自动化测试概述(四)-执行？开发？

测试执行

他们是做什么的?

如之前所说，处于脚本周期的生命末端，其职责包含但不限于

- 测试环境配置
- 测试脚本执行
- 测试结果收取
- 脚本结果分析
- 测试问题汇报

可以发现一个自动化测试执行工程师除了不手动跑测试用例，其实和普通的测试工程师没有任何区别

为什么没有存在感?

一旦抛弃测试脚本的设定，一个基本的执行工程师≥测试工程师，在一个团队内，往往也是测试工程师"兼职"执行脚本，并承当相应的职责。当团队自动化程度越高，这两者就越统一直至某一个岗位彻底消失。
无论说执行工程师还是测试工程师，他们承担了大部分测试活动中最主要的工作，即执行用例(脚本)、完成测试、收取结果、提交issue并给出意见。因此两者可以无缝转换；伴随测试团队的发展，几乎所有的手动测试工程师都会开始转为自动化测试工程师。同时执行工程师也是手动测试意图进入自动化测试领域的第一步。先懂做什么，再说怎么做。

脚本开发

重点来了
脚本开发是自动化测试过程中**非常重要**的一段，好的脚本开发会大大减轻测试执行者的工作量；可以极大的提高脚本覆盖率；也能间接降低脚本接口(DLL)和测试工具的开发量。但现实之中，普通的脚本开发者没有完备的架构体系知识，只知道一味的写脚本。***我们不缺写脚本的人，因为人人都能写脚本，我们缺的是编写脚本架构的人***

正如上文所说，为什么现在的脚本毫无灵魂？伴随着待测试软件的不断发展，功能迭代、待测试的功能点也越来越多，需要覆盖的范围和深度越来越大；随之而来的脚本开发工作也在不断加大，但几乎9成的人会选择不断堆脚本还提高覆盖率。虽然量变可以引发质变，但也需要诱因啊！

以下会有举例的方式来描述我的思想，看不懂的多看几遍

我应该怎么分类?

如同干垃圾和湿垃圾一般，针对脚本也应该做分类，这是脚本开发的第一步
我要测的是什么?
APP、桌面软件、WEB、接口？不管什么，统称***测试对象***
我们走的是什么测试路子?
BDT、DDT？***确定什么测试路子就会开始形成不同的分路***
我该怎么给我的脚本分类?

- 按功能：以单个功能为最小维度，不同的功能放置不同的脚本内，按照数据流、操作流链接各个脚本
- 按操作：以单个操作为最小维度，不同的操作放置在不同的脚本内，按照操作流链接各个脚本
- 按用例：照搬手动用例，单个手动用例为最小维度，不同的用例放置在不同的脚本内，按照用例的执行链接各个脚本
- 按验证类型：以单个验证点为最小维度，不同的验证点放置在不同的脚本内，按照验证流连接各个脚本
- 其他：

综上就是就是脚本的大致分类，由此可以得出两个量：
点：将N个步骤操作按照以上的分类进行分拆，成一个个再也无法拆的点；
线：连接各个点的操作、数据

太复杂了，先说术语

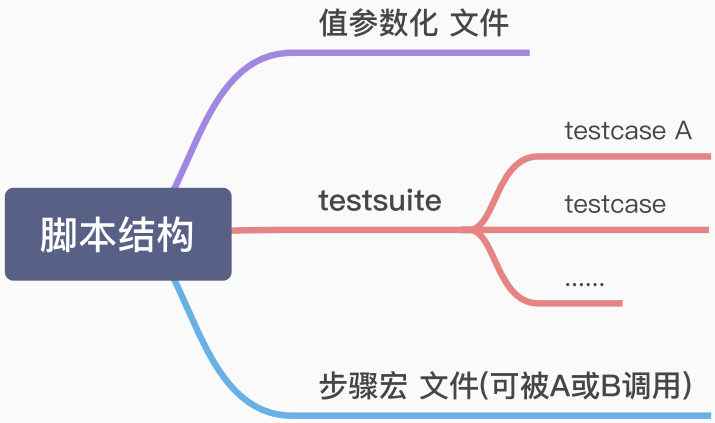
为了方便，我们可以定义以下的术语(自己定义的，不要见怪)

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

一般脚本会按照正常的操作进行开发编写，这样会获得一份*testsuite*，并包含N个*testcase*。在针对独特输入值或独特操作时，全步骤 是"Pass"的，但针对其他输入值或操作，则必然 "Fail"，那怎么办？***值参数化***

针对输入值、验证值、独特操作进行参数化，在执行到相应*teststep*时，替换为对应的参数值带入执行，并返回值与验证值对比。***值参数化***可以让*testcase*不受数据的约束。只需保持*teststep*顺序即可。但又有些功能需要一些前置*teststep*才能正常进行怎么办？***步骤宏(引用)***

将一些常用的、通用性质的*teststep*封装为对象，在其需要的时候直接引用即可，***步骤宏(引用)*** 可以让*testcase*只需要关注*点自*身，而非投入过多精力去描述该*点*的其他部分。可以看出是***步骤宏(引用)***是一种特殊的*testcase*



因此一份脚本结构正常可以由3个部分组成

1. 值参数文件：保存参数名和参数值，Key-Value
2. 步骤文件：保存testsuite
3. 宏文件：保存可以调用的一个个可以调用的对象

这样的分类可以使任意一方的修改可以不去影响其他两方，降低了耦合，也提高了开发效率

BDT/DDT/KDT

其实百度一下你就知道了，但为了大家的时间，我贴出来，当然其他这些概念基本都是从开发模式那借鉴过来的。

1. 什么是BDT

由BDD引申出的一个概念
BDD全称Behavior Driven Development，译作"行为驱动开发"，是基于TDD (Test Driven Development 测试驱动开发)的软件开发过程和方法。
BDD可以让项目成员（甚至是不懂编程的）使用自然语言来描述系统功能和场景，从而根据这些描述步骤进行系统自动化的测试。
而BDT（Behavior Driven Test），一句话概述，**操作优先**。*teststep具有连贯性，testcase具有连贯性。*

2.什么是DDT

即数据驱动测试（Data Driven Test）
将测试数据从test case分离，通过改变这些数据我们可以得到不同的执行结果（比如说加法计算器，我们输入1,1得到2，而输入1,2得到结果为3），输入的数据是结果的决定因素。
一句话概述，**数据优先**。*teststep具有连贯性，testcase不具有连贯性。*

3.什么是KDT

KDT(Keyword-driven testing)，即关键词驱动
KDT又称为"action word based testing"，改变"action word"（含使用的数据）测试结果也将发生变化，我们可以理解为是词驱动的测试。
一句话概述，**步骤优先**。*teststep不具有连贯性，testcase不具有连贯性。*

知道了三个概念，就会发现上文的脚本结构已经完整的包含以上三类测试驱动行为。
teststep就是KDT，不同的teststep会形成不同的实际结果
testcase就是BDT，用以描述一个测试对象中的单独的功能。
值参数化就是DDT，不同的参数针对相同的testcase或teststep会产生不同的实际结果
而步骤宏(引用)可以作为特殊的testcase或teststep被步骤文件引用，且自身也支持值参数化

我是栗子

我想要测试登陆百度这一个功能 ,步骤如下

```
1. 打开浏览器 (When I open web browser)
2. 输入网址:https://www.baidu.com input (And I input "https://www.baidu.com" )
3. 点击登陆 (When I click "login" button)
4. 输入账号{username} (Then I input "{username}")
5. 输入账号{password} (And I input "{password}")
```

```
6 6. 点击登陆 (Then I open web browser)
```

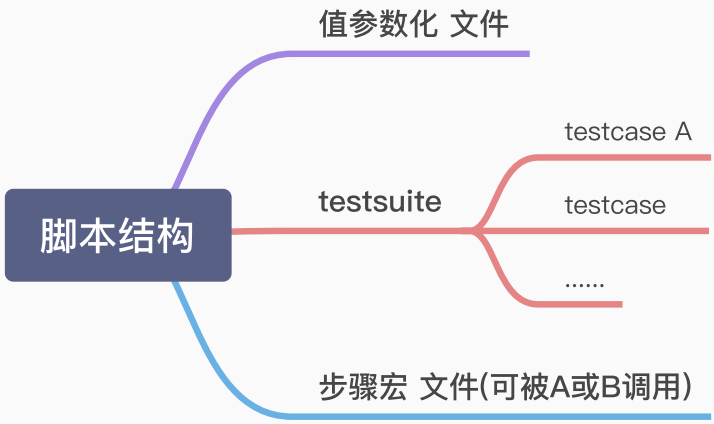
我们可以看到：

- 每一步都是*teststep*，因为都是最小化且不可分隔的操作，而"打开"、"输入"、"点击" 则是代表*KDT*
- 每个步骤对应的括号内是使用BDD的方式描述步骤，整个6个*teststep*组成了一个单独的登陆功能，即一个*testcase*，代表*BDT*
- 其中 "{username}" 和 "{password}"可以*值参数化*，代表*DDT*
- 而这个功能可以被作为一个*步骤宏(引用)* 被其他*testcase*引用，并作为前置*teststep*执行

可以看出*teststep*代表的是真实的一步执行操作，*testcase*代表的是测试对象的一个单一功能。*DDT*则允许带入不同数值执行操作

综上，可以看出脚本结构三层模式的优势，无论脚本开发人员究竟选择哪种测试驱动模式，都能发挥其优势。

温故而知新



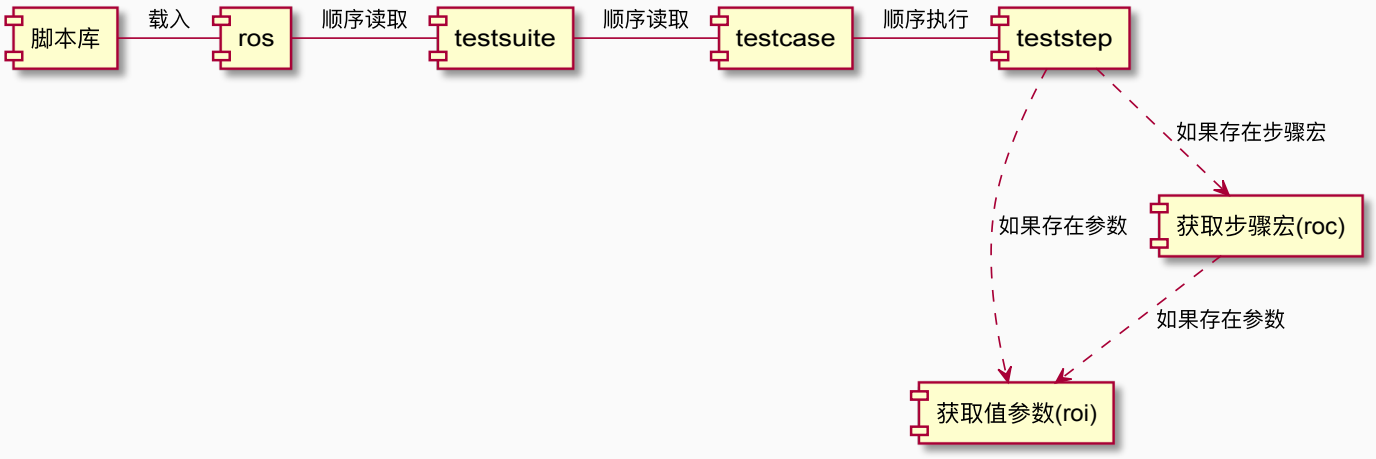
给个名字吧

在实际操作中，没有名字是万不可取的，因此在实际工作中，我将这三类文件分别取名

- 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)

三类文件组成脚本库脚本库

调用关系



- roi文件处于最底层，可被当成一个文本的存储器，仅存放*值参数*
- roc和ros都可以调用roi文件内的存储的*值参数*
- ros可以调用roc内存储的*步骤宏(引用)*
- *从脚本库到步骤是线性状态*

这就是最简单也是适应性最为宽泛的脚本架构，在实际工作中，无论是使用哪一类测试驱动模型，基本都脱不开这样的架构模式。仅仅是各有侧重点不同，只有适合自己的，没有落伍和out的。现在很多测试从业人员尤其是管理者为某一类DT为天下最好，根本不看自身项目到底适合不适合，这种思想不可取，所以脚本开发工程师一定要谨记: ***适合项目的就是最好的***

别急，还没结束

当你按照上文架构进行设计和脚本编写后，会发现一个非常验证的问题，就是当脚本数量达到一定数量后，脚本的管理会变得非常复杂。

1. testcase/testsuite与testcase/testsuite也存在互有影响。有可能是其他testcase/testsuite的前置或后置条件，但又不适合写入roc文件作为步骤宏(引用)
2. 某些特定的teststep需要引用其他testcase/testsuite或有关联
3. ros文件变的非常大，数量已经逐渐变的不可控
4. roi文件变的非常大，数量已经逐渐变的不可控；部分值参数也开始依赖其他的值参数
5. 其他你可能想到的复杂情况

Q1&Q2

很多针对1、2，很多脚本开发工程师会提出的多加几层结构上的"逻辑概念"，多个具有逻辑从属关系的testsuite组成一份testlibrary。再依次组成若干逻辑层上的testlibrary链。且不说这样的链能连接多少testsuite，一旦无上限的叠加逻辑层次的概念，很容易让人产生这到底归属到哪一个testlibrary的问题。我并不非常推荐这样的叠加结构，它只是为了解决问题而解决问题。

还有一些脚本开发工程师借用系统中"文件夹->文件"的概念，通过将不同的testsuite划分到文件夹内来达到分类的结果。但实际上，在执行工具UI层看这样的结构，形成的casetree会非常难看，且不具有可选择

性，***脚本是具有一定逻辑性的代码文本，***单纯的"文件夹->文件"的概念无法正确的归纳各个testsuite。

实际上问题1与2是考验一位脚本开发工程师的标准题。在三层次架构下，怎么选择、开创脚本管理结构非常考验脚本开发工程师的能力。即便如我，也没有太好的办法管理自己维护的脚本。

但开发可能可以给予我们答案，那就是导入（import/using/include）

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

实际上脚本既可以用编程语言编写，也可以使用标记语言编写；如果用编程语言编写，可以直接通过引用（import/using/include）的方式建立逻辑上的连接。而脚本自身可以通过"文件夹->文件"进行管理。

而用标记语言编写则应该在testsuite内添加相应的描述，在脚本加载时，通过执行工具获取引用。而脚本自身可以通过"文件夹->文件"进行管理。

这肯定不是最正确的办法，但应该是目前最简单的管理办法

脚本自身作为文本文档可以随意管理；而作为脚本则通过内在逻辑管理进行处理。当然这一切都脱离不开脚本接口和执行工具的支持

Q3&Q4

ros是需要限制的，并非所有通用性质的步骤都可以直接存储；如针对上文给定的方案来看，ros已经更加趋向逻辑管理的一种辅助，但脚本之间的testsuite、testcase无法通过直接建立逻辑连接时，可以通过ros的方式建立。因此ros存储的是大部分脚本都会用的一些步骤；或无法一些通过直接建立逻辑连接的脚本需要使用的步骤

而针对roi文件，值参数应该是一种键值对，Key-Value模式，而某些键值对可能依赖其他的键值，从人为上就应该杜绝，键值的嵌套从开发上很难处理，同时roi是一种无限接近纯文本的文件，自身不应该带入过多的逻辑处理

我该怎么做

从上文可以看出，***身为一位脚本开发工程师，最需要的并不是开发、编程的能力；而是思考的能力。***
因为开发、编程的高低只决定了脚本的下限，而思考的能力则决定了脚本的上限。多思考，多去想，不要直接写脚本，那样最终累的只有自己；直接写的脚本终究只是失去了灵魂的文本。