目 录 (四十七期)

自动化回归测试执行效率探讨

◆ 作者：万晓光

摘要

本文通过分析大型系统自动化回归测试执行效率低下的问题，总结了从测试用例、测试脚本、最佳实践、持续集成等方面的解决方案。通过最后的实施结果来看，综合采用这些方案后取得了很好的效果，大幅度地提高了自动化回归测试的执行效率，使产品的质量状态得到了及时的反馈。

一、前言

在大型 Web 系统的自动化回归测试中，回归测试通常会耗费大量的时间，随后的结果分析以及对产品的质量反馈周期就很漫长。若是执行中一旦发生中断失败，可能还需要重启回归测试，那样的话，对产品质量的反馈就更慢了。笔者最近参加过一个大型的自动化测试项目，在项目前、中期我们也遇到了这样的问题，团队对自动化回归测试的执行进行了全面而仔细的分析，然后通过一些方案大幅度地提高了执行的效率。

二、项目概况

项目是基于 Selenium 的自动化测试，测试框架为 TestNG，持续集成平台为 Jenkins。所测试的系统业务规则很多、业务比较复杂，系统三大业务模块（系统配置、排班管理和打卡管理）总共有约 2000 个自动化测试用例。自动化回归测试按照每个模块单独执行。项目从开始进行到中期时，每个模块的自动化回归测试仍是在单个 Jenkins 任务中以单线程模式执行的。各模块的用例数以及在 IE11 上回归测试的时长如表 1 所示。

表 1、测试用例数及在 IE11 上的执行时长

业务领域	测试用例数量	回归测试时长（小时）
系统配置	255	9.5

业务领域	测试用例数量	回归测试时长（小时）
排班管理	240	9.6
打卡管理	260	10.2

从表 1 可以看出，在每个模块只有 240 到 260 个测试用例的情况下，单个模块在单线程的执行模式下已经需要 10 个小时了。如果单个模块的测试用例增加到 700 个以上，在当前模式下的执行时间将以天计，这样的执行效率是不能被接受的。

三、耗时分析

为了解决执行效率的问题，团队分析了各业务模块的回归测试流程和测试结果。从不同的层面，团队发现了一些导致执行耗时较长的因素。限于篇幅，本文仅以“打卡管理”这个业务模块作为例子来阐述发现的问题。

3.1 持续集成层面

3.1.1 并发

在持续集成平台 Jenkins 上配置的回归测试任务中，没有嵌套的层级任务结构配置，没有充分利用 Jenkins 插件的优势去并行地执行测试任务。

另外，也没有利用 Selenium Grid 并行地执行测试用例。当时的配置是，一个 Jenkins 任务对应一台测试客户端，单线程地执行模块内的所有测试用例。

3.1.2 浏览器窗口+登录

Jenkins 的任务以及自动化测试脚本采用策略是每个测试用例的执行都需要重新启动一个新的浏览器窗口、重新登录以排除其他测试用例的干扰。在结束时，还要退出登录并关闭浏览器窗口。

在分析测试执行日志后发现，每次重新打开浏览器和进行登录都会消耗比较多的时间。图 1 显示了 Testng 从启动一个测试用例到登录完成的日志信息。

```
2017-07-17 20:50:49 INFO  BaseTest:142 - Start test case [test_ALM81043_Open                                    Button]
2017-07-17 20:50:49 INFO  BaseTest:151 - Execution context parameter set ExecutionContext [tenant=                , frontEndServer=http://objectiva-
b.int.dev.                /, backEndServer=http://objectiva-b-back.int.dev.                /, openAmServer=http://keng01-dev01-ath21-
oam21.int.dev.mykronos.com:8080/]
2017-07-17 20:50:49 INFO                UtilsFactory:274 - Browser parameter is IE
2017-07-17 20:50:59 INFO  BasicPageSyncHelper:106 - getCurrentWindowHandle: [913a2d6b-720d-41da-8e41-af6052a9cadf]
2017-07-17 20:50:59 INFO                BasePageFactory:39 - Page initilization is OK.
2017-07-17 20:51:01 ERROR BasicPageElementHelper:179 - findElementByNotPresence: [ By.cssSelector: #          _logo ] with timeout: 5
2017-07-17 20:51:01 INFO  BaseElement:770 - isDisplayed: true [ kronosLogo : By.cssSelector: #          _logo ]
2017-07-17 20:51:01 INFO  BasicPageElementHelper:86 - findElementByVisible: [ By.cssSelector: #idToken1.form-control ] with timeout: 60
2017-07-17 20:51:01 INFO  BaseElement:1229 - waitForEnabled: [ nameTxt : By.cssSelector: #idToken1.form-control ]
2017-07-17 20:51:02 INFO  BaseElement:132 - setText:                in [ nameTxt : By.cssSelector: #idToken1.form-control ]
2017-07-17 20:51:02 INFO  BasicPageElementHelper:86 - findElementByVisible: [ By.cssSelector: #idToken2.form-control ] with timeout: 60
2017-07-17 20:51:02 INFO  BaseElement:1229 - waitForEnabled: [ passwordTxt : By.cssSelector: #idToken2.form-control ]
2017-07-17 20:51:03 INFO  BaseElement:132 - setText:                d [ passwordTxt : By.cssSelector: #idToken2.form-control ]
2017-07-17 20:51:03 INFO  BasicPageElementHelper:86 - findElementByVisible: [ By.cssSelector: .btn-primary[id*='loginButton'] ] with timeout:
60
2017-07-17 20:51:03 INFO  BaseElement:861 - executeScript: arguments[0].click() with parameter: [Ljava.lang.Object;@1f72f2e
2017-07-17 20:51:03 INFO  BaseElement:884 - clickAsJScript:  [ loginButton : By.cssSelector: .btn-primary[id*='loginButton'] ]
2017-07-17 20:51:03 INFO  BasicPageSyncHelper:106 - getCurrentWindowHandle: [913a2d6b-720d-41da-8e41-af6052a9cadf]
2017-07-17 20:51:03 INFO                BasePageFactory:39 - Page initilization is OK.
2017-07-17 20:51:12 INFO  BasicPageElementHelper:57 - findElementByClickable: [ By.cssSelector: .hamburger-menu.btn ] with timeout: 60
```

图 1 启动浏览器并进行登录的日志

从日志信息可以看出，每次启动 IE 浏览器到登录的总耗时约为 23 秒。

- 20:50:49 - Testng 开始启动测试用例

- 20:51:12 - 登录成功开始检查首页上的菜单按钮是否可点击

打卡管理模块的 260 个测试用例，如果每个测试类平均按照 5 个测试用例来算，再加上一些必需的重新登录，应该有 60%的测试用例可以重用浏览器窗口，也不需要登录，总共可以节省的时间为：260 x 60% x 23 = 3588 秒（1 小时）。

3.2 测试用例和脚本层面

经过检查测试用例和测试脚本，发现用例和脚本中有比较多不必要的验证点。这些验证点对某些测试用例是必需的，但是对于其他很多的测试用例则是多余的。这些验证点不仅会浪费执行的时间，而且也可能导致测试用例失败在不相关的验证点上。

另外，团队还发现了一些测试用例的执行耗时过长。比如，有的测试用例耗时 30 分钟左右，甚至近 1 个小时。经分析发现了如下的问题会导致单个测试用例执行时间过长：

测试用例的问题。有的测试用例要求验证大量的界面元素，或者重复验证大量的界面元素。

准备、销毁测试数据的问题。测试脚本中通过界面创建、销毁比较多的测试数据，这部分界面操作消耗了额外的时间。

测试脚本的问题。有些测试脚本本身存在一些性能问题。

3.3 浏览器层面

在不同浏览器上自动化脚本的执行效率，业界已有很多的讨论和验证。总体来说，在 Chrome 上的执行效率最高，在 IE 上的效率最差。

就本项目来说，测试脚本在各个浏览器上的执行效率也基本符合业界的统计。如表 2 所示，就“打卡管理”模块来说，在三个主要浏览器上，以单线程模式执行 260 个测试用例时，效率和通过率的差别比较大。在 Chrome 和 Firefox 上执行效率差别不大，通过率都很高，但是在 IE11 上的执行效率大幅下降，且通过率大打折扣。

表 2、不同浏览器上的执行效率对比

浏览器	执行时长（小时）	通过率
Chrome	7	90.36
Firefox	7.1	88.27
IE11	10	54.82

四、回归测试效率提升方案及实践

通过以上的分析，团队发现了在持续集成、浏览器覆盖策略、测试用例和脚本等方面可以改进的地方，团队也有针对性地提出了一系列相关的解决方案并和客户进行过充分的讨论，方案确定并应用后取得了不错的效果。

4.1 并行+串行+并行

根据系统的情况和测试脚本中对测试数据的使用，为了能高效并行地执行测试用例，团队提出了基于 Jenkins Multijob 和 Selenium Grid 的“并行+串行+并行”的回归测试执行方案，如图 2 所示。

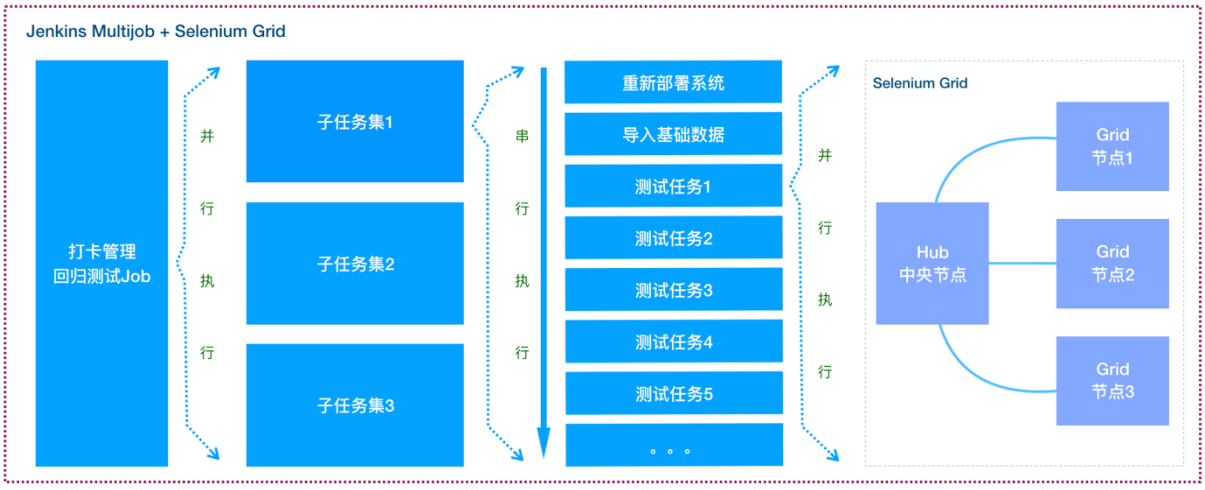


图 2 “并行+串行+并行”方案

利用 Jenkins Multijob 插件，在执行回归测试时，同时启动三个子任务集、并行执行。各子任务集所执行的环境互相独立，各子任务集的测试用例不会互相干扰。

在并行执行的三个子任务集中，每次执行都会重新部署系统，对所包含的测试任务实行串行调用。

在串行执行的测试任务中，采用 Selenium Grid 来并行执行其中的测试用例。

其中 Jenkins 的多任务配置如图 3 所示。三个并行执行的子任务集被配置在父任务的第一个“阶段任务”中。在同一个阶段，他们就会被同时执行。

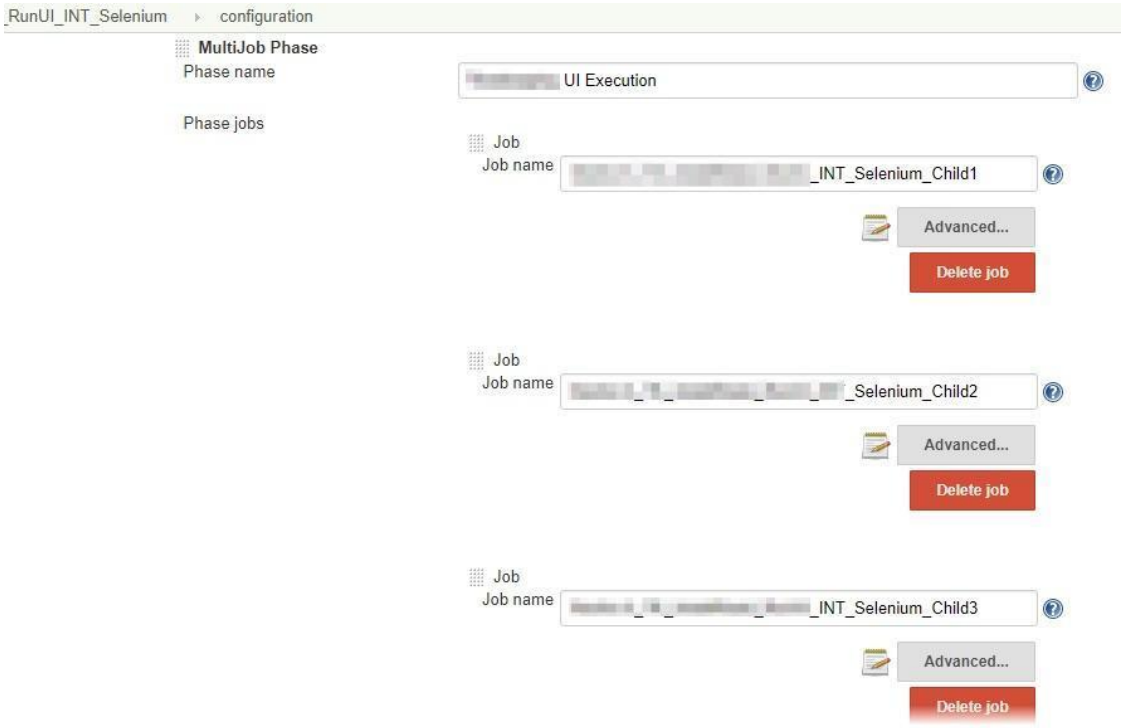


图 3 Jenkins Multijob 配置

4.2 用 Chrome 来执行

Selenium WebDriver 针对不同浏览器的性能表现不一，同样的测试用例在 IE 上执行比在 Chrome 上要慢 43%，请参考表 2。

考虑到本项目进行的是功能测试，用 2000 个功能测试用例来覆盖不同浏览器的兼容测试在时效和成本上是不切实际的，兼容测试并不需要这么多的测试用例。而且，只在 IE11 上测试，也并没有覆盖了 Windows 平台的所有浏览器。相对于 Chrome 和 Fireforx，IE 不同版本之间的差异性更大，而 Microsoft Edge 甚至是另外一个完全不同于 IE 的产品。

所以，团队提出了只在 Chrome 上进行自动化测试的执行方案并予以实施。

4.3 测试用例优化

在项目已经开发的测试脚本中，通过分析执行结果及日志，发现了如表 3 中所列出的几个主要问题。针对这些需要优化的问题，团队提出并验证了相关的解决方案，见表 3。目前这些方案已经被应用到项目中，现在已经没有任何一个测试用例的执行时间会超过 10 分钟。

表 3、测试用例存在的问题

问题	原有方案	解决方案
1. 验证一个表格中所有单元格的背景颜色	遍历所有单元格，定位并验证每个单元格的背景色。	先计算出单元格的个数，按照所期望的 CSS 和属性来一次性定位出所有的单元格，然后取数量进行比较。
2. 验证表格中大量单元格是否可编辑	遍历所有的单元格，然后对每个单元格进行双击操作，依据操作的结果来判定单元格是否可编辑。	由于产品实现机制的限制，这里无法用 CSS 和属性一次性找出所有可编辑的单元格。这里，双击单元格是不可避免的。我们的新方案为进行点检查，随机抽查 5 个单元格来进行检查是否可编辑。
3. 验证 26 个员工在一周内所有的排班	虽然不是每个员工在每天都有排班，但通过界面操作去挨个检查，会消耗大量的时间。	使用 API 来准备和创建排班数据，在界面上用点检查的方式来随机抽查几个排班，从而达到验证的效果。
4. 测试用例过于复杂	按照测试用例的要求，实现所有的步骤。	用 API 来准备、销毁数据；把测试用例分拆为几个小些的测试用例。一个测试用例过于复杂会给调试和维护带来很多挑战。

4.4 最佳实践和脚本优化

4.4.1 重用浏览器和登录状态

每个测试用例都重开一个浏览器，然后进行登录、执行、退出登录，最后关闭浏览器，这样做虽然可以最廉价地隔离测试用例。但是，重复开启浏览器并进行登录浪费了大量的执行时间。从 3.1.2 节的图 1 我们可以看到，在 IE11 上每次都会消耗额外的 23 秒

钟。如果重用浏览器窗口，也不用进行登录，启动一个测试用例几乎是不花时间的，请参看图 4 所示的日志。

```
2017-07-13 17:13:02 INFO  BaseElement:441 - click:  [ notificationBtn : By.xpath: //div[@class='preview-header-panel']//*
[@id='alerts_button'] ]
2017-07-13 17:13:02 INFO  BaseTest:142 - Start test case [test_ALM109168_                                    FromHomePage]
2017-07-13 17:13:02 INFO  BaseTest:151 - Execution context parameter set ExecutionContext
[tenant=com2017071008423771_nonprd_01, frontEndServer=http://cx-b.int.dev.          .com/dashboard/, backEndServer=http://cx-b-
back.int.dev.          .com/, openAmServer=http://keng01-dev01-ath21-oam21.int.dev.          .com/]
2017-07-13 17:13:02 INFO  BasicPageElementHelper:210 - findElementsByPresence: [ By.cssSelector: li[id^='group'] ] with
timeout: 120
```

图 4 重用浏览器以及不进行重新登录的启动日志

17:13:02 - 开始启动测试用例

17:13:02 - 已经开始查找界面上相关的元素了

所以，针对这个问题，我们的优化方案如下:

基于测试类来重用浏览器窗口。原则上，在每个测试类的@BeforeClass 中启动一个浏览器窗口，类中的所有测试方法都重用这个浏览器窗口，最后在@AfterClass 方法中退出登录并销毁浏览器。

重用登录状态。在同一个测试类中，如果前后的测试用例使用了的相同用户，就可以在@BeforeClass 登录一次，类中所有的测试用例都不用再登录了。

按测试类分组执行测试用例。如果不分组，在运用了方案#1 之后，执行时会出现很多个浏览器窗口的情况（每个测试类对应的浏览器窗口只有在该类中所有的测试用例都执行完了才会被关闭），有时会导致测试客户端内存资源耗尽。利用 Testng 的 group-by-instances 把测试用例按测试类分组，然后执行。这样，一个类中的所有测试用例都按顺序都执行完后，才会执行下一个测试类，保证在整个执行过程中只有一个浏览器窗口。

4.4.2 用 API 来准备和销毁测试数据

在一些测试用例中，业务数据的创建和清除并不是测试用例所要关注的，测试用例关注的是对这些数据的后续操作。对于这些测试用例，我们利用系统 API 来创建、销毁数据，极大地提高了执行效率并减少了出错的几率（通过界面操作出错的几率远远大于通过 API）。

在测试类中，利用@BeforeClass 通过 API 来创建或清除数据。

在测试类中，利用@AfterClass 通过 API 来清除数据。

在界面的测试步骤中，如果需要用 API 来创建或清除大量的数据，那么一定要在相关页面打开或刷新之前进行，因为 Web 服务器有缓存机制，需要保证 API 对数据库的直接操作能反映到 UI 层。

只使用系统 API 网关提供的 API，其他的 API 用于系统微服务之间的调用，只用经 API 网关提供的 API 才是给客户端使用的。

另外，如果系统不是很复杂，在条件允许的情况下，也可以通过 SQL 来直接创建、销毁数据。

4.4.3 去掉不必要的验证

用例和脚本中有一些不必要的验证。比如下面的这个例子。

1. 登录，验证首页上面公司 Logo 是否出现。

2. 打开系统菜单界面，验证菜单界面上公司 Logo 是否出现。

3. 通过菜单界面，跳转到目标测试页面，进行测试……

在这个例子中，第一步和第二步的验证点都是不必要的，原因如下。

如果登录不成功，或者菜单界面没有成功打开，那么第三步根本就不会成功，这个测试用例的执行一定是失败的。

如果首页和菜单页面上的公司 Logo 被改变了（由于产品变化或者产品缺陷），那么这个测试用例的执行也会因此而失败。但是，该测试用例所测试的内容根本就不是公司 Logo（公司 Logo 应该由其他测试用例来覆盖）。

任何的验证都需要时间，这个例子中第一、二步的验证，都需要时间去定位对应的元素。

4.4.4 同一功能点只验证一次

项目中的很多测试用例在验证着相同的内容，比如创建某天排班的测试用例、编辑某天排班的测试用例，还有很多要先通过界面创建一个排班，然后再进行其他测试的用例。在这些测试中，创建排班这个界面被重复测试了很多次，这个界面上的元素被重复定位和验证了很多次，在回归测试中就会浪费大量的时间，也会带来不必要的失败。针对这样的问题，我们提出了如下的解决方案。

只在创建排班、编辑排班这两个测试用例中对“创建/编辑排班”这个界面的进行测试，并验证其上面的所有元素。

在所有其他的测试用例中，需要创建、编辑排班数据的，都用 API 在@BeforeClass 中提前进行。

五、结语

到项目结束时，打卡管理模块已经开发了 750 个测试用例，在 Chrome 上的执行时间如表 4 所示。

表 4、优化后的回归测试执行时间

任务	测试用例数	执行时长（小时）
子任务集 1	246	3.3
子任务集 2	244	3.2
子任务集 3	260	3.6

从表 4 的数据可以看出，在对测试流程、测试用例、脚本优化后，打卡管理模块的回归测试执行效率得到了很大的提升。子任务集 3 包含 260 个测试用例，在 Chrome 上的执行时间只需要 3.6 小时，相对于中期 260 个用例在 Chrome 的执行需要 7 个小时，优化后的执行效率大幅度提高了 50%。

在测试者的易用性测试工具套装中他们需要什么

◆译者：枫叶

摘要:
一个软件测试人员的适用性测试套装应该需要包含各种工具，既可以帮助测试人员"在他们的用户中行走"，又可以很快地定位明显的问题以及暴露适用性的特点（或者缺少这些特点）。高绩效只能通过人类技能实现，但是这些工具将能帮助你发现潜在的问题并让你的产品为更广泛的用户提供更好的用户体验。

软件需要被尽可能广泛的用户使用的概念已经存在 20 多年了，然而，在相当长的一段时间里，它仍然脱离了测试和开发工作的主流。

近年来这种情况一直在发生变化。我们已经看到多元化和数字包容性成为社会的优先事项。除了隐含的社会契约之外，我们现在还有明确的法律契约，就像美国康复法 508 部分和加拿大省级立法（在安大略湖的安大略湖残疾人立法的适用性，适用性的魁北克标准，和其他），定义了政府和公共部分软件的易用标准。这为整体市场设立了一个趋势性例子。

就像设计师、业务分析师和程序员，测试员需要在适用性领域获取新的专业技能。

数字可访问性暗示着不仅仅在 web 浏览器中对那些有视觉、听觉或者运动障碍的人使用方便，而且还包括在用户辅助技术支持的软件上。使用工具去识别适用性问题时一种普遍方法，但是它隐藏了陷阱。适用性主要是关于人类感知、认知和交互，并且那些方面很难被机器方法检测。这儿，让我们通过分类，强调它们的优势和风险来看适用性测试工具。

为何对适用性测试来说工具是首要的

测试者识别软件问题基于他们的思维模式、经验和感觉。但是我们如何能有效地测试一个具有不同认知和思维模式的一些产品呢？

一种方法是学习可获得性的特定的、有效的、启发式的原则，为了能够识别和分类

可能被残疾人遇到的障碍。另一个重要的办法是部署工具去替换你自己的思维和认知，这样你可以对客户的使用模式进行建模。

用户导航和操作软件产品的方式可能依赖于他们与软件的交互的认知，运动和习惯特点而变化。它也依赖于用户的域知识，特殊产品的经验，一般以特别的信息技术使用辅助性技术的技能水平。那要求多样性的工具被测试者们部署。

适用性测试工具包含哪些被特别创造于识别错误并且被测试者们以跟残疾人的相同方式使用的规则辅助技术。让我们看一看其中一些最流行的工具。

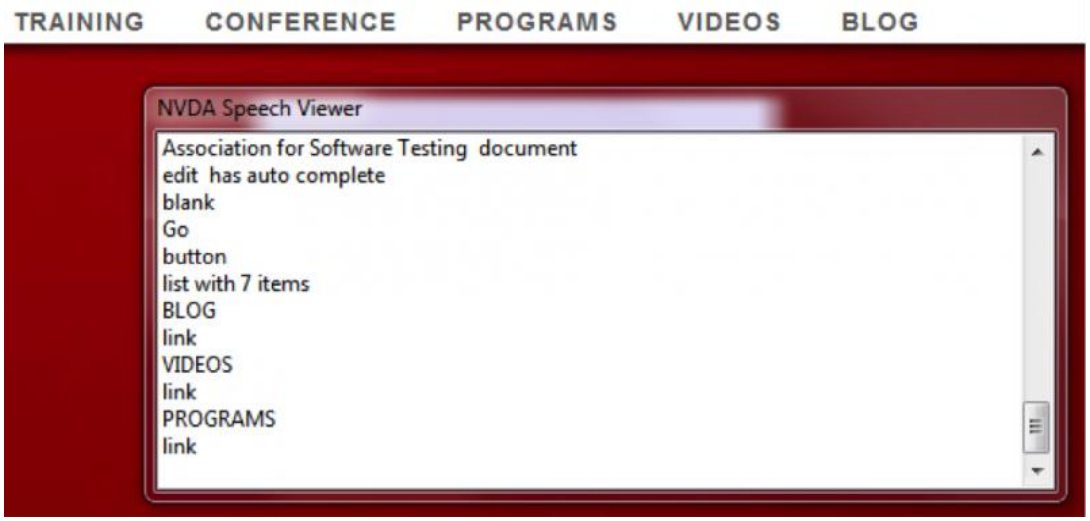
屏幕阅读器

视力障碍的人使用屏幕阅读器，软件程序能使用速度合成仪或盲文显示器读出显示在屏幕上的文字。用户能通过按不同的组合键发送指令去指导速度合成仪说什么，读什么或者拼一个单词或者全屏文字，计算机的光标位置发音，或者更多的。

屏幕阅读器作为商业的免费后者开源软件是可获得的（就像流行的说英语选项大白鲨和英伟达），并且它们也是视窗系统，苹果系统，苹果手机操作系统，安卓系统的内置选项。

屏幕阅读器不是打算作测试工具的，但是它们对适用性测试者们去仿真他们的用户亲身体验是无价的。

一些屏幕阅读器带着有用的特性比如演说日志，帮助测试者们辨别问题并备份问题报告。在下面的例子里，免费和开源屏幕阅读器英伟达通过它的演讲观看器显示了网页的口述 Tab 键顺序和所见顺序不匹配——Tab 键被设想为按从左到右的顺序，然而只键盘操作，用户从右向左，会先被给予“博客”。



超文本标记语言-检查工具

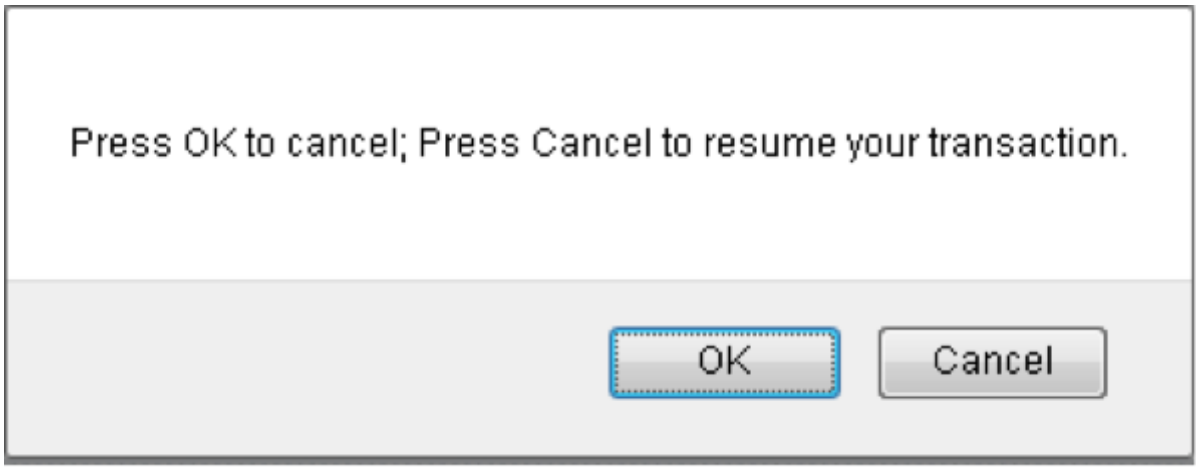
就像名字所暗示的，超文本标记语言工具扫描网页的内容并检查在它们编码里的违反规则的语法。规则都是基于国际标准的网站内容易用指南，或者 WCAG。

在市场上有多样这些工具，免费且开源选项同时收费的企业产品。它们是可获得的就像独立应用程序（比如分类网站），在线应用程序（检查器和波纹），应用程序接口集成（凸榫），以及多种浏览器插件（波纹也有一样工具）。

这类工具对于快速搜索“低挂的水果”非常有用，比如由于缺少适用性所需的超文本标记语言元素引起的明显的问题。比如，假如一幅图片没有关联的文本替换物，或者换文本，一样工具将标记它。

无论如何，该工具仍然需要一个人去决定另一个文本是否能够充分地描述特定上下文中的图像。一件工具不会区分出一张只用于视觉装饰的必须含空替代文字的图片与一张必须含有意义的替代文字的例证插图。

事实上，任何关于用户接口恰当性要求人的评估。举个例子，这个模板对话框屏幕截图没有超文本标记语言违例，但是有相关于理解这条信息的明显的可用性问题。



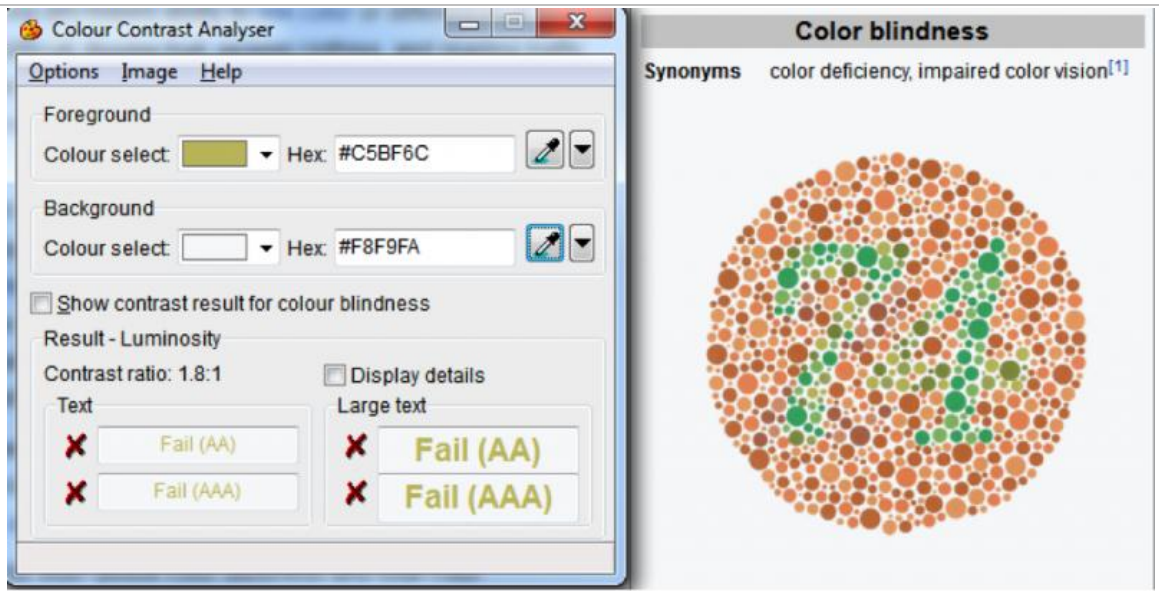
尤其是，超文本标记语言-检查工具评价一个单一网页或者网站。它们没有能力去自动导航到一个网站应用程序因为那需要数据输入和用户操作。它们也没有提供所有 WCAG 标准的完全覆盖，而且它们也无法捕捉任何标准的所有问题。

颜色和对比度检查器

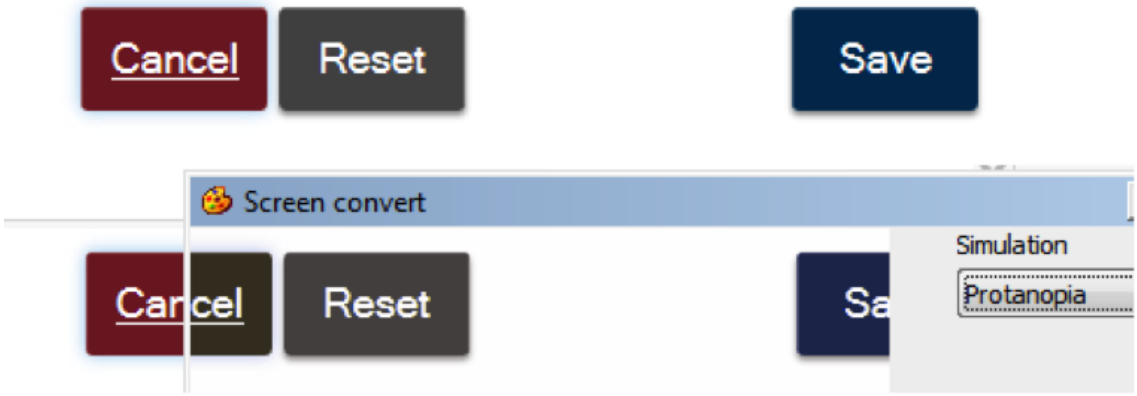
一些人不能区分某些颜色。他们可能不需要特殊的辅助技术，但是因为这个困难性，一个用户接口不应该孤立依赖于转换信息的颜色。另外，有癫痫的人是对光敏感的，摇曳或者闪烁的光能引发抽搐，所以这些元素也不应该被用于网站。

颜色和对比度检查器有助于检测这些可能有问题的元素。就像名字所暗示的，这个工具家族被用于改变颜色代码和对比度。典型地，它们要求测试者手动选择背景和前台颜色为了计算比率。

在这个例子里，颜色对比度分析器工具评估从维基百科中的图片在色盲和不可预见地，展示文字没通过测试。



更先进的工具可能包含一个仿真模式提供给测试者们亲身感觉体验，所以他们知道他们的用户将要看什么。在下面，相同的工具以仿真模式展示了对于红色盲或者红-绿色盲的人无法从红色的取消按钮和灰色重置按钮区分开来。



图形用户界面自动化工具

常规的图形用户界面自动化工具，比如同一标准的功能测试（UFT）或者 Selenium，能被成功地部署给覆盖部分适用性需求的回归测试。因为图形用户界面自动化工具做交互和导航网页应用程序，有时它对合并自定义检查点去克服通用的适用性检查器是可行的。在 HTML 中适用性特点的验证不同于 ID，标签，和通过图形用户界面自动化通常验证的其他元素的验证。

可读性分析器

对于那些有限能力去处理和记忆信息，做决定，或者在很长一段时间内集中注意力的人来说，过于复杂的语言会很使得迷茫或者沮丧。这类也包括压力或者分心的条件下

的人们或者在语言上不流利的人。为了辅助这些用户，软件必须以一种清晰和有组织的方式展示信息，提醒人们关于重要点，并且允许验证和纠正。这些特点一起被称为是可读性。

有语言方法去计算写好文本（比如，弗莱什-金凯德可读性测验）的复杂指数。这个指数估算一个人在第一次阅读时需要理解这个文本的正式教育的年数，并且这能被绑定到认知努力的级别。有免费的在线工具用于基于在这些和其他标准上的文本估算。由于其他机械论的意思，你不能单独依赖这些工具——人的验证仍然是更可取的——但是它们对于快速，粗略地检查为标记潜在的有问题的语言是有用的。

普通工具

低视力，散光视力或者斑点视力的人可能不需要使用屏幕阅读器但是仍然喜欢以阅读的文本协助。他们可能使用在浏览器里的缩放功能，降低屏幕分辨率，或者增加字体大小，对比色阶，和色彩极性。

暂时性或永久地无法控制自己身体的人，特别是他们的胳膊或者手，可能使用很多种物理和电的辅助技术，但是在软件操作方面，实际上是没有鼠标帮助只键盘使用模式。

这些选项不要求任何特殊的测试工具。比如，只使用键盘（tab 和 shift-tab）导航整个网页的能力是基本的遵从性。当“指定通过”时，测试者也须检查焦点是否经常是可见的和可很好区分的。

其他对测试者们有用的工具被固定进浏览器里。按 F12 打开检查模式，允许测试者们去检查一个正在考虑的用户界面元素的超文本标记语言源。

在下面的例子，为图片转换的文本被使用网页开发工具栏高亮，并且我们有既充份又不合适的文本描述的例子。

这样浏览器工具可能不特别指出适用性问题，但是它们依然极大帮助测试者们找到它们。

加入你的工具箱

一个软件测试者的适用性测试工具箱应该包含多样工具，不仅帮助测试人员“在他们的用户中行走”，又可以很快地定位明显的问题以及暴露适用性的特点（或者缺少这些特点）。就像在有技术测试的每一个方面一样，高绩效只能在人类技能的深度参与下才能实现，而不是仅依赖于工具，但这些会帮助你发现潜在的问题，并使得你的产品为更广泛的用户有更好的用户体验。

TDP—测试驱动性能

◆ 作者：万晓光

摘要

本文讲述了软件开发过程中普遍出现的性能问题，分析了性能问题产生的根源，并提出了解决性能问题的策略—TDP（Test-driven Performance 测试驱动性能）。结合公司的实际情况，制定了软件开发过程中应用 TDD 的解决方案。在实际的项目中，也对 TDD 进行了实践。结果表明，依照测试驱动性能的实践，软件的性能有了可靠的保障。

一、前言

在很多软件项目中，开发者们通常在前期都不太注重性能等非功能性需求，直到项目进入到交付阶段或系统上线之后，许多在压力负载较大情况下发生的问题才被发现，导致开发工作陷入一片混乱。然而，即使在设计阶段对性能进行过比较完善的考虑和设计，总还是有一些现实的问题是无法仅仅通过设计就能解决的。面对性能问题，本文通过分析公司的实际案例，结合公司的实际情况，提出了性能问题的解决方案—TDP（Test-Driven Performance，测试驱动性能），并应用到实际的项目交付流程中，到目前为止取得了比较好的效果。

二、一次真实的性能危机

笔者所在的公司，前段时间一个“云端数字资产管理平台”项目发生了一次性能危机。系统上线之后，在高并发的情况下，生产环境的两台认证 LDAP 服务器都宕机了。而且，在一个月左右的时间内，接连发生了两次。更为严重的是，每次宕机都导致了一小部分数据的丢失。毫无疑问，这是一个非常典型的、由于系统性能缺陷而引发的问题。

当时，客户满意度急剧下降，整个项目因此进入了一个非常危机时刻。面对危机，公司的应对比较及时，调用了很多专家对问题进行分析、找解决方案。最终，我们成功地解决了问题，度过了危机，但也付出了相当大的代价。如图 1 所示，我们在分析、重

现、解决、验证、部署等工程活动上，至少花了 10 个人周的工作量。

图 5 解决危机的工作量

总而言之，任何项目一旦发生性能问题，特别是后期或系统上线之后，其修复代价往往是巨大的，成本上是难以承受的。

该项目的这个性能问题虽然最终得到了妥善解决，从软件工程的角度来看，这个性能问题的根本原因是什么？我们如何又能避免类似的问题再次出现呢？

三、性能危机的根源

就云端数字资产管理平台项目的人员构成、以及整个事件的处理和结果来看，这次项目性能危机的根源在于性能测试环节。

在项目初期，项目团队对系统选型进行过比较充分的讨论和研究，对系统架构也进行了比较完善的设计和评审，团队成员的技术能力也很强。而且，在项目进入中期后，团队规划并实施了系统级别的性能测试。按道理说，这样的团队发布的产品应该不会有大问题。但是，性能测试方案有问题而没有真实地模拟出生产环境的状况，这就导致了应该在测试阶段暴露的问题遗留到了生产环境。

四、性能问题的普遍性

2016 年下半年，我们挑选了 5 个项目进行了一次调查，对比了五个项目在性能方面碰到过的问题、挑战、以及应对策略。

根据得到的数据和信息，对被调查的项目在基本技能、设计平衡、测试、调试、实践分享这五个方面进行了对比统计。采用的方法是：对于某一个方面，比如基本技能，如果一个项目做得还可以，就记 1 分，做得不好，就扣 1 分，与项目不太相关的方面就不参与统计计算，然后求和。

表 1、项目性能工程活动比较分析

工程活动	项目 1	项目 2	项目 3	项目 4	项目 5	合计
基本技能和经验	+1	+1	-1	+1	-1	1

需求设计平衡	n/a	n/a	-1	+1	-1	-1
性能测试	-1	-1	-1	-1	-1	-5
分析和调优	n/a	-1	n/a	+1	+1	1
实践分享	+1	+1	-1	+1	-1	1

最终结果是，在性能测试上的分数最低。有的项目是性能测试本身的方案有问题，有的项目是在生产环境出现性能问题后再补性能测试，有的项目因客户没有明确需求而甚至没有做过性能测试。

这五个项目在生产环境都暴露过比较严重的性能问题，而普遍最缺失的是性能测试。

五、TDP—测试驱动性能

出了问题不可怕，从问题中吸取经验和教训，找到解决问题的方案，预防类似问题的再次出现才是最重要的。结合我们公司的交付流程和各项目的实际情况，经过 TEC（Technology Excellence Committee，技术卓越委员会）内部充分讨论，我们提出了 TDP—测试驱动性能的解决方案。

在项目的实施过程中，前期的设计固然重要，但并不能解决所有的性能问题。而且，如果过度设计的话，反而会适得其反，导致项目成本上升或交付延期。所以，在项目前期对性能进行适当的考虑和设计，在敏捷迭代的过程中，再用测试来驱动是比较客观而有效的方式。

性能测试可以尽早地发现性能问题。可以在很大程度上预防性能问题延迟到生产环境才被暴露出来。

敏捷。测试驱动是基于结果的驱动，是敏捷方法的核心实践。用性能测试进行一定的敏捷迭代，可以保证系统的性能。

倒逼设计进行恰当的重构。性能测试可以及时发现设计上的缺陷，以免架构的问题累积到后期而一发不可收拾。

测试驱动更加客观、直接。测试的结果是定量的，数字结果一目了然，大家不会有太大的分歧。

性能测试的数据可以指导系统扩容，有助于确定什么时候应该进行扩容。

性能测试的基准数据也很大的商业价值。好的性能数据在市场、销售环节可以吸引

和打动客户，促成交易。

六、某公司的 TDP 方案

性能测试一般由专业团队来进行，但考虑到我们公司项目的实际情况，性能测试一般是开发团队自己进行的，而有的项目团队在性能测试方面的经验不足，所以让有经验的 TEC 成员参与进来是一个比较好的实践。

结合我们公司项目交付的流程，TDP 也需要 DOO（Delivery Operation Office，项目交付办公室）的参与，在一些项目关键里程碑节点来检查性能测试相关活动是否按计划进行。

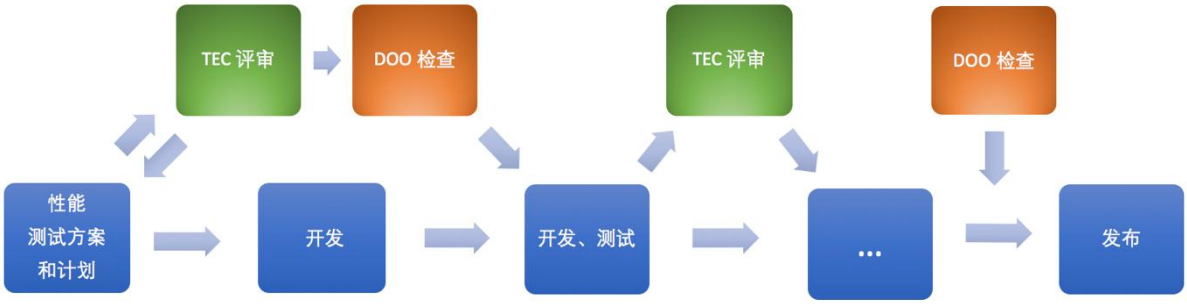


图 6 奥博杰天的测试驱动性能

图 2 示意了在项目开发的过程中，项目团队、TEC 和 DOO 的互相配合来保证性能测试。

在项目初期根据需求定出性能目标，并根据性能目标确定好性能测试的方案和计划。

交由 TEC 评审性能测试方案和计划。TEC 成员着重关注如下方面:

- 性能目标 – 项目设定的性能目标是否合理，是否可行。
- 性能测试方案 – 是否模拟了真实的产品环境，真实的压力。
- 性能测试计划 – 评估在系统上线之前是否有足够轮次的性能测试。

DOO 对上面的项目计划和工程活动进行检查。

如果上面都没有问题，在开发到了一定的阶段之后，性能测试就可以按计划介入了。

性能测试的结果需要和相关 TEC 成员一起评审，排查问题，解决问题。

产品上线之前，DOO 还需要最后一次检查性能测试结果，看看是否满足项目性能指标。

上面这个流程，保障了项目的性能测试，让系统的性能在上线之前得到检验。如果有问题，那么问题也会在上线之前被发现、被解决。从而保障交付的软件系统在性能方面没有重大的质量问题。

七、TDP 实践

2017 年初，公司在一个“数字内容驱动排版系统”项目开始了 TDP 实践，如图 3 所示。

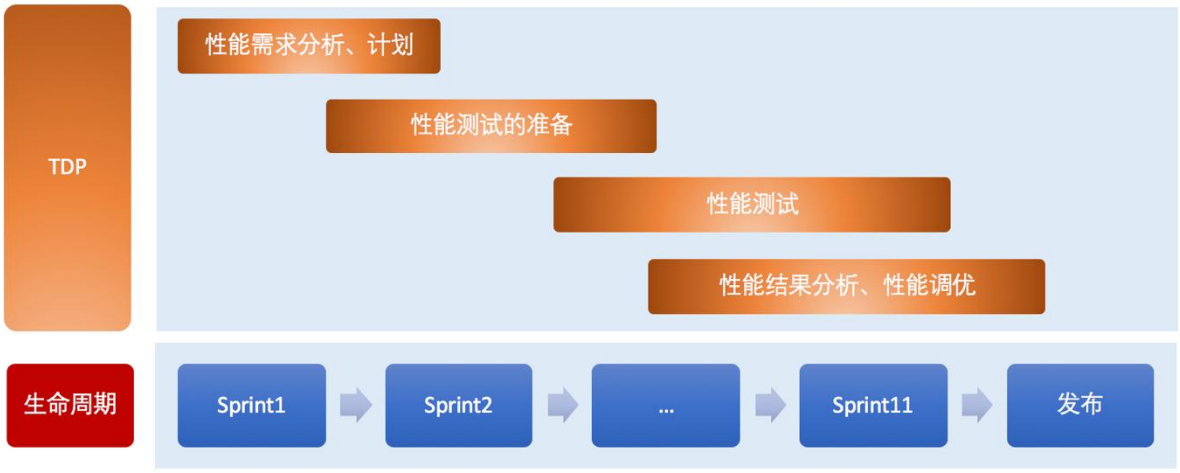


图 7 TDP 的应用

在项目前期，团队分析了性能需求，在设计中考虑了性能要求，制定了性能及测试的相关计划。公司 TEC 和 DOO 对性能方案、测试、计划等进行了的评审和审核。在基本核心功能端到端实现后，团队为后续的性能测试进行了准备工作 。从第 3 个 Sprint，项目就开始了性能测试的迭代。到目前为止（Sprint 8），团队通过 TDP 已经发现了 5 个比较显著的性能问题和一个设计方案的问题。

八、TDP 解决的设计方案问题

该项目有一个文档装配引擎，团队在一开始有两个备选方案。如下面的图 4 和图 5 所示。

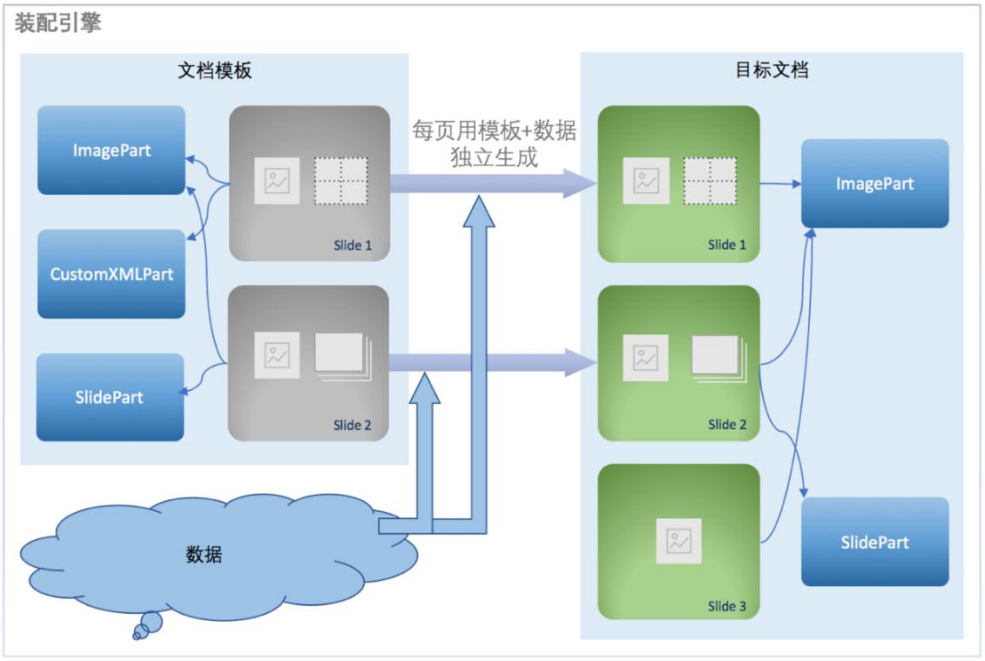


图 8 方案一

方案一先创建一个空白文档作为目标文档，然后一页一页地依据文档模板和数据来创建内容，在这个过程中每一页上不需要的 Part 已经被删除了。

方案二采用了另外一个思路，直接用数据来填充内存中的文档模板从而直接得到目标文档，但由于模板中可能有目标文档并不需要的 Part，最后还要删除文档中不需要的 Part（如 slide part，image part 和 custom xml part 等） 从而得到目标文档。

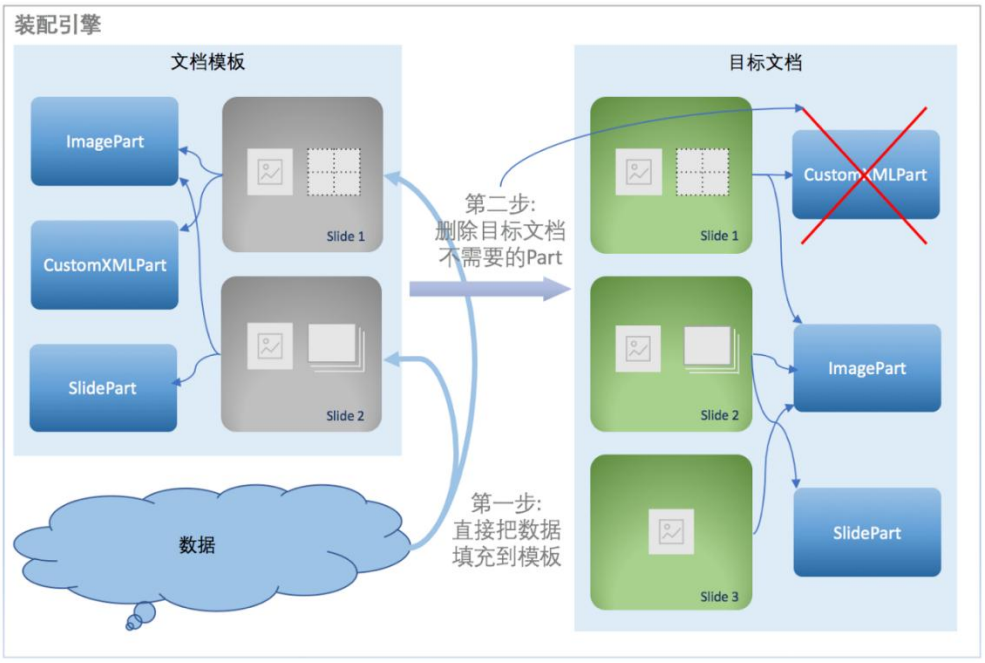


图 9 方案二

一般情况下，文档模板中有相当比例的静态内容，方案一就会带来一些不必要的操作：基于静态内容来创建静态内容。为了节省这部分不必要的操作，团队决定采用第二个方案，这也比较符合人们的思考习惯。

团队快速地进行了方案二的概念验证，在功能上没有问题。依据 TDP 实践，团队继续对方案二的快速原型进行了性能验证。经测试发现，方案二的第二步操作效率很低下。分析后发现是由于第三方文档删除 Part 的机制所限：每删除一个 Part，就需要对整个文档遍历一遍。

团队继续对方案一、二进行对比性能验证，结果如表 2 所示，方案一的整体性能显著地优于方案二。所以，团队最后采用了方案一（空间换性能）来实现装配引擎。

表 2、两个方案的性能对比验证结果

方案	模板文档	目标文档	平均耗时（秒）
方案一	16 页 （包含 3 页纯静态内容）	53 页	17
方案二			240

TDP 帮助团队在早期纠正了装配引擎方案的问题。如果在早期没有进行性能测试和性能对比测试，方案二的性能问题可能要延迟到中后期才会被发现，那时候解决的成本和难度就要大多了。

九、结语

在项目过程中，用 TDP 对性能进行敏捷迭代是行之有效的方法，直接、客观的性能测试可以提前发现系统的性能问题，在早期对架构和相关设计进行性能验证，从而保障系统发布之后的性能质量，最终大大节省项目成本。

如何使用 REST Assured 来提取 Spring Boot API 测试的值

◆译者：linda liu

目前，对 API 测试来讲，最重要的是能提取出 API 测试的数值，并在后续的 WEB 请求中使用这些值。在本文中我将展示如何利用 Java，Gradle，Intellij 以及 REST Assured 来提取 API 测试的数值并使这些值能真正生效。

Intellij 是一个集成开发环境，REST Assured 是一个开源的框架，主要用来测试 Java 版的 REST web 服务，而 Gradle 是一个构建工具。大家可以选择不同的工具来做 API 测试，这取决于你的喜好，当然也取决于你要测试的项目。

作为演示，我们创建一个 Spring Boot 和 Hibernate 之间的 Java API 测试。此 API 有 4 个控制器分别为到达、出发、用户和航班。它们之间有两个或多个路径来调用不同类型的请求。路径上的请求大部分都是 GET 请求，但在某些路径上也有 POST 和 DELETE 请求。大家可以自己尝试运行测试。

这里展示的测试仅包括一些简单的路径测试，例如 JSON 响应的语法验证，请求时间，允许的方法等，但单元测试不包括在内。

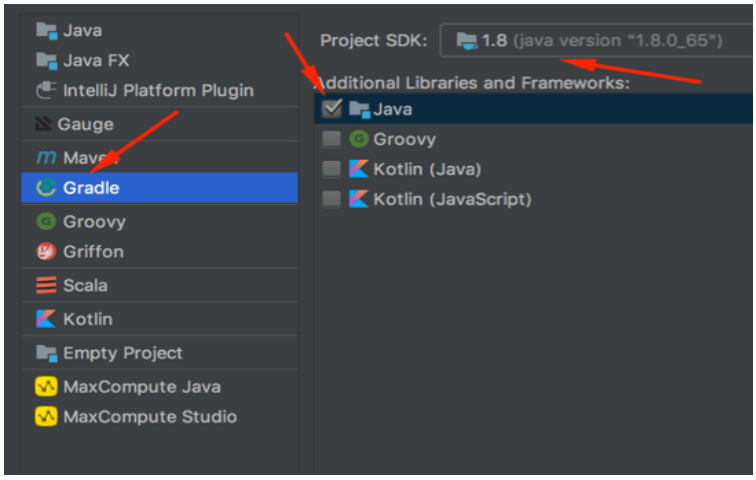
第一步：创建一个 API 测试项目

1、安装 Intellij IDEA;

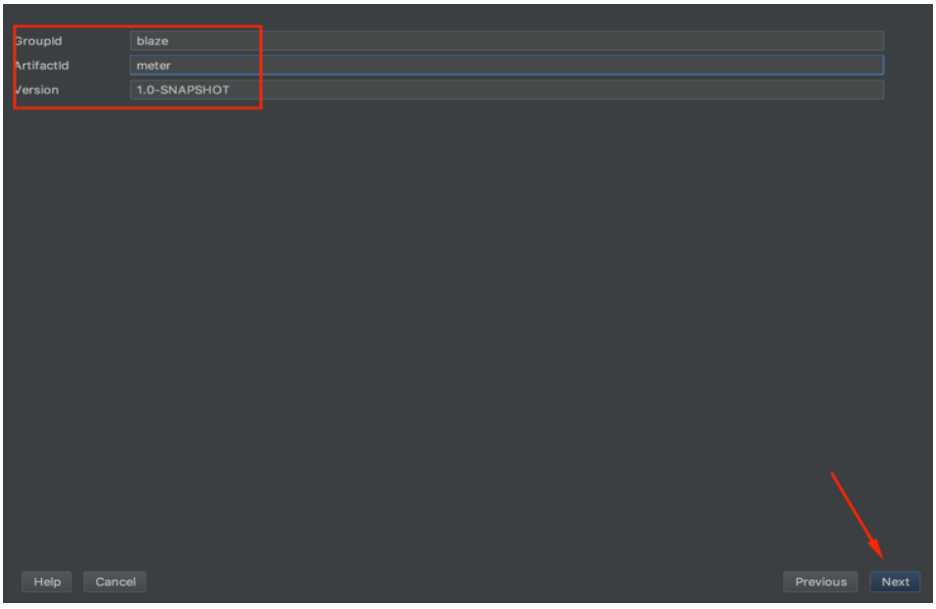
2、确认已经安装了 Java JDK（至少 1.8.x 的版本），现在我们就开始创建一个新的工程;

3、打开 IntelliJ 并点击“Create New Project”;

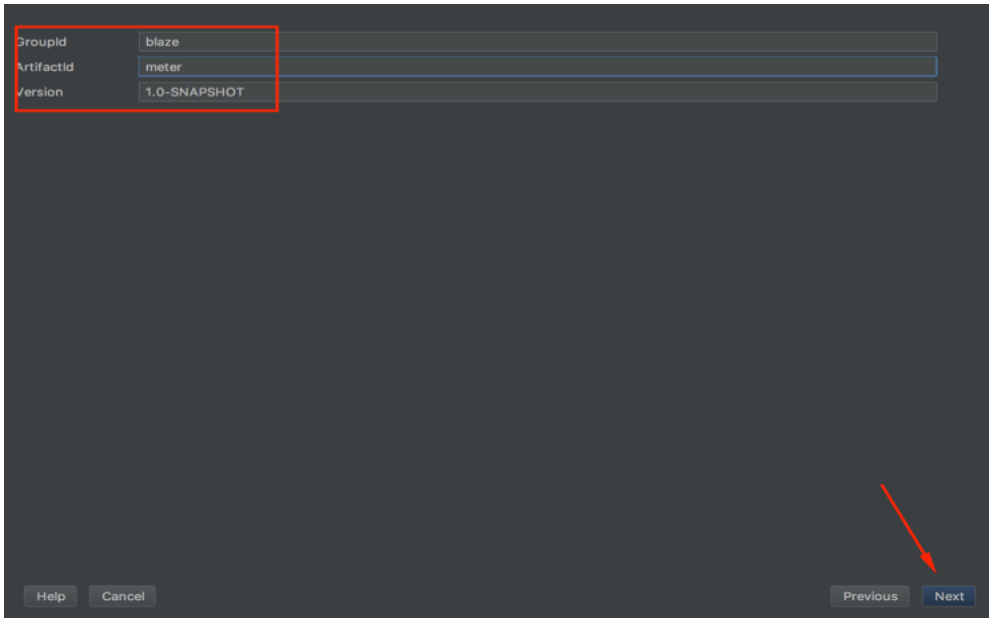
4、选择 Gradle，勾选 Java 并选择 JDK 版本；



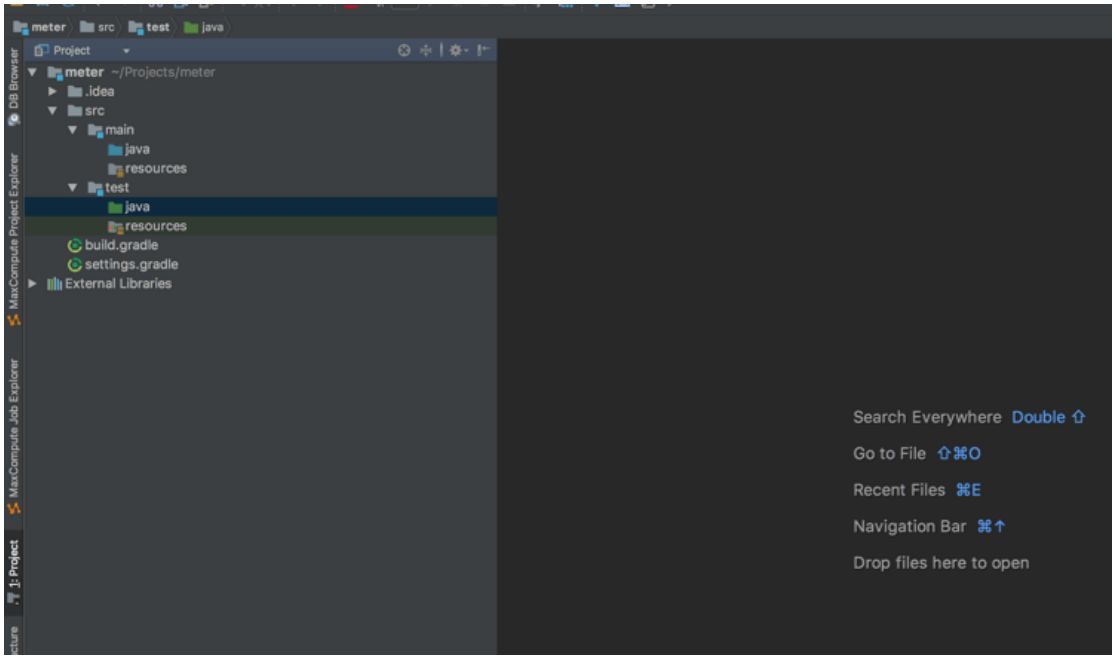
5、给你的工程命名；



6、选择工程文件的属性；



如果你做的一切都正确的话，这时你应该看到一个空的 Java 工程的窗口。



第二步：添加依赖

现在我们已经创建了一个工程，接下来需要建立依赖。由于这些依赖都是公有的，我们所有人都可以使用。怎样使用依赖呢？双击你的 build.gradle 文件，并添加下面的 gradle 配置文件：

```
group 'blaze'
version '1.0-SNAPSHOT'
```

```
buildscript {
    repositories {
        jcenter()
        mavenCentral()
        maven { url "http://repo.spring.io/libs-snapshot" }
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.5.2.RELEASE")
    }
}

apply plugin: 'java'
apply plugin: 'idea'
apply plugin: 'io.spring.dependency-management'
apply plugin: 'org.springframework.boot'

sourceSets {
    main.java.srcDir "src/main/java"
    main.resources.srcDir "src/main/resources"
    test.java.srcDir "src/test/java"
    test.resources.srcDir "src/test/resources"
}

jar {
    baseName = 'blaze-demo-api'
    version =    '1.0'
}

bootRepackage {
    mainClass = 'com.demo.BlazeMetterApi'
}

dependencyManagement {
```

```
    imports {
        mavenBom 'io.spring.platform:platform-bom:Brussels-SR2'
    }
}

repositories {
    mavenCentral()
    jcenter()
    maven { url "http://repo.spring.io/libs-snapshot" }
}

sourceCompatibility = 1.8
targetCompatibility = 1.8

dependencies {
    compile group: 'org.springframework', name: 'spring-core'
    compile group: 'org.springframework.boot', name: 'spring-boot-starter-jdbc'
    compile group: 'org.springframework.boot', name: 'spring-boot-starter-web'
    compile group: 'org.springframework.boot', name: 'spring-boot-starter-actuator'
    compile group: 'org.springframework.boot', name: 'spring-boot-starter-security'
    compile group: 'org.springframework.boot', name: 'spring-boot-starter-data-jpa'
    compile group: 'org.springframework.security.oauth', name: 'spring-security-oauth2'
    compile group: 'com.fasterxml.jackson.datatype', name: 'jackson-datatype-hibernate4'
    compile group: 'mysql', name: 'mysql-connector-java'
    compile group: 'io.rest-assured', name: 'rest-assured', version: '3.0.3'
    compile group: 'io.rest-assured', name: 'json-schema-validator', version: '3.0.3'

    testCompile group: 'org.springframework.boot', name: 'spring-boot-starter-test'
    testCompile group: 'junit', name: 'junit'
}
```

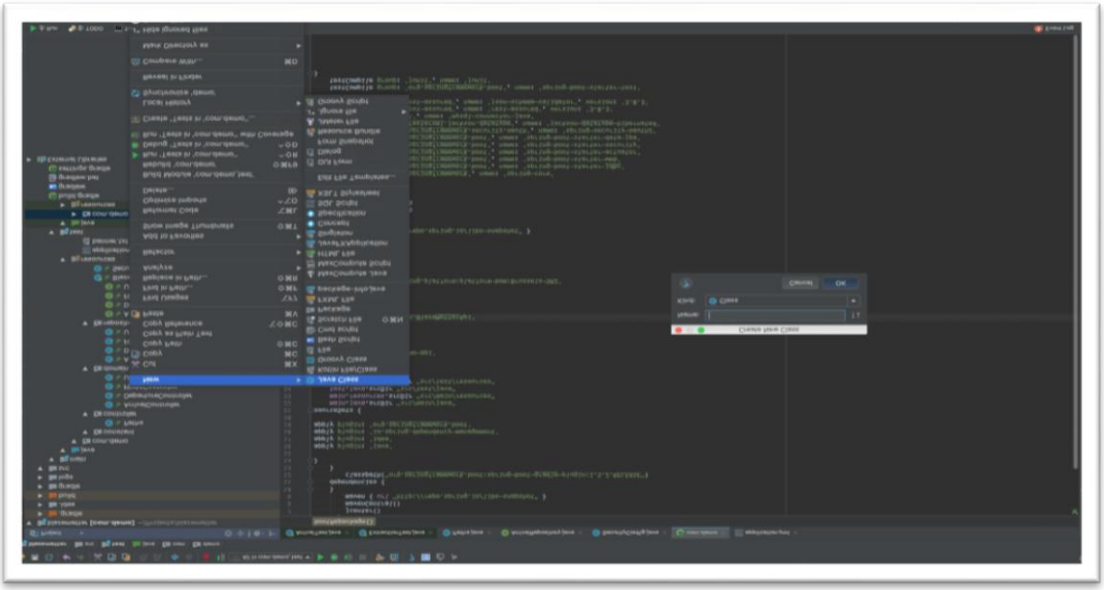
第三步：基于 REST Assured 书写 API 功能测试代码

要使用 REST Assured 来开始测试，首先需要创建测试类。REST Assured 能提供给我

们所需要的简单的、容易阅读的 API 调用和解析能力。

在 IntelliJ 上右键单击 test/java，选择 select New，然后选择 Java Class。

填入名称 ArrivalTest 然后点击 enter 键。我们在这里使用这个名字，是因为我们要提取和使用的数据是用来填充查找航班的表单的（如下图所示）:



让我们来写第一个测试类吧！这个测试类调用 API（ /arrival）的路径并执行了如下测试:

checkArrivalEndpointStatus -验证响应代码为 200 的路径(SC_OK)

checkSchemaValidity - 验证响应体包含的 json 串是否匹配有效语法

checkResponseTimeAll - 验证对路径/all 的响应时间少于 2000 毫秒

checkResponseTimeById - 验证对路径/all 的响应时间少于 200 毫秒

checkPutMethod - 验证不允许使用 Put 方法

checkPostMethod - 验证不允许使用 Post 方法

checkDeleteMethod - 验证不允许使用 Delete 方法

```
import org.junit.Test;
import static io.restassured.module.jsv.JsonSchemaValidator.matchesJsonSchemaInClasspath;
import static org.apache.http.HttpStatus.SC_METHOD_NOT_ALLOWED;
import static org.apache.http.HttpStatus.SC_OK;
```

```
import static org.hamcrest.Matchers.lessThan;

public class ArrivalTest extends BaseTest {
    @Test
    public void checkArrivalEndpointStatus() {
        prepareGet(ARRIVAL_ALL).statusCode(SC_OK);
    }
    @Test
    public void checkSchemaValidity() {
        prepareGet(ARRIVAL_ALL)
                .assertThat()
                .body(matchesJsonSchemaInClasspath("schemas/arrival_schema.json"));
    }
    @Test
    public void checkResponseTimeAll() {
        prepareGet(ARRIVAL_ALL)
                .time(lessThan(2000L));
    }
    @Test
    public void checkResponseTimeById() {
        prepareGet(ARRIVAL_ALL_BY_ID)
                .time(lessThan(ENDPOINT_RESPONSE_TIME));
    }
    /*
     * Negative testing section
     */
    @Test
    public void checkPutMethod() {
        preparePut(ARRIVAL_ALL_BY_ID, DUMMY_TEST_JSON)
                .then()
                .statusCode(SC_METHOD_NOT_ALLOWED);
    }
    @Test
```

```
    public void checkPostMethod() {
        preparePost(ARRIVAL_ALL, DUMMY_TEST_JSON)
                .then()
                .statusCode(SC_METHOD_NOT_ALLOWED);
    }
    @Test
    public void checkDeleteMethod() {
        prepareDelete(ARRIVAL_ALL_BY_ID)
                .statusCode(SC_METHOD_NOT_ALLOWED);
    }
}
```

第二个测试类将调用 API（/departure）路径并执行如下测试:

- checkDepartureEndpointStatus - 验证响应代码为 200 的路径(SC_OK)
- checkSchemaValidity - 验证响应体包含的 json 串是否匹配有效语法
- checkResponseTimeAll - 验证对路径/all 的响应时间少于 2000 毫秒
- checkResponseTimeById - 验证对路径/all 的响应时间少于 200 毫秒
- checkPutMethod - 验证不允许使用 Put 方法
- checkPostMethod - 验证不允许使用 Post 方法
- checkDeleteMethod - 验证不允许使用 Delete 方法

```
import org.junit.Test;
import static io.restassured.module.jsv.JsonSchemaValidator.matchesJsonSchemaInClasspath;
import static org.apache.http.HttpStatus.SC_METHOD_NOT_ALLOWED;
import static org.apache.http.HttpStatus.SC_OK;
import static org.hamcrest.Matchers.lessThan;

public class DepartureTest extends BaseTest {
    @Test
    public void checkDepartureEndpointStatus() {
        prepareGet(DEPARTURE_ALL).statusCode(SC_OK);
    }
```

```
@Test
public void checkSchemaValidity() {
    prepareGet(DEPARTURE_ALL)
            .assertThat()
            .body(matchesJsonSchemaInClasspath("schemas/departure_schema.json"));
}
@Test
public void checkResponseTimeAll() {
    prepareGet(DEPARTURE_ALL)
            .time(lessThan(ENDPOINT_RESPONSE_TIME));
}
@Test
public void checkResponseTimeById() {
    prepareGet(DEPARTURE_ALL_BY_ID)
            .time(lessThan(ENDPOINT_RESPONSE_TIME));
}
/*
 * Negative testing section
 */
@Test
public void checkPutMethod() {
    preparePut(DEPARTURE_ALL_BY_ID, DUMMY_TEST_JSON)
            .then()
            .statusCode(SC_METHOD_NOT_ALLOWED);
}
@Test
public void checkPostMethod() {
    preparePost(DEPARTURE_ALL, DUMMY_TEST_JSON)
            .then()
            .statusCode(SC_METHOD_NOT_ALLOWED);
}
@Test
public void checkDeleteMethod() {
```

```
        prepareDelete(DEPARTURE_ALL_BY_ID)
                .statusCode(SC_METHOD_NOT_ALLOWED);
    }
}
```

第三个测试类将调用 API （/flight）路径并且执行如下测试:

- checkArrivalEndpointStatus - 验证响应代码为 200 的路径(SC_OK)
- checkSchemaValidity - 验证响应体包含的 json 串是否匹配有效语法
- checkResponseTimeAll - 验证对路径/all 的响应时间少于 2000 毫秒
- checkResponseTimeById - 验证对路径/all 的响应时间少于 200 毫秒
- checkPutMethod - 验证不允许使用 Put 方法
- checkPostMethod - 验证不允许使用 Post 方法
- checkDeleteMethod - 验证不允许使用 Delete 方法

```
import org.junit.Test;
import static io.restassured.module.jsv.JsonSchemaValidator.matchesJsonSchemaInClasspath;
import static org.apache.http.HttpStatus.SC_METHOD_NOT_ALLOWED;
import static org.apache.http.HttpStatus.SC_OK;
import static org.hamcrest.Matchers.lessThan;

public class FlightTest extends BaseTest {
    @Test
    public void checkArrivalEndpointStatus() {
        prepareGet(FLIGHT_ALL).statusCode(SC_OK);
    }
    @Test
    public void checkSchemaValidity() {
        prepareGet(FLIGHT_ALL)
                .assertThat()
                .body(matchesJsonSchemaInClasspath("schemas/flight_schema.json"));
    }
    @Test
```

```
    public void checkResponseTimeAll() {
        prepareGet(FLIGHT_ALL)
                .time(lessThan(ENDPOINT_RESPONSE_TIME));
    }
    @Test
    public void checkResponseTimeById() {
        prepareGet(FLIGHT_ALL_BY_ID)
                .time(lessThan(ENDPOINT_RESPONSE_TIME));
    }
    /*
     * Negative testing section
     */
    @Test
    public void checkPutMethod() {
        preparePut(DEPARTURE_ALL_BY_ID, DUMMY_TEST_JSON)
                .then()
                .statusCode(SC_METHOD_NOT_ALLOWED);
    }
    @Test
    public void checkPostMethod() {
        preparePost(DEPARTURE_ALL, DUMMY_TEST_JSON)
                .then()
                .statusCode(SC_METHOD_NOT_ALLOWED);
    }
    @Test
    public void checkDeleteMethod() {
        prepareDelete(DEPARTURE_ALL_BY_ID)
                .statusCode(SC_METHOD_NOT_ALLOWED);
    }
}
```

第四个测试类将调用 API（/flight）路径并且执行如下测试:

- checkArrivalEndpointStatus - 验证响应代码为 200 的路径(SC_OK)

- checkSchemaValidity - 验证响应体包含的 json 串是否匹配有效语法
- checkResponseTimeAll - 验证对路径/all 的响应时间少于 2000 毫秒
- checkResponseTimeById - 验证对路径/all 的响应时间少于 200 毫秒
- checkPutMethod - 验证不允许使用 Put 方法
- checkPostMethod - 验证不允许使用 Post 方法
- checkDeleteMethod - 验证不允许使用 Delete 方法

```
import org.junit.Test;
import static io.restassured.module.jsv.JsonSchemaValidator.matchesJsonSchemaInClasspath;
import static org.apache.http.HttpStatus.SC_METHOD_NOT_ALLOWED;
import static org.apache.http.HttpStatus.SC_OK;
import static org.hamcrest.Matchers.lessThan;

public class UsersTest extends BaseTest {
    @Test
    public void checkArrivalEndpointStatus() {
        prepareGet(USERS_ALL).statusCode(SC_OK);
    }
    @Test
    public void checkSchemaValidity() {
        prepareGet(USERS_ALL)
                .assertThat()
                .body(matchesJsonSchemaInClasspath("schemas/users_schema.json"));
    }
    @Test
    public void checkResponseTimeAll() {
        prepareGet(USERS_ALL)
                .time(lessThan(ENDPOINT_RESPONSE_TIME));
    }
    @Test
    public void checkResponseTimeById() {
```

```
            prepareGet(USERS_ALL_BY_ID)
                    .time(lessThan(ENDPOINT_RESPONSE_TIME));
    }
    /*
     * Negative testing section
     */
    @Test
    public void checkPutMethod() {
        preparePut(USERS_ALL_BY_ID, DUMMY_TEST_JSON)
                .then()
                .statusCode(SC_METHOD_NOT_ALLOWED);
    }

    @Test
    public void checkPostMethod() {
        preparePost(USERS_ALL, DUMMY_TEST_JSON)
                .then()
                .statusCode(SC_METHOD_NOT_ALLOWED);
    }
}
```

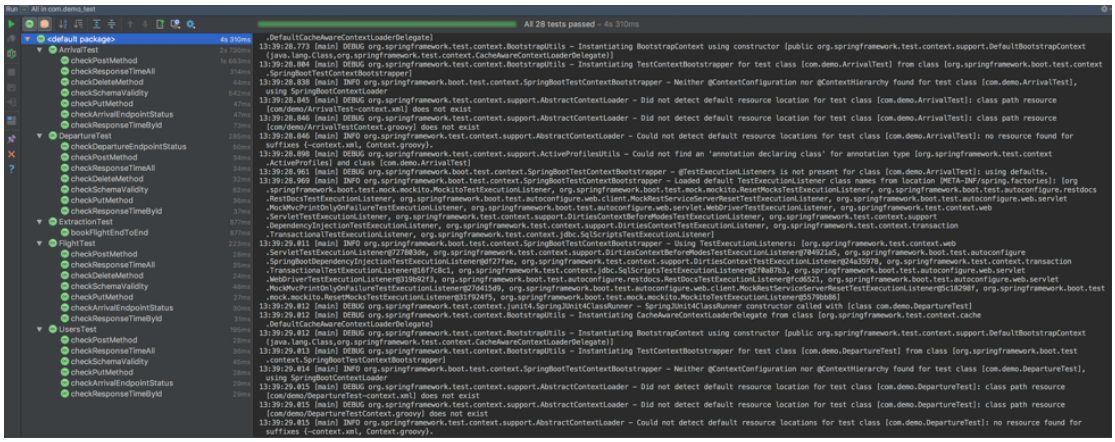
最后一个测试类（bookFlightEndToEnd）将调用所有的 API 路径，基于每一条路径，我们需要提取数据，检查端到端的工作流。你能看到从“fromCity”, “toCity”以及 “flightId”中提取出来的数值。

```
import io.restassured.response.Response;
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class ExtractionTest extends BaseTest {
    @Test
    public void bookFlightEndToEnd() {
        String fromCity = prepareGet(DEPARTURE_ALL_BY_ID).extract().path("city");
        String toCity = prepareGet(ARRIVAL_ALL_BY_ID).extract().path("city");
```

```
String jsonCities = "{\"from\":\"" + fromCity +
        "\", \"to\":\"" + toCity +
        "\"}";
String flightId = preparePost("search", jsonCities).then().extract().path("id");

String cardNumber = "2864528645765429346";

String jsonUserData = "{\n" +
        "\t\"name\": \"Greg\",\n" +
        "\t\"address\": \"somewhere\",\n" +
        "\t\"city\": \"yerevan\",\n" +
        "\t\"state\": \"hayastan\",\n" +
        "\t\"zipCode\": \"0006\",\n" +
        "\t\"cardType\": \"master\",\n" +
        "\t\"cardNumber\": \"" + cardNumber + "\",\n" +
        "\t\"cardExpirationMonth\": 9,\n" +
        "\t\"cardExpirationYear\": 20,\n" +
        "\t\"cardNameOnCard\": \"Greg\",\n" +
        "\t\"flightId\": " + flightId + "\n" +
        "}";

Response resultJson = preparePost("users/", jsonUserData)
        .then()
        .extract().response();

String cN = resultJson.path("cardNumber");
int id = resultJson.path("id");
assertEquals(cardNumber, cN);

//clean up of DB
prepareDelete("users/" + id)
        .statusCode(200);
```

```
    }
}
```

当所有的事情做完之后，需要执行创建的所用测试类。如果一切顺利的话，我们会看到如下图所示的结果。如果有某个测试类失败，我们就会在左侧看到测试类上有红圈标记。



祝贺你！至此，你已经可以创建并运行自己的基于 REST Assured 的 API 测试了。

测试者如何开始像用户那样思考？

◆ 译　者：枫　叶

摘要:

当提到测试者应该关注什么的时候，人们常说你需要像用户一样思考。阿列克西斯图隆恩惯于觉得他擅长那样做——直到他开始坐在他的应用程序用户身边，然后他想起测试人员必须经常考虑的所有问题。他讨论了你可以从你的用户身上学到什么。

当提到测试者需要关注什么的时候，人们经常说需要像用户一样思考。我认为我十分擅长这样做——实际上我从一开始就坐在用户身边。

现在我知道有很多种我需要作为一个测试者更经常考虑的问题。用户在什么样的物理环境下使用这款应用程序？当他们尝试解决应用程序的问题时什么样的压力正影响他们？什么使他们受挫？他们重视什么？

如果你能坐在那些在日常生活中使用你的应用程序的人旁边，所有这些问题都将比较容易回答。我喜欢分享坐在用户身边的体验，用户教了我关于我们的用户怎么想，我们的应用程序如何满足他们的需求，以及当你们的用户与你们的产品交互时你应该考虑什么。

和用户坐下来

我在芬兰一家保险公司供职，在信息技术债权解决方案单位里做一名测试专家。我们开发应用程序（一些是对电脑，其他是对网页的）是为了处理我们的客户汇报的索赔。这包括了我们的内部债权顾问使用的为客户服务的应用程序。

这些内部应用程序是当我坐在我们的债权顾问旁时我有能力去观察的东西。顾问使用这些应用程序，比如，当他们尝试找到关于客户的信息或者当他们正代表客户注册一份索赔。当他们日常基本使用的一些应用程序被我们单位开发时，很多已被其他部门或

者外包开发了。

除了测试者以外，我们的开发者、经理和团队领导者们都被鼓励至少 1 年 1 次坐到我们的用户身边。在过去的六个月里我已经 3 次忘了那样做。对计划一个坐在一个债权顾问身边的会议是很容易的。如果他们和我在同一座城市里，这个会议在第二天能被安排出来。我通常观察一个顾问的工作将近 2 个小时。我发现这是一段很长的时间，因为专注于更长的时间和保持这种观察的能力是很累人的，至少在没有休息的情况下是如此。

因为债权顾问主要的一部分时间花到了解答从客户打来的电话，我也戴上了耳机这样我能听到谈话（虽然不是我自己发表评论，但我被静音）并能更好地理解正发生了什么。(专业的保密性约束了我，所以任何客户相关的信息都不能写下或者在会议外讨论。)在电话期间，我经常问顾问以理清问题并听他们如何描述他们用我们的应用程序所面对的挑战。

开始观察

当我坐在债权顾问身边时，我立即开始观察。

首先我注意到他们的工作环境。在其中的一次会议中，我发现用户有两个相当小的外部监控——可能 17 英寸。但是我注意到她的同事只有一个外部监控器，27 英寸的。这对我们后来开发的一款应用程序来说是有用的信息因为我知道去付出额外的注意力去以不同屏幕尺寸测试它。

我还注意到当用户从一个应用程序跳转到另一个时他们做了什么。有一个用户在使用应用程序期间复制并粘贴信息的会话。粘贴的信息在以一种不被欢迎的形式，但是我感兴趣的事实上是用户在使用另一个应用程序所需要的信息。这种常识能在即将到来的项目里帮助我们，我们能构建一个应用程序去代替用户拷贝信息的那个应用。我们至少能尝试做一份比它现在更简单的信息的备份。

甚至即使你能在你看到的你的用户正做什么的基础上改进你的产品，对于他们所有的问题不能被你的应用程序解决是个好机会。观察他们使用的其他应用程序或工具来解决他们的问题是很重要的。比如说，我观察到我们的其中一位用户用一个网页计算城市之间的距离。这不是我们立即要做的一些事，但是它能有助于当我们进一步开发我们的应用程序或者用新品代替它们。如果需求只反应一个应用程序的已存在的特性，这些类型的问题会很快被遗忘掉。

与用户交互

除了当债权顾问处理来自用户电话时观察他们，我也在每一个电话后询问了要澄清的问题。假如有关于债权顾问受挫或者疑惑的，当电话过后立即知道是有用的。对他们来说是什么使服务客户变得困难呢？什么使它变得容易些？

而且它不仅是关于学习债权顾问如何与我们的系统交互；还能感同身受。当你开始了解你的用户作为人类，他们更多将“他们”替换为“我们”。出于这些原因，我也会关注在他们办公桌上的个人物品，有时还会加入一些非正式的话题讨论中。

通过听电话呼叫顾客开始使我们的债权顾问也同感地受教，因为我更好地理解当他们使用我们的应用程序所面对的压力。顾问正在一次意外中或者其他压力环境中讨论，要求产品知识的同时还有从债权顾问那的精神投入。这是当我测试时经常忽略的，因为我已经只享受关注在测试应用程序上。

做记录

除非你有超常的记忆力，你将很可能在观察和用户交互时，从做记录中获益。

我个人喜欢在我的专业定做绑带本上写下笔记。一本实体笔记本给了我自由不仅快速地写或画，取决于我想要让什么成文档。基于我自己的经验，我也感受到一本实体笔记本不会创造我关于我们的会议以外的其他事物的印象。假如我能在一个笔记本电脑上写笔记，会有一种让债权顾问以为我在做其他事情的危险。无论如何，我知道了很多人喜欢在电脑上做笔记，所以如果你选择了那样做，我想他可能有助于清晰地对用户提起你正在做笔记。

你记下来的内容是一些你需要自己指出的，但是有一些事情我代表性地做记录:

- 什么使我们的用户失望
- 我们的业务领域我所学到的东西
- 电话是关于什么内容的，这样我能更容易地作为整体回忆起那个会议
- 顾问的课桌上有什么（监控器，笔记本，计算器等等）
- 改进的建议

启动

假如你受鼓舞了并想要坐在你的产品使用者的旁边，你将从哪开始记录呢？

首先，当然了，你需要找到一个使用者。对于我来说很简单因为我们应用程序的很多使用者和我一样在相同的公司里工作并且我们有坐在我们的用户身边的文化，但是假如你没有这种文化，假如他们允许你在他们使用你的产品时去观察他们你仍然可询问一位内部用户。人们经常强调有些人想要听他们并且给他们机会去分享他们的经验。

在没有计划用于内部用户的应用程序例子里，它要求很多创造性的。两年以前我测试一个全球性商业网站。我们想要从不太熟悉这个网站的人们那儿得到反馈，所以我们决定开始群众外包测试。我发送一封邮件给 15 人，这些人都涉及这个产品但是他们自己没有使用它，要求他们帮忙反馈——而且我提出可得到零食。

大约 10 人到达了一个探索性测试会议，他们一对一在我给他们的高级别目标的基础上测试网站。选择不熟悉网站的人证实了是个好的策略，因为他们做了大量的观察和改进的建议。

不论你从内部还是外部得到反馈，假如在你观察的现状中用户正使用你的产品将会尽可能地扣紧真实的每天用户典型经历的状态、记得更有价值的信息会很重要。

当你努力地观察一个用户并得到关于产品的新信息，与你的团队分享那些信息。这为我们提供了一个机会，让我们更好地了解我们的应用程序拥有的实际影响。

❖ 拓展学习

■变身全能人才，进阶全栈测试开发工程师：http://www.atstudy.com/classroom/5/introduction

成功的自动化测试实施的 5 大支柱

◆译者：枫　叶

摘要:
谈到什么能组成一次自动化测试的"恰当实施"，经常会关注你需要用的工具，但是那仅仅是一部分。巴斯 迪杰斯特拉详细说明了你需要考虑的其他四件事，它们如何有助于你的自动化测试的成功，以及未关注它们任一件事会存在适当的风险。

对组织机构来讲，指望快速交付质量，执行自动化测试是软件开发周期中一个重要组成部分。测试自动化，无论如何，当恰当实施时只能成功。论及什么组成测试自动化的“恰当实施”经常关注于任务必须使用的什么工具，或者最好（假如甚至有这样一个事物）或者最高效的为给予的任务使用特定工具的方法。

依我看来，虽然，使用的工具只是整体自动化测试的一个部分。任何成功的自动化测试的实施是有五个独特的部分组成的。

在这篇文章，我们看一看这些部分的每一个，它们如何致力于你的测试自动化实施的成功，以及关联到不能适当关注它们中任一件的风险。

一、测试自动化工具

虽然不是在成功的测试自动化实施中充当角色的唯一因素，工具显然在你的自动化努力整体结果中起了影响。选择一款工具与你测试的应用程序不兼容的工具，或者一个不适合你的自动化团队的技能的工具，将导致不太理想的结果。

然而甚至比选择工具更重要的是，自问什么正是你的自动化测试想要覆盖的，然后再决定更高效的获取那个结果的方法。一个要被问到的问题的典型例子是在什么级别上一个特定的功能模块或者业务逻辑需要被确认。

你想要确认你的用户能打开你的网店，搜索一个特定的产品，随后加入购物车并支付一个订单吗？你可能将想要使用端到端的用户接口驱动测试去检查这个。假如你正要

确认一块逻辑的正确性以决定是否一个用户被允许去购买一个特定的物品（比如由于国家或国家层面的规定），然后你将可能在较低的级别上编写测试，将其连接到你的应用程序中，比如一个应用程序接口或者甚至简单的代码分类。这构成了一个不同的测试范围和方法，因此需要一个不同的工具。

简而言之，确保你首先知道花时间在如何达到预期结果之前你的自动化测试需要验证什么。记住在使你的工具做它并不被设计的事上有一个重大风险。

二、测试数据

任何严肃的测试自动化解决方案的另一个重要的因素是管理测试数据的方法。测试范围越广，越重要，而且也要求更高的，测试数据管理来了。

当在单元测试里你能远离模拟所有你的测试依赖的数据，当你开始在集成上工作时或者端到端测试，你将需要在测试下呈现你应用程序里特定的数据。并且，为了使事情更加复杂，你将经常在处于试验阶段同时在某一个特定的国家里要其他系统里的数据以与你的连通的应用程序交互。

有一些在这些测试类型中处理测试数据的方法:

- 在测试开始阶段创造要求的测试数据
- 在开始测试前为了存在的测试数据去查询系统
- 开始测试执行前在测试阶段初始化你的应用程序数据库

这些方法每一个都有它的潜在危险:

- 在测试的开始阶段创建测试数据增加测试执行时间，在测试本身甚至开始启动前增加失败的风险，并且导致很多无用的测试数据假如没有合适的数据清理程序。
- 当你在开始测试前为存在的测试数据去查询系统时，你启动了偶然使用非法测试数据的风险，或者没有有着合适的优先级出现在系统里的测试数据。
- 在测试执行前初始化数据库让你能够管理和保持最新的数据库快照——那就是，如果允许你首先执行一个数据库恢复过程。

注意没有合适的为集成和端到端测试的测试数据处理方法。无论如何，选择错误的程序，或者根本未能定位测试数据问题，将可能导致自动化测试解决方案变得不可重复利用，不可维护，或者不可测量。

三、测试环境

莫诺利斯正在快速地重蹈恐龙的覆辙。现代信息技术系统由许多相互连通的组件，服务，和为传递业务价值而工作一起的应用程序组成。对测试目的，无论如何，这通常不是好消息：必须管理和依赖相关项的可用性，尤其是那些在你的控制循环外的那些，对于你的集成和端到端测试，可能会导致大量开销，挫败，和测试时间的延迟。当你想要创造和使用自动测试作为你的测试方法的一部分时可靠性和易管理的测试环境仍然是关键。

减少失败或者不存在测试环境风险的一个办法是使用仿真技术比如桩，模拟，和服务虚拟化以复制在你的测试环境下关键性的而难获取的独立性的行为。做模拟真实的独立性行为的仿真足够完成你想要执行的测试用例，还能大大提升你的自动化测试速度——因此也提升你的开发工作力度。

此外，当虚拟环境被恰当地创建（比如，通过杠杆作用集装箱化）时，重建一个相同测试环境的新实例，完成相同的测试数据和其他特性，使它可能从一个自动化移向确实的持续性测试，假如你正要适应持续性的传递时这作为一个越灵活的方法且更好地对市场需求增长的反应继而成为先决条件。

四、报告

自动化测试执行作为结果生成的报告，应该成为任何可靠的测试自动化方法的重要的一部分。创造好的测试结果报告经常被忽略，然而，在任何测试自动化项目里保存任务，这是一个潜在的时间（和生命）。好的报告远超乎展示测试运行的数量，通过的，和失败的，虽然只有它也比什么都没有好。

为了使一份测试执行报告变得确实有价值，需要被执行的测试（注意为了任何好的报告清晰的命名你的测试并且不含糊的格式是基础的）是什么可视化并且不仅是结果是什么（通过或失败），而且万一测试失败一些导致错误的东西在哪里，尽可能精确地详细说明。

这与提供信息过载不同，只需将任何东西复制到你的测试报告中——这将不必要地延迟测试失败的根本原因。一份好的报告展示了错误的一些东西，在测试错误发生处（在什么步骤上），错误信息是什么（取决于你的报告用户，这就像堆栈跟踪一样简单，但是其他案例里你可能需要提供也可被非技术人员读的错误的信息），以及在测试下应用程序

的状态是在失败的时候（比如说，为用户接口驱动测试使用一个截图）。

注意，良好的报告策略可能涉及在每次测试运行时创建多个报告。假如你的测试是一个可持续集成构建管道的一部分，你可能想创建低层次的报告，可以由你的构建引擎解释，以决定是否继续构建。但是你也可能想要以 HTML 格式创建一个可读性的报告，按测试目的的原文描述完成，在失败的测试情况下还有人类可读的信息和屏幕截图。它都取决于用户。

五、技能

最后，最重要的是为创建一个强有力且有效的测试自动化解决方案而设计的难题是负责实施它的人。没有熟练的自动化顾问、架构师、工程师和开发人员关注本篇文章中提到的所有测试自动化的其他方面，你的测试可能很快就会结束。

你的测试自动化团队应该在测试领域都是熟练的，因此他们能回答为什么测试自动化将首先是一个合适的解决方案，同时什么测试要被自动化；并且对软件开发熟悉，这意味着他们知道如何去创建一个既强大又可维护的测试自动化实现。这不意味着你的测试自动化每一个成员需要在这两个领域都很熟练，但是作为一个整体，为了软件交付你的团队需要这两者的达到平衡。

把它所有的放在一起

一个好的测试自动化解决方案需要关注更多不仅仅是驱动测试的工具。为了自动化真正的成功，你需要想办法在你的测试数据策略上，在你如何管理你的测试环境上，并且在你通知你的用户关于你自动化的测试执行结果的方式上。最重要的是，无论如何，它是关于创建一个知道如何做以上所有的团队。

Python 自动化测试应用—Python 调用安卓 adb 命令（上篇）

◆ 作者：lamecho 辣么丑

1.1、概要

今天是《Python 自动化测试应用》的第十篇，本篇文章为什么会讲 Python 调用执行 adb 命令呢？因为在 Web 的自动化测试框架 Pyswat 完不成后，就开始编写 Android APP 的测试框架了，目前 Pyapp 的框架也基本功能都完成，所以借此机会分享一下，Python 怎么玩转 adb 命令。

1.2、Pyapp 到底用的是 Appium 还是 adb?

使用过 Pyswat 框架的同学可能知道，Pyswat 是底层调用的 Selenium 那一套，而 Pyapp 实现完全和 Appium 没有一点关系。可能有同学就会问了，为什么我会这样去做？在着手设计 Pyapp 框架之初，确实也纠结了，考虑了几天时间，我到底是用 Appium 去实现自动化框架，还是 adb 命令去实现。最后还是确定利用 adb 命令来作为操控 app 的基础方式实现 Pyapp 的自动化测试框架。接下来我先说明这样做的几点原因:

1、由于 Pyapp 同样是采用录制，回放的模式（与 Pyswat 框架模式一致）。那么录制的话就要考虑录制的实现，一方面是核实触发录制，也就是什么时候知道测试人员点击了 APP 界面，另一方面要考虑测试人员点击的是当前 APP 界面的哪个元素。解决了这两个问题基本上录制就没问题了。那么考虑到这两个因素，我们来看看 Appium 能做到监控到用户的操作吗，应该没有因为我在 Pyswat 里是利用了 Python 写的钩子程序监控电脑的鼠标和键盘的操作，从而达到监控用户输入的目的，而在手机上是不能按照写钩子去监控到手机的。既然 Appium 不行，我们转换目标看看 adb 可以吗，它有对应的命令吗？使用过 adb 命令的同学应该知道可以通过 adb shell input tap/swipe/text 命令去做点击，划屏，

输入等操作，那么反过来 adb 有没有获取手机操作事件的命令呢，大家可能会想到 logcat 日志可以记录手机的操作日志，但是其实 adb 有一个 adb shell getevent 命令可以获取到当前屏幕点击事件及坐标，所以能够获取到这个重要的信息我们的录制功能基本上实现起来就没问题了。

2、第二个没有选择 Appium 的原因是，大家知道 Python 脚本如果搭配 Appium 的话，你需要首先把 Appium 的使用环境搭建起来，这一步对于新手来说或者是对于第一次使用 Pyapp 框架的同学来说非常不友好（因为 Pyswat 在对外发布后，发现很多同学在环境搭建上非常不熟练，产生过很多问题，导致无法正常使用 Pyswat 框架），而且每次运行脚本都要启动 Appium 的客户端，非常耗时，如果程序出问题还需要反复的启动关闭服务才行，无形中增加的操作成本，所以这也是抛弃 Appium 作为框架底层的原因。反过来看 adb 是 android 的原生命令，你只要搭好 android sdk 就可以使用了，搭建步骤也很简单。

3、最后 adb 命令由于是 android 的原生操作命令，支持实现的功能非常多。这里举几个 Pyapp 里实现的功能例子：获取，修改手机当前使用的输入法（adb shell ime list），获取当前手机界面的活动 activity（adb shell dumpsys activity activities）,安装，卸载，启动 app，点击，划屏，长按，硬件输入，截屏等。

所以最后总结一下，这里只是说明 adb 命令更适合我开发 Pyapp 测试框架，而不是说 Appium 不好。针对只是单纯的使用 Python 开发自动化脚本的同学来说，Appium 更为简单些。

1.3、Python 如何调用 adb 命令

Python 中执行 cmd 命令可以用到 os 和 subprocess 两个模块。区别在于 os 是阻塞式的，subprocess 是非阻塞式的，所以我们使用 subprocess 是比较适合的。接下来我先举一个查询连接设备的命令来看看 Python 中怎么样的写法。用到的命令为 adb devices。

```
import subprocess
order='adb devices'   #获取连接设备
pi= subprocess.Popen(order,shell=True,stdout=subprocess.PIPE)
print pi.stdout.read() #打印结果
```

```
List of devices attached
4d0041b1be98b01f    device
58899a5f    device
0110c90a    device


Process finished with exit code 0
```

实际打印结果，可以看到当前电脑连接了三台设备。这里需要再说明一下 adb devices 命令的结果返回是一次性的，所以我们用 read 方法读取数据是没有问题的，然而 adb 命令里还有一些是实时返回结果的，比如输出手机日志的命令 logcat，结果会不断的打印出来当前的设备操作日志信息内容，这种类型的命令我们在 Python 中如果需要获取打印结果，如果还是用 read 方法的话，等待结果的返回时间会非常长，这里我们就要换一种方法读取结果，写法如下。

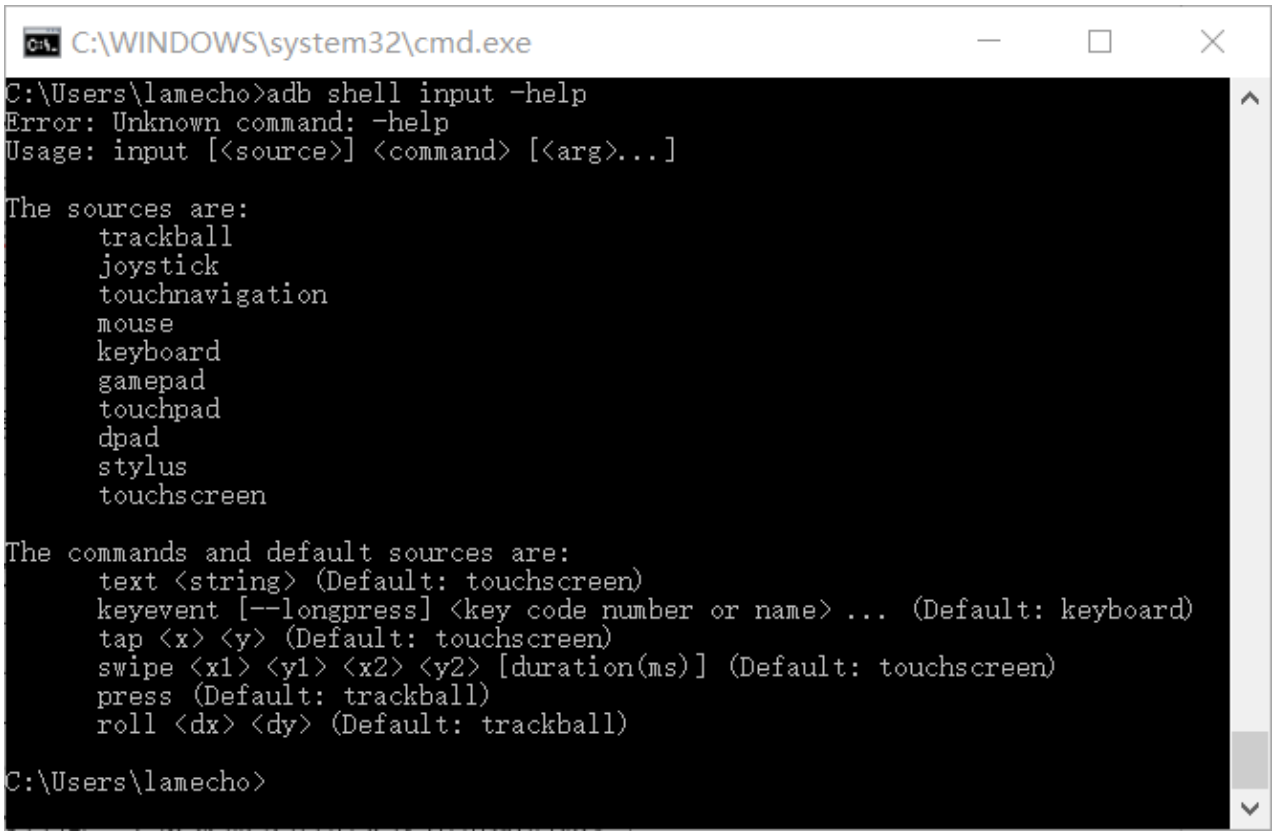
```
import subprocess
order='adb logcat'
pi= subprocess.Popen(order,shell=True,stdout=subprocess.PIPE)
for i in iter(pi.stdout.readline,'b'):
print i
```

```
D/dalvikvm(20119): GetMethodID: not returning static method Landroid/text/util/Linkify;.applyLink (Ljava/lang/String;IILandroid/t
D/dalvikvm(20119): GetMethodID: not returning static method Landroid/text/util/Linkify;.gatherTelLinks (Ljava/util/ArrayList;Land
D/dalvikvm(20119): GetMethodID: not returning static method Landroid/text/util/Linkify;.gatherLinks (Ljava/util/ArrayList;Landroi
 Landroid/text/util/Linkify$MatchFilter;Landroid/text/util/Linkify$TransformFilter;)V
D/dalvikvm(20119): GetMethodID: not returning static method Landroid/text/util/Linkify;.addLinks (Landroid/text/Spannable;I)Z
D/dalvikvm( 3705): GC_CONCURRENT freed 1925K, 45% free 13910K/25144K, paused 4ms+4ms, total 40ms
E/cutils  ( 2535): Failed to mkdirat(/storage/extSdCard/Android): Read-only file system
W/ContextImpl(20119): Failed to ensure directory: /storage/extSdCard/Android/data/com.miui.mipub/files
W/Vold    ( 2535): Returning OperationFailed - no handler for errno 30
E/cutils  ( 2535): Failed to mkdirat(/storage/extSdCard/Android): Read-only file system
W/ContextImpl(20119): Failed to ensure directory: /storage/extSdCard/Android/data/com.miui.mipub/cache
```

这样的打印效果，如同 cmd 里操作一致，实时的打印出日志信息。这里我们就用到了 readline 方法，其实这种写法类似我们读取文件，单行读取和全部内容读取。因为目前 Pyapp 的框架已经基本写完了，所以有了写这篇文章的想法，分享一些 Python 在处理 adb 命令上的一些心得，就目前来看 Python 在调用 adb 命令上区别主要就是这两点，最终目

的是我们找到需要的功能命令获取结果数据，然后再去通过 Python 处理这些返回数据，实现自动化测试的目的。大家要用好 adb 命令，还要注意一点的是每条命令的各种参数的搭配使用，比如 Pyapp 的实现是支持多设备连接的，那么我们在针对某个手机进行 adb 命令操作时，就需要带上-s 加设备号，表示操作的具体设备否则命令会报错。比如我们针对一个设备去进行点击操作，命令的写法应该是这样：adb -s 49dsd4554wdsa shell input tap 600 900，其中‘49dsd4554wdsa’是设备号，‘600 900’点击屏幕坐标。所以可以看到增加了-s 之后就可以很方便的同时操作多台设备。

至于 adb 的相关的命令，本篇文章不会再过多的介绍，因为网上有很多相关的帖子博客都有介绍 adb 命令的，大家可以自己查阅然后通过 Python 去操作实现。而后续我还要将花一篇的文章的时间去介绍一下我在编写 Pyapp 框架时，遇到的一些 adb 命令需要注意的地方。可能大家在网上查阅一些 adb 命令后，尝试这自己去执行一下，发现“诶，怎么不起作用？”，这里可能就要注意一些命令的参数搭配或者是自己所使用的设备具体的情况来定，因为目前市面上的安卓手机设备种类非常多，需要做一些适配。这里如果大家在每条的命令后加一句 –h 或是-help，可以查询一下该条命令的相关使用帮助信息。比如我们在 cmd 中输入 adb shell input –help，会返回命令相关的使用方法，我这里大概讲解一下，大家遇到其他的命令也可以自己看明白每条命令的用法了。在 Usage 这行具体告诉我们 input 后面跟的写法，[]中括号里的内容表示可以带也可以不带，<>尖括号就是必须带上的内容。然后接下来给我们了 sources 的一些常见可用的用法，这里有看到 trackball（轨迹球），joystick（手柄），mouse（鼠标）等等，表示模拟输入的源硬件的意思，而最后是具体的命令，有 text（输入文本），keyevent（硬件操作），tap（点击）等等，这里还需要注意在具体的命令后还需要跟上具体的命令参数，怎么理解呢？比如我们是要操作 tap 命令，点击屏幕，那么我们肯定要告诉设备我们要点击的位置（坐标），所以在 tap 后把我们点击的坐标 x，y 传进去。



最后感谢大家耐心读完本篇文章，有关 adb 命令本篇只是做了一个抛砖引玉的作用，更多的还是需要大家自己多去实践不同的命令，当然在命令的选择使用上我们要有的放矢，也就是要为我们测试所用，在下一篇文章我将更多时结合具体的 adb 命令来做讲解，也会配合实际测试问题来介绍给大家如何使用 adb 命令。

❖ 拓展学习

■快速突击 Python，玩转自动化>>http://www.atstudy.com/course/156

测试女巫之石头变宝石篇之五

◆作者：王平平

一、前言：

测试女巫又开始将石头变宝石了，为什么没有延续前几期的主题：python 自动化呢？因为测试女巫近期在学习前端开发框架：Angular，这个酸爽，谁学谁知道~~~，让我先哭会~~~当然并不是女巫转行做前端开发了，而是我们后续需要搭建自己公司的测试 Cloud，需要前端开发的人员，女巫就要带着团队的兄弟姐妹将这个前端开发搞定，另外，女巫换老板了，老板说他是从疯人院跑出来的，跟着一个近乎“癫狂”的主管，这个酸爽，谁跟，谁知道~~~，让我先哭会~~~老板说需要对我们的自动化开发的各个工具做一个架构升级，所以注定接下来，女巫又会经历一番血风腥雨的折腾。嗯，擦干了眼泪，女巫相信，越折腾，功力就越强，所以尽管放马过来吧，姐怕过什么！？

所以有关自动化开发，有关搭建 Cloud 的前端开发，等女巫抗过这番折腾后再跟大家分享吧，女巫是所在公司的 6 sigma 培训讲师，所以女巫有很多使用 6 sigma 工具应用到工作上的例子，所以这期就继续石头变宝石吧！

这次我们学习新的“魔法”是什么呢？是如何使用统计学的知识去理清一个复杂的问题，进而形成一个测试流程，进而改进我们的测试工作。哦，是不是听得有点蒙圈，举例吧，例如我们测试一款 Router 产品，在测试 WIFI 时，会发现 WIFI 不稳定，从测试人员的直觉或者可以说女人的直觉，这个 Function 存在异常。但是软件工程师，会有一堆托辞：你所在的 WIFI 环境太脏了，是不是存在 Channel 冲突，即几台 Client 抢 channel 问题导致……其实他们的说法看起来发散，其实中心思想非常明确：测试环境有问题，测试手法有问题，就是我的产品没问题！

接下来就是没完没了的让测试人员做各种无效的实验，最后问题还是不了了之~~

但是我们学习了统计学的知识，我们测试人员可以自己设计科学的实验，然后用 Python 开发自动化脚本搜集数据，然后用 Minitab 的各个工具进行分析，产出专业的客观

的数据，让软件工程师无处可逃，哈哈，听起来是不是很爽，没错，女巫就是这样做的！结果是非常之爽！所以我们测试人员需要掌握多个工具，我们不仅会测试，会发现 bug，还会用统计学的方法设计科学的实验，然后用自动化工具进行测试，搜集数据，再用统计学的工具，分析这些数据，产出科学客观的结果。这还不是人才，那什么才是人才？!! 嗯，没错，测试女巫来了，带着魔法来了，带着魔法轻轻的来了，你学或不学，测试女巫就在这里，在这里等着你，成为做事深刻，走心的职场达人^_^。

此期学习的魔法就是如何应用已经学习过的统计学知识应用到我们的实际测试工作中！使用的统计学知识如下：标准差和平均值的概念（此观念在第 39 期杂志上做了详细说明），趋中定律（此概念在第 39 期杂志上做了详细说明），DOE 实验(此概念在第 37 期杂志是做了说明)的方法，箱形图，中位数检验

对于上述已经讲过的知识点，这里不再详细说明，亲如果忘记了，翻开对应期刊杂志看看吧，温故而知新，不亦乐乎嘛。

闲话不多说，我们开始正式学习吧！

二、6 sigma 新常用工具基础知识介绍

1、箱形图

"盒式图"或叫"盒须图""箱形图"boxplot（也称箱须图(Box-whiskerPlot）须图又称为箱形图，其绘制须使用常用的统计量，能提供有关数据位置和分散情况的关键信息，尤其在比较不同的母体数据时更可表现其差异

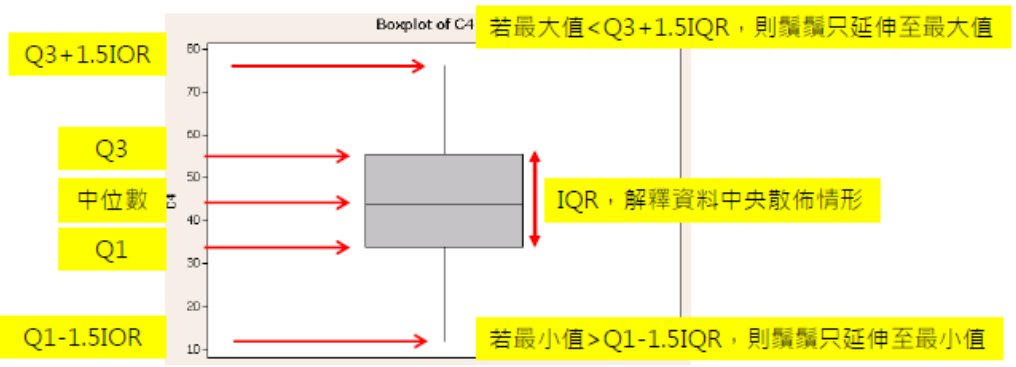
它提供有关数据的位置和分散的信息，在不同的组数据比对时更可表现差异

所有统计图形中，包含了最多种信息的图形

箱型图中间的『箱子』由 Q1 及 Q3 所组成，因此箱内包含了 50%的数值数据

其中 Median 描述了数据中心位置，而 IQR 解释了中央散布的情形。

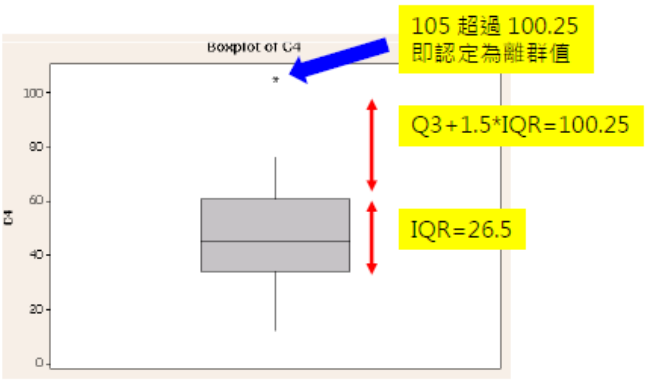
箱型图的『须』：由箱子延伸至 Q3 +1.5IQR 与 Q1-1.5IQR



离群值(outlier)：一组数据内，与其他观测值差异过大的值

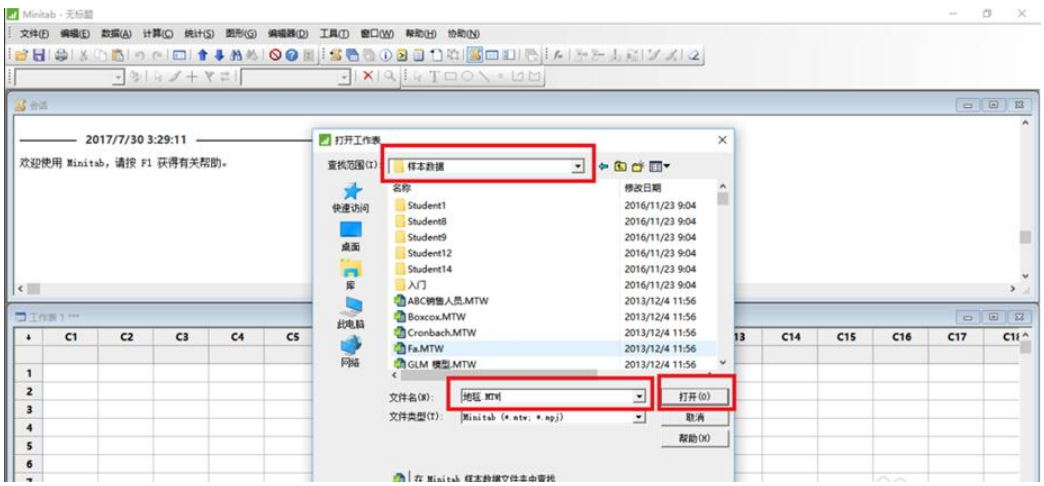
当须从箱子的边界向外延伸 1.5 倍 IQR 的距离，若观测值超出此范围，则视为离群值。

当观测值的最大值或最小值分别未超出 Q3 + 1.5IQR 与 Q1 -1.5IQR ，则箱型图的须仅延伸至极值(最大值或最小值)。

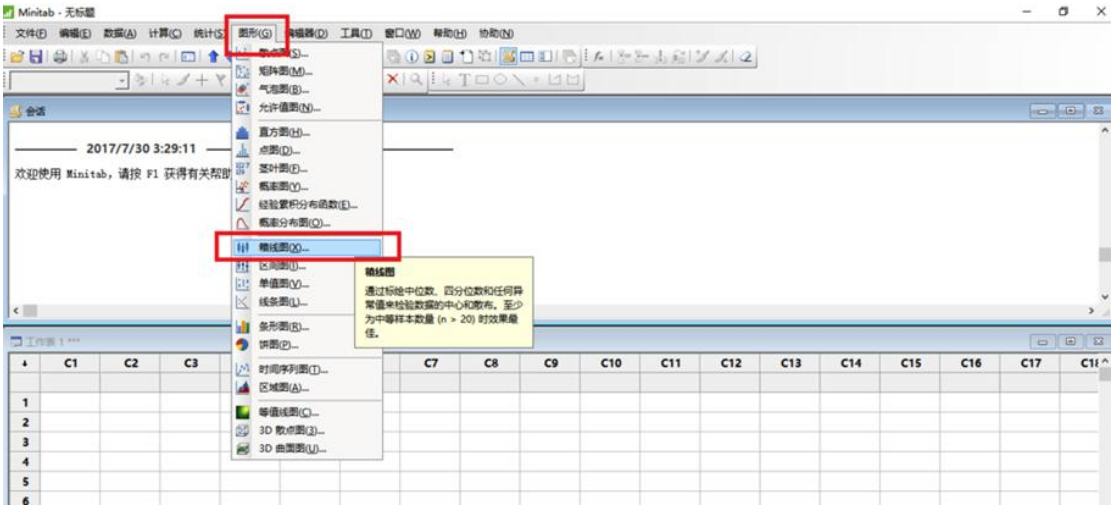


箱型图如何在 Minitab 上操作

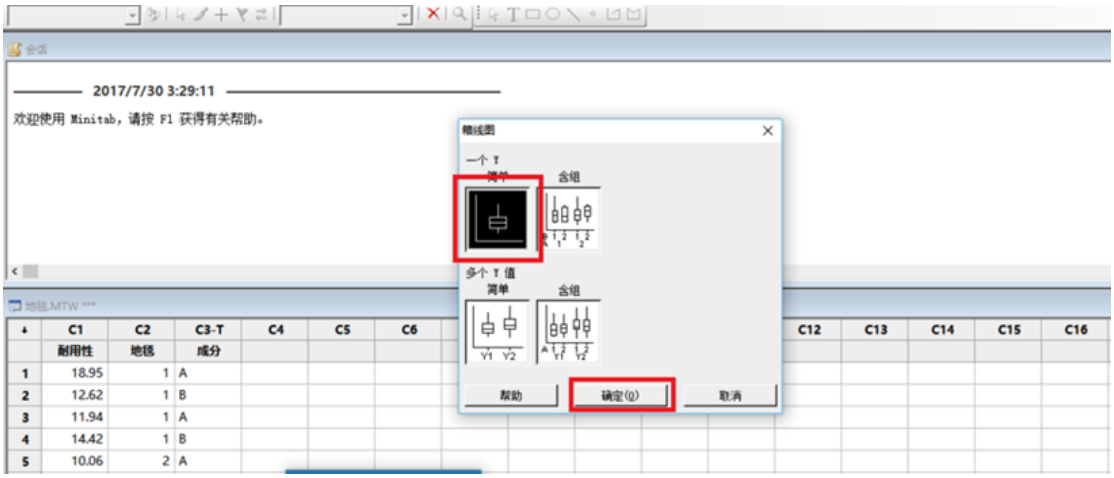
1) 打开样本数据



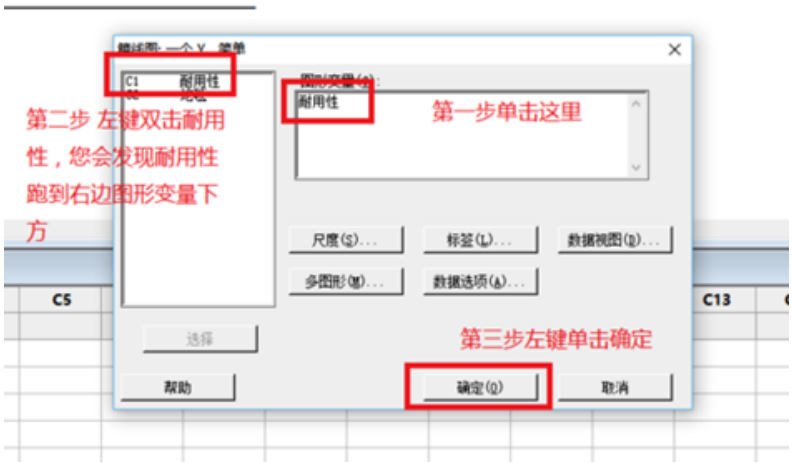
2) 选择图形，箱型图



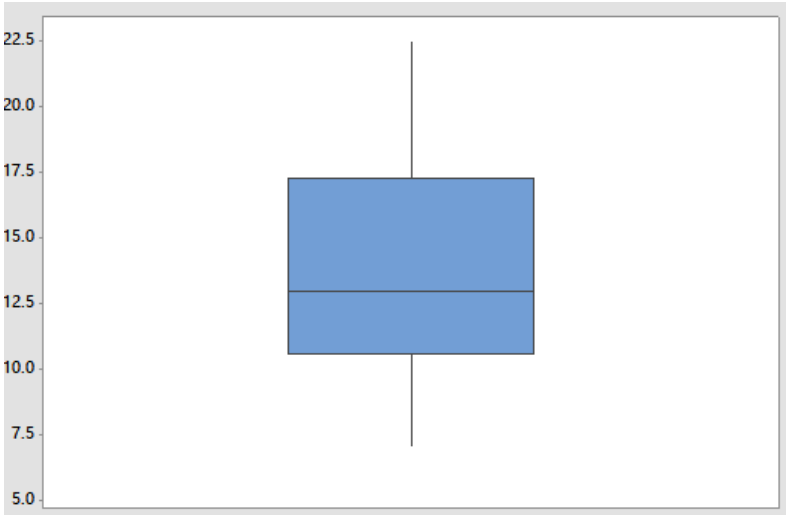
3) 根据你的需求选择画一个图还是多个图



4) 将需要的画图的变量选进来



5) 最后生成你想要的箱型图

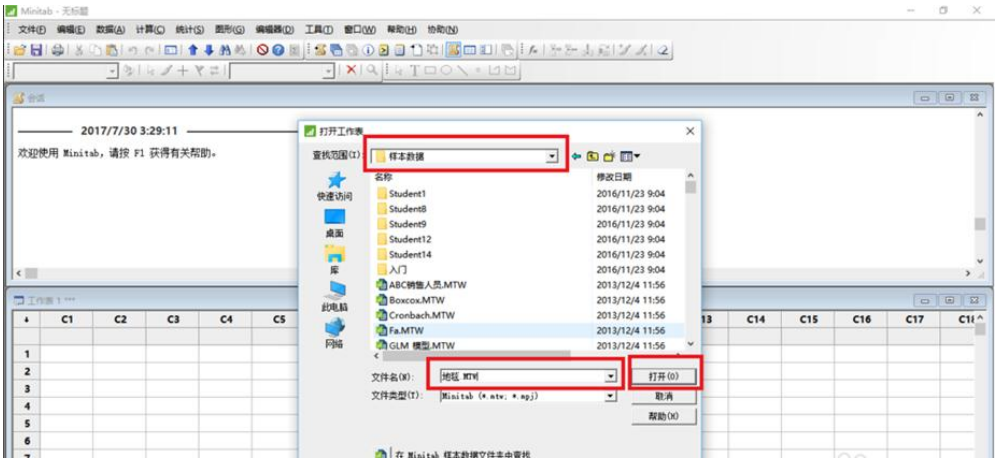


2、中位数检验

它是应用假设检验的理论，当待分析的数据为“非常态分布”时，会使用到这个工具，它的原理与待分析的数据为“常态分布”是完全相同的，这部分在女巫变宝石的其它系列有详细的说明，这里就不再赘述~

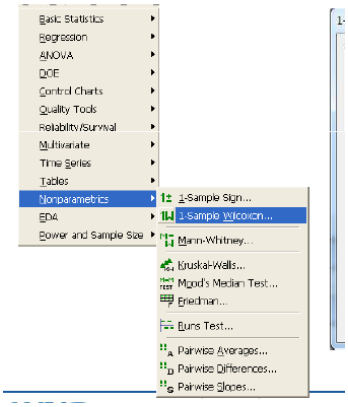
使用 Minitab 分析的方法:

1）打开样本数据

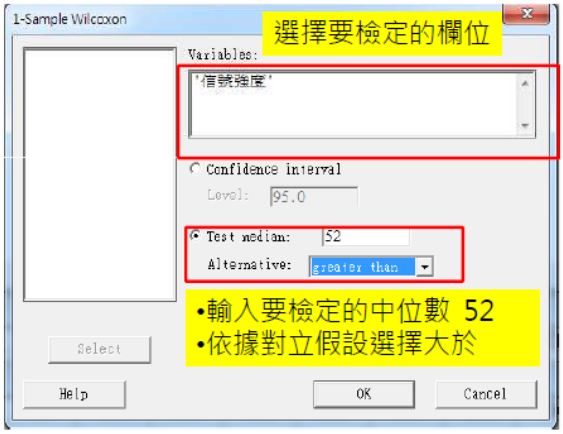


2）选择统计(Stat)->非参数(Nonparametric)->单样本威尔科克森检验(1 sample Wilcoxon)

1. 指令: Stat ➔ Nonparametric ➔ 1 sample Wilcoxon



3）选择要检验的变量以及中位数



4）对产出结果进行分析，得出结论

Wilcoxon Signed Rank Test: 信號強度

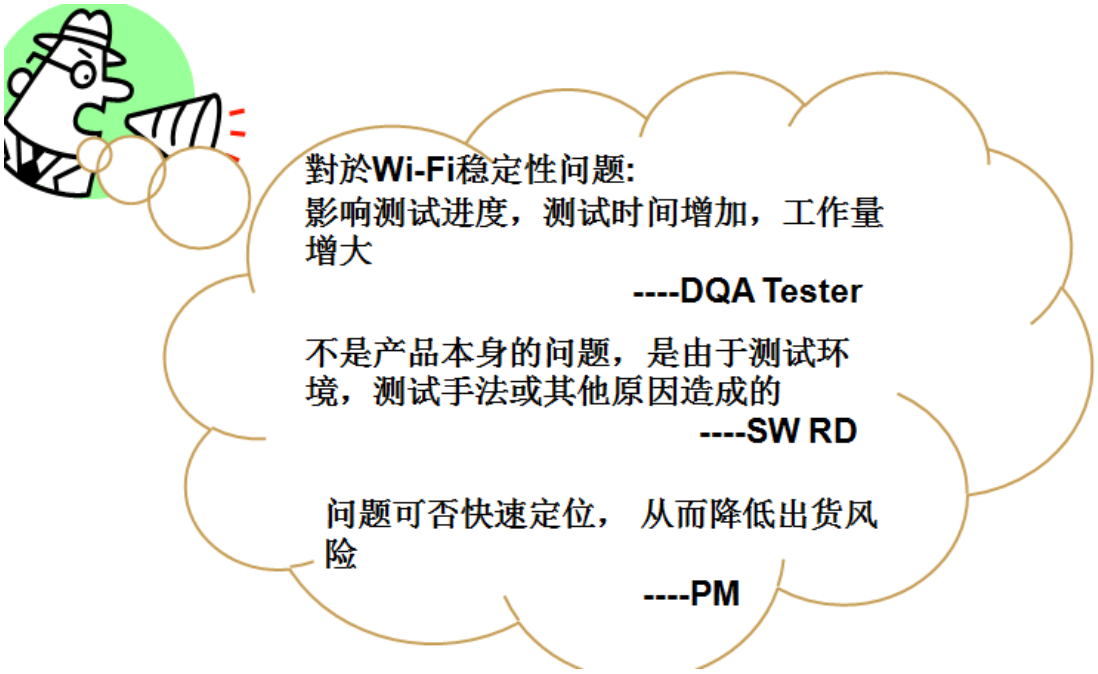
Test of median = 52.00 versus median > 52.00

	N	N for Test	Wilcoxon Statistic	P	Estimated Median
信號強度	24	19	60.0	0.923	51.50

三、应用到实际工作之如何分析 WIFI 稳定性的问题

对于生产终端的通讯公司在实际工作中会有这样的事情发生：在 router 的测试中，最让测试人员揪心的就是类似 WIFI 稳定性的问题，因为这类问题，争议很多，往往就是测试部门花费了很多人力物力，最后还是无法彻底、理清问题，下面这张图形象地说明了三个部门对于这类问题的不同的诉求，所以这类问题，我们需要结合统计学的知识进行设计科学的实验，并使用统计学的工具，分析实验的结果，尽快将这种有争议性的问题

解决，才是王道！

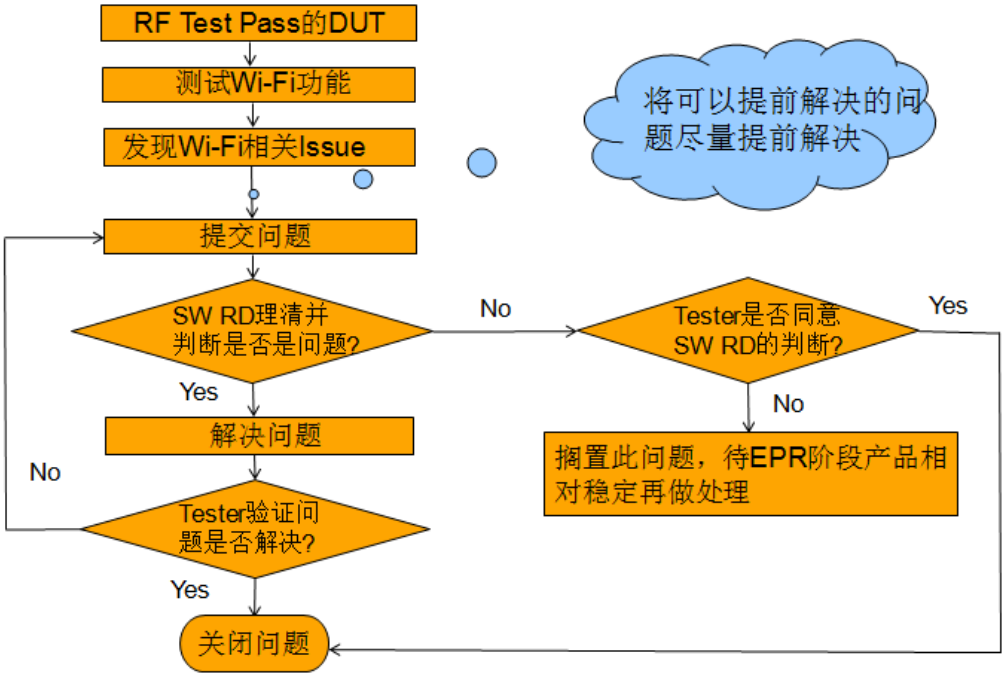


1、分析步骤

我们先看一下现有的工作流程是什么

黄色框框就是现有的工作流程，蓝色部分，就是我们希望解决的问题

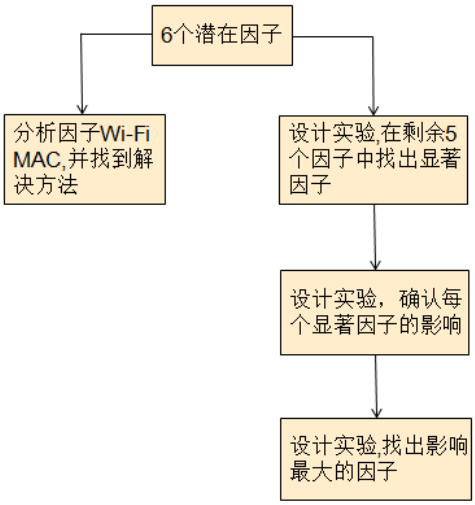
即希望我们在 bug 追踪系统上传 bug 前，引入一个机制，避免这个 bug 存在争议。



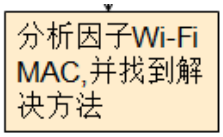
2）我们先把与影响 WIFI 稳定性的因子列出来，即共有 6 个可控因子，将这 6 个可

控因子分成两类，第一类是简单因子，即可以直接下对策的因子，另外一类是复杂因子，这些因子就要使用统计学的方法进行设计实验，分析数据，才能下结论的因子。

Factor	Control?
Wi-Fi MAC	可控
Channel Setting	可控
Wi-Fi Mode Setting	可控
PC	可控
WLAN Card	可控
Test environment(Shielding Room, Office)	可控



3）简单因子：分析因子 WIFI MAC,并直接找到解决方法



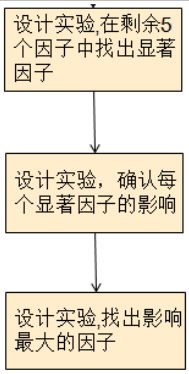
问题描述:

当拿到新的 DUT 时，有的 DUT 在工厂已经被烧录的 WIFI MAC 地址，有的未烧录，仍然是同一个预设值。若 WIFI MAC 相同，数据链路层通过 MAC 从一个节点传输数据到另一个节点时，可能会将数据传到非所要连接的 DUT，从而导致 WIFI 连接断开或连不上。

解决方法:

当拿到新的 DUT 时，先查看 WIFI MAC 地址，若相同则用命令手动修改，确保所有 DUT 的 WIFI MAC 不相同。

4）分析另外 5 个复杂因子



根据我们在实际测试时发现的 bug，我们针对这 5 个因子，设计了两个实验

Test	Purpose
Wi-Fi 连接/断开压力测试	因为测试中发现此操作较易造成Wi-Fi连不上的状况，为了理清此问题是DUT的问题还是由于环境导致的(软件RD认为是环境造成的，DQA认为是DUT的bug)
Wi-Fi长时间连接测试	模拟用户实际使用状况

5）实验 1：WIFI 连接/断开压力测试

Step1：修改 DUT WIFI Setting

Step2：DUT 的 WIFI 自动重启并自动再次连接 PC

重复操作 100 次，此实验用自动化执行

Result：成功连接次数

注意哦，这个实验可以结合我们的 Python，开发一个自动化脚本，修改 DUT WIFI Setting 需要用到 selenium 模块，判断是否连接成功，可以使用是否 Ping 通 DUT 的方式，这又会用到 Pywinauto 模块，所以千万不要让测试人员傻傻的执行这个无聊的测试，要运用我们的 Python 语言，开发一个测试工具。

6）WIFI 连接/断开压力测试搜集的数据如下：

注意哦，是使用自动化工具执行这些压力测试，我们才能得到这么多测试结果，如

果你不会开发自动化工具，这个实验真的会花费很多很多人力，我相信老板也不会同意的~

针对修改 WIFI Setting 我们根据平时测试经验，找出如下几个因子:

Wifi mode，不同的 channel，不同的 PC,不要的无线网卡，不用的测试环境。确认这些因子的不同组合是否会影响到 WIFI 的连接性能

PC	Wi-Fi Mode	Channel	WLAN Card	Environment	Connection Success
ACER	Single Mode	Auto	TP-Link	Office	100
DELL	Single Mode	Specified	ASUS	Shielding Room	100
ACER	Dual Mode	Specified	ASUS	Office	100
DELL	Single Mode	Auto	TP-Link	Office	100
ACER	Single Mode	Specified	ASUS	Shielding Room	100
DELL	Dual Mode	Auto	ASUS	Shielding Room	100
DELL	Single Mode	Auto	TP-Link	Shielding Room	100
DELL	Dual Mode	Auto	TP-Link	Shielding Room	100
DELL	Dual Mode	Specified	TP-Link	Shielding Room	100
ACER	Single Mode	Specified	ASUS	Office	100
ACER	Single Mode	Auto	TP-Link	Shielding Room	100
ACER	Single Mode	Auto	ASUS	Shielding Room	100
DELL	Dual Mode	Specified	TP-Link	Office	100
ACER	Dual Mode	Auto	ASUS	Shielding Room	100
ACER	Dual Mode	Specified	TP-Link	Shielding Room	100
ACER	Dual Mode	Auto	TP-Link	Office	100
DELL	Single Mode	Auto	ASUS	Office	100
DELL	Dual Mode	Auto	TP-Link	Office	100
ACER	Single Mode	Specified	TP-Link	Office	100
DELL	Dual Mode	Specified	ASUS	Shielding Room	100
ACER	Dual Mode	Auto	ASUS	Office	100
ACER	Single Mode	Auto	ASUS	Office	100
DELL	Dual Mode	Specified	ASUS	Office	100

各组连接成功率都是 100%，说明修改 WIFI Settings 不会影响 WIFI 的稳定性

7）实验二 WIFI 长时间连接测试

PC Ping DUT

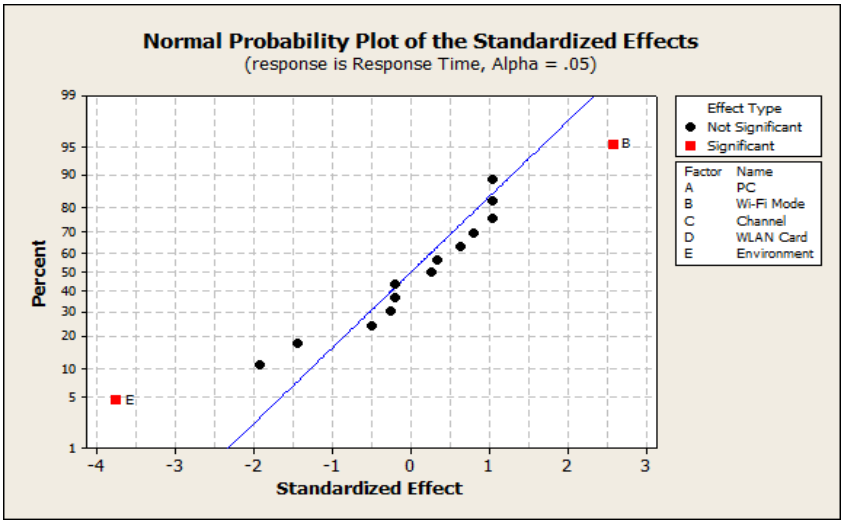
时长：下午 18:00~第二天上午 9:00

Result: response time 的平均数(单位 ms)

8）WIFI 长时间连接测试数据:

PC	Wi-Fi Mode	Channel	WLAN Card	Environment	Response Time
ACER	Dual Mode	Specified	ASUS	Office	25
ACER	Dual Mode	Specified	ASUS	Shielding Room	11
DELL	Dual Mode	Auto	TP-Link	Shielding Room	15
DELL	Dual Mode	Specified	ASUS	Office	14
DELL	Single Mode	Auto	TP-Link	Shielding Room	4
ACER	Single Mode	Auto	TP-Link	Shielding Room	4
ACER	Dual Mode	Auto	ASUS	Shielding Room	11
DELL	Single Mode	Specified	TP-Link	Office	17
ACER	Single Mode	Auto	ASUS	Office	6
DELL	Dual Mode	Auto	ASUS	Office	28
DELL	Dual Mode	Specified	ASUS	Shielding Room	12
DELL	Single Mode	Auto	TP-Link	Office	8
DELL	Single Mode	Specified	TP-Link	Shielding Room	5
ACER	Single Mode	Specified	TP-Link	Shielding Room	4
ACER	Single Mode	Specified	TP-Link	Office	14
DELL	Single Mode	Auto	ASUS	Shielding Room	4
DELL	Dual Mode	Auto	ASUS	Shielding Room	12
ACER	Dual Mode	Specified	TP-Link	Shielding Room	3
ACER	Dual Mode	Auto	ASUS	Office	17
DELL	Dual Mode	Specified	TP-Link	Office	28
ACER	Single Mode	Auto	ASUS	Shielding Room	3
ACER	Single Mode	Specified	ASUS	Shielding Room	12

对于实验 2 进行 DOE 分析，具体的如何操作 Minitab，进行 DOE 分析这里就不再赘述，因为我们在[第 37 期的杂志]上已经进行了详细的说明，如果有记不住的小伙伴，可以翻看 [37 期的杂志]。



Factorial Fit: Response Time versus PC, Wi-Fi Mode, ...

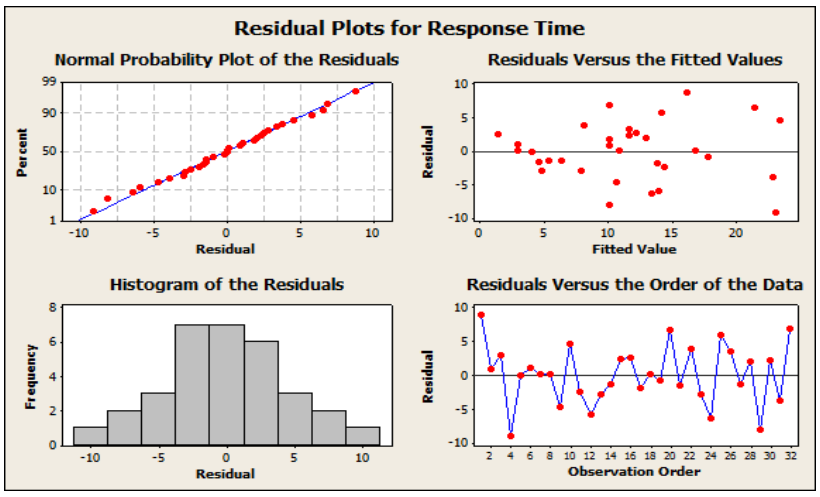
Estimated Effects and Coefficients for Response Time (coded units)

Term	Effect	Coef	SE Coef	T	P
Constant		11.594	1.059	10.95	0.000
PC	-4.062	-2.031	1.059	-1.92	0.073
Wi-Fi Mode	5.438	2.719	1.059	2.57	0.021
Channel	-1.062	-0.531	1.059	-0.50	0.623
WLAN Card	1.687	0.844	1.059	0.80	0.437
Environment	-7.938	-3.969	1.059	-3.75	0.002
PC*Wi-Fi Mode	-3.063	-1.531	1.059	-1.45	0.167
PC*Channel	0.688	0.344	1.059	0.32	0.750
PC*WLAN Card	2.187	1.094	1.059	1.03	0.317
PC*Environment	1.313	0.656	1.059	0.62	0.544
Wi-Fi Mode*Channel	2.188	1.094	1.059	1.03	0.317
Wi-Fi Mode*WLAN Card	2.188	1.094	1.059	1.03	0.317
Wi-Fi Mode*Environment	-0.437	-0.219	1.059	-0.21	0.839
Channel*WLAN Card	-0.562	-0.281	1.059	-0.27	0.794
Channel*Environment	-0.437	-0.219	1.059	-0.21	0.839
WLAN Card*Environment	0.562	0.281	1.059	0.27	0.794

S = 5.98827 R-Sq = 66.12% R-Sq(adj) = 34.37%

从上述两幅图可以看出

WIFI Mode 和 Environment 是显著因子

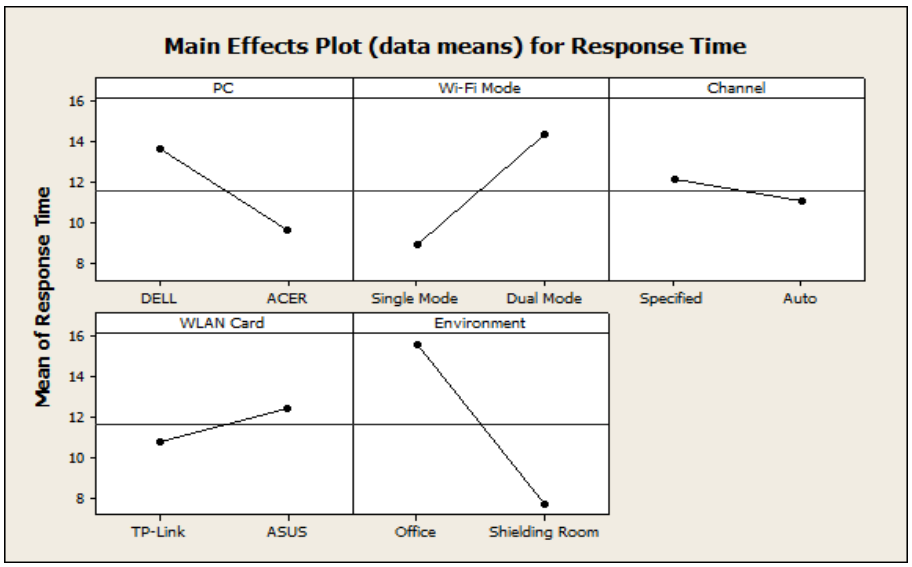


- ✓ 残值符合常态分布
- ✓ 残值符合常态分布和残差平均为 0
- ✓ 各组变异相同
- ✓ 残值相互独立且为随机分布

9) 传承上述实验得出两个显著因子：WIFI Mode 以及 Environment

首先分析显著因子 WIFI Mode:

WIFI 长时间连接实验主效应图



从上述图中可以看出：WIFI Mode 为 single mode 时 WIFI 比较稳定

另外一种实验，分别用三个项目进行 WIFI 最大连接数测试:

Project	Max Clients in Single Mode	Max Clients in Dual Mode	Max Clients in Single Mode
D35Z1	32	22	32
D35D1	32	24	32
WLZ2Q	32	15	32

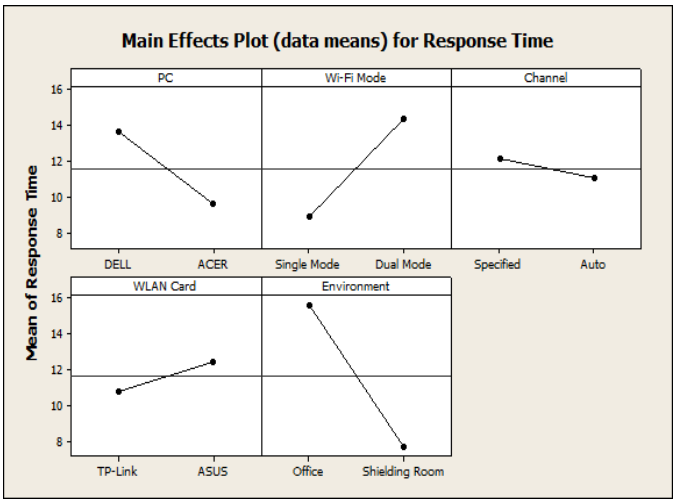
Dual Mode 时，DUT 的最大连接数无法达到 Spec 要求的 32。

对于 Dual Mode 时，WIFI 不稳定的问题，Qualcomm 已提供文档说明此问题确实存在。

10) 显著因子 Environment 分析

WIFI 长时间连接实验主效应图

从下面的图可以看出：Environment 为 shielding room 时 WIFI 比较稳定



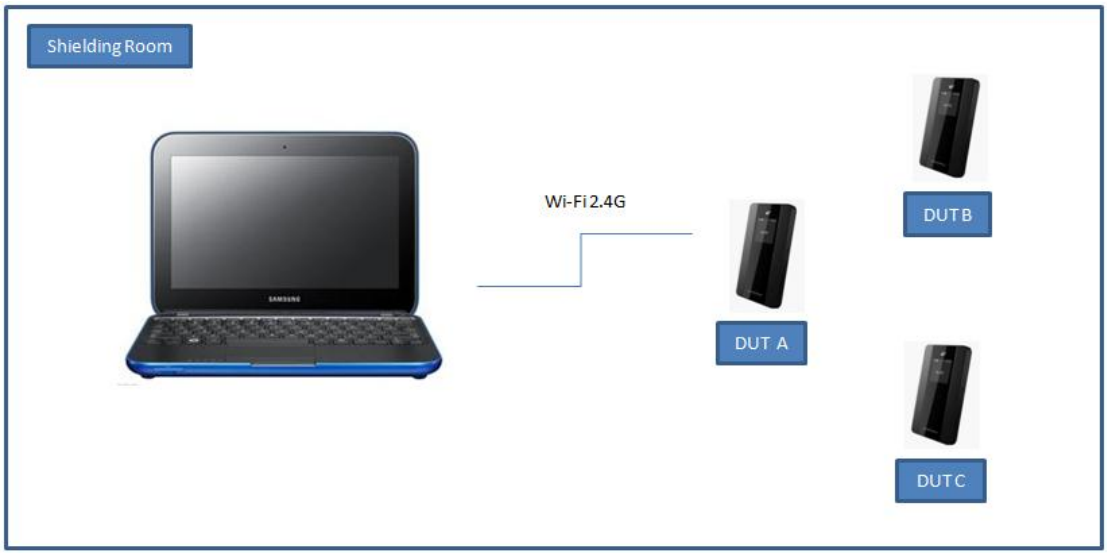
11) 继续设计实验验证什么样的环境是影响 WIFI 性能

验证多台 DUT 同时只开启 WIFI 2.4G 时，对 WIFI 2.4G 的稳定性是否有影响（同频干扰的影响)实验的环境架构:

PC 分别在不同条件下 Ping DUT A

时长：10mins

Result: response time 的平均数(单位 ms)



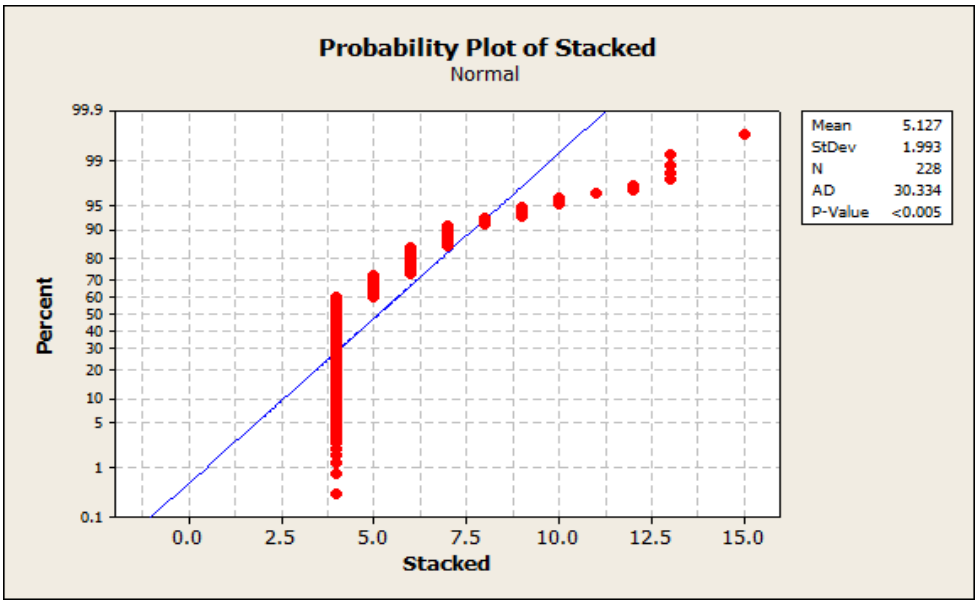
实验的 Test condition:

DUT(Single Mode)	DUT A Channel	DUT B Channel	DUT C Channel	Test condition
DUT A	6			Test condition1
DUT A,B	6	6		Test condition2
DUT A,B	6	4		Test condition3
DUT A,B,C	6	4	8	Test condition4
DUT A,B,C	6	6	8	Test condition5
DUT A,B,C	6	6	6	Test condition6

DUT A，B，C 都处于 WIFI single mode.

每个 Test Condition 测试 40 次，此实验用自动化执行

数据堆叠后做常态检定 ： P-Value<0.05 表示资料为非常态分布



使用 **Kruskal-Wallis** 中位数检定分析:

Kruskal-Wallis Test: Stacked versus Group

Kruskal-Wallis Test on Stacked

Group	N	Median	Ave Rank	Z
Test condition1	38	4.000	94.9	-2.00
Test condition2	38	4.000	111.9	-0.27
Test condition3	38	4.000	96.7	-1.83
Test condition4	38	4.000	97.2	-1.77
Test condition5	38	5.000	135.3	2.13
Test condition6	38	5.000	151.0	3.74
Overall	228		114.5	

H = 24.21 DF = 5 P = 0.000
H = 31.08 DF = 5 P = 0.000 (adjusted for ties)

P<0.05 表示 6 种不同的条件下 response time 值有差异，所以多台 DUT 同时只开启 WIFI 2.4G 时，对 WIFI 2.4G 的稳定性有影响，即同频干扰对 WIFI 2.4G 的稳定性有影响。

12) WIFI Mode 和 Environment 的影响力比较

实验设计:

验证在 WIFI Mode 和 Environment(同频干扰)两个因子中，哪个因子对于 WIFI 2.4G 的稳定性影响较大

说明:

当 SSID1 和 SSID2 设置为相同 WIFI Mode 时,对于其中一个 SSID 来说，相当于多了一个同频干扰源。

实验的环境架构:

PC 分别在不同条件下 Ping DUT A

时长：10mins

Result: response time 的平均数(单位 ms)

实验的 Test condition:

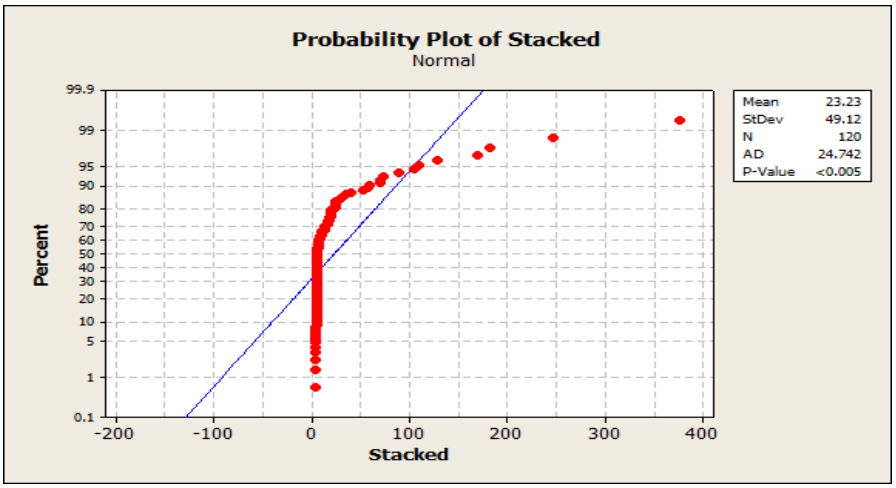
DUT	Wi-Fi Mode	DUT A Channel	DUT B Channel	DUT C Channel	Test condition
DUTA,B,C	SSID1: Wi-Fi 2.4G SSID2: Wi-Fi 5G	SSID1: 6 SSID2: Auto	SSID1: 6 SSID2: Auto	SSID1: 6 SSID2: Auto	Test condition1
DUTA,B,C	SSID1:Wi-Fi 2.4G SSID2:Wi-Fi 2.4G	SSID1: 6 SSID2: 6	SSID1: 6 SSID2: 6	SSID1: 6 SSID2: 6	Test condition2
DUTA,B,C	SSID1:Wi-Fi 2.4G SSID2:Wi-Fi 2.4G	SSID1: 6 SSID2: 6	SSID1: 6 SSID2: 6	SSID1: 4 SSID2: 4	Test condition3

DUT A，B，C 都开启 2 个 SSID

每个 Test Condition 测试 40 次，此实验用自动化执行

数据堆叠后做常态检定 P-Value<0.05 表示资料为非常态分布

P-Value<0.05 表示资料为非常态分布



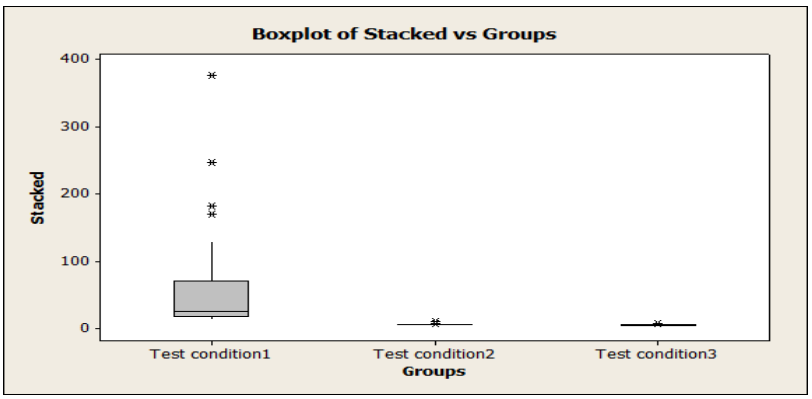
使用 Kruskal-Wallis 中位数检定分析:

P<0.05 表示 3 种不同的条件下 response time 值有差异，即 WIFI Mode 和 Environment 对 WIFI 2.4G 的稳定性影响有差异。

```
Kruskal-Wallis Test on Stacked

Groups              N  Median  Ave Rank      Z
Test condition1    40  25.000     100.5   8.91
Test condition2    40   5.000      47.8  -2.83
Test condition3    40   5.000      33.2  -6.08
Overall           120              60.5

H = 82.86  DF = 2  P = 0.000
H = 87.04  DF = 2  P = 0.000  (adjusted for ties)
```

13) WIFI Mode 和 Environment 的影响力比较

- Test condition1 的数据基本集中在 18~70 之间，还有些离群值
- Test condition2 的数据基本集中在 5~6 之间
- Test condition3 的数据基本集中在 4.25~5 之间



Test condition1，2，3 各个条件下 Response Time 的平均值

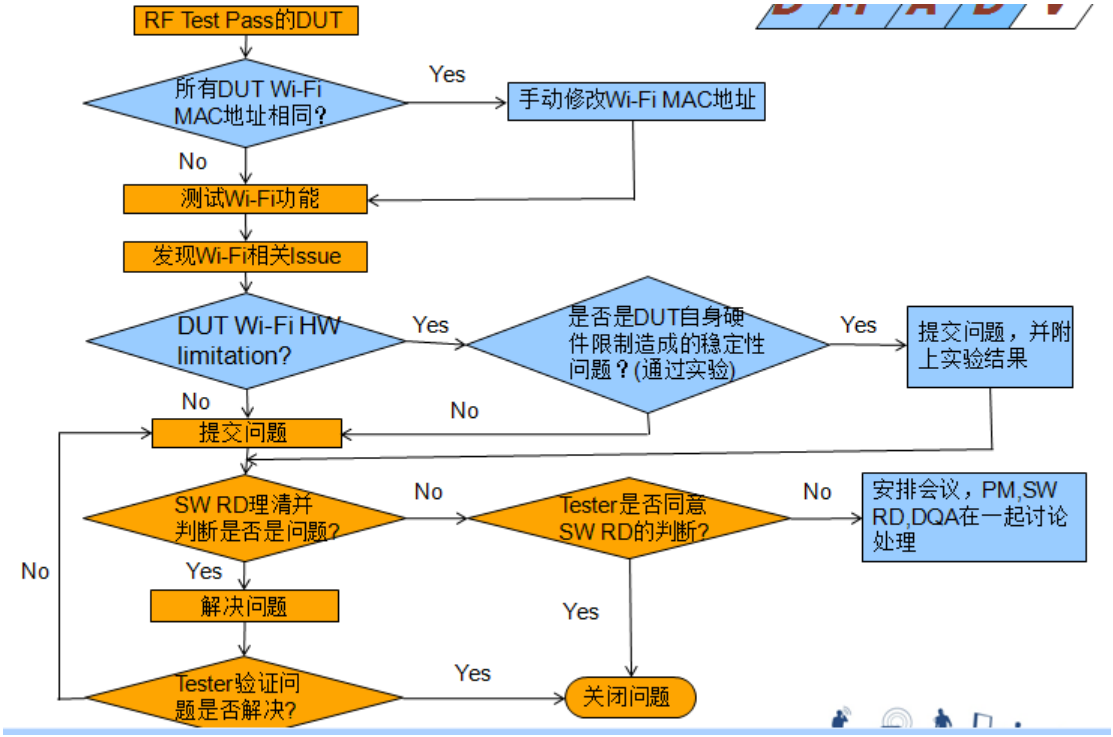
Descriptive Statistics: Test condition1, Test condition2, Test condition3

Variable	N	N*	Mean
Test condition1	40	0	58.5
Test condition2	40	0	6.025
Test condition3	40	0	5.150

Test condition2 和 Test condition3 平均值差异较小，即同频干扰存在，但影响较小

Test condition1 相较于其他 2 个 Test condition 有较大差异，所以，与同频干扰相比，当 DUT WIFI 2.4G 和 WIFI 5G 都开启时，对 WIFI 2.4G 稳定性影响较大。

2、最后改进的流程



四、总结

我们在这一期中主要介绍了统计学中推论学上的重要的知识点：箱型图的基本概念以及如何使用 Minitab 画箱型图；中位数检验的理论与我们之前学习的假设检定理论是同一个理论，只是待分析的数据是“非常态分布”的数据，所以才会使用此新的检验方法，之前几期杂志，我们学习的都是基于“平均数”进行假设检验。也给大家展示了如何在 Minitab 上使用此工具进行检验。

这一期的重点是如何将我们实际工作中遇到的问题，结合我们测试的经验设计试验方法，并结合 Python 脚本运用自动化工具搜集测试数据，然后利用统计学的知识，使用统计学的工具 Minitab 分析这些使用自动化工具搜集的数据，最后得出一个非常客观的结论，我们根据这些结论改善我们的工作流程。大家回味一下这个过程我们用到了“黑盒测试理念”(测试实验的设计必须是基本黑盒测试经验，才能设计得出来)，“自动化开发”(这些实验需要产生大量的数据，需要花费大量人力，使用人工是很不现实的，必须要用自动化工具辅助我们测试)，“统计学知识以及 Minitab 工具”(对于这些大量的数据，按照统计学的知识，运用统计学的工具进行分析)，最后就是能产生一个非常令人信服的结论，而对于我们测试人员，真的就不仅仅是个测试人员了，而是一个具有综合知识，多种工具的复合型人才!!

这次使用的工具也不是很困难，但是我还是要强调，“简单”的工具下隐藏了“不简单”的统计学原理，如果希望后续在大家自己的工作中灵活的运营这些工具，必须要理解这些统计学的原理。

还是那句话：对于 6 sigma 可以带来的奇妙旅程，我将不懈的坚持探索，怀着赤子之心去探索如何使用这些工具去改善我的工作，反之在不断的使用这些工具的过程中又大大加深了我对统计学原理的认识，所以“路漫漫其修远兮，吾将充满欢喜的上下而求索”！

参考文献：

1、有关标准差基本概念介绍的网站：
http://baike.baidu.com/link?url=4v_m3jJhHUKllt1lA0tRe_I-pVurV1tNNrztJ6_PbZf0Me5rJr-bxIdp4fRCdsrsSIRd-hSFBglCEyZCkunk6K
2、有关介绍统计过程控制的网站：
http://baike.baidu.com/link?url=M9BLOK736USXqiqCHa2uhW624zYJpD-UXRAotTwo9PyTn-AWZv3GDHofg2Gz_uOqC42RK1GoKdLf-Wr30fWJR0_sgWYIj_ItTG-JSdB1Amm

外包公司带给了我什么

◆ 作者：桃 子

一次偶然的求职经历，让我接触到了外包公司，当时非常急于找到工作，综合利弊还是选择进入了我现在所在职的外包公司。下面就入职前前后我个人对外包公司的看法做以总结，希望对从来没有接触过外包公司的测试人员有所帮助。

待遇不好，我为什么要选择外包公司

刚开始 HR 和我谈工资待遇的时候，除了工资比我以往的公司给的高之外，既没有 13 薪，保险还是最低档，没有什么条件吸引我，那么我为什么选择外包公司了呢？

环境所迫

介于我当时马上要结婚，又处于比较尴尬的年纪，外面的公司一般都不太考虑我这种已婚未孕的求职者，这点我深有体会

好奇心

之前经常听到周边的同事谈及到外包公司，都是一些不好的方面：比如，任务多，经常加班，没有加班费等等，但是在我心底有一个声音在告诉我“路都是自己去走的，你不走怎么知道自己不适合呢”，同时，那时候刚刚接触 APP 测试，这份工作也正是我以后想要研究的方向，它将给我一个机会进一步研究 App 测试，为何不可呢？

开始喜欢

入职一段时间后我被分到一个项目组，其中的接口人在异地，每天通过办公软件进行沟通交流，当时确实有点小困惑，做为新入职的员工，与身边的同事不在一个项目组，而且再加上这个公司的企业文化，可想而知，当时我的日子过得不是很舒心。

一段时间后，这段小苦涩的日子基本告一段落。我这人有一个最大的优点是适应环

境能力强，基本上新进入一个项目，都会快送融入到工作中去，因此领导有时候会把安排不过来的任务交给我，也因此我接触了比较多的项目，从而接触到很多有经验的前辈，跟他们学到了很多测试的小工具，以后有机会会整理成文档跟大家进行分享。

现在的项目平时工作量不多，公司每隔半个月会有培训，闲下来的时候研究研究自己感兴趣的技术，丰富自己的知识体系，对我来说从中也会感觉到幸福。

有点忧虑

最近公司项目组进行重组，很多项目收回交给本部，一些测试人员进行了释放，很残忍的事实，也许也是外包公司当今这么热门的最主要的原因了吧。总之，这也是我忧虑所在，如果项目整体收回，那么我的求职之路将会再次开启。

你怎么看?

上面就是我在外包公司的一点个人经历，先从利弊两个方面总结一下，最后给出一点建议，方便大家做出权衡

利

- 接触到不同的项目，增长个人阅历
- 了解不同的测试工具，加强测试广度
- 有完善的培训体系，加强测试深度

弊

- 项目不稳定时，如果公司还有其他项目需要用人或者协力公司会给推荐，还能继续保留原公司，否则基本上就会被释放
- 有些公司会对本职员工和协力员工区别对待，比如食堂饭卡、会议召开等
- 心里归属感差，总感觉自己不是这里的员工一样
- 大部分外包公司在待遇上不好（公积金和保险）,有些外包公司节假日、生日礼物都没有

建议

如果你考虑外包公司这份工作的话，下面几点最好向 HR 咨询清楚

合同类型：项目合同还是年限合同

一般项目合同，项目结束后人员就会释放，如果年限合同，外包公司还会继续提供工作进行面试等

所在公司的企业文化

是否有定期培训、项目多长时间（一般被告知很长）、所在公司外包人数占比（如果外包人数较多的话，对于你在心里上会舒适一些）

总之，每个人的做事方式方法会不一样，有喜欢稳定点的，有喜欢挑战自己的，路在脚下，适合自己就好。最后也非常希望大家后台留言，就你所在的外包公司的感受回复给小编，感谢大家的积极参与。

一次性能测试的分享

◆作者：寒　冰

性能测试分:

前段时间出差上海做了一次性能测试，在这个过程中遇到了较多的困难，也收获了很多，在这里想把解决问题的方法分享出来。

去现场之前了解现场的基本情况。这里主要是分为两点:

（1） 客户生产环境的软硬件设施

（2） 压力机的软硬件设施

对于客户的生产环境的软硬件设施我们一般可以从生产环境的网络拓扑图入手，对生产环境的网络架构有一个初步的认识。

这里有两点需要大家特别注意的:

（1） 客户生产环境是否采用了负载均衡，以及所采用的负载采用的算法。这里有一种算法是大家需要注意的：源地址哈希法，因为采用这种算法时，负载均衡器会将同一 IP 的请求映射到同一后台服务器进行处理。上次出差时客户现场使用的正是这种算法，压力机的请求却都是被服务器 B 处理的，而且客户也未告知采用了负载均衡，导致只监控了一台 web 服务器 A，但是因此得到的监控数据均是 0，导致不得不重新压了一遍。

（2） 服务器采用的操作系统。不同操作系统所需要的监控软件有一定差异，对于 windows 一般采用 LR 自带的监控就够了，但是如果是 linux 系统我们一般会使用 nmon 进行监控，因此根据不同的系统，做好不同的监控软件准备是我们到客户现场之前所需要的。

对于压力机的软硬件设施我们一般需要注意的是:

（1） 压力机采用的操作系统版本。现在大多使用的 LR 版本还是 11，LR11 是不兼容 win7 以上的版本的，上次出差就是因为客户压力机采用的是 win server 2012

导致不能兼容 LR11 导致浪费了一天时间来解决版本不兼容问题。因此根据要使用的 LR 版本让客户提供相应操作系统的压力机。

（2） 压力机测硬件设施主要关注对象为：压力机的内存。在执行场景过程中，随着虚拟用户的上升，占用的压力机内存也会越大，LR 官方提供的数据是一个虚拟大概吃掉 3-5M 的内存，因此根据客户的需求评估需要多少压力机是我们工作的前提。

（3） 注意压力机和服务器之间的网络连接，压力机和服务器之间最好是局域网连接，否则在压测时很容易造成网络流速达到上限，不能分析出系统的瓶颈所在。

测试过程中需要注意的地方：

（1） 压测过程中，开始人数从一个比较小的数值开始。在压测过程中，人数直接从 100 开始时，发现场景开始时的 tps 一直没什么变化，分析了许久才想明白应该是一开始就达到了系统的瓶颈，然后重新压了一次，人数从 0 开始，到 50 的时候系统网络流速就达到了极限，怪不得出现第一次的哪种场景。

（2） 性能测试的关键点在于产生有效的压力，因此，在这过程中，参数的取值模式很重要。在一次压测过程中，通过的事务达到了 1000，但是数据库里只增加了 100 条记录，后来分析一下发现由于参数化时，参数类型选的是：unique+once 导致每个虚拟用户只能取得参数列表中的一条记录，事务其实并未执行成功。后来将参数类型改：unique+each iteration 时，问题得到解决。

（3） 对于数据缓存的选择。在压测过程中，发现当 50 人并发登录时，网络流速达到 120MB/S，后来检查后就才发现脚本里包含了太多的 JS，CSS，图片的请求，这些数据的请求导致了网络流速较高，因此在压测过程中如果不是最大压力情况下，可以选择不录制 JS，CSS，图片的请求连接，可以通过设置 LR 中关于设置清空缓存的方法进行解决，这里就不再赘述。