它不仅是一本Selenium2 自动化测试书
同时还是一本实用的Python基础编程书

Broadview
www.broadview.com.cn

Selenium 2 自动化测试实战 基于Python语言

虫师 编著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Selenium 2 自动化测试实战 基于Python语言

虫师 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书共分 14 章。第 1 章是自动化测试相关基础知识的介绍；第 2 章到第 10 章是本书的重点，循序渐进地介绍了自动化测试所用到的技术；第 11 章通过一个具体的项目综合运用了前面章节所介绍的技术与技巧；第 12 章到第 14 章选取了当前最热门的技术进行了介绍，旨在扩展测试人员的综合技术能力。

本书的写作目的并不是为了简单地告诉读者如何使用一个自动化测试工具，而是希望读者在学习本书的内容后能够提高综合的技术高度与宽度，从而摆脱简单的手工测试，向高级测试工程师的道路迈进。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Selenium 2 自动化测试实战：基于 Python 语言 / 虫师编著. —北京：电子工业出版社，2016.1
ISBN 978-7-121-27853-2

Ⅰ. ①S… Ⅱ. ①虫… Ⅲ. ①软件－测试 Ⅳ.①TP311.5

中国版本图书馆 CIP 数据核字(2015)第 301408 号

责任编辑：安　娜
印　　刷：三河市鑫金马印装有限公司
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：20.25　字数：378 千字
版　　次：2016 年 1 月第 1 版
印　　次：2016 年 1 月第 1 次印刷
印　　数：3000 册　　定价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

推荐序

记得很久之前接触自动化的时候看了一本关于某早期自动化测试工具的书，书名已经记不得了，内容给我留下了深刻印象。因为那本书根本就是把官方文档有选择性地翻译一遍，对于实际应用来说其作用几乎是零。因此，从那时起我就一直认为，对工具的介绍不应该仅仅停留在理论和了解的程度，如果没有实战，那么我们之于工具无非就是叶公好龙，一直远观，无法赏玩。

很欣慰的是，虫师这本《Selenium 2 自动化测试实战——基于 Python 语言》并没有停留在夸夸其谈的表面，我仔细研究了书中的代码和示例，显然，很大一部分都是他在日常工作中勤奋总结而得出的一手范例，对技术书籍来说，这是难能可贵的。

对一本介绍自动化测试的书来说，我们其实不缺理论。国外的很多同行都在不停地传经布道，他们的结论其实很简单，自动化测试是好东西，对提升软件质量来说，自动化测试带给我们生产力的解放收益要远远大于成本。结论我们都知道，工具我们也清楚，无非就是单元测试用 xunit、rspec 之类，BDD 测试用 cucumber，UI 层面的 aut 用 Selenium，等等。这些工具我们都会用，而且也许用得还不错，不过为什么我们的自动化测试除了耗时费力，根本就看不出什么效果呢？

答案其实很简单，我们自以为用对了，但实际上也许我们才刚上路，甚至跟最佳实践背道而驰。这就像是拿 iPad 当砧板，用宰牛刀杀鸡一样，路子不对，只能越练越野，直到走火入魔。

这本书其实提到了很多不错的实践，路子正宗，从者自然不会误入歧途，这也是我推荐它的另外一个理由。

在看本书前面几章的时候，我总是不断地想起当年初出茅庐入行时所遇到的初学者困局。我很想学一门技术，但是到底应该从哪里开始？

万事开头难，没有师傅领进门，在不停的挫败和各种不同信息的正反面轰炸下，我们很容易举白旗放弃。

试想一下这样的场景：假如你是一个初学者，你从某种渠道得知 Selenium 是代表未来测试趋势的测试工具，是提升生产力的重要手段，是提升自己收入水平的一个不错的投资，于是你下定决心大干一场，准备好好地学习一下这个东西。你开始兴冲冲地去网络上搜索资料，然后你开始困惑，因为 Selenium 有两个版本：Selenium 1 和 Selenium 2，并且 Selenium 1 到现在都没有完全废弃，是学 Selenium 1 还是学 Selenium 2？Selenium 1 很经典，Selenium 2 很前沿；然后你继续深挖，你发现如果学 Selenium 2，就会遇到 Selenium 1 中一个叫 Webdriver 的东西，那是啥？然而如果要学 Selenium 1，你将一直不停地看到一个叫 Selelnium RC 的字眼，这又是什么？现在的你也许已经有一些动摇了吧，不过噩梦才刚刚开始。你也许会遇到一些靠谱的人，他们会建议你先学习一门语言，因为如果要用 Selenium，基本上就意味着你应该学会编程。但是 Selenium 支持太多的编程语言，例如，Ruby、Python、Java、Javacript、Objective-C、PHP，到底应该从哪一门语言入手，你开始陷入经典的语言之争的泥沼，你悲哀地发现自己花了很多时间去潜心研究，但最终只是模模糊糊知道了 Selenium 在远方，面前是一堆分岔路口，每种语言看起来都不错，每个分岔路口都可以到达终点，但你就是不知道该怎样迈出第一步。这便是选择的成本，选择是有风险的，选对了事半功倍，选不对只能半途而废。

对很多初学者来说，上面的困局应该都是存在的。花了很多精力和时间，但最终却发现一切都是徒劳；做了很多功课，但真正该做的事情却未曾开始。

好在这本书能够很好地解决这个困局。你不需要选择，这本书描述的就是 Selenium 2，代表了主流，也代表了未来；用的语言是 Python，全世界都在用，它简单、高效、经典、优雅。很有意思的是，Python 自身的哲学里也认为最好只用一种方法来做一件事（你可以打开 Python 解释器，然后 import this 试试），免去选择的苦恼。你应该把精力放在更有意义的事情上，比如多写几个自动化测试用例，而不是纠结于各种选择，徘徊不前。

然后便是初学者困局里更加常见的一个问题：如何搭建环境？我遇到过不少人倒在这里，而且前赴后继，无限循环。如果你有这本书，那这些问题应该都不是问题，跟着虫师描述的步骤一步步来吧，循序渐进而又节奏轻快。

搭建好环境并写好脚本之后，敢问路在何方又是初学者常见的问题。因为 Selenium 没有官方中文文档，啃英文实在不是一件愉快的事情。也许你好不容易看懂了解释，却发现官方的示例离自己身处的环境相差太多，官方的例子一直是 Google 搜索，发 Gmail，而你却悲哀地发现根本就没有 Gmail 这个网站。这本书不仅非常详细地介绍了 Selenium 的 API，而且给出了非常多可以运行的本土化的示例，这对初学者和其他使用者来说都是福音，有些例子很棒，你试过就知道。

其实试完 WebDriver 中的各种 API 后，你应该算入门了。入门了之后便是更多的困惑，比如如何去写测试用例，如何做基于数据的用例设计，如何分布式执行用例，如何多线程执行用例等，而这些答案都在本书中。

最后便是 BDD 和 CI，如果一本介绍自动化测试的书没有这两项内容，那它一定是不完整的。如果自动化是彩虹，那么 CI 便是风雨，不经历风雨怎会见彩虹。没有 CI，自动化其实找不到太多的应用场景；没有自动化，CI 更是无从说起。尽管本书的重点不是 CI，但本书的终点在 CI，就像是指出了宿命的依归，独具匠心。

还记得上次跟虫师见面时的情景，应该已经有两年多了。时过境迁，曾经我们上班的地点就在隔壁，如今发现时间逝去，很多东西都已改变。不过难能可贵的是虫师对测试技术的追求一直不曾更改，也一直笔耕不辍，这本书其实也算是水到渠成的结果。勿忘初心，坚持自己，最后希望这本书能多帮助一些人，我想这应该也是虫师的初衷吧。

乙醇

前　言

记得在 2013 年，笔者计划要学习一门脚本语言用来辅助测试工作，当时在 Ruby 与 Python 之间犹豫不定。后来开始接手社区项目的测试工作，由于社区项目基于 Python 开发，所以，就自然选择了 Python。Python 语言的简单易用与丰富的类库给我带来了很大惊喜。

后来考虑到公司的 Web 产品比较适合进行自动化测试，再加上对产品的开发进度有很好的节奏把控，于是就有了充足的时间尝试开展自动化测试。在此之前，我对 QTP 和 Selenium 两个自动化测试工具都有过接触，考虑在这两者之间选择其一。一方面是我个人更偏向于使用开源工具，另一个重要原因是 Selenium 支持多种编程语言，包括 Python。于是，就选择了 Python 与 Selenium 这样的组合进行产品自动化测试的尝试。

初期的学习遇到了不少问题。首先，Selenium 本身并不是一个单独的工具，它包含 IDE、Gird 和 WebDriver 等几个部分；其次，Selenium 与编程语言的关系，以及它在编程语言中所扮演的角色；最后，如何开发一个完整的自动化测试项目。初学者都会有这样的疑问。

当时，基于 Python 语言的 Selenium 自动化测试资料并不太多，大多资料都是基于 Java 语言的，所以学习过程也颇费周折。不过，在此过程中也得到了许多朋友的帮助，其中，乙醇的文档和 MarkRabbit 的细心指导对我的帮助很大，在此表示感谢。

从事软件测试工作不久后，我便养成了写博客的习惯，把平时的学习与积累用简单易懂的方式整理成博文，自然也会把这个技术以一个系列整理分享。后来，为了方便读者阅读，把这个系列的十几篇博文整理成了 PDF 格式，并命名为 *Selenium WebDriver* (*python*)，这应该可以看作本书的原型。再后来，不断地更新与扩充这份文档的内容，技术不再局限于 WebDriver API 的操作，于是更名为《Selenium 2 Python 自动化测试实战》，基本知识体系已经确定。与此同时，与乙醇合作的自动化相关课程也在同步开展中，使本文档中的内容具有很强的实战性，以解决具体的问题为出发点，用大量的实例来说明自动化实施的思想与概念。

本书的写作目的并不是为了简单地告诉读者如何使用一个自动化测试工具，这并非我的初衷，我希望读者在学习本书的内容后能提高综合的技术高度与宽度，从而摆脱简单的手工测试，向高级测试工程师的道路迈进。为此，我用了一定的章节来介绍 Python 的基础与应用、BDD 行为驱动、GitHub 的使用，以及持续集成工具的使用等。

本书能够出版首先需要感谢编辑安娜，她为本书的出版提供了许多意见与帮助。其次，需要感谢以往各期的学生，在传授你们技术的过程中我同样也收获颇多。另外，还要感谢王成成、符志辉、张超、刘玉凤、吴宇、许晶晶、李娜、李朋程等学生，是你们帮助我校对了这本书中的内容。

虫师

目　录

第 1 章　自动化测试基础 1
1.1　软件测试分类 1
1.2　分层的自动化测试 6
1.3　什么样的项目适合自动化测试 9
1.4　自动化测试及工具简述 10
1.5　Selenium 工具介绍 11
1.6　前端技术介绍 14
1.7　前端工具介绍 16
1.8　开发语言的选择 19

第 2 章　测试环境搭建 21
2.1　Windows 下的环境搭建 21
2.1.1　安装 Python 22
2.1.2　安装 setuptools 与 pip 24
2.1.3　安装 Selenium 26
2.1.4　ActivePython 26
2.2　Ubuntu 下的环境搭建 28
2.3　使用 IDLE 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 34
2.6　不同编程语言下使用 WebDriver 35

第 3 章　Python 基础 37
3.1　Python 哲学 37

3.2 输出与输入 39
3.2.1 print 打印 40
3.2.2 input 输入 41
3.2.3 引号与注释 41
3.3 分支与循环 43
3.3.1 if 语句 43
3.3.2 for 语句 45
3.4 数组与字典 47
3.4.1 数组 47
3.4.2 字典 47
3.5 函数、类和方法 49
3.5.1 函数 49
3.5.2 类和方法 50
3.6 模组 52
3.6.1 引用模块 52
3.6.2 模块调用 55
3.6.3 跨目录模块调用 56
3.6.4 进一步讨论跨目录模块调用 56
3.7 异常 60
3.7.1 认识异常 60
3.7.2 更多异常用法 64
3.7.3 抛出异常 66
本章小结 66

第 4 章 WebDriver API 68

4.1 从定位元素开始 68
4.1.1 id 定位 72
4.1.2 name 定位 73
4.1.3 class 定位 73
4.1.4 tag 定位 73
4.1.5 link 定位 73
4.1.6 partial link 定位 74

4.1.7 XPath 定位 75
4.1.8 CSS 定位 78
4.1.9 用 By 定位元素 82
4.2 控制浏览器 83
4.2.1 控制浏览器窗口大小 83
4.2.2 控制浏览器后退、前进 84
4.2.3 模拟浏览器刷新 85
4.3 简单元素操作 86
4.3.1 126 邮箱登录 86
4.3.2 WebElement 接口常用方法 87
4.4 鼠标事件 89
4.5 键盘事件 92
4.6 获得验证信息 93
4.7 设置元素等待 95
4.7.1 显式等待 95
4.7.2 隐式等待 98
4.7.3 sleep 休眠方法 99
4.8 定位一组元素 100
4.9 多表单切换 104
4.10 多窗口切换 106
4.11 警告框处理 108
4.12 上传文件 110
4.12.1 send_keys 实现上传 110
4.12.2 AutoIt 实现上传 111
4.13 下载文件 116
4.14 操作 Cookie 117
4.15 调用 JavaScript 120
4.16 处理 HTML5 的视频播放 123
4.17 窗口截图 124
4.18 关闭窗口 125
4.19 验证码的处理 125
4.20 WebDriver 原理 128
本章小结 130

第 5 章　自动化测试模型 132
5.1　自动化测试模型介绍 133
5.1.1　线性测试 133
5.1.2　模块化驱动测试 134
5.1.3　数据驱动测试 134
5.1.4　关键字驱动测试 136
5.2　模块化驱动测试实例 138
5.3　数据驱动测试实例 140
5.3.1　参数化邮箱登录 141
5.3.2　参数化搜索关键字 142
5.3.3　读取 txt 文件 143
5.3.4　读取 csv 文件 144
5.3.5　读取 xml 文件 146
本章小结 150

第 6 章　Selenium IDE 151
6.1　Selenium IDE 安装 151
6.1.1　在线安装 151
6.1.2　下载插件安装 153
6.2　Selenium IDE 界面介绍 154
6.3　创建测试用例 155
6.3.1　录制脚本 155
6.3.2　编辑脚本 156
6.4　Selenium IDE 命令 159
6.5　断言与验证 162
6.5.1　断言 164
6.5.2　验证 165
6.6　等待与变量 167
6.6.1　等待 167
6.6.2　变量 167
本章小结 169

第 7 章 unittest 单元测试框架 170

7.1 认识 unittest 171

7.1.1 认识单元测试 171

7.1.2 重要的概念 174

7.1.3 断言方法 176

7.1.4 组织单元测试用例 181

7.1.5 discover 更多测试用例 184

7.2 关于 unittest 还需要知道的 187

7.2.1 用例执行的顺序 188

7.2.2 执行多级目录的用例 190

7.2.3 跳过测试和预期失败 191

7.2.4 fixtures 193

7.3 带 unittest 的脚本分析 195

7.4 编写 Web 测试用例 200

本章小结 203

第 8 章 自动化测试高级应用 204

8.1 HTML 测试报告 204

8.1.1 修改 HTMLTestRunner 205

8.1.2 生成 HTML 测试报告 206

8.1.3 更易读的测试报告 208

8.1.4 测试报告文件名 209

8.1.5 项目集成测试报告 212

8.2 自动发邮件功能 213

8.2.1 发送 HTML 格式的邮件 216

8.2.2 发送带附件的邮件 217

8.2.3 查找最新的测试报告 218

8.2.4 整合自动发邮件功能 219

8.3 Page Object 设计模式 221

8.3.1 认识 Page Object 221

8.3.2 Paget Object 实例 223

本章小结 228

第 9 章　Selenium Grid2 229
9.1　Selenium Server 环境配置 229
9.2　Selenium Grid 工作原理 233
9.3　Remote 应用 236
9.3.1　WebDriver 驱动分析 237
9.3.2　Remote 实例 243
9.3.3　参数化平台及浏览器 243
9.4　WebDriver 驱动 246
9.4.1　Edge 浏览器 249
9.4.2　Opera 浏览器 250
9.4.3　Safari 浏览器 251
9.4.4　HtmlUnit 模式 251
9.4.5　PhantomJS 模式 252
本章小结 254

第 10 章　Python 多线程 255
10.1　单线程的时代 256
10.2　多线程技术 258
10.2.1　threading 模块 258
10.2.2　优化线程的创建 260
10.2.3　创建线程类 262
10.3　多进程技术 263
10.3.1　multiprocessing 模块 263
10.3.2　Pipe 和 Queue 266
10.4　应用于自动化测试 268
10.4.1　多线程执行测试用例 269
10.4.2　多线程分布式执行测试用例 270
本章小结 272

第 11 章　自动化测试项目实战 273
11.1　自动化测试用例设计 273
11.1.1　手工测试用例与自动化测试用例 273

11.1.2 测试类型 275
11.1.3 自动化测试用例编写原则 276
11.2 BBS 社区项目实战 277
11.2.1 准备工作 277
11.2.2 项目结构介绍 278
11.2.3 编写公共模块 281
11.2.4 编写 Page Object 283
11.2.5 编写测试用例 286
11.2.6 执行测试用例 289
本章小结 291

第 12 章 BDD 框架之 Lettuce 入门 292

12.1 什么是 BDD 292
12.2 安装 Lettuce 294
12.3 阶乘的例子 295
12.3.1 什么是阶乘 295
12.3.2 编写 BDD 实现 296
12.3.3 添加测试场景 300
12.3.4 Lettuce 目录结构与执行过程 303
12.4 Lettuce_webdriver 自动化测试 303
本章小结 308

（第 13 章和第 14 章内容可到博文视点网站免费下载）

第 13 章 GitHub 托管项目 309

13.1 注册与安装 310
13.1.1 注册 GitHub 310
13.1.2 安装 Git 310
13.1.3 建立连接 311
13.2 Git/GitHub 基本使用 314
13.2.1 GitHub 创建项目 314
13.2.2 本地创建项目 315
13.2.3 克隆项目 318

13.2.4　更新项目 319
本章小结 323

第 14 章　持续集成 Jenkins 入门 324
14.1　环境搭建 325
14.2　创建任务 328
14.3　运行构建 333
14.4　定时执行构建 335
本章小结 338

第1章
自动化测试基础

在正式开始本书的学习之前，我们有必要先来了解什么是软件测试，以及软件自动化测试相关的概念和工具，这将有助于对本书后面内容的学习。

1.1 软件测试分类

软件测试领域名词颇多，许多测试新手容易混淆概念。因为从不同的角度对软件测试有不同的分类方法，所以，这里汇总常见软件测试的相关名词，让读者对软件测试领域有个概括性了解。

1. 根据项目流程阶段划分软件测试

图 1.1 是一个典型的“V”模型软件开发流程，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。

1）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。

2）集成测试：集成测试是在单元测试的基础上，先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。

3）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需求规格的要求。

4）验收测试：验收测试是部署软件之前的最后一个测试阶段。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能够满足用户的需求。

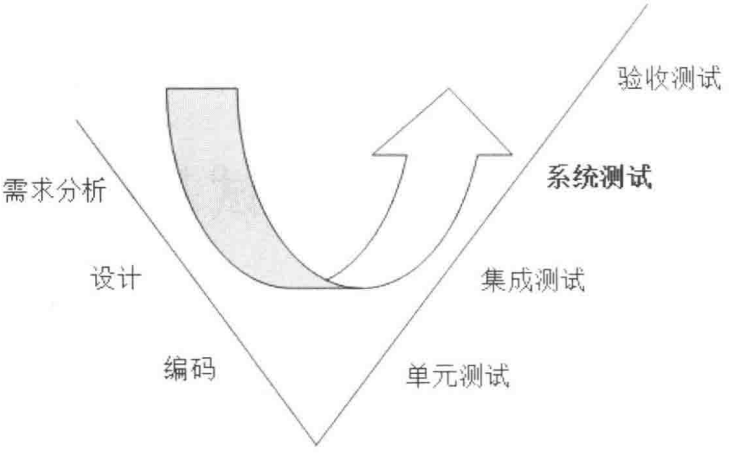

图 1.1　项目流程与对应的测试

2. 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之一，如图 1.2 所示。

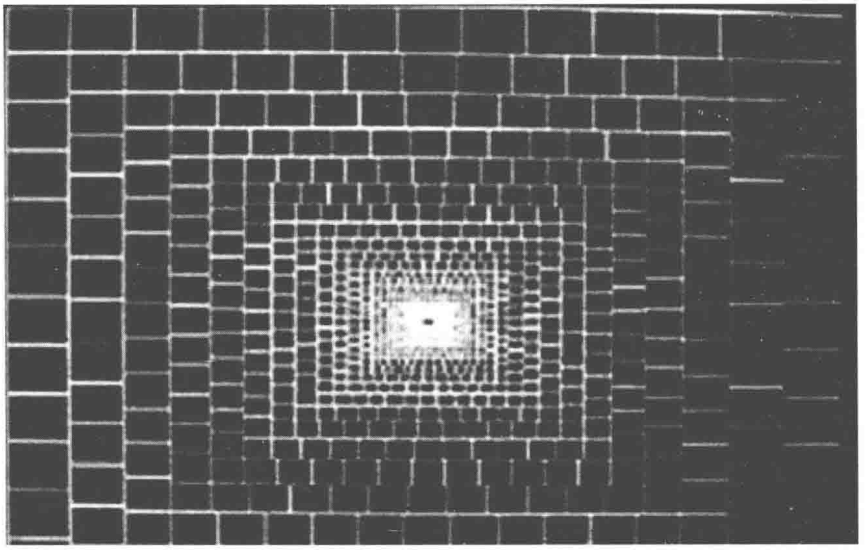

图 1.2　黑盒测试与白盒测试

1）黑盒测试

黑盒测试，指的是把被测的软件看作一个黑盒子，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、程序是否能接收输入数据并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。

2）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，检验程序中的每条逻辑路径是否都能按预定要求正确工作。

3）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，灰盒测试既关注输出对于输入的正确性，同时也关注内部表现。但这种关注不像白盒测试那样详细、完整，它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，因此需要采取灰盒测试的方法。

3. 功能测试与性能测试

从软件的不同测试面可以划分为功能测试与性能测试。

1）功能测试

功能测试主要检查实际功能是否符合用户的需求，因此测试的大部分工作也是围绕软件的功能进行。设计软件的目的就是满足用户对其功能的需求，如果偏离了这个目的，则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。

2）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的测试。

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。例如一个登录所需要的时间，一个商品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，它需要在搭建好的一个具体且独立的测试环境下进行。
- 空间性能：主要指软件运行时所消耗的系统资源，例如硬件资源，CPU、内存、网络带宽消耗等。

4. 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。

1）手工测试

手工测试就是由测试人员一个一个地去执行测试用例，通过键盘鼠标等输入一些参数，并查看返回结果是否符合预期结果。

手工测试并非专业术语，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，为了更明显地与自动化测试进行区分，这里使用了手工测试这种说法。

2）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则流程一步步执行测试，把得到的实际结果与期望结果进行比较。在此过程中，为了节省人力、时间和硬件资源，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。

- 功能自动化测试：是把以人为驱动的测试行为转化为机器执行的一种过程。通过测试工具（或框架）录制/编写测试脚本，对软件的功能进行测试，并验证测试结果是否正确，从而代替部分的手工测试工作，达到节约人力成本和时间成本的目的。
- 性能自动化测试：通过性能工具来模拟成千上万的虚拟用户向系统发送请求，从

而验证系统的处理能力。

5. 冒烟测试、回归测试、随机测试、探索性测试和安全测试

这几种测试出现在软件测试的周期中，既不算具体明确的测试阶段，也不是具体的测试方法。

1）冒烟测试

是指在对一个新版本进行大规模的系统测试之前，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式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运行通过，则打回开发组重新开发。这样做的好处是可以节省时间和人力投入到不可测的项目中。

2）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第二轮软件测试时开始的，验证第一轮软件测试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，直到所有问题回归通过为止。

3）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，例如测试不系统、无法统计代码覆盖率和需求覆盖率、发现的问题难以重现等。一般是放在测试的最后执行。随机测试更专业的升级版叫做探索性测试。

4）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。探索性测试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。

5）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至发布阶段，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。

安全测试现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，尤其是互联网产品，最容易遭受各种安全攻击。

1.2 分层的自动化测试

测试金字塔的概念由敏捷大师 Mike Cohn 在他的 *Succeeding with Agile* 一书中首次提出，如图 1.3 所示。他的基本观点是：我们应该有更多的低级别的单元测试，而不仅仅是通过用户界面运行的高层的端到端的测试。

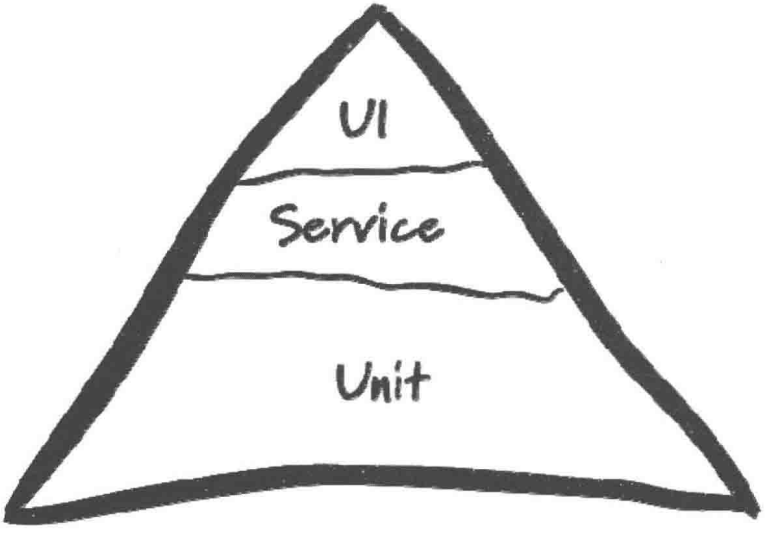

图 1.3 测试金字塔

Martin Fowler 大师在测试金字塔模型的基础上提出分层自动化测试的概念。在自动化测试之前加了一个“分层”的修饰，以区别于“传统的”自动化测试。那么什么是传统的自动化测试？为何要提倡分层自动化测试的思想呢？

所谓传统的自动化测试我们可以理解为基于产品 UI 层的自动化测试，它是将黑盒功能测试转化为由程序或工具执行的一种自动化测试。

在目前的大多数研发组织当中，都存在开发与测试团队割裂（部门墙）、质量职责错配（测试主要对质量负责）的问题，在这种状态下，测试团队的一个“正常”反应就是试图在测试团队能够掌控的黑盒测试环节进行尽可能全面的覆盖，甚至是尽可能全面的 UI 自动化测试。

这可能会导致两个恶果：一是测试团队规模的急剧膨胀；二是所谓的全面 UI 自动化测试运动。因为 UI 是非常易变的，所以 UI 自动化测试维护成本相对高昂。

分层自动化测试倡导的是从黑盒（UI）单层到黑白盒多层的自动化测试体系，从全面黑盒自动化测试到对系统的不同层次进行自动化测试，如图 1.4 所示。

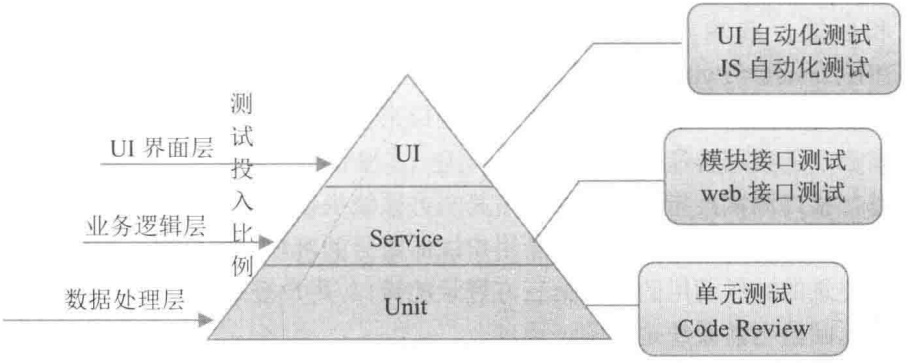

图 1.4　分层自动化测试

1. 单元自动化测试

单元自动化测试是指对软件中的最小可测试单元进行检查和验证。对于单元测试中单元的含义，一般来说，要根据实际情况去判定其具体含义，如 C 语言中单元是指一个函数，Java 中单元是指一个类，图形化的软件中单元是指一个窗口或一个菜单等。总的来说，单元就是人为规定的最小的被测功能模块。规范的进行单元测试需要借助单元测试框架，如 Java 语言的 Junit、TestNG，C#语言的 NUnit，以及 Python 语言的 unittest、pytest 等，目前几乎所有的主流语言都有其相应的单元测试框架。

Code Review 中文翻译为代码评审或代码审查，是指在软件开发过程中，通过对源代码进行系统性检查的过程。通常的目的是查找系统缺陷、保证软件总体质量以及提高开发者自身水平。与 Code Review 相关的插件和工具有很多，例如 Java 语言中基于 Eclipse 的 ReviewClipse 和 Jupiter、主要针对 Python 语言的 Review Board 等。

2. 接口自动化测试

根据笔者的理解，Web 应用的接口自动化测试大体分为两类：模块接口测试和 Web 接口测试。

1）模块接口测试，主要测试模块之间的调用与返回。当然，我们也可以将其看作是单元测试的基础。它主要强调对一个类方法或函数的调用，并对返回结果的验证，所用到的测试工具与单元自动化测试相同。

2）Web 接口测试又可分为两类：服务器接口测试和外部接口测试。

- 服务器接口测试：指测试浏览器与服务器的接口。我们知道 Web 开发一般分前端和后端，前端开发人员用 HTML/CSS/JavaScript 等技术，后端开发人员用 PHP/Java/C#/Python/Ruby 等各种语言。用户的操作是在前端页面上，需要后端提供服务器接口，前端通过调用这些接口来获得需要的数据，通过 HTTP 协议实现前后端的数据传递。
- 外部接口测试：指调用的接口由第三方系统提供。典型的例子就是第三方登录，例如新上线的产品为了免于新用户注册账号的麻烦会提供第三方登录，那么用户在登录的时候调用的就是第三方登录的接口，用户登录信息的验证由第三方完成，并返回给当前系统是否验证通过。

当然，接口测试也有相应的类库或工具，例如测试 HTTP 的有 HttpUnit、Postman 等。

3. UI 自动化测试

UI 层是用户使用该产品的入口，所有功能都通过这一层提供并展示给用户，所以测试工作大多集中在这一层进行。为了减轻这一层的测试人力和时间成本，早期的自动化测试工具主要针对该层设计。目前主流的测试工具有 UFT、Watir、Robot Framework、Selenium 等。

除 UI 层所展示的功能外，前端代码同样需要进行测试。在前端开发中最主要的莫过于 JavaScript 脚本语言，而 QUnit 就是针对 JavaScript 的一个强大的单元测试框架。

图 1.4 中的测试金字塔映射了不同测试阶段所投入的自动化测试的比例，UI 层被放到了塔尖，这也说明 UI 层应该投入较少的自动化测试。如果系统只关注 UI 层的自动化测试并不是一种明智的做法，因为其很难从本质上保证产品的质量。如果妄图实现全面的 UI 层的自动化测试，那么需要投入大量的人力和时间，然而，最终获得的收益可能远低于所投入的成本。因为对于一个系统来讲，越接近用户其越容易变化，为了适应这种变化就必

须要投入更多的成本。

既然 UI 层的自动化测试这么劳民伤财，那么我们是不是只做单元测试与接口测试就可以了呢？答案是否定的，因为不管什么样的产品，最终呈现给用户的都是 UI 层的功能，所以产品才需要招聘大量的测试人员进行 UI 层的功能测试。也正是因为测试人员在 UI 层投入了大量的时间与精力，所以我们才有必要通过自动化的方式帮助测试人员解放部分重复的工作。所以，笔者更提倡“半自动化”的开展测试工作，把可以自动化测试的工作交给工具或脚本完成，这样测试人员就可以把更多的精力放在更重要的测试工作上，例如探索性测试等。

至于在金字塔中每一层测试的投入比例则要根据实际的产品特征来划分。在《Google 测试之道》一书中提到，Google 对产品测试类型划分为：小测试、中测试和大测试，采用 70%（小）/20%（中）/10%（大）的比例，大体对应测试金字塔中的 Unit、Service 和 UI 层。

在进行自动化测试中最担心的是变化，因为变化会直接导致测试用例的运行失败，所以需要对自动化脚本进行不断调整。如何控制失败、降低维护成本是对自动化测试工具及人员能力的挑战。反过来讲，一份永远都运行通过的自动化测试用例已经失去了它存在的价值。

1.3　什么样的项目适合自动化测试

相信在你拿到这本书时已经对要进行自动化的项目做了一些分析和考量，但在这里我们还是有必要说明一下什么样的项目适合尝试进行自动化测试，以免读者在不太适合自动化测试的项目中痛苦挣扎，既浪费了大量的人力和时间，又收效甚微。

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）被测软件系统开发较为规范，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自动化测试工作，因此需要读者做出权衡。在我们普遍的自动化测试经验中，一般满足以下三个条件就可以对项目开展自动化测试。

1）软件需求变动不频繁

自动化测试脚本变化的大小与频率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，那么测试人员就需要根据变动的需求来不断地更新自动化测试用例，从而适应新的功能。而脚本的维护本身就是一个开发代码的过程，需要扩展、修改、调试，有时还需要对架构做出调整。如果所花费的维护成本高于利用其节省的测试成本，那么自动化测试就失去了它的价值与意义。

一种折中的做法是先对系统中相对稳定的模块与功能进行自动化测试，而变动较大的部分用手工进行测试。

2）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，而这个过程本身就是一个软件的开发过程，如果项目的周期较短，没有足够的时间去支持这样一个过程的话，那么就不需要进行自动化测试了。

3）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：一是所测试的项目之间是否存有很大的差异性（如 C/S 系统架构与 B/S 系统架构的差异）；二是所选择的测试技术和工具是否适应这种差异；三是测试人员是否有能力设计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：广义上来讲，所有借助工具来辅助进行软件测

试的方式都可以称为自动化测试；狭义上来讲，主要指基于 UI 层的功能自动化测试。

注意：如果没有特别说明，则本文所说的“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。

1）UFT

UFT（Unified Functional Testing）由 QTP（Quick Test Professional software）与 ST（Service Test）合并而来，由 HP 公司开发。它是一个企业级的自动测试工具，提供了强大易用的录制回放功能，同时兼容对象识别模式与图像识别模式两种识别方式，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。

2）Robot Framework

Robot Framework 是一款基于 Python 语言编写的自动化测试框架，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。

3）Watir

Watir（Web Application Testing in Ruby）是一个基于 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。

4）Selenium

Selenium 也是一个用于 Web 应用程序测试的工具，支持多平台、多浏览器、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，除上面所列的自动化测试工具外，根据不同的应用还有很多商业的或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

1. 什么是 Selenium?

Selenium 主要用于 Web 应用程序的自动化测试，但并不局限于此，它还支持所有基于 Web 的管理任务自动化。

Selenium 的特点如下：

- 开源，免费；
- 多浏览器支持：Firefox、Chrome、IE、Opera、Edge；
- 多平台支持：Linux、Windows、MAC；
- 多语言支持：Java、Python、Ruby、C#、JavaScript、C++；
- 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 不是由单独一个工具构成的，而是由一些插件、类库组成，每个部分都有其特点和应用场景，Selenium 1.0 家谱，如图 1.5 所示。

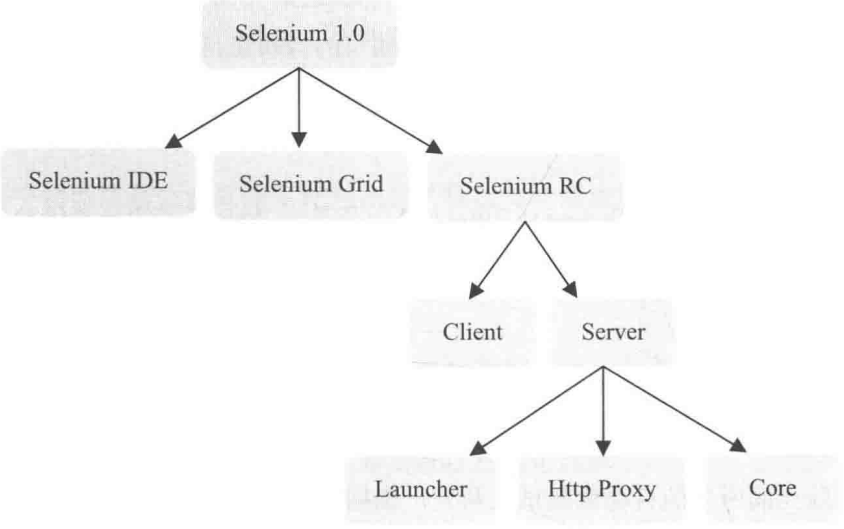

图 1.5 Selenium 1.0 家谱

2. 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？官方给出了它自身作用的定位：

快速地创建 bug 重现脚本，在测试人员测试过程中，发现 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地重现 bug。

IDE 录制的脚本可以转换成多种语言，从而帮助我们快速地开发脚本。关于这个功能在后面的章节中我们会着重介绍。

3. 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 可以很方便地实现在多台机器上和异构环境中运行测试用例。

4. Selenium RC

Selenium RC（Remote Control）是 Selenium 家族的核心部分。Selenium RC 支持多种不同语言编写的自动化测试脚本，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 分为 Client Libraries 和 Selenium Server。Client Libraries 库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 主要包括三个部分：Launcher、Http Proxy 和 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 JavaScript 函数的集合，即通过这些 JavaScript 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。

5. Selenium 2.0

搞清了 Selenium 1.0 的家族关系，再来看看 Selenium 2.0。Selenium 2.0 就是把 WebDriver 加入到了这个家族中，简单用公式表示为：

Selenium 2.0 = Selenium 1.0 + WebDriver

需要强调的是，在 Selenium 2.0 中主推的是 WebDriver，可以将其看作 Selenium RC 的替代品。因为 Selenium 为了保持向下的兼容性，所以在 Selenium 2.0 中并没有彻底地抛弃 Selenium RC。如果是初次使用 Selenium 开发一个新的自动化测试项目，那么可以直接使用 WebDriver。Selenium RC 与 WebDriver 有什么区别呢？

Selenium RC 是在浏览器中运行 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 是通过原生浏览器支持或者浏览器扩展来直接控制浏览器。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，与浏览器紧密集成，因此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之外，WebDriver 还利用操作系统级的调用，模拟用户输入。

Selenium 与 WebDriver 原是属于两个不同的项目，WebDriver 的创建者 Simon Stewart 早在 2009 年 8 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（例如能够绕过 JavaScript 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 是基于 Web 的自动化测试技术，而我们要操作的对象是 Web 页面，所以有必要对前端 Web 的技术与工具作一个简单介绍。

1. HTML 简介

HTML（Hyper Text Markup Language）中文为超文本标记语言，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 HTML 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

```
<html>
    <head>
        <title>标题</title>
    </head>
    <body>
        <h1>正文</h1>
    </body>
```

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

<html> 与 </html> 之间的文本用于描述网页。

<head> 与</head>之间的文本用于定义文档的头部，它是所有头部元素的容器。

<title> 与</title> 之间的文本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 表示为一号字体。

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

当然，HTML 还定义了许多其他标签，读者可以登录 w3school 网站学习。

2. 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，则需要添加<script>标签，并通过 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()可以向文档输出内容。

3. 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 文件。

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，才能传送、接收和显示出这个文档。

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

上面只是简单介绍了 HTML、JavaScript 以及 XML 等前端技术，了解这些技术将有助于我们顺利地进行 Web 自动化测试工作。

1.7 前端工具介绍

1. FireBug

FireBug 是 Firefox 浏览器下的一套开发类插件，相信很多读者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，如图 1.6 所示。

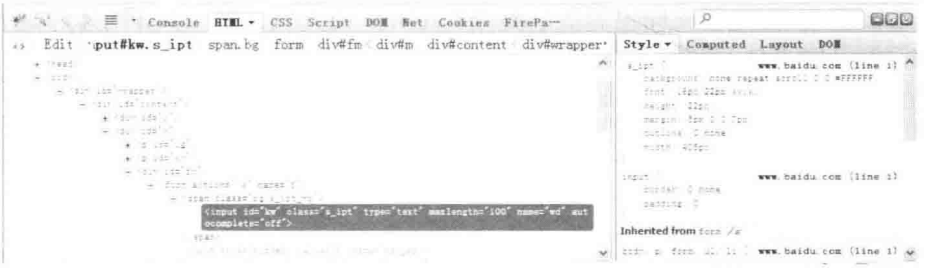

图 1.6　FireBug

我们可以通过它方便地查看页面上的元素，从而根据其属性进行定位。在 Web 自动化测试脚本的编写过程中，此工具起着至关重要的作用。

FireBug 安装方式：首先在 Firfox 浏览器的菜单栏中单击 tools（工具）→add-ons Manager（添加组件），搜索 FireBug。然后对搜索到的插件进行安装，安装完成后重启浏览器，即可在工具栏中看到 FireBug 的按钮。

2. FirePath

FirePath 是 FireBug 插件扩展的一个工具，用来编辑、检查和生成的 XPath1.0 表达式、CSS 3 选择器以及 jQuery 选择器。可以帮助我们通过 XPath 和 CSS 来快速定位页面上的元素，如图 1.7 所示。

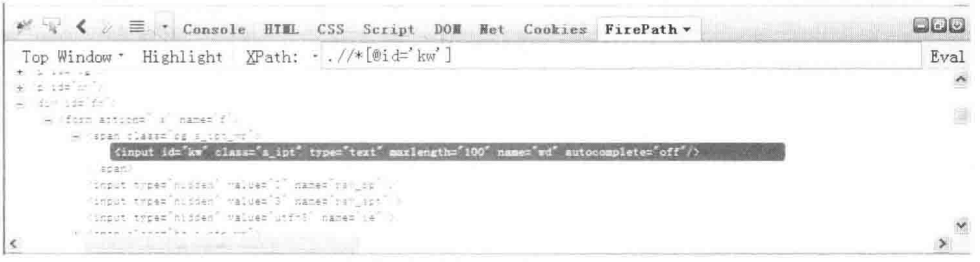

图 1.7　FirePath

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，快速地帮助我们定位元素。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。

3. Chrome 开发人员工具与 IE 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 开发人员工具，单击 Chrome 浏览器右上角的菜单按钮，在下拉菜单中选择“工具”→“开发人员工具”即可打开，还可以通过快捷键 Ctrl+Shift+I 或 F12 打开，如图 1.8 所示。

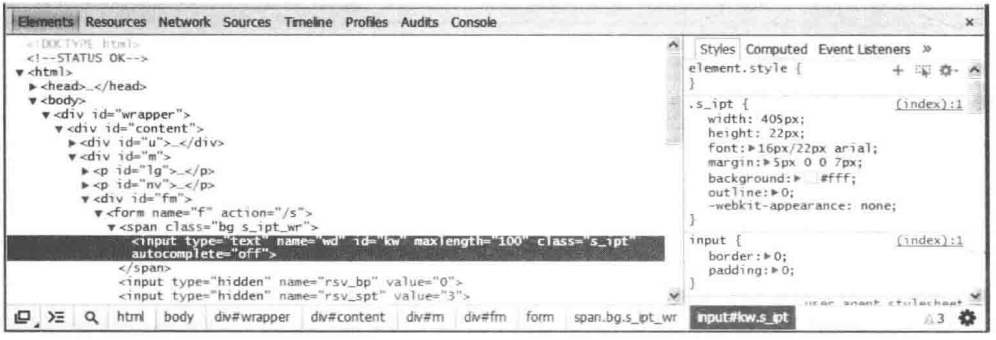

图 1.8 Chrome 开发人员工具

IE 浏览器从 IE8 版本开始加入了开发人员工具，用起来也非常方便。单击菜单栏“工具”→“F12 开发人员工具”或者通过快捷键 F12 即可打开，如图 1.9 所示。值得一提的是，它提供了浏览器的兼容模式，我们可以通过选择浏览器模式切换到不同的 IE 版本，这将非常方便地帮助我们测试 IE 浏览器的兼容性。

图 1.9 IE 开发人员工具

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，如 Java、Python、Ruby、PHP、C#、JavaScript 等，那么我们应该选择哪一种语言结合 Selenium WebDriver 进行自动化开发呢？这里笔者给出一点自己的看法。

有人说我们公司的软件是用某语言开发的，所以自动化测试也要选某语言；其实软件开发语言和软件自动化测试语言没有必然联系。也就是说，基于 Python（+ Selenium）编写的自动化测试脚本既可以测试基于 Java 开发的 Web 项目，也可以测试基于 PHP 开发的 Web 项目。所以，在选择 Selenium 自动化测试语言时不需要考虑与开发语言的一致性。

选择与开发相同的语言当然有有利的一面，测试人员通过编写自动化测试实践，既提高了自己的编码能力，也有助于其他开发测试工作的进行。例如，协助开发人员定位代码级的 bug，协助开发人员进行单元测试等，而这些工作就需要熟悉开发人员所使用的语言了。

本书并没有像市面上已经出版的几本 Selenium 书一样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的考虑。

对编程能力较弱的初学者来说，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，读者不仅可以掌握自动化测试技术，还能掌握一门语法简单且功能强大的编程语言。那为什么选 Python 而不选 Ruby 呢？这其中存在一定的个人偏好，因为 Ruby 是一个“魔法”语言，时常会给你带来很多惊喜，而 Python 的宗旨是使处理问题变得更简单，它们之间存在不同的设计哲学。当然，不管选择哪一种编程语言，都会有个人的需求驱动与偏好在里面。

Python 语言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、科学计算以及游戏开发等多个领域都应用得非常广泛，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为广阔的大门。

对有编程经验的读者来说，学习 Python 语言的成本很低，你完全可以在很短的时间内学习和使用 Python 来处理问题。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，对于同样想学习 Selenium 自动化测试技术，而不愿意尝试使用 Python 语言的读者来说，本书的例子虽然基于 Python 语言，但更多的是提供处理问题的思路与方法，所以，同样可以把本书作为参考资料。

虽然本书中涉及 Python 语言的地方都会进行单独讲解，但为了初学者能系统全面地使用 Python 语言，笔者建议准备好一本 Python 基础教程放在身边，以便有疑问的地方随时翻阅。

第 2 章
测试环境搭建

也许你已经迫不及待地坐在了电脑前面，想要开始自动化测试之旅。不要着急，在此之前，我们需要先搭建好测试所需的开发环境。如果你选择使用 Python 和 Selenium 来从事自动化测试工作，从这本书开始，没错的！

2.1 Windows 下的环境搭建

如果想要学习一门编程语言，对于新手来说只需到其官方网站上去下载最新版本安装即可，但对于想要学习 Python 的新手来说，将会面临一个版本选择的问题。因为 Python 同时存在两个版本（Python 2 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，是因为早期的 Python 版本在基础方面设计存在着一些不足之处，Python 3 在设计的时候很好地解决了这些遗留问题，并且在性能上也有了一定的提升，但同时带来的新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。

就目前情况来看，两个版本的更新与维护都在继续。并且 Python 2 的开发者依然过半。在笔者看来读者选择哪一个版本进行入门学习都可以，对有丰富经验的 Pythoner 来说，选择使用哪个版本取决于他们当前要使用的库、框架是否支持该版本，当然，对于新手来说选择 Python 3 的最大好处就是可以很大程度地避免编码问题。

在该书出版时，笔者纠结该选择 Python 2 还是 Python 3，因为本书中涉及的部分库目前还不支持 Python 3，例如，第 8.1 节的 HTMLTestRunner，第 12 章中的 Lettuce 等。站在未来的角度，决定将所有代码基于 Python 3 实现并说明与 Python 2 的不同之处，这样读者不管选用哪一版本都可以按照本书的内容进行学习。

2.1.1 安装 Python

访问 Python 官方网站：https://www.Python.org/。

找到下载页面下载最新版本的 Python 3，截至作者发稿，最新版本为 Python 3.5。读者可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，如果是 32 位系统则选择 x86 版本；如果是 64 位系统，则选择 64 版本。下载完成后会得到一个以.msi 为后缀名的文件，双击进行安装，如图 2.1 所示。

图 2.1　Python 安装界面

安装过程与一般的 Windows 程序类似。安装完成后，可在开始菜单中看到安装好的 Python 目录，如图 2.2 所示。

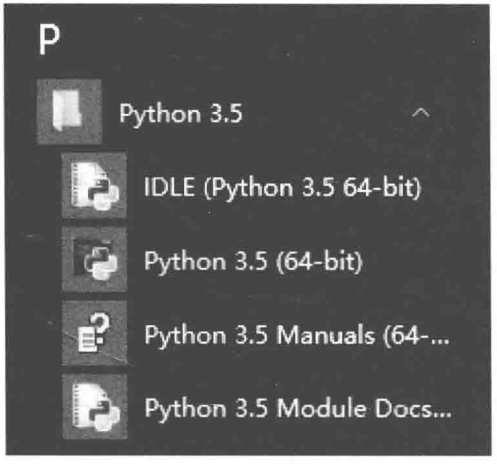

图 2.2　Python 目录

打开 Python 自带的 IDLE，就可以编写 Python 程序了，Python Shell 界面如图 2.3 所示。

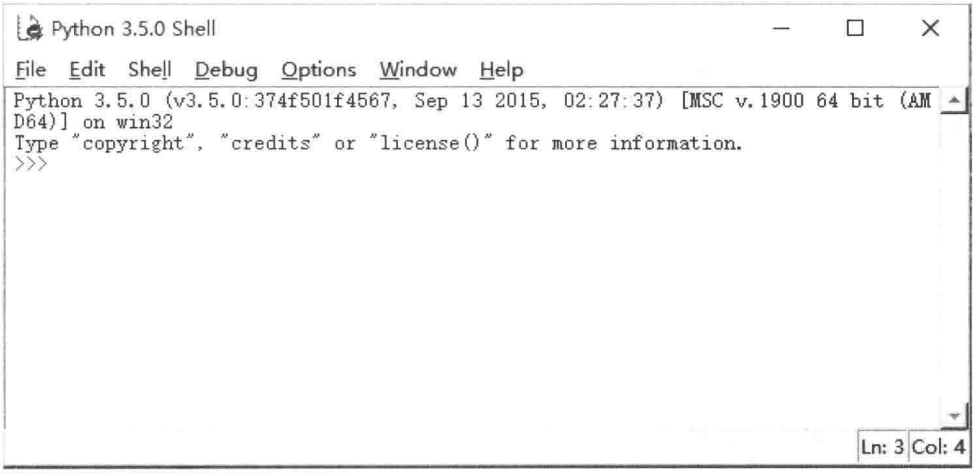

图 2.3　Python Shell 界面

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，如图 2.4 所示。

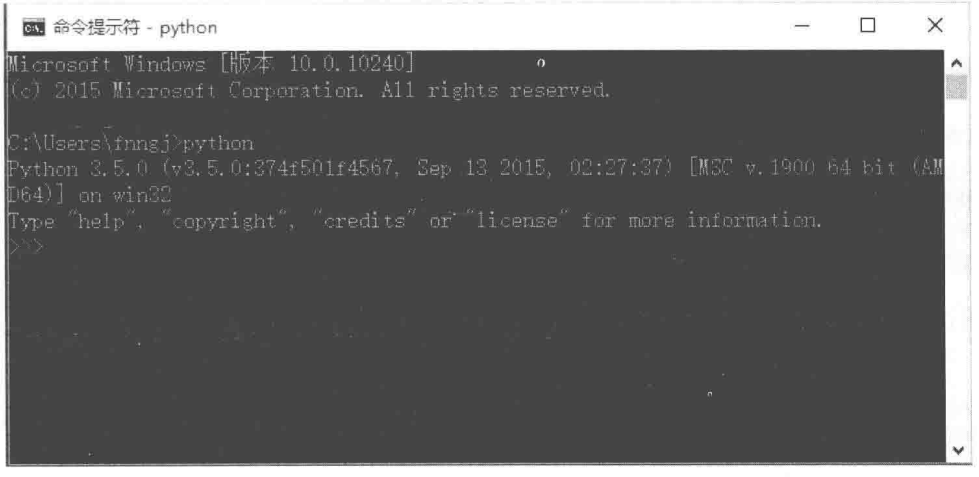

图 2.4　命令提示符

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，右击桌面“我的电脑”，打开右键菜单，在属性→高级→环境变量→系统变量的 Path 中添加：

变量名：PATH

变量值：;C:\Python35

也可以在图 2.1 所示 Python 安装界面，勾选 Add Python 3.5 to PATH 复选框，在安装完成后自动完成 PATH 配置工作。

2.1.2 安装 setuptools 与 pip

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

经常使用 Python 的读者可能会注意到，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 包里带的一个简易安装命令，因此使用 easy_install 时实际上是在调用 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 来安装 Python 包变得十分简单，我们将省去搜索→查找版本→下载→安装等烦琐的过程。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 Python 3 并不支持 setuptools，因此需要使用 distribute。

setuptools 与 pip 下载地址如下：

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（随着时间变化，包的版本号会有所更新）。

setuptools-18.4.zip

pip-7.1.2.tar.gz

通过解压缩工具进行解压，将得到相应的文件夹。在 Windows 命令提示符下进入文件解压缩目录，通过 python 命令执行 setup.py 进行安装。以下分别为安装 setuptools 与 pip 的命令：

cmd.exe

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

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

不过，在最新 Python 安装包中已经集成了 pip，读者可以到 Python 安装目录 C:\Python35\Script\下查看是否有 pip.exe 或 pip3.exe 文件。如果有则可以直接在 Windows 命令提示符下输入 pip 或 pip3 命令：

cmd.exe

```
C:\Users\fnngj>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，则可以手动将 C:\Python35\Scripts\目录添加到系统环境变量下的 Path 下面，重新

打开 cmd 命令行验证。

2.1.3 安装 Selenium

Selenium 这里不再过多介绍，前面安装 pip 是为了更方便地安装 Selenium 包，通过 pip 命令可直接安装 Selenium 包：

cmd.exe

```
C:\Users\fnngj> pip install Selenium
```

安装 pip 的好处是可以使用 pip 命令方便地安装 Python 第三方库，就像当前安装 Selenium 一样简单。在通过 pip 安装 Python 第三方库时，如果只输入包名，则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，则可以在包名后面加版本号。

cmd.exe

```
C:\Users\fnngj>pip install selenium==2.48.0   //指定版本号安装
C:\Users\fnngj>pip show selenium       //查看当前包的版本信息
---
Metadata-Version: 1.1
Name: selenium
Version: 2.48.0
Summary: Python bindings for Selenium
Home-page: https://github.com/SeleniumHQ/selenium/
Author: UNKNOWN
Author-email: UNKNOWN
License: UNKNOWN
Location: c:\python35\lib\site-packages
Requires:
C:\Users\fnngj>pip uninstall selenium   //卸载当前安装的包
```

pip 下面包含了很多命令，正如我们前面只输入一个有 pip 后回车所得到的提示。show 命令可查看安装包的版本及安装路径。

2.1.4 ActivePython

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

ActivePython 包含了完整的 Python 内核，可直接调用 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，同时还提供了访问 Windows APIs 的所有服务。ActivePython 虽然不像纯 Python 那样是开源的，但也可以免费下载使用。

使用 ActivePython 的好处是它已经集成了 pip 包管理工具，可以直接通过 pip 命令来安装 Python 第三方库。

ActivePython 下载地址如下：

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

ActivePython 同样支持 Windows、Mac 和 Linux 等平台，读者可根据自己的平台下载相应的 ActivePython 版本，安装界面如图 2.5 所示。

图 2.5　ActivePython 安装界面

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

安装 ActivePython 后，可以以同样的方式使用 pip 命令安装 Selenium 库，过程同上，这里不再赘述。

2.2 Ubuntu 下的环境搭建

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。笔者曾因不小心卸载了 Ubuntu 系统自带的 Python，从而导致系统无法正常启动，这一点也说明了 Python 在不同领域都有非常广泛的应用。

当前在 Ubuntu 系统中已经同时集成了 Python 2 与 Python 3，打开终端，输入“python 2”或“Python 3”命令回车，即可进入相应版本的 Python Shell 模式。

ubuntu 终端

```
fnngj@fnngj-PC:~$  python
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> quit()

fnngj@fnngj-PC:~$  python3
Python 3.4.3 (default, Jul 28 2015, 18:20:59)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> quit()
```

下面我们在 Ubuntu 的 Python 3 下安装 setuptools 与 pip，因为它们已经存在于 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 等发行版 Linux 系统的包管理工具。

安装 setuptools 的命令如下：

ubuntu 终端

```
fnngj@fnngj-PC:~$ sudo apt-get install python3-setuptools
```

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

例：sudo apt-get xxxx

用同样的方法安装 pip。

ubuntu 终端

```
fnngj@fnngj-PC:~$ sudo apt-get install python3-pip
```

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

如果想使用 Python 3 的 pip 安装 Selenium，可以通过以下命令。

ubuntu 终端

```
fnngj@fnngj-PC:~$ python3 -m pip install selenium
```

2.3　使用 IDLE 编写 Python

通过上面烦琐的配置后我们终于搭建好需要的自动化开发环境了，那么你一定迫不及待要跟着我一起写自动化脚本了，别急！在此之前我们需要先找到合适的 IDE（Integrated Development Environment，集成开发环境）。如果你是一位编程老手，那么你一定有自己趁手的 IDE；如果是一位编程菜鸟，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 IDE，允许你在这个 IDE 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 2.6 所示。

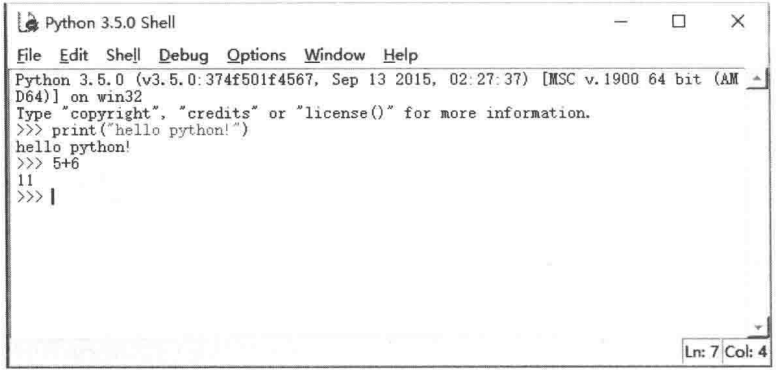

图 2.6　Python Shell 下输入代码

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。

1. 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 会自动匹配出相应的关键字。通过键盘上下键进行选择，从而提高代码输入速度，降低输错率，如图 2.7 所示。

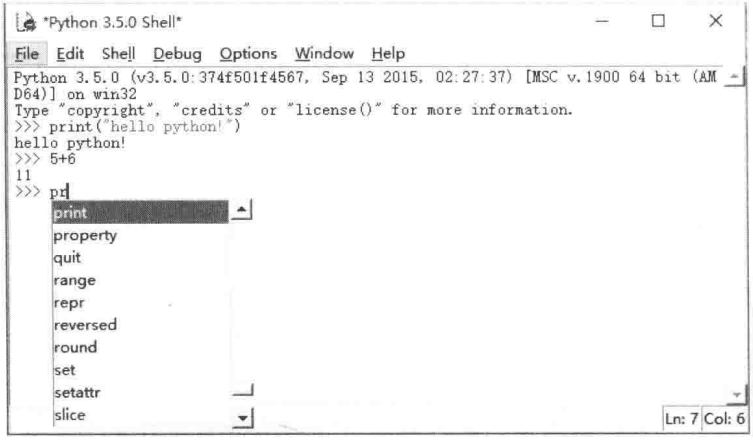

图 2.7　Tab 键自动补全

2. 回退代码语句

可以通过组合键 Alt+P 回退到上一次编辑的 Python 代码，组合键 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 2.8 所示。

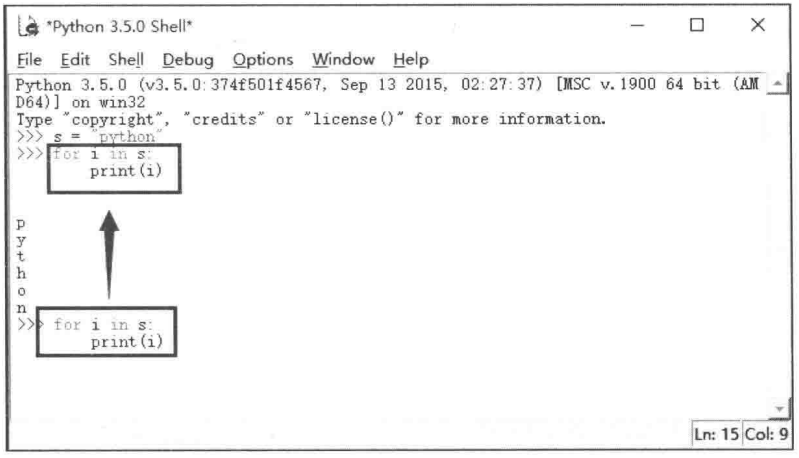

图 2.8　Alt+P 回退

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。如果我们想把代码写到文件里保存起来，则可以单击菜单栏 File→New File，或通过组合键 Ctrl+N 打开新的窗口，在此文件中编写代码，完成后单击菜单栏 File→Save 或通过组合键 Ctrl+S 保存，如图 2.9 所示。

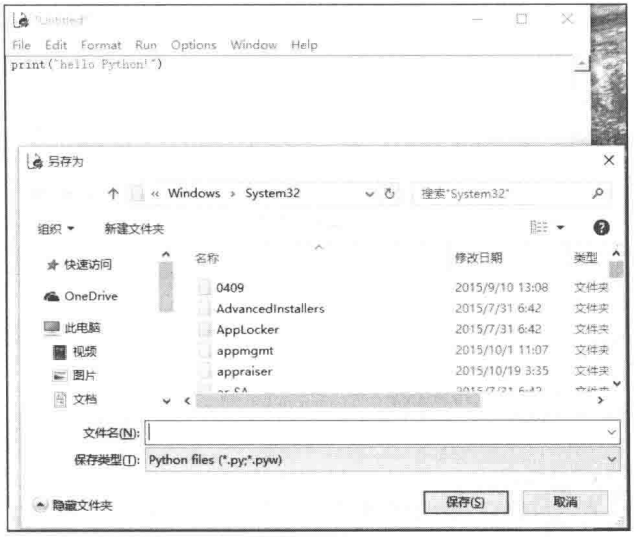

图 2.9　保存 Python 程序

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 IDLE 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，来编写你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()
```

在敲这段代码的时候，你一定充满了疑问，这段代码到底做了什么事情？下面就来逐行解释代码的含义。

1）#coding=utf-8

为了防止乱码问题，以及方便地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，下面的写法也可以起到相同的作用。

2）# -*- coding: utf-8 -*-

在 Python 2 时代，一般需要在程序文件头部指定编码类型，不过到了 Python 3，编码类型的指定就显得不那么重要了。

3）from selenium import webdriver

导入 Selenium 的 WebDriver 包，只有导入 WebDriver 包，才能使用 WebDriver API 进行自动化脚本开发。在 Python 下面，通过 from... import ...或 import...引入模块。

4）driver= webdriver.Firefox()

把 webdriver 的 Firefox 对象赋值给变量 driver。只有获得了浏览器对象后，才可以启动

浏览器。打开网址，操作页面元素，Firefox 浏览器驱动默认已经在 Selenium WebDriver 包里了，所以可以直接调用。如果要使用 IE 或 Chrome 浏览器运行 Web 自动化测试用例，则需要先安装相应的浏览器驱动才行。

5）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）。

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

关于页面元素的定位在后面会详细介绍，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。

7）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() 。

8）driver.quit()

退出并关闭浏览器及相关的驱动程序。

保存文件为 baidu.py，按快捷键 F5 运行脚本，会看到脚本启动 Firefox 浏览器后进入百度主页，输入“Selenium2” 后，单击搜索按钮，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 2.10 所示。

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)等浏览器。除此之外，它还支持 Android (AndroidDriver)和 iPhone (IPhoneDriver) 的移动应用测试，而且还包括一个基于 HtmlUnit 的无界面实现，相关驱动为 HtmlUnitDriver。

各个浏览器驱动下载地址：

http://www.seleniumhq.org/download/

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:\Python35 ）添加到了系统环境变量 Path 下面，所以可以将 chromedriver.exe 放到 C:\Python35\目录下。

安装 IE 浏览器驱动，下载 IEDriverServer_Win32_x.xx.zip，解压得到 IEDriverServer.exe，同样放置到 C:\Python35\目录下。

在 Linux 系统下，同样需要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 环境变量设置也会有所区别，这里不再介绍。

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

将 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，则说明我们的浏览器驱动安装成功。

2.6 不同编程语言下使用 WebDriver

W3C（万维网联盟）对 WebDriver 对做定义和规范。

http://www.w3.org/TR/webdriver/

WebDriver 是一个远程控制界面，支持反省和用户代理的控制权。它提供了一个独立于系统平台和编程语言的线协议（Wire Protocol）作为一种远程进程程序指示 Web 浏览器的行为。

该 WebDriver API 通过通信协议和一组接口发现页面上 DOM 元素中定义的操作，包括控制浏览器的行为。

可以这样理解，例如，国标标准定义好插板和插头的生产标准后，所有的电器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与生产，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 可以理解成对操作浏览器和页面元素的一套“国标”，不同的编程语言都可以按照这套标准实现自己语言的 WebDriver 库。

下面展示在不同编程语言下使用 WebDriver 实现百度搜索的例子。

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

BaiduTest.java

```
package com.test.case;

//添加Selenium(webdriver)引用
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.*;

public class BaiduTest {

   public static void main(String[] args) {

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

       driver.findElement(By.id("kw")).sendKeys("selenium2");
```

```
        driver.findElement(By.id("su")).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 :firefox
driver.navigate.to "http://www.baidu.com"

driver.find_element(:id, 'kw').send_keys "selenium2"
driver.find_element(:id, 'su').click()

driver.quit
```

在不同的编程语言中会有语法的差异，我们抛去这些差异，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（百度输入框和按钮）。

⑤ 通过句柄关闭浏览器。

所以，WebDriver 支持多种编程语言，也可以看作是多种语言都支持 WebDriver，唯一的不同在于不同语言实现的类与方法名的命名差异性。当然，这样做的好处不言而喻：每个人都可以根据自己熟悉的语言来使用 WebDriver 编写自动化测试脚本。

第 3 章
Python基础

本书中所介绍的 Selenium WebDriver 技术是基于 Python 语言实现的，所以，本章我们就先来学习和使用 Python 语言。需要说明的是，本章默认读者具备一定的编程基础，因此这里不会花费章节来解释什么是变量、什么是运算符等基础概念。如果本章内容你学习起来有难度的话，建议找一本 Python 基础教程来系统地学习 Python 语言。

当然，如果你有一定编程基础，那么 Python 对你来说将非常容易掌握。相信本章的学习一定可以让你快速地了解并使用这一门编程语言。如果你已经完全掌握了 Python 语言的使用，那么可以直接跳过本章。

我们在第 2 章中提到了 Python 自带 IDE--Python IDLE 的使用，所以后面的所有练习默认也将会在这个 IDE 下完成。

3.1 Python 哲学

在进入 Python 的具体学习之前，我们先来看一些有趣的东西。打开 Python Shell，输入 import this，你会看到下面的一段话。

Python Shell

```
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:27:37) [MSC v.1900 64 bit
(AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
```

```
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Beautiful is better than ugly. 优美胜于丑陋。

Explicit is better than implicit. 明了胜于晦涩。

Simple is better than complex. 简单胜过复杂。

Complex is better than complicated. 复杂胜过凌乱。

Flat is better than nested. 扁平胜于嵌套。

Sparse is better than dense. 间隔胜于紧凑。

Readability counts. 可读性很重要。

Special cases aren't special enough to break the rules.

即使假借特例的实用性之名，也不违背这些规则。

Although practicality beats purity. 虽然实用性次于纯度。

Errors should never pass silently. 错误不应该被无声的忽略。

Unless explicitly silenced.　　除非明确的沉默。

In the face of ambiguity, refuse the temptation to guess.

当存在多种可能时，不要尝试去猜测。

There should be one-- and preferably only one --obvious way to do it.

应该有一个，最好只有一个，很明显可以做到这一点。

Although that way may not be obvious at first unless you're Dutch.

虽然这种方式可能不容易，除非你是 Python 之父

Now is better than never.　　现在做总比不做好。

Although never is often better than *right* now.

虽然过去从未比现在好

If the implementation is hard to explain, it's a bad idea.

如果这个实现不容易解释，那么它肯定是坏主意。

If the implementation is easy to explain, it may be a good idea.

如果这个实现容易解释，那么它很可能是个好主意。

Namespaces are one honking great idea -- let's do more of those!

命名空间是一种绝妙的理念，应当多加利用！

这就是 Python 之禅，也可以看作是 Python 设计哲学。在我们接下来的 Python 学习中也将体会到这种设计哲学。

3.2　输出与输入

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

3.2.1 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

```
>>> print("hello python")
hello python
```

使用 print()方法：用双引号（""）把需要打印的字符串引起来，然后就可以看到字符串内容被打印出来了。可是，有时候我们打印的信息不是固定的，下面来看如何把变量的信息打印出来。

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!
```

虽然两次打印语句一样，但由于定义的变量 name 两次赋值不同，所以打印出的结果也不完全相同。%s（string）只能打印字符串，如果想要打印数字，则需要使用%d（data）指定打印信息的类型。

Python Shell

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

可是，有时候我们并不知道自己要打印的是什么类型的信息，这时可以用%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' .

>>> name = "zhangsan"
>>> age = 22
>>> print("student info : %s %d ." %(name,age))
student info : zhangsan 22 .
```

3.2.2　input 输入

其实，上面的例子中打印的变量信息都是事先拟定好的，例如 name=“zhangsan”。

如果希望打印的信息是在程序运行过程中由用户来决定，则可以用 Python 提供的 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。

input_demo.py

```
n = input("Enter any content: ")
print "Your input is %r " %n
```

按快捷键 F5 运行程序，当运行到 input()时，需要用户输入一些信息，而 print 将会把用户输入的内信息打印出来。

Python Shell

```
==================== RESTART :D:/demo/input_demo.py ========================
Enter any content: Tom
Your input is 'Tom'
```

在 Python 2 中，input()要求用户输入的字符串必须加引号（“Tom”），为了避免读取非字符串类型发生的一些危险行为，不得不使用 raw_input()代替 input()。

3.2.3　引号与注释

在 Python 当中，不区分单引号（''）与双引号（""），也就是说，单引号和双引号都可以用来表示一个字符串。

Python Shell

```
>>> print("hello")
hello
>>> print('world')
world
```

单引号与双引号可以互相嵌套使用，但不能交叉使用。

Python Shell

```
>>> print("你说：'早上你好'")
你说：'早上你好'
>>> print('我说："今天天气不错"')
我说："今天天气不错"
>>> print ("你微笑着'向我道别" 。')
SyntaxError: invalid character in identifier
```

再来看看注释，基本上每种语言都会提供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，同样不区分单、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
'''
```

3.3　分支与循环

结构化程序实质上是由有限个顺序、分支和循环三种基本结构排列、 嵌套而成。下面来学习 Python 如何实现分支与循环。

3.3.1　if 语句

和很多语言一样，Python 通过 if 语句来实现分支判断，一般语法为 if...else...。

Python Shell

```
>>> a = 2
>>> b = 3
>>> if a > b:
        print("a max!")
 else:
        print("b max!")

b max!
```

上面的语句分别对 a 和 b 赋值，通过 if 语句判断 a 与 b 的大小，如果 a 大于 b 则输出“a max!”，否则输出“b max!”。

需要强调的是，Python 没有像其他大多数语言一样使用“｛｝”表示语句体，所以，它通过语句的缩进来判断语句体，缩进默认为 4 个空格。

Python 中的 if 语句通过“==”运算符判断相等。

Python Shell

```
>>> student = "xiaoming"
>>> if student == "xiaoming":
        print("xiaoming, You are on duty today.")
    else:
        print("Please call xiaoming to duty")


xiaoming, You are on duty today.
```

在 Python 中，如果判断不相等，则用“!=”运算符表示。除此之外，Python 的 if 语句

还可以用“in”和“not in”表示包含的关系。

Python Shell

```
>>> hi = "hello world"
>>> if "hello" in hi:
        print("Contain")
else:
        print("Not Contain")


Contain
```

if 语句甚至可以进行布尔类型的判断。

Python Shell

```
>>> a = True
>>> if a:
        print("a is True")
    else:
        print("a is not True")


a is True
```

下面通过一个多重条件判断来结束 if 语句的学习。

if_demo.py

```
results = 72

if results >= 90:
   print('优秀')
elif results >= 70:
   print('良好')
elif results >= 60:
   print('及格')
else:
   print('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！

3.3.2　for 语句

当然，Python 语言同样提供了 while 循环，但从大多数程序员的习惯来看，它的使用率远不及 for 循环，所以这里重点介绍 for 循环的使用。Python 中 for 循环的使用更加灵活、简单，例如，我们可以直接对一个字符串进行遍历。

Python Shell

```
>>> for i in "hello world":
        print(i)


h
e
l
l
o

w
o
r
l
d
```

当然，也可以对一个字典（稍后学习 Python 中的字典）直接遍历。

Python Shell

```
>>> fruits = ['banana', 'apple',  'mango']
>>> for fruit in fruits:
        print(fruit)


banana
apple
mango
```

如果需要进行一定次数的循环，则需要借助 range()函数。

Python Shell

```
>>> for i in range(5):
        print(i)


0
1
2
3
4
```

range()函数默认从零开始循环，我们也可以为其设置起始位置和步长。例如，打印 1 到 10 之间的奇数：

Python Shell

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


1
3
5
7
9
```

range(start,end[,step])

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

在 Python 2 中 range()是一个生成器，xrange()是一个数组，后者在性能上要优于前者，因为不需要一上来就开辟一块很大的内存空间，但它们的用法完全相同。而 Python 3 中的 range()与 Python 2 的 xrange()相同，是一个数组。

3.4　数组与字典

数组与字典是最常见的两种用于存放数据的形式，Python 中字典与数组的用法非常灵活，在这里进行简单的介绍。

3.4.1　数组

数组用方括号（[]）表示，里面的每一项用逗号（，）隔开。

Python Shell

```
>>> lists = [1,2,3,'a',5]
>>> lists
[1, 2, 3, 'a', 5]
>>> lists[0]
1
>>> lists[4]
5
>>> lists[4]='b'
>>> lists[4]
'b'
>>> lists.append('c')
>>> lists
[1, 2, 3, 'a', 'b', 'c']
```

Python 允许在数组里面任意地放置数字或字符串。需要注意的是，数组下标是从 0 开始的，所以，lists[0]会输出数组中的第一项。append()函数可以向数组末尾追加新的项。

3.4.2　字典

字典用花括号（{}）表示，里面的项成对出现，一个 key 对应一个 value；key 与 value 之间用冒号（：）分隔；不同的项之间用逗号（,）分隔。

Python Shell

```
>>> dicts = {"username":"zhangsan",'password':123456}
>>> dicts.keys()
['username', 'password']
>>> dicts.values()
['zhangsan', 123456]
```

```
>>> dicts.items()
[('username', 'zhangsan'), ('password', 123456)]
>>> for k,v in dicts.items():
        print("dicts keys is %r " %k)
        print("dicts values is %r " %v)


dicts keys is 'username'
dicts values is 'zhangsan'
dicts keys is 'password'
dicts values is 123456
```

注意：Python 规定一个字典中的 key 必须独一无二，value 可以相同。

keys()函数返回字典 key 的列表，values()函数返回字典 value 的列表，items()函数将所有的字典项以列表方式返回，这些列表中的每一项都包含 key 和 value，但是项在返回时并不会按照它们在字典中的存放顺序。如果想按存放的顺序输出，则可以通过下面的方法。

zidian.py

```
# 通过 zip 方法合并两个 List 为 Dictionary
# 遍历会按原先的顺序
keys = ["b", "a", "c", "e", "d"]
values = ["2", "1", "3", "5", "4"]

for key,value in zip(keys, values):
   print(key, value)
```

输出结果：

```
==================== RESTART :D:/demo/zidian.py ============================
b 2
a 1
c 3
e 5
d 4
```

3.5　函数、类和方法

利用前面讲的知识只能搭建一个鸡窝，要想建造一个庞大且结构复杂的大厦，就不得不介绍函数、类和方法的使用。

3.5.1　函数

在 Python 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()函数，此函数接收两个参数 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。

通常 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
```

有时我们在调用 add()函数的时候不想传参，这时可以为 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。

3.5.2 类和方法

在面向对象编程的世界里，一切皆为对象，抽象的一组对象就是类。例如，汽车是一个类，而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。

class_test.py

```
class A(object):

    def add(self, a, b):
        return a + b

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

输出结果：

```
==================== RESTART :D:/demo/class_test.py ========================
8
```

上面创建了一个 A 类（在 Python 3 中 object 为所有类的基类，所有类在创建时默认继承 object，所以不声明继承 object 也可以），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，方法的第一个参数必须是存在的，一般习惯命名为“self”，但是在调用这个方法时不需要为这个参数传值。

一般在创建类时会首先声明初始化方法__init__()。

注意：init 的两侧是双下画线，当我们在调用该类时，可以用来进行一些初始化工作。

class_test.py

```
class A():

    def __init__(self, a, b):
        self.a = int(a)
```

```
        self.b = int(b)

    def add(self):
        return self.a + self.b


count = A('4', 5)
print(count.add())
```

输出结果：

```
==================== RESTART :D:/demo/class_test.py ==========================
9
```

当我们调用 A 类时首先会执行的它的__init__()方法，所以需要对其进行传参。初始化所做的事情就是将输入的参数类型转化为 int 类型，这样可以在一定程度上保证程序的容错性。而 add()方法可以直接拿初始化方法__init__()的 self.a 和 self.b 两个数进行计算。所以，我们在调用 A 类下面的 add()方法时，不需要再进行传参。

下面再来了解一下 Python 中类的继承。

class_test.py

```
class A():

    def add(self, a, b):
        return a + b


class B(A):

    def sub(self, a, b):
        return a - b

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

输出结果：

```
==================== RESTART :D:/demo/class_test.py ==========================
9
```

首先，我们创建了一个 A 类，在其下面创建 add()方法用于计算两个参数相加；接着创建 B 类，继承 A 类，并且又继续创建了 sub()方法用于计算两个参数相减。因为 B 类继承了 A 类，所以 B 类自然也拥有了 add()方法，从而可以直接通过 B 类调用 add()方法。

3.6 模组

模组更通俗地叫类库或模块。前面的练习中我们没有用到模块，但这也只是在练习的时候，在实际开发中我们不可能不用到系统的标准模块，或第三方模块。

如果想实现与时间有关的功能，就需要调用系统的 time 模块。如果想实现与文件和文件夹有关的操作，就需要要用到 os 模块。再例如我们通过 Selenium 实现的 Web 自动化测试，那么 Selenium 对于 Python 来说就是一个第三方扩展模块。

3.6.1 引用模块

在 Python 中，通过 import...或 from...import...的方式引用模块，下面引用 time 模块。

imp.py

```
import time

print(time.ctime())
```

输出结果：

```
======================== RESTART :D:/demo/imp.py ============================
Tue Dec 09 22:35:57 2014
```

在 time 模块下面有一个 ctime()方法用于获得当前时间，通过 print()将当前时间打印出来。当然，如果确定了只会用到 time 下面的 ctime()方法，也可以这样引入。

imp.py

```
from time import ctime

print(ctime())
```

输出结果：

```
======================== RESTART :D:/demo/imp.py ============================
Tue Dec 09 22:36:38 2014
```

现在使用时就不必告诉 Python，ctime()方法是 time 模块所提供的了。但是有时候我们可能还会用到 time 模块下面的 sleep()休眠方法，当然，我们也可以把 sleep()方法引入进来。或许还会用到其他方法，这时可以一次性把 time 模块下的所有方法都引入进来。

imp.py

```
from time import *

print(ctime())
print("休眠两秒")
sleep(2)
print(ctime())
```

输出结果：

```
======================== RESTART :D:/demo/imp.py ============================
Tue Dec 09 22:47:35 2014
休眠两秒
Tue Dec 09 22:47:37 2014
```

星号“*”用于表示模块下面的所有方法。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 语言提供的核心方法，而且经过了编译，所以我们无法看到 ctime 是如何取到系统的当前时间的，不过，可以通过 help()方法查看 time 的帮助说明。

Python Shell

```
>>> import time
>>> help(time)
Help on built-in module time:

NAME
    time - This module provides various functions to manipulate time values.

DESCRIPTION
    There are two standard representations of time.  One is the number
```

```
of seconds since the Epoch, in UTC (a.k.a. GMT).  It may be an integer
or a floating point number (to represent fractions of seconds).
The Epoch is system-defined; on Unix, it is generally January 1st, 1970.
The actual value can be retrieved by calling gmtime(0).

The other representation is a tuple of 9 integers giving local time.
The tuple items are:
  year (including century, e.g. 1998)
  month (1-12)
  day (1-31)
  hours (0-23)
  minutes (0-59)
  seconds (0-59)
  weekday (0-6, Monday is 0)
  Julian day (day in the year, 1-366)
  DST (Daylight Savings Time) flag (-1, 0 or 1)
If the DST flag is 0, the time is given in the regular time zone;
if it is 1, the time is given in the DST time zone;
if it is -1, mktime() should guess based on the date and time.

Variables:

timezone -- difference in seconds between UTC and local standard time
altzone -- difference in  seconds between UTC and local DST time
daylight -- whether local time should reflect DST
tzname -- tuple of (standard time zone name, DST time zone name)

Functions:

time() -- return current time in seconds since the Epoch as a float
clock() -- return CPU time since process start as a float
sleep() -- delay for a number of seconds given as a float
gmtime() -- convert seconds since Epoch to UTC tuple
localtime() -- convert seconds since Epoch to local time tuple
asctime() -- convert time tuple to string
ctime() -- convert time in seconds to string
mktime() -- convert local time tuple to seconds since Epoch
strftime() -- convert time tuple to string according to format specification
strptime() -- parse string to time tuple according to format specification
tzset() -- change the local timezone
.....
```

---

pip 所安装的 Python 的第三方类库或框架可以查看其类方法的实现。例如我们做 Web

自动化所用到的 Selenium 类库。

Python 所安装的第三方类库或框架模块默认存放在..\Python35\Lib\site-packages\目录下面，如果你已经学习了第 2 章并安装了 Selenium，那么就会在这个目录下找到 Selenium 目录。

3.6.2　模块调用

既然可调用系统模块，那么可不可以自己创建一个模块，然后通过另一个程序调用？当然可以，对于一个软件项目来说不可能把所有代码都放在一个文件中实现，它们一般会按照一定规则在不同的目录和文件中实现。

下面创建一个目录 project，并在目录下创建两个文件，结构如下：

```
project/
├── pub.py
└── count.py
```

在 pub.py 文件中创建 add 函数。

pub.py

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

在相同的目录下再创建一个文件 count.py，调用 pub.py 文件中的 add()函数。

count.py

```
from pub import add

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

输出结果：

```
===================== RESTART :D:/project/count.py ==========================
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__pycache__/pub.cpython-35.pyc 文件，那么它的作用是什么呢？

为了提高模块加载的速度，每个模块都会在__pycache__文件夹中放置该模块的预编译模块，命名为 module.version.pyc， version 是模块的预编译版本编码，通常会包含 Python 的版本号。例如在 CPython 发行版 3.5 中，pub.py 文件的预编译文件就是：__pycache__/pub.cpython-35.pyc。

3.6.3 跨目录模块调用

如果调用文件与被调用文件在一个目录下面，则可以非常方便地调用。那么如果被调用的文件与调用文件不在同一目录下呢？假设文件目录结构如下：

```
project/
├── model/
│   └── pub.py
└── count.py
```

count.py

```
from model.pub import add

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

输出结果：

```
==================== RESTART :D:/project/count.py ==========================
9
```

在 Python 2 中将会抛出 ImportError：找不到名字为 model 的模块。我们稍后再讨论 Python 2 如何才能找到 model 下面的 pub.py 文件。

3.6.4 进一步讨论跨目录模块调用

当项目变得复杂之后，需要涉及多个文件跨目录之间的调用。我们进一步探讨下面的

结构。

```
project/
├── model/
│   ├── count.py
│   └── new_count.py
└── test.py
```

代码实现如下：

count.py

```
class A():

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

new_count.py

```
from count import A

class B(A):

    def sub(self,a, b):
        return a - b

resule = B().add(2, 5)
print(resule)
```

输出结果

```
===================== RESTART :D:/project/model/test.py =====================
7
```

到目前为止不管是 Python 2 还是 Python 3 执行 new_count.py 都没有问题。接下来与 model 目录平级创建 test.py。

test.py

```
from model import new_count

test = new_count.B()
test.add(2,5)
```

输出结果

```
===================== RESTART :D:/project/test.py ==========================
Traceback (most recent call last):
  File "D:\project\test.py", line 3, in <module>
    from model import new_count
  File "D:\project\model\new_count.py", line 1, in <module>
    from count import A
ImportError: No module named 'count'
```

通过提示信息，在 new_count.py 文件中，找到不到“count”模块，可刚才在执行 new_count.py 时是可以正常运行的。那么，要想弄清错误的原因，首先需要知道当 Python 执行“import”时到底做了哪些操作？

知识延伸 当 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 module 对象（它可能包含多个 module）；

第 2 步，把这个 module 对象插到 sys.module 中；

第 3 步，装载 module 的代码（如果需要，则必须先编译）；

第 4 步，执行新的 module 中对应的代码。

在执行第 3 步时，首先需要找到 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），PythonPATH，再后是 Python 安装时设置的相关的默认路径。正因为存在这样的顺序，所以如果当前路径或 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，那么在执行 import xml 时，导入的是当前目录下的 module，而不是系统标准的 xml。

了解了这些后，我们就可以先构建一个 package，以普通 module 的方式导

入，这样即可直接访问此 package 中的各个 module。Python 2 中的 package 必须包含一个__init__.py 的文件。

理解了上面的过程，就很好理解报错原因了。站在 new_count.py 的位置，执行“from count import A”，可查看当前目录下是否存“count”名字的文件或目录，当然是可以找到的；但是，站在 test.py 的位置执行“from count import A”时，同样会在当前目录下找“count”名字的文件或目录，这个时候就找不到了。那么如何解决这个问题呢？

简单的做法是将导入方法修改为“from .count import A”，在 count 的前面加个点（.），用来告诉调用程序（test.py）count 是相对于 new_count.py 的一个引入。读者可以尝试这样修改后再次运行 test.py 看是否还会报错？

不过这样的修改有副作用，当我们再次执行 new_count.py 时会引起新的错误。

new_count.py

```
from .count import A

class B(A):

    def sub(self,a, b):
        return a - b

resule = B().add(2, 5)
print(resule)
```

输出结果

```
===================== RESTART :D:/project/model/test.py =====================
Traceback (most recent call last):
  File "D:\project\model\new_count.py", line 1, in <module>
    from .count import A
[Finished in 0.1s with exit code 1]SystemError: Parent module '' not loaded,
cannot perform relative import
```

Python 3 提示：“未加载父模块，不能执行相对导入”。如果你认真阅读了 Python 的 import 规则，就知道我们可以将导入模块所在目录（.../model/目录）添加到系统环境变量 path 下，这样 Python 就可以找到了。

下面还原对 new_count.py 的修改，修改 test.py 文件如下。

test.py

```
import sys
sys.path.append("./model")  #将 model 目录添加到系统环境变量 path 下
from model import new_count

test = new_count.B()
test.add(2,5)
```

输出结果

```
===================== RESTART :D:/project/test.py ==========================
7
```

如果读者使用的是 Python 2 的话还需要在.../model/目录下创建一个__init__.py 文件（文件内容可以为空），用来标识这是一个标准的包含了 Python 模块的目录。

3.7 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，则程序就会用所谓的回溯（Traceback，一种错误信息）来终止执行。

在实际脚本开发中，有时程序并不像我们设计它时那样工作，它也有“生病”的时候，这时我们就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。例如一个普通人，大冬天洗冷水澡，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。

3.7.1 认识异常

下面来看程序在执行时所抛出的异常。

Python Shell

```
>>> open("abc.txt",'r')
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
```

```
    open("abc.txt",'r')
FileNotFoundError: [Errno 2] No such file or directory: 'abc.txt'
```

我们通过 open()方法以读“r”的方式打开一个 abc.txt 的文件。然后 Python 抛出一个 FileNotFoundError 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，因为我们根本就没创建这个文件。

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

abnormal.py

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

输出结果：

```
====================== RESTART: D:\demo\abnormal.py ========================
异常了!
```

再来运行程序，因为已经用 except 接收了这个 FileNotFoundError 错误，所以“异常了！”会被打印出来。修改程序，使其打印一个没有定义的变量。

abnormal.py

```
try:
    print(aa)
except FileNotFoundError:
    print("异常了!")
```

输出结果：

```
====================== RESTART: D:\demo\abnormal.py ========================
Traceback (most recent call last):
  File "D:\demo\abnormal.py", line 2, 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？如果细心查看错误信息就会发现，这一次抛出的是个 NameError 类型的错误，而 except FileNotFoundError 只能接收到找不到文件的错误。就好像小明是肚子痛，但我们拿感冒药给他吃，当然解决不了问题。

这时我们只需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!")


输出结果:
====================== RESTART: D:\demo\abnormal.py ========================
这是一个 name 异常!
```

那么问题来了，我们的程序是怎么抛出不同类型错误的呢？

知识延伸 异常的抛出机制：

1. 如果在运行时发生异常，则解释器会查找相应的处理语句（称为 handler）。

2. 如果在当前函数里没有找到的话，则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，那么解释器就会退出，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，但异常不一定代表错误，有时候它们只是一个警告，有时候是一个终止信号，例如退出循环等。

在 Python 中所有的异常类都继承 Exception，所以可以使用它来接收所有类型的异常。

abnormal.py

```
try:
```

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

输出结果：

```
====================== RESTART: D:\demo\abnormal.py ========================
异常了!
```

从 Python 2.5 版本之后，所有的异常类都有了新的基类 BaseException。Exception 同样也继承自 BaseException，所以我们也可以使用 BaseException 来接收所有类型的异常。

abnormal.py

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

输出结果：

```
====================== RESTART: D:\demo\abnormal.py ========================
异常了!
```

对于上面的例子，只要其中一行出现了异常就会 print()异常信息，但是当打印异常时，我们并不能准确地知道到底是哪一行代码引起了异常，那么如何让 Python 直接告诉我们异常的原因呢？

abnormal.py

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

输出结果：

```
====================== RESTART: D:\demo\abnormal.py ========================
[Errno 2] No such file or directory: 'abc.txt'
```

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

此处的写法与 Python 2 有所不同，在 Python 2 中用逗号“，”代替“as”。

Python 中常见的异常如表 3.1 所示。

表 3.1 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
FileNotFoundError	试图打开一个不存在的文件或目录
AttributeError	试图访问的对象没有属性
OSError	当系统函数返回一个系统相关的错误，包括 I / O 故障，如“找不到文件”或“磁盘已满”时，引发此异常
NameError	使用一个还未赋值对象的变量
IndexError	当一个序列超出了范围
SyntaxError	当解析器遇到一个语法错误时引发
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

3.7.2 更多异常用法

通过前面的学习，我们了解了异常的一般用法，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 as msg:
    print(msg)
else:
    print("没有异常!")
```

输出结果：

```
====================== RESTART: D:\demo\abnormal.py ======================
```

```
异常测试
没有异常!
```

这里我们对 aa 变量进行了赋值，所以没有异常将会执行 else 语句后面的内容。通常 else 语句只有在没有异常的情况下才会被执行，但有些情况下不管是否出现异常，这些操作都希望能被执行，例如文件的关闭、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...except...finally...语句来实现这样的需求。

abnormal.py

```
try:
    print(aa)
except Exception as e:
    print(e)
finally:
    print("不管是否异常，我都会被执行。")
```

```
输出结果:
====================== RESTART: D:\demo\abnormal.py ========================
name 'aa' is not defined
不管是否异常，我都会被执行。
```

下面修改代码，定义 aa 变量。

abnormal.py

```
try:
    aa = "异常测试："
    print(aa)
except Exception as e:
    print(e)
finally:
    print("不管是否异常，我都会被执行。")
```

```
输出结果:
====================== RESTART: D:\demo\abnormal.py ========================
异常测试:
不管是否异常，我都会被执行。
```

对比两次的执行结果，就可以理解了 finally 语句的作用了。

3.7.3 抛出异常

print()方法只能打印错误信息，Python 中提供了 raise 方法来抛出一个异常信息。下面例子演示了 raise 的用法。

abnormal.py

```
from random import randint

# 生成一个 1 到 9 之间的随机整数
number = randint(1,9)

if number % 2 == 0:
   raise NameError("%d is even" % number)
else:
   raise NameError("%d is odd" % number)
```

输出结果：

```
====================== RESTART: D:\demo\abnormal.py ========================
Traceback (most recent call last):
  File "D:\project\count.py", line 8, in <module>
    raise NameError("%d is even" % number)
NameError: 4 is even
```

通过 randint()方法随机生成 1 到 9 之间的整数，然后判断这个数字是奇数还是偶数，最后通过 raise 抛出 NameError 异常。其实，判断奇偶数与 NameError 之间没有任何关系，这里只是为了演示如何通过 raise 抛出各种类型的异常。

需要注意的是，raise 只能使用 Python 中所提供的异常类，如果自定义一个 abcError 的异常，则 Python 会告诉你 abcError 没有定义。

本章小结

本章主要针对 Python 的一些语法与规则作了简单的介绍，但介绍的知识点并不全面，主要是以我们自动化测试中所用到的知识为出发点，所以，建议读者找一本 Python 教程来系统地学习 Python 语言。

另外，需要提醒初学者一些不专业的地方与坑。

1．项目不要创建在 Python 的安装目录中，初学者的心态是只有把程序建在 Python 的安装目录下才能运行，其实不然。例如你在 C 盘安装了音乐播放器，则只要把音乐文件设置为由该播放器打开，那么在硬盘任何一个角落的音乐文件都能由该播放器打开。Python 程序也是如此，只要正确地把 Python 目录配置到环境变量 path 下，任何目录下的 Python 程序都可以被执行。

2．项目的路径中不要出现中文或空格。例如，D:\自动化测试\xx 项目\test case list\test.py，这可能会导致有些 IDE 打开该程序后无法执行，例如 Sublime Text 就无法运行这种目录下的文件。

3．项目的目录与文件名不要与引用类库同名。例如，D:\selenium\webdriver.py ，这里会存在一个大坑，在创建目录与文件夹时一定要避免。如果不知道为什么，请回过头去看一下 Python 的引包机制。

第 4 章
WebDriver API

从本章开始正式学习 WebDriver，WebDriver 属于 Selenium 体系中设计出来操作浏览器的一套 API，站在 WebDriver 的角度，因为它针对多种编程语言都实现了一遍这套 API，所以它可以支持多种编程语言；站在编程语言的角度，WebDriver 是 Python 的一个用于实现 Web 自动化的第三方库。

本章内容参考官方 API，将最常用的一些方法结合具体 Web 应用展示给读者。

4.1 从定位元素开始

在本章学习开始之前，我们先来看一张 Web 页面，如图 4.1 所示。

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、页面底部的文字，以及左侧的下拉框等。自动化要做的就是模拟鼠标和键盘来操作这些元素，或单击，或输入，或鼠标悬停等。

操作这些元素的前提是需要找到它们，自动化工具无法像测试人员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，那么如何找到它们呢？下面来看看这些元素的真实面目。

图 4.1　Web 页面

如图 4.2 所示，通过前端工具，我们看可以看到，页面上的元素都是由一行行的代码组成的，它们之间有层级地组织起来，每个元素有不同的标签名和属性值。WebDriver 就是通过这些信息来找到不同的元素的。

图 4.2　通过 FireBug 查看页面元素

WebDriver 提供了八种元素定位方法，在 Python 语言中，所对应的方法如下：

- id → name
- class name → tag name
- link text → partial link text
- xpath → css selector
- 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()

下面我们就逐一介绍这些定位方法的使用。在此之前，我们复制百度首页的前端代码，并以此为例来讲解页面上元素的定位方法。

baidu.htmlindex.

```
<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>
....
    </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的页面代码，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>
```

html、div 就是标签的标签名。

② 标签有各种属性：

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

就像人也会有各种属性一样，如身份证号（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 标签有层级关系：

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

```
    </from>
  <div>
```

对于上面的结构，如果把 input 看作子标签，那么 form 就是它的父标签。

理解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，百度输入框和搜索按钮的代码如下。

```
……
<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">
……
```

如果把页面上的元素看作人的话，在现实世界中如何找到某人呢？

首先，可以通过人本身的属性，例如他的姓名，手机号，身份证号，性别，这些都是用于区别他人的属性。在 Web 页面上的元素也有这些属性，例如，id、name、class name、tag name 等。

其次，在找查找某人的时候可以通过位置属性，例如，x 国、x 市、x 路、x 号。XPath 和 CSS 就提供了这种以标签名为层级关系的定位方式。

最后，可以借助相关人的属性来找到某人，例如，我没有小明的联系方式，但我有他爸爸的手机号，那么通过他爸爸的手机号最终也可以找到小明。XPath 和 CSS 同样提供了这种通过相关元素来查找最终元素的方式。

理解了这些查找规则之后，下面所要介绍的几种定位方式就很好理解了。

4.1.1 id 定位

HTML 规定 id 属性在 HTML 文档中必须是唯一的，这类似于公民的身份证号，具有很强的唯一性。WebDriver 提供的 id 定位方法就是通过元素的 id 属性来查找元素。通过 id 定位百度输入框与百度搜索按钮，用法如下：

```
find_element_by_id("kw")

find_element_by_id("su")
```

find_element_by_id()方法通过 id 属性定位来元素。

4.1.2　name 定位

HTML 规定 name 来指定元素的名称，因此它的作用更像是人的姓名。name 的属性值，在当前页面中可以不唯一。通过 name 定位百度输入框：

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

find_element_by_name()方法通过 name 属性来定位元素。由于百度搜索按钮并没有提供 name 属性，因此我们不能通过 name 属性来定位它。

4.1.3　class 定位

HTML 规定 class 来指定元素的类名。其用法与 id、name 类似，下面通过 class 属性定位百度输入框和搜索按钮：

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

find_element_by_class_name()方法通过 class 属性来定位元素。

4.1.4　tag 定位

HTML 的本质就是通过 tag 来定义实现不同的功能，每一个元素本质上也是一个 tag。因为一个 tag 往往用来定义一类功能，所以通过 tag 识别某个元素的概率很低。例如我们打开任意一个页面，查看前端都会发现大量的 、<input>、<a>等 tag，所以很难通过标 tag name 去区分不同的元素。

通过标 tag name 定位百度的输入框与百度按钮会发现它们完全相同：

```
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。

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，它专门用来定位文本链接。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>
```

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。不过这里是为了演示 link 定位的使用，通过 link 定位链接如下：

```
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。

4.1.6 partial link 定位

parial link 定位是对 link 定位的一种补充，有些文本链接会比较长，这个时候我们可以取文本链接的一部分定位，只要这一部分信息可以唯一地标识这个链接。

```
<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("文本链接")
```

find_element_by_partial_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。

前面介绍的几种定位方法相对来说比较简单，理想状态下，在一个页面当中每一个元素都有一个唯一 id 和 name 属性值，我们可以通过它们的属性值来找到它们。但在实际项目中并非想象得这般美好，有时候一个元素并没有 id、name 属性，或者页面上多个元素的 id 和 name 属性值相同，又或者每一次刷新页面，id 值都会随机变化，这些情况下，我们如何来定位元素呢？

下面介绍 Xpath 与 CSS 定位，与前面介绍的几种定位方式相比，它们提供了灵活的定位策略，可以通过不同的方式定位到想要的元素。

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 可以看作 XML 的一种实现，所以 Selenium 用户可以使用这种强大的语言在 Web 应用中定位元素。

绝对路径定位

XPath 有多种定位策略，最简单直观的就是写出元素的绝对路径。如果仍把一个元素看作是一个人的话，假设这个人没有任何属性特征（手机号、姓名、身份证号），但这个人一定存在于某个地理位置，如 xx 省 xx 市 xx 区 xx 路 xx 号。对于页面上的元素而言也会有这样一个绝对地址。

参考 baidu.html 前端工具所展示的代码，我们可以通过下面的方式找到百度输入框和搜索按钮。

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

find_element_by_xpath()方法使用 XPath 语言来定位元素。XPath 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。在 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，例如 div[2]表示当前层级下的第二个 div 标签。

利用元素属性定位

除了使用绝对路径外，XPath 也可以使用元素的属性值来定位。同样以百度输入框和搜索按钮为例：

```
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[@name='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，则也可以用星号（*）代替。当然，使用 XPath 不局限于 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

如果一个元素本身没有可以唯一标识这个元素的属性值，那么我们可以找其上一级元素，如果它的上一级元素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，那么我们可以查找它的上一级属性。例如，“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”时可以先找到小明的爸爸，因为他爸爸是有很多属性特征的，找到了小明的爸爸后，就可以

找到小明了。通过 XPath 描述如下：

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

span[@class='bg s_ipt_wr'] 通过 class 属性定位到父元素，后面/input 就表示父元素下面的子元素。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地向上查找，直到找到最外层的<html>标签，这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地区分一个元素，我们还可以使用逻辑运算符连接多个属性来查找元素。

```
……
    <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，假设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 将会与第三行元素重名。如果同时使用 id 和 class 就会唯一地标识这个元素，这个时候就可以通过逻辑运算符“and”来连接两个条件。

```
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地标识一个元素。

我们在本书第 1 章中介绍的 Firebug 前端调试工具和 FirePath 插件可以方便地辅助生成 XPath 语法。打开 Firefox 浏览器的 FireBug 插件，单击插件左上角的鼠标箭头，再单击页面上需要定位的元素，在元素行上右击弹出快捷菜单，选择“复制 XPath”，将会获得当前元素的 XPath 语法，如图 4.3 所示。

图 4.3　通过 FireBug 复制 XPath 语法

FirePath 插件的使用更加方便和快捷，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 所示。

图 4.4　通过 FirePath 生成 XPath 语法

4.1.8　CSS 定位

CSS（Cascading Style Sheets）是一种语言，它用来描述 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 可以较为灵活地选择控件的任意属性，一般情况下定位速度要比 XPath 快，但对于初学者来说学习起来稍微有点难度，下面我们就详细地介绍 CSS 的语法与使用。

CSS 选择器的常见语法如表 4.1 所示。

表 4.1　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 id="firstname"的所有元素
*	*	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 的所有 <input> 元素
element + element	div + input	选择同一级中紧接在 元素之后的所有 <input> 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>
……
```

1）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。

2）通过 id 属性定位：

```
find_element_by_css_selector("#kw")

find_element_by_css_selector("#su")
```

3）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找到想要的元素。

1）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，它的标签名为 span，查找它的所有标签名叫 input 的子元素。

2）通过属性定位：

```
find_element_by_css_selector("[autocomplete=off]")
find_element_by_css_selector("[name='kw']")
find_element_by_css_selector('[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标识这个元素。对于属性值来说，可加引号，也可以不加，但注意和整个字符串的引号进行区分。

3）组合定位：

我们当然可以把上面的定位策略组合起来使用，这就大大加强了定位元素的唯一性。

```
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。

我们可以通过使用 Firebug 工具帮助我们生成 CSS 语法，生成方法与 XPath 相同，通过 Firebug 定位元素，在元素上右击，选择“复制 CSS 路径”，如图 4.5 所示。

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，如图 4.6 所示

图 4.6　通过 FirePath 生成 CSS 语法

需要说明的是，CSS 的语法远不止上面所介绍的内容，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如表 4.2 所示。

表 4.2 XPath 与 CSS 的类似功能对比

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 >h1

通过前面的学习我们了解到，XPath 和 CSS 都提供了非常强大而灵活的定位方法。相比较而言 CSS 语法更加简洁，但理解和使用的难度更大一点。根据笔者的经验，这两种定位方式我们只需掌握一种就可解决大部分定位问题，至于读者的选择就看个人喜好了。

对于 Web 自动化来说，学会元素的定位相当于自动化已经学会了一半，剩下的就是 WebDriver 中所提供的各种方法的使用，后面我们将通过大量的实例来介绍这些方法的具体使用。

4.1.9 用 By 定位元素

针对前面介绍的 8 种定位方法，WebDriver 还提供了另外一套写法，即统一调用 find_element()方法，通过 By 来声明定位的方法，并且传入对应定位方法的定位参数。具体如下：

```
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,"新闻")
find_element(By.PARTIAL_LINK_TEXT,"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，第一个参数是定位的类型，由 By 提供；第二个参数是定位的具体方式。在使用 By 之前需要将 By 类导入。

from selenium.webdriver.common.by import By

通过查看 WebDriver 的底层实现代码发现它们其实是一回事儿，例如，find_element_by_id()方法的实现：

webelement.py

```
.....
    def find_element_by_id(self, id_):
        """Finds element within this element's children by ID.

        :Args:
            - id_ - ID of child element to locate.
        """
        return self.find_element(by=By.ID, value=id_)
.....
```

但 WebDriver 更推荐前面介绍的写法。

4.2　控制浏览器

WebDriver 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的一些方法，例如控制浏览器的大小、操作浏览器前进和后退等。

4.2.1　控制浏览器窗口大小

有时候我们希望能以某种浏览器尺寸打开，让访问的页面在这种尺寸下运行。例如可以将浏览器设置成移动端大小(480* 800)，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

```
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()方法使打开的浏览器全屏显示，其用法与 set_window_size() 相同，但它不需要参数。

4.2.2 控制浏览器后退、前进

在使用浏览器浏览网页时，浏览器提供了后退和前进按钮，可以方便地在浏览过的网页之间切换，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

```
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()
```

为了看清脚本的执行过程，下面每操作一步都通过 print()来打印当前的 URL 地址，执行结果如下：

Python shell

```
========================= RESTART: D:/pyse/test.py =========================
now access http://www.baidu.com
now access http://news.baidu.com
back to  http://www.baidu.com
forward to  http://news.baidu.com
```

4.2.3　模拟浏览器刷新

有时候需要手动刷新（F5）页面。例如图 4.7 中的广播数，当发送一条广播之后，广播的数量不自动变化，需要手动刷新，广播数才会加 1。

图 4.7　发送广播功能

test.py

```
......
driver.refresh()  #刷新当前页面
......
```

4.3 简单元素操作

前面我们已经学习了定位元素，定位只是第一步，定位之后需要对这个元素进行操作，或单击（按钮）或输入（输入框），下面就来认识 WebDriver 中最常用的几个方法：

- clear()：　　清除文本。
- send_keys(*value)：　　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

```
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；但如果直接向输入框中输入数据，则可能会与输入框中的提示信息拼接。例如，本来用户输入的是“username”，但与提示信息拼接则变为“账号 username”，从而造成输入信息错误。这个时候可以先使用 clear()方法来清除输入框中的默认提示信息。

send_keys()方法模拟键盘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登录信息。当然，它的作用不仅于此，我们还可以用它发送键盘按键，甚至用它来模拟文件上传。

click()方法可以用来单击一个元素，前提是它是可以被单击的对象，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()方法不仅可用于单击一个按钮，它还能单击任何可以单击的文字/图片链接、复选框、单选框、下拉框等。

4.3.2　WebElement 接口常用方法

通常有趣的和需要与页面交互的方法都由 WebElement 接口提供，包括 4.1 节中所介绍的 8 种定位方法和上面介绍的 3 个方法。除此之外，WebElement 还提供了一些非常有用的方法，下面我们就来学习这些方法的使用。

submit()

submit()方法用于提交表单。例如，在搜索框输入关键字之后的“回车”操作，就可以通过 submit()方法模拟。

youdao.py

```
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()
```

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，同样达到单击“搜索”按钮的效果。有时候 submit()可以与 click()方法互换来使用，submit()同样可以提交一个按钮，但 submit()的应用范围远不及 click()广泛。

- 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

```
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 或其他任意属性
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()
```

输出结果

```
========================= RESTART: D:/pyse/baidu.py =========================
{'width': 500, 'height': 22}
©2015 Baidu 使用百度前必读 意见反馈 京 ICP 证 030173 号
text
True
```

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

当然，WebElement 接口还提供了其他方法，读者可以参考 WeBDriver API 官方文档学习。

4.4　鼠标事件

通过前面的例子了解到，可以使用 click()来模拟鼠标的单击操作，现在的 Web 产品中提供了更丰富的鼠标交互方式，例如鼠标右击、双击、悬停、甚至是鼠标拖动等功能。在 WebDriver 中，将这些关于鼠标操作的方法封装在 ActionChains 类提供。

ActionChains 类提供了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。

由网盘所提供的的右键快捷菜单功能如图 4.8 所示。

图 4.8　网盘右键快捷菜单

1. 鼠标右击操作

对于 ActionChains 类所提供的鼠标方法与前面学过的 click()方法的用法有所不同，如图 4.8 所示，对 360 网盘右键快捷菜单的操作如下。

yunpan.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
  导入提供鼠标操作的 ActionChains 类。
- ActionChains(driver)
  调用 ActionChains()类，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操作，在调用时需要指定元素定位。
- perform()
  执行所有 ActionChains 中存储的行为，可以理解成是对整个操作的提交动作。

2. 鼠标悬停

鼠标悬停弹出下拉菜单也是一个十分常见的功能设计，如图 4.9 所示。

图 4.9 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。

mouse.py

```
# ……

# 定位到要悬停的元素
```

```
above = driver.find_element_by_id("id")
# 对定位到的元素执行悬停操作
ActionChains(driver).move_to_element(above).perform()
# ……
```

3. 鼠标双击操作

double_click 方法用于模拟鼠标双击操作。

mouse.py

```
# ……

# 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()
# ……
```

4. 鼠标拖放操作

drag_and_drop(source, target)在源元素上按住鼠标左键，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。

mouse.py

```
# ……

# 定位元素的原位置
element = driver.find_element_by_id("xx")
# 定位元素要移动到的目标位置
target = driver.find_element_by_id("xx")

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

4.5 键盘事件

Keys()类提供了键盘上几乎所有按键的方法。前面了解到，send_keys()方法可以用来模拟键盘输入，除此之外，我们还可以用它来输入键盘上的按键，甚至是组合键，如 Ctrl+A、Ctrl+C 等。

baidu.py

```
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("seleniumm")

# 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("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')

# 通过回车键来代替单击操作
driver.find_element_by_id("su").send_keys(Keys.ENTER)

driver.quit()
```

需要说明的是，上面的脚本没有什么实际意义，仅向我们展示模拟键盘各种按键与组

合键的用法。

from selenium.webdriver.common.keys import Keys

在使用键盘按键方法前需要先导入 keys 类。

以下为常用的键盘操作：

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

在编写功能测试用例时，会假定一个预期结果，在执行用例的过程中把得到的实际结果与预期结果进行比较，从而判断用户的通过或失败。自动化测试用例是由机器去执行的，通常机器并不像人一样有思维和判断能力，那么是不是模拟各种操作页面的动作没有报错就说明用例执行成功呢？并非如此，例如我们模拟百度搜索的用例，当新的迭代版本上线后，每一页的搜索结果少一条，但用例的执行不会报错，因此这个 bug 永远不会被自动化测试发现。

那么是不是在运行自动化测试用例时需要由测试人员盯着用例的执行来辨别执行结果呢？如果是这样的话，自动化测试就失去了“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 方法在前面已经讲过，它用于获取标签对之间的文本信息。

下面仍以 126 邮箱为例，介绍如何获取这些信息。

login126.py

```
from selenium import webdriver
import time

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()
time.sleep(5)

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()
```

运行脚本后执行结果如下。

Python Shell

```
======================== RESTART: D:/pyse/login126.py ========================
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/js6/main.jsp?sid=VBqseScEOCvclcjRdjEEYbWiQFuWVamg&df=ma
il126_letter#module=welcome.WelcomeModule%7C%7B%7D
username@126.com
```

title：用于获得当前页面的标题。

current_url：用户获得当前页面的 URL。

通过打印结果，我们发现登录前后的 title 和 URL 明显不同。我们可以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，是不能拿来做验证信息的。title 可以拿来做验证信息，但它并不能明确地表示是哪个用户登录成功了，因此通过 text 获取用户文本（username@126.com）是很好的验证信息。

4.7　设置元素等待

如今大多数 Web 应用程序使用 AJAX 技术。当浏览器在加载页面时，页面上的元素可能并不是同时被加载完成的，这给元素的定位增加了困难。如果因为在加载某个元素时延迟而造成 ElementNotVisibleException 的情况出现，那么就会降低自动化脚本的稳定性，我们可以通过设置元素等待改善这种问题造成的不稳定。

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 WebdDriver 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

```
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 类是由 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 ：浏览器驱动。

timeout ：最长超时时间，默认以秒为单位。

poll_frequency ：检测的间隔（步长）时间，默认为 0.5S。

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

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 True。
- 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。

在本例中，通过 as 关键字将 expected_conditions 重命名为 EC，并调用 presence_of_element_located()方法判断元素是否存在。

expected_conditions 类所提供的预期条件判断的方法如表 4.3 所示。

表 4.3　expected_conditions 类提供的预期条件判断的方法

方　法	说　明
title_is	判断当前页面的标题是否等于预期
title_contains	判断当前页面的标题是否包含预期字符串
presence_of_element_located	判断元素是否被加在 DOM 树里，并不代表该元素一定可见
visibility_of_element_located	判断元素是否可见（可见代表元素非隐藏，并且元素的宽和高都不等于 0）
visibility_of	与上一个方法作用相同，只是上一个方法参数为定位，该方法接收的参数为定位后的元素
presence_of_all_elements_located	判断是否至少有一个元素存在于 DOM 树中。例如，在个页面中有 n 个元素的 class 为“wp”，那么只要有一个存在就返回 True
text_to_be_present_in_element	判断某个元素中的 text 是否包含了预期的字符串
text_to_be_present_in_element_value	判断某个元素的 value 属性是否包含了预期的字符串
frame_to_be_available_and_switch_to_it	判断该表单是否可以切换进去，如果可以，返回 True 并且 switch 进去，否则返回 False
invisibility_of_element_located	判断某个元素是否不存在于 DOM 树或不可见
element_to_be_clickable	判断元素是否可见并且是可以点击的
staleness_of	等到一个元素从 DOM 树中移除
element_to_be_selected	判断某个元素是否被选中，一般用在下拉列表
element_selection_state_to_be	判断某个元素的选中状态是否符合预期
element_located_selection_state_to_be	与上一个方法作用相同，只是上一个方法参数为定位后的元素，该方法接收的参数为定位
alert_is_present	判断页面上是否存在 alert

除 expected_conditions 所提供的丰富的预期条件判断方法外，还可以使用前面学过的 is_displayed()方法来判断元素是否可见。

baidu.py

```
from selenium import webdriver
from time import sleep, ctime

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

print(ctime())
for i in range(10):
    try:
        el = driver.find_element_by_id("kw22")
        if el.is_displayed():
```

```
            break
    except: pass
    sleep(1)
else:
    print("time out")
driver.close()
print(ctime())
```

相对来说，这种方式更容易理解，通过 for 循环 10 次，每次循环判断元素的 is_displayed() 状态是否为 True：如果为 True，则 break 跳出循环；否则 sleep(1)后继续循环判断，直到 10 次循环结束后，打印“time out”信息。

执行结果如下：

Python Shell

```
========================= RESTART: D:/pyse/baidu.py =========================
Fri Oct 23 22:51:25 2015
time out
Fri Oct 23 22:51:35 2015
```

4.7.2 隐式等待

隐式等待是通过一定的时长等待页面上某元素加载完成。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得多。

baidu.py

```
from selenium import webdriver
from selenium.common.exceptions import NoSuchElementException
from time import ctime

driver = webdriver.Firefox()

# 设置隐式等待为 10 秒
driver.implicitly_wait(10)
driver.get("http://www.baidu.com")

try:
```

```
    print(ctime())
    driver.find_element_by_id("kw22").send_keys('selenium')
except NoSuchElementException as e:
    print(e)
finally:
    print(ctime())
    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，它并不针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，如果元素可以定位，则继续执行；如果元素定位不到，则它将以轮询的方式不断地判断元素是否被定位到。假设在第 6 秒定位到了元素则继续执行，若直到超出设置时长（10 秒）还没有定位到元素，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，通过打印的两个时间可以看出，当执行对百度输入框的操作时，超过了 10 秒的等待。

Python Shell

```
========================= RESTART: D:/pyse/baidu.py =========================
Fri Oct 23 22:57:05 2015
Message: Unable to locate element: {"method":"id","selector":"kw22"}
Stacktrace:
    at FirefoxDriver.prototype.findElementInternal_
(file:///C:/Users/fnngj/AppData/Local/Temp/tmpwi0luzkz/extensions/fxdriver@
googlecode.com/components/driver-component.js:10659)
    at fxdriver.Timer.prototype.setTimeout/<.notify
(file:///C:/Users/fnngj/AppData/Local/Temp/tmpwi0luzkz/extensions/fxdriver@
googlecode.com/components/driver-component.js:621)
Fri Oct 23 22:57:15 2015
```

4.7.3　sleep 休眠方法

有时候我们希望脚本在执行到某一位置时做固定时间的休眠，尤其是在脚本调试过程中。这时可以使用 sleep()方法，需要说明的是，sleep()方法由 Python 的 time 模块提供。

baidu.py

```
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()方法时会固定休眠一定的时长，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，则可以用小数表示，如 sleep(0.5)表示休眠 0.5 秒。

4.8 定位一组元素

在 4.1 节我们已经学习了 8 种定位方法，这 8 种定位方法是针对单个元素定位的，WebDriver 还提供了与之对应的 8 种用于定位一组元素的方法。

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()

定位一组元素的方法与定位单个元素的方法类似，唯一的区别是在单词 element 后面多了一个 s 表示复数。定位一组元素一般用于以下场景：

- 批量操作元素，例如勾选页面上所有的复选框。
- 先获取一组元素，再从这组对象中过滤出需要操作的元素。例如定位出页面上所

有的 checkbox，然后选择其中的一个进行操作。

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 页面，为了使页面更美观，在代码中添加了 Bootstrap 样式的引用。用浏览器打开后，效果如图 4.10 所示。

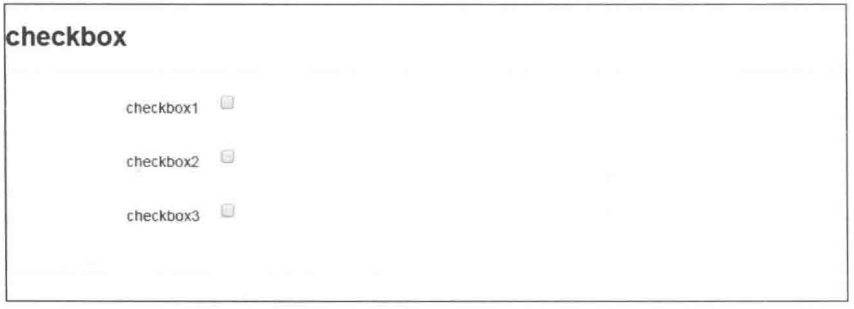

图 4.10 复选框

下面就通过例子来操作页面上的这一组复选框。

checkbox.py

```
from selenium import webdriver
import os,time

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 tpye 为 checkbox 的元素，单击勾选
for i in inputs:
    if i.get_attribute('type') == 'checkbox':
        i.click()
        time.sleep(1)

driver.quit()
```

前面提到，通过 tag name 的定位方式很难定位到单个元素，因为元素标签名重名的概率很高，因而在定位一组元素时，这种方式就派上用场了。在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，在遍历过程中，通过 get_attribute()方法获取元素的 type 属性是否为“checkbox”，如果为“checkbox”，就认为这个元素是一个复选框，对其进行勾选操作。

需要注意的是，在上面的例子中，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，我们还可以使用 XPath 或 CSS 来直接判断属性值，从而进行单击操作。

checkbox.py

```
from selenium import webdriver
import os,time

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()
    time.sleep(1)

# 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 的钩给去掉
driver.find_elements_by_css_selector('input[type=checkbox]').pop().click()

driver.quit()
```

通过 XPath 或 CSS 来查找一组元素时，省去了判断步骤。因为定位方法已经做了判断，只需循环对这一组元素进行勾选即可。

除此之外，例子中还用到了 Python 所提供的两个有趣的方法。len()方法可以用来计算元素的个数，通过 print()打印出计算的结果。pop()方法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。因为前面的循环已经将所有复选框都勾选上了，再对这一组元素执行 pop().lick()，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该如何操作呢？

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任意一个元素了，只需数一数需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/iframe 表单嵌套页面的应用，WebDriver 只能在一个页面上对元素识别与定位，对于 frame/iframe 表单内嵌页面上的元素无法直接定位。这时就需要通过 switch_to.frame()方法将当前定位的主体切换为 frame/iframe 表单的内嵌页面中。

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，用浏览器打开后如图 4.11 所示。

frame

图 4.11　iframe 嵌入百度首页

这时候直接定位页面上的百度的输入框一定会报错：找不到元素。因此可以使用 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

```
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。如果 iframe 没有可用的 id 和 name 属性，则可以通过下面的方式进行定位。

frame.py

```
……
#先通过 xpth 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.parent_frame()
```

如果完成了在当前表单上的操作，则可以通过 switch_to.parent_content()方法跳出当前一级表单。该方法默认对应于离它最近的 switch_to.frame()方法。除此之外，在进入多级表单的情况下，还可以通过 switch_to.default_content()跳回最外层的页面。

4.10 多窗口切换

在页面操作过程中有时候点击某个链接会弹出新的窗口，这时就需要主机切换到新打开的窗口上进行操作。WebDriver 提供了 switch_to.window()方法，可以实现在不同的窗口之间切换。

以百度首页和百度注册页为例，在两个窗口之间的切换如图 4.12 所示。

图 4.12 多窗口

windows.py

```
from selenium import webdriver
import time

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('登录').click()
driver.find_element_by_link_text("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')
        time.sleep(2)
        # ……


# 回到搜索窗口
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(2)

driver.quit()
```

脚本的执行过程：首先打开百度首页，通过 current_window_handle 获得当前窗口的句柄，并赋值给变量 sreach_handle。接着打开登录弹窗，在登录弹窗上单击“立即注册”，从而打开新的注册窗口。通过 window_handles 获得当前打开的所有窗口的句柄，并赋值给变量 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，因为脚本执行过程中只打开了两个窗口。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，然后进行搜索操作。

在本例中所涉及的新方法如下：

current_window_handle：获得当前窗口句柄。

window_handles：返回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()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 十分简单，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 等方法进行操作。

- text：返回 alert/confirm/prompt 中的文字信息。
- accept()：接受现有警告框。
- dismiss()：解散现有警告框。
- send_keys(keysToSend)：发送文本至警告框。keysToSend：将文本发送至警告框。

如图 4.13 所示，百度搜索设置弹出的窗口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。

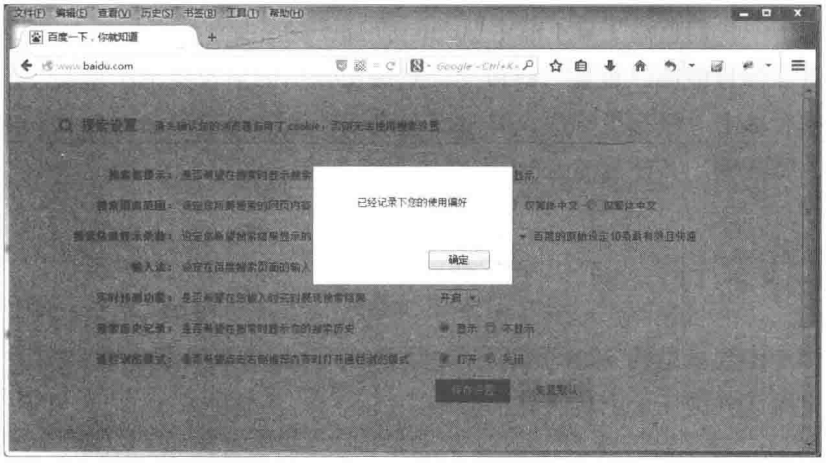

图 4.13　百度搜索保存设置弹窗

alert_.py

```
from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains
import time

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

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

# 打开搜索设置
driver.find_element_by_link_text("搜索设置").click()

# 保存设置
driver.find_element_by_class_name("prefpanelgo").click()
time.sleep(2)

# 接受警告框
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“设置”链接上，然后在弹出的下拉菜单中单击“搜索设置”按钮，设置完成后单击“保存设置”，弹出保存确认警告框。通过 switch_to_alert()方法获取当前页面上的警告框，并使用 accept()方法接受警告框。

4.12 上传文件

上传文件是比较常见的 Web 功能之一，但 WebDriver 并没有提供专门用于上传的方法，如何实现上传操作关键在于上传文件的思路。

一般 Web 页面的上传功能的操作需要单击“上传”按钮后打开本地的 Window 窗口，从窗口中选择本地文件进行上传。而 WebDriver 是无法操作 Windows 控件的，所以，对于初学者来说，一般思路会卡在如何识别 Window 控件这个问题上。

对于 Web 页面的上传功能实现一般有以下两种方式。

- 普通上传：普通的附件上传是将本地文件的路径作为一个值放在 input 标签中，通过 form 表单将这个值提交给服务器。
- 插件上传：一般是指基于 Flash、JavaScript 或 Ajax 等技术所实现的上传功能。

4.12.1 send_keys 实现上传

对于通过 input 标签实现的上传功能，可以将其看作是一个输入框，即通过 send_keys()指定本地文件路径的方式实现文件上传。

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>
```

通过浏览器打开 upfile.html 文件，效果如图 4.14 所示

图 4.14　普通上传功能

upfile.py

```
from selenium import webdriver
import os

driver = webdriver.Firefox()
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，就避免了操作 Windows 控件的步骤。如果能找到上传的 input 标签，那么基本上就可以通过 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 目前最新版本是 v3，它是一个使用类似 BASIC 脚本语言的免费软件，它被设计用来进行 Windows GUI(图形用户界面)的自动化测试。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，安装完成后，在菜单中会看到如图 4.15 所示的 Auto Lt 菜单：

AutoIt Windows Info：　　用于识别 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 上传过程。

1．首先打开 AutoIt Windows Info 工具，用鼠标单击 Finder Tool，鼠标将变成一个小风扇形状的图标，如图 4.16 所示。按住鼠标左键，将其拖动到需要识别的控件上（如“打开”按钮控件），如图 4.17 所示。

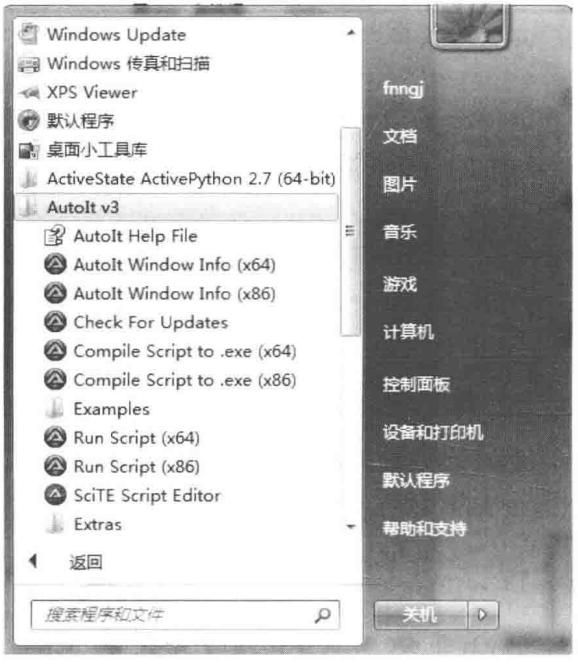

图 4.15　AutoIt 菜单

图 4.16　AutoIt Windows Info 识别“文件名”输入框控件

图 4.17　AutoIt Windows Info 识别“打开”按钮控件

如图 4.16 和图 4.17 所示，可以通过 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 AutoIt 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()方法设置 10 秒钟用于等待窗口的显示。ControlSetText()方法用于向“文件名”输入框内输入本地上传文件的路径。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()用于单击上传窗口中的“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）来运行脚本。

注意：在运行时文件上传窗口应处于打开状态。

3．脚本运行正常，将其保存为 upfile.au3 文件。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为 exe 程序。打开 Compile Script to.exe 工具，将其生成为 exe 可执行文件，如图 4.18 所示。

图 4.18　Compile Script to.exe 生成 exe 程序

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

4．接下来通过自动化测试脚本调用 upfile.exe 程序，实现上传。

upfile.py

```
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)

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

driver.quit()
```

通过 system()方法可以调用并执行 upfile.exe 程序。

虽然这种方式可以解决文件上传（或文件下载）的操作问题，但笔者不太推荐这种解决方案，因为通过 Python 调用的 exe 程序并不在 Python 的可控范围内。换句话说，exe 执行多长时间，执行是否出错，Python 程序都无法得知。

4.13 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放到设置的目录中。下面以 Firefox 浏览器为例，执行文件的下载。

downfile.py

```
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 浏览器能实现文件下载，我们需要通过 FirefoxProfile()对其做一些设置。

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，Flase 为不显示。

browser.download.dir

用于指定所下载文件的目录。os.getcwd() 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 4.19 所示。

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。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。

上面例子中的设置只针对 Firefox 浏览器，不同的浏览器设置方法会有所不同。通用的方法还是借助 AutoIt 来操作 Windows 控件进行下载，由于 4.13 节已经详细地讲解了 AutoIt 的使用，因而这里不再重复介绍。

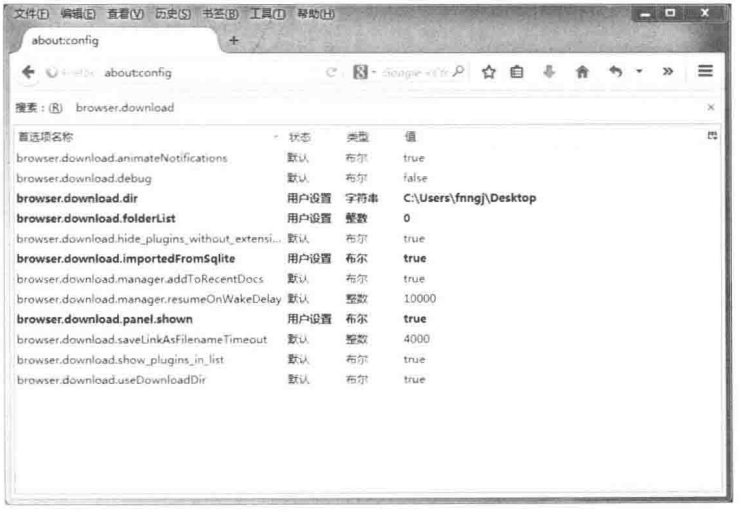

图 4.19　Firefox 参数设置

4.14　操作 Cookie

有时候我们需要验证浏览器中 cookie 是否正确，因为基于真实 cookie 的测试是无法通过白盒和集成测试的。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：

- get_cookies()：获得所有 cookie 信息。
- get_cookie(name)：返回字典的 key 为“name”的 cookie 信息。
- add_cookie(cookie_dict)：添加 cookie。“cookie_dict”指字典对象，必须有 name 和 value 值。
- delete_cookie(name,optionsString)：删除 cookie 信息。“name”是要删除的 cookie 的名称，“optionsString”是该 cookie 的选项，目前支持的选项包括“路径”，“域”。
- 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

```
from selenium import webdriver

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

输出结果：

```
======================== RESTART: D:/pyse/cookie.py ========================
[{'httpOnly': False, 'name': 'YOUDAO_MOBILE_ACCESS_TYPE', 'secure': False,
'expiry': 1477324093, 'value': '1', 'path': '/', 'domain': '.youdao.com'},
{'httpOnly': False, 'name': 'OUTFOX_SEARCH_USER_ID', 'secure': False, 'expiry':
2391868093, 'value': '2045063256@112.90.37.235', 'path': '/', 'domain':
'.youdao.com'}, {'httpOnly': False, 'name': 'JSESSIONID', 'secure': False,
'expiry': None, 'value': 'abcWVA6WwVSR2bIW4RFcv', 'path': '/', 'domain':
'www.youdao.com'}, {'httpOnly': False, 'name': 'CNZZDATA1256118513', 'secure':
False, 'expiry': 1461512894, 'value': '1183881050-1445787498-%7C1445787498',
'path': '/', 'domain': 'www.youdao.com'}, {'httpOnly': False, 'name':
```

```
'lzstat_uv', 'secure': False, 'expiry': 1761407294, 'value':
'13013917332303058731|3601912', 'path': '/', 'domain': '.youdao.com'},
{'httpOnly': False, 'name': 'lzstat_ss', 'secure': False, 'expiry': None,
'value': '449876553_0_1445816894_3601912', 'path': '/', 'domain':
'.youdao.com'}]
```

从执行结果可以看出，cookie 数据是以字典的形式进行存放的。知道了 cookie 的存放形式，接下来我们就可以按照这种形式向浏览器中写入 cookie 信息。

cookie.py

```
from selenium import webdriver

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

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

# 遍历 cookies 中的 name 和 value 信息并打印，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()


输出结果：
======================== RESTART: D:/pyse/cookie.py ========================
YOUDAO_MOBILE_ACCESS_TYPE -> 1
_PREF_ANONYUSER__MYTH -> aGFzbG9nZ2VkPXRydWU=
OUTFOX_SEARCH_USER_ID -> -1046383847@218.17.158.115
JSESSIONID -> abc7qSE_SBGsVgnVLBvcu
key-aaaaaaa -> value-bbbbbb
```

从执行结果可以看到，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。通过遍历得到所有的 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 value。如果找不到 username 或对应的 value 为空，那么说明 cookie 没有成功地 保存到浏览器中。

delete_cookie() 和 delete_all_cookies() 的使用也很简单，前者通过 name 删除一个特定的 cookie 信息，后者直接删除浏览器中的所有 cookies()信息。

4.15 调用 JavaScript

虽然 WebDriver 提供了操作浏览器的前进和后退方法，但对于浏览器滚动条并没有提供相应的操作方法。在这种情况下，就可以借助 JavaScript 来控制浏览器的滚动条。WebDriver 提供了 execute_script()方法来执行 JavaScript 代码。

一般我们想到的必须使用滚动条的场景是：注册时的法律条文的阅读。判断用户是否阅读完的标准是：滚动条是否拉到页面底部。当然，有时候为了使操作更接近用户行为也会使用滚动条，例如用户要操作的元素在页面的第二屏，一般用户不会对看不到的元素进行操作，那么就需要先将滚动条拖动到页面的第二屏再进行操作。

用于调整浏览器滚动条位置的 JavaScript 代码如下：

html

```
……
<!-- window.scrollTo(左边距,上边距); -->
window.scrollTo(0,450);
……
```

window.scrollTop()方法用于设置浏览器窗口滚动条的水平和垂直位置。方法的第一个参数表示水平的左间距，第二个参数表示垂直的上边距。其代码如下：

baidu.py

```
from selenium import webdriver
from time import sleep

# 访问百度
driver=webdriver.Firefox()
driver.get("http://www.baidu.com")
```

```
# 设置浏览器窗口大小
driver.set_window_size(600, 600)

# 搜索
driver.find_element_by_id("kw").send_keys("selenium")
driver.find_element_by_id("su").click()
sleep(2)

# 通过 javascript 设置浏览器窗口的滚动条位置
js="window.scrollTo(100,450);"
driver.execute_script(js)
sleep(3)

driver.quit()
```

通过浏览器打开百度进行搜索，并且提前通过 set_window_size()方法将浏览器窗口设置为固定宽高显示，目的是让窗口出现水平和垂直滚动条。然后通过 execute_script()方法执行 JavaScripts 代码来移动滚动条的位置，如图 4.20 所示。

图 4.20　通过 JavaScript 控制浏览器滚动条位置

当然，JavaScript 的作用不仅仅体现在浏览器滚动条的操作上，还可以用它向页面中 textarea 文本框输入内容，如图 4.21 所示。

图 4.21　富文本框

文本框的前端代码如下：

html

```
……
<textarea id="id" style="width: 98%" cols="50" rows="5" class="txtarea">
</textarea>
……
```

虽然我们可以通过 id 的方式将其进行定位，但却不能通过 send_keys()向文本框中输入文本信息。这种情况下，就需要借助 JavaScript 代码完成输入。

js_test.py

```
……
text = "input text"
js = "var sum=document.getElementById('id'); sum.value='" + text + "';"
driver.execute_script(js)
……
```

首先定义了要输入的内容 text，然后将 text 与 JavaScript 代码通过“+”进行拼接。这样做的目的是为了使输入内容变得可自定义。最后，通过 execute_script()执行 JavaScript 代码。

4.16　处理 HTML5 的视频播放

目前 HTML5 技术已渐渐成为主流，主流的浏览器都已支持 HTML5。越来越多的应用使用了 HTML5 的元素，如 canvas、video 等，另外网页存储功能更增加了用户的网络体验，使得越来越多的开发者在使用这样的标准，所以我们也需要学习如何使用自动化技术来测试它们。

WebDriver 支持在指定的浏览器上测试 HTML5，另外，我们还可以使用 JavaScript 来测试这些功能，这样就可以在任何浏览器上测试 HTML5 了。

大多数浏览器使用控件（如 Flash）来播放视频，但是，不同的浏览器需要使用不同的插件。HTML5 定义了一个新的元素<video>，指定了一个标准的方式来嵌入电影片段，如图 4.22 所示，IE9+、Firefox、Opera、Chrome 都支持该元素。

在本节中，我们将探索如何自动化测试<video>，该元素提供了 JavaScript 接口和多种方法及属性。

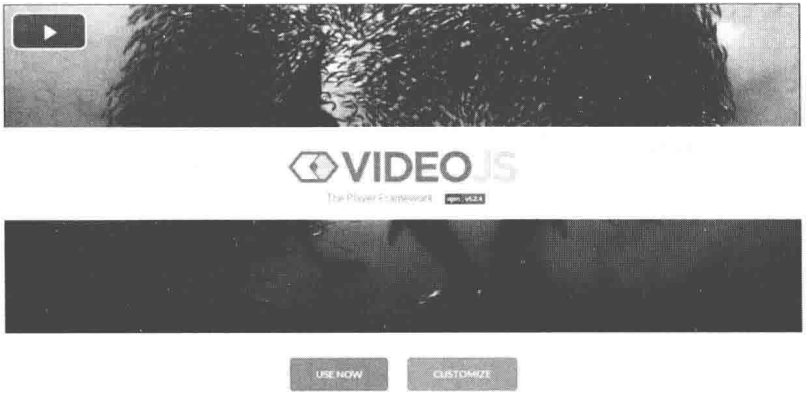

图 4.22　HTML5 Video Player

test_video.py

```
from selenium import webdriver
```

```
from time import sleep

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

video = driver.find_element_by_xpath("body/Setion[1]/div/video")

# 返回播放文件地址
url = driver.execute_script("return arguments[0].currentSrc;", video)
print(url)

# 播放视频
print("start")
driver.execute_script("return arguments[0].play()", video)

# 播放 15 秒钟
sleep(15)

# 暂停视频
print("stop")
driver.execute_script("arguments[0].pause()", video)

driver.quit()
```

JavaScript 函数有个内置的对象叫做 arguments。argument 对象包含了函数调用的参数数组，[0]表示取对象的第 1 个值。

currentSrc 熟悉返回当前音频/视频的 URL。如果未设置音频/视频，则返回空字符串。

load()、play()、pause() 等控制着视频的加载、播放和暂停。

4.17 窗口截图

自动化用例是由程序去执行的，因此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能对当前窗口截图保存，那么通过图片就可以非常直观地看出出错的原因。WebDriver 提供了截图函数 get_screenshot_as_file()来截取当前窗口。

baidu.py

```
from selenium import webdriver
```

```
from time import sleep

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

driver.find_element_by_id('kw').send_keys('selenium')
driver.find_element_by_id('su').click()
sleep(2)

# 截取当前窗口，并指定截图图片的保存位置
driver.get_screenshot_as_file("D:\\pyse\\baidu_img.jpg")

driver.quit()
```

脚本运行完成后打开 D 盘，就可以找到 baidu_error.jpg 图片文件了。

4.18　关闭窗口

在前面的例子中我们一直使用 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，用来关闭当前窗口。例如 4.10 节多窗口的处理，在用例执行的过程中打开了多个窗口，我们想要关闭其中的某个窗口，这时就要用到 close()方法进行关闭了。

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。验证码的类型很多，有字母数字的、有汉字的，甚至还有需要用户输入一道算术题的答案的。对于系统来说，使用验证码可以有效地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，这里笔者根据自己的经验来谈谈处理验证码的几种常见方法。

1. 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，这样做可省去测试人员不少的麻烦；但如果自动化脚本是在正式环境测试，那么这种做法就给系统带来了一定的风险。

2. 设置万能验证码

去掉验证码的主要问题是安全，为了应对在线系统的安全威胁，可以在修改程序时不取消验证码，而是在程序中留一个“后门”，即设置一个“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，否则就判断用户输入的验证码是否正确。

设计万能验证码的方式非常简单，只需对用户的输入信息多加一个逻辑判断，下面通过例子演示。

number.py

```
from random import randint

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

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

if number == verify:
    print("登录成功!!")
elif number == 132741:
    print("登录成功!!")
else:
    print("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、万能验证码和错误的验证码，执行结果如下：

Python Shell

```
======================== RESTART: D:/pyse/number.py ========================
```

```
生成的随机数:8396
请输入随机数:8396
8396
登录成功!!
======================== RESTART: D:/pyse/number.py ========================
生成的随机数:5113
请输入随机数:132741
132741
登录成功!!
======================== RESTART: D:/pyse/number.py ========================
生成的随机数:1996
请输入随机数:1234
1234
验证码输入有误!
```

3. 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，大多验证码识别技术，识别率都很难达到 100% 。

4. 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。例如我们在第一次登录某网站时勾选“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。这样自然就绕过了验证码问题。这个“记住密码”的功能其实就记录在了浏览器的 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 ，当再次访问网站时，服务器将直接读取浏览器的 cookie 进行登录。

cookie_login.py

```
# ……
# 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_UserNumber', '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/")
# ……

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，并传输入对应的登录信息。可以用 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，更直接的方式是询问开发人员。

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 发送请求并做出响应。

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，例如打开浏览器，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的，也就是 Remote Server。Remote Server 接受请求，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。启动的浏览器实例将作为 WebDriver 的 Remote Server。

② Client 端通过 CommandExcuter 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）来转化浏览器的 native 调用。

Python 提供了 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。开启 debug 模块，就可以捕捉到客户端向服务器发送的请求。

baidu.py

```
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()
```

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发送的 POST 请求，而无法获取服务器所返回的应答信息。我们在后面的章节中将会学习 Selenium Server，通过 Selenium Server 可以获取到更详细的请求与应答信息。

Python Shell

```
======================== RESTART: D:/pyse/baidu.py ========================
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/ele
ment {"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/ele
ment/{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/ele
ment {"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/ele
ment/{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
```

本章小结

本章用了 21 个小节来讲元素的定位与操作，主要介绍了如何利用 WebDriver 提供的方法对 Web 页面上的常见功能进行操作。

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，所遇到的最多的问题仍然是元素的定位与操作。有时元素定位非常简单，例如，我们只要知道这个元素的 id 和 name 就可以轻松地定位到它们。有时元素的定位又非常的令人头疼，尽管我们试尽了各种办法，但仍无法定位到它们。在这里，笔者也没有万能的方法来帮助读者解决实际中的问题。

因此，在项目进行 UI 自动化评估的时候，页面元素的定位难度也是评估的标准之一，如果处处都是很难定位的元素，那么无疑会增加自动化测试的开发成本，这时候就需要考虑是否将更多的自动化测试放在单元或接口层来进行。

对于自动化测试人员来说，如果精通前端技术将会非常有助于自动化用例的编写，熟练使用 XPath 和 CSS 技术会使你的定位变得容易很多，精通 JavaScript、jQuery 可以让我们有更多的方式去操作 Web 页面。

在我们尝试开展自动化的 Web 项目中，大多数产品在设计初期并没有考虑是否易于自动化测试的进行，更多的会以实现功能为目标，这也是到后期开展自动化测试困难重重的

原因之一。如果开发人员在设计代码的时候就考虑是否容易自动化，为必要的元素加上规范的 id 和 name 属性的话，那么我们的自动化工作也会变得轻松很多。

测试人员要想更顺利地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，我们要清楚地认识到，自动化技术的应用与实践不是一个人的战斗，需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，这就需要读者去综合评估目前的项目是否真的适合引入自动化测试，或者目前的阶段是否真的迫切需要开展自动化测试。

假如，你已经动手开始进行自动化了，那么笔者再提几点建议：

1．熟练掌握 XPath\CSS 定位的使用，这样在遇到各种难以定位的问题时才不会变得束手无策。

2．准备一份 WebDriver API 文档，以便随时查阅 WebDriver 所提供的方法。

3．学习掌握 JavaScript、jQuery 技术，它可以让我们使用该技术去操作 Web 页面。

第 5 章
自动化测试模型

在介绍自动化测试模型之前，我们试着来解释自动化测试库、框架和工具之间的区别。

库的英文单词叫 Library，库是由代码集合成的一个产品，供程序员调用。面向对象的代码组织形成的库叫类库，面向过程的代码组织形成的库叫函数库。所以从这个角度来看，我们在第 4 章介绍的 WebDriver 就属于库的范畴，因为它提供了一组操作 Web 页面的类与方法，所以，我们可以称它为 Web 自动化测试库。

框架的英文单词叫 Framework，框架是为解决一个或一类问题而开发的产品，用户一般只需要使用框架提供的类或函数，即可实现全部功能。所以从这个角度来理解 unittest 框架，它主要用于实现测试用例的组织和执行，以及测试结果的生成。因为它的主要任务就是帮助我们完成测试工作，所以我们通常把它叫做单元测试框架。

工具的英文单词叫 Tools，在笔者看来工具与框架所做的事情类似，只是工具会有更高的抽象，屏蔽了底层的代码，一般会提供单独的操作界面供用户操作。例如，Selenium IDE 和 QTP 就是自动化测试工具。

回到自动化测试模型的概念上，笔者认为自动化测试模型可以看作自动化测试框架与工具设计的思想。随着自动化测试技术的发展，演化为以下几种模型：线性测试、模块化驱动测试、数据驱动测试和关键字驱动测试。

5.1　自动化测试模型介绍

下面分别介绍这几种自动化测试模型的特点。

5.1.1　线性测试

通过录制或编写对应用程序的操作步骤产生相应的线性脚本，每个测试脚本相对独立，且不产生其他依赖与调用，这也是早期自动化测试的一种形式：它们其实就是单纯的来模拟用户完整的操作场景。本书第 4 章所编写的测试脚本就属于线性测试，如图 5.1 所示。

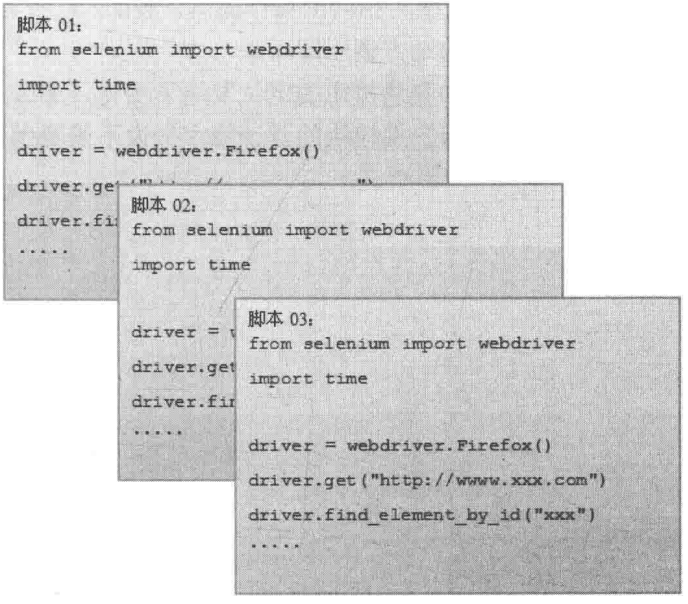

图 5.1　线性测试结构

这种模型的优势就是每一个脚本都是完整且独立的。所以，任何一个测试用例脚本拿出来都可以单独执行。当然，缺点也相当明显，测试用例的开发与维护成本很高：

- 开发成本很高，测试用例之间可能会存在重复的操作，不得不为每一个用例去录制或编写这些重复的操作。例如每个用例中重复的用户登录和退出操作等。
- 维护成本很高，正是因为测试用例之间存着重复的操作，所以当这些重复的操作发生改变时，就需要逐一地对它们进行修改。例如登录输入框的定位发生了改变，就需要对每一个包含登录的用例进行调整。

5.1.2 模块化驱动测试

正是由于线性测试的缺陷非常明显，因此早期的自动化测试专家就考虑用新的自动化测试模型来代替线性测试。做法也很简单，借鉴了编程语言中模块化的思想，把重复的操作独立成公共模块，当用例执行过程中需要用到这一模块操作时则被调用，这样就最大限度地消除了重复，从而提高测试用例的可维护性。

如图 5.2 所示，（需要说明的是，早期的自动化测试以工具为主，而非图 5.2 所示的代码形式。）模块化的结构很好地解决了线性结构的两个问题：

- 提高了开发效率，不用重复编写相同的操作脚本。假如，已经写好一个登录模块，后续测试用例在需要登录的地方调用即可。
- 简化了维护的复杂性，假如登录按钮的定位发生了变化，那么只需修改登录模块的脚本即可，对于所有调用登录模块的测试脚本来说不需要做任何修改。

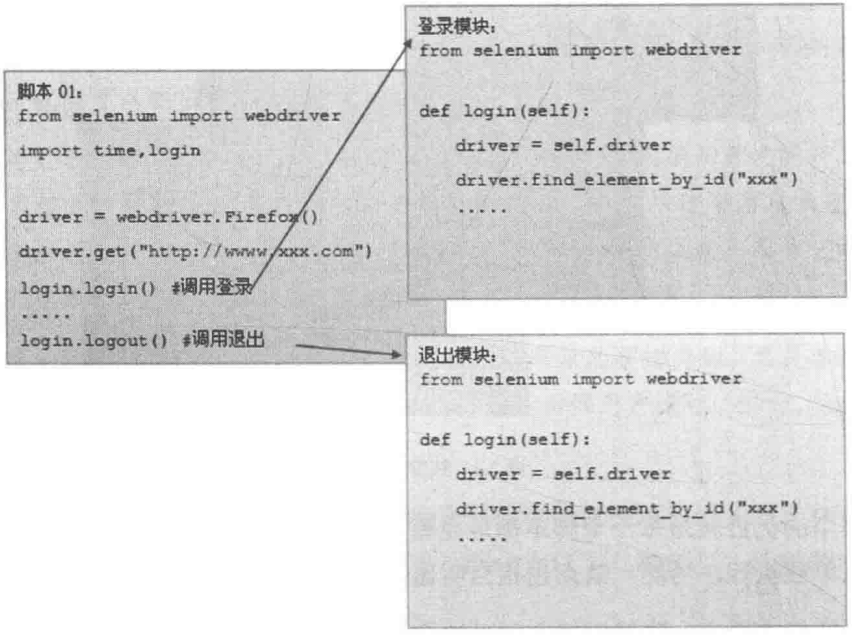

图 5.2 模块化结构

5.1.3 数据驱动测试

虽然模块化驱动测试很好地解决了脚本的重复问题，但是，自动化测试脚本在开发的

过程中还是发现了诸多不便。例如，现在我要测试不同用户的登录，首先用的是“张三”的用户名登录；下一个测试用例要换成“李四”的用户名登录。在这种情况下，还是需要重复地编写登录脚本，因为虽然登录的步骤相同，但是登录所用的测试数据不同。

于是，数据驱动测试的概念就为解决这类问题而被提出。从它的本意来解释，就是数据的改变从而驱动自动化测试的执行，最终引起测试结果的改变。这听上去的确是个高大上的概念，而在早期的商业自动化工具中，也的确把这一概念作为一个卖点。对于数据驱动所需要的测试数据，也是通过工具内置的 Datapool 管理。

如图 5.3 所示，数据驱动说的直白点就是数据的参数化，因为输入数据的不同从而引起输出结果的不同。

不管我们读取的是定义的数组、字典，或者是外部文件（excel、csv、txt、xml 等），都可以看作是数据驱动，它的目的就是实现数据与脚本的分离。

图 5.3　通过脚本读取数据文件

这样做的好处同样显而易见，它进一步增强了脚本的复用性。同样以登录为例，首先是重新设计登录模块，使其可以接收不同的数据，把接收到的数据作为登录操作的一部分。这样就可以很好地适应相同操作、不同数据的情况。当指定登录用户是“张三”时，那么登录之后的结果就是“欢迎张三”；当指定登录用户是“李四”时，登录结果就显示“欢

迎李四”。这就是数据驱动所希望达到的目的。

5.1.4 关键字驱动测试

理解了数据驱动后，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（RIDE）工具为主。这类工具封装了底层的代码，提供给用户独立的图形界面，以“填表格”的形式免除测试人员对写代码的恐惧，从而降低脚本的编写难度，我们只需使用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，Selenium 家族中的 Selenium IDE 也可以看作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 录制产生，它把每一个动作分为三部分：

- 做什么？例如打开、输入、点击等动作。
- 对谁做？通过定位方式找到要操作的对象。
- 如何做？例如输入框输入的内容为“selenium”等。

当然，关键字驱动技术也在不断发展和进步。下面以功能更为强大的关键字驱动测试框架 Robot Framework 为例，它也可以像编程一样写测试用例。

1. 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 小于或等于 5
...	ELSE	log	上面两个条件都不满足	

首先，定义 *a*、*b* 两个变量，分别赋值 2 和 5；然后，通过 run keyword if 关键字判断 *a*

是否大于或等于 1，如果满足条件则通过 log 输出“*a* 大于或等于 1”，否则继续判断 *b* 是否小于或等于 5，如果满足条件则通过 log 输出“*b* 小于或等于 5”。如果以上两个条件都不满足，则通过 log 输出“上面两个条件都不满足”。

2. 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 1 到 10，每循环一次都通过 log 输出 *i* 的值。

3. 读取外部文件

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

通过 Import Resource 关键字读取指定的外部文件。

4. import 引入外部类库

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

通过 Import Library 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；不过这样的用例同样需要较高的学习成本，与学习一门编程语言几乎相当。这样的框架越到后期越难维护，可靠性也会变差，关键字的用途与经验被局限在自己的框架内，你所学到的知识也很难重用到其他地方。所以，从测试人员的经验与技术积累价值来讲，笔者更倾向于直接通过编程的方式开发自动化脚本。

这里简单介绍了几种测试模型的发展过程与特点，虽然是从自动化测试模型的发展顺序逐一介绍的，但它们并非后者淘汰前者的关系。在实际自动化实施过程中，应以项目需求为出发点，综合运用上述模型来开展自动化测试。

5.2 模块化驱动测试实例

通过对自动化测试模型的介绍，我们了解了模块化设计的优点。本节我们就以具体的例子来介绍模块的具体应用，当然，使用它的基础是 Python 语言中函数与类方法的调用。下面以 126 邮箱为例。

mail126.py

```
from selenium import webdriver

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

# 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()

# 收信、写信、删除信件等操作
# ……

# 退出
driver.find_element_by_link_text("退出").click()
driver.quit()
```

从 126 邮箱业务流程分析，邮箱所提供的功能都需要登录之后进行，例如收信、写信、删除信件等操作。对于手工来说，测试人员在执行用例的过程中可以一次登录后验证多个功能再退出，但自动化测试的执行有别于手工测试的执行，需要保持测试用例的独立性和完整性，所以每一条用例在执行时都需要登录和退出操作。这个时候就可以把登录和退出的操作封装为公共函数。当每一条用例需要登录/退出时，只需调用它们即可，从而消除代码重复，提高脚本的可维护性。

下面对登录和退出进行模块封装。

mail126.py

```
from selenium import webdriver


# 登录
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("退出").click()
    driver.quit()

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

# 收信、写信、删除信件等操作
# ……

logout()  # 调用退出模块
```

现在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，对于用例本身只需调用这两个函数即可，可以把更多的注意力放到用例本身的操作步骤中。

当然，如果只是把操作步骤封装成函数并没简便太多，我们需要将其放到单独的脚本文件中供其他用例调用。

public.py

```
class Login():

    # 登录
    def user_login(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("123456")
        driver.find_element_by_id("loginBtn").click()

    # 退出
    def user_logout(self, driver):
        driver.find_element_by_link_text("退出").click()
        driver.quit()
```

当函数被独立到单独的脚本文件中时做了一点调整，主要是为函数增加了浏览器驱动的入参。因为函数实现的操作需要通过浏览器驱动 driver，driver 需要通过具体调用的用例给定。

mailTest.py

```
from selenium import webdriver
from public import Login

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

# 调用登录模块
Login().user_login(driver)

# 收信、写信、删除信件等操作
# ……

# 调用退出模块
Login().user_logout(driver)
```

首先，需要导入当前目录下 public.py 文件中的 Login()类，在需要的位置调用类中的 user_login()和 user_logout()函数。这样对于每个用例的编写与维护就方便了很多。

5.3 数据驱动测试实例

前面提到关于数据驱动的形式有很多，我们既可以通过定义变量的方式进行参数化，

也可以通过定义数组、字典的方式进行参数化，还可以通过读取文件（txt\csv\xml）的方式进行参数化。本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　参数化邮箱登录

同样以 126 邮箱的登录为例，现在的需求是测试不同用户的登录。对于测试用例来说，不变的是登录的步骤，变化的是每次登录的用户名和密码，这种情况下就需要用到数据驱动方式来编写测试用例。基于前面的例子做如下修改。

public.py

```
……
# 修改接口需要驱动、用户名和密码等参数
def user_login(self, driver, username, password):
    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()

……
```

修改 user_login()方法的入参，为其增加 username、password 的入参，将得到的具体参数作为登录时的数据。

mailTest.py

```
from selenium import webdriver
from public import Login

class LoginTest():

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

    # admin 用户登录
    def test_admin_login(self):
        username = 'admin'
        password = '123'
```

```
        Login().user_login(self.driver, username, password)
        self.driver.quit()

    # guest 用户登录
    def test_guest_login(self):
        username = 'guest'
        password = '321'
        Login().user_login(self.driver, username, password)
        self.driver.quit()


LoginTest().test_admin_login()
LoginTest().test_guest_login()
```

创建 LoginTest 类，并在__init__()方法中初始化浏览器驱动、等待超时长和 URL 等。这样 test_admin_login()与 test_guset_login()两个测试方法只需关注登录的用户名和密码，通过调用 Login()类的 user_login()方法并传入具体的参数来测试不同用户的登录。

5.3.2 参数化搜索关键字

再来看一个百度搜索的例子。我们每天上网一般要用很多次百度搜索，而我们每次在使用百度搜索时步骤都是一样的，不一样的是每一次搜索的“关键字”不同。下面我们就以数组的方式对搜索的关键字进行参数化。

baidu.py

```
from selenium import webdriver

search_text = ['python', '中文', 'text']

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

这个例子比较简单，首先创建一个数组 search_text 用来存放搜索的关键字，通过 for 循环来遍历数组，最后把遍历的数组元素作为每次百度搜索的关键字。这个例子可以更充

分地体现出数据驱动的概念：因为测试数据的不同从而引起测试结果的不同。

5.3.3　读取 txt 文件

txt 文件是我们经常操作的文件类型，Python 提供了以下几种读取 txt 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。

回到前面登录的例子，现在使用 txt 文件来存放用户名和密码数据，并通过读取该文件中的数据作为用例的测试数据。

user_info.txt

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

首先将用户名和密码按行写入 txt 文件中，这里把用户名和密码用逗号“,”隔开。

user_info.py

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

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

运行结果：

```
======================= RESTART: D:/pyse/user_info.py =======================
zhangsan 123

lisi 456

wangwu 789
```

首先通过 open()方法以读（“r”）的形式打开 user_info.txt 文件，使用 readlines()方法按行读取 txt 文件，将获取到的每一行数据通过 split()方法拆分出用户名和密码。split()可以将一个字符串通过某一个字符为分割点拆分成左右两部分，这里以逗号(,)为分割点。split()拆分出来的左右两部分以数组的形式存放，所以[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。

在上面的例子中循环遍历出每一行数据的用户名和密码，得到想要的数据后就可以将其用于自动化测试脚本了。

5.3.4 读取 csv 文件

那么新的问题来了，假设现在每次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，这时再使用 txt 文件来存放这些数据，读取起来就没那么方便了。对于这种类型的数据可以通过 CSV 文件来存放。

创建 info.csv 文件，首先通过 WPS 表格或 Excel 创建表格，文件另存为 CSV 格式进行保存。注意不要通过直接修改文件的后缀名来创建 CSV 文件，这样创建的并非真正的 CSV 类型的文件，如图 5.4 所示。

图 5.4 CSV 文件

下面编写 csv_read.py 文件进行循环读取。

csv_read.py

```
import csv  # 导入 csv 包
```

```
# 读取本地 CSV 文件
date = csv.reader(open('info.csv', 'r'))

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

运行结果：

```
====================== RESTART: D:/model/csv_read.py ========================
['testing', '123456@126.com', '23', 'man']
['testing2', '123456@qq.com', '18', 'woman']
['testing3', '123456@128.com', '29', 'woman']
```

首先导入 cvs 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，读取的每一行数据均是以数组的形式存储的。如果想取用户的某一列数据，只需指定数组下标即可。

csv_read.py

```
import csv

date = csv.reader(open('info.csv', 'r'))

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

运行结果：

```
====================== RESTART: D:/model/csv_read.py ========================
123456@126.com
123456@qq.com
123456@128.com
```

假如现在需要所有用户的邮箱地址，那么只需指定邮箱地址所在列的下标即可。数组下标是以 0 开始的，邮箱位于数组的第二列，所以指用户邮箱的下标为[1]。

通过这种 CVS 文件来存放数据可以方便地解决读取多列数据的问题。当然，用 Excel 文件来存放这些数据也是一个不错的选择，只是所调用的模块就需要从 csv 切换为 xlrd，针对 Excel 文件操作的方法也会有所不同。

5.3.5 读取 xml 文件

有时候我们需要读取的数据是不规则的。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、登录的用户名和密码等，这时候就可以考虑选择使用 XML 文件来存放这些信息。

什么是 XML 文件？我们在本书 1.6 节做过简单介绍，这里不再赘述。

info.xml

```
<?xml version="1.0" encoding="utf-8"?>
<info>
  <base>
    <platform>Windows</platform>
    <browser>Firefox</browser>
    <url>http://www.baidu.com</url>
      <login username="admin" password="123456"/>
      <login username="guest" password="654321"/>
  </base>
  <test>
      <province>北京</province>
      <province>广东</province>
          <city>深圳</city>
          <city>珠海</city>
      <province>浙江</province>
          <city>杭州</city>
  </test>
</info>
```

下面以 info.xml 文件为例介绍读取 XML 文件的方法。

1. 获得标签信息

read_xml.py

```
from xml.dom import minidom
```

```
# 打开 xml 文档
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: D:/model/read_xml.py =======================
info
None
1
1
```

首先导入 xml 的 minidom 模块，用来处理 XML 文件，parse()用于打开一个 XML 文件，documentElement 用于得到 XML 文件的唯一根元素。

每一个节点都有它的 nodeName、nodeValue、nodeType 等属性。nodeName 为节点名称；nodeValue 为节点的值，只对文本节点有效；nodeType 为节点的类型。

2. 获得任意标签名

read_xml.py

```
from xml.dom import minidom

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

# 得到文档元素对象
root = dom.documentElement

tagname = root.getElementsByTagName('browser')
print(tagname[0].tagName)

tagname = root.getElementsByTagName('login')
```

```
print(tagname[1].tagName)

tagname = root.getElementsByTagName('province')
print(tagname[2].tagName)
```

运行结果：

```
====================== RESTART: D:/model/read_xml.py =======================
browser
login
province
```

getElementByTagName()可以通过标签名获取标签，它所获取的对象是以数组形式存放。假如“login”和“province”标签在 info.xml 文件中有多个，则可以通过指定数组的下标的方式获取某个具体标签。

- getElementsByTagName('province') 获得的是标签名为“province”的一组标签；
- getElementsByTagName('province').tagname[0] 表示一组标签中的第一个；
- getElementsByTagName('province').tagname[2] 表示一组标签中的第三个。

3. 获得标签的属性值

read_xml.py

```
from xml.dom import minidom

# 打开 xml 文档
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 password 属性值
password = logins[0].getAttribute("password")
print(password)
```

```
# 获得第二个 login 标签的 username 属性值
username = logins[1].getAttribute("username")
print(username)

# 获得第二个 login 标签的 password 属性值
password = logins[1].getAttribute("password")
print(password)
```

运行结果：

```
====================== RESTART: D:/model/read_xml.py =======================
admin
123456
guest
654321
```

getAttribute()方法用于获取元素的属性值。它和 WebDriver 中所提供的 get_attribute()方法相似。

4. 获得标签对之间的数据

read_xml.py

```
from xml.dom import minidom

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

# 得到文档元素对象
root = dom.documentElement

provinces = dom.getElementsByTagName('province')
citys = dom.getElementsByTagName('city')

# 获得第二个 province 标签对的值
p2 = provinces[1].firstChild.data
print(p2)

# 获得第一个 city 标签对的值
c1 = citys[0].firstChild.data
print(c1)
```

```
# 获得第二个 city 标签对的值
c2 = citys[1].firstChild.data
print(c2)
```

```
运行结果：
====================== RESTART: D:/model/read_xml.py ========================
广东
深圳
珠海
```

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，它和 WebDriver 中提供的 text 方法类似。

本章小结

在本章的学习过程中，我们首先介绍了自动化测试模型的发展，然后通过实例介绍了模块化的应用。模块化和数据驱动在脚本开发过程中是必不可少的两个知识点，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。在数据驱动小节里，分别介绍了不同形式的数据驱动，读者可以根据实际需求选择合适的数据驱动形式。

第 6 章
Selenium IDE

相信有不少读者学习 Selenium 是从 Selenium IDE 开始的，作为基于 Firefox 浏览器的一个插件，Selenium IDE 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。

本书的目的是帮助读者开发自动化测试脚本，Selenium IDE 作为 Selenium 家族的成员之一，我们有必要掌握它的使用。从另一个角度考虑，可以使用 Selenium IDE 将录制的脚本生成相应的带单元测试框架的自动化脚本，这将有助于下一章的学习。

6.1 Selenium IDE 安装

早期 Selenium IDE 的安装非常简单，打开 Firefox 浏览器，选择菜单栏“工具”→“附加组件”，然后搜索“Selenium IDE”，从搜索结果中选择 Selenium IDE 进行安装，安装完成后重启浏览器即可。但现在 Firefox 的“附加组件”中已经搜索不到 Selenium IDE 了，所以我们只能通过 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，单击版本号链接，如图 6.1 所示。

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 2.9.0 released on 09/Mar/2015 or view the Release Notes and then install some plugins.

Download previous version 2.8.0 released on 29/Sep/2014.

图 6.1　Selenium IDE 介绍

Firefox 浏览器将自动识别需要下载的 Selenium IDE 插件，如图 6.2 所示，单击“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，或通过组合键 Ctrl+Alt+S 打开，如图 6.3 所示。

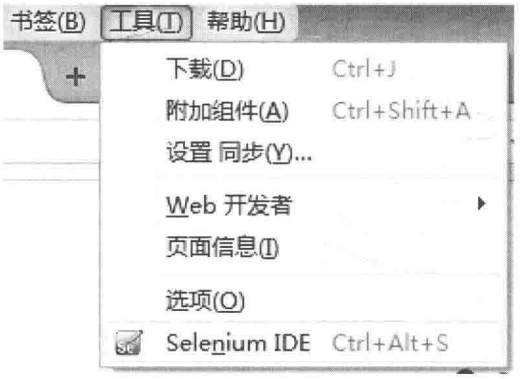

图 6.3　打开 Selenium IDE

6.1.2　下载插件安装

通过下载插件的安装方式与前面的安装方法类似，如图 6.1 所示。如果使用非 Firefox 浏览器单击 Selenium IDE 的版本号链接，那么将会提示下载 Selenium IDE，下载完成后将得到一个 selenium-ide-x.x.x.xpi 的文件。

打开 Firefox 浏览器，选择菜单栏“工具”→“附加组件”，单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（I）...”，如图 6.4 所示。

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-x.x.x.xpi 文件后按“确定”按钮，将会弹出如图 6.2 所示窗口，单击“立刻安装”按钮进行安装，安装完成后重启浏览器即可。

6.2 Selenium IDE 界面介绍

打开 Selenium IDE，界面如图 6.5 所示。

图 6.5 Selenium IDE 界面

Selenium IDE 界面介绍：

1. **文件（F）**：创建、打开和保存测试案例和测试案例集。

   编辑（E）：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。

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 侧则缓慢执行。

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．**定时任务**：用于设置测试套件的定时执行（Selenium IDE 2.9 中的新增功能）。

9．**录制**：记录用户对浏览器的操作步骤并生成脚本。

10．**Test Case**：　表示案例集列表。

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

12．**Runs/Failures**：记录用例运行通过/失败的个数。Runs 表示用例执行成功；Failures 表示用例执行失败。

13．**Table/Source**：分别以不同的格式展示测试脚本。在 Table 标签中用例的一条命令由 Command、Target、Value 三个部分组成。

14．**Log/Reference/UI-Element/Rollup**：

Log：当你运行测试时，错误和信息将会自动显示。

Reference：当在表格中输入和编辑 selenese 命令时，面板上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 https://www.baidu.com），脚本录制完成后，关闭录制按钮，如图 6.6 所示。

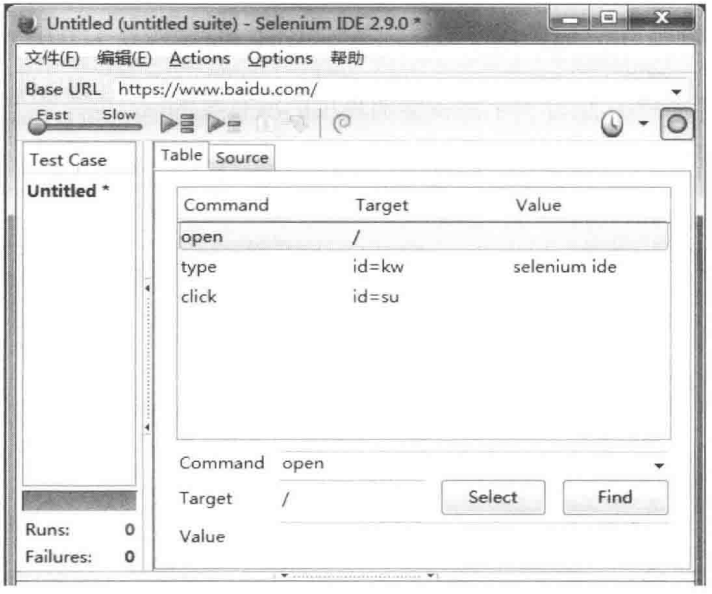

图 6.6 Selenium IDE 录制脚本

6.3.2 编辑脚本

Selenium IDE 录制的脚本通常不是百分之百地符合我们的需求，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进行编辑，如图 6.7 所示。

2. 插入命令

在某一条命令上右击，选择“Insert New Command”命令，即可插入一个空白命令，然后对空白行进行编辑，如图 6.8 所示。

3. 插入注解

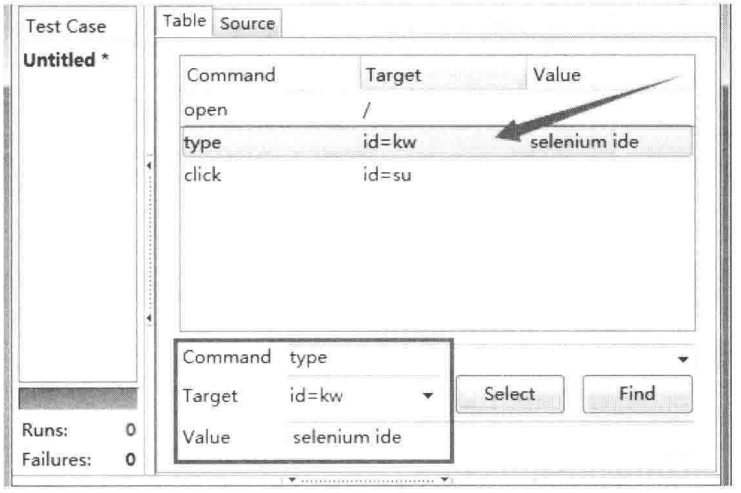

图 6.7　Selenium IDE 编辑脚本

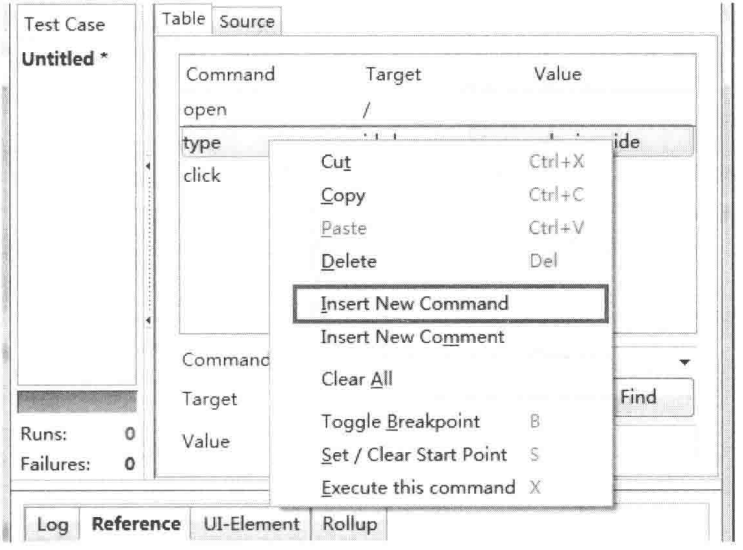

图 6.8　插入一条命令

以同样的方式右击选择“Insert New Comment”命令，插入注解行，以便帮助我们阅读脚本，本行内容不被执行，插入的内容以紫色字体显示，如图 6.9 所示。

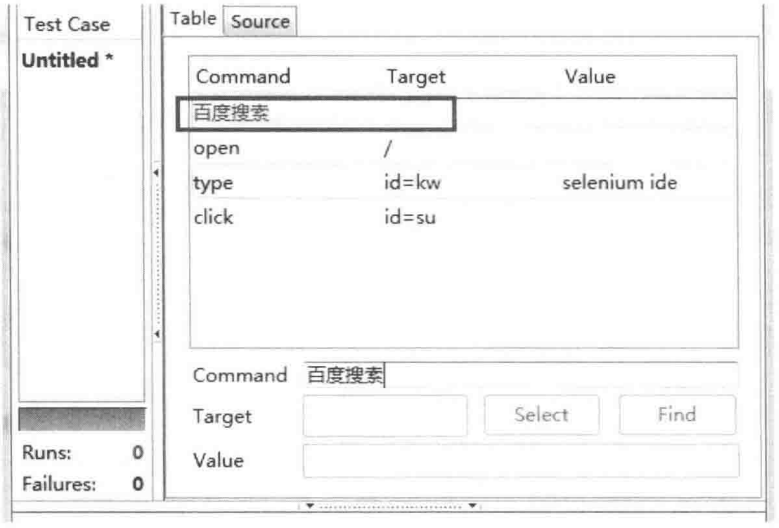

图 6.9　添加一条注释

4. 移动操作

有时我们需要移动某行命令的顺序，只需单击鼠标拖动到相应的位置即可，如图 6.10 所示。

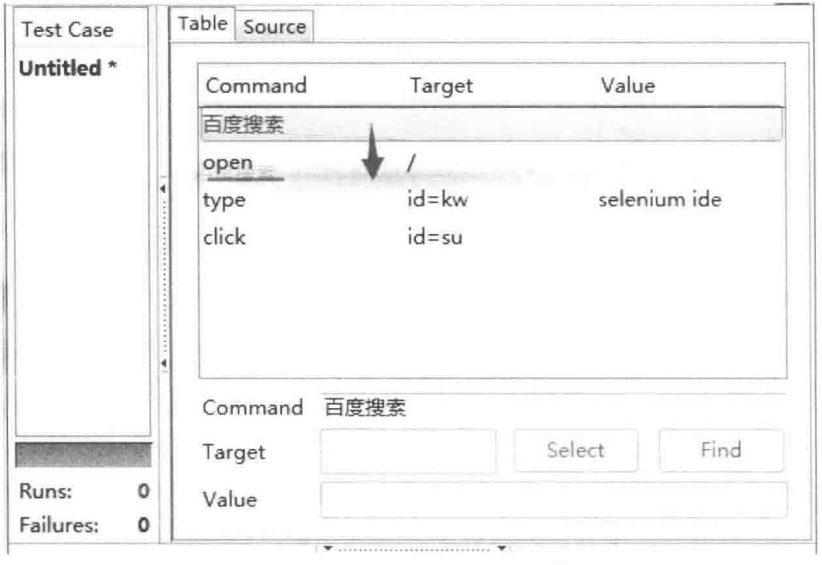

图 6.10　移动元素

5. 定位辅助

当 Selenium IDE 录制脚本时，Target 会生成针对当前元素的所有定位方式，可以单击 Target 下拉框选择元素定位方式，如图 6.11 所示。

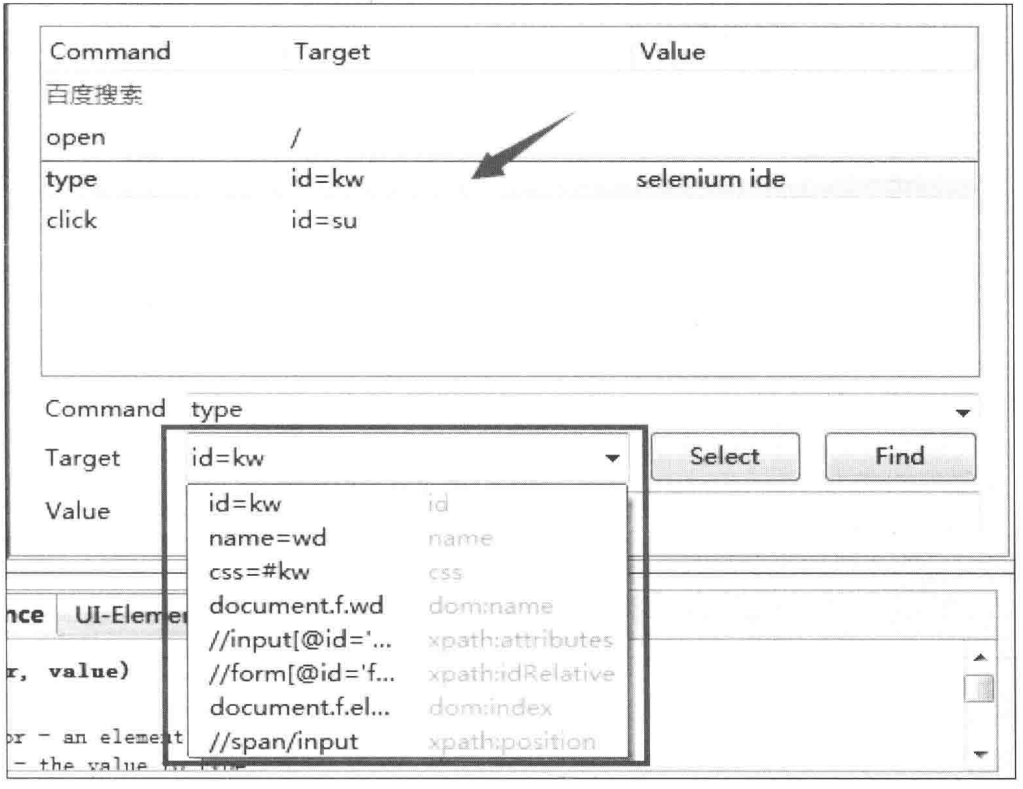

图 6.11　选择元素定位方式

6.4　Selenium IDE 命令

Selenium IDE 中提供了丰富的操作命令，在 Selenium IDE 的 Command 的下拉列表框中可以选择使用这些命令，如图 6.12 所示。

1. 下面介绍一些常用命令的使用。

open

open(url)

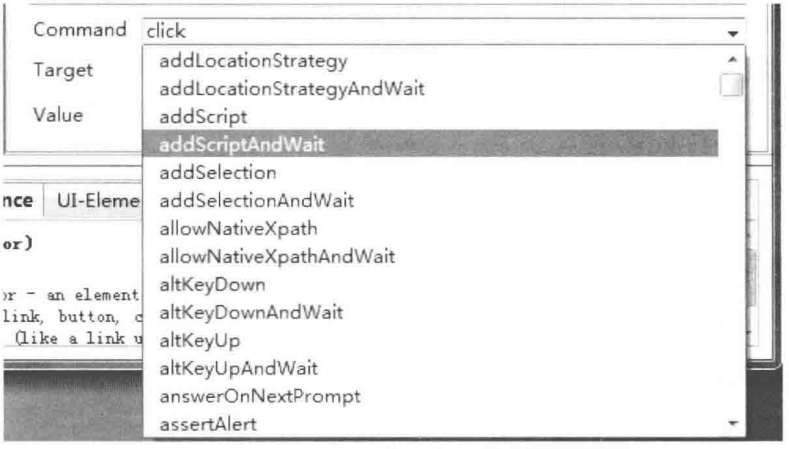

图 6.12　Selenium IDE 命令

- 在浏览器中打开 URL，可以接受相对路径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/	

2. click

click(elementLocator)

- 单击链接、按钮、复选和单选框。

- 如果单击后需要等待响应，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	

3. type

type(inputLocator，value)

- 模拟键盘的输入，向指定的 input 中输入值。

- 也适合给复选框和单选框赋值。

- 在这个例子中，只是给勾选了的复选框赋值。注意，只是赋值，而不是改写其文本。

Command	Target	Value
type	nameField	John Smith
typeAndWait	textBoxThatSubmitsOnChange	newValue

4. select

select(dropDownLocator，optionSpecifier)

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

- 当多于一个选择器的时候，如在用通配符模式“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

5. goBack

goBack()

模拟单击浏览器的后退按钮。

Command	Target	Value
goBack		

6. selectWindow

select(windowId)

- 选择一个弹出窗口。

- 当选中那个窗口时，所有的命令将会转移到被选择窗口中执行

Command	Target	Value
selectWindow	myPopupWindow	
selectWindow	null	

7. pause

pause(millisenconds)

- 根据指定时间暂停 Selenium 脚本执行。

- 常在调试脚本或等待服务器响应时使用。

Command	Target	Value
pause	5000	
pause	2000	

8. 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

9. close

- 模拟单击浏览器关闭按钮。

Command	Target	Value
close		

6.5 断言与验证

严格意义上讲，我们前面所写的自动化测试脚本不能叫测试用例，真正的测试用例一

定是需要做断言或验证的，一定要有预期结果与实际结果进行比较的过程。在功能测试用例执行过程中，断言由测试人员通过眼睛来完成；所以，我们在写自动化测试用例时，往往只注重操作步骤的模拟而忽视对实际结果的断言。

在 Selenium IDE 中提供了断言与验证来对结果进行比较。

如何向 Selenium IDE 的脚本中添加断言与验证呢？首先打开 Selenium IDE，右击页面（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”，如图 6.13 所示。

图 6.13　右键快捷菜单

可以选择添加断言与验证命令，如图 6.14 所示。

通过图 6.14 的菜单，获得了 4 类命令：assert（断言）、verify（验证）、waitFor（等待）、store（定义变量）。这四类命令又分为 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。

```
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　断言与验证命令

6.5.1　断言

如果使用断言，测试用例将会在断言失败后停止运行。有时候，这正是我们想要的结果。如果测试失败，我们会立刻知道测试没有通过。

优点：我们可以直截了当地看到检查是否通过。

缺点：若检查失败，后续的检查不会被执行，无法收集检查的结果状态。

通过图 6.14 可以帮助我们向脚本中添加断言命令，黑色的选项表示可选，灰色的选项表示当前不可选。

Baidu Test Case		
Command	Target	Value
open	http://www.baidu.com/	
type	id=kw	selenium ide
click	id=su	

（续）

Baidu Test Case		
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添加了三种断言：Title、Text 和 ElementPresent，它们分别用来断言浏览器的标题、文本信息和当前元素。

6.5.2　验证

与断言相比，当执行验证命令失败后不会终止测试。如果你的测试只使用验证，可以得到的保证是：假设没有意外异常，则测试会被执行完毕，而不管是否发现缺陷。缺点：你必须做更多的工作，以检查测试结果。

同样参考图 6.14 添加了三种验证命令。

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，你想让测试脚本怎么做？是想让测试终止，还是想让测试简单地记录检查失败后继续执行。

下面我们通过一个例子来解释断言与验证的差异。

Baidu Test Case		
Command	Target	Value
open	http://www.baidu.com/	
type	id=kw	selenium ide

（续）

Baidu Test Case		
click	id=su	
a ssert Title	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，特意设置有误的断言信息“selenium ide_百度搜索 sss”使断言失败。Selenium IDE 执行的 log 信息如下：

- [info] Playing test case baidu_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”位置时，断言失败，脚本终止执行。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 baidu_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，但没有终止脚本的执行。

6.6　等待与变量

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

6.6.1　等待

Selenium IDE 中提供了 pause 来设置固定时间的休眠，而 waitFor 则用于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 的 Value 如果为空，则默认时间为 60 秒。上面例子中的 waitForTitle、waitForText 和 waitForElementPresent 分别用来等待浏览器的标题、文本信息和当前元素。

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

（续）

Baidu Test Case		
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		

可把页面中获取到的标题、文本信息和元素分别定义成为 title、text 和 element 变量，下面通过定义的变量作为断言与验证的比较参数。

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		

就像在编程语言中一样，我们时常会用到变量的定义，store 用于定义一个普通的变量。下面来看看它的使用场景。

store 格式：

store(expression, variableName)

- expression：定义的变量值，该值可以由其他变量组合而成，或通过 JavaScript 表达式赋值给变量；variableName：定义的变量名。

Command	Target	Value
store	Mr John Smith	fullname
store	$. { title } $. { firstname } $. { suname }	fullname
store	javascript. { Math.round(Math.PI*100)/100 }	PI

例如，将百度搜索的关键字定义为变量。

Command	Target	Value
store	selenium ide	value
type	id=kw	${value}

本章小结

通过对本章的学习，我们已经基本掌握了 Selenium IDE 的使用、如何录制与回放脚本、如何修改与编辑脚本，以及如何在脚本中添加断言与验证。Selenium IDE 用法简单，可以满足小型且相对简单的自动化测试工作。

我们学习它的目的并不是为了使用它来进行自动化测试，但对于新手来讲，正如本章节开头所说的，它可以帮助我们编写自动化测试脚本。

第7章
unittest单元测试框架

对于不熟悉编程的测试新手来讲，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胜任这项工作。其实，它并非想象得那么高级，本章我们就来揭开单元测试的面纱。

可能读者还有个疑问，我们不是在学 Web 自动化么？为什么要去学习单元测试框架，本书又不是教我们写单元测试的书。其实单元测试框架并非只能用于代码级别的测试，对于单元测试框架来讲，笔者认为它主要完成以下三件事。

提供用例组织与执行：当你的测试用例只有几条时，可以不必考虑用例的组织，但是，当测试用例达到成百上千条时，大量的测试用例堆砌在一起，就产生了扩展性与维护性等问题，此时需要考虑用例的规范与组织问题了。单元测试框架就是用来解决这个问题的。

提供丰富的比较方法：不论是功能测试，还是单元测试，在用例执行完成之后都需要将实际结果与预期结果进行比较（断言），从而断定用例是否执行通过。单元测试框架一般会提供丰富的断言方法。例如，判断相等/不等、包含/不包含、True/False 的断言方法等。

提供丰富的日志：当测试用例执行失败时能抛出清晰的失败原因，当所有用例执行完成后能提供丰富的执行结果。例如，总执行时间、失败用例数、成功用例数等。

一般的单元测试框架都会提供这些功能，从单元测试框架的这些特性来看，它同样适用于 Web 自动化用例的开发与执行。

7.1　认识 unittest

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 doctest、unittest、pytest、nose 等，unittest 框架（原名 PyUnit 框架）为 Python 语言自带的单元测试框架，Python 2.1 及其以后的版本已将 unittest 作为一个标准模块放入 Python 开发包中。

7.1.1　认识单元测试

可能读者会问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段代码，所以不用单元测试框架也可以写单元测试，下面就通过例子演示不用测试框架的单元测试。

首先创建一个被测试类 calculator.py。

calculator.py

```
# 计算器类
class Count:

    def __init__(self, a, b):
        self.a = int(a)
        self.b = int(b)

    # 计算加法
    def add(self):
        return self.a + self.b
```

程序非常简单，创建一个 Count 类用于两个整数的计算，通过__init__()方法对两个数进行初始化，接着创建 add()方法返回两个数相加的结果。

根据上面所实现的功能，不用测试框架所编写的单元测试如 test.py。

test.py

```
from calculator import Count
```

```
# 测试两个整数相加
class TestCount:

    def test_add(self):
        try:
            j = Count(2, 3)
            add = j.add()
            assert(add == 5), 'Integer addition result error!'
        except AssertionError as 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 calculator 文件中的 Count 类；然后在 test_add()方法中调用 Count 类并传入两个参数 2 和 3；最后调用 Count 类中的 add()方法对两个参数做加法运算，并通过 assert()方法判断 add()的返回值是否等于 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

```
#assert 比较相等的结果：
======================== RESTART: D:/test/test.py ==========================
Test pass!

#assert 比较不相等的结果：
======================== RESTART: D:/test/test.py ==========================
Integer addition result error!
```

不难发现这种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，不统一的代码维护起来会十分麻烦。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，而且这仅仅是一个测试用例，对一个单元模块来说，只编写一条测试用例显然是不够的。

为了让单元测试代码更容易维护和编写，最好的方式是遵循一定的规范来编写测试用

例，这也是单元测试框架诞生的初衷。接下来讲如何通过 unittest 单元测试框架编写单元测试用例。

test.py

```
from calculator import Count
import unittest


class TestCount(unittest.TestCase):

    def setUp(self):
        print("test start")

    def test_add(self):
        j = Count(2, 3)
        self.assertEqual(j.add(), 5)

    def tearDown(self):
        print("test end")

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

分析上面的代码，首先引入 unittest 模块，创建 TestCount 类继承 unittest 的 TestCase 类，我们可以将 TestCase 类看成是对特定类进行测试的集合。

setUp()方法用于测试用例执行前的初始化工作，这里只简单打印“test start”信息。tearDown()方法与 setUp()方法相呼应，用于测试用例执行之后的善后工作，这里打印“test end”信息。

在 test_add()中首先调用 Count 类并传入要计算的数，通过调用 add()方法得到两数相加的返回值。这里不再使用烦琐的异常处理，而是调用 unittest 框架所提供的 assertEqual()方法对 add()的返回值进行断言，判断两者是否相等，assertEqual()方法由 TestCase 类继承而来。

unittest 提供了全局的 main()方法，使用它可以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以“test”命名开头的测试方法，并自动执行它们。

Python 知识补充

if __name__ == "__main__"：语句说明

在后面实例中我们会经常用到这个语句，在解释它之前先补充点 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，简单地说，就是.py 文件的调用方式。最后是__main__，如上所述，.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__"就表示是直接使用。

7.1.2 重要的概念

在 unittest 的文档中开篇就介绍了 4 个重要的概念：test fixture、 test case、 test suite 和 test runner，只有理解了这几个概念才能理解单元测试的基本特征。

1. 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)。单元测试(unit test)的本质也就在这里，一个测试用例就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。

2. Test Suite

一个功能的验证往往需要多个测试用例，可以把多个测试用例集合在一起来执行，这就产生了测试套件 TestSuite 的概念。Test Suite 用来组装单个测试用例。可以通过 addTest 加载 TestCase 到 TestSuite 中，从而返回一个 TestSuite 实例。

3. Test Runner

测试的执行也是单元测试中非常重要的一个概念，一般单元测试框架中都会提供丰富的执行策略和执行结果。在 unittest 单元测试框架中，通过 TextTestRunner 类提供的 run()方法来执行 test suite/test case。test runner 可以使用图形界面、文本界面，或返回一个特殊的值等方式来表示测试执行的结果。

4. Test Fixture

对一个测试用例环境的搭建和销毁，就是一个 fixture，通过覆盖 TestCase 的 setUp()和 tearDown()方法来实现。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()中通过建立数据库连接来进行初始化，在 tearDown()中清除数据库产生的数据，然后关闭连接等。

注意：tearDown 的过程很重要，要为下一个 test case 留下一个干净的环境。

理解了前面几个概念之后，我们再结合例子来学习。

test.py

```
from calculator import Count
import unittest


class TestCount(unittest.TestCase):

    def setUp(self):
        print("test start")

    def test_add(self):
        j = Count(2, 3)
        self.assertEqual(j.add(), 5)

    def test_add2(self):
        j = Count(41, 76)
        self.assertEqual(j.add(), 117)

    def tearDown(self):
        print("test end")

if __name__ == '__main__':
```

```
# 构造测试集
suite = unittest.TestSuite()
suite.addTest(TestCount("test_add2"))
# 执行测试
runner = unittest.TextTestRunner()
runner.run(suite)
```

在前面例子的基础上编写了第二个测试用例 test_add2()。由于第一个测试用例已经运行通过，因此这次只需运行第二条测试用例。在代码的最后，我们去掉了 main()方法，采用构造测试集的方法来加载与运行测试用例，实现了有选择地执行测试用例。当然，也可以通过注释的方式注释掉第一条用例，但这种做法并不优雅。

首先，调用 unittest 框架的 TestSuite()类来创建测试套件，通过它所提供的 addTest()方法来添加测试用例 test_add2()。接着调用 unittest 框架的 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: D:/test/test.py ==========================
test start
test end
.
----------------------------------------------------------------------
Ran 1 test in 0.016s

OK
```

从执行结果可以看到，setUp/tearDown 作用于测试用例的开始与结束。

7.1.3 断言方法

在执行用例的过程中，最终用例是否执行通过，是通过判断测试得到的实际结果与预期结果是否相等决定的。unittest 框架的 TestCase 类提供下面这些方法用于测试结果的判断。

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	

（续）

方　法	检　查	版　本
assertIs(a, b)	a is b	3.1
assertIsNot(a, b)	a is not b	3.1
assertIsNone(x)	x is None	3.1
assertIsNotNone(x)	x is not None	3.1
assertIn(a, b)	a in b	3.1
assertNotIn(a, b)	a not in b	3.1
assertIsInstance(a, b)	isinstance(a, b)	3.2
assertNotIsInstance(a, b)	not isinstance(a, b)	3.2

- assertEqual(first, second, msg=None)

断言第一个参数和第二个参数是否相等，如果不相等则测试失败。msg 为可选参数，用于定义测试失败时打印的信息。

test.py

```
import unittest


class Test(unittest.TestCase):

    def setUp(self):
        number = input("Enter a number:")
        self.number = int(number)

    def test_case(self):
        self.assertEqual(self.number, 10, msg="Your input is not 10!")

    def tearDown(self):
        pass

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

注意：此例使用 Python 自带的 IDLE 运行，因为执行过程需要用户输入。

在 setUp()方法中要求用户输入一个数，在 test_case()中通过 assertEqual()比较输入的数是否等于 10，如果不相等则输出 msg 中定义的提示信息。执行结果如下。

Python Shell

```
========================= RESTART: D:/test/test.py ==========================
Enter a number:12
F
======================================================================
FAIL: test_case (__main__.Test)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\test\test.py", line 11, in test_case
    self.assertEqual(self.number, 10, msg="Your input is not 10!")
AssertionError: 12 != 10 : Your input is not 10!

----------------------------------------------------------------------
Ran 1 test in 3.760s

FAILED (failures=1)
```

从执行结果看到，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“Your input is not 10!”。

- assertNotEqual(first, second, msg=None)

assertNotEqual()与 assertEqual()相反，它用于断言第一个参数与第二个参数是否不相等，如果相等则测试失败。

- assertTrue(expr, msg=None)

- assertFalse(expr, msg=None)

测试表达式是 true（或 false）。

下面来实现判断一个数是否为质数的功能，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

```
# 用于判断质数
def is_prime(n):
    if n <= 1:
        return False
```

```
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

创建 is_prime()函数用于实现对质数的判断。当得到一个数字 *n* 后，首先判断它是否小于或等于 1，如果小于或等于 1，则直接返回 False；如果大于 1，则对其进行循环判断；若能整除 2 到其自身之间的任意一个数，则不为质数，返回 False，否则返回 True。

test.py

```
from count import is_prime
import unittest


class Test(unittest.TestCase):

    def setUp(self):
        print("test start")

    def test_case(self):
        self.assertTrue(is_prime(7), msg="Is not prime!")

    def tearDown(self):
        print("test end")

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，在上面的例子中传值为 7，显然是一个质数，所以通过 assertTrue()断言得到的结果为 True。

- assertIn(first, second, msg=None)

- assertNotIn(first, second, msg=None)

断言第一个参数是否在第二个参数中，反过来讲，第二个参数是否包含第一个参数。

test.py

```
import unittest
```

```
class Test(unittest.TestCase):

    def setUp(self):
        print("test start")

    def test_case(self):
        a = "hello"
        b = "hello world"
        self.assertIn(a, b, msg="a is not in b")

    def tearDown(self):
        print("test end")

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，如果不包含则打印 msg 定义的信息。

- assertIs(first, second, msg=None)

- assertIsNot(first, second, msg=None)

断言第一个参数和第二个参数是否为同一对象。

- assertIsNone(expr, msg=None)

- assertIsNotNone(expr, msg=None)

断言表达式是否为 None 对象。

- assertIsInstance(obj, cls, msg=None)

- assertNotIsInstance(obj, cls, msg=None)

断言 obj 是否为 cls 的一个实例。

在 unittest 中还提供了其他检查比较的方法，因为不常用，所以不再一一介绍。读者可参考 Python 官方文档 unittest 章节进行学习。

7.1.4　组织单元测试用例

当我们增加被测功能和相应的测试用例之后，再来看看 unittest 单元测试框架是如何扩展和组织新增的测试用例的。

我们同样以测试 7.1.1 节中的 calculator.py 文件为例，为其扩展 sub()方法，用来计算两个数相减的结果。

calculator.py

```
# 计算器类
class Count():

    def __init__(self, a, b):
        self.a = int(a)
        self.b = int(b)

    # 计算加法
    def add(self):
        return self.a + self.b

    # 计算减法
    def sub(self):
        return self.a - self.b
```

因为对计算器（calculator）又新增了减法功能（sub），所以需要针对新功能编写测试用例，扩展后的 test.py 文件如下。

test.py

```
from calculator import Count
import unittest


class TestAdd(unittest.TestCase):

    def setUp(self):
        print("test add start")

    def test_add(self):
        j = Count(2, 3)
```

```
        self.assertEqual(j.add(), 5)

    def test_add2(self):
        j = Count(41, 76)
        self.assertEqual(j.add(), 117)

    def tearDown(self):
        print("test add end")


class TestSub(unittest.TestCase):

    def setUp(self):
        print("test sub start")

    def test_sub(self):
        j = Count(2, 3)
        self.assertEqual(j.sub(), -1)

    def test_sub2(self):
        j = Count(71, 46)
        self.assertEqual(j.sub(), 25)

    def tearDown(self):
        print("test sub end")


if __name__ == '__main__':
    # 构造测试集
    suite = unittest.TestSuite()
    suite.addTest(TestAdd("test_add"))
    suite.addTest(TestAdd("test_add2"))
    suite.addTest(TestSub("test_sub"))
    suite.addTest(TestSub("test_sub2"))

    # 运行测试集合
    runner = unittest.TextTestRunner()
    runner.run(suite)
```

上例中创建了 TestAdd()和 TestSub()两个测试类，分别测试 calculator.py 文件中的 add()和 sub()两个功能。通过 TestSuite 类的 addTest()方法把不同测试类中的测试方法组装到测试套件中，执行结果如下。

Python Shell

```
========================== RESTART: D:/test/test.py ==========================
test add start
test add end
.test add start
test add end
.test sub start
test sub end
.test sub start
test sub end
.
----------------------------------------------------------------------
Ran 4 tests in 0.047s

OK
```

通过测试结果可以看到，setUp()和 tearDown()方法分别作用于每个测试用例的开始与结束。如果每个类中的 setUp()和 tearDown()所做的事情是一样的，那是不是可以封装一个自己的测试类呢？

test.py

```
from calculator import Count
import unittest


class MyTest(unittest.TestCase):

    def setUp(self):
        print("test case start")

    def tearDown(self):
        print("test case end")


class TestAdd(MyTest):

    def test_add(self):
        j = Count(2, 3)
        self.assertEqual(j.add(), 5)
```

```
    def test_add2(self):
        j = Count(41, 76)
        self.assertEqual(j.add(), 117)


class TestSub(MyTest):

    def test_sub(self):
        j = Count(2, 3)
        self.assertEqual(j.sub(), -1)

    def test_sub2(self):
        j = Count(71, 46)
        self.assertEqual(j.sub(), 25)

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

创建 MyTest()类的好处显而易见，对于测试类和测试方法来说，应将注意力放在具体用例的编写上，无须关心 setUp()和 tearDown()所做的事情。不过，前提条件是 setUp()和 tearDown()所做的事情是每个用例都需要的。

7.1.5 discover 更多测试用例

随着软件功能的不断增加，对应的测试用例也会呈指数级增长。一个实现几十个功能的项目，对应的单元测试用例可能达到上百个。如果把所有的测试用例都写在一个 test.py 文件中，那么这个文件会越来越臃肿，后期维护起来也比较麻烦。需要将这些用例按照所测试的功能进行拆分，分散到不同的测试文件中。

对上例中 test.py 文件的测试用例进行拆分，拆分后的目录结构如下：

```
testpro/
├── count.py
├── testadd.py
├── testsub.py
└── runtest.py
```

文件拆分后的实现代码如下。

testadd.py

```
from calculator import Count
import unittest


class TestAdd(unittest.TestCase):

    def setUp(self):
        print("test case start")

    def tearDown(self):
        print("test case end")

    def test_add(self):
        j = Count(2, 3)
        self.assertEqual(j.add(), 5)

    def test_add2(self):
        j = Count(41, 76)
        self.assertEqual(j.add(), 117)

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

testsub.py

```
from calculator import Count
import unittest


class TestSub(unittest.TestCase):

    def setUp(self):
        print("test case start")

    def tearDown(self):
        print("test case end")

    def test_sub(self):
        j = Count(2, 3)
```

```
        self.assertEqual(j.sub(), -1)

    def test_sub2(self):
        j = Count(71, 46)
        self.assertEqual(j.sub(), 25)

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

接着创建用于执行所有用例的 runtest.py 文件。

runtest.py

```
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(testsub.TestSub("test_sub"))
suite.addTest(testsub.TestSub("test_sub2"))

if __name__ == '__main__':
    # 执行测试
    runner = unittest.TextTestRunner()
    runner.run(suite)
```

这样的拆分带来了好处，可以根据不同的功能创建不同的测试文件，甚至是不同的测试目录，测试文件中还可以将不同的小功能划分为不同的测试类，在类下编写测试用例，整体结构更加清晰。

这样的设计看上去很完美，但依然没有解决添加用例的问题，当用例达到成百上千条时，在 runtest.py 文件中通过 addTest()添加/删除测试用例就变得非常麻烦，那么有没有方法让 unittest 单元测试框架自动识别测试用例呢？答案是肯定的，TestLoader 类中提供的 discover()方法可以解决这个问题。

TestLoader

该类负责根据各种标准加载测试用例，并将它们返回给测试套件。正常情况下，不需要创建这个类的实例。unittest 提供了可以共享的 defaultTestLoader 类，可以使用其子类和方法创建实例，discover()方法就是其中之一。

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。此处匹配文件名以“test”开头的“.py”类型的文件，星号“*”表示任意多个字符。
- top_level_dir=None：测试模块的顶层目录，如果没有顶层目录，默认为 None。

现在通过 discover()方法重新实现 runtest.py 文件的功能。

runtest.py

```
import unittest

# 定义测试用例的目录为当前目录
test_dir = './'
discover = unittest.defaultTestLoader.discover(test_dir, pattern='test*.py')

if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    runner.run(discover)
```

discover()方法会自动根据测试目录（test_dir）匹配查找测试用例文件（test*.py），并将查找到的测试用例组装到测试套件中，因此，可以直接通过 run()方法执行 discover ，大大简化了测试用例的查找与执行。

7.2　关于 unittest 还需要知道的

关于 unittest 单元测试框架，还有一些问题值得进一步探讨。也许你已经在 7.1 节的学习过程中产生了一些疑问，也许你会在本节中找到答案。

7.2.1 用例执行的顺序

用例的执行顺序涉及多个层级：在多个测试目录的情况下，先执行哪个目录？在多个测试文件的情况下，先执行哪个文件？在多个测试类的情况下，先执行哪个测试类？在多个测试方法（用例）的情况下，先执行哪个测试方法？

我们先来运行一个例子，再来解释 unittest 的执行策略。

test.py

```
import unittest


class TestBdd(unittest.TestCase):

    def setUp(self):
        print("test TestBdd :")

    def test_ccc(self):
        print("test ccc")

    def test_aaa(self):
        print("test aaa")

    def tearDown(self):
        pass


class TestAdd(unittest.TestCase):

    def setUp(self):
        print("test TestAdd :")

    def test_bbb(self):
        print("test bbb")

    def tearDown(self):
        pass

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

用例的执行结果如下：

Python Shell

```
========================= RESTART: D:/test/test.py =========================
test TestAdd :
test bbb
.test TestBdd :
test aaa
.test TestBdd :
test ccc
.
----------------------------------------------------------------------
Ran 3 tests in 0.047s

OK
```

无论执行多少次，结果都是一样的，通过上面的结果，相信你已经找到了 unittest 执行测试用例的规律。

unittest 框架默认根据 ASCII 码的顺序加载测试用例，数字与字母的顺序为：0~9，A~Z，a~z。所以，TestAdd 类会优先于 TestBdd 类被执行，test_aaa()方法会优先于 test_ccc()被执行，因而它并没有按照用例从上到下的顺序执行。

对于测试目录与测试文件来说，unittest 框架同样是按照这个规则来加载测试用例的。

那么可不可以让 test_ccc()先执行？答案是肯定的，只是不能使用默认的 main()方法了，而是需要通过 TestSuite 类的 addTest()方法按照一定的顺序来加载。

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: D:/test/test.py =========================
test TestBdd :
test ccc
.test TestAdd :
test bbb
.test TestBdd :
test aaa
.
----------------------------------------------------------------------
Ran 3 tests in 0.037s

OK
```

现在的执行顺序就是 addTest()方法所加载的顺序。discover()的加载测试用例的规则与 main()方法相同。所以，我们只能通过测试用例的命名来提高被执行的优先级。例如，将希望先被执行的测试用例命名为“test_a”，将希望最后执行的测试用例命名为“test_z”。

7.2.2 执行多级目录的用例

我们要控制 Web 用例的数量，但是当测试用例达到一定量级时，就要考虑划分目录，比如规划如下测试目录：

```
test_project/test_case/
        ├── test_bbb/
        │   ├── test_ccc/
        │   │   └── test_c.py
        │   └── test_b.py
        ├── test_ddd/
```

```
│       └── test_d.py
└── test_a.py
```

对于上面的目录结构，如果将 discover()方法中的 start_dir 参数定义为“./test_case/”目录，那么只能加载 test_a.py 文件中的测试用例。怎样让 unittest 框架查找到 test_case/的子目录中的测试文件呢？方法很简单，在每个子目录下放一个__init__.py 文件。__init__.py 文件的作用请参考本书第 3.6 节。

7.2.3　跳过测试和预期失败

在运行测试时，有时需要直接跳过某些测试用例，或者当用例符合某个条件时跳过测试，又或者直接将测试用例设置为失败。unittest 提供了实现这些需求的装饰器。

- unittest.skip(reason)

无条件地跳过装饰的测试，说明跳过测试的原因。

- unittest.skipIf(condition, reason)

跳过装饰的测试，如果条件为真时。

- unittest.skipUnless(condition, reason)

跳过装饰的测试，除非条件为真。

- unittest.expectedFailure()

测试标记为失败。不管执行结果是否失败，统一标记为失败。

test.py

```
import unittest


class MyTest(unittest.TestCase):

    def setUp(self):
        pass

    def tearDown(self):
        pass
```

```
    @unittest.skip("直接跳过测试")
    def test_skip(self):
        print("test aaa")

    @unittest.skipIf(3 > 2, "当条件为 True 时跳过测试")
    def test_skip_if(self):
        print('test bbb')

    @unittest.skipUnless(3 > 2, "当条件为 True 时执行测试")
    def test_skip_unless(self):
        print('test ccc')

    @unittest.expectedFailure
    def test_expected_failure(self):
        assertEqual(2, 3)


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

执行结果如下。

Python Shell

```
========================= RESTART: D:/test/test.py =========================
xsstest ccc
.
----------------------------------------------------------------------
Ran 4 tests in 0.016s

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

上例中共创建了 4 条测试用例。第一条测试用例通过@unittest.skip()装饰，直接跳过不执行。第二条用例通过@unittest.skipIf()装饰，当条件为真时不执行，3>2 条件为真（True），跳过不执行。第三条用例通过@unittest.skipUnless()装饰，当条件为真时执行，判断 3>2 条件为真（True），第三条用例执行。第四条用例通过@unittest.expectedFailure 装饰，不管执行结果是否失败，统一标记为失败，但不会抛出错误信息。

当然，这些方法同样可以作用于测试类，只需将它们定义在测试类上面即可。

test.py

```
import unittest


@unittest.skip("直接跳过测试该测试类")
class MyTest(unittest.TestCase):
......
```

7.2.4　fixtures

fixtures 的概念前面已经有过简单的介绍，可以形象地把它看作是夹心饼干外层的两片饼干，这两片饼干就是 setUp/tearDown，中间的心就是测试用例。除此之外，unittest 还提供了更大范围的 fixtures，例如对于测试类和模块的 fixtures。

test.py

```
import unittest


def setUpModule():
    print("test module start >>>>>>>>>>>>>>>")


def tearDownModule():
    print("test module end >>>>>>>>>>>>>>>")


class Test(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        print("test class start =======>")

    @classmethod
    def tearDownClass(cls):
        print("test class end  =======>")

    def setUp(self):
        print("test case start -->")
```

```
    def tearDown(self):
        print("test case end -->")

    def test_case(self):
        print("test case1")

    def test_case2(self):
        print("test case2")


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

执行结果如下。

Python Shell

```
====================== RESTART: D:/test/csv_read.py ========================
test module start >>>>>>>>>>>>>>
test class start =======>
test case start -->
test case1
test case end -->
test case start -->
test case2
test case end -->
test class end  =======>
test module end >>>>>>>>>>>>>>
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

setUpModule/tearDownModule ： 在整个模块的开始与结束时被执行。

setUpClass/tearDownClass： 在测试类的开始与结束时被执行。

setUp/tearDown ： 在测试用例的开始与结束时被执行。

需要注意的是，setUpClass/tearDownClass 的写法稍微有些不同。首先，需要通过 @classmethod 进行装饰，其次方法的参数为 cls。其实，cls 与 self 并没有什么特别之处，

都只表示类方法的第一个参数，只是大家约定俗成，习惯于这样来命名，当然也可以用 abc 来代替。

7.3　带 unittest 的脚本分析

也许你现在心中还有疑问，unittest 框架与我们前面所编写的 Web 自动化测试之间有什么必然联系么？当然有，既然 unittest 可以组织、运行测试用例，为什么不能组织、运行 Web 自动化测试用例呢。我们先利用第 6 章的 Selenium IDE 来帮助我们理解它们之间的关系。你可能会问，这怎么又跟 Selenium IDE 扯上了关系？好吧，按照下面的步骤操作，你将会得到答案。

首先，通过 Selenium IDE 录制一个测试用例（本书第 6 章介绍了 Selenium IDE 的安装与使用），选择菜单栏“文件”→“Export Test Case As...”，如图 7.1 所示，在二级菜单中列出 Selenium IDE 所支持导出的编程语言、测试框架以及 Selenium 版本（WebDriver/Remote Control）。

Selenium IDE 录制脚本所支持的导出类型如下：

- 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 提供了多语言与测试框架的自动化脚本导出功能，对学习开发不同编程语言下的自动化测试脚本开发提供了很好的帮助与参考。

图 7.1　Selenium IDE 所支持的导出类型

因为我们当前使用的编程语言为 Python，单元测试框架为 unittest，自动测试脚本类型为 WebDriver，所以选择“Python2/unittest/WebDriver”选项，将脚本保存到指定位置。

需要说明的是，生成的代码为 Python 2，在 Python 3 的环境中并不能完全执行，不过，对于我们解读这个脚本影响不大。下面通过 Python ILDE 打开。

baidu.py

```
# -*- coding: utf-8 -*-
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import Select
from selenium.common.exceptions import NoSuchElementException
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/"
        self.verificationErrors = []
        self.accept_next_alert = True

    def test_baidu(self):
        driver = self.driver
        driver.get(self.base_url + "/")
        driver.find_element_by_id("kw").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([], self.verificationErrors)
```

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

相信读者现在再看这个脚本时已经不再感到陌生了，下面我们就来分析一下这些代码都做了哪些事情。

```
import unittest
```

首先引入 unittest 框架。

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

BaiduTest 类继承 unittest 框架的 TestCase 类成为标准的测试类。

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

setUp 用于设置初始化工作，在执行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()在前面已经学过，设置页面上元素的隐性等待时间为 30 秒。

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

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，这部分我们已经很熟悉了，这里不再解释。

```
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 方法用于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）。如果定位到元素则返回 True，否则抛出异常并返回 False。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()方法用于判断当前页面是否存在警告框，利用 WebDriver 提供的 switch_to_alert()方法来捕捉页面上警告框。如果捕捉到警告框则返回 True，否则将抛出 NoAlertPresentException 类型异常，并返回 False。

不过，经过笔者验证，不管页面是否出现警告框，返回结果都为 True。所以，可以调整该方法为 driver.switch_to_alert().text，用于获取当前页面上的警告提示信息。可以获取到就返回 True，获取不到则返回 False。

```
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()中已经初始化状态为 True，如果为 True，则通过 accept()接受警告，否则 dismiss()忽略此警告。

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

tearDown() 方法在每个测试方法执行后调用，这个方法用于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，如果为空则说明用例执行的过程中没有出现异常，否则将抛出 AssertionError 异常。

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

通过 unittest.main()方法来运行当前文件中的测试方法，其默认匹配并运行以 test 开头的方法。

7.4 编写 Web 测试用例

前面用了相当大的篇幅详细介绍了 unittest 单元测试框架，其目的是用它来运行 Web 自动化测试脚本。在此之前，需要简单规划一下测试目录：

```
test_project/
├── test_case/
│   ├── test_baidu.py
│   └── test_youdao.py
├── report/
│   └── login.txt
└── runtest.py
```

创建 Web 测试用例。

test_baidu.py

```
from selenium import webdriver
import unittest
import 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, "unittest_百度搜索")

    def tearDown(self):
        self.driver.quit()

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

test_youdao.py

```
from selenium import webdriver
import unittest
import 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, "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，并在测试文件中编写 Web 自动化测试用例。

runtest.py 文件的创建请参考 7.1.5 节中 runtest.py 的代码实现，唯一需要改动的就是指定新的测试目录为“./test_project/test_case”，之后就可以通过它来执行 test_case 目录下的测试用例了。

保存测试结果

你可能还有个疑问，report 目录是做什么的？也许从命名上你已经猜到它是用来存放测试报告的，那么怎样把测试结果生成一个有 log.txt 的文件呢？这里需要借助 dos 命令来实现。

首先打开 Windows 命令提示符，进入到.../test_project/目录下执行命令，如图 7.2 所示。

cmd.exe

```
> python runtest.py >> report/log.txt 2>&1
```

图 7.2　执行 runtest.py 文件

打开……/report/log.txt 文件，内容如下。

log.txt

```
..
----------------------------------------------------------------------
Ran 2 tests in 19.391s

OK
```

本章小结

本章详细学习了 Python 自带的单元测试框架 unittest，当然，学习本章的目的并不是为了编写单元测试，而是为了用它来编写 Web 自动化测试用例。利用其组织测试用例、断言预期结果以及批量执行测试用例等功能，可以很好地进行 Web 自动化测试的开发。

第 8 章
自动化测试高级应用

当学完第 7 章的内容后，就可以开始动手写你自己的自动化测试脚本了。其实，本章所谓的高级应用也并无什么高级之处，只是介绍的一些扩展库与技术可以让你的自动化测试做得更好。话不多说，下面看看本章都介绍了哪些干货。

8.1 HTML 测试报告

对软件测试人员来讲，测试的产出很难衡量。换句话说，测试人员的价值比较难以量化和评估，相信这一点对软件测试人员来说深有体会。我们花费了很多时间与精力所做的自动化测试也是如此。所以，需要一份漂亮且通俗易懂的测试报告来展示自动化测试成果。显然，一个简单的 Log 文件是不够的。

HTMLTestRunner 是 Python 标准库 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 许可证下发布的。

下载地址如下：

http://tungwaiyip.info/software/HTMLTestRunner.html

这个扩展非常简单，只有一个 HTMLTestRunner.py 文件，选中后单击鼠标右键，在弹出的快捷菜单中选择目标另存为，将它保存到本地。安装方法也很简单，将其复制到 Python 安装目录下即可。

Windows：将下载的文件保存到...\Python35\Lib 目录下。

Linux：以 Ubuntu 为例，首先需要打开终端，找到 Python 的安装目录。打开终端后，输入 Python 命令进入 Python 交互模式，通过 sys.path 可以查看本机 Python 的安装目录。

Ubuntu 终端

```
fnngj@fnngj-pc:~$ Python3
Python 3.4.3 (default, Oct 14 2015, 20:28:29)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', '/usr/lib/python3.4', '/usr/lib/python3.4/plat-x86_64-linux-gnu',
'/usr/lib/python3.4/lib-dynload', '/usr/local/lib/python3.4/dist-packages',
'/usr/lib/python3/dist-packages']
```

以 root 身份将 HTMLTestRunner.py 文件复制到/usr/local/Python3.4/dist-packages/ 目录下。

Ubuntu 终端

```
root@fnngj:/home/user/test# cp HTMLTestRunner.py  /usr/local/Python3.4/dist
-packages/
```

提示：默认情况下，不允许普通用户向./dist-packages/目录下复制文件，我们可通过 root 用户执行该操作，或者为该目录添加权限。

在 Python 交互模式下引入 HTMLTestRunner 模块，如果系统没有报错，则说明添加成功。

Python Shell

```
>>> import HTMLTestRunner
>>>
```

8.1.1　修改 HTMLTestRunner

因为 HTMLTestRunner.py 是基于 Python 2 开发的，为了使其支持 Python 3 的环境，需要对其中的部分内容进行修改。下面通过编辑器打开 HTMLTestRunner.py 文件。

HTMLTestRu...

```
# 第 94 行
import StringIO
修改为：
```

```
import io

#第 539 行
self.outputBuffer = StringIO.StringIO()
修改为:
self.outputBuffer = io.StringIO()

第 631 行
print >>sys.stderr, '\nTime Elapsed: %s' % (self.stopTime-self.startTime)
修改为:
print(sys.stderr, '\nTime Elapsed: %s' % (self.stopTime-self.startTime))

#第 642 行
if not rmap.has_key(cls):
修改为:
if not cls in rmap:


第 766 行
uo = o.decode('latin-1')
修改为:
uo = e

第 772 行
ue = e.decode('latin-1')
修改为:
ue = e
```

8.1.2 生成 HTML 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

```
from selenium import webdriver
import unittest
from HTMLTestRunner import HTMLTestRunner


class Baidu(unittest.TestCase):

    def setUp(self):
```

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

    def test_baidu_search(self):
        driver = self.driver
        driver.get(self.base_url)
        driver.find_element_by_id("kw").send_keys("HTMLTestRunner")
        driver.find_element_by_id("su").click()

    def tearDown(self):
        self.driver.quit()

if __name__ == "__main__":

    testunit = unittest.TestSuite()
    testunit.addTest(Baidu("test_baidu_search"))

    # 定义报告存放路径
    fp = open('./result.html', 'wb')
    # 定义测试报告
    runner = HTMLTestRunner(stream=fp,
                            title='百度搜索测试报告',
                            description='用例执行情况：')

    runner.run(testunit)  # 运行测试用例
    fp.close()  # 关闭报告文件
```

代码分析

首先，将 HTMLTestRunner 模块用 import 导入进来。

其次，通过 open()方法以二进制写模式打开当前目录下的 result.html，如果没有，则自动创建该文件。

接着，调用 HTMLTestRunner 模块下的 HTMLTestRunner 类。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。

最后，通过 HTMLTestRunner 的 run()方法来运行测试套件中所组装的测试用例。最后通过 close()关闭测试报告文件。

用例运行完成，打开当前目录下的“result.html”文件查看生成的测试报告，如图 8.1 所示。

百度搜索测试报告

Start Time: 2015-07-30 23:17:46
Duration: 0:00:08.357000
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 8.1 测试报告

8.1.3 更易读的测试报告

现在生成的测试报告还不易读，因为它只罗列了一堆测试类和测试方法，我们需要用心地为测试类和测试方法命名才能提高测试报告的可读性。如果随意命名为“test_case1”、“test_case2”等，那么这份报告就失去了可读性，也许时间久了连脚本开发者都不清楚“test_case1”是测试什么功能了。

在编写功能测试用例时，每条测试用例都有标题，那么我们能不能也为自动化测试用例加上标题呢？在此之前我们先来学习另外一个知识点：Python 的注释。Python 的注释有两种，一种叫 comment，另一种叫 doc string，前者为普通的注释，后者用于函数、类和方法的描述。

Python Shell

```
>>> def add(a, b):
        "add()函数需要两个入参，并返回两个参数相加的值。"
        return a + b

>>> add(2, 4)
6
>>> help(add)
Help on function add in module __main__:

add(a, b)
```

add()函数需要两个入参，并返回两个参数相加的值。

在类或方法的下方，通过三引号（"""""" 或 '''''）来添加 doc string 类型的注释，这类注释在平时调用的时候不显示，可以通过 help()方法来查看类或方法的这种注释。

回到问题的原点，HTMLTestRunner 可以读取 doc string 类型的注释。所以，我们只需给测试类或方法添加这种类型的注释即可。

baidu.py

```
# ……
class Baidu(unittest.TestCase):
    '''百度搜索测试'''

# ……
    def test_baidu_search(self):
        '''搜索关键字：HTMLTestRunner'''
# ……
```

再次运行测试用例，查看测试报告，如图 8.2 所示。

百度搜索测试报告

Start Time: 2015-10-16 15:08:51

Duration: 0:00:17.379000

Status: Pass 1

用例执行情况：

Show Summary Failed All

Test Group/Test case	Count	Pass	Fail	Error	View
Baidu: 百度搜索测试	1	1	0	0	Detail
test_baidu_search: 搜索关键字：HTMLTestRunner	pass				
Total	1	1	0	0	

图 8.2　易读的测试报告

8.1.4　测试报告文件名

在每次运行测试之前，都要手动修改报告的名称，如果忘记修改，就会把之前的报告

覆盖，这样做显然很麻烦，那么有没有办法使每次生成的报告名称都不重复并且有意义？最好的方法是在报告名称中加入当前时间，这样生成的报告既不会重叠，又能更清晰地知道报告的生成时间。

Python 的 time 模块中提供了丰富的关于时间操作的方法，可以利用这些方法来完成这个需求。

Python Shell

```
>>> import time
>>> time.time()
1445694559.2290168

>>> time.ctime()
'Sat Oct 24 21:49:29 2015'

>>> time.localtime()
time.struct_time(tm_year=2015, tm_mon=10, tm_mday=24, tm_hour=21, tm_min=49,
tm_sec=49, tm_wday=5, tm_yday=297, tm_isdst=0)

>>> time.strftime("%Y_%m_%d %H:%M:%S")
'2015_10_24 21:50:15'
```

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）如表 8.1 所示。

表 8.1　Python 中时间日期格式化符号

指　令	含　义
%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

（续）

指　令	含　义
%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
%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

继续打开测试用例，做如下修改。

test_baidu.py

```
import time

# ……

if __name__ == "__main__":

    testunit = unittest.TestSuite()
    testunit.addTest(Baidu("test_baidu_search"))

    # 按照一定格式获取当前时间
    now = time.strftime("%Y-%m-%d %H_%M_%S")

    # 定义报告存放路径
    filename = './' + now + 'result.html'

    fp = open(filename, 'wb')
```

```
runner = HTMLTestRunner(stream=fp,
                        title='百度搜索测试报告',
                        description='用例执行情况：')

runner.run(testunit)
fp.close()
```

通过 strftime()方法以指定的格式获取当前时间，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 8.3 所示。

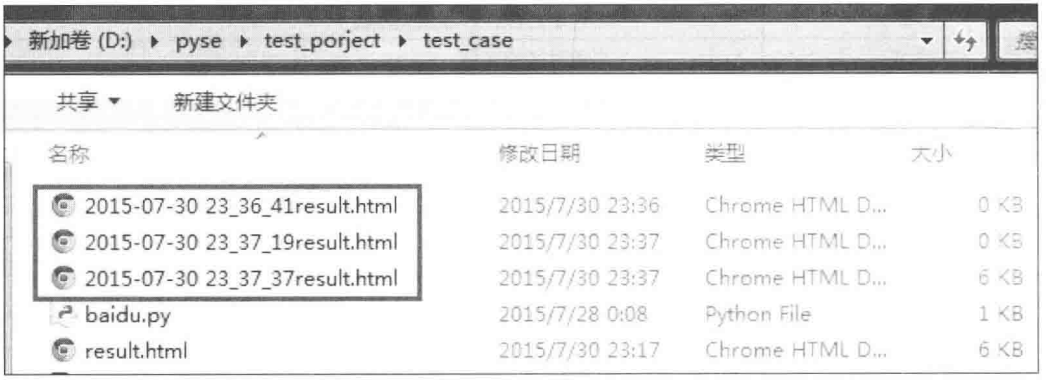

图 8.3　测试目录

8.1.5　项目集成测试报告

目前 HTMLTestRunner 只是针对单个测试文件生成测试报告，我们的最终目的是希望将它集成到 runtest.py 文件中，使其作用于整个测试项目。下面打开 runtest.py 文件进行修改。

runtest.py

```
import unittest, time
from HTMLTestRunner import HTMLTestRunner

# 指定测试用例为当前文件夹下的 test_case 目录
test_dir = './test_case'
```

```
discover = unittest.defaultTestLoader.discover(test_dir,pattern ='test_*.py')


if __name__ == '__main__':

    now = time.strftime("%Y-%m-%d %H_%M_%S")
    filename = test_dir + '/' + now + 'result.html'
    fp = open(filename, 'wb')
    runner = HTMLTestRunner(stream=fp,
                            title='测试报告',
                            description='用例执行情况：')
    runner.run(discover)
    fp.close()
```

生成的 HTML 测试报告如图 8.4 所示。

测试报告

Start Time: 2015-07-30 23:55:04

Duration: 0:00:21.158000

Status: Pass 2

用例执行情况：

Show Summary Failed All

Test Group/Test case	Count	Pass	Fail	Error	View
test_baidu.Baidu: 测试百度搜索	1	1	0	0	Detail
test_baidu: 搜索关键字：unittest			pass		
test_youdao.Youdao: 测试有道搜索	1	1	0	0	Detail
test_youdao: 搜索关键字：webdriver			pass		
Total	**2**	**2**	**0**	**0**	

图 8.4　项目测试报告

8.2　自动发邮件功能

自动发邮件功能也是自动化测试项目的重要需求之一。例如，我们想在自动化脚本运行完成之后，邮箱就可以收到最新的测试报告结果。假设生成的测试报告与多人相关，每个人都去测试服务器查看就会比较麻烦，如果把这种主动的且不及时的查看变成被动且及时的查收，就方便多了。

SMTP（Simple Mail Transfer Protocol）是简单邮件传输协议，它是一组用于由源地址

到目的地址传送邮件的规则，由它来控制信件的中转方式。

Python 的 smtplib 模块提供了一种很方便的途径用来发送电子邮件。它对 SMTP 协议进行了简单的封装。我们可以使用 SMTP 对象的 sendmail 方法发送邮件，通过 help()查看 SMTP 所提供的方法如下。

Python Shell

```
>>> from smtplib import SMTP
>>> help(SMTP)
Help on class SMTP in module smtplib:

 | connect(self, host='localhost', port=0)
 |     Connect to a host on a given port.
 |
 |     If the hostname ends with a colon (`:') followed by a number, and
 |     there is no port specified, that suffix will be stripped off and the
 |     number interpreted as the port number to use.
......

 | login(self, user, password)
 |     Log in on an SMTP server that requires authentication.
 |
 |     The arguments are:
 |         - user:     The user name to authenticate with.
 |         - password: The password for the authentication.
......

 | quit(self)
 |     Terminate the SMTP session.
......

 | sendmail(self, from_addr, to_addrs, msg, mail_options=[], rcpt_options=[])
 |     This command performs an entire mail transaction.
 |
 |     The arguments are:
 |         - from_addr    : The address sending this mail.
 |         - to_addrs     : A list of addresses to send this mail to.  A bare
 |                          string will be treated as a list with 1 address.
 |         - msg         : The message to send.
 |         - mail_options : List of ESMTP options (such as 8bitmime) for the
 |                          mail command.
 |         - rcpt_options : List of ESMTP options (such as DSN commands) for
```

```
|                    all the rcpt commands.
|
|     If there has been no previous EHLO or HELO command this session, this
|     method tries ESMTP EHLO first.  If the server does ESMTP, message size
|     and each of the specified options will be passed to it.  If EHLO
|     fails, HELO will be tried and ESMTP options suppressed.

......
```

导入 SMTP 对象，通过 help()查看对象的注释，从中找到 sendmail()方法的使用说明。

connect(host,port)方法参数说明如下。

- host:　　指定连接的邮箱服务器。
- port:　　指定连接服务器的端口号。

login(user, password)方法参数说明如下。

- user:　　登录邮箱用户用。
- password:　　登录邮箱密码。

sendmail(from_addr, to_addrs, msg,..)方法参数说明如下。

- from_addr:　　邮件发送者地址。
- to_addrs:　　字符串列表，邮件发送地址。
- msg:　　发送消息。

quit()方法：用于结束 SMTP 会话。

一般我们发邮件时有两种方式。方式一：自己邮箱的 Web 页面（如 mail.126.com），输入自己邮箱的用户名和密码登录，打开发邮件页面，填写对方的邮箱地址及邮件标题与正文，完成后单击发送。方式二：下载安装邮箱客户端（如 Outlook、Foxmail 等），填写邮箱账号、密码及邮箱服务器（如 smtp.126.com），一般的邮箱客户端会默认记下这些信息，所以，这个过程只需填写一次，后面发邮件的过程与方法一相同。

而我们通过 Python 的 SMTP 对象发邮件则更像方式二，因为需要填写邮箱服务器。

当然，在具体发邮件时会涉及诸多需求，例如，邮件的正文的格式、是否带图片、邮件

是否需要添加附件（及多附件）、邮件是否需要同时向多人发送等。

8.2.1 发送 HTML 格式的邮件

send_mail.py

```
import smtplib
from email.mime.text import MIMEText
from email.header import Header

# 发送邮箱服务器
smtpserver = 'smtp.sina.com'
# 发送邮箱用户/密码
user = 'username@sina.com'
password = '123456'
# 发送邮箱
sender = 'username@sina.com'
# 接收邮箱
receiver = 'receive@126.com'
# 发送邮件主题
subject = 'Python email test'

# 编写 HTML 类型的邮件正文
msg = MIMEText('<html><h1>你好! </h1></html>','html','utf-8')
msg['Subject'] = Header(subject, 'utf-8')


# 连接发送邮件
smtp = smtplib.SMTP()
smtp.connect(smtpserver)
smtp.login(user, password)
smtp.sendmail(sender, receiver, msg.as_string())
smtp.quit()
```

本例中，除 SMTP 模块外，我们还用到了 email 模块，它主要用来定义邮件的标题和正文；Header()方法用来定义邮件标题；MIMEText()用于定义邮件正文，参数为 html 格式的文本。登录 receive@126.com 邮箱，查看邮件内容如图 8.5 所示。

图 8.5　阅读 HTML 格式邮件

8.2.2　发送带附件的邮件

在发送文件时，有时需要发送附件。下面的实例实现了带附件的邮件发送。

send_mail.py

```
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart

# 发送邮箱服务器
smtpserver = 'smtp.sina.com'
# 发送邮箱
sender = 'username@sina.com'
# 接收邮箱
receiver = 'receiver@126.com'
# 发送邮箱用户/密码
user = 'username@sina.com'
password = '123456'
# 邮件主题
subject = 'Python send email test'
# 发送的附件
sendfile = open('D:\\testpro\\report\\log.txt', 'rb').read()

att = MIMEText(sendfile, 'base64', 'utf-8')
att["Content-Type"] = 'application/octet-stream'
att["Content-Disposition"] = 'attachment; filename="log.txt"'

msgRoot = MIMEMultipart('related')
```

```
msgRoot['Subject'] = subject
msgRoot.attach(att)

smtp = smtplib.SMTP()
smtp.connect(smtpserver)
smtp.login(user, password)
smtp.sendmail(sender, receiver, msgRoot.as_string())
smtp.quit()
```

相比上一个实例，通过 MIMEMultipart()模块构造的带附件的邮件如图 8.6 所示。

图 8.6 带附件的邮件

8.2.3 查找最新的测试报告

现在已经知道如何通过 Python 编写发邮件程序，但要想和自动化测试项目结合还需要解决一个问题，因为测试报告的名称是根据当前时间生成的，所以如何找到最新生成的测试报告是实现发邮件功能的关键。

find_file.py

```
import os

# 定义文件目录
result_dir = 'D:\\testpro\\report'

lists = os.listdir(result_dir)

# 重新按时间对目录下的文件进行排序
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 ，os.listdir()可以获取目录下的所有文件及文件夹。利用 sort()方法对目录下的文件及文件夹按时间重新排序。list[-1]取到的就是最新生成的文件或文件夹。程序运行结果如下。

Python Shell

```
======================= RESTART: D:/test/find_file.py =======================
最新的文件为： 2015-10-24 22_45_25result.html
D:\testpro\report\2015-10-24 22_45_25result.html
```

8.2.4　整合自动发邮件功能

解决了前面的问题后，现在就可以将自动发邮件功能集成到自动化测试项目中了。下面打开 runtest.py 文件重新进行编辑。

runtest.py

```
from HTMLTestRunner import HTMLTestRunner
from email.mime.text import MIMEText
from email.header import Header
import smtplib
import unittest
import time
import os


# =============定义发送邮件==========
def send_mail(file_new):
    f = open(file_new, 'rb')
    mail_body = f.read()
    f.close()

    msg = MIMEText(mail_body, 'html', 'utf-8')
    msg['Subject'] = Header("自动化测试报告", 'utf-8')

    smtp = smtplib.SMTP()
```

```
    smtp.connect("smtp.126.com")
    smtp.login("useranem@126.com", "123456")
    smtp.sendmail("username@126.com", "receive@126.com", 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 new_report(testreport):
    lists = os.listdir(testreport)
    lists.sort(key=lambda fn: os.path.getmtime(testreport + "\\" + fn))
    file_new = os.path.join(testreport, lists[-1])
    print(file_new)
    return file_new


if __name__ == '__main__':

    test_dir = 'D:\\testpro\\test_case'
    test_report = 'D:\\testpro\\report'

    discover = unittest.defaultTestLoader.discover(test_dir,
                                                   pattern='test_*.py')
    now = time.strftime("%Y-%m-%d_%H_%M_%S")
    filename = test_report + '\\' + now + 'result.html'
    fp = open(filename, 'wb')
    runner = HTMLTestRunner(stream=fp,
                            title='测试报告',
                            description='用例执行情况：')
    runner.run(discover)
    fp.close()

    new_report = new_report(test_report)
    send_mail(new_report)   #发送测试报告
```

整个程序的执行过程可以分为三个步骤：

① 通过 unittest 框架的 discover()找到匹配测试用例，由 HTMLTestRunner 的 run()方法执行测试用例并生成最新的测试报告。

② 调用 new_report()函数找到测试报告目录（report）下最新生成的测试报告，返回测试报告的路径。

③ 将得到的最新测试报告的完整路径传给 send_mail()函数，实现发邮件功能。

整个脚本执行完成后，打开接收邮箱，即可看到最新测试执行的测试报告，如图 8.7 所示。

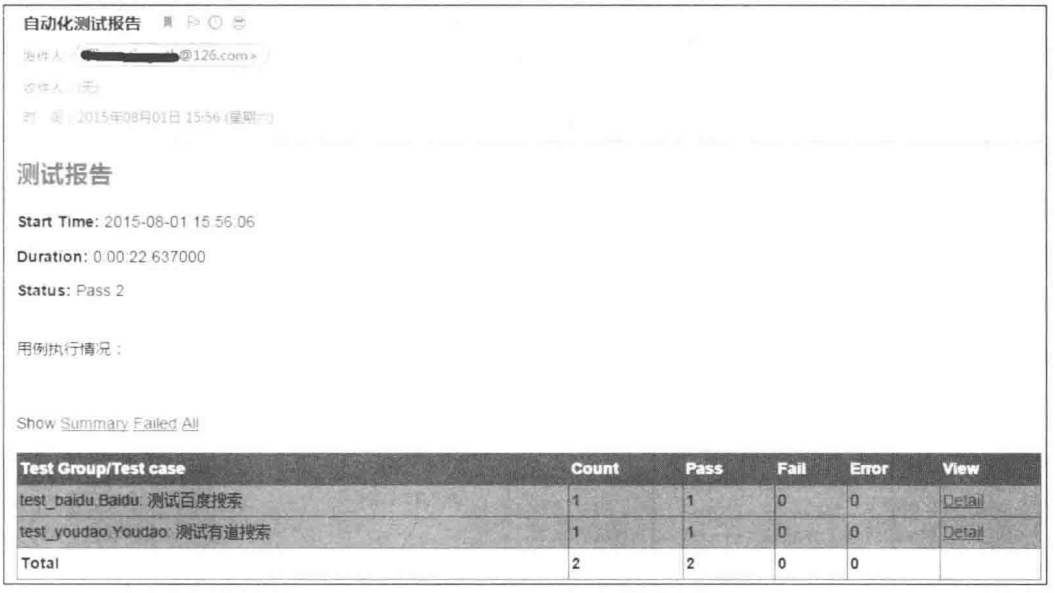

图 8.7　查看邮箱中的自动化测试报告

8.3　Page Object 设计模式

Page Object 是 Selenium 自动化测试项目开发实践的最佳设计模式之一，它主要体现在对界面交互细节的封装，这样可以使测试案例更关注于业务而非界面细节，从而提高测试案例的可读性。

8.3.1　认识 Page Object

Page Object 设计模式的优点如下：

- 减少代码的重复。
- 提高测试用例的可读性。
- 提高测试用例的可维护性，特别是针对 UI 频繁变化的项目。

当为 Web 页面编写测试时，需要操作该 Web 页面上的元素。然而，如果在测试代码中直接操作 HTML 元素，那么你的代码是极其脆弱的，因为 UI 经常变动。我们可以将一个

page 对象封装成一个 HTML 页面，然后通过提供的应用程序特定的 API 来操作页面元素，而不是在 HTML 中四处搜寻。Page Object 原理如图 8.8 所示。

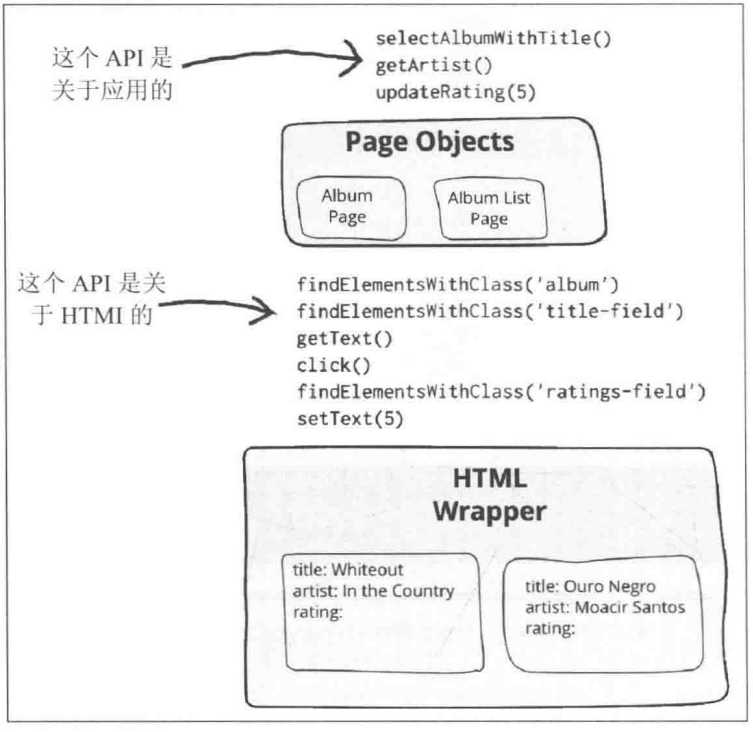

图 8.8　Page Object 原理

page 对象的一个基本经验法则是：凡是人能做的事，page 对象通过软件客户端都能够做到。因此，它也应当提供一个易于编程的接口并隐藏窗口中底层的部件。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，即使改变具体的控件，page 对象的接口也不应当发生变化。

尽管该术语是“页面”对象，但并不意味着需要针对每个页面建立一个这样的对象，例如，页面有重要意义的元素可以独立为一个 page 对象。经验法则的目的是通过给页面建模，使其对应用程序的使用者变得有意义。

8.3.2 Paget Object 实例

下面以登录 126 邮箱为例，通过 Page Object 设计模式来实现。

po_model.py

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


class Page(object):
    '''
    基础类，用于页面对象类的继承
    '''

    login_url = 'http://www.126.com'

    def __init__(self, selenium_driver, base_url=login_url):
        self.base_url = base_url
        self.driver = selenium_driver
        self.timeout = 30

    def on_page(self):
        return self.driver.current_url == (self.base_url + self.url)

    def _open(self, url):
        url = self.base_url + url
        self.driver.get(url)
        assert self.on_page(), 'Did not land on %s' % url

    def open(self):
        self._open(self.url)

    def find_element(self, *loc):
        return self.driver.find_element(*loc)


class LoginPage(Page):
    '''
    126 邮箱登录页面模型
    '''
```

```
    url = '/'

    # 定位器
    username_loc = (By.ID, "idInput")
    password_loc = (By.ID, "pwdInput")
    submit_loc = (By.ID, "loginBtn")

    # Action
    def 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()


def main():
    try:
        driver = webdriver.Firefox()
        username = 'username'
        password = '123456'
        test_user_login(driver, username, password)
        sleep(3)
        text = driver.find_element_by_xpath("//span[@id='spnUid']").text
        assert(text == 'username@126.com'), "用户名称不匹配，登录失败！"
    finally:
        # 关闭浏览器窗口
        driver.close()
```

```
if __name__ == '__main__':
    main()
```

Page Object 设计模式的实现方法显然使结构变得复杂了很多。下面我们对其进行逐段分析，来体会这样设计的好处。

1. 创建 page 类

po_model.py

```
# ……
class Page(object):
    '''
    基础类，用于所页面的继承
    '''

    login_url = 'http://www.126.com'

    def __init__(self, selenium_driver, base_url=login_url):
        self.base_url = base_url
        self.driver = selenium_driver
        self.timeout = 30

    def on_page(self):
        return self.driver.current_url == (self.base_url + self.url)

    def _open(self, url):
        url = self.base_url + url
        self.driver.get(url)
        assert self.on_page(), 'Did not land on %s' % url

    def open(self):
        self._open(self.url)

    def find_element(self, *loc):
        return self.driver.find_element(*loc)
# ……
```

首先创建一个基础类 Page，在初始化方法__init__()中定义驱动（driver）、基本的 URL（base_url）和超时时间（timeout）等。

定义 open()方法用于打开 URL 网站，但它本身并未做这件事情，而是交由_open()方法来实现。关于 URL 地址的断言部分，则交由 on_page()方法来实现，而 find_element()方法用于元素的定位。

2. 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。下面根据登录页的特点再创建 LoginPage 类并继承 Page 类，这也是 Page Object 设计模式中最重要的对象层。

po_model.py

```
# ……
class LoginPage(Page):
    '''
    126 邮箱登录页面模型
    '''

    url = '/'

    # 定位器
    username_loc = (By.ID, "idInput")
    password_loc = (By.ID, "pwdInput")
    submit_loc = (By.ID, "loginBtn")

    # Action
    def 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()
# ……
```

LoginPage 类中主要对登录页面上的元素进行封装，使其成为更具体的操作方法。例如，用户名、密码和登录按钮都被封装成了方法。

3. 创建 test_user_login()函数

po_model.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()
……
```

test_user_login()函数将单个的元素操作组成一个完整的动作，而这个动作包含了打开浏览器、输入用户名/密码、点击登录等单步操作。在使用该函数时需要将 driver、username、password 等信息作为函数的入参，这样该函数具有很强的可重用性。

4. 创建 main()函数

po_model.py

```
# ……
def main():
    try:
        driver = webdriver.Firefox()
        username = 'username'
        password = '123456'
        test_user_login(driver, username, password)
        sleep(3)
        text = driver.find_element_by_xpath("//span[@id='spnUid']").text
        assert(text == 'username@126.com'), "用户名称不匹配，登录失败！"
    finally:
        # 关闭浏览器窗口
        driver.close()


if __name__ == '__main__':
    main()
```

main()函数更接近于用户的操作行为。对用户来说，要进行邮箱的登录，需要关心的就是通过哪个浏览器打开邮箱网址、登录的用户名和密码是什么，至于输入框、按钮是如何定位的，则不需要关心。

这样分层的好处是，不同的层关心不同的问题。页面对象层只关心元素的定位问题，测试用例只关心测试的数据。

一个有分歧的地方是 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，可以更容易地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。

笔者赞成在 page 对象中不包含断言，虽然我们可以通过为常用的断言提供断言库的方式来消除重复，提供更好的诊断，但从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，不应该交由页面对象层来完成。

使用 Page Object 模式之后的另一个好处就是有助于降低冗余。如果需要在 10 个用例中输入不同的用户名/密码登录，那么用 main()方法写将会变得非常简洁。

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，因为你不需要和开发、业务交流案例，也不会写很多重复的动作。但是，当你真正开始尝试 ATDD 或 BDD，当你开始写一些重要的异常分支流程时，当你开始为新需求频繁维护修改案例时，就会意识到 Page Object 的作用。

最后，Page Object 不是万灵药，也不是唯一方案，提高测试案例的可读性，避免案例步骤冗余才是终极目标。

本章小结

本章详细介绍了三个知识点，分别为：利用 HTMLTestRunner 生成测试报告、集成自动发邮件功能和 Page Object 设计模式，虽然它们之间并没有直接的联系，但将它们合理地运行到自动化测试项目中将有助于自动化测试项目的开展。

第 9 章
Selenium Grid2

在 Selenium 家族中，我们已经学习了 WebDriver 和 Selenium IDE ，本章将介绍最后一位成员——Selenium Grid。利用 Selenium Grid 可以在不同的主机上建立主节点（hub）和分支节点（node），可以使主节点上的测试用例在不同的分支节点上运行。对不同的节点来说，可以搭建不同的测试环境（操作系统、浏览器），从而使一份测试用例得到不同环境下的执行结果。

Selenium Grid 版本

Selenium Grid（以下简称 Grid）同样分为两个版本：Grid1 和 Grid2，其实它的两个版本并非对应于 Selenium 的两个大版本发布（即 Grid2 的出现要晚于 Selenium 2）。不过幸运的是，现在的 Selenium Grid2 完全能支持 Selenium 2 的所有功能。

Grid 的两个版本的原理和基本工作方式都是一样的，只是 Grid2 同时支持 Selenium 1 和 Selenium 2 两种协议，并且在一些小的功能和易用性上进行了优化，例如，指定测试平台的方式等。

Grid2 不再提供单独的包，其功能已经集成到 Selenium Server 中，所以，需要下载和运行 Selenium Server 才可以使用 Grid2 的功能。

9.1 Selenium Server 环境配置

下面下载、配置并运行 Selenium Server。

① 下载 Selenium Server。

下载地址为：http://www.seleniumhq.org/download/

通过浏览器打开页面，找到 Selenium Standalone Server 的介绍，单击版本号链接下载，例如 Download version_2.48.2，下载完成后将得到 selenium-server-standalone-xxx.jar。由于该 jar 包由 Java 开发，所以对于 jar 包的运行需要 Java 环境。

② 配置 Java 环境。

Java 下载地址为：http://www.java.com/zh_CN/download/manual.jsp

小知识 Java 环境分为 JDK 和 JRE 两种。JDK 的全称为 Java Development Kit，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 的全称为 Java Runtime Environment，是指 Java 的运行环境，面向的是 Java 程序的使用者，而不是开发者。

根据本机操作系统环境选择相应的版本进行下载，下面以在 Windows 下安装 JDK 为例进行介绍。

双击下载的 JDK，并设置安装路径。这里选择默认安装到 C:\Program Files\Java\jdk1.7.0_60\目录下，如图 9.1 所示。

图 9.1 安装 JDK

安装完成后，设置环境变量，右击“计算机”，在弹出的右键单中单击属性→高级系统设置→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：CALSS_PATH

变量值：.;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar;

找到“path”变量名→编辑，追加：

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 环境是否配置成功。

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
```

……

“java”命令用于运行 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。

③ 运行 Selenium Server。

现在可以通过“java”命令运行 Selenium Server 了。切换到 Selenium Server 所在目录下并启动，如图 9.2 所示。在 Windows 命令提示符（或 Linux 终端）下启动 Selenium Server。

```
> java -jar selenium-server-standalone-2.47.0.jar
```

图 9.2　启动 Selenium Server

本书的第 1 章中提到 Selenium1.0 RC 脚本的执行依赖于 Selenium Server，如果读者感兴趣，可以运行下面的 RC 脚本来体会它与 Web Driver 的不同。

sel_rc.py

```
from selenium import selenium

sel = selenium("localhost", 4444, "*firefox", "http://www.baidu.com/")

sel.start()
```

```
sel.open("/")
sel.type("id=kw", "selenium grid")
sel.click("id=su")
sel.wait_for_page_to_load("30000")

sel.stop()
```

9.2　Selenium Grid 工作原理

Grid 是用于设计帮助我们进行分布式测试的工具，其整个结构由一个 hub 主节点和若干个 node 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，并且接收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 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
```

上面的代码分别启动了一个 hub 和一个 node，hub 默认端口号为 4444，node 默认端口号为 5555。若是同一台主机上要启动多个 node，则需要注意指定端口号，可以通过下面的方式来启动多个 node 点节。

```
> 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 Grid 的基本结构图如图 9.3 所示。

当你的测试用例需要验证的环境比较多时，可以并行地执行这些用例进而缩短测试总耗时。并行的能力需要借助编程语言的多线程技术，本书第 11 章将会介绍 Python 的多线程技术。Grid 可以根据用例中指定的平台配置信息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 上用 Firefox 版本进行测试，那么 Grid 会自动匹配注册信息为 Linux 且安装了 Firefox 的代理节点，如果匹配成功，则转发测试请求，如果匹配失败则拒绝请求。调用的基本结构图如图 9.4 所示。

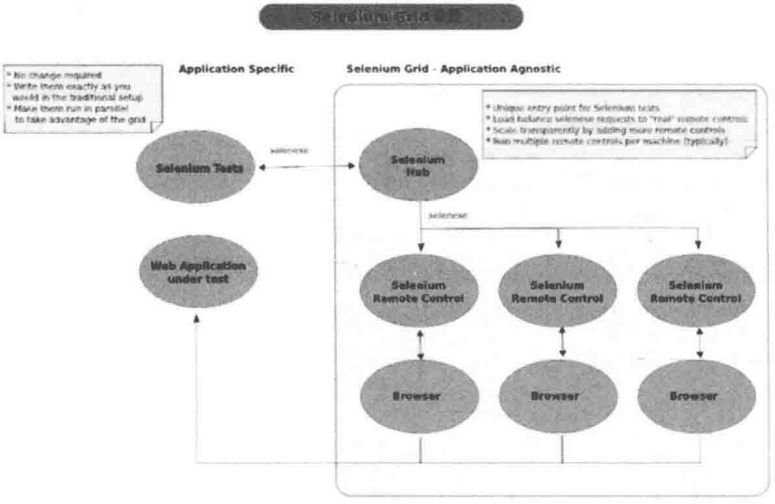

图 9.3　Selenium Grid 设置

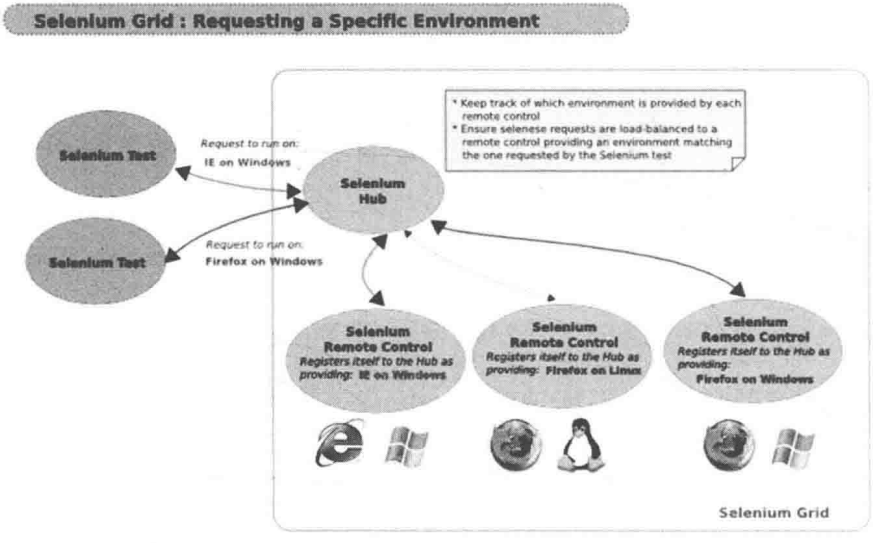

图 9.4　请求一个特定的环境

下面在同一台主机上启动一个 hub 主节点和两个 node 分支节点，如图 9.5 所示。

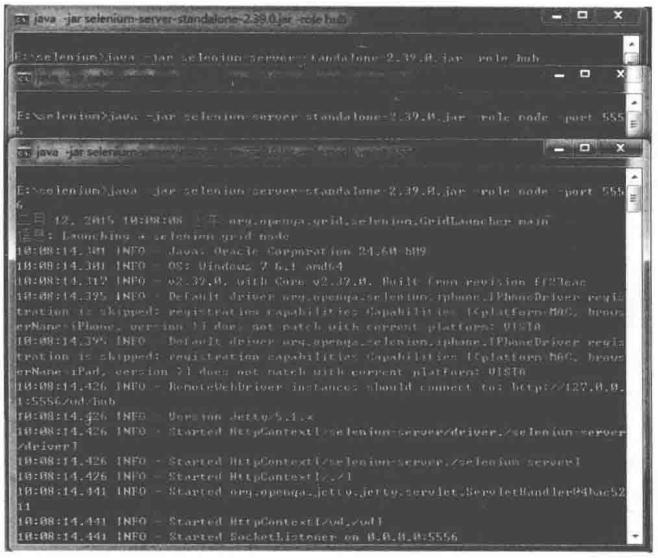

图 9.5　启动 hub 和 node

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点信息，如图 9.6 所示。

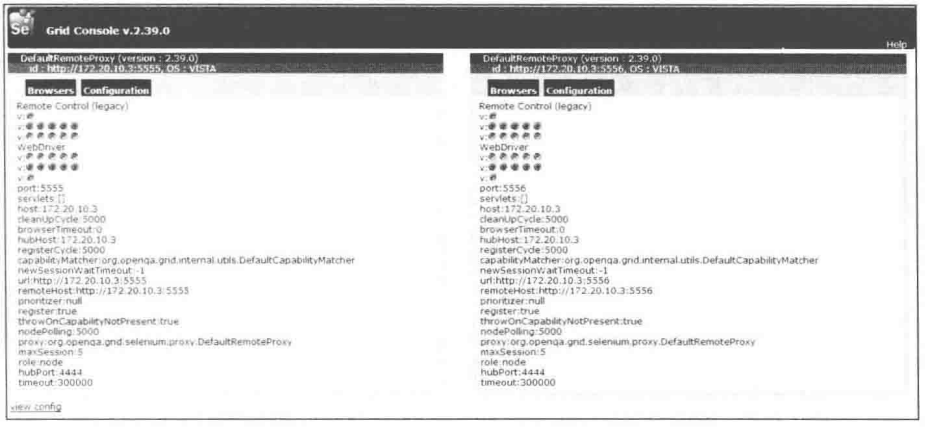

图 9.6　node 节点详细信息

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
```

9.3 Remote 应用

要解释清楚 Remote 的作用并不太容易，不过我们可以通过分析 Selenium 代码的方式来理解它的作用。我们知道 WebDriver 支持多浏览器下的执行，这是因为 WebDriver 针对每一种浏览器驱动都重写 WebDriver 方法。所以，在脚本运行之前需要先确定浏览器驱动，具体如下：

```
driver = webdriver.Firefox()

driver = webdriver.Chrome()

driver = webdriver.Ie()
```

下面就对这些驱动进行简单分析。

9.3.1　WebDriver 驱动分析

在 selenium 包的 webdriver 目录下可以看到如图 9.7 所示的目录结构。

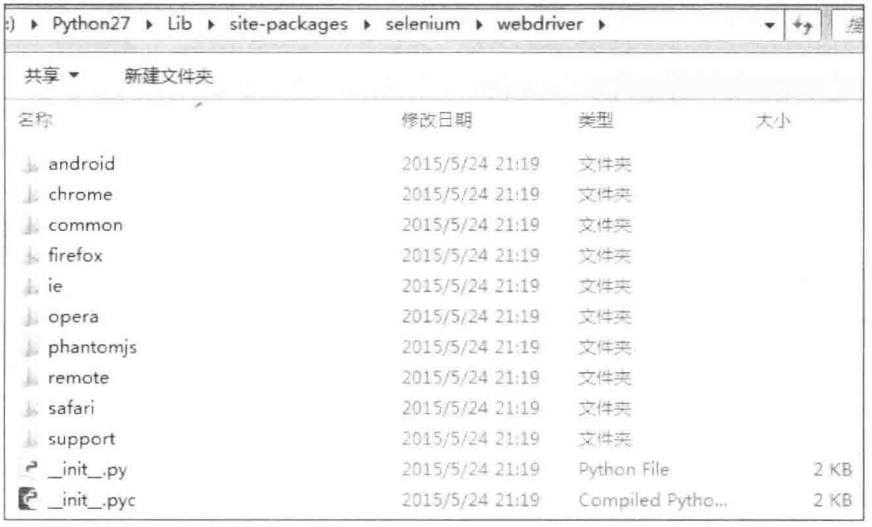

图 9.7　webdriver 目录结构

查看其中任何一个驱动的目录发现都有一个 webdriver.py 文件，除了我们熟悉的 Firefox、Chrome、IE 等驱动外，其中还包括非常重要的 remote。从这个角度看，也可以把它看作是一种驱动类型，而这种驱动类型比较特别，它不是支持某一款特定的浏览器或平台，而是一种配置模式，我们在这种配置模式下指定任意的平台或浏览器，这种模式的执行都需要 Selenium Server 的支持。

打开 selenium 包下的 webdriver/firefox 目录，先看 Fireofox 中 webdriver.py 文件的实现。

webdriver.py

```
……
from .firefox_binary import FirefoxBinary
from selenium.webdriver.firefox.firefox_profile import FirefoxProfile
from selenium.webdriver.remote.webdriver import WebDriver as RemoteWebDriver
……
class WebDriver(RemoteWebDriver):

    # There is no native event support on Mac
    NATIVE_EVENTS_ALLOWED = sys.platform != "darwin"
```

```
    def __init__(self, firefox_profile=None, firefox_binary=None, timeout=30,
                 capabilities=None, executable_path='wires'):

        self.binary = firefox_binary
        self.profile = firefox_profile
……
```

主机查看 WebDriver 类的__init__()初始化方法，因为 Selenium 自带 Firefox 浏览器驱动，所以，这个驱动的重要配置在于 firefox_profile 和 firefox_binary 两个参数。而这两个参数分别调用当前目录下的 firefox_binary.py 和 firefox_profile.py 文件，感兴趣的读者可以进一步研究这两个文件的实现。

我们在脚本中调用 Firefox 浏览器驱动时的路径为：selenium.webdriver.Firefox()，那么，它是如何指向../selenium/webdriver/firefox/webdriver.py 文件中 WebDriver 类的呢？秘密在于../selenium/webdriver/目录下的__init__.py 文件。

__init__.py

```
……
from .firefox.webdriver import WebDriver as Firefox
from .firefox.firefox_profile import FirefoxProfile
from .chrome.webdriver import WebDriver as Chrome
from .chrome.options import Options as ChromeOptions
from .ie.webdriver import WebDriver as Ie
from .edge.webdriver import WebDriver as Edge
from .opera.webdriver import WebDriver as Opera
from .safari.webdriver import WebDriver as Safari
from .blackberry.webdriver import WebDriver as BlackBerry
from .phantomjs.webdriver import WebDriver as PhantomJS
from .android.webdriver import WebDriver as Android
from .remote.webdriver import WebDriver as Remote
from .common.desired_capabilities import DesiredCapabilities
from .common.action_chains import ActionChains
from .common.touch_actions import TouchActions
from .common.proxy import Proxy

__version__ = '2.47.0'
```

通过查看该文件就明白了它的原理，它其实对不同驱动的路径做了简化，并且将不同目录下的 WebDriver 类重命名为相应的浏览器（Firefox、Chrome、IE 等），所以，在调用

不同浏览器的驱动时就简化了层级。

再打开 selenium 包下的 webdriver/chrome 目录，查看 Chrome 中 webdriver.py 文件的实现。

webdriver.py

```
......
from selenium.webdriver.remote.webdriver import WebDriver as RemoteWebDriver

......

class WebDriver(RemoteWebDriver):
    """
    Controls the ChromeDriver and allows you to drive the browser.

    You will need to download the ChromeDriver executable from
    http://chromedriver.storage.googleapis.com/index.html
    """

    def __init__(self, executable_path="chromedriver", port=0,
                 chrome_options=None, service_args=None,
                 desired_capabilities=None, service_log_path=None):
        """
        Creates a new instance of the chrome driver.

        Starts the service and then creates new instance of chrome driver.

        :Args:
         - executable_path - path to the executable. If the default is used it
assumes the executable is in the $PATH
         - port - port you would like the service to run, if left as 0, a free
port will be found.
         - desired_capabilities: Dictionary object with non-browser specific
           capabilities only, such as "proxy" or "loggingPref".
         - chrome_options: this takes an instance of ChromeOptions
        """

......
```

同样查看 WebDriver 类的__init__()初始化方法，因为 Selenium 模块不自带 chromedriver.exe 驱动，所以，executable_path 参数会指定 chromedriver 驱动。

通过查看两个文件注意到一个细节，是 Firefox 和 Chrome 的 WebDriver 类都继承 RemoteWebDriver 类，也就是 remote 的 WebDriver 类，那么我们很好奇这个 WebDriver 类

实现了什么功能?

打开 selenium 包下 webdriver/remote 目录下的 webdriver.py 文件。

webdriver.py

```
.....
class WebDriver(object):
    """
    Controls a browser by sending commands to a remote server.
    This server is expected to be running the WebDriver wire protocol as defined
    here: http://code.google.com/p/selenium/wiki/JsonWireProtocol

    :Attributes:
     - command_executor - The command.CommandExecutor object used to execute
commands.
     - error_handler - errorhandler.ErrorHandler object used to verify that the
server did not return an error.
     - session_id - The session ID to send with every command.
     - capabilities - A dictionary of capabilities of the underlying browser for
this instance's session.
     - proxy - A selenium.webdriver.common.proxy.Proxy object, to specify a proxy
for the browser to use.
    """

    def __init__(self, command_executor='http://127.0.0.1:4444/wd/hub',
        desired_capabilities=None, browser_profile=None, proxy=None,
keep_alive=False):
        """
        Create a new driver that will issue commands using the wire protocol.

        :Args:
         - command_executor - Either a command.CommandExecutor object or a string
that specifies the URL of a remote server to send commands to.
         - desired_capabilities - Dictionary holding predefined values for
starting a browser
         - browser_profile - A
selenium.webdriver.firefox.firefox_profile.FirefoxProfile object.  Only used
if Firefox is requested.
        """

.....
```

WebDriver 类的__init__()初始化方法提供了一个重要信息，即 command_executor 参数，它默认指向本机（127.0.0.1）的 4444 端口号，通过修改这个参数可以使其指向任意的某台主机。

除此之外，我们还需要对浏览器进行配置。浏览器的配置由 desired_capabilities 参数决定，这个参数的秘密在 selenium 包的 webdriver/common 目录下的 desired_capabilities.py 文件中。

desired_ca...

```
.....

class DesiredCapabilities(object):

.....
    FIREFOX = {
        "browserName": "firefox",
        "version": "",
        "platform": "ANY",
        "javascriptEnabled": True,
        "marionette": False,
    }

.....
    CHROME = {
        "browserName": "chrome",
        "version": "",
        "platform": "ANY",
        "javascriptEnabled": True,
    }

.....
```

'browserName': 'chrome'	浏览器（chrome、Firefox)。
'version': ''	浏览器版本。
'platform': 'ANY'	测试平台（ANY 表示默认平台）。
'javascriptEnabled': True	JavaScript 启动状态。

"marionette": False　　marionette 是 Python 客户端允许你远程控制基于 gecko 的浏览器或设备运行一个 marionette 服务器，包括桌面 Firefox 和 Firefox OS。该参数为 Fiefox 特有。

DesiredCapabilities 平台及浏览器的参数如下

```
FIREFOX = {"browserName": "firefox", "version": "", "platform": "ANY","javascriptEnabled": True, "marionette": False, }

INTERNETEXPLORER = {"browserName": "internet explorer", "version": "", "platform": "WINDOWS", "javascriptEnabled": True, }

EDGE = {"browserName": "MicrosoftEdge", "version": "", "platform": "WINDOWS" }

CHROME = {"browserName": "chrome", "version": "", "platform": "ANY", "javascriptEnabled": True, }

OPERA = {"browserName": "opera", "version": "", "platform": "ANY", "javascriptEnabled": True, }

SAFARI = {"browserName": "safari", "version": "", "platform": "ANY", "javascriptEnabled": True, }

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

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

IPHONE = {"browserName": "iPhone", "version": "", "platform": "MAC", "javascriptEnabled": True, }

IPAD = {"browserName": "iPad", "version": "", "platform": "MAC", "javascriptEnabled": True, }

ANDROID = {"browserName": "android", "version": "", "platform": "ANDROID", "javascriptEnabled": True, }

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

值得一提的是，在 Windows 10 发布当天，Selenium 更新到了 2.47.0 版本，第一时间支持了基于 Windows 10 的 Edge 浏览器。

9.3.2　Remote 实例

下面通过 Remote 来运行测试用例。

首先，通过 Windows 命令提示符（或 Linux 终端）启动 Selenium Server。

```
> java -jar selenium-server-standalone-2.47.0.jar
```

编写自动化测试脚本。

remote_ts.py

```
from selenium.webdriver import Remote

# 调用 Remote 方法
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.find_element_by_id("kw").send_keys("remote")
driver.find_element_by_id("su").click()

driver.quit()
```

从上面的 Remote()方法配置来看，它相当于我们直接使用 webdriver.Chrome()，但是 Remote()却大大增加了配置的灵活性。

9.3.3　参数化平台及浏览器

通过 Selenium Server 可以轻松地创建本地节点和远程节点。而 Remote 的作用就是配置测试用例在这些节点上执行，下面就通过例子来演示它们两者的组合。

在本机打开 cmd 命令提示符窗口，分别启动一个 hub 和两个 node（节点）。

```
> java -jar selenium-server-standalone-2.47.0.jar  -role hub

> java -jar selenium-server-standalone-2.47.0.jar -role node -port 5555

> java -jar selenium-server-standalone-2.47.0.jar -role node -port 5556
```

下面修改脚本使其在不同的节点和浏览器上运行。

remote_ts.py

```
from selenium.webdriver import Remote

# 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()
```

首先，创建 lists 字典，定义不同的主机 IP、端口号及浏览器。然后，通过 for 循环读取 lists 字典中的数据作为 Remote()的配置信息，从而使脚本在不同的节点及浏览器下执行。

1. 启动远程 node

我们目前启动的 hub 与 node 都是在同一台主机上，要想在其他主机上启动 node，则必

须满足以下要求。

- 本地 hub 主机与远程 node 主机之间可以用 ping 命令连通。
- 远程主机必须安装用例执行的浏览器及驱动，并且驱动要放在环境变量 path 的目录下。
- 远程主机必须安装 Java 环境，因为需要运行 Selenium Server。

2. 操作步骤

① 启动本地 hub 主机（本地主机 IP 为：172.16.10.66）。

```
> java -jar selenium-server-standalone-2.47.0.jar -role hub
```

② 启动远程 node 主机（操作系统：Ubuntu ，IP 地址：172.16.10.34）。

```
$ java -jar selenium-server-standalone-2.47.0.jar -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 主机 IP 为 172.16.10.66。

③ 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。

remote_ts.py

```
......
# 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':'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　在启动 Selenium Server 时，每次都要输入一长串命令，非常麻烦。我们可以将启动命令生成批处理文件，方法很简单。首先创建一个 startup.bat 文件，例如，Selenium Server 存放于 D 盘 selenium 目录下，那么可以在.bat 文件中输入：

```
java -jar D:\\selenium\selenium-server-standalone-2.47.0.jar -role hub
```

然后，在需要启动 Selenium Server 的时候双击 startup.bat 文件即可。另外，我们还可以通过 VisGrid 工具来启动和管理节点，如图 9.8 所示。

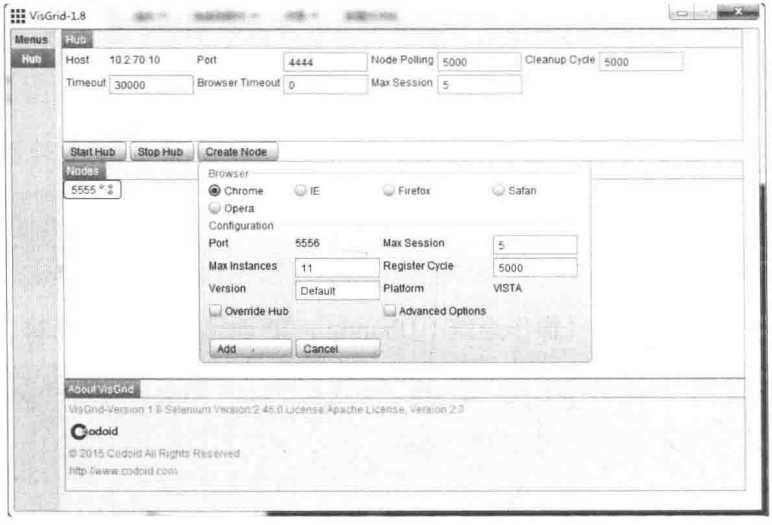

图 9.8 VisGrid 管理节点

9.4 WebDriver 驱动

在 9.3 节中，我们对 WebDriver 驱动的实现进行了简单分析，到目前为止，我们所熟悉的浏览器驱动有：Firefox Driver、Chrome Driver 和 IEDriverServer 等。除此之外，WebDriver 还支持哪些平台及驱动呢？本节将会对这些驱动作简单介绍。

WebDriver 所支持的平台/浏览器/模式如表 9.1 所示。

表 9.1 WebDriver 支持的平台/浏览器/模式

平台/浏览器/模式	驱　动	说　明
Android		支持脚本在 Android WebView 应用的测试，一般指移动端浏览器
BlackBerry		支持脚本在黑莓浏览器上运行
Firefox	包含在 Selenium 安装包中	包含在各语言的 Selenium(WebDriver)包里，这也是为什么安装完 Selenium 后就可以直接使用 Firefox 浏览器运行脚本的原因

（续）

平台/浏览器/模式	驱　动	说　明
Chrome	chromedriver.exe	因为 WebDriver 原本为谷歌的项目，之后与 Selenium 项目合并，所以对 Chrome 浏览器的支持也非常好
IE	IEDriverServer.exe	需要下载驱动支持，用于驱动脚本在 IE 系列的浏览器下运行
Edge	MicrosoftWebDriver.exe	支持脚本在 Windows10 操作系统 Edge 浏览器下执行
Opera	operadriver.exe	关于 Opera 浏览器的前世今生比较复杂，总之，现在的 OperaChromiumDriver（原 OperaDriver）基于 ChromeDriver
Safari	包含在 Selenium Server 中	Safari 浏览器由苹果公司开发，最早运行于苹果自家 MAC 平台。目前也提供 Windows 版本
HtmlUnit	包含在 Selenium Server 中	HtmlUnit 将请求返回文档模拟成 HTML，从而模拟浏览器的运行，但又非真正地启动一款浏览器执行脚本
PhantomJS	phantomjs.exe	PhantomJS 是一个拥有 JavaScript API 的无界面 WebKit，和 HtmlUnit 类似，可以看作是一款无界面的浏览器

1. 支持平台

WebDriver 支持 Android 和 BlackBerry 两个移动平台的浏览器测试，至于支持性如何，笔者并没有做过测试。Android 目前为市场占有率第一的移动平台，对于在其上面进行自动化测试，笔者推荐 Appium，Appium 扩展了 WebDriver 的协议，支持 iOS 平台和 Android 平台上的原生应用、Web 应用和混合应用等。

BlackBerry 平台目前的市场占有率很小，国内市场就更少了。

2. 支持浏览器

WebDriver 目前所支持的浏览器包括：Firefox、Chrome、IE、Edge、Opera、Safari。

为什么会选择上面几款浏览器进行支持呢？主要与浏览器的内核有关。

3. 支持模式

HtmlUnit 和 PhantomJS 是两个比较特殊的模式，我们可以把它们看作是伪浏览器，在这种模式下支持 html、Java Saript 等的解析，但不会真正地渲染出页面。由于不进行 CSS 及 GUI 渲染，所以运行效率上要比真实的浏览器快很多，主要用在功能性测试上面。

关于浏览器内核

浏览器最重要或者说核心的部分是“Rendering Engine”，可大概译为“渲染引擎”，不过我们一般习惯称其为“浏览器内核”，负责对网页语法的解释（如标准通用标记语言下的一个应用 HTML、JavaScript）并渲染（显示）网页。所以，所谓的浏览器内核，通常也就是浏览器所采用的渲染引擎，渲染引擎决定了浏览器如何显示网页的内容以及页面的格式信息。

Trident

Trident（IE 内核）：该内核程序在 1997 年的 IE 4 中首次被采用，是微软在 Mosaic 代码基础之上修改而来的，并沿用到 IE 11，也被普遍称作“IE 内核”。Trident 实际上是一款开放的内核，其接口内核设计得相当成熟，因此才有许多采用 IE 内核而非 IE 的浏览器（壳浏览器）涌现。

国内早期的浏览器基本上都是基于该内核的，如遨游浏览器、世界之窗浏览器、腾讯 TT、360 安全浏览器等。

Gecko

Gecko（Firefox 内核）：Netscape 6 开始采用的内核，后来的 Mozilla Firefox（火狐浏览器）也采用了该内核。Gecko 的特点是代码完全公开，因此，其可开发程度很高，全世界的程序员都可以为其编写代码，增加功能。

Presto

Presto（Opera 前内核）（已废弃）：该内核在 2003 年的 Opera 7 中首次被使用，该款引擎的特点就是渲染速度的优化达到了极致，然而代价是牺牲了网页的兼容性。但从 Opera 12.17 之后被 Opera 废弃，Opera 现已改用 Google Chrome 的 Blink 内核。

Webkit

Webkit（Safari 内核，Chrome 内核原型，开源）：它是苹果公司自己的内核，也是苹果的 Safari 浏览器使用的内核。苹果公司在 2005 年将 Webkit 公开为开源软件。谷歌当时采用苹果的 Webkit 核心打造了 Chrome 浏览器。随着 Chrome 浏览器以安全和快速的特点不断占领市场，国内的一些浏览器也开始基

于 Webkit 内核，如傲游浏览器 3、360 极速浏览器、搜狗高速浏览器等。

Blink

Blink 是一个由 Google 和 Opera Software 开发的浏览器排版引擎，该渲染引擎是开源引擎 WebKit 中 WebCore 组件的一个分支，并且在 Chrome（28 及往后版本）、Opera（15 及往后版本）和 Yandex 浏览器中使用。

9.4.1 Edge 浏览器

目前 Selenium 2.47.0 版本已经支持 Edge 浏览器，不过，Edge 浏览器只能运行于 Windows 10。与 IE 浏览器一样，要想让测试用例在该浏览器上运行，需要该浏览器的驱动，但是我们现在并不知道驱动名称和下载地址。当然，我们可以到 Selenium 官网上寻找答案。不过，这里介绍一种简单粗暴的方式，直接使用报错大法。

Python Shell

```
>>> from selenium import webdriver
>>> driver = webdriver.Edge()
Traceback (most recent call last):
  File "C:\Python35\lib\site-packages\selenium\webdriver\edge\service.py",
line 54, in start
    stdout=PIPE, stderr=PIPE)
  File "C:\Python35\lib\subprocess.py", line 950, in __init__
    restore_signals, start_new_session)
  File "C:\Python35\lib\subprocess.py", line 1220, in _execute_child
    startupinfo)
FileNotFoundError: [WinError 2] 系统找不到指定的文件。

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    driver = webdriver.Edge()
  File "C:\Python35\lib\site-packages\selenium\webdriver\edge\webdriver.py",
line 34, in __init__
    self.edge_service.start()
  File "C:\Python35\lib\site-packages\selenium\webdriver\edge\service.py",
line 59, in start
    "The EdgeDriver executable needs to be available in the path. "
```

```
selenium.common.exceptions.WebDriverException: Message: The EdgeDriver
executable needs to be available in the path. Please download from
http://go.microsoft.com/fwlink/?LinkId=619687
```

在错误信息的最后给出了 Edge 驱动的下载地址，复制链接下载 MicrosoftWebDriver.msi 文件。双击安装，将安装目录添加到系统环境变量 path 下，或将安装文件夹下的 MicrosoftWebDriver.exe 文件复制到 C:\Python35 目录下（该目录已经添加到系统环境变量 path 下）。

下面就可以使用 Edge 浏览器运行测试脚本了。

baidu.py

```
from selenium import webdriver

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

driver.find_element_by_id("kw").send_keys("Edge")
driver.find_element_by_id("su").click()
driver.close()
```

9.4.2 Opera 浏览器

Opera 浏览器的发展在浏览器内核介绍部分有所介绍。最近 Opera 浏览器采用 Blink 内核，需要 OperaChromiumDriver 驱动支持。

GitHub 地址：https://github.com/operasoftware/operachromiumdriver

下载 operadriver_win64.zip，解压后将得到 operadriver.exe 文件，同样将其放到 C:\Python35 目录下，原因同上。

baidu.py

```
from selenium import webdriver

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

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

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

9.4.3　Safari 浏览器

Safari 为苹果公司的浏览器，最早支持 MAC 平台，不过，现在已经支持 Windows 平台。与其他浏览器有所不同，Safari 浏览器没有相应的驱动文件，它的驱动被集成到了 Selenium Server 中，所以，需要通过 Remote 配置运行。

首先，在 Windows 命令提示符（或 Linux 终端）下启动 Selenium Server。

```
> java -jar selenium-server-standalone-2.47.0.jar
```

然后，运行脚本。

baidu.py

```
from selenium.webdriver import Remote

dc = {'browserName': 'safari'}
driver = Remote(command_executor=' http://127.0.0.1:4444/wd/hub',
                desired_capabilities=dc)

driver.get("http://www.baidu.com")
driver.find_element_by_id("kw").send_keys("safari")
driver.find_element_by_id("su").click()
driver.quit()
```

9.4.4　HtmlUnit 模式

HtmlUnit 官方网站：http://htmlunit.sourceforge.net/

HtmlUnit 是一款开源的 Java 页面分析工具，读取页面后，可以有效地使用 HtmlUnit 分析页面上的内容。项目可以模拟浏览器运行，被誉为 Java 浏览器的开源实现。这个没有界面的浏览器，其运行速度非常迅速。Selenium Server 中同样包含了 HtmlUnit 驱动。

首先，在 Windows 命令提示符（或 Linux 终端）下启动 Selenium Server。

```
> java -jar selenium-server-standalone-2.47.0.jar
```

然后，运行脚本。

baidu.py

```
from selenium.webdriver import Remote
from selenium.common.exceptions import WebDriverException

dc = {'browserName': 'htmlunit'}
driver = Remote(command_executor=' http://127.0.0.1:4444/wd/hub',
                desired_capabilities=dc)

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

driver.find_element_by_id("kw").send_keys("htmlunit")
driver.find_element_by_id("su").click()
driver.get_screenshot_as_file("D:\\run_ok.jpg")
driver.quit()
```

这种模式下运行脚本并不会真正地打开浏览器，整个过程都是在后台执行的。所以，为了证明运行是成功的，或者证明脚本确实执行了，可以在适当的步骤添加截图。

9.4.5 PhantomJS 模式

PhantomJS 官方网站：http://phantomjs.org/

PhantomJS 是一个拥有 JavaScript API 的无界面 WebKit 内核，与 HtmlUnit 类似。正如我们所知道的，Webkit 是 Safari 和其他一些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。这或许有些不可思议，所以我们可以把它作为一个可编程的浏览器终端。

在使用 PhantomJS 之前，需要先下载它。PhantomJS 支持 Windows、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。

下载完成后解压得到 phantomjs.exe 程序，将其复制到 C:\Python35 目录下，即可通过 PhantomJS 模式运行测试脚本。

baidu.py

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

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

try:
    driver.find_element_by_id("kw").send_keys("phantomjs")
    driver.find_element_by_id("su").click()
    sleep(1)
    driver.get_screenshot_as_file("D:\\baidu_ok.jpg")
except WebDriverException as msg:
    print(msg)
    driver.get_screenshot_as_file("D:\\baidu_error.jpg")
finally:
    driver.quit()
```

通过 HtmlUnit 或 PhantomJS 进行的自动化测试运行不会真正打开一个浏览器，在我们看来，可见的东西才会觉得是真实的，这时可以在脚本必要的位置添加截图，另一方面，截图也可以帮助定位。打开 D 盘下的 baidu_ok.jpg，截图如图 9.9 所示。

图 9.9　phantomJS 截图

本章小结

本章重点介绍了 Selenium Grid 的原理与使用，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。接着对 Remote 代码实现做了简单分析，以及如何进行平台及浏览器的参数化。最后，介绍了测试用例在不同浏览器及模式下的运行。

第 10 章
Python多线程

在学完第 9 章 Selenium Grid 之后，我们了解到 Selenium Grid 虽然可以分布式执行测试用例，但它并不支持并行。“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用例远程调用到不同的环境下执行；而并行强调“同时”执行多个任务。如何实现并行呢？可以利用编程语言提供的多线程（或多进程）技术来实现并行。

本章将学习 Python 的多线程与多进程技术，并将其应用到自动化测试用例的执行中。

在使用多线程之前，我们首先要理解进程和线程的概念。

什么是进程？

计算机程序只不过是磁盘中可执行的二进制（或其他类型）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周期。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，以及其他记录其运行轨迹的辅助数据。操作系统管理在其上面运行的所有进程，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）与进程有些相似，不同的是，所有的线程都运行在同一个进程中，共享相同的运行环境。我们可以想象成是在主进程或“主线程”中并行运行的“迷你进程”。

10.1 单线程的时代

在单线程时代，当处理器需要处理多个任务时，必须对这些任务安排执行顺序，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（movie），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

```
from time import sleep, ctime


# 听音乐任务
def music():
    print('I was listening to music! %s' % ctime())
    sleep(2)

# 看电影任务
def movie():
    print('I was at the movies! %s' % ctime())
    sleep(5)

if __name__ == '__main__':
    music()
    movie()
    print('all end:', ctime())
```

分别创建了两个任务 music 和 movie，执行时间分别为 2 秒和 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: D:/thread_test/onethread.py ====================
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
```

从运行结果可看到，程序从 11 分 04 秒开始播放 music，11 分 06 秒结束并开始 movie

的播放，最后，到 11 分 11 秒 movie 播放结束，总耗时 7 秒。

现在。我们对上面的例子做些调整，使它看起来更加有意思。

首先 music 和 movie 作为播放器，在用户使用时，可以根据用户的需求来播放任意的歌曲和影片，并且我们希望播放器能够提供循环播放的功能，尤其对于音乐播放器来说这个很重要，改造后的程序如下。

onethread2.py

```
from time import sleep, ctime


# 音乐播放器
def music(func, loop):
    for i in range(loop):
        print('I was listening to %s! %s' % (func, ctime()))
        sleep(2)


# 视频播放器
def movie(func, loop):
    for i in range(loop):
        print('I was at the %s! %s' % (func, ctime()))
        sleep(5)

if __name__ == '__main__':
    music('爱情买卖',2)
    movie('阿凡达',2)
    print('all end:', ctime())
```

给 music() 和 movie()两个函数设置参数：播放文件和播放次数。而函数中通过 for 循环控制播放的次数。再次运行，结果如下。

Python Shell

```
=================== RESTART: D:/thread_test/onethread2.py ===================
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
```

从运行结果可以看到，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 movie；42 分 50 秒两个任务结束，最终总耗时 14 秒。

10.2 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。

10.2.1 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，所有的子线程不管它们是否还在工作，都会被强行退出。有时我们并不希望发生这种行为，这时就引入了守护线程的概念。threading 模块支持守护线程，所以，我们直接使用 threading 来改进上面的例子。

threads.py

```
from time import sleep, ctime
import threading


# 音乐播放器
def music(func, loop):
    for i in range(loop):
        print("I was listening to %s! %s" % (func, ctime()))
        sleep(2)


# 视频播放器
def movie(func, loop):
    for i in range(loop):
        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=('爱情买卖',2))
threads.append(t1)

# 创建线程 t2,并添加到线程数组
t2 = threading.Thread(target=movie, args=('阿凡达', 2))
threads.append(t2)

if __name__ == '__main__':
    # 启动线程
    for t in threads:
        t.start()
    # 守护线程
    for t in threads:
        t.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。

通过 for 循环遍历 threads 数组中所装载的线程；start()开始线程活动，join()等待线程终止。如果不使用 join()方法对每个线程做等待终止，那么在线程运行的过程中可能会去执行最后的打印“all end：..”。

class threading.Thread()方法说明：

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.

运行结果。

Python Shell

```
===================== RESTART: D:/thread_test/threads.py ====================
I was listening to 爱情买卖! Mon Nov  2 21:11:24 2015I was at the 阿凡达! Mon Nov
2 21:11:24 2015

I was listening to 爱情买卖! Mon Nov  2 21:11:26 2015
I was at the 阿凡达! Mon Nov  2 21:11:29 2015
all end: Mon Nov  2 21:11:34 2015
```

从上面的运行结果可以看出，两个子线程（music、movie）同时启动于 11 分 24 秒，直到所有线程结束于 11 分 34 秒，总耗时 10 秒。movie 的两次电影循环共需要 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达到了并行工作。

10.2.2 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、...），当创建的线程较多时这样极其不方便。下面对例子进行改进。

player.py

```
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
lists = {'爱情买卖.mp3':3,'阿凡达.mp4':5,'我和你.mp3':4}

threads = []
files = range(len(lists))

# 创建线程
for file_, time in lists.items():
    t = threading.Thread(target=super_player, args=(file_, time))
    threads.append(t)

if __name__ == '__main__':
    # 启动线程
    for t in files:
        threads[t].start()
    for t in files:
        threads[t].join()

    print('end: %s' % ctime())
```

有趣的是我们对播放器的功能也做了增强。首先，创建了一个 super_player()函数，这个函数可以接收播放文件和播放时长，可以播放任何文件。

然后，我们创建了一个 lists 字典用于存放播放文件名与时长，通过 for 循环读取字典，并调用 super_player()函数创建字典，接着将创建的字典都追加到 threads 数组中。

最后，通过循环启动线程数组 threads 中的线程，运行结果如下。

Python Shell

```
===================== RESTART: D:/thread_test/player.py ====================
Start playing: 阿凡达.mp4! Mon Nov  2 21:15:51 2015Start playing: 我和你.mp3! Mon
Nov  2 21:15:51 2015Start playing: 爱情买卖.mp3! Mon Nov  2 21:15:51 2015

Start playing: 爱情买卖.mp3! Mon Nov  2 21:15:54 2015
```

```
Start playing: 我和你.mp3! Mon Nov  2 21:15:55 2015
Start playing: 阿凡达.mp4! Mon Nov  2 21:15:56 2015
end: Mon Nov  2 21:16:01 2015
```

10.2.3 创建线程类

除直接使用 Python 所提供的线程类外，我们还可以根据需求自定义自己的线程类。

mythread.py

```
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.func = func
        self.args = args
        self.name = name

    def run(self):
        self.func(*self.args)


def super_play(file_, time):
    for i in range(2):
        print('Start playing: %s! %s' % (file_, ctime()))
        sleep(time)

lists = {'爱情买卖.mp3':3,'阿凡达.mp4':5,'我和你.mp3':4}

threads = []
files = range(len(lists))

for file_, time in lists.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()
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。

在 Python 2 中，apply(func [, args [, kwargs]])函数的作用是当函数参数已经存在于一个元组或字典中时，apply()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，则任何参数都不会被传递，kwargs 是一个包含关键字参数的字典。

Python 3 中已经不再支持 apply()函数，所以将

apply(self.func,self.args)

修改为

self.func(*self.args)

最后，线程的创建与启动与前面的例子相同，唯一的区别是创建线程使用的是 MyThread 类，线程的入参形式也有所改变。

10.3　多进程技术

10.3.1　multiprocessing 模块

多进程 multiprocessing 模块的使用与多线程 threading 模块的用法类似。multiprocessing 提供了本地和远程的并发性，有效地通过全局解释锁（Global Interceptor 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

```
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)

lists = {'爱情买卖.mp3':3,'阿凡达.mp4':5,'我和你.mp3':4}

threads = []
files = range(len(lists))

# 创建进程
for file_, time in lists.items():
    t = multiprocessing.Process(target=super_player, args=(file_, time))
    threads.append(t)

if __name__ == '__main__':
    # 启动进程
    for t in files:
        threads[t].start()
    for t in files:
        threads[t].join()
    print('end:%s' % ctime())
```

从上面的实例中可以看到，多进程的用法几乎与多线程一样。

我们利用 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()等方法。

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 实际上不使用。

运行结果如下。

Python Shell

```
===================== RESTART: D:/thread_test/process.py ====================
Start playing:  爱情买卖.mp3! Sun Oct 25 12:23:31 2015
Start playing:  阿凡达.mp4! Sun Oct 25 12:23:31 2015
Start playing:  我和你.mp3! Sun Oct 25 12:23:31 2015
Start playing:  爱情买卖.mp3! Sun Oct 25 12:23:34 2015
Start playing:  我和你.mp3! Sun Oct 25 12:23:35 2015
Start playing:  阿凡达.mp4! Sun Oct 25 12:23:36 2015
end:Sun Oct 25 12:23:41 2015
```

从执行结果并不能看出采用多进程 multiprocessing 与多线程 threading 之间的差异。

扩展阅读

在 Unix\Linux 上面创建新的进程使用的是 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情，但如果初始参数或者传入的变量不同，则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，这是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，是指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当中直接访问到。在 Windows 平台上，因为没有 fork 语义的系统调用，所以基于继承的共享对象比 Unix\Linux 有更多的限制，最主要的体现就是要求 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。

10.3.2 Pipe 和 Queue

multiprocessing 提供了 threading 包中没有的 IPC（进程间通信），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 Pip 可以是单向(half-duplex)，也可以是双向(duplex)。我们通过 mutiprocessing.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

```
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')

if __name__ == '__main__':
    multiprocessing.freeze_support()
    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()
```

这里的 pipe 是双向的。pipe 对象建立的时候，返回一个含有两个元素的表，每个元素代表 pipe 的一端（Connection 对象）。我们对 pipe 的某一端调用 send()方法来传送对象，在另一端使用 recv()来接收。

Python Shell

```
===================== RESTART: D:/thread_test/pipe.py =====================
proc2 rec: hello
proc1 rec: hello, too
```

② Queue 类与 Pipe 相类似，都是先进先出结构。但 Queue 类允许多个进程放入，多个进程从队列取出对象。Queue 类使用 Queue（maxsize）创建，maxsize 表示队列中可以存放对象的最大数量。

queue.py

```
import multiprocessing
import os, time


def inputQ(queue):
    info = str(os.getpid()) + '(put):' + str(time.time())
    queue.put(info)


def outputQ(queue, lock):
    info = queue.get()
    lock.acquire()
    print((str(os.getpid()) + '(get):' + info))
    lock.release()

if __name__ == '__main__':

    record1 = []
    record2 = []
    lock = multiprocessing.Lock()    # 加锁，为防止散乱的打印
    queue = multiprocessing.Queue(3)

    for i in range(10):
        process = multiprocessing.Process(target=inputQ,args=(queue,))
        process.start()
```

```
        record1.append(process)

    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()
```

运行结果：

Python Shell

```
====================== RESTART: D:/thread_test/queue.py ======================
784(get):4988(put):1445747594.7366996
5460(get):2344(put):1445747594.7679493
4752(get):2648(put):1445747594.83045
9172(get):10964(put):1445747594.83045
9660(get):10888(put):1445747594.8460755
6300(get):5292(put):1445747594.9710765
7232(get):6960(put):1445747595.002327
11052(get):7992(put):1445747595.002327
5384(get):260(put):1445747595.0179522
8448(get):6204(put):1445747595.0179522
```

多进程的 IPC 不是本书的重点，感兴趣的读者可以参考其他资料进一步学习。

10.4 应用于自动化测试

因为多进程与多线程技术在编程语言中属于比较高级的应用，对于初学者来说理解起来会有一定的难度，所以我们运用多个实例循序渐进地进行了讲解。下面就将其与自动化测试用例执行结合起来，从而节省用例的总体运行时间。

10.4.1　多线程执行测试用例

这里同样以百度搜索为例，通过不同的浏览器来启动不同的线程。

baidu_thread.py

```
from threading  import Thread
from selenium import webdriver
from time import ctime, sleep


# 测试用例
def test_baidu(browser, search):
    print('start:%s' % ctime())
    print('browser:%s ,' % browser)
    if browser == "ie":
        driver = webdriver.Ie()
    elif browser == "chrome":
        driver = webdriver.Chrome()
    elif browser == "ff":
        driver = webdriver.Firefox()
    else:
        print("browser 参数有误，只能为 ie、ff、chrome")

    driver.get('http://www.baidu.com')
    driver.find_element_by_id("kw").send_keys(search)
    driver.find_element_by_id("su").click()
    sleep(2)
    driver.quit()

if __name__ == '__main__':
    # 启动参数（指定浏览器与百度搜索内容）
    lists = {'chrome':'threading', 'ie': 'webdriver', 'ff':'python'}

    threads = []
    files = range(len(lists))

    # 创建线程
    for browser, search in lists.items():
        t = Thread(target=test_baidu, args=(browser, search))
        threads.append(t)

    # 启动线程
```

```
    for t in files:
        threads[t].start()
    for t in files:
        threads[t].join()

    print('end:%s' % ctime())
```

运行结果。

Python Shell

```
====================== RESTART: D:/test/baidu_thread.py ======================
start:Mon Nov  2 21:35:23 2015start:Mon Nov  2 21:35:23 2015start:Mon Nov  2
21:35:23 2015

browser:chrome ,browser:ie ,browser:ff ,

end:Mon Nov  2 21:35:40 2015
```

创建 lists 字典，对浏览器与搜索的内容进行参数化。通过多线程来运行 test_baidu()的测试用例，在执行用例前使用多重 if 来判断通过哪个浏览器运行测试用例，并通过百度搜索相应的关键字。

10.4.2 多线程分布式执行测试用例

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测试用例，这个我们在前面已经反复强调。下面就通过演示使用多线程技术结合 Selenium Grid 实现分布式并行地执行测试用例。

启动 Selenium Server

在本机打开两个命令提示符窗口。

本机启动一个主 hub 和一个 node 节点（端口号别分为 4444 和 5555），本机 IP 地址为：172.16.10.66。

```
C:\selenium>java -jar selenium-server-standalone-2.47.0.jar -role hub

C:\selenium>java -jar selenium-server-standalone-2.47.0.jar -role node -port 5555
```

启动一个远程 node（设置端口号为 6666），IP 地址为：172.16.10.34。

```
fnngj@fnngj-VirtualBox:~/selenium$ java -jar selenium-server-standalone-2.47.0ar
-role node -port 6666 -hub http://172.16.10.66:4444/grid/register
```

运行测试脚本。

grid_thread.py

```
from threading import Thread
from selenium import webdriver
from time import sleep, ctime


# 测试用例
def test_baidu(host,browser):
    print('start:%s' % ctime())
    print(host, browser)
    dc = {'browserName': browser}
    driver = webdriver.Remote(command_executor=host,
                              desired_capabilities=dc)
    driver.get('http://www.baidu.com')
    driver.find_element_by_id("kw").send_keys(browser)
    driver.find_element_by_id("su").click()
    driver.close()

if __name__ == '__main__':
    # 启动参数（指定运行主机与浏览器）
    lists = {'http://127.0.0.1:4444/wd/hub': 'chrome',
            'http://127.0.0.1:5555/wd/hub': 'internet explorer',
            'http://172.16.10.34:6666/wd/hub': 'firefox',  # 远程节点
             }
    threads = []
    files = range(len(lists))

    # 创建线程
    for host, browser in lists.items():
        t = Thread(target=test_baidu, args=(host, browser))
        threads.append(t)

    # 启动线程
    for i in files:
```

```
        threads[i].start()
    for i in files:
        threads[i].join()

    print('end:%s' % ctime())
```

运行结果如下。

Python Shell

```
====================== RESTART: D:/test/grid_thread.py ======================
start:Sun Jul 19 13:18:25 2015
http://127.0.0.1:5555/wd/hub internet explorer
start:Sun Jul 19 13:18:25 2015
http://127.0.0.1:6666/wd/hub firefox
start:Sun Jul 19 13:18:25 2015
http://127.0.0.1:4444/wd/hub chrome
end:Sun Jul 19 13:18:40 2015
```

与前一个例子类似，只是这次多线程根据 lists 字典中节点与浏览器来启动线程数，test_baidu()测试用例根据节点与浏览器来参数化 Remote()，从而在不同的节点上运行测试用例。

本章小结

本章我们使用了大量的实例来介绍 Python 的多线程与多进程技术。在实际应用中，多线程与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，因此对于读者来说，理解并掌握多线程技术可有效提高编程能力。

最后比较遗憾的是，unittest 单元测试框架本身并不支持多线程技术，它不能像 Java 的 TestNG 框架一样通过简单的配置就可以使用多线程技术执行测试用例。

第 11 章
自动化测试项目实战

学完前面十个章节的内容，相信读者此时已经具备了开发自动化测试项目的能力。如果你依然对如何开展自动化测试无从下手，那么可能有两个原因：一方面可能是前面的内容学得不够扎实，没有达到理解运行程度。另一方面可能是对被测项目的理解不够，不能挖掘出相关的自动化需求。不管出于哪一方面原因，本书都有责任帮助你强化前面所学的内容。这也是我在计划编写本书时，大多数读者向我提议希望能看到的有实战的内容。

本章将尽量通过一个具体的项目来告诉你如何进行自动化测试的开发，当然，作者经验有限，这个项目的设计并非完美，更多的是希望向读者传达一些项目中的开发技巧和设计思想。

11.1 自动化测试用例设计

对于测试人员来说，不管是进行功能测试、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地体现测试人员的经验、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，我们有必要聊聊自动化测试用例的一些特点，以及如何编写自动化测试用例。

11.1.1 手工测试用例与自动化测试用例

手工测试用例是针对功能测试人员的，而自动化测试用例则是针对自动化测试框架或工具的；前者是功能测试用例人员通过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。两者各自最大的特点在于，前者具有较好的异常处理能力，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，能够细致地定位问题；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如下。

1. 手工测试用例特点：

- 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是否正确实现
- 人工执行用例具有一定的步骤跳跃性
- 人工测试步步跟踪，能够细致地定位问题
- 主要用来发现功能缺陷

2. 自动化测试用例特点：

- 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和完整，让测试人员从烦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存在较大的差异，所以，不能直接把手工测试用例 “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地认识到，自动化测试不能完全地替代手工测试，自动化测试的目的仅仅在于让测试人员从烦琐重复的测试过程中解脱出来，把更多的时间和精力放到更有价值的测试中，例如探索性测试。而自动化测试更多的是用来进行冒烟测试和回归测试。

3. 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成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，分多个用例，多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，这部分用例作冒烟测试，那部分用例作回归测

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，很烦琐的部分。例如，字段验证、提示信息验证这类，这部分适用于回归测试。

6）选取的用例可以是主体流程，这部分适用于冒烟测试。

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，但也算用例拓展的一部分，项目负责人可以有选择地增加。

8）平时在手工测试时，如果需要构造一些复杂的数据或重复一些简单的机械式动作，则告诉自动化脚本，让它来帮你，或许你的效率会因此而得到提高。

11.1.2　测试类型

1. 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吗？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？每个页面都有正确的头部文本吗？

你可能需要（也可能不需要）对页面内容进行自动化测试。如果你的网页内容是不易受到影响，则手工对内容进行测试就足够了。假设你的应用文件的位置被移动了，则内容测试就非常有价值。

2. 测试链接

Web 站点的一个常见错误为失效的链接或链接指向无效页。链接测试涉及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，则手动测试就足够了。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，则链接测试应该实现自动化。

3. 功能测试

在你的应用程序中，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，但通常也是最重要的。典型的测试是登录、注册网站账户、用户账户操作、账户设置变化、复杂的数据检索操作，等等。功能测试通常对应着你的应用程序的描述应用特性或设计的使用场景。

4. 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，也可以是动态生成值，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 ，而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。

5. Ajax 的测试

Ajax 是一种支持以及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。Ajax 有无数更新网页上元素的方法。最简单的方式是在 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，而不需要重新加载整个页面，只有一小部分的页面，或者只有元素本身被重新加载。

11.1.3 自动化测试用例编写原则

在编写自动化测试用例过程中应该遵循以下原则：

1）一个用例为一个完整的场景，从用户登录系统到最终退出并关闭浏览器。

2）一个用例只验证一个功能点，不要试图在用户登录系统后把所有的功能都验证一遍。

3）尽可能少的编写逆向逻辑用例。一方面因为逆向逻辑的用例很多（例如，手机号输错有几十种情况）；另一方面自动化脚本本身比较脆弱，复杂的逆向逻辑用例实现起来较为麻烦且容易出错。

4）用例与用例之间尽量避免产生依赖。

5）一条用例完成测试之后需要对测试场景进行还原，以免影响其他用例的执行。

11.2　BBS 社区项目实战

本节以一个 BBS 社区项目为例，BBS 社区属于互联网比较典型的应用，主要有登录、个人中心、发帖、查看帖子、搜索、签到等功能。

11.2.1　准备工作

1. 项目开发是个循序渐进的过程

需要向读者说明的是，我接下来要介绍的这个自动化测试项目，并非项目最初的形态，其间经历了多次代码迭代与结构的重构，并且仅仅只是符合当前的项目需求。为什么要强调这些呢？

相信我们都知道一个只有几条测试用例的项目和有一个几百条测试用例的项目结构肯定是不一样的。对于只有几条测试用例的项目，我不需要考虑太多结构方面的问题，甚至只用线性模型来编写用例，其维护成本也不会太高；但是，当用例达到几百条时就不得不考虑各种问题，例如，如何降低测试代码的冗余、对代码进行抽象与分层、采用哪种设计模式，等等。

自动动化测试的开发，是个不断调整代码与结构的过程，也许第一天你编写了二十条用例，到第二天的时候，你需要花三分之一的时间对昨天的部分代码进行调整或重构。只有三分之二的时间用于编写新的用例。类、方法和函数的命名也是需要考究的方面，既要尽量保持简洁，又要见名知义。代码的编写更是如此，如何写出简洁优雅的代码是对我们编程功底的考验。遗憾的是无法带着读者去复盘这样一个过程。其实，这个过程也必须由读者自己在不断实践中积累和总结。

2. 选择合适的 IDE

工欲善其事，必先利其器，在开始开发自动化项目之前，我们有必要先来聊一聊 Python 有哪些 IDE。前面章节中的例子默认以 Python 自带的 IDLE 运行，但这只推荐于 Python 语言的初学者，在真正利用 Python 开发项目时它就显得比较鸡肋了。当然，关于 IDE 的讨论一直属于热门话题，这里并不是要分辩个孰优孰劣，这里只是想告诉读者不同的编程阶段应选择适合自己的 IDE。

Python IDLE ：如果读者初学 Python，并且不精通其他编程语言及 IDE，则建议从这个 IDE 入手，它自带的 Shell 模式可以帮助我们快速练习 Python 语法，笔者初学 Python 时用了半年。

UliPad ：轻量级的 Python IDE，由国内用户基于 wxPython 开发，代码着色及自动补全功能很不错，配置也相对比较简单。

Sublime ：通用型轻量级 IDE，支持多种编程语言。有许多功能强大的快捷键（如 Ctrl+d），如果平时需要在多种编程语言间切换，那么这将是不错的选择。这也是笔者最常用的 IDE 之一。

PyCharm ：Python 重量级 IDE，功能强大，自动检测语法，可以帮助我们写出更规范的 Python 代码。对于处女座的开发者来说是个不错的选择。笔者试用半天后果断拥抱之。

Eclipse + pydev ：Eclipse 也属于重量级 IDE。相信学习 Java 语言的同学一般都会选择此 IDE，配置 pydev 插件后同样可以用来编写 Python 程序，对于熟悉 Eclipse 的同学是个不错的选择。

Vim 与 Emacs ：一直是程序员大神口中的神器，学习成本很高。

通过简单的介绍，想信读者已经找到了适合自己的 IDE，下面就跟着笔者一起动手开发自动化项目吧。

11.2.2 项目结构介绍

自动化测试项目结构如图 11.1 所示。

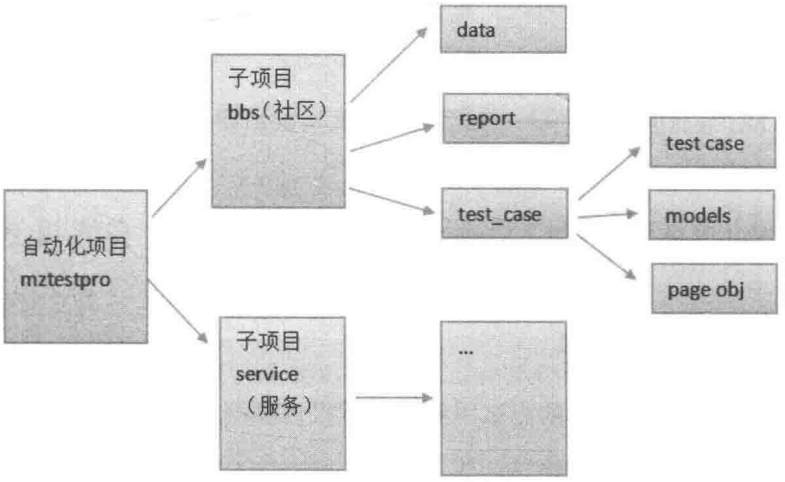

图 11.1　自动化测试项目结构

下面逐级介绍此目录与文件的作用：

```
mztestpro/
├── bbs/
│   ├── data/
│   ├── report/
│   │   └── image/
│   └── test_case/
│       ├── models/
│       │   ├── driver.py
│       │   ├── function.py
│       │   └── myunit.py
│       ├── page_obj/
│       │   └── *Page.py
│       └── *_sta.py
```

```
├── driver/
├── package/
├── run_bbs_test.py
├── startup.bat
└──自动化测试项目说明文档.docx
```

1. mztestpro 测试项目

bbs ：用于存放 BBS 项目的测试用例、测试报告和测试数据等。

driver：用于存放浏览器驱动。如 selenium-server-standalone-2.47.0.jar、chromedriver.exe、IEDriverServer.exe 等。在执行测试前根据执行场景将浏览器驱动复制到系统环境变量 path 目录下。

package：用于存放自动化所用到的扩展包。例如，HTMLTestRunner.py 属于一个单独模块，并且对其做了修改，所以，在执行测试前需要将它复制到 Python 的 Lib 目录下。

run_bbs_test.py ：项目主程序。用来运行社区（BBS）自动化用例。

startup.bat ：用于启动 Selenium Server，默认启动 driver 目录下的 selenium-server-standalone-2.47.0.jar。

自动化测试项目说明文档.docx ：介绍当前项目的架构、配置和使用说明。

2. bbs 目录

data ：该目录用来存放测试相关的数据。

report：用于存放 HTML 测试报告。其下面创建了 image 目录用于存放测试过程中的截图。

test_case：测试用例目录，用于存放测试用例及相关模块。

3. test_case 目录

models ：该目录下存放了一些公共的配置函数及公共类。

page_obj ：该目录用于存放测试用例的页面对象（Page Object）。根据自定义规则，以“*Page.py”命名的文件为封装的页面对象文件。

*_sta.py ：测试用例文件。根据测试文件匹配规则，以“*_sta.py”命名的文件将被当作自动化测试用例执行。

11.2.3　编写公共模块

首先定义驱动文件：

...\mztestpro\bbs\test_case\models\driver.py

driver.py

```
from selenium.webdriver import Remote
from selenium import webdriver


# 启动浏览器驱动
def browser():
    # driver = webdriver.Chrome()
    host = '127.0.0.1:4444'    # 运行主机：端口号 （本机默认：127.0.0.1:4444）
    dc = {'browserName': 'chrome'}   # 指定浏览器（'chrome','firefox',）
    driver = Remote(command_executor='http://' + host + '/wd/hub',
                    desired_capabilities=dc)
    return driver


if __name__ == '__main__':
    dr = browser()
    dr.get("http://www.baidu.com")
    dr.quit()
```

定义浏览器驱动函数 browser()，该函数可以进行配置，根据我们的需求，配置测试用例在不同的主机及浏览器下运行。如果不知道如何配置请参考本书第 10 章。

自定义测试框架类：

...\mztestpro\bbs\test_case\models\myunit.py

myunit.py

```
from selenium import webdriver
from .driver import browser
import unittest
import os


class MyTest(unittest.TestCase):

    def setUp(self):
        self.driver = browser()
        self.driver.implicitly_wait(10)
        self.driver.maximize_window()

    def tearDown(self):
        self.driver.quit()
```

定义 MyTest()类用于继承 unittest.TestCase 类，因为笔者创建的所有测试类中 setUp()与 tearDown()方法所做的事情相同，所以，将它们抽象为 MyTest()类，好处就是在编写测试用例时不再考虑这两个方法的实现。

定义截图函数：

...\mztestpro\bbs\test_case\models\function.py

function.py

```
from selenium import webdriver
import os


# 截图函数
def insert_img(driver, file_name):
    base_dir = os.path.dirname(os.path.dirname(__file__))
    base_dir = str(base_dir)
    base_dir = base_dir.replace('\\', '/')
    base = base_dir.split('/test_case')[0]
    file_path = base + "/report/image/" + file_name
    driver.get_screenshot_as_file(file_path)
```

```
if __name__ == '__main__':
    driver = webdriver.Chrome()
    driver.get("https://www.baidu.com")
    insert_img(driver, 'baidu.jpg')
    driver.quit()
```

创建截图函数 insert_img()，为了保持自动化项目的移植性，采用相对路径的方式将测试截图保存到.\report\image\目录中。

11.2.4　编写 Page Object

关于 Page Object 设计模式，在本书第 8.3 节已经有过介绍，这里我们将使用该设计模式来编写测试用例。

首先创建基础 Page 基础类：

...\mztestpro\bbs\test_case\page_obj\base.py

base.py

```
class Page(object):
    '''
    页面基础类，用于所有页面的继承
    '''

    bbs_url = 'http://bbs.meizu.cn'

    def __init__(self, selenium_driver, base_url=bbs_url, parent=None):
        self.base_url = base_url
        self.driver = selenium_driver
        self.timeout = 30
        self.parent = parent

    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 find_elements(self, *loc):
        return self.driver.find_elements(*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)
```

创建页面基础类，通过__init__()方法初始化参数：浏览器驱动、URL 地址、超时时长等。定义基本方法：open()用于打开 BBS 地址；find_element()和 find_elements()分别用来定位单个与多个元素；创建 script()方法可以更简便地调用 JavaScript 代码。当然我们还可以对更多的 WebDriver 方法进行重定义。

登录页面如图 11.2 所示。

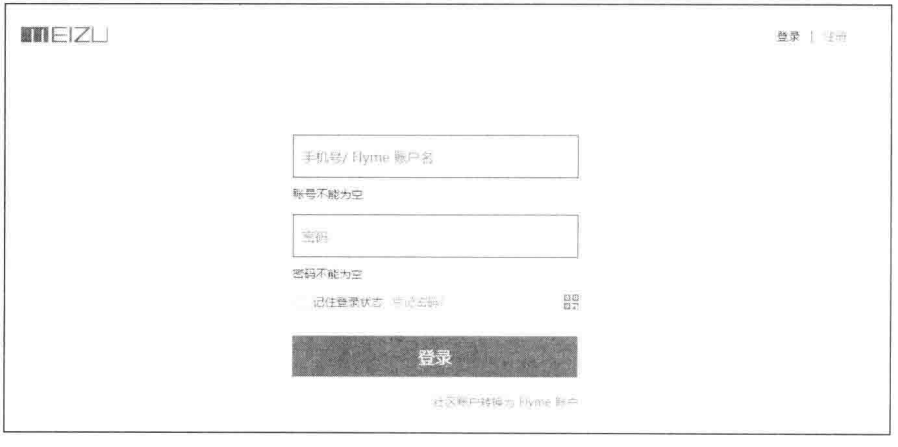

图 11.2　登录页面

创建 BBS 登录对象类：

...\mztestpro\bbs\test_case\page_obj\loginPage.py

loginPage.py

```
from selenium.webdriver.common.action_chains import ActionChains
```

```
from selenium.webdriver.common.by import By
from .base import Page
from time import sleep


class login(Page):
    '''
    用户登录页面
    '''

    url = '/'

    # Action
    bbs_login_user_loc =(By.XPATH, "//div[@id='mzCust']/div/img")
    bbs_login_button_loc = (By.ID, "mzLogin")

    def bbs_login(self):
        self.find_element(*self.bbs_login_user_loc).click()
        sleep(1)
        self.find_element(*self.bbs_login_button_loc).click()

    login_username_loc = (By.ID, "account")
    login_password_loc = (By.ID, "password")
    login_button_loc = (By.ID, "login")

    # 登录用户名
    def login_username(self, username):
        self.find_element(*self.login_username_loc).send_keys(username)

    # 登录密码
    def login_password(self, password):
        self.find_element(*self.login_password_loc).send_keys(password)

    # 登录按钮
    def login_button(self):
        self.find_element(*self.login_button_loc).click()

    # 定义统一登录入口
    def user_login(self, username="username", password="1111"):
        """ 获取的用户名密码登录"""
        self.open()
        self.bbs_login()
        self.login_username(username)
```

```
        self.login_password(password)
        self.login_button()
        sleep(1)

    user_error_hint_loc = (By.XPATH, "//span[@for='account']" )
    pawd_error_hint_loc = (By.XPATH, "//span[@for='password']")
    user_login_success_loc = (By.ID, "mzCustName")

    # 用户名错误提示
    def user_error_hint(self):
        return self.find_element(*self.user_error_hint_loc).text

    # 密码错误提示
    def pawd_error_hint(self):
        return self.find_element(*self.pawd_error_hint_loc).text

    # 登录成功用户名
    def user_login_success(self):
        return self.find_element(*self.user_login_success_loc).text
```

创建登录页面对象，对用户登录面上的用户名/密码输入框、登录按钮和提示信息等元素的定位进行封装。除此之外，还创建 user_login()方法作为系统统一登录的入口。关于对操作步骤的封装既可以放在 Page Object 当中，也可以放在测试用例当中，这个主要根据具体需求来衡量。这里之所以存放在 Page Object 当中，主要考虑到还有其他用例会调用到该登录方法。为 username 和 password 入参设置了默认值是为了方便其他用例在调用 user_login()时不用再传递登录用户信息，因为该系统大多用例的执行使用该账号即可，同时也方便了在账号失效时的修改。

11.2.5 编写测试用例

现在开始编写测试用例程序，因为前面已经做好了基础工作，此时测试用例的编写将会简便很多，更能集中精力考虑用例的设计与实现。

创建 BBS 登录类：

```
...\mztestpro\bbs\test_case\login_sta.py
```

此处需要注意文件名的创建。例如，假设登录页的对象命名为 loginPage.py ，那么关于测试登录的用例文件应该命名为 login_sta.py，这样方便后期用例报错时问题的追踪。尽

量把一个页面上的元素定位封装到一个“*Page.py”文件中，把针对这个页面的测试用例集中到一个“*_sta.py”文件中。

login_sta.py

```
from time import sleep
import unittest, random, sys
sys.path.append("./models")
sys.path.append("./page_obj")
from models import myunit, function
from page_obj.loginPage import login


class loginTest(myunit.MyTest):
    '''社区登录测试'''

    # 测试用户登录
    def user_login_verify(self, username="", password=""):
        login(self.driver).user_login(username, password)

    def test_login1(self):
        '''用户名、密码为空登录'''
        self.user_login_verify()
        po = login(self.driver)
        self.assertEqual(po.user_error_hint(), "账号不能为空")
        self.assertEqual(po.pawd_error_hint(), "密码不能为空")
        function.insert_img(self.driver, "user_pawd_empty.jpg")

    def test_login2(self):
        '''用户名正确,密码为空登录'''
        self.user_login_verify(username="pytest")
        po = login(self.driver)
        self.assertEqual(po.pawd_error_hint(), "密码不能为空")
        function.insert_img(self.driver, "pawd_empty.jpg")

    def test_login3(self):
        '''用户名为空,密码正确'''
        self.user_login_verify(password="abc123456")
        po = login(self.driver)
        self.assertEqual(po.user_error_hint(), "账号不能为空")
        function.insert_img(self.driver, "user_empty.jpg")
```

```
    def test_login4(self):
        '''用户名与密码不匹配'''
        character = random.choice('zyxwvutsrqponmlkjihgfedcba')
        username = "zhangsan" + character
        self.user_login_verify(username=username, password="123456")
        po = login(self.driver)
        self.assertEqual(po.pawd_error_hint(), "密码与账号不匹配")
        function.insert_img(self.driver, "user_pawd_error.jpg")

    def test_login5(self):
        '''用户名、密码正确'''
        self.user_login_verify(username="zhangsan",password="123456")
        sleep(2)
        po = login(self.driver)
        self.assertEqual(po.user_login_success(), '张三')
        function.insert_img(self.driver, "user_pawd_ture.jpg")


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

首先创建 loginTest()类，继承 myunit.MyTest()类，关于 MyTest()类的实现，请翻看前面的代码。这样就省去在了在每个测试类中实现一遍 setUp()和 tearDown()方法。

创建 user_login_verify()方法，并调用 loginPage.py 中定义的 user_login()方法。为什么不直接调用呢？因为 user_login()的入参已经设置了默认值，原因前面已经解释，这里需要重新将其入参的默认值设置为空即可。

前三条测试用例很好理解，分别验证：

- 用户名密码为空，点击登录；
- 用户名正确，密码为空，点击登录；
- 用户名为空，密码正确，点击登录。

第四条用例验证错误的用户名和密码登录。在当前系统中如果反复使用固定且错误的用户名和密码，系统会弹出验证码输入框。为了避免这种情况的发生，就需要用户名进行随机变化，此处的做法用固定的前缀“zhangsan”，末尾字符从 a~z 中随机一个字符与前缀进行拼接。

第五条用例验证正确的用户名和密码登录，通过获取用户名作为断言信息。

在上面的测试用例中，每条测试用例结束时都调用 function.py 文件中的 insert_img 函数进行截图。当用例运行完成后，打开.../report/image/目录将会看到用例执行的截图文件，如图 11.3 所示。

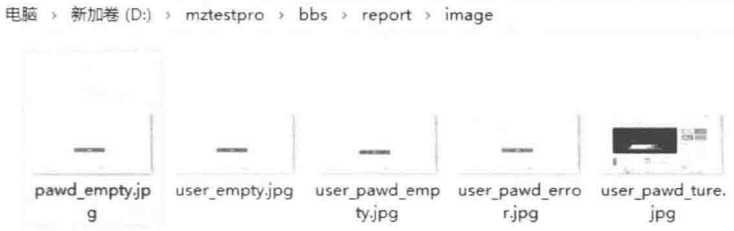

图 11.3　测试用例截图

11.2.6　执行测试用例

为了在测试用例运行过程中不影响做其他事，笔者选择调用远程主机或虚拟机来运行测试用例，那么这里就需要使用 Selenium Grid（其包含在 Selenium Server）来调用远程节点。

创建...\mztestpro\startup.bat 文件，用于启动...\mztestpro\driver\目录下的 Selenium Server。

startup.bat

```
java -jar ./driver/selenium-server-standalone-2.47.0.jar -role hub
```

双击 strtup.bat 文件，启动 Selenium Server 创建主 hub 节点。在远程主机或虚拟机中同样需要启动 Selenium Server 创建 node 节点，创建方式参考本书第 9.3 节。

创建用例执行程序：...\mztestpro\run_bbs_test.py

run_bbs_test.py

```
from HTMLTestRunner import HTMLTestRunner
from email.mime.text import MIMEText
from email.header import Header
import smtplib
import unittest
import time
import os
```

```
# =============定义发送邮件==========
def send_mail(file_new):
    f = open(file_new, 'rb')
    mail_body = f.read()
    f.close()

    msg = MIMEText(mail_body, 'html', 'utf-8')
    msg['Subject'] = Header("自动化测试报告", 'utf-8')

    smtp = smtplib.SMTP()
    smtp.connect("smtp.126.com")
    smtp.login("username@126.com", "123456")
    smtp.sendmail("username@126.com", "receive@126.com", 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 new_report(testreport):
    lists = os.listdir(testreport)
    lists.sort(key=lambda fn: os.path.getmtime(testreport + "\\" + fn))
    file_new = os.path.join(testreport, lists[-1])
    print(file_new)
    return file_new


if __name__ == '__main__':
    now = time.strftime("%Y-%m-%d %H_%M_%S")
    filename = './bbs/report/' + now + 'result.html'
    fp = open(filename, 'wb')
    runner = HTMLTestRunner(stream=fp,
                            title='魅族社区自动化测试报告',
                            description='环境: windows 7 浏览器: chrome')
    discover = unittest.defaultTestLoader.discover('./bbs/test_case',
                                                   pattern='*_sta.py')
    runner.run(discover)
    fp.close()  # 关闭生成的报告
    file_path = new_report('./bbs/report/')  # 查找新生成的报告
    send_mail(file_path)  # 调用发邮件模块
```

执行过程中并没有做任何改动，集成了 HTMLTestRunner 生成 HTML 测试报告，以及

集成自动发邮件功能等。唯一需要注意的是，脚本中的路径建议使用相对路径，以便于项目被移动到任意目录下执行。

打开...\models\driver.py 文件，修改脚本运行的节点及浏览器。现在可以通过运行 run_bbs_test.py 来执行测试项目了。

本章小结

如果你完成了前面的操作，那么这只是自动化项目的开始，不过，我们已经把基本架构设计完成，后面的大部分工作就是编写各个页面的*Page.py 以及测试用例*_sta.py。在这个过程中会遇到各种各样的问题，如元素的定位、架构的扩展等，需要读者自己去克服这些问题。

最后送上一份 BBS 自动化测试报告截图，以示鼓励，如图 11.4 所示。

魅族社区自动化测试报告

Start Time: 2015-07-28 10:33:31
Duration: 0:45:19.369000
Status: Pass 201

环境: windows 7 浏览器: chrome

Show Summary Failed All

Test Group/Test case	Count	Pass	Fail	Error	View
atest_login_sta.loginTest: 社区登录测试	5	5	0	0	Detail
bbs_home_page_sta.homePageFirmwareGet: 社区首页--固件下载	2	2	0	0	Detail
bbs_home_page_sta.homePageLink: 社区首页--板块链接测试	12	12	0	0	Detail
bbs_home_page_sta.homePageStatisticsTest: 社区首页--统计	3	3	0	0	Detail
bbs_home_page_sta.homePageTset: 社区首页--主要功能测试	5	5	0	0	Detail
bbs_home_page_sta.homePageWeiboTest: 社区首页--社区微博	1	1	0	0	Detail
bbs_sign_sta.bbsSignTset: 社区--签到功能测试	3	3	0	0	Detail
bbs_sreach_sta.bbsSreachTset: 社区--搜索功能测试	5	5	0	0	Detail
my_dynamic_sta.dating: 个人中心--我的动态--大厅标签页测试	4	4	0	0	Detail
my_dynamic_sta.guangbo: 个人中心--我的动态--广播标签页测试	4	4	0	0	Detail
my_dynamic_sta.guangbo_shu: 个人中心--我的动态--用户广播数测试	2	2	0	0	Detail
my_dynamic_sta.guanzhu: 个人中心--我的动态--关注标签页测试	4	4	0	0	Detail
my_friend_sta.Friend: 个人中心--我的好友---好友	3	3	0	0	Detail
my_friend_sta.Sreach: 个人中心--我的好友---搜索	3	3	0	0	Detail
my_friend_sta.listenIn: 个人中心--我的好友---收听	1	1	0	0	Detail
my_friend_sta.listenOut: 个人中心--我的好友---听众	2	2	0	0	Detail
my_invitation_sta.Status: 个人中心--我的帖子--主题--状态测试	4	4	0	0	Detail
my_invitation_sta.otherTags: 个人中心--我的帖子--其它标签页	3	3	0	0	Detail
my_invitation_sta.selectPlate: 个人中心--我的帖子--主题--选择板块测试	8	8	0	0	Detail
my_message_sta.personMessage: 个人中心--我的消息---个人消息	5	5	0	0	Detail
my_message_sta.remindTest: 个人中心--我的消息---提醒	3	3	0	0	Detail
my_message_sta.systemMessage: 个人中心--我的消息---系统消息	2	2	0	0	Detail
my_setting_sta.activityInfo: 个人中心--我的设置---活动信息	5	5	0	0	Detail
my_setting_sta.baseInfo: 个人中心--我的设置---基本资料	12	12	0	0	Detail
my_setting_sta.integral: 个人中心--我的设置---积分	1	1	0	0	Detail
my_setting_sta.modifyAvatar: 个人中心--我的设置---修改头像	1	1	0	0	Detail
my_setting_sta.personalInfo: 个人中心--我的设置---个人信息	6	6	0	0	Detail
my_setting_sta.privacy: 个人中心--我的设置---隐私筛选	5	5	0	0	Detail
my_setting_sta.professionInfo: 个人中心--我的设置---职业信息	6	6	0	0	Detail
my_setting_sta.userGroup: 个人中心--我的设置---用户组	3	3	0	0	Detail
my_vest_sta.myvestTset: 个人中心--我的马甲	6	6	0	0	Detail

图 11.4　BBS 自动化测试报告

第 12 章
BDD框架之Lettuce入门

自本章开始，所介绍的技术与前面的章节并无必然联系。之所以要选择这几种技术，是因为它们与开发或测试工作相关，学习和了解他们会提高我们的测试技能。希望读过此书后，读者不仅仅是学会了一个自动化测试工具，而是提高了整体的技能，这也是笔者编写本书的初衷。本章介绍的是一个新的概念——BDD。

遗憾的是截至笔者发稿，本章中介绍的 Lettuce 框架还尚不支持 Python 3，需要读者将环境切换到 Python 2。Python 2 的安装方式与 Python 3 相同，具体请参考本书第 2 章。

12.1 什么是 BDD

相信读者或多或少听说过 TDD、ATDD、BDD 等概念，那么它们分别指的是什么技术？在什么样的场景下会用到它们呢？。

1）TDD：测试驱动开发（Test-Driven Development）

测试驱动开发是敏捷开发中的一项核心实践和技术，也是一种设计方法论。TDD 的原理是在开发功能代码之前，先编写单元测试用例代码，测试代码确定需要编写什么产品代码。TDD 的基本思路就是通过测试来推动整个开发的进行，但测试驱动开发并不只是单纯的测试工作，而是把需求分析、设计和质量控制量化的过程。TDD 首先考虑使用需求（对象、功能、过程、接口等），主要是编写测试用例框架对功能的过程和接口进行设计，而测试框架可以持续进行验证。

2）ATDD：验收测试驱动开发（Acceptance Test Driven Development）

验收测试驱动开发是一种实践。在准备实施一个功能或特性之前，团队首先需要定义出期望的质量标准和验收细则，以明确且达成共识的验收测试计划（包含一系列测试场景）来驱动开发人员的功能开发实现和测试人员的测试脚本开发。面向开发人员，强调如何实现系统以及如何通过验收测试。

3）BDD：行为驱动开发（Behavior Driven Development）

行为驱动开发是一种敏捷软件开发技术，它鼓励软件项目中的开发者、QA、非技术人员或商业参与者之间的协作。主要是从用户的需求出发，强调系统行为。BDD 最初由 Dan North 在 2003 年命名，它包括验收测试和客户测试驱动等极限编程的实践，作为对测试驱动开发的回应。

不同语言下的 BDD 框架：

Cucumber（Ruby）	https://cucumber.io
Jdave（Java）	http://jdave.org
Behat（PHP）	http://docs.behat.org/en/v2.5
Behave（Python）	http://pythonhosted.org/behave
Lettuce（Python）	http://Lettuce.it

基于 Ruby 编写的 Cucumber 由于发展较高且比较成熟，在 BDD 领域有相当的知名度。而 Lettuce 可以看作是 Python 版的 Cucumber。它用于 Python 项目的自动化测试，它可以执行纯文本的功能描述，一个非常有用且迷人的 BDD（行为驱动开发）框架。除官方文档外，关于 Lettuce 的介绍并不多，所以本章的讲解也以官方文档为基础。

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行为，你很难想象这些描述可以自动测试你的系统，如图 12.1 所示。

- [a]描述的行为。
- [b]用 Python 定义步骤。
- [c]运行并观看它失败。
- [d]编写代码以使其通过。

- [e]运行并观看它通过。

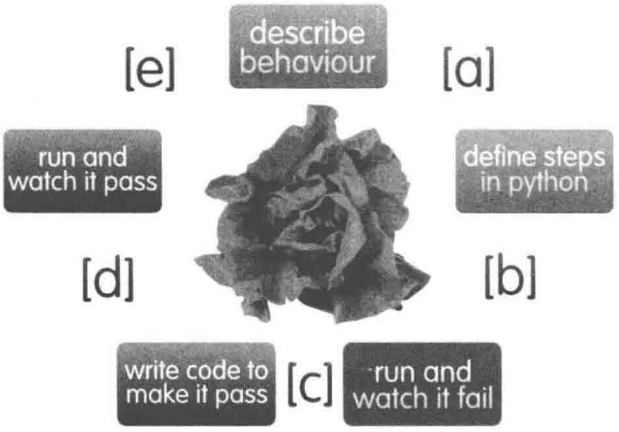

图 12.1　Lettuce 功能描述

12.2　安装 Lettuce

Lettuce 官方网址：http://Lettuce.it/。

最方便的是通过 pip 安装 Lettuce。

cmd.exe

```
> pip install lettuce
```

安装好 Lettuce 后，打开 Windows 命令提示符，在任意目录下输入“Lettuce”命令，如图 12.2 所示。

图 12.2　Lettuce 命令

如果提示“could not find features at \features”（哎呀！在 features 目录下没有找到 features），说明已经安装成功。因为还没有创建 Lettuce 项目，所以会出现如图 12.2 所示的提示。

12.3　阶乘的例子

下面借鉴官网的例子来讲解 Lettuce 的使用。

12.3.1　什么是阶乘

什么是阶乘？一个正整数的阶乘（英语：factorial）是所有小于及等于该数的正整数的积，0 的阶乘为 1。计算方法：

0！=1

1！=1

2！=2×1=2

3！=3×2×1=6

……

10!=10×9×8×7×6×5×4×3×2×1=3628800

……

下面是用 Python 语言实现阶乘的两种方式。

factorial.py

```
# 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

if __name__ == '__main__':
    # 调用方法
    print(f1(10))
    print(f2(10))
```

基于对阶乘的理解，来看看 BDD 是如何实现的。

12.3.2 编写 BDD 实现

创建以下目录结构：

```
projects/mymath/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
```

对于 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
```

是不是很接近自然语言的描述？！第一段为功能介绍，描述需要实现什么功能；第二段为场景描述，也可以看作是一条测试用例，当我输入什么数据，执行了什么操作后，预期程序应该返回什么结果。

Lettuce 虽然使用了自然语言的描述，却也有语法规则。非常简单，有以下几个关键字就可以了：

- Feature（功能）
- Scenario（情景）
- Given（给定）
- And（和）
- When（当）
- Then（则）

Lettuce 关键字的含义与单元测试框架中概念的对比如表 12.1 所示。

表 12.1　Lettuce 关键字的含义与 unittest 中概念的对比

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 run（触发测试执行）
Then（测）	assert（断言，验证结果）

有了上面的 zero.feature 文件的作指导，下面打开 steps.py 文件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_fatorial(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
```

初次接触这样的代码结构可能会产生许多疑问，下面我们就来逐步分析 steps.py 和 zero.feature 是如何产生联系的。

```
import  lettuce import *
```

① 引入 Lettuce 下面的所有类和方法。

```
@step('I have the number (\d+)')
def have_the_number(step, number):
    world.number = int(number)
```

@step 是 Python 装饰器的写法，也就是 have_the_number()函数由@step()进行装饰。

I have the number (\d+)对应于 zero.feature 文件中的第六句：“Given I have the number 0 ”。

(\d+) 是一个正则表达式，\d 表示匹配一个数字，+ 表示匹配的数字至少有一个或多个。关于 Python 的正则表达式，读者可以自行查阅相关资料。

② 定义一个 have_the_number 函数，把@step(I have the number (\d+))匹配到的数字 0 作为函数的入参，然后将其转换成整型（int）赋值给 world.number 变量。

```
@step('I compute its factorial')
def compute_its_fatorial(step):
    world.number = factorial(world.number)
```

③ 把 have_the_number()函数中 world.number 的变量值 0 作为 factorial() 函数的入参，并把 factorial() 函数的返回值赋值给变量 world.number。

I compute its factorial 对应于 zero.feature 文件中的第七句：“When I compute its factorial。”

```
def factorial(number):
    number = int(number)
    if (number == 0) or (number == 1):
        return 1
    else:
        return number
```

④ 计算整数的阶乘，在第二步中调用此函数。判断参数如果等于 0 或 1 的话就直接返回 1，否则返回具体的数。该函数在 have_the_number()中被调用。

```
@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 zero.feature 文件中的预期结果，与第三步返回的实际结果 world.number 进行对比；通过 assert 函数进行断言结果是否正确。

I see the number (\d+)对应于 zero.feature 文件中的第八句：“Then I see the number 1。”

运行 Lettuce

运行 cmd，切换到 tests 目录下，执行“Lettuce” 命令，如图 12.3 所示。

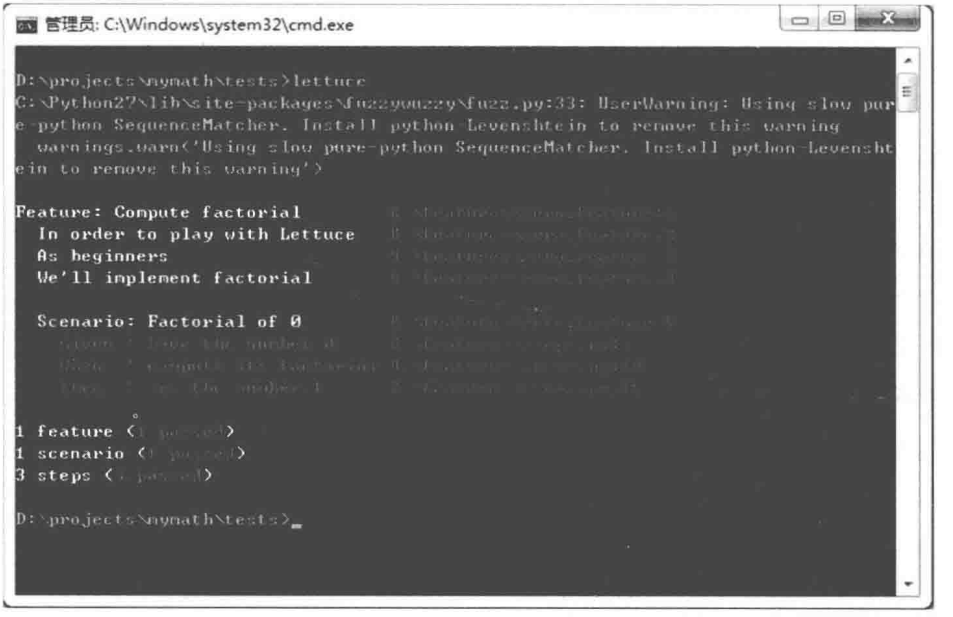

图 12.3 执行 Lettuce 命令

运行过程很清晰，首先是 zero.feature 文件里的功能描述（feature），然后是场景（scenario）每一步所对应的 steps.py 中的哪一行代码。

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。

12.3.3 添加测试场景

接下来我们在 zero.feature 中继续添加场景（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”命令进行测试，如图 12.4 所示。

第四个场景没能通过，3!的预期结果为 6，与实际执行结果不相符，断言失败。如果读者细心的话一定发现了 steps.py 中的 factorial()函数并未正确的实现阶乘，下面修改 factorial()函数的代码。

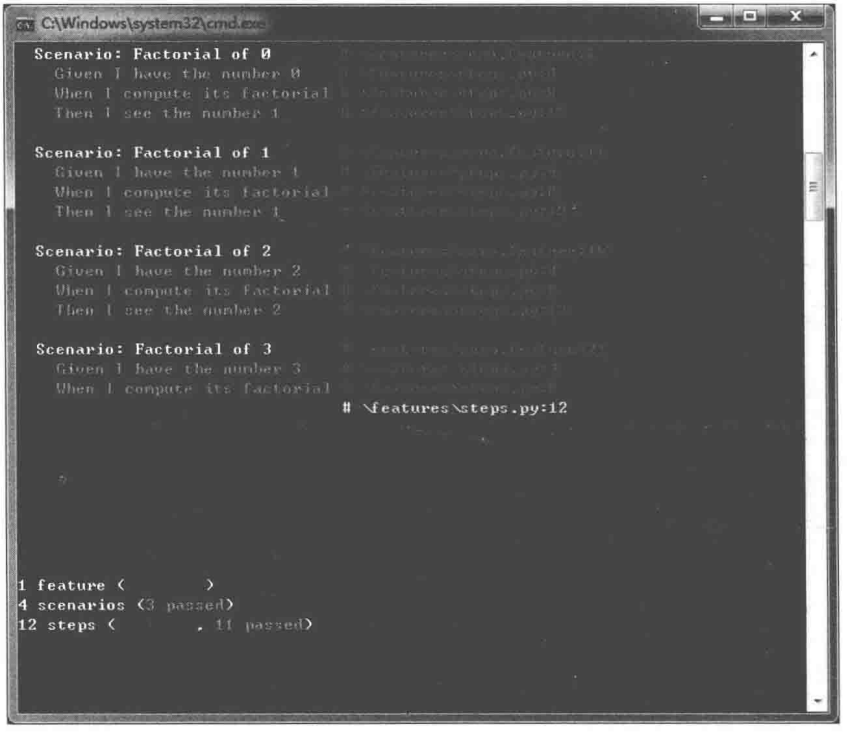

图 12.4　执行 Lettuce 失败

steps.py

```
.....
def factorial(number):
    number = int(number)
    if (number == 0) or (number == 1):
        return 1
    else:
        return number*factorial(number-1)
```

之前的代码只判断计算的数字是否为 0 或 1，对于 1 以上的数字，则直接返回数字本身，并未进行阶乘计算。修改代码后，通过递归的方式实现了阶乘。再次执行 Lettuce 进行验证，结果如图 12.5 所示。

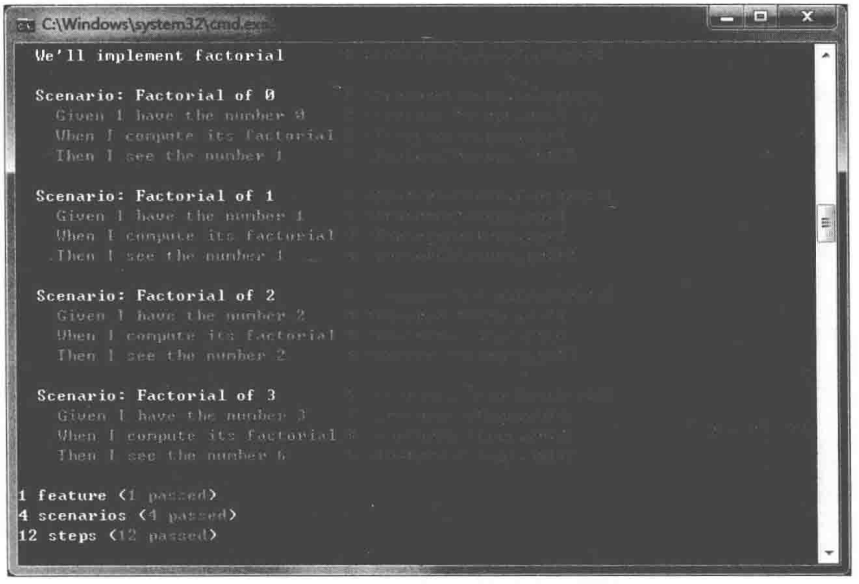

图 12.5　再次执行 Lettuce 成功

12.3.4　Lettuce 目录结构与执行过程

通过对计算阶乘例子的学习，我们了解到 BDD 开发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 程序文件，Feature 文件中的每一个 Given-When-Then 步骤在 Step 文件中都有对应的 Python 执行代码，两类文件通过正则表达式相关联。

另外需要注意的是，Feature 文件一定要在 features 目录下，否则会提示“could not find features at \features”。而 Step 文件可放在任意目录下都能被执行到。

12.4　Lettuce_webdriver 自动化测试

Lettuce_webdriver 属于独立的 Python 第三方扩展，它支持通过 Lettuce 运行 Selenium WebDriver 自动化测试用例。

安装 Lettuce

参考第 12.2 节。

安装 Lettuce_webdriver

Lettuce_webdriver 下载地址：https://pypi.python.org/pypi/Lettuce_webdriver。

可以直接通过 pip 进行安装。

cmd.exe

```
> pip install lettuce_webdriver
```

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 框架。

nose 下载地址：https://pypi.python.org/pypi/nose/

nose 同样支持 pip 的安装方式。

cmd.exe

```
> pip install nose
```

同样以百度搜索为例，首先创建如下目录结构：

```
tests/features/
    ├── step_definitions/
    │   └── setps.py
    ├── support/
    │   └── terrain.py
    └── baidu.feature
```

打开 baidu.feature 文件，遵循 BBD 行为描述的规则，编写如下内容。

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 "kw" with "selenium"
  And  I click id "su" with baidu once
  Then I should see "seleniumhq.org" within 2 second
  Then I close browser
```

接下来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，用于定义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，作用于所有测试用例。

在.../tests/目录下输入“Lettuce”命令执行测试脚本，如图 12.6 所示。

图 12.6 执行 Lettuce_webdriver 测试脚本

打开 biadu.feature 文件，继续添加新的测试场景：

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 "kw" with "selenium"
    And  I click id "su" 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 "kw" with "webdriver"
    And  I click id "su" with baidu once
    Then I should see "www.w3.org/TR/webdriver/" within 2 second
    Then I close browser
```

唯一需要注意的是，浏览器的关闭“Then I close browser”放在最后一个场景执行，否则会出现错误。再次通过“Lettuce”命令执行自动化测试脚本，如图 12.7 所示。

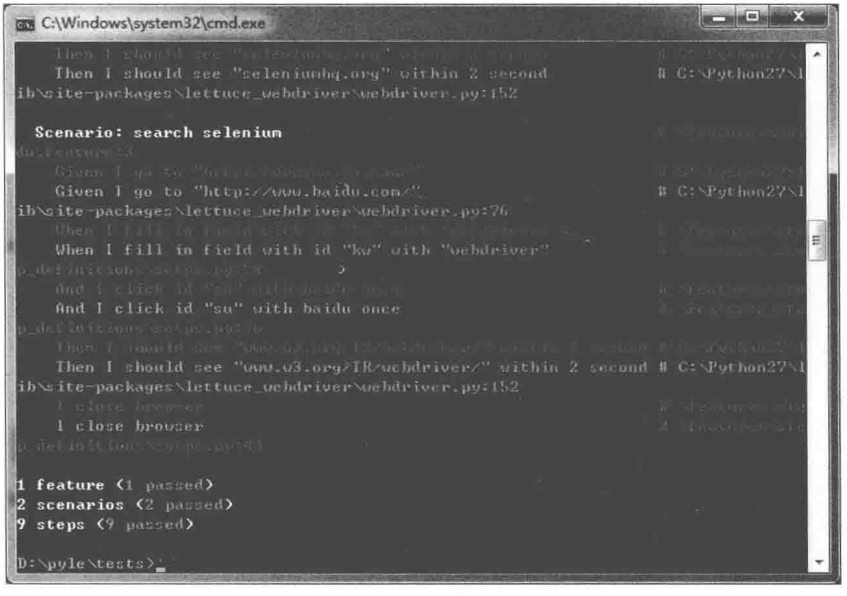

图 12.7　再次执行 Lettuce_webdriver 测试脚本

本章小结

本章介绍了什么是 BDD，并以 Lettuce 为例介绍了如何通过 BDD 模式开发软件功能。最后，结合本书的主题，介绍了如何通过 Lettuce_webdriver 进行 Web 自动化测试的编写。当然，本章的讲解只是抛砖引玉，关于 BDD 的更多使用请参考官方文档与相关书籍。

第 13 章和第 14 章内容可到博文视点网站免费下载

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

Selenium 2 自动化测试实战 基于Python语言

基于Python语言实现，以实例的方式详细讲解了WebDriver的类库体系。不仅可以为初学者提供教科书级的系统指导，还可以帮助自动化工程师加深对WebDriver原理的掌握。

——畅捷通支付测试经理，狄枫

在经历互联网浪潮后，自动化测试已被行业所重视，而Selenium作为Web自动化测试的首选利器越来越受到业界青睐。本书作为Selenium自动化测试的入门书籍，由浅入深地介绍了Selenium各方面，并在最后结合实际项目，给出基于Selenium的自动化测试实践经验，尤为可贵，本书非常适合测试人员阅读与参考。

——多玩资深测试开发工程师，陈曦明（hyddd）

本书源于技术讨论和分享，写于资深测试工程师；从测试基本概念讲起，并用Python语言带领大家窥探UI自动化测试的一斑。

——钱包金服测试主管，MarkRabbit

本书分量很足，对必要的知识点拓展拿捏也很到位，可以作为UI自动化测试人员的案头书。

——互众广告公司测试主管，多可笑

上架建议：软件测试

ISBN 978-7-121-27853-2

9 787121 278532 >

定价：59.00元

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

责任编辑：安　娜
封面设计：侯士卿

[General Information]
书名=SELENIUM 2 自动化测试实战　基于PYTHON语言
作者=虫师编著
页数=308
SS号=13918627
DX号=
出版日期=2016.01
出版社=电子工业出版社