**哈尔滨工业大学软件学院**

**SQA大作业报告**

**测试工具：*Selenium Rspec+ Loadrunner***

**二〇一三年六月**

|  |
| --- |
|  |

目录

[目录 1](#_Toc359675978)

[Ruby + Selenium + Rspec进行web功能测试 3](#_Toc359675979)

[1. 简介 3](#_Toc359675980)

[2. 安装 3](#_Toc359675981)

[2.1 安装Ruby： 3](#_Toc359675984)

[2.2 安装Selenium： 3](#_Toc359675985)

[2.3 安装rspec： 4](#_Toc359675986)

[3. 使用 4](#_Toc359675987)

[3.1 Selenium教程 4](#_Toc359675994)

[3.1.1 第一个简单脚本： 4](#_Toc359675995)

[3.1.2 简单的浏览器操作： 4](#_Toc359675996)

[3.1.3 如何执行一段js 6](#_Toc359675997)

[3.1.4 页面元素定位 7](#_Toc359675998)

[3.1.5 定位frame中的元素 10](#_Toc359675999)

[3.1.6 捕获弹出窗口 12](#_Toc359676000)

[3.1.7 处理alert和confirm 14](#_Toc359676001)

[3.1.8 使用page object设计模式 15](#_Toc359676002)

[3.1.9 操作select下拉框 20](#_Toc359676003)

[3.1.10 智能等待页面加载完成 23](#_Toc359676004)

[3.1.11 使用jquery进行辅助测试 24](#_Toc359676005)

[3.1.12 Fire event的替代方案 26](#_Toc359676006)

[3.1.13 如何处理table 28](#_Toc359676007)

[3.2 Rspec教程： 31](#_Toc359676008)

[3.2.1 Rspec定义的方法 31](#_Toc359676009)

[3.2.2 文件目录结构： 32](#_Toc359676010)

[3.2.3 运行测试用例 32](#_Toc359676011)

[3.2.4 生成测试报告 32](#_Toc359676012)

[LoadRunner进行web压力测试 33](#_Toc359676013)

[1. LoadRunner基础知识 33](#_Toc359676014)

[1.1 LoadRunner简介 33](#_Toc359676015)

[1.2 LoadRunner 包含的组件 34](#_Toc359676016)

[1.3 LoadRunner 术语 34](#_Toc359676017)

[1.4 负载测试流程 34](#_Toc359676018)

[2. Vusr发生器 34](#_Toc359676019)

[2.1 脚本录制 35](#_Toc359676020)

[2.1.1 启动LoadRunner 35](#_Toc359676021)

[2.1.2 打开 VuGen，创建一个空白Web脚本 35](#_Toc359676022)

[2.1.3 使用 VuGen 向导模式 36](#_Toc359676023)

[2.2 脚本完善 37](#_Toc359676024)

[2.2.1 插入事务 37](#_Toc359676025)

[2.2.2 插入集合点 38](#_Toc359676026)

[3. 执行负载测试 39](#_Toc359676027)

[3.1创建场景 39](#_Toc359676028)

[3.2设计场景 40](#_Toc359676029)

[3.2.1 添加Load Generator 40](#_Toc359676030)

[3.2.2 配置全局计划 40](#_Toc359676031)

[3.2.3 配置运行时设置 41](#_Toc359676032)

[3.3运行场景 42](#_Toc359676033)

[4. 分析场景 42](#_Toc359676034)

[4.1 摘要报告 42](#_Toc359676035)

[4.2 Analysis常见图分析 43](#_Toc359676036)

[4.2.1 Vuser图 43](#_Toc359676037)

[4.2.2 每秒点击数图 43](#_Toc359676038)

[4.2.3 吞吐量图 44](#_Toc359676039)

[4.2.4 平均事务响应时间图 44](#_Toc359676040)

[5.总结 45](#_Toc359676041)

[5.1 使用总结 45](#_Toc359676042)

[5.2 LoadRunner总结 45](#_Toc359676043)

Ruby + Selenium + Rspec进行web功能测试

# 简介

Ruby：一种为简单快捷的面向对象编程（面向对象程序设计）而创的脚本语言。他强调系统设计必须注重人性化，而不是一味从机器的角度设想。目的是让编程的人更高兴。

Selenium：ThoughtWorks公司开发的web自动化测试工具。Selenium测试直接运行在浏览器中，就像真正的用户在操作一样。支持的浏览器包括IE、Mozilla Firefox、Mozilla Suite等。这个工具的主要功能包括：测试与浏览器的兼容性——测试你的应用程序看是否能够很好得工作在不同浏览器和操作系统之上。测试系统功能——创建衰退测试检验软件功能和用户需求。支持自动录制动作和自动生成。Net、Java、Perl等不同语言的测试脚本。Selenium 是ThoughtWorks专门为Web应用程序编写的一个验收测试工具。

Rspec：Ruby的一个测试框架插件，类似JAVA中的Junit，他为你更好的进行测试提供支持。利用Rspec可以轻松地实现测试驱动开发（敏捷开发模型中的一个主要方法）

# 安装



##### 安装Ruby：

Win7：<http://www.ruby-lang.org/zh_cn/downloads/> 下载安装包，单击运行即可

Linux：<http://ruby-china.org/wiki/install_ruby_guide> 执行命令安装即可

##### 安装Selenium：

打开ruby command console run命令：gem install selenium-webdriver

安装chrome浏览器或火狐浏览器，或使用ie。

下载ie浏览器driver

<http://code.google.com/p/selenium/downloads/detail?name=IEDriverServer_x64_2.33.0.zip>

下载chrome浏览器driver

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

下载 firefox 浏览器driver

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

将下载后的包放在ruby的安装路径下：Ruby200/bin/

##### 安装rspec：

打开ruby command console run命令：gem install rspec

# 使用



##### Selenium教程

* + 1. 第一个简单脚本：

1. require 'rubygems'
2. require 'selenium-webdriver'
4. driver = Selenium::WebDriver.for :firefox
5. driver.navigate.to "http://google.com"
6. sleep 3
8. element = driver.find\_element(:name, 'q')
9. element.send\_keys "Hello WebDriver!"
10. element.submit
12. puts driver.title
14. driver.quit
    * 1. 简单的浏览器操作：

如何打开一个测试浏览器

做自动化测试一般情况下我们都需要首先打开测试浏览器，浏览器开启后我们方可"命令"浏览器去打开新页面，点击特定的链接，判断具体的逻辑等等。因此该操作为"万里长征的第一步"，必须给以重视。具体代码如下。需要注意的是如果使用chrome进行测试，那么必须下载安装**chrome driver**。

1. require 'rubygems'
2. require 'selenium-webdriver'
3. # 打开firefox
4. dr = Selenium::WebDriver.for :firefox
5. dr = Selenium::WebDriver.for :ff
6. # 打开ie
7. dr = Selenium::WebDriver.for :ie
8. dr = Selenium::WebDriver.for :internet\_explorer
9. # 打开chrome
10. dr = Selenium::WebDriver.for :chrome

require 'rubygems'

require 'selenium-webdriver'

# 打开firefox

dr = Selenium::WebDriver.for :firefox

dr = Selenium::WebDriver.for :ff

# 打开ie

dr = Selenium::WebDriver.for :ie

dr = Selenium::WebDriver.for :internet\_explorer

# 打开chrome

dr = Selenium::WebDriver.for :chrome

如何打开1个具体的url

打开浏览器后我们需要转到我们的测试url。下面的代码可以达成这个目的。

1. require 'rubygems'
2. require 'selenium-webdriver'
3. dr = Selenium::WebDriver.for :firefox
4. # 使用get方法
5. dr.get url
6. # 使用navigate方法，然后再调用to方法
7. dr.navigate.to url

require 'rubygems'

require 'selenium-webdriver'

dr = Selenium::WebDriver.for :firefox

# 使用get方法

dr.get url

# 使用navigate方法，然后再调用to方法

dr.navigate.to url

如何关闭浏览器

测试结束后往往需要关闭浏览器，下面的代码可以完成这个任务。

1. require 'rubygems'
2. require 'selenium-webdriver'
3. dr = Selenium::WebDriver.for :firefox
4. dr.get url
5. # 使用quit方法
6. dr.quit
7. # 使用close方法
8. dr.close  require 'rubygems'

require 'selenium-webdriver'

dr = Selenium::WebDriver.for :firefox

dr.get url

# 使用quit方法

dr.quit

# 使用close方法

dr.close

如何返回当前页面的url

有时候我们需要返回当前测试页面的url。比如在使用soso进行搜索时，当我们提交了搜索请求后，soso返回的url应该是包含我们所需要搜索的关键字的。

例如如果我们搜索webdriver，那么提交搜索请求后，页面应当转到url为http://www.soso.com/q?pid=s.idx&cid=s.idx&w=webdriver的页面，这时候我们取到这个页面的url，然后通过正则表达式去匹配一下就能够得到我们所搜索的关键字了。具体代码如下。

1. require 'rubygems'
2. require 'selenium-webdriver'
3. dr = Selenium::WebDriver.for :firefox
4. url = 'http://www.soso.com'
5. dr.navigate.to url
6. search\_input = dr.find\_element :id => 's\_input'
7. search\_input.send\_keys 'webdriver'
8. search\_input.submit
9. match = dr.current\_url.match(/\b\w+$/)
10. keyword = match[0] if match

require 'rubygems'

require 'selenium-webdriver'

dr = Selenium::WebDriver.for :firefox

url = 'http://www.soso.com'

dr.navigate.to url

search\_input = dr.find\_element :id => 's\_input'

search\_input.send\_keys 'webdriver'

search\_input.submit

match = dr.current\_url.match(/\b\w+$/)

keyword = match[0] if match

如何返回当前页面的title

1. require 'rubygems'
2. require 'selenium-webdriver'
3. dr = Selenium::WebDriver.for :firefox
4. url = 'http://www.soso.com'
5. dr.navigate.to url
6. puts dr.title  require 'rubygems'

require 'selenium-webdriver'

dr = Selenium::WebDriver.for :firefox

url = 'http://www.soso.com'

dr.navigate.to url

puts dr.title

其他方法

* window\_handles : 返回当前所有打开浏览器的窗口句柄
* window\_handle ： 返回当前的浏览器的窗口句柄
* page\_source : 返回当前页面的源码
* visible? : 当前浏览器是否可见，并不保证支持所有浏览器
  + 1. 如何执行一段js

有时候在进行自动化测试时需要在页面上执行一段js脚本，这个时候就需要用到**execute\_script**方法了。

1. require 'rubygems'
2. require 'selenium-webdriver'
3. dr = Selenium::WebDriver.for :ie
4. url = 'http://www.soso.com'
5. dr.navigate.to url
6. sleep 3
7. js = <<JS
8. q = document.getElementById("tb");
9. q.style.border = "1px solid red";
10. JS
12. dr.execute\_script js

require 'rubygems'

require 'selenium-webdriver'

dr = Selenium::WebDriver.for :ie

url = 'http://www.soso.com'

dr.navigate.to url

sleep 3

js = <<JS

q = document.getElementById("tb");

q.style.border = "1px solid red";

JS

dr.execute\_script js

上面的代码打开了SoSo的首页，并高亮显示了id为"tb"的div。

下面的例子演示了在打开QQ首页的时候如何自动focus到页面上的soso搜索框

1. require 'rubygems'
2. require 'selenium-webdriver'
3. dr = Selenium::WebDriver.for :ie
4. url = 'http://www.qq.com'
5. dr.navigate.to url
6. sleep 3
7. js = <<JS
8. p = document.getElementById("smart\_input")
9. p.focus()
10. JS
12. dr.execute\_script js
    * 1. 页面元素定位

测试对象定位一般都是各种web自动化测试框架或工具的核心内容。

selenium-Webdriver的对象定位方法非常的丰富和强大。一般来说强大的对象定位都会提供如下的一些方法。

* 单个对象的定位方法
* 多个对象的定位方法
* 层级定位

selenium-Webdriver很好的支持了上述的3种定位方式。

定位单个对象

在定位单个对象时,selenium-Webdriver支持使用如下的一些属性对元素进行定位。

1. :class             => 'class name',
2. :class\_name        => 'class name',
3. :css               => 'css selector',
4. :id                => 'id',
5. :link              => 'link text',
6. :link\_text         => 'link text',
7. :name              => 'name',
8. :partial\_link\_text => 'partial link text',
9. :tag\_name          => 'tag name',
10. : xpath             => 'xpath',

:class => 'class name',

:class\_name => 'class name',

:css => 'css selector',

:id => 'id',

:link => 'link text',

:link\_text => 'link text',

:name => 'name',

:partial\_link\_text => 'partial link text',

:tag\_name => 'tag name',

: xpath => 'xpath',

使用class或class\_name进行定位

当所定位的对象具有class属性的时候我们可以通过class或class\_name来定位该对象。

下面的例子定位了soso首页上class为"new"的span。

1. require 'rubygems'
2. require 'selenium-webdriver'
3. require 'pp'
5. url = %q{http://www.soso.com/}
7. dr = Selenium::WebDriver.for :ie
8. dr.navigate.to url
9. sleep 1
11. new\_icon = dr.find\_element(:class => 'ico\_new')
12. puts new\_icon.tag\_name # ---> span  require 'rubygems'

require 'selenium-webdriver'

require 'pp'

url = %q{http://www.soso.com/}

dr = Selenium::WebDriver.for :ie

dr.navigate.to url

sleep 1

new\_icon = dr.find\_element(:class => 'ico\_new')

puts new\_icon.tag\_name # ---> span

使用id属性定位

soso首页的搜索输入框的html代码如下：

1. <input type="text" name="w" smartpid="sb.idx" smartch="sb.c.idx" autocomplete="off" id="s\_input" value="">

<input type="text" name="w" smartpid="sb.idx" smartch="sb.c.idx" autocomplete="off" id="s\_input" value="">

在进行定位前我们先动态定义highlight方法，该方法的作用是高亮显示有id属性的页面元素。

随后的代码演示了如何使用id属性来定位soso首页上的搜索输入框。

1. require 'rubygems'
2. require 'selenium-webdriver'
3. require 'pp'
5. Selenium::WebDriver::Element.module\_eval do
6. def highlight
7. e\_id = self.attribute('id')
8. puts "#{e\_id} = e\_id"
9. js = <<JS
10. document.getElementById("#{e\_id}").style.border = "3px solid red"
11. JS
12. @bridge.executeScript(js) if e\_id
13. end
14. end
16. url = %q{http://www.soso.com/}
18. dr = Selenium::WebDriver.for :ie
19. dr.navigate.to url
20. sleep 1
22. s\_input = dr.find\_element(:id => 's\_input')
23. # 或者使用语法糖衣
24. # s\_input = dr['s\_input']
25. s\_input.highlight

require 'rubygems'

require 'selenium-webdriver'

require 'pp'

Selenium::WebDriver::Element.module\_eval do

def highlight

e\_id = self.attribute('id')

puts "#{e\_id} = e\_id"

js = <<JS

document.getElementById("#{e\_id}").style.border = "3px solid red"

JS

@bridge.executeScript(js) if e\_id

end

end

url = %q{http://www.soso.com/}

dr = Selenium::WebDriver.for :ie

dr.navigate.to url

sleep 1

s\_input = dr.find\_element(:id => 's\_input')

# 或者使用语法糖衣

# s\_input = dr['s\_input']

s\_input.highlight

当使用id定位到正确的元素后，highlight方法会将该元素以红色高亮显示，借此也可以验证代码是否工作正常。

使用name属性定位

soso首页的搜索输入框的html代码如下：

1. <input type="text" name="w" smartpid="sb.idx" smartch="sb.c.idx" autocomplete="off" id="s\_input" value="">
3. # 同样定位soso首页的搜索框
4. s\_input = dr.find\_element(:name => 'w')

使用css属性定位<input type="text" name="w" smartpid="sb.idx" smartch="sb.c.idx" autocomplete="off" id="s\_input" value="">

# 同样定位soso首页的搜索框

s\_input = dr.find\_element(:name => 'w')

soso首页的搜索输入框的html代码如下：

1. <input type="text" name="w" smartpid="sb.idx" smartch="sb.c.idx" autocomplete="off" id="s\_input" value="">

<input type="text" name="w" smartpid="sb.idx" smartch="sb.c.idx" autocomplete="off" id="s\_input" value="">

官方文档上说Selenium-WebDriver支持css3语法，这对使用jquery的同学来说无疑是一个好消息。

1. # css属性定位soso首页搜索框
2. s\_input = dr.find\_element(:css => '#s\_input')

# css属性定位soso首页搜索框

s\_input = dr.find\_element(:css => '#s\_input')

使用xpath定位

soso首页的搜索输入框的html代码如下：

1. <input type="text" name="w" smartpid="sb.idx" smartch="sb.c.idx" autocomplete="off" id="s\_input" value="">
3. # 使用xpath定位soso首页搜索框
4. s\_input = dr.find\_element(:xpath => %Q{//input[@id='s\_input' and @name='w']})

使用其他方式定位<input type="text" name="w" smartpid="sb.idx" smartch="sb.c.idx" autocomplete="off" id="s\_input" value="">

# 使用xpath定位soso首页搜索框

s\_input = dr.find\_element(:xpath => %Q{//input[@id='s\_input' and @name='w']})

在定位link对象的时候，可以使用link和link\_text属性；

另外还可以使用tag\_name属性定位任意元素；

定位多个元素

**find\_elements**方法可以定位一组对象，例如

1. # 定位页面上所有的link对象
2. all\_links = driver.find\_elements(:tag\_name, 'a')
3. all\_links.each do {|l| puts l.class} # ---> Selenium::WebDriver::Element

# 定位页面上所有的link对象

all\_links = driver.find\_elements(:tag\_name, 'a')

all\_links.each do {|l| puts l.class} # ---> Selenium::WebDriver::Element

上面的代码返回页面上所有link对象的数组

下面代码演示了如何选取页面上所有的button对象：

1. all\_buttons = driver.find\_elements(:tag\_name, 'input').select do |i|
2. i['type'] == 'button'
3. end

all\_buttons = driver.find\_elements(:tag\_name, 'input').select do |i|

i['type'] == 'button'

end

层级定位

层级定位的思想是先定位父对象，然后再从父对象中精确定位出其我们需要选取的后代元素。

层级定位一般的应用场景是无法直接定位到需要选取的元素，但是其父元素比较容易定位，通过定位父元素再遍历其子元素选择需要的目标元素，或者需要定位某个元素下所有的子元素。

下面的代码演示了如何使用层级定位选取id为bm的div下所有的link元素，并打印出所有其href

1. links = dr.find\_element(:id, 'bm').find\_elements(:css => 'a')
2. links.each do |l|
3. puts l['href']
4. end
   * 1. 定位frame中的元素

在web ui自动化测试中，frame一直是令人头痛的问题，就像上班必须挤公车坐地铁一般，frame的问题总是令人气闷纠结为之黯然神伤。

以前在使用watir 1.6x的时候，frame也是颇为棘手的一个问题。不但要照本宣科的进行一系列的设置，而且在进行实际代码编写的过程中会遇到各种奇奇怪怪的问题。frame就像中国男足的后防线，问题多多难以解决。

selenium webdriver处理frame比较简单，这点比某些测试工具要先进一些，令人身心愉悦。

以下面的html代码为例，我们看一下如何定位frame上的元素。

1. frame.html
2. <html>
3. <head>
4. <title>Frame</title>
5. <style>
6. #f\_1 {width: 10em; height: 10em; border: 1px solid #ccc; }
7. #f\_2 {display: none}
8. </style>
9. </head>
10. <body>
11. <p id = "p">Outside frame</p>
12. <iframe id = "f\_1" f1" src = "part1.htm"></iframe>
13. <iframe id = "f\_2" src = "part2.htm"></iframe>
14. </body>
15. </html>
16. part1.htm
17. <html>
18. <head><title>Part1</title></head>
19. <body>
20. <p id = "f\_p">This is part 1</p>
21. <input id = "btn" type = "button" value = "click me" onclick = "alert('hello')" />
22. </body>
23. </html>

frame.html

<html>

<head>

<title>Frame</title>

<style>

#f\_1 {width: 10em; height: 10em; border: 1px solid #ccc; }

#f\_2 {display: none}

</style>

</head>

<body>

<p id = "p">Outside frame</p>

<iframe id = "f\_1" f1" src = "part1.htm"></iframe>

<iframe id = "f\_2" src = "part2.htm"></iframe>

</body>

</html>

part1.htm

<html>

<head><title>Part1</title></head>

<body>

<p id = "f\_p">This is part 1</p>

<input id = "btn" type = "button" value = "click me" onclick = "alert('hello')" />

</body>

</html>

switch\_to方法会new1个TargetLocator对象，使用该对象的frame方法可以将当前识别的"主体"移动到需要定位的frame上去。

1. require 'rubygems'
3. require 'selenium-webdriver'


7. dr = Selenium::WebDriver.for :firefox
9. frame\_file = 'file:///'+File.expand\_path(File.join(File.dirname(\_\_FILE\_\_), 'frame.html'))
11. dr.navigate.to frame\_file
13. # 定位default content上的p元素
15. p dr.find\_element(:id => 'p')
17. # 将当前识别主体移动到id为f\_1的frame上去
19. dr.switch\_to.frame('f\_1')
21. # 点击frame上的button
23. dr.find\_element(:id =>'btn').click # --> a alert will popup
25. # 此时再去定位frame外的p元素将出现错误
27. p dr.find\_element(:id => 'p') # --> error
29. # 将识别的主体切换出frame
31. dr.switch\_to.default\_content
33. p dr.find\_element(:id => 'p') # --> ok

require 'rubygems'

require 'selenium-webdriver'

dr = Selenium::WebDriver.for :firefox

frame\_file = 'file:///'+File.expand\_path(File.join(File.dirname(\_\_FILE\_\_), 'frame.html'))

dr.navigate.to frame\_file

# 定位default content上的p元素

p dr.find\_element(:id => 'p')

# 将当前识别主体移动到id为f\_1的frame上去

dr.switch\_to.frame('f\_1')

# 点击frame上的button

dr.find\_element(:id =>'btn').click # --> a alert will popup

# 此时再去定位frame外的p元素将出现错误

p dr.find\_element(:id => 'p') # --> error

# 将识别的主体切换出frame

dr.switch\_to.default\_content

p dr.find\_element(:id => 'p') # --> ok

webdriver的frame处理方式让人感觉那个不痛越来越轻松，这点进步值得肯定。

* + 1. 捕获弹出窗口

在web自动化测试中点击一个链接然后弹出新窗口是比较司空见惯的事情。

webdriver中处理弹出窗口跟处理frame差不多，以下面的html代码为例

1. window.html
3. <html>
5. <head><title>Popup Window</title></head>
7. <body>
9. <a id = "soso" href = "http://www.soso.com/" target = "\_blank">click me</a>
11. </body>
13. </html>

window.html

<html>

<head><title>Popup Window</title></head>

<body>

<a id = "soso" href = "http://www.soso.com/" target = "\_blank">click me</a>

</body>

</html>

下面的代码演示了如何去捕获弹出窗口

1. require 'rubygems'
3. require 'pp'
5. require 'selenium-webdriver'


9. dr = Selenium::WebDriver.for :firefox
11. frame\_file = 'file:///'.concat File.expand\_path(File.join(File.dirname(\_\_FILE\_\_), 'window.html'))
13. dr.navigate.to frame\_file
15. dr.find\_element(:id =>'soso').click
17. # 所有的window handles
19. hs = dr.window\_handles
21. # 当前的window handle
23. ch = dr.window\_handle
25. pp hs
27. pp ch
29. hs.each do |h|
31. unless h == ch
33. dr.switch\_to.window(h)
35. p dr.find\_element(:id => 's\_input')
37. end
39. end

require 'rubygems'

require 'pp'

require 'selenium-webdriver'

dr = Selenium::WebDriver.for :firefox

frame\_file = 'file:///'.concat File.expand\_path(File.join(File.dirname(\_\_FILE\_\_), 'window.html'))

dr.navigate.to frame\_file

dr.find\_element(:id =>'soso').click

# 所有的window handles

hs = dr.window\_handles

# 当前的window handle

ch = dr.window\_handle

pp hs

pp ch

hs.each do |h|

unless h == ch

dr.switch\_to.window(h)

p dr.find\_element(:id => 's\_input')

end

end

捕获或者说定位弹出窗口的关键在于获得弹出窗口的handle。

在上面的代码里，使用了window\_handles方法获取所有弹出的浏览器窗口的句柄，然后使用window\_handle方法来获取当前浏览器窗口的句柄，将这两个值的差值就是新弹出窗口的句柄。

在获取新弹出窗口的句柄后，使用switch\_to.window(new\_window\_handle)方法，将新窗口的句柄作为参数传入既可捕获到新窗口了。

如果想回到以前的窗口定位元素，那么再调用1次switch\_to.window方法，传入之前窗口的句柄既可达到目的。

* + 1. 处理alert和confirm

在selenium webdriver中，confirm和alert的处理再也不需要借助任何第三方工具了。

下面的html页面上有1个名为click的button，点击该button后就会弹出1个alert窗口。

1. <html>
2. <head>
3. <title>Alert</title>
4. </head>
6. <body>
7. <input id = "btn" value = "click" type = "button" onclick = "alert('hello');"/>
8. </body>
10. </html>

<html>

<head>

<title>Alert</title>

</head>

<body>

<input id = "btn" value = "click" type = "button" onclick = "alert('hello');"/>

</body>

</html>

selenium webdriver处理alert的代码如下：

1. require 'rubygems'
2. require 'selenium-webdriver'
4. dr = Selenium::WebDriver.for :firefox

7. frame\_file = 'file:///'.concat File.expand\_path(File.join(File.dirname(\_\_FILE\_\_), 'alert.html'))

10. dr.navigate.to frame\_file

13. dr.find\_element(:id =>'btn').click

16. a = dr.switch\_to.alert

19. puts a.text #--> hello

22. a.accept

require 'rubygems'

require 'selenium-webdriver'

dr = Selenium::WebDriver.for :firefox

frame\_file = 'file:///'.concat File.expand\_path(File.join(File.dirname(\_\_FILE\_\_), 'alert.html'))

dr.navigate.to frame\_file

dr.find\_element(:id =>'btn').click

a = dr.switch\_to.alert

puts a.text #--> hello

a.accept

上面代码的思路是先点击id为btn的按钮，然后a = dr.switch\_to.alert返回了1个alert element(暂时如此理解好了)并赋值给变量a。这样a就代表了alert,使用puts a.text语句可以输出alert的内容，这里会打印出'hello'。**a.accept**表示点击确认，当弹出窗口为confrim时，a.accept也表示确认，如果需要取消的话，那么则可以使用**a.dismiss**方法。

* + 1. 使用page object设计模式

在进行web前端自动化测试的过程中，Page Object设计模式可以称得上是杀人放火，居家旅行的常备武器。

Page Object将测试对象及单个的测试步骤封装在每个Page对象中，以page为单位进行管理。举例来说，在没有使用Page Object模式的情况下，脚本可能是这样写的：

1. require 'rubygems'
2. require 'selenium-webdriver'
4. url = 'www.soso.com'
5. dr = Selenium::WebDriver.for :ie
6. dr.navigate.to url
8. ua\_links = dr.find\_element(:id => 'ua').find\_elements(:css => 'a')
9. ua\_links[3].click
10. ua\_links[3].send\_keys(:enter)
11. sleep 2
12. dr.switch\_to.frame('login\_frame')
13. user = {:name => 'test', :psd => 'test'}
14. dr.find\_element(:id => 'u').send\_keys(user[:name])
15. dr.find\_element(:id => 'p').send\_keys(user[:psd])
16. dr.find\_element(:id => 'login\_button').click
17. sleep 2
18. dr.close

require 'rubygems'

require 'selenium-webdriver'

url = 'www.soso.com'

dr = Selenium::WebDriver.for :ie

dr.navigate.to url

ua\_links = dr.find\_element(:id => 'ua').find\_elements(:css => 'a')

ua\_links[3].click

ua\_links[3].send\_keys(:enter)

sleep 2

dr.switch\_to.frame('login\_frame')

user = {:name => 'test', :psd => 'test'}

dr.find\_element(:id => 'u').send\_keys(user[:name])

dr.find\_element(:id => 'p').send\_keys(user[:psd])

dr.find\_element(:id => 'login\_button').click

sleep 2

dr.close

这段脚本的作用是去到soso，然后点击【登陆】link，待登陆的dialog弹出后输入用户名和密码，点击【登陆】按钮进行登陆。

看上去这段脚本是不错的，因为其很好的完成了登陆的任务，但不妨深入思考一下，如果我们需要验证错误的用户名正确的密码的话，那么上面关于登陆的那几行脚本是不是需要重复写一遍？

答案是否定的，因为登陆的功能可以抽象成函数，如下所示：

1. def login usr, psd
2. dr.find\_element(:id => 'u').send\_keys(usr)
3. dr.find\_element(:id => 'p').send\_keys(psd)
4. dr.find\_element(:id => 'login\_button').click
5. end

def login usr, psd

dr.find\_element(:id => 'u').send\_keys(usr)

dr.find\_element(:id => 'p').send\_keys(psd)

dr.find\_element(:id => 'login\_button').click

end

这样就可以通过构造不同的数据，每次测试时只需要调用login函数就可以了。

再深入想一下，假设需要测试不输入用户名和密码直接点击登陆按钮的情况，那该怎么办呢？

首先最容易想到的一点是改造login函数，当然usr和psd没有传入的时候就直接点点击登陆按钮，这是没有问题的。不过如果本着代码增强可读性原则，亦可以定义一个名为login\_without\_usr\_psd的函数，如下所示

1. def login\_without\_usr\_psd
2. dr.find\_element(:id => 'login\_button').click
3. end

def login\_without\_usr\_psd

dr.find\_element(:id => 'login\_button').click

end

这样在编写测试用例代码的时候就可以直接凭借函数名来揣测出该函数的作用，起到了self explain的作用。在自动化测试代码中，这一点是被鼓励的。

不过这样多定义1个函数就会带来另外的问题 dr.findelement(:id => 'loginbutton').click这行代码就在loin函数和login\_without\_usr\_psd中重复使用。

为了解决这个有些丑陋的问题，使得代码能够稍微美化一点，可以定义另外一个函数来实现**点击登陆按钮**的功能。

1. def click\_login\_btn
2. dr.find\_element(:id => 'login\_button').click
3. end
5. # now login\_without\_usr\_psd will like below
6. def login\_without\_usr\_psd
7. click\_login\_btn
8. end
10. # login function will like this
11. def login usr, psd
12. dr.find\_element(:id => 'u').send\_keys(usr)
13. dr.find\_element(:id => 'p').send\_keys(psd)
14. click\_login\_btn
15. end

def click\_login\_btn

dr.find\_element(:id => 'login\_button').click

end

# now login\_without\_usr\_psd will like below

def login\_without\_usr\_psd

click\_login\_btn

end

# login function will like this

def login usr, psd

dr.find\_element(:id => 'u').send\_keys(usr)

dr.find\_element(:id => 'p').send\_keys(psd)

click\_login\_btn

end

好了，那么本着使代码更加灵活，self explain 特性更加明显的原则，我们又可以将**输入用户名**，**输入用户密码**功能抽象成函数，这时候login函数看起来可能会是这样的：

1. def login usr, psd
2. set\_usr usr
3. set\_psd psd
4. click\_login\_btn
5. end

def login usr, psd

set\_usr usr

set\_psd psd

click\_login\_btn

end

如果我们重复上面的步骤，一步一步的提升代码的复用性和可读性的话，我们就会发现除了上面演示的setusr，setpsd这样的【**基本动作**】外，有些测试对象也是可以复用的。比如在登陆的时候我们会用到密码输入框，也许在修改密码的时候我们也会用到这一对象。

很自然的就会想到，如果将一些测试对象以及操作这些测试对象的动作或步骤封装在1个类中，那么代码的灵活性和适用性将会更强。那么按照什么纬度来划分这些类呢？也是很自然的，就像睡醒了就会睁眼，挖完煤就要洗脸一样，我们会发现按照**页面**也就是page来组织这些类将是很好的解决方案。

这就是Page Object设计模式，将每一个测试页面抽象为1个Page类，并在该类中封装了本页面的测试对象和基本的测试步骤，以提高代码的可读性复用性通用性和一致性。Page Object设计模式带来的好处是显而易见的。比如使用了page object模式后，测试用例可能会如下面所示

1. soso\_page = Site.new(dr).soso\_main\_page(url).open
2. soso\_page.login wrong\_usr, wrong\_psd
3. soso\_page.error\_msg.should be\_eql('error')

soso\_page = Site.new(dr).soso\_main\_page(url).open

soso\_page.login wrong\_usr, wrong\_psd

soso\_page.error\_msg.should be\_eql('error')

测试用例简单且易读，而且代码的复用性极佳。其他用例需要使用到login功能时候只需要new 包含该功能的Page对象，调用login方法既可。

下面的代码演示了如何使用Page Object设计模式重构本文开头所实现的soso主页用户登陆功能。

**[ruby]** [view plaincopyprint?](http://blog.csdn.net/nbkhic/article/details/6897061)

1. base\_page.rb
3. class BrowserContainer
4. def initialize driver
5. **@dr** = driver
6. end
7. end # BrowserContainer
9. class Site < BrowserContainer
10. def soso\_main\_page url
11. **@soso\_main\_page** = SosoMainPage.new(**@dr**, url)
12. end
14. def close
15. **@dr**.close
16. end
17. end #Site
19. class BasePage < BrowserContainer
20. attr\_reader :url
22. def initialize dr, url
23. super(dr)
24. **@url** = url
25. end
27. def open
28. **@dr**.navigate.to **@url**
29. self
30. end
31. end #BasePage
33. class SosoMainPage < BasePage
34. require './login\_dialog'
35. include LoginDialog
37. def login usr, psd
38. open\_login\_dialog
39. to\_dialog\_frame
40. usr\_field.send\_keys usr
41. psd\_field.send\_keys psd
42. login\_btn.click
43. end
45. def open\_login\_dialog
46. login\_link.click
47. login\_link.send\_keys(:enter)
48. sleep 2
49. end
51. private
53. def ua\_links
54. **@dr**.find\_element(:id => 'ua').find\_elements(:css => 'a')
55. end
57. def login\_link
58. ua\_links[3]
59. end
60. end #SosoMainPage
62. login\_dialog.rb
64. module LoginDialog
65. def to\_dialog\_frame
66. begin
67. **@dr**.switch\_to.frame('login\_frame')
68. rescue
69. raise 'Can not switch to login dialog, make sure the dialog was open'
70. exit
71. end
72. end
74. def usr\_field
75. **@dr**.find\_element(:id => 'u')
76. end
78. def psd\_field
79. **@dr**.find\_element(:id => 'p')
80. end
82. def login\_btn
83. **@dr**.find\_element(:id => 'login\_button')
84. end
85. end #LoginDialog
87. login.rb
89. require 'rubygems'
90. require 'selenium-webdriver'
91. require './base\_page'
93. dr = Selenium::WebDriver.for :firefox
94. url = 'http://www.soso.com'
95. soso\_page = Site.new(dr).soso\_main\_page(url).open
96. soso\_page.login 'test', 'test'

base\_page.rb

class BrowserContainer

def initialize driver

@dr = driver

end

end # BrowserContainer

class Site < BrowserContainer

def soso\_main\_page url

@soso\_main\_page = SosoMainPage.new(@dr, url)

end

def close

@dr.close

end

end #Site

class BasePage < BrowserContainer

attr\_reader :url

def initialize dr, url

super(dr)

@url = url

end

def open

@dr.navigate.to @url

self

end

end #BasePage

class SosoMainPage < BasePage

require './login\_dialog'

include LoginDialog

def login usr, psd

open\_login\_dialog

to\_dialog\_frame

usr\_field.send\_keys usr

psd\_field.send\_keys psd

login\_btn.click

end

def open\_login\_dialog

login\_link.click

login\_link.send\_keys(:enter)

sleep 2

end

private

def ua\_links

@dr.find\_element(:id => 'ua').find\_elements(:css => 'a')

end

def login\_link

ua\_links[3]

end

end #SosoMainPage

login\_dialog.rb

module LoginDialog

def to\_dialog\_frame

begin

@dr.switch\_to.frame('login\_frame')

rescue

raise 'Can not switch to login dialog, make sure the dialog was open'

exit

end

end

def usr\_field

@dr.find\_element(:id => 'u')

end

def psd\_field

@dr.find\_element(:id => 'p')

end

def login\_btn

@dr.find\_element(:id => 'login\_button')

end

end #LoginDialog

login.rb

require 'rubygems'

require 'selenium-webdriver'

require './base\_page'

dr = Selenium::WebDriver.for :firefox

url = 'http://www.soso.com'

soso\_page = Site.new(dr).soso\_main\_page(url).open

soso\_page.login 'test', 'test'

上面的代码由3个文件组成。

base\_page.rb文件中定义了

* Site类 主要用于管理测试中所需要用到的各种页面，提供生成这些页面对象的快捷方法。比如Site.new(dr).soso\_main\_page(url)方法就实例化了1个SosoMainPage对象。
* BasePage类 所有Page对象的基类
* SosoMainPage类 代表了soso主页的Page Object类，封装了首页的一些测试对象，原子操作及基本步骤，如login

login\_dialog.rb文件中定义了代码登陆弹出框的LoginDialog。由于login dialog可能会出现在多个页面，比如qq音乐的登陆页面也有该弹出框，所以将其抽象成module，需要用到的页面直接include该module既可。

login.rb文件调用page object并实现了具体的测试逻辑，这个文件中可以使用你熟悉的测试框架来组织用例，如unit test和rspec等

关于Page Object设计模式的介绍就要告一段落了，从上面的代码中我们可以感觉到，Page Object模式加上测试用例框架就基本上可以等于简单的自动化测试框架了。所以在构建自动化测试框架的过程中，活用Page Object将为我们带来一系列的实惠和惊喜，就像超级的代金券一样，看上去不起眼但用起来却其乐无穷，浑身舒爽。

* + 1. 操作select下拉框

在selenium-webdriver中定位select list的方法比较简单，用id和name等属性可以很方便的将select给找出来，但是怎么去选择下拉框中的某一项呢？

思路是这样的，首先定位到select list元素，然后找出该select list下所有的option，点击该option element既可，以下面的html代码为例

**[ruby]** [view plaincopyprint?](http://blog.csdn.net/nbkhic/article/details/6897070)

1. <html>
2. <head>
3. <title>Select</title>
4. </head>
5. <body>
6. <span>select demo</span>
7. <select id = "s" name = "ns">
8. <option value = "0">Op1</option>
9. <option value = "1">Op2</option>
10. <option value = "2">Op3</option>
11. <option value = "3">Op4</option>
12. </select>
13. </body>
14. </html>

<html>

<head>

<title>Select</title>

</head>

<body>

<span>select demo</span>

<select id = "s" name = "ns">

<option value = "0">Op1</option>

<option value = "1">Op2</option>

<option value = "2">Op3</option>

<option value = "3">Op4</option>

</select>

</body>

</html>

通过下面的代码可以选择到下拉框中的第2个option，也就是text为Op2的选项

**[ruby]** [view plaincopyprint?](http://blog.csdn.net/nbkhic/article/details/6897070)

1. require 'rubygems'
2. require 'selenium-webdriver'
4. dr = Selenium::WebDriver.for :firefox
5. select\_file = 'file:///'.concat File.expand\_path(File.join(File.dirname(\_\_FILE\_\_), 'select.html'))
6. dr.navigate.to select\_file
7. dr.find\_element(:id => 's').find\_elements(:tag\_name => 'option')[1].click

require 'rubygems'

require 'selenium-webdriver'

dr = Selenium::WebDriver.for :firefox

select\_file = 'file:///'.concat File.expand\_path(File.join(File.dirname(\_\_FILE\_\_), 'select.html'))

dr.navigate.to select\_file

dr.find\_element(:id => 's').find\_elements(:tag\_name => 'option')[1].click

不难看出这样的代码还是不太直观的。为了能够更好的操作select元素，我们可以对其做一个简单的封装，示例代码如下：

**[ruby]** [view plaincopyprint?](http://blog.csdn.net/nbkhic/article/details/6897070)

1. easy\_wrap.rb
3. require 'rubygems'
4. require 'selenium-webdriver'
5. module EasyWrap
6. class Select
7. def initialize e
8. raise NotValidElementError unless e.is\_a?(Selenium::WebDriver::Element)
9. **@e** = e
10. child\_options
11. end
13. def select\_by\_value value
14. **@ops**.each do |op|
15. op.click if op['value'] == value
16. end
17. end
19. def select\_by\_index index
20. **@ops**[index].click if valid\_index? index
21. end
23. def select\_by\_text text
24. **@ops**.each do |op|
25. op.click if op['text'] == text
26. end
27. end
29. def valid\_index? index
30. index.to\_i < (**@ops**.size - 1)
31. end
33. def child\_options
34. begin
35. **@ops** = **@e**.find\_elements(:tag\_name => 'option')
36. rescue
37. raise CanNotFindOptionError, "can not find options in #{@o}"
38. exit
39. end
40. end
42. def options
43. **@ops\_text\_arr** = []
44. **@ops**.each do |op|
45. **@ops\_text\_arr** << op['text']
46. end
47. **@ops\_text\_arr**
48. end
49. end
51. class EasyWrapError < StandardError; end
52. class NotValidElementError < EasyWrapError; end
53. class CanNotFindOptionError < EasyWrapError; end
54. end #EasyWrap

easy\_wrap.rb

require 'rubygems'

require 'selenium-webdriver'

module EasyWrap

class Select

def initialize e

raise NotValidElementError unless e.is\_a?(Selenium::WebDriver::Element)

@e = e

child\_options

end

def select\_by\_value value

@ops.each do |op|

op.click if op['value'] == value

end

end

def select\_by\_index index

@ops[index].click if valid\_index? index

end

def select\_by\_text text

@ops.each do |op|

op.click if op['text'] == text

end

end

def valid\_index? index

index.to\_i < (@ops.size - 1)

end

def child\_options

begin

@ops = @e.find\_elements(:tag\_name => 'option')

rescue

raise CanNotFindOptionError, "can not find options in #{@o}"

exit

end

end

def options

@ops\_text\_arr = []

@ops.each do |op|

@ops\_text\_arr << op['text']

end

@ops\_text\_arr

end

end

class EasyWrapError < StandardError; end

class NotValidElementError < EasyWrapError; end

class CanNotFindOptionError < EasyWrapError; end

end #EasyWrap

通过引用该文件，我们的代码此时就应该是如下所示：

**[ruby]** [view plaincopyprint?](http://blog.csdn.net/nbkhic/article/details/6897070)

1. select.rb
3. require './easy\_wrap'
5. dr = Selenium::WebDriver.for :firefox
6. select\_file = 'file:///'.concat File.expand\_path(File.join(File.dirname(\_\_FILE\_\_), 'select.html'))
7. dr.navigate.to select\_file
8. s = EasyWrap::Select.new dr.find\_element(:id => 's')
9. # 选择value是1的option
10. s.select\_by\_value('1')
11. sleep 2
12. # 选择index是2的option,也就是第3个option
13. s.select\_by\_index(2)
14. sleep 2
15. # 选择text是Op4的option
16. s.select\_by\_text('Op4')
17. sleep 3
18. dr.close

select.rb

require './easy\_wrap'

dr = Selenium::WebDriver.for :firefox

select\_file = 'file:///'.concat File.expand\_path(File.join(File.dirname(\_\_FILE\_\_), 'select.html'))

dr.navigate.to select\_file

s = EasyWrap::Select.new dr.find\_element(:id => 's')

# 选择value是1的option

s.select\_by\_value('1')

sleep 2

# 选择index是2的option,也就是第3个option

s.select\_by\_index(2)

sleep 2

# 选择text是Op4的option

s.select\_by\_text('Op4')

sleep 3

dr.close

通过上面的代码可以看出，在选择了下拉框的某1个option时，如果下拉框上绑定有onchange事件，那么onchange事件是不会被触发的，这应该是上面解决方案的一个缺陷。

另外EasyWrap由于简单起见并没有支持选择多个option时的情况，好在选择多个option的情况一般不太多见，所以EasyWrap还是有一点实用价值的。

* + 1. 智能等待页面加载完成

web的自动化测试中，我们经常会遇到这样一种情况：点击1个按钮，页面上会弹出1个iframe，这时候脚本就需要去等待iframe加载完毕才能进行后续的操作。

在这种情况下，我们一般的处理思路是等待被等待对象上的某个子元素出现，当这个子元素出现时我们就认为该对象已经加载完毕，代码可以继续往下执行了。

selenium-webdriver为我们提供了一个Wait类来完成类似的等待功能。

下面的html代码实现了这样的一种效果：点击click按钮5秒钟后，页面上会出现一个红色的div块。我们需要写一段自动化脚本去捕获这个出现的div，然后高亮之。

**[ruby]** [view plaincopyprint?](http://blog.csdn.net/nbkhic/article/details/6897082)

1. set\_timeout.html
3. <html>
4. <head>
5. <title>Set Timeout</title>
6. <style>
7. .red\_box {background-color: red; width = 20%; height: 100px; border: none;}
8. </style>
9. <script>
10. function show\_div(){
11. setTimeout("create\_div()", 5000);
12. }
14. function create\_div(){
15. d = document.createElement('div');
16. d.className = "red\_box";
17. document.body.appendChild(d);
18. }
19. </script>
20. </head>
21. <body>
22. <button id = "b" onclick = "show\_div()">click</button>
23. </body>
24. </html>

set\_timeout.html

<html>

<head>

<title>Set Timeout</title>

<style>

.red\_box {background-color: red; width = 20%; height: 100px; border: none;}

</style>

<script>

function show\_div(){

setTimeout("create\_div()", 5000);

}

function create\_div(){

d = document.createElement('div');

d.className = "red\_box";

document.body.appendChild(d);

}

</script>

</head>

<body>

<button id = "b" onclick = "show\_div()">click</button>

</body>

</html>

下面的代码实现了高亮动态生成的div块的功能：

**[ruby]** [view plaincopyprint?](http://blog.csdn.net/nbkhic/article/details/6897082)

1. require 'rubygems'
2. require 'selenium-webdriver'
4. dr = Selenium::WebDriver.for :firefox
5. select\_file = 'file:///'.concat File.expand\_path(File.join(File.dirname(\_\_FILE\_\_), 'set\_timeout.html'))
6. dr.navigate.to select\_file
8. dr.find\_element(:id => 'b').click
9. wait = Selenium::WebDriver::Wait.new({:timeout => 30})
10. box = wait.until {dr.find\_element(:css => '.red\_box')}
11. dr.execute\_script('arguments[0].style.border = "5px solid yellow"', box) #div will be highlight

require 'rubygems'

require 'selenium-webdriver'

dr = Selenium::WebDriver.for :firefox

select\_file = 'file:///'.concat File.expand\_path(File.join(File.dirname(\_\_FILE\_\_), 'set\_timeout.html'))

dr.navigate.to select\_file

dr.find\_element(:id => 'b').click

wait = Selenium::WebDriver::Wait.new({:timeout => 30})

box = wait.until {dr.find\_element(:css => '.red\_box')}

dr.execute\_script('arguments[0].style.border = "5px solid yellow"', box) #div will be highlight

Wait类的构造方法Wait.new接收1个hash参数。上面代码中使用:timeout这个key值表示最长等待时间。

Wait类的until方法接收1个block代码块，如果代码块返回值不为true的话，该方法将一直等待直到达到最长等待时间为止。如果一旦代码块中的值为true了，则返回该代码块的返回值。box = wait.until {dr.find\_element(:css => '.red\_box')}的作用就是等待class为red\_box的div出现并返回该div对象。

进一步的思想下，如果某些页面在加载完成后会执行一些js函数，这些函数会延迟对dom树进行一些操作或者进行一些异步请求的处理，那么webdriver目前是无法智能的等待这些函数执行完毕的，所以有时候就会出现页面在没有加载完毕的情况下(实际上dom已经加载完毕，只是异步请求或延迟函数正在执行)，webdriver继续进行后续代码的执行情况。这时候我们就需要灵活的使用Wait类进行等待了。

* + 1. 使用jquery进行辅助测试

Jquery是当下比较流行的1个js框架，通过使用webdriver的execute\_script方法，我们可以将jquery库结合到自动化测试中去。

结合jquery进行自动化测试的思想是这样的：首先将jquery的源码读到1个string中去，然后使用execute\_script执行该string。执行完毕后我们就可以通过execute\_script方法来调用jquery库了。

下面的html代码中有一个隐藏的div，当鼠标移动到(mouseover)页面上名为Mouse Over Here的链接时，隐藏的div将会显示出来。

**[ruby]** [view plaincopyprint?](http://blog.csdn.net/nbkhic/article/details/6897089)

1. <html>
2. <head>
3. <title>FireEvent</title>
4. <style>
5. .mo {color: blue;}
6. .tips {display:none;border: 1px solid #ccc; background-color:#EFEFEF;width: 100px;height:100px}
7. </style>
8. <script>
9. function show\_tips(){
10. document.getElementById("t").style.display = "block";
11. }
12. function hide\_tips(){
13. document.getElementById("t").style.display = "none";
14. }
15. </script>
16. </head>
17. <body>
18. <a class = "mo" href = "#" onmouseover = "show\_tips()" onmouseout = "hide\_tips()">Mouse Over Here</a>
19. <div id = "t" class = "tips">This is the tips of link</div>
20. </body>
21. </html>

<html>

<head>

<title>FireEvent</title>

<style>

.mo {color: blue;}

.tips {display:none;border: 1px solid #ccc; background-color:#EFEFEF;width: 100px;height:100px}

</style>

<script>

function show\_tips(){

document.getElementById("t").style.display = "block";

}

function hide\_tips(){

document.getElementById("t").style.display = "none";

}

</script>

</head>

<body>

<a class = "mo" href = "#" onmouseover = "show\_tips()" onmouseout = "hide\_tips()">Mouse Over Here</a>

<div id = "t" class = "tips">This is the tips of link</div>

</body>

</html>

下面的代码使用jquery的库函数实现了不去触发Mouse Over Here链接而直接显示隐藏div的效果(仅在ruby1.9.2下测试过，ruby1.8x应该都不支持)

**[ruby]** [view plaincopyprint?](http://blog.csdn.net/nbkhic/article/details/6897089)

1. jquery\_helper.rb
3. #encoding: utf-8
4. module JqueryHelper
5. def load\_jquery dr,jquery\_path
6. jq = read\_jquery(jquery\_path)
7. jq.force\_encoding('utf-8')
8. dr.execute\_script jq
9. end
11. def read\_jquery(jquery\_path)
12. js = ''
13. File.open(File.expand\_path(jquery\_path), 'r') do |f|
14. js = f.read
15. end
16. js
17. end
18. end
20. fire\_event.rb
22. require 'rubygems'
23. require 'selenium-webdriver'
24. require './jquery\_helper'
25. include JqueryHelper
26. dr = Selenium::WebDriver.for :firefox
27. select\_file = 'file:///'.concat File.expand\_path(File.join(File.dirname(\_\_FILE\_\_), 'fire\_event.html'))
28. dr.navigate.to select\_file
30. jquery\_path = './jquery-1.6.4.min.js'
31. load\_jquery dr, jquery\_path
32. jq = <<JQ
33. $("#t").show();
34. JQ
36. dr.execute\_script jq

jquery\_helper.rb

#encoding: utf-8

module JqueryHelper

def load\_jquery dr,jquery\_path

jq = read\_jquery(jquery\_path)

jq.force\_encoding('utf-8')

dr.execute\_script jq

end

def read\_jquery(jquery\_path)

js = ''

File.open(File.expand\_path(jquery\_path), 'r') do |f|

js = f.read

end

js

end

end

fire\_event.rb

require 'rubygems'

require 'selenium-webdriver'

require './jquery\_helper'

include JqueryHelper

dr = Selenium::WebDriver.for :firefox

select\_file = 'file:///'.concat File.expand\_path(File.join(File.dirname(\_\_FILE\_\_), 'fire\_event.html'))

dr.navigate.to select\_file

jquery\_path = './jquery-1.6.4.min.js'

load\_jquery dr, jquery\_path

jq = <<JQ

$("#t").show();

JQ

dr.execute\_script jq

使用jquery来辅助测试实用性应该不是很强，不过有些时候可以使用jquery方法来获得dom节点的css属性，从而达到简化脚本的目的。

* + 1. Fire event的替代方案

webdriver里面已经没有了fire\_event方法，就像世界上再也没有萨达姆，本拉登和卡扎菲一样。

不过我们可以通过其他方法来实现fire\_event的相似功能。

考虑下面的html，当鼠标悬停到Mouse Over Here链接上时，js的mouseover事件被触发，shwo\_tips()函数将被执行，隐藏的tips div会显示在页面上。

**[ruby]** [view plaincopyprint?](http://blog.csdn.net/nbkhic/article/details/6897100)

1. <html>
2. <head>
3. <title>FireEvent</title>
4. <style>
5. .mo {color: blue;}
6. .tips {display:none;border: 1px solid #ccc; background-color:#EFEFEF;width: 100px;height:100px}
7. </style>
8. <script>
9. function show\_tips(){
10. document.getElementById("t").style.display = "block";
11. }
12. function hide\_tips(){
13. document.getElementById("t").style.display = "none";
14. }
15. </script>
16. </head>
17. <body>
18. <a class = "mo" href = "#" onmouseover = "show\_tips()" onmouseout = "hide\_tips()">Mouse Over Here</a>
19. <div id = "t" class = "tips">This is the tips of link</div>
20. </body>
21. </html>

<html>

<head>

<title>FireEvent</title>

<style>

.mo {color: blue;}

.tips {display:none;border: 1px solid #ccc; background-color:#EFEFEF;width: 100px;height:100px}

</style>

<script>

function show\_tips(){

document.getElementById("t").style.display = "block";

}

function hide\_tips(){

document.getElementById("t").style.display = "none";

}

</script>

</head>

<body>

<a class = "mo" href = "#" onmouseover = "show\_tips()" onmouseout = "hide\_tips()">Mouse Over Here</a>

<div id = "t" class = "tips">This is the tips of link</div>

</body>

</html>

如果存在fire\_event方法的话，直接在Mouse Over Here链接上触发oumouseover就能达到显示隐藏div的效果了，但是webdriver取消了fire\_event，所以这时候我们就需要求助于另一个功能强大的类，Action类。

Action类给用户提供了一些模拟用户交互方法，比如模拟key\_down，key\_up, double\_click等。

下面的代码使用Action类的move\_to方法模拟了鼠标的悬停事件，需要注意3点：

* Action类并不需要显示的实例化，调用时只需要通过driver.action直接调用该实例既可；
* move\_to方法执行完毕后悬停的效果也就消失了，所以代码中使用了循环10次的方法来人为"延长"事件的持续时间；
* 调用move\_to方法只是注册但并未真正的触发实际动作，需要调用**perform**方法来执行注册了的动作；

**[ruby]** [view plaincopyprint?](http://blog.csdn.net/nbkhic/article/details/6897100)

1. require 'rubygems'
2. require 'selenium-webdriver'
3. dr = Selenium::WebDriver.for :firefox
4. select\_file = 'file:///'.concat File.expand\_path(File.join(File.dirname(\_\_FILE\_\_), 'fire\_event.html'))
5. dr.navigate.to select\_file
7. m = dr.find\_element(:css => '.mo')
8. 10.times do
9. dr.action.move\_to(m).perform
10. end

require 'rubygems'

require 'selenium-webdriver'

dr = Selenium::WebDriver.for :firefox

select\_file = 'file:///'.concat File.expand\_path(File.join(File.dirname(\_\_FILE\_\_), 'fire\_event.html'))

dr.navigate.to select\_file

m = dr.find\_element(:css => '.mo')

10.times do

dr.action.move\_to(m).perform

end

另外也可以直接调用js引擎执行show\_tips函数，这样就不需要去模拟事件了。

**[ruby]** [view plaincopyprint?](http://blog.csdn.net/nbkhic/article/details/6897100)

1. js = %q[show\_tips();]
2. dr.execute\_script js
   * 1. 如何处理table

Table对象是自动化测试中经常需要处理的对象。由于webdriver中没有专门的table类，所以我们需要简单的封装出一个易用易扩展的Table类来帮助简化代码。

**[ruby]** [view plaincopyprint?](http://blog.csdn.net/nbkhic/article/details/6897118)

1. module EasyWrap
2. class EasyWrapError < StandardError;end
3. class NotValidElementError < EasyWrapError;end
4. class IncorrectdIndexError < EasyWrapError;end
6. class TableBase
7. attr\_reader :e
8. def initialize e
9. raise NotValidElementError unless e.is\_a?(Selenium::WebDriver::Element)
10. **@e** = e
11. end
13. def method\_missing(m, \*params, &blk)
14. **@e**.send(m, \*params) if **@e**.respond\_to?(m)
15. end
16. end #TableBase
17. class Table < TableBase
18. def initialize e
19. super(e)
20. \_\_rows
21. end
22. def \_\_rows
23. **@rows** = **@e**.find\_elements(:css => 'tr')
24. end
25. private :\_\_rows
27. def rows
28. all\_rows = []
29. **@rows**.each { |r| rows << TableRow.new(r)}
30. all\_rows
31. end
33. def row\_count
34. **@rows**.size
35. end
37. def [](index)
38. valid\_index? index
39. TableRow.new **@rows**[index]
40. end
42. def valid\_index?(index)
43. raise IncorrectdIndexError if index.to\_i > row\_count
44. end
45. end #Table
47. class TableRow < TableBase
48. def initialize e
49. super(e)
50. \_\_cells
51. end
53. def \_\_cells
54. **@cells** = **@e**.find\_elements(:tag\_name => 'td')
55. # 如果找不到td那么试着去找th
56. **@cells** = **@e**.find\_elements(:tag\_name => 'th') if **@cells**.empty?
57. end
58. private :\_\_cells
60. def cells
61. all\_cells = []
62. **@cells**.each {|c| all\_cells << TableCell.new(c)}
63. end
65. def cell\_count
66. **@cells**.size
67. end
69. def [](index)
70. valid\_index? index
71. TableCell.new **@cells**[index]
72. end
74. def valid\_index?(index)
75. raise IncorrectdIndexError if index.to\_i > cell\_count
76. end
77. end #TableRow
79. class TableCell < TableBase
80. end #TableCell
81. end #EasyWrap

module EasyWrap

class EasyWrapError < StandardError;end

class NotValidElementError < EasyWrapError;end

class IncorrectdIndexError < EasyWrapError;end

class TableBase

attr\_reader :e

def initialize e

raise NotValidElementError unless e.is\_a?(Selenium::WebDriver::Element)

@e = e

end

def method\_missing(m, \*params, &blk)

@e.send(m, \*params) if @e.respond\_to?(m)

end

end #TableBase

class Table < TableBase

def initialize e

super(e)

\_\_rows

end

def \_\_rows

@rows = @e.find\_elements(:css => 'tr')

end

private :\_\_rows

def rows

all\_rows = []

@rows.each { |r| rows << TableRow.new(r)}

all\_rows

end

def row\_count

@rows.size

end

def [](index)

valid\_index? index

TableRow.new @rows[index]

end

def valid\_index?(index)

raise IncorrectdIndexError if index.to\_i > row\_count

end

end #Table

class TableRow < TableBase

def initialize e

super(e)

\_\_cells

end

def \_\_cells

@cells = @e.find\_elements(:tag\_name => 'td')

# 如果找不到td那么试着去找th

@cells = @e.find\_elements(:tag\_name => 'th') if @cells.empty?

end

private :\_\_cells

def cells

all\_cells = []

@cells.each {|c| all\_cells << TableCell.new(c)}

end

def cell\_count

@cells.size

end

def [](index)

valid\_index? index

TableCell.new @cells[index]

end

def valid\_index?(index)

raise IncorrectdIndexError if index.to\_i > cell\_count

end

end #TableRow

class TableCell < TableBase

end #TableCell

end #EasyWrap

EasyWrap定义了3个类

* Table类，代表1个table对象，可以通过该对象来访问该table的行和列；
* TableRow类 代表table的某1行，可以通过其访问table的某一列；
* TableCell类 代表table的某1个单元格

假如我们需要点击table的第3行第1列，通过EasyWrap，我们可以这样来实现

**[ruby]** [view plaincopyprint?](http://blog.csdn.net/nbkhic/article/details/6897118)

1. talbe = EasyWrap::Table(dr.find\_element(:id => 'table\_id'))
2. table[2][0].click

talbe = EasyWrap::Table(dr.find\_element(:id => 'table\_id'))

table[2][0].click

以下面的html代码为例：

**[ruby]** [view plaincopyprint?](http://blog.csdn.net/nbkhic/article/details/6897118)

1. <html>
2. <head>
3. <title>Table</title>
4. <style>
5. table {border: 1px solid #ccc}
6. </style>
7. </head>
8. <body>
9. <table id = "t">
10. <th>C1</th><th>C2</th><th>C3</th>
11. <tr>
12. <td>v1</td>
13. <td>v2</td>
14. <td>v3</td>
15. </tr>
16. <tr>
17. <td>k1</td>
18. <td>k2</td>
19. <td>k3</td>
20. </tr>
21. </table>
22. </body>
23. </html>

<html>

<head>

<title>Table</title>

<style>

table {border: 1px solid #ccc}

</style>

</head>

<body>

<table id = "t">

<th>C1</th><th>C2</th><th>C3</th>

<tr>

<td>v1</td>

<td>v2</td>

<td>v3</td>

</tr>

<tr>

<td>k1</td>

<td>k2</td>

<td>k3</td>

</tr>

</table>

</body>

</html>

接下来的脚本实现了高亮页面上table元素首行首列的功能

**[ruby]** [view plaincopyprint?](http://blog.csdn.net/nbkhic/article/details/6897118)

1. require 'rubygems'
2. require 'selenium-webdriver'
3. require './easy\_wrap'
4. include EasyWrap
6. dr = Selenium::WebDriver.for :firefox
7. table\_file = 'file:///'+File.expand\_path(File.join(File.dirname(\_\_FILE\_\_), 'table.html'))
8. dr.get table\_file
9. t = dr.find\_element(:id => 't')
10. table = Table.new t
11. p table.row\_count
12. highlight = <<JS
13. arguments[0].style.border = "2px solid red"
14. JS
15. dr.execute\_script highlight, table[0][0].e
16. # 打印第2行第2列的值
17. puts table[1][1].text

##### Rspec教程：

要了解RSpec，我们首先需要了解什么是行为驱动开发（Behaviour Driven Development，简称BDD），BDD是一种融合了可接受性测试驱动计划（Acceptance Test Driven Planning），域驱动设计（Domain Driven Design）以及测试驱动开发（Test Driven Development，简称TDD）的敏捷开发模型。RSpec为BDD开发提供TDD支持。

我们将使用行为（behavior）和样例（example）来代替测试例（test case）和测试方法（test method）。

* + 1. Rspec定义的方法
       1. **describe()方法：**

我们用describe()方法定义一个测试用例组，describe()方法的参数可以是我们要测试的对象(如例中的People)，可以是一个字符串，用来描述该组，describe方法后的字符串所描述的内容可以对应一个用户故事。

注意describe()方法是可以嵌套的，两个describe()方法衔接起来可以更细化一个用户故事，如上边的里面的describe()方法内就表示：”People have enough money pay for house”。

* + - 1. **it()方法**：

我们用it()方法定义一个具体的测试用例（在RSpec中，称一个测试用例为一个example）。其后的字符串为该方法的参数，用来描述一个具体的场景，it方法体就是我们对系统在该场景下的行为的定义。

It()方法和describe()方法衔接起来，可以完整的描述一个系统行为，以上边的最后的一个测试用例为：”People have enough money pay for house should travel ”。

* + - 1. **context**()方法：

Context()方法和describe()方法的作用一样，不同点在于describe倾向于描述我们的测试对象，而context()方法倾向于用字符串描述用户故事。

before()和after()：

这两个方法很多测试框架都支持，需要说明的是这两个方法的参数，例子中为符号’:each’，表明对于每一个测试用例，先执行 before()方法中的代码，用例完成后执行after()方法中的代码，若参数为’:all’，则表示在所有的测试用例执行之前，先执行 before()方法中的代码，所有的用例完成后执行after()方法中的代码。

* + - 1. 共享**行为**：

系统的某一个行为是很多场景下都具有的，那么我们可以把它定义为一个共享行为，我们通过share\_examples\_for()方法 定义共享行为，使用it\_behaves\_like()方法共享定义过的系统行为，如例子中的share\_examples\_for “any people”， it\_behaves\_like “any people”。

* + - 1. **pending**()方法：

我们确定系统拥有一个行为，但是还没有具体定义，这时我们可以将该行为使用pending()方法来设置该行为为待定义，其后的字符串参数将在生成的报告中显示。

pending()方法的另外一种用法就是，当一个测试用例失败时，我们可以利用pending方法设置其为一个待修复的bug，方法体内包含使用例失败的代码。例如最后一个测试用例的内容。

* + 1. 文件目录结构：
       1. 文件path/spec/

在spec/下建立一系列测试用例命名为 XXX\_spec.rb。

* + 1. 运行测试用例

来到path/下 执行rspec –f doc 即可。

* + 1. 生成测试报告

来到path/下 执行rspec –f html >result.html 即可。

LoadRunner进行web压力测试

# LoadRunner基础知识

##### LoadRunner简介

LoadRunner是一种预测系统行为和性能的负载测试工具。以模拟上千万用户并发负载并实时监测系统性能的方式来确认和查找问题。LoadRunner能够对整个企业架构进行测试。通过使用LoadRunner，企业能最大限度地缩短测试时间，优化性能和加速应用系统的发布周期。

目前企业的网络应用环境都必须支持大量用户同时访问，网络体系架构中包含各类应用环境及由不同供应商提供的软件和硬件产品，难以预知的用户负载和愈来愈复杂的应用环境使公司不得不担心发送用户响应速度过慢、系统崩溃等问题，从而导致公司收益的损失。LoadRunner能让企业保护自己的收入来源，在无需购置额外硬件的情况下最大限度地利用现有的资源，并确保终端用户在应用系统的各个环节中对其测试应用的质量、可靠性和可扩展性都有良好的评价。

LoadRunner具有以下几个特点：

➤轻松创建虚拟用户

使用LoadRunner的Virtual User Generator，能便捷地创立起系统负载。该引擎能够生成虚拟用户，以虚拟用户的方式模拟真实用户的业务操作行为。它先记录下业务流程，然后将其转化为测试脚本。利用虚拟用户，可以在Windows、UNIX或Linux机器上同时产生成千上万个用户访问。所以LoadRunner能极大地减少负载测试所需的硬件和人力资源。另外，LoadRunner的TurboLoad专利技术能够提供很高的适应性，TurboLoad使您可以产生每天几十万名在线用户和数以百万的点击数负载。

➤创建真实的负载

Virtual Users建立后，需要确定负载方案、业务流程组合和虚拟用户数量；使用LoadRunner的Controller，能很快组织起多用户的测试方案。Controller的Rendezvous功能提供一个互动的环境，在这种环境下既能建立持续且循环的负载，又能管理和驱动负载测试方案，而且还可以利用它的日程计划服务来定义用户在什么时候访问系统以产生负载。完成上述步骤后，测试过程便自动化了。

➤定位性能问题

LoadRunner内含集成的实时监测器，在负载测试过程的任何时候，都可以观察到应用系统的运行性能。这些性能监测将实时显示交易性能数据和其他系统组件，包括Application Server、Web Server、网络设备和数据库等的实时性能。这样，就可以在测试过程中从客户和服务器双方评估这些系统组件的运行性能，从而更快地发现问题。

➤重复测试保证系统的高性能

负载测试是一个重复过程。每次处理完一个出错情况，都需要对应用程序在相同的方案下再进行一次负载测试，以检验所做的修改是否改善了运行性能。

##### LoadRunner 包含的组件

➤ Virtual User Generator 录制最终用户业务流程并创建自动化性能测试脚本，即 Vuser 脚本。

➤ Controller 组织、驱动、管理并监控负载测试。

➤ Load Generator 通过运行 Vuser 产生负载。

➤ Analysis 用于查看、剖析和比较性能结果。

➤ Launcher 使您可以从单个访问点访问所有 LoadRunner 组件。

##### LoadRunner 术语

场景：场景文件根据性能要求定义每次测试期间发生的事件。

Vuser: 在场景中， LoadRunner 用虚拟用户（或称 Vuser）代替真实用户。Vuser 模仿真实用户的操作来使用应用系统。一个场景可以包含数十、数百乃至数千个 Vuser。

Vuser脚本: Vuser 脚本描述 Vuser 在场景中执行的操作。

事务：要评测服务器性能，需要定义事务。事务代表要评测的终端用户业务流程。

##### 负载测试流程

负载测试一般包括 5 个阶段：规划、创建脚本、定义场景、执行场景和分析结果。



➤ 规划负载测试。定义性能测试要求，例如并发用户数量、典型业务流程和要求的响应时间。

➤ 创建 Vuser 脚本。在自动化脚本中录制最终用户活动。

➤ 定义场景。使用 LoadRunner Controller 设置负载测试环境。

➤ 运行场景。使用 LoadRunner Controller 驱动、管理并监控负载测试。

➤ 分析结果。使用 LoadRunner Analysis 创建图和报告并评估性能。

# Vusr发生器

Vuser发生器（Visual User Generation，简称VuGen）用来捕获最终用户业务流程和创建自动化测试脚本，即生成测试脚本、编辑和完善测试脚本的一个平台，支持C语言文法。

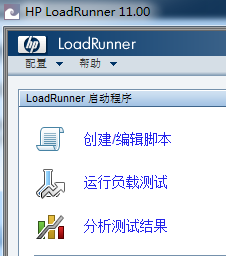
VuGen录制浏览器主要是通过代理的方式来实现的。开始录制时，VuGen打开浏览器，并以VuGen作为代理来访问目标服务器。这样，VuGen就可以捕获客户端与服务器之间通过的数据包。

在使用VuGen进行录制用户操作时，VuGen会对捕获的数据进行分析，并将期还原成对应协议的由API组成的脚本。同时，VuGen会将这些函数生成的脚步插入到VuGen编辑器中，以创建原始的Vuser脚本。

##### 脚本录制

* + 1. 启动LoadRunner

选择开始 > 程序 > HP LoadRunner > LoadRunner。这时将打开 HP LoadRunner主界面，如下图：



三个主链接会启动主要的三个组件

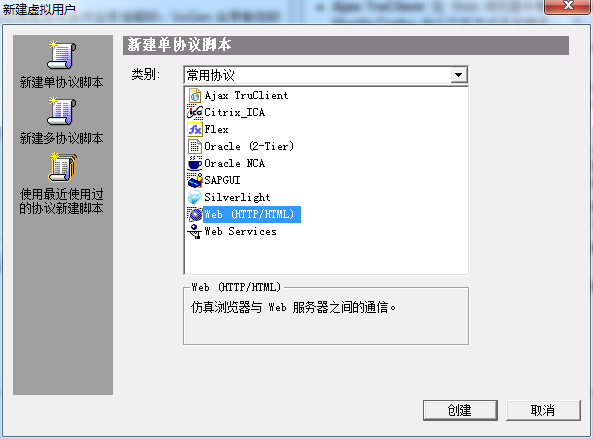
VuGen— 创建脚本VuGen

Controller — 设计和运行场景

Analysis — 分析场景

* + 1. 打开 VuGen，创建一个空白Web脚本

单击新建脚本按钮这时将打开“新建虚拟用户”对话框，LoadRunner显示“新建单协议脚本”选项。

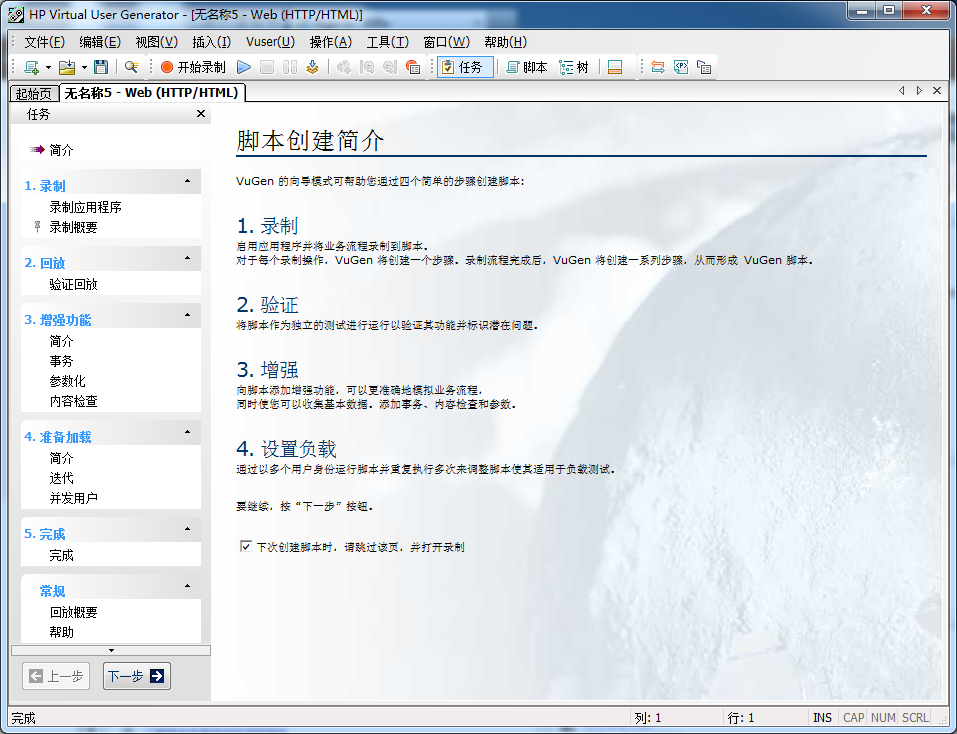


协议是客户端用来与系统后端进行通信的语言。我们测试的网站是一个基于Web的应用程序，因此您将创建一个Web Vuser脚本。在列表中选择Web(HTTP/HTML)并单击创建，创建一个空白Web脚本。

注：在多协议脚本中，高级用户可以在一个录制会话期间录制多个协议。录制其他类型的单协议或多协议脚本的过程与录制Web脚本的过程类似。

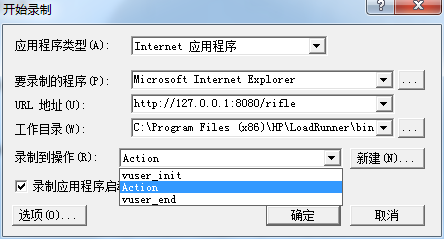
* + 1. 使用 VuGen 向导模式

打开 VuGen 的向导时将出现空白脚本，并且该向导的左侧将显示任务窗。（如果任务窗格没有显示，请单击工具栏上的“任务”按钮）VuGen 向导将指示您逐步创建脚本并根据所需的测试环境编辑此脚本。任务窗格列出了脚本创建过程中的每个步骤或任务。在您执行每个步骤时，VuGen 将在该窗口的主区域中显示详细的说明和规则。



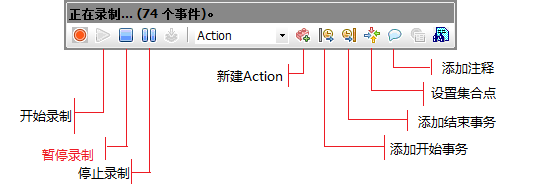
⑴单击“开始录制”

协议选择好后可以开始录制脚本，录制时系统弹出一个录制窗口，如图所示：



在URL地址中输入要录制的站点地址。录制到操作选项表示讲录制的代码放到哪个部分。

⑵开始录制后，会出现如图所示的工具栏



该工具栏从左到右依次代表开始录制、暂停录制、停止录制、新建Action、、添加开始事务标识、添加结束事务标识、设置集合点和添加注释。

⑶将自动打开IE浏览器，并显示输入URL地址相应的网页

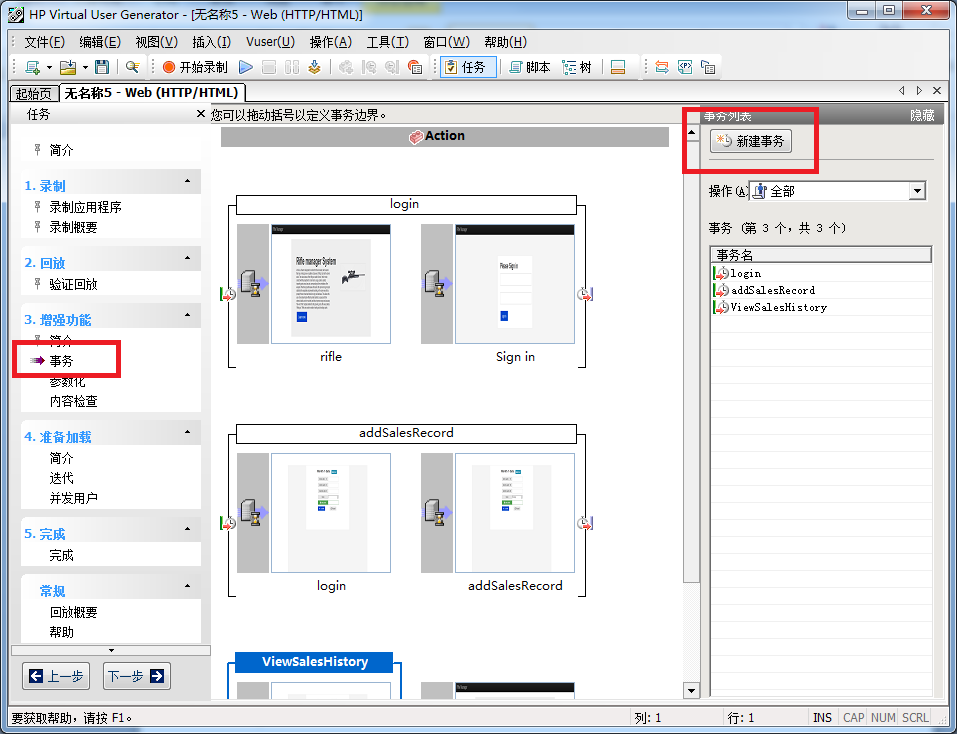
开始录制：

登录，并添加一组数据，然后查看6月分其他的销售情况，然后退出。

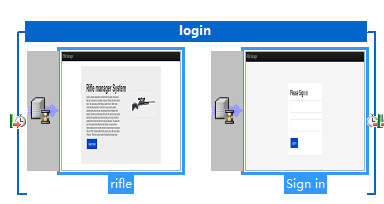
##### 脚本完善

* + 1. 插入事务

使用事务(Transaction)可以测出一个录制的过程中的一部分的执行时间。在Tasks面板中选“事务”，再在右边点“新建事务”就能创建，这里我们创建login事务。

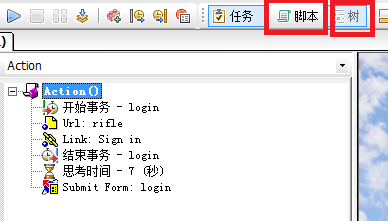


创建时可以选起始点和结束点



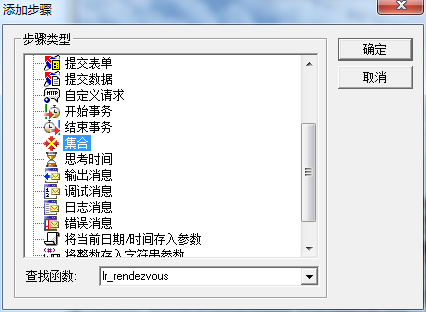
* + 1. 插入集合点

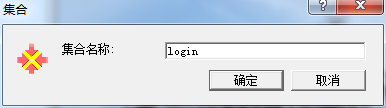
集合点可以在脚本录制过程中插入，也可以在脚本录制完成后插入，但需要注意的是，集合点只能插入Action部分的脚本中，不能插入vuser\_int和vuser\_end两部分脚本中。



有两种查看方式，一种是脚本，一种是树。点击“树”，则如上图所示。

选中开始事务-login，单击右键，选择在之前插入，双击集合，添加集合login





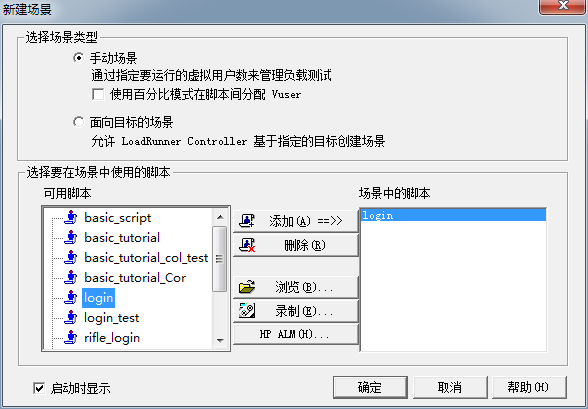
保存脚本，取文件名为login。

# 执行负载测试

Controller 是用来创建、管理和监控测试的中央控制台。使用 Controller 可以运行用来模拟实际用户执行的操作的示例脚本，并可以通过让多个虚拟用户同时执行这些操作来在系统中创建负载。

##### 3.1创建场景

打开LoadRunner，点击运行负载测试。默认情况下， LoadRunner Controller 打开时将显示“新建场景”对话框。



在可用脚本中选中我们刚刚创建的脚本login，点击添加，将其添加至新建的场景中。

有两种场景类型：

手动场景：可以控制正在运行的 Vuser 数目及其运行时间，另外还可以测试出应用程序可以同时运行的 Vuser 数目。可以使用百分比模式，根据业务分析员指定的百分比在脚本间分配所有的 Vuser。安装后首次启动 LoadRunner 时，默认选中百分比模式复选框。如果已选中该复选框，请取消选中。

面向目标场景：用来确定系统是否可以达到特定的目标。例如，可以根据指定的事务响应时间或每秒点击数/事务数确定目标，然后 LoadRunner 会根据这些目标自动创建场景。

Controller 窗口的设计选项卡分为三个主要部分：

“场景组”窗格：在“场景组”窗格中配置 Vuser 组。可以创建代表系统中典型用户的不同组，指定运行的 Vuser 数目以及运行时使用的计算机。

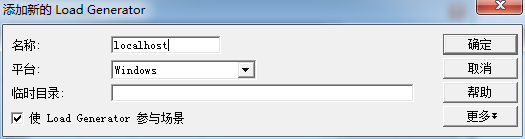
“服务水平协议”窗格：设计负载测试场景时，可以为性能指标定义目标值或服务水平协议 (SLA)。运行场景时， LoadRunner 收集并存储与性能相关的数据。分析运行情况时， Analysis 将这些数据与 SLA 进行比较，并为预先定义的测量指标确定 SLA 状态。

“场景计划”窗格：在“场景计划”窗格中，设置加压方式以准确模拟真实用户行为。可以根据运行 Vuser 的计算机、将负载施加到应用程序的频率、负载测试持续时间以及负载停止方式来定义操作。

##### 3.2设计场景

3.2.1 添加Load Generator

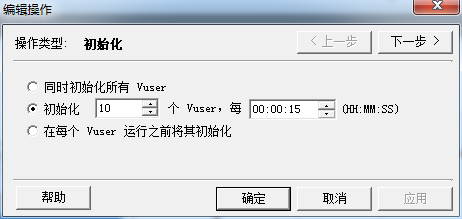
LoadRunner Controller 将在“Design”选项卡中打开场景(Scenario)。点击Scenario Scripts区域中的(Load Generators)，然后添加一个Load Generator。



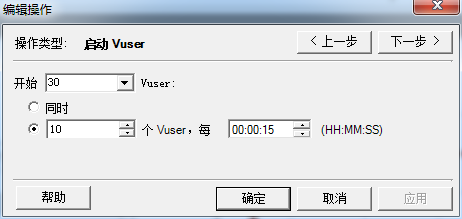
3.2.2 配置全局计划

⑴初始化

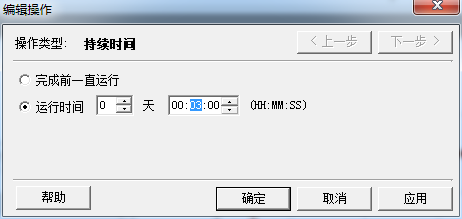
双击操作里初始化，选择“同时初始化所有Vuser”，来并发测试。



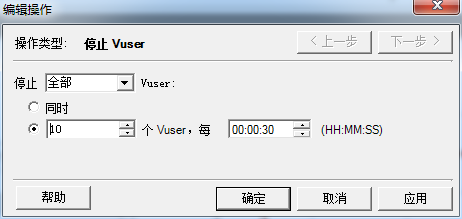
⑵启动Vuser



⑶设置持续时间

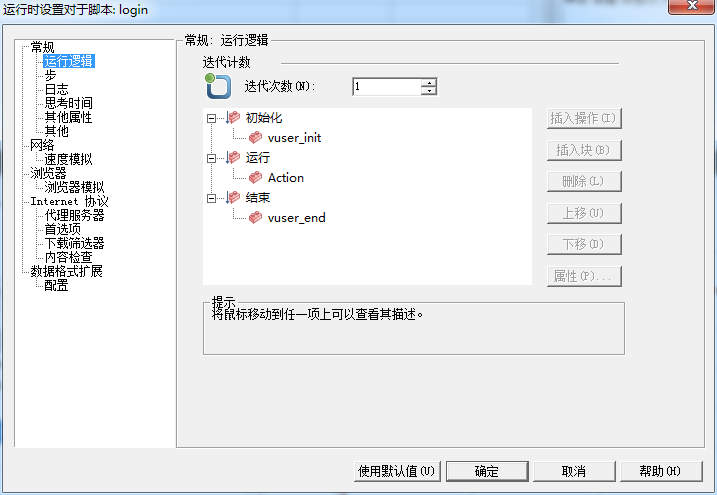


⑷停止Vuser



3.2.3 配置运行时设置

点击



通过运行时设置，可以模拟各种用户活动和行为。其中包括：

运行逻辑：用户重复一系列操作的次数。

步：重复操作之前等待的时间。

日志：希望在测试期间收集的信息的级别。

思考时间：用户在各步骤之间停下来思考的时间。

由于用户是根据其经验水平和目标与应用程序交互，因此，技术上更加精通的用户工作速度可能会比新用户快。通过启用思考时间，可使 Vuser 在负载测试期间更准确地模拟对应的真实用户。

速度模拟：使用不同网络连接（例如调制解调器、DSL 和电缆）的用户。

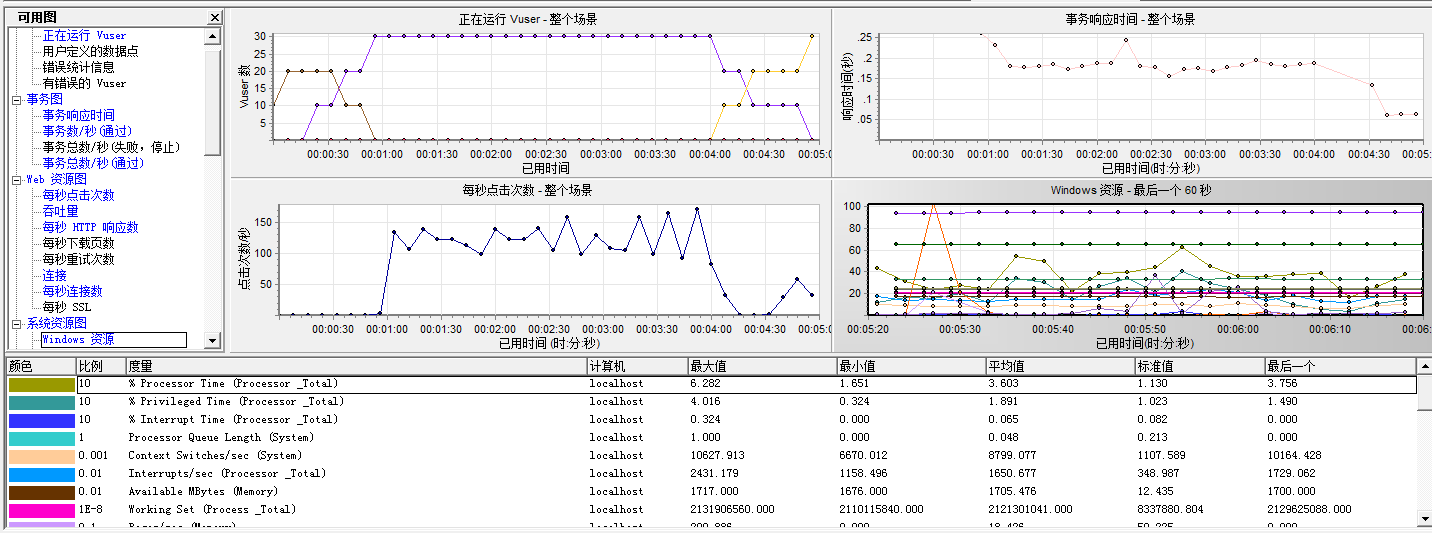
浏览器模拟：使用不同浏览器查看应用程序性能的用户。（一般只支持IE）

内容检查：用于自动检测用户定义的错误。

##### 3.3运行场景

切换到“运行”选项卡，在“运行”选项卡中主要包括两部分：场景组运行控制信息和数据图两部分。点击，场景运行。

测试过程中，测试的各种相关数据都会在不同的图中显示。在“可用图”窗格中，可用的图都用蓝色高亮，双击它们可以切换显示不同的图。



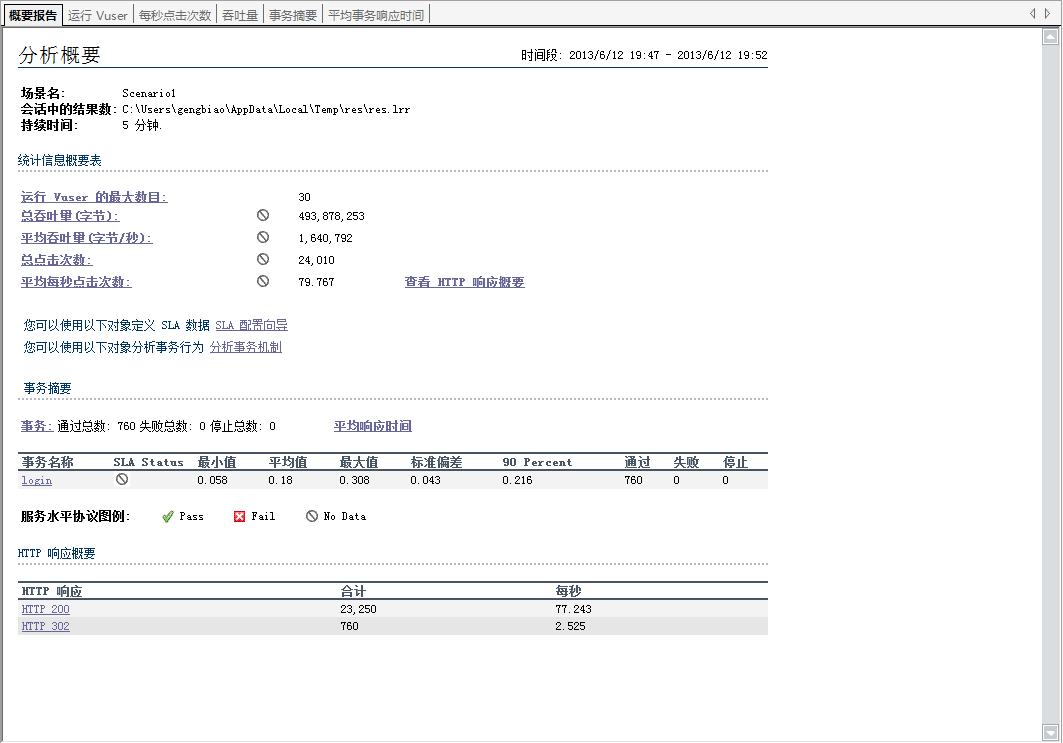
具体图的作用在下一节说明。

# 分析场景

点击Controller结果->分析结果，自动打开Analysis分析器，并自动载入测试结果。在会话浏览器中显示有一个报告，5个图。在左边可以看到报告和图的详细情况。点击Analysis 工具->HTML report，自动生产报告。

##### 4.1 摘要报告

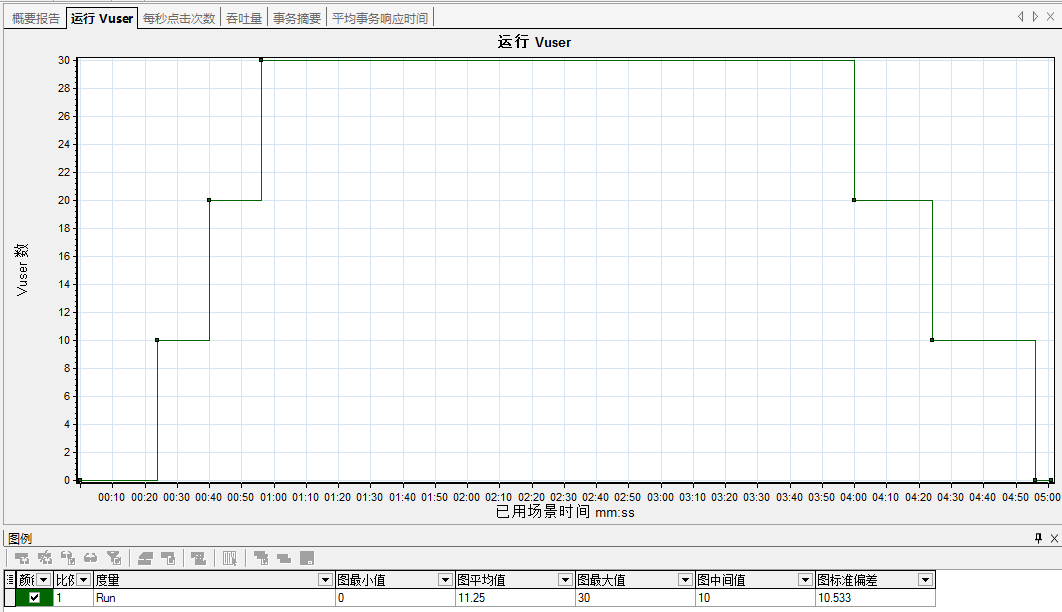
摘要报告提供了场景执行的一般信息。该报告始终存在于树视图或者作为Analysis窗口中的选项卡。主要包括摘要、统计、事务统计和HTTP响应统计四个部分。



##### 4.2 Analysis常见图分析

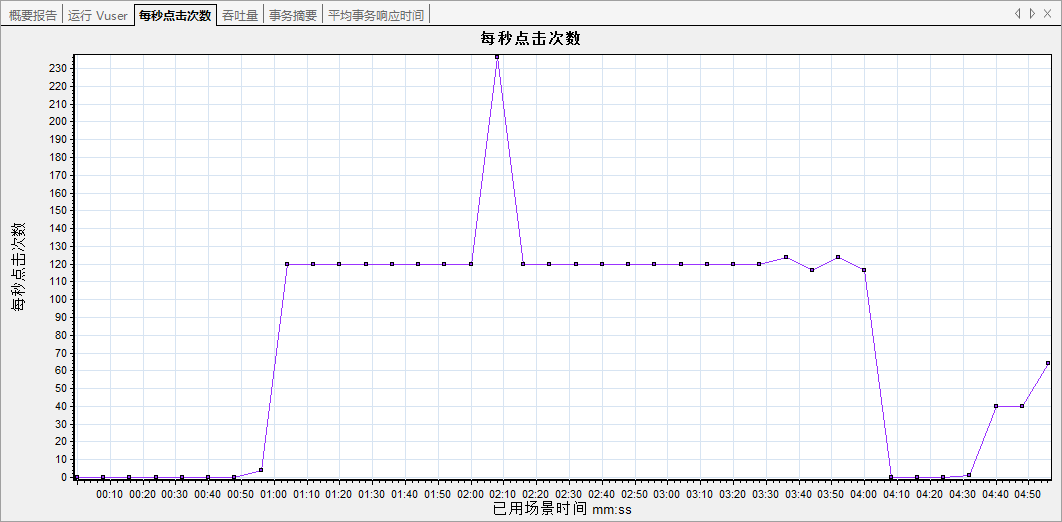
4.2.1 Vuser图

运行的Vuser图显示在测试期间的每一秒内，执行Vuser脚本的Vuser的数量及它们的状态。可以帮助确定任何给定环境中服务器上的Vuser负载。将此图与事务图结合使用可以确定Vuser的数量对事务响应时间产生的影响。



4.2.2 每秒点击数图

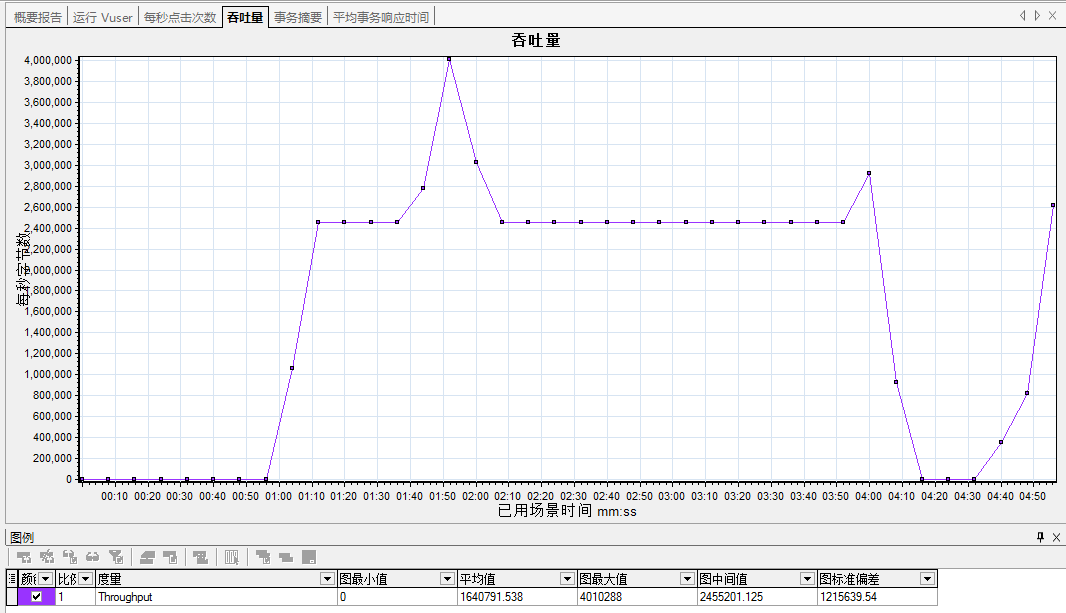
每秒点击数图显示在方案运行过程中Vuser每秒中向Web服务器提交的HTTP请求数。借助此图可以依据点击次数来评估Vuser产生的负载量。一般会将此图与平均事务响应时间图放在一起进行查看，观察点击数对事务性能产生的影响。



X轴表示方案从开始运行以来所用的时间，Y轴表示服务器上的点击数。通过上图能看出来随着方案的运行，每秒点击数趋于稳定，说明服务器运行比较稳定。

4.2.3 吞吐量图

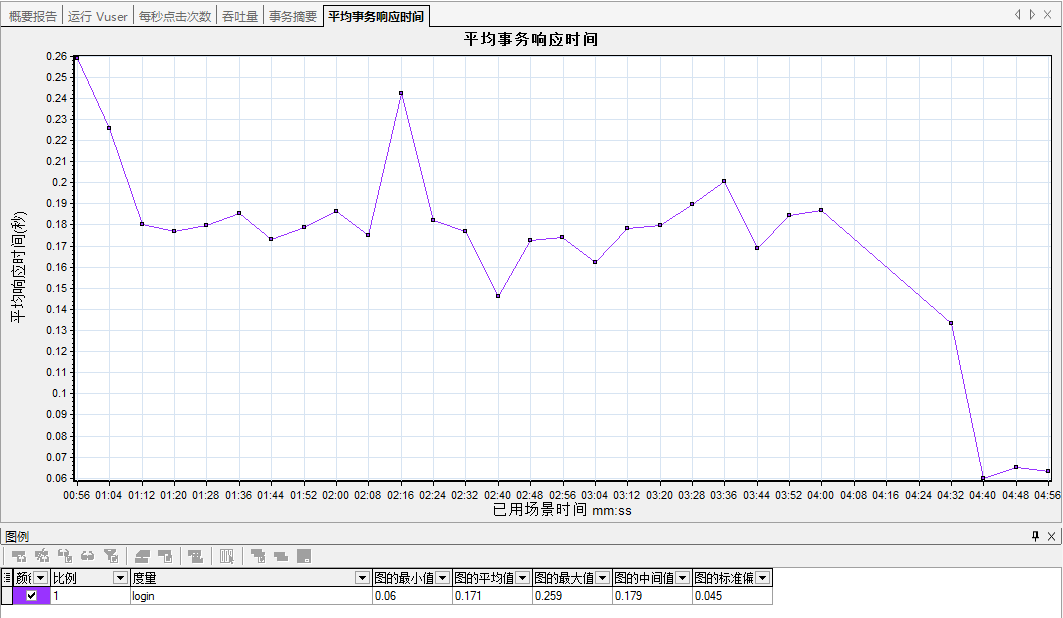
吞吐量显示方案运行过程中服务器上每秒的吞吐量。吞吐量表示Vuser在一秒时间内从服务器获得的数据量。借助此图可以依据服务器吞吐量来评估Vuser产生的负载量。可以和平均事务响应时间对照观察，以查看吞吐量对事务性能产生的影响。



上图显示随着服务器的运行，吞吐量趋于平衡状态，在2400000字节/秒附近波动。

4.2.4 平均事务响应时间图

平均事务响应时间图显示方案在运行期间执行事务所用的平均时间。平均事务响应时间最直接地反应了事务的性能情况。



# 5.总结

##### 5.1 使用总结

这次我们使用LoadRunner进行的两次测试都是基本的Web应用，除了基本的Web应用外，强大的LoadRunner还可以对C、VB、Ajax等进行测试。对所有这些，LoadRunner都提供了详细的文档。而且测试时使用的Scenario类型都是Manual Scenario，没有用Goal-Oriented Scenario。对于Goal-Oriented Scenario的使用，LoadRunner也有详尽的文档提供。但我们的两个测试已经足以表现用LoadRunner软件来测试的主要步骤与思想。

##### 5.2 LoadRunner总结

LoadRunner是著名的预测系统行为和性能的负载测试工具。与之类似的软件还有Apache JMeter、PureLoad等，但要和LoadRunner相比，大部分的这些工具都只能算是轻量级的。它们大多只限于对Web应用的测试，功能没有LoadRunner强大。

但也正因为LoadRunner功能强大，导致它体积大，我们这次使用的LoadRunner 11安装用的光盘镜像有近4G。而其他一些类似的工具大多连100M都不到。在软件的价格方面，LoadRunner也没有优势，虽然LoadRunner为不同需求的用户提供了不同类型的License，使用户不需要为他们用不着的功能花钱，但归根结底，它还是收费的。而其它的一些类似的工具，如Apache JMeter，甚至是开源的，不需要花钱。