

# 每次不重样， 带你收获最新测试技术！

Puppeteer自动化测试简易入门教程（三） .....	1
不会测试算法，借助AI来帮忙？ .....	10
轻松搭建Graphql API 模拟服务器 .....	15
软件测试下的AI之路（6） .....	31
Playwright实战 .....	47
数据迁移测试 .....	64
探索性测试：理论根基、实践精髓与未来趋势 .....	71
测试江湖：漫谈职场中的“卷”“和” 996 ” .....	90



微信扫一扫关注我们

✉ 投稿邮箱：[editor@51testing.com](mailto:editor@51testing.com)

# Puppeteer 自动化测试简易入门教程（三）

◆ 作者：小 T

## 一、前言

第二篇中我们介绍了怎么在 51testing 论坛网站 (<http://bbs.51testing.com>) 上完成一个自动发帖的测试脚本。那么在这第三篇，也就是整个《Puppeteer 自动化测试简易入门教程》系列的最后一篇文章中，我们将结合 PageObject 设计模式和 Mocha 测试框架来让这个脚本更模块化和结构化，使其真正变成一个自动化测试工程项目。

## 二、PageObject 设计模式

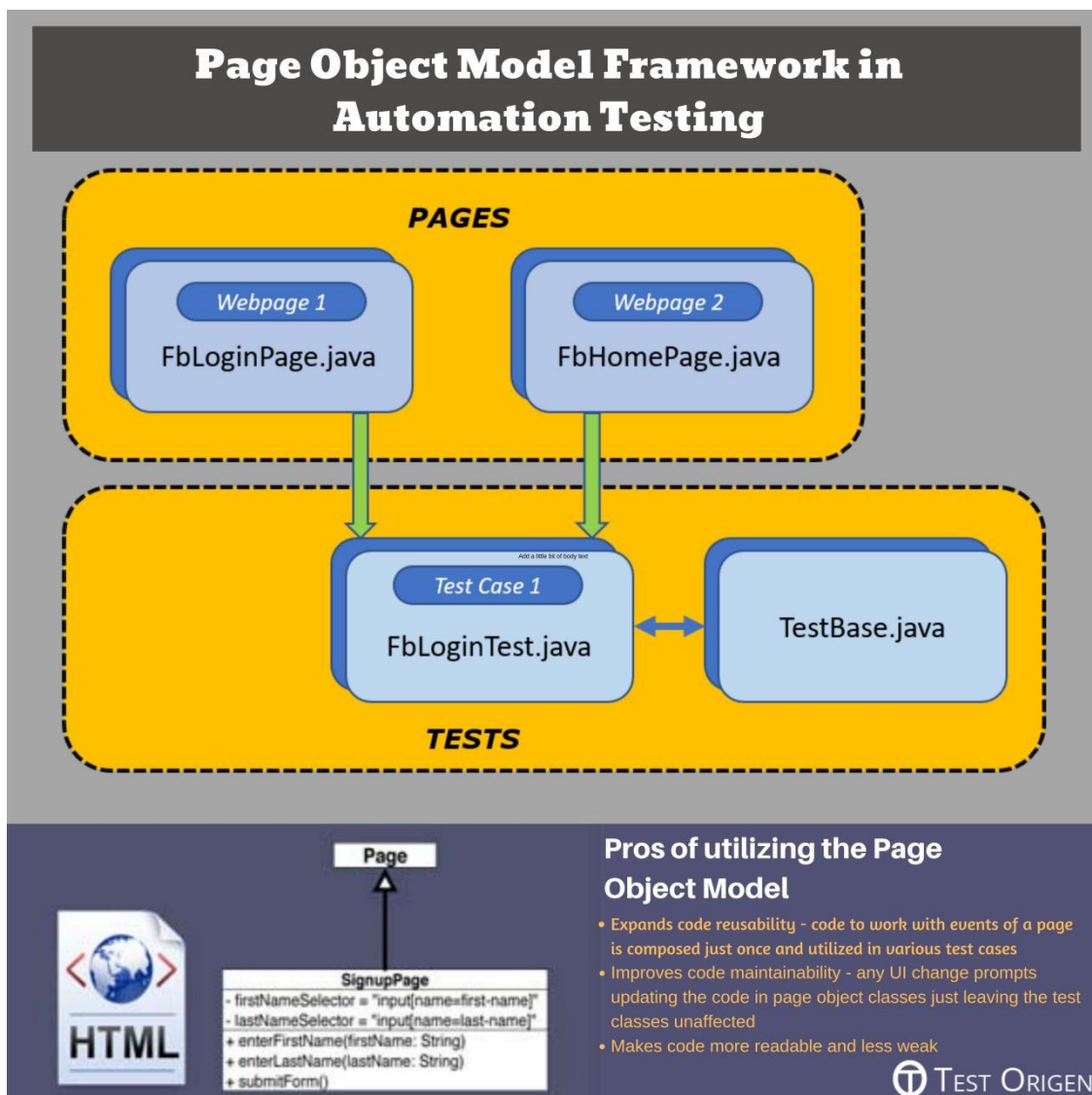


## Page Object Model



Page Object Model 是自动化测试领域最著名的一种工程设计模式，最早的概念由 Selenium 项目的创始人之一 Simon Stewart 于 2006 年提出，之后便广泛应用于几乎所有自动化测试项目中，可见在软件测试领域工程师们对这种设计模式非常认可。大家在网上可以找到无数关于 PO 模式的介绍，我在这里就不浪费篇幅赘述。

我们的项目在使用了 PO 设计模式之后，用例脚本中将不会出现具体的定位元素和测试数据，它们会被分类存放到相应的页面文件中，我们写不同的用例脚本时要用到什么元素或数据直接找到相关页面文件并从中调用它们就行了，这样我们的定位元素和测试数据就可以复用且相当好维护。



举一个我们在第二篇写过的一个场景例子：登录功能。我们登录的步骤一般就是输入用户名+密码，然后点【登录】就完成了，那这里的【用户名】输入框、【密码】输入



框和【登录】按钮三个定位元素我们就可以写到一个叫 LoginPage 的文件中，用户数据的话一般也可以写在这个文件中，有人会再新建一个 TestData 目录另作存放，这样实现了定位元素和测试数据也相互隔离也行。

```
//loginPage.js locators: {
  usernameInputCss: '#ls_username', passwordInputCss: '#ls_password',
  loginBtnCss: 'button[type="submit"].pn.vm', nameTextCss: '.vwmy',
  logoutBtnCss: '#um p:nth-child(2) > a:last-child',
  logoutMsgCss: '#messagetext',
},
testData: {
  url: 'http://bbs.51testing.com',
  username: 'tino',
  password: 'xxx',
}
//testcase.js
await page.type(homePage.locators.usernameInputCss,
homePage.testData.username);//输入用户名
await page.type(homePage.locators.passwordInputCss,
homePage.testData.password);//输入密码
await page.click(homePage.locators.loginBtnCss);//点击【登录】按钮
await page.waitForSelector(homePage.locators.nameTextCss);//验证是否登录成功
```

### 三、Mocha 测试框架



simple, flexible, fun



Mocha 是一款 JavaScripts 测试框架，它就等于 Python 界的 `unittest` 和 Java 界的 `JUnit`，只不过后两者是已经集成在各自语言中了，而 Mocha 是需要安装的。其实 JS 有很多测试框架，除了 Mocha 还有 Jest、Jasmine、CasperJS 等，这里介绍 Mocha 是因为它简洁的语法和优雅的风格，更便于我们学习。

## 1. 安装

在项目目录下执行 `npm install mocha` 安装或 `npm install --global mocha` 全局安装，这里我建议是全局安装，这样运行测试用例脚本时不用指定路径直接写 `mocha xxx.js`。

## 2. 示例脚本

官网（<https://mochajs.org/#getting-started>）上示例脚本如下：

```
var assert = require('assert'); describe('Array', function() {
describe('#indexOf()', function() {
it('should return -1 when the value is not present', function() { assert.equal([1, 2, 3].indexOf(4), -1);
});
});
});
```

代码第一行引入的还是 Node.js 自带的断言库，但 Mocha 是支持任何一个第三方断言库的，下载你喜欢的断言库并导入就可以在脚本中使用：

### ASSERTIONS

Mocha allows you to use any assertion library you wish. In the above example, we're using Node.js' built-in `assert` module — but generally, if it throws an `Error`, it will work! This means you can use libraries such as:

[should.js](#) - BDD style shown throughout these docs

[chai](#) - `expect()`, `assert()` and `should`-style assertions

[expect.js](#) - `expect()` style assertions

[better-assert](#) - C-style self-documenting `assert()`

[unexpected](#) - "the extensible BDD assertion toolkit"

第二行开始是示例脚本正文，`describe()` 是指测试套件，单引号内的是套件名，`it()` 是指测试用例，单引号内的是用例名，`describe()` 内还可以嵌套 `describe()`，`it()` 则是



可以有并列多个意味着测试套件内有多个测试用例，如果 `describe()` 内只有一个 `it()`，那套件名和用例名基本上可以写成一样。最后一行代码是断言语句。

### 3. 钩子函数

Mocha 的钩子函数有 4 个：`before()`、`after()`、`beforeEach()` 和 `afterEach()`，看名字就知道它们的作用和 `Unit` 里的 `setUp` 和 `tearDown` 是一样的，用来给测试用例执行做前置操作和清理操作，来看看官方给出的描述：

```
describe('hooks', function () {
  before(function () {
    // runs once before the first test in this block
  });

  after(function () {
    // runs once after the last test in this block
  });

  beforeEach(function () {
    // runs before each test in this block
  });

  afterEach(function () {
    // runs after each test in this block
  });

  // test cases
});
```

## 四、修改脚本

了解了 PO 设计模式和 Mocha 框架之后，接下来我们就要改造上一篇文章中使用的 51testing 论坛网站 (<http://bbs.51testing.com>) 测试脚本，只不过我把发帖用例去掉，换成退出登录的用例，这样能使代码简单些。

### 1. 自定义浏览器

这里我新建了个名为 `browser.js` 的文件，里面自定义了 2 个不同尺寸大小的浏览器，一个是默认尺寸，一个是最大化尺寸，我可以在用例脚本中任意调用一个来使用。这里要注意，要把方法名通过 `module.exports` 来导出，否则将无法在别的文件中调用：



```
//browser.js
const puppeteer = require('puppeteer');
module.exports = { browser_default, browser_max
} //导出模块
async function browser_default(){
const browser = await puppeteer.launch({ headless: false,
slowMo: 100, executablePath:
`C:\\Program Files\\Google\\Chrome\\Application\\chrome.exe`
});
return browser;
}
async function browser_max(){
const browser = await puppeteer.launch({ headless: false,
slowMo: 100, executablePath:
`C:\\Program Files\\Google\\Chrome\\Application\\chrome.exe`,
args: [
'--window-size=1920,1080',
],
defaultViewport: null
});
return browser;
}
```

## 2. 完善用例脚本

将第二节中的 PO 文件（存放定位元素和测试数据）和测试用例脚本加以完善，注意 PO 文件中 的所有内容也需要导出供调用：

```
//homePage.js module.exports = {
locators: {
usernameInputCss: '#ls_username', passwordInputCss: '#ls_password', loginBtnCss:
'button[type="submit"].pn.vm', nameTextCss: '.vwmy',
logoutBtnCss: '#um p:nth-child(2) > a:last-child', logoutMsgCss: '#messagetext',
},
}
```



```
testData: {
  url: 'http://bbs.51testing.com', username: 'tino',
  password: 'xxx',
},
assertMsg: {
  logoutMsgTxt: '您已退出站点，现在将以游客身份转入退出前页面，请稍候……'
}
}

//51testing.js

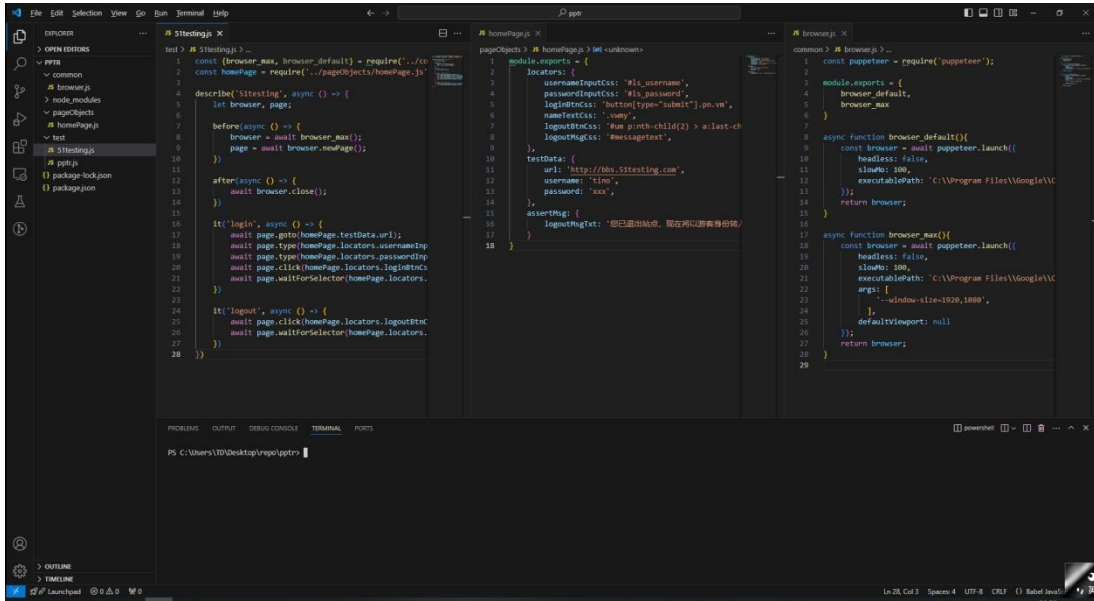
const browser_max = require('../common/browser.js').browser_max;//导入自定义浏览器 const
homePage = require('../pageObjects/homePage.js');//导入定位元素

describe('51testing', async () => { let browser, page;
  before(async () => {
    browser = await browser_max();//使用最大化浏览器 page = await browser.newPage();
  })
  after(async () => {
    await browser.close();
  })
  it('login', async () => {
    await page.goto(homePage.testData.url);
    await page.type(homePage.locators.usernameInputCss, homePage.testData.username);
    await page.type(homePage.locators.passwordInputCss, homePage.testData.password);
    await page.click(homePage.locators.loginBtnCss);
    await page.waitForSelector(homePage.locators.nameTextCss);
  })
  it('logout', async () => {
    await page.click(homePage.locators.logoutBtnCss);
    await page.waitForSelector(homePage.locators.usernameInputCss);
  })
})
```





### 3. 所有项目文件总览



### 五、执行测试脚本

执行之前有两点需要注意：由于 Mocha 有一个默认的运行用例时间 2S，超过这个时间如果用例 还没执行完将会被判定为失败，所以我们在执行命令中加一个参数 `--timeout 30000`，这里的时间是 HS（毫秒）。另外一点是 Mocha 会执行 `test` 目录下的测试用例脚本，所有我们约定俗成 是在工程中新建一个 `test` 的目录，然后把所有测试用例脚本都放进去，这样就可以在工作目 录下直接使用 `mocha` 命令运行脚本。

```
PS C:\Users\TD\Desktop\repo\pptr> mocha --timeout 30000
```

以下是脚本执行情况，执行之前我把使用的浏览器改成了默认大小的浏览器，这样能方便大家 看到代码编辑器 Terminal 中的一些输出信息：

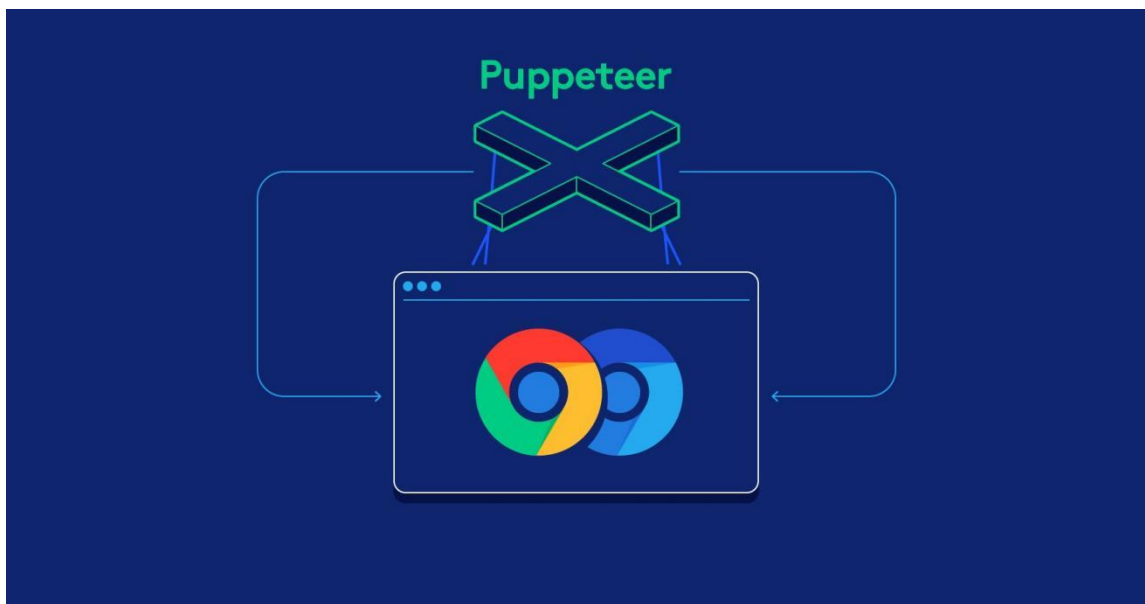


我们可以看到两个测试用例 `login` 和 `logout` 已经执行完毕，全部通过，并且在执行前后也顺利执行了启动和关闭浏览器操作。

## 六、系列总结

到此，连续三篇的《Puppeteer 自动化测试简易入门教程》就结束了，我们介绍了 Puppeteer 的安装、使用和简单应用，还有一些自动化测试项目中所涉及到的一些知识。其实 Puppeteer 在国内算是一款冷门的自动化测试工具，但由于它是 Google 出品，所以它是现在市面上操控 Chrome 浏览器能力最强的自动化测试工具，功能十分强大，它还有很多其它高级的玩法，想了解的话就去官网上（<https://pptr.dev/>）看看吧。另外它还是爬虫工程师的选择之一，它也有 Python 版本——Pyppeteer，会写 Python 的小伙伴可以尝试以下。

最后附上一张 Puppeteer 高清壁纸~祝大家玩得开心！



# 不会测试算法，借助 AI 来帮忙？

◆ 作者：M&T.

本文从测试人员测试过程中遇到的问题入手，如测试一个算法，不知道该如何进行入手测试，借助 AI 平台巧妙提问，提供测试思路，进而根据提供的方案进行测试。同时提供案例实操作如何借助 AI 全流程进行测试验证。包括内容如下：

- 理解需求
- AI 巧妙提问
- 找到关键数据
- 执行测试

## 1、理解需求

这是测试过程的起始点，测试人员需要与项目团队沟通，确保对算法的功能需求有深入的理解。理解需求包括了解算法的预期输入、输出，以及算法需要满足的业务目标和用户期望。

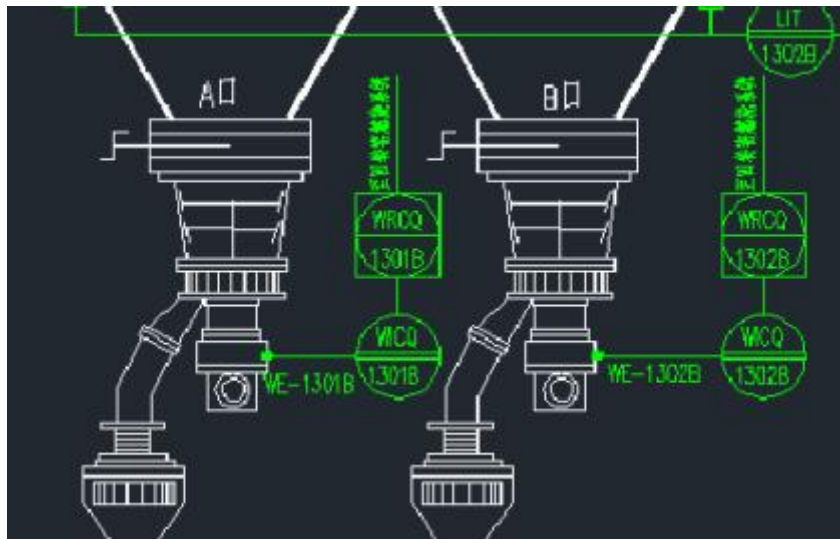
厂矿	设备名称	是否参与总量统计	设备类型	所属工序	原煤(t)	煤粉(t)
球团二线	粉煤仓料位秤	是		二线其他工序	—	28.00(进仓)
球团二线	给入回转窑煤粉计量A口申克称	是	煤粉计量设备	二线其他工序	—	9.44(用量)
球团二线	给入回转窑煤粉计量B口申克称	是	煤粉计量设备	二线其他工序	—	9.44(用量)
球团二线	原煤仓料位秤	是	原煤计量设备	二线其他工序	37.12(进仓)	—

上图中煤粉值是我需要验证的字段，验证公式为：

$$x \text{ 煤粉 (2)} = x \text{ 煤粉 (1)} (A \text{ 终止数} + B \text{ 终止数}) - (A \text{ 起始读数} + B \text{ 起始读数})$$

其中 A\B 分别对应下图两个申克秤。





## 2、AI 巧妙提问

这个阶段测试人员利用 AI 平台来提出问题，这些问题可能涉及算法的逻辑、边界条件、异常处理等方面。AI 平台可以帮助测试人员识别测试过程中可能忽视的点，提供测试思路和方法。

以下是我的提问：

x 煤粉 (2) = x 煤粉 (1) (A 终止数+B 终止数) - (A 起始读数+B 起始读数) 这是算法公式， x 煤粉 (最终) 等于 28.00(进仓)，要验证 x 煤粉 (2) 的数据值是否正确，我需要怎么做？

## AI 回答

### 算法公式：

$$[x_{\text{煤粉}(2)} = x_{\text{煤粉}(1)} \times (A_{\text{终止数}} + B_{\text{终止数}}) - (A_{\text{起始读数}} + B_{\text{起始读数}})]$$

### 已知条件：



-  $x_{\text{煤粉(1)}}$  是已知的初始粉煤量。

-  $x_{\text{煤粉(2)}}$  是我们想要计算的最终粉煤量。

### 验证步骤：

1. **确定设备读数**：您需要知道设备的终止读数和起始读数。

- 设备 A 的终止读数  $A_{\text{终止数}}$
- 设备 B 的终止读数  $B_{\text{终止数}}$
- 设备 A 的起始读数  $A_{\text{起始读数}}$
- 设备 B 的起始读数  $B_{\text{起始读数}}$

2. **代入公式**：将已知的  $x_{\text{煤粉(1)}}$  和设备的读数代入公式中。

3. **计算  $x_{\text{煤粉(2)}}$** ：通过公式计算出  $x_{\text{煤粉(2)}}$  的值。

4. **验证**：如果计算出的  $x_{\text{煤粉(2)}}$  等于 28.00，那么数据值是正  
确的。

### 示例：

假设  $x_{\text{煤粉(1)}}$  已知，设备读数如下：

- $A_{\text{终止数}} = 120$
- $B_{\text{终止数}} = 180$
- $A_{\text{起始读数}} = 60$
- $B_{\text{起始读数}} = 90$

代入公式计算  $x_{\text{煤粉(2)}}$ ：

$$x_{\text{煤粉(2)}} = x_{\text{煤粉(1)}} \times (120 + 180) - (60 + 90)$$

$$x_{\text{煤粉(2)}} = x_{\text{煤粉(1)}} \times 300 - 150$$

假设  $x_{\text{煤粉(1)}}$  为某个具体的数值，比如  $x_{\text{煤粉(1)}} = 1.0$ ：

$$x_{\text{煤粉(2)}} = 1.0 \times 300 - 150$$

$$x_{\text{煤粉(2)}} = 300 - 150$$



$\backslash(x_{\text{煤粉}(2)}) = 150 \backslash$

如果  $\backslash(x_{\text{煤粉}(2)})$  的计算结果为 150，而不是 28.00，这表明数据值可能不正确，或者设备读数有误。

### 结论：

- 确保  $\backslash(x_{\text{煤粉}(1)})$  和设备读数的准确性。
- 使用公式计算  $\backslash(x_{\text{煤粉}(2)})$ 。
- 比较计算结果与预期的 28.00，验证数据值是否正确。

如果您有具体的  $\backslash(x_{\text{煤粉}(1)})$  值和设备读数，请提供，我可以帮您进行更准确的计算。

### 3、找到关键数据

在这个阶段测试人员需要确定哪些数据对于算法测试至关重要，包括正常数据、边界数据、异常数据等。找到关键数据是确保测试覆盖到所有重要场景的关键步骤，有助于发现潜在的问题和缺陷。

上面问题的关键数据分别为：A 终止数?、B 终止数?、A 起始读数?、B 起始读数?、煤粉(1)

所以在数据库中或者业务中找到对应的值即可。

通过跟开发沟通了解到：

A/B 终止数-A/B 起始读数 在数据库存了数据 分别为：A 9.4376970+ B 9.4368320

煤粉(1)：等于实时表 16 号最后一条数据-第一条数数据：

code_1706852102350954496	给入回转窑煤粉计量A口申克称	pulverized_coal	fam2_process_code4	2024-07-16	9.437697
code_1706852103147872256	粉煤仓料位秤	bunker_coal	fam2_process_code4	2024-07-16	27.995935
code_1706852102925574144	给入回转窑煤粉计量B口申克称	pulverized_coal	fam2_process_code4	2024-07-16	9.436832



#### 4、执行测试

最后一步是使用收集到的数据和信息来执行测试或计算，以验证解决方案或分析结果是否符合预期。包括数学计算、模拟、实验或其他形式的验证过程。

上面的数据带入公式：

$$x \text{ 煤粉 (2)} = 15.5964790 - 6.1668680 + 9.4376970 + 9.4368320 = 28.30$$

28.3 与 28 不相等，可以判定这是一个 bug

于是在 bug 系统提交了关于这个算法的 bug

#### 回顾：

总结，以上就是利用 AI 指导算法测试的全过程，大致分为需求理解-提问-找到关键数据-测试这几个过程，其实 AI 更像是一位老师，大脑里容纳了各种知识储备，遇到不懂的测试问题随时可以提问，尤其没有思路时能够完善的提供解决方案。

#### 拓展学习

[1] 人工智能深度学习全栈开发工程师

咨询：微信 atstudy-js 备注：AI



# 轻松搭建 Graphql API 模拟服务器

◆ 作者：NA

**学习目的：**敏捷开发，数据横行的时代，测试的是要求越来越重要，复杂。在接受挑战和变化时，测试团队需要不断学习和适应新的技术和方法。本例我们将一步步搭建一个 jetty-server 实现 Graphql API 的请求与返回，然后通过 Postman 讲解 Graphql API 请求的 2 种请求方式，最后接合 rest-Assured 在 testNg 框架下模拟 API 的请求与数据返回的测试，最终实现 Graphql API 在模拟服务器上测试的自动化。

**温馨提示：**如果您要按着示例一起做，请务必配置如下工具以及学习相关的知识。用例主要是搭建 Graphql API 服务器，结合 rest-Assured 完成 API 的自动化测试，对于不了解 rest-Assured，Graphql API 的小伙伴，请做些额外功课吧（可以参考我之前写的 rest-Assured 的文章，有详细的配置与应用步骤）。

- IDE: IntelliJ IDEA
- 语言: Java
- API 测试开发: Rest-Assured
- API : Graphql API
- 测试框架: TestNg
- 项目类型: Maven
- API 测试工具: Postman





知识重点:

- Graphql API 服务器搭建: 创建 Graphql API Schema, 根据参数灵活返回请求
- Graphql API 请求: 理解 Graphql API 请求方式通过 Postman
- Rest Assured : 模拟 API 请求并验证

## 一 Maven 项目配置

配置构建 Graphql API 服务器, rest-Assured 插件在 POM.xml

如下所示: 需要将对应的插件正确的引入到 pom.xml 的<dependency>节点下。(本次用例的插件比较多哦) 提示: 如果 dependency 没有自动加载, 可以手动加载, 相应的 jar 包就会下载了。

### Graphql

```
<dependency>  
    <groupId>com.graphql-java</groupId>  
    <artifactId>graphql-java</artifactId>  
    <version>22.0</version>  
</dependency>
```

### Jetty server 以及相关插件

```
<dependency>  
    <groupId>org.eclipse.jetty</groupId>  
    <artifactId>jetty-server</artifactId>  
    <version>11.0.21</version>  
</dependency>  
<dependency>  
    <groupId>org.eclipse.jetty</groupId>  
    <artifactId>jetty-util</artifactId>  
    <version>11.0.21</version>  
</dependency>  
<dependency>
```



```
<groupId>org.eclipse.jetty</groupId>
<artifactId>jetty-servlet</artifactId>
<version>11.0.21</version>
</dependency>
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>4.0.1</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-core</artifactId>
  <version>2.17.1</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.17.0</version>
</dependency>
<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.11.0</version>
</dependency>
rest-Assured
<dependency>
  <groupId>io.rest-assured</groupId>
  <artifactId>rest-assured</artifactId>
  <version>5.4.0</version>
  <scope>compile</scope>
</dependency>
```



## 二. 测试用例分解

**Graphql API:** 首先了解一下要搭建的服务器的接收数据参数、类型以及返回的数据类型、格式，下图 postman 中显示了 graphql api 的请求与数据返回格式。想像一下有这个样服务器，当用户请求调阅某一个文档时，服务器就会返回对应的用户信息以及是否可以调阅访文档的信息。

The screenshot shows a Postman interface for a GraphQL API. The URL is `http://localhost:8080/graphql`. The query is:

```
query GetStaffLevel {
  getStaffLevel(
    name: "jun"
    professional: "staff"
    fileName: "project_cost_confidential.doc"
  ) {
    department
    name
    web
    level
  }
}
```

The response body is:

```
{
  "data": {
    "getStaffLevel": [
      {
        "department": "LAB A",
        "name": "jun",
        "web": "http://www.jun.com",
        "level": "low, you are not allowed to review confidential file project_cost_confidential.doc"
      }
    ]
  }
}
```

**API 数据分析:** 首先要搭建一个接受 API 的服务器，并且能够对有符合条件的 API 请求做出响应。

地址: `http://localhost:8080/graphql`

参数: `name, professional, fileName`

参数格式: 字符型

响应: `department, name, web, level`

响应格式与类型: 格式 `.json`, 数据字符型

当服务器接收到请求时，需要返回用户是否可以调阅文档以及其他基本信息。



返回数据结果可以调阅文档是否，需要按预置条件：

- professional 是 staff 可以查阅非机密文档
- professional 是 manager 可以查阅 任何类型的文档

如果定义用户调阅 的文档是 否机密呢？因为本例主要是学习 graphql 服务器的搭建，所以通过 fileName 验证是否为机密文档。如果 fileName 包含 “confidential” 就判定该文档为机密，否则就是非机密文档。

有了上面的理解，现在就是实现步骤：

1. 构建 API 服务器 （定义请求，返回数据，启动服务，关闭服务）
2. 验证 API 是否成功 （通过 postman）
3. 自动测试 - rest-Assured 请求并验证返回状态与调阅权限

提示：这一步非常的重要，使得后续的服务器的搭建、测试条理化，能系统性的、逻辑性的组织工作任务。

### 三. 构建 GraphQL API 服务器

此用例中构建服务器需要 2 个 java 文件：

1. MockGraphQLServer.java - 此文件主要功能是定义 API 的 schema，定义数据返回内容依据请求参数，以及服务器的开启与关闭。

schemaBuilder()方法定义并返回 API schema，以下代码 有 2 个 object，staffLevelType 对象 定义了响应数据应该有的字段和字段的类型。 queryType 对象定义了请求时的需要的参数以及参数类型。

```
public GraphQLSchema schemaBuilder() {
    // define staff level type (response filed , type)
    GraphQLObjectType staffLevelType = GraphQLObjectType.newObject()
        .name("StaffLevel")
        .field(GraphQLFieldDefinition.newFieldDefinition()
            .name("department").type(GraphQLString))
        .field(GraphQLFieldDefinition.newFieldDefinition()
```



```

        .name("name").type(GraphQLString))
    .field(GraphQLFieldDefinition.newFieldDefinition()
        .name("web").type(GraphQLString))
    .field(GraphQLFieldDefinition.newFieldDefinition()
        .name("level").type(GraphQLString))
    .build();
// define query parameter type
GraphQLObjectType queryType = GraphQLObjectType.newObject()
    .name("Query")
    .field(GraphQLFieldDefinition.newFieldDefinition()
        .name("getStaffLevel")
        .type(new GraphQLList(staffLevelType))
        .argument(argument -> argument.name("name").type(GraphQLString))
        .argument(argument -> argument.name("professional").type(GraphQLString))
        .argument(argument -> argument.name("fileName").type(GraphQLString))
        .dataFetcher(getStaffLevelDetails())) // get response data
    .build();
GraphQLSchema schema = GraphQLSchema.newSchema().query(queryType).build();
return schema;
}

```

2.getStaffLevelDetails()方法是对请求参数的具体数据返回。这里为了比较直观地说明返回数据与请求参数的验证关系，显示数据 定义的 details 就直接 写在了每个判断条件里，这样的结果是代码看起来不专业，但是容易简单明了地知道业务逻辑关系，调试也比较容易。在实际开发中，待业务确定以后，就可以优化这部分代码。

```

private DataFetcher<?> getStaffLevelDetails() {
    return environment -> {
        String name = environment.getArgument("name");
        String professional = environment.getArgument("professional");
        String fileName = environment.getArgument("fileName");
        List<Map<String, Object>> staffLevelList = new ArrayList<>();
        if (professional.contentEquals("manager")) {

```



```

Map<String, Object> details = new LinkedHashMap<>();
details.put("department", "LAB A");
details.put("name", name);
details.put("web", "http://www." + name + ".com");
details.put("level", "high, your are allowed to review " + fileName);
staffLevelList.add(details);
} else {
    if (fileName.contains("confidential")) {
        Map<String, Object> details = new LinkedHashMap<>();
        details.put("department", "LAB A");
        details.put("name", name);
        details.put("web", "http://www." + name + ".com");
        details.put("level", "low, you are not allowed to review confidential file " +
fileName);
        staffLevelList.add(details);
    } else {
        Map<String, Object> details = new LinkedHashMap<>();
        details.put("department", "LAB A");
        details.put("name", name);
        details.put("web", "http://www." + name + ".com");
        details.put("level", "low, your are allowed to review none confidential file " +
fileName);
        staffLevelList.add(details);
    }
}
return staffLevelList;
};
}

```

### 服务开启与关闭

下面这 2 个 方法就很容易理解了，把 **port** 作为一个参数是为了服务器启动的更灵活在测试中。



这里 `new GraphQLHandler(graphQL)` 是另一个 `GraphQLHandler.java` 里的构造方法。

```
public void serverStart(GraphQLSchema schema, int port) {
    GraphQL graphQL = GraphQL.newGraphQL(schema).build();
    server = new Server(port);
    server.setHandler(new GraphQLHandler(graphQL));
    try {
        server.start();
        System.out.println("Local server with port "+ port +" has been started");
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

public void serverStop() {
    if (server != null) {
        try {
            server.stop();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

2. `GraphQLHandler.java` - 此文件功能是处理 API 的请求，并对接受到请求进行优化再执行，当请求符合定义的 `schema` 格式时返回请求成功数据，另外此处也添加了用请求方法的验证。

1) `GraphQLHandler Class extends AbstractHandler`

2) 此处的 `GraphQLHandler()` 方法被用在 服务器启动时。

3) `handle()` 方法： 首先验证请求 的路径 (`/graphql`) 与方法 (`post`)

4) 要解释方法中具体的功能，先了解一下 `graphql api` 请求方式。在接下的 `postman` 使用中我们详细讲解。而这里需要了解的是 `graphql api` 的请求分 2 部分，一部分是 查



询，一部分是 参数的变量值。在执行时，代码不能按一个整体发送，必须拆分成 2 部分然后在重定义成 `executionInput` 才可以请求成功。在每一个步骤里我都加了注释，期望。

```
public class GraphQLHandler extends AbstractHandler {
    private final GraphQL graphQL;
    public GraphQLHandler(GraphQL graphQL) {
        this.graphQL = graphQL;
    }
    @Override
    public void handle(String s, Request request, HttpServletRequest httpRequest,
        HttpServletResponse httpResponse) throws IOException, ServletException {
        try {
            if ("/graphql".equalsIgnoreCase(s) && "POST".equalsIgnoreCase(request.getMethod()))
            {
                // get request body from request
                String requestBody =
request.getReader().lines().collect(Collectors.joining(System.lineSeparator()));
                // covert request boy as jsonObject
                JsonObject jsonObject = JsonParser.parseString(requestBody).getAsJsonObject();
                // extract query part from jsonObject
                String query = jsonObject.get("query").getAsString();
                // extract variables part from jsonObject
                JsonObject variablesObj = jsonObject.getAsJsonObject("variables");
                // reform variables as map type
                LinkedHashMap<String, Object> variables = new LinkedHashMap<>();
                variablesObj.entrySet().forEach(entry ->{
                    variables.put(entry.getKey(),entry.getValue().getAsString());
                });
                // covert query with variables as execution input
                ExecutionInput executionInput =
ExecutionInput.newExecutionInput().query(query).variables(variables).build();
                // perform execution
                ExecutionResult executionResult = graphQL.execute(executionInput);
                httpResponse.setContentType("application/json");
            }
        }
    }
}
```





```

        httpServletResponse.setStatus(HttpServletResponse.SC_OK);
        // reform returned data as json
        ObjectMapper objectMapper = new ObjectMapper();

objectMapper.writeValue(httpServletResponse.getWriter(),executionResult.toSpecification());
        request.setHandled(true);
    }
} catch (Exception e)
{
    e.printStackTrace();

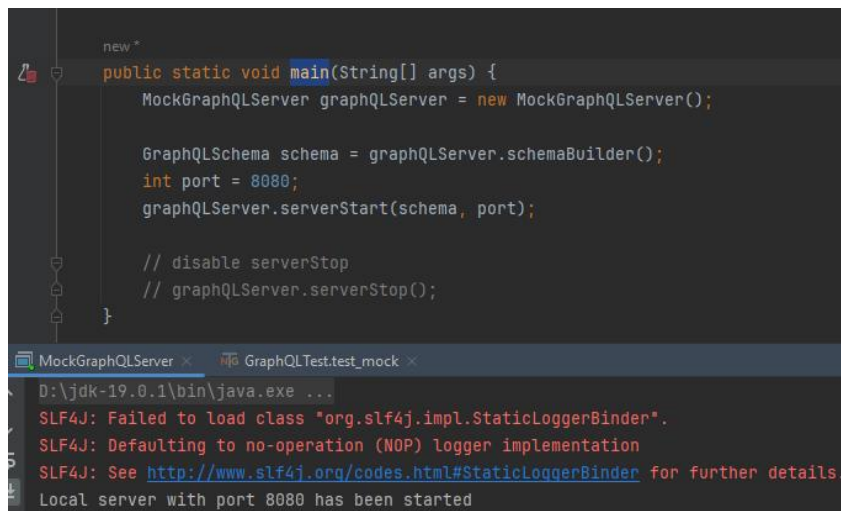
httpServletResponse.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
    httpServletResponse.getWriter().println("Internal Server Error");
    request.setHandled(true);
}
}
}
}

```

#### 四.Postman 请求验证

1.启动服务器， 在 MockGraphQLServer.java 文件我们创建 一个 main 方法， 在此方法调用 schemaBuilder() 和 serverStart() ， 点击运行这个方法。

在控制台会看到 启动成功的信息： local server with port 8080 has been started



```

new *
public static void main(String[] args) {
    MockGraphQLServer graphQLServer = new MockGraphQLServer();

    GraphQLSchema schema = graphQLServer.schemaBuilder();
    int port = 8080;
    graphQLServer.serverStart(schema, port);

    // disable serverStop
    // graphQLServer.serverStop();
}
}
}
}

```

```

D:\jdk-19.0.1\bin\java.exe ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Local server with port 8080 has been started

```



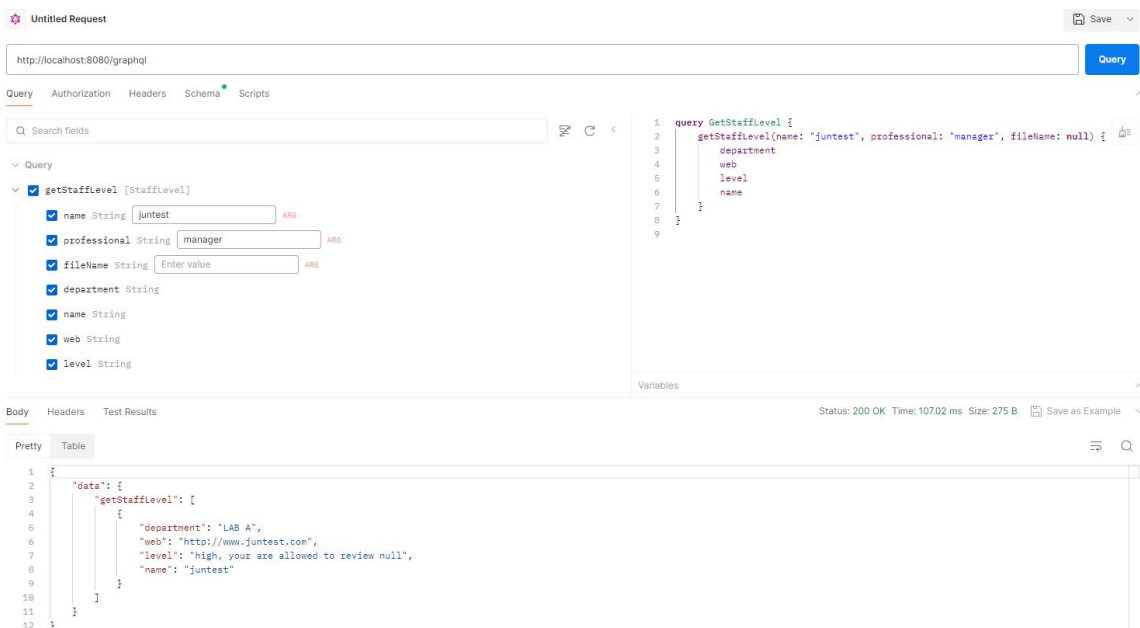
## 2.Postman 请求方法一: GraphQL

打开 postman, 在主功能菜单下选择 文件 > 新建, 在弹出的窗口下选择 并创建 GraphQL。



输入 地址: `http://localhost:8080/graphql` 等待加载成功。在 query 部分可以看到 `getStaffLevel` (`StaffLevel` 代码中定义部分) 的各个字段以及对应的字段类型。其中可以 `name,professional,filename` 是 3 个变量。选择每个字段并 输入相应的参数, 点击 Query 按钮。

按示例中, `professional: manager`, 可以调阅所有的文件。返回的数据格式 json。



## 3.Postman 请求方法二: Http

打开 postman, 在主功能菜单下选择 文件 > 新建, 在弹出的窗口下选择 并创建 HTTP。

输入地址: `http://localhost:8080/graphql`, 请求方式: POST. 在 Body > GraphQL > 输入 Query 与 GraphQL Variables 2 部分, 点击 Send 按钮, 验证返回结果在 Body 部分。



从下面 query 可以到 这里分 2 个部份,第 1 部分是参数(name, professional, fileName), 第 2 部分是需要返回的数据 (department,name,web,level)。(注: 这 2 个部分的字段可以调整顺序,同时也可以选择性取舍,但是不能去掉必填参数,或者返回字段是需求验证的。比如这个用例 中 fileName, professional 不可去掉, level 不可能去掉)

### Query:

```
query StaffLevel($name: String, $professional: String, $fileName: String) {
  getStaffLevel(name: $name, professional: $professional, fileName: $fileName) {
    department    name    web level }}
```

### GraphQL Variables:

```
{
  "name": "jun",
  "professional": "staff",
  "fileName": "projectA_design_confidential.doc"
}
```

现在理解了 Graphql Api 的 http 查询方式,那么在前面 构建服务器 GraphQLHandler.java 里的 handle()方法,所谓的不能用一个整体发送这个查询,就是指查询与变量不能是一个字符串的整体,而是要 Query 与 GraphQL Variables 分解再组合成 executionInput 当接收到 rest-Assured 的请求时。

The screenshot shows a REST client interface with a POST request to http://localhost:8080/graphql. The request body is a GraphQL query and variables. The response is a JSON object containing the query results.

**QUERY**

```
1 query StaffLevel($name: String, $professional: String, $fileName: String)
2 {
3   getStaffLevel(name: $name, professional: $professional, fileName: $fileName)
4   {
5     department
6     name
7     web
8     level
9   }
10 }
11 }
12 }
```

**GRAPHQL VARIABLES**

```
1 {
2   "name": "jun",
3   "professional": "staff",
4   "fileName": "projectA_design_confidential.doc"
5 }
```

**Body**

```
1 {
2   "data": {
3     "getStaffLevel": [
4       {
5         "department": "LAB A",
6         "name": "jun",
7         "web": "http://www.jun.com",
8         "level": "low, you are not allowed to review confidential file projectA_design_confidential.doc"
9       }
10     ]
11   }
12 }
```



## 五.TestNg 用例

依照上面分析，让我们一步一步看下代码的实现吧。

提示：本示例用的 Java + testNg + rest-assured

### 代码详解：GraphQLTest.java

1.基于前面 http 查询的请求详解，requestBody()方法是对 Query 与 GraphQL Variables 组合并返回。

```
public String requestBody(String name, String professional,String fileName){
    String query = "query StaffLevel($name: String, $professional: String, $fileName: String)" +
        "    {   getStaffLevel(name: $name, professional: $professional,fileName: $fileName)"
+
        "    {   department" +
        "        name" +
        "        web" +
        "        level" +
        "    }" +
        "};";
    return "{\"query\":\""+query+"\","variables\":{\"n" +
        "\"name\": \""+name+"\""," +
        "\"professional\": \""+professional+"\""," +
        "\"fileName\": \""+fileName+"\" } }";
}
```

2. test\_mock()方法分 4 部分 1)服务器启动 baseURI 设置; 2)GraphQL Variables 定义赋值与 request 请求 ; 3) response 数据验证 ; 4) 服务器关闭

@Test

```
public void test_mock(){
    // setup server and baseURI
    MockGraphQLServer graphQLServer = new MockGraphQLServer();
    GraphQLSchema schema = graphQLServer.schemaBuilder();
```



```

int port = 808;

graphqlServer.serverStart(schema, port);

baseURI = "http://localhost:"+port+"/graphql";

// query

String name="jun";

String professional="staff";

String fileName="projectA_design_confidential.doc";

String requestBody = requestBody(name,professional,fileName);

Response response = given().contentType("application/json").body(requestBody).log().all().post();

// verify

response.then().statusCode(200);

String verifyLevelMessage = "low, you are not allowed to review confidential file "+ fileName;

response.then().body("data.getStaffLevel[0].level",containsString(verifyLevelMessage));

response.getBody().prettyPrint(); // print out response

//stop server

graphqlServer.serverStop();

}

```

## 六. 用例运行

好了，先在运行下测试结果吧。这里我就直接在 IDE 里运行了。为更好地捕获 请求与返回时的具体信息：在 request 与 response 中都应用了.log().all(), response.then().log().all().statusCode(200);

```

Test passed! /.../...
Local server with port 808 has been started
[AppClassLoader@4d73180a] warning javax.* types are not being woven because the weaver option '-Xset:weaveJavaxPackages=true' has not been specified
Request method: POST
Request URI: http://localhost:808/graphql
Proxy: <none>
Request params: <none>
Query params: <none>
Form params: <none>
Path params: <none>
Headers: Accept/*/*
Content-Type=application/json
Cookies: <none>
Multiparts: <none>
Body:
{
  "query": "query StaffLevel($name: String, $professional: String, $fileName: String) { getStaffLevel(name: $name, professional: $professional, fileName: $fileName) { department name web level }}",
  "variables": {
    "name": "jun",
    "professional": "staff",
    "fileName": "projectA_design_confidential.doc"
  }
}
HTTP/1.1 200 OK
Date: Fri, 05 Jul 2014 11:12:07 GMT
Content-Type: application/json
Content-Length: 187
Server: Jetty(11.0.21)
{
  "data": {
    "getStaffLevel": [
      {
        "department": "LAB A",
        "name": "jun",
        "web": "http://www.jun.com",
        "level": "low, you are not allowed to review confidential file projectA_design_confidential.doc"
      }
    ]
  }
}

```



## 七. 错误排查与代码优化

### 问题 一: InvalidSyntax

在搭建服务器时我遇到的最头疼、最费时解决的一个错误信息就是 `InvalidSyntax`。  
(如下图所示)

这个运行中, `executionInput` 是由 `requestBody` 直接创建的。

```
// directly use requestBody
```

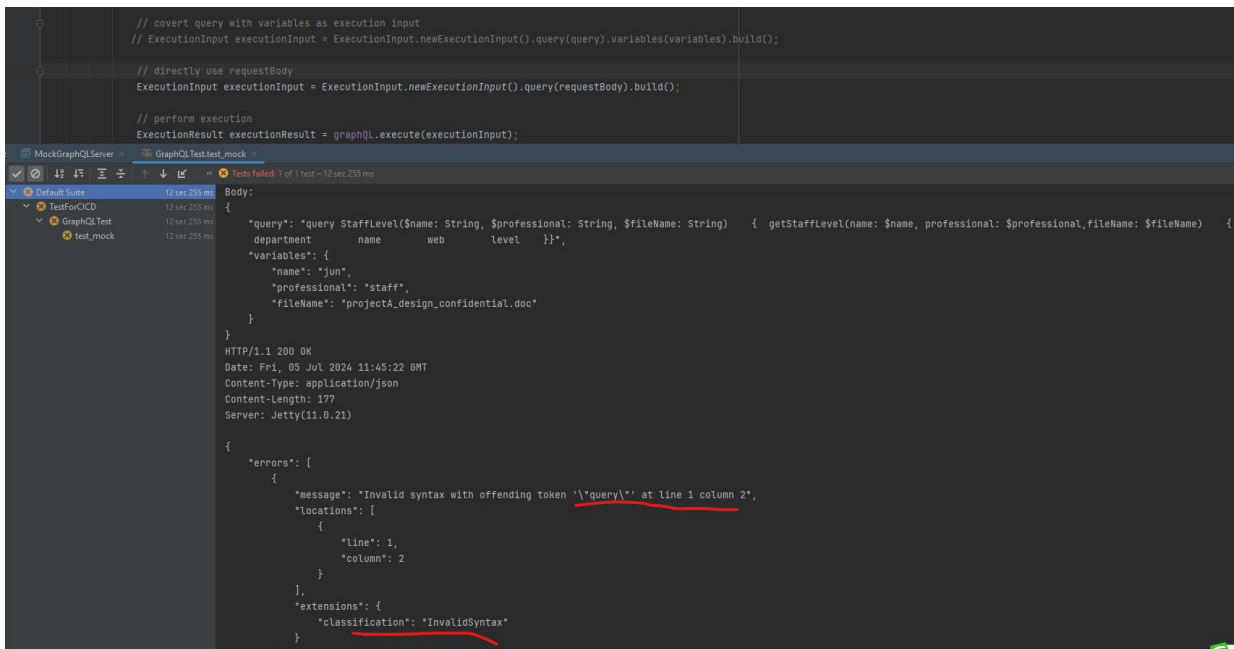
```
ExecutionInput executionInput = ExecutionInput.newExecutionInput().query(requestBody).build();
```

在排除这个问题时, 起初以为是请求 `Body` 的格式写错了, 就转到请求真实的服务器进行验证。然后以为是 `schema` 定义的错误, 反复检查重写这部分代码。最后认为是 `handle()` 功能出的错误, 但是定义不到具体的出错代码, 不知道原因, 就束手无策。一通查询资料, 问 ChatGPT, 毫无结果, 这个问题整整花费了 5 个多小时的不停地调试, 最后才确定不能直接用 `requestBody` 而需要把 `Query` 与 `GraphQL Variables` 分开。这里分享出来希望大家不要走弯路。

```
// covert query with variables as execution input
// ExecutionInput executionInput = ExecutionInput.newExecutionInput().query(query).variables(variables).build();

// directly use requestBody
ExecutionInput executionInput = ExecutionInput.newExecutionInput().query(requestBody).build();

// perform execution
ExecutionResult executionResult = graphql.execute(executionInput);
```



```
{
  "errors": [
    {
      "message": "Invalid syntax with offending token '\query\' at line 1 column 2",
      "locations": [
        {
          "line": 1,
          "column": 2
        }
      ],
      "extensions": {
        "classification": "InvalidSyntax"
      }
    }
  ]
}
```

代码优化: 除了上面提到定义返回数据的代码部分, 小伙伴们也可以尝试进一步优化 `handle()`, 比如当接收到请求地址, 方法错误时返回 `bad request` 等信息。



## 八. 总结

构建服务器主要是提前实现测试用例代码开发，一旦产品功能开发完成，测试就可以立即执行验证开发的功能是否符合要求。不过这里有一个很严格的需求在文档与沟通方面，接口文档的定义包含数据字段、类型、格式以及业务要求都需要预先梳理清晰，开发与测试人员要对使用接口以及业务逻辑有明确且一致的理解。如果服务器 API 的开发方不在一个团队，测试人员还得需要与 API 开发方进行沟通交流，没有相关的知识，是无从构建一个模拟服务器的。在实际环境中，这也是很多公司忽略或者说在思想认知上标识为“没必要”。

测试工作范围可大可小，可深可浅，相信做测试的小伙伴们都有自己的理解，期望共勉。谢谢大家。



# 软件测试下的 AI 之路 (6)

◆ 作者: Austin

随着科技日新月异的发展,人工智能正逐渐渗透到我们生活的各方各面,从智能语音助手到自动驾驶汽车,从智能家居到人脸识别技术,AI 正以其卓越的智能和学习能力引领着新时代的发展方向。在这个快速演进的时代中,软件测试领域也受到了不小的冲击。虽然在当下,传统的软测技术仍然是绝对的主力,但是身为 IT 行业中的一员,近几年 AI 的全新业务体验与其超强的算力所带来的震撼感受也应该远超其他行业。所以为了跟上时代的步伐,作为软测的大家是不是也应该考虑如何让 AI 辅助我们更加完整高效的完成日常各类质量保障工作呢?

上一期我们介绍了测试中一个非常重要的元素:断言,以及在 mabl 中如何利用各种类型的断言方式来满足不同测试场景需求的方式。当然 mabl 平台不仅仅只支持 web ui 自动化测试而已,它还能进行各种类型的测试活动,而今天我们就来继续介绍如何在 mabl 平台中进行另一种非常常见的测试活动类型——接口测试。

接口测试(API 测试)相信也是大家在日常测试工作中接触频率仅次于黑盒测试的一种测试活动了,其本质就是验证软件系统之间或系统内部模块之间的交互是否符合预期,当然这是在消息层的动作,所以简单来说相比起界面自动化测试来说,它的执行效率更高。自然测试设计的关注点更加的倾向于接口的功能、性能、安全性等方面,确保数据在传输和交换过程中的正确性、完整性和安全性。一般是通过软件工具来模拟实际的数据请求和响应,接口测试能够全面覆盖各种场景和异常情况,从而确保软件系统的稳定性和可靠性。

市面上用来做自动化接口测试的工具数不胜数,大家在日常工作中也有自己较为趁手的相应软件,这里我们直接先说结果,这里不分高下,只因众口难调,而相较于 mabl 来说,我们只说说它在自动化接口测试中的一些优势。

相较于其他的软件来说, mabl 相对的智能化和自动化程度更高,因为它本身就是一





种革命化的自动化测试的解决方案，就比如我们之前在 UI 自动化测试中介绍的用例的自我修复，元素的平移替换等一些基于 AI 的新颖功能。

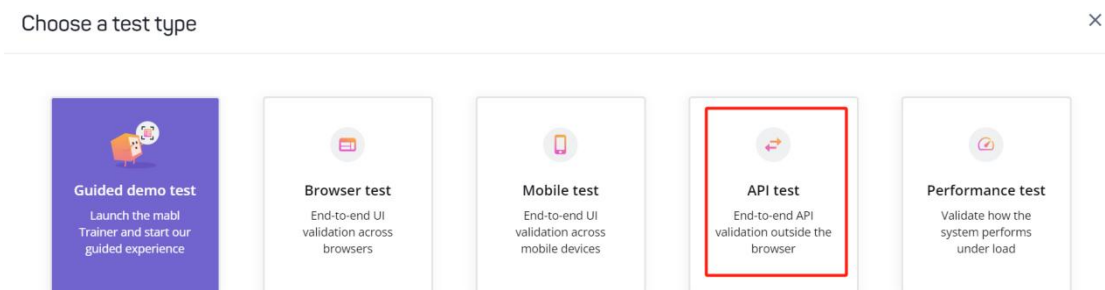
mabl 提供了专用的提供了数据驱动测试引擎，允许我们将测试数据集参数化并将其注入到测试自动化中，相比于 postman 等主流软件的一些参数化测试，其灵活性与业务适用性会高出不少程度。

另外 mabl 基于云服务的架构特性，这使得谗言团队的小伙伴们可以轻松地访问和使用其服务，而无需担心硬件和软件环境的问题。此外，mabl 还提供了可扩展性强的测试平台，支持大规模的并发测试和分布式测试，对于企业级测试的需求也可以轻易的满足。对于一些虽然也拥有网页版的软件来说，其整体的云服务特性与高扩展性的表现，只能说有过之而不及。

mabl 内的版本控制也是相当的出色，它提供了更为完善和灵活的团队协作工具，使得测试人员在团队中可以更方便地共享和管理测试用例、测试结果和脚本。而以上说涉及到的内容，我们也会在后面利用不同篇幅的文章来进行详细的介绍。

接下来我们就来看看如何在 mabl 平台中进行最基本的接口测试设计。

在 mabl 中先点击左边侧边栏内的 "New test"，这里我们选择 "API test"，来创建一个端对端的 API 测试类型。



在弹出的接口测试信息配置窗口中，填入对应的信息，以下信息具体为：

**test name:** 这里填入对应的测试活动名，下面的测试描述内容栏同理，用来区别不同测试活动内容。

**test labels:** 测试标签根据之前创建过的标签来进行分类，方便测试活动多了之后快速搜索和定位。



核心设置里的协议和基础路径（Protocol，URL）懂得都懂，输入被测对象页面的访问路径即可。

### Basic information

Branch  
master

Test name \*

My New Test

Test description

Test labels

### Core settings

Enter a starting API URL to use to train your test.

Protocol \*

https://

Base API URL \*

Cancel

Create test

测试活动的基础信息填写完成后，就会跳转至接口测试编辑器画面，这里我们先添加一个简单的接口测试步骤。右侧会有两个选项，左边的自然是添加一步测试步骤，右边则是可以通过 **postman** 将其软件内的实现已经设计完的 **collection** 直接导入进来。我们先选择左边的"Add step"。

The screenshot shows a web browser window titled "mabl: Create API test - RTA\_Fusion\_DSY\_TEST\_DEV". The interface includes a toolbar with icons for play, refresh, edit, and save. A "Branch" dropdown is set to "master". The main heading is "Create an API test" with the instruction "Get started by adding a step or importing a Postman collection". There are two main buttons: "+ Add step" and "Import Postman API collection". On the left, there is a "Variables" table:

Name	Default	Current
api.url		https://wei

Below the table is an "Add row" button. At the bottom, there is a note: "You can import collections without folders above. In order to import a collection with folders, please follow our documentation here".



然后来到接口编辑界面，我们还是来依次讲解一下：

**Request name:** 按需来填写。

**Method 与 URL:** 同理，按需填写，这里需要注意的是它的 URL 里有提示可以使用 `{{@api.url}}` 变量，这个可以在同界面的左侧去进行定义和赋值。

**AUTH、PARAMS 等等:** 关于接口头部、认证、入参等一些要素，用单选按钮进行选择后在下方具体的内容框中填入，具体的就不再展开了，都是基础技能。

**Results:** 接口的结果显示区，发送请求后的结果会直接显示在此区域。

**Validation and variable assignment:** 这里可以进行测试的断言、变量、脚本的一些设置。

The screenshot shows the 'Request details' section with a 'Request name' field. Below it, the 'Method' is set to 'GET' and the 'URL' is set to '{{@api.url}}'. There is a 'Send' button to the right. Below the URL field, there are tabs for 'AUTH', 'PARAMS', 'HEADERS', 'BODY', 'PRE-REQUEST', and 'SETTINGS'. Under the 'BODY' tab, there are radio buttons for 'none', 'formdata', 'x-www-form-urlencoded', 'raw', 'binary', and 'graphql'. Below this is the 'Results' section with a message: 'Click the request "Send" button and see the response here'. At the bottom is the 'Validation and variable assignment' section with tabs for 'ASSERTIONS', 'VARIABLES', and 'SCRIPT'. An 'Add' button is present. Below the 'ASSERTIONS' tab, there is an assertion: 'Assert Status equals 200'.

这里我们设置一个登录界面的登录功能接口，输入一些必要的接口设置条件后直接点"Send"，来看下接口运行的结果。

The screenshot shows the 'Request details' section for a login endpoint. The 'Request name' is 'test\_001\_user\_login'. The 'Method' is 'POST' and the 'URL' is a redacted HTTPS address. The 'Send' button is highlighted with a red box. Below the URL field, there are tabs for 'AUTH', 'PARAMS', 'HEADERS', 'BODY', 'PRE-REQUEST', and 'SETTINGS'. Under the 'BODY' tab, there are radio buttons for 'none', 'formdata', 'x-www-form-urlencoded', 'raw', 'binary', and 'graphql'. The 'raw' option is selected. Below this is the 'Type' dropdown set to 'JSON'. Below the 'Type' dropdown, there is a JSON payload: `1 [{"logincode": "...", "loginType": 1, "machineId": "...", "machineName": "...", "machineType": "...", "password": "..."}]`



运行接口后，在结果区域会显示一些接口的运行情况，包括状态、执行时间、包的大小和出参等具体内容。

**Results** Status: 200 OK Time: 3546 ms Size: 15696 B

https://

REQUEST RESPONSE ASSERTIONS ERRORS CONSOLE LOGS

Headers Body

Show

```
{
  "msg": "操作成功",
  "requestUUID": "fc081be98ec8f09c8",
  "responseTime": "2024-05-22 16:53:48",
  "ret": "0",
  "success": "0",
  "val": {
    "businessTypeId": "1",
    "channelType": "30",
    "companyId": "1",
    "completeStatus": "1",
    "currentId": "1",
    "currentName": "1",
    "currentVersion": "1",
    "currentEndTime": "1760284800000",
    "currentExpiredYn": "0",
  }
}
```

有了以上这些信息后，我们就可以在断言区域进行对应的业务断言了，每次创建接口测试步骤的时候都会有一个默认的断言，判断接口的返回状态码是否为 200。但这个见仁见智，有些公司的业务可能只要服务响应就 200，所以还是要根据自己的实际业务去进行断言的设置，千万别轻易的使用默认的断言了。

**Validation and variable assignment**

ASSERTIONS VARIABLES SCRIPT Add

Description: status\_assertions

Target\*: Status Response status code is 200

Assert type: Equals Value: 200

接口执行完成后，在界面的左侧将会将本次的用例执行结果显示出来，此时注意如果要退出该界面前需要先保存这个测试活动，不然整个测试活动就会被废弃，导致从第一步开始录入的信息全部丢失。（或者直接点退出，在二次确认框中进行保存操作。）



1. POST test\_001\_user\_login

✓ Passed

+ Add step

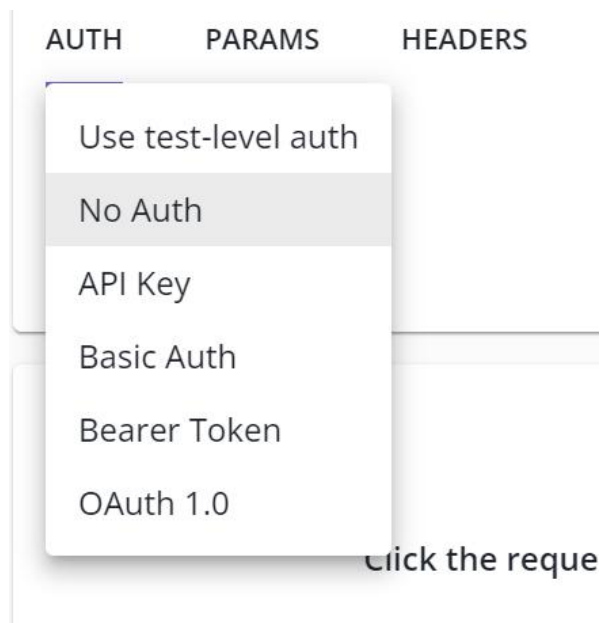
### Variables

Name	Default	Current
api.url		https://wel

Add row

这里额外说一下关于 **mabl** 中接口测试身份认证，在我们的日常业务中部分的接口是需要进行身份认证与授权的，所以在我们创建和编辑测试用例的时候，对应的认证信息也是要进行正确的填写。

如果要进行对应的身份认证，我们在"Request details"中是需要对"AUTH"（认证）项进行一些设置的，我们可以在对应的下拉框内进行对应的认证类型选择。



具体的类型根据各自的业务来进行选择，这里我们举个例子，比如我们的业务端口的认证使用 **API Key** 的形式，那我们就选择"API Key"类型，然后在对应的输入框填入必



要信息即可。最下面的"Add to"是用来指定你将认证信息添加至请求的头部还是内容中的。

AUTH   PARAMS   HEADERS   BODY   PRE-REQUEST   SETTINGS

Authentication  
API Key

Key

Value

Add to  
Header

以上这些就是基础的一个接口测试用例的创建过程，当然实际的肯定不会是那么简单的流程。有了这些步骤后，我们就可以对基础的用例进行后续的设计了。首当其中的肯定是变量的应用，无论是日常的测试执行还是 DDT，变量的重要性就不言而喻了。

我们从测试活动列表中找到刚才创建的接口测试，进入后点击"Edit test"，运行训练器之后再次进入刚才的 API 测试编辑器。界面左侧除了显示当前的用例名之外，"Variables"这一栏就是我们设置变量的地方了。当前我们看到变量栏位内已经有一个叫做 `api.url` 的变量名，"Current"（当前）这一栏内是我们当初创建测试活动时被测对象的访问路径地址。

▶ ↺ ✎ | 📁 ⬇

1. POST test\_001\_user\_login

+ Add step

**Variables**

Name	Default	Current
api.url		https://wel

Add row



我们要添加新变量的话，就只需点击下方的"Add row"来添加新的一栏变量，一般来说在我们的请求中的入参都可以在这里进行设置，方便后续统一搞笑的进行参数与具体测试数据的管理。这里我们输入的值就是刚才那个登录接口执行时所需要的入参。

## Variables

Name	Default	Current	
api.url		https://wel	🗑️
loginCode		189	🗑️
password		c5925b8ba	🗑️

Add row

添加完成之后我们就可以直接在用例中开始调用这些变量了，变量的调用方法之前文章已经介绍过了，书写方法大致如下，唯一需要注意的就是调用的时候注意下实际数据的具体类型即可。另外就是 **mabl** 也支持 **postman** 中的动态随机生成数据类型的变量 `{{RandomVariable}}`。

Method: POST URL: `{{@api.url}}api/web/`

AUTH PARAMS HEADERS **BODY** PRE-REQUEST SETTINGS

none  formdata  x-www-form-urlencoded  raw  binary  graphql

Type: JSON

```

1 {
2   "loginCode": "{{@loginCode}}",
3   "loginType": 1,
4   "machineId": "",
5   "machineName": "",
6   "machineType": "",
7   "password": "{{@password}}",
8   "smsCode": "",
9   "smsCodeToken": ""
10 }
```

相较于上面说的手动创建变量，**mabl** 也支持我们从响应中直接创建对应的变量，当我们执行了一个对应的接口后，在最下面的"Validation and variable assignment"内选择"VARIABLES"点击"Add"进行变量的添加。



### Validation and variable assignment

ASSERTIONS **VARIABLES** SCRIPT

Add

展开界面后会看到以下内容，这里简单介绍一下：

**Description:** 变量的描述，自己定义就行。

**Target:** 这里会有多个选择 Header、Status、Size、Json Body，这里根据具体的测试业务要求来进行选择。

**Json Body path:** 这里只是选择了 Json Body 之后对应展示出来的，不同的 target 会有不同的输入内容。

**Assign to variable:** 用来定义变量名。

ASSERTIONS **VARIABLES** SCRIPT

Add

Description

Target\*  
JSON Body

JSON Body path

Found an object with 6 properties

Assign to variable

全部设定完成之后，此变量就会自动添加至左侧的变量栏中，唯一有区别的就是从响应内添加的变量会有一个默认值，当我们再次调用这个变量在入参后，每次执行的结果可能会根据实际业务情况发生改变，所以一般来说我们可以借助这一机制来完成某些关键入参或指标的断言，来判断两次的值是否与默认或预期值一致。





## Variables

Name	Default	Current	
api.url		https://wel	🗑️
loginCode		1891	🗑️
password		c5925b8ba	🗑️
token	829b1d66-	829b1d66-	🗑️

Add row

另外我们可以将之前 UI 自动化测试中所定义完的变量来进行共享，基于 UI 与接口两者会有大量的相同入参的特点，这样就可以利用共享变量来极大的减少冗余变量和情况。

我们可以回到之前设置的 UI 测试活动界面，点击编辑图标后在更新测试信息的界面中勾选"Share variable"选项即可。

The screenshot shows a configuration page for a test. At the top, there is a navigation bar with a back arrow, an edit icon (highlighted with a red box), a save icon, a trash icon, and a forward arrow. Below this is a dropdown menu set to 'master'. The main title is 'tests / RTS-Rubies-DEV-TEST'. Underneath, it says 'Saas software for store operation and management' and 'Updated May 16, 2024 3:43 PM by austin willson <zhaiyinjie@round-table-union.com>'. There is a toggle switch labeled 'On'. At the bottom, there are several tabs: 'DETAILS', 'RUN HISTORY', 'PLANS (0)', 'CHANGE HISTORY', 'COMPARE', and 'PERFORMANCE'.



DESKTOP    MOBILE WEB  


---

 Desktop browser test  
 DataTables ⓘ  
  
 Share variables

最后就是变量同样可以用在断言中，我们在"Validation and variable assignment"中的"ASSERTIONS"中新建一个断言，下面的一些内容与变量中的相同，我们主要来看下最后的"Assert type"，这其中包含的就是一些判断关键字，例如"等于与否、包含与否、存在与否"，那我们在进行响应断言的时候就可以填入之前定义好的一些变量，提高测试维护与执行效率。

ASSERTIONS    VARIABLES    SCRIPT  


---

 Description  
 Target\*  
      
 Found an object with 6 properties  
 Assert type

这里我们设置了一个对于用户认证状态的判断，从图上可以看到我们运行了接口后对应的"completeStatus"字段的值返回了 1，同样我们在此处对于 Json Body 的整体内容进行断言，预期的值使用了变量，当然变量是已经在左侧的变量栏中进行了对应的设置的。这里就只需要执行接口后 mabl 就会自动判断该字段的返回值是否与变量中的预设值是否



相同，从而通过断言的成功与否来反应这个结果。

Description  
user\_authentication

Target\*  
JSON Body

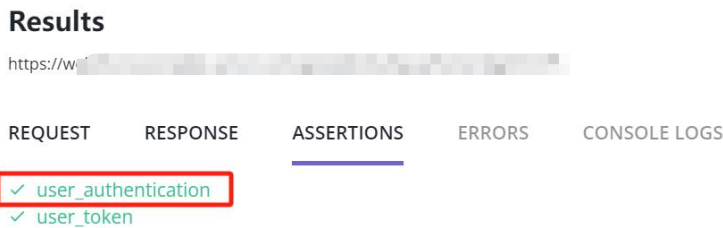
JSON Body path  
val.completeStatus

Found a property with value 1

Assert type  
Equals

Value  
{{@user\_auth\_status}}

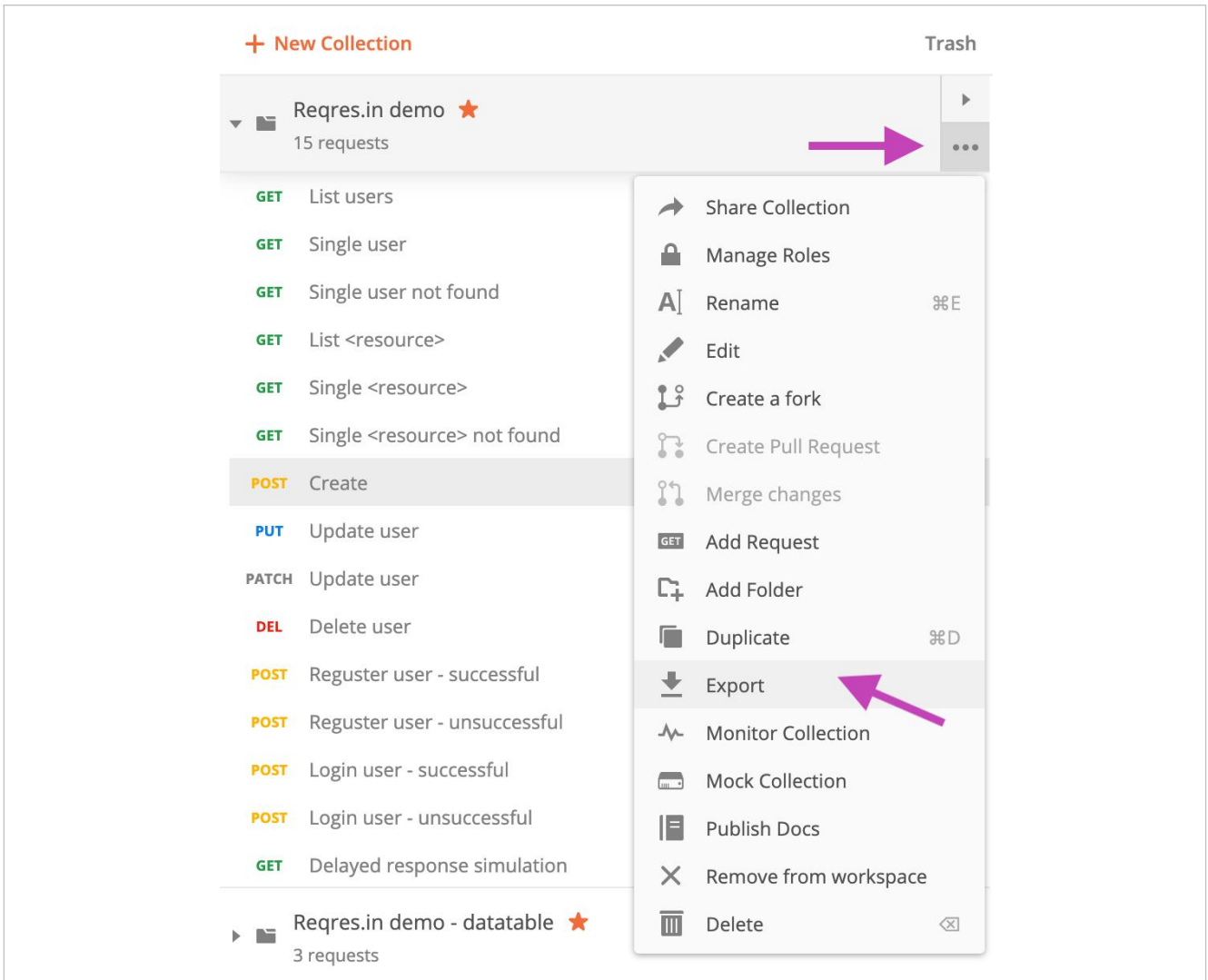
设置完断言后，执行接口测试用例，我们从结果区域来查看断言的执行情况，可与看到这里设置的两个断言都是成功的。



除了我们手动创建接口测试活动之外，我们也可以从导入 **postman** 的接口合集，这样可以实现接口测试用例从 **postman** 到 **mabl** 的无缝衔接。

我们先在 **postman** 中选中一个合集，点击右侧的三个小点，选择其中的"Export"，将其保存为 2.1 版本的 json 文件，这里要声明的是一定要选 2.1 版本的，这个也是 **postman** 官方推荐的导出版本格式。另外还有一个注意点，就是你的测试用例文件大小不能超过 1M，不然是无法导入的，这里推荐的做法就是先在 **postman** 中预先做好分割，导入 **mabl** 后再做整合即可。

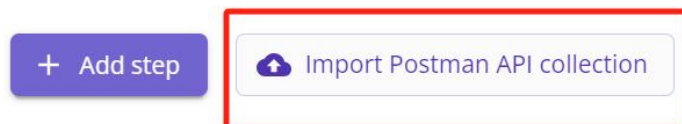




导出后，我们在 **mabl** 创建接口测试活动的界面选择 "Import Postman API collection" ，此时会弹出文件选择窗口，我们选择之前导出的 **postman** 的 **json** 文件。

## Create an API test

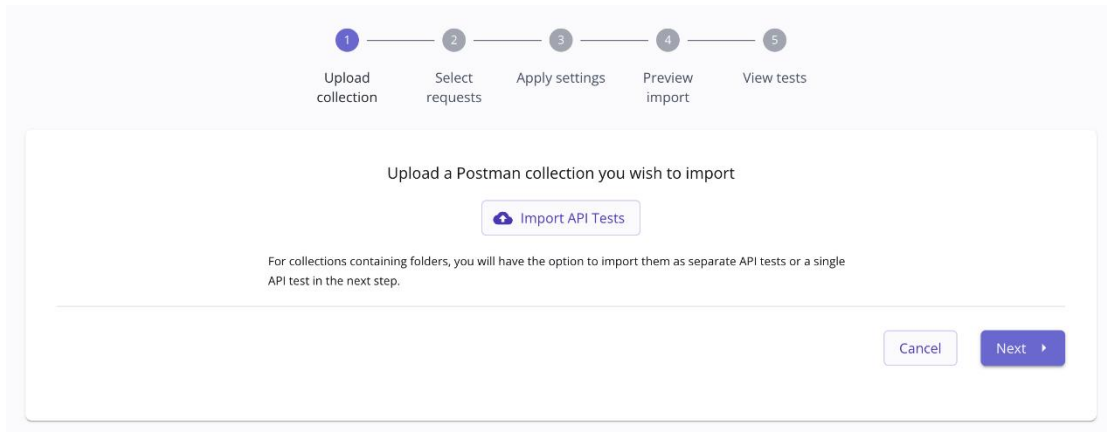
Get started by adding a step or importing a Postman collection



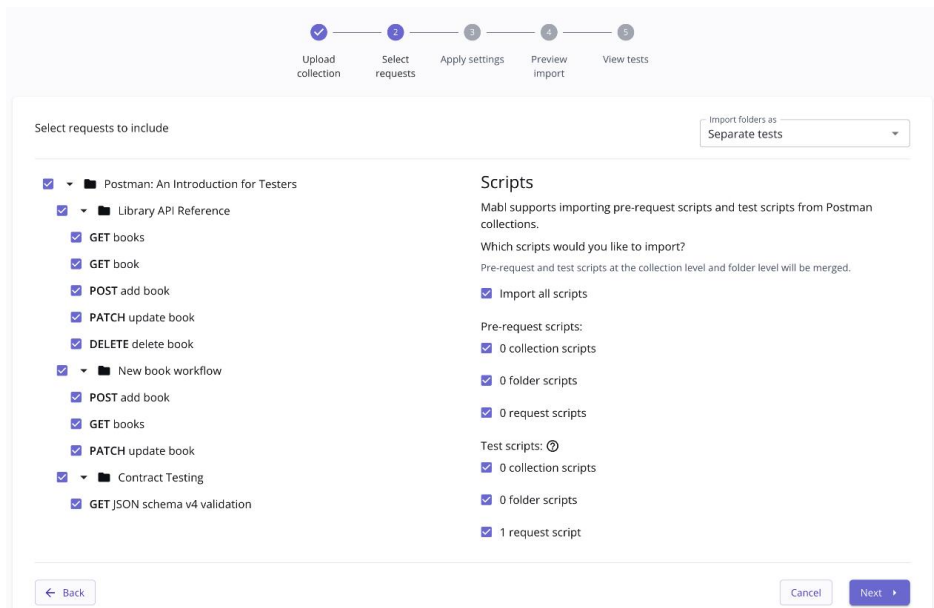
You can import collections without folders above. In order to import a collection with folders, please follow our documentation [here](#)



此步骤跟着引导一步步操作即可，点击"Next"。

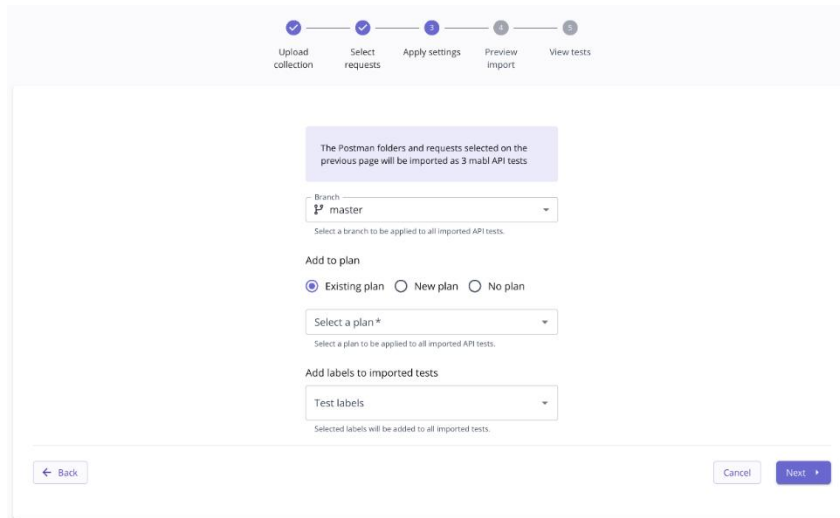


此步骤我们需要选择对应的请求，这里如果有内容结构的话可以按照左侧的实际文件结构进行勾选，全要的话全选即可。

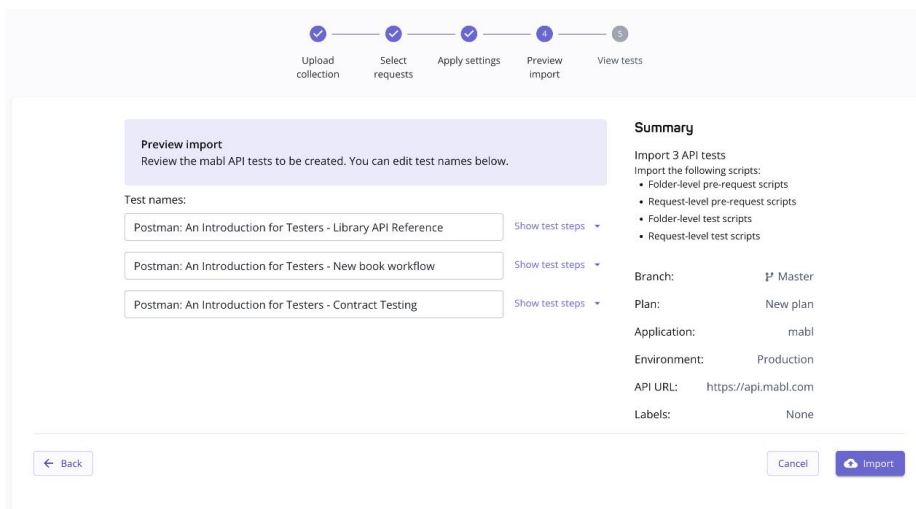


这一步是需要将导入的接口测试用例与你已创建的接口测试活动或应用进行关联。



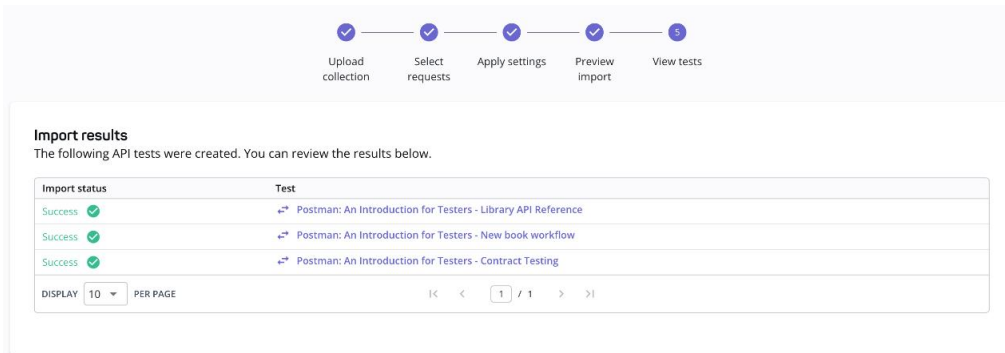


关联完毕后会跳转至确认页面，这里会显示所有导入的相关信息，做最后的确认，如果确认无误的话就可以点击"Import"按钮执行导入操作。



耐心等待导入完毕，最后的界面会显示导入结果，点击右侧的测试活动链接可以快速的跳转至对应的测试界面。





好了，至此，关于在 **mabl** 平台上设计、执行接口测试活动的基础操作已经大致介绍完了，其实 **mabl** 上的接口测试优势远远不止这些，后续将会为大家带来更多的接口测试相关的 **mabl** 平台玩法与最佳实践。相信对 **mabl** 平台感兴趣的同学也会在日后的持续学习与使用过程中总结出适合自己与团队风格的各类测试手段。

## 拓展学习

[2] 人工智能深度学习全栈开发工程师

咨询：微信 **atstudy-js** 备注：AI



# Playwright 实战

◆ 作者：晴空

## 一：实施的背景

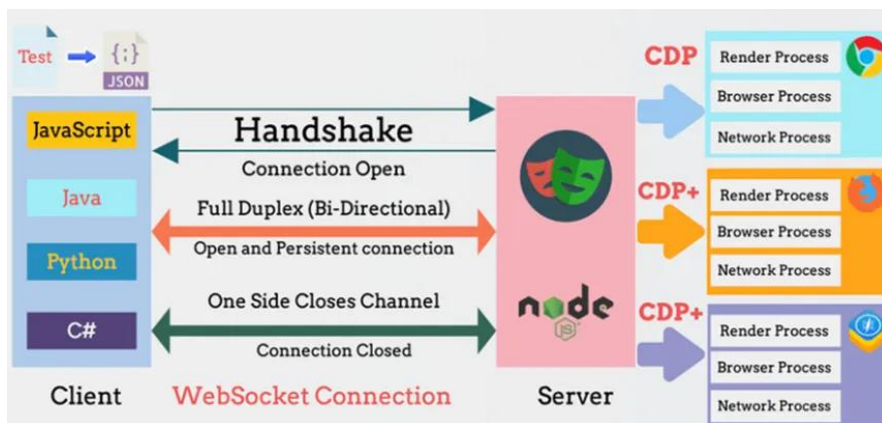
相信使用过 Selenium WebDriver 的小伙伴对其最大的诟病有 3 点，一是浏览器的 driver 和版本对应问题，第二是 Selenium 的执行速度，最后一个槽点是对页面元素文本值的断言非常不便。在我们长期维护大量 UI 自动化测试用例的过程中这两个痛点会让我们耗费不少精力和时间。

## 二：Playwright 最佳实践

### 2.1 为何要选择 Playwright?

对 Selenium 的痛点，Playwright 给出了完美的解决方案！

Playwright 的架构：



Playwright 使用 Chrome DevTools 协议与 Chromium 通信。一旦触发测试，client 端代码将被转换为 JSON 格式，然后使用 websocket 协议发送到服务器。





Playwright 通过单个 `websocket` 协议连接传达所有请求，该连接将保持不变，直到所有测试执行完成。由于命令是在单个连接上发送的，因此测试失败或不稳定的可能性较小，并且命令可以快速执行。

Playwright 启动速度比较快,拥有更多的定位方式,不需要安装驱动,能够在多个平台上运行,提供录制功能实现录制用例视屏,使用上来说 Playwright 也比较容易,无需过多封装即可直接使用。

## 2.2 环境部署

安装 playwright 前先配置好 `node` 和 `python` 环境，之后通过 `pip` 来安装 playwright 和其他库。

以 Mac OS 为例，执行如下命令：

安装 playwright 库

```
pip install playwright
```

然后安装 `browsers`(会安装 `chromium`,`firefox`,`webkit` 浏览器)

```
playwright install
```

如果只想安装指定的 `browser`，则执行如下命令

```
playwright install chromium
```

## 2.3 重要的名词

在开始尝试使用 `playwright` 之前，需要先了解它的几个重要概念。

**Browser:** 是一个浏览器实例，代表一个浏览器会话。它是一个全局的上下文，可以包含多个 `BrowserContext`。

使用：创建浏览器实例管理浏览器生命周期：可以启动浏览器、关闭浏览器等。

```
from playwright.sync_api import Playwright
def test_api_show(playwright: Playwright):
    browser = playwright.chromium.launch()
```



**BrowserContext:** 是浏览器中的一个隔离环境，可以包含多个 Page。每个 BrowserContext 有自己的浏览器存储，例如 cookies 和本地存储。

使用：创建新的上下文，管理上下文。可以添加页面、关闭页面等。

```
from playwright.sync_api import Playwright

def test_api_show(playwright: Playwright):

    browser = playwright.chromium.launch()

    context = browser.new_context()
```

**Page:** 是浏览器中的一个标签页，可以进行页面导航、操作 DOM 元素、捕获页面截图等。

使用：导航页面，操作 DOM 元素，捕获页面截图，处理页面事件。

```
from playwright.sync_api import Playwright

def test_api_show(playwright: Playwright):

    browser = playwright.chromium.launch()

    context = browser.new_context()

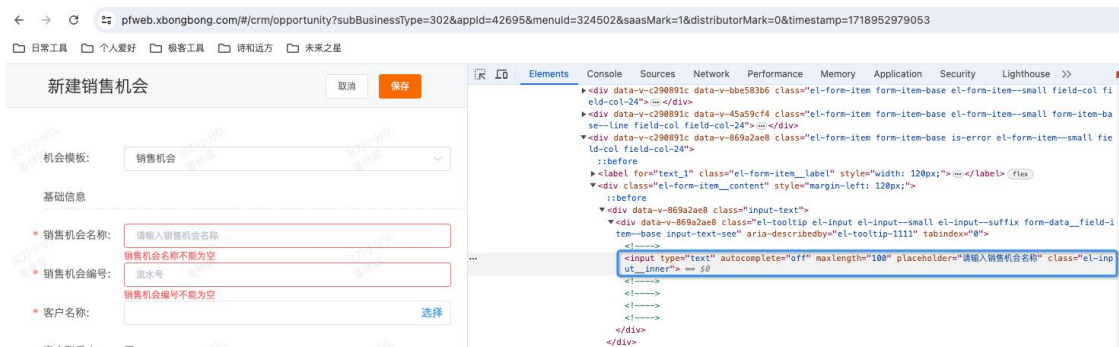
    page = context.new_page()

    page.goto("")
```

## 2.4 元素定位

元素定位是 PlayWright 的核心部分，我们会详细演示常用方法的使用。

**get\_by\_placeholder:** 根据页面元素的 placeholder 属性值定位。



以上图为例，我们想定位<销售机会名称>输入框可以使用 get\_by\_placeholder 方法。

```
page.get_by_placeholder("销售机会名称").fill("商机名称")
```

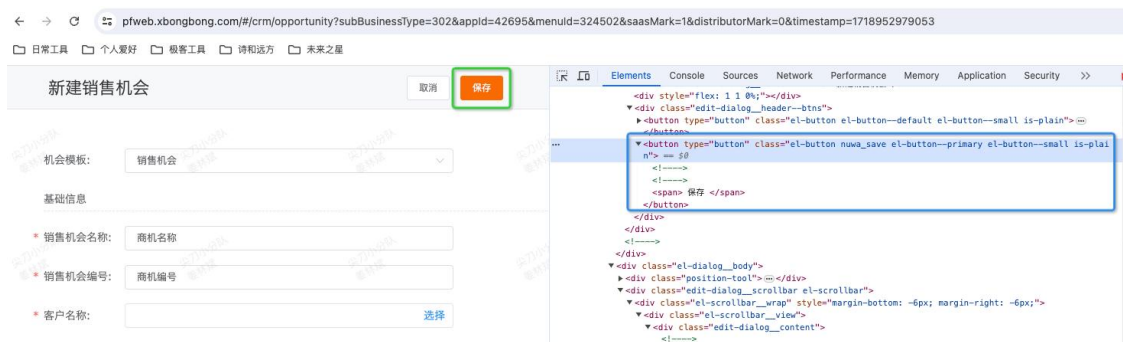


`get_by_role`: 允许通过元素的 ARIA 角色、ARIA 属性和可访问名称来定位它们。

初次使用 `get_by_role` 方法可能会有点懵，因为不知道元素的 `role` 该写为什么！

PlayWright 官网上给出了 `role` 的类型：

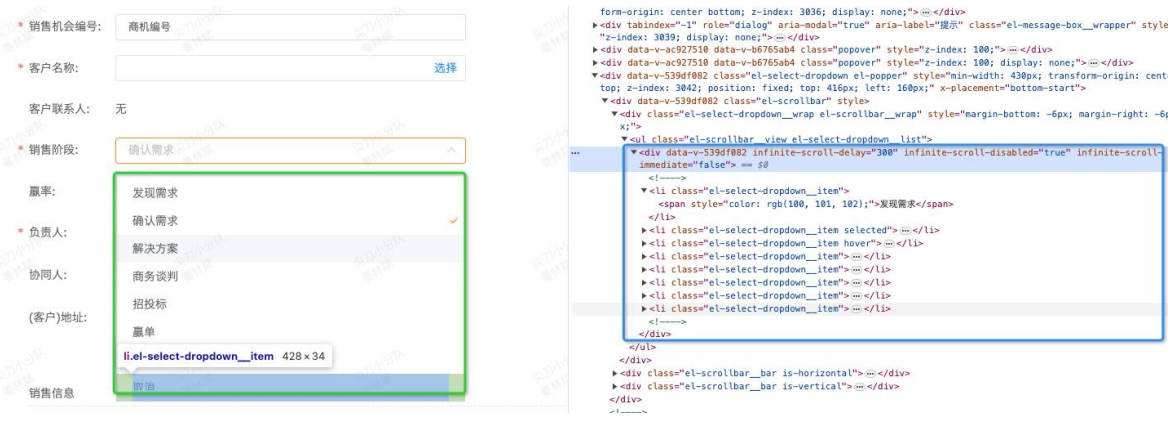
`role?`"alert" | "alertdialog" | "application" | "article" | "banner" | "blockquote" | "button" | "caption" | "cell" | "checkbox" | "code" | "columnheader" | "combobox" | "complementary" | "contentinfo" | "definition" | "deletion" | "dialog" | "directory" | "document" | "emphasis" | "feed" | "figure" | "form" | "generic" | "grid" | "gridcell" | "group" | "heading" | "img" | "insertion" | "link" | "list" | "listbox" | "listitem" | "log" | "main" | "marquee" | "math" | "meter" | "menu" | "menubar" | "menuitem" | "menuitemcheckbox" | "menuitemradio" | "navigation" | "none" | "note" | "option" | "paragraph" | "presentation" | "progressbar" | "radio" | "radiogroup" | "region" | "row" | "rowgroup" | "rowheader" | "scrollbar" | "search" | "searchbox" | "separator" | "slider" | "spinbutton" | "status" | "strong" | "subscript" | "superscript" | "switch" | "tab" | "table" | "tablist" | "tabpanel" | "term" | "textbox" | "time" | "timer" | "toolbar" | "tooltip" | "tree" | "treegrid" | "treeitem"



在上图销售机会新建页右上角，有<取消>和<保存>两个 `button` 组件。定位它们，可以用 `get_by_role`。

```
page.get_by_role("button", name="保存").click()
```





上图中，销售阶段的下拉框选项值是一个 div 元素，除了 xpath 或者 css-selector 方式外很难想到其他发方式来定位到它。但是在 PlayWright 中，我们可以使用 `get_by_role` 来处理它，具体的代码如下：

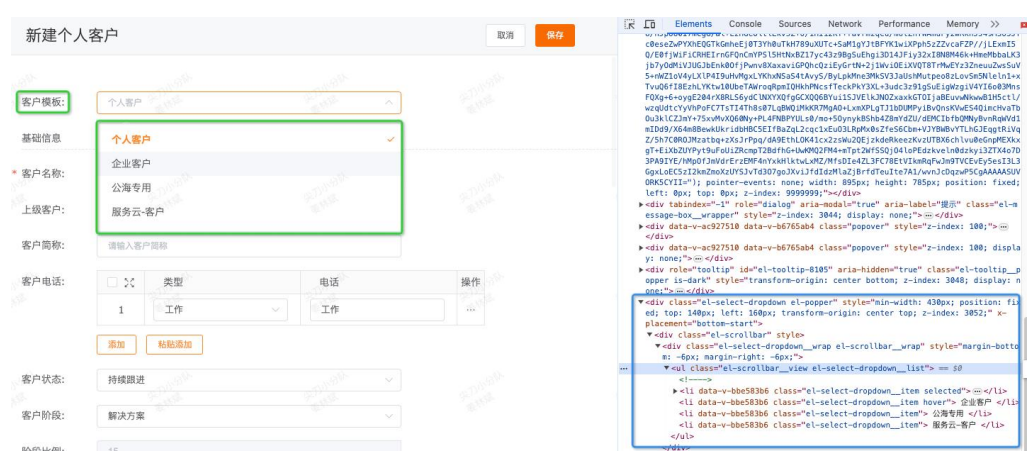
// 点击下拉框选项值的父级元素，即:div。

```
page.locator(parent_locator_path).locator("visible=true").click()
```

// 直接处理选项值，根据 has\_text="" 来选中下拉选项值

```
page.get_by_role("listitem").filter(has_text=item_name).click()
```

**get\_by\_text:** 根据包含的文本值定位元素。



```
page.locator("//div[@x-placement='bottom-start']").get_by_text(model_name,exact=True).locator("visible=true").click()
```

**locator:** 可以在定位器的子树中找到一个与指定选择器匹配的元素。它还可以过滤选项值。



locator 可以接受任何形式定位方式，也就是说 locator\_path 可以是 id, xpath, css, role 等。

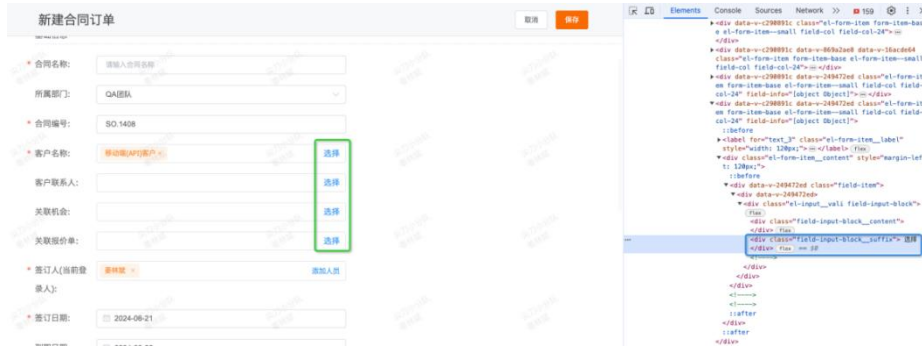
```
locator(locator_path).locator("visible=true").click()
```

```
page.locator("locator_path", has_text="")
```

```
page.locator("locator_path", has_not_text="")
```

### 索引 nth

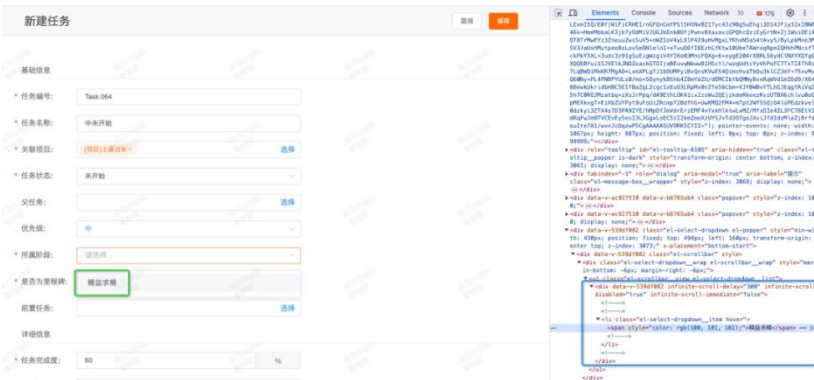
大多数时候，我们会遇到在一个页面上存在多个类似元素的情况。如下图，页面中存在多个类似的元素，这种业务场景下，我们可以考虑先定位一组元素然后根据返回的元素列表的索引来定位到具体的组件。



```
relation_list =page.locator('//div[@class="field-input-block__suffix"]').locator("visible=true")
```

```
relation_list.nth(element_index).click()
```

filter: 这个方法根据选项缩小现有定位器的范围，例如按文本过滤。它可以被链接起来进行多次过滤。



```
page.get_by_role("listitem").filter(has_text=item_name).click()
```



## 2.5 交互动作

在 UI 自动化测试工作中，我们经常用到的页面交互无非是点击，输入，刷新这三个动作。

PlayWright 对上面三种交互动作都提供了支持。

**click:** 点击元素

```
page.get_by_placeholder(place_holder, exact=True).click()
```

**fill:** 输入文本

```
page.locator(locator_path).locator("visible=true").fill(input_context)
```

**reload():** 重载页面

```
page.reload()
```

**select\_option:** 从下拉选项值中选择

官网示例的页面元素：

```
<select multiple>
  <option value="red">Red</div>
  <option value="green">Green</div>
  <option value="blue">Blue</div>
</select>
```

使用 **select\_option** 的示例：

```
element.select_option("blue")
element.select_option(label="blue")
element.select_option(value=["red", "green", "blue"])
```

**set\_checked:** 设置复选框或单选按钮元素的状态

```
page.get_by_role("checkbox").set_checked(True)
```

## 2.6 强大的断言

PlayWright 自带断言功能，并且我们可以很方便地获取到页面里各种容器元素下所有的文本值。



```
all_inner_texts/inner_text
```

```
# 获取容器内的文本值
```

```
def get_inner_text_on_list(self, data_key_name):
```

```
    # 根据列表页的主键获取一行数据的文本值(返回的是 str, 也是奇葩)
```

```
    text_list = self.page.get_by_role("row", name=data_key_name).locator("visible=true").inner_text()
```

```
    return text_list
```

断言 expect

```
# 判断系统里一闪而过的 tips 文案
```

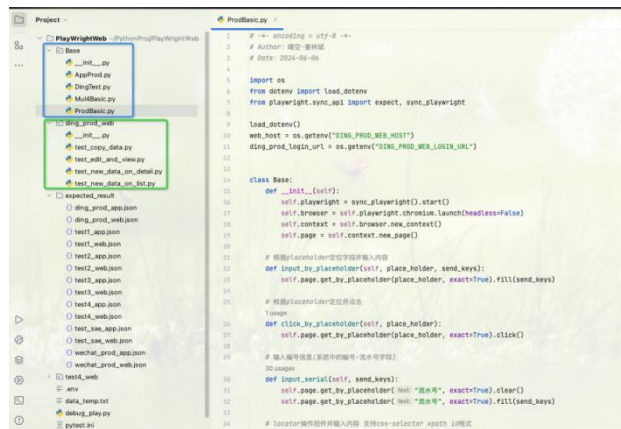
```
def assert_action_success(self):
```

```
    expect(self.page.locator('//p[@class="el-message_content"]')).to_contain_text("成功")
```

## 2.7 PageObject 模式封装

PO 模式不作过多介绍了, 相信读者朋友们或多或少都有了解。下面直接上代码。

封装 playwright 的基础操作。



Base 包里封装了页面元素的基本操作, 包含了元素定位, 点击输入, 断言等通用功能点的封装。

```
# -*- encoding = utf-8 -*-
```

```
# Author: 晴空-姜林斌
```

```
# Date: 2024-06-06
```

```
from playwright.sync_api import sync_playwright, expect
```



```

web_host = "https://appwebfront.xbongbong.com/"

class Base:

    def __init__(self):

        self.playwright = sync_playwright().start()

        self.browser = self.playwright.chromium.launch(headless=False)

        self.context = self.browser.new_context()

        self.page = self.context.new_page()

        # 地址路由

        self.url_dict = {"MULTI_PROD_WEB_MARKET_LIST":
"/#/market-manage/market-activity?subBusinessType=8100&appId=185800&menuId=2000341&saasMark=1&distributorMark=0",

                        "MULTI_PROD_WEB_CLUE_LIST":
"/#/market-manage/sales-leads?subBusinessType=8001&appId=185800&menuId=2000343&saasMark=1&distributorMark=0"}

        # 根据 placeholder 定位字段并输入内容

        def input_by_placeholder(self, place_holder, send_keys):

            self.page.get_by_placeholder(place_holder, exact=True).fill(send_keys)

        # 根据 placeholder 定位并点击

        def click_by_placeholder(self, place_holder):

            self.page.get_by_placeholder(place_holder, exact=True).click()

        # 输入编号信息(系统中的编号-流水号字段)

        def input_serial(self, send_keys):

            self.page.get_by_placeholder("流水号", exact=True).clear()

            self.page.get_by_placeholder("流水号", exact=True).fill(send_keys)

        # locator 操作控件并输入内容 支持 css-selector xpath id 格式

        def input_by_locator(self, locator_path, input_context):

            self.page.locator(locator_path).locator("visible=true").fill(input_context)

        # 根据 locator 定位组件并点击

        def click_by_locator(self, locator_path):

            self.page.locator(locator_path).locator("visible=true").click()

            self.sleep(500)

        # 点击 button 组件

        def click_button(self, button_name):

            self.page.get_by_role("button", name=button_name, exact=True).click()

```





```

        if str(button_name) in "保存修改删除":
            self.sleep(2000)
        else:
            self.sleep(1000)
# 根据下列选项的值选择
def set_opportunity_stage(self, item_name):
    self.click_by_locator("//form/div[7]/div/div/div/div[1]/span")
    self.page.get_by_role("listitem").filter(has_text=item_name).click()
# 设置任务的所属阶段
def set_task_stage(self, item_name):
    self.click_by_locator("//form/div[8]/div/div/div/div/input")
    self.page.get_by_role("listitem").filter(has_text=item_name).click()
# 设置风险的所属阶段
def set_risk_stage(self, item_name):
    # 设置任务的所属阶段
    self.click_by_locator("//form/div[8]/div/div/div/div[1]/input")
    self.page.get_by_role("listitem").filter(has_text=item_name).click()
# 设置企微销售机会的阶段
def set_wechat_opportunity_stage(self, item_name):
    self.click_by_locator("//form/div[7]/div/div/div/div[1]/span")
    self.page.get_by_role("listitem").filter(has_text=item_name).click()
# 关闭详情页
def close_detail(self):
    self.click_by_locator("//div[@class='title__close-btn']")
# 新建页切换模板
def switch_model(self, model_name):
    self.click_by_placeholder("请选择模板")
    self.page.locator("//div[@x-placement='bottom-start']").get_by_text(model_name,
exact=True).locator("visible=true").click()
    self.sleep(1000)
# 选择单选
def select_single_member(self, add_index, member_name):
    radio_button_list = self.page.locator("//div[@class='user-radio-btn']").locator("visible=true")

```



```

        radio_button_list.nth(add_index).click()

        self.page.locator('//div[@class="person-tabs"]').get_by_placeholder("请输入内容")
        .fill(member_name)

        self.sleep(1000)

        self.page.locator('//div[@class="user-tab-content__user"]/div/label/span[1]').click()

        self.sure_on_dialog()

# 服务云 独用的成员单选

def single_team_for_service_cloud(self, add_index, member_name):

    radio_button_list =
self.page.locator('//div[@class="user-radio-btn-block"]').locator("visible=true")

        radio_button_list.nth(add_index).click()

        self.page.locator('//div[@class="person-tabs"]').get_by_placeholder("请输入内容")
        .fill(member_name)

        self.sleep(1000)

        self.page.locator('//div[@class="user-tab-content__user"]/div/label/span[1]').click()

        self.sure_on_dialog()

# 点击超链接

def partial_link(self, link_text):

    locator_path = 'a:has-text("' + str(link_text) + "')'

    self.page.locator(locator_path).locator("visible=true").click()

    self.sleep(1000)

# 单据详情页 更多-关联新建

def new_on_detail(self, business_name):

    link_business_locator = 'li:has-text("' + str(business_name) + "')'

    self.page.get_by_role("button", name=" ", exact=True).click()

    self.page.locator(link_business_locator).click()

# 单据新建/编辑页 选择所有关联产品

def select_all_products(self):

    self.page.locator("span:has-text('添加产品)').locator("visible=true").click()

    self.page.locator("//table/thead/tr/th[1]/div/label/span").locator("visible=true").click()

    self.click_button("确定")

# 单据详情页查看 更多-tab 栏

def view_more_tab_on_detail(self, tab_name):

```



```

self.page.locator("//div[@class='detail-tabs-more-button']").locator("visible=true").click()
self.page.wait_for_timeout(1000)

tab_path = "li:has-text('" + str(tab_name) + "')"

self.page.locator(tab_path).locator("visible=true").click()

# 详情页的复制/编辑/删除

def copy_and_edit_on_detail(self, action_name):

    button_list =
self.page.locator("//div[@class='detail-permission-bts']/div').locator("[type='button']").all()

    for button_element in button_list:

        if button_element.inner_text().__contains__(action_name):

            button_element.click()

            self.sleep(1000)

            break

        else:

            continue

# 列表搜索

def search_on_list(self, search_condition):

    self.page.locator("//div[@class='search']/div/input').fill(search_condition)

    self.page.locator("//div[@class='search']/div/div').locator("visible=true").click()

# 列表页新建

def new_on_list(self):

    self.click_button("新建")

# 详情页的复制-编辑-删除

def common_action_on_detail(self, action_name):

    locator_name = " " + str(action_name)

    self.page.get_by_role("button", name=locator_name).locator("visible=true").click()

# 跳转到指定列表

def go(self, url_key):

    list_url = self.url_dict[url_key]

    self.page.goto(web_host + str(list_url))

    self.sleep(3000)

# 强制等待，单位毫秒

def sleep(self, wait_duration):

```



```

        self.page.wait_for_timeout(wait_duration)

# 刷新页面

def refresh(self):

    self.page.reload(timeout=10000)

# 关联数据选择 支持系统和自定义

def select_relation_data(self, model, element_index, relation_data_name):

    if str(model) == "saas":

        saas_relation_list =
self.page.locator("//div[@class='field-input-block__suffix']").locator("visible=true")

        saas_relation_list.nth(element_index).click()

        # 搜索关联数据

        self.page.get_by_placeholder("搜索",
exact=True).locator("visible=true").fill(relation_data_name)

        self.sleep(1000)

        self.page.locator("//table[@class='el-table__body']/tbody/tr/td[1]/div/label').locator(
            "visible=true").click()

        self.sure_on_dialog()

    elif str(model) == "paas":

        paas_relation_list =
self.page.locator("//div[@class='relation-data-input__right']").locator("visible=true")

        paas_relation_list.nth(element_index).click()

        # 搜索关联数据

        self.page.get_by_placeholder("请输入内容",
exact=True).locator("visible=true").fill(relation_data_name)

        self.sleep(1000)

        self.page.locator("//table[@class='el-table__body']/tbody/tr/td[1]/div/label').locator(
            "visible=true").click()

        self.sure_on_dialog()

# 非 nvwa 业务模块的关联数据选择

def set_not_nvwa_relation_data(self, element_index, relation_data_name):

    relation_list =
self.page.locator("//div[@class='field-input-block__suffix']").locator("visible=true")

    relation_list.nth(element_index).click()

    # 搜索关联数据

```



```

        self.page.get_by_placeholder("搜索",
exact=True).locator("visible=true").fill(relation_data_name)

        self.sleep(1000)

        self.page.locator("//table[@class='el-table__body']/tbody/tr/td[1]/div/span/label").locator(
            "visible=true").click()

        self.sure_on_dialog()
# 在列表全选后进行批量删除/彻底删除
def batch_del_on_list(self, batch_action_name):
    # 列表页全选

    self.page.locator("#select-all").locator("visible=true").click()

    self.click_button(batch_action_name)

    self.sure_on_dialog()
# 在系统的对话框中点击确定
def sure_on_dialog(self):
    self.click_button("确定")

    self.sleep(1000)
# 切换详情页 tab
def switch_tab(self, tab_name):
    self.sleep(1000)

    tab_list = self.page.locator("//div[@role='tablist']").locator("//div[@role='tab']").locator(
        "visible=true").all()

    for tab in tab_list:
        if str(tab.inner_text()).__contains__(tab_name):
            tab.click()

            self.sleep(1000)

            break
        else:
            continue
# 获取容器内的文本值
def get_inner_text_on_list(self, data_key_name):
    # 根据列表页的主键获取一行数据的文本值(返回的是 str, 也是奇葩)

    text_list = self.page.get_by_role("row",
name=data_key_name).locator("visible=true").inner_text()

```



```

        return text_list

    # 获取基本信息栏容器内的文本值

    def get_inner_text_on_basic(self):

        basic_text = self.page.locator("div.base-detail__bottom--left >
form").locator("visible=true").inner_text()

        return basic_text

    # 获取单据详情页-负责团队 tab 信息

    def get_sales_team_info(self):

        team_text =
self.page.locator("//div[@class='sale-team-detail__body']").locator("visible=true").inner_text()

        return team_text

    # 获取 tab 栏下的内容 model 分为 base 和 more 分别对应基本 tab 和更多 tab

    def get_inner_text_of_tab(self, model, tab_name):

        if str(model) == "base":

            self.switch_tab(tab_name)

        elif str(model) == "more":

            self.view_more_tab_on_detail(tab_name)

        else:

            pass

        # 获取 tab 栏下的 inner-text

        tab_context =
self.page.locator("//div[@class='detail-tab-containt']").locator("visible=true").inner_text()

        return tab_context

    # 获取编辑页的 inner-text

    def get_inner_text_of_update(self):

        update_page_text = self.page.locator("//div[@class='edit-dialog__content']").locator(
            "visible=true").inner_text()

        return update_page_text

    # 判断新建/编辑/删除成功

    def assert_action_success(self):

        action_tips_list =
self.page.locator("//p[@class='el-message__content']").locator("visible=true")

        expect(action_tips_list.nth(0)).to_contain_text("成功")

```



## 测试用例：

考虑到实际的业务场景，笔者在该项目里并没有封装具体的页面而是封装好了 base 通用操作操作后直接使用了，如果读者朋友们有需要可以自行封装页面的操作。

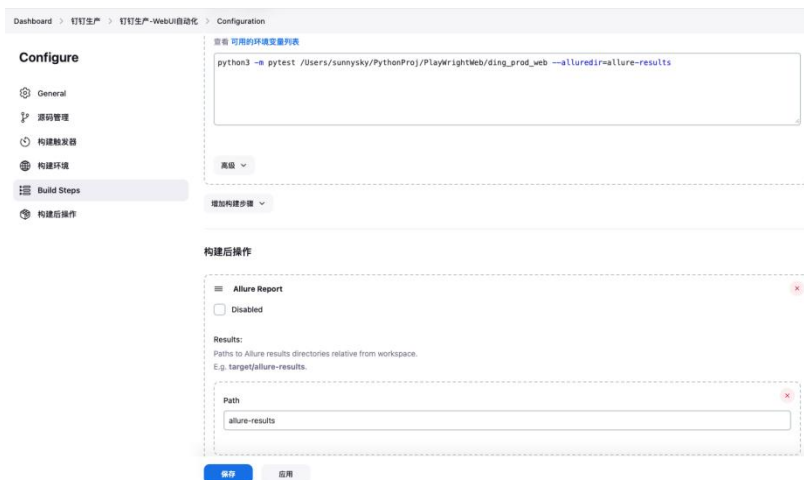
ding\_prod\_web 是具体的 UI 自动化测试用例的目录，具体的结构如下图。

```

1  -*- encoding = utf-8 -*-
2  # Author: 晴空-姜林斌
3  # Date: 2024-06-06
4
5
6  import unittest
7  from Base.MultiProd import Base
8
9
10 class TestNewDataOnList(unittest.TestCase):
11     @classmethod
12     def setUpClass(cls):
13         cls.base = Base()
14         # 登录独立版
15         cls.base.page.goto("https://appwebfront.xbongbong.com/stand-alone-login.html/")
16         cls.base.page.get_by_placeholder("请输入手机号").fill("13912345678")
17         cls.base.page.get_by_placeholder("请输入密码").fill("12345678")
18         cls.base.page.locator("//span[@class='el-checkbox__inner']").click()
19         cls.base.page.get_by_role("button", name="登录", exact=True).click()
20         cls.base.page.get_by_text("独立版").click()
21         cls.base.page.get_by_role("button", name="立即进入", exact=True).click()
22         cls.base.page.wait_for_timeout(5000)
23
24     def test_new_market_on_list(self):
25         self.base.go("MULTI_PROD_WEB_MATERIAL_LIST")
26         self.base.new_on_list()
27         self.base.input_by_placeholder(place_holder="请输入活动名称", send_keys="PW列表页创建")
28         self.base.click_button("保存")
29         self.base.assert_action_success()
30         self.base.search_on_list("PW列表页创建")
31         self.base.partial_link("PW列表页创建")
32         self.base.copy_and_edit_on_detail("删除")
33         self.base.sure_on_dialog()
34         self.base.assert_action_success()
    
```

## 2.8 集成到 Jenkins

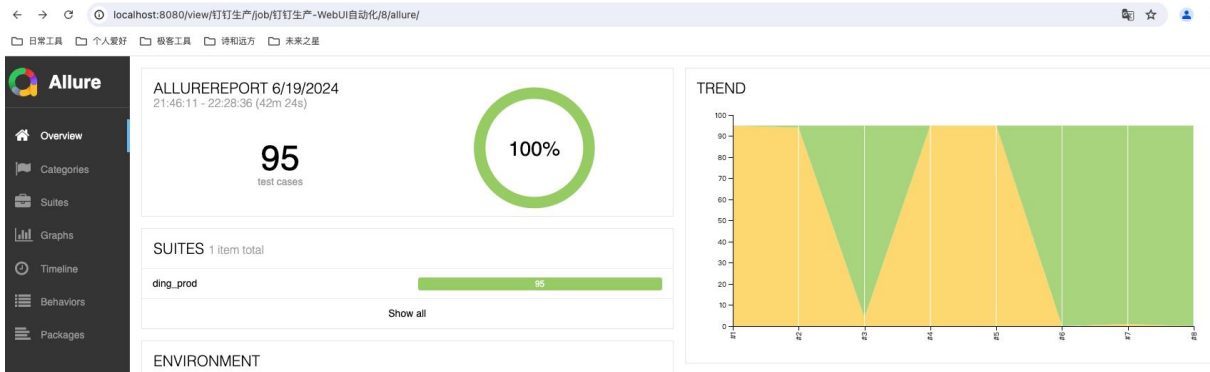
在 Jenkins 里配置自由风格的 Job



构建步骤是执行 sh

```
python3 -m pytest /Users/sunnysky/PythonProj/PlayWrightWeb/ding_prod_web  
--alluredir=allure-results
```

构建后的操作配置为 Allure Report, 以下是生成的 Allure 报告



### 三：展望未来

人工智能技术将对自动化测试产生深远的影响。人工智能技术通过大数据、AI 和机器学习，机器将学会如何测试；这将对自动化测试进一步提升测试的有效性。自动化测试的自愈技术、大数据预测、图象识别等技术，将使得测试更加智能化、精准化。

结合了 LLM 的 AI 框架开始小荷才露尖尖角：<https://github.com/Skyvern-AI/skyvern>。

作为测试人员，我们需要不断学习新技术，适应新变化，以确保我们的测试工作能够为软件质量提供坚实的保障。

希望我的分享能为大家带来一些启发和帮助。让我们共同期待自动化测试技术更加美好的未来！

#### 拓展学习

[3] 云端全链路测试技术全栈学习

咨询：微信 atstudy-js 备注：全栈

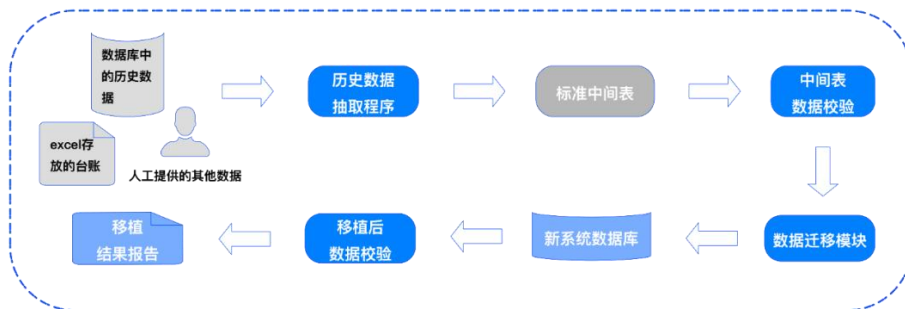




# 数据迁移测试

◆作者：有房车的直男

相信很多同学在测试过程中会遇到新老系统数据迁移的情况，这时候就需要进行数据迁移的测试了，数据迁移测试是在数据从一个系统或环境迁移到另一个系统或环境时执行的测试过程。这种类型的测试旨在确保数据在迁移过程中的完整性、一致性、准确性和安全性。



## 一. 数据迁移面临的风险

### 1. 迁移时间成本

一般来说要进行数据迁移的系统都是大型甚至超大型的，这种情况下要迁移的数据都是数以亿计的，在生产环境的数据库上进行迁移，如果按照顺序迁移，则可能花费几个月甚至几年的时间，这是相当恐怖的。

### 2. 迁移过程中服务是否停止

很多系统都是要时刻为用户提供服务而产生效益，现在基本上所有的系统升级、版本迭代都是在线进行，你玩的王者和吃鸡游戏就是典型，如果停机去迁移维护那会导致很大的损失，所以迁移测试时也要尽量保证服务 100%处于可用状态。



### 3.数据的正确性、可用性、一致性

正确性：大家都知道老系统的很多代码都是直接使用数据库表中的数据，那么在迁移的过程中，要确保新系统也能取到对应的数据；

可用性：无论是迁移前，迁移过程中还是钱以后，数据都是可用状态；

一致性：在迁移过程中，要确保迁移过来的数据和旧系统的数据一致。

## 二. 迁移前的准备工作

### 1.确认此次迁移的范围

这个工作很有可能是开发、运维和测试等众多角色参加，通过分析原有系统和现有系统的功能模块、大致的处理流程，分析两者的区别，以及历史数据对用户的影响程度等，在充分评估工作量及人力资源的情况下划定数据迁移的范围，因为对于大型项目来说，数据全部迁移花费的成本太大。

### 2.其次需要对比不同版本之间的区别

如果你对整个项目模块不太清楚的（比如我就当过空降兵去临时支援一个项目数据迁移），这种情况可以让产品、项目经理，运营提供新老系统功能差异对照文档，包括迁移之后，每个版本相关的功能迁移成新版本之后对应的功能变化，以及数据要如何处理，这个也需要出一份文档，可以方便在迁移的时候，对照着文档进行测试，容易看出来新旧版本的差异。

当时我们的数据迁移任务重，时间紧，加上人力相对较少，这里就必须要求我们用上一些工具和自动化脚本来加持，我这边也分享一些自己的心得：

**A.数据库对比工具：**可以使用数据库对比工具来比较源数据库和目标数据库之间的数据差异。一些流行的数据库对比工具包括：**Beyond Compare**、**WinMerge**、**SQL Data Compare** 等。

**B.数据校验框架：**可以使用 **Python** 编写数据校验框架，用于对比源数据和目标数据之间的差异。例如，可以使用 **pandas** 库来加载源数据和目标数据，并进行数据比较和校验。



C.数据校验脚本：可以使用 Python 脚本编写数据校验脚本，通过查询源数据库和目标数据库，对比数据差异并生成校验报告。

下面是的Python脚本用于对比两个CSV文件中的数据差异：

```
import pandas as pd
1 usage
def compare_csv_files(file1, file2):
df1 = pd.read_csv(file1)
df2 = pd.read_csv(file2)
diff = df1.compare(df2)
if diff.empty:
print("数据完全一致！")
else:
print("数据差异：")
print(diff)
if __name__ == "__main__":
file1 = "source_data.csv"
file2 = "target_data.csv"
compare_csv_files(file1, file2)
```

### 3.迁移前数据备份

在迁移前将相关业务报表数据导出备份，后续迁移测试进行核对，开发也会将迁移前预埋数据库备份，预防迁移问题需回滚，这就好比你在重装电脑系统前备份重要资料一个道理。

## 三. 迁移测试的执行流程

### 1.需要熟悉迁移方案

首先我们要知道此次迁移采用的是哪种迁移策略，是逐步迁移（Phased Migration），并行迁移（Parallel Run）还是实时迁移（Real-time Sync）？上线后，旧系统设备如何处



理，是废弃还是继续使用。在数据并账方面，流程如何处理，账务如何迁移。在数据同步方面，了解同步时间策略和同步范围策略，是全量同步还是增量同步。

举例：以购物商城系统为例，迁移总体技术方案采用编写迁移 shell 脚本，先在旧库建立与新库业务表相同字段的临时表，在旧库进行数据的筛选、加工至临时表，再将临时表中的数据导出成表数据文件，到 oracle 数据库使用 sqlldr 导入 informix 数据文件。存量数据一次性迁移，增量数据实时迁移方式进行迁移。迁移时点为日初后迁移，新系统上线后，旧系统依然在使用，故测试时不仅要验证迁移数据在新系统的业务验证，还需要对旧系统进行回归验证。

## 2. 制定测试计划

确定数据迁移的测试目标、范围和资源需求。制定详细的测试计划，包括测试的阶段、测试用例设计、测试数据准备等。测试主要通过技术手段和业务手段的验证来确保迁移的正确性。测试前，根据 SIT 各轮次测试安排，规划数据迁移验证时间。通常在 SIT 三个轮次中安排两轮迁移测试，如果有 UAT5 回归测试，可在回归环境验证一轮。迁移涉及环境的重置，如果有外部联调单位，还需根据联调单位来调整迁移测试时间安排。

## 3. 理解业务需求和迁移规则

仔细了解业务需求，包括数据迁移的目的、范围和规则。

确定迁移的数据规则，包括字段映射、数据转换和数据清理等。

## 4. 创建测试用例

根据业务需求和迁移规则，编写详细的测试用例。确保测试用例覆盖了各种情况，包括正常情况和异常情况。

## 5. 准备测试数据

创建测试数据，以模拟实际生产环境中的数据。测试数据应涵盖各种数据类型、数



据量和数据情境,这步喂数据是在测试执行中特别重要的一环,除了要尽量模拟实际生产环境,还要考虑到各种测试点,这边罗列下需要涵盖到的测试点:

- (1) 构造界面字段最长值的数据,测试两边字段长度限制差异
- (2) 各种格式的数据,测试两边字段格式限制差异
- (3) 字段全填的数据,测试两边字段是否会丢失或者迁移错位
- (4) 字段默认值的数据,测试两边字段默认值是否显示一致
- (5) 重要下拉字段的所有情况,测试迁移是否显示正确
- (6) 对旧系统有新系统没有的字段,验证是否迁移过去
- (7) 旧系统没有,新系统有的字段,验证是否给与正确的默认值
- (8) 针对新旧系统字段的唯一值判断构造数据测试验证
- (9) 针对新旧系统字段的是否为空构造数据测试验证
- (10) 字段各种区间数据,测试新旧系统字段的范围限制

## 6.执行数据迁移

执行实际的数据迁移过程,使用测试数据将数据从源系统迁移到目标系统。在迁移期间记录详细的日志,以便在发生问题时进行故障排除,这点非常重要。

## 7.验证数据一致性

验证目标系统中的数据是否与源系统中的数据一致。比较迁移前后的数据,确保每个数据项都正确迁移。

## 8.检查数据完整性

检查目标系统中的数据完整性。确保所有必需的字段和关联数据都正确迁移。



## 9.测试性能和容量

针对大规模的数据迁移，测试迁移的性能和容量，确保在合理的时间内完成数据迁移。

## 10.测试回滚和恢复（这步特别重要）

测试回滚计划，确保在发生问题时可以成功地回滚到迁移前的状态。

恢复测试，验证回滚操作的有效性，切记回滚是迁移出现重大问题后最有效快速的应急方案。

## 11.审计和日志分析

审计迁移后的数据，确保可以追溯每个数据项的变更历史。

分析迁移过程中生成的日志，查找异常情况并进行故障排除。

## 12.生成测试报告

汇总测试结果，包括成功的迁移和发现的问题。

提供建议和改进意见，并记录测试过程中的经验教训。

## 13.更新相关文档

更新相关文档，包括数据字典、数据模型等，以反映新系统中的数据结构，也方便后面相关人员的查验。

通过执行上述步骤，可以确保数据迁移过程是可控的、准确的，并且满足业务需求。在数据迁移测试中，及早发现和解决问题对确保成功的数据迁移至关重要。



## 总结

在软件生命周期中，由于业务发展，系统迭代更新常会遇到需要数据迁移的情况，虽然这种情况在一般公司或中小型项目中比较少见，但是作为测试人员也要掌握该工作流程，未雨绸缪也是一名测试人员的必备素质，这点非常重要！

## 拓展学习

[4] **【Python 自动化测试学习交流群】** 学习交流

咨询：微信 atstudy-js 备注：学习群



# 探索性测试：理论根基、实践精髓 与未来趋势

◆ 作者：William Wang

## 引言

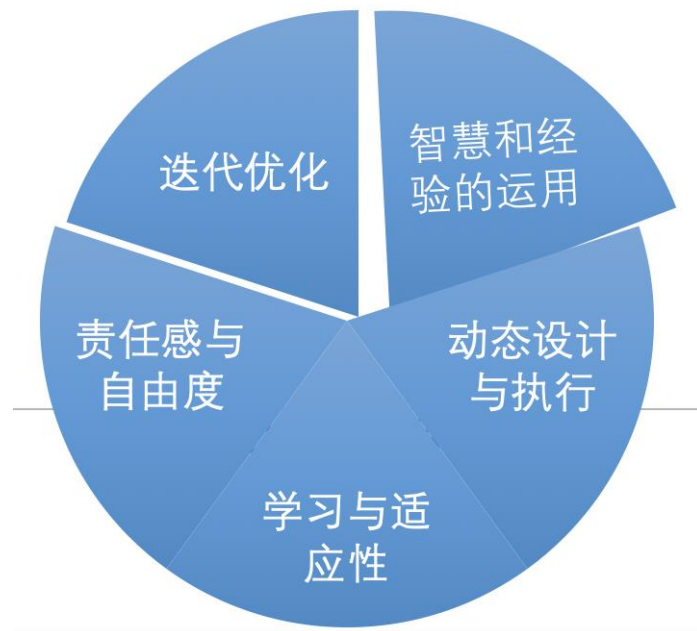
在当今信息化社会，软件已成为支撑各行各业高效运作的关键载体，其质量直接影响到用户体验、业务运行乃至企业竞争力。随着软件复杂度的不断提升以及市场对快速迭代和高质量交付的需求日益迫切，传统的线性、预定义的测试方法逐渐显现出局限性。在这种背景下，探索性测试（Exploratory Testing, ET）作为一种强调测试人员主观能动性、即时创造性和系统行为探索的测试方法，逐渐受到广泛关注并被应用于各类软件项目的测试实践中。在接下来的章节中，笔者会给大家介绍探索性测试的理论知识、实践操作和未来发展，相信通过这篇文章能够让大家对探索性测试方法有更多的了解认知，同时能够熟练运用到提升软件质量和优化测试流程中。

## 一、定义与核心理念

首先我们来了解什么是探索性测试？它是一种非线性、灵活且高度依赖测试人员专业知识与经验的测试方法。它并不依赖于详尽的预定义测试用例集，而是鼓励测试人员在了解系统特性和需求的基础上，运用自身的直觉和创造性思维，实时地设计和执行测试，以发现潜在的软件缺陷和质量问题。其核心理念包括以下几个方面：







探索性测试重视测试人员的才能和专业知识，强调他们在测试过程中的主观能动性和创造性思维。假设一名经验丰富的测试工程师在测试一个新的在线支付系统时，他不仅按照既定的测试案例执行，还根据自己对用户行为的理解，模拟了不同网络环境下的支付流程（如弱网络、断网重连等），以及用户可能的非线性操作序列（比如中途取消支付、切换支付方式等），发现了系统在网络不稳定情况下的处理漏洞。

测试设计和执行是同步进行的，而不是提前完全规划好的。这意味着测试人员一边探索软件一边构造测试用例，能够随着对软件理解的加深和新发现的问题不断调整测试策略。比如，开发团队提供了一个 API 的初步版本，测试工程师开始用 Postman 进行基础的 GET 和 POST 请求测试。在测试过程中，他们发现该 API 在处理大量并发请求时响应缓慢。于是，测试团队立刻设计了一系列高负载测试用例，模拟实际使用中可能出现的高并发场景，并使用 JMeter 进行压力测试。通过实时监控和调整测试策略，他们不仅定位了性能瓶颈，还提出了解决方案建议。

探索性测试强调在测试过程中学习，测试人员持续学习软件的行为模式、系统特性以及潜在问题，根据实时反馈快速适应和优化测试活动。笔者所在公司的电商网站有一次引入了智能推荐系统，测试团队在探索性测试中不仅仅验证推荐算法的基本功能，还通过分析用户行为日志和访谈产品经理，学习用户在使用推荐功能时的习惯和偏好。他们发现，用户在特定时段（如节假日前夕）对某些类别的商品推荐更为敏感。基于此洞



察，测试人员设计了模拟这些特定情境的测试场景，比如高流量时段的系统稳定性测试、个性化推荐准确性的深度测试等，以此来更贴近用户实际使用情况，优化测试效果。这就是测试团队深度挖掘并学习用户的行为模式，然后用来优化测试的成功实践。

测试人员在探索性测试中有较高的个人自由度，同时也承担着相应的责任，即在无严格预定计划的情况下，也要尽可能地发现和预防潜在的问题。在之前一个与外部小游戏程序供应商的合作项目里，笔者所在团队的一位测试工程师基于对玩家行为的假设，设计了一系列创新的测试场景，如快速连续点击、长时间无操作后的游戏状态恢复、以及在游戏内同时进行多个复杂操作的测试，旨在发现可能的性能瓶颈和逻辑错误。这种创造性的测试设计不仅依赖于测试人员的自由度，也体现了其对产品质量负责的态度。

在 AI 驱动的软件产品测试中，团队首次使用了一套基本的测试用例集。通过分析测试结果和机器学习模型的行为，发现模型在处理某些边缘输入时预测准确性下降。基于这些发现，第二个迭代中，测试人员调整了测试策略，引入更多极端和边缘案例，比如极端数值、不常见格式的输入数据等，同时利用自动化工具生成更多此类测试用例，以更全面地挑战模型的边界。随着迭代次数的增加，测试集越来越丰富，模型的鲁棒性得到了显著增强。通过这种迭代的方式不断优化测试活动的价值输出，测试人员不断在每次测试迭代中积累经验，优化测试路径和策略，寻找到可能导致软件失败的新场景和边缘案例。

综上所述，探索性测试以其独特的理念和方法，为企业应对复杂软件项目的测试挑战提供了有力武器，成为现代软件测试不可或缺的一部分。本文后续章节中，将进一步探讨探索性测试的实施方法、应用场景以及与其它测试方法的结合，旨在帮助读者深入理解和掌握探索性测试，提升软件测试的效果与效率。

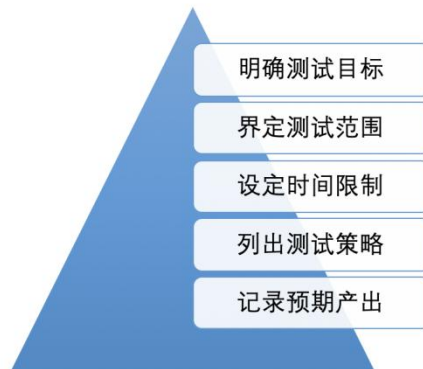
## 二、探索性测试的实施方法与技巧

### 2.1 制定测试章程 (Charter)

测试章程 (Charter) 是探索性测试的行动指南，为测试人员提供明确的目标、范围和方法。制定 Charter 有助于确保测试活动的聚焦和高效，具体步骤包括下图所示：



## 制定Charter



首先，明确本次探索性测试希望达成的具体目标，如验证某个新功能、检查特定性能指标、找出特定风险区域的问题等。假设一款跨平台应用程序最近添加了一个实时视频编辑功能。本次探索性测试的目标是验证该功能在不同操作系统（如 Windows、macOS、Android、iOS）和浏览器（Chrome、Firefox、Safari）上的兼容性与表现是否一致，特别是关注视频渲染速度、音频同步及特效应用等方面是否存在差异或问题。

然后界定测试范围，确定测试对象的具体范围，包括涉及的功能模块、数据类型、用户角色、系统环境等，避免测试活动过于分散。在对一个移动应用进行跨平台测试时，如果资源有限，那么假定本轮探索性测试仅覆盖 Android 操作系统，版本范围为 Android 8.0 到 Android 12。测试将针对不同品牌和型号的设备，但排除了 iOS 和其他旧版 Android 系统。这样的限定确保了测试的深度，避免因环境过多而导致测试覆盖面不够深入。

接着是设定时间限制，为本次探索性测试设定合理的时间预算，以保持测试的紧迫感和专注度。我们时常会在一个敏捷开发团队中，为新功能模块安排了一次为期半天（4 小时）的探索性测试冲刺。在这段时间内，测试团队集中精力对即将发布的用户故事进行全面的探索，包括功能验证、界面测试、基本的兼容性检查等。时间的紧迫促使团队成员保持高度专注，快速识别并报告问题。

还需要列出测试策略，列出可能采用的测试方法、工具、数据源等，为测试人员提供探索的线索和思路。例如要设计一个 App 应用性能的测试策略：

**测试方法：**进行压力测试和稳定性测试，模拟大量用户同时操作应用的不同功能；采用探索性负载测试，逐步增加并发用户数和操作复杂度，直至应用崩溃或响应时间不可接受。



**工具:** 使用 JMeter 或 LoadRunner 进行性能测试脚本编写和执行; 利用 Android Studio 的 Profiler 或 Xcode 的 Instruments 监控应用资源消耗, 包括 CPU、内存、网络和电池使用情况。

**数据源:** 基于生产环境的日志分析, 选取高流量时间段的真实用户行为数据作为测试依据; 创建虚拟用户场景, 模拟不同网络条件 (如 2G、3G、4G、WiFi) 下的应用表现。

最后是记录预期产出, 明确本次探索性测试期望的产出, 如发现的问题数量、撰写的问题报告、提出的改进建议等, 便于后期评估测试效果。假定一个新提交测试项目, 预期发现并记录至少 10 个高优先级的系统缺陷, 其中包括至少 2 个安全性问题、3 个性能瓶颈, 以及 5 个影响用户体验的功能性错误。所有缺陷需按照严重程度和优先级进行分类, 并在测试管理工具 (如 Jira) 中详细记录, 包括重现步骤、影响范围和建议解决方案。

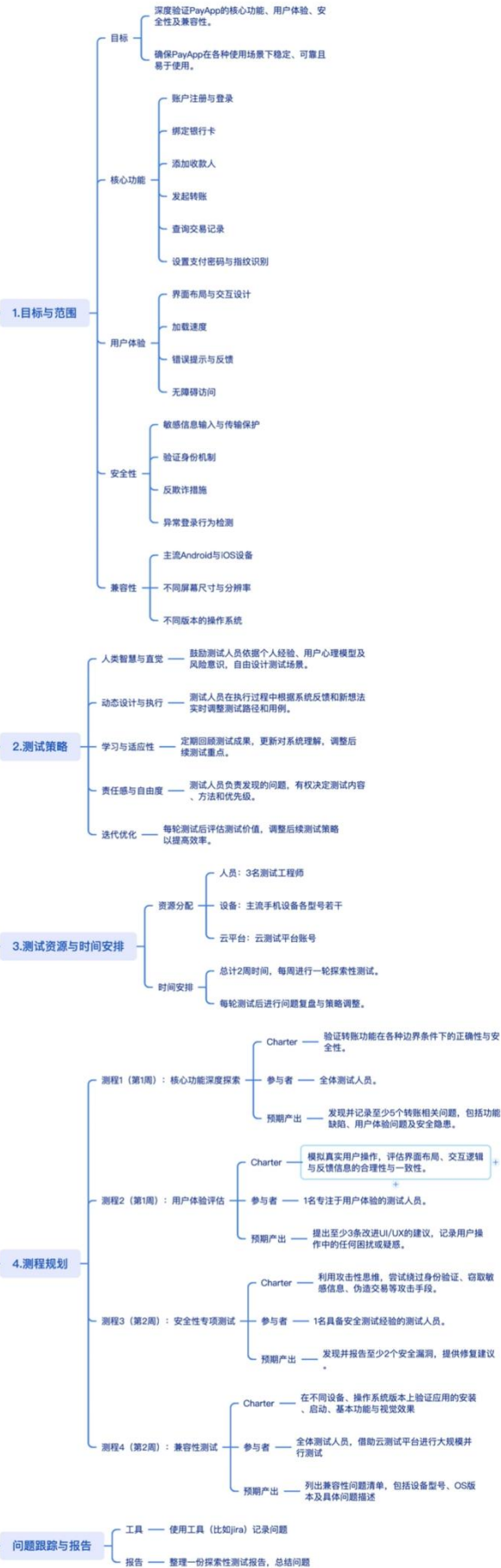
## 2.2 结合思维导图进行测试设计

思维导图是一种可视化思维工具, 可以帮助测试人员梳理复杂系统的关系, 激发创新思维, 为探索性测试设计提供支持。具体应用包括: 绘制系统架构图, 梳理业务流程, 构建测试思维树, 以问题或风险点为核心, 通过思维导图展开相关联的功能、数据、边界条件等, 形成多层次、全方位的测试思维树, 指导测试设计。

为了便于读者理解, 下面附录一个详细的支付类的 PayApp 的 Charter 范例的思维导图, App 版本是 V1.0, 可以作为参考:



探索性测试计划：  
PayAppv1.0



## 2.3 使用疑问驱动测试法 (Question-driven Testing)

疑问驱动测试法以提出和解答关于软件系统的问题为主线，引导测试人员进行深入探索。具体实施步骤如下：

**2.3.1 提出核心问题：**针对测试目标或风险点，提出一到两个核心问题，如“该功能在高并发下是否稳定？”、“用户数据在不同设备间能否无缝同步？”、“商城订单是否支持积分+优惠券+信用卡的混合支付模式？”等等。

**2.3.2 分解子问题：**围绕核心问题，进一步细化出一系列子问题，涵盖功能细节、边界条件、异常处理、性能指标等方面。比如针对 2.3.1 提出的核心问题 3：商城订单是否支持积分+优惠券+信用卡的混合支付模式？可以细化出以下子问题：

子问题 1：当用户选择同时使用积分和优惠券时，系统是否能正确计算折后金额？

子问题 2：支持的信用卡种类是否全面，包括国际信用卡和本地银行卡？

子问题 3：在混合支付过程中，若积分不足或优惠券过期，系统是否能给出明确提示并允许用户调整支付方式？

子问题 4：当支付过程中发生网络中断，重新连接后，是否能继续完成支付且保证支付安全？

子问题 5：对于大额订单，混合支付时系统是否能正确处理银行或支付平台的交易限额限制？

**2.3.3 设计测试用例：**根据子问题，设计对应的测试用例，确保覆盖问题的所有方面，形成完整的测试逻辑。举例仍然结合上一小节 2.3.2 的子问题，可以设计测试用例如下：

### 混合支付计算验证

**步骤：**创建一个订单，使用积分抵扣部分金额，再应用一张优惠券，剩余用信用卡支付。

**预期结果：**订单总额计算正确，系统展示的计算金额无误。

### 信用卡兼容性测试

**步骤：**使用不同种类的信用卡（含国际卡）完成支付。



**预期结果：**所有支持的信用卡类型均能顺利完成支付。

### 支付失败处理

**步骤：**在支付过程中模拟网络中断，恢复后继续支付。

**预期结果：**系统保留支付状态，允许用户继续且支付安全完成。

### 限额测试

**步骤：**尝试使用大额订单支付，超过单一支付方式的限额。

**预期结果：**系统提示清晰，允许用户调整支付方式或分批支付。

**2.3.4 执行并记录答案：**执行测试用例，记录测试结果和观察到的现象，形成对问题答案的初步认识。

**2.3.5 反思与调整：**根据测试结果，反思问题设定的合理性，调整后续的测试方向和策略。

## 2.4 应用平行探索 (Parallel Exploratory Testing)

平行探索是指多名测试人员同时进行探索性测试，通过共享发现、交叉验证，提高测试效率和问题发现率。实施要点包括：组建探索小组，划分探索任务，

定期同步信息，整合测试结果。

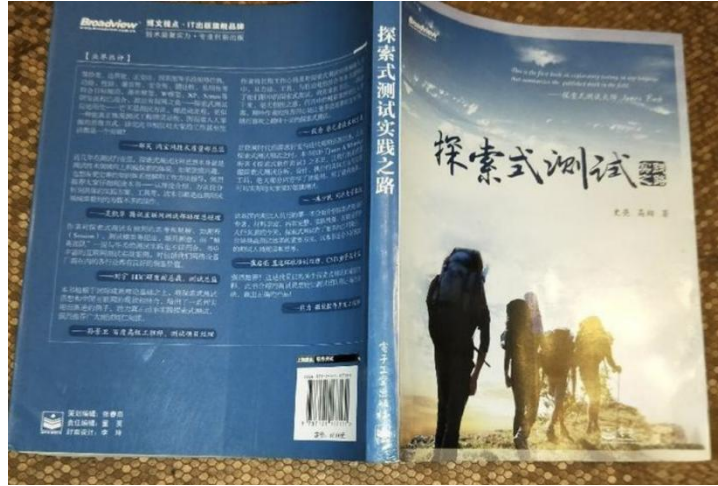
## 2.5 利用测程的测试管理 (Session-based Test Management, SBTM)

测程管理是一种管理探索性测试过程的方法，通过记录测试测程的详细信息，帮助测试人员回顾、分析和改进测试活动。

（注：在《探索式测试实践之路》这部权威著作中，史亮与高翔两位专家携手合作，对 SBTM 这一专业术语给出了严谨且贴切的中文诠释，将其译作“基于测程的测试管理”。



这一译名精准传达了 SBTM 的核心理念与运作机制，因此在本文中涉及部分的内容，笔者也参考了两位专家的诠释）SBTM 主要包括以下要素：测试测程，测试笔记，测程回顾，测程评估。



注：在测试过程中，测试人员记录下测试步骤、观察到的现象、发现的问题、思考的过程等详细信息，形成测试笔记。例如下图是一个 Session Sheet 的例子：（注：图源自期刊《Software Quality Journal（软件测试与质量工程）》杂志）

### Example Session Sheet

---

**CHARTER**

Analyze MapMaker's View menu functionality and report on areas of potential risk.

**#AREAS**  
OS | Windows 2000  
Menu | View  
Strategy | Function Testing  
Strategy | Functional Analysis

---

**START**

5/30/00 03:20 pm

---

**TESTER**

Jonathan Bach

---

**TASK BREAKDOWN**

**#DURATION**  
short

**#TEST DESIGN AND EXECUTION**  
65

**#BUG INVESTIGATION AND REPORTING**  
25

**#SESSION SETUP**  
20

**#CHARTER VS. OPPORTUNITY**  
100/0

---

**DATA FILES**

#N/A

**TEST NOTES**

I touched each of the menu items below, but focused mostly on zooming behavior with various combinations of map elements displayed.

**View:**

- Welcome Screen
- Navigator
- Locator Map
- Legend
- Map Elements
  - Highway Levels
  - Street Levels
- Airport Diagrams
- Zoom In
- Zoom Out
- Zoom Level  
(Levels 1-14)
- Previous View

**Risks:**

- Incorrect display of a map element.
- Incorrect display due to interrupted repaint.
- CD may be unreadable.
- Old version of CD might accidentally be used.
- Some function of the product may not work at a certain zoom level.

---

**BUGS**

**#BUG 1321**  
Zooming in makes you put in the CD 2 when you get to a certain level of granularity (the street names level) — even if CD 2 is already in the drive.

**#BUG 1331**  
Zooming in quickly results in street names not being rendered.

**#BUG <not\_entered>**  
Instability with slow CD speed or low video RAM. Still investigating.

---

**ISSUES**

**#ISSUE 1**  
How do I know what details should show up at what zoom levels?





通过以上章节的介绍，我们可以看到探索性测试的实施方法与技巧丰富多样，涵盖了从测试计划制定、测试设计、测试执行到测试结果管理的全过程，旨在帮助测试人员更高效、更深入地进行探索性测试，提升测试效果。

### 三、探索性测试的最佳实践与案例分享

#### 3.1 某知名企业的探索性测试成功案例分析

以某 XXX 科技公司为例，该企业在其最新发布的人工智能物联网平台项目中成功应用了探索性测试，显著提升了产品质量和用户满意度。具体实践包括：

**3.1.1 早期介入与跨职能协作：**测试团队在需求讨论初期，与产品团队一起模拟智能物联网平台的典型业务流程，如设备注册、数据上报、远程控制等，通过角色扮演的方式理解业务场景，识别潜在的测试点和风险区域。这种早期介入有助于测试人员对业务有深刻理解，为后续测试打下基础。

**3.1.2 灵活测试框架设计：**建立了一套适应探索性测试的灵活测试框架，包含核心功能测试、边界条件测试、异常场景测试等多个维度，确保测试覆盖全面且具有针对性。框架实例：采用模块化设计，使得新加入的设备类型或服务模块能快速对接到测试框架中，无需大规模重构。同时，利用容器化技术（如 Docker）快速部署和管理测试环境，便于在不同环境（开发、测试、生产）之间迁移和复制测试配置。

**3.1.3 定期探索性测试工作冲刺：**设立每两周一次的“探索周”，在这期间，测试团队集中精力进行非计划的、目标导向的探索性测试。例如，在最近的一次探索周中，团队重点测试了平台在极端条件下（如大量设备同时请求、异常数据注入）的表现，发现了几个之前未预料到的性能瓶颈并及时修复。

**3.1.4 实时反馈与快速修复：**采用敏捷开发模式，测试人员在探索过程中发现问题立即反馈给开发团队，实现问题的快速定位与修复。这种高效的反馈机制有效缩短了问题解决周期，保证了产品迭代速度。例如，测试人员在探索性测试中发现一个影响用户体验的延迟问题后，通过系统直接创建了工单，并附上了详细的复现步骤，开发团队在收到通知后两小时内即完成了修复和验证。

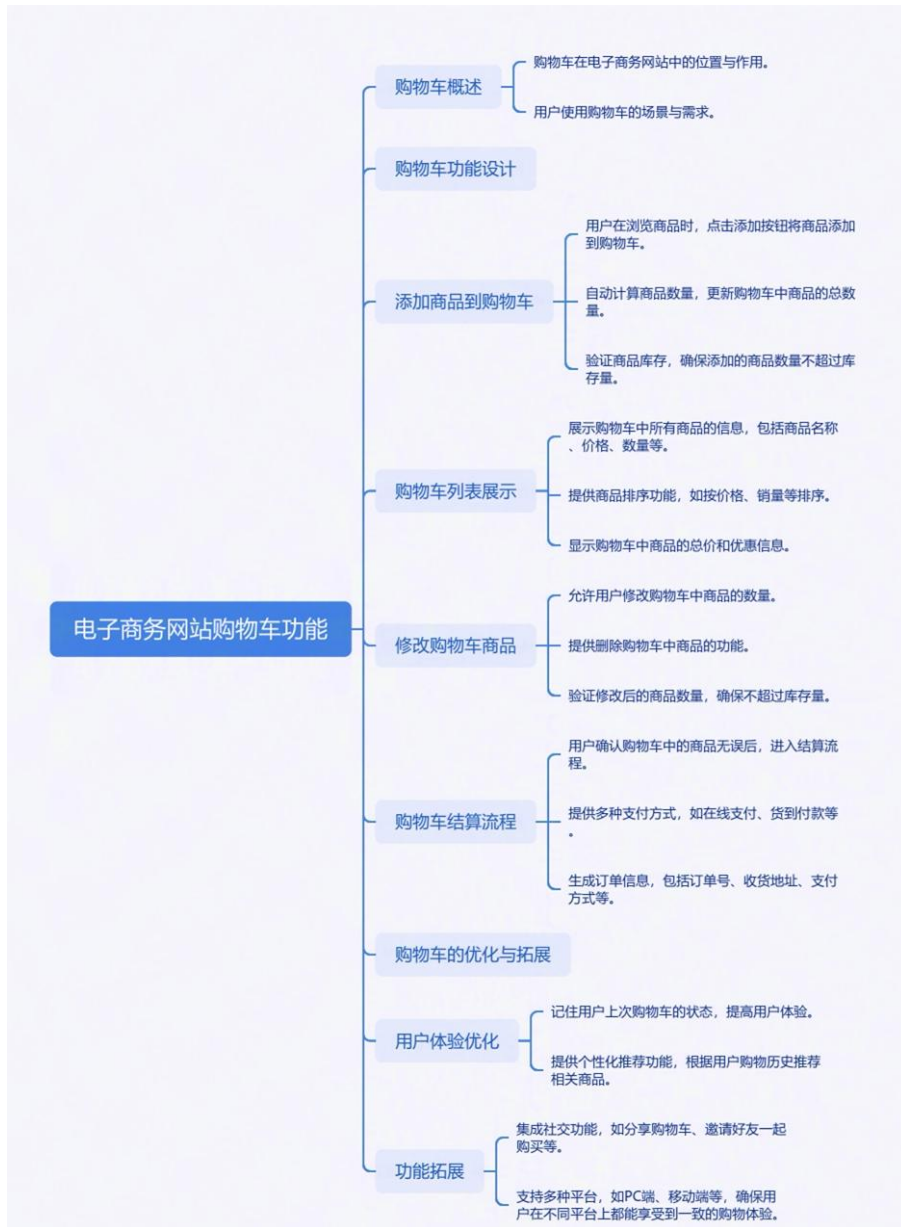
**3.1.5 量化评估与持续改进：**引入了一系列量化指标来评估探索性测试的效果，包括缺陷发现率、修复时间、用户反馈改善指数等。每季度进行一次全面的测试效果评估，



根据数据反馈调整测试策略。例如，通过分析发现探索性测试对提高系统稳定性的贡献度达到了 30%，于是决定增加探索性测试的资源投入，并针对用户反馈频繁的功能模块设计了更深入的测试计划。

案例成果：XXX 科技公司的智能物联网平台在发布后，用户反馈的故障率较以往同类项目下降了 35%，用户满意度评分提高了 20 个百分点。这一成功案例充分证明了探索性测试在提升产品质量、优化用户体验方面的显著作用。

### 3.2 具体场景下探索性测试策略与效果展示



如上图所示场景：电子商务网站购物车功能



探索性测试的策略设计可以按照下面几个方面实施：

- 深度遍历策略：测试人员模拟各种添加商品、修改数量、删除商品的操作，尤其关注商品类型、数量、优惠券叠加等边界条件。下面是几个具体的测试场景：卡券类商品和食品类商品叠加是否能触发满减优惠？优惠券勾选后，再删减商品，优惠券是否仍然能够使用？商品数量能否正确的增加或减少？（曾经有一年京东 APP 的部分用户遇到了一个异常情况，他们发现购物车中所有商品的数量无缘无故地被统一设置为了 10 件。即使用户之前添加的商品数量各不相同，这个 bug 也使得每个人的购物车内所有商品显示为同样的数量。此问题并非针对特定用户或商品，所以当时是广泛影响到了一部分用户群体。）

- 异常注入：故意触发网络延迟、库存不足、价格变动等异常情况，观察系统的响应与恢复能力。

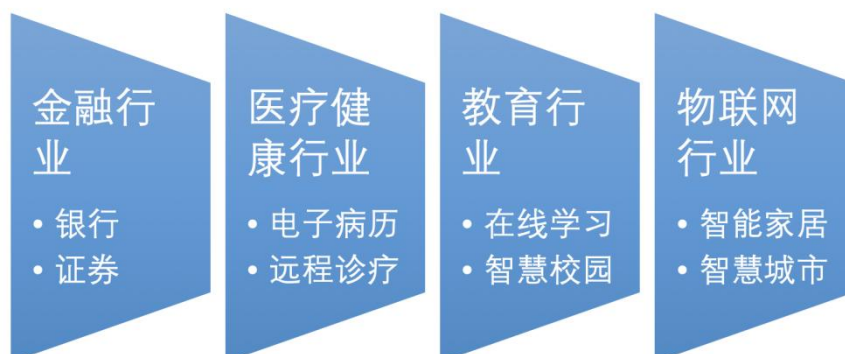
- 用户体验模拟：从用户角度出发，测试购物车在不同屏幕尺寸、浏览器版本、操作流程中的易用性。

**效果：**

- 发现并修复了多起因特定商品类型组合导致的计费错误。
- 确保了在库存波动、价格调整等情况下，购物车状态更新的准确性和及时性。
- 提升了购物车在多种设备和浏览器环境下的兼容性和用户体验，减少了用户投诉。

### 3.3 探索性测试在不同行业领域的应用实例

#### 在不同行业里的应用



**3.3.1 金融行业：**在银行交易系统、证券交易平台等项目中，探索性测试用于深度挖掘复杂的交易规则、风控策略、并发处理等场景下的潜在问题，保障系统的稳定性和安全性。以下是几个具体例子，展示了探索性测试在金融行业应用的深度和广度：

#### 案例 1 合规性与法规遵从性验证

场景：金融交易需遵循严格的法律法规，如反洗钱（AML）、客户身份验证（KYC）规定。

探索性测试：设计测试案例，检查系统对用户身份信息验证的严格性，如是否能准确识别虚假信息、拒绝不符合规定的用户注册。模拟可疑交易行为，测试系统是否能触发报警机制并按合规要求处理。确保系统设计和操作流程符合当地及国际金融监管要求。

#### 案例 2 复杂交易规则的探索

场景：在银行系统中，存在复杂的贷款利率计算逻辑，包括固定利率、浮动利率、提前还款罚款计算等。

探索性测试：设计测试用例模拟不同贷款额度、期限、还款方式下的利率计算，特别关注边界条件，如最小还款额、最长还款期限等，验证计算结果的准确性。同时，探索极端情况，如用户频繁变更还款计划对系统的影响，确保计算逻辑无误且系统响应合理。

**3.3.2 医疗健康行业：**在电子病历系统、远程诊疗平台等项目中，探索性测试关注数据完整性、患者隐私保护、紧急情况处理等关键环节，确保医疗服务的准确性和合规性。以下是具体案例：

#### 案例 1 数据完整性与准确性的探索

场景：在电子病历系统中，医生录入和修改患者的诊断、治疗方案等信息。

探索性测试：模拟医生在不同网络条件下（如网络延迟、断网后重连）操作病历记录，验证数据是否能完整、准确地保存和同步。设计测试用例，如同时编辑同一条病历记录，测试并发处理逻辑，确保数据不丢失或冲突。此外，检查系统对数据输入的校验，比如剂量单位、日期格式的正确性，防止数据录入错误。



## 案例 2 紧急情况处理能力的深入测试

场景：在远程诊疗平台，医生需快速响应患者的紧急咨询或病情变化。

探索性测试：模拟紧急呼叫场景，测试平台的响应速度、通知机制（如短信、电话通知医生）是否及时且准确。检查系统在高负载（如疫情期间大量患者同时求助）下的稳定性，确保服务不中断。设计用例测试系统对患者紧急状况的自动识别和优先级处理，比如自动标记并优先推送危重病例给医生。

**3.3.3 教育行业：**在在线学习平台、智慧校园系统等项目，探索性测试着重测试课程资源加载、互动功能、个性化推荐等模块，提升教学质量和用户体验。比如以下几个案例：

### 案例 1 个性化推荐算法的深入测试

场景：平台根据用户学习行为和兴趣提供个性化课程推荐。

探索性测试：创建多组模拟用户资料，每组代表不同学习风格、兴趣和学习进度，测试推荐算法的精准度。检查系统在用户行为改变后的推荐更新速度，比如完成一门编程课后是否能迅速推荐进阶课程。探索算法的公平性和多样性，确保推荐内容不会过度集中，同时考虑用户的长期学习路径规划。

### 案例 2 兼容性和可访问性测试

场景：用户使用不同设备和操作系统访问平台。

探索性测试：在各种设备（PC、平板、手机）、操作系统（Windows、iOS、Android）、浏览器上测试平台的兼容性，确保界面布局合理、功能正常。进行可访问性测试，确保视障用户通过屏幕阅读器等辅助工具也能顺利使用平台，包括测试音频描述、字幕功能。

### 案例 3 课程资源加载与播放体验测试

场景：在线学习平台提供视频教程、PPT、文档等多种学习资源。

探索性测试：模拟不同网络环境（如 3G、4G、Wi-Fi、弱网）下，测试视频加载速度和播放流畅性，确保无卡顿或缓冲时间过长。检查文档和 PPT 加载速度，验证资源格式兼容性，如测试不同版本的 PDF、PPT 是否都能正常打开。测试资源切换时的响应速度，比如从视频跳转到相关章节的文档是否无缝衔接。



**3.3.4 物联网行业：**在智能家居、智慧城市等项目中，探索性测试针对设备互联互通、数据同步、异常处理等场景进行深度探索，确保物联网系统的可靠性和响应速度。

#### 案例 1 异常处理能力的深入测试

场景：智能安防系统在遭遇入侵或异常行为时的响应。

探索性测试：设计模拟入侵场景，如门窗传感器触发、运动检测异常，测试系统是否能及时报警并准确记录事件。测试报警系统的多渠道通知机制（如 APP、短信、电话）的响应速度和准确性。验证系统在处理大量异常报警时的稳定性，避免漏报或误报。探索设备自我诊断能力，如智能摄像头镜头遮挡、传感器故障时的自我检测和通知机制。

#### 案例 2 设备互联互通测试

场景：在智能家居项目中，各类智能设备如智能灯泡、智能门锁、温控系统需协同工作。

探索性测试：模拟不同品牌、不同协议（如 Zigbee、Wi-Fi、蓝牙、Z-Wave）的设备在同一个智能家居平台下的集成，测试设备的发现、配对、连接稳定性。在不同网络环境下（如网络切换、弱信号）测试设备的自动重连能力，以及设备间通信的延迟性。设计异常场景，如某一设备断电或故障，测试系统如何处理并通知用户，以及对其他设备的影响。

以上实例表明，无论是在传统行业还是新兴领域，探索性测试都能凭借其灵活性、深度挖掘能力，适应各行业的特性和需求，为产品质量保驾护航。

## 四、探索性测试的未来趋势与挑战？

### 技术进步对探索性测试的影响（如 AI、大数据等）

人工智能（AI）赋能：AI 技术，特别是机器学习和自然语言处理，可以辅助探索性测试。例如，AI 可以通过学习历史测试数据和用户行为模式，生成更有针对性的测试策略和用例；智能聊天机器人可以模拟用户交互，进行大规模的探索性对话测试。当然为了达到良好效果，以上提到的 AI 或者智能机器人都需要提前进行大量训练。

大数据驱动：大数据分析能够帮助测试人员理解用户行为模式、系统性能瓶颈、故障模式等，为探索性测试提供数据支持。以下是一个具体应用实例：



## 用户行为模式分析

场景：电商平台在节假日前准备促销活动。

实践：通过分析历史大数据，识别用户在类似促销活动期间的行为模式，比如浏览习惯、购物车使用频率、支付转化率等。发现特定时间段（如晚上黄金时段）的用户活跃度激增，以及某些商品类别（如电子产品、时尚服饰）的搜索量暴增。基于此信息，测试团队重点测试这些高流量时段和热门商品的页面加载速度、库存更新准确性、支付流程的稳定性，确保系统能承受高并发访问。

自动化与智能化结合：AI 和自动化工具的结合，可以实现更高级别的自动化探索性测试。例如，智能测试代理能够根据实时测试结果动态调整测试路径，自动探索系统边界和异常情况，提高测试效率和覆盖率。算法的话，推荐遗传算法蜂群算法，完成测试用例自动生成部分工作。

## 新兴测试模式对探索性测试的启示（如持续测试、云端测试等）

持续测试：随着 DevOps 和 CI/CD 的普及，持续测试成为主流。探索性测试可以嵌入到持续测试管道中，作为自动化测试的有效补充，尤其是在新功能上线、快速迭代等场景下，提供即时的、深度的验证。以下是将探索性测试融入持续测试管道的几个具体做法和示例：

### 环境即服务（EaaS）支持探索性测试

实践：在持续测试中，快速提供与生产环境尽可能一致的测试环境，让探索性测试人员可以在接近真实的条件下进行测试，发现生产中可能出现的问题。

例子：利用云基础设施，实现一键部署与生产环境配置相同的测试环境。每当有新版本即将发布，测试人员可以即时获得这样一个环境，进行深入的手动探索，包括极端条件下的系统响应、用户旅程的完整性验证等，确保在真实部署前问题被充分暴露。

### 集成探索性测试于自动化测试流

实践：在 CI/CD 流水线中，自动化测试（如单元测试、API 测试）之后，安排探索性测试环节。当代码合并到主分支并触发自动化测试成功后，随即启动探索性测试任务。这可以是手动执行或借助智能化的探索性测试工具（如使用机器学习辅助生成测试场景）。

例子：开发团队在即将发布新功能的迭代末期，将功能分支合并至主分支，自动触



发了包括单元测试、集成测试的 CI 流程。一旦这些自动化测试通过，系统自动通知测试团队开始探索性测试，专注于新功能的边界条件、用户体验和与其他功能的交互，确保没有遗漏的逻辑错误或交互问题。

**云端测试：**云环境为探索性测试提供了无限的资源和灵活的测试环境。例如，测试人员可以轻松模拟全球用户、大规模并发、复杂网络条件等场景，进行大规模、高仿真的探索性测试。

**Shift-Left（左移）测试：**探索性测试的理念和方法可以进一步向开发早期延伸，例如需求分析、设计阶段就引入探索性思维，提前发现潜在问题，降低后期修复成本。

### 探索性测试面临的挑战与应对策略

**技能与培训需求：**探索性测试对测试人员的技能要求较高，需要具备深厚的技术知识、良好的问题发现能力、较强的沟通协作能力等。应对策略是加强内部培训、引进外部专家、建立学习型组织，提升团队整体能力。

**度量与管理难题：**探索性测试的结果往往难以量化，难以用传统的测试覆盖率、缺陷率等指标衡量。应对策略是发展适应探索性测试的度量模型，如基于测程的测试管理（SBTM）、基于风险的测试等，同时强调定性反馈和业务价值。

**工具支持不足：**现有的测试工具对探索性测试的支持不够完善，缺乏专门针对探索性测试的设计、执行、记录、分析等功能。应对策略是推动工具供应商改进产品，或研发定制化工具，同时充分利用现有工具的灵活性，如使用通用笔记工具记录探索过程，使用自动化工具辅助探索。

总结而言，探索性测试将在技术进步和新兴测试模式的推动下，进一步融入软件开发生命周期，发挥更大价值。同时，面对挑战，测试团队需不断提升自身能力，创新管理方法，寻求合适的工具支持，以适应未来的测试需求。

## 五、结论

探索性测试作为一种富有创新性和实效性的测试方法，其价值与独特优势体现在以下几个方面：深度问题发现、快速响应变化、提升测试效率、强化团队协作与沟通、促进质量文化。笔者发现，倡导探索性测试的企业，通常更注重持续改进与创新，鼓励员





工质疑现状、发现问题，有助于营造积极的质量文化，提升整个组织的产品质量意识。

面对探索性测试的诸多优势，企业与测试人员应积极采纳并实践这一方法，企业高层应认识到探索性测试的价值，将其纳入质量管理策略，提供必要的资源支持，并通过培训、奖励机制等手段鼓励团队采用。定期举办内部研讨会、虚拟团队或邀请行业专家进行培训，提升测试人员的探索性测试技能。鼓励团队内部分享实践经验，形成知识沉淀与传承。将探索性测试无缝融入现有的测试生命周期，如结合持续集成、敏捷开发等实践，确保其在关键决策点发挥作用。建立适合探索性测试的度量体系，比如，以问题发现率、问题发现数量、问题解决速度、用户反馈改善（或者用户客诉率）等可量化指标来评价其效果，同时建设重视定性反馈，持续优化测试策略。

## 附录

### A. 探索性测试常用资源与工具推荐

#### 工具：

- 1、Session-based Test Management (SBTM)?软件，如 Rapid Reporter、TestPad，用于记录和管理探索性测试测程。
- 2、Bug 追踪系统，如 JIRA、Bugzilla，用于报告和跟踪发现的问题。
- 3、自动化辅助工具，如 Selenium、Appium、Jmeter，用于执行重复性任务或特定场景的自动化测试，与探索性测试相结合。
- 4、探索性测试思维导图工具，如 MindMeister、XMind，用于可视化地规划测试思路和记录发现。

#### 在线社区与论坛：

- 1、Ministry of Testing (<https://ministryoftesting.com/>): 提供丰富的探索性测试文章、教程、活动信息及测试人员交流平台。
- 2、Software Testing Club (<https://www.softwaretestingclub.com/>): 专业的测试社区，涵盖各类测试方法论讨论，包括探索性测试。
- 3、51testing 软件测试网 (<http://www.51testing.com/>): 人气最旺的软件测试技术门户,提供软件测试社区交流,软件测试博客



专业书籍:

Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing?by Elisabeth Hendrickson

The Art of Software Testing?by Glenford J. Myers, Tom Badgett, and Rex Black

Agile Testing: A Practical Guide for Testers and Agile Teams?by Lisa Crispin and Janet Gregory

#### B. 探索性测试相关研究与文献资料列举

Exploratory Software Testing?by James A. Whittaker

Ad hoc software testing: a guide to testing your software project without a test plan?by John Koomen

Exploratory testing in agile projects: an empirical study?by Hakan Erdogmus and Cem Kaner

An exploratory study of exploratory testing?by Cem Kaner, James Bach, and Bret Pettichord

A survey of the state of the art in exploratory testing research?by Yijun Yu, Lionel C. Briand, and Jeff Kramer

[Bach2000] Jonathan Bach. Session Based Test Management.<http://www.satisfice.com/articles/sbtm.pdf>, 2000.

[Bach2004] Jonathan Bach. Testing in Session - How to measure exploratory testing.

张唯.基于人工蜂群优化算法的测试用例生成及优先级排序. (Doctoral dissertation, 浙江理工大学).



# 测试江湖：漫谈职场中的“卷”和“996”

◆作者：风落几番

“996”这个梗，可谓是程序猿群体的心酸与无奈的代名词。咱既然打算敞开心扉，聊聊大实话，那就不能回避这个热议话题。毕竟，“996”早已渗透进程序员的世界，成为了日常生活的一部分。

所谓“996”，顾名思义，就是每天早晨9点打卡上班，夜晚9点打卡下班，每周工作六天，且多数情况下无偿加班。这个词儿真正火起来，还是因为一批程序员在网上发起的“996.ICU”运动，一时间舆论沸腾，赞同者有之，反对者亦不在少数。

有人振臂高呼“996是奋斗的象征”，“趁年轻不多拼何时拼”，“996纯属自愿，不爱干可以走人”。但反对者则坚定表示：“996”严重违反劳动法，把人变成了工作机器，而且这跟工作效率、成果、奋斗精神八竿子打不着。

咱不打算在辩论赛中挑边站，只想掏心窝子说几句。在互联网江湖混迹多年，身边不乏一线大厂的同学、朋友，可以说，“996”对我们来说早就习以为常，甚至有时能在“某宝大楼”、“某为阵地”、“某条堡垒”彻夜灯火通明的盛景中，找到一丝丝“幸福感”。毕竟，在众多一线互联网公司，能保证“996”，已经是种奢侈。

虽然很少有公司明目张胆地说“强制996”，大多挂着“鼓励不强迫”的羊头，但实际情况往往是，你可能有个热衷加班的领导，或者遇到一个自己不怎么加班却喜欢看你看加班的主儿。如果你胆敢准点下班，那简直是撞枪口上了。

在互联网这个光鲜亮丽却又竞争激烈的行业中，驱使着广大程序员乃至各类职员奋不顾身投身于加班浪潮的，并非仅仅是挂在嘴边的“996”口号，也不是靠着什么激情澎湃的奋斗精神或被渲染得热血沸腾的“狼性文化”。真正让他们夜以继日奋战在电脑屏幕前的，其实是那套冷冰冰、却威力无比的绩效考核机制。其中最具代表性的一种玩法，



就是被各大公司广泛采用的“361”评级制度。

这个“361”就像一个无情的天平，每到季度末尾，它就会毫不留情地对全体员工进行重新洗牌。根据这套规则，每个部门都得按照规定的比例划分出人群层次：30%的员工被戴上“优秀”的桂冠，他们将享受到荣誉与物质双重丰收的喜悦；而剩下的60%则被划入“普通”阵营，他们的表现尚可，但也并未脱颖而出；至于最后那可怜巴巴的10%，他们则会成为“垫底”的“拖油瓶”，不幸被贴上“不合格”的标签。

各部门在这场评级游戏中，不得不硬着头皮，严格按照比例挑选出那几位“倒霉蛋”。对于这些“不合格”的同志来说，迎接他们的不仅仅是奖金大幅度缩水的打击，更可怕的是晋升之路就此戛然而止，加薪和转岗的机会也瞬间化为泡影。这就是著名的“末位淘汰”法则，它如同一把悬在头顶的达摩克利斯之剑，时刻提醒着每位员工：要么勇攀高峰，要么跌入谷底。

而这套制度无形中给员工施加了巨大的压力，仿佛每个人都在一条看不见的赛道上奔跑，稍有懈怠就可能被甩到队伍的末端。于是乎，加班便成为一种常态，大家都在尽力冲刺，希望能够挤进那30%的优秀行列，远离那可怕的10%红线。这样一来，“996”似乎就自然而然地演化成了互联网行业的生存法则，成为大家努力奋斗，渴望摆脱“拖油瓶”命运的无奈之举。

每当“996”引发热议，我和朋友们也会忍不住吐槽、愤慨一番，但更多的还是无奈。无奈在于我们舍不得放弃“996”换来的高薪，而去“955”的舒适圈；无奈在于肩负房贷、车贷、养家糊口的压力，无法让自己悠闲度日；无奈在于整个软件行业普遍存在“996”，跳槽也只是换个地方继续“996”；更无奈在于，我们深知自己无力改变这个世界。

再换个角度想想，“996”恐怕也不仅仅是疲惫、不仅仅是消耗身体，近年来互联网屡见不鲜的“裁员潮”又跟“996”没有一点关系么？

在互联网职场中，“鲑鱼效应”这个比喻生动而形象地刻画了职场竞争环境的动态变化。这个效应来源于一个渔业现象：在运输鲜活鱼类时，如果放入一条活泼好动的鲑鱼，其他安静的鱼为了躲避鲑鱼的追逐和攻击，会被迫活跃起来，从而降低了死亡率。在职场语境中，这条“鲑鱼”可以代表一种激烈的竞争态势，比如某位员工突然高强度工作、频繁加班，或者有着超越常人的业绩表现，这就像在平静的池塘中投下了一条鲑鱼。

当一个人在团队中展现出超出常规的“卷”态，即表现出极度的工作热情、高强度



的工作投入和优异的业绩成果时，很容易引起其他同事的紧张感和危机意识。于是乎，周围的同事们为了不被比下去，纷纷效仿这种高强度工作模式，形成了全员“卷起来”的趋势。

然而，这种“卷”文化的蔓延可能会带来一系列连锁反应。一方面，短期内可能会促使团队整体效能提升，但另一方面，长期高强度工作会导致员工身心疲惫，创造力下降，甚至出现人才流失。更严重的是，企业在追求短期业绩提升的同时，可能会对人力成本进行重新考量，进而采取降薪、裁员等措施来调整人力资源结构，以适应这种异常激烈的竞争状态。

所以即便是想卷又身体好能卷的同学，对你自己而言，卷也未必是件好事。

言归正传，说完了现状，我们得聊聊在无法改变的“996”大潮下，还可以思考些什么？借用辩论圈流行的一句话：“今天我们讨论‘996’，到底要讨论什么呢？”好吧，前面我们剖析了现实，那么接下来，咱们聊聊在无法改变环境的情况下，如何改变自己，做个能在丛林法则中生存的“适者”。

### 修炼成“无人可替”的大神

如果你立志逃离“996”，又非“富二代”，那么最重要的一招就是——修炼成“无人可替代”的大神。

举个栗子，身边有两个70后的老哥。一位是业内赫赫有名的低调技术大牛，公司诸多架构、方案都倚仗他点头，甚至底层框架修改都离不开他；另一位呢，技术虽也不错，但为人勤奋到爆肝，可一旦家里有点状况影响了加班，竟就被冠以“不合格”，黯然离开了奋斗多年的公司。

现实虽然残酷，但员工的核心竞争力就是你的价值。假如老板认为你的价值远超你的薪水，且别人还无法替代你，那么什么“996”、“361”，统统都跟你无关。

反之，若在老板眼里你轻易可被取代，那你即使不是最懒散的那个，也可能在一轮考核中被无情标记为“废物”。



## 变“被迫”为“主动”，顺便提升自己

如果暂时没能修炼成大神，不得不加入“996”大军，那就要努力从“被迫加班”变为“主动加班”。在工作中找到自己的兴趣点，利用加班时间去深化、拓宽自己的技能树，朝着更高的目标和更好的自己迈进。

听着挺鸡汤是吧？其实我是想说，不是每天都得“996”才能完成工作，就算在忙碌的互联网行业，一个月里也就那么几天忙到飞起。所以，与其在“996”时假装忙碌，不如找点自己喜欢的事，让加班时间变得有意义。

我有个匿名的朋友（之所以匿名，是因为我不想免费给他打广告，哈哈！）最初也是抵触加班，后来他利用加班时间提升技术能力，拓宽知识面，现在已经成功跳出“996”牢笼，摇身一变成了知名公众号作家，收入直接吊打我们这些一线搬砖工。

## 调整心态，学会娱乐

当然，并非所有人都能逃脱“996”的魔掌，这时候心态就显得尤为重要。除非你真的热爱这份工作到骨髓里，否则工作绝不是生活的全部（老板要是看到这句话，估计会提着菜刀追着我跑！）

见过身边有人因“996”倒在办公桌前，而多半并非因为身体透支，而是心理压力过大。因此，适时释放压力相当关键。

我会在固定的日子给自己放个小假，哪怕就是去海边晒晒太阳，陪媳妇追追轻松的偶像剧。不加班的时候，也会约上三五好友，喝个小酒，吐吐槽，或者培养些小爱好，比如玩个手游、打打球、骑车四处逛逛。

热爱生活的人才能更好地应对工作，保持良好的心态，在工作中发挥出色，以更阳光的态度面对“996”的挑战。毕竟，工作是为了更好的生活，而生活不止眼前的“996”，还有诗和远方的田野，以及偶尔摸鱼的快乐时光！

----- END -----

