知乎 首发于 软件测试开发成长之路

Junit5 + YAML 轻松实现参数化和数据驱动，让 App 自动化测试更高效(二)

测吧
已认证的官方帐号

29 人赞同了该文章

> 本文为霍格沃兹测试学院优秀学员课程学习笔记，想一起系统进阶的同学文末加群交流。

上篇文章提到了数据驱动可以在几个方面进行：

- 测试数据的数据驱动
- 测试步骤的数据驱动
  - 定位符
  - 行为流
- 断言的数据驱动

下面将详细解说如何进行数据驱动。

5. 数据驱动

5.1 测试数据的数据驱动

5.1.1 Junit5的 参数化

> 说到测试数据的数据驱动，就必然离不开测试框架的参数化，毕竟测试数据是传给用例的，用例是由框架来管理的，这里以目前最推荐使用的Junit5框架为例，介绍参数化的使用

@ParameterizedTest+@ValueSource参数化

在Junit5中，提供了 `@ParameterizedTest` 注解来实现方法的参数化设置，另外 `@ValueSource` 注解用来存放数据,写法如下：

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
void palindromes(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}
```

@ParameterizedTest+@CsvSource参数化

Junit5还提供了 `@CsvSource` 注解来实现 `csv` 格式的参数传递，写法如下：

```
@ParameterizedTest
@CsvSource({
        "滴滴,滴滴出行",
        "alibaba,阿里巴巴",
        "sougou,搜狗"
})
public void searchStocks(String searchInfo,String exceptName)   {
    String name = searchpage.inputSearchInfo(searchInfo).getAll().get(0);
    assertThat(name,equalTo(exceptName));
}
```

@ParameterizedTest+@CsvFileSourc数据驱动

最终，Junit5提供了 `@CsvFileSourc` 注解来实现csv数据格式的数据驱动，可以传入
来读取数据，写法如下：

```
pdd
xiaomi
pdd
```

- 用例实现：

```
@ParameterizedTest
@CsvFileSource(resources = "/data/SearchTest.csv")
void choose(String keyword){
ArrayList<String> arrayList = searchPage.inputSearchInfo(keyword).addSelected();
}
```

> 对于简单的数据结构，可以使用CSV，上面也说过，较为复杂的数据结构，推荐使用yaml，接下来看如何用yaml文件完成测试数据驱动。

@ParameterizedTest+@MethodSource参数化

- 先来看Junit5提供的另一个注解—— `@MethodSource` ，此注解提供的方法是我们做测试数据驱动的核心，它可以让方法接收指定方法的返回值作为参数化的入参，用法是在注解的括号中填入数据来源的方法名，具体用法如下:

```
@ParameterizedTest
@MethodSource("stringProvider")
void testWithExplicitLocalMethodSource(String argument) {
    assertNotNull(argument);
}

static Stream<String> stringProvider() {
    return Stream.of("apple", "banana");
}
```

@ParameterizedTest+@MethodSource参数化 + jackson yaml数据驱动

有了 `@MethodSource` 的参数化支持，我们就可以在方法中利用 `jackson` 库对 `yaml` 文件进行数据读取，从而完成数据驱动了

- 现有如下yaml数据文件，我需要取出testdata中的测试数据

```
  password: 666
  testdata:
    滴滴: 滴滴出行
    alibaba: 阿里巴巴
    sougou: 搜狗
```

- 创建 Config 类:

```
import java.util.HashMap;

public class Config {
    public String username;
    public String password;
    public HashMap<String,String> testdata = new HashMap<>();
}
```

- 创建 Config 对象，与 yaml 文件建立映射关系，读取数据，通过 @MethodSource 完成数据的参数化传递

```
public class TestSteps {

  @ParameterizedTest
    @MethodSource("YamlData")
    public void search(String searchInfo,String exceptName)   {
        String name = searchpage.inputSearchInfo(searchInfo).getAll().get(0);
        assertThat(name,equalTo(exceptName));
    }

    static Stream<Arguments> YamlData() throws IOException {
    ObjectMapper mapper = new ObjectMapper(new YAMLFactory());
    Config data = mapper.readValue(Config.class.getResourceAsStream("/demo2.yaml"), Co
    List<Arguments> list = new ArrayList<>();
    Arguments arguments = null;
    for (String key : data.testdata.keySet()) {
        Object value = data.testdata.get(key);
        arguments = arguments(key, value);
        list.add(arguments);
    }
    return Stream.of(list.get(0),list.get(1),list.get(2));
   }
```

5.2 测试步骤的数据驱动

对于测试步骤的数据驱动主要针对两点：

- 定位符：
  我们做App自动化的时候可以把定位符合定位器直接写在PO中，也可以将其剥离出来，写在类似yaml的文件中，定义好格式个对象的映射关系即可完成定位符的数据驱动。
- 行为流：
  与定位符的剥离思想一致，行为流原本也是写在PO中的各个方法，这些行为流和定位符是紧密关联的，因此也可以剥离出来，和定位符在一起组成测试步骤的数据驱动。

好比下面这样的，以雪球App的搜索场景为例：

```
public class SearchPage extends BasePage{
  //定位符
    private By inputBox = By.id("search_input_text");
    private By clickStock = By.id("name");
    private By cancel = By.id("action_close");
  //行为流
  //搜索股票
    public SearchPage search(String sendText){
```

```
            return this;
        }
    //取消返回
    public App cancel(){
            click(cancel);
            return new App();
        }
}
```

> 注：测试步骤的数据驱动是指把PO中变化的量剥离出来，不是对用例里的调用步骤进行封装。在上面已经提到过不要在测试用例内完成大量的数据驱动：用例通过PO的调用是能够非常清晰展现出业务执行场景的，业务才是用例的核心；
>
> 一旦在用例里使用了大量数据驱动，如调用各种 yaml、csv 等数据文件，会造成用例可读性变差，维护复杂度变高；

5.2.1 设计思路

首先来考虑我们的剥离到 `yaml` 中的数据结构

- 做测试步骤的数据局驱动我们希望可以将一个用例中的步骤方法清晰的展示出来，在对应的方法中包括了方法对应的定位符和行为流，这样能和PO中的结构保持一致，更易读易维护；如下：

```
search:
    steps:
      - id: search_input_text
        send: pdd
      - id: name
cancel:
    steps:
      - id: action_close
```

- 另外我们还要考虑扩展性，之前提到了还有测试断言的数据驱动，另外还有一点没提到的是，框架的健壮程度还要考虑被测系统(Android，IOS)的通用性、版本变更、元素定位符的多样性等。这样考虑的话就应该有多个分类，一个分类中包含了PO中的所有方法，一个分类中包含了版本、系统等信息等,如下(`SearchPage.yaml`)：

```
#方法
methods:
```

```
      - id: search_input_text
        send: pdd
      - id: name
  cancel:
  steps:
    - id: action_close

#定位符对应系统、版本信息
elements:
  search_input_text:
    element:
    ...

#断言
asserts:
  search:
    assert:
    ...
  cancel:
    assert:
    ...
```

- 按照上述的思路，以搜索步骤为例，我们需要一个 Model 类，用来映射不同的数据模块(方法、版本、断言)，对不同的模块需要一一对应的类，类的成员变量结构与yaml文件中的结构保持一致：

1）创建 PageObjectModel 类

```
import java.util.HashMap;
public class PageObjectModel {
  public HashMap<String, PageObjectMethod> methods = new HashMap<>();
    public HashMap<String, PageObjectElement> elements = new HashMap<>();
  public HashMap<String, PageObjectAssert> asserts = new HashMap<>();
}
```

2）创建对应数据模块的类 PageObjectMethod

```
public class PageObjectMethod {
    public List<HashMap<String, String>> getSteps() {
        return steps;
```

```
    public void setSteps(List<HashMap<String, String>> steps) {
        this.steps = steps;
    }

    public List<HashMap<String,String>> steps = new ArrayList<>();
}
```

3）实现解析 yaml 数据的方法，完成 PO 中行为流的封装；

- 首先按照之前介绍过的通过 jackson 来解析 yaml 数据，我们需要文件的 地址 ，另外我们还需要知道当前执行的 方法 ，用来去 yaml 中取方法对应的 定位符 和 行为流 ，所以初步设想应该有 method 和 path 两个参数：

```
public void parseSteps(String method,String path){
        ObjectMapper mapper = new ObjectMapper(new YAMLFactory());
        try {
            PageObjectModel model = mapper.readValue(BasePage.class.getResourceAsStrea
            parseStepsFromYaml(model.methods.get(method));
        }catch (IOException e) {
            e.printStackTrace();
        }
    }
```

- 上面的方法中可以看到调用了一个 parseStepsFromYaml 方法，这个方法是将从 yaml 中获取到的数据进行处理，拿到对应 方法 的 定位符 再拿到定位符紧跟的 行为流 完成对应的 操作步骤 (点击、输入、获取属性等)；之所以将这个方法单独抽离出来，是因为后面会对 parseSteps 重载，方便复用，后面会介绍到。
  如下：我们要通过 methods 里的 search 方法拿到对应的步骤 steps 里的 id ,在根据 id 下的 send 值进行输入操作

```
methods:
search:
 steps:
  - id: search_input_text
    send: pdd
 - id: name
```

```
        steps.getSteps().forEach(step ->{
            WebElement element = null;
            if (step.get("id") != null){
                element = findElement(By.id(id));
            }else if (step.get("xpath") != null){
                element = findElement(By.id(step.get("xpath")));
            }else if (step.get("aid") != null){
                element = findElement(MobileBy.AccessibilityId(step.get("aid")));
            if (step.get("send") != null){
                element.sendKeys(step.get("send"));
            }else if (step.get("get") != null){
                  findElement(by).getAttribute(get);
            }
            else {
                element.click();  //默认操作是点击
            }
        });
    }
```

4）这个时候再回到我们的PO里，就变成了这个样子,看一下PO是不是一下子变得简洁了许多：

```
public class SearchPage extends BasePage{
  //行为流
  //搜索股票
    public SearchPage search(String sendText){
        parseSteps("search","/com.xueqiu.app/page/SearchPage.yaml");
        return this;
    }
  //取消返回
  public App cancel(){
      parseSteps("cancel","/com.xueqiu.app/page/SearchPage.yaml");
      return new App();
    }
}
```

到这里，测试步骤的数据驱动算是完成了一个基本模板，还有很多可以优化的地方，比如上面的 `SearchPage` 的 `PO` 中， `parseSteps` 的两个参数 `method` 和 `path` 都是有规律可循的：

- `method` 和当前执行的方法名是定义好保持一致的
- 当前 `PO` 所对应的 `yaml` 文件的 `path` 是固定的

下面针对这个点做个小优化

> 这里将会对上一节中的 parseSteps 方法进行优化，减少重复性工作。

- 先来解决方法名 `method` 的问题，来看Thread的一个方法：
  `Thread.currentThread().getStackTrace()` 利用这个方法可以打印出当前方法执行的全部过程，写单测来验证,将每一步的方法名都打印出来：

```
void testMethod(){
        Arrays.stream(Thread.currentThread().getStackTrace()).forEach(stack ->{
            System.out.println(stack.getMethodName());
        });
        System.out.println("当前调用我的方法是："+Thread.currentThread().getStackTrace()
    }

@Test
void getMethodName(){
    testMethod();
    }
```

执行结果：

```
getStackTrace
testMethod   //当前执行的方法
getMethodName  //调用testMethod的方法
invoke0
invoke
invoke
invoke
invokeMethod
proceed
//...这里省略中间很多不重要的部分
execute
execute
startRunnerWithArgs
startRunnerWithArgs
prepareStreamsAndStart
main
当前执行的方法是：getMethodName
```

- 再来解决 yaml 文件路径的 path 参数，这里可以借助 java.lang.Class.getCanonicalName() 方法，此方法可以返回当前类名，包括类所在的包名，如下：

```
@Test
void getPath(){
    System.out.println(this.getClass().getCanonicalName());
}

//打印结果
com.xueqiu.app.testcase.TestSteps
```

- 稍加改造就可以变成地址信息：

```
@Test
void getPath(){
    System.out.println(this.getClass().getCanonicalName());
    String path = "/com.xueqiu.app" + this.getClass().getCanonicalName().split("app")[
    System.out.println(path);
}
```

打印结果：

```
com.xueqiu.app.testcase.TestSteps
/com.xueqiu.app/testcase/TestSteps.yaml
```

这样我们就将当前类的信息转变成了一个地址信息，后面我们只需要将对应的 yaml 文件以和类 相同的命名 ， 相同路径结构 存放在 resources 目录下即可

- 现在 method 和 path 参数的问题都解决了，在来看现在的 parseSteps 方法：

```
//解析步骤
public void parseSteps(String method) {
    ObjectMapper mapper = new ObjectMapper(new YAMLFactory());
    String path = "/com.xueqiu.app" + this.getClass().getCanonicalName(
    try {
```

```
    }catch (IOException e) {
        e.printStackTrace();
    }
}

public void parseSteps(){
    String method = Thread.currentThread().getStackTrace()[2].getMethodName();
    parseSteps(method);
}
```

- 此时再次回到 SearchPage 的PO中，可以看到更加的简洁了，甚至变成了“傻瓜操作”：

```
public class SearchPage extends BasePage{

    public SearchPage search(){
        parseSteps();
        return this;
    }

    public App cancel(){
        parseSteps();
        return new App();
    }
}
```

send参数化处理

- 看似好像大功告成，又出现了新的问题，不知道大家注意到没有， search 方法其实是需要 send 值的，而现在我们的 send 值是写死在 yaml 中的，这反而违背了我们参数化和数据驱动的原则：

```
methods:
  search:
    steps:
      - id: search_input_text
        send: pdd  #send的内容被写死在了这里
      - id: name
```

- 所以我们需要继续解决这个问题，将 send 的值进行 参数化

1） 既然是参数化，那就要把send的值变成参数，这里用 $sendText 来表示是参数

```
methods:
  search:
    steps:
      - id: search_input_text
  #      send: pdd
        send: $sendText #表示参数化
      - id: name
```

2）在 search 方法中使用 HashMap 将用例传递过来的测试数据保存至其中，用来传递到 parseSteps 步骤解析方法中。

```
public SearchPage search(String sendText){
    HashMap<String,Object> map = new HashMap<>();
    map.put("sendText",sendText);
    setParams(map);
    parseSteps();
    return this;
}
```

3）再在 parseSteps 方法所处的类中添加 HashMap 类型的 params 变量，用来接收PO传过来的 sendText 测试数据

```
private static HashMap<String,Object> params = new HashMap<>();

public HashMap<String, Object> getParams() {
    return params;
}
//测试步骤参数化
public void setParams(HashMap<String, Object> params) {
    this.params = params;
}
```

4）最后修改 parseStepsFromYaml 方法中的 send 值获取方式，将占位的参数 $sendText 替换成实际传递过来的测试数据 sendText

```
if (step.get("send") != null){
        String send = step.get("send").replace("$sendText",params.get("
        element.sendKeys(send);
```

getAttribute实现

> 在文章前面提到过获取元素属性，在自动化测试过程中，经常要获取元素属性来作为方法的返回值，以供我们进行其他操作或断言，其中text是我们最常获取的属性，下面来实现此方法的数据驱动

在上面的搜索股票场景下，加上一步获取股票的价格信息

- 先看一下思路，按照之前的设计，在 `yaml` 中的定位符后面跟着的就是行为流，假定有一个 `getCurrentPrice` 方法，通过 `get text` 来获取 `text` 属性，写法如下：

```
getCurrentPrice:
    steps:
      - id: current_price
        get: text
```

- 这个时候就可以在 `parseStepsFromYaml` 方法中加入属性获取的解析逻辑,通过 `get` 来传递要获取的属性

```
if (step.get("send") != null){
        String send = step.get("send").replace("$sendText",params.get("sendText").toSt
        element.sendKeys(send);
}else if (step.get("get") != null){
        String attribute = element.getAttribute(step.get("get"));
                }
```

- 接着我们到 `SearchPage` 的 `PO` 中实现 `getCurrentPrice` 方法，这个时候就会发现一个问题：

```
public Double getCurrentPrice(){
        parseSteps();
       // return ???;
       }
```

没错， `text` 属性获取到了，可以没有回传出来， `getCurrentPrice` 方法没有retu
将 `parseStepsFromYaml` 获取到的属性值通过一个 “中间商 "给传递到 `getCurrentP`

- 语言描述比较晦涩，下面我以一个市场供需买卖的场景来说明整个设计流程：

1）产生 市场需求 ， yaml 中定义好数据结构

```
methods:
  search:
    steps:
      - id: search_input_text
        send: $sendText
      - id: name

  getCurrentPrice:
    steps:
      - id: current_price
        get: text
        dump: price

  cancel:
    steps:
      - id: action_close
```

2） 实现 “中间商” ， 这个 “中间商” 就是一个 HashMap ， 将它取名为 result

```
private static HashMap<String,Object> result = new HashMap<>();

//测试步骤结果读取
public static HashMap<String, Object> getResult() {
```

3) 供应商 根据 市场需求 产生 产品 并提供给 中间商 ， 获取 属性 并将 属性值 存入 result

```
if (step.get("send") != null){
        String send = step.get("send").replace("$sendText",params.get("sendText").toSt
        element.sendKeys(send);
}else if (step.get("get") != null){
        String attribute = element.getAttribute(step.get("get"));
        result.put(step.get("dump"),attribute);
                }
```

4) 消费者 根据自己的 需求 去 中间商 那里拿到 商品 ， 从 result 中 get 到 price 的值

```
public Double getCurrentPrice(){
    parseSteps();
    return Double.valueOf(getResult().get("price").toString());
}
```

这样就成功完成了这个交易场景的闭环，股票价格 price 被成功返回至用例中

5.3 断言的数据驱动

> 有了上面的铺垫，断言的数据驱动就显得简单多了，我个人有时候也简单的把它归为测试数据的驱动中

- 因为每个测试数据在传入用例跑完后，都会对应有断言来进行结构判定，因此将测试数据对应的断言数据在一个 yaml 文件中，写入一个数组里，再同测试数据一起获取传入到用例中

```
-
 - didi
 - 100d
-
 - alibaba
 - 120d
-
```

- 回到最初的测试数据数据驱动，把数据获取传入

```
@ParameterizedTest
@MethodSource("searchYamlData")
void search(String searchInfo,String exceptPrice ){
    Double currentPrice = searchPage.search(searchInfo).getCurrentPrice();
    assertThat(currentPrice,greaterThanOrEqualTo(Double.parseDouble(exceptPrice)));
}

static Stream<Arguments> searchYamlData() throws IOException {
    Arguments arguments = null;
    List<Arguments> list = new ArrayList<>();
    ObjectMapper mapper = new ObjectMapper(new YAMLFactory());

    String path1 = "/com.xueqiu.app" + TestSearch.class.getCanonicalName().split("app"
    Object[][] searchData = mapper.readValue(TestSearch.class.getResourceAsStream(path
    for (Object[] entrySet : Arrays.asList(searchData)){
        String key = Arrays.asList(entrySet).get(0).toString();
        String value =  Arrays.asList(entrySet).get(1).toString();
        arguments = arguments(key,value);
        list.add(arguments);
    }
    return Stream.of(list.get(0),list.get(1),list.get(2));
}
```

注：其实这里应该说还是测试数据驱动，并不能算是断言的驱动，真正想做成断言的驱动还需要封装类似测试步骤驱动的形式。目前没有做这层封装，因为在使用中发现断言的类型很多，直接在用例里面写也很方便易读，加上目前时间精力也有限，待后续需要的时候再继续补充~

6. 运行效果

> 说的再多，不如实际跑一下，检验一下框架封装后的实际运行效果

- 用例运行结果：

折腾了这么久，总算是“大功告成”了，之所以加个引号，是因为这个仅仅是个开始，只能算是初具雏形，像文章中提到的被测系统切换、版本切换、多元素查找等都还未实现，后续会持续学习更新。有很多错误或表述不恰当的地方，请大家多指正！

福利福利：

知乎 首发于 **软件测试开发成长之路**

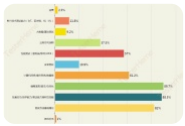
--推荐阅读：

测试开发是什么？为什么现在那么多公司都要招聘测试开发？

软件测试的岗位会越来越少吗？

软件测试真的干到35就干不动了吗？

二十五岁零基础转行做软件测试怎么样？
www.zhihu.com



测吧：软件测试如何获得高薪？
zhuanlan.zhihu.com

发布于 2020-03-18

自动化测试 软件测试 软件测试工程师

文章被以下专栏收录

软件测试开发成长之路
最前沿的技术、最潮流的思想，让程序员的世界变得活色生香~

进入专栏

推荐阅读

软件测试岗位会越来越少吗？

先说结论：软件测试的岗位不会越来越少，但是要求会越来越高。说个比较现实一定的结
可怕，要求越来越高

▲ 赞同 29 ▼

1 条评论 切换为时间排序

写下你的评论...

爱的辉煌 20 天前

没评论，准备入行的菜鸟

赞