自动化测试框架设计实践

—实现基于 ruby 及 watir-webdriver 的自动化测试框架

作者：郝强

摘要

本文讲述了如何运用 ruby 和 watir 来实现一个可配置化的关键字驱动的 web 自动化测试框架，文中既讲述了框架实现思想又给出了具体实现，是目前为数不多讲述自动化测试框架实现的文章。

关键词

自动化测试框架，关键字驱动

前言

截止至 2013 年 7 月，我已经工作 12 年了，此时我已经在软件测试行业整整工作了 11 个年头，从手工测试到自动化测试，以及软件测试项目管理，一直在苦苦挣扎着，痛并快乐着。

目前，软件测试行业在中国已经越来越被认可，而且呈现出一派欣欣向荣的景象，越来越多的测试书籍充满了书架，当我看到有人将自己的测试经验写成书分享给大家的时候，让我感觉到这个行业人才颇多，虽然自己还是个菜鸟，但出于尝试的目的，通过本书描述了最基本的自动化测试框架功能，争取用最简要的方式，带给大家。

我从来都不是技术高手，也不会写高质量的代码，计算机对于我来说只是一种工具。所以在解决软件测试过程中遇到的问题的时候，我会选择务实的方式来处理，即能够快速有效地解决问题。所有当高手看到我写出了比较垃圾的代码的时候，请您来信告之我，给我提升和改正的机会。

在这个自动化测试被炒得越来越热的时期，我也做了几个自动化测试项目。从大块头的 QTP 以及 RFT 到开源的 ruby，我几乎都做过尝试。

选择适合使用的，并且能够为公司节省成本的工具才是明智之选。当我学习和使用一段时间 ruby 后，对它非常着迷，所以我非常期望将这个自动化测试框架入门级的文章能够呈现给大家。

我目前从事软件测试咨询顾问职业，目前主要在做电子商务的软件测试咨询服务，工作方向包括测试过程管理，自动化测试，性能测试以及测试队伍培养等。说到性能测试以及自动化测试，在互联网企业使用的各种技术，企业其实更期望使用成熟稳定的开源软件，一方面是因为经过市场检验，另一方面使用开源软件可以节省昂贵的软件费用。所以在选型时，我们对开源软件进行了更多的关注，

在能够满足业务使用需求，并且能够达到商业软件同样的效果的时候，我们肯定会选择开源软件。

在针对电子商务系统进行自动化测试时，我们更多的时候是与浏览器打交道，使用 WEB APP。所以在我们需要实现一个支持多浏览器的自动化测试框架的时候，Ruby 和 Watir 进入了我们的视野，同时我们对于我们要实现的框架有一些基本的要求，如必须是关键字驱动的，测试脚本必须是可配置的。当我们熟知这些基本需求后，我们花了一点时间设计了这个框架，最后我们整体用了不到一周的时间就实现了它，实际我们只用了 5 个人天就完成了，我这么讲是想向大家推荐 ruby,因为用 ruby 来实现某些功能是真的是可以用神速来形容。

在本书中，作者假定您已经掌握 ruby 基本语法，有过软件测试经验，了解一些基础的自动化测试理论或有一定的自动化测试经验。当然，如果要应用此框架，你必须 掌握 HTML 元素的意义 ，会使用 Firebug，Firebug 使用用来捕获 web 元素的属性值。基于以上要点，你就可以应用本框架，进行自动化测试脚本配置，并且扩展本框架功能，实现贵公司 特定的自动化测试过程。

本书主要从三个方面来讲解本自动化测试框架。第一是讲解本框架的设计思想，第二讲解框架核心 功能的实现，最后讲解框架的应用示例。关于本框架的名字，我们命名为 AutoLancer，我们的官方网站是 TestSolution.cn，我们可以在那里更多的讨论本框架以及各种软件测试相关的技术。

2013 年 8 月于大连

目录

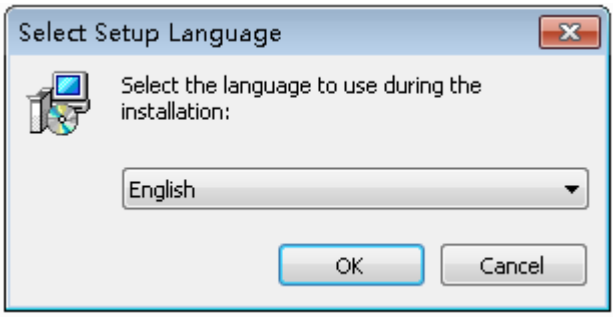
第一章、 框架开发环境安装

1.1 安装 Ruby 1.9.3

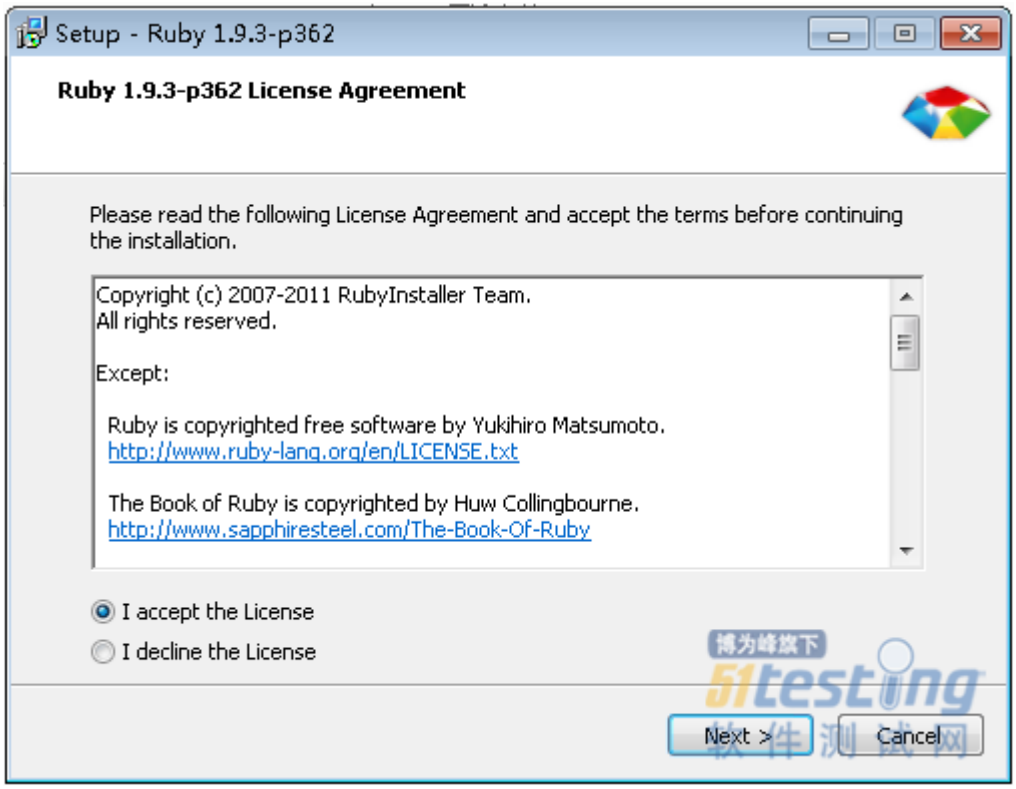
Ruby 1.9 的 Windows 版本安装文件位于其官方网站：http://rubyinstaller.org/downloads/，各位可以直接选择对应版本进行安装，以下是本人将相关软件下载到本地的文件目录截图。

名称	修改日期	类型
subclipse1.8.18	2013/1/28 11:27	文件夹
DevKit-tdm-32-4.5.2-20111229-1559-...	2013/1/28 11:27	应用程序
eclipse	2013/1/28 11:27	WinRAR 压缩文件
gems	2013/1/28 11:27	WinRAR 压缩文件
ie与chrome的webdriver驱动	2013/1/28 11:27	WinRAR 压缩文件
jre-6u32-windows-x64	2013/1/28 11:27	应用程序
rubyinstaller-1.9.3-p362	2013/1/28 11:27	应用程序
site-1.8.18	2013/1/28 11:27	WinRAR ZIP 压缩...

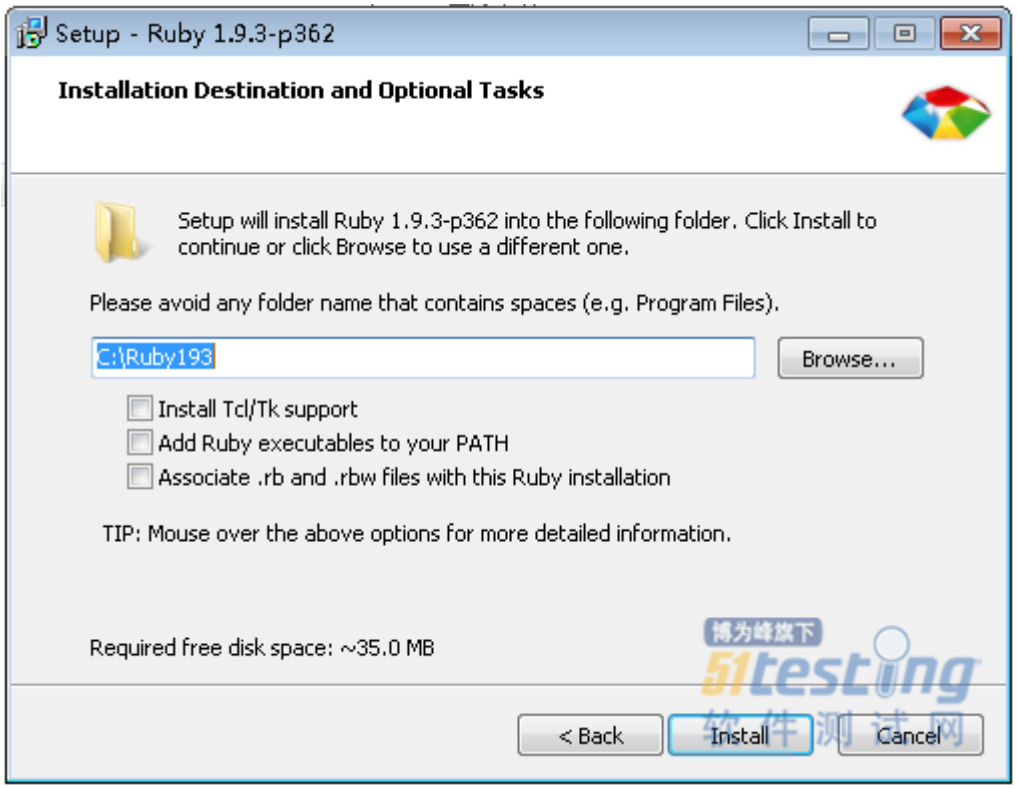
双击 rubyinstaller-1.9.3-p362.exe，弹出选择安装语言界面，如下图所示：



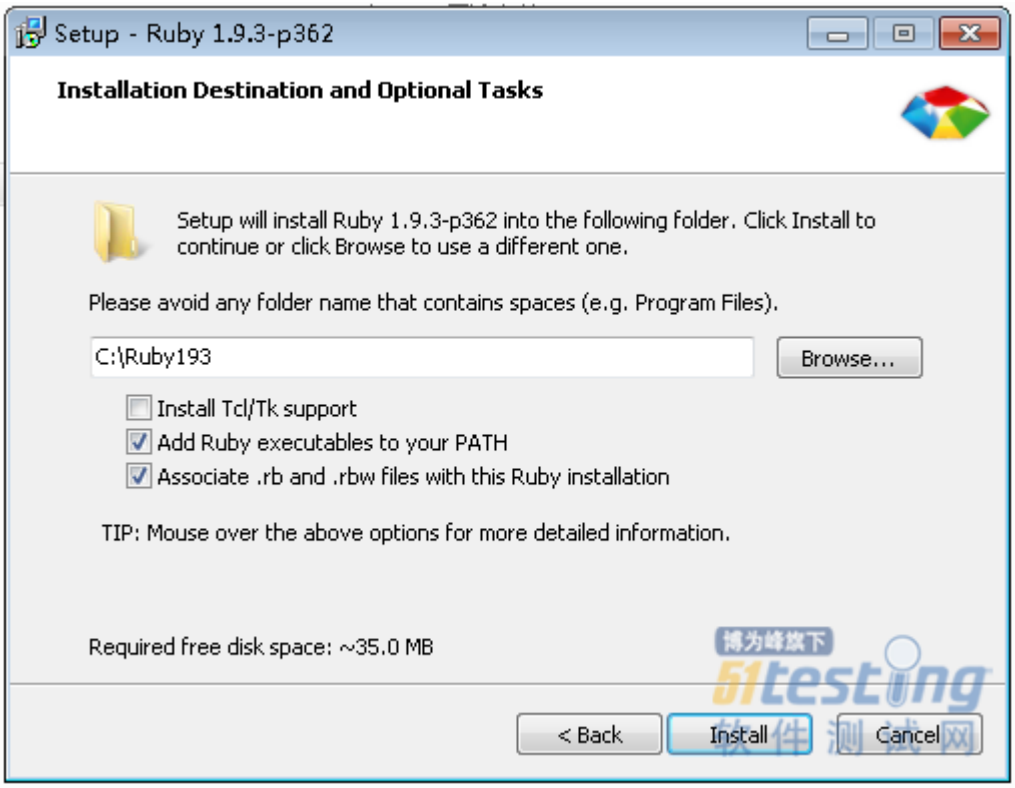
选择安装语言为英语(English)后，点击 OK 按钮，系统提示许可协议，参见下图：



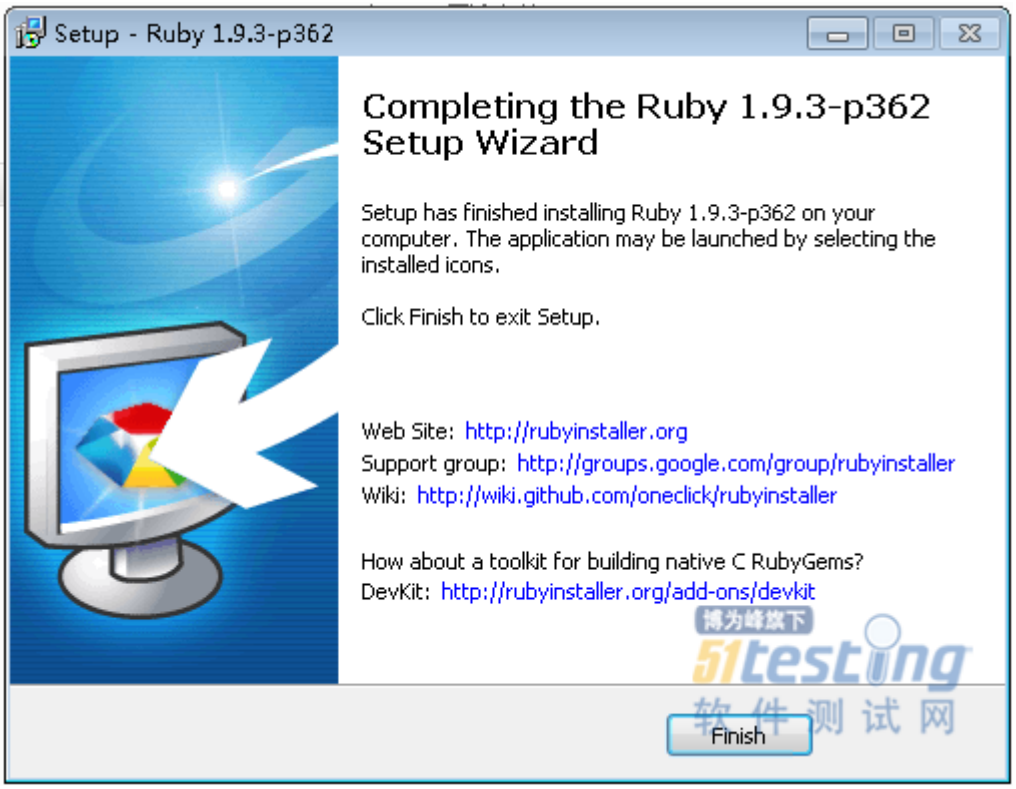
选择接受协议并点击 Next 按钮。



如上图，将安装目录设置为 c:\Ruby193 目录，并点击 Install 按钮，



选中 Add ruby executables to your PATH 及 Associate.rb and .rbw files with this Ruby installation 并点击 Install 按钮，系统将进行安装操作，完成后如下图所示:



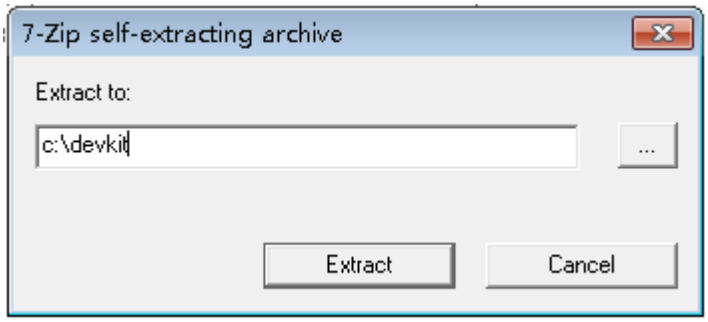
点击 Finish 完成安装 Ruby 1.9.3 操作。

安装完成后，我们测试一下安装是否正确。我们打开一个 cmd 命令窗口，输入 ruby –v 测试一下 ruby 的版本号；再输入 gem -v 测试一下 rubygems 版本号，参见下图：

```
C:\Users\Administrator>ruby -v
ruby 1.9.3p362 (2012-12-25) [i386-mingw32]

C:\Users\Administrator>gem -v
1.8.24

C:\Users\Administrator>_
```

1.2 安装 DevKit

DevKit 也是位于 http://rubyinstaller.org/downloads/下载页中，ruby 1.9.3 版本需要选择 DevKit-tdm-32-4.5.2 版本来安装，请下载后运行该程序。



自解压程序要求我们指定一个安装，通常我们设置为 c:\devkit，之后我们要在命令窗口中运行命令安装 DevKit.

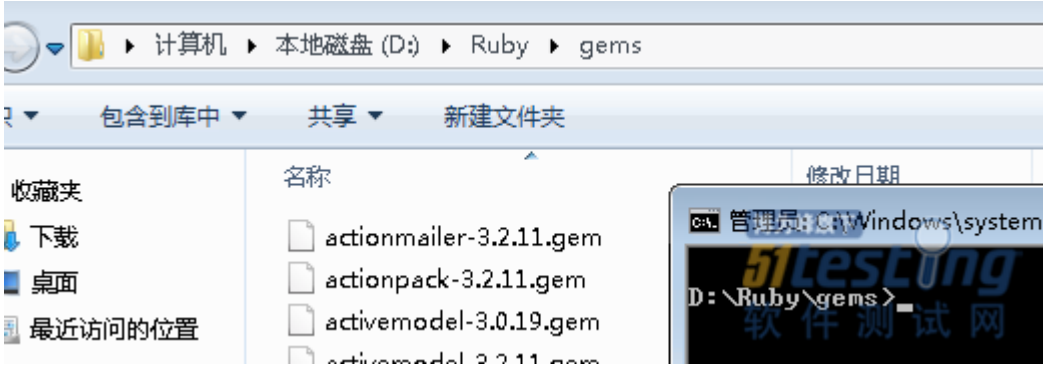
安装只需要运行两条命令即可：

首行需要进入到 c:\devkit 目录下，执行的命令是 cd c:\devkit，运行 ruby dk.rb init 进行初始化操作，之后运行 ruby dk.rb install 完成安装，具体操作过程参照下图所示：

```
C:\devkit>ruby dk.rb init
[INFO] found RubyInstaller v1.9.3 at C:/Ruby193

Initialization complete! Please review and modify the auto-generated
'config.yml' file to ensure it contains the root directories to all
of the installed Rubies you want enhanced by the DevKit.

C:\devkit>ruby dk.rb install
[INFO] Updating convenience notice gem override for 'C:/Ruby193'
[INFO] Installing 'C:/Ruby193/lib/ruby/site_ruby/devkit.rb'

C:\devkit>_
```

1.3 安装 Watir

Ruby 的包通过 gem 来管理，官方网站是 http://rubygems.org/ ，大家可以在上边下载所需要的包，我们主要是要安装 waitr 及 watir-webdriver 包。本人习惯将所用到的包先下载到本地进行安装。



Ruby 用 gem 安装包非常简单，通过 gem install 命令可以直接安装。

Gem install watir --no-ri --no-rdoc

Gem install watir-webdriver --no-ri --no-rdoc

以上两条命令可以完成 watir 及 watir-webdriver 的安装操作

```
D:\Ruby\gems>gem install watir --no-ri --no-rdoc
Fetching: ffi-1.3.1-x86-mingw32.gem (100%)
Fetching: childprocess-0.3.7.gem (100%)
Fetching: websocket-1.0.7.gem (100%)
Fetching: selenium-webdriver-2.29.0.gem (100%)
Successfully installed multi_json-1.5.0
Successfully installed rubyzip-0.9.9
Successfully installed ffi-1.3.1-x86-mingw32
Successfully installed childprocess-0.3.7
Successfully installed websocket-1.0.7
Successfully installed selenium-webdriver-2.29.0
Successfully installed watir-webdriver-0.6.2
7 gems installed

D:\Ruby\gems>gem install watir-classic-3.4.0.gem --no-ri --no-rdoc
Fetching: win32-api-1.4.8-x86-mingw32.gem (100%)
Fetching: nokogiri-1.5.6-x86-mingw32.gem (100%)
Fetching: hoe-3.5.0.gem (100%)
Successfully installed win32-process-0.7.1
Successfully installed win32-api-1.4.8-x86-mingw32
Successfully installed windows-api-0.4.2
Successfully installed windows-pr-1.2.2
Successfully installed nokogiri-1.5.6-x86-mingw32
Successfully installed rautomation-0.7.3
Successfully installed xml-simple-1.1.2
Successfully installed hoe-3.5.0
Successfully installed s4t-utils-1.0.4
Successfully installed builder-3.1.4
Successfully installed user-choices-1.1.6.1
Successfully installed subexec-0.0.4
Successfully installed mini_magick-3.2.1
Successfully installed win32screenshot-1.0.7
Successfully installed watir-classic-3.4.0
15 gems installed
```

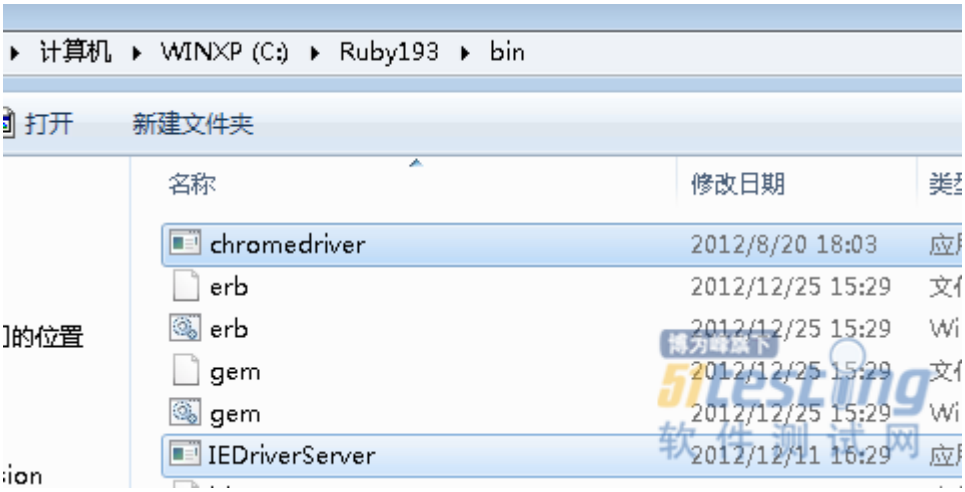
1.4 安装 IE 及 chrome 驱动

IEDriver 驱动地址：http://code.google.com/p/selenium/downloads/list

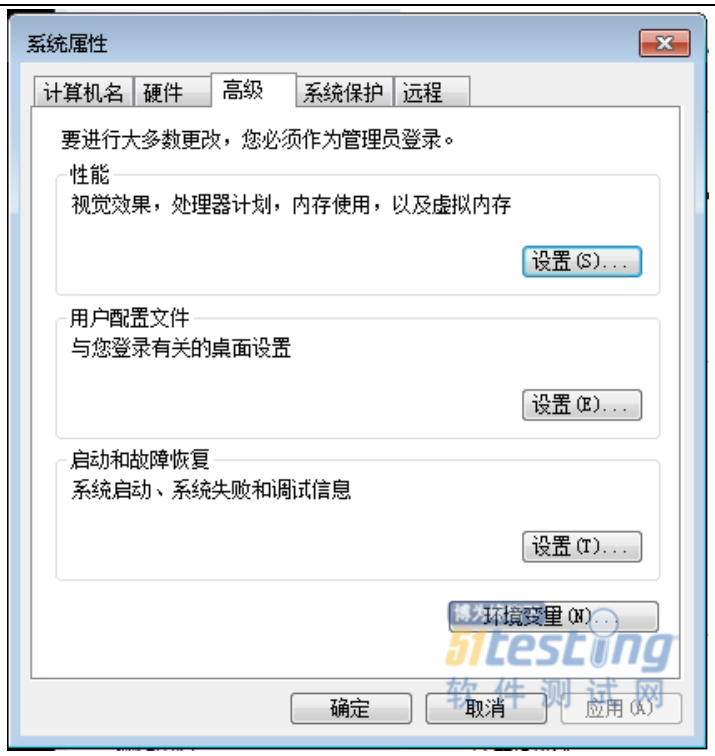
ChromeDriver 驱动地址：http://code.google.com/p/chromedriver/downloads/list

请先将两个 driver 文件下载到本地。

将 IEDriverServer.exe 及 Chromedriver.exe 复制到 ruby 的 bin 目录下，即 c:\Ruby193\bin 下



之后我们需要将两个 driver 文件加到系统的 path 路径中，在 windows 的系统属性里选择环境变量选项。



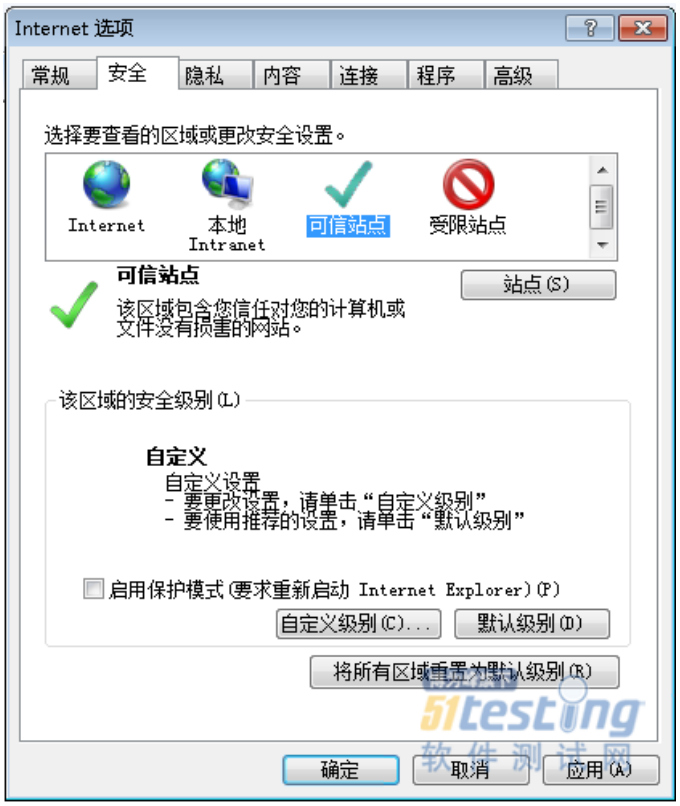
在 path 中添加执行文件路径。

点击“确定”后环境变量添加完成。

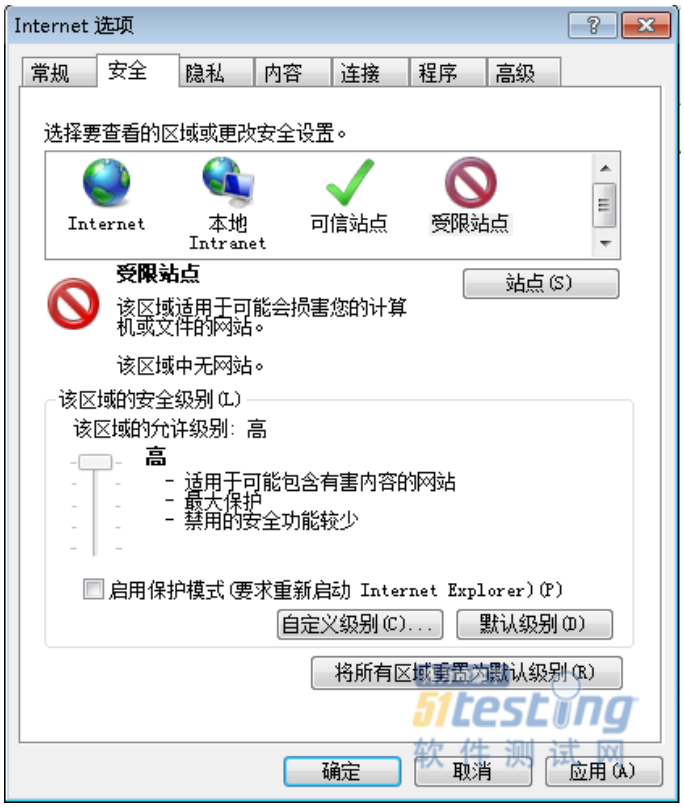
之后我们需要在 IE 的选项里全部取消选中保护模式或全部选中保护模式以保证 driver 生效。这里以全部取消选中保护模式为例。

如上图所示，在 Internet 上取消选择“启用保护模式”。

在本地 Intranet 上也取消选择“启用保护模式”。



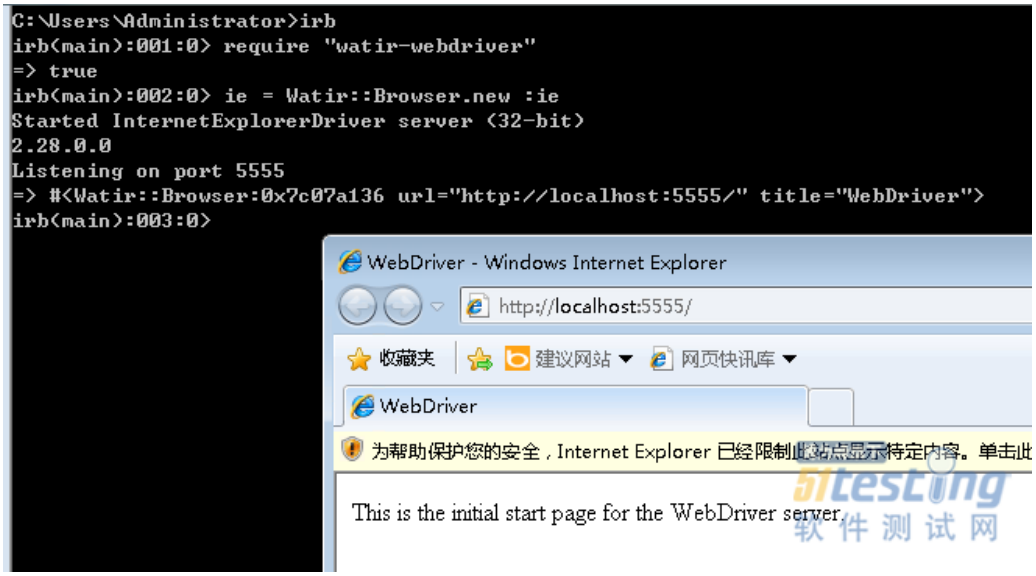
在可信站点上也取消选择“启用保护模式”。



在受限站点上也取消选择“启用保护模式”。下面我们来测试一下 IEdriver 是否生效。

我们在命令行运行 irb 命令。输入 require “watir-webdriver”, 引入 watir-webdriver 包。

再输入 ie = Watir::Browser.new :ie 来启动一个 IE 实例。



如上图所示，如果你看到一个 IE 窗口被打开，就表示配置正常了。

至此，ruby、Watir 以及 IE 和 Chrome 的基本配置工作完成。

第二章、 全面了解 Waitr

Waitr 是 Web Application Testing in Ruby 的缩写。在 Watir 官方网站有一句广告语，是“Automated testing that doesn’t hurt”，翻译过来就是自动化测试不再受伤。

2.1 Watir 是什么？

Watir 是用于自动化 WEB 浏览器的一组开源（BSD）的 Ruby 代码库，发音等同于英语中的 Water。Watir 目前可以驱动微软的 IE，Mozilla 的 Firefox, 苹果的 Safari, Google 的 Chrome 以及 Opera，它目前支持主流的操作系统，包括微软 Windows，苹果的 Mac OS X 以及 Linux。

2.2 Watir 能做什么？

简单来讲，Watir 几乎可以做任何手工与浏览器打交道的动作。例如：打开一个页面，点击一个链接或按钮，在输入框中输入内容等等。除此外，它也能够检查在浏览器中打开的页面，例如：列出当前页的所有链接，复选框是否选中，单选钮是否显示等等。了解到这些，我相信，聪明的读者已经知道我们可以利用 Watir 做一些软件测试需要做的事情了。

2.3 Watir 不能做什么？

非常重要的一点是，虽然 Watir 能够驱动浏览器，但它不能够控制浏览器插件，像 Flash，Java applet 以及微软的 Silverlight。

2.4 Watir 对 HTML 元素的支持

不同的 Watir Gem 包对 HTML 元素的支持也不尽相同。Watir-webdriver 的 Gem 包支持所有的 HTML 元素。Watir Gem 包支持大多数公共的 HTML 元素，但它比较容易扩展成为支持更多的元素。SafariWatir 对于大多数 HTML 元素都不支持，它只支持最常用的那些。

网站 http://wiki.openqa.org/display/WTR/HTML+Elements+Supported+by+Watir 列出了 Watir-Webdriver 对 HTML 元素支持的表格。

Watir method	HTML tag	:action	:after?	:alt	:class	:css	:for	:href	:html	:id	:index	:method	:name	:src	:text	:title	:url	:value	:xpath	Multiple Attributes
area	<area>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
button	<button> <input type="button"> <input type="image"> <input type="reset"> <input type="submit">	✗	?	✓	✓		✗	✗	?	✓	✓	✗	✓	✓ he input type is image	✓	✗		✓	✓	✓
cell	<td>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
checkbox	<input type="checkbox">	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✓	✓	✓
dd	<dd>																			
div		✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
dl	<dl>	?	?	?	?		?	?	?	?	?	?	?	?	?	?		?	?	?
dt	<dt>	?	?	?	?		?	?	?	?	?	?	?	?	?	?		?	?	?
em		?	?	?	?		?	?	?	?	?	?	?	?	?	?		?	?	?
file_field	<input type="file">	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✗	✓		✓	✓	✓
form	<form>	✓	?	?	✓		✗	✗	?	✓	✓	✓	✓	✗	✗	✗		✗	✓	✓
frame	<frame> <iframe>	✗	?	?	✗		✗	✗	?	✓	✓	✗	✓	✓	✓	✗		✗	✗	✓
h1 h2 h3 h4 h5 h6	<h1> <h2> <h3> <h4> <h5> <h6>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
hidden	<input type="hidden">	✗	?	?	✓		✗	✗	?	✓	✓	✓	✓	✗	✓	✗		✓	✓	✓
image	<img>	✗	?	✓	✓		✓	✗	?	✓	✓	✗	✓	✓	✓	✗		✓	✓	✓
label	<label>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
li		✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
link	<a>	✗	✓	?	✓		✗	✓	✓	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
map	<map>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✓	✓	✓
p	<p>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
pre	<pre>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
radio	<input type="radio">	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✓	✓	✓
row	<tr>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
select_list	<select>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✓	✓	✓
span		✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
strong		?	?	?	?		?	?	?	?	?	?	?	?	?	?		?	?	?
table	<table>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
tbody	<tbody>	?	?	?	?		?	?	?	?	?	?	?	?	?	?		?	?	?
text_field	<input type="password"> <input type="text"> <textarea>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✓	✓	✓
ul		✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
New Browser Windows	n/a	✗	?	?	✗		✗	✓	?	✗	✗	✗	✗	✗	✗	✓		✗	✗	?

2.5 Watir-WebDriver 介绍

Watir-WebDriver 目前支持四种浏览器：分别是 IE，Chrome，FireFox，Safari，同时它也对移动设备的测试提供了支持。

由于 Watir-WebDriver 对所有 Html 元素支持，并且可以同时支持四个浏览器，所以理所当然当你要实现一个测试框架时，Watir-WebDriver 就成为了首选。

参见如下代码可以启动 Chrome 浏览器:

```
require "watir-webdriver"
b = Watir::Browser.new :chrome
```

如果要启动其它浏览器，参见如下代码：

```
ie = Watir::Browser.new :ie
ff = Watir::Browser.new :firefox
sf = Watir::Browser.new :safari
```

2.6 Watir-WebDriver 元素识别

所有的 html 元素都被 watir-webdriver 支持，这里我们准备一些例子供大家

参考：

2.6.1 Text Fields 输入框

```
require 'watir-webdriver'
b = Watir::Browser.start 'www.baidu.com'
t = b.text_field :id => 'kw'
p t.exists?
t.set '自动化测试'
p t.value
```

2.6.2 Buttons 按钮

```
require 'watir-webdriver'
b = Watir::Browser.start 'www.baidu.com'
t = b.text_field :id => 'kw'
t.set '自动化测试'
btn = b.button :value, '百度一下'
p btn.exists?
btn.click
```

2.6.3 Links 超链接

```
require 'watir-webdriver'
b = Watir::Browser.start 'www.baidu.com'
l = b.link :text => '百科'
p l.exists?
l.click
```

2.6.4 Divs & Spans

```
require 'watir-webdriver'
b = Watir::Browser.start 'www.baidu.com'
d = b.div :id => 'content'
p d.exists?
p d.text
s = b.span :class => ' s_btn_wr'
p s.exists?
p s.text
```

2.6.5 处理等待

当你访问动态 web 界面时等待通常是不确定的，特别是拥有大量 AJAX 异步请求的站点。通过情况下大家会用 sleep 进行延时等待，但在不同的网络环境 sleep 的时间是不确定的。

2.6.6 Explicit waits 显式等待

Watir::Wait.until { ... }：等待直到块即{}内的内容为真

object.when_present.set：对象出现再做动作

object.wait_until_present：一直等待到对象出现

object.wait_while_present：一直等待到对象消失

2.6.7 Implicit waits 隐式等待

隐式等待用于设置 Watir-WebDriver 最大的超时时间，以秒为单位。

```
require 'watir-webdriver'
b = Watir::Browser.new
b.driver.manage.timeouts.implicit_wait = 3    #3 秒
```

2.7 Watir-WebDriver 高级交互技术

Watir-WebDriver 高级交互技术主要用于处理浏览器认证，证书，下载，代理，弹出窗口，javascript 对话框等，这里会分别讲解相关用法。

2.7.1 Basic Browser Authentication

浏览器的 http basic authenticaio 认证方式，watir 处理起来是非常容易，只需要在 url 中将相关参数传送过去即可以完成认证。

```
require 'watir-webdriver'
b = Watir::Browser.start 'http://admin:password@TestSolution.cn'
```

2.7.2 处理浏览器下载

自动处理下载并保存到某个目录 。

Firefox

```
download_directory = "#{Dir.pwd}/downloads"
download_directory.gsub!("/", "\\") if Selenium::WebDriver::Platform.windows?
profile = Selenium::WebDriver::Firefox::Profile.new
profile['browser.download.folderList'] = 2 # custom location
profile['browser.download.dir'] = download_directory
profile['browser.helperApps.neverAsk.saveToDisk'] = "text/csv,application/pdf"
b = Watir::Browser.new :firefox, :profile => profile
```

Chrome

```
download_directory = "#{Dir.pwd}/downloads"
download_directory.gsub!("/", "\\") if    Selenium::WebDriver::Platform.windows?

profile = Selenium::WebDriver::Chrome::Profile.new
profile['download.prompt_for_download'] = false
profile['download.default_directory'] = download_directory
b = Watir::Browser.new :chrome, :profile => profile
```

2.7.3 处理 JAVAScript 对话框

JavaScript Alerts

检查 Alert 是事被显示

browser.alert.exists?

获得 Alert 的文件内容

browser.alert.text

关闭 alert

browser.alert.ok

browser.alert.close

JavaScript Confirms

点击确定

browser.alert.ok

点击取消

browser.alert.close

JavaScript Prompt

输入文本

browser.alert.set "Prompt answer"

点击确定

browser.alert.ok

点击取消

browser.alert.close

第三章、 自动化测试框架设计思想

3.1 框架逻辑结构

本框架的逻辑模型我命名它为 RTP 模型，即由 Resource (资源)，Task(任务)，Presentation(展示)三部分组成,参照下表:

Resource (资源)	TestDatas
	TestScripts
	TestObjects
	RunList
Task (任务)	Action
	CustomizeAction
	Verification
	CustomizeVerification
	RunJobList
Presentation (展示)	Html Results
	RunTimeData
	RunJobResults
	Email Results

资源模型（Resource Model）

狭义上资源模型的组成是由测试对象(TestOjbects)，测试数据(TestDatas)，测试脚本(TestScripts)以及运行任务列表(RunJobLists)组成。

广义上的资源模型还包括一切测试中使用到的元素，它包括测试环境，分布式测试机，人员等，此部分不在本书讨论范围之内。

测试对象是一组用于识别被测试元素的集合。Web 测试中，它可能是页面上的某个链接，提交按钮，也可能是一段文本。它有一系列固有属性以保证它能够被唯一识别，本书提到的 Watir-WebDriver 提供了对 Web 测试对象识别技术的支持。

测试数据是支撑测试脚本进行测试所使用的数据资源，主要与实际测试业务模型相关。例如在电子商务系统测试中，它可能包括商品信息，价格信息，库存信息，订单信息等。

测试脚本实际上使用了测试数据以及测试对象，并且实现了被测试产品的业务逻辑的验证。

运行任务列表是一组待执行的测试脚本列表的集合。在本书所定义的运行列表包括了要运行的脚本集合，是否运行，以及使用哪个浏览器来运行测试等信息。

任务模型（Task Model）

任务模型主要用于实现一组测试，其实就是我们通常理解的测试脚本，它对应着具体的测试用例，其实现又分为动作(Action)与验证(Verification)。在TestCase(测试用例)中的每个 Step(步骤)，配置了具体的动作(Action)或验证(Verification)。当然，其实现也可以包括自定义动作(CustomizeAction)与自定义验证(CustomizeVerification)，这样在 TestCase(测试用例)中的每个 Step(步骤)上，也可以配置具体的自定义动作(CustomizeAction)与自定义验证(CustomizeVerification)。

动作(Action)用于指示用户实现一个动作。在测试用例中可以表现为点击提交订单按钮，或是输入收货地址。它对应到测试用例中的输入。

自定义动作(CustomizeAction)用于指示用户实现一组动作。在测试用例中可以表现为如登录操作，其操作实际是多个动作(Action)的组合，对于登录来说，可能包含了输入用户名，再输入密码，最后点击登录按钮，这些单个动作组合成了一个自定义动作(CustomizeAction)。它也对应到测试用例中的输入。

验证(Verification)用于检查一项输出。在测试用例中可以表现检查订单号，消费金额，审批是否通过等。它对应到测试用例中的输出检查。

自定义验证(CustomizeVerification)用于检查一组输出。在测试用例中可以表

现订单提交页面中的多个元素或结果的检查，其操作实际是多个验证(Verification)的组合，对于订单提交页检查来讲，它可能检查商品价格是否正确，商品是否正确，总订单金额是否正确等等。它也对应到测试用例中的输出检查。

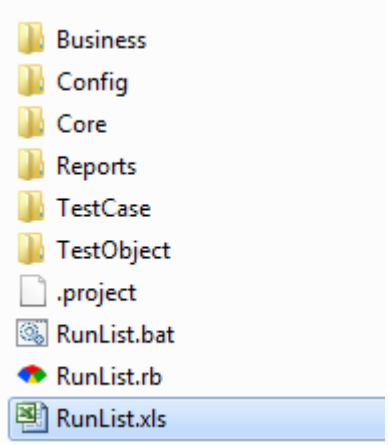
任务调度（RunJobList）也属于任务模型中的一部分，任务调度(RunJobList)其实就是资源中的 RunList，在自动化任务执行中，它会调用不同的测试用例，并且指定其执行的浏览器等，它也可以被指定为分配到不同的机器上来执行一批测试用例，此场景应用于分布式测试环境中。当然如果框架支持分布式测试，它也会包括，在哪台负载机上执行哪些测试，你所实现的框架功能越强大，本模型所支持的范围越大。

展示模型（Presentation Model）

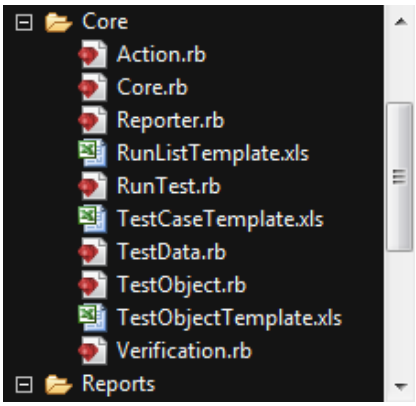
展示模型（Presentation Model）用于表达以什么样的形式将测试结果展示给用户看。通常我们会有一个基于 HTML 的结果，也可能会有一个 EMAIL 结果通知。我们通常会在运行的测试结果中展示一些内容，如本次测试运行一共花费了多长时间，有多少用例被执行，有多少测试用例通过，有多少测试用例失败。更宽泛的讲，我们也可以将一些测试度量数据展示给用户，这取决于你最终的框架实现是否支持此功能。

3.2 框架物理结构

框架物理结构决定了框架的文件组织方式，即目录结构方式，如下图所示:



从上图可以看出，主要有六个目录，分别是 Business，Config，Core，Reports，TestCase，TestObject 以及运行需要调用的 Runlist 相关文件。



Core 目录下存储了框架核心，其实现了对测试对象、测试数据以及任务的读取并且完成测试任务运行，最终将测试结果在 Reports 下。

TestCase 目录主要用于存储所有的测试用例，单个 Excel 文件，一个文件代表一条测试用例，其内容指示测试执行时所使用的资源，如进行何种操作的动作，以及如何进行检查的验证指示。

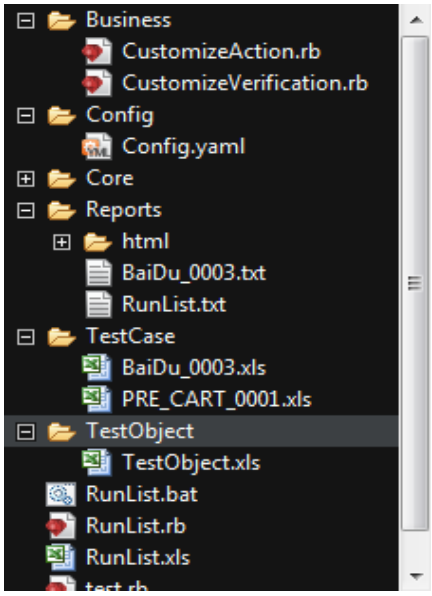
TestObject 目录存储了所有的测试对象。

BusinessLib 用于实现对特殊业务逻辑的封装，其本质 为 CustomizeAction 和 CustomizeVerification 的具体实现。

Reports 目录里边存储了每次测试执行结束后的 HTML 格式报告。

Config 目录里边存储了一些配置信息。

RunList.xls 用于定制本次执行的任务。



3.3 框架主要数据定义

框架中定义了四类主要数据

1. RunList 数据，即运行列表数据。

	A	B	C	D
1	TestCaseID	Description	Browser	Run Flag
2	PRE_CART_0001	验证注册用户未登录情况下，将商品加入购物车，进行结算功能正确	IE	Y
3	PRE_CART_0002	验证未注册用户未登录情况下，商品加入购物车，进行结算功能正确	FireFox	N
4	PRE_CART_0003	验证在商品详情页将商品加入购物车功能正确	CHROME	Y
5	PRE_CART_0004	验证商品详情页将商品加入购物车并去购物车结算功能正确	OPERA	N

运行列表数据定义了要运行的任务，并告之框架在哪个浏览器上运行，以及是否运行。

TestCaseID 列定义了要执行的测试用例编号，其值必须是唯一。它决定后续要调用的具体测试用例文件。

Description 是对测试用例的描述。

Browser 定义了要启用的浏览器，其值可以是 IE, FireFox, CHROME, OPERA。

RunFlag 定义了其测试用例是否要执行.其值 可以为 Y 或 N。Y 代表要执行，N 代表不执行。

RunList 数据通常命名为 RunList.xls 并存储在框架根目录下。

2. Config 数据,即配置数据

```
Config:
  wait: 5
  present: 15
  alert: 30
  refresh: 2
  ok: ok
  cancel: close
  tolerance: 20
```

配置数据主要定义一些框架运行时的配置信息，其内容是可以根据需求来扩展的，我们目前使用的主要是配置框架在识别对象的等待时间。

wait 定义了运行测试时要等待的时间，单位为秒，默认为 5 秒。

present 定义了运行测试时等待元素出现的时间，单位为秒，默认为 15 秒。

alert 定义了运行测试时等待 alert 窗口出现，单位为秒，默认为 30 秒。

refresh 定义了运行测试时间隔的刷新时间，单位为秒，默认为 2 秒。

ok 定义了运行测试时点击 ok 按钮要使用的元素值。

cancel 定义了运行测试时点击 cancel 按钮要使用的元素值。

tolerance 定义了运行测试时等待要忍耐最长的时间，单位是秒，默认为 20 秒。

配置数据通常命名为 Config.yaml 并存储在框架根下的 Config 目录中，关于 yaml 文件格式，请参考 ruby 的 yaml 文档。

3. TestCase 数据，即测试用例数据

	A	B	C	D	E	F	G	H	I	J
1	Step	Desc	StepMethod	ObjectName	CallMethod	RequireValue	ActualValue	Result	Capture	RunFlag
2	Step1	进入百度首页	Action	baidu	go_to				Y	Y
3		点击新闻	Action	news	click				Y	Y
4	Step2	进入百度首页	Action	baidu	go_to				Y	Y
5		输入郝强	Action	KeyWords	setvalue	郝强			Y	Y
6	Step3	用户点击百度一下	Action	Search	click				Y	Y

Steps 列定义了步骤值，它是经过排序的字符串，用于标识测试用例的步骤。推荐用 Step1,Step2 这样的排序字串。

Desc 列定义了当前测试步骤的描述。

StepMethod 列定义了此步骤的执行方法是用于做动作，还是用于检查执行结果。其值可以是 Action，Verification，CusAction 和 CustVerification，Action 代表本步骤要进行一个动作。Verification 代表本步骤要做一次结果的检查。CusActionj 是自定义动作，CustVerification 是自定义的检查，它们通常代表一组动作或检查。

MethodName 列定义了此步骤要调用的自定义方法，其值可以为空或为自定义方法名。

ObjectName 列定义了此步骤要使用的对象名，其值会从 TestObject 中读取。

RequireValue 列定义了输入值，其值 是配合 MethodName 或 ObjectType 来使用的，相当于输入或输出要检查的参数。

ActureValue 列定义了测试用例执行时返回的实际结果值。

Results 列定义了测试结果，是否通过或失败。其值只代表当前测试步骤是否成通过或失败。

Capture 定义了当前测试用例步骤是否需要抓图.其值 可以为 Y 或 N。Y 代表要本步骤截图，N 代表本步骤不截图。

RunFlag 定义了当前测试用例步骤是否要执行.其值 可以为 Y 或 N。Y 代表要执行，N 代表不执行。

4. TestObject 数据，即测试对象数据

ObjectName 是对象的命名，它配合 TestCase 中的 ObjectName 一起使用来识别对象。

	A	B	C	D	E
1	ObjectName	ObjectType	How	PropertyValue	Page
2	StoreStatus	Em	Xpath	//*[@id="storestatus"]	http://www.prelive.com/cart.jsp
3	AddToCart	Link	Id	buybtn_a	http://www.prelive.com/cart.jsp
4	KeyWords	text_field	Id	kw	http://www.baidu.com
5	Search	Button	Id	su	http://www.baidu.com
6	baidu	-	-	http://www.baidu.com	http://www.baidu.com
7	news	Link	text	hao123	http://www.baidu.com

ObjectType 定义了要使用的控件类型，其值主要是 Watir-webdriver 所支持的 html 元素名称，如 button,Checkbox,Link,Text_Field 等等。

How 列定义了 Watir 识别对象的方式，Watir 所支持的方法这里均支持。其值可以是 ID,Xpath,Name 等等。

PropertyValue 定义了 Watir 识别对象时所用的属性值，其值可以是 xpath 对应的属性值，id 值或是 name 值。

Page 列说明了该对象所存在的页面。

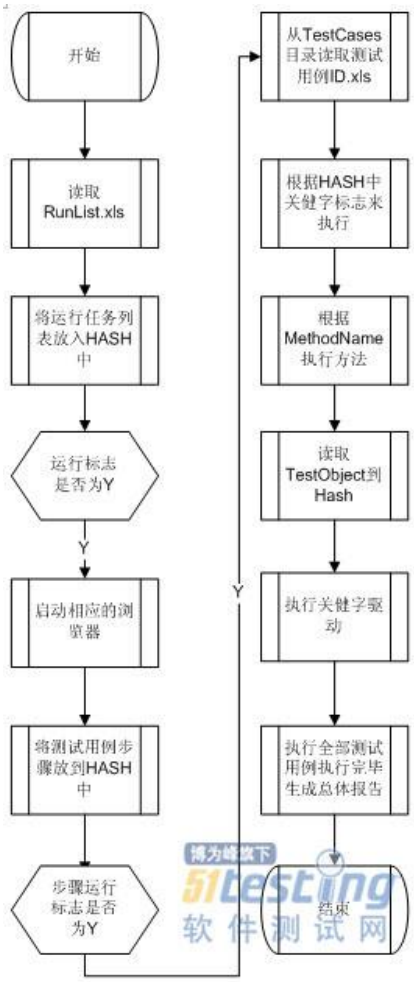
上述三个列 ObjectType, How, PropertyValue 组合在一起就形成 Watir 对于一个 WEB 元素识别的一个基本属性。

下列表列出了对于 ObjectType，以及 How 的支持参考。

常用	全部	全部
ObjectType	ObjectType	How
Button	Button	Action
CheckBox	Cell	After
Link	CheckBox	Alt
Radio	DD	Class
SelectList	Div	Css
TextField	DL	For
CheckBox	DT	Href
	EM	Html
	File_Field	Id
	Form	Index
	Frame	Method
	H1	Name
	H2	Src
	H3	Text
	H4	Title
	H5	Url
	H6	Value
	Hidden	Xpath
	Label	Multiple
	LI	
	Link	
	Map	
	P	
	PRE	
	Radio	
	Row	
	SelectList	
	Span	
	Strong	
	Table	
	Tbody	
	TextField	
	UL	
	NewWindow	

3.4 框架核心流程

框架中的数据结构定义决定了框架的处理流程。上一小节中我们讲述了四个主要数据的定义：即运行列表数据，配置数据，测试用例数据以及测试对象数据。



当框架开始运行时，首先要从 RunList.xls 文件中读取中要执行的测试用例列表，并将其放入到 HASH 中，之后进行判断其运行标记位是否为 Y,如果是 Y 根据浏览器标识值启动相应的浏览器。

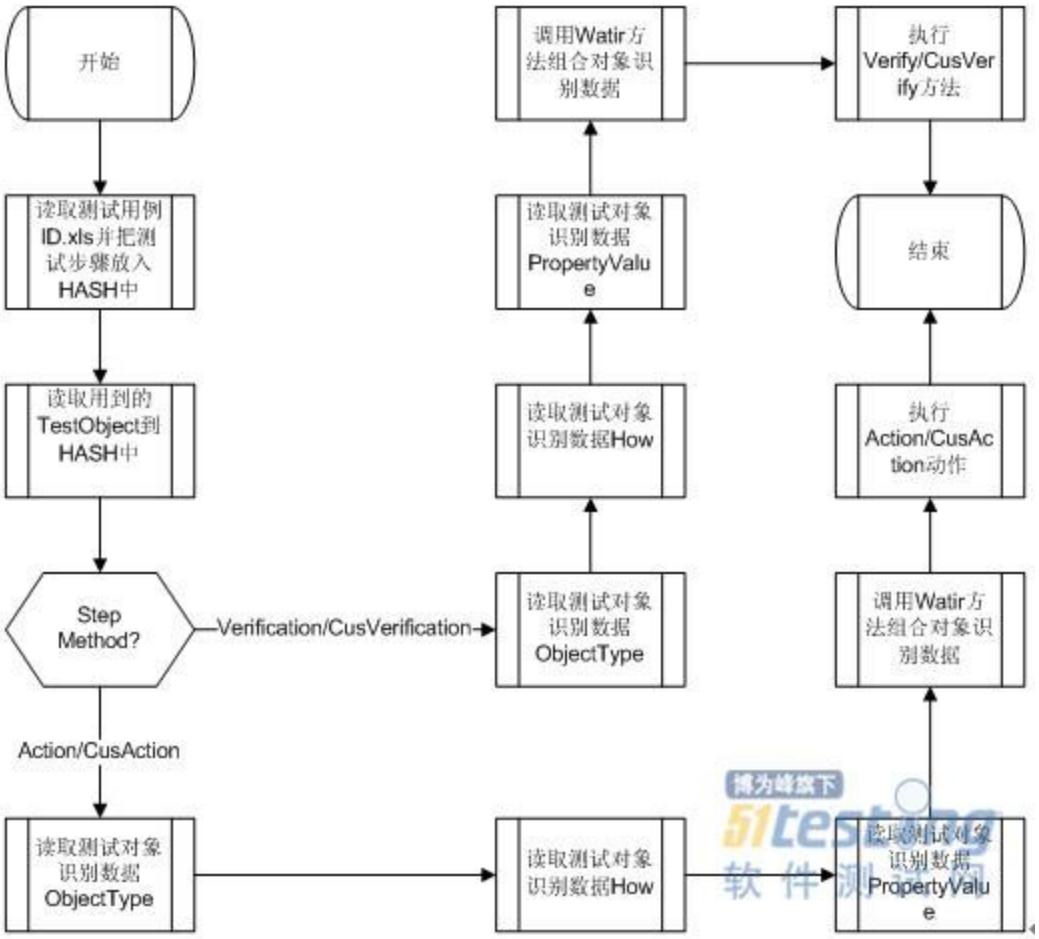
此时还需要在 TestCase 目录中读取相应的 TestCaseID.xls，然后将相应的测试步骤读取出来放入 HASH 中并做好执行测试用例步骤的准备。

运行到这个阶段时，框架已经准备开始运行测试了。计算机根据关键字 Action 或 Verficaion 来执行步骤或进行验证工作。当然关键字也包括 CusAction 和 CusVerification，自定义的动作或验证。当然在此时也会读取 TestObject 信息到 hash 中，然后根据 Watir-WebDriver 来驱动定位元素。

之后执行关键字驱动程序运行测试并根据配置信息来决定是否截屏，最后并

生成测试报告，当全部测试用例执行结束后，生成整体测试报告。

至此，一个基本的测试流程结束。下面我们来看一下关键字驱动实现流程：



当框架运行时，从 TestCases 目录下的 TestCase.xls 文件中读取测试步骤后，会将其步骤信息存储在 Hash 中。框架对步骤 hash 以及测试对象 hash 进行了一系列的处理，利用 Watir 的对象识别技术，组合调用的关键字信息，最终才完成了关键字驱动功能。

对于关键字驱动，是根据 StepMethod 方法来区分的是执行动作还是执行检查。所以，框架先判断 StepMethod 的值 ，如果是 Action,表明当前是要做一个动作，它可能是点击一个链接，它也可能是输入值到文本框，也可能是点击一个按钮。然后，框架会读取待测试对象识别数据 ObjectType，接着读取测试对象识别数据 How,最后读取测试对象识别数据 PropertyValue，经过组合这几个字段值，通过 Watir-WebDriver 驱动就可以识别到页面上的对象，接着执行一个动作。

还有一种情况在上图中并没有详细展示出来，我们还提到过一个 CusAction，

用于自定义动作，我们看 一下 MethodName 和 RequireValue 这两个字段，其时这两个字段实际上在调用自定义方法时起了很关键的作用，比如我们在一个测试场景中需要做登录操作，我们调用动作中就需要输入用户名，输入密码，再点击登录，我们可以上述三个分离的动作合并成一个通用的公共方法叫 login,我们就可以在 MethodName 处设置值为 login，并且把 StepMethod 设置为 CusAction，就可以进行调用了。

当 StepMethod 值的被设置为 Verification 时，表明发前是要做一个检验。它可能是检查页面上某个元素是否存在，比如退出链接是否存在。它也可能是检查支付金额是否是 100 元，它也可能检查某个弹出窗口是否被显示等。然后，框架会读取待测试对象识别数据 ObjectType，接着读取测试对象识别数据 How,最后读取测试对象识别数据 PropertyValue，经过组合这几个字段值，通过 Watir-WebDriver 驱动就可以识别到页面上的对象，接着做一次检验。

我们还是要说 CusVerification 字段 ，当 MethodName 和 RequireValue 被设置时，这表示这时一个自定义的 verification 方法，它可能在同一个方法中检查订单是否被成功提交？支付金额是否是 100 元，支付方式是否是在线支付等。我们将一系列的检查动作，进行封装，就可以实现自定义的验证功能了。

至此，关于关键字驱动基本上介绍清楚了。

所以我认为要实现一个通用的自动化测试框架，框架应该具备如下特点：

1.逻辑结构划分清晰

2.物理结构划分清晰

3.数据处理清晰

4.便于应用及推广

根据上述讨论的框架设计思想，我们会在后续章节中来讨论如何具体实现一个关键字驱动，多浏览器支持，可配置化的基于 Web 测试的自动化测试框架。

第四章、 框架核心功能实现

本书所讨论的自动化测试框架，是基于开源软件 Ruby 以及 Watir 能够实现关键字驱动，多浏览器支持并且自动化测试脚本完全可配置化的一组进行 web 应用软件测试的通用程度代码集的实现。

4.1 多浏览器支持

当前的 web 应用越来越多，通常情况下，我们的产品需要提供多浏览器支

持，像 IE，FireFox，Chrome，Safari，Opera 等等我们可能都需要能够支持。多浏览器测试我们在手工测试中也非常常见，以致于我们花费大量的精力用于同样的功能在不同的浏览器上进行测试，如果这部分能够被自动化，就可以节省大部分人力。当然，如果我们的自动化测试框架如果提供了对多浏览器的支持，那就更好了。

首先我们看一下 Watir 对浏览器的支持情况。我们看到 Watir 项目由几个 Gem 文件组成。

我们先来看一下 watir gem，这个 gem 包可以驱动 IE 浏览器，目前它是对 IE 支持最好的驱动，如果测试基于 IE 的应用，推荐使用 Watir gem。

接着我们看一下 firewatir，它可以运行在 Windows，Linux 以及 Mac 上并且能够驱动 Firefox，但它只能支持到 FireFox3.6 版本。

我们再来看一下 safariwatir，它只能在 MAC 上驱动 Safari 浏览器，而且它没有 Windows 版本。

曾经在很短的一段时间内还有一个叫 chromewaitr 的 gem 包，它只能在 windows 上驱动 Chrome 浏览器。

接下来要介绍的 gem 包叫 watir-webdriver，它能够在 Windows，Linux，Mac 上驱动 IE，FireFox，Chrome 和 Opera。介绍到这里，大家应该看到了一个非常适合用做框架的 watir 包。是的，我们选用 watir-webdriver 来实现多浏览器支持。

我们来看一下在框架的 RunTest 中如何实现根据需要来启动不同的浏览器。

```
require "rubygems"
require "watir-webdriver"
require "Core"
require "TestData"
require "Action"
require "Verification"
require "CustomizeAction"
require "CustomizeVerification"
require "Reporter"
```

```
class RunTest
def initialize
@testObject = TestData.new(Core::TO_PATH + "TestObject.xls","Objects")
@testCase = TestData.new(Core::ROOT_PATH + "RunList.xls","Runlists")
end
def launch(testcase)
Core.browser = Watir::Browser.new testcase["browser"].downcase.to_sym
Core.browser.window.maximize
end
```

第二行 require "watir-webdriver"用于引入 watir-webdriver 包，然后定义了 initialize 方法，通过它将生成一个 RunTest 的实例时，实例化类变量@testObject，@testCase，我们可以看到它们通过 TestData 类来取得对应数据。

让我们再看一下 launch 方法，在这里我们看到，我们根据 testcase[“browser”]的不同值来生成不同的浏览器实例，如 IE，Firefox，Chrome 以及 Opera，之后我们将浏览器窗口进行了最大化操作。这里大家看到 Core.browser，我们这里为了避免使用全局变量，将 browser 放在了叫 Core 的模块里。

4.2 测试数据的获取

本框架的数据使用微软 office excel 来管理，我们需要编写一个 TestData 类来完成对 excel 的操作，并将我们需要的几类数据即 Runlist 数据，TestCase 数据以及 TestObject 数据读出来以备用。

我们做框架物理结构组织时，将 RunList 数据存储在框架的根目录下并命名

叫做 RunList.xls，这个 excel 文件中包括一个叫 RunLists 的 sheet 页，存储的是运行列表数据。

我们先看 一下我们如下的 TestData 类代码，是如何实现基本的 excel 存取功能。

```
require "win32ole"

# Common class for data
# Author: HaoQiang haonumen@gmail.com
# Create Date: 2013/01/10
# Modifier: Hao Qiang
# Modify date: 2013/07/09
class TestData
def initialize(excelName,sheetName)
@excelName = excelName
@sheetName = sheetName
@testobject = []
@testcase = []
end

def open_excel
WIN32OLE.codepage = WIN32OLE::CP_UTF8
@excel = WIN32OLE::new('Excel.Application')
@excel.visible = false
@excel.DisplayAlerts = false
@workbook = @excel.Workbooks.open(@excelName)
@worksheet = @workbook.Worksheets(@sheetName)
@worksheet.Select
rescue Exception => e
@workbook.close unless @workbook.nil?
@excel.quit unless @excel.nil?
close_excel
exit 1
end

def close_excel
@excel.quit
end
```

我们先来看一下 TestData 的 initialize 方法，它带有两个参数，一个是 excelName，一个是 sheetName，当实例化一个 TestData 类时，我们需要将要操作的 excel 文件名及 sheet 名字传送给它。

openExcel 方法实现了打开 excel 并将 worksheet 定位到指定的 sheet 上。

closeExcel 方法实现退出 excel 功能。

看到这里大家会想到我们如何把 RunList.xls 的 RunLists 页里的数据读出来。

我们需要在 TestData 里实现一个方法叫 run_list_data，这里我们先给出其实现。

```
def excel_data(range)
tmp = []
row = 1
open_excel
while @worksheet.range("c#{row}").value
truerange = range.gsub(/\:/,row.to_s + ":")    + row.to_s
tmp << @worksheet.range(truerange).value.flatten
row += 1
end
close_excel
tmp.flatten
end
```

大家可以看到，我们首先声明了一个方法 excel_data，其功能是将指定范围

```
def run_list_data
testcase = excel_data "a:f"
start = 0
while start < testcase.size
@testcase.push(
{ "testcase"       => testcase[start],
"description" => testcase[start + 1],
"browser"         => testcase[start + 2],
"result"          => testcase[start + 3],
"runtime"         => testcase[start + 4],
"runflag"         => testcase[start + 5],
})
start = start + 6
end
return @testcase
end
```

内的 excel 数据放入数组之中，之后使用循环来将数组中的数据放入到 Hash 中，最终将它返回给调用者。

get_test_case 方法实现了将指定名字的 testcase 数据取出来。

那我们如何将指定的测试用例步骤信息取出来呢？我们看一下下面的实现：

```
def get_test_case(name)
run_list_data
@testcase.each{|o| return o if o["testcase"] == name }
return nil
end
```

利用上述实现逻辑，我们可以非常轻松的读取 testSteps 的数据，并将其也存储在 hash 中。

```
def test_step_data
teststep = excel_data "a:j"
tc = []
start = 10
while start < teststep.size
tc.push(
{"step"            => teststep[start],
"desc"             => teststep[start + 1],
"stepmethod"       => teststep[start + 2],
"objectname"       => teststep[start + 3],
"callmethod"       => teststep[start + 4],
"requirevalue" => teststep[start + 5],
"acturevalue"    => teststep[start + 6],
"result"           => teststep[start + 7],
"capture"          => teststep[start + 8],
"runflag"          => teststep[start + 9],
})
start = start + 10
end
return tc
end
```

```
def test_object_data
testobject = excel_data("a:e")
start = 5
while start < testobject.size
@testobject.push(
{ "objectname"        => testobject[start],
"type"        => testobject[start + 1],
"how"          => testobject[start + 2],
"property" => testobject[start + 3],
# "page"        => testobject[start + 4]
})
start = start + 5
end
return @testobject
end

def get_test_object(name)
test_object_data
@testobject.each{|o| return o if o["objectname"] == name }
return nil
end
```

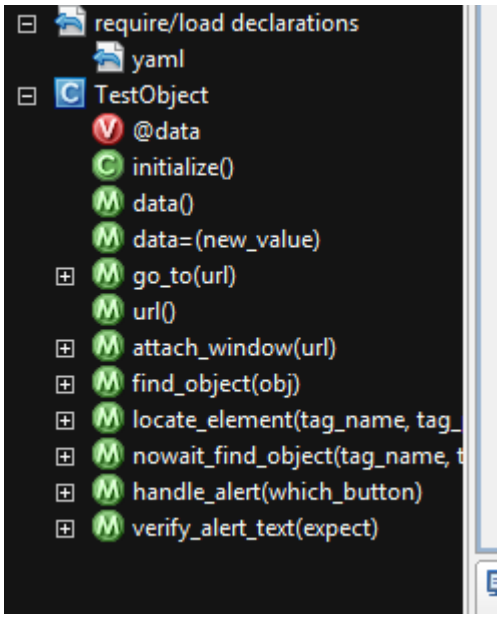
看到这里读者也会想到如何把测试对象数据读出，请看如下实现:

测试对象首先也是从 excel 中读取指定的范围数据并存入数组，然后再将其存入到 hash 中，最后我们实现 get_test_object 来取得指定 name 的 test object 对象。

到这里，基本的数据读取类已经完成。

4.3 TestObject 类的实现

TestObject 类是本框架中非常非常核心的一个功能，它提供了对测试对象识别二次封装，并提供一些基础的关键字驱动功能，它是 Action 及 Verification 的父类。



在 TestObject 类中，我们引入了 yaml 格式的配置文件，我们在之前的章节曾经提到过它。

```
class TestObject
attr_accessor :data

def initialize
@data ||= YAML.load_file(Core::CONFIG_PATH + "Config.yaml")["Config"]
end
```

我们首先将 Config.yaml 读入并存入到类实例变量@data 中。其次，我们看一下几个方法的实现 。

```
# description: go to the url.
# example:
# ====      go_to("TestSolution.cn")
def go_to(url)
Core.browser.goto(url)
return Core::PASS
end
```

```
# description: return current url
# example:
# =====       url => "TestSolution.cn"
def url
Core.browser.url
# return Core::PASS
end

# description: attach a popup window
# example:
# ====       attach_window("TestSolution.cn")
def attach_window(url)
url = eval url if url =~ /^\/.*\/$/
Core.browser.window(:url,url).use
return Core::PASS
end
```

go_to 实现了让浏览器去访问指定的 url 地址功能

url 实现了返回当前 url 地址的功能。

attach_window 用于当前浏览器对象 attach 到指定的 url 窗口上，这在我们的 web 程序如果有弹出窗口时非常有用。

现在我们来看一下，TestObject 如何实现对对象识别技术的二次封装。

```
# description: locate element
# example:
# ====      find_obj(:text_field,:id,"hr") => a textfield with para: 'id=>hr'
def find_object(obj)
return locate_element(obj["type"], obj["how"], obj["property"])
end
```

```
def locate_element(tag_name,tag_property,pro_value)
return nowait_find_object(tag_name,tag_property,pro_value).when_present(@data["wait"])
end

def nowait_find_object(tag_name,tag_property,pro_value)
pro_value = eval pro_value if pro_value =~ /^\/.*\/$/
pro_value = pro_value.to_i if tag_property.to_s == "index"
tag_name = tag_name.downcase.to_sym
tag_property = tag_property.downcase.to_sym
return Core.browser.send(tag_name,tag_property,pro_value)
end
```

我们先看一下 nowait_find_object，它利用 watir-webdriver 的对象识别技术来操作对象，我们这里使用 ruby 的元编程技术将其处理实现。

而 locate_element 主要是实现了要等到被测试对象显示出来。最终我们通过 find_object 功能可以实现在指定的延迟时间内完成对象识别功能。

最后一部分我们实现了对 javascript 的 alert 窗口处理功能。

```
# description: handle alert
# example:
# =====     handle_alert("OK") => click ok button
# =====     handle_alert("CANCEL") => click cancel button
def handle_alert(which_button)
0.step(@data["alert"],@data["refresh"]) do |hdat|
return Core.browser.alert.send(@data["#{which_button}"]) if Core.browser.alert.exists?
sleep(@data["refresh"])
end
return Core::ALERT_MISSING
end
```

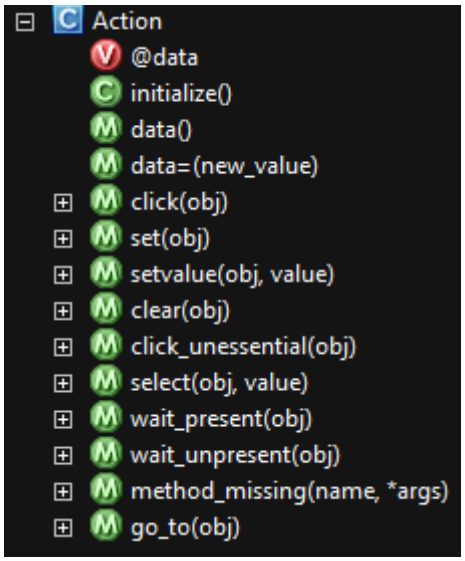
```
# description: verify alert's text and click "OK".
# params: expect
# example:
# =====     verify_alert_text("TestSolution")
def verify_alert_text(expect)
0.step(@data["alert"],@data["refresh"]) do |atxt|
break if Core.browser.alert.exists?
sleep(@data["refresh"])
return Core::ALERT_MISSING if atxt == @data["alert"]
end
actual_text = Core.browser.alert.text
Core.browser.alert.ok
return Core::PASS if expect == actual_text
return Core::FAIL + "#actual_text"
end
```

handle_alert 主要实现了在指定的延迟时间内对按钮的点击功能，点 ok 或是 cancel。这里要多说一下，我们在实际运行自动化测试过程，经常会出现 alert 出现的时间点不一样，有时候一秒内就出现，有时候需要好几秒，所以我们这边实现了一个在指定最大时间内来判断它是否出现，再进行点击的功能。

verify_alert_text 用于检查 alert 窗口上显示的文本是否正确，并点 ok 按钮将其关闭。在这里我们也指定了最大等待时间点。

4.4 Action 类的实现

Action 类继承于 TestObject，在其基础上我们又实现了一系列的动作关键字方法，先看一下结构。



Action 实现的关键字有：click，set，setvalue，clear，click_unessential，select，wait_present， wait_unpresent 以及 go_to，当然我们还用到一个元编程技术 method_missing，用于处理未定义关键字的调用。

```
def click(obj)
find_object(obj).click
return Core::PASS
end

def set(obj)
find_object(obj).set
return Core::PASS
end

def setvalue(obj, value)
find_object(obj).set(value)
return Core::PASS
end

def clear(obj)
find_object(obj).clear
return Core::PASS
end
```

click 实现对测试对象的点击功能。

set 实现对复选框等测试对象的选中功能。

setvalue 实现对文本域输入值的功能。

clear 实现对文本域等清空功能。

```
def click_unessential(obj)
find_object(obj).click rescue return Core::PASS
return Core::PASS
end

def select(obj, value)
find_object(obj).select(value)
return Core::PASS
end

def wait_present(obj)
nowait_find_object(obj["type"],obj["how"],
obj["property"]).wait_until_present(@data["tolerance"]) rescue return Core::ERROR +
$@.to_s
return Core::PASS
end

def wait_unpresent(obj)
nowait_find_object(obj["type"],obj["how"],
obj["property"]).wait_while_present(@data["tolerance"]) rescue return Core::ERROR +
$@.to_s
return Core::PASS
end

def method_missing(name,*args)
return Core::ERROR + "#{name} hasn't been defined in action."
end

def go_to(obj)
super(obj["property"])
return Core::PASS
end
```

click_unessential 实现对点击后可以展示的页面的功能支持。

select 实现对下拉列表的选择功能。

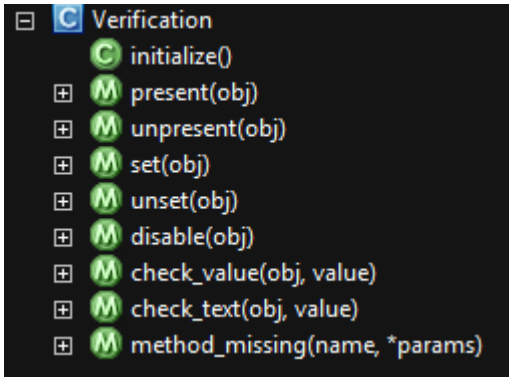
wait_present 实现等待到测试对象出现，其超时时间在 config 中进行过设置。

wait_unpresent 实现等待到测试对象消失，其超时时间在 config 中进行过设置。

go_to 直接调用父类的 go_to 实现浏览器跳转。

最后通过 method_missing 元编程技术实现对其它未定方法的容错处理。

4.5 Verification 类的实现



Verification 类继承于 TestObject，在其基础上我们又实现了一系列的验证关键字方法，我们了也先了解一下结构。

Verification 类提供了一组用于检查状态或值的方法，它们分别是 present，unpresent，set，unset，disable，check_value，check_text 以及 method_missing 方法。

```
def present(obj)
return Core::PRESENT_FAIL unless nowait_find_object(obj["type"], obj["how"], obj["property"]).present?
return Core::PASS
end

def unpresent(obj)
return Core::UNPRESENT_FAIL if find_object(obj).present?
return Core::PASS
end

def set(obj)
return Core::SET_FAIL unless find_object(obj).set?
return Core::PASS
end

def unset(obj)
return Core::UNSET_FAIL if find_object(obj).set?
return Core::PASS
end

def disable(obj)
return Core::DISABLE_FAIL unless find_object(obj).attribute_value("disabled")
return Core::PASS
end
```

present 用于检查测试对象是否被展示，unpresent 正好相反，用于检查测试对象是否未被显示。

set 用于检查复选框等测试对象是否被选中，unset 正好相反，用于检查复选框等测试对象是否未被选中。

disable 用于检查测试对象是否为 disable 状态。

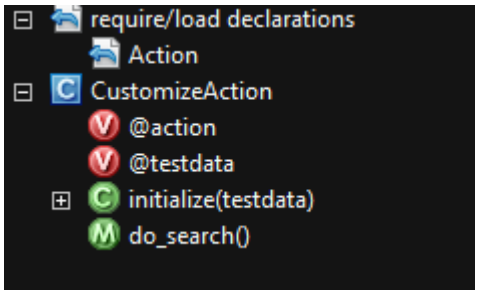
```
def check_value(obj, value)
return Core::FAIL + "#{find_object(obj).value}" unless find_object(obj).value == value
return Core::PASS
end

def check_text(obj, value)
return Core::FAIL + "#{find_object(obj).text}" unless find_object(obj).text == value
return Core::PASS
end

def method_missing(name,*params)
return Core::ERROR + "method: #{name} hasn't been defined in verification."
end
```

check_value 用于检查值是否相等，check_text 用于检查文本是否相等。

method_missing 用于处理未定义方法的容错。

4.6 CustomizeAction 类的实现

CustomizeAction 通常根据业务逻辑实现大量的自定义动作方法，在本框架

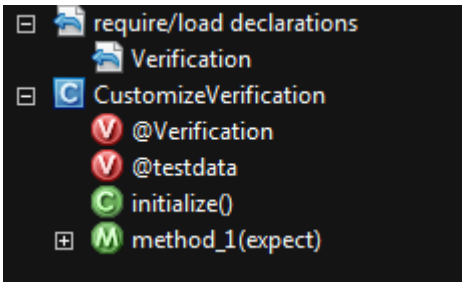


应用时，你需要根据你的业务逻辑在此类中增加大量 action 功能。

do_search 是我做的一个示例，我们看一下它的实现。

```
class CustomizeAction

def initialize(testdata)
@action = Action.new
@testdata = testdata
end

def do_search
@action.go_to(@testdata.get_test_object("baidu"))
@action.setvalue(@testdata.get_test_object("KeyWords"),'TestSolution.cn')
@action.click(@testdata.get_test_object("Search"))
return Core::PASS
end
```

enddo_search 实现了打开百度首页，之后输入 TestSolution.cn，最后点击搜索按钮，完成一组搜索动作。



4.7 CustomizeVerification 类的实现

CustomizeVerification 通常根据业务逻辑实现大量的自定义验证方法，在本框架应用时，你需要根据你的业务逻辑在此类中增加大量 verification 功能。

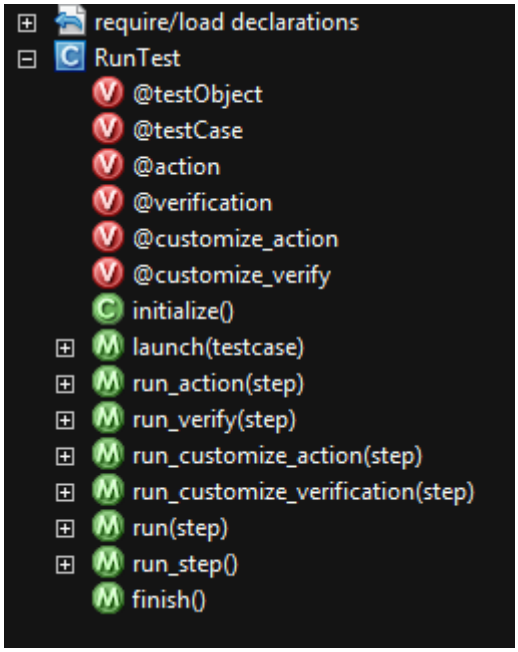
method_1 是示例，我们看一下实现过程。

```
class CustomizeVerification

def initialize
@Verification = Verification.new
@testdata = testdata
end

def method_1(expect)
return Core::FAIL if expect != "TestSolution.cn"
return Core::PASS
end
```

endmothod_1 只是简单实现了文本的对比功能。实际应用中我们可能会在这里填写验证订单中商品信息，如价格，库存，也可能同时验证支付金额，发票信息等。总之，这里你需要根据你的业务逻辑，实现大量的自定义验证方法。

4.8 RunTest 类的实现

RunTest 类也是非常重要的一个实现，其时它整个框架的调度中心，它既是框架的入口，又是调用 action，verification 及自定义动作及验证的指挥官，它同时还要完成测试结果的保存等功能。



RunTest 实现的方法有 launch，run_action，run_verify，run_customize_action，run_customize_verification，run，run_step 以及 finish。

launch 在之前曾有提到过，其功能主要是根据需要启动相应的浏览器，并将浏览器窗口最大化，其实现如下。

```
def launch(testcase)
Core.browser = Watir::Browser.new testcase["browser"].downcase.to_sym
Core.browser.window.maximize
end
```

run_action 用于执行一个动作，当其被调用时，TestCase 的 StepMethod 必须是 Action，我们先看一下其实现过程。

```
# When StepMethod is ACTION.
# callMethod should be some activities, just as CLICK/RESET/SET/CLEAR...
def run_action(step)
@action ||= Action.new
obj = @testObject.get_test_object(step["objectname"]) unless step["objectname"] == ""
# p obj
params = [] << step["callmethod"].downcase.to_sym << obj << step["requirevalue"]
params = params.compact
#p params
return @action.send(*params)
end
```

首先，我们从测试对象库中取出要使用的对象名，之后使用元编程技术将相关参数发送给系统，由系统来执行一个动作，即运行我们的 run_action。

run_verification 用于执行一个验证，当其被调用时，TestCase 的 StepMethod 必须是 Verification，我们看一下其实现过程。

```
# When StepMethod is VERIFICATION.
# callMethod should be basic checker, just as SET/UNSET/PRESENT/UNPRESENT.
# inputValue should be the elements on browser.
# expectValue should be the expect result.
def run_verify(step)
@verification ||= Verification.new
obj = @testObject.get_test_object(step["objectname"]) unless step["objectname"] == ""
params = [] << step["callmethod"].downcase.to_sym << obj << step["requirevalue"]
params = params.compact
return @verification.send(*params)
end
```

run_verification 也先从测试对象库中取出要使用的对象名，之后使用元编程技术将相关参数发送给系统，由系统来执行一个验证，即运行我们的 run_verification。

下面我们来看一下自定义的 run_action 及 run_verification 方法。

```
# When StepMethod is CustomizeAction.
# callMethod should be the name of customize methods.
# the params should be shown as step.inputValue.
def run_customize_action(step)
@customize_action ||= CustomizeAction.new
params = [] << step["callmethod"].downcase.to_sym << step["requirevalue"]
params = params.compact
return @customize_action.send(*params)
end

# When StepMethod is CustomizeVerification.
# callMethod should be the name of customize methods.
# the params can be contained step.inputValue and step.expectValue.
def run_customize_verification(step)
@customize_verify ||= CustomizeVerification.new
params = [] << step["callmethod"].downcase.to_sym << step["requirevalue"]
params = params.compact
return @customize_verify.send(*params)
end
```

这两个方法实现原理与 run_action 及 run_verification 是一致的，差别在于我们用的是 CustomizeAction 和 CustomizeVerification 来发送自定义动作或验证。

上述定义的四个方法是通过 run 方法调用并分别执行的，我们看一下是如何实现的。

```
# Entrance for different step.
# According to SepMethod, choose the method to call.
def run(step)
begin
case step["stepmethod"].downcase
when "action"
return run_action(step)
when "verification"
return run_verify(step)
when "cusaction"
return run_customize_action(step)
when "cusverify"
return run_customize_verification(step)
end
rescue
return Core::ERROR+ $!.to_s
end
end
```

大家可以看出，run 方法识别了 step[“stepmethod”]中的关键字，如果是 action，则调用 run_action，如果是 Verification，则调用 run_verify，如果是 cusaction，则调用 run_customize_action，如果是 cusverify 则调用 run_customize_verification 方法。

接下来我们看一下本框架的核心入口方法 run_step。

```
def run_step
testcases = @testCase.run_list_data
rpt = nil
if !testcases.nil?
testcases.each do |tc|
if tc["runflag"].to_s.downcase == 'y'
rpt = Reporter.new(tc["testcase"])
rpt.clear_testcase_directory
launch(tc)
steps = TestData.new(Core::TC_PATH + tc["testcase"] + ".xls","TestCase").test_step_data
if !steps.nil?
tc_result = []
timestart = Time.now
allstep = []
steps.each do |sp|
result = run(sp)
if sp["capture"].to_s.downcase == 'y'
sp["capture"] = rpt.capture
end
sp["acturevalue"] = result.to_s.gsub(/(PASS|FAIL|ERROR),/,"")
sp["result"] = result.to_s.scan(/(PASS|FAIL|ERROR)/).join
tc_result << result.to_s.scan(/(PASS|FAIL|ERROR)/).join
allstep << sp
end
```

```
if tc_result.include?("FAIL")
sresult = "FAIL"
elsif tc_result.include?("ERROR")
sresult = "ERROR"
else
sresult = "PASS"
end
tc["result"] = sresult
tc["runtime"] = Time.now - timestart

@testCase.save_txt_result(Core::REPORT_PATH + tc["testcase"] + ".txt",allstep)
@testCase.save_txt_result(Core::REPORT_PATH + tc["testcase"]
 + ".htm",rpt.testcase_report(tc["testcase"],allstep))
end
finish
end
end

@testCase.save_txt_result(Core::REPORT_PATH + "RunList.txt",testcases)
@testCase.save_txt_result(Core::REPORT_PATH + "index.htm",rpt.index_report(testcases))
end
end
```

首先，run_step 拿到要运行的测试用例列表，然后查看相应的测试用例的运行标记是否为 y,为 y 表示可以执行。接着调用 Reporter 对象先将测试结果目录删除，接着启动测试。

然后根据测试用例 id 读取相应的测试步骤信息。根据测试步骤里的信息调用 run 方法来运行相应的测试步骤。之后判断步骤信息是否要抓图，如果其标记位为 y,则抓图。

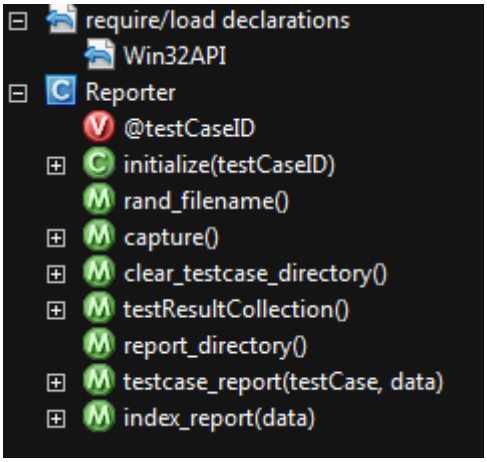
之后进行测试步骤结果的处理。循环处理完所有测试步骤后，处理测试用例的结果信息，得出其是成功还是失败。最后调用 Reporter 对象生成 html 结果。

```
def finish
Core.browser.close
end
```

finish 方法完成关闭掉已经启动了的浏览器功能。

4.9 测试结果的展现

Reporter 类实现了测试结果的展现功能，其主要实现了测试用例整体执行状态的 html 格式结果及单个测试用例步骤被执行的详细结果信息，也用 html 格式保存。



这里我们主要看一下 index_report 方法，它实现了测试用例整体执行状态的

用例编号	用例描述	浏览器	测试结果	运行时长	运行标记
PRE_CART_0001	验证注册用户未登录情况下，将商品加入购物车，进行结算功能正确	IE			N
PRE_CART_0002	验证未注册用户未登录情况下，商品加入购物车，进行结算功能正确	CHROME			N
BaiDu_0003	验证百度搜索功能	FF	✓	18.206026	N
PRE_CART_0004	验证商品详情页将商品加入购物车并去购物车结算功能正确	OPERA			N

html 格式结果信息，如下图所示：

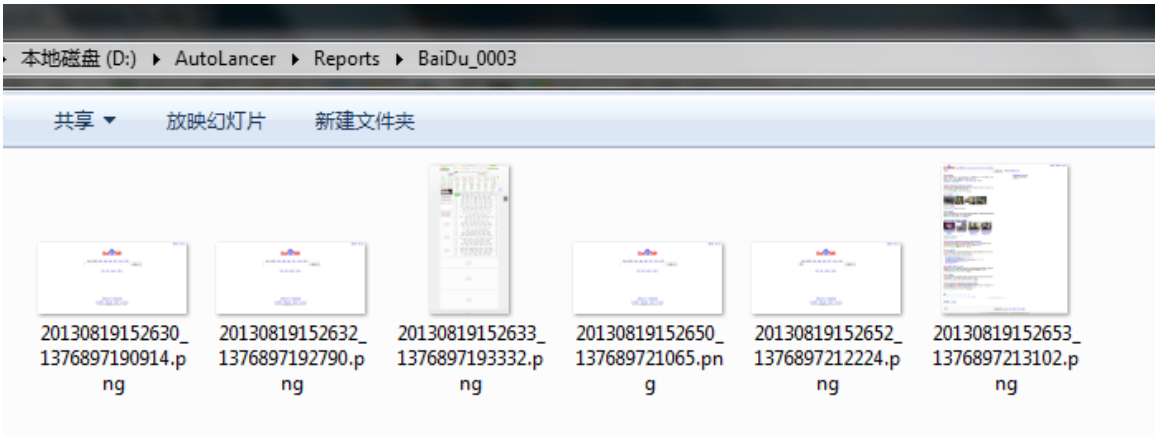
对于其实现，这里不过多讲述。

另外一个方法是 testcase_report,它实现了单个测试用例步骤被执行的详细结果信息，也用 html 格式保存，如下图所示。

步骤	描述	步骤方法	对象名称	调用方法	输入/预期值	实际值	结果	抓图链接
Step1	进入百度首页	Action	baidu	go_to			✓	点击查看
	检查hao123链接	Verification	hao123	check_text	hao123		✓	点击查看
	点击hao123	Action	hao123	click			✓	点击查看
Step2	进入百度首页	Action	baidu	go_to			✓	点击查看
	输入郝强	Action	KeyWords	setvalue	郝强		✓	点击查看
Step3	用户点击百度一下	Action	Search	click			✓	点击查看

我们在生成的 html 上点击一下最后一步骤的点击查看，会看到最后一步的测试结果， 如下图所示：

我们进入到框架的 Reports 目录中，查看一下抓取的图片 ，会如下图所示：



至此，你已经看到一个基本框架的功能已经完备了。

附录 需要了解的一些知识

使用本框架应具备的知识

为了能够使用本框架，您需要具备如下基础知识：

1. 掌握 ruby 语法，可以用 ruby 进行编程，本框架中部分功能需要您去扩展

CustomizeAction 和 CustomizeVerification 类。

2. 掌握 watir-webdriver 库。

3. 掌握 HTML 基本语法。

4. 掌握 Firebug，知道如何运用其对 web 元素关键属性进行获取。

框架新功能的展望

本框架目前还存在一定的限制：

1. 因为数据管理使用了微软 excel 来管理，致使本平台只能在 windows 平台上使用。

2. 没有邮件通知功能。

3. 不支持分布式执行。

对于新功能的展望来讲：

1. 我们要实现跨平台功能

2. 可以邮件结果通知功能

3. 支持分布式执行

4. 测试结果的自动统计等功能。

截止到本文完成之日起，对于新功能的展望部分提及的功能，我们已经实现并开始应用。