

白洛 编著



基于Selenium 2 的 自动化测试

——从入门到精通

★ 零起点使用Selenium顶级大全

★ 零起点读者必看的经典级读本

★ 理论知识与实战代码完美结合

★ 知识深度以及广度的全面展开



基于 Selenium 2 的自动化测试 ——从入门到精通

白 洛 编著



机械工业出版社

本书向开发人员和测试人员展示了如何使用 Selenium 进行 Web 自动化测试。本书从自动化测试的特点娓娓道来，引出了主角 Selenium；介绍了 Selenium IDE 的使用；讲述了获取页面元素和定位页面元素的多种方式；讲解了 WebDriver 与 Selenium RC 的区别、WebDriver 的架构和设计理念；阐述了 WebDriver 的部署、基本使用方法、对 HTML5 特性的支持，以及如何迁移已有的 Selenium RC 代码到 Selenium WebDriver 的解决方案；展示了在嵌入式系统中使用 Selenium 进行自动化测试的方法，涵盖 Android、iOS 和 Raspberry Pi 等；此外，本书还描述了 Selenium Grid 的架构和部署方法；最后介绍了 Selenium 周边的测试工具和套件。无论从深度还是广度上，本书为开发人员和测试人员学习并掌握 Selenium 提供了一定的辅助作用。

本书适合开发人员、测试人员、测试管理人员使用，也适合作为大中专院校相关专业师生的学习用书，以及培训学校的教材。

图书在版编目（CIP）数据

基于 Selenium 2 的自动化测试：从入门到精通 / 白洛编著. —北京：机械工业出版社，2014. 7

ISBN 978-7-111-46783-0

I. ①基… II. ①白… III. ①软件工具-自动检测 IV. ①TP311. 56

中国版本图书馆 CIP 数据核字（2014）第 106088 号

机械工业出版社（北京市百万庄大街 22 号 邮政编码 100037）

策划编辑：张俊红 责任编辑：闻洪庆 版式设计：赵颖喆

责任校对：张玉琴 封面设计：路恩中 责任印制：李 洋

北京市四季青双青印刷厂印刷

2014 年 8 月第 1 版第 1 次印刷

184mm×260mm • 13.75 印张 • 334 千字

0 001 — 3 000 册

标准书号：ISBN 978-7-111-46783-0

定价：39.80 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务

网络服务

社服中心：(010) 88361066 教材网：<http://www.cmpedu.com>

销售一部：(010) 68326294 机工官网：<http://www.cmpbook.com>

销售二部：(010) 88379649 机工官博：<http://weibo.com/cmp1952>

读者购书热线：(010) 88379203 封面无防伪标均为盗版

前 言

多年以前，自动化测试还不够成熟。人们往往把自动化测试当作手工测试的附属品，就像当年把测试当成开发的附属品一样。从 1997 年的开源运动开始，开源软件在过去的十多年间蓬勃发展。大到操作系统，小到一个功能模块，开源运动已经渗透到各个领域并结出丰硕的果实。在软件测试领域，特别是自动化测试领域，开源软件涵盖了方方面面。从单元测试、功能测试到性能测试，从 Web 页面测试到数据库、多媒体、通信等应用领域的测试，都可以发现开源软件的身影。开源软件给自动化测试技术带来巨大变革和进步。

Web 自动化测试从无到有、由浅入深，已经逐步走向成熟。互联网的出现亦改变了许多软件研发的模式。在互联网和软件产业，一切变化都如此迅速，以至于许多最近几年才出版的软件测试方面的书籍内容已跟不上软件更新迭代的速度。许多红极一时的测试工具和测试实践，在当前的环境下效率会大打折扣。作为 Web 测试工具中的佼佼者——Selenium 已经走过了近 10 个春秋，从 Selenium 1 发展到 Selenium 2，可谓“十年磨一剑”。它是如何做到与时俱进并继续引领业界潮流的呢？本书将由浅入深地展示 Selenium 的独特魅力之所在。

而国内的 Web 自动化测试资源，经过这么多年的发展，已然在自动化测试领域遍地开花。无论测试人员的水平，还是测试工具的成熟度，抑或测试资料的完备性，都已达到蓬勃发展的阶段。正所谓“乱花渐欲迷人眼”，如何在众多的 Web 自动化测试工具和资料中选取最适合自己的学习、并且能应用到实际项目中的测试套件，已经成为不少自动化测试人员犹豫不决的选择题。

许多刚入行的测试人员希望获取入门级的书籍，以此引导他们入门和上手某测试工具。本书由浅入深，从最基本的 Selenium IDE 的使用进行介绍并逐步深入，进而讲解了 Selenium 2/WebDriver 的架构、功能、使用等方面的相关知识点；除了纵向的知识深入，本书还横向涵盖了一些拓展知识点，包括 Web HTML5、嵌入式系统的自动化测试、软件持续集成、其他 Web 自动化测试工具等。希望本书不仅仅只是一本为初级测试人员准备的入门书籍，还能成为资深测试人员的案头参考资料。

为了便于读者实践，本书还提供了配套的代码包和工具包，基本囊括了本书中的示例代码、配置文件和自动化脚本文件。所有的示例代码文件、配置文件和自动化脚本文件都以章节进行了划分，并且通过第 4 章的内容配置好 WebDriver 和 Eclipse 后，导入相关的 Java 示例代码即可运行。而配套的配置文件和自动化脚本文件可根据对应章节所阐述的方法部署到合适的环境中运行。

特别感谢我的幺爷爷黄忠霖教授，我才有机会结识机械工业出版社电工电子分社张俊红副社长。同时感谢张俊红副社长，没有他的鼓励、支持和指导，我很难有勇气和毅力完成这样一本书。此外，还需要感谢张讯讯和张慧智在文字方面的审校工作。最后，要特别感谢我的妻子黄静嶧以及家人对我的理解和支持。

由于水平和时间的限制，书中难免会存在一些错误，还望见谅，并恳请读者提出宝贵意见和建议。

作者于上海

目 录

前言

第 1 章 初识 Selenium	1
★1.1 简介	1
★1.2 自动化测试	1
★1.3 Web 自动化测试	3
★1.4 Selenium 的前世今生	3
★1.5 Selenium 1	4
★1.6 Selenium 2	5
★1.7 Selenium 3	6
★1.8 Selenium IDE	6
★1.9 Selenium Grid	6
★1.10 Selenium 与嵌入式	7
★1.11 Selenium 与云计算	7
★1.12 小结	8
第 2 章 牛刀小试之 Selenium IDE	9
★2.1 简介	9
★2.2 安装 Selenium IDE	9
★2.3 Selenium IDE 界面一览	11
★2.4 创建测试用例	13
★2.5 存储页面信息	14
★2.6 与 AJAX 页面进行交互	15
★2.7 处理多窗口	16
★2.8 Rollup 的简介	17
★2.9 小结	21
第 3 章 Selenium 玩转页面元素	22
★3.1 简介	22
★3.2 浏览器调试工具	22



3. 2. 1 Google Chrome	22
3. 2. 2 Mozilla Firefox	23
3. 2. 3 Internet Explorer	24
★3. 3 查找页面元素	26
3. 3. 1 通过 ID 查找元素	26
3. 3. 2 通过 Name 查找元素	27
3. 3. 3 通过 ClassName 查找元素	28
3. 3. 4 通过 TagName 查找元素	29
3. 3. 5 通过 LinkText 查找元素	30
3. 3. 6 通过 PartialLinkText 查找元素	31
3. 3. 7 通过 CSS 选择器查找元素	32
3. 3. 8 通过 XPath 查找元素	32
3. 3. 9 通过 jQuery 查找元素	34
★3. 4 元素的 Actions	40
★3. 5 小结	43

第 4 章 初识 Selenium WebDriver 44

★4. 1 简介	44
4. 1. 1 概述	44
4. 1. 2 WebDriver 与 Selenium RC 的区别	44
★4. 2 WebDriver 的架构	44
4. 2. 1 synthesized 事件和 native 事件	44
4. 2. 2 RPC 调用	45
4. 2. 3 兼容性矩阵	45
4. 2. 4 缺陷	46
4. 2. 5 与 DOM 交互	46
★4. 3 WebDriver、Eclipse 和 Java	47
★4. 4 WebDriver 的部署	49
4. 4. 1 使用 Firefox Driver	49
4. 4. 2 使用 Chrome Driver	52
4. 4. 3 使用 Internet Explorer Driver	56
★4. 5 WebDriver 与浏览器	60
4. 5. 1 操作页面元素之单选按钮	60
4. 5. 2 操作页面元素之多选按钮	62
4. 5. 3 操作弹出窗口之验证标题	64
4. 5. 4 操作弹出窗口之验证内容	67
4. 5. 5 操作警告框、提示框和确认框	69
4. 5. 6 操作浏览器最大化	72



4.5.7 操作浏览器 Cookies	73
4.5.8 操作浏览器前进后退	76
4.5.9 操作页面元素等待时间	78
★4.6 WebDriver 与文件系统	79
4.6.1 屏幕截图操作	79
4.6.2 复制文件操作	81
4.6.3 创建目录操作	82
4.6.4 删除目录操作	83
4.6.5 读取文件操作	83
4.6.6 压缩目录操作	84
4.6.7 临时目录操作	85
4.6.8 文件权限操作	85
★4.7 小结	86
第 5 章 玩转 Selenium WebDriver	87
★5.1 WebDriver 与 HTML5	87
5.1.1 HTML5 中的 Video	87
5.1.2 HTML5 中的 Canvas	89
5.1.3 HTML5 中的 Drag/Drop	90
5.1.4 HTML5 中的 Geolocation	94
★5.2 RemoteWebDriver	97
5.2.1 RemoteWebDriver 简介	97
5.2.2 RemoteWebDriver 的优缺点	97
5.2.3 RemoteWebDriver 服务器端	97
5.2.4 RemoteWebDriver 客户端	99
★5.3 WebDriver 的事件处理	100
5.3.1 自定义事件监听	100
5.3.2 事件处理实例	101
★5.4 Page Object 与 Page Factory	103
5.4.1 不使用 Page Object	104
5.4.2 使用 Page Object	108
5.4.3 使用 Page Object、Page Factory、@FindBy 和 How	118
★5.5 Selenium RC 迁移到 WebDriver	130
5.5.1 简介	130
5.5.2 从 Selenium RC 迁移到 WebDriver 的优势	130
5.5.3 迁移 Selenium 运行实例	130
5.5.4 迁移测试代码到 WebDriver API	131
★5.6 小结	131



第6章 Selenium 玩转 Android	132
★6.1 简介	132
★6.2 玩转 Android	132
6.2.1 架构	132
6.2.2 搭建 Android WebDriver 环境	133
6.2.3 最简单的测试用例	137
6.2.4 旋转屏幕	137
6.2.5 触摸和滚动	139
★6.3 当 Android 遇到 HTML5	141
6.3.1 HTML5 中的 Web Storage	141
6.3.2 HTML5 中的 Application Cache	143
★6.4 在 Cloud 中测试 Android	145
★6.5 小结	148
第7章 Selenium 玩转 iOS	149
★7.1 简介	149
★7.2 ios-driver	149
7.2.1 ios-driver 简介	149
7.2.2 ios-driver 的 Web app 实例	150
7.2.3 ios-driver 的 Native app 实例	153
7.2.4 ios-driver 的源码编译	158
★7.3 Appium	160
7.3.1 Appium 简介	160
7.3.2 Appium 的 iOS 配置	161
7.3.3 Appium 的 Web app 实例	162
★7.4 小结	169
第8章 Selenium 玩转 Raspberry Pi	170
★8.1 简介	170
★8.2 操作系统层面的准备工作	170
★8.3 依赖包的安装	171
★8.4 运行 Python 版的 Selenium	172
★8.5 运行 Standalone 版的 Selenium Server	175
★8.6 小结	179
第9章 Selenium Grid	180
★9.1 简介	180



9.1.1	Selenium Grid 是什么	180
9.1.2	何时使用 Selenium Grid	180
9.1.3	Selenium Grid 2.0 & 1.0	181
★9.2	Selenium Grid 的架构	181
★9.3	Selenium Grid 的部署	182
★9.4	Selenium Grid Hub	182
9.4.1	默认启动 Hub	182
9.4.2	配置 Hub 端口	182
9.4.3	JSON 配置文件	184
★9.5	Selenium Grid Node	184
9.5.1	默认启动 Node	184
9.5.2	注册 Mac OS X & Opera	185
9.5.3	注册 Linux & Firefox	187
9.5.4	注册 Windows & Internet Explorer	187
9.5.5	注册 Android & Chrome	188
9.5.6	注册 Appium-iOS & Safari	189
9.5.7	注册多个不同类型的浏览器	190
★9.6	编写 Selenium Grid 的测试用例	193
★9.7	小结	194
第10章 Selenium 的“兄弟姐妹们”		195
★10.1	简介	195
★10.2	Jenkins	195
★10.3	Web 前端性能	198
10.3.1	BrowserMob Proxy	198
10.3.2	HttpWatch	200
★10.4	Ruby 的光芒	203
10.4.1	Watir-WebDriver	203
10.4.2	Capybara	204
★10.5	JMeter	205
★10.6	Sikuli	208
★10.7	小结	209
参考文献		210

第 1 章

初识 Selenium

1.1 简介

Selenium 中文释义为“硒”，在化学中是一种非金属元素。可以用作光敏材料、电解锰行业催化剂等，是动物体必需、对植物有益的营养元素。

本书要介绍的 Selenium 是用于 Web 应用程序的自动化测试工具。基于 Selenium 的测试用例会直接运行在浏览器中，就像真正的用户在操作一样。其支持的浏览器范围非常广泛，包括各个平台的主流浏览器。Selenium 的主要功能包括：

- 1) 功能性测试：创建回归测试验证软件功能和用户需求。
- 2) 兼容性测试：测试应用程序在不同的操作系统和不同的浏览器中是否运行正常。

值得注意的是，Selenium 并不适用于进行网站后台性能方面的测试。但是结合其他第三方工具，Selenium 可被用于对网站前端性能进行适当的评估。

Selenium 源于 ThoughtWorks 公司，已经在全球范围内遍地开花。Selenium 能在测试领域成为一棵常青树，与其开源性和开放性密不可分。感谢开源先驱 Richard Stallman，为我们这个世界带来了开源运动这个先进的理念并为之奋斗一生。感谢 Jason Huggins 为我们创造了 Selenium 和现在流行的嵌入式自动化测试套件 Appium。

正如 Selenium 的化学特性一样，它作为测试领域的有益元素，已经为千千万万的测试人员带来了营养丰富的盛宴，滋养着一代又一代的测试同僚们。

1.2 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。手工测试通常是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程一步步执行测试，得到实际结果并将其与期望结果进行比较。在此过程中，为了节省人力、时间或者硬件资源，提高测试效率，便引入了自动化测试的概念。通过自动化测试可以极大地提升回归测试、稳定性测试和兼容性测试的工作效率，在保障产品质量和持续构建等方面起到举足轻重的作用。特别是在敏捷开发模式下，自动化测试更是必不可少的步骤。

近年来，测试领域和几年前相比发生许多令人欣喜的变化：

- 1) 敏捷开发模式的盛行掀起自动化测试的一轮热潮，测试和开发合作越来越密切。
- 2) 测试工作的技术性越来越强，以往常见的“基于开源软件提升开发效率”的模式也被广泛应用到测试工作中。



3) 组件化、软件产品线开发模式的进一步成熟，开发效率和测试效率随之进一步得到提高。

自动化测试相较于手工测试的优势在于：

- 1) 自动化测试可以完成某些手工测试难以完成的工作，如并发测试、压力测试等。
- 2) 自动化测试可以提高手工测试的工作效率，如执行具有多个重复步骤的测试用例。
- 3) 自动化测试在敏捷开发过程中，可以快速验证代码修改的正确性。
- 4) 自动化测试和手工测试相辅相成，互相促进。

实施自动化测试前需要对软件开发过程进行分析，以观察其是否适合使用自动化测试。通常需要满足以下几个条件：

1) 需求变动不频繁。测试脚本的稳定性决定了自动化测试的维护成本。如果软件需求变动过于频繁，测试人员需要根据变动的需求来更新测试用例和相关的测试脚本，而脚本的维护本身就是一个代码开发的过程，需要修改和调试，必要时还需要修改自动化测试的框架。如果耗费的成本高于节省的测试成本，那么自动化测试便是失败的。如果项目中的某些模块相对稳定，而某些模块需求变动性很大，可以针对相对稳定的模块进行自动化测试，而变动较大的仍采用手工测试。

2) 项目周期足够长。自动化测试需求的确定、框架的设计、测试脚本的编写和调试都需要相当长的时间来完成，这个过程本身就是一个测试软件的开发过程，需要较长的时间来完成。如果项目的周期比较短，没有足够的时间去支持这样一个过程，那么自动化测试便成为笑谈。

3) 自动化测试脚本可重复使用。如果费尽心血开发了一套近乎完美的自动化测试脚本，而脚本的重复使用率很低，致使期间所耗费的成本大于所创造的经济价值，自动化测试便成为了测试人员的练手之作，而并非是真正可产生效益的测试手段。

4) 手工测试无法完成的测试工作。某些测试采用手工的方式无法完成，或者需要投入大量时间与人力，此时就可以考虑引入自动化测试，如性能测试、配置测试、兼容性测试、大数据量输入测试等。

自动化测试虽然有如此多的优势，那是不是意味着自动化测试就是“包治百病”的软件“银弹”呢？有些人可能会有如下误区：

1) 自动化测试是一种比人工测试更先进、更高级的测试手段。自动化测试既有自身的优点，也有其局限性。例如对于需求不明确，或者界面经常发生变动的产品就不适合使用自动化测试。自动化测试与手工测试的关系应该是相辅相成，互相弥补各自的局限性，相互促进。

2) 所有的手工测试都应该被 100% 的自动化。一味片面地追求自动化率，不仅软件的质量得不到提高，而且还会让测试人员疲于奔命，投入和产出的性价比很低。有不少负面测试就只能通过手工测试的方式完成并进行验收。自动化测试不是万能的，需要根据实际情况引入并有的放矢地设定其覆盖率。

3) 自动化测试能够发现大量的缺陷，它比手工测试更有效。实际情况是，自动化测试只能发现 30% 以下的软件缺陷，而手工测试反而能发现更广泛且很深层次的问题。自动化测试在回归测试时可以节省很多时间并快速验收，但这并不意味着其发现问题的能力比手工测试更强。单从发现缺陷的角度而言，自动化测试的效率低于手工测试。



4) 即使一次性的软件项目也应该采用自动化测试。自动化测试的投入成本，至少要在好几个发布版本之后才能体现其价值。因此对于一次性的软件项目，应该避免采用自动化测试方案。

5) 自动化测试只是测试工程师的事情，与开发人员没有关系。在软件开发过程中，首先需要考虑软件本身的可测试性。如果开发人员一开始就不把软件的可测试性考虑进来，会导致开发的软件难以测试，甚至无法实现自动化测试。

6) 商业自动化测试工具更靠谱，一定要选用商业自动化测试工具。就自动化测试工具而言，测试团队应该根据自身实际情况来选择自动化测试工具。商业自动化测试工具有技术团队进行支持，遇到问题也许能尽快得到支持。但是如果有特殊的需求，这类软件往往没有自由的可定制功能。而开源自动化测试工具由于源代码都是开放的，如果团队有特殊的定制需求，可以由测试团队自行修改开源自动化测试工具来满足团队需要。

1.3 Web 自动化测试

当前绝大多数企业应用都是基于 Web 的应用系统，人们可以通过 Web 浏览器便捷地访问它们。现在炙手可热的“云计算”大潮也将这一趋势推向了高潮。很多组织和公司采用持续改进的开发模式来应对这种趋势，并采用自动化测试来适应这种高强度的持续开发的模式。相较传统桌面软件的自动化测试，Web 自动化测试指的是 Web 应用系统从用户界面层面进行的自动化测试，通过用户界面测试内部的业务逻辑。Web 自动化测试的自身特点如下：

- 1) Web 页面上出现的元素可能具有不确定性。
- 2) 不同操作系统上不同 Web 浏览器之间的兼容性。
- 3) Web 应用的高并发性和容错性。
- 4) 移动设备上的 Web 客户端兼容性、旋转性和各种触摸特性。

Web 自动化测试一直都不是一件容易的事情。尤其是在研发团队广泛采用 UI 框架和敏捷开发来提升交付效率的今天，Web 自动化测试变得愈发困难；未来随着 Web 组件式开发技术和云计算技术的逐渐成熟和落地，又会对 Web 自动化测试提出怎样的挑战？Web 开发过程中 UI 框架的广泛采用极大提高了开发效率和用户体验，也从技术的角度保障了敏捷开发的蓬勃发展。然而 UI 框架自动生成的海量页面源码却让原本就举步维艰的 Web 自动化测试变得雪上加霜：测试脚本的维护成本更大、回放稳定性更差、断言更困难。Web 组件式框架的采用可以让用户非常容易地卸载、替换软件的部分模块。原本是个“整体”的软件被“分解”之后可支持灵活组装。我们又该如何保障按照模块依赖关系组装出的所有软件版本都是可用的呢？云计算又能为 Web 自动化测试带来哪些机遇呢？关于 Web 自动化测试，有如此多的问题摆在我们面前，就让我们勇敢地去面对和迎接这些新的挑战吧。

1.4 Selenium 的前世今生

早在 2004 年，Jason Huggins 还供职于 ThoughtWorks 公司。他当时正在开发公司内部的时问和费用系统，该系统使用了大量的 JavaScript 代码来进行构建。虽然当时 Internet Explorer 还



占有绝对的统治地位，但是 ThoughtWorks 公司内部有不少员工在使用其他类型的浏览器，尤其是 Mozilla 基金会的浏览器。当员工使用不同的浏览器访问这个时间和费用系统时，一旦发现异常，就会提交 Bug 报告。当时的开源测试工具一般只支持占主流地位的 Internet Explorer 浏览器，或者仅仅只是模拟浏览器的行为，如 HttpUnit。购买商业工具授权的成本对于这个小型项目的有限预算而言也不太现实。

正是在这样的情况下，Jason Huggins 和他所在的团队决定自主开发一个基于浏览器并且采用 JavaScript 编程语言的测试工具。Selenium 自动化测试工具自此诞生，并且在 2004 年基于 Apache 2.0 开源协议对外发布，正式命名为 Selenium Core。

Selenium Core 的设计之初并不能绕过浏览器的“同源”规则，因此开发人员不得不将待测试的产品、Selenium Core 和测试脚本均部署到同一台服务器上来完成自动化测试工作。但在实际研发过程中，将上述各项分拆在不同的机器上运行的需求却与日俱增。为了解决这个问题，Jason Huggins 所有的研发团队编写了 HTTP 代理，以让 Selenium 截获所有的 HTTP 请求。使用 HTTP 代理的优势之一就是可以绕过浏览器的“同源”规则，让待测试的产品、Selenium Core 和测试脚本三者分散在不同的机器上。此外，这个代理的设计方式还使得采用多语言编写 Selenium 测试脚本的能力成为可能，因为测试脚本只需要关心将标准的 HTTP 请求发送到指定的 URL 即可，而 Selenium 本身并不需要关心这些 HTTP 请求是由什么程序语言编写而成。正是由于这种离散的分布方式使得测试脚本可以远程控制浏览器并执行测试用例，Selenium Remote Control 也因此得名，也可简称为 Selenium RC。Selenium RC 主要包括两个部分：一个是 Selenium RC Server；另一个是提供各种编程语言绑定的客户端驱动。

时间一晃就到了 2007 年年初，ThoughtWorks 公司发布了酝酿已久的 WebDriver 雉形。WebDriver 的设计理念是将端到端的测试与底层具体的测试工具隔离开，并采用了设计模式中常见的适配器（Adapter）模式来达到目标。

Selenium RC 与 WebDriver 最为显著的差异体现在 API 的组织上。Selenium RC 采用了基于字典方式的 API，而 WebDriver 的 API 则更加地面向对象。此外，Selenium Core（Selenium RC 的核心）基本上是 JavaScript 应用，主要运行在浏览器的安全沙箱中。而 WebDriver 则是原生绑定到浏览器中并绕开了浏览器的安全模型，其代价就是框架本身的开发投入显著增加，每个浏览器的 WebDriver 都需要单独实现。

常言道，合久必分，分久必合。历史的车轮运转到公元 2009 年 8 月，Selenium RC 和 WebDriver 项目宣布合并，也就是后来大家熟知的 Selenium WebDriver。合并以后，WebDriver 也支持了多语言绑定，克服了最初只支持 Java 绑定的缺陷。同时 Selenium WebDriver 除了支持 PC 上传统的浏览器 Chrome、Firefox、Internet Explorer，还支持嵌入式设备上基于 Webkit 内核的浏览器，包括 Android、iOS 上的浏览器等。

1.5 Selenium 1

Selenium 1 即我们所熟知的 Selenium RC。其已经为众多的测试机构和部门服役了多年并贡献卓越。Selenium RC 的典型使用方式如下：

- 1) 测试人员基于客户端驱动所提供的 API 来编写测试用例脚本。
- 2) 测试程序打开浏览器，此时 Selenium RC Server 绑定 Selenium Core 并自动将它嵌入



到浏览器中。Selenium Core 实际上是一系列 JavaScript 函数，它们使用浏览器内置的 JavaScript 翻译器来翻译和执行 Selense Command。

3) 客户端驱动持续执行测试用例脚本并发送特定的命令到 Selenium RC Server。这些特定的命令即 Selense Command。

4) Selenium RC Server 解释 Selense Command，并触发 Selenium Core 执行对应的 JavaScript 代码来完成相应操作。

5) 浏览器上所有的请求和响应都通过 Selenium RC 的 HTTP 代理与实际的 Web 应用服务器进行交互，并且 Selenium RC 一旦收到响应就将页面传递给浏览器。但它会篡改源，使得页面看上去好像来自于与 Selenium Core 同源的服务器（这样 Selenium Core 就遵循了同源规则）。

6) 浏览器接收到 Web 页面后，便在框架或者窗口中展示页面。

但毕竟事物都有更新换代的阶段，Selenium 1 也不例外。正所谓老树发新芽，Selenium 2 就是从 Selenium 1 这棵历史老树上发出的新芽。截至目前，Selenium 1 还继续被维护并提供一些 Selenium 2 尚不支持的功能，如某些编程语言的支持和某些浏览器的支持。因此，为了避免大量的基于 Selenium 1 的测试架构和代码被遗弃，Selenium 团队提供了一种折中的方式来让基于 Selenium 1 的测试架构和代码能继续发挥余热。

某些使用 Selenium 多年的公司和机构，希望能够继续维护基于 Selenium RC 的测试集合，并继续添加新的测试代码；但是还有一部分使用 Selenium 多年的公司和机构，他们希望能将基于 Selenium RC 的测试代码迁移到 WebDriver 上来，让那些老旧但成熟的代码和测试集合重新焕发新生。本书在 5.5 节展示了如何从 Selenium RC 迁移已有代码到 WebDriver 的解决方案。

1.6 Selenium 2

Selenium 2 的主要新特性就是将 WebDriver API 集成进 Selenium RC，从而解决 Selenium 1 所面临的一系列局限性问题。WebDriver 的创建者 Simon Stewart 曾在 WebDriver 和 Selenium 社区中回答了合并的原因：

“WebDriver 和 Selenium 为什么会合并？究其根本，是 WebDriver 和 Selenium 可以互相弥补对方的缺点。而且，Selenium 开源项目的资助者们也希望两者可以合并。我认为这种合并方式就是用户可以获取的最好的组合架构方式。”

WebDriver 与 Selenium RC 合并的结晶就是 Selenium 2。其 API 的设计非常精巧，既易于理解又易于拓展。Selenium 2 不与任何的测试框架绑定，这样便于与其他测试工具进行集成，如 JUnit 或 TestNG 等。

WebDriver 的实现和具体的浏览器相关，包括 HtmlUnit Driver、Firefox Driver、Chrome Driver、Internet Explorer Driver 等。

1) HtmlUnit Driver：HtmlUnit Driver 是目前运行速度最快和最轻量级的 WebDriver 实现。正如其名字所体现的，HtmlUnit Driver 基于 HtmlUnit，优点是纯 Java 实现，所以容易跨平台使用。

2) Firefox Driver：Firefox Driver 是最容易配置和使用的 WebDriver，因为所有的准备工作都伴随 Java 语言绑定的客户端被打包在一起。只要下载 WebDriver Java Client Driver 就能



够使用 Firefox WebDriver。

3) Chrome Driver: Chrome Driver 是针对 Google Chrome 浏览器开发的 WebDriver, 因此其跨平台性也是非常的优异。

4) Internet Explorer Driver: Internet Explorer Driver 只能运行在 Windows 操作系统上, 相较于 Firefox Driver 和 Chrome Driver, 其运行速度略显缓慢。

Selenium 2 相较于 Selenium 1 还有一个重要变化, 用户可以通过 WebDriver 来测试手机应用, 无论在模拟器上还是真实设备上。这是在 Selenium 1 时代无法实现的功能。

1.7 Selenium 3

在本书撰写之际, Selenium 3 也已经由 Selenium 团队提上议程, 并正在紧锣密鼓地开发中。据悉, Selenium 3 会移除原有的 Selenium Core 的实现部分, 并且 Selenium RC 的 API 也将被去掉。其他一些变化包括但不限于以下内容:

- 1) 所有的 Driver 在消息响应中统一使用状态字符串, 而不是状态码, 这样更容易辨识。
- 2) 为基于 Html 格式的测试集合提供一个新的运行器, 特别是针对 Selenium IDE 所导出的 Html 测试集合。其原理是基于 WebDriver-backed 的 RC 实现。
- 3) 将 Selenium 3 的分支和包含 Selenium RC 的代码分支分开进行编译打包。
- 4) 在 WebDriver.quit() 方法之后如果还使用 WebDriver, 则会报错: IllegalStateException。
- 5) 对 WebDriver.quit() 方法的多次调用也是允许的。
- 6) 重构 WebDriver 的构造函数。
- 7) 架构迁移到 Netty 或者 Webbit 服务器上。

1.8 Selenium IDE

Selenium IDE 的优点如下:

- 1) 录制功能快捷方便, 上手快。
- 2) 代码转换功能易用, 容易生成其他编程语言的测试用例代码。
- 3) 支持跨域。
- 4) 不依赖 Java 运行时环境。

Selenium IDE 的缺点如下:

- 1) 录制回复方式的稳定性和可靠性有限。
- 2) 只支持 Mozilla Firefox。
- 3) 只支持 Selense Command 语言, 虽然可以导出成其他编程语言的测试用例。
- 4) 对于复杂的页面逻辑其处理能力有限。

1.9 Selenium Grid

Selenium Grid 运用多个机器同时并列运行, 目的在于加快测试用例运行的速度, 从而减

少测试运行的总时间。对于大型测试套件和需要处理海量数据验证的测试套件，Selenium Grid 毫无疑问可以节约大量时间。Selenium Grid 的另一个优势在于可以通过节省测试时间而更快地将测试结果返还给开发人员。越来越多的软件开发团队运用敏捷开发方式，他们希望在最短时间内获得测试人员的测试结果而不是在漫漫长夜中等待着测试通过。

Selenium Grid 还可被用于在多种运行环境中进行测试，即并行测试多种浏览器。当测试套件运行起来时，Selenium Grid 会接收到每个测试用例及其对应浏览器的组合信息，并分配每个测试用例去测试其对应浏览器。

除此之外，对于相同类型和版本的浏览器来建立测试矩阵也是可行的。Selenium Grid 的使用相当灵活，以上所列举的并行测试浏览器的例子也可结合起来使用，用于测试每种类型和版本的浏览器的多个实例。

Selenium Grid 包含一个 Hub 和至少一个 Node。Hub 会接收到即将被执行的测试用例及其相关信息，即测试用例将在哪种浏览器和操作系统上运行。Hub 将记录每个“注册过”的 Node 的配置信息，并能通过这些信息自动选择可用的且符合浏览器与平台搭配要求的 Node。Node 被选中后，测试用例所调用的 Selenium 命令就会被发送至 Hub，Hub 再将这些命令发送到指定给该测试用例的 Node。随即 Node 开始启动浏览器，并执行这些 Selenium 命令对指定的 Web 程序或 Native 程序进行测试。

1.10 Selenium 与嵌入式

随着 Android、iOS、Raspberry Pi、Firefox OS、Ubuntu Phone OS、Sailfish OS 这一系列的嵌入式系统平台和操作系统的崛起，人们的手持设备和各式各样的嵌入式设备正变得越来越多样化。同一个网站，如果需要在这么多种不同平台的设备上都能够通过 Web 方式正常访问，其兼容性测试极具挑战性。而 Selenium 正在成为这场百家争鸣没有硝烟的战争中极具竞争力的测试工具和手段。

嵌入式设备中，尤其是需要在手机设备上测试 Web 应用程序，一直以来都是采用手工测试的方式。而 Selenium WebDriver 赋予了我们采用自动化方式来测试嵌入式设备上基于浏览器应用的能力，可谓一代神器。WebDriver 使得测试人员能够方便地编写自动化测试代码以确保自己的网站除了在 PC 上，还能在手机上通过浏览器正常访问。

WebDriver 为手机设备提供了专门针对触摸屏的 API，因此可以让测试程序模拟人通过手指操作触摸屏的真实动作，如单击、触划、滚动、长按等有别于普通 PC 上用鼠标操作的特殊行为。此外，还可以让测试程序对屏幕上的内容进行旋转，以及与 HTML5 的特性进行交互，如本地存储、会话存储和应用程序缓存等。

本书会基于 Android、iOS 和 Raspberry Pi 这几个平台为例来讲解 Selenium 在嵌入式平台上的使用。

1.11 Selenium 与云计算

迎着云计算普及的春风，并配合 Selenium Grid，基于 Selenium 来完成云端的测试已经变得可行。除了可以测试传统桌面系统上的 Web 应用，还可以将嵌入式设备放到云计算平台



中来完成自动化测试工作。其优势在于资源的合理调配，并且能够尽可能多地解决测试多种不同操作系统类型或者不同尺寸屏幕的兼容性问题。特别是现在热门的 Android 平台和 iOS 平台，它们都有相应的模拟器程序，使得在云中进行测试变得可行。

基于 Selenium 且比较流行的云测试平台当属 Sauce Labs，它是一个提供自动化功能测试的云测试服务公司。而其创始者兼首席技术官就是 Selenium 的创始人 Jason Huggins，可见 Sauce Labs 的技术实力非同一般。虽然 Selenium 在 Web 自动化测试方面有得天独厚的优势——兼容性测试，可以涵盖多个操作系统上的不同版本的多种浏览器，但要完成如此庞大規模的兼容性测试，就需要保留与兼容性矩阵所含格子一样多的虚拟机。这对于小团队而言，维护如此庞大規模的虚拟机并不现实。Sauce Labs 正是基于这样的需求应运而生。对于初创团队而言，他们可以在云端去完成兼容性矩阵的测试而不需要自己购买大量的硬件并自行维护成千上万台测试虚拟机。

为了应对嵌入式系统的大量普及，尤其是 Android 和 iOS，Jason Huggins 又再一次开发了针对嵌入式系统且基于 WebDriver 的自动化测试框架 Appium，并且开源托管在 GitHub 上。本书也会针对 Appium 的使用展示 WebDriver 的魅力所在。Appium 除了可以支持 Web 应用，还可以支持原生的 app 程序和 Hybrid（即 Web 和原生 app 的混搭模式）app 的测试。这也是 Sauce Labs 针对嵌入式设备所提供的云端测试解决方案。

1.12 小结

本章从自动化测试的特点、Web 自动化的独有特性娓娓道来，引出了主角 Selenium。从 Selenium 的前身介绍到 Selenium 的现状，让大家对 Selenium 有一个初步的认识。虽然其发展历史算不上惊心动魄，但也经历了跌宕起伏的过程。作为测试领域的一朵奇葩，Selenium 已然功成名就，并将继续引领时代潮流。

第 2 章

牛刀小试之 Selenium IDE

2.1 简介

Selenium IDE 最初是由 Shinya Kasatani 基于 Firefox 浏览器开发出来的一个插件。它可以调用 JavaScript 脚本并与浏览器的 DOM 对象进行交互，为脚本开发、修改测试用例提供了方便灵活的接口。

录制功能是 Selenium IDE 的一个重要特性。它允许开发或测试人员录制需要进行测试的业务流程，保存用户在测试过程中的操作，使测试用例的回放能够完全模拟用户的测试过程。此外，它还提供了校验和验证接口，方便测试人员校验测试的期望值与实际值是否相符。

2.2 安装 Selenium IDE

1. 下载 Mozilla Firefox 浏览器

在开始学习本章之前应确保机器已经正确安装了 Mozilla Firefox 浏览器。如果没有，可到以下网址下载最新版本的 Firefox 浏览器：

<https://www.mozilla.org/firefox>

2. 下载 Selenium IDE

可到 Selenium 的官方下载页面（见图 2.1）下载 Firefox 浏览器的 Selenium IDE 插件。网址如下所示：

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

插件名字类似于以 xpi 为扩展名的文件，如 selenium-ide.xpi。

3. 安装 Selenium IDE

如果是使用 Firefox 浏览器下载该文件，下载完成后 Firefox 浏览器会自动安装该插件。如果使用其他浏览器下载该文件，那么将 xpi 文件直接拖入某个打开的 Firefox 浏览器也可以进行安装。在 Firefox 浏览器中安装 Selenium IDE 插件会遇到图 2.2 所示的提示，确认安装即可。

4. Selenium IDE 安装成功

在 Firefox 浏览器中成功安装 Selenium IDE 之后，会在 Tools 菜单中出现 Selenium IDE 选

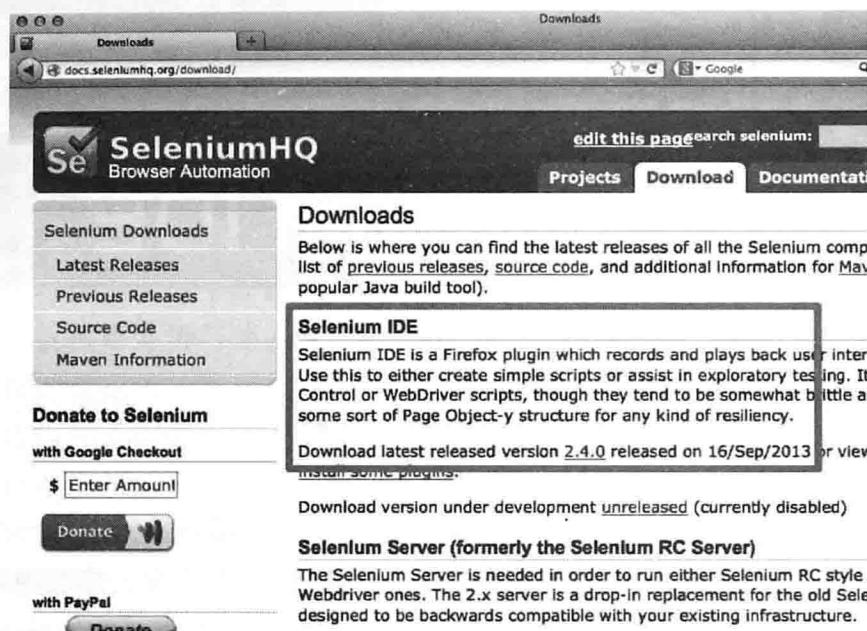


图 2.1 下载 Selenium IDE

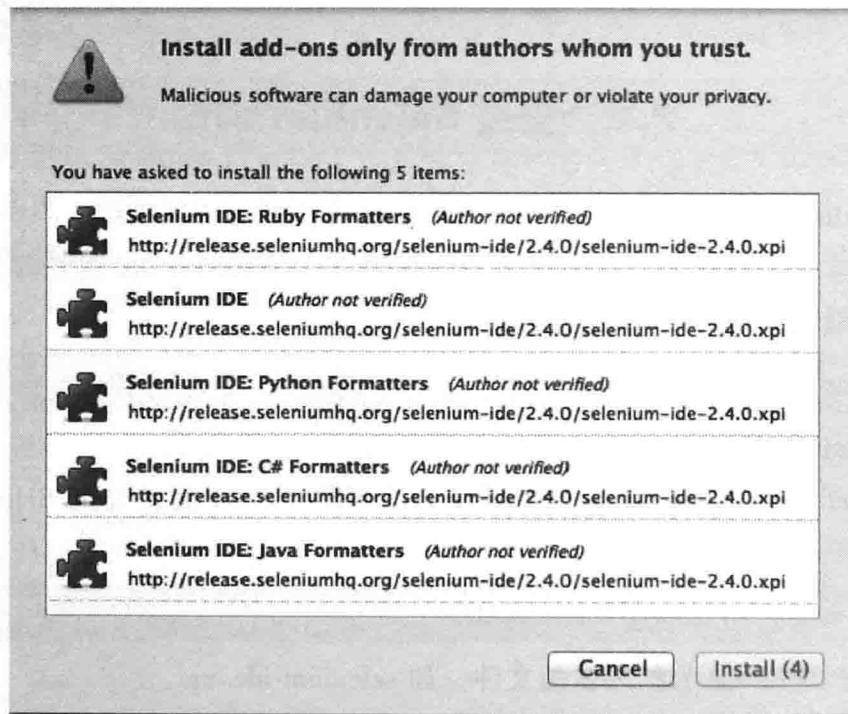


图 2.2 安装 Selenium IDE

项，如图 2.3 所示。单击此选项出现 Selenium IDE 主界面，如图 2.4 所示，则说明 Selenium IDE 安装成功。

同样，也可以通过 Firefox 浏览器的工具栏上的 Selenium IDE 快捷按钮来启动它，如图 2.5 所示。

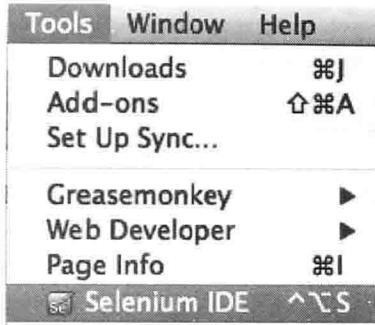


图 2.3 Selenium IDE 菜单

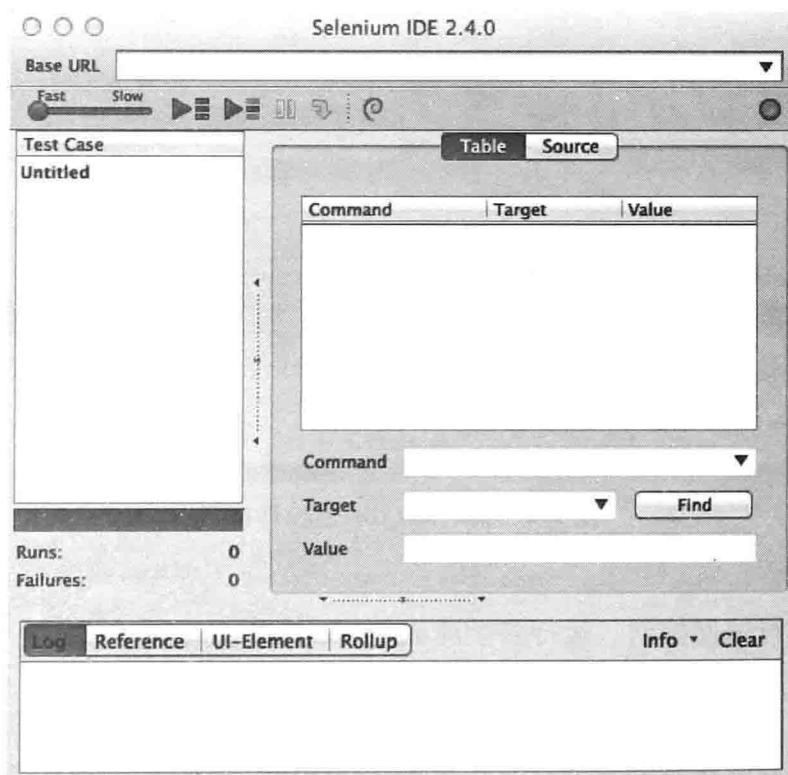


图 2.4 Selenium IDE 主界面



图 2.5 Selenium IDE 快捷按钮

2.3 Selenium IDE 界面一览

如图 2.6 所示，Selenium IDE 的界面可以算得上是小清新类型。接下来逐个认识一下界面上的控件。

1. Base URL

Base URL 是被测试系统的初始 URL 地址。

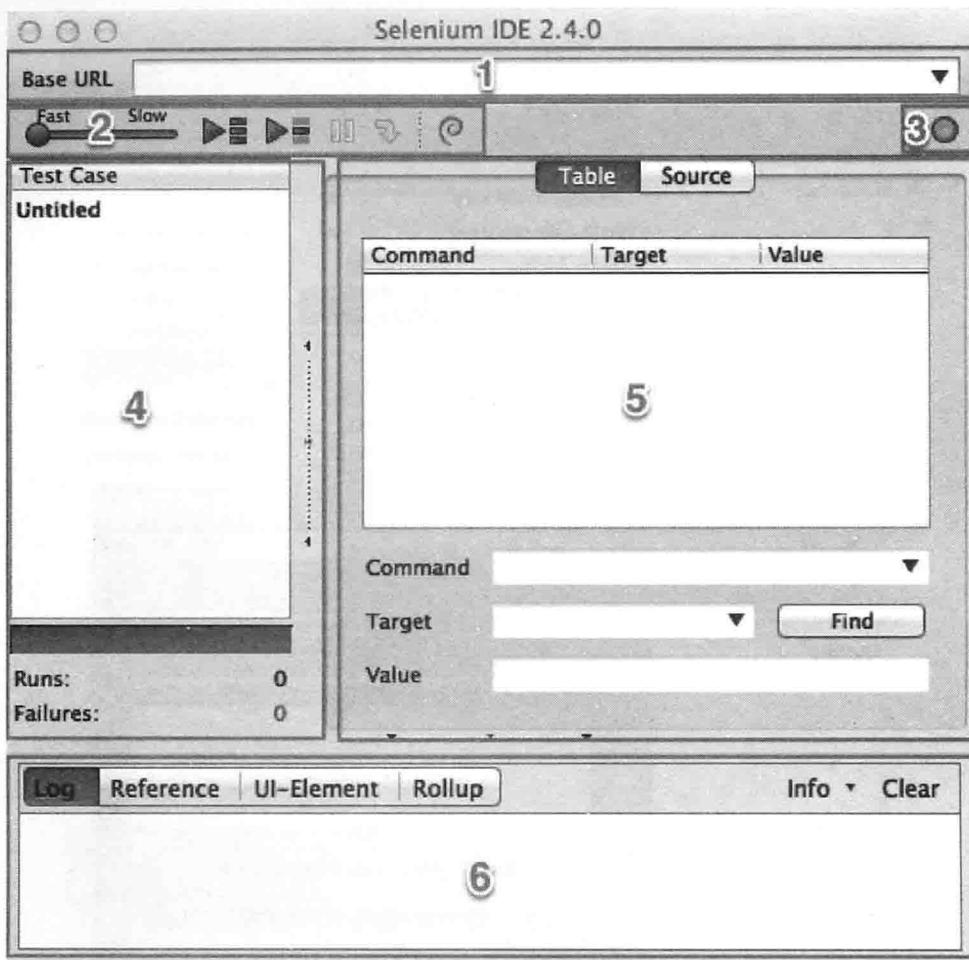


图 2.6 Selenium IDE 各控件布局

2. 工具栏

工具栏包括运行速度控制、运行多个测试用例、运行单个测试用例、暂停脚本运行、单步执行等按键。

3. 录制键

录制键用于录制需要进行测试的业务流程，后续用于回放。

4. 测试用例

列出所有的测试用例集合。

5. 测试步骤

列出单个测试用例的所有执行步骤，每个步骤由三个字段组成，包括 Command、Target 和 Value。

- 1) **Command**: Command 下拉框会列出所有的命令，可以通过自动补全的方式或者下拉框来选择命令。
- 2) **Target**: Target 文本框用于输入定位网页元素的表达式。
- 3) **Value**: Value 文本框用于输入数值。例如，需要在某个文本框中输入数值，那么在这里输入即可。
- 4) **Find**: 当在 Target 文本框中输入某个网页元素的定位语句后，可以通过单击 Find 按钮在网页中查找并高亮该元素。

6. 信息栏

这一部分包括 Log、Reference、UI-Element 和 Rollup。

- 1) Log: 显示测试运行过程中相关的日志信息。
- 2) Reference: 显示 Command 的相关文档信息。
- 3) UI-Element: 显示实际的 UI 元素的映射信息。
- 4) Rollup: 显示 Rollup 的规则信息。

2.4 创建测试用例

接下来尝试通过 Selenium IDE 来创建第一个测试用例。以百度的主页为例，步骤如下：

- 1) 用 Firefox 浏览器打开百度主页。地址如下：

```
http://www.baidu.com
```

- 2) 通过 Firefox 浏览器的 Tools 菜单打开 Selenium IDE。
- 3) 单击 IDE 上区域 3 中的录制键，开始录制测试用例。
- 4) 在百度主页的搜索框中输入 Selenium，并单击“百度一下”搜索按钮。
- 5) 验证文本 selenium 是否在搜索结果列表中。在网页上单击鼠标右键并选择 assertText，如图 2.7 所示。
- 6) 再次单击 IDE 上区域 3 中的录制键，停止录制测试用例。

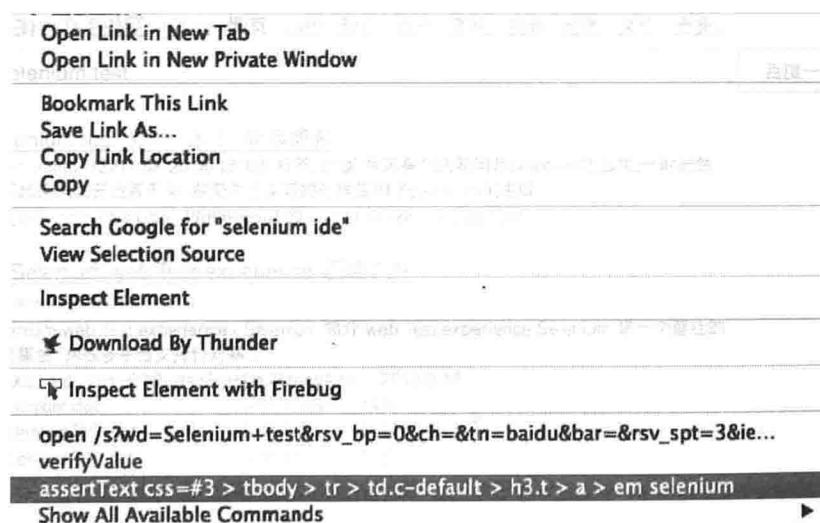


图 2.7 添加 assertText

这样就完成了第一个测试用例的录制过程。接下来可以通过单击工具栏中的“运行单个测试用例”来回放已经录制好的测试用例。

IDE 执行回放后的界面如图 2.8 所示。如果没有错误发生，界面会显示如下内容：

- 1) 面板 4 中的 Runs 数值为 1 且 Failures 为 0。
- 2) 面板 5 中测试用例的执行步骤都将呈现绿色状态，并且 assert 语句为高亮深绿色。
- 3) 面板 6 中信息栏的 Log 显示执行日志信息，而 Reference 显示单个 Command 的参考信息。

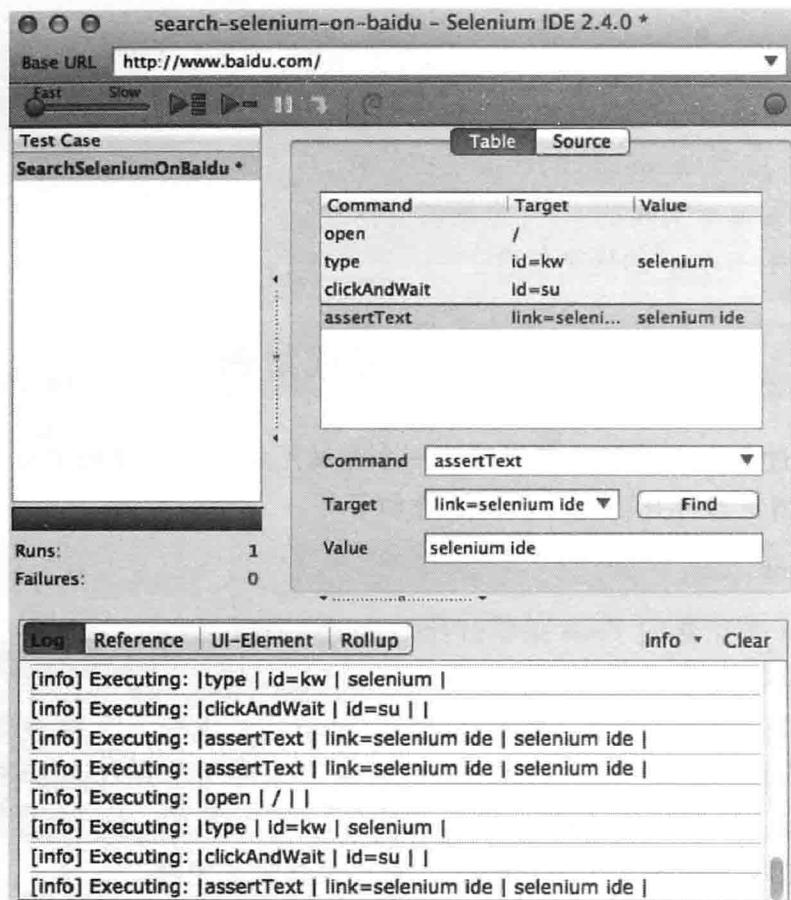


图 2.8 通过 Selenium IDE 在百度中搜索字符串

如果需要添加新的 Command 到当前的测试用例，则可以通过如下步骤来达到目标：

- 1) 选中需要在后面添加 Command 的那个测试步骤。
- 2) 单击录制键打开录制功能，操作网页完成动作。
- 3) 单击录制键完成当前录制，则新动作将作为新的步骤添加到 Command 执行列表中。

2.5 存储页面信息

通过 Selenium IDE，可以临时存储页面上的数据并在后续使用该数据。常用的数据存储方法包括 store、storeValue 和 storeText。

下面以 storeText 为例并结合 2.4 节中的测试用例来进行讲解。步骤如下：

- 1) 打开 2.4 节中已经录制好的测试用例 SearchSeleniumOnBaidu。
- 2) 选择测试用例步骤中的最后一行。
- 3) 单击鼠标右键弹出菜单，选择 Insert New Command。
- 4) 添加 storeText 命令并存储相应的文本信息，如图 2.9 所示。
- 5) 重复前三个步骤至 Insert New Command 并添加 echo 命令来打印该存储变量 storedText 的文本信息。

Command	Target	Value
open	/	
type	id=kw	selenium
clickAndWait	id=su	
storeText	link=seleni...	storedText
echo	\${storedText}	
assertText	link=seleni...	selenium ide

图 2.9 添加 storeText 命令并存储相应的文本信息

测试用例的步骤经过调整后如图 2.10 所示。运行日志中出现了 echo 命令的打印信息，正是 storedText 的文本信息，为 selenium ide。

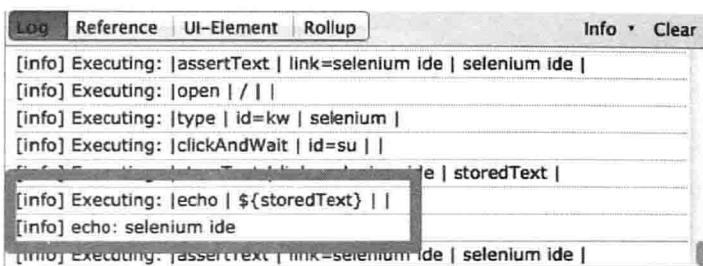


图 2.10 命令 echo 在 Log 中的打印信息

2.6 与 AJAX 页面进行交互

许多网页应用中大量使用 AJAX（Asynchronous JavaScript and XML，异步 JavaScript 和 XML）技术。AJAX 在浏览器与 Web 服务器之间使用异步数据传输 HTTP 请求，这样就可以使网页从服务器请求少量的信息，而不是整个页面。

使用 Selenium IDE 测试 AJAX 页面时会面临的一个问题也正是由于 AJAX 的异步优势所造成的。由于请求和从服务器返回的数据是异步进行的，所以如果没有合适的等待机制，Selenium IDE 很容易因为没有得到需要的数据结果而无法正常执行步骤。下面以 jQuery UI 中的 autocomplete 这个 demo 为例来讲解如何使用 Selenium IDE 与采用了 AJAX 技术的页面进行交互，并确保测试用例的正常执行。待测试页面地址为

<http://jqueryui.com/autocomplete/>

这个 demo 的功能展示了在 Tags 输入框中输入英文首字母，则会自动弹出备选编程语言列表，然后用户可以通过鼠标或者键盘选择列表中的字段后就自动填充到 Tags 输入框中，如图 2.11 所示。

Autocomplete

Enables users to quickly find and select from a pre-populated list of values as they type and filter.

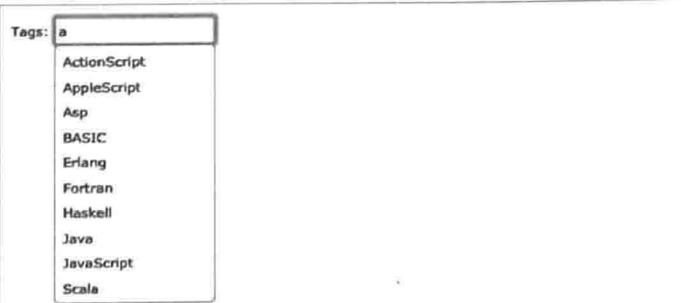


图 2.11 Autocomplete 待测试页面

使用 Selenium IDE 完成该测试用例的步骤如下：

- 1) 启动 Selenium IDE 并确保录制键被按下。



- 2) 打开待测试页面。
 3) 录制的测试用例步骤如图 2.12 所示。其中 waitForText 就是为了解决 AJAX 的异步问题而采用的。

注意，在最新版本的 Selenium IDE 中，有些老旧的 Command 不再支持。如 waitForTextPresent，最新的命令为 waitForText，正如上述测试用例的步骤中所使用的一样。通过查看该命令的 Reference 就可以看到相关信息，不再支持的命令会标注出来，如图 2.13 中所示的 This command is deprecated. Use the waitForText command with an element locator instead。

Command	Target	Value
open	/autocomplete/	
sendKeys	id=tags	a
waitForText	link=ActionScript	ActionScript
mouseOver	link=ActionScript	ActionScript
click	id=ui-id-2	ActionScript
verifyValue	id=tags	ActionScript

图 2.12 与 AJAX 页面进行交互的脚本

The screenshot shows the Selenium IDE interface with the 'Reference' tab selected. The command 'waitForTextPresent(pattern)' is listed. A note states: 'This command is deprecated. Use the waitForText command with an element locator instead.' It also mentions it's generated from 'isTextPresent(pattern)'. Below, 'Arguments:' lists 'pattern - a pattern to match with the text of the page'. Under 'Returns:', it says 'true if the pattern matches the text, false otherwise' and 'Verifies that the specified text pattern appears somewhere on the rendered page shown to the user.'

图 2.13 waitForTextPresent 的 Reference 信息

2.7 处理多窗口

在 Selenium IDE 中可以处理多个浏览器窗口。虽然 Selenium IDE 并不像本书后续章节要介绍的 WebDriver 那么强大，但通过在测试用例的步骤中使用各种命令的组合和临时变量，可以让 Selenium IDE 变得更为厉害，甚至可以操作多个浏览器窗口的切换。下例就展示了如何通过 Selenium IDE 来完成一些简单的多浏览器窗口切换的工作。当然，测试用例中的步骤并不是一次录制而成，而是通过调试修改而成。

首先需要确保 Firefox 浏览器允许 pop-up 窗口的弹出，否则本示例无法正常进行演示。如果出现如图 2.14 所示的提示信息，则需要进行合理的设置才能保证本例的后续演示。

由于本示例以百度主页为第一窗口，然后通过 pop-up 窗口打开第二个窗口，因此需要确保能在百度主页的地址中允许 pop-up 窗口的弹出。设置信息如图 2.15 所示。

接下来就可以利用 Selenium IDE 来操作多个浏览器窗口了。具体的测试用例步骤如图 2.16 所示。

- 1) 在 Selenium IDE 中打开百度主页。storeTitle 命令用于将百度主页的标题保存到变量 i



图 2.14 Firefox 阻止网站弹出 pop-up 窗口的提示信息

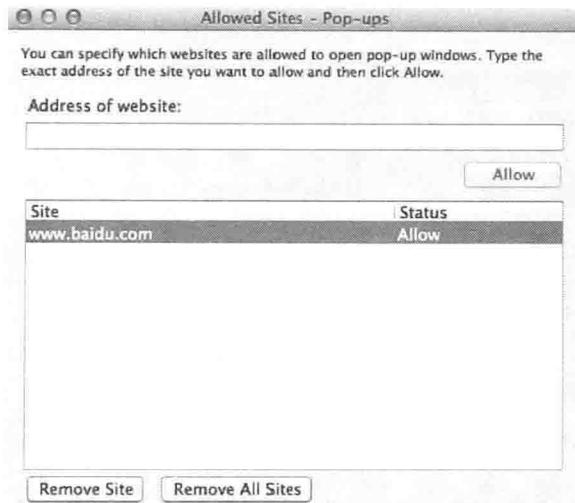


图 2.15 允许网站弹出 pop-up 窗口的设置

Command	Target	Value
open	http://www.baidu.com	
storeTitle	i	
openWindow	https://www.google.com	
pause	5000	
selectWindow	Google	
clickAndWait	link=Images	
storeTitle	j	
selectWindow	\$(i)	
clickAndWait	link=图片	
storeTitle	i	
selectWindow	\$(j)	
close	i	
selectWindow	\$(i)	
close		

图 2.16 Selenium IDE 操作多窗口的测试脚本

中并在后续步骤中使用它，同时暂且定义这个加载了百度主页的活动窗口为父窗口。openWindow 命令将会在新 pop-up 的窗口中打开 Google 主页，并通过 pause 命令将 Selenium IDE 的执行暂停 5 秒，让新 pop-up 的窗口有足够的时间加载 Google 主页。

2) 操作新 pop-up 的窗口，但此时还不能直接对新窗口进行操作。首先需要通过 selectWindow 命令将新窗口选中，定义这个加载了 Google 主页的活动窗口为子窗口。

3) 单击 Google 主页上的 Images 链接。如果此时需要在原来的父窗口——百度主页上进行一些操作，那么首先需要将当前活动页面的标题用变量 j 保存起来，这样方便在后续可以再次找到这个子窗口并在其中进行操作。

4) 通过 selectWindow \$(i) 选中父窗口，也就是之前加载了百度主页的窗口，继而单击“图片”链接。

5) 选中子窗口 Google Images 并关闭该窗口，再选中父窗口“百度图片”并关闭该父窗口。

请确保在多个浏览器窗口之间进行切换时，先保存当前活动窗口的标题以备后续能再次选中并切换回该窗口。

2.8 Rollup 的简介

Rollup 命令用于将一系列 Selenium IDE 中能用的命令打包在一起。其在 Selenium IDE 中可以充当“奇兵”，因为通过它可以帮测试用例进行“减肥”。接下来以一个实例来阐述它的“特异功能”，以网易 126 邮箱登录的页面为例，如图 2.17 所示。为讲解此例，特申请 126 邮箱的测试账号供演示。

邮箱地址：http://www.126.com

用户名：seleniumiderollup

密码：rollup123

场景 1：



- 1) 登录邮箱。
- 2) 单击“收件箱”文件夹。
- 3) 退出邮箱。

场景 2：

- 1) 登录邮箱。
- 2) 单击“已发送”文件夹。
- 3) 退出邮箱。

可以看到，用户有两次登录邮箱的操作。在不使用 Rollup 功能的情况下，使用 Selenium IDE 完成上述用户场景的测试用例步骤如图 2.18 所示。其中登录的步骤显得有些冗余，一共占用了 8 条命令。

在讲解 Rollup 的功能之前，先预习一下在 Selenium IDE 中如何使用用户自定义拓展文件。单击 Selenium IDE 的 Options 菜单，打开 General 界面。用户自定义拓展文件的添加地方如图 2.19 所示。



图 2.17 126 邮箱的“收件箱”文件夹和“已发送”文件夹

Command	Target	Value
open	/	
type	id=idInput	seleniumiderollup
type	id=pwdInput	rollup123
click	id=loginBtn	
waitForElementPresent	css=span.nui-tree-item-text	
click	css=span.nui-tree-item-text	
clickAndWait	link=退出	
open	/	
type	id=idInput	seleniumiderollup
type	id=pwdInput	rollup123
click	id=loginBtn	
waitForElementPresent	css=#_mail_component_12_12 > span.nui-tree-item-text	
click	css=#_mail_component_12_12 > span.nui-tree-item-text	
clickAndWait	link=退出	

图 2.18 Selenium IDE 不使用 Rollup 功能的测试脚本

在 Options 页面上，有几个选项需要注意：

1) Default timeout：默认情况下，timeout 的数值为 30000ms。这个数值会影响 Selenium IDE 中脚本的执行时间。可以自行根据实际需要调整该数值。如果某个命令的执行时间超时，那么在 Log 框中会得到如下错误信息：[error] Timed out after 30000ms。

2) Selenium Core extensions：用户可以根据需要来拓展 Selenium IDE 功能，其方法就是引入用户自定义的拓展文件，即 user-extension.js，而且必须是以 js 为扩展名的文件。单击 Browser 按钮并选中需要添加的用户拓展文件即可。

注意：

在添加用户自定义拓展文件之后，需要重新启动 Selenium IDE，否则该改动无法生效。

3) Selenium IDE extensions：用于添加数据文件，同样只能解析以 js 为扩展名的文件，并且也需要重启 Selenium IDE 来使其生效。

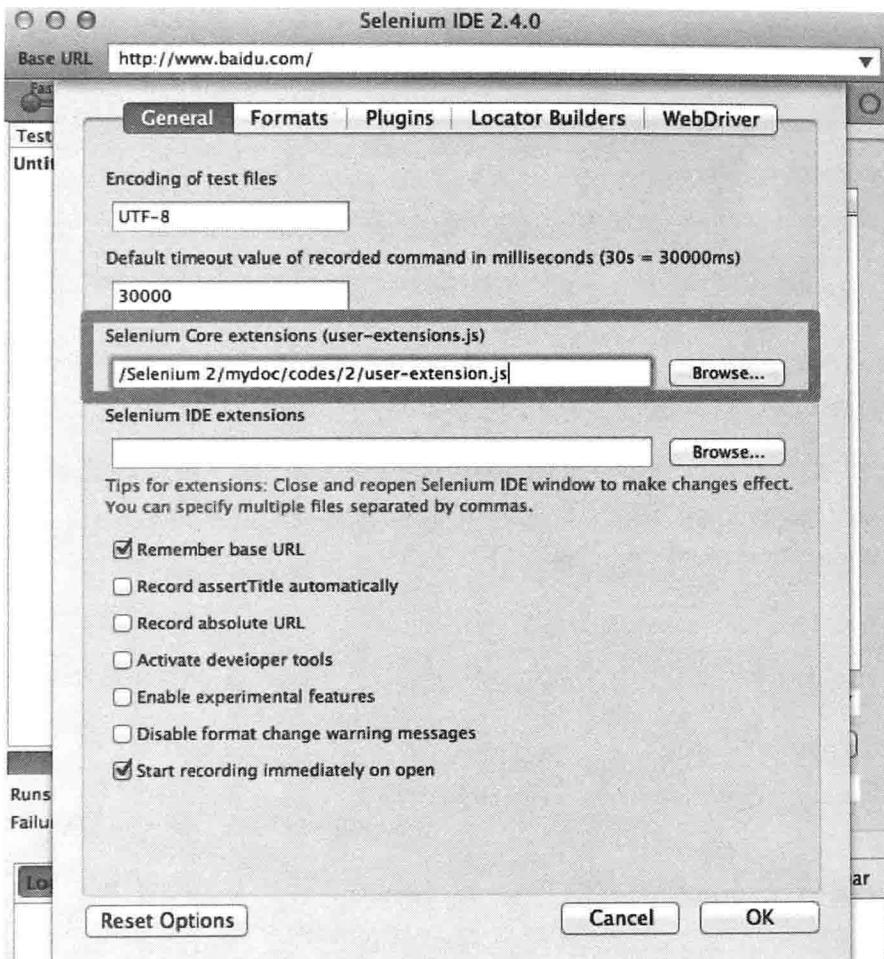


图 2.19 设置 user-extension.js 的方式

- 4) Remember base URL: 如果勾选此项, Base URL 会记住最近一次使用的链接地址。
- 5) Record assertTitle automatically: 如果勾选此项, 在每浏览一个网页时, Selenium IDE 会自动添加“assertTitle”命令到测试脚本的步骤中。
- 6) Record absolute URL: 如果勾选此项, Selenium IDE 会记录网页的绝对路径, 如以 http: 或者以 https: 开头的链接地址。
- 7) Start recording immediately on open: 如果勾选此项, Selenium IDE 就会在每次打开时自动开启录制功能。

针对本示例的用户自定义拓展文件的 user-extension.js 源码如下:

```
var manager = new RollupManager();

manager.addRollupRule( {
  name:'logincommands',
  description:'log in to 126.com',
  args:[],
  commandMatchers:[],
  getExpandedCommands:function(args) {
```



```
var commands = [ ];

commands.push( {
    command:'open',
    target:'http://www.126.com' //126 邮箱地址的 URL
} );

commands.push( {
    command:'type',
    target:'id=idInput' //用户名输入框的 id
    ,
    value:'seleniumiderollup'
} );

commands.push( {
    command:'type',
    target:'id=pwdInput' //密码输入框的 id
    ,
    value:'rollup123'
} );

commands.push( {
    command:'clickAndWait',
    target:'id=loginBtn' //登录按钮的 id
} );

return commands;
}

} );
}
```

在简单介绍了如何在 Selenium IDE 中添加用户自定义拓展文件之后，再次回到 Rollup 的主题。

在 Selenium IDE 主界面上添加 rollup 命令，如图 2.20 所示，可以看到 Target 栏提示信息为 user-extension.js 中出现的 logincommands，并且可以看到其描述信息为 log in to 126. com。

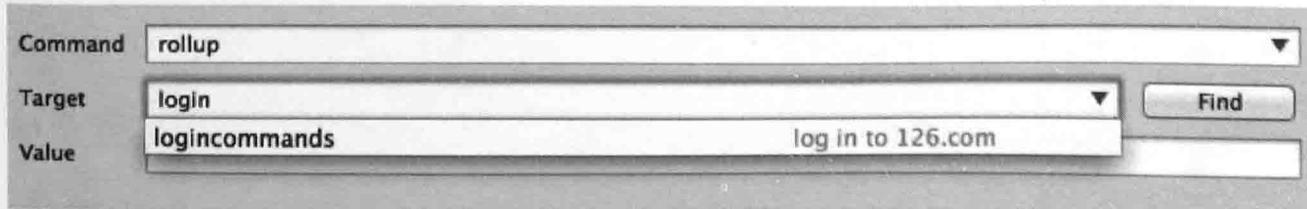


图 2.20 添加 rollup 命令

使用 Rollup 后，测试用例的完整步骤如图 2.21 所示。

Command	Target	Value
rollup	logincommands	
waitForElementPresent	css=span.nui-tree-item-text	
click	css=span.nui-tree-item-text	
clickAndWait	link=退出	
rollup	logincommands	
waitForElementPresent	css=#_mail_component_12_12 > span.nui-tree-item-text	
click	css=#_mail_component_12_12 > span.nui-tree-item-text	
clickAndWait	link=退出	

图 2.21 Selenium IDE 使用 Rollup 功能的测试脚本

从图 2.21 中可以看到最显著的差异在于使用了 rollup 命令以及 Target 为 logincommands，即 user-extension.js 中定义的 rollup 规则。该规则通过 RollupManager 添加进来。

接下来是信息栏的 Rollup，如图 2.22 所示。logincommands 的 rollup 操作实际上是将以下 4 个步骤打包起来使用：

```
打开 126 邮箱的主页: open | http://www.126.com/
输入用户名: type | id=idInput | seleniumiderollup
输入密码: type | id=pwdInput | rollup123
单击登录按钮: clickAndWait | id=loginBtn
```

这样就确保了只使用单独一条 logincommands 就可以替代原先 4 条命令需要完成的动作。即使重复登录，增加的测试用例的步骤也只是多一条 rollup 命令而已。可见 rollup 功能在封装多条相同命令时非常高效。

Log | Reference | UI-Element | **Rollup**

logincommands

log in to 126.com

preconditions:

postconditions:

current rollup expands to:

```
open | http://www.126.com/
type | id=idInput | seleniumiderollup
type | id=pwdInput | rollup123
clickAndWait | id=loginBtn
```

图 2.22 Rollup 命令的详细步骤

2.9 小结

本章从 Selenium IDE 的起源开始逐步展开，介绍了如何在 Mozilla Firefox 浏览器上安装 Selenium IDE，及其布局的安排。接下来从如何创建第一个测试用例开始逐步引导大家熟悉 Selenium IDE 各个命令的使用，包括如何存储页面信息、与 AJAX 页面进行交互，以及处理多窗口的情况等。最后以 Rollup 和在 Selenium IDE 中如何使用用户自定义拓展文件收尾。

第 3 章

Selenium 玩转页面元素

3.1 简介

在基于 UI 元素的自动化测试中，无论基于桌面的 UI 自动化测试，还是基于 Web 的 UI 自动化测试，对于 UI 元素的识别和查找均起到了决定性的作用。在基于 Web 的 UI 自动化测试领域，Selenium 为查找 Web 页面上的各个 UI 元素提供了强而有力的支持。

3.2 浏览器调试工具

工欲善其事，必先利其器。查找页面上的元素，必须先了解页面的结构、元素的属性，以及 JavaScript 函数调用等信息。现代浏览器都带有方便的 Web 开发调试工具，接下来一一进行介绍。

★3.2.1 Google Chrome

Google Chrome 浏览器自带 Web 开发调试工具，通过以下两种方式可以打开此工具：

方法 1：选择 View→Developer→Developer Tools 菜单命令，如图 3.1 所示。

方法 2：任意页面上，单击鼠标右键，选择 Inspect Element 菜单命令，如图 3.2 所示。

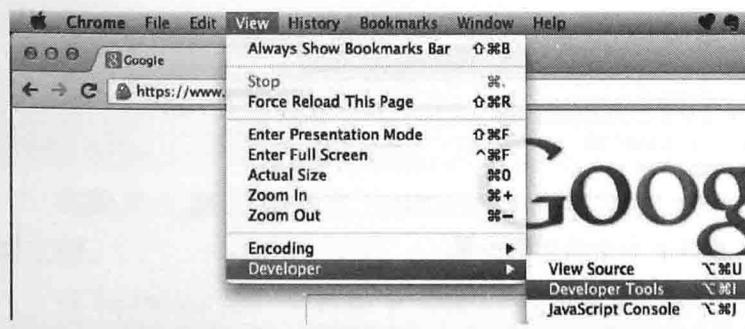


图 3.1 通过菜单打开 Developer Tools

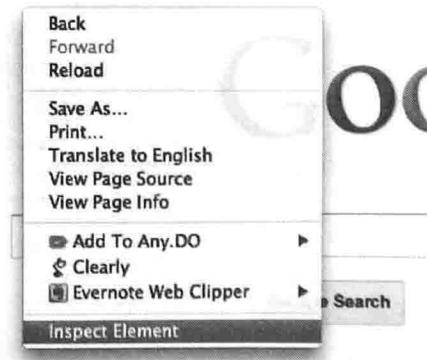


图 3.2 通过鼠标右键打开 Developer Tools

HTML 代码在 Developer Tools 中以树形结构显示，方便开发人员了解页面的各个元素以及它们的属性、DOM 结构、JavaScript 调用和 CSS 样式等信息，如图 3.3 所示。

下面会探讨如何通过 XPath 来查找元素，借助 Google Chrome 浏览器的 Web 开发调试工具就可以很方便地获取某个具体元素的 XPath。只需要将鼠标移向所需查找的元素并单击鼠



图 3.3 HTML 代码的树形结构

标右键，选择 Copy XPath 即可，如图 3.4 所示。

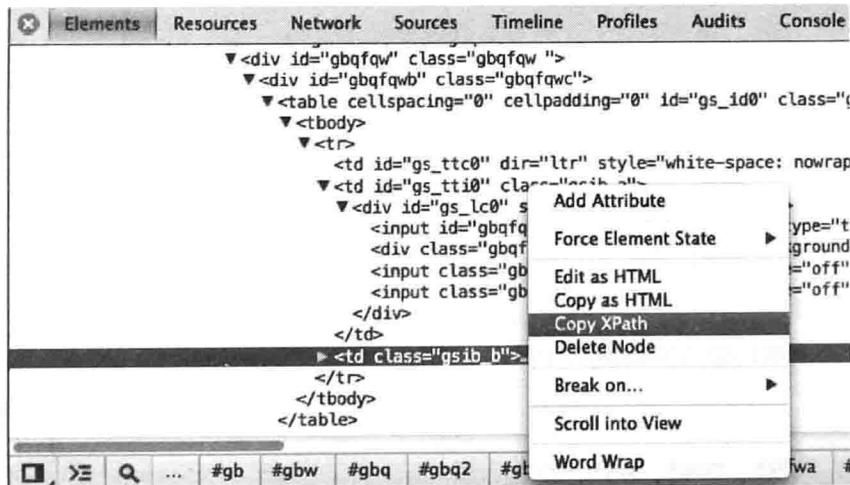


图 3.4 在 Chrome 浏览器中获取页面元素的 XPath

在 WebDriver 中定位页面元素是一件很惬意的事情。针对不同的编程语言绑定，WebDriver 都提供了两种方式来定位页面元素，分别是 findElement 和 findElements。第一种方式的结果是在正常情况下返回一个页面元素对象，一旦出现异常就会报错。第二种方式的结果是在正常情况下返回包括多个页面元素对象的列表，如果没有任何一个 DOM 元素能匹配此次查询请求，则此返回列表为空。

★3.2.2 Mozilla Firefox

Mozilla Firefox 浏览器的 Firebug 插件无疑是 Firefox 开发工具类别中的王者。获取 Firebug 可以有两种方式：一种方式是到其官方主页下载并添加到 Firefox 浏览器中；另一种方式是



在 Firefox 的附属组件网站上进行下载。地址分别为

<https://getfirebug.com/>

<https://addons.mozilla.org/zh-cn/firefox/addon/firebug/>

待 Firebug 成功安装后，可以在 Tools 菜单中找到 Firebug 的启动选项，如图 3.5 所示。



图 3.5 通过菜单启动 Firebug

通过单击 Firefox 浏览器的工具栏上 Firebug 的快捷按钮也可完成相同的启动步骤，如图 3.6 所示。



图 3.6 通过快捷按钮启动 Firebug

在 Firebug 调试界面上，有一个 Click an element in the page to inspect 按钮，如图 3.7 所示。这样方便开发者在用鼠标移动到页面上某个具体的元素时，实时确保在 HTML 代码信息框中跳转到对应页面元素的源码位置。

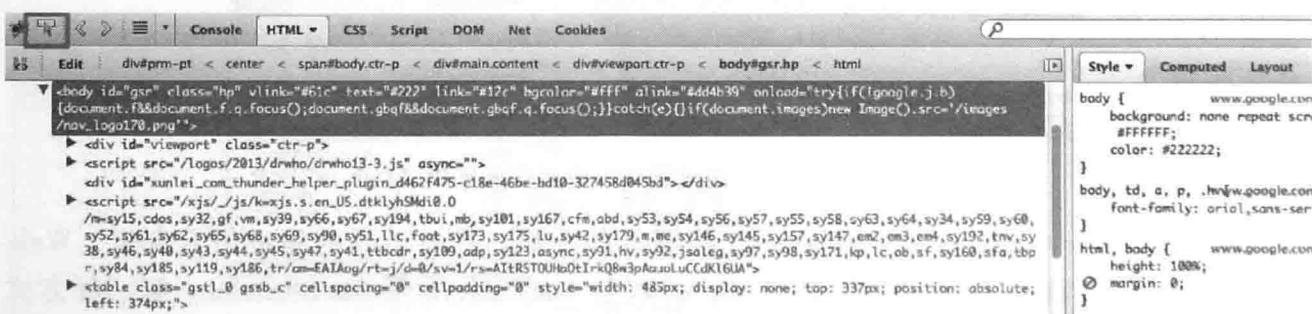


图 3.7 开启实时追踪页面元素的源代码

如果需要得到对应页面元素的 XPath 地址，对着该元素的源码单击鼠标右键，并选择 Copy XPath 即可完成相应功能，如图 3.8 所示。

★3.2.3 Internet Explorer

Windows 自己出品的 Internet Explorer 浏览器同样自带“开发人员工具”，如图 3.9 所

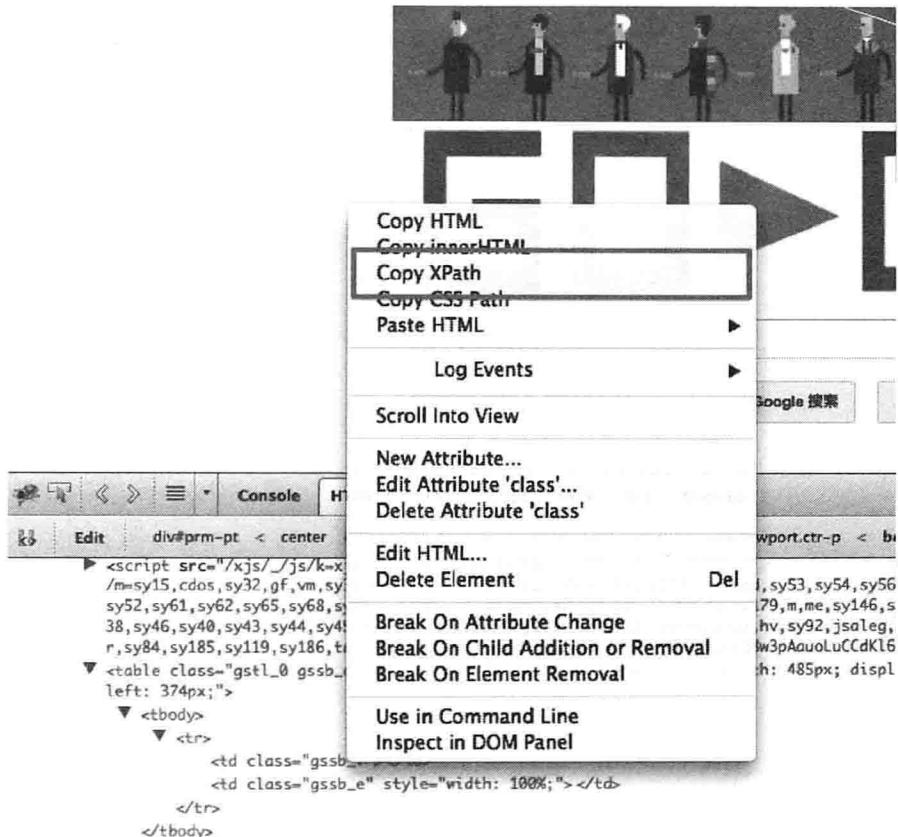


图 3.8 在 Firefox 浏览器中获取页面元素的 XPath

示。通过“工具”菜单或者快捷键 F12 均可以触发该项功能。



图 3.9 启动 Internet Explorer 浏览器的“开发人员工具”



启动 Internet Explorer 浏览器的开发人员工具后，可以通过单击“查找”→“单击选择元素”菜单选项来方便开发者在用鼠标移动到页面上某个具体的元素时，实时确保在 HTML 代码信息框中跳转到对应该页面元素的源码位置。如图 3.10 所示，开发人员工具会高亮此元素为蓝色，且 HTML 代码将以树形结构显示。

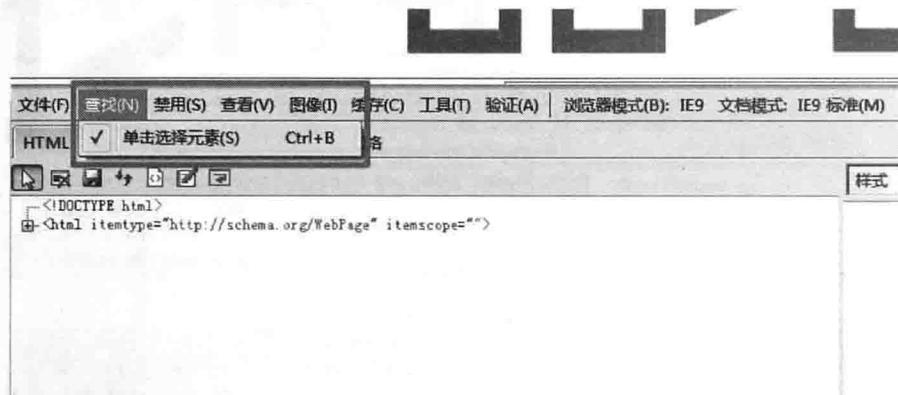


图 3.10 开启实时追踪页面元素的源代码

3.3 查找页面元素

★3.3.1 通过 ID 查找元素

通过页面元素的 ID 来查找元素是最为推荐的方式。W3C 标准推荐开发人员为每一个页面元素都提供一个独一无二的 ID 属性，因此开发人员应该避免在单个页面上的所有元素存在 ID 不唯一的情况和元素 ID 是自动生成的情况。一旦页面元素被赋予了唯一的 ID 属性，它就能够很容易地被浏览器调试工具或者测试工具识别并查找到。在浏览器解析 DOM (Document Object Model，文档对象模型) 时，页面元素的 ID 被作为首选的识别属性，因为这是最快的识别策略。

以百度主页为例，搜索框的 HTML 示例代码如下，其 ID 为 kw：

```
<span class="bg s_ipt_wr">
    <input type="text"
        name="wd"
        id="kw"
        maxlength="100"
        class="s_ipt"
        autocomplete="off">
</span>
```

“百度一下”搜索按钮元素的 HTML 示例代码如下，其 ID 为 su：

```
<span class="bg s_btn_wr">
    <input type="submit"
        value="百度一下"
```

```

    id ="su"
    class ="bg s_btn"
    onmousedown ="this.className ='bg s_btn s_btn_h'"
    onmouseout ="this.className ='bg s_btn'">
</span >
```

在 WebDriver 中通过 ID 查找元素的 Java 示例代码如下：

```

public class testBaiduById {
    public static void main( String[ ] args ) {
        WebDriver driver = new FirefoxDriver();
        driver.get("http://www.baidu.com");

        WebElement searchBox = driver.findElement( By.id("kw") );
        searchBox.sendKeys("test Baidu By Id");

        WebElement searchButton = driver.findElement( By.id("su") );
        searchButton.submit();

        driver.close();
    }
}
```

示例代码详解：

- 1) 指定 WebDriver 为 FirefoxDriver。
- 2) 打开百度主页。
- 3) 通过 ID 为 kw 来查找搜索框，代码段如下。其中，findElement() 方法通过 By.id() 在页面上查找指定 ID 的元素，并将查找结果返回给一个 WebElement 实例对象并保存下来。WebElement 对象具有多种行为，如 click、clear、submit、wait 等操作。

```
WebElement searchBox = driver.findElement ( By.id ("kw") );
```

- 4) 在搜索框中输入字符串 test Baidu By Id。
- 5) 通过 ID 为 su 来查找搜索按钮。代码段如下：

```
WebElement searchButton = driver.findElement ( By.id ("su") );
```

- 6) 触发搜索按钮的提交操作，进行搜索。

★3.3.2 通过 Name 查找元素

以豆瓣网的主页搜索框为例，其搜索框的 HTML 代码如下，其 Name 为 q：

```
<span class ="inp">
    <input type ="text"
```



```
maxlength = "60"  
size = "12"  
placeholder = "书籍、电影、音乐、小组、小站、成员"  
name = "q"  
autocomplete = "off" >  
</span >
```

Selenium WebDriver 中通过 Name 查找豆瓣主页上搜索框元素的 Java 示例代码如下：

```
public class testDoubanByName {  
    public static void main( String[ ] args ) {  
        WebDriver driver = new FirefoxDriver();  
        driver.get("http://www.douban.com/");  
  
        WebElement searchBox = driver.findElement( By.name("q") );  
        searchBox.sendKeys("test Douban By Name");  
        searchBox.submit();  
  
        driver.close();  
    }  
}
```

示例代码详解：

- 1) 使用 FirefoxDriver 来打开豆瓣主页。
- 2) 通过 Name 为 q 来调用 findElement() 方法找到豆瓣主页的搜索框元素，并保存到 WebElement 实例对象中。代码段如下：

```
WebElement searchBox = driver.findElement ( By.name ("q") );
```

- 3) 在搜索框中输入字符串 test Douban By Name 并提交进行搜索。

★3.3.3 通过 ClassName 查找元素

以淘宝网的主页搜索框为例，其搜索框的 HTML 代码如下，ClassName 为 search-combo-box-input：

```
<div class = "search-combo-box-input-wrap" >  
    <input id = "q"  
           name = "q"  
           aria-label = "请输入搜索文字"  
           accesskey = "s"  
           autofocus = "true"  
           autocomplete = "off"
```

```

aria-haspopup = "true"
aria-combobox = "list"
role = "combobox"
class = "search-combobox-input"
x-webkit-speech =
x-webkit-grammar = "builtin:translate" >
</div >

```

Selenium WebDriver 中通过 `ClassName` 查找淘宝主页上搜索框的 Java 示例代码如下：

```

public class testTaobaoByClassName {
    public static void main( String[ ] args ) {
        WebDriver driver = new FirefoxDriver();
        driver.get("http://www.taobao.com/");

        WebElement searchBox =
            driver.findElement( By.className("search-combobox-input") );
        searchBox.sendKeys("test Taobao By ClassName");
        searchBox.submit();

        driver.close();
    }
}

```

示例代码详解：

- 1) 使用 `FirefoxDriver` 来打开淘宝主页。
- 2) 通过 `ClassName` 为 `search-combobox-input` 来调用 `findElement()` 方法，找到淘宝主页的搜索框元素并保存到 `WebElement` 实例对象中。代码段如下：

```

WebElement searchBox =
    driver.findElement( By.className ("search-combobox-input") );

```

- 3) 在搜索框中输入字符串 `test Taobao By ClassName` 并提交进行搜索。

★3.3.4 通过 `TagName` 查找元素

通过 `TagName` 来查找元素的方式与前述通过 `ID` 或 `Name` 查找元素的方式略有不同。其原因是同一个页面上具有相同 `TagName` 的元素可能一个都没有，也可能有多个。以小米主页为例，如果搜索以 `script` 为 `TagName` 的元素就会返回多个结果。因此建议在使用 `TagName` 为查找元素的条件时，使用 `findElements()` 来替代 `findElement()` 函数。

以小米主页为例，得到 `TagName` 为 `script` 的元素个数的示例代码如下：



```
public class testXiaomiByTagName {  
    public static void main( String[ ] args ) {  
        WebDriver driver = new FirefoxDriver();  
        driver.get("http://www.xiaomi.com");  
  
        List<WebElement> scriptList =  
            driver.findElements( By.tagName("script") );  
        System.out.println("There are " +  
            scriptList.size() +  
            " scripts on Xiaomi's main page.");  
  
        driver.close();  
    }  
}
```

示例代码详解：

- 1) 使用 FirefoxDriver 打开小米主页。
- 2) 通过 TagName 为 script 来调用 findElements() 方法，找到小米主页上所有的 script 元素并保存到 WebElement 实例对象列表中。代码段如下：

```
List<WebElement> scriptList = driver.findElements( By.tagName("script") );
```

- 3) 打印小米主页上 TagName 为 script 的元素的数量，打印信息如下，一共有 14 个：

```
There are 14 scripts on Xiaomi's main page.
```

★3.3.5 通过 LinkText 查找元素

以 CSDN 的主页为例，页面最下面有一个“联系方式”的链接地址，其 HTML 代码如下：

```
<a href="http://www.csdn.net/company/contact.html" target="_blank">联系方式 </a>
```

如上所示，页面上的 HTML 链接元素一般由 `<a>` 标签来表示，即 anchor 的缩写。其中 `href` 表示该链接被单击后会跳转的页面地址。Selenium 可以通过 anchor 上的文本信息来查找该元素并进行操作。示例代码如下：

```
public class testCsdnByLinkText {  
    public static void main( String[ ] args ) {  
        WebDriver driver = new FirefoxDriver();  
        driver.get("http://www.csdn.com/");  
  
        WebElement contactLink = driver.findElement( By.linkText("联系方式") );
```

```

        contactLink.click();

        driver.close();
    }
}

```

示例代码详解：

1) 使用 FirefoxDriver 打开 CSDN 主页。

2) 通过 LinkText 为“联系方式”来调用 findElement() 方法，找到 CSDN 主页上的“联系方式”链接元素并保存到 WebElement 实例对象中。代码段如下：

```
WebElement contactLink = driver.findElement(By.linkText("联系方式"));
```

3) 单击此“联系方式”链接并打开新页面查看 CSDN 的联系方式。

★3.3.6 通过 PartialLinkText 查找元素

依旧以 CSDN 的主页为例，页面最下方有一个“联系方式”的链接地址，其 HTML 代码如下：

```
<a href="http://www.csdn.net/company/contact.html" target="_blank">联系方式</a>
```

Selenium 可以通过 anchor 上的部分文本信息来查找该元素并进行操作，如通过“联系”二字来查找该链接地址。示例代码如下：

```

public class testCsdnByPartialLinkText {
    public static void main( String[ ] args ) {
        WebDriver driver = new FirefoxDriver();
        driver.get("http://www.csdn.com/");

        WebElement contactLink =
            driver.findElement(By.partialLinkText("联系"));
        contactLink.click();

        driver.close();
    }
}

```

示例代码详解：

1) 使用 FirefoxDriver 打开 CSDN 主页。

2) 通过 PartialLinkText 为“联系”来调用 findElement() 方法，找到 CSDN 主页上的“联系方式”链接元素并保存到 WebElement 实例对象中。代码段如下：

```
WebElement contactLink = driver.findElement(By.partialLinkText("联系"));
```



3) 单击此“联系方式”链接并打开新页面查看 CSDN 的联系方式。

注意：

`findElement()` 方法只会返回页面上第一个满足 `PartialLinkText` 为“联系”的元素。如果希望找到页面上所有包含部分文本为“联系”的链接元素，则使用 `findElements()` 方法来替换 `findElement()` 方法。

★3.3.7 通过 CSS 选择器查找元素

以 Google 主页的搜索按钮为例，其 HTML 代码如下：

```
<input value="Google 搜索" jsaction="sf.chk" name="btnK" type="submit">
```

对应的 CSS 路径代码如下：

```
#st-lb
```

Selenium WebDriver 中通过 CSS 查找元素的 Java 示例代码如下：

```
public class testGoogleByCssSelector {  
    public static void main( String[ ] args ) {  
        WebDriver driver = new FirefoxDriver();  
        driver.get("http://www.google.com/");  
  
        WebElement searchBox = driver.findElement( By.cssSelector("#st-lb") );  
        searchBox.sendKeys("webdriver");  
        searchBox.submit();  
  
        driver.close();  
    }  
}
```

示例代码详解：

- 1) 使用 `FirefoxDriver` 打开 Google 主页。
- 2) 通过 CSS 路径查找到搜索框。代码段如下：

```
WebElement searchBox = driver.findElement ( By.cssSelector ("#st-lb") );
```

- 3) 输入 `webdriver` 字符串并进行搜索。

★3.3.8 通过 XPath 查找元素

以 Google 主页为例，可以通过 XPath 来查找到搜索框和搜索按钮，并且对于期望的结果也可以通过 XPath 来查找。

Selenium WebDriver 中通过 XPath 查找元素的完整示例代码如下：

```
package com.learningselenium findelement;

import org.testng.annotations.Test;
import org.testng.annotations.AfterClass;
import org.testng.annotations.BeforeClass;
import org.openqa.selenium.chrome.*;
import org.openqa.selenium.*;
import org.openqa.selenium.support.ui.WebDriverWait;
import static org.openqa.selenium.support.ui.ExpectedConditions.
visibilityOfElementLocated;

public class testXPath {

    WebDriver driver;

    @BeforeClass
    public void setUp() {
        System.setProperty("webdriver.chrome.driver",
            "/Selenium 2/selenium/chromedriver");
        driver = new ChromeDriver();
    }

    @AfterClass
    public void tearDown() {
        driver.close();
        driver.quit();
    }

    @Test
    public void testGoogle() throws InterruptedException {
        driver.get("http://www.google.com/");
        WebElement searchBox = driver.findElement(
            By.xpath("// * [@id = \"lst-ib\"]"));
        searchBox.sendKeys("selenium");
    }
}
```



```
WebElement searchButton = driver.findElement(
    By.xpath("// *[@id = \"tsf\"] /div[2] /div[3] /center /input[1]"));
searchButton.click();

Wait<WebDriver> wait = new WebDriverWait(driver, 30);
wait.until(ExpectedConditions.visibilityOfElementLocated(
    By.xpath("// *[@id = \"rso\"] /li[1] /div /h3 /a /em")));
}

}
```

示例代码详解：

- 1) 设置 WebDriver 以 Chrome 浏览器为驱动模型并设定 Driver 的文件路径。
- 2) 打开 Google 主页。
- 3) 通过 XPath 查找到搜索框并在其中输入 selenium。示例代码段如下：

```
WebElement searchBox = driver.findElement(
    By.xpath("// *[@id = \"lst-ib\"]"));
searchBox.sendKeys("selenium");
```

- 4) 通过 XPath 查找搜索按钮并单击它。示例代码段如下：

```
WebElement searchButton = driver.findElement(
    By.xpath("// *[@id = \"tsf\"] /div[2] /div[3] /center /input[1]"));
searchButton.click();
```

- 5) 等待指定 XPath 的页面元素的出现，这里使用了 Selenium 的 Wait 对象和期待条件 visibilityOfElementLocated。示例代码段如下：

```
Wait<WebDriver> wait = new WebDriverWait(driver, 30);
wait.until(ExpectedConditions.visibilityOfElementLocated(
    By.xpath("// *[@id = \"rso\"] /li[1] /div /h3 /a /em")));
```

★3.3.9 通过 jQuery 查找元素

jQuery 允许开发人员通过简单的步骤快速识别页面上的元素。这里将展示在 Selenium 中如何使用 jQuery 来简化工作。在使用 jQuery 简化 Selenium 查找元素操作之前，需要首先确认页面是否已经加载了 jQuery 库。在这里将分情况进行阐述，一种情况是页面本身已经加载 jQuery 库，另外一种情况是页面本身并没有加载 jQuery 库。

1. 处理已经加载 jQuery 库的页面

以 jQuery Foundation 官方网站为例，其已经自动加载了 jQuery 库，因此可以在测试代码中直接使用 jQuery 的语法。下面以从页面上查找到导航栏上的菜单元素并确认其总数，然后筛选出菜单的第 3 个元素并验证其文本信息为例进行说明，如图 3.11 所示。

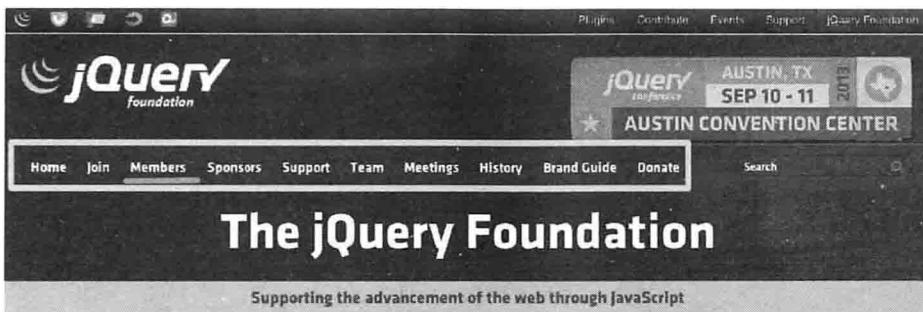


图 3.11 jQuery 页面的导航栏

Java 代码示例片段如下：

```
WebDriver driver = new ChromeDriver();

JavascriptExecutor jse = (JavascriptExecutor)driver;

driver.get("http://www.jquery.org/");

List<WebElement> elements =
    (List<WebElement>) jse.executeScript
    ("return jQuery.find('.menu-item')");

assertEquals(10,elements.size());
assertEquals("Members",elements.get(2).getText());

driver.close();
```

上述代码使用了 jQuery 的 find() 方法来查找元素，在 Selenium 的 Java 代码中，需要调用 JavascriptExecutor 类来执行 jQuery 的 find() 方法。

2. 处理未加载 jQuery 库的页面

虽然 jQuery 是非常流行的库，但并不是所有的网站都加载并使用到该库。对于未加载 jQuery 库的页面，如果希望利用 jQuery 的优势来查找页面元素，则可以通过其他方式来达到目的。也就是在加载页面时手工注入 jQuery 库。注入代码如下：

```
private void injectjQueryIfNeeded() {
    if (!jQueryLoaded()) {
        injectjQuery();
    }
}
```



```
public Boolean jQueryLoaded() {  
    Boolean loaded = true;  
    try {  
        loaded = (Boolean) jse.executeScript("return jQuery() != null");  
    } catch (WebDriverException e) {  
        loaded = false;  
    }  
    return loaded;  
}  
  
public void injectjQuery() {  
    //load jQuery dynamically in the head of web page  
    jse.executeScript(  
        "var headID = document.getElementsByTagName('head')[0];" +  
        "var newScript = document.createElement('script');" +  
        "newScript.type ='text/javascript';" +  
        "newScript.src ='http://ajax.googleapis.com/ajax/libs/" +  
        "/jquery/1.10.1/jquery.min.js';" +  
        "headID.appendChild(newScript)");  
}
```

injectjQueryIfNeeded（）方法会调用 jQueryLoaded（）来检测当前页面是否已经加载 jQuery 库。如果页面尚未加载 jQuery 库，injectjQueryIfNeeded（）会调用 injectjQuery 方法，通过添加 `< script >` 元素将 jQuery 库注入到页面的 head 部分。`< script >` 元素通过引用 Google CDN（Content Delivery Network，内容分发网络）来加载 jQuery 库，该 CDN 信息可以从 jQuery 的官方网站获取，如图 3.12 所示。

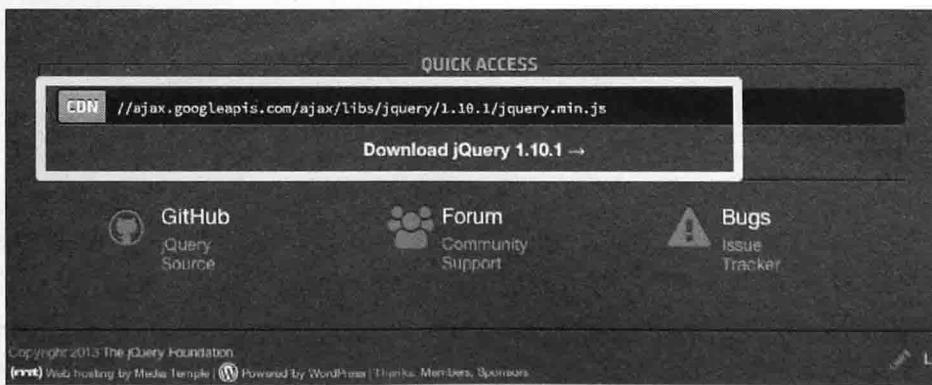


图 3.12 jQuery 库的 CDN 信息

以 Google 的主页为例，默认是没有加载 jQuery 库的。首先尝试在 Google Chrome Web 调试工具 Console 中能否正常加载 jQuery 库。在 Console 中输入加载 jQuery 的 JavaScript 脚本，如果出现如图 3.13 所示的提示，则说明加载成功。

```
> var headID = document.getElementsByTagName("head")[0];
> var newScript = document.createElement('script');
> newScript.type = 'text/javascript';
> newScript.src = 'http://ajax.googleapis.com/ajax/libs/jquery/1.10.1/jquery.min.js';
> headID.appendChild(newScript);
> <script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/1.10.1/jquery.min.js"></script>
>
```

图 3.13 通过 Google Console 查看 jQuery 是否加载成功

接下来，尝试定位 Google 主页面左上角的 Images 菜单，如图 3.14 所示。其中，1 为 Images 菜单的位置，2 为 Images 菜单在 DOM 结构中的位置，3 为在 jQuery 中用于查找的路径。

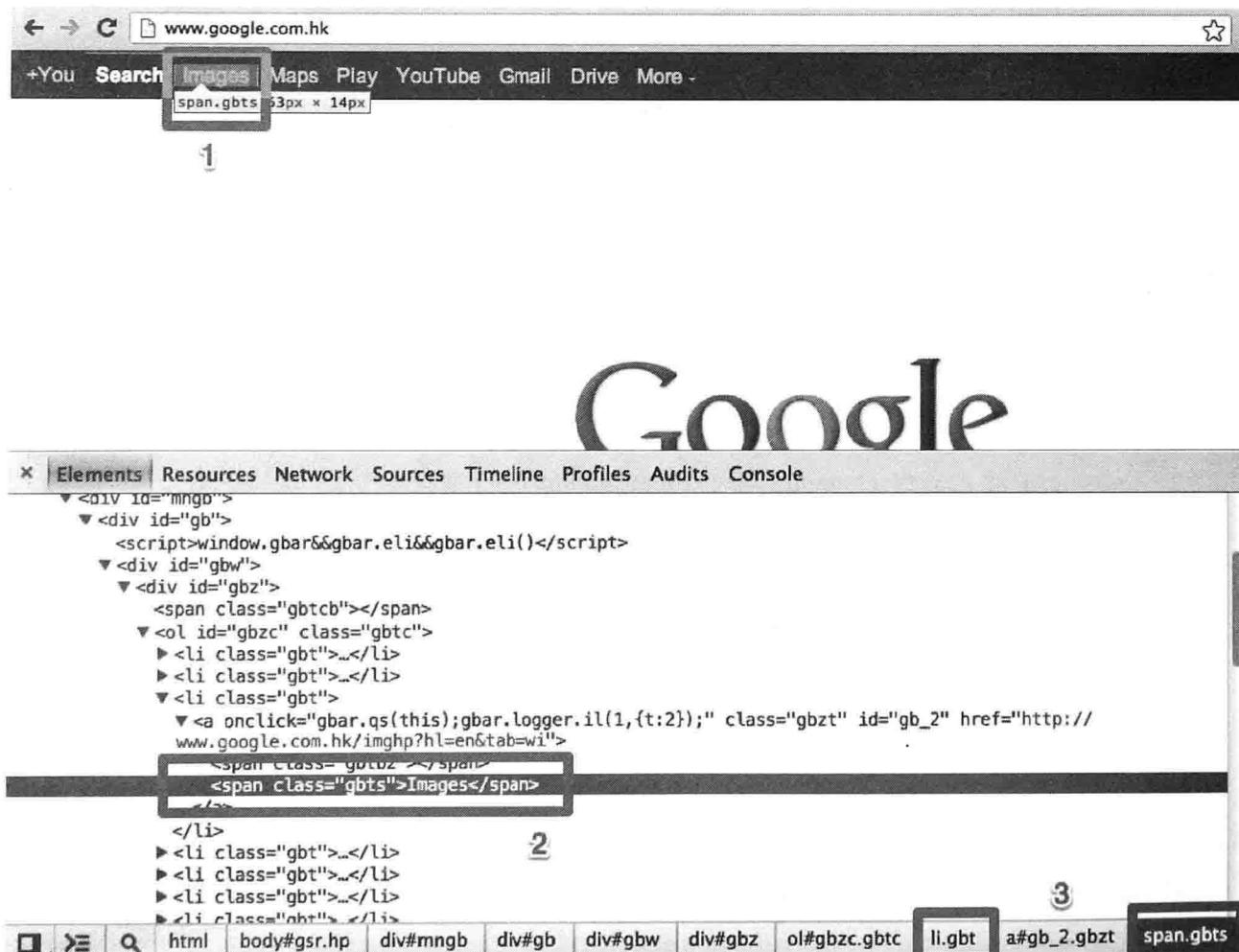


图 3.14 定位 Google 主页面左上角的 Images 菜单

接下来在 Console 中输入 jQuery 查找语句，如图 3.15 所示。其中，1 为 jQuery 查找语句，2 为 Images 菜单在查找结果中的位置，由于排在第 3 个位置，其索引下标为 2。

```
> elements = jQuery.find('li.gbt span.gbts')
[<span class="gbts">+You</span>, <span class="gbts">Search</span>, <span class="gbts">Images</span>, <span class="gbts">Maps</span>,
 <span class="gbts">Play</span>, <span class="gbts">YouTube</span>, <span class="gbts">Gmail</span>, <span class="gbts">Drive</span>,
 > <span id="gbzms" class="gbts gbtsa">...</span>, > <span id="gbgs4" class="gbts">...</span>, <span class="gbts"></span>]
```

图 3.15 在 Console 中输入 jQuery 查找语句



然后通过 elements [2] 来进行验证，如图 3.16 所示。

```
> elements[2]
<span class="gbts">Images</span>
>
```

图 3.16 通过 elements [2] 来验证 Images

查看 Images 菜单的属性，其中有一项.textContent 的赋值为 Images，如图 3.17 中的 1 所示。

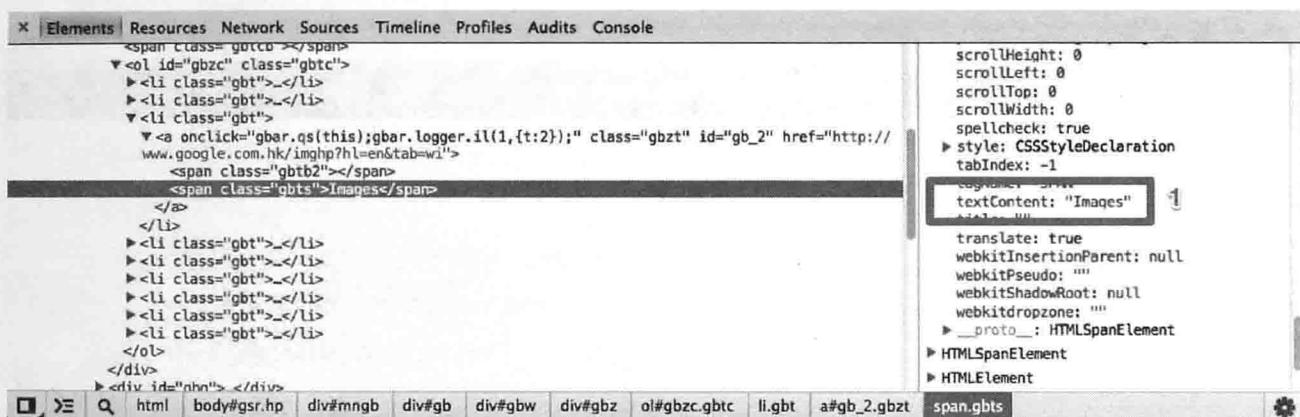


图 3.17 查看 Images 菜单的属性

在 Console 中输入命令 elements [2].textContent 进行验证，如图 3.18 所示。

```
> elements[2].textContent
"Images"
>
```

图 3.18 通过 elements [2].textContent 验证 Images

以上是通过浏览器的调试工具动态加载 jQuery 库，并验证元素属性的过程。接下来将这一过程用 Java 代码组织起来。完整的示例代码如下：

```
package com.learning selenium. findelement;

import static org. testng. AssertJUnit. assertEquals;
import java. util. List;
import org. testng. annotations. Test;
import org. testng. annotations. AfterClass;
import org. testng. annotations. BeforeClass;
import org. openqa. selenium. chrome. * ;
import org. openqa. selenium. * ;

public class testInjectJQuery {
```

```
WebDriver driver;
JavascriptExecutor jse;

@BeforeClass
public void setUp() {
    System.setProperty("webdriver.chrome.driver",
        "/Users/bob/Dropbox/mypublication/Selenium 2/selenium/
chromedriver");
    driver = new ChromeDriver();
    jse = (JavascriptExecutor) driver;
}

@AfterClass
public void tearDown() {
    driver.close();
    driver.quit();
}

@Test
public void testGoogle() throws InterruptedException {
    driver.get("http://www.google.com/");

    //load jQuery dynamically if needed
    injectjQueryIfNeeded();

    //find the top menu items on google page
    List<WebElement> elements = (List<WebElement>) jse
        .executeScript("return jQuery.find('li.gbt span.gbts')");

    //verify the count of the menu items
    assertEquals(12, elements.size());

    //verify the property of element "Images"
    assertEquals("Images",
        elements.get(2).getAttribute("textContent"));
}

private void injectjQueryIfNeeded() {
```



```
if ( ! jQueryLoaded() ) {
    injectjQuery();
}

public Boolean jQueryLoaded() {
    Boolean loaded = true;
    try {
        loaded = ( Boolean ) jse.executeScript("return jQuery() != null");
    } catch ( WebDriverException e ) {
        loaded = false;
    }
    return loaded;
}

public void injectjQuery() {
    //load jQuery dynamically in the head of web page
    jse.executeScript(
        "var headID = document.getElementsByTagName('head')[0];" +
        "var newScript = document.createElement('script');" +
        "newScript.type ='text/javascript';" +
        "newScript.src ='http://ajax.googleapis.com/ajax/libs/" +
        "/jquery/1.10.1/jquery.min.js';" +
        "headID.appendChild( newScript )");
}
```

3.4 元素的 Actions

不同的页面元素具有不同的 Actions，例如一个按钮就可以具备“单击”Action，但你肯定无法在按钮上“输入”文字。虽然在 WebDriver 的 API 中，所有的 Actions 罗列在 WebElement 中，但测试开发人员务必弄清楚这些 Actions 的适用对象。如果在错误的页面元素上使用了无效的 Actions，不会看到有任何的错误提示信息并且相关的操作也不会被触发，因为 WebDriver 会忽略当前页面元素所不支持的 Actions。下面逐个认识一下页面元素可能具备的 Actions。

1. sendKeys ()

适用于具备文本编辑区域的页面元素。常见的使用方式是在文本框中输入字符串。示例代码如下：

```
WebElement searchBox = driver.findElement(By.name("q"));
searchBox.sendKeys("webdriver");
```

如果希望在文本框中输入某些特殊字符，如 Shift，则需要使用 WebDriver 中的 Keys 类。Keys 是一个数组类，用于模拟多种不同的特殊按键输入。例如，希望输入字母的大写形式，手工的方式就是按住 Shift 键的同时输入相应字母即可。为了使用 Keys 达到这个效果，示例代码如下：

```
WebElement searchBox = driver.findElement(By.name("q"));
searchBox.sendKeys(Keys.chord(Keys.SHIFT, "webdriver"));
```

2. clear ()

适用于具备文本编辑区域的页面元素，作用是清除文本编辑区域中输入的文本信息。虽然用户可以通过上述 sendKeys() 方法配合 Key.BACK_SPACE 来达到目的，但这么处理略显繁琐。最简单的方式就是直接调用 clear() 方法。基本用法如下：

```
WebElement searchBox = driver.findElement(By.name("q"));
searchBox.clear();
```

3. submit ()

适用于 form 或者 form 中的页面元素，作用是提交 form 到 Web 的服务器端。示例代码如下：

```
WebElement searchBox = driver.findElement(By.name("q"));
searchBox.submit();
```

4. isDisplayed ()

适用于任意的页面元素，用于判断该元素是否在页面上可见。示例代码如下：

```
WebElement searchButton = driver.findElement(By.name("btnK"));
System.out.println(searchButton.isDisplayed());
```

5. isEnabled ()

适用于任意的页面元素，用于判断该元素是否为启用状态。示例代码如下：

```
WebElement searchButton = driver.findElement(By.name("btnK"));
System.out.println(searchButton.isEnabled());
```

6. isSelected ()

适用于单选按钮、多选按钮，以及选项等页面元素，用于判断某个元素是否被选中。如果在其他页面元素上调用该方法，程序会返回 false。示例代码如下：

```
WebElement searchButton = driver.findElement(By.name("btnK"));
System.out.println(radioButton.isSelected());
```

7. getAttribute ()

适用于任意的页面元素，用于获取当前页面元素的属性。例如，Google 页面中搜索按钮



的 HTML 代码如下：

```
<input value="Google Search" jsaction="sf.chk" name="btnK" type="submit">
```

如果已经通过搜索按钮的 name 查找到该元素，并且希望获取其 value 值，则示例代码如下：

```
WebElement searchButton = driver.findElement(By.name("btnK"));
System.out.println("Value of the button is:" +
    searchButton.getAttribute("value"));
```

8. getText()

适用于任意的页面元素，用于获取元素上的可见文本内容。如果文本内容为空，则该方法返回空。示例代码如下：

```
WebElement searchButton = driver.findElement(By.name("btnK"));
System.out.println(searchButton.getText());
```

9. getTagName()

适用于任意的页面元素，用于获取元素的 tag name。例如，Google 搜索按钮的 HTML 如下，其中 input 就是搜索按钮的 tag name：

```
<input value="Google Search" jsaction="sf.chk" name="btnK" type="submit">
```

如果已经通过搜索按钮的 name 查找到该元素，并且希望获取其 tag name 值，则示例代码如下：

```
WebElement searchButton = driver.findElement(By.name("btnK"));
System.out.println(searchButton.getTagName());
```

10. getCssValue()

适用于任意的页面元素，用于获取当前页面元素的 CSS 属性信息，如 cursor、font-family、font-size、height、background-color、background-image 等。其使用示例代码如下：

```
WebElement searchButton = driver.findElement(By.name("btnK"));
System.out.println(searchButton.getCssValue("height"));
```

11. getLocation()

适用于任意的页面元素，用于获取元素在页面上的相对位置，其中坐标系原点位于页面的左上角。该方法的返回值是一个包括 (x, y) 的坐标信息。示例代码如下：

```
WebElement searchButton = driver.findElement(By.name("btnK"));
System.out.println(searchButton.getLocation());
```

12. getSize()

适用于任意可见的页面元素，用于获取元素的宽度和高度信息，其返回值是一个包括 (width, height) 的长宽组合。示例代码如下：



```
WebElement searchButton = driver.findElement(By.name("btnK"));
System.out.println(searchButton.getSize());
```

3.5 小结

本章首先展示了如何通过不同的主流浏览器的开发者工具来获取页面元素的相关信息，如页面元素的 ID、Name、CSS 等。接下来详细介绍了通过 WebDriver 来定位页面元素的多种方式，包括如何以 jQuery 库为基础来查找元素。最后讲解了不同的页面元素所具有的 Actions。

第 4 章

初识 Selenium WebDriver

4.1 简介

★4.1.1 概述

2011年7月，Selenium团队对外正式发布了Selenium 2.0，又名Selenium WebDriver。Selenium 2.0主要的新特性就是将WebDriver集成进来。Selenium WebDriver不仅可以用于传统意义上的网页测试，还能支持嵌入式设备如智能手机上的应用程序测试。

Selenium 2.0 支持大多数的浏览器，并且能绑定的编程语言也极为广泛，包括 C#、Java、Python、Ruby、JavaScript 等。Selenium 2.0 (WebDriver) 是从操作系统层面直接控制浏览器的行为，因此测试用例程序与浏览器的交互将非常高效。

★4.1.2 WebDriver 与 Selenium RC 的区别

1. WebDriver

- 1) WebDriver 不需要 Selenium Server 就可以运行测试用例。
- 2) WebDriver 独立使用原生浏览器来运行测试用例。
- 3) WebDriver 既可以测试传统桌面 Web 应用，也可以测试手机上的应用程序，如 iPhone 或 Android 上的 app 程序。
- 4) WebDriver 能支持大多数浏览器的最新版本。

2. Selenium RC

- 1) Selenium RC 需要 Selenium Server 才能运行测试用例。
- 2) Selenium RC 使用 JavaScript 来驱动浏览器运行测试用例。
- 3) Selenium RC 只能支持 Web 应用的测试。
- 4) Selenium RC 能支持所有浏览器但并不能及时支持最新版本。

关于 WebDriver 的更多信息可查阅官方网站：

<http://docs.seleniumhq.org/projects/webdriver/>

4.2 WebDriver 的架构

★4.2.1 synthesized 事件和 native 事件

需要解释一下什么是 synthesized 事件。与 Selenium RC 的采用远低于用户操作级别的底



层 API 来操作浏览器的理念不同，Selenium WebDriver 的设计思路是尽可能准确地模拟用户与 Web 应用交互的方式。为了绕开浏览器安全策略方面的限制，Selenium WebDriver 总是尽可能在操作系统层面去触发“native 事件”。由于这些“native 事件”不是由浏览器生成，因此这种方式可以有效地绕开浏览器的安全限制。此外，由于这种实现是基于特定的操作系统来编码，一旦在某个操作系统的一种浏览器中运行良好，那么在移植到同一个操作系统的另一个浏览器中时，重用已有的实现代码会相对容易一些。

以早期 Selenium RC 输入元素的 API 为例，如下所示，需要如此多的 API 接口：type、typeKeys、typeKeysNative、keydown、keypress、keyup、keydownNative、keypressNative、keyupNative、attachFile。

而在 WebDriver 中，对应的方法只需要一个 API 接口——sendKeys 即可。

★4.2.2 RPC 调用

从上述例子可以看出 WebDriver 和 RC 设计理念的差异，WebDriver 尽可能地模拟用户行为，而 RC 提供了较低层次的 API 事件。在 RC 中会比较多地使用 synthesized 事件，而 WebDriver 则更多地使用操作系统的 native 事件。可以直接将事件发送到浏览器窗口的处理函数，这意味着即使没有获取浏览器窗口焦点，也可以使用 WebDriver 操作浏览器。

由于 WebDriver 的调用都是 RPC（Remote Procedure Call，远程过程调用）方式，因此其性能好坏取决于网络延迟。虽然大多数操作系统会优化到本地路由 localhost，但是随着待测浏览器数量的增加，以及测试代码量的增加而导致的网络延迟增加，原本高效的 RPC 也会逐步变得低效。正是出于性能上的考虑，对于 WebDriver 的 API 设计就显得尤为重要。通过设计一个通用型的 API，将多次函数调用合并在一次调用里来减少网络延迟，以此来达到 API 调用的高效性。当然，任何事物都有两面性，通用型的 API 就意味着其命名不是针对某一个特定的操作，由此可能导致难以记忆和理解。所以在设计 API 的时候，需要综合权衡 API 的调用高效性和可读性。

★4.2.3 兼容性矩阵

WebDriver 从设计之初就围绕着支持多种操作系统、多种浏览器和多种编程语言绑定而展开。可以想象，如果在框架和 API 设计上稍有不慎，就会深陷维护成本急剧攀升的旋涡。

尝试减少 WebDriver 支持的编程语言绑定是一种降低成本的途径。但 Selenium 团队并没有选择这种方案。主要因素有两个方面，从个人的角度而言，学习一门新的编程语言需要时间成本。而从公司的角度出发，为了选择一门工具而引入另外一种编程语言会让编码规范和技术单一纯正性得到挑战。当然，第二个因素随着业界的成熟已经在慢慢淡化，很多解决方案需要混合多种模块和编程语言。

选择减少对不同浏览器的兼容性支持也不失为一种选择，但这么做难免会遭到业界的不满。举一个小例子，即使当初 Selenium 团队决定对市场份额不到 1% 的 Firefox 2 不支持 WebDriver 时，也遇到了相当大的阻力和抗议。因此，浏览器的兼容性是绝对需要得到保障的。

综上所述，Selenium 团队唯一能做的就是竭尽全力保障 WebDriver 能支持的兼容性矩阵



尽可能达到最大。而达到这个目标的唯一方法就是让不同浏览器上不同编程语言绑定的接口都保持简洁一致。

★4.2.4 缺陷

像 WebDriver 这种基于原生浏览器进行开发和发布的方式，相较于 RC 的优势已经在前面进行了详述。并且由于紧密绑定浏览器，WebDriver 获取了更多针对浏览器操作的控制权。

正是因为这种机制，WebDriver 也有其自身的一些缺点。第一个缺点是某些特定接口无人问津，这在 WebDriver 发布初期显得尤为突出。但随着文档的完善，这种情况已经得到了改善。第二个缺点是在支持新浏览器时需要花费较大的精力来更新配套的 WebDriver。目前各大厂商的浏览器迭代速度非常快，甚至在核心驱动部分也重新调整了多次，Selenium 团队需要尝试多次修正 WebDriver 的代码才能适配成功。

★4.2.5 与 DOM 交互

Selenium 采用了分层模式的技术堆栈，如图 4.1 所示。在堆栈的底层采用了 Google 的 Closure 库，提供原语和模块化机制，确保了源文件划分的细小粒度。

在 Closure 库之上是 Atom 层。Atom 库提供的函数能够完成一些基本的任务，如获取某个元素的属性值、判断某个元素是否可见。此外，Atom 库还包括一些类似于通过 synthesized 事件模拟用户单击的复杂操作。Atom 库可以被看作浏览器自动化过程中粒度最小的单元。

Atom 库之上则是适配层，主要由 2 代的 WebDriver 和 1 代的 Selenium Core 组成。它们联手组合这些 Atom 单元来完成更为高级的自动化工作。

Selenium 团队之所以选择 Closure 库有如下几个原因：

- 1) Closure 编译器能很好地理解库本身所采用的模块化技术。
- 2) Closure 编译器的目标输出是 JavaScript。
- 3) Selenium 团队中的几位成员对 Closure 库非常的熟悉。

Atom 库的代码在与 DOM 进行交互时会被用于项目中的方方面面。针对不同的语言绑定，处理方式略有差异：

- 1) JavaScript：可以直接使用 Atom 库，并且通常被编译成单一的文件脚本。
- 2) Java：来自 WebDriver 适配层的各个函数在编译的时候会启用优化。生成的 JavaScript 在 JAR 中作为资源被包含进来。
- 3) C：如 iPhone WebDriver 和 IE WebDriver，不仅各个函数在编译时都启用了优化，而且生成的输出文件也被转换定义成头文件中的常量，并被 WebDriver 中的 JavaScript 执行模块所调用。这样处理的好处之一就是 JavaScript 被置于底层驱动中，无须在代码各个地方直接暴露裸代码。

广泛使用 Atom 库的第一个优势在于可以确保在不同浏览器之间的行为一致性。并且由

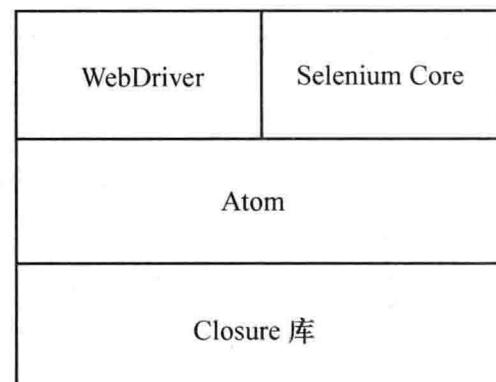


图 4.1 Selenium 分层模式的技术堆栈



于采用 JavaScript 编写，其开发周期普遍比较短。通过 Closure 库动态加载各种库依赖，Selenium 开发人员可以在浏览器中仅仅通过单击刷新按钮就能查看刚刚修改代码的结果。一旦在某一个浏览器中测试通过，那么很容易就能够在另一个浏览器中加载代码并确保通过，因为 Closure 库在隔离浏览器差异性方面有着不错的抽象机制。

Atom 库的第二个优势在于降低了修改代码的成本。原先一个需要在多种实现中反复验证和修复的问题，现在只需要在一个地方修改便可收工。这种方式除了降低维护成本，同时还提高了软件的稳定性和有效性。

采用 Atom 库的第三个好处是使一种很重要的代码迁移手段成为了现实。Selenium 团队编写了一个模拟层去模拟现有的 RC 实现，而在底层实际调用了 WebDriver 的实现代码。通过这种方式来保证已经编写了大量基于 RC 实现的机构或组织可以在只调整几行代码的情况下就能适配到最新的 WebDriver API 上，并享受到 WebDriver 带来的各种优势。由于 Selenium Core 是原子化的，所以可以单独编译每一个函数，这就确保了编写这个模拟层易于实现并且精准无误。本书会在后面具体阐述这种实用有效的迁移方式。

4.3 WebDriver、Eclipse 和 Java

Selenium WebDriver 的下载和安装并不是一件难事。这里为了配合较为广泛使用 Java 语言的程序员，仅以 WebDriver 的 Java 语言的语言绑定进行讲解。除了需要安装 JDK，基本上不会有什么其他让人烦心的事情。作为一个入门教程，还是一步一步看看如何下载、安装和配置 WebDriver，还有其他附属组件的安装和配置。

步骤 1：下载并安装 Java 开发环境

1) 在系统中安装 JDK (Java 开发工具包，Java Development Kit)。

注意：

这里需要安装的是 JDK，而不是单纯的 JRE (Java 运行时环境，Java Runtime Environment)。

2) 由于 Sun 公司已经被 Oracle 收购，所以请到 Oracle 官方网站下载 JDK。地址如下：

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

步骤 2：下载并安装 Eclipse

到 Eclipse 官方网站下载针对 Java 开发人员的版本：

<http://www.eclipse.org/downloads/>

步骤 3：下载 WebDriver 的 Java 客户端驱动

1) Selenium WebDriver 支持多种编程语言的绑定并且每种语言都有自己的客户端驱动。这里所展示的是基于 Java 编程语言的范例，因此需要下载 WebDriver Java Client Driver。请到 Selenium 的官方地址进行下载：

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

2) 由于客户端驱动的版本更新较快，用户可根据具体需要下载相应的版本，如图 4.2



所示。这里所涉及的版本只用于将范例阐述清楚。

Language	Client Version	Release Date	Download	Change log	Javadoc
Java	2.37.0	2013-10-18	Download	Change log	Javadoc
C#	2.37.0	2013-10-18	Download	Change log	API docs
Ruby	2.37.0	2013-10-18	Download	Change log	API docs
Python	2.37.1	2013-10-21	Download	Change log	API docs
Javascript (Node)	2.37.0	2013-10-18	Download	Change log	API docs

图 4.2 Selenium 的 Java 客户端

3) 下载的 WebDriver Java Client Driver 默认为 zip 包格式。将其解压后如图 4.3 所示，可以看到其包含一个 libs 文件夹、两个 jar 包和 CHANGELOG 文件。下面将讲述如何将它们添加到 Eclipse 中。

步骤 4：启动 Eclipse 并配置 Selenium 2 (WebDriver)

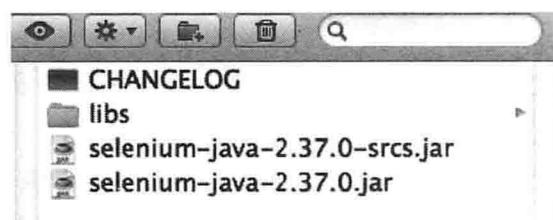


图 4.3 Selenium 的 Java 客户端解包后文件夹

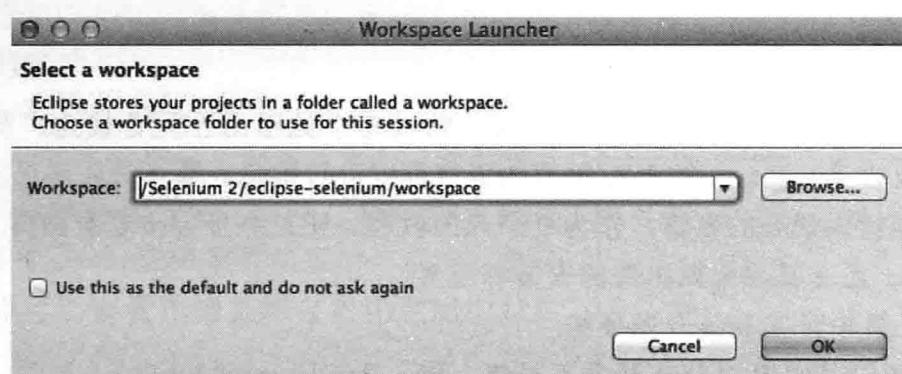


图 4.4 启动 Eclipse 并配置 Selenium 的 Workspace

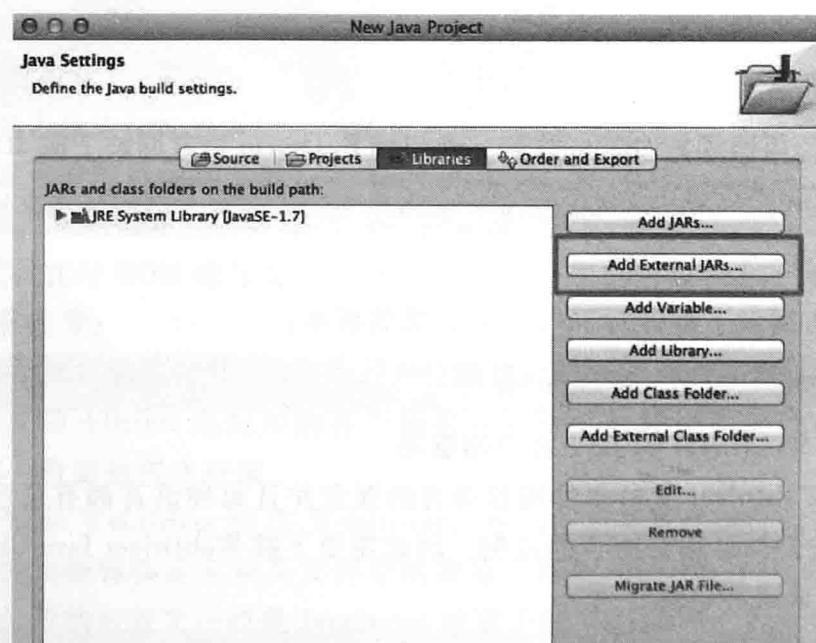


图 4.5 在 Eclipse 中添加 Selenium 库

1) 在 Eclipse 启动时选择 Workspace。如图 4.4 所示，创建一个新目录用于保存 WebDriver 的工作空间。

2) 通过 Eclipse 的菜单选择 File→New→Project→Java Project 命令来创建一个 Java 新项目。在进行新项目设置步骤中，通过 Add External JARs 将之前下载的 WebDriver Java Client Driver 的 libs 和另外两个 jar 包添加到项目中来，如图 4.5 所示。单击 Add External JARs 按钮，选择 libs 下的所有 jar 包并添加进来。单击 Add External JARs 按钮，选择 selenium-java-2.37.0.jar 和 selenium-java-2.37.0-sources.jar 这两个 jar 包并添加进来。

3) 如果一切顺利，恭喜你，接下来可以开始编写基于 WebDriver 的测试用例程序。

4.4 WebDriver 的部署

通过前一节 WebDriver 的 Java 语言绑定客户端，我们已经知道如何将该绑定导入到 Eclipse 开发环境中。但这还只是确保了测试用例可以用 Java 语言来编写并且能与真正的 WebDriver 进行交互。接下来才是真正的部署 WebDriver 来确保测试用例可以驱动相对应的浏览器并执行测试用例。不同浏览器所对应的 WebDriver 下载地址如下：

1) Firefox Driver：由于 Firefox Driver 是直接打包在 WebDriver Java Client Driver 中，因此如果已经按照 4.3 节中所描述的步骤成功下载了 WebDriver Java Client Driver，就不需要再另外下载 Firefox Driver。

2) Chrome Driver：支持三种不同的操作系统平台，包括 Windows、Linux 和 Mac OS。下载地址为

<http://code.google.com/p/chromedriver/>

3) Internet Explorer Driver：只能在 Windows 操作系统平台上运行，但是要区别 32 位版本和 64 位版本。下载地址为

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

在使用相应的 WebDriver 之前，应确保运行环境的操作系统和浏览器都与 WebDriver 所需的运行环境相匹配。

★4.4.1 使用 Firefox Driver

Firefox Driver 是最容易配置和使用的 WebDriver，因为所有的准备工作都伴随 Java 语言绑定的客户端被打包在一起。只要按照前述章节下载 WebDriver Java Client Driver 就能够使用 Firefox WebDriver。

下面以一个简单的例子展示如何在测试用例中调用 Firefox Driver。示例代码如下：

```
package com.learningselenium.simplewebdriver;

import org.openqa.selenium.*;
import org.openqa.selenium.firefox.FirefoxDriver;
```



```
public class testFirefoxDriver {  
    public static void main( String[ ] args ) {  
        WebDriver driver = new FirefoxDriver();  
        driver.get("http://www.baidu.com");  
        String url = driver.getCurrentUrl();  
        System.out.println(url);  
        driver.close();  
    }  
}
```

示例代码详解：

1) 导入 Selenium 库和 FirefoxDriver 库。

2) 启用 FirefoxDriver，代码段如下：

```
WebDriver driver = new FirefoxDriver();
```

3) driver.get() 方法将在 Firefox 浏览器中打开百度的主页。

4) driver.getCurrentUrl() 方法将获取当前页面的 URL 地址并存储在变量 url 中。如果相同的操作需要在 Selenium IDE 中完成，那么需要通过 storeLocation 命令来完成。

5) 通过系统命令将变量 url 的字符串值打印出来，如图 4.6 所示。

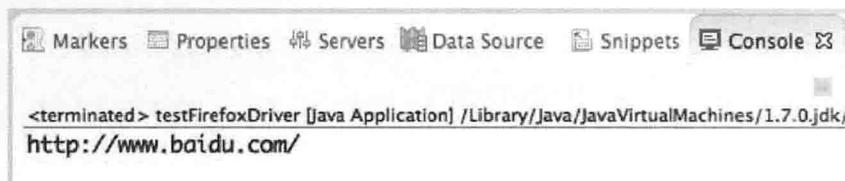


图 4.6 在 Console 中打印的日志信息

以上就是最简单的通过 FirefoxDriver 来驱动浏览器并访问 Web 页面的示例程序。可以自行修改 URL 并替换成自己的网页地址来进行测试。

小贴士：如何使用自定义的 FirefoxProfile？

FirefoxProfile 用于定制待测试的 Firefox 浏览器的特定属性，其中包括所存储的密码、书签、历史信息、Cookies 等。某些测试用例需要用到特定的用户信息，因此可以通过定制当前 Firefox 运行实例的 FirefoxProfile 来达到目标。

1) 如果需要查看当前 Firefox 运行实例的 FirefoxProfile，可以通过选择 Help→Troubleshooting Information→Profile Folder 来获取。

2) 如果希望在 Firefox 启动的时候已经加载某个插件，可以通过 addExtension 提前加载以 .xpi 为扩展名的插件。

3) 如果希望 Firefox 以某些特定偏好设置启动，则可以通过 setPreference 达到目的。

4) 如果希望 Firefox 对 SSL 证书的处理机制进行调整, 可以通过 `setAssumeUntrustedCertificateIssuer` 和 `setAcceptUntrustedCertificates` 来达到目的。

5) 如果希望将 `FirefoxProfile` 导出成 JSON 格式, 可以通过 `toJson` 来处理。

使用 `FirefoxProfile` 的示例代码如下:

```
public class testFirefoxProfile {
    public static void main( String[ ] args ) {
        String profileInJson = " ";
        FirefoxProfile profile = new FirefoxProfile( );
        try {
            profile.addExtension( new File( "/path/to/extension.xpi" ) );

            profile.setPreference( "browser.startup.homepage",
                "about:blank" );

            profile.setAssumeUntrustedCertificateIssuer( false );
            profile.setAcceptUntrustedCertificates( false );

            profileInJson = profile.toJson( );
            System.out.println( profileInJson );

        } catch ( IOException e ) {
            e.printStackTrace( );
        }

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

        driver.close( );
    }
}
```

小贴士: 测试机器上安装了多个 Firefox, 如何指定运行哪一个?

可以通过 `FirefoxBinary` 来指定运行某个路径下的 Firefox, 示例代码如下:

```
public class testFirefoxBinary {
    public static void main( String[ ] args ) {
```





```

FirefoxBinary firefoxBinary =
    new FirefoxBinary( new File("/path/to/chrome") );
FirefoxProfile firefoxProfile = new FirefoxProfile();

WebDriver driver = new FirefoxDriver( firefoxBinary, firefoxProfile );
driver.get("http://www.baidu.com");
driver.close();
}
}

```

★4.4.2 使用 Chrome Driver

前面学会了如何使用 Firefox Driver 来运行程序。接下来展示的是如何使用 Chrome Driver 来完成同样的测试功能。正如前面所述，有别于 Firefox Driver，Chrome Driver 需要单独进行下载。下载地址已经在前面引用，这里就不再赘述。

如图 4.7 所示，请针对相应的操作系统进行下载，并将下载到的 zip 文件解压得到 chromedriver 文件。

Name	Last modified	Size	ETag
Parent Directory		-	
chromedriver_linux32.zip	2013-11-22 23:02:01	7.08MB	f054ab7718126c55daa73dd8b13d10bf
chromedriver_linux64.zip	2013-11-22 23:54:23	7.24MB	07e80e731ee84bc16834c36142d03b8e
chromedriver_mac32.zip	2013-11-23 00:37:41	8.11MB	7a548bf2d0802a9ea6b9425d682d40a5
chromedriver_win32.zip	2013-11-23 13:18:28	2.96MB	f9b86861545951bd24da0235f9793f67
notes.txt	2013-11-22 23:02:09	0.00MB	8da18f0b553807a79265aabca9929bd3

图 4.7 下载 Chrome Driver

使用 Chrome Driver 的示例代码如下：

```

package com.learning selenium.simplewebdriver;

import java.util.concurrent.TimeUnit;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class testChromeDriver {
    static Thread thread = new Thread();
}

```

```

public static void main( String[ ] args ) {
    System. setProperty( "webdriver. chrome. driver",
        "/Selenium 2/selenium/chromedriver" );

    WebDriver driver = new ChromeDriver();
    driver. get("http://www. baidu. com");
    driver. manage( ). timeouts( ). implicitlyWait(10, TimeUnit. SECONDS);

    if( driver. findElement( By. id("kw") ). isEnabled( ) ) {
        System. out. println("Baidu Search text box is editable! ");
        driver. findElement( By. id("kw") ). sendKeys("selenium");
        driver. findElement( By. id("su") ). click();
    }
    else {
        System. out. println("Baidu Search text box is not editable! ");
    }

    try{
        thread. sleep(5000);
    }
    catch( Exception e ) {
        System. out. println("Error");
    }

    driver. close();
}
}

```

示例代码详解：

- 1) 导入 Selenium 库和 ChromeDriver 库。
- 2) 开启一个延时的线程，用于处理页面出错的情况。
- 3) 通过以下方式来加载 ChromeDriver，其中第二个参数为 chromedriver 的具体路径：

```

System. setProperty( "webdriver. chrome. driver",
    "/Selenium 2/selenium/chromedriver" );

```

- 4) 通过 ChromeDriver 打开 Google Chrome 浏览器并访问百度主页。代码段如下：

```

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

```



5) 显式地让 ChromeDriver 等待 10 秒以让百度主页完全加载成功。代码段如下：

```
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
```

6) 如果搜索输入框为可编辑状态，则输入 selenium 并单击搜索按钮进行搜索，否则打印提示搜索输入框为不可编辑状态。代码段如下：

```
if(driver.findElement(By.id("kw")).isEnabled()) {  
    System.out.println("Baidu Search text box is editable!");  
    driver.findElement(By.id("kw")).sendKeys("selenium");  
    driver.findElement(By.id("su")).click();  
}  
else {  
    System.out.println("Baidu Search text box is not editable!");  
}
```

程序成功运行后在控制台会有图 4.8 所示的打印信息，包括 ChromeDriver 的版本信息和测试程序中的正常打印信息。

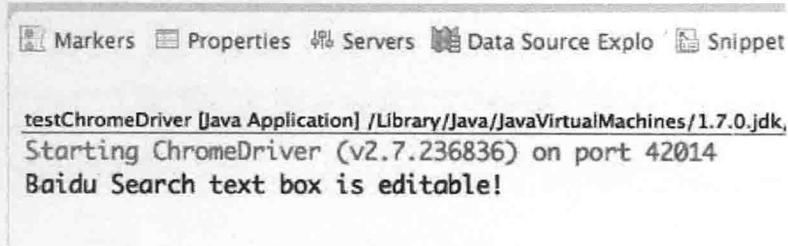


图 4.8 在 Console 中的 ChromeDriver 的版本信息和测试程序的打印日志

注意：

由于 Chrome Driver 只兼容 Chrome 浏览器 12.0.712.0 和之后的新版本，因此如果需要在老版本的 Chrome 浏览器上使用 Selenium，则只能使用 Selenium RC 来完成任务。示例代码段如下：

```
URL seleniumRC = new URL ("http://localhost: 4444");  
URL testWebLink = new URL ("http://www.google.com");  
CommandExecutor executor = new SeleneseCommandExecutor (  
    seleniumRC, testWebLink, DesiredCapabilities.chrome());  
WebDriver driver = new RemoteWebDriver (executor);
```

小贴士：如何使用 ChromeOptions？

ChromeOptions 类似于 FirefoxProfile，用于定制待测试的 Chrome 浏览器的特定属性。

- 1) 如果希望测试某个浏览器插件，则可以通过 addExtensions 方式提前加载以 .crx 为扩展名的插件。
- 2) 如果希望 Chrome 浏览器启动时附带启动参数，则可以通过 addArguments 方式来加载。
- 3) 如果希望指定机器上特定的某个 Chrome 版本来运行测试，尤其是同一台机器上安装了多个不同版本的 Chrome 时，则可以通过 setBinary 来指定待测试的 Chrome。

使用 ChromeOptions 的示例代码如下：

```
ChromeOptions options = new ChromeOptions();

options.addExtensions( new File("/path/to/extension.crx") );
options.addArguments("arguments list");
options.setBinary("/path/to/chrome");

WebDriver driver = new ChromeDriver(options);
```

小贴士：如何使 ChromeDriver 每次启动的端口不会随机变化？

用心的读者会发现，ChromeDriver 在不指定任何参数的情况下，启动监听端口会随机变化。如果需要保证其端口固定不变，则可以通过 ChromeDriverService 来达到目的。示例代码如下：

```
public class testChromeService {
    public static void main( String[ ] args ) {
        System.setProperty("webdriver.chrome.driver",
                           "/Selenium2/selenium/chromedriver");

        ChromeDriverService. Builder builder =
            new ChromeDriverService. Builder();
        ChromeDriverService chromeService = builder
            .usingDriverExecutable(
                new File("/Selenium2/selenium/chromedriver"))
            .usingPort(3333). build();

        try {
            chromeService.start();
        } catch ( IOException e ) {
```



```
e.printStackTrace();
}

WebDriver driver = new ChromeDriver(chromeService);

driver.get("http://www.google.com");
WebElement searchBox = driver.findElement(By.name("q"));
searchBox.sendKeys("webdriver");
searchBox.submit();

driver.quit();
chromeService.stop();

}
```

★4.4.3 使用 Internet Explorer Driver

Internet Explorer Driver 只能运行在 Windows 操作系统上，相较于 Firefox Driver 和 Chrome Driver，其运行速度略显缓慢。Internet Explorer Driver 分为 32 位和 64 位两个版本。具体是哪个版本的浏览器被启动，完全取决于所使用的 Internet Explorer Driver 的版本。

配置 Internet Explorer Driver 的注意事项：

- 1) 请确保 IEDriverServer 的可执行文件在系统环境变量的 PATH 中。
- 2) 在 IE7 和以上版本的 Internet Explorer 上，必须确保保护模式的正确设置。设置方式为 Tools→Internet Options→Security。每个不同的 Zone 的 Protected Mode 都需要保持一致，要么都是 Enable 状态，要么都是非 Enable 状态。
- 3) Internet Explorer 的缩放尺寸都必须设置为 100%，这样原生的鼠标事件才能在正确的坐标系中工作正常。

这里简单阐述一下如何在 Java 代码中调用 Internet Explorer Driver。通过如下代码段就可以达到目的：

```
WebDriver driver = new InternetExplorerDriver();
```

如果之前忘记将下载的 IEDriverServer 可执行文件的路径配置到 PATH 中，还有一种方式可以告诉程序到什么地方找到 IEDriverServer 的可执行文件。代码段如下：

```
System.setProperty ("webdriver.ie.driver",
"D:\Driver\IEDriverServer_Win32_2.37.0_latest\IEDriverServer.exe");
```



在使用 IEDriverServer 可执行文件时，从理论上来说是可以通过它来创建并使用多个同时存在的 Internet Explorer Driver 实例的。但在实际使用过程中，总是会碰到与 cookie 相关的问题、窗口焦点的问题、浏览器多实例等可能会面临的问题。如果真心希望使用 Internet Explorer Driver 的多个实例并且尽可能地避免前述可能遇到的问题，建议考虑 RemoteWebDriver 的方式，并通过多台虚拟机来隔离干扰。

有一种解决方案来应对使用 Internet Explorer Driver 多个实例时的 cookie 共享问题，即在 Internet Explorer 启动时先清理会话中的脏数据。这可以通过将 IE_ENSURE_CLEAN_SESSION 参数传递给 Internet Explorer Driver 并在此模式下启动 Internet Explorer Driver 来达到目的。完整的示例代码如下：

```
package com.learning selenium.simplewebdriver;

import org.openqa.selenium.*;
import org.openqa.selenium.ie.InternetExplorerDriver;
import org.openqa.selenium.remote.DesiredCapabilities;

public class testInternetExplorerDriver {
    public static void main(String[] args) {
        System.setProperty("webdriver.ie.driver",
            "D:\\Driver\\IEDriverServer_Win32_2.37.0_latest\\IEDriverServer.exe");
    }

    DesiredCapabilities capab = DesiredCapabilities.internetExplorer();
    capab.setCapability(InternetExplorerDriver.IE_ENSURE_CLEAN_SESSION,
        true);

    WebDriver driver = new InternetExplorerDriver(capab);

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

示例代码详解：

- 1) 导入 InternetExplorerDriver 包。
- 2) 导入 DesiredCapabilities 包。代码段如下：

```
import org.openqa.selenium.remote.DesiredCapabilities;
```

- 3) 如果在 PATH 中忘记设置 IEDriverServer 可执行文件的路径，在代码中也可以通过系统设置来达到目的。
- 4) 设置 DesiredCapabilities 的属性包含 IE_ENSURE_CLEAN_SESSION 以确保在浏览器实例启动之前清理会话的脏数据。代码段如下：



```
DesiredCapabilities capab = DesiredCapabilities.internetExplorer();
capab.setCapability(InternetExplorerDriver.IE_ENSURE_CLEAN_SESSION, true);
```

5) 启动 Internet Explorer Driver 并通过浏览器打开百度主页。

小贴士：如何绕过 IE 的安全保护模式？

自从 IE7 引入 Protected Mode 以来，IE 浏览器的安全性的确得到了一定程度的提高。其原理从本质上讲，在浏览某些需要启用保护模式的页面时，会开启一个新的浏览器会话以完成任务，而此时你无法控制前一个会话中的 COM 对象。随之而来的问题是，WebDriver 在这种情况下会遇到如下错误提示信息：

```
org.openqa.selenium.WebDriverException: Unexpected error launching
Internet Explorer. Protected Mode must be set to the same value (enabled or disabled)
for all zones.
```

解决方案 1：

可以通过设置 Internet Explorer 浏览器对于所有 Zone 的 Protected Mode 一致来达到目标。Zone 包括 Internet、Local intranet、Trusted sites 和 Restricted sites。这 4 个 Zone 上的选项 Enable Protected Mode 要么全部勾选，要么全部不勾选。

解决方案 2：

解决方案 1 对于只有少量的 Windows 测试机器而言具有可行性。而对于具有大规模的 Windows 测试机器的机构而言，去设置每一台 Windows 机器上的 Internet Explorer 的 Protected Mode 的工作量过于庞大。有没有更加高效的解决方案来达到同样的效果呢？答案是通过设置 Internet Explorer Driver 的 Capability 为 IE_ENSURE_CLEAN_SESSION 达到目的。示例代码如下：

```
public class testInternetExplorerDriver {
    public static void main( String[] args ) {
        System.setProperty("webdriver.ie.driver",
        "D:\\Driver\\IEDriverServer_Win32_2.37.0\\IEDriverServer.exe");

        DesiredCapabilities capab =
            DesiredCapabilities.internetExplorer();
        capab.setCapability(
            InternetExplorerDriver.IE_ENSURE_CLEAN_SESSION, true);

        capab.setCapability(
```

```

InternetExplorerDriver.

INTRODUCE_FLAKINESS_BY_IGNORING_SECURITY_DOMAINS,
true);

WebDriver driver = new InternetExplorerDriver(capab);

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

```

小贴士：如何使 InternetExplorerDriver 每次启动的端口不会随机变化？

用心的读者会发现，Internet Explorer Driver 在不指定任何参数的情况下，启动监听端口会随机变化。如果需要保证其端口固定不变，可以通过 InternetExplorerDriverService 来达到目的。示例代码如下：

```

public class testInternetExplorerService {
    public static void main(String[] args) {
        System.setProperty("webdriver.ie.driver",
"D:\Driver\IEDriverServer_Win32_2.37.0\IEDriverServer.exe");

        InternetExplorerDriverService builder =
            new InternetExplorerDriverService.Builder();
        InternetExplorerDriverService internetExplorerService =
            builder.usingPort(5678).withHost("127.0.0.1").build();

        DesiredCapabilities capab =
            DesiredCapabilities.internetExplorer();
        capab.setCapability(
            InternetExplorerDriver.IE_ENSURE_CLEAN_SESSION, true);

        capab.setCapability(
            InternetExplorerDriver.
            INTRODUCE_FLAKINESS_BY_IGNORING_SECURITY_DOMAINS,
            true);

        WebDriver driver = new InternetExplorerDriver(
            internetExplorerService, capab);
    }
}

```



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

4.5 WebDriver 与浏览器

★4.5.1 操作页面元素之单选按钮

在某个网站注册时，“性别”字段一般有两个选项：“男”和“女”。如果不允许用户选择多个选项，可以使用表单元素的单选按钮对象。单选按钮对象用于一组互相排斥的值，也就是用户只能从选项列表中选择一项。单选按钮组中所有按钮共享同一个名称，所以浏览器知道将按钮组合在一起，通过选中其中一个按钮，其他按钮自动变为未选中状态。

接下来以如下页面上的单选按钮为例进行讲解，如图 4.9 所示。

http://www.w3schools.com/html/html_forms.asp

Radio Buttons

```
<input type="radio"> defines a radio button. Radio buttons let a user sele  


---



```
<form>
<input type="radio" name="sex" value="male">Male

<input type="radio" name="sex" value="female">Female
</form>
```


```

How the HTML code above looks in a browser:

| |
|---------------------------------------|
| <input checked="" type="radio"/> Male |
| <input type="radio"/> Female |

图 4.9 单选按钮的测试页面

示例代码如下：

```
package com.learning selenium.normalwebdriver;  
  
import static org.junit.Assert.*;  
  
import org.junit.After;  
import org.junit.Before;  
import org.junit.Test;  
import org.openqa.selenium.By;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.firefox.FirefoxDriver;
```

```

import org.openqa.selenium.WebElement;
public class testRadioButton {

    WebDriver driver = new FirefoxDriver();

    @Before
    public void setUp() throws Exception {
        driver.get("http://www.w3schools.com/html/html_forms.asp");
    }

    @Test
    public void testRadioButton() throws Exception {
        WebElement femaleRadioButton =
            driver.findElement(
                By.xpath("// *[@id = \"main\"]/form[3]/input[2]"));

        if( ! femaleRadioButton.isSelected() ) {
            femaleRadioButton.click();
        }

        assertTrue(femaleRadioButton.isSelected());
    }

    @After
    public void tearDown() throws Exception {
        driver.quit();
    }
}

```

示例代码详解：

- 1) 使用 FirefoxDriver 来操作浏览器并打开测试页面。
- 2) 使用 XPath 来定位 radio 元素，此例中以 Female 这个 radio 按钮作为测试目标。
- 3) 如果 Female radio 按钮没有选中，则对其进行 click 操作。
- 4) 在执行 click 操作以后，再次验证 Female radio 是否被选中。

在 WebDriver 中，元素的状态检测一般分为如表 4.1 所示几种。

表 4.1 WebDriver 中的元素状态检测表

| 方法 | 作用 | 方法 | 作用 |
|--------------|-----------|---------------|----------|
| isEnabled() | 检测元素是否启用 | isDisplayed() | 检测元素是否可见 |
| isSelected() | 检测元素是否被选中 | | |



★4.5.2 操作页面元素之多选按钮

本节阐述如何通过 WebDriver 来操作页面上的多选按钮，即 Checkbox。依旧以 w3schools 的页面为例进行讲解。WebDriver 基于 Firefox Driver，并且本例会分别以 XPath 和 CSS 两种方式展示 Checkbox 元素的定位。

测试页面如图 4.10 所示，需要测试的 Checkbox 元素分别为 I have a bike 和 I have a car。

Checkboxes

```
<input type="checkbox"> defines a checkbox. Checkboxes let a user select ZERO or MORE options.
```

```
<form>
<input type="checkbox" name="vehicle" value="Bike">I have a bike<br>
<input type="checkbox" name="vehicle" value="Car">I have a car
</form>
```

How the HTML code above looks in a browser:

<input checked="" type="checkbox"/> I have a bike
<input type="checkbox"/> I have a car

图 4.10 多选按钮的测试页面

示例代码如下：

```
package com.learningselenium.normalwebdriver;

import static org.junit.Assert.assertTrue;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

public class testCheckbox {
    WebDriver driver = new FirefoxDriver();

    @Before
    public void setUp() throws Exception {
        driver.get("http://www.w3schools.com/html/html_forms.asp");
    }

    @Test
```

```

public void testCheckbox() throws Exception {
    WebElement bikeCheckbox =
        driver.findElement(
            By.xpath("// *[@id = \"main\"] /form[4] /input[1]"));

    if( ! bikeCheckbox.isSelected() ) {
        bikeCheckbox.click();
    }

    assertTrue( bikeCheckbox.isSelected() );

    WebElement carCheckbox =
        driver.findElement(
            By.cssSelector("input[value='Car']"));

    if( ! carCheckbox.isSelected() ) {
        carCheckbox.click();
    }

    assertTrue( carCheckbox.isSelected() );
}

@After
public void tearDown() throws Exception {
    driver.quit();
}
}

```

示例代码详解：

1) 基于 Firefox Driver 来驱动浏览器并打开测试页面：

```
http://www.w3schools.com/html/html_forms.asp
```

2) 通过 XPath 的方式定位到 Bike 这个多选按钮。代码段如下：

```
WebElement bikeCheckbox = driver.findElement(
    By.xpath("// *[@id = \"main\"] /form[4] /input[1]"));
```

3) 如果该多选按钮没有被选中，则单击它。然后验证单击后的状态是否和预期一致。

4) 通过 CSS 的方式定位到 Car 这个多选按钮。代码段如下：



```
WebElement carCheckbox = driver.findElement(  
    By.cssSelector("input[value='Car']"));
```

5) 如果该多选按钮没有被选中，则单击它。然后验证单击后的状态是否和预期一致。

★4.5.3 操作弹出窗口之验证标题

WebDriver 除了可以处理浏览器默认窗口上的元素，还可以处理各种弹出窗口，包括但不限于识别弹出窗口，在新弹出的窗口中执行测试步骤，还能切换到原始窗口进行后续的操作。这些操作在前述介绍 Selenium IDE 的章节中，已经展示过如何完成以上的多窗口步骤。接下来一起见证如何通过 WebDriver 来完成这些复杂的动作。

本节以 Firefox Driver 为主进行阐述。首先打开父窗口，通过变量来记录父窗口的控点，如图 4.11 所示。然后在父窗口中单击某按钮弹出子窗口，如图 4.12 所示。接下来获取所有打开窗口的控点列表，查找到弹出子窗口的控点并切换到子窗口进行操作。最后根据之前保存的父窗口控点切换回父窗口。



图 4.11 待测试的父窗口



图 4.12 待测试的子窗口

示例代码如下：

```
package com.learningselenium.normalwebdriver;

import static org.junit.Assert.*;
import java.util.Set;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

public class testMultipleWindowsTitle {
    WebDriver driver = new FirefoxDriver();

    @Before
    public void setUp() throws Exception {
        driver.get("http://www.w3schools.com/jsref/met_win_open.asp");
    }

    @Test
    public void testMultipleWindowsTitle() throws Exception {
        String parentWindowId = driver.getWindowHandle();

        assertEquals("Window open() Method", driver.getTitle());

        WebElement tryItButton =
            driver.findElement(By.xpath("// *[@id = \"main\"] / div[2] / a"));
        tryItButton.click();

        Set<String> allWindowsId = driver.getWindowHandles();

        for (String windowId : allWindowsId) {
            if (driver.switchTo().window(windowId)
                .getTitle().contains("Tryit")) {
```



```
        driver.switchTo().window(windowId);
        break;
    }
}

assertEquals("Tryit Editor v1.8", driver.getTitle());

driver.switchTo().window(parentWindowId);
assertEquals("Window open() Method", driver.getTitle());

}

@After
public void tearDown() throws Exception {
    driver.quit();
}
}
```

示例代码详解：

1) 本例以 Firefox Driver 为主进行阐述。首先打开父窗口，并且通过变量 parentWindowId 记录下父窗口的控点。示例代码段如下：

```
String parentWindowId = driver.getWindowHandle();
```

2) 验证父窗口的完整标题是 Window open () Method，如图 4.11 中 1 所示。

3) 通过 XPath 定位到父窗口中的 Try it yourself 按钮，如图 4.11 中 2 所示，单击它以打开弹出窗口。

4) 获取所有打开窗口的控点列表。示例代码段如下：

```
Set<String> allWindowsId = driver.getWindowHandles();
```

5) 在该控点列表中查找包含 Tryit 字符串作为标题的窗口控点，找到后则停止查找。

6) 切换到包含 Tryit 字符串作为标题的窗口，并验证其完整标题为 Tryit Editor v1.8。示例代码段如下：

```
driver.switchTo().window(windowId).getTitle().contains("Tryit")
```

7) 通过父窗口的控点再次切换回原父窗口，并再次验证其完整标题为 Window open () Method，确认此处窗口切换功能的完成。示例代码段如下：

```
driver.switchTo().window(parentWindowId);
assertEquals("Window open() Method", driver.getTitle());
```

★4.5.4 操作弹出窗口之验证内容

本节沿用了4.5.3节中的示例内容，唯一的区别在于找到新弹出窗口的方式不再是通过其标题来进行确定，而是直接通过网页上的内容来进行确认。因为有些弹出窗口本身就不具备标题或者名字，因此只能通过其页面自身所具有的元素内容来进行确认。本例中以新弹出窗口上的一部分文本字符串作为确认目标，如图4.13中4所示：open a new browser window。

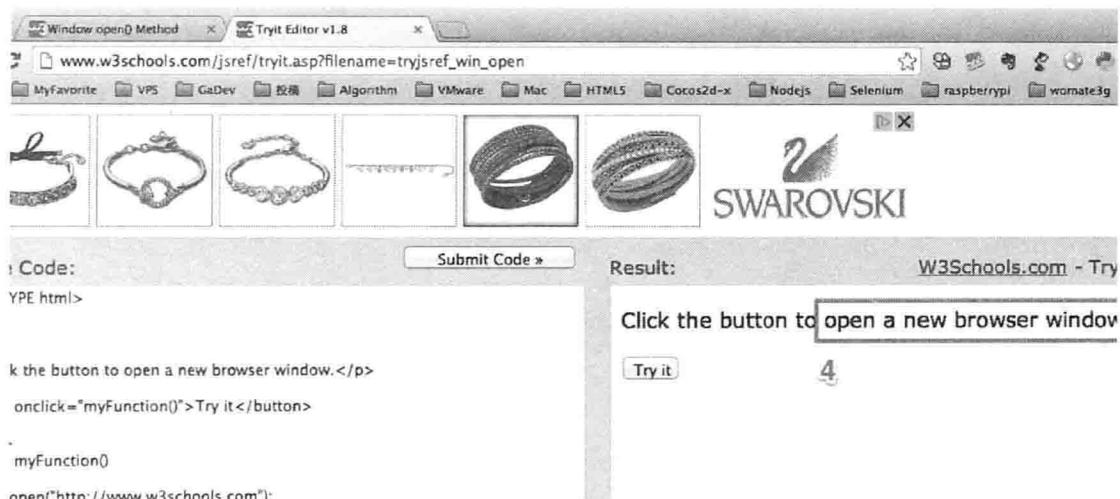


图4.13 弹出窗口和窗口上的字符串信息

示例代码如下：

```

package com.learningselenium.normalwebdriver;

import static org.junit.Assert.assertEquals;

import java.util.Set;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

public class testMultipleWindowsPageContain {
    WebDriver driver = new FirefoxDriver();

    @Before

```



```
public void setUp() throws Exception {
    driver.get("http://www.w3schools.com/jsref/met_win_open.asp");
}

@Test
public void testMultipleWindowsPageContain() throws Exception {
    String parentWindowId = driver.getWindowHandle();

    assertEquals("Window open() Method", driver.getTitle());

    WebElement tryItButton =
        driver.findElement(By.xpath("// * [@id = \"main\"] /div[2]/a"));
    tryItButton.click();

    Set<String> allWindowsId = driver.getWindowHandles();

    for( String windowId : allWindowsId ) {
        if( driver.switchTo().window(windowId)
            .getPageSource().contains("open a new browser window") ) {
            driver.switchTo().window(windowId);
            break;
        }
    }

    assertEquals("Tryit Editor v1.8", driver.getTitle());
}

driver.switchTo().window(parentWindowId);
assertEquals("Window open() Method", driver.getTitle());
}

@Before
public void tearDown() throws Exception {
    driver.quit();
}
```

示例代码详解：

- 1) 本例以 Firefox Driver 为例进行阐述。首先打开父窗口，并且通过变量 parentWindowId 记录下父窗口的控点。
- 2) 验证父窗口的完整标题是 Window open() Method，如图 4.11 中 1 所示。
- 3) 通过 XPath 定位到父窗口中的 Try it yourself 按钮，如图 4.11 中 2 所示，单击它以打开弹出窗口。
- 4) 获取所有打开窗口的控点列表。
- 5) 在该控点列表中查找页面内容中包含 open a new browser window 字符串的窗口控点，找到后则停止查找。示例代码段如下：

```
driver.switchTo().window(windowId)
    .getPageSource().contains("open a new browser window")
```

- 6) 切换到包含 Tryit 字符串作为标题的窗口，并验证其完整标题为 Tryit Editor v1.8。
- 7) 通过父窗口的控点再次切换回原父窗口，并再次验证其完整标题为 Window open() Method，确认此处窗口切换功能的完成。

★4.5.5 操作警告框、提示框和确认框

Web 前端开发人员通常需要利用 JavaScript 弹出对话框来给用户一些提示信息，包括以下几种类型但不限于：

- 1) 警告框：用于提示用户相关信息的验证结果、错误或警告等。
- 2) 提示框：用于提示用户在当前对话框中输入数据，一般需要用户单击取消或者确认按钮。
- 3) 确认框：用于提示用户确认或者取消某个操作。一般需要用户单击取消或者确认按钮。

本节以如下页面为例进行讲解，如图 4.14 所示，包括了警告框、提示框和确认框。

<http://sislands.com/coin70/week1/dialogbox.htm>

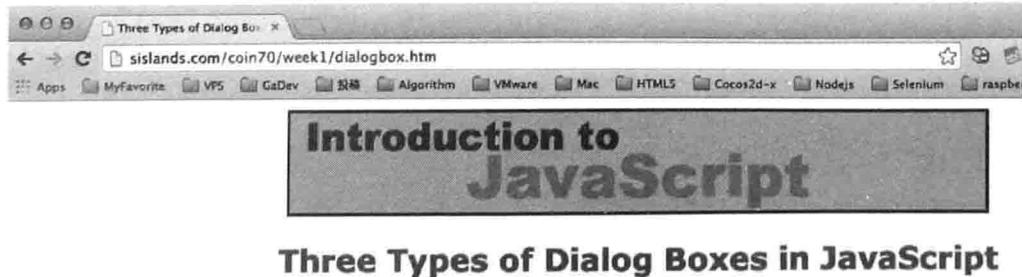


图 4.14 包括警告框、提示框和确认框的待测试页面

示例代码如下：



```
package com.learning selenium. normalwebdriver;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.Alert;

public class testDialogs {
    WebDriver driver = new FirefoxDriver();

    @Before
    public void setUp() throws Exception {
        driver.get("http://sislands.com/coin70/week1/dialogbox.htm");
    }

    @Test
    public void testAlertDialog() throws Exception {
        WebElement alertButton =
            driver.findElement(By.xpath("//input[@value = 'alert']"));

        alertButton.click();
        Alert javascriptAlert = driver.switchTo().alert();
        System.out.println(javascriptAlert.getText());
        javascriptAlert.accept();
    }

    @Test
    public void testPromptDialog() throws Exception {
        WebElement promptButton =
            driver.findElement(By.xpath("//input[@value = 'prompt']"));

        promptButton.click();
        Alert javascriptPrompt = driver.switchTo().alert();
        javascriptPrompt.sendKeys("This Is Learning Selenium");
    }
}
```

```
javascriptPrompt.accept();

System.out.println(javascriptPrompt.getText());

javascriptPrompt = driver.switchTo().alert();
javascriptPrompt.accept();

promptButton.click();
javascriptPrompt = driver.switchTo().alert();
System.out.println(javascriptPrompt.getText());
javascriptPrompt.dismiss();

javascriptPrompt = driver.switchTo().alert();
System.out.println(javascriptPrompt.getText());
javascriptPrompt.accept();

}

@Test
public void testConfirmDialog() throws Exception {
    WebElement confirmButton =
        driver.findElement(By.xpath("//input[@value='confirm']"));

    confirmButton.click();
    Alert javascriptConfirm = driver.switchTo().alert();
    javascriptConfirm.accept();

    javascriptConfirm = driver.switchTo().alert();
    System.out.println(javascriptConfirm.getText());
    javascriptConfirm.accept();

    confirmButton.click();
    javascriptConfirm = driver.switchTo().alert();
    System.out.println(javascriptConfirm.getText());
    javascriptConfirm.dismiss();

    javascriptConfirm = driver.switchTo().alert();
    System.out.println(javascriptConfirm.getText());
}
```



```
    javascriptConfirm.accept();
}

{@After
public void tearDown() throws Exception {
driver.quit();
}
}
```

代码执行完的日志信息如下：

```
This is an alert!!
Your favorite color is: This is Learning Selenium
What is your favorite color?
You pressed Cancel or no value was entered!
Your response was OK!
Confirm Test: Continue?
Your response was Cancel!
```

示例代码详解：

1) 运行警告框的测试用例。通过 XPath 定位到警告按钮并单击它，然后切换到警告框并将其窗口控点赋予 Selenium 的 Alert，以此来操作弹出的警告框。最终执行警告框上确认按钮的功能。

2) 运行提示框的测试用例。在单击页面上的提示框按钮后，会弹出提示框，在文本输入框中输入 This is Learning Selenium 的文本信息，紧接着单击确认按钮。然后再次对提示框进行操作并单击取消按钮。

3) 运行确认框的测试用例。同理也是先执行确认框上确认按钮的功能，再执行确认框上取消按钮的功能。

★4.5.6 操作浏览器最大化

从 2.21 版本开始，WebDriver 也支持浏览器的最大化操作。示例代码如下：

```
package com.learningselenium.normalwebdriver;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public class testMaximizeBrowser {
```

```

WebDriver driver = new FirefoxDriver();

@Before
public void setUp() throws Exception {
    driver.get("http://www.baidu.com/");
}

@Test
public void testMaximizeBrowser() throws Exception {
    driver.manage().window().maximize();
}

@After
public void tearDown() throws Exception {
    driver.quit();
}
}

```

示例代码详解：

- 1) 本例选用 Firefox Driver 为浏览器驱动。
- 2) 打开百度主页。
- 3) 通过 WebDriver 的 window 方法将浏览器窗口最大化。代码段如下：

```
driver.manage().window().maximize();
```

★4.5.7 操作浏览器 Cookies

现在许多社交类的网站必须登录后才能进行后续的操作，而反复地登录需要多次填写用户名和密码。WebDriver 提供了一系列 Cookies 的操作来获取、填写、删除 Cookies 的方法，节省了多次在登录页面的查找元素并填写登录信息的时间。首先以获取 Cookies 的方法为例进行讲解，并且将获取的 Cookies 信息保存到文件中以备后续使用。示例代码如下：

```

package com.learningselenium.normalwebdriver;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileWriter;

import org.openqa.selenium.By;
import org.openqa.selenium.Cookie;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

```



```
public class testGetCookies {  
    public static void main( String... args ) {  
        WebDriver driver = new FirefoxDriver();  
  
        driver.get("http://www.zhihu.com/#signin");  
  
        driver.findElement( By.name("email") )  
            .sendKeys("seleniumcookies@126.com");  
        driver.findElement( By.name("password") ).sendKeys("cookies123");  
        if ( driver.findElement( By.name("rememberme") ).isSelected() ) {  
            driver.findElement( By.name("rememberme") ).click();  
        }  
        driver.findElement( By.className("sign-button") ).click();  
  
        File cookieFile = new File(  
            "zhihu.cookie.txt");  
        try {  
            cookieFile.delete();  
            cookieFile.createNewFile();  
            FileWriter fileWriter = new FileWriter(cookieFile);  
            BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);  
  
            for ( Cookie cookie : driver.manage().getCookies() ) {  
                bufferedWriter.write( ( cookie.getName() + ";"  
                    + cookie.getValue() + ";"  
                    + cookie.getDomain() + ";"  
                    + cookie.getPath() + ";"  
                    + cookie.getExpiry() + ";"  
                    + cookie.isSecure() ));  
                bufferedWriter.newLine();  
            }  
            bufferedWriter.flush();  
            bufferedWriter.close();  
            fileWriter.close();  
        } catch ( Exception ex ) {  
            ex.printStackTrace();  
        }  
        driver.quit();  
    }  
}
```



示例代码详解：

- 1) 打开知乎的登录页面。
- 2) 填写相关用户信息，包括用户名、密码等，然后进行登录。
- 3) 新建一个本地文件 zhihu.cookie.txt，用于存储后续获取的 Cookies 信息。
- 4) 通过 driver.manage().getCookies() 获取 Cookies 信息并通过缓存和文件写操作，保存到 zhihu.cookie.txt 文件中。

接下来展示的是如何读取之前保存的 Cookies 信息并用于自动填充到新打开的浏览器 Cookies 中，然后直接进入登录状态后的页面。示例代码如下：

```
package com.learningselenium.normalwebdriver;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.util.Date;
import java.util StringTokenizer;

import org.openqa.selenium.Cookie;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public class testAddCookies {
    private static BufferedReader bufferedReader;

    public static void main(String... args) {
        WebDriver driver = new FirefoxDriver();

        driver.get("http://www.zhihu.com/#signin");

        try {
            File cookieFile = new File(
                "zhihu.cookie.txt");
            FileReader fr = new FileReader(cookieFile);
            bufferedReader = new BufferedReader(fr);
            String line;
            while ((line = bufferedReader.readLine()) != null) {
                StringTokenizer stringTokenizer =
                    new StringTokenizer(line, ";");
                while (stringTokenizer.hasMoreTokens()) {

```



```
String name = stringTokenizer.nextToken();
String value = stringTokenizer.nextToken();
String domain = stringTokenizer.nextToken();
String path = stringTokenizer.nextToken();
Date expiry = null;
String dt;
if (! (dt = stringTokenizer.nextToken()).equals("null")) {
    expiry = new Date(dt);
}
boolean isSecure =
    new Boolean(stringTokenizer.nextToken()).booleanValue();
Cookie cookie = new Cookie(name,
                            value,
                            domain,
                            path,
                            expiry,
                            isSecure);
driver.manage().addCookie(cookie);
}
}
} catch (Exception ex) {
    ex.printStackTrace();
}

driver.get("http://www.zhihu.com");
}
}
```

示例代码详解：

- 1) 打开知乎的登录页面。
- 2) 读取之前保存 Cookies 信息的 zhihu.cookie.txt 文件并通过缓存操作和 driver.manage().addCookies() 操作将登录信息填充到浏览器的 Cookies 中。
- 3) 打开知乎的主页面，可以看到已经直接进入登录状态后的页面。

★4.5.8 操作浏览器前进后退

WebDriver 还提供了直接操作浏览器进行前进、后退、刷新等操作。示例代码如下：

```
package com.learningselenium.normalwebdriver;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public class testNavigate {
    public static void main(String... args) {
        WebDriver driver = new FirefoxDriver();

        driver.get("http://www.baidu.com");
        System.out.println("Go to url: " + driver.getCurrentUrl());

        driver.navigate().to("http://www.cnblogs.com");
        System.out.println("Navigate to url: " + driver.getCurrentUrl());

        driver.navigate().refresh();

        driver.navigate().back();
        System.out.println("Back to url: " + driver.getCurrentUrl());

        driver.navigate().forward();
        System.out.println("Forward to url: " + driver.getCurrentUrl());

        driver.quit();
    }
}
```

示例代码详解：

- 1) 打开百度主页并打印日志。
- 2) 浏览到 cnblogs 主页并打印日志。
- 3) 刷新浏览器。
- 4) 回退到前一个页面并打印日志。
- 5) 再前进到最后一个页面并打印日志。

测试程序执行成功后的完整打印信息如下：

```
Go to url: http://www.baidu.com/
Navigate to url: http://www.cnblogs.com/
Back to url: http://www.baidu.com/
Forward to url: http://www.cnblogs.com/
```



★4.5.9 操作页面元素等待时间

WebDriver 在操作页面元素等待时间时，提供了两种等待方式：一个为显式等待，另一个为隐式等待。其区别在于：

1) 显式等待：明确地告诉 WebDriver 按照特定的条件进行等待，条件未达到就一直等待。这在等待某个元素需要非常长的时间时非常有效。

2) 隐式等待：告诉 WebDriver 一个最大的超时时间，如果等待的条件在超时以前就满足，则立即执行后续操作而无须等待超时达到。

如何使用显式等待和隐式等待的示例代码如下：

```
package com.learning selenium. normalwebdriver;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.ui.WebDriverWait;
import org.openqa.selenium.support.ui.ExpectedCondition;

public class testWait {
    public static void main(String... args) {
        WebDriver driver = new FirefoxDriver();

        driver.get("http://www.google.com");

        WebElement searchBox = driver.findElement(By.name("q"));
        searchBox.sendKeys("Selenium 2");
        searchBox.submit();

        //Explicit wait
        (new WebDriverWait(driver, 10)).
            until(new ExpectedCondition<Boolean>() {
                public Boolean apply(WebDriver d) {
                    return d.getTitle().toLowerCase().startsWith("selenium");
                }
            });
    }
}
```

```

System.out.println("Page title is: " + driver.getTitle());

driver.navigate().back();

//Implicit wait
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

driver.findElement(By.name("btnK")).click();

driver.quit();
}
}

```

示例代码详解：

- 1) 打开 Google 主页，并在搜索框中输入 Selenium 2 字符串并进行搜索。
- 2) 利用显式等待，等待条件为：直到浏览器的标题以 selenium 为开头出现才进行后续操作。示例代码如下：

```

(new WebDriverWait(driver, 10)).
    until(new ExpectedCondition<Boolean>() {
        public Boolean apply(WebDriver d) {
            return d.getTitle().toLowerCase().startsWith("selenium");
        }
    });

```

- 3) 打印浏览器的标题信息。
- 4) 操作浏览器回退到前一个搜索页面。
- 5) 利用隐式等待，等待条件为：最长等待 10 秒；如果在 10 秒内就已经回退到前一个页面，则直接执行后续操作。示例代码段如下：

```
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
```

- 6) 单击搜索按钮再次进行搜索。

打印浏览器标题信息的日志如下，可以看到代码中使用 toLowerCase（）方法对于包含大写的字符串全部进行了小写转换：

```
Page title is: Selenium 2 - Google ??
```

4.6 WebDriver 与文件系统

★4.6.1 屏幕截图操作

Selenium WebDriver 提供了截图的功能，其接口函数是 TakesScreenshot。该功能是在运



行测试用例的规程中，需要验证某个元素的状态或者显示的数值时，可以将屏幕截取下来进行对比；或者在异常或者错误发生的时候将屏幕截取并保存起来，供后续分析和调试所用。

这里以百度首页为例学习一下截图功能的接口如何使用。示例代码如下：

```
package com.learning selenium.normalwebdriver;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.*;
import java.io.File;
import org.apache.commons.io.FileUtils;

public class testTakesScreenshot {
    WebDriver driver = new FirefoxDriver();

    @ Before
    public void setUp() throws Exception {
        driver.get("http://www.baidu.com/");
    }

    @ Test
    public void testTakesScreenshot() throws Exception {
        File srcFile =
            ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
        FileUtils.copyFile(srcFile, new File("/Selenium 2/screenshot.png"));
    }

    @ After
    public void tearDown() throws Exception {
        driver.quit();
    }
}
```

示例代码详解：

- 1) 本例采用 Firefox Driver 进行讲解，首先打开百度主页。
- 2) TakesScreenshot 接口提供了 getScreenshotAs() 方法以截取屏幕。在这里指定了



OutputType. FILE 作为参数传递给 getScreenshotAs() 方法，其含义为将截取的屏幕以文件形式返回。示例代码段如下：

```
File srcFile = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
```

3) 使用 FileUtils 工具类的 copyFile() 方法保存 getScreenshot() 返回的文件对象。

注意：

TakesScreenshot 接口是依赖于具体的浏览器 API 操作的，所以在 HtmlUnit Driver 中并不支持该操作。

在本例中 OutputType 类采用了文件类型。实际上 OutputType 类可以支持多种数据类型。例如，它还支持 BASE64 编码或者 BYTE 的字符串。以 BASE64 编码为例的示例代码如下：

```
String base64 = ((TakesScreenshot) driver).  
    getScreenshotAs(OutputType.BASE64);
```

★4.6.2 复制文件操作

WebDriver 提供了一个文件操作的类，即 FileHandler。可以使用这个类的 FileHandler.copy() 方法对文件和目录进行复制操作。示例代码如下：

```
package com.learning selenium. file;  
  
import java. io. File;  
import java. io. IOException;  
  
import org. openqa. selenium. io. FileHandler;  
  
public class testCopyFile {  
    public static void main( String... args ) {  
        try {  
            FileHandler. copy( new File( "/source_directory" ) ,  
                new File( "/destination_directory" ) );  
  
            FileHandler. copy( new File( "/source_directory/file. txt" ) ,  
                new File( "/destination_directory/file. txt" ) );  
  
            FileHandler. copy( new File( "/path/of/source_directory" ) ,  
                new File( "/path/of/destination_directory" ) ,  
                "suffix. txt" );  
        } catch ( IOException e ) {  
            e. printStackTrace();  
        }  
    }  
}
```



```
        } catch ( IOException e ) {
            e.printStackTrace( );
        }
    }
}
```

示例代码详解：

1) 将源目录下的所有文件复制到目的目录下面。示例代码段如下：

```
FileHandler.copy( new File( "/source_directory" ) ,
    new File( "/destination_directory" ) );
```

2) 将指定的文件从源目录复制到目的目录。示例代码段如下：

```
FileHandler.copy( new File( "/source_directory/file.txt" ) ,
    new File( "/destination_directory/file.txt" ) );
```

3) 将以 suffix.txt 为扩展名的所有文件从源目录复制到目的目录。示例代码段如下：

```
FileHandler.copy( new File( "/path/of/source_directory" ) ,
    new File( "/path/of/destination_directory" ) ,
    "suffix.txt" );
```

★4.6.3 创建目录操作

FileHandler 除了可以进行复制文件操作，还可以利用 FileHandler.createDir() 方法创建目录。示例代码如下：

```
package com.learning selenium.file;

import java.io.File;
import java.io.IOException;

import org.openqa.selenium.io.FileHandler;

public class testCreateDirectory {
    public static void main( String... args ) {
        try {
            FileHandler.createDir( new File( "/new_created_directory" ) );

        } catch ( IOException e ) {
            e.printStackTrace( );
        }
    }
}
```

★4.6.4 删除目录操作

FileHandler既然有能力将目录创建出来，也具备将目录删除的能力。使用 FileHandler.delete()删除目录的示例代码如下：

```
package com.learningselenium.file;

import java.io.File;

import org.openqa.selenium.io.FileHandler;

public class testDeleteDirectory {
    public static void main(String... args) {
        FileHandler.delete(new File("/new_created_directory"));
    }
}
```

★4.6.5 读取文件操作

FileHandler读取文件操作会调用 FileHandler.readAsString()方法，其示例代码如下：

```
package com.learningselenium.file;

import java.io.File;
import java.io.IOException;

import org.openqa.selenium.io.FileHandler;

public class testReadFile {
    public static void main(String... args) {
        try {
            String file = FileHandler.readAsString(
                new File("/directory/fileToRead.txt"));
            System.out.println(file);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



★4.6.6 压缩目录操作

上述为 WebDriver 所提供的文件基本操作功能，接下来展示如何压缩目录。压缩操作对于需要从远程机器获取大量的日志文件或截屏文件而言非常有效。zip 类的使用方法如下，包括压缩和解压缩：

```
package com.learning selenium. file;

import java. io. File;
import java. io. IOException;

import org. openqa. selenium. io. FileHandler;
import org. openqa. selenium. io. Zip;

public class testZipFile {
    public static void main( String... args ) {
        Zip zip = new Zip();
        try {
            zip. zip( new File( "/directory_to_zip" ),
                    new File( "/final_directory/zipped_file. zip" ) );

            System. out. println(
                FileHandler. isZipped( "/final_directory/zipped_file. zip" ) );
        }

        zip. unzip( new File( "/final_directory/zipped_file. zip" ),
                    new File( "/final_unzipped_directory" ) );
    }

    } catch ( IOException e ) {
        e. printStackTrace();
    }
}
```

示例代码详解：

1) 将指定目录下的所有文件打包压缩成单个 zip 文件。示例代码段如下：

```
zip. zip( new File( "/directory_to_zip" ),
        new File( "/final_directory/zipped_file. zip" ) );
```

2) 判断一个文件是否为压缩文件。示例代码段如下：

```
System.out.println(
    FileHandler.isZipped("/final_directory/zipped_file.zip"));

```

3) 将一个 zip 文件进行解压缩。示例代码段如下：

```
zip.unzip(new File("/final_directory/zipped_file.zip"),
    new File("/final_unzipped_directory"));

```

★4.6.7 临时目录操作

WebDriver 还提供了一个操作临时文件系统的类 TemporaryFilesystem。下面展示获取临时目录的绝对路径和临时目录可用空间的方法。示例代码如下：

```
package com.learningselenium.file;

import java.io.File;

import org.openqa.selenium.io.TemporaryFilesystem;

public class testTemporaryFileSystem {
    public static void main(String... args) {
        File tempDirectory = TemporaryFilesystem.
            getDefaultTmpFS().
            createTempDir("prefix", "suffix");
        System.out.println(tempDirectory.getAbsolutePath());
        System.out.println("Free Space of Temporary Directory is: "
            + tempDirectory.getFreeSpace());
    }
}

```

★4.6.8 文件权限操作

在某些情况下，需要调整文件的权限，如让某个脚本文件变成可执行文件。下面展示如何通过 FileHandler 来调整文件的权限。示例代码如下：

```
package com.learningselenium.file;

import java.io.File;
import java.io.IOException;

import org.openqa.selenium.io.FileHandler;

```



```
public class testFilePermission {  
    public static void main( String... args ) {  
        if( ! FileHandler. canExecute( new File( "/directory/file1.sh" ) ) ) {  
            try {  
                FileHandler. makeExecutable(  
                    new File( "/directory/file1.sh" ) );  
            } catch ( IOException e ) {  
                e. printStackTrace( );  
            }  
        }  
  
        try {  
            FileHandler. makeWritable( new File( "/directory/file2.txt" ) );  
        } catch ( IOException e ) {  
            e. printStackTrace( );  
        }  
    }  
}
```

示例代码详解：

1) 判断文件本身是否是可执行文件。示例代码段如下：

```
FileHandler. canExecute( new File( "/directory/file1.sh" ) )
```

2) 修改文件的权限使其变成可执行文件。示例代码段如下：

```
FileHandler. makeExecutable( new File( "/directory/file1.sh" ) );
```

3) 修改文件的权限使其变为可写文件。示例代码段如下：

```
FileHandler. makeWritable( new File( "/directory/file2.txt" ) );
```

4.7 小结

本章首先介绍了 Selenium 2 的主角 WebDriver。从介绍 WebDriver 与 Selenium RC 的区别逐步深入，进而展示了 WebDriver 的架构，帮助读者理解 WebDriver 的设计理念。接下来阐述了 WebDriver 的部署和 WebDriver 的基本使用方法，包括与页面元素的交互、浏览器的交互、文件系统的交互。在下一章中会介绍更多关于 WebDriver 的进阶知识和高级使用方法。

第 5 章

玩转 Selenium WebDriver

5.1 WebDriver 与 HTML5

★ 5.1.1 HTML5 中的 Video

HTML5 定义了新元素 <video>，本示例将展示如何完成对 <video> 的测试。

本例采用了 JUnit 的方式来组织代码。关于 JUnit 的相关知识可参考其官方主页：

<http://junit.org>

本例采用 Firefox Driver 来进行阐述，并以 videojs 官方页面为例进行讲解。通过视频播放元素的 id 定位到该 <video> 元素，接下来通过 HTML 代码获取视频的播放源并进行确认，最后控制视频的播放和暂停，如图 5.1 所示。

示例代码如下：

```
package com.learningselenium.html5;

import static org.junit.Assert.*;

import org.junit.*;
import org.openqa.selenium.*;
import org.openqa.selenium.firefox.FirefoxDriver;

public class testHTML5VideoPlayer {

    WebDriver driver = new FirefoxDriver();

    @Before
    public void setUp() throws Exception {
        driver.get("http://videojs.com/");
    }

    @Test
    public void testHTML5Video() throws InterruptedException {
    }
}
```





```

WebElement video = driver.findElement(By.id("home_video_html5_api"));
JavascriptExecutor jse = (JavascriptExecutor) driver;

String source = (String) jse.executeScript(
    "return arguments[0].currentSrc;", video);
assertEquals("http://vjs.zencdn.net/v/oceans.webm", source);

jse.executeScript("return arguments[0].play()", video);
Thread.sleep(5000);

jse.executeScript("arguments[0].pause()", video);
}

@After
public void tearDown() throws Exception {
    driver.quit();
}
}

```



图 5.1 测试页面中视频元素的 id 和播放源的 HTML 代码信息

示例代码详解：

- 1) 打开 videojs 的主页面。
- 2) 通过视频播放元素的 id 定位到该元素，其 id 信息如图 5.1 中 1 所示的 home_video_html5_api。
- 3) 通过 HTML 代码获取视频的播放源并进行确认，如图 5.1 中 2 所示的网络地址：

<http://vjs.zencdn.net/v/oceans.webm>



4) 通过 JavaScript 代码播放视频。代码段如下：

```
jse.executeScript("return arguments[0].play()", video);
```

5) 睡眠 5 秒。

6) 通过 JavaScript 代码暂停视频的播放。代码段如下：

```
jse.executeScript("arguments[0].pause()", video);
```

★5.1.2 HTML5 中的 Canvas

HTML5 中新增的 canvas 可谓一代神器，提供了通过 JavaScript 绘制图形的方法。它在 HTML 页面上是一块以 <canvas> 为标签的矩形区域。Canvas 具有多种绘制图形的方法，包括直线、圆圈、字符及绘制图片等。

下面将以如何通过 WebDriver 在 Canvas 上绘制图形为例来进行讲解。地址为

<http://literallycanvas.com>

WebDriver 以 Chrome Driver 为例进行阐述，通过 Actions 在 Canvas 上绘制一个封闭图形。对于 Canvas 上的操作，推荐 Chrome Driver 或者 Firefox Driver。示例代码如下：

```
package com.learning selenium.html5;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.interactions.Actions;

public class testHTML5Canvas {

    WebDriver driver;

    @Before
    public void setUp() throws Exception {
        System.setProperty("webdriver.chrome.driver",
                           "/Selenium 2/selenium/chromedriver");
        driver = new ChromeDriver();
        driver.get("http://literallycanvas.com");
    }
}
```





```
@Test  
public void testHTML5Canvas() throws Exception {  
    WebElement canvas = driver.findElement(  
        By.xpath("// * [ @id = \"about\"] /div[1] /  
        canvas"));  
  
    Actions drawing = new Actions(driver);  
    drawing.clickAndHold(canvas).moveByOffset(10, 50).  
        moveByOffset(50, 10).  
        moveByOffset(-10, -50).  
        moveByOffset(-50, -10).  
        release().perform();  
}  
  
@After  
public void tearDown() throws Exception {  
    driver.quit();  
}  
}
```

示例代码详解：

- 1) 配置 Chrome Driver 的路径并通过 Chrome 浏览器打开 literallycanvas 主页面。
- 2) 通过 XPath 定位到 Canvas 元素，如图 5.2 所示。代码段如下：

```
WebElement canvas =  
driver.findElement(By.xpath("// * [ @id = \"about\"] /div[1] /canvas"));
```

- 3) 通过 Actions 在 Canvas 上绘制一个封闭图形。代码段如下：

```
Actions drawing = new Actions(driver);  
drawing.clickAndHold(canvas).moveByOffset(10, 50).  
    moveByOffset(50, 10).  
    moveByOffset(-10, -50).  
    moveByOffset(-50, -10).  
    release().perform();
```

- 4) 最终绘制的图形如图 5.3 所示。

★5.1.3 HTML5 中的 Drag/Drop

iFrame 用来表示文档中的浮动框架。以下示例将展示如何在 Selenium WebDriver 中操作 iFrame，并通过 Actions 来拖曳和移动元素的位置。本例以 jQuery UI 官方网站的元素进

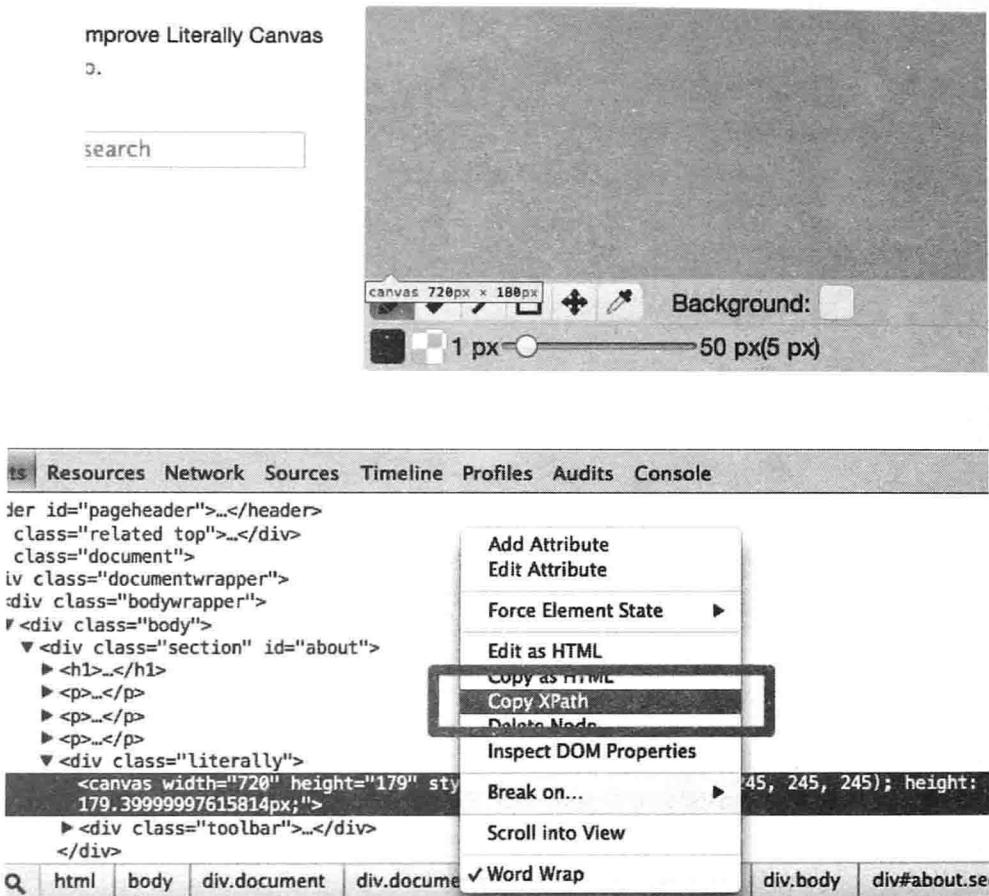


图 5.2 获取 Canvas 元素的 XPath

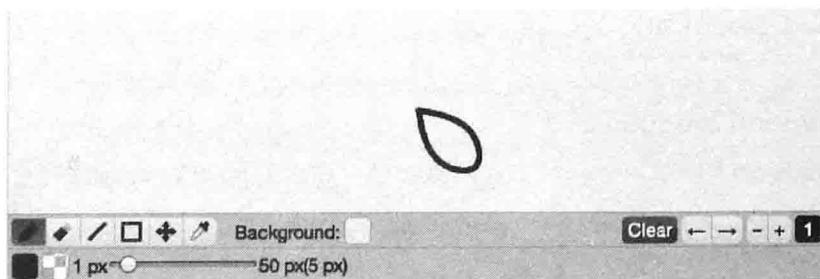


图 5.3 在 Canvas 上绘制的封闭图形

行讲解。地址为

<http://jqueryui.com/draggable>

示例代码如下：

```
package com.learningselenium.html5;

import java.util.NoSuchElementException;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
```



```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.interactions.Actions;

public class testActionDragAndDrop {

    WebDriver driver = new FirefoxDriver();

    @Before
    public void setUp() throws Exception {
        driver.get("http://jqueryui.com/draggable/");
    }

    @Test
    public void testDragAndDrop() throws Exception {
        driver.switchTo().
            frame(driver.findElement(By.className("demo-frame")));
        Thread.sleep(3000);
        if (!isElementPresent(By.xpath("//div[@id='draggable']"))){
            Thread.sleep(3000);
        }
        WebElement draggable =
            driver.findElement(By.xpath("//div[@id='draggable']"));
        new Actions(driver).dragAndDropBy(draggable, 200, 10).
            build().perform();
        Thread.sleep(10000);
    }

    @After
    public void tearDown() throws Exception {
        driver.quit();
    }

    private boolean isElementPresent(By by) {
        try {
            driver.findElement(by);
        }
        catch (Exception e)
            return false;
        return true;
    }
}
```

```
        return true;  
    } catch ( NoSuchElementException e ) {  
        return false;  
    }  
}
```

示例代码详解：

- 1) 使用 Firefox Driver 作为浏览器驱动。
 - 2) 打开 jQuery UI 官方可拖曳例子的页面，如图 5.4 所示为需要关注的 iFrame 部分。

Droppable

Resizable

Selectable

Sortable

Widgets

Accordion

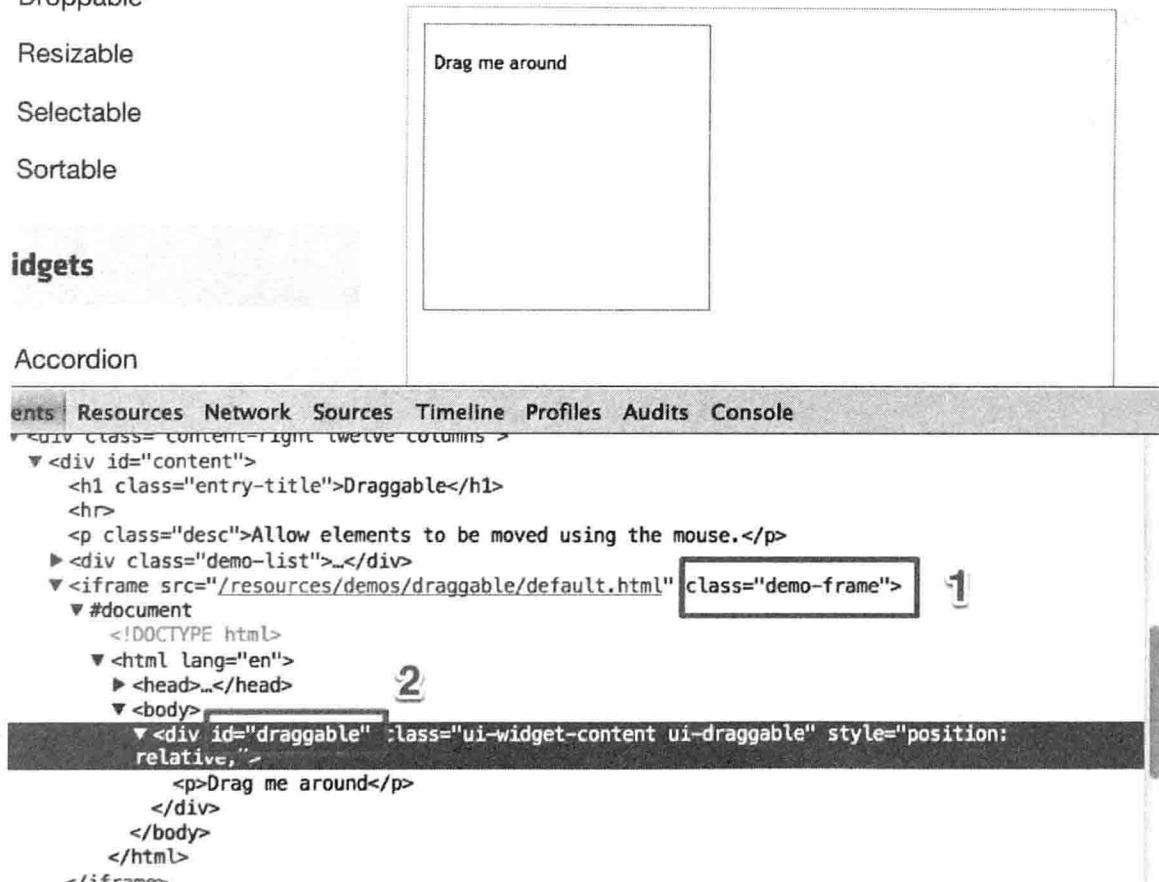


图 5.4 待拖曳矩形方框的 id 信息

- 3) 使用 `switchTo().frame` 来切换焦点到 iFrame 上, 如图 5.4 中 1 所示。此处根据其 `className` 来进行定位。代码段如下:

```
driver.switchTo().frame(driver.findElement(By.className("demo-frame")));
```

- 4) 需要移动的元素为图中具有 Drag me around 的矩形方框，其 id 为 draggable，如图 5.4 中 2 所示。

① 确认该矩形方框在当前页面中可见。代码段如下：



```
isElementPresent(By.xpath("//div[@id='draggable']"))
```

② 确认该矩形方框存在后，通过 Actions 来拖曳该矩形并放置到新的位置。代码段如下：

```
new Actions(driver).dragAndDropBy(draggable, 200, 10).build().perform();
```

★5.1.4 HTML 中的 Geolocation

在 HTML5 中，新特性 Geolocation 用于定位用户的位置信息。由于用户位置信息为敏感信息，所以需要在得到用户的允许后，才能让程序通过 API 获取当前用户信息。Selenium WebDriver 程序每次重新运行都是新的会话进程，即使之前在浏览器中已经通过手工方式运行浏览器访问用户的位置信息，但是在当前运行环境中依旧无法获取之前的用户设置。这个问题的解决方法就是让浏览器在每次执行 Selenium WebDriver 测试程序时，依旧加载之前的用户设置即可。

以 Firefox 为例，在 Mac OS 平台上，可以通过如下命令打开用户 Profile 管理器。

```
$ /Applications/Firefox.app/Contents/MacOS/firefox-bin -ProfileManager
```

打开后新建一个名为 geolocation 的 Profile，如图 5.5 所示。



图 5.5 新建名为 geolocation 的 Profile

其他平台上的打开方式可查阅官方开发者文档：

https://developer.mozilla.org/en-US/docs/Mozilla/Multiple_Firefox_Profiles

创建 geolocation Profile 成功后，单击 Start Firefox 启动 Firefox 浏览器。接下来打开与 HTML5 Geolocation 相关的演示网站：

http://www.w3schools.com/html/html5_geolocation.asp

通过手工的方式单击 try it 按钮并且允许 Share Location，如图 5.6 所示。

接下来便可以看到当前用户的位置信息显示在地图上，如图 5.7 所示。



图 5.6 打开浏览器的 Share Location 设置

HTML5 Geolocation

< Previous

Next Chapter >

HTML5 Geolocation is used to locate a user's position [Try It](#)

图 5.7 HTML5 之 Google 地图

为了演示完整的示例代码，还需要创建一个包含 Geolocation 信息的 JSON 文件。这里命名为 location.json。具体内容如下：

```
{
    "status": "OK",
    "accuracy": 10.0,
    "location": { "lat": 52.1771129, "lng": 5.4099848 }
}
```

这里采用了 TestNG 来组织代码，完整的示例代码如下：

```
package com.learning selenium.html5;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
```



```
import org.openqa.selenium.firefox.FirefoxProfile;
import org.openqa.selenium.firefox.internal.ProfilesIni;
import org.testng.annotations.*;

public class testHTML5Geolocation {
    private static WebDriver driver;

    @BeforeClass
    public void setUp() throws Exception {
        FirefoxProfile profile = new ProfilesIni().getProfile("geolocation");
        profile.setPreference("geo.wifi.uri",
            "/Selenium 2/mydoc/codes/4/location.json");
        driver = new FirefoxDriver(profile);
        driver.get("http://www.w3schools.com/html/html5_geolocation.asp");
    }

    @Test
    public void testGetLocation() throws Exception {
        driver.findElement(By.cssSelector("p#demo button")).click();
    }

    @AfterClass
    public void tearDown() throws Exception {
        driver.quit();
    }
}
```

示例代码详解：

- 1) 选取 geolocation Profile。示例代码段如下：

```
FirefoxProfile profile = new ProfilesIni().getProfile("geolocation");
```

- 2) 通过 location.json 文件配置 Geolocation 信息。示例代码段如下：

```
profile.setPreference("geo.wifi.uri",
    "/Selenium 2/mydoc/codes/4/location.json");
```

- 3) 通过定制的 Profile 信息启动 Firefox 浏览器。
- 4) 通过 Firefox 浏览器打开 HTML5 Geolocation 演示页面。
- 5) 通过 CSS 选择器定位 try it 按钮并且通过 WebDriver 的接口单击该按钮。
- 6) 可以看到地图上会显示当前用户的位置信息。

5.2 RemoteWebDriver

★5.2.1 RemoteWebDriver 简介

RemoteWebDriver 包括两个部分：客户端和服务器端。

- 1) 客户端是基于 WebDriver 运行的测试用例。
- 2) 服务器端是一个简单的 Java Servlet，可运行在任意 Java 运行时环境中。服务器端一般与被测试浏览器运行在同一台主机上。可以通过两种方式启动 RemoteWebDriver 的服务器端：一种是通过命令行的方式，还有一种是在代码中进行配置。

★5.2.2 RemoteWebDriver 的优缺点

1. RemoteWebDriver 的优点

- 1) 使得测试用例和被测试的浏览器不一定要部署在同一台机器上。
- 2) 即使当前运行测试用例的操作系统不支持某浏览器 A，也可以通过发送远程命令到远端待测试的浏览器 A 上。

2. RemoteWebDriver 的缺点

- 1) 需要额外的 Servlet 容器运行环境。
- 2) 从远端服务器发送来的字符串可能存在结尾符号不兼容问题。
- 3) 增加了网络延时。

★5.2.3 RemoteWebDriver 服务器端

首先从 Selenium 官方主页下载 Selenium Server。下载链接如图 5.8 所示。

Selenium Server (formerly the Selenium RC Server)

The Selenium Server is needed in order to run either Selenium RC style scripts or Remote Selenium Webdriver ones. The 2.x server is a drop-in replacement for the old Selenium RC server and is designed to be backwards compatible with your existing infrastructure.

[Download version 2.37.0](#)

To use the Selenium Server in a Grid configuration [see the wiki page](#).

图 5.8 下载 Selenium Server

1. 命令行方式启动 RemoteWebDriver 服务器端

- 1) 命令行启动 RemoteWebDriver：

```
$ java -jar selenium-server-standalone-{VERSION}.jar
```

- 2) 超时设置：针对 Selenium Server 有两种超时设置，如表 5.1 所示。

表 5.1 Selenium Server 的超时设置

超时类型	描述
timeout	在单次会话中允许客户端断开的时长，单位为 s
browserTimeout	浏览器停止响应的时长，单位为 s



3) 带超时设置的命令行启动方式如下：

```
$ java -jar selenium-server-standalone-{VERSION}.jar -timeout=30  
-browserTimeout=60
```

4) 在 Selenium Server 2.21 版本之后，原超时属性 selenium.server.session.timeout 不再被支持。

5) browserTimeout 是作为 timeout 超时机制的后备方案而存在，这样可以确保在 Grid/Server 的测试环境下，一旦进程发生崩溃时至少有一个超时会被触发。这样可以保证进程崩溃的状况不会持续太久就能被发现，以防止运行时环境变得复杂且难以维护。

2. 在代码中配置 RemoteWebDriver 服务器端

1) RemoteWebDriver 的 DriverServlet 可以封装在一个轻量级的 Servlet 容器中，如 Jetty。Jetty 是一个开源的 Servlet 容器，可以作为嵌入式服务器使用。Jetty 的运行速度较快，而且是轻量级的，可以在 Java 代码的测试用例中控制其运行。从而可以使自动化测试不再依赖外部环境，顺利实现自动化测试。

2) 将下载的 selenium-server.jar 放置到系统的 CLASSPATH 下。

3) 创建一个新的类 jettyEmbeddedSeleniumServer。

示例代码如下：

```
package com.learningselenium.remotewebdriver;  
import org.mortbay.jetty.Server;  
import org.mortbay.jetty.nio.SelectChannelConnector;  
import org.mortbay.jetty.webapp.WebAppContext;  
  
public class jettyEmbeddedSeleniumServer {  
    public static void main(String s[]) throws Exception {  
        Server server = new Server();  
  
        WebAppContext context = new WebAppContext();  
        context.setContextPath("");  
        File st = new File(".");  
  
        context.setWar(st.getPath());  
        server.addHandler(context);  
  
        context.addServlet(DriverServlet.class, "/wd/*");  
  
        SelectChannelConnector connector = new SelectChannelConnector();  
        connector.setPort(3002);  
        server.addConnector(connector);  
    }  
}
```



```
server.start();  
}  
}
```

★5.2.4 RemoteWebDriver 客户端

在前面的章节中，已经阐述了如何在 WebDriver 中使用 TakesScreenshot 接口来截取屏幕信息。但是在 RemoteWebDriver 中不能直接使用该接口来完成相同的操作，因为 RemoteWebDriver 没有实现 TakesScreenshot 这个类。如果浏览器本身的 Driver 具备截屏的功能，就可以通过 TakesScreenshot 接口的 Augmenter 类来间接完成在 RemoteWebDriver 中截取屏幕的操作。

示例代码如下：

```
package com.learningselenium.remotewebdriver;  
  
import java.io.File;  
import java.net.URL;  
import org.apache.commons.io.FileUtils;  
import org.junit.After;  
import org.junit.Before;  
import org.junit.Test;  
import org.openqa.selenium.OutputType;  
import org.openqa.selenium.TakesScreenshot;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.firefox.FirefoxDriver;  
import org.openqa.selenium.remote.Augmenter;  
import org.openqa.selenium.remote.DesiredCapabilities;  
import org.openqa.selenium.remote.RemoteWebDriver;  
  
public class testRemoteWebDriverTakesScreenshot {  
    WebDriver driver;  
  
    @Before  
    public void setUp() throws Exception {  
        driver = new RemoteWebDriver(  
            new URL("http://localhost:4444/wd/hub") ,  
            DesiredCapabilities.firefox());  
    }  
  
    @Test
```



```
public void testRemoteWebDriverTakesScreenshot() throws Exception {  
    driver.get("http://www.google.com");  
  
    WebDriver augmentedDriver = new Augmenter().augment(driver);  
  
    File screenshot = ((TakesScreenshot)augmentedDriver).  
        getScreenshotAs(OutputType.FILE);  
    FileUtils.copyFile(screenshot,  
        new File("/Selenium 2/remotewebdriver_screenshot.png"));  
}  
  
@After  
public void tearDown() throws Exception {  
    driver.quit();  
}  
}
```

示例代码详解：

- 1) 以 Firefox 为指定浏览器，然后启动本地的 RemoteWebDriver。
- 2) 打开 Google 主页。
- 3) 由于 Firefox Driver 本身具备截屏功能，所以可以通过 Augmenter 来增强 RemoteWebDriver 的功能，完成之前所不具备的截屏功能。代码段如下：

```
WebDriver augmentedDriver = new Augmenter().augment(driver);  
  
File screenshot = ((TakesScreenshot)augmentedDriver).  
    getScreenshotAs(OutputType.FILE);
```

- 4) 保存截图到具体的文件中。

5.3 WebDriver 的事件处理

在 WebDriver 中存在两个事件处理的类，一个是 EventFiringWebDriver，另一个是 WebDriverEventListener。所有的 WebDriverEventListener 对象都需要注册到 EventFiringWebDriver 上。

★5.3.1 自定义事件侦听

自定义事件侦听有两种方法。一种是实现 WebDriverEventListener 接口，在实现中添加自定义的逻辑；由于是接口，所以需要用户将所有的接口方法全部实现一遍。另外一种是通过继承 AbstractWebDriverEventListener 抽象类来自定义事件侦听类并添加自定义的逻辑，只



需要对有必要添加自定义逻辑的方法编写覆盖的逻辑代码即可。下面以后者为例进行讲解。示例代码如下：

```
package com.learning selenium.event;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.events.AbstractWebDriverEventListener;

public class MyEventListener extends AbstractWebDriverEventListener {

    @Override
    public void afterNavigateTo( String url, WebDriver driver) {
        System.out.println("After Navigate To " + url);
    }

    @Override
    public void afterNavigateBack( WebDriver driver) {
        System.out.println("After Navigate Back To " +
                           driver.getCurrentUrl());
    }

    @Override
    public void afterClickOn( WebElement webElement, WebDriver driver) {
        System.out.println("After Click On " + webElement.getText());
    }

}
```

示例代码详解：

- 1) 通过继承 AbstractWebDriverEventListener 抽象类，只需覆盖其中需要添加自定义逻辑的方法。
 - 2) 覆盖 afterNavigateTo() 方法，表示需要侦听浏览器浏览到某个页面后触发的事件。
 - 3) 覆盖 afterNavigateAfter() 方法，表示需要侦听浏览器回退浏览某个页面后触发的事件。
 - 4) 覆盖 afterClickOn() 方法，表示需要侦听鼠标单击页面上某个元素后触发的事件。
- 接下来通过一个实例来展示如何使用上述自定义的事件侦听。

★5.3.2 事件处理实例

在自定义了事件侦听之后，接下来以百度主页和 Google 主页之间的切换，以及页面元



素的单击操作来讲解自定义事件侦听在这里做了哪些处理。示例代码如下：

```
package com.learning selenium.event;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.events.EventFiringWebDriver;
import com.learning selenium.event.MyEventListener;

public class test WebDriverEvent {
    public static void main(String... args) {
        WebDriver driver = new FirefoxDriver();

        EventFiringWebDriver eventFiringDriver =
            new EventFiringWebDriver(driver);
        MyEventListener myEventListener = new MyEventListener();
        eventFiringDriver.register(myEventListener);

        eventFiringDriver.get("http://www.google.com");
        eventFiringDriver.get("http://www.baidu.com");
        eventFiringDriver.navigate().back();

        eventFiringDriver.findElement(By.name("btnK")).click();

        driver.quit();
    }
}
```

示例代码详解：

- 1) 用 FirefoxDriver 实例化 WebDriver 对象。
- 2) 实例化 EventFiringWebDriver 对象，并将 WebDriver 对象实例传递给 EventFiringWebDriver。
- 3) 实例化自定义事件侦听对象 MyEventListener，并将 MyEventListener 对象注册到 EventFiringWebDriver 对象中。示例代码段如下：

```
MyEventListener myEventListener = new MyEventListener();
eventFiringDriver.register(myEventListener);
```

- 4) 使用 EventFiringWebDriver 对象打开 Google 主页，此时会触发自定义事件侦听中的 afterNavigateTo() 方法并打印日志。



5) 使用 EventFiringWebDriver 对象打开百度主页，此时会触发自定义事件监听中的 afterNavigateTo() 方法并打印日志。

6) 使用 EventFiringWebDriver 对象执行浏览器回退操作，此时会触发自定义事件监听中的 afterNavigateAfter() 方法并打印日志。

7) 使用 EventFiringWebDriver 对象查找 Google 主页上的搜索按钮并进行单击，此时会触发自定义监听中的 afterClickOn() 方法并打印日志。

所有测试代码执行完成后的打印信息如下，包括两次正常浏览、一次回退和一次单击：

```
After Navigate To http://www.google.com  
After Navigate To http://www.baidu.com  
After Navigate Back To https://www.google.com/  
After Click On Google Search
```

通过自定义事件监听，可以更方便地记录测试代码的执行与各种页面操作的预期步骤是否一致；也可以很方便地在某件事件发生后，添加一些辅助代码来协助后续测试用例的执行。

5.4 Page Object 与 Page Factory

所谓 Page Object，顾名思义为“页面对象”，就是将单个页面作为一个对象来处理。每个页面对象都可以看作单个良好结构的页面对象，包括属性和方法。其中，属性对应于每个页面元素，而方法对应于如何去操作这些元素，以及如何与其他的页面对象进行交互。

以面向对象的方式来处理页面和业务流程的好处在于，如果某个页面元素的属性有了变化，只需要在包含这个元素的页面对象中调整操作该元素的属性或者方法即可。

还记得 2.8 节的 Rollup 命令吗？回顾一下，Rollup 命令用于将一系列 Selenium IDE 中能用的命令打包在一起，在 Selenium IDE 中可谓奇兵一枚，因为通过它可以帮测试用例进行“减肥”。Rollup 让 Selenium IDE 只需要使用单独一条 logincommands 就可以替代原先 4 条命令才能完成的动作。即使重复登录，增加的测试用例的步骤也只是多一条 Rollup 命令而已。可见，Rollup 功能在封装多条相同命令时非常高效。

下面以 cnblogs 网站的登录、短消息处理、退出等流程来阐述如果不采用页面对象方式的劣势所在，以及如何通过页面对象来解决问题。页面对象的模式与 Rollup 方法具有异曲同工之妙。

已申请的 cnblogs 的测试账号信息如下：

```
邮箱地址：http://www.cnblogs.com  
用户名：seleniumpageobject  
密码：pageobject123
```

场景 1：

- 1) 打开网站。
- 2) 登录网站。





3) 进入“短消息”文件夹。

4) 发送两条“短消息”。

5) 退出登录状态。

场景 2：

1) 打开网站。

2) 登录网站。

3) 进入“短消息”文件夹。

4) 发送一条“短消息”。

5) 进入“收件箱”文件夹。

6) 删除所有短消息。

7) 退出登录状态。

可以看到，用户有两次登录网站、两次进入“短消息”、三次发送“短消息”、一次删除“短消息”，以及两次退出登录状态的操作。

★5.4.1 不使用 Page Object

在不使用页面对象方式的情况下，完成上述两个场景的测试用例代码如下：

```
package com.learning selenium.pageobject.notuse;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.Alert;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

public class testMessageNoPageObject {
    public static void main(String... args) {
        WebDriver driver = new FirefoxDriver();

        //login
        driver.get("http://www.cnblogs.com");
        driver.findElement(By.linkText("登录")).click();
        WebDriverWait wait = new WebDriverWait(driver, 300);
        wait.until(ExpectedConditions.
            visibilityOfElementLocated(By.id("tbUserName")));
    }
}
```

```
driver.findElement(By.id("tbUserName")).  
    sendKeys("seleniumpageobject");  
driver.findElement(By.id("tbPassword")).sendKeys("pageobject123");  
driver.findElement(By.id("btnLogin")).click();  
driver.manage().timeouts().implicitlyWait(3, TimeUnit.SECONDS);  
  
//message page  
driver.findElement(By.partialLinkText("短消息")).click();  
driver.manage().timeouts().implicitlyWait(3, TimeUnit.SECONDS);  
  
//send message 1  
driver.findElement(By.linkText("撰写新短消息")).click();  
driver.manage().timeouts().implicitlyWait(3, TimeUnit.SECONDS);  
driver.findElement(By.id("txtIncept")).  
    sendKeys("seleniumpageobject");  
driver.findElement(By.id("txtTitle")).  
    sendKeys("testSendingMessage1");  
driver.findElement(By.id("txtContent")).sendKeys("text messages 1");  
driver.findElement(By.id("btnSend")).click();  
  
//send message 2  
driver.findElement(By.partialLinkText("短消息")).click();  
driver.manage().timeouts().implicitlyWait(3, TimeUnit.SECONDS);  
driver.findElement(By.linkText("撰写新短消息")).click();  
driver.manage().timeouts().implicitlyWait(3, TimeUnit.SECONDS);  
driver.findElement(By.id("txtIncept")).  
    sendKeys("seleniumpageobject");  
driver.findElement(By.id("txtTitle")).  
    sendKeys("testSendingMessage2");  
driver.findElement(By.id("txtContent")).sendKeys("text messages 2");  
driver.findElement(By.id("btnSend")).click();  
  
//logout for the 1st time  
driver.findElement(By.linkText("退出")).click();  
wait.until(ExpectedConditions.alertIsPresent());  
Alert logoutPrompt = driver.switchTo().alert();  
logoutPrompt.accept();
```



```
//login
driver.get("http://www.cnblogs.com");
driver.findElement(By.linkText("登录")).click();
wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("tbUserName")));
driver.findElement(By.id("tbUserName")).sendKeys("seleniumpageobject");
driver.findElement(By.id("tbPassword")).sendKeys("pageobject123");
driver.findElement(By.id("btnLogin")).click();
driver.manage().timeouts().implicitlyWait(3, TimeUnit.SECONDS);

//message page
driver.findElement(By.partialLinkText("短消息")).click();
driver.manage().timeouts().implicitlyWait(3, TimeUnit.SECONDS);

//send message 3
driver.findElement(By.linkText("撰写新短消息")).click();
driver.manage().timeouts().implicitlyWait(3, TimeUnit.SECONDS);
driver.findElement(By.id("txtIncept")).sendKeys("seleniumpageobject");
driver.findElement(By.id("txtTitle")).sendKeys("testSendingMessage3");
driver.findElement(By.id("txtContent")).sendKeys("text messages 3");
driver.findElement(By.id("btnSend")).click();

//message page
driver.findElement(By.partialLinkText("短消息")).click();
driver.manage().timeouts().implicitlyWait(3, TimeUnit.SECONDS);

//select all received messages
driver.findElement(By.linkText("收件箱")).click();
WebElement checkBoxSelectAll =
    driver.findElement(By.id("chkSelAll"));
if(!checkBoxSelectAll.isSelected()) {
    checkBoxSelectAll.click();
}

//delete all received messages
```



```
driver.findElement(By.id("btnBatDel")).click();
wait.until(ExpectedConditions.alertIsPresent());
Alert confirmDeletePrompt = driver.switchTo().alert();
confirmDeletePrompt.accept();

//logout
driver.findElement(By.linkText("退出")).click();
wait.until(ExpectedConditions.alertIsPresent());
driver.switchTo().alert().accept();

driver.quit();

}
```

示例代码详解：

- 1) 打开 cnblogs 主页。
- 2) 单击“登录”链接，并利用显式的条件等待来确保登录页面正确打开后再执行后续操作。示例代码段如下：

```
driver.findElement(By.linkText("登录")).click();
WebDriverWait wait = new WebDriverWait(driver, 300);
wait.until(ExpectedConditions.
    visibilityOfElementLocated(By.id("tbUserName")));
```

- 3) 在登录页面输入相关用户信息并进行登录。
- 4) 以“短消息”为链接的部分关键字找到页面元素（因为在“收件箱”有未读短消息的情况下，“短消息”文本后会附带收到的未读短消息的数量），然后单击进入“短消息”页面，并隐式等待 3 秒。示例代码段如下：

```
driver.findElement(By.partialLinkText("短消息")).click();
driver.manage().timeouts().implicitlyWait(3, TimeUnit.SECONDS);
```

- 5) 撰写两条新的短消息并进行发送。
- 6) 退出登录状态，此时会弹出一个确认退出的提示框。这里利用显式的条件等待来确保提示框正常弹出后，再确认退出即可。示例代码段如下：

```
driver.findElement(By.linkText("退出")).click();
wait.until(ExpectedConditions.alertIsPresent());
Alert logoutPrompt = driver.switchTo().alert();
logoutPrompt.accept();
```





- 7) 再次登录并进入“短消息”页面，然后发送一条短消息。
- 8) 进入“收件箱”，选中所有已接收的短消息并进行删除，此时会弹出一个确认删除的提示框，确认删除即可。示例代码段如下：

```
//select all received messages
driver.findElement(By.linkText("收件箱")).click();
WebElement checkBoxSelectAll = driver.findElement(By.id("chkSelAll"));
if(!checkBoxSelectAll.isSelected()) {
    checkBoxSelectAll.click();
}

//delete all received messages
driver.findElement(By.id("btnBatchDel")).click();
wait.until(ExpectedConditions.alertIsPresent());
Alert confirmDeletePrompt = driver.switchTo().alert();
confirmDeletePrompt.accept();
```

- 9) 再次退出登录状态。

从上述代码可以看出，有大量的重复代码存在。想象一下，如果某个页面元素的 ID 有变化，如“撰写新短消息”页面中的“主题”的 ID 调整了，那就不得不修改多处代码来满足需求。如果这样的重复代码有多达 100 多处，那修改调整的工作量就太大了。

为了解决这个问题，可以采用面向对象的方式来处理页面之间的交互。将单个页面上的页面元素和相应操作封装到一个页面对象中，接下来就介绍主角：Page Object。

★5.4.2 使用 Page Object

在上述场景中，存在着这么几个基本页面，以及它们之间的切换：

- 1) 网站主页面，包括“登录”链接、“退出”链接： MainPage。
- 2) 登录页面，包括“用户名”输入框、“密码”输入框和“登录”按钮： LoginPage。
- 3) 短消息页面，包括“短消息”链接、“撰写新短消息”链接、“收件箱”链接： MessagePage。
- 4) 撰写短信页面，包括“收件人”输入框、“主题”输入框、“内容”输入框和“发送”按钮： SendMessagePage。
- 5) 收件箱页面，包括“选中所有”复选框、“删除选中的短消息”按钮： ReceivedMessagePage。
- 6) 退出页面： LogoutPage。

下面分别介绍一下如何通过页面对象的方式分别封装这些页面，以及最终的测试代码。

1. LoginPage

登录页面包括“用户名”输入框、“密码”输入框和“登录”按钮，如图 5.9 所示。

这个页面的作用是验证用户名和密码并登录网站，将其封装为 LoginPage 类。其示例代码如下：

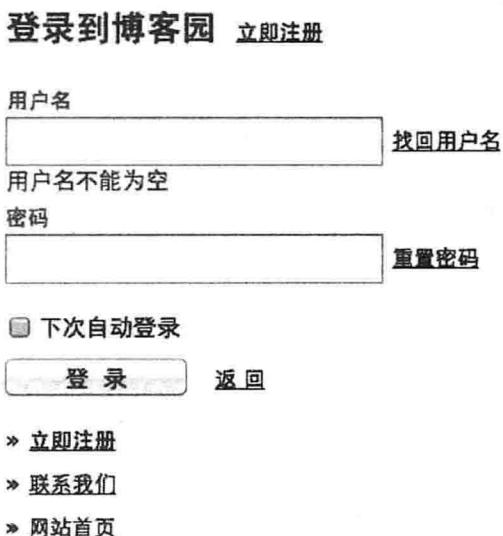


图 5.9 LoginPage 页面

```
package com.learningselenium.pageobject.normaluse;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;

public class LoginPage1 {
    WebDriver driver;
    WebElement username;
    WebElement password;
    WebElement loginbutton;

    public LoginPage1(WebDriver driver) {
        this.driver = driver;

        username = driver.findElement(By.id("tbUserName"));
        password = driver.findElement(By.id("tbPassword"));
        loginbutton = driver.findElement(By.id("btnLogin"));
    }

    public void login(String userName, String passWord) {
        username.sendKeys(userName);
    }
}
```



```
password.sendKeys(password);
loginbutton.click();

driver.manage().timeouts().implicitlyWait(3, TimeUnit.SECONDS);
}

}
```

示例代码详解：

- 1) 在类中定义了三个 WebElement 属性，用于记录“用户名”、“密码”和“登录”按钮这三个页面元素。
- 2) 除了属性，还定义了一个 login() 方法，用于发送用户名和密码并登录网站。

2. LogoutPage

退出页面并不是一个普通意义上的 Web 页面，只是一个弹出确认框而已，如图 5.10 所示。既然如此，为什么还需要将它封装成一个类呢？这么处理的原因在于为了将退出的细节封装起来，以保证测试用例代码不需要关心退出的细节内容。对于测试用例代码而言，所谓“退出”就是单个运行步骤而已。其优势显而易见是在最终的测试用例代码中只需要一句代码，测试逻辑清晰可见。

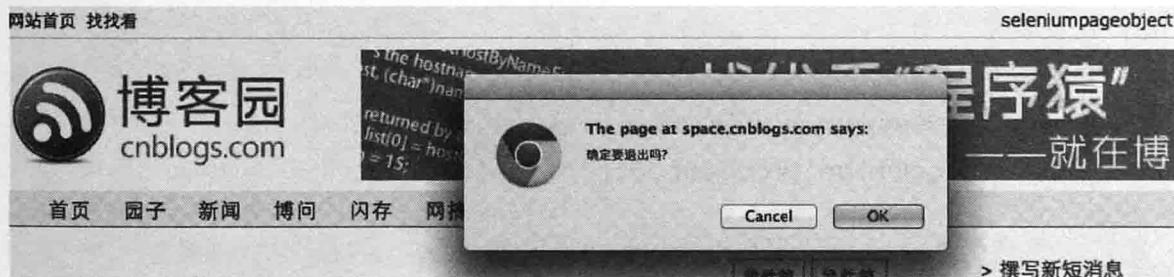


图 5.10 LogoutPage 的退出确认弹出框

LogoutPage 类的示例代码如下：

```
package com.learningselenium.pageobject.normaluse;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

public class LogoutPage1 {
    WebDriver driver;

    public LogoutPage1(WebDriver driver) {
        this.driver = driver;
    }
}
```

```

public void logout() {
    WebDriverWait wait = new WebDriverWait(driver, 300);
    wait.until(ExpectedConditions.alertIsPresent());
    driver.switchTo().alert().accept();
}
}

```

示例代码详解：

- 1) 在构造函数中记录 WebDriver。
- 2) 定义 logout () 方法，封装弹出框的处理细节，包括显式等待弹出框的出现，以及确认退出。

3. MainPage

主页面主要负责“登录”和“退出”两个链接，并通过单击来触发 LoginPage 和 LogoutPage，如图 5.11 和图 5.12 所示。在图 5.12 所示页面上除了“退出”链接，还有“短消息”链接。在 MainPage 类中没有将“短消息”链接元素封装进来，而放置到 MessagePage 中，是为了保证“撰写新短消息”操作的连贯性。



图 5.11 MainPage 的待登录状态

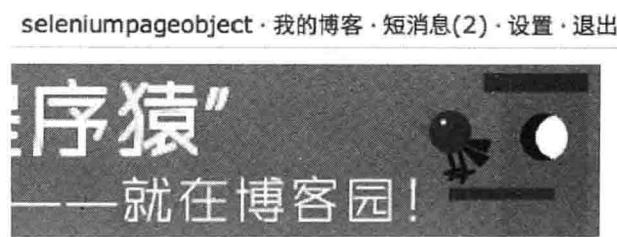


图 5.12 MainPage 的待退出状态

MainPage 类的示例代码如下：

```

package com.learningselenium.pageobject.normaluse;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

public class MainPage1 {
    WebDriver driver;
    WebElement loginLink;
    WebElement logoutLink;

    public MainPage1(WebDriver driver) {

```



```
this.driver = driver;
}

public void open MainPage( String url ) {
    driver.get( url );
    loginLink = driver.findElement( By.linkText("登录") );
}

public void login( String userName, String passWord ) {
    loginLink.click();
    WebDriverWait wait = new WebDriverWait( driver, 300 );
    wait.until( ExpectedConditions.
        visibilityOfElementLocated( By.id("tbUserName") ) );
}

LoginPage1 loginPage = new LoginPage1( driver );
loginPage.login( userName, passWord );
}

public void logout() {
    logoutLink = driver.findElement( By.linkText("退出") );
    logoutLink.click();

    LogoutPage1 logoutPage = new LogoutPage1( driver );
    logoutPage.logout();
}
}
```

示例代码详解：

- 1) 在类中定义了两个 WebElement 属性，用于记录“登录”和“退出”链接这两个页面元素。
- 2) 定义打开主页面的方法 openMainPage ()。
- 3) 定义登录的方法 login，其中构造了 LoginPage 的实例对象并调用 LoginPage. login () 方法。
- 4) 定义退出的方法 logout，其中构造了 LogoutPage 的实例对象并调用 LogoutPage. logout () 方法。

4. SendMessagePage

撰写短信页面，包括“收件人”输入框、“主题”输入框、“内容”输入框和“发送”按钮，如图 5.13 所示。

SendMessagePage 类的示例代码如下：



图 5.13 SendMessagePage 页面

```

package com.learning selenium.pageobject.normaluse;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;

public class SendMessagePage1 {
    WebDriver driver;
    WebElement toUser;
    WebElement titleContent;
    WebElement textContent;
    WebElement sendButton;

    public SendMessagePage1(WebDriver driver) {
        this.driver = driver;
        toUser = driver.findElement(By.id("txtIncept"));
        titleContent = driver.findElement(By.id("txtTitle"));
        textContent = driver.findElement(By.id("txtContent"));
        sendButton = driver.findElement(By.id("btnSend"));
    }

    public void sendNewMessage(String to, String title,
                               String content) {
        toUser.sendKeys(to);
        titleContent.sendKeys(title);
        textContent.sendKeys(content);
    }
}

```



```
sendButton.click();
}
}
```

示例代码详解：

1) 在类中定义了 4 个 WebElement 属性，用于记录“收件人”输入框、“主题”输入框、“内容”输入框和“发送”按钮。

2) 定义了 SendMessagePage.sendNewMessage() 方法，用于发送新短消息。

5. ReceivedMessagePage

收件箱页面如图 5.14 所示，包括“选中所有”复选框、“删除选中的短消息”按钮。其中删除所有消息的待确认弹出框如图 5.15 所示。

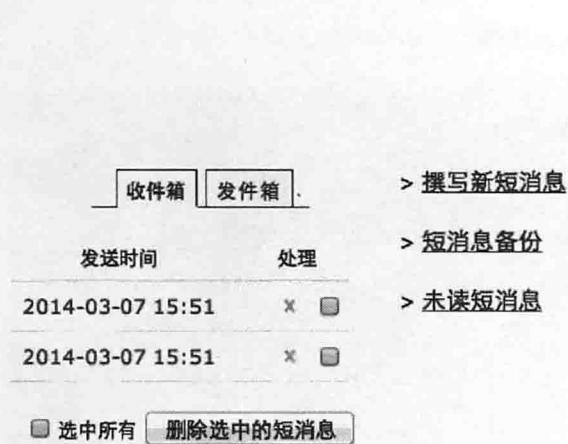


图 5.14 ReceivedMessagePage 页面

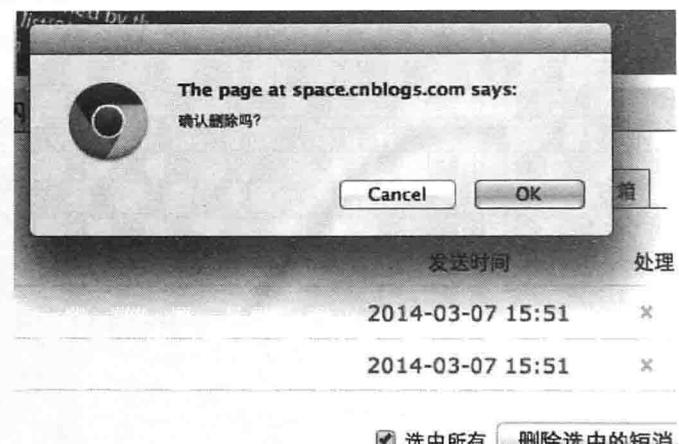


图 5.15 ReceivedMessagePage 中删除所有消息的待确认弹出框

ReceivedMessagePage 类的示例代码如下：

```
package com.learningseleium.pageobject.normaluse;

import org.openqa.selenium.Alert;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

public class ReceivedMessagePage1 {
  WebDriver driver;
  WebElement deleteAllMessagesButton;
  WebElement checkBoxSelectAll;
```

```

public ReceivedMessagePage1( WebDriver driver ) {
    this. driver = driver;
    checkBoxSelectAll = driver. findElement( By. id("chkSelAll") );
    deleteAllMessagesButton =
        driver. findElement( By. id("btnBatDel") );
}

public void deleteAllMessages( ) {
    //check box
    if( ! checkBoxSelectAll. isSelected( ) ) {
        checkBoxSelectAll. click( );
    }

    //delete all received messages
    deleteAllMessagesButton. click( );
    WebDriverWait wait = new WebDriverWait( driver, 300 );
    wait. until( ExpectedConditions. alertIsPresent( ) );
    Alert confirmDeletePrompt = driver. switchTo(). alert();
    confirmDeletePrompt. accept( );
}
}

```

示例代码详解：

- 1) 在类中定义了两个 WebElement 属性，用于记录“选中所有”复选框、“删除选中的短消息”按钮。
- 2) 在构造函数中通过 driver. findElement() 方法查找到相应页面元素。
- 3) 定义删除所有已接收短消息的方法 ReceivedMessagePage. deleteAllMessages()。这里使用了显式等待的方法，以及处理弹出框的方法。

6. MessagePage

短消息页面包括“短消息”链接、“撰写新短消息”链接、“收件箱”链接。MessagePage 类的示例代码如下：

```

package com. learningselenium. pageobject. normaluse;

import java. util. concurrent. TimeUnit;

import org. openqa. selenium. By;
import org. openqa. selenium. WebDriver;
import org. openqa. selenium. WebElement;

```



```
public class MessagePage1 {  
    WebDriver driver;  
    WebElement messageBox;  
    WebElement newMessage;  
    WebElement receivedMessage;  
  
    public MessagePage1( WebDriver driver ) {  
        this. driver = driver;  
    }  
  
    public void enterMessageBox( ) {  
        messageBox = driver.findElement( By.partialLinkText("短消息") );  
        messageBox.click();  
        driver.manage().timeouts().implicitlyWait( 3, TimeUnit.SECONDS );  
    }  
  
    public void sendMessage( String toUser, String titleContent,  
                           String textContent ) {  
        enterMessageBox();  
        newMessage = driver.findElement( By.linkText("撰写新短消息") );  
        newMessage.click();  
        SendMessagePage1 sendMessagePage = new SendMessagePage1( driver );  
        sendMessagePage.sendNewMessage( toUser, titleContent, textContent );  
    }  
  
    public void deleteAllMessage( ) {  
        enterMessageBox();  
        receivedMessage = driver.findElement( By.linkText("收件箱") );  
        receivedMessage.click();  
        ReceivedMessagePage1 receivedMessagePage =  
            new ReceivedMessagePage1( driver );  
        receivedMessagePage.deleteAllMessages();  
    }  
}
```

示例代码详解：

- 1) 在类中定义了三个 WebElement 属性，用于记录“短消息”链接、“撰写新短消息”链接、“收件箱”链接。
- 2) 定义发送短消息的方法 MessagePage. sendMessage()。在其中构造了实例对象 Send-

MessagePage，并调用其发送短消息的方法 SendMessagePage.sendMessage()。

3) 定义删除所有已接收消息的方法 MessagePage.deleteAllMessage()。在其中构造了实例对象 ReceivedMessagePage 并调用其删除所有已接收短消息的方法 ReceivedMessagePage.deleteAllMessages()。

经过上述采用页面对象的方式封装待测试的页面后，最终的完整测试用例代码如下：

```
package com.learningselenium.pageobject.normaluse;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public class testMessageWithPageObject1 {
    public static void main( String... args ) {
        WebDriver driver = new FirefoxDriver();

        MainPage1 mainPage = new MainPage1( driver );
        MessagePage1 messagePage = new MessagePage1( driver );

        mainPage.open MainPage("http://www.cnblogs.com");
        mainPage.login("seleniumpageobject", "pageobject123");
        messagePage.sendMessage("seleniumpageobject",
            "testSendingMessage1",
            "text messages 1");
        messagePage.sendMessage("seleniumpageobject",
            "testSendingMessage2",
            "text messages 2");
        mainPage.logout();

        mainPage.open MainPage("http://www.cnblogs.com");
        mainPage.login("seleniumpageobject", "pageobject123");
        messagePage.sendMessage("seleniumpageobject",
            "testSendingMessage3",
            "text messages 3");
        messagePage.deleteAllMessage();
        mainPage.logout();

        driver.quit();
    }
}
```



示例代码详解：

- 1) 构造 MainPage 的实例对象。
- 2) 构造 MessagePage 的实例对象。
- 3) 打开 cnblogs 主页并登录，然后发送两条短消息后退出登录状态。
- 4) 再次打开 cnblogs 主页并登录，然后发送一条短消息并删除所有已接收短消息，接着退出登录状态。

可以看到，在采用 Page Object 方式的面向对象机制后，测试用例代码的逻辑更加清晰易懂。

★5.4.3 使用 Page Object、Page Factory、@FindBy 和 How

读者可能会问，虽然 Page Object 的思路很不错，自己也会使用面向对象编程方式来进行思考。可这与 WebDriver 有什么关系呢？接下来要介绍的 Page Factory，@FindBy 和 How 数组就是 WebDriver 专门提供给用户以更好地利用 Page Object 方式。可称其为加强版的 Page Object 机制。

1. LoginPage

看一下使用了@Findby 的 LoginPage 有什么变化。示例代码如下：

```
package com.learning selenium.pageobject.findby.pagefactory;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.How;

public class LoginPage2 {
    WebDriver driver;

    @FindBy(how = How.ID, id = "tbUserName")
    WebElement username;
    @FindBy(how = How.ID, id = "tbPassword")
    WebElement password;
    @FindBy(how = How.ID, id = "btnLogin")
    WebElement loginbutton;

    public LoginPage2(WebDriver driver) {
        this.driver = driver;
    }
}
```

```
* the driver.findElement() methods are removed *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * /  
}  
  
public void login( String userName, String passWord ) {  
    username.sendKeys( userName );  
    password.sendKeys( passWord );  
    loginbutton.click();  
  
    driver.manage().timeouts().implicitlyWait( 3, TimeUnit.SECONDS );  
}  
}
```

示例代码详解：

- 1) 在页面中的任意一个页面元素都可以使用@ FindBy 注解来进行标记。@ FindBy 可用于替换 driver.findElement() 方法的查找机制来定位页面元素。
 - 2) 结合@ FindBy，同时使用 How 数组来替换 By 的作用。此例中包括三个 WebElement，分别是“用户名”、“密码”和“登录”按钮。示例代码段如下：

```
@FindBy( how = How.ID, id = "tbUserName")
WebElement username;
@FindBy( how = How.ID, id = "tbPassword")
WebElement password;
@FindBy( how = How.ID, id = "btnLogin")
WebElement loginbutton;
```

- 3) 通过使用@ FindBy 和 How 数组，替换了 driver.findElement（） 和 By，使得原来的构造函数变得更为简洁。示例代码段如下：

小贴士：How 数组能够支持的页面元素包括以下类型：



```
ID_OR_NAME  
CSS  
NAME  
CLASS_NAME  
LINK_TEXT  
PARTIAL_LINK_TEXT  
TAG_NAME  
XPATH
```

2. LogoutPage

退出页面基本上没有太大变化，因为没有页面元素需要特殊处理。

```
package com.learningselenium.pageobject.findby.pagefactory;  
  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.support.ui.ExpectedConditions;  
import org.openqa.selenium.support.ui.WebDriverWait;  
  
public class LogoutPage2 {  
    WebDriver driver;  
  
    public LogoutPage2(WebDriver driver) {  
        this.driver = driver;  
    }  
  
    public void logout() {  
        WebDriverWait wait = new WebDriverWait(driver, 300);  
        wait.until(ExpectedConditions.alertIsPresent());  
        driver.switchTo().alert().accept();  
    }  
}
```

3. MainPage

主页面由于存在两个页面元素，所以同样使用了@ FindBy 注解和 How 数组来处理。此外，还使用了期待已久的 PageFactory 机制。PageFactory 会替换传统的通过 new 来实例化对象的方式。示例代码如下：

```
package com.learningselenium.pageobject.findby.pagefactory;  
  
import java.util.concurrent.TimeUnit;
```

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.PageFactory;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.How;

public class MainPage2 {
    WebDriver driver;

    @FindBy(how = How.LINK_TEXT, linkText = "登录")
    WebElement loginLink;
    @FindBy(how = How.LINK_TEXT, linkText = "退出")
    WebElement logoutLink;

    public MainPage2(WebDriver driver) {
        this.driver = driver;
    }

    public void openMainPage(String url) {
        driver.get(url);
        driver.manage().timeouts().implicitlyWait(300, TimeUnit.SECONDS);
    }

    public void login(String userName, String passWord) {
        /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
        * the driver.findElement() method is removed *
        /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

        loginLink.click();
        WebDriverWait wait = new WebDriverWait(driver, 300);
        wait.until(ExpectedConditions.
            visibilityOfElementLocated(By.Id("tbUserName")));
    }

    LoginPage2 loginPage =
        PageFactory.initElements(driver, LoginPage2.class);
}
```



示例代码详解：

1) 同样采用@ FindBy 和 How 数组来替换 driver.findElement () 方法。代码段如下：

```
@FindBy( how = How.LINK_TEXT, linkText = "登录")
WebElement loginLink;

@FindBy( how = How.LINK_TEXT, linkText = "退出")
WebElement logoutLink;
```

2) 在 MainPage.logout() 方法中移除了 driver.findElement() 方法。

3) 在 MainPage.logout() 方法中，用 PageFactory.initElements() 方法替换了之前传统的 LogoutPage 的对象实例化方法，并将 driver 示例传递给对象。代码段如下：

```
LogoutPage2 logoutPage =  
    PageFactory.initElements(driver, LogoutPage2.class);
```

4. SendMessagePage

撰写短信页面，包括“收件人”输入框、“主题”输入框、“内容”输入框和“发送”按钮这4个页面元素。同样使用@ FindBy 和 How 数组来重新组织代码。示例代码如下：

```
package com.learningseleium.pageobject.pagefactory;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.How;
```

```

public class SendMessagePage2 {
    WebDriver driver;

    @FindBy( how = How.ID, id = "txtIncept")
    WebElement toUser;
    @FindBy( how = How.ID, id = "txtTitle")
    WebElement titleContent;
    @FindBy( how = How.ID, id = "txtContent")
    WebElement textContent;
    @FindBy( how = How.ID, id = "btnSend")
    WebElement sendButton;

    public SendMessagePage2( WebDriver driver) {
        this.driver = driver;

        /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
        * the driver.findElement( ) methods are removed *
        /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
    }

    public void sendNewMessage( String to, String title,
                               String content) {
        toUser.sendKeys( to);
        titleContent.sendKeys( title);
        textContent.sendKeys( content);
        sendButton.click();
    }
}

```

示例代码详解：

1) 在类中定义了 4 个 WebElement 元素，因此调整后的代码段如下：

```

@FindBy( how = How.ID, id = "txtIncept")
WebElement toUser;
@FindBy( how = How.ID, id = "txtTitle")
WebElement titleContent;
@FindBy( how = How.ID, id = "txtContent")
WebElement textContent;
@FindBy( how = How.ID, id = "btnSend")
WebElement sendButton;

```





2) 在其构造函数中，省略了 driver.findElement() 方法。代码段如下：

```
public SendMessagePage2( WebDriver driver ) {
    this.driver = driver;

    /**
     * the driver.findElement( ) methods are removed
     */
}
```

5. ReceivedMessagePage

收件箱页面包括“选中所有”复选框、“删除选中的短消息”按钮。经过@ FindBy 和 How 数组处理后的代码如下：

```
package com.learningseleium.pageobject.findby.pagefactory;

import org.openqa.selenium.Alert;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.How;

public class ReceivedMessagePage2 {

    WebDriver driver;

    @FindBy(how = How.ID, id = "chkSelAll")
    WebElement checkBoxSelectAll;

    @FindBy(how = How.ID, id = "btnBatDel")
    WebElement deleteAllMessagesButton;

    public ReceivedMessagePage2(WebDriver driver) {
        this.driver = driver;

        /**
         * the driver.findElement() methods are removed
         */
    }
}
```





```
public void deleteAllMessages() {
    //check box
    if( ! checkBoxSelectAll.isSelected( ) ) {
        checkBoxSelectAll.click( );
    }

    //delete all received messages
    deleteAllMessagesButton.click( );
    WebDriverWait wait = new WebDriverWait( driver, 300 );
    wait.until( ExpectedConditions.alertIsPresent( ) );
    Alert confirmDeletePrompt = driver.switchTo().alert();
    confirmDeletePrompt.accept();
}
```

示例代码详解：

1) 在类中定义了两个 WebElement 元素，因此调整后的代码段如下：

```
@FindBy( how = How.ID, id = "chkSelAll")
WebElement checkBoxSelectAll;
@FindBy( how = How.ID, id = "btnBatDel")
WebElement deleteAllMessagesButton;
```

2) 在其构造函数中，省略了 driver.findElement() 方法。代码段如下：

6. MessagePage

短消息页面包括“短消息”链接、“撰写新短消息”链接和“收件箱”链接。MessagePage 类经过@ FindBy、How 数组和 Page Factory 改造后的示例代码如下：

```
package com.learningseleium.pageobject.findby.pagefactory;  
  
import java.util.concurrent.TimeUnit;
```



```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.How;
import org.openqa.selenium.support.PageFactory;

public class MessagePage2 {
    WebDriver driver;

    @FindBy(how = How.PARTIAL_LINK_TEXT, partialLinkText = "短消息")
    WebElement messageBox;
    @FindBy(how = How.LINK_TEXT, linkText = "撰写新短消息")
    WebElement newMessage;
    @FindBy(how = How.LINK_TEXT, linkText = "收件箱")
    WebElement receivedMessage;

    public MessagePage2(WebDriver driver) {
        this.driver = driver;
    }

    public void enterMessageBox() {
        /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
         * the driver.findElement() method is removed *
         * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
        messageBox.click();
        driver.manage().timeouts().implicitlyWait(3, TimeUnit.SECONDS);
    }

    public void sendMessage(String toUser, String titleContent,
                          String textContent) {
        enterMessageBox();

        /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
         * the driver.findElement() method is removed *
         * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
        newMessage.click();
    }
}
```



```
SendMessagePage2 sendMessagePage =
    PageFactory.initElements(driver, SendMessagePage2.class);
sendMessagePage.sendMessage(toUser, titleContent, textContent);
}

public void deleteAllMessage() {
    enterMessageBox();

    /**
     * the driver.findElement() methods is removed
     */
    receivedMessage.click();
}

ReceivedMessagePage2 receivedMessagePage =
    PageFactory.initElements(driver, ReceivedMessagePage2.class);
receivedMessagePage.deleteAllMessages();
}
```

示例代码详解：

1) 在类中定义了三个 WebElement 元素，因此调整后的代码段如下：

```
@FindBy(how = How.PARTIAL_LINK_TEXT, partialLinkText = "短消息")
WebElement messageBox;
@FindBy(how = How.LINK_TEXT, linkText = "撰写新短消息")
WebElement newMessage;
@FindBy(how = How.LINK_TEXT, linkText = "收件箱")
WebElement receivedMessage;
```

2) 在构造函数中去掉了 driver.findElement() 方法。代码段如下：

```
public void enterMessageBox() {
    /**
     * the driver.findElement() method is removed
     */
    messageBox.click();
    driver.manage().timeouts().implicitlyWait(3, TimeUnit.SECONDS);
```





3) 在 MessagePage.sendMessage() 方法中去掉了 driver.findElement() 方法，并且用 PageFactory.initElements() 替换了 SendMessagePage 对象实例化的代码段：

```
public void sendMessage( String toUser, String titleContent,
    String textContent ) {
    enterMessageBox();

    /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
    * the driver.findElement() method is removed *
    * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

    newMessage.click();
    SendMessagePage2 sendMessagePage =
        PageFactory.initElements( driver, SendMessagePage2.class );
    sendMessagePage.sendNewMessage( toUser, titleContent, textContent );
}
```

4) 在 MessagePage.deleteAllMessage() 方法中去掉了 driver.findElement() 方法，并且用 PageFactory.initElements() 替换了 ReceivedMessagePage 对象实例化的代码段：

```
public void deleteAllMessage() {
    enterMessageBox();

    /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
    * the driver.findElement() methods is removed *
    * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

    receivedMessage.click();
    ReceivedMessagePage2 receivedMessagePage =
        PageFactory.initElements( driver, ReceivedMessagePage2.class );
    receivedMessagePage.deleteAllMessages();
}
```

经过 Page Object、@FindBy、How 数组和 Page Factory 改造后的最终测试用例代码如下。唯一的变化是 MainPage 对象的实例化和 MessagePage 对象的实例化由 PageFactory.initElements() 方法处理掉了。

```
package com.learningselenium.pageobject.findby.pagefactory;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.PageFactory;

public class testMessageWithPageObject2 {
    public static void main(String... args) {
        WebDriver driver = new FirefoxDriver();

        MainPage2 mainPage =
            PageFactory.initElements(driver, MainPage2.class);
        MessagePage2 messagePage =
            PageFactory.initElements(driver, MessagePage2.class);

        mainPage.openMainPage("http://www.cnblogs.com");
        mainPage.login("seleniumpageobject", "pageobject123");
        messagePage.sendMessage("seleniumpageobject",
            "testSendingMessage1",
            "text messages 1");
        messagePage.sendMessage("seleniumpageobject",
            "testSendingMessage2",
            "text messages 2");
        mainPage.logout();

        mainPage.openMainPage("http://www.cnblogs.com");
        mainPage.login("seleniumpageobject", "pageobject123");
        messagePage.sendMessage("seleniumpageobject",
            "testSendingMessage3",
            "text messages 3");
        messagePage.deleteAllMessage();
        mainPage.logout();

        driver.quit();
    }
}
```

综上所述，采用 Page Object、@ FindBy、How 数组和 Page Factory 可以使得测试代码的逻辑更为清晰，维护起来也更加方便。



5.5 Selenium RC 迁移到 WebDriver

★5.5.1 简介

正如第1章所述，在WebDriver结合Selenium并孕育出Selenium 2之前，Selenium RC在相当长的一段时间里都是作为Selenium的“主力选手”上场。为了与Selenium 2进行区别，通常称Selenium RC时代的Selenium为Selenium 1。截至目前，Selenium 1还继续被维护并提供一些Selenium 2尚不支持的功能，如某些编程语言的支持和某些浏览器的支持。

虽然Selenium RC已经无法阻止历史的演进，作为即将完全退休的“老干部”，其仍旧散发着余热。某些使用Selenium多年的公司和机构，需要继续维护基于Selenium RC的测试集合，并继续添加新的测试代码；还有一部分使用Selenium多年的公司和机构，他们尝试将已有的基于Selenium RC的测试代码迁移到WebDriver上来，让那些老旧但成熟的代码和测试集合重新焕发新生。

本章将向用户展示如何从已有的测试代码迁移到新的API，并基于WebDriver的特性来添加新的测试代码。

★5.5.2 从Selenium RC迁移到WebDriver的优势

想要说服项目管理者对已有的代码进行重构需要很大的勇气和推动力，因为项目管理者往往关心的是项目的开发进度而非代码质量本身。此外，从技术层面而言，将测试集合从一套API迁移到另外一套API需要投入大量的工作。既然如此，为什么在这里依旧推荐大家从已有的Selenium RC迁移到WebDriver上来呢？看看WebDriver强大的优势所在：

- 1) API更紧凑，集合更小。WebDriver的API所提供的编程方式比原有的Selenium RC的API更加面向对象。
- 2) 对用户的交互行为模拟得更为真实。WebDriver会尽可能地使用浏览器的原生事件去处理页面交互。此外，WebDriver还提供了更多高级的用户交互式API来完成复杂的交互测试工作。
- 3) 为浏览器厂商提供了更多的自由度。Google、Mozilla等浏览器厂商都在针对自己出品的浏览器积极地开发相对应的WebDriver，这就意味着WebDriver与浏览器逐渐变得密不可分，其稳定性和性能都将得到强有力的保障。

★5.5.3 迁移Selenium运行实例

正所谓万事开头难，从Selenium RC迁移到WebDriver需要规划好执行步骤。第一步要做的是将原有的Selenium运行实例迁移到WebDriver中来。此时，首先需要确保的是之前基于Selenium RC的代码可以在最新的Selenium发行版本上正常运行。

Selenium RC中的Selenium运行实例代码如下：

```
Selenium selenium = new DefaultSelenium(  
    "localhost", 4444, "*firefox", "http://www.yoursite.com");  
selenium.start();
```

迁移到 WebDriver 中，示例代码如下：

```
WebDriver driver = new FirefoxDriver();
Selenium selenium = new WebDriverBackedSelenium(
    driver, "http://www.yoursite.com");
```

一旦完成了第一步，可以尝试将原有的测试代码集合全部运行一次。如果发现有新的 error 出现，请调试代码直到所有新的 error 都被修复。

★5.5.4 迁移测试代码到 WebDriver API

基于前述步骤，确保所有新增 error 都被修复后，便可以开始将测试代码迁移到 WebDriver API 的工作。迁移所有代码的工作量取决于之前的代码质量和抽象程度。

如果希望将 WebDriver 的底层实现从 Selenium 实例中剥离出来，那么可以使用如下方法来达到目的。示例代码段如下：

```
WebDriver driver = ((WrapsDriver)selenium).getWrappedDriver();
```

通过 WrapsDriver 这种方式，可以保证原有的 Selenium 实例能正常使用，同时确保 WebDriver 的实例在需要时能被使用。

某些时候，我们会希望原有的 RC 语法能继续工作，又能同时利用 WebDriver 的强大优势。Selenium Core 开发团队便为我们提供了 WebDriverBackedSelenium 这个对象来完成这个功能。它允许我们在继续沿用 RC 编程语法的同时，以付出最小的修改代价就能利用 WebDriver 的功能。示例代码段如下：

```
Selenium selenium = new WebDriverBackedSelenium(driver, baseUrl);
```

WebDriverBackedSelenium 对象将 RC 的 API 映射到 WebDriver 的 API。虽然有一小部分 RC 的功能不能被 WebDriverBackedSelenium 支持，但 WebDriverBackedSelenium 已经能够保证绝大部分基于 RC 的旧代码在不被修改的情况下就可以使用 WebDriver 的新功能，迁移的工作量将大大减少。

5.6 小结

本章介绍了 WebDriver 对 HTML5 特性的支持，包括 Video、Canvas、Drag 和 Drop 等。接着对 RemoteWebDriver 的优缺点分别进行了分析。最后对于已经在使用 Selenium RC 的机构如何迁移已有代码到 Selenium WebDriver 给出了基本的解决方案。

第 6 章

Selenium 玩转 Android

6.1 简介

Android 是一种基于 Linux 的自由和开放源代码的操作系统，主要用于移动设备，如智能手机和平板电脑，由 Google 公司和开放手机联盟领导及开发。Android 操作系统最初由 Andy Rubin 的公司开发，主要支持手机。2005 年 8 月由 Google 公司收购注资。2007 年 11 月，Google 公司与 84 家硬件制造商、软件开发商和电信营运商组建开放手机联盟，共同研发改良 Android 系统。随后 Google 公司以 Apache 开源许可证的授权方式，发布了 Android 的源代码。第一部 Android 智能手机发布于 2008 年 10 月。现在，Android 已逐渐扩展到平板电脑和其他领域上，如电视、数码相机、游戏机等。

6.2 玩转 Android

★6.2.1 架构

Android WebDriver 的架构如图 6.1 所示。在手机上 Android WebDriver 被打包成一个 app，然后安装到模拟器或者真机上。该 app 内嵌了 RemoteWebDriver Server，以及一个轻量级的 HTTP Server，用于接收并响应从 WebDriver Client 发送过来的自动化测试请求。

RemoteWebDriver Server 主要基于 WebView（WebView 是 Android 上针对 Web 应用的渲染组件）来运行测试用例。而 WebDriver Client 则主要负责运行测试用例，可以用任意 Selenium 支持的绑定语言来编写，如 Java、Python 等。RemoteWebDriver Server 与 WebDriver Client 之间基于 REST JSON 请求并通过 HTTP 进行交互，并且两者既可以存在于同一个物理设备上，也可以分布在不同设备上。针对 Android，可以直接使用 Android 自带的测试框架来取代 RemoteWebDriver Server。

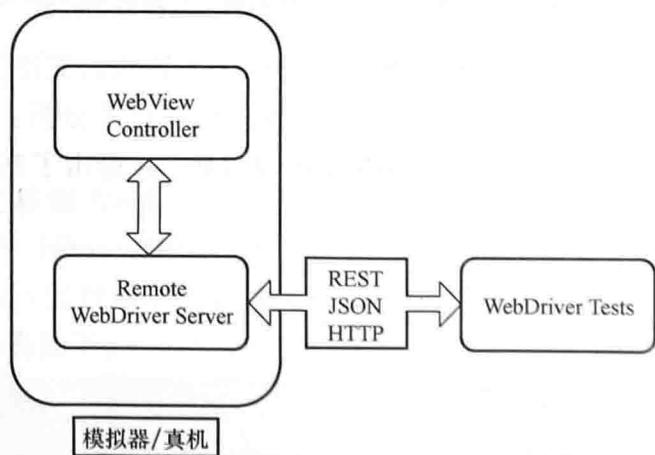


图 6.1 Android WebDriver 的架构

★6.2.2 搭建Android WebDriver环境

本章中使用的Android SDK版本为adt-bundle-mac-x86_64-20130917，Android Server版本为2.32.0。

1. 安装Android SDK

到如下网址下载Android SDK：

```
http://developer.android.com/sdk/index.html
```

2. 创建Android虚拟设备

解包刚下载的SDK压缩包，并进入tools目录，然后启动AVD管理器（Android Virtual Device Manager）。

```
$ cd adt-bundle-mac-x86_64-20130917/sdk/tools  
$ ./android avd
```

3. 新建一个Android虚拟设备

在AVD管理器上单击New新建一个Android虚拟设备，如图6.2所示。

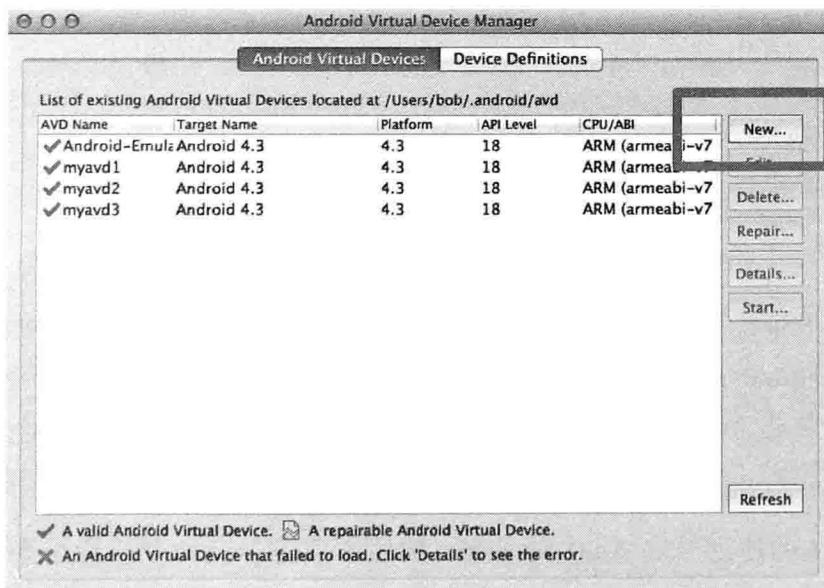


图6.2 AVD管理器

新的Android虚拟设备创建成功后，就可以启动它了。具体步骤请参考Android开发者官网。

4. 设置Android虚拟设备

有两种方式来运行Android WebDriver。一种是基于RemoteWebDriver Server，还有一种是基于Android Test Framework。它们的优缺点分别如表6.1所示。

本书将着重介绍RemoteWebDriver Server方式的使用，此方式的使用包括服务器端和客户端两部分。

1) 客户端：一般指使用jUnit或者TestNG组织的测试用例代码，运行的时候可以选择直接在IDE中运行，也可以通过命令行的方式运行。



表 6.1 RemoteWebDriver Server 与 Android Test Framework 的优缺点对比

RemoteWebDriver Server	Android Test Framework
可用任意 Selenium 支持绑定的编程语言	只能使用 Java 编程语言
运行速度慢, 因为每个命令都是基于 HTTP 的 RPC 方式	运行速度快, 因为测试直接在模拟器或者真实设备上运行
可移植性更强, 同一套代码测试多种不同浏览器	已经在使用 Android Test Framework 并且不打算用同一套代码测试其他浏览器

2) 服务器端: 一般指运行在 Android 设备上, 并且包含了 HTTP 服务器的应用程序。

在运行测试用例时, 客户端的每个 WebDriver 命令都会发送一个采用 JSON 协议的 RESTful HTTP 请求到服务器端。而远端的 HTTP 服务器会将客户端发送过来的请求转发给 Android WebDriver, 并将请求结果发送回客户端。具体的 JSON 协议规范可参考官方网址:

<http://code.google.com/p/selenium/wiki/JsonWireProtocol>

5. 安装并设置 Android WebDriver Server

1) 无论 Android 模拟器还是 Android 真实设备, 都会有一个序列号, 即 serial ID。通过如下命令先获取模拟器或者设备的序列号, 此例中序列号为 f14c451c:

```
$ cd adt-bundle-mac-x86_64-20130917/sdk/platform-tools/
$ ./adb devices
List of devices attached
f14c451c    device
```

2) 下载 Android server 的 apk 文件。下载地址为

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

3) 在安装 android-server.apk 文件到 Android 模拟器或者真实设备上之前, 应确保 Android 系统允许安装非 Android 官方市场下载的应用程序。安装命令如下:

```
$ ./adb -s <serialId> -e install -r android-server.apk
```

结合本例具体的序列号和 Android Server 的版本而言, 其命令执行结果如下:

```
$ ./adb -s f14c451c -e install -r android-server-2.32.0.apk
2950 KB/s (1592370 bytes in 0.527s)
pkg: /data/local/tmp/android-server-2.32.0.apk
Success
```

4) 有两种方式启动 Android WebDriver 应用程序。一种是从 Android 模拟器或者设备的 UI 上启动, 如图 6.3 所示。

5) 还有一种是从命令行启动, 如下:

```
$ ./adb -s <serialId> shell am start -a android.intent.action.MAIN -n
org.openqa.selenium.android.app/.MainActivity
```

结合本例具体的序列号而言, 其命令执行结果如下:



图 6.3 从 Android UI 上启动 WebDriver Server

```
$ ./adb -s f14c451c shell am start -a android.intent.action.MAIN -n org.openqa.selenium.android.app/.MainActivity
Starting: Intent { act = android.intent.action.MAIN cmp = org.openqa.selenium.android.app/.MainActivity }
```

6) 如果需要进入 debug 模式, 可以打开此选项。命令如下:

```
$ ./adb -s <serialId> shell am start -a android.intent.action.MAIN -n org.openqa.selenium.android.app/.MainActivity -e debug true
```

结合本例具体的序列号而言, 其命令执行结果如下:

```
$ ./adb -s f14c451c shell am start -a android.intent.action.MAIN -n org.openqa.selenium.android.app/.MainActivity -e debug true
Starting: Intent { act = android.intent.action.MAIN cmp = org.openqa.selenium.android.app/.MainActivity (has extras) }
```

7) 设置端口转发规则:

```
$ ./adb -s <serialId> forward tcp:8080 tcp:8080
```

结合本例具体的序列号而言, 设定本地的端口为 8888, 其命令执行结果如下:

```
$ ./adb -s f14c451c forward tcp:8888 tcp:8080
```

这样就完成了 Android WebDriver Server 的安装和设置, 接着就可以从本机的如下地址来查看测试代码与 Android WebDriver Server 进行交互的状态:

<http://localhost:8888/wd/hub/status>



以下方式可检查确认 Android WebDriver Server 安装并设置正确：

- 1) 在浏览器中查看端口转发是否设置成功，正常会显示 {status: 0}，如图 6.4 所示。



图 6.4 在浏览器中查看端口转发设置状态

- 2) 使用 telnet 命令：

```
$ telnet localhost 8888
```

- 3) 使用 curl 命令：

```
$ curl http://localhost:8888/wd/hub/status
```

- 4) 使用 wget 命令：

```
$ wget http://localhost:8888/wd/hub/status
```

接下来就可以运行测试用例。在此有几个地方需要注意：

- 1) 确保 Android 手机上“开发人员选项”中的“USB 调试”、“保持唤醒状态”和“允许模拟地点”处于打开状态，如图 6.5 所示。
- 2) 确保 Android 手机上的“USB 数据存储”处于关闭状态，如图 6.6 所示。否则会导致执行测试用例时出现如下不可操作 sdcard 的错误提示：

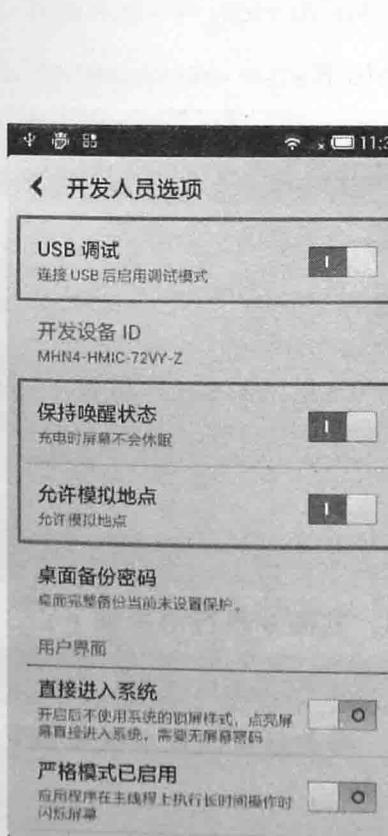


图 6.5 设置 Android 手机的“开发人员选项”

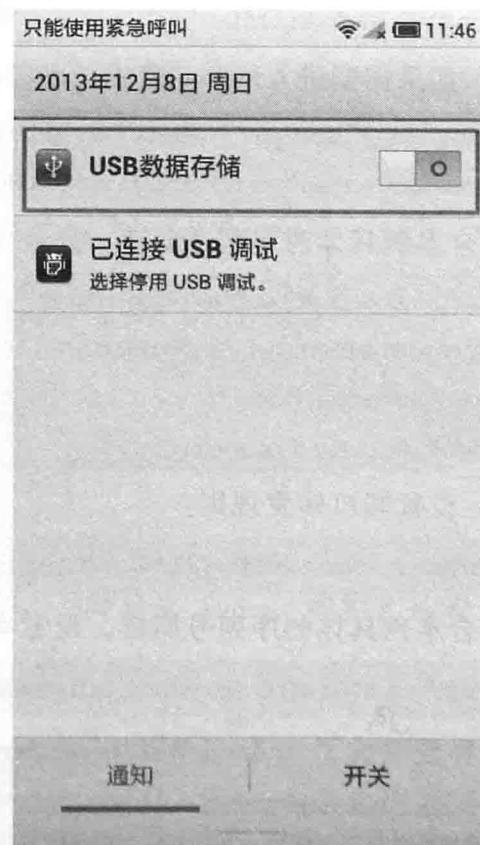


图 6.6 设置 Android 手机的“USB 数据存储”

```
org.openqa.selenium.WebDriverException:  
cannot create temp directory: /sdcard/1339054625829
```

★6.2.3 最简单的测试用例

下面以一个最简单的测试用例来阐述如何使用代码与 Android WebDriver 进行交互。示例代码如下：

```
package com.learingselenium.android;  
  
import junit.framework.TestCase;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.android.AndroidDriver;  
  
public class testAndroidBaidu extends TestCase {  
  
    public void testBaidu() throws Exception {  
  
        WebDriver driver = new AndroidDriver(  
                "http://localhost:8888/wd/hub");  
  
        driver.get("http://www.baidu.com");  
  
        String url = driver.getCurrentUrl();  
  
        System.out.println(url);  
  
        driver.close();  
    }  
}
```

示例代码详解：

1) 启动 Android WebDriver，并指定消息的转发地址为如下所示。因为在 6.2.2 节中将本地的端口转发设定在 8888 端口。

```
http://localhost:8888/wd/hub
```

- 2) 在 Android 设备上的 WebDriver 中会打开百度主页，如图 6.7 所示。
- 3) 在 Eclipse 的 Console 中打印当前 URL 的地址，如图 6.8 所示。

★6.2.4 旋转屏幕

有别于普通 PC 上的 WebDriver，作为嵌入式设备上的 Android WebDriver，其特有的功能就包括屏幕的旋转操作。示例代码如下：





地图 贴吧 视频 图片 hao123

新闻 应用 音乐 文库 更多

小说 游戏 下载

把百度放到桌面上，搜索最方便

触屏版|电脑版

Baidu 京ICP证030173号

图 6.7 Android WebDriver 打开
百度主页



图 6.8 在 Eclipse 的 Console 中打印当前 URL 地址

```
package com.learningselenium.android;

import junit.framework.TestCase;

import org.openqa.selenium.Rotatable;
import org.openqa.selenium.ScreenOrientation;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.android.AndroidDriver;

public class testAndroidBaiduLandscape extends TestCase {
    public void testBaidu() throws Exception {
        WebDriver driver = new AndroidDriver(
                "http://localhost:8888/wd/hub");
        ((Rotatable) driver).rotate(ScreenOrientation.LANDSCAPE);

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

        driver.close();
    }
}
```

示例代码详解：

与前述“最简单的测试用例”最大的不同在于对于屏幕旋转的设定代码段：

```
((Rotatable) driver).rotate(ScreenOrientation.LANDSCAPE);
```

其执行结果如图 6.9 所示，相较于前述示例，此时屏幕旋转了 90°。



图 6.9 Android WebDriver 的屏幕旋转效果

★6.2.5 触摸和滚动

除了屏幕的旋转，针对 Android 设备的特有操作还包括触摸和滚动。以国内 IT 新闻网站 CNBeta 的移动版本为例进行讲解，其主页面如图 6.10 所示。



图 6.10 Android WebDriver 打开 CNBeta 主页



示例代码如下：

```
package com.learngselenium.android;

import junit.framework.TestCase;
import org.openqa.selenium.Rotatable;
import org.openqa.selenium.ScreenOrientation;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.android.AndroidDriver;
import org.openqa.selenium.interactions.touch.TouchActions;

public class testAndroidCnBetaTouchAndScroll extends TestCase {

    public void testBaidu() throws Exception {
        WebDriver driver = new AndroidDriver(
            "http://localhost:8888/wd/hub");

        driver.get("http://m.cnbeta.com");

        ((Rotatable) driver).rotate(ScreenOrientation.PORTRAIT);

        TouchActions touch = new TouchActions(driver);

        touch.scroll(0, 400).build().perform();

        driver.close();
    }
}
```

示例代码详解：

- 1) 以 Android WebDriver 来驱动 WebView 打开 CNBeta 主页面。
- 2) 确保 WebView 的屏幕方向为 PORTRAIT。示例代码段如下：

```
((Rotatable) driver).rotate(ScreenOrientation.PORTRAIT);
```

- 3) 采用 TouchActions 操作页面的滚动，向下滚动 400 个像素偏移量。示例代码段如下：

```
TouchActions touch = new TouchActions(driver);
touch.scroll(0, 400).build().perform();
```

6.3 当Android遇到HTML5

★6.3.1 HTML5中的Web Storage

在HTML5中，Web Storage这个新特性可以让用户将数据存储在本地的浏览器中。在早期的浏览器中可以通过cookies来完成这个任务。但是相较而言，Web Storage会更加安全和高效。并且Web Storage通过键值对进行存储，只有自身的网页应用才可以访问存储的数据。

Web Storage分为以下两种类型：

- 1) localStorage：存储的数据永久不会过期。
- 2) sessionStorage：存储数据只在当前会话中有效。

通过Google Chrome的开发者工具在Resources选项卡中可以查看LocalStorage和SessionStorage的信息，如图6.11所示。

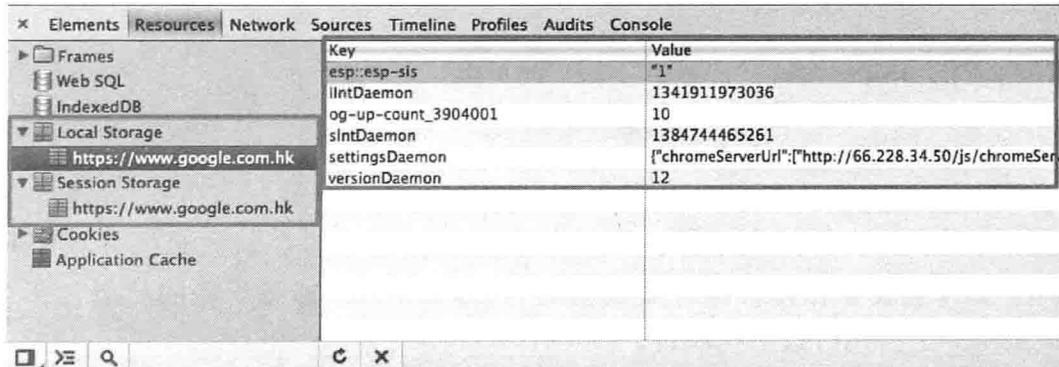


图6.11 查看LocalStorage和SessionStorage

以Android上的HTML5浏览器为例，操作LocalStorage的示例代码如下：

```
package com.learingselenium.android;

import static org.junit.Assert.assertEquals;

import org.junit.*;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.html5.LocalStorage;
import org.openqa.selenium.html5.WebStorage;
import org.openqa.selenium.android.AndroidDriver;

public class testAndroidHTML5LocalStorage {
    @Test
    public void testHTML5LocalStorage() throws Exception {
        WebDriver driver = new AndroidDriver(
            "http://127.0.0.1:4723",
            new AppiumDesiredCapabilities()
                .setPlatformVersion("4.4")
                .setPlatformName("Android")
                .setAutomationName("Appium")
                .setApp("file:///android_asset/index.html")
        );
        LocalStorage ls = (LocalStorage) driver.manage().localStorage();
        ls.setItem("key", "value");
        assertEquals("value", ls.getItem("key"));
    }
}
```



```
"http://localhost:8888/wd/hub");

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

LocalStorage localStorage =
    ((WebStorage) driver).getLocalStorage();

System.out.println("The size of LocalStorage is : "
    + localStorage.size());

localStorage.setItem("key1", "learningselenium");

System.out.println(localStorage.getItem("key1"));

driver.quit();
}
}
```

从图 6.12 所示的执行结果来看，LocalStorage 会保留之前存储的数据并且 LocalStorage 的长度为 6。如果需要保证每次运行测试用例时 LocalStorage 都是干净的状态，那么需要在使用 LocalStorage 之前执行以下代码段：

```
LocalStorage localStorage = ((WebStorage) driver).getLocalStorage();
localStorage.clear();
```

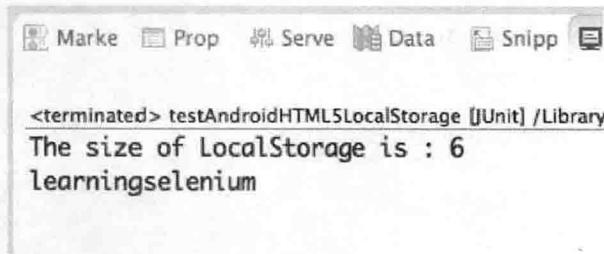


图 6.12 LocalStorage 测试用例的执行结果

以 Android 上的 HTML5 浏览器为例，操作 SessionStorage 的示例代码如下：

```
package com.learningselenium.android;

import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.android.AndroidDriver;
import org.openqa.selenium.html5.SessionStorage;
import org.openqa.selenium.html5.WebStorage;
```

```

public class testAndroidHTML5SessionStorage {
    @Test
    public void testHTML5SessionStorage() throws Exception {
        WebDriver driver = new AndroidDriver(
            "http://localhost:8888/wd/hub");
        driver.get("http://www.baidu.com");

        SessionStorage sessionStorage =
            ((WebStorage) driver).getSessionStorage();

        System.out.println("The size of SessionStorage is : "
            + sessionStorage.size());
        sessionStorage.setItem("key1", "learningselenium");

        System.out.println(sessionStorage.getItem("key1"));

        driver.quit();
    }
}

```

从图 6.13 所示的执行结果来看，SessionStorage 不会保留前一次 session 所存储的数据，即 sessionStorage 中存储的数据只在当前 session 中有效。



图 6.13 SessionStorage 测试用例的执行结果

★6.3.2 HTML5 中的 Application Cache

HTML5 中引入了 Application Cache，这意味着 Web 应用程序可以被缓存到本地，并且可以在没有网络的情况下也能访问该 Web 应用程序。

Application Cache 在以下三个方面具有明显优势：

- 1) 离线浏览：在没有网络的情况下用户也可以使用 Web 应用程序。
- 2) 速度快：缓存的资源被加载时速度很快。



3) 服务器负载小：浏览器只会从服务器更新有变化或新增的资源。

确认 Web 应用程序是否使用了该缓存特性的最简单方式就是直接查看网页 HTML 源码，如果其源码中具有如下包括 manifest 属性的标签则说明使用了 Application Cache 特性。

```
<!DOCTYPE HTML>
<html manifest ="demo.appcache">
...
</html>
```

Selenium WebDriver 具有一个名为 AppCacheStatus 的枚举量，用于标记当前的 Application Cache 的状态，包括 0 (UNCACHED)、1 (IDLE)、2 (CHECKING)、3 (DOWNLOADING)、4 (UPDATEREADY)、5 (OBSOLETE)。

下面以获取当前 Web 应用程序的缓存状态为例来展示 WebDriver 针对 Application Cache 的接口如何使用。示例代码如下：

```
package com.learnselenium.android;

import static org.junit.Assert.assertEquals;

import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.android.AndroidDriver;
import org.openqa.selenium.html5.AppCacheStatus;
import org.openqa.selenium.html5.ApplicationCache;

public class testAndroidHTML5AppCache {

    @Test
    public void testHTML5LocalStorage() throws Exception {
        WebDriver driver = new AndroidDriver(
                "http://localhost:8888/wd/hub");

        driver.get(
                "http://www.w3schools.com/html/tryhtml5_html_manifest.htm");

        AppCacheStatus status = ((ApplicationCache) driver).getStatus();

        assertEquals (status, AppCacheStatus.DOWNLOADING);

        System.out.println ("Application Cache's status is : "
                + status.toString ());
    }
}
```

```

        driver.quit();
    }
}

```

示例代码详解：

- 启动 AndroidDriver，并且让 Android 设备上的 WebView 打开如下页面：

```
http://www.w3schools.com/html/tryhtml5\_html\_manifest.htm
```

- 获取 Application Cache 的状态，在本例中为 DOWNLOADING 状态。示例代码段如下：

```
AppCacheStatus status = ((ApplicationCache)driver).getStatus();
```

- 验证并打印 Application Cache 的状态，执行结果如图 6.14 所示。

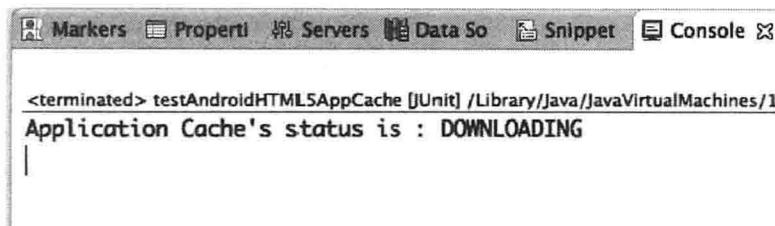


图 6.14 Application Cache 测试用例的执行结果

6.4 在 Cloud 中测试 Android

迎着云计算普及的春风，可以将嵌入式设备放到云计算平台中来完成自动化测试工作。其优势在于资源的合理调配，并且能够尽可能多地解决测试多种不同操作系统类型或者不同尺寸屏幕的兼容性问题。特别是现在热门的 Android 平台和 iOS 平台，它们都有相应的模拟器程序，使得在云中进行测试变得可行。

首先，如果希望在云中能够针对手机真机进行测试，那么真机实验室是不可或缺的。这一点似乎非常明显，无须赘述。在这里将分析 Android 模拟器的特点，供大家参考。

Android 模拟器可以在多个不同的操作系统上运行，包括 Windows、Mac、Linux，因此搭建基于 Android 模拟器的云计算平台可以选择任意熟悉的操作系统平台。虽然 Android 模拟器可以带来极为完整的测试体验，包括虚拟的 CPU、MMU，以及其他硬件设备，但不能高兴得太早。因为 Android 模拟器的运行速度实在不敢恭维。因此，为了能在云计算平台上真正做到使用 Android 模拟器来执行并高效地完成测试任务，需要将模拟器中的动画、音频、皮肤等与功能测试本身不相关的附属特性都关闭，甚至让模拟器运行在 headless 模式下都可以有效地提高模拟器的运行效率。具体的操作步骤是在启动模拟器时，附带如下参数：

```
$ ./emulator -no-boot-anim -no-audio -no-skin -no-window
```

除了以上通过禁止某些特性的方式来加速运行效率，还可以通过使用 Snapshot（快照）镜像来启动模拟器的方式来减少模拟器启动时间，从之前的 2min 减少到不超过 3s。这太疯狂了！在创建 AVD（Android Virtual Device）的时候，将 Emulation Options 中的 Snapshot 选项勾选上，如图 6.15 所示。

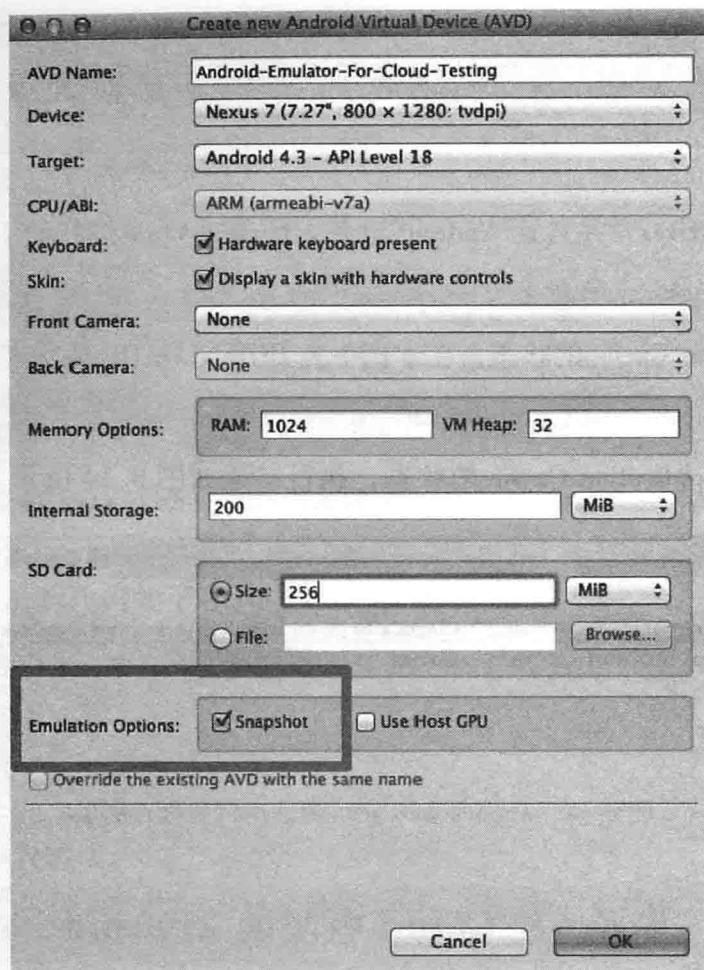


图 6.15 创建附带 Snapshot 的 AVD

接下来启动这个被创建的 AVD，并且在 Launch Options 页面上，将 Launch from snapshot 和 Save to snapshot 都勾选上，如图 6.16 所示。

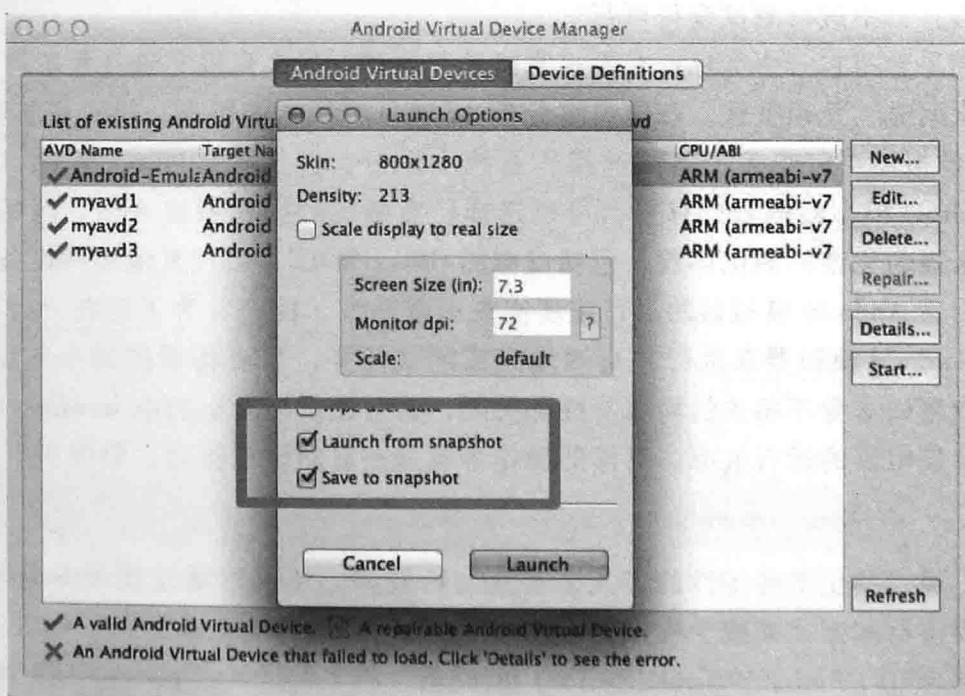


图 6.16 启动 AVD 时附带 Snapshot 选项

在 AVD 第一次启动时,由于还没有生成快照镜像,因此这次启动效率还是维持在一般水平。当系统启动后,单击关闭按钮来保存一次快照镜像并退出模拟器。生成快照镜像的时间会略显漫长,但这种等待是值得的。一旦快照镜像生成,那么下次启动速度将是飞一般的感觉。原因很简单,因为直接跳过了原来的系统启动时间。

接下来就可以通过模拟器的命令行方式启动镜像。命令如下:

```
$ ./emulator -snapshot <name>
```

更多关于快照镜像的操作步骤和注意事项可以参考:

```
http://tools.android.com/recent/emulatorsnapshots
```

结合前面所述的优化参数和快照模式,基于 Android 模拟器的云计算测试平台,较为推荐的模拟器启动参数组合为

```
$ ./emulator -no-boot-anim -no-audio -no-skin -no-window -snapshot <name> -avd <name>
```

小贴士

1. Android 模拟器启动很慢怎么办?

可以参考对于模拟器启动的优化部分。唯一需要注意的是,如果模拟器启用了-no-window 参数,那么只能通过命令行的方式来启动模拟器中的 Android WebDriver App 了。

2. adb 提示找不到设备或模拟器怎么办?

```
$ adb kill-server
```

```
$ adb start-server
```

3. 如果地理定位没起作用怎么办?

确保 Android 手机上“开发人员选项”中的“USB 调试”、“保持唤醒状态”以及“允许模拟地点”处于打开状态。

4. 加载 HTTPS 页面失败怎么办?

请添加如下代码段来解决加载 HTTPS 页面失败的情况:

```
DesiredCapabilities caps = DesiredCapabilities.android();
caps.setCapability(CapabilityType.ACCEPT_SSL_CERTS, true);
```

```
AndroidDriver driver = new AndroidDriver(caps);
```

5. 安装 android-server.apk 提示证书不一致怎么办?

```
Failure [INSTALL_PARSE_FAILED_INCONSISTENT_CERTIFICATES]
```

安装时如果遇到以上提示,则说明 apk 所签名的证书与已经安装的 apk 文件的签名证书来源不一致。可以通过先卸载之前的 apk,然后再安装新的 apk 来解决。卸载命令如下:

```
$ adb uninstall org.openqa.selenium.android.MainActivity
```

6. 如果需要在 headless X 环境中运行模拟器怎么办?

由于 Android 模拟器不兼容 xvfb,因此不能在 xvfb 中运行。如果需要在 headless X 环境中运行模拟器,则需要启用-no-window 选项。



6.5 小结

本章介绍了如何在 Android 平台上使用 Selenium WebDriver 进行测试。从搭建 Android WebDriver 测试环境入手，进一步阐释了如何操作手机上特定的功能，包括旋转屏幕、触摸和滚动。由于 Android 上的浏览器内核基于 Webkit，其对 HTML5 的支持也在本章的阐释范畴内。通过介绍 Web Storage 和 Application Cache，展示了 WebDriver 与 Android 上 HTML5 的交互过程。最后，以 Android 模拟器入手，介绍了如何在 Cloud 中测试 Android 应用。

第 7 章

Selenium 玩转 iOS

7.1 简介

从 Selenium 的官方文档来看，推荐用户使用 ios-driver 或者 appium 而不是官方发布的 iPhone Driver。它们的地址分别是

```
http://ios-driver.github.io/ios-driver/  
http://appium.io/
```

7.2 ios-driver

★7.2.1 ios-driver 简介

ios-driver 基于两种不同的框架构建起来，一种是针对原生 app 进行构建的，还有一种是针对 Web 的 app 或者混合式 app 进行构建的。鉴于两种不同 app 的设计原理，需要满足不同的开发环境需求。

1. 原生 app

由于使用 UIAutomation 框架，所以需要确保 iOS SDK 的版本大于 5.0。检查方法如下：

```
$ xcodebuild -showsdk
```

执行结果类似如下内容：

OS X SDKs:

OS X 10.8	-sdk macosx10.8
-----------	-----------------

iOS SDKs:

iOS 7.0	-sdk iphoneos7.0
---------	------------------

iOS Simulator SDKs:

Simulator -iOS 7.0	-sdk iphonesimulator7.0
--------------------	-------------------------

2. Web app 或者混合式 app

针对这种方式的 app，需要用到远程 Webkit 的调试协议，并且 iOS 的版本要求为 6+，





Safari 的版本要求为 6+。如果无法满足以上条件，也可以继续测试原生 app，但是无法在 Safari 上运行 Web 页面，也不能使用 DOM 选择器来与 UIWebviews 进行交互。

★7.2.2 ios-driver 的 Web app 实例

首先，进入 ios-driver 官方网站并下载 ios-server-0.6.5-jar-with-dependencies.jar，如图 7.1 所示。

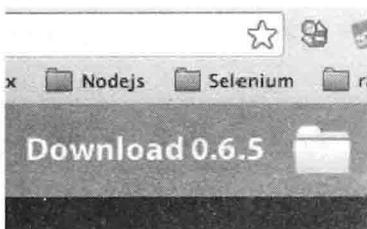


图 7.1 下载 ios-driver

在第一次运行 ios-driver 之前，应先确保以下目录和文件的权限更新：

```
$ sudo chmod a+rwx /Applications/Xcode.app/Contents/Developer/Platforms/  
iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator7.0.sdk/Applications  
  
$ sudo chmod a+rwx /Applications/Xcode.app/Contents/Developer/Platforms/  
iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator7.0.sdk/Applications/Mo-  
bileSafari.app
```

然后更新 MobileSafari Info.plist 的权限以允许 ios-driver 编辑它。执行命令如下：

```
$ sudo chmod 666  
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneSimulator.  
platform/Developer/SDKs/iPhoneSimulator7.0.sdk/Applications/MobileSafari.app/In-  
fo.plist
```

在启动 ios-driver 之前，请确保系统上安装的 Java 版本至少为 1.7.0 版本：

```
$ java -version  
java version "1.7.0_04"  
Java (TM) SE Runtime Environment (build 1.7.0_04-b21)  
Java HotSpot (TM) 64-Bit Server VM (build 23.0-b21, mixed mode)
```

接着进入刚下载的 ios-server-0.6.5-jar-with-dependencies.jar 的路径。执行如下命令：

```
$ java -jar ios-server-0.6.5-jar-with-dependencies.jar -simulators  
43:55:954 INFO ApplicationStore. <init> App archive folder:/Selenium2/  
selenium/applications  
44:00:611 INFO IOSServer. initDriver
```

```

Beta features enabled (enabled by -beta flag): false
Simulator enabled (enabled by -simulators flag): true
Inspector: http://0.0.0.0:5555/inspector/
Tests can access the server at http://0.0.0.0:5555/wd/hub
Server status: http://0.0.0.0:5555/wd/hub/status
Connected devices: http://0.0.0.0:5555/wd/hub/devices/all
Applications: http://0.0.0.0:5555/wd/hub/applications/all
Capabilities: http://0.0.0.0:5555/wd/hub/capabilities/all
Monitoring '/Selenium2/selenium/applications' for new applications
Archived apps: /Selenium2/selenium/applications
Build info:ios-driver 0.6.5(built:20131219-1439,sha:b9727e58d8c840ab82b73bae333be235697fac37)
Running on: Mac OS X 10.9 (x86_64)
Using java: 1.7.0_04
Using Xcode install: /Applications/xcode/Xcode5.app
Using instruments: version: 5.0.1, build: 51168
Using iOS version 7.0
iOS >= 6.0. Safari and hybrid apps are supported.

```

Applications :

```

-----
CFBundleName = Safari, CFBundleVersion = 9537.53, /Users/bob/.ios-driver/sa-
fariCopies/safari-7.0.app

```

2013-12-19 14:44:00.709: INFO:: jetty-7.x.y-SNAPSHOT

2013-12-19 14:44:00.930: INFO:: Started SelectChannelConnector@0.0.0.0: 5555

默认的端口号为 5555，可以在浏览器中访问如下地址，如果看到图 7.2 中类似 JSON 对象的信息，则说明前述操作成功。

`http://localhost:5555/wd/hub/status`

接下来以百度首页为例进行阐述。

注意：

在 Eclipse 中要添加之前下载的 `ios-server-0.6.5-jar-with-dependencies.jar` 文件，因为需要如下库的支持：

`org.uitautomation.ios.IOSCapabilites`



```
{
  "class": "org.openqa.selenium.remote.Response",
  "hCode": 514067204,
  "sessionId": null,
  "state": null,
  "status": 0,
  "value": {
    "build": {
      "revision": "120cf4311807e2e137e519f4c4877cf6340d0cbc",
      "time": "20130927-1435",
      "version": "0.6.5"
    },
    "ios": {"simulatorVersion": "7.0"},
    "java": {"version": "1.7.0_04"},
    "os": {
      "arch": "x86_64",
      "name": "Mac OS X",
      "version": "10.9"
    },
    "state": "success",
    "supportedApps": [
      {
        "CFBundleDevelopmentRegion": "English",
        "CFBundleDisplayName": "Safari",
        "CFBundleExecutable": "MobileSafari",
        "CFBundleIcons": {"CFBundlePrimaryIcon": {
          "CFBundleIconFiles": [
            "icon-spotlight~ipad.png",
            "icon-table~ipad.png",
            "icon-about~ipad.png",
            "icon-table~ipad.png",
            "icon-spotlight@2x.png",
            "icon-table@2x.png",
            "icon~ipad.png",
            "icon-about@2x.png",
            "icon@2x~iphone.png",
            "icon~iphone.png"
          ]
        }}
      }
    ]
  }
}
```

图 7.2 通过浏览器查看 ios-driver 启动状态

示例代码如下：

```
package com.learningselenium.ios;

import junit.framework.TestCase;

import java.net.URL;
import org.openqa.selenium.remote.RemoteWebDriver;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.uiautomation.ios.IOSCapabilities;

public class testIOSBaidu extends TestCase {

    public void testBaidu() throws Exception {
        DesiredCapabilities safari = IOSCapabilities.iphone("Safari");
        RemoteWebDriver driver = new RemoteWebDriver(
            new URL("http://localhost:5555/wd/hub"),
            safari);
        driver.get("http://www.baidu.com");
        String title = driver.getTitle();
        assertEquals("百度一下，你就知道", title);
        driver.quit();
    }
}
```

```

new URL("http://localhost:5555/wd/hub"), safari);

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

String url = driver.getCurrentUrl();

System.out.println(url);

driver.close();
}

}

```

示例代码详解：

1) 设置 DesiredCapabilities，其中 IOSCapabilities 设置为 iphone。如果应用程序运行在 iPad 模拟器上，那么相应的设置为 ipad 即可。参数 Safari 表示待测试应用程序的 bundle name，因为这里是测试 Web 页面，默认是用 Safari 浏览器打开。示例代码段如下：

```
DesiredCapabilities safari = IOSCapabilities.iphone("Safari");
```

- 2) 通过 iOS RemoteWebDriver 启动应用程序。
- 3) 打开百度页面，并获取当前打开页面的 URL。

★7.2.3 ios-driver 的 Native app 实例

接下来以苹果官方的示例程序 InternationalMountains 为例，对原生 app 的测试过程进行讲解，如图 7.3 所示。示例代码的下载地址如下：

<https://developer.apple.com/legacy/library/samplecode/InternationalMountains/Introduction/Intro.html>



图 7.3 下载 InternationalMountains 示例代码

该示例程序的运行界面如图 7.4 所示。

请确保你的 app 程序和 UIAutomation 已关联，方法如下：

在 Xcode 的菜单中选择 Product→Profile，并且选择 Automation，如图 7.5 所示。该设定会构建 app 并且启动 Instruments。

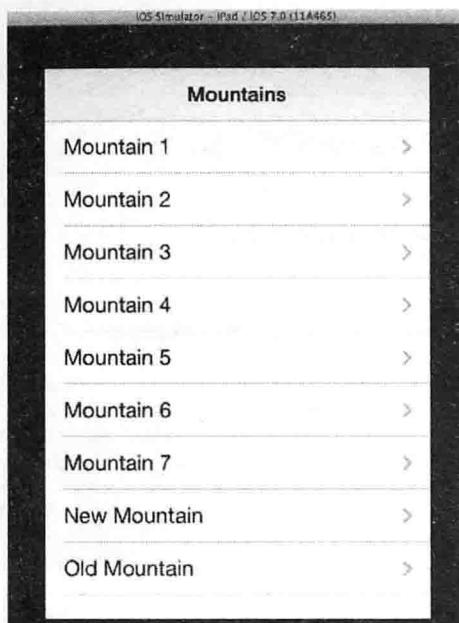


图 7.4 InternationalMountains 运行界面

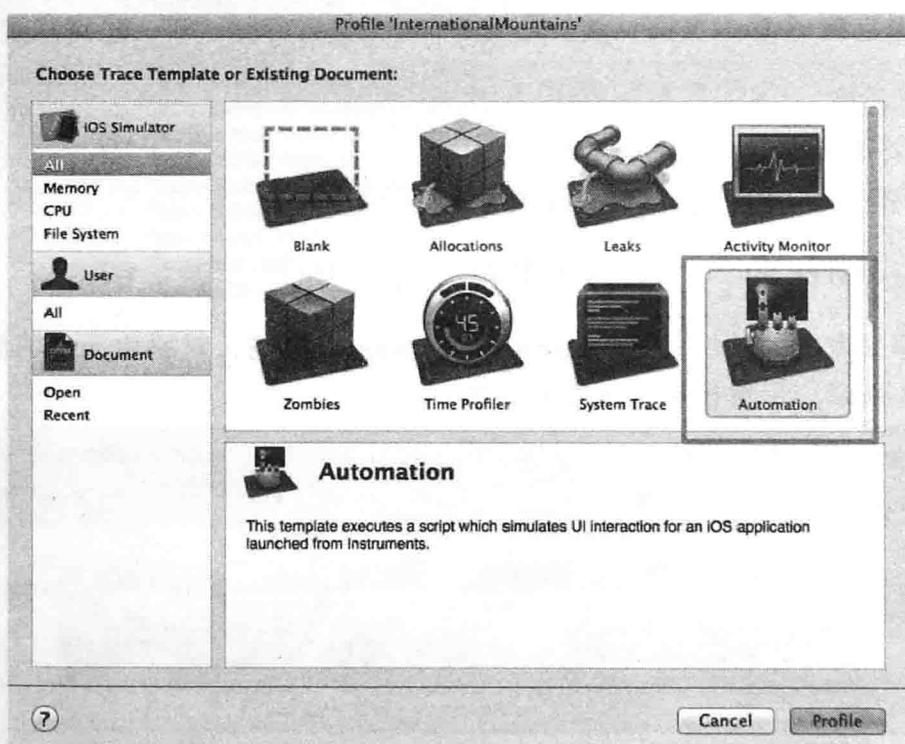


图 7.5 关联 InternationalMountains 程序和 UIAutomation

将 Instruments 关闭，并且在 Xcode 中选择 Window→Organizer→Projects，可以看到 InternationalMountains 项目的 app 文件所在路径，如图 7.6 所示。

InternationalMountains.app 的路径为

```
~/Library/Developer/Xcode/DerivedData/InternationalMountains-
eordguimrxknwoaynobkvplrkacs/Build/Products/Debug-
iphonesimulator/InternationalMountains.app
```

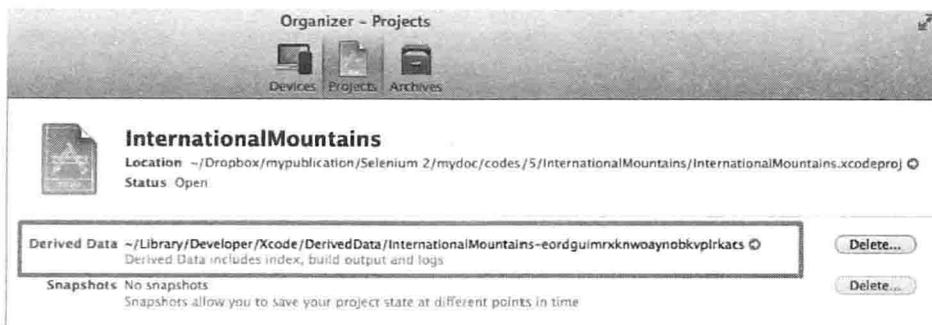


图7.6 查看InternationalMountains项目的app文件所在路径

接着进入刚下载的ios-server-0.6.5-jar-with-dependencies.jar的路径，执行如下命令：

```
$ java -jar ios-server-0.6.5-jar-with-dependencies.jar -aut
~/Library/Developer/Xcode/DerivedData/InternationalMountains-
eordguimrxknwoaynobkvplkacs/Build/Products/Debug-
iphonesimulator/InternationalMountains.app -port 4444
```

接下来确认ios-driver启动成功并且可以访问该app，通过浏览器访问如下地址，如果看到图7.7中类似JSON对象的信息，则说明前述操作成功。

<http://localhost:4444/wd/hub/status>



图7.7 通过浏览器查看ios-driver启动状态



如果是在模拟器中运行 app，则需要在启动 ios-driver 时使用-simulators 参数，执行命令及打印信息如下：

```
$ java -jar ios-server-0.6.5-jar-with-dependencies.jar -aut  
~/Library/Developer/Xcode/DerivedData/InternationalMountains-eordguimrxkn-  
woaynobkvplrkacs/Build/Products/Debug-  
iphonesimulator/InternationalMountains.app -port 4444 -simulators
```

```
54:54:605 INFO ApplicationStore. <init> App archive folder:/Selenium2/  
selenium/applications
```

```
54:57:193 INFO IOSServer.init
```

```
Beta features enabled ( enabled by -beta flag ) : false
```

```
Simulator enabled ( enabled by -simulators flag ) : true
```

```
Inspector: http://0.0.0.0:4444/inspector/
```

```
tests can access the server at http://0.0.0.0:4444/wd/hub
```

```
server status: http://0.0.0.0:4444/wd/hub/status
```

```
Connected devices: http://0.0.0.0:4444/wd/hub/devices/all
```

```
Applications: http://0.0.0.0:4444/wd/hub/applications/all
```

```
Capabilities: http://0.0.0.0:4444/wd/hub/capabilities/all
```

```
Monitoring '/Selenium2/selenium/applications' for new applications
```

```
Archived apps /Selenium2/selenium/applications
```

```
using xcode install : /Applications/xcode/Xcode5.app
```

```
using iOS version 7.0
```

```
ios >= 6.0. Safari and hybrid apps are supported.
```

```
Applications :
```

```
-----  
CFBundleName = InternationalMountains, CFBundleVersion = 1.1
```

```
CFBundleName = Safari, CFBundleVersion = 9537.53
```

```
2013-12-18 20:54:57.194:INFO::jetty-7.x.y-SNAPSHOT
```

```
2013-12-18 20:54:57.264:INFO::Started SelectChannelConnector@ 0.0.0.0:4444
```

应确保已经通过 Xcode 启动了模拟器，否则需要通过浏览器查看 ios-driver 的启动状态是否正常。

针对 InternationalMountains 的测试用例代码如下：

```
package com.learningselenium.ios;
```

```
import java.net.URL;  
import java.util.List;
```

```
import java.io.File;

import junit.framework.TestCase;

import org.openqa.selenium.By;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.TakesScreenshot;
import org.openqa.selenium.OutputType;
import org.openqa.selenium.remote.RemoteWebDriver;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.remote.Augmenter;
import org.openqa.selenium.ios.IOSCapabilities;

public class testiOSInternationalMountains extends TestCase {
    public void tesInternationalMountains() throws Exception {
        DesiredCapabilities nativeAppCap =
            IOSCapabilities.iphone("InternationalMountains", "1.1");

        RemoteWebDriver driver = new RemoteWebDriver(new URL(
            "http://localhost:4444/wd/hub"), nativeAppCap);

        List<WebElement> cells = driver.findElements(
            By.className("UIATableCell"));

        assertEquals(9, cells.size());

        WebElement first = cells.get(0);
        first.click();

        TakesScreenshot screen =
            (TakesScreenshot) new Augmenter().augment(driver);
        File ss = new File("Screenshot.png");
        screen.getScreenshotAs(OutputType.FILE).renameTo(ss);
        System.out.println("Screenshot take :" + ss.getAbsolutePath());

        By selector = By.xpath("//UIAStaticText[contains(@name,'climbed')]");
        WebElement text = driver.findElement(selector);
```





```

        System.out.println(text.getAttribute("name"));

    driver.quit();
}
}

```

示例代码详解：

- 1) 设置 DesiredCapabilities，其中 Bundle version 的获取方式如图 7.8 所示，在 Xcode 中可以查询到。

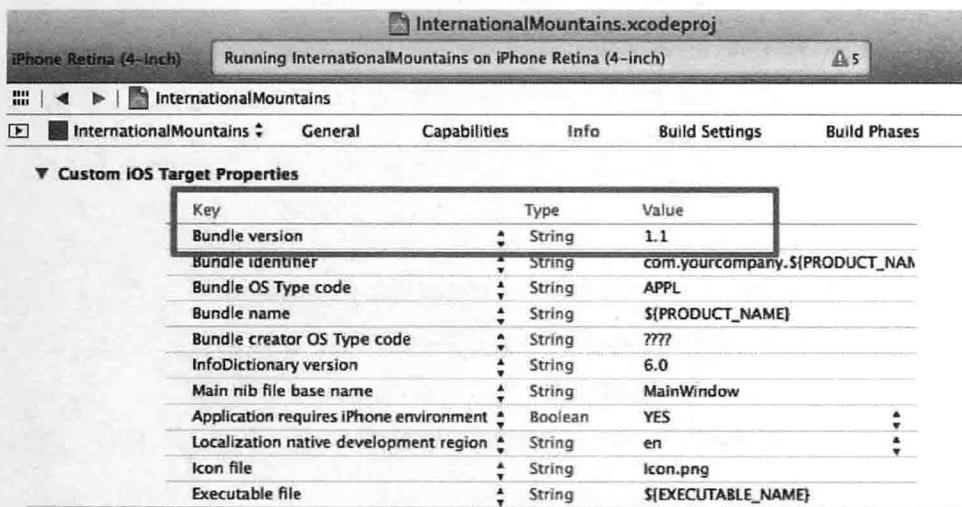


图 7.8 查看 InternationalMountains 的 Bundle version

- 2) 通过 iOS RemoteWebDriver 启动应用程序。
- 3) 验证应用程序启动后有 9 个元素在列表中。
- 4) 操作列表中的第一个元素。
- 5) 截屏操作和操作页面元素。

如果需要在真实设备上运行 app，则需要在启动 ios-driver 时使用 -beta 参数。执行命令如下：

```
$ java -jar ios-server-0.6.5-jar-with-dependencies.jar -beta -port 4444
```

★7.2.4 ios-driver 的源码编译

前置条件为系统已经安装 Git、JDK7 和 apache-maven。然后在 /etc/profile 中添加如下命令：

```

export JAVA_HOME = /Library/Java/JavaVirtualMachines/jdk1.7.0_xx.jdk/
Contents/Home

export M2_HOME = /Users/{YourAccountName}/Desktop/apache-maven-3.1.x

export PATH = $JAVA_HOME/bin: $PATH
export M2 = $M2_HOME/bin
export PATH = $M2: $PATH

```



通过如下地址并使用 Git 获取源码后解包：

```
https://github.com/ios-driver/ios-driver/
```

在解压后的源码根目录执行如下命令：

```
sudo mvn clean package
```

或者

```
sudo mvn clean install
```

如果最后编译成功，会看到如下日志信息：

```
[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] ios-automation ..... SUCCESS [1.506s]
[INFO] ios-common ..... SUCCESS [9.641s]
[INFO] ios-client ..... SUCCESS [3.921s]
[INFO] ios-server ..... SUCCESS [1:8.951s]
[INFO] ios-selenium-tests ..... SUCCESS [28.072s]
[INFO] libimobiledevice-wrapper ..... SUCCESS [6.816s]
[INFO] ios-grid ..... SUCCESS [1.487s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 2:20.837s
[INFO] Finished at: Thu Dec 19 14:42:09 CST 2013
[INFO] Final Memory: 37M/89M
[INFO] -----
```

如果不需要运行测试用例，则加上如下参数即可。

```
-DskipTests
```

如果在编译过程中，出现如下错误信息，说明系统中安装了 JDK1.6 和 JDK1.7 两个不同的版本。

```
[WARNING] Rule 0: org.apache.maven.plugins.enforcer.RequireJavaVersion
failed with message:
Detected JDK Version: 1.6.0-43 is not in the allowed range 1.7.
```

解决方案为，在用户根目录新建 .mavenrc 文件并添加如下内容：

```
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.7.0_xx.jdk/
Contents/Home
```



7.3 Appium

★7.3.1 Appium 简介

Appium 是一个开源的、跨平台的测试框架，可以用来测试移动设备上 Native 或者 Hybrid 的应用程序。Appium 同时支持 iOS、Android 和 FirefoxOS 等多种移动平台。

1. Appium 的工作原理

1) iOS：通过 WebDriver 的 JSON Wire 协议来驱动 iOS 系统的 UIAutomation 库。

2) Android：通过 WebDriver 的 JSON Wire 协议来驱动 Android 系统的 UIAutomator 框架。

3) FirefoxOS：通过 WebDriver 的 JSON Wire 协议来驱动基于 Gecko 的 Marionette 框架。

2. Appium 的优势

1) Appium 在不同的移动平台上均使用了标准的自动化 API，所以用户不需要重新编译或者修改 app。

2) Appium 支持 Selenium WebDriver 能支持的所有绑定编程语言，如 Java、Python、JavaScript、C#、Ruby 等。

3) Appium 支持的测试框架相当广泛。

在 Appium 出现之前，如果使用 iOS 的 UIAutomation 框架，则只能使用 JavaScript 来编写测试用例，用 Instruments 来运行测试用例。如果使用 Android 的 UIAutomator 框架，则只能使用 Java 来编写测试用例。

3. 安装 Appium

1) 使用 Node.js 安装 Appium，命令如下。安装之前应先确认系统中已经安装了 Node.js。

```
$ sudo npm install -g appium
```

2) 直接从官网下载并解压使用，地址如下：

```
https://github.com/appium/appium/releases
```

3) 直接下载 Appium.dmg 文件并解压使用，这种方式是带 UI 的 app。地址如下：

```
https://bitbucket.org/appium/appium.app/downloads/
```

4. 启动 Appium

```
$ appium &
```

启动成功后可以看到如下信息：

```
Info: Welcome to Appium v0.13.0
```

```
Info: Appium REST http interface listener started on 0.0.0.0:4723
```

```
Info: - socket.io started
```

如果下载的是 Appium.app 文件，则双击打开即可。界面如图 7.9 所示。

本书将重点介绍 Appium 与 iOS 系统的交互，而与 Android 系统交互的更多信息，可查询 Appium 的官方网站。

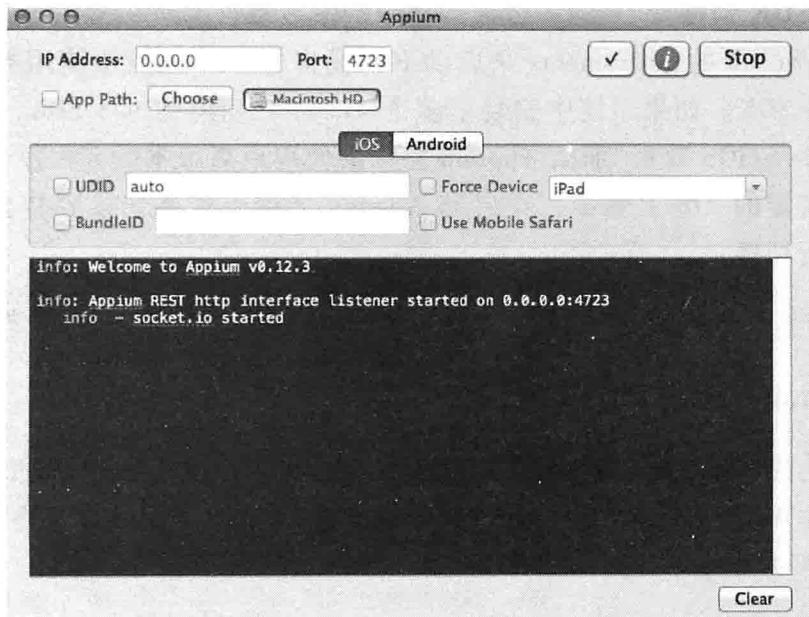


图 7.9 Appium 启动界面

★7.3.2 Appium 的 iOS 配置

1. 配置信息

- 1) 确保 Mac OS X 的最低版本为 10.7。
- 2) Xcode/iOS 兼容 Xcode 4.6.3/iOS 6.1.x，或者 Xcode 5/iOS 7.x。勿使用高版本的 Xcode 配合低版本的 iOS SDK，如 Xcode 5/iOS 6.x。
- 3) 确保授权 iOS 模拟器的使用。

① 如果通过 Node.js 安装 Appium，则运行如下命令即可：

```
$ sudo authorize_ios
```

其中，authorize_ios 是 Appium npm 包中的一个二进制文件。授权成功后的信息如下：

```
Enabling DevToolsSecurity
Updating security db for developer access
Granting access to built-in simulator apps
Authorization successful
```

② 如果通过在官网下载源码包的方式来运行 Appium，则运行如下命令即可：

```
$ sudo grunt authorize
```

③ 如果直接使用 Appium.app，则在 UI 上操作即可，如图 7.10 所示。

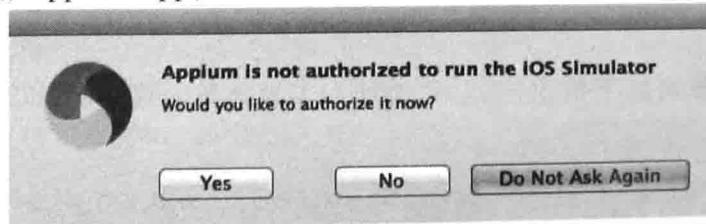


图 7.10 Appium 授权 iOS 模拟器的使用



2. 多个 iOS SDK 切换

Appium 使用 Xcode 的 Instruments 来启动 iOS 模拟器，并且默认使用当前安装的 Xcode 的最高版本的 iOS SDK。如果系统中安装了多个 Xcode 和相应的 iOS SDK，如 Xcode 4.6.3/iOS 6.1.x 和 Xcode 5/iOS 7.x，那么 Appium 会强制使用最高版本的 iOS 7.x。

如果需要在特定的 iOS 上测试，在启动 Appium 之前，需要切换到特定的测试版本上。使用如下命令进行切换：

```
$ sudo xcode-select -switch /Applications/Xcode.app/Contents/Developer/
```

★7.3.3 Appium 的 Web app 实例

首先以 iOS 上的 Web app 为例阐述 Appium 的使用。请先确认 iOS 模拟器或者真实设备上 Safari 浏览器的 Web Inspector 为打开状态，如图 7.11 所示。具体操作路径为 Settings→Safari→Advanced→Web Inspector。

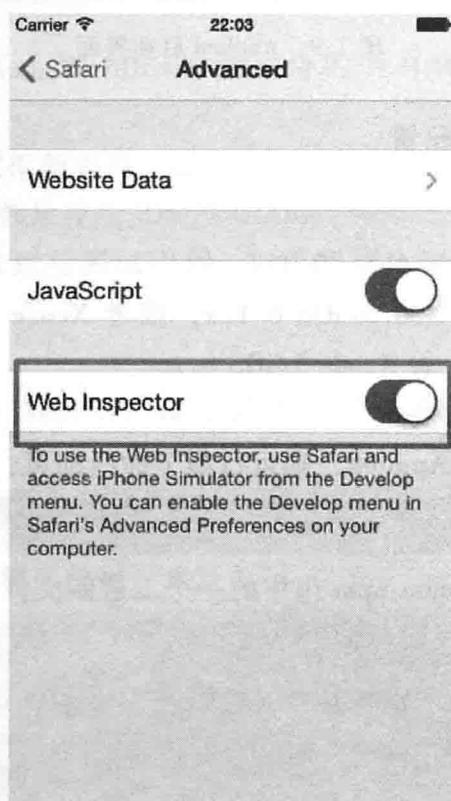


图 7.11 开启 iOS Safari 的 Web Inspector 选项

如前面所述，接下来启动 Appium。如果以 iOS 7 为测试目标版本，那么建议 Appium 的最低版本至少为 0.13.0。启动 Appium 的命令如下：

```
$ appium &
```

启动成功后可以看到如下信息，其 IP 地址为本地地址，端口为 4723。

```
Info: Welcome to Appium v0.13.0  
Info: Appium REST http interface listener started on 0.0.0.0:4723  
Info: - socket.io started
```

以打开百度首页为例，以下为测试用例代码：

```
package com.learningselenium.ios;

import java.net.URL;

import junit.framework.TestCase;

import org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.remote.RemoteWebDriver;

public class testiOSAppiumBaidu extends TestCase {

    public void testBaidu() throws Exception {
        DesiredCapabilities safari = new DesiredCapabilities();
        safari.setCapability("app", "safari");
        RemoteWebDriver driver = new RemoteWebDriver(
            new URL("http://localhost:4723/wd/hub"), safari);

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

        String url = driver.getCurrentUrl();

        System.out.println(url);

        driver.close();
    }
}
```

示例代码详解：

- 1) 设置 DesiredCapabilities 为 Safari 浏览器 app。
- 2) 打开 RemoteWebDriver，URL 地址为 Appium 启动时的本地地址和端口 4723。
- 3) 打开百度主页面并获取当前页面的 URL 地址。

Appium 的执行日志解析：

```
//接收启动参数
debug: Appium request initiated at /wd/hub/session
debug: Request received with params: {"desiredCapabilities": {"app": "safari"}}

//启动模拟器
debug: Launching device: iPhone Retina (4-inch)
```



```
debug: Creating instruments

//启动 Instruments 并等待接收新命令
info: Instruments launched. Starting poll loop for new commands.
info: Device launched! Ready for commands (will time out in 60secs)

//打开百度主页,执行成功并返回客户端结果
debug: Request received with params: {"url":"http://www.baidu.com"}
info: Responding to client with success: {"status":0,"value":null,"sessionId":"ed3078c3-884c-4fe0-b052-413d6c8ceaa3"}

//无新的命令发送过来,超时后的清理和关闭操作
info: Didn't get a new command in 60 secs, shutting down...
info: Shutting down appium session...
info: Stopping ios
info: Sending sigkill to instruments
info: Cleaning up after instruments exit
info: [REMOTE] Disconnecting from remote debugger
info: Killing the simulator process and daemons
info: [REMOTE] Debugger socket disconnected
info: Cleaning up appium session
```

而针对 iOS 真实设备的 Web app 测试，需要预先准备以下几个方面。

1. Homebrew

Homebrew 用于在 Mac 系统上安装各种开源软件包，其安装形式有点类似于 Ubuntu 系统上的 apt。下载地址如下，主页面如图 7.12 所示。

```
http://brew.sh/
```

如果系统上没有安装过 Homebrew，则需要通过如下命令来进行安装。如果已经安装了 Homebrew，则可以跳过第一条命令，直接更新 brew 至最新版本。

```
$ ruby -e "$(curl -fsSL https://raw.github.com/mxcl/homebrew/go/install)"
$ brew update
```

2. ios_webkit_debug_proxy

ios_webkit_debug_proxy 是 Google 开发的一个开源工具，用于让开发者通过 Chrome 浏览器的开发者工具以及 Webkit 的 Remote Debugging Protocol 来与 iOS 真实设备或者模拟器上的 MobileSafari 和 UIWebView 进行交互。通过 Homebrew 安装 ios_webkit_debug_proxy 的命令如下：

```
$ brew install ios_webkit_debug_proxy
```

ios_webkit_debug_proxy 的原理如图 7.13 所示。其中，Chrome 的开发者工具中所涉及的各种请求会被转换为 Apple 的 Remote Web Inspector Service 调用。



图 7.12 Homebrew 主页面

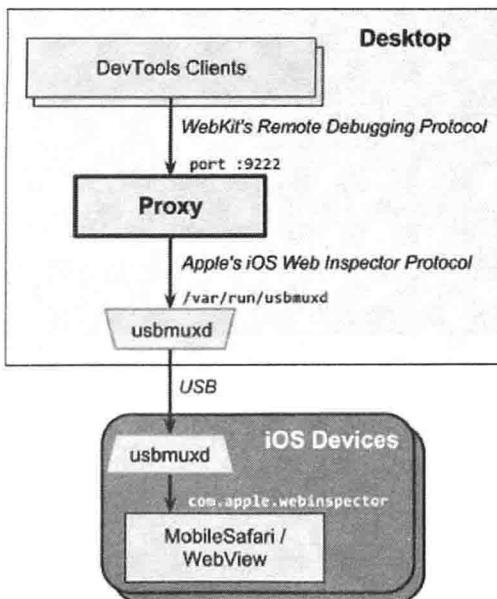


图 7.13 ios_webkit_debug_proxy 的原理图

3. iOS 真实设备的 Identifier

每一台 iOS 真实设备都有唯一的一个 Identifier。查看它的方式如图 7.14 所示，通过 iPhone Configuration Utility 即可获取。

以上三个准备工作就绪后，接下来便可进入与 iOS 真实设备进行交互的阶段。

启动 ios_webkit_debug_proxy 的命令如下，其中 Identifier 即为 2d6467d798b7f65d5c54e8c71043a98a540c2f17：

```
$ ios_webkit_debug_proxy -c <Identifier>: 27753-d
```

Identifier 替换为真实的 UDID 后命令如下，启动成功后的界面如图 7.15 所示。

```
$ ios_webkit_debug_proxy -c  
2d6467d798b7f65d5c54e8c71043a98a540c2f17: 27753 -d
```

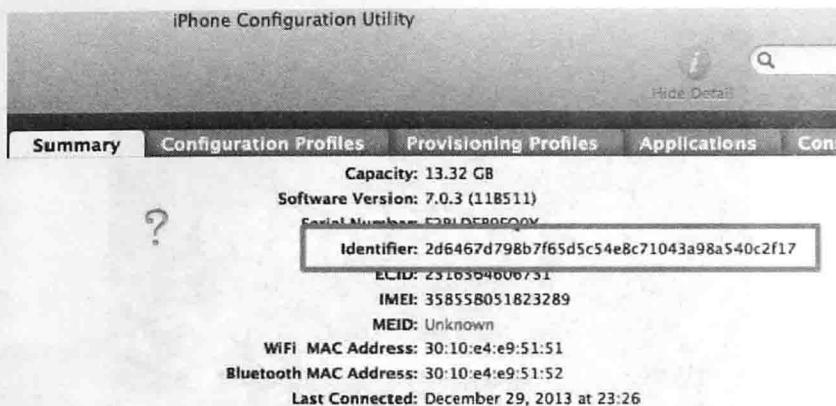


图 7.14 查看 iOS 真实设备的 Identifier

```
$ ios_webkit_debug_proxy -a 2d6467d798b7f65d5c54e8c71043a98a540c2f17:27753 -d
ss.add_fd(3)
select_port() failed
ss.recv fd=3 len=294
ss.recv fd=3 len=685
ss.add_server_fd(4)
ss.add_fd(6)
wi.send_packet[245]:
00 00 00 F1 62 70 6C 69 73 74 30 30 D1 01 02 5F 10 12 57 ... .bplist00..._.W
49 52 46 69 6E 61 6C 40 65 73 73 61 67 65 4B 65 79 4F 10 IRFinalMessageKeyO.
AA 62 70 6C 69 73 74 30 30 D2 01 03 02 04 5A 5F 5F 73 65 .bplist00....Z_se
6C 65 63 74 6F 72 5F 10 16 5F 72 70 63 5F 72 65 70 6F 72 lector... rpc_repor
74 49 64 65 6E 74 69 66 69 65 72 3A 5A 5F 5F 61 72 67 75 tIdentifier:z_argu
6D 65 6E 74 D1 05 06 5F 10 1A 57 49 52 43 68 62 62 65 63 ment.... WIRConnec
74 69 6F 6E 49 64 65 6E 74 69 66 69 65 72 4B 65 79 5F 10 tionIdentifierKey_.
24 31 37 41 32 45 41 42 32 2D 41 36 39 33 2D 34 46 30 30 $17A2EAB2-A693-4F00
2D 42 39 45 41 2D 37 42 44 46 31 45 38 37 42 34 42 33 08 -B9EA-7BDF1E87B4B3.
0D 18 31 3C 3F 5C 00 00 00 00 00 00 01 01 00 00 00 00 00 00 ..1<?...
00 00 07 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 83 ...
A0 08 0B 20 00 00 00 00 00 00 01 01 00 00 00 00 00 00 00 00 ...
03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 CE ...
ss.recv fd=6 len=167
```

图 7.15 ios_webkit_debug_proxy 启动成功的界面

接下来需要下载 Appium 的最新源码并进行编译，以支持真实设备的测试。命令如下：

```
$ git clone https://github.com/appium/appium.git
$ cd appium
$ ./reset.sh --ios --real-safari
$ ./reset.sh --ios --real-safari --code-sign '<code signing identity>'
```

在这个过程中，Appium 源码包已经将 SafariLauncher 包含进来并进行编译，其文件夹路径如下：

```
$ pwd
~/appium/submodules/SafariLauncher
```

编译成功后，可以看到如下信息：

```
Done, without errors.
----reset.sh completed successfully----
```

启动 Appium，其中 Identifier 在本例中即为 2d6467d798b7f65d5c54e8c71043a98a540c2f17：

```
$ node lib/server/main.js -U <Identifier>
```

Identifier 替换为真实的 UDID 后命令如下：

```
$ node lib/server/main.js -U 2d6467d798b7f65d5c54e8c71043a98a540c2f17
```

启动Appium成功后的信息如图7.16所示，默认端口为4723。

```
$ node lib/server/main.js -U 2d6467d798b7f65d5c54e8c71043a98a540c2f17
info: Welcome to Appium v0.13.0
info: Appium REST http interface listener started on 0.0.0.0:4723
  info - socket.io started
info: Spawning instruments force-quitting watcher process
info: [FQInstruments] Force quit unresponsive instruments v0.0.1
```

图7.16 Appium启动成功的界面

下面以SafariLauncher打开真实设备上的Safari为例进行阐述，该测试用例会打开百度主页并通过WebDriver获取当前页面的URL信息。

```
package com.learning selenium.ios;

import java.net.URL;
import java.util.concurrent.TimeUnit;

import org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.remote.RemoteWebDriver;

import junit.framework.TestCase;

public class testiOSAppiumBaiduRealDevice extends TestCase {
    public void testBaidu() throws Exception {
        DesiredCapabilities safariCapabilities = new DesiredCapabilities();
        safariCapabilities.setCapability("device", "iphone");
        safariCapabilities.setCapability("version", "7.0");
        safariCapabilities.setCapability("app", "safari");

        RemoteWebDriver driver = new RemoteWebDriver(
            new URL("http://localhost:4723/wd/hub"), safariCapabilities);

        driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);

        driver.get("http://www.baidu.com");
        String url = driver.getCurrentUrl();
```





```
System.out.println(url);
```

```
driver.close();
```

```
}
```

```
}
```

示例代码详解：

- 1) 设置 DesiredCapabilities，必须设置 device 为 iphone 才表示指定在 iOS 真实设备上运行测试用例。
- 2) 启动 RemoteWebDriver，即与之前启动的 Appium 进行交互，端口为 4723。这个过程会自动将 Appium 中编译好的 SafariLauncher.app 安装到真实设备上，然后在设备上启动 SafariLauncher 并打开 Safari 浏览器。其中，SafariLauncher 启动后的界面如图 7.17 所示。
- 3) 设置等待超时的时间间隔，因为在真实设备上首先要启动 SafariLauncher，然后再启动 Safari，中间有比较长的时间开销。
- 4) 在 Safari 中打开百度主页并且获取当前页面 URL 信息。



图 7.17 SafariLauncher 的界面

小贴士

如何在不启动 Xcode 的情况下，直接打开 iOS 模拟器？

进入目录：

```
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneSimulator.  
platform/Developer/Applications
```

然后将 iOS Simulator.app 拖到 Dock 上，就可以随时启动 iOS 模拟器。

7.4 小结

本章介绍了如何在 iOS 平台上运行 Selenium WebDriver。主要介绍了两种不同的第三方工具套件，一个是 ios-driver，还有一个是 Appium。它们各有千秋，都有自己的独门绝技。无论对于 iOS 模拟器或真实设备的支持，还是对于 Web app 和 Native app 的支持，Selenium WebDriver 都在不断地前进发展中。

第 8 章

Selenium 玩转 Raspberry Pi

8.1 简介

Raspberry Pi（中文名为“树莓派”，简写为 RPi，或者 RasPi/RPi）是为学生计算机编程教育而设计的，只有信用卡大小的卡片式计算机，其系统基于 Linux。

它由注册于英国的慈善组织“Raspberry Pi 基金会”开发，EbenUpton（埃·厄普顿）为项目带头人。2012 年 3 月，英国剑桥大学 Eben Epton 正式发售世界上最小的台式机，又称卡片式计算机。其外形只有信用卡大小，却具有计算机的所有基本功能，如图 8.1 所示。这就是 Raspberry Pi 电脑板。

这一基金会以提升学校计算机科学及相关学科的教育，让计算机变得有趣为宗旨。基金会期望这一款计算机无论在发展中国家还是在发达国家都会有更多的其他应用不断被开发出来，并应用到更多领域。在 2006 年，树莓派早期概念是基于 Atmel 的 ATmega644 单片机，首批上市的 10000 “台” 树莓派的“板子”，由中国厂家制造。截至 2013 年 10 月 1 日，树莓派只有 A 和 B 两个型号，主要区别在于 A 型为 1 个 USB、无有线网络接口、功率 2.5W，500mA；B 型为 2 个 USB、支持有线网络、功率 3.5W，700mA。

树莓派配备一枚 700MHz 的 ARM 架构处理器，用 SD 卡当作存储介质。它可以运行像雷神之锤Ⅲ竞技场这样的游戏和进行 1080p 影片的播放。其操作系统采用开源的 Linux，如 Debian、ArchLinux 等，自带的浏览器和办公软件能够满足基本的网络浏览、文字处理和计算机学习需要。最重要的是，它默认以 Python 作为主要编程语言。

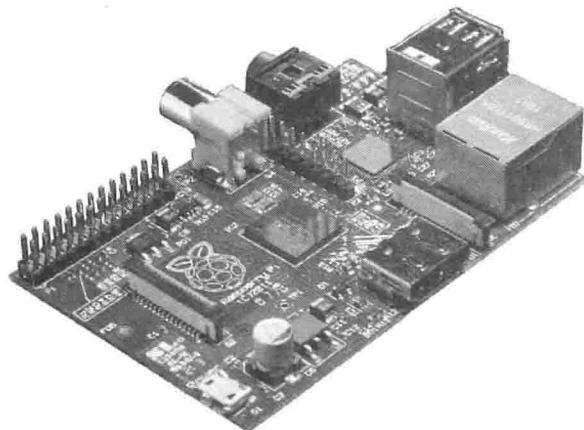


图 8.1 Raspberry Pi 计算机

8.2 操作系统层面的准备工作

在这里假设大家都已经知道如何安装操作系统到 SD 卡并且启动树莓派，具体操作步骤可参看树莓派官方网站的用户指南。推荐使用 Raspbian，因为它专门针对树莓派的硬件特点进行了优化。这款 OS 对浮点运算有更好的支持，能为用户带来更快的上网浏览体验。另

外，在固件、核心、应用方面也都有了改进，所以它是最适合普通用户使用的 OS。下面所阐述的步骤也是围绕 Raspbian 来展开的。

Raspbian 版本下载地址为

```
http://www.raspberrypi.org/downloads
```

其运行界面如图 8.2 所示。

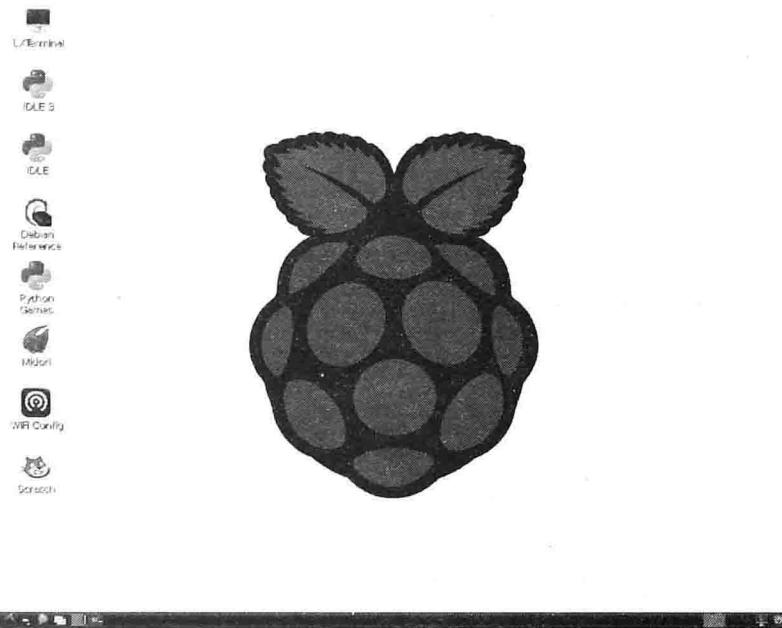


图 8.2 Raspberry Pi 的运行界面

8.3 依赖包的安装

为了演示如何在树莓派上运行 Selenium，需要安装如下依赖包。并且由于树莓派的默认编程语言是 Python，也会借此顺水推舟，用 Python 语言来展开实验。而且在性能并不高的嵌入式设备上运行脚本式语言，调试和修改起来会比编译式语言方便很多。主角 Selenium 则会基于 Firefox WebDriver 展开阐述。

1) 在安装软件包之前，先执行如下命令以保证安装的软件都是最新的。

```
sudo apt-get update
```

2) 安装 Iceweasel。

```
sudo apt-get install iceweasel
```

Iceweasel 是一个网络浏览器，其实就是 Mozilla Firefox 浏览器的 Debian 再发行版本。从 Debian Linux 4.0 开始，Debian Linux 均默认使用 Iceweasel，而不是 Mozilla Firefox。因为 Mozilla 组织注册了 Firefox 商标，为了避免可能存在的版权问题和法律上的纠纷，Debian 组织则另外采用 Iceweasel 的标识来引用原本是 Firefox 的浏览器。

3) 安装 python pip。pip 是安装和管理 Python 包的工具，用来替代 easy-install。

```
sudo apt-get install python-pip
```





4) 安装 Python 版的 Selenium。

```
sudo pip install selenium
```

注意：

受制于树莓派极其有限的 CPU/GPU 资源，需要对 Python 版本的 Selenium Firefox WebDriver 做如下调整才能让测试代码在树莓派上正常运转起来。

```
sudo vim /usr/local/lib/python2.7/dist-packages/selenium/
webdriver/firefox/firefox_binary.py
```

在 `wait_until_connectable` 函数中，将 `count == 30` 调整为 `count == 80`，这样才能在树莓派上给予 Iceweasel 充足的时间以启动，并正常调用 Firefox WebDriver 的功能。

8.4 运行 Python 版的 Selenium

本示例以百度主页为例进行阐述，如图 8.3 所示。在百度的搜索输入框中输入 10 次字符串并且进行搜索，每次搜索后都回退到前一个页面再次进行搜索。示例代码如下：

```
import time
from selenium import webdriver

class RaspberryPiDemo():
    def __init__(self):
        print "Demo: Selenium WebDriver with Firefox on Raspberry Pi"
        self.demo_impl = RaspberryPiImpl()

    print
    print "Set up selenium (this will take a while, be patient) ..."
    self.setUp()

    print
    print "Run cases ..."
    self.doClick()

    print
    print "Shutting down ..."
    self.tearDown()
```



```
def setUp(self):
    self.demo_impl.setUp()

def doClick(self):
    self.demo_impl.doClick()

def tearDown(self):
    self.demo_impl.tearDown()

class RaspberryPi_Impl():
    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(15)
        self.base_url = "http://www.baidu.com"

    def doClick(self):
        self.count = 0
        while self.count < 10:
            self.driver.get(self.base_url + "/")
            self.driver.find_element_by_id("kw").send_keys("selenium" + str(self.count))
            self.driver.find_element_by_id("su").click()
            print self.driver.title

            self.driver.back()
            time.sleep(1)
            self.count += 1

    def tearDown(self):
        self.driver.quit()

if __name__ == "__main__":
    RaspberryPiDemo()
```

如果不希望启动 GUI 而直接在命令行里面以后台的方式执行，则可以通过以下两种方式达到目标：

- 1) 在运行程序之前执行如下命令，然后再执行之前的 Python 代码。





图 8.3 Raspberry Pi 上 Iceweasel 打开百度主页

```
export DISPLAY = 0
```

2) 通过 Python 的 Virtual Display 来完成，它是 xvfb 的 Python 封装版本。

```
sudo pip install PyVirtualDisplay
```

再在代码中添加如下语句：

```
from pyvirtualdisplay import Display
```

然后在 RaspberryPi_Impl() 的 setUp() 函数中添加两句代码：

```
display = Display(visible =0, size =(800, 600)) display.start()
```

修改后的 setUp() 函数看起来如下：

```
class RaspberryPi_Impl():

    def setUp (self):
        display = Display (visible =0, size = (800, 600))
        display.start ()
        self.driver = webdriver.Firefox ()
        self.driver.implicitly_wait (15)
        self.base_url = "http://www.baidu.com"
```



8.5 运行Standalone版的Selenium Server

1. 安装Java运行环境

```
sudo apt-get install default-jdk
```

2. 安装Selenium Standalone Server

```
mkdir /home/pi/selenium  
cd /home/pi/selenium  
wget http://selenium.googlecode.com/files/selenium-server-  
standalone-2.37.0.jar  
sudo mkdir -p /var/log/selenium  
sudo chmod a+w /var/log/selenium
```

3. 在系统中添加启动和关闭Selenium的脚本

```
sudo vim /etc/init.d/selenium
```

/etc/init.d/selenium的脚本内容如下：

```
#!/bin/bash  
  
case "$1" in  
    'start')  
        if test -f /tmp/selenium.pid  
        then  
            echo "Selenium is already running."  
        else  
            export DISPLAY=:0  
            java -jar /home/pi/selenium/selenium-server-standalone-  
2.37.0.jar -port 4443 > /var/log/selenium/selenium-output.log 2> /var/log/selenium/  
selenium-error.log & echo $! > /tmp/selenium.pid  
            echo "Starting Selenium..."  
  
            error=$?  
            if test $error -gt 0  
            then  
                echo "$1 Error $error! Couldn't start  
Selenium! $1 boff"  
            fi  
        fi  
    esac
```



```
;;
'stop')
    if test -f /tmp/selenium.pid
    then
        echo "Stopping Selenium..."
        PID=`cat /tmp/selenium.pid`
        kill -3 $PID
        if kill -9 $PID ;
        then
            sleep 2
            test -f /tmp/selenium.pid && rm -f /tmp/
selenium.pid
        else
            echo "Selenium could not be
stopped..."
        fi
    else
        echo "Selenium is not running."
    fi
;;
'restart')
    if test -f /tmp/selenium.pid
    then
        kill -HUP `cat /tmp/selenium.pid`
        test -f /tmp/selenium.pid && rm -f /tmp/selenium.pid
        sleep 1
        export DISPLAY=:0
        java -jar /home/pi/selenium/selenium-server-standalone-2.37.0.jar-
port 4443 > /var/log/selenium/selenium-output.log 2> /var/log/selenium/selenium-
error.log & echo $! > /tmp/selenium.pid
        echo "Reload Selenium..."
    else
        echo "Selenium isn't running..."
    fi
;;
* )      # no parameter specified
echo "Usage: $SELF start|stop|restart|reload|force-reload|sta-
tus"
exit 1
;;
esac
```

保存/etc/init.d/selenium 脚本，然后将其添加为默认的启动服务：

```
sudo chmod 755 /etc/init.d/selenium sudo update-rc.d selenium defaults
```

4. 启动 Selenium Standalone Server 作为后台服务

```
/etc/init.d/selenium start
```

看到如下提示信息就说明启动成功了。

```
Starting Selenium...
```

注意：

脚本中的 export DISPLAY=:0,这样可以保证即使系统没有安装 xvfb,也可以让测试用例在无 UI 的环境下正常运行。

使用 Selenium Standalone Server 的方式和前述单纯用 Python 的 WebDriver 的最大区别主要体现在两个方面：

1) 导入新 DesiredCapabilities 包：

```
from selenium.webdriver.common.desired_capabilities import DesiredCapabilities
```

2) RaspberryPiImpl() 的 setUp() 中设定 WebDriver 的方式变成了 Remote 模式：

```
self.driver =
    webdriver.Remote(command_executor='http://192.168.0.107:4443/wd/hub',
                     desired_capabilities=DesiredCapabilities.FIREFOX)
```

完整的测试代码如下：

```
import time

from selenium import webdriver
from selenium.webdriver.common.desired_capabilities import DesiredCapabilities

class RaspberryPiDemo():
    def __init__(self):
        print "Demo: Remote Selenium WebDriver with Firefox on Raspberry Pi"
        self.demo_impl = RaspberryPiImpl()

    print
    print "Set up selenium (this will take a while, be patient) ..."
    self.setUp()

    print
```



```
print "Run cases ..."  
self. doClick( )  
  
print  
print "Shutting down..."  
self. tearDown( )  
  
def setUp( self ):  
    self. demo_ impl. setUp ( )  
  
def doClick ( self ):  
    self. demo_ impl. doClick ( )  
  
def tearDown ( self ):  
    self. demo_ impl. tearDown ( )  
  
class RaspberryPi_ impl ( ):  
  
    def setUp ( self ):  
        self. driver = webdriver. Remote ( command_ executor ='http://192.168.0.107:  
4443/wd/hub',  
                                         desired_capabilities =DesiredCapabilities. FIREFOX)  
        self. driver. implicitly_ wait ( 15 )  
        self. base_ url = "http://www. baidu. com"  
  
    def doClick ( self ):  
        self. count = 0  
        while self. count < 10:  
            self. driver. get ( self. base_ url + "/" )  
            self. driver. find_ element_ by_ id ("kw") . send_ keys ("selenium" + str  
( self. count ))  
            self. driver. find_ element_ by_ id ("su") . click ( )  
            print self. driver. title  
  
            self. driver. back ( )  
            time. sleep ( 1 )  
            self. count += 1  
  
    def tearDown ( self ):
```

```

        self.driver.quit()

if __name__ == "__main__":
    RaspberryPiDemo()

```

打开浏览器，可以看到有一个新的 Session 连接到 Selenium Standalone Server。运行结果如图 8.4 所示。

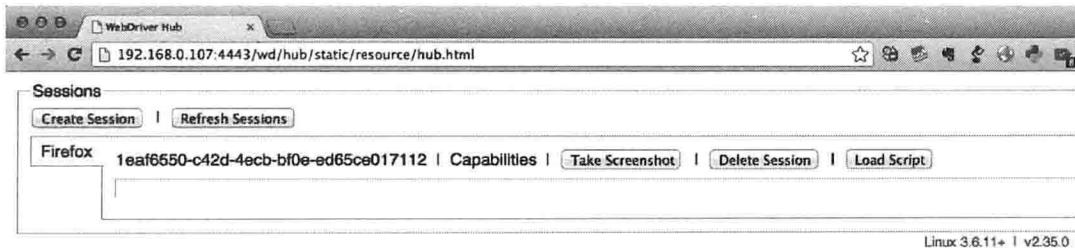


图 8.4 Raspberry Pi 上运行 Selenium Standalone Server

Python 代码的执行结果如图 8.5 所示。

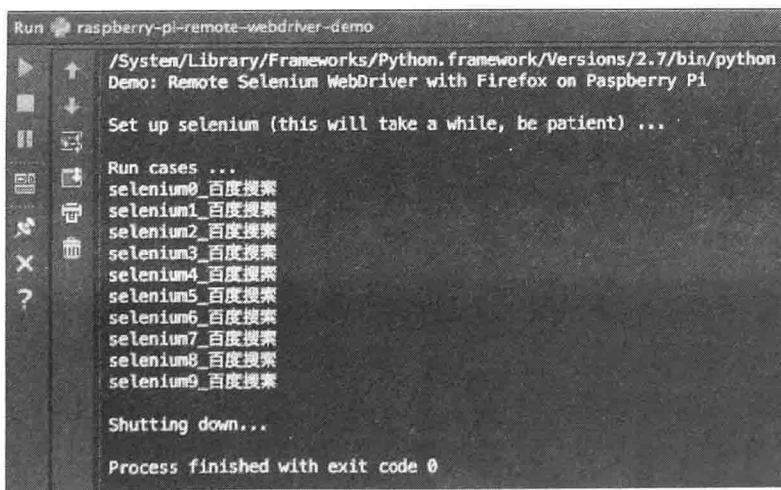


图 8.5 Python 客户端运行结果

8.6 小结

本章以 Raspberry Pi 为例介绍了如何在这个嵌入式平台上运行 Python 版本的 Selenium WebDriver 和 Standalone 版本的 Selenium Server。

第 9 章

Selenium Grid

9.1 简介

★9.1.1 Selenium Grid 是什么

有 Selenium Grid 利器在手，便可以同时在不同机器上测试不同浏览器，即同时测试运行多种浏览器和操作系统的不同机器。从本质上说，Selenium Grid 支持分布式测试，使测试人员可以在分布式环境中进行测试。

★9.1.2 何时使用 Selenium Grid

一般而言，在面临以下情况时可以考虑使用 Selenium Grid：

- 1) 测试多个浏览器或单个浏览器多种版本，又或者测试不同操作系统上的各种浏览器。
- 2) 减少测试套件运行的时间。

Selenium Grid 运用多个机器同时并列运行，目的在于加快测试用例运行的速度，从而减少测试运行的总时间。如对于包含 100 个测试用例的套件来说，如果用 Selenium Grid 同时在 4 台机器上（虚拟机或者物理机）运行这些测试用例，完成测试所花时间只需在单台机器上连续测试所花时间的 1/4。对于大型测试套件和需要处理海量数据验证的测试套件，Selenium Grid 毫无疑问可以节约大量时间。Selenium Grid 的另一个优势在于可以通过节省测试时间而更快地将测试结果返还给开发人员。越来越多的软件开发团队运用敏捷开发方式，他们希望在最短时间内获得测试人员的测试结果而不是在漫漫长夜中苦等测试通过。

Selenium Grid 还可用于在多种运行环境中进行测试，即并行测试多种浏览器。举例来说，在一组虚拟机网格中，可以设置每台虚拟机都支持被测试程序所支持的某种浏览器，如机器 1 使用 IE8，机器 2 使用 IE10，机器 3 使用 Chrome 的最新版本，机器 4 使用 Firefox 的最新版本。当测试套件运行起来时，Selenium Grid 会接收到每个测试用例及其对应浏览器的组合信息，并分配每个测试用例去测试其对应浏览器。

除此之外，对于相同类型和版本的浏览器来建立测试矩阵也是可行的。假设虚拟机测试矩阵中的四台机器都分别打开了三个 Firefox 浏览器，这就形成了可用的 Firefox 测试矩阵，总计 12 个 Firefox 浏览器实例。当测试套件运行起来时，Selenium Grid 会接收到每个测试用例的信息并分配每个测试用例去测试可用的 Firefox 实例。想象一下 12 个测试用例并行运

行，这可大大减少测试的运行时间。

Selenium Grid 的使用相当灵活，以上所列举的并行测试浏览器的例子也可结合起来使用，用于测试每种类型和版本的浏览器的多个实例。

★9.1.3 Selenium Grid 2.0 & 1.0

Selenium Grid 2.0 是目前的最新版本。Selenium Grid 2.0 与 Selenium Grid 1.0 大不相同。2.0 版本中 Selenium Grid 与 Selenium RC 服务器合并了。下载 selenium-server-standalone.jar 文件即可获得同时包含 Selenium RC 与 Selenium Grid 的文件包。

Selenium Grid 1.0 是 Selenium Grid 的第一个版本。如果是初次使用 Selenium Grid，建议使用最新版本 Selenium Grid 2.0。2.0 版本相较于 1.0 版本有很多更新，并支持 Selenium WebDriver。

9.2 Selenium Grid 的架构

Selenium Grid 包含一个 Hub 和至少一个 Node，两者都可由 selenium-server-standalone.jar 文件启动。后面将列举相关例子说明。

Hub 会接收到即将被执行的测试用例及其相关信息，即测试用例将在哪种浏览器和操作系统上运行。Hub 将记录每个“注册过”的 Node 的配置信息，并能通过这些信息自动选择可用的且符合浏览器与平台搭配要求的 Node。Node 被选中后，测试用例所调用的 Selenium 命令就会被发送至 Hub，Hub 再将这些命令发送到指定给该测试用例的 Node。随即 Node 开始启动浏览器，并执行这些 Selenium 命令对指定的 Web 程序或 Native 程序进行测试。

图 9.1 所示诠释了 Selenium Grid 的架构。

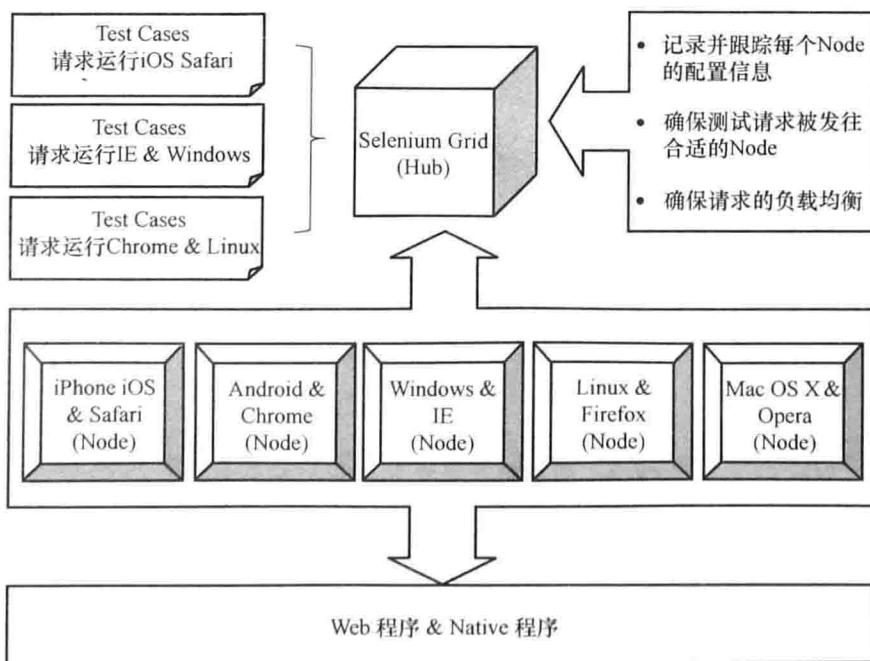


图 9.1 Selenium Grid 架构



9.3 Selenium Grid 的部署

1. 安装 Selenium Grid

Selenium Grid 的安装简单易行。从 Selenium 官方网站下载 selenium-server-standalone.jar 文件即可。Java 可执行文件的路径必须正确，这样才能从命令行运行文件。如果运行出错，检查系统的路径变量看其是不是已经包括 Java 可执行文件的路径。

2. 启动 Selenium Grid

一般而言，运行 Selenium Grid 需要先启动 Hub，因为 Node 的调用依赖于 Hub。然而这并不是意味着只能这样，因为 Node 在 Hub 启动时能进行识别，同时 Hub 也能在 Node 被启动时进行识别。如果是出于学习的目的，建议还是先启动 Hub，否则在第一次使用 Selenium Grid 时会出现错误消息。

9.4 Selenium Grid Hub

★9.4.1 默认启动 Hub

用下面的命令可启动默认设置的 Hub：

```
$ java -jar selenium-server-standalone-2.37.0.jar -role hub
```

所有可支持操作系统，如 Windows、Linux、Mac OS 都能调用该命令。需要注意的是，根据 selenium-server-standalone 版本的不同，jar 文件名里的版本号需要有相应改动。Hub 启动的默认端口为 4444，用户可以通过-port 参数来自定义其启动端口。

可以通过在浏览器中输入如下地址来检查 Hub 是否启动成功：

```
http://localhost:4444/grid/console
```

如果 Hub 启动成功，浏览器显示如图 9.2 所示。

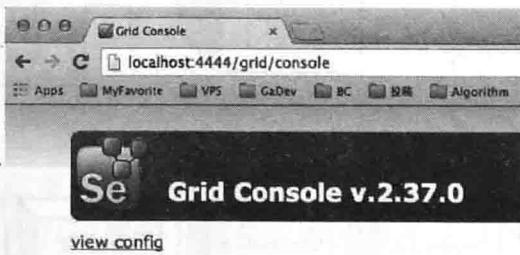


图 9.2 通过浏览器查看 Hub 启动状态

单击 view config 可以查看 Selenium Grid 的配置信息，如图 9.3 所示。默认情况下，单个 Hub 支持的最大会话数为 5 个。

★9.4.2 配置 Hub 端口

Hub 使用的默认端口是 4444。当自动化测试用例连接到 Selenium Grid Hub 时，监听的

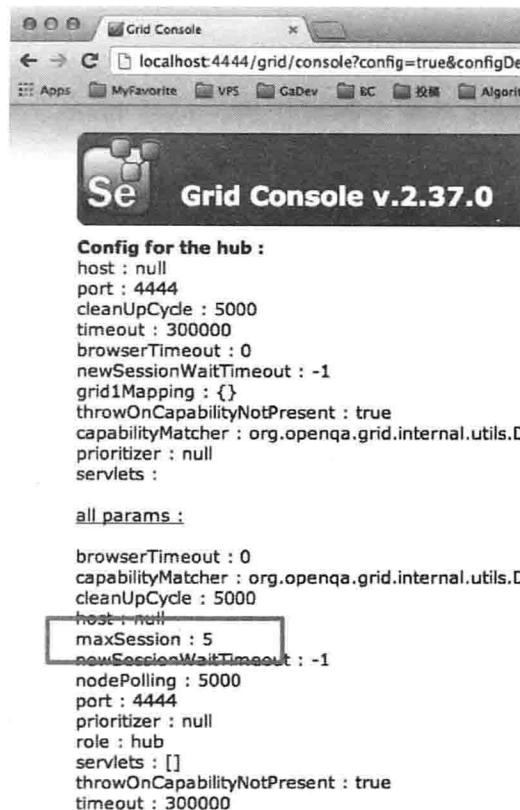


图 9.3 查看 Selenium Grid 的配置信息

端口就是 TCP/IP 端口。如果机器上已经有另一个程序在使用这个端口，或者 selenium-server-standalone 已经被启动，日志输出会显示以下消息提示端口已经被占用而无法启动 Selenium Grid Hub：

```
$ java -jar selenium-server-standalone-2.37.0.jar -role hub
Dec 21, 2013 9:58:38 PM org.openqa.grid.selenium.GridLauncher main
INFO: Launching a selenium grid server
view config
... ...
图 9.4 [INFO] ContextHandler: started o.s.j.s.
ServletContextHandler{/ , null}
2013-12-21 21:58:44.700:WARN:osjuc.AbstractLifecycle:FAILED SocketConnector @
0.0.0.0:4444: java.net.BindException: Address already in use
java.net.BindException: Address already in use
```

解决方案之一是关闭正在使用 4444 端口的程序，解决方案之二是让 Selenium Grid 的 Hub 使用另一个端口。在命令行中用 -port 选项可更换 Hub 使用的端口：

```
$ java -jar selenium-server-standalone-2.37.0.jar -role hub -port 8888
```

该方法即使在已有 Hub 在机器上运行时也能奏效，只要这两个 Hub 所使用的端口不一样。

一般情况下可以让 Hub 使用默认端口。如果想知道机器上所有正在运行的程序使用哪



些端口，可用下面这个命令进行查看：

```
$ netstat -a
```

该命令在所有可支持的操作系统上都可作用，包括 Linux、Mac OS X 和 Windows。不过有时候可能需要另外加上-a 参数。

★9.4.3 JSON 配置文件

除了在启动命令中指定参数来定制 Hub 的配置，还可以通过预定义好的配置文件来启动 Hub。这些配置信息可以写在一个 JSON 格式的配置文件中。示例如下：

```
{  
    "host": null,  
    "port": 4444,  
    "newSessionWaitTimeout": -1,  
    "servlets": [ ],  
    "prioritizer": null,  
    "capabilityMatcher": "org.openqa.grid.internal.utils.DefaultCapabilityMatcher",  
    "throwOnCapabilityNotPresent": true,  
    "nodePolling": 5000,  
  
    "cleanUpCycle": 5000,  
    "timeout": 300000,  
    "browserTimeout": 0,  
    "maxSession": 5  
}
```

然后通过在启动命令中添加-hubConfig 参数来加载该 JSON 配置文件：

```
$ java -jar selenium-server-standalone-2.37.0.jar -role node  
hub-json-cfg.json
```

9.5 Selenium Grid Node

★9.5.1 默认启动 Node

通过如下命令可启动默认设置的 Node：

```
$ java -jar selenium-server-standalone-2.37.0.jar -role node -hub  
http://localhost:4444/grid/register
```

这里假设默认设置的 Hub 已经被启动。Hub 用来监听新请求的默认端口是 4444，所以定位 Hub 的 URL 中使用了 4444 这个端口。使用本地主机是假定 Node 和 Hub 在同一台机器

上运行。对于初学者来说，这种方式最简单易懂。如果在不同机器上运行 Node 和 Hub，则需要将本地主机名替换成运行 Hub 的机器的主机名。

在这种情况下，Node 会将本机操作系统所能支持的浏览器信息全部注册到 Hub 上，如图 9.4 所示。本机由于是 MAC OS X，不支持 Internet Explorer 浏览器，在 Node 启动的日志中可以看到如下提示信息：

```
INFO - Default driver org.openqa.selenium.ie.InternetExplorerDriver registration is
skipped: registration capabilities Capabilities [ {platform = WINDOWS, ensureCleanSession =true, browserName =internet explorer, version =} ] does not match with current platform: MAC
```

如果将参数-role node 替换成-role webdriver，则表示该 Node 只兼容 WebDriver 的执行模式。相对应地，如果替换成-role rc，则表示该 Node 只兼容 Remote Control 的执行模式。

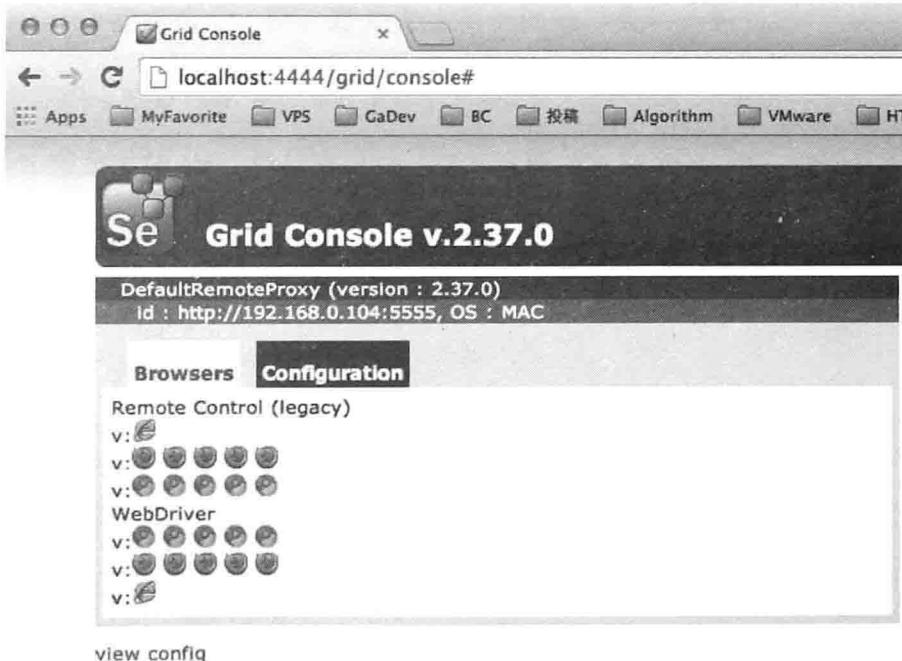


图 9.4 查看 Node 注册到 Hub 的信息

★9.5.2 注册 Mac OS X & Opera

接下来尝试一下在 Hub 上注册多个不同的 Node。首先以 Mac OS X 操作系统和 Opera 浏览器为组合进行阐述。

有至少两种方式将 Node 注册到 Hub 上。第一种方式是通过命令行的方式添加，只需要加上相应的参数即可。参数包括 browserName、version、maxinstance、platform 等。

```
$ java -jar selenium-server-standalone-2.37.0.jar -role node -browser
"seleniumProtocol = WebDriver, browserName = opera, version =15, maxInstances =
1, platform =MAC" -hubHost localhost
```



注册成功后，从 Hub 所在机器的浏览器中打开以下地址：

`http://localhost:4444/grid/console`

可以看到注册上去的 Node 信息，如图 9.5 所示。OS 为 MAC，执行模式为 WebDriver，浏览器类型为 Opera，浏览器版本为 15。



图 9.5 MAC OS X & Opera 注册到 Hub 后的 Node 信息

第二种方式是创建一个 JSON 格式的配置文件，并且在启动 Node 时加载该 JSON 配置文件。

```
{  
    "class": "org.openqa.grid.common.RegistrationRequest",  
    "capabilities": [  
        {  
            "seleniumProtocol": "WebDriver",  
            "browserName": "opera",  
            "version": "15",  
            "maxInstances": 1,  
            "platform": "MAC"  
        }  
    ],  
    "configuration": {  
        "proxy": "org.openqa.grid.selenium.proxy.DefaultRemoteProxy",  
        "maxSession": 5,  
        "port": 5555,  
        "host": "localhost",  
        "register": true,  
        "registerCycle": 5000,  
        "hubPort": 4444,  
        "hubHost": "localhost",  
        "nodePolling": 3000  
    }  
}
```

利用此 JSON 配置文件的启动命令如下：

```
$ java -jar selenium-server-standalone-2.37.0.jar -role node -nodeConfig
node-json-mac-opera.cfg.json
```

通过 JSON 配置文件将 Node 注册到 Hub，与在启动参数中指定 Node 配置信息的执行效果一致。

★9.5.3 注册 Linux & Firefox

本示例展示了以 Linux 操作系统和 Firefox 浏览器为组合的 Node 如何注册到 Hub。如果系统中存在多个浏览器版本，则需要指定浏览器的可执行程序路径。本示例设定的配置项如下：

- 1) 执行模式为 WebDriver。
- 2) 以 Firefox 最新版本的路径 firefox_binary 为例进行阐述。
- 3) maxInstances 设置为 2，表示最大支持两个浏览器实例的执行。

```
$ java -jar selenium-server-standalone-2.37.0.jar -role node -browser
"seleniumProtocol = WebDriver, browserName = firefox, version = 25, firefox_
binary = /home/selenium2/firefox25/firefox maxInstances = 2, platform = LINUX" -
hubHost 192.168.0.104
```

如图 9.6 所示，Linux Node 中包含两个 Firefox 浏览器的图标，即表示最多支持两个 Firefox 浏览器实例的执行。



图 9.6 Linux & Firefox 注册到 Hub 后的 Node 信息

★9.5.4 注册 Windows & Internet Explorer

本示例以 Windows 操作系统和 Internet Explorer 浏览器为组合进行阐述，其中：

- 1) seleniumProtocol 设置为 Selenium，表示执行模式为 Remote Control (legacy)。
- 2) maxInstances 设置为 4，表示浏览器启动的实例最多为 4 个。

完整的命令行如下：





```
C:\Selenium2 > java -jar selenium-server-standalone-2.37.0.jar -role node -browser
"seleniumProtocol = Selenium, browserName = iexplore, version = 10,
maxInstances = 4, platform = WINDOWS" -hubHost 192.168.0.104
```

如图 9.7 所示，Windows Node 中包含 4 个 Internet Explorer 浏览器的图标，即表示最多支持 4 个 Internet Explorer 浏览器实例的执行。

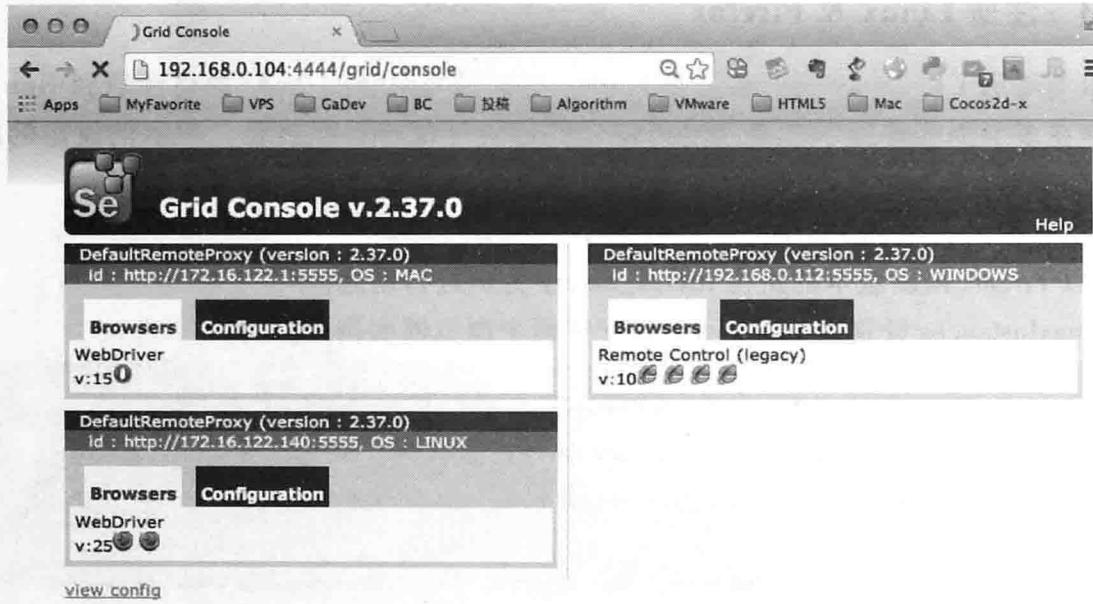


图 9.7 Windows & Internet Explorer 注册到 Hub 后的 Node 信息

★9.5.5 注册 Android & Chrome

本示例以 Android 和其自带的 Chrome 浏览器为组合进行阐述。

正如前面所涉及的 Android WebDriver 环境的搭建，需要首先设置转发接口，如下命令所示。其作用是确保主机与 Android 模拟器或者真机可以进行通信。

```
$ ./adb -s <serialId> forward tcp:8080 tcp:8080
```

如果是在 Linux 环境下操作，如 Ubuntu 操作系统，可通过如下命令安装 socat 并设置端口转发。其作用是确保在另外一台机器上也可以访问当前机器上的 Android WebDriver Server。

```
$ sudo apt-get install socat
```

```
$ socat TCP-LISTEN:8889,fork TCP:localhost:8080
```

通过上面的设置，就可以在网络上任意一个地方通过如下地址来访问 Android WebDriver Server 了：

```
http://hostname:8889/wb/hub
```

接下来注册 Android 和自带 Chrome 浏览器组合的 Node 到 Hub，命令如下。其中端口

为 8889。

```
$ java -jar selenium-server-standalone-2.37.0.jar -role node -browser
"seleniumProtocol = WebDriver, browserName = android, version = 4, maxInstances =
1, platform = ANDROID" -port 8889 -hubHost 192.168.0.104
```

如图 9.8 所示，注册成功后的 Node 信息显示在 Hub 的 Console 中。可以看到那个可爱的绿色 Android 小机器人。

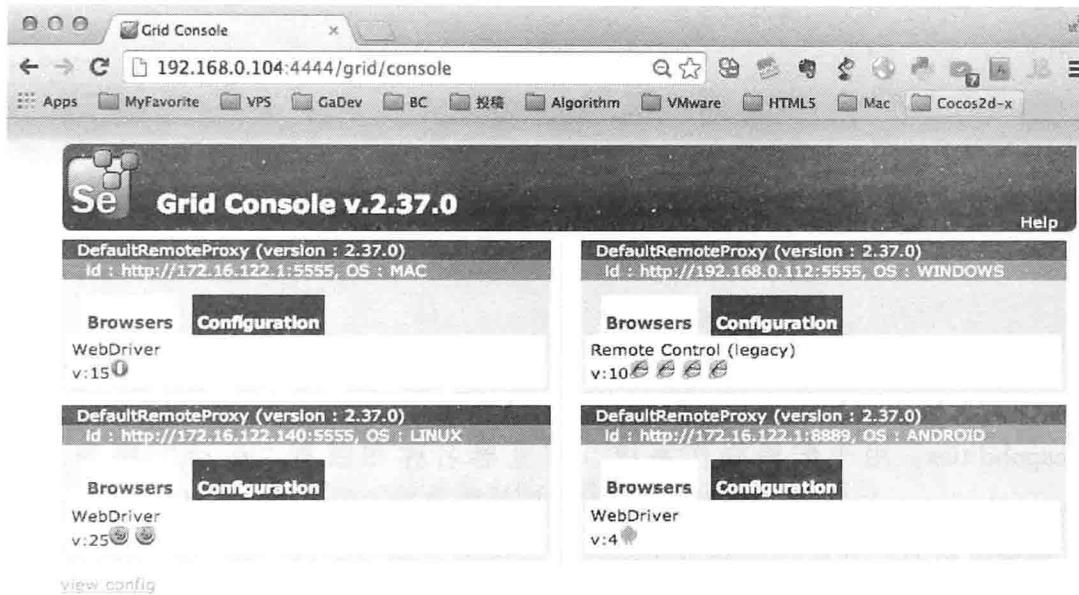


图 9.8 Android & Chrome 注册到 Hub 后的 Node 信息

★9.5.6 注册 Appium-iOS & Safari

针对 iOS 平台，在本书中重点关注如何通过 Appium 这个框架注册 Node 到 Hub 上。可以编写一个 JSON 的配置文件，包括相关的操作系统信息、浏览器类型和版本信息等，然后通过 Appium 的命令行方式将该 JSON 配置文件加载进去。

JSON 配置文件的范例如下：

```
{
  "capabilities": [
    {
      "browserName": "safari",
      "version": "7.0",
      "maxInstances": 1,
      "platform": "MAC"
    }
  ],
  "configuration": {}
```



```
{  
    "cleanUpCycle": 2000,  
    "timeout": 30000,  
    "proxy": "org.openqa.grid.selenium.proxy.DefaultRemoteProxy",  
    "url": "http://192.168.0.104:4723/wd/hub",  
    "maxSession": 1,  
    "port": 4723,  
    "host": "192.168.0.104",  
    "register": true,  
    "registerCycle": 5000,  
    "hubPort": 4444,  
    "hubHost": "192.168.0.104"  
}  
}  
}
```

JSON 配置文件主要包括两个部分：

- 1) capabilities：用于配置操作系统、浏览器名称和版本、运行实例等信息。由于 org.openqa.selenium.platform 中目前并没有官方直接支持 iOS 类型，所以 platform 使用 MAC 即可。
- 2) configuration：用于配置与 Hub 交互时的信息。

接下来通过如下命令和参数--nodeconfig 加载写好的 JSON 配置文件：

```
$ appium --nodeconfig node-json-appium-ios.cfg.json
```

如果加载成功，会看到如下打印信息：

```
info: Welcome to Appium v0.13.0  
info: Appium REST http interface listener started on 0.0.0.0:4723  
info: - socket.io started  
info: starting auto register thread for grid. Will try to register every 5000 ms.  
info: Appium successfully registered with the grid on 192.168.0.104:4444  
debug: Appium request initiated at /wd/hub/status  
debug: Request received with params: {}  
info: Responding to client with success: {"status":0,"value":{"build":  
{"version":"0.13.0","revision":"20c368f2963c73eb20cc3bca1585346c31fec387"}}}
```

如图 9.9 所示，DefaultRemoteProxy 的版本号是 0.13.0，正是本示例中所使用的 Appium 的版本号。其中，platform 为 MAC，Safari 浏览器的版本为 7.0（对应当前最新的 iOS 版本 7.0）。

★9.5.7 注册多个不同类型的浏览器

如果需要在一个 Node 上同时运行多个不同类型的浏览器实例，可以通过在 JSON 配置

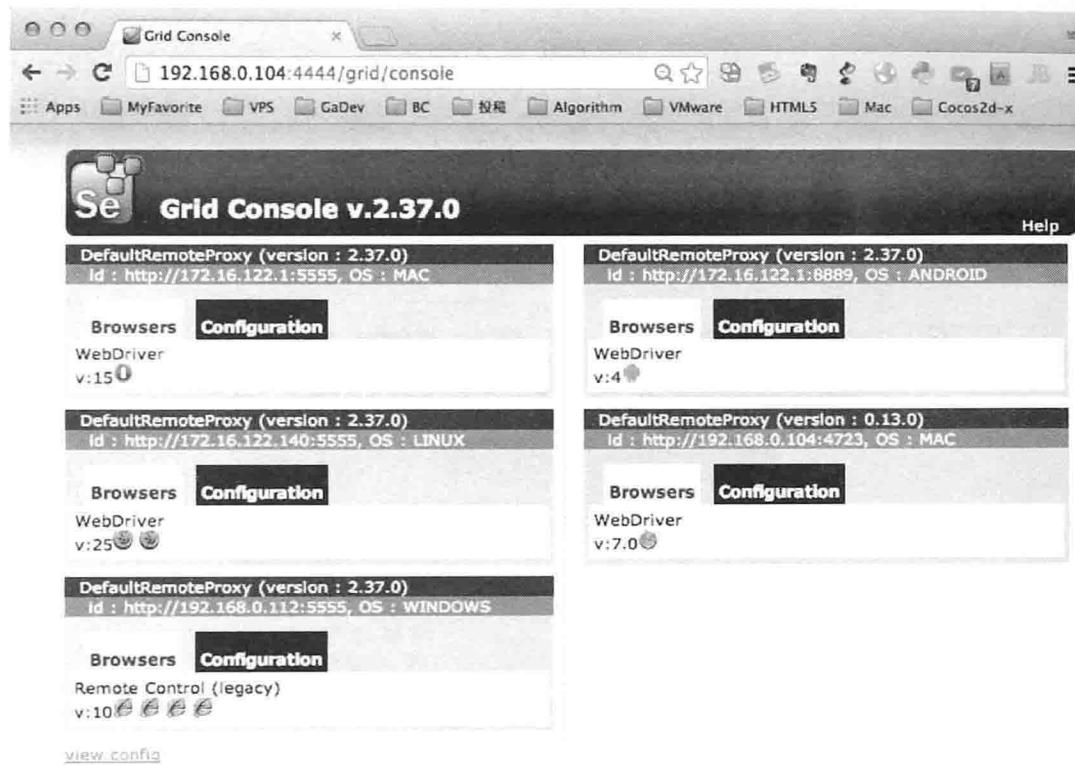


图 9.9 Appium-iOS & Safari 注册到 Hub 后的 Node 信息

文件中将所有能支持的浏览器信息定义进去，然后再通过命令行方式一次性加载并将相关信息注册到 Hub 上。

JSON 配置文件示例如下：

```
{
  "capabilities": [
    {
      "browserName": "firefox",
      "version": 23,
      "maxInstances": 5,
      "seleniumProtocol": "WebDriver"
    },
    {
      "browserName": "firefox",
      "version": 3.6,
      "maxInstances": 5,
      "seleniumProtocol": "WebDriver"
    },
    {
      "browserName": "chrome",
    }
  ]
}
```



```
"version": 28,
"maxInstances": 5,
"seleniumProtocol": "WebDriver"
},
{
"browserName": "opera",
"version": 15,
"maxInstances": 1,
"seleniumProtocol": "WebDriver"
},
{
"browserName": "safari",
"version": 7.0,
"maxInstances": 3,
"seleniumProtocol": "WebDriver"
}
],
"configuration":
{
"proxy": "org.openqa.grid.selenium.proxy.DefaultRemoteProxy",
"maxSession": 5,
"port": 5555,
"host": "192.168.0.104",
"register": true,
"registerCycle": 5000,
"hubPort": 4444,
"hubHost": "192.168.0.104"
}
```

该 JSON 配置文件一共定义了 4 种不同类型的浏览器，其中 Firefox 又分两个不同的高低版本。每个浏览器的最多运行实例上限亦不相同。

通过以下命令来加载该 JSON 配置文件，并同时注册到 Hub 上：

```
$ java -jar selenium-server-standalone-2.37.0.jar -role node -nodeConfig node-json-multiple-browsers.cfg.json
```

如图 9.10 所示，该 Node 上的操作系统为 MAC，可以支持的浏览器信息分别如下：

- 1) Chrome 浏览器，版本为 28，运行实例上限为 5 个。
- 2) Opera 浏览器，版本为 15，运行实例上限为 1 个。

- 3) Firefox 浏览器，版本为 3.6，运行实例上限为 5 个。
- 4) Firefox 浏览器，版本为 23，运行实例上限为 5 个。
- 5) Safari 浏览器，版本为 7，运行实例上限为 3 个。

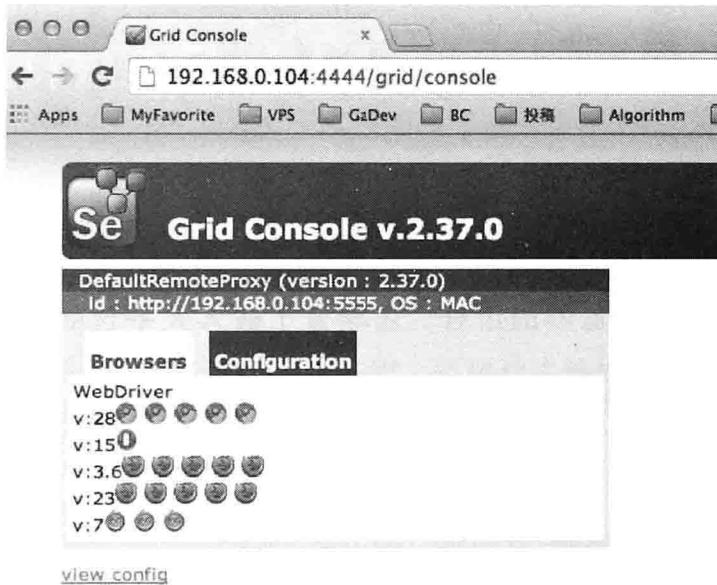


图 9.10 多个不同类型的浏览器注册到 Hub 后的 Node 信息

9.6 编写 Selenium Grid 的测试用例

针对 Selenium Grid 的特点，需添加如下代码来使已有的测试用例代码能运行在 Selenium Grid 环境中。

1) 对于执行模式为 WebDriver 的 Node，要使用 `RemoteWebDriver` 和 `DesiredCapatilities`。示例代码段如下：

```
DesiredCapabilities firefoxCap = DesiredCapabilities.firefox();

firefoxCap.setBrowserName("firefox");
firefoxCap.setVersion("25");
firefoxCap.setPlatform("LINUX");

WebDriver driver = new RemoteWebDriver(
    new URL("http://localhost:4444/wd/hub"), firefoxCap);
```

如上述代码所示，需通过 `DesiredCapabilities` 来设置测试用例适用的浏览器名称、浏览器版本和操作系统平台。这样就可以确保该测试用例被 Hub 分配到同时满足以上三个条件的 Node 上执行。

2) 对于执行模式为 Remote Control 的 Node，则需使用 `DefaultSelenium`。示例代码段如下：



```
Selenium selenium = new DefaultSelenium("localhost",
                                         4444,
                                         "* firefox",
                                         "http://www.baidu.com");
```

9.7 小结

本章介绍了什么是 Selenium Grid，及其架构和部署方法。分别介绍了 Hub 和 Node 的特点以及如何将两者结合起来使用，并针对不同的操作系统平台和不同的浏览器进行阐述。除了传统的操作系统平台和浏览器的组合，还涵盖了嵌入式平台的浏览器，包括 Android 和 iOS。

第 10 章

Selenium 的“兄弟姐妹们”

10.1 简介

Selenium 是本书介绍的主角，当然光芒四射。但要让主角的闪光点更为耀眼，配角的戏份也不能少。本章将介绍与 Selenium 相关的其他辅助性测试工具或者套件，为测试人员知识面的广泛性提供一些引导。

10.2 Jenkins

在当前流行的敏捷开发过程中，CI（Continuous Integration，持续集成）的方法和工具变得越来越重要。在众多的 CI 工具中，脱颖而出的就是这里要介绍的 Jenkins。而且不负众望，Selenium 和 Jenkins 集成得非常棒。

1) 到这里下载并安装 Jenkins：

<http://jenkins-ci.org/>

2) 单击 Jenkins Dashboard 上的 Manage Jenkins，如图 10.1 所示。



图 10.1 Manage Jenkins

3) 进入 Plugin Manager，切换到 Available 选项卡。然后在 Filter 文本框中输入 selenium plugin，在过滤列表中选中 Selenium Plugin 并进行安装，如图 10.2 所示。



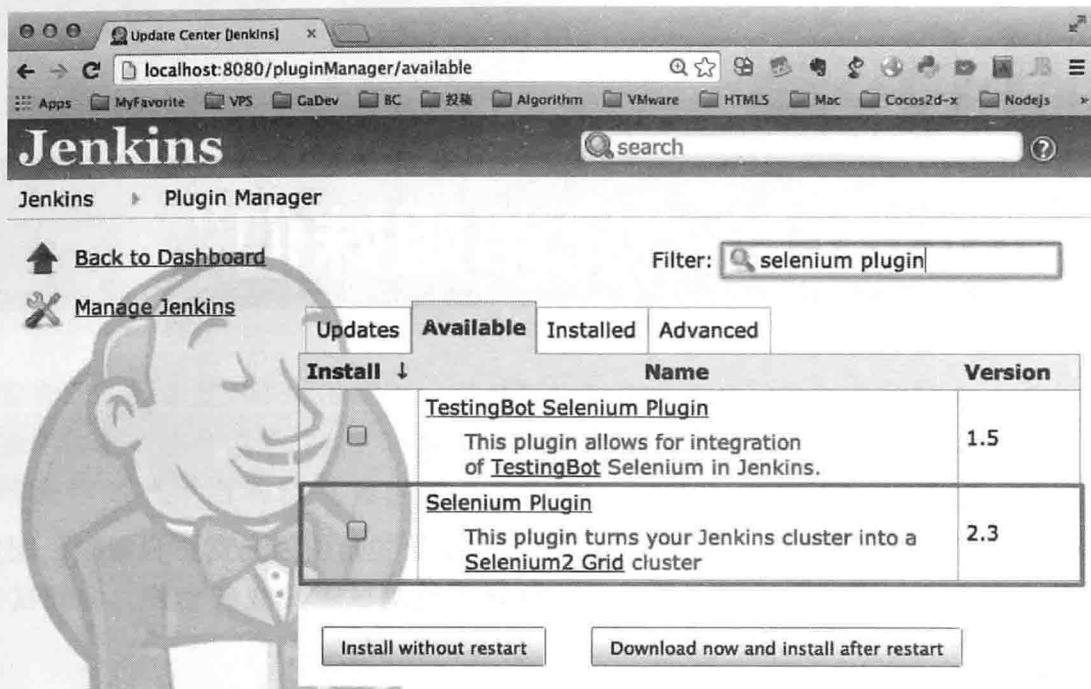


图 10.2 安装 Selenium Plugin

4) 安装 Selenium Plugin 成功后，在 Dashboard 界面会出现 Selenium Grid 的菜单选项，如图 10.3 所示。



图 10.3 Selenium Grid 菜单

5) 进入 Jenkins→Selenium Grid→Console Output，可以看到 Selenium Grid 已经随 Jenkins 启动，其版本为 2.29.0，监听端口为 4444，如图 10.4 所示。

6) 进入 Jenkins→Selenium Grid→Configuration→New configuration，新建 Node 配置文件并命名为 Selenium Grid-WebDriver。建立成功后的信息如图 10.5 所示。

7) 进入 Jenkins→Manage Jenkins→Manage Nodes→Master→Selenium Node Management，可以看到之前新建的 Selenium Grid-WebDriver，如图 10.6 所示。确认该 Node 的 Status 为 Started 状态，如果状态不是 Started，请单击 Start 按钮启动该 Node 使其 Status 变为 Started。

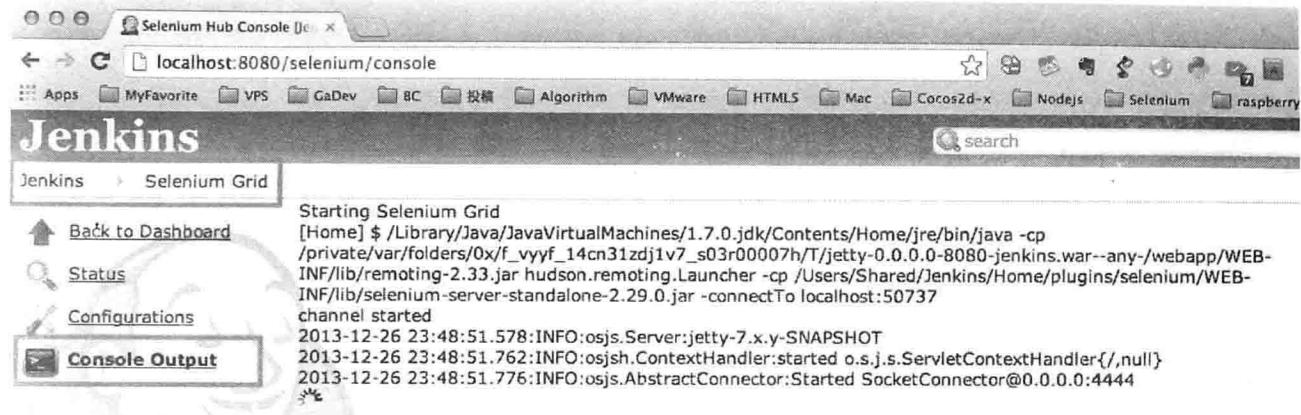


图 10.4 Selenium Grid 启动后的 Console Output

Type	Delete	Name	Matching type	Description summary
		Selenium Grid - WebDriver	Match all nodes	1 instances of Internet Explorer (version : Not specified) 5 instances of Firefox (version : Not specified) 3 instances of Safari (version : Not specified)

图 10.5 Selenium Grid 的 Configurations 信息

Name	Status	Environment variables	JVM options	Selenium options	Service actions
Selenium Grid - WebDriver	Started			-browser seleniumProtocol=WebDriver,browserName=internet explorer,maxInstances=1 -browser seleniumProtocol=WebDriver,browserName=firefox,maxInstances=5 -browser seleniumProtocol=WebDriver,browserName=safari,maxInstances=3	

图 10.6 Selenium Node Management

8) 再次在浏览器中打开如下地址, 如图 10.7 所示, 可以看到该 Node 的信息。

<http://localhost:4444/grid/console>

DefaultRemoteProxy

listening on http://192.168.0.42:5555

test session time out after 300 sec.

Supports up to 5 concurrent tests from:

[view config](#)

图 10.7 注册到 Hub 后的 Node 信息



关于 Jenkins Selenium Grid Plugin 的信息，请关注其官方页面：

<https://wiki.jenkins-ci.org/display/JENKINS/Selenium+Plugin>

9) 通过 Jenkins 添加更多的 Nodes 就可以完成组建完整的 Selenium Grid 的任务。

10.3 Web 前端性能

★10.3.1 BrowserMob Proxy

BrowserMob Proxy 是一个开源的工具，用于截取浏览器中页面加载性能相关的数据信息。截取的数据格式为 HTML Archive (HAR)，其本质上是 JSON 格式的文件，用来存储 HTTP 请求/响应的相关信息。HAR 满足了 HTTP 监测工具以一种通用的格式导出待收集的数据，这些数据可以被其他支持 HAR 的 HTTP 分析工具所使用，包括 Firebug、HttpWatch、Fiddler 等。通过这些收集到的数据，可以分析网站的 Web 前端性能瓶颈。HAR 文件必须是 UTF-8 编码，有无 BOM 并无影响。

结合主角 Selenium WebDriver，可以配合 BrowserMob Proxy 来收集 Web 前端的性能数据信息。

BrowserMob Proxy 的下载地址如下：

<https://github.com/lightbody/browsermob-proxy>

将下载后的 BrowserMob Proxy 文件夹中 lib 目录下的 JAR 文件均导入到已有的 Selenium WebDriver 项目中。

下面以打开百度主页为例进行阐述，示例代码如下：

```
package com.learningselenium.web.frontier.perf;

import java.io.File;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.remote.CapabilityType;
import org.openqa.selenium.Proxy;
import net.lightbody.bmp.proxy.ProxyServer;
import net.lightbody.bmp.core.har.Har;

import junit.framework.TestCase;

public class testBrowserMobProxy extends TestCase {
    public void testBaidu() throws Exception {
```



```
ProxyServer server = new ProxyServer(9090);
server.start();

Proxy proxy = server.seleniumProxy();

DesiredCapabilities capabilities = new DesiredCapabilities();
capabilities.setCapability(CapabilityType.PROXY, proxy);

WebDriver driver = new FirefoxDriver(capabilities);

server.newHar("baidu.com");

driver.get("http://www.baidu.com");
Har har = server.getHar();

File harFile = new File("/Selenium2/HAR-baidu.com.json");
har.writeTo(harFile);

server.stop();

driver.quit();
}

}
```

示例代码详解：

- 1) 启动 BrowserMob Proxy，并设定端口为 9090。
- 2) 获取 Selenium proxy 对象。
- 3) 配置 DesiredCapabilities 为 CapabilityType. PROXY。
- 4) 启动 WebDriver。
- 5) 新建 HAR 并命名为 baidu.com。示例代码段如下：

```
server.newHar("baidu.com");
```

- 6) 打开百度主页并获取 HAR 数据。示例代码段如下：

```
driver.get("http://www.baidu.com");
Har har = server.getHar();
```

- 7) 将获取到的 HAR 数据写入文件 HAR-baidu.com.json。示例代码段如下：

```
File harFile = new File("/Selenium2/HAR-baidu.com.json");
har.writeTo(harFile);
```





8) 停止 BrowserMob Proxy 并退出 WebDriver。

接下来打开如下 chromeHAR 地址：

<http://ericduran.github.io/chromeHAR/>

将生成的 HAR-baidu.com.json 文件拖放进去，可以查看数据加载的时间消耗，以此来评估 Web 前端的性能好坏，如图 10.8 所示。

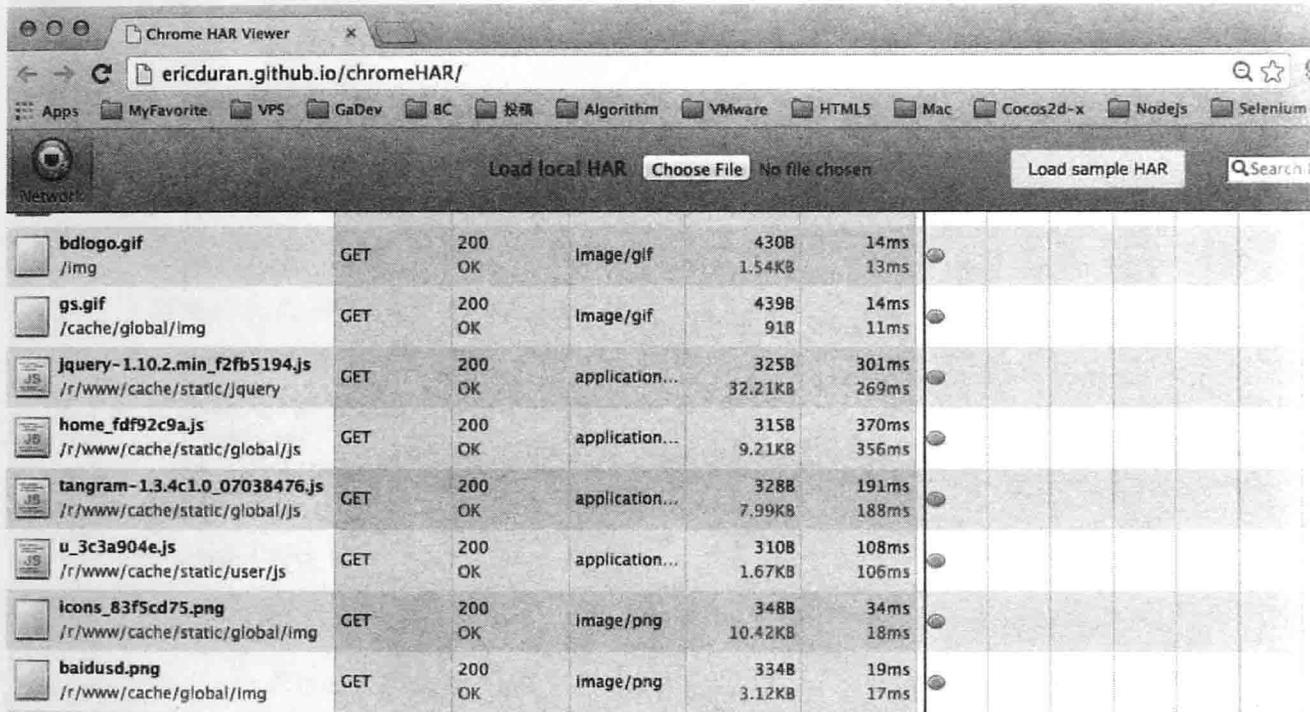


图 10.8 通过 HAR 评估 Web 前端的性能

★10.3.2 HttpWatch

这里阐述的 HttpWatch 是 iOS 系统上的版本。在 App Store 上搜索 HttpWatch 即可，软件安装界面如图 10.9 所示。这里会利用 7.3.3 节中的 Appium 在 iOS 真实设备上启动 SafariLauncher，进而启动 HttpWatch 来测试网站的前端性能。

如果希望通过代码来启动 iOS 真实设备上的 HttpWatch 软件来打开待测试页面，则需要在待测试页面的 URL 前面添加 hw，如下所示：

将“<http://>”调整为“<hwhttp://>”，将“<https://>”调整为“<hwhttps://>”。

结合 7.3.3 节中的示例代码，唯一需要增加的工作量是将

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

调整为

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

以 Google 主页为例的完整测试代码如下，细节不再赘述，请参考 7.3.3 节的示例代码详解。



图 10.9 iOS 版本的 HttpWatch

```
package com.learningselenium.web.frontier.perf;
```

```
import java.net.URL;  
import java.util.concurrent.TimeUnit;  
  
import org.openqa.selenium.remote.DesiredCapabilities;  
import org.openqa.selenium.remote.RemoteWebDriver;  
  
import junit.framework.TestCase;  
  
public class testIOSAppiumHttpWatch extends TestCase {  
    public void testGoogle() throws Exception {  
        DesiredCapabilities safariCapabilities =  
            new DesiredCapabilities();  
        safariCapabilities.setCapability("device", "iphone");  
    }  
}
```



```
safariCapabilities.setCapability("version", "7.0");  
safariCapabilities.setCapability("app", "safari");  
  
RemoteWebDriver driver = new RemoteWebDriver(  
    new URL("http://localhost:4723/wd/hub"), safariCapabilities);  
  
driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);  
  
driver.get("http://www.google.com");  
  
driver.close();  
}  
}
```

HttpWatch 会记录打开页面上元素加载的性能，如图 10.10 所示。HttpWatch 的开发团队也正在继续给 iOS 版本的 HttpWatch 添加更多的自动化接口，方便开发人员调用和调试，包括日志的分析等功能。



图 10.10 在 iOS HttpWatch 中查看页面元素加载性能



10.4 Ruby的光芒

★10.4.1 Watir-WebDriver

Watir (Web Application Testing in Ruby) 是一个用于 Web 自动化的 Ruby 库。它能够驱动各主流浏览器。而 Watir-WebDriver 是基于 Selenium WebDriver 的改良版本。Jari Bakken 通过将 Selenium 2.0 的 API 封装成 Ruby 语言版本的 Watir API 来达到目的。其官方网址如下：

```
http://watirwebdriver.com/
```

1. 安装 Watir-WebDriver

```
$ sudo gem install watir-webdriver
```

注意：

在 Mac 系统上安装 watir-webdriver 时，要确保系统中已经安装了 Command Line Tools。如果没有，则通过如下命令进行安装：

```
$ xcode-select -install
```

或者到苹果官网的开发者社区下载 Command Line Tools 的 pkg 包进行安装。注意安装后要重启系统。接下来可以继续安装 watir-webdriver。

注意：

如果遇到如下错误提示信息：

```
mkmf.rb can't find header files for ruby at /System/Library/Frameworks/Ruby.framework/Versions/2.0/usr/lib/ruby/include/ruby.h
```

则通过建立软链接的方式来解决：

```
$ sudo ln -s /System/Library/Frameworks/Ruby.framework/Versions/2.0/usr/include/ruby-2.0.0 /System/Library/Frameworks/Ruby.framework/Versions/2.0/usr/lib/ruby/include
```

然后可以继续安装 watir-webdriver。



2. Watir-WebDriver 的示例

下面以打开百度主页，并在搜索框中输入 watir-webdriver demo 关键字，然后进行搜索为例来初步展示使用 watir-webdriver 的 Ruby 代码基本结构。

```
require 'watir-webdriver'

browser = Watir::Browser.new :firefox
browser.goto "http://www.baidu.com/"

browser.text_field (:id => "kw").set ("watir-webdriver demo")
browser.button (:id => "su").click

browser.close
```

示例代码详解：

- 1) 导入 watir-webdriver 库。
- 2) 指定 Firefox 浏览器来完成后续工作。
- 3) 启动 Firefox 并打开百度主页，然后完成后续输入关键字并进行搜索的工作。
- 4) 任务完成后关闭浏览器。

★10.4.2 Capybara

Capybara 也是 Ruby 用户经常用到的模拟用户与应用程序之间交互的自动化工具。它一般使用 Ruby 编写的领域专用语言（Domain Specific Language，DSL）来描述测试用例。其官方主页如下：

<https://github.com/jnicklas/capybara>

安装 Capybara：

```
$ sudo gem install capybara
```

典型的测试代码如下：

```
require "capybara/dsl"

Capybara.app_host = 'http://www.google.com'
Capybara.default_wait_time = 15

Capybara.current_driver = :selenium

include Capybara::DSL

visit "/"
```

```

fill_in "q", :with => "Capybara"
find("#gbqfbw button").click

page.should have_content 'Capybara'

```

示例代码详解：

- 1) 加载 Capybara 的 DSL 库。
- 2) 设置待测试页面为 Google 主页和等待延时。
- 3) 设置测试驱动为 Selenium WebDriver。
- 4) 将 Capybara::DSL 引入进来。
- 5) 打开测试页面并搜索 Capybara。
- 6) 验证页面上的内容包含有 Capybara 关键字。

10.5 JMeter

JMeter 是 Apache 基金会开发的基于 Java 语言的性能测试工具。JMeter 可以模拟巨大的负载，或模拟不同类型的压力从而对被测试对象进行性能测试。其官方主页如下：

<http://jmeter.apache.org/>

如果希望在 JMeter 中使用 Selenium WebDriver，则可以通过安装 WebDriver 插件的方式实现。其下载地址如下：

<http://jmeter-plugins.org/downloads/all/>

如图 10.11 所示，下载相应的 JMeter 版本即可。



图 10.11 下载 JMeterPlugins-WebDriver

将下载的 JMeterPlugins-WebDriver.zip 文件解压缩后的/lib 中的所有文件放置到 JMeter 的/lib 目录下。接下来启动 JMeter 并开始示例。

在 JMeter 中使用 WebDriver 插件的步骤如图 10.12 所示。具体步骤包括：

- 1) 在 Test Plan 中新建 Thread Group。
- 2) 添加 Config Element→HTTP Cookie Manager。
- 3) 添加 Config Element→jp@gc-Firefox Driver Config。



- 4) 添加 Sampler→jp@gc-WebDriver Sampler。
- 5) 添加 Listener→View Results Tree。

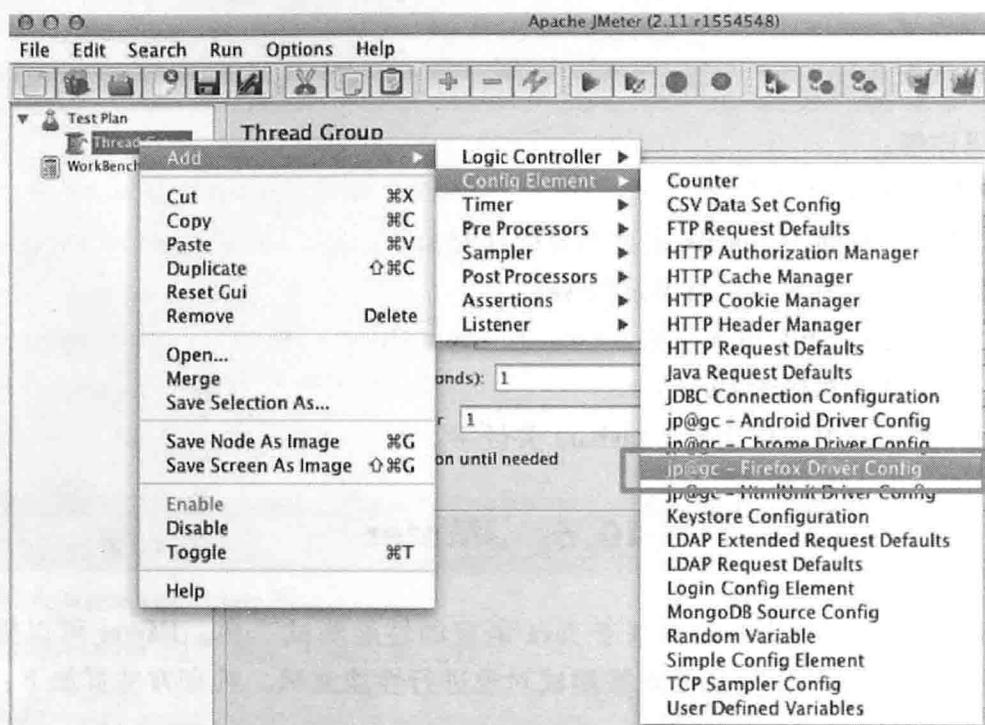


图 10.12 在 JMeter 中添加 WebDriver 插件

按步骤建立好的 Test Plan 如图 10.13 所示。

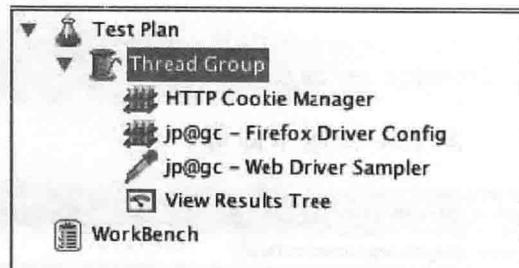


图 10.13 包含 WebDriver 的 Test Plan

接下来在 jp@gc-Web Driver Sampler 中添加测试代码，如图 10.14 所示。

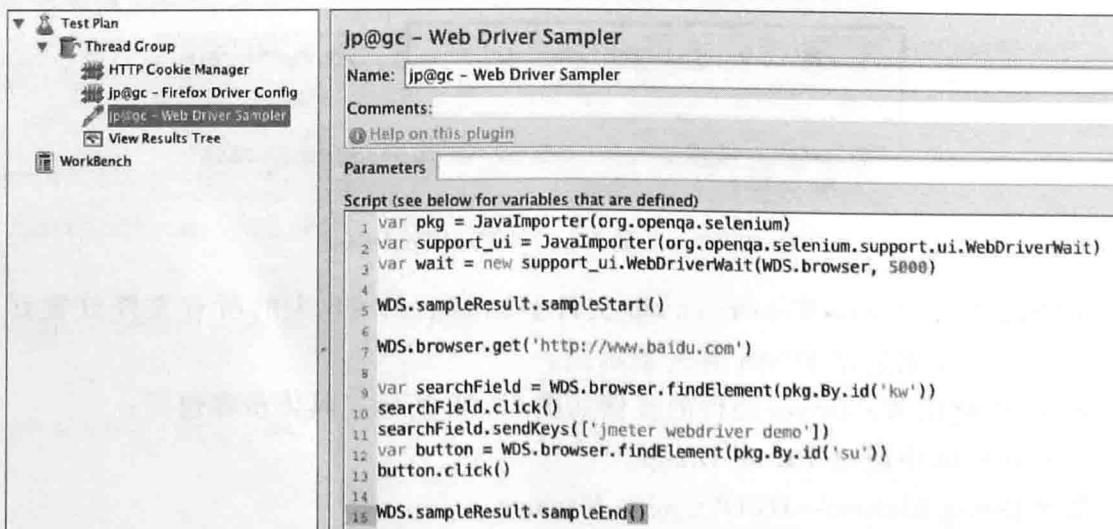


图 10.14 Web Driver Sampler 中的测试代码



完整测试代码如下：

```

var pkg = JavaImporter(org.openqa.selenium)
var support_ui = JavaImporter(org.openqa.selenium.support.ui.WebDriverWait)
var wait = new support_ui.WebDriverWait(WDS.browser, 5000)

WDS.sampleResult.sampleStart()

WDS.browser.get('http://www.baidu.com')

var searchField = WDS.browser.findElement(pkg.By.id('kw'))
searchField.click()
searchField.sendKeys(['jmeter webdriver demo'])
var button = WDS.browser.findElement(pkg.By.id('su'))
button.click()

WDS.sampleResult.sampleEnd()

```

示例代码详解：

- 1) 导入 Selenium 的 Java 包并设置延时。
- 2) WDS.sampleResult.sampleStart() 和 WDS.sampleResult.sampleEnd() 用于跟踪 sampler 的执行时间。
- 3) 打开待测试页面地址，此处为百度主页。
- 4) 在搜索栏中输入关键字并单击搜索按钮进行搜索。
- 5) 测试结束。

接下来就可以通过打开 View Result Tree 来查看页面的相关性能，如图 10.15 所示。本例在百度中进行搜索为例的页面加载时间为 7595ms。

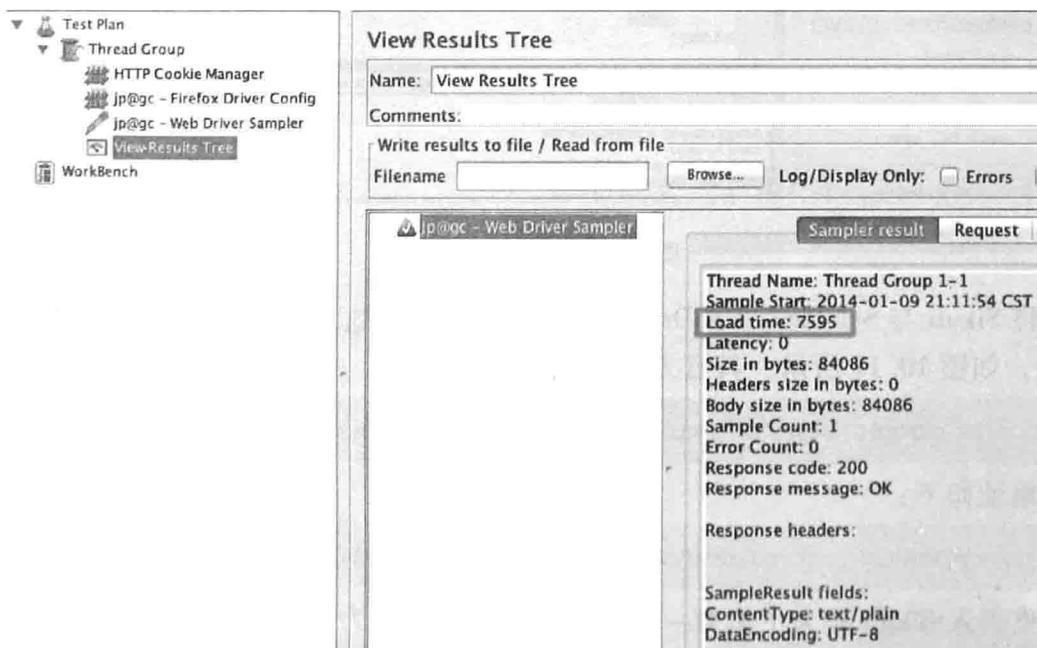


图 10.15 通过 View Result Tree 查看页面的相关性能



10.6 Sikuli

20世纪80年代以前，使用计算机软件需要记忆大量的命令才能完成相应的任务。随后出现了图形用户界面（Graphical User Interface，GUI），用户只需要借助直观的按钮或其他视觉元素即可操作计算机。但GUI的出现并没有给程序员带来根本性的便利，因为他们还是需要通过编写代码来实现相应功能。即使让不同的软件相互配合，也需要进行编码才能完成调用工作。

而Sikuli的出现使这一过程变得简单而有趣，只需要略懂一点编程语言的用户就可以完成简单的编程和程序之间的调用。通过Sikuli，用户可以利用截图来轻易完成程序的操作和互相调用，用最少的代码量来完成所需操作。

Sikuli的官方主页在这里：

<http://www.sikuli.org/>

已经有不少测试机构利用这一工具来完成功能测试，其IDE的界面如图10.16所示。

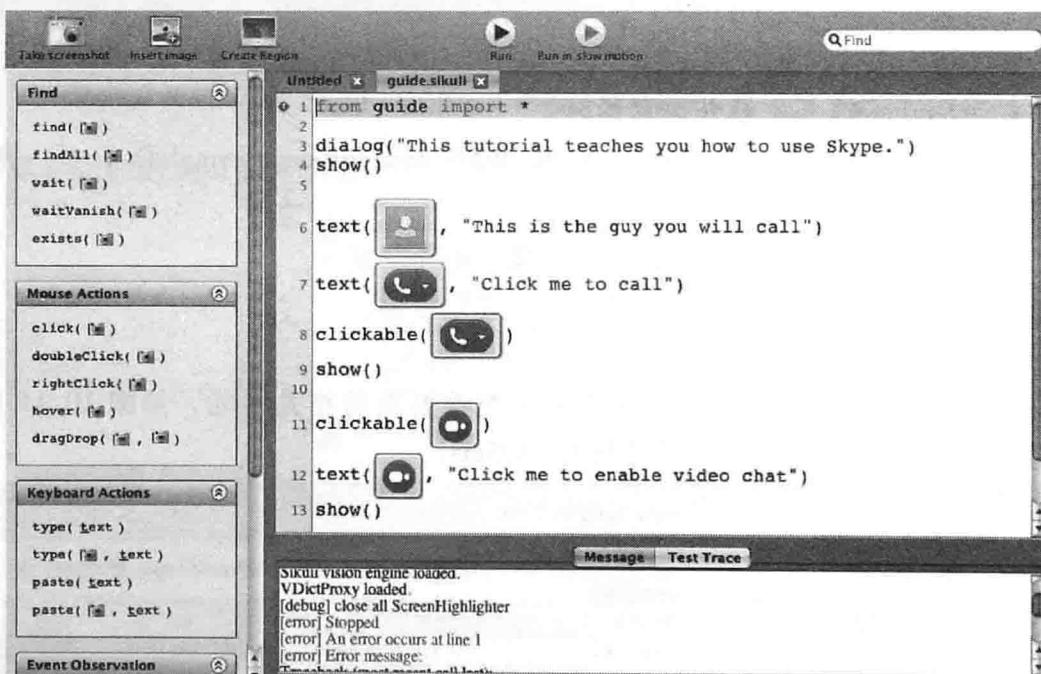


图10.16 Sikuli的主界面

为了将Sikuli与Selenium WebDriver结合起来使用，这里推荐一个解决方案，即Sikuli-WebDriver，如图10.17所示。其官方地址如下：

<https://code.google.com/p/sikuli-api/wiki/SikuliWebDriver>

下载地址如下：

<http://mvnrepository.com/artifact/org.sikuli/sikuli-webdriver>

在这里引入Sikuli是为了拓宽一下读者的视野，不作为本书的阐述对象。在其官方主页上亦给出了如何与Google地图进行交互的示例代码和详细说明，本书不再赘述。



图 10.17 SikuliFirefoxDriver 的组成

10.7 小结

本章介绍了围绕 Selenium 的一些其他的测试工具和套件，范畴涵盖了持续集成开发工具、Web 前端性能评估、Ruby 自动化测试套件、性能测试工具等，为测试人员拓宽测试思路起到了一定的导向作用。

参 考 文 献

- [1] 温素剑. 零成本实现 Web 自动化测试——基于 Selenium 和 Bromine [M]. 北京: 电子工业出版社, 2011.
- [2] 关春银, 王林, 周晖, 吴付华. Selenium 测试实践——基于电子商务平台 [M]. 北京: 电子工业出版社, 2011.
- [3] 朱少民. 轻轻松松自动化测试 [M]. 北京: 电子工业出版社, 2009.
- [4] Unmesh Gundecha. Selenium Testing Tools Cookbook [M]. Birmingham: Packet Publishing Ltd. , 2012.
- [5] David Burns. Selenium 2 Testing Tools—Beginner's Guide [M]. Birmingham: Packet Publishing Ltd. , 2012.
- [6] Roy de Kleijn. Learning Selenium [M]. Vancouver: Lean Publishing, 2013.
- [7] Bayo Erinle. Performance Testing with JMeter 2.9 [M]. Birmingham: Packet Publishing Ltd. , 2013.
- [8] Diego Torres Milano. Android Application Testing Guide [M]. Birmingham: Packet Publishing Ltd. , 2013.
- [9] Cédric Beust Hani Suleiman. Next Generation Java Testing: TestNG and advanced concepts [M]. Boston: Pearson Education, Inc. , 2007.
- [10] Matthew Robbins. Application Testing with Capybara [M]. Birmingham: Packet Publishing Ltd. , 2013.
- [11] John Ferguson Smart. Jenkins: The Definitive Guide [M]. Sebastopol: O'Reilly Media, Inc. , 2011.
- [12] Mark Lutz. Learning Python, Fifth Edition [M]. Sebastopol: O'Reilly Media, Inc. , 2013.
- [13] Satya Avasarala. Selenium WebDriver Practical Guide [M]. Birmingham: Packet Publishing Ltd. , 2014.
- [14] Simon Monk. Raspberry Pi Cookbook [M]. Sebastopol: O'Reilly Media, Inc. , 2014.
- [15] Michael Bolin. Closure: The Definitive Guide [M]. Sebastopol: O'Reilly Media, Inc. , 2010.
- [16] Christopher Schmitt, Kyle Simpson. HTML5 Cookbook [M]. Sebastopol: O'Reilly Media, Inc. , 2011.
- [17] Adam Freeman. Pro jQuery 2.0 [M]. New York: Apress Media LLC, 2013.
- [18] Petar Tahchiev, Felipe Leme, Vincent Massol, et al. JUnit in Action: Second Edition [M]. Stamford: Manning Publications Co. , 2011.
- [19] Simon Stewart. Selenium WebDriver [J/OL]. The Architecture of Open Source Applications. <http://aos-book.org/en/selenium.html>, 2014.
- [20] Suarez Ordoñez, Santiago. Official documentation of Selenium [J/OL]. Selenium Documentation. <http://docs.seleniumhq.org/docs/>, 2014.

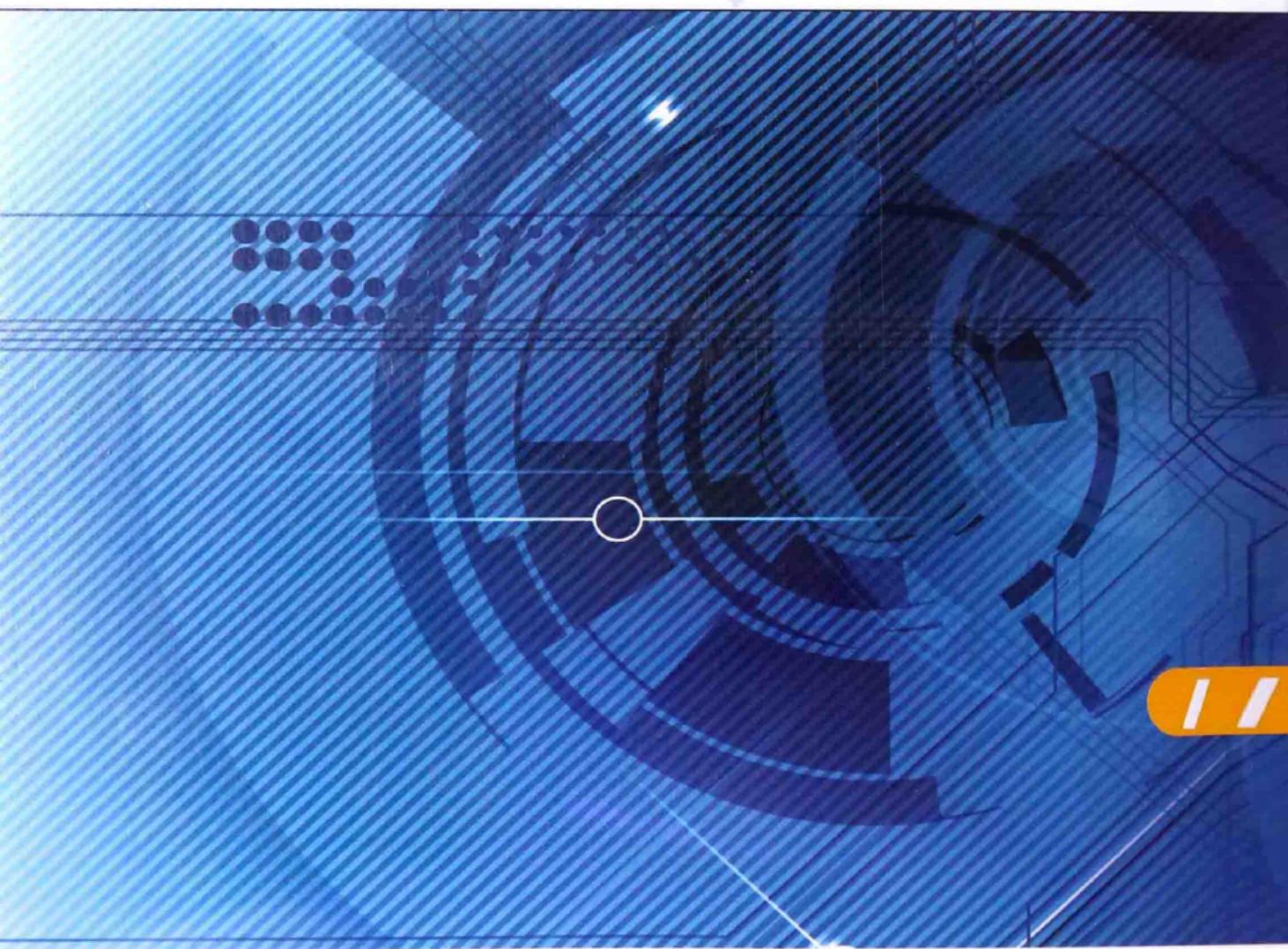


作者团队提供读者贴心服务

电子邮箱: selenium2014@126.com

新浪微博: selenium2014

公众微信: selenium2014



地址:北京市百万庄大街22号

邮政编码:100037

电话服务

社服务中心: 010-88361066

销售一部: 010-68326294

销售二部: 010-88379649

读者购书热线: 010-88379203

网络服务

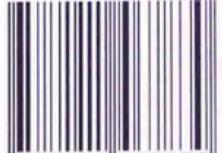
教材网: <http://www.cmpedu.com>

机工官网: <http://www.cmpbook.com>

机工官博: <http://weibo.com/cmp1952>

封面无防伪标均为盗版

ISBN 978-7-111-46783-0



9 787111 467830 >

上架指导 计算机/软件工程/开发测试

ISBN 978-7-111-46783-0

策划编辑◎张俊红

定价: 39.80元