目录

软件自动化测试大纲

在软件开发周期中，测试越来越被人们所重视，经过这些年测试行业的迅猛发展，测试也惭惭的在向技术性测试进行转变，随着敏捷开发这一模式的流行，自动化测试的重要性也越来越被测试团队所看重，快速的迭代，快速的回归，快速的响应所发现的问题，使自动化测试的作用越来越放大。本套课程重在带领大家去认识自动化测试，同时也让大家在认识的基础上，能独立设计自动化测试的框架，且能对项目中产生的问题进行分析，及解决问题。

课程内容：
本套课程主要分两大部分，第一部分主要包括环境搭建，基础知识入门，第二部分包括一些框架设计知识点。学习完这两课程，大家会对自动化测试及相关的工具会有一些认识，且在编码能力上会有一些提升，基本可以独立的完成一些自动化测试项目的设计及脚本。

第一部分：基础入门

第一周: SELENIUM2 的原理介绍及环境搭建

SELENIUM2 的原理

本节课主要讲解 SELENIUM2 的原理，让大家了解 SELENIUM2 的发展历程，同时解惑大家对自动化测试中产生的一些误区。

自动化测试最近几年在测试人员中很火，出去面试时，要是说不会自动化测试，都能感觉到面试官的鄙夷之情溢于言表，甚至在公司做自动化测试的同事都感觉比做手工测试的有技术含量一些，正因为有这样的情怀，所以导致自动化测试越来越火，但是火的背后，却伴随着很多的盲目，盲目的追求自动化，盲目的夸大自动化的作用，甚至投入很多的人力物力去做自动化，但往往最后的效果并不太理想，失败的案例太多太多，于是，我们是时候停下来去冷静的思考一下，同时提出：做高质量的自动化测试。

在做好自动化之前，我们得搞清楚几个问题：

1. 是不是所有的项目都适合自动化测试？答案是否定的，并不是所有的项目适合自动化。列举一些不适合自动化的情况：
   - 短平快的项目(一次性的项目)，做完交付即可，没有后期维护，这样的项目不适合做自动化，做了也白做
   - 项目周期短的项目，要你一周做完，你有时间去写自动化吗？
   - 易用性测试，这些不适合做自动化，这些属于产品经理的事情
   - 系统不稳定，这个也不适合自动化，系统都不稳定，盲目的去自动化，只会让团队疲于奔命。
   - 涉及一些与硬件交互的系统，这也不适合自动化。
2. 自动化测试的效率体现在，反复运行，快速运行，快速回归，所以，自动化测试适合于敏捷项目中，快速迭代，手工测试新功能，自动化回归老功能。
3. 自动化测试能发现新的 BUG 吗？答案也是否定的，自动化测试不会发现新的 BUG，只是限于对老功能中的一些问题。

自动化测试，工具很多，如何选择工具？ 我们简单比较一下大家都熟悉的 QTP 与 selenium2

- QTP，商业工具，笨重，但是功能强大，易上手，是一套完整的自动化解决方案，且适用于 C/S,B/S 结构
- Selenium2,开源工具，轻量级，适合多种语言编写，能在多种主流浏览器上运行，但是只支持 B/S 系统，且需要一定的编程基础，上手比较难

了解这些后，我们来看一下 selenium2 的原理：

Selenium1 是 thoughtworks 公司的一个产品经理为了解决重复烦燥的验收工作，写的一个自动化测试工具，其是用 JS 注入的方式来模拟人工的操作，但是由于 JS 的同源策略，也就是 JS 只能在一个域中的页面进行通讯，如果跨域，则就不能访问了，所以导致了如果页面中有 iframe 时，就无法操作了。Selenium2 是 selenium+webdriver 的产物，webdriver 是 google 的产品，google 收购 selenium1 后，整合了其自己的 webdriver,推出了 selenium webdriver,也就

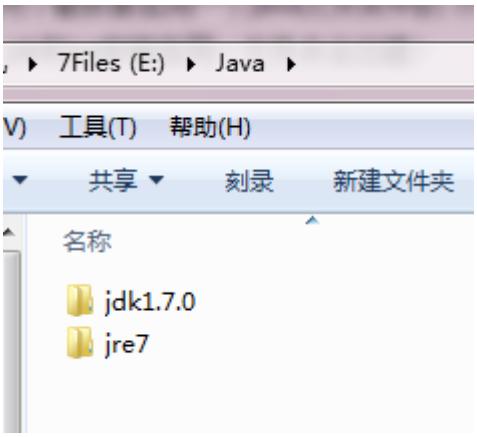
是我们所说的 selenium2。Selenium2 很好的解决了这个 JS 注入的安全性问题，其实现原理就从根本上改变了，selenium2 在启动 webdriver 的过程中,会首先确认比较浏览器与 driver 是否匹配，如果匹配，就会启动浏览器，然后把浏览器绑定在某一个端口，且在浏览器中启动一整套 web service,这套 web service 使用了 selenium2 自己定义的协议，这套协议在主流浏览器中是通用的，然后浏览器此时就作为了一个 server，测试脚本就作为了一个 client，client 发送的任何一个 selenium2 的 API 都被转成了一个 http request，当 web service 接收到这个 request 后，就会响应，这套协议就会告诉浏览器这时候需要干什么事。

环境搭建中 JDK 的下载及安装

JDK 下载地址：
http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html
选择好对应的系统，下载

安装 JDK 选择安装目录 安装过程中会出现两次 安装提示 。第一次是安装 jdk ，第二次是安装 jre 。建议两个都安装在同一个 java 文件夹中的不同文件夹中。(不能都安装在 java 文件夹的根目录下，jdk 和 jre 安装在同一文件夹会出错）

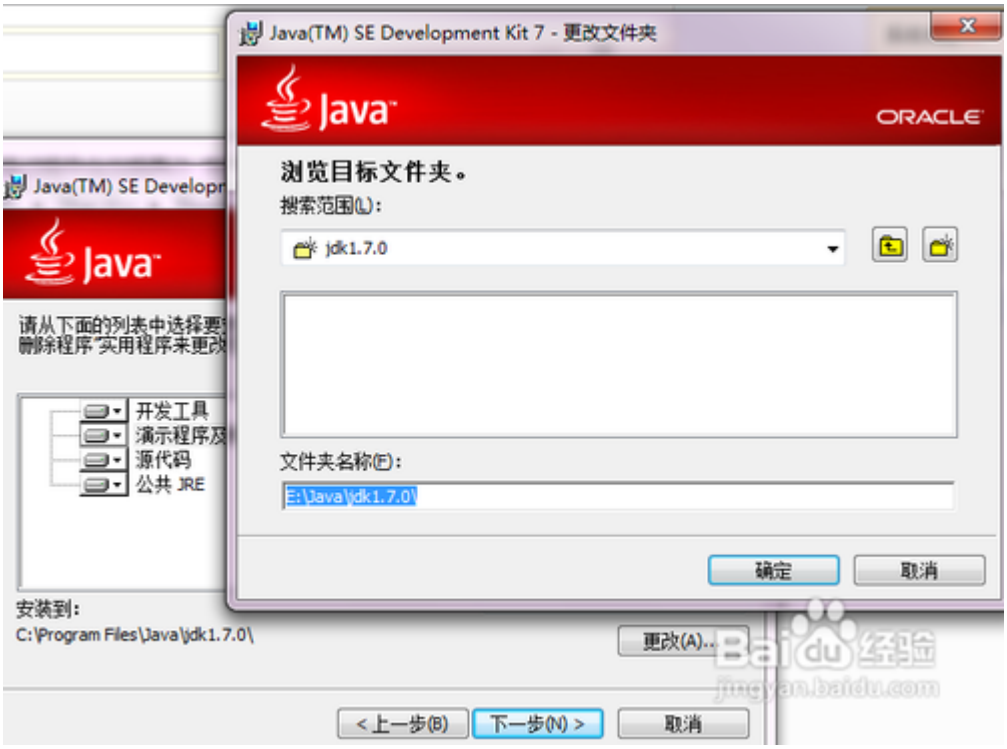
如下图所示



- 1：安装 jdk 随意选择目录 只需把默认安装目录 \java 之前的目录修改即可

2：安装 jre→更改→ \java 之前目录和安装 jdk 目录相同即可

注：若无安装目录要求，可全默认设置。无需做任何修改，两次均直接点下一步。



- 安装完 JDK 后配置环境变量　计算机→属性→高级系统设置→高级→环境变量

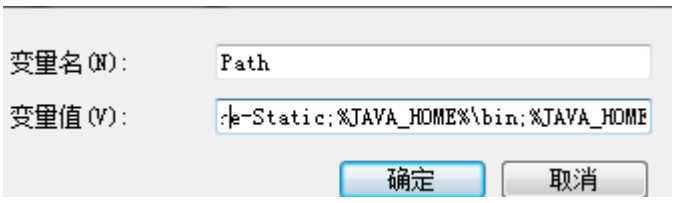
- 系统变量→新建 JAVA_HOME 变量 。

变量值填写 jdk 的安装目录（本人是 E:\Java\jdk1.7.0)

系统变量→寻找 Path 变量→编辑

在变量值最后输入 %JAVA_HOME%\bin;%JAVA_HOME%\jre\bin;

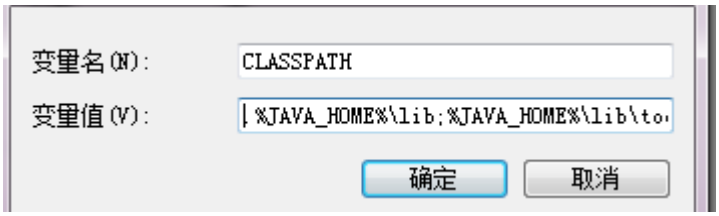
（注意原来 Path 的变量值末尾有没有;号，如果没有，先输入；号再输入上面的代码）



- 系统变量→新建 CLASSPATH 变量

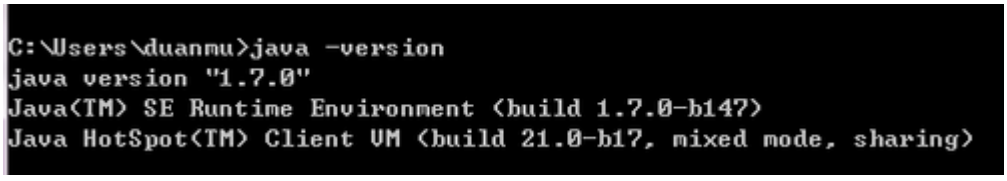
变量值填写 .;%JAVA_HOME%\lib;%JAVA_HOME%\lib\tools.jar（注意最前面有一点）

系统变量配置完毕



- 检验是否配置成功 运行 cmd 输入 java -version （java 和 -version 之间有空格）

若如图所示 显示版本信息 则说明安装和配置成功。



Eclipse 的配置，SELENIUM2 的 JAR 包下载及配置

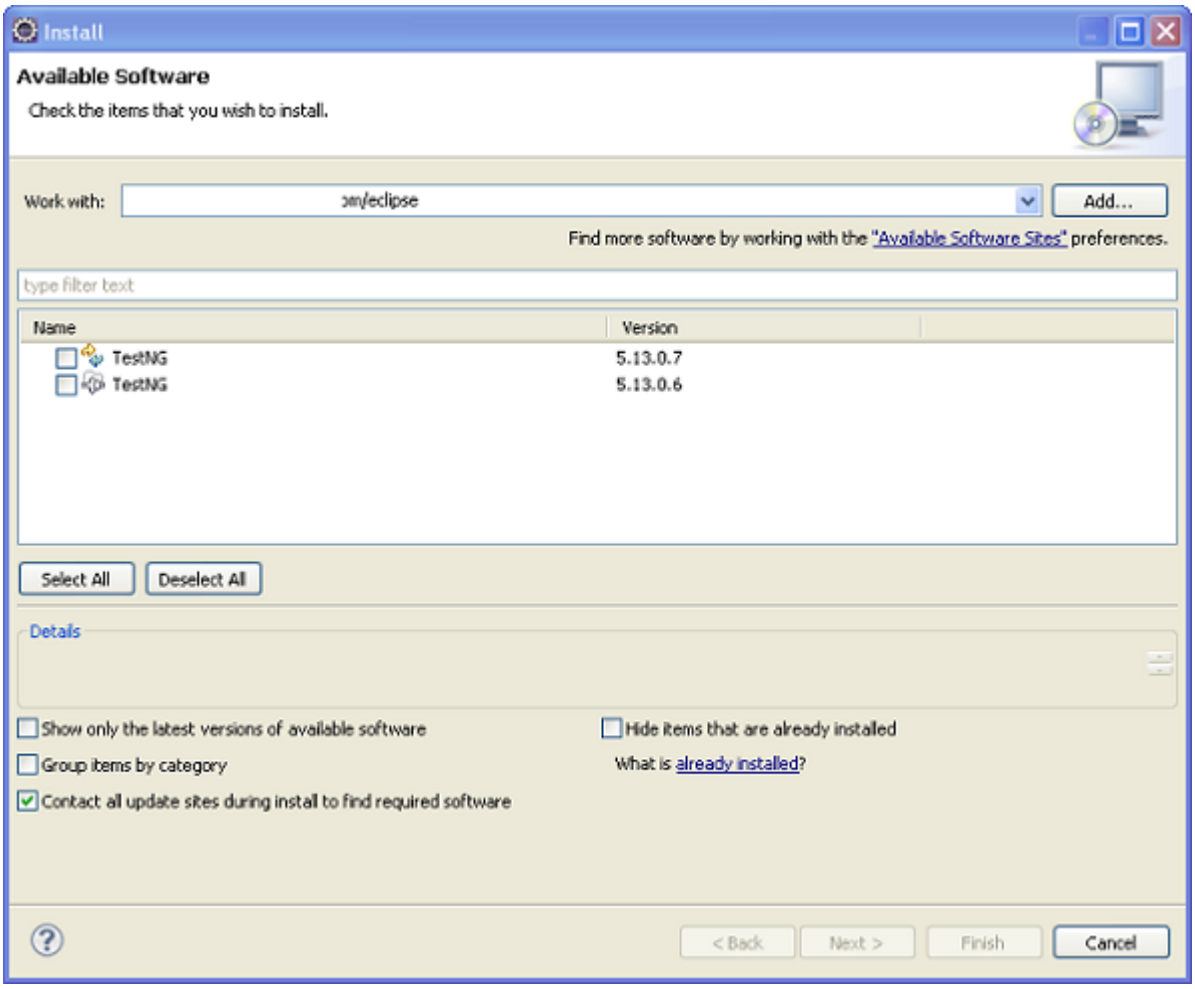
Selenium2 的官网被墙了，所以，大家自行的翻墙去下载吧，或者找老师要也可以

在 eclipse 中安装 testng，testng 我们后面会用到，所以在这里先安装：

1.Eclipse 中点击 Help->Install new software

2.点击 Add 在 name 中输入 testNg,在 Location 输入 http://beust.

com/eclipse



4. 选中 Testng 版本，点击 Next，按照提示安装，安装的过程中会有一次警告，直接点是即可，安装完之后重启 Eclipse

Selenium2 的 jar 包安装，在 eclipse 中新建工程后，在工程名上右键->build path->configure build path->libraries->add jars/add external jars，然后选择 jar 包即可。

Firefox 及 firebug 的介绍

Firefox 最好选择 32 版本的，或者延长版本(esr 版本)，安装好 firefox 后，第一件事就是把自动更新给关闭掉，刚说了，jar 包与浏览器版本要适配。

Firebug 是一个 firefox 的插件，打开 firefox,工具->附加组件->在搜索栏中输入 firebug,

用同样的方法安装上 firepath,在安装时，如果提示安装失败，可以 HOST 中加上：

117.18.237.29 ocsp.digicert.com

第二周：SELENIUM2 启动浏览器

启动主流浏览器:firefox, chrome, IE

System.setProperty(“webdriver.chrome.driver”, "files/chromedriver.exe");
如果不想这样，也可以讲 chromedriver.exe 放在” C:\Windows\System32”路径下即可
WebDriverdriver = new ChromeDriver();
System.setProperty(“webdriver.firefox.bin”, “D:/Program Files/Mozilla firefox/firefox.exe”);
driver = new FirefoxDriver();
所有 Selenium 版本的都有对 xp 下的 IE6,7,8 和 windows7 下的 IE9 支持。
System.setProperty("webdriver.ie.driver", "files/IEDriverServer64.exe");
WebDriver driver = new InternetExplorerDriver ();

SELENIUM2 如何加载 profile 完成对浏览器的插件定制

```
Firefox:
File file = new File(".\\res\\firebug-1.9.1-fx.xpi");
 FirefoxProfile firefoxProfile = new FirefoxProfile();
 try {
  firefoxProfile.addExtension(file);
 } catch (IOException e) {
  // TODO Auto-generated catch block
  e.printStackTrace();
 }
 firefoxProfile.setPreference("extensions.firebug.currentVersion","1.9.1");
Chrome:
ChromeOptions options = new ChromeOptions();
options.addExtensions(new File("files/Video-Sorter-for-YouTube_v1.1.2.crx"));
WebDriver driver = new ChromeDriver(options);
```

Firefox 的启动设置说明

```
FirefoxProfile profile = new FirefoxProfile();
profile.setPreference("browser.download.downloadDir", "c:\\data");
profile.setPreference("browser.download.folderList", 2); //browser.download.folderList 设置
Firefox 的默认 下载 文件夹。0 是桌面；1 是“我的下载”；2 是自定义
profile.setPreference("browser.helperApps.neverAsk.saveToDisk", "application/octet-stream,
application/vnd.ms-excel, text/csv, application/zip");
// 使用代理
profile.setPreference(“network.proxy.type”, 1);
```

```
// http 协议代理配置
profile.setPreference(“network.proxy.http”, proxyIp);
profile.setPreference(“network.proxy.http_port”, proxyPort);
driver = new FirefoxDriver(profile);

启用默认情况下被 firefox 禁用的功能
以本地事件例，很简单直接设置为 true 就可以了。
FirefoxProfile profile = new FirefoxProfile();
profile.setEnableNativeEvents(true);
WebDriver driver = new FirefoxDriver(profile);

如果要启动本机器的 firefox 的配置则用：
ProfilesIni allProfiles = new ProfilesIni();
FirefoxProfile profile = allProfiles.getProfile("default");
WebDriver driver = new FirefoxDriver(profile);

如果要用其它机器的 firefox 的配置，则用：
将其它机器上的 Profiles 文件夹” C:\Users\zhangfei\Application Data\Mozilla\Firefox\Profiles”
给拷贝出来，
File profileDir = new File("Profiles/raphlnmz.default");
FirefoxProfile profile = new FirefoxProfile(profileDir);
WebDriver driver = new FirefoxDriver(profile);
```

Chrome 的启动设置说明

```
将 user data 文件夹” C:\Users\zhangfei\AppData\Local\Google\Chrome\User Data”拷贝出来
ChromeOptions options = new ChromeOptions();
options.addArguments("--test-type");
File file = new File("User Data");
options.addArguments("user-data-dir="+file.getAbsolutePath());
WebDriver driver = new ChromeDriver(options);
```

IE 的启动设置说明

```
DesiredCapabilities capabilities = DesiredCapabilities.internetExplorer();
capabilities.setCapability(InternetExplorerDriver.INTRODUCE_FLAKINESS_BY_IGNORING_SECURITY_D
OMAINS, true);
capabilities.setCapability("ignoreProtectedModeSettings",true);//IE 默认的就是开启保护模式，
要么手动的在浏览器的设置中去关闭保护模式，要么加上这一句，即可
driver = new InternetExplorerDriver(capabilities);
```

第三周：元素定位方法介绍

本节课已然开始正题，主要让大家了解自动化测试中元素定位的重要性，以及定位元素的几种方法，其中重点介绍 XPATH 的写法。包括控件定位及层定位，配合 firepath 正确理解定位原理。

Xpath 用法:

Xpath 常用的几个符号：

/ 表示绝对路径，绝对路径是指从根目录开始

// 表示相对路径

. 表示当前层

.. 表示上一层

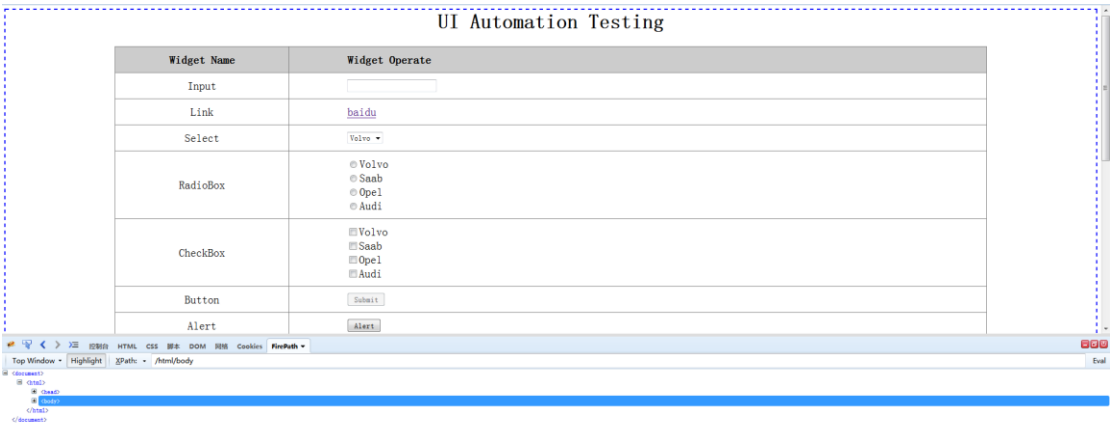
* 表示通配符

@ 表示属性

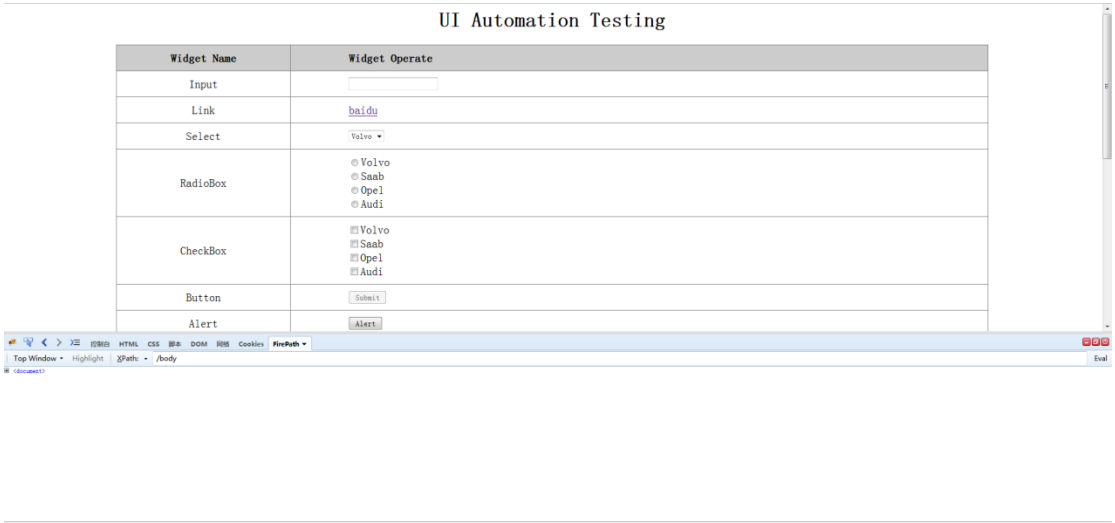
[] 属性的判断条件表达式

先看一个例子：

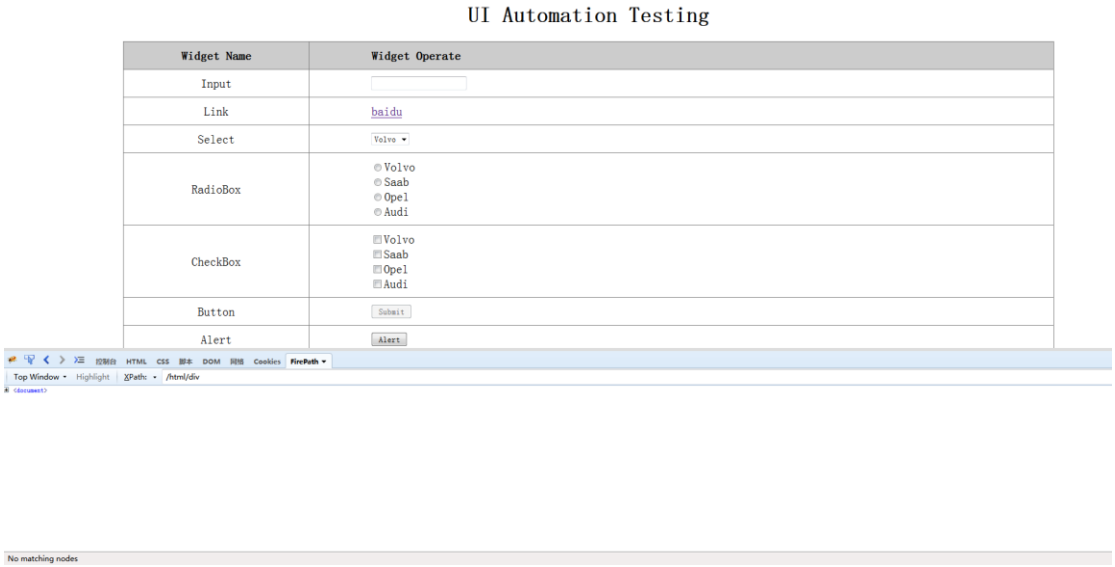
/html/body 见图 1，表示选择根目录下 html 下的 body



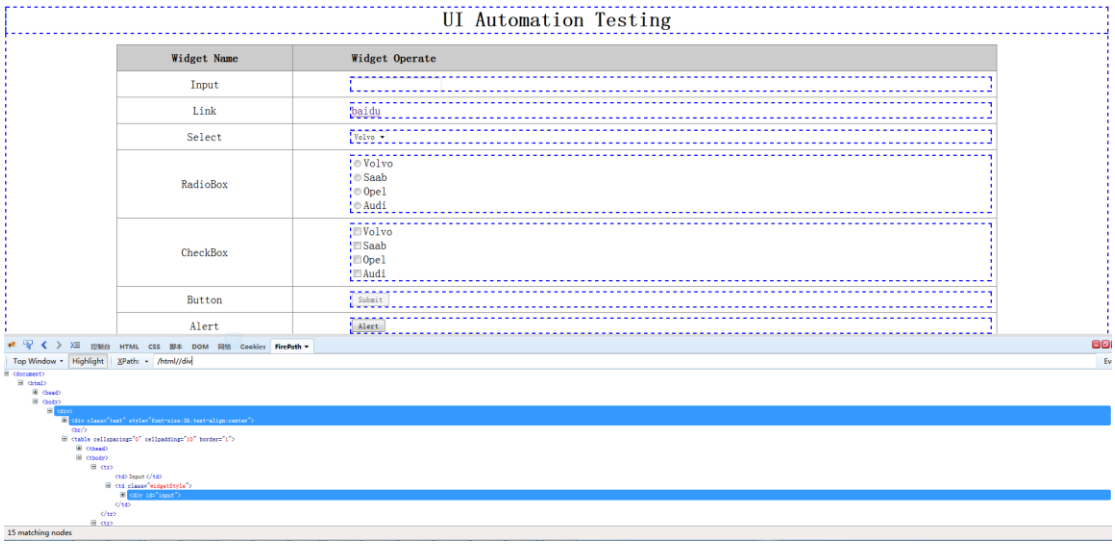
/body 见图 1.1，会看到没有节点可以被选择，因为/如果用在开头的话，表示根目录，而根目录是 html 节点



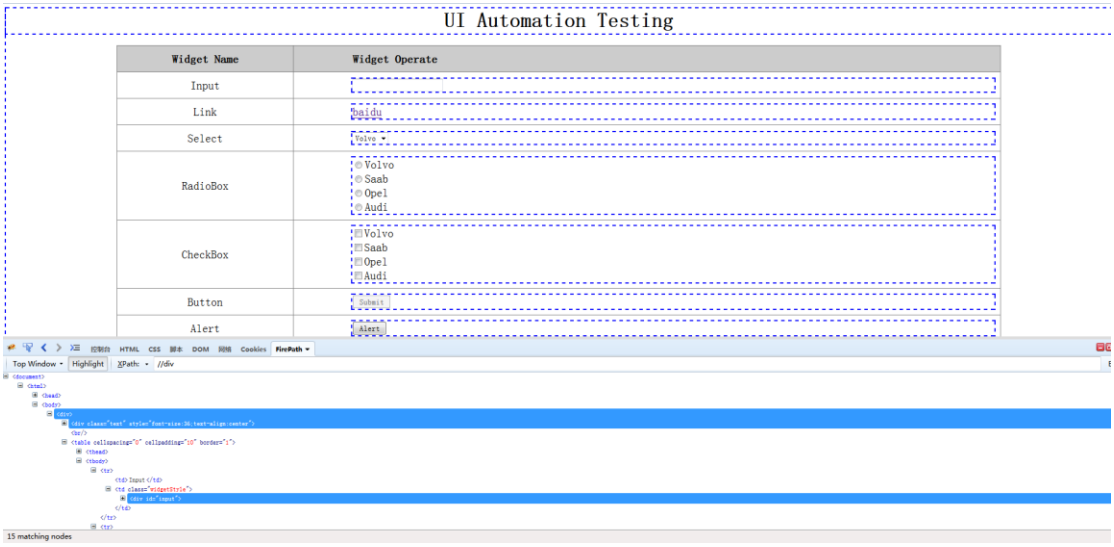
/html/div 见图 1.2，没有节点可以被选择，因为/如果用在中间，表示绝对路径，是上一个节点的子结点，而 html 的子节点是 head 与 body



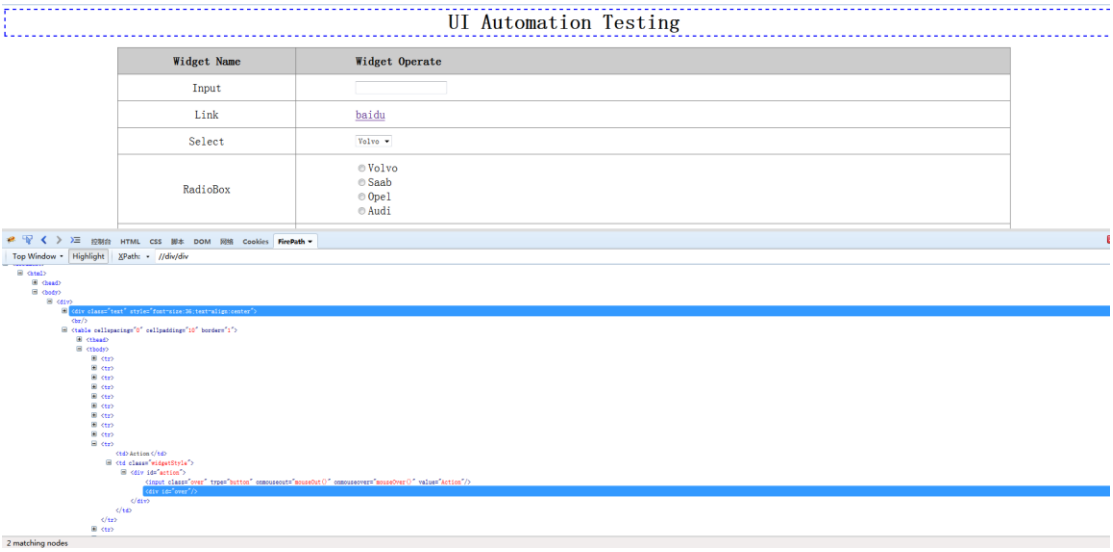
/html//div 见图 2，表示选择根目录下的所有的子孙后代节点中的 div 节点，//表示相对路径



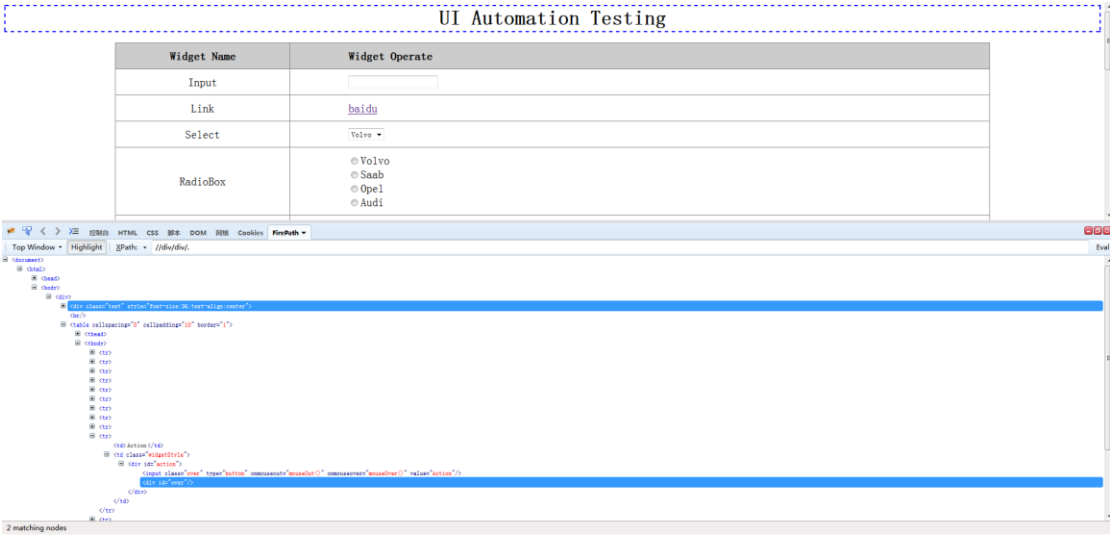
//div 见图 3，表示选择所有的 div 节点，可以想想图 2 与图 3 为什么结果是一样的！



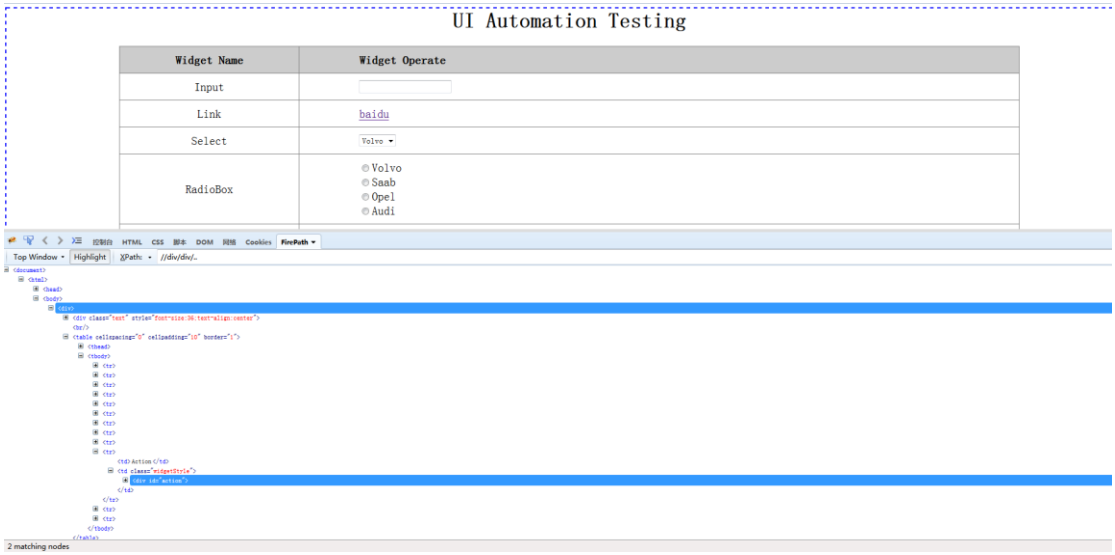
//div/div 见图 4，表示选择所有的 div 节点的子节点中含有 div 的节点



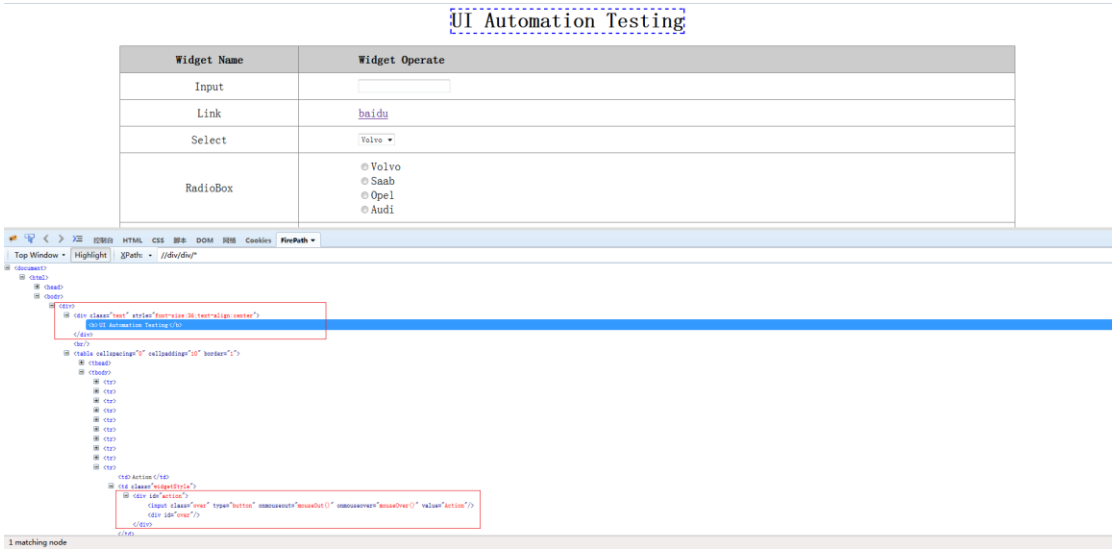
//div/div/. 见图 5，表示选择//div/div 节点的当前层的节点，与//div/div 的结果相同



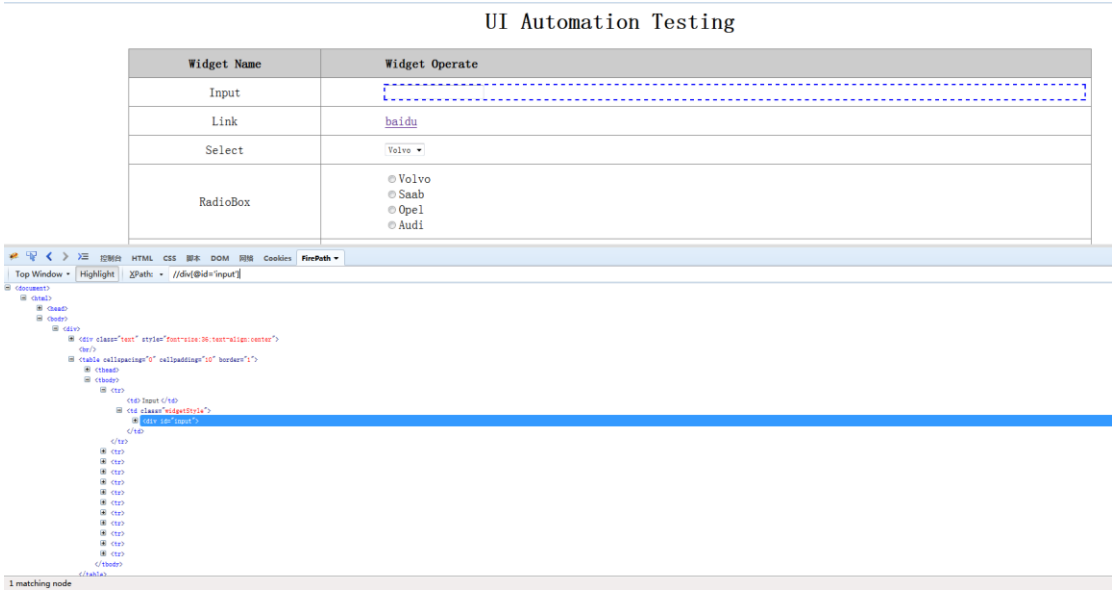
//div/div/.. 见图 6，表示选择//div/div 节点的上一层节点，也就是选择一个 div 节点，该 div 节点的子节点有 div 节点。有点绕口，但细细理解，会恍然大悟的



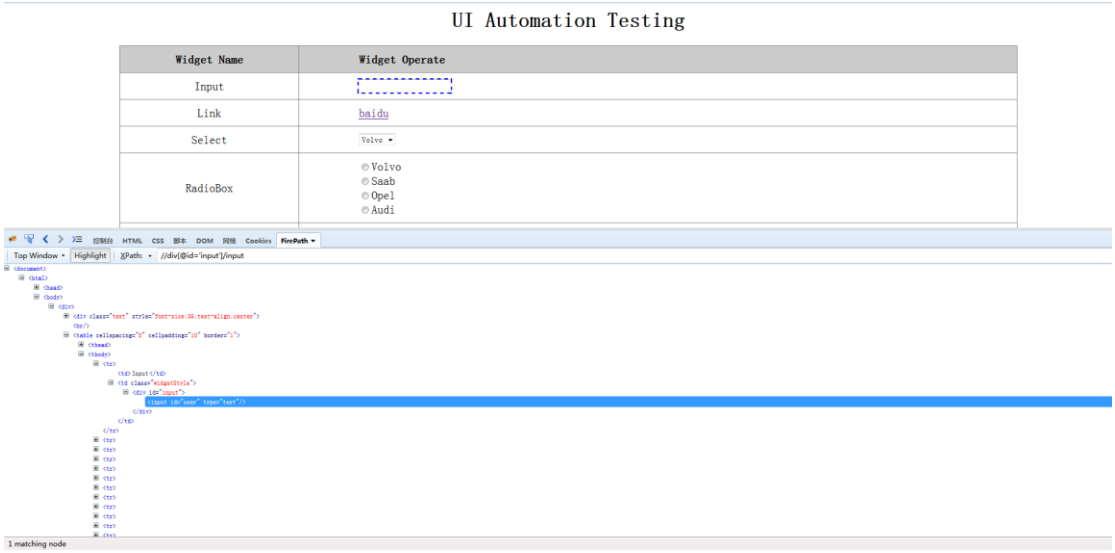
//div/div/* 见图 7，表示选择//div/div 的所有子节点，//div/div 会有两个匹配出来的节点，但为什么//div/div/*只有一个了呢？这是因为第二个//div/div 下面没有子节点了，所以只匹配出来了一个



//div[@id='input'] 见图 8，表示选择一个 id 为' input' 的 div 节点



//div[@id='input']/input 见图 9，表示选择一个 id 为’ input’ 的 div 节点的 input 子节点



//table//input[@id='user'] 见图 10，表示选择 table 的子孙后代中 id 为 user 的 input 节点

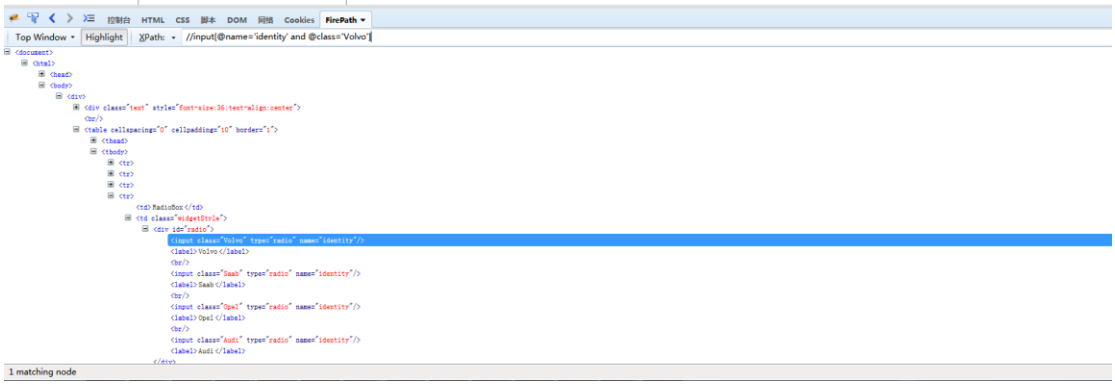
UI Automation Testing

Widget Name	Widget Operate
Input	
Link	baidu
Select	Volvo
RadioBox	Volvo Saab Opel Audi

//input[@name='identity' and @class='Volvo'] 见图 11，有的节点，只用一个属性无法定位出来，必须要用到多个属性进行组合定位，用连接符 and。这个 XPATH 表示选择一个 name 为 identity 并且 class 为 Volvo 的 input 节点

UI Automation Testing

Widget Name	Widget Operate
Input	
Link	baidu
Select	Volvo
RadioBox	Volvo Saab Opel Audi



//input[@name='identity' or @class='Volvo'] 见图 12,这个多属性组合用的是 or 的连接符,这个 XPATH 表示选择一个 name 为 identity，或者 class 为 Volvo 的节点，所以，这个 XPATH 匹配出来了 4 个节点

Xpath 的 index 选择

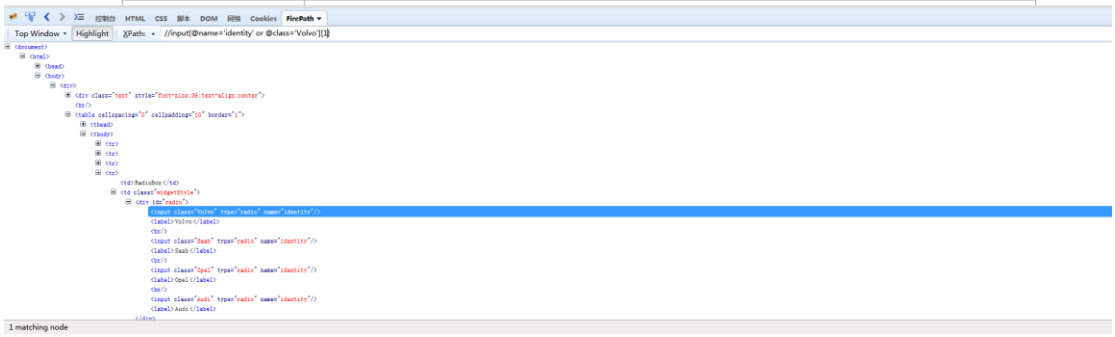
UI Automation Testing

Widget Name	Widget Operate
Input	
Link	baidu
Select	Volvo
RadioBox	Volvo Saab Opel Audi

//input[@name='identity' or @class='Volvo'][1] 见图 13，我们刚知道了，//input[@name='identity' or @class='Volvo']匹配出 4 个，我们只需要第一个，怎么办？加 index 即可：//input[@name='identity' or @class='Volvo'][1]，请注意，xpath 的 index 是以 1 开头的，并不是 0，请切记！

UI Automation Testing

Widget Name	Widget Operate
Input	
Link	baidu
Select	Volvo
RadioBox	Volvo Saab Opel Audi



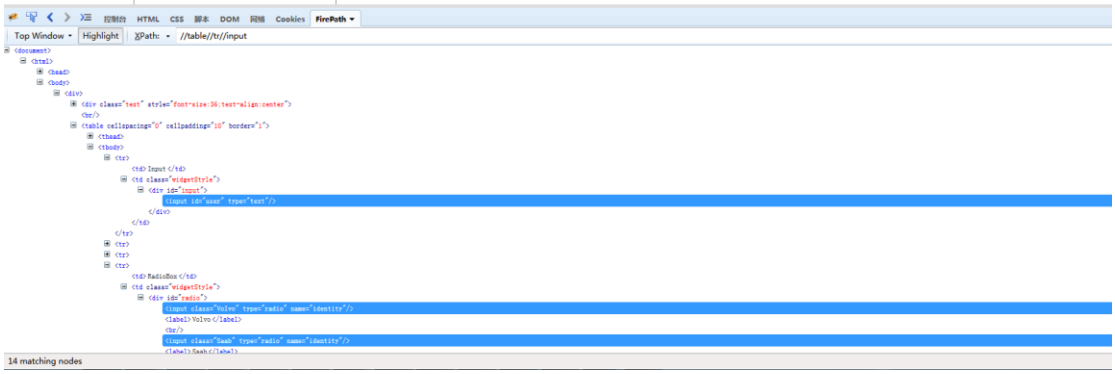
同理，取最后一个，//input[@name='identity' or @class='Volvo'][last()]，这里用到了 last()这个函数，后面我们会介绍几个常用的函数

需要特别注意的一个地方：

//table//tr//input 见图 14，这个匹配出来的，有 14 个节点，但是如果我们需要取到第一个，怎么办？

UI Automation Testing

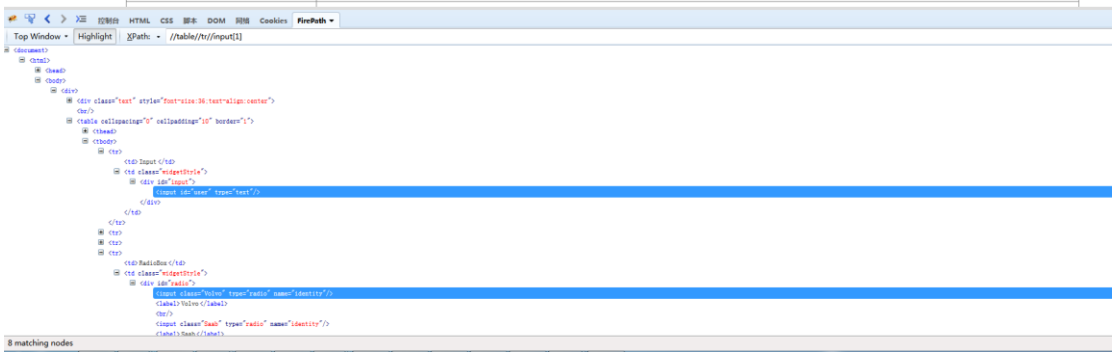
Widget Name	Widget Operate
Input	
Link	baidu
Select	Volvo
RadioBox	Volvo Saab Opel Audi



有可能会用到：//table//tr//input[1],但是我们来看看结果，见图 15，

UI Automation Testing

Widget Name	Widget Operate
Input	
Link	baidu
Select	Volvo
RadioBox	Volvo Saab Opel Audi

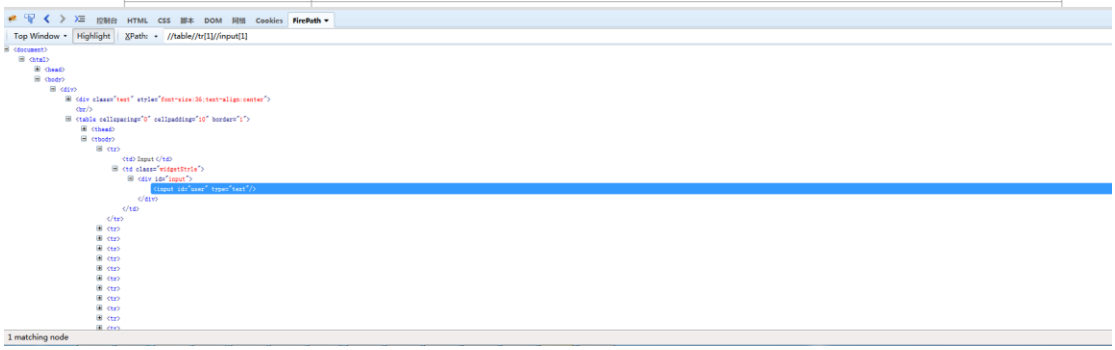


匹配出来的节点居然是 8 个，而不是 1 个，这是因为//table//tr//input[1]是指先匹配出//table 下面的所有的 tr 子孙后代节点，并且再此基础上，再匹配出 tr 节点的所有的子孙后代中的 input 结点的第一个，由于 tr 众多，所以匹配出的结果肯定不是一个，但如何能匹配出 1 个？也就是说我们需要把众多的 tr 给固定出一个，这时候再看：

//table//tr[1]//input[1]，见图 16，

UI Automation Testing

Widget Name	Widget Operate
Input	
Link	baidu
Select	Volvo
RadioBox	Volvo Saab Opel Audi



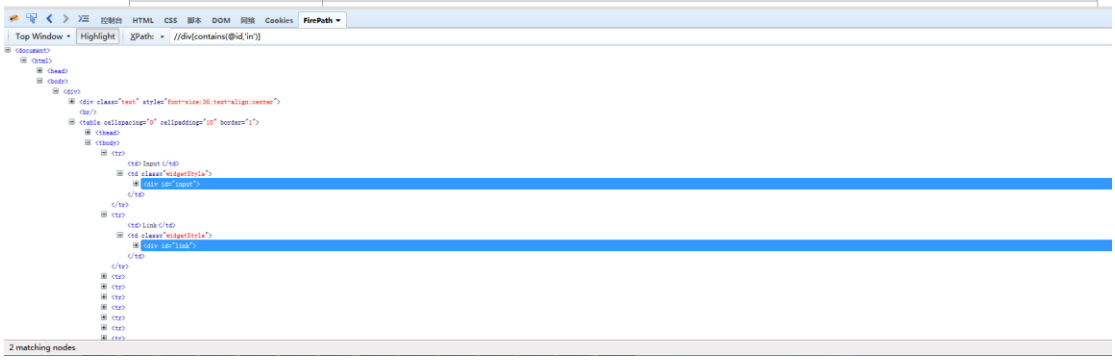
这时候就只有一个匹配出来的节点，所以，请大家仔细揣摩这里面的区别，细细体会

几个常用函数：contains(),text(),last(),starts-with(),not()

//div[contains(@id,'in')] 见图 17，

UI Automation Testing

Widget Name	Widget Operate
Input	
Link	baidu
Select	Volvo
RadioBox	Volvo Saab Opel Audi



表示选择 id 中包含有’ in’ 的 div 节点

由于一个节点的文本值不属于属性，比如

“baidu”,所以，用 text()函数来匹配节点：

//a[text()='baidu'] 见图 18，

UI Automation Testing

Widget Name	Widget Operate
Input	
Link	baidu
Select	Volvo
RadioBox	Volvo Saab Opel Audi

last()函数用法上面已介绍

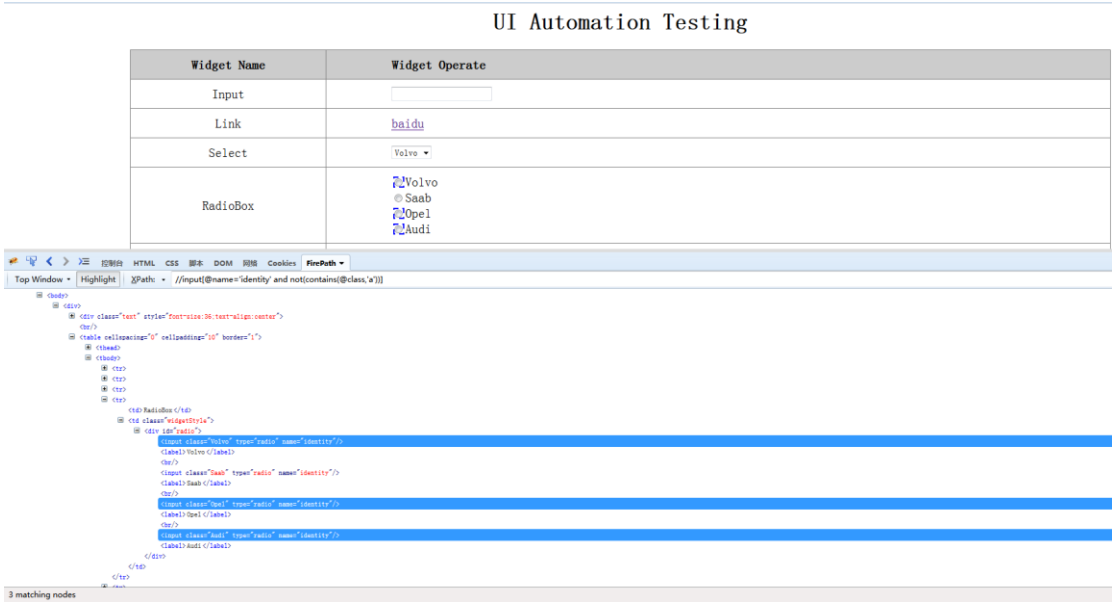
//div[starts-with(@id,'in')] 见图 19，表示选择以’ in’ 开头的 id 属性的 div 节点

UI Automation Testing

Widget Name	Widget Operate
Input	
Link	baidu
Select	Volvo
RadioBox	Volvo Saab Opel Audi

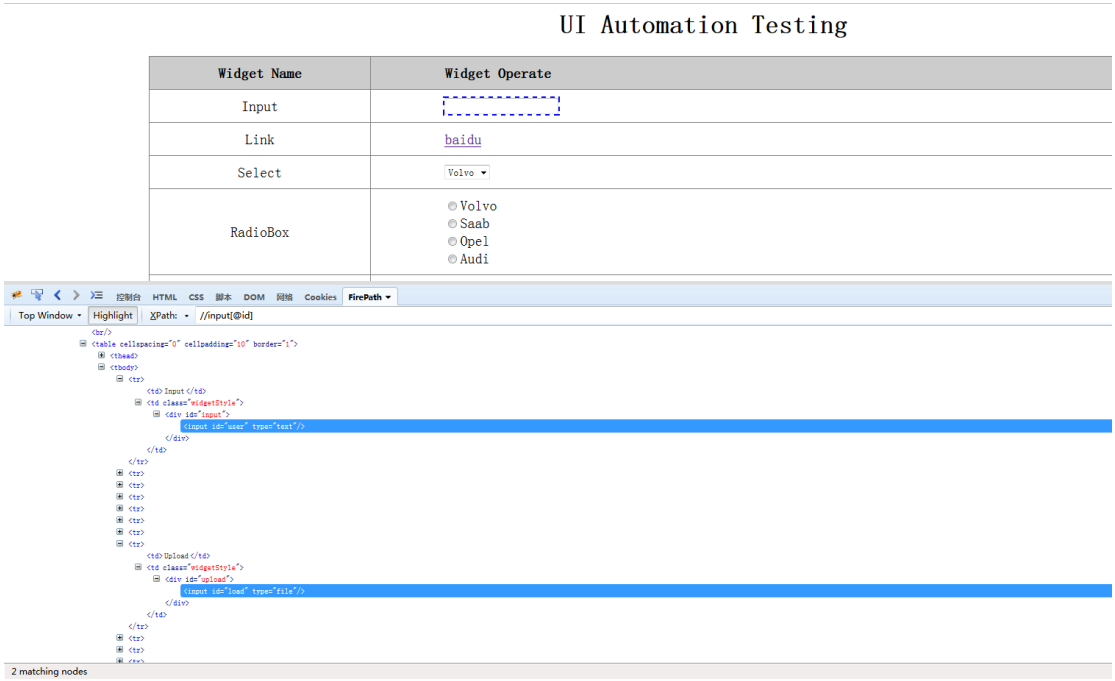
ends-with()函数在 XPATH1.0 中没有，2.0 中有，但是现在浏览器实现的都是 1.0，所以，暂时不建议使用 ends-with()函数

not()函数，表示否定，//input[@name=’identity’ and not(contains(@class,’a’))] 如图 20，表示匹配出 name 为 identity 并且 class 的值中不包含 a 的 input 节点

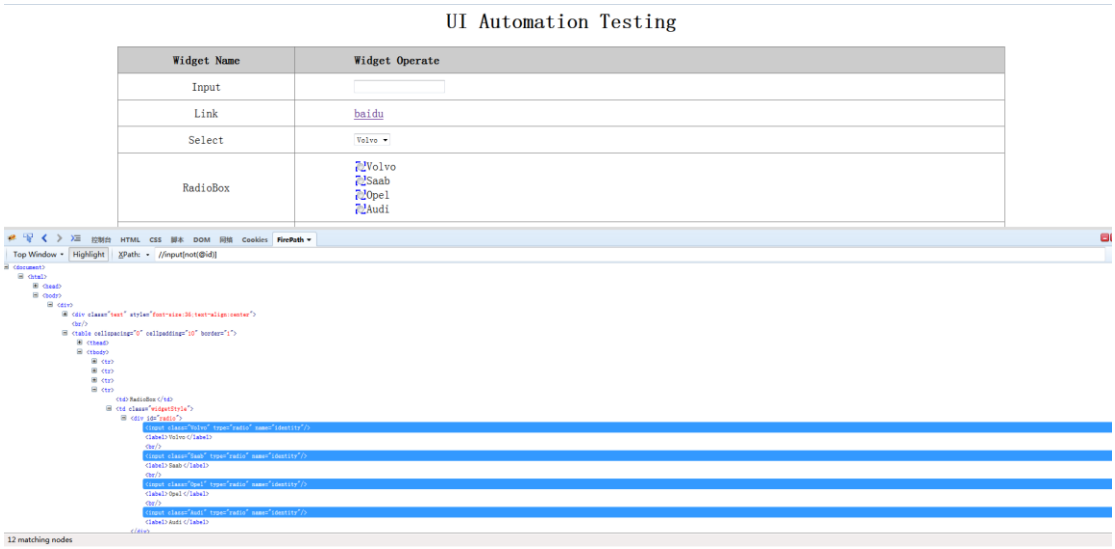


所以，not()函数通常与返回值为 true or false 的函数组合起来用，比如 contains(),starts-with()等，但有一种特别情况请注意一下：

我们要匹配出 input 节点含有 id 属性的，写法如下：//input[@id] 见图 21，



如果我们要匹配出 input 节点不含用 id 属性的，则为：//input[not(@id)] 见图 22



Xpath 轴的概念

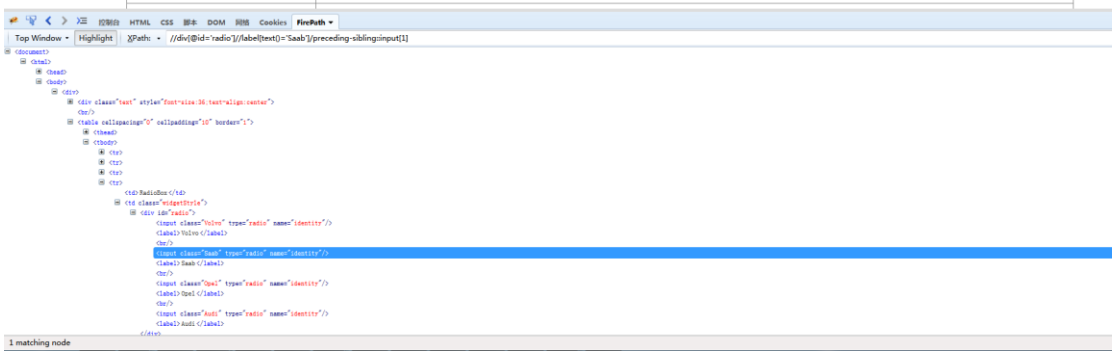
ancestor	选取当前节点的所有先辈（父、祖父等）。
ancestor-or-self	选取当前节点的所有先辈（父、祖父等）以及当前节点本身。
attribute	选取当前节点的所有属性。
child	选取当前节点的所有子元素。
descendant	选取当前节点的所有后代元素（子、孙等）。
descendant-or-self	选取当前节点的所有后代元素（子、孙等）以及当前节点本身。
following	选取文档中当前节点的结束标签之后的所有节点。
following-sibling	选取当前节点之后的所有同级节点。
namespace	选取当前节点的所有命名空间节点。
parent	选取当前节点的父节点。
preceding	选取文档中当前节点的开始标签之前的所有节点。
preceding-sibling	选取当前节点之前的所有同级节点。
self	选取当前节点。

轴的用法：

//div[@id='radio']//label[text()='Saab']/preceding-sibling::input[1] 见图 23，

UI Automation Testing

Widget Name	Widget Operate
Input	
Link	baidu
Select	Volvo
RadioBox	Volvo Saab Opel Audi



这个是选择 label 的 text 为 Saab 的节点之前的同级节点中为 input 节点的第一个，有点绕口，看实例更易明白，从中可以看出，我们可以根据一个 radiobox 的 label 来匹配出这个 radiobox，用 XPATH 轴时应该注意的几个问题：

1. 调用轴时，最好用’/’
2. 轴后面要加上符号”::”
3. “::”后面可以接节点名称，也可以接”*”

轴的另外一种写法：

//input[following-sibling::label[1][text()='Saab']]

这个的作用与//div[@id='radio']//label[text()='Saab']/preceding-sibling::input[1]的作用是一样的！

CSS 选择器(通常在 selenium2 中不赞成用 css selector,因为会有兼容性的问题，因为标准的 css 选择器支持的东西很少，我下面介绍的都是标准的 css 选择器所支持的，不在其列的，可能就不支持了！)

常见符号：

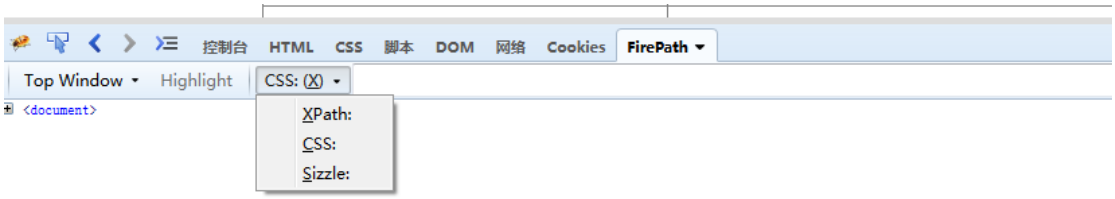
#表示 id

.表示 class

>表示子元素，层级

一个空格也表示子元素，但是是所有的后代子元素，相当于 xpath 中的相对路径

Firepath 中也可以检测 CSS，如图 24



可演示：

#input 选择 id 为 input 的节点

.Volvo 选择 class 为 Volvo 的节点

div#radio>input 选择 id 为 radio 的 div 下的所有的 input 节点

div#radio input 选择 id 为 radio 的 div 下的所有的子孙后代 input 节点

div#radio>input:nth-of-type(4) 选择 id 为 radio 的 div 下的第 4 个 input 节点
div#radio>:nth-child(1) 选择 id 为 radio 的 div 下的第 1 个子节点
div#radio>input:nth-of-type(4)+label 选择 id 为 radio 的 div 下的第 4 个 input 节点之后挨着的 label 节点
div#radio>input:nth-of-type(4)~label 选择 id 为 radio 的 div 下的第 4 个 input 节点之后的所有 label 节点
input.Volvo[name='identity'] 选择 class 为.Volvo 并且 name 为 identity 的 input 节点
input[name='identity'][type='radio']:nth-of-type(1) 选择 name 为 identity 且 type 为 radio 的第 1 个 input 节点
input[name^='ident'] 选择以 ident 开头的 name 属性的所有 input 节点
input[name$='entity'] 选择以'entity'结尾的 name 属性的所有 input 节点
input[name*='enti'] 选择包含'enti'的 name 属性的所有 input 节点
div#radio>*:not(input) 选择 id 为 radio 的 div 的子节点中不为 input 的所有子节点
input:not([type=radio]) 选择 input 节点中 type 不为 radio 的所有节点

第四周：SELENIUM2 基础 API 介绍

SELENIUM2 对基础控件的操作
SELENIUM2 对常用 API 的调用演示

1. 新建测试类及启动浏览器

```
public class Test3 {

    public WebDriver driver;

    public Test3(){
        driver = new FirefoxDriver();
        driver.manage().window().maximize();
    }

    public void close(){
        driver.close();
        driver.quit();
    }

    public static void main(String[] args) {
        Test3 t = new Test3();
        t.close();
    }
}
```

Test3 类中的 Test3 方法是一个构造器，JAVA 里的构造器会在对象被 new 时就自动的调用。在构造器里，产生了一个 driver 的实例化的对象，driver 的实例化对象产生后，就会打开浏览器，

driver.manage().window().maximize();这一句是指把打开的浏览器进行最大化。

Close 方法是关闭，其中 driver.close()是指关闭打开的浏览器，driver.quit()是指退出 driver,这两者的区别，在后面的实例中，会给大家进行讲解。

上面的代码中，启动了浏览器了，接下来，就可以进行一些我们想要进行的操作了，

如何进入到我们指定的 URL?我们再来加一个方法：

```
    public void goTo(){
//      driver.get("http://www.baidu.com");
        driver.navigate().to("http://localhost:8080/demo.html");
    }
```

调用：

```
    public static void main(String[] args) {
        Test3 t = new Test3();
        t.goTo();
        t.close();
    }
```

从上面的 goTo()方法中，首先，方法名，T 大写了，在 JAVA 中，方法名通常是以驼峰式来命名方法名的，这是一些规范，大家最好还是遵守一下，goTo（）方法里，我注释掉了第一句，其实，进入到指定的 URL，这两句都可以达到相同的目的，记住一种方法即可，"http://localhost:8080/demo.html"是参数，是指我们要打开的 URL 链接，main()方法是指入口方法，在里面可以进行类的实始化，方法的调用，t.goTo()是对在类中加的方法进行了调用。

在打开浏览器后,我们来对一些常见的控件进行操作,主要包括 input(输入框) a select radiobox checkbox button

先来看看 input,input 在 html 中一般为：<input id="user" type="text">，其中 type=" text" 表示其为一个输入框，对 input 输入框的操作一般有:输入/清除输入值/取得输入值

```
    public void testInput(){
        driver.findElement(By.id("user")).sendKeys("test");
    }
```

Selenium2 中，需要先定位到一个元素，再进行操作，定位的方法我们已经讲过了，现在的问题是如何将定位的方法用上？用 driver.findElement(By.id("user")),findElement 方法的返回值是一个 WebElement 对象，findElement 里的参数是一个 by 对象，by 对象的写法有如下几种：

```
        By.id("") //根据id来生成一个by对象
        By.className("")//根据class属性来生成一个by对象
        By.linkText("")//根据链接的字符串来生成一个by对象
        By.name("")//根据id来生成一个by对象
        By.xpath("")//根据xpath来生成一个by对象
        By.cssSelector("")//根据css selector来生成一个by对象
```

上面的代码中可以改为:

```
    public void testInput(){
//      driver.findElement(By.id("user")).sendKeys("test");
        WebElement element = driver.findElement(By.id("user"));
        element.sendKeys("test");
    }
```

以上是对输入框的输入，用 sendKeys 方法，参数为要输入的值" test"，

清除输入值:

```
    public void testInput(){
//      driver.findElement(By.id("user")).sendKeys("test");
        WebElement element = driver.findElement(By.id("user"));
        element.sendKeys("test");
        element.clear();
    }
```

clear 是对刚才输入的”test”进行了清空。

取得输入值，这个比较特殊，我们在 a 标签操作时讲。

a 标签的操作

a 标签在 html 中一般为: baidu,也就是一个超链接，对 a 标签，也就是链接的操作一般是点击：

```
    public void testLink(){
        WebElement element =
driver.findElement(By.xpath("//div[@id='link']/a"));
        element.click();
    }
```

点击以后，到了一个新的页面，这时候，我们想要回到原来的页面，用如下的：

driver.navigate().back();

我们再看刚才的超链接,有属性 class,还有 href,如果我们想要得到这些属性对应的值如何办？selenium2 提供了一个 API：getAttribute，用法如下：

```
        String href = element.getAttribute("href");//取得属性值
        System.out.println(href);//输出属性值
```

同理：

```
        String className = element.getAttribute("class");//取得属性值
        System.out.println(className);//输出属性值
```

上面的 a 标签中，如果我们想要取得”baidu”这个文本值如何办？（在 HTML 中，两个节点间的字符串称为该节点的文本值，比如上面的 a 标签的”baidu”）,用如下的 api：getText(),用法如下

```
          String text = element.getText();
          System.out.println(text);
```

针对上面的 input 输入框，我们如果想要得到其输入框里的值，如果我们用 getText(),试着输出一下，发现其为空，并没有输出”test”,这是为什么呢？因为在 input 中，其输入的值，是一个属性，而不是 text,输入”test”后，其 input 节点此时应该为：<input id="user" type="text" value="test">,所以，这时候我们应该用 getAttribute,总结一下这两个方法：

```
    public void testInput(){
//      driver.findElement(By.id("user")).sendKeys("test");
        WebElement element = driver.findElement(By.id("user"));
        element.sendKeys("test");
        element.clear();
        element.sendKeys("test");
        String text = element.getAttribute("value");
        System.out.println(text);
    }

    public void testLink(){
```

```
        WebElement element =
driver.findElement(By.xpath("//div[@id='link']/a"));
        element.click();
        driver.navigate().back();
        String href = element.getAttribute("href");
        System.out.println(href);
        String className = element.getAttribute("class");
        System.out.println(className);
        String text = element.getText();
        System.out.println(text);
    }
```

Select 控件操作，select 控件，也就是下拉选择，在 HTML 中一般为：

```
<select  name="select">
    <option  value="volvo">Volvo</option>
    <option  value="saab">Saab</option>
    <option  value="opel">Opel</option>
    <option  value="audi">Audi</option>
</select>
```

option 为可以选择的值。先来看下面这段代码：

```
    public void testSelect() {
        WebElement element =
driver.findElement(By.cssSelector("select[name='select']"));
        Select select = new Select(element);
        select.selectByValue("opel");
        String text = select.getFirstSelectedOption().getText();
        System.out.println(text);
    }
```

上面的代码中，可以看到有一个 Select 的对象被 new 出来了，这点要特别注意，也就是说在 SELENIUM2 中针对 select 控件，有一个专门的类 Select,在 new 这个对象时，参数为定位出来的元素对象 WebElement,然后就可以 selectByValue，这个 value 也就是 option 中的 value 属性的值，当然还支持其它的 select 方式：

```
        select.selectByIndex(2);
        select.selectByVisibleText("Opel");
```

这两个产生的效果与 select.selectByValue("opel");是一样的，这两者希望同学们自己去下实验一下并细细的体会一下。getFirstSelectedOption 这个方法是得到被选中的选项对象，结合 getText，就是指得到被选中的选项的 text 值，是 text 值，而不是 value 值。

Radiobox 操作：

Radiobox 在 HTML 中一般为：<input class="Volvo" type="radio" name="identity">，但 radiobox 一般为一个 group,并且其中一个选中后，其它的就为非选中的状态了，所以，这个 group 一般为:

```
<input  class="Volvo"  type="radio"  name="identity">
<label>Volvo</label>
<br>
<input  class="Saab"  type="radio"  name="identity">
```

<label>Saab</label>

<input class="Opel" type="radio" name="identity">
<label>Opel</label>

<input class="Audi" type="radio" name="identity">
<label>Audi</label>

从上面的例子中，可以看出，是以 name 来判断是否是同一个 radiobox group 的，同一个 group 的 name 值都是一样的，radiobox 的控件标签也是 input,但其 type 为 radio。我们来看对其的操作：

```
    public void testRadioBox() {
        List<WebElement> elements = driver.findElements(By.name("identity"));
        elements.get(2).click();
        boolean select = elements.get(2).isSelected();
        System.out.println(select);
    }
```

上面的例子中，我们看到了一个新的方法：findElements，这个方法，表示返回的是一个 WebElement 集合，由于 radio box 是以 name 相同的一个 group,所以，我们要把 name 相同的全部找出来，然后再来操作，于是就用了 findElements 方法，找出集合后，就是把集合里的 WebElement 进行点击的操作，click 后，就表示被选中了，isSelected 是用来判断当前 radiobox 对象是否被选中

Checkbox 操作：

Checkbox 在 HTML 中一般为：

<input type="checkbox" name="checkbox1">
<label>Volvo</label>

<input type="checkbox" name="checkbox2">
<label>Saab</label>

<input type="checkbox" name="checkbox3">
<label>Opel</label>

<input type="checkbox" name="checkbox4">
<label>Audi</label>

Checkbox 控件标签也是 input,其 type 为 checkbox,这都是固定定法，我们来看看对其的操作：

```
    public void testCheckBox() {
        List<WebElement> elements = driver.findElements(By
                .xpath("//div[@id='checkbox']/input"));
        WebElement element = elements.get(2);
        element.click();
        boolean check = element.isSelected();
        System.out.println(check);
    }
```

上面的代码中，也是用 XPATH 先定位出所有的符合条件的 checkbox，但其实也可以

WebElement element = driver.findElements(By.xpath("//div[@id='checkbox']/input[2]"));

同学们可以自己试验一下，定位出 checkbox 对象后，click 就表示对其进行了操作，我们可以看到 click

后，有可能是选中，也有可能是反选了，于是，用 isSelected 来判断其是否进行了选中并输出其结果。

Button 控件的操作：

Button 控件在 HTML 中，一般有两种表现形式：

<input class="button" type="button" disabled="disabled" value="Submit">

<button class="button" disabled="disabled" > Submit</button>

如果是 input 标签时，其 type 一定是为 button,两种写法在操作时，都是一样的，看如下的代码：

```
    public void testButton() {
        WebElement element = driver.findElement(By.className("button"));
        element.click();
        boolean button = element.isEnabled();
        System.out.println(button);
    }
```

Button 一般为点击，用 click 方法点击，有的 button 是灰掉的，也就是 disabled 的，所以，用 isEnabled 去判断该 button 是否可用及输出其结果。

现在我们来把上面的代码都来总结一遍：（详情请见 com.demo.Test3）

第五周：SELENIUM2 常用类介绍

Alert 类介绍

Alert 类，是指 windows 弹窗的一些操作，在 selenium1 及一些其它的自动化测试的工具中，对 windows 弹窗也是没有很完美的解决方案，有的或许要借助一些其它的工具，如 autoit 等，效果都不是很理想，在 selenium2 中，加入了 Alert 类，专门用来处理 windows 弹窗问题，我们先来看一个 Alert 弹窗在 HTML 中的写法，一般为：

<input class="alert" type="button" onclick="display_alert()" value="Alert">

即点击这个 button,触发一个 display_alert 的 JS 函数，会弹出一个 windows 的弹出框。对于 Alert 类的用法，我们看如下代码：

```
    public void testAlert() {
        WebElement element = driver.findElement(By.className("alert"));
        Actions action = new Actions(driver);
        action.click(element).perform();
        Alert alert = driver.switchTo().alert();
        String text = alert.getText();
        System.out.println(text);
        alert.accept();
     }
```

上面的代码中，先定位出要触发 alert 的 button,因为这个 button 要在 click 时，执行一个 display_alert 函数，所以要用到 Action 类，Action 类的具体用法，下面再讲。点击这个 button 后，弹窗出现，注意，这时候，我们要把 driver 给切换到 alert 弹窗上去，切换上去后，就产生了一个 Alert 对象，也就是这一句：Alert alert = driver.switchTo().alert(); alert 对象产生后，就可以用这个对象来做一些操作了，

alert.getText();是指得到弹窗上面的字符串，弹窗出现后，有的上面有两个按钮，一个”是”，一个”否”，有的只有一个按钮，一个“确定”，alert.accept();是指点击弹窗上面的”是”或者”确定”，alert.dismiss();是指点击弹窗上面的”否”或者“取消”之类的。所以我们简单的记住 alert 类的使用:driver 切换到弹窗上面，产生一个 alert 对象，然后就可以对 alert 进行取字符串或者点击相应按钮的操作了。但是需要强调的一点是，Alert 类也只能处理一些简单的 windows 弹窗，对于复杂的，比如上传文件的那种弹窗，可能就不适用了。

Action 类介绍

Action 类的使用，可能会很多，一般是在要触发一些 js 函数或者一些 JS 事件时，要用到这个类，比如上面的要触发一个 display_alert 的 JS 函数，就要用到 Action 类，否则，如果直接用 element.click();则是不会有弹框出现的，我们再来看下面的一段 HTML：

<input class="over" type="button" onmouseout="mouseOut()" onmouseover="mouseOver()" value="Action">

这段 HTML 是指当鼠标放在这个”Action”按钮上时，就会触发 mouseOver()函数，出现一段红色的”Hello World!”的字符串，当鼠标从这个按钮上移开时，就会触发 mouseOut()函数，这段红色的”Hello World!”字符串消失。我们来模拟一段鼠标移上移出的操作，就要用到 Action 类，看下面一段代码：

```
    public void testAction() {
        WebElement element = driver.findElement(By.className("over"));
        Actions action = new Actions(driver);
        action.moveToElement(element).perform();
        String text = driver.findElement(By.id("over")).getText();
        System.out.println(text);
    }
```

同样是首先要定位到要操作的元素对象上去，然后这时候就要new一个Action的对象了，new时要传一个driver参数过去，action对象产生之后，就可以操作了，鼠标移上去的操作，我们可以用action.moveToElement(element)，onclick事件时，要用到action.click(element)，不同的事件，所用到的方法不同，这是最常用的两个，当调用后，一定要加上perform方法，这样，这个动作才会完成，所以请大家务必记住这个写法：

```
Actions action = new Actions(driver);
action.moveToElement(element).perform();
```

当红色的字符串出现后，我们把这一段字符串取出来，并输出，就是后面那两句代码。

上传文件操作

上传文件，可能也是用的比较多的，先来看段上传文件的HTML代码：

<input id="load" type="file">

也是一个input标签，但是type是file，上传文件时，我们手动操作时，就是选择好一个文件，选择好后，在这个input框里，就会出现该文件的本地路径，这时候就能提交了，由于对于WINDOWS弹窗出现后，选择文件这个过程，用selenium2无法实现，只能用其它的外部工具，这里我们不介绍，我们只介绍一个最普遍被用到的方法，刚才说了，只要一个本地路径出现在input框里就可以了，所以，我们可以把本地路径通过sendKeys的方式给输入进去，看如下的代码：

```
    public void testUpload() {
```

```
        WebElement element = driver.findElement(By.id("load"));
        element.sendKeys("c://test.txt");
    }
```

以上代码就是实现文件上传的操作，但是有的上传文件的输入框被做了限制，不让输入，当出现这种情况时，有这能是这个输入框被设定成了readonly="readonly"

我们可以执行一段JS语句，把这个readonly的属性给去掉，让这个输入框能够被输入，或者让前端开发人员把readonly属性给去掉。至于这段JS语句如何写，大家可以自己去查阅一下，网上资料很多，如何执行JS呢，下一步会介绍到。

调用 JS 介绍

上面我们也介绍了，在自动化时，执行一段 JS，去改变某些元素对象的属性或者进行一些特殊的操作，问题来了，如何执行一段 JS？我们来看如下的代码：

```
    public void testJavaScript(){
        JavascriptExecutor j = (JavascriptExecutor)driver;
        j.executeScript("alert('hellow rold!')");
        Alert alert = driver.switchTo().alert();
        String text = alert.getText();
        System.out.println(text);
        alert.accept();
    }
```

我们首先要产生一个执行 JS 的对象，也就是 JavascriptExecutor 对象，这个对象是由 driver 进行强制类型转换得来的，或者说大家记住这种写法：JavascriptExecutor j = (JavascriptExecutor)driver;然后这个对象 j 就可以调用 executeScript 方法来执行一段 JS，这段 JS 的语句是以一段字符串的形式给传参到 executeScript 中去的，上面的"alert('hellow rold!')"是指弹出一个 hellow rold!的 windows 对话框，当这个 windows 对话框弹出后，就变成了用 Alert 类来进行操作了，这个上面已讲过了，请大家再次理解一下这一段代码。

Iframe 操作

Iframe 也是做自动化测试经常碰到的，iframe 里面嵌套的是一个新的 HTML 页面，我们来看 IFRAME 的 HTML：

<iframe width="800" height="330" frameborder="0" src="./demo1.html" name="aa">

这里面的 src 属性，就是新的 html 页面地址，由于 driver 只是对一个页面进行操作，如果不在该页面了，就不能操作该页面了，既然有嵌套，也就是对 iframe 里面的页面不能直接进行操作了，所以，就得想办法把 driver 给切到 iframe 里面的页面上去，我们来看如下的代码：

```
     public void testIframe(){
         driver.switchTo().frame("aa");
//       driver.switchTo().frame(0);
//       WebElement iframe =
driver.findElement(By.xpath("//iframe[@name='aa']"));
//       driver.switchTo().frame(iframe);
```

```
        driver.findElement(By.id("user")).sendKeys("test");
        driver.switchTo().defaultContent();
    }
```

上面的代码中，driver.switchTo().frame("aa");这一句是指切换到一个 aa 的 frame 里面去，这个”aa”的参数是如何确定的？如果一个页面中，有很多个 iframe,那如何切到我想要到达的 iframe 里面？主要有三种方式：

1. 如果 iframe 标签有能够唯一确定的 id 或者 name,就可以直接用 id 或者 name 的值：
   driver.switchTo().frame("aa");
2. 如果 iframe 标签没有 id 或者 name,但是我能够通过页面上确定其是第几个(也就是通过 index 来定位 iframe,index 是从 0 开始的)：
   driver.switchTo().frame(0);
3. 也可以通过 xpath 的方式来定位 iframe,写法如下：
   WebElement iframe = driver.findElement(By.xpath("//iframe[@name='aa']"));
   driver.switchTo().frame(iframe);

切进 iframe 后，操作就跟没有 iframe 的情况一样的了，比如
driver.findElement(By.id("user")).sendKeys("test");就是指在 iframe 里面的 id 为 user 的 input 框里输入 test.
操作完后，我们要切出来，这样才能再次操作最外面的 html 控件，切出来的方法是：
driver.switchTo().defaultContent();

多窗口操作

有时候当我们点击一个链接时，就会在一个新的窗口打开对应的页面，我们来看一下这种情况的 HTML：
Open new window
上面的 HTML 中，主要加了一个 target="_bank",这个属性的意思就是指在新窗口打开对应的页面。
刚才我们说了，driver 只能在当前页面或当前窗口上进行操作，打开新窗口后，driver 只能是切到这个新打开的窗口中后，才能对这个新的窗口进行操作。浏览器窗口都是有句柄的，我们可以根据句柄来决定需要把 driver 切到哪个对应的窗口上面去。看如下代码：

```
    public void testMultiWindow() {
        WebElement element = driver.findElement(By.className("open"));
        element.click();
        Set<String> handles = driver.getWindowHandles();
        String handle = driver.getWindowHandle();
        handles.remove(driver.getWindowHandle());
        driver.switchTo().window(handles.iterator().next());
        driver.close();
        driver.switchTo().window(handle);
    }
```

上面的代码中，点击链接后，就出现了一个新的窗口，这时候就有两个窗口了，我们要做的就是把两个窗口的句柄都确定下来，然后切换，
Set<String> handles = driver.getWindowHandles();getWindowHandles 这个方法返回的是一个所有句柄的集合 Set 对象，Set<String>里面的 String 是指这个集合里面的所有数据类型都是 String,
getWindowHandle 这个方法是得到当前 driver 所在的页面的句柄，Set 中有两个句柄，且知道了当前所在

的窗口的句柄，如果我从 Set 中把当前所在窗口的句柄给 remove 掉，那剩下的一个就是要切换到的窗口的句柄了，于是 handles.remove(driver.getWindowHandle());这个是 remove,那个 Set 中此时就只有一个句柄了，取得这个句柄，Set 的取值时，可以把 Set 对象转换成一个 iterator 对象，然后用 next 方法取得值，handles.iterator().next()，这个就是返回新窗口的句柄了，于是切过去就用这一句：
driver.switchTo().window(handles.iterator().next());至于 Set 循环取值的方法，请大家自己上网查一下资料，我这上面写的，已基本把使用方法告诉大家了。我们可以看到，切到新的页面后，就可以直接对新的页面进行操作了，操作后，如果想关掉这个页面，调用 driver.close();,这个时候就看出 close 与 quite 的区别了，close 是指关闭当前 driver 所在的窗口，但是 driver 还可以继续使用，quite 后，driver 就不能继续使用了。
上面的代码中，driver.close()后，这时候把新打开的页面关闭了，但是 driver 却不在剩下的那个页面中，于是，我们仍然需要切换到剩下的页面中去，由于我们已经把这剩下的页面的句柄的值给赋予到了 handle 变量，所以只需要用这句代码即可切换：driver.switchTo().window(handle);
多窗口的操作也很重要，请牢记上面的写法。

Wait 机制及实现

自动化测试时，有一个很大的缺点就是不稳定，运行时很不稳定，常常有一些 TIMEOUT 超时的异常抛出，为什么会超时呢，比如：
driver.findElement(By.id(“button1”)).click()这一句执行时，是点击 button1,但是接着自动化脚本会跟着执行 driver.findElement(By.id(“searchReult”)).getText();这一句是取得点击操作后，出现在页面上的一串字符串，但是如果点击操作后，由于网络很慢或者其它的一些什么原因，要取得的这一串字符串始终没有出现在页面上，那么执行 driver.findElement(By.id(“searchReult”)).getText();这一句就会报错了，那如何让其不报错，不超时呢，那么我们就需要在这两句代码中加一个 Thread.sleep(10000),10000 指 10 秒钟，也就是让自动化脚本在执行完点击操作后，等待 10 秒钟，然后在 10 秒钟内，这个字符串可能就出现在了页面上了，我们再执行取值操作时，就不会产生错误了，但如果网络超慢，10 秒钟后，这个字符串还没出现在页面上，取值操作仍然会报错，也就是说，我根本无法判断我需要去等待多少秒才是一个合理的等待时间。还有一种情况，如果等待 10 秒，但是只需要 1 秒钟字符串就出现在页面上了，那么余下的 9 秒就只有干等了，这样虽然不超时，但是会严重影响脚本运行的速度，也是极不合理的，那么如何让自动化脚本能够更智能些？当我们期待出现的字符串一出现时，等待就结束，这样是最理想的情况，如何实现这种情况？QTP 中的同步点就是实现这个原理，如果用过 QTP 的人应该很清楚。这个实现的原理也很简单，就是我们不断的去判断要取值的这个字符串是否出现在了页面上，只要一出现，立刻返回，这样就解决了问题，在 selenium2 中提供了两种方法来实现这个原理：

```
    public void testWait() {
       WebElement element = driver.findElement(By.className("wait"));
       element.click();
//          driver.manage().timeouts().implicitlyWait(12, TimeUnit.SECONDS);
       boolean wait = new WebDriverWait(driver, 10)
              .until(new ExpectedCondition<Boolean>() {
                 public Boolean apply(WebDriver d) {
                    return
d.findElement(By.className("red")).isDisplayed();
                 }
              });
```

```
        System.out.println(wait);

System.out.println(driver.findElement(By.className("red")).getText());
    }
```

第一种方式是用 driver.manage().timeouts().implicitlyWait(20, TimeUnit.SECONDS);这个是在 driver 对象 new 出来时，就加上这一句，是一个全局的等待超时时间的设置，但这个不可控，不太靠谱，不是太推荐大家用这种方法。

第二种方式也就是上述代码中的方法，主要是这一句：

```
boolean wait = new WebDriverWait(driver, 10).until(new
ExpectedCondition<Boolean>() {
                    public Boolean apply(WebDriver d) {
                        return
d.findElement(By.className("red")).isDisplayed();
                    }
                });
```

这一句的意义是指等待 class 为 red 的元素对象显示在页面上，在 10 秒内，什么时候出现，什么时候即退出等待，这个 10 是自己设置，也可以设置的时间长些。apply 里面的方法体可以自己写，也可以这样：

```
WebElement e = new WebDriverWait(driver, 10).until(new
ExpectedCondition<WebElement>() {
                    public WebElement apply(WebDriver d) {
                     if(d.findElement(By.className("red")).isDisplayed()){
                         return d.findElement(By.className("red"));
                     }else{
                         return null;
                     }
                    }
                });
```

希望大家在理解第一种写法后，再体会下第二种写法，务必掌握其中一种写法。

上面 testWait 方面的主要目的是，点击一个按钮，然后等待一个 class 为 red 的元素对象出现，出现后，取得这个对象的文本值。具体可以在 UI 上点击手动的点击一下 Wait 按钮，并演示运行一下代码。

以上所有的代码（详情请见 com.demo.Test3）

第六周：testNg 使用

写在前面的话：

为什么要用 testng,在前面的介绍中，我们都是在 main 方法中来自己调用方法来运行程序，但是我们的自动化的脚本会非常的多，难道每一个脚本都自己在 main 方法里面调用？如果这样的话，管理起来就会非常麻烦，有没有一种方法可以代替我们，testng 就应运而生了，testng 提供了一整套的解决方案，包括用例的组织，用例的运行，运行方式，报告的生成，且提供出了非常丰富的接口供扩展，下面我们来探索一下 testng 的一些运用方式吧！

Testng 的常用注脚介绍

@Test

见图 25，表示该方法是一个测试方法，在运行时，会自动的运行有@Test 注脚的方法。

```
public class Test1 {

    @Test
    public void test1(){
        System.out.println("test1");
    }

}
```

@BeforeMethod

见图 26，这个注脚是指加了该注脚的方法在测试方法运行之前会自动的被调用运行。

```
public class Test1 {

    @BeforeMethod
    public void setUp(){
        System.out.println("setUp method");
    }

    @Test
    public void test1(){
        System.out.println("test1");
    }

}
```

什么时候用这个注脚？比如每一个测试方法，都需要登录，那是不是得在每一个方法里面调用一下 login 的方法？那这样就太麻烦了，testng 的这个注脚就解决了这个问题，只需要在加了该注脚的方法中，调用一下 login 方法，在运行测试方法时，就会自动的调用了

@AfterMethod 见图 27，

```
public class Test1 {

    @BeforeMethod
    public void setUp(){
        System.out.println("setUp method");
    }

    @Test
    public void test1(){
        System.out.println("test1");
    }

    @AfterMethod
    public void tearDown(){
        System.out.println("tearDown method");
    }

}
```

这个注脚与 @BeforeMethod 呼应，也就是该注脚的方法在每一个测试方法运行之后会自动的被调用运行，值得注意的是，测试方法有可能会运行出现异常，比如断言失败或脚本本身异常，但是即使测试方法异常，@AfterMethod

这个注脚的方法也会被调用

多个测试方法时运行 见图 28

```
public class Test1 {

    @BeforeMethod
    public void setUp(){
        System.out.println("setUp method");
    }

    @Test
    public void test1(){
        System.out.println("test1");
    }

    @Test
    public void test2(){
        System.out.println("test2");
    }

    @AfterMethod
    public void tearDown(){
        System.out.println("tearDown method");
    }

}
```

@BeforeClass 与@AfterClass，这两个注脚的方法，也就是在每个类运行之前与之后会自动的被调用。

什么时候用这两个注脚？比如在自动化脚本运行时，一个类里面的所有测试方法设计的是在同一个浏览器里面运行，那么就是说在这个类对象产生之前，就要把浏览器给启动起来，这时候@BeforeClass 可以启动浏览器，@AfterClass 就可以关闭浏览器了。

@BeforeSuite，@AfterSuite 与@BeforeTest，@AfterTest，这四个注脚的方法，后面会介绍到什么是一个 suite，什么是一个 test

Testng 的数据驱动方法介绍

什么是数据驱动？数据驱动是指在一个脚本固定的情况下，用数据来控制该脚本是否运行，以及运行的次数，还有每次运行时对应的参数，

什么场景可以用数据驱动？比如我们在测试登录时，要测试用不同的帐户登录，难道我们需要针对每一个帐户去写一个脚本吗？这样显然是不明智的，于是，testng 为我们提供了这样一个注脚，让我们只需要提供出数据，就可以控制脚本运行的次数及相应的参数。

这个注脚就是@DataProvider 见图 29，

```java
public class Test2 {

    @BeforeMethod
    public void setUp(){
        System.out.println("setUp method");
    }

    @Test
    public void test1(){
        System.out.println("test1");
    }

    @Test
    public void test2(){
        System.out.println("test2");
    }

    @DataProvider
    public Object[][] dataProvider(){
        return new Object[][]{{"1"},{"2"}};
    }

    @Test(dataProvider="dataProvider")
    public void testData(String a){
        System.out.println(a);
    }

    @AfterMethod
    public void tearDown(){
        System.out.println("tearDown method");
    }

}
```

@DataProvider 需要注意的几个地方：

1.@DataProvider 注脚的方法的返回值是 Object 对象的二维数组

2. @DataProvider 可以指定名称如@DataProvider(name="testData")，这样在测试方法中 @Test(dataProvider="testData")，如果没有跟(name="testData"),则测试方法中的 dataProvider 的值就应该为 @DataProvider 注脚的方法名

Testng 使用 xml 去运行脚本

Testng 如果不借助于其它的工具，是有三种方式可以运行脚本的，第一种是直接在 eclipse 里运行，刚才已经演示过了，第二种是打成 jar 包，然后在 cmd 里用命令行运行：

java org.testng.TestNG -testclass org.test.MyTest

这种方式没什么实际意义，很少这样用，所以，只是介绍给大家，有兴趣的可以自己去试试！

我们在这重点介绍下第三种方式，用 XML 方式运行：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<suite name="Suite" verbose="1" parallel="false" thread-count="1">
    <test name="Test1">
        <classes>
            <class name="com.demo.Test2" />
        </classes>
```

```
    </test>
</suite>
```

从上面的 xml 中，可以看出 suite 为根节点，也就是一个 XML 文件中只能有一个 suite 节点，这里面的 suite 对应注脚中的@BeforeSuite，@AfterSuite, suite 节点最常用的几个属性都写在上面了，其中 name 属性为必需的属性，其 value 可以随便填写，verbose 属性是指“在控制台中如何输出”，一般情况下可不不需要这个属性，要写的话，值为 1 即可。Paraller 属性是指” 是否使用多线程测试”, thread-count 属性是指“如果设置了 parallel，可以设置线程数”，Paraller 与 thread-count 这两个属性一般是用在多线程并发时可以使用，这样可以提高运行效率。Test 节点，对应注脚中的@BeforeTest，@AfterTest，其属性 name 也是必需的属性。上面这个 xml 的意思是指运行 class “com.demo.Test2”中的所有和测试方法.

Xml 的第二种写法：

```
<?xml version="1.0" encoding="UTF-8"?>
<suite name="Suite" verbose="1" parallel="false" thread-count="1">
    <test name="Test1" preserve-order="true">
        <classes>
            <class name="com.demo.Test2">
                <methods>
                    <include name="test2" />
                    <include name="test1" />
                </methods>
            </class>
        </classes>
    </test>
</suite>
```

我们可以看到这个在这个 xml 多了一个 methods 的节点及 include 的子节点，这个 xml 的意思是指运行 class “com.demo.Test2”中的测试方法中的”test2”与”test1”，但是我们可以注意到在 test 节点中有一个属性是 preserve-order,这个属性的意思是指 include 的这些测试方法的执行顺序，默认值为 false,在为 false 时，include 的测试方法是按照测试方法名的升序排列运行的，如果改为 true 后，就会按照 include 中给定的顺序来进行运行，比如在上面的 xml 中，为 true 时，就会先运行 test2 再运行 test1,如果为 false 时，就会先运行 test1,再运行 test2,这样我们可以控制测试方法的执行顺序。值得注意的是，在一个 class 中的所有的测试方法的运行顺序默认是以测试方法名的升序排列的。

Xml 的第三种写法:

```
<?xml version="1.0" encoding="UTF-8"?>
<suite name="Suite" verbose="1" parallel="false" thread-count="1">
    <test name="Test1">
        <packages>
            <package name="com.demo" />
        </packages>
    </test>
</suite>
```

从这个 xml 中，我们可以看出是按照 package 为维度来执行测试方法的，上面的 XML 是指会执行 com.demo 这个 package 中的所有类中的所有测试方法。

还有一种按 group 的方法来运行的，现在不推荐大家使用，避免大家出现用例管理很混乱的局面，大家如果有兴趣的，可以去 testng 官网查看一下 group 的具体使用方法。

如何使用 Testng 完成测试用例及业务管理（case，suit）

上面的 XML 中介绍了几种运行方式，其实，都是以不同的维度来管理用例的，testng 也支持用例之间的依赖，也支持用例的优先级等等，但是个人建议，用例之间最好不要有依赖，一旦依赖，在分布式运行或者运行失败时，都会产生一些用例执行不完全的情况，用例的优先级，也暂时不支持大家去使用。

第七周：断言，截图，Log4j 介绍

如何完成检查点，断言类的使用

在做自动化测试中，断言是个很重要的概念，断言也就是检查点，重在判断我们通过页面得出来的值与期望值是否相等，如果相等，则代表断言成功，程序会继续往下执行，中果不相等，则代表断言失败，程序就会在断言失败处中止。看一段断言的代码：

```
public class Test4 {
    @Test
    public void testAssert1(){
        System.out.println("开始断言");
        Assert.assertEquals("1", "1");
        System.out.println("结束断言");
    }
    @Test
    public void testAssert2(){
        System.out.println("开始断言");
        Assert.assertEquals(1, 2,"比较两个数是否相等：");
        System.out.println("结束断言");
    }
}
```

用到了 Assert 类，我们这里的 Assert 类里 testng 里的 Assert 类，可以看出，Assert 类里面的都是静态方法，也就是(static)的方法，static 的方法可以不 new 对象，直接用类名就可以调用了，在 testAssert1 方法中，assertEquals 里面有两个参数，前一个是代表实际值，后一个是代表期望值，在 testAssert2 方法中，assertEquals 方法里有三个参数，第一个是实际值，第二个是期望值，第三个测试用户自定义的一段字符串，这第三个参数可以写，也可以不写。如果断言失败后，这段字符串会输出。从上面的代码中，可以看出 assertEquals 方法里的实际值与期望值，即可以是 int 型的，也可以是 String 类型的，当然，其它类型的也可以，也就是说 java 的数据类型的值都可以当作参数传给 assertEquals 方法中去,包括 List,Map,数组等都可以用这个方法进行比较。我们来看一下运行后的输出结果：

```
开始断言
结束断言
开始断言
PASSED: testAssert1
FAILED: testAssert2
java.lang.AssertionError: 比较两个数是否相等： expected [2] but found [1]
    at org.testng.Assert.fail(Assert.java:94)
```

testAssert1运行通过，testAssert2在输出”开始断言”后，没有输出”结束断言”，这是因为断言失败后，该测

试方法后面的语句就不执行了，就跳出测试方法并且标注该测试方法为失败了，且会输出失败的信息：java.lang.AssertionError: 比较两个数是否相等： expected [2] but found [1]
这一句中，AssertionError 是失败的异常类，”比较两个数是否相等：”这一句是我们在 assertEquals 方法中自定义的第三个，expected [2] but found [1]这是断言类给输出的，标注出了期望值与实际值。at org.testng.Assert.fail(Assert.java:94)这一种是指在程序的哪一行失败了。
Assert 类还有很多的静态断言方法去使用，不光只是基本数据类型的比较，还有

Assert.assertFalse(condition);//判断传进去的参数是为是false

Assert.assertNotEquals(actual1, actual2);//判断实际值与期望值是否不相同

Assert.assertNotNull(object);//判断传进去的参数值是否不为null

Assert.assertNotSame(actual, expected, message);//same与equals是不一样的，equals是比较实际值与期望值的值是否相等，same中实际值与期望值如果是基本数据类型与String类型的值，则是比较值是否相同，如果是非基本数据类型与String的，则是比较实际值与期望值的引用地址是否一样，举个例子：

```
String[] string1 = { "1", "2" };
String[] string3 = string1;
String[] string4 = { "1", "2" };
Assert.assertSame(string1,string3,"string1和string3不相同");
Assert.assertSame(string1, string4, "string1和string4不相等");
```

上面的代码中，string1与string3就是same的，而string1与string4就是notsame的

Assert.assertNull(object, message);//判断传进来的参数值是否为null

Assert.assertSame(actual, expected);//与assertNotSame对应

Assert.assertTrue(condition);//判断传进来的参数是否为 true

到此，对断言应该有了一个认识了，但还有一种场景是：假如对一个表格里的数据做一个循环比较，如果一断言失败就退出，那我们就无法一下子找出全部不符合要求的数据了，那么我们可不可以在断言时，如果断言失败则不退出，等到把整个循环做完后，再整体判断是否有断言失败的地方？解决方案如下：

```
public class Assertion {

    public static void verifyEquals(Object actual, Object expected){
        try{
            Assert.assertEquals(actual, expected);
        }catch(Error e){

        }
    }

    public static void verifyEquals(Object actual, Object expected, String
message){
        try{
            Assert.assertEquals(actual, expected, message);
        }catch(Error e){

        }
    }

}
```

增加了一个类，里面加了两个静态方法 verifyEquals,在方法体里面调用了 Assert.assertEquals，这样，当 Assert.assertEquals 断言失败后，异常就会 catch 住了，就不会跳出测试方法了，我们来看调用的方式：

```
@Test
    public void testAssert3(){
        System.out.println("开始断言3");
        Assertion.verifyEquals(1, 2, "比较两个数是否相等：");
        System.out.println("结束断言3");
    }
```

运行后，发现问题又来了，testAssert3 这个虽然断言后没有跳出测试方法，但是这个断言理论上是失败的，但被 testng 判断为了运行成功，这样就没达到目的，我们要在最后让 testng 判断 testAssert3 方法为失败，于是解决办法如下：

```
public class Assertion {

    public static boolean flag = true;

    public static void verifyEquals(Object actual, Object expected){
        try{
            Assert.assertEquals(actual, expected);
        }catch(Error e){
            flag = false;
        }
    }

    public static void verifyEquals(Object actual, Object expected, String
message){
        try{
            Assert.assertEquals(actual, expected, message);
        }catch(Error e){
            flag = false;
        }
    }

}
```

在里面加了一个标志符号，如果 catch 住异常后，就把标志符号赋为 false,这样在测试方法的最后判断一个标志符号是否为 true,如果为 true,测试整个断言过程中没有错误，如果为 false，则测试方法最后就被判定为失败：

```
    @Test
    public void testAssert3(){
        Assertion.flag = true;
        System.out.println("开始断言3");
        Assertion.verifyEquals(1, 2, "比较两个数是否相等：");
        System.out.println("结束断言3");
        Assert.assertTrue(Assertion.flag);
    }
```

这时候执行，就会发现 testAssert3 方法在断言失败后也全部运行完了，且最后也被判定为了运行失败。
具体的代码请参考：com.demo.Test4, com.demo.Assertion

如何在脚本中随意轻松的截图

截图是做测试是基本技能，在有 BUG 的地方，截个图，保留失败的证据，也方便去重现 BUG，所以，在自动化的过程中，也要能截图，也要能在我们想要截取的地方去截图，且能在错误产生时，自动的截图，我们先来看看在 selenium2 中是如何去实现截图的：

```
public class ScreenShot {
    public WebDriver driver;

    public ScreenShot(WebDriver driver) {
        this.driver = driver;
    }

    private void takeScreenshot(String screenPath) {
        try {
            File scrFile = ((TakesScreenshot) driver)
                    .getScreenshotAs(OutputType.FILE);
            FileUtils.copyFile(scrFile, new File(screenPath));
        } catch (IOException e) {
            System.out.println("Screen shot error: " + screenPath);
        }
    }

    public void takeScreenshot() {
        String screenName = String.valueOf(new Date().getTime()) + ".jpg";
        File dir = new File("test-output/snapshot");
        if (!dir.exists())
            dir.mkdirs();
        String screenPath = dir.getAbsolutePath() + "/" + screenName;
        this.takeScreenshot(screenPath);
    }
}
```

以上是一个截图的类，有了这个类以后，在测试方法中，我们想要截取的地方，new 一个 ScreenShot 的类，然后调用 takeScreenshot 方法即可：

```
    @Test
    public void testBaidu(){
        WebDriver driver = new FirefoxDriver();
        driver.manage().window().maximize();
        driver.navigate().to("http://www.baidu.com");
        ScreenShot ss = new ScreenShot(driver);
        ss.takeScreenshot();
        driver.close();
```

```
        driver.quit();
    }
```

至于具体如何在产生错误的地方去截图，我们在框架部分再去讲解,请大家再次深刻理解一下这个截图的方法

以上的代码请参考：com.demo.ScreenShot, com.demo.Test5

Log4j 的使用，构建更加详细的日志体系

有了断言，有了截图，当出现失败的用例时，我们就会根据这些来分析一下用例为什么失败了，但往往这些信息在我们查找失败原因时发挥不了作用，要说查找错误最直接的，还是 LOG 了，根据 LOG 的内容，就可以大概的判断出在哪一行以及大概的错误，LOG4J 是一个被泛使用的 LOG 框架，不管是开发还是测试，LOG4J 都很流行，LOG4J 就是提供了一个 JAR 包，然后我们配置一下，配置完成后，在程序里直接调用即可，可能有的人认为 LOG 是自动生成的，其实不是的，LOG 的内容都是我们在程序中写好的，所以，一个好的程序员，要学会写个注释及加上好的 LOG。我们来看一相配置方法：

log4j.properties

```
log4j.rootLogger=INFO, stdout, fileout
log4j.logger.TestProject=INFO
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %c : %m%n
log4j.appender.fileout=org.apache.log4j.FileAppender
log4j.appender.fileout.File=c:/test.log
log4j.appender.fileout.layout=org.apache.log4j.PatternLayout
log4j.appender.fileout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %c : %m%n
```

其中 log4j.logger.TestProject=INFO 这一句是给这个叫 TestProject 的 logger 定义输出信息的级别为 INFO，一般我们做自动化测试的，定义为 INFO 即可，其它配置的含义大家如果有兴趣的可以上网搜索一下，自己搜集一下这些配置的意义，我上面给出的这个配置，大家直接用就可以了，大部分的需求都可以满足。

配置好以后，我们就要定义一个 Log 类了，好让我们在测试方法中去调用：

```
public class Log {
    private static Logger logger;

    private static String filePath = "src/log4j.properties";

    private static boolean flag = false;

    private static synchronized void getPropertyFile() {
        logger = Logger.getLogger("TestProject");
        PropertyConfigurator.configure(new
File(filePath).getAbsolutePath());
        flag = true;
    }

    private static void getFlag() {
```

```
        if (flag == false)
            Log.getPropertyFile();
    }

    public static void logInfo(String message) {
        Log.getFlag();
        logger.info(message);
    }

    public static void logError(String message) {
        Log.getFlag();
        logger.error(message);
    }

    public static void logWarn(String message) {
        Log.getFlag();
        logger.warn(message);
    }
}
```

在测试方法中的使用(一般情况下我们做测试的，只使用 logInfo 这个方法即可):

```
public class Test6 {
    @Test
    public void testLog(){
        Log.logInfo("输出log");
    }
}
```

运行这个测试方法后，在控制台及配置的 c:/test.log 中都可以看到输出的 log。

具体的代码详情: com.demo.Log, com.demo.Test6,log4j.properties

第八周：page-object 模式介绍

Page-object 思想介绍

什么是 page-object？顾名思义就是页面对象，引申到自动化测试中去，就是一个页面当成一个对象，这样做有何好处？在自动化测试中，后期维护是一个大难题，我们来看一句代码：driver.findElement(By.id(“user”)).sendKeys(“123”),这句代码里包含了两个信息，一个是 id(“user”),一个是数据”123”,id(“user”)这个，也就是定位一个元素对象，可能在很多脚本中都要用到，于是乎，问题来了，如果哪天前端工程师随手把这个元素对象的 id 给改了，改成了 id(“user1”)，那这时候涉及到这个元素对象的脚本就都要失败了，运行完后，报告一出来，一片红啊，这报告能发出去吗？能给开发人员看吗？久而久之，开发人员还能相信你的自动化测试报告吗？然后，慢慢的，自动化就变成了坑，以上的例子绝对不是危言耸听，自动化之所以在很多公司只是个摆设，与前期的设计，与测试人员的水平，与沟通不够都有很大的关系，我们现在能做的，就是在前期的设计中把好关，摆脱这种情况。那么问题来了，当元素对象改动时，

如何花最小的代价，来更改我们的脚本？这里可以用到 JAVA 的封装的特性，就是把这个元素对象给找个地方集中管理起来，比如 By.id(“user”)是一个 By 对象，我们只需要管理集中管理 By 对象:

```
public class Test7 {

    public static By input = By.id("user");

    public static By link = By.xpath("//div[@id='link']/a");

}
```

这样在用例中调用的时候这样：

```
WebElement element = driver.findElement(Test7.input);
```

这样管理起来后，如果元素发生了改变，我们只需要在 Test7.java 文件里，找到这个元素对象，然后进行更改即可，就不用去更改脚本本身了，这样的代价就小了很多，但是这样也有不方便的，所有的元素对象都放在这里，量级一大，那么在寻找的时候就比较麻烦了，那么我们再细分一下，把在同一个页面中的元素对象都放到一个具体的类里面，这样呢，在寻找的时候，只需要根据事先定义好的类名，就知道是哪一个页面了，这样，把同一个页面的元素放到一个页面类里，这就是 page-object 模式，当然，由于这个页面放有元素对象，我们也可以把这个页面上的一些逻辑组合封装成方法，然后放到这个页面类里，这样是不是又更加的方便了。

运用 page-object 重构脚本及实例演示

我们来把原来写的 Test3.java 进行重构，用 page-object，再结合 testng.

先建一个 page 类：DemoPage.java

```
public class DemoPage {
    public static By input = By.id("user");
    public static By link = By.xpath("//div[@id='link']/a");
    public static By select = By.cssSelector("select[name='select']");
    public static By radio = By.name("identity");
    public static By check = By.xpath("//div[@id='checkbox']/input");
    public static By button = By.className("button");
    public static By alert = By.className("alert");
    public static By action = By.className("over");
    public static By upload = By.id("load");
    public static String iframe = "aa";
    public static By multiWin = By.className("open");
    public static By wait = By.className("wait");
}
```

再建一个 iframe 的 page 类:IframePage.java

```
public class IframePage {
    public static By input = By.id("user");
}
```

这样在调用时，就成了：

```
WebElement element = driver.findElement(DemoPage.input);
```

我们以前的时候讲 Wait 时，就说过，要有同步点的概念，就是要等元素对象出现的时候才进行操作，要不然，就没有意义了，是吧，所以，可以把这个操作进行一下封装，进行改造一下,增加一个 wait 的方法:

```
public boolean waitToDisplayed(WebDriver driver, final By key){
        boolean waitDisplayed = new WebDriverWait(driver, 10).until(new
ExpectedCondition<Boolean>() {
            public Boolean apply(WebDriver d) {
                return d.findElement(key).isDisplayed();
            }
        });
        return waitDisplayed;
    }
```

但是这个 waitToDisplayed 如何跟 driver.findElement(DemoPage.input)这一句结合起来使用呢？不管如何，我们最终要返回的是一个 WebElement 对象，所以再加一个方法：

```
    public WebElement getElement(WebDriver driver, By key){
        WebElement element = null;
        if(this.waitToDisplayed(driver, key)){
            element = driver.findElement(key);
        }
        return element;
    }
```

这样，我们来看在封装逻辑方法时，如何写：

```
    public void input(WebDriver driver){
        WebElement element = this.getElement(driver, input);
        element.sendKeys("test");
        element.clear();
        element.sendKeys("test");
        String text = element.getAttribute("value");
        System.out.println(text);
    }
```

这样是不是方便多了？但是，如果有多个 page,我们是不是得把 waitToDisplayed 与 getElement 方法在每一个 page 里都写一遍？于是，我们就用到 java 中的继承来解决这个问题，只需要把这两个方法写到一个父类里，其它的 page 都继承这个父类:

```
public class Page {
    private boolean waitToDisplayed(WebDriver driver, final By key){
        boolean waitDisplayed = new WebDriverWait(driver, 10).until(new
ExpectedCondition<Boolean>() {
            public Boolean apply(WebDriver d) {
                return d.findElement(key).isDisplayed();
            }
        });
        return waitDisplayed;
    }

    protected WebElement getElement(WebDriver driver, By key){
```

```
        WebElement element = null;
        if(this.waitToDisplayed(driver, key)){
            element = driver.findElement(key);
        }
        return element;
    }
}
```

在 DemoPage.java 中

```
public class DemoPage extends Page{
    public static By input = By.id("user");
    public static By link = By.xpath("//div[@id='link']/a");
    public static By select = By.cssSelector("select[name='select']");
    public static By radio = By.name("identity");
    public static By check = By.xpath("//div[@id='checkbox']/input");
    public static By button = By.className("button");
    public static By alert = By.className("alert");
    public static By action = By.className("over");
    public static By upload = By.id("load");
    public static String iframe = "aa";
    public static By multiWin = By.className("open");
    public static By wait = By.className("wait");

    public void input(WebDriver driver){
        WebElement element = this.getElement(driver, input);
        element.sendKeys("test");
        element.clear();
        element.sendKeys("test");
        String text = element.getAttribute("value");
        System.out.println(text);
    }
}
```

现在每一个方法里，都要传一个 driver 的对象进去，这样也不方便，可以把 driver 搞成类的一个属性：

```
public class Page {

    protected WebDriver driver;

    public Page(WebDriver driver) {
        this.driver = driver;
    }

    private boolean waitToDisplayed(final By key){
        boolean waitDisplayed = new WebDriverWait(driver, 10).until(new
ExpectedCondition<Boolean>() {
            public Boolean apply(WebDriver d) {
```

```
                return d.findElement(key).isDisplayed();
            }
        });
        return waitDisplayed;
    }

    protected WebElement getElement(By key){
        WebElement element = null;
        if(this.waitToDisplayed(key)){
            element = driver.findElement(key);
        }
        return element;
    }
}
```

子类中的变化：(由于这些元素对象就只在 page 中使用了，不会在脚本中出现了，所以把 static 给去掉)

```
public class DemoPage extends Page{

    public By input = By.id("user");
    public By link = By.xpath("//div[@id='link']/a");
    public By select = By.cssSelector("select[name='select']");
    public By radio = By.name("identity");
    public By check = By.xpath("//div[@id='checkbox']/input");
    public By button = By.className("button");
    public By alert = By.className("alert");
    public By action = By.className("over");
    public By upload = By.id("load");
    public String iframe = "aa";
    public By multiWin = By.className("open");
    public By wait = By.className("wait");

    public DemoPage(WebDriver driver) {
        super(driver);
    }

    public void input(){
        WebElement element = this.getElement(input);
        element.sendKeys("test");
        element.clear();
        element.sendKeys("test");
        String text = element.getAttribute("value");
        System.out.println(text);
    }
}
```

我们现在来看一下如何的写自动化的脚本了：在 Test9.java 中

```
public class Test9 {

    private WebDriver driver;

    @BeforeClass
    public void setUp(){
        driver = new FirefoxDriver();
        driver.manage().window().maximize();

    driver.navigate().to("file:///D://%E4%B8%AA%E4%BA%BA%E6%96%87%E6%A1%A3/
/demo.html");
    }

    @Test
    public void testInput(){
        DemoPage dp = new DemoPage(driver);
        dp.input();
    }

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

}
```

这里我只演示了 input,其它的几个方法的重构，请大家自己动手来重构一下！

第二部分：进阶

第九周：框架思想介绍

为什么要写框架？可维护性，提高编写脚本效率，提高脚本的可读性
框架的几大要素：driver 管理，脚本，数据，元素对象，LOG，报告，运行机制，失败用例重复运行等
框架的分层思想：脚本，数据，元素对象分离
框架如何持续集成？如何定时运行？

框架，这个词经常出现在开发人员口中，随着自动化测试的深入发展，测试人员也越来越多的去追求所谓的框架了，都想写出一个框架，但什么是框架呢？比如在做房子时，脚手架一搭好，工人就按着这个架子

来添砖加瓦，所以，大家可以把这个框架理解为一个脚手架，搭好后，只需要按着一定的约定，往里面堆用例即可，这样肯定能提高效率了，但做房子还要讲究朝向，还要是否好看，还要把质量做好，真出现问题时，要能最快的修理好，还要有良好的供水供电供气及排水等各种基础设施，所以，自动化的框架中，也注定了要有这些基础设施，所以，自动化的框架最基本的要能提高编写脚本的效率，写出来的脚本，要能让别人一眼看懂，要不然，这房子就没人会买，脚本的可维护性也要高，当脚本运行出现错误后，要能最快的定位到错误，分析出原因，所以就要有一些基础设施，比如，log,报告等，这就是框架的好处。现在我们来回归到脚本本身，一个脚本，无非就是包含了业务逻辑，元素对象，测试数据，这三大主要部分，所以，我们把这三块最主要的先给处理好，也就是我们常说的分层思想，为什么要分层呢？我们常听一个写代码的概念：高类聚，低藕合。也就是说要模块化，对象化。只有把业务逻辑，元素对象，测试数据进行分享后，我们就能更好的维护我们写的脚本，让编写脚本的人只关注业务逻辑本身，这样就更容易提高编写脚本的效率。下面的章节中，我们会介绍到如何编写一个框架。

要写一个框架，首先会有一些基础的工作要做，比如会有一些全局的配置文件，我们今天会介绍如何去做一些基础的工作，以及读取这些全局的配置文件，配置文件会用 XML 格式的文件，当然也可以用其它格式的文件，大家自己去解析即可，我们下面来介绍如何解析：

1.新建一个 TestAutomation 的 Java 项目

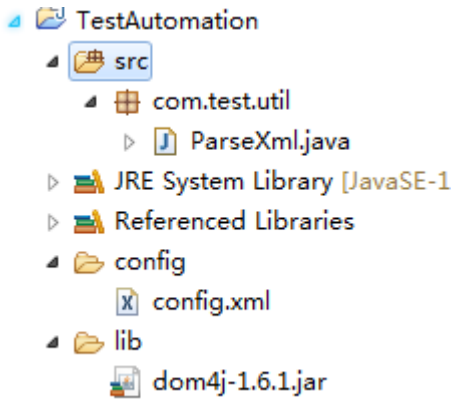
2.在这个项目下面新建一个 config 的文件夹，在 config 文件夹里面新建一个叫 config.xml 的文件

3.新建一个包：com.test.util

4.在项目下面再新建一个 lib 的文件夹，放入 dom4j.jar 与 jaxen.jar，用来解析 config.xml 文件

5.再新建一个 ParseXml.java 的文件。

结构及 config.xml 的内容如下：



config.xml

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <config>
3     <browser>firefox</browser>
4     <waitTime>10</waitTime>
5 </config>
```

在 config.xml 中，browser 结点表示我们所需要运行的浏览器，waitTime 是指我们前面讲过的等待元素对象出现的最长 timeout 时间，我们暂时只加这两个结点，以后需要的话，就接着来添加，我们现在在 ParseXml.java 中来解析这个 xml 文件：

```
public class ParseXml {
    /**
     * 解析xml文件，我们需要知道xml文件的路径，然后根据其路径加载xml文件后，生成一个Document的对象，
```

```
     * 于是我们先定义两个变量String filePath,Document document
     * 然后再定义一个load方法，这个方法用来加载xml文件，从而产生document对象。
     */
    private String filePath;

    private Document document;

    /**
     *  构造器用来new ParseXml对象时，传一个filePath的参数进来,从而初始化filePath
的值
     * 调用load方法，从而在ParseXml对象产生时，就会产生一个document的对象。
     */
    public ParseXml(String filePath) {
        this.filePath = filePath;
        this.load(this.filePath);
    }

    /**
     * 用来加载xml文件，并且产生一个document的对象
     */
    private void load(String filePath){
        File file = new File(filePath);
        if (file.exists()) {
            SAXReader saxReader = new SAXReader();
            try {
                document = saxReader.read(file);
            } catch (DocumentException e) {
                System.out.println("文件加载异常: " + filePath);

            }
        } else{
            System.out.println("文件不存在 : " + filePath);
        }
    }

}
```

Document 的对象产生后，dom4j 提供了一种可以用 xpath 来解析 xml 的方法，于是可以看到：

```
/**
     * @param elementPath elementPath是一个xpath路径,比如"/config/driver"
     * @return 返回的是一个节点Element对象
     */
    private Element getElementObject(String elementPath) {
        return (Element) document.selectSingleNode(elementPath);
    }
```

我们再加一个返回的这个 Element 对象是不是存在的方法：

```
/**
     * 用xpath来判断一个结点对象是否存在
     */
    public boolean isExist(String elementPath){
        boolean flag = false;
        Element element = this.getElementObject(elementPath);
        if(element != null) flag = true;
        return flag;
    }
```

我们最终是要得到这个结点的值：

```
/**
     * 用xpath来取得一个结点对象的值
     */
    public String getElementText(String elementPath) {
        Element element = this.getElementObject(elementPath);
        if(element != null){
            return element.getText().trim();
        }else{
            return null;
        }
    }
```

我们来测试一下这些个方法的用途：

```
public static void main(String[] args) {
        ParseXml px = new ParseXml("config/config.xml");//给定config.xml的路
径
        String browser = px.getElementText("/config/browser");
        System.out.println(browser);
        String waitTime = px.getElementText("/config/waitTime");
        System.out.println(waitTime);
    }
```

我们已经把 xml 里面的值给取出来了，这时候，我们再把 config 给静态化，持久化一下，新建一个类:com.test.util.Config.java

```
public class Config {

    public static String browser;

    public static int waitTime;

    /**
     * static{}，这种用法请大家务必搞清楚，这代表在用到Config这个类时，这个static{}
里面的内容会被执行一次，且只被执行一次，就算多
     * 次用到Config类，也只执行一次，所以，这个static{]一般就在加载一些配置文件，
也可以说类似于单例模式。
     */
```

```
    static{
        ParseXml px = new ParseXml("config/config.xml");
        browser = px.getElementText("/config/browser");
        waitTime = Integer.valueOf(px.getElementText("/config/waitTime"));
    }

    public static void main(String[] args) {
        System.out.println(Config.browser);
        System.out.println(Config.waitTime);
    }
}
```

如些一来，配置文件解析成功，那么我们可以根据其给定的 browser 节点的值来判定该用什么样的 driver，该启动什么样的浏览器,我们再新建一个 com.test.base.SeleniumDriver 类，来启动 driver,(首先在 lib 中加入 selenium-server 的 jar 包,再新建一个 files 的文件夹，把 chromedriver.exe 与 iedriver.exe 都放进去)

```
public class SeleniumDriver {

    private WebDriver driver;

    public WebDriver getDriver() {
        return driver;
    }

    public SeleniumDriver() {
        this.initialDriver();
    }

    private void initialDriver(){
        if("firefox".equals(Config.browser)){
            driver = new FirefoxDriver();
        }else if("ie".equals(Config.browser)){
            System.setProperty("webdriver.ie.driver",
"files/IEDriverServer64.exe");
            DesiredCapabilities capabilities =
DesiredCapabilities.internetExplorer();

capabilities.setCapability(InternetExplorerDriver.INTRODUCE_FLAKINESS_BY_IG
NORING_SECURITY_DOMAINS, true);
           capabilities.setCapability("ignoreProtectedModeSettings",true);
            driver = new InternetExplorerDriver(capabilities);
        }else if("chrome".equals(Config.browser)){
            System.setProperty("webdriver.chrome.driver",
"files/chromedriver.exe");
            ChromeOptions options = new ChromeOptions();
            options.addArguments("--test-type");
```

```
            driver = new ChromeDriver(options);
        }else{
            System.out.println(Config.browser+" 的值不正确，请检查！");
        }
    }
}
```

这样在 config.xml 里面的 browser 节点的值对应后，就启动相应的浏览器，我们来试验一下：

```
public static void main(String[] args) {
        SeleniumDriver selenium = new SeleniumDriver();
        WebDriver driver = selenium.getDriver();
        driver.navigate().to("http://www.baidu.com");
        driver.close();
        driver.quit();
    }
```

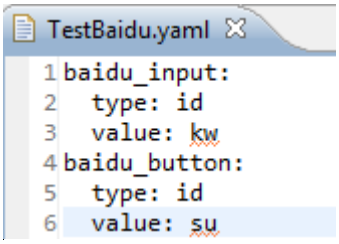
于是，我们这节课完成了配置文件的解析，初始化 driver 对象，下一节课，我们将介绍如何对元素进行管理。

第十周：搭建框架一（元素管理）

为何要单独的进行元素管理？
Yaml 文件进行元素管理
在元素管理中引入同步点机制
元素管理实例解析

元素单独管理的好处，我们前面 PO 模式中也讲过了，前面用的是在一个 PAGE 中把元素对象都写在里面，但是 JAVA 是需要编译的，如果元素对象需要改动，则每次要重新编译，甚至要重新打包，这样会很麻烦，在前端开发中，开发人员通常是把 UI 样式放在 CSS 文件中，受此影响，我们也可以把我们的 locator 放在一个专门的 文件中，按照页面来分类，提取其公共的 locator 放在公共的文件中，这样或许可以提升些许编写脚本速度及后期维护成本，效果就是如果 UI 变了，我们只 需要修改对应的页面中的 locator 就行了，脚本都不需要重新编译（如果是用需要编译的语言，如 JAVA），这些专门的外部文件有很多，如 XML，EXCEL，YAML 等格式的文件，这些格式文件各有各的好处，根据个人的习惯来挑选合适的文件来保存，在这里给大家演示一下如何用 YAML 文件来保存元素对象：

1、 文件类型------yaml



2、 java 解析 yaml 文件所需要的 jar 包:jyaml-1.3.jar，需自己在网上下载。

3、 格式介绍：

a. baidu_input 后面接上":",直接回车，然后空两格

b. type 与 value 这两个 key 是固定的，后面接上":",然后空一格，也可以不空，如果 value 后面是 xpath，建议用加上引号，具体去看下 yaml 的格式，百度一大堆。

c. 在 webdriver 中，有 By.id,By.name,By.xpath,By.className,By.linkText 等，我们选取这几种常见的，所以 type 的冒包后面可用的值为 id,name,xpath

d. value 的值为 type 中对应的类型的值，比如百度首页上的输入框的 id='kw',所以在 yaml 文件中的写法如上图所示

4、 解析上述的 yaml 文件：

```
public class Locator {

    private String yamlFile;

    public Locator() {
        yamlFile = "demo";
        this.getYamlFile();
    }

    private Map<String, Map<String, String>> ml;

    @SuppressWarnings("unchecked")
    public void getYamlFile() {
        File f = new File("locator/" + yamlFile + ".yaml");
        try {
            ml = Yaml.loadType(new FileInputStream(f.getAbsolutePath()),
                    HashMap.class);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

}
```

可以在本地创建一个 demo.yaml 文件，保存在 locator 目录中，locator 与 src 同级目录，然后写个 main 方法来调用一个 getYamlFile 方法，可以看到解析 demo.yaml 后的值都赋给了变量 ml。解析过程如此简单，解析速度如此之快，yaml 文件也比较直观，这 是我选择用 yaml 文件的原因，当然可能还有其它更好的选择，大家可以自行尝试。

5、 我们在写脚本时，元素对象一般是这样写的 WebElement element = driver.findElement(By.id("kw"));所以接下来我们要把 ml 变量里的"value"转换成 By 对象。添加如下代码

```
    private By getBy(String type, String value) {
        By by = null;
        if (type.equals("id")) {
            by = By.id(value);
        }
        if (type.equals("name")) {
            by = By.name(value);
        }
        if (type.equals("xpath")) {
            by = By.xpath(value);
```

```
        }
        if (type.equals("className")) {
            by = By.className(value);
        }
        if (type.equals("linkText")) {
            by = By.linkText(value);
        }
        if(type.equals("css")){
        by = By.cssSelector(value);
        }
        return by;
       }
```

这样通过 ml 中的 type 与 value 的值就对产生一个 By 对象。

6、 By 对象产生后，就可以把这个对象传给 driver.findElement 方法，继而生成一个 WebElement 对象.

```
public class Locator {

    private String yamlFile;

    private WebDriver driver;

    public Locator(WebDriver driver) {
        yamlFile = "demo";
        this.getYamlFile();
        this.driver = driver;
    }

    private Map<String, Map<String, String>> ml;

    @SuppressWarnings("unchecked")
    public void getYamlFile() {
        File f = new File("locator/" + yamlFile + ".yaml");
        try {
            ml = Yaml.loadType(new FileInputStream(f.getAbsolutePath()),
                    HashMap.class);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    private By getBy(String type, String value) {
        By by = null;
        if (type.equals("id")) {
            by = By.id(value);
        }
```

```
        if (type.equals("name")) {
            by = By.name(value);
        }
        if (type.equals("xpath")) {
            by = By.xpath(value);
        }
        if (type.equals("className")) {
            by = By.className(value);
        }
        if (type.equals("linkText")) {
            by = By.linkText(value);
        }
        if(type.equals("css")){
        by = By.cssSelector(value);
        }
        return by;
    }

    public WebElement getElement(String key) {
        String type = ml.get(key).get("type");
        String value = ml.get(key).get("value");
        return driver.findElement(this.getBy(type, value));
    }

    public static void main(String[] args){
    SeleniumDriver selenium = new SeleniumDriver();
    Locator d = new Locator(selenium.getDriver());
        WebElement element = d.getElement("百度输入框");
        element.sendKeys("");
    }

}
```

7、 到这里，已经成功了一半，因为已经把 yaml 文件中保存的元素成功的转化成了 WebElement 对象。但是还不够，接下来我们引入一下同步点的概念，就 是在调用 locator 时，保证 locator 是显示在页面上的，webdriver 中有个 WebDriverWait 对象，代码如下：

```
    private WebElement watiForElement(final By by) {
        WebElement element = null;
        int waitTime = Config.waitTime;
        try {
            element = new WebDriverWait(driver, waitTime).until(new
ExpectedCondition<WebElement>() {
                public WebElement apply(WebDriver d) {
                    return d.findElement(by);
                }
```

```
            });
        } catch (Exception e) {
            System.out.println(by.toString() + " is not exist until " +
waitTime);
        }
        return element;
    }
```

于是乎 getElement 方法里面就可以改为

```
    public WebElement getElement(String key) {
       String type = ml.get(key).get("type");
       String value = ml.get(key).get("value");
       return this.watiForElement(this.getBy(type, value));
    }
```

8、 到这一步，又改进了一点，新的问题也随之产生了，watiForElement 这个方法，返回的 WebElement 对象包括隐藏的，如果是隐藏的，那么在操作的时候，自然而然会报错，所以，我们得把隐藏的去掉，只显示 displayed 的元素对象，增加一个方法。

```
    private boolean waitElementToBeDisplayed(final WebElement element) {
        boolean wait = false;
        if (element == null)
            return wait;
        try {
            wait = new WebDriverWait(driver, Config.waitTime).until(new
ExpectedCondition<Boolean>() {
                public Boolean apply(WebDriver d) {
                    return element.isDisplayed();
                }
            });
        } catch (Exception e) {
            System.out.println(element.toString() + " is not displayed");
        }
        return wait;
    }
```

如此一来，getElement 方法又可以改进一下了。

```
    public WebElement getElement(String key) {
        String type = ml.get(key).get("type");
        String value = ml.get(key).get("value");
        WebElement element = this.watiForElement(this.getBy(type, value));
        if(!this.waitElementToBeDisplayed(element)){
         element = null;
        }
        return element;
    }
```

9、 既然有等待元素对象显示的，那么反之就有等待元素对象消失的方法。

```
   public boolean waitElementToBeNonDisplayed(final WebElement element) {
```

```
        boolean wait = false;
        if (element == null)
            return wait;
        try {
            wait = new WebDriverWait(driver,Config.waitTime).until(new
ExpectedCondition<Boolean>() {
                        public Boolean apply(WebDriver d) {
                            return !element.isDisplayed();
                        }
                    });
        } catch (Exception e) {
            System.out.println("Locator [" + element.toString() + "] is also
displayed");
        }
        return wait;
        }
```

10、 看上去一切很美好了，but....如果我们要验证一个元素对象不出现在页面上，就会出现问题了，于是增加一个方法

```
    public WebElement getElementNoWait(String key) {
        WebElement element = null;
        String type = ml.get(key).get("type");
        String value = ml.get(key).get("value");
        try{
            element = driver.findElement(this.getBy(type, value));
        }catch(Exception e){
            element = null;
        }
        return element;
    }
```

11、 现在的问题是 getElement 与 getElementNoWait 的方法体很接近，于是我们来重构下这部分的代码，先增加一个方法，存放相同的方法体

```
    private WebElement getLocator(String key, boolean wait) {
        WebElement element = null;
        if (ml.containsKey(key)) {
            Map<String, String> m = ml.get(key);
            String type = m.get("type");
            String value = m.get("value");
            By by = this.getBy(type, value);
            if (wait) {
                element = this.watiForElement(by);
                boolean flag = this.waitElementToBeDisplayed(element);
                if (!flag)
                    element = null;
            } else {
```

```
                try {
                    element = driver.findElement(by);
                } catch (Exception e) {
                    element = null;
                }
            }
        } else
            System.out.println("Locator " + key + " is not exist in " + yamlFile
                    + ".yaml");
        return element;
    }
```

再把 getElement 与 getElementNoWait 方法进行修改

```
public WebElement getElement(String key) {
        return this.getLocator(key, true);
    }

    public WebElement getElementNoWait(String key) {
        return this.getLocator(key, false);
    }
```

12、 到现在为止，已经可以满足绝大部分的需求了，完全可以使用了，下面的任务就是来点锦上添花了，举个例子，在 yaml 文件中，允许使用参数，比如

```
baidu_input:
    type: id
    value: "%s%s"
baidu_button:
    type: id
    value: "%s"
```

在这里面的参数用%s 来表示，于是在脚本中，我们调用 getElement 与 getElementNoWait 方法时需要我们把 value 给传进去，我们再来处理下这部分，增加一个方法。

```
    private String getLocatorString(String locatorString, String[] ss) {
        for (String s : ss) {
            locatorString = locatorString.replaceFirst("%s", s);
        }
        return locatorString;
    }
```

在上面的方法中，我们可以看到，对于 value 值，我们是通过一个数组循环的去替代里面的%s,再把该方法结合到 getLocator 方法中去。

```
private WebElement getLocator(String key, String[] replace, boolean wait) {
        WebElement element = null;
        if (ml.containsKey(key)) {
            Map<String, String> m = ml.get(key);
            String type = m.get("type");
            String value = m.get("value");
            if (replace != null)
```

```
                value = this.getLocatorString(value, replace);
            By by = this.getBy(type, value);
            if (wait) {
                element = this.watiForElement(by);
                boolean flag = this.waitElementToBeDisplayed(element);
                if (!flag)
                    element = null;
            } else {
                try {
                    element = driver.findElement(by);
                } catch (Exception e) {
                    element = null;
                }
            }
        } else
            System.out.println("Locator " + key + " is not exist in " + yamlFile
                    + ".yaml");
        return element;
    }
```

可以看到 getLocator 方法的参数变了，于是要重新的更改 getElement 与 getElementNoWait 方法，同时重载这两个方法：

```
    public WebElement getElement(String key) {
        return this.getLocator(key, null, true);
    }

    public WebElement getElementNoWait(String key) {
        return this.getLocator(key, null, false);
    }

    public WebElement getElement(String key, String[] replace) {
        return this.getLocator(key, replace, true);
    }

    public WebElement getElementNoWait(String key, String[] replace) {
        return this.getLocator(key, replace, false);
    }
```

13、 惊喜？更大的还在后面。再举个例子：

```
baidu_input:
    type: xpath
    value: "//div[@id='%productId%']//div"
baidu_button:
    type: xpath
    value: "//div[@id='%productId%']//input[@name='button']"
```

类似于上面这种，整个里面都含有 productId，于是我们可以通过一个方法，调用这个方法后，里面的都会

被替换掉，该方法如下。

```
    public void setLocatorVariableValue(String variable, String value){
    if(ml!=null){
        Set<String> keys = ml.keySet();
            for(String key:keys){
                String v =
ml.get(key).get("value").replaceAll("%"+variable+"%", value);
                ml.get(key).put("value",v);
            }
    }
    }
```

14、 再比如，有一些元素对象是每个页面都会出现的，是公共的，这些公共的 locator 只是有时候要用到，大部分时候都不用，所以，我们把这些公共的放在一个特定的文件里，在需要的时候通过外部加载来使用这些公共的 locator,增加一个变量与方法。

```
    private HashMap<String, HashMap<String, String>> extendLocator;

    @SuppressWarnings("unchecked")
    public void loadExtendLocator(String fileName){
        File f = new File("locator/" + fileName + ".yaml");
        try {
            extendLocator = Yaml.loadType(new
FileInputStream(f.getAbsolutePath()),
                    HashMap.class);
            ml.putAll(extendLocator);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
```

15、 到此为止，整个元素对象管理就结束了，这只是提供一个思路，大家如果有耐心从上到下的给按着来写一遍，应该会了解不少。最后来个总结性的代码。

```
public class Locator {

    private String yamlFile;

    private WebDriver driver;

    private Map<String, Map<String, String>> extendLocator;

    public Locator(WebDriver driver) {
        yamlFile = "demo";
        this.getYamlFile();
        this.driver = driver;
```

```
    }

    private Map<String, Map<String, String>> ml;

    @SuppressWarnings("unchecked")
    public void getYamlFile() {
        File f = new File("locator/" + yamlFile + ".yaml");
        try {
            ml = Yaml.loadType(new FileInputStream(f.getAbsolutePath()),
                    HashMap.class);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    private By getBy(String type, String value) {
        By by = null;
        if (type.equals("id")) {
            by = By.id(value);
        }
        if (type.equals("name")) {
            by = By.name(value);
        }
        if (type.equals("xpath")) {
            by = By.xpath(value);
        }
        if (type.equals("className")) {
            by = By.className(value);
        }
        if (type.equals("linkText")) {
            by = By.linkText(value);
        }
        if(type.equals("css")){
         by = By.cssSelector(value);
        }
        return by;
    }

    private WebElement watiForElement(final By by) {
        WebElement element = null;
        int waitTime = Config.waitTime;
        try {
            element = new WebDriverWait(driver, waitTime).until(new
ExpectedCondition<WebElement>() {
```

```
                public WebElement apply(WebDriver d) {
                    return d.findElement(by);
                }
            });
        } catch (Exception e) {
            System.out.println(by.toString() + " is not exist until " +
waitTime);
        }
        return element;
    }

    private boolean waitElementToBeDisplayed(final WebElement element) {
        boolean wait = false;
        if (element == null)
            return wait;
        try {
            wait = new WebDriverWait(driver, Config.waitTime).until(new
ExpectedCondition<Boolean>() {
                public Boolean apply(WebDriver d) {
                    return element.isDisplayed();
                }
            });
        } catch (Exception e) {
            System.out.println(element.toString() + " is not displayed");
        }
        return wait;
    }

    public boolean waitElementToBeNonDisplayed(final WebElement element) {
        boolean wait = false;
        if (element == null)
            return wait;
        try {
            wait = new WebDriverWait(driver,Config.waitTime).until(new
ExpectedCondition<Boolean>() {
                    public Boolean apply(WebDriver d) {
                        return !element.isDisplayed();
                    }
                });
        } catch (Exception e) {
            System.out.println("Locator [" + element.toString() + "] is also
displayed");
        }
        return wait;
```

```
    }

    private String getLocatorString(String locatorString, String[] ss) {
        for (String s : ss) {
            locatorString = locatorString.replaceFirst("%s", s);
        }
        return locatorString;
    }

    private WebElement getLocator(String key, String[] replace, boolean wait)
{
        WebElement element = null;
        if (ml.containsKey(key)) {
            Map<String, String> m = ml.get(key);
            String type = m.get("type");
            String value = m.get("value");
            if (replace != null)
                value = this.getLocatorString(value, replace);
            By by = this.getBy(type, value);
            if (wait) {
                element = this.watiForElement(by);
                boolean flag = this.waitElementToBeDisplayed(element);
                if (!flag)
                    element = null;
            } else {
                try {
                    element = driver.findElement(by);
                } catch (Exception e) {
                    element = null;
                }
            }
        } else
            System.out.println("Locator " + key + " is not exist in " + yamlFile
                    + ".yaml");
        return element;
    }

    public WebElement getElement(String key) {
        return this.getLocator(key, null, true);
    }

    public WebElement getElementNoWait(String key) {
        return this.getLocator(key, null, false);
    }
```

```
    public WebElement getElement(String key, String[] replace) {
        return this.getLocator(key, replace, true);
    }

    public WebElement getElementNoWait(String key, String[] replace) {
        return this.getLocator(key, replace, false);
    }

    public void setLocatorVariableValue(String variable, String value){
    if(ml!=null){
        Set<String> keys = ml.keySet();
            for(String key:keys){
                String v =
ml.get(key).get("value").replaceAll("%"+variable+"%", value);
                ml.get(key).put("value",v);
            }
    }
    }

    @SuppressWarnings("unchecked")
    public void loadExtendLocator(String fileName){
        File f = new File("locator/" + fileName + ".yaml");
        try {
            extendLocator = Yaml.loadType(new
FileInputStream(f.getAbsolutePath()),
                    HashMap.class);
            ml.putAll(extendLocator);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }


    public static void main(String[] args){
    SeleniumDriver selenium = new SeleniumDriver();
    Locator d = new Locator(selenium.getDriver());
        WebElement element = d.getElement("百度输入框");
        element.sendKeys("");
    }

}
```

第十一周：搭建框架二（数据驱动）

数据驱动的原理
全局数据，局部数据，私有数据的扩展
数据驱动实例解析
上面的元素管理的代码完成了，现在我们来看数据管理，以前就讲过数据驱动，用 dataprovider 的注脚来完成，但是，我们把数据都是写在了 java 文件中，如何把数据用外部文件来保存？这里我们以 XML 文件为例，我们以一个测试类为一个区分点，一个测试类中能包含多个测试方法，所以，我们的 XML 文件为了能与测试类联系起来，让测试类在运行的时候，自动的去找到对应的 XML 文件及在 XML 中想要的数据？为了实现这个，我们来做点约定，就是 XML 文件的文件名与测试类的类名相同，然后 XML 文件中的结点名与测试类的方法名相同，这样一来，就能找到对应的 XML 及对应的数据了，我们先来构建一个 xml,(以 Test1.xml, 方法名为 testLogin 来为例)：

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
    <testLogin>
        <username>test1</username>
        <password>123456</password>
    </testLogin>
    <testLogin>
        <username>test2</username>
        <password>123456</password>
    </testLogin>
</data>
```

我们之间有写过解析 XML 的方法，但是 `getElementObject` 是针对 single 结点的，我们上面那个 testLogin 结点有两个，所以，得在 `ParseXml` 类里加一个方法：

```
@SuppressWarnings("unchecked")
    public List<Element> getElementObjects(String elementPath) {
        return document.selectNodes(elementPath);
    }
```

这样，得出来的就是这个结点的集合了，但是要把 testLogin 下面的子结点的信息再全部取到：

```
@SuppressWarnings("unchecked")
    public Map<String, String> getChildrenInfoByElement(Element element){
        Map<String, String> map = new HashMap<String, String>();
        List<Element> children = element.elements();
        for (Element e : children) {
            map.put(e.getName(), e.getText());
        }
        return map;
    }
```

把信息取到后，再提供给 dataprovider 注脚的方法里运回给测试方法就行了：

```
public class TestBase {

    @DataProvider
    public Object[][] providerMethod(Method method){
```

```
        ParseXml px = new
ParseXml("test-data/"+this.getClass().getSimpleName()+".xml");
        String methodName = method.getName();
        List<Element> elements = px.getElementObjects("/*/"+methodName);
        Object[][] object = new Object[elements.size()][];
        for (int i =0; i<elements.size(); i++) {
            Object[] temp = new
Object[]{px.getChildrenInfoByElement(elements.get(i))};
            object[i] = temp;
        }
        return object;
    }

}
```

这样就返回出来了数据，我们来做个试验：

```
public class Test1 extends TestBase{

    @Test(dataProvider="providerMethod")
    public void testLogin(Map<String, String> param){
        System.out.println(param.get("username"));
        System.out.println(param.get("password"));
    }

}
```

运行后，我们来看一下运行的结果：

```
test1
123456
test2
123456
PASSED: testLogin({username=test1, password=123456})
PASSED: testLogin({username=test2, password=123456})

===============================================
    Default test
    Tests run: 2, Failures: 0, Skips: 0
```

只有一个测试方法，配置了两个 testLogin 的结点，就运行两次，每次运行对应的数据就出来了！

在一个 XML 文件里面，有很多个测试方法的结点，这些结点中的数据也有很多，但有可能这些结点中的数据有些可能是重复的，比如在这个测试类中的所有测试方法，都需要在一个输入框里输入同一个值，那么在每个测试方法的结点中都写一遍这个输入框<inputValue>test</inputValue>，如果这样的话，那脚本哪天想这个输入框都输入 test1，那岂不是每个结点里改一遍？这样不方便，于是，可以把这些个公共的值给抽像出来，放在 common 结点下面：

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
```

```
    <common>
        <inputValue>test</inputValue>
    </common>
    <testLogin>
        <username>test1</username>
        <password>123456</password>
    </testLogin>
    <testLogin>
        <username>test2</username>
        <password>123456</password>
    </testLogin>
</data>
```

这样一来，我们还有个原则要遵守一下，就是 common 节点下面的子结点如果与测试方法中的子结点相同，那么就以测试方法中的子结点的值为准，于是我们要改造一个 providerMethod 方法，在改造前，我们得写一个方法，把 common 与测试方法的子节点进行合并起来：

```
    private Map<String, String> getMergeMapData(Map<String, String> map1,
Map<String, String> map2){
        Iterator<String> it = map2.keySet().iterator();
        while(it.hasNext()){
            String key = it.next();
            String value = map2.get(key);
            if(!map1.containsKey(key)){
                map1.put(key, value);
            }
        }
        return map1;
    }
```

然后再加两个初始化的方法：

```
private ParseXml px;

    private Map<String, String> commonMap;

    private void initialPx(){
        if(px==null){
            px = new
ParseXml("test-data/"+this.getClass().getSimpleName()+".xml");
        }
    }

    private void getCommonMap(){
        if(commonMap==null){
            Element element = px.getElementObject("/*/common");
            commonMap = px.getChildrenInfoByElement(element);
        }
```

```
    }
```
这两个方法，都用到了单例模式，在调用时，为 null 就 new 一个对象，如果不为 null,那就不产生新的对象，我们来看 providerMethod 方法：
```
@DataProvider
    public Object[][] providerMethod(Method method){
        this.initialPx();
        this.getCommonMap();
        String methodName = method.getName();
        List<Element> elements = px.getElementObjects("/*/"+methodName);
        Object[][] object = new Object[elements.size()][];
        for (int i =0; i<elements.size(); i++) {          ;
            Object[] temp = new
Object[]{this.getMergeMapData(px.getChildrenInfoByElement(elements.get(i)),
commonMap)};
            object[i] = temp;
        }
        return object;
    }
```
我们再看一下测试脚本：
```
public class Test1 extends TestBase{

    @Test(dataProvider="providerMethod")
    public void testLogin(Map<String, String> param){
        System.out.println(param.get("username"));
        System.out.println(param.get("password"));
        System.out.println(param.get("inputValue"));
    }

}
```
运行一下观察下结果吧！

既然有 common 了，那对于所有的测试类的公共的结点，比如一个 URL，所有的测试类都要用到，那在每一个 XML 的 common 都写一遍？这样也太麻烦了，于是乎，我们定义一个 global 的 xml，让这个里面的数据也如 common 一样加载到最后返回给测试方法的数据中去，同理，优先级是测试方法数据>common>global,

我们先定义一个 global.xml:
```
<?xml version="1.0" encoding="UTF-8"?>
<data>
    <url>http://www.baidu.com</url>
</data>
```
由于这个 global 的是针对所有测试脚本的，所以在一开始就要加载，依照 config.xml 的方式，我们来加一个 Global 的静态类：
```
public class Global {

    public static Map<String, String> global;
```

```
    static{
        ParseXml px = new ParseXml("test-data/global.xml");
        global = px.getChildrenInfoByElement(px.getElementObject("/*"));
    }

}
```

同样的改造下 providerMethod 方法：

```
@DataProvider
    public Object[][] providerMethod(Method method){
        this.initialPx();
        this.getCommonMap();
        String methodName = method.getName();
        List<Element> elements = px.getElementObjects("/*/"+methodName);
        Object[][] object = new Object[elements.size()][];
        for (int i =0; i<elements.size(); i++) {
            Map<String, String> mergeCommon =
this.getMergeMapData(px.getChildrenInfoByElement(elements.get(i)),
commonMap);
           Map<String, String> mergeGlobal = this.getMergeMapData(mergeCommon,
Global.global);
            Object[] temp = new Object[]{mergeGlobal};
            object[i] = temp;
        }
        return object;
    }
```

以上就是数据驱动的所有内容，里面包含了几个知识点：解析 XML，单子模式，合并数据，数组的初始化，私有数据，局部数据，全局数据的思想，请大家仔细的体会下。

第十二周：搭建框架三（框架中要用到的常用类）

操作数据库（数据库管理 mysql）
读取 EXCEL
Date 类的应用
随机数的生成

操作数据库，请先安装好 mysql,并建好库，建好表（user）

```
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `test_name` varchar(64) NOT NULL,
  `test_age` varchar(64) NOT NULL,
  `test_height` varchar(32) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

可以用上述 SQL 语句直接建好表。

既然谈到面向对象，所以，先把连接信息给搞个对象出来：

```
public class DBInfo {

	private String driver;

	private String host;

	private String port;

	private String user;

	private String pwd;

	private String dataBase;

	public DBInfo(){
		this.driver = "com.mysql.jdbc.Driver";
		this.host = "";
		this.port = "";
		this.user = "";
		this.pwd = "";
		this.dataBase = "";
	}

	public String getDriver() {
		return driver;
	}

	public void setDriver(String driver) {
		this.driver = driver;
	}

	public String getHost() {
		return host;
	}

	public void setHost(String host) {
		this.host = host;
	}

	public String getDataBase() {
		return dataBase;
	}
```

```
    public void setDataBase(String dataBase) {
        this.dataBase = dataBase;
    }

    public String getPort() {
        return port;
    }

    public void setPort(String port) {
        this.port = port;
    }

    public String getUser() {
        return user;
    }

    public void setUser(String user) {
        this.user = user;
    }

    public String getPwd() {
        return pwd;
    }

    public void setPwd(String pwd) {
        this.pwd = pwd;
    }

}
```

既然是操作数据库，我们就把数据库的字段给对象化一下，也就是持久化：在定义变量时，我们搞个约定，比如，数据库字段名为:test_login_name,则定义变量时为：testLoginName.

```
public class UserInfo {

    private int id;

    private String testName;

    private String testAge;

    private String testHeight;

    public int getId() {
        return id;
    }
```

```
    public void setId(int id) {
        this.id = id;
    }

    public String getTestName() {
        return testName;
    }

    public void setTestName(String testName) {
        this.testName = testName;
    }

    public String getTestAge() {
        return testAge;
    }

    public void setTestAge(String testAge) {
        this.testAge = testAge;
    }

    public String getTestHeight() {
        return testHeight;
    }

    public void setTestHeight(String testHeight) {
        this.testHeight = testHeight;
    }

}
```

好，现在有了 javabean,有了数据库连接的对象了，再加一个枚举来保存数据库与 bean 之间的映射关系：

```
public enum TableBean {

    USER_INFO("com.test.bean.UserInfo");

    private String value;

    private TableBean(String value){
        this.value = value;
    }

    public String getValue(){
        return value;
    }
```

```
    @Override
    public String toString() {
            return value;
    }

    public static void main(String[] args){
        System.out.println(TableBean.USER_INFO);
    }

}
```

再加一个保存 ResultSetMetaData 信息的类:

```
public class MetaData {

    public static Map<String, ResultSetMetaData> metaData = new HashMap<String,
ResultSetMetaData>();

}
```

余下就是操作数据库了：

```
public class ConnectToDB {

    private DBInfo dbInfo;

    private Connection conn = null;

    private Statement stmt = null;

    public ConnectToDB(){
        dbInfo = new DBInfo();
    }

    public DBInfo getDbInfo() {
        return dbInfo;
    }

    public void setDbInfo(DBInfo dbInfo) {
        this.dbInfo = dbInfo;
    }

    public void connect() {
        this.close();
        this.connectMySQL();
    }

    public synchronized void close() {
```

```
        try {
            if (stmt != null) {
                stmt.close();
                stmt = null;
            }
            if (conn != null) {
                conn.close();
                conn = null;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private synchronized void connectMySQL() {
        try {
            Class.forName(dbInfo.getDriver()).newInstance();
            conn = (Connection) DriverManager.getConnection("jdbc:mysql://"
                    + dbInfo.getHost() + "/" + dbInfo.getDataBase()
+"?useUnicode=true&characterEncoding=utf-8",
dbInfo.getUser(),dbInfo.getPwd());
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }

    }

    private void statement() {
        if (conn == null) {
            this.connectMySQL();
        }
        try {
            stmt = (Statement) conn.createStatement();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

```
    private ResultSet resultSet(String sql) {
        ResultSet rs = null;
        if (stmt == null) {
            this.statement();
        }
        try {
            rs = stmt.executeQuery(sql);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return rs;
    }

    private void executeUpdate(String sql){
        if (stmt == null) {
            this.statement();
        }
        try {
            stmt.executeUpdate(sql);
        } catch (SQLException e) {
            System.out.println(sql);
            e.printStackTrace();
        }
    }

    public List<Object> query(String tableInfo, String sql) {
        List<Object> list = new ArrayList<Object>();
        ResultSet rs = this.resultSet(sql);
        try {
            ResultSetMetaData md = rs.getMetaData();
            int cc = md.getColumnCount();
            while (rs.next()) {
                Object object = this.getBeanInfo(tableInfo);
                for (int i = 1; i <= cc; i++) {
                    String cn = md.getColumnName(i);
                    this.reflectSetInfo(object,
this.changeColumnToBean(cn,"set"), rs.getObject(cn));
                }
                list.add(object);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return list;
```

```
    }

    public void insert(String table, Object object){
        String sql = "";
        try {
            this.getMetaData(table);
            ResultSetMetaData md = MetaData.metaData.get(table);
            int cc = md.getColumnCount();
            String insertColumn = "";
            String insertValue = "";
            for (int i = 2; i <= cc; i++) {
                String cn = md.getColumnName(i);
                Object gValue = this.reflectGetInfo(object,
this.changeColumnToBean(cn,"get"));
                if(gValue.getClass().getSimpleName().equals("String")){
                    gValue = "\""+gValue+"\"";
                }
                if("".equals(insertColumn)){
                    insertColumn += cn;
                    insertValue += gValue;
                }else{
                    insertColumn += ","+cn;
                    insertValue += ","+gValue;
                }
            }
            sql = "insert into "+table+" ("+insertColumn+") values
("+insertValue+")";
            this.executeUpdate(sql);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private void getMetaData(String table){
        if(!MetaData.metaData.containsKey(table)){
            ResultSet rs = this.resultSet("select * from "+table+" limit 0,1");
            try {
                MetaData.metaData.put(table, rs.getMetaData());
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
```

```
    private Object getBeanInfo(String tableInfo){
        Object object = null;
        try {
            object = Class.forName(tableInfo).newInstance();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        return object;
    }

    private void reflectSetInfo(Object object, String methodName, Object
parameter){
        try {
            Class<? extends Object> ptype = parameter.getClass();
            if(parameter.getClass().getSimpleName().equals("Integer")){
                ptype = int.class;
            }
            Method method = object.getClass().getMethod(methodName, ptype);

            method.invoke(object, parameter);
        } catch (SecurityException e) {
            e.printStackTrace();
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalArgumentException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }

    private Object reflectGetInfo(Object object, String methodName){
        Object value = null;
        try {
            Method method = object.getClass().getMethod(methodName);

            Object returnValue = method.invoke(object);
            if(returnValue!=null){
```

```
                value = returnValue.toString();
            }else{
                value = "";
            }
        } catch (SecurityException e) {
            e.printStackTrace();
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalArgumentException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return value;
    }

    private String columnToBean(String column){
        if(column.contains("_")){
            int index = column.indexOf("_");
            String beanName = column.substring(0, index)
                            +column.substring(index+1,
index+2).toUpperCase()
                            +column.substring(index+2, column.length());

            return beanName;
        }
        return column;
    }

    private String changeColumnToBean(String column, String ext){
        String[] col = column.split("_");
        for (int i = 0; i < col.length; i++) {
            column = this.columnToBean(column);
        }
        column =column.replaceFirst(column.substring(0, 1),
column.substring(0, 1).toUpperCase());
        column = ext+column;
        return column;
    }

    public static void main(String[] args) throws SQLException {
        ConnectToDB c = new ConnectToDB();
```

```
        c.connect();
        List<Object> list = c.query(TableBean.USER_INFO.toString(), "select *
from user");
        UserInfo ui = (UserInfo)list.get(0);
        System.out.println(ui.getTestAge());
        c.insert("user", ui);
        c.close();
    }
}
```

仔细看看吧，query 出来就是对象的集合，insert 时，就是表名与对象就行了，至于 update 与 delete，大家自己扩展吧！如果把这个摸清楚，spring 操作 mysql 数据库的原理，你也就差不多了！

读取 EXCEL

```
public class ExcelReader {
    private String filePath;
    private String sheetName;
    private Workbook workBook;
    private Sheet sheet;
    private List<String> columnHeaderList;
    private List<List<String>> listData;
    private List<Map<String,String>> mapData;
    private boolean flag;

    /**
     * 所需jar包：
poi-3.8.jar,poi-ooxml.jar,poi-ooxml-schemas.jar,xmlbeans.jar
     * 提供解析excel，兼容excel2003及2007+版本
     * @param filePath excel本地路径
     * @param sheetName excel的sheet名称
     */
    public ExcelReader(String filePath, String sheetName) {
        this.filePath = filePath;
        this.sheetName = sheetName;
        this.flag = false;
        this.load();
    }

    /**
     * 加载EXCEL文件内容，产生WorkBook对象，再产生Sheet对象
     */
    private void load() {
        FileInputStream inStream = null;
        try {
            inStream = new FileInputStream(new File(filePath));
            workBook = WorkbookFactory.create(inStream);
```

```
            sheet = workBook.getSheet(sheetName);
        } catch (Exception e) {
            e.printStackTrace();
        }finally{
            try {
                if(inStream!=null){
                    inStream.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    /**
     * 根据cell对象，来取得每个cell的值，所有的值的数据类型都转化为了String类型
     * @param cell Cell对象
     * @return
     */
    private String getCellValue(Cell cell) {
        String cellValue = "";
        DataFormatter formatter = new DataFormatter();
        if (cell != null) {
            switch (cell.getCellType()) {
                case Cell.CELL_TYPE_NUMERIC:
                    if (DateUtil.isCellDateFormatted(cell)) {
                        cellValue = formatter.formatCellValue(cell);
                    } else {
                        double value = cell.getNumericCellValue();
                        int intValue = (int) value;
                        cellValue = value - intValue == 0 ?
String.valueOf(intValue) : String.valueOf(value);
                    }
                    break;
                case Cell.CELL_TYPE_STRING:
                    cellValue = cell.getStringCellValue();
                    break;
                case Cell.CELL_TYPE_BOOLEAN:
                    cellValue = String.valueOf(cell.getBooleanCellValue());
                    break;
                case Cell.CELL_TYPE_FORMULA:
                    cellValue = String.valueOf(cell.getCellFormula());
                    break;
                case Cell.CELL_TYPE_BLANK:
```

```
                    cellValue = "";
                    break;
                case Cell.CELL_TYPE_ERROR:
                    cellValue = "";
                    break;
                default:
                    cellValue = cell.toString().trim();
                    break;
            }
        }
        return cellValue.trim();
    }

    /**
     * 取得sheet的data,listData是一行一个list,这个list里面放该行的所有列的值
     * mapData是一行一个list,这个list里面存放的是map,map的key是第一列的header值。
     */
    private void getSheetData() {
        listData = new ArrayList<List<String>>();
        mapData = new ArrayList<Map<String, String>>();
        columnHeaderList = new ArrayList<String>();
        int numOfRows = sheet.getLastRowNum() + 1;
        for (int i = 0; i < numOfRows; i++) {
            Row row = sheet.getRow(i);
            Map<String, String> map = new HashMap<String, String>();
            List<String> list = new ArrayList<String>();
            if (row != null) {
                for (int j = 0; j < row.getLastCellNum(); j++) {
                    Cell cell = row.getCell(j);
                    if (i == 0){
                        columnHeaderList.add(getCellValue(cell));
                    }
                    else{
                        map.put(columnHeaderList.get(j),
this.getCellValue(cell));
                    }
                    list.add(this.getCellValue(cell));
                }
            }
            if (i > 0){
                mapData.add(map);
            }
            listData.add(list);
        }
```

```
            flag = true;
        }

        /**
         * 根据行与列的index来得到相应的cell的值
         * @param row 从1开始
         * @param col 从1开始
         * @return
         */
        public String getCellData(int row, int col){
            if(row<=0 || col<=0){
                return null;
            }
            if(!flag){
                this.getSheetData();
            }
            if(listData.size()>=row && listData.get(row-1).size()>=col){
                return listData.get(row-1).get(col-1);
            }else{
                return null;
            }
        }

        /**
         * 根据行数及第一列的列名，取得相应的cell的值
         * @param row 从1开始
         * @param headerName 第一列的列名
         * @return
         */
        public String getCellData(int row, String headerName){
            if(row<=0){
                return null;
            }
            if(!flag){
                this.getSheetData();
            }
            if(mapData.size()>=row &&
    mapData.get(row-1).containsKey(headerName)){
                return mapData.get(row-1).get(headerName);
            }else{
                return null;
            }
        }
```

```
    public static void main(String[] args) {
        ExcelReader eh = new ExcelReader("E:\\workspace\\test.xls","Sheet1");
        System.out.println(eh.getCellData(1, 1));
        System.out.println(eh.getCellData(1, "test1"));
    }
}
```

Date 类的应用

```
public class TimeString {

    private String valueOfString(String str, int len) {
        String string = "";
        for (int i = 0; i < len - str.length(); i++) {
            string = string + "0";
        }
        return (str.length() == len) ? (str) : (string + str);
    }
    /**
     * 返回当前时间，格式为：2014-12-18 15:11:50
     * @return
     */
    public String getSimpleDateFormat(){
        SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        return df.format(new Date());
    }

    /**
     * 返回当前时间戳
     * @return
     */
    public String getTime(){
        return String.valueOf(new Date().getTime());
    }

    /**
     * 生成一个长度为17的时间字符串，精确到毫秒
     * @return
     */
    public String getTimeString() {
        Calendar calendar = new GregorianCalendar();
        String year = String.valueOf(calendar.get(Calendar.YEAR));
        String month =
this.valueOfString(String.valueOf(calendar.get(Calendar.MONTH) + 1),2);
```

```
        String day =
this.valueOfString(String.valueOf(calendar.get(Calendar.DAY_OF_MONTH)),2);
        String hour =
this.valueOfString(String.valueOf(calendar.get(Calendar.HOUR_OF_DAY)),2);
        String minute =
this.valueOfString(String.valueOf(calendar.get(Calendar.MINUTE)),2);
        String second =
this.valueOfString(String.valueOf(calendar.get(Calendar.SECOND)),2);
        String millisecond =
this.valueOfString(String.valueOf(calendar.get(Calendar.MILLISECOND)),3);
        return year+month+day+hour+minute+second+millisecond;
    }

    public static void main(String[] args) {
        TimeString ts = new TimeString();
        System.out.println(ts.getTime());
        System.out.println(ts.getSimpleDateFormat());
        System.out.println(ts.getTimeString());
    }
}
```

随机数的生成

```
public class RandomUtil {

    /**
     * 返回一个0-count（包含count）的随机数
     * @param count
     * @return
     */
    public int getRandom(int count) {
        return (int) Math.round(Math.random() * (count));
    }

    private String string = "0123456789abcdefghijklmnopqrstuvwxyz";

    /**
     * 从0123456789abcdefghijklmnopqrstuvwxyz中选随机生成长度为length的字符串
     * @param length
     * @return
     */
    public String getRandomString(int length){
        StringBuffer sb = new StringBuffer();
        int len = string.length();
        for (int i = 0; i < length; i++) {
            sb.append(string.charAt(this.getRandom(len-1)));
```

```
        }
        return sb.toString();
    }

    public static void main(String[] args) {
        RandomUtil ru = new RandomUtil();
        for (int i = 0; i < 10; i++) {
            System.out.println(ru.getRandomString(6));
        }

    }
}
```

第十三周：搭建框架四（整合框架）

在元素管理与数据驱动基础上，加上 LOG4J，配置管理，形成一个框架
框架实例演示与代码讲解
框架的主要特点
框架如何维护
CI 每日构建的集成策略及思想

经过上面的元素管理，数据驱动，以及一些基础的类的封装，我们的框架也越来越接近完成了，现在再来修理下边角，加上 LOG，之前的代码中有，加上即可。于是，我们把整个工程中的 System.out.println 全部给替换成 Log。替换之后，我们还差把 page 与 locator 联系起来，因为之前一直只有一个 Locator 类，所以我们加一个 Page 类：

```
public class Page extends Locator{

    private WebDriver driver;

    public Page(WebDriver driver) {
        super(driver);
        this.driver = driver;
    }

}
```

之前在 Locator 类中，yamlFile = "demo";这个是固定的为”demo”，这是不行的，在这里，我把也搞一个约定，就是一个 Page 类，对应一个 yaml 文件时，名称一样，比如 LoginPage 那么 yaml 文件也为 LoginPage.yaml，这样就可以自动的找到对应的 yaml 文件了，修改如下：先在 Locator 类里加一个这方法，然后在 Locator 类里删除 yamlFile = "demo";这一句：

```
    public void setYamlFile(String yamlFile) {
        this.yamlFile = yamlFile;
    }
```

同时把 Locator 类的 protected WebDriver driver;由 private 改为 protected,这样这个 driver 对象在 Page 中就可以直接使用了：

```
public class Page extends Locator{

    public Page(WebDriver driver) {
        super(driver);
        this.setYamlFile(this.getClass().getSimpleName());
        this.getYamlFile();
    }

    /**
     * 该方法为示例方法
     */
    public void test(){
        driver.navigate().to("");
    }

}
```

把 getYamlFile 的方法的调用放到了 Page 类中了，这样是为了先 setYamlFile 后，使 yaml 的路径有了后再调用 getYamlFile 方法，这样不至于为 null，同时，去掉在 Locator 的构造器中调用的 getYamlFile 方法。

现在我们还有一个问题，就是需要把 driver 给初始化一下，我们在这定的策略是一个测试类对应一个 driver，那么根据 testng 的特性，我们可以把初始化 driver 的动作放在@BeforeClass 中去，在 TestBase 类中添加一个方法：

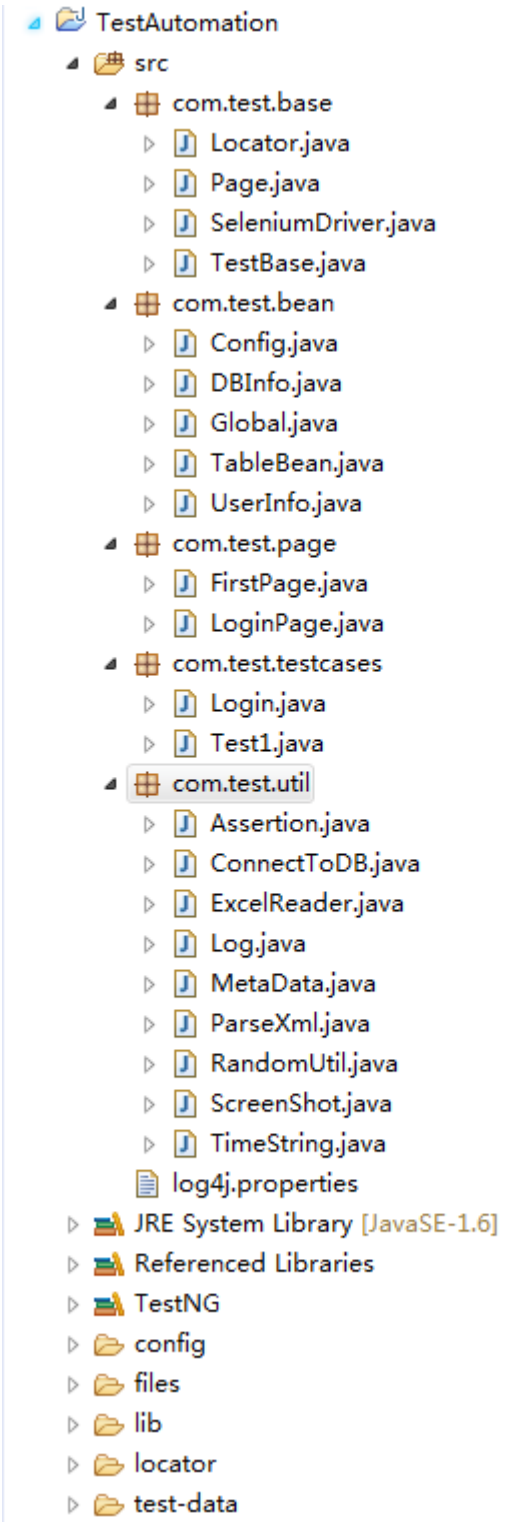
```
protected WebDriver driver;

    @BeforeClass
    public void initialDriver(){
        SeleniumDriver selenium = new SeleniumDriver();
        driver = selenium.getDriver();
    }

    @AfterClass
    public void closeDriver(){
        if(driver!=null){
            driver.close();
            driver.quit();
        }
    }
```

由于 LOG4J 只是提供了 LOG 输出在控制台与本地文件中，但有一个问题是，如何把 LOG 输出到报告中去？TESTNG 提供了一个 API：Reporter.log()，我们可以利用这个 API，在 Log 类里面进行扩展，如下：

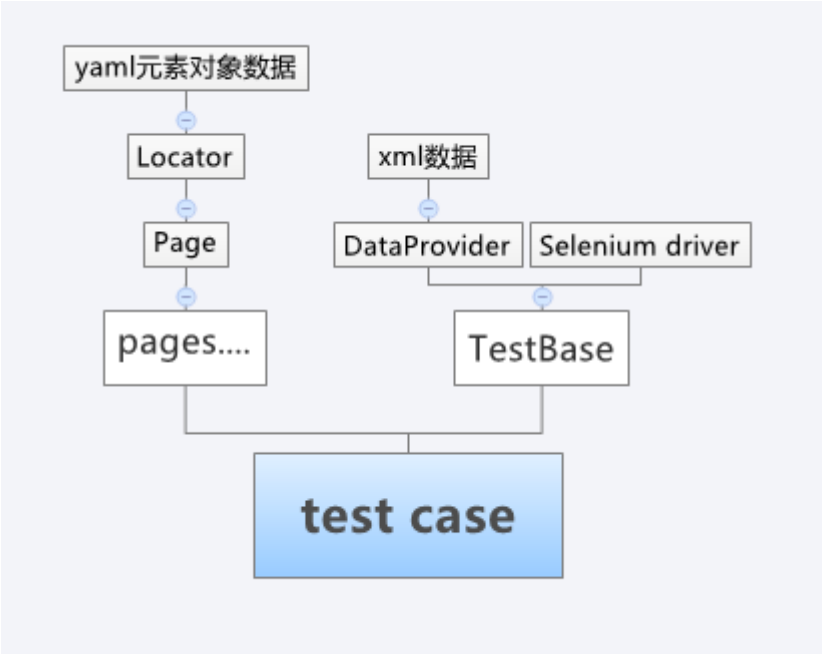
```
    public static void logInfo(Object message) {
        Log.getFlag();
        logger.info(message);
        Reporter.log(new TimeString().getSimpleDateFormat()+" : "+message);
    }
```

这样输出的 LOG 在报告中也能看到了，还有我们之前写的 Assertion 类及 ScreenShot 类加上，

至此，我们的框架基本上就完成了，我们来整体的看一下代码的结构图：



大概可以分为这样：



下面我们来进行一下实战。（我们以京东为例吧。）

场景一如下：

1. 打开页面：http://www.jd.com/
2. 点击登录
3. 输入用户名与密码
4. 点击登录
5. 验证是否登录成功。

我们来看实现步骤：

1. 先在 global.xml 中增加一个 url 结点，值为 http://www.jd.com
2. 建一个 Login 的测试类，并加上测试方法

```
public class Login extends TestBase{

    @Test(dataProvider="providerMethod")
    public void testLogin(Map<String, String> param){

    }
   }
```

3. 建一个 Login.xml 的数据文件

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
    <common>
    </common>
    <testLogin>
    </testLogin>
   </data>
```

4. 这些建好后，我们来建 yaml 文件，也就是元素对象。

FirstPage.yaml

```
login_link:
  type:xpath
  value:"//li[@id='loginbar']//a[text()='[请登录]']"
```

LoginPage.yaml

```
login_name:
  type:id
  value:loginname
login_pwd:
  type:id
  value:nloginpwd
login_button:
  type:id
  value:loginsubmit
```

5. 相对应的，我们得建出 FirstPage.java 与 LoginPage.java 这两个 page 类：

```
public class FirstPage extends Page{

    public FirstPage(WebDriver driver) {
        super(driver);
    }

}
```

LoginPage 可以仿这个写出来。

6. login_name 与 login_pwd 里面要输入值，于是，得在 Login.xml 里面加上数据节点：

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
    <common>
    </common>
    <testLogin>
        <username>test1</username>
        <password>123456</password>
    </testLogin>
</data>
```

于是，脚本可以如下：

```
public class Login extends TestBase{

    @Test(dataProvider="providerMethod")
    public void testLogin(Map<String, String> param){
        Assertion.flag = true;
        driver.manage().window().maximize();
        driver.manage().timeouts().pageLoadTimeout(Config.waitTime,
TimeUnit.SECONDS);
        driver.get(param.get("url"));
```

```
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        FirstPage fp = new FirstPage(driver);
        Log.logInfo("在首页点击登录按钮");
        fp.getElement("login_link").click();
        LoginPage lp = new LoginPage(driver);
        Log.logInfo("登录用户名为:"+param.get("username"));
        lp.getElement("login_name").sendKeys(param.get("username"));
        Log.logInfo("登录密码为:"+param.get("password"));
        lp.getElement("login_pwd").sendKeys(param.get("password"));
        lp.getElement("login_button").click();
        String errorMsg = lp.getElement("loginpwd_error").getText();
        Assert.assertEquals(errorMsg, "您输入的账户名和密码不匹配，请重新输入");
    }
}
```

这个加 Thread.sleep 是指硬等待 1 秒，一般不用加，加在这是因为 chrome 运行太快，如果不加的话，就会报错，于是，我们可以把 Thread.sleep 放在一个单独的类里面：

```
public class Util {

    public static void sleep(int secs){
        try {
            Thread.sleep(secs*1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

}
```

然后把关于 driver 的启动配置，可以放在 SeleniumDriver 类中去：

```
    private void initialDriver(){
        if("firefox".equals(Config.browser)){
            driver = new FirefoxDriver();
        }else if("ie".equals(Config.browser)){
            System.setProperty("webdriver.ie.driver", "files/iedriver.exe");
            DesiredCapabilities capabilities =
DesiredCapabilities.internetExplorer();

capabilities.setCapability(InternetExplorerDriver.INTRODUCE_FLAKINESS_BY_IG
NORING_SECURITY_DOMAINS, true);
           capabilities.setCapability("ignoreProtectedModeSettings",true);
            driver = new InternetExplorerDriver(capabilities);
```

```
        }else if("chrome".equals(Config.browser)){
            System.setProperty("webdriver.chrome.driver",
"files/chromedriver.exe");
            ChromeOptions options = new ChromeOptions();
            options.addArguments("--test-type");
            driver = new ChromeDriver(options);
        }else{
            Log.logInfo(Config.browser+" 的值不正确，请检查！");
        }
        driver.manage().window().maximize();
        driver.manage().timeouts().pageLoadTimeout(Config.waitTime,
TimeUnit.SECONDS);
    }
```

然后把 driver.get 封装在 TestBase 中去：

```
    public void goTo(String url){
        driver.get(url);
        if(Config.browser.equals("chrome")){
            Util.sleep(1);
        }
    }
```

最后脚本就可以变成为：

```
public class Login extends TestBase{
    @Test(dataProvider="providerMethod")
    public void testLogin(Map<String, String> param){
        Assertion.flag = true;
        this.goTo(param.get("url"));
        FirstPage fp = new FirstPage(driver);
        Log.logInfo("在首页点击登录按钮");
        fp.getElement("login_link").click();
        LoginPage lp = new LoginPage(driver);
        Log.logInfo("登录用户名为:"+param.get("username"));
        lp.getElement("login_name").sendKeys(param.get("username"));
        Log.logInfo("登录密码为:"+param.get("password"));
        lp.getElement("login_pwd").sendKeys(param.get("password"));
        lp.getElement("login_button").click();
        String errorMsg = lp.getElement("loginpwd_error").getText();
        Assert.assertEquals(errorMsg, "您输入的账户名和密码不匹配，请重新输入");
    }
}
```

这种脚本的可读性如何？

场景二如下：

1. 打开页面：http://www.jd.com/
2. 在全部商品分类下面选择”手机,数码,京东通信”

3. 选择手机
4. 在商品筛选中选择华为
5. 价格区间选择 800-1299
6. 机身选择黑色
7. 商品类型选择京东配送
8. 点击第一个搜索出来的结果，进入单品页,并记录下商品名称，商品价格
9. 点击加入购物车
10. 点击去购物车结算
11. 在购物车中验证：

    商品名称

    商品价格

    是否被勾选
12. 点击删除，确定
13. 验证购物车是否为空或者购物车中是否还有刚选中的商品

在这个场景中，我们先把页面给定下来：

FirstPage.yaml

```
login_link:
  type:css
  value:"#loginbar a:nth-of-type(1)"
手机数码京东通信:
  type:xpath
  value:"//div[@id='_JD_ALLSORT']//span//a[@href='http://shouji.jd.com/']"
  #"//div[@id='_JD_ALLSORT']//span//a[@href='http://shouji.jd.com/']"
手机品类入口:
  type:xpath
  value:"//div[@id='_JD_ALLSORT']//a[@href='http://list.jd.com/9987-653-655.html']"
```

MobileList.yaml

```
商品筛选华为:
  type:xpath
  value:"//div[@class='tabcon show-logo']//a[@title='华为（HUAWEI）']"
商品价格筛选:
  type:xpath
  value:"//div[@class='a-values']//a[text()='800-1299']"
商品颜色筛选:
  type:xpath
  value:"//div[@class='a-values']//a[text()='黑色']"
商品类型筛选:
  type:xpath
  value:"//div[@id='filter']//a[contains(text(),'京东配送')]"
第一个商品:
  type:xpath
  value:"//div[@id='plist']//li[1]//img"
商品名称:
  type:xpath
  value:"//div[@id='plist']//li[1]//div[@class='p-name']/a"
商品价格:
  type:xpath
  value:"//div[@id='plist']//li[1]//div[@class='p-price']/strong"
```

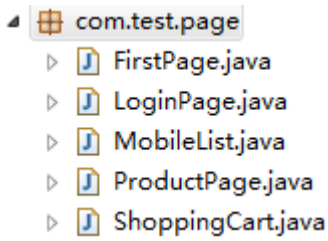
ProductPage.yaml

```
商品名称:
  type:id
  value:name
商品价格:
  type:id
  value:jd-price
加入购物车:
  type:id
  value:InitCartUrl
去购物车结算:
  type:id
  value:GotoShoppingCart
```

ShoppingCart.yaml

```
商品名称:
  type:xpath
  value:"//div[@class='cell p-quantity' and @for-stock='for-stock-%s']/parent::div//div[@class='p-name']/a"
商品价格:
  type:xpath
  value:"//div[@class='cell p-quantity' and @for-stock='for-stock-%s']/parent::div//span[@class='price']"
勾选商品:
  type:xpath
  value:"//div[@class='cell p-quantity' and @for-stock='for-stock-%s']/parent::div//input[@class='checkbox']"
删除商品:
  type:xpath
  value:"//div[@class='cell p-quantity' and @for-stock='for-stock-%s']/parent::div//a[@class='cart-remove']"
确定删除:
  type:id
  value:btnRemoveConfirm
```

然后把 page 类建立好。



在 testcase 中新建一个 Shopping 的类以及一个 Shopping.xml,然后就可以开始写脚本了，我们一步步的来实现：

在 FirstPage 中封装两步，就是 moserover 到手机上，然后点击手机的链接：

```
/**
     * 因为chromedriver2.13版本的在movetoelement上有BUG，在2.14版中会解决，所以这个脚本只在FIREFOX上运行
     */
    public void linkToMobileList(){
        this.getAction().moveToElement(this.getElement("手机数码京东通信")).perform();
        this.getElement("手机品类入口").click();
    }
```

脚本现在如下：

```
@Test(dataProvider="providerMethod")
    public void testShopping(Map<String, String> param){
        Assertion.flag = true;
        this.goTo(param.get("url"));
```

```
        FirstPage fp = new FirstPage(driver);
        Log.logInfo("从首页进入手机搜索页面");
        fp.linkToMobileList();

    }
在商品筛选中选择华为
        MobileList ml = new MobileList(driver);
        ml.getElement("商品筛选华为").click();
价格区间选择 800-1299
ml.getElement("商品价格筛选").click();
机身选择黑色
ml.getElement("商品颜色筛选").click();
商品类型选择京东配送
ml.getElement("商品类型筛选").click();
点击第一个搜索出来的结果，进入单品页,并记录下商品名称，商品价格
ml.getElement("第一个商品").click();
点击以后，单品页在一个新的页面打开了，所以这时候要切换页面，我们封装一个方法在Page类里面：
    public void switchWindowByIndex(int index){
        Object[] handles = driver.getWindowHandles().toArray();
        if(index>handles.length){
            return;
        }
        driver.switchTo().window(handles[index].toString());
    }
ml.switchWindowByIndex(1);
ProductPage pp = new ProductPage(driver);
String productName = pp.getElement("商品名称").getText();
String productPrice = pp.getElement("商品价格").getText();
String productUrl = driver.getTitle();
    int s = productUrl.lastIndexOf("/");
    int e = productUrl.lastIndexOf(".");
    String[] sku = new String[]{productUrl.substring(s+1, e)};
点击加入购物车
pp.getElement("加入购物车").click();
点击去购物车结算
pp.getElement("去购物车结算").click();
在购物车中验证：
    商品名称
    Assertion.verifyEquals(true, productName.contains(sc.getElement("商品名
    称", sku).getText()));
    商品价格
    Assertion.verifyEquals(productPrice.substring(1),sc.getElement("商品价格
    ", sku).getText().substring(1));
    是否被勾选
```

```
    Assertion.verifyEquals(true, sc.getElement("勾选商品",
    sku).isSelected());
    点击删除，确定
    sc.getElement("删除商品").click();
    sc.getElement("确定删除").click();
    验证购物车是否为空或者购物车中是否还有刚选中的商品
    sc.waitElementToBeNonDisplayed(sc.getElement("商品名称", sku));
    Assertion.verifyEquals(false, sc.isExist(sc.getElementNoWait("商品名称",
sku)));
```

我们来看一下最后的脚本：

```
public class Shopping extends TestBase{

    @Test(dataProvider="providerMethod")
    public void testShopping(Map<String, String> param){
        Assertion.flag = true;
        this.goTo(param.get("url"));
        FirstPage fp = new FirstPage(driver);
        Log.logInfo("从首页进入手机搜索页面");
        fp.linkToMobileList();
        MobileList ml = new MobileList(driver);
        ml.getElement("商品筛选华为").click();
        ml.getElement("商品价格筛选").click();
        ml.getElement("商品颜色筛选").click();
        ml.getElement("商品类型筛选").click();
        ml.getElement("第一个商品").click();
        ml.switchWindowByIndex(1);
        ProductPage pp = new ProductPage(driver);
        String productName = pp.getElement("商品名称").getText();
        String productPrice = pp.getElement("商品价格").getText();
        String productUrl = driver.getCurrentUrl();
        int s = productUrl.lastIndexOf("/");
        int e = productUrl.lastIndexOf(".");
        String[] sku = new String[]{productUrl.substring(s+1, e)};
        pp.getElement("加入购物车").click();
        pp.getElement("去购物车结算").click();
        ShoppingCart sc = new ShoppingCart(driver);
        Assertion.verifyEquals(true, productName.contains(sc.getElement("商
品名称", sku).getText()));
        Assertion.verifyEquals(productPrice.substring(1),sc.getElement("商品
价格", sku).getText().substring(1));
        Assertion.verifyEquals(true, sc.getElement("勾选商品",
sku).isSelected());
        sc.getElement("删除商品",sku).click();
        sc.getElement("确定删除").click();
```

```
        sc.waitElementToBeNonDisplayed(sc.getElement("商品名称", sku));
        Assertion.verifyEquals(false, sc.isExist(sc.getElementNoWait("商品名
称", sku)));
        Assert.assertTrue(Assertion.flag);
    }

}
```

至此我们的脚本演示完毕，希望大家能仔细体会这个框架的用法，用熟，仔细看下第二个场景的脚本，里面有窗口切换，有 XPATH 轴的用法，有 yaml 中参数的用法，用等待元素消失的用法，有验证，有字符串的操作。。等等等等。还是大家细细体会去吧！！！

第十四周：搭建框架五（自动化脚本的报告及结果分析）

如何定制化报告
对失败的脚本进行分析
构建自动化测试数据评估平台

上一节课中，我们把框架进一步完善了，且演示了如何使用，也带领大家写了两个脚本，我们以前讲过，testng 还有用 xml 来运行的写法，我们再来温习一下：

建一个 run 文件夹，在里面有个 testng.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<suite name="Suite" verbose="1" parallel="false" thread-count="1">
    <test name="Test" preserve-order="true">
        <classes>
            <class name="com.test.testcases.Login" />
            <class name="com.test.testcases.Shopping" />
        </classes>
    </test>
</suite>
```

这样我们把要运行的两个测试类都放进去了，在 eclipse 中运行这个 testng.xml 就行了，但是每次运行都要开 eclipse 吗？这样会不会太麻烦？JAVA 也是需要编译的，有没有能够编译 JAVA 的工具？且如何来定制一个报告？这些我们需要借助一个外部的工具:ANT，今天最后一节课，会带领大家如何使用 ANT:

一。安装 ANT

1、 到 http://ant.apache.org/bindownload.cgi 下载 ant 发布版本

2、 将下载后的 zip 文件解压缩到任意目录，比如 D:\ant

3、 在环境变量中增加 ANT_HOME=D:\ant(替换成你解压缩的目录)

4、 在环境变量 path 中增加 ;D:\ant\bin;

5、 打开 cmd ，输入 ant ，如果提示一下信息证明成功了

Buildfile: build.xml does not exist!

Build failed

或者

安装 ant

将你下载的压缩包解压，然后放在你喜欢的任何位置，如:c:/ant/，然后在“我的电脑－>属性－>高级－>环境变量 －>新建”指定:ANT_HOME,值为：c:/ant，并在 classpath 中添加：%ANT_HOME%\bin;

二。说明

1、安装完成后，

打开 cmd ，输入 ant ，如果提示一下信息证明成功了

Buildfile: build.xml does not exist!

Build failed

这里的 failed 并不是指你的 Ant 安装失败了，而是因为你只输入 ant 命令后，会在你当前目录下去寻找一个叫 build.xml 的文件，如果你 当前目录下没有这个 build.xml 的文件，则会报 build.xml does not exist!，而 build.xml 里存放的是你需要去干的一些事情，比如构建，执行，等

2、我在工程下面，即 bin 与 src 的同目录下建了一个 lib 的文件夹，把所有需要用到的 jar 包全放到里面，然后在 build.xml 里面去引用，

由此可以看出，我们最重要的就是去完成这个 build.xml 文件了：

```
<?xml version="1.0" encoding="UTF-8" ?>
<project name="selenium" default="start_run_tests" basedir=".">
    <property name="src" value="src"/>
    <property name="dest" value="classes"/>
    <property name="lib.dir" value="${basedir}/lib"/>
    <property name="suite.dir" value="${basedir}/run"/>
    <property name="jar" value="selenium.jar"/>
    <path id="compile.path">
         <fileset dir="${lib.dir}/">
            <include name="*.jar"/>
        </fileset>
        <pathelement location="${src}"/>
        <pathelement location="${dest}"/>
    </path>
    <target name="init">
        <mkdir dir="${dest}"/>
    </target>
    <target name="compile" depends="init">
        <javac srcdir="${src}" destdir="${dest}" classpathref="compile.path">
         <compilerarg line="-encoding UTF-8 "/>
        </javac>
    </target>
     <!--run testng ant task-->
     <taskdef resource="testngtasks" classpathref="compile.path"/>

     <target name="start_run_tests" depends="compile" description="start
selenium server and run tests">
         <parallel>
             <antcall target="run_tests">
             </antcall>
         </parallel>
     </target>
```

```
    <target name="run_tests" depends="compile">
        <testng classpathref="compile.path"  failureproperty="test.failed">
            <!--xml test suite file -->
            <xmlfileset dir="${suite.dir}">
                <include name="testng.xml"/>
            </xmlfileset>
        </testng>
        <antcall target="sendReport"/>
        <fail message="ERROR: test failed!!!!!" if="test.failed"/>
    </target>
   <target name="sendReport">
       <delete dir="${dest}"/>
       <antcall target="transform"/>
   </target>
   <target name="transform">
       <xslt in="${basedir}/test-output/testng-results.xml"
style="${basedir}/test-output/testng-results.xsl"
out="${basedir}/test-output/report.html" classpathref="compile.path">
           <param name="testNgXslt.outputDir"
expression="${basedir}/test-output/"/>
           <param name="testNgXslt.showRuntimeTotals" expression="true"/>

       </xslt>
   </target>
</project>
```

这个 build.xml，执行就是在 cmd 中输入：ant build.xml(也可以输入 build.xml 的全路径),简单讲解一下，project 结点的 default 属性，是代表入口函数，表示会从” start_run_tests”这个 target 开始执行起，property 结点就是变量的定义，target 可以理解为函数方法，target 中的 depends 表示依赖，在执行 target 时，会优先执行其依赖的 target,所以，我们可以看到上述这个 build.xml 的执行顺序：init-》compile-》start_run_tests-》run_tests-》sendReport-》transform，这个顺序简单来讲，就是初始化，编译，执行 testng.xml，生成报告。

最后的 transform 是用了一个 testng-xslt 的报告插件，这个插件的用法：

1. 下载 TestNG-xslt 把 saxon-8.7.jar 复制到测试项目的 lib 下
2. 从你下载的包中拷贝文件 testng-results.xsl 到 test-output 目录下。 testng-results.xsl 文件的位置是 testng-xslt-1.1.1\src\main\resources ，为什么要这个文件呢？因为我们的测试报告就是用这个 style 生成的。
3. 用 ant 运行这个 xml 就会在 test-output 目录下生成 index1.html ， 打开它就能看到新生成的测试报告 ，通过生成的报告我们能看到总体的情况，比如通过了多少 case ，失败了多少，跳过了多少没执行。第二个好处是我们可以查看失败的 case 抛出的异常，有具 体的函数和行号。我们还可以通过 case 执行后的状态来过滤查询等等。

从上面的 build.xml 中，可以看出会生成一个新的 report.html 的报告，这个报告中有详细信息，有 LOG 输出，有错误原因，大家可以从这些信息中来分析脚本失败原因。

分析脚本失败原因：

一般来说，脚本失败原因分为两种，一是断言失败，二是某个脚本本身有问题，断言失败好说，从报告中直接可以看出是哪个断言失败了，这时候，我们就可以手动去查看一下，如果核实，则再去问产品人员这

个断言失败是不是由于需求的变动而产生的，如果不是，则可以确定为一个 BUG，如果不是断言失败，这时候，我们就要去分析了，有可能是元素对象没找到，有可能是空指针异常等，这时候，我们通常再去单独的执行一下这个失败的脚本，然后查看是否再次失败，如果再次失败且是元素对象发生了改动，我们跟着更改元素对象即可，如果不是，则需要慢慢的 DEBUG 了。脚本失败的原因，在排除 BUG 的情况下，与网络环境，与机器的配置等很多因素有关系，很多经验的积累过程得同学们慢慢的摸索了！

结束语：

在此学习过程中，带着同学们敲了很多代码，可能很多代码与自动化并没有直接的关系，但我想说的是，只有爱上写代码，且写好代码，才能真正的做好自动化，才能对测试脚本有宏观的把控，才能在测试技术的道路上越走越远，希望同学们在这个阶段的课程中，能真正有所收获。最后，祝大家工作一帆风顺！