目 录

让“用户体验测试”有法可依.....1

精确项目估计.....8

启用和简化移动性的八个首要考虑项.....12

软件测试：宏观视角——超越测试方法论.....17

生生不息的人性化测试-让我们更深入的理解测试.....28

Android 自动化框架浅析.....37

让“用户体验测试”有法可依

作者:琦少

前言

“用户体验测试”，“交互测试”这两个词，近来如雷贯耳。

公司亦身体力行。 然见效甚微，或略显盲目。

天道酬勤，我曰天道酬悟。

快跑却摒弃思考，久之必原地打转。

本无路，走的人多了就成了路。

驻足看看身后，其实已经有一条依稀的小路，通向远方。

是开路者，只愿做引路人。

回首，其实路在脚下。

三板斧之一信（1）

“信”者，准确也！古语云人无信不立，app 亦然。

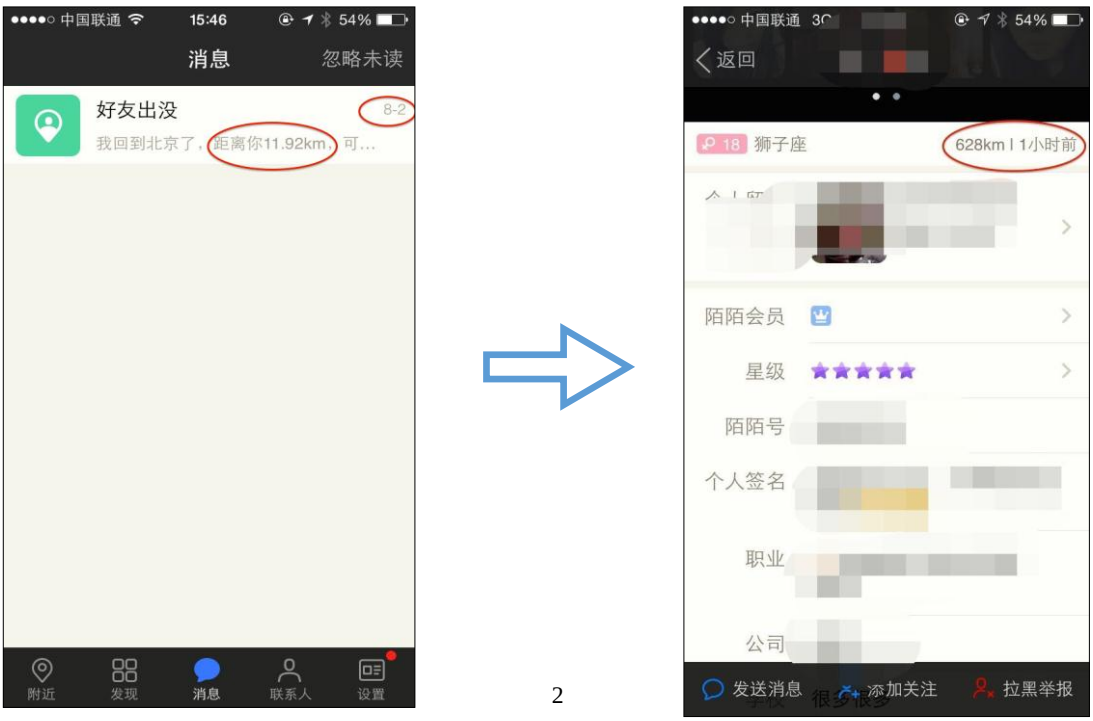
说明：注意圈起来部分。提示和实际差了 0.02G，即 20M。20M 意味着 1 首无损音乐还是 10 张图片……

三板斧之一信（2）

说明：注意圈起来部分。

左图为 app 检测出的剩余空间，右图是从系统设置中查看的。相差 200M。我想大概是 app 忽略了系统预留的空间。但是！如此一来用户如何敢信任清理结果？如何敢信任评分，可见“信”乃根本。

三板斧之一信（3）



说明：注意圈起来部分。

提示距离 11.92 千米，实际距离 628 千米，可知如今交通何等发达。千里江陵一日还不是难事。由于没注意距离信息的时效性，导致的不可信问题。一款基于位置的社交软件，没有时效性的位置提示，可谓无“信”不立的风险。

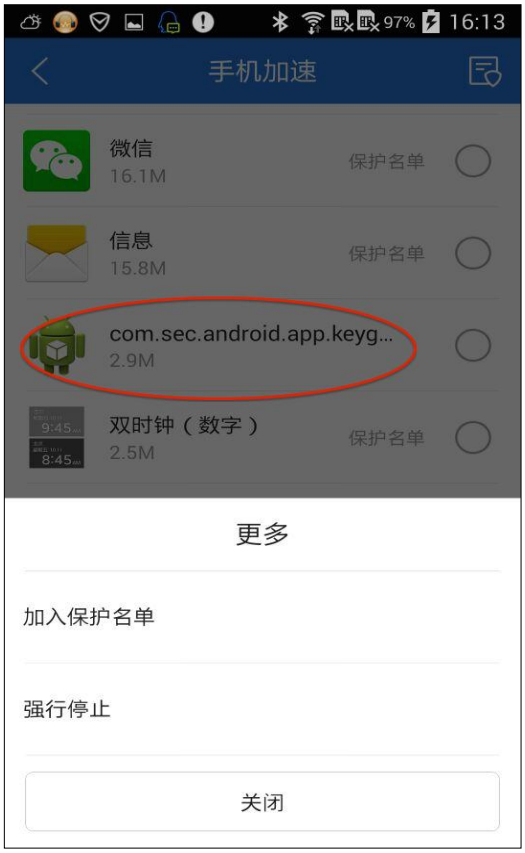
三板斧之一达（1）

“达”者，通顺也！古语有云，间关莺语花底滑，幽咽泉流冰下难。

表意清楚为“达”，操作便利为“达”，响应迅捷为“达”。

说明：这些相同的名称有何不同？不告知清楚，亦无预览，实难猜测。未知带来恐惧，恐惧带别疏远。

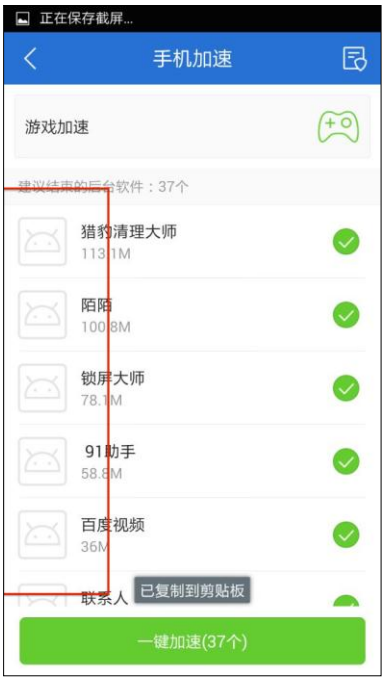
三板斧之一达（2）

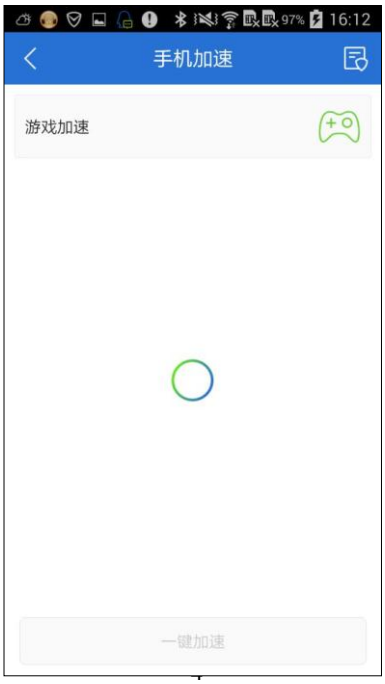


说明：左图，自己都不清楚的东西，叫用户如何敢相信是垃圾？

右图，未能告知用户将被清理的为何物，如何让人信服？

三板斧之一达（3）





说明：速度快才通畅，有多少“爱”禁得起等待……

三板斧之一雅（1）

“雅”者，优雅也！这也是衡量你是为了“做出”一个 app 还是“做好”一个 app。古语云：有美人兮，见之不忘，一日不见兮，思之如狂。让用户如此思念你的软件。

三板斧之一雅（2）

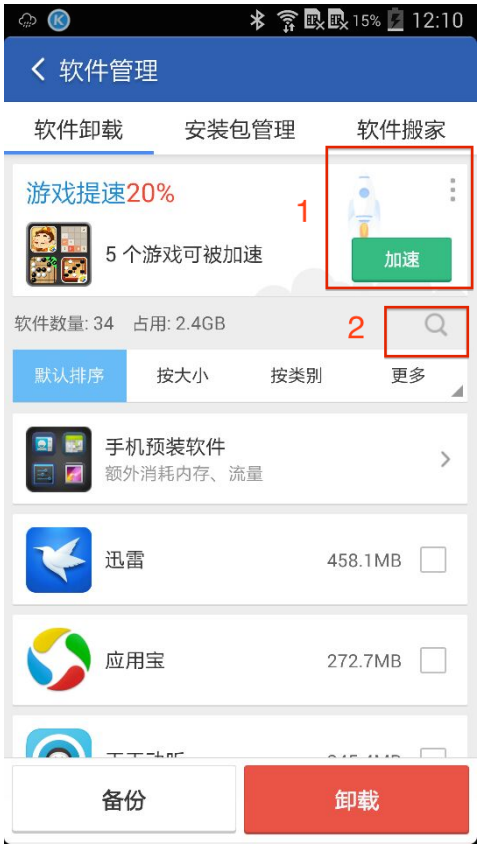
右侧是淘宝的排序，左侧是陌陌的排序。

1）虽然都没错，但左侧没有明确当前的排序规则。

2）有些只需提供单向排序。

例如淘宝料定用户只会按人气从高到低，所以并没有提供从低到高的排序，同理，陌陌的大多数用户希望的是登陆时间从近到久的排序。

三板斧之一雅（3）



说明：1. 处好看吗？没看出不对称有多美？

2. 处好操作嘛？甚至看得见嘛？靠点击率说话吧。

3. 整体凌乱否？现在有个词叫“做密集恐惧症”。

致那个终将来到的时代

那是一个崭新的时代，一个移动互联的时代，一个用户为王体验至上的时代。

更是一个终将到来的时代。

那是一个变革的时代，一个物竞天择的时代，一个大浪淘沙的时代。

也是一个人才辈出的时代。

时代，就像一个婴儿渐渐长大的脚丫，我们只是时代的鞋子。

那里没有测试只有 QA，没有 checker 只有真正的 tester。

那里是连需求也可以质疑，但反馈却显得倍加珍贵。

那里淡化了边界值，只因“大爱”无疆（注 1）
那里冷落的等价类，只因“苦海”无边（注 2）
那里遗忘了决策表，只因唯快不破
那里拥抱着“痛点”
那里崛起了“交互”
那里充满着“探索”
那里有你，有我更有千万的 QA 同胞，一切都在并不遥远的远方。
只有路，在脚下

释义：

注 1：“大爱”在这里特指用户喜欢的软件。该句意思为很多优秀的软件，往往是没有所谓的边界概念。

注 2：“苦海”在这里指的是无效等价类。该句意思是无效等价类太多了，有效等价类的验证谁都会做。无效等价类的把控才是体现测试水平的地方。

精确项目估计

译者：于芳

“把工作分解成小的任务，精确地将要点分配到这些任务中，追踪完成时间，查漏补缺，避免损害团队成员之间信任的行为，这些有可能建造一个精确的项目估计的文化。”

有效的项目估计是软件工程师们在工作中遇到的最严峻的挑战之一。不管团队大小，干脆利索地定义、估计和分配团队工作是很重要的事情。当团队变得越来越大，养成良好的计划和估计工作习惯就更加重要。缺乏规划和估计工作会降低一个项目的信心，破坏团队成员和业务之间的关系，从而使得开发对每个人来说都更难。在本文中，我提供了可以使估计工作更简单的技能，给出怎样在这些估计工作中建立团队成员之间的信任的方法。

我们从定义一个工作单元开始。敏捷团队使用像叙事诗和用户故事这样的词语，而传统的团队可能使用任务或性能这样的字眼，而所有的软件团队成员使用缺陷这个词！我要将工作定义为一个变化，一个需要一个或更多人来做出努力的当前状态中的变化。已定义的工作大概也包括清晰的成功度量，所以团队成员知道什么时候工作算完成了。

从产品路线图开始

每个团队都需要建造一个路线图。对开发团队来说，了解代码库将怎样随着时间的推移而演化至关重要。路线图很可能会有四到六个大的启动项。每个启动项内部会有几个大的性能组成。每个性能又可能有多个组件。而为工作交付，每个组件需要一组任务和一组用户故事。

将每个组件分成 8 到 16 个的系列任务很重要。最有效的估计过程需要整个团队参与。作为团队计划，有关个体工作流如何融合到一起的疑问将会跳出来。这是一件好事。这些问题驱使有更多关于项目的根本性的谈话和最小化惊讶之事的发生。如果一个团队新进到一起估计工作中，至少要从产品所有者，Scrum 专家和开发团队人员（包括开发和测试工程师）开始。

你大概会想：“我不可能对我整个路线路做那么详细的估计！”说得对。你不应该那么做。只对功能管线上最初的几个项做详细的估计。一旦团队成员了解了发布最初的几个项所需要的努力，就会对预测备份日志里工作范围有更深的帮助。这突出了一个重要的原则：估计工作的准确性随着工作范围的增大而降低。

通过使用待办事项最小化浪费行为

通过使用待办事项，敏捷团队享有在计划和发布功能上更大的灵活性。一个工作待办事项其实就是在团队和义务上达成的一个按优先级排列的工作方式。最重要的项在待办事项的顶端。这些项随后列在日渐减少的优先级上，逐渐从待办事项里走出。当一个组完成了顶端的这些项，他们继续向列表上的下一个项走去。

在项目的过程中，业务优先级会有变化。产品拥有者可以容易地移动待办事项里的团队不在用的东西。该团队不受影响，每个人也都关注在前面的目标实现上。

相对估计案例

对初进的新手，我强烈推荐使用相对估计方法。我在 Scrum 培训中学到了使用故事要点的方法。故事要点分配斐波那契值（1，2，3，5，8，13 等）而不是时间估计到工作主体中。我们举一个简单的例子：想象尝试着去估计完全消费一片水果的相对复杂度。待办事项的第一个项是去吃一个苹果。我们给一个故事点分派 2 的值。当看下一个项，一个香蕉，我们知道它至少与一个苹果一样复杂，额外的要去皮的成本除外。我们给香蕉一个 3 的值。橘子更复杂，所以我们给它 5.一旦我们吃了另一片水果，我们回顾过来，将那个经验跟吃苹果的经验相比较。然后我们再确认估计两种经验的成本。如果吃水果比我们估计的要难得多，这就是一个重要的学问，然后我们做调整。理解了流程中的学问可以帮助我们改善估计文化。

那么为什么使用故事要点而不是时间值呢？故事要点可以允许团队成员关注发布一片工作涉及的复杂度和时间。团队成员将新工作与他们已经完成的工作做比较。他们对比过去的挑战工作比较新任务的复杂度，然后将需要的时间和难度排个列。

使用时间价值方法，我们不经常对“做业务的成本”负责。会议，邮件，代码评审，等等---所有这些都是每个工作日要做的必不可少的工作，但是他们并不能算作“工作”。故事要点将工作流项和非工作流项分离开，因此估计就更加一致、始终如一了。

最后，一个团队的流程会随着时间的推移而不断演化。当团队成员使用了更好的开发方法，其发布工作的能力也会随之增加。故事要点关注在工作的复杂性，正好与持续时间相对，所以故事点方法更能抵抗变化。相同地，如果一个团队对完成工作的定义改变了，故事要点方法会随着时间的变化更能站得住脚。

估计工作是自我证明的

在软件团队中，关于估计工作的恐惧有很多在“长腿”。我们都经历过像这样宣言的项目，“这个功能点将在三个月内完成”，真正的含义是“全身投入到开发这个功能中，不管多少成本就要在三个月内完成”。这给失败和错过最后期限创造了潜在可能。当将推迟日程表上的日程有舒适可循，它就限制了商业随着项目不断演化做出正确决定的能力。

“把工作分解成小的任务，精确地将要点分配到这些任务中，追踪完成时间，查漏补缺，避免损害团队成员之间信任的行为，这些有可能建造一个精确的项目估计的文化。”

敏捷组织机构的重要特征是其导航和在改变中推进的能力。估计工作会转变。重要的是，关注待办事项上的开发团队和业务团队。产品拥有者负责最大化开发团队人员的价值。开发团队的估计帮助产品拥有者决定工作成本，对产品发布工作做优先级。

开发团队最具战略性的关系是与产品拥有者的关系。当开发团队成员对项目工作做了更好的估计，产品拥有者才可以更精确地计划产品的路线图。结果，产品拥有者也不会去参与诸如可以给开发团队增加内容转换和逆行功能的中途冲刺的反模式活动中。

信任为何是根本的粘合剂

软件开发工作的估计没有组织内的信任是不可能有效的。从工程师到财务到市场营销，我们因为专业人员的技能雇佣了不同的接受过训练的人。保持像一个团队工作，建立成员间的信任很重要，只有这样才能开发出最好的产品和项目成果。

不幸的是，下面这种类型的对话经常出现：

“发布性能 X 要多长时间？”

“我们认为要三个月”

“我需要它两个月之内完成。想办法做到。”

“呃……”

如果项目范围不做改变来适应新的发布日期，开发团队会认识到估计工作没什么关系。工程师感到非常有压力，要走捷径去影响产品质量。信任坍塌，项目可能在还未真正开始之前就崩溃了。每个人都受苦。

现实是有时候产品的确需要有一个交付日期底线。下面是一些帮助促进有效率的谈话的问题：

- 这个启动项最重要的一部分是什么？（“所有的”不是一个真正的答案。）

- 有没有能解构和在团队内平行工作的方法？
- 我们怎样能在开发循环早些时候得到反馈来确保我们有对的最不可行的产品？

在少数案例中，一些特别的工作流不得不赶快做完。产品拥有者和业务领导可以推动团队成员多增加工作时间---但是要意识到耗尽精力长期上会降低产品质量，这个很重要。

授权给产品拥有者去关注正确项上的施工时间可以让每个人从瀑布式开发的挑战中得到解放。团队是受制于范围而不是时间。工程施工能力是一个固定的实体，这个实体可以为产品拥有者使用来发布产品。

创建一个估计文化

早些时候我提到你不想为待办事项中的项做很详细的估计。估计一堆永远不会完成的工作是对团队人员时间的浪费。当团队完成一个里程碑时，花点时间来评审估计实际交付产品需要的时间。

如果有显著的差别的话，回过头去理解为什么。如果预测的待办事项中的估计工作有被影响的，花时间去更新估计结果。公开谈论这些问题会让每个人在产品演化的时候保持在同一个进度上。

从 Scrum 开始

开始学新东西可能会比较有挑战。我推荐专注学习 Scrum 框架内的故事要点。聚焦于分配故事要点值到冲刺内部的个体项中。记录下有多少故事要点达成了每次冲刺。几次冲刺之后，该团队就会习惯于一个速度—该团队在一定迭代中可以交付的故事要点的数量。

持续的速度解开估计更大数量级工作的力量之锁。如果一个团队每次冲刺可以完成 50 个故事要点，他们可以在 10 次迭代中完成 500 个故事要点。但是要记住，关键的是要在团队中创建估计文化。学习什么时候估计是正确的，如果有遗漏的话，花时间去弄明白原因，并且用那个所学知识去指导以后的工作。

启用和简化移动性的八个首要考虑项

译者：于芳

通过拥抱不断增加的移动终端和到处存在的网络连接带来的机会，部门可以使自己的企业更敏捷，节约成本和具有竞争性。事实上，支持更具移动性的劳动力和顾客基数不再是部门的一个选项—它是一个必备项。员工和顾客期望从其移动终端与企业互动，他们还期望自己的移动业务解决方案能与他们在办公室外使用的移动个人解决方案一样易用、设计优美。同时，将自己的人事搬出办公室进入劳动力市场可以最大化当今复杂设备的独特特征的工具的劳动力，部门可以比现在更有效率。幸运地是， 拥抱移动性并不需要很复杂或者花费很多。现在告诉你怎么做。

智能移动设备的销量在 2012 年超过了 10 亿台，在接下来的几年里应当会达到 20 亿到 30 亿，取决于你选择哪种调查方法。但是这些数字并没有说明事情的全部情况：移动性不仅仅定义了计算领域的新时代的特征，也定义了我们生活，工作，沟通和互动的新方式的特征。为了让商业能够完全参与到这个移动的时代，部门必须以下面这些术语来思考：

移动性—启动企业

这意味着使得顾客能够从移动设备与业务互动，并且允许员工能够从任何他们选择的移动终端安全、便捷地工作。部门应当能够部署简单和廉价的工具和流程到启用了移动性能的新定制的应用程序，和简单地为现存企业程序创建移动方法。

移动—盘活 IT 部门

这意味着利用常在连接状态和有独特特征的移动终端来加强，并且在某些方面，重造服务的使用。当与业务决定者工作距离更近，并且装备正确的工具来快速解决问题和增强服务质量的时候，人事可以变得对业务需要更有效率，更具责任感。

这个新时代的成功的一个典型的方面是，识别基于移动的交互的独特特征，它意味着保证设计移动程序来利用移动设备俄日的特征和功能，同时在此基础上建立更多属性，让移动设备很方便、引人注目和令人兴奋。

很多 领导者运营移动设备是基于认为增加移动功能是一项复杂和费钱的事情的认识上。但是，实际上添加移动功能不需要这样，如果使用了正确的解决方案的话。

事实上，添加移动性能和支持可以而且应当是自动化的。这里是怎样成功地添加移动性能和支持的八大首要考虑项，前三条讲添加移动视图和功能到业务程序中，接下来的三条讲使用增加的移动性来帮助重建，最后两条将所有的东西放到一起。

简化添加移动视图的过程

在当今的环境中，任何一个有基于浏览器的界面的程序应当也有一个移动界面，而你应当能够在笔记本，台式机，平板电脑，手机或任何其他设备上访问这个程序。现实中，标准的浏览器界面和移动界面的区别不应当减少，而应该是一起消失。移动界面应当对用户来说很熟悉，并且不需要管理员做任何返工工作。如果你每次想添加移动性能到一个程序的时候都要重造组、部件的话，而如果没有平台能使你简单地添加移动视图，你就会要浪费很多钱进去。你也许需要购买另外的服务器，存储器或网络带宽，但这些花费和令人头疼的事情不应当有。一旦一个程序部署完毕，启动移动视图应当是一个相对简单的过程，没有额外的配置需求，没有额外的需要购买的硬件或软件，没有定制，也没有要安装的任何其他东西。

使用移动设备的本地性能

添加移动视图应当是一个简单的过程，每个移动程序应当能够使用移动终端的本地性能和其独特的特征。工人与移动终端交互和使用移动终端的方式与他们使用笔记本或台式机工作的方式是不一样的，而应当加强移动程序来反映这些不同处。移动用户倾向于关注较大量的简单人物，而且他们期望用最小量的交互来完成这些任务。如果你让你的用户快速地访问看过的文件，或者你让他们能使用简单的命令包含更多点击动作而不是输入打字动作，你会从移动程序中得到更多的价值。

另外一个关键点是确保你的程序和方案使用了移动设备的本地性能，很多性能对连接台式机的用户是不提供的。例如，你可以使用内置的摄像头来将图片附在任何文件上，或简单地扫描条形码。 很多机构也在趁机利用移动设备的位置感知性能 GPS， 还有像地理位置标记，自动导航路线性能，因为移动设备具有存在感知和位置感知性能。关键是移动设备给人更多的创新机会，这也是为什么有一个能使你不仅能简化移动终端支持功能，也能简化开发程序和最大化潜能的方案的平台很重要的原因。

启用重要趋势如 BYOD 和 IT 定制

把你自己的设备带过来（即 BYOD）和定制是驱动今天移动应用在大部分业

务的两大趋势。支持 BYOD， 你要确保移动程序可以通过任何设备访问，不管是什么操作系统，因此他们应当除了支持台式机浏览器界面外，也会支持 iOS，安卓，黑莓和 Windows Phone 用户。定制的概念是今天的用户---不管他们是部门的成员，还是机构的员工或者顾客，都期望他们很多业务程序的特征与其个人应用程序的特征相同。这意味着你要保证你的移动程序是有直觉性的，易用，设计优美；同时，在前面提到过的，是功能增强的，可以使用移动设备的本地性能。最近你仍然会看到太多程序看起来是他们想要表现的那样：遗留的客户/服务器程序，它们已经被重新配置过来支持移动性但是没有现代 Web 程序的样子，感觉和功能。

让 IT 从幕后走出来，走进商业视野

目前我们讨论的很多东西都与企业的移动性能加强有关。但是认识到移动访问的天价和移动程序的添加能且应当用来转换部门向商业提供服务的范例也很关键。通过使团队成员移动化，让他们能使用如 iPad 之类的设备，你的机构可节省金钱、提高效率，要把技师放在劳动力之间而不是将他们封闭在数据中心里。使用移动设备，台式机支持团队应能够创造一个用户桌面的事件或者变更请求—如此增加的移动性能将不仅仅是一个省钱机器，同时也是一个供部门提高商业服务质量的一个工具。一个移动化的部门能够改善访问性，提升责任感，在企业内提供更好的服务，不管用户在哪个地方。除此之外，如果移动应用程序使用设备的本地性能，部门的服务可以在能使人事只点击几下就能处理问题的直觉性的界面上优化其性能和导航功能。

提供 IT 管理以更好的信息来更快做决定

另外一个移动劳动力的大的好处是它可以给管理部以访问实时信息的访问权和进入到基础设施内部看故障探测和问题解决的可见性。如果决策者有一个访问所有基础设施的度量学的移动接口，他们就能快速地评估和解决问题，使用移动设备的灵活性来解决问题。在今天的环境下，商业活动每周 7 天 24 小时的运行着，移动性在减少机器崩溃事件和保证关键任务程序的性能上的任何退化状况都能立即高效地解决会起到一个关键的作用。

增强合作，加强定制程序的开发

让部门更具移动性的最大的好处之一是它使得部门获得更多访问，同时与商业连接更多。商业决策者在每日的商业活动中与人事交互更多，他们知道部门的人随时可以找到---和正确的工具 — 来处理时效性强的难题。这种每日的交互可以在很多机构内部创造一个文化范例，这里部门和商业决策者在处理定制程序和

商业服务的开发商更加合作。这种商业和更紧密地结盟会对具有一个平台，一个能够支持简单开发定制程序和部署定制程序的平台的环境中有非常大的影响。

拥抱未来

移动终端将会继续在部门内部激增，在整个劳动力市场和顾客中激增。对大多数个人，他们会成为主要的设备，很多情况下，他们的唯一设备。为了说明事情变化有多快，看下面这个事实：在过去两年里，活动的 Facebook 的移动端用户数量几乎翻了 2 倍，其中三分之二的 Facebook 用户至少有些时候是通过移动设备访问该服务的。超过六分之一的 Facebook 用户只使用一个移动设备来访问 Facebook 的服务。同样类型的更多的移动平台的使用也开始在商业程序中初现雏形。因此，每个组织必须提供其人事和其他组织以真正移动的，同时具有那种可以与个人用户习惯使用的从消费者世界看事情的移动程序相媲美的具有那种样子、感觉和功能的应用程序。移动性是所有下一代程序和环境的特征之一。越晚启动移动性，你就会在竞争中愈加落后与他人。

使用正确的工具来加速和简化具有移动性的程序的部署和使用

为你公司添加移动性能时，你需要一个计划，策略和平台以保证最大程度的成功。做计划和部署对的平台时有几个特点你应当要注意：

简单

太多的移动化尝试被过多的管理和配置流程毁坏掉。确保你有一个合适的平台，在部署移动性程序时不需要额外的资源。

节约成本

启用移动视图和移动程序不应当要求你购买新硬件、软件或者测试框架以保证跨终端的支持。

对程序功能的完全支持

你应当能够很容易地添加设备特定的视图，而不用修改底层的程序流程图。

移动特定的性能和功能

你要确保你的移动程序支持移动设备使用的独特的方式，具备像快速访问最近使用的文件的性能和简单的访问到频繁使用的程序的性能。

支持下一代 IT 的工具

为了处理日前商业的快速敏捷的需求，部门必须变得更加移动化。确保你的

服务自动化平台优化过具有移动性功能，能够让实时访问到任何设备、任何地点的系统。

下一步

顾客应用程序呈现很重要的价值。他们提供了自动化工作流程的机会使得商业活动更加有创新性，更加有效率和更具竞争力。使用对的工具，机构们可以通过加速开发速度和简化开发过程，让人员在与商业决策者紧密联盟的状况下管理程序的开发最大化定制程序的价值。而通过使用定制程序开发优化的平台，商业活动可以降低成本，减少总成本，增强敏捷性，加速新商业服务的价值。这个商业案例是显著的，技术也有。没有理由来等待。找出你的公司在简化和加速定制程序的开发中如何获益吧。

软件测试：宏观视角——超越测试方法论

作者：殷亮

决定的内容总是具体的，因此无法预计；它总得由人去发明。

——让-保罗•萨特（Jean-Paul Sartre）[①]

宏观问题

检验一个软件系统所需要的测试是不可穷尽的，因为我们不可能遍历客户的所有可能操作，或者说即使能够穷举这些可能的操作，并且我们在实验室所进行的测试与客户所做的现场操作步骤完全一样，但我们仍只能说这是不完全相同的两个场景，就像古希腊哲学家赫拉克利特（Heraclitus）的名言：人不能两次踏入同一条河流。场景的每一次执行既相同又不完全相同，而时空的不可穷尽则是测试证明的本质障碍。

我们既存在又不存在。

当认识到我们不可能证明一个无穷的场景集合时，软件测试就成了一门有关选择的科学：其核心内容便是研究如何从无穷集合中挑选一个有限的集合来对软件系统进行测试。在这里，我们将测试选择的这一根本问题视为宏观问题，并将在下文中展开更为详细地分析；而将关于测试我们知道什么以及不知道什么的问题，划归到哲学视角，这已在上一篇论文中得到了比较全面的探讨[②]。

测试的比较

我们必须承认的是，假如真能够从证明的角度来看待软件测试的话，那么一切与测试相关的理论无疑都将是纯粹的和精致的，同时其明确的结果也满足了人对于预期、秩序与简单的追求；但我们仍不得不放弃它，因为在我们有限的生命与有限的理性里，我们不大可能解决归纳与证明问题。也可以说，通过放弃对证明的奢求，我们因而得以回避了一个无法解决的问题，也不再被无穷集合所困扰。但是，这样的选择并非是没有代价的：一方面，与测试相关的理论变得不那么简洁和优雅，而是不可避免的略显晦涩；另一方面，当我们绕过证明这一不可征服的高山时，却与另一难题狭路相逢。

这一难题就是对于不完备的测试，我们如何比较两种测试设计——也就是两个不同测试子集的优劣。在将测试视为一种证明时，因为我们的目的是证明所有场景，所以除了完全集合，任何不完全子集都是不完整的。而如果比较的是两个不同子集，则只要子集内部不存在冗余，那么两个子集的比较便是简单的元素数比较。因此可以说，在证明的世界里，这一难题并不存在；但是，当我们从证明

的世界里走出时，我们却不得不面对这一难题的纠缠。

同一个受测系统的两种测试设计哪一种更好，这并不是一个容易的问题，尽管我们可以教条式地回答说：两相比较，我们选择“以最少的测试检验出最多的缺陷”的那个。然而这个的原则本身就有诸多疑点，在 Jorgensen 2002 中，保罗•乔根森（Paul Jorgensen）认为测试有效性的最佳解释是最困难的：“首先，要假定我们知道程序中的所有缺陷。形成死循环的是，如果我们知道程序中的所有缺陷就会采取有针对性的措施。由于我们不知道程序中的所有缺陷，因此永远也不会知道给定方法所产生的测试用例能否发现这些缺陷”[③]。实际上这里的困难是，我们不能将最终检验出的缺陷数作为评价一个测试设计的标准。试想假如一个包含了 1000 个测试用例的测试设计，最终结果是只有其中 1 个测试用例检验出了 1 个缺陷，另外的 999 个测试用例均测试通过；而另一个测试设计就仅仅包含了这个检查出缺陷的测试用例，然后它也同样地检验出了 1 个缺陷，这时难道我们可以说后一测试设计比前者更好吗？当然，对于这个特定的软件实现（哪里有缺陷，哪里没缺陷）来说，后一测试设计的确更有效率，因为它发现了同样多的缺陷，却只用了相对于另一测试设计 1/1000 的投入，完全符合“以最少的测试用例检验出最多的缺陷”的标准。但这里的关键是，对测试设计的评价是不能依赖具体的软件实现的，某一测试用例是否有必要，不能以它最终有没有检验出缺陷来衡量。无论是否检验出缺陷，对于测试设计来说这都应该被视为一种偶然、一种对未知。在前面的这个例子里，我们可以这样认为：后一种测试设计它仅仅只是偶然地遇到了这么一个特定的软件实现——在这个软件实现里，正好在它检验了的那部分存在缺陷、同时也正好在它没检验的那部分表现正常。因此，我们没任何理由认为这样的一种“结果高效”的测试设计是优于其它测试设计的，就像我们不能因为有人购买一注彩票中奖就否定其它投注者的投注方式一样。对测试设计的评价不应该建立在任何特定的软件实现的基础上，因为任何特定的软件实现在这个不可能精确计算的世界里都应当被认为是偶然，而最终的缺陷分布情况也是偶然。也就是说，当我们以结果来评价两个测试设计时，如果一定要分出优劣，我们也只能说在某种特定的软件实现与缺陷分布下，一个测试设计偶然地高效于另一个测试设计。

当然，还有另一种说法，即评价测试设计的高效的标准是“以最少的测试覆盖到最多的缺陷”。随着从“检验出”到“覆盖到”的转变，“缺陷”一词的意义也随之变化了，这里的缺陷一词就特指潜在可能的缺陷，因而它不是结果论的，也由此避免了对特定软件实现与缺陷分布的依赖。但这一标准并不能给我们多大的指导意义，因为如果违背这一标准，则意味着对于同一缺陷我们加入了多余的测试。但是在什么情况会出现多余的测试呢？完全相同的测试肯定算是，但不完

全相同的测试呢？比如对于一个接受 1~255 的输入，我们设计了 5 个测试：分别 1、2、250、254、255，如果在此基础上，我们再加入一个测试用来测试输入 3，那么这个测试算是多余的吗？在这里，除非我们能断定导致测试输入 3 可能失败的潜在缺陷与其它输入可能检验出的缺陷是源自同一错误，否则我们就不能将这个测试视为多余。但是很显然，我们并不能做出此类的断定，因为既然讨论的是潜在可能缺陷，那么我们所能断言的就只是其结果（缺陷的形式），而不是其原因（错误根源）。

因此，我们只能说对于测试设计效率的客观评估是困难的，不管是遵循什么样的标准。当我们试图比较两个测试设计时，如果以最终检验的缺陷数来衡量，那么这一评估也就仅仅只是对当前的软件实现是客观的，而对于其它可能的实现或缺陷分布来说则是未知的；另一方面，如果我们用覆盖的潜在缺陷来衡量，那么这似乎又是另一种形式的倒退，因为我们必须对两个缺陷是否源自同样的错误做出断言，但我们也知道，这一过程既是不可能的、同样也是不充分的。

测试的模式

选择的存在，一方面赋予了软件测试以新的核心与意义，标志着测试不仅仅只是执行检测，其更为重要的内涵是做出决策；但另一方面它也使得测试工作者不得不面对着新的困难:即如何做出比较与选择的问题。在测试学科数十年的发展历程中，人们发明了各种各样的测试方法，这些测试方法所基于的假设与侧重点并不一样，但目的都是为测试工作者的选择提供一定的指导。

在这些方法中，无论是功能性测试还是结构性测试，本质上都是关于选择的。在功能性测试中，边界值方法指导我们在边界与非边界中作出选择（例如在测试 1~255 时，我们选择输入 1、2，而没有选择 3）；在等价类方法中，我们实际上也是通过划分互不相交的一组子集，然后从子集中选择一个元素作为测试输入；决策表方法同样是关于选择的，满足一个规则的变量可能有许多，但最终我们只会选择其中一个。而对于结构性测试而言，尽管在方法的立足点与技巧上相异甚远，但是结构性测试方法的本质上仍然还是选择：因为使得程序在某一特定路径上执行的选择大多数时候并非一个。

因此，我们可以说测试方法简化了测试选择，或者说，这些测试方法为我们所遇到的特定的选择问题提供了成熟的解决方案。比如边界值方法指导我们在面对一组输入范围时，如何选择测试用例，以更高效地找出系统里最可能出现的与边界相关的错误；但这一方法同时又对布尔变量和大部分 GUI 操作无解。其它测试方法也是如此，实际上每一种测试方法既有其擅长解决的问题，也有其不适宜的场景。比如，当我们比较功能性测试与结构性测试时，我们就发现：功能测

试揭示不了软件实现了未被描述的行为，而结构性测试则发现不了软件未实现的被描述的行为。因此，我们可以把测试方法看作是解决特定问题的捷径，这些捷径的存在使得我们在面对这些特定问题时能快速做出成熟的选择，而不需要完全从头开始思考。建筑学家克里斯托夫·亚历山大（Christopher Alexander）作为模式与模式语言的奠基人，他对模式有如下的定义：“首先描述在我们的环境中反复出现的问题，然后给出该问题的核心解决方法，以这样的方式，你可以上百万次地使用这种解决方法”④，因而我们可以认为模式就是对一类特定问题的抽象与简化，并给出解决方案。正如建筑领域的建筑模式与软件设计中的设计模式一样，我认为测试方法就是软件测试领域里的模式。这类模式的存在，使得测试工作者并不需要彻底弄懂边界值测试方法的原理（不小于可能会误写为大于、不大于可能会误写为小于），也能设计出清晰的一致的测试用例集。就像掌握设计模式后，一些新人也能写出优雅且极具扩展性的设计一样。

相比起深层次理解要解决的问题，应用已有的模式无疑更为简单；同时，由于模式是建立在前人的经验上，应用模式显然能避免无谓的错误。因此我们可以这样理解:模式是一种具备了高度的简单性与准确性的理论，但它并不具备足够的普遍性，因为模式原本就是站在普遍性的对面，它针对的就是某类特定的问题。或者我们可以这样理解，模式牺牲了一定的普遍性，换取了高度的简单性与准确性。这就是沃伦•桑纳葛特（Thorngate 1976）所提出的相对称的复杂性原理⑤：一个社会理论通常无法同时具备普遍性、准确性以及简单性。当我们试图将简单性与准确性收入囊中的时候，那我们就自动地牺牲了普遍性。反之亦然，当我们说好的测试设计就是“用最少的测试用例检验出最多的缺陷”时，实际上我们是陈述了一个普遍的、简单的观点，因而当我们发现这个观点并不那么准确时，我们也不应该感到意外。

然而，在当今各个领域的研究中所存在的诸多问题，其根源多为不愿接受前文中所提及的不可避免的折中选择，研究者们表现得似乎可以在他们的方法或理论中同时兼顾这三个目标，实现鱼与熊掌兼得。甚至在那些获奖的项目管理书籍中（比如 Rothman 2007），我们也能看到一些看似有理并且言之凿凿的论点，可是只要进一步思考这些论点，我们就不难发现它们要么没有断言任何确切的因果关系，要么断言了因果关系却不够精确。我们必须认识到，软件项目自是千差万别，而软件团队也是百态纷呈的，再加上技术变革的层出不穷，在这么一个复杂的领域里，要想提出一个既简单清晰、又准确无误，同时还能放之四海皆准的全新理论无疑是极为困难的。我们必须知道我们想要的是什么，以及为了得到想要的我们必须牺牲什么。

就测试方法而言，尽管没有哪种测试方法能适应所有的问题，但它们所提供

的简单性与准确性依旧是宝贵的，而且也将永远都是我们在测试设计过程中所能依赖的最重要的工具。然而，借助测试方法这一工具，我们并不能回答为何如此设计测试的问题。方法不是选择的原理，因为方法自身也需要原理。因此，当我们决意探究有关测试设计的普遍性原理时，我们就必须超越易于理解的方法论，同时我们也必须作好牺牲简单性的准备——假如我们并不打算放弃准确性的话。

测试的选择

如果不存在匮乏，就不存在选择。如果测试可以零成本地执行，既不需要人力资源成本，也不需要时间成本，那么测试选择将没任何意义。但是正因为人力与时间均是有成本的，所以我们才有需要在不可穷尽的可能测试中做出选择，选择其中一部分而放弃余下的；选择值得测试的，而放弃不值得测试的。在这里使用的“值得”一词，实际上与我们常说的投入回报比，或者经济学中的边际收益—边际成本是一个意思，而边际收益—边际成本分析工具仍然是现代选择逻辑的核心部分。因此，涉及到有关测试的选择：选择一部分场景而放弃另一部分场景，选择某一个场景而放弃另一个场景，最终都可还原成对收益与成本的综合评估。

在通常语境下，如果我们说测试某一个场景的收益大于另一场景的收益，这意味着什么呢？如果考虑到测试是一种批判，那么我们显然应该选择更有可能检测出问题的场景，比如在测试 1~255 的输入范围时，我们在 1 与 2 之间选择 1，这是因为我们认为相比起忘记累加，程序员更容易犯将不大于写成小于的错误。而当我们在各种环境配置中进行选择，测试某一种而不去测试另一种时，也是因为我们判断这样的选择相对而言会更有可能检测出缺陷，或者说至少也是同等可能性的。另一方面，当我们判断两种场景都有同样的可能性检测出缺陷时，选择可以是随意的。比如我们就不会觉得选择测试输入 10 与 11 会有什么本质上的差别。但是，当两个场景具有不同的重要性——也就是说一个场景的故障后果可能严重于另一个故障时，即使我们认为两者具有同等的可能性，我们也会倾向于选择更为重要的那个场景。根据这个原则，我们往往不会选择测试最终用户不会使用或较少使用的环境配置，而在选择测试主要场景还是扩展场景之时，答案也不言而喻的，即使一般来说扩展场景似乎更有可能检测出缺陷。因此，在比较的两个场景的测试收益时，我们将可能检测出缺陷的概率，以及缺陷的严重程度为依据做出评估：

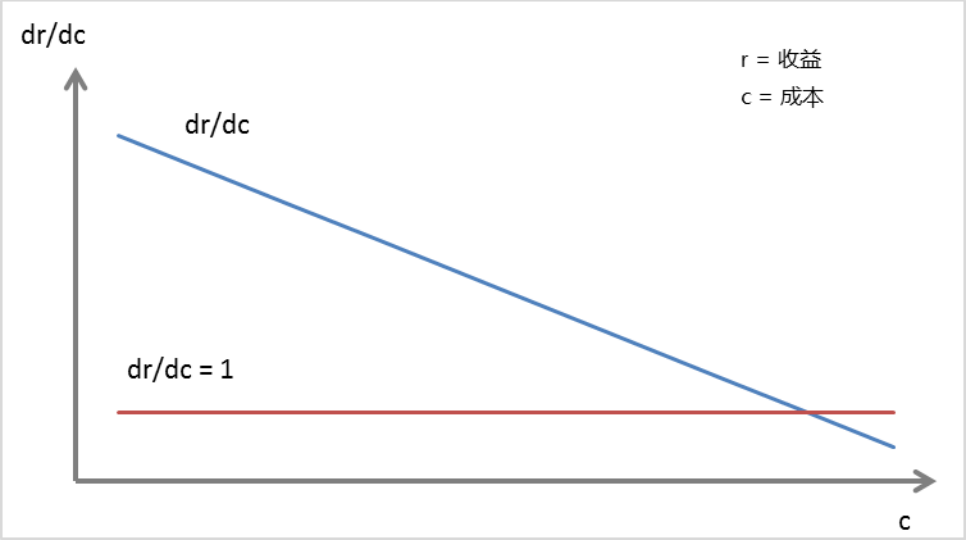
测试收益 = 检测出缺陷的可能性 ╳ 检测出缺陷所能避免的损失

上述公式已能帮助我们对测试选择形成一个比较直观的认识了：在检测出缺陷的可能性相等的情况下，选择故障后果更严重的场景将收益更大；而在两个故障后果差不多的场景之间，公式指引我们选择更容易检测出缺陷的那个。在这两

个变量的共同影响下，我可以认为，每一个测试都对应着一个特定的测试收益值。如果要在两个可能的测试中做出选择，那么在其它条件都相同的前提下，我们选择收益最大的那个；当两者的收益无差异时，如何选择也将是无差异的。

当然，测试收益并非是我们评估的唯一要素，对于测试选择来说，它必要但并不充分；因为除了测试收益外，还有另外一个不可忽视的因素，那就是测试成本，正如前文所说的，测试不可能零成本地进行。一般来说，测试成本相对而言比较好理解，因为通常我们将其解释为人月、人天或人时，视之为一种可度量的资源投入。在加入了测试成本这一因素后，我们的分析将更为全面，从效率的角度看，我们希望所选择的测试收益尽可能地大，同时所付出的成本又尽可能地小。因而，我们有可能会在评估可能的测试成本后，放弃掉一些收益可观的测试。在这种情况下，并非是该测试收益不够大，而是因为这一收益在更大的测试成本面前，显得并不值得。测试是一个获取信息的过程，测试能让我们知道得更多，但在有些时候，信息的成本有可能会大于无知的成本。

综合测试收益与测试成本，我们将测试选择演变成了一个关于效率的问题，也就是说我们将选择更有效率的测试，并且希望接下来的每一单位测试成本的投入所产生的收益能够最大化。在进行测试设计时，测试工作者实际上面临的是一个无比巨大的测试集合（因为有无数的可能场景），我们从零开始，不断地从集合中挑选出能让接下来一单位的测试成本投入产生出最大收益的测试用例，之后一直重复这一过程，直到在原集合中再也找不出单位收益能大于单位成本的测试用例。



由于我们总是先选择现存的、能令单位成本收益最大化的测试，因此随着选择的持续与投入的增加，单位成本的收益（dr/dc）必将递减，最终会小于 1，并

从上方穿过成本收益相等线（dr/dc=1）。这是因为，总会有一些场景是收益小于回报的，此时 dr/dc<1。因此，最佳测试投入的选择就在于收益等于回报的那一点，在这一点上，继续往前是得不偿失的，而后退一点则是未充分最大化收益。上述过程描述了我们对测试用例的理性选择，如同测试场景的不可穷尽论指出完备的测试是不可能的一样，测试的效率论认为：完备的测试不仅是不可能的，同时也是不经济的。

测试的评估

在上文中，我们在尽量避免提出这样一种简单的观点：即对测试的选择，实际上就是选择一切测试收益大于执行成本的测试。从结果上看，这一观点与我们前面的讨论似乎是一致的，两者最终都包含了所有收益大于成本的测试，也都排除了一切得不偿失的场景。但是它们之间存在着一个不明显但非常关键的差异：在这个简单的观点中，其核心前提是测试的收益与成本是明确的，同时也是固定的；但在前面的持续选择理论里，却并不要求有这样的前提。尽管将测试的成本与收益看作是明确的与不变的——这一点既符合我们的直觉也吻合了我们的期望，但是这并不是现实。在现实中，测试的收益既不是客观明确的，也不是持续不变的。我们以边界值选择为例，在一个边界值测试方法中，我们通常会选择 min、min+、nom、max-、max 这五种场景，其中 nom 项是随意的，而其它四项是明确的。假如在 1~255 的测试中，我们可以将 200 作为 nom 项，根据“选择一切收益大于成本的测试”的观点，我们之所以选择输入 200 这一场景，必定是因为测试它的收益大于其执行成本。但这样一来，问题就出现了：我们既然认为测试输入 200 是有效率的，并将其纳入我们的选择集中，那么我们又该如何解释像输入 199、输入 201 这些看上去与输入 200 并无差异但却被排除在外的场景呢？因为我们没有任何理由认为它们在测试收益与执行成本上与输入 200 有何不同。

因此，我们有必要将测试的选择看作是一个持续和动态的过程，在这一过程中，每一个测试的收益都不会是固定和绝对的。它之所以不是绝对的，是因为测试的收益通常不会孤立地存在，它取决于我们已经选择了哪些测试。同时这一收益也只存在于选择前，并且随着我们选择的完成而变得模糊。比如一个可以在 4 个浏览器上执行的测试用例，当我们在这 4 个可能的测试场景中做出选择前，每一个场景都有它的收益值，这个收益值可能因具体浏览器的市场份额不同而不同，此时的收益值是当我们只打算从中选择一个场景时的测试收益。然而一旦我们选择了其中一个测试场景后，比如用户数最多的那个，那么受已选择的这个场景影响，其它场景的收益值必将有一定程度的降低。因为场景的测试收益部分地

取决于发现新问题的可能性，而当测试用例已在某一环境中执行后，我们在另一类似的环境中再执行这个测试用例时能发现新问题的可能性已经大大降低，因此收益值必将小于还未选择任何类似场景时的收益值。在边界值问题中也是如此：在我们没选择任何一个 nom 值时，所有的非边界值都是收益大于成本的，然而一旦我们选择了一个 nom 值后，原本还值得测试的那些非边界全部变得得不偿失起来，因为我们很难从它们身上测试出已选的 nom 所不能测试出的新问题来。因此，测试的收益不是固定不变的，在测试的选择这一持续的过程中，随着每一次选择的完成，未被选择的测试的收益将被重新赋值，而被选择的测试的收益也会随决策过程的进行而失去其独立性，变成总收益不可分割的一部分。

当然，测试的收益也不可能是客观明确的。尽管我们在前面已经建立了一个确定的收益公式，但事实上，无论是收益的价值还是收益的可能性，都是建立在预期的基础之上，它必然是一个向前看的或者事前的概念，因而是不确定的。在面临不确定性时，实际决策者对于测试收益的评价可能会不同于任何外部观察者。决策者必须为外部事件赋予主观概率，而这里不存在任何可加以客观决定的概率系数；同时，由于面对的是并不重复的单一事件，因而也没有任何的统计系数可供利用。在这里，我们所运用的数学、所建立的公式，实际上仅仅只是一种直观的语言、一种简洁的表达方式，目的只是用来说明收益评估过程中的关键因果关系。我们对收益的理解和评估，也并不会因为用了数学而更准确或者发现更多。尝试对单一事件的可能性进行“科学的”评估，实际上在许多学科的历史上都有误用这种或然率计算法的记录，这类情形正如约翰•穆勒（John Mill）所说，使其成为了“数学真正的耻辱” [⑥]。可以这样认为，由于存在不确定性，我们对收益的价值评价或多或少都是基于主观的，而这一收益在未来是否发生也是或然的，这种主观性并不会因为我们使用公式表达而变得客观，而其发生的或然性也不会因为数学计算的使用而得以消除。正如诺贝尔经济学奖得主詹姆斯•布坎南（James Buchanan）所指出的：“在一个完全确定的世界里，不存在决策问题，即便真有‘选择’存在的话，那么一台计算机就完全可以全然处理。只有在不确定的世界中，才会有真正的选择” [⑦]。而测试选择就是这样的一种选择。

这些不同的因素都强调了这样一个事实：对测试的选择不存在任何简单的、准确的，同时又是普遍的方法，能让测试工作者可以一学便会、一蹴而就。相反，测试的选择是一个复杂的、主观的过程，我们必须持续地进行的取舍抉择，并且在每一个选择点上，既要前瞻又要后顾，既要分析又要猜测。然而，也正是这些不确定性与复杂性的存在、正是选择的智慧与勇气在这里的不可或缺，使得测试既是科学，又是艺术。

测试的成本

测试的收益与成本共同决定了最终的测试选择，在上文中我们已经完成了对测试收益全方位地探讨，而对于测试成本，我们仅仅只是暂时地将其视为资源投入的成本，这样处理一方面是与人的直觉相符，另一方面也是为了更专注地讨论收益问题。当我们把测试成本用人月、人日或者人时来衡量时，我们实际上是把测试成本看成一种量化的会计成本，一个可以事前预测、事后核算的数字。同时，也意味着我们还将其定义成了一种独立于收益的变量，既不影响收益也不为收益所影响。

然而，成本的概念并没有这样简单，尤其是它与收益之间，存在着某些不同于表面的微妙关系。在经济学中，海狸与鹿的故事有着极高的出现频率:“如果在一个以狩猎为生的国度里，捕杀一只海狸耗费的劳动通常两倍于捕杀一头鹿所耗费的劳动，那么一只海狸自然就应当交换两头鹿，或者说值两头鹿”[⑧]。上述论断来自亚当•斯密（Adam Smith），这也概括了古典的交换理论。正常的或自然的交换价值（也就是收益）由相对生产成本（海狸与鹿 2：1）决定，这回答了古典经济学的核心内容。当然，这一理论是有缺陷的，至少从测试成本的角度来看，我们不能因为测试 A 场景的投入两倍于 B 场景，就说测试场景 A 的收益就应当两倍于场景 B，这显然是谬论。然而，我们却可以说，海狸的成本是鹿而鹿的成本是海狸，或者说一个测试的成本就是因执行它而放弃的另一个测试的可能收益，在弗兰克•奈特（Frank Knight）看来，这一理解是“成本概念中唯一客观和科学的内容”[⑨]。一个商品的成本由可替代的或者舍弃的产品价值来衡量，这就是机会成本思想。因此，可以这样认为，海狸的成本是 2 头鹿，而鹿的成本是 1/2 头海狸，如果海狸的市场价值刚好是 2 头鹿的话，那么是选择捕杀海狸还是捕杀鹿并没有区别。但如果海狸的市场价值小于 2 头鹿，那么人们就不会选择捕杀海狸了，因为这样的选择是收益（1 头海狸）小于成本（2 头鹿）的。

同样，基于机会成本的思想，当银行存款年利是 5%时，一笔投入 100 元、一年后收到 101 元的投资实际上就是不值得的，因为这一收益所面对的成本并非表面上的会计成本 100 元，而是选择了这一投资而放弃的潜在收益 105 元（机会成本）。因此，在测试选择中，我们倘若将成本简单地理解成资源的投入便有可能做出错误的选择，因为成本实际上是做出当前选择而必须放弃的其它选择的收益。也就是说，当一个人说一种特定的测试是“不值得”的，这并不是说其收益相比起对应的资源投入不值得，而仅仅只是意味着他偏好另外一个测试——如果选择另一个测试，收益将会更多。

成本就是我们所放弃的收益，我们在测试选择这一持续的过程中，不断地做出选择——选择收益大于机会成本的测试，一直到最后一个收益大于机会成本的

测试。因为机会成本就是另一可能选择的收益，因此我们又可以将这一过程视为收益与收益之间的比较过程，成本的概念因之淡化，而仅仅是为比较两个选择的收益充当桥梁的作用。这一过程的最终结果，与我们在一系列选择中做出利润最大化的选择结果在大多时候是一致的，因此罗纳德•科斯（Ronald Coase）认为："弥补成本和使利润最大化实际上是表达同一个现象的两种方式"[10]。但是，机会成本的思想能使我们站得更高、也看得更远，使得我们可以超出当前项目，看到更为宽广、更为全局化的选择。因为从机会成本的角度看来，将资源投入重新配置到另一项目或者另一类型的投资也是一种可能的选择。这样，我们就有可能正确地拒绝那些在当前项目看来有效率、但在全局上却并非最好选择的方案。很明显，在现实中，我们争论对某一个软件的测试是过多还是过少时、争论某一测试是必要还是没必要时，往往都是陷入了会计成本的误区。

小结

在软件测试领域里，不存在简单的问题。尤其是当我们不满足于简单方法论，试图寻求一种更普遍更一般化的选择理论时，各因素的不确定性与因果关系的复杂性便随之浮现出来。而对于这些不确定性与复杂性来说，它们既不可能被完全量化，也不可能被彻底消除。我们所能做的，仅仅只是适应与管理这些不确定性与复杂性，而不是假装其不存在。因为我们知道，引入数学和公式并不会从根本上使问题变简单，收益公式中有关或然率的计算，在某种意义上仅仅只是借助数学符号来表示这一有缺陷的知识。它既不扩张、也不拓深，更不补充我们的知识，它只是把这些已知的知识转变成数学语言。就像经济学家路德维希•米塞斯（Ludwig Mises）所说："在评价、选择和行动中决没有什么可度量和可相等的，有的只是等级之差，也即取或舍"[11]。

经过前面的分析，现在我们应该可以轻松回答 Paul Jorgensen 在其经典著作中所提出的"什么时候测试可以停止"的问题了。答案的关键就在于机会成本，机会成本也是一切选择型问题的核心思想。然而，我们也必须清醒地认识到：我们仅能明确地回答"什么时候测试可以停止"，这是一个宏观的、实证的陈述；但我们无法明确地回答"什么时候某一软件的测试可以停止"，因为这涉及到有关收益的主观判断，因而不可能存在完全客观的答案。

注释

[1]. Jean-Paul Sartre 《Existentialism is a Humanism》. 1946.

[2]. 见前一篇《软件测试：哲学视角》.

[3]. Paul Jorgensen. 《Software Testing: A Craftsman's Approach》. 2002.

[4]. Christopher Alexander. 《A Pattern Language》. 1977.

[5]. Warren Thorngate. 《"In general" vs. "it depends"》. 1976

[6]. John Stuart Mill. 《A System of Logic Ratiocinative and Inductive》. 1936.

[7]. James Buchanan. 《Cost and Choice》. 1999.

[8]. Adam Smith. 《The Wealth of Nations》. 1937.

[9]. Frank Knight.《A Suggestion for Simplifying the Statement of the General Theory of Price》. 1928

[10]. Ronald Coase. 《Business Organization and the Accountant》. 1938

[11]. Ludwig Mises. 《Human Action: A Treatise On Economics》. 1949

生生不息的人性化测试-让我们更深入的理解测试

作者：吴晨昊

生命是最重要的。生命是由活力组成的。活力是一切事物的根源，测试保持活力，则能够让整个团队生生不息，保持良性循环。在有活力的团队中，大家是否会都变得身体健康，我想一定会的！

最活跃，最稚嫩的婴儿，它也是最无知的，需要不断的学习，在学习中，得到知识，得到成长。

作为一个测试工程师，我们谁都不会说自己是无知的，因为我们有大学的学历，有工作的经验，有测试的理论。但是，在面对每一次的测试项目，我们该如何测试，这个问题值得考虑，每一次测试的时候，如果能把自己当作一个婴儿，那样才能完美的测试，得到最真实的结果，测试也能生生不息。

以下是我在测试中总结的生生不息的测试理念：

第一， 对于需求要彻底，细致的了解。

第二， 按照需求，逐步尝试，完善测试用例。

第三， 归纳总结，验证自己的想法。

第四， 写出测试感想，和测试文档，比如：测试用例，截图，感受，以便备案，这样下一次遇到同样的测试，就能够作为参考，同时也可以在下一次测试中，丰富这样的文档。 作为测试工程师，写下文档，是非常重要的。

而文档可以放在测试用例中，也可以放在附件中，很多地方都可以。

第五， 就算是，按照如上的步骤进行了万全而周密的过程，还是难保在生产上（实际使用中）会出错，因为这是一个不可避免的现象。由于每一次程序所改的部分，很可能会涉及到其他的模块，而那些相关模块可能非常之小，非常之隐密，对于初次测试这个项目的测试工程师来说，是无法考虑到的。

我们谁也不应该责怪谁，而是应该继续丰富我们的测试文档，在下次测试到相关内容的时候，就应该不要忘记之前的“教训”。

第六， 对待每一个测试工程师，我们应该像是对待一个婴儿一样，进行知识的交流。 测试资料需要共享，以便每个人都能彻底的了解所要测试的内容和其相关的逻辑部分； 在测试中如果遇到任何问题，都应该互相交流，彼此了解大家的个性，合作顺畅。

我们需要通过合作，了解彼此的性格，有的人是外向性格，有的人是内向性格，有的人反应快，有的人反应慢，有的人反应慢而做事仔细。因此彼此的思考方式，都是不同的，这些都是需要长期合作才可以充分了解的。

第七， 在测试中，谁都会犯错，因为人都不会是神仙。我们应该能原谅别

人犯的错误，理解别人，这样也是团队默契的体现。

第八， 如何考核测试人员呢？ 我认为测试人员的考核，是最困难的事，因为大家都在努力，而每个人都在进步。

那些平时认真测试，喜欢分享测试心得，回家还努力学习测试理论的同事，任何分数都无法体现出他的奉献精神，他的执着精神。

那些能够为别的同事因为种种原因而无法按时测试，而奉献自己的宝贵时间，且无怨无悔的认真测试的测试工程师，我们无法给他的分值少于满分。

仅仅按照，开的 bug 数量，仅仅按照，测试后有无严重的错误反馈，等等量化的分值是缺乏人性化的评价标准。

以下是我对测试的理解，也许这最基础的理念，却会透射出，大家视而不见的理念。

例 1：

请写出( )中间的数，谢谢。

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

( ) ( ) ( ) ( ) ( ) ( )

( ) ( )

解这个题恐怕并不难，因为这是一个杨辉三角形，大家都知道。

让我们从第一行开始：

<多次尝试>

首先我们会看第一行是一个 1，第二行是 2 个 1，这 2 行并不能得出什么结论。但是到了第三行的时候，我们发现，1，2，1。此时，我们也看不出什么明显的规律来。

<联想规律>

直到第四行，出现了：1，3，3，1。我们就联想到了这个 3 或许会是 1+2 所得。

<证实规律>

回过头来查看上面一行的 2，它也正好等于第二行的 “1+1”。 为了更充分的表明这个猜想的正确性，我们在看第五行，它的值 1，4，6，4，1 正好都是上面一行中对应的左右 2 数之和。

<运用规律>

从而第六行，第七行的空格就很容易的可以填上去。

可见多次尝试→联想规律→证实规律→运用规律这才是测试人员必备的技能。而最初的多次尝试，就是要从最开始地方做起。

很多人都是在**<多次尝试>**这里停下了脚步。

例 2：科学和测试的关联和区别。

有时候，我们只是想，测试是为了发现 bug，可是这和科学有什么关系呢？

测试的程序是一个不完善的程序，因为还存在 bug，而科学家所要发现的规律是一个大自然完善的规律，如果不是这个规律，那就是那个规律，是寻找正确的程序，也就是说，科学家在寻寻觅觅一个正确的程序，而测试人员是在帮助完善一个正确的程序。

以下是我参照的人们通常使用的推理模式，这些模式都是在《数学与猜想》第二卷中提及。

典型例子：

A 蕴含 B

B 假 | B 真

A 假 | A 更可靠

其中 A 是结论，B 是某个猜想。

如果我们能够找出一个特例否定这个猜想，那么就使结论为假。

如果我们能够找出一个特例支持这个猜想，那么结论就更加可靠。

作为一个科学家，如果我们找到了一个例子证明了这个猜想是假的，那么就会立即放弃这个猜想，转而研究或者寻觅其他猜想。

但是作为测试工程师，我们应该举出所有的能够否定这个猜想的例子，我们的目的是通过修复错误，让这个猜想变成正确的。

一旦我们无法举出否定这个猜想的特例，那么我们就可以认为这个猜想是正确的，而这个猜想也就是我们的需求。

所以，测试工程师和科学家的目的是不同的，但是却有着相似行为，使用更多的例子，套用猜想。

测试工程师→猜想（需求）是正确的，但是结论是错误的，结论不符合需求，因此程序不符合需求。

科学家→猜想（规律）是错误的，需要寻找其他的规律，一直到找到正确的（需求）。

而它的具体实现方法，也就是例 1 中提及的：

多次尝试→联想规律→证实规律→运用规律

以下是一个著名的猜想举例，我想把这个猜想和软件测试结合起来：

F+V=E+2 公式的猜想

我们把凸多面体的面，顶点，和棱的数目分别记为 F，V 和 E。

然后画表格显示

多面体	面（F）	顶点（V）	棱（E）
三棱锥	4	4	6
方棱锥	5	5	8
三棱柱	5	6	9
五棱锥	6	6	10
立方体	6	8	12
八面体	8	6	12
五棱柱	7	10	15
截角立方体	7	10	15
“塔顶”体	9	9	16

如果只看前 5 行数据，我们会觉得 E 是随着 F 的增大而增大的，或者 E 是随着 V 的增大而增大的，但是看第 6 行数据的时候，我们就发现之前的猜想是错误的，E 并没有随着 V 的增大而增大。

对于科学家来说，就会放弃之前的猜想，开始寻找其他的规律。而真正的规律是 F+V=E+2。

对于软件测试工程师来说，在测试开始的时候，总是事先设定了一个需求，而这个需求是一个“真理”一般的需求，也就是说它是 F+V=E+2。

一开始的时候，开发所开发的程序是： E 是随着 F 的增大而增大的，或者 E 是随着 V 的增大而增大的。

所以当测试工程师运行测试用例到八面体的时候，因为不符合程序的逻辑，所以程序报了 Server Error。这时候测试人员开始提交 bug。等开发人员修复完毕之后，当真正正确的程序变为 F+V=E+2 的时候，测试人员运行所有的测试用例，就都能顺利的符合这个程序了。

如此一来，所有的数据都可以完美的证实这个猜想：F+V=E+2。

若有更多兴趣，测试工程师或者科学家还可以使用更多的测试用例支持这个猜想：

多面体	面（F）	顶点（V）	棱（E）
有 n 个侧面的棱柱	n+2	2n	3n
有 n 个侧面的棱锥	n+1	n+1	2n

真理就是需求，而开发的程序没有真正掌握真理，需要测试人员将没有达到真理的地方找出来，并确保程序和真理是一致的。

如果测试工程师并没有运用这个八面体以及其之后的测试用例，那么就永远都无法发现这个 bug，从而我可以认为，测试工程师必须要联想到可能会发生错误的地方，也就是测试工程师必须懂得猜想，和联想，比如说，考虑到是不是只有当 E 是随着 F 的增大而增大的，或者 E 是随着 V 的增大而增大的测试用例？考虑到寻找规律的方法。

因此这些符合测试用例的，有效的等价类，也是有其不同区分的：

所谓等价类，最关键的是“类”这个字。

大家都学习过数学中的类

{x|x=中国三大最有名的高等学校}

也可以表示为

{清华大学，北京大学，复旦大学}

用列举法表示集合{x|0<x<10，x∈Z}

答案是 {1,2,3,4,5,6,7,8,9}

在这个类中的元素就是有效等价类。

我认为，类最重要的是有公共点，也就是有规则。有了规则，就有了满足规则的元素，那些满足规则的元素相互之间就是等价类，而不满足规则的元素就是无效等价类。

而有效等价类也不仅仅是一类，我们可以想象，{1,2,3,4,5,6,7,8,9}虽然都属于有效等价类，但是我们自己可以将其分为奇数，偶数，或者是不是素数，是不是斐波那契数列。这些也是测试人员在整理测试用例的时候，需要考虑到的地方。

例 3：发现规律

现在我想讲一下究竟如何寻找规律，

隐喻-这是一种非常常用的方法。借鉴《数学的发现》这本书我们可以学到。在《数学的发现》这本书中有这么一句话：

发现解法，就是在原先是隔开的事物或想法之间去找出联系。被联系的事物原来离得越远，发现联系的功劳就越大。P169——《数学的发现》

将已知量利用公式求解和未知量相近的公式，将不同的公式进行不断地尝试，找到共同点，一旦都能找到那个相互关联的那个点，那么就是一种闪光点的

存在，于是就可以迎来曙光了。

这样求解成功之后，再回过头来进行重复的查看和演算这样的过程是非常悠然自得的享受。

如果我们有以下三张表，同时，我们有以下的已知条件：

Student No='1'

Class type='C'

Teacher status='0'

题目要求的未知条件是 Teacher No：

A		
Class no	Class	Student No
001	Maths	1
002	Chinese	2
003	Music	3

T	
Teacher No	Teacher status
1	0
2	1
3	0

Relation		
Teacher no	Class no	class type
1	003	A
2	002	B
3	001	C

请写出查询语句：

分析如下：

首先我们会看到通过 Teacher status 可以取到 Teacher no 的值，但是如果直接这么取，那么 Student No，Class type 就没有运用到，所以这是不对的。

通过 table A， 我们可以发现，通过 Student No，我们是可以查询到 Class no 的，所以说我们可以说，table A 是连接 Student No 和 Class no 的。

如果能找到一张表，它只要能触及到 Class no 或者 student no，而且它要能连接到 Teacher no ，class type 这时候我们发现这张表就是 Relation

```
select  distinct  [Teacher No]  from  T  where  [Teacher No]  in
(  select [Teacher no] from  Relation  where  [Class no]  in  (select [Class no] from  A
        where [Student No] ='1' )  and [Class type] ='C')
and  [Teacher Status]  ='0';
```

首先确定一个“主表”，这个“主表”中含有需要查询的属性，也就是“Teacher no”。

然后，我们发现只有 T 和 Relation 这两个表可以作为“主表”。

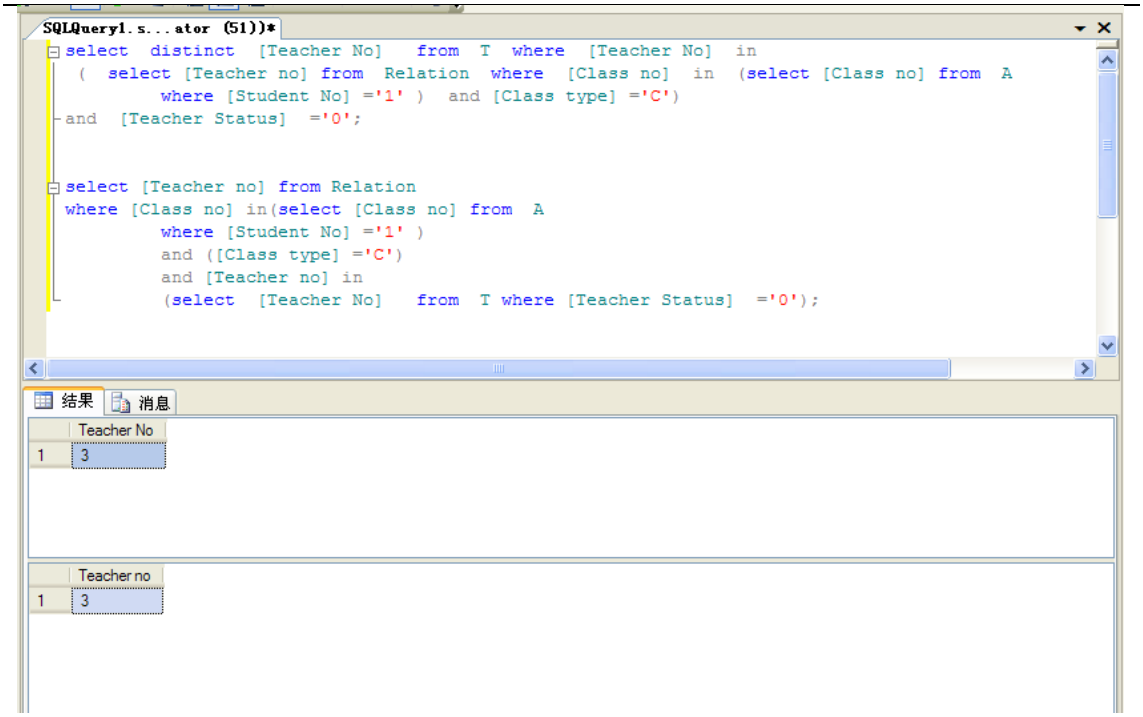
如果用“Relation”表作为主表，也可以通过“Teacher no”，“Class no”，“class type”来链接到所有的已知数据的。

```
select [Teacher no] from Relation
where [Class no] in(select [Class no] from  A
        where [Student No] ='1' )
        and ([Class type] ='C')
        and [Teacher no] in
        (select  [Teacher No]  from  T where [Teacher Status]  ='0');
```

通过以上的 SQL 语句，我们就可以运用已知的条件，查询出题目要求的未知条件是 Teacher No

为了能更好的验证这样的结果，我们可以把数据表，添加到 SQL Server 中，然后运行 SQL 语句脚本。

两种语句运行的结论都是“3”，说明这是正确的猜想。



例 4：对于地铁的猜想和遐想

我喜欢测试，因为测试可以和科学家一样的猜想，我不喜欢被扼杀猜想的能力，猜想的能力就是人的活力，人的活力就是生命，就是快乐。 有时候，坐地铁的时候，我也会有自己的猜想：

上海地铁有两种形式，一种是环线（如四号线），一种是普通的线路（除了四号线之外的线路），环线就是一旦坐上车，就可以不用下车到达四号线所途径的任何一个站点。而普通的线路则只能开往唯一的方向，到了终点站之后，乘客就必须下车。

如果你不小心多乘了一站路，这时候，不管是乘四号线，还是普通线路，依靠乘到底再回乘的方式到达那个站点，都肯定是非常浪费时间的。这时候，聪明的你肯定会选择立刻下车，然后到对面乘反方向的列车。

从这点看，四号线和普通线路是没有区别的，因为地铁不可能随时改变行驶方向。

我们也许会觉得四号线的运行是一个圈，而一号线是一个直线，因此是完全不同的，但是仔细研究就会发现其实它们是有相同之处的：

1）假设一号线有 6 站，那么一号线的一方是从第一站开到六站，另一方是从第六站开到第一站。

2）对于四号线则也是一方是从第一站开到六站，另一方是从第六站开到第一站。

四号线的终点站和起点站是自己定的，而且相隔只有一站路的距离。

或许我们可以认为四号线和一号线是基本一样的，它们的区别是：

第一：它们本身的路程一个就是环形的而另一个是线形的。

第二：或许如果一号线能到了终点站就马上通过时空隧道到达第一站，那就和四号线完全一致了。可惜一号线到达终点之后，那个时间点在起点开始运行的那一辆列车，永远不可能是那一辆刚到达了终点的列车。

人生也许就像一号线而不是四号线，如果人生是四号线，就可以重头再来了。而且还是同一列车。岂不是很完美？因此一号线无法“重头再来”。我把人生就和一号线进行了类比。

以上是我对于地铁的猜想。

是否可以说，人生和一号线是可以在同一个等价类上的呢？

Android 自动化框架浅析

作者：于龙（王超）

笔者在 2011 年的时候接触过一段时间的 android 自动化，那时候 Android 自动化测试刚刚兴起，可用的框架不多，功能也不完善。

三年间 Android 突飞猛进，系统版本从当初的 Froyo2.2 发展到如今的 AndroidL 并细分出了 Android Wear，原有框架得到了长足的改进或是消亡，一些新的自动化测试框架也在这期间出现了。

首先介绍的是它们的鼻祖 Instrumentation。

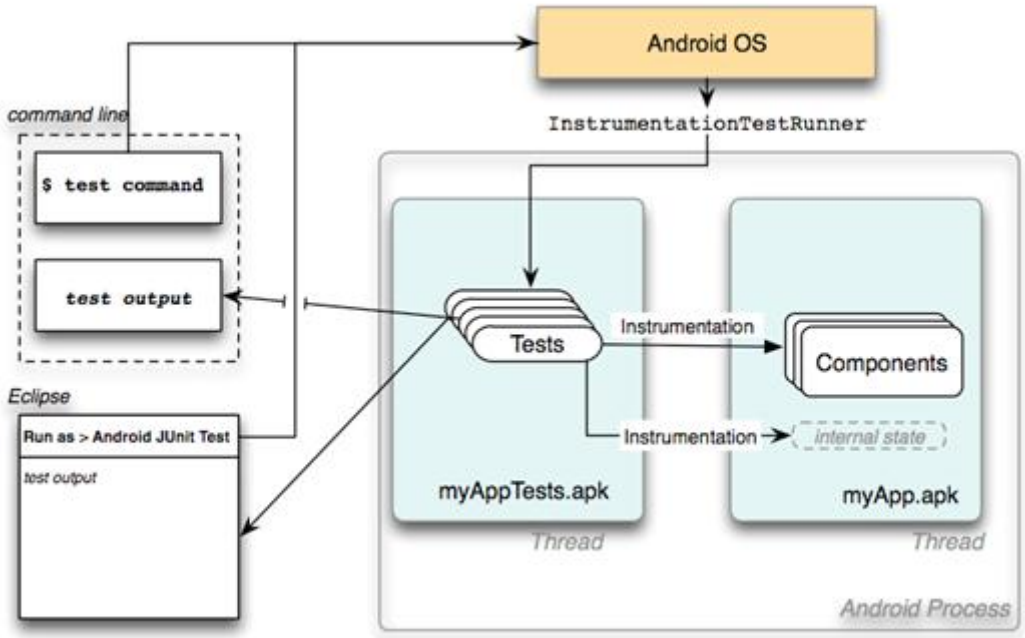
说明：下面有些框架不止支持 Android，有的还同时支持 iOS 甚至 Windowsphone，本文只介绍 Android 的部分。本文不会介绍各个框架的具体使用方法，需要的话可以通过万能的 Google。

1.Instrumentation

Instrumentation 是 Google 官方提供的框架，包含在 AndroidSDK 中，最初的一些框架比如 Robotium 都是基于它进行开发的，所以有必要首先对它进行了解。通过跟踪 Android 的核心组件 android.app.Activity 类的源码，可以发现 android.app.Instrumentation 类介入了 Activity 的启动过程，进一步查看 Instrumentation 类的源码可以发现它具有发送按键事件的功能（更多细节有兴趣的同学可以自行阅读 Android 的源代码）。

Google 正是以 android.app.Instrumentation 类为核心，遵循 Junit 规范建立了 Instrumentation 测试框架，有关代码都包含在 AndroidSDK 的 android.test 包下。该体系比较重要的两个类是 InstrumentationTestRunner 和 InstrumentationTestCase，它们都通过继承或组合的方式使用了 Instrumentation 类。

Instrumentation 框架的特点是需要往设备上安装两个 apk，被测的 app 和包含测试代码的 testApp，要求两个 apk 具有相同的签名，运行测试时，两个 app 运行在同一个 android 进程中，因此 Instrumentation 和所有基于它开发的框架都无法做到跨应用的测试。比如说在测试支付宝客户端的时候调用起了系统的浏览器，那 Instrumentation 是无法对浏览器进行操作的。下面是官方的示意图：



2.Robotium

项目首页：https://code.google.com/p/robotium/

Robotium 是最早的开源 Android 自动化测试框架之一，也是发展至目前用户最多，社区活跃度最高的。

它基于 Instrumentation 开发，其核心技术是通过反射获取了当前界面上所有的 View 对象（有兴趣的同学可以查看 Android 源码 com.android.internal.policy.impl.PhoneWindow.DecorView 这个内部类，同时可以了解 Android 的 View 体系），并以多种 API 的形式提供出来，方便在编写用例时获取、操作 View 对象。

与 Instrumentation 一脉相承，基于 Robotium 编写的 case 遵循 Junit 规范，而且让笔者不得不吐槽的是，到目前为止，Google 在 Android 里面集成的还是 Junit3。

可以通过扩展开发支持 Junit4，下面的章节会介绍到。

Robotium 的主要特点：

API 很易于测试人员使用，与 WebDriver 体系面向对象的 API 不同，它提供了一套面向过程的 API，贴近测试的步骤。

支持对 WebView 的操作。现在比较流行的一种 App 开发模式是在应用内大量使用 WebView，集团内部的淘宝、支付宝客户端都可以见到这种模式，这种开发模式的最大好处是一些频繁变化的内容只需要重复发布服务器端即可，不需要用

户升级客户端。Robotium 这个特性还是非常实用的。

使用方便，除了上面提到的 API 之外，用户只需要将其 jar 包含至测试工程即可，使用简单足以让很多人选择使用它。

3.Nativedriver

项目首页：https://code.google.com/p/nativedriver/

Nativedriver 是由 Google 日本的一个团队开发，也是早期的框架之一，和 Robotium 的命运不同，它目前应该是已经停止维护了。

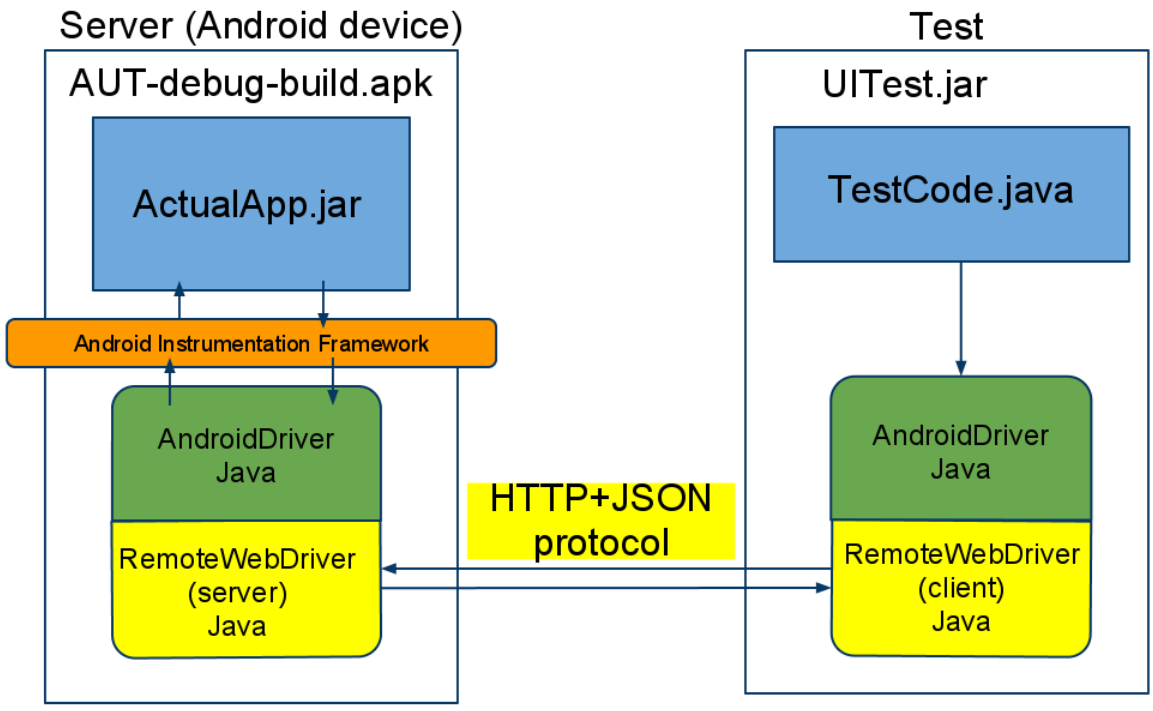
这里介绍它的原因是它的设计思想为后面其它一些框架所用，也算是薪火相传吧。

与 Robotium 相同，Nativedriver 也是基于 Instrumentation 开发，所不同的主要是两点：

采用了 C/S 架构，这种相对于 Robotium 的复杂带来的直接好处就是在编写用例的时候可以摆脱 Instrumentation 体系的束缚，从中解放出来，可以使用 Junit4 或者 TestNG 来编写测试用例。更进一步，这种架构它对 case 端屏蔽了自动化底层的依赖，理论上可以用任意编程语言编写 case，也可以用多种方式实现 Server 端的自动化。

它提供的 API 遵循了 WebDriver API 的规范，是一种面向对象的风格，对于使用过 selenium 的同学来说会更容易上手。

下面是 Nativedriver 的官方架构图：



可以清楚的看到 NativeDriver 分为 Server 端和 Client 端，其中 Server 运行在移动设备端也就是安装有被测应用的一端，PC 端也就是运行 case 的一端为 Client。

Client 端将用户编写的 case 通过 http 协议发送至 Server 端，Server 解析之后再通过 Instrumentation 来执行这些指令，进行实际的测试操作。

事物都是具有两面性的，NativeDriver 的缺点也是由其优点带来的，它使用起来比较复杂（具体流程请参看官网），稳定性也有待提高。

虽然 NativeDriver 已经停止维护了，但是它的这种设计思想却流传了下来并在最后要介绍的 Appium 中发扬光大了。

4.Monkeyrunner

Monkeyrunner 也由 Google 官方提供，在 Android2.3 版本加入，内置于 AndroidSDK 之内。

不同于 Instrumentation，它是站在系统的层面进行测试，是通过 adb 命令来实现对设备的操作，测试脚本通过 adb 让设备执行指定的操作，不足之处是点击的是坐标点，因此面对 Android 严重的碎片化问题，这个工具的实用性大打折扣。在 Android4.0 及以上版本可以通过传入 view 的 id 来进行测试，但是需要模安装有 debug 版本系统的设备，通常是模拟器。因为涉及到另一款工具 hierarchyviewer，它只能在 debug 版本的系统上运行。

Monkeyrunner 的最主要特点是使用 Python 来编写测试脚本。

关于 Monkeyrunner 也没必要介绍更多了，有兴趣的同学可通过 Google 查找相应的资料，因为 Android4.1 及之后的版本，我们有更好的选择，那就是下面要介绍的 UIAutomator。

5 .UIAutomator

UIautomator 是 Google 在 Android4.1 的时候发布的一款全新的 AndroidUI 自动化测试工具，同时一起发布的还有 uiautomatorviewer ，一个图形界面工具来扫描和分析应用的 UI 控件，可在编写测试用例时辅助定位需要的 View。

它同时具备了 Robotium、Nativedriver，Monkeyrunner 的一些优点：

- 可以跨应用进行测试
- 可通过多种方式找到需要的 View
- 面向对象的 API
- Google 亲儿子

它的缺点主要是：

- 只能用于 Android4.1 及以后的版本上

- 调试不方便，目前只能通过打印日志的方式，所以需要开发的配合才能发挥最大作用
- 无法测试 Webview 内的内容

使用略复杂，需要将包含测试代码的工程打成 jar 包，push 到安装有被测应用的设备上，然后通过命令行运行。

6.Appium

官网地址：http://appium.io/

项目地址：https://github.com/appium/appium

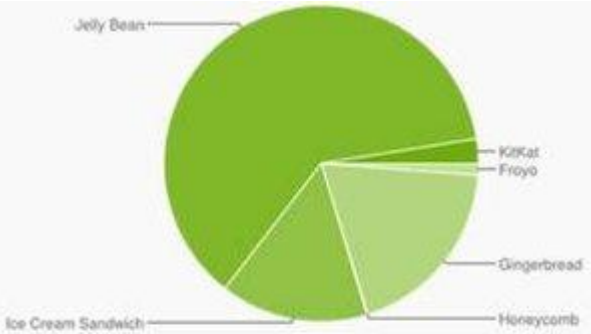
Appium 是基于 WebDriver 规范对 UIautomator 的一次扩展开发，这正是 NativeDriver 的设计思想。

Appium 集成了另一款框架 selendroid 来保持兼容，从而将支持的最低版本扩展至 Android2.3，并支持了 WebView。

selendroid 可以看做是 NativeDriver 和 Appium 之间的产物，三个都是遵循 WebDriver 规范对 Google 开放的自动化接口进行扩展开发，这里不详细介绍（官网 http://selendroid.io/）。

考虑到目前 Android 个版本的市场占有率（下图来自 Google 今年三月发布的统计数据），Appium 可以说是覆盖了绝大部分的测试需求。

Version	Codename	API	Distribution
2.2	Froyo	8	1.2%
2.3.3 - 2.3.7	Gingerbread	10	19.0%
3.2	Honeycomb	13	0.1%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	15.2%
4.1.x	Jelly Bean	16	35.3%
4.2.x		17	17.1%
4.3		18	9.6%
4.4	KitKat	19	2.5%



Data collected during a 7-day period ending on March 3, 2014.
Any versions with less than 0.1% distribution are not shown.

以下文字摘自 Appium 的宣传：

Appium was designed to meet mobile automation needs according to a certain philosophy. The key points of this philosophy can be stated as 4 requirements:

You shouldn’t have to recompile your app or modify it in any way in order to

automate it.

You shouldn't be locked into a specific language or framework to write and run your tests.

A mobile automation framework shouldn't reinvent the wheel when it comes to automation APIs.

A mobile automation framework should be open source, in spirit and practice as well as in name! 还是非常霸气外露的，这四点相信是每个设计和使用自动化测试框架梦寐以求的

7.总结

目前主流的开源框架从内核上主要分为两大派系：基于 Instrumentation 的，和基于 UIautomator 的。换句话说，如果 Google 大神没开放接口，大家都没得玩。而 Appium 正是这两大体系的集大成者，功能最为强大，唯一的缺点就是使用起来比较麻烦。下面是几个主流框架的对比：

框架	Case 语言	跨应用	更改被测应用	WebView	兼容性
Robotium	Java	不支持	不需要	支持	所有版本
NativeDriver	Java	不支持	需要	不支持	Android2.2 及以上
UIAutomator	Java	支持	不需要	不支持	Android4.1 及以上
Appium	任意	支持	不需要	支持	Android2. 3 及以上

对 Instrumentation 体系的扩展

笔者今年接到需求，需要建立商户 App 自动化回归体系。

一个自动化回归体系通常分为两部分：

- 自动化测试框架
- 自动化任务管控体系 考虑到商户自动化的急切需求，笔者对 Appium 不熟悉的实际情况。这里选择了简便易用的 Robotium 而不是 Appium。 而自动化任务管控体系，这里直接使用已有的 AQC 平台中的 wauto 应用。

笔者并没有对 Robotium 的源码做出任何修改，而是对 Instrumentation 体系的 Junit3 做了扩展，使得它支持 Junit4 的用例书写方式。

将 Android 自带的 Junit3 扩展至 Junit4

这里也没有重复的造轮子，没事多逛逛 StackOverfolw 总是有好处的，因为我们遇到的问题，上面很有可能已经早有人遇到了。

这里使用了一款开源的项目，它的核心思想是对 Instrumentation 体系中的 InstrumentationTestRunner 类做了扩展开发，从而达到支持 Junit4 的目的，当然实现远没有一句话这么简单，项目主页 http://github.com/esmasui/AndroidJUnit4。扩展至 Junit4 的好处就是可以在此基础上遵循 Junit4 规范进行进一步的扩展，比如编写我们自己的 Runner。

笔者这里扩展了 Junit4 的 BlockJUnit4ClassRunner，让编写用例时可以使用数据驱动，从而编写以下形式的 case：

```
@Desc(name = "测试收银界面", project = "收银用例")
@Test
@DataProvider(location = "DataFile.csv")
public void testCash(String userName, String password) {
//test code
}
```

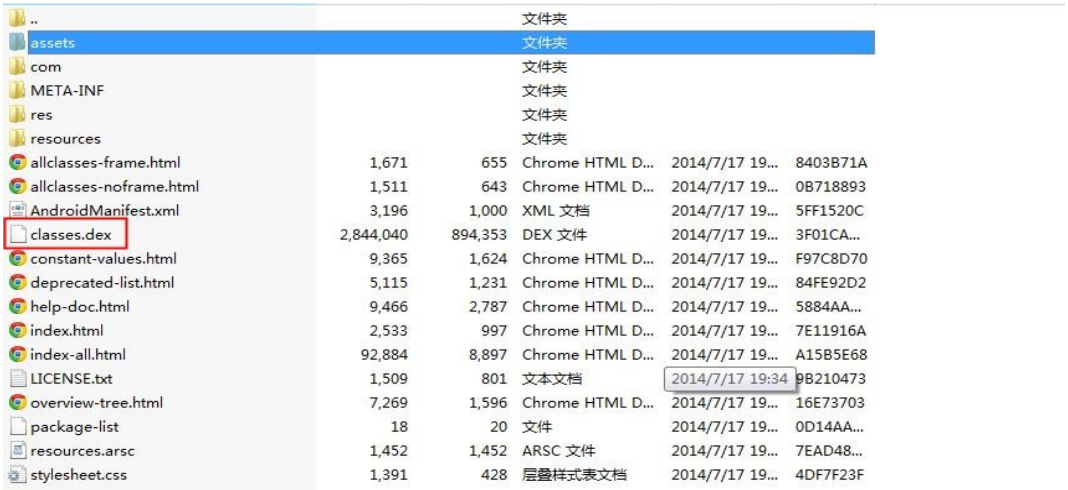
这里的技术难点就是包含测试数据的文件（这里以 csv 为例，也可以使用 xml）放在哪个目录，如何读取的问题。

按照 Android 的惯例，笔者将 csv 文件放在了 Android 测试工程的 assets 目录下面，后面问题来了，如何从 assets 目录中读取文件呢。

如果是在 Android 应用工程里面直接调用 Activity 的 getAssets().open(String fileName)方法即可。

但是这里是 Android 测试工程，那就只有使用普通读文件的方式了。

考虑到 assets 下面的文件都会原封不动的打包至 apk 文件中，apk 文件的目录结构如下：



所有的 java 文件都编译成 class 文件，最终打包成 android 特有的 dex 文件。

dex 文件与 assets 文件夹处于同一目录，查阅 Java 的 API，最终使用 Class

类的 getResourceAsStream("/assets/" + fileName)读取成功。

AQC 自动化管控体系

AQC 自动化管控体系包含以下概念：

1.Project，在这里对应一个测试工程

2.Testcase，测试工程中包含的每一条 case

3.Plan，针对测试工程建立的测试计划，一个 Project 可以建立多个 Plan，包含以下信息：

- 多条 Testcase（Project 中包含 Testcase 的子集）
- 运行时间，结束时间信息
- 分配的设备
- 运行策略

4.PlanRecord，一个 Plan 一次运行的记录，一个 Plan 可以运行多次

5.Result，一条 case 在一个 PlanRecord 的运行结果

6.PlanDevice，一个 Plan 分配的设备

首先建立 Project 然后是 Plan，可以通过 wen 界面和 api 请求两种方式建立。

Plan 建立完毕之后就会将自己添加至 AQC 平台统一的任务调度系统，在指定的时间启动 Plan，在 Plan 指定的设备上运行指定的 case，Case 的运行结果会保存至 AQC 的 Result 数据库中。

商户自动化实施

整体流程

商户采用 API 请求的方式建立 Project 和 plan，整体流程如下：

- jenkins 定时打包，同时打出商户 Apk 和测试 apk
- jenkins 调用 AQC 提供的 API，将两个 apk 的地址以及其它参数传入 AQC
- AQC 根据参数建立 project，plan
- AQC 任务调度系统启动 plan 运行
- AQC 收集测试结果 ####碰到的问题 商户接入 AQC 的一个难点是商户 App 的测试环境是通过另外一个 App 来设置的，如果每次测试需要预先在设备上安装设置的 App 并进行设置显然配不上自动化这三个字。

解决方案

通过分析设置 App 和商户 App 的代码，发现商户 App 中使用到的各个服务器 url 默认写死为线上环境，在应用初始化的时候会尝试先从一个 ContentProvider 中获取。

此 ContentProvider 包含在设置 App 中，底层实现是 sqlite 数据库，设置 App 提供界面来修改 ContentProvider 中的值，这样每次重新启动商户 App，就能使用最新的设置了。

笔者的解决方案是将在 AQC 中提供类似远设置 App 的界面，可预先设置多组配置。

每一组配置对应一个测试环境，包含一组商户 App 各个服务器的 url 地址。AQC 在启动测试后，会根据传入的参数读取指定的一组配置，然后包含在 adb 命令中发送给测试工程。

另一方面，将 ContentProvider 移植到测试工程中，同时在测试工程中包含一个 Activity，这样测试工程也同时是一个 Android 应用工程。

然后在安装完毕测试工程 apk 后，通过 adb 命令启动 Activity，Activity 会在 onCreate 方法中将 adb 命令包含的参数写入 ContentProvider 中，设置完毕之后 finishActivity。

这里使用 Activity 而不是 Service 的原因是，adb 启动 Activity 的命令适用性更广，有些设备不支持 adb 启动 Service。

AQC 在每台设备上执行测试的流程

- 清理设备上可能已安装的商户 Apk 和测试 apk
- 全新安装两个 apk
- 根据参数读取商户服务器 url 配置
- 将 url 配置包含在 adb 命令中，发送至设备上的测试工程
- 通过 adb 命令启动 Robotium 测试
- 如果遇到 Robotium 的 crash，最多重新执行 3 次
- 每运行完一条 case 将结果存入 Result 数据库
- 设备分配的 case 运行完毕之后，释放设备 目前运行结果展示

  http://aqc.alipay.net/wauto/resultList.htm?planId=2040&planRecordId=11244

结束语

随着 Appium 的发展和成熟，相信将无线自动化框架迁移至 Appium 上是一个大的趋势。笔者也在准备进行这方面的工作。