自动化测试概述(五)-接口的重要性

什么是脚本接口

上文已经详细描述了自动化测试活动中测试执行和脚本开发的职责，重点阐述了如何定义一个脚本的结构和调用关系。那什么是脚本接口应该做的呢？

testsuite：脚本文件，可以包含N个testcase
testcase：N个teststep组成的点集合
teststep：代表模拟你的操作，是一个具有最小且不可分隔性质的执行步骤

其实每一个teststep就是一个所谓的接口，每一个快速执行方法也都是一个所谓的接口，不理解？，看下面

脚本的定义:
脚本是具有一定逻辑性的代码文本：这意味着脚本既可以保存为无逻辑的文本，也能当作有逻辑概念的代码被引用

说穿了脚本结构只是定义和规范了你的执行顺序，首先做什么，然后做什么，自身不具有任何实际的意义，意味着任何编程语言和标记语言甚至是口述性质的Gherkin语言都可以作为脚本开发语言。因此脚本语言筛选没有先后、优劣之分。不过是脚本解释器你是直接使用编程语言的，还是自己写一套而已。

当其中的步骤不绑定实际操作，它就是文本；当步骤绑定了实际操作，它就是真实意义的脚本。

那脚本接口的意义就是在于赋予它实际操作。

接口工程师的职责

针对不同的脚本语言，实际的接口定义是完全不相同的，甚至存在"不兼容"的状态，从这里开始，脚本语言的优劣性就会变的非常巨大。你管不住脚本开发人员会自我放飞一般给脚本内添加各种东西，也管不住脚本开发人员的脑洞到底会开多大。因此脚本的语言选择应该优先选择强规范的，只有在团队人员水平普遍在一层的时候才选择弱规范类型的语言。

推荐链如下
标记语言(XML)->强规范语言(C#/JAVA/C/C++/Ruby等，你用Typescript我都不反对)->弱规范语言(Python/JavaScript等)-Gherkin

*XML、Gherkin都需要脚本接口开发工程师自行编写解释器。*是的，你没有看错，接口工程师是要自己写解释器的!不止是解释器，脚本执行器的开发也是你的活!

- 指定脚本的步骤、方法接口
- 开发脚本解释器
- 开发脚本执行器

这就是接口工程师的职责，那么我该如何做呢?

栗子仪式感

当脚本开发工程师的开发范围是跟随实际脚本语言的，因此你和脚本开发工程师需要合力完成以下任务

选择脚本语言

语言的选择非常关键，各类语言都有它天然的优势并可以加速开发过程。

- XML是强制性的标记语言，具有天然的规范性和读取顺序。作为脚本语言，是入门最为简单，书写最为规范和不可出错的；天然的执行顺序可以保证你的脚本执行顺序是完全符合真实的执行顺序
- 强规范的编程语言弱化了读取顺序，允许开发者在一定规则内执行多重路径，书写规范较之XML弱，但必须符合语言自身规则
- 弱规范的编程语言更加弱化了读取顺序和书写规范，但强化了开发速度，与强规范的编程语言，允许开发者在一定规则内执行多重路径

- Gherkin最为接近自然语言，通读性最高，更加符合操作行为，与XML相等的读取顺序，比弱规范语言更低的书写规范。只有几个关键词组成

可以看出，各个语言都具有自身的天然优势，但哪一个都可以充当脚本语言，如何筛选和确定脚本语言是非常重要的，脚本语言自身无优劣，但使用者却天然存在技术优劣。能和接口工程师一定探讨问题的脚本开发工程师必然是技术型的，但脚本开发工程师肯定不止一个。从测试执行升级上来的脚本开发天然的会呈现技术劣势，因此脚本语言也必然考虑他们的实际情况和接收程度。

我们也必须考虑的是脚本的可移植度，这里并不是说从Windows移植到Linux,而是当脚本完成后必然需要大规模的执行，我们需要考虑当整个脚本体系放入各个测试机时，环境配置的成本代价。

推荐

平衡型选项：C#，强规范语言，.Net运行库已经全面普及，天然打包所有的引用库，无论充当脚本还是充当接口的开发、工具的开发，都能简单入门
入门型选项：XML或Gherkin，用最简单的语言写最复杂的执行，说的就是他们，强标记保证不会有书写错误，强顺序保证不会有步骤遗漏。但需要接口工程师编写解释器
都想干选项：Python。入门简单，编写简单，不用编写解释器，甚至可以做到0基础就能开始工作

于是我选择C#，因为时代变了。
语言的选择到此为止，这是一个仁者见仁智者见智的问题，各有各的说法，只有自己喜欢不喜欢而已。

开发步骤、方法接口

在确定了脚本语言后，就要开始编写步骤和方法的接口
步骤的接口开发需要遵循以下几点

- *步骤尽量简洁，能够一眼就知道是做什么的*

```
<Element.Clear ElementID="控件id" />
```

```
Desktop.Click("控件id");
```

```
Then I Click "控件id"
```

- *步骤所需参数尽量少，能不带参数尽量不带参数*

```

```

```
<Element.SendKeys ClearFirst="true" ElementID="控件id">
   <Value>输入值</Value>
</Element.SendKeys>
```

```
WebFunc.SendKeys("控件id","输入值");
```

```
Then I input "输入值" in "控件id"
```

- 要善于应用语言的优势，缩短步骤的长度

```
C#支持类型的方法扩展
"控件id".Web().SendKeys("输入值");
```

- *Assert应该尽量通用，尽量通过参数或字段区分assert的类别*

```
<Assert.Objects>
   <WaitInfo TrueOrFalses="true" ExpectedValue="预期值" ActualType=
"Text" ElementID="控件id" />
<Assert.Objects>
```

```
Desktop.Assert("控件id", new { Value = "9" }, AssertType.Text);
```

- *善于使用方法，努力将一些常使用的步骤合并为一个方法供调用*

```
//验证输入范围,(控件id,最小输入值,最小预期值,最大输入值,最大预期值)
WebFunc.AssertInputRange("控件id", 0.99, 1, 0, 1);
```

```
Then I input "0.99" and "0" will get "1" and "1" in "控件id"
```

通俗的说就是 ***简单，简单，简单，*** 怎么简单怎么来

解释器开发

所谓的脚本解释器是解析脚本的，当脚本语言为非编程语言时，脚本解释器更加重要

非编程语言的脚本解释器

将非编程语言的脚本作为一个可提取信息的文本，从中提取所有的有效信息价值

最经典也是最有价值的例子就是Cucumber, Cucumber 是一个能够理解用普通语言 描述的测试用例的支持行为驱动开发（BDD）的自动化测试工具，用多种语言编写，支持Java、JavaScript、Ruby、.Net等多种开发语言。其脚本语言就是Gherkin。其语言自身是纯粹的文本，依赖各类语言的脚本解释器转为各类语言的实际代码

作为脚本，它的脚本是这样

```
Feature:Google search
    Scenario: search for keyword
        Given I am on google home page
        When I search for 'ThoughtWorks'
        Then I should be able to view the search result of 'Thoug
htWorks'

功能：谷歌搜索
    场景：输入搜索
    当我进入谷歌主页
    当我搜索'ThoughtWorks'
    我应该可以看到搜索'ThoughtWorks'的结果
```

而作为实际的实际脚本接口，它又是这样的

```ruby
require "rubygems"
require "watir-webdriver"
require 'rspec'

 Given /^I am on google home page$/ do
   @browser = Watir::Browser.new :chrome
   @browser.goto("www.google.com")
 end

```

```
When /^I search for  '([^"]*)'$/ do |search_text|
  @browser.text_field(:name => "q").set(search_text)
  @browser.button(:name => "btnK").click
end

Then /^I should be able to view the search result of '([^"]*)'$/
do |result_text|
  @browser.text.should include(result_text)
end
```

在实际操作中，可能比这还要略复杂，如果实际执行操作与接口再次解绑，那接口仅仅是调用执行器内的方法
接口的作用桥梁，就是告诉脚本有什么功能步骤支持且可用，自身再带调用执行器方法告知执行器要做这些步骤了

那XML怎么办?

```
<web:RoWebElement.Click web:RoWebElementID="FramePlan.New_OnceTime
_Btn" />

```

我们无法直接通过绑定的方式去执行xml，而是应该将XML变为我们想要的方法类，下面例子就是解析一个XML的步骤节点，转为一个ElementAction类型。

```
public class ElementAction : Basic
    {
        #region 属性

        public string SelectType { get; set; }
        public string SelectValue { get; set; }
        public bool ClearFirst { get; set; }
        public string SendKeys { get; set; }
        public bool SetFocus { get; set; }

        #endregion

```

```
13          /// <summary>
14          ///     构造函数
15          ///     解析控件操作
16          /// 默认架构存在
17          /// </summary>
18          public ElementAction()
19          {
20          }
21
22          /// <summary>
23          ///     构造函数
24          ///     解析控件操作
25          /// ROS
26          /// </summary>
27          /// <param name="element">传入的基础元素</param>
28          public ElementAction(XElement element) : base(element)
29          {
30              //SelectType值
31              SelectType = element.Attribute(XName.Get("SelectType"
    , Const.WebStr))?.Value ?? "ByText";
32
33              //SelectValue值
34              SelectValue = element.Element(XName.Get("Value", Cons
    t.WebStr))?.Value ?? string.Empty;
35
36              //是否优先清除
37              XAttribute clearfirst = element.Attribute(XName.Get(
    "ClearFirst", Const.WebStr));
38              ClearFirst = clearfirst != null && Convert.ToBoolean(
    clearfirst.Value);
39              //发送信息值
40              SendKeys = element.Element(XName.Get("Value", Const.W
    ebStr))?.Value ?? string.Empty;
41
42              //是否设置焦点
43              XElement focus = element.Element(XName.Get("SetFocus"
    , Const.WebStr));
44              SetFocus = focus != null && Convert.ToBoolean(focus?.
    Value);
45          }
```

```
46      }
```

可以发现，都是将非编程语言的脚本步骤转为对应的解释器开发语言，或是方法，或是属性类，再提交给执行器去执行。

编程语言的脚本解释器

编程语言的脚本解释器可以说是非常的简单，但也可以非常的复杂。
如下所示：纯粹的C#代码脚本，在其自身被解释时，实际是.NET运行库本身在做识别。

```
1 await AsyncDesktop.Assert("TitleName_Text", new { Value = "Calcula
  tor" }, AssertType.Text);
2 await AsyncDesktop.Assert("Nav_Btn", new { }, AssertType.Displayed
  );
3 var r1 = (await "Num0".Deskop().Click()).TResult;
4 await AsyncDesktop.Click("plusBtn");
```

但它依然是可以写出一个解释器的

```
1 /// <summary>
2         /// 点击控件
3         /// </summary>
4         /// <param name="element">元素</param>
5         /// <param name="strackstepinfo"></param>
6         /// <returns></returns>
7         public async Task<StepResult> Click(string element, dynam
  ic strackstepinfo = null)
8         {
9             dynamic rsi = strackstepinfo == null ? StrackStepInfo
  () : strackstepinfo;
10            ElementAction action = new ElementAction
11            {
12                ElementId = element,
13                ActionType = "Element.Click",
14                LineInfo = rsi
15            };
16
17            try
18            {
```

```
19                  var result = await CsStepEvents.OnTransStepToExec
   uteAsync("Element.Click", action);
20                  return result;
21              }
22              catch (NullReferenceException)
23              {
24                  return null;
25              }
26          }
```

可以发现，无论是XML还是C#的脚本，最终识别完成后，都会被解析为一个ElementAction类。实际上这是为了开发脚本执行器的方便

设计解释器时，设计思路应遵循以下几点

- *无论语言如何，解释器都应该是尽量获取一个通用的信息结构，方便转交给执行器*
- *解释器是解析步骤接口和方法的，方法内使用步骤接口，步骤接口被解释器赋予实际意义*
- *解释非编程语言时，解释器的作用是理解相应的步骤是做什么，它就是个真·解释器*
- *解释编程语言时，解释器的作用就是解析接口和拼接内容，它就是个集成器*

执行器开发

脚本执行器是处于整个接口开发的最底层，它的实际作用的直接执行接口方法，操作浏览器、程序或APP直接执行对应的操作

因此，它基本是需要和测试框架集成的，也是直接使用者，而在设计执行器时，设计思路应遵循以下几点

- *从解释器传入的对象应该无视脚本语言，执行器只对执行工具负责*
- *回传信息应该尽量丰富，包含所有应该包含的信息和对象本体*
- *执行器内方法应该尽量简单，最好是实现单步操作，复杂操作和复杂内容应该优先让脚本接口完成*

```
1  /// <summary>
2          ///      点击某控件
3          /// </summary>
4          public async Task<StepResult> Click()
5          {
6              StepResult result = await Task.Run(() =>
7              {
8                  StepResult stepResult = BackStepResultObj.Result(
   _elementAction);
```

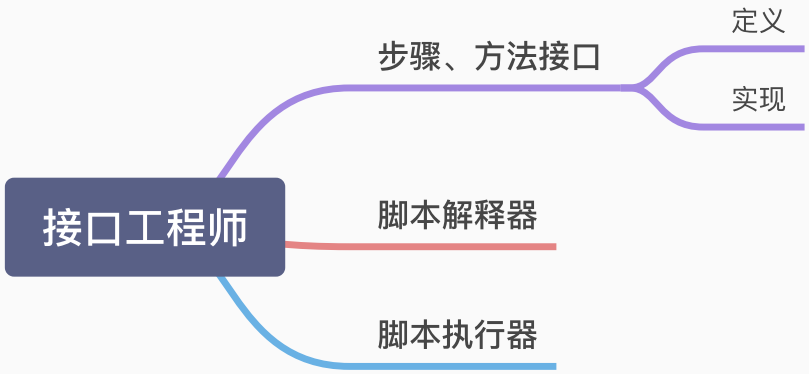
```
 9
10                 try
11                 {
12                     WindowsElement findele = FindDesktopElement.Q
  ueryRoElement(_elementAction.ElementId, _elementAction.Timeout);
13                     if (findele == null)
14                     {
15                         stepResult.Result = false;
16                         stepResult.ExtraInfo = $"{FindDesktopElem
  ent.ExtraInfo}";
17                     }
18                     else
19                     {
20                         //点击操作
21                         findele.Click();
22 
23                         stepResult.Result = true;
24                         stepResult.ExtraInfo = "已点击对应控件";
25                     }
26                 }
27                 catch (WebDriverTimeoutException)
28                 {
29                     stepResult.Result = false;
30                     stepResult.ExtraInfo = "RoWarlock操作超时";
31                 }
32 
33                 catch (Exception e)
34                 {
35                     stepResult.Result = false;
36                     stepResult.ExtraInfo = $"类:{GetType().Name}中
  方法:{MethodBase.GetCurrentMethod().Name}发生异常";
37                     stepResult.ExpcetionInfo = new ExpcetionInfo
38                     {
39                         StackTrace = e.StackTrace,
40                         Message = e.Message,
41                         Tostring = e.ToString()
42                     };
43                 }
44 
45                 return stepResult;
```

```
46              });
47
48              return result;
49          }
```

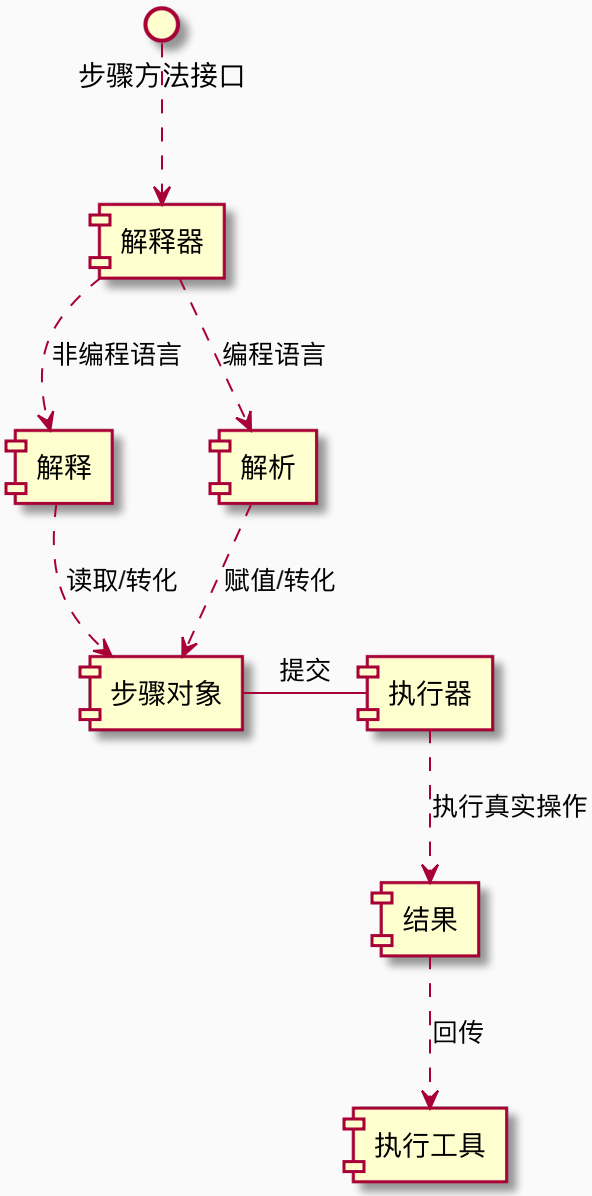
这是就是一个点击的执行方法，可以看出回传了非常多的信息，同时也考虑了很多异常情况，这样才不会在执行层面出错进而影响上层

不说了，上图

工作内容



接口库


步骤方法接口
解释器
非编程语言
编程语言
解释
解析
读取/转化
赋值/转化
步骤对象
提交
执行器
执行真实操作
结果
回传
执行工具