HZ BOOKS
华章科技
PEARSON

本书是Dorothy Gramham、Mark Fewster、Lisa Crispin、Alan Page等30多位来自世界各地的自动化测试专家和大师的经验结晶

通过对30多个来自世界各行业的经典测试案例的分析和研究讲述了自动化测试的工具、技术、方法，以及自动化测试的实施、引导和管理，包含大量最佳实践和反面教训

华章程序员书库

Experiences of Test Automation

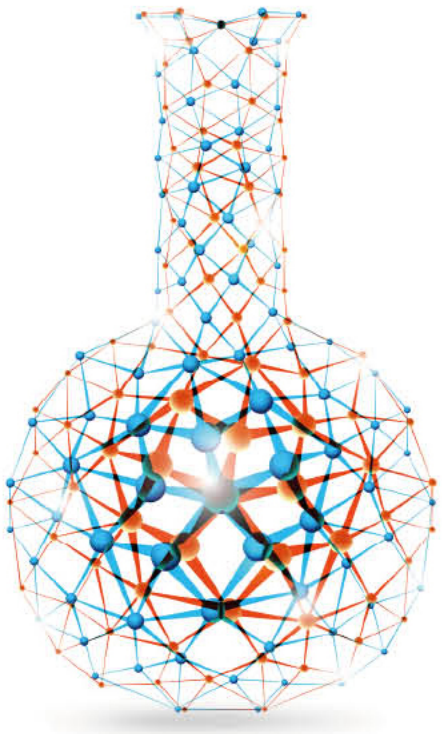
Case Studies of Software Test Automation

自动化测试最佳实践

来自全球的经典自动化测试案例解析

（英）Dorothy Graham　Mark Fewster　著

朱少民　张秋华　赵亚男　译



机械工业出版社
China Machine Press

自动化测试最佳实践：

来自全球的经典自动化测试案例解析

Experiences of Test Automation: Case Studies of Software Test Automation

（英）Dorothy Graham Mark Fewster 著

朱少民　张秋华　赵亚男　译

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

自动化测试最佳实践：来自全球的经典自动化测试案例解析 /（英）格雷（Graham，D.），（英）福斯特（Fewster，M.）著；朱少民，张秋华，赵亚男译 . —北京：机械工业出版社，2013.3
书名原文：Experiences of Test Automation: Case Studies of Software Test Automation

ISBN 978-7-111-41676-0

Ⅰ. 自…　Ⅱ. ①格…　②福…　③朱…　④张…　⑤赵…　Ⅲ. 软件—测试—自动化—案例　Ⅳ. TP311.5

中国版本图书馆 CIP 数据核字（2013）第 039386 号

本书是世界名著《Software Test Automation》的姊妹篇，是自动化测试领域的至宝，是每一位测试人员都应该反复研读的一本书。

它是 Dorothy Gramham、Mark Fewster、Lisa Crispin、Alan Page 等 30 多位来自世界各地的自动化测试专家和大师的经验结晶，通过对 30 多个来自世界各行各业的经典测试案例的分析和研究，讲述了最新的自动化测试工具、技术、方法，以及自动化测试的实施、引导和管理，不仅包含大量最佳实践，还包含很多失败的教训。让读者不仅能吸取前人留下的宝贵经验和远见卓识，少走弯路，还能更好地领会自动化测试的自然规律。

机械工业出版社（北京市西城区百万庄大街 22 号　　邮政编码　100037）
责任编辑：谢晓芳
　　印刷
2013 年 4 月第 1 版第 1 次印刷
186mm × 240mm • 28.75 印张
标准书号：ISBN 978-7-111-41676-0
定价：89.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

本书赞誉

“你手中的这本书是关于自动化测试的不可多得的至宝，它讲述了在自动化测试的实践中什么是好实践，什么不是。本书使你避免陷入自动化测试的泥潭，从而更可能取得成功。”

——Linda Hayes

“在本书中，Dorothy Graham 和 Mark Fewster 通过一系列引人注目的案例研究展示了自动化测试中的各种学习经历，每个案例研究都讲述了自动化使用的工具和方法。本书内容如此全面，是同类书中第一本带读者深入到自动化测试世界中的书籍。本书包含了许多案例研究，案例研究讲述了在自动化过程中都发生了什么，案例包含的项目来自各行各业，测试环境包括了各种技术环境。通过找出各个案例之间的类似点以及对某些主题的重复，能够帮助读者更加理解应该避免哪些错误。通过本书，读者可以了解到需要做些什么才能取得测试自动化的成功。

——Andrew L. Pollner，ALP International Corporation 的总裁和 CEO

“在畅销书《Software Test Automation: Effective Use of Test Execution Tools》[⊖]出版之后很多年，Mark Fewster 和 Dorothy Graham 又写出了一本畅销书。敏捷方法已经赋予自动化测试在当今测试实践中的主导地位。本书从各种不同视角讲述了自动化测试的案例研究，这些案例写得都很好，这使得本书非常有实用价值。我强烈地将本书推荐给那些从事或者想要从事自动化测试的人。”

——Erik van Veenendaal，Improve Quality Services 的创始人，TMMi Foundation 的副主席

“阅读本书就像参加一个测试学术会议一样，书中包含了多个案例研究和对自动化测试深刻的见解。但是本书要比参加学术会议便宜得多，而且还不需要到处奔波去参加会议。令我印象尤其深刻的是，本书在第 0 章中浓缩了我能想到的使得自动化成功的各个方面。这是在学术会议中得不到的。

——Hans Buwalda

“本书包含了大量令人兴奋的、写得很好的、涉及范围极广的案例研究，案例讲述了在现实世界中自动化测试的经验、技巧、教训和其他值得记住的要点。对于任何需要向经理和同事证明在自动化实践中什么是好的、什么是不好的人员来说，本书都是非常有用的。”

——Isabel Evans，FBCS CITP，质量经理，Dolphin Computer Access

⊖ 该书后面统称为《Software Test Automation》。——编辑注

“本书首先讲述了使得自动化测试有效的基本手段。之后讲述了各种情境下的自动化测试中值得注意的事项。本书会指导你：因合理的缘由而应用自动化测试，如何采取适合于公司和项目的自动化方法，以及如何避免各种错误。对于任何参与测试的人——无论是管理人员、测试人员，还是自动化测试人员，本书都非常有价值。”

——Martin Gijsen，自由的测试自动化架构师

“Fewster 和 Graham 为我们提供了连接自动化测试理论与现实之间的一座重要桥梁。自动化测试的框架设计和实施是一种非精确的科学，亟须一套可复用的标准，而这套标准只能从不断涌现的先例中总结出来，而本书则能帮助建立这样的先例。就如同在司法系统中使用先前判决案例作为支持当下做出法律判决的依据一样。在自动化框架设计与实施上，从本书所能习得的各式案例，适用于帮助人们做出当下的决定，支持这样的活动或教育相关人员。[1]

——Dion Johnson，Automated Testing Institute（ATI）软件测试顾问及首席咨询师

“即使我一贯秉持‘自动化测试无用’的立场，本书也的确让我驻足思考。它让我开阔了思维，同时也让我做出‘噢，原来这种情况我没考虑到’的反省。对于那些想要参与自动化测试的公司，我推荐将本书作为入门参考书。”

——Audrey Leng

“本书是一个惊人的成就。我相信它是自动化测试方面写得最好的书之一。Dot 和 Mark 通过对 28 个案例研究的叙述给予我们一个完全崭新的概念，包括引人注目的小窍门、真知灼见以及经验教训。这些案例研究来自于生活经验中，既有成功的，也有失败的，包括了自动化的多个方面、不同的环境以及多种混合的解决方案。书本来就是智慧之源，而作者采用了非常好的方式——利用叙述故事的形式给我们留下很深的印象，从而增强学习效果。无论读者处于何种层次，本书是所有想要进入或者已经进入自动化测试领域的人所必备的。它的确是同类型书中独一无二的。”

——Mieke Gevers

译 者 序

时光荏苒，转眼间本书的翻译工作已经进行了半年多，算是没有辜负出版社的期望，按时完成翻译任务。当初，看到本书的英文版，就有翻译本书的强烈愿望。本书作者 Dorothy Gramham 和 Mark Fewster 之前写的《Software Test Automation》就很有影响，作为其姊妹篇，本书一定会更胜一筹。更让我感兴趣的是本书的组织结构和众多引人入胜的故事。

本书的作者实际不只是 Dorothy Gramham 和 Mark Fewster 他们两位，而是 30 多位来自世界各地的自动化测试的成功探索者，包括 Lisa Crispin、微软的 Alan Page 等测试大师。本书可以看做是这些自动化人员的自传，每一章自成体系，一章一个案例，进行深入的剖析，30 多个案例形成了自动化测试的一顿大餐。这些从事自动化测试多年的测试人遍布全球，分布在美国、英国、德国、印度、荷兰、挪威、丹麦、比利时、加拿大、澳大利亚等地，同我们分享他们在自动化测试过程中所经历的酸甜苦辣，其中有过迷茫，有过失败，也有喜悦和激动。自动化测试的故事，有浪漫之旅，也有历经坎坷、披荆斩棘才走完的艰辛之路；更有凤凰涅槃、绝地逢生，迎来自动化测试春天的感人故事。这些自动化测试之旅，短的有一年，而长的会跨越六年、持续十多年，甚至像《星球大战》那样，有前传，还有后传。每个故事都是他们亲身经历的，让我们感同身受。

本书就像是一本写实的小说，里面有很多个性鲜明的人物。如喜欢将还没有通过各项测试的正在开发中的版本向客户展示的经理，也有偶然间因为某些突然冒出来的想法从开发转向测试的技术人员；既有人认为自己作为测试人员不应该写代码，也有经理认为能写代码的都是程序员，都应该去开发部门等；既有自己做了很伟大的事情却因为公司制度以及测试衡量标准的原因导致没有能够升迁的技术人员，也有为各种软件公司提供咨询的专家。当我们阅读本书的时候，仿佛自己也参与到了这些项目的自动化测试之中。当看到项目所取得的惊人成就时，一种自豪感会油然而生；当看到应用自动化导致项目越来越糟时，自己也会感到迷茫，不知何去何从；在突破层层困难，拨开云雾见天日之时，自己心中便会有“柳暗花明又一村”的顿悟；当自己辛辛苦苦搭建起来的框架和之前无数个日日夜夜的努力，却因为管理层的原因导致项目中断时，不禁会感到彷徨与无奈。

本书展示了丰富的自动化测试对象，除了经典的政府信息系统、企业 ERP 系统、通用的数据库系统之外，还包括从主机到 Android 移动应用，从汽车电子系统、医疗设备到设备仿真、硬件接口等的自动化测试。而且，本书自动化测试所处的环境也是千差万别，所涉及的项目之广也是同类书籍所不及的：

- 有涉及传统的 V 模型，也有涉及敏捷的开发模型；

- 有基于关键系统（如北大西洋东部领空的空中交通控制系统）的自动化测试，也有一般商业系统（如保险公司）的自动化测试；
- 有基于模型的自动化测试，让我们领略数学严谨的同时，见证低成本高收益的自动化测试；
- 有针对自由的猴子测试、探索式测试来实现其自动化方式，让手动测试和自动化测试相辅相成；
- 有针对云端实施自动化测试、在线的持续回归测试，将基于产品的自动化测试变更为基于服务的自动化测试。

在翻译本书的同时，译者深深地感受到了自动化的魅力和力量。测试自动化在节省了大量人力的同时，也在推动整个软件行业的前进。正如本书所述，正确地采用自动化能够极大地提升软件的质量和测试的效率。然而在错误的地方应用自动化只会带来更大的开销。本书呈现了一些失败的案例，目的是告诉读者，单单技术方面的因素并不能决定自动化测试的结果，管理方面的因素也要考虑到其中。如自动化框架很好、人员积极性很高，并且已取得了不少积极的效果，但是却因为其他一些无法抗拒的因素最终走向了失败。在很多故事中，我们可以看到作者反复强调：“一些经理本身不懂测试自动化，他处在经理这个位置更多是像被强插在这个位置的。”这在现实中并不少见，公司应该对经理进行培训，使其可以促进自动化测试的良好发展，而不是作为阻碍因素。本书可以为这些经理提供各种真知灼见，指导他们如何促进自动化的发展，也可以指导技术人员如何斗智斗勇，从经理那里获得应有的资源，同时尽量从自动化测试启动时就让经理们也参与其中，并通过不断地向他们展示自动化进度和成果来获取其支持。

正如一句广告词“简约而不简单”，这句话用来描述本书恰到好处。每个章节都用一种平易近人的语气来讲述，故事好像一个接着一个地发生。但是作者想要表达的事情并不简单，作者不断地去找寻各种方式，向读者呈现一个个故事背后的必然性和偶然性，并且强调什么样的实践是好的，什么是坏的。作者的一些观点在本书中多次出现，如“自动化测试并不便宜”、“不要忽略管理因素”、“自动化测试是对手动测试的补充而非替代”。作者反复强调这些，就是要警示读者，在自动化领域有很多经验教训，我们应该对此有着清晰的认识。这些观点一遍一遍被反复强调，警示着我们不要重蹈覆辙。在阅读本书时，除了了解和掌握自动化测试的各种应用方法、技术和工具之外，还会对自动化实施及其引导、管理等方面的各项影响因素有所明察，从而杜绝虚幻、脱离实际的想法。看完本书，我们能更好地领会自动化测试的自然规律，吸取很多前人留下的宝贵经验，增长见识，使我们自己少走弯路，采用更有效的自动化测试方式和方法，以完成我们的测试使命。

最后，感谢机械工业出版社华章公司引进了这么优秀的英文原著，并给予了我们大力支持，使得本书的中译本能够和读者见面。我们还要感谢天津策艺公司柴阿峰、迈瑞血球研发中

心肖利琼、宁波大学刘慰等三位友人以及同济大学郭智超、俞诗嘉等两位同学。他们从百忙中抽出宝贵时间，帮忙校对，进一步提高了翻译质量。

希望本书能成为一种有效的助推力量，让更多的测试人员参与到自动化测试的实践中来，促进国内自动化测试百花齐放、百家争鸣。使自动化测试不再是奢侈品而成为软件测试的必需品，惠及千千万万的软件项目，最终促进国内软件行业的发展。

译者

于美丽的同济大学嘉定校区

序

自动化测试——集“圣杯”、“青春之泉”、“点金石”为一身。在过去的几十年中，测试者为了能从手动测试——构建测试实例与测试数据，设置系统先决条件，运行测试，比较实际数据与预期数据并报告可能的缺陷——的苦海中脱离而不断追寻自动化测试。自动化测试能够简化所有这些操作，甚至还能简化更多的测试操作。

然而遗憾的是，成功有效并且经济实用的自动化测试往往很难实现。自动化测试项目大多在启动后便步履蹒跚、时常跌倒，最终迷失路途，被遗弃在不断增长的废弃项目堆上。

自动化的失败可以归咎于很多原因，其中无法实现预期效果或许是最常见的原因，还有就是不充足的资源分配（例如时间、人员和资金）。其他因素包括：难以满足需求的开发工具，阻碍工作质量的欲速则不达的急躁情绪，缺乏对自动化测试的理解——自动化测试也是一种软件开发，需要用与软件开发一致的专业方法来进行。

Dorothy 和 Mark 在 1999 年出版的著作《Software Test Automation》，为同类书籍设定了标准。该书的第一部分详细列举了在大多数成功的自动化实例中发掘出的方法——脚本技术、自动比较、测试件架构和有用的衡量标准。第二部分讲述了一些机构在实现自动化测试过程中所吸取的经验教训。现在，又经过了 10 年在行业内的摸爬滚打，Dorothy 和 Mark 提供了另外一些机构和个人的实践经历，用以指导自动化作业。本书同时描述了经典的和最新潮的自动化测试方法。每一章通过叙述某一独特自动化实例的一个故事（既有成功的也有失败的），给我们提供指导。

某些主题在本书中再次出现：合理、可以达成的目标；管理层支持；衡量标准，包括 ROI（Return On Investment）；需要的技术；规划；设定期望；建立关系；工具；培训；以及政策——所有的一切都是成功的测试自动化所必需的。然后，这些主题同样也可运用在项目及个人层面。本书一个非常大的好处在于拓展了测试自动化领域，从更全面的角度来考虑这些主题。

我第一次见到 Dorothy 和 Mark 是在 1998 年的“慕尼黑 Euro STAR 软件测试会议”上。他们对自动化测试的认知以及帮助其他人完成自动化测试的热情，给我留下了非常深刻的印象。我真心祝贺他们取得如此出众的成就，并向大家推荐本书。

——Lee Copeland

前　言

用于软件测试自动化的工具已经发展了近 30 年，但还是有很多自动化测试的尝试会失败，或者至少并非那么成功，这是为什么呢?

在笔者的前一本书《Software Test Automation》中，提出了一些有效的自动化测试原则，为了验证这些原则今天是否依然有效，以及是否还有其他的原则可以应用于自动化测试中，所以我们开始收集现实世界中自动化测试实施案例。结果很高兴地发现：在过去 10 年中，许多人在软件测试自动化领域取得了很大的成功，其中不少人参考了我们的书。当然，我们并不是唯一描述或发现了这些好的自动化实践的人，然而找寻一些成功的而且能在一段时间内都适用的实践在今天看来还是难以实现。我们希望本书中的故事会帮助更多的人在自动化测试的尝试中取得成功。

本书收集了当前的很多关于自动化测试的真实故事。自从我们上一本关于自动化测试的书在 1999 年出版以来，自动化测试技术有了很大的飞跃。我们想要找出什么样的方法是成功的、哪种应用程序正在通过自动化测试来完成测试的，以及近几年来测试自动化是如何改变的。不同的人会用不同的方式去解决自动化相关的问题——我们想要知道我们从他们的经历中能学到什么，以及在什么样的情形下如何用新的方式应用自动化测试。

本书中的案例研究包括一些成功的方法，同时也包括一些不成功的方法。本书旨在教会读者如何避免错误，以及如何从实际成功的例子里汲取经验教训。本书旨在帮助读者从其他一些专家真实的经历中学到大部分自动化测试的经验。

本书中的案例研究主要是关于测试执行过程中的自动化，其他一些自动化的类型在一些章节中也会提及。我们着重研究系统级的自动化（包括一些用户验收测试），尽管其他的一些章节也会包括单元测试或集成测试。我们将在不同的应用程序、环境和平台下描述测试自动化，每章用到的工具包括商业的、开源的以及公司内部的程序，这些工具可以用在传统开发或者敏捷开发项目中。我们很惊奇地发现有近 90 种商业或者开源软件正在使用，这些软件的名称列在附录 A 中（附录包含了每章作者用到的各种工具，不仅仅是测试用的工具）。

本书描述的实例都是真实的，虽然在一些案例中没有透露作者或者公司名称。我们鼓励这些案例研究的作者去描述到底发生了什么，而不是仅仅给我们一些通常的建议，所以本书的描述是相当真实的。

在收集这些实例的过程中，我们也遇到了很多困难，比如，我们很难涉及各种行业中的应用。那些在公司里坚持不懈地致力于发展自动化测试的人给我们留下了深刻的印象。然而，我们也遇到了他们中的许多人遇到的一些困难，这些困难有时候会导致测试失败。相信本书中讲述的经历会帮助你在自动化测试道路上更加成功。

案例研究的附加内容（我们得到的经验）

本书并不是一系列文章的简单组合，我们和每一章的作者一起工作，以便将一些最有价值的信息传达给读者。我们对于每一章的审校都是非常全面的，我们通常进行很多轮审校，提出问题及改动建议（特别感谢每章作者的耐心和他们传达的额外信息）。我们最开始收集的文档超过了 500 篇，所以这里每一章的内容都是精挑细选的。

我们会提供一些真知灼见、经验教训和小窍门，帮助读者从本书中尽可能汲取大部分知识。每一章都有我们的一些评论，这些评论包含了一些我们认为非常有用的要点。仔细注意以下这些有用的标记：

真知灼见（good point）：值得注意的观点（即使它们并不是那么新）。

> **真知灼见**
>
> 管理层的支持至关重要，但是期望必须是符合实际的。

经验教训（lesson）：从失败中学到的教训——我们最好不要做的事情。

> **经验教训**
>
> 和软件开发一样，自动化开发也需要进行规范化。

小窍门（tip）：一些看起来新的、有趣的诀窍，可用来帮助我们解决特定的问题。

> **小窍门**
>
> 使用一个“转换表”记录一些可能发生改变的事情，这样自动化可以使用一些不变的标准术语。

在你读每一章的时候，我们会用一些感叹词来提醒读者重要的地方，比如，“注意这里”，“看这里”或者“这会非常有用”。

如何阅读本书

每一章都是独立的，读者能够以任何次序阅读各章节。本书的章节安排是为了让那些从头读到尾的读者更好地体验自动化测试的各个方面。

为了帮读者确定读各章的顺序，请看表 P-1，其中总结了各章的特点。这张表让读者能够初步了解各章包含的针对特殊应用的工具和开发方法等，读者也能快速找到和其最相关的章节。

在“前言”之后，第 0 章描述了全书概况，并且对书中关于管理层和技术的问题发表了我们自己的观点（以及自己的测试件体系结构的图例）。第 0 章将各作者在自动化过程中涉及的或与任务相关的建议重点提炼出来。这一章是本书的“行动纲要”（executive summary）。

第 1~ 28 章讲述的是多个案例研究，每一章都是由一名或多名作者撰写，在这些章中，作者主要讲述了：他们做了什么，哪些方案可行性良好，什么方案出现了问题，以及学到了哪些教训。一些章涉及了非常专门的信息，如文件结构和自动化代码；而其他章讲述的内容则更加通用。有一章（第 10 章）的内容来自于《Software Test Automation》这本书，我们更新了其内容，其他章节都是新写的。

第 29 章可以单独看成一本迷你书——它包含了一系列短篇故事，取材自几十个人身上发生的真实故事，故事的长度从半页到几页不等，每个故事都很有趣，并且包含了很多有用的信息。

在本书的最后，附录包含了在文章中提到过的工具，这些工具有些是商业工具，有些是开源工具。

致谢

本书包含的资料来自于我们认识的、在会议中遇见的那些人，还有一些人是在看到了我们发布在网站和实事通信上找寻软件测试自动化方面作者的材料后，响应并投稿的。2009 年秋天，我们开始了本书的编写工作，从 2009 年 12 月到 2011 年 12 月，我们两个人在本书上花费了 850 多个小时，这还不算所有章节和奇闻轶事的作者所花费的时间。

感谢所有章节的作者，感谢他们花费大量时间和精力写作，并且在本书编写的整个阶段都的我们保持联系。同时感谢那些参与第 29 章写作的作者：Michael Albrecht、Mike Bartley、Tessa Benzie、Jon Hagar、Julian Harty、Douglas Hoffman、Jeffrey S. Miller、Molly Mahai、Randy Rice、Kai Sann、Adrian Smith、Michael Snyman、George Wilkinson 和 Jonathon Lee Wright。

感谢所有在本书编写过程中给予帮助的人：Claudia Badell、Alex Blair、Hans Buwalda、Isabel Evans、Beth Galt、Mieke Gevers、Martin Gijsen、Daniel Gouveia、Linda Hayes、Dion Johnson、Audrey Leng、Kev Milne、Andrew Poller、David Trent 以及 Erik Van Veenendaal。同时感谢参与各章审校的作者。

感谢 Pearson 出版团队，感谢他们的帮助和鼓励，以及为出版这本书所做的工作，他们是：Christopher Guzikowski、Raina Chrobak、Sheri Cain、Olivia Basegio 和自由职业者 Carol Lallier。

——Dorothy Graham
Macclesfiled，英国
www.dorothygraham.co.uk
——Mark Fewster
Llandeilo，英国
www.grove.co.uk
2011 年 12 月

表 P–1

章　号	作　者	应用程序领域	地　点	生命周期	项目人员数量 / 人
1	Lisa Crispin	金融、Web	美国	敏捷开发	9~12
2	Henri van de Scheur	数据库	挪威		3~30
3	Ken Johnson, Felix Deschamps	企业服务器	美国	带有敏捷因素的传统开发周期	>500
4	Bo Roop	测试工具	美国	瀑布模型	12~15
5	John Kent	从大型主机到基于 Web 的系统	英国	传统开发周期	40
6	Ane Clausen	两个项目：保险业和养老金	丹麦	没有特定模型，敏捷开发	3~5
7	Elfriede Dustin	政府：国防部	美国	敏捷开发	几百
8	Alan Page	设备驱动程序	美国	传统开发周期	几百
9	Stefan Mohacsi, Armin Beer	欧洲航天局的服务	奥地利、意大利和德国	传统开发周期	>100
10	Simon Mills	金融：保险业	英国	混乱的模型，而且一直在变动	几十
11	Jason Weden	网络设备	美国	传统开发周期（瀑布模型）	25
12	Damon Yerg（笔名）	政府服务	澳大利亚	V 模型	几百
13	Bryan Bakker	医疗设备	荷兰	V 模型	50
14	Antti Jääskeläinen, Tommi Takala, Mika Katara	Android 平台上的智能手机应用程序	芬兰		2
15	Christoph Mecke, Melanie Reinwarth, Armin Gienger	ERP 系统（SAP），两个项目：卫生服务系统和银行系统	德国和印度	传统开发周期	10
16	Björn Boisschot	能源部门的 SAP 应用程序	比利时	传统开发周期	12

案例研究特性

时间跨度	工具类型	是否进行试点研究	ROI 测量	是否成功	是否还在运行
1 年，6 年后进行报告	开源工具	否	否	是	是
5~6 年	内部工具	否	否，但是效率提升了 2400 倍	是	是
约 3 年	商业工具和内部工具	否	否	是	是
1 年两个月	商业工具	否	否	否	否
23 年	商业工具	是	否	是	是
6 个月、1 年	商业工具	否 是	否 是	否 是	否 是
4 年半	商业工具、开源工具和内部工具	是	是	是	是
9 年	商业工具，内部工具	否	否	是	是
6 年以上	商业工具、开源工具和内部工具	否	是，4 个周期后收回成本	是	是
15 年	商业工具	否，但开始时规模很小	否，但现在每月运行 500 万个测试	是	是，客户群仍在增长
3 年	内部工具	否	否	最终成功了，是	是
11 年	内部工具	是	否，但是计算了与手动测试相比带来的好处	是（有起伏）	是，项目很兴盛而且还在继续发展
1 年半	商业工具、开源工具和内部工具	开始小规模进行	是	是	是
6~8 个月	商业工具、开源工具	整个项目都用于试点研究	否	是	是
4 年、2 年	商业工具，内部工具	否	否	是	是
6 个月	商业工具	是	否	是	是

章　号	作　者	应用程序领域	地　点	生命周期	项目人员数量 / 人
17	Michael Williamson	基于 Web 的系统，分布式系统	美国	敏捷开发	15
18	Lars Wahlberg	金融市场系统	瑞典	增量开发到敏捷开发	20（通常情况下）
19	Jonathan Kohl	各种系统，从 Web 到嵌入式	加拿大	敏捷开发和传统开发周期	几个 ~60
20	Albert Farré, Benet,Christian Ekiza Lujua, Helena Soldevila Grau, Manel Moreno Jáimez, Fernando Monferer Pérez, Celestina Bianco	4 个项目，都是医疗软件	西班牙、美国和意大利	螺旋式开发，原型开发，瀑布模型	2~17
21	Seretta Gamda	保险业	德国	迭代开发	27
22	Wim Demey	定制的软件包	比利时	传统的 V 模型	
23	Ursula Friede	保险业	德国	传统开发周期（V 模型）	30
24	John Fodeh	医疗应用程序和设备	丹麦	传统开发周期（V 模型），增量模型	30
25	Mike Baxter, Nick Flynn, Christopher Wills, Michael Smith	空中交通控制	英国	传统开发周期	15~20
26	Ross Timmerman, Joseph Stewart	嵌入式：汽车系统	美国	分阶段的瀑布模型	8
27	Ed Allen, Brian Newman	基于 Web 的、移动的、桌面的、社交渠道的（语音、聊天、email）	美国	传统的开发周期	28
28	Harry Robinson, Ann Gustafson Robinson	针对电话系统的错误报告系统	美国	瀑布模型	总共 30 人，4 人直接参与项目

（续）

时间跨度	工具类型	是否进行试点研究	ROI 测量	是否成功	是否还在运行
6 个月	商业工具，开源工具	是	否	否	否
约 10 年	开源工具	是	是，项目用于每日、每周、每月的测试，并且收回了成本	是	是
不同的项目周期不同	商业工具、开源工具、内部工具	是，在一些案例中进行了试点	否	是	是，一些项目现在仍在使用
5 年、 2 年、 几个月、 1 年	商业工具 内部工具 商业工具 商业工具	否	否	是 部分成功 否 是	是 是 否 计划中
12 个月	商业工具、内部工具	是	否	是	是
4 个月	商业工具、开源工具	是	否	是	是
约 6 个月	商业工具	否	否，但是基本上每次发布节省约 12 万欧元	是	是
6 年	商业工具、内部工具	是	否	是	是
每个周期约 3~12 个月	商业工具、开源工具和内部工具	是	否	是	是
5 年	带有商业硬件的内部工具	否	否	是	是
1 年	商业工具、内部工具	否	否，但衡量了因此带来的好处	是	是
1 年半	内部工具	否	否	是	否

目　　录

第 0 章　案例研究反思

成功的自动化测试需要智慧和毅力。你的经验可能和本书所描述的有一些相似之处，但每个人的故事都是独一无二的。通向成功的道路并不简单，但是正如书中案例研究所描述的那样，自动化测试已经在各种应用领域、各种环境和项目的各个生命周期中取得了成功。

通过思考，我们根据书中出现的案例和奇闻轶事总结出了一些方法。本章高度总结了现有经验中需要吸取的最重要的教训。你可能会在阅读完后面各个案例研究章节后再次阅读本章。

哪些主要因素促成了自动化测试的成功？导致自动化测试失败最常见的因素有哪些？

对这些问题没有简单而通用的答案，但存在一些公共的要素。我们认为最重要的两个要素是管理问题和测试件架构：

对自动化的管理层支持，包括设置切实可行的目标，以及提供足够合适的资源来取得已计划的 ROI。

一个好的自动化测试件架构，拥有正确的抽象层，在降低自动化测试件维护和自动化测试各个方面成本的同时提供灵活性和适应性。

除了共同的要素外，还有其他一些方面，甚至是一些令人吃惊的要素，将它们也考虑进去可以帮助你在自己的自动化测试中取得更大的成功。这是我们写这本书的希望和目的！

在以下的大多数小节中，我们会着重强调一些章节号，并讨论一些特定的主题。这些主题可能也在其他章节涉及，我们会在这里列出各章针对某一特定主题进行的专门讨论。

我们首先在 0.1 节讨论管理层问题，但经理们也需要注意 0.2 节描述的技术问题。

0.1　管理层问题

从许多案例研究中可以清晰地了解到，管理层的支持力度关系到自动化测试的成功与失败。举例而言，第 4、6、11、17 和 20 章都叙述了管理层支持欠缺导致自动化测试失败的情形。

0.1.1　自动化测试目标

制订一个合适的目标对自动化测试的成功实施至关重要！这似乎是显而易见的，但令人惊讶的是，在没有任何清晰目标或仅仅是只有一些模糊的陈词滥调作为目标（如“更快的测试”、“做得更好”、“节省时间”）的情况下，一个自动化测试就开始了，这类事情经常发生。目标越具体，自动化测试越有可能得到好的评价并取得成功。

将软件测试所要达到的目标与自动化所要达到的目标区分开来是很重要的。自动化是运行测试的一种方法，无论这些测试是好还是坏。一个好的测试目标是发现许多 bug。这没有必要

成为一个好的自动化测试目标：在近期对项目进行一些改动之后，需要进行回归测试以确保变更的部分不会影响系统的行为，而这类测试很少发现新的错误。这并不意味着自动化测试不成功，只是因为它有了错误的目标。如果一个高层经理感到自动化测试没有达到预期目标（即使这个目标是一种误导），那么资金也可能被撤掉。

0.2.9 节讨论了一些能够有效地找到 bug 的自动化测试示例。一些好的自动化测试目标在第 1、2、3、6、7、10、11、12、14、20、21、25、27 和 29 章中讨论。

0.1.2 管理层支持

在一个无法对自动化测试进行精心培育并有效引导的组织中，自动化测试很难成功发展起来。对于个人来说最好先亲自试验一下，如果想要大规模地进行自动化测试，并能收获自动化测试对于产品最终发布所带来的巨大好处，则需要管理层的大力支持。一种“自下而上”的方法并不是通向良好自动化测试的一条长期可持续的道路。

管理层支持对自动化测试的成功至关重要；我们可以在本书的很多案例研究中看到这一点。管理层支持包括多种形式：对工具的资金支持，对于一个试点项目和测试件架构的开发过程的资金支持，以及为此需要付出的时间，管理层还要有兴趣去理解自动化行为以监督成本与回报（参见 A.3 节的 ROI）。

对于经理来说很重要的一点是，对于自动化过程他能够提供什么，以及对要达到期望的结果需要付出时间与努力有着很明确的了解。

在某些案例中，一些高层的经理并不十分了解好的自动化测试意味着什么。这可能是因为他们没有很好地亲自调查，但另一个因素可能是做自动化测试的人员没有积极沟通，虽然沟通是他们本应该做好的事情。

与管理层沟通的重要性主要在第 1、4、6、13、20 和 29 章中进行强调，而具体作法则在第 16、19 和 29 章中涉及。管理层支持作为示例学习的关键因素在第 1、2、6、11、18、21 章中进行了强调。

0.1.3 ROI 和度量标准

一个普遍的误解是，成功的自动化测试仅仅需要购买相应工具的投资（如果你得到一个开源工具，那就不需要其他任何花费）。如果在自动化方面的投资为 0，则其投资回报率可能为负值。简而言之，如果你什么也不投入，你将会陷入麻烦！

要在以下方面进行投资：对于观点的研究和实验；设计和开发一个好的自动化测试件架构；学习和了解失败和成功的因素；找到符合特定情况的解决方案；就自动化测试计划、进度及测试方法进行沟通。

常常在一个自动化测试开始时评估 ROI。这是一个很明智的做法：相对于自动化测试的投入，是否获得了更多的收益并节约了资金？执行自动化测试的理由通常是比较执行同一测试手动和自动所花费的时间。尽管这是证明自动化测试有用的方法，但仅仅执行自动化测试并不是

全部。实现自动化测试的时间，对自动化测试的维护，分析错误测试的时间以及其他一些可能比手动测试花费更多时间的任务。这些其他任务可能是一个重要的额外开销，也应该考虑。要知道如果你使用一个工具供应商的 ROI 计算器，这些其他的开销可能不包括在 ROI 的计算中。

其他需要考虑的因素包括：更高的覆盖率（已经测试过的系统数量），最少的时间推向市场，以及增长的信心。但这些益处在早期的自动化测试实施中可能无法实现。它们会在自动化测试一旦完成之时变成现实。它们同样难以量化，因此也可被看做额外的收益。

一旦一个自动化测试建立，看它能否达到预期也是非常重要的，因此最好定期做这样类似的比较（将所有因素考虑进去），并且将这一信息和负责资金的管理层交流，这是非常重要的。

许多人混淆了收益（benefit）和 ROI。收益只是收益，而 ROI 则是将收益与成本有效地进行比较。

请记住，你决定收集的度量标准可能被误解，也许它没能传达你所希望的意思。也请注意，那些在新的上下文中不再有用的度量标准；第 28 章描述了自动化测试的职业生涯的影响。

ROI 和量化收益在第 1、2、9、12、13、18、23、26 和 29 章中讨论。在第 9 章有一个用来评判投资的基于模型测试的 ROI 计算器的范例。

0.1.4 在敏捷开发中的自动化测试

随着敏捷开发变得更加普遍，其自动化测试也变得越来越重要。持续的集成就是测试自动化；回归测试每天都要进行，有时候甚至更为频繁。自动化测试也需要在出现变更时能做出响应，就像敏捷开发中那样，于是测试件架构便显得更加至关重要（参见 0.2.1 节）。自动化测试在传统开发和敏捷开发中都取得了成功，没有自动化测试，敏捷开发就不会成功。

敏捷技术，如测试驱动的开发（Test-Driven development，TDD）可以确保自动化的单元测试，但是对使用敏捷方法开发的系统仍需要进行系统测试。第 1 章主要讲述的是敏捷开发中的自动化测试；敏捷项目中的自动化测试也在第 6、7、17、18、19 和 21 章中提及。

0.1.5 技能

测试人员所需要的技能和自动化测试人员不同，自动化测试人员的角色可以看做测试人员和工具之间的桥梁（参见 0.2.1 节）。

自动化测试人员的角色有着严格的区分：一类是高层次的自动化设计人员（测试架构师，test architect），另一类是自动化测试件的实现人员，自动化测试人员（test automator）可以用于这两类人员。自动化测试架构师的任务是设计自动化测试的整体结构，或者为了创建好的测试件架构而选择框架，或是将现有框架进行改进以适应新的需求。自动化测试人员的任务是设计、编写、维护自动化测试的软件、脚本、数据、期望结果以及额外的实用工具。自动化测试人员负责实现多个层次的抽象（在 0.2.1 节中讨论），这使得测试人员不必学会编程就可以使用这些自动化手段。自动化测试人员同样可以帮助测试人员，包括解决技术上的问题、决定花费 / 收益的比率，以及实现新的测试需求等。自动化测试人员必须有好的编程基础。

有这样一种趋势，即测试人员同时也是开发人员。如果测试人员同时还是一名好的程序员，那么这个人既可以成为测试人员也可以成为自动化测试人员——我们讨论的是不同的角色，他们不一定是不同的个人。

然而，很多测试人员并不一定擅长技术，而且他们也不想成为程序员。就如 Hans Buwalda 说的，强迫一名非技术的测试人员去做程序员，不仅会使你失去一名好的测试人员，还会让你得到一名不称职的程序员。非技术的测试人员也应该参与自动化测试！如果将他们从自动化测试的过程中排除，就不能充分发挥这些测试人员的技能，从而也无法充分挖掘出自动化测试的潜力。

测试人员和测试自动化人员的角色在第 10、12、18、19、20、23 和 29 章中讨论。

0.1.6 计划、范围和期望

当自动化测试有好的计划时，它通常更有可能获得成功。计划应当包括，在自动化测试应用于整个项目前花些时间进行一些实验。例如，一个限制时间的试点项目，让你能更清晰地看到如何达到自动化测试长期目标的方法，当然这个项目也要有清晰的目标和足够的资源。

不要期望自动化项目中不会出现任何问题；没有什么事情是没有问题的！应该随时准备应对可能出现的问题。记住即使是最好的计划也仅仅是一项指导——要根据情况去重新审视这些计划。

设定符合实际的目标，即在规定时限内完成任务，并且定义好项目的范围。不要太过于关注细节，否则就无法在公司中获得那些潜在的收益。要关注在早期就能得到的一些有用的结果，而不要以减少有用的自动化测试为代价，去构建过量用于支持测试复用的库。一旦自动化测试开始运行，要继续寻找方法去改进这些自动化测试，并且为自动化测试设立新的目标以减少花费和增加收益。

第 1、5、6、11、16、20、25 和 29 章讨论了这些问题，第 1、3、23 和 27 章还讨论了如何持续地改进自动化测试过程。

0.1.7 和开发人员的关系

在成功的自动化测试实践中通常有一项因素，即在测试人员、自动化测试人员和开发人员之间保持良好的关系。如果他们的关系不好，那么自动化过程就会更加艰难，即便自动化测试在最后还是会带来一些收益。如果软件在设计时没有考虑到自动化，那么自动化测试就会变得非常困难。例如，如果软件使用了非标准的控件，那么自动化测试就很难和软件进行交互。测试人员或者自动化测试人员或许会对开发人员说："请仅仅使用标准的控件——这会使得我们的工作更加容易。"但是开发人员的时间很仓促，可能会说："我们为什么要做一些不会给我们带来好处的事情呢？"这个开发人员并非很无理，这确实是一个相当合理的原因（虽然一些测试人员不会同意这个观点）。

更好的方法是告诉开发人员自动化测试是如何让他们受益的，并且和他们保持良好的合作

关系。例如，如果你能够在 15 分钟内在测试环境中对一个新编码的函数运行测试，你就能够在一定时间内向开发人员提供有用的信息（找到了 bug 或者测试已经通过），这对他们是极大的帮助。

关于开发人员和自动化测试人员的关系在第 1、2、6、9、17、20、27 和 29 章中讨论。

0.1.8 促进项目改变并启动自动化测试的触发器

是什么让一家公司决定它们应该采用自动化测试？有时，因为测试不足所带来的严重问题或者近期的灾难，促使公司做出改变；有时，来自公司外部的观点也会带来更好的解决方案；有时，管理层决定公司需要自动化测试，这甚至决定了公司的生存。人们总是先尝到了苦头，然后才会进行实际的改变。

对于那些打算应用自动化测试的公司，最重要的建议是要从小范围开始应用。试点项目（pilot project）是个好主意，在将自动化测试扩展到更广的范围之前先在小范围内尝试不同的方法，来判断哪种方法最好。

这些问题在第 1、9、10、12、17、23、26、27 和 29 章中讨论。

0.1.9 工具和培训

人们经常会问一个问题，哪个工具是最好的？这就像问哪个汽车最好一样。一个人认为最好的车是能够容纳四个小孩和两条狗的车；另一个人可能更关注于车的速度和性能；还有一些人关注哪个车更经济一些。因此，没有完美的工具，但是有很多工具是足以应对某些特定场景的。

事实上，正如第 17 章讲述的那样，有时可能会选择错误的工具，因此为所要做的工作选择合适的工具是很重要的。在第 17 章中错误使用的工具却在第 7 章和第 25 章中取得了成功。

但是工具不是测试自动化最重要的因素。是的，通常确实需要使用工具来执行测试，但是在大多情况下，好的自动化测试的其他方面远远比因单独工具之间的差异所带来的影响要大得多。拥有好的工具不能保证在测试自动化中取得成功——必须对整个测试框架进行良好地计划、定制和维护，工具仅仅是一小部分。

使用一个工具失败并不意味着使用其他工具就能取得成功；一些公司尝试了一些工具但是在每次尝试中都以相同的方式失败了。遗憾的是，公司经常将这些失败归咎于工具或者个人，而实际原因在于自动化测试项目没有进行足够的计划和管理。

工具的主要用处是为人员提供支持！那些将要使用工具的测试人员应该在如何使用这些工具上有话语权，而且公司应当为工具提供基础设施以支持他们。

无论使用什么工具，培训都是很重要的。那些将会直接使用工具的人应该在早期就接受一些深入的培训，无论是通过工具生产商的课程或者在线教程。如果公司引进组织外部的顾问或者工具生产商的技术支持人员来进行培训，将每次会议的间隔日期进行适当调整，以便在这段时间中测试人员能够吸收这些知识并且对他们所学到的内容有实践的时间。之后，为那些需要

使用这个自动化工具的人提供培训，告诉他们自动化测试应该如何进行——这是内部培训，而不是外部培训。良好的培训能够避免浪费很多时间。

关于工具和培训相关的问题会在第 1、6、11、12、13、18、19、21、23、25、26 和 29 章中讨论。

0.1.10 行政因素

在自动化过程中，有一些因素是测试人员、测试自动化人员，甚至是经理或者其他利益相关者无法控制的；例如，可能会因为一个主要项目的取消，导致为成功的自动化测试付出的努力也变成徒劳。

很多测试人员和自动化测试人员之所以艰难地进行一些项目，可能仅仅是因为他们经理的一句看起来随意的话。第 29 章中的奇闻轶事就举了这方面的例子，还有在第 4 章中，经理的行为“谋杀”（虽然这可能是“过失杀人”，而不是“谋杀”）了自动化测试。第 28 章举了一个例子，即当自动化测试带来的改进是如此巨大，以致经理都不肯相信这些结果。

行政因素是生活的一部分；做出的决定并不经常像看上去那样合理。

0.2 技术因素

在我们看来，最重要的技术因素是测试件架构以及多个层次上的抽象。测试件（testware）是所有创建的用于测试的事物，包括脚本、数据、文档、文件、环境信息等。测试件架构就是这些事物是如何组织的，以及它们彼此之间是如何依赖的——例如，高层次的脚本使用了低层次的脚本用来与被测软件进行交互。

0.2.1 抽象、抽象、再抽象：测试件架构

在最初得到了一个测试执行工具后，通常会期望测试人员开始使用这个工具。当这名测试人员不是开发人员时会发生什么？现在突然需要他们成为程序员，去学习工具的脚本语言。如果他们没有编程背景，他们就不知道如何构建坚固的（自动化）代码，于是当对被测软件进行改动，同时这项改动也需要在自动化代码中进行时，维护他所写的代码可能就要付出很大的代价。通常，公司放弃一个测试执行工具是因为测试脚本维护费用很高！

对于成功的自动化测试执行来说，（至少）有两个主要的抽象层次：将工具和工具特定细节与结构化的测试件分离开来；将测试（即测试人员所从事的工作）与结构化的测试件分离开来。这些内容会在 0.2.1.1 节和 0.2.1.2 节中进行更详细的描述。

自动化测试人员负责实现这些层次的抽象；如果这些层次实现得不好，那么自动化维护起来就会很昂贵而且很难使用。图 0-1 展示的是我们认为好的测试件架构，其中还包含了为了达到这些层次的抽象自动化测试人员所扮演的角色。

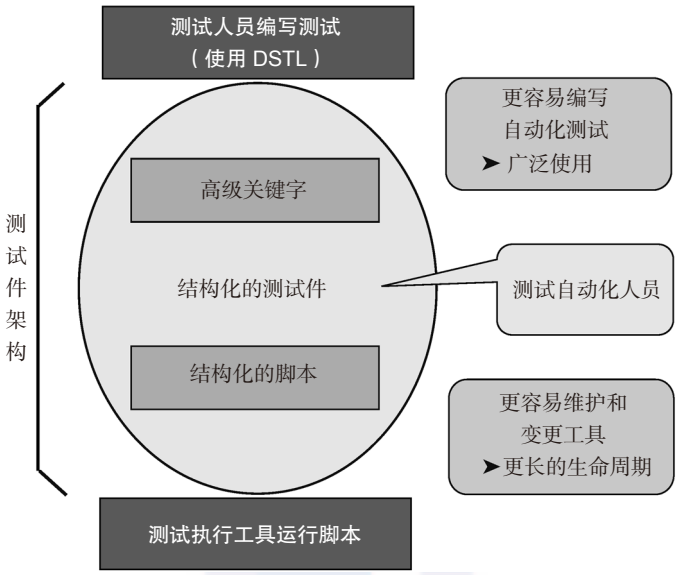


图 0-1　抽象的层次

0.2.1.1　将测试执行工具与测试件分离开来

一个好的测试件架构能够让测试自动化发挥真正的力量。自动化测试架构师负责设计测试件的架构：脚本、数据文件和实用工具等，这些都是自动化测试所需要的。其他的自动化测试人员在之后必须对其进行补充和完善。这里需要好的编程实践：良好的结构，遵守一些合理标准的模块化代码，这些标准鼓励代码的简易性和复用性。当软件的一些方面有所改变时，也许测试件需要进行改动，但是一个好的架构会让这些变动最小。

可以通过框架来实现测试件架构，框架可能是测试执行工具的一部分，也可能和测试执行工具分开；或者也可以通过分开的测试控制程序或者测试解释器来实现测试件架构。

有时人们在用户界面（UI）对象（如按钮和编辑框）周围使用包装器（wrapper）。包装器是个好的开始而且对于好的自动化测式来说也很有必要（但是并不足够）。为 GUI 对象的名字使用对象映射（object map）也是另一个用来实现合适层次抽象的例子。

另外一个重要的方面是较高级的测试件不应该直接与任何特定的工具或者工具语言绑定在一起。当然，在工具接口上，脚本必须是以选择的工具语言所编写，但是这仅仅是对于最底层而言；结构化的脚本应该以与工具无关的方式编写。你也许会认为，当前使用的工具很完美，为什么还要这么做？但是随着时间的推移工具会发生变化，应用程序也已意料之外的方式发生改变，系统可能要迁移到其他环境或者平台上，于是 3 年前看起来完美的工具现在看起来并不那么合适。如果测试件和工具紧紧地绑定到一起，那么很有可能你最终会不得不抛弃测试件的很多部分。如果达到了这个层次的抽象，那么就可以保留结构化的测试件的大部分，仅仅需要改动最底层的那些与工具相关的脚本。

这个层次的抽象能让自动化测试有着更长的生命周期。通过使用正确层次的抽象，无论是

在应用程序上进行了改动，还是在当前使用的工具发生了变动，依然可以使得自动化测试继续进行下去。在本书提到的所有成功案例研究中都在一定程度上实现了这一层次的抽象。

0.2.1.2 将测试与测试件分离开来

顶层（高层）的抽象是将测试的基本想法（即对什么进行测试）与测试的实现（如何构建自动化测试）分离。测试人员使用对他们有意义的高级（领域相关的）关键字编写测试，以描述他们想要运行的测试。

这一级别的抽象对于那些没有编程技能的测试人员是最重要的。他们的专业技能可能是有关业务或者应用程序的。如果是程序员在开发测试，他们会更加理解这些测试技术是如何实现的，于是这层的抽象对他们来说就没有那么重要；这些对于开发人员最合适的高级关键字可能对那些业务用户来说过于技术性。但是不要忽略这一点，即对于那些编写和维护测试的人员来说，易于使用这些测试是很重要的，而不管他们是不是程序员。

测试人员的工作是测试（见 0.1.5 节中关于技能的部分）。在关注如何最好地对一些事物进行测试时，不应该考虑过多的实现细节。他们应该能够使用对他们来说有意义的语言来编写测试，而不是使用只对工具有意义的编程语言。他们关于测试的想法与他们的领域有关（即他们测试的应用程序），于是他们的测试也应该与这个领域有关。

例如，一个拥有保险业务知识的测试人员可能会关注保险索赔和续保。一名专注移动设备的测试人员可能会想对网络连接、会议电话等进行测试。对于不同行业部门的人来说，这些高层次关键字可能非常不同，即使一些低层次的关键词可能会相同。这就是为什么我们喜欢这个词目——“领域相关的测试语言”（Domain-Specific Test Language, DSTC），这点 Martin Gijsen 曾也提到（www.deanalist.nl）。

如果没有任何编程经验的测试人员既能编写也能运行自动化测试，那么自动化测试就能对业务领域有巨大的帮助，而且更有可能广泛地应用，进一步增加 ROI。

专门地讨论这个层次抽象的案例研究包含在第 2、11、12、14、15、20、21、27 和 29 章。

0.2.2 自动化测试标准

如果人们仅仅关注自己的事情，他们将会以只对他们有意义的方式创建自动化测试和测试件，然而他们创建的东西彼此之间都不相同。如果想要很多人使用自动化测试，那么定义自动化的标准并去遵守这一标准是很重要的。标准的命名约定，存储测试件条目的统一的地方，创建测试的标准方式，归档测试的标准例子，对象、脚本和数据文件的标准名称；每个脚本的标准文件头——需要达成一致并且普遍使用的事情远比你能想到的还要多。之外，使用自动化测试的人越多，遵循标准就越重要，并能带来更大的收益。

对这些自动化测试标准的定义最好在试点项目中开始，在试点项目中可以尝试各种方法并且达成一致。之后必须将大家都同意的标准告知所有将要使用自动化测试的人员——无论使用领域相关的测试语言定义自动化测试的人员，还是开发可复用脚本的人员。我们还建议将一些标准构建到支持测试的工具中，以便更容易地创建更多的脚本，而且还能避免很多人为错误。

然而这些标准也不应该一成不变。随着自动化测试变得成熟，保证标准可以变更以及遵循新的（改进的）方法进行更新是个很好的主意。另外很重要的是，允许标准中出现例外情况，当然，要对这些例外进行验证。随着自动化测试规模变大，将标准进行集中管理并且就其与所有人员进行沟通就变得更加重要。关于标准的讨论参见第 6、9、12、15、21 和 29 章。

0.2.3 可复用性、文档和灵活性

虽然有时构建一些“可以丢弃的脚本”（disposable script，参见第 19 章）是很有用的，然而大部分自动化测试中，单个脚本都有很长的生命周期，并且需要以多种方式复用。如果脚本使用多次，那么就值得花费时间去保证它构建良好，如其中包含良好的错误处理方法，并且保证对其进行过测试和审阅。去找寻那些可以复用的模块，而不是一切都从头编写。

为了使得测试自动化人员能够找到并且使用可复用的模块，这些模块必须有良好的文档。应该可以很容易地访问这些测试件条目的信息，而且这些信息应该以标准的格式呈现出来。例如，在每个脚本头，写出这个脚本的目的、如何使用这个脚本（它期望接收的数据以及返回的数据），以及一些前置条件和后置条件。一些额外的信息也很有用，如在一些可能的错误条件下会发生什么。这些信息应该可以很容易地搜索到。事实上，将这些信息收集起来就可以作为测试件的文档，而且通常可以很直接地将这些信息收集起来，并将其分类到单独的文档或者集中进行管理。

在运行测试时，灵活性是很重要的。例如，如果一个脚本正在处理从电子表格中得到的数据，而且并不需要电子表格中所有项都填满，那么一个更加灵活的脚本应该允许跳过那些没有值的字段，这可使测试人员很容易创建各种不同的测试。

为每个测试标注上该测试属于哪一类也是很有用的，例如，可以标记其为冒烟测试、修复 bug 的测试或者对函数 X 进行的测试。使用这种“测试选择器”可以很快地选择在给定时间运行测试的不同子集。

这些主题在第 3、5、11、14、21 和 29 章中讨论。

0.2.4 测试结果和报告

评估测试的结果是测试所做的最基本的事情：如果不关心结果是什么，那么它就不是一个测试！然而对自动化测试的结果进行检查并不是一件显而易见的事情，尤其是当检查结果是以屏幕截图的位图存储时（要尽可能避免），对结果进行自动化检查就更加困难。

工具不会告诉你一个测试通过还是失败；它仅仅能告诉你测试结果和你期望的结果是否匹配。如果期望结果就是错误的，那么工具可能很“高兴地”认为所有的测试都通过，即使它正在做错误的事情！

有时，自动化测试可能愚弄你，让你认为它们正在检查一些东西，而实际上没有——通过的测试通常不会有人进行检查，然而这有时会带来严重的问题，正如第 15、19 和 29 章所讨论的（第 15 章称这些测试为“僵尸测试”）。

对于测试结果的报告应该让收到报告的人能够理解这些测试；为此可能需要做一些额外的工作来过滤以及解释原始的测试结果，但是这些工作很值得，因为这能让各类人员对这些测试有着更好的交流。这些主题在第 3、5、13 和 20 章中讨论。

有时通过使用部分的预言（oracle），自动化测试仅仅能够检查测试结果的一部分而不是所有。例如，探索性的自动化测试（第 28 章和第 29 章）、猴子测试（第 24 章）和可靠性测试（第 13 章），可能会检查系统是否还在运行而不是检查系统的结果是否正确。当能够自动地生成大量测试时，仅仅检查系统是否能存活下来也是很有价值的。

更多的关于测试结果和报告的讨论见第 3、5、10、13、14、17、19、20 和 29 章。

0.2.5 对测试进行测试：审阅、静态分析、对测试件进行测试

自动化代码（包括脚本）也是软件，正如所有的软件一样，它们也会有 bug，并且必须对其进行测试。对自动化测试件进行审阅，甚至检查，也能帮助找到在测试件本身中的 bug 和问题，并且保证自动化测试与它的目标是一致的。这还能够有助于团队内沟通知识，让团队人员更好地了解测试件的细节。不需要审阅所有的事情——选择一个具有代表性的测试方案，并且与利益相关者对其进行审阅是一种很有效的方式，有助于确定正在进行的工作是否正常。

正如第 15 章所述，另一个来自软件开发的技术，是对测试脚本的静态分析。静态分析工具能够发现代码（包括自动化代码）中的 bug，这些 bug 在评审过程中以及测试中可能发现不到。虽然静态分析不能找到所有的 bug，但其发现 bug 的速度很快而且成本很低。

这些主题在第 1、5、7、10、15、17、21 和 29 章中讨论。

0.2.6 对哪些测试进行自动化

第 2 章和第 12 章证明了对那些没有价值的测试进行自动化是没有意义的——首先，保证将进行自动化的测试是值得自动化的；其次仅仅自动化那些重复使用的测试。

首先应该对哪些测试进行自动化？那些对测试人员来说执行起来很无聊的、那些普通的并且重复的、或者那些非常复杂的测试，都是开始自动化的最佳候选测试，但是也要根据当前最需要解决的问题来进行选择，总的来说，要对那些能够为测试带来价值的测试进行自动化。

第 25 章给出了一个用于决定是否对某个测试进行自动化的检查列表。知道不要对什么进行自动化以及应该对什么进行自动化是很重要的。过多地进行自动化也会影响自动化的结果，因为这可能使你难于获得一些“快速的回报”（quick win），虽然这些回报能更有效地证明自动化的价值。

这些主题在第 2、8、19、20、22、25 和 29 章中讨论。

0.2.7 自动化测试不仅仅是执行测试

大多数人认为测试执行工具仅仅是用来执行测试的工具。然而，要达到好的自动化，仅仅对一项活动有着工具支持是远远不够的。书中很多案例研究都讲到，创造性地思考通过工具

能支持自动化过程中的哪些活动——不论是通过使用传统商业工具还有内部实用工具、脚本、宏等，都能给项目带来惊人的收益。第 19 章主要关注这个主题，除此之外，第 3、4、5、14、21、22、24、25 和 29 章还包含了一些有用的小窍门和主意。

在自动化测试过程中经常被忽略的一个领域就是，将手动测试引入到自动化过程中是很有用的。第 21 章讲述的例子很有趣，它在改进自动化的同时也加入了对手动测试的支持，在第 1、8、12 和 19 章也讨论了这个主题。

另一个常被忽略的领域是对于预处理和后续处理任务的自动化。如果不能对这些任务进行自动化，那么自动化测试就不得不加入手动过程——这在很大程度上违背了自动化的目标！关于这个主题的更多讨论请参见第 2、10、11、18 和 22 章。

0.2.8 失败分析

当测试失败时分析原因可能会花费大量时间，对自动化测试失败的分析所花的时间要多于手动测试所花的时间。例如，手动测试时，在发现 bug 前你都知道做了些什么，所以在发现 bug 时就掌握了 bug 的上下文信息。然而，当自动化测试失败时，你可能仅仅知道测试的 ID，而并不清楚这个测试到底是做什么、它所处的位置，以及发生了哪些其他的事情。在写 bug 的报告前需要重建 bug 的上下文信息，这需要一些时间。考虑到这种情况，自动化测试应该记录一些日志，这些额外的信息对需要检查测试失败的人是很有用的。失败分析在第 20、23、27 和 29 章中讨论。

0.2.9 自动化测试是否用来发现 bug

0.1.1 节提到过，发现 bug 对于测试来说是一个好的目标，然而对于自动化的回归测试来说，找到 bug 却不是一个好目标。然而，在一些示例中自动化确实可以发现新的 bug。如果自动化能够允许运行一些测试，而这些测试不能通过其他手段运行（因为手动运行它们需要大量的时间而且不实用），那么自动化就能增加被测软件的测试覆盖率，而且还能发现那些其他方法找不到的 bug。

基于模型的测试（model-based testing，参见第 9、14、24、28 和 29 章）不仅能够在执行测试时发现 bug，还能在模型的开发过程中发现 bug。

当自动化测试能够运行那些依靠手动几乎不可能执行的长序列的测试时，如猴子测试、可靠性测试和探索性自动化测试，就可以发现手动测试发现不了的 bug。

如果想要记录哪些 bug 是通过自动化测试发现的，而不是通过其他方式发现的，第 11 章对 bug 跟踪系统提出了一些建议用来找到这些 bug。第 27 章有一些有趣的统计数据，比较了自动化测试发现的 bug 数量（<10%）与手动脚本测试和探索性测试发现的数量。

0.2.10 工具和技术方面

0.1.9 节从管理的角度讨论了工具，同时也必须考虑到这些工具的技术方面。如果要开发自

己的工具，就可以用这些工具做你想要做的事情；理想情况下，你将会得到适用于需求的最好的工具。然而，开发工具需要进行大量的工作（通常比预计的更多），于是通常需要在期望得到的功能和可以利用的资源之间做出折中。

可以尝试的是使用多个工具来达成某个目标，而这个目标无法用单独的工具来完成。在采用这些工具时，要充分挖掘它们的用途。

那些和当前测试领域密切相关的工具能保证正确使用这些领域知识，而且使用起来更容易，也更少出错。

如果你在考虑测试件该使用什么编程语言，那么就挑选那些开发人员已经熟知的并且受欢迎的编程语言。这样就能更好地得到开发人员的帮助来解决测试件中出现的问题。

你也需要仔细注意测试的环境；测试发现的失败应该是真正发生的问题，而不是因为在环境中忘记了加入一些因素。很多环境因素的设置也能自动化地进行，这也有效地减少了人为错误。

还需要关注测试和软件的同步，尤其是当被测系统并没有限定只能使用标准的 GUI 控件时。虽然有很多工具对于同步提供很好的支持，并不是在所有的情况下工具都能很好地解决问题。而且有时无法充分地解决所有同步问题。这也就意味着软件中有些部分是不能应用自动化测试的，至少目前不行（或许未来会出现新的技术手段来解决这些问题）。

第 7、8、14、17、18、19、24、26、27 和 29 章讨论了这些问题。

0.3 总结

自动化测试不再是件奢侈品而成为系统的必需品；随着应用程序和系统变得越来越大、越来越复杂，仅仅依赖手动测试已经无法全面地测试系统。随着技术的进步，测试也必须进行调整——而且要快。

本书描述的自动化测试的实践包含了在 21 世纪第二个十年开始时测试自动化的情况。这些故事中既有痛楚也有荣耀，既有失败也有成功，既有卓越的想法也有难以置信的决定。倾听这些故事并且注意这些作者学习到了什么，这样你自己的测试自动化的实践便更有可能取得成功！

第 1 章　敏捷团队的自动化测试之旅：第一年

Lisa Crispin

浏览“如何阅读本书”和“案例研究反思”，了解本书章节安排。

Lisa Crispin 以其特有的迷人方式描述了当一个敏捷团队决定实施自动化测试时所发生的事情。由于 Lisa 在敏捷技术方面的专业能力，当看到这支团队在实践中确实非常敏捷时，我们一点儿也没有感到惊讶。这个项目中一件有趣的事情就是：团队（小型团队）里面的每个人都参与了自动化。他们不仅擅长敏捷开发，而且非常敏捷地对其进行了自动化——并且他们成功了。实施敏捷开发并不是这支团队取得成功的唯一要素，其他要素也同等重要，其中包括通过良好的沟通建立稳固的管理关系，以及建立自动化平台来支持创造性的手动测试。另一个关键要素是在团队将过程改进嵌入到整个流程的决策力，包括一年两次的自动化重构的安排。毫无疑问，Lisa 和她的团队在他们第一年的时间里所取得的成就是非常显著的。这个项目是为一家美国公司的财务部完成的，特征见表 1-1。

表 1-1　案例研究特征

特　　征	本案例研究
应用程序领域	金融理财，基于 Web 的 J2EE 应用程序
应用程序规模	226 000 代码行：95 000 行遗留代码，其中大多数已通过 GUI 冒烟测试；131 000 个新架构，被 4364 个 JUnit 测试所覆盖（在 2004 年，我们有 128 000 行遗留代码，20 000 行新代码通过 473 个 JUnit 测试）
地点	美国
生命周期	遗留系统的维护和新系统的开发周期都是灵活的
项目人员数量	9 ~ 12 人 (程序员 4 ~ 5 人，测试人员 1 ~ 2 人，数据库管理人员 1 人，系统管理人员 1 ~ 2 人，敏捷教练 1 人，项目经理 1 人)
测试人员数量	1 ~ 2 人
自动化测试人员数量	所有人均参与了自动化
时间跨度	案例研究主要是在第 1 年，但会明确 6 年之后的方向
年代	2003—2011

（续）

特　征	本案例研究
工具类型	开源工具
是否进行试点研究	否
ROI 测量	无确定的测量标准，虽然我们跟踪缺陷数量和开发速度（每个冲刺阶段的故事（story）数量）
是否成功	是
是否还在运行	是

1.1　本案例研究的背景

我们必须面对这样的事实：对于从未进行过自动化测试的人来说，自动化测试是具有一定难度的。本故事告诉我们，面对无任何自动化的测试和有着糟糕设计的遗留系统，这支团队通过一年多的努力，将所有的回归测试都实现了自动化。在接下来的几年时间里，我也与数十个其他面临同样困境并找到类似解决方案的团队进行了交谈。看看我们所遇到的这些困难是否与你所遇到的相似，并考虑用类似的方法进行尝试。

1.1.1　问题

从这里开始着手：每两周我们都需要把新的功能添加到产品中，但是代码 bug 成灾并且也没有自动化测试，更严重的是，产品中有大量随时会导致系统中断的 bug。我们如何摆脱这种情况呢?

1.1.2　目标

我们决心尽自己所能编写出最高质量的代码，但是从哪里开始呢？作为一支自组织的敏捷开发团队，让我们感到欣慰的是整个团队的紧密协作。那是在 2003 年，我们中有些人在别的团队中曾有过良好的自动化测试经验，他们相信总是会有办法的。我们发现，一个安全的自动化回归测试网络可以让我们更快速地工作。如果我们知道是由于某段特殊的代码而引入的非预期的操作，那么我们能够立即稳定我们的代码库。通过充分的测试覆盖来不断进行集成，使我们每天都有一个稳定的构建过程。而在迭代的后期，现在很难得到稳定的集成，所以这个想法虽然并非那么容易实现，但听起来很不错!

接下来，看看到底是什么帮助我们创建了一个成功的策略来实现自动化回归测试套件。

1.2　整个团队的承诺

我们是由多个程序员、一个测试人员、一个数据库管理人员、一个系统管理人员和一个敏

捷教练所组成的团队。公司的业务专家可以随时协助我们。我们整个团队致力于在每次发布之前运行我们的手动回归测试脚本。因为我们每两周发布一次，这意味着两周的迭代周期中我们要花 1 ~ 2 天的时间进行手动测试。在最终用户使用之前，我们没有足够的时间来进行探索性测试，虽然这种测试可能会帮助我们找到严重的缺陷。

【真知灼见】

敏捷开发团队中的每个人都进行手动测试，所以他们更能体会到自动化测试的好处。

我们都很关心质量，并且我们都致力于保证所有的测试活动都是有计划的且在每一次迭代中执行。这是一个好的开始！

1.3 建立自动化策略

我们需要在不破坏现有功能的前提下发布产品的新功能特性。而且，需要尽快知道一个新的代码变动是否会引起回归测试的失败。手动回归测试在每两周的迭代后期才能给予我们反馈，以至于没有时间进行充分的回归测试。

我们中一些人曾经在其他敏捷团队中进行过测试驱动开发（Test-Driven Development, TDD）。我们发现 TDD 能帮助创建出设计良好的、健壮的代码。

我们现有的回归测试是手动操作的，整个团队通过使用团队 Wiki 上所记录的脚本进行手动测试。在每两周的迭代周期中，这就花费了整个团队的 20% 的时间。这些测试仅仅为程序最核心的部分提供了最小程度的覆盖。产品中报告的缺陷表明，回归测试的失败仍有可能发生。在每次迭代周期中，我们至少要花 20% 的时间修复这些产品缺陷，从而限制了我们能开发的新功能的数量。

自动化的回归测试会带来更高、更准确的覆盖率，这需要投入大量的时间、金钱和精力，我们可能需要硬件和软件来建立构建过程、持续运行测试所需要的环境，我们也需要使测试实现自动化的架构和驱动。然而，我们可以计算出这一自动化将会节省我们 40% 的时间，利用这些时间可以进行更多有价值的活动，比如进行新的开发，所以，其收益远大于成本。

继续进行手动回归测试必将会失败，我们需要一个明确的决策来进行自动化。因为我们都在进行手动回归测试，每个人都感受到了没有自动化测试的痛苦。所以，我们有了解决这一问题的动力。首先，我们需要一个可测试的架构……

1.3.1 一个可测试的架构

TDD 这条路是要走的，但是当前的代码在业务逻辑、数据库访问和 UI 等处相结合时，情况比较复杂，自动化单元测试也就变得很难了。往往很难隔离任何一个组件单独进行测试。

这样看来，似乎找到一种把软件进行分层的新架构是非常明智的。我们开发出了这一新架构的所有新功能。

【小窍门】

不要尝试解决老问题，进行新的开发来开展更好的实践。

如果进行自动化测试的成本比其收益高，那么进行自动化测试就没什么意义。我们研究图 1-1 所示的自动化测试金字塔（这一金字塔是由我们当时的经理 Mike Cohn 提出的）。单元级别的测试一般 ROI 最高。程序员可以很快写出它们并运行，而且测试可以根据需要进行更新。因此可以将单元级别的测试作为自动化回归测试的坚实基础。

我们的业务逻辑相当复杂，而且新架构将这一逻辑从数据库和用户接口层中分离出来，这样就可以通过设置内存中数据并在其上运行产品代码来进行测试。这是金字塔的中间层———比底层运行的测试要少但是依然很重要。

我们还需要测试 UI，但是通过 UI 进行的自动化测试本身就非常脆弱、维护费用高且运行缓慢。因为最终想使 UI 测试所占的比例尽量最小，所以它就处在金字塔的最顶端。尽管如此，与其他团队一样，我们从图形用户界面（GUI）冒烟测试（smoke test）开始，来获取一些代码防护。所以，从这个角度，我们的金字塔是上下颠倒的，但是没关系———最终我们会将它翻转过来。（我们最终花了 4 年时间才获得为之努力的三角形形状！）

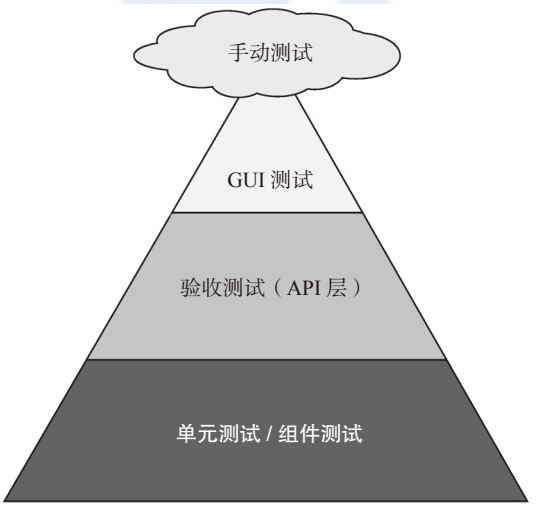


图 1-1　自动化测试金字塔

【小窍门】

解决问题的最直接途径未必是最佳途径。

我们现在明白了我们需要做什么，所以我们开始着手实施那个能给我们带来最大实惠的任务。

1.3.2 建立构建过程

我们只有 4 位程序员，但是他们会经常检查源代码控制系统中的变化。我们需要确保他们没有不小心修改别人的内容或者破坏现有的任何功能。同时，我（测试人员）有时候不得不等待好几天，部署在测试环境中的代码才能完成一次构建。一个自动化的构建过程（build process）将保证在每次检入后的几分钟内，最新代码的可部署版本就是可用的。

管理层向我们强调了质量是我们的第一目标，他们乐意宽限一些时间来让我们搭建一个好的基础结构。

【真知灼见】

好的管理者会准许团队花时间去开发自动化的基础结构。

我们停下正在做的事情，使用 CruiseControl 和一个新的 Linux 服务器建立了一个持续集成（Continuous Integration, CI）过程。因为我们还没有可以运行的任何测试实例，所以只是简单编译代码和构建可部署的二进制文件。我可以看到每次构建的检入文件并能够选择想部署到产品中的二进制文件。这很有用，但是还不够。

1.3.3 获取测试的基准：GUI 冒烟测试

程序员正在学习如何使单元测试自动化并编写测试先行（test-first）的代码，但是要真正实施测试驱动开发需要花好几个月的时间。针对遗留代码，使用 GUI 冒烟测试可能是一个快速获得自动化测试覆盖的途径。但是使用什么工具最好呢？

我们有购买一个付费工具的预算资金，但是团队里的程序员是 Java 程序员，他们万不得已是不愿意使用另一种脚本语言来进行测试的。捕获 / 回放并不适合我们，因为我们需要可维护的、稳定的测试脚本。我们选中了 Cannoo WebTest——一个可以让我们修改 XML 文件中的测试用例，并使用 Ant[⊖]运行它们的开源框架。而且它很容易与我们的构建过程集成。

【真知灼见】

昂贵的商业工具不一定是最好的选择。

我让业务专家把需要用冒烟测试来保护的系统核心区域按优先等级进行划分。每个冲刺时间段里，都安排了时间让我用 WebTest 工具自动运行测试脚本。我们首先追求的是“快赢”，针对系统中每个用户角色的基本功能实现自动测试。

首先，CI 构建过程只运行了少量单元测试和一些覆盖系统高得分点的 GUI 冒烟测试。随着在这两个级别引入更多测试，我们将 GUI 测试移到单独的构建过程中并只在晚上运行，这样，我们可以更快得到的反馈信息。

⊖ 一种基于 Java 的编译工具。——译者注

与此同时，我们将单元测试和每个新用户故事（user story）的 GUI 冒烟测试都放在迭代中完成。新的功能将会在自动化的单元测试和 GUI 测试中被覆盖。一旦程序员能够熟练使用 TDD，我们就将进入金字塔的中间层。

1.3.4 在单元级别驱动开发

我们的程序员中只有一人曾经实施过 TDD，但每个人都支持这种理念。我们根据别人的介绍，找到了我们能找到的最好的顾问来帮助我们学习如何实施 TDD，很多时候我们都是自带午餐以便挤出时间来进行 TDD 试验。但是很遗憾的是：它很难学！我们团队需要时间来掌握它。

我们的管理层知道这个道理：目标是写出优美的代码，优美到可以拿回家给妈妈，当做冰箱贴。我们的公司——一个刚刚成立 3 年的商业公司，因为网络应用问题和无能力及时发布新功能而面临倒闭。我们的商业合作伙伴也准备放弃我们了，但自从我们实施敏捷开发（Scrum）以后，他们看到了我们为提高稳定性和响应能力所做的努力。我们的执行者致力于具有长远收益的投资。

【经验教训】

一个迫在眉睫的灾难可能是巨大进步的推动力。

我们知道这些单元级别的测试具有最好 ROI。我们也理解 TDD 事实上是代码设计，而不是测试。考虑“代码是用来做什么的”可以帮助程序员写出正确的代码，而快速实施测试，才能及时地提供重要的反馈信息。

单元级别测试的最大好处就是能够提供最快速的反馈。在研究一些好的实践之后，我们决定将代码构建过程的时间控制在 10 分钟内。这需要单元测试的随机重构。早些时候，单元测试需要访问数据库，之后，程序员掌握了如何构造、模拟它，以及消除它的影响，使测试快速运行又能提供正确的测试覆盖。

1.4 利用验收测试驱动开发，使用 FitNesse 测试 GUI

现在已经是我们自动化之旅的第 8 个月了，程序员已经建立了一个自动化单元测试的实用库。对于应用程序的核心区域我们已经进行了冒烟测试，覆盖微量代码的大约 100 个 JUnit 测试已经完成了。但是中间层还什么都没有，TDD 此时变成了一个空壳。现在我们开始对自动化测试金字塔的中间层进行填充。

1.4.1 内存内测试

我们的金融理财产品有许多复杂的算法，这可以通过在内存中提供输入来进行测试。与单

元测试相比，这种测试的级别要高很多，但我们还是不想通过 GUI 来进行测试，因为其速度慢、成本高。我们发现可以很迅速、方便地编写 FitNesse 夹具（fixture）来自动进行这类测试。在客户测试层次，我们开始用 FitNesse 测试来驱动新的用户故事的开发。这些测试使用夹具在内存中构建测试输入，并把这些输入发送到应用层代码，就如同产品在实际使用时进行的输入一样。然后夹具会返回代码的实际输出，并把它与 FitNesse 表中的期望结果进行对比。

测试人员和客户填写 FitNesse 测试用例，之后程序员用夹具来自动运行它们。这意味着我们需要交流！提高团队的交流能力是使用这种工具进行自动化测试的最大好处之一。我们测试人员与产品所有者和其他利益相关者坐在一起分析每个用户故事预期的和非预期的行为。我们将这些用户故事转化为 FitNesse 测试用例表，并与客户核对以保证当测试用例通过测试的时候能够满足客户的要求。我们与开发人员核对测试，以保证我们清楚需求并保证测试设计与代码设计兼容。开发人员编写夹具来自动运行测试。这一过程在单个用户故事的很多小的迭代中不断重复，直到开发和测试都完成。

1.4.2 使用数据库的测试

我们的应用是数据密集型应用，我们想自动运行更加端到端（end-to-end）的测试。我们也可以使用 FitNesse 在数据库中构建测试数据，并使用遗留代码在它上面运行，以此来测试遗留代码。

这种类型的测试脚本编写和维护都更加昂贵。作为 FitNesse 的初学者，我们犯了一些错误，如，不知道将测试组件模块化，而这可以通过 FitNesse 中现有的部件来完成。我们彻底违反了代码设计中的“不要重复自我”准则。例如，在几十个不同的测试页面中，有包含同样员工信息的表，如果又有一列新数据需要添加到员工信息表中，那么在每个测试页面中都要加进去，这很糟糕。等到我们知道如何正确去做的时候，又遇到了一个更难以处理的麻烦。

【经验教训】

在前期进行工具使用的培训，之后就可以避免因工具使用不熟练而造成的巨大的时间浪费。

我们将每个能想到的测试用例都进行了自动化，包括发生几率低、影响小的边缘测试用例，并把它们都放在自动回归测试套件（test suite）里面。上述过程花费的时间并不长，但是接下来，这个测试套件要花费大量时间和主机功率（machine power）来运行，维护成本也随之提高了。所以，我们要学会慎重选择测试用例，确保它们能提供充分的测试覆盖，并只将这些测试用例放在回归测试套件中。

【小窍门】

精化回归测试套件可以在保证收益的同时降低维护费用。

正如在产品应用代码中所做的那样，我们现在不断地重复访问和重构 FitNesse 测试用例，以保证我们所需要的测试覆盖，同时不会过多地延长反馈周期或者花费太多时间维护测试。

【真知灼见】

经常检查自动化测试用例以保证它们是有效的。

1.4.3 使用 FitNesse 测试的好处

我们的 FitNesse 测试提供了比 GUI 测试套件更快的反馈，尽管比 JUnit 测试要慢很多。FitNesse 测试套件需要 60 ~ 90 分钟来运行，而 JUnit 测试的运行时间仅仅只需要不到 8 分钟。像 GUI 测试脚本那样，我们将 FitNesse 测试集成到构建过程中，并且在其中运行。一开始，我们只在晚上运行这种“完全构建”（full build），但这无法提供及时的反馈，并且如果测试失败的话，我们只有在第二个晚上才能知道问题是否修复了。我们投入了更多的硬件，这样我们就可以“连续地”运行单元级别上的所有测试的完全构建。如同运行单元级别测试的构建一样，这是为了使源代码控制每接收到一次检入就能运行。大概需要 90 分钟运行一次，所以经常同时测试几个新的检入。

1.5 使用增量方法

与很多团队一样，我们发现在每两周的迭代后期都有未完成的测试任务。有时候用户故事未完成。比如，我们从一个包含 5 页向导程序的 UI 开始一个故事（story），其中只有 4 页完成了。

有一位程序员提议用一个“强线程”（steel thread）来标识复杂故事——一个将功能点从不同终端隔离开来的小的功能点。我们为它编写测试、写出代码，然后将测试自动化然后转移到下一个线程。那样的话，测试自动化即便在 GUI 层面，也能与开发保持一致。第一个自动化的测试可能会过于简单，但可以逐步进行补充。

【小窍门】

强线程是一种保证自动化在时间里程碑内完成的方法。

这种方法很有效，对于每一个计划的复合功能点，我们首先将线程画在白板上，并确保在转移到下个线程之前，当前线程的自动化测试及手动测试都已经完成了。

1.6 正确度量

如何知道我们是否取得进步了呢？如何衡量成功？我们需要一些度量。我们的构建过程会

对测试进行统计，所以很容易跟踪每一个层次的测试数量：JUnit、FitNesse 断言和测试页面、Cannoo WebTest 断言等。

这些原始数据并不能代表整个趋势。我们，包括业务经理，都希望看到这些数字在每次迭代中都在增长。在每一次迭代中，指定一份染色的日历——如果某天所有的回归测试至少通过了一次，则将其日期标记为绿色，否则标记为红色。业务人员也非常关注这些数据，如果他们在一行里面看到两个标记为红色的日期，他们会向我们询问原因。他们也会记录每一次迭代中 JUnit、FitNesse 和 Canoo WebTest 的测试数量。当测试数量减少时，他们也会关注并询问原因。

【真知灼见】

让经理看到自动化测试的好处。你需要不断“推销”自动化的好处。

让测试结果对外可见也是一种宣传的形式，可在整个公司范围内加强公众对自动化回归测试的了解。客户看到测试数量增加，并且发布的功能点越来越多。所以，当我们申请时间来重构产品或者改善构建过程时，我们的客户就能够理解了。

我们需要快速的反馈，所以每一次构建花费的时间是一种重要的度量。如果运行 JUnit 测试的持续构建花费的时间超过 10 分钟，就开始备份检入信息，那么就很难找出是哪一个测试导致了失效。当 JUnit 构建花费的时间超过 10 分钟时，我们停下来查看是什么原因导致了运行缓慢。我们重构测试、添加硬件、升级操作系统、配置测试使其并发运行等，这些都会使反馈循环更快。对运行更高级测试的构建也是这么做的。如果 FitNesse 测试要花好几个小时来运行，那么反馈太慢。我们要在测试覆盖和速度之间权衡。幸运的是，虚拟机可以同时运行多个测试套件，减少反馈周期并且非常廉价。

【小窍门】

不要使自动化停滞不前：在需要的时候进行重构、使用新的硬件、重组测试。记住我们的目标（这里指更快速的反馈），并修改自动化测试来达到我们的目标。

1.7 庆祝成功

100 个 JUnit 测试并不是很多，但每一个测试代表了许多断言，并且是一个值得庆祝的里程碑。当达到 1000 个 JUnit 测试时，我们觉得这值得全公司聚会庆祝一下，并为全公司的人购买了比萨。当庆祝第 2000 个 JUnit 测试时，我们组织了一个有饮料和拼盘的小型聚餐。有人曾经问过我，这些庆祝会不会鼓励一种不好的事情：盲目增加不必要的测试数量。其实，因为每个测试都是刚好在让其通过的那部分代码之前编写的，所以我们没有进行任何无关的单元测试。如果包含测试的代码移除了，那么测试也会移除。

当接近 3000 个 JUint 测试时，敏捷教练让我撰写一份说明书，谈谈在单元测试级别中一个健壮的回归测试套件对业务的重要意义。我写了一份关于 TDD 的高级描述报告，即它是如何帮助我们开发出健壮的、易维护的代码，并在单元测试级别为我们提供了回归测试覆盖的安全网。我试图阐明 JUnit 的目的：帮助设计代码。也就是说，我们不是胡乱编写单元测试的。敏捷教练把这份报告发给公司的每个人，并在一家本地饭店预订了一个聚会。不仅是为了庆祝我们完成任务，而且让所有的业务利益相关者也能体会其中的意义。

1.8 引入工程冲刺

因为我们已经花时间向业务经理解释了测试的整个过程和实践，所以他们了解了技术债务的观念。每当我们为了顾及截止日期而走捷径的时候，代码就变得很难处理。如果我们没有时间将工具升级到最新版本，或者没有时间尝试能让我们更高效地工作的最新软件，我们的速度就会下降。这就好比如果我们拖欠信用卡债务，利率就会上升，我们所欠的债务就越来越多。如果我们被技术债务缠身，就无法以客户期待的速率来发布大量的软件。

所以，我们每 6 个月会进行一次特别的迭代，称为“工程冲刺”在这次迭代中不需要发布任何新的功能点（engineering sprint）。我们可以利用这段时间完成以下工作：升级到最新的 Java 版本、进行大型重构、尝试新的测试工具、重构 FitNesse 测试来消除冗余。在一次工程冲刺中，我们将源代码版本控制系统从 CVS 转变为 SVN（Subversion）。在另外一次工程冲刺中，我们将构建过程从 CruiseControl 转变为 Hudson，从而为缩短反馈周期提供了更多的选择。每当在更短的时间内发布了更多更好的功能点时，业务人员就可以看到这些投资的回报。

1.9 团队成功

我们的团队在一年的时间内，从没有自动化测试发展到将所有的回归测试进行自动化。我们的自动化金字塔还不完全是理想的形状，但是我们有了好的测试框架和在每个级别（见图 1-2）实现测试的驱动程序。尽管这样，我们却没有满足现状，而是继续寻找在各个级别有效地进行自动化回归测试的方法。当我们可以控制功能测试的自动化之后，我们将着手性能测试的自动化。我们有处理不完的挑战！

最好的是，我们有时间对每个用户故事或功能集合进行充分的探索性测试，这样的话，在产品中就很少会有问题浮出水面。同时自动化测试也可以帮助我们进行探索性测试。我们希望创建：使用 Ruby 的具有高灵活性的脚本，Watir（Ruby 中的 Web 应用测试）测试驱动，以及测试 / 单元框架。它们接收运行参数，允许我们在几秒或者几分钟内建立复杂的测试数据和情景，而不再需要花费时间搭建繁琐的手动测试平台。通过脚本可以直接对需要进行测试的部分进行测试，这样我们就可以拥有更多的时间来深入探索软件。

【真知灼见】

使用自动化测试来支持创新型手动测试。

我们的目标是使所有回归测试实现自动化，但是这一目标有几点需要说明。要达到一个合理的 ROI，自动化测试在设计时必须要考虑到长期可维护性。因此要求程序员拥有好的设计技巧，可以帮助设计各个级别的自动化测试。我们不断对测试进行重构以使维护费用较低。同时，在选择将哪些测试保留在自动化回归测试套件中时，团队要有准确的判断力，需要“正好”（good enough）覆盖。测试太多意味着反馈周期太长和维护费用过高。同时，我们还有少量的遗留代码没有被自动化测试覆盖，当我们进行可能对遗留代码造成影响的大范围的代码重构时，我们要留一些时间进行手动回归测试。

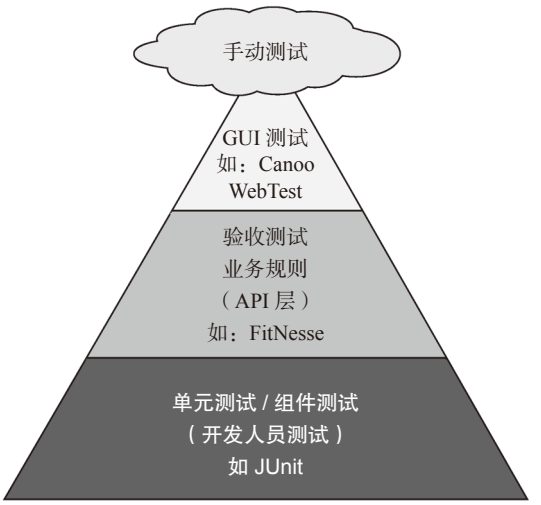


图 1-2　自动测试金字塔使用的工具

我们成功的关键是采用“完整团队”（whole-team）的方法 , 团队里的每个人都致力于将所有回归测试实现自动化，我们有很多的技术和观点来帮助解决自动化中遇到的问题。自动化测试金字塔的每一层都包含不同的工具，而且大家可以关注不同层。开发人员像写代码那样编写单元测试用例，测试人员更多的是进行 GUI 测试的自动化，在 FitNesse 中，双方在中间层合作。然而，在每一个用户故事完成之前，团队中的每个人都负责每个用户故事的所有测试活动。

自动化测试给我们一个快速的反馈周期。如果在几小时内任何一个现有的功能点遭到破坏，在工程冲刺时，我们应该腾出时间来做出改变，使加入更多测试用例后的反馈周期仍然很快。我们知道是代码的哪些改变引起的这个问题，并立刻进行修复。我们可以在经常发布新业务价值的同时，达到公司的要求，开发出最高质量的产品。

1.10 持续改进

2011 年是我们自动化测试之旅的第 8 年，总是要面临新的挑战。正如本章所述，我们的 GUI 测试套件已经增长到需要 2 个多小时来运行。这个时间太长了，所以我们将它划分为两个测试套件，并在两个从属机器上并行运行。这需要进行大量的工作，因为有些测试依赖于其他测试，过去没有好好实施而是采取了折中的方法，现在要为此付出代价。我们有超过 5400（这个数字还在增长）个 JUnit，并且重构的 FitNesse 测试套件在 30 分钟内完成。

我们知道在单元级别中的测试覆盖率，但是并不知道在功能性或 GUI 级别的测试覆盖率。我们目前采用一个的工具进行实验，可以测量出 FitNesse 测试的代码覆盖率。

相比于 8 年前，我们现在对自动化测试设计有了更深入的了解，并且我们的测试需要大规模的重构。无论何时，当加入一个测试用例或者对其进行修改的时候，我们都尝试去改善测试，但是只在工程冲刺阶段才进行大规模重构。我们通过一些开源工具进行观测，并不断尝试我们认为可以降低测试维护费用或者提供更快反馈的新工具。

在 GUI 测试中我们也是这么做的。几年前，当我们发现 Canoo WebTest 并不能很好地支持 JavaScript 的时候，就在所有回归测试中开始使用 Watir 编写新的功能点。约一年之后，Canoo WebTest 升级到比那时的 Watir 可以更好地处理 JavaScript 和 Ajax，而 Watir 更难与构建过程整合，所以在保留现有的 Watir 回归测试脚本的基础上，我们又切换回 WebTest。

我们也进行了调查，准备使用 Slim 来替代 Fit 进行我们的 FitNesse 测试。同样，我们可能不会立刻把所有 FitNesse 测试转为 Slim，但我们同时发现维护多种工具也没有很大问题。

这些年来，我们的员工相当稳定。一位新程序员和一位测试人员来了又走，但是团队的核心成员一直保留着。我认为这是因为优秀的人才乐意留在他们可以尽最大努力工作的地方，那就是我们团队。比如，我们允许他们做类似自动化回归测试的事情。我们有一些有趣的转变，一位测试人员决定成为一位程序员，所以他参加了一些 Java 课程的培训，另一位程序员跟他一起合作，他的学习速度非常快。同时，他仍然保留了自己在测试方面的兴趣，尤其在性能测试方面。另一位程序员一直都无法采用 TDD 方法，尽管经过了大量的指导和培训，因为他的代码始终没有达到标准，最终只好让他离开。我们从不会去刻意填补空缺职位，但我们的团队效率却在提高。每一个团队都需要恰当的人选，而且那些人需要时间来学习、实验和提高。

1.11 总结

你的团队如何？阻碍你进行自动化测试尝试的最大问题是什么？你的团队是否缺乏一个特殊的技能组合？你是否只是需要时间来制定和实施策略？你是否还在等待合适的硬件或者软件？想想你现在能做一些什么事情来提高自动化测试并缩短反馈周期。要有耐心，一步步慢慢来。对你得到的结果进行试验和分析评估，不断进行哪怕只是很小的改进。不知不觉间，你就会享受自动化测试的好处。

第 2 章　终极数据库自动化

Henri van de Scheur

Henri van de Scheur 讲述了一个跨越 6 年的故事，是有关他和他的同事在开发一款适用于多个环境的测试数据库工具时所发生的事情。他们为自己的自动化测试设定了良好的目标，并为测试工具搭建了一个良好的架构。他们使很多测试实现了自动化，从而为自动化测试构建了一个完整的生命周期，包括周期性的 bug 清理。测试是每晚、每周运行的，或者根据特殊的时间表来安排执行。虽然取得了巨大的成功，但他们也遭遇过一系列的问题，Henri 都如实地进行了描述。这个数据库测试工具（现在是开源的）是由一支小团队短短几年内在挪威开发的，并且取得了非常可观的投资回报率（ROI），如表 2-1 所示。

表 2-1　案例研究特征

特　　征	本案例研究
应用程序领域	数据库
应用程序规模	
地点	挪威特隆赫姆（挪威中部港市）
生命周期	遗留系统的维护和新系统的开发周期都使用敏捷周期
项目人员数量	开始 30 人，最后 3 人
测试人员数量	开始 20 人，最后 5 ~ 6 人
自动化测试人员数量	开始 5 ~ 6 人，最后 2 人
时间跨度	5 ~ 6 年
年代	
工具类型	内部工具
是否进行试点研究	否
ROI 测量	无，但提高了 2400 倍
是否成功	是
是否还在运行	是

2.1　本案例研究的背景

这个案例研究描述了我（指 Henri）在一个快速成长的新公司里的经历：刚开始的时候，大约有 50 个员工，但仅仅过了五六个月，员工数量就增加到超过 100 人。由于人员数量增加

太迅速，导致后来加入的很多开发人员和测试人员缺乏对产品的基本了解。知识的交流被极大地忽视了。结果，产品和测试的质量都很差。

办公室里有大约 50 ~ 60 个开发人员及 20 个专用测试人员，分别分布在同一办公大楼的各个楼层。测试人员报告产品中存在的问题，但由于测试质量差以及缺乏对产品的充分了解，以致这些问题往往被忽视。通过改善开发人员和测试人员两个群体间的交流，开发和测试工作都从中受益并且质量都得到了提高。所以，良好的交流是开始考虑自动化测试的先决条件。

【真知灼见】

不要尝试对设计很差的测试实施自动化，先完善测试，再进行自动化。

一个包含大约 30 ~ 40 个测试的典型的发布测试只在一个平台上运行，这就可能需要 15 ~ 20 个测试人员花费 3 ~ 4 周的时间。同时，由于发布测试中有更多的 bug 修复周期和测试周期（一般是 4 ~ 6 个周期），因此，在可以发布一个产品的新版本之前，过去需要大量的时间。由此可见，将以上过程实现自动化的需求是非常显著的：周期必须缩短，人力资源必须减少，并且测试人员必须在多个平台上运行测试。

一开始，我们使用 Java 语言开发了一个能够满足以上需求的内部工具，随后再将更多的新功能逐渐添加进去。到目前为止，这个工具的一个更新的版本也已经开发出来了，且又增加了一些新功能。测试也是用 Java 语言开发的。虽然自动化测试是从 GUI 自动化开始的，但是基于命令行界面的测试也变得日益重要，因为它可以使团队更方便地通过已安排的分工协作进行自动化。

虽然当时有一些现成的测试套件可用于测试数据库，但是我们并不知道使用什么自动化工具来进行数据库的测试。另外，我们的数据库有当时市场上还没出现的一些特殊功能，而这是现有的测试工具无法测试的。虽然当时已经开发出了一款内部测试工具，但是它根本没法满足我们的需求。那时，我们突然有一些可用资源来开始这样的一个项目，基于这些原因，我们进一步开始测试工具的开发。

2.2 测试中的软件

该项目中要测试的软件是比较特殊的，因为它仅仅只包含数据库。虽然有一些现存的测试套件可用于测试数据库，例如测试多个数据库 API 和查询语言之间兼容性的测试套件，包括 JDK（Java Development Kit）、JDBC、ODBC（Open DataBase Connectivity）和 SQL，但是这些工具的使用并不广泛，并且（或者）它们仅仅只是为使用它们的数据库量身定做的。所以本案例研究中的测试和测试工具都是内部开发的。

我们定义了一个操作系统组合而成的平台，包括它的品牌和 JDK 版本。我们后来又将另一项包含进去：平台是 32 位还是 64 位的。

例如，包括以下几项：

- Solaris 10 SPARC, JDK 6, 32 位；
- Solaris 10 × 64, JDK 6, 64 位；
- Solaris 10 × 64, JDK 6, 32 位；
- Solaris 10 × 86, JDK 6, 32 位；
- Solaris 10 × 86, JDK 5, 32 位；
- Windows Vista 64, JDK 6, 64 位。

2.3 自动化测试的目标

在《Software Test Automation》（Addison-Wesley，1999）一书中，第 8 章有一张非常有用的表，列出了自动化测试的不同目标。当我们开始自动化测试时，根据这张表，按照优先级顺序，列出下列这些测试目标：

- 增强在软件质量方面的信心；
- 更早进入市场；
- 减少测试开销；
- 保持可重复性测试的一致性；
- 自动运行测试；
- 找出回归测试中的 bug；
- 经常运行测试。

第一阶段实施完成之后，又加入了以下测试目标：

- 质量更好的软件；
- 测试更多的软件；
- 性能测量；
- 找出更多的 bug；
- 在不同的操作系统上测试。

【真知灼见】

刚开始不要设定太多目标，最初先重点完成某些目标，再逐步添加新的目标。

另外，在更深入的自动化的全新迭代周期中，我们又制订了新的目标：

1）收集统计数据来制定度量标准（用以回答以下问题）：

① 在哪个操作系统上找到的 bug 数量最多？

② 在同一个操作系统上将测试进行分割会不会漏掉一些 bug？

③ 哪些 bug 是在特定操作系统上才出现的？是在哪种操作系统上出现？

④ 哪些测试发现的故障数量最多？

⑤ 哪些测试从来都没有发现故障?

⑥ 哪些失效是由发布测试发现的，而不是由每晚的测试所发现的？这有利于我们分析为什么这类故障在发布过程中比较晚时才发现。

2）将未使用的硬件用来做其他的事情。

3）尽量达到每周 7 天、每天 24 小时不间断使用硬件。

4）缩短 bug 从发现到反馈给相关人员的时间间隔（最初从几周缩短到几天，然后缩短到只隔一夜，最后缩短到检查代码之后就直接完成反馈。）

5）使用我们自己的数据库来收集统计数据，这样就可以在真实的产品环境中拥有自己的数据，并且有可能会遇到在其他没有发现的故障（用你自己的方法来解决）。

6）使测试场景不可能手动运行。

7）使场景维持几天。

8）使场景具有多个用户。

9）重新使用预处理任务，并使其自动化。

10）重新使用后续处理任务，并使其自动化。

11）对于测试结果自动生成测试报告。

12）完全的自动化测试。

2.4 开发内部测试工具

该内部测试工具的基本功能是由 3 ~ 4 位开发人员在 6 ~ 9 个月的时间内开发出来的，是用 Java 语言编写的。第一个版本开发之后，一个人专门负责对其进行维护和进一步的开发，显然维护和进一步开发的工作量是逐步减少的。

图 2-1 是测试的 Java 引擎（Java Engine for Testing, JET）架构的一个概览。每个大的矩形都是一台运行某些软件的计算机。

我们在图 2-1 中的运行机（runner）处开始运行一组测试集合。它使用 JETBatch 来开始运行测试并收集运行结果。客户端（client）运行与 JETBatch 进行交互的 Jet 代理程序（JET Agent，JAG），即使用 JAG 来启动 JET 运行单个测试。这些 JET 读入一个 XML 文档，该文档会给 JET 提供测试运行的内容，并与服务器的 JAG 进行交互来启动我们需要测试的软件。整套系统还包括一个测试装置，通过使用不同的机器操作不同的任务，避免了架构本身的资源消耗对被测任务的影响。

因为我们确实投入了大量的时间和精力来设计自动化测试，所以几乎实现了所有的目标（只有性能测试是由我们的另一款内部开发工具完成的，这不在本案例研究的讨论范畴）。几乎所有的测试都是通过我们的工具自动运行的，其中只有几个测试，我们认为自动化的价值并不是很大，是由手动完成的。最后，我们有了一款可以在很多不同领域使用的、功能非常强大的工具。

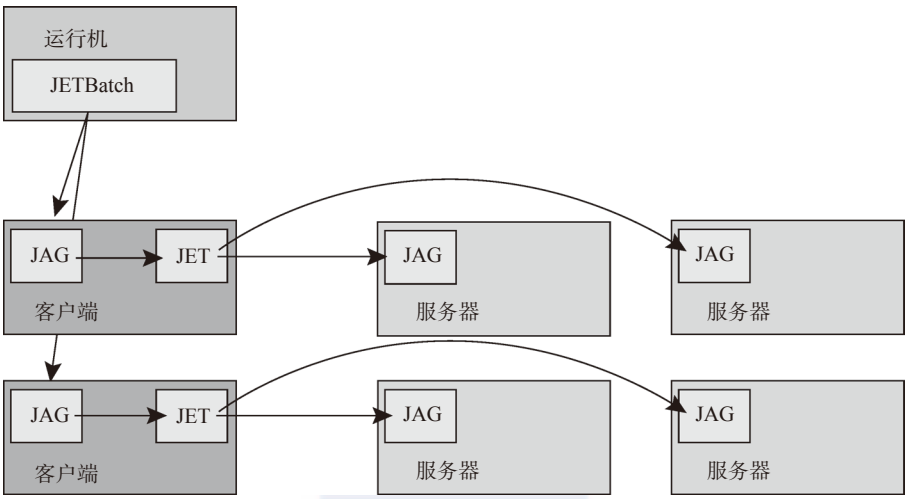


图 2-1　测试工具的架构（出自 http://kenai.com/projects/jet）

2.4.1　配置

测试配置：我们的测试是定义在一个数据库中的，并且可以单选、成组选择或者根据特定序列来进行选择。工具在每次测试之前都会进行一次初始化，避免前面的测试结果影响到后续的测试，工具在每次测试之后也会将测试环境清理干净。工具还自动将测试件进行收集和归档。

【小窍门】

把实施预处理和后续处理作为启动一个测试套件的一部分。

测试时应用程序的配置：可以对产品版本进行选择，包括调试版本和来自开发人员“沙箱”（sandbox）的本地版本。

服务器和客户端平台的配置：我们使平台的定义和在运行测试的平台组的定义变得简单。根据测试装置在一个平台组（例如，Windows Vista，64 位，JDK 6）里面是否可用而将其划分成不同的测试装置。我们可以用一到两个平台为服务器设置一个单独的配置，通常也可以用一个平台为客户端设置一个单独的配置。对于客户端和服务器来说，都可以选择不同的操作系统。

一个标准的测试需要 4 台机器：一台测试机器，一台客户机，两台包含被测数据库的服务器。

2.4.2　资源优化

通过添加更多的机器到测试装置池中，我们可以并行地运行测试。另外，测试具有排队机制：一个测试装置上的测试完成之后，候选队列里面的下一个测试就开始运行。

2.4.3 测试报告

这个内部工具创建了网站来记录测试报告，所有的结果在一个数据库中也进行了详细存档，这有利于我们建立详细的度量，比如下面的度量：

1）在哪些平台上会有一些什么样的 bug 及其出现的频率（可以帮助指定 bug 的优先级）。

2）每个平台上的一般信息统计。

3）测试中 bug 的检出率。

4）测试的冗余。

一个测试完成之后，会自动发送一个包含测试结果的汇总邮件，同时生成一个 XML 文件，它包含了用以导入到其他数据库或者报表生成器的所有必要信息。

该工具也使得从开源测试覆盖工具（EMMA）中导入信息变得更容易，而且让我们能观察到测试的质量——至少从表面上看是这样的。

2.4.4 故障分析

在测试失效分析达到 60% ~ 80% 正确率之后，才能对失效进行自动识别。比如，通过定义描述故障的模式和标记来进行识别，在大多数情况下，我们定义一些模式或者签名对故障进行描述，这些模式或者签名通常与测试产生的错误信息、测试名称和产生这条实效信息的测试语句直接相关。一个 bug 可能有多个标记，如果要对新的故障或者已知故障的新症状进一步进行人工分析，就需要新的标记信息。

用这个新工具实施的某产品的首次发布测试中，要求不论何种原因，无论是产品原因或者是测试原因，至少 75% 的测试运行的时候不会出现故障。最后，要求至少 96% 的测试运行的时候不会出现故障。

2.5 结果

该工具经过 3 年的开发和使用，在 6 ~ 10 个平台上有 200 个测试的发布测试，可以仅由一个人在 3 ~ 4 天内运行完成。这等价于（在实施自动化之前）在 16 天的时间里（16:4）20 个人（20:1）在一个平台上（6:1）运行了 40 个测试（200:40）。所以自动化测试过程帮助我们提高了至少 2400 倍的工作效率！

【真知灼见】

用一种最恰当的方式向希望知道结果的人们来表述你的成功（这里的表述是，“将效率提高了 2400 倍”）。

现在所有的测试人员都可以集中精力进行测试的开发和进一步的工具开发，跟刚开始需要进行重复性的工作相比，他们现在的工作也不会那么枯燥了。我们的产品和测试的质量都得到了极大的提高。开发人员和测试人员都得到了应有的相互尊重，并且他们互相通过挑战来激励

自己，这样使他们工作的兴趣更浓。

通过实施自动化，维护工作只占测试人员整个工作量的不到 1/10，而且比之前的工作量要低很多，部分原因是因为产品进一步开发中的其中一个需求是后向兼容的。这是一个罕见的机会，测试与测试工具本身因为新的功能而需要改变。另一方面，新的功能通常需要新的测试，在某些情形下用以替代旧的测试。

2.6 管理自动化测试

我们的测试过程在持续改进，并且我们为测试设计了一个可记录的生命周期，如图 2-2 所示。

测试被开发出来之后，会进行评审，如果审查通过，这个测试就会被包含到候选队列中（一个测试集合用来尝试是否应该包含到整个自动化套件中）。如果一个候选测试在一行中有 4 天都失败了，那么它会被提取出来重新进行开发。在测试本身没有任何失效一周之后，这个测试会设置为“有效”状态，并可以包含到每晚的或者每周的测试套件中。

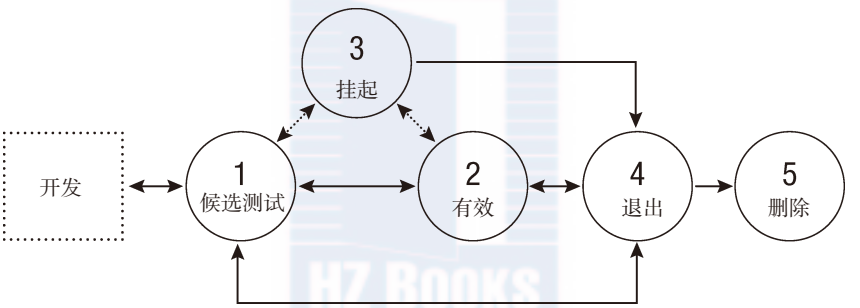


图 2-2 测试的生命周期

如果产品的功能改变了但是其测试没有更新，测试可以“挂起”。根据挂起的原因，测试将来可能会成为“有效”状态或者候选测试状态（故障的原因被修复之后）。

不同测试套件的内容会进行周期性分析。度量指标用来衡量运行这些特定测试的收益。根据这一过程的结果，一个测试可以从一个测试套件移到另一个测试套件（依据测试的运行频率），或者在某些情况下转移到“退出”状态。如果某一个测试可能不会再用到了，团队就会考虑删除它。

我们制定了很多度量标准，都使管理层非常满意，而且非常关注我们，并提高了团队的优先级。毫无疑问，相比于之前，他们对产品审批过程的信任有了大幅提升。

2.7 测试套件和类型

最后，我们用这个工具批准开发人员代码检入：在允许提交新的或者修改后的代码之前，他们必须在三个不同平台上运行一个最小的验收测试套件（Minimum Acceptance Test Suite, MATS）来测试其代码。通过实验，我们选择这些平台来发现特定的或者罕见的故障。这一步

骤有利于在变更被引入源代码之前，减少回归测试和失效的数量。这些测试的运行时间被控制在最短时间内，所以这些测试是有用的，在时间上并没有成为影响进一步开发的障碍。

【真知灼见】

提供及时的反馈，并且将日常开支减少到最小，为开发人员提供最好的支持。

该工具用作夜间的回归测试，这样开发人员白天一上班就能获得关于他们代码变更的反馈信息。这个测试套件包括大部分运行时间稍短的回归测试，并运行在有限选择的平台上：一般是在 3 ~ 5 个平台，并且运行时间为将近 12 小时。

我们通常还在 3 个不同的平台上进行每周测试，每一个测试套件运行 4 ~ 5 天。

这些回归测试具有很高的优先级，并获得了管理层的大力支持。这里出现的故障必须尽快修复。

除了这些回归测试外，其他测试是在候选批次中运行的。这些测试，是对新版本中新功能或者已变更功能的典型测试，具有更低的优先级，因为它们往往是由负责的开发人员和测试开发人员进行监控的。候选批次也包括夜间和每周运行的批次。项目团队可以根据需要定义更适合自己团队需求的测试套件。

这些测试在我们的内部测试工具上运行，性能测试与它们并行运行，并且与基准线进行对比。由于它们对测试和结果的特殊需求，它们都有自己的框架。这些测试通常仅在一个平台上运行。

发布测试一般包含以上描述的所有类型的测试，但是在至多 22 个不同的平台上运行。如图 2-3 所示。

此外，发布测试还包括在测试工具上运行的其他非功能性测试，比如：

- 长时间测试：不同的场景下运行时间至少为 10 天的测试。
- 扩展性测试：通过增加硬件来减轻负荷的测试，一般采用 24 台服务器和 12 台客户机来运行测试。

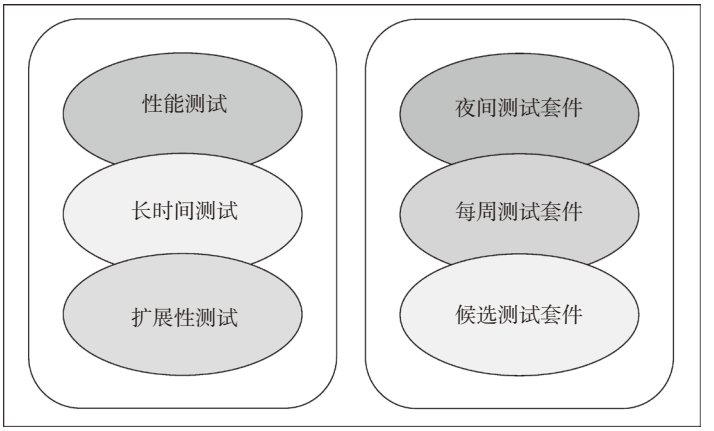


图 2-3　发布测试的内容

2.8 现状

该工具已经用于不同的数据库产品的测试当中，而且现在是一个开源工具，请参阅 http: // kenai.com/projects/jet。

2.9 在经过一段很艰难的时光后才得到的经验教训

在过去的三四年里，我们遇到了无数的困难：

- 软件测试变得非常成熟，有时反而会使我们忽略一些最简单的测试。比如，我们往往测试了复杂的 SQL 查询语句，却忽略了对在用户组之外创建新用户这类简单操作的测试。事实上，这种语句是不允许出现的，因为可能会由于在用户组的权限设置中没有考虑到这种情况而导致重大的错误。
- 我们过于关注大规模的自动化测试中的变化，导致有些非功能性测试有时并未得到足够的重视。比如，用户可能会获得只有专业人员才能看懂的错误提示消息，软件产品中经常缺少帮助功能，这些都是软件设计和测试中存在的问题。
- 使用随机生成的输入数据来进行测试时，有时虽然发现了严重的缺陷，但是因为测试人员能力有限，不能再重新生成导致故障的数据，所以并不能对调试过程产生任何帮助。经常讨论上述这种测试，一般是从经济的角度：它们通常需要额外的资源来辅助进行分析，并且在大多数情况下，并不能通过它们找到产生 bug 的原因，因此，也不可能通过它们找到产生软件缺陷的根本原因。
- 对自动化投入的评估主要看 ROI 的效果。如果不在比较的过程中引入其他因素，很有可能得出错误的故障报告。比如，我们要仔细地对结果进行比较，考虑到地域因素，有时要先进行转换再进行比较等。例如，当你在比较一台位于挪威境内的 PC 上的日期和一台位于美国境内的 PC 上的日期的时候，不同的时间格式（挪威为“日 / 月”而美国境内为“月 / 日”）导致时间显示的结果也会不同。
- 有时候通过软件来模拟物理故障并不是很容易，并且要模拟这些故障在多台电脑上几乎同时发生也是很困难的。此外，通过软件来模拟的断电和断网情形与实际发生的断电和断网也可能并不一样。
- 有时候可能会发生结果误报问题，因为测试时，即便遇到一个或多个失效也可能报告正确的结果。对于那些长期使用都未出现过故障的测试用例，往往更容易忽略对其正确性的检查。但随着时间的流逝，这些测试可能会不断积累许多错误，因此在报告中要不定期地对其正确性进行检查。
- 如果一些优先级比较低的次要 bug 没有立即修复，那么它们可能会掩盖一些主要 bug，而这些主要 bug 由于引入的时间太长，往往更难对其进行分析。
- 必须要插入和修改等待时间来保证在测试继续运行之前，前面的强制性过程已经完成了。用更新的硬件取代现有硬件通常意味着要对这一过程再一次进行同步。

【真知灼见】

预测可能会发生改变的事物，并使它们在必要的时候更容易改变（例如，保存同步时间的核心列表）。

- 在测试套件的某些部分中，期望结果模板用来与实际结果进行比较。由于这类模板需要大量的维护工作，因此我们试图将这些基于模板的测试改为基于断言的测试。
- 引入新的平台有时会产生一些问题，并需要大量的资源来解决这些问题。同时，对操作系统的监控力度也要加大。
- 对于基于 Windows 的测试，要关闭自动更新，并将更新放在等待队列中，在测试可以中断的那个时段之前，再运行这些更新。
- 我们要清楚正在运行测试的网络中所发生的一切。比如，每隔一个月午夜时候出现一些无法解释的故障，最后发现，故障是由前雇员的一台电脑上的夜间执行工作所导致的：这台电脑没有关闭，而是仍然非常活跃地向本地网络发送查询请求的垃圾邮件。

【小窍门】

回头看有些错误是非常显而易见的，但你如果没有想到，它们可能就会困扰你。

建议定期做探索性测试，你将会对所发现的问题感到非常惊奇，同时，有时候你还可以将这些经验用于新的自动化测试中。

2.10 如何使用自动化测试书中的建议

在开发自动化测试过程中，我们运用了《Software Test Automation》一书中许多有用的知识点：

- 在进行自动化测试工具开发之前，首先对工具进行需求分析并列出需求清单，我们对需求清单中的每一个需求进行讨论和评审，结果表明这是整个开发取得成功的坚实基础。在评审过程中，参与人员中有代表不同需求的关键人物：经理、IT 运营商、发布工程师、测试经理、开发人员和测试人员。
- 测试自动化只是从以下几个方面来对测试进行自动化：测试的准备、执行、核对、清空、存档、生成报告和度量。而测试执行之前的过程，例如，测试用例的设计等，是没有进行自动化的。
- 我们得到了管理层的大力支持来实施这项自动化工作，并且他们有着切实的期望。

【真知灼见】

管理层的支持是至关重要的，但是他们的期望必须要符合实际。

- 如果没有来自不同领域的杰出专家，我们就不可能取得成功。整个实现过程和解决方案都很复杂并具有挑战性。
- 幸运的是，在大部分产品中都没有要求进行 GUI 测试，这使整个自动化过程不会显得很冗长。
- 数据库中的 GUI 测试属于可用性测试，识别这种测试类型使我们取得了很多重大的改进，并使可用性测试延期。到今天为止，可用性测试还没有完成，但是因为考虑到这点我们受益匪浅！
- 测试工具的大部分开发是集中在 GUI 部分的开发。然而，在后面，GUI 几乎不会用到，因为所有的自动化都是通过命令行界面进行的。我们前期之所以会集中精力进行 GUI 的开发，可能是因为我们大脑中还存在进行手动测试的定势思维。

【小窍门】

工具中良好的用户界面可能在自动化项目的前期最有用。

- 经过本次自动化项目的实践，测试人员也学会了别的技术并在他们感兴趣的领域变得更加专业。有些测试人员更精通测试开发，有些则在测试的执行和生成报告方面更精通。
- 只有心中牢记不断取得进步这一目标，才能让我们大步前进。通过小组讨论来分析现有问题以及如何对其进行自动化，最终就会找到解决方案。
- 让所有的测试人员参加国际软件测试认证委员会（International Software Testiing Qualifications Board, ISTQB）的基础认证课程，这样有助于术语的统一使用和理解，从而有助于增进测试人员间的交流。
- 有时候项目中人员数量突然减少从某种程度上也可以促进自动化过程——使用更少人力资源的需求变得更为突出。
- 能详细地向整个项目中的其他成员介绍每一步是怎么实施的，很有用，因为相比于将它们整合到产品之后再进行测试，显然单个部分的测试更容易执行。

【真知灼见】

在走向成功时，步子要快，但也要稳。

- 通过自动化进行的测试是整个项目的核心过程，无论什么时候出现了故障，它们都会尽力地报告给整个部门的每个人。这可能会对开发产生负面影响，因为看到这些失效之后，开发人员在修改代码时会格外小心，虽然这种格外留心有助于提高产品质量，但他们可能要为此花费过多的精力和时间。对于软件产品来说，对于“怎样的质量才算足够好”是没有明确定义的，这可能会导致在开发中投入过多的精力。这仅仅只是我个人毫无根据的假设，并没有事实能证明这一观点，但是开发人员太关注质量这在软件行业里是不寻常的！

2.11 总结

刚开始的时候数据库产品和测试的质量都很差，但是在自动化测试开始之前都得到了显著提高。

接着开始了自动化测试工具的开发过程。首先，成立一个包含信息传递人员、专家和利益相关者的团队，进行需求定义。然后，用最专业的人员完成了开发，自动化测试也逐步实施，在这一过程中，每个参与人员都起到了非常重要的作用：工具开发人员、促变者、管理层、工具管理人员以及整个实施团队。

早期我们达到了开发这个工具的第一目标，然后随着时间的推移，完成了越来越多的目标。我们的效率至少提高了 2400 倍，并为公司开发了一款非常好的工具。在实施小的修复或者增强功能的时候，需要的维护工作量非常少，这也为我们的成功帮了不少忙。我们的硬件资源都达到了最合理的使用:大部分机器除了短暂的休息以外，都是 24 小时 / 天,7 天 / 周地工作。

对我们来说，这就是最终的自动化测试。

2.12 致谢

首先，要感谢 Yngve Svendsen 和 Jørgen Austvik 在我写这篇案例研究时提供了非常有用的建议和反馈信息。此外，还要感谢所有为这个项目的成功而努力的人们，尤其是 William Franklin，感谢他对我们自动化测试的大力支持和鼓励。同时，也要感谢本书的两位作者，感谢他们给了我公布这个案例研究故事的机会。

第 3 章　移动到云端：TiP 的演化——在线的持续回归测试

Ken Johnston
Felix Deschamps

来自微软公司的 Ken Johnston 和 Felix Deschamps 讲述了他们是如何通过在云端实施自动化测试，从而将基于产品的自动化测试变更为基于服务的自动化测试的。微软的邮件服务产品 Microsoft Exchange Server 中绝大多数的测试已经实施自动化了，而且其中大量的自动化是可以重用的。大多数测试人员对于“在线[⊖]测试”（Testing in Production, TiP）这个概念还比较陌生，但是这一章解释了为什么进行持续监测是必要且有益的，同时为那些也正在考虑这么做的人提供了一些非常有用的小窍门。案例发生在美国，经历了 3 年多的时间，毫无疑问，使用到了微软公司的一些工具，如表 3-1 所示。

表 3-1　案例研究特征

特　　征	本案例研究
应用程序领域	Exchange Server 2010———企业级的电子邮件服务器
应用程序规模	非常大
地点	美国、华盛顿和雷德蒙德（微软总部）
生命周期	传统的多年发布周期，加入敏捷元素后，产品开发周期变为 6 个月
项目人员数量	>500 人
测试人员数量	>150 人
自动化测试人员	>150 人
时间跨度	大约 3 年
年代	2007—2009 年
工具类型	Microsoft Visual Studio Team System, 其他工具是内部开发的
是否进行试点研究	否
ROI 测量	无；我们被迫去适应产品策略的改变，所以从来没有测量 ROI
是否成功	是
是否还在运行	是

⊖　即实际产品运行环境。——译者注

3.1 本案例研究的背景

你会将一个价值十亿美元的业务作为赌注放在云端吗？

Exchange 2007 的发布对我们团队来说是非常令人振奋的。在刚推出这款产品的时候，我们已经成功地对产品架构进行重新设计，以使它能在本地的 .Net 平台上运行，从而转化到通过“角色”（role）来支持服务器管理，把 64 位机器作为目标平台，采用 Windows PowerShell 作为自动化服务器管理的工具箱。我们已经做好了迎接下一个挑战的准备！

那时，微软的总架构师 Ray Ozzie 正准备构建微软云计算未来的发展蓝图。我们都很清楚，云计算将会越来越重要，而正好 Exchange 产品也在寻求这种百年一遇的重大商机。通过云计算，我们可以在为客户降低成本的同时，给他们提供更好的服务，同时，我们也可以将 Exchange 产品中的业务发展壮大。

实施云计算这一举措也为我们引入了更多的问题。我们如何构建一组特性来吸引 IT 专业人员升级到 Wave 14（发布名称为 Microsoft Exchange Server 2010）版本，同时又能以某一服务作为目标呢？我们又该如何对产品架构重新进行设计，以使它能够在运行遍布全球的服务前提下实现规模经济效益呢？何况现在我们不仅仅需要构建所需的软件，而且还要构建数据中心，这从何做起？

我们进入到完整的原型和发现模式。我们学习了很多新的 web 服务概念，以便通过冗余将服务进行纵向和横向扩展：

- 我们需要对多租户进行架构设计，这样单个服务实例就能服务于多个客户组织（租户）。
- 我们将为了满足某个功能的一组逻辑上的机器确定为不同的单元（有时称为小群组），这样有利于规划所获取的单元。
- 服务必须是遍布全球的，以支持业务连续性规划（Business Continuance Planning，BCP），并通过区域市场减少延迟。

我们学习了如何使服务成为一种业务，以及为什么必须对固定资产费（Capital Expenditure, CAPEX，如购买新的服务器开销等）、运营费用（Operational Expenditure，OPEX，比如给经营服务的员工发放工资等）以及总销货成本（Cost Of Goods Sold，COGS）进行管理。懂得了服务器并不是像变魔法似的出现在数据中心，恰恰相反，我们必须提前一年就拟定采购计划，同时，对整个数据中心的空间大小、供电和网络配置也要进行安排。对于那些曾经从事过一段时间的服务工作的人来说，这听起来都是些基本的知识，但是我们想让整个团队的人员都清楚这些服务理念，并真正懂得我们将服务业务实施云策略的意义。

受到以上学习过程的启发，我们决定使用以下的一组原理来推动 Wave 14 的发布：

- 复用和扩展现有的基础设施。
- Exchange 将保留一个代码库。
- 我们的团队是一个整体，不会分裂为服务工程团队和服务运营团队两个独立的团队。

3.2 将测试移到云端

作为专业的测试团队，我们很容易厘清该如何同时地测试一台服务器和一项服务。但是如何将现有的自动化测试丰富的资产应用到服务空间？最终的答案是：在线测试（Testing in Production, TiP），在当时看来这是我们之前决不会做的事情。我们的起点是一些现有的工具和资产，如表 3-2 所示。

表 3-2 Exchange 产品测试的现有工具和资产

资产和过程	用 途
Topobuilder	用于构建多种高度复杂的服务器拓扑结构的自动化工具
测试实验室中的 5000 台机器	在这些机器上每天多次运行 70 000 个自动化测试用例
具有长集成阶段的基于里程碑的开发	在代码完成之后和大规模测试和稳定阶段，验证大量的场景
“狗食”（dog food）	“吃自己的狗食”[①]；这不仅对自己有好处，而且能使产品对我们的用户更有用（接下来会进行解释）
复杂的分支管理	带有独立源代码控制分支的多个产品单元最终会集成到单个分支中并发布

① 用于描述软件公司使用自己的产品这种情况。

因为 Exchange 是世界上功能最复杂的产品之一，所以我们已经建了一个庞大的工程流水线来开发和测试我们的产品。其中仅测试实验室就有大约 5000 台机器用于运行日常的测试和预检入（pre-check-in）的测试。在这些机器模拟测试场景（如具有多级动态目录的网站、不同的故障转移配置、多角色配置等）的复杂拓扑结构中，我们每周对 Exchange 的 80 000 次自动部署进行相关测试。到 2007 年年末，我们已经编写了将近 70 000 个自动化测试用例来对产品进行验证，我们每天都运行这些自动化测试用例，且每次提交代码之前必须运行其中的某些测试用例。我们使用 Microsoft Visual Studio 作为我们自动化测试工具的开发环境，大部分是使用 C# 语言进行开发的，但也有少部分是使用 JavaScript 和 C++ 语言进行开发。

与大多数微软的产品研发团队一样，我们要吃自己的狗食。在很多文章和博客上都谈到“如何把产品当狗食来吃”的概念。在这里，指整个 Exchange 团队将他们的邮箱装在团队的 dog food（beta 版）环境中或者基于云端的 dog food 环境中，为了增加复杂性，我们让两个 dog food 环境无缝地协同工作，即便每两周都要升级到最新版本，也是如此。

这一级别的自动化允许我们建立单一的主代码分支，并使之在多个产品单元之间具有一致的文件版本。这是通过维护一定数量的编译错误（构建过程中断）和最小范围的回归来完成的。在 dog food 中有了自己的邮箱，可以允许我们很快地找出被测试忽略的功能问题，并能增强对正在构建的产品性能的自信心。因为我们正转向基于服务的自动化，所以继续获得这种自信是非常重要的。我们采用的新工具和模型如表 3-3 所示。

表 3-3 新的工具和模型

创 新	描 述
OneBoxes	在每个服务器 / 操作系统上运行主要服务组件的实例，这使得我们可以并行地、互不干扰地完成测试。对于 Exchange 来说，这意味着在单个 OS 中运行 Exchange 存储、SMTP、POP3 和 Web 前端程序。我们还为主要的平台组件（如 Windows Live ID）实现了 OneBox
服务 dog food	Exchange 团队的一部分人员完成服务 dog food，而其他人员完成企业级服务器的 dog food
TiP 服务	将现有的通常运行在测试实验室的 70 000 多个自动化测试用例用于对产品的测试和验证
TiP OneBox	在开发过程中，尽可能早地在产品数据中心中运行 OneBox 环境

我们的每一个新工具和过程都吸取了推出 Exchange Server 产品时的经验。对于我们来说，解决如何与别的服务进行集成，以及如何在产品数据中心进行传统的测试活动，是一个新的转折点。

3.2.1 TiP 测试策略的目标

我们设定的 TiP 测试策略目标是：

- 积极主动地发现产品线所存在的问题；
- 积极测试服务 dog food；
- 验证新的产品上线；
- 确认配置变更或升级不会导致任何中断；
- 通过我们的附属服务进行合作方注销测试，如 Windows Live ID；
- 衡量并帮助提高现有的系统中心运行管理器（system center operations manager，SCOM）监控方案；
- 发现潜在的缺陷；
- 工程团队可以使用这些测试来了解产品试验。

有了适当的目标之后，就要将它们作为测试——尤其是 TiP 测试的指导原则，并保证它们与整个 Exchange 项目的目标是一致的。我们最终选中了一组专注于效率和重用的原则。效率原则是指在复杂度最低的环境中用最快的速度找出正确的 bug 集合，OneBox 是最主要的资产。另外，我们还选择将许多 OneBox 测试环境搭建在数据中心。使用以上方法时，会出现很多因数据中心网络和安全设置所产生的额外 bug。重用变得至关重要，因为它意味着将现有的资产和过程扩展到数据中心和产品中，用于更快地产生好的测试结果。

3.2.2 指导原则

我们使用以下原则来指导我们的开发：

- 没有独立的团队，现有功能的测试团队都要参与。

- 产品中同样的代码库意味着我们应该尝试着复用整个产品和测试资产（自动化测试、测试装置、测试工具和流程），但是现在就在产品线上。
- 在实验室完成功能测试，意味着我们要在产品线上做进一步的测试，通过扮演客户的角色来验证用户体验。一些 TiP 方法利用可测试性特性深入到在线网站的堆栈中去，但是在最初实施时，我们坚持使用模拟终端用户进行黑盒测试的方法。
- 对测试场景的设计应该考虑其测试的广度，而非深度（即尽可能多地描述测试场景的覆盖面，同时测试场景的执行尽可能地快）

【真知灼见】

明确自己的目标，并为自动化测试设计可以实现的指导原则。

3.3 如何实施 TiP

第一步是让测试经理也支持这些指导原则，然后再与团队的资深成员紧密合作，并根据所有主要功能区对原则进行评审。这一过程保证了我们之间没有隔阂，并且能够很好地在不同领域利用测试人员的专业技术知识。这个虚拟团队定义了 40 多个场景，这些场景代表了视为最重要的功能的宽度。

【真知灼见】

避免白费力气做重复的工作；尽可能地复用现有的自动化测试件。

第二步是决定如何将上面的 40 多个场景在整个产品系统中具体实施。如前所述，我们要尽可能地重用已有的资产，所以我们集中精力尽快定义和开发出所需的测试。最初实施时，我们决定使用现有的测试执行框架来运行测试，那样的话，就可以重用现有的测试报告工具、机器管理等，这也使得我们可以利用现有的自动化测试库。

我们的第一次实施的具体结构见图 3-1。每小时执行引擎都会自动部署一台新机器，并安装相应的客户库、工具和测试，也安装一个“云定义”文件，该文件以一种通用方法描述了目标环境。测试本身并不知道目标环境的任何信息，通过这种抽象的方法我们可以指向某个数据中心、某个逻辑单元或者某台特定的机器（实际上现在是在预检入工作流中完成的）。

【小窍门】

另一种级别的抽象：将测试从机器和它们所运行的环境中分离出来。

图 3-1 是 TiP 系统拓扑结构的第一个版本，具有如下特点：

1）部署一台自动化测试主机，在特定的数据中心运行测试。

2）将测试结果发送到 Focus 测试执行（Focus Test Execution, FTE）服务器上，它将对测试

结果进行处理（Focus 是工具的名称）。

3）结果将保存到一个公用数据库上。

4）在运行过程中，TiP 诊断服务周期性地对结果进行收集。

5）收集的结果会返回给数据库。

6）遇到问题时，产品 bug 会自动显示出来。

7）诊断服务给管理区发送一个请求信息，包含分页调度和报警信号的逻辑信息。

8）请求信息被发送到 SCOM 根管理系统（Root Management System，RMS）进行处理。SCOM 的警报解除了，同时启动相应的响应处理机制。

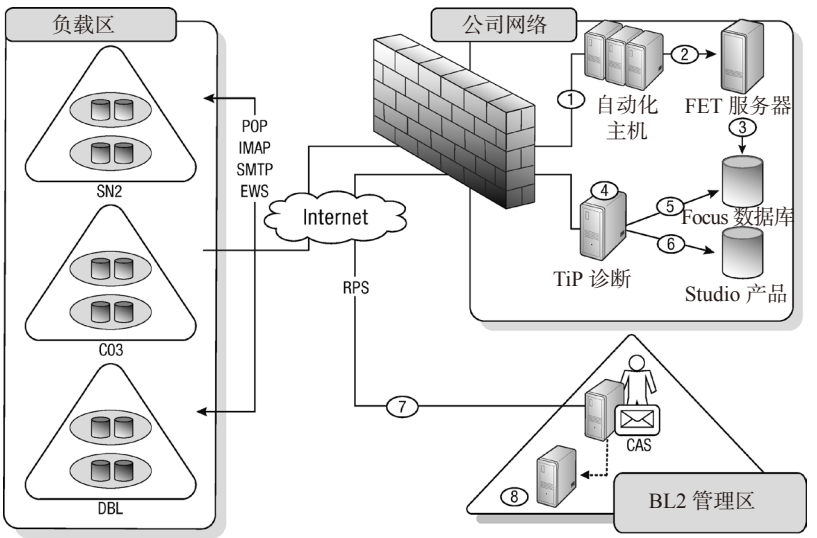


图 3-1　Exchange TiP 第一个版本的系统拓扑结构

最初，在将服务提供（为用户注册和新建邮箱）给新用户之前，我们通过测试来验证服务的新部署。后来我们发现，尽管如此，与传统的以软件为中心的世界中我们只需保证软件质量不同的是，我们现在所经营的服务是一直变化的。不仅配置（域名系统（Domain Name System, DNS）、补丁、新的租户等）经常发生变化，更新的变化也很多，这意味着随着时间的增长，可能会遇到没有测试过或预料到的更新或者小的配置变动。根据我们的经验，即便是对产品来说安全的改变也可能会使服务中断。这也是为什么我们决定采用连续运行测试的方式，而不仅仅只是在进行部署的时候运行一次。

TiP 具有一点类似产品服务监控器的作用；然而，相比于传统的轻量级服务监控，我们所运行的测试集更深入，端到端（end-to-end）场景鲁棒性更强。同样，TiP 测试就变成了煤矿中的金丝雀，能对潜在用户面临问题提供更早的警报。过去是使用其他构建在 Microsoft System Center 顶端的基于代理的监控解决方案，然而，这些代理只是停留在单机器层面。TiP 测试作为最终用户来运行，并且当它们使用性能降低的时候，会发出警告。

【真知灼见】

不要仅仅通过一种途径来实施自动化，尽可能使用多种有用的方法。不同方法之间可以互补并且往往比单独使用一个方法更有效。

作为我们整个决策的一部分，我们决定停止使用第三方服务，像 Gomez 和 Keynote。虽然在整个决策中非常重要，这种类型的监控关注的主要是一些比较片面的场景（比如登录）的服务可用性。与此同时，TiP 场景的宽度比其他测试要小（高级测试只有 40 个测试用例，而实验室运行的端到端的场景中有 70 000 多个测试），所以比一般服务的深度肯定是要大些。

通过使用自己的基础设施，例如，我们可以很容易地给手机的 ActiveSync 协议添加新的验证信息，这在传统的黑盒监控环境中是很难做到的，因为协议具有复杂性。另一方面是敏捷度，根据产品环境和测试本身，在数据中心我们可以进行更改并对更改作出快速回应。因此，正如只关注单机器的 SCOM 监控基础设施一样，这些 TiP 测试对外部黑盒测试是一种补充，而不是取代它。

3.4 每月服务评审记分卡样例

每个月都会对总体的服务质量（Quality of Service, QoS）进行一次评审，同时，根据上个月的结果进行有针对性的改进也是要进行评审的。这种评审有利于持续改进总体服务，并帮助改进 TiP 套件。这种每月的评审是由经理发起的，并且他每个月都参与其中，推动问答 (Q&A) 环节的进行。这也是他每个月深入实况网站并对其进行改进的一次机会。经理的支持和带动作用对任何一个类似这样的项目都是至关重要的，而我们从一开始就很幸运。图 3-2 所示是一个记分卡的例子。

记分卡第 61~144 行	事故及调整情况			
每月用户影响调整情况提供的调整 • TiP 测试失败 •对 SyndC 的响应 • SyndC 的登录 •大量的假报警 OWA 调整 •泄漏（app 池） •可操作的负载 数据中心调整情况 • 产品中添加新服务器时发生的明显的假报警 •可操作的负载 下一次回归测试泄漏自动化的长期响应 发布问答				

度量	12 月	11 月	10 月	
几分钟内响应时间中的业务时间段	2.9	10.6	3.6	○
几分钟内响应时间中的中断时间段	5.7	9	8.2	○
业务时间增长	34	52	30	○
中断时间增长	18	19	14	○
总的调整	**52**	**71**	**44**	●
OWA	1	8	12	○
POP, IMAP, ActiveSync	17	48	1	○
Prov, Gal, Recipients, OAB	21	49	16	○
HA	3	51	3	○
Store	3	34	9	○
操作系统	9	19	10	○
数据中心	35	17	10	●
MOMT	1	3	2	○
传送	1	1	0	○
影响事故的服务	**0**	**3**	**0**	○
监视器遗漏的	3	4	4	○
外部状态交互	-	-	-	

图 3-2　调整记分卡中的事故和调整情况

3.4.1 阅读记分卡

当你看到 TiP 记分卡时，提出的第一个最典型的问题就是：怎么阅读记分卡？这是一个很好的问题。

首先需要注意的是，图 3-2 中所示的记分卡只是每月进行评审的幻灯片中的某一页。首先将每月的数据放到一个很大的 Excel 数据表格中，然后高级管理层和其他团队将 Excel 中的每项数据放到一页幻灯片中进行评审。

图 3-3 显示了将记分卡按不同的区域进行分解后的情况。区域 1 提供了指向 Excel 表格中具体行的标记。因为幻灯片中空间有限，所以只显示了最近 3 个月的数据，但事实上，Excel 电子表格包含的不仅仅只是这 3 个月的数据。在评审过程中，每个人都有这个 Excel 电子表格的一份副本，并通过在自己的笔记本电脑上进行评审来对幻灯片的内容进行更新。

区域 2 是细分（drill-down）后的区域的名字。在给出的例子中该区域的名称是“事故及调整情况”。

区域 3 是从 Excel 表格报表中拉出的数据。包括度量的名称以及最近 3 个月的数据。在图 3-3 所示的样例中，数据根据事故数量和服务组件，按月显示。当整个 Exchange 云端服务的某个组件发生了一次故障，并需要人工干预来进行解决，则称为一次事故（incident）。通过最近 3 个月的数据，即便在已经达到每月目标的前提下，还可以帮助我们确认服务的发展趋势是好是坏。

图 3-3 事故记分卡区域中的事故和调整

区域 4 是整个记分卡最重要的部分。在每个月评审之前还有一个预评审，是由负责改进该区域服务的工程师进行的。在遇到事故和调整的情况下，测试、开发和运营团队中的成员都会派代表参与预评审。他们分析数据，找出异常值和负面走向线。风险区域和关注区域分别用绿色和红色的圆点标记。在图 3-2 中，黑色的实心圆点代表红色，或者是 PPT 幻灯片中应关注的区域。有时候他们知道某一个度量的趋势走向不好的具体原因，但是更多的时候，他们只能进行猜测。此时就要依靠虚拟小组的成员来找出负面走向度量和异常值的根本原因。上述调查的结果就是图 3-3 中记分卡区域 4 的内容。通常，如果造成负面走向的根本原因是已知的，那么区域 4 中的内容就是一些总结性的建议补救方法。

【真知灼见】

对报表进行裁剪，使它仅提供你所需要的有用信息。

3.4.2 对事故和调整报表的处理

根据事故和调整记分卡，可以分析各个方面引起的事故。引起事故的原因包括 SCOM 服务器级别的监控器、TiP 服务级别的监控器，以及与第三方监控一起运行的一些监控器，旨在保证我们与全球市场都有联系。影响我们减少用户方面 bug 的能力的两个主要因素是：一是监控过程中遗漏的真正问题的数量和严重性，另一个是等待时间 (Time To Engage, TTE)。在整个行业和微软公司内部都有很多计算 TTE 的公式。对于 Exchange 来说，TTE 是指从产品事故开始到找到合适的工程师（开发人员或测试人员）着手修复该故障所花费的时间（以分钟计算）。一般来说，不管是在业务时间还是之外，导致 TTE 很慢的最典型的原因是监控器遗漏。这两个度量紧密相关，并且是每个月关注的重点之一。它们中只要有一个出现问题，我们就要考虑需要更新哪个监控方案（SCOM、TiP，或第三方监控），有时候会给这 3 种监控方案都增加监控器。

TiP 功能可用性记分卡用来提供粒子级别上服务可用性指标。可用性是通过以下公式来计算的：

$$\text{可用性 \%} = \frac{\text{每月总的时间（分钟）} - \text{故障停机时间（分钟）}}{\text{每月总的时间（分钟）}}$$

通过为每个特性运行 TiP，我们可以发现非客户影响的小的服务中断的发生，如 ActiveSync 中断。子服务中这种短暂的中断可能并不会对客户产生影响，但是却代表了服务的风险和退化。间歇失效或者（挂起）队列，与服务提供一样，通常都是可以在这个记分卡上显示出来的，但是并不是在关注调整的那张记分卡上（见图 3-4）。

记分卡第 142~177 行	客户体验
服务提供和 ECP 上个月合作方负载产生的可用性问题这个月已经修复了 上个月的 IMAP 日志文件已解决并恢复到 100% 可用 新问题：Powershell 脚本出现了间歇性故障 改进 • 添加内容分发网络和 DOMT 测试来处理最近的事故	TiP 功能可用性

TiP 功能可用性	12 月	11 月	10 月	9 月	8 月
服务提供	99.93%	94.01%	100.00%	100.00%	100.00%
RBAC	99.93%	98.81%	100.00%	100.00%	99.93%
Outlook	100.00%	98.81%	100.00%	100.00%	99.95%
ECP	100.00%	97.18%	100.00%	100.00%	99.97%
Mailflow	100.00%	94.01%	100.00%	100.00%	99.50%
ActiveSync	100.00%	99.73%	100.00%	100.00%	99.63%
UM	100.00%	99.68%	100.00%	100.00%	99.68%
Calendaring	100.00%	100.00%	100.00%	100.00%	100.00%
Web Services	100.00%	100.00%	100.00%	100.00%	100.00%
OWA	100.00%	100.00%	100.00%	100.00%	100.00%
POP	100.00%	100.00%	100.00%	100.00%	100.00%
IMAP	100.00%	89.87%	100.00%	100.00%	100.00%
Remote Powershell	99.93%	100.00%	100.00%	100.00%	100.00%

图 3-4 TiP 功能可用性记分卡

【真知灼见】

经常利用自动化测试生成的信息来监控服务的发展、寻求进一步提高、保持自动化优势的前景，这是非常重要的。

3.5 Exchange TiP v2——将 TiP 迁移到 Windows Azure 云端

虽然第一个版本的服务毫无疑问地已经为公司带来了收益，并且是度量和改进在线服务的主要工具，但是我们有一个大胆的计划，那就是处理当前服务存在的问题，并使它具有更大的价值。第一个版本中最需要处理的 3 个问题是：

- 将执行移到公司网络之外，来处理雷德蒙德地区的代理防火墙问题。
- 将测试执行的频率增加到 5 分钟一次或者更少时间。
- 将覆盖扩展到所有的数据库可用性组（Database Availability Group, DAG）和邮箱数据库。

我们考虑了几种方法来解决上述这些问题，如表 3-4 所示。

虽然我们还想维持现有测试执行用具的简单易操作性，但很显然的是，没有任何潜在可用的环境能够支持它。同时，与用具相关的所有报告、失效调查和工作流对于复合结果并不是非常有效，这使我们得出了一个结论：必须为测试更换执行夹具（harness）。最后我们决定将 TiP 基础设施移到具有平台优势的 Windows Azure 上面的运行。最值得骄傲的是，因为我们设计测试架构的方式，所以移动到另一个平台的时候不需要修改（事实上，我们只需要简单地将测试移到新的夹具，而不需要重新编译）。

表 3-4 TiP 框架潜在执行环境中的决策矩阵

需　求	整套直接互联网	边缘网络	Azure
公司外部	绿	绿	绿
能力	黄	绿	绿
成本	红	黄	绿
可管理性	红	红	绿
执行用具	红	红	红

将 TiP 框架从测试实验室移到产品中，正是采用了 TiP 中将测试基础设施从实验室移到数据中心的思想。在这种情况下，我们进驻云端并构建了一个可扩展的、灵活的 TiP 服务。

图 3-5 中的框架以插图的形式说明了现在是如何执行产品测试的。通过使用 Windows Azure，我们不仅具有了覆盖现有标度单位的能力，还具有了对之前无法预期的新场景进行模拟的功能（比如，在提交给客户之前，即便没有成千上万的用户，仍然可以对数百个用户并发地使用产品进行模拟）。随着时间的推移，这个框架当然会不断演进来满足日益增长的需求。

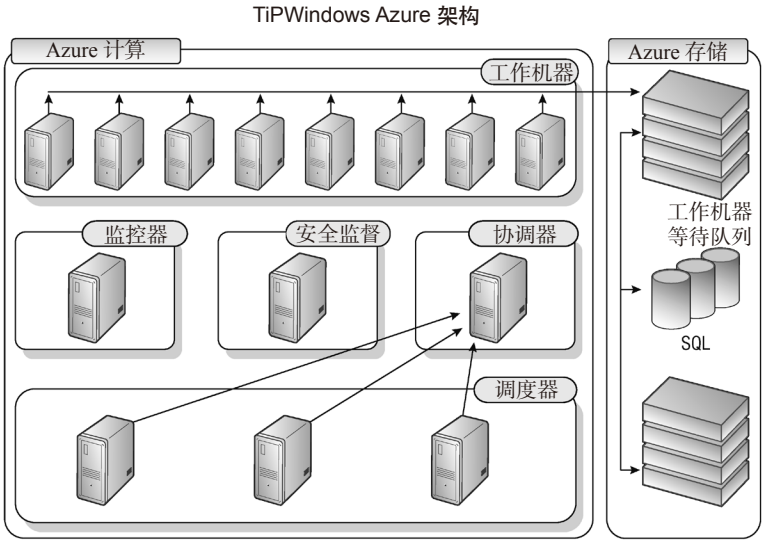


图 3-5　Exchange TiP 第二个版本的系统拓扑结构

3.6　我们的心得

TiP 测试使我们可以找出数据中心上线过程中引入的问题和有时在产品中客户还没发现的问题。随着时间的推移，还可以让我们找出潜在问题、性能和总体功能表现的非常有趣的发展趋势。

3.6.1　合作方服务相关的问题

对 TiP 测试至关重要的一个组成部分就是：找出我们所依赖的并在其上进行构建的与合作方服务相关的问题。我们的服务，与几乎所有其他云端服务一样，依赖于许多特定领域相关的不同服务。例如，我们依赖 Windows Live ID 来提供专门的身份验证层。又如，命名空间管理和提供是依赖于在线域名服务的。在找出上面这些通过别的方面无法捕获的依赖问题方面，我们的测试是非常有帮助的。

3.6.2　云端监视的挑战性

Exchange 作为服务很独特的一个方面是它的目标客户是 IT 专业人员，他们在升级到新的服务时总是非常谨慎，而且很多大客户都定制了一些写在 Exchange 顶层的产品。此外，有一些客户在混合模式下运行，而另一个客户仅仅只在公司的 Exchange Servers 中运行，也有一些是在云端运行的。在处理单个云端中版本的一些值得注意的、更持久的变化方面，标准监控解决方案设计得并不是很好。TiP 测试在流经系统时，其版本和配置变化使我们能更好地在异

构的云端测量和保证质量。这些挑战对于很多云端服务来说是非典型的，如 Bing、Facebook、Twitter, 更多的是使用同构的和松耦合的服务架构来获取连续的部署模型。

尽管如此，我们的服务是分层的，且依赖于比 Exchange 团队更快速的发布周期。网络本身就是随着新的 Firmware 版本和访问控制列表（Access Control List, ACL）的改变而不断更新的一个层。当那些层发生灾难性的改变时，由监控服务（如 Gomez 和 Keynote）提供的黑盒方法会向运营中心发出警报，但是对于间歇型或边缘情况却不会发出警报。TiP 使我们可以在这些依赖问题变成灾难性问题之前深入分析并捕获它们。在某些情况下，我们甚至可以在合作方服务升级过程中捕获问题，这样就可以提醒他们将这些变更进行回滚。

3.6.3 在线 TiP 测试所发现的一些问题

以下是通过 TiP 测试在产品中找到问题的一些样例：

- 都柏林（爱尔兰共和国的首都）的服务提供的一个队列挂起了，但是监控器并没有检测到。
- 检测到服务提供的延迟。
- TiP 检测到了 Hotmail 运行中断，Exchange 团队可以暂停并等待 Hotmail 的修复。
- Live ID 运行中断影响了 Exchange Cloud 的客户；Live ID 的运营需要进一步加强。
- TellMe, 一个可以将电话在我们的系统和手机交换器（Quest/Verizon 负责的地上通信线和 T-Mobile 的移动电话）之间转换的 VoIP 网关系统，需要对最终用户连接场景的运行中断进行监控。TiP 是找出集成试点测试中所使用的电话号码故障的唯一方法。
- 也常用 TiP 测试来鉴定发生随时间不断流逝的间歇失效的根本原因（随着时间流逝出现故障的百分比，而不仅仅只是在单个事故中出现故障）。

3.6.4 聚集处理结果中的“噪声”

我们学到了很多关于对一个实时服务如何执行测试的知识。学到的第一点就是，这种自动化的运行并不简单。例如，因为是在公司防火墙后面运行的，所以只好按照雷德蒙德的代理服务器的路线来发送请求。这些代理服务器并非旨在拥有这种用途，所以导致了有时会出现请求丢失、主机查找失败和其他奇怪的网络故障等。这很快导致了第二种实现方法的出现，即在产品系统中运行自动化测试时，重要的不是个体的通过或者失败的结果，而是随着时间推移，这些结果的聚集体。聚集可以帮助识别一个问题是持久中断问题还是仅仅只是一次网络故障，因此可以将噪声级减少到足够低，这能对服务安全性进行更精确的提高。同时还可以帮助预测未来的发展趋势，而只通过简单地适时看一看统计结果是无法做到这些的。

【小窍门】

对细节过程进行监控是非常重要的，但留意总体趋势也同样重要。

当你注意到测试一个小时运行一次的时候，上述最后一点（关于噪声消减）就变得十分重要。这一频率受到测试执行框架上的内置假设的影响，即关于测试通过之后怎么配置（比如，假设每一个测试运行执行都必须出现在一个新部署的机器上）和关于测试本身是如何执行的。我们发现那些假设因为很多原因存在一些漏洞。

它们一个小时运行一次的另外一个原因是担心消耗太多的生产力。我们发现一个服务必须要留出一些额外的设备来支持实时网站监控、使用过程中的高峰和低谷、拒绝服务攻击和成长。如果以一个增长的频率，比如每 5 分钟一次，在每个产品簇中运行一个自动化功能测试集合来对服务的边缘进行检测，那么运行的这项服务的时间就太密集了，需要增加设备。但事实上，对许多 IT 组织来说，采购和安置专用硬件的成本太高，以至于他们觉得准备额外设备留作备用是很不可思议的。然而，随着转移到进行云计算并具有了对电脑和存储资源的动态增长能力，很多组织都能负担得起通过一些合理的额外设备构建一个服务以用于在线测试。

3.6.5 易犯的错误

我们得到的一个教训就是，测试自动化比由外到内的基本监控更完整，并且由于这些测试的质量不错，团队很快就想将 TiP 系统作为一个加强的监控解决方案。但问题是，我们不可能对频率为 1 小时的测试进行适时的反应，因为收集足够的样本来分析一个问题是否真的需要花费的太长时间。另一种办法是在测试中实施重试逻辑（retry logic），这样有可能进一步降低通过率并有可能因为瞬态问题而给你提供假阳性的结果（例如，因节流机制而引起发送信息失败）。

我们得到的第二个教训是，当处理横向扩展的产品系统时，如果没有一个类似的自动化测试横向验证策略，就有可能导致一种虚假的安全感。在最初的实施过程中，我们为每一个规模化单元的每个人创建了单个邮箱账户集合，问题是，规模化单元还有进一步的粒度分割，因为它们是由多服务器、可用组织和其他最初未说明的资源组成的。事实上，这意味着每个站点只能覆盖一个可用的组织或者邮箱数据库。这会导致事故不被基础设施所捕获的情形，因为规模化单元受到影响的那部分并不是我们所覆盖的那部分。

除了作为回归测试的金丝雀之外，TiP 测试像我们的系统一样，应该将其当作更健壮的持续改进项目（continuous improvement project，CIP）中的一员来对待。与大多数 CIP 一样，高层管理人员的参与和带动是成功的关键。管理人员与团队的同心协力将保证工程团队获得支持以改善产品服务、弥补 TiP 和整个监控解决方案中其他元素（单服务器并且由外到内）的空缺。

3.7 总结

在线测试在这个快节奏的在线服务领域中是很有必要的，测试实验室中的自动化测试应该扩展到产品中去，因为没有哪一个服务是封闭的孤岛，持续执行那些贯穿和验证主要端到端

（end-to-end）场景的测试用例，这将会增强简单的单服务器（SCOM）或由外而内（Gomez or Keynote）可用性监控。持续回归测试具有类似监视器的作用，可以针对服务运行中断向产品服务团队发出警报，而且该方法对于早期鉴定非用户影响的间歇性 bug 特别有用。

虽然本案例研究是基于 Exchange 的，但测试自动化的趋势，尤其是丰富自动化场景，在微软服务团队中发展非常迅速。同时，在产品云端运行这些解决方案的趋势也越来越受到关注。我们期待通过自动化测试来加强监视作用的这类活动会，在整个微软延续下去。

【小窍门】

在云端进行测试可能是你将来工作的一部分，从本章作者的这些经历中学习知识吧！

3.8 致谢

感谢 Keith Stobie 和 Andy Tischaefer 的同行审校，并感谢 Karen Johnston 对本章进行审稿。

Experiences of Test Automation
Case Studies of Software Test Automation

自动化测试不再是奢侈品而成为软件测试的必需品。随着应用程序和系统规模越来越大、越来越复杂，仅仅依赖手动测试已经无法全面地测试系统。随着技术的变动，越来越多的公司加入到了敏捷开发的阵营中，必须对这些程序进行测试，而且速度要快。测试自动化是非常基本的需求，但有时糟糕的自动化只会适得其反——如何才能知道应该在哪里进行自动化呢？

作者Dorothy Gramham和Mark Fewster之前写的《Software Test Automation》这本书在自动化测试领域影响深远，它为很多公司指明了测试成功的方向。本书讲述了一些公司应用自动化的经历，其中涉及了很多项目，从复杂的政府部门系统到医疗设备，从SAP业务过程开发到Android移动应用和云计算。本书讨论了管理方面和技术方面的因素，讲述了成功和失败的例子、卓越的想法以及灾难性的决定，为读者提供了可以借鉴的经验教训。

本书对于那些考虑、实施、使用、管理测试自动化的人来说是无价之宝。测试人员、开发人员、自动化人员以及自动化架构师、测试经理、项目经理、分析师、QA专家和技术总监都可以从本书受益。

本书主要内容：

- 敏捷开发中的测试自动化
- 管理层的支持力度是如何导致自动化成功和失败的
- 设计良好的测试件结构和抽象层的重要性
- 如何衡量收益和ROI
- 管理的因素，包括技能、项目规划、项目领域和项目期望
- 基于模型测试（MBT）、猴子测试和探索式测试的自动化
- 标准、沟通、文档和灵活性在企业级测试自动化中的重要性
- 测试支持性活动的自动化
- 明智的选择：对哪些测试进行自动化、不要对哪些测试进行自动化
- 自动化测试的隐藏开销：对其进行的维护和错误分析
- 测试自动化的正确目标：为什么“为了发现bug”并不是一个好的目标
- 重点强调一些学到的教训、真知灼见和有用的小技巧

PEARSON
www.pearson.com

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604

读者信箱：hzjsj@hzbook.com
华章网站：www.hzbook.com
网上购书：www.china-pub.com

上架指导：计算机/软件工程/测试
ISBN 978-7-111-41676-0
9 787111 416760 >
定价：89.00元