组件化case

Component Case，组件化测试用例，借用了vue.js的component概念，希望case是以独立的个体存在，但又具有关联性，同时与自动化脚本又能高度适配或对应。

概念

Component (组件)是vue.js内的一个概念，每一个或多个类似的控件组成一个组件，单一组件具有一致性，仅根据传入数据不同，返回不同的显示效果。

Component Case则具有类似的概念，即每一个用例允许被拆分成有限度的组件，组件与组件之间没有关联；仅根据初始化条件的不同，返回不同的测试执行结果。

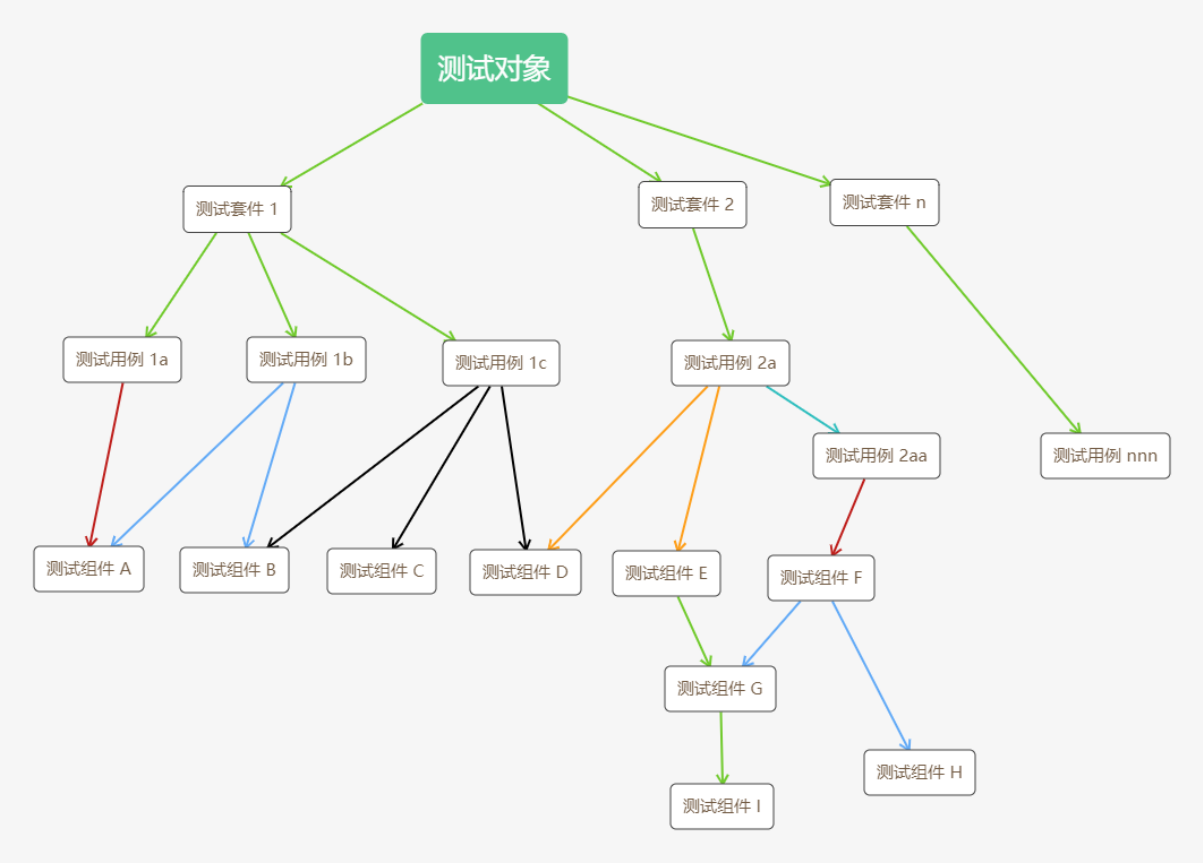
定义

在后续的文档内，仅使用简写，在部分地方使用全称

名称

Aa 名称	全称	简写	含义
测试对象	test object	tobject	被测试的软件或app
测试套件	test suite	suite	针对测试对象内单一功能点的测试操作集合，允许包含多个测试用例
测试用例	test case	case	测试套件的执行实体，由一个或多个测试组件构成；一个测试用例是针对单一功能的测试操作
组件	component case	component	原子化的测试操作，仅包含一个初始条件，一系列具有连续性的测试操作，一个最终结果
步骤	step	step	测试操作的真实描述，具有不可再分割性，只有返回为“真”的操作结果
Untitled			

结构



描述

1. case允许调用component，case，或不调用
2. component只允许调用component，或不调用
3. component是case的极端原子化，只包含
    - 一个初始条件
    - 多个具有连续执行性的测试操作
    - 一个返回结果

> ⚠ 与DDD测试类似，Component Case可以看做DDD版本的case，但与DDD不同的是，Component Case仍追求操作的连贯性，这点又与BDD类似

Component Case也并非是新创的词汇和测试手段，有兴趣的可以参考

https://www.softwaretestinghelp.com/what-is-component-testing-or-module-testing/

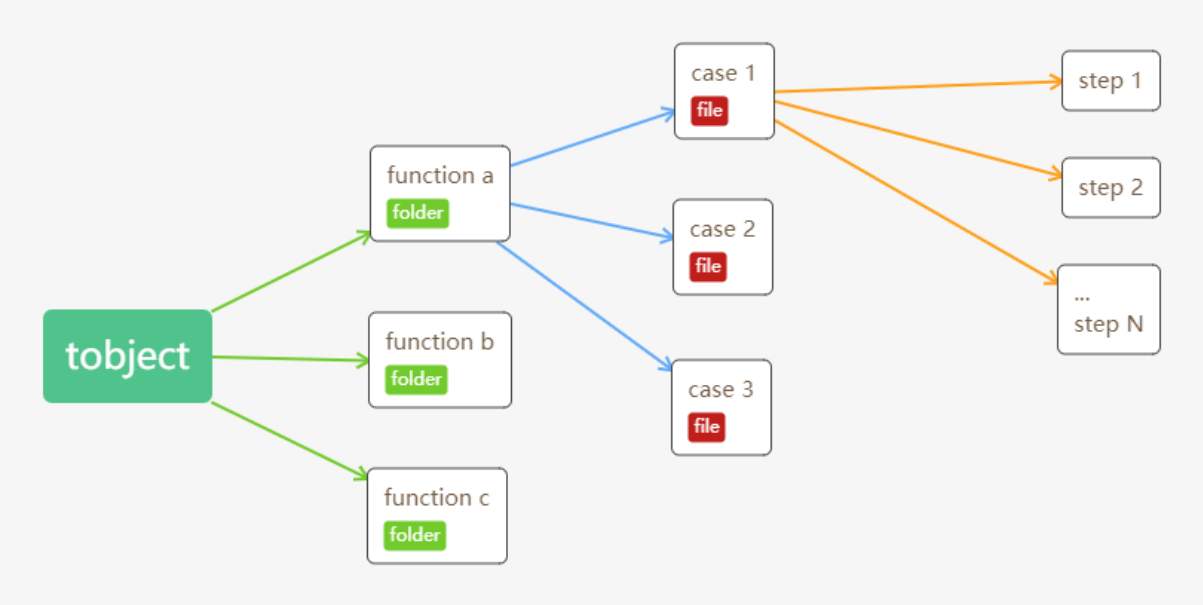
构成与思路

简述

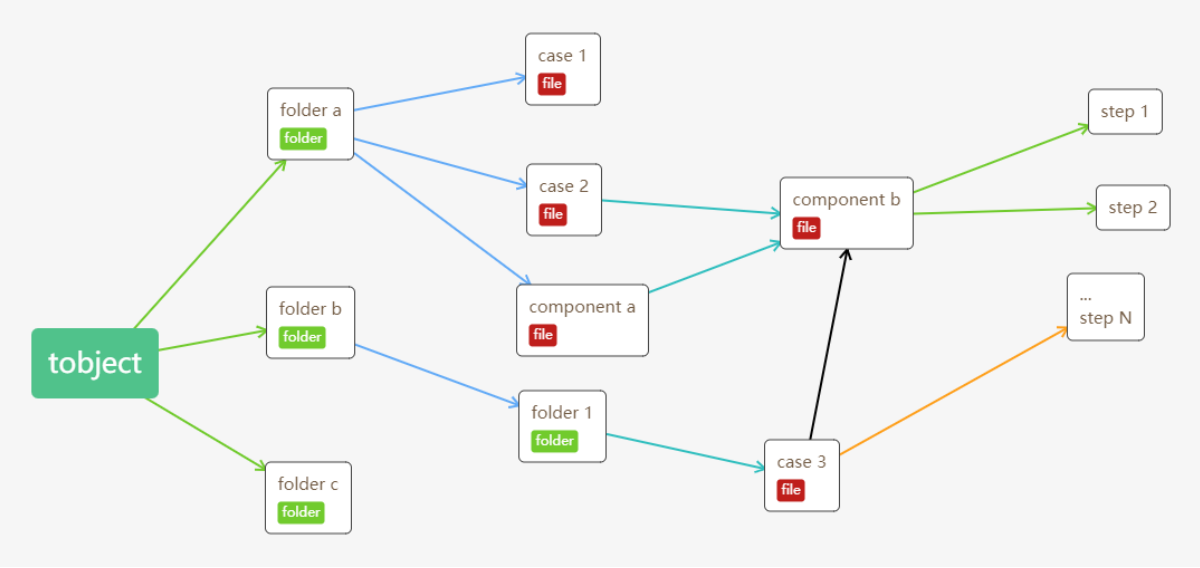
在创建Component Case时，其思路是参考了vue.js的component组件化，因此在实际撰写Component Case理论思路时，也是希望不偏离这个思路

与vue.js一致，Component Case在管理中也应该依赖router(链路)进行，router作为"主干"存在，负责管理和处理所有的component case，原则上与vue.js的router格式接近或一致，即：定义→引用。

在真实的case编写中，往往使用的是文本编辑器，一个单纯的txt，excel或使用一些软件进行管理，但追溯其有效内容时，均是文本内容；在其管理上，也基本是按照功能进行分类，形成了大致的三级管理体系，而实际保存中就形成了文件夹-文件的关系，这种关系是一种强关联，不仅是一种存放关系，更是一种从属关系。



而在Component Case内，传统的文件夹-文件关系只是一种单纯的存放关系，而不在赋予从属关系，suite、case、component都是一个文件，仅依赖其各自的调用，从属关系的影响范围只在各自的文件范围内，最终会形成一种调用结构。



1.各个suite、case、component各自管理各自的子层，且影响范围只是自身,并不具有穿透性
2.各个suite、case、component的关联只在其被调用时才会生成
3.step并不可被调用，因为它是最小原子化状态
4.在内，逻辑关系在文件内定义；在外，文件夹与文件只是纯粹的存储的关系

这种调用关系类似编程语言的调用，类似于这样使用的方式，其调用方式与各类编程语言保持一致。

```
/*
* 文件名: a.js
* case名: caseA
* componet名: component_a
*/

//1.调用a文件的caseA
import caseA from 'filepath/a.js'

//2.使用组件component_a
caseA.component_a();
```

💡 这意味着suite、case、component、step都是一种"类代码"，可以是类、方法、函数、属性、参数；在其使用时，依据规则进行编写和使用，而根据大致的相同点保存在各个文件夹内

为什么要这么做

定义一种新颖的case结构并不是哗众取宠和否定前人的成果，当确定收益大于成本后，我们应该允许自己重新定义新的结构，而非升级、改造原有的大楼。同时Component Case依然是一种老旧的测试概念，本文借用老旧的Component Case概述和理念进而转为新的Component Case理念。

Component Case依然是需要依赖于文本编辑或用例管理软件进行关联，Component Case最大的"新"在于剥离了传统的文件夹-文件依赖，转向将依赖关系赋予文件自身管理。

🔒 在Test Track、TestLink、QC、JIR、等软件内，均可以用Component Case理念去管理你的case，本质上Component Case是一种现有case管理的补充，而非破坏。

在自动化测试日益重要的今天，Component Case页与automation是互补的关系；

- 真实的执行→step
- 执行的集合→component
- 集合的连续使用→case

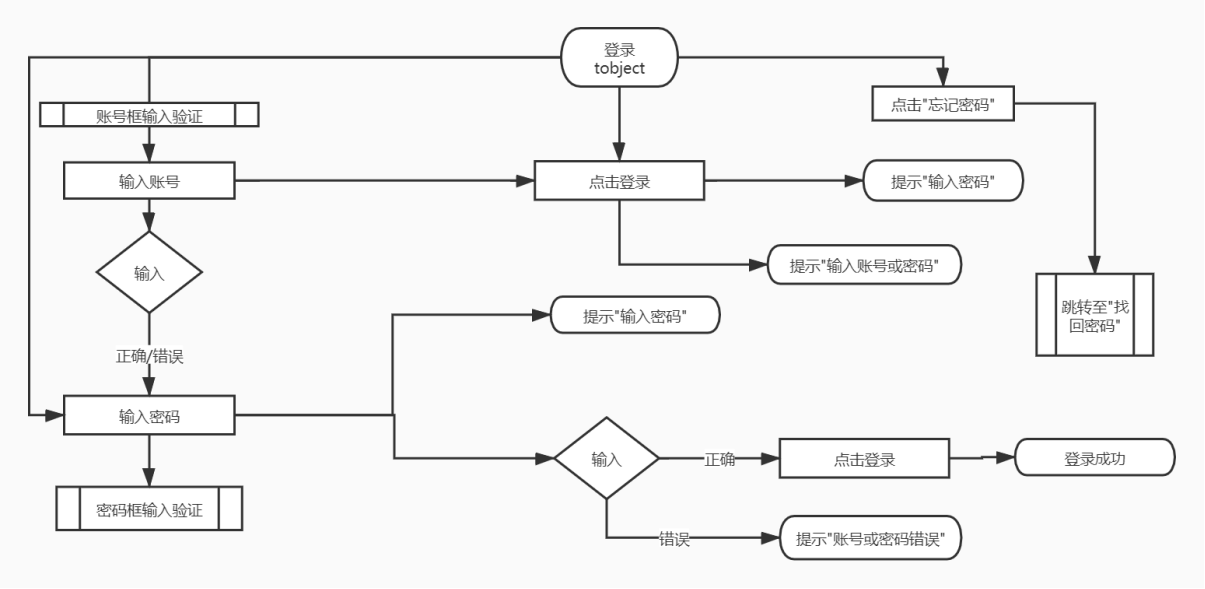
💡 本质来说，Component Case是automation的一种文字描述

这种关系的互补，可以帮助关联case与script的映射，间接的管理更为上层的TR(技术需求文档)与US(客户需求)

简单的例子

以登录百度为例，登录是一个整体的功能，我们测试不仅包含正常登录，还包含异常登录等测试路径。因此作为一个测试实体。

1. "登录"是一个tobject，但是一种"虚态"
2. 所有的测试路径的视为一个suite
3. 每一条测试路径均视为case
4. 每一个case(测试路径)内包含的各类通用步骤，均被视为component
5. 每一个componet中可映射为真实操作的，均被视为step



以上是一个登录的测试流程图，基于流程图编写和描绘case，是一种典型的测试用例编写法，其优点是输入与结果结构很清晰，每一个输入都有对应的输出结果，但最大的限制是它的复杂程度与你的tobject复杂度成正比；而解耦又是一件非常痛苦的事情。它不是主流的测试结构法，但却是测试人员必备的测试结构法。在基于小功能点进行测试时，流程图方法可以很好的帮助测试人员进行测试点归纳。

但Component Case可以参考以上的流程图进行分拆操作；在图中，子流程均可被视为典型的component成员，它们与主流程没有强相关性。而每一个从开始→结束可被视为case，每一个流程可被视为一个component，内部的单个操作可被视为step。

假设整个测试都可被automation，则转为以下代码

```
//引用
import element from '/tobject/element' //元素
import operation from '/tobject/operation' //操作

import assert from '/component/assert' //验证组件
import commoncase from 'case/common' //通用性case

/**
 * 测试套件
 * login
 */
class Suite_Login {
  /**
   * 点击登录，无任何输入值
   * case
   */
  LoginWithoutAny() {
    operation.click(element.login_button) //使用step
    assert(element.error_text, '请输入账号或密码') //调用component
  }
  /**
```

```
 * 账号/密码输入框
 * 输入
 * case
 */
InsertInput() {
  commoncase.assertInput(element.accout) //调用case
  commoncase.assertInput(element.password) //调用case
}

/**
 * 验证账号或密码一处为空
 * case
 */
AssertEmpty() {
  //定义参数
  let para = [
    {
      emptyelement: element.accout,
      clickelement: element.login_button,
      errorelement: element.error_text,
      errortext: '请输入账号'
    },
    {
      emptyelement: element.password,
      clickelement: element.login_button,
      errorelement: element.error_text,
      errortext: '请输入密码'
    }
  ]
  //循环执行
  para.forEach(
    sigele => assert.empty(sigele) //调用component
  )
}

/**
 * 输入错误值登录
 * case
 */
InputError() {
  commoncase.InputError(element.accout) //调用case
  commoncase.InputError(element.password) //调用case
}

/**
 * 跳转到忘记密码
 * case
 */
JumpToForgot() {
  commoncase.JumpToForgot() //调用case
}

/**
 * 登录成功
 * case
 */
EnterRight() {
  operation.input(element.accout, '账号') //使用step
```

```
    operation.input(element.password, '密码') //使用step
    operation.click(element.login_button) //使用step
  }
}
```

在代码内，class是一个suite，每一个function是一个case，在class定义前进行case与component的

- case内允许直接使用step，允许调用case,component
- case并不关心最终结果，最终返回结果由其最后的step, component构成
- case与component之间没有非常明显的界定，component是case的原子化结果
- component相对case来说，会使用更多的参数，由其定义初始化条件，并返回一个为"真"的结果
- component的step较少，只关心其测试操作

那如何写case

前言

以上是Componet Case的理论构成与实际automation应用，在中间环节中，该如何使用？我们依然需要依靠流程图来帮助我们进行case的处理。

首要要明确一点，case的颗粒度需要多细才是一个好的case?

> 💡 case的颗粒度并不是越细小越好，纵然测试case并不需要保证所有人都能看懂，在工业分工精细的现代软件工程内，case只需对测试人员负责! 因此会衍生出如此众多的测试流派

书写case是多个流派融合的过程，针对完整的产品进行测试

- 基于功能点进行case分拆是完美的做法，由你的行为定义你的测试，我们称之为BDT
- 而针对单点功能，往往我们需要混入多数据，此时，数据将构成你的测试行为，我们称之为DDT
- 具体表现在，由BDT界定我们的case，由DDT展现我们的component
- case是单一的测试功能点的集合，我们的行为作为初始化条件会带来不同的测试结果

- component是原子化的测试操作，不同的数据带入，会带来不同的测试结果
- 而在真实的测试过程中，我们很难界定我们处于哪一个流派，我们更应该关注我们的测试需求进而选择正确的测试风格

综上，我们可以确认的是

> 💡 case是BDT的具现形式，component是DDT的具现形式，而两者没有异常清晰的界定；component是case的原子化结果

我们依然使用登录作为例子，因为我们拥有了一张流程图，一份自动化脚本，因此我们可以清晰的定义出Component Case如何操作。

在任何case管理软件内，我们都是可以建立文件夹、文件的，现在，我们清晰的了解文件夹-文件仅仅是文件保存关系，因此我们创建多个文件夹供以我们存放各类case和component文件，而具体的命名规则，我依然建议依据功能点来区分。

▼ componet

▼ assert

输入值验证.file

▼ case

▼ 跳转

跳转.file

▼ 功能

空字符点击登录.file

我们以接近这样的形式来大致区分文件夹和功能点，用以确保大致的功能点是存放在一个文件夹内

解析测试

解析我们的测试是非常重要的一步，它确保整个测试的颗粒度大小，以及参照哪类规则进行测试的分解。

以"登录"为例，整个测试范围是测试"登录"功能，如果以tree来解析，树干是输入正确的账号→输入正确的密码→点击登录→登录成功；而分支则是文本框验证和异常输入、异常操作，每个tree的分支终点就是结果，因此我们解析case的第一步就是提取主干(trunk)。

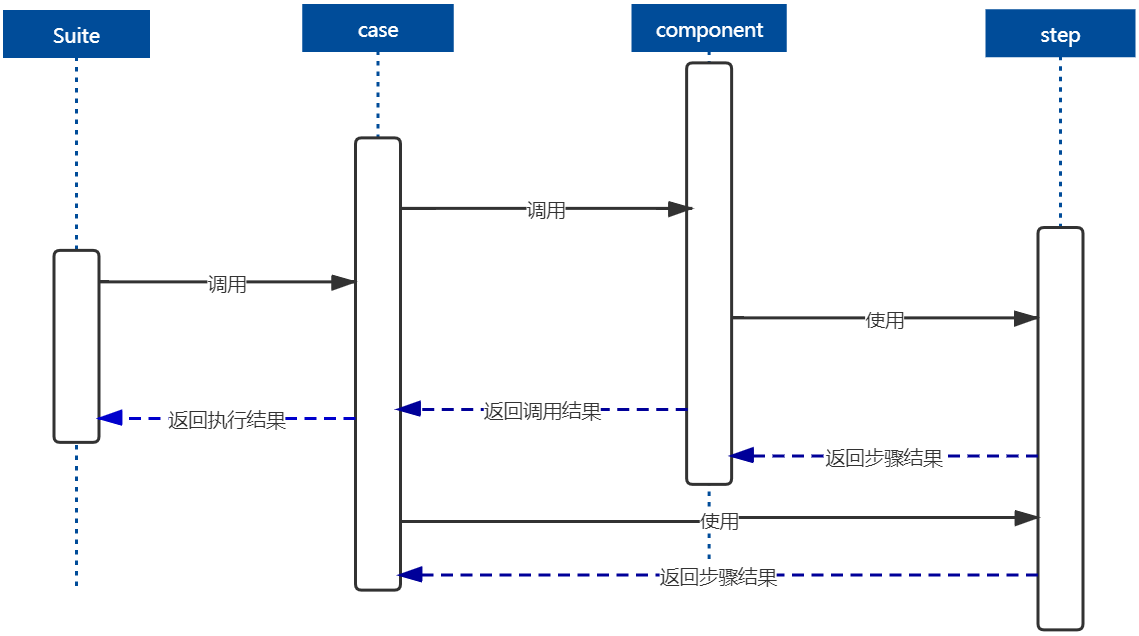
> 📢 我们借用excel的格式来展现我们的思路

case

Sr No	TC Name	Steps to Execute	Expected System Reponose	Actual Result	Pre Condition	Comments	Type
1	启动程序	1.打开APP、Web等，进入程序	打开成功				step
2	控件验证	1.验证账号框输入各类字符2.验证密码框输入各类字符	不符合输入类型或长度时,错误时有提示			data 1->accoutdata 2->password	case(data->component)
3	信息未完整点击登录	1.账号密码均不填写2.只填写账号3.只填写密码	不符合验证条件，错误时有提示				component*3
4	点击"忘记密码"	1.点击忘记密码按钮	跳转至"忘记密码"页面				case

从文档内，我们可以看出Component Case的书写风格，Component Case的要求就是保持简单，在不影响主干case的前提下，尽量将分支作为case、component调用。

如举例说明则是主干是main函数，每一个被调用的case、component都是一个方法，按照顺序执行，当调用case、component时，main函数暂停并等待case、component的结果。



如以上的时序图一般，从suite到step的调用、使用关系，很清晰的看出

- step是底层，它代表的是真实的执行
- component是第二层，它是多个step的集合，也是case的原子化结果；可以看成是固定的操作的流程，在"登录"case中，Sr No 3， 无论填写不完整是哪个控件，它的操作都是

  1. 清空控件
  2. 填入其他控件信息
  3. 点击"登录"按钮

  它的操作是一致的，仅初始化条件和待验证的结果不一致，因此我们可以视为一个整体作为component，仅需带入不同的初始化条件和最终验证即可
- case是第三层，它可以直接使用step，也可以调用case、component；它可以是连续的执行结果；也可以是多个类似操作的集合。在"登录"case中，Sr No 2，它可以看成是一个case的调用。而作为被调用的case，它包含了一个完整的测试内容，且是对不同控件进行的测试操作。它的初始化条件比component更加简单，因为自身是完整的。同时，我们也不需要它是否有返回结果

> 💡 component和case最大的区别在于是否可以原子化

- suite则是总集合，它代表着完整的"登录测试"过程

补充

很多人看到这里还是觉得很迷茫，无法分清component和case的区别和怎么书写case。在我看来其实很简单

> component是case的原子化结果

component关注由component和step构成，它只关注操作本身，即使你的操作在实际执行者看来是"错误"的，但针对操作本身来说，依然是一个"真"的结果。

而我们想要验证component带来的结果是否是预期值时，assert其实发生在component之后，且包裹在case内；例如我们输入错误账号和密码。这个操作的初始化条件是→"错误账号和密码",component执行了整个操作后，我们的assert错误值是发生在case内。而整个case我们并不会影响整体的suite的执行，在遭遇case失败时，我们应该清晰的认识到是流程已经与预期值不再符合，此时的执行已经是没有意义的。

> ⚠ component和case并不要求人们严格区分，但一定要有这样的理念

而关于书写case，首先要做到你真的熟悉测试对象，拥有一份已经较为完整的case且很熟悉，Component Case的精髓在于分拆和解绑耦合，只有熟悉了，才能展开。