建一个.py 文件保存。输入下面的内容：

pr.py

```
#coding=utf-8
n = input("Enter any content: ")
print "Your input is %r " %n
```

按键盘 F5 运行程序，当运行到 input() 时，则需要用户输入一些信息。print 将会把用户输入的内容打印出来。

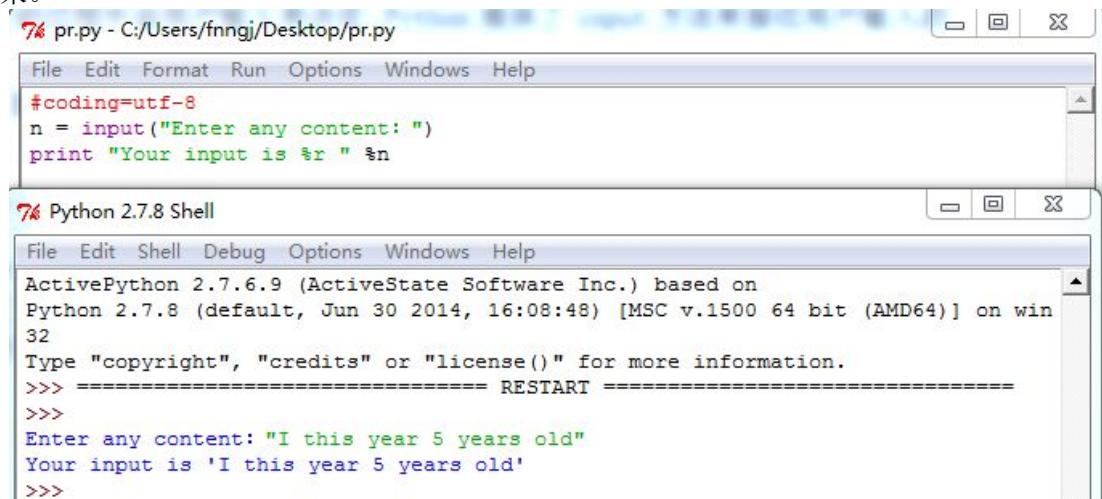


图 3.1 input 输入

但 input()方法比较矫情，要求用户输入的数据类型一定正确，比如字符串一定要加引号，如何不加将会报错。

Python shell

```
>>> ===== RESTART =====
>>>
Enter any content: I this year 5 years old

Traceback (most recent call last):
  File "C:/Users/fnngj/Desktop/pr.py", line 2, in <module>
    n = input("Enter any content: ")
  File "<string>", line 1
    I this year 5 years old
           ^
SyntaxError: invalid syntax
```

那么我们可以使用 `raw_input()` 方法，它倒是没那么矫情（任何类型的输入它都可以接收），用户输入什么就是什么。

你可以将 `input()` 方法替换为 `raw_input()` 再来运行输入一个不加引号的字符串，看看会不会还抛类型错误。

3.1.3 引号与注释

在 Python 当中，不区分单引号 ('') 与双引号 ("")，也就是说单、双引号都可以表示一个字符串。

Python shell

```
>>> print "hello"
hello
>>> print 'world'
world
```

可以嵌套使用，但不能交叉使用。

Python shell

```
>>> print "你说: '你好'"
你说: '你好'
>>> print '我说: "今天天气不错"'
我说: "今天天气不错"
>>> print "你笑了笑'离开了"。'
SyntaxError: invalid syntax
```

再来看看注释，基本上每种语言都会分单行注释和多行注释。Python 的单注释用井号 (#) 表示。

Python shell

```
>>> #单行注释  
>>> print "hell world" #打印 hello world  
hell world
```

多行注释用三对引号表示，不分单、双引号。

xx.py

```
"""  
我们实现一个伟大的程序  
那么是  
print 一行数据 ^_~  
"""  
  
'''  
This is a  
Multi line comment  
'''
```

3.2 分支与循环

Python 的设计哲学就是简单，没那么花里胡哨的东西，分支一般就用 if..else.. 语句，循环一般就用 for 语句。

3.2.1 if 语句

来个写个无聊有 if...else... 语句：

Python shell

```
>>> if 2 > 3:  
     print 2  
else:  
    print 3
```

3

如果 2 大于 3 打印 2，否则打印 3。果然很无聊，那么下面练习一个不是那么无聊的例子，多分支语句。

xx.py

```
results = 72
```

```
if results >= 90:  
    print u'优秀'  
elif results >= 70:  
    print u'良好'  
elif results >= 60 :  
    print u'及格'  
else:  
    print u'不及格'
```

根据分数划分成四个级别“优秀”、“良好”、“及格”、“不及格”，那么72分属于哪个级别，练习一下吧。

3.2.2 for语句

循环一个字符串中的每一个字符。

xx.py

```
strings = "hello world"  
  
for l in req strings:  
print l
```

打印结果：

```
>>> ===== RESTART =====  
>>>  
h  
e  
l  
l  
o  
  
w  
o  
r  
l  
d
```

循环数字，循环数字要借助range()函数。

xx.py

```
for i in range(1,10):  
print i
```

打印结果：

```
>>> ===== RESTART =====  
>>>  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

如果我只想打印 1 到 10 之间的奇数：

xx.py

```
for i in range(1,10,2):  
print i
```

打印结果：

```
>>> ===== RESTART =====  
>>>  
1  
3  
5  
7  
9
```

`range(start, end, scan)`

`range()` 函数，`start` 表示开始位置，`end` 表示结束位置，`scan` 表示每一次循环的步长。

3.3 数组与字典

在我们的自动化脚本里除了字符串与数字外，用到最多的就是数组与字典了，所以，我们来熟悉一下这种数据存储的概念。

3.3.1 数组

数组用中括号（`[]`）表示，里面的每一个元素用逗号（`,`）隔开。

Python shell

```
>>> shuzu = [1,2,3,'a',5]
>>> print shuzu
[1, 2, 3, 'a', 5]
>>> print shuzu[0]
1
>>> print shuzu[4]
5
```

需要注意的是，数组下标是从 0 开始的。

3.3.2 字典

字典以花括号 ({}) 表示，里面的元素是成对出现的，一个 key 对应一个 value；一对元素用冒号 (:) 分割；不同对元素用逗号 (,) 分开。

Python shell

```
>>> zidian = {"username":"password", 'man':'woman', 1:2}
>>> zidian.keys()
['username', 1, 'man']
>>> zidian.values()
['password', 2, 'woman']
>>> zidian.items()
[('username', 'password'), (1, 2), ('man', 'woman')]
```

字典里的每一对元素准确的来说是键值对，一个键 (key) 对应一个值 (value)。keys() 函数可以输出所有键的值；values() 函数可以输出所有值的值；items() 函数输出一对键值的值。

3.4 函数与类、方法

利用前面讲的知识只能搭建一个鸡窝，那么要想建造一个庞大而且结构复杂的大厦，那么就离不开函数、类和方法的使用。

3.4.1 函数

在 Python 当中通过 def 关键字来定义函数，下面来定义一个函数。

Python shell

```
>>> def add(a,b):
    print a+b

>>> add(3,5)
8
```

创建一个 add()，这个函数接收两个参数 a, b，通过 print 打印 a+b 的结果。调用 add() 函数，并且传两个参数 3, 5 给 add() 函数。

更多的情况下，add() 函数不会直接打印结果，而是将处理结果通过 return 关键字返回。

Python shell

```
>>> def add(a,b):
    return a+b

>>> add(3,5)
8
```

有时候我们在调用 add() 函数的时候不想传参，那么可以为 add() 函数设置默认参数。

Python shell

```
>>> def add(a=1,b=2):
    return a+b

>>> add()
3
>>> add(3,5)
8
```

如果调用时不传参，那么 add() 函数就使用默认参数进行计算。如果传参则计算参数的值。

3.4.2 类与方法

在面向对象编程的世界里一切皆为对象，那么抽象的一组对象就是类。例如，汽车就是一个类，张三家的奇瑞汽车就是一个具体的对象。在 Python 中用 class 关键字来创建类。

xx.py

```
class A():
    def add(self,a,b):
        return a+b

count = A()
print count.add(3,5)
```

打印结果：

```
>>> ===== RESTART =====
>>>
8
```

创建了一个 A() 类，在类下面创建了一个 add() 方法。方法的创建同样使用关键字 def，唯一不同的是方法必须有一个且必须是第一个默认参数 self，但是这个参数不用传值。

xx.py

```
class A:
    def add(self,a,b):
        return a+b

class B(A):
    def sub(self,a,b):
        return a-b

count = B()
print count.add(4,5)
```

打印结果：

```
>>> ===== RESTART =====
>>>
8
```

用 count 变量来等于 B 类，因为 B 类继承了 A 类，所以 B 类自然也拥有了 add() 方法，所以 count 可以调用 add() 方法。

当然，学到这里你会产生很多疑问，为什么要写类里面包含方法，用函数不是更简单么？还有那个 self 非要写出来，但看上去又没什么用。当然，类有很多特性，比如，上面所介绍的类的继承。

3.5 模组

更通俗的讲叫包或模块，或统称为“头文件”，前面的练习中我们并没用到模块，但这也只是在练习的时候，在实际的开发中我们不可能不用到系统所提供的方法，或第三方模块。如果，我们想实现与时间有关功能那么就可以调用系统的 time 模块，如果我们想实现与文件和文件夹有关的操作，那么就要用到 os 模块。再比如我们通过极 selenium 实现的 web 自动化测试，那么 selenium 对于 Python 来说就是一个模块。

3.5.1 引用模块

在 Python 语言中通过 from... import... 的方式引用模块，下面引用 time 模块。

xx.py

```
import time

print time.ctime()
```

打印结果：

```
>>> ===== RESTART =====
>>>
Tue Dec 09 22:35:57 2014
```

在 time 模块下面有一个 ctime() 方法获得当前时间，通过 print 将当前时间打印出来。当然，如果我确定了只会用到 time 下面的 ctime() 方法，那么可以这样：

xx.py

```
from time import ctime

print ctime()
```

打印结果：

```
>>> ===== RESTART =====
>>>
Tue Dec 09 22:36:38 2014
```

那么现在在使用的时候就不用告诉 Python ctime() 方法是 time 模块所提供的了。但是有时候我们还可能会用到 time 模块下面的 sleep() 休眠方法；当然，我们还可以把 sleep() 方法也导入进来。或许还会用到其它方法呢，那么为什么不一次性把 time 模块下面的所有方法都引入进来呢。

xx.py

```
#coding=utf-8
from time import *

print ctime()
print "休息一两秒"
sleep(2)
print ctime()
```

打印结果：

```
>>> ===== RESTART =====
>>>
Tue Dec 09 22:47:35 2014
休息一两秒
Tue Dec 09 22:47:37 2014
```

星号 “*” 用于表示模块下面的所有方法。那么你一定很好奇， time 到底在哪儿？为什么 import 进来就可以用了。这是 Python 语言所提供的核心方法，而且已经经过了编译，所以我们无法看到 ctime 到底是如何实现的可以取到系统的当前时间，但是我们可以看到 Python 所安装的第三方模块。比如我们做自动化所用到的 selenium 模块。

Python 所安装的模块存放在 C:\Python27\Lib\site-packages\目录下面，如果你已经通过第二章的学习安装了 selenium，那么你一定会在这个目录下找到 selenium 目录。

3.5.2 模块调用

下一个你所关心的问题，既然可调用系统模块，那么我可不可以自己创建一个模块，然后通过另一个程序调用，当然是可以的，有谁见过一个软件项目是只有一个文件，它们一定是有结构地分散在不同的目录和文件中。

下面创建一个目录（project），在目录下创建一个文件（pub.py），在文件中创建一个函数。

project\pub.py

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

在相同的目录下再创建一个文件（count.py）

project\count.py

```
import pub
print pub.add(4,5)
```

打印结果：

```
>>> ===== RESTART =====
>>>
9
```

这样就实现了跨文件的函数调用。

知识延伸：

如果你细心一定会发现在 project 目录下多了一个 pub.pyc 文件，那么这个.pyc 是什么？

pyc 是一种二进制文件，是由 py 文件经过编译后，生成的文件，是一种 byte code，py 文件变成 pyc 文件后，加载的速度有所提高，而且 pyc 是一种跨平台的字节码，是由 Python 的虚拟机来执行的。

3.5.3 跨目录模块调用

现在的情况是 pub.py 和 count.py 两个文件同在一个目录下面，可以非常非常方便的调用。那么如果被调用的文件与调用文件不在同一目录下面呢？调用文件目录如下：

```
---project/model/pub.py
---project/count.py
```

注意删除刚才调用时所生成的 pub.pyc 文件。再来执行 count.py 文件

project\count.py

```
import pub

print pub.add(4,5)
```

打印结果：

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

```
Traceback (most recent call last):
  File "F:\project\count.py", line 1, in <module>
    import pub
ImportError: No module named pub
```

现在 Python 告诉我们，它找不到 pub 模块。那么 Python 是如何找模块的呢？

知识延伸：

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。

那么现在，我们就知道如何让 count.py 找到 pub.py 文件了，可以将... project/model/ 目录添加到系统环境变量的 Path 下面，当然，这样做的确是不够灵活。更灵活的方式是调用 Python 的 sys 模块来实现。

project\count.py

```
import sys
sys.path.append('\model')
from model import pub

print pub.add(4,5)
```

调用 sys 模块，把 model 目录通过 append() 方法追加到系统环境变量 Path 下面。append() 是一个非常有用的方法，它通常用于向一个数据或集合尾部添加新的数据。注意 “\model” 是一个相对路径，如果 pub.py 在别的目录下面，那么就要用绝对路径了，如：append('D:\\project\\model')。

现在来运行程序依然会告诉我们找不到 model，我们还需要在 model 目录下面创建一个 `__init__.py` 文件，内容可以为空。这个文件告诉 Python model 是可以被调用的一个模块。好了，现在可以正常运行 count.py 程序了。

那么我们可以在 `__init__.py` 目录下面放点什么呢？例如在 model 目录下有两三个文件（a.py、b.py、c.py）打开 `__init__.py` 文件：

`__init__.py`

```
import a,b
```

project\count.py

```
import sys
sys.path.append('model')
from model import *

print pub.add(4,5)
```

现在我们用星号 (*) 表示导入 model 下面所有文件，这个所有具体包含什么由 `__init__.py` 决定，因为只 import 了 a 和 b 了。所以，在运行程序时会有下面的提示：

Python Shell

```
>>> ===== RESTART =====
>>>
Traceback (most recent call last):
  File "F:\project\count.py", line 5, in <module>
    print pub.add(4,5)
NameError: name 'pub' is not defined
```

3.6 异常

Python 用异常对象（exception object）来表示异常情况。遇到错误后，会引发异常。如果异常对象并未被处理或捕捉，程序就会用所谓的 回溯（Traceback，一种错误信息）终止执行。

在实际的脚本开发中，有时程序并不会像我们设计它时那样工作，它也有“生病”的时候，那么我们可以通过异常处理机制，有预见性地获得这些病症，并开出药方。比如，对一个正常人，大冬天的洗冷水澡，那么就有可能感冒，我们可以事先在洗冷水澡时准备好感冒药，假如真感冒了，就立刻吃药。

3.6.1 认识异常

下面来看看程序在执行时所抛的异常。

Python Shell

```
>>>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'
```

我们通过 open() 方法打开一个 abc.txt 的文件，以读“r”的方式打开。然后 Python 抛出一个 IOError 类型的异常，它告诉我们：No such file or directory: abc.txt（没有 abc.txt 这样的文件或目录），当然找不到啦，因为我们根本就没创建过这个文件嘛。

既然知道执行 open() 一个不存在的文件时会抛 IOError 异常，那么我们就可以通过 Python 所提供的 try...except... 语句来接收这个异常。

abnormal.py

```
try:
    open("abc.txt",'r')
except IOError:
    print "异常了!"
```

打印结果：

```
>>> ===== RESTART =====
>>>
异常了！
```

再来运行程序，因为我们用 except 接收了这个 IOError 错误。所以“异常了！”的信息会被打印出来。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？

abnormal.py

```
try:
    print aa
except IOError:
    print "异常了!"
```

打印结果：

```
>>> ===== RESTART =====
>>>
Traceback (most recent call last):
  File "F:\project\count.py", line 12, in <module>
    print aa
NameError: name 'aa' is not defined
```

不是已经通过 except 去接收异常了么？为什么错误又出现了，如果你细心查看错误信息会发现这一次抛的是个 NameError 类型的错误，而我们的 except IOError 只能接收到 IO 类型的错误。这一次小明

得的是肚子痛，我们依然拿感冒药去吃，当然是医不好病的。

那么我们只需要换一个接收异常的类型就可以了。

abnormal.py

```
try:
    print aa
except NameError:
    print "这是一个 name 异常!"
```

打印结果：

```
>>> ===== RESTART =====
>>>
这是一个 name 异常!
```

那么问题来了，我们一个操作怎么会抛出什么类型的错误呢？

知识延伸：

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。

在 Python 中所有的异常类都继承 Exception，所以我们可以使用它来接收所有的异常。

abnormal.py

```
try:
    open("abc.txt", 'r')
    print aa
except Exception:
    print "异常了!"
```

打印结果：

```
>>> ===== RESTART =====
>>>
异常了!
```

从 Python2.5 版本之后，所有的异常类有了新的基类 BaseException，Exception 同样也继承 BaseException，所以我们也就可以使用 BaseException 来接收所有的异常。

abnormal.py

```
try:
```

```

open("abc.txt",'r')
    print aa
except BaseException:
    print "异常了!"

```

打印结果：

```

>>> ===== RESTART =====
>>>
异常了!

```

那么对于上面例子，如果我在执行程序的当前目录下创建了 abc.txt，那么就不会报 IOError，如果定义了 aa 变量，那么就不会报 NameError 错误。可问题是自定义的 print 异常信息，并不能准确的告诉是哪一行代码所引起的异常。那么何不让 Python 直接告诉我们呢。

abnormal.py

```

try:
    open("abc.txt",'r')
        print aa
except BaseException,msg:
    print msg

```

打印结果：

```

>>> ===== RESTART =====
>>>
[Errno 2] No such file or directory: 'abc.txt'

```

我们在 BaseException 后面定义 msg 变量用于接收异常信息，通过 print 将其打印出来。

下面来认识一下 Python 中常见的异常：

异常	描述
BaseException	新的所有异常类的基类
Exception	所有异常类的基类，但继承 BaseException 类
AssertionError	assert 语句失败
AttributeError	试图访问一个对象没有属性
IOError	输入输出异常，试图打一个不存的文件（包括其它情况）时引起
NameError	使用一个还未赋值对象的变量
IndexError	在使用序列中不存在的所引进引发
IndentationError	语法错误，代码没有正确的对齐
KeyboardInterrupt	Ctrl+C 被按下，程序被强行终止
TypeError	传入的对象类型与要求不符
SyntaxError	Python 代码逻辑语法出错，不能执行

3.6.2 更多异常用法

通过上面的学习我们会使用异常了，下面来学习异常的更多用法。try...except 与 else 配合使用：

abnormal.py

```
try:
    aa = '异常测试'
    print aa
except Exception,msg:
    print msg
else:
    print '没有异常！'
```

打印结果：

```
>>> ===== RESTART =====
>>>
异常测试
没有异常！
```

这里我们对 aa 变量进行了赋值，所以没有异常，那么将会执行 else 语句后面的内容。那么 else 语句只有在没有异常的情况下才会被执行，但是有些情况下不管是否出现异常这些操作都能被执行，比如文件的关闭，锁的释放，把数据库连接返还给连接池等操作。我们可以使用 Try...finally... 语句来完成这样有需求。

首先我们来创建一个 poem.txt 文件。

pome.txt

```
abc
efg
hijk
lmn
opq
```

下面我们通过一个小程序来读取文件中的内容。

abnormal.py

```
#coding=utf-8
import time

files = file("poem.txt",'r')
strs = files.readlines()

try:
    for l in strs:
        print l
```

```

        time.sleep(1)
finally:
    files.close()
    print 'Cleaning up ...closed the file'

```

这个程序比我们之前练习的要复杂一些，首先导入有了 time 包，我们主要用到 time 下面的 sleep()方法，使程序在运行到某个位置时做休眠。通过 file 方法打开 poem.txt 文件，以读“r”的方式打开。通过调用 readlines()方法逐行的来读取文件中的数据。

在 try 的语句块中，通过一个 for 循环来逐行的打印 poem.txt 文件中的数据，每循环一次休眠一下。在 finally 语句块中执行文件的 close()操作，为了表示 close 操作会被执行，我们随后打印一行提示“Cleaning up ...closed the file”。下面来运行程序。

abnormal.py

打印结果：

```

>>> ===== RESTART =====
>>>
abc

efg

hijk

lmn

opq
Cleaning up ...closed the file

```

打印结果：

```

>>> ===== RESTART =====
>>>
abc

efg

Cleaning up ...closed the file

Traceback (most recent call last):
  File "F:\project\count.py", line 8, in <module>
    time.sleep(1)
KeyboardInterrupt

```

我们总共运行了两次程序，第一次程序被正常的执行并打印“Cleaning up ...closed the file”，表示 finally 语句块中的程序被执行。第一次在程序运行的过程中通过键盘 Ctrl+C 意外的终止程序的执行。那么程序将抛出 KeyboardInterrupt 错误，那么同样打印出了“Cleaning up ...closed the file”信息，表示 finally 语句块

依然被执行。

3.6.3 抛出异常

对于 print 方法来说只能打印错误信息，Python 中提供 raise 方法来抛出一个异常，下面例子演示 raise 的用法。

abnormal.py

```
filename = raw_input('please input file name:')

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

运行结果：

```
>>> ===== RESTART =====
>>>
please input file name:hello

Traceback (most recent call last):
  File "F:\project\count.py", line 5, in <module>
    raise IOError('input file name error !')
NameError: input file name error !
```

运行程序，要求用户输入文件名，通过用户输入的为“hello”，那么将抛出一个 NameError。其实用户输入什么样的信息与 NameError 之间没有什么关系。但我们可以使用 raise 自定义一些异常信息，这看上去比 print 更专业。需要注意的是 raise 只能使用 Python 中所提供的异常类，如果你自定义了一个 abcError 可不起作用。

第 4 章 WebDriver API

从本章正式开始学习 Selenium，对于本章标题的命名比较纠结，要想起一个确切的符合主题的名字并不太容易。WebDriver 属于 Selenium 体系中设计出来操作浏览器的一套 API，它支持多种语言，这个我们前面已经说过。那么站在编程语言的角度，Selenium WebDriver 只是 Python 的一个第三方框架，和 Django web 开发框架属于一个性质，只是 Django 只在 Python 语言中存在，其它语言也有用于 web 开发的框架，但不叫 Django。那么 Selenium WebDriver 对于 Python 来说是一个用于实现 web 自动化的第三方框架。好吧！你明白我在说什么。

网上有一本文档中叫《Selenium Python Bindings》，也许这个名字更能贴切的说明我们这一章所要介绍的内容。我们其实要讲的内容是，在 Python 语言中，如何通过 Selenium Webdriver 所提供的各种方法来实现 web 自动化测试。

4.1 从定位元素开始

在本章开始之前，我们先来看一张页面。



图 4.1 web 页面

这其实就是百度的首页，在这张页面上有输入框、按钮和文字链接，当然还有图片，页面的底部还有一行文字，左侧还有一个下拉框。自动化要做的就是使用鼠标和键盘来操作这些元素，或点击，或输入，或右击，甚至是鼠标拖动等操作。

那么我们要想操作这些元素的前提是需要找到它们。那么如何找到他们呢？自动化工具可不像我们一

样可以通过肉眼来分辨页面上的元素，并且知道是它们是做什么用的。那么我们来看看这些元素的真实面目。

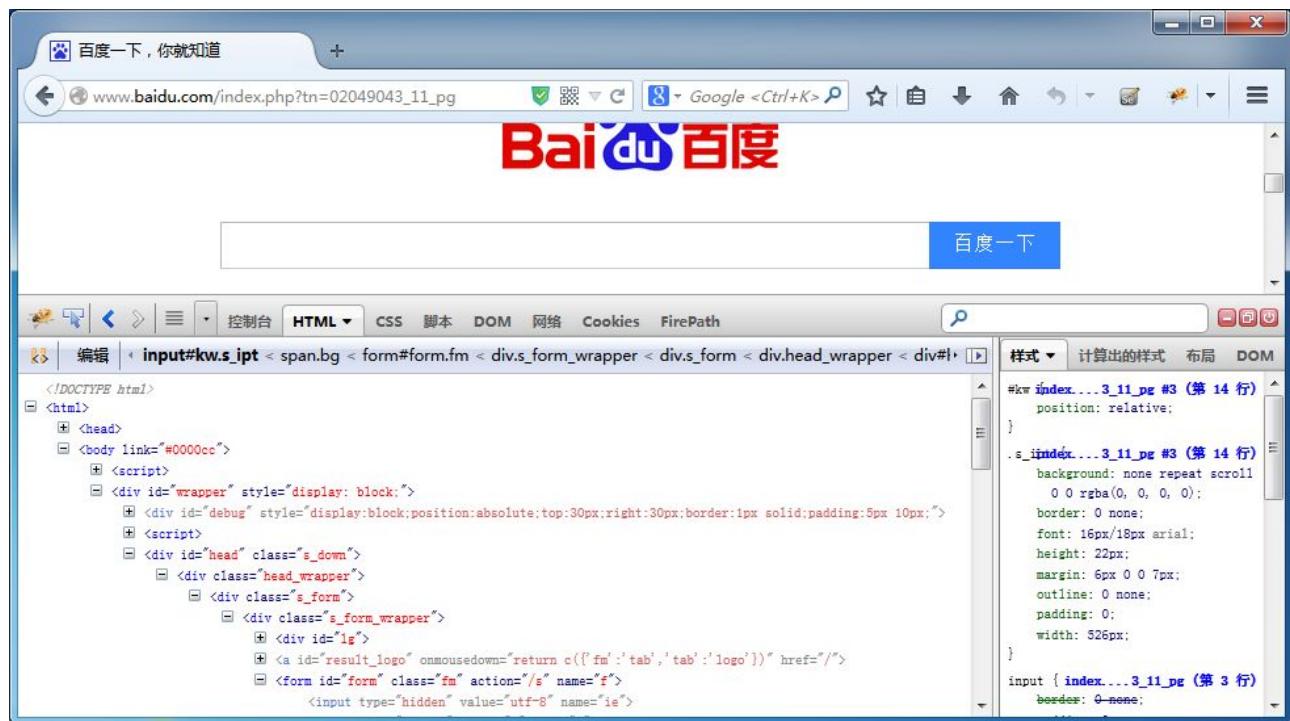


图 4.2 通过 FireBug 查看页面元素

通过前端工具，我们可以看到页面上的元素都是由一行一行的代码组成。它们之间有层级的组织起来，每个元素有不同的标签名和属性值。那么在 Selenium 当中就是通过这信息来找到不同的元素的。

webdriver 提供了八种元素定位方法：

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

在 Python 语言中对应的定位方法如下：

```
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()
```

下面我们就逐一的来看这些定位方法的使用。在此之前，我们拷取百度首页的前端代码，以定位页面

上的元素为例进行讲解。

baidu.html

```
<html>
  <head>
  <body>
    <script>
      <div id="wrapper" style="display: block;">
        <div id="debug" style="display:block;position:..">
          <script>
            <div id="head" class="s_down">
              <div class="head_wrapper">
                <div class="s_form">
                  <div class="s_form_wrapper">
                    <div id="lg">
                      <a id="result_logo" onmousedown="return .." href="/">
                      <form id="form" class="fm" action="/s" name="f">
                        <input type="hidden" value="utf-8" name="ie">
                        <input type="hidden" value="8" name="f">
                        <input type="hidden" value="1" name="rsv_bp">
                        <input type="hidden" value="1" name="rsv_idx">
                        <input type="hidden" value="" name="ch">
                        <input type="hidden" value="02.." name="tn">
                        <input type="hidden" value="" name="bar">
                        <span class="bg s_ipt_wr">
                          <input id="kw" class="s_ipt" autocomplete="off"
                                 maxlength="100" value="" name="wd">
                        </span>
                        <span class="bg s_btn_wr">
                          <input id="su" class="bg s_btn" type="submit"
                                 value="百度一下">
                        </span>
                    ....
                  </div>
                </div>
              </div>
            </div>
          </div>
        </div>
      </div>
    </body>
</html>hello
```

注意这段代码并非百度首页的页面源代码，而是通过前端工具查看所得到页面代码与结构。那么这样的HTML结构有如下特征。

(1) 它们由标签对组成：

```
<html></html>
<body></body>
<div></div>
```

```
<form></form>
```

那么 html、div 就是标签的标签名。

(2) 标签各种属性属性：

```
<div id="head" class="s_down">  
<from class="well">  
<input id="kw" name="wd" class="s_ipt">
```

就像人一样也会有各种属性，身份证号（id）、姓名（name）、职业（class）等。

(3) 标签对之间可以有文本数据。

```
<a>新闻</a>  
<a>hao123</a>  
<a>地图</a>
```

(4) 标签有由层级关系

```
<html>  
  <body>  
    </body>  
  </html>  
  
<div>  
  <from>  
    <input />  
  </from>  
  
<div>
```

对于上面结构，如果把 input 看作是子标签，那么 form 就是它的父标签。

理解了上面这些特性是学习定位方法的基础。我们以百度输入框和百度搜索按钮为例来学习不同的定位方法，两个元素的代码如下。

```
.....  
<input id="kw" class="s_ipt" autocomplete="off" maxlength="100" value=""  
name="wd">  
.....  
<input id="su" class="bg s_btn" type="submit" value="百度一下">  
.....
```

4.1.1 id 定位

name 如果把页面上看元素看作一个人的话，如果我们想找一个人如何去找，那么这个人一定有其别于其它人的“属性”，比如他的身份证号一定和别人不一样，他的名字和别人不一样。那么我们就可以通过身份证号和名字来找到一个人。那么 id 就可以看做是一个人的身份号，当然这个 id 并不像我们现实中的身份证号有那么强的唯一性，如果在一个页面上发现有两个元素的 id="kw"也是不足为奇的，这个取决于前端代码的规范程度。

对百度首页上的输入框与百度搜索按钮来说，定位方法如下：

```
find_element_by_id("kw")
```

```
find_element_by_id("su")
```

find_element_by_id()方法用于元素中 id 属性的定位。

4.1.2 name 定位

name 的定位与 id 类似，每一个人都会有名字，那么 name 就可作是一个元素的名字。通过 name 定位输入框：

```
find_element_by_name("wd")
```

find_element_by_name()方法用于元素中 name 属性的定位，百度搜索按钮并没有提供 name 属性，那么我们就不能通过 name 去定位百度搜索按钮。

4.1.3 class 定位

class 也是不少元素会有的一个属性，它的定位和 name 以及 id 类似，下面通过 class 去定位百度输入框和百度搜索按钮：

```
find_element_by_class_name("s_ipt")
```

```
find_element_by_class_name("bg s_btn")
```

find_element_by_class_name()方法用于元素中 class 属性的定位。

4.1.4 tag 定位

tag 定位取的是一个元素的标签名，通过标签名去定位单个元素的唯一性最底，因为在同一个页面中有

太多的元素标签为<div>和<input>了，所以很难通过标签名去区分不同的元素。

通过标签名定位百度首页上的输入框与百度搜索按钮：

```
find_element_by_tag_name("input")
find_element_by_tag_name("input")
```

`find_element_by_tag_name()`方法通过元素的 tag name 来定位元素。通过上面的例子，我们并不能区别不同的元素，因为在一个页面上标签名相同很难以避免。

4.1.5 link 定位

link 定位与前面介绍的几种定位方法有所不同，它专门用来定位本链接。百度输入框上面的几个文本链接的代码如下：

```
<a class="mnav" name="tj_trnews" href="http://news.baidu.com">新闻</a>
<a class="mnav" name="tj_trhao123" href="http://www.hao123.com">hao123</a>
<a class="mnav" name="tj_trmap" href="http://map.baidu.com">地图</a>
<a class="mnav" name="tj_trvideo" href="http://v.baidu.com">视频</a>
<a class="mnav" name="tj_trtieba" href="http://tieba.baidu.com">贴吧>
```

通过查看上面的代码，我们发现通过 name 属性定位是个不错的选择。不过我们这里为了要学习 link 定位，通过 link 定位实现如下：

```
find_element_by_link_text("新闻")
find_element_by_link_text("hao123")
find_element_by_link_text("地图")
find_element_by_link_text("视频")
find_element_by_link_text("贴吧")
```

`find_element_by_link_text()`方法通过元素标签对之间的文本信息来定位元素。不过，需要强调的是 Python 对于中文的支持并不好，如查 Python 在执行中文的地方出现在乱码，可以在中文件字符串的前面加个小“u”可以有效的避免乱码的问题，加 u 的作用是把中文字符串转换成 unicode 编码，如：

```
find_element_by_link_text(u"新闻")
```

4.1.6 partial link 定位

partial link 定位是对 link 定位的一个种补充，有些文本连接会比较长，这个时候我们可以取文本链接的

有一部分定位，只要这一部分信息可以唯一的标识这个链接。

```
<a class="mnav" name="tj_lang" href="#">一个很长很长的文本链接</a>
```

通过 partial link 定位如下：

```
find_element_by_partial_link_text("一个很长的")
```

```
find_element_by_partial_link_text("文本连接")
```

`find_element_by_link_text()`方法通过元素标签对之间的部分文本信息来定位元素。

前面所介绍的几种定位方法相对来说比较简单，我们理想状态下在一个页面当中每一个元素都会有一个唯一 `id` 和 `name` 属性值，我们通过它的属性值来找到他们，但在实际的项目中并非想象的这般美好。有时候一个元素并没有 `id` 或 `name` 属性，或者会有多个元素的 `id` 和 `name` 属性值是一样的，又或者每一次刷新页面，`id` 的值都会随机变化。那么在这种情况下我们如何来定位元素呢？

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，但他们的灵活的定位能力远比上面的几种方式要强大得多。

4.1.7 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。

绝对路径定位：

`XPath` 有多种定位策略，最简单和直观的就是写元素的绝对路径。如果仍然把一个元素看做一个人的话，那么现在有一个人，他没有任何属性特征，那么这个人一定会存在于某个地理位置，如：xx 省 xx 市 xx 区 xx 路 xx 号。那么对于一个元素在一个页面当中也会有这样的一个绝对地址。

参考 `baidu.html` 前端工具所展示的代码，我们可以用下面的方式来找到百度输入框和搜索按钮。

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

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

`find_element_by_xpath()`方法用于 `XPath` 语言定位元素。`XPath` 的绝对路径主要用标签名的层级关系来定位元素的绝对路径。最外层为 `html` 语言，`body` 文本内，一级一级往下查找，如果一个层级下有多个相同的标签名，那么就按上下顺序确定是第几个，`div[2]`表示第二个 `div` 标签。

利用元素属性定位：

除了使用绝对路径的以外，`XPath` 也可以使用使素的属性值来定位。同样以百度输入框和搜索按钮为例了：

```
find_element_by_xpath("//input[@id='kw']")
```

```
find_element_by_xpath("//input[@id='su'])")
```

//表示当前页面某个目录下，input 表示定位元素的标签名，[@id='kw'] 表示这个元素的 id 属性值等于 kw。下面通过 name 和 class 属性值来定位。

```
find_element_by_xpath("//input[@id='wd'])")
```

```
find_element_by_xpath("//input[@class='s_ipt'])")
```

```
find_element_by_xpath("//*[@class='bg s_btn'])")
```

如果不想指定标签名也可以用星号 (*) 代替。当然，使用 XPath 不仅仅只局限在 id、name 和 class 这三个属性值，元素的任意属性值都可以使用，只要它能唯一的标识一个元素。

```
find_element_by_xpath("//input[@maxlength='100'])")
```

```
find_element_by_xpath("//input[@autocomplete='off'])")
```

```
find_element_by_xpath("//input[@type='submit'])")
```

层级与属性结合：

如果一个元素本身并没有可以唯一标识这个元素的属性值，我们可以找其上一级元素，如果它的上级有可以唯一标识属性的值，也可以拿来使用。参考 baidu.html 文本。

```
.....  
<form id="form" class="fm" action="/s" name="f">  
    <input type="hidden" value="utf-8" name="ie">  
    <input type="hidden" value="8" name="f">  
    <input type="hidden" value="1" name="rsv_bp">  
    <input type="hidden" value="1" name="rsv_idx">  
    <input type="hidden" value="" name="ch">  
    <input type="hidden" value="02.." name="tn">  
    <input type="hidden" value="" name="bar">  
    <span class="bg s_ipt_wr">  
        <input id="kw" class="s_ipt" autocomplete="off"  
              maxlength="100" value="" name="wd">  
    </span>  
    <span class="bg s_btn_wr">  
        <input id="su" class="bg s_btn" type="submit"  
              value="百度一下">  
    </span>  
.....
```

假如百度输入框本身没有可利用的属性值，我们可以查找它的上一级属性。比如，“小明”刚出生的时候没有名字，没上户口（没身份证号），那么亲朋好友来找“小明”可以先到小明的爸爸，因为他爸爸是有很多属性特征的，找到了小明的爸爸，抱在怀里的一定就是小明了。通过 XPath 描述如下：

```
find_element_by_xpath("//span[@class='bg s_ipt_wr']/input")
```

```
find_element_by_xpath("//span[@class='bg s_btn_wr']/input")
```

span[@class='bg s_ipt_wr'] 通过 class 属性定位到是父元素，后面/input 也就表示父元素下面标签名为 input 的子元素。如果父元素没有可利用的属性值，那么可以继续向上查找“爷爷”元素。

```
find_element_by_xpath("//form[@id='form']/span/input")
```

```
find_element_by_xpath("//form[@id='form']/span[2]/input")
```

我们可以通过这种方法一级一级的向上打找，直到找到最外层的<html>标签，那么就是一个绝对路径的写法了。

使用逻辑运算符

如果一个属性不能唯一的区分一个元素，我们还可以使用逻辑运算符连接多个属性来区别于其它属性。

.....

```
<input id="kw" class="su" name="ie">  
<input id="kw" class="aa" name="ie">  
<input id="bb" class="su" name="ie">
```

如上面的三行元素，假如我们现在要定位第一行元素，如果使用 id 将会与第二行元素重名，如果使用 class 将会与第三行元素的重名。那么如果同时使用 id 和 class 就会唯一的标识这个元素。那么这个时候就可以通过逻辑运算符号连接。

```
find_element_by_xpath("//input[@id='kw' and @class='su']/span/input")
```

当然，我们也可以用 and 连接更多的属性来唯一的标识一个元素。

我们在本书的第一章中介绍的 Firebug 前端调试工具和 FirePath 插件可以方便的辅助 XPath 语法。打开 FireFox 浏览器的 FireBug 插件，点击插件左上角的鼠标箭头，再点击页面上需要定位的元素，在元素行上右键弹出快捷菜单，选择“复制 XPath”，将会获得当前元素的 XPath 语法，（如图4.3）。

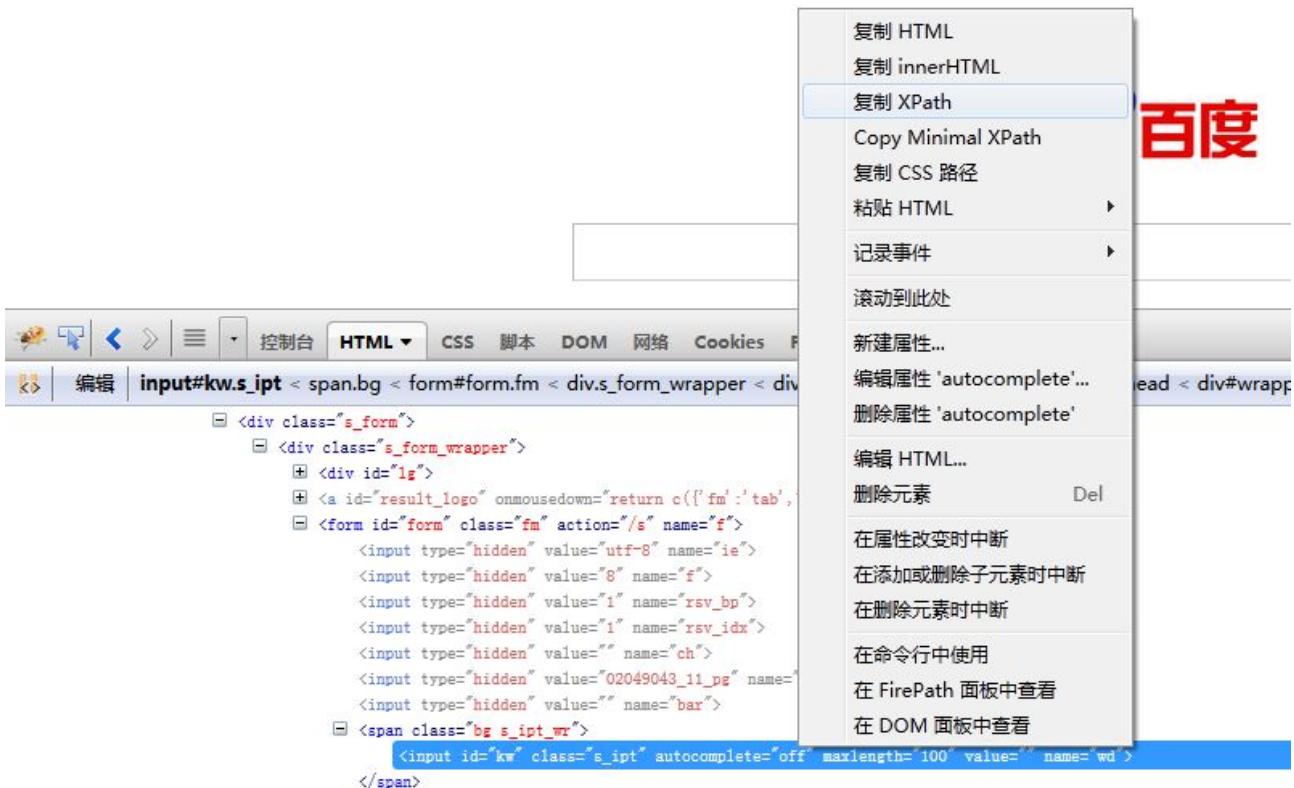


图 4.3 通过 FireBug 复制 XPath 语法

FirePath 插件的使用就更加方便和快捷了, 选中元素后, 直接在 XPath 的输入框中生成当前元素的 XPath 语法 (如图4.4)。

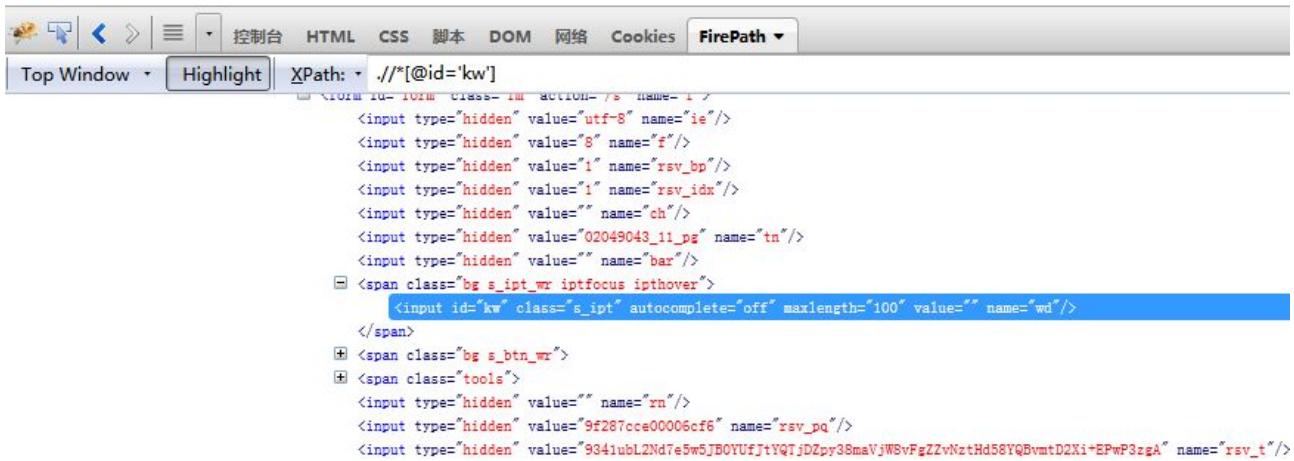


图 4.4 通过 FirePath 的生成 XPath 语法

4.1.8 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：

选择器	例子	描述
.class	.intro	class 选择器，选择 class="intro"的所有元素
#id	#firstname	id 选择器，选择所有 id="firstname"所有元素
*	*	选择所有元素
element	p	元素所有<p>元素
element > element	div > input	选择父元素为 <div> 元素的所有 <input> 元素
element + element	div + input	选择紧接在 <div> 元素之后的所有 <p> 元素。
[attribute=value]	[target=_blank]	选择 target="_blank" 的所有元素。

下面同样以百度输入框和搜索按钮为例介绍 CSS 定位的用法。

.....

```
<span class="bg s_ipt_wr">
    <input id="kw" class="s_ipt" autocomplete="off"
           maxlength="100" value="" name="wd">
</span>
<span class="bg s_btn_wr">
    <input id="su" class="bg s_btn" type="submit"
           value="百度一下">
</span>
```

.....

通过 **class** 属性定位：

```
find_element_by_css_selector(".s_ipt")
```

```
find_element_by_css_selector(".bg s_btn")
```

find_element_by_css_selector()方法用于 CSS 语言定位元素，点号（.）表示通过 class 属性来定位元素。

通过 **id** 属性定位：

```
find_element_by_css_selector("#kw")
```

```
find_element_by_css_selector("#su")
```

井号（#）表示通过 id 属性来定位元素。

通过标签名定位：

```
find_element_by_css_selector("input")
```

在 CSS 语言中用标签名定位元素不需要任何符号标识，直接使用标签名即可，但我们前面已经了解到标签名重复的概率非常大，所以通过这种方式很难唯一的标识一个元素。

通过父子关系定位：

```
find_element_by_css_selector("span>input")
```

上面的写法表示有父亲元素，它的标签名叫 span，查找它的所有标签名叫 input 的子元素。

通过属性定位：

```
find_element_by_css_selector("input[autocomplete='off']")
```

```
find_element_by_css_selector("input[maxlength='100']")
```

```
find_element_by_css_selector("input[type='submit']")
```

在 CSS 当中也可以使用元素的任意属性，只要这些属性可以唯一的标识这个元素。

组合定位：

我们当然可以把上面的定位策略组合起来使用，这样就大大加强了元素的唯一性。

```
find_element_by_css_selector("span.bg s_ipt_wr>input.s_ipt")
```

```
find_element_by_css_selector("span.bg s_btn_wr>input#su")
```

有一个父元素，它的标签名叫 span，它有一个 class 属性值叫 bg s_ipt_wr，它有一个子元素，标签名叫 input，并且这个子元素的 class 属性值叫 s_ipt。好吧！我们要找的就是具有这么多特征的一个子元素。

我们通过可以使用 Firebug 工具帮助我们生成 CSS 语法。通过 Firebug 定位元素，在元素上右键点击选择“复制 CSS”。

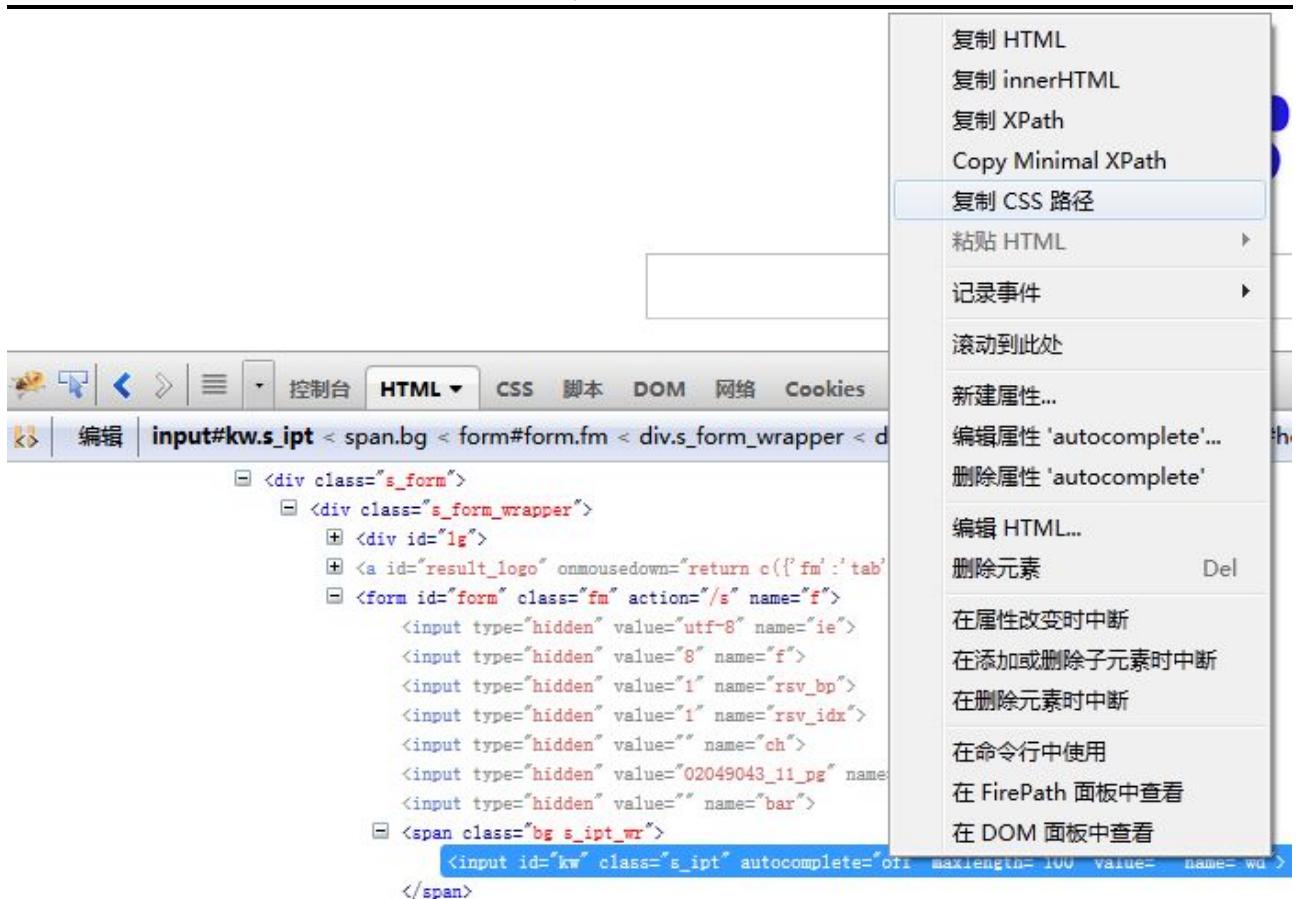


图 4.5 通过 FireBug 复制 CSS 路径

当然也可以使用 FirePath 插件来帮助生成 CSS 语法。

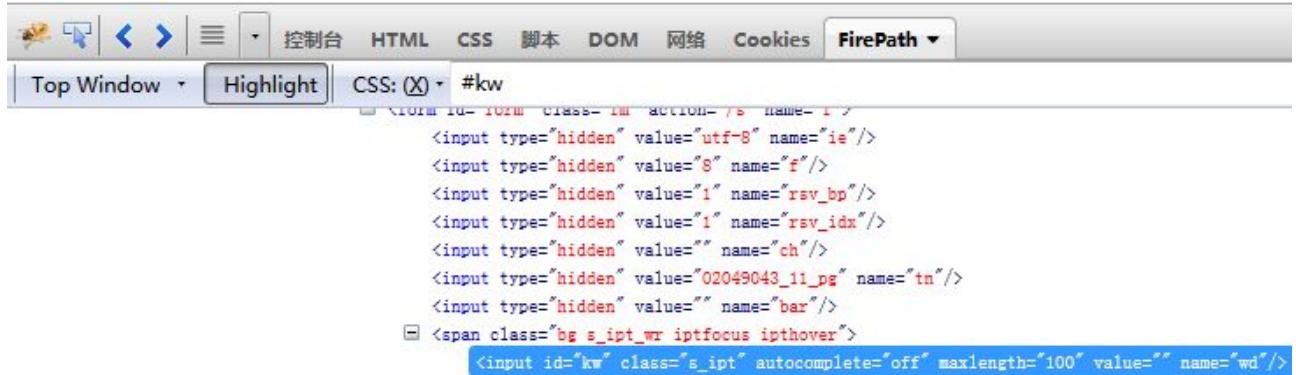


图 4.6 通过 FirePath 的生成 CSS 语法

需要说明的是 CSS 的语法远不止我们上面所介绍的内容。更多前端技术读者可以参考 w3cschool 网站。

XPath 与 CSS 的类似功能的简单对比

定位方式	XPath	CSS
标签	//div	div
By id	//div[@id='eleid']	div#eleid
By class	//div[@class='eleclass']	div#eleid

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

通过前面的学习 XPath 和 CSS 都提供了非常强大和灵活的定位方法。相比较 CSS 语法更加简洁，但真正的理解和使用学难度要更大一点。按照作都的经验这两种定位方式我们只要掌握一种就可解决大部分定位问题了，至于读者的选择就看个人喜好了。

对于 web 自动化来说，学会元素的定位那么自动化已经学习会了一半，剩下的就是 WebDriver 中所提供的各种方法的使用，我们将通过下面大量的实例来介绍这些方法的使用。

4.1.9 用 By 定位元素

有时需要使用定位方法，在具体通过哪种定位方式（id 或 name）根据实际场景而定位，By 就可以设置定位策略。

```
find_element(By.ID,"kw")
find_element(By.NAME,"wd")
find_element(By.CLASS_NAME,"s_ipt")
find_element(By.TAG_NAME,"input")
find_element(By.LINK_TEXT,u"新闻")
find_element(By.PARTIAL_LINK_TEXT,u"新")
find_element(By.XPATH,"//*[@class='bg s_btn']")
find_element(By.CSS_SELECTOR,"span.bg s_btn_wr>input#su")
```

find_element()方法只用于定位元素。它需要两个参数，第一个参数是定位方式，这个由 By 提供；另第二个参数是定位的值。在使用 By 时需要将 By 类导入。

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

4.2 控制浏览器

Selenium 主要提供的是操作页面上各种元素的方法，但它也提供了操作浏览器本身的方法，比如浏览器的大小以及浏览器后退、前进按钮等。

4.2.1 控制浏览器窗口大小

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(480x800)，然后访问移动站点，对其样式进行评估；WebDriver 提供了 `set_window_size()` 方法来设置浏览器的大小。

test.py

```
#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()
```

在 PC 端运行执行自动化测试脚本大多的情况下是希望浏览器在全屏幕模式下执行，那么可以使用 `maximize_window()`方法，其用法与 `set_window_size()` 相同，但它不需要传参。

4.2.2 控制浏览器后退、前进

在使用浏览器浏览网页的时候，浏览器提供了后退和前进按钮，可以方便的对浏览过的网页之间切换，那么 WebDriver 也提供了对应的 `back()` 和 `forward()` 方法来模拟后退和前进按钮。下面通过例子来演示这两个方法的使用。

test.py

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

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，在每操作一步都通过 print 来打印当前的 URL 地址。执行结果如下：

Python shell

```
>>> ===== 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
```

4.3 简单元素操作

在前面我已经学会了定位元素，定位只是第一步，定位之后需要对这个元素进行操作，或点击（按钮）或输入（输入框）或提交（表单），下面我们就来认识一下这些最常用的方法。

在 WebDriver 中，大多简单有趣的页面交互的方法都将通过 WebElement 接口提供，最常用的操作页面元素的方法有以下几个：

- clear() 清除文本，如果是一个文件输入框
- send_keys(*value) 在元素上模拟按键输入
- click() 单击元素

4.3.1 126 邮箱登录

下面通过126邮箱登录来演示这些方法的使用。

login126.py

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

```

driver = webdriver.Firefox()
driver.get("http://www.126.com")

driver.find_element_by_id("idInput").clear()
driver.find_element_by_id("idInput").send_keys("username")
driver.find_element_by_id("pwdInput").clear()
driver.find_element_by_id("pwdInput").send_keys("password")
driver.find_element_by_id("loginBtn").click()

driver.quit()

```

`clear()`方法用于清除文本输入框中的内容，例如登录框内一般默认会有“账号”“密码”等提示信息用于引导用户输入正确的数据，如果直接向输入框中输入数据，可能会与输入框中的提示信息拼接，本来用户输入为“username”，结果与提示信息拼接为“帐号 username”，从而造成输入信息的错误；这个时候可以先使用 `clear()`方法清除输入框内的提示信息再进行输入。

`send_keys()`方法模拟键盘输入向输入框里输入内容。如上面的例子中通过这个方法向用户名和密码框中输入用户名和密码。

`click()`方法可以用来单击一个按钮，前提是它是可以被点击元素，它与 `send_keys()`方法是 web 页面操作中最常用到的两个方法。其实 `click()`方法不仅仅用于点击一个按钮，还可以单击任何可以点击文字/图片连接、复选框、单选框、甚至是下拉框等。

4.3.2 WebElement 接口常用方法

通常所有有趣的要与页面交互的方法都是由 `WebElement` 接口提供。包括前本章第一小节所讲到的定位方法和上一节中的3个方法都由其提供。除此之外，`WebElement` 还提供了另外一些非常有用的方法。下面我们就来学习这些方法的用处。

`submit()`

`submit()`方法用于提交表单，这里特别用于没提交按钮的情况，例如搜索框输入关键字之后的“回车”操作，那么就可以通过 `submit()` 来提交搜索框的内容。

youdao.py

```

#coding=utf-8
from selenium import webdriver

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

driver.find_element_by_id('query').send_keys('hello')
#提交输入框的内容

```

```
driver.find_element_by_id('query').submit()

driver.quit()
```

通过上面的例子，我们通过定位有道搜索框并通过 submit() 提交搜索框的内容，同样达到点击“搜索”按钮的效果。有些时候 submit() 可以与 click() 方法互换来使用，submit() 同样可以提交一个按钮。

- size 返回元素的尺寸。
- text 获取元素的文本。
- get_attribute(name) 获得属性值。
- is_displayed() 设置该元素是否用户可见。

baidu.py

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

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

#获得输入框的尺寸
size=driver.find_element_by_id('kw').size
print size

#返回百度页面底部备案信息
text=driver.find_element_by_id("cp").text
print text

#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

#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

driver.quit()
```

运行结果

```
>>> ===== RESTART =====
>>>
{'width': 526, 'height': 22}
©2014 Baidu 使用百度前必读 京 ICP 证 030173 号
True
```

执行上面的程序并获得执行结果：size 用于获取百度输入框的宽、高。text 用于获得百度底部的备案信息。get_attribute() 用于获百度输入的 type 属性的值。is_displayed() 用于返回一个元素是否可见，如果可见返回 True，否则返回 False。

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

4.4 鼠标事件

通过前面例子了解到可以使用 click() 来模拟鼠标的单击操作，在现在的 web 产品中，随着前端技术的发展，页面越来越华丽，鼠标的操作也不单单只有单击，现在页面中随处可以看到需要右击、双击、鼠标悬停、甚至是鼠标拖动等操作的功能设计。在 WebDriver 中这些关于鼠标操作的方法由 ActionChains 类提供。

ActionChains 类提供的鼠标操作的常用方法：

- perform() 执行所有 ActionChains 中存储的行为
- context_click() 右击
- double_click() 双击
- drag_and_drop() 拖动
- move_to_element() 鼠标悬停

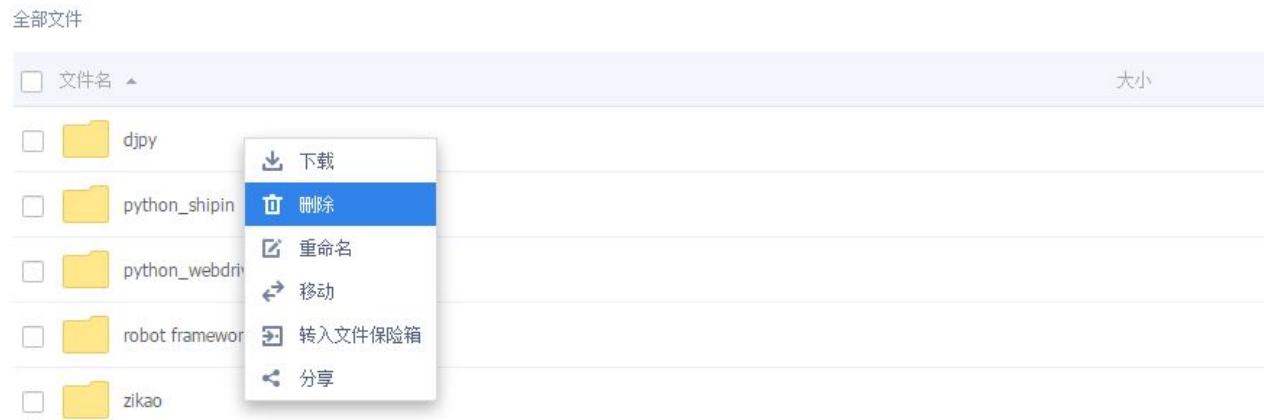


图 4.7 网盘右键快捷菜单

如图4.7中，360网盘对于操作单个文件或文件夹所提供的快捷菜单，对于这样的操作就可以使用 context_click() 方法来模拟右键操作。

鼠标右击操作

对于 ActionChains 类下所提供的鼠标方法与前面学过的 click() 方法有所不同，那么简单 context_click()

右键点击一个元素。

xx.py

```
.....  
from selenium import webdriver  
#引入ActionChains 类  
from selenium.webdriver.common.action_chains import ActionChains  
  
driver = webdriver.Firefox()  
driver.get("http://yunpan.360.cn")  
....  
  
#定位到要右击的元素  
right_click = driver.find_element_by_id("xx")  
#对定位到的元素执行鼠标右键操作  
ActionChains(driver).context_click(right_click).perform()  
....
```

```
from selenium.webdriver import ActionChains
```

对于 ActionChains 类下面的方法，在使用之前需要先将模块导入。

ActionChains(driver)

调用 ActionChains()方法，在使用将浏览器驱动 driver 作为参数传入。

context_click(right_click)

context_click()方法用于模拟鼠标右键事件，在调用时需要传入右键的元素。

perform()

执行所有 ActionChains 中存储的行为，可以理解成是对整个操作事件的提交动作。

鼠标悬停

鼠标悬停弹出下拉菜单也是一个非常见的一个功能设计，如图4.8。

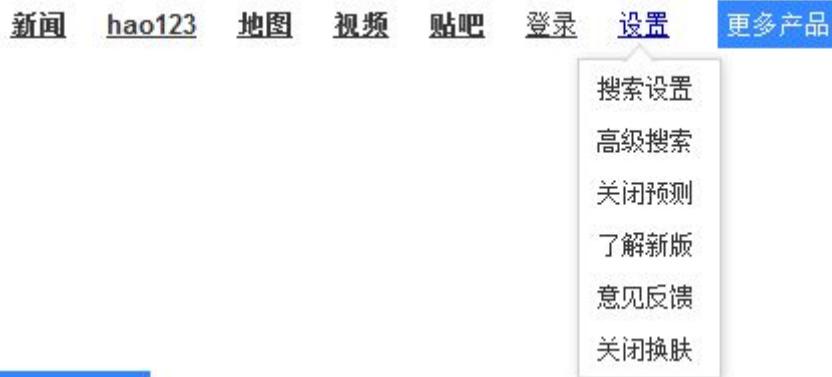


图 4.8 鼠标悬停菜单

`move_to_element()`方法可以模拟鼠标悬停的动作，其用法与 `context_click()`相同。

xx.py

```
.....
from selenium import webdriver
#引入 ActionChains 类
from selenium.webdriver.common.action_chains import ActionChains

driver = webdriver.Firefox()
driver.get("http://www.baidu.com")
.....
#定位到要悬停的元素
above = driver.find_element_by_id("xx")
#对定位到的元素执行悬停操作
ActionChains(driver).move_to_element(above).perform()
....
```

鼠标双击操作

`double_click(on_element)`方法用于模拟鼠标双击操作，用法同上。

xx.py

```
.....
from selenium import webdriver
#引入 ActionChains 类
from selenium.webdriver.common.action_chains import ActionChains

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

```
#定位到要悬停的元素
double_click = driver.find_element_by_id("xx")
#对定位到的元素执行双击操作
ActionChains(driver).double_click(double_click).perform()
....
```

鼠标推放操作

`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`: 鼠标拖动的源元素。
- `target`: 鼠标释放的目标元素。

xx.py

```
.....
from selenium import webdriver
#引入ActionChains 类
from selenium.webdriver.common.action_chains import ActionChains

driver = webdriver.Firefox()
.....
#定位元素的源位置
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")

#执行元素的拖放操作
ActionChains(driver).drag_and_drop(element,target).perform()
....
```

4.5 键盘事件

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, `Keys()`类提供键盘上几乎所有按键的方法，前面了解到 `send_keys()`方法可以模拟键盘输入，除此之外它还可以模拟键盘上的一些组合键，例 `Ctrl+A`、`Ctrl+C` 等。

xx.py

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

```

from selenium import webdriver
#引入 Keys 模块
from selenium.webdriver.common.keys import Keys

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

#输入框输入内容
driver.find_element_by_id("kw").send_keys("seleniummm")

#删除多输入的一个 m
driver.find_element_by_id("kw").send_keys(Keys.BACK_SPACE)

#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")

#ctrl+a 全选输入框内容
driver.find_element_by_id("kw").send_keys(Keys.CONTROL, 'a')

#ctrl+x 剪切输入框内容
driver.find_element_by_id("kw").send_keys(Keys.CONTROL, 'x')

#ctrl+v 粘贴内容到输入框
driver.find_element_by_id("kw").send_keys(Keys.CONTROL, 'v')

#通过回车键盘来代替点击操作
driver.find_element_by_id("su").send_keys(Keys.ENTER)

driver.quit()

```

需要说明的是上面脚本没什么实际意义，但向我们展示了模拟键盘各种按键与组合键用法。

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)
send_keys(Keys.F1)	键盘 F1
.....	
send_keys(Keys.F12)	键盘 F12

4.6 获得验证信息

当我们在编写功能测试用例时，一般会有预期结果，这个预期结果是由测试人员在执行用例的过程中通过眼睛与思维进行判断的。自动化测试用例由交给机器去执行的，机器并不像人一样是有思维和判断能力的，那么是不是模拟的各种操作页面的动作没有报错就说明用例执行成功的呢？并不见得，比如我们模拟百度搜索的脚本，结果有一天在新版本上线后搜索的结果显示少一条，但脚本执行并没有报错，那么这个 bug 就永远不会被自动化测试发现。

那么是不是在跑自动化测试的时候需要一个人盯着脚本的执行来辨别执行结果呢？如果是这样的话，自动化测试也就失去了“自动化”的意义。在自动化用例执行完成之后，我们可以从页面上获取一些信息来“证明”用例执行是成功还是失败。

通常我们用得最多的几种验证信息分别是 title 、 URL 和 text，text 方法在前面已经讲，它用于获取标签对之间的文本信息。

下面仍以 126 邮箱例子，来获取这些信息：

login126.py

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

driver = webdriver.Firefox()
driver.get("http://www.126.com")

print 'Before login====='

#打印当前页面 title
title = driver.title
print title
```

```
#打印当前页面 URL
now_url = driver.current_url
print now_url

#执行邮箱登录
driver.find_element_by_id("idInput").clear()
driver.find_element_by_id("idInput").send_keys("username")
driver.find_element_by_id("pwdInput").clear()
driver.find_element_by_id("pwdInput").send_keys("password")
driver.find_element_by_id("loginBtn").click()

print 'After login=========='

#再次打印当前页面 title
title = driver.title
print title

#打印当前页面 URL
now_url = driver.current_url
print now_url

#获得登录的用户名
user = driver.find_element_by_id('spnUid').text
print user

driver.quit()
```

运行脚本执行结果：

Python Shell

```
>>> ===== RESTART =====
>>>
Before login=====
126 网易免费邮--你的专业电子邮局
http://www.126.com/
After login=====
网易邮箱 6.0 版
http://mail.126.com/jsp?sid=VBqseScEOCvclcjRdjEEYbWiQFuWVamg&df=mai
l126_letter#module=welcome.WelcomeModule%7C%7B%7D
testingwtb@126.com
```

`title` 用于获得当前页面的标题。

`current_url` 用户获得当页面的 URL。

通过打印信息，我们发现登录前后的 `title` 和 URL 明显不同，那么我们可把登录之后的这些信息存放

起来，作为登录是否成功的验证信息，当然，这里 URL 每次登录都会有所变化是不能拿来作验证信息的。title 可以拿来做验证信息但它并不能很明确的表示是哪个登录登录成功了。那么通过 text 获取的用户文本（testingwtb@126.com）是很好的验证信息。

4.7 设置元素等待

如今大多数的 web 应用程序使用 AJAX 技术。当浏览器在加载页面时，页面内的元素可能并不是同时被加载完成的，这给元素的定位添加的困难。如果因为在加载某个元素时延迟而造成 ElementNotVisibleException 的情况出现，那么就会降低的自动化脚本的稳定性。

WebDriver 提供了两种类型的等待：显式等待和隐式等待。

4.7.1 显式等待

显式等待使 WebDriver 等待某个条件成立时继续执行，否则在达到最大时长时抛弃超时异常（TimeoutException）。

baidu.py

```
#coding=utf-8
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

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

element = WebDriverWait(driver, 5, 0.5).until(
    EC.presence_of_element_located((By.ID, "kw"))
)
element.send_keys('selenium')
driver.quit()
```

WebDriverWait()

它是由 webdirver 提供的等待方法。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。具体格式如下：

WebDriverWait(driver, timeout, poll_frequency=0.5, ignored_exceptions=None)

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

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。

`until()`

`WebDriverWait()`一般由 `until()`（或 `until_not()`）方法配合使用，下面是 `until()`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，直到返回值为 `Ture`。

`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`。

Expected Conditions

在本例中，我们在使用 `expected_conditions` 类时对其时行了重命名，通过 `as` 关键字对其重命名为 `EC`，并调用 `presence_of_element_located()`判断元素是否存在。

`expected_conditions` 类提供一些预期条件的实现。

`title_is` 用于判断标题是否 `xx`。

`title_contains` 用于判断标题是否包含 `xx` 信息。

`presence_of_element_located` 元素是否存在。

`visibility_of_element_located` 元素是否可见。

`visibility_of` 是否可见

`presence_of_all_elements_located` 判断一组元素的是否存在

`text_to_be_present_in_element` 判断元素是否有 `xx` 文本信息

`text_to_be_present_in_element_value` 判断元素值是否有 `xx` 文本信息

`frame_to_be_available_and_switch_to_it` 表单是否可用，并切换到该表单。

`invisibility_of_element_located` 判断元素是否隐藏

`element_to_be_clickable` 判断元素是否点击，它处于可见和启动状态

`staleness_of` 等到一个元素不再是依附于 DOM。

`element_to_be_selected` 被选中的元素。

element_located_to_be_selected	一个期望的元素位于被选中。
element_selection_state_to_be	一个期望检查如果给定的元素被选中。
element_located_selection_state_to_be	期望找到一个元素并检查是否选择状态
alert_is_present	预期一个警告信息

除了 `expected_conditions` 所提供的预期方法，我们也可以使用前面学过的 `is_displayed()` 方法来判断元素是否可。

baidu.py

```
#coding=utf-8
from selenium import webdriver
from selenium.webdriver.support.ui import WebDriverWait

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

input_ = driver.find_element_by_id("kw")
element = WebDriverWait(driver,5,0.5).until(
    lambda driver : input_.is_displayed()
)
input_.send_keys('selenium')
driver.quit()
```

`lambad` 为 Python 创建匿名函数的关键字。关于 Python 匿名函数的使用这里不再进行讲解，请参考 Python 相关教程。

定义 `input_` 为百度输入框，通过 `is_displayed()` 判断其是否可见。

4.7.2 隐式等待

隐式等待是通过一定的时长等待页面所元素加载完成。如果超出了设置的时长元素还没有被加载则抛出 `NoSuchElementException` 异常。WebDriver 提供了 `implicitly_wait()` 方法来实现隐式等待，默认设置为 0。它的用法相对来说要简单的多。

baidu.py

```
#coding=utf-8
from selenium import webdriver
from selenium.webdriver.support.ui import WebDriverWait
```

```

driver = webdriver.Firefox()
driver.implicitly_wait(10)
driver.get("http://www.baidu.com")

input_ = driver.find_element_by_id("kw22")
input_.send_keys('selenium')

driver.quit()

```

`implicitly_wait()`默认参数的单位为秒，本例中设置等待时长为 10 秒，首先这 10 秒并非一个固定的等待时间，它并不影响脚本的执行速度。其次，它并不真对页面上的某一元素进行等待，当脚本执行到某个元素定位时，如果元素可定位那么继续执行，如果元素定位不到，那么它将以轮询的方式不断的判断元素是否被定位到，假设在第 6 秒钟定位到元素则继续执行。直接超出设置时长（10 秒）还没定位到元素则抛出异常。

在上面的例子中，显然百度输入框的定位 `id=kw22` 是有误的，那么在超出 10 秒后将抛出异常。

Python Shell

```

>>> ===== RESTART =====
>>>
Traceback (most recent call last):
  File "C:\Users\fnngj\Desktop\aaa.py", line 19, in <module>
    bb = driver.find_element_by_id("kw22")
  File "C:\Python27\lib\site-packages\selenium\webdriver\remote\webdriver.py",
line 206, 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 662, in find_element
    {'using': by, 'value': value})['value']
  File "C:\Python27\lib\site-packages\selenium\webdriver\remote\webdriver.py",
line 173, in execute
    self.error_handler.check_response(response)
  File
"C:\Python27\lib\site-packages\selenium\webdriver\remote\errorhandler.py",
line 166, in check_response
    raise exception_class(message, screen, stacktrace)
NoSuchElementException: Message: Unable to locate element:
{"method":"id","selector":"kw22"}

```

4.7.3 sleep 休眠方法

有时间我们希望脚本执行到某一位置时做固定时间的休眠，尤其是在脚本调试的过程中。那么可以使

用 sleep()方法，需要说明的是 sleep()由 Python 的 time 模块提供。

baidu.py

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

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

sleep(2)
driver.find_element_by_id("kw").send_keys("webdriver")
driver.find_element_by_id("su").click()
sleep(3)

driver.quit()
```

当执行到 sleep()方法时会固定的休眠所设置的时长，然后再继续执行。sleep()方法默认参数以秒为单位，如果设置时长小于 1 秒，可以用小数点表示，如：sleep(0.5)

4.8 定位一组元素

在本章的第一节我们已经学习了 8 种定位方法，那 8 种定位方法是真对单元素定位的，WebDriver 还提供了与之对应的 8 种定位方法用于定位一组元素。

```
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()
```

定位一组对象的方法与定位单个对象的方法类似，唯一的区别是在单词 element 后面多了一个 s 表示复数。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复选框都被勾选。

- 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>
<link href="http://cdn.bootcss.com/bootstrap/3.3.0/css/bootstrap.min.css" rel="stylesheet" />
<script
src="http://cdn.bootcss.com/bootstrap/3.3.0/css/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>
```

这里手动创建一个 `checkbox.html` 的页面，通过浏览器打开，效果如图。

checkbox

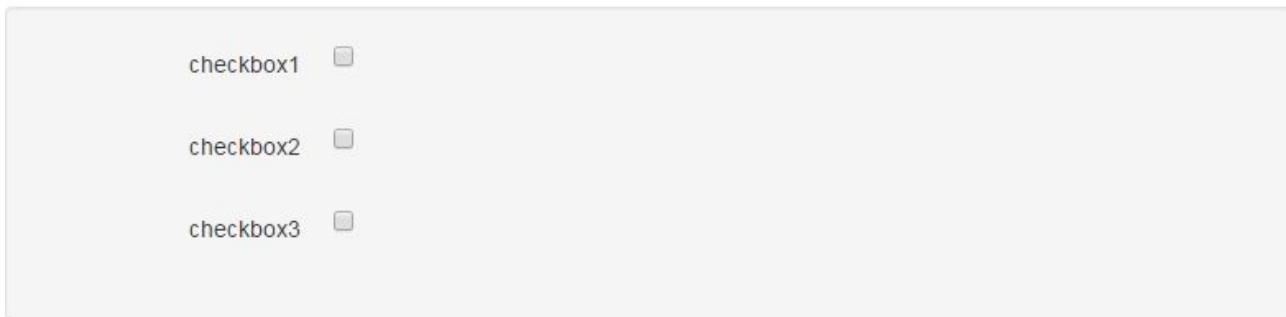


图 4.9 多复选框

为了使页面更美观，在代码中添加了 bootstrap 在线样式的引用。下面就通过例子来操作页面上的这一组复选框。

checkbox.py

```
#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 i in inputs:
    if i.get_attribute('type') == 'checkbox':
        i.click()

driver.quit()
```

前面在提到通过 tag name 的定位方式很难定位到单个元素，因为元素标签名重复的概率很高。那么在定位一组元素时，这种方式就派上了用场。上面的例子中先通过 `find_elements_by_tag_name()` 找到一组标签名为 `input` 的元素。然后通过 `for` 循环进行遍历。在遍历的过程中通过 `get_attribute()` 方法获取元素的 `type` 属性，判断是否为“checkbox”，如果为“checkbox”那么这个元素就是一个复选框。对其进行 `click()` 勾选操作。

在本例中，因为通过浏览器打开的是一个本地的 `html` 文件，所以需要用到 Python 的 `os` 模块，`path.abspath()` 方法用于获取当前路径下的文件。

除此之外，我们也可以使用 XPath 或 CSS 来判断属性值，从而进行点击操作。

checkbox.py

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

driver = webdriver.Firefox()
file_path = 'file:///+' + os.path.abspath('checkbox.html')
driver.get(file_path)

#通过 XPath 找到 type=checkbox 的元素
checkboxes = driver.find_elements_by_xpath("//input[@type='checkbox']")

#通过 CSS 找到 type=checkbox 的元素
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(checkboxes)

# 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()
```

通过 XPath 或 CSS 来查找一组元素，少去判断步骤，因为它本身已经做了判断，只需要循环对每一个元素进行 click() 勾选即可。

除此之外例子中还用到了 Python 提供的两个有趣的方法。len() 可获取元素的个数，通过 print 打印会得到一个 3 的结果。pop() 函数用于获取列表中的一个元素（默认为最后一个元素），并且返回该元素的值。因为前的循环已经所有复选框都勾选上了，再对这一组元素执行 pop().click() 其实是对后一个元素取消勾选。如果只想勾选一组元素中的某一个呢。

pop()或 pop(-1) 默认获取一组元素中的最后一个。

pop(0) 默认获取一组元素中的第一个。

pop(1) 默认获取一组元素中的第二个。

.....

这样就可以操作这一组元素中的元素了，只用数一数操作的元素是这一组中的第几个。

4.9 多表单切换

在 web 应用中经常会遇到 frame 嵌套页面的应用，页 WebDriver 每次只能在一个页面上识别元素，对于 frame 嵌套内的页面上的元素，直接定位是定位不到的。这个时候就需要通过 `switch_to_frame()` 方法将当前定位的主体切换了 frame 里。

frame.html

```
<html>
<head>
<link href="http://cdn.bootcss.com/bootstrap/3.3.0/css/bootstrap.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="if" name="nf" src="http://www.baidu.com" width="800"
height="300">
        </iframe>
    </div>
</div>
</body>
<script
src="http://cdn.bootcss.com/bootstrap/3.3.0/css/bootstrap.min.js"></script>
</html>
```

在上面的 html 代码中通过 iframe 表单嵌入一个百度页面，通过浏览器打开如图 4.10



图 4.9 iframe 嵌入百度首页

这个时候直接定位百度的输入框一定会报找不到元素的错误。那么可以使用 `switch_to_frame()` 先找到

frame.html 中的<iframe>标签，然后再定位百度输入框。

frame.py

```
#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)

#切换到 iframe (id = "if")
driver.switch_to_frame("if")

#下面就可以正常的操作元素了
driver.find_element_by_id("kw").send_keys("selenium")
driver.find_element_by_id("su").click()
time.sleep(3)

driver.quit()
```

`switch_to_frame()` 默认可以直接取表单的 id 或 name 属性进行切换。如

frame.py

```
.....
#id = "if"
driver.switch_to_frame("if")

#name = "nf"
driver.switch_to_frame("nf")
.....
```

那么如果 iframe 没有可用的 id 和 name 可以通过下面的方式进行定位：

frame.py

```
.....
#先通过 xpath 定位到 iframe
xf = driver.find_element_by_xpath('//*[@@class="if"]')

#再将定位对象传给 switch_to_frame() 方法
driver.switch_to_frame(xf)
.....

driver.switch_to_default_content()
```

如果完成了在当前表单上的操作可以通过 `switch_to_default_content()`方法返回到上一层表单。该方法

不用指定某个表单的返回， 默认对应与它最近的 `switch_to_frame()`方法。

4.10 多窗口切换

有时候需要在不同的窗口切换，从而操作不同的窗口上的元素。在 selenium1.0 中这个问题比较难处理。但 WebDriver 提供了 `switch_to_window()`方法可以切换到任意的窗口。

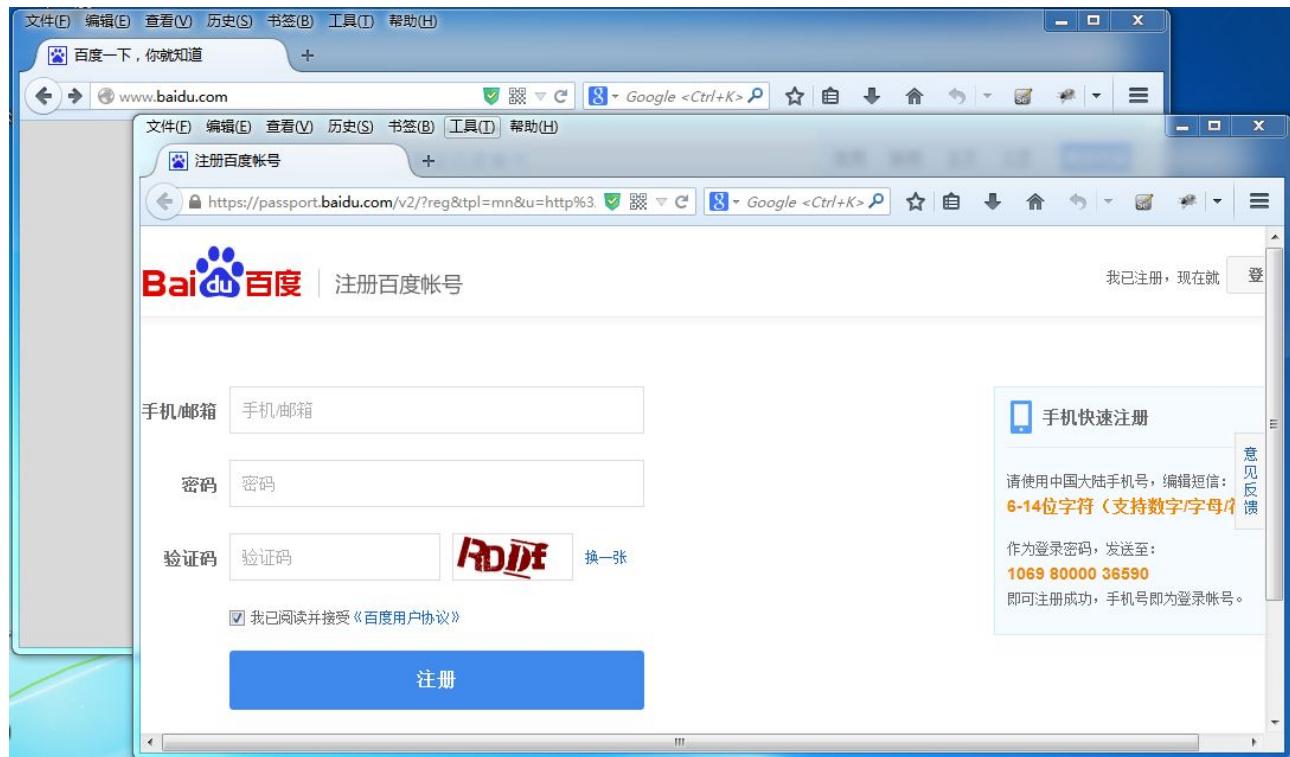


图 4.10 多窗口

这里以百度首页与注册页为例，演示在不同窗口切换。

windows.py

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

driver = webdriver.Firefox()
driver.implicitly_wait(10)
driver.get("http://www.baidu.com")

#获得百度搜索窗口句柄
sreach_windows= driver.current_window_handle

driver.find_element_by_link_text(u'登录').click()
driver.find_element_by_link_text(u"立即注册").click()
```

```

#获得当前所有打开的窗口的句柄
all_handles = driver.window_handles

#进入注册窗口
for handle in all_handles:
    if handle != sreach_windows:
        driver.switch_to_window(handle)
        print 'now register window!'
        driver.find_element_by_name("account").send_keys('username')
        driver.find_element_by_name('password').send_keys('password')
        #......

#进入搜索窗口
for handle in all_handles:
    if handle == sreach_windows:
        driver.switch_to_window(handle)
        print 'now sreach window!'
        driver.find_element_by_id('TANGRAM__PSP_2__closeBtn').click()
        driver.find_element_by_id("kw").send_keys("selenium")
        driver.find_element_by_id("su").click()
        time.sleep(5)

driver.quit()

```

整个脚本的处理过程：首先打开百度首页，通过 current_window_handle 获得当前窗口的句柄，并给变量 sreach_handle。接着打开登录弹窗，在登录窗口上点击“立即注册”从而打开新的注册窗口。通过 window_handles 获得当前打开的所窗口的句柄，赋值给变量 all_handles。

第一个循环遍历 all_handles，如果 handle 不等于 sreach_handle，那么一定是注册窗口，因为脚本执行只打开的两个窗口。所以，通过 switch_to_window() 切换到注册页进行注册操作。第二个循环类似，不过这一次判断如果 handle 等于 sreach_handle，那么切换到百度搜索页，关闭之前打开的登录弹窗，然后执行搜索操作。

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()	

用于切换到相应的窗口，与上一节的 switch_to_frame() 是类似，前者用于不同窗口的切换，后者用于不同表单之间的切换。

4.11 警告框处理

在 WebDriver 中处理 JavaScript 所生成的 alert、confirm 以及 prompt 是很简单的。具体做法是使用 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。

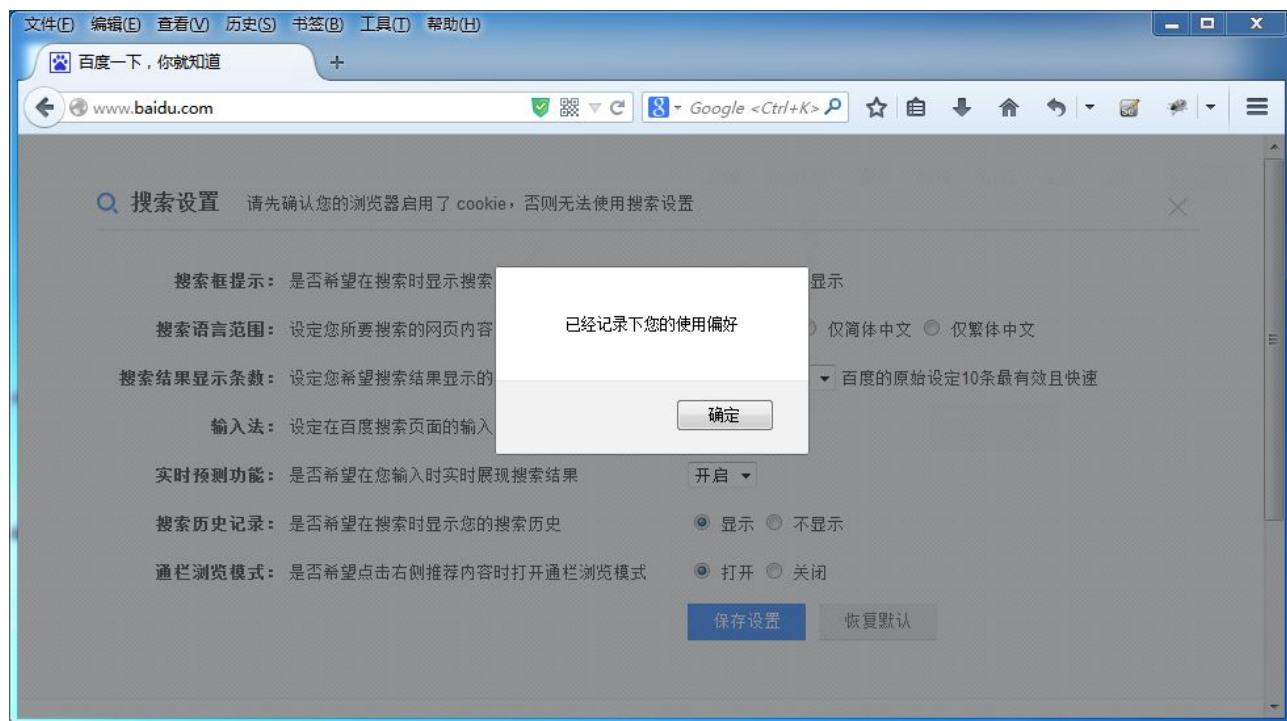


图 4.11 百度搜索保存设置弹窗

如图 4.11 百度搜索设置的弹出的弹窗是不能通过前端工具对其进行定位的，这个时候就可以通过 switch_to_alert()方法接受这个弹窗。

alert.py

```
#coding=utf-8
from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains

driver = webdriver.Firefox()
driver.implicitly_wait(10)
driver.get('http://www.baidu.com')

#鼠标悬停相“设置”链接
link = driver.find_element_by_link_text(u'设置')
ActionChains(driver).move_to_element(link).perform()

#打开搜索设置
```

```

driver.find_element_by_class_name('setpref').click()

#保存设置
driver.find_element_by_css_selector('#gxszButton > a.prefpanelgo').click()

#接收弹窗
driver.switch_to_alert().accept()

driver.quit()

```

从这个例子中我们重温了 4.4 节中 ActionChains 类所提供的 move_to_element() 鼠标悬停的使用，将鼠标悬停在“搜索”链接上然后弹出下拉菜单。在菜单中点击“搜索设置”按钮。设置完成点击“保存设置”弹出警告框。通过 switch_to_alert() 方法获取当前页上的警告框， accept() 接受警告框。

4.12 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。

对于 web 页面的上功能，点击“上传”按钮需要打开本地的 Window 窗口，从窗口选择本地文件进行上传，那么 WebDriver 对于 Windows 的控件是无能为力的。所以，对于初学者来说一般思路会卡在如何实现 Window 控件的问题上。

对于 web 页面的上传功能一般会有以下几种方式。

普通上传：普通的附件上传都是将本地文件的路径作为一个值放 input 标签中，通过 form 表单提交的时候将这个值提交给服务器。

插件上传：一般是指基于 Flash 与 JavaScript 或 Ajax 等技术所实现的上传功能或插件。

4.12.1 send_keys 实现上传

对于通过 input 标签实现的通过上传，可以将其看作一个输入框，通过 send_keys() 传入本地文件路径从而模拟上传功能。

upfile.html

```

<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8" />
<title>upload_file</title>
<link href="http://cdn.bootcss.com/bootstrap/3.3.0/css/bootstrap.min.css"
rel="stylesheet" />

```

```

</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://cdn.bootcss.com/bootstrap/3.3.0/css/bootstrap.min.js"></script>
</html>

```

通过浏览器打 upfile.html 文件，效果如图 4.12



图 4.12 普通上传功能

upfile.py

```

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

driver = webdriver.Firefox()

#打开上传功能页面
file_path = 'file:///+' + os.path.abspath('upfile.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:\\upload_file.txt')

driver.quit()

```

通过这种方法上传，就绕开了操作 Windows 控件的步骤。如果能找上传的 input 标签，那么基本都可以通过 send_keys()方法向其输入一个文件地址来实现上传。

4.12.2 AutoIt 实现上传

AutoIt 目前最新是 v3 版本,这是一个使用类似 BASIC 脚本语言的免费软件,它设计用于 Windows GUI(图形用户界面)中进行自动化操作。它利用模拟键盘按键,鼠标移动和窗口/控件的组合来实现自动化任务。

官方网站: <https://www.autoitscript.com/site/>

从网站上下载 AutoIt 并安装, 安装完成在菜单中会看到图 4.13 的目录:



图 4.13 AutoIt 菜单

AutoIt Windows Info 用于帮助我们识别 Windows 控件信息。

Compile Script to.exe 用于将 AutoIt 生成 exe 执行文件。

Run Script 用于执行 AutoIt 脚本。

SciTE Script Editor 用于编写 AutoIt 脚本。

下面以操作 upload.html 上传弹出的窗口为例讲解 AutoIt 实现上传过程。

1、首先打开 AutoIt Windows Info 工具, 鼠标点击 Finder Tool, 鼠标将变成一个小风扇形状的图标, 按住鼠标左键拖动到需要识别的控件上。

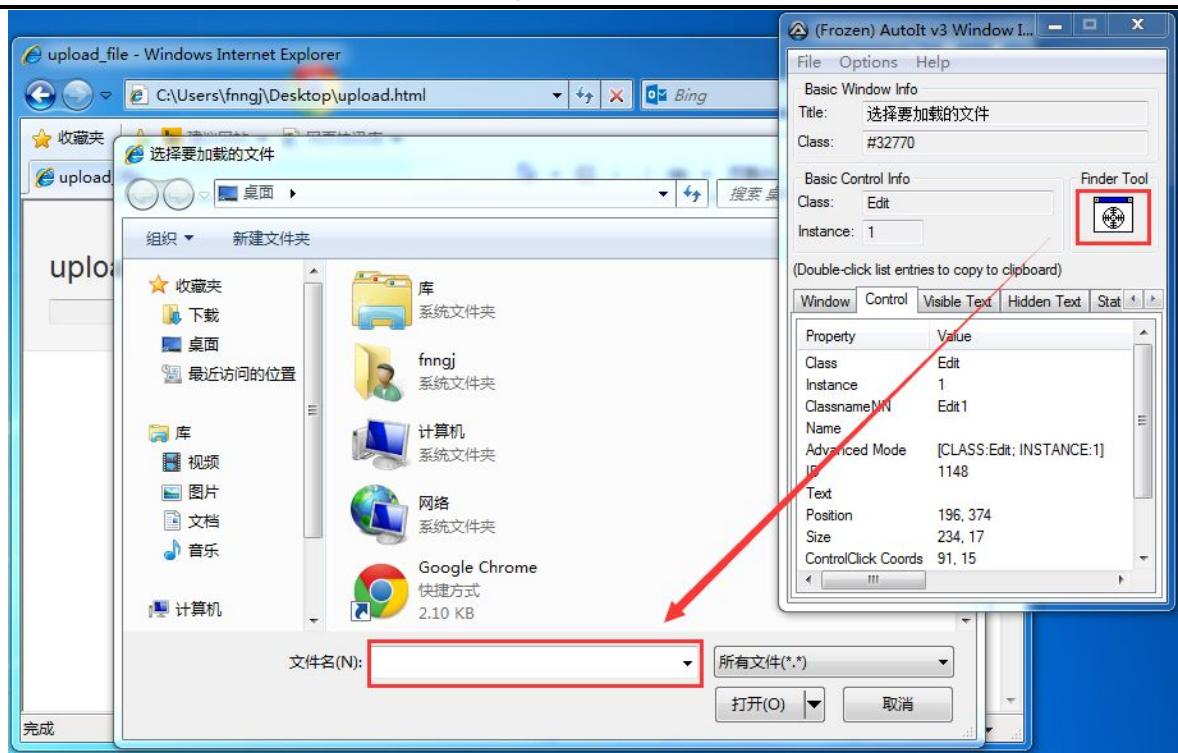


图 4.14 AutoIt Windows Info 识别“文件名”输入框控件

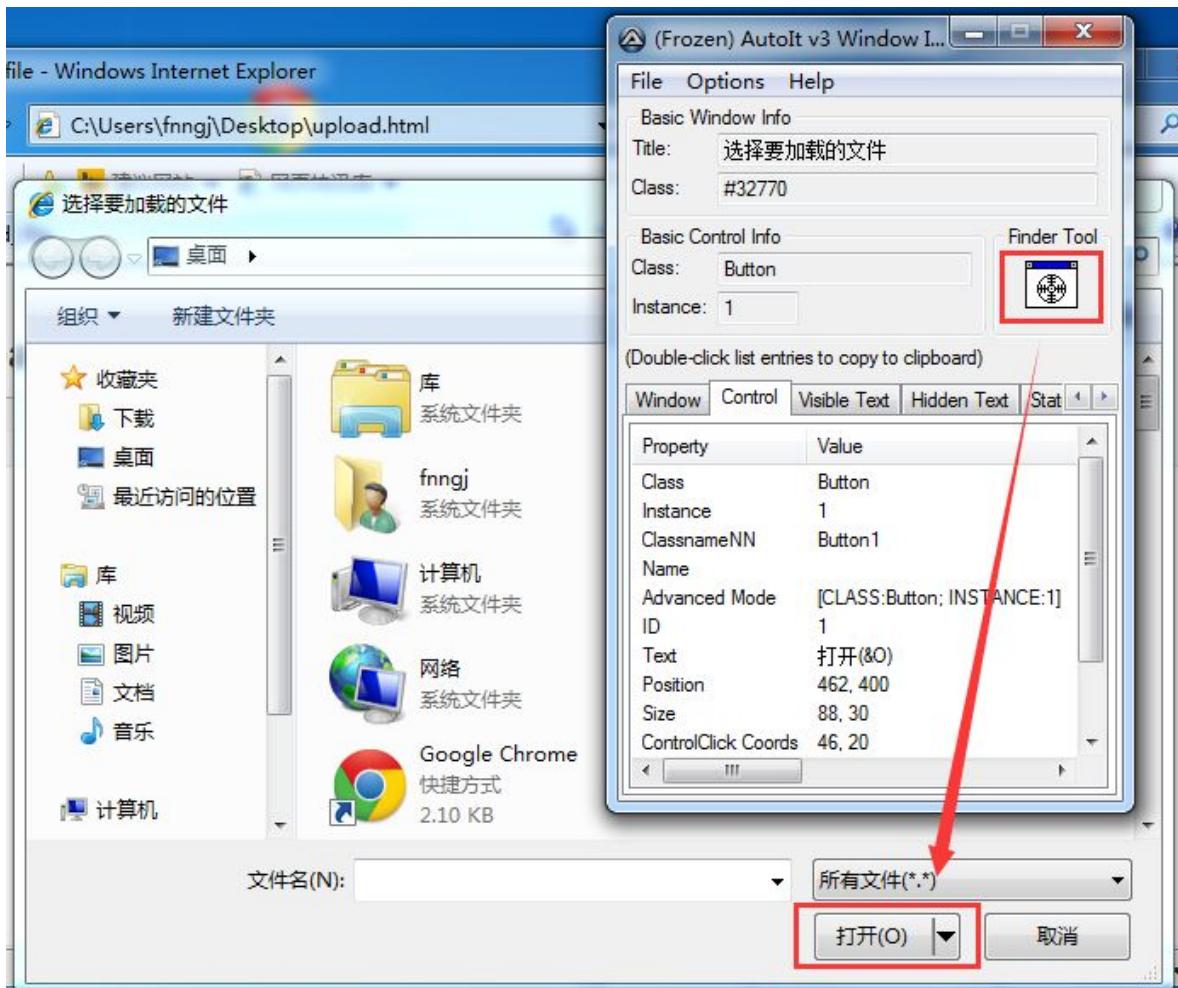


图 4.15 AutoIt Windows Info 识别“打开”按钮控件

如图 4.14、4.15，通过 AutoIt Windows Info 获得以下信息。

窗口的 title 为“选择要加载的文件”，标题的 Class 为“#32770”。

文件名输入框的 class 为“Edit”，Instance 为“1”，所以 ClassnameNN 为“Edit1”。

打开按钮的 class 为“Button”，Instance 为“1”，所以 ClassnameNN 为“Button1”。

2、根据 AutoIt Windows Info 所识别到的控件信息打开 SciTE Script Editor 编辑器，编写脚本。

upfile.au3

```
;ControlFocus("title","text",controlID) Edit1=Edit instance 1
ControlFocus("选择要加载的文件", "", "Edit1")

; Wait 10 seconds for the Upload window to appear
WinWait("[CLASS:#32770]", "", 10)

; Set the File name text on the Edit field
ControlSetText("选择要加载的文件", "", "Edit1", "D:\\upload_file.txt")
Sleep(2000)

; Click on the Open button
ControlClick("选择要加载的文件", "", "Button1");
```

ControlFocus()方法用于识别 Window 窗口。WinWait()设置 10 秒钟用于等待窗口的显示，其用法与 WebDriver 所提供的 implicitly_wait()类似。ControlSetText()用于向“文件名”输入框内输入本地文件的路径。这里的 Sleep()方法与 Python 中 time 模块提供的 Sleep()方法用法一样，不过它是以毫秒为单位，Sleep(2000)表示固定休眠 2000 毫秒。ControlClick()用于点击上传窗口中的“打开”按钮。

AutoIt 的脚本已经写好了，可以通过菜单栏“Tools”-->“Go”（或按键盘 F5）来运行一个脚本吧！注意在运行时上传窗口当前处于打开状态。

3、脚本运行正常，将其保存为 upfile.au3，这里保存的脚本可以通过 Run Script 工具将其打开运行，但我们的目的是希望这个脚本被 Python 程序调用，那么就需要将其生成 exe 程序。打开 Compile Script to.exe 工具，将其生成为 exe 可执行文件。如图 4.16，

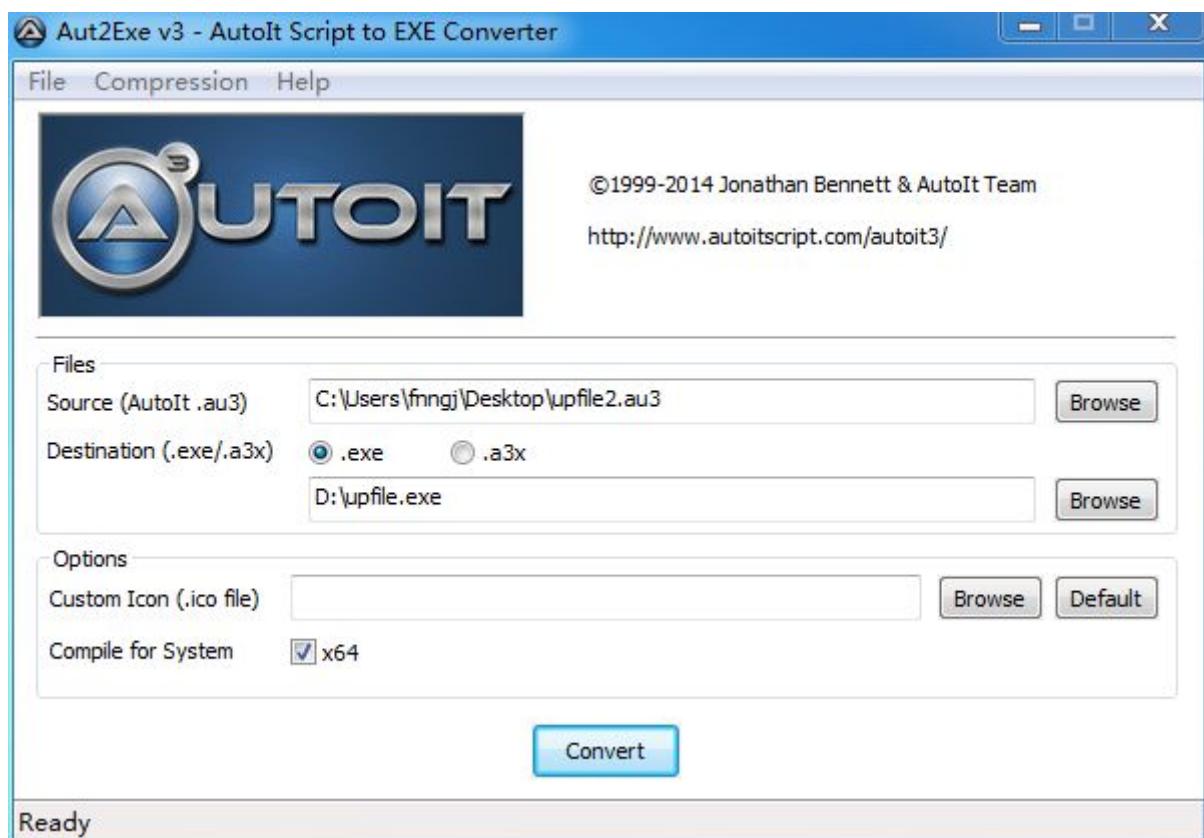


图 4.16 Compile Script to.exe 生成 exe 程序

点击“Browse”选择 upfile.au3 文件，点击“Convert”按钮将其生成为 upfile.exe 程序。

4、下面就是通过自动化测试脚本调用 upfile.exe 程序实现上传了。

upfile.py

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

driver = webdriver.Firefox()

#打开上传功能页面
file_path = 'file:/// + os.path.abspath('upfile.html')
driver.get(file_path)

#点击打开上传窗口
driver.find_element_by_name("file").click()
#调用 upfile.exe 上传程序
os.system("D:\\upfile.exe")

driver.quit()
```

通过 Python 的 os 模块的 system()方法可以调用 exe 程序并执行。

4.14 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。下面以 FireFox 为例执行文件的下载。

downfile.py

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

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") #下载文件的类型

driver = webdriver.Firefox(firefox_profile=fp)
driver.get("http://pypi.Python.org/pypi/selenium")
driver.find_element_by_partial_link_text("selenium-2").click()
```

为了让 FireFox 让浏览器能实现文件的载，我们需要通过 FirefoxProfile() 对其参数做一个设置。

`browser.download.folderList`

设置成 0 代表下载到浏览器默认下载路径；设置成 2 则可以保存到指定目录。

`browser.download.manager.showWhenStarting`

是否显示开始，True 为显示，False 为不显示。

`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。

`browser.helperApps.neverAsk.saveToDisk`

指定要下载页面的 Content-type 值，“application/octet-stream”为文件的类型。HTTP Content-type 常用对照表：<http://tool.oschina.net/commons>

这些参数的设置可以通过在 Firefox 浏览器地址栏输入：about:config 进行设置，如图 4.17。

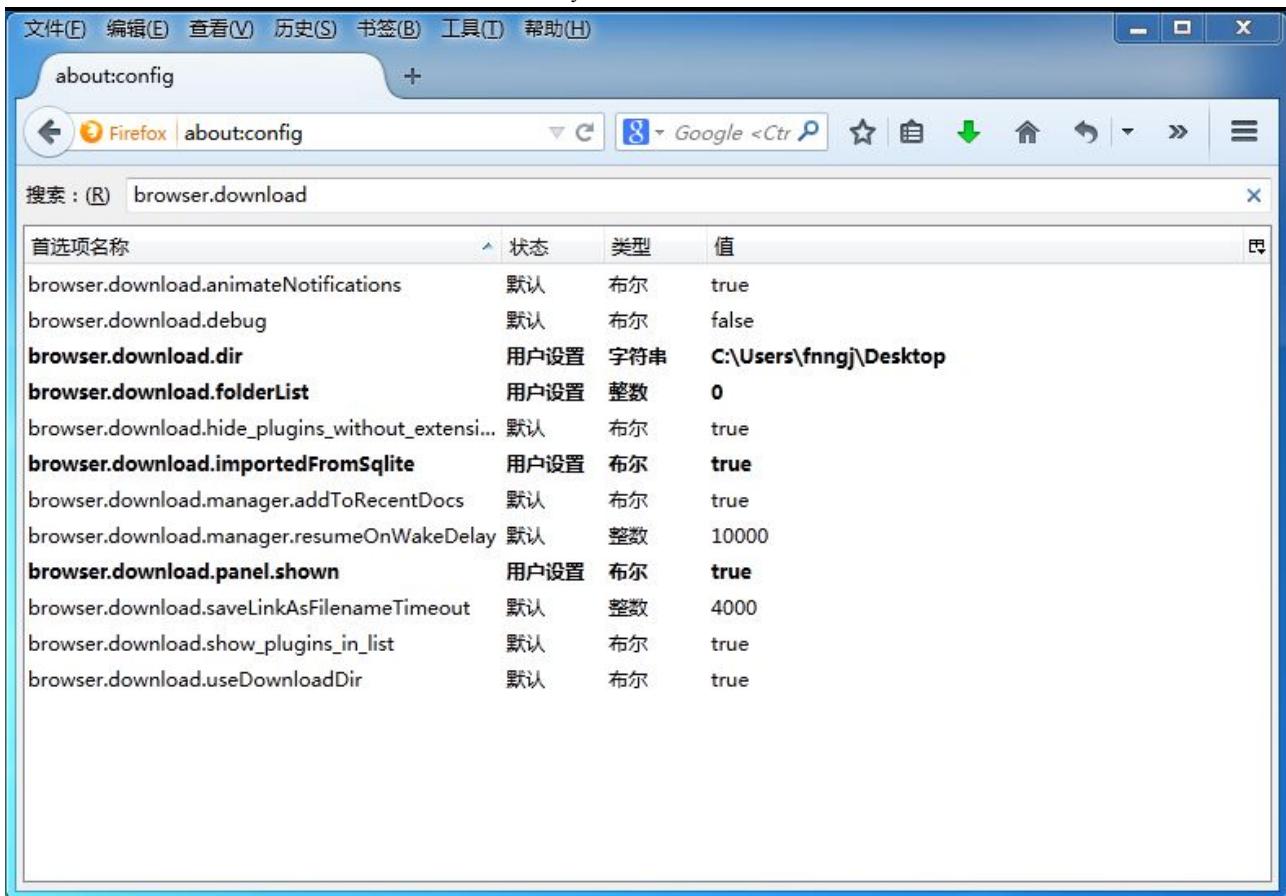


图 4.17 FireFire 参数设置

将所有设置信息在调用 webdriver 的 Firefox()方法时作为参数传递给浏览器。下面 FireFox 浏览器在下载时就根据这些设置信息将文件下载的当前脚本的目录下。

那么对于上面的方法只适用于 Firefox 浏览器，对于其它浏览器浏览设置方法会有所不同。那么比较通用的方法还是借助 AutoIt 来操作 Windows 控件进行下载，由于上一小节是中已经详细的讲解了 AutoIt 的使用，这里就不再重复介绍。

4.15 操作 Cookie

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 提供了操作 Cookie 的相关方法可以读取、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) 返回有特定 name 值的 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

下面通过 get_cookies()来获取当前浏览器的 cookie 信息。

cookie.py

```
#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()

```

执行结果：

```

>>> ===== RESTART =====
>>>
[{'domain': u'.youdao.com',
 u'secure': False,
 u'value': u'aGFzbG9nZ2VkPXRydWU=',
 u'expiry': 1408430390.991375,
 u'path': u'/',
 u'name': u'_PREF_ANONYUSER__MYTH'},
 {'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'},
 {'domain': u'',
 u'name': u'JSESSIONID',
 u'value': u'abcUX9zdw0minadIhtvcu',
 u'secure': False}]

```

通过打印结果可以看出，cookie 是以字典的形式进行存放的，知道了 cookie 的存放形式，那么我们就可以按照这种形式向浏览器中写入 cookie 信息。

cookie.py

```

#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':'keyaaaaaaaa', 'value':'value-bbbbbbb'})

#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'])

driver.quit()
```

执行结果：

```
>>> ===== RESTART =====
>>>
YOUDAO_MOBILE_ACCESS_TYPE -> 1
_PREF_ANONYUSER__MYTH -> aGFzbG9nZ2VkPXRYdWU=
OUTFOX_SEARCH_USER_ID -> -1046383847@218.17.158.115
JSESSIONID -> abc7qSE_SBGsVgnVLBvcu
keyaaaaaaaa -> value-bbbbbbb
```

从打印结果可以看到最后一条 cookie 信息是在脚本执行过程中通过 add_cookie()方法添加的。通过遍历得到的所 cookie 信息从而找到 key 为“name”和“value”的特定 cookie 的 value。

那么在什么情况下会用到 cookie 的操作呢？例如开发人员开发一个功能，当用户登录后，会将用户的用户名写入浏览器 cookie，指定的 key 为“username”，那么我们就可以通过 get_cookies() 找到 useranme，打印 vlaue，如果找不到 username 或对应的 value 为空，那么说明保存浏览器的 cookie 是有问题的。

delete_cookie() 和 delete_all_cookies() 的使用也很简单，前者通过 name 值到一个特定的 cookie 将其删除，后者直接删除浏览器中的所有 cookies()信息。

4.16 调用 JavaScript

WebDiver 不能操作本地 Windows 控件，但对于浏览器上的控件也不是都可以操作的。比如浏览器上的滚动条，虽然 WebDriver 提供操作浏览器的前进和后退按钮，但对于滚动条并没有提供相应用的方法。那么在这种情况下就可以借助 JavaScript 方法来控制浏览器滚动条。WebDriver 提供了 execute_script()方法来执行 JavaScript 代码。

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

html

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

document.body.scrollTop

网页被卷去的高。scrollTop 设置或获取滚动条与最顶端之间的距离。如果想让滚动条处于顶部，那么可以设置 scrollTop 的值为 0，如果想让滚动条处于最底端，可以将这个值设置的足够大，大于窗口的高度即可。scrollTop 的值以像素为单位。

html

```
#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()
```

通过浏览器打开百度进行搜索，搜索的一屏无法完全显示将会出现滚动条。这个时候就可以通过 JavaScript 代码控制滚动条在任意位置，需要改变的就是 scrollTop 的值。通过 execute_script()方法来执行这段 JavaScript 代码。如图 4.18。



图 4.18 通过 JavaScript 控制浏览器滚动条位置。

当然，JavaScript 的作用不仅于此，它同样可操作页面上的元素或让，或让这个元素隐藏。学习和使用 JavaScript 不仅可以让你对前端技术有更深的认识，还可以让你对 web 页面上元素的操作变得游刃有余。

4.17 窗口截图

自动化脚本是交给工具去执行，有时候打印的错误信息并不十分明确，如果在脚本执行出错的时候将对当前窗口截图保存，那么通过图片信息会更直观帮助我们找到脚本出错的原因。Webdriver 提供了截图函数 `get_screenshot_as_file()` 来截取当前窗口。

baidu.py

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

driver = webdriver.Chrome()
driver.get('http://www.baidu.com')

try:
    driver.find_element_by_id('kw_error').send_keys('selenium')
    driver.find_element_by_id('su').click()
```

```
except :  
    driver.get_screenshot_as_file("D:\\baidu_error.jpg")  
    driver.quit()
```

在本例中用到了 Python 的异常处理，要本例中百度输入框的 id=kw_error 会定位不到元素，那么 try 就会捕捉到这个异常，从而执行 except，在 except 中执行 get_screenshot_as_file() 对当前窗口进行截图，这里需要指定图片的保存路径及文件名，并且关闭当前驱动。

脚本运行完成打开 D 盘就可以找到 baidu_error.jpg 图片文件了。

4.18 关闭窗口

在前页的例子中我们一直在使用 quit() 方法，其含义为退出相关的驱动程序和关闭所有窗口。除此之外 WebDriver 还提供了 close() 方法，用于关闭当前窗口。当脚本在执行时打开了多个窗口，如本章的第 10 小节多窗口的处理，这个时候只能关闭其中的某一个窗口，这个时候就需要使用 close() 来关闭。

4.19 验证码的处理

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果的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比较棘手的问题。在 WebDriver 中并没有提供相应的方法来处理验证码。这里我根据自己的经验来谈谈处理验证码的几种常见方法。



图 4.19 登录框的验证码

对验证码的常见处理方式有以下几种。

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。

设计万能码的方式非常简单，只对于用户的输入信息多加一个逻辑判断，如下面的小例子：

```
baidu.py
#coding=utf-8
import random

#生成一个 1000 到 9999 之间的随机整数
verify = random.randint(1000, 9999)
print u"生成的随机数:%d" % verify

number = input(u"请输入随机数:")
print number
```

```

if number == verify:
    print u"登录成功!!"
elif number == 132741:
    print u"登录成功!!"
else:
    print u"验证码输入有误！"

```

`randint()`用于生成随机数，设置随机数的范围为1000~9999之间。运行程序分别输入正确的验证码、万能码和错误的验证码，执行结果如下：

Python Shell

```

>>> ===== RESTART =====
>>>
生成的随机数:8396
请输入随机数:8396
8396
登录成功!!
>>> ===== RESTART =====
>>>
生成的随机数:5113
请输入随机数:132741
132741
登录成功!!
>>> ===== RESTART =====
>>>
生成的随机数:1996
请输入随机数:1234
1234
验证码输入有误！

```

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。比如我们在第一次登录某网站可以勾选“记住密码”的选项，当下次再访问该网站时自动就处于登录状态了。这样其实也绕过验证码问题。那么这个“记住密码”的功能其实就记在了浏览器的 cookie 中。前面已经学了通过

WebDriver 来操作浏览器的 Cookie，可以通过 add_cookie()方法将用户名密码写入浏览器 cookie ，再次访问网站时服务器直接读取浏览器 Cookie 登录。

Python Shell

```
....  
#访问 xx 网站  
driver.get("http://www.xx.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 xx 网站，将会自动登录  
driver.get("http://www.xx.cn/")  
time.sleep(3)  
....  
driver.quit()
```

这种方式最大的问题是如何从浏览器的 Cookie 中找到用户名和密码对应的 key 值，并传传输入对应的登录信息。可以 get_cookies()方法来获取登录的所有的 cookie 信息，从中找到用户名和密码的 key。当然，如果网站登录时根本不将用户名和密码写 Cookie，这会存在一定的安全风险。那么这种方式就不起作用了。

4.20 WebDriver 原理

WebDriver 是按照 server – client 的经典设计模式设计的。

server 端就是 remote server，可以是任意的浏览器。当我们的脚本启动浏览器后，该浏览器就是 remote server，它的职责就是等待 client 发送请求并做出相应。

client 端简单说来就是我们的测试代码，我们测试代码中的一些行为，比如打开浏览器，转跳到特定的 url 等操作是以 http 请求的方式发送给被 测试浏览器，也就是 remote server； remote server 接受请求，并执行相应操作，并在 response 中返回执行状态、返回值等信息。

webdriver 的工作流程：

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 WebDriver 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。

在 Python 提供了 logging 模块，logging 模块给运行中的应用提供了一个标准的信息输出接口。它提供了 basicConfig()方法用于基本信息的定义。将 debug 模块开启。就可以捕捉到客户端与服务器的交互信息。

test.py

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

logging.basicConfig(level=logging.DEBUG)
diver = webdriver.Firefox()
diver.get("http://www.baidu.com")

diver.find_element_by_id("kw").send_keys("selenium")
diver.find_element_by_id("su").click()
diver.quit()
```

运行脚本，basicConfig()所捕捉的 log 信息。不过 basicConfig()开启的 debug 模式只能捕捉到客户端向服务器所发送的 POST 请求，而无法获取服务器所返回应答信息。我们在后面的章节中将会用 Selenium Server，通过 Selenium Server 将会获取到更详细请求与应答信息。

Python Shell

```
>>> ===== RESTART =====
>>>
DEBUG:selenium.webdriver.remote.remote_connection:POST
http://127.0.0.1:34229/hub/session {"desiredCapabilities": {"platform": "ANY",
"browserName": "firefox", "version": "", "javascriptEnabled": true}}
DEBUG:selenium.webdriver.remote.remote_connection:Finished Request
DEBUG:selenium.webdriver.remote.remote_connection:POST
http://127.0.0.1:34229/hub/session/0f0d51f5-affc-4af0-9c45-4b3c4931c601/url
{"url": "http://www.baidu.com", "sessionId":
"0f0d51f5-affc-4af0-9c45-4b3c4931c601"}
DEBUG:selenium.webdriver.remote.remote_connection:Finished Request
DEBUG:selenium.webdriver.remote.remote_connection:POST
http://127.0.0.1:34229/hub/session/0f0d51f5-affc-4af0-9c45-4b3c4931c601/element {"using": "id", "sessionId": "0f0d51f5-affc-4af0-9c45-4b3c4931c601",
"value": "kw"}
DEBUG:selenium.webdriver.remote.remote_connection:Finished Request
DEBUG:selenium.webdriver.remote.remote_connection:POST
http://127.0.0.1:34229/hub/session/0f0d51f5-affc-4af0-9c45-4b3c4931c601/element/{12722a5d-58f3-457c-ad5e-348b230c6f6a}/value {"sessionId":
```

```
"0f0d51f5-affc-4af0-9c45-4b3c4931c601", "id":  
"12722a5d-58f3-457c-ad5e-348b230c6f6a}", "value": ["s", "e", "l", "e", "n", "i",  
"u", "m"]}  
DEBUG:selenium.webdriver.remote.remote_connection:Finished Request  
DEBUG:selenium.webdriver.remote.remote_connection:POST  
http://127.0.0.1:34229/hub/session/0f0d51f5-affc-4af0-9c45-4b3c4931c601/element {"using": "id", "sessionId": "0f0d51f5-affc-4af0-9c45-4b3c4931c601",  
"value": "su"}  
DEBUG:selenium.webdriver.remote.remote_connection:Finished Request  
DEBUG:selenium.webdriver.remote.remote_connection:POST  
http://127.0.0.1:34229/hub/session/0f0d51f5-affc-4af0-9c45-4b3c4931c601/element/{8090ac84-2d92-4b48-8320-dadcfbf15f40}/click {"sessionId":  
"0f0d51f5-affc-4af0-9c45-4b3c4931c601", "id":  
"8090ac84-2d92-4b48-8320-dadcfbf15f40"}  
DEBUG:selenium.webdriver.remote.remote_connection:Finished Request  
DEBUG:selenium.webdriver.remote.remote_connection:DELETE  
http://127.0.0.1:34229/hub/session/0f0d51f5-affc-4af0-9c45-4b3c4931c601  
{"sessionId": "0f0d51f5-affc-4af0-9c45-4b3c4931c601"}  
DEBUG:selenium.webdriver.remote.remote_connection:Finished Request
```

总结:

虽然，本章花了20小节来讲元素的定位与操作，对于碰到的一些常见功能，如何通过技巧来定位它们，但是在实际的自动化脚本开发中，不管是新手还是具有一定经验的老手，所遇到最多的问题仍然是元素的定位。

有时元素定位非常简单，例如，我们只要知道这个元素有的 id 和 name 就可以轻松的来定位到它；有时元素的定位却非常的令人头疼，尽管我们想尽了办法，仍然无法定位到它。在这里笔者也没万能的方法来帮你解决这些实际问题。

对于不同的 web 项目，所用到的前端技术也不同，有些项目会用到 EXT（一个强在的 js 类库），有些会用到 AJAX（一种创建交互式网页应用的网页开发技术），这些技术的应用无疑对于前端开发人员可以快速的生成所需要的页面，但对于 UI 自动化测试人员来说，增加了定位页面元素的难度。

所以，在进行项目实现 UI 自动化评估的时候，页面元素的定位难度也是一个评估标准，如果处处都是很难定位的元素，那么无疑会增加脚本的开发与维护的成本，得不偿失。这个时候我可以考虑将更新多的精力放在单元或接口层的自动化上。

对于自动化测试人员来说，如果熟悉前端技术也会大大降低你定位元素的难度，熟练使用 XPath 和 CSS 技术会使你的定位变得容易很多，如果精通 javascript、jquery 等技术，那么使你的定位之路变得更加随心所欲。

在我们尝试开展自动化的 web 项目中，大多数在设计初期并没有考虑是否易于进行自动化，所以更多的会以实现功能为目的，这个也是后期开展自动化困难重重的重要原因。如果开发人员在设计代码的时候

就考虑是否容易自动化，为必要的元素加上 id 和 name 属性的话，那么我们自动化工作会变得容易很多。

测试人员如何更顺利的实施自动化测试工作。一方面要努力学好技术，克服技术难题。另一方面，我们要清楚的认识到，自动化技术的应用与实践不是一个人的战斗。一定要得到整个团队的配合与支持。

当然，站在公司的立场，不能带来收益的事情是很难得到支持的，这个就需要读者去综合评估目前的产品真的是否适合引入自动化，或者目前的阶段是否真的迫切需要开展自动化。

假如，你已经动手开始进行自动化了，笔者再提几点建议。

- 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 Selenium Python Bindings，及时查阅 WebDriver 所提供的方法。
- 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5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，线性测试、模块驱动测试、数据驱动测试、关键字驱动测试。本章就带领读者了解这几种测试模型。

5.1 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。

5.1.1 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。



图 5.1 线性测试结构

通过上面的图中的脚本，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。

5.1.2 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。

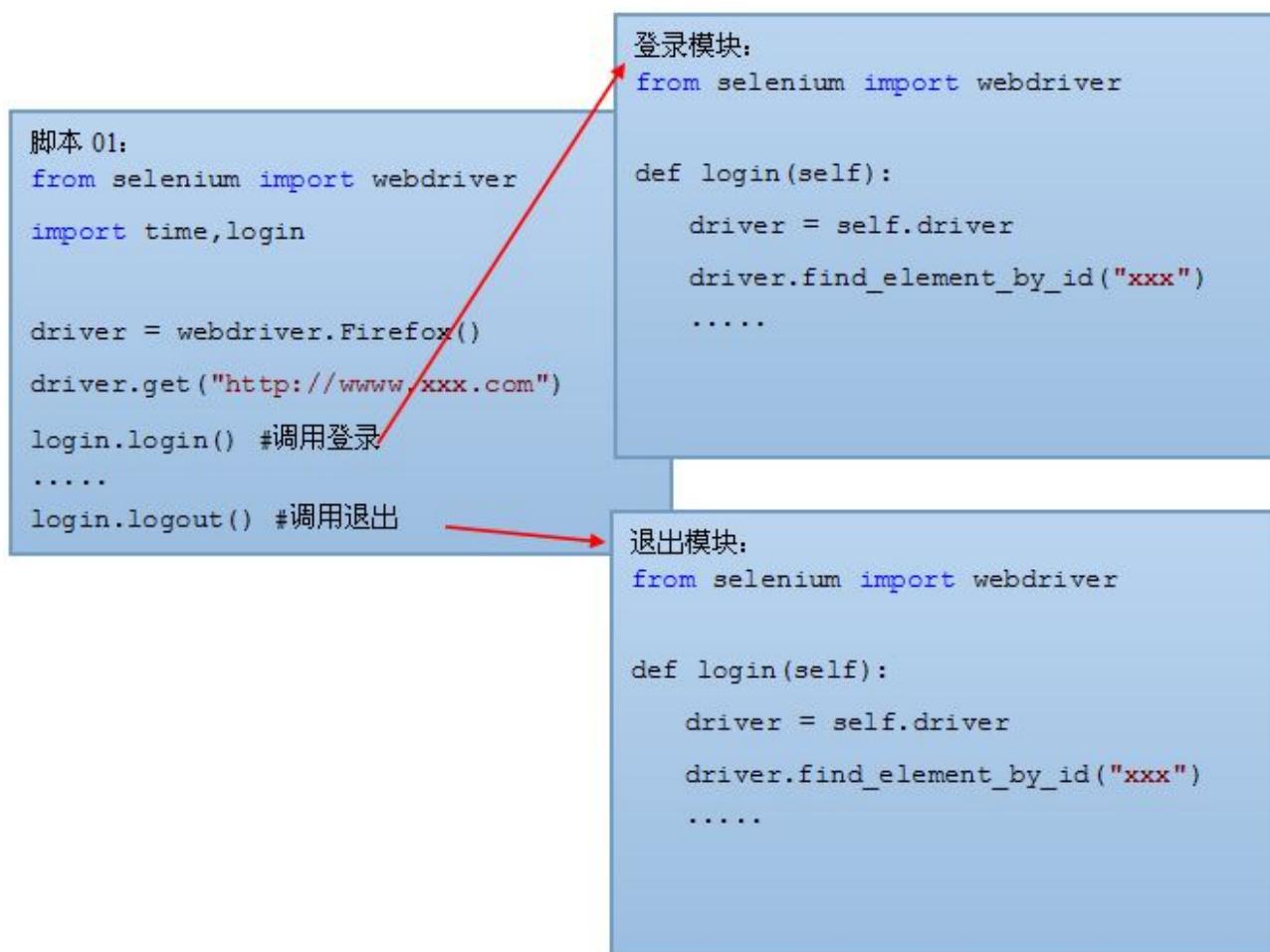


图 5.2 模块化结构

通过上面的代码结构发现，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：

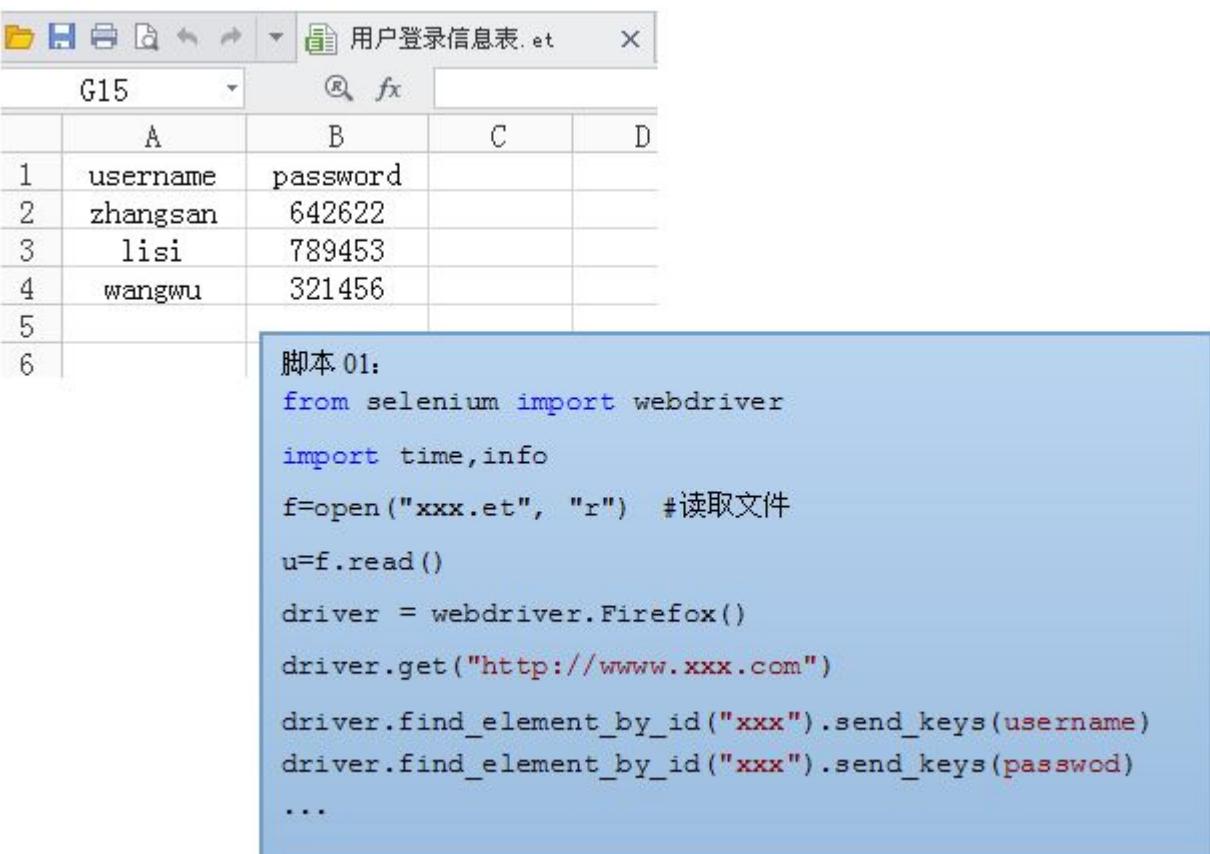
一方面提高了开发效率，不用重复的编写相同的脚本；假如，我已经写好一个登录模块，我后续需要

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。

5.1.3 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。这显然是一个非常“高级”的概念和想法。其实，我们可以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。



	A	B	C	D
1	username	password		
2	zhangsan	642622		
3	lisi	789453		
4	wangwu	321456		
5				
6				

```

脚本 01:
from selenium import webdriver
import time,info
f=open ("xxx.et", "r") #读取文件
u=f.read()
driver = webdriver.Firefox()
driver.get ("http://www.xxxx.com")
driver.find_element_by_id ("xxx").send_keys (username)
driver.find_element_by_id ("xxx").send_keys (passwod)
...

```

图 5.3 通过脚本读取数据文件

不管我们读取的是数组、字典，又或者是 excel/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。对于同段脚本来说，由于我们传入了 100 条数据，那么会真对这 100 条数据返回对应的 100 结果。

同样的脚本执行不同的数据从而得到了不同的结果，这样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。

5.1.4 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。

目前市面上典型关键字驱动工具以 QTP（目前已更名为 UFT--Unified Functional Testing）、Robot Framework 工具为主。因为这工具类封装了底层的代码，呈现给用户的是带图形界面形式，以“填表格”的形式避免测试人员对写代码的恐惧，从而降低脚本编写难度，我们只要工具所提供的关键字以“过程式”的方式来编写用例即可。

当然，我们的 Selenium IDE 也可以看做是一种传统的关键字驱动的自动化工具。

Baidu Test Case (Selenium IDE)		
open	http://www.baidu.com	
type	id=kw	selenium
click	id=su	

上面的脚本由 Selenium IDE 录制并保存的一种形式。当然，QTP 和 Robot Framework 形式也与之类似。这样的编写脚本的方式虽然比写代码简便了很多，但它又回到了我们线性测试的阶段，当用例从 1 变成 1000 的时候，维护成本会有重级的增长。

当然，关键字驱动技术也在不断发展和进步，以 Robot Framework 为例，它也可以像编程一样写测试用例。

if 分支语句：

If (Robot Framework)				
\${a}	Set variable	2		
\${b}	Set variable	5		
run keyword if	\${a}>=1	log	a 大于 1	
...	ELSE IF	\${b}<=5	log	b 小于等于 5
...	ELSE	log	上面两个条件都不满足	

定位 a、b 两个变量，分别赋值 2 和 5，首先通过 run keyword if 判断 a 是否大于等于 1，如果是通过 log 输出“a 大于 1”。否则接着判断 b 是否小于等于 5，如果是通过 log 输出“b 小于等于 5”。如果上面两个条件都不满足，log 输出“上面两个条件都不满足”

for 循环：

For (Robot Framework)			
:FOR	\${i}	in range	10
	log	\${i}	

这个例子很好理解，for 循环 i 从到 1 到 10，每循环一次通过 log 输出 i 的值。

读取外部文件：

ReadFile (Robot Framework)	
Import Resource	\${CURDIR}/resource.txt
Import Resource	\${CURDIR}/../resources/resource.html

通过 import resource 关键字读取指定的文件。

import 引入外部类库：

Import (Robot Framework)				
Import Library	MyLibrary			
Import Library	`\${CURDIR}/Libaray.py`	esom	args	
Import Library	`\${CURDIR}/.../libs/Lib.java`	arg	WITH NAME	JavaLib

通过有 Import Libray 关键字引入外部文件。

关键字驱动也可以像写代码一样写用例，在编程的世界中，没有什么不能做的，不过这样的用例同样需要一定的学习成本，这样的学习成本与学习一门编程语言几乎相当。不过这样的框架越到后期越难维护，可靠性也会变差，关键字的用途被局限在自己的框架内，你所学习的知识也很难重用到其它的测试代码的编写中。所以，对于测试人员的经验积累与发展来讲，笔者更建议通过编程的方式开发自动化脚本。

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。

5.2 模块化实例

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 126 邮箱为例：

mail126.py

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

driver = webdriver.Firefox()
driver.implicitly_wait(10)
driver.get("http://www.126.com")
#登陆
driver.find_element_by_id("idInput").clear()
driver.find_element_by_id("idInput").send_keys("username")
driver.find_element_by_id("pwdInput").clear()
driver.find_element_by_id("pwdInput").send_keys("password")
driver.find_element_by_id("loginBtn").click()

#收信、写信、删除信件等操作
.....
#退出
driver.find_element_by_link_text(u"退出").click()
```

```
driver.quit()
```

从 126 邮箱业务流程分析，所有的邮箱功能需要登录用户登录，如收信、写信、删除信件等操作。对后功能测试用例来说，测试人员在执行的过程中可以一次登录验证所有的功能之后退出，但自动化测试要保持用例之间的独立性，所以每一条自动化测试用例都需要登录和退出操作。那么，我们就可以把用例抽象为公共函数，那么对于每一条用例当需要登录/退出时，只需要调用公共函数即可，降低了代码的重复编写。从另一个角度来讲，哪果登录和退出的定位发生了变化，那么我们只用修改公共函数即可，省去了对每一个用例中登录/退出脚本修改的工作量。

下面对登录和退出进行模块的封装。

mail126.py

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

#登陆
def login():
    driver.find_element_by_id("idInput").clear()
    driver.find_element_by_id("idInput").send_keys("username")
    driver.find_element_by_id("pwdInput").clear()
    driver.find_element_by_id("pwdInput").send_keys("password")
    driver.find_element_by_id("loginBtn").click()

#退出
def logout():
    driver.find_element_by_link_text(u"退出").click()
    driver.quit()

driver = webdriver.Firefox()
driver.implicitly_wait(10)
driver.get("http://www.126.com")
login() #调用登陆模块

#收信、写信、删除信件等操作

logout() #调用退出模块
```

现在我们将登录的操作步骤封装到 login() 函数中，把退出的操作封装到 logout() 函数中，对于用例本身只用调用这两个函数即可，可以把更多的注意力放到本例本身的操作步骤中。

当然，如果只是把步骤封装成函数并没简便太多，我们需要将其放到单独的文件中供其它用例调用。

public.py

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

```
#登陆
def login(driver):
    driver.find_element_by_id("idInput").clear()
    driver.find_element_by_id("idInput").send_keys("username")
    driver.find_element_by_id("pwdInput").clear()
    driver.find_element_by_id("pwdInput").send_keys("password")
    driver.find_element_by_id("loginBtn").click()

#退出
def logout(driver):
    driver.find_element_by_link_text(u"退出").click()
    driver.quit()
```

当函数被独立到单独的文件中时做了一点调整，主要在函数的传参上，因为函数内部的操作需要使用 driver，但是 driver 并没有在此文件中定义，所以需要调用的用例传递 driver 给调用的函数。

mail126.py

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

driver = webdriver.Firefox()
driver.implicitly_wait(10)
driver.get("http://www.126.com")

#调用登陆模块
public.login(driver)

#收信、写信、删除信件等操作

#调用退出模块
public.logout(driver)
```

首先，需要导入当前目录中的 public.py，在需要的位置调用文件中的 login() 和 logout() 函数。这样对于每个用例来说就简便了许多，也更易于维护。

5.3 数据驱动实例

这一小节我们就通过一些例子来展示数据驱动在自动化测试中的应用。

5.3.1 126 邮箱登录

我们同样有以 126 邮箱的登录为例了，现在我们的需求是测试登录。那么在测试登录的用例中我们需要通过不同的用户名与密码进行验证。对于测试脚本来说，不变的是登录的步骤，变化的是每次所输入的用户名和密码不同，这种情况下就需求用到数据驱动方式来完成这个需求。

mail126.py

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

driver = webdriver.Firefox()
driver.implicitly_wait(10)
driver.get("http://www.126.com")

class Account(object):
    """docstring for Account"""
    def __init__(self,username = '', password = ''):
        self.username = username
        self.password = password

    def do_login_as(user_info):
        driver.find_element_by_id("idInput").clear()
        driver.find_element_by_id("idInput").send_keys(user_info.username)
        driver.find_element_by_id("pwdInput").clear()
        driver.find_element_by_id("pwdInput").send_keys(user_info.password)
        driver.find_element_by_id("loginBtn").click()

#实例化登陆信息
admin = Account(username='admin',password='123')
guset = Account(username='guset',password='321')

#调用登陆函数
do_login_as(admin)
do_login_as(guset)
```

首先创建表 Account 类，对用户名密码进行初始化设置，紧接着创建 do_login_as() 函数用于实现用户的登录操作，它需要一个 user_info 参数用于接收用户的登录信息。取 user_info 中的 username 输入到用户名输入框，取 user_info 中的 password 输入密码输入框。

紧接着下面的操作就是通过调用 Account 实例化用户 admin 和 guset，进行个性化的参数设置。最后分别调用 do_login_as() 函数来实现不同用户的登录。

5.3.2 百度搜索

如果 126 邮箱登录的例子对于理解数据驱动概念不够清晰，那么我们再看一个百度搜索的例子，对于这个例子前面已经出现过多次。我们每个上网的人每天都要用很多次百度，那么我们每一次使用的步骤都是一样的，不一样的是每一次搜索的“关键字”不同，从而会得到不同的搜索结果。

info.txt

```
selenium
webdriver
Python
```

创建 info.txt 文件，每一行写上需要搜索的“关键字”。

baidu.py

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

file_info = open('info.txt','r')
values = file_info.readlines()
file_info.close()

for serch in values:
    driver = webdriver.Firefox()
    driver.implicitly_wait(10)
    driver.get("http://www.baidu.com")
    driver.find_element_by_id('kw').send_keys(serch)
    driver.find_element_by_id('su').click()
    driver.quit()
```

首先通过 open() 方法以读（‘r’）的方式打开当前目录下的 info.txt 文件，通过 readlines() 获取文件中所有行的数据，并赋值给变量 values。通过 close() 关闭文件。

接下来的步骤就是循环的执行百度搜索的脚本，每一次取 values 中的一行数据做为“搜索关键字”传递给百度输入框。通过这个例子更充分的体现有数据驱动的概念。

5.3.3 读取 txt 文件

上面的例子中我们已经使用到了 txt 文件的读取。Python 提供了以下几种读取文件的方式。

- read() 读取整个文件。
- readline() 读取一行数据。
- readlines() 读取所有行的数据。

readlines() 方法我们通过上面的列子已经了解了它使用，那么问题来了，现在有一个新需求需求第次读取一个用户名和一个密码，上面所提供的几个读取文件数据的方法并不能解决这个需求。这个需求如

何通过读取 txt 文件实现呢？看下面的例子。

user_info.txt

```
zhangsan,123
lisi,456
wangwu,789
```

首先在数据的存放上做了调整，每一行存放用户名和密码，它们之间用逗号隔开。

user_info.txt

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

user_file = open('user_info.txt', 'r')
values = user_file.readlines()
user_file.close()

for serch in values:
    username = serch.split(',') [0]
    print username
    password = serch.split(',') [1]
    print password
```

运行结果：

```
>>> ===== RESTART =====
>>> zhangsan
123

lisi
456

wangwu
678
```

在读取 user_info.txt 文件的时候，同样使用 readlines() 方法读取所有行的数据，那么获取到的一行数据如何拆分出用户名和密码是关键，这里我们使用 split() 进行拆分，它可将一个字符串通过某一符号拆分成左右两部分，这里逗号（，）为分割点。split() 拆分出来的左右两部分以数据的形式存放，所用 [0] 可以取到左半部分的字符串，[1] 可以取到右半部分的字符串。

这样就可以方便有使用 txt 文件来存放用户名和密码数据了，或者其它必须成对调用的数据。

5.3.4 读取 csv 文件

那么新的问题来了，假如，现在要读取的是一组用户数据，这一组数据包括用户名、邮箱、年龄、性

别等信息。这个时候再使用 `aplit()` 方法拆分就可不那么方便了，因为它一次只能将字符串拆分成左右两部分。

下面通过读取 csv 文件的方法来解决这个每次要读取多个信息的问题。

首先创建 `userinfo.csv` 文件，通过 WPS 表格 或 Excel 创建表格，文件另存为选择 CSV 格式进行保存，注意不要直接修改 Excel 的后缀名来创建 CSV 文件，这样创建出来的并非真正的 CSV 文件。

	A	B	C	D	E
1	testing	123456@126.com	23	man	
2	testing2	123456@qq.com	18	woman	
3	testing3	123456@128.com	29	woman	
4					
5					

图 5.4 csv 文件

下面修改 `loop_reader.py` 文件进行循环读取：

user_info.txt

```
#coding=utf-8
import csv #导入 csv 包

#读取本地 CSV 文件
my_file='info.csv'
date=csv.reader(file(my_file,'rb'))

#循环输出每一行信息
for user in date:
    print user
```

运行结果：

```
>>> ===== RESTART =====
>>>
['testing', '123456@126.com', '23', 'man']
['testing2', '123456@qq.com', '18', 'woman']
['testing3', '123456@128.com', '29', 'woman']
```

首先表导入 `csv` 模块，通过 `reader()` 方法读取 csv 文件。然后通过 `for` 循环遍历文件中的每一行数据。

从打印结果可以看出是以数组的形式存的。那么如果想取用户的某一列信息，只需要指定下标即可。

user_info.txt

```
#coding=utf-8
import csv

my_file='info.csv'
date=csv.reader(file(my_file,'rb'))

#取用户的邮箱地址
for user in date:
    print user[1]
```

运行结果：

```
>>> ===== RESTART =====
>>>
123456@126.com
123456@qq.com
123456@128.com
```

在上面的例子中只需要取所有用户的邮箱地址，那么只需要指定邮箱地址的所在例即可。数组下标是以 0 开始的，取第二例的信息下标为 1。

通过这种方式就解决读取多列数据的问题，在本中是以读取 CSV 文件为例，读取 Excel 文件的方式也类似，只是所调用的模块就需要从 csv 切换为 xlrd，真对 Excel 文件操作的方法也有所不同。

5.3.4 读取 xml 文件

在时间我们所需要读取的文件并有固定的行和例，而是一些不规则的配置信息，例如我们需要一个配置文件来配置当前自动化测试脚本的 URL、浏览器、登录用户名/密码等。这个时候可以选择 XML 文件来配置这些信息。

什么是 XML？

XML 即可扩展标记语言，它可以用来标记数据、定义数据类型，是一种允许用户对自己的标记语言进行定义的源语言。

xml 有如下特征：

首先，它是有标签对组成：<aa></aa>

标签可以有属性： <aa id='123'></aa>

标签对可以嵌入数据： <aa>abc</aa>

标签可以嵌入子标签（具有层级关系）：

```
<aa>
  <bb></bb>
</aa>
```

下面以读取 info.xml 文件为例介绍读取 XML 文件的方法。

info.xml

```
<?xml version="1.0" encoding="utf-8"?>
<catalog>
  <maxid>4</maxid>
  <login username="pytest" passwd="123456">
    <caption>Python</caption>
    <item id="4">
      <caption>test</caption>
    </item>
  </login>
  <item id="2">
    <caption>Zope</caption>
  </item>
</catalog>
```

获得标签信息

read_xml.py

```
#coding=utf-8
import xml.dom.minidom

#打开 xml 文档
dom = xml.dom.minidom.parse('info.xml')

#得到文档元素对象
root = dom.documentElement
print root.nodeName
print root.nodeValue
print root.nodeType
print root.ELEMENT_NODE
```

运行结果：

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

```
>>>  
catalog  
None  
1
```

```
xml.dom.minidom
```

xml.dom.minidom 模块被用来处理 xml 文件，所以要先引入。

```
parse()
```

xml.dom.minidom.parse() 用于打开一个 xml 文件，并将这个文件对象 dom 变量。

```
documentElement
```

documentElement 用于得到 dom 对象的文档元素，并把获得的对象给 root

每一个结点都有它的 nodeName, nodeValue, nodeType 属性。

nodeName 为结点名字。

nodeValue 是结点的值，只对文本结点有效。

nodeType 是结点的类型。

获得任意标签名

```
read_xml.py
```

```
#coding=utf-8  
import xml.dom.minidom  
  
#打开 xml 文档  
dom = xml.dom.minidom.parse('info.xml')  
  
#得到文档元素对象  
root = dom.documentElement  
  
tagname = root.getElementsByTagName('maxid')  
print tagname[0].tagName  
  
tagname = root.getElementsByTagName('caption')  
print tagname[2].tagName  
  
tagname = root.getElementsByTagName('item')  
print tagname[1].tagName
```

运行结果：

```
>>> ===== RESTART =====
>>>
maxid
caption
item
```

getElementsByTagName() 可以通过标签名获取某个标签。它所获取的对象是以数组形式存放。如“caption”和“item”标签在 info.xml 文件中有多个，那么可以指定数组的下标在获取某个标签。

root.getElementsByTagName('caption') 获得的是标签为 caption 一组标签；

tagname[0] 表示一组标签中的第一个。

tagname[2] 表示一组标签中的第三个。

获得标签的属性值

read_xml.py

```
#coding=utf-8
import xml.dom.minidom

#打开 xml 文档
dom = xml.dom.minidom.parse('info.xml')

#得到文档元素对象
root = dom.documentElement

logins = root.getElementsByTagName('login')

#获得 login 标签的 username 属性值
username=logins[0].getAttribute("username")
print username

#获得 login 标签的 passwd 属性值
password=logins[0].getAttribute("passwd")
print password

items = root.getElementsByTagName('item')
#获得第一个 item 标签有 id 属性值
id1=items[0].getAttribute("id")
print id1

#获得第二个 item 标签有 id 属性值
id2=items[1].getAttribute("id")
```

```
print id2
```

运行结果：

```
>>> ===== RESTART =====  
>>>  
pytest  
123456  
4  
2
```

```
getAttribute()
```

getAttribute()方法可以获得元素的属性所对应的值。

获得标签对之间的数据

read_xml.py

```
#coding=utf-8  
import xml.dom.minidom  
  
#打开 xml 文档  
dom = xml.dom.minidom.parse('info.xml')  
  
#得到文档元素对象  
root = dom.documentElement  
  
captions=dom.getElementsByTagName('caption')  
  
#获得第一个标签对的值  
c1=captions[0].firstChild.data  
print c1  
  
#获得第二个标签对的值  
c2=captions[1].firstChild.data  
print c2  
  
#获得第三个标签对的值  
c3=captions[2].firstChild.data  
print c3
```

运行结果：

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

Python
test
Zope

firstChild. data

firstChild 属性返回被选节点的第一个子节点， data 表示获取该节点的数据。

小结：

在本章的学习中，我们首先介绍了几种自动化测试模型，然后通过实例介绍了模块化的应用。在数据驱动的小节里，分别介绍了 txt、csv 和 xml 三种文件的读取，读者可以根据需求选择这三种文件来存放数据。

模块化和数据驱动在脚本开发过程是必不可少的两个知识点，这也是开发出可复用和可维护的脚本的基础，希望读者灵活运用。

当然，除了前面介绍的几种测试模型外，Page Object Model 也是逐渐流行起来的一种设计模式。我们将在后面的章节中重点讨论这种设计模式。

第6章 Selenium IDE

相信有不少读者学习 Selenium 是从 Selenium IDE 开始的，做为嵌入在 Firefox 浏览器的一个小插件，结合浏览器提供了脚本的录制、回放以及编辑脚本功能，可以帮助我们快速理解和学习自动化测试。

按照本书的作者的目的是帮助读者开发自动化测试脚本，不应用把精力放这个小插件的学习，但事实上对 Selenium IDE 做为 Selenium 的成员之一，任何一本讲 Selenium 的书都不能忽略 Selenium IDE，作者从另一个重要角度考虑，对于 Selenium IDE 学习，有助于我们后面学习单元测试框架，以及如何在脚本中添加断言和验证等。所以，这一章我们来学系统的了解 Selenium 的使用。

6.1 Selenium IDE 安装

Selenium IDE 的安装在早前非常简单有，打开 FireFox 浏览器，选择菜单栏“工具”→“附加组件”，然后搜索“Selenium IDE”，从搜索结果中点击 Selenium IDE 对应的“安装”按钮进行安装，安装完成重启浏览器即可。但笔者验证 FireFox 的“附加组件”中已经搜索不到 Selenium IDE 了，那么我们只能通过 Selenium 官方网站进行安装。

下面介绍两种安装方式。

6.1.1 在线安装

通过 FireFox 浏览器访问 selenium 下载页面：

<http://docs.seleniumhq.org/download/>

在页面中找到 Selenium IDE 介绍，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 Selenium IDE 介绍

FireFox 浏览器将自动识别需要下载的 selenium IED 插件，如图 6.2，点击“立刻安装”按钮进行下载安装。

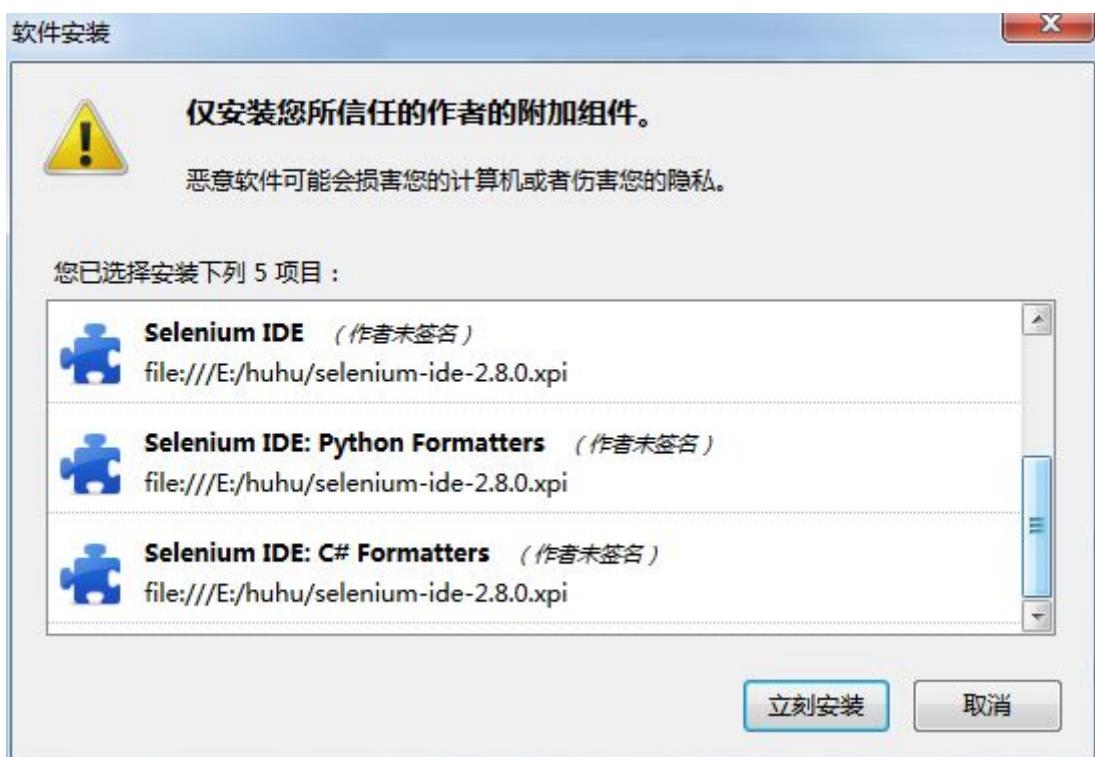


图 6.2 Firefox 浏览器识别 Selenium IDE 插件

安装完成后重启 firefox 浏览器，通过菜单栏“工具”---> selenium IDE 打开，或通过 Ctrl+Alt+S 快捷键打开。如图 6.3。

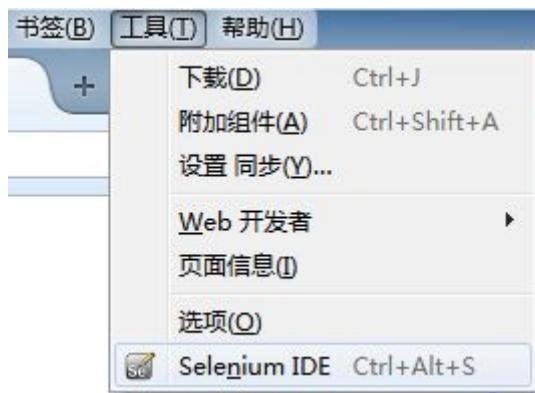


图 6.3 打开 Selenium IDE

6.1.2 通过添加插件安装

通过添加插件的安装方式与前面一种类似，如图 6.1，如果我们使用的非 FireFox 浏览器点击 Selenium IDE 的版本号链接，那么将会提示下 Selenium IDE 下载，下载完成将得到一个 selenium-ide-2. x. x. xpi 的文件。

打开 FireFox 浏览器，选择菜单栏“工具”-->“附加组件”，点击附加组件搜索框左侧的小齿轮按钮，选择“从文件安装附加组件...”，如图 6.4。



图 6.4 FireFox 添加本地附加组件

弹出本地文件选择框，选择 selenium-ide-2.x.x.xpi 文件确定，将会弹出图 6.2 的窗口，点击“立刻安装”将会进行安装，安装完成重启浏览器即可。

6.2 Selenium IDE 界面介绍

打开 Selenium IDE 如图 6.5。

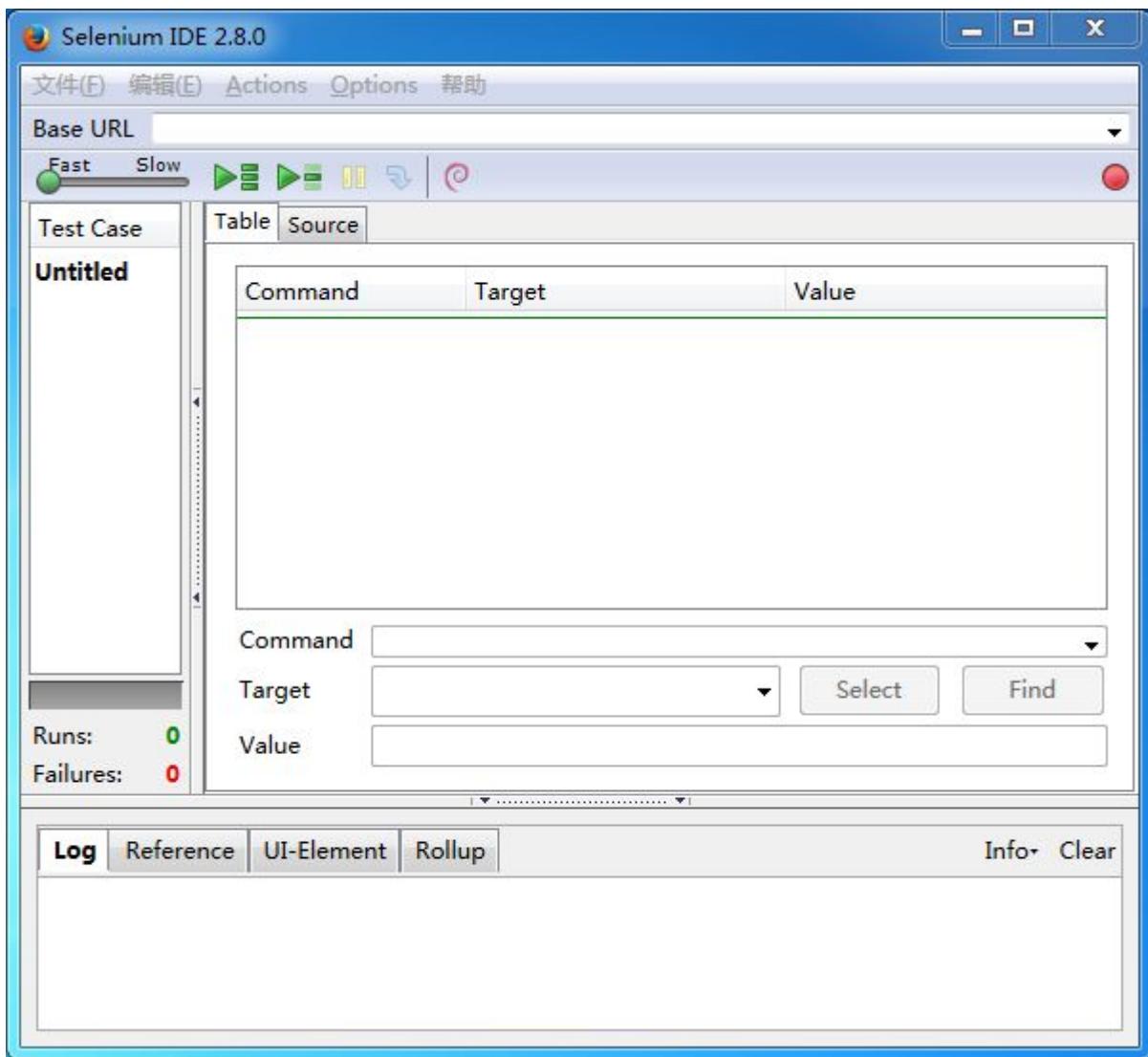


图 6.5 Selenium IDE 窗口

Selenium IDE 界面介绍：

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。

Actions (行为)：设置的脚本的录制与运行。

Options (设置)：用于设置 selenium IDE。

2、Base URL：用来填写被测试的基础 URL 地址。



图 6.5 Selenium IDE 窗口

3、速度控制：控制案例的运行速度。滑动按钮拖到 Fast 侧用例将快速执行，相反拖动到 Slow 侧缓慢执行。

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、Test Case 表示案例集列表。
 - 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、Runs/Failures: 查看脚本运行通过/失败的个数。Runs 表示用例执行成功的个数。Failures 表示用例失败的个数。
 - 12、当选中前命令对应参数。一条命令由 command、Target、value 三个部分组成。
 - 13、Log/Reference/UI-Element/Rollup

Log: 当你运行测试时，错误和信息将会自定显示。

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

6.3 创建测试用例

6.3.1 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 6.6

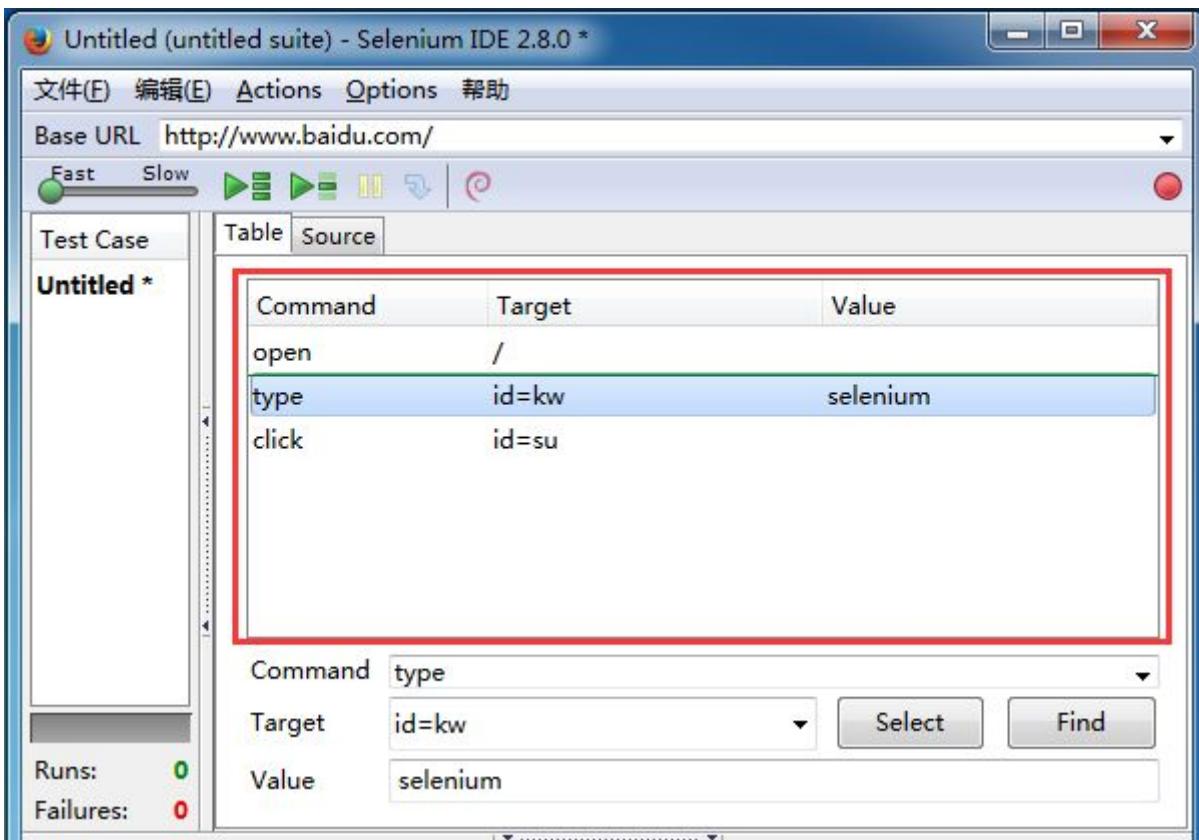


图 6.6 Selenium IDE 录制脚本

6.3.2 编辑脚本

selenium IDE 为我们录制的脚本不是百分百符合我们的需求的，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。

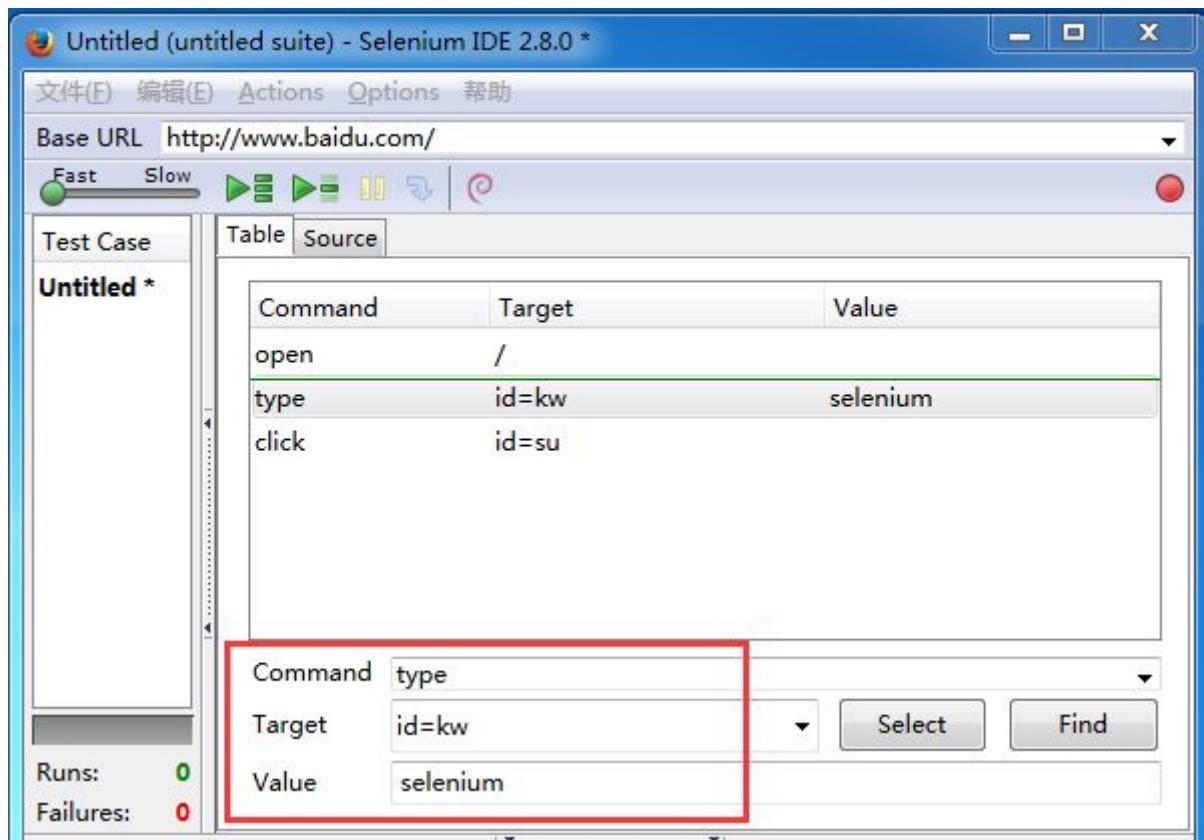


图 6.7 Selenium IDE 编辑脚本

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，然后对空白行进行编辑。

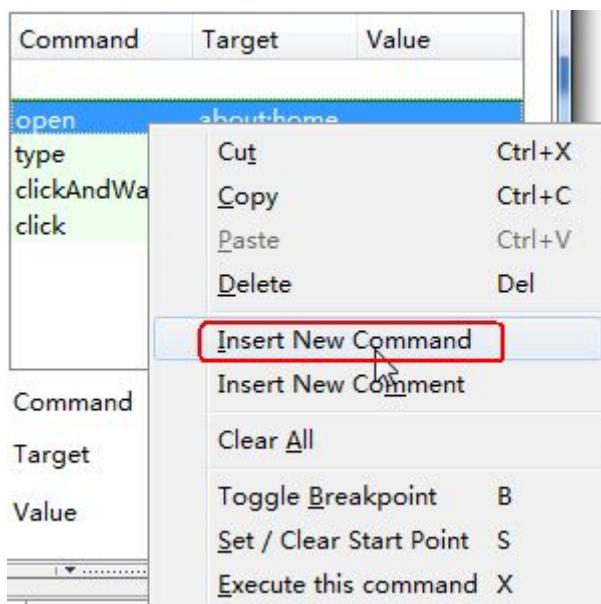


图 6.8 插入一条命令

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。



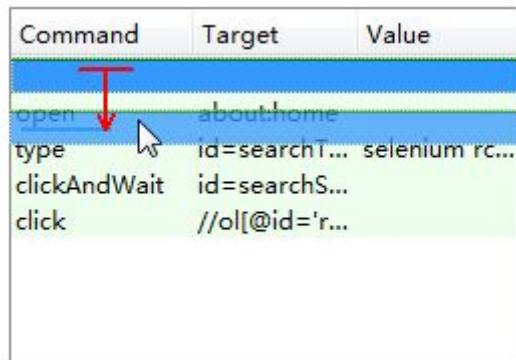
Command	Target	Value
谷歌网站		
open	about:home	
type	id=searchT... selenium rc...	
clickAndWait	id=searchS...	
click	//ol[@id='r...	

Command: ▼
 Target: Find
 Value:

图 6.10 添加一条注释

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 6.11 移动元素

5. 定位辅助

当 selenium IDE 录制脚本时，Targetg 会生成针对当前元素的所有定位方式，我们可以点击 Target 输入框右侧的下拉框选择其他定位方式。

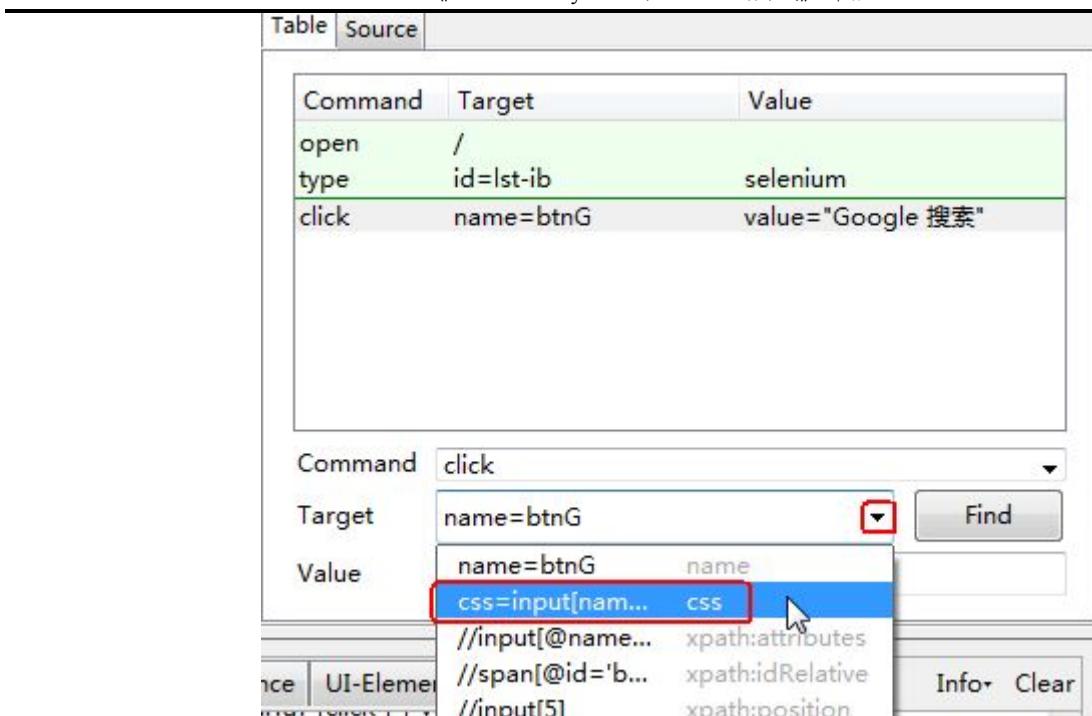


图 6.12 选择元素定位方式

6.4 Selenium IDE 命令

在 Selenium IDE 中提供了大量的命令，在 Selenium IDE 的 Command 的下拉列表框中可以选择使用这些命令，如图6.13。

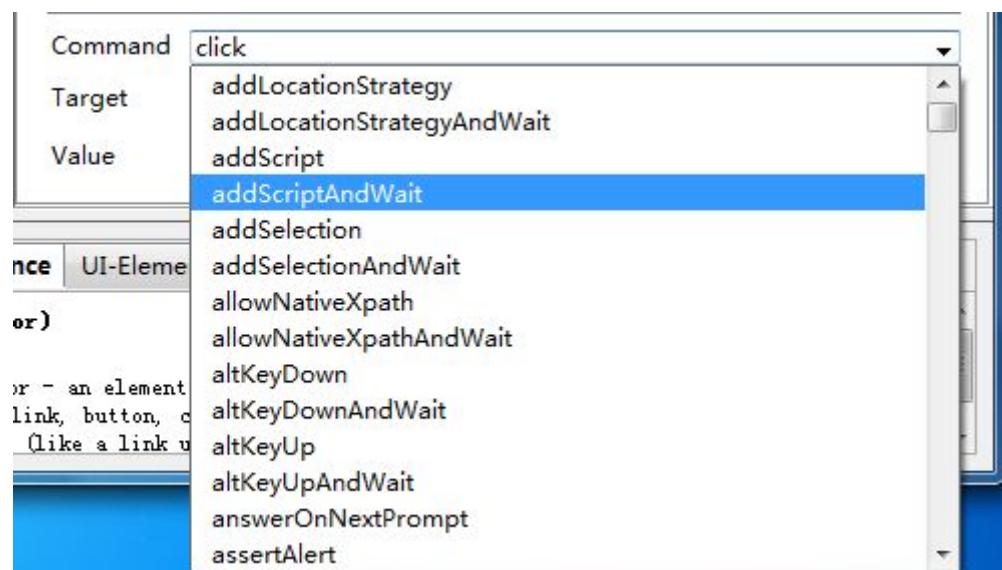


图 6.13 Selenium IDE 命令

下面我们介绍一些常用 Action 命令的使用：

Action 有两种形式: action 和 actionAndWait, action 会立即执行, 而 actionAndWait 会假设需要较长时间才能得到该 action 的响应, 而作出等待, open 则是会自动处理等待时间。

open

open(url)

- 在浏览器中打开 URL, 可以接受相对和绝对路径两种形式。
- 注意: 该 URL 必须在与浏览器相同的安全限定范围之内。

Command	Target	Value
open	/mypage	
open	http://localhost/	

click

click(elementLocator)

- 点击连接, 按钮, 复选和单选框。
- 如果点击后需要等待响应, 则用"clickAndWait"。
- 如果是需要经过 JavaScript 的 alert 或 confirm 对话框后才能继续操作, 则需要调用 verify 或 assert 来告诉 Selenium 你期望对对话框进行什么操作。

Command	Target	Value
click	aCheckbox	
clickAndWait	submitButton	
clickAndWait	anyLink	

type

type(inputLocator, value)

- 模拟人手的输入过程, 往指定的 input 中输入值。
- 也适合给复选和单选框赋值。
- 在这个例子中, 则只是给勾选了的复选框赋值, 注意, 而不是改写其文本。

Command	Target	Value
type	nameField	John Smith
typeAndWait	textBoxThatSubmitsOnChange	newValue

select

select(dropDownLocator, optionSpecifier)

- 根据 optionSpecifier 选项选择器来选择一个下拉菜单选项。

- 如果有多于一个选择器的时候，如在用通配符模式，如“f*b*”，或者超过一个选项有相同的文本或值，则会选择第一个匹配到的值。

Command	Target	Value
select	dropDown	Australian Dollars
select	dropDown	index=0
selectAndWait	currencySelector	value=AUD
selectAndWait	currencySelector	label=Auslian D*rs

goBack

goBack()

模拟点击浏览器的后退按钮。

Command	Target	Value
goBack		

selectWindow

select(windowId)

- 选择一个弹出窗口。

- 当选中那个窗口的时候，所有的命令将会转移到那窗口中执行

Command	Target	Value
selectWindow	myPopupWindow	
selectWindow	null	

pause

pause(milliseconds)

- 根据指定时间暂停 Selenium 脚本执行。

- 常用在调试脚本或等待服务器段响应时。

Command	Target	Value
pause	5000	
pause	2000	

fireEvent

fireEvent(elementLocatore, evenName)

- 模拟页面元素事件被激活的处理动作。

Command	Target	Value
fireEvent	textField	focus
fireEvent	dropDown	blur

close

close()

模拟点击浏览器关闭按钮。

Command	Target	Value
close		

6.5 断言与验证

对于测试用例来说需要预期结果，在测试的过程中需要通过实际结果与预期结果进行比较才知道用例是否成果。在 Selenium 中提供了断言与验证来对结果进行比较。

如何像 Selenium IDE 的脚本中添加额断言与验证呢？首先打开 Selenium IDE，右键页面（FireFox 浏览器）上的任意元素弹出快捷菜单，选择最后一个选项“Show All Available commands”

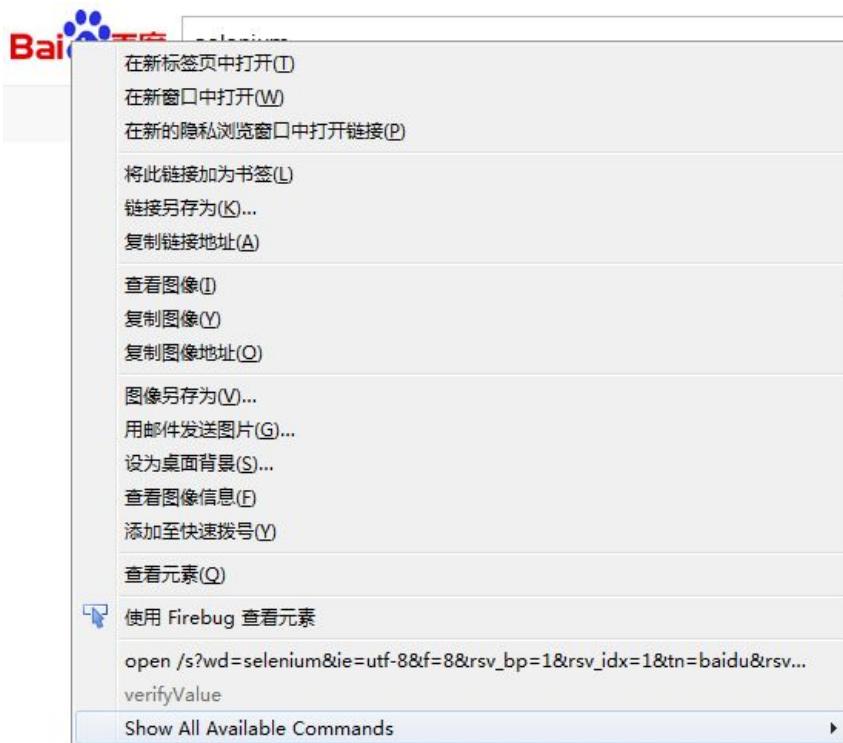


图 6.13 右击页面任意元素弹出菜单

```

open /s?wd=selenium&ie=utf-8&f=8&rsv_bp=1&rsv_idx=1&tn=baidu&rsv...
assertTitle selenium_百度搜索
assertValue
assertText css=img[alt="到百度首页"]
assertTable
assertElementPresent css=img[alt="到百度首页"]

verifyTitle selenium_百度搜索
verifyValue
verifyText css=img[alt="到百度首页"]
verifyTable
verifyElementPresent css=img[alt="到百度首页"]

waitForTitle selenium_百度搜索
waitForValue
waitForText css=img[alt="到百度首页"]
waitForTable
waitForElementPresent css=img[alt="到百度首页"]

storeTitle selenium_百度搜索
storeValue
storeText css=img[alt="到百度首页"]
storeTable
storeElementPresent css=img[alt="到百度首页"]

```

图 6.14 页面所有可用命令

通过图 6.14 的快捷菜单，大概获得了 4 类命令：assert 断言、verify 验证、waitFor 等待、store 定义变量。这四类命令又分 5 种验证手段。

Title，获取页面的标题。

Value 获得元素的值。

Text 获得元素的文本信息。

Table 获得元素的标签。

ElementPresent 获得当前元素。

6.5.1 断言

如果使用的断言，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。

通过图 6.14 可帮助我们向脚本添加断言命令，黑色的选项表示可选，灰色的选项表示不可选。

Baidu Test Case		
Command	Target	Value
open	http://www.baidu.com/	
type	id=kw	selenium ide
click	id=su	
assertTitle	selenium ide_百度搜索	
assertText	css=img[alt="到百度首页"]	
assertElementPresent	css=img[alt="到百度首页"]	
close		

在上面的测试用例中，我添加了三种验证，分别是 Title、Text 和 ElementPresent。

6.5.2 验证

相比断言，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一测试会被执行完毕，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。

同样参考图 6.14，添加验证命令。

Baidu Test Case		
Command	Target	Value
open	http://www.baidu.com/	
type	id=kw	selenium ide
click	id=su	
verifyTitle	selenium ide_百度搜索	
verifyText	css=img[alt="到百度首页"]	
verifyElementPresent	css=img[alt="到百度首页"]	
close		

什么时候使用断言命令，什么时候使用验证命令？这取决于读者。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。

Baidu Test Case		
Command	Target	Value
open	http://www.baidu.com/	

type	id=kw	selenium ide
click	id=su	
verifyTitle	selenium ide_百度搜索 sss	
type	id=kw	selenium webdriver
click	id=su	
close		

执行上面的脚本，在断言的时候，特意设置有误的验证信息“selenium ide_百度搜索 sss”使验证失败，以下 Selenium IDE 的 log 信息。

- [info] Playing test case baide_case
- [info] Executing: |open | http://www.baidu.com ||
- [info] Executing: |type | id=kw | selenium ide |
- [info] Executing: |click | id=su ||
- [info] Executing: |pause | 2000 ||
- [info] Executing: |assertTitle | selenium ide_百度搜索 sss ||
- [error] Actual value 'selenium ide_百度搜索' did not match 'selenium ide_百度搜索 sss'
- [info] Test case failed
- [info] Test suite completed: 1 played, 1 failed

当脚本执行到“assertTitle”的位置时，断言失败，脚本终止继续执行。同样是上面的一段脚本，现在把“assertTitle”替换为“verifyTitle”，再次执行脚本。查看 log 信息：

- [info] Playing test case baide_case
- [info] Executing: |open | http://www.baidu.com ||
- [info] Executing: |type | id=kw | selenium ide |
- [info] Executing: |click | id=su ||
- [info] Executing: |pause | 2000 ||
- [info] Executing: |verifyTitle | selenium ide_百度搜索 sss ||
- [error] Actual value 'selenium ide_百度搜索' did not match 'selenium ide_百度搜索 sss'
- [info] Executing: |type | id=kw | selenium webdriver |
- [info] Executing: |click | id=su ||
- [info] Executing: |close |||
- [info] Test case failed
- [info] Test suite completed: 1 played, 1 failed

当脚本执行到“verifyTitle”验证时失败，但并没有终止后面脚本的执行。

6.6 等待与变量

继续参考图 6.14 继续介绍等待（waitFor）和定义变量（store）的使用。

6.6.1 等待

在 Selenium IDE 中提供了 pause 来设置固定的休眠时间，waitFor 在一定时间内等待某一元素显示。

Baidu Test Case		
Command	Target	Value
open	http://www.baidu.com/	
type	id=kw	selenium ide
click	id=su	
waitForTitle	selenium ide_百度搜索	
waitForText	css=img[alt="到百度首页"]	
waitForElementPresent	css=img[alt="到百度首页"]	
close		

waitFor 如果 Value 为空， 默认为 60 秒。除了上面例子用到的 waitForTitle、waitForText、waitForElementPresent 外还有其它一些 waitFor 方法。

waitForCondition

waitForCondition(JavaScriptSnippet, time)

- 在限定时间内，等待一段 JavaScript 代码返回 true 值，超时则停止等待

Command	Target	Value
waitForCondition	var value=selenium.getText("foo"); value.match(/bar/);	3000

waitForValue

waitForValue(Locator, pattern)

- 等待某元素的 value(如百度搜索按钮)被赋予某值。
- 会轮流检测该值，所以要注意如果该值长时间一直不赋予该 input 该值的话，可能会导致阻塞。

Command	Target	Value
waitForValue	id=su	百度一下

6.6.2 变量

store 用于定义变量。

Baidu Test Case

Command	Target	Value
open	http://www.baidu.com/	
type	id=kw	selenium ide
click	id=su	百度一下
storeTitle	selenium ide_百度搜索	title
storeText	css=img[alt="到百度首页"]	text
storeForElementPresent	css=img[alt="到百度首页"]	element
close		

可以把页面中获取到的标题、文本信息和元素定义成变量 title、text 和 element，下面完整的用例配合断言与验证使用这些变量。

Baidu Test Case		
Command	Target	Value
open	http://www.baidu.com/	
type	id=kw	selenium ide
waitForValue	id=su	
click	id=su	百度一下
pause	2000	
storeTitle	selenium ide_百度搜索	title
storeText	css=img[alt="到百度首页"]	text
storeForElementPresent	css=img[alt="到百度首页"]	element
verifyTitle	selenium ide_百度搜索	title
verifyText	css=img[alt="到百度首页"]	text
assertElementPresent	css=img[alt="到百度首页"]	element
close		

这里例子中就综合的运行了前面所学的方法，命 store 定义实际结果进行验证与验证。下面看一下 store 的其它用方法的应用。

```
store
store(expression, variableName)
```

定义一个变量里。

该值可以由自其他变量组合而成或通过 JavaScript 表达式赋值给变量。

Command	Target	Value
store	Mr John Smith	fullname

store	<code>\$. { title } \$. { firstname } \$. { surname }</code>	fullname
store	<code>javascript. { Math.round(Math.PI*100)/100 }</code>	PI

例如，百度要搜索的关键字保存到变量中

Command	Target	Value
store	selenium ide	value
type	id=kw	<code> \${value}</code>

storeAttribute

`storeAttribute({ } elementLocator@attributeName, variableName. { })`

把指定元素的属性的值赋予给变量

Command	Target	Value
storeAttribute	input1@class	classOfInput1
verifyAttribute	input2@class	<code> \$. { classOfInput1 }</code>

小结：

通过本章的学习，我们已经基本掌握了通过 Selenium IDE 来编写自动化测试脚本。读者可能还是比较疑惑，这和我们通过 Selenium + Python 开发自动化脚本有什么联系么？那么我们掌握的 Selenium IDE 是否用这个来写自动化测试用例呢？读者当然可以使用 Selenium IDE 来写自动化测试用例。当 Selenium IDE 却有不少缺点，比如它对不支持跨域，就是从一个 URL 地址跳到另一个完全不同的 URL 地址，例如在用例呈数量级增加的时间，Selenium IDE 很难维护这些脚本。

至于与 Selenium + Python 的联系，我们可方便的将 Selenium IDE 录制的脚本导出成不同编程语言，不同测试框架，的 Selenium RC 或 WebDriver 脚本，从而帮助我们编写脚本。这个将在后面的章节介绍。

第7章 unittest 单元测试框架

对于不熟悉编程的测试人员讲，单元测试是个听起来高大上的话题，貌似只有高级测试或开发人员才能接触到这项工作，对于初、中级测试人员很少接触到这项工作。其实，它并非想象的那么困难，这一节我们揭秘单元测试面纱。

可以读者还有个疑问，我们不是在学 web 自动化么？怎么又跳单元测试上了呢？从第一章分层思想的介绍中，它们属于不同层次的测试工作呀。可没有人告诉你单元测试框架只能测试代码级别的测试用例。对于单元测试框架来，笔者认为它主完成三件事。

提供用例组织与执行：对一个功能编写几个测试用例来说，当然不同讲究用例的组织，但测试用例达到成百上千时，就需要考虑编写用例的规范与组织，否则后期的维护成本是呈现指数级增加的。可能，我们都无法顾及开发新功能，而花费大量的时间和精力在维护测试用例。这也是单元测试用例的目的。

提供比较方法：不管是功能测试用例，还是单元测试都会有一个预期结果，测试的过程其实就是执行某些操作之后拿实际的结果与预期结果进行比较，当然，这个比较有我们前面介绍来的断言与验证。

提供丰富的日志：当所有用例执行完成，要有足够清晰的日志信息告诉我哪些用例失败以及它们失败的位置，如果没这清晰的日志，那么将会大大提供 debug 问题也是一件非常耗时的工作。

一般的单元测试框架都会提供这些功能，从这些特性来看，单元测试框架当然是可以执行 web 自动化脚本的。

7.1 分析带 unittest 自动化测试脚本

你现在一定比较迷惑，如何通过 unittest 单元测试框架来组织 Selenium 自动化测试脚本，或者说如何将这两个技术组合起来做自动化测试。我们可以将 Selenium IDE 编写导出为所需要的格式，从而帮助我们理解自动化脚本的编写。

首先，通过 Selenium IDE 录制一个测试用例，选择菜单栏“文件”→“Export Test Case As...”弹出二级菜单，这里罗列 Selenium IDE 所支持导出的编程语言、测试框架、WebDriver/Remote Control，如图7.1。

Selenium IDE 所支持的导出类型：

- Ruby/RSpec/WebDriver
- Ruby/RSpec/Remote Control
- Ruby/Test::Unit/WebDriver
- Ruby/Test::Unit/Remote Control
- Python2/unittest/WebDriver
- Python2/unittest/Remote Control
- Java/Junit4/WebDriver
- Java/Junit4/WebDriver Backed
- Java/Junit4/Remote Control
- Java/Junit3/Remote Control
- Java/TestNG/Remote Control
- C#/Nunit/WebDriver

C#/Nunit/Remote Control

Selenium IDE 提供多语言与测试框架的自动化脚本生成，对于学习不同编程语言下的自动化测试脚本开发提供很好的帮助。

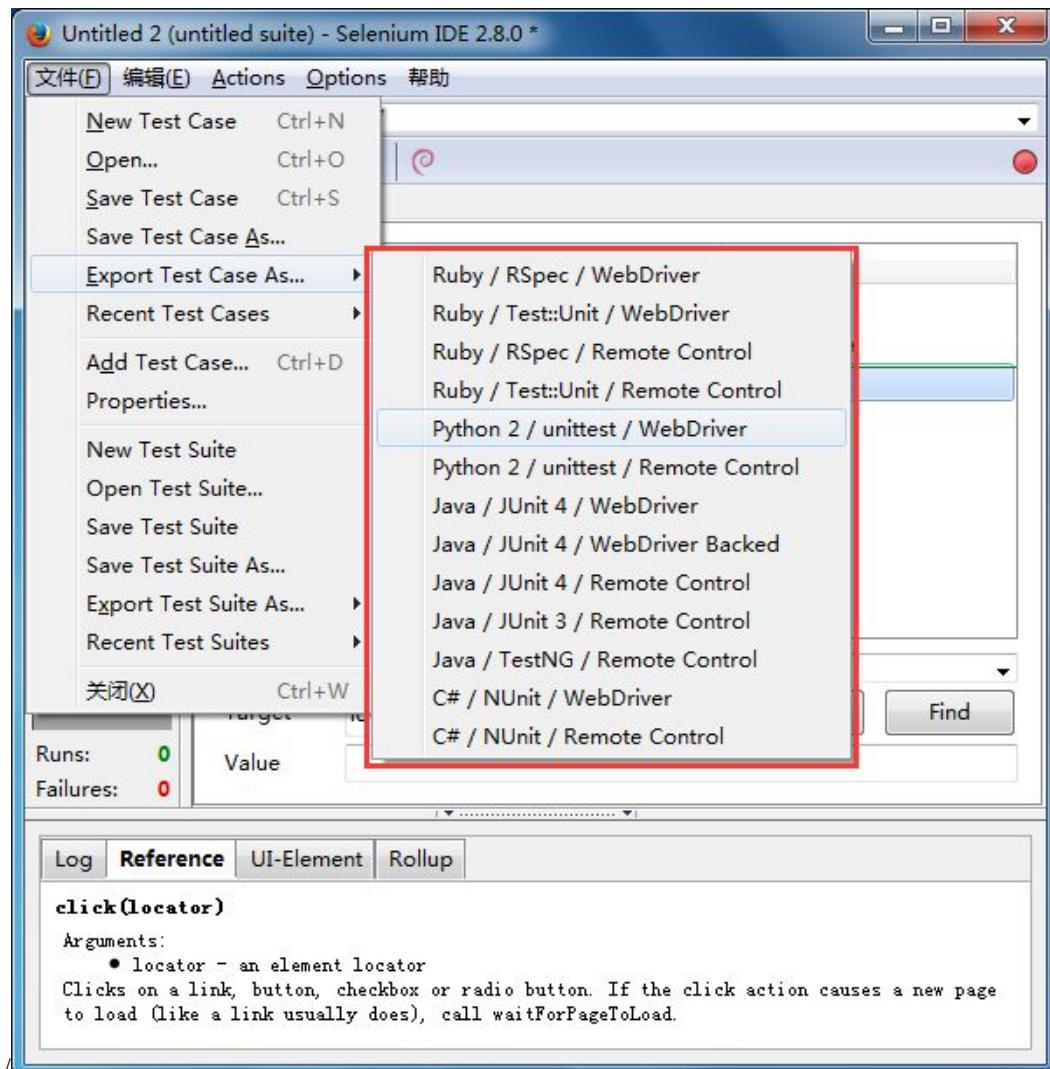


图 7.1 Selenium IDE 所支持的导出类型

因为我们当前使用的自动化测试开发语言为 Python2，单元测试框架为 unittest，自动测试工具为 WebDriver，所以选择“Python2/unittest/WebDriver”选项，将脚本保存到指定位置。

下面通过 Python 编辑器打开保存的脚本。

Python shell

```
# -*- 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
from selenium.common.exceptions import NoAlertPresentException
import unittest, time, re
```

```
class BaiduTest(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_baidu(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 ide")
        driver.find_element_by_id("su").click()

    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 单元测试框架的自动化脚本多了很多东西，而且一些方法在前面是没有接触到的，现在我们来逐一的分析这段脚本中都做了什么。

```
import unittest
```

首先要引入 unittest 框架包。

```
class BaiduTest(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.driver.implicitly_wait(30)
    self.base_url = "http://www.baidu.com/"
    selfverificationErrors = []
    self.accept_next_alert = True
```

setUp 用于设置初始化工作，在每一个测试用例前先被执行，它与 tearDown 方法相呼应，后者在每一个测试用例执行后被执行。这里的初始化工作定义了浏览器启动和基础 URL 地址。

implicitly_wait() 在前面章节已经学过，它用于设置整个页面元素的隐性等待，30 秒。

接下来定义空的 verificationErrors 数组，脚本运行时的错误信息将被打印到这个数组中。

定义 accept_next_alert 变量，表示是否继续接受下一个警告，初始化状态为 True。

```
def test_baidu(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 ide")
    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，这里就不再解释。

```
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` 方法用来查找页面元素是否存在，通过 `find_element()` 来接收元素的定位方法（how）和定位值（what），如果定位到元素返回 Ture，否则出现异常并返回 Flase。try...except... 为 Python 语言的异常处理。

```

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

```

`is_alert_present()` 方法用来处理弹出的警告框，用 WebDriver 所提供的 `switch_to_alert()` 方法来捕捉警告框。如果捕捉到警告框返回 Ture，否则异常，返回 Flase。

```

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

```

`close_alert_and_get_its_text()` 关闭警告并且获得警告信息。首先通过 `switch_to_alert()` 获得警告，通过 `.text` 获得警告框信息。接着通过 `if` 语句判断 `accept_next_alert` 的状态，在 `setUp()` 已经初始化状态为 Ture，如果为 Ture，通过 `accept()` 接受警告。否则 `dismiss()` 忽略此警告。

```

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

```

`tearDown` 方法在每个测试方法执行后调用，这个方法用于完成测试用例执行后的清理工作，如退出浏览器、关闭驱动，恢复用例执行状态等。

在 `setUp()` 方法中定义了 `verificationErrors` 为空数组，这里通过 `assertEqual()` 比较其是否为空，

如果为空说明用例执行的过程过程中没有出现异常，否则将抛出 AssertionError 异常。

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

整个测试过程集成在 unittest.main() 模块中，其默认执行以 test 开头的方法。

Python 知识补充：

if __name__ == “__main__”: 语句说明

后面实例中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。

7.2 unittest 单元测试框架解析

相信通过上一节的学习我们已经对 unittest 了初步的认识，起码已经知道了用它写自动化测试用例是什么样子。这一节我们将更详细的学习 unittest 单元测试框架。

什么是单元测试？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。在 Python 语言下有诸多单元测试框架，如 unittest、Pytest、nose 等，其中 unittest 框架（原名 PyUnit 框架）为 Python 语言自带的单元测试框架，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。

7.2.1 认识单元测试

首先要说明一个问题，不用单元测试框架能写单元测试么？答案是肯定的，单元测试本身就是通过一段代码去验证另一个代码，所以不用单元测试框架也可以写单元测试，下面就看一下不用测试框架的测试。

首先创建一个被测试类 count.py

count.py

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

```
#计算器类
```

```
class Count:
```

```

def __init__(self,a,b):
    self.a = a
    self.b = b

#计算加法
def add(self):
    return self.a + self.b

```

程序非常简单，创建一个 Count 类用来计算两个数，通过 `__init__()` 方法对两个数进行初始化，接着常见 `add()` 方法返回两个数相加的结果。

根据上面所实现的功能，通过手工方式所进行的单元测试可能是下面（`test.py`）这个样子的。

test.py

```

#coding=utf-8
from count import Count

#测试两个整数相加
class TestCount:

    def test_add(self):
        try:
            j = Count(2,4)
            add = j.add()
            assert(add==6),'Integer addition result error!'
        except AssertionError,msg:
            print msg
        else:
            print 'test pass!'

#执行测试类的测试方法
mytest=TestCount()
mytest.test_add()

```

首先，引入 `count` 文件下的 `Count` 类，在 `test_add()` 方法中调用 `Count` 类并传入两个参数 `2,3`，调用 `Count` 类中的 `add()` 方法对两个参数做加法运算，通过 `assert()` 方法进行比较返回结果 `add` 是否等于 `5`，如果不相等则抛出自定义的“`Integer addition result error!`”异常信息，如果相等则打印“`test pass!`”。

Python Shell

```

#执行结果
>>>=====
RESTART =====
>>>
test pass!

```

#assert 比较不相等的结果如下：

```
>>>===== RESTART =====
>>>
Integer addition result error!
```

不难发现这种手工测试方法存在许多问题。首先，测试程序的写法没有一定的规范可以遵循，十个程序员完全可能写出十种不同的测试程序来，如果每个 Python 程序员都有自己不同的设计测试类的方法，光维护被测试的类就够麻烦了，谁还顾得上维护测试类。其次，需要编写大量的辅助代码才能进行单元测试，在 test.py 中用于测试的代码甚至比被测试的代码还要多，而这毫无疑问将增大 Python 程序员的工作量。

为了让单元测试代码能够被测试和维护人员更容易地理解，最好的解决办法是让开发人员遵循一定的规范来编写用于测试的代码，具体到 Python 程序员来讲，则是要采用 unittest 这一自动测试框架来构造单元测试用例。

接下来看看通过 unittest 来写单元测试是什么样子的。

test.py

```
#coding=utf-8
from count import Count
import unittest

class TestCount(unittest.TestCase):

    def setUp(self):
        self.j = Count(2,3)

    def test_add(self):
        self.add = self.j.add()
        self.assertEqual(self.add,5)

    def tearDown(self):
        pass

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

采用 unittest 单元测试框架后，用于测试的代码做了相应的调整。

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。

在 setUp() 方法中进行测试前的初始化工作，这里初始化了 Count 类。在 tearDown() 方法中执行测试后的清除工作，因为没什么可清理的工作，通过 pass 表示其为一个空的方法，什么也不做。setUp() 和 tearDown() 都是 TestCase 类中定义的方法。

在 `test_add()` 中调用 `assertEqual()` 方法，对 `Count` 类中的 `add()` 方法的返回值和预期值进行比较，确保两者是相等的，`assertEqual()` 也是 `TestCase` 类中定义的方法。

`Unittest` 提供了全局的 `main()` 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。

7.2.2 unittest 中的概念

的 `unittest` 的文档中开篇就介绍了 4 个重要的概念： `test fixture`, `test case`, `test suite`, `test runner`，我觉得只有理解了这几个概念，才能真正的理解单元测试的基本原理。

test case

一个 `TestCase` 的实例就是一个测试用例。什么是测试用例呢？就是一个完整的测试流程，包括测试前准备环境的搭建(`setUp`)，实现测试过程的代码(`run`)，以及测试后环境的还原(`tearDown`)。元测试(unit test)的本质也就在这里，一个测试用例是一个完整的测试单元，通过运行这个测试单元，可以对某一个功能进行验证。

test suite

对一个功能的验证往往是需要多测试用例的，可以把多的测试用例集合在一起执行，这就产生了测试套件 `TestSuite` 的概念，它用来组装单个测试用例，而且 `TestSuite` 也可以嵌套 `TestSuite`。

可以通过 `addTest` 加载 `TestCase` 到 `TestSuite` 中，再返回一个 `TestSuite` 实例。

test runner

`TextTestRunner` 是来执行测试用例的，其中的 `run(test)` 用来执行 `TestSuite/TestCase`。测试的结果会保存到 `TextTestResult` 实例中，包括运行了多少测试用例，成功了多少，失败了多少等信息。

test fixture

对一个测试用例环境的搭建和销毁，是一个 `fixture`，通过覆盖 `TestCase` 的 `setUp()` 和 `tearDown()` 方法来实现。这个有什么用呢？比如说在这个测试用例中需要访问数据库，那么可以在 `setUp()` 中建立数据库连接以及进行一些初始化，在 `tearDown()` 中清除在数据库中产生的数据，然后关闭连接。注意 `tearDown` 的过程很重要，要为以后的 `TestCase` 留下一个干净的环境。

再次对 `test.py` 的代码做调整。

test.py

```
#coding=utf-8
from count import Count
import unittest

class TestCount(unittest.TestCase):

    def setUp(self):
        self.j = Count(2,3)

    def test_add(self):
        self.add = self.j.add()
        self.assertEqual(self.add,5)

    def tearDown(self):
        pass

if __name__ == '__main__':
    # 构造测试集
    suite = unittest.TestSuite()
    suite.addTest(TestCount("test_add"))

    # 执行测试
    runner = unittest.TextTestRunner()
    runner.run(suite)
```

唯一的变化是在底部的几行代码，首先调用 unittest 提供的 TestSuite()类，通过它下面的 addTtest()方法来添加 TestCount 类下面的 test_add 测试方法。接着调用 TextTestRunner()类，通过它下面的 run()方法来运行 suite 所组装的测试用例，执行结果如下。

Python Shell

```
>>>===== RESTART =====
>>>
.
-----
Ran 1 test in 0.000s

OK
```

7.2.3 编写单元测试用例

在编写用例的过程中，不管用例的执行过程是怎样的，最初都会有一个预期结果，用例的执行就是通过执行用例的步骤，验证实际的结果是否与预期结果相等。unittest 框架的 TestCase 类提供一些方法用检查比较和报告失败。

方法	检查	版本
assertEqual(a, b)	a == b	
assertNotEqual(a, b)	a != b	
assertTrue(x)	bool(x) is True	
assertFalse(x)	bool(x) is False	
assertIs(a, b)	a is b	2.7
assert IsNot(a, b)	a is not b	2.7
assertIsNone(x)	x is None	2.7
assertIsNotNone(x)	x is not None	2.7
assertIn(a, b)	a in b	2.7
assertNotIn(a, b)	a not in b	2.7
assertIsInstance(a, b)	isinstance(a, b)	2.7
assertNotIsInstance(a, b)	not isinstance(a, b)	2.7

```
assertEqual(first, second, msg=None)
```

测试第一个和第二个是否是相等的，如果值不相等测试失败。msg 是否可选有参数，用于定义测试失败所要提示的信息。

test.py

```
#coding=utf-8
import unittest

class Test(unittest.TestCase):

    def setUp(self):
        number = input("Enter a number:")
        self.number = number

    def test_case(self):
        self.assertEqual(self.number, 10, msg="You input is not 10!")

    def tearDown(self):
        pass

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

在 `setUp()` 方法中要求用户输入一个数，在 `test_case()` 中通过 `assertEqual()` 比较输入的数是否与 10 的相等，如果不相等则输出 `msg` 中定义的提示信息。

Python Shell

```
>>> ===== RESTART =====
>>>
Enter a number:12
F
=====
FAIL: test_case ( __main__.Test )
-----
Traceback (most recent call last):
  File "D:\fnngj\test.py", line 11, in test_case
    self.assertEqual(self.number, 10, msg="You input is not 10!")
AssertionError: You input is not 10!
-----
Ran 1 test in 2.984s

FAILED (failures=1)
```

从执行结果看到，我们输入了一个 12，显然与预期的 10 不相等，`msg` 所定义的提示信息告诉我们“`You input is not 10!`”。

`assertNotEqual(first, second, msg=None)`

`assertNotEqual()` 的判断与 `assertEqual()` 刚好相反，它用于判断第一个与第二个是否是不相等的，如果相等测试失败。

`assertTrue(expr, msg=None)`

`assertFalse(expr, msg=None)`

用于测试表达式是 `true`（或 `false`）。

下面来实现一个判断是否为质数功能，所谓的质数（又叫素数）是指只能被 1 和它本身整除的数。

count.py

```
#coding=utf-8

#用于判断素数
def is_prime(n):
    if n <= 1:
        return False
```

```

for i in range(2, n):
    if n % i == 0:
        return False
return True

```

关于判断是否为质数的实现很简单，当拿到一个数后如果能整除从 2 与自身减 1 之间的任意一个数说明其不为质数，返回 False，否则返回 Ture。

test.py

```

#coding=utf-8
import unittest
import number

class Test(unittest.TestCase):

    def setUp(self):
        pass

    def test_case(self):
        self.prime = number.is_prime(4)
        self.assertTrue(self.prime,msg="Is not prime!")

    def tearDown(self):
        pass

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

```

在调用 `is_prime()` 函数时分别传不同的值来执行测试用例，在上面的例中传值为 4，显然不是一个质数，所以通过 `assertTrue()` 比较的结果一定不是 Ture，那么 `msg` 中所定义的信息将会被打印输出。

```
assertIn(first, second, msg=None)
```

```
assertNotIn(first, second, msg=None)
```

测试第一个是否在第二个中，反过来讲第二个是否包含第一个。

test.py

```

#coding=utf-8
import unittest

class Test(unittest.TestCase):

    def setUp(self):

```

```
pass

def test_case(self):
    self.a = "hello"
    self.b = "hello world"
    self.assertIn(self.a, self.b, msg="a is not in b")

def tearDown(self):
    pass

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

这个很好理解，定义字符串 a 为 “hello” , b 为 “hello world” 。通过 assertIn 判断 b 是否包含 a,如果不包含将打印 msg 的定义的信息。

```
assertIs(first, second, msg=None)

assert IsNot(first, second, msg=None)
```

测试第一个和第二个是否为同一对象。

```
assertIsNone(expr, msg=None)

assert IsNotNone(expr, msg=None)
```

测试表达式是否为 None 对象。

```
assertIsInstance(obj, cls, msg=None)

assertNotIsInstance(obj, cls, msg=None)
```

测试对象 (obj) 是否有一个实例 (cls) 。

在 unittest 中还提供更多用于检查比较的方法，请参考 Python 官方文档 unittest 章节。

7.2.4 组织单元测试用例

在前面的讲解，在使用单元测试框架时，我们只写了一个用例，这显然是不符合实际需求的，在实际的测试过程中真对一个功能，我们甚至要编写几个，甚至几十个测试用例。下面就来介绍如何组织这些测试用例。

我们同样以测试 7.2.1 中节中 count.py 文件为例：

test.py

```
#coding=utf-8
from count import Count
import unittest

class TestCount(unittest.TestCase):

    def setUp(self):
        pass

    # 测试整数相加
    def test_add(self):
        self.j = Count(2,3)
        self.add = self.j.add()
        self.assertEqual(self.add,5)

    # 测试小数相加
    def test_add2(self):
        self.j = Count(2.3,4.2)
        self.add = self.j.add()
        self.assertEqual(self.add,6.5)

    # 测试字符串相加
    def test_add3(self):
        self.j = Count("hello"," world")
        self.add = self.j.add()
        self.assertEqual(self.add,"hello world")

    def tearDown(self):
        pass

if __name__ == '__main__':
    # 构造测试集
    suite = unittest.TestSuite()
    suite.addTest(TestCount("test_add"))
    suite.addTest(TestCount("test_add2"))
```

```

suite.addTest(TestCount("test_add3"))

# 执行测试
runner = unittest.TextTestRunner()
runner.run(suite)

```

在上面的测试过程中，分别创建了三个测试方法（用例）用于测试验证整数、小数和字符串的相加。

在执行构造并执行测试的过程中并没有直接使用 main() 方法，而是分别调用了 TestSuite() 类和 TextTestRunner() 分别来组织和执行测试用例。这样做的好处是可以通过 addTest() 方法自由的添加想要被执行的用例。

扩充测试用例：

在 count.py 中继续开发新的新的功能是再正常不过的事情了，扩展后的 count.py 如下：

count.py

```

#coding=utf-8

#计算器类
class Count:

    def __init__(self,a,b):
        self.a = a
        self.b = b

    #计算加法
    def add(self):
        return self.a + self.b

    #计算减法
    def subtraction(self):
        return self.a - self.b

```

因为又开发了新的功能，所以我们需要针对新的功能编写测试用例，扩展后的 test.py 如下：

test.py

```

#coding=utf-8
from count import Count
import unittest

class TestAdd(unittest.TestCase):

    def setUp(self):

```

```
pass

# 测试整数相加
def test_add(self):
    self.j = Count(2,3)
    self.add = self.j.add()
    self.assertEqual(self.add,5)

# 测试小数相加
def test_add2(self):
    self.j = Count(2.3,4.2)
    self.add = self.j.add()
    self.assertEqual(self.add,6.5)

# 测试字符串相加
def test_add3(self):
    self.j = Count("hello"," world")
    self.add = self.j.add()
    self.assertEqual(self.add,"hello world")

def tearDown(self):
    pass

class TestSubtraction(unittest.TestCase):

    def setUp(self):
        pass

    # 测试整数相减
    def test_subtraction(self):
        self.j = Count(2,3)
        self.sub = self.j.subtraction()
        self.assertEqual(self.sub,-1)

    # 测试小数相减
    def test_subtraction2(self):
        self.j = Count(4.2,2.2)
        self.sub = self.j.subtraction()
        self.assertEqual(self.sub,2)

    def tearDown(self):
        pass

if __name__ == '__main__':
    # 构造测试集
```

```

suite = unittest.TestSuite()
suite.addTest(TestAdd("test_add"))
suite.addTest(TestAdd("test_add2"))
suite.addTest(TestAdd("test_add3"))

suite.addTest(TestSubtraction("test_subtraction"))
suite.addTest(TestSubtraction("test_subtraction2"))

# 执行测试
runner = unittest.TextTestRunner()
runner.run(suite)

```

创建了 TestAdd() 和 TestSubtraction() 两个测试类分别测试 count.py 中实现的 add() 和 Subtraction() 两个功能。用例的组装也非常简单，通过 addTest() 把不同的测试类和测试方法组装到测试套件中。

7.2.5 discover 更多测试用例

随着功能的不断增加，对应的测试用例也呈现指数级的增了，对于实现十几个功能程序来讲，对应的单元测试用例可能就会达到上百个，对于这种情况 test.py 文件会变得异常臃肿，我们不得不将这些用例进行划分，分散到不同的文件中，这样更便于维护。

当然，我们首先想到的是对 test.py 文件的测试用例进行拆分，拆分后的目录结构如下：

```

.../test_project/all_testcase.py

    /testadd.py

    /testsub.py

    /count.py

```

这三个文件的实现代码如下：

testadd.py

```

#coding=utf-8
from count import Count
import unittest

class TestAdd(unittest.TestCase):

    def setUp(self):
        pass

    # 测试整数相加

```

```

def test_add(self):
    self.j = Count(2,3)
    self.add = self.j.add()
    self.assertEqual(self.add,5)

# 测试小数相加
def test_add2(self):
    self.j = Count(2.3,4.2)
    self.add = self.j.add()
    self.assertEqual(self.add,6.5)

# 测试字符串相加
def test_add3(self):
    self.j = Count("hello"," world")
    self.add = self.j.add()
    self.assertEqual(self.add,"hello world")

def tearDown(self):
    pass

if __name__ == '__main__':
    # 构造测试集
    suite = unittest.TestSuite()
    suite.addTest(TestAdd("test_add"))
    suite.addTest(TestAdd("test_add2"))
    suite.addTest(TestAdd("test_add3"))

    # 执行测试
    runner = unittest.TextTestRunner()
    runner.run(suite)

```

testsub.py

```

#coding=utf-8
from count import Count
import unittest

class TestSubtraction(unittest.TestCase):

    def setUp(self):
        pass

    # 测试整数相减
    def test_subtraction(self):

```

```

    self.j = Count(2,3)
    self.sub = self.j.subtraction()
    self.assertEqual(self.sub,-1)

#测试小数相减
def test_subtraction2(self):
    self.j = Count(4.2,2.2)
    self.sub = self.j.subtraction()
    self.assertEqual(self.sub,2)

def tearDown(self):
    pass

if __name__ == '__main__':
    # 构造测试集
    suite = unittest.TestSuite()
    suite.addTest(TestSubtraction("test_subtraction"))
    suite.addTest(TestSubtraction("test_subtraction2"))

    # 执行测试
    runner = unittest.TextTestRunner()
    runner.run(suite)

```

接着创建用于执行所有用例的 all_test.py 文件的所有用例。

all_test.py

```

#coding=utf-8
import unittest
#加载测试文件
import testadd
import testsub

# 构造测试集
suite = unittest.TestSuite()

suite.addTest(testadd.TestAdd("test_add"))
suite.addTest(testadd.TestAdd("test_add2"))
suite.addTest(testadd.TestAdd("test_add3"))

suite.addTest(testsub.TestSubtraction("test_subtraction"))
suite.addTest(testsub.TestSubtraction("test_subtraction2"))

if __name__ == '__main__':
    # 执行测试
    runner = unittest.TextTestRunner()

```

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

这样的划分带来了很多好处，首先根据不同的功能创建不同的文件，文件下面还可以针对不同的小功能进行类的划分，在类下对这些小功能编写不同的测试用例。从整体机构上更加清晰。

在上同的代码中，创建 testadd.py 和 testsub.py 分别用于测试 count.py 中的 add() 和 subtraction() 两个方法。在每个测试文件中，可以通过 addTest() 任意的加载当前要运行测试用例。

接着创建 all_test.py 文件，在这个文件中首先导入不同的测试文件，通过 addTest() 来加载所有的测试文件下面的测试类下面的测试用例，最终通过 TextTestRunner() 类来运行所有用例。

这样的设计看上去很完美，当用例达到成百上千条时在 all_test.py 文件中 addTest() 测试用例就变成了一条纯的体力活儿，那么有没有更好的办法 unittest 可以自己识别不同测试文件不同测试类下面的测试用例呢。在 TestLoader 类中提供了 discover() 方法可以解决这个问题。

TestLoader

该类根据各种标准负责加载测试用例，并它们返回给测试套件。正常情况下没有必要创建这个类的实例。unittest 提供了可以共享了 defaultTestLoader 类，可以使用其子类和方法创建实例，所以我们可以使用其下面的 discover() 方法来创建一个实例。

```
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。

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。

现在通过 discover 方法重新实现 all_test.py 文件的功能。

all_test.py

```
#coding=utf-8
import unittest

def createsuite():
    testunit=unittest.TestSuite()
    #定义测试文件查找的目录
    test_dir='E:\\test_project'
```

```

# 定义 discover 方法的参数
discover=unittest.defaultTestLoader.discover(test_dir,
                                              pattern ='test*.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

if __name__ == '__main__':
    runner =unittest.TextTestRunner()
    runner.run(alltestnames)

```

创建 createsuite() 函数用于查找指定条件下的所有测试用例，并将其组装到测试套件中。

调用 discover() 方法，首先通过 test_dir 定义查找测试文件的目录，pattern 用来定义匹配测试文件的文件名，如果文件名以 test 开头的. py 文件，那么就认为它是一个测试文件。top_level_dir 默认为 None。

discover() 只能找到了测试文件，接下来通过 for 循环来找到测试文件下的每一个测试用例，并且将其添加测试套件中。

Python Shell

```

>>> ===== RESTART =====
>>>
<unittest.suite.TestSuite tests=[<unittest.suite.TestSuite
tests=[<testadd.TestAdd testMethod=test_add>,
      <testadd.TestAdd testMethod=test_add2>,
      <testadd.TestAdd testMethod=test_add3>]>]>
<unittest.suite.TestSuite tests=[<unittest.suite.TestSuite
tests=[<testsub.TestSubtraction testMethod=test_subtraction>,
      <testsub.TestSubtraction testMethod=test_subtraction2>]>]>

.....
-----
Ran 5 tests in 0.000s

```

OK

在 for 循环遍历测试文件中测试用例时，对每一个用例进行打印，从打印结果中也可以很清晰的看到每一条用例的所属文件和类。

如果你创建你创建的测试文件在' E:\\test_project' 目录下，并且文件以 test 开头的. py 文件，那么对于 all_test.py 来说，它不需要做任何修改就可以新创建的测试用例，这是多么美妙的事情。

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。对于文件名的匹配规则完全由 discover() 方法中的 pattern 参数决定。

7.3 用 unittest 编写 web 自动化

学以致用是本书的出发点，我们用了相当篇幅来详细学习了 unittest 单元测试框架，目的是将其应用到 web 自动化测试中，用它来组织运行自动化测试脚本是一个不错搭配。

在动手写脚本之后前，我们先来简单的规划一个测试项目的目录：

```
.../test_project/all_test.py
    /test_case/test_baidu.py
    /test_case/test_youdao.py
    /report/log.txt
```

创建 web 测试用例：

test_baidu.py

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

class MyTest(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.maximize_window()
        self.driver.implicitly_wait(10)
        self.base_url = "http://www.baidu.com"

    def test_baidu(self):
        driver = self.driver
        driver.get(self.base_url + "/")
        driver.find_element_by_id("kw").clear()
        driver.find_element_by_id("kw").send_keys("unittest")
        driver.find_element_by_id("su").click()
        time.sleep(2)
        title = driver.title
```

```

    self.assertEqual(title,u"unittest_百度搜索")

def tearDown(self):
    self.driver.quit()

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

```

test_youdao.py

```

#coding=utf-8
from selenium import webdriver
import unittest,time

class MyTest(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.maximize_window()
        self.driver.implicitly_wait(10)
        self.base_url = "http://www.youdao.com"

    def test_youdao(self):
        driver = self.driver
        driver.get(self.base_url + "/")
        driver.find_element_by_id("query").clear()
        driver.find_element_by_id("query").send_keys("webdriver")
        driver.find_element_by_id("qb").click()
        time.sleep(2)
        title = driver.title
        self.assertEqual(title,u"webdriver - 有道搜索")

    def tearDown(self):
        self.driver.close()

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

```

在 test_case/ 目录下分别创建百度搜索 test_baidu.py 和有道搜索 test_youdao.py 测试文件，分别在文件内创建测试用例。

all_test.py 文件的创建请参考上一节中 all_test.py 的代码实现，唯一需要做的改动就是指定新的测试目录 “/test_project/test_case” ，好了，现在可以运行它来跑所有测试用例了。

保存测试结果：

你可能还有个疑问，report 目录是干嘛的？从命名上猜到是输出测试报告的，那么怎么把测试结果写到它下面的 log.txt 文件中呢？

首先打开 Windows 命令提示符，进入到.../test_project/目录下执行命令：

```
>Python all_test.py >> report/log.txt 2>&1
```

Python all_test.py 通过 Python 执行 all_test 文件

>>report/log.txt 将测试输出写入到 report 目录下的 log.txt 文件中

file 2>&1 标准输出被重定向到文件 file，然后错误输出也重定向到和标准输出一样，所以错误输出也写入文件 file。

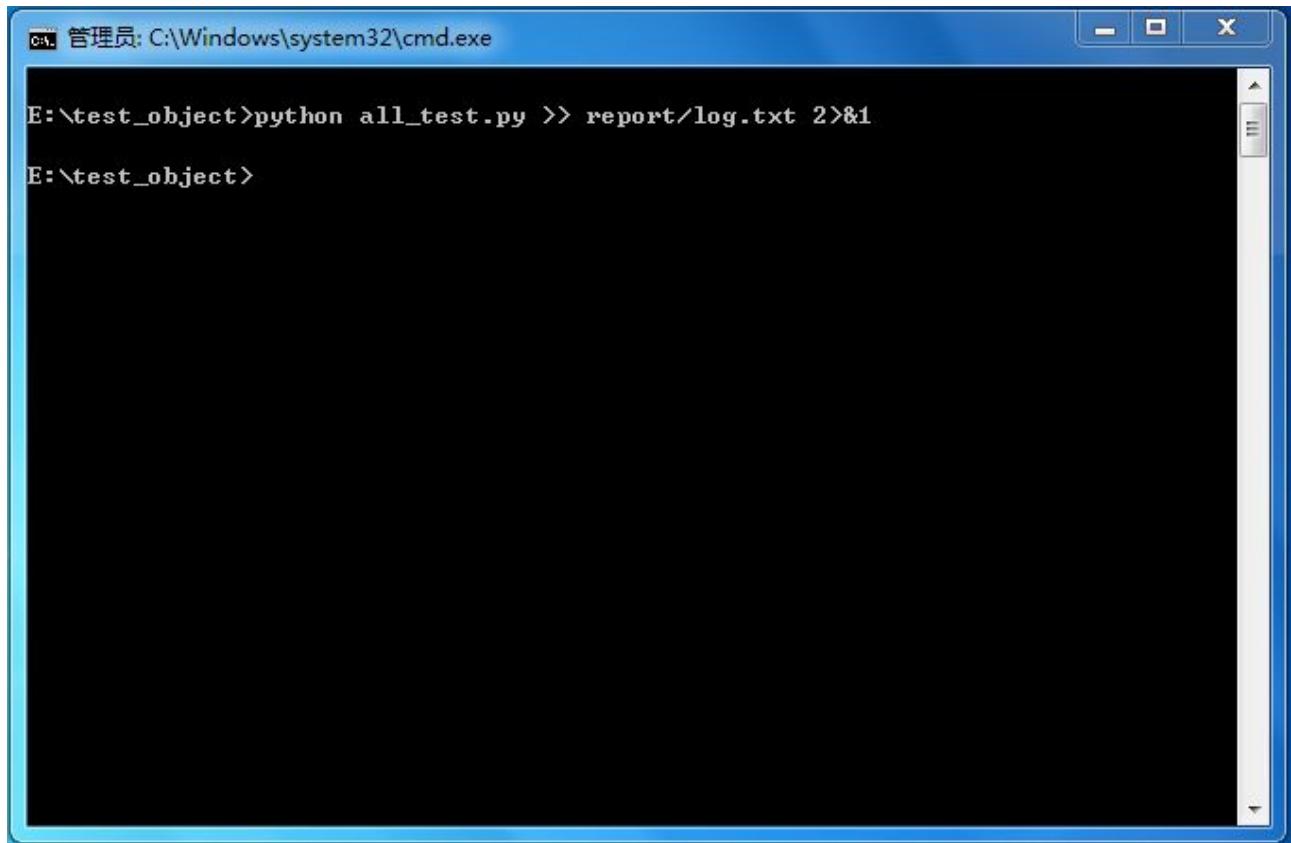


图 7.1 执行 all_test.py 文件

下面打 log.txt 文件，内容下如：

log.txt

```
<unittest.suite.TestSuite tests=[<unittest.suite.TestSuite
tests=[<test_youdao.MyTest testMethod=test_youdao>]>]>
<unittest.suite.TestSuite tests=[<unittest.suite.TestSuite
tests=[<test_baidu.MyTest testMethod=test_baidu>]>]>
```

..

Ran 2 tests in 43.060s

OK

7.4 用例执行的疑惑

在结束本章之前，还有两个问题需要和读者交代一下，也许你已经产生了这样人疑惑，也许你会在接下来的实践过程中产生这样人疑惑，不管怎样我先来回答这些问题。

7.4.1 用例执行的顺序

这个顺序分两个部分，首先是同步测试问题，不同测试类测试方法的执行顺序的，同一目录下不同测试文件的执行顺序。

我们继续以 count.py 的测试文件 test.py 为例子。

test.py

```
#coding=utf-8
from count import Count
import unittest

class TestBdd(unittest.TestCase):

    def setUp(self):
        pass

    # 测试整数相加
    def test_ccc(self):
        self.j = Count(2,3)
        self.add = self.j.add()
        self.assertEqual(self.add,5)
        print 'test1'

    # 测试字符串相加
    def test_aaa(self):
        self.j = Count("hello"," world")
        self.add = self.j.add()
        self.assertEqual(self.add,"hello world")
```

```

print 'test3'

def tearDown(self):
    pass

class TestAdd(unittest.TestCase):

    def setUp(self):
        pass

    # 测试小数相加
    def test_bbb(self):
        self.j = Count(2.3,4.2)
        self.add = self.j.add()
        self.assertEqual(self.add,6.5)
        print 'test2'

    def tearDown(self):
        pass

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

```

用例的执行结果如下：

Python Shell

```

>>> ===== RESTART =====
>>>
test2
test3
test1
...
-----
Ran 3 tests in 0.000s

OK

```

首先在 test.py 的例子中修改了用例的名称，并且在每个用例结尾打印当前用例的编号。通过执行结果发现的了规律。

先执行了 TestAdd 类中的 test_bbb() 测试用例，接着执行个 TestBdd 类中的 test_aaa() 测试用例，最后执行了 TestBdd 类中的 test_ccc() 测试用例。从上下顺序的来将，应该先执行 test_ccc() 才对，显然 main() 的加载不中从上到下的顺序。它的默认加载顺序是根据 ASCII 码的顺序，数字与字母的顺序为：

`0~9, A~Z, a~z, main()`的加载顺序是从小到大的，所以 `TestAdd` 比 `TestBdd` 先加载，`test_aaa()` 比 `test_ccc()` 先执行。

那我就想让 `test_ccc()` 先执行可不可以呢？当然可以，我们可以不用默认的 `main()` 方法，通过测试套件的 `addTest()` 根据需求来加载。

test.py

```
....
```

```
if __name__ == '__main__':
    # 构造测试集
    suite = unittest.TestSuite()
    suite.addTest(TestBdd("test_ccc"))
    suite.addTest(TestAdd("test_bbb"))
    suite.addTest(TestBdd("test_aaa"))

    # 执行测试
    runner = unittest.TextTestRunner()
    runner.run(suite)
```

执行结果如下：

Python Shell

```
>>> ===== RESTART =====
>>>
test1
test2
test3
...
-----
Ran 3 tests in 0.000s
```

OK

这一次就按照我们想要的加载顺序来执行测试用例了，但这只限制于单个文件，如果用例多了，最省力的还是直接使用 `main()` 来加载测试用例。

那么 `discover()` 方法加载用例也是按照 ASCII 码的从小到大的顺序来加载同一目录下测试文件的，这个就更难控制了，我们唯一能做的就是通过测试文件的命名来为其排列执行顺序。

7.4.2 执行多级目录的用例

有时候测试的业务多了就要对测试用例进行划分目录，例如我们下面的测试目录：

```
.../test_case/test_aa.py
    /test_aaa/test_a.py
        /test_aaa/test_ccc/test_c.py
            /test_bbb/test_b.py
```

对于上面的目录结构，如果将 discover() 方法中 start_dir 参数定义为 “.../test_case/” 目录，只能执行 test_aa.py 文件，对于 “/test_aaa/” 、 “/test_aaa/test_ccc/” 和 “/test_bbb/” 等二级、三级目录下的用例将不会被执行。如果想被 discover() 读取执行非常简单，在目录下添加 __init__.py 文件，对于 __init__.py 文件的作用请回到本书第三章学习。

修改后的目录如下：

```
.../test_case/test_aa.py
    /test_aaa/test_a.py
        /test_aaa/__init__.py
            /test_aaa/test_ccc/test_c.py
                /test_aaa/test_ccc/__init__.py
                    /test_bbb/test_b.py
                        /test_bbb/__init__.py
```

执行结果如下：

Python Shell

```
>>> ===== RESTART =====
>>>
<unittest.suite.TestSuite tests=[<unittest.suite.TestSuite
tests=[<test_aa.MyTest testMethod=test_baidu>]>]>
<unittest.suite.TestSuite tests=[<unittest.suite.TestSuite
tests=[<test_aaa.test_a.MyTest testMethod=test_baidu>]>]>
<unittest.suite.TestSuite tests=[<unittest.suite.TestSuite
tests=[<test_aaa.test_ccc.test_c.MyTest testMethod=test_baidu>]>]>
<unittest.suite.TestSuite tests=[<unittest.suite.TestSuite
tests=[<test_bbb.test_b.MyTest testMethod=test_baidu>]>]>
...
-----
Ran 4 tests in 59.757s
```

OK

通过上面的执行结果也可以很清楚的看出所读取测试用例的目录名、文件名、类名和方法名。

小结：

本章从分析 selenium IDE 所录制导出的 Python/unittest/webdriver 脚本开始学习 unittest 单元测试框架，接下来详细的学习了 unittest 单元测试的概念，如何编写单元测试用例，以及通过有 discover() 方法来组织运行测试用例。最终又回到原点，学习如何通过 unittest 编写 web 自动化测试。相信通过本章的学习，读者对单元测试已经基本掌握了使用技巧。

第8章 自动化测试项目实战

通过前面章节的学习，我们已经学到了不少知识，通过 webdriver API 的学习已经可以轻松操作页面元素，通过自动化测试模型的学习，已经可以轻松的创建模块，根据需求读取各种类型的文件数据，通过 unittest 单元测试的框架的练习，已经可以轻松的将用例组织起来运行。本章将会把这些知识揉合到一起运用到实际项目，使我们具备基本的项目经验。

8.1 自动化测试用例设计

测试人员不管是在进行功能测试、自动化测试还是性能测试都需要编写测试用例，测试用例的好坏往往能准确的体现了测试人员的经验、能力以及对项目需求的理解深度。所以，在正式开展自动化测试工作之前，我们很好必要聊聊自动化测试用例的一些特点，以及如何有编写自动化测试用例。

8.1.1 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步骤进行测试，只能在已知的步骤与场景中发现问题，而且往往因为网络问题或功能的微小变化导致用例执行异常，自动化的执行也很难发现新的 bug。

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较大的差异。所以，不能直接拿手工测试用例直接“翻译”成自动化测试脚本。

通过它们之间的特点对比也可清晰的认知到，自动化测试交不能完全的替代手工测试，自动化测试的目的仅仅在于让测试人员从繁琐重复的测试过程解脱出来，把更多的时间和精力投入到更有价值的测试中，比如探索性测试。而自动化测试更多的是来进行冒烟测试和回归测试；所以笔者更推崇的是半自动化测试，把重复的工作交给工具，测试员完成更有价值非重复性工作。

自动化测试用例选型注意事项：

- 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。

8.1.2 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’。验证文件是否存在的测试脚本,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。

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。

8.2 126 邮箱项目实战

本节以 126 邮箱为例,综合运用前面所讲到的知识点,来完成一个自动化测试项目。

8.2.1 编写自动化测试用例

在编写用例过程中应该遵守以下几点原则：

- 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。

关于用例的存放与维护大概分两类，一类通过 word、excel 等文档工具自行编写与维护，另一类由缺陷管理工具维护，如 QC、禅道项目管理等都自带的有用例编写与维护工能。应为自动化测试用例对执行的步骤有着更苛刻的要求，所以用 excel 描述是个不错的选择。

用例001：

用例 id	test_login.py	用户登录与退出	
步骤	动作	数据	验证点
1	打开登陆页	http://www.126.com	
2	用户登陆	username 123456	匹配用户昵称“xxx”
3	用户退出		

将测试用例分为动作、数据和验证点几个部分，动作用于描述要如何操作；数据是这一步操作所用到的数据，验证点是这一步操作完成需要验证的信息。这种形式的自动化测试用例看上去更像关键字驱动的脚本。优点是可以清晰的根据用例来实现自化测试脚本。

8.2.2 录制测试脚本

什么？学了这么多知识，居然还要录制脚本？如果你已经非常熟练的编写脚本当然不需要录制，但如果你的编写能力还不够，那么我们可以选择通过 selenium IDE 录制一个脚本出来，在此基础来进行编写。

第一步、录制脚本

录制 126 邮箱登录脚本，打开 firefox 浏览器，通过 selenium IDE 录制第一个 126 邮箱登录脚本。如图 8.1。

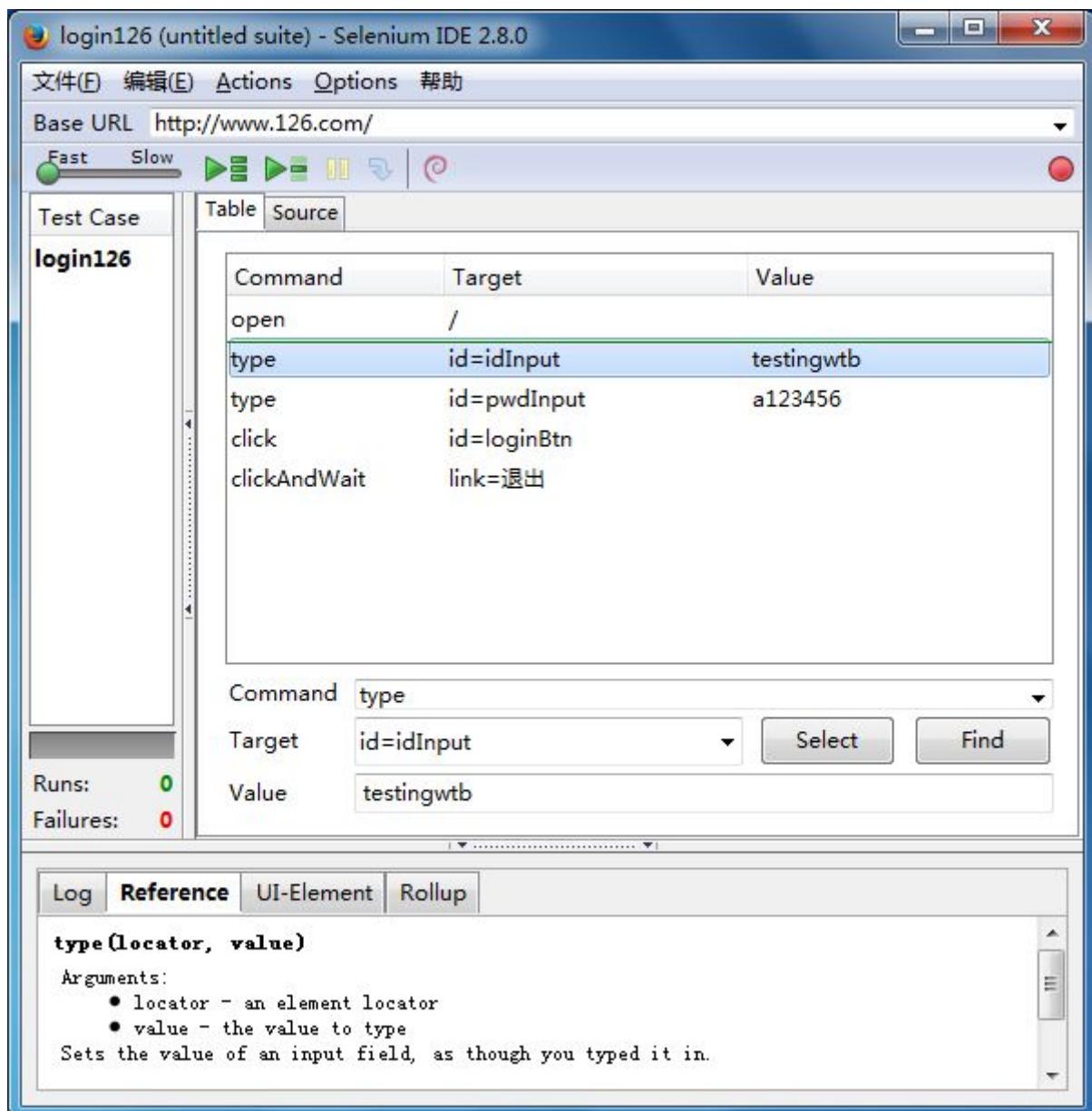


图 8.1 录制 126 邮箱登录脚本

第二步，添加断言

登录邮箱成功，我们可以邮箱帐号作为断言信息来判断是否登录成功。如图 8.2



图 8.2 确定断言信息

在脚本上右击，选择“insert new command”选项，插入一条空白行，通过前端工具获取信息的定位，从而手动编写一条断言信息。或直接在要断言的信息上右键“Show All Available Commands”选项来添加一条断言。添加断言的脚本如图 8.3

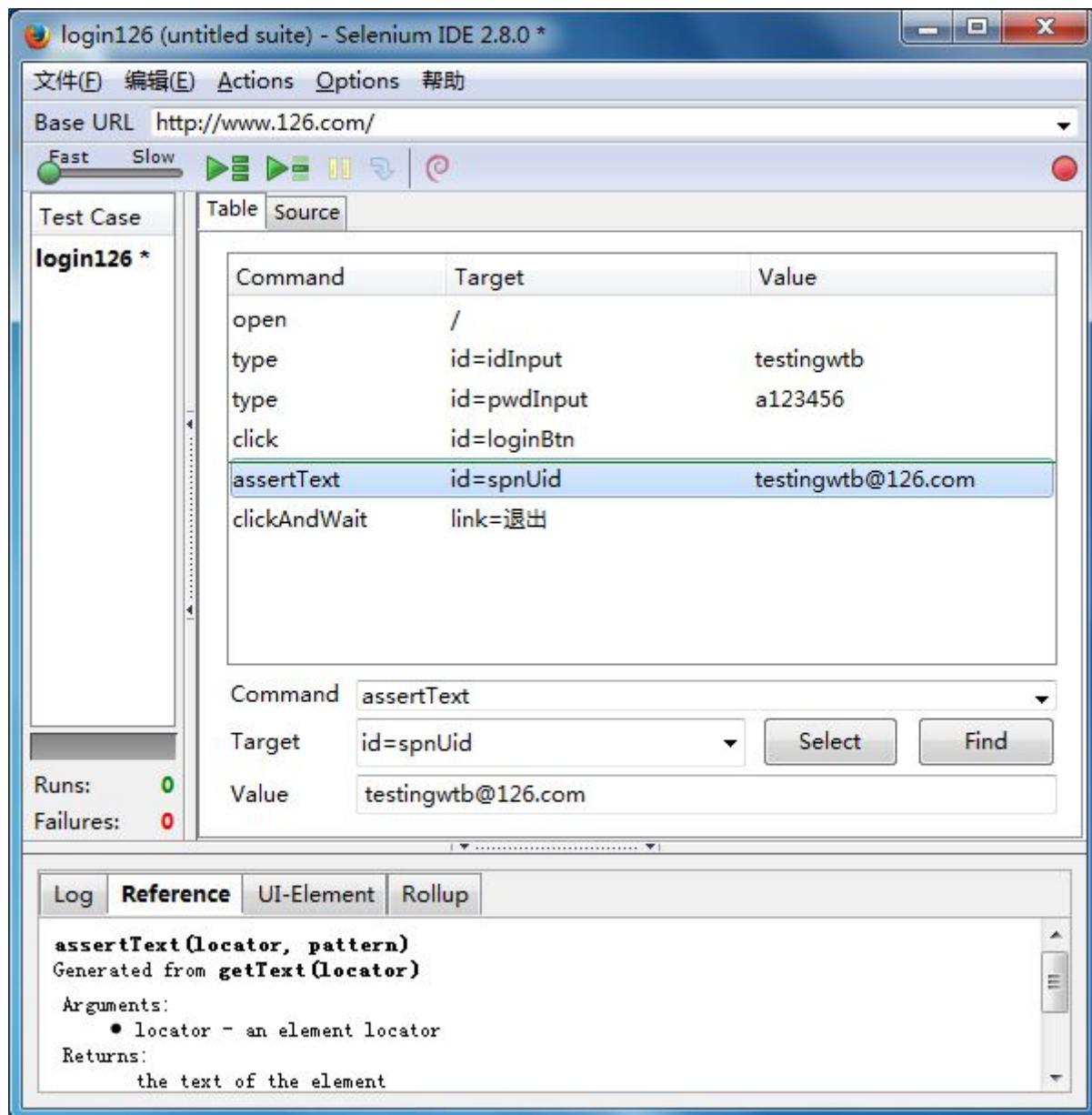


图 8.3 添加断言

第三步，导出脚本

现在将脚本导出为我们需要的格式，在 selenium IDE 窗口选择菜单栏，File --> Export Test Case As... --> Python2 /unittest /WebDriver，将脚本导出，保存为 test_login.py，对脚本进行简单调整，如下：

```
test_login.py
```

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

```

import unittest, time

class TestLogin(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.base_url = "http://www.126.com/"
        selfverificationErrors = []
        self.accept_next_alert = True

    def test_login(self):
        driver = self.driver
        driver.get(self.base_url)
        #登录
        driver.find_element_by_id("idInput").clear()
        driver.find_element_by_id("idInput").send_keys("testingwtb")
        driver.find_element_by_id("pwdInput").clear()
        driver.find_element_by_id("pwdInput").send_keys("a123456")
        driver.find_element_by_id("loginBtn").click()
        #获取断言信息进行断言
        text = driver.find_element_by_id("spnUid").text
        self.assertEqual(text, "testingwtb@126.com")
        #退出
        driver.find_element_by_link_text(u"退出").click()

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

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

```

我们从录制导的脚本中删除了一些断言方法，添加相应的注释信息，使脚本看上去更清爽且更简单。对于脚本的实现这里就不再解释，我们前面已经花费了相当精力来学习这段脚本。

8.2.3 进行模块化设计

通过对 126 邮箱的分析，我们发现所有的用例都是需要登录邮箱之后的操作，例如，邮件的发送，接收，删除等操作。所以，可以对登录和退出进行模块化设计。

结构如下：

test_project/test_case/test_login.py

test_case/public/login.py

test_case/public/__init__.py

public 目录存一些公共模块，供用例调用。login.py 内容如下：

login.py

```
#coding=utf-8

#登录
def login(self,username,password):
    driver = self.driver
    driver.find_element_by_id("idInput").clear()
    driver.find_element_by_id("idInput").send_keys(username)
    driver.find_element_by_id("pwdInput").clear()
    driver.find_element_by_id("pwdInput").send_keys(password)
    driver.find_element_by_id("loginBtn").click()

#退出
def logout(self):
    driver = self.driver
    driver.find_element_by_link_text(u"退出").click()
```

接下来修改 test_login.py 文件，使引用 login.py 中所定义的函数。

test_login.py

```
#coding=utf-8
from selenium import webdriver
import unittest, time
#导入 login 文件
from public import login

class TestLogin(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.base_url = "http://www.126.com/"
        selfverificationErrors = []

    def test_login(self):
        driver = self.driver
        driver.get(self.base_url)
        #调用登录函数
```

```

login.login(self,"testingwtb","a123456")
#获取断言信息进行断言
text = driver.find_element_by_id("spnUid").text
self.assertEqual(text,"testingwtb@126.com")
#调用退出函数
login.logout(self)

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

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

```

在 test_login.py 测试用例文件中，首先导入 public 目录下的 login 文件，然后在后 test_login() 方法中调用 login() 和 logout() 两个方法，这样对于测试用例来把注意力更集中在登录之后的验证上面。

8.2.4 参数化的应用

接下来我们要创建一组测试用例，用于验证 126 邮箱的登录功能，在此之前我们需要先编写好测试用例。

用例 id	test_login.py	登录功能验证	
步骤：	动作	数据	验证点
1	打开登陆页	http://www.126.com	
2	不输入用户名、密码登陆	null, null	用户名或密码不能为空
3	只输入用户名，密码为空登录	username, null	用户名或密码不能为空
4	用户名为空，只输入密码登录	null, passworld	用户名或密码不能为空
5	输入错误的用户名、密码登录	error, error	用户名或密码错误

当然，对于 126 邮箱的登录功能来说测试不仅仅这么几条，这里只是举几条一般登录的验证为例。

下面，我们就通过数据驱动的方式来编写自动化测试用例，对于登录功能来讲，每一条用例都需要三条数据：用户，密码，提示信息。因为不同的用例要取不同数据进行测试与验证，所有用 xml 文件存放数据更为何时。下面我们就创建 xml 文件来存放这些数据。

login.xml

```

<?xml version="1.0" encoding="utf-8"?>
<info>
    <explain>126 邮箱登录</explain>
    <url>http://www.126.com</url>
    <null username="" password="">请先输入您的邮箱帐号</null>
    <pawd_null username="testingwtb" password="">请输入您的密码</pawd_null>

```

```

<user_null username="" password="a123456">请先输入您的邮箱帐号
</user_null>
<error username="xxx" password="xxx">帐号或密码错误</error>
</info>

```

根据测试用例编写测试脚本：

test_login.py

```

#coding=utf-8
from selenium import webdriver
import unittest, time
from public import login
import xml.dom.minidom

#打开 xml 文档
dom = xml.dom.minidom.parse('E:\\test_object\\test_date\\login.xml')
#得到文档元素对象
root = dom.documentElement

class TestLogin(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        logins = root.getElementsByTagName('url')
        self.base_url=logins[0].firstChild.data
        selfverificationErrors = []

    #用户名、密码为空
    def test_null(self):
        driver = self.driver
        driver.get(self.base_url)
        logins = root.getElementsByTagName('null')
        #获得 null 标签的 username、password 属性值
        username=logins[0].getAttribute("username")
        password=logins[0].getAttribute("password")
        prompt_info = logins[0].firstChild.data
        #登录
        login.login(self,username,password)
        #获取断言信息进行断言
        text = driver.find_element_by_xpath("//div[@class='error-tt']/p").text
        self.assertEqual(text,prompt_info)

    #输入用户名、密码为空

```

```
def test_pawd_null(self):
    driver = self.driver
    driver.get(self.base_url)
    logins = root.getElementsByTagName('pawd_null')
    #获得 null 标签的 username、password 属性值
    username=logins[0].getAttribute("username")
    password=logins[0].getAttribute("password")
    prompt_info = logins[0].firstChild.data
    #登录
    login.login(self,username,password)
    #获取断言信息进行断言
    text = driver.findElement(By.xpath("//div[@class='error-tt']/p")).text
    self.assertEqual(text,prompt_info)

#用户名为空，只输入密码
def test_user_null(self):
    driver = self.driver
    driver.get(self.base_url)
    logins = root.getElementsByTagName('user_null')
    #获得 null 标签的 username、password 属性值
    username=logins[0].getAttribute("username")
    password=logins[0].getAttribute("password")
    prompt_info = logins[0].firstChild.data
    #登录
    login.login(self,username,password)
    #获取断言信息进行断言
    text = driver.findElement(By.xpath("//div[@class='error-tt']/p")).text
    self.assertEqual(text,prompt_info)

#用户名密码错误
def test_error(self):
    driver = self.driver
    driver.get(self.base_url)
    logins = root.getElementsByTagName('error')
    #获得 null 标签的 username、password 属性值
    username=logins[0].getAttribute("username")
    password=logins[0].getAttribute("password")
    prompt_info = logins[0].firstChild.data
    #登录
    login.login(self,username,password)
    #获取断言信息进行断言
    text = driver.findElement(By.xpath("//div[@class='error-tt']/p")).text
    self.assertEqual(text,prompt_info)

def tearDown(self):
```

```

    self.driver.quit()
    self.assertEqual([], selfverificationErrors)

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

```

请读者先回顾第五章通过 Python 读取 xml 的知识点。

首先我们对要访问的 URL 也进行参数化，通过读取 login.xml 文件中的 url 标签对文本信息进行 URL 的访问。

接着就是用户登录与验证信息的获取。通过 getElementsByTagName() 获取标签名，getAttribute() 方法获取 useranem 和 password() 的属性值。通过 firstChild.data 获取登录的验证信息。在用例执行的过程中通过 text 方法捕捉前端的错误提示信息给 text 变量。assertEqual() 用于断言前端捕捉的错误提示是否与 xml 文件中读取的信息一致。

其实，对于上面的一组测试用例来说唯一不同的就是通过 getElementsByTagName() 读取不同标签的数据。

8.2.4 输出测试结果

我们希望测试结果都保存到文件中，这样方便查看。在此之前很有必要梳理一下当测试项目的目录结构：

test_project/test_case/test_login.py	----测试用例
/test_case/public/login.py	---测试用调用的公共模块
/test_case/public/__init__.py	
/test_date/login.xml	----用于存放测试数据文件
/report/log.txt	----测试执行的结果
/report/all_test.py	----测试所有测试用例

关于 all_test.py 文件的创建请参考第 7 章中的实例代码，唯一需要调整的就是 discover() 方法的一个参数。这里就不再重复粘贴这段代码。现在运行 all_test.py 测试文件进行吧。

cmd.exe

E:\test_object>Python all_test.py >> report/log.txt 2>&1

执行的测试结果如下：

log.txt

```

<unittest.suite.TestSuite tests=[<unittest.suite.TestSuite
    tests=[<test_login.TestLogin testMethod=test_error>,
           <test_login.TestLogin testMethod=test_null>,
           <test_login.TestLogin testMethod=test_pawd_null>,
           <test_login.TestLogin testMethod=test_user_null>]]>
...
-----
Ran 4 tests in 57.322s

OK

```

8.3 扩展自动化测试用例

当前测试项目的目录结构已经形成，接下来要做的事情是继续丰富我们的测试用例。继续以 126 邮箱为例，实现发送邮件、搜索邮件、删除邮箱的自动化测试用例。

8.3.1 发送邮件

首先编写测试用例

用例 001：

用例 id	test_sendmail.py	发送邮件	
步骤	动作	数据	验证点
1	打开邮箱	http://webcloud.126.com	
2	用户登陆	username, password	
3	写信	只输入收信人发送	发送成功
4	退出		

test_send.py

```

#coding=utf-8
from selenium import webdriver
import unittest, time
from public import login
import xml.dom.minidom

#打开 xml 文档
dom = xml.dom.minidom.parse('E:\\test_object\\test_date\\login.xml')
#得到文档元素对象
root = dom.documentElement

```

```

class TestSendMail(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.maximize_window()
        self.driver.implicitly_wait(30)
        logins = root.getElementsByTagName('url')
        self.base_url=logins[0].firstChild.data
        self.verifyErrors = []

    #只填写收件人发送邮件
    def test_send_mail(self):
        driver = self.driver
        driver.get(self.base_url)
        #登录
        login.login(self,"testingwtb","a123456")
        #写信
        driver.find_element_by_css_selector("#_mail_component_47_47 > span.oz0").click()
        #填写收件人
        driver.find_element_by_xpath("//*[@class='bz0']/div[2]/div/input")
            .send_keys('testingwtb@126.com')
        #发送邮件
        driver.find_element_by_xpath("//html/body/div[2]/div/div[2]/header
                                    /div/div/div/span[2]").click()
        driver.find_element_by_xpath("//*[@class='nui-msgbox-ft-btns']
                                    /div/span").click()
        #断言发送结果
        text = driver.find_element_by_class_name('tK1').text
        self.assertEqual(text,u'发送成功')
        login.logout(self)

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

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

```

在只输入收件人发送邮件时，系统会弹框提示“确定真的不需要写主题吗？”，在用例中默认 click “确定”。邮件发送成功，验证“发送成功”的提示信息。

用例 002：

用例 id	test_sendmail.py	发送邮件	
-------	------------------	------	--

步骤	动作	数据	验证点
1	打开邮箱	http://webcloud.126.com	
2	用户登陆	username, password	
3	写信	只输入收信人与主题发送	发送成功
4	退出		

在 test_send.py 文件中添加测试用例：

test_send.py

```
.....
#填写收件人、主题发送邮件
def test_send_mail2(self):
    driver = self.driver
    driver.get(self.base_url)
    #登录
    login.login(self,"testingwtb","a123456")
    #写信
    driver.find_element_by_css_selector("#_mail_component_47_47 >
span.oz0").click()
    #填写收件人和主题
    driver.find_element_by_xpath("//*[@class='bz0']/div[2]
/div/input").send_keys('testingwtb@126.com')
    driver.find_element_by_xpath("//input[@class='nui-ipt-input' and
@type='text' and @maxlength='256']").send_keys(u'给小明的信')
    #发送邮件
    driver.find_element_by_xpath("//html/body/div[2]/div/div[2]/header
/div/div/div/span[2]").click()
    time.sleep(3)
    #断言发送结果
    text = driver.find_element_by_class_name('tK1').text
    self.assertEqual(text,u'发送成功')
    login.logout(self)

.....
```

在这个用例中，相比较前一条用例，增加了邮件主题的填写，填写邮件主题为“给小明的信”。有了收件人和主题就算是一封完整的邮件了，点击发送就可以发送成功了。

用例 003：

用例 id	test_sendmail.py	发送邮件	
步骤	动作	数据	验证点
1	打开邮箱	http://webcloud.126.com	
2	用户登陆	username, password	

3	写信	只输入收信人、主题和附件发送	发送成功
4	退出		

继续在 test_send.py 中添加用例：

test_send.py

```
.....
#填写收件人、主题和附件发送邮件
def test_send_mail3(self):
    driver = self.driver
    driver.get(self.base_url)
    #登录
    login.login(self,"testingwtb","a123456")
    #写信
    driver.find_element_by_css_selector("#_mail_component_47_47 >
span.oz0").click()
    #填写收件人和主题
    driver.find_element_by_xpath("//*[@class='bz0']/div[2]
/div/input").send_keys('testingwtb@126.com')
    driver.find_element_by_xpath("//input[@class='nui-ipt-input' and
@type='text' and @maxlength='256']").send_keys(u'给小明的信')
    #上传附件
    driver.find_element_by_class_name("O0").send_keys('D:\\upfile.txt')
    #发送邮件
    driver.find_element_by_xpath("/html/body/div[2]/div/div[2]/header
/div/div/div/span[2]").click()
    time.sleep(3)
    #断言发送结果
    text = driver.find_element_by_class_name('tK1').text
    self.assertEqual(text,u'发送成功')
    login.logout(self)

....
```

126 邮箱上传附件也是通过<input>标签实现的，所以可以通过 send_keys() 方法发送一个本地文件的绝对路径从而实现附件的上传。

用例 004：

用例 id	test_sendmail.py	发送邮件	
步骤	动作	数据	验证点
1	打开邮箱	http://webcloud.126.com	
2	用户登陆	username, password	
3	写信	只输入收信人、主题和正文发送	发送成功

test_send.py

```
.....
#填写收件人、主题和正文发送邮件
def test_send_mail4(self):
    driver = self.driver
    driver.get(self.base_url)
    #登录
    login.login(self,"testingwtb","a123456")
    #写信
    driver.find_element_by_css_selector("#_mail_component_47_47 >
span.oz0").click()
#填写收件人和主题
driver.find_element_by_xpath("//*[@class='bz0']/div[2]
/div/input").send_keys('testingwtb@126.com')
driver.find_element_by_xpath("//input[@class='nui-ipt-input' and
@type='text' and @maxlength='256']").send_keys(u'给小明的信')

#定位富文本表单
class_name = driver.find_element_by_class_name('APP-editor-iframe')
driver.switch_to_frame(class_name)
#编写邮件正文
driver.find_element_by_tag_name('body').send_keys(u'你好，小明好久不见。')
#断言发送结果
text = driver.find_element_by_class_name('tK1').text
self.assertEqual(text,u'发送成功')
login.logout(self)

.....
```

126 邮箱是一个富文本输入框，通过前端定位发现它是嵌套在页面里的 iframe。

Firebug

```
.....
<div class="APP-editor-edtr-mask" style="display:none"></div>
<iframe class="APP-editor-iframe" frameborder="0"
style="position:absolute" tabindex="1">
<html webdriver="true">
<head></head>
<body class="nui-scroll" contenteditable="true">邮件正文
</body>
</html>
</html>
```

</iframe>

.....

通过编写邮件正文内容发现，邮件的内容是写在 body 标签之间的，这和我们往常操作的 input 标签并不一样，但其实 send_key() 同样可以向 body 标签之间输入内容。如图



图 8.4 编写邮件正文

8.3.2 搜索邮件

用例 005：

用例 id	test_sendmail.py	发送邮件	
步骤	动作	数据	验证点
1	打开邮箱	http://webcloud.126.com	
2	用户登陆	username, password	
3	搜索邮件	输入搜索关键字	显示“搜索结果”
4	退出		

test_serach.py

.....

```
#搜索邮件
def test_search_mail(self):
    driver = self.driver
    driver.get(self.base_url)
    #调用登录模块
```

```

login.login(self,'testingwtb','a123456')

#搜索邮件
driver.find_element_by_xpath("//input[@class='nui-ipt-input' and
@type='text']").send_keys(u'小明')
driver.find_element_by_xpath("//input[@class='nui-ipt-input' and
@type='text']").send_keys(Keys.ENTER)

#断言搜索邮件标签页面
text= driver.find_element_by_xpath("//div[@id='dvMultiTab']/ul
/li[5]/div[3]").text
self.assertEqual(text,u'搜索邮件')

#调用退出
login.logout(self)

```

.....

搜索邮件的操作相对来说就简单了很多，登录邮箱，在搜索框内输入关键字“小明”进行搜索。注意，因为这里没有搜索按钮，所以使用，Keys 类所提供 Keys.ENTER 来完成对搜索结果的“回车”动作。从“搜索结果”判断是否搜索成功的字样。如



图 8.5 搜索邮件

8.3.3 删除邮件

用例 006：

用例 id	test_sendmail.py	发送邮件	
步骤	动作	数据	验证点
1	打开邮箱	http://webcloud.126.com	
2	用户登陆	username, password	
3	删除邮件	勾选第一封邮件删除	显示“已删除”
4	退出		

test_del.py

```
.....  
def test_del_mail(self):  
    driver = self.driver  
    driver.get(self.base_url)  
    #登录  
    login.login(self,'testingwtb','a123456')  
    driver.find_element_by_class_name('nui-tree-item-text').click()  
    time.sleep(2)  
    driver.find_elements_by_xpath("//span[@class='nui-chk-symbol']  
                                /b").pop(1).click()  
    try:  
        spans = driver.find_elements_by_tag_name('span')  
        for s in spans:  
            if s.text == u'删除':  
                s.click()  
    except:  
        pass  
  
    #断言是否已删除  
    text = driver.find_element_by_css_selector("br/>                                                span.nui-tips-text>a").text  
    self.assertEqual(text,u'已删除')  
    #退出  
    login.logout(self)  
.....
```

登录邮箱，首先点击“收件箱”查看邮件列表，通过 find_selements 查找一组 class='nui-chk-symbol' 的下面的 p 标签，也就是每一封邮件前面的复选框，通过 pop(1) 找到这一组复选框的第一个进行勾选。接着点击“删除”按钮，删除按钮的定位是通过一组 span 标签中找到 text 等于“删除”的元素。最终通过断言“已删除”的提示信息来判断是否删除成功。



图 8.6 删除邮件

小结：

在这一章中，我们对手工测试用例与自动化测试用例进行了简单的对比。然后通过 126 邮箱实战教大家如何把一个基本的测试项目搭建起来。接下来在项目的基础上扩展自动化测试用例。在用例的编写中回顾了前面所学到的一些元素定位与操作的技巧。

通过这一章的学习，相信大家已经可以开展测试工作了。至于用例编写的技巧要通过不段的实践来提高。

第9章 自动化测试高级应用

到目前位置已经可以开展自动化测试工作了，但是既然要做自动化测试，那就让测试真正的“自动化”起来。下面要学的技术点可以让我们当前的自动化项目更加完善。

9.1 使用 HTMLTestRunner 生成测试报告

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 单元测试框架的一个扩展。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>

HTMLTestRunner.py 本是一个.py 文件，将它放到 Python 安装目录下即可调用。

Windows：将下载的文件放入...\\Python27\\Lib 目录下。

Linux (ubuntu)：下需要先确定 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。

Ubuntu 终端

```
fnngj@fnngj-pc:~$ Python
Python 2.7.6 (default, Mar 22 2014, 22:59:38)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', '/usr/local/lib/Python2.7/dist-packages/sleekxmpp-1.2.4-py2.7.egg',
 '/usr/local/lib/Python2.7/dist-packages/peewee-2.3.3-py2.7.egg',
 '/usr/lib/Python2.7', '/usr/lib/Python2.7/plat-i386-linux-gnu',
 '/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/PILcompat',
 '/usr/lib/Python2.7/dist-packages/gst-0.10',
 '/usr/lib/Python2.7/dist-packages/gtk-2.0', '/usr/lib/pymodules/Python2.7']
```

以 root 身份将 HTMLTestRunner.py 文件考本到 </usr/lib/Python2.7/dist-packages/> 目录下：

Ubuntu

```
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。

Python Shell

```
>>> import HTMLTestRunner
>>>
```

9.1.1 生成 HTMLTestRunner 测试报告

下面继续以 test_baidu.py 文件为例生成 HTMLTestRunner 测试报告：

test_baidu.py

```
#coding=utf-8
from selenium import webdriver
import unittest, time
import HTMLTestRunner #引入 HTMLTestRunner 包

class Baidu(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(10)
        self.base_url = "http://www.baidu.com/"
        selfverificationErrors = []

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

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

if __name__ == "__main__":
```

```

#测试套件
testunit=unittest.TestSuite()

#添加测试用例到测试套件中
testunit.addTest(Baidu("test_baidu_search"))

#定义个报告存放路径
filename = 'E:\\test_object\\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)
#关闭报告文件
fp.close()

```

代码分析：

首先将 HTMLTestRunner 模块 import 进来。定义测试报告的存放路径 filename，通过 file() 将文件以读写的方式打开。

接着调用 HTMLTestRunner 模块下的 HTMLTestRunner 方法。stream 指定测试报告文件；title 用于定义测试报告的标题；description 用于定义测试报告的副标题。

现在通过 HTMLTestRunner 的 run() 方法来运行测试套件中所组装的测试用例。最后 fp.close() 来关闭测试报告文件。

用例运行完成，打开“E:\\test_object\\report\\result.html”文件查看生成测试报告，如图 9.1

百度搜索测试报告

Start Time: 2015-01-24 17:17:13

Duration: 0:00:13.841000

Status: Pass 1

用例执行情况：

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

Test Group/Test case	Count	Pass	Fail	Error	View
Baidu	1	1	0	0	Detail
test_baidu_search			pass		
Total	1	1	0	0	

图 9.1 测试报告

9.1.2 更易读的测试报告

虽然在编写测试用例（方法）时我们为每一个用例添加了注释，但这个注释信息只是在阅读代码时才能看到，那么如何给测试报告中的用例添加注释，用来说明“test_baidu_search”是一个什么测试用例这是我们这节所要解决的问题。

在此有之有我们先来学习一个技巧，通过 Python 提供的 help() 来查看类和方法的说明。

Python Shell

```
>>> import HTMLTestRunner
>>> help(HTMLTestRunner)
Help on module HTMLTestRunner:

NAME
    HTMLTestRunner

FILE
    c:\Python27\lib\htmltestrunner.py

DESCRIPTION
    A TestRunner for use with the Python unit testing framework. It
    generates a HTML report to show the result at a glance.
```

The simplest way to use this is to invoke its main method. E.g.

```
import unittest
import HTMLTestRunner

... define your tests ...

if __name__ == '__main__':
    HTMLTestRunner.main()
```

For more customization options, instantiates a HTMLTestRunner object. HTMLTestRunner is a counterpart to unittest's TextTestRunner. E.g.

```
# output to a file
fp = file('my_report.html', 'wb')
runner = HTMLTestRunner.HTMLTestRunner(
    stream=fp,
    title='My unit test',
```

```

        description='This demonstrates the report output by
HTMLTestRunner.'
    )

# Use an external stylesheet.
# See the Template_mixin class for more customizable options
runner.STYLESHEET_TMPL = '<link rel="stylesheet"
href="my_stylesheet.css" type="text/css">'

# run the test
runner.run(my_test_suite)
.....

```

首先 import 相关类或方法，通过 help() 方查看类和方法的说明信息。这里以 HTMLTestRunner 为例。优秀的开发者所开发的类和方法会在代码块中提供足够说明信息，让使用者可以通过说明信息就能完全掌握类和方法的使用。显现 HTMLTestRunner 的开发者这一点做得很规范。因为它提供的足够的信息来告诉我们 HTMLTestRunner 的使用。

那么代码中什么样的信息可以通过 help() 看到呢？打开 HTMLTestRunner.py 可以看到这些信息通过三引号（“““” 或 ‘‘‘‘’）括起来的注释信息。在 Python 中三引号用于多行注释，这样的注释可以通过 help() 查看。

下面我们来定义并查看这些注释。

model.py

```

#coding=utf-8
u"""
这是我们自定义的一个模块，这个模块用于完成计算。
"""

def add(a,b):
    u"""用于计算两数相加"""
    return a+b

```

通过 help() 方法查看创建的 model.py 文件。

model_.py

```

#coding=utf-8
import model

help(model)

```

执行结果：

```

>>> ===== RESTART =====
>>>

```

Help on module model:

NAME

model - 这是我们自定义的一个模块，这个模块用于完成计算。

FILE

c:\users\fnngj\desktop\model.py

FUNCTIONS

add(a, b)

用于计算两数相加

通过上面的例子可以看到在 add() 方法下面所定义的注释，这个注释信息在调用 add() 函数时不会显示。可以通过 help() 方法查看。在 test_baidu_search() 方法中添加注释。

test_baidu.py

.....

#百度搜索用例

```
def test_baidu_search(self):
    u'''百度搜索用例'''
    driver = self.driver
    driver.get(self.base_url)
    driver.find_element_by_id("kw").send_keys("HTMLTestRunner")
    driver.find_element_by_id("su").click()
```

.....

再次运行测试用例，查看测试报告。如图 9.2

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

Test Group/Test case	Count
Baidu	1
test_baidu_search: 百度搜索用例	
Total	1

图 9.2 测试报告

9.1.3 测试报告文件名

每次运行测试之前都要手动的去修改报告的名称，如果忘记修改就会把之前的报告覆盖，这样做显然会麻烦，那么有没有办法使每次生成的报告名称都不一样并且更有意义，我们可以在报告名称中加入当前时间，这样报告不会重叠并且更清晰的知道生成的前后时间。

Python 所提供的 time 模块中有许多关于时间的方法。可以利用这些方法来完成这个需求。

model_.py

```
>>> ===== RESTART =====
>>>
>>> import time

>>> time.time()
1422099225.566

>>> time.ctime()
'Sat Jan 24 19:33:53 2015'

>>> time.localtime()
time.struct_time(tm_year=2015, tm_mon=1, tm_mday=24, tm_hour=19, tm_min=34,
tm_sec=9, tm_wday=5, tm_yday=24, tm_isdst=0)

>>> time.strftime("%Y_%m_%d %H:%M:%S")
'2015_01_24 19:40:05'
```

[time.time\(\)](#) 获取当前时间戳。

[time.ctime\(\)](#) 当前时间的字符串形式。

[time.localtime\(\)](#) 当前时间的 struct_time 形式。

[time.strftime\(\)](#) 用来获得当前时间，可以将时间格式化为字符串。

Python 中时间日期格式化符号（区分大小写）：

Directive	Meaning
%a	星期几的简写
%A	星期几的全称
%w	十进制表示的星期几（值从 0 到 6，星期天为 0）
%d	十进制表示的每月的第几天
%b	月份的简写
%B	月份的全称
%m	十进制表示的月份
%y	不带世纪的十进制年份（值从 0 到 99）
%Y	带世纪部分的十进制年份
%H	24 小时制的小时
%I	12 小时制的小时
%p	本地的 AM 或 PM 的等价显示
%M	十时制表示的分钟数

Directive	Meaning
%S	十进制的秒数
%f	十进制的微秒，零填充左边
%Z	当前时区的名称
%j	十进制表示的每年的第几天
%U	一年中的星期数（00-53）星期天为星期的开始
%W	一年中的星期数（00-53）星期一为星期的开始
%x	本地相应的日期表示
%X	本地相应的时间表示
%%	%号本身

打开..\\test_test.py 做如下修改：

test_baidu.py

```
.....
if __name__ == "__main__":
    testunit=unittest.TestSuite()
    testunit.addTest(Baidu("test_baidu_search"))

    #获取当前时间
    now = time.strftime("%Y-%m-%d %H_%M_%S")
    #定义个报告存放路径
    filename = 'E:\\test_object\\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：')

    #运行测试用例
    runner.run(testunit)
    #关闭报告文件
    fp.close()
```

通过 strftime()方法以指定的格式获取当时间，将当前时间的字符串赋值给 now 变量。将 now 通过加号 (+) 拼接到生成的测试报告的文件名中。再次运行测试用例，生成的测试报告文件名如图 9.3。

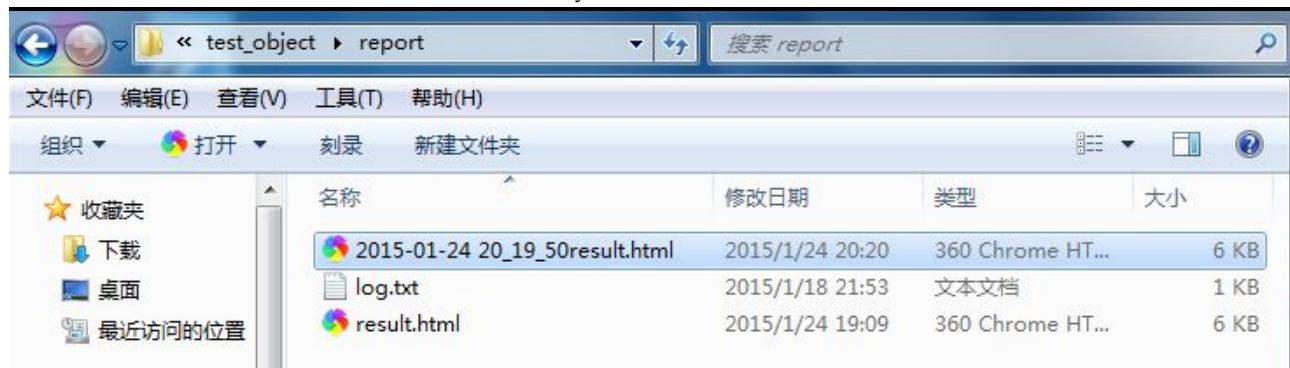


图 9.3 测试目录

目前测试报告只集成到了单个测试文件中，我们的最终目的是将其集成表 all_test.py 文件中。下面打开 all_test.py 文件，做如下修改：

```
all_test.py
.....
#coding=utf-8
import unittest
import HTMLTestRunner
import time

def creatsuite():
    testunit=unittest.TestSuite()
    #定义测试文件查找的目录
    test_dir='E:\\test_object\\test_case'
    #定义 discover 方法的参数
    discover=unittest.defaultTestLoader.discover(test_dir,pattern='test*.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_case in discover:
        print test_case
        testunit.addTests(test_case)
    return testunit

now = time.strftime("%Y-%m-%d %H_%M_%S")
filename = 'E:\\test_object\\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：')

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

```

alltestnames = creatsuite()
runner.run(alltestnames)
fp.close()

```

查看生成的 HTML 测试报告：

百度搜索测试报告

Start Time: 2015-01-24 21:03:03

Duration: 0:03:56.167000

Status: Pass 8 Failure 1 Error 1

用例执行情况：

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

Test Group/Test case	Count	Pass	Fail	Error	View
test_alogin.TestLogin	4	4	0	0	Detail
test_error: 用户名密码错误			pass		
test_null: 用户名、密码为空			pass		
test_pawd_null: 输入用户名、密码为空			pass		
test_user_null: 用户名为空, 只输入密码			pass		
test_delete.TestDel	1	1	0	0	Detail
test_del_mail: 删除邮件			pass		
test_send.TestSend	4	2	1	1	Detail
test_send_mail: 只填写收件人发送邮件			fail		
test_send_mail2: 填写收件人、主题发送邮件			pass		
test_send_mail3: 填写收件人、主题和附件发送邮件			pass		
test_send_mail4: 填写收件人、主题和正文发送邮件			error		
test_sreach.TestSearch	1	1	0	0	Detail
test_search_mail: 搜索邮件			pass		
Total	10	8	1	1	

图 9.4 项目测试报告

9.2 创建定时任务

为了让自动化测试“自动化”起来，现在我们来创建定时任务，使自动化测试脚本在指定的时间自动化运行。创建定时任务的方法有很多，比如，我们可以写一段程序让其在指定的时间运行 all_test.py 文件，或者使用系统的定时任务功能在指定的时间运行 all_test.py 文件。

9.2.1 通过程序创建定时任务

现在我们来回顾一下运行 Python 程序 (all_test.py) 有多少种方式。如果是通过 Python 自带的编辑器 IDLE (Python GUI) 的话 F5 运行，如果通过 sublime 编辑器的话按 Ctrl+B 运行，在 cmd 命令提示器的话通过“Python all_test.py”命令执行。

在 Python 的 os 模块中提供了 system() 来执行系统命令。比如我们要执行 E:\\\\test_object\\\\ 目录下的 all_test.py 文件，可以这样来实现：

```
start_run.py
```

```
#coding=utf-8
import os

os.system('E:\\test_object\\all_test.py')
```

或者我们先切换到相应的目录，然后通过 Python 命令去执行 all_test.py 文件更符合我们在 cmd 下面的操作顺序。

start_run.py

```
import os

os.chdir("E:\\test_object")
os.system('Python all_test.py')
```

chdir()用切换到某个目录下，当相于 shell 下的“cd”命令，system 可执行任意 shell 或 DOS 命令。

好吧！回到问题的出发点上我们的重点是定时任务，我们前面已经学过 time 模块了，要实现这个功能非常简单，获取当前时间判断是不是要执行自动化的时间。

start_run.py

```
#coding=utf-8
import os,time

k=1
while k <2:
    now_time=time.strftime('%H_%M')
    if now_time == '21_00':
        print u"开始运行脚本:"
        os.chdir("E:\\test_object")
        os.system('Python all_test.py') #执行脚本
        print u"运行完成退出"
        break
    else:
        time.sleep(10)
        print now_time
```

程序的实现很简单，首先定义变量 k 的值为 1，通过 while 判断 k 的值是否小于 2，在不改变 k 的值的情况下，k 会永远小于二，然后通过 strftime()方法获取当前的小时和分钟，然后通过 if 判断是否等于 21:00，如果不相等，休眠 10 秒（这个休眠时间只要不超过 60 秒即可，超过了 60 秒，可能直接会从 20:59 跳到 21:01，从而错过了 20:00），当前时间为 21:00 时执行 all_test.py 程序从而执行自动化测试。并且 break 结束循环。

或者你想每天 20:00 跑自动化测试用例，那么可以不要 break，程序继续运行，直到遇见下一个 20:00，自动化测试再次被执行。运行过程如图 9.5。

```

C:\Users\fnngj\Desktop>python start_run.py
20_58
20_58
20_59
20_59
20_59
20_59
20_59
20_59
开始运行脚本:
<unittest.suite.TestSuite tests=[<unittest.suite.TestSuite tests=[<test_send.Tes
tSend testMethod=test_send_mail>, <test_send.TestSend testMethod=test_send_mail2>
, <test_send.TestSend testMethod=test_send_mail3>, <test_send.TestSend testMeth
od=test_send_mail4>]]>
<unittest.suite.TestSuite tests=[<unittest.suite.TestSuite tests=[<test_sreach.T
estSearch testMethod=test_search_mail>]]>
...
Time Elapsed: 0:03:05.754000
运行完成退出

C:\Users\fnngj\Desktop>

```

图 9.5 运行 start_up.py 程序

Python 身为脚本语言处理此类问题是它强项，所以不用怀疑它这方法的能力，如果有类似这方面需求可以进一步学习。

9.2.2 Windows 添加任务计划

哪果时刻都在电脑上开这么一个程序，会不会觉得比较“碍事”呢！？其实操作系统本身也提供了强大的定时任务功能。我们先以 Windows 系统为例介绍如何设置定时任务。根据当前市场 Windows 7 市场占有率最高，所以我们以 Windows 7 为例。

通过控制面板-->管理工具-->任务计划程序（或者在“开始”菜单中搜索“任务计划程序”）：

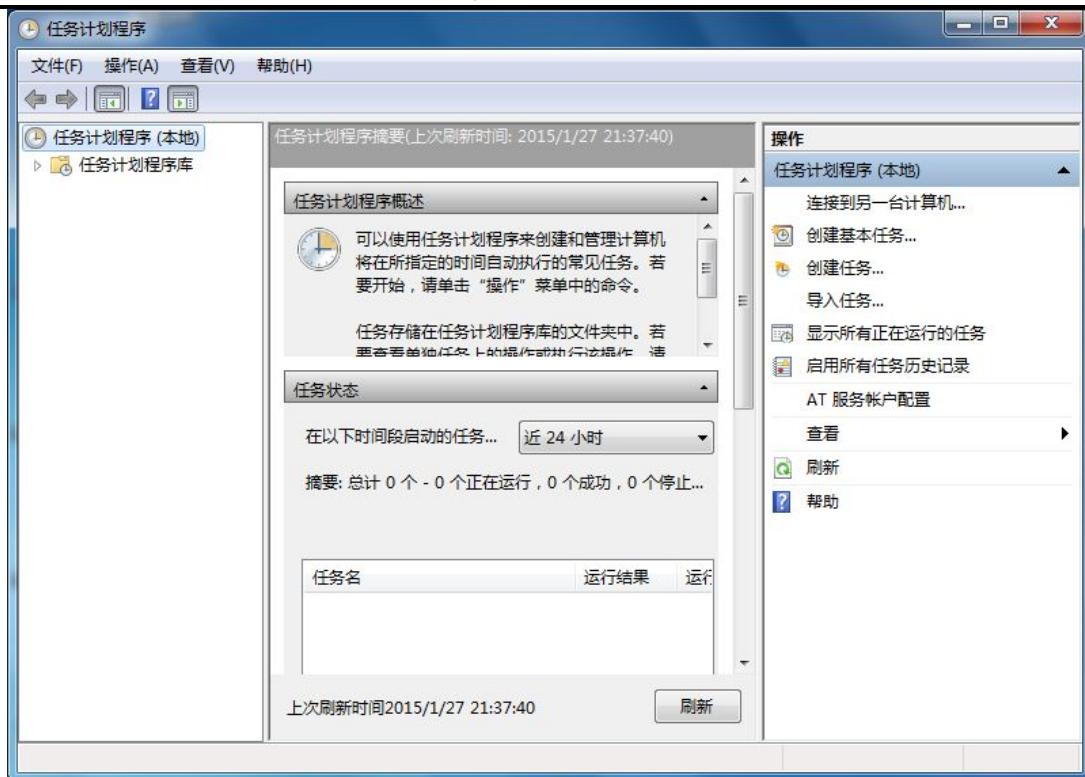


图 9.6 任务计划程序

选择菜单栏“操作”选项中可以选择“创建基本任务”和“创建任务”，前者创建过程比较简单，我们选择后者进行设置。

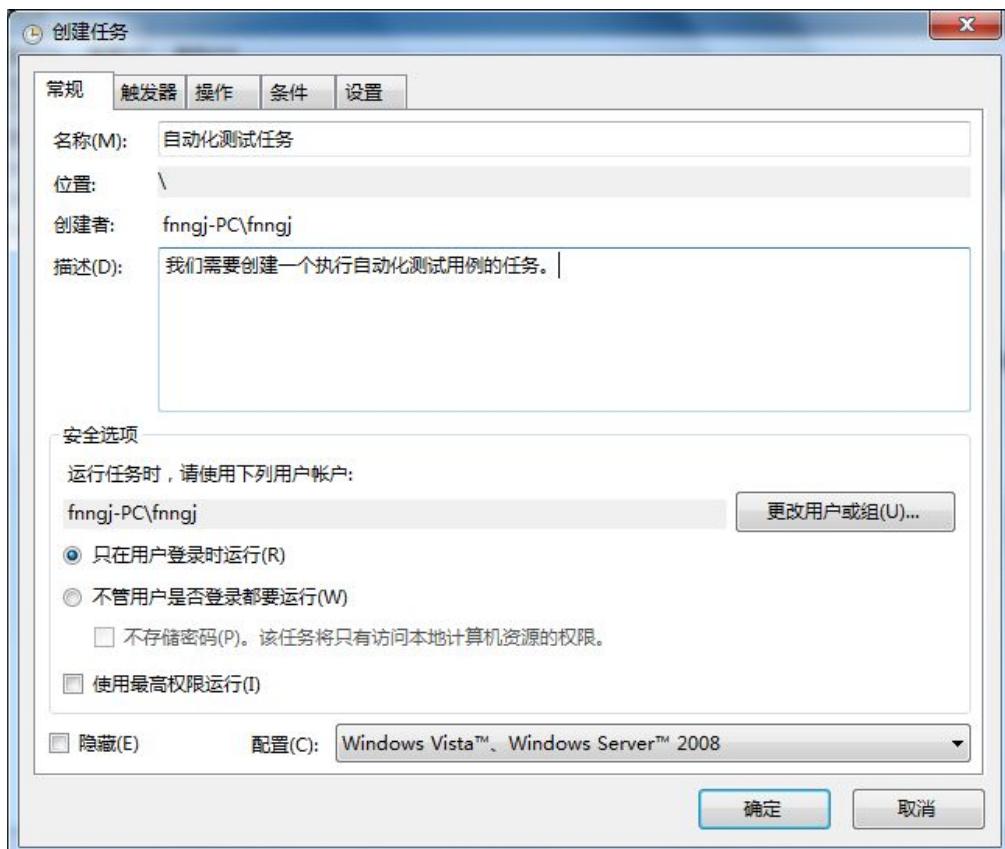


图 9.7 创建任务

如图 9.7 创建任务，设置任务名称和任务描述，以及执行任务的用例。切换到“触发器”标签页，点击“新建”设置出发任务的条件。

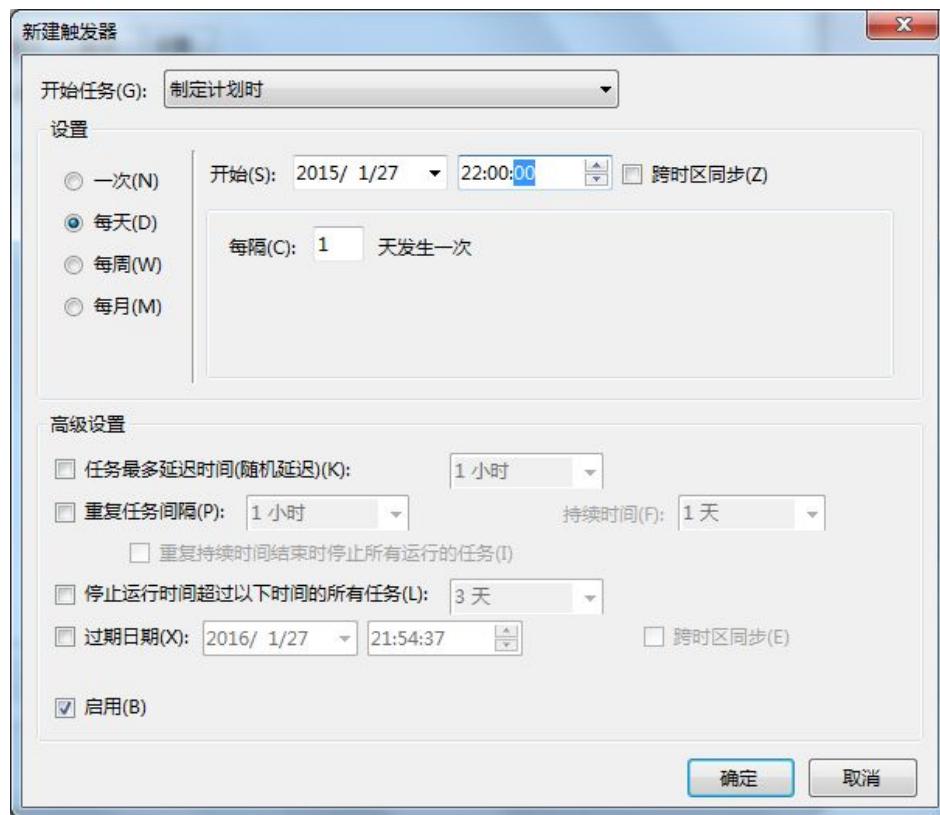


图 9.8 新建触发器

如图 9.8 设置每天 22:00:00 触发任务，然后点击“确定”。然后，切换到“操作”标签页：点击“新建”设置执行的操作。

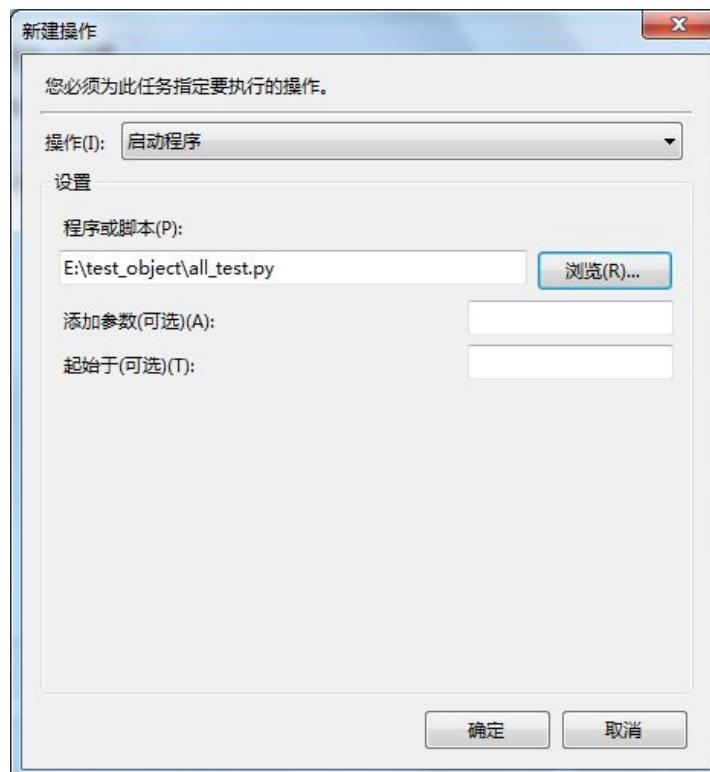


图 9.10 新建操作

如图 9.10 选择运行程序或脚本，点击“浏览”按钮找到 all_test.py 程序，点击“确定”。然后，我们就创建完成了一个任务计划，在任务计划库中就可以看到创建的任务计划了。

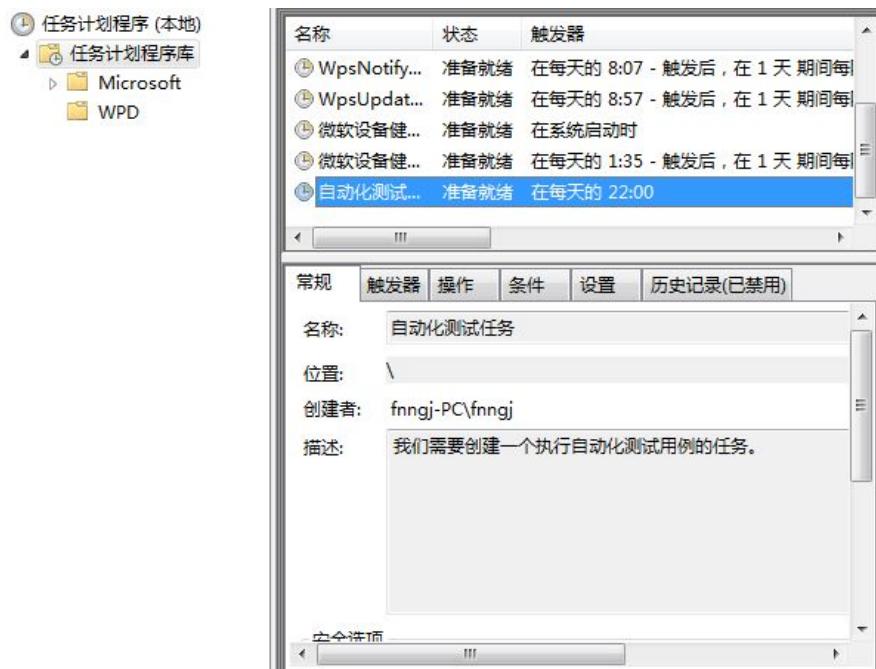


图 9.11 查看的任务计划

OK，现在我们可以验证一下任务设置的是否成功，在任务计划上右键选择“运行”，任务就可以开始运行了。

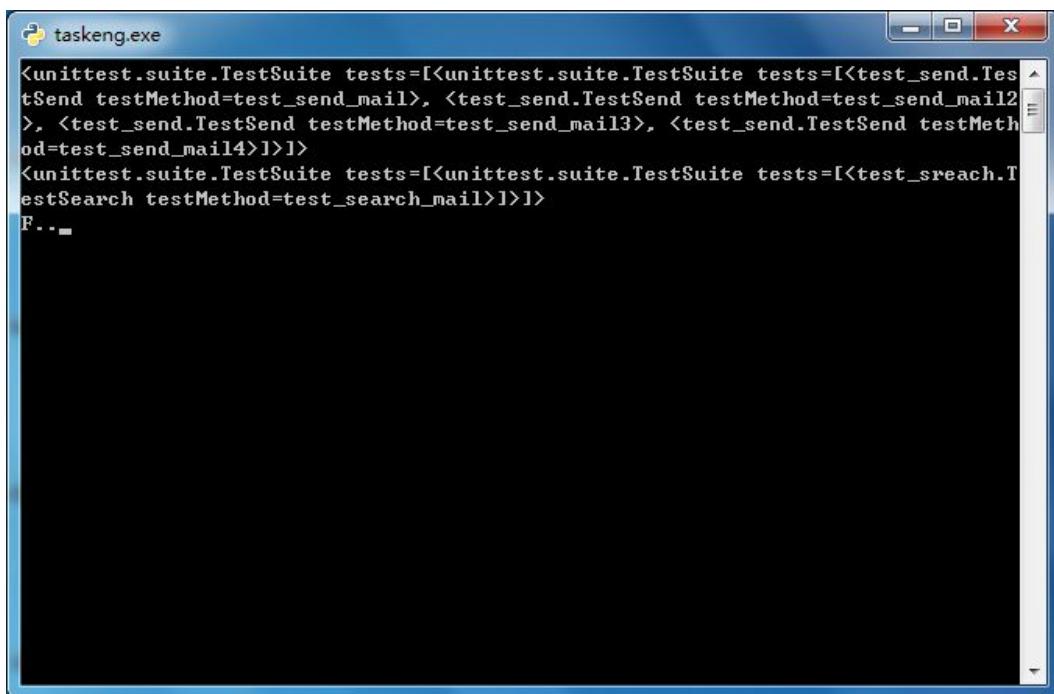


图 9.12 任务运行窗口

9.2.2 Linux 添加任务计划

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。这里我们同样选择使用比较广泛的 Ubuntu 为例。

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。同样在当前系统下创建 test_object 测试项目。

[/home/fnngj/test_object/all_test.py](#)

Ubuntu 终端

```
fnngj@fnngj-pc:~/test_object$ pwd
/home/fnngj/test_object
fnngj@fnngj-pc:~/test_object$ ls
all_test.py  report  test_case  test_date

fnngj@fnngj-pc:~/test_object$ at now+5 minutes
warning: commands will be executed using /bin/sh
at> Python /home/fnngj/test_object/all_test.py
at> <EOT>
job 1 at Tue Jan 27 23:08:00 2015
```

now+5 minutes now 表示当前时间的5分钟之后执行，回车，设置要执行的文件。

Python /home/fnngj/test/fiele.py 通过 Python 命令设置要执行的 all_test.py 的完整路径。

Ctrl+d 保存退出，提示：2015年1月27日23:08:00创建了第一个工作任务。

查看创建的任务

Ubuntu 终端

```
fnngj@fnngj-pc:~/test_object$ at -l
1      Tue Jan 27 23:08:00 2015 a fnngj

fnngj@fnngj-pc:~/test_object$ atq
1      Tue Jan 27 23:08:00 2015 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

Ubuntu 终端

```
fnngj@fnngj-pc:~/test_object$ at -l  
8       Wed Jan 28 23:40:00 2015 a fnngj  
10      Thu Jan 29 23:31:00 2015 a fnngj  
9       Wed Jan 28 23:42:00 2015 a fnngj  
  
fnngj@fnngj-pc:~/test_object$ atrm 9 --删除任务  
  
fnngj@fnngj-pc:~/test_object$ at -l  
8       Wed Jan 28 23:40:00 2015 a fnngj  
10      Thu Jan 29 23:31:00 2015 a fnngj
```

通过 atrm 命令删除定时任务的编号。

启动 atd 进程

linux 一般默认会启动 atd 进程，如果没有启动，可以手动将进程开启。

Ubuntu 终端

```
fnngj@fnngj-pc:~/test_object$ ps -ef|grep atd  
daemon     814      1  0 22:27 ?          00:00:00 atd  
fnngj     5995    5683  0 23:35 pts/6      00:00:00 grep --color=auto atd  
  
fnngj@fnngj-pc:~/test_object$ /etc/init.d/atd status  #启动进程  
* atd is running
```

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
mow+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
```

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。

下面通过 crontab 来创建任务，通过 rontab -e 命令进入 crontab 文件。

Ubuntu 终端

```
fnngj@fnngj-pc:~/test_object$ crontab -e

# 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/
#
# For more information see the manual pages of crontab(5) and cron(8)
#
# m h dom mon dow   command
3 * * * *     Python /home/fnngj/test_object/all_test.py
```

默认通过 vi/vim 打开了 crontab 文件，关于 vi/vim 的学习与使用参考 linux 相关书籍。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。

Crontab 任务格式如下：

m	H	Dom	Mon	Dow	command
分钟	小时	天	月	星期	命令/脚本
*	21	*	*	*	Python /home/fnngj/test_object/all_test.py

上面的任务表示：每天的21:00，执行 all_test.py 程序。

任务编写完成。按键盘 Esc 退出编辑模式，输入 “:wq!” 保存并退出文件。

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：

Ubuntu 终端

```
root@fnngj-pc:~/test_object$ cd /var/spool/cron/crontabs/
root@fnngj-pc:~/var/spool/cron/crontabs$ ls
fnngj
root@fnngj-pc:~/var/spool/cron/crontabs$ cat fnngj
.....
# m h dom mon dow   command
3 * * * *     Python /home/fnngj/test_object/all_test.py
```

注意：在查看 crontab 任务计划时必须要用 root 用户，否则会提示权限不足。

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 9.13 crontab 格式说明

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：

假如，我们每天早上4点跑自动化测试用例。

m	H	Dom	Mon	Dow	command
分钟	小时	天	月	星期	命令/脚本
*	4	*	*	*	Python /home/fnngj/test_object/all_test.py

假如，我们每周一和三下午的6点运行自动化测试用例：

m	H	Dom	Mon	Dow	command
分钟	小时	天	月	星期	命令/脚本
*	18	*	*	1, 3	Python /home/fnngj/test_object/all_test.py

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

m	H	Dom	Mon	Dow	command
分钟	小时	天	月	星期	命令/脚本
30	17	*	*	1-5	/usr/bin/wall < /hz/h/test/close.wall
45	17	*	*	1-5	/usr/bin/shutdown -h now

9.3 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@126.com，那就直接连接126.com 的邮件服务器，把信投給 zzz@126.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@126.com，你不是直接投到126.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126.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。

9.3.1 发送 HTML 格式的邮件

send_mail.py

```
#coding=utf-8
import smtplib
from email.mime.text import MIMEText
from email.header import Header

#发送邮箱
sender = 'testingwtb@126.com'
#接收邮箱
receiver = 'xiaoming@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 = 'testingwtb@126.com'
password = '123456'

#编写 text 类型的邮件正文
msg = MIMEText ('<html><h1>你好! </h1></html>', 'html', '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()

```

发送邮件的实现本身没有什么逻辑性，建立在我们发电子邮件的理解上，如果你使用过电子邮件，那么一定自己的邮箱地址，如果要发送邮件还需要对方的邮箱地址，在编写邮件的过程中需要填写邮件主题、正文，或者有时还要加个附件。如果使用你平时使用的是邮箱客户端，如 foxmail 等，那么在首次使用的时候，还要配置邮箱服务器，一般需要用的 SMTP/POP3 协议。好吧，知道这些就已经足够了。上面的代码其实也就是在配置这些。

```
import smtplib
```

导入 smtplib 发邮件模块，程序中邮件的发送、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（xiaoming@126.com），会看一封发来的邮件。如图 9.14。



图 9.14 阅读 HTML 邮件

9.3.2 发送带附件的邮件

有时在发送文件时，需要发送附件。下面实例实现带附件的邮件。

send_mail.py

```
#coding=utf-8
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart

#发送邮箱
sender = 'testingwtb@126.com'
#接收邮箱
receiver = 'xiaoming@126.com'
#发送邮箱服务器
smtpserver = 'smtp.126.com'
#发送邮箱用户 / 密码
username = 'testingwtb@126.com'
password = '123456'

msgRoot = MIMEMultipart('related')
#邮件主题
msgRoot['Subject'] = 'Python email test'

#构造附件
att = MIMEText(open('E:\\test_object\\report\\log.txt', 'rb').read(), 'base64',
'utf-8')
att["Content-Type"] = 'application/octet-stream'
att["Content-Disposition"] = 'attachment; filename="log.txt"'
```

```

msgRoot.attach(att)

smtp = smtplib.SMTP()
smtp.connect('smtp.126.com')
smtp.login(username, password)
smtp.sendmail(sender, receiver, msgRoot.as_string())
smtp.quit()

```

与上一个实例相比，通过 `MIMEMultipart` 模块构造带附件的邮件，附件本身同样使用 `MIMEText()` 方法定义路径和格式。运行程序，将看到如图 9.14。



图 9.15 带附件的邮件

9.3.3 查找最新的测试报告

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测试报告。

```

new_file.py

#coding=utf-8
import os

#定义文件目录
result_dir = 'E:\\test_object\\report'

lists=os.listdir(result_dir)

#重新按时间对目录下的文件进行排列
lists.sort(key=lambda fn: os.path.getmtime(result_dir+"\\"+fn))

print ('最新的文件为:  '+lists[-1])
file = os.path.join(result_dir,lists[-1])
print file

```

首先定义测试报告的目录 `result_dir`，通过 `os.listdir()` 可以获取目录下的所有文件。这也只是获取一个目录下的所有文件而已，我们最重要的任务是找到目录下最新生成的文件。

```
lists.sort(key=lambda fn: os.path.getmtime(result_dir+"\\"+fn))
```

在这短短的一行代码中包含了不少的 Python 知识，下面我们来逐一分析这段代码中的知识点。

1、lambda 函数

Lambda 函数又叫匿名函数，下面通过一个例子来看普通函数和匿名函数的区别：

Python Shell

```
#普通函数
>>> def fun(x):
    return x*2

>>> print fun(5)
10

#匿名函数
>>> f = lambda x:x*2
>>> print f(5)
10
```

用 `lambda` 来声明匿名函数，冒号（`:`）前面 `x` 表示入参，冒号后面 `x*2` 表示返回值。`lambda` 和普通的函数相比，省去了函数名称，“匿名函数”也是由此而来。

2、sort 方法

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。

Python Shell

```
>>> ls = ['cc', 'dd', 'bb', 'aa']

#对数据进行排序
>>> ls.sort()

#打印排序后的数组
>>> print ls
['aa', 'bb', 'cc', 'dd']
```

`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：

Python Shell

```
#定位一个数组
>>> ls=['c.txt', 'b.txt', 'd.txt', 'a.txt']
>>> ls
['c.txt', 'b.txt', 'd.txt', 'a.txt']

#取数组中的 key 做排序
>>> ls.sort(key=lambda lists:lists[0])
>>> print ls
['a.txt', 'b.txt', 'c.txt', 'd.txt']

>>> ls.sort(key=lambda lists:lists[1])
>>> print ls
['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。

3、`getmtime` 方法

`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）

通过上面一行代码的作作，`lists` 中的文件按照进间重时行的排列组合。如果想获最新的文件就取 `lists[-1]`。

通过 `os.path.join()` 方法可以将文件目录与文件名进行拼接，从而打印出目录下最新文件的绝对路径。这正是我们想要的，执行 `new_file.py` 文件，打印结果如下：

Python Shell

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

最新的文件为： 2015-01-27 22_10_14result.html
E:\test_object\report\2015-01-27 22_10_14result.html

9.3.4 整合自动发邮件功能

通过前面的学习，我们已经学会了两个知识点，如何利用 Python 实现发邮件功能，如何查找目录下最新的文件。下面要做的事情就是把这些功能整合到 all_test.py 文件。

all_test.py

```
#coding=utf-8
import smtplib
from email.mime.text import MIMEText
import unittest
import HTMLTestRunner
import time,os

#=====定义发送邮件=====
def send_mail(file_new):
    #发信邮箱
    mail_from='testingwtb@126.com'
    #收信邮箱
    mail_to='xiaoming@126.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"自动化测试报告"
    #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('testingwtb@126.com','123456')
    smtp.sendmail(mail_from,mail_to,msg.as_string())
    smtp.quit()
    print 'email has send out !'

#=====查找测试报告目录，找到最新生成的测试报告文件=====
```

```

def send_report(testreport):
    result_dir = testreport
    lists=os.listdir(result_dir)
    lists.sort(key=lambda fn: os.path.getmtime(result_dir+"\\"+fn))
    #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
    send_mail(file_new)

#=====将用例添加到测试套件=====
def creatsuite():
    testunit=unittest.TestSuite()
    #定义测试文件查找的目录
    test_dir='.\test_case'
    #定义discover方法的参数
    discover=unittest.defaultTestLoader.discover(test_dir,pattern
='test*.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_case in discover:
        print test_case
        testunit.addTests(test_case)
    return testunit

if __name__ == '__main__':
    now = time.strftime("%Y-%m-%d %H_%M_%S")
    testreport = 'E:\test_object\report\\'
    filename = testreport+now+'result.html'
    fp = file(filename, 'wb')
    runner =HTMLTestRunner.HTMLTestRunner(
        stream=fp,
        title=u'自动化测试报告',
        description=u'用例执行情况：')

    alltestnames = creatsuite()
    runner.run(alltestnames)
    fp.close() #关闭生成的报告
    send_report(testreport) #发送报告

    send_mail(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。

```
send_report()
```

定义 sendreport() 接收一个测试报告的目录，找到目录下最新生成的测试报告文件 file_new。调用并将 file_new 传给 send_mail() 函数。

程序执行过程：

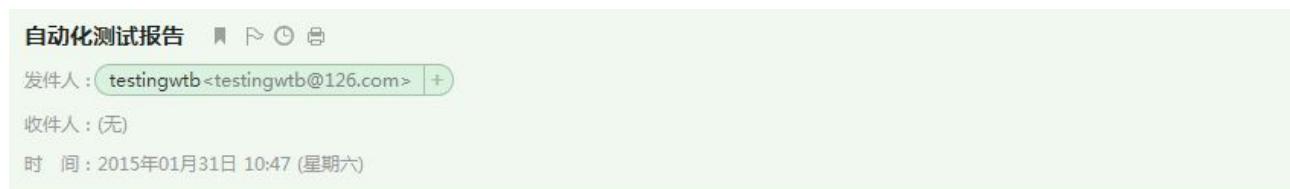
执行 all_tests.py 文件，执行过程如下：

首先，调用 createsuit() 函数，将所测试用例装载到测试套件中。

其次，调用 HTMLTestRunner 模块的 run() 函数，执行测试套件中的用例，并生成测试报告，然后，fp.close() 关闭生成的测试报告文件。

接着，调用 send_report() 函数，查找测试报告目录下面最新生成的测试报告文件，并将最新报告文件的路径传给 send_mail() 函数，send_mail() 函数实现发邮件功能。

整个过程执行完成，打开我们的接收邮箱将看到最新测试执行的测试报告。如图 8.3



自动化测试报告

Start Time: 2015-01-31 10:47:04

Duration: 0:00:16.760000

Status: Pass 1

用例执行情况：

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

Test Group/Test case	Count	Pass	Fail	Error	View
test_sreach.TestSearch	1	1	0	0	Detail
Total	1	1	0	0	

图 9.16 查看邮箱中的自动化测试报告

小结：

本章讲解了三个知识点对自动化测试项目的进行了增加，首先引入 HTMLTestRunner 第三方模块，可以使 unittest 单元测试结果生成更易读的 HTML 报告。在定时任务中，分别介绍了通过 Python 程序和 Windows、Linux 系统所提供的定时任务的方法。在自动发邮件的小节中学会了如何 Python 实现一个发邮件的功能。

第 10 章 Selenium Grid2

在 Selenium 家族中，我们已经学习了 WebDriver 和 IDE，这一章介绍最后一位成员 Grid。利用 Grid 可以在不同的主机上建立主节点（hub）和分支节点（node）。使主节点上的测试用例在不同的分支节点上运行。对于不同的节点来说，可以搭建不同的环境（浏览器/操作系统），从而使一份测试用例完成在不同环境下的验证。

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。

10.1 Selenium 工作原理

Selenium 分 1.0 与 2.0 两个版本，这在本书第一章中已经介绍了，简单区分一下这两个版本工作原理。

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 Selenium1.0 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 JavaScript 调用，即非同源不可调用；而 selenium1.0 中的工作方式就是在宿主页面注入 JavaScript 并且通过调用 JavaScript 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 JavaScript 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 JavaScript 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 Selenium1.0 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 JavaScript 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(RC) 代码

我们可以通过 Selenium IDE 将录制的脚本导出为“Python2/unittest/Remote Control”格式。通过编辑器打开导出的脚本。

Selenium 1(RC) 代码：

se_rc.py

```
#coding=utf-8
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.verificationErrors)

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

与 WebDrover 最大的不同是 RC 在执行指定代理的主机：

```
self.selenium = selenium("localhost", 4444, "*chrome", "http://www.baidu.com/")
```

localhost 表示本机；

4444 表示端口号；

*chrome 表示所运行的浏览器；

http://www.baidu.com 访问的 RUL 地址。

RC 的运行依赖于 Selenium Server； Selenium Server 是由 java 开发的一个 jar 包。所以，要想启动 Selenium Server 必须先安装 java 环境。

10.2 Selenium Server 环境配置

前面已经介绍 Selenium Grid2 已经集成到 Selenium Server 中，所以要想使用 Selenium Grid2 必须启动 Selenium Server。下面我们就来配置 Selenium Server。

第一步、 配置 java 环境

java 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。

打开下载链接选择相应的版本进行下载。我们以 Windows 安装 JDK 为例，



图 10.1 安装 JDK

双击下载的 JDK，设置安装路径。这里我们选择默认安装在 C:\Program Files\Java\jdk1.7.0_45\ 目录下。

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→新建..

变量名: JAVA_HOME
变量值: C:\Program Files\Java\jdk1.7.0_45\
变量名: CLASSPATH
变量值: .;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar;

找到 path 变量名→“编辑”添加:

变量名: PATH
变量值: %JAVA_HOME%\bin;%JAVA_HOME%\jre\bin;

在 Windows 命令提示符下验证 java 是否成功:

cmd.exe

```
.....
C:\Users\fnngj>java
用法: java [-options] class [args...]
          (执行类)
```

或 java [-options] -jar jarfile [args...]
(执行 jar 文件)

其中选项包括：

-d32	使用 32 位数据模型 (如果可用)
-d64	使用 64 位数据模型 (如果可用)
-server	选择 "server" VM
-hotspot	是 "server" VM 的同义词 [已过时]
	默认 VM 是 server.

.....

C:\Users\fnngj>javac

用法：javac <options> <source files>

其中，可能的选项包括：

-g	生成所有调试信息
-g:none	不生成任何调试信息
-g:{lines,vars,source}	只生成某些调试信息
-nowarn	不生成任何警告
-verbose	输出有关编译器正在执行的操作的消息
-deprecation	输出使用已过时的 API 的源位置
-classpath <路径>	指定查找用户类文件和注释处理程序的位置
-cp <路径>	指定查找用户类文件和注释处理程序的位置
.....	

java 命令可以运行 class 文件字节码。

javac 命令可以将 java 源文件编译为 class 字节码文件

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

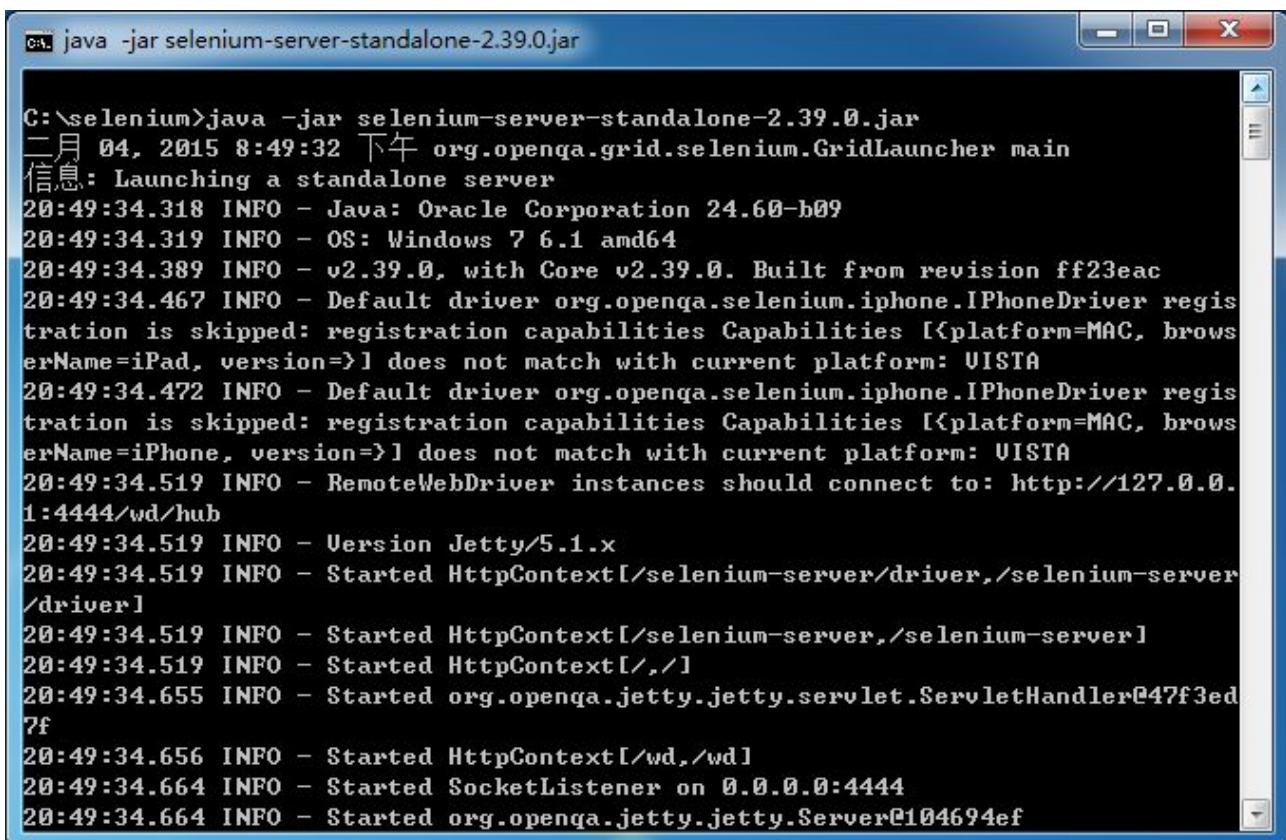


图 10.2 启动 Selenium Server

执行 se_rc.py 脚本，Selenium Server 将会做为代理，执行客户端与服务器端的请求与返回信息。

```

cmd.exe
.....
21:09:55.705 INFO - Checking Resource aliases
21:09:55.705 INFO - Command request: getNewBrowserSession[*chrome,
http://www.baidu.com/, ] on session null
21:09:55.705 INFO - creating new remote session
21:09:55.768 INFO - Allocated session b15d2335a6204d9b96a67996ff94b3ea for http:
//www.baidu.com/, launching...
jar:file:/C:/selenium/selenium-server-standalone-2.39.0.jar!/customProfileDi
rCUS
TFFCHROME
21:09:55.955 INFO - Preparing Firefox profile...
21:09:58.727 INFO - Launching Firefox...
21:10:05.439 INFO - Got result: OK,b15d2335a6204d9b96a67996ff94b3ea on session
b
15d2335a6204d9b96a67996ff94b3ea
21:10:05.439 INFO - Command request: open[/, True] on session b15d2335a6204d9b96
a67996ff94b3ea
21:10:08.098 INFO - Got result: OK on session b15d2335a6204d9b96a67996ff94b3ea
21:10:08.098 INFO - Command request: type[id=kw, selenium grid] on session b15d2
335a6204d9b96a67996ff94b3ea

```

```
21:10:08.113 INFO - Got result: OK on session b15d2335a6204d9b96a67996ff94b3ea
21:10:08.129 INFO - Command request: click[id=su, ] on session b15d2335a6204d9b96a67996ff94b3ea
21:10:08.145 INFO - Got result: OK on session b15d2335a6204d9b96a67996ff94b3ea
21:10:08.160 INFO - Command request: waitForPageToLoad[30000, ] on session b15d2335a6204d9b96a67996ff94b3ea
21:10:08.489 INFO - Got result: OK on session b15d2335a6204d9b96a67996ff94b3ea
21:10:08.489 INFO - Command request: testComplete[, ] on session b15d2335a6204d9b96a67996ff94b3ea
21:10:08.489 INFO - Killing Firefox...
21:10:08.738 INFO - Got result: OK on session b15d2335a6204d9b96a67996ff94b3ea
```

通过 Selenium Server 运行 Selenium RC , 这就是 Selenium 1 来做自动化的方式。

10.3 Selenium Grid 工作原理

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，hub 默认端口号为 4444，node 默认端口号为 5555；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：

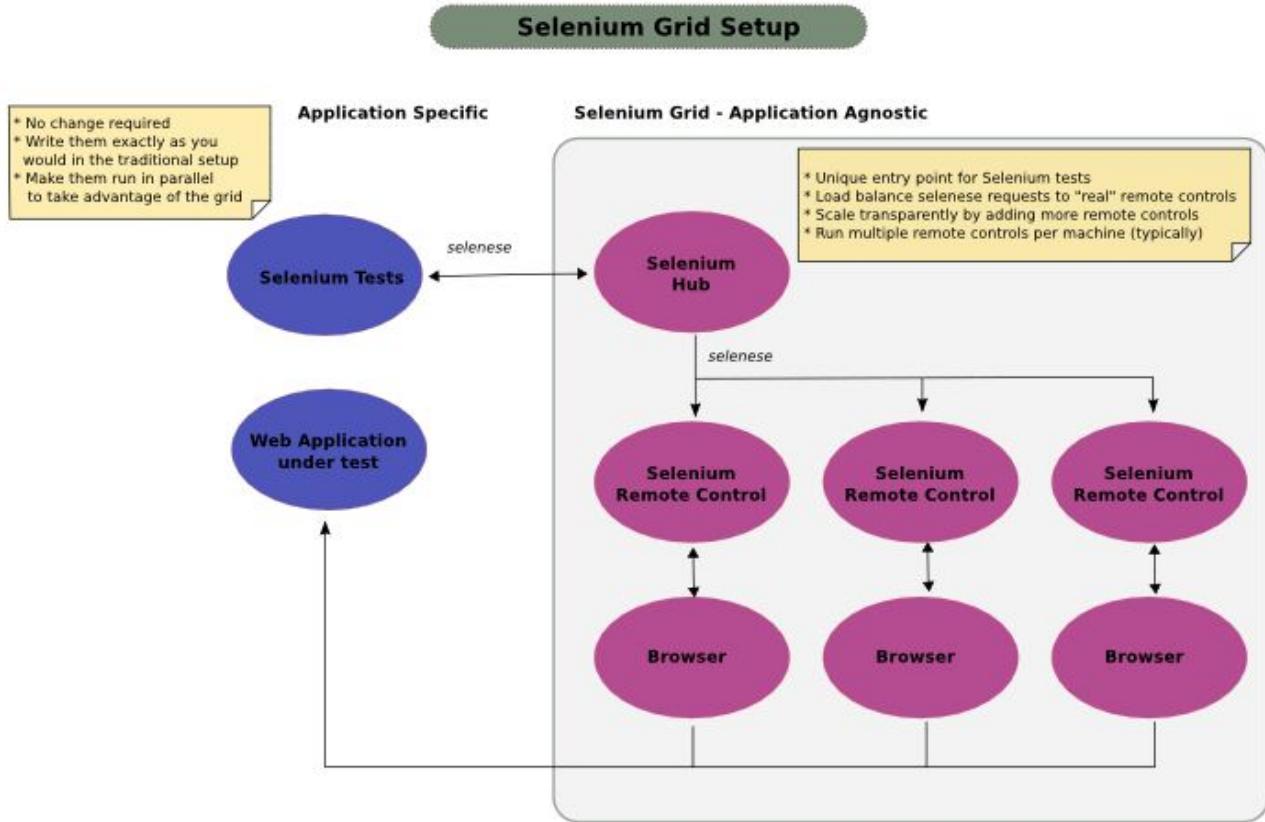


图 10.4 Selenium Grid setup

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；关于并发的技术需要借助编程语言的多线程技术，我们会后面的章节深入学习 Python 的多线程技术。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

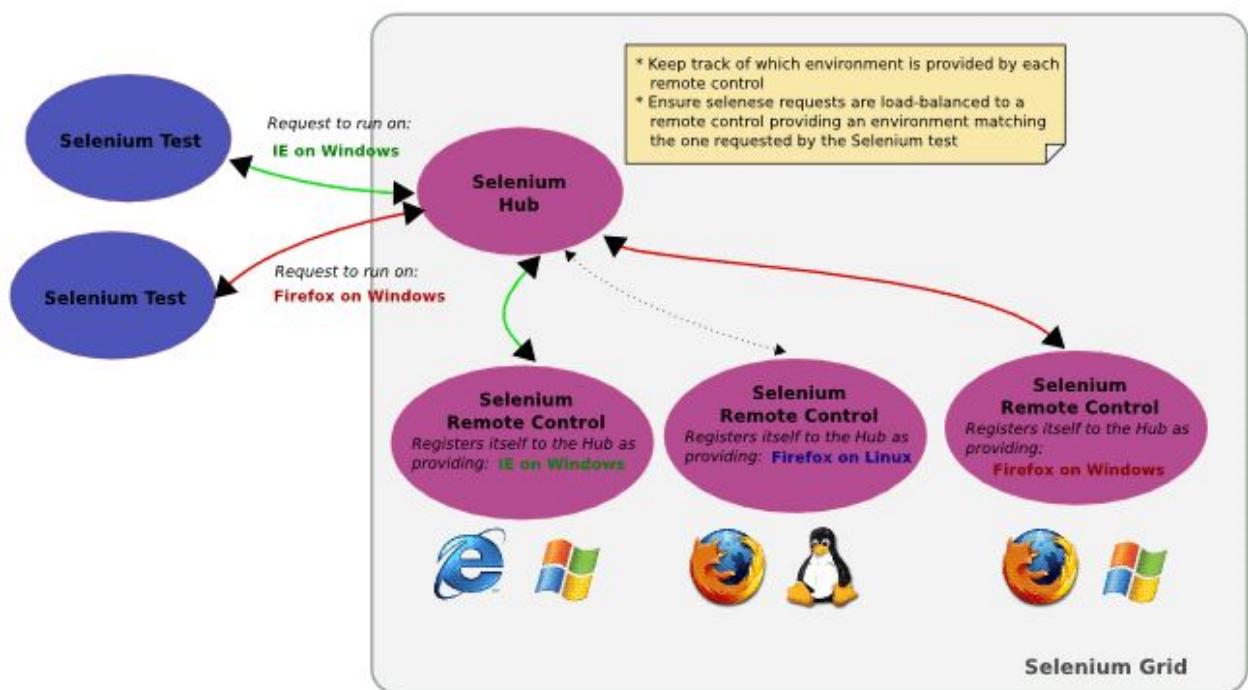


图 10.5 Selenium Grid 原理

下面我们就来演示启动一个 hub 主节点和两个 node 分支节点。

The image shows four separate terminal windows, each running a different command related to the Selenium Grid. The windows are arranged vertically.

- The top window shows the command: `java -jar selenium-server-standalone-2.39.0.jar -role hub`.
- The second window shows the command: `java -jar selenium-server-standalone-2.39.0.jar -role node -port 5555`.
- The third window shows the command: `java -jar selenium-server-standalone-2.39.0.jar -role node -port 5555`.
- The bottom window shows the command: `java -jar selenium-server-standalone-2.39.0.jar -role node -port 5556`.

Each window displays the output of the command, which includes the date and time, the role being executed, and various system information and logs from the Selenium Grid launcher.

```
java -jar selenium-server-standalone-2.39.0.jar -role hub
E:\selenium>java -jar selenium-server-standalone-2.39.0.jar -role hub
java -jar selenium-server-standalone-2.39.0.jar -role node -port 5555
E:\selenium>java -jar selenium-server-standalone-2.39.0.jar -role node -port 5555
java -jar selenium-server-standalone-2.39.0.jar -role node -port 5556
二月 12, 2015 10:08:08 上午 org.openqa.grid.selenium.GridLauncher main
信息: Launching a selenium grid node
10:08:14.301 INFO - Java: Oracle Corporation 24.60-b09
10:08:14.301 INFO - OS: Windows 7 6.1 amd64
10:08:14.317 INFO - v2.39.0, with Core v2.39.0. Built from revision ff23eac
10:08:14.395 INFO - Default driver org.openqa.selenium.iphone.IPhoneDriver regis-
tration is skipped: registration capabilities Capabilities [{platform=MAC, brows-
erName=iPhone, version=>}] does not match with current platform: VISTA
10:08:14.395 INFO - Default driver org.openqa.selenium.iphone.IPhoneDriver regis-
tration is skipped: registration capabilities Capabilities [{platform=MAC, brows-
erName=iPad, version=>}] does not match with current platform: VISTA
10:08:14.426 INFO - RemoteWebDriver instances should connect to: http://127.0.0.
1:5556/wd/hub
10:08:14.426 INFO - Version Jetty/5.1.x
10:08:14.426 INFO - Started HttpContext[/selenium-server/driver,/selenium-server/
driver]
10:08:14.426 INFO - Started HttpContext[/selenium-server,/selenium-server]
10:08:14.426 INFO - Started HttpContext[/,/]
10:08:14.441 INFO - Started org.openqa.jetty.jetty.servlet.ServletHandler@4bac52
11
10:08:14.441 INFO - Started HttpContext[/wd,/wd]
10:08:14.441 INFO - Started SocketListener on 0.0.0.0:5556
```

图 10.6 启动 hub 和 node

通过浏览器访问 grid 的控制台：<http://127.0.0.1:4444/grid/console>

通过控制台查看启动的节信息，如图 10.4。

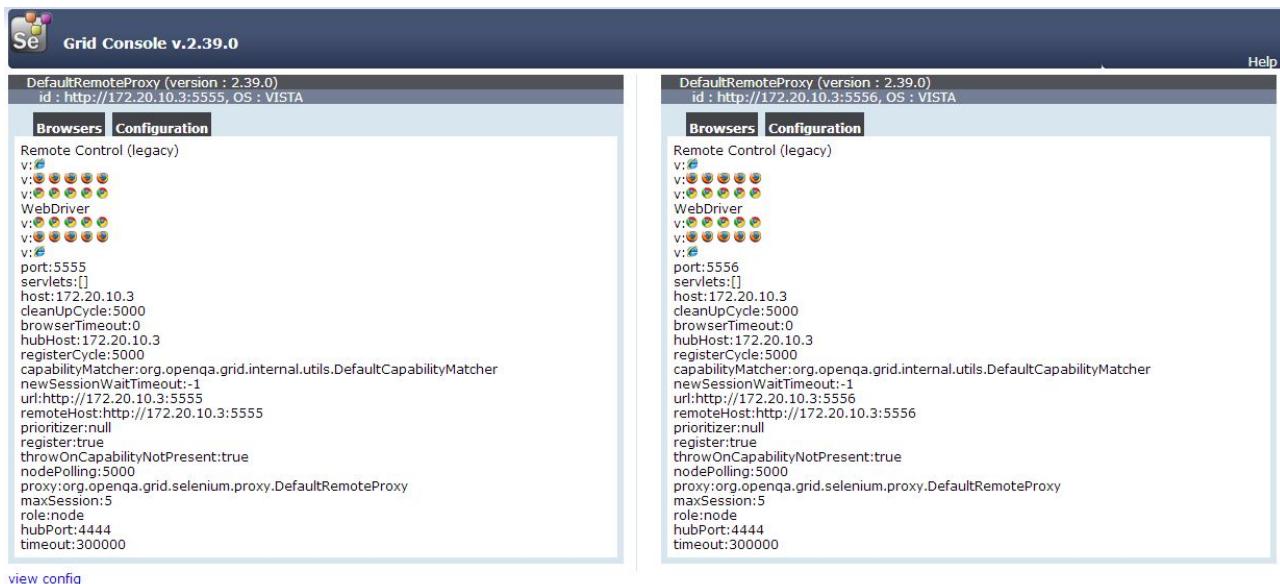


图 10.7 node 节点详细信息

node(5555)

Remote Control (legacy)

v:

v:

v:

WebDriver

v:

v:

v:

port:5555

servlets:[]

host:172.20.10.3

cleanUpCycle:5000

browserTimeout:0

hubHost:172.20.10.3

registerCycle:5000

capabilityMatcher:org.openqa.grid.internal.utils.DefaultCapabilityMatcher

newSessionWaitTimeout:-1

url:http://172.20.10.3:5555

remoteHost:http://172.20.10.3:5555

prioritizer:null

register:true

throwOnCapabilityNotPresent:true

nodePolling:5000

proxy:org.openqa.grid.selenium.proxy.DefaultRemoteProxy

maxSession:5

role:node

hubPort:4444

timeout:300000

以上的信息为端口为 5555 的 node 信息。

10.4 Selenium Grid 应用

到目前为止，我们编写的脚本只能在固定的某一款浏览器上执行，因为在编写测试用例之前需要先指定好浏览器的驱动（webdriver.Firefox、webdriver.Chrome），一旦确定了浏览器驱动是不能进行更改的。我们希望写好的测试用例可以自由的切换成不同的浏览器来执行。

10.4.1 认识 Remote

WebDriver 提供了 Remote 可以发送指令到远程服务器控制浏览器。首先我们先来认识 Remote 的格式。

remote_ts.py

```
from selenium.webdriver import Remote
from selenium.webdriver.common.desired_capabilities import DesiredCapabilities

driver = Remote(command_executor='http://127.0.0.1:4444/wd/hub',
                 desired_capabilities=DesiredCapabilities.CHROME
                 )

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

(注意：脚本的执行需要启动 Selenium Server。)

command_executor 为命令执行器，它用于指定脚本执行的主机及端口。

desired_capabilities 先不做解释，我们先看看它里面都包含了些什么？

Python Shell

```
>>> from selenium.webdriver.common.desired_capabilities import
DesiredCapabilities
>>> a=DesiredCapabilities.CHROME
>>> print a
{'platform': 'ANY', 'browserName': 'chrome', 'version': '', 'javascriptEnabled':
True}
```

Desired Capabilities 本质上是 key value 的对象，它告诉 Selenium Server 脚本执行的基本运行环境：

`'platform': 'ANY'` 平台默认可以是任何 (Windows, 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

认识了 Desired Capabilities 所包含的信息，在创建驱动 (driver) 时，可以这样来描述运行的环境：

remote_ts.py

```
from selenium.webdriver import Remote
from selenium.webdriver.common.desired_capabilities import DesiredCapabilities

driver = Remote(command_executor='http://127.0.0.1:4444/wd/hub',
                 desired_capabilities={'platform': 'ANY',
                                      'browserName': 'chrome',
                                      'version': '',
                                      'javascriptEnabled': True
                                      })

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

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}

```

10.4.2 参数化浏览器执行用例

通过对 Desired Capabilities 的分解，浏览器的配置是由 browserName 所对应的值所指定的，那么实现浏览器的参数化将变得非常简单。

在运行脚本之前我们需要先启动 Selenium Server，因为只对浏览器进行参数化，所以启动一个节点即可：

```
C:\selenium> java -jar selenium-server-standalone-2.39.0.jar
```

实现脚本如下：

browsers.py

```

#coding=utf-8
from selenium.webdriver import Remote
from selenium.webdriver.common.desired_capabilities import DesiredCapabilities

#浏览器数组
lists=['chrome','firefox','internet explorer']

#读取不同的浏览器执行脚本
for browser in lists:
    print browser
    driver = Remote(
        command_executor='http://127.0.0.1:4444/wd/hub',
        desired_capabilities={'platform': 'ANY',
                             'browserName':browser,
                             'version': '',
                             'javascriptEnabled': True
                            })

    driver.get("http://www.baidu.com")
    driver.find_element_by_id("kw").send_keys(browser)
    driver.find_element_by_id("su").click()
    driver.close()

```

创建 lists 数组，罗列不同的浏览器，通过 for 循环读取数组中的浏览器，传参给当 browserName 做为当前要执行的浏览器来运行测试用例。

在实际的测试中，我们可以将浏览器的配置写到 info.xml 配置文件中，在测试用例的脚本中读取配置文件，请参考第五章自动化测试模型中关于参数化的讲解。

10.4.3 多节点执行用例

在 Selenium Grid 原理的小节中，我们演示了如何启动多个节点，下面来启动多个节点，使同一个脚本在不同的节点上执行。

在本机打开两个命令提示符窗口分别启动一个 hub：

```
C:\selenium>java -jar selenium-server-standalone-2.39.0.jar -role hub
```

启动两个 node（节点）：

```
C:\selenium>java -jar selenium-server-standalone-2.39.0.jar -role node -port 5555
```

```
C:\selenium>java -jar selenium-server-standalone-2.39.0.jar -role node -port 5556
```

下面修改脚本使其在不同的节点与浏览器上运行：

hosts.py

```
#coding=utf-8
from selenium.webdriver import Remote
from selenium.webdriver.common.desired_capabilities import DesiredCapabilities

#定义主机与浏览器
lists={'http://127.0.0.1:4444/wd/hub':'chrome',
       'http://127.0.0.1:5555/wd/hub':'firefox',
       'http://127.0.0.1:5556/wd/hub':'internet explorer'}

#通过不同的浏览器执行脚本
for host,browser in lists.items():
    print host,browser
    driver = Remote(
        command_executor=host,
        desired_capabilities={'platform': 'ANY',
                             'browserName':browser,
                             'version': '',
                             'javascriptEnabled': True
                            })
```

```

driver.get("http://www.baidu.com")
driver.find_element_by_id("kw").send_keys(browser)
driver.find_element_by_id("su").click()
driver.close()

```

创建 lists 字典，定义不同的主机端口号与浏览器，通过 for 循环读取字典中的主机与浏览器给 Remote，使脚本执行时调用不同的主机和浏览器。

启动远程 node

我们目前启动的 hub 与 node 都是在一台主机上。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运行环境（Python、Selenium、浏览器及浏览器驱动）。
- 远程主机必须安装 java 环境，因为需要运行 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)

修改远程主机的 IP 地址及端口号，在其上面的 firefox 浏览下运行脚本。

hosts.py

```

.....
#define host and browser
lists={'http://127.0.0.1:4444/wd/hub':'chrome',
       'http://127.0.0.1:5555/wd/hub':'internet explorer',
       'http://172.16.10.34:5555/wd/hub': 'firefox'}
.....

```

现在再来运行脚本，你将会在 172.16.10.34 主机上看到脚本被执行。

小技巧：

自从使用 Selenium Server 执行自动化脚本后，常常会因为忘记启动 Selenium Server 导致脚本运行报错。其实我们可以在脚本中执行 cmd 命令。如在我们创建的 all_test.py 文件中添加：

```
import os
os.chdir('E:\\selenium')
os.system('java -jar selenium-server-standalone-2.39.0.jar')
```

这样就再也不会因为忘记启动 Selenium Server 导致测试脚本运行失败了。

10.5 WebDriver 驱动

到目前为止，我们用到的浏览器驱动有 Firefox (Selenium 集成) 、 Chrome (chromedriver) 以及 IE (IEDriverServer) 等，除此 WebDriver 还支持脚本在其它驱动下运行。

WebDriver 所支持的驱动：

驱动	说明
Firefox Driver	包含在各语言的 selenium(WebDriver) 包里，这也是为什么安装完 selenium 就可以直接使用 firefox 启动脚本的原因。
Chrome Driver	需要下载 chromedriver，因为 webdriver 原本是谷歌的项目，之后与 selenium 合并，所以对 chrome 的支持也非常好。
IE Driver	需要下载 IEDriverServer。
Opera Driver	OperaDriver 是 WebDriver 厂商 Opera Software 和志愿者开发了对于 Opera 的 WebDriver 实现。
HtmlUnit Driver	HtmlUnit 回文档模拟成 HTML，从而模拟浏览器的运行，但又非真正的去启动一款浏览器去执行脚本。 Selenium Server 中已经包含了 HtmlUnit。
PhantomJS Driver	PhantomJS 是一个拥有 JavaScript API 的无界面 WebKit，和 HtmlUnit 类似，可以被看作一个款无界面的浏览器。
Appium	Appium 可以被看作移动端的 Selenium，它同样支持多平台 (iOS、Android 及 FirefoxOS) 的 app 及移动端 web 的自动化测试

10.5.1 HtmlUnit

Htmlunit 官方网站：

<http://htmlunit.sourceforge.net/>

HtmlUnit 是一个“GUI-Less browser for Java programs”。该模型的 HTML 文件，并提供了一个 API，允许您调用页面，填写表格，点击链接等..... 就像你在“正常”的浏览器做的。它具有相当不错的 JavaScript 的支持(这是不断提高)，即使有相当复杂的 Ajax 库的工作，无论是模拟 Firefox 或 IE 浏

览器都可以根据你的需要进行配置。因为 Selenium Server 中已经包含了 HtmlUnit，所以我们可以直接使用它来进行测试。

htmlunit.py

```
#coding=utf-8
from selenium import webdriver
from selenium.webdriver.common.desired_capabilities import DesiredCapabilities
from time import ctime

#通过 htmlunit 执行脚本
print ctime() #打印脚本运行开始时间

driver = webdriver.Remote(
    command_executor=' http://127.0.0.1:4444/wd/hub',
    desired_capabilities={'platform': 'ANY',
                         'browserName': 'htmlunit',
                         'version': '',
                         'javascriptEnabled': True
    })

driver.get("http://www.baidu.com")
print 'open baidu'
driver.find_element_by_id("kw").send_keys("hello")
driver.find_element_by_id("su").click()
print 'search completed'
driver.close()
print ctime() #打印脚本运行结束时间
```

运行脚本之前需要先启动 Selenium Server，脚本运行结果如下：

Python Shell

```
>>> ===== RESTART =====
>>>
Sat Feb 14 17:19:20 2015
open baidu
search completed
Sat Feb 14 17:19:27 2015
```

从执行时间上看要快于运行在真实浏览器下执行脚本，这也是使用 HtmlUnit 驱动的优点，可以加快测试用例的运行时间。

10.5.2 PhantomJS

PhantomJS 官方网站: <http://phantomjs.org/>

PhantomJS 是一个拥有 JavaScript API 的无界面 WebKit。正如你所知道的，Webkit 是 Chrome、Safari 和其他一些小众浏览器使用的布局引擎。因此，PhantomJS 是一个浏览器，而且是一个无界面的浏览器。这意味着，渲染后的网页实际上绝不会显示。这对你来说可能不可思议，所以你可以把它作为一个可编程的浏览器终端。

在使 PhantomJS 之前，需要先下载。PhantomJS 支持 Windwos、MAC、Linux 等平台，我们可以根据自己的环境选择相应的版本进行下载。

名称	修改日期	类型	大小
bin	2015/2/14 17:40	文件夹	
examples	2015/2/14 17:40	文件夹	
ChangeLog	2015/1/24 11:13	文件	16 KB
LICENSE.BSD	2015/1/24 11:13	BSD 文件	2 KB
README.md	2015/1/24 11:13	MD 文件	5 KB
third-party.txt	2015/1/24 11:13	文本文档	2 KB

图 10.5 PhantomJS 解压目录

下载完成解压会得 phantomjs-2.0.0-windows 目录，在 bin 目录会看到 phantomjs.exe 程序，将当前目录添加到系统环境变量 path 下。（或直接将 phantomjs.exe 丢到 Python 的安装目录下，因为我们最开始已将 Python 添加到了系统环境变量的 path 下）。

下面就可以使用 phantomjs 驱动进行测试了。

phantomjs.py

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

print ctime()
#指定 phantomjs 驱动
driver = webdriver.PhantomJS(executable_path='C:\\Python27\\phantomjs')
driver.get("http://www.baidu.com")

driver.find_element_by_id('kw').send_keys('PhantomJS')
driver.find_element_by_id('su').click()
print 'Search completed!'
```

```
driver.quit()
print ctime()
```

executable_path 用于指定 phantomjs.exe 的路径。

脚本运行结果如下：

Python Shell

```
>>> ===== RESTART =====
>>>
Sat Feb 14 19:23:09 2015
Search completed!
Sat Feb 14 19:23:26 2015
```

当然，为了方便驱动的参数化，参考 htmlunit.py 脚本，将 browserName 的参数替换为“phantomjs”就使用 PhantomJS 运行测试脚本。

从脚本的执行时间上看和 HtmlUnit 旗鼓相当。

下面将真实的浏览器驱动和 HtmlUnit 以及 PhantomJS 做个简单的对比。

驱动类型	优点	缺点	应用
真实的浏览器驱动	真实模拟用户行为	运行效率、稳定性低	兼容性测试
HtmlUnit	运行速度快	js 引擎不是主流的浏览器支持	包含少量 js 的页面测试
PhantomJS	运行速度快、 模拟行为 接近真实浏览器	不能模拟特定浏览器的行为	非 GUI 的功能性测试

小结：

本章重点介绍了 Selenium Grid 的原理，因为其被集成到了 Selenium Server 中，所以我们以 Selenium Server 为载体进行使用。通过实例介绍如何通过 Selenium Grid 完成分布式测试，最后介绍了 HtmlUnit 和 PhantomJS 两个特别的驱动来运行脚本。

第 11 章 Python 多线程

在学习 Selenium Grid 的时候，你可能会存在疑问，Selenium Grid 不是可以做并行测试么？在这里更正一下，“分布式”和“并行”是两个完全不同的概念，分布式只负责将一个测试脚本可调用不同的远程环境来执行；并行强调“同时”的概念，它可以借助多线程或多进程技术并行的来执行测试脚本，从而缩短测试脚本的运行时间。

这一章我们将学习 Python 的多线程/多进程技术，并将其应用到我们的自动化测项目中。

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”。

11.1 单线程的时代

在单线程的时代，当处理器要处理多个任务时，必须要对这些任务排一下执行顺序并按照这个顺序来执行任务。假如我们创建了两个任务，听音乐（music）和看电影（move），在单线程中我们只能按先后顺序来执行这两个任务。下面就通过一个例子来演示：

onethread.py

```
#coding=utf-8
from time import sleep, ctime

# 创建听音乐任务
def music():
    print 'I was listening to music! %s' %ctime()
    sleep(2)

# 创建看电影任务
def move():
    print 'I was at the movies! %s' %ctime()
    sleep(5)
```

```
if __name__ == '__main__':
    music()
    move()
print 'all end:', ctime()
```

我们分别创建了两个任务 music 和 move，执行 music 需要 2 秒，执行 move 需要 5 秒，通过 sleep() 方法设置休眠时间来模拟任务的运行时间。

运行结果：

Python Shell

```
>>> ===== RESTART =====
>>>
I was listening to music! Sat Feb 14 20:11:04 2015
I was at the movies! Sat Feb 14 20:11:06 2015
all end: Sat Feb 14 20:11:11 2015
```

从执行结果可看到，程序从 11 分 04 秒开始播放 music，11 分 06 秒结束并开始 move 播放，直到 11 分 11 秒 move 播放结束，总耗时 7 秒。

好吧！我们对上面的例子做些调整，使它看起来更加有意思。

我们希望 music() 和 move() 两个方法完成的工作更灵活一些，首先 music() 和 move() 应该做为播放器，在用户使用时，根据用户的需求来播放相应的歌曲和影片；其次我们希望能提供循环播放功能，尤其对于音乐播放器来说这个很重要，对吧！？。我们改造后的程序如下：

onethread.py

```
#coding=utf-8
from time import sleep, ctime

#音乐播放器
def music(func):
    for i in range(2):
        print 'I was listening to %s! %s' %(func,ctime())
        sleep(2)

#视频播放器
def move(func):
    for i in range(2):
        print 'I was at the %s! %s' %(func,ctime())
        sleep(5)

if __name__ == '__main__':
    music(u'爱情买卖')
    move(u'阿凡达')
```

```
print 'all end:', ctime()
```

给 music() 和 move() 两个方法设置参数，用于接收要播放的歌曲和视频，通过 for 循环控制播放的次数，从改造后的程序可以看出，我们分别让歌曲和电影播放了 2 次。

运行结果：

Python Shell

```
>>> ===== RESTART =====
>>>
I was listening to 爱情买卖! Sat Feb 14 20:42:36 2015
I was listening to 爱情买卖! Sat Feb 14 20:42:38 2015
I was at the 阿凡达! Sat Feb 14 20:42:40 2015
I was at the 阿凡达! Sat Feb 14 20:42:45 2015
all end: Sat Feb 14 20:42:50 2015
```

从执行可看到，程序从 42 分 36 秒开始播放 music，42 分 40 秒 music 两轮播放结束并开始播放；42 分 50 秒两个任务结束，总耗时 14 秒。

11.2 多线程技术

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。

11.2.1 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。所以，我们直接使用 threading 来改进上面的例子。

threads.py

```
#coding=utf-8
import threading
from time import sleep, ctime

#音乐播放器
def music(func):
    for i in range(2):
        print "I was listening to %s! %s" %(func,ctime())
        sleep(2)
```

```

#视频播放器
def move(func):
    for i in range(2):
        print "I was at the %s! %s" %(func,ctime())
        sleep(5)

#创建线程数组
threads = []

#创建线程 t1，并添加到线程数组
t1 = threading.Thread(target=music,args=(u'爱情买卖',))
threads.append(t1)

#创建线程 t2，并添加到线程数组
t2 = threading.Thread(target=move,args=(u'阿凡达',))
threads.append(t2)

if __name__ == '__main__':
    #启动线程
    for i in threads:
        i.start()
    #守护线程
    for i in threads:
        i.join()

    print 'all end: %s' %ctime()

```

import threading 引入线程模块。

threads = [] 创建线程数组，用于装载线程。

threading.Thread()通过调用 threading 模块的 Thread() 方法来创建线程。

class threading.Thread():

class threading.Thread(group=None, target=None, name=None, args=(), kwargs={})

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to () .

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {} .

If the subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

`start()` 开始线程活动。

`join()` 等待线程终止。

通过 `for` 循环遍历 `thread` 数组中所装载的线程；然后通过 `start()` 函数启动每一个线程。

`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。

运行结果：

Python Shell

```
>>> ===== RESTART =====
>>>
I was listening to 爱情买卖! Sun Feb 15 10:17:09 2015
I was at the 阿凡达! Sun Feb 15 10:17:09 2015
I was listening to 爱情买卖! Sun Feb 15 10:17:11 2015
I was at the 阿凡达! Sun Feb 15 10:17:14 2015
all end: Sun Feb 15 10:17:19 2015
```

从上面的运行结果可以看出，两个子线程（`music`、`move`）同时启动于 17 分 09 秒，直到所有线程结束于 17 分 19 秒，总耗时为 10 秒。`move` 的两次电影循环需要 10 秒，`music` 的歌曲循环需要 4 秒，从执行结果可以看出两个线程真正达到了并行工作。

11.2.2 优化线程的创建

从上面例子中发现线程的创建是颇为麻烦的，每创建一个线程都需要创建一个 `t (t1, t2, ...)`，如果创建的线程较多时这样极其不方便。下面对通过例子进行改进：

player.py

```
#coding=utf-8
from time import sleep, ctime
import threading

def muisc(func):
    for i in range(2):
        print 'Start playing: %s! %s' %(func,ctime())
        sleep(2)

def move(func):
    for i in range(2):
        print 'Start playing: %s! %s' %(func,ctime())
        sleep(5)
```

```

#判断文件类型，交给相应的函数执行
def player(name):
    r = name.split('.')[1]
    if r == 'mp3':
        muisc(name)
    elif r == 'mp4':
        move(name)
    else:
        print 'error: The format is not recognized!'

list = ['爱情买卖.mp3', '阿凡达.mp4']

threads = []
files = range(len(list))

#创建线程
for i in files:
    t = threading.Thread(target=player, args=(list[i],))
    threads.append(t)

if __name__ == '__main__':
    #启动线程
    for i in files:
        threads[i].start()
    for i in files:
        threads[i].join()

    #主线程
print 'end:%s' %ctime()

```

有趣的是我们又创建了一个 player() 函数，这个函数用于判断播放文件的类型。如果是 mp3 格式的，我们将调用 music() 函数，如果是 mp4 格式的我们调用 move() 函数。哪果两种格式都不是那么只能告诉用户你所提供的文件我播放不了。

然后，我们创建了一个 list 的文件列表，注意为文件加上后缀名。然后我们用 len(list) 来计算 list 列表有多少个文件，这是为了帮助我们确定循环次数。

接着我们通过一个 for 循环，把 list 中的文件添加到线程中数组 threads[] 中。接着启动 threads[] 线程组，最后打印结束时间。

运行结果：

Python Shell

```

>>> ===== RESTART =====
>>>

```

```
Start playing: 爱情买卖.mp3! Sun Feb 15 10:49:42 2015
Start playing: 阿凡达.mp4! Sun Feb 15 10:49:42 2015
Start playing: 爱情买卖.mp3! Sun Feb 15 10:49:44 2015
Start playing: 阿凡达.mp4! Sun Feb 15 10:49:47 2015
end:Sun Feb 15 10:49:52 2015
```

现在向 list 数组中添加一个文件，程序运行时会自动为其创建一个线程。

通过上面的程序，我们发现 player() 用于判断文件扩展名，然后调用 music() 和 move()，其实，music() 和 move() 完整工作是相同的，我们为什么不做一台超级播放器呢，不管什么文件都可以播放。再次经过改造，我们的超级播放器诞生了。

super_player.py

```
#coding=utf-8
from time import sleep, ctime
import threading

# 创建超级播放器
def super_player(file, time):
    for i in range(2):
        print 'Start playing: %s! %s' %(file, ctime())
        sleep(time)

# 播放的文件与播放时长
list = {'爱情买卖.mp3':3, '阿凡达.mp4':5, '我和你.mp3':4}

threads = []
files = range(len(list))

# 创建线程
for file, time in list.items():
    t = threading.Thread(target=super_player, args=(file, time))
    threads.append(t)

if __name__ == '__main__':
    # 启动线程
    for i in files:
        threads[i].start()
    for i in files:
        threads[i].join()

    # 主线程
    print 'end:%s' %ctime()
```

首先创建字典 list，用于定义要播放的文件及时长（秒），通过字典的 items() 方法来循环的取 file

和 time，取到的这两个值用于创建线程。

接着创建 super_player() 函数，用于接收 file 和 time，用于确定要播放的文件及时长。

最后是线程启动运行。运行结果：

Python Shell

```
>>> ===== RESTART =====
>>>
Start playing: 爱情买卖.mp3! Sun Feb 15 11:05:53 2015
Start playing: 我和你.mp3! Sun Feb 15 11:05:53 2015
Start playing: 阿凡达.mp4! Sun Feb 15 11:05:53 2015
Start playing: 爱情买卖.mp3! Sun Feb 15 11:05:56 2015
Start playing: 我和你.mp3! Sun Feb 15 11:05:57 2015
Start playing: 阿凡达.mp4! Sun Feb 15 11:05:58 2015
end:Sun Feb 15 11:06:03 2015
```

11.2.3 创建线程类

除了使用 Python 所提供的线程类外，我们也可以根据需求来创建自己的线程类。

mythread.py

```
#coding=utf-8
import threading
from time import sleep, ctime

#创建线程类
class MyThread(threading.Thread):

    def __init__(self, func, args, name=''):
        threading.Thread.__init__(self)
        self.name=name
        self.func=func
        self.args=args

    def run(self):
        apply(self.func, self.args)

def super_play(file, time):
    for i in range(2):
        print 'Start playing: %s! %s' %(file,ctime())
        sleep(time)
```

```

list = {'爱情买卖.mp3':3,'阿凡达.mp4':5}

#创建线程
threads = []
files = range(len(list))

for file,time in list.items():
    t = MyThread(super_play,(file,time),super_play.__name__)
    threads.append(t)

if __name__ == '__main__':
    #启动线程
    for i in files:
        threads[i].start()
    for i in files:
        threads[i].join()

    #主线程
    print 'end:%s' %ctime()

```

MyThread(threading.Thread)

创建 MyThread 类，用于继承 threading.Thread 类。

`__init__()` 类的初始化方法对 func、args、name 等参数进行初始化。

`apply()` 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。

由于 MyThread 类继承 threading.Thread 类，所以，我们可以使用 MyThread 类来创建线程。

11.3 多进程技术

11.3.1 multiprocessing 模块

多进程 multiprocessing 模块的使用与多线程 threading 模块类似，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。

修改多线程的例子，将 threading 模块中的 Thread 方法替换为 multiprocessing 模块的 Process 就

实现了多进程。

process.py

```
#coding=utf-8
from time import sleep, ctime
import multiprocessing

def super_player(file, time):
    for i in range(2):
        print 'Start playing: %s! %s' %(file, ctime())
        sleep(time)

#播放的文件与播放时长
list = {'爱情买卖.mp3':3, '阿凡达.mp4':5, '我和你.mp3':4}

threads = []
files = range(len(list))

#创建线程
for file, time in list.items():
    t = multiprocessing.Process(target=super_player, args=(file, time))
    threads.append(t)

if __name__ == '__main__':
    #启动线程
    for i in files:
        threads[i].start()
    for i in files:
        threads[i].join()

    #主线程
    print 'end:%s' %ctime()
```

从上面的实例中可以看到，多进程的使用，几乎与多线程一样。

我们利用 `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。

运行结果：

Python Shell

```
>>> ===== RESTART =====
>>>
Start playing: 爱情买卖.mp3! Sun Feb 15 11:35:46 2015
Start playing: 爱情买卖.mp3! Sun Feb 15 11:35:49 2015
Start playing: 我和你.mp3! Sun Feb 15 11:35:46 2015
Start playing: 我和你.mp3! Sun Feb 15 11:35:50 2015
Start playing: 阿凡达.mp4! Sun Feb 15 11:35:46 2015
Start playing: 阿凡达.mp4! Sun Feb 15 11:35:51 2015
end:Sun Feb 15 11:35:56 2015
```

从运行结果中还是看出细微的差别，虽然利用多进程任务时间没有变化，但多程的结果显示是以进程组为顺序进行显示的。

扩展阅读：

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。

11.3.2 Pipe 和 Queue

multiprocessing 提供了 threading 包中没有的 IPC，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）。

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：

Ubuntu

```
'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 的某一段调用 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 multiprocessing
import time,os

#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

Ubuntu

```

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
```

11.4 应用于自动化测试

因为多进程/多线程特性在 Python 中属于比较高级的应用，对于初学者来说理解起来有一定的难度，所以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/多线程技术。

11.4.1 应用于自动化测试项目

为实现多进程运行测试用例，我需要对文件结构进行调整：

```
/test_project/thread1/baidu_sta.py      ----测试用例
    /thread1/__init__.py
    /thread2/youdao_sta.py      ----测试用例
    /thread2/__init__.py
    /report/                  ----测试报告目录
    /all_tests_pro.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。

test_all_pro.py

```
#coding=utf-8
import unittest, time, os, multiprocessing
import HTMLTestRunner

#查找多有含有 thread 的文件，文件夹
def EEEcreatsuit():
    casedir=[]
    listaa=os.listdir('/home/fnngj/test_project')
    for xx in listaa:
        if "thread" in xx:
            casedir.append(xx)

    suite=[]
```

```

for n in casedir:
    testunit=unittest.TestSuite()
    discover=unittest.defaultTestLoader.discover(n,
                                                pattern ='test*.py',
                                                top_level_dir=n)
    print discover
    for test_suite in discover:
        for test_case in test_suite:
            testunit.addTests(test_case)
    suite.append(testunit)

print "====casedir====",casedir
print "++++++++"*50
print "!!!suite:!!!",suite
return suite,casedir

#多线程运行测试套件，将结果写入 HTMLTestRunner 报告
def EEEEmultiRunCase(suite,casedir):
    now = time.strftime("%Y-%m-%d-%H_%M_%S",time.localtime(time.time()))
    filename = '/home/fnngj/test_project/report/'+now+'result.html'
    fp = file(filename, 'wb')
    proclist=[]
    s=0
    for i in suite:
        runner = HTMLTestRunner.HTMLTestRunner(
            stream=fp,
            title=u'测试报告',
            description=u'用例执行情况：'
        )

        proc = multiprocessing.Process(target=runner.run(i),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=EEEcreatesuit()
EEEEmultiRunCase(runtmp[0],runtmp[1])

```

all_tests_pro.py 程序稍微复杂，我们分段来进行讲解。

在 EEEcreatsuite1() 函数中：

test_all_pro.py

```
.....
casedir=[]
listaa=os.listdir('/home/fnngj/test_project')
for xx in listaa:
    if "thread" in xx:
        casedir.append(xx)
.....
```

定义 casedir 数组，读取 test_project 目录下的文件夹，找到文件夹的名包含“thread”的文件夹添加到 casedir 数组中（本例中为 thread1 和 thread2 两个文件夹）。

test_all_pro.py

```
.....
suite=[]
for n in casedir:
    testunit=unittest.TestSuite()
    discover=unittest.defaultTestLoader.discover(n,
                                                pattern ='test*.py',
                                                top_level_dir=n)
    print discover
    for test_suite in discover:
        for test_case in test_suite:
            testunit.addTests(test_case)
    suite.append(testunit)
.....
```

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 test*.py 规则的用例文件，将所有用例文件添加到 testunit 测试条件下，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：

test_all_pro.py

```
.....
def EEEEmultiRunCase(suite,casedir):
    now = time.strftime("%Y-%m-%d-%H_%M_%S",time.localtime(time.time()))
    filename = '/home/fnngj/test_project/report/'+now+'result.html'
    fp = file(filename, 'wb')
    proclist=[]
```

```

s=0
for i in suite:
    runner = HTMLTestRunner.HTMLTestRunner(
        stream=fp,
        title=u'测试报告',
        description=u'用例执行情况：'
    )

    proc = multiprocessing.Process(target=runner.run(i), args=(i,))
    proclist.append(proc)
    s=s+1

for proc in proclist: proc.start()
for proc in proclist: proc.join()
fp.close()

.....

```

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。

本小节中所运行的测试用例（baidu_sta.py 和 youdao_sta.py）请参考前面的章节编写，运行 all_tests_process.py 文件，最终生成的报告如图：

测试报告

Start Time: 2014-12-21 11:16:25

Duration: 0:00:25.668000

Status: Pass 2

用例执行情况：

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

Test Group/Test case	Count	Pass	Fail	Error	View
firefox_sto.Baidu	2	2	0	0	Detail
Total	2	2	0	0	

测试报告

Start Time: 2014-12-21 11:16:51

Duration: 0:00:23.003000

Status: Pass 2

用例执行情况：

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

Test Group/Test case	Count	Pass	Fail	Error	View
chrome_sto.Ch	2	2	0	0	Detail
Total	2	2	0	0	

图 11.1 多线程测试报告

11.4.2 应用于 Grid2 分布式测试

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 分布式并行的执行测试脚本，那么需要结合 Python 多线程技术。

启动 Selenium Server

在本机打开两个命令提示符窗口分别：

启动一个 hub（默认端口 4444），ip 地址为：172.16.10.66

```
C:\selenium>java -jar selenium-server-standalone-2.39.0.jar -role hub
```

启动一个本地 node（节点）

```
C:\selenium>java -jar selenium-server-standalone-2.39.0.jar -role node -port 5555
```

启动一个远程 node（节点），ip 为：172.16.10.34：

```
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
```

grid_thread.py

```
#coding=utf-8
from threading import Thread
from selenium import webdriver
import time

#启动参数（指定运行主机与浏览器）
list = {'http://127.0.0.1:4444/wd/hub':'chrome',
        'http://127.0.0.1:5555/wd/hub':'internet explorer',
        'http://172.16.10.34:5555/wd/hub': 'firefox'
       }

#测试用例
def test_baidu(host,browser):
    print 'start:%s' %ctime()
    print host,browser
    driver = webdriver.Remote(
                                command_executor=host,
```

```

desired_capabilities={'platform': 'ANY',
                      'browserName': browser,
                      'version': '',
                      'javascriptEnabled': True
                    })

driver.get('http://www.baidu.com')
driver.find_element_by_id("kw").send_keys(browser)
driver.find_element_by_id("su").click()
sleep(2)
driver.close()

threads = []
files = range(len(list))

# 创建线程
for host,browser in list.items():
    t = threading.Thread(target=test_baidu,args=(host,browser))
    threads.append(t)

if __name__ == '__main__':
    # 启动线程
    for i in files:
        threads[i].start()
    for i in files:
        threads[i].join()

    # 主线程
    print 'end:%s' %ctime()

```

运行结果：

Python Shell

```

>>> ===== RESTART =====
>>>
start:Fri Apr 25 11:04:12 2014
http://127.0.0.1:5555/wd/hub internet explorer
start:Fri Apr 25 11:04:12 2014
http://127.0.0.1:4444/wd/hub chrome
start:Fri Apr 25 11:04:12 2014
http://172.16.10.34:5555/wd/hub firefox
end:Fri Apr 25 11:04:23 2014

```

到此，我们才真正解决了同一个用例不同的主机与浏览器下并行执行的目的。Grid 本身并没有太神奇

的地方，他主要实现在本机与远程主机启动多个节点的作用，要想实现并行的效果还需要借助 Python 的多线程技术。

小结：

本章中我们使用了大量的实例来介绍 Python 的多线程/多进程技术。在实际的应用中多线程/多进程是非常有用的技术，尤其对于性能要求比较高的应用。所以，真正的理解掌握多线程技术的开发对于读者来说是一种能力的提高的体现。

最后一小节通过实例介绍了多线程/多进程技术在自动化测试项目中的应用。

第 12 章 Page Object 设计模式

页面对象（Page Object）设计模式，并越来越多的自动化测试人员所推崇。页面对象是 Selenium 实践中的一种最佳实践设计模式。在该设计中，功能类（PageObjects）所代表的应用程序的页面之间的逻辑关系。

12.1 认识 Page Object

使用 Page Object 设计模式有如下优点：

- 减少了代码的重复
- 让测试更具可读性和强大的
- 提高了测试的可维护性，特别是当有频繁变化的 AUT(被测试的应用程序)。

当你在为 Web 页面编写测试时，你需要操作该 Web 页面上的元素来点击链接或验证显示的内容。然而，如果你在测试代码中直接操作 HTML 元素，那么你的代码是及其脆弱的，因为 UI 会经常变动。一个 page 对象可以封装一个 HTML 页面或部分页面，你可以通过提供的应用程序特定的 API 来操作页面元素，而不需要在 HTML 中四处搜寻。

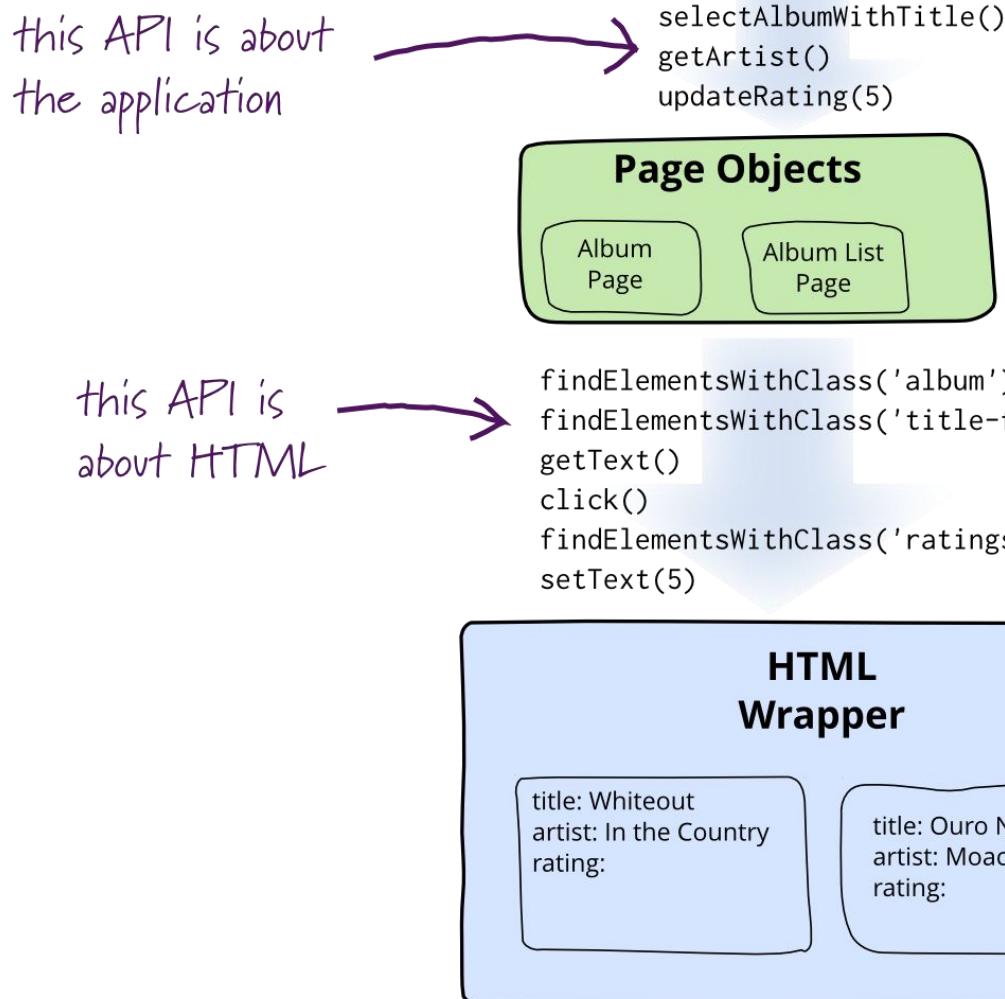


图 11.1 Page Object 原理

page 对象的一个基本经验法则是，凡是人类能做的事，page 对象通过软件客户端都能够做到。它也应当提供一个易于编程的接口并隐藏窗口中低层的部件。所以访问一个文本框应该通过一个访问方法（accessor method）来实现字符串的获取与返回，复选框应当使用布尔值，按钮应当被表示为行为导向的方法名。page 对象应当将在 GUI 控件上所有查询和操作数据的行为封装为方法。一个好的经验法则是，即使改变具体的控制，page 对象的接口也不应当发生变化。

尽管该术语是“页面”对象，并不意味着针对每个页面建立一个这样的对象，比如页面有重要意义的元素可以独立为一个 page 对象。经验法则的目的在于通过给页面建模，从而对应用程序的使用者变得有意义。

12.2 Page Object 实例

我们以 126 邮箱登录为例，通过 Page Object 设计模式来实现：

login126_po.py

```
#coding=utf-8
from selenium import webdriver
from selenium.webdriver.common.by import By
from time import sleep

class Page(object):
    """
    基本类，用于所页面的继承
    """
    login_url = 'http://www.126.com'

    def __init__(self, selenium_driver, base_url=login_url, parent=None):
        self.base_url = base_url
        self.driver = selenium_driver
        self.timeout = 30
        self.parent = parent
        self.tabs = {}

    def _open(self,url):
        url = self.base_url + url
        self.driver.get(url)
        assert self.on_page(),'Did not land on %s' % url

    def find_element(self,*loc):
        return self.driver.find_element(*loc)

    def open(self):
        self._open(self.url)

    def on_page(self):
        return self.driver.current_url == (self.base_url + self.url)

    def script(self,src):
        return self.driver.execute_script(src)

    def send_keys(self,loc,value, clear_first=True, click_first=True):
        try:
            loc = getattr(self, '_%s' % loc)
            if click_first:
                self.find_element(*loc).click()
            if clear_first:
                self.find_element(*loc).clear()
            self.find_element(*loc).send_keys(value)
        except:
            pass
```

```

except AttributeError:
    print '%s page does not have "%s" locator' %(self,loc)

class LoginPage(Page):
    """
    登录页面模型
    """
    url = ''

    #定位器
    username_loc = (By.ID,"idInput")
    password_loc = (By.ID,"pwdInput")
    submit_loc = (By.ID,"loginBtn")

    #Action
    def open(self):
        self._open(self.url)

    def type_username(self,username):
        self.find_element(*self.username_loc).send_keys(username)

    def type_password(self,password):
        self.find_element(*self.password_loc).send_keys(password)

    def submit(self):
        self.find_element(*self.submit_loc).click()

def test_user_login(driver, username, password):
    """
    测试获取的用户名密码 是否可以登录
    """
    login_page = LoginPage(driver)
    login_page.open()
    login_page.type_username(username)
    login_page.type_password(password)
    login_page.submit()
    sleep(3)
    assert(username == 'testingwtb@126.com'),u"用户名不匹配，登录失败!"

def main():
    try:
        # Selenium

```

```

driver = webdriver.Firefox()
username = 'testingwtb'
password = 'a123456'
test_user_login(driver, username, password)
finally:
    # 关闭浏览器窗口
    driver.close()

if __name__ == '__main__':
    main()

```

本来几行代码就可以实现的 126 邮箱登录脚本，通过 Page Object 模式实现之后变得复杂了很多。下面我们就对其进行逐段分析，来体会这样设计的好处。

创建 page 类

login126_po.py

```

.....
class Page(object):
    """
    基本类，用于所页面的继承
    """
    login_url = 'http://www.126.com'

    def __init__(self, selenium_driver, base_url=login_url, parent=None):
        self.base_url = base_url
        self.driver = selenium_driver
        self.timeout = 30
        self.parent = parent
        self.tabs = {}

    def _open(self, url):
        url = self.base_url + url
        self.driver.get(url)
        assert self.on_page(), 'Did not land on %s' % url

    def find_element(self, *loc):
        return self.driver.find_element(*loc)

    def open(self):
        self._open(self.url)

    def on_page(self):
        return self.driver.current_url == (self.base_url + self.url)

```

```

def script(self,src):
    return self.driver.execute_script(src)

def send_keys(self,loc,value, clear_first=True, click_first=True):
    try:
        loc = getattr(self, '_%s' % loc)
        if click_first:
            self.find_element(*loc).click()
        if clear_first:
            self.find_element(*loc).clear()
        self.find_element(*loc).send_keys(value)
    except AttributeError:
        print '%s page does not have "%s" locator' %(self,loc)

```

.....

我们创建了一个基本类 Page，Page 类继承自 object 类，在面向对象的世界里 object 类是所有的类的祖先类，换名话说所有的类都是从 object 类继承而来。

创建__init__() 方法做了一些初始化工作，定义驱动（driver）和基本的 RUL 等。 创建_open() 方法用于接收 URL 地址，find_element()方法用于元素的定位，script() 方法用于调用 javascript 程序，send_key() 方法用于接收用户键盘输入。

创建 LoginPage 类

Page 类中定义的这些方法都是页面操作的基本方法，那么下面根据登录页的特点再创建LoginPage 类：

login126_po.py

```

class LoginPage(Page):
    """
    登录页面模型
    """
    url = ''

    # 定位器
    username_loc = (By.ID,"idInput")
    password_loc = (By.ID,"pwdInput")
    submit_loc = (By.ID,"loginBtn")

    # Action
    def open(self):
        self._open(self.url)

```

```

def type_username(self,username):
    self.find_element(*self.username_loc).send_keys(username)

def type_password(self,password):
    self.find_element(*self.password_loc).send_keys(password)

def submit(self):
    self.find_element(*self.submit_loc).click()

.....

```

在 LoginPage 类中，我们的操作对象就具体了很多，首先我们定义了用户名输入框，密码输入框，和登录按钮的元素定位。

open() 方法用于调用父类（Page）的_open()方法，type_username() 方法调用父类（Page）的find_element()方法，用于接收当前类中所定义的用户名的元素定位 username_loc，接着 send_keys() 用于接收具体的用户名参数。

接着定义的 type_password 和 submit 方法类似。

创建 test_user_login() 函数

login126_po.py

```

.....
def test_user_login(driver, username, password):
    """
    测试获取的用户名密码 是否可以登录
    """
    login_page = LoginPage(driver)
    login_page.open()
    login_page.type_username(username)
    login_page.type_password(password)
    login_page.submit()
    sleep(3)
    assert(username == 'testingwtb@126.com'), u"用户名不匹配，登录失败!"

.....

```

test_user_login() 函数用户接收三个参数驱动（driver），用户名（username），密码（password）；定义 login_page 变量用于指向 LoginPage() 类，接着就可以调用 LoginPage() 类中所定义的方法去实现操作。

创建 main() 函数

login126_po.py

```
.....  
def main():  
    try:  
        # Selenium  
        driver = webdriver.Firefox()  
        username = 'testingwtb'  
        password = 'a123456'  
        test_user_login(driver, username, password)  
    finally:  
        # 关闭浏览器窗口  
        driver.close()  
  
if __name__ == '__main__':  
    main()
```

main() 函数所完成的工作非常简单，来定义具体需要打开的浏览器，以及登录的用户名和密码，这也是用户所关心内容，从而将这些数据传递给 test_user_login() 函数，至于 test_user_login() 函数将这些数据如何分配处理。对于用户（main() 函数）来说并不关心。

那么对于 def test_user_login() 方法来得到数据后，关心的是调用哪些方法，并且将这些数据传给这些方法。

那么对于 LoginPage() 类来说，他需要关注页面元素的变化。根据元素位置或属性变化实时的来调整元素的定位方式。好吧谁让叫 LoginPage 呢，这是他的职责。

一个有意见分歧的地方是 page 对象是否应自身包含断言，或者仅仅提供数据给测试脚本来设置断言。在 page 对象中包含断言的倡导者认为，这有助于避免在测试脚本中出现重复的断言，可以更容易的提供更好的错误信息，并且提供更接近只做不问风格的 API。不在 page 对象中包含断言的倡导者则认为，包含断言会混合访问页面数据和实现断言逻辑的职责，并且导致 page 对象过于臃肿。

我赞成在 page 对象中不包含断言。我认为你可以通过为常用的断言提供断言库的方式来消除重复，这还可以提供更好的诊断。从站在用户的角度去自动化的观点来看，判断是否登录成功是用户需要做的事情。不应该交给 page 对象来完成。

使用 PO 之后的另外一个大好处，就是有助于降低冗余，如果我在 10 个用例里面都需要输入不同用户名密码登录，那么我 main() 方法将写起来非常简便。

因此，Page Object 模型的作用在一个测试人员自己写主场景测试案例时是不容易体会到的，因为他不需要和开发、业务交流案例，他也不会写很多重复动作。但是，我相信，当他真正开始尝试 ATDD，BDD 或 SbE 时，当他开始写一些重要的异常分支流程时，当他开始为新需求频繁维护修改案例时，我想他会更意识到 Page Object 的作用。

最后一句，Page Object 不是万灵药，也不是唯一真理，提高测试案例可读性，避免案例步骤冗余才是终极目标。

第 13 章 BDD 框架之 lettuce 入门

相信不少同学对于 TDD/BDD（测试驱动开发/行为驱动开发）等技术既熟悉又陌生，熟悉的是时常能通过各种技术文章了解到这概念，陌生的是很少见到有项目中真正使用这些技术。本章将揭开 BDD 行为驱动开发的面纱，带领读者理解如何通过 BDD 进行开发。

学习 BDD 行为驱动开发，Ruby 的 cucumber 是个相对比较成熟的 BDD 框架，但我是 Python 流的，自然找来它的兄 lettuce，从官方版本（0.2.20）来看确实够年轻的，不过有 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）框架。用于 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，它有很好的可扩展性、可读性，它允许我们用自然语言去描述一个系统的功能，你不能想象这些描述可以自动测试你的系统。

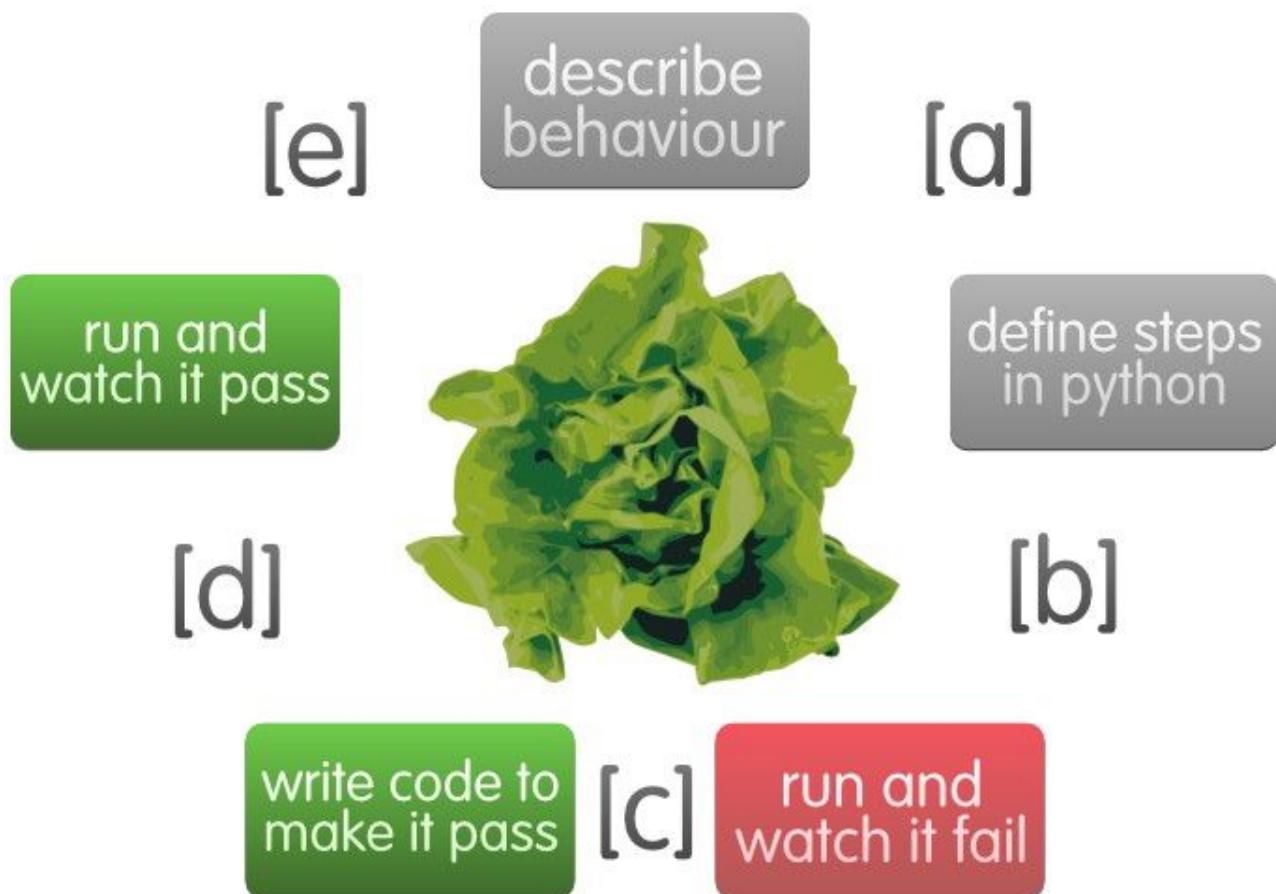


图 13.1 lettuce 功能

13.1 安装 lettuce

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。最为方便是通过 pip 安装 lettuce:

```
cmd.exe
C:\Python27\Lib\site-packages>pip install lettuce
```

安装好 lettuce 后，打开 cmd. exe 在任意目录下输入“lettuce”命令：



图 13.2 lettuce 命令

如果出如图13.2的提示，说明 lettuce 已经安装成功；因为还没有创建 lettuce 项目，所会提示：“哎呀！在 features 目录下不能发现 features”，lettuce 期望在当前目录下创建 features 子目录

13.2 认识 BDD(lettuce)

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! =2×1=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 语言实现阶乘两种方式:

factorial.py

```
#coding=utf-8

#循环实现阶乘
def f1(n):
    c = 1
    for i in range(n):
        i=i+1
        c=c*i
    return c

#递归实现阶乘
def f2(n):
    if n > 1:
        return n*f2(n-1)
    else:
        return 1

#调用方法
print f1(10)
print f2(10)
```

基于我们理解上面阶乘的基础上，来看看 BDD 是如何实现的。

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

zero.feature

```
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

对上 zero.feature 的描述，我们来做个简单的翻译：

zero.feature

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，不然一千个人会有一千种描述，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	unittest
Feature (功能)	test suite (测试用例集)
Scenario (情景)	test case (测试用例)
Given (给定条件)	setup (测试步骤)

When (当)	test (触发被测事件)
Then (测)	assert (断言, 验证结果)

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 文件来编写实现阶乘的代码。

steps.py

```
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Python代码相差甚远, 下面我们逐步分析, steps.py 和 zero.feature 是如何产生联系的。

```
from lettuce import *
```

引入 lettuce 下面的所有包类和方法。

第一步:

steps.py

```
.....
@step('I have the number (\d+)')
def have_the_number(step, number):
```

```
world.number = int(number)
.....
```

@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。请读者参考其它资料学习 Python 的正则表达式。

定义一个 have_the_number 函数, 把@step(I have the number (\d+)) 匹配到的数字 (0) 作为 number 的参数, 然后将其转换成整型 (int) 赋值给 world.number 变量。

第二步:

steps.py

```
.....  
@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:

steps.py

```
.....  
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。(处理结果给了第二步的 world.number)

第四步:

steps.py

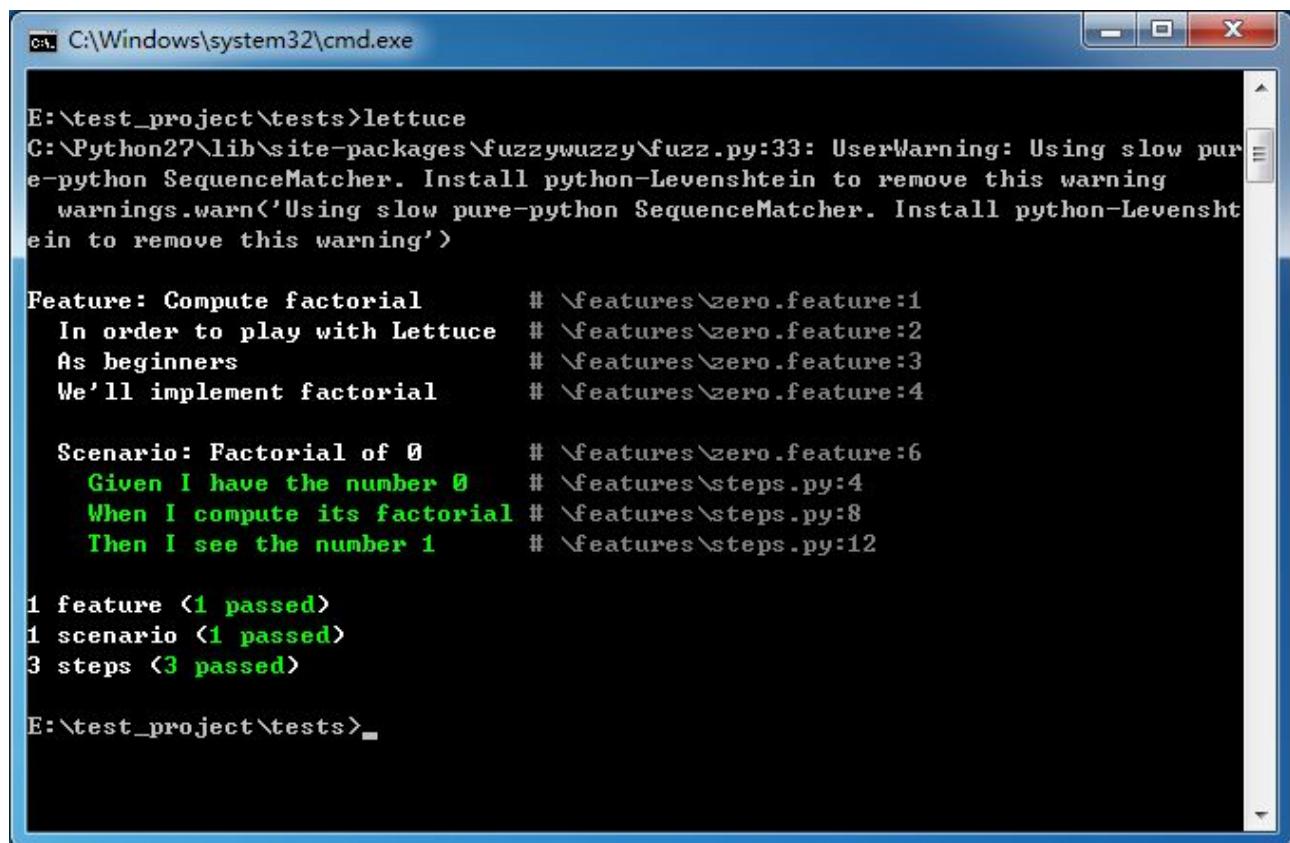
```
.....
@step('I see the number (\d+)')
def check_number(step, expected):
    expected = int(expected)
    assert world.number == expected, "Got %d" % world.number
.....
```

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+) 对应的就是 zero.feature 文件中的第八句：Then I see the number 1。

运行 lettuce

切换到 tests 目录下，运行 lettuce 命令：



```
E:\test_project\tests>lettuce
C:\Python27\lib\site-packages\fuzzywuzzy\fuzz.py:33: UserWarning: Using slow pure-python SequenceMatcher. Install python-Levenshtein to remove this warning
  warnings.warn('Using slow pure-python SequenceMatcher. Install python-Levenshtein to remove this warning')

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:6
    Given I have the number 0   # \features\steps.py:4
    When I compute its factorial # \features\steps.py:8
    Then I see the number 1     # \features\steps.py:12

1 feature <1 passed>
1 scenario <1 passed>
3 steps <3 passed>

E:\test_project\tests>_
```

图 13.3 执行 lettuce

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 程序里的哪一行代码。

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

13.3 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）：

zero.feature

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

再次执行 lettuce 进行测试：

```

Scenario: Factorial of 0      # \features\zero.feature:6
  Given I have the number 0    # \features\steps.py:4
  When I compute its factorial # \features\steps.py:8
  Then I see the number 1      # \features\steps.py:12

Scenario: Factorial of 1      # \features\zero.feature:11
  Given I have the number 1    # \features\steps.py:4
  When I compute its factorial # \features\steps.py:8
  Then I see the number 1      # \features\steps.py:12

Scenario: Factorial of 2      # \features\zero.feature:16
  Given I have the number 2    # \features\steps.py:4
  When I compute its factorial # \features\steps.py:8
  Then I see the number 2      # \features\steps.py:12

Scenario: Factorial of 3      # \features\zero.feature:21
  Given I have the number 3    # \features\steps.py:4
  When I compute its factorial # \features\steps.py:8
  Then I see the number 6      # \features\steps.py:12
Traceback (most recent call last):
  File "C:\Python27\lib\site-packages\lettuce\core.py", line 144, in __call__
    ret = self.function(self.step, *args, **kw)
  File "E:\test_project\tests\features\steps.py", line 14, in check_number
    assert world.number == expected, "Got %d" % world.number
AssertionError: Got 3

1 feature <0 passed>
4 scenarios <3 passed>
12 steps <1 failed, 11 passed>

List of failed scenarios:
  Scenario: Factorial of 3      # \features\zero.feature:21

```

图 13.3 执行 lettuce 失败

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，steps.py 文件中 factorial() 函数，使其真正可以处理阶乘：

steps.py

```

.....
def factorial(number):
    number = int(number)
    if (number == 0) or (number == 1):
        return 1
    else:
        return number*factorial(number-1)

```

因为 0 和 1 的阶乘都为 1，之前的代码直接判断计算的数字是否为 0 或 1，然后返回 1；如果是 2 的话直接返回数字本身，对于 2 以上的数字并没进行计算。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。再来执行 lettuce 进行验证吧！

```

We'll implement factorial      # \features\zero.feature:4

Scenario: Factorial of 0      # \features\zero.feature:6
  Given I have the number 0    # \features\steps.py:4
  When I compute its factorial # \features\steps.py:8
  Then I see the number 1      # \features\steps.py:12

Scenario: Factorial of 1      # \features\zero.feature:11
  Given I have the number 1    # \features\steps.py:4
  When I compute its factorial # \features\steps.py:8
  Then I see the number 1      # \features\steps.py:12

Scenario: Factorial of 2      # \features\zero.feature:16
  Given I have the number 2    # \features\steps.py:4
  When I compute its factorial # \features\steps.py:8
  Then I see the number 2      # \features\steps.py:12

Scenario: Factorial of 3      # \features\zero.feature:21
  Given I have the number 3    # \features\steps.py:4
  When I compute its factorial # \features\steps.py:8
  Then I see the number 6      # \features\steps.py:12

1 feature <1 passed>
4 scenarios <4 passed>
12 steps <12 passed>

```

图 13.4 再次执行 lettuce 成功

13.4 lettuce 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道，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：

cmd.exe

```
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。

13.5 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

参考 13.1 节通过 pip 安装。

第二步，安装 lettuce_webdriver

lettuce_webdriver 下载地址：https://pypi.python.org/pypi/lettuce_webdriver

cmd.exe

```
C:\Python27\Lib\site-packages>pip install lettuce_webdriver
```

第三步，安装 nose

nose 继承自 unittest，属于第三方的 Python 单元测试框架，且更容易使用，lettuce_webdriver 的运行依赖于 nose 模块。

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

cmd.exe

```
C:\Python27\Lib\site-packages>pip install nose
```

下面以为百度搜索为例，好吧！谁让这个例子能简单明了的讲解问题呢。所以，我们再次以百度搜索为例。

创建目录结构如下：

... /test/features/baidu.feature

/step_definitions/**steps.py**

/support/**terrain.py**

先来回顾一下我们去写百度搜索脚本的过程：

baidu.feature

功能：百度搜索测试用例

场景：搜索 selenium

我去访问百度的“<http://www.badiu.com>”

通过 id 为 “kw” 找到输入框并输入 “selenium” 关键字

点击 id 为 “su” 按钮

然后，我在搜索结果上找到了 “seleniumhq.org” 的网址

最后，我关闭了浏览器

确定了我们要做的事情，下面将其转换成 BDD 的描述文件。

baidu.feature

Feature: Baidu search test case

Scenario: search selenium

Given I go to "<http://www.baidu.com>"

When I fill in field with id "kw1" with "selenium"

And I click id "su1" with baidu once

Then I should see "seleniumhq.org" within 2 second

下面根据行为描述文件在 step_definitions 目录下编写相应的测试脚本：

steps.py

```
#coding=utf-8
from lettuce import *
from lettuce.webdriver.util import assert_false
from lettuce.webdriver.util import AssertContextManager

def input_frame(browser, attribute):
    xpath = "//input[@id='%s']" % attribute
    elems = browser.find_elements_by_xpath(xpath)
    return elems[0] if elems else False

def click_button(browser, attribute):
    xpath = "//input[@id='%s']" % attribute
```

```

elems = browser.find_elements_by_xpath(xpath)
return elems[0] if elems else False

#定位输入框输入关键字
@step('I fill in field with id "(.*?)" with "(.*?)"')
def baidu_text(step,field_name,value):
    with AssertContextManager(step):
        text_field = input_frame(world.browser, field_name)
        text_field.clear()
        text_field.send_keys(value)

#点击“百度一下”按钮
@step('I click id "(.*?)" with baidu once')
def baidu_click(step,field_name):
    with AssertContextManager(step):
        click_field = click_button(world.browser,field_name)
        click_field.click()

#关闭浏览器
@step('I close browser')
def close_browser(step):
    world.browser.quit()

```

在 support 目录下创建 terrain.py 文件，用于定于测试脚本的基本配置。

terrain.py

```

from selenium import webdriver
from lettuce import before, world
import lettuce.webdriver.webdriver

@Before.all
def setup_browser():
    world.browser = webdriver.Firefox()

```

terrain 文件配置浏览器驱动，可以作用于所有测试用例。

在.../test/目录下输入 lettuce 命令，运行结果如图 13.5

```
fnngj@fnngj-VirtualBox: ~/selenium/test
1 feature (1 passed)
1 scenario (1 passed)
4 steps (4 passed)
fnngj@fnngj-VirtualBox:~/selenium/test$ lettuce

Feature: Go to baidu                                # features/baidu.feature:
1

  Scenario: Visit baidu                            # features/baidu.feature:
3
    Given I go to "http://www.baidu.com/"          # usr/local/lib/python2.7
    Given I go to "http://www.baidu.com/"          # usr/local/lib/python2.7
/dist-packages/lettuce_webdriver/webdriver.py:76
    When I fill in field with id "kw1" with "selenium" # features/setps.py:16
    And I click id "su1" with baidu once           # features/setps.py:23
    Then I should see "seleniumhq.org" within 2 second # usr/local/lib/python2.7
    Then I should see "seleniumhq.org" within 2 second # usr/local/lib/python2.7
/dist-packages/lettuce_webdriver/webdriver.py:152
    Then I close browser                           # features/setps.py:29

1 feature (1 passed)
1 scenario (1 passed)
5 steps (5 passed)
fnngj@fnngj-VirtualBox:~/selenium/test$
```

图 13.5 执行 lettuce WebDriver 测试脚本

对于当前测试用例来说，添加新的测试场景也非常简单，我们只用修改 `baidu.feature` 即可：

baidu.feature

```
Feature: Baidu search test case

Scenario: search selenium
  Given I go to "http://www.baidu.com/"
  When I fill in field with id "kw1" with "selenium"
  And I click id "su1" with baidu once
  Then I should see "seleniumhq.org" within 2 second

Scenario: search lettuce.webdriver
  Given I go to "http://www.baidu.com/"
  When I fill in field with id "kw1" with "lettuce.webdriver"
  And I click id "su1" with baidu once
  Then I should see "pypi.Python.org" within 2 second
  Then I close browser
```

再次执行 lettuce 命令执行自动化测试脚本：

```
fnngj@fnngj-VirtualBox: ~/selenium/test

Scenario: Visit baidu # features/baidu
.feature:3
  Given I go to "http://www.baidu.com/" # usr/local/lib/
  Given I go to "http://www.baidu.com/" # usr/local/lib/
python2.7/dist-packages/lettuce_webdriver/webdriver.py:76
  When I fill in field with id "kw1" with "selenium" # features/setps
  When I fill in field with id "kw1" with "selenium" # features/setps
.py:16
  And I click id "sui" with baidu once # features/setps
  And I click id "sui" with baidu once # features/setps
.py:23
  Then I should see "seleniumhq.org" within 2 second # usr/local/lib/
  Then I should see "seleniumhq.org" within 2 second # usr/local/lib/
python2.7/dist-packages/lettuce_webdriver/webdriver.py:152

Scenario: search lettuce.webdriver # features/baidu
.feature:10
  Given I go to "http://www.baidu.com/" # usr/local/lib/
  Given I go to "http://www.baidu.com/" # usr/local/lib/
python2.7/dist-packages/lettuce_webdriver/webdriver.py:76
  When I fill in field with id "kw1" with "lettuce.webdriver" # features/setps
  When I fill in field with id "kw1" with "lettuce.webdriver" # features/setps
.py:16
  And I click id "sui" with baidu once # features/setps
  And I click id "sui" with baidu once # features/setps
.py:23
  Then I should see "pypi.python.org" within 2 second # usr/local/lib/
  Then I should see "pypi.python.org" within 2 second # usr/local/lib/
python2.7/dist-packages/lettuce_webdriver/webdriver.py:152
  Then I close browser # features/setps
  Then I close browser # features/setps
.py:29

1 feature (1 passed)
2 scenarios (2 passed)
9 steps (9 passed)
fnngj@fnngj-VirtualBox:~/selenium/test$
```

图 13.6 执行 lettuce WebDriver 测试脚本

通过 lettuce 框架来编写自动化脚本将是一件非常有趣的事儿。

第 14 章 Git 管理项目

当我们的自动化测试项目的实践中逐渐成熟并形成一定规模之后，自动化测试的脚本开发与维护就不是一人所能完成的，那么必定会有新人员参与到自动化测试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一层学习难度，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。

14.1 Git/GitCafe 托管测试项目

14.1.1 Git 环境配置

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 各版本下安装 Git:

```
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 14.1 Windows 下安装 Git

Ubuntu 下通过 ap-get 仓库安装 git。

Ubuntu

```
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。

2.1 进入 SSH 目录

Ubuntu

```
root@fnngj-H24X:/home/fnngj/Python/pyse# cd ~/.ssh
root@fnngj-H24X:~/ssh# pwd
/root/.ssh
```

2.2 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）

Ubuntu

```
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

3.1 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。



```
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCVkekJlT6UxK+/
hrbWqwr0q0wbttPjjrS0CWjifdiF+uCb36mKBs1
+Axz8FsVZrEzR84qkB0uFBu9zJ7stpNyZbiF86Nm44SqFM
+nvEmfg41CGcj5VIz9c49m7biepUrTGV3kiNpQiDaH81SzefzyY
+82NSrKW50pUmSU6TBacfja8VE8rZADkVrgVBc8FueASt/
U4sSkhWs1j8c4x0AhkNrgrgryA
+0Ep9bURIH0V0VE8rWp1NC1bChmZD0IvivZopBvJN9sBhMyXEWytFIIYst+cd5
+0ez6Frz0J04T0mfh2SQfmpMK0xlXL7DEG7S6pQ0l0s3jq/277kuaYhYrhf
fnngj@126.com
```

图 14.2 查看生成的公钥

3.2 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 14.3 添加公钥到 GitCafe

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。

Ubuntu

```
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.
```

14.1.2 提交代码

首先，在 GitCafe 注册帐号并登录，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	选择文件 ... 尺寸: 128x128
初始化项目	<input 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="font-size: 1em; width: 100px; height: 40px; border-radius: 10px; border: none; color: white; text-decoration: none; font-weight: bold; cursor: pointer;" type="button" value="创建"/>	

图 14.4 在 GitCafe 上创建项目

填写项目相关信息点击“创建”，创建完成如图14.5：



图 14.5 项目创建完成

Git 的基本命令：

在任意目录下输入“git”，查看 git 所提供的命令。

Ubuntu

```
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：

Ubuntu

```

root@fnngj-H24X:/home/fnngj# git config --global user.name 'fnngj'
root@fnngj-H24X:/home/fnngj# git config --global user.email 'fnngj@126.com'

```

(注：这一步必不可少！)

在本地创建一个项目：

Ubuntu

```

root@fnngj-H24X:/home/fnngj/pyse# ls
baidu.py  baidu.py~

```

创建 pyse 目录，在目录下创建了一个简单的测试脚本 baidu.py。

Ubuntu

```

root@fnngj-H24X:/home/fnngj/pyse# git init
初始化空的 Git 版本库于 /home/fnngj/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。

Ubuntu

```

root@fnngj-H24X:/home/fnngj/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 改的状态。); 更详细的告诉我可以通过 git add <file> 来对文件进行跟踪。

Ubuntu

```
root@fnngj-H24X:/home/fnngj/pyse# git add .
root@fnngj-H24X:/home/fnngj/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 . git add 命令可以对指定指定的文件添加跟踪。例如: git add baidu.py。

“.” 点号表示对当前目录下的所有文件/文件夹进行跟踪，也就是提交给 Git 进行管理。

git status 通过 git status 命令查看当前 Git 仓库的信息。

Ubuntu

```
root@fnngj-H24X:/home/fnngj/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
```

git commit 将文件 (git add 进行管理的文件) 提交到本地仓库。-m 参数对本次的提交加以描述。一般提交的描述必不可少，从而方便可追溯每次提交都做了哪些修改。

提示信息告诉我，更改提交到 master 主分支，对2个文件做了修改，插入22行代码，修改的文件为 baidu.py。

准备工作已经完成，下面提交代码到 GigCafe:

Ubuntu

```
root@fnngj-H24X:/home/fnngj/pyse# git remote add origin
```

```
'git@gitcafe.com:fnngj/pyse.git'
```

```
root@fnngj-H24X:/home/fnngj/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那个仓库建立连接。

```
git push -u origin master
```

将本地的项目提交到远程仓库中。

现在已经可以访问 GitCafe 上看到我们提交的项目了！

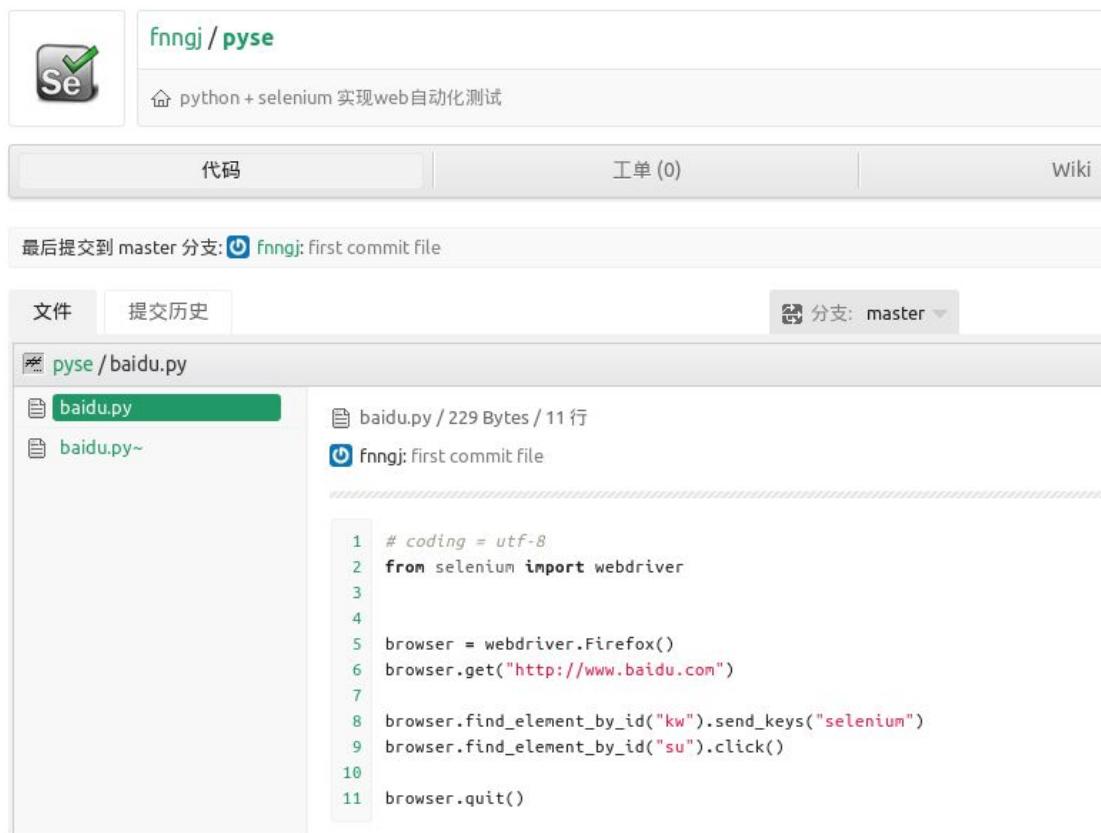


图 14.6 查看 GitCafe 上传的代码

14.1.3 更新代码

克隆代码

上面的代码编写与提交是我在公司的电脑上完成的，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家里的电脑上。

Git

```
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)
```

<git://gitcafe.com/fnngj/pyse.git>

为提交的项目在 GitCafe 上的地址。通过 git clone 命令就可以把项目克隆到本地。

更新项目

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。

```
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 14.7 查看本地项目更新

如图 14.7，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。

提交更新

去克隆一个项目很简单，提交一个项目比较麻烦，必须有相应的权限。因为我已经更换了电脑，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。

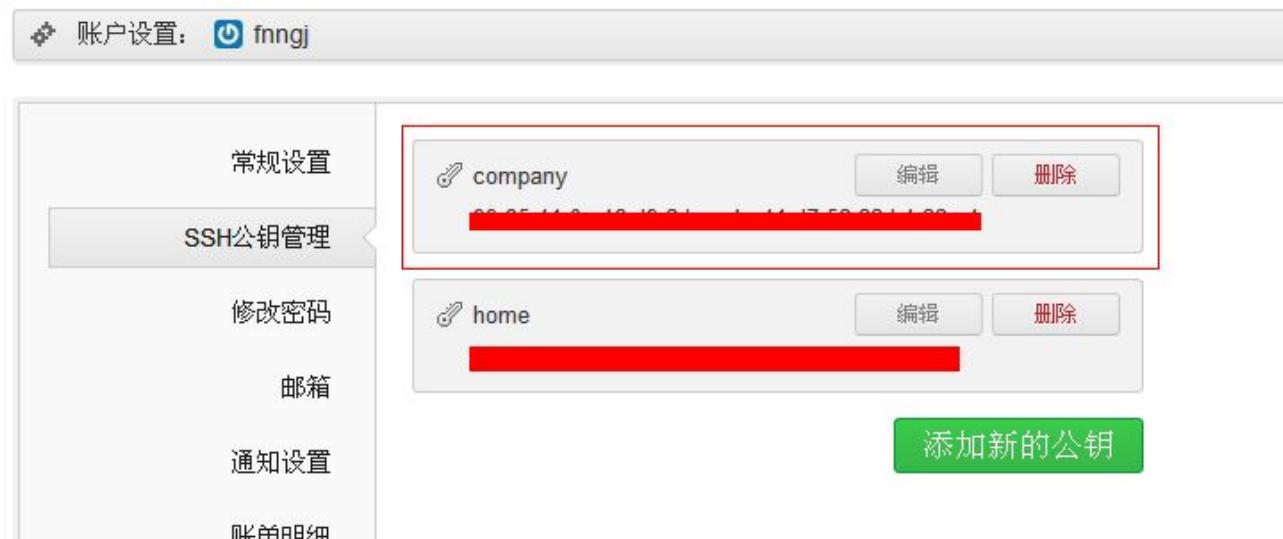


图 14.8 添加本地生成的公钥

测试链接：

```
Administrator@XP-201210.. /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.
```

提交代码到 GitCafe

```
Administrator@XP-201210.. /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-201210.. /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-201210.. /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-201210.. /d/selenium_use_case/Python_selenium2/pyse (master)
$ 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！但是这个 Git 地址只能用于克隆文件到本地，无法通过这个地址 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。

Git

```
Administrator@XP-201210.. /d/selenium_use_case/Python_selenium2/pyse (master)
$ git remote rm origin --删除 origin
```

```
Administrator@XP-201210.. /d/selenium_use_case/Python_selenium2/pyse (master)
$ git remote add origin 'git@gitcafe.com:fnngj/pyse.git' --重新链接
```

```
Administrator@XP-201210.. /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-201210.. /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.
```

这一次就可以正常 push 项目了，这也是新手容易迷惑的地方；在 GitCafe 中，克隆项目与提交项目的地址存在细微的差别，笔者已经在这里踩了坑，希望读者可以避免。

删除提交

通过 Git 管理的项目，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 命令来删除文件。

Git

```
Administrator@XP-201210.. /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-201210.. /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-201210.. /d/selenium_use_case/Python_selenium2/pyse (master)
$ git commit -m "delete baidu.py~"

Administrator@XP-201210.. /d/selenium_use_case/Python_selenium2/pyse (master)
$ git push origin master
```

pull 最新代码到本地

为了避免冲突我们应该形成良好的习惯，在每次 push 代码之先把服务器上最新的代码 pull 到本地。

Git

```
Administrator@XP-201210.. /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.
```

通过这一节的学习，我们可以自由的随时随地提交自己的代码到 GitCafe 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/>

14.2 Git/Git Server 搭建

上一节我们通过 CitCafe 作为 Git 服务器来提交项目，如果是内部项目话托管给 CitCafe 就变成公开的了，当然，也可以创建私有的服务器，但需要缴纳一定的费用，如果你刚好一台闲置的服务器的话来做为 Git 服务器是个不错的选择。

下面通过一种最简单的方式来构建我们的 Git Server。

我们在虚拟机（ubuntu）来创建一个 Git Server，本地通过模拟 A、B 两个用户向 Git Server 克隆项目，然后通过不同的用户 push 和 pull 项目。

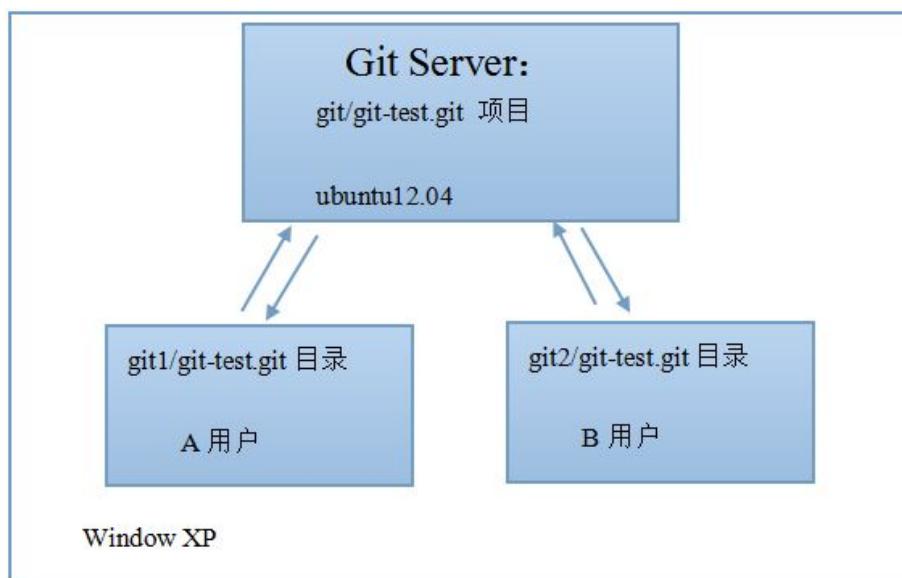


图 14.9 Git/Git Server 结构

通过上面结果图可以看到，本机系统为 windows XP，通过 Oracle VM VirtualBox 安装 Ubuntu12.04 系统，在 Ubuntu 上我们创建了 git server，本机系统上创建了分别两个目录（git、git2）来表示两个不同的用户向 git server 上提交和拉取项目。

14.2.1 配置 Git Server

首先登录 Ubuntu 系统创建 Git Server

1、安装 Git 和 openssh:

Ubuntu

```
fnngj@fnngj-VirtualBox: sudo apt-get install git-core  
fnngj@fnngj-VirtualBox: sudo apt-get install openssh-server  
fnngj@fnngj-VirtualBox: sudo apt-get install openssh-client
```

2、创建一个 Git 用户为: gituser, 密码为: git。

Ubuntu

```
fnngj@fnngj-VirtualBox:/home/fnngj# adduser gituser  
正在添加用户"gituser"...  
正在添加新组"gituser" (1001)...  
正在添加新用户"gituser" (1001) 到组"gituser"...  
创建主目录"/home/gituser"...  
正在从"/etc/skel"复制文件...  
输入新的 UNIX 密码:      ----输入密码  
重新输入新的 UNIX 密码:  ----确认密码  
passwd: 已成功更新密码  
正在改变 gituser 的用户信息  
请输入新值, 或直接敲回车键以使用默认值  
全名 []:  
房间号码 []:  
工作电话 []:  
家庭电话 []:  
其它 []:  
这些信息是否正确? [Y/n] Y
```

3、切换到 gituser 用户

Ubuntu

```
fnngj@fnngj-VirtualBox:~/git$ su - gituser  
密码:  
gituser@fnngj-VirtualBox:~$  
gituser@fnngj-VirtualBox:~$ pwd --查看当前路径  
/home/gituser  
gituser@fnngj-VirtualBox:~$ mkdir git --创建 git 目录  
gituser@fnngj-VirtualBox:~$ cd git/ --进入 git 目录
```

注意: 通过 root 用户切换到其它用户是不需要密码的, 如果通过普通用户切换到 root 用户或其它普通用户时需要输入用户密码。的 Linux 操作系统下 root 具有最高权利。

4、创建 git-test.git 项目

Ubuntu

```
gituser@fnngj-VirtualBox:~/git$ git --bare init git-test.git -- 创建
git-test.git 仓库
Initialized empty Git repository in /home/gituser/git/git-test.git/
```

现在创建了一个空项目。

14.2.2 检查连接

在访问虚拟机之前我们需要确认本地系统与虚拟机是否可以正常连接，一般虚拟机会提供了三种工作模式，host-only(主机模式)、NAT(网络地址转换模式)、bridged(桥接模式)，读者可以参考其它相关资料进行配置。

查看虚拟机 IP：

Ubuntu

```
root@fnngj-VirtualBox:/home/gituser/git# ifconfig
eth0      Link encap:以太网 硬件地址 08:00:27:46:84:bf
          inet 地址:172.16.10.3
          inet6 地址: fe80::a00:27ff:fe46:84bf/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 跳点数:1
          接收数据包:506949 错误:0 丢弃:6 过载:0 帧数:0
          发送数据包:10263 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:36217083 (36.2 MB)  发送字节:1305373 (1.3 MB)

lo       Link encap:本地环回
          inet 地址:127.0.0.1 掩码:255.0.0.0
          inet6 地址: ::1/128 Scope:Host
          UP LOOPBACK RUNNING MTU:16436 跳点数:1
          接收数据包:1396 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:1396 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:0
          接收字节:235438 (235.4 KB)  发送字节:235438 (235.4 KB)
```

在 linux 下可以通过 ifconfig 命令查看本地网络配置，通过上面的信息可以了解到当前的虚拟机 IP 为：172.16.10.3

本地 ping 虚拟机 IP 地址：

```

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>ping 172.16.10.3

Pinging 172.16.10.3 with 32 bytes of data:

Reply from 172.16.10.3: bytes=32 time<1ms TTL=64

Ping statistics for 172.16.10.3:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms

C:\Documents and Settings\Administrator>_

```

本地系统通过 ping 命令可以访问到虚拟机，那么我们就可以正常进行后面的操作。

14.2.3 A 用户访问 Git 服务器

Git 服务器已经搭建完成，并且本机与服务器之间通信正常，下面模拟 A 用户操作 Git 项目。

克隆项目到本地：

Git

```

Administrator@XP-201210141900 /e/git1
$ git clone gituser@172.16.10.3:/home/gituser/git/git-test.git
Cloning into 'git-test.git'...
gituser@172.16.10.3's password: ---要求输入 gituser 密码
warning: You appear to have cloned an empty repository.
Checking connectivity... done.

```

```

Administrator@XP-201210141900 /e/git1
$ ls
git-test.git

```

```

Administrator@XP-201210141900 /e/git1
$ cd git-test.git/

```

Git 的常用命令我们已经比较熟悉了，git clone 可以将项目克隆到本地，有所不同的是这次我们克隆的是虚拟主机（172.16.10.3）下的（/home/gituser/git/git-test.git）目录下的项目，注意我们是

通过 gituser 用户访问的，所以需要输入 gituser 用户的密码（git）。

提交项目

我们在 git/git-test.git 目录下创建 hello.py 文件：

Git

```
Administrator@XP-201210141900 /e/git1/git-test (master)
$ git status
On branch master
Initial commit
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    hello.py
nothing added to commit but untracked files present (use "git add" to track)

Administrator@XP-201210141900 /e/git1/git-test (master)
$ git add .

Administrator@XP-201210141900 /e/git1/git-test (master)
$ git commit -m 'hello'
[master (root-commit) 4c2337b] hello
 1 file changed, 1 insertion(+)
 create mode 100644 hello.py

Administrator@XP-201210141900 /e/git1/git-test (master)
$ git push origin master
gituser@172.16.10.3's password: --要求输入 gituser 密码
Counting objects: 3, done.
Writing objects: 100% (3/3), 222 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To gituser@172.16.10.3:/home/gituser/git/git-test
 * [new branch]      master -> master
```

14.2.3 B 用户访问 Git 服务器

A 用户将 Git 服务器上的空项目克隆到本地，并且创建了项目进行了提交。接下来由 B 用户对项目进行操作。

克隆项目

Git

```
Administrator@XP-201210141900 /e/git2
$ git clone gituser@172.16.10.3:/home/gituser/git/git-test
Cloning into 'git-test'...
gituser@172.16.10.3's password:
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
Checking connectivity... done.
```

```
Administrator@XP-201210141900 /e/git2
$ ls
git-test
```

```
Administrator@XP-201210141900 /e/git2
$ cd git-test/
```

```
Administrator@XP-201210141900 /e/git2/git-test (master)
$ ls
hello.py
```

B 用户克隆项目的时候发现多了一个 hello.py 文件，是 A 由用户创建并提交的文件。

提交项目

B 用户在 git2/git-test.git 目录下又创建了一个 git.py 文件，然后向 git server 项目提交：

Git

```
Administrator@XP-201210141900 /e/git2/git-test (master)
$ git add .
```

```
Administrator@XP-201210141900 /e/git2/git-test (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)
    new file:   git.py
```

```
Administrator@XP-201210141900 /e/git2/git-test (master)
$ git commit -m 'add git file'
[master f29b6df] add git file
 1 file changed, 1 insertion(+)
 create mode 100644 git.py
```

```
Administrator@XP-201210141900 /e/git2/git-test (master)
$ git push origin master
gituser@172.16.10.3's password: --要求输入 gituser 密码
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 297 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To gituser@172.16.10.3:/home/gituser/git/git-test
  4c2337b..f29b6df master -> master
```

A 用户拉取 (pull) 项目

最后，我们再切换回 A 用户，向 git 服务器拉取更新后的项目。

Git

```
Administrator@XP-201210141900 /e/git1/git-test (master)
$ git pull origin master
gituser@172.16.10.3's password: --输入 gituser 用户密码
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From 172.16.10.3:/home/gituser/git/git-test
 * branch      master    -> FETCH_HEAD
   4c2337b..f29b6df  master    -> origin/master
Updating 4c2337b..f29b6df
Fast-forward
 git.py | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 git.py

Administrator@XP-201210141900 /e/git1/git-test (master)
$ ls
git.py  hello.py
```

通过 pull 命令来拉取 B 用户所提交的 git.py 文件。最终，我们达到多人协作开发的目的。

需要强调的是我们的 A、B 两个用户与 git server 保持通信所使用的是 SSH 协议，用户为 gituser，每次在提交和拉取的时候都需要输入 gituser 用户密码。

如果读者想实现更强大的 git 管理可以使用：

- 使用 gitosis 来管理(gitosis 可以设定到“谁”可以存取此专案)
- 使用 gitolite 来管理(gitolite 可以设定“谁”可以存取此专案，而且，可以设定只能存取哪个 branch 等路径)

第 15 章 持续集成 Jenkins 入门

提到持续集成（CI---Continuous integration）的概念，相信读者并不陌生，但真正在项目中用起来的并不多，持续集成的作用以及给项目带来的好处是值得被推广和使用。当然，并不是好的技术和工具就适合所以的项目。这一章我将带领读者一起领略持续集成的面纱。这里我们将选用当前主流的持续集成工具---Jenkins。

什么是持续集成？

集成软件的过程不是新问题，如果项目开发的规模比较小，比如一个人的项目，如果它对外部系统的依赖很小，那么软件集成不是问题，但是随着软件项目复杂度的增加（即使增加一个人），就会对集成和确保软件组件能够在一起工作提出了更多的要求—要早集成，常集成。早集成，频繁的集成帮助项目在早期发现项目风险和质量问题，如果到后期才发现这些问题，解决问题代价很大，很有可能导致项目延期或者项目失败。

定义：

大师 Martin Fowler 对持续集成是这样定义的：持续集成是一种软件开发实践，即团队开发成员经常集成他们的工作，通常每个成员每天至少集成一次，也就意味着每天可能会发生多次集成。每次集成都通过自动化的构建（包括编译，发布，自动化测试）来验证，从而尽快地发现集成错误。许多团队发现这个过程可以大大减少集成的问题，让团队能够更快的开发内聚的软件。

什么是 Jenkins？

提到 jenkins 就不得不提另一个持续集成工具---hudson，hudson 是由 sun 公司开发，2010 年 sun 公司由 oracle 公司收购，oracle 公司声称对 hudson 拥有商标所有权；jenkins 是从 hudson 中分支出来的，并将继续走 open source 的道路。当然，二者都在持续开发中。

Jenkins 主要用于监视执行重复工作，如建立一个软件项目或工作运行的计划任务。当前 Jenkins 关注以下两个工作：

不断的进行项目的构建/测试软件：就像 CruiseControl 的或 DamageControl。概括地说，Jenkins 提供了一个易于使用的所谓的持续集成系统，使开发人员更容易修改整合到项目中，并使它更容易为用户获得一个新的版本。自动，连续生成提高了生产率。

监控外部运行的作业：如计划任务作业和 procmail 的工作，即使是那些在远程机器上运行的执行。例如，cron 的，你收到的是定期的电子邮件，捕获输出，它是由你来看看他们的努力和注意的时候就坏了。Jenkins 保持这些产出和很容易让你注意到什么是错的。

15.1 环境搭建

Jenkins 是基于 Java 开发的一种持续集成工具，所以，Jenkins 需要 Java 环境，关于 Java 环境的配置我们在第九章使用 Selenium Grid 时已经做了介绍，这里就不在讲解。

1、安装 tomcat

tomcat 是真对 Java 的一个开源中间件服务器（容器），基于 Java Web 的项目需要借助 tomcat 才能运行起来。

tomcat 官方网站：<http://tomcat.apache.org/>

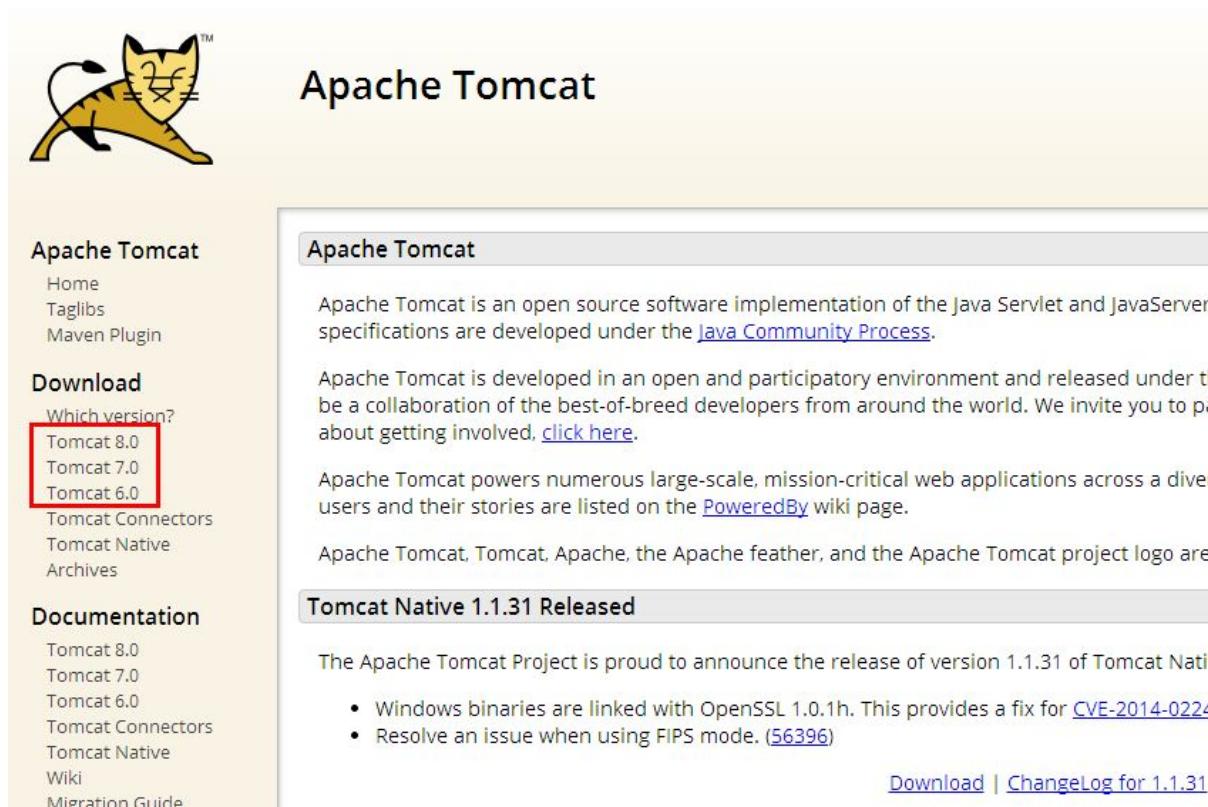
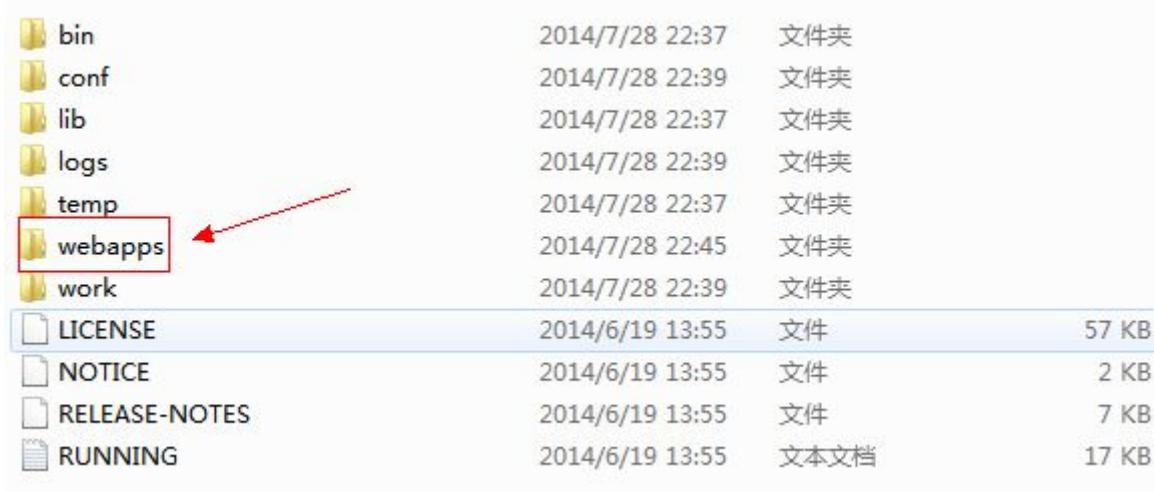


图 15.1 下载 tomcat

点击页面右侧 tomcat 版本进行下载，对下载的压缩包进行解压，目录结构如下：



bin	2014/7/28 22:37	文件夹
conf	2014/7/28 22:39	文件夹
lib	2014/7/28 22:37	文件夹
logs	2014/7/28 22:39	文件夹
temp	2014/7/28 22:37	文件夹
webapps	2014/7/28 22:45	文件夹
work	2014/7/28 22:39	文件夹
LICENSE	2014/6/19 13:55	文件 57 KB
NOTICE	2014/6/19 13:55	文件 2 KB
RELEASE-NOTES	2014/6/19 13:55	文件 7 KB
RUNNING	2014/6/19 13:55	文本文档 17 KB

图 15.2 webapps 目录用于 web 项目

通常将需要运行的应用放到 webapps/ 目录下，进入 bin/ 目录下，双击 startup.bat 来启动 tomcat 服务器。

2、安装 Jenkins

Jenkins 官方网站：<http://jenkins-ci.org/>

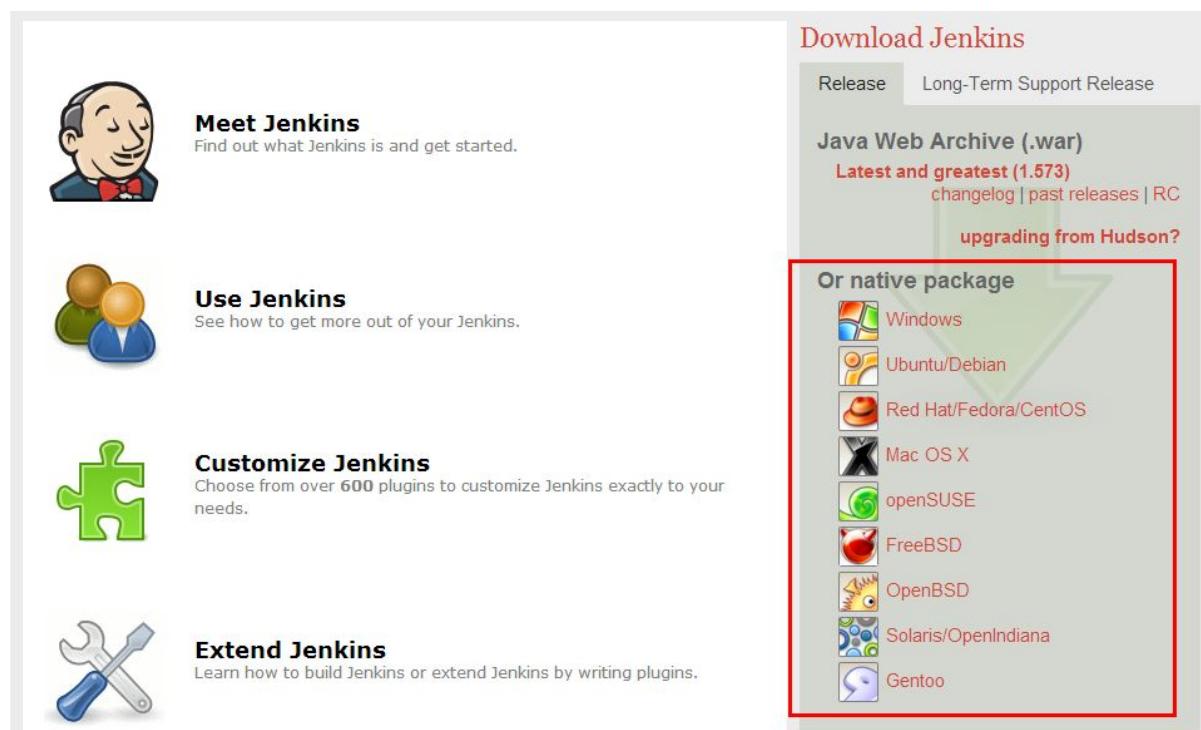


图 15.3 下载 Jenkins

打开首页后，我们可以在页面右侧找到不到系统所对应的 Jenkins 版本，读者可以根据自己的系统版本进行下载。

下载完成，双击进行安装：



图 15.4 双击 Jenkins 安装

点击“next”按钮，我们直接将其安装到 tomcat 的 webapps\ 目录下。

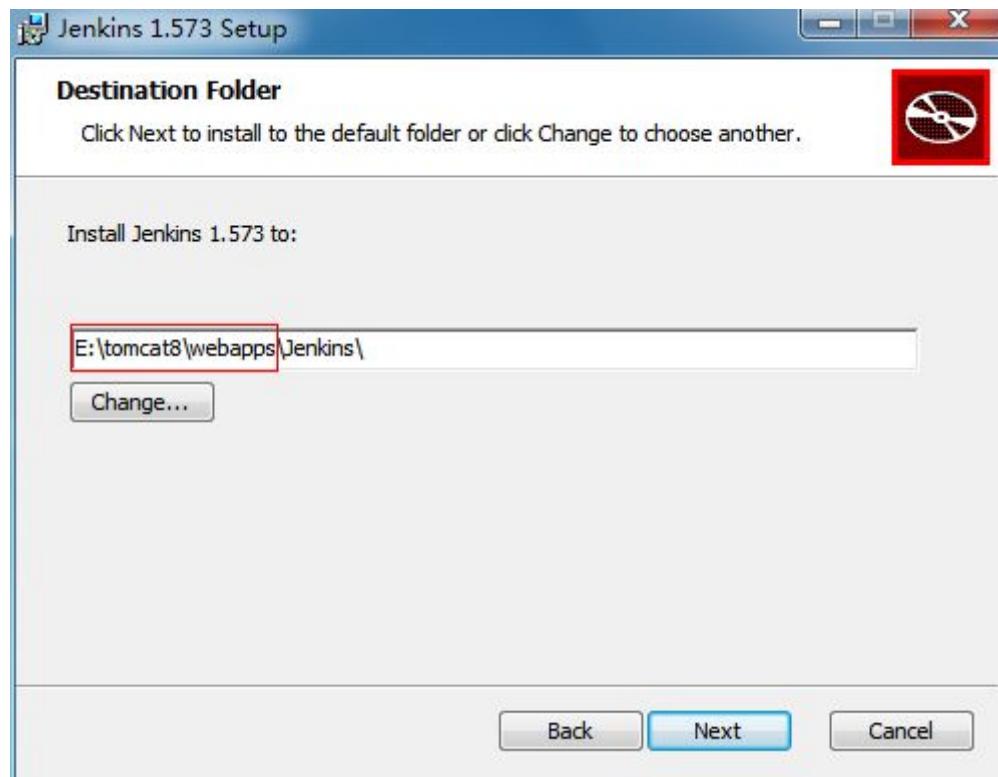


图 15.5 选择 tomcat 的 webapps 目录

运行 Jenkins

进行 tomcat 的 bin/ 目录下启动 startup.bat , 通过浏览器访问: <http://localhost:8080/>



图 15.6 访问 Jenkins

当你看到图 15.6, 说明我们的 Jenkins 已经安装成功, 下面就跟着我来创建我的任务吧。

15.2 创建任务

点击首页“创建一个新任务”的链接, 如下:



图 15.7 选择 Jenkins 任务类型

Jenkins 提供了四种类型的任务：

● 构建一个自由风格的软件项目

这是 Jenkins 的主要功能. Jenkins 将会结合任何 SCM 和任何构建系统来构建你的项目，甚至可以构建软件以外的系统.

● 构建一个 maven 项目

构建一个 maven 项目. Jenkins 利用你的 POM 文件, 这样可以大大减轻构建配置.

● 构建一个多配置项目

适用于多配置项目, 例如多环境测试, 平台指定构建, 等等.

● 监控一个外部的任务

这个类型的任务允许你记录执行在外部 Jenkins 的任务, 任务甚至运行在远程机器上. 这可以让 Jenkins 作为你所有自动构建系统的控制面板.

选择第一个项“构建一个自由风格的软件项目”，点击“OK”。进入项目的详细配置页面如下：

添加项目描述：

项目名称	pyse_test
描述	这是一个python + selenium 自动化脚本执行的项目。
[Escaped HTML] 预览	
<input type="checkbox"/> 丢弃旧的构建 ? <input type="checkbox"/> 参数化构建过程 ? <input type="checkbox"/> 关闭构建 (重新开启构建前不允许进行新的构建) ? <input type="checkbox"/> 在必要的时候并发构建 ?	

图 15.8 添加项目描述

高级选项:

高级项目选项

安静期 ?

安静期 秒钟 ▲ ▼

重试次数 ?

SCM 签出重试次数 ▲ ▼

该项目的上游项目正在构建时阻止该项目构建 ?

该项目的下游项目正在构建时阻止该项目构建 ?

使用自定义的工作空间 ?

目录 ?

Custom workspace is empty.

显示名称 ?

图 15.9 高级选项

源码管理:

源码管理

None ?

CVS ?

CVS Projectset ?

Subversion ?

图 15.10 源码管理

构建触发器:

构建触发器

Build after other projects are built ?

Projects to watch ?

No project specified

Trigger only if build is stable ?

Trigger even if the build is unstable ?

Trigger even if the build fails ?

Build periodically ?

日程表 ?

No schedules so will never run

Poll SCM ?

日程表 ?

图 15.11 构建触发器

增加构建步骤:

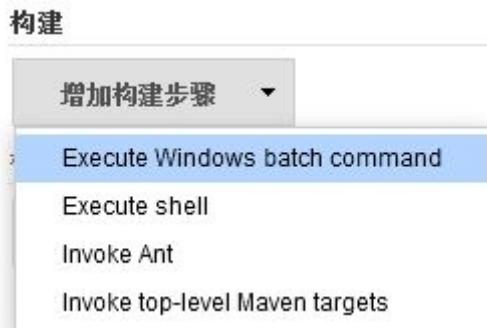


图 15.12 构建步骤

我们选择 execute Windows batch command , 来创建一个 windows 下的批处理。

假如, 我在 d:\ 盘根目录下有一个 test.py 的脚本, 我们要在命令提示符下来运行这个脚本, 如何做呢?



图 15.13 dos 命令

我需要两步操作, 首先, 需要切换到 D 盘, 接着通过 Python 执行 test.py 文件。

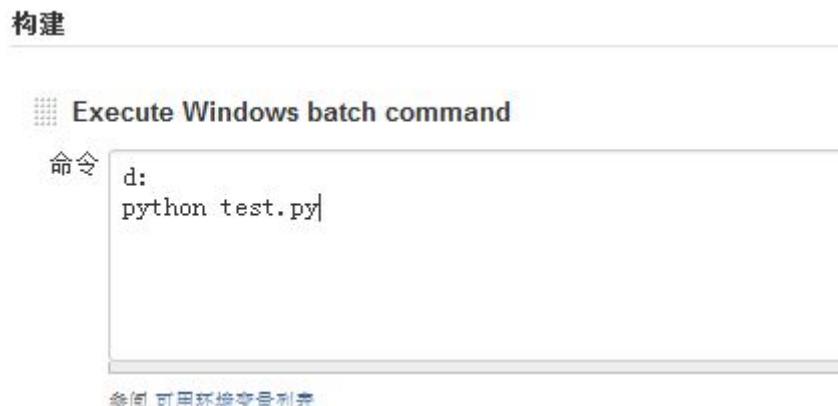


图 15.14 构建命令

创建完成, 点击保存。

15.3 运行构建

项目首页如下：

图 15.15 查看创建的项目

左侧列表是关于项目的操作。

如果需要修改刚才的配置信息，可以点击“配置”链接重新进行修改。

点击“立即构建”选项，Build History 将显示项目的构建状态。

Build History	(构建历史) -
#1 2014-7-29 22:52:38	
RSS 全部 RSS 失败	

Build History	(构建历史) -
#2 2014-7-29 23:00:15	
#1 2014-7-29 22:59:06	
RSS 全部 RSS 失败	

图 15.16 构建项目

点击构建历史时间（如上图：2014-7-29 23:00:15）



图 15.17 构建版本

点击“Console Output”查看：

控制台输出

```
Started by user anonymous
Building in workspace E:\tomcat8\webapps\Jenkins\jobs\pyse_test\workspace
[workspace] $ cmd /c call C:\Windows\TEMP\hudson6757166230655067771.bat

E:\tomcat8\webapps\Jenkins\jobs\pyse_test\workspace>d:

D:\>python test.py

D:\>exit 0
Finished: SUCCESS
```

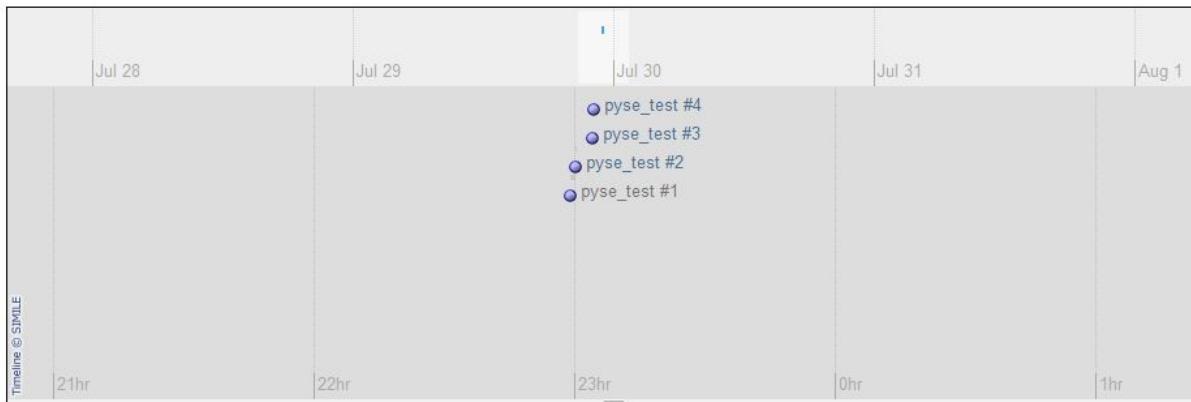
图 15.18 构建日志

跟我们在命令提示符下的操作一致，最后提示：SUCCESS，表示此次构建是成功的。

查看构建历史：

点击“返回到工程”返回到项目首页，点击“构建历史”，如果进行了多次构建，可以看到项目的构建历史图表。

时间轴



构建时间趋势

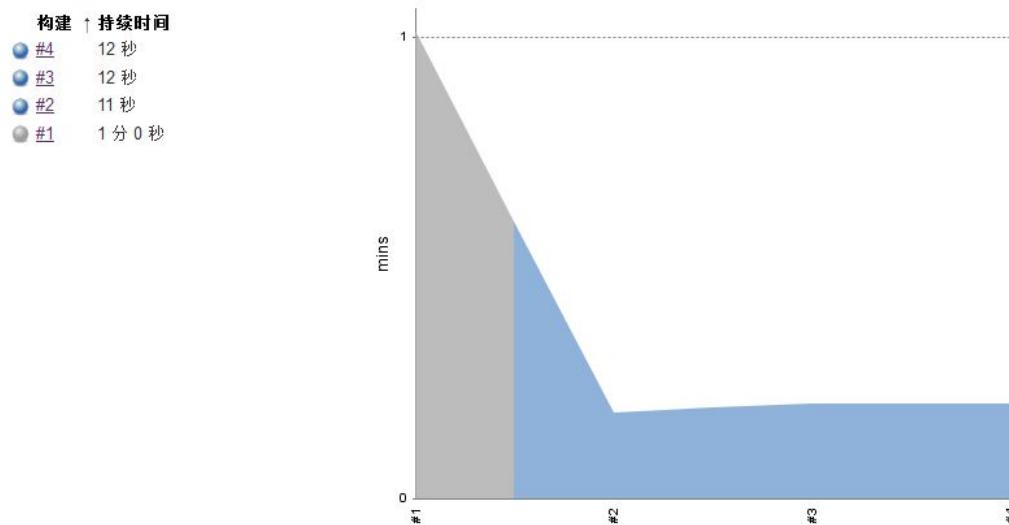


图 15.19 查看历史构建

进一步探索：

- 通过 Jenkins 实现定时任务。比操作系统的定时任务更方便灵活。
- 与 CVS 和 SVN 等版本控制工具集合，使版本工具发生更新时进行项目的构建。

附录

XPath 语法

XML 实例文档

我们将在下面的例子中使用这个 XML 文档。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bookstore>
<book>
    <title lang="eng">Harry Potter</title>
    <price>29.99</price>
</book>
<book>
    <title lang="eng">Learning XML</title>
    <price>39.95</price>
</book>
</bookstore>
```

选取节点

XPath 使用路径表达式在 XML 文档中选取节点。节点是通过沿着路径或者 **step** 来选取的。

下面列出了最有用的路径表达式：

表达式	描述
nodename	选取此节点的所有子节点。
/	从根节点选取。
//	从匹配选择的当前节点选择文档中的节点，而不考虑它们的位置。
.	选取当前节点。
..	选取当前节点的父节点。
@	选取属性。

实例

在下面的表格中，我们已列出了一些路径表达式以及表达式的结果：

路径表达式	结果
bookstore	选取 bookstore 元素的所有子节点。
/bookstore	选取根元素 bookstore。 注释：假如路径起始于正斜杠（/），则此路径始终代表到某元素的绝对路径！
bookstore/book	选取属于 bookstore 的子元素的所有 book 元素。
//book	选取所有 book 子元素，而不管它们在文档中的位置。
bookstore//book	选择属于 bookstore 元素的后代的所有 book 元素，而不管它们位于 bookstore 之下的什么位置。
//@lang	选取名为 lang 的所有属性。

谓语 (Predicates)

谓语用来查找某个特定的节点或者包含某个指定的值的节点。谓语被嵌在方括号中。

实例

在下面的表格中，我们列出了带有谓语的一些路径表达式，以及表达式的结果：

路径表达式	结果
/bookstore/book[1]	选取属于 bookstore 子元素的第一个 book 元素。
/bookstore/book[last()]	选取属于 bookstore 子元素的最后一个 book 元素。
/bookstore/book[last()-1]	选取属于 bookstore 子元素的倒数第二个 book 元素。
/bookstore/book[position()<3]	选取最前面的两个属于 bookstore 元素的子元素的 book 元素。
//title[@lang]	选取所有拥有名为 lang 的属性的 title 元素。
//title[@lang='eng']	选取所有 title 元素，且这些元素拥有值为 eng 的 lang 属性。
/bookstore/book[price>35.00]	选取 bookstore 元素的所有 book 元素，且其中的 price 元素的值须大于 35.00。
/bookstore/book[price>35.00]/title	选取 bookstore 元素中的 book 元素的所有 title 元素，且其中的 price 元素的值须大于 35.00。

选取未知节点

XPath 通配符可用来选取未知的 XML 元素。

通配符	描述
*	匹配任何元素节点。
@*	匹配任何属性节点。
node()	匹配任何类型的节点。

实例

在下面的表格中，我们列出了一些路径表达式，以及这些表达式的结果：

路径表达式	结果
/bookstore/*	选取 bookstore 元素的所有子元素。
//*[@*	选取文档中的所有元素。
//title[@*]	选取所有带有属性的 title 元素。

选取若干路径

通过在路径表达式中使用“|”运算符，您可以选取若干个路径。

实例

在下面的表格中，我们列出了一些路径表达式，以及这些表达式的结果：

路径表达式	结果
//book/title //book/price	选取 book 元素的所有 title 和 price 元素。
//title //price	选取文档中的所有 title 和 price 元素。
/bookstore/book/title //price	选取属于 bookstore 元素的 book 元素的所有 title 元素，以及文档中所有的 price 元素。

CSS 选择器参考手册

CSS3 选择器

在 CSS 中, 选择器是一种模式, 用于选择需要添加样式的元素。"CSS" 列指示该属性是在哪个 CSS 版本中定义的。(CSS1、CSS2 还是 CSS3。)

选择器	例子	例子描述
.class	.intro	选择 class="intro" 的所有元素。
#id	#firstname	选择 id="firstname" 的所有元素。
*	*	选择所有元素。
element	p	选择所有 <p> 元素。
element,element	div,p	选择所有 <div> 元素和所有 <p> 元素。
element element	div p	选择 <div> 元素内部的所有 <p> 元素。
element>element	div>p	选择父元素为 <div> 元素的所有 <p> 元素。
element+element	div+p	选择紧接在 <div> 元素之后的所有 <p> 元素。
[attribute]	[target]	选择带有 target 属性所有元素。
[attribute=value]	[target=_blank]	选择 target="_blank" 的所有元素。
[attribute~=value]	[title~=flower]	选择 title 属性包含单词 "flower" 的所有元素。
[attribute =value]	[lang =en]	选择 lang 属性值以 "en" 开头的所有元素。
:link	a:link	选择所有未被访问的链接。
:visited	a:visited	选择所有已被访问的链接。
:active	a:active	选择活动链接。
:hover	a:hover	选择鼠标指针位于其上的链接。
:focus	input:focus	选择获得焦点的 input 元素。
:first-letter	p:first-letter	选择每个 <p> 元素的首字母。
:first-line	p:first-line	选择每个 <p> 元素的首行。
:first-child	p:first-child	选择属于父元素的第一个子元素的每个 <p> 元素。
:before	p:before	在每个 <p> 元素的内容之前插入内容。
:after	p:after	在每个 <p> 元素的内容之后插入内容。
:lang(language)	p:lang(it)	选择带有以 "it" 开头的 lang 属性值的每个 <p> 元素。
element1~element2	p~ul	选择前面有 <p> 元素的每个 元素。
[attribute^=value]	a[src^="https"]	选择其 src 属性值以 "https" 开头的每个 <a> 元素。
[attribute\$=value]	a[src\$=".pdf"]	选择其 src 属性以 ".pdf" 结尾的所有 <a> 元素。
[attribute*=value]	a[src*="abc"]	选择其 src 属性中包含 "abc" 子串的每个 <a> 元素。

:first-of-type	p:first-of-type	选择属于其父元素的首个 <p> 元素的每个 <p> 元素。
:last-of-type	p:last-of-type	选择属于其父元素的最后 <p> 元素的每个 <p> 元素。
:only-of-type	p:only-of-type	选择属于其父元素唯一的 <p> 元素的每个 <p> 元素。
:only-child	p:only-child	选择属于其父元素的唯一子元素的每个 <p> 元素。
:nth-child(n)	p:nth-child(2)	选择属于其父元素的第二个子元素的每个 <p> 元素。
:nth-last-child(n)	p:nth-last-child(2)	同上，从最后一个子元素开始计数。
:nth-of-type(n)	p:nth-of-type(2)	选择属于其父元素第二个 <p> 元素的每个 <p> 元素。
:nth-last-of-type(n)	p:nth-last-of-type(2)	同上，但是从最后一个子元素开始计数。
:last-child	p:last-child	选择属于其父元素最后一个子元素每个 <p> 元素。
:root	:root	选择文档的根元素。
:empty	p:empty	选择没有子元素的每个 <p> 元素（包括文本节点）。
:target	#news:target	选择当前活动的 #news 元素。
:enabled	input:enabled	选择每个启用的 <input> 元素。
:disabled	input:disabled	选择每个禁用的 <input> 元素
:checked	input:checked	选择每个被选中的 <input> 元素。
:not(selector)	:not(p)	选择非 <p> 元素的每个元素。
::selection	::selection	选择被用户选取的元素部分。

Python 编辑器之 UliPad

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。

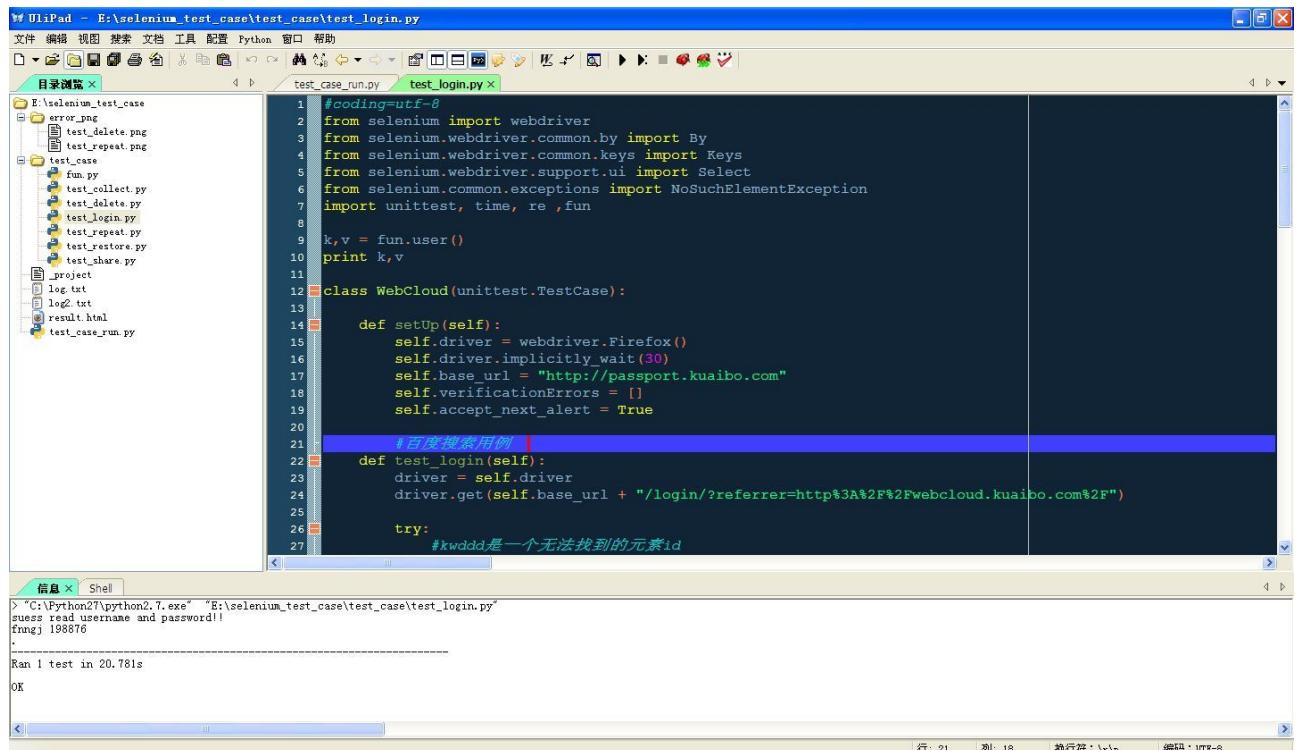
其实, Python 下面能找到一款不错的开发工具是不太容易的。Python 自带的 IDLE 写写单个小程序很好,但一个程序文件与执行信息是两个独立的窗口,程序开多了就分不清哪个程序窗口对应哪个执行窗口。

linux 会有一些非常不错的交互式 Python IDE ,如 iPython 、 bPython 等。

UliPad 是我找到的写 Python 较为舒服的一个 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 says "UliPad - E:\selenium_test_case\test_case\test_login.py". The menu bar includes "文件" (File), "编辑" (Edit), "视图" (View), "搜索" (Search), "文档" (Document), "工具" (Tools), "配置" (Configure), "Python" (Python), "窗口" (Window), and "帮助" (Help). The toolbar has icons for file operations like Open, Save, Find, and Run. The left sidebar shows a file tree with a project structure: E:\selenium_test_case, test_case, and several Python files like fun.py, test_collect.py, test_delete.py, test_login.py, test_repeat.py, test_restore.py, and test_share.py. The main code editor window displays a Python script named test_login.py:

```
#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:
            #kwddid是一个无法找到的元素id

```

Below the code editor is a terminal window titled "信自" (Terminal) with the command "C:\Python27\python2.7.exe" "E:\selenium_test_case\test_case\test_login.py" and output showing "suezz read username and password!! fnngj 198876". The status bar at the bottom indicates "行: 21 列: 18 执行符: \r\n 编码: UTF-8".

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:

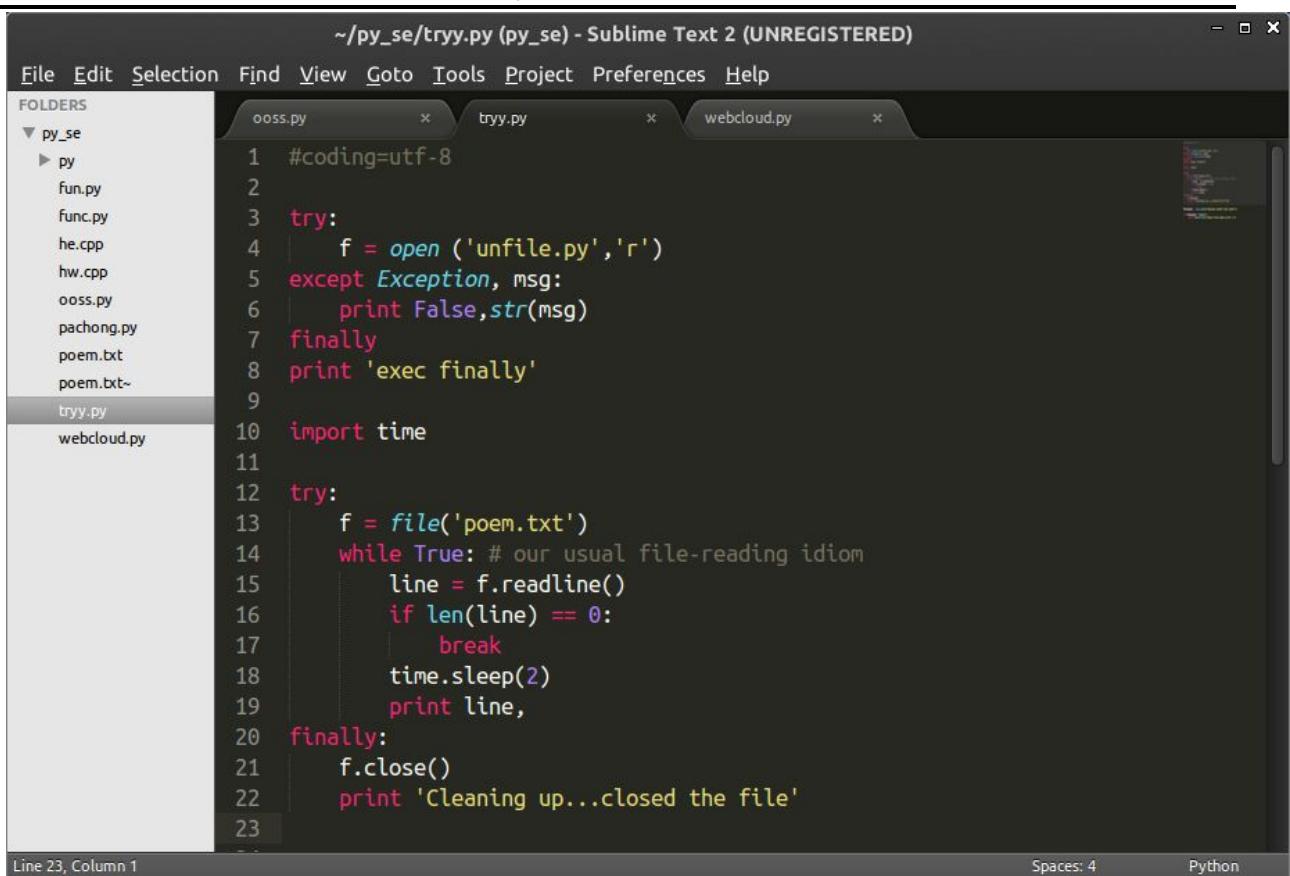
<http://www.cnblogs.com/fnng/p/3393275.html>

Python 编辑器之 Sublime

Sublime Text 是我发现的另一款好用的编辑器, 它不单单只支持 Python, 支持目前多种主流的编程语言, 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/>



```

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

~/py_se/tryy.py (py_se) - Sublime Text 2 (UNREGISTERED)

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

```

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。



(一) 在当前项目中，快速搜索文件

1. 搜索文件

```

1.sh — Downloads • 1.sh — Desktop • untitled • repairMisBLK.d
1
2
3
4 echo "start"
5
6 function f()
7 {
8     echo ok
9 }
10
11 function b()
12 {
13     echo ok

```

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

```

1.sh — Downloads • 1.sh — Desktop • untitled • repairMisBLK-default.con
1
2
3
4 echo "start"
5
6 function f()
7 {
8     echo ok
9 }
10

```

3. 搜索到了2个结果，可以按上下键来在多个结果中跳转

```

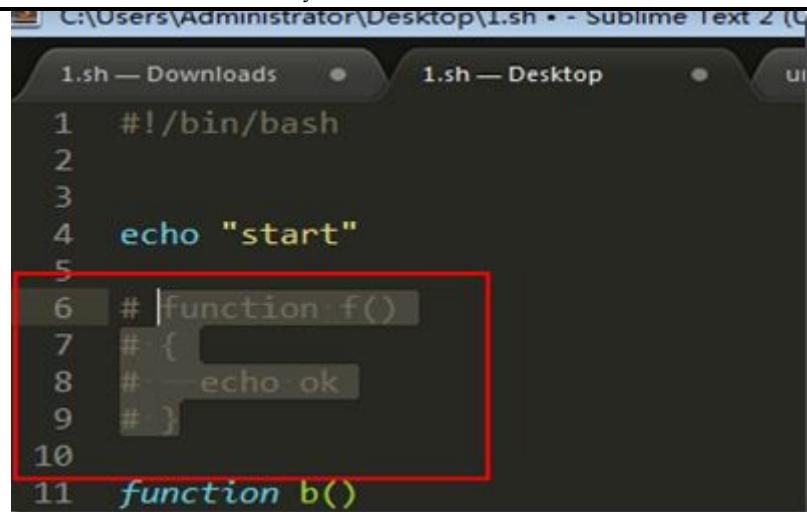
121
122
123
124 ##### get missing or corrupt block in file
125 echo -e "\n====="
126
127 ##### get missing or corrupt block in file
128 i=0
129 while :
130 do

```

(二) 添加注释

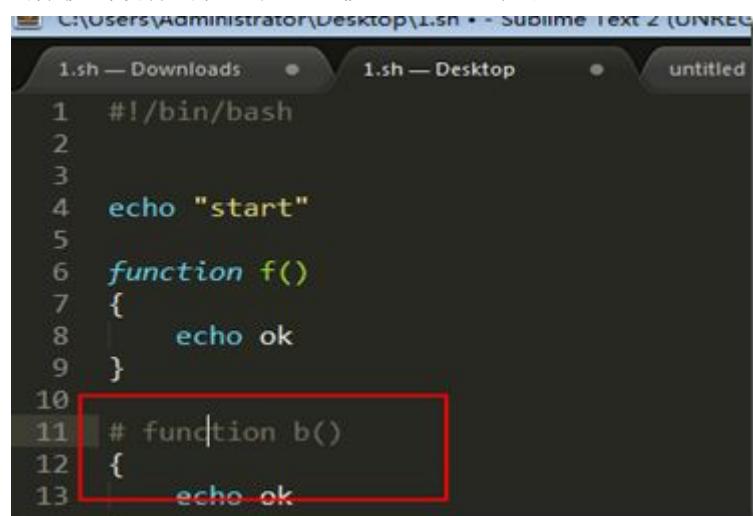
1. 添加块注释，类似于/* */用这种方法来添加的注释一样。

先选择要注释的内容，然后按 **ctrl + /**



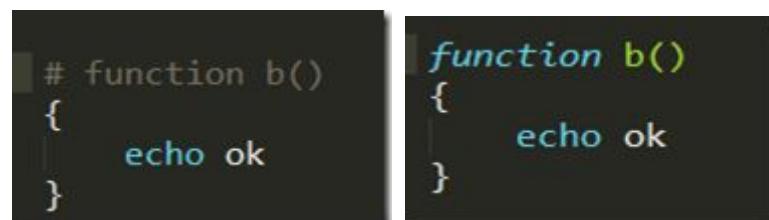
```
1. sh — Downloads • 1. sh — Desktop • ui
1  #!/bin/bash
2
3
4 echo "start"
5
6 # |Function f()
7 #
8 #   echo ok
9 #
10
11 function b()
```

2. 添加行注释，把鼠标移到改行的任意位置，按 $ctrl + /$ 即可



```
1. sh — Downloads • 1. sh — Desktop • untitled
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
```

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 $ctrl + /$ 即可

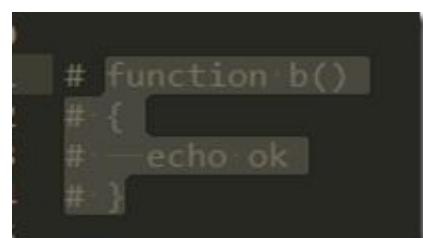


```
# function b()
{
    echo ok
}
```

```
function b()
{
    echo ok
}
```

4. 取消块注释

选择要取消的内容，按 $ctrl + /$ 即可



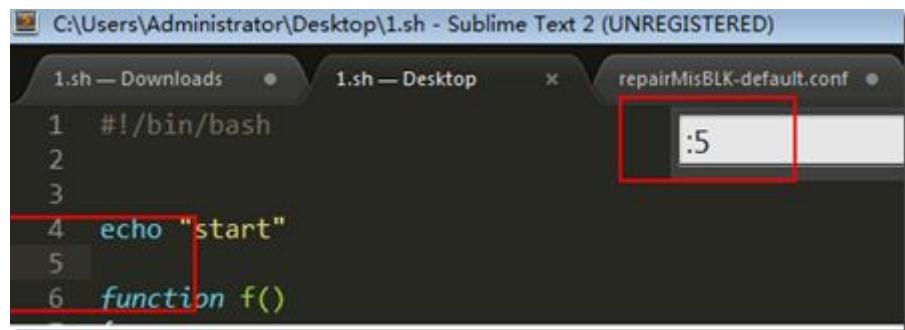
```
# function b()
#
#   echo ok
#
# }
```

```
function b()
{
    echo ok
}
```

5. 即取消注释和添加注释是逆操作

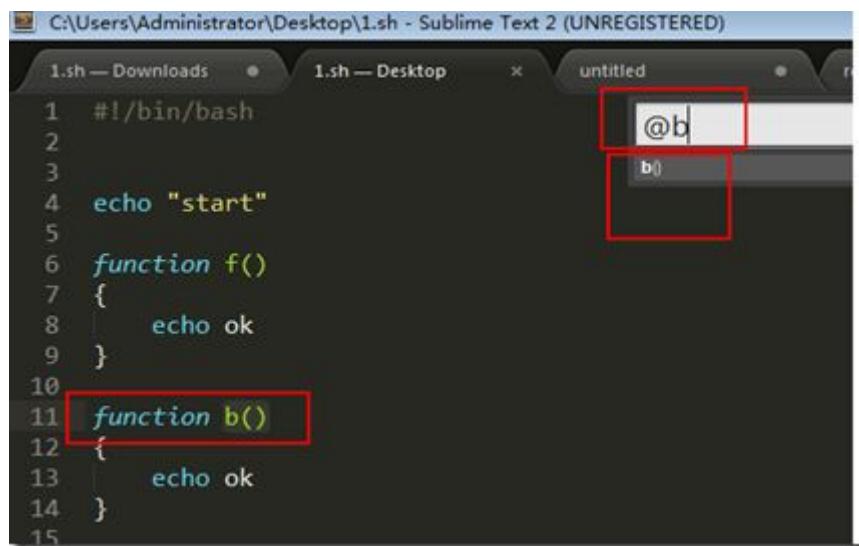
(三) 快速跳转到指定的行

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 :



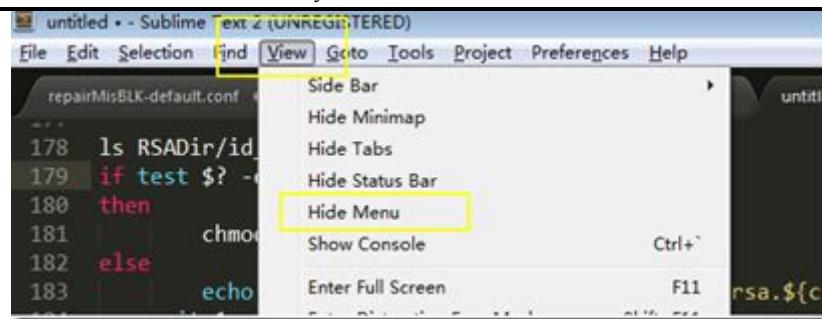
(四) 搜索函数

按 ctrl + r 或 ctrl + p ,在执行@。之后填写要搜索的函数名



(五) 隐藏菜单和显示菜单栏

1. 隐藏菜单栏: view --> Hide Menu



2. 隐藏菜单栏后，要显示菜单栏：

i. 这是隐藏之后



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



sublime 主要快捷键列表:

快捷键	说明
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

参考互联网文献：

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>

w3school：

<http://www.w3school.com.cn/css/index.asp>

<http://www.w3school.com.cn/xpath/index.asp>

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