自动化测试概述(七)-"奥利奥"架构

前面已经详细描述了自动化测试活动中，四类人的工作内容，本文将重点描述自动化测试的大架构

名称

脚本

- *ROI：值参数文件，存储所有的键值对*
- *ROS：保存单个testsuite*
- *ROC：步骤宏(引用)文件，存储所有的步骤宏(引用)*

ROS

- *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*

接口

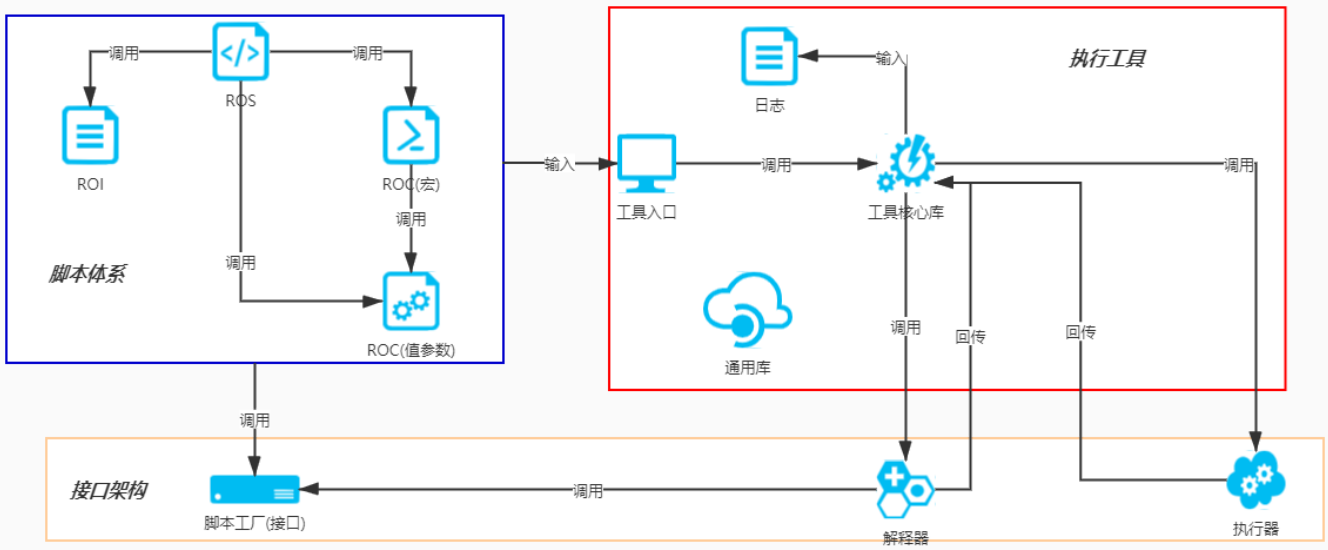
- *步骤、方法接口：定义和实现teststep步骤*
- *解释器：解析脚本架构*
- *执行器：执行真实的teststep操作*

工具

- *入口：读取脚本和显示日志，配置执行参数*
- *集成解析器与扩展：*
- *集成脚本接口与扩展：*
- *集成执行器与扩展*
- *日志与结果保存：*
- *通用库*

架构简述

凑合看吧



这就是"奥利奥"架构，三三三体系

所谓奥利奥架构：

- ***即脚本体系、执行工具、接口架构互相分离***
- ***脚本体系中，执行逻辑、元素定位、值参数互相分离***
- ***接口架构中，执行器、脚本接口、解释器互相分离***
- ***通过执行工具从中调和和引用***

低度的耦合，可以在不影响其他组件的情况下，进行适度的修改；
各个职责都有他的使用范围，互补干涉；

通过有限的沟通可以让工具跑起来并*最终塞给用户一个完整的"奥利奥"饼干；*
来人啊，喂两位公子吃饼。（梗懂的都是自己人）

脚本体系

- ROS:脚本逻辑，纯粹的脚本，无论语言，它只是一份文档，用以记录执行顺序
- ROC:存储常用的步骤宏，可被任意ROS调用；存储键值对参数，可被任意步骤宏和ROS调用
- ROI:一种特殊的键值对，用以存储元素信息，针对使用的元素赋予一个ID，并根据自身需求添加定位元素的值。在任意步骤宏和ROS中，如果使用元素则直接使用ID即可

接口架构

- 脚本工厂(接口):定义元素属性ROI的元素结构；定义ROS中的teststep的结构，实现ROS中的teststep
- 解释器:解析整个脚本体系；解析ROS，载入ROC、ROI；解析ROS包含的所有teststep，相应ROS引用的其他ROS文件等；
- 执行器：执行真实的操作

执行工具

- 工具入口: GUI或命令行，读取脚本体系和日志、结果显示
- 工具核心库:调用接口架构，用以执行各类操作；初步的脚本筛选和解析
- 日志与通用库:保存结构和存储所有库可能用到的对象

这就是"奥利奥"架构，三三三体系，其实图还能画的更炫，更好看

TIPS

一

在此架构中，你会发现执行工具更偏向于架构体系，所有的解析和执行已经被接口架构库所替代；因此执行工具更像一个工作台；它起到了沟通与桥梁的作用；它是在接口架构基础上做的拓展，用以服务于接口架构。因此我们可以做的更为激进；执行工具都能分拆为Server和Client。

- 执行工具server:执行工作台，主要就是读取和呈现脚本、结果
- 执行工具Client:带有除入口外的执行工具

两者可以通过网络连接，为大规模集群、实施提供了基础，事实上任何一个支持集群测试的自动化框架都是这样的架构

二

我们可以发现脚本工厂和执行器也并不相连，这！就！是！重！点！*脚本开发权利归于任何人！*
在合理范围内，实际上脚本开发人员只要开发脚本工厂和执行器，就能覆盖任何测试对象，添加和执行任意的执行步骤，且不用去更改自动化测试工具的任何地方！我的脚本我做主， 我想加什么步骤就加什么步

骤，我想用来测试什么对象就测试什么对象，web/client/app统统可以！
当然真实情况并不是如此完美，但依然非常简单，基于.net体系，可能你从创造脚本到完成执行就开始测试client只需要最多2周时间！我想说的是一种理念：*脚本开发权利归于任何人！，当你的脚本已经无法写下去的时候，认为应该添加更多的方法和接口，你就可以自己来，让脚本开发者的工作变得更加有趣，我创造，我实现！*

三

在真实的开发中，脚本解释器很多时候并不与工具核心库区分，往往两者是合二为一，所以我也多次说过，开发接口和开发工具往往需要一起协同作战。因为脚本解释器是一个测试工具的核心内容，它的确定基本就确定了你到底走那条测试之路，DDT/KDT/BDT/ADT 以及任何的xDT 在这一刻就会确定。

测试架构有且只有一个主线xDT，任何附带的xDT都无法完美的替换主线xDT!

四

脚本体系其实是最复杂的，真的很需要有一个脚本架构师来指导和创建脚本架构。

单一的脚本

- XML

```
<ros:TestDefinition xmlns:ros="http://tempuri.org/RoFramework.xsd"
                    xmlns:web="http://tempuri.org/RoWebAutomation.xsd"
                    xmlns:gui="http://tempuri.org/RoDesktopAutomation.xsd"
                    xmlns:xs="http://www.w3.org/2001/XMLSchema"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xmlns:sco="http://tempuri.org/RosFile.xsd">

  <!--对整个测试文件进行描述,整个Annotation相关信息会加载进入Reprot中-->
  <ros:Annotation>
    <ros:Description>
      对整个测试脚本进行描述详情
    </ros:Description>
    <ros:Created ros:Author="nate" ros:Date="2017-08-16" />
    <ros:LastUpdated ros:Author="nate" ros:Date="2017-08-16" />
  </ros:Annotation>

  <!--测试设置-->
  <ros:TestConfig>

    <!--脚本使用参数设置，参数可以设置多个-->
    <ros:Properties> ... </ros:Properties>

    <!--roi文件引入，支持带入参数-->
    <ros:Imports> ... </ros:Imports>

  </ros:TestConfig>

  <!--程序驱动配置-->
  <ros:StartApp> ... </ros:StartApp>


  <!--测试正文-->
  <ros:Tests>

    <!--测试步骤，允许多个TestCase-->
    <ros:TestCase ros:ID="Web"> ... </ros:TestCase>


    <ros:TestCase ros:ID="Desktop"> ... </ros:TestCase>
  </ros:Tests>

  <!--是否关闭测试驱动-->
  <ros:CloseApp ros:Keep="true" />

  <!--log处理，写入另存为Log的路径，默认情况下桌面会自动生成一份-->
  <ros:LogFunction> ... </ros:LogFunction>

</ros:TestDefinition>
```

- 纯态的C#

```
1 namespace MtstarTest
2 {
3     public class CalDebug : Ro.Common.Format.Script.PureCs.Script
  Format
4     {
5         #region 属性
```

```
6
7           /// <summary>
8           /// 定义脚本的参数
9           /// </summary>
10          public sealed override ScriptProperty ScriptProperty { ge
    t; set; }
11
12          /// <summary>
13          /// 编译执行
14          /// 异步方法
15          /// </summary>
16          public AsyncDesktopMain AsyncDesktop { get; set; }
17
18          #endregion
19
20
21          #region 构造函数
22
23          /// <summary>
24          /// 构造函数
25          /// 所有定义的属性，都需要在构造函数内初始化
26          /// </summary>
27          public CalDebug()
28          {
29              ScriptProperty = new ScriptProperty(); //初始化
30              AsyncDesktop = new AsyncDesktopMain(); //初始化
31          }
32
33          #endregion
34
35
36          #region 脚本结构
37
38          /// <summary>
39          /// 初始脚本
40          /// </summary>
41          public override void InitScript()
42          {
43              //! 定义的脚本参数将被转为实体类型
44              ScriptProperty.UseBrowser = Browser.ChromeHeadLess;
```

```
            ScriptProperty.CloseBrowser = true;
            ScriptProperty.LogFilePath = "${LogPath}";
            ScriptProperty.RoiMain = "FrameCal";
        }

        /// <summary>
        /// 执行脚本
        /// </summary>
        public override void RunScript()
        {
            //! 方法内部为空，当执行本方法时会自动解析、执行实际case方法
        }

        /// <summary>
        /// 释放脚本
        /// </summary>
        public override void DisposeAll()
        {
            //! 方法内部为空，当执行本方法时，会进行清理脚本、关闭测试软件、保存日志、释放内存
        }

        #endregion


        #region 实际case方法

        [CaseDesc("Launch", 1)]
        private async Task Launch()
        {
        }

        [CaseDesc("TestClick", 2)]
        private async Task TestClick()
        {
        }

        [CaseDesc("Close", 3)]
        private async Task Close()
        {
```

```
84          }
85
86          #endregion
87      }
88  }
```

多脚本的调用

显示往往比之理论来的更加复杂，之前就已经说过，脚本存在互相调用的关系，有些前置有些后面必须执行，但脚本数量越多，就显得愈加复杂(看这里)，我也说过到底应该如何做，这里我就再多嘴说一句

- 脚本必须是单一的个体

无论它有多复杂，有多少调用关系，它都是单一的个体，只有单一的个体，它才是最好操作的

- 调用关系放在脚本内

诸如XML 通过添加 "Import" 节点，编程语言使用 using, import等，让调用关系只是在代码层面产生关联，而不是等到要执行时才去先跑哪个再跑哪个，脚本执行人员不是奴隶，人家没义务为了你复杂的调用关系还特地去记录一堆先后执行顺序！同样这也是你写脚本的人的失职！你为什么不写在脚本内，真实的载入和识别让开发工具去做，不然要工作干什么的，这是你的需求，他的工作！不要让工作复杂化！

- 让脚本只作为文本

如果是脱离各种xUnit形式的自动化测试工具，要让脚本只是一个文本个体，它不应该受到拘束。它只对你的测试工具有作用。

- 与CASE的关系要简单

现在自动化测试脚本往往对很多文档关联，需求文档，技术文档，手动case，而他们的直接的相互作用有时候又特别复杂。我们应该找对关系链，而不是一股脑的全关联起来，从一个需求到一个自动化脚本，中间必然是会丢失信息的，这些丢失的信息也必然会映射到其他文档。自动化脚本应该只对一个对象文档负责，而不是所有文档!

a. **脚本覆盖case->BDT/KDT**
b. **脚本覆盖技术点->DDT/KDT**
c. **脚本覆盖需求-BDT/KDT**

简单的说就是这么简单，不要复杂，只对一个负责，不要试图关联所有！技术实现也只关联技术需求文档，也不会直接关联需求，凭什么要让脚本关联所有？记住关系链！只要P2P，不要P2C！脚本不是最后一环，其他人没权利、脚本也没义务让你们都知道我到底关联了几个关系！

- 善用执行链

可以针对某一个操作、某一个验证点、某一个控件可以直接执行一系列的操作步骤，

```
"Name_Text".Web().SendKeys("${SendLargeLength_50}").Result.Focus()
.Result.Assert(new { Value = "25", Type = "Equal" }, AssertType.Le
ngth);
```

例如我针对 "Name_Text"控件 做的一系列操作，在日益多的E2E测试框架中，用的越来越多，这个链只对这个控件负责，让操作有节奏感，而不是分拆写成一堆废操作

- 不要随便拉高语言的上限

显示往往比理论来的更梦幻，诸如针对XML写入多种逻辑判断，具有更多的逻辑操作、引用和其他非标记语言该干的事情，简直在突破XML的语言上限！这些应该是编程语言该干的事情
所谓XML 指可扩展标记语言（eXtensible Markup Language）。
XML 被设计用来传输和存储数据。

这定义，从最初设计到使用，从来就没有考虑过在里面添加逻辑判断！每个语言都有自己的先天优势和劣势，当它作为脚本存在时，应该扩大优势而不是拉高劣势！编译器要写成什么样才能完全覆盖和不会容易产生错误！切换成其他语言就不香么？就算你想用XML并且做一定的逻辑执行操作，你就不能考虑VUE是怎么写前端网页的？

```
<div id="app-3">
  <p v-if="seen">现在你看到我了</p>
</div>
```

```
var app3 = new Vue({
  el: '#app-3',
  data: {
    seen: true
  }
})
```

某种程度上，非编程语言的测试脚本和VUE写前端几乎一模一样！XML作为执行逻辑显示结构，应该只是展现执行逻辑，真实的执行逻辑应该隐藏在XML之下！否则解释器还需要判断各类符号来确保脚本可以流畅执行！

```
<Set name="myPrintType" config="PrintType"/>
<If condition="#myPrintType==1">
  ...
</If>
```

诸如这样的设定，切换为C#或者其他语言就是两句话

```
private string myPrintType=PrintType.ToString();
if(myPrintType==1)
{}
else
{}
```

实时执行时只要判断回传即可，而使用XML还需要让解释器记录和判断condition到底是什么
不要随便挑战被选定脚本语言的上限，真就想使用标记语言写出逻辑判断，也请将逻辑判断的所有要素归于编程语言!

最后一句

道生一，一生二，二生三，三生"奥利奥"架构(抖个机灵!)