在介绍完 TestNG 基础内容后，接下来我们将介绍如何基于TestNG 进行二次开发，增加部分特性，以实现基于接口测试场景，一定程度解决以下几类问题：

**测试数据与代码耦合程度较高，用例的可扩展性和可复用性较低。**

**测试覆盖度与测试用例维护成本之间的线性关系。**

**缺乏灵活统一的期望比对方式，存在较高的用例开发成本（重复度高）。**

在介绍完 TestNG 基础内容后，接下来我们将介绍如何基于TestNG 进行二次开发，增加部分特性，以实现基于接口测试场景，一定程度解决以下几类问题：

测试数据与代码耦合程度较高，用例的可扩展性和可复用性较低。

测试覆盖度与测试用例维护成本之间的线性关系。

缺乏灵活统一的期望比对方式，存在较高的用例开发成本（重复度高）。

**增强特性**

接下来，将通过几个章节逐一介绍以下特性：

**数据驱动，基于Yaml 实现测试用例与测试数据的解耦。**

参数化，支持指定参数项的参数化，实现参数的排列组合、顺序组合，为低维护成本下实现高覆盖提供可能，除了能够实现入参的参数化，也支持了对期望结果的参数设置。

提供全局统一且唯一的数据驱动方法，支持根据测试类、测试方法在指定目录下自动加载对应的配置文件。

提供全局统一且唯一的测试用例入参。

提供统一灵活的结果比对方法，支持JsonObject、JsonArray、String、Integer等基本数据类型的精确、模糊比较。

**本章节主要介绍在保持全局唯一的数据驱动方式，实现基于Yaml 实现测试用例与测试数据的解耦。**

**TestNG 数据驱动特性设计**

我们使用YAML文件作为测试数据存储的载体，YAML语言的设计参考了JSON，XML和SDL等语言。YAML 强调以数据为中心，简洁易读，编写简单，YAML基本格式要求，如下：

大小写敏感

通过缩进表示层级关系

禁止使用tab缩进，只能使用空格键

缩进的空格数目不重要，只要相同层级左对齐即可

使用#表示注释

**YAML 多文档块特性**

在对某些方法进行测试时，通常会使用不同的数据对方法进行覆盖，如边界值测试，YAML 多文档块（“---”）实现了在一个yaml中，隔离不同测试数据的目的。

---

# 用例描述

testcase: 验证 XXX 功能是否符合预期

# 参数配置

parameter:

jsonObjecta: {"Id":"1","code":"Connect","name":"连接","sentenceDesc":"","type":"DEVICE","grade":[1,2,3,4]}

# 期望配置

expectResult:

expect: {"Id":"2","code":"Connect","name":"连接","sentenceDesc":"","type":"DEVICE","grade":[1,2,3,4]}

---

# 用例描述

testcase: 验证 XX 功能是否符合预期

# 参数配置

parameter:

jsonObjecta: {"Id":"3","code":"Connect","name":"连接","sentenceDesc":"","type":"DEVICE","grade":[1,2,3,4]}

# 期望配置

expectResult:

expect: {"Id":"3","code":"Connect","name":"连接","sentenceDesc":"","type":"DEVICE","grade":[1,2,3,4]}

**全局统一且唯一的数据驱动方法设计**

package framework.factory;

import java.lang.reflect.Method;

import org.testng.annotations.DataProvider;

public abstract class AbstractAiTestFramework {

   /\*\*

    \* 定义一个数据驱动类

    \* @return

    \*/

   @DataProvider(name = "TestDataProvider")

   public Object[][] getTestData(Method method) {

       // 利用反射获取类/方法的注解，获取测试数据，进行测试数据装配

       return DataProviderFactory.assembleDataProvider(this.getClass(), method);

  }

}

如上， 实现 AbstractAiTestFramework 类提供的统一数据驱动方法（TestDataProvider），该方法根据实现根据类名、方法名在指定yaml目录下加载对应的yaml配置文件中测试数据，然后通过Map<String, Object> parameter 参数传递给待测方法，同时当无其对应的yaml文件时，会自动创建该文件。

TestDataProvider 根据yaml中多文档快的数据隔离的特性，使用不同配置块（测试数据）依次驱动测试，实现数据驱动，。

Yaml 测试数据配置 demo 如下，其中"---"实现了配置的隔离：

---

# 用例描述

testcase: 验证 XXX 功能是否符合预期

# 参数配置

parameter:

jsonObjecta: {"Id":"1","code":"Connect","name":"连接","sentenceDesc":"","type":"DEVICE","grade":[1,2,3,4]}

# 期望配置

expectResult:

expect: {"Id":"2","code":"Connect","name":"连接","sentenceDesc":"","type":"DEVICE","grade":[1,2,3,4]}

---

# 用例描述

testcase: 验证 XX 功能是否符合预期

# 参数配置

parameter:

jsonObjecta: {"Id":"3","code":"Connect","name":"连接","sentenceDesc":"","type":"DEVICE","grade":[1,2,3,4]}

# 期望配置

expectResult:

expect: {"Id":"3","code":"Connect","name":"连接","sentenceDesc":"","type":"DEVICE","grade":[1,2,3,4]}

其中TestDataProvider的 DataProviderFactory.assembleDataProvider 核心实现方法如下：

package framework.factory;

​

import java.io.File;

import java.io.FileInputStream;

import java.lang.reflect.Method;

import java.text.MessageFormat;

import java.util.ArrayList;

import java.util.List;

​

import com.alibaba.fastjson.JSON;

import com.alibaba.fastjson.JSONObject;

​

import framework.utils.CreateYamlDemo;

import framework.utils.GetFilesUtils;

import org.apache.commons.lang3.StringUtils;

import org.yaml.snakeyaml.Yaml;

​

import static framework.utils.Parameterization.parameterComposition;

​

public class DataProviderFactory {

​

   /\*\*

    \* @Description: 测试用例的相对路径

    \*/

   private static String USER\_CASE\_DATA\_PATH = "src/test/yaml/";

​

   /\*\*

    \* @Description: 组装dataProvider，以配置块为单位，单个配置块作为一次数据驱动，二维数组中一个元素为一个配置块。

    \* @Param: [testClass, method]

    \* @return: java.lang.Object[][]

    \*/

   public static Object[][] assembleDataProvider(Class<?> testClass, Method method) {

​

       // 获取所有的用例文件

       List<String> useCaseFileNames = extractUseCaseFileNames(testClass, method);

       if (useCaseFileNames.size() < 1) {

           throw new RuntimeException(MessageFormat.format("测试数据缺失, 测试类className={0},测试方法 methodName={1}",

               testClass.getName(), method.getName()));

      }

       Yaml yaml = new Yaml();

       // 各yaml文件中所有yaml配置块列表

       List<Object> yamlBlocks = new ArrayList<Object>();

       // 遍历所有yaml文件

       for (String useCaseFileName : useCaseFileNames) {

           try {

               File file = new File(useCaseFileName);

               if (file.exists()) {

                   Iterable<Object> objIterable = yaml.loadAll(new FileInputStream(file));

                   for (Object object : objIterable) {

                       String objectString = JSON.toJSONString(object);

                       JSONObject jsonObject = JSONObject.parseObject(objectString);

                       if (jsonObject.containsKey("parameterization")){

                           ArrayList<JSONObject> jsonObjectArrayList = parameterComposition(jsonObject);

                           yamlBlocks.addAll(jsonObjectArrayList);

                      }else {

                           yamlBlocks.add(jsonObject);

                      }

                  }

              } else {

                   // 不存在则创建，提高测试数据文件创建效率，避免手动创建

                   CreateYamlDemo.demo(useCaseFileName);

              }

          } catch (Exception e) {

               e.printStackTrace();

          }

      }

       // 定义一个二维数组，长度为配置块个数，一个配置块在二维数组中作为一个元素

       Object[][] result = new Object[yamlBlocks.size()][];

       // 填充二维数组

       for (int n = 0; n < yamlBlocks.size(); n++) {

           List<Object> tmp = new ArrayList<Object>();

           tmp.add(yamlBlocks.get(n));

           result[n] = tmp.toArray();

      }

​

       return result;

  }

测试用例代码示例，如下：

package frameworkTest;

​

import framework.factory.AbstractAiTestFramework;

import org.testng.Assert;

import org.testng.annotations.Test;

import java.util.Map;

​

public class CompareJsonTest extends AbstractAiTestFramework {

   @Test(dataProvider = "TestDataProvider")

   public void compareJsonObjectTest1(Map<String, Object> parameter){

Assert.assertNotNull();

  }

   @Test(dataProvider = "TestDataProvider")

   public void compareJsonObjectTest2(Map<String, Object> parameter){

Assert.assertNotNull();

  }

}

**配置获取方式**

**wady** 通过 com.alibaba.fastjson.JSONPath 以 "$.parameter.jsonObjecta" 形式灵活获取配置中具体的内容，如下：

package frameworkTest;

import com.alibaba.fastjson.JSONArray;

import com.alibaba.fastjson.JSONObject;

​

import com.alibaba.fastjson.JSONPath;

import framework.base.CompareBaseResultDTO;

import framework.factory.AbstractAiTestFramework;

import framework.utils.CompareJsonUtils;

​

import org.testng.Assert;

import org.testng.annotations.Test;

​

import java.util.Map;

​

public class CompareJsonTest extends AbstractAiTestFramework {

​

​

   @Test(dataProvider = "TestDataProvider")

   public void compareJsonObjectTest(Map<String, Object> parameter){

       JSONObject paramsObj = new JSONObject(parameter);

​

       // 获取参数

       JSONObject methodParameter = (JSONObject) JSONPath.eval(paramsObj,"$.parameter.jsonObjecta");

​

       // 方法调用,此处省略. 假设 jsonObject1 同样作为方法返回结果.

​

       // 获取期望结果

       JSONObject expectResult = (JSONObject) JSONPath.eval(paramsObj,"$.expectResult.expect");

​

​

       // 统一的结果比对接口, 根据配置实现即可灵活选择、过滤比对方式 及精确、模糊校验角度.

       CompareBaseResultDTO compareBaseResultDTO

           = CompareJsonUtils.compareJson(methodParameter, expectResult, paramsObj);

​

       // 结果断言

       Assert.assertEquals(compareBaseResultDTO.getRetCode(), 0,

           String.valueOf(compareBaseResultDTO.getRetValue()));

  }

​

}