自动化测试执行框架(AutoCore)使用手册

Author: [Xiaowang347@163.com](mailto:Xiaowang347@163.com)

1. 框架背景：

我们自动化团队工作于一家通信公司, 原先的自动化项目开发与用例维护都在GTR平台上开展, GTR与用例管理平台(TMSS)有着极好地互通性, 同时GTR具有良好地用例管理与日志报告功能, 成为自动化工具的不二之选. 但是写过TCL语言, 尤其是维护过TCL脚本的同事就知道那绝对是一件让人非常头疼的事情, 脚本语言不能调试, 同时重新一条用例需要加载一次GTR, 那个庞然大物, 也是非常让人头疼. 出于超高的维护成本, 我们开始讨论：

用例一定要用脚本语言编写么？

不能将开发环境切换到C,C++,C#么？(之所以选择C#也有客观原因)

我们开发了一套轻量级的C#开发环境的海盗船自动化平台来取代GTR. 其中AutoCore就是该自动化平台中执行自动化用例的框架.

1. 自动化测试执行框架的说明：

AutoCore 相当于一个很大的容器(内有烹饪技术优良的厨师), 自动化用例就是每一道’菜’的原材料, 然后一批批待加工的菜丢进容器内, 过了一段时间之后, 输出的就是一道又一道可口美味的佳肴(代指自动化报告, 用于人工分析失败与成功用例). 厨师的手艺就是自动化执行框架所做的事情. 每道菜的原材料需要清理(加载测试集), 切片(测试逻辑的环境初始化), 烹饪(执行测试逻辑), 洗锅(执行测试逻辑的环境恢复), 装碗(恢复测试集) 这么五道工序. 其中是不是发现很多菜可以一起被清洗, 还有一些菜可以烧了之后不用恢复环境, 直接烧下一道菜, 这里就是测试集与测试逻辑的概念了. 一个测试集可以包含几个环境相同的测试逻辑, 而一个测试逻辑又可以绑定几个相同的测试数据.

这个就是自动化执行流程了. 见图1.

1. 测试集, 测试数据, 测试逻辑, 测试用例的概念

一个测试集可以包含N个测试逻辑, 一个测试逻辑可以绑定N个测试数据. 打个比方, 一个大学有N个学院, 每个学院有N个班级, 而每个班级有N个学生, 那么N个学生都相当于测试用例的概念, 测试用例 = 测试逻辑 +　测试数据．

自动化流程比作是一所大学对每个学院的学生进行思想工作辅导. 那么理想的有序的顺序应该这样子(这儿要抛开并行, 因为辅导老师只有一个(就像环境就只有一套)), 那个老师告诉其中一个学院, “从你们这儿开始辅导, 请学院做好准备”, 这个就是初始化测试集了, 然后从班级一开始, 这个就是初始化测试逻辑, 然后对学生进行辅导(执行测试用例), 辅导完之后就收拾一下那张桌子, 重新放回表格(测试用例环境的恢复), 然后班级一的学生都结束辅导之后, 就把班级的留下的东西收拾完(测试逻辑的环境恢复), 然后下一个班级, 直到整个学院都结束辅导, 收拾一下学院留下来的东西(测试集的环境恢复)。。。

1. 设计说明

该测试框架的设计主要包含2个部分

1. 测试逻辑的类接口设计:
2. 测试集类

|  |
| --- |
| /// <summary>  /// 测试集接口  /// </summary>  public interface ITestCluster  {  /// <summary>  /// 返回测试框架类  /// </summary>  Core GtrCore { get; }  /// <summary>  /// 测试集的初始化工作  /// </summary>  /// <returns></returns>  bool SetupScripts();  /// <summary>  /// 测试集的恢复环境工作  /// </summary>  /// <returns></returns>  bool RestoreScripts();  } |

1. 测试逻辑接口

|  |
| --- |
| /// <summary>  /// 测试逻辑  /// </summary>  public interface ILogic  {  /// <summary>  /// 返回测试框架  /// </summary>  Core GtrCore { get; }  /// <summary>  /// 返回测试逻辑所在的程序集  /// </summary>  Assembly CurrAssembly { get; }  /// <summary>  /// 测试逻辑所在的测试集  /// </summary>  ITestCluster iTestCluster { get; set; }  /// <summary>  /// 测试逻辑的类型  /// </summary>  Type TestClusterType { get; }  /// <summary>  /// 测试逻辑的测试数据  /// </summary>  /// <returns></returns>  Dictionary<string, string>[] Data();  /// <summary>  /// 测试逻辑的预处理  /// </summary>  /// <returns></returns>  bool SetupScripts();  /// <summary>  /// 测试逻辑  /// </summary>  /// <param name="data"></param>  void AwLogic(Dictionary<string, string> data);  /// <summary>  /// 测试逻辑每次执行完之后进行恢复环境处理  /// </summary>  /// <param name="data"></param>  void RestoreEnv(Dictionary<string, string> data);  /// <summary>  /// 当前测试逻辑执行完之后进行恢复处理  /// </summary>  /// <returns></returns>  bool RestoreScripts();  } |

1. 测试逻辑的详细信息类

|  |
| --- |
| /// <summary>  /// 测试逻辑的详情信息  /// </summary>  public class LogicInfo : IComparable  {  /// <summary>  /// 当前测试逻辑  /// </summary>  private ILogic Logic = null;  /// <summary>  /// 测试逻辑详细信息的构造方法  /// </summary>  /// <param name="\_Logic"></param>  public LogicInfo(ILogic \_Logic)  {  this.Logic = \_Logic;  }  /// <summary>  /// 返回当前的测试逻辑  /// </summary>  public ILogic logic { get { return this.Logic; } }  /// <summary>  /// 比较当前测试逻辑所在的测试集是否与另外一个测试逻辑的测试集相同  /// </summary>  /// <param name="obj"></param>  /// <returns></returns>  public int CompareTo(object obj)  {  var p = obj as LogicInfo;  if (p == null)  throw new Exception("obj not Type of LogicInfo");  // 是否logic.iTestCluster.GetType().FullName一样  //似乎这里的意思是 判断是否继承于相同的模板.  return this.logic.TestClusterType.FullName.CompareTo(p.logic.TestClusterType.FullName);  }  } |

1. 测试逻辑模板类

|  |
| --- |
| /// <summary>  /// 测试模板类  /// </summary>  /// <typeparam name="T">该测试逻辑所属的测试集</typeparam>  public abstract class LogicBase<T> : ILogic where T : TestCluster  {  /// <summary>  /// 返回当前的自动化框架  /// </summary>  public Core GtrCore  {  get { return Core.GetInstance(); }  }  /// <summary>  /// 测试逻辑所在的程序集  /// </summary>  public Assembly CurrAssembly  { get { return Assembly.GetAssembly(this.GetType()); } }  /// <summary>  /// 测试逻辑所属的测试集  /// </summary>  public ITestCluster iTestCluster { get; set; }  /// <summary>  /// 测试逻辑所属的测试集的类型  /// </summary>  public Type TestClusterType { get { return typeof(T); } }  /// <summary>  /// 测试逻辑的测试数据  /// </summary>  /// <returns></returns>  public abstract Dictionary<string, string>[] Data();  /// <summary>  /// 测试逻辑的初始化  /// </summary>  /// <returns></returns>  public virtual bool SetupScripts()  {  return true;  }  /// <summary>  /// 测试逻辑的环境恢复  /// </summary>  /// <returns></returns>  public virtual bool RestoreScripts()  {  return true;  }  /// <summary>  /// 测试逻辑  /// </summary>  /// <param name="data"></param>  public virtual void AwLogic(Dictionary<string, string> data) { }  /// <summary>  /// 测试逻辑在每次执行完测试数据之后恢复环境  /// </summary>  /// <param name="data"></param>  public virtual void RestoreEnv(Dictionary<string, string> data) { }  } |

1. 自动化框架类设计
2. 自动化框架类

|  |
| --- |
| /// <summary>  /// 默认的测试逻辑运行的测试平台  /// 自动化测试框架的核心算法类  /// </summary>  public class Core  {  /// <summary>  /// debug模式的测试逻辑执行  /// </summary>  /// <param name="arrILogic"></param>  /// <returns></returns>  public bool ExecuteLogic(params ILogic[] arrILogic){}  /// <summary>  /// 执行所选的测试逻辑(debug模式)  /// </summary>  /// <param name="arrILogic"></param>  public void LoadTestCase(params ILogic[] arrILogic){}  /// <summary>  /// 通过加载测试逻辑的dll来反射出所有的测试逻辑与测试集  /// </summary>  /// <param name="fileName"></param>  void LoadTestCase(string fileName){}  } |

1. 自动化框架执行流程

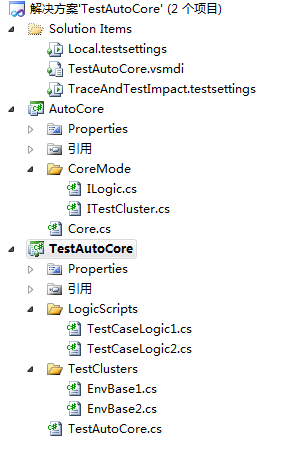
|  |
| --- |
| 加载测试逻辑程序集  将测试逻辑与测试集进行绑定  删选出需要执行的测试用例编号链表  执行测试用例，找到该测试用例所在的测试逻辑单元  测试逻辑所在的测试集是否已经执行  执行测试集的预处理工作  遍历该测试集内的所有测试逻辑  属于测试集的测试逻辑是否已经执行完毕  恢复测试集的环境工作  执行该测试逻辑的环境预处理工作  执行与当前测试逻辑相配套的测试逻辑  恢复测试逻辑环境  当前的测试逻辑的测试数据是否遍历完了  执行与当前测试逻辑的环境恢复工作  标注：绿色表示否  褐色表示是 |

图 1

1. 自动化框架使用说明：

使用自动化框架来开发扩展性强, 可维护性强的用例, 特别要注意”抽象”与”解耦”, 测试集与测试逻辑的解耦, 测试逻辑与测试数据的解耦, 测试逻辑的抽象. 我在这边推荐2种测试用例编写的方式：

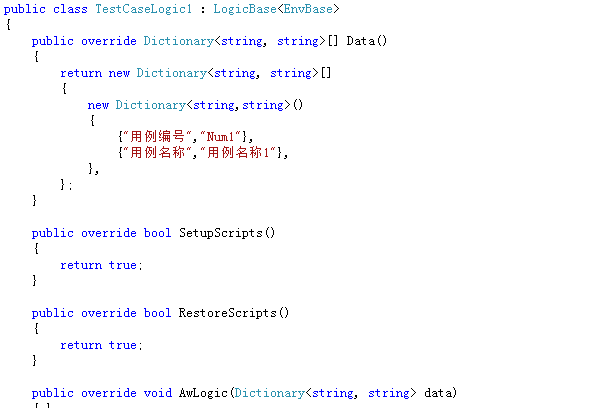
1. 比较土的方式, 一个测试逻辑+N组测试数据



在测试工程下新建LogicScripts(存放测试逻辑的文件)和TestClusters(存放测试集的文件)，将逻辑文件存放在LogicScirpts文件夹下，测试集文件存放在TestCluster文件夹下.

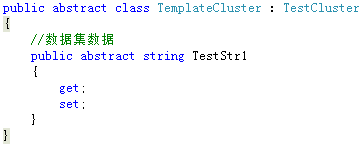
新增的测试用例必须继承于LogicBase<T>模板，其中T为该测试逻辑所属于的测试集.

测试集必须继承于TestCluster类.其中测试数据写入Data()的字典集中.测试逻辑重载Awlogic(dictionary<string,string>)方法.

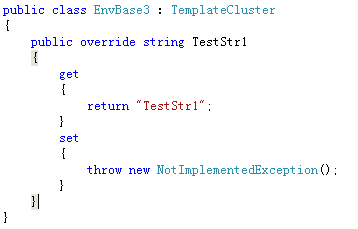


1. 抽象测试逻辑, 使用模板逻辑：一部分测试逻辑具有相同的测试逻辑.

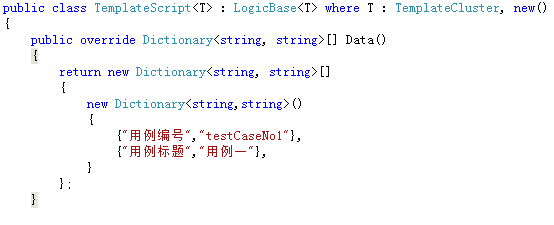
测试集模板



比如说你定义的测试集中需要为TestStr1赋值，那么你可以使用抽象的方式等待真实测试集来重载，如下图:

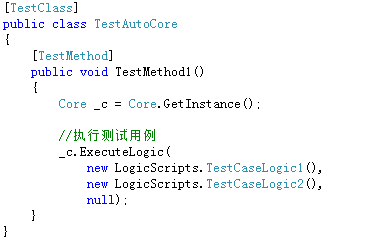


测试逻辑模板:



表明该TemplateScript逻辑与测试末班TemplateCluster绑定，相信你应该懂了测试末班的使用方式了吧，最重要就是抽象测试逻辑.

最后将测试用例丢入框架内执行即可：



1. 作者展望：

AutoCore是轻量级的自动化执行框架，能够很好地批量执行测试用例，能够快速进行持续集成工作，因此作者在此希望大家可以更好地发展该框架，为其增加特性.

2013-07-01