

最佳实践

微软的软件测试之道

How we test software at Microsoft

阿伦·培智，肯·约翰斯顿，BJ·罗里森著
张奭，高博，欧琼，赵勇等译

献给我的妻子 Christina , 以及我们的孩子 Cole 和 Winona。为了能让我专心致志写作, 他们牺牲了和我在一起的家庭时间。献给我的父母 Don 和 Arlene, 他们为我提供了写作的地点, 还自始至终给予了我各方面的支持。

——阿伦·培智 (Alan Page)

感谢我的孩子 David 和 Grace, 给了我写作的时间。还要感谢我的妻子 Karen, 当我还在准备测试研讨会的讲稿时, 正是 Karen 建议把我的讲稿取名为“微软的软件测试”。要不是 Karen 的这句话, 以及后来阿伦的领导, 我们根本不会开始写这本书, 更谈不上完成这个写书的项目。

——肯·约翰斯顿(Ken Johnston)

感谢我的父母亲在我写书过程中给予我的爱、智慧和包容。特别感谢我六岁的女儿 Elizabeth, 她那种对新事物的持续不断的好奇心, 以及克服困难和挑战的坚忍不拔的决心, 使我认识到: 只要设法找到解决办法, 就没有克服不了的困难。

——BJ·罗里森 (BJ·Rollison)

此书献给微软全体测试工程师。他们不畏软件开发过程中的各种挑战, 勇于打破传统测试过程的条条框框, 为我们用户出厂了技术领先的、高质量的、成熟的软件产品。能为微软这么多的专业测试人员提供一本参考书, 我们很感荣幸。从我们指导他们的测试实践, 加上与他们并肩作战的交流接触中, 我们也从他们那里学到了更多的软件测试知识。

全书目录

全书目录.....	3
致谢.....	10
序	11
译者介绍.....	15
第 1 章 微软的软件工程.....	22
微软的愿景和价值观，为何我们“爱微软”	22
微软是大型的软件工程公司.....	25
拓展大型且高效的业务	26
共享团队模式.....	27
在“大”公司中做“小”项目.....	29
聘用多种类型的工程师.....	31
工程类职种的划分.....	32
作为一家全球化的软件开发公司	34
本章小结.....	37
第 2 章 微软的软件测试工程师.....	38
职位名称的含义.....	39
微软测试工程师的职称并非一直都是 SDET.....	40
我需要更多的测试工程师，立刻就要！	42
校园招聘.....	43
业界招聘.....	45
学习怎样成为一名微软的 SDET.....	45
微软工程师的职业发展.....	46
测试职种的发展道路.....	47
测试架构师.....	47
测试独立贡献者	48
成为管理人员并不意味着升职.....	49
测试管理人员.....	51
本章小结.....	51
第 3 章 工程生命周期.....	52
微软的软件工程.....	52
传统的软件工程模型.....	53
里程碑模式.....	55
敏捷开发在微软.....	57
宏观视野.....	59
流程改进.....	60
微软的正规流程改进系统.....	61
从“作战室”发布软件.....	62
强制性惯例.....	63
本章小结：把饭做完.....	64
第二部分 关于测试.....	65

第 4 章 软件测试用例设计的实用方法.....	65
实践良好的软件设计和测试设计.....	65
使用测试模式.....	66
估计测试时间.....	67
从测试开始.....	68
问问题.....	68
要有好的测试策略.....	69
思考可测试性.....	70
测试设计规格说明.....	71
用好的和坏的数据测试.....	71
在测试用例设计中要考虑的其他因素.....	72
黑盒测试、白盒测试和灰盒测试.....	72
探索性测试在微软.....	73
本章小结.....	74
第 5 章 功能测试相关技术.....	75
功能测试的需求.....	75
等价类划分.....	79
变量分解.....	80
等价类划分实战.....	82
参数子集分析.....	84
等价类划分测试.....	86
等价类划分小结.....	88
边界值分析.....	89
边界测试的定义.....	90
边界值分析：一个全新的公式.....	91
隐性边界.....	94
边界值分析小结.....	96
组合分析.....	96
组合测试的途径.....	97
组合分析实践.....	99
组合测试的有效性.....	105
组合测试小结.....	106
本章小结.....	107
第 6 章 结构测试技术.....	108
块测试（Block Testing）.....	110
块测试小结.....	118
决策测试（Decision Testing）.....	118
决策测试小结.....	120
条件测试（Condition Testing）.....	121
条件测试小结.....	124
基础路径测试（Basis Path Testing）.....	124
基础路径测试小结.....	133
本章小结.....	133
第 7 章 用代码复杂度分析风险.....	134

风险行业.....	134
复杂问题.....	134
计算代码行数.....	136
回路复杂度.....	136
Halstead 度量.....	139
面向对象的度量.....	140
回路复杂度高并不表示缺陷累累.....	141
如何利用复杂度度量.....	142
本章小结.....	144
第 8 章 基于模型的测试.....	145
建模的基本要点.....	146
采用模型测试.....	147
设计模型.....	147
模型化软件.....	148
建立有限状态模型.....	150
模型自动化.....	151
不带测试的建模.....	156
贝叶斯图解模型（Bayesian Graphical Modeling）.....	157
Petri 网.....	157
微软的基于模型的测试工具.....	158
Spec Explorer.....	158
语言和引擎.....	164
建模提示.....	167
本章小结.....	168
推荐资料和工具.....	168
第三部分 测试工具和系统.....	169
第 9 章 缺陷和测试用例管理.....	169
缺陷工作流程.....	169
缺陷的跟踪.....	170
一个缺陷的生命周期.....	171
缺陷跟踪系统的特征.....	172
为什么撰写缺陷报告.....	173
解剖缺陷报告.....	174
缺陷会审.....	177
缺陷报告中常见的错误.....	179
数据使用.....	181
何时不能使用缺陷数据：缺陷作为绩效度量.....	184
缺陷门槛（缺陷 Bars）.....	185
测试用例管理.....	188
什么是测试用例？.....	188
测试用例的价值.....	190
剖析测试用例.....	190
测试用例误区.....	191
管理测试用例.....	192

测试用例和测试点：计算测试用例.....	193
追踪和解释测试结果.....	194
本章小结.....	195
第 10 章 测试自动化.....	196
自动化的价值.....	196
自动化还是不自动化，就是这个问题.....	196
用户界面自动化.....	199
自动化测试中包括什么？.....	203
微软中的“SEARCH”.....	206
设置.....	206
执行.....	208
分析.....	213
报告.....	215
清理.....	215
帮助.....	216
自动化测试，跑起来！.....	216
把一切装配起来.....	217
大型自动化测试系统.....	217
测试自动化中的常见错误.....	218
本章小结.....	218
第 11 章 非功能测试.....	220
功能之外.....	220
属性测试.....	221
性能测试.....	222
怎样测量性能.....	222
压力测试.....	224
分布式压力测试.....	226
分布式压力架构.....	226
多客户端压力测试的特点.....	228
兼容性测试.....	228
应用软件库.....	229
应用程序检验器.....	230
吃我们自己的“狗食”.....	231
可达性测试.....	232
可达性角色.....	234
可达性测试.....	234
微软 Active Accessibility（MSAA）测试工具.....	235
可用性测试.....	235
安全测试.....	236
威胁建模.....	237
模糊测试.....	237
本章小结.....	238
第 12 章 其他工具.....	239
代码改动量.....	239

一切尽在掌握.....	240
追踪变更.....	240
什么改变了？	241
为何改变？	244
集中型的源代码管理控制.....	245
软件构建.....	246
每日构建.....	247
静态分析.....	252
本机代码分析.....	253
托管代码分析.....	254
工具只是工具.....	256
测试代码分析.....	256
测试代码也属于产品代码.....	257
更多工具.....	258
解决特定问题的工具.....	258
服务大众的工具.....	258
本章小结.....	259
第 13 章 用户反馈系统.....	260
测试和质量.....	260
测试提供了信息.....	260
质量感知.....	261
用户救援.....	262
游戏也不例外!	265
Windows 错误报告（WER）.....	266
WER 的工作原理	267
填入存储桶（Buckets）	269
清空存储桶.....	269
测试和 WER	271
微笑，微软和你一同微笑.....	271
发送微笑的影响.....	273
连接用户.....	273
本章小结.....	276
第 14 章 测试“软件加服务”	277
两个部分：关于服务和测试技术.....	277
第一部分：关于服务.....	278
微软的服务战略.....	278
业务重心向 Internet 服务迁移.....	278
从大规模成长为超大规模.....	279
能源是成长的瓶颈.....	281
服务与盒装软件.....	282
从单机软件到多层次服务	285
第二部分：测试软件加服务	286
创新的浪潮.....	286
设计合适的软件加服务测试方法.....	287

软件加服务测试技术.....	292
另一些关于软件加服务的重要思想.....	307
持续质量提高计划.....	307
我见过的被忽略的常见缺陷.....	311
本章小结.....	312
第四部分 关于未来.....	313
第 15 章 今天解决明天的问题.....	313
自动失败分析(Automatic Failure Analysis).....	313
克服分析瘫痪.....	313
匹配游戏.....	314
好的日志记录实践.....	315
日志文件剖析.....	316
集成 AFA	317
机器虚拟技术.....	318
虚拟的好处.....	318
虚拟机测试场景.....	320
当测试时发生失败.....	322
不建议使用的测试场景.....	323
代码评审和检验.....	324
代码评审的类型.....	324
核查清单.....	325
其他考虑.....	325
评审的两面性.....	327
工具、工具、到处都是工具.....	328
提炼、重用、回收.....	328
问题在哪？	328
开放式的开发.....	329
本章小结.....	330
第 16 章 建设未来.....	331
前瞻性思考的需求.....	331
“通过追本溯源来进行前瞻性思考”	331
努力培养质量文化.....	332
测试和质量保证.....	332
质量该谁管？	333
质量成本.....	334
测试的新角色.....	334
测试领域的领导力.....	335
微软测试领导团队.....	335
测试领导团队主席.....	336
测试领导力在行动.....	337
测试架构师团队.....	337
卓越测试.....	339
帮助.....	340
沟通.....	341

关注未来.....	343
微软公司卓越测试主任.....	343
三方面的领导.....	344
为未来创新.....	344
索引.....	346

致谢

如果没有每一位微软测试人员的帮助，此书根本不会诞生。他们有些是通过参与审阅、还有些是通过写出他们在微软测试的经验，而直接帮助了本书的写作。更多测试人员的帮助，则是通过他们在微软从事的测试成为了历史，或者在怎样进行测试方面的不断创新来实现的。

当然，要把微软 9000 余名测试人员的名字一一列出是不现实的，(尤其是这里面包括了对完成此书有贡献的微软其它部门的同仁、前微软员工，以及微软以外的审阅者)。另一方面，我们还是想要在此具名致谢三十几位对此书出版有突出贡献的人。

此书是在对许多微软现任和前微软员工的意见、建议和反馈，集思广益的基础上形成的。对此书作出杰出贡献的人员包括：Michael Corning、Ed Triou、Amol Kher、Scott Wadsworth、Geoff Staneff、Dan Travison、Brain Rogers、John Lambert、Sanjeev Verma、Shawn Mcfarland、Grant George、Tara Roth、Karen Carter-Schwendler、Jean Hartman、James Whittaker、Irada Sadykhova、Alex Kim、Darrin Hatakeda、Marty Riley、Verkat Narayanan、Karen Johnston、Jim Pierson、Ibrahim El Far、Carl Tostevin、Nachi Nagappan、Keith Stobie、Mark Davis、Mike Blaylock、Wayne Roseberry、Carole Cancler、Andy Tischaefer、Lori Ada-Kilty、Matt Heusser、Jeff Raikes、微软研究院（特别是 Amy Stevenson）、微软卓越工程（Engineering Excellence）测试团队、微软测试领导团队(Test Leadership Team)、和微软测试构架师团队（Test Architect Group）。

我们还要感谢此书的项目编辑 Lynn Finnel 在著书的过程中所给予我们的持续的鼓励和支持。

序

我记得在 2007 年秋天的一个早晨，我当时的经理肯·约翰斯顿(Ken Johnston)冷不丁地说出这五个字：“你应该写书”。

当时他刚从某行业测试会议作了个演讲(巧合的是他的题目就是“我们怎样在微软测试软件”)回来，还在为得到听众的一致肯定而兴奋。肯很爱演讲，但他突发奇想地认为我应该是那个写书的人。

我打趣地回复他说，“行，为什么不呢”。我接下去说，这书可以包括很多我们在微软讲授软件测试课程的内容，以及在微软流行的其他测试实践和方法。它可能很有趣，但我知道已经有很多很多测试的书籍，我至少看过几十本。其中有些写得确实很好。再写一本能对软件测试界提供什么价值呢？

我原本打算否定肯的写书想法，但我突然意识到了很关键的现实：在微软，我们具有很多世界上最先进的软件测试培训。培训的课程内容和结构都无懈可击，但这些并不是最绝妙的地方。我们的教员在讲课中穿插的轶事、成功故事，以及在测试过程中的一些小琐事、小花絮才是对听众影响最大和印象最深的。如果我们能把这些故事、插曲，以及微软怎样在实践中应用等内容写在书里，那该书可能会是很有趣的。我开始超越了我们的教学去思考，去考虑更多的能和世界各地的测试同行分享的测试想法和成功故事。我还意识到我喜爱的一些编程的书都把所有纯技术的讲解，穿插了有趣的故事。

下一步我知道要做的是写一个提案。书的框架和大纲逐步成形了。主要体现在四个主题。首先通过谈论微软一般员工和工程实践来引入上下文是有道理的，第二部分和第三部分将集中谈谈我们在微软怎样做测试和我们使用的工具，最后的部分将探讨微软未来的测试。我把该提案送到了微软出版社(Microsoft Press)。虽然我对于本书的潜力一直很兴奋，但我也希望微软出版社将告诉我说：这想法挺傻的，我应该趁早放弃。可是，那个结果并没有发生，而且紧接着，我发现自己凝视着计算机屏幕，开始思考这本书第一个句子会是什么样。

从最开始，我就知道我想让肯写前两个章节。肯是在微软任职多年的一位经理，写关于在微软‘人(people)’的方面是他拿手的。就在我递交写书提案的那段时间里，肯离开了我们部门去管理微软 Office 的一个在线(Online)产品部门。很快地，事情变得明朗了：肯应该也写本书关于我们怎样测试“软件+服务”的章节。他从此成为在公司领导着决定微软怎么测试 Web 服务的关键人物。让他写第 14 章“测试软件+服务”是最合适的。稍后，我又找了我的同事 BJ·罗里森 (BJ·Rollison)，他是微软公司中一个最有资历的测试人员之一，让他写本书中关于功能和结构测试技术的章节。BJ 是我们部门设计微软核心软件测试课程的主要负责人。他比我认识的了解这些测试方面知识的其他任何人，都了解得更多。他也是我知道的唯一一位，比我读了更多测试书的人。于是肯、BJ 和我三人成了本书的作者团队。我们完成任务和写出资料都有相当不同的风格，但在最后，我们不同的写作风格和材料使我们感到，我们正好反映出了微软多元化的测试人群。我们经常开玩笑说，BJ 是教授，肯设法成为史学家和讲故事者，而我却是吸收信息，并且陈述事实。虽然我们每人都分别负责几章的写作，但我们每人都对另外两人的章节有所贡献，所以，本书融合了我们三位作者的风格。

当“写书”的任务总担在你的肩上时，我不可能描述出每个生活中小挫折怎么会变得硕大。从编写这本书开始，我接手了肯的老工作：担任微软测试卓越部门的总监（Director of Test Excellence）。我自己都不知道为什么我会决定在写书过程中，还承担这样一个全新的、极具挑战性的工作。然而，事实上，承担的这个角色，迫使我深入了解一些微软测试领导的内幕，因而极大地有益于这本书的写作。

在写这本书时，我最大的担心是，我知道有很多可写的内容，但我必须要省略。微软有 9000 多个测试人员。这本书里谈到了微软测试人员做的大多数事情，但也有很多很“酷”的在微软的测试人员做的事，不可能全包括在这本书里。此外，这本书谈到的几乎每个主题都可能有大大小小的变异。当讲关于我们认为是最重要的有关测试的故事时，我们尽量设法捕捉许多不同的想法。我也要承认对于这本书的标题，我有点紧张。“我们在微软怎么测试软件”表面意义可能蕴含着在这本书写的一切，都是每一位微软测试人员做到的，那可是不切实的。有着这么多测试人员和这么多种多样软件产品的微软公司，根本不可能确切地写出能代表每一位微软测试人员的测试方法和实践。所以，我们不得不妥协。这本书只包括微软多数测试人员使用的，普遍的测试实践、工具和技术。并非每个团队都实际做了我们书里写的一切，但多数还是适用的。我们选择写在这本书里的，都是成功地用在测试微软产品实践的，因此这本书的内容是我们所知道的、微软实践中可行的信息汇集。

最后，我认为我们成功了，但是作为测试人员，我们知道我们可以做得更好。可惜的是，已到发布的时间了，不过我们已有产品发布后的支持计划！如果您对与这本书作者谈论的任何主题感兴趣，欢迎您参观我们的网站：www.hwtsam.com。我们乐于听取您的想法。

——阿伦·培智 (Alan Page)

这本书是为谁而写

这本书是为所有对微软测试角色感兴趣的、或为那些想知道更多关于微软怎样搞测试的人写的。这本书不是来替代任何其他许多关于软件测试的写作。相反，它是来描述微软怎么运用很多的已知测试技术和方法改进我们的软件产品及质量。

微软的测试人员可能会对本书很感兴趣，因为它描述了微软公司的实用方法和技术。想知道关于微软测试的非测试人员，也会对本书感兴趣。

这本书是关于什么？

这本书是以使读者熟悉微软产品、微软工程师、微软测试人员、测试的作用和对软件工程的通常做法作为开始。书的第二部分讨论许多在微软常用的测试实践和工具。书的第三部分探讨某些我们工作中使用过的工具和系统。书的最后一部分探讨在微软测试和质量的未来方向，以及我们打算怎么创造未来。

第一部分，“关于微软”

第 1 章，“微软的软件工程”

第 2 章，“微软的软件测试工程师”

第 3 章，“工程生命周期”

第二部分，“关于测试”

第 4 章，“软件测试用例设计的实用方法”

第 5 章，“功能测试相关技术”

第 6 章，“结构测试技术”

第 7 章，“用代码复杂度分析风险”

第 8 章，“基于模型的测试”

第三部分，“测试工具和系统”

第 9 章，“缺陷和测试用例管理”

第 10 章，“测试自动化”

第 11 章，“非功能测试”

第 12 章，“其他工具”

第 13 章，“用户反馈系统”

第 14 章，“测试软件加服务”

第四部分，“关于未来”

第 15 章，“今天解决明天的问题”

第 16 章，“建设未来”

网上的更多内容

当这本书有补充作用的新增或更新材料出现时，我们会张贴在微软新闻在线软件开发者和工具的网站上。这类材料也许包括更新的内容、文章、内容的链接，勘误表、样章和其他信息。这个网站是在 www.microsoft.com/learning/books/ 上，并会周期性地更新。

更多微软测试的故事和花絮将张贴在 www.hwtsam.com。

本书提供的支持

如果您有关于这本书的评论、疑问或者想法，却在查询网站过程中得不到答案，那么请通过电子邮件联系微软出版社：

mspinput@microsoft.com.

或邮寄到：

Microsoft Press
Attn: How We Test Software at Microsoft Editor
One Microsoft Way
Redmond, WA 98052-6399.

请注意以上提供的地址并非微软提供软件产品支持的联系地址。

译者介绍



张 艳

张艳是总部在美国的 I-Connect 公司卓越工程与管理总经理。曾是微软经验丰富的测试主管。有 11 年微软软件测试、发布、项目管理和团队管理经验。曾任微软卓越管理领导团队的成员、微软华协理事会成员、微软第一届亚太地区员工领导和发展大会职业发展委员会负责人、西雅图地区 IEEE 理事会成员。

高 博



欧 琼

欧琼是微软总部高级软件开发测试主管。她有十余年丰富的软件开发测试、项目管理和团队管理经验，参与过多版本的 Windows, Office, Visual Studio 和 Windows Live 的开发测试管理工作。她是微软总部有 2500 多名会员的华人协会的理事。她拥有中南大学学士学位，日本神户大学硕士学位并进修博士课程。

赵 勇



赵勇在微软从事大型广告平台的开发。此前在 IBM 和美国 Argonne 国家实验室担任实习研究员。赵勇持有美国芝加哥大学的计算机博士学位，清华大学工学硕士以及北京师范大学理学学士学位。是 ACM 和 IEEE 会员。



鲍 臣礼 鲍臣礼，高级开发测试工程师，曾在微软总部 Windows 核心开发部从事 Windows 7 的开发测试工作，主持设计开发自动化测试框架。

参与过 Hotmail 等多项 Windows Live 产品的开发测试工作。实践中，尝试用中国传统理论如中医理论来思考软件测试问题。毕业于中国郑州大学，获得计算机科学学士学位。



陈 思清 陈思清现任微软总部高级软件开发测试主管。2000 年从伊利诺伊大学厄巴纳—香槟（University of Illinois at Urbana-Champaign）计算机系和计算语言学系毕业后通过校园招聘就职微软，具有多年丰富的软件开发测试，项目管理和团队管理经验，曾两次获得微软员工杰出贡献奖（Gold Star Award）。



邓 安桐 邓安桐是微软 SQL Server 中国研发中心资深软件开发测试主管。她有丰富的软件开发测试，本地化和项目团队管理经验，参与过多个版本的 Office, SQL Server 的本地化和开发测试工作，积累了上百软件外包项目的管理经验。邓安桐拥有北京理工大学计算机图形学硕士学位，2006 PMP。



杜 彬 毕业于浙江大学计算机学院，毕业后加入了微软服务器与开发工具事业部，参与了多个 Visual Studio 版本的开发测试工作。



方 敏 方敏现任微软 Windows 国际软件工程首席测试部门主管，有着近 20 年软件测试管理和开发的丰富经验，曾参加过微软多项重大产品和技术的研制，包括 Windows Server/Client/Security, SQL Server, Exchange Server, MSN, COM+ Services, Windows Media, 以及微软内部 IT 工具等。方敏曾在清华大学获得电子工程学士和硕士学位，在美国新墨西哥技术学院获得计算机硕士学位。



冯 志强 冯志强现任微软 Office 服务器产品的开发工程师。此前在微软 SQL Server 部门从事数据库产品研发工作。曾在 Computer Associates 和 Siebel Systems 等多家公司任职。拥有 17 年软件设计、开发、项目管理和测试经验。冯志强毕业于清华大学物理系，拥有物理学硕士学位。



顾 广宇 顾广宇有 8 年微软软件测试和团队管理经验，先后在 Windows Security, Connected Systems 等部门担任软件开发测试工程师和测试主管。他是微软“Community Leadership Excellence”获奖者之一，曾连续三届担任微软华协理事会理事，负责“职业发展”，通过举办关于软件开发、测试和项目管理的讲座和培训帮助华协成员提升业务水平，分享在微软获得成功的心得。他拥有北京清华大学电子工程学士和美国西北大学计算机工程硕士学位。



贾 近 现为微软总部软件开发测试工程师。毕业于美国俄勒冈州立大学计算机专业，此后加入微软 Windows 部门，参与开发测试 Windows Vista 中的 Desktop Window Manager 和 AERO 界面。目前在 Windows 图形组从事桌面图形子系统的开发测试工作。



蒋 晓华 蒋晓华从 06 年至今任职微软总部 SQL 团队核心引擎部门的软件开发测试工程师，曾参与了数据压缩，文件流等多个 SQL Server2008 主要功能的开发测试。她拥有上海交通大学的计算机学士学位和美国弗吉尼亚大学的计算机硕士学位。



梁 梅 梁梅现任微软社区高级项目经理，负责全球微软最有价值专家和社区论坛项目。她拥有十年丰富的微软软件技术支持，开发测试，项目和团队管理经验。曾参与多版本的 Office, Exchange 和 Visual Studio 的技术支持和开发测试工作。她毕业于南京大学，并获得德克萨斯州立大学硕士学位，赴美前曾活跃于中国金融，建筑经济行业。



林 俊彦 毕业于复旦大学计算机专业，此后加入微软中国研发集团服务器与开发工具事业部，现为 Visual Studio 软件开发测试工程师，曾参与 Visual Studio 多个版本的开发测试工作，业余时间热心于专业资料的翻译工作。



刘 俐 现为微软雷德蒙总部测试软件开发工程师，拥有五年自动化软件测试经验。曾参与微软在线广告中心(AdCenter)的测试和发布工作。现从事微软 Office Communicator 软件开发测试工作。刘俐拥有北京理工大学硕士学位和北京工业大学学士学位。



吕 灵 吕灵于 2007 年加入微软 Windows 部门，现为微软总部软件开发测试工程师。曾参与测试 Windows Vista SP1 和 Windows Server 2008，目前在 Windows 查找和组织（Find and Organize）组从事 Windows 7 Libraries 和 Homegroup 的开发测试工作，并负责 Accessibility 测试。拥有电子科技大学通信与信息系统工学硕士学位和美国伊利诺伊理工学院计算机工程硕士学位。



孙 展波 1999 年取得美国弗吉尼亚理工学院及州立大学计算机硕士学位。毕业至今就职于微软。先后在 Windows Media Services, Windows Presentation Foundation 和 WPF/Silverlight Designer 产品组工作。他的中文博客在 <http://blog.joycode.com/zhanbos/>。



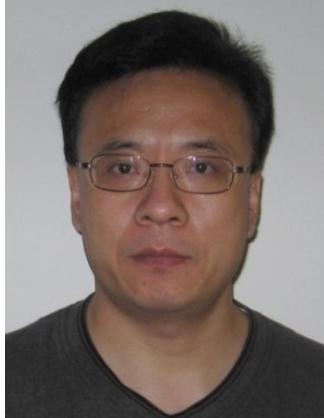
王 文婧 王文婧曾在 Office 部门的 Windows Sharepoint Services，现在 Windows Live 担任软件开发测试工程师。她毕业于加拿大滑铁卢大学电脑工程系，并即将获得华盛顿大学 MBA 学位。她曾获得加拿大多伦多中文学术比赛翻译组亚军，并拥有加拿大多伦多大学高级专业翻译证书。她也曾是微软华协理事会理事和微软华协电子报纸总编。



于 国军 山东烟台人，现任微软总部资深软件测试开发工程师，曾获微软员工杰出贡献奖。于国军一直致力于微软办公系统产品（Office）及共享服务平台（SharePoint）的测试和测试自动化，拥有清华大学机械设计与制造专业学士及硕士学位，美国明尼苏达大学计算机硕士学位。



张 胜 张胜是微软总部高级软件开发测试主管，10 余年软件开发测试和团队管理经验，参与 Visual Studio, SQL Server, Office Live 的开发测试和发布，现主管 Office Communications Server 本地化软件开发测试工作。张胜拥有复旦大学计算机系硕士和学士学位。



张 帷 张帜在微软 Flexgo 团队从事软件测试以及海外测试成果检查及协调工作。曾经参加 BizTalk Server 2009, Quadrant 等产品的测试工作。在加入微软前，张帜曾经在 ORACLE 及 TEK Systems 公司从事软件测试工作。张帜持有 Concordia 大学的计算机学士学位，大连理工大学工学硕士学位，以及武汉大学工学学士学位。



郑 洪流 郑洪流是微软总部测试工程师。1998 年加入微软后一直在测试第一线工作至今。参与测试过的主要项目包括 MCIS(ISP 管理软件)；微机健康(PC Health)；系统还原(System Restore)；远程帮助(Remote Assistance)；符号服务器(Symbol Server)；视窗安全模式(Safe Mode)；系统重启监视系统(SET)；视窗恢复系统(Windows Recovery Environment)等等。



钟 颂东 钟颂东 2006 年至今任微软总部 SQL 团队引擎部的开发测试工程师，负责包括 SQL2005, SQL2008 及开发中的 SQL 服务器的测试任务。曾在 NACT/Verso 等公司任职。拥有 10 多年软件设计、开发、网络及数据库管理和测试经验。钟颂东毕业于中国科技大学，持有物理学士学位，曾在杨百翰大学进修过计算机硕士及物理博士课程。



周 仲哲 周仲哲, 现为微软总部软件开发测试工程师。曾就职于加拿大及美国公司从事软件开发并担任技术主管, 2005 年加入微软总部, 拥有十年软件开发及自动化软件测试经验。曾参与 Windows Vista 及 office Communicator 的开发。后致力大型自动化软件框架系统及相关工具的设计和开发, 拥有丰富的自动化软件设计和开发经验。目前在 Office 从事自动化软件本地化和软件外包项目团队管理



朱 健俊 朱健俊是微软总部高级软件开发测试经理。他有 9 年微软工作经验, 参与过多种版本的 Windows, Office, Visual Studio 和 Windows Live 的开发测试工作。他拥有同济大学学士学位, 德州农工硕士学位。



邱 扬 邱扬拥有 5 年的软件开发测试经验。他于 2006 年加入微软, 参与了 Expression Studio 的两个版本以及 Visual Studio 的测试和发布。他拥有东华大学的本科和硕士学位。



李 晶 毕业于哈尔滨工业大学软件工程专业, 此后加入微软中国研发集团服务器与开发工具事业部, 现为 Visual Studio 软件开发测试工程师。

第一部分 关于微软

第 1 章 微软的软件工程

肯·约翰斯顿

本书的第一部分，也就是第一章到第三章，将介绍微软的基本情况，包括微软的目标、微软是如何组织软件产品开发，以及微软是如何推出产品的。本章的大多数内容都广为公众所知，不过，我们也提供了一些微软的内部信息。

很多书籍、文章、网站都已详尽地介绍了微软的历史，本书将着重于介绍微软的软件工程，这将有助于读者更深入地了解微软的软件测试技术和方法。我们介绍的微软的历史、愿景、以及企业目标，对于我们在以后章节里讨论的所有软件测试方法和工具是有重要影响的。

微软的愿景和价值观，为何我们“爱微软”

除了足球世界杯或者板球赛事，让理性的工程师表现得热情奔放是比较难的。微软的工程师也是如此。不过，有一件事可能是例外。

在每年的十月初前后，位于美国华盛顿州雷德蒙德(Redmond)市的微软总部，都召开全体员工会议。届时，将近有两万名员工乘坐几十辆大客车前往 Mariners 棒球队的主场 Safeco 体育馆出席会议，此外，还有数万名员工在线参加该年会。

年会上，最后一个演讲者总是我们的执行总裁斯蒂夫·巴尔默(Steve Ballmer)。在雷鸣般的欢呼声和响亮的摇滚音乐声（比如 Rocky 电影里的“虎眼”的音乐）中，斯蒂夫快速跑上主席台，他热情奔放，陈词激昂，把与会者的情绪推向了高潮。然后，他跑下主席台，到观众席和大家互动。当他穿过观众席中的人群时，他不停做出象征胜利的手势，时不时和大家击掌相庆，收到很多员工热情的回应。他在途中会多次停留，继续激发大家的热烈情绪。回到主席台后，平复一下呼吸，他会带领大家一起齐声呼喊：“我爱微软！”你可以很容易地在 www.live.com 网站找到这类录像，你只要搜索“Steve Ballmer 公司会议”就可以了。

不错，微软是个很酷的公司，在本书以下各个章节里，大家会看到，作为软件测试工程师，我们所工作的公司是世界上最棒的。我们丰富的历史、出色的产品、和与众不同的传统等（例如公司全体员工会议和大声呼喊鼓舞员工热情的执行总裁），使得微软成为工程师们的理想工作单位之一。

在微软早期的 24 年里，公司的愿景就是“每家每户的每张桌面上都有一台个人电脑 (Personal Computer，常简称作“PC”)。”从许多方面衡量，微软已经达到了这个目标。当然，在互联网时代，公众对诸如 XBox 一类的设备的需求、以及对服务类软件的关注，使得微软对愿景也作了相应的调整。从 1999 年到 2002 年，微软的愿景是：“杰出的软件无论在何时何地运用于何种装置，都能赋予人们巨大的力量。”比起仅仅着眼于 PC 的愿景，这个愿景显然更加远大，更加大胆。在 2002 年，微软又提出了更远大的新使命“帮助全球的个人用户和企业展现他们所有的潜力。”我们在图 1-1 中，可以看见 2008 年微软的徽标，这个标志反映了微软的新愿景。



图 1-1 微软 2008 徽标

2008年9月，在公司全体员工大会上，斯蒂夫·巴尔默宣布了公司的新使命：“把软件的神奇和网络的力量集合起来，在全世界的设备上创造崭新体验 (Create experiences that combine the magic of software with the power of Internet services across a world of devices)。”当时，会场的屏幕上还显示了个人电脑图像、服务器、浏览器应用程序、以及从手机到游戏机等设备。这个使命的真正含义就是把软件加服务应用于每一个功能强大的互联网连接设备上。这个新使命，加上2002年提出的愿景，使微软既有了使命也有了愿景。在推出新使命的时候，斯蒂夫一度停下来，回答了一个很多人关心的问题：我们如何对待公司旧的愿景：“您的潜力就是我们的动力”？斯蒂夫认为，多年来公司一直以“每家每户的每张桌面上都有一台电脑(PC)”作为愿景和使命，今后恐怕不会再有这样的单一明确的目标来指导公司的发展了，这一转变也标志着我们的成功和对世界的巨大影响。虽然我个人喜欢着重于帮助人们实现自己的潜能的愿景，但是作为一名软件工程师，我更喜欢软件、互联网和设备这些具体的东西，作为一名软件测试工程师，新的使命是对我的巨大挑战，因为这意味着我的测试矩阵又要再一次的扩大了。

为了实现这个使命，微软公司认同6个价值观：正直诚实；直率和相互尊重；勇于迎接巨大的挑战；充满热情；值得信赖；和坚持自我提高和完善¹(Integrity and Honesty, Open and Respectful, Big Challenges, Passion, Accountable, and Self-Critical)。其中，勇于迎接巨大挑战是软件工程师最常用的价值观。当软件工程师在谈论下一个重大项目的时候，以及为什么现在就要开始这一项目的时候，就会体现出勇于迎接巨大挑战的价值观。巨大的挑战还包括机智地冒险；坚持但不僵化；勇敢但不鲁莽。

维护我们的使命和勇于迎接巨大的挑战是微软的工程师前进的动力。这些价值观鞭策着微软的软件测试工程师努力不懈地找出下一个缺陷、再下一个缺陷，以便我们的顾客能用上高质量的产品。

当被问及Stephen Elop为什么要加盟微软，取代即将卸任的Jeff Raikes的企业部门(Business Division)主席职位，即掌管整个企业部时，Stephen谈到了使他下决心的几个原因，但他最后归结到一句很能引起听众共鸣的话：影响力。“加入微软公司提供了一个机会，一个对全世界数百万计人们的生活和工作有积极影响的机会。比方说在飞机场，如果一个人走过来对你说：‘请看我能用你们的软件完成的事。’那该是多么有成就感。”

用软件改变世界，需要出色的软件产品和创造这些软件的优秀团队。众所周知，微软经常重新整编工程队伍，但是到我们写这本书时，最上层的三大产品工程部门还保持着很多年一直没变的结构：

- 平台产品以及服务本部 (Platform Products and Services Division, 简称PSD)：包括客户服务分部(Client Group)，服务器及工具分部(Server& Tools Group)，以及在线服

¹ 微软公司价值观定义可参考：<http://www.microsoft.com/china/CRD/corpvalue.mspx>

务分部(Online Services Group)。

- 企业本部(Business Division, MBD)(见小提示): 包括信息工作者分部(Information Worker Group), 微软商务解决方案分部(Microsoft Business Solutions Group), 以及统一通信分部(Unified Communications Group)
- 娱乐及设备事业本部(Entertainment and Devices Division, E&D): 包括家庭娱乐分部(Home & Entertainment Group)和移动与嵌入设备分部(Mobile & Embedded Devices Group)

提示: 在微软内部, 企业本部(Business Division)的缩写本应是BD而不是MBD, 但使用三字母缩写是微软根深蒂固的传统。所以前面加上微软(Microsoft), 就成了“Microsoft Business Division”, 于是MBD就成了企业本部的英文缩写。

微软所有的软件开发都由这三个大部门负责。每个本部由一个部门主席负责, 直接向执行总裁斯蒂夫·巴尔默汇报, 当然, 斯蒂夫还有很多其他直接下属。在本书出版的时候, 比尔·盖茨(Bill Gates)应该已经退出了微软的全职工作, 但是, 在未来许多年, 他还是会出现在微软组织示意图中的。图1-2是简化版微软组织示意图。



图1-2 三个软件开发本部在微软组织机构中的位置示意图

每个本部都为公司贡献几十亿美元利润和100到200亿美元的营业收入。从营业收入来看, 这些产品本部比很多全世界500强公司还要大。每一个本部都负责几十个产品, 其中很多产品都有很高的销售额和很高的利润, 而其它的一些产品则是微软的投资, 也许在许多年内尚无回报。

划分各个本部的原则是每个本部致力于不同的市场划分和不同的客户群: E&D专攻娱乐, MBD做企业软件, 而PSD则致力于建立的平台, 我们的生意伙伴和微软的其他部门用这些平台来为其他公司和顾客提供解决方案。

我们的首席软件架构师Ray Ozzie是斯蒂夫·巴尔默的直接下属。许多其他高级领导人也是斯蒂夫的直接下属, 如首席运营官(COO)Kevin Turner, 法律和公司事务团队(LCA)的高级副总裁Brad Smith。虽然在公司的各个部门, 甚至在人力资源部门, 都有一些软件工程师, 但是, 绝大多数的软件开发工程师和软件测试工程师都分布在这三个部门。

是订购新 T 恤衫的时候了

(译注：微软的文化之一，是不同的项目团队都有自己独特的文化衫，上面印着该项目的标识等图案。大家很喜欢穿着项目文化衫上班。这段文字里，换文化衫意味着更换项目或团队重组。团队重组是件大事，经常发生，对于每一个人来讲都意味着机会。为了不和一般的文化衫混淆，我们还是把它翻译做项目 T 恤衫。)

这是 1998 年的一个炎热而晴朗的夏日。斯蒂夫·巴尔默还没有出任微软首席执行官或总裁，他把所有 Atlas 项目工程师召集到一个大会议室。就在一天前，Atlas 项目 T 恤衫已经分发下来了，我当天就穿着那件 T 恤衫。在发 T 恤衫的时候，我应该知道我们有麻烦了。T 恤衫上印的图案是一个人把世界扛在肩膀上，其寓意是 Atlas 项目的目标：在任何时候，任何地点，任何设备上提供软件和服务。我们一小群约 200 名工程师接受了这项任务。这一设想最终成为微软企业的使命，所以我们这个小团队的资源显然是不够的。

在从办公楼走到会议室的路上，一个测试人员走到我身边，对我说：“是订购新 T 恤衫的时候了。”

我问：“你是什么意思？”

“很简单，伙计，”他转身面对着我，一边退着一边说。“我们拿到了这个项目的 T 恤衫。我们去见斯蒂夫·巴尔默。难道你不明白吗？我们的团队要重组了。”

“这是我今年拿到的第三件项目 T 恤衫了，”我说。“我们团队不会再次重组的。”

但是，我的朋友对了，我错了，我们的团队进行了重组。从那时起，每次听到“是订购新 T 恤衫的时候了”，我的感觉总是很复杂。一方面，它可能意味着，你要放下手头的一切工作，开始一个新项目，但另一方面，这可能意味着你有机会获得一个伟大项目的 T 恤衫，即一个并不多见的良机，你穿着这个项目的 T 恤衫去办公室，常常会引发别人的兴趣，“噢，我记得这个项目...”，由此而引发兴奋的交谈。我呀，就有两件这样的项目 T 恤衫。

有一次，我们为全部门的测试工程师定做了 T 恤衫，印上了 Doonesbury 的漫画。如果你想看的话，该漫画条发表于 1996 年 3 月 19 日，其中包括一个测试工程师最喜爱的短语：“找出缺陷，非常酷”。Garry Trudeau，Doonesbury 的创建者，还在其中一件 T 恤衫上签了名，我们自豪地把那件 T 恤衫在我们办公楼的大厅内展示多年。

其实，多次转换项目反而使我获得了很好的经验。在接下来的几年中，我并没有更换团队或上司，但却参与了多个服务器产品和服务类产品的测试工作，经历了几个项目的发布过程，每当想起那些发布后的庆祝会，总是感到有趣而且令人难忘。

微软是大型的软件工程公司

微软是个大公司这对任何人来说都是不足为奇的：截止 2008 年 1 月，微软全球的雇员总数超过九万人。因此，了解微软在如此大的规模下，如何完成各项工作是很重要的。

对于微软来说，“大”其实意味着“广”，而“广”是指每年投入市场的产品线之广，微软产品销售与竞争的市场种类之广，以及公司达到所有这些要求所面临的工程技术的挑战之广等等。

例如，微软提供的软件产品的种类是庞大的。在 2007 财政年度(7 月 1 日至 6 月 30 日)，微软发布了 100 多种主要产品，比如 Microsoft Office 系统、Windows 操作系统、游

戏软件和游戏机硬件、家庭娱乐产品、商务解决方案例如客户关系管理- (CRM) 软件、移动嵌入式设备(Mobile Embedded Devices)， 网络在线服务如 Live 邮件和网上搜索，以及小型商务网络服务软件。微软还开始投资于企业到企业 (business-to-business, 即 B2B) 中的服务，并继续在其他新兴的软件市场中拓展。微软继续投资在机器人软件，基于 Internet 协议的电视 (Internet Protocol-based Television, 即 IPTV) 和车载电脑(automotive PCs)等。

同样惊人的是微软产品投入的市场数目。当微软发布了一种新软件产品或服务时，我们通常会在全球统一发布。这几乎要求每种产品的内容都译成 80 多种语言和方言。表 1-1 列出的产品及市场销量的统计数据应对了解 Microsoft 产品线的广泛应用有更具体的概念。

表 1-1 微软产品的简介

产品	简要介绍
Windows 操作系统	Windows 操作系统占有桌面操作系统的超过 90% 的市场份额。截止 2008 年中(大约本书出版的时间)，装机量已超过十亿台。
Microsoft Office	Office 95 版本支持 27 种语言。Office 2007 支持的语言超过 Office 95 的 4 倍。随着全球软件市场的不断拓展，支持的语言数量仍将不断增加。
Windows Mobile	Windows Mobile 销量超过两千万，在智能手机的市场占有量首屈一指 (2007 的数据)
Xbox 360 游戏机	截止 2007 年底，全球已售超过 1 千 4 百万台
Halo 3 游戏	Halo 3 (仅限于 XBOX 360 游戏机) 游戏首日销售额达 1.7 亿美元，创游戏史上首日销售纪录。
Windows Live 邮件	世界上最大的电子邮件服务，有超过 4.25 亿的活跃用户帐户
虚拟地球	虚拟地球每天提供 6 亿多图块的浏览服务 (图块就像瓷砖，是地图一个部分，用户可以自由缩放)。

微软公司的巨大规模、多元化的产品组合，解释了为什么在微软没有一个统一的方法来制造和发布所有的产品。从其核心意义上来说，微软是一个由广大员工的创造力所推动的软件公司。即使横跨各种不同的产品团队和流程，很多测试团队仍共享许多相同的最佳实践和测试工具。这本书第一部分将重点介绍微软测试工作与人相关的一面：我们如何组织测试团队、如何面对测试公司各种软件的不同挑战。第二至四部分将具体介绍和演示我们测试的最佳实践方法和工具。

拓展大型且高效的业务

我们主要用两个模式来组织我们工程师队伍。某个产品业务从培育期到成熟期，可能会从一个模式转换到另一个模式。当 Office 刚开始的时候，它根本不是 Office 现在的样子。先从 Word 和 Excel 开始，然后增加了 PowerPoint 和 Access 等。最开始的几个版本，每个产品都各自独立开发并发布。这种偏重于产品独立发布的模式通常称为 PUM(Product Unit Manager, 即产品部门经理) 模式，这在微软内是最常见的。

在 PUM 模式下，一个产品部门单一组织及管理他们需要的(或至少可以获得的)所有工程资源。他们通常没有依赖其他部门的地方，除非另一个部门提供了一些比较成熟的技术。PUM 模型比较适合那些需要迅速发布的产品并能灵活应对竞争对手，但它不适用于资源的集中和共享，例如产品编译或测试自动化工具的生成等。在这种模式下，重复性工作和团队间交流的开销是较高的。几乎每个产品，不论其如何成熟，在其中仍有一些这样的小团队。Office 和 Windows 这两个部门在发布主要版本的同时，仍使用 PUM 模式管理其孕育新的产品功能的团队。

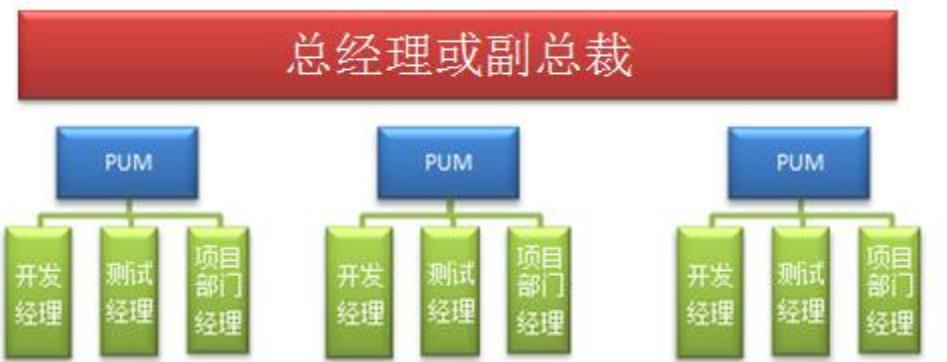


图 1-3 工程师团队内最常见的 PUM (产品部门经理) 模型

共享团队模式

当产品以及产品类别日趋成熟时，其团队通常也随之变大，并自然而然地想通过集中管理来提高效率和降低成本。这种模式有很多名称，但共享团队模式可能是最好的名称。在此模式下，产品的公共功能的开发测试被集中在一个共享团队，其它的产品团队必定依赖于这个共享团队，不然的话，他们的产品功能就难以开发成功。

Office 是一个很好的例子。自从有人想出这个聪明的点子：捆绑我们的与办公相关的应用程序软件，并把他们作为一整套系列来销售，Office 团队便开始了从 PUM (Product Unit Manager) 模式到共享团队模式的演变。在实际工作中，Office 的员工称该演变成了的部门为 Office 共享部门 (Office Shared Services，简称 OSS)。



图 1-4 典型的 Office 部门的组织结构

在 Office 组织内，有包括 Word、Excel 和 Sharepoint 在内的许多产品团队注重于特定的应

用程序和与之相关的用户群，也有许多共享团队如用户界面、编译生成和文档流程(User Interface, Build, and Document Lifecycle)等团队，注重于集中与共享技术和用户情景。产品团队倾向于关注其特定客户群(如 Excel 最关心电子表格用户)，但他们同时必须遵从共享原则(最佳的用户界面是界面的一致性，这种一致性使得用户要学用 Office 的应用软件，只要学其中一种Office 的应用软件就可以了，而且可以有助于学会其他各种Office的应用软件功能，可谓是一通百通)。通过这种实践，Office 部门在每个产品不断创新的同时仍能保持整个Office 产品系列的一致性。这些目标可能有些矛盾，但通过每个产品团队的通力合作，就能整合设计、开发和测试各方面，创造出一个 Office 部门考虑所有产品特点和共性的整体发布愿景和创新设计。

同样的事情也发生在各工程团队中，特别是测试团队。每个产品团队都有一个测试团队，其任务是用创新的方法来测试和验证产品的功能。不过拥有一套统一的创新测试工具和方法，对于产品会是大有益处的。因为认识到这点，Office 部门已探讨、寻求和创新某些方法、实践和流程等，并应用在特定的Office 产品开发过程中。如产品功能小组(Feature Crews)(第3 章会详细介绍)、Office 自动化测试系统 OASYS，以及代码提交到整个 Office 代码管理系统前(pre-checking)，用于验证代码能正常运行的工具：“大按钮”(Big Button)等。

共享模式不仅提高效率，而且助长创意。一个共享团队可能对普通用户界面有独特见解，但最初设计不一定能达到特定用户群的要求。共享基础设施质量欠缺会减慢其相关团队的工作进度，如果其质量得不到改善，共享团队会成为其他团队的绊脚石，而其开发的基础设施也将会被弃用。

在完全的 PUM 模式和完全的共享团队模式之间有很多变种。其实在微软内部，尚没有发现具有完全相同的组织结构和相同的开发产品流程的两个团队。很多人比喻微软从外面看起来像一个战舰，但您从里面看它时，微软更像一群快艇，都急速地驶向一个的共同的终点，但又各显神通。对于微软工程队伍和产品组织，我的一个朋友想到了一个更好的比喻。

象创作戏剧一样研制和发布软件产品

当我刚开始在微软工作时，我很难理解不同的产品制造流程，最佳实践和不同的工程师角色等。这些都引导我思考如何在这里做事。从一面来说，我们需要发明和创新。从另一面，我们又是一个大公司，二者会莫名其妙地相互抵触；即大公司很少能在根本层面上成功地创新。可一个比喻却帮助我理解了这个难题，我把这个环境与剧团进行类比。与任何剧团一样，我们有董事、生产商、演员(工程师)，当然还有 A 档和 B 档明星、舞台制作者和其他参与的人。这个比喻的关键是：尽管表演艺术主要在于创意和完美的表现，但也是出自剧院的产品，需要票房收入和被观众接受。对潮流和长远趋势的把握，在软件开发中与在表演艺术中同样重要，尤其是在当今以用户体验为主导的世界。

从事营销、销售和法律等也都是角色分明，各司其职。我是这样来与微软类比的，行政及财务管理是制片人，部门总经理是导演，工程师们是演员，这与剧院内戏剧制作的角色一一对应。同样，与剧院一样创新是产品的驱动力。一个突破性产品就象引领潮流的戏剧作品。一旦我把二者联系起来，我觉得像是站在我所钟爱的基洛夫芭蕾舞剧院的侧翼，看我的好朋友及邻居们翩翩起舞；我觉得我身心象进入到一个光怪陆离，妙不可言的微软奇异世界。我内心世界的爱丽丝就象已探到迷宫似的兔子洞的底部。[译注：西方童话典故]

整个比喻的关键是设法平衡一个矛盾体：一个大型软件公司必须产出高额利润，又同时保持其机动和创造性。巨大的生产规模和创造性是有内在冲突的，成功地平衡好二者是微软成功的核心。谷歌(Google)、苹果(Apple)、或升阳(Sun Microsystem)公司似乎都还没有在这么大的规模上，证明能做好应对的平衡，但是 Cirque du Soleil 大马戏团和微软已经证明了他们能成功地应对。

—Irada Sadykhova，总监，卓越工程部门，有效学习&组织小组

在“大”公司中做“小”项目

我们在微软不仅仅做大生意，我们也在寻找下一个伟大产品的过程中，同时开展许多较小的项目。

我曾经听到比尔 盖茨比较微软产品开发和电影业的异同。他比喻每个大电影制片厂同时投注在不同的电影类别。一个类别是“大片”，如最近重拍的“金刚”。通常这些电影的制作成本非常昂贵，因此它们也带来很大的风险——它们或者成为票房毒药，例如 2002 年 Eddie Murphy 的电影“普鲁图 纳什的冒险”（该片耗资近亿美元而票房收入仅四百四十万，被认为是最大的失败作品），或者成为突破性的赢家，例如统治票房的“泰坦尼克”全球收入十八亿美元，又如“拿破仑 戴纳麦特”以四十万的投资收获四千六百万票房（不计算 DVD 和其它销售）。另一个类别包括知名影片系列的续集，如“蜘蛛侠二”和“蜘蛛侠三”或“星球大战”系列的续集。通常我们期望这些影片带来高额票房。

所有电影制造商都有一个目标，就是要盈利。但经常是通过降低投资量来减低风险。比尔认为意想不到的成功制作和成功的系列影片续集一直都有，但长期成功的关键是制片商通过同时制作很多类别的影片将风险分摊，和尝试找到下一个可拍摄很多续集的知名电影。一部知名电影的续集拍摄带来的可预期的长期利润可以用来资助其他项目，很可能下一个大的知名影片系列就因此产生。彼得 杰克逊的“指环王”系列和“哈利波特”系列都是大胆的投注拍摄新的“大片”而后来产出多部高盈利续集的范例。事实上，这两个系列都有新续集在制作中——彼得·杰克逊正在制作两部来源于“指环王”系列的影片；还有三部“哈利波特”的书将被拍成电影。

微软的产品战略和上述电影业很相似。大型而且成功的微软产品部门（如 Microsoft Office, Windows, Visual Studio, Exchange, SQL Server, Hotmail, and MSN Messenger）的部分利润被用于“孵化培育”其他项目的投资里。这些项目有望成就未来的大型而成功的微软产品。在软件产业，“孵化培育”是很普遍的。孵化培育 (incubation) 这个术语通常用于小型的软件创业公司。这些公司有很好的想法和值得期待的目标市场，但它们却还没有发布第一个产品。微软较大的创业团队通常由总经理(General Manager)管理，较小的创业团队通常由产品部门经理 (Product Unit Manager, PUM) 管理。这些团队通常自成一体，拥有大部分工程开发所需的资源。这样，他们的命运就掌握在他们自己的手中。

我曾为 BUM²工作过

在 1997 年，我在从 MSN 分离出来的 Internet Services Business Unit (ISBU, 因特网服务事业部)工作。在原来的 MSN 团队，我们研发了许多新技术来推出 MSN。我们有比 Microsoft Exchange Server 扩展性更好的 e-mail 服务，我们有比 Microsoft Active Directory 目录服务扩展性更好的登录服务器，我们有内容管理服务器和许多其他服务器。几乎每家互联网服务提供商 (Internet Service Provider, ISP) 如 CompuServe, Comcast, AOL 都拥有或需要这样的服务器。然后可能是有人提议打包这些东西来出售给小型的 ISP 和电话公司，因此就成立了 ISBU 部门。很多来自 MSN 的工程师加入了 ISBU，一些小公司也被 ISBU 收购了。ISBU 的产品后来发展为 Site Server(网站服务器)、Commerce Server(商业服务器)和 Microsoft Commercial Internet Server (Microsoft 商

²译者注：bum 原意为流浪汉，懒汉

用互联网服务器，MCIS)。这种情况下，与上文所述的 PUM 或 GM 不同，当时管理着整个工程团队的总领导被称为“商务单位经理”(Business Unit Manager)。我们称他们为“BUM”。可英文“BUM”的意思可不怎么好听：原意为流浪汉，懒汉。所以呢，我们 ISBU 的所有工程师就有为几位“流浪汉，懒汉”工作过的历史了。

微软经常投资到内部的技术孵化培育项目，例如我们的 IPTV、车载电脑和机器人项目，以及微软研究院 (Microsoft Research, <http://research.microsoft.com>) 所做的所有工作。我们也投资可以促进软件产业整体变化的项目。在 2007 年 4 月，和超过 10 个合作伙伴一起，我们推出了微软公司软件作为服务(Microsoft Software as a Service, SaaS) 技术孵化培育中心计划。

在微软，有很多技术孵化培育的模式。成熟的产品往往孵化培育新的功能特性。例如，Office 2007 的彩带(Ribbon)是一项改变了 Office 用户界面的新功能，又如 Windows Vista 的 Shuffle 功能。最新版的 Visual Studio 所带的源代码覆盖率工具最初是作为微软研究院和 Windows 的内部工具开发的。

我们用过的孵化培育创新的另一种方法是使用内部的风险投资团队。任何雇员可以提交一个想法，然后风险投资团队将使用类似风险投资业界公司的流程来审查这个想法的可行性。就像在风险投资业界一样，只有极少数的超前想法通过这一机制得到资助。

“孵化培育”的想法的另一个来源是比尔 盖茨的“思考周”(ThinkWeek)。微软员工可以提交白皮书详细阐述自己新的想法和创新。在大约每年两次的“思考周”中，比尔阅读和评论许多这样的白皮书。“思考周”结束时，比尔的批注（曾经是手写的）会被整个公司共享。在 2005 年，华尔街日报这样描述“思考周”：

这是一个每年两次的仪式，它可以影响微软和高科技行业的未来。一个“思考周”的想法能为一项被千百万人使用的技术打开绿灯，或将微软送入新的市场。1995 年的一个“思考周”促使盖茨先生写出了论文“互联网浪潮”，引领微软开发了它的互联网浏览器并将网景公司 (Netscape) 彻底打败。诸如创造微软的平板电脑，开发更安全的软件，开始在线视频游戏业务等计划都是在各个“思考周”中孕育出来的³。

就 2008 年秋季的“思考周”而言，员工们提交了 375 份白皮书，比尔评论了其中的 125 份。设立这项计划就是为了鼓励员工提出创新和前瞻性的想法。许多非常有希望的想法得到了开始最初的研发资助。比尔还向全公司公布他推荐的阅读清单。最近，我写的关于我们如何改进软件服务测试的白皮书进入了比尔的推荐清单。这使我感觉提出的理论被盖上了批准的图章。

微软平面的诞生

微软平面 (Microsoft Surface, <http://www.microsoft.com/surface/>) 这个概念曾经是微软研究院的几个人写的一份白皮书。在比尔读过这份白皮书后，一个产品单位经理(PUM)参加了该团队，并开始推动将想法变成可行的产品。其结果是一台计算机被内置于一个桌子里，而计算机的显示屏幕就是桌子的表面。我认为这看起来很像 90 年代一度出

³ Robert A. Guth, “In Secret Hideaway, Bill Gates Ponders Microsoft’s Future,” Wall Street Journal, March 28, 2005, http://online.wsj.com/article_email/SB111196625830690477-IZjgYNklaB4o52sbHmJa62Im4.html.

现在酒吧中的桌面视频游戏。用户可以与微软平面使用手势交流，或通过把实际物体，如名片或 Zune MP3 播放器，放在平面上来实现交流。

微软最近开始使用一种被称为“寻求”（Quest）的流程来创新想法。字典中将“寻求（quest）”这个动词定义为“搜索和寻找”。在微软，“寻求”是于我们制定的长期愿望和目标有关的。“寻求”这个流程召集微软各部门的资深技术思想领导人来创建一个 5 - 10 年的关于技术创新的远景规划。通过“寻求”这个流程，微软公司的顶级技术专家横跨不同组织和产品部门进行协作，他们和企业领导人员一起规划最可能改变人们工作和生活并为微软及其客户和合作伙伴创造新的商业机会的技术革新。

“寻求”的过程中不会引出只有着简单答案的简单问题。通常，一个“寻求”历程需要很多年才能完成。常常在创建一个新的市场之前，会需要崭新的研究、建造众多的样品和发现独特的见解。一个成功的“寻求”历程必须是严谨的、有远见和有方向，而且是以客户为中心的。所提到的严谨指的是我们待人接物，正直、诚实、开通、尊重对方这些价值观所共识的部分。严谨对于微软的“寻求”历程来说意味着结合同事之间广泛的检阅和审查，和包括三位本部主席，斯蒂夫括三位寻求和其他几位执行长的公司高层领导组的监督。

这些“寻求”历程的目的不是在于制作一个产品的另一个版本，而是怎样管理一系列长远技术问题的组合。这种管理需要在跨组合作中找到和使用协同作用和力量。这些“寻求”历程的数量和焦点随着旧的“寻求”结束和新“寻求”的开始而年年变化。目前，50 多个包括从客户，信息工作者到 IT 精英和程序编写者，跨越整个微软企业和所有客户群的“寻求”历程正在运行中。

聘用多种类型的工程师

因为微软是一个既有产品又有平台的公司。所以微软有许多的有着自己软件工程师团队的合作伙伴，这些合作伙伴在微软推出的软件上再创新。与微软合作的供应商和合作伙伴加起来使微软工程师数目在全世界超过十万人。在世界各地，数以百万计的软件工程师为苹果、IBM、Sun、Oracle 和为如同 Linux 的开放源码软件而工作。与他们所组成的完整生态系统相比，微软工程师团队占有了很小的比例。然而，这也许是在当今市场最强大和最有影响力的工作团队。

提示：世界各地 40 个国家内有超过 3.5 万位软件工程师为微软全职工作。每年微软会招聘超过 5000 位软件工程师和超过 1000 位软件开发测试工程师。

与大多数其他公司相比，微软的独特之处在于它的工程程序和它一套管理软件工程师的方法。有些人可能会看到微软在软件产业中的成功而把这些独特的不同之处看成关键的竞争优势。

另一些人则指责我们的另一些产品和他们的缺陷而把这些不同看作是一种弱点。
其实，究竟有哪些软件工程的因素，使微软与世界上其他公司相比有着独特和不同之处呢？

工程类职种的划分

当你把在微软工作的超过 80000 名全球员工分类的话，超过 35000 人在销售，市场营销和信息技术部门工作。产品工程是公司开发和支持软件产品的部分。截至 2008 年初，微软在全球雇用了近三万五千名工程师。其他一万名员工的工作领域跨越了如企业管理，法律等其他职种范围。虽然微软在继续迅速增长，不过。、销售、市场营销、信息技术与产品工程部分的平衡这些年一直是相对不变的。

产品工程师是实际参与产品创造和发行的员工。这些产品包括硬件、软件、和相关服务。微软工程角色可分解成以下 10 个产品工程职种(disciplines):

- 软件测试

软件开发测试工程师 (Software Development Engineers in Test, SDETs)，通常称呼为测试人员(Tester)，有时也叫软件测试员。SDETs负责为微软所有产品维持高标准的测试和质量保证。

- 软件开发

软件开发工程师 (SDEs) 常常被称为软件开发员。SDEs 写的软件程序造就了微软产品以及软件产品的升级。

- 项目经理 (PM)

在微软，Program Manager (简称 PM)是一个相当独特的角色。该角色结合了项目管理、产品计划、和设计的要素。项目经理的工作包括给一个新产品的技术方面下定义，并监督产品的发展过程。

- 运营(Operation)

运营部门 (Ops) 是微软信息技术 (IT) 的一部分。运营部门管理和维护着微软的在线服务，以及企业内部从网络到服务器的 IT 基础设施。运营部门和产品部门有着紧密的合作，以求在软件服务架构方面能降低生产成本，和使我们的软件加服务更加可靠。

- 可用性和设计(Usability and Design)

可用性体验和设计 (Usability Experience and Design, 简称 UX) 结合了我们宣传的产品设计和产品可用性的两个角色。设计师关注的是前端用户对软件在视觉和功能方面的体验。可用性则在关注视觉和功能方面的体验的同时，还研究用户怎样使用现有的产品和新样品，然后在产品发展过程中把分析结果用于改进产品。

- 内容(Content)

在我们外部网站上，内容仍称为用户帮助和教育(User Assistance and Education)。这一职位侧重于设计和提供包括用户界面(UI) 文字、网络信息、培训、模板、栏目、书籍、测验和帮助文件等各类用户帮助，以便用户从微软的产品中获得最大程度的信息。改名为“内容”是强调微软需要专注于所有不同产品、运载工具的信息内容。

- 创意(Creative)

创意职位最多存在于游戏组。在这个职位的工程师开发和改善微软在个人电脑上使用和在 Xbox 游戏机上使用的尖端游戏软件。创意职位包括游戏设计师以及游戏艺术设计者。

- 研究(Research)

研究包括软件开发和测试两种角色。软件开发工程师和软件开发研究工程师的差异是软件开发研究工程师的重点放在研究，发表论文，和培养新的技术，而不是按照时间表发行产品。

- 地域化 / 本地化(Localization)

国际项目工程（International Project Engineer, 简称 IPE）曾被称为本地化。

本地化重点在于把微软软件的文字翻译成多种语言，并使微软软件适用于不同的国家和地域文化。国际项目工程人员还负责根据具体的地域市场需求，而修订微软的软件的展示和功能。

- 工程管理(Engineering Management)

工程管理是由管理多个职种，比如测试、开发、项目管理等团队的经理们组成的。他们的头衔通常是产品部门经理(PUM)、总经理(General Manager)、或部门经理(Group Manager)。

提示：硬件被认为是一个特殊的职种领域，其中包括硬件开发工程师、硬件测试员、及硬件项目经理。这些职种与在软件和软件服务方面的相关职种相似。但不同之处足以令他们有着自己的职业道路和培训支持。

产品工程师们由他们的职种所划分排列。工程师的人数在每一个职种都不同，多至万人，少至几百人。最大的三个职种是软件开发 (SDE)，项目管理 (PM)，与软件测试 (SDET)。第四大的职种是迅速增长的 IT 运营 (Operation)，这与公司对服务类(Services)软件日趋重视的转变有关。

微软以前并没有把这些职种分离。在早年，每个人的名称都是“技术员”。专业化角色始于 1979 年，用不同的标准职业名称来辨认确定工程师的职业道路开始在 1980 年代初期。

三位组合(Triad)这个名字指的是软件开发、软件测试和项目管理三个职种(见图 1-5)。这是三个在微软的最大的工程职种。来自于三位组合职种的大量工程师都在产品开发团队里。随着重点转向软件和软件服务，IT 业务在迅速增长。



图 1-5 测试、Dev 和 PM 三位组合。

一般的大型软件公司都会雇用以上几类工程人员。其中每一类都有它自己的行业交流活动，例如测试行业的软件测试分析和讨论会议 (Software Testing Analysis and Review

(STAR)), 又如开发行业的软件安全开发会议 (DEFCON (<http://www.defcon.org>) for security developers) 等。会议是为了提高专业技能和促进整体行业发展。

在微软和在整个软件产业，软件测试都是其中一个较大的行业。在这本书里，我们希望通过与您分享微软测试人员的故事来阐述我们的一些观点，揭示是什么使微软测试工程师既类似又区别于其它公司的同行。

测试工程师突破 128KB 内存障碍

1985 年，为测试人员购买设备的资金还非常有限。当时 Excel 的 Mac 版本可以在至少 128 KB 的内存下运行，所以测试人员所具有的最好的计算机就只有 128 KB 的内存。当团队开始研发 Excel 的下一个版本，内存要求增长为 512 KB 时，有好一阵子，测试人员的计算机都没能得到升级。最后，当情形已经发展到测试人员不能进行必需的测试时，才终于让我们进行了内存升级。

- Carole Cancler, 前微软测试工程师

作为一家全球化的软件开发公司

人们经常问起是否微软的所有产品都是在美国华盛顿的 Redmond 开发的。简短的答复是：多数是，但不全是。在 1986 年 2 月搬到 Redmond 总部之前很久，微软已经是一家全球性公司。实际上，早在 1979 年微软就在日本设立了第一个国际办公机构。但直到 1998 年，微软大部分产品开发仍然是在 Redmond 的主园区完成的，比例大约在 90% 以上。

今天，在美国和世界各地，微软都有相当规模并迅速成长的研究或开发中心。在美国，较大的分部有：加利福尼亚、北达科他、马萨诸塞、纽约、南卡罗来那、得克萨斯和科罗拉多州。在世界，微软在中国和印度的开发中心每一个都有超过 1200 位工程师。新发行的操作系统 Windows Vista 包括了许多在北美洲、欧洲和亚洲开发的主要组件。未来发行的 Windows 操作系统和 Office 办公系统将会有更多部分在世界各地完成。

如图 1-6 所显示，截止 2004 年，在 Redmond 的员工人数占员工总数的比例已从 90% 下降到 81%。



地区	占总数的%
美国(华盛顿州 Redmond)	81%
美国(加利福尼亚州)	4%
爱尔兰	2%
日本	2%
印度	2%
美国(得克萨斯州)	1%
美国(北达科他州)	1%
英国	1%
丹麦	1%
中国	1%
以色列	0.5%
法国	0.3%

图 1-6：2004 年人数列前 12 位的微软开发中心

这个将软件工程人员分布在多个开发中心的趋势还在继续并加速。近年来，微软完成了一些大的并购，并决定将原公司的雇员保留在他们原来的地点。近几年来，在中国和印度雇用员工的比例的增长都比 Redmond 的增长要快。如图 1-7 所显示，到 2008 年初，Redmond 的员工人数只占全球员工人数的 73%。



地区	占总数的%
美国(华盛顿州 Redmond)	73%
印度(Hyderabad)	7%
美国(加利福尼亚州)	4%
中国(北京)	2.3%
爱尔兰	1.6%
英国	1.6%
丹麦	1.3%
美国(北达科他州)	1%
日本	1%
中国(上海)	1%
以色列	0.9%
美国(马萨诸塞州)	0.5%

图 1-7: 2008 年人数前列 12 位的微软开发中心

微软的国际化趋向是按公司的计划进行的。但其目的不同于许多其他跨国公司之在于，它不是为了更加廉价的劳力。微软的工程师全球化的努力是为了开拓人才渠道，获得市场和利用新技术。由于美国计算机专业毕业生数量继续下降及等待美国的工作签证时间继续增长，微软必须直接到人才所在的地方。有一些杰出的工程师就是不愿搬到美国华盛顿州的 Redmond。另一个考虑就是通过在一个特定国家雇用许多工程师，

有助于微软开发和稳定那里的市场。例如，在中国，有许多微软工程师在迅速增长的各个技术领域工作。中国政府现在从自己的利益考虑必须减少盗版软件，因为它直接影响到它自己的公民和他们的工作成果。另外，微软开发的技术也使有效的全球性合作成为可能。现在，零散的小团队比几年前能更好地被联系和整合到大团队。根据目前的变化速度，我觉得在 Redmond 的工程师数目在下一个十年内会减少到总工程师数目的 50%。

提示：2008 年年初，微软印度开发中心(IDC)雇用了超过 2400 位工程师。微软在上海和北京分部雇用了超过 1400 位工程师。

本章小结

微软是世界最大的软件公司，也是世界上软件测试工程师工作的最佳地方。我们成功地建立了能创造巨额销售和利润的国际水平的企业。微软的丰富历史，帮助全世界的个人和企业充分发挥他们最大的潜力的宽广愿景，以及一组核心价值观必将引导微软测试工程师们生产出高质量的世界级软件。

随着企业的演变，借鉴一些大而且成功的工程团队的经验，我们将基于 PUM 的组织结构逐步演变成了更加容易扩展和具有更高效率的共有的团队组织结构。微软仍将不断地投资在能孕育新产品的概念和技术上，这包括已成功的产品和全新的产品。

微软有超过 35000 名各类工程师工作在横跨 10 个不同工程行业。他们开发硬件、桌面应用软件、服务器和服务产品。微软的产品开发变得越来越全球化。在 2008 年，超过 28% 的产品开发是在美国华盛顿州的 Redmond 以外的地方完成的。并且这种开发全球化发展趋势预计将会继续。

第 2 章 微软的软件测试工程师

肯·约翰斯顿

起初，微软并没有测试工程师，没有本地化工程师，没有项目经理，也没有可用性工程师。刚开始的时候，所有的人都叫做工程师；当然，还有销售和市场人员。

在第一章“微软的软件工程”里，我们介绍了今天微软的 10 种不同的工程师职种。微软开始区分职种之前，其实没有太多的技术职称。比如软件支持工程师，只被看作工程师的不同工作之一，并没有自己的职业发展轨道。在公司早期，所有的工程师有着相同的职称和统一的职业发展轨道。当然那时微软只有不到 50 名员工，软件开发本身还没有真正成为一个行业，微软也还不是上市公司。

很长时间以后，微软才发展出不同的工程师职种，并制定出其相应的职业发展规划。相对来说，项目管理和可用性工程是两个比较早出现的职种。在 1990 年左右，可用性工程师成了一个正式的职种。可用性工程师的工作就是确保我们的软件易于被最终用户使用。微软有些软件设计的太复杂，普通人不知怎么用。举例来说，Office 软件中的 Word 邮件插入功能（把邮件插入到文档中印刷或作标签），很多用户开始都不会用。有些读者也许会说，就是在今天这个功能的设计仍有待改进，不过那应是另一本在微软怎样设计软件的书的内容了。

当微软创建了软件开发这个职位不久，就把软件测试职位独立出来成为一个新的职种。微软最早的测试工程师，是一位在 1979 年加入微软的高中实习生，叫罗伊德·福林克（Lloyd Frink）。他的故事曾经登载在西雅图时报上（参见下图 2-1 和故事）。1983 年，微软 Archive 产品开发组雇佣了第一个全职的软件测试工程师，接着在 1985 年又招聘了一批测试工程师。但测试作为一个正式的职称，并有自己的职业发展轨道却是 80 年代后期的事情了。



上图 2-1 微软登在 1985 年西雅图时报上的招聘测试人员的广告

故事：也许我们需要在软件发布之前测试它的功能

之前我见过比尔几次，其实我就是这么得到实习生的职位的。那年我正要进西雅图私立湖畔学校（Lakeside School）高中部学习。我妈妈认识比尔的妈妈玛丽（Mary）。在一次学校组织的拍卖募捐活动中，两个妈妈闲聊讲起自己的儿子都喜欢计算机。又正巧我和比尔都在会场，就介绍我们认识。那时我 14 岁，比尔 24 岁。我们决定一起吃个中饭。几个星期之后，我和妈妈就到微软和比尔以及他的妈妈和妹妹丽碧（Libby）吃饭。丽碧在学校比我高一个年级。我把我的、并卖出的计算机游戏演示给比尔看。他就给了我暑期实习生的职位。这就是整个事情的开始。

第一个实习生的夏天，我主要给格雷克工作（Greg Whitten），帮助他测试 BASIC 的编译器功能。我们把很多的 BASIC 程序在编译器下运行，看是否能够得到正确的期待值。

--罗伊德·福林克 (*Lloyd Frink*)，前微软员工以及 zillow.com 的创始人

职位名称的含义

如果你给玫瑰花起个不同的名字，它闻起来可能还是同样的香。但你给工程师不同的职称，他们对自己职责的理解就会大有不同。对那些编程序写代码，开发产品功能的软件工程师，微软的通用职称是“软件开发工程师”（Software Development Engineer），英文简称 SDE。而微软测试工程师的正式职称是“软件开发测试工程师”（Software Development Engineer in Test），英文简称 SDET。这两个职称听起来很相像，顾名思义，也是因为微软的测试工程师也同时搞开发写程序。测试工程师的其他主要职责包括：制定测试计划、设计测试用例、分析缺陷（bug）的根本原因、参与程序代码的审查和产品设计的审查、以及开发测试自动化程序。有时测试工程师也参与产品源代码的开发，或是缺陷的修正等工作。但总的来说：因为测试工程师的工作量很大，所以直接参与产品的开发并不是很普遍。

雇佣对测试有激情的软件工程师来做测试工作是微软行之有效的创新。它完全不同于软件行业其他公司的做法。对此外界常得出一种结论，认为我们雇佣会编写程序代码的工程师，来做测试工作是因为我们想让他们写有效的自动化测试程序，从而排除手工测试。其实这只是其中的一小部分原因。那些懂得怎样写程序和计算机基本构造的测试工程师，常常具有软件测试所必备的分析技能。他们能够尽早发现缺陷（bug）并分析其根本起因，这使得他们能很快地发现类似的缺陷。拥有和开发工程师一样的计算机基础增强了测试工程师的技能，给了我们的测试工程师更灵活和动态的生产力。

在一些业界的交流活动中我们经常被问到：为什么微软不雇佣特定领域的专家（subject matter experts, SMEs）来做测试工作。例如：国际财会的规则很复杂，一个只有工程师基础的测试人员不能够得心应手的明白其所有的规则。又比如一些纵向很深的产品像客户关系管理系统（CRM）等。持这种理论的人认为：如果我们雇佣这样的领域专家，那微软只需要训练他们计算机科学和工程的知识就行了。对这个论点持不同意见的人则反问：一个公司是否会雇佣一个职业会计专家，然后训练他成为世界一流的软件开发工程师并由他来设计和编写其本公司的会计软件？当然大家都知道这是不切实际的。成为一个一流的软件开发工程师既需要对技术有激情，又需要通过多年的技术训练。

每一个软件公司都会雇用及培训它的软件开发工程师，增强其对将要开发产品的认识及相应

的知识。这不管是对专攻操作系统的软件公司还是对开发控制电能流程的软件公司都一样。而我们在软件测试上面临着两种挑战：一是我们要培养测试工程师成为被测试的产品领域的专家，二是培训他们怎样测试。

所以一个常用的规则是：雇用那些有坚实工程技术基础的人，他们有和初级软件开发工程师一样的编程能力并具备一个优秀的测试人员所需的其他属性特征。我们称这些属性特征为测试 DNA，我们会在这一章的后几节里来具体讨论。

像任何规则一样，这里也有特例。微软大多数的测试人员是软件开发测试工程师，但在有些领域，我们会找一些特定的领域专家来做测试工作。比如，懂得全球会计规则的专家和语音识别的研究员等。当公司朝着消费领域方向发展时，我们雇用那些懂得制造流程的专家来测试我们的设计是否符合大批量的制造流程。在这种情况下，这些测试工程师的职称不是 SDET，而是更符合他们工作性质的：语言学测试工程师（Linguistics Test Engineer）或制造业测试工程师（Manufacturing Test Engineer）。

微软测试工程师的职称并非一直都是 SDET

2005 年以前，微软实际上用过两种不同的测试职称：软件测试工程师（Software Test Engineer - STE）和软件开发测试工程师（SDE/T）。同时有两种测试职称的过程非常让人费解。在有些部门，SDE/T 职称表明该员工从事测试工具方面的开发工作，在其他部门则意味着，SDE/T 拥有计算机科学学位并写过许多测试自动化程序。那时 SDE/T 甚至没有一个明确的职业发展轨道。表 2-1 列出了 SDE/T 和 STE 各自的职责。

提示：这句话是从 2004 年 SDET 梯级指南 中节选的：“一个 SDE/T 应使用测试或开发梯级指南中最合适的一种。”

表格 2-1 SDE/T 和 STE 的任务

SDE/T 常见任务	STE 常见任务
开发测试工具	写测试计划
开发针对安全或性能测试的工具	写测试案例文档
API 或协议测试自动化	执行手工测试
参加 bug 大扫除（bug bashes） ⁴	核心测试案例自动化
发现，调试，新开和回归测试 bug	发现，调试，新开和回归测试 bug
参与设计方案审查	参与设计方案审查
参与代码审查	

即使没有一个清晰的职业发展轨道，能够同时涉足测试和开发两个领域的想法是非常诱人的。随着时间的推移，拥有 SDE/T 头衔的员工数量不断增长，以至于我们不得不决定将俩个职称合并。2002 年，微软曾想要去掉 SDE/T 职称，因为它在员工中引起许多误会。到了 2005 年，SDE/T 职

⁴ bug 大扫除（bug bashes）：很多人同时一起找 bug 的竞赛

称改成了 SDET。那时也曾想要合并 STE 和 SDET 职业发展轨道，如图 2-2 所示。测试架构师在 2003 年首次被正式认可，并包含在 SDET 职种类别里。

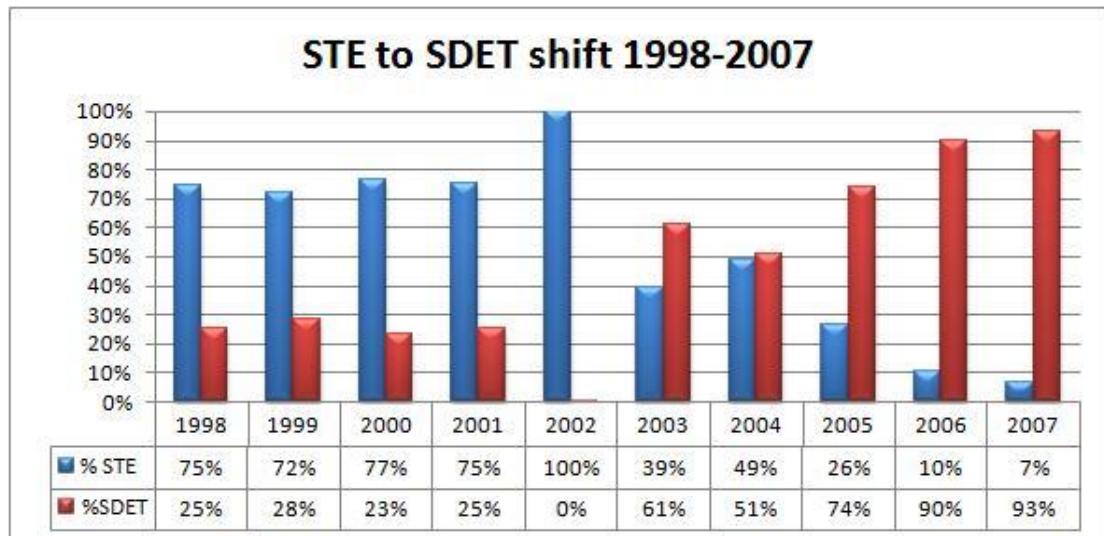


图 2-2 从 STE 到 SDET 的转变，1998-2007

我们不能称他们为 SQE（软件质量工程师）

Grant George 对会议室里的人说：“好了，让我们暂时把他们全都叫斑马好了。”我们正在辩论测试工程师的职称应当是什么，辩论已经进行了几个小时，但还没有达成共识。

“我们都同意软件测试本质上是一个工程问题。我们都同意出色的软件工程师需要有很强的工程学基础，在理想的情况下最好是有计算机科学方面的技能。我们也都同意一个优秀的测试工程师和开发工程师有所不同。他们都内在地拥有测试 DNA。”边说着，Grant 离开座位走向白板，手里拿着记号笔。

我环视了一下会议室里的测试总监们。如果把出席这个会议中所有人的微软测试管理经验加起来，总共超过了 200 年。我们有些人随着微软 Windows 一起成长，有些人在微软 Visual Studio 部门工作，还有些来自 Web Services 部门的。Grant 从他在微软 Office 工作过的背景来看待这个问题。他同时也是这里所有人中资历最高的：是当时唯一的一个微软测试副总裁。

Grant 开始在白板上写下下一个列表。“测试 DNA 应当具有系统范围内思考问题的本能、分解问题的技能、对提高产品质量充满热情、喜欢研究事物是如何工作、又怎样能被搞坏（break）。”他放下记号笔，注视着会议室里的人们：“这其实正是测试工程师不同于开发工程师的地方。通过测试软件，我们把这些 DNA 和工程学技能结合起来。我们选择的名字应该反映这个事实，并且对我们想聘用的人具有吸引力。需要表明我们使用软件开发技能去驱动测试工作。”

“我们已经建立了所有这些，Grant，”桌子边有人插话道，然后陷入了沉默。

“我想选 SDE/T。我们过去使用它，本质上描述了你上面所讲的，但我并不清楚使用 SDE/T 的过去。”Gregg 讲道。

“我们使用不带斜杠的 SDET 怎么样？”David White 提了个建议。他是人事资源部门的职业模型项目经理，他和这些公司级测试领导团队一起合作，来统一制定所有测试工程师的在微软的不同职业发展轨道。“它可以使你很快地向学校和社会上的待聘候选人表明，表明我们需要软件开发程序员，但他们的工作集中在测试方面。”

“这个名字比 SQE 好，”一个来迟了只能坐在侧面桌上、桌下就是垃圾箱和废品回收箱的人喊道，“SQE⁵总

⁵ SQE 的英文发音与擦车窗的刷子 squeegee 发音相近。

是使我想起那些在马路交通灯边上拿着刷子要给你洗车窗换钱的家伙。”（见本页脚注）

Windows 测试部门的总经理 Darrin Muir 插话道：“我喜欢这个名字。很简单，只把斜杠去掉就好。”

我们又辩论了一个小时，最后决定微软所有测试工程师的新职称是 **SDET**。**SDE** 和 **SDET** 的不同处归结为：激发工程师致力于测试问题而不是开发问题的核心 DNA。

从我们提高产品质量和测试效率的策略来讲，使用一个强有力的职业称谓来强调其与开发工程师的相似性是非常关键的。本书的下面三部分将详细介绍微软测试软件的技术。我们在微软之所以有可能采取该职业称谓，是由于我们在聘用测试工程师时就体现了要强调的技能，同时这个称谓也正好体现了在微软测试工程师平时工作中所要培养的技能。

在早期，**SDE/T** 职称的使用仅仅是为了帮助公司从大学中招聘。许多候选人已获得计算机科学学位，他们想在工作中有机会使用他们学到的编程技能。因此 **SDE/T** 职称和在测试中做工具开发员角色的职称比 **STE** 职称更具有吸引力。

虽然对所有 **STE** 的期望是他们能够在必要时编写自动测试程序，但以前我们的许多产品并不支持今天所期望的高层次的自动化测试，因此 **STE** 花在编写代码上的时间，仅占他们所有测试时间的一小部分。

2001 年，微软对其产品支持的策略发生了重大变动，这对软件测试部门产生的影响，大过于任何其他工程部门。这个变动是关于我们的产品支持周期。大多数主要产品，包括 Windows 操作系统，变更为承诺十年支持。软件对于企业至关重要，而一个企业从一个操作系统更新到另一个操作系统，或者从一套生产软件更新到另一套，则要经历一个很长的过程。因而与我们那时的只支持前一版 (N-1) 的产品支持策略就有所矛盾。随着软件产品发布周期的缩短，产品支持期限发生重叠。我们采用了一种基于年数的模型，取代了基于产品版本数量的策略。对于每年发布一次的消费者产品，支持期限是 3 年，而对于服务器、操作系统和关键性软件产品，其支持期限是 10 年。

开发 Windows 95 时，我们从未想过要支持它到 2005 年。虽然 Windows 95 有一套牢固的核心测试自动程序，能够验证大多数主要功能和关键用户场景，但它也包含为数众多的有文档和没有文档的手工测试案例。在第 4 章“测试用例设计的一种实用方法”中，我们谈到了手工试探性测试。对 90 年代末发布的产品，它是一种普遍的方法。我们雇用了大量的有短期合同的临时工和正式员工一起来参加这种按按钮式的测试。

产品支持周期的改变意味着自动测试程序可以使用很多年，因此在产品开发的研发阶段它具有无可争辩的重要性。这个共识驱使我们注重聘用更多的 **SDE/T**。随着越来越多的计算机科学专业的毕业生加盟测试部门，我们发现：测试对影响产品的设计和提高可测试性方面的能力增强了。

诸如不同层次的集成度和复杂度的提高、具有挑战性的安全测试，比如威胁建模(threat modeling)、模糊测试(fuzz testing) 或错误注入(fault injection) 等其他因素，都持续地推动我们对测试中的计算机科学和编程技巧方面更高的要求。而且公司所经历的，向服务性软件的转型和产品的快速发布周期，也都驱使我们转向自动化测试的新模型。在第 14 章，“测试软件+服务”，我们会讨论在线服务对我们测试策略的影响。

我需要更多的测试工程师，立刻就要！

微软每年都扩充其技术队伍，仅测试职种一年就新添大约 500 个职位。聘用的测试人员一半来自于其他公司，一半来自于美国和世界各地名牌大学的计算机系应届毕业生。本章前面已谈过软件开发测试职位的最佳候选人所拥有的测试 DNA，除此之外，我们还注重候选人的一些具体的技能，用人事部的话来说就是“胜任特征”。为了避免此章节读起来像人事部的聘用手册，我们这里就简单谈谈什么叫“胜任特征”及其与测试 DNA 的关系。

胜任特征用于描述一个人的行为，这些行为决定了这个人是否能够创造可观的业绩。胜任特征针对能力的强弱区分为不同的等级。大多数成功的应聘者都具备一些胜任特征，但程度却有限。虽然测试犹如微软其他十大工程领域，也要求同样的胜任特征，但是有些比如问题的分析和解决能力的特征，随着时间的推移，在测试人员身上更能体现出来。

十大工程胜任特征是所有微软工程师必备的核心能力，除此之外，针对具体的职位，比如管理、财务、市场销售等，还要求员工具备一些其他的胜任特征。下面介绍一下这十大工程胜任特征：

- **问题的分析和解决能力**：这个能力对测试人员非常重要，因为对问题进行解剖和找出问题的症结是提高产品质量的关键。
- **面向客户的创新**：应聘者是否以客户为本，是否能够充分理解软件如何才能帮助客户解决问题，并对此充满兴趣和热情。
- **精湛的技术**：我们注重的是应聘者是否通晓网络和操作系统，不仅能写代码，而且能够优化代码。
- **项目管理**：对测试人员来说，这个能力是指如何有效支配个人的时间，以及如何策划和确保一个有许多互相牵制成分的计划得以按时完成。
- **对质量的执着追求**：如果不具备这个素质，应聘者就无法胜任任何工程技术工作，更不必说测试工作了。
- **战略远见**：新员工刚开始这方面比较弱，但是如果我们将招聘能够帮助我们找到突破口的员工，以至于能够在竞争中遥遥领先，增加股东价值，那么应聘者一开始就必须具备这个素质。
- **自信**：在微软，测试人员找出的软件错误并不一定都能得到修正，在必要时，测试人员需要有自信而去据理力争。
- **冲击力和影响力**：影响力来自于自信和经验，冲击力来自于敢于革新。多数应聘者在谈到如何给自己的公司带来变革，或者如何在学校带领团队出色地完成项目时，都会体现这个特征。
- **跨界合作**：创新往往来之于各部门之间的合作，只顾埋头自己的项目，甘做井底之蛙的员工是不会成功的。

人际意识：主要指自我意识，许多优秀的应聘候选人能够认识到自己的不足之处，并且知道如何不断地提高自己，也就是有一个不断提升自身素质的计划。

校园招聘

校园招聘在微软的招聘计划中占很大的分量，“校园招聘”就是招聘应届的大学本科生或研究生。数以百计的招聘人员一年到头马不停蹄地同各大院校联系，加强与学校和教授的关系，大力介绍微软及其微软的职业发展轨道。

一旦发现合格的求职者，微软就安排面试。初次面试通常安排在校园或地区性的招聘活动场所，有时候求职者也来微软进行面试。招聘经理，比如软件开发测试工程师经理或软件开发工程师经理会亲自主持正式面试。在微软，第一流的面试人员才能参与校园面试。微软对面试极其重视，以至于对面试人员的面试技巧不断地加以跟踪和评审。

应聘者有时在面试前临时抱佛脚，在网上搜索曾经参加过微软面试的应聘者张贴的面试题目，这也不足为奇。但他们殊不知我们也经常查看这些网页，不断地修正改进我们的面试题目。经过了几轮面试之后，面试小组将做出是否录用的决定。无论应聘者是否能被录用为全职员工，我们都必须确保所有的应聘者都能对整个应聘过程留下美好的印象。这些应聘者聪明，并有进取心，正因为如此，他们才能参加正式的面试。

一旦下达聘用通知，我们通常会对职位进行宣传介绍。比如就测试职位，我们将介绍微软的测试是如何与其他公司的测试的不同。许多年来，我们都鼓励应聘者阅读像 Cem Kaner, Jack Falk 和 Hung Quoc Nguyen 撰写的《测试计算机软件》(*Testing Computer Software*) (Wiley, 1999, 第二版) 之类的著作，但是这些著作并没有真正涵盖微软测试的技术层面。

幸运的是，微软的一名测试架构师Keith Stobie，说服了《测试面向对象系统：模型、模式和工具》(*Testing Object-Oriented Systems: Models, Patterns, and Tools*) (Wiley, 1999) 的作者 Robert Binder，允许微软向应聘者推荐其著作中有关测试的章节，即第三章，“测试简介”，此章节对测试言简意赅的描述恰好与微软的测试方向非常吻合。虽然此书整体涉及的内容远远超出对测试的简介，我还是推荐所有的测试人员阅读第三章。

我能和从事具体工作的业务人员聊一聊吗？

担任微软的测试经理是我真正喜欢的工作之一。我一毕业就去了一家公司做软件开发，十二年前，我跳槽到微软做测试工程师，因为我知道，微软测试工程师的角色是不同凡响的。

微软的招聘工作做得非常出色，应聘者通过公司所提供的资料和与招聘人员的交谈中，就能充分了解微软所有的工作角色和所能提供的发展机会，但他们总希望能和每天从事具体工作的业务人员交谈。当我参加校园招聘活动时，我就经常充当这种业务人员的角色。在多数院校，测试都是在写完项目的代码后进行的，我当时就是先编写完教授布置的代码，然后保证其运行正常。测试只是计划外的事，并没有真正列入教程。

在微软，软件开发测试工程师的角色非常重要，它担负着发行优质产品的重大责任。每当和应聘者交谈时，我总是急切地想让他们明白，测试与编写代码和理解算法一样，是一门很深的学问，一名称职的测试工程师需要胜任各种各样的任务。

很多应聘者还不了解，测试工程师的一大任务就是设计优秀的自动化测试系统，测试工程师面临的挑战就是设计出自动化测试系统将产品的运行推向极限，远远超出产品的常规运行，模拟产品对数据的处理，并使产品在短短的测试时间内处理一年的数据。自动化测试系统还必须能够在多种浏览器上运行，并能测试所有不同语言的版本。多数应聘者对此了解甚少，其实测试工作还不只局限于设计和开发自动化测试系统。

测试工程师不但要对自己负责的功能了如指掌，还必须挑战自己去通晓其他相关的功能，比如，了解客户端和服务器软件的相互运作及最终用户的体验。测试工程师还必须非常熟悉系统是如何由防火墙、路由器、后端服务器以及其他部分组成的。测试人员还必须对客户充分了解，在可用性、辅助功能、安全性等各个方面完全站在客户的立场看待问题。无论是为了自身提高还是满足工作需要，测试学问是永无止境的。

许多应聘者认为测试人员对软件设计没有影响力，意见也不会被采纳。其实微软最可贵之处就是任何人都可以对设计提出自己的见解，只要言之有理，尤其是测试人员，他们既了解客户的需求又熟悉产品的运用。在微软，产品是集体智慧的结晶，绝非个人所为。测试人员有权利和义务探讨产品的功能特性，以及如何实现这些功能，甚至于推荐产品所需的功能。我和我的测试人员就经常这么做。

测试犹如选修一门课，成绩完全由期末考试来决定。我曾经记得参加期末考试，然后等待成绩的公布，知道成绩后，我不是喜就是悲。现在在微软做测试，我勤奋数年开发优秀产品、确保质量，然后送到成千上万的用户手上，等待着成千上万的用户给我打分，这种心情是难以形容的。

- Microsoft Office 测试 经理 — Patrick Patterson

业界招聘

业界招聘是指我们所聘用的人员已经在软件或相关的行业全职工作了至少一年或一年以上。业界招聘旨在聘用资深的技术人员，而校园招聘旨在挖掘有潜力的毕业生。不过我们在业界招聘的技术人员也有等级比较低的，所以资历并不代表一切。

每年我们新聘用的测试人员大约有一半来自校园，一半来自业界。虽然有些来自业界的员工在原来的公司也是做测试的，但是大多数新雇员，包括本书作者在内，都未曾做过全职的测试工作。

当我们从业界招聘时，我们通常会寻找有测试经验的或者是从事过质量管理的软件开发人员。从我们收到的个人简历上看，往往有数年测试经验的人员却没有计算机知识或编程技术。最理想的候选人是那些曾身兼两职，既从事产品开发也负责产品测试的软件工程师。

最使许多来自企业单位的新雇员大吃一惊的是微软测试的规模及其巨大的影响力。在微软，软件开发人员和软件测试人员的比例是一比一，而在软件行业，典型的比例是五比一，有的是十比一，甚至更高。当比例如此悬殊时，测试组只能抓根本而无暇顾及测试技术和方法的改进。

学习怎样成为一名微软的 SDET

在微软，新职员开始上班后，首先要经过上岗培训过程，其中包含最初几天的新员工培训 (New Employee Orientation, NEO)。新员工培训不分职别，通通集中在一个大课堂里。新员工培训结束之后，新的 SDET 们找到所在组的管理员，然后找到他们的办公室以及直接的主管和技术指导(mentor)。

在所有部门中，微软的卓越工程部(Engineering Excellence Group, EEG)专门向员工提供技术培训。卓越工程部给新 SDET 上的第一门测试课是面向 SDET 的微软的软件测试 (*Testing at Microsoft for SDETs*)。这门课通常在员工自进微软起始之日的 12 个月内完成。本书的部分章节涵盖了这门 24 小时的课程的大部分内容。尽管许多技术课程和讲座系列采用在线授课，但是，诸如此类的在 SDET 培训计划中具有里程碑意义的课，如图 2-3 所示，是由经验丰富的测试工程师在课堂授课的。通过这种方式，学员可以得到大量的机会广泛讨论，并且可以在很多侧重方面有深入的练习。

S D E T 培训路线图

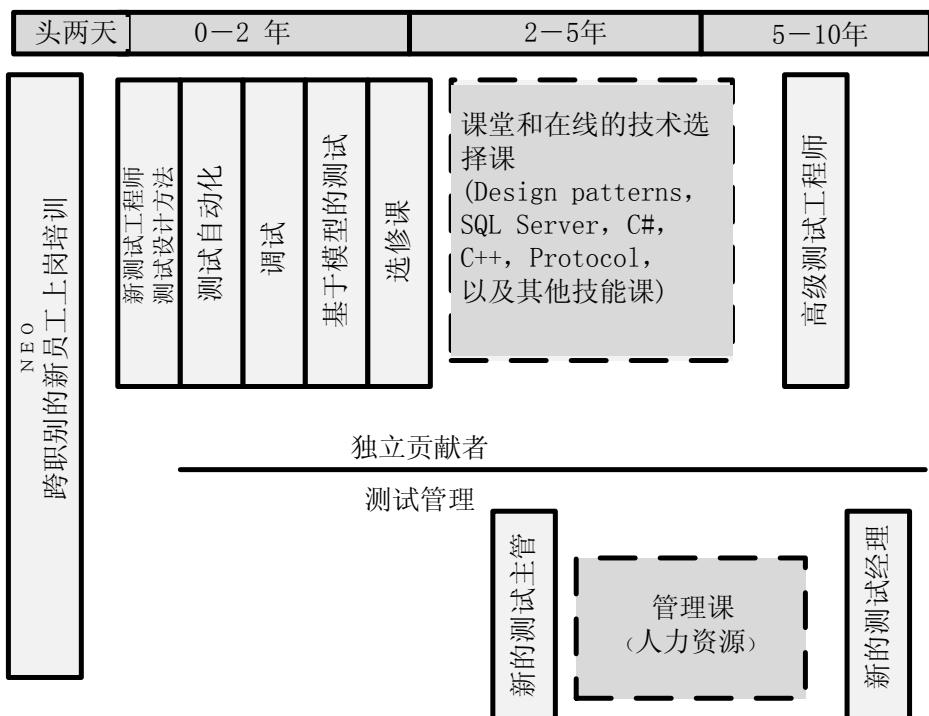


图 2-3: 截止 2008 年, SDET 培训路线图概略

微软工程师的职业发展

在微软每个员工都可以换职种，每年在每一个职种中都有许多人这样做。其实不管哪一个职种，说到底都只有两种选择：做独立贡献者(Individual Contributor, IC) 或做主管经理。所有低级别的工程师都从独立贡献者做起。每一个职业生涯都有一个主要的转折点。对主管经理来说，这个转折点可能是从管理几个工程师的主管到管理其他主管的经理；对个人来说，可能是从影响一个产品到影响一个系列产品。这些转折点被称为职业发展阶段(Career stages)。微软采用的是 Stephen Drotter⁶在通用公司(General Electric)工作时创立的职业发展阶段模式。微软对每一个职业发展阶段，都制定了详细的职业要求，以此来帮助员工了解公司对此阶段员工的期望结果。

有时一个员工会去试一下做管理者，如果不合适再回去做独立贡献者。这在微软是再普通不过的事了。但是随着高级开发测试工程师的增加，他们需要具备跟高级工程经理一样的领导才能和商业敏感性。如图表 2-4 所示。在某些情况下，即使很资深的工程师也会在作单纯技术工作和团队管理之间跳来跳去。

⁶ Ram Charan, Stephen Drotter, and James Noel, *The Leadership Pipeline: How to Build the Leadership Powered Company* (San Francisco: Jossey-Bass, 2000).

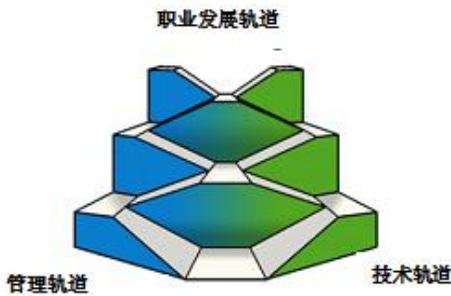


图 2-4：管理和技术轨道交叉发展

譬如，大卫·卡特勒(David Cutler)于 1988 年加入微软，主持 Windows NT 的开发。这个部分就是在今天也是每一个发布的 Windows 操作系统中的核心部分。尽管大卫在微软曾经管理过许多团队，但是大家普遍认为他是一个架构师，是微软最高级别的独立贡献者。相比于他的管理能力和帮助员工成长的能力，他的精湛技艺，渊博的系统结构知识和在业界的影响力得到了大家的一致公认。

另外两个例子是 Eric Rudder（技术策略部门高级副总裁）和 Jon DeVaan（Windows 核心操作系统部门高级副总裁）。这两个人都曾经既做过多年的小型团队的策略领导，又都管理过有几千名员工的大型部门。

微软的“技术院士(Technical Fellow)”是独立贡献者职业阶梯上的最高级别。这个头衔相当于管理职业阶梯的高级副总裁。“杰出工程师(Distinguished Engineers)”则相当于公司的副总裁。两者在软件业界被视为其所属专业领域里的精英，经常参与诸如制定业界标准的活动。这些高级工程师除了参与日常开发软件产品工作之外，还对公司范围内一些重要技术活动有积极的影响。比如给微软内部两年一度、由比尔·盖茨主持的技术战略思想周(ThinkWeek⁷)提交他们写的白皮书或为别人的白皮书提出反馈意见。

测试职种的发展道路

有些公司认为测试是低级工作，而开发职位则是一个测试工程师将来可能发展的方向。但在微软，测试职位和开发职位是平等的，并且具有同样多的职业发展机会。

测试架构师

微软于 1999 年设立测试架构师职称。这是专门为那些对产品有影响的高级独立贡献者而设的。测试架构师的职称反映了一个 SDET 对其所工作的产品的广泛影响力；而高级 SDET，首席(Principal) SDET，合伙人(Partner) SDET 的职称则主要是给那些对某个产品的功能产生影响的人。不过请记住，很重要的一点是测试架构师是一种角色而不是一个职位。尽管一个高级测试工程师可能会晋升为测试架构师，但是并不是所有的人都会成为测试架构师。通常，有些部门会出于商业需要或策略需求而设立一个测试架构师的角色。但有时候你也会看到一个高级测试工程师发挥测试架构师的作用，但是却没有测试架构师的职称。注意，此处所讨

⁷ Thinkweek：微软一年两次，为期一周的会议，Bill Gates 和微软其他高层管理者汇聚一堂，商讨未来公司有潜力的新道路

论的是测试架构师的角色而不是测试架构师的职称。

微软并没有具有普遍性或代表性的测试架构师角色。微软的测试架构师们致力于各种各样的目标、承担着各种各样的任务。有些人花时间开发测试的基本结构、测试框架、或者评估产品功能、创立复杂的测试。有些则负责管理其所在部门里的某项特殊技术。还有些则负责怎样提高测试效率的咨询。所有测试架构师的角色的共同点和主要职责是为本部门提供技术指导和制定策略方向。测试架构师的级别通常表明他们的职责范围——是致力于产品的一组功能、一条产品线，还是跨越整个部门。当然，除了被期待对现有产品负责外，高级测试架构师还应该超越产品的现有版本而能高瞻远瞩，并且可能的话，还拥有两三个不与某个特别产品的发行有直接关系的成果。

微软的测试架构师不仅要有效地影响测试领域而且也要在开发和项目管理方面发挥影响力。测试架构师必须能驾驭产品的质量、提供指导、反馈和建议，以提高整个工程部门的质量规范。

前面谈了测试架构师“是什么”，那么，测试架构师“不是什么”呢？测试架构师的职称不是因级别或经验而授予的。设立测试架构师是一种投资。这种投资结合了市场对有能力帮助做灵活变化的人才的需求。值得强调的是测试架构师并不是一种职业发展轨道。对测试架构师的技能要求和对相似级别的其他职务的职业阶段发展要求是一致的，都强调跨组织的沟通和推动变革的能力。

提示：截至至 2008 年，微软全球的 9000 多测试工程师中只有四十几位测试架构师

测试独立贡献者

软件设计测试人员的职业道路从 SDET1(也叫 IC1)开始，持续到合伙人（Partner）SDET(也就是 IC6)。(参看表 2-2)。不同的级别主要是根据技术深度、技术广度和影响力范围来区分的。一个 SDET1 通常处于学习测试、学习关于微软怎样开发软件、如何在一个定义明确的产品功能中找寻漏洞的阶段。而一个经验更加丰富的 SDET 也许更专攻在一个领域，比如对性能或者安全性的测试。当一个 SDET 达到合伙人级别的时候，他应该具有在丰富的、不同方面的经验，也许还担任过一段时间的测试架构师。

表 2-2 从 SDET 独立贡献者开始的职业发展历程简介

职业阶段名称	软件开发测试工程师	软件开发测试工程师 2	高级软件开发测试工程师	首席软件开发测试工程师	合伙人软件开发测试工程师
对用户的影响	从产品支持服务部门和其他的渠道来探访用户的反馈，从而阐明产品功能并写出测试用例	直接与用户交流，从而提供有建设性的反馈意见以避	对用户关心的产品整合和特定使用场景，角色的创建提供方案，以达到客户期望	具有直接建立用户关系的技能，促进用户和产品部门之间的交流	领导覆盖整个产品线的客户需求的深度理解，从而促进产品设计
对测试的影响	阐明产品功能如何运行以避	提供有建设性的反馈意见以避	识别有可能引入漏洞的高风	在主要产品上领导测试方法	在整个产品线上领导测试方

免模糊不清的 要求	提高产品规范 和技术设计	险设计模式 和技术创新	法和技术的创 新
--------------	-----------------	----------------	-------------

影响力范围从一个狭窄定义的产品功能扩展到一个系列产品功能、一个完整的产品。比如 Microsoft Office Word 或者 Microsoft Media Player，直到最后达到一条产品线，比如 Office 或者 Windows 系统。影响力可以像测试架构师的职位那样，基于测试的各个方面横向延伸，也可以基于一个像协议安全那样的技术领域纵向延伸。

合伙人 SDET 并不是一个工程师的职业发展轨道的最高点，但却是测试职业发展轨道的最高点。合伙人 SDET 比杰出工程师 (Distinguished Engineers)(相当于公司副总裁级别)要低一个级别。这并不是因为微软觉得在测试领域不需要杰出工程师。而是我们相信随着工程师们在职业道路上不断地前进发展，他们的行为表现变得越来越相似，不同学科差异所带来的影响价值减少。在某种意义上，每一个从事 10 个工程学科中的任何一个学科的工程师，随着他们的技术领域发展都可以达到成为另外一种学科的工程师的水平。

很多非常著名的工业界的杰出人物都是微软测试工程师行列中的一员。他们有很成功的个人日志，经常参加各种会议和书籍的写作。例如，James Whittaker，是 "How to Break Software" 系列书的作者。他在 2005 年加入微软成为可信赖计算(TrustWorthy Computing)组的一员，主攻提高软件的可靠性。Keith Stobie，Spaces.live.com 上 TestMuse 日志的作者，是策划和支持一年一度在俄勒冈州波特兰市举办的太平洋西北地区软件质量会议 Pacific Northwest Software Quality Conference(PNSQC) 的活跃分子。Tom Ball 领导的测试验证和度量组是一个小型的研究人员梯队，他们通过在数据使用中产生新颖而独特的结果来推动产品质量的提高。组里的每一位成员都因为他们在测试度量上的研究而被广泛认可。

这些只是我们在微软如何投入去发展和支持世界一流的测试专家的几个例子。

成为管理人员并不意味着升职

测试管理是 SDET 的另外一条主要的职业发展轨道。一个测试管理者可以在测试员工梯队里向上发展，管理更大的团队和领域。当然，管理的职业发展轨道向上的方向是一个不断收缩的金字塔，很多测试管理者在远没有达到副总裁级别的时候就因种种原因停滞不前了。在测试管理的职业道路上还有一个主要的转折点，那就是成为部门总经理，如表 2-3 所示。正如在第一章描述的，工程总经理所领导的部门包括开发、测试和项目管理。

表 2-3 从 SDET 管理开始的职业发展历程简介

职业阶段名称	软件开发测试主管	软件开发测试经理	软件开发测试总监
职业阶段	经理	经理的经理	
产品范围	产品功能	产品	产品线
	一个 SDET 主管的工作范 围通常是一组产品功能，	一个 SDET 经理的工作 范围通常是一个主要的	一个 SDET 总监的工作范 围包括代表一个损益中

	或是自成一个小的子系统的非常复杂的产品功能或者是组件，或是一个简单的产品。产品功能的例子包括语音识别服务器，C#编译器，Microsoft Office PowerPoint 的图形引擎和 IP 栈。	产品，或是构成一个产品的非常复杂的产品功能，或是一条简单的产品线。SDET 经理是产品线的主要贡献者。产品的例子包括 Word、Microsoft Money、和 Windows 内核。	心的产品线，或是一条产品线下的非常复杂的系统和结构。产品线的例子包括 Windows、Office、MSN 和 Microsoft Exchange。
招聘职责	领导一个组的招聘流程	积极地优化整个团队的招聘流程和实践	领导一个覆盖整个产品线的全面有效的招聘计划

在微软，同时在整个软件工业界，一个很普遍的问题是如何被“提升”到测试管理者职位。这是一个很意味深长的问题，因为它似乎意味着管理者会拿到比普通工程师更高的工资或者是有一个更好的办公室。而事实是，在微软，当一个工程师转向管理职位时，这是一个“平级”的变化——也就是说，并不包含升职成分。将来的升职是基于这个工程师的技术水平和领导他的小组的个人成果。这是一个微软有别于很多公司和政府资助企业的模式。在那些公司或者企业，职位决定工资水平，而管理的职位的工资水平总是比非管理职位的要高。

SDET 主管（SDET Lead） 是测试管理的第一级。一个 SDET 主管通常管理一个由 2 到 10 人组成的小组。组的规模大小取决于为了发布某一特定产品功能，组件(比如打印，图形)或者一个共享的产品功能的工作量。一个组也可能负责某个特定的测试领域，比如性能、规模、或者安全性。所有这些职员由 SDET 主管直接领导。在主管和职员之间没有其他管理层。

注意到有一点很重要，那就是，在决定一个 SDET 主管能否在职业上继续前进和发展的因素中，技术复杂度和他的技术水平远比他所管理的测试组的大小来的重要。负责测试产品安全的小组就是一个例子。这些规模小，但技术性高的组通常对产品的质量有重要的影响，所以在这样的组里，你可能常会发现一个更资深的 SDET 主管。在一个相对规模较小的组，SDET 主管自己也需要做很多的测试、编程、分析和记录软件漏洞的工作。从最高级别的执行董事到刚加入的新员工，产品组的每一个成员在发现产品漏洞的时候都应该对漏洞记录存档。随着一个组的规模越来越大，主管需要承担更多的管理职责，而在具体的技术工作上会少花些时间。无论组的规模大小，SDET 主管都应该具备很强的技术能力，同时也担负起一个组的技术领导工作。SDET 主管通常都是组里对一个产品功能懂得最多的工程师，同时也是组里最好的测试和开发人员之一。

希望主管有很强的动手能力和很高的技术水平的期望是与所有工程学科的期望相一致的。开发主管通常对产品开发的贡献是不会比其他任何组员少。项目管理主管通常涉及最复杂的产品功能，或者处理最复杂的协调工作。无论是不是在管理的职位上任职，每一个工程师都应该具备很强的动手能力和很高的技术水平。这种期望是微软精髓的核心。这种企业文化可以很容易地追溯到微软刚成立的时候。那时候，比尔 盖茨会在晚上把代码从头读到尾，甚至重写其中的某些部分。

测试管理人员

如第一章所述，微软是一个大公司，对任何的规则总有一些特例。管理位置的职称也是一样。测试经理有着不同的职称也承担着比测试主管更广的职责。表 2-4 列出了最普遍的测试管理职称和他们相应的团队大小和组织的深度。

表 2-4: 测试管理职称

职称	团队大小	组织的深度
软件开发测试主管和高级软件开发测试主管	2 - 10	1
测试经理, 高级软件开发测试经理, 软件开发测试经理	15 - 50	2
部门测试经理, 首席测试经理, 测试总监	30 - 100	3-4
部门总经理, 测试副总裁	200+	4-5

测试经理较少需要亲自作具体的测试工作事项，比如编写和执行测试用例。但每一个在测试领域的人，无论他在什么级别，都会亲自动手找软件的缺陷（bug）。测试经理仍然需要懂得技术，但会被要求多注重于建立测试的流程和工具，而不是在具体的功能测试上。

一个测试经理会花很多时间在培养和提高其测试团队的素质和技能上。同时测试经理会和产品部门的管理层一起做质量评估来决定其产品的质量是否达到提供给客户的标准。

本章小结

微软采取了一种独特的不同于业界其他公司的软件测试方法。公司的测试工程师的数目比开发工程师更多。而且我们对所有的测试工程师都强调其软件工程技术能力。这种独特的方法甚至表现在我们给测试工程师的职称上：软件开发测试工程师（SDET）。

公司每年都招收 500 多名新的开发测试工程师。我们通过积极的招聘渠道既雇佣那些其他公司的有经验的测试工程师，也雇佣那些刚从学校计算机或是其他相关专业毕业的新人。对那些没有测试经验的新毕业生，我们有完备的培训计划和课程来增强他们的测试技能。

对软件测试技术技能上的强调，使得在微软的软件测试成为了一个全面的职业发展轨道。这点无论对测试管理者还是对独立贡献者都一样。测试工程师可以像开发工程师一样通过自己技术技能的提高来获得高级的技术级别。

微软的 9000 名软件开发测试工程师在我们的产品开发中担任着确保产品高质量的重要角色。这个充满活力的大部队通过应用广泛的技术来不断地提高我们的工程水平和产品质量。

第3章 工程生命周期

阿伦·培智

我喜欢做饭。对我来说，做饭的过程是很有趣的，你需要协调处理不同的菜肴，还要保证它们都能按时做好。从我的“天才厨师”母亲那儿我学到了自己的一套方法——边做边补。简而言之，就是“跟着感觉走”。我已经做过很多菜了，因此，我可以很自如的打开橱柜看看哪些原料合适。菜谱只是我的参考——从菜谱上或者可以得到一些基本概念，比如需要些什么材料，要煮多久，或者可以得到新的灵感。我的方法非常灵活，不过也有一定的风险。我也许在选择替代品的时候出错（比如，制作 strata 的时候我建议你一定不要用豆奶代替牛奶 [译注：strata，可加入蛋、肉、蔬菜等佐料的烘培奶酪食品]）。

就像软件测试一样，我做饭的方式也是根据情况灵活调整的。比如，如果有客人来吃晚饭，我会比只为家人做菜时对原料份量多做衡量，还会减少使用替代品。为了减少我的意大利烩饭味道不好的风险，我会在做的时候稍微正式些。对于那些负责为上百人准备晚宴的大厨，我只能去想象他们是如何完成任务的。当你为这么多人做饭的时候，用量和均衡变得更加重要。此外，为了满足不同客人的要求，大厨的挑战是做出符合客人们各种口味的菜。当然，最后所有的菜品都要按时做好并且端上桌的时候保持新鲜。在这个例子中，“出货时间”是没法改变的。

做软件和做饭有许多相似之处。按部就班有按部就班的好处，灵活机动有灵活机动的好处。当然，一旦用户多了，不管做什么东西都会带来新的挑战。本章将会介绍一些微软所使用的开发软件产品的方法。

微软的软件工程

在微软，并没有一个单一的软件开发模式适用于所有的产品部门。根据产品的规模和应用范围、市场条件、部门大小以及过往经验，每个组都会决定最适合自己的模式。一个新产品的开发可能会由 TTM (Time to market, 指产品从构思到市场销售的时间) 来推动，这样能在该类软件出现领头羊之前抢得先机。现有的产品可能要更注重以创新来撼动领先的对手，或者保持住领先地位。每种不同的情况都需要用不同的方式来设计、实现和发布软件产品。尽管有时也要有变通，很多实践方法已经被广泛采用，同时也可以在工程实践中进行显著的实验和创新。

对软件测试人员来讲，理解这些常用工程模式的区别，所在的组使用的模式，以及所在组正工作在该模式的哪一阶段，不仅有利于产品的规划（了解将要做什么），也有利于计划的实施（了解该模式现阶段的目标）。理解软件开发的整个过程以及自己所扮演的角色对整个产品的成功至关重要。

传统的软件工程模型

很多模式都是用于开发软件的。有些模式已经存在了几十年，而新的模式则几乎每个月都在不断涌现。有些模式非常的正式和结构化，而有些却非常灵活。当然，没有任何一个单一的模式适合所有的软件开发部门，但是，通常来说遵循一些已有的模式可以帮助工程部门创造出更好的产品。理解了在产品开发周期的哪些阶段需要做哪些开发和测试可以帮助整个部门预期可能出现的问题，并且提前了解设计和质量问题可能会影响到产品的按时发布。

瀑布模式

瀑布模式是最常见（也是经常滥用）的软件开发模式之一。如图 3-1 所示，在“瀑布”这种软件开发方式中，每一个阶段的结束同时也是下一个阶段的开始。工作流程按照指定的顺序进行。工作的实现从一个阶段“流动”到另一个阶段，就像瀑布从山上流下。

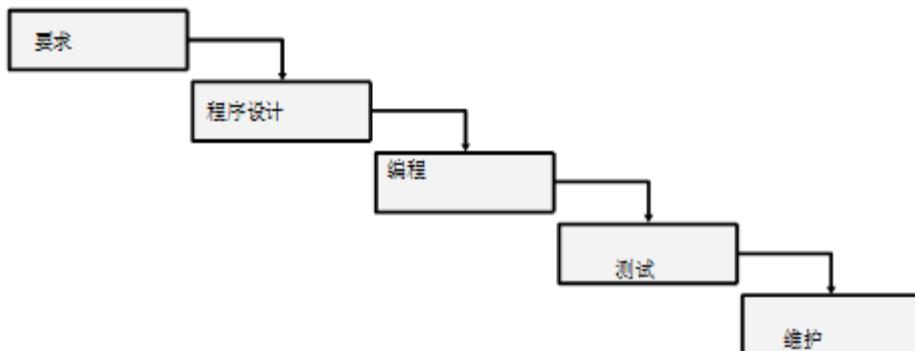


图 3-1 瀑布模式

这个模式的好处是，当你开始一个新的阶段的时候，前一阶段的所有工作都已完成。例如，在需求阶段的工作没有完成之前，设计阶段就根本不会开始。这个模式的另一个潜在好处是，它能够强制你在动手写程序以前尽可能多的进行思考和设计。照字面意义理解，瀑布模式不灵活，因为它看上去不允许阶段的重复。例如，如果测试的时候发现了一个由于设计缺陷而导致的 bug，你该怎么办呢？设计阶段已经“结束”了。这个明显不灵活性使瀑布模式饱受批评。因为每个阶段都有可能延迟整个产品周期，特别是对于长周期的产品，早期的某些设计很可能在实施时已经是可有可无的了。

有趣的是，瀑布模式的设计者，Winston Royce，本意是把它设计成一种迭代的过程。Royce 关于这个模式的原始论文⁸讨论了至少进行两次迭代，以及使用先前迭代中获得的信息改进之后迭代的必要性。瀑布模式的发明，是为了改进已沿用几十年的基于阶段的模式。其改进是通过对不同阶段之间反馈链的认识，和对减小返工影响提出的指导来实现的。可不管怎么说，现在瀑布模式已经在某种程度上被许多软件工程师所取笑，尤其是对于敏捷开发的支持者而言。在软件工程界的许多圈子中，“瀑布”已经成为用来描述“任何”具有严格流程要求的工程系统的通用术语。

⁸ Winston Royce, “Managing the Development of Large Software Systems,” *Proceedings of IEEE WESCON 26* (August 1970).

螺旋模式

1988 年，Barry Boehm 提出了软件开发的螺旋模式⁹。如图 3-2 所示，“螺旋”是一个包含四个主要阶段的迭代过程：确定目标、风险评估、工程实现、和下一迭代规划。

- 确定目标：为项目的当前阶段确认和设定特定的目标。
- 风险评估：确认主要风险、减少风险及应急计划。风险可能包括超支或资源问题。
- 工程实现：是完成工作的阶段，包括需求、设计、开发、测试，等等。
- 下一迭代规划：对项目评估，并开始计划下一轮的“螺旋”。

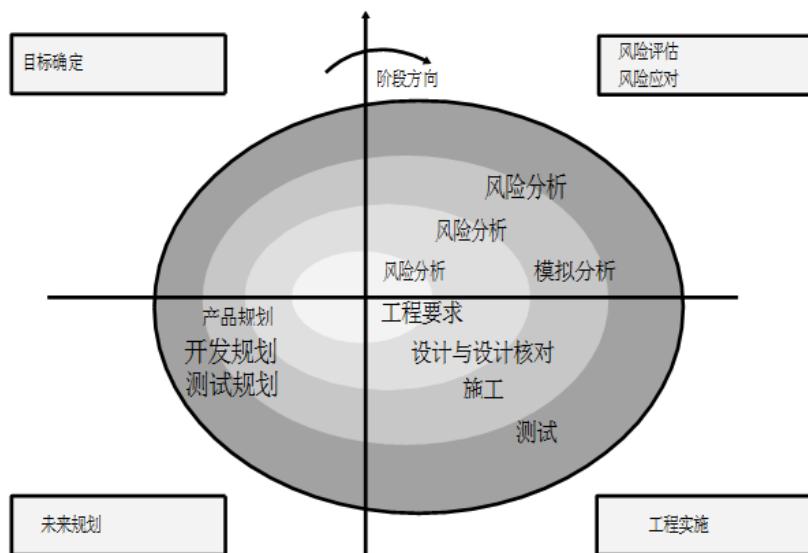


图 3 - 1 简化的螺旋模式

重复使用原型（prototype）来让风险最小化是螺旋模式的另一重要概念。初始模式构建于早期设计之上并接近最终产品的特性。在后续迭代过程中，原型可以帮助我们评估工程系统的优缺点和风险。

软件开发团队可以通过初始规划、设计、以及创建产品的原型版的方式来实现螺旋模式。接着他们收集用户对已完成部分的反馈，分析数据以评估风险，并决定下一步的工作。这个过程持续进行直到产品完成或者项目因风险过高而取消。

敏捷开发

在上一迭代成功或失败的基础之上，使用螺旋开发模式的团队能以迭代的方式开发软件。螺旋模式的规划和风险评估环节对于许多大型软件产品来说是至关重要的，但对其它很多软件项目来说却太过繁琐。有别于上述严格的开发模式（如瀑布模式），敏捷开发模式侧重于轻灵和小步递进的开发方法。

⁹ Barry Boehm, “A Spiral Model of Software Development,” *IEEE* 21, no. 5 (May 1988): 61–72.

敏捷开发当前在软件界非常流行。很多不同的开发模式都可以归类到敏捷开发，它们大致具有以下特征：

- 频繁而短暂的迭代：敏捷团队致力于快速发布软件，并且有做到这一点的纪录。
- 强调面对面的交流和协作：敏捷团队注重队员相互间和与顾客之间的交流。
- 对产品要求变化的适应力：敏捷团队灵活机动，他们擅于应对在产品开发周期中的任何环节顾客要求的变化。短暂迭代允许他们频繁地对变化区分优先次序和应对。
- 贯穿始终的质量责任制：在敏捷开发团队里，单元测试广泛地为开发人员所用。很多人使用 TDD (Test-driven Development, 即测试驱动的开发方法)。这是一种开发人员先写单元测试程序，然后再实现产品功能以通过该测试的开发方法。

在软件开发中，敏捷意味着团队能够快速地按需要调整前进的方向。以通过小改动而随时提供可用软件为目标，积少成多就能取得可观的成果，而且工程团队几乎总能知道产品的当前状态。与此相反，在非敏捷模式中，我记得一个项目曾经连续三个月“95%完成”。事后才发现：当时我们实际上不知道还余下多少工作量没完成，因为我们试图同时做所有的事，结果做了好几个月还是没有交出任何可用软件。敏捷的目标是每一次做一点，而不是一次全部完成。

其他方法

现在存在几十种软件开发模式，还有更多的模式及变体将会继续流行。没有哪一个模式是最好的，但是充分了解所选用的模式，并且在该模式的规范内开发软件，那么你开发出优质产品的机会就一定能更大。

里程碑模式

不清楚是否有意所为，我参与的大多数的微软产品都使用螺旋或其变体的开发模式¹⁰。当我加入微软的 Windows 95 团队时，他们是在“里程碑 8”（我们管它叫 M8）的初始阶段。就像它的一个前身 M6一样，M8 最后成了一个公共试用版（public beta）。每个里程碑都对产品功能和质量有具体的目标。在微软，每一个我直接或间接参与开发的产品都使用过里程碑模式。

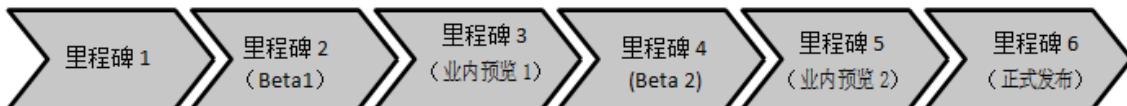


图 3 - 2 里程碑模式示例

里程碑进度计划建立了项目发布的时间表，也包含了关键的过渡安排和中期版本（例如 Beta 版和业内预览版）的时间表。里程碑进度计划帮助每个团队了解整个项目的期望值和现状。如图 3-3 所示是一个里程碑模式的例子。

里程碑模式的优点是里程碑不仅仅是写在日历上的一个日期。要完成一个里程碑，必须满足具体的、事先定义好的阶段结束标准(Exit Criteria)，下面简称阶段标准。阶段标准通常包括以下几条：

- 关键功能编程完毕 即使尚未完全测试，但关键功能已经实现
- 中期测试目标达到 如代码覆盖目标或测试完成率目标达成

¹⁰ 自从我 2005 年离开产品开发，加入卓越工程部后，很多微软开发团队开始接纳和使用敏捷开发模式。

- Bug(缺陷)目标达到 如无第一类严重(P1)Bug 或无“当机”的 Bug
- 非功能目标达到 如性能、负荷测试完成且无严重问题。

各阶段的标准随着每一个里程碑的推进而日趋严格，直到团队达到了最终发布产品的标准。表 3-1 是一个里程碑项目的各阶段标准示例。

测试范畴	里程碑1 (M1)	里程碑2 (M2)	里程碑3 (M3)	发布
用例运行		全部第一优先 (P1) 用例运行	全部第一和第二优先用例运行	所有用例运行
代码覆盖	代码覆盖率测量完成并生成报告	65% 覆盖率	75% 覆盖率	80% 覆盖率
压力测试	第一优先压力测试每晚运行	所有压力测试每晚在至少 200 台机器上运行	所有压力测试每晚在至少 500 台机器上运行且无新问题	所有压力测试每晚在至少 500 台机器上运行且无新问题
可靠性		解决 M 1 阶段中前 50% 的用户报告的“当机”问题	解决 M 2 阶段中前 60% 的用户报告的“当机”问题	解决 M 3 阶段中前 70% 的用户报告的“当机”问题
功能		产品 20% 拥有新用户界面	产品 50% 拥有新用户界面，且可用性测试完成	产品 100% 拥有新用户界面，且采纳可用性反馈意见
性能	完成性能测试计划，包括可扩展性目标	为所有主要客户场景确立性能基准	全面性能测试自动化完成，性能稳步改善	所有性能测试通过，性能目标达到

表 3-1 里程碑模式的阶段标准示例(不完全列表)

里程碑（或其它任何一种迭代模式）的另一个优点是，在每一个里程碑阶段中，团队都能通过一步步迈向发布而积累经验。他们学会怎么应对意外，如何为未达标的项目正确探寻解决方案，以及如何预估和处理缺陷出现率。还有一个目的是：每一里程碑发布的产品都是一个完整的产品，都可以用于大范围的试用（即使该里程碑发布的并不是外部测试版）。因为每个里程碑发布的都是软件产品的完整版本，所以产品开发团队或者微软的任何其他产品团队都可以以此为基础（尽管还很粗糙）进行后续的开发。

质量里程碑(MQ)

几年前，我来到处于产品开发周期正当中的一个团队。我参与每天的 bug 诊断会。Bug 经审查通过后，或指派(assign)或推迟处理(Postpone)。推迟处理有多种原因，也是软件开发中不可避免的。发布前几个月的一次会议散会前，我们还有些时间，我就问我们是否能够快速的看一下推迟到下一版本处理的 bug。这些 bug 的数量大到可以用“bug 潮”来形容——意味着现有产品发布以后，新版本的开发工作将背负着大量积压的 bug 而开始。

Bug 的积压，不完整的文档和“将来某天”必须修补的不可靠的测试，全都可以计入“技术负债¹¹”。在软件开发中，我们经常不得不妥协。许多妥协最终成了“技术负债”。技术负债很难对付，我们不理它的话，它不会自己走开。所以，我们必须做点什么。通常我们尝试在干别的某件事情或是在日程表的某个少有的空闲时间试图解决它。这样做的效率如同杯水车薪。

¹¹ Matthew Heusser 在他的博克中常写有关技术负债的问题 (xndev.blogspot.com). Matt 暂时还没有为微软工作。.

很多微软团队处理技术负债的另一个方法是引入“质量里程碑”(MQ)。这个里程碑阶段存在于产品发布以后，下一版本产品开发开始之前，它为开发团队提供了一个修补 bug，重新设计测试方法以及修补那些在前一个开发中被搁置的问题的机会。MQ 也是一个改进工程系统、开发初始模式和诞生新点子的好机会。

积压的 bug 清理干净了，测试工具到位了，工程系统改进了，还有其他所有在上一发布周期中烦人的问题也解决了，然后开始新的产品周期——这样才是开发一个成熟产品的正确途径。

敏捷开发在微软

敏捷方法在微软很流行。一个内部的专门讨论敏捷方法的电子邮件组有超过 1500 人。一份对超过 3000 名测试和开发工程师进行的问卷调查表明，大约有三分之一的人回答他们使用了某种形式的敏捷开发模式¹²。

● 功能小组

多数的敏捷开发专家认为不超过 10 人的团队是最合适的。这点对于动辄几千或更多人的大型开发团队比较困难。对于这个把敏捷方法推广到大型团队的问题，微软的解决方式就是使用功能小组。

功能小组是一个小型的、跨职种的小组，由 3 至 10 个来自不同职种部门（通常包括开发、测试和项目管理）的人组成。这个小组自主地从头至尾负责整个系统中的一个功能块的实现。典型的小组结构是一个项目经理，三到五个测试人员，三到五个开发人员。他们协同工作，用较短的周期设计、实现、测试和整合该功能到整个产品中。如图 3-4 所示。

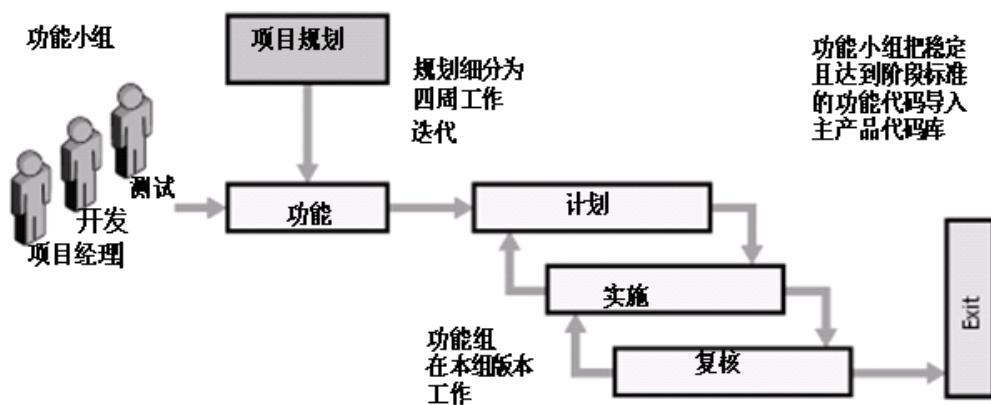


图 3 - 3 功能小组模式

这个团队的关键元素是：

- 足够独立，甚至有团队自己的方式方法。

¹² Nachiappan Nagappan and Andrew Begel, "Usage and Perceptions of Agile Software Development in an Industrial Context: An Exploratory Study," 2007, <http://csdl2.computer.org/persagen/DLabsToc.jsp?resourcePath=/dl/proceedings/&toc=comp/proceedings/esem/2007/2886/00/2886toc.xml&DOI=10.1109/ESEM.2007.85>.

- 可以从定义、开发、测试和整合等方面来全方位推动一个产品模块，直到能直接向顾客展示其价值

Office 和 Windows 部门的所有团队都采用这种方式来激发更多的责任感，更好的自主性 而同时又能有效管理整个产品的进度。Office 2007 的项目下有超过 3000 个功能小组。

● 完工

为了在每个周期结束的时候能交出高质量的功能组件，功能小组需要认真定义什么是“完工”，并且按此要求交货。最通常的做法是为团队定义质量门槛以保证功能组件的完备性，并确保功能整合时尽量不会出问题。质量门槛类似于里程碑的阶段标准——严格甚至挑剔，并且往往需要做大量的工作才能达到。

表3- 1 功能小组质量门槛示例¹³

质量门槛	说明
测试	所有计划过的自动测试和手工测试完成且通过
功能 Bug 已关闭	功能方面的所有已知的 bug 已全部修复或关闭
性能	新功能已达到产品的性能要求
测试计划	测试计划书完成，为所有计划的自动和手工测试提供说明
代码审查	所有新代码都经过审查以确保符合代码设计准则
功能说明	功能说明书完成并经过功能小组批准
文档计划	功能的文档计划已启动
安全	该功能的安全威胁建模已制定，减低可能的安全风险
代码覆盖	新代码单元测试完成，在新功能代码上保证 80% 覆盖率
本地化	新功能经验证可在多种语言版本下工作

当产品缺陷问题仍需要解决时，功能小组编写必要的代码、编译后发布内部私用版本（private build）、然后对其加以测试，并且重复此过程，直到解决其缺陷问题。当该小组能够达到的质量门槛的期望目标时，他们把此时的代码提交并合并到软件产品的主要源代码库（main product source branch）中，然后开始进入下一个功能的开发。I.M. Wright 的《代码之道》（微软出版社，2008 年）一书对微软的功能小组有更多的讨论。

敏捷迭代和里程碑

敏捷迭代并不完全取代在微软盛行的里程碑模式。敏捷实践和里程碑模式是携手合作的 -- 在大型产品团队，里程碑是确保所有团队可以将其共同创造的功能整合，以创建一个软件产品的完好机会。虽然敏捷团队的目标是在任何时候可以发布产品（或特性），但多数微软里程碑团队每隔几个月才发布给 Beta 版用户和其他早期试用者。Beta 版和其他的早期发布几乎总是与产品的里程碑计划相一致。

¹³ 此列表取自 Ade Miller 和 Eric Carter, “Agile and the Inconceivably Large”, IEEE (2007)

宏观视野

在微观层面，从开发人员输出的最小单位是代码(code)。代码转化成函数(functionality)，函数转化成产品功能(feature)。(在这个过程中的某点上，测试参与进来，成为流程的一部分，以提供高质量的程序和功能。)

在许多情况下，汇集一个很大集合的功能成为一个项目(project)。每个项目的开发进程有一个特定的开始、结束以及(里程碑的)阶段检查、使用场景、角色和许多其他细节。最后，在最高层次，其后要发布的有关项目成为一条产品线。例如，微软 Windows 是一条产品线，Windows Vista 操作系统是在该产品线内的一个项目，而数以百计的功能组成这一项目。

如图 3-5 所示，调度(scheduling)和计划(planning)发生在每一层面的输出，但有不同的涵义。在产品(product)层面，计划基本上是根据长期战略和业务需求而作的。另一方面，在功能(feature)层面，计划几乎纯粹是战术——目标是让所做的工作越来越有效率。把工作归入战略或战术并不重要，但关键是要把战略和执行整合成为宏观计划。

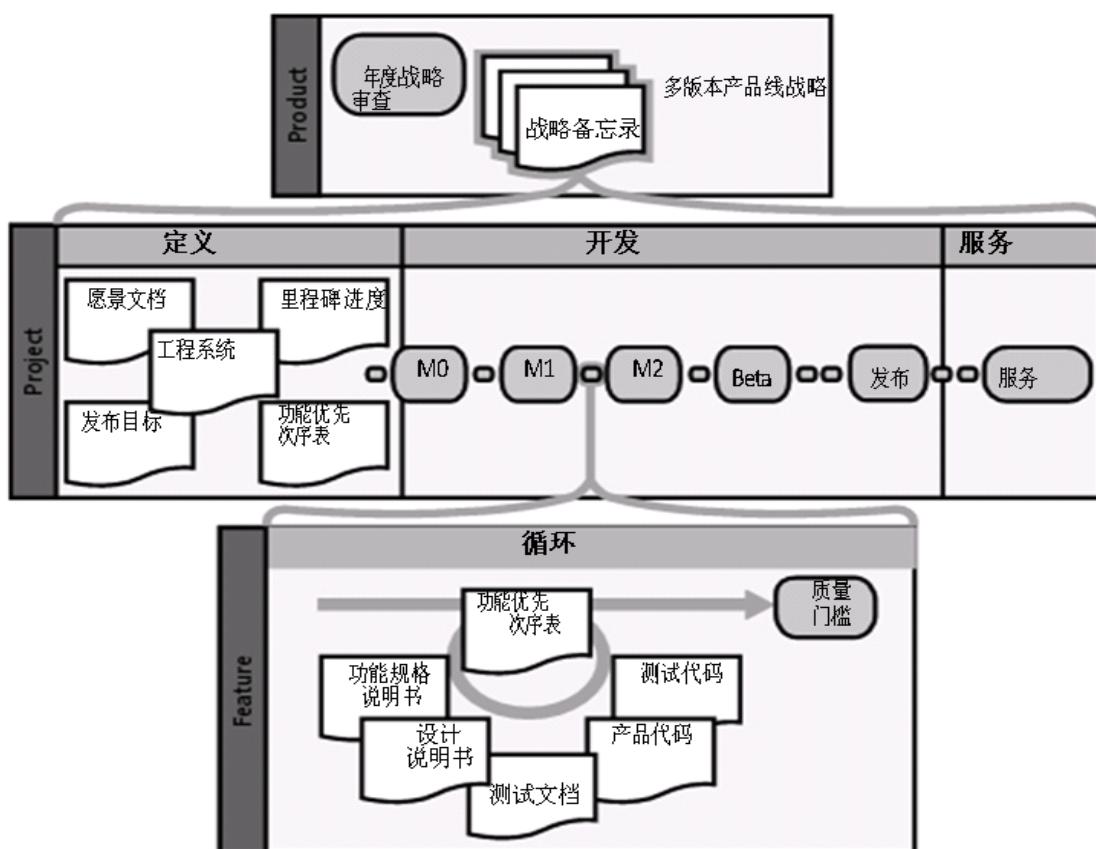


图 3-5：软件生命周期的工作流程

流程改进

在我认真对待的事情上，我总是想持续地加以改进。无论是我正准备的一顿饭，还是我的足球技能，或是练习的单簧管奏鸣曲，我都要做得更好。良好的软件团队具有相同的目标——常常反映在他们总是想着如何持续改进他们在做的事情。

美国爱德华兹·戴明(W. Edwards Deming)博士在质量和流程改进的工作得到广泛承认。他对质量改进最著名的贡献之一是简单的计划(Plan)、执行(Do)、检查(Check/Study)、处理(Act)循环。有时被称为修哈特循环(Shewhart cycle)，或PDCA循环。以下PDCA循环各阶段在图3-6表明：

- Plan：事先计划，分析、确定过程和预测结果。
- Do：执行计划和过程。
- Check/Study：分析结果（值得注意的是，为了更清楚，戴明后来把此阶段改名为“学习”）。
- Act：审查所有步骤，并采取行动改善这一进程。

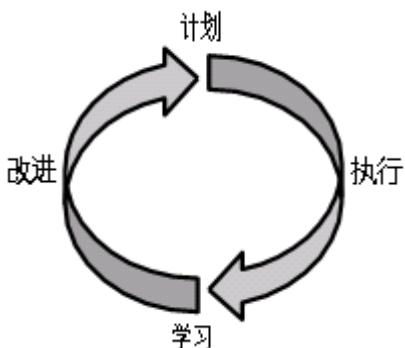


图3-6：戴明的PDCA循环周期

对于很多人来说，似乎这循环是这么简单，以至于他们认为它不比普通常识判断的结果多很多。但无论如何，它的简单性使它成为了一个强有力的模式。事实上该模式是六西格玛(Six Sigma)DMAIC(定义、度量、分析、改进、控制)模式，ADDIE(分析、设计、开发、实施、评价)教学设计模式，和许多其他不同的行业改进模式的基础。

软件领域里可以找到应用这一流程改进模式的无数例子。例如，考虑一个团队：他们意识到在最后一个里程碑测试中找到的许多bug，在代码审查中本来也是可以发现的。于是该团队的做法是：

- 1) 首先，计划一个代码审查的流程——也许需要同事之间审查所有代码的变化。他们还可能进行一些深入的分析，并拿出一个通过代码审查有多少bug可以在之前的里程碑代码审查过程中被发现的准确衡量。
- 2) 在下一个里程碑进行代码审查。
- 3) 在下一个里程碑期间，该团队监视有关的缺陷(bug)度量。
- 4) 最后，他们审查整个过程，质量度量和结果，并确定它们是否需要进行任何变更，以改善整体进程。

微软的正规流程改进系统

流程改进项目被广泛使用在软件产业中。ISO 9000 国际标准、六西格玛、能力成熟度模式集成（CMMI 模式），精益六西格玛和其他许多举措都存在帮助组织改善和满足新的目标和目的。所有不同的举措法侧重于流程改进，但具体细节和执行略有不同。表 3-3 简要介绍了一些这种举措。

流程/方法	概念特点
ISO 9000	该体系通过满足质量要求、监测流程和实现持续改进来获得客户满意。
六西格玛	由摩托罗拉开发。利用统计工具和 DMAIC（定义、衡量、分析、实施、控制）的流程来衡量和改进流程
CMMI	五个级别成熟度模式，侧重于项目管理、软件工程和流程管理实践。 CMMI 模式的重点是组织而不是项目
精益六西格玛	重点在于从工程的流程中消除浪费（例如，缺陷、延迟和不必要的工作）

表 3-1：正规流程改进系统比较

尽管微软还没有全心全意地推广普遍使用任何这类流程改进举措或方法，但实际上流程改进的实践（无论是正式的或随意的）在微软是随处可见的。微软继续认真考虑流程改进的项目，而且常常会认真地“尝试”某方法，以更好地了解该方法在微软的产品实践中该如何应用。例如，在过去几年中，微软已经在若干项目中试行了基于六西格玛和精益六西格玛流程改进的实践。使用并充分发挥这些流程改进方法的优点，这样的战略目的是要了解如何最好地在快出成果的愿望与有严格步骤要求的六西格玛或精益六西格玛方法之间取得平衡。

微软和 ISO9000

拥有 ISO9000 认证的公司必须经审核确认他们的工作流程和他们对该流程的贯彻执行是符合 ISO 标准的。这个认证可以给产品用户带来一定的安全感或信心，因为知道该产品的质量控制流程是开发流程中不可或缺的。

有客户问过微软对 ISO 质量标准的符合程度的问题，因为他们想大致知道，在微软开发的产品是否能满足 ISO 质量标准。我们对这种问题的回应是：我们的开发流程、记录流程所有步骤的文档、管理团队对质量流程的支持、还有在建文档和可重复流程方面（包括档案结果）的制度化，所有这些都是 ISO 核心标准的元素。在大多数情况下，我们达到或超过了这些标准。

当然，这并不是说微软不重视 ISO9000，也不是说微软以后都不会拥有 ISO9000 认证的产品；我的意思是，在写这本书的时候，在大多数情形，我们认为我们的流程和标准符合我们的工程师和客户的需要，也符合 ISO9000 的要求。不过，什么都可能变，也许下个星期回答就不一样了。

从“作战室”发布软件

不管是产品周期较短的 Web 服务,还是需要几年产品周期的 Windows 或 Office,到某一个时间,这些软件总是要发布给顾客使用。这些决定产品是否能够发布的决策,以及那些为确保产品在正确轨道上而做出的决定和分析,它们都是在作战室 (*War Room*) 或发布室(*Ship Room*)里做出。作战室小组的运作贯穿于整个产品周期,他们同时也扮演着发布质量监督委员会的角色。“作战室小组“这个名字已经存在很多年了,它描述在会议中所发生的事:“在对立力量或原则之间的冲突”。

作为每天为产品做出决定的组织,作战室小组需要对整个产品的所有组成部分和系统有一个全面的视角。确定哪些 bug 需要修正,哪些功能需要裁掉,团队的哪个部分需要更多资源,或者是否改变发布日期,作战室小组需要做出这些很重要并且能造成严重潜在后果的决定。

一般作战室小组由产品团队测试、开发和项目管理等部门出一个代表(通常是经理)组成。如果代表不能出席,那人就提名他所在队伍中的其他人参加,以便能做出一致决定和集体通过,特别是对于大家认为要记录在案的项目。

作战室小组开会频率可能从发布周期早期时的每周一次逐步增加到每天一次,在接近发布日期时甚至会每天二到三次。

作战, 它到底有什么好处?

作战室是产品队伍的脉搏。如果作战室小组有效,大家就可以集中精力完成正确的工作,并且理解决定为什么做和怎样做。如果作战室小组缺乏组织或没有效率,队伍的脉搏也会很弱,从而导致由于缺乏方向和领导无方而带来的诸多问题。

对一个成功的作战室小组和作战室会议来说以下几点是需要考虑的:

- 保证适当的人在屋子里。代表缺席不好,但人太多会同样糟糕。
- 不要设法在会议中解决每个问题。如果遇到问题需要更多的调查,则指定某人跟踪进度,然后继续。
- 清楚地辨认行动项目、责任人和到期日。
- 清楚地跟踪问题,并且前后一致地处理问题。时间长了,人们就会逐渐熟悉处理过程并且更有准备。
- 要明确什么是你所想要的。大多数发布室都是焦点集中而干脆。有些人想要更多的合作。确定大家有共识。如果您想要它简短舒畅,不要讨论设计问题,如果是非正式的,尽量不要打断别人。
- 集中在事实而不是猜想上。“我认为”,“它可能”,“它或许”这些词语都是有问题应该警觉的。要么有,要么没有,没有中间状态。
- 每个人的意见都是重要的。在许多作战室都会听见的一句话是“不要听河马(HiPPO)的” - 河马是一个首字母缩略词,表示工资最高的人的观点 (*Highest Paid Person's Opinion*)。
- 在里程碑初期事先设置阶段目标,并且遵照执行。应当为质量目标设定期望值。

- 只有一个人主持会议并且让它按部就班进行。
- 气氛可以轻松活跃一些。

定义发布 – **Microspeak**

在发布小组会议上所使用的许多术语也许会把一个旁观者搞糊涂。随机词组和三个字母的缩略词(TLAs)在交谈中随处可见。

某些最常用的术语包括：

- **LKG(Last Known Good)**: 已知的最近一个符合特定质量标准的版本。通常这和下面的自用版类似。
- **Self-host (自用版)**: 质量足够稳定的预发行版，产品团队将其使用到自己的日常工作中。比如 **Windows** 团队在产品周期中就一直用内部的预发行版的 **Windows**。
- **Self-toast (自糊版)**: 这是已被完全破坏，或者“烤糊”您做日常工作的能力的版本。亦称做 **Self-hosed**(自己失手破坏的)。
- **Self-test(自检版)**: 该版本对多数测试运行很好，但有一个或多个必须解决的问题，因而不能达到自用版的状态。
- **Visual freeze (界面冻结版)**: 在产品开发周期的某一点或里程碑，视觉效果和用户界面变动被锁定、且不会在发行之前改变的版本。
- **Debug/checked build (调试版)**: 促进调试和测试的功能被激活的版本。
- **Retail/free build (销售版)**: 为发行而优化的版本。
- **Alpha 版**: 一个很早期的版本。其目的是为了能得到功能和可用性的初步反馈。
- **Beta 版**: 被送到顾客和伙伴的产品的预发布版本，以得到评测和反馈。

强制性惯例

微软行政管理不硬性规定部门、小组或者团队应怎样开发和测试软件。团队可以自由试验，也可以使用成熟可靠的技术，或者兼而有之。他们也可以根据实际情况自由决定在团队或部门一级的强制性要求。比如，**Office** 软件就有几个标准，任何一个 **Office** 组件都必须满足它们才能发布，但那些标准对于一个发布 **Web** 服务的小团队可能就没有必要。开发过程中的自由权使得团队能在产品开发中创新和做出他们自己的选择。然而，还是有为数不多的惯例和政策需要微软的每个团队必须遵守。

这些强制性要求与软件发布细节没有多少相关。这些政策是要确保一个产品在发布之前必须完成几个重要的步骤。

虽说强制性的工程要求并不多，但不符合这些要求的产品是不允许发布的。强制性政策所包括的领域举例来说包括对隐私问题做计划，对第三方组件的授权、地域政治评审、病毒扫描和安全审核等。

可预期性与强制性

强制性惯例，如果不是一致地和系统地执行，只会对顾客和微软造成无法接受的风险。

可预期地执行惯例是每个产品组应该使用的有效的方式(除非存在技术局限)。最典型例子

是使用静态分析工具。(参见第 11 章，“非功能性测试”), 例如, 当我们最开始开发 C# 的时候, 我们没有这种语言的静态代码分析工具。但语言发布没多久我们的团队就开发了 C# 的静态代码分析工具。

一站式(One Stop)购物

通常, 在产品团队中有一人负责发布管理。这个人的职责包括确定所有的硬性惯例都满足了。为了保证大家都了解硬性惯例并且贯彻到工作中, 每项惯例的要求及其相关的工具和详细说明, 都放在一个内部的网络门户, 以便微软能尽量减少硬性惯例的数量, 并给团队提供一致的工具使他们尽可能容易地达到要求。

本章小结：把饭做完

就像做饭一样, 做软件需要考虑很多问题, 尤其是当膳食 (或软件)的大小和复杂性越来越高时。再试想一下如果要设计整周的菜单, 或发行一个软件程序的多个版本, 需要考虑的因素就会变得更多。

思考一下软件是怎样开发出来的可以帮助我们很好的理解整个过程——软件工程团队准备了一碗“应用程序汤”, 那么我们应该何时、何地把哪些“软件佐料”加进去。一个计划、食谱或者菜单在许多情况下是有用的, 但就像艾森豪威尔 (Eisenhower) 说的, “在战斗开始时, 我总发现那些战前计划是无用的, 但计划是不可少的”。要记住的一点是花一些时间把所有事理一遍, 从实现细节到产品的愿景等, 这样可以帮助实现结果。没有所谓 **最好的** 做软件的方式, 但是有很多 **好的** 方式。我所工作过的那些优秀团队虽然也关心实际用的是什么开发流程, 但更加在意的是如何成功地执行所选的流程。

第二部分 关于测试

第 4 章 软件测试用例设计的实用方法

阿伦·培智

当你在为一个初始版本发布以后还会被运行很久的软件设计测试用例时，下很大的工夫来设计持久有效的测试用例是非常重要的。很多微软的应用程序都有长达十年的技术支持计划。在测试用例被设计和实现以后，它们会持续不断地运行直到产品发布，但是那并不是这些测试用例寿命的终结点。在一个应用程序发布给终端用户之后，源代码的所有权、文档、测试工具、测试自动化系统，和所有其他相关的附属材料会转给一个独立的团队，或者现产品团队的一个下属团队。他们负责所有必要的后续改动工作。由这些可持续工程团队做的改动包括安全补丁，快速修补工程（QFE-quick fix engineering）和服务包（service pack）开发。为了一个产品的某个发布而创建的测试会在整个客服周期中运行几千几万次。几乎每一个测试团队在设计测试用例时都知道他们创建的这些测试会运行很多年。

在微软因为对长期支持性的强调，我们广泛使用自动化测试。但是这并不意味着我们不重视或者不进行手工测试。一个好的测试策略能辨识在哪些领域自动化是可行的，在哪些领域是不可行的，并且决定一个恰当的测试方法。在第十章“测试自动化”中将具体讨论自动化的测试方法。

设计是一个在开始具体实现解决方案以前，系统化地思考或计划这个解决方案的行为。一个建筑师会认真仔细地计划和设计，以确保其大楼设计满足住户需要。认真仔细地计划和设计可以提高测试在其整个生命周期中的价值。这一章我们将讨论测试设计的基本原理。

提示：微软 Office 2007 就有过上百万的测试用例。

实践良好的软件设计和测试设计

设计可能是软件开发过程中最重要的一步。软件设计包括制定计划和解决难题。它包括对用户体验的预见和对解决方案以及备选方案的重要分析。一个设计良好的软件能预见很多可能发生的问题，而设计是创建能很好为用户服务的软件中至关重要的一步。优秀的设计并不总是要求 BDUF- Big Design Up Front（译者注：一种在着手进行程序代码的撰写之前，就先按照既定的程序分析、设计、制图、撰写文件等等耗时费力的工作方式）。运用敏捷开发（也就是代码也被视为设计）的软件工程，同样要求计划和预见，也同样可以被很好地设计。任何产品，无论它是软件程序还是厨房用具，都需要良好的设计以免让用户感到混乱和烦躁。如果没有完全充分地重视设计，用户必定会在使用过程中遇到问题。

测试设计和好的软件设计有很多相似处，甚至笼统地说，和好的设计都有相似之处。测试设计需要从计划和问题解决方面来决定做哪些测试，以及哪种测试在验证功能和确认错误路径处理得当上是最有效的。测试设计中最重要的方面之一是能预见用户的需要和期望，然后创建能够恰当地处理这些需要的测试。良好的测试设计通常从对软件设计的审查或批评开始。通常，对待设计审查和对待代码审查是很相似的，也就是设计者对设计进行解释，参与者提

出问题并提供反馈。一个好的设计审查会对所有主要的设计决策中的各种备选方案作深度比较。比较的目的是要就建立什么、怎样去建立、还有更重要的，如何去测试它们等方面，达成共识。良好的设计和良好的执行在成功的软件开发工程中占举足轻重的地位。

使用测试模式

设计模式，也就是对软件设计中的常见问题的解决方法，在软件开发中是很热门的。它们提供了可以在很多不同情况下运用的方针和策略。最重要的是，它们为程序员交流解决方案提供了一种统一语言。

测试模式的概念已经存在很多年了。在“*Testing Object-Oriented Systems: Models, Patterns, and Tools*”一书中，Robert Binder¹⁴收录了 37 种测试设计模式和另外 17 种针对自动化测试的模式。2001 年，Brian Marick，“*The Craft of Software Testing*”的作者，引导了多个软件测试模式（PoST: Patterns of Software Testing）的研讨会，在定义测试模式的种类和使用上取得更多进展。

如同设计模式一样，测试模式解决了测试工程师在设计测试上的很多普遍问题并提供了有针对性的方针和策略。有一些测试模式是结构测试的方法，一些是启发性的方法，还有一些模式是各种想法的组合或是完全不同类的东西。测试模式的重要性在于测试工作人员可以用它来交流一个测试方法的内容，而且可以用一种可以被理解和执行的方法来共享不同的测试设计技术。

测试模式共享的常用形式是模板。Robert Binder 的测试设计模板包括了 10 个不同的属性。微软内对测试设计模式感兴趣的测试工程师通常使用一种基于他的模板的简易模板。这个模板包括以下几个属性：

- 名称 提供一个容易记的名字——在对话中可以被谈及
- 问题 提供一句话来描述这个模式所能解决的问题
- 分析 描述问题的领域（或者提供一小段对问题领域的描述）并回答这个技术如何比简单地随意选择式的方法要好。
- 设计 解释这个模式如何被执行（模式如何从设计转变为测试用例）
- 预言 解释期望的结果（也可以包括在设计这一部分中）
- 用例 列出这个模式如何发现程序缺陷的例子
- 缺陷和局限 解释在什么环境和情况下，应避免使用这个模式
- 相关的模式 列出任何相关的模式（如果有的话）

这个特定的模板给创建不同类型模式提供了充足的灵活性，又提供了足够多的信息，从而方便使用模式进行测试设计的测试工程师之间的交流。表 4-1 是一个大家熟知的测试设计方法使用这个模板的例子。

表 4-1 边界值分析测试模式

名称	边界值分析法(BVA)
问题	软件中的很多错误发生在数据域的边界。比如，使用 > 而不是 >=，或者相差为一的索

¹⁴ Robert V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools* (Indianapolis, IN: Addison-Wesley, 1999).

	引错误（是以 0 为基准的索引，还是以 1 为基准的索引）
分析	根据很多问题都倾向于集中在输入极值附近的基本原理，选择变量输入域在边界或接近边界的测试用例。在安全测试中一个经典的例子是创建一个很长的字符串作为输入来探测可能的缓冲区溢出。更普遍的是，在边界情况下不安全的行为经常是程序员不能预见的。他们更倾向于专注在表面上的情况。
设计	对于每一个输入量，确定允许的最小和最大值(min and max)。设计一套测试用例来测试最小值，最大值，最小值减 1，最大值加 1。(注意 BVA 有时候被定义为包括最小值加 1 和最大值减 1)。 测试用例应该包括： <ul style="list-style-type: none">• 组件的输入量• 要被执行的分区边界• 测试用例的期望输出
预言	最小值和最大值应该通过。在这个范围以外的值应该失败并有对失败情况的恰当处理
举例	对允许范围是 1 到 10 之间的一个数字的输入域，测试 0, 1, 10 和 11
缺陷或局限	边界并不总是明确的。需要有对被测产品相关领域的知识或者参照源代码才能做有用边界值分析。如果输入域包含特殊的值(指在允许范围内的值，但是应用程序处理它们的方法不一样)， BVA 可能会错过和这些值相关的问题。
相关模式	等价类划分法

基于模式的方法对于测试设计来说，是一个有用的交流测试想法的系统，它还可以加速测试设计进程。它也是一种让新来的测试工程师学习关于测试设计、让有经验的测试工程师分享想法、让测试设计的整个知识库从一个组织传递到另一个组织的简单方法。

估计测试时间

"测试需要多少时间？"这个问题对于任何测试团队来说都可能是一个不太容易回答的问题。为了给出一个准确的答案，花时间思考和计划是非常必要的。我看见过一些不成功的团队只是简单地在产品周期的末端加上几个星期的“缓冲”或是“稳定”时间。就像你可以预料到的那样，用这种方式计划的工程很难达到用户的期望。正确地估计测试任务至少和写软件功能一样重要，而且应该得到同等的重视。

那怎样才能估计一个功能或是应用程序的测试需要多长时间呢？我看到的一个经常被使用的实用方法是拷贝开发时间。比如，如果某个开发任务计划需要一个人工作两星期，那么估

计写自动化测试和描述手工测试用例也需要一个人工作两个星期会很有用。这个方法通常是对的，可是在实际实践中这只是一个出发点，因为有太多的因素可以影响到测试的任务。产品部门的目标、用户的期望、测试组的技术能力，以及工程的复杂度都影响着测试的进程，所以每个测试人员在估计测试时间的时候都应该把这些因素考虑在内。表 4-2 列举了在测试时间估计中应该要考虑的一些因素的例子。

表 4-2 测试时间估计的因素

属性	如何考虑这个属性
历史数据	至少你可以基于以前的项目来估计测试设计。
复杂度	复杂度直接和可测试度相关。测试简单的应用程序比测试复杂的应用程序可以快得多。
商业目标	这个应用程序是一个原型系统还是一个示范程序？或者这个是宇宙飞船的飞行控制软件吗？商业目标影响着测试要花的功夫的广度和深度。
一致性和服从性	如果一个应用程序必须服从某个标准，在估计测试任务的时候就必须考虑到这些要求。

从测试开始

当 15 年前我刚开始软件测试工程师生涯的时候，我经常听到项目经理说，“为什么我们需要雇佣更多的测试工程师？我们还需要至少一个月才能完成代码”。我很高兴那是很久以前的事了，可是在某些情况下，如果我们有幸参与产品开发的初始阶段，我们测试人员经常会争论该做什么正确的事情。

测试设计的一个起点可以是对软件的功能规格和需求的审查。如果已经具备了良好的产品功能需求，基于需求的测试会是一个不错的出发点。如果没有产品需求（或者是写的不太好），那么最好的测试设计的出发点就是问问题。问一下这个软件应该如何工作。问一下这个软件如何处理数据。问一下这个软件如何处理错误。问与软件相关的问题可以帮助测试人员在代码还没写以前就在设计上有好的开始。

问问题

如果代码已经存在，但是需求或者功能规格并没有到位，最好的开始测试设计的方法就是运行这个应用程序。问问自己这个程序是如何工作的。你也许可以回答这些问题，或者它们会引导你问更多的问题。如果有什么东西让你很困惑，问问题。如果源代码已经存在，可以看看代码，而且如果需要的话，问更多的问题。探索性测试，也就是测试和设计测试同时进行，可以很大程度地影响测试设计而且对于整个测试设计过程也是很有利的一部分。

用调试器（debugger）进行探索式测试

当我第一次测试一个组件或者功能，而且我有源代码的时候，我经常用调试器来测试。在我写测试用例时，我先写一些基本的测试程序。它们可能是自动化的测试程序，也可能只是一些我随手写在便笺本上的点子。我在组件初始化的某个地方设置一个断点，然后用调试器来理解每条代码路径是如何被执行到的。我留心在什么地方边界条件需要被测试，外部数据又在哪里被用到。我通常花好几个小时（有时候是好几天）来探索（和学习、执行）直到我感觉我对这个组件有很好的理解。到那时候，我对于怎样创建一套针对这个组件的有效的并且能在这个产品生命周期中都被使用的测试有了很好的想法。

如果你在测试一个你以前从来没有测试过的功能或者应用程序类型，甚至如果你在测试一些你所熟悉的东西的过程中遇到难题而停滞不前了，咨询其他刚测试过类似的东西的测试人员的想法是很有用的。很多在测试过程中被错过了的软件缺陷都是因为测试人员没有问足够多的问题或者他们没有问对问题。测试设计需要对被测试的软件进行广泛的、仔细的察看检查。问问题是获得所需要的知识以进行检查的最好的方法之一。

要有好的测试策略

一个测试策略引导着测试设计，而且可以为测试团队的测试设计提供方向。一个好的测试策略为团队提供了愿景，帮助每个人决定什么测试活动是最重要的，而且帮助他们决定在什么时候和在什么地方去应用不同种类的测试。

这个策略包括测试的种类、进程，和测试时测试团队会采用什么方法。它包括对风险的预估以帮助团队决定错误最有可能发生在哪或者某些组件是否可能需要更加全面彻底的测试。这个策略通常包括对测试团队培训和教育的计划。教育的策略可以包括会议，研讨会或者是由顾问领导的培训，或者包括测试团队同事之间的信息共享。无论培训是怎样进行的，一个一流的测试策略要包括测试团队提高技术和知识水平的计划。表 4-3 是测试策略的属性的几个例子：

表 4-3 测试策略属性的实例

属性	如何考虑这个属性
介绍	提供这个策略的概貌，描述这个策略如何被使用。策略应该基于项目的功能和质量目标。
需求规格	列出测试团队的文档计划，还有来自其他工程学科对文档的期望。
关键情景	什么是主要的驱动测试努力的客户使用场景？这部分回答了这一问题，并把测试所花的功夫和产品计划联系起来。
测试方法	这部分描述了将被使用来测试产品的测试方法和使用某些方法会带来的收益和风险。代码覆盖率，测试自动化，测试用例管理和其他的方法，工具也可以包括在这个部分里。
可交付使用的测试	什么是测试团队声明的期望？可交付使用的例子可以包括以下这几条的状态： <ul style="list-style-type: none">• 测试结果• 代码覆盖率

	<ul style="list-style-type: none"> • 规格说明完成的状况 • 缺陷的比率和趋势 • 使用场景的性能测试结果
培训	如果策略的成功需要对测试人员的培训,那么这些需求应该在这里描述,包括培训如何帮助策略成功的分析。

思考可测试性

为了影响产品和帮助设计,另外一件测试人员可以而且应该做的事情是早点思考可测试性。可测试性是指软件可以被完全有效测试的程度。选择不同的设计,选择简单的算法,使用测试“钩子(hook)”(仅仅为了测试方便而写的额外的功能)和让内部变量可见都是如何提高可测试性的例子。

一个测试人员可以用来提高可测试性的最普遍的方法就是在需求或设计评审中简单地问,“我们怎么来测试这个东西?”这不仅是用来澄清有模糊陈述的绝妙方法,同时随着时间的推移,它将开发可被测试软件的这个想法植入开发人员的头脑中。对那些已经将写单元测试的活囊括到日常工作中的开发团队来说,他们已经开始理解可测试性的重要性。但是请记住:可测试性的问题影响的范围远远超过小团体而且必须在产品的各个级别上都要考虑到。表4-4定义了SOCK这个缩写词,它是一种提高软件可测试性的简单模式

表4-4一个简单的可测试性的模式:SOCK

术语	定义
简单(Simple)	简单的组件和应用程序测试起来更容易(而且花费也更低)。
可见(Observable)	内部结构和数据的可见性可以让测试程序准确地决定测试是否通过。
控制(Control)	如果一个应用程序有阈值,能够设置和重新设置那些阈值可以简化测试。
知识(Knowledge)	通过参考文档(规格,帮助文件等等),测试人员可以确保结果是正确的。

如何测试上百个调制解调器?

在测试Microsoft Windows NT远程访问服务器(RAS)时,我们曾经需要利用有限的硬件资源对调制解调器的拨号服务器进行可伸缩性测试。我们碰到了一个可测性的问题——为了能准确地模仿真实的用户部署,我们需要测试上百台调制解调器同时连接拨号服务器的情况,而现有的资金和实验室基础设施仅可供测试十几个调制解调器。测试团队想出了一个办法,用软件来模拟调制解调器并将其与以太网相连。我们称之为RASETHER。这个测试工具最终被证明了一个很好的想法。因为它是人们第一次在一个网络里创建另一个专有网络。如今,这个技术被称为虚拟专用网络或者VPN。起初为了Windows NT调制解调器的服务器设计的可伸缩性测试的测试工具变成了一个巨大的商业成功并且成了用户进入公司网络的重要工具。

— David Catlett, 测试架构工程师

测试设计规格说明

讨论可测试性会促使大家考虑如何对软件的某个部分进行测试。这也会要求测试工程师去考虑测试用例的设计。设计测试的过程通常至少和设计终端用户软件的行为一样重要。测试工程师经常需要写一些正规的测试设计文档用来描述测试的策略和方法。一个测试设计规格说明（TDS）一般可适用于手动和自动化测试。测试设计规格说明也应通过评审过程，这与软件工程中的功能规格说明和设计文档是一样的。因为测试设计规格说明描述了测试过程中的方法和意图，所以它就成为整个测试过程的不可分割的一部分并贯穿整个产品的生命周期。特别是在产品发布后，需要一个产品维护团队继续为用户提供产品支持的时候，这个测试设计规格说明将是不能缺少的一部分。

测试设计规格说明的基本元素示例

下面是一些经常会在测试设计规格说明中出现的条目：

- 概要、目标、目的
- 策略
- 功能测试
- 组件测试
- 集成测试或系统测试
- 互操作性测试
- 一致性测试
- 国际化测试（Internationalization）和全球化测试（Globalization）
- 性能测试
- 安全测试
- 安装或部署测试
- 依赖关系
- 度量

用好的和坏的数据测试

二加二等于四，但当你用二除以零的时候会怎么样呢？随微软 Windows 操作系统一起发布的应用程序运算器用“不可被零除”这一串文字来表示除零的结果。在其它的应用程序中，除零也许会得到类似于这个结果，抑或引起程序的崩溃。测试用例一般包括验证测试（使用期望的输入来验证产品功能的测试）和错误避免测试（使用预期之外的数据来检验产品是否能适当处理的测试）。验证测试是必要的，它证明了应用程序可以按照设计原意工作。而错误避免测试则可能会更重要一些。应用程序需要健全并能够在自身不出错的情况下处理坏的数据。

许多年前，对于使用坏的数据导致的一个应用程序的出错或者是崩溃，程序员一般的回应都是“用户绝对不会那么做的”。我就有过一次很偶然地将一个应用程序的后缀名改为这个应用程序的文档文件所用的后缀。再用运行正常的同一应用程序试图打开这个修改过的程序，结果导致了程序崩溃。但是这个 Bug 被认为是“不需要解决”因为“这不是一个用户使用场景”。这个缺陷一直存在在这个应用程

序里很多年，终于在 2002 年微软大力推行产品安全性的时期被修正了。事实上，正因为微软极度强调软件的安全性，一件很了不起的事情发生了，那就是那些被证伪性测试发现的缺陷，很少会因为“用户决不会那么做”的理由而不被修正。

“开心路径”的测试结果就应该总是通过

有一次早晨，我刚到公司就看到一封来自我们组的软件开发工程师 Adam 的邮件。上面说，他已经将他在周末写完的新组件签入了，他希望我在产品被编译出来以后能尽快做一些随机（ad-hoc）测试。我的时间很有限，但是我很开心终于可以测试这个产品组件了。实际上，我已经写了我很想试试的几十个测试用例，这些测试用例都是基于设计阶段我跟他讨论的内容设置的。

过了不久，产品被编译出来并发布给测试组。我在我的测试计算机上安装了那个软件，然后立刻就找到包含了 Adam 写的新功能的那个部分，输入了一些数据，点击其中一个按钮。按钮没有效果，什么都没发生。我并不是想要看这个功能是怎么处理坏的数据的。我用的只是一些应该能工作的简单输入。这里，我所指的应该总能工作的“开心路径”就是说“简单的输入参数”。

就是因为开心路径应该总是可以工作的，我立刻就假设一定是我犯了一个错误（*我知道我安装的太快了——我一定是忘记了选择某个选项*）。在我的办公室，我有第二台未曾安装过这个软件的计算机，于是我就花了一些时间仔细地安装了这个产品。我没有发现任何看上去将会影响这个功能的设置，因此我就猜测是不是我的测试计算机上有一些以前的文件使整个功能不能正常工作。很不幸，再次运行这个应用程序之后，我看到了相同的结果。*我一定做错什么事情了*。

于是我离开了我的办公室。顺着走廊一路走过去，我停在几个正在工作的同事的办公室门口，问他们我是否可以借他们的测试计算机用一会。我竭尽可能地尝试，然而我就是找不到一台计算机可以让这个新加入的功能工作。终于，我回到了我的办公室然后打电话给 Adam，告诉他这个坏消息。当我描述了我在过去的一个小时里所作的所有事情以后，他说，“嗯，我在代码检入之前做了一个改动，但我没觉得这个改动会引起什么变化。我想我错了”。这个时候，我已经浪费掉了一个多小时的时间，我变得有点儿不满，回答道，“Adam，我很严肃地跟你说，开心路径就应该总是能通过的”。

现如今，很少见到“开心路径”上的测试用例不能通过了。但每次出现了应该通过而不通过的情况，我总是记得并且重复这句话。

在测试用例设计中要考虑的其他因素

进度，资源（预算）以及质量是可以影响软件测试的依赖属性，就如同它们对软件开发的影响。比如，如果时间和金钱都不是问题，测试就可以无限期的继续下去亦或是只要项目需要就继续添加测试工程师。然而，软件是需要发布给用户使用的，并且在很多情况下，添加人手一般不是一个可选项。到最后，做一个充分的测试用例设计需要测试工程师事先考虑好可能的测试范围，也需要测试工程师能够全盘考虑并确定优先级，以使测试能够在满足项目进度要求的同时对产品进行了充分的测试。测试所有的方面从来都是不可行的，因此一个测试工程师是否可以选择最好的一组测试并有效地、周到地在指定时间内完成测试就变得非常重要。

设计测试用例时，需要考虑产品的范畴，用户基数的大小，测试团队的大小以及测试团队的技能。回答与这些因素相关的问题能够帮助你选择一个可以验证产品功能、找出错误并有效处理用户问题的测试集合。

黑盒测试、白盒测试和灰盒测试

一种划分测试用例设计的方法是根据测试工程师和测试程序对被测对象的了解程度来划分的。一

一个普遍被大家所知的系统是黑盒测试和白盒测试。 黑盒测试 (*Black box testing*) 是指一种设计测试用例的方法，它不需要知道任何关于应用程序内部是如何实现各种功能的知识。 根据一个应用程序的用户只关心这个应用程序是否满足了他们的需求，而不关心（他们也不应该）这个应用程序是如何被设计和实现的原理，黑盒测试的方法是一个有效的方法去模仿和预估用户会如何使用这个产品。 而从另一面看，单纯的黑盒测试方法也经常会导致过度测试一个应用程序的某些部分而对另一些部分没有做足够的测试。 与黑盒测试相反， 白盒测试 则是通过对应用程序内部代码或者用户看不到的模式进行分析，并以此设计测试用例。 单一凭借白盒测试方法设计的测试用例一般都非常详尽，但无一例外的总是会错过关键的终端用户使用场景。

解决这个设计测试用例的两难处境就是使用灰盒测试（有时也称为玻璃盒测试）。 设计测试首先是从用户关心的角度出发的（即黑盒测试），然后再利用白盒测试方法保证测试用例能够有效并全面的覆盖被测对象。 测试工程师需要从两个方面进行测试： 用户角度和确定应用程序的正确性的角度。 为了能够有效地涵盖这两个角度，就必须考虑使用黑盒测试和白盒测试两个方法。 所以，微软的测试工程师必须对测试有一个完整的观点。 同时在设计测试用例的时候，还需要考虑到各个方面。

探索性测试在微软

微软最近对自动化测试的强调使得人们认为微软低估了探索性测试的重要性。 探索性测试（一般来讲）是一种手工测试方法，每一步的测试和验证都是基于前一步的操作。 在进行探索性测试时，测试工程师们需要根据已了解的被测产品的知识以及掌握的测试方法学，快速地找到产品的缺陷。 Windows 应用程序兼容性测试组就是一个例子。 他们在开发新版本的操作系统期间，很大程度上就是依靠探索性测试来验证上百个应用程序的功能的。

在一个不断强调自动化测试的团队里，探索性测试方法在早期的测试设计阶段很有帮助，它可以影响自动化测试的结构和目标。 测试团队经常会在产品开发周期中做“缺陷大扫除（bug bash）”，这种时候测试工程师一般都会用几个小时的工作时间用探索性方法对他们的产品进行测试。 这种方法旨在模拟用户体验，而且通常能够成功发现其他测试方法可能会错过的缺陷。 在缺陷大扫除结束之后，大部分团队会分析那些刚发现的缺陷，并用这些发现影响后续的自动化测试用例的设计。

在那些需要较高自动化测试水平的团队里，测试工程师经常需要使用探索性测试方法将详细功能说明与其他相关信息联系在一起，并影响测试用例的设计。 简言之，在做测试的时候，及早发现重要的缺陷是非常必要的，与此同时，还要努力设计出可以发现缺陷，并保证可以在整个产品生命周期之内验证产品的功能以及正确性的测试用例。

在微软，另一个新奇且成功的探索性测试方法是结对测试 (*pair testing*)。 这种测试方法是受结对编程 (*pair programming*) 的启发而产生的，两个测试工程师会一起完成探索性测试。 一个测试工程师控制键盘，执行某个产品功能或者应用软件，而另一个工程师则在执行工程师的旁边一同引导测试。 两个测试工程师都在做探索性测试，但其中一个关注于如何调用各项功能，而另一个则是从高层面考虑被测程序。 两个工程师每过一段时间互换一次角色。 在八个小时期间，15 对测试开发工程师们一共发现了 166 个缺陷，其中包括 40 个被列为严重程度为 1（这表示必须马上解决）的缺陷。 从针对这三十个参与者的问卷调查中收集到的反馈意见来看，仅有 3 人认为结队测试要比单独测试乏味，只有 4 人认为结队测试没有单独测试有效。

本章小结

为了能使你设计的测试用例被持续地使用十年以上，设计测试用例的一个关键是不断地实践各种技术和方法，并将得到的所有信息用于后续的测试活动。这里没有所谓的正确的或者错误的测试方法，并且也没有所谓的“银弹”技术可以保证优秀的测试。最重要的就是要花时间去了解每个组件，每个功能，或是应用程序，并且要基于对各种各样技术的了解来设计测试用例。可以看到，使用多种技术设计测试用例的策略要远比使用有限几种技术设计的测试用例更有可能成功。

大多数测试工程师在开始他们的测试职业生涯的时候并没有很多测试设计经验。尽管许多优秀的测试工程师是通过实践或者与更为博学或有经验的同事沟通中得到测试用例设计的好主意，但最好还是在测试工程师从业初期，就给他们教授和传达如何设计测试用例的优秀方法。

接下来的几章将会介绍在微软内部测试工程师培训课程中参照的几种测试用例的设计方法。微软公司的测试工程师通常在设计测试用例时都会考虑到这些技术。

第 5 章 功能测试相关技术

BJ • 罗里森

当我还是个小男孩的时候，我就对事物的运作机制满怀好奇。有一年圣诞节，我从圣诞老人手里得到了一套撞车大赛的车模。不知我和我父亲谁更加兴奋，反正我们立刻在餐桌上将车模组装起来，尔后父子俩把圣诞节当天顶好的时间都投入了撞车大赛中，我们绕着赛道飙车，并想方设法地互相猛撞。最有意思的是，赛车的车身会在某些剧烈撞击之下散得七零八落。大概正是在人生如此的早期阶段，我萌生了把东西拆解的兴趣吧。最终，虽然意犹未尽，但母亲非让我们把车轨从餐桌上拆下来给圣诞大餐腾地方。不过，用过晚饭以后，我们父子俩又回到餐桌上继续我们的互撞游戏，直到就寝时间为止。那一天过得真是太棒了！

从那以后，如果父亲没空，我就会缠着姐姐陪我玩这个。可是，比起我娴熟、高超的赛车控制手法来说，他们统统不是对手。游戏本身当然乐趣十足，但几个星期以后，好奇心逐渐在我心里占了上风。**我希望**，不对，**我必须**——地弄明白这赛车到底是怎么运作的。某个晚上，当只有我一个人在卧室里时，我把其中的一辆赛车拆了个底朝天，要弄清电动机是如何通过一组小零件驱动车轮转动的。但是电动机本身又是如何运作的呢？我仔细地查看，发现了一块磁铁，以及绕在塑料轴承上的一些细小的铜线圈。我惊讶地看到：电动机只要在我接触到车轨的触点时就会转动。我又想，铜线圈下是否还暗藏玄机？于是我刮去了线圈上涂敷的薄漆层，还把线圈拆开了其中一圈。回想之下，那个主意是打错了，因为我再也没法把赛车发动起来了。我父亲对我拿新玩具大拆特拆一事当然不太开心，但他还是带我去了车模商店买回了一辆新赛车以继续我们的比赛。

那一天，我对电动机和赛车零件有了不少心得。这成了我以后的人生经验，总能让我对某种事物获知良多的做法就是先将其分解，然后再逐个部分地将其拼凑起来以了解其各个部分是如何运作的。这种格究事物运作机制的满腔热情至今仍流淌在我的血管之中。持久的好奇心看来是绝大多数测试工程师身上都根深蒂固的特质。从用户界面出发，对软件进行探究固然重要，但如果测试工程师想切实理解软件的运作机制、了解软件究竟能够具备何种能力，我们就必须潜到用户界面之下，深究我们正在测试的系统就可以了。杰出的测试工程师不仅依靠其好奇心去探究产品，还会对其作深入挖掘、在更加精细得多的粒度上，对正在测试的软件的能力及属性实施深度分析。

收集更多深度信息的途径之一是将产品的功能集进行分解，然后针对分解后的各个组件的功能属性及能力分别实施测试。功能测试相关技术赋予测试工程师系统化的途径，以对各个功能点和组件进行更为全面的考察。功能测试相关技术为数不少，但本章中，我只讨论一部分关键的，也是在整个微软公司范围内被广泛采用功能测试相关技术。

功能测试的需求

现代意义上的软件极其复杂，而对当今软件测试的实施尤其是个巨大的挑战。因为测试工程师必须有能力设计和执行一组测试，并使项目复杂责任能够有适当的信息，一方面突显潜藏的风险之所在，另一方面又对软件的重要属性，和能力指标做出质量评估。测试工程师必须从所有可能实施的测试中定义出一个有限的测试集，还必须给整个业务组以信心，那就是测

试已经揭示了关键问题，对产品的重要功能点也给出了恰如其分的质量评估。而且，人们总是期望测试工作在一段相当有限的时间内完成！所以，我们在挑选任何可能采用的软件测试途径时，都要保证它具有合理的系统性和条理性，能够给出相对小巧的有效测试子集，以对预先提出的假设予以证明或否定证明。

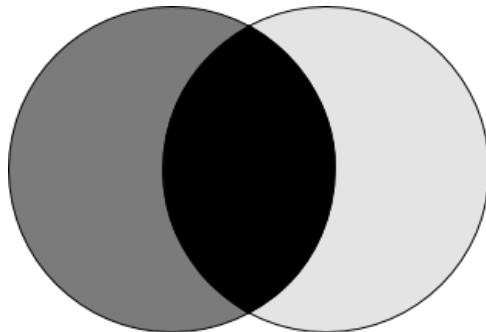
有一种软件测试的途径被称为探索性测试¹⁵。探索性测试（Exploratory testing, ET）是一种常用的测试途径，它主要关注于行为测试。ET 在对软件进行初步评估摸底的时候特别有用，它有助于使测试工程师对于测试的软件迅速形成一个概念。ET 也能够有效地对软件的操作能力和整体可用性给出一个初步的、宏观的总体评价。探索性测试对于小型的软件项目、发行范围有限的软件以及有效期不长的软件来说可能已经足够了。

然而，探索性测试总的来说不能适应大型的复杂项目，或是任务非常关键的软件。在微软内部，我们的经验也指出探索性测试并非是针对需要，对软件发行版本进行长期维护的可持续型软件工程（sustained engineering）进行测试的最佳途径。我们发现，仅仅依靠测试工程师和领域专家，通过软件的用户界面来探索和审阅软件的能力，并不能给项目主管提供必要的信息，以保证他们做出既全面又合理的有关产品质量和风险的决策。巨大的缺陷数目只能使那些就会盯着数字看的家伙们满意，但像未经任何加工的缺陷计数和针对某个功能领域测试时间累计这样的数据并未给出比“在考察某个东西时花了多长时间”以及“有否发现多少个缺陷”更多的信息。所以，如果管理层需要更多有关软件的信息来尽可能地降低风险，以及做出合理决策的话，我们就必须对于正在测试的软件的各个功能组件做出更为全面的分析。

Boris Beizer 指出，使用黑盒方法进行软件行为测试，仅能企及所有测试能够覆盖范围的 35% 到 65% 之间¹⁶。从用户界面出发进行软件行为测试的确很重要，但如果它被用作唯一或主要的测试途径，我们就很有可能浪费时间在效果不彰的测试上，并且会漏过产品的某些重要的领域，如图 5-1 所示。微软内部以及其它业内公司的研究得出的经验数据稳定地彰显了行为测试在有效性方面的局限（参见下一个标题为“Weinberg 三角形：再考察”的边栏条目）。是故，软件测试工程师肯定要问：我们如何提高测试工作的有效性以减少冗余，并且使我们整个项目组面临的风险都得到降低？

¹⁵ 探索性测试在本书第 12 章“其它工具”中有更详细的讨论。

¹⁶ Boris Beizer, *Black-Box Testing: Techniques for Functional Testing of Software and Systems* (New York: John Wiley, 1995)



- 由典型的黑盒测试途径能够企及的总体有效度
- 潜在的冗余或不能提供有效信息的测试工作
- 未经测试的软件领域——理论上，风险应该 100%位于未经测试的领域

图 5-1 演示黑盒测试有效性的文氏图

Weinberg 三角形：再考察

五年多以来，我们给予软件测试工程师（Software Development Engineers in Test, SDET¹⁷s）的内部培训都依靠一个从 Gerald Weinberg 的原始问题实现的三角形仿真软件进行基准技能摸底。新上岗的软件测试工程师被要求在 15 分钟内，利用以前学过的技能和知识来对该软件进行探索，然后定义出能够判定软件是否已经满足了功能需求文档所指定的能力的测试集。功能需求文档规定，软件读取三个整型值，分别代表三角形的三条边的长度。然后软件显示一个消息，指出该三角形是不规则三角形、等腰三角形还是等边三角形。有些人抱怨给的时间太短，但这种紧迫性模拟了实际工作中的进度压力，也能看出一个人在给定时间里遴选出最关键的、能够给出最有价值信息的测试的能力。

在审阅了五千多份样本以后，我们发现大多数软件测试工程师只写了一个针对合法整型值导致非法三角形的测试、一个针对等边三角形的测试、一个针对不规则三角形的测试和一个针对等腰三角形的测试。这四个测试仅仅覆盖了软件中最关键路径的 50%而已。对测试输入的进一步分析表明，只有极少数测试包含了，用以测试合法整型值导致非法三角形的复合谓词表达式中的第三个条件分表达式。经验数据证明，从所有测试中任意采样，都只有 10%不到的测试会完整地执行复合谓词表达式的三个条件分表达式。我在华盛顿大学任教时和在技术研讨会上，都使用了同一个仿真程序来做摸底，结果也大同小异。幸运的是，我们的内部研究表明，仅仅经过了几个小时针对测试技术之合理运用的实际训练，对于类似的仿真软件而言，测试用例的有效性就提高了 60%到 75%，测试的优先级排定也更合理了。

提高测试的有效性并收集产品特定方面关键信息的一种途径是采用功能测试相关技术。功能测试相关技术是指某些系统化的过程，使我们能够对软件的属性和能力实施全面的考察。功能测试相关技术通常应用于从用户界面出发的测试，但是同样也能够应用于设计黑盒和白盒视角出发的测试。当我们能够正确地运用功能测试相关技术，并将其用于恰当的场合时，它就可以帮助我们达成对系统更为彻底的分析，并降低测试对于“杀虫剂困境”的感染机率。

¹⁷ Boris Beizer, *Software Testing Technologies*, 2nd ed. (New York: Van Nostrand Reinhold, 1990).

杀虫剂困境

我的一个业余爱好是园艺。我和我的女儿早春时分在暖房里播种，然后等待到四月底或五月初，然后再把幼苗移栽到菜园里。但是，一旦植物被移到了菜园里，它们就面临着数种昆虫和寄生虫的威胁，我也必须找到某种自然的防护措施来阻止野兔、蛞蝓或其它害虫，否则我们的蔬菜作物就会被吃得七零八落。野兔对菜园的破坏其实并不算很大，而且我女儿总是一看到它们就会作出反应，所以那时我后院里最令人深恶痛绝的害虫就是蛞蝓。

信不信由你，蛞蝓对啤酒相当偏爱。所以，啤酒筑就的陷阱出奇地有效，在蛞蝓成功进入菜园之前就将它们拦下。我觉得区区蛞蝓还不至于能够品出廉价的罐装啤酒和太平洋西北部出产的特等专酿的高下，所以我还是采用了省钱的办法。用铜栅栏沿着菜园周围的土堆扎一圈儿，就形成了一道阻止蛞蝓的屏障。定期在土堆外部洒盐当然也是击溃蛞蝓入侵的有效办法。木屑和蛋壳是对付这些蛞蝓小贼的另外一些途径。可不管我在菜园里布下怎样的天罗地网来捕杀和阻挡蛞蝓，总有一些漏网之徒还是会成功突破。

和我的蛞蝓阻击战十分类似地，我必须在实施一个以揭示潜在缺陷为目的的软件测试时，采取各种方法和手段的搭配。富有经验的测试工程师会告诉你，没有任何一种单一的测试方法能够有效地揭示出所有种类的缺陷或，和对软件的能力做出完整的评估；这种困境被称为杀虫剂困境。杀虫剂困境这个术语源自 Beizer 第一定律：“任何你用以防止或发现缺陷的方法都会留下一些残余的、更为微妙的缺陷，而对于这些缺陷而言，前面那些方法会统统失效。”本质上，没有任何一种单一的技术或手段对于软件测试来说能够百试百灵。所以，增加测试方法的多样性，以及考虑从更多的视角来审视软件的话，我们就更有可能发现更多的潜在问题的同时，也提高了测试工作的有效性。

园艺工作确实很费神，也会遭遇一些特别的挑战。但是当我看到六岁的女儿从枝头摘下一支甜椒或一个小蕃茄美美地吃掉时，我觉得一切都是值得的。同样地，软件测试工作是辛劳的，并且面临着特有的困难，但当经由一系列的手段达成了恰当的结果时，对于整个产品组的回报也是令人鼓舞的。

功能测试相关技术带来的另一个好处在于它具备这样的潜能：它能够把针对产品特定功能范围内的测试覆盖率逐渐提升至一个很高的置信度，同时降低测试用例的冗余数量。不过，一如前述，运用任何测试技术的价值和有效性取决于个别测试工程师的对于系统和领域的知识以及将适当技术运用于适当场合的能力。关键在于，功能测试相关技术只是专业测试工程师使用的工具，用以收集软件的能力和属性的关键信息，以及揭示潜在的缺陷。我们必须了解，在特定的场合应该使用何种技术，以及如何有效地使用这些技术。

测试技术只是愚蠢的噱头吗？

有些人以为，所谓技术不过是愚蠢的噱头，或觉得它们实在不值一提。在我看来，对于未经训练的观察者来说，技术的确像是未经大脑思考的花拳绣腿。但是技术是一种系统化的过程，它提供了一种途径，用以解决一种复杂的问题，通常它源于某种良构的启发式思维。许多专业工程师都倚仗技术，而一旦技术为熟练的测试工程师所正确运用，它们就会成为得力的工具。测试技术在提供更为彻底、严格的软件功能点的能力和属性分析、及提高有效测试覆盖率方面很是实用。

十分重要的一点在于，我们要知道，测试技术的正确运用并非死记硬背。技术不是测试工作的万金油，它们当然也并非我们实施软件测试的不二法门。事实上，功能测试和结构化测试相关的系统化软件测试相关技术要求广博的领域和系统级知识、逻辑推导能力、分析论证能力、辩证思维，以及测试工程师的慧眼，在一个给定的上下文中检视测试的结果或是预想假定的正确性。另外，很多技术对于我们证明或证伪某种特定的缺陷类型或分类的存在性，以及校验产品的功能性时特别有用。

功能测试相关技术，如边界值分析（boundary value analysis, BVA）、等价类划分（equivalence class partitioning, ECP）、组合分析，以及状态转换测试等就像工具箱中的各种工具，每种工具都适用于不同的用途。若在适当的场合中加以正确应用，这些工具都能够十分有效地帮助测试工程师系统地评估特定的软件功能点的功能属性。系统化的功能测试相关技术并非专业测试工程师的工具箱中的独有工具，而且最好在要求更多细化的手动工作的场合下加以运用。但我们必须记住，要想让我们工具箱的任何工具物尽其用，都需要聪明而且知识丰富的人，他们对于把工具在适当上下文中用作潜在的解决方案的方法和时机胸有成竹。

等价类划分

测试工程师应该首先了解并掌握的功能测试相关技术是等价类划分（equivalence class partitioning, ECP）。理解这种技术的用法十分重要，因为它是很多其它软件测试技术和途径的基础。ECP 技术，简单地说就是一种工具，使得测试工程师能够以系统化的方式评估一个功能点中每个参数的输入和输出变量。不过，要等价类划分技术发挥出最大功效，就要求我们针对特定系统的上下文中的每个参数的变量数据实施全面的分析。所以，在设计 ECP 测试之前，我们有必要仔细为每个输入和输出参数涉及的变量数据实施解耦和建模以将其模型成分离的合法及非法分类子集。然后，ECP 测试就可以经由这样的手段导出：创建每个合法分类子集的合集，直到测试取遍合法分类的所有子集为止；尔后，再对非法数据子集依次评估。这个解释看起来平凡无奇。但是，这种技术对于知识既有深度也有广度的要求，方能将输入和输出变量数据精确解耦成分离的合法及非法分类子集。这些子集以这样的方式定义：给定子集中的任一元素与任何同一子集中的其它元素都能够给出在逻辑意义上相同的结果。我们既可以定义正向测试，也可以定义负向测试，以针对输入和输出参数在功能方面的能力，以及系统是否对于所有种类的错误都具备了足够的处理能力实施系统的评估。

论及 ECP 技术的功效，可谓一举两得。首先，它有助于我们系统化地降低从所有可能的测试中遴选的测试的数目，但仍然为我们提供了高度的信心：在同一个子集中的其它变量或变量组合会重复地给出相同的期望结果。举例而言，假设你在测试一个文本框控件，它接受一个由大写的 A 到 Z 之间的 Unicode 字符组成的字符串，其最小长度为 1，最大长度为 25。穷举测试包括每个字母出现一次 (26^1) 以及每种可能的字符串长度条件下的所有字母组合。所以，为了测试所有的可能输入，总的测试数目就等于 $26^{25} + 26^{24} + 26^{23} + \dots + 26^1$ ，这是一个蔚为可观的数目字（really big number, RBN）！但让我们做这样一个假设，即对于字符串“BCD”的测试和字符串“WXY”的测试是相似的。这样，对于指定参数的合理分类子集就将包括以下几点：

- 大写的 A 到 Z 之间的大写 Unicode 字符

■ 一串长度在 1 到 25 之间的字符

基于这样的建模来判定例子中的输入字符串的合法性时，字符串“ABCDEF”所产生的结果应该与字符串“ZYXWVUTSRQPONM”无异，如果字符串由“KUGVDSFHKIFSDFJKOLNBF”组成，也应该得到相同的结果。

ECP 技术的另一个优势在于我们可以通过从给定的测试子集中随机选择元素用作测试数据，以有效地提高数据的覆盖率，因为在特定的合法或非法子集中任何一个元素都会产生与同一个子集中其它元素相同的结果（前提是数据已经正确地被分解成离散的子集，而且也没有偏差的软件行为）。再看前例，针对参数为“名字”的测试必然会包括实际中常用的数据如“Bob”、“Nandu”、“Elizabeth”或“Ling”。但是静态的测试数据在提供新的信息方面贡献甚微，也不能在首次迭代之后对算法实施健壮性的评估（除非算法有了根本性的更改）。静态数据同样地也不能应付随机的变量全排列。从合法和非法子集中随机选择元素为软件用例的后续迭代提供了巨大的多变性，而且增加了将不常见的畸形用例曝光的机率，而它们本来是不太可能通过典型的或是实际中常用的合法类的静态数据测试出来的。这是使用随机概率测试数据来丰富测试覆盖率的基础。

再怎么强调 ECP 技术的整体效率主要依赖于测试人员的这种变量分解能力，即依据给定参数把变量数据分解为定义良好的子集，也不过分。逻辑上，特定子集中的任何一个元素和该子集中的其他任何元素将产生相同的预期结果。我们花费大量的时间用于复习巩固 SDET 培训中的这些概念。当然，把变量数据建模为等价类子集需要对给定上下文环境系统的认识和理解。因此，测试人员了解系统和领域空间的知识越少，那么越可能不准确地运用这项技术。同时，越有可能遗漏某些关键性的检测而执行冗余测试。

变量分解

等价类划分中最困难、且最具挑战性的方面是把数据分解为唯一合法和非法类子集。我们必须非常熟悉数据的各种类型，了解程序和系统运行、操作、转换和存储数据的机制。同时必须知道输入变量的可能值以及回顾失败指示信息，例如：先前有问题的变量（已知的问题测试数据和高风险数据）。我们也必须考虑外部因素，例如：用户知识结构、规范和需求，这些通常能帮助我们定义合适的上下文环境。

变量数据不充分地分解将会导致在特定类中子集数目的减少，同时增加遗漏错误的可能性。相反，把变量数据过量的分解为非唯一的子集将会增加潜在的冗余测试数量；然而，冗余测试可能不会揭示额外的错误，且不太可能提供新的相关信息。我一再提醒你要谨慎，是因为你总会有可能执行这种冗余测试。然而，假如由于你过于宽泛地归纳测试数据而遗漏了关键性的问题，这个代价可能会相当昂贵。等价类划分的数据分解理论提供了这样一种基础，即理解系统和领域空间对于 ECP 技术的巨大成功和极大化效率具有关键性作用的原因。

ECP 数据分解理论

过于宽泛的变量数据归纳减少了 ECP 测试的数量，但增加了遗漏错误的可能性或产生负向错误（*false negatives*）或正向错误（*false positives*）。变量数据过量的分解会增加冗余测试的可能性，这会减低 ECP 测试的整体效率。

最初，我们必须把数据分为两类。合法类数据包括这样一种变量元素集，即这些数据在通常环境下返回正的结果。换句话说，这种数据不应产生错误的条件或导致非预期的失败。非法类数据包括这样一种变量元素集，即这些数据应产生错误条件。大部分的错误条件是错误管理者设下的，但偶尔非法类数据也有可能增加揭示错误行为或失效的可能性。

一旦数据被划分为合法和非法数据类，测试者必须仔细地分析每一类数据，进而将数据进一步分解为分离的子集。本质上，在合法类或非法类独立子集中的任何一个元素在测试时应产生相同的结果。这就是为什么我们无须测试在 A 到 Z 每一个大写 Unicode 码，或像先前例子中每一种可能的字母组合。

ECP 和神秘错误信息的实例

当我在开发一个内部培训课程时，偶然发现了在 Windows XP 操作系统中有关文件名的奇怪问题。我设计的训练要求学生把 ASCII 码分解为等价类子集，使用 Windows XP 操作系统的记事本程序中保存对话框将其保存为文档。这是一项不太容易的任务，因为要获得成功需要对硬件平台、操作系统环境和字符集有着深入的理解。

过去，我通过在母板上配置不同的软件来为客户构建 PC 系统。这是早先时候，当时我对为 BIOS 刷新 EPROM 和为 IRQ 和 DMA 通道改变跳线非常熟悉，使这些从不同生产商获得的硬件能协调地互相工作。我知道一些其他深奥的细节，如：为 LPT 保留的存储空间和被 PC/AT BIOS 保留的 COM 端口。这就意味着由字符串 LPT1 或 COM6 组成的基本文件名在 Windows 平台上应导致相似的错误信息，以指示文件名是保留给设备名称或类似的原因。因此，当我给这个训练准备解决方案时，我逻辑上总结出使用了字符串 LPT1~LPT9 或 COM1~COM9 的基本文件名是等价的，都将产生相同的预期结果。

但是，当我测试我的断言以证实方案的准确性时，我遇到了别的没有预期到的行为。当我输入 LPT2 作为基本文件名的参数时，系统提示了预期的错误信息，“这个文件名是保留给设备名的”。但是，当我输入 COM7 作为文件名参数来测试我的等价理论时，我得到了完全不同的信息，“以上的文件名是非法的”。

我没花多长时间得出结论：LPT1~LPT4 和 COM1~COM4 得到预期的结果，文件名被保留为设备名的错误信息，LPT5~LPT9 和 COM5~COM9 得到不同的错误信息，一个更加迷惑的信息。只能推测为何逻辑上等

价的集合会被强行分离了呢。幸运的是，这个小小编码上的错误在 Windows Vista 操作系统中已被解决了。但是我们仍然在培训中使用这个例子来强化最好的实践，即在每一个等价类数据集合中使用多个元素来增加覆盖的宽度并减少错误功能的可能性。

为了帮助我们把数据分解为在 Glenford Myers 每一个类中的离散子集，提出了在书《软件测试的艺术》¹⁸中的四个启发式方法。启发式方法是指导方针、原则或拇指规则（rule of thumb），它在决策制定、故障检修和解决问题这样的任务中十分有用。可以帮助我们把数据分解为在合法和非法类中的离散集合的四个启发式方法包括：值的范围、变量相似组、唯一值和特殊值。

■范围 在数据的邻近集合中最小值和最大值间的任何数据点应产生相同的结果。例如，你可以从 1 到 999 间输入一个整数。合法的等价类是 ≥ 1 和 ≤ 999 。非法的等价类范围包括 < 1 和 > 999 的整数。

■组 只要元素被等价地处理，那么元素组是允许的。例如，车辆的元素列表通常决定了征税的类别，包括：卡车、小汽车、摩托车、房车和拖车。如果卡车、小汽车和房车属于相同的征税类别，那么这个元素组认为是等价的。

■唯一值 在类或子集中的数据可能以不同于同一类或子集中的其他数据的方式被处理。例如，数据 1 月 19 日,2038,3:14:07 在处理时间日期数据的应用中是唯一的。它应被分解为离散的子集。

■特殊值 条件必须提供或必须不被提供。例如，在 SMTP 协议中，字符“@”是一个特殊字符，不应该作为 e-mail 名称或域名的一部分。

等价类划分实战

一个简单的例子能够帮助你更好地理解如何把参数变量划分为离散的合法和非法数据子集。在图 5-2 所示的“下一个日期”程序需要输入三个整数，代表了阳历中特定的月、日和年。同时以月/日/年的格式返回下一个阳历日期。决定下一个日期的算法是基于阳历的，即目前世界上通用的日期。这个程序用 C# 实现，不管用户的本地设置如何，输出格式总是为月/日/年（意味着输出格式对文化不敏感）。

¹⁸ Glenford Myers, *The Art of Software Testing* (New York: John Wiley, 1979).



图 5-2 用 C# 实现的“下一个日期”程序

表面上，这个程序的输入和输出数据是相当简单的。在这个特定的例子中，测试人员必须拥有阳历、**papal bull Inter gravissimas**¹⁹的基本知识，同时甚至必须了解一点英格兰的历史。系统知识也用于确定潜在的子集。例如，系统包括硬件 BIOS 时钟和用于开发该程序的编程语言。两者对准确地确定和逻辑上把变量划分为离散的合法和非法类子集是十分重要的考虑因素。表 5-1 展示了这个特定程序最有效的数据集划分。

表 5-1 “下一个日期”程序的输入输出参数的 ECP 表

输入/输出	合法类子集	非法类子集
月	v1—拥有 30 天的月份 v2—拥有 31 天的月份 v3—二月	i1—>=13 i2—<=0 i3—非整数 i4—空 i5—>=3 位整数
日	v4—1-30 v5—1-31 v6—1-28	i6—>=32 i7—<=0 i8—任何非整数

¹⁹ 译者注: *Inter gravissimas* 是 February 24, 1582 教皇 [Gregory XIII](#) 发布的关于阳历规定的教皇训令(*papal bull*)

	v7—1-29	i9—空 i10—>=3位整数
年	v8—1582-3000** v9—非闰年 v10—闰年 v11— s 世纪非闰年 v12—世纪闰年	i11—<=1581 i12—>=3001 i13—任何非整数 i14—空 i15—>=5位整数
输出	v13—1/2/1582- 1/1/3001	i16—<=1/1/1582 i17—>=1/2/3001
唯一的或 特殊日期	v14—9/3/1752- 9/13/1752 v15—1/19/2038	i18—10/5/1582- 10/14/1582

参数子集分析

第一个参数是月份输入。这个参数接受从 1 到 12 的整数类型变量，代表了一年中的十二个月份。本质上，这个程序必须能区分拥有 30 天的月份、拥有 31 天的月份以及闰年和非闰年的二月份。因此，可以把月份参数变量划分为三个合法子集，即拥有 30 天的月份、拥有 31 天的月份以及二月份。这是相比于其他合法类子集一个唯一变量状态。你也可以创造 1 到 12 的第四个合法类子集，来确认这个输入参数最小和最大边界条件。但是给月份仅仅创造一个 1 到 12 的子集可能是过于宽泛的归纳，从而导致遗漏某些关键测试。非法类子集包括大于 12、小于 1、任何非整数值、空字符串和三位及三位以上的参数。

日参数的变量数据基于拥有 30 天的月份、拥有 31 天的月份以及闰年和非闰年的二月份被划分的四个子集。依据每种月类型把日参数划分为离散子集，其基本的理由是帮助确定 1 到 31 间特殊的边界条件。等价类子集一个额外的好处是它们通常能指示出特殊边界条件。例如，日参数明确地指出五个特殊边界条件（取决于实际实现），即 1、28、29、30 和 31。

此外，把日范围划分成独立的子集有助于明确你并没有产生正向错误。例如，假如你把月参数和日参数合法类子集分别宽泛地归纳为 1 到 12 和 1 到 31。自动化测试可能随机产生 2 月和 30 日，并且期望得到下一个日期为 2/31/XXXX，这是一个正向错误。

年参数包含合法年份范围的子集、闰年子集、非闰年子集、世纪闰年子集和非世纪闰年子集。闰年、世纪闰年和非闰年在程序中可以使用如下的数学公式较容易地决定。

```
//Example algorithm used to calculate leap years
public static bool IsLeapYear(int year)
{
    return (year % 4 == 0 && year % 100 != 0 || year % 400 == 0);
}
```

因为闰年的决定是基于数学公式的，测试在 1582 和 3000 之间的任何闰年和世纪闰年总会产生相同的结果并返回真。也要注意到对于闰年或世纪闰年没有特殊的边界条件。对于这个参数，输入最小值和最大值分别为 1582 和 3000。年 3000 对于这个特定的程序是禁止的，因为假设到那时我们都已经出身了。

最后，有两个特殊的日期范围和一个特殊日期比其他任何日期都要令人感兴趣。第一个日期范围包括 10/5/1582 到 10/14/1582 的非法类子集。这些日期被排除在阳历之外。这个范围中的任何一个日期应导致一个错误信息或指示该日期按照功能需求是非法信息。合法类日期范围是在 9/3/1752 到 9/13/1752。英格兰和她的殖民地直到 1752 年才采用阳历，因此他们排除这十天以同步以月亮为参照的历法。但是，这个程序是基于最初的阳历，因此这个日期范围比其他日期更加特别。了解英格兰和美国历史的开发者可能不经意间排除了这些日期，而实际上这些日期在这个特例中是一个缺陷。

日期 1/19/2038 比起其他合法日期而言是一个特殊的日期。BIOS 时钟给开发者提供了一个计数用于测量时间。虽然在这个程序的实现中，开发者不太可能利用这种计数。在测试者能证实程序实现的准确性之前，他们都应利用所有可获取的信息，至少最初应把这个日期作为一个唯一的子集。

周五第十三个缺陷

千年虫问题影响十分广泛，在金融界几乎引起了恐慌。但是一个潜在的更为严重的问题正在隐约呈现。在 PC/AT 计算机上的 BIOS 时钟是从 1970 年 1 月 1 日 00:00:00 时开始的。逻辑上这个日期没有任何问题，但是 BIOS 时钟使用 32 位整数进行计数来跟踪时间。这就意味着 2038 年 1 月 19 日 03:14:07 时，BIOS 时钟计数将达到最大值界限或为 32 位整数值的界限。因此，在 2038 年 1 月 19 日 03:14:08 时，BIOS 将重置计算机时间为 1901 年 12 月 13 日 00:00:00。足够令人感兴趣的是，12 月 13 日正好也是星期五！幸运的是，

到那时我计划出航去南太平洋的某处。

对系统实现所使用的编程语言的了解也能有助于测试者把数据划分为离散子集。在本例中，所使用的编程语言为 C#。并且，在 C# 中所有的输入变量都以字符串类型进行提交。开发者必须采用 Int32.Parse 或 Convert.ToInt32 的方法将字符串转换为整数值。对于超出 32 位有符号整数所能表达范围的整数，这些方法都会抛出溢出异常。对于任何不能转换为整数的输入变量，这些方法都会抛出格式异常。因此，假如开发者使用其中一种方法把输入变量转换为整数值，你可以放心地断言：输入字符 A 跟表意字符“金”或拥有小数点或千位分隔符的数都是相同的非法等价子集。这是因为任何不能转换为整数的输入字符都会导致格式异常的抛出。空字符串也会抛出格式异常；但是，我建议把空字符串定义为输入参数的唯一子集来进行测试。

等价类划分测试

这个特例程序完整无漏的测试包括在 1/1/1582 到 12/31/3000 之间的所有合法日期。正向测试（positive tests）的总体数量将近 500,000。如果花费 5 秒进行手动输入和验证在合法日期范围内的所有合法日期的正确结果，那么这将会花费测试者近 29 天的时间来完整地测试合法输入。你可能没有足够的时间来测试所有可能的变量输入组合，这么做可能不是你时间的最佳使用。并且许多测试比先前测试提供新信息的可能性极低。使用 ECP 测试你可以得出的结论是：合法输入 6/12/2001 将会产生下一个阳历日期，可能是 9/29/1899 和 4/3/2999。

正如前面提到的，ECP 技术的一个好处是，它能帮助测试者从所有可能的测试中系统地减少测试数量。当数据划分为离散子集之后，ECP 技术的下一步是定义在测试之中如何使用数据子集。

最常用的方法是对于所有参数创建合法类子集的组合直至在验证测试中所有合法类子集已经至少包含一次。接着，对于每一个参数单独测试的每一个非法类子集，把其他参数设置为一些合法值。例如，对于合法输入数据的第一次测试可以是合法类子集 v1、v4(或 v6 或 v7) 和 v8。由这三个子集的组合所表示的测试数据是任何拥有 30 天的月份、任何在 1 到 30 之间的日和任何在 1582 到 3000 之间的年。表 5-2 列出了用于“下一个日期”程序的测试数据输入。

表 5-2 ECP 测试设计矩阵

ECP 测试	月	日	年	其他	期望的结果
1	$V_1 \cup V_2 \cup V_3$	V_6	V_8		下一日期
2	V_1	V_4	$V_9 \cap V_8$		下一日期
3	V_2	V_5	$V_{10} \cap V_8$		下一日期

4	V3	V6	$V11 \cap V8$	下一日期
5	V3	V7	$V12 \cap V8$	下一日期
6			V13	下一日期
7			V14	下一日期
8			V15	1/20/2038
9	i1	V4	V8	错误信息
10	i2	V4	V8	错误信息
11	i3	V4	V8	错误信息
12	i4	V4	V8	错误信息
13	i5	V4	V8	错误信息
14	$V1 \cup V2 \cup V3$	i6	V8	错误信息
15	$V1 \cup V2 \cup V3$	i7	V8	错误信息
16	$V1 \cup V2 \cup V3$	i8	V8	错误信息
17	$V1 \cup V2 \cup V3$	i9	V8	错误信息
18	$V1 \cup V2 \cup V3$	i10	V8	错误信息
19	$V1 \cup V2 \cup V3$	V6	i11	错误信息
20	$V1 \cup V2 \cup V3$	V6	i12	错误信息
21	$V1 \cup V2 \cup V3$	V6	i13	错误信息
22	$V1 \cup V2 \cup V3$	V6	i14	错误信息
23	$V1 \cup V2 \cup V3$	V6	i15	错误信息
24			I16	错误信息
25			I18	错误信息

本例中，ECP 技术阐释了关于有效输入的等价类测试数量可以从 500000 可能测试降低到 8 个正测试和 17 个负测试。由于无效子集 I17 的测试和无效子集 I19 的 ECP 测试产生了重复，所以最终只有 17 个负测试。同样，你也可能消除有效子集类 V13 的第 6 号测试，这是由于它和第 1、2、4、5 号 ECP 测试产生了重复。但是，你不应该消除这些 ECP 表中的冗余子集，因为当你进行边界值分析的时候，它们可以有助于识别潜在的边界条件。

验证测试的数量限制在 8 个似乎有些冒险。我也从不提倡只执行仅仅 8 个测试。你肯定想根据不同的输入元素来运行更多的正测试。利用随机概率生成测试数据或随机选取给定测试子集中的特定测试数据元素可以覆盖更多的可能变量数据。现在，我们来设计测试 1 的测试输入（或手工或自动的），你可以 1-12 月中随机选择一个月份（或者按序遍历每月），1-28 中随机选一个天数，从 1582-3000 中随机选一个年份。通过运行若干次测试并从测试子集中随机选择可变元素，你可以扩展所有子集中数据点的覆盖面，同样也可以设计能为测试员和自动化测试提供更大灵活度的测试，并且证明（或驳斥）测试假说。通过把每次遍历 ECP 测试 1 中随机选取的月、日、年保存在文件或数据库中，你可以将原算法作为一种预演，以此来验证本例中每个自动化 ECP 的输出。

ECP 测试为何不包括边界条件的测试？

注意ECP所有的测试都不专门针对边界条件。这是由于ECP测试是用于识别特定类型的错误，而边界测试是用于识别不同类型的错误。ECP表中已分解的数据子集可以识别在实际数值的线性范围中，极小和极大的边界条件，也有助于识别在一组数值中的特定实际值，它可能代表一种特定情况的边界条件。而ECP技术通过填入的可能数据来寻找存在于普通可变

数据和盖然数据样本中的问题。ECP测试不专门针对极大极小边界条件，同样它也不包括边界值分析和边界测试。

我常常看见许多测试新手试图通过边界测试和 ECP 技术来减少测试量，节约时间。虽然 ECP 技术常常与边界测试结合使用，因为 ECP 子集提供了有助于识别潜在边界条件的基本框架，但是试图绑定两者技术是很危险的，因为这样会导致测试的遗漏以及错误的假设。总体而言，当应用测试技术来验证软件时，最好在一定时期内只专注于一项技术，以达到最好的有效性。

等价类划分小结

ECP 是个用于设计一组黑盒或白盒的功能性测试技术，并以此来评估输入输出参数的功能性。该技术不能用于评估边界条件、相互依赖参数的组合、按序或已经排序的输入。输入和输出条件可以概括为 2 大类别：有效数据类，他会返回正向的结果积极的结果，或不产生错误条件和失效数据类，他通常会产生错误条件。每个类的数据会进一步分解成子类。在测试中，任何来自于给定类中特定子类的数据元素，无论它所在的子集是否被使用，都会产生同样的结果。现在，有 4 个有用的分解数据的启发式方法：数值的范围、变量的相似组、唯一数值和特定数值。等价类划分是个有效减少测试数量的好方法。当结合随机生成数据之后，ECP 测试执行可以扩展除静态变量以外的覆盖面，同时测试健壮性也能增加整体的可信度。

虽然相比于简单地插入可能的变量和一些之前证明有疑问的变量而言，ECP 技术的应用需要花费更多的时间，但是这对于等价类的划分非常有益。当该技术被正确应用时，它可以做到如下：

- 迫使测试人员对功能集和用于输入输出参数的变量数据进行更细致的分析。
- 帮助测试人员识别之前被忽视的边角案例
- 提供明显的事实，以表明哪些数据集被测试过，如何测试的
- 系统性地增加测试的有效性，以有助于减少风险
- 通过逻辑性地减少冗余测试来提高效率

然而，等价类的划分不是银弹，ECP 技术检测软件中异常的有效性和覆盖面的增加主要取决于测试人员的技术以及领域空间和全局系统知识的掌握程度。如果被合理使用，ECP 技术是个测试工具箱中有价值的工具，而且可以暴露边角案例或不被其他测试方法所使用的测试数据。

单点故障的假设

在可靠理论中的单点故障假设规定中，故障很少是2个或多个故障共同作用的结果。因此，ECP测试和边界测试的主要目的是暴露单点故障，尤其是处在线性边界的变量或处在所谓边角案例的等价数据可能被程序相应地处理。当把其他参数设定到有效的条件下，ECP 和BVA技术都提供了更为严格的检查和对离散输入输出参数的系统评估机制。当然，这不是唯一的、需要考虑的故障模式，但是当你对每个参数独特的职能功能有信心的时候，你可以

轻易地包括其他测试来评估参数间的相互作用和被怀疑容易产生多故障模型的调查区域。

边界值分析

边界值分析（BVA）可能是最著名的功能性测试技术。不幸的是，由于许多测试者认为边界值分析相对比较简单或者微不足道的，这技术常被误用。历史经验表明，相当多的问题出现在线性变量的边界值上，所以为了防止忽略边界类的缺陷，我们必须仔细分析线性变量的边界条件。当 BVA 与等价类划分结合使用，则 BVA 这个功能性技术可以有效地分析独立输入和输出参数等线性参数的边界值。BVA 或边界测试是特别有用的检测以下类型的错误：

- 错误的数据类型的人工约束
- 错误地分配关系运算符
- 数据类型的封装
- 差一(Off-by-one)错误

什么是软件中的边界值？

在一个软件程序的环境中，边界值是个特定值，该数值处在独立线性变量或它的等价类子集的极限边界上。

和如何定义一个国家实际领土的边界一样，边界值是在线性变量极限范围上的一些特定数据。例如，在下个日期程序中，最小的输入日期是 1/1/1582，而最大边界是 12/31/3000。这些数值分别代表极限最大和最小边界。最小的输入 1 和最大的输入 12 是月份参数的极限范围。

许多国家被分成州和领域哪里。同样的，线性在最小和最大范围中也可以有子边界值。这些子边界值可用来界定各种等价类划分范围集。例如，表 5-1 的 ECP 数据表示 10/5/1582 到 10/14/1582 之间的范围。该 ECP 范围也表示允许输入日期范围中的另外两个子边界值。

许多国家占有一个大洲的某些物理空间，但是并不完整。同样，虽然年份参数的输入变量已预先定义在 1582 和 3000，开发者必须声明变量是某种数据类型。许多的工具可以帮助测试者来决定函数中变量的数据类型，但是假设开发者在 C# 中定义了一个 16 位的整型来保存年份变量。16 位整型的范围是从 -32,768

到 +32,767。显然，开发者人为地限制把 16 位整型的全局线性范围限制在 1582 到 3000 之间（理想的情况是，开发者也可以修改输入文本框控制的属性，只允许 4 个字符）。因此，虽然年份参数可以接受任何在范围内的带符号 16 位整型数值，但是实际可接受的范围已经人为地限制在 1582 到 3000 的范围内。

BVA 在系统地分析在软件中常见的两类边界值：不变常数的数值，固定变量数值。不变边界常数是数学常数或在运行阶段保持不变而且通常不会被用户改变的实数。例如，在下个日期程序中，1582 和 3000 是预先定义的常数，它指定了允许范围内的年度输入变量。在下个日期程序中，另一个预先定义的常量是用户填入的年份文本框中最大的数值。我们已经预先定义年份文本框的属性，并只允许用户只输入 4 个字符。

边界值的第二个类型是固定边界，它意味着测量可以被改变但是在特定时段必须不变。例如，窗口的高度和宽度可以延 x 轴和 y 轴改变大小；但在任何一个时段，他们都是可被像素，毫米或英寸测量的固定线性值。由于线性测量在用户界面上不是随时可见的，因而固定变量数值难以被识别。图 5-3 表明在微软画图板中，关于帆布高度和宽度属性的固定边界

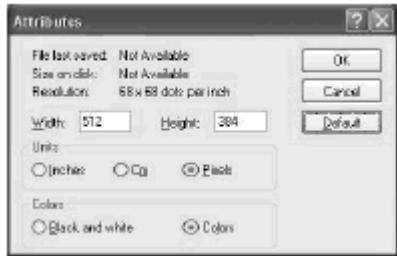


图 5-3 宽度和高度变量，关于在运行期间可被用户修改固定边界的例子

边界测试的定义

Paul Jorgensen²⁰建议用于基本边界值分析的测试数量可以用 $4n+1$ 来计算（或者 $6n+1$ 包括健壮性边界测试的最小值-1 和最大值+1 数值），这里的 n 等于独立参数的个数。Jorgensen 的公式包括了一个数值测试。虽然该公式在分析简单独立参数的极端线性变量范围非常有用，但是对于复杂情况下，该公式显得过于简单了。

例如，更全面的数据分析表明 Jorgensen 的公式会无意间忽视了关键的边界条件，它们确定的等价类子集代表了线性变量在极小和极大范围内的一组数值。而且，等价类子集中的唯一数值可能不是线性变量的极限范围，或者相比于其他数值，一个特殊的等价类更有益去分析。例如，图 5-4 说明了微软视窗 874ANSI 的代码页。该 ANSI 码的字符码点范围是从 0x00 到 0xFF。

然而，如果输入参数明确不使用未赋值的字符码点，则要求若干有效和无效的等价类分配关于已赋值和未赋值字符码点的取值范围，如表 5-3 所示。虽然全局的范围已经扩展至 0x00 到 0xFF，但如果你不仔细分析未赋值和已赋值字符码的边界，你很可能忽视类似的潜在问题。

²⁰ Paul C. Jorgensen, Software Testing: A Craftsman's Approach (Boca Raton, FL: CRC Press, 1995).

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	NUL 0000	STX 0001	SOT 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
10	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
20	SP 0020	! 0021	" 0022	# 0023	\$ 0024	% 0025	& 0026	' 0027	(0028) 0029	*	+	,	-	.	/
30	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	:	;	<	=	>	?
40	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
50	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C] 005D	^ 005E	— 005F
60	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
70	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	 007C	} 007D	~ 007E	DEL 007F
80	€ 20AC 2026
90	„ 2018	„ 2019	„ 201C	„ 201D	„ 2022	„ 2013	„ 2014	„ 2015	„ 2016	„ 2017	„ 2018	„ 2019	„ 201A	„ 201B	„ 201C
A0	NBSP 00A0	ؐ 0E01	ؑ 0E02	ؒ 0E03	ؓ 0E04	ؔ 0E05	ؕ 0E06	ؖ 0E07	ؗ 0E08	ؘ 0E09	ؙ 0E0A	ؚ 0E0B	؛ 0E0C	؜ 0E0D	؝ 0E0E	؞ 0E0F
B0	ؠ 0E10	ء 0E11	آ 0E12	أ 0E13	ؤ 0E14	إ 0E15	ئ 0E16	ا 0E17	ب 0E18	ة 0E19	ت 0E1A	ث 0E1B	ج 0E1C	ح 0E1D	خ 0E1E	د 0E1F
C0	ؠ 0E20	ء 0E21	آ 0E22	أ 0E23	ؤ 0E24	إ 0E25	ئ 0E26	ا 0E27	ب 0E28	ة 0E29	ت 0E2A	ث 0E2B	ج 0E2C	ح 0E2D	خ 0E2E	د 0E2F
D0	؊ 0E30	؋ 0E31	، 0E32	؍ 0E33	؎ 0E34	؏ 0E35	ؐ 0E36	ؑ 0E37	ؒ 0E38	ؔ 0E39	ؕ 0E3A	ؖ 0E3B	ؖ 0E3C	ؖ 0E3D	ؖ 0E3E	ؖ 0E3F
E0	؄ 0E40	؅ 0E41	؆ 0E42	؇ 0E43	؈ 0E44	؉ 0E45	؊ 0E46	؋ 0E47	، 0E48	؍ 0E49	؎ 0E4A	؏ 0E4B	ؐ 0E4C	ؑ 0E4D	ؒ 0E4E	ؔ 0E4F
F0	؀ 0E50	؁ 0E51	؂ 0E52	؃ 0E53	؄ 0E54	؅ 0E55	؆ 0E56	؇ 0E57	؈ 0E58	؉ 0E59	؊ 0E5A	؋ 0E5B	، 0E5C	؍ 0E5D	؎ 0E5E	؏ 0E5F

图 5-4 视窗 ANSI 码的第 874 页

表 5-3 关于已赋值和未赋值字符码点的 ECP 子集

ANSI 代码页 874	
已赋值字符码点	未赋值字符码点
0x00-0x80	0x81-0x84
0x85	0x86-0x90
0x91-0x97	0x98-0x9F
0xA0-0xDA	0xDB-0xDE
0xDF-0xFB	0xFC-0xFF

边界值分析：一个全新的公式

改良地分析那些可能并不处在独立参数极限范围内的边界值，这要求一个可选的方式和一个用于计算边界值分析的最小测试子集的、不同的公式。需要明确的是，有效边界值分析取决于测试员准确识别关于任何特定独立参数的所有特定边界条件的能力。等价类子集对于

帮助我们确定潜在的边界条件也非常有用。ECP 子集不仅揭示了数值的极限线性范围，而且如果所有的变量数据都被正确分解，则 ECP 子集也会暴露一些可能表明额外重要边界条件的数值。

在确定所有独立输入输出参数的边界条件之后，我们可以通过一个简单的公式：3(BC) 来计算最小的测试数据子集，这里的 BC 等于特定边界条件的数量。为了说明 Jorgensen 健壮性公式和 3(BC)公式的区别，我们可以通过下一日期程序来比较这两种计算方式。

月份、日期和年份的极限输入数值分别是 1 和 12, 1 和 31 以及 1582 和 3000。用 Jorgensen 的 $6n+1$ 公式来进行健壮性边界测试，输入变量的边界测试数量是 $(6*3) + 1$ 或 19 个测试，在 BVA 测试矩阵中具体表示，见表 5-4。

表 5-4 基于 Jorgensen 健壮性 $6n+1$ 公式的简单 BVA 测试矩阵

测试	月	日	年	期望的结果	备注
1	0	1-28	1582-3000	错误	月最小值-1, 日 标定的, 年 标定的
2	1	1-28	1582-3000	下一日	月最小值, 日 标定的, 年 标定的
3	2	1-28	1582-3000	月最小值+1, 日 标定的, 年 标定的	
4	11	1-28	1582-3000	月最大值-1, 日 标定的, 年 标定的	
5	12	1-28	1582-3000	月最大值, 日 标定的, 年 标定的	
6	13	1-28	1582-3000	月最大值+1, 日 标定的, 年 标定的	
7	1-12	0	1582-3000	日最小值-1, 月 标定的, 年 标定的	
8	1-12	1	1582-3000	日最小值, 月 标定的, 年 标定的	
9	1-12	2	1582-3000	日最小值+1, 月 标定的, 年 标定的	
10	1-12	30	1582-3000	日最大值-1, 月 标定的, 年 标定的	
11	1-12	31	1582-3000	日最大值, 月 标定的, 年 标定的	
12	1-12	32	1582-3000	日最大值-1, 月 标定的, 年 标定的	
13	1-12	1-28	1581	年最小值-1, 日 标定的, 月 标定的	
14	1-12	1-28	1582	年最小值, 日 标定的, 月 标定的	
15	1-12	1-28	1583	年最小值+1, 日 标定的, 月 标定的	
16	1-12	1-28	2999	年最大值-1, 日 标定的, 月 标定的	
17	1-12	1-28	3000	年最大值, 日 标定的, 月 标定的	
18	1-12	1-28	3001	年最大值-1, 日 标定的, 月 标定的	
19	1-12	1-28	1582-3000	都是标定的条件	

然而，如表 5-1 所示，利用等价类进行数据的分解只明确了一些额外的、无需考虑的边界数值。例如，日期的输入参数有 3 个额外边界条件，其最大范围是 31。对于有效和无效的日期范围，也有 4 个额外的边界条件。而输出参数也有 2 个边界条件。将 3(BC) 公式应用到边界测试中，揭示了边界测试测试需要更充分地分析每个参数的、潜在边界条件，因此，将测试集增加到 $(3*18)$ 或 54，如表 5-5 的 BVA 矩阵所示。

表 5-5 关于应用 3(BV)公式的下一日期程序的 BVA 测试矩阵

BVA	边界 测试	月 条件	日	年	期望的	备注
1	1	0	1-28	1582-3000	错误信息	月 最小值-1
2		1	1-28	1582-3000	下一日期	月 最小值
3		2	1-28	1582-3000	下一日期	月 最小值+1
4	2	11	1-28	1582-3000	下一日期	月 最小值-1
5		12	1-28	1582-3000	下一日期	月 最小值
6		13	1-28	1582-3000	错误信息	月 最小值+1
7	3	31 天 的月份	0	1582-3000	错误信息	日 最小值-1 (31 天的月份)
8		31 天 的月份	1	1582-3000	下一日期	日 最小值 (31 天的月份)
9		31 天 的月份	2	1582-3000	下一日期	日 最小值+1 (31 天的月份)
10	4	31 天 的月份	30	1582-3000	下一日期	日 最大值-1 (31 天的月份)
11		31 天 的月份	31	1582-3000	下一日期	日 最大值 (31 天的月份)
12		31 天 的月份	32	1582-3000	错误信息	日 最大值+1 (31 天的月份)
13	5	30 天 的月份	0	1582-3000	错误信息	日 最小值-1 (30 天的月份)
14		30 天 的月份	1	1582-3000	下一日期	日 最小值 (30 天的月份)
15		30 天 的月份	2	1582-3000	下一日期	日 最小值+1 (30 天的月份)
16	6	30 天 的月份	29	1582-3000	下一日期	日 最大值-1 (30 天的月份)
17		30 天 的月份	30	1582-3000	下一日期	日 最大值 (30 天的月份)
18		30 天 的月份	31	1582-3000	错误信息	日 最大值+1 (30 天的月份)
19	7	2	0	闰年	错误信息	日 最小值-1 (闰年区间)
20		2	1	闰年	下一日期	日 最小值 (闰年区间)
21		2	2	闰年	下一日期	日 最小值+1 (闰年区间)
22	8	2	28	闰年	下一日期	日 最大值-1 (闰年区间)
23		2	29	闰年	下一日期	日 最大值 (闰年区间)
24		2	30	闰年	错误信息	日 最大值+1 (闰年区间)
25	9	2	0	非闰年	错误信息	日 最小值-1 (非闰年)
26		2	1	非闰年	下一日期	日 最小值 (非闰年)
27		2	2	非闰年	下一日期	日 最小值+1 (非闰年)
28	10	2	27	非闰年	下一日期	日 最大值-1 (非闰年)

29	2	28	非闰年	下一日期	日 最大值 (非闰年)
30	2	29	非闰年	错误信息	日 最大值+1 (非闰年)
31	11	1-12	1-28	1581	错误信息 年 最小值-1
32		1-12	1-28	1582	下一日期 年 最小值
33		1-12	1-28	1583	下一日期 年 最小值+1
34	12	1-12	1-28	2999	下一日期 年 最大值-1
35		1-12	1-28	3000	下一日期 年 最大值
36		1-12	1-28	3001	错误信息 年 最大值+1
37	13	12	31	1581	错误信息 输出 最小值-1
38		1	1	1582	输出 最小值
39		3	2	1582	3/3/1582 输出 最小值+1
40	14	12	30	3000	12/31/1582 输出 最大值-1
41		12	31	3000	1/1/3001 输出 最大值
42			1	3001	错误信息 输出 最大值+1
43	15		2	1752	独特的日期区间 最小值-1
44			3	1752	9/4/1752 独特的日期区间 最小值
45			4	1752	9/5/1752 独特的日期区间 最小值+1
46	16		12	1752	9/13/1752 独特的日期区间 最大值-1
47			13	1752	9/14/1752 独特的日期区间 最大值
48			14	1752	9/15/1752 独特的日期区间 最大值+1
49	17		4	1582	10/15/1582 独特的日期区间 最小值-1
50			5	1582	错误信息 独特的日期区间 最小值
51			6	1582	错误信息 独特的日期区间 最小值+1
52	18		13	1582	错误信息 独特的日期区间 最大值-1
53			14	1582	错误信息 独特的日期区间 最大值
54			15	1582	10/16/1582 独特的日期区间 最大值+1

两个例子中，当其他参数都被设定成额定值时，每个参数都被评估为边界值和忽上忽下的数值。这些参数的 ECP 测试已经证实额定数值可以如预期地执行。这里再次强调，当额定数值给定，我们设计 BVA 测试来使用有效范围内的随机概率数值，如此可以提供更大的灵活度，并使得测试中的硬编码或静态数据最小化。在某些案例中，边界测试必须指定变量的组合来驱动测试评估特定的边界。当随机测试数据被用作变量的额定数值时，测试人员吧必须依靠他们的直觉来决定测试的输出，而自动化测试可以安心地依靠原算法作为比较实际结果和预期结果的预演。

隐性边界

不是所有的边界都能通过数字型输入或输出来确定，或者它们是代表直接面向用户的线性测量数据。一个视窗控件的宽度和高度可以通过像素或其他线性方式来测量。另外，代码中存在许多的循环算法，循环结构在边界条件中是众所周知的问题。例如，以下的函数统计字符串中的字符数目（事实上，它统计Unicode字符码点的个数）。该函数的边界测试包括 每次遇见空字符串（就是最小-1）就绕过循环一次，然后进过一个字符（最小）的字符串和两个字符（最小+1）的字符串。分析最大范围的测试包括带有2,147,483,646字符（最

大 - 1), 2,147,483,647 characters (最大) 和 2,147,483,648 字符 (最大+1)。 ToCharArray 函数将 String 转成了字符数组，复制了最多的 Unicode 字符，相当于一个带符号的 32 位整型数据。因此，虽然这极不可能发生在实际应用中，将一个由 1,073,741,824 个 Unicode 代理配对字符组成的 string 数据传给一个函数，cArray 似乎有 2 个 Unicode 码点组成，而当下标可以增至 2,147,483,648，程序就会抛出越界的异常。

```
private static int GetCharacterCount(string myString)
{
    try
    {
        char[] cArray = myString.ToCharArray();
        int index = 0;
        while (index < cArray.Length)
        {
            index++;
        }
        return index;
    }
    catch (ArgumentOutOfRangeException)
    {
        throw;
    }
}
```

但是，有时候我们很难确定循环结构的边界。熟悉编程语言、数据类型和算法结构都会帮助测试人员深度挖掘那些隐性的边界。在之前的例子中，如果你不清楚不同的 Unicode 编码模式（尤其是代理配对编码），然后只要通过简单 Unicode 字符测试极限边界条件，那么该函数会以一串字符串的形式返回正确的字符数，这串字符串包含的字符数长度小于等于 2,147,483,647。但是，在传递带有 147,483,648 字符的字符串 (String) 数据时，甚至任意一个字符都是代理配对，都会抛出越界的异常。

边界测试和 the déjà vu heuristic

循环是个软件中常用的结构，而且（取决于编程语言）很容易受到边界缺陷的影响。循环结构的边界值分析包括（最小）绕过循环，一次遍历循环，2 次遍历循环，最大次数地遍历循环，最大次数-1 地遍历循环，最后是超过最大次数地遍历循环。当仅仅通过一个黑盒测试设计方案来制定测试，很难确定循环的结构。

在视窗 XP 中，用户总是试着用预先设定的设备名 (LPT1, COM) 作为基本文件名来保存文件而扩展名.txt 作为文件名编辑控制器的参数，系统显示表明文件已存在的错误信息，“你想替换它吗？”。当然，你不能保存与视窗系统中已保存设备同名的文件，因此当我在测试该条件时，我在对话框中点击了“YES”。该文档视窗的标题改变为 LPT1-Notepad；但是，文件并未被保存。当我关闭该文件之后，内容就丢失了。视窗 XP 的更新版试图修正这个缺陷。但是，边界测试表明该问题只能被部分修正。

修正之后，我重复之前同样的步骤，但是当我点击“YES”按钮之后，新的错误信息会显示“无法创建[LPT1]文件”。可以确定的是路径和文件名都是正确的。在我点击“OK”之后，控制进程返回到记事本。每次我遇见一个错误信息，我多次重复同样的步骤，以此确认我会获得同样的错误信息。猜猜结果如何？第二次我遍历同样的路径，获得同样的错误信息，这表明在修正之

前，我们都遇到同样的问题。

我是如何了解在保存对话框中需要测试循环程序的？很简单，每次我遇见一个错误信息，我都会采用拇指理论，称作 *the déjà vu heuristic*（启发式的方法是意图增加解决一些问题可能性的通用规则（或规则集）。这种方法通常会增值，所以非常有价值，但也可能出错。）*the déjà vu heuristic* 特地定义了最小边界值和忽上忽下于边界值的数值。假如这样，最小值-1 的数值不能执行错误分支，最小边界条件正在执行初始化错误信息的分支，而最小值+1 的数值反复执行相同的步骤来合理地遍历代码中的相同路径。

边界值分析小结

边界值分析是个以介于特定边界值上下的数据作为研究目标的功能性技术。历史经验和递归问题的根本原因分析表明异常总是发生在独立输入输出参数的边界条件附近。对处在边界条件的变量进行系统分析增加测试的有效性，提供更优的信息，增加检测特定类缺陷的可能性。这种缺陷可能在早期测试周期中，源于输入或输出参数的线性变化。

结合 3 (BC) 公式，你可以更好地估计所需边界测试的数量，并相比于其他公式，可以提供更广泛的测试集合。因为测试都是基于确定的边界数值，而不是基于参数的数量。如果只关注实际参数的极限范围或通过对额定值的随机输入，那么只能通过更具体的数据检查才能找出更多的边界条件。为了合格地评估独立参数的变化，边界测试提供了更为精确的系统性技术。然而，你必须注意测试中没有哪个单一技术是完美无缺的。边界值分析的有效性取决于我们是否有能力将离散变量分解成等价类，并确定重要的边界条件。而且，基本边界测试是基于单点故障的前提。基于以下的前提，BVA 测试常常不能有效地评估依赖或半依赖参数的复杂组合。

组合分析

迄今为止，我已经重点测试每个输入或输出参数的离散功能，并同时确保其他参数的状态处于有效范围内。然而，一些参数的可变状态是依赖于，或者至少是半耦合于 (semicouple) 其他参数的可变状态。这就意味着一个参数的输出状态取决于输入参数可变状态的各种组合。为了系统地测试几个相关参数可变状态间的互相作用，相对其他可选策略而言，组合分析是最优方法。

为了正确地认识上述观点，我们来审视一下微软 IE 浏览器中的安全设定对话框。该对话框有若干参数。如果你假定所有的参数都是半耦合的，则穷举测试的总量等于所有可变状态数的笛卡尔乘积。而且如何按每毫秒完成一个测试的速度计算，大约需要 3,300 年才能完成所有的测试。显然，只有在次要的程序中，才有可能穷举组合测试。但是，为了更有效地测试复杂变量状态的互动性，业界经验明确地表明组合分析师最有效的解决方案。

组合分析是被微软的测试工程师普遍使用的功能测试技术，通过从所有可能的组合中选择一个有效的子测试系统，并借此分析一个复杂的特征集中的系统性依赖和半耦合参数的相互作用。

当在合适环境下被正确使用时，组合分析技术有许多优点。这些优点包括：

- 能最大程度地识别由于变量作用所导致的缺陷
- 提供了更大的结构性覆盖
- 具有很大的潜力来降低整体测试成本（如果使用得当）

重要的是：该技术的优势是用于你在测试一个功能时，该功能的参数直接相互依赖或者半耦合，而且参数输入都是无序的。有时，新测试人员会尝试应用该技术来测试一组参数或者输入集，而无需进行全面的特性分析。然而，你必须理解本技术不适用于任何多重输入的程序或者功能中。组合分析并不是一个测试独立参数集，这参数集中没有直接或间接的相互作用，数学计算，有序参数输入或者需要顺序操作的输入。

组合测试的途径

目前，通常有两种测试参数相互作用的途径。第一种途径一般涉及随机或者特定的方法，而第二种包括了更加系统的过程。随机评估的方法包括最优猜测或特定测试和随机选择。而系统性评估方法包括测试每个变量或者分支（EC），基准测试（BC），正交阵列(OA)，组合测试（经过 n -对的配对或者 $t=n$ ）和穷举测试（AC）。

最优猜测或特定方法主要依靠测试人员的直觉和运气。这种方法比较适合测试常用的组合或者快乐路径，也能暴露源于异常组合情境下的细微缺陷。然而，在可控环境下的实验都表明测试者如果轻易能达到覆盖率和额外测试的极限量，往往是执行相同的代码路径，对于全局的测试工作而言意义不大。随机选择方法通过从所有可能的组合中选择一个有效的子测试集合。Schroeder 和他同事²¹最近的一份研究发现 n -对 (n -wise) 测试和从所有可能的组合中随机选择一套组合测试，就故障检测的效力而言，并无明显差别。从所有可能的组合中随机选择的测试会包括这样的测试，该测试涉及互斥的参数或者特定参数必须是不同的常规条件。这方法从学术的角度来看非常有趣，但是用于测试商业软件不太实用，尤其现在涌现了更多高效的工具。

分支测试(EC)就是将每个变量测试至少一次。相比于任何其他的组合分析途径，EC 使用了最少量的测试。然而，通常它对于任何复杂系统都是无效的。基准测试（BC）指定一组变量作为基准测试。这组变量通常是快乐路径中常用变量状态的组合。额外的测试在一段时间内改变一个参数的变量状态，同时保持在基准测试中的其他参数变量状态不变。BC 测试满足 $t=1$ 或 1-对（1-wise）的覆盖，而且可以有效地检测单个的组合错误。但是，一些研究表明当与 n -对组合测试相结合时，BC 测试非常有效。

正交阵列(OA)包括工业制造所采用的步骤。在软件测试中正交阵列的使用是个举足轻重的步骤。一个简单的正交阵列方法需要每个相互依赖的参数具有同等数量的变量状态，那些状态都对应在数组中。而在那些独立参数的可变状态数量不尽相同的较复杂功能中，正交阵列的选择就变得更为复杂了。正交阵列的输出却相对只是配对覆盖。而且，正交阵列的输出并不是最理想的，那是因为它包括了每代测试的每个元组，这就导致配对测试组合的冗余。正交阵列非常有益于实验、性能分析和优化的开展。但是，正交阵列是个应对复杂问题的复杂解决方案。而且鉴于更多有效的关于组合测试的解决方案，正交阵列对于相互独立参数的

⁷ Patrick J. Schroeder, Pankaj Bolaki, and Vijayram Gopu, Comparing the Fault Detection Effectiveness of N-Way and Random Test

功能测试不太实际。

测试相互依赖的参数可变组合最有效的解决方案之一是组合分析或用到覆盖阵列的 n -对 (n -wise) 测试。有些组合分析工具是可供利用的，而且我们可以采用多种算法来产生一个组合测试矩阵。为了基本认识一个通用的覆盖阵列算法在配对分析中的应用，我们假设单个字体的字体属性可以是粗体、斜体、删除线或者是下划线的组合。在这个例子中，有 4 个参数（粗体、斜体、删除线和下划线），而且每个参数都有 2 个可变状态（选中和没选中）。

这是个简单的配对或 2-对分析，首先我们从所有可能的组合中，识别粗体和斜体参数组合的特有可变状态。下图中，c 代表选中，u 代表没选中。

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Bold	c	c	c	c	u	u	u	c	u	u	c	c	u	c	u	u
Italic	u	c	c	c	c	c	c	c	u	u	u	u	u	u	c	u
Underline	u	u	c	c	u	c	c	u	c	c	c	c	u	u	u	u
Strikethrough	u	u	u	c	u	u	c	c	u	c	u	c	c	c	c	u

其次，我们从所有可能的组合中，识别相应之后配对参数的可变状态组合，并将这些可变状态集合与之前其他参数组对的测试集合重新组合，如下表所示。

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Bold	c	c	c	c	u	u	u	c	u	u	c	c	u	c	u	u
Italic	u	c	c	c	c	c	c	c	u	u	u	u	u	u	c	u
Underline	u	u	c	c	u	c	c	u	c	c	c	c	u	u	u	u
Strikethrough	u	u	u	c	u	u	c	c	u	c	u	c	c	c	c	u
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Bold	c	c	c	c	u	u	u	c	u	u	c	c	u	c	u	u
Italic	u	c	c	c	c	c	c	c	u	u	u	u	u	u	c	u
Underline	u	u	c	c	u	c	c	u	c	c	c	c	u	u	u	u
Strikethrough	u	u	u	c	u	u	c	c	u	c	u	c	c	c	c	u
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Bold	c	c	c	c	u	u	u	c	u	u	c	c	u	c	u	u
Italic	u	c	c	c	c	c	c	c	u	u	u	u	u	u	c	u
Underline	u	u	c	c	u	c	c	u	c	c	c	c	u	u	u	u
Strikethrough	u	u	u	c	u	u	c	c	u	c	u	c	c	c	c	u
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Bold	c	c	c	c	u	u	u	c	u	u	c	c	u	c	u	u
Italic	u	c	c	c	c	c	c	c	u	u	u	u	u	u	c	u
Underline	u	u	c	c	u	c	c	u	c	c	c	c	u	u	u	u
Strikethrough	u	u	u	c	u	u	c	c	u	c	u	c	c	c	c	u
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Bold	c	c	c	c	u	u	u	c	u	u	c	c	u	c	u	u
Italic	u	c	c	c	c	c	c	c	u	u	u	u	u	u	c	u
Underline	u	u	c	c	u	c	c	u	c	c	c	c	u	u	u	u
Strikethrough	u	u	u	c	u	u	c	c	u	c	u	c	c	c	c	u

我们结合各表来看，从 16 个可能的组合测试中揭示了 5 个测试的矩阵（数字 2, 7, 11, 14 和 16）。这 5 个测试有效地将每个参数至少测试了一遍。注意这个测试集合并不包括所有字体都选中的测试，而且如果测试人员认为该测试比较重要，那他肯定会把该测试加入到输出矩阵中。

组合分析实践

用来测试 n 对参数作用组合的工具只是个辅助性的，它完全依赖于测试人员的对于整个系统的全面认识，如此才能最有效。任何工具的输出受限于工具的性能以及使用者所掌握的技术和知识。被未受过培训的或者不熟练的人所使用的工具显然是无效的，甚至可以说是愚蠢的工具；同样，未受过培训的或者不熟练的人来使用工具，说明他不知道这是个愚蠢的错误。微软的软件测试工程师主要用配对独立组合测试（PICT）工具来进行相互依赖参数的组合测试。PICT 是个高度定制的工具，克服了其他工具的许多限制。相对于手工生成测试组合而言，它能帮助测试人员在较少的时间内设计出更有效的测试。

但是，使用 PICT(或任何测试工具)和组合分析技术的仪器不只是鉴别输入参数和可变状态，并将这些数据导入工具这么简单的。整个过程需要测试人员对测试功能进行深度的分析，

并识别直接相关或半耦合的参数，分解这些参数，对每个相互依赖的参数合理定义可变状态。测试人员完成功能的全面分析和定义每个参数的可变状态之后，他就为确定一套组合测试的系统步骤做好了准备。

1. 新建 BC 测试的输入矩阵
2. 新建关于通用或可能的场景和历史性故障指标的输入矩阵
3. 为每个参数新建可变参数的输入模型文件
4. 定制模型文件来排除互斥的可变状态或固定的参数
5. 将输入文件应用到 PICT 工具上
6. 复审工具的输出信息
7. 如果有必要。提炼和定制模型文件
8. 将输入文件重新导入 PICT 工具
9. 重新验证输出信息
10. 执行测试

你可以通过按序检查每个步骤来理解艺术这套技术是如何在相对简答的仿真中应用的。假设你在测试简单字体的对话框，如图 5-5 所示。该对话框允许用户从 4 个可能的字体中选择一个，在粗体和斜体中 2 选 1，删除线和下划线中 2 选 1，字体颜色（黑色、白色、红色、绿色、蓝色或黄色）6 选 1 以及选择字体大小（1-1,638 包括半号尺寸）。如果穷尽这些组合，那测试总量是 1,257,600（既定每个字体大小都要被测试）。测试所有的组合是不可能的，也是不可行的，因此你必须选择一个适当的测试集合，来有效地降低全局的风险。

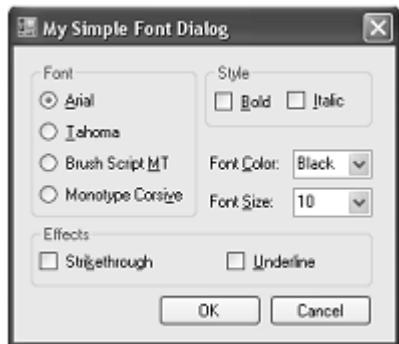


图 5-5 简单字体对话框来图解直接相关和半耦合参数的识别

首先，识别相互依赖的参数。本例中，直接相关的参数是铅字和字形，因为某些铅字不可能只是粗体或者斜体，它也有可能是既是粗体也是斜体，或者只是粗体，或者只是斜体，或者都不是。字体大小，颜色和效果是半耦合的参数，因为他们可以应用到所有字形中，并相应地影响输出结果。字形的可变状态是 Arial, Tahoma, Brush Script MT 和 Monotype Corsive。

这些字体和效果都有 2 个参数（粗体和斜体，以及删除线和下划线），而且每个参数都有 2 个可变状态（选中或未选中）。字体颜色有 6 种（黑色，白色，红色，绿色，蓝色和黄色）。字体大小有 3275 种可能值。测试所有的数值是不可能的，因此你从每个 ECP 的子集中，构建同等范围的数值，并随机选择一个来进行测试。

字体大小参数根据等价类子集定义了 5 个可变状态，该子集包括小号（字体大小 1-9.5），常规号（字体大小 10-12），大号（字体大小 12.5-72），超大号（字体大小 72.5-1638）和半号（字体大小 1.5-1637.5）。这样，可以确保测试组合中至少有一个测试，它的字体大小不是整型。通过创建字体大小范围的等价类子集，测试设计者为测试人员（或者设计优良的自动化测试）提供了较大的灵活度，使得他们可以在一个给定组合中的特定范围内来选取一个测试数值。

本例中，将字体大小划分成等价类的子集也排除了硬编码测试数值，这些数值将测试数据无谓地限制在一个范围较小的、良莠不齐的数据集合中。

当彻底分析功能，决定哪些参数是相互依赖，并分解每个参数的可变状态，进而选择合适的变量来测试之后，你就可以建立 BC 矩阵，如图 5-6 所示。BC 矩阵非常重要，这是由于研究表明在关联较高级别的 n-对测试，基准选择的元素会更有可能暴露更多潜在的，源于参数相互作用的缺陷，而配对或 n-对组合测试不一定包括在 BC 组合中。BC 矩阵通常定义了最通用的可变状态组合，然后在保持其他参数可变状态处于初始状态的同时，每次改变一个参数的可变状态，直到每个变量都被测试过为止。

	A	B	C	D	E	F	G
1	Font	Bold	Italic	Strikethrough	Underline	Color	Size
2	Arial	Uncheck	Uncheck	Uncheck	Uncheck	Black	Nominal
3	Tahoma	Uncheck	Uncheck	Uncheck	Uncheck	Black	Nominal
4	BrushScript	Uncheck	Check	Uncheck	Uncheck	Black	Nominal
5	Monotype	Uncheck	Uncheck	Uncheck	Uncheck	Black	Nominal
6	Arial	Check	Uncheck	Uncheck	Uncheck	Black	Nominal
7	Arial	Uncheck	Check	Uncheck	Uncheck	Black	Nominal
8	Arial	Uncheck	Uncheck	Check	Uncheck	Black	Nominal
9	Arial	Uncheck	Uncheck	Uncheck	Check	Black	Nominal
10	Arial	Uncheck	Uncheck	Uncheck	Uncheck	White	Nominal
11	Arial	Uncheck	Uncheck	Uncheck	Uncheck	Red	Nominal
12	Arial	Uncheck	Uncheck	Uncheck	Uncheck	Green	Nominal
13	Arial	Uncheck	Uncheck	Uncheck	Uncheck	Blue	Nominal
14	Arial	Uncheck	Uncheck	Uncheck	Uncheck	Yellow	Nominal
15	Arial	Uncheck	Uncheck	Uncheck	Uncheck	Black	Small
16	Arial	Uncheck	Uncheck	Uncheck	Uncheck	Black	Large
17	Arial	Uncheck	Uncheck	Uncheck	Uncheck	Black	ExtraLarge
18	Arial	Uncheck	Uncheck	Uncheck	Uncheck	Black	HalfSize

图 5-6 用于“我的字体对话框”的 BC 组合测试矩阵

许多组合测试工具的一个普遍问题是由于测试人员的能力所限，定义了高盖然的组合。PICT 通过一个重要的功能克服了以上的瓶颈，该功能允许测试人员将带有种子组合的、并制表符分隔的文件进行了定义，并导成工具。带有成熟组合的、制表符分隔的输入文件使得测试人员更大程度地控制具体的测试组合，同时仍旧允许工具来确定用于完整 n-对覆盖的整体测试。

在完成 BC 测试之后，接下来找到任何在故障指标上问题的，但未在 BC 矩阵中定义的常用组合。你可以将这些组合记录在制表符分隔的种子文件中，然后导成 PICT 工具，并作

为部分的测试组合输出，如图 5-7 所示。例如，如果你了解到大部分客户会使用带有粗体、下划线、大号的 Arial 字体，你可以将它记录在种子文件中。或者你断定带有粗体、斜体、删除线、下划线、小号、黄色的 Tahoma 字体常常会导致错误输出，你可以将它记录在种子文件中。种子文件会 PICT 包含以上这些组合，确保高盖然的、高风险的组合常常被测试到。

The screenshot shows a Microsoft Excel window titled "Seeded_Variables_Model - Microsoft Excel". The active cell is A1. A toolbar above the table has a "Font" button. The table has columns labeled A through G. Row 1 contains the header: "Font", "Bold", "Italic", "Strikethrough", "Underline", "Color", and "Size". Rows 2 and 3 contain data: Row 2 has "Arial" in column A, and "Check" in columns B, D, E, and F; "Black" in column G. Row 3 has "Tahoma" in column A, and "Check" in columns B, C, D, and E; "Yellow" in column F and "Small" in column G. The status bar at the bottom of the Excel window says "FontModel_1".

Font	Bold	Italic	Strikethrough	Underline	Color	Size
Arial	Check	Uncheck	Uncheck	Check	Black	Large
Tahoma	Check	Check	Check	Check	Yellow	Small

图 5-7 定义了基于客户信息或其他故障指标的可能组合，并能导入 PICT 工具、带有制表符分隔的种子文件

接下来，你必须创建输入文件来模拟每个参数的、适当的可变状态。利用简单的文本编辑器如 Notepad，你可以创建简单的文本，其中列举了参数和对应的可变状态。

```
Font: Arial, Tahoma, BrushScript, Monotype
Bold: Check, Uncheck
Italic: Check, Uncheck
Strikethrough: Check, Uncheck
Underline: Check, Uncheck
Color: Black, White, Red, Green, Blue, Yellow
Size: Small, Nominal, Large, ExtraLarge, HalfSize
```

现在，你准备使用 PICT 工具来生成配对测试组合的默认初始输出。PICT 是个命令行工具，它能生成关于 n-对覆盖测试组合的、制表符分隔的输出文件。在种子文件和模型文件的基础上，生成测试组合输出的命令行语法如下：

```
pict.exe myModelFile.txt /e:mySeededInput.txt > output.xls
```

一会，你就会得到由模型参数和测试人员提供的种子输入组成的配对输出，该输出是由制表符分隔的，因而可以通过微软电子表格办公软件轻松地校验。通过种子输入和早期定义的模型文件，你可以了解到现在只有 218 种可变状态的组合，利用 PICT 工具进行可变状态的配对分析可以把测试总量降低到 30 个。但是，你必须仔细地分析结果，通过检查条件或特定参数或可变状态的不变约束来确认输出组合的有效性。图 5-8 表明为什么验证工具的输出和修正输入模型对于避免假阴性是至关重要的。

	Font	Bold	Italic	Strikethrough	Underline	Color	Size
2	Arial	Check	Check	Uncheck	Check	Red	Nominal
3	Monotype	Uncheck	Uncheck	Check	Uncheck	Yellow	Nominal
4	Tahoma	Uncheck	Check	Uncheck	Uncheck	Black	ExtraLarge
5	BrushScript	Check	Uncheck	Check	Check	Green	ExtraLarge
6	BrushScript	Uncheck	Check	Check	Check	White	Small
7	Tahoma	Check	Uncheck	Check	Check	Black	Nominal
8	Arial	Check	Uncheck	Check	Uncheck	Red	Large
9	Monotype	Uncheck	Check	Uncheck	Check	Yellow	Large
10	BrushScript	Check	Uncheck	Uncheck	Uncheck	Yellow	Small
11	Arial	Uncheck	Check	Uncheck	Uncheck	Green	Small
12	Monotype	Check	Check	Check	Uncheck	Green	HalfSize
13	Monotype	Uncheck	Check	Uncheck	Check	Black	Small
14	BrushScript	Uncheck	Check	Check	Uncheck	Green	Nominal
15	BrushScript	Uncheck	Uncheck	Uncheck	Check	Red	HalfSize
16	Arial	Check	Check	Check	Uncheck	Blue	HalfSize
17	Tahoma	Check	Uncheck	Uncheck	Uncheck	White	Large
18	BrushScript	Uncheck	Uncheck	Uncheck	Check	Blue	Nominal
19	Tahoma	Uncheck	Check	Check	Check	Red	Small
20	Arial	Uncheck	Uncheck	Uncheck	Check	White	ExtraLarge
21	Tahoma	Check	Uncheck	Uncheck	Check	White	HalfSize

图 5-8 初始配对测试矩阵来说明带有互斥变量的组合测试

负测试如何？

本例中，你需要创建正测试的模型文件，而正测试就意味着每个组合所预计的输出不会导致错误的状态。当点击对话框中的“ok”按钮时，文本编辑器会相应地改变所选字型的字体、效果、颜色，而且你也可以通过手工测试或者利用 `GetFont` 的应用程序接口(API)来进行自动化测试，以此检验结果的正确性。

但是，产生你所预计的错误状态或条件的负测试怎么办呢？PICT 工具也包括定义模型文件中的无效可变状态。当测试无效变量状态时，为了避免输入屏蔽，每个测试应该只有一个无效参数值。当程序对第一个无效变量抛出错误之后就不会再测试之后的数值，导致输入屏蔽了。

当制定测试案例（尤其是自动化测试），我更倾向于使用正测试分解模型，用负测试来简化测试代码，并通过保持基于特定测试目标、适当的指示来降低假阴性。

要知道，字体直接依赖于特定的字型。例如，Brush Script 字体可以只有斜体或者即是粗体也是斜体，正如图 5-8 第 5, 12 和 16 行中错误的负面影响。而且，Monotype Corsive 字体可是默认的、或者只是粗体或斜体。图 5-8 第 4 和 15 行违反了依赖性规则，这样会导致出现测试中的假阴性。但是，一般很少遇到类似的功能，它依靠参数间的互斥状态来限制或约束互斥的特定事件和行为。手工改变这些输出数值是件愚蠢的事，因为你不清楚其他那些参数的变量状态会影响 n- 对组合。为了避免上述问题，PICT 可以在模型文件中，通过使用基本的 if-then 语法，为特定的变量状态定义了有条件的和不可变约束。

在模型文件中，加入条件的约束来确保在任何情况下字体变量是 Brush Script，不选粗体，选中斜体或者粗体和斜体都被选中。同样，你也可以加入条件的约束来确保在任何情况下字体变量是 Brush Script，不选粗体，选中斜体或者粗体和斜体都被选中。以下是修改过的模型文件：

```

Font: Arial, Tahoma, BrushScript, Monotype
Bold: Check, Uncheck
Italic: Check, Uncheck
Strikethrough: Check, Uncheck
Underline: Check, Uncheck
Color: Black, White, Red, Green, Blue, Yellow
Size: Small, Nominal, Large, ExtraLarge, HalfSize
#
# Conditional constraints for Brush Script and Monotype
#
IF [Font] = "BrushScript" AND [Italic] = "Uncheck" THEN [Bold] <=> "Check";
IF [Bold] = "Uncheck" AND [Italic] = "Uncheck" THEN NOT [Font] = "BrushScript";

```

在根据指定适当的限制来修改模型文件，你可以重新生成测试的输出文件。重要的是每当模型或种子输入文件被修改时，都要审查输出以验证工具输出的 n-对组合。既然你已经如图 5-9 更新模型文件，则没有出现带有粗体或无字体风格的 Brush Script。

Font	Bold	Italic	Strikethrough	Underline	Color	Size
Arial	Check	Check	Uncheck	Check	Red	Nominal
Monotype	Uncheck	Uncheck	Check	Uncheck	Yellow	Nominal
Tahoma	Uncheck	Check	Uncheck	Uncheck	Black	ExtraLarge
Tahoma	Check	Uncheck	Check	Check	Green	ExtraLarge
BrushScript	Uncheck	Check	Check	Check	White	Small
Arial	Check	Uncheck	Check	Check	Black	Nominal
Arial	Check	Uncheck	Uncheck	Uncheck	Black	Small
Monotype	Check	Check	Uncheck	Check	Yellow	Large
Arial	Check	Uncheck	Uncheck	Uncheck	Yellow	HalfSize
Arial	Check	Uncheck	Uncheck	Uncheck	Blue	ExtraLarge
BrushScript	Uncheck	Check	Uncheck	Uncheck	Green	HalfSize
Tahoma	Check	Uncheck	Check	Uncheck	Yellow	ExtraLarge
BrushScript	Uncheck	Uncheck	Check	Uncheck	Red	Small
Tahoma	Check	Uncheck	Uncheck	Uncheck	White	Nominal
Arial	Uncheck	Uncheck	Check	Uncheck	Red	Large
BrushScript	Uncheck	Check	Check	Check	Blue	Nominal
Tahoma	Check	Uncheck	Uncheck	Check	Red	ExtraLarge
Tahoma	Check	Uncheck	Uncheck	Uncheck	White	Large
Arial	Check	Uncheck	Check	Check	Black	Large
BrushScript	Check	Check	Check	Check	Black	HalfSize

图 5-9 初始配对测试矩阵来说明带有互斥变量的组合测试

你可以通过取别名或创建等价的变量状态来进一步定制 PICT 模型文件。例如，Arial 和 Tahoma 字体是很相似的字体，因此在与其他参数进行组合测试的时候，你可以断言 Arial 等同于 Tahoma 字体。你可以在 PICT 模型文件中，通过将逗号替换为(|)分隔符，给 Arial 和 Tahoma 字体起别名，如代码例子所示。利用 PICT 的另一个功能，你可以潜在地增加高盖然组合或变量状态的出现次数。例如，你可以断言最常用的字体颜色是黑色，而最常用的字体大小就是常规字体大小。再如之下代码所示，你可以修改 PICT 输入模型文件，为变量状态赋予权重。一旦某个变量状态满足所有的 n-对条件，则加权有助于为该变量赋予高优先级。

```

Font: Arial | Tahoma, BrushScript, Monotype
Bold: Check, Uncheck
Italic: Check, Uncheck
Strikethrough: Check, Uncheck
Underline: Check, Uncheck
Color: Black (10), White, Red, Green, Blue, Yellow
Size: Small, Nominal (10), Large, ExtraLarge, HalfSize
#
# Conditional constraints for Brush Script and Monotype
#
IF [Font] = "BrushScript" AND [Italic] = "Uncheck" THEN [Bold] <=> "Check";
IF [Bold] = "Uncheck" AND [Italic] = "Uncheck" THEN NOT [Font] = "BrushScript";

```

当输出包括了高盖然的测试组合以此确保高可能的用户设置被彻底测试到，输出排除了互斥组合以避免测试不经意地陷入假阴性之后，你就可以准备执行测试了。测试可以手工执行，PICT 工具的输出可以被导入数据驱动的自动化测试中。但是，由于每次测试，简单重用相同的变量组，导致输出结果意义不大，你需要改变变量组合的输出信息。同样，试图手工更改输出可能不太明智，因为这样的行为在不经意间、尤其是高复杂度的输出情况下，干扰了 n-对组合。因而，PICT 包括另一个重要的功能，也就是在保持所有变量状态的 n-对覆盖同时，将输出随机化。通过/r:[seed]参数导入 PICT 工具，使得输出随机化，如此可以有效地增加覆盖面的广度，甚至可以识别之前错过组合的缺陷。

BC 和配对测试就足够了？

历史经验表明大多数（大于 50%）、源于参数间相互作用的缺陷常出现在简单配对组合。但是，除了测试 BC 矩阵和把 PICT 工具中的配对输出随机化以外，最近的研究表明持续增加的 n-对组合测试可以暴露之前未察觉的细微缺陷。最近的研究表明 6-对覆盖，就缺陷去除效率和结构性代码覆盖策略而言，堪比穷举覆盖。然而，测试的增加数量大约是 n-对覆盖每个阶段测试增加数量的平方，进而增加了测试的总成本。

我推荐使用 PICT 工具中输出矩阵的随机化功能，以此来扩展配对测试覆盖面的广度。微软专家，Jacek Czerwonka，推荐你从 BC 和配对测试起步，然后进行 3-对和 4-对组合测试。Jacek 还提出 5-对和 6-对测试可能只暴露些细微的问题，但是在测试开销较低的情况下，还是建议执行一下，以保证合理的覆盖深度。7-对或更高的测试不太可能进一步检测到缺陷或扩展覆盖范围。Jacek 认为配对测试网站(<http://www.pairwise.org>)提供了相当丰富的测试信息。

组合测试的有效性

目前，有许多方法来评估测试技术或方法的有效性。检测缺陷的有效性 (DDE) 是个最常用的测试有效性的方法之一（虽然它完全是盲目的，除非处在缺陷的总量都可知或者可以通过缺陷密度和其它方式预测的环境下）。由于暴露缺陷是最明显的测试结果，因此你可以分析 n-对测试的 DDE。许多研究都表明很多缺陷是由相互作用参数的变量状态组合造成的，而所有相互作用的故障缺陷中，50%以上是由简单对作用或配对组合造成的。一份学术研究²²表明 n-对的 DDE 和随机选择组合测试套件相比“并未发现明显的区别”，该组合套件中

⁸ Patrick J. Schroeder, Pankaj Bolaki, and Vijayram Gopu, Comparing the Fault Detection Effectiveness of N-Way and Random Test Suites, 2004 International Symposium on Empirical Software Engineering.

每个测试都是由程序从所有可能的组合选取的。另一项行业研究²³使用 BC 和配对测试检测到了 98% 的组合故障。最近，更多的研究判断随着组合数量持续增加到 6-对覆盖，细微的组合缺陷仍旧会暴露。因此，我建议测试人员从 BC 和配对测试开始，然后将配对输出结果随机化，并持续增加 n-对测试组合总量直到 6-对覆盖。

另一种评估测试有效性的方法是查看测试人员提供的资料。许多经验数据表明多数黑盒测试都将结构性地覆盖 65% 的复杂程序。随着复杂算法的结构测试增加，测试人员提供的额外信息有助于降低全局风险。一个研究表明配对测试，相比于随机输入测试，其代码块和分支的结构性测试数量增加了 25%。如表 5-6 所示，微软已经收集的经验数据表明相比于黑盒测试而言，使用配对测试的，在产品代码块和弧覆盖的数量方面会有所增加。数据还表明随着 n-对覆盖的持续增长，代码块和弧覆盖率都会有额外的增加。虽然结构性测试的增加有助于降低全局的风险系数，但同时也会额外增加成本。

表 5-6 利用增长的代码覆盖来检测组合测试的有效性

总块数=1,317	手工测试	配对测试	n-3 覆盖	n-4 覆盖
测试次数	236	136	800	3, 533
覆盖的块	960	979	994	1, 006
代码覆盖率	73%	74%	75%	76%
未覆盖的函数	11	11	10	10

另一种检测有效性的途径是降低运作的成本。在微软，某些组通过多种方式显著地降低了运作的成本。例如，由于 PICT 工具的输出是个制表符分隔的文件，它可以轻易地导入数据驱动的测试自动化中。通过随机化所涉及的测试变量，许多组已经大大增加自动化测试的数量，并扩展覆盖的广度。其他组也显著的减少环境配置的数量，最终导致越来越多的测试其总体有效性有所退步。相比于其他的、软件发布前的测试，PICT 的使用已经显露了之前从未察觉的缺陷。

组合测试小结

组合分析不是解决所有测试问题的银弹。初学者有可能滥用工具或误用该项技术来测试不牵涉相互依赖参数的功能区域。测试人员必须有能力正确分析功能参数，鉴别依赖参数和半耦合参数，并分解变量状态，识别有条件的和不变的约束。相比于其他的组合测试方法，当具有深入的系统知识、高水平的测试人员部署本书所描述的步骤，则该技术能高效地尽早检测出缺陷、增加结构性覆盖和降低运作成本。事实上，在微软这是既定的最佳做法。

⁹ Mats Grindal, Birgitta Lindstrom, Jeff Offutt, and Sten F. Andler, "An Evaluation of Combination Testing

本章小结

本章描述了在微软所部署的功能性测试技术一个样例，这种方式能高效地评估软件程序的功能性属性和性能。这些技术系统地分析了一个程序的功能性组件，并增加测试覆盖，降低开销冗余，更好地评估全局风险，为决策者提供合格的信息，使得他们能做出更明智的选择。我们还发现，这些技术的应用往往迫使我们以不同的方式研究软件测试。所使用的、附加功能性技术包括原因和效果图形、决策表和状态转换测试。这些额外的技术基于等价类的划分、边界值的分析和组合分析，要求我们通过一个程序系统检查输入和输出的相互作用、顺序操作或交易流量。

当技术被正确使用时，该技术就擅长于暴露特定类型的、有意被识别的缺陷。但是，它们不能发现所有问题，这就是为什么你需要学会和理解各种方法和技术，并在合适的环境下适当地使用它们。结合其他技术如勘测试验，功能性测试技术就非常有用，尤其是当决策者需要对软件项目更系统的分析。这些技术是依赖于试探法、故障模型和历史经验的系统过程，它们有助于解决测试复杂系统中的复杂问题。为了达到最大的有效性，功能性技术要求全面的领域和系统知识。这些技术不会替代其他软件测试技术，但可以显著地增加测试有效性，帮助鉴别和降低冗余性，也能减少测试工作中的感性成分，以克服杀虫剂困境。

第 6 章 结构测试技术

BJ·罗里森

一位学前班老师问她的学生：“谁知道苹果是什么颜色的？”有一个同学自信满满地举起了手。“伊森，苹果是什么颜色的？”，老师问道。小伊森自豪地站了起来，说：“苹果是红色的”。话音刚落，又有几只手举了起来。老师点了其中一个同学，“爱玛，你说呢？”爱玛答道：“有些苹果是绿色的。”“正确！”老师高兴地说。接着，又有更多的手举了起来。这次老师还是选了一位用力挥着手的急切地想要回答的同学。不过，还没等到这个同学回答，小卡洛琳已经等不及了，她大声喊道：“还有一些苹果是黄色的。”“是的！苹果的颜色有红色、绿色和黄色”，老师总结说。不过，在继续讲课之前，老师注意到了一个坐在教室角落里的安静的小女孩，她仍然举着手。老师点了这个小女孩回答。小女孩说：“苹果也是白色的。”老师听了以后的确有点吃惊，她礼貌地对小女孩说：“伊丽莎白，我们有红苹果，有绿苹果，还有黄苹果，但我从没见过白苹果。”教室里的其他小朋友开始笑了起来。伊丽莎白站了起来，扶了扶眼镜，然后诚恳的解释道：“所有的苹果内部都是白色的！”

我们常常习惯于简单地从行为方面来评价软件质量，却忽略了考虑那些隐藏在用户界面之下的计算特征。医生们都知道，在治疗某些疾病的时候，我们不能只看病症的物理表现，有时候还必须考虑对人的内部系统进行各种各样的测试。深度诊断（例如血液测试或核磁共振扫描）有时候能够帮助我们在一些潜在疾病恶化之前及早发现它们。类似的，对于那些非常关键的系统，或是需要立即对一个极其复杂的软件进行更详细分析的时候，我们必须透过外表对那些隐藏在表面之下部分进行全面的检查。结构测试技术通过对函数控制流程的详细分析来帮助我们降低风险。在某些情况下，结构测试对于研究分析程序的某部分代码非常有效，这些代码先前并没有经过大规模的行为测试，也没有被基于黑盒理论而设计的方法性的功能测试所验证过。

与那些既能应用于设计黑盒测试也能应用于设计白盒测试的功能测试方法不同，结构测试技术是一种白盒测试设计方法。以白盒的观点来设计测试是基于程序的内部结构和具体实现。这并不是说一个测试员如果想要通过设计白盒测试来提供更多的信息和进一步降低风险，那么他就必须要熟悉在程序实现中所使用的程序设计语言。不过，这里我们也必须注意，使用白盒测试方法所设计的测试既能在用户界面（UI）层次执行，也能通过使用存根（stub）或虚拟对象的方式在用户界面之下的组件层次执行——如果我们假设程序的功能与用户界面是分开的。

关于白盒测试方法的一个常见误解是，测试员会受到代码的影响，从而设计出的那些测试只能验证函数本来所应该完成的功能。我并不同意这样的观点，因为一个好的测试员不仅仅是简单地验证代码做了它应该做的事。与其他测试方法比较起来，一个了解数据类型、函数调用和程序结构的优秀测试员能够更有效的发现不同类型的问题。当然，这也并不意味着白盒测试比其他方法更优越。基于白盒观点（例如使用结构化测试方法）设计的测试只是对行为和功能测试技术和方法的补充。它们并不能取代其他测试方法。我们从来不鼓励简单地基于白盒测试方法来设计测试，或者设计那些仅用于验证程序的结构完整性的测试。结构测试及应用技术通常与代码覆盖分析一并使用，因为它们可以通过设计更多的测试来提高代码覆

盖率。此外，当我们需要对程序进行深度分析和详细检查以降低整体风险时，结构测试也可以提供更多有价值的信息。

我们需要结构测试吗？

毫无疑问，行为和探索性测试（一种测试执行方法，测试员使用测试过程中获得的信息来直观地衍生出更多的测试）具有重要价值。行为和探索性测试通常可用于评估项目的外在体验（*look and feel*），但最近的一些研究对软件测试中行为测试和常用的探索性测试方法的总体效果和效率提出了质疑。赫尔辛基大学（Helsinki University）的一项研究²⁴发现，当比较基于测试用例的测试和探索性测试时，两种方法“在缺陷检测的效率（或者说在检测出的缺陷的类型和严重性）上没有显著的差异”。一项更早的研究²⁵也提供了经验数据，反驳那些宣称探索性测试提高了工作效率的说法。由 Marnie Hutcheson 进行的另一项案例研究²⁶发现，与经过正式培训之后所进行的基于完整需求的测试相比，探索性测试在发现特定的问题的方面效果更差。微软的一项尚未发表的内部研究发现，就代码覆盖率来讲，基于脚本的测试和探索性测试没有显著差别。

在微软进行的五年研究中，我们发现，与脚本化的测试相比，探索性测试或行为测试在代码覆盖率上并无显著差异。超过 3000 名测试员参与了这项实验，每 25 人一组，实验结果在所有组中都是一致的。在这项研究中，根据样式书设计的脚本化测试在被测程序上达到了标称 83% 的代码覆盖率。然后，实验参与者允许进行每人 15 分钟，累计 5 小时的探索性测试。令人惊讶的是，代码覆盖率平均只增加了 3 个百分点。但是，当实验参与者能够分析探测过的（Instrumented）代码的运行结果并使用白盒技术设计测试以后，不到 20 分钟的时间代码覆盖率就提高到了 91%（这是不使用代码突变或故障注入所能达到的最大实际代码覆盖率）。同时，测试员们也能够更好的从代价和收益的角度解释为什么剩下的 9% 未覆盖代码是不可测试的。图 6-1 显示了不同测试技术的代码覆盖效果。

²⁴ Juha Itkonen, Mika V. Mantyla, and Casper Lassenius, “Defect Detection Efficiency: Test Case Based vs. Exploratory Testing,” *ESEM*, 2007, 61–70, First International Symposium on Empirical Software Engineering and Measurement.

²⁵ Juha Itkonen and Kristian Rautiainen, “Exploratory Testing: A Multiple Case Study,” *International Symposium on Empirical Software Engineering*, November 2005.

²⁶ Marnie L. Hutcheson, “Exploratory Testing versus Requirements-Based Testing: A Comparative Study,” paper presented at Practical Software Quality and Testing Conference, 2007.

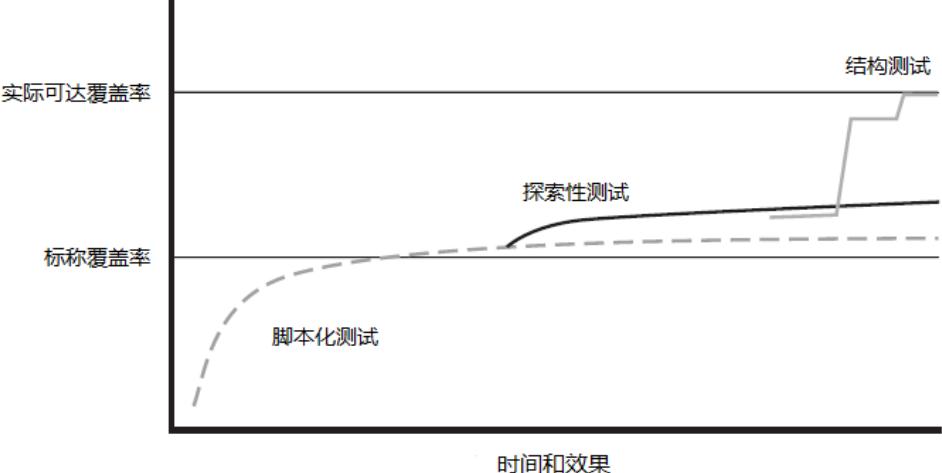


图 6-1 不同测试技术的代码覆盖效果

这些研究并没有暗示探索性测试或其他行为测试方法效率低下。事实上，探索性测试在特定的条件下很有效，并能比其他方法更可靠地发现特定类型的缺陷。但是，行为测试方法的整体效果在很大程度上取决于测试员对系统和领域的深入理解和经验。而且，现在的经验也表明简单地增加测试人员的数量或雇佣具有不同观点的测试人员来进行行为或探索性测试并不能显著降低风险，并有可能导致产品的某些关键部分没有被测试或者被充分测试。此外，任何一种测试方法都不是永远有效的，测试员必须采用各种不同的方法来进一步研究和评估被测软件。这些研究告诉我们，当决策者们需要更多有用的信息来减少或分析那些高度复杂的系统或者关键系统的潜在风险时，这些能够评估程序的结构完整性的系统化测试技术也许是不可或缺的。

块测试（Block Testing）

函数的基本结构测试方法使用了一些简单技术，包括语句测试和块测试。语句测试的目的是执行一个函数中的所有语句。与之相对应，块测试则执行成组的连续语句或不包含分支或函数调用的语句块。因为块测试为评估函数的控制流程提供了更高的敏感度，并且当前使用的很多代码覆盖工具测量的都是块覆盖，所以我们将使用块覆盖来设计我们最简单的结构测试。

块覆盖和语句覆盖

语句覆盖测量一个程序在测试过程中被执行过的语句的数量。块覆盖测量无分支的连续语句组的数量。导致控制流程转向分支的条件语句可以包含若干块。这个看起来似乎只是一个极小的区别，然而，语句测试和块测试的区别是相当重要的。因为相较于语句测试，块测试对控制流程提供了更好的敏感度。

Example1 函数

```

public static void BlockExample1 (bool condition)
{
    int x = 0, y = 0, z = 0;
    if (condition)
    {
        x = 1;
        y = 2;
        z = 3;
    }
    return x + y + z;
}

```

Example1 函数显示了块覆盖和语句覆盖测量的基本差异。在这个例子中有 5 条语句，但只有 4 个基本块。一个 *condition* 参数的值为 *false* 的测试具有 40% 的语句覆盖，因为只有两条语句被执行了。但是块覆盖将会是 75%，因为这个测试执行了总共 4 个代码块中的 3 个。当然，如果 *condition* 参数的值为 *true*，那么语句覆盖和块覆盖都能达到 100%。

块覆盖相较于语句覆盖的优势可以在更复杂的结构中体现出来。例如，在 *Example2* 函数中，同时传递 *true* 的值给参数 *condition1* 和 *condition2* 可以得到 100% 的语句覆盖，但只能达到 85.71% 的块覆盖。*Example2* 函数需要另外一个测试 (*condition1* 为 *false*, *condition2* 为 *true*) 才能达到 100% 的块覆盖。

Example2 函数

```

public static void BlockExample1 (bool condition1, bool condition2)
{
    int x = 0, y = 0, z = 0;
    if (condition1 && condition2)
    {
        x = 1;
        y = 2;
        z = 3;
    }
    return x + y + z;
}

```

当然，这两个例子也显示了语句覆盖或块覆盖的不足之处。对于简单的函数而言，要使用少数测试来达到较高的代码覆盖是比较容易的。这也是为什么我们必须记住，代码覆盖与质量之间没有直接的关系，并且代码覆盖与测试效果也仅具有间接的关联。结构测试的价值取决于测试员分析控制流程以及设计测试以确保各个不同的控制路径至少执行一次的能力。

在下面这个 *SimpleSearch* 函数例子中，我们可以通过一个测试来执行所有的语句，只需传递字符串参数 *AB* 和字符参数 *B* 到函数的参数列表即可。在这个简单的测试中，因为变量 *index* 的值小于数组长度，并且通过参数 *myCharacter* 传递给函数的字符变量值不等于由当前索引值所指定的数组元素，所以控制流程将进入 *while* 循环。在 *while* 循环中，我们执行一条语

句让变量 *index* 的值加 1，然后返回 *while* 循环来重新计算两个条件语句的值。变量 *index* 的值仍然小于数组长度，但数组的第二个元素是字符 B，与变量 *myCharacter* 相等。第二个条件语句为真，因为变量 *index* 的值小于数组长度，因此我们进入这个块并且让 *index* 的值加 1，然后将它赋值给变量 *retVal*。最后，控制流程退出条件决策块并返回变量 *retValue* 的值给调用函数。因此，使用一个简单的测试，我们能够执行函数中的所有语句。然而，没有任何一个专业的测试员会满足于对这个程序仅仅做一个如此简单的测试！

SimpleSearch 函数

```
// Simple search function that searches for the first instance of a character in a
// string and returns the position of that character in the string, or a -1.
public static int SimpleSearch (string myString, char myCharacter)
{
    int index = 0, retVal = -1;
    char[] strArray = s.ToCharArray();
    while ( index < strArray.Length && strArray[index] != c)
    {
        index++;
    }
    if (index < strArray.Length)
    {
        retVal = index++;
    }
    return retVal;
}
```

作为对比，我们需要至少两个测试来对这个函数进行块测试。在第一个块测试中，我们将分别传递字符串 A 和字符 A 给函数的第一和第二个参数。这个测试将执行 11 个块中的 9 个，如图 6-2 所示。

The screenshot shows a debugger interface with two windows. The top window is titled 'Search.Program' and displays the C# code for 'Main(string[] args)' and 'SimpleSearch(string s, char c)'. The bottom window is titled 'Code Coverage Results' and shows a table with the following data:

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
SimpleSearch(string, char)	2	10.10 %	9	81.82 %

图 6-2 第一个测试的块覆盖结果

在第二个测试中，我们传递字符串参数 A 和字符参数 B 给函数并执行了 11 个块中的 10 个，如图 6-3 所示。注意，第二个测试执行了先前没有测试过的块，但也跳过了一个已经在先前测试中执行过的块。使用这两个简单的块测试，我们可以获得 100% 的块覆盖。

The screenshot shows a debugger interface with two windows. The top window is titled 'Search.Program' and displays the C# code for 'Main(string[] args)' and 'SimpleSearch(string s, char c)'. The bottom window is titled 'Code Coverage Results' and shows a table with the following data:

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
SimpleSearch(string, char)	1	9.09 %	10	90.91 %

图 6-3 第二个测试的块覆盖结果

简单地说，控制流程图（Control flow diagrams，简称 CFD）就是函数或类的模型。一个 CFD 必须有一个入口和一个出口。在函数的入口和出口之间，CFD 提供了程序代码的一种抽象表示或模型，从而帮助测试人员通过遍历一个复杂函数的不同代码路径来追踪控制流程。CFD 有两种基本类型，如图 6-4 所示。

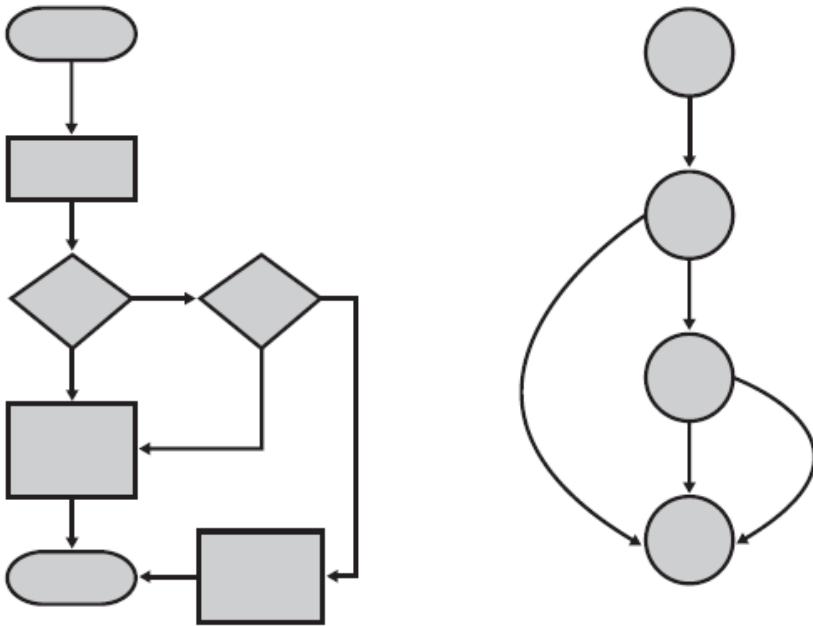


图 6-4 控制流程图示例

一个基本的 CFD 使用典型的流程图符号来建立函数的模型。基本的 CFD 用矩形代表语句或语句块，用菱形代表决策或条件子句。对函数不熟悉的人也容易阅读和理解基本的 CFD。基本的 CFD 通过使用流程块和条件子句，一步一步地把控制流程模型建立起来。简化的 CFD 用圆形代表函数中的决策点，在这些决策点上控制流程可能根据一个关系条件的结果而改变。简化的 CFD 是函数的更抽象的表示，因为它隐藏了语句块的细节。有时这个层次的抽象一开始不易理解，但是它提高了建模过程的效率。

需要记住的是，如果目的是进行单元测试，块测试一般已经足够了，但它只是结构测试的一个相对较弱的形式。然而，有些程序结构还是需要测试人员专门设计块测试，以确保函数有足够的结构覆盖。适合于块测试的一种结构类型是 `case/switch` 语句。`switch` 语句处理对一个参数的多重选择，是处理函数内部控制流程的一个有效方法。当我们在调试器里步进跟踪一个 `case/switch` 语句时，我们能看到控制流程立刻转移到与传递给 `switch` 语句的控制参数相匹配的 `case` 语句。

比如说，程序化平台配置文件相当普遍地应用于自动化测试中，这些测试的过程或结果在不同的 Microsoft Windows 操作系统版本上可能有所不同，一个类似于 `SimpleGetNT5ClientVersion` 函数的函数可以帮助我们确定程序或自动化的测试是运行在基于 Windows 2000 还是基于 Windows XP 的客户端。图 6-5 所示的顺序控制流程图分别显示了在非 NT5 内核操作系统、Windows Server 2003 操作系统、Windows 2000 Server 以及 Windows

XP 操作系统上控制流经 *SimpleGetNT5ClientVersion* 函数的情况。

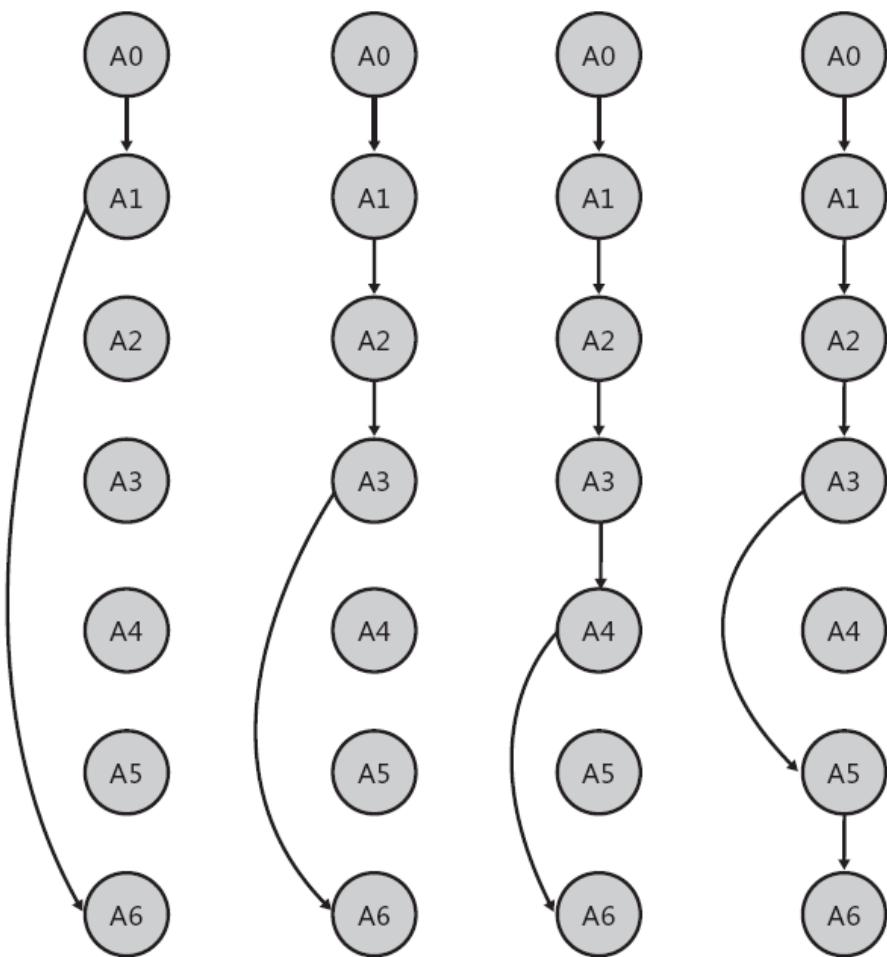


图 6-5 *GetOSVersion* 函数的顺序控制流程图

SimpleGetNT5ClientVersion 函数

```

//*****
// SimpleGetNT5ClientVersion.cs © 2008 by Bj Rollison
// Returns the Windows NT operating system environment
//*****
private const int WINDOWS_NT5_KERNEL = 5;
private const int WINDOWS_2000 = 0;
private const int WINDOWS_XP = 1;

A0 private static string SimpleGetNT5ClientVersion()
{
    OperatingSystem osVersionInfo = Environment.OSVersion;
    string osVersion = string.Empty;

A1     switch(osVersionInfo.Version.Major)
    {
A2         case WINDOWS_NT5_KERNEL:
A3             switch(osVersionInfo.Version.Minor)
                {
A4                 case WINDOWS_2000:
                        osVersion = "Win2K";
                        break;
A5                 case WINDOWS_XP:
                        osVersion = "WinXp";
                        break;
                }
            break;
    }
    return osVersion;
A6 }

```

第一个 CFD 展示了当操作系统的主版本号不等于整数值 5 时，控制流程会怎样简单地绕过 `case` 语句并且返回一个空字符串。第二个 CFD 说明当操作系统环境是 Windows Server 2003 时，控制流程从 `switch` 语句 A3 跳到 A6，并且也返回一个空字符串。第三个 CFD 显示了当操作系统环境是 Windows 2000 Server 时，控制流程从 A3 到 A4 的 `case` 语句并返回字符串 `Win2K`。并且，如果操作系统环境是 Windows XP，第四个 CFD 展示了控制流程怎样从 A3 跳转到 A5 的 `case` 语句并返回字符串 `WinXP`。

块测试体现其重要性的另一个地方是异常处理（exception handling）。很有意思的是，行为测试通常并不执行程序中的许多异常处理，因此，设计结构测试以减少风险和保证异常的正确处理是很重要的。如果应用程序本身能处理异常，并且异常发生在应用程序代码的执行过程中，那么系统将搜寻适当的异常处理程序并且控制流程也会转移到相应的位置。

ConvertToPositiveInteger 函数

```

A0  private static int ConvertToPositiveInteger(string s)
{
    try
    {
A1        return (int)Convert.ToInt32(s);
    }
A2    catch(FormatException)
    {
        return -1;
    }
A3    catch(OverflowException)
    {
        return -1;
    }
A4 }

```

上面的 `ConvertToPositiveInteger` 函数是一个包含异常处理的控制流程的简单例子。图 6-6 `ConvertToPositiveInteger` 函数的顺序 CFD 显示了控制流程怎样跳过或跳转到合适的异常处理程序。就如图 6-6 的第一个 CFDs 所示那样，一个块测试传递一个只含 0 到 2147483647 之间的某一个整数的字符串，它将这个传递给参数 `number` 的字符串转换成一个整数，并退出函数。为了测试异常处理程序，我们需要至少两个附加测试。在第二个测试中，我们传递一个非整数值（字符串、空字符串、字符、及其他值）给参数 `number`。任何一个非整数值都将触发一个格式异常错误，控制从 A1 跳到 A2，在这里返回值被设为-1，控制流程结束并返回到调用函数。第三个测试必须执行溢出的异常处理程序。一个代表大于 2147483647 的整数或任何负数的字符串变量会在 `try` 的代码块 A1 里触发溢出异常，这时控制会跳转到溢出异常处理程序。

在 CFD 中，语句 `int.Parse()` 和 `(int)Convert.ToInt32()` 都由单个节点 A1 代表，因为进行块测试时我们并不需要区分这两个语句的差异。然而，测试者必须也认识到这两个流程语句对参数 `number` 设置了特定的最小和最大值的输入边界条件。`int.Parse()` 函数把一个字符串表示的整数转换为一个介于 -2,147,483,648 到 2,147,483,647 之间的有符号的整数值。`Convert.ToInt32()` 函数把一个以 `System.String` 的形式表示的整数转换为一个介于 0 和 4,294,967,295 间的整数值。虽然我们只需要一个块测试来评估溢出异常处理程序的控制流程，但是专业的测试人员知道，要充分地评估这个函数，我们还需要更多的边界条件测试。

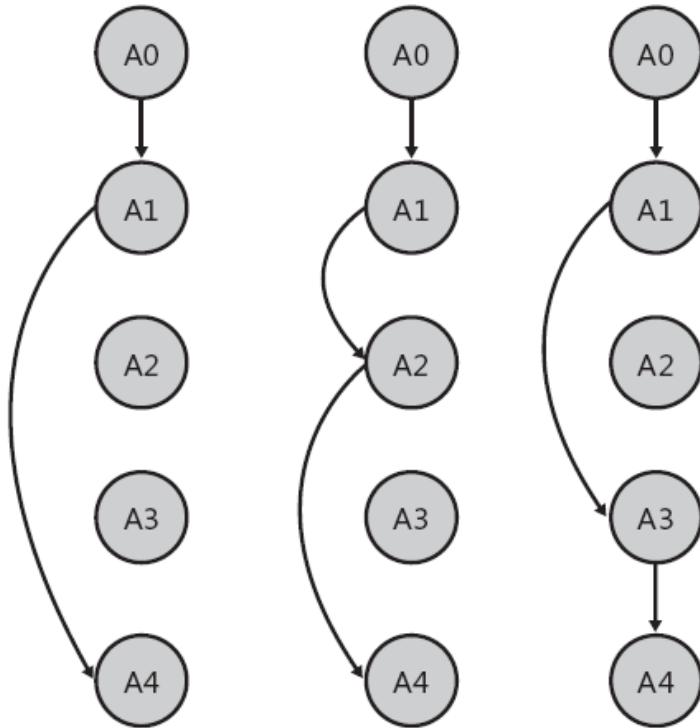


图 6-6 *ConvertToPositiveInteger* 函数的顺序控制流程图

块测试小结

块测试是用于单元测试的一种普遍方法，它非常适合于迅速地评估某函数的基本功能。对于设计用于执行 `switch/case` 语句和异常处理程序控制流程的测试来说，它也是一个很有价值的技术。然而，块测试是健壮的结构测试中相对较弱的标准，它还可能漏掉控制流程的一些重要的分支。此外，块测试还容易忽略一些潜在的问题，特别是在我们测试的目的只是要提高代码覆盖率而不是要仔细分析被测试代码的情况下。

决策测试（Decision Testing）

决策测试对条件子句求值，根据条件为真或假，简单的布尔表达式就可以确定控制流程的分支。决策测试的主要目标是设计既能够验证布尔表达中的真（`true`）也能验证假（`false`）结果的测试。这种结构化测试的设计与块测试类似，但是设计的测试不是用于执行代码的连续代码块，而是注重于评价函数中的条件子句。与块测试相比，决策测试的优点在于它对控制流提供更好的敏感度。下面的 `CreateNewFile` 函数有助于说明决策测试与块测试相比的有效性。

`CreateNewFile` 函数

```

// Simplified function to check for an existing file and delete that file if it exists
// and create a new file
A0 private static void CreateNewFile (string myFilename)
{
A1    if (File.Exists (myFilename))
    {
        File.Delete (myFilename);
    }
    File.Create (myFilename);
A2 }

```

在本例中，我们仅用一个测试就可以达到对 *CreateNewFile* 函数的 100% 的块覆盖率。通过传递一个有效的文件名给函数的 *myFilename* 参数（文件已经在系统中指定的地点存在），我们可以执行该函数中的所有块。然而，正如图 6-7 中的控制流程图显示的，该函数的简单条件语句的结果中，只有为真的结果被执行了。如图 6-7 的 CFD 中的虚线所示，A1 的条件语句的为“假”的结果没有被历经。这是块覆盖度量的又一个弱点。

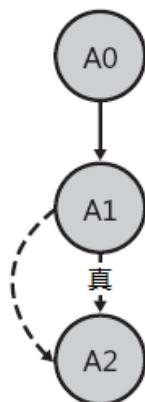


图 6-7 *CreateNewFile* 函数的控制流程图

CreateNewFile 函数的决策测试要求最少两个测试。第一个测试，我们传递一个在系统中存在的文件的文件名给函数，使条件语句评估它为“真”的结果。第二个决策或分支测试要求我们传递一个并不存在的有效文件名给函数，以此评估条件语句为“假”的结果。虽然这只是简单的例子，它说明了与块覆盖比较而言，决策测试如何为简单条件语句提供更好的敏感度。

请注意，我们没有传递无效的文件名字符串给 *myFilename* 参数来检查各种错误条件。条件测试的目的不是测试所有可能的输入或输出，而是简单的帮助我们测试贯穿函数的控制流程。在这个例子中，我们假设字符串 *myFilename* 在别的地方验证，然后一个有效的字符串作为参数传递给 *myFilename*。

不过，测试员在处理循环结构的时候必须小心谨慎，因为很容易设计出一些无效的测试，这些测试没有有效的测试循环结构却仍然验证了条件语句的真、假两种结果。例如，我们可以为函数 *GetCharacterCount* 设计一个测试，这个测试传递变量 A 给参数 *myString* 以及变量 A 给参数 *myCharacter*，这样就能使 *for* 循环的条件语句为真。下一次循环计算出的条件语句为假，因为索引值与 *myStringArray* 变量的长度相等。

GetCharacterCount.cs

```
// This function counts the number of specified characters in a string
private static int GetCharaacterCount (string myString, char myCharacter)
{
    char[] myStringArray = myString.ToCharArray();
    int result = 0;

    for (int i = 0; i < myStringArray.Length; i++)
    {
        if (myStringArray[i] == myCharacter)
        {
            result++;
        }
    }
    return result;
}
```

为了对这个函数进行更加全面的决策测试，一个更有效的决策测试策略是执行最少三个测试。在本章后边描述的基础路径测试技术提供了一种新的技术，它能确保那些使用简单条件语句的函数能够获得一套更加详尽的结构测试。有一点要指出的是，我们也许需要设计更多的测试来全面测试这个函数的功能，但是对于函数的决策测试来说，我们只需要两个测试。

决策测试小结

总的来说，与块测试相比，决策测试为控制流程提供了更好的敏感度。决策测试对简单条件语句是行之有效的，这些条件语句只需要计算布尔条件表达式的值，比如 `if` 语句或循环结构。但与块测试类似，决策测试并不能充分验证条件语句中的关系运算符。对条件语句

```
if ( x <= 5)
```

的决策测试包括测试 `x` 等于 5 的情况，这样该条件判断就为真。第二个测试的是 `x` 大于 5 而使该条件表达式为假。这两个测试使我们能够评估该条件判断结果为真和假的两种结果，但不能充分测试这个关系运算符。测试此关系运算符需额外测试 `x` 小于 5 的情况。虽然第三个测试似乎是不言自明的，但不正确地使用关系运算符是导致边界条件一类的软件缺陷的主要成因之一。

决策测试也不能有效地评估复合条件判断的控制流程。例如，`SimpleSearch` 函数中的循环结构包含两个条件语句或子表达式。决策测试被用来测试复合条件判断的结果为真和假的情况，不论有多少个条件判断表达式。所以，如调用 `SimpleSearch` 函数时，将字符串“ABC”传给参数 `myString`，将字符“B”传给参数 `myCharacter`，循环结构中的条件语句在不同时间会被估值为真和假。然而，我们没有充分评估该循环一直执行到 `i < myStringArray.Length` 不再满足的情况。为充分测试有多个条件子表达式的复合条件语句，我们使用一种称为“条件测试”的略有不同的结构测试技术。

条件测试（Condition Testing）

有时一个函数中的控制流程取决于多个条件语句的结果。开发工程师通常可以使用一个由多个布尔子表达式经逻辑运算符 AND 或 OR 连接成的条件语句来简化代码，而不是用级联的多个条件语句。例如，*IsNumberBetweenMinAndMax* 函数检查一个整数是否介于最小值和最大值之间。下面这个实现使用两个简单条件语句来返回一个布尔类型的结果。

IsNumberBetweenMinAndMax() 函数

```
private static bool IsNumberBetweenMinAndMax (int number)
{
    int minValue = 1;
    int maxValue = 10;

    if !(number < minValue)
    {
        if !(number > maxValue)
        {
            return true;
        }
    }
    return false;
}
```

但是，我们可以重写这个函数，使用由逻辑运算符 AND 连接的两个布尔子表达式来代替这两个条件语句。这样重构调整代码（如下面的例子）的好处包括代码行数减少，可维护性提高，以及减少语句块的数目。之前的代码需要三个决策测试，下面的重构后的代码还是有三个条件测试（如果我们假设在逻辑运算中使用“短路”）。

RefactoredIsNumberBetweenMinAndMax 函数

```
private static bool IsNumberBetweenMinAndMax (int number)
{
    int minValue = 1;
    int maxValue = 10;

    if (!(number < minValue) && !(number > maxValue))
    {
        return true;
    }
    return false;
}
```

当一个条件语句包含两个或两个以上的布尔子表达式，我们可以使用条件测试这种结构测试技术。条件测试类似于决策测试，但条件测试的测试是用于评估复合条件语句中每个子表达

式估值为“真”和“假”的结果。在测试复合条件语句时，条件测试比决策测试对控制流程提供了更好的敏感度。

例如，我们在第 5 章测试的 `Next Date` 程序用函数 `IsInvalidGregorianCalendarDate` 来检查输入变量不在表 5-1 中 i18 所指的范围内。`IsInvalidGregorianCalendarDate` 函数用一个复合条件语句来判断 `year`, `month`, `day` 这三个参数的输入变量是正确（变量不在特定日期区间）或错误（变量在特定日期区间）。用决策测试只需两个测试，分别将条件语句评估为“真”和“假”。但是，很显然，简单的决策测试不能充分有效地测试此函数的结构和控制流程。

`IsInvalidGregorianCalendarDate()` 函数

```
// The following function checks for dates between 10/5/1582 and 10/15/1582
// which are dates excluded on the original Gregorian Calendar
private static bool IsInvalidGregorianCalendarDate (int year, int month, int day)
{
    if (year == 1582 && month == 10 && !(day < 5) && !(day > 14))
    {
        return true;
    }
    return false;
}
```

条件测试函数 `IsInvalidGregorianCalendarDate` 需要五个测试来有效评估该复合条件语句中每个布尔子表达式的真、假结果。图 6-8 中的控制流程图代表一个控制流程到达本函数中每个条件语句时的模型。每个条件语句的测试和结果列在表 6-1 所示的真值表中。

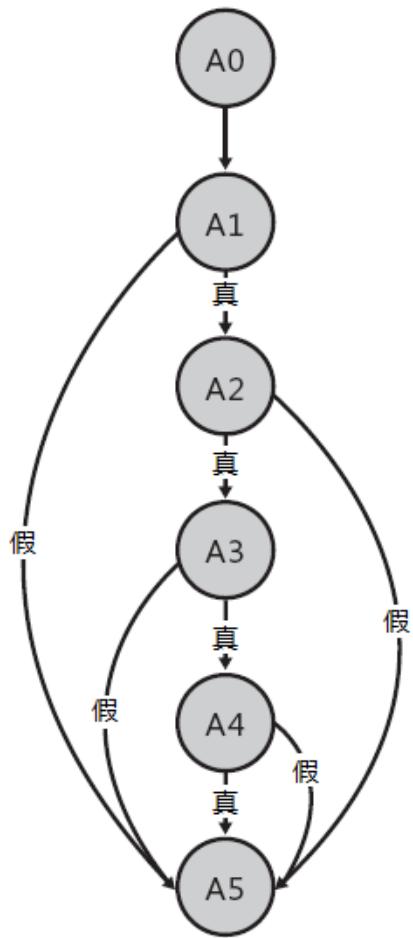


图 6-8 *IsInvalidGregorianCalendarDate* 函数控制流程图

表 6-1 *IsInvalidGregorianCalendarDate* 函数真值表

测试	参数			条件子句				期待值	
	Month	Day	Year	Year	Month	$!(day < 5) \wedge !(day > 14)$			
						<	>		
1	10	11	1582	真	真	真	真	真	
2	10	21	1582	真	真	真	假	假	
3	10	3	1582	真	真	假		假	
4	5	7	1582	真	假			假	
5	10	5	1994	假				假	

如表 6-1 所示, 这些作为实参传递给形式参数 month, day, year 的变量是每个形式参数的有效等效类子集 的代表性例子。在表 6-1 中用灰色突出的是测试 3、4、5 的布尔结果。它们被列出仅为说明用途, 由于我们假设逻辑表达式求解时 “短路”, 它们在程序执行中不会被评估。当第一个条件语句解析为 “假” 时, 控制流程将跳转 “return false” 语句。另外, 表 6-1 不包括每个条件语句的可能结果的所有组合。重申, 这是因为我们假定控制流程的“短路”。如果 “短路” 并不是一种普遍的惯例或不能被假定, 那 *IsInvalidGregorianCalendarDate*

函数将需要 16 个测试来运行每个条件语句的可能结果的所有组合。

条件测试小结

条件测试提供了对于多个布尔子表达式组成的复合条件语句的控制流程更好的敏感度。因此，条件测试在应对含有复合条件语句的函数时包含并超越了块测试和决策测试。条件测试 旨在评估复合条件语句每个子表达式的真假结果。因为微软的惯例是执行“短路”的，所以条件测试不验证复合条件语句的真假结果的所有组合。通常没有一个好的理由不执行“短路”，但是测试人员应该和开发人员联系来决定“短路”是否是一个标准的惯例。

基础路径测试 (Basis Path Testing)

路径测试试图遍历程序中所有可能的途径。一个函数中的循环结构构成一个对路径测试特别具有挑战性的问题，因为每次控制流程通过一次循环体就被认为是一条不同的路径。因此，在任何较复杂的函数中测试一切可能的路径是不切实际的。考虑图 6-9 所示的函数，一个循环结构中有四个条件语句，控制流经循环最多达 20 次。如果我们认为，每次通过循环的路径是一条不同的路径，那就有 5^{20} 个或约 100 万亿可能的测试。即使每个测试只花 1 毫秒，用穷举测试也要花 3000 多年！

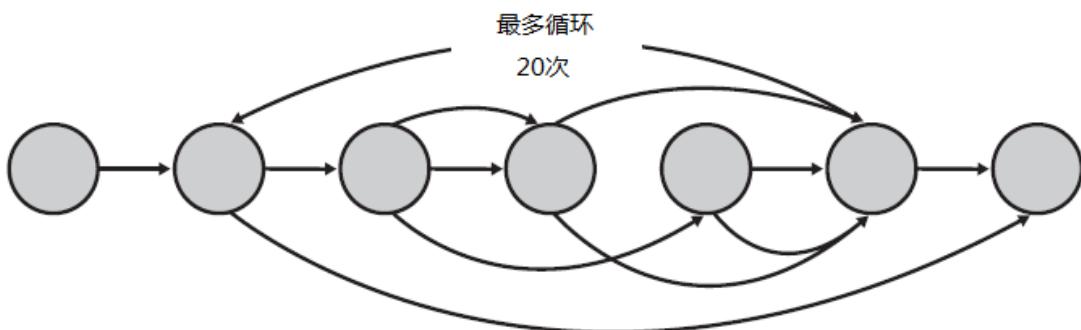


图 6-9 一个可循环多达 20 次的函数的控制流程图

测试函数中每一个可能的路径是数学上不可行的，而且一般也不实用。Thomas McCabe 提出了一个可能的方案。McCabe 假设圈复杂性（cyclomatic complexity）度量和作足够的路径测试所需的测试数目有直接相关性。这一假设的例外存在于同一条件语句被评估两次的函数中。当相同条件语句被评估时，可能的基础路径和相应的基础路径测试的数目要小于圈复杂性度量。这方面的一个例子在 `IsValidMod10Number` 函数中有描述，我们会在本章后面谈到。

什么是圈复杂性？

圈复杂性是用来帮助开发人员测定他们的函数复杂性的一个测量单位。这是一个常用的度量，用于在软件开发生命周期中评价一个模块的潜在可靠性、可测试性和可维护性。但它也可被测试人员用来确定最低的测试数量，从而对一个函数的控制流程进行比块测试和决策测试更加严格的测试。圈复杂性衡量控制一个模块或函数流程的决策逻辑。计算圈复杂性的公式是 $V(G) = \text{边的数目} - \text{节点数} + 2$ 。但如果您必须手工计算圈复杂性，一个更容易的公式是简单数一下条件语句的数目并加 1 ($V(G) = p + 1$)。

一条基础路径的定义为通过一个函数的一条线性独立的路径。一个完整的关于线性独立的解释并不简单，且在本书范围以外。然而，为了简单起见，可以认为线性独立的基础路径是通过一个函数的独特路径的有限集合。例如，在有一个条件语句的函数中有两个线性独立的基础路径，如图 6-10 的控制流程图所示。线性独立的基础路径集合的线性组合代表了所有可能的通过某一函数的路径，而且任何其他通过该函数的路径都是这个基础路径集合的超集（*superset*）。基础路径测试这一结构测试技术提供了一个有效的方法来处理穷举性路径测试几乎无法解决的难题。

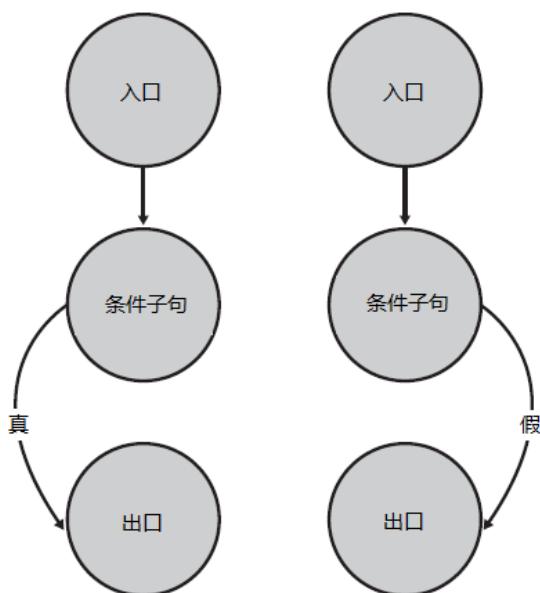


图 6-10 线形独立的控制流程图

McCabe 和 Watson 还提出了基准路径技术（baseline path technique），测试人员可以使用此技术来系统地查明通过一个函数的线性独立的基础路径集合。实际上测试人员可用两种不同的方法查明这个基础路径集合。简化基准路径技术(simplified baseline path technique) 也许是广为人知的和最系统化的方法。简化基准路径技术采用了一个有条理的流程，测试人员需要完成如下步骤：

1. 确认从被测函数入口到出口的最短基准路径（经过最少条件语句的路径）。

2. 回到函数入口。
3. 跟踪控制流程，从入口点到第一个未被先后评估为真和假两种结果的条件语句。
4. 改变该条件语句的结果（由真改到假，或由假到真）
5. 按最短路径从这个条件语句到函数出口。
6. 重复步骤 2 至 6，直到所有的基础路径（等同于圈复杂度量）被定义。

虽然简化基准路径技术能有效快速地查明的一个基础路径集合，但它会导致限制过于严格的基础路径集合，或者该集合可能包括不可能的通过路径。因此，为提供一个替代性的和更富创造性地方法来查明基础路径，McCabe 和 Watson 还提出了实用基准路径技术（**practical baseline path technique**）。实用基准路径技术也是一个系统化的流程，但测试人员并不跟踪通过函数的最短路径，而是使用下列步骤：

1. 确定一个有可能的功能性的通过函数的基准路径，它代表一个非常可能的通过函数的控制流程。该流程是运行时最常见的，或是最重要或最关键的路径。
2. 回到函数入口点。
3. 跟踪控制流程，从函数入口点到第一个未被先后评估为真和假两种结果的条件语句。
4. 改变该条件语句的结果（由真改变到假，或由假改变为真）。
5. 沿一条包括最大数量的被基准路径所遍历的条件语句到函数出口点的路径。
6. 重复步骤 2 至 6，直到每一个条件语句都被先后评估为真和假两种结果，而且所有基础路径都被定义。

基础路径测试的有效性在一个样板性范例中被清楚地展示，该范例刊登在国家标准与技术研究所特别出版刊物（National Institute of Standards and Technology Special Publication）500-235²⁷。

CountC 函数

²⁷Arthur H. Watson and Thomas J. McCabe, *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*, NIST Special Publication 500-235 (Gaithersburg, MD: National Institute of Standards and Technology, 1996).

```

// This function counts the instances of the letter C
// in strings that begin with the letter A
A0  private static int CountC (string myString)
{
    int index = 0, i = 0, j = 0, k = 0;
    char A = 'A', B = 'B', C = 'C';
    char[] strArray = myString.ToCharArray();

A1    if (strArray[index] == A
{
A2        while (++index < strArray.Length)
{
A3            if (strArray[index] == B)
{
                j = j + 1;
}
A4            else if (strArray[index] == C)
{
                i = i + j;
                k = k + 1;
                j = 0;
}
}
A5        i = i + j;
}
return i;
A5  }

```

该 *CountC* 函数包含了四个条件语句。如果我们只想评估每个条件语句的真假结果至少一次，我们只需要两个测试。表 6-2 是两个测试的真值表。第一个测试传递了一个 D 的参数，而第二个测试传递给 *CountC* 的参数是 ABCD。

表 6-2 *CountC* 函数决策测试的真值表

测试	参数	条件子句	结果					
			A1	A2	A3	A4	期待值	真实值
1	D	假					0	0
2	ABCD	真	真/假	真/假	真/假	真/假	1	1

图6-11的控制流程图展示了在使用表6-2中决定测试2时，*CountC*函数中每个条件语句所产生的真假结果。决定测试2遍历了如下路径A0 → A1(T) → A2(T) → A3(T) → A2(T) → A3(F) → A4(T) → A2(T) → A3(F) → A4(F) → A2(F) → A5。

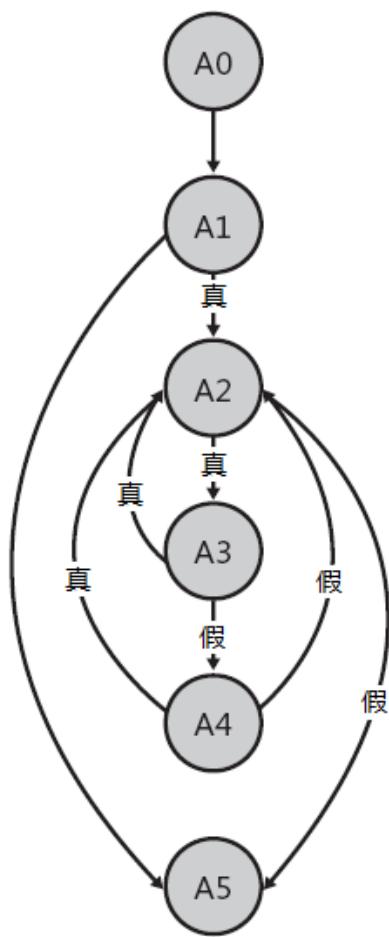


图 6-11 *CountC* 控制流程图

然而，运用简单基准路径方法来确定一套基础路径，并预测每个路径测试的结果会导致两个错误。该 *CountC* 函数有四个条件语句和第五级圈复杂性。因此，这个函数有五个基础路径。表 6-3 列出了一系列基础路径，有关测试输入和预测的结果。图 6-12 展示了每一个线性独立基础路径。

表 6-3 *CountC* 函数的基础路径表

基 础 路 路经 经		输入值	期待值
1	A0→A1(F)→A5	D	0
2	A0→A1(T)→A2(F)→A5	A	0
3	A0→A1(T)→A2(T)→A3(T)→A2(F)→A5	AB	0
4	A0→A1(T)→A2(T)→A3(F)→A4(T)→A2(F)→A5	AC	1
5	A0→A1(T)→A2(T)→A3(F)→A4(F)→A2(F)→A5	AD	0

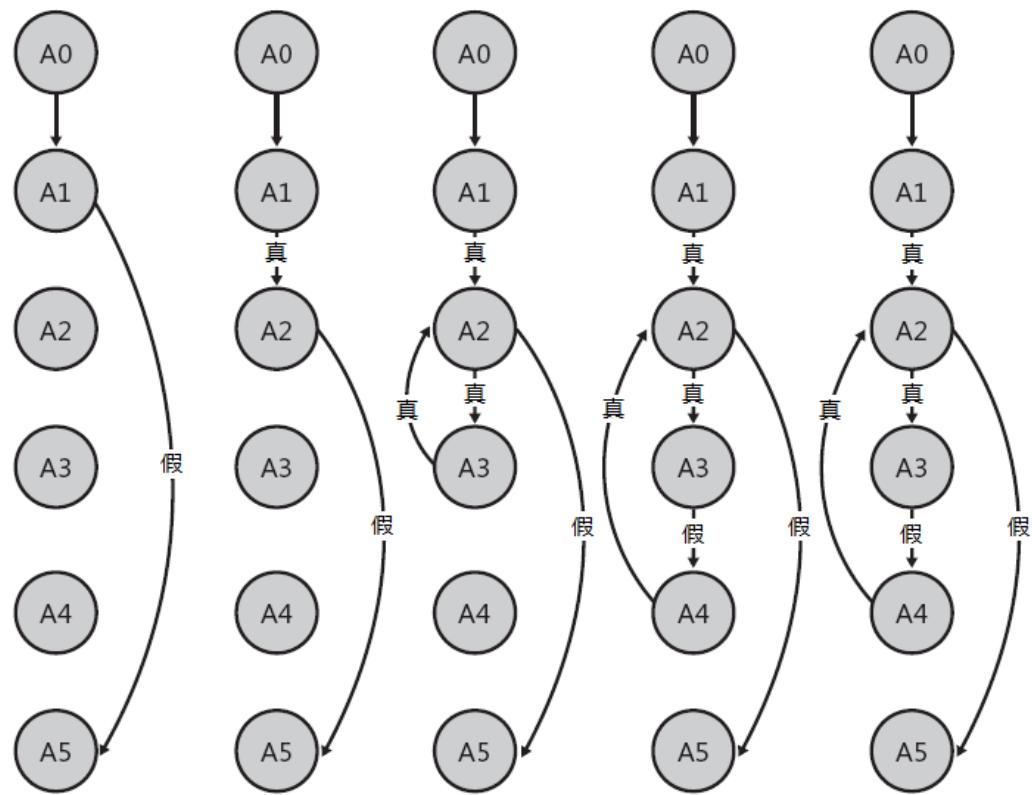


图 6-12 CountC 函数的线性独立基础路径

表6-4显示CountC函数每个条件语句的独立真假结果至少运行过一次。但当我们运行基础路径测试和系统地遍历每个线性独立路径时，我们马上会意识到这个简单的函数中有两个错误。第一个错误是在基础路径测试3中发现的。

预料的结果是0，但其实是1。第二个错误是在基础路径测试4中找到的。预料的结果是1，但返回值其实是0。

表6-4 CountC函数的真值表

测试	参数	条件子句				期待值	真实值
		A1	A2	A3	A4		
1	D	假				0	0
2	A	真	假			0	0
3	AB	真	真/假	真		0	1
4	AC	真	真/假	假	真	1	0
5	AD	真	真/假	假	假	0	0

这只是一个来解释基础路径测试原理和如何有条理地分析函数的简单例子。有些人看CountC函数时会把焦点放在代码质量差上面。例如，他们会问，为什么我们要检查字符B？或者，他们会意识到变量k什么都没做。有趣的是，我们有时会在发布代码中看到这样的情况。该

条件语句可能将来会被使用，或者该条件语句曾经是已被移除的功能的一部分。*k*变量是死代码的一个例子。有时人们仅注重写得不好的代码，而一个专业的测试工程师能为最费解的函数设计结构测试。

当然，这两个错误可能在行为性测试过程中会被发现。但死代码无法从用户界面上用行为性或试探性的测试来探测到。如果字母*B*计数的错误被发现，测试工程师无法确定在超过十万个其他Unicode字符中哪些字符也可能在递增返回值。因此，你会选择十万多Unicode字符中的哪些？你会尝试所有的吗？还是用结构性测试来减少测试数量，提高效率，改善测试覆盖率，提供更多合格的信息，辨明不同类型的错误，并帮助我们理解它们为什么发生，和有可能有助于降低整体风险？

*IsValidMod10Number*函数是另一个好例子。做为一个真实世界的例子，它显示了基本路径测试与块测试和决策测试相比，基本路径测试在控制流程中提供了更大的敏感度。

IsValidMod10Number.cs

```
// This function takes a string input, converts it to a number array, and applies a
// mathematical formula to check the number's ability to meet the criteria for a valid
// Mod 10 number and returns a Boolean result.
A0 public static bool IsValidMod10Number (string number)
{
    int[] numberArray = new int[number.Length];
    bool checkBit = false;
    int sumTotal = 0;

A1    for (int i = 0; i < number.Length; i++)
    {
        numberArray[i] = int.Parse(number.Substring(i, 1));
    }

A2    for (int index = numberArray.Length - 1; index >= 0; index--)
    {
A3        if (checkBit)
        {
            numberArray[index] *= 2;
A4        if (numberArray[index] > 9)
            {
                numberArray[index] = 9; // correct statement is number -= 9;
            }
        }
        sumTotal += numberArray[index];
        checkBit = !checkBit;
    }
    return sumTotal % 10 == 0;
A5 }
```

什么是一个可以被 10 取模的数字？

Hans P. Luhn 取得了一个简单的校验和算法的专利。这个校验和算法被称为“模数 10”或“Luhn 公式”，常用来验证某些 ID 号码（用于识别用途的号码）。基本上，从个数位算起，由右往左，一个数字的每个数位的数乘以 1 或 2。如果一个数位的数计算结果大于 10，那就把结果的两个数位的数相加（或从结果减去 9）。所有计算的数字的总和再除以 10。如果没有剩余商数（总和 $\% 10 == 0$ ），那就等于通过了模数 10 算法。该算法常用于验证如信用卡号，和加拿大社会保险号的号码。

我们可以用一个输入值，在 *IsValidMod10Number* 函数里达到 100% 的区和决策代码覆盖率。测试输入 4291 会在每个条件语句评估真假结果，并向主调函数返回一个真的布尔值，这是预期的结果。第二次试验用 1 的输入会向主调函数返回一个假的布尔值。表 6-5 显示，两项测试通过后达到 100% 的区和决策代码覆盖率，但都错过了 *IsValidMod10Number* 函数中关键的错误。

表 6-5 *IsValidMod10Number* 函数的决定测试真值表

测试序号	参数值	条件子句				结果	
		A1	A2	A3	A4	期待值	真实值
1	4291	真/假	真/假	假/真	真/假	真	真
2	1	真/假	真/假	假		假	假

IsValidMod10Number 函数有四个条件语句。使用条件语句总数加 1 的简化公式，我们计算出第 5 级圈复杂性。逻辑上有 5 个基础路径，然而，在这个例子中，我们只能够遍历四个基础路径。这是因为不可能在函数里重复决定，而制造出有冲突的结果。如果两个条件语句评估了同样的布尔运算式，相同的条件语句的结果必须是相同的。在 A1 和 A2 中的条件语句是相同的，因为它们一个是把字符串的字符数目与递增指数相评估，另一个是把字符串转换成数组后数组长度与递减指数相评估。这种情况下，字符串的字符数目和数组的元素数目相等，所以条件语句基本是一样的，我们只有 4 个基础路径，如图 6-13 所显示。

表 6-6 显示的四个基础路径测试反映了五个关键错误中的三个。（两个不能被基础路径测试所查出的错误包括当参数值是负数或非整数时所出现的未处理的格式异常，和当参数值是一个大于 2,147,483,647 的整数时所出现的未处理的溢出异常。）

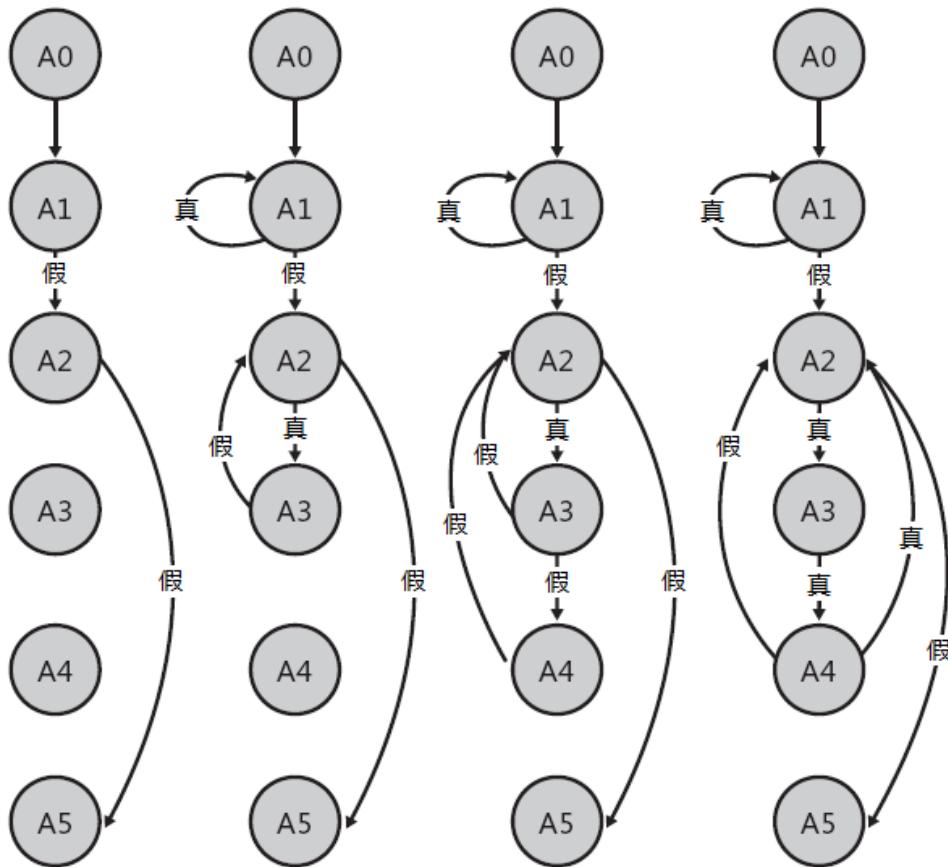


图 6-13 *IsValidMod10Number* 函数的基础路径集

表 6-6 *IsValidMod10Number* 函数基础路径测试的真值表

测试序号	参数值	条件子句				结果	
		A1	A2	A3	A4	期待值	真实值
1	空	假	假			假	真
2	0	真/假	真/假	假		假	真
3	10	真/假	真/假	假/真	假	假	假
4	59	真/假	真/假	假/真	真	真	假

头两个被基础路径测试所发现的关键错误的出现也是因为错误的输入验证。如果输入参数是一个空字符串或字符串 0 (注: 字符串 `null`)，返回的布尔值会是真而不是假。第三个关键错误被基础路径测试 4 所发现，使用最低有效数字会导致 A4 评估为真，而满足模数 10 算法这个数学公式。

当然，我们如果传递字符串 50 到 `number` 参数，就可能找不到函数的关键错误。这是一个将遍历与测试 4 一样的结构路径，能导致条件语句 A4 为真的最低输入值。在这种情况下，预期的结果和真正的结果都是假的。这个例子突出了与“杀虫剂悖论” (pesticide paradox) 相关的问题，也提示了测试工程师不能依赖一个单一的方法或技术来进行有效的测试。

这个例子也说明代码的透视性可以显示隐藏的边界条件或其他角落案例。例如，通过应用边界值分析的原理，我们能发现几个子边界的条件。例如，假设程序开发工程师验证一个参数值为大于 0 的整数后，传递给 `IsValidMod10Number` 函数。但边界条件极端范围的最低限度是 1，一个无效的数字。但最低有效的号码是 18，根据模数 10 算法，是相等类划分的最低有效数字中所确定的边界条件。把输入值 18 放在这个错误的函数将返回与预料一样的真的值。然而，能遍历条件语句 A4 的真正路径的最低有效数字是 59。所以，59 是一个子边界条件，是在有效数字范围中独特的值。因为它的处理不同于其他有效数字的处理。在这个例子中，确定边界条件和利用它们作为结构测试的测试输入可更有效率地帮助识别特定类型的错误。

基础路径测试小结

基础路径测试与块测试和决策测试的不同之处在于：每一个条件语句的每个结果必须被独立地测试。例如，用一个（设计得不好的）测试来评估一个 `while` 或 `for` 循环中的条件语句所产生的真假值，就可以满足块和决策测试的要求。但是，基础路径测试需要一个测试把控制流程绕过循环，另一个测试的程序控制流程经过循环结构。所以，类似于条件覆盖范围，基础路径测试包含了块和决策测试。

与块和决策测试相比，当分析有简单条件语句的函数，特别是使用循环结构的函数时，基础路径测试提供了更好的控制流程敏感性，但是，当评估一个复合条件语句时，基础路径测试看起来会产生类似条件测试的结果。

本章小结

结构测试技术是一套系统过程，它能帮助测试人员在分析程序代码，提高代码覆盖率时更有效的设计测试。但是，我们决不主张把结构测试作为唯一或最先使用的办法来测试软件。结构测试技术的目的是支持和加强其他测试方式和方法。通过设计和运行能遍历其他测试方法所不能遍历的代码路径的测试，结构测试还可以提供更深入的信息和降低风险。这是很重要的，因为逻辑错误与一个代码路径的执行概率有着反比关系。如果我们不执行经过某个代码路径的测试，我们必须承担 100% 的风险。如果我们执行经过该代码路径的测试，那我们的风险就会降低一些百分比，我们越多地成功遍历代码的一条路径，特别是用不同数据，我们就越能降低我们所面临的风险。

行为测试方法、系统化的功能测试方法和结构测试方法，它们每一个都旨在提供不同类型的信息和从不同的角度评测软件。复杂的问题需要从大量不同的视角来提供合格的信息，以满足决策者的需要，从而更好地评估风险，作出明智的业务决策。结构测试只是提供一个不同的角度来分析复杂的问题。但结构测试是一个团体组织的投资。这一级别的测试需要花费测试人员更多的时间和技能去设计白盒结构测试。然而，结构测试可以被证明是非常有价值的，它能够降低长期成本，使那些需要高度可靠性的非常关键或复杂的系统的风险降到最小。

第 7 章 用代码复杂度分析风险

阿伦·培智

我的叔叔弗兰克是个很棒的渔夫，他一辈子都在蒙大拿州的河流里捕鱼。他总是十分熟悉什么样的钓竿、钓线和渔饵一起使用能够捉到鱼，是朋友当中公认的捕鱼专家。然而，无论他对捕鱼有多在行，如果河里没有鱼，他也只能空手而归。他明白有了好的装备和诱饵还不行，所以他還知道如何观察河流的深度和流向，利用这些信息来准确地预测鱼儿会在哪里出现。熟练的捕鱼技术和对渔区的分析策略，两者结合令他无往不胜。

一些测试技术比如边界值分析、以及结对测试（pairwise），可以有效地帮助我们在尽量少增加风险的同时，减少所需测试用例的数目。然而常见的问题在于产品缺陷，并不是平均分布在代码里的。在一些典型的软件项目里，总有些组件比其他组件存在更多的缺陷。就像我的叔叔用了很多不同的技术去预测鱼儿在哪里一样，软件测试的一个必要环节是预测哪个项目区域存在着更多缺陷，并有针对性地投入测试力量。

风险行业

测试常常是一个管理风险的过程。基于风险的测试，就是基于缓解产品中潜在风险的测试方法。这种方法倾向于把可用的测试资源集中在最需要的区域。从某种意义上讲，所有测试都是基于风险的。既然无法进行“面面俱到”的测试，那么作为测试工程师，我们就需要基于一系列的标准，有选择地集中投入测试力量。

意大利经济学家 *Vilfredo Pareto* 创建了一条公式用来描述国内财富不规则分配的现象，即 20% 的人掌握 80% 的财富。很多人都相信 Pareto 定律²⁸（80/20 法则）也同样适用于软件项目。在一般应用上，Pareto 定律认为在很多度量标准下 80% 的结果源自 20% 的原因。如果应用在软件领域，Pareto 定律可以解读为，80% 的用户使用仅仅 20% 的功能，80% 的缺陷存在于 20% 的产品中，或者说 80% 的运行时间耗费在 20% 的代码上。一方面，基于风险的测试方法是尝试归类出哪部分产品组件拥有更普及的用户场景，从而投入更多的测试力量；另一方面则是这种方法的风险性，它依赖于对测试重点的精确选择从而忽略了一个事实，还是有用户会去使用那些 20% 以外的功能和代码。

对最有可能产生缺陷的产品部分编写更多的测试用例是基于风险测试的另一种应用。就如同我的叔叔知道向哪里投竿更容易捕到鱼一样，基于风险的测试知道运行哪些用例更有机会发现产品的严重缺陷。

复杂问题

复杂性是无处不在的。我还记得小时候，自打我刚刚够格碰烤炉开始，我就开始做巧克力曲奇。我用的是我祖母的配方，那张配方只写了简单的几样原料。我都不大去参考那张配方，但做出的曲奇每次都非常的棒。原料的份量始终如一（两根牛油棒，两杯面粉，一杯红糖一杯

²⁸ Wikipedia, “Pareto Principle,” http://en.wikipedia.org/wiki/Pareto_principle.

白糖，一包碎巧克力），烘烤时间也是一个大概的数字（10分钟）。我成功的关键在于，配方是简单的。除非天灾人祸（比如黄油变质了，烤炉坏了），要我做出难吃的巧克力曲奇还真不容易。

长大以后我仍旧喜爱做吃的，但当我想要如法炮制出一道美食，或者受烹饪书上图片的鼓舞想要复制一份食谱的时候，却时常遇到形形色色的问题。因为一旦面对大堆的原料和步骤就太容易出错了。通常我可以顺利过关，但也有很多次，当有三个锅子在火上烤的时候，我突然发现我做错了一整个步骤，或是读错了配方，把一茶匙的红辣椒粉读成了一汤匙放进了汤里。配方的复杂性和一下子同时处理多个配方让我更容易犯错。与此类似，复杂的代码和不同代码的复杂部分之间的交互也通常会更容易地带来错误。

有时，我们用一种寻找高复杂度代码的办法来预测缺陷出现的地方。复杂的代码容易产生更多的缺陷，简单的代码少些。复杂的代码还有个很大的缺点就是更难维护。代码复杂度是衡量代码“难度”的重要标准。代码复杂度越高，测试也将越困难。

作粗略的代码审阅时凭感觉或主观地来衡量代码复杂度有时就已经足够了。“**代码异味**”是敏捷编程领域中的一个术语，用来描述因为函数太大或庞大的依赖度而可能过于复杂的代码。异味通常是一种主观度量方式，因编程语言或环境而异，也用在识别可能要被重新分解或重写的代码上。

别碰它

我曾经在视窗95开发组进行网络组件的测试工作。在测试过的一个组件里，我们会偶尔发现微小缺陷，而在极少情况下，Beta版的用户也会报告同一个组件的问题。幸运的是，所有这些缺陷都是小问题，因为都被解决为“无需修复”，意即这个缺陷在本产品发布中将不会修复，或者在将来的任何发布中也不会。其实如此解决组件缺陷是出于这样一个“简单”原因：当初开发此代码的开发员已经离开公司了，而现在我们“不敢碰”这些代码。这些代码过于困难和复杂，以至于没有一个独立开发人员愿意承担惹出一大堆新缺陷的风险去修复它。

所幸这个组件在之后的微软视窗操作系统中被彻底重写了，但是我时常仍然听到一些因为没人敢碰而放弃修复的故事。

阿伦·培智

任何时候只要不在软件初始设计和实现中考虑简单性原则，代码就有可能成长得非常混乱而难以维护。主观的方法比如直觉判断可以有效地判定复杂的代码，但是也有一些客观的方法来衡量代码的复杂度，从而帮助确定产品缺陷可能会隐藏在哪里。

最简单的代码复杂度测量方式可能是代码行数（LOC）。一个1000行代码的程序通常没有一个10000行代码的程序来得复杂。直接做算术你可能会相信10000行的项目产生的缺陷会是1000行代码程序的10倍，但是在实践中，10000行代码程序经常会隐藏不止10倍的缺陷。不过，由于计算代码行数方法的不同（参见后面的章节），以及大量的外部因素，代码行数计算基本上不会作为一种精确的复杂度衡量方式。有许多其他的复杂度衡量方式可以用来预测缺陷所在，其中一些我们会在后面的章节中解释。

计算代码行数

一个程序中有多少行代码？为什么这个问题会这么难回答？让我们从简单的例子开始：

```
if (x < 0)
    i = 1;
else
    i = 2;
```

上面的代码有四行。但如果用下面的格式表示的话，是几行？

```
if (x < 0) i = 1;
else i = 2;
```

你很难说第二个例子有四行——它只有两行，或者至少说它是格式化成两行。两个例子的格式我个人都不喜欢，我会这样写：

```
if (x < 0)
{
    i = 1;
}
else
{
    i = 2;
}
```

那么，这个例子到底两行，四行还是八行？答案取决于你问谁。一些代码行数计算方式只计入语句（以 C 语言为例，只计入以分号结束的行），另一些计算方式计入除空行和注释外所有的行，还有一些仍旧计入代码产生的汇编指令（前面的每个例子都产生同样的汇编指令，或者在编译托管代码时的中间语言）。

尽管有很多计量 LOC 的方法，你却只需要挑选一种你喜欢的，并从一而终。根据代码行数来测量程序长度很少作为主要测量方法，但是它计算简单，并且可能帮助一些任务比如比较产品的两个版本或者项目的两个组件间的差别。

回路复杂度

计算机程序包含了数以千计的判断：如果这件事发生了，就做那件事...除非另外的事情发生了，否则首先做这件事。包含了很多选择和判断的程序往往更容易包含漏洞也更难测试。确定程序中判断的数目的最常用的方法中，有一种称为**回路复杂度** (cyclomatic complexity) 的度量方法，这种方法由 Thomas McCabe²⁹ 开发，它是辨别函数中线性独立路径（或者判断）数目的度量方法。一个没有包含条件操作（比如条件语句，循环，或者三元算子）的函数在整个程序中只有一条线性独立的路径。条件语句在程序流中加入了分支也就在函数中创建了另外的路径。

随着回路复杂度的增加，程序的维护成为了一个问题。心理学家的研究显示平常人可以在短期记忆中同时保留 5 到 9 条信息。所以当相互关联的选择超过 5 到 9 条时，程序员就很有可能出错。

²⁹ Thomas McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering SE-2, No. 4 (December 1996), 308-320.

能会在修改代码时犯错。特别多的判断会导致代码很难维护也更难测试。

计算 McCabe 复杂度最常用的方法是首先基于源代码创建一个控制流程图，然后基于此图计算出结果。比如，考虑在代码清单 7-1 中的代码。

代码清单 7-1 简单的回路复杂度

```
int CycloSampleOne(int input)
{
    int result;
    if (a < 10)
        result = 1;
    else
        result = 2;

    return result;
}
```

这段代码被表示成图 7-1 中的控制流程图。

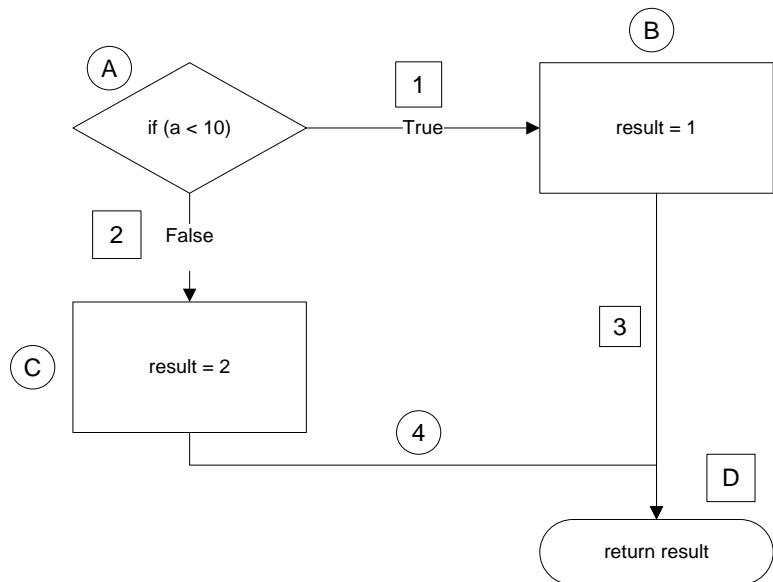


图 7 - 1 代码列表7-1中简单回路复杂度例子的控制流程图

基于控制流程图，McCabe 识别出计算回路复杂度的公式是一个边（edges）-节点（nodes）+2 的函数。在图 7-1 中，节点以平面图形显示，而边则作为线连接那些表示程序中可能路径的节点。图 7-1 中标记从 A 到 D 的节点代表了函数中的语句。用之前提到的公式，函数中有 4 个节点，4 条边或者说节点中的路径。分析表明这个函数的复杂度是 2（边（4）-节点（4）+2=2）。有一个计算回路复杂度准确而又快速的方法是简单地将条件（判断）语句的数量加 1。在先前的例子中，函数中只有一个条件（`if(a<10)`），所以回路复杂度就是 2。

代码清单 7-2 复杂的回路复杂度

```

void CycloSampleTwo(int value)
{
    if (value !=0)
    {
        if (value < 0)
            value += 1;
        else
        {
            if (value == 999) //special value
                value = 0;
            else //process all other positive numbers
                value -= 1;
        }
    }
}

```

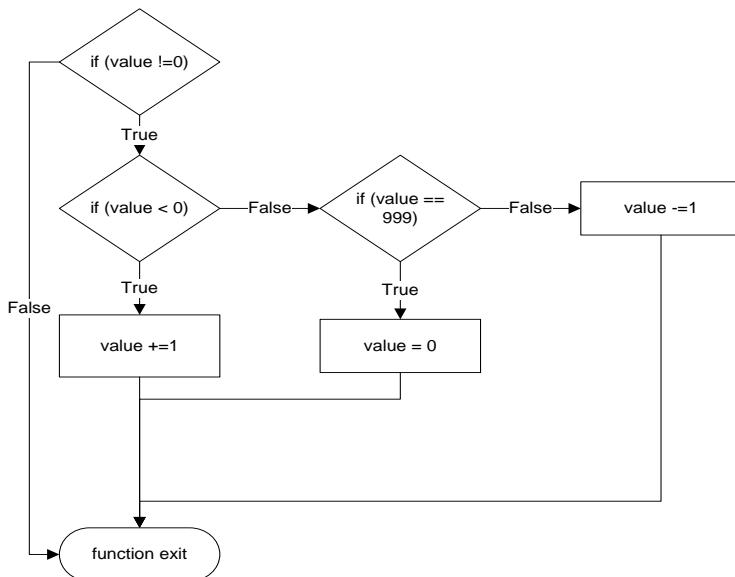


图 7-2 对于代码清单 7-2 中稍微复杂一些的回路复杂度例子的控制流程图

之前的两个例子很有可能比你在产品中发现的大部分代码都要简单很多，因此针对这些例子代码创建其测试用例全集可能会很快。对于简单的函数，计算回路复杂度是很直接而且可以简单地通过手工来完成。我们也有很多工具（包括由 McCabe 开发的 McCabe IQ³⁰，或者免费工具诸如 CCCC、NDepend 和 Code Analyzer）用来针对那些有很多判断点的大规模函数，或者自动计算大规模代码的回路复杂度。对于托管代码开发，市场上有几款免费工具用来计算指定代码块的回路复杂度，包括 Sourcemonitor、Reflector 和 FxCop。
回路复杂度主要用在衡量函数的可测试度。

表 7-1 基于回路复杂度的风险分析

³⁰ McCabe IQ, 主页, <http://www.mccabe.com/iq.htm>

回路复杂度	与之联系的风险
1–10	低风险的简单程序
11–20	中等的复杂度和风险
21–50	高复杂度和风险
50+	非常高的风险/不可测试

以上只是参考值。我曾经看到过回路复杂度低的代码到处是缺陷而回路复杂度超过 50 的代码却被很好地测试并且几乎没有什么缺陷。

Halstead 度量

Halstead 度量是一套完全不同的复杂度度量，基于对程序中语法要素的以下 4 个度量：

- 独特算子（Operators）的数量（n1）
- 独特算域（Operands）的数量（n2）
- 所有算子出现的总数（N1）
- 所有算域出现的总数（N2）

一套复杂的度量集合是由这些度量值推导的。例如，代码长度的度量值可以由 N1 加上 N2 得到。Halstead 度量也可以通过接下来的公式计算难度（Difficulty）度量： $(n1/2)*(N2/n2)$ 。

下面的例子代码包含了 6 个独特算子和 4 个独特算域，其分别被引用了 6 次和 12 次。表 7-2 表明了如何来做这些计算。

```
void HalsteadSample(int value)
{
    if (value != 0)
    {
        if (value < 0)
            value += 1;
        else
        {
            if (value == 999) //special value
                value = 0;

            else //process all other positive numbers
                value -= 1;
        }
    }
}
```

表 7-2 Halstead 度量的例子

算子	算域
----	----

算子	数量	算域	数量
1 !=	1	Value	6
2 <	1	999	1
3 + =	1	0	3
4 ==	1	1	2
5 =	3		
6 -=	1		
总计	6 (n1)	8 (N1)	4 (n2)
			12 (N2)

- $n1 = 6$
- $n2 = 4$
- $N1 = 8$
- $N2 = 12$

用 Halstead 度量进行计算,这个函数的长度(Length)是 $18(N1+N2)$,难度系数($(n1/2)*(N2/n2)$)是 $9((6/2)*(12/4))$ 。

和回路复杂度相似, Halstead 度量的价值在于衡量程序的可维护性,但是对于这个值的范围,从“不可靠”³¹到“属于可维护性最高度量”³²,业界有一些不同的观点。与回路复杂度一样(和其他几乎所有我能想到的度量),运用这种度量主要是用来标识可能需要返工或者额外的分析的代码。

面向对象的度量

面向对象度量是在各种语言(例如 C++, Java 和 C#)中类和类结构相关的度量。其中最流行的面向对象度量是由 Chidamber 和 Kemerer³³创造的,即通常所说的 CK 度量。CK 度量包括了以下的内容:

- **每个类的权重方法 (WMC)** 一个类中方法的数目
- **继承树的深度 (DIT)** 一个类所继承的类的数目
- **对象类之间的耦合 (CBO)** 一个类引用其他类的方法或者实例变量的数目

面向对象度量的提倡者认为那些引用很多方法,拥有很深继承性树状结构,或者过度耦合的类更难测试和维护,也更容易包含漏洞。

在面向对象的编程中,扇入(fan-in)和扇出(fan-out)度量分别用来计算有多少类调用到

³¹ Capers Jones, "Software Metrics: Good, Bad, and Missing," Computer 27, no. 9 (September 1994): 98–100.

³² P. Oman, HP-MAS: A Tool for Software Maintainability, Software Engineering (#91-08-TR) (Moscow, ID: Test Laboratory, University of Idaho, 1991).

³³ S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design," IEEE Transactions in Software Engineering 20, no. 6 (1994): 476-493

某一个特定的类和有多少类被某一个特定的类调用。例如，如果一个类包含的方法被其他 5 个类调用而且这些方法也调用了其他 10 个类，那么它的扇入就是 5，扇出就是 10。这些度量值作为可维护性度量常常是非常有效的，同时也表明了需要额外的测试的区域。例如，如果一个类被很多类调用（很高的扇入度），那么很有可能在这个类中任何代码的改动都会导致在其中一个调用类中产生新的漏洞。

当被用在函数或者模块层面，扇入和扇出度量对于非面向对象编程也很有价值。那些被很多函数调用的函数很可能是一些最难维护和测试的函数，往往不能被轻易地修改。在我的职业生涯中所见到的最极致的例子便是 Windows 应用程序编程接口（API）函数。很多 Windows 核心函数被上万个应用程序调用。对于这个函数的改动，甚至最细微的改动都有可能导致其中一个调用函数突然失灵。维护任何有高扇入度的函数或者模块都需要特别小心。从测试的角度讲，这些恰恰是需要积极预防的地方。测试人员需要及早地在程序中发现哪些函数，模块，或者类具有最高扇入度，从而将测试力量集中在这些区域。

另一方面，扇出度量值表明了程序中有多少依赖关系。如果一个程序或者类调用了 10 个或者 20 个方法，那就意味着这 10 个或者 20 个不同的方法的改动会影响到你的程序。如果是其他程序员或者团队完全负责这些被调用的函数时，这个问题就常常会变得更加严重。

复杂度在 Windows 支持工程团队的应用

Windows 支持工程（SE）团队负责维护所有已经发行的 Windows 版本，包括修补程序，安全补丁，更新（严重和不严重），安全补丁集，功能包，和服务包。

每当一个修补程序发布，SE 团队必须决定在构成 Windows 的 4000 多个二进制文件中有多少文件需要测试。对于一个修补程序，只有很短的时间来保证一个二进制文件中的一个改动不会影响其他的二进制文件，根本没有足够时间对每一个修补程序测试 Windows 所有的二进制文件。于是他们综合各种复杂度度量来排列每一个二进制文件风险程度，这个排列是基于针对特定修补程序的改动以及这个改动所关联的单个二进制文件历史上曾经存在缺陷的可能性进行的。

一旦排列了风险，他们就采用一种非常保守的测试方式并将风险最小的 30% 的二进制文件从回归测试中去除。这就意味着 1000 多个二进制文件不需要测试，从而将测试力量集中在其余更高风险的文件上。随着这个过程的不断深入，他们能够排除更多的二进制文件，既提高了测试效率又有信心保证改动不会造成漏网的缺陷。

——Koushik Rajaram

回路复杂度高并不表示缺陷累累

能够量化给定软件块的复杂度并不意味着测试团队一定要采取行动。几乎完美的代码在上述的每一个度量上还是有可能有很高的复杂度。我常将这些度量称之为烟雾警告度量（smoke alarm metrics）。当烟雾警告开始响起时，它并不保证一定有火灾，但是它的确指示你应该搜寻火苗并且采取相应的行动。类似地，当复杂度度量高时，并不意味着代码缺陷累累或者不可维护；但是的确意味着你需要仔细看一看代码。

举例来说，考虑回路复杂度和包含一个长 `switch` 语句的代码，例如 Windows 编程中常用到的消息循环。在循环中的每一个 `case` 语句都创建了一个独立的路径也就增加了一个回路复杂度（和所需要的测试用例的数量），但是没有造成应用程序难以测试。以微软画图板（Microsoft Paint）为例，画图板相对其他图形操作程序非常简单，但是它有 40 种菜单选择，16 种绘图工具选择，和 28 种颜色选择。我没有看过画图板的源代码，但是如果它是用单个 `case` 语句来处理这些选择的代码，我一点也不会惊奇！在此基础上加上程序必须处理的绘画和尺寸相关的消息，你就会有一个被一些复杂度度量告知根本不可测的函数，这时你该抓狂了（但也许你不应该）。

下面的代码展示了一个典型的 Windows 消息循环的一部分。根据设计，消息循环有很多 `case` 语句从而造成很高的回路复杂度。消息循环中肯定会有缺陷发生，但是肯定不会接近到回路复杂度度量所暗示的那样。

```
int HandleMessage(message)
{
    switch (message)
    {
        case Move :
            // code omitted...
            break;

        case Size :
            // code omitted...
            break;

        ...
        // dozens more deleted
    }
}
```

虽然如此，在很多其他情况下，你还是会发现那些有高复杂度的源代码区域也是那些包含更多缺陷的区域。无数的研究已经表明了复杂度度量和漏洞之间存在非常高的关联度。通常复杂度也和源代码的可维护性联系在一起。总的来说复杂的代码对于维护人员而言极端困难。熟悉这些代码需要更多的时间，同时即使不增加代码的复杂度，修补缺陷和增加新的功能也会非常艰难。

如何利用复杂度度量

还有其它几种衡量代码复杂度的方法。每一个著名的复杂度测量方法都是有各自的优势，而且已经在识别包含很多缺陷或难以维护的代码区域这一方面取得了一定的成功。复杂度度量也有可能误诊。误诊指的是通过分析显示出代码具有高复杂度，但是却几乎没有缺陷而且易于维护。一种减少误诊的方法就是同时检视几种不同的复杂度度量，通过结合和权衡这些数据来减少误诊。如果几种不同的复杂度度量全都认为这个函数、模块或文件是高度复杂的，那这段代码就极有可能包含缺陷或者是难于维护。另一方面，如果一个度量显示这段代码是

高复杂度的，但是其它度量不这么认为，那么这段代码可能就不太复杂，也不会太难以维护或包含缺陷。

对于复杂代码该怎么办呢？如果是新代码，高复杂度意味着这段代码需要重新分解。微软的一些团队曾经尝试过给新功能设定回路复杂度限制。（大多数做过试验的小组发现，回路复杂度本身并不总能准确地说明代码需要重新分解。）而其它组（包括 Windows 组）则将几种不同的复杂度度量组合起来，使用组合后的数据来决定产品周期后期改动组件的风险。

例如表 7-3 中列出的函数。函数 `OpenAccount` 具有很高的回路复杂度但是几乎没有调入函数（扇入），代码行数也很少。这个函数可能需要通过额外的修改来减少回路复杂度，也可能经过审阅后证明这段代码易于维护，代码的所有分支也易于测试。函数 `CloseAccount` 有很多个调入函数，但是几乎没有分支。这个函数可能需要更多的审阅来减少调入函数的个数，但是这个函数应该不会有太多风险。在这三个函数中，`UpdatePassword` 的风险可能最大。回路复杂度虽然中等，但是它有将近 20 个调入函数，而且这个函数的长度几乎是其他函数的三倍。当然对于这三个函数中的改动，你都需要测试。但是函数 `UpdatePassword` 的复杂度度量显示了你需要更为关注对它的任何改动。

表 7-3 复杂度示例

函数名	回路复杂度	调入函数数量	代码行数
<code>OpenAccount</code>	21	3	42
<code>CloseAccount</code>	9	24	35
<code>UpdatePassword</code>	17	18	113

微软 Visual Studio 工具集的最新版本包含了多个复杂度度量的测量方法，包括回路复杂度、Halstead 度量、（在图 7-3 中表示为 maintainability）和代码行数的度量方法。

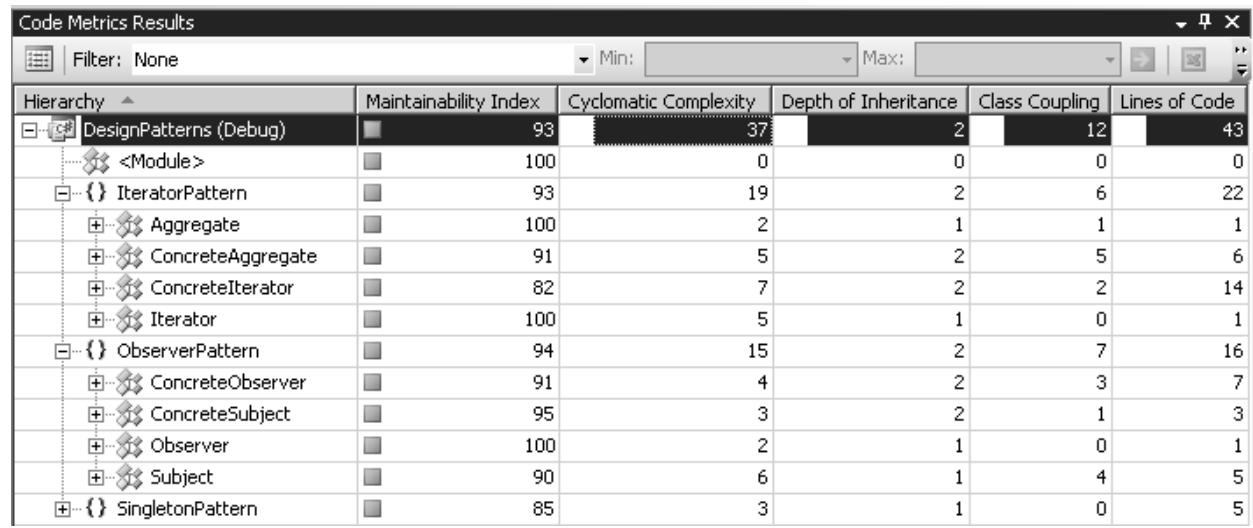


图 7-2 在 Visual Studio 2008 的复杂度度量

本章小结

代码复杂度是识别应用程序中可能存在缺陷的一种基本度量，对于识别代码维护性的问题也具有同等价值。如果你不需要度量代码所有部分的复杂度，那么就先从度量最关键的函数或功能开始。慢慢地逐渐扩大复杂度度量的使用范围，并且开始比较组件间或功能领域间的复杂度。

复杂度也是容易误用的度量，所以重要的是如何聪明地使用这个度量并且监视度量值的变化以保证该度量所指示的情况正是你所期望它能发现的。记住：高的复杂度只能告诉你这段代码可能会有很多缺陷，你仍然需要额外更多的调查来证实这个结果。

第 8 章 基于模型的测试

阿伦·培智

当波音设计新一代的民用客机如 787 梦想号时，他们使用软件为飞机建立模型。他们在软件模型上运行上百万次的模拟测试，来理解飞机的机身形状，零部件的重量，驾驶舱的位置，以及其他各种变量会怎样影响飞机的提升性能和油效。

我第一次学开手动车，感觉真是一塌糊涂。不知为什么，我就是掌握不好换挡和踩油门之间的时间差。最终，有人给我画了一张模型图，告诉我换挡和踩油门时的机械原理。奇妙的是，一旦我能在大脑中描绘出整个系统，我就再也没有问题开手动车了。我觉得整个系统都很对路。

模型帮助我理解了复杂系统是怎样工作的。在学校里，我通过模型学会了基础数学。我有一个老师，用图 8-1 所示的模型来解释“ $9 - 4 = 5$ ”，而不是直接告诉我们答案。

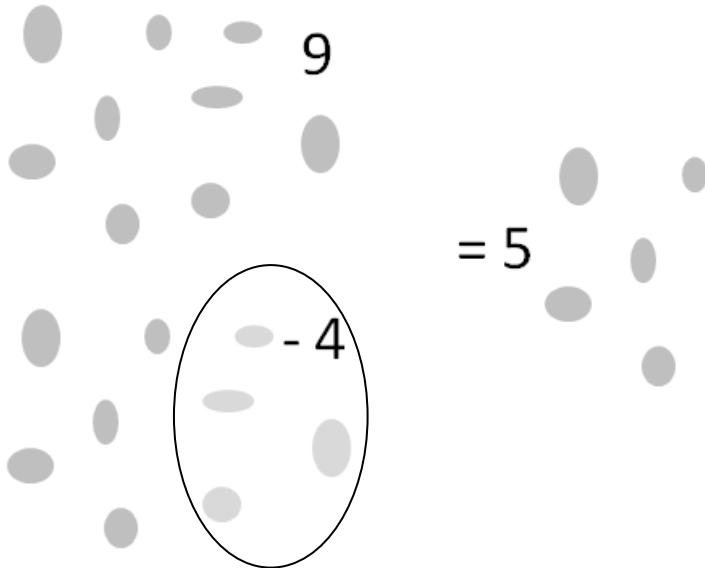


图 8-1 数学模型

如果你开始有 9 个圆球（物件），在去掉 4 个后，你就只剩下 5 个。用上面的图型来向六岁的孩子解释抽象的减法，是那么简单和有效。在我们一生中，模型帮助我们理解我们没有完全明白的事物。它可以将抽象的事物变得具体，回答类似于如果就怎么样的问题，以及帮助设计新的事物。采用模型迫使我们将复杂的事物分割成许多清晰，易理解的单元。最终让我们更好的理解整个问题。

基于模型的测试（MBT - Model-based testing）是一种（很自然的）方法可以解决长久困扰我们的问题。这一章节概括了微软测试工程师是如何将在脑海中的模型转变成可操作形式的基本流程。这个形式有可能只是办公室里白板上的一幅设计图，或是信封背后的一张草图。那些更认真的建模者，则会使用类似于微软 Visio 的图形工具或者一些特制的制模工具来建立他们的模型。当然测试工程师需要的不只是模型，他们还需要测试用例。测试工程师喜欢的制模工具当然是能生成测试用例的那种。

建模的基本要点

图 8-2 是微软测试工程师使用的一个模型，而这个模型本身是关于他们如何建立和使用模型的。测试工程师常常会在午餐中使用的纸巾上，在办公室的白板上，或者是在规格说明文档的背后写下模型。他们甚至有可能在 Visio 或是其它内部使用的帮助测试工程师建立基于模型的测试的工具上直接创立模型。

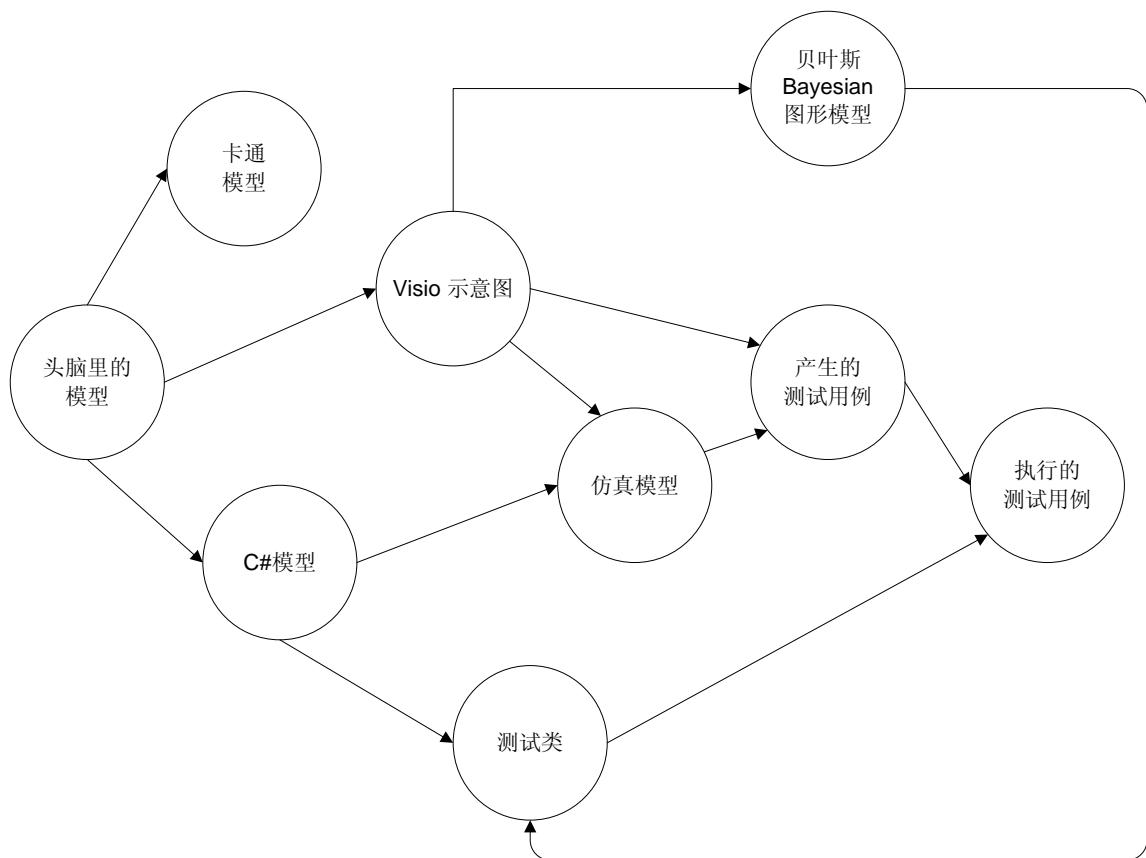


图 8-2 建模的模型

一个模型可以是对系统的任意描述。一个典型的行为模型中会有一个开始状态，一个或多个转变，以及一个结束状态。在图 8-1 中，开始状态是 9，中间过程是减去 4，结束状态就是 5。

FSM (Finite State Machine) 有限状态机是一个专有名词用来描述一系列的状态和中间的变化过程。它很自然地表达了由状态和中间过程所代表的功能。图 8-3 是为数学表达式“9—4”所做的 FSM。



图 8-3 表示 9-4 的有限状态机

采用模型测试

测试可以而且常常通过模型来进行。其实很多测试工程师在不知不觉中已经用到了模型。我本人就曾经看到很多测试工程师在办公室白板上描绘出他们所需测试的基本功能。然后通过跟踪程序的流程来理解这个功能究竟是怎么工作的。我觉得那些测试工程师如果能学习了解并且有意识地利用模型来测试的话，他们测试效果会更好。

设计模型

建立模型其实并不太难。难的是知道什么时候应该停止建立模型。我的第一辆车是 1962 年的 AMC Rambler。当年开那辆车我感觉非常愉快，操作也不难。当时那辆车用几个按钮来换档，而不是大家常见的排档。其中有行驶按钮，一挡，二挡和倒车按钮。另外一个既可作启动也可作空档的按钮。当我坐进车内准备去什么地方的时候，车子总是处于同样的状态 - 停止且熄火。在我坐定之后，我都会按绿色按钮，让它转变到一个新的状态一起动。之后，我通常会让我的车跑起来(这又是另一个状态)带我去学校。有时候我会忘带作业，书本，或者其它一些我该带走的东西。那我就不得不停好我的车(车又回到了先前的静止状态)，跑回房子里。在行驶过程中，如有必要我可以换挡 (不同状态)。从技术上讲，在任何时候我都可以将车切换到倒车档，但我从来没试过。为简单起见，我们姑且认为我们必须先换到空挡，然后才能换到倒车档，如图 8-4 所示。

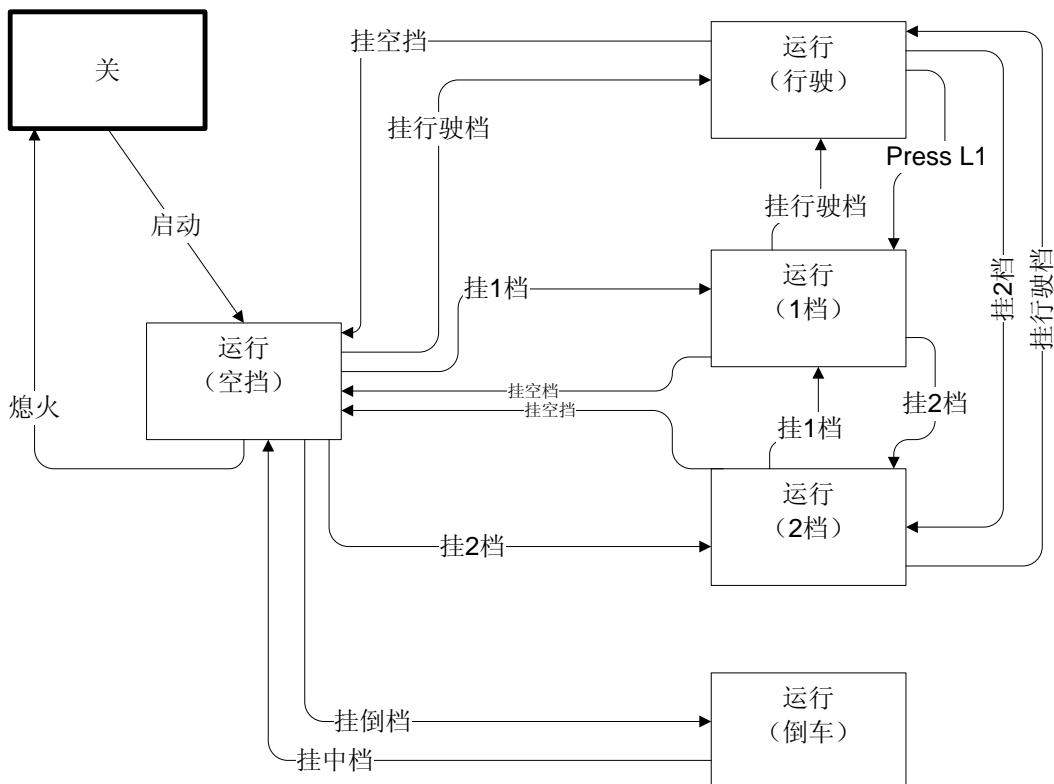


图 8-4 我的 Rambler 模型

这个模型（就像我的 Rambler）非常简单。问题是对我自己的车还可以进行很多其它的操作。图 8-4 中的模型并没有把刹车或是油门的操作也考虑进去，而这些也同样是可以模型化的。车窗有不同的开启状态，车灯有三种状态（关，开，大灯）。空调系统，雨刷器和收音机都有不同的模型，有些还会跟其他互动。模型可能会很快地增长，就像我的模型一样，我总会想起多年前听到的一个建议：“一个好的模型的规模总是‘很小’的。”由一些小的模型开始，可以在弄清不同系统间的互动之前完整地理解系统中的一个分区。当我刚开始学习驾驶的时候，我学习了有关车子各部分的基础知识（如何开动，如何转弯，还有如何停车）。在我有了些经验，又对车子的各个子系统进行学习以后，我开车开得更好了。我从来没有为我的 Rambler 画出模型，但是我确实在某种程度上把这个系统中的不同部分想象成了模型。

模型化软件

有很多软件是基于状态的，而且能从基于状态的测试方法中获益。图 8-5 显示了一个有三个按钮的应用程序。第一个按钮打印“Hello”到一个文本框中，第二个按钮打印“World”到一个文本框中，而第三个按钮则清除所有的域。当应用程序开始运行时，无论两个文本框的状态在应用程序终止的时候是怎样的，它们都是空的。



图 8-5 简单的应用程序模型

在应用程序中，这个是你能找到的最小最不复杂的。测试工程师第一件要做的事可能就是写出简单的序列化的测试用例，如图 8-6 所示。

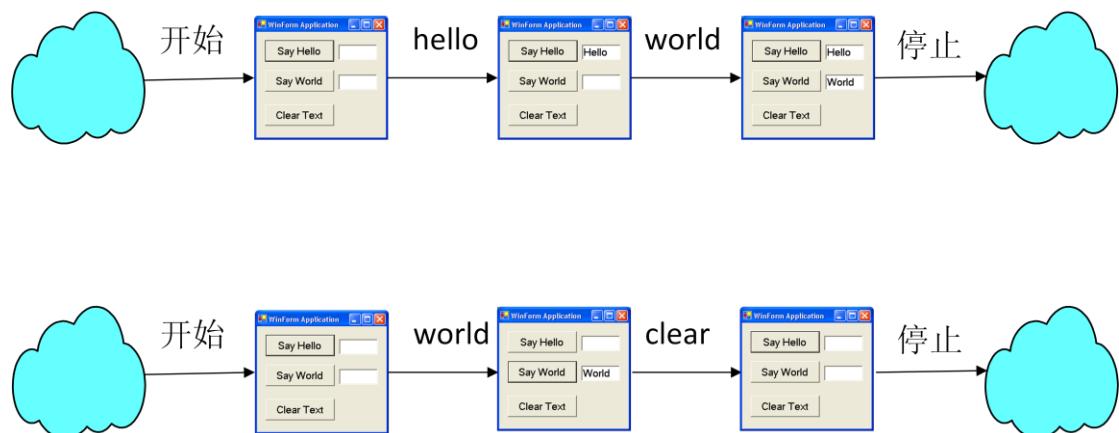


图 8-6 简单的应用程序模型

第一眼看你可能会认为只需要测试三个操作，一个按钮对应一个操作。有经验的测试工程师可能会想，“如果我连续两次按下“Say Hello”按钮会发生什么，或者连续按五十次呢？”

（好的测试工程师都会明白不管连续按一个按钮五十次听起来多么荒谬，在某个地方某个用户就会这么做。）对这个应用程序的脚本化的测试看起来就象表 8-1 所列出来的一样。

测试 1	测试 2	测试 3	测试 4
开始	开始	开始	开始
Hello	Hello	清除	World
清除	World	World	停止
Hello	Hello	清除	
Hello	World	World	
停止	清除	World	
	停止	Hello	
		停止	

表 8-1 测试实例

对于这个应用程序，这些测试用例也许足够了。但是，这样的测试也有一些缺点。首先，这些测试用例需要手动的维护。如果一个测试工程师忘记增加一个新的测试用例（比如，测试工程师意识到他需要增加一个测试用例，不点击任何按钮就关闭应用程序），这个测试工程师需要更新测试脚本。更重要的是，这些测试是静态的，它们总是在测试同样的东西。尽管通常这是个好的（和想要的）属性（任何在功能上的回归错误都能很快地被发现），但是很多脚本化的测试用例在创建和执行一次之后什么缺陷都没找到。从某种意义上说，是测试在训练软件成功地通过测试用例。图 8-7 显示了这个应用程序的状态模型，粗线显示了在表 8-1 中的模型化测试所“访问”的状态和状态的转换。

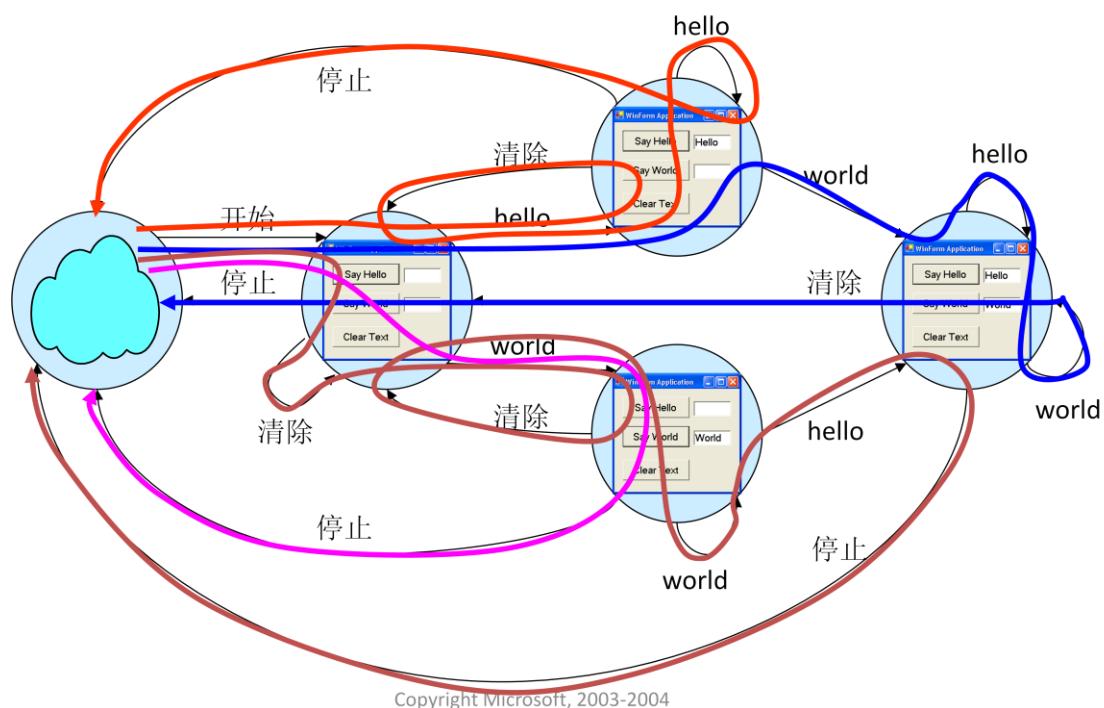


图 8-7 静态测试的模型的执行

状态模型可以澄清对应用程序如何工作和不同测试组合的理解。图 8-8 包含了这个应用程序的状态模型。4 个方框代表应用程序 4 种可能的状态，连线代表了基于各种状态可以进行的操作。这些状态如下：

1. 2 个文本框都是空的 (S1)。
2. 第一个文本框里面是 “Hello”，第二个文本框是空的 (S2)。
3. 第一个文本框是空的，第二个文本框里面是 “World” (S3)。

4. 第一个文本框里面是“Hello”，第二个文本框里面是“World”（S4）。

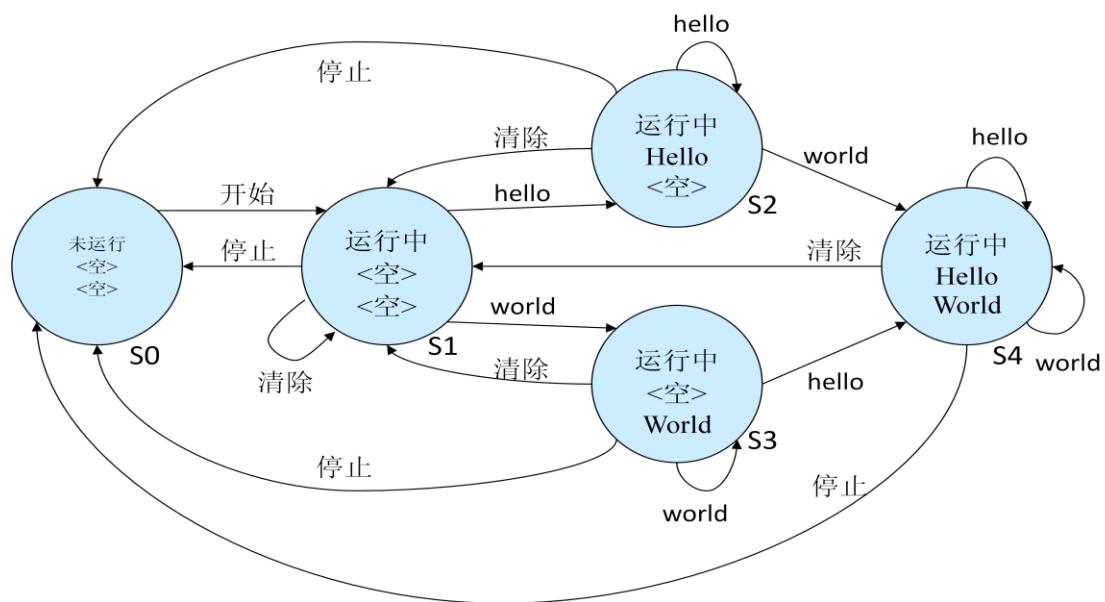


图 8-8 基于状态的模型（FSM）

那些连接线代表了造成状态转换的操作。例如，从开始状态（两个文本框全空），有三个可能的操作。单击清空按钮使应用程序回到相同的状态（或者从你的角度来看什么都不做）。单击“Say Hello”或者“Say World”将应用程序改变到一个新的状态。

对状态转换进行随机行走，期望的状态可能会如表 8-2 中所示。

测试用例	操作	文本框 1	文本框 2
0	启动应用程序	空	空
1	Say World	空	World
2	清除	空	空
3	Say Hello	Hello	空
4	Say Hello	Hello	空
5	空	空	空
6	Say World	空	World
7	Say Hello	Hello	World
8	Say World	Hello	World
9	清除	空	空
10	停止应用程序	空	空

表 8-2 对基于状态模型的随机行走

建立有限状态模型

构建模型对于多数测试工程师来说在开始时都会有点难，但通过不断实践，会逐渐成为一种下意识的形为。我在研究应用程序或规格说明书并绘制模型时，会发现自己在每次生成模型时都会要考虑下面这三个问题：

1. **我在哪里？** 我需要知道现在应用程序处于何种状态，而且我需要能够描述（或知道如何验证）当前状态。
2. **我可以进行什么操作？** 根据我当前的状态，我可以做哪些不同的事情？
3. **我做这些事的结果会怎样？** 如果我进行某项操作，它会将我带入怎样的状态？

随着模型的成长，我现在工作时继续考虑场景和期望值是很有用的。因为模型会让我提早看到所有状态会发生怎样的交互作用，我经常会在实际运行任何测试之前就发现缺陷。

模型自动化

对于基于状态的模型的自动化途径，它与传统的自动化途径稍有不同。模型的测试自动化并不是对端到端场景（end-to-end scenario）进行自动化，而是将重点放在过渡自动化和状态验证上。

图 8-8 中建模的应用程序包含四种可能的状态，和三种可能的操作：“Say Hello”、“Say World” 和 “Clear Text”。每种操作在所有四种状态中都有效。

为了给这个应用程序创建基于模型的自动化，我要实现几项功能，三个按钮（操作）每项，还要再实现四项其它功能来验证每种状态。

此时，我可以执行任何可用操作，而神奇地确保应用程序处于期望的状态。**MBT** 的工具和框架（本章稍后介绍）通过使用大量不同路径来穷尽所有的变换和状态，把此测试带到一个新的高度。

图论和 MBT

1737 年，莱昂哈德·欧拉（Leonhard Euler）解决了“哥尼斯堡七桥”问题。哥尼斯堡城座落于一条河中，通过七座桥与大陆和两个岛屿相连。显然，当时大家普遍的兴趣是想知道能否找到这样一条路径，它通过每座桥一次，且仅有一次。

欧拉明确了这个任务是不可能完成的。在证明过程中，他定义了节点（即此例中的陆地）和边（桥）之间的关系，并创建了定理来演示可能完成该任务的条件（使用不同数量的桥或连接）。欧拉的解决方案被认为是图论的第一个定理。

在数学上，图形是一系列的边（或链接）和节点。在 **MBT** 中，边和节点分别代表转移和状态。研究后发现图形遍历背后的数学对于测试状态模型将会是非常有趣的。**MBT** 的主要功效体现在遍历算法中。随机地通过每种状态的测试是有趣的，并且经常会发现 缺陷，但将图论的概念应用于遍历会有力而且有效。

随机行走遍历会随机选择一个可用的转换。它没有指导或计划；它只是在您设定的时间内尽可能地走遍各个状态。随机行走通常会发现 缺陷（有人将这些视为“聪明猴子测试 – smart monkey test”），但可能会花特别长的时间来遍历大的模型。

加权遍历是稍好一点的解决方案。加权遍历在某种程度上是有指导的随机行走。选择哪个转换仍然是随机的，但最有可能的选择都进行了加权，以便它们的发生更频繁。

最短路径遍历使用最少量的转换走过两个节点间的路径。

使用图论算法还有许多其他方法可以遍历状态模型。“所有转换路径”，正如其名称所示，确保所有转换路径都被尝试过。请注意，所有转换路径也包括所有节点。相反，“所有状态遍历”则不保证所有转换路径都被尝试过。通过重新访问图 8-8 提供的模型，图 8-9 展示了

经历所有状态的遍历的示例，图 8-10 展示了到达所有转换的遍历。

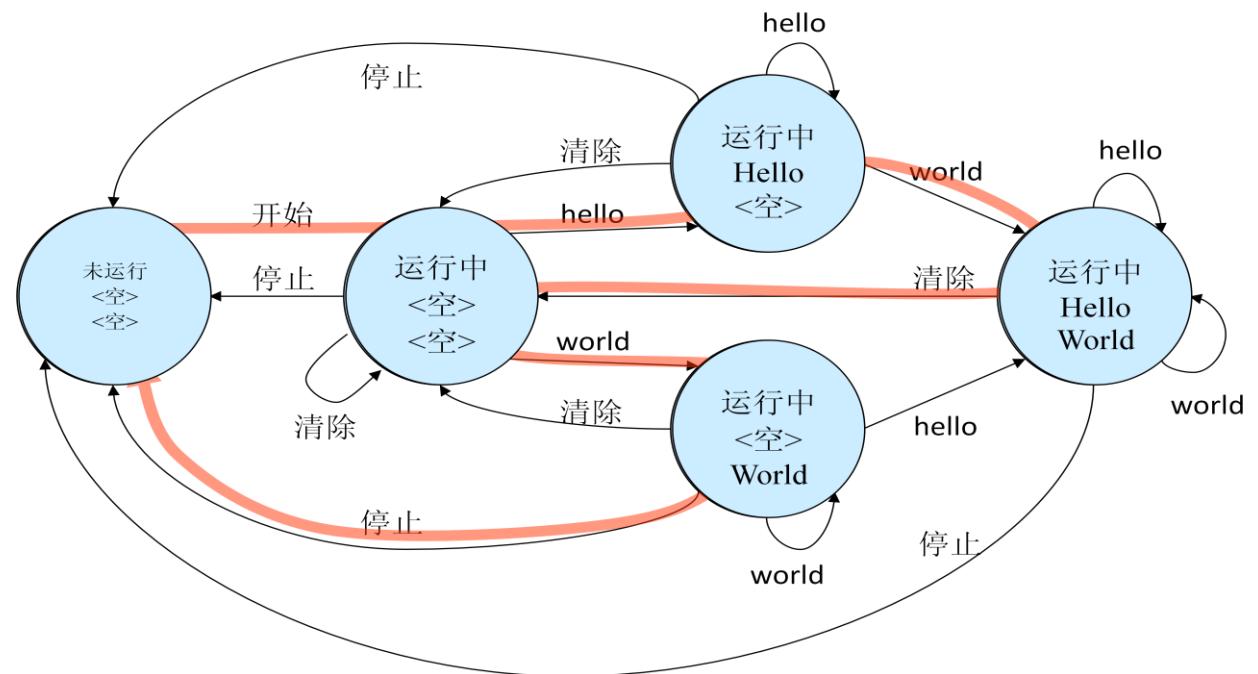


图 8-9 所有状态遍历

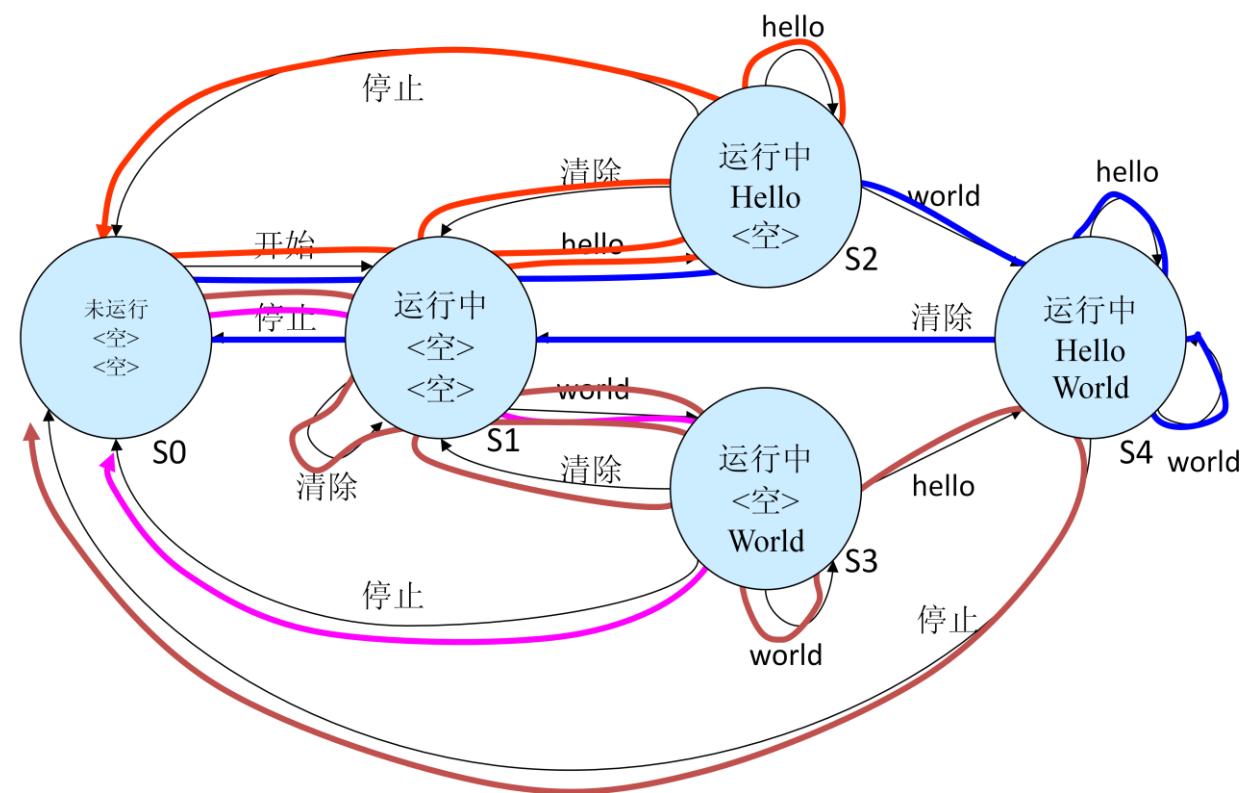


图 8-10 所有转换遍历

用于 API 测试的模型

在较低级别，许多操作系统或平台功能也可以由基于状态的方式方法来测试。常用的文件读写功能就是一个这样的例子。常用的文件 API 可通过创建文件、打开文件、修改文件和关闭文件来更改文件系统状态。

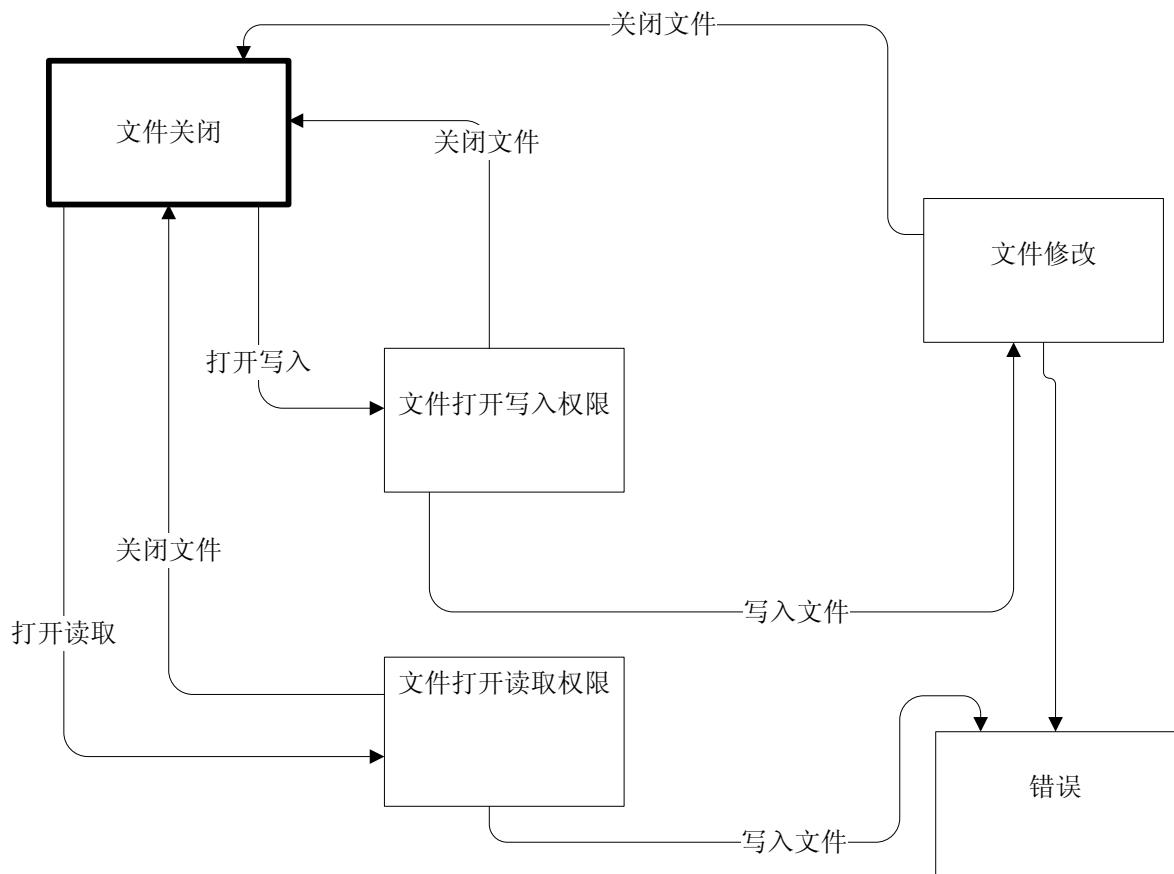


图 8-11 文件 API 的局部模型

图 8-11 中显示的是一个简单模型，并提供了测试整套文件功能的起始点。随着该模型的成长，我可以区别对待打开文件、新建文件或试图打开一个已打开的文件。之后，我将会增加从文件中读取、多次写入同一文件以及关闭以前关闭过的文件等操作。其他相关的文件功能，如在文件内设置读取器的位置，也可以包括在此模型中。模型可以快速成长，所以开始时可以小一些，然后随着您对要建模的功能的逐渐了解而成长。这一点是很重要的。

随机模型

基于模型的另一简单形式是猴子测试。猴子测试³⁴会生成随机（或假随机）输入，以期找到 缺陷。猴子测试在压力测试中很通用，而且通常也不难创建。几年前，我写过一个特别简单的，它看上去有点像列表 8-1 中的伪代码。

```
启动文本编辑器

// 模拟敲键
重复1000次
{
    重复500次
    {
        任意敲一个键
    }
    选择一部分文本
    随机选择字体、字体大小、和字体颜色Select Random Font, Font Size, and Font Color
    按工具条按钮中的一个[Cut剪切|Copy拷贝|粘贴| ]
}
关闭文本编辑器
```

列表 8-1 猴子测试示例

此测试看着是蛮有趣的，但它所发现的唯一缺陷是轻微的内存泄漏（这在嵌入式系统上是严重 缺陷）。猴子测试的缺点是这种测试一般难于进行调试。为了找到导致内存泄漏的确切命令组合，我需要再编写一些其他测试。在此例中，它是“剪贴板”功能中的泄漏，所以针对剪切、复制和粘贴的有目标的测试会更早地发现该 缺陷。

语法模型

MBT 的另一常用形式是使用语法模型进行测试。语法模型描述数据的特征和结构，例如 电子邮件地址、名称或文本等。

例如，正则表达式就是一种对语法模型的实现。正则表达式的一个常见用途是搜索文本。如果我想在一大批文档中找出所有出现我名字 (*Alan*) 的地方，我可以使用搜索工具来查找文本中出现的 *Alan*。不过，如果我想同时搜索其他的拼法如*Allen*和*Allan*，则可建立一个使用正则表达式的模型来同时搜索所有三种情形。例如，搜索 *Alan|Allan|Allen*（模型）会找出所有出现我名字三种拼写的地方。利用正则表达式模型语言，我还可以将该模型简化（不过有人会有不同意见）为 $(A|Al)|(a|e)n$ 。³⁵ 列表 8-2 中是前一模式的模型

³⁴ 猴子测试的概念来自无穷猴子定理，定理说一千只猴子和一千台打字机最终能打出威廉·莎士比亚的全套著作。

³⁵ 注意这个正则表达式还能找到 *Alen*。这很容易修正，但不是我们本书要讨论的。

```
<Name>      ::= Alan | Allan | Allen
```

列表 8-2 正则表达式模型

语法模型对于创建测试数据而言是很有用的。例如，您要测试 Microsoft Windows Live Hotmail，希望发送包含多种不同输入的 10,000 封电子邮件（假定您有充足的测试帐户和足够的自动化来完成这项测试）。您可以做的第一件事就是创建您关注的邮件数据的模型。该模型看起来可能会是像列表 8-3 那样，并且可以生成表 8-3 中的数据：

```
<收件人>      ::= <有效地址 | 无效地址>
<抄送>      ::= <有效地址 | 无效地址>
<隐藏抄送>    ::= <有效地址 | 无效地址>
<主题>      ::= <空 | 不空>
<正文>      ::= <空 | 不空>
```

列表 8-3 电子邮件字段的语法模型

有效地址	无效地址	空	不空
testacc1@hotmail.com	<空>	长度 == 0	长度 > 0
testacc2@hotmail.com	nobodythere1@hotmail.com		
testacc3@hotmail.com	nobodythere2@hotmail.com		
...	...		

测试生成器可以在使用模型时结合包含有效和无效电子邮件地址、随机主题行和邮件正文的数据库，以生成 10,000 封各不相同的电子邮件。另外请注意，另一名测试工程师可轻松创建另一语法模型来为主题和邮件正文创建随机字符串。

建模的成功例子

我们的测试团队对 Windows Vista 操作系统研发时间段中的几项新功能使用了基于模型的测试，并且发现可以通过少量工作实现非常广泛的（测试）覆盖。最困难的问题之一是创建模型来描述我们所讨论的系统。一旦通过正确定义所有可能的输入、转换和输出，我们就会发现准确地定义从任何可能的参数组中得出的结果是很容易的。该模型本质上会变成您要测试的系统的理想表现形式，并且任何偏离都显而易见。

我们发现基于模型的测试非常强壮，因为我们可以应用它来测试从 API 到 UI 的各种领域。这样，当模型实现时，我们就可以完全取消手动测试。尤其是对于我们的 UI，这样就节省了我们大量的手动验证 UI 功能对于所有可能出现的操作（单击按钮、输入文本，等等）的时间。经过适当调整，我们的基于模型的测试可以提供从快速验证测试到全面功能测试的覆盖。这是通过将我们的输入、转换和输出列表删减到我们感兴趣的子集来实现的。如果我们依赖于逐个开发特定的测试案例，那么实现这样广泛的覆盖面就会非常困难。

我们实现了许多成功的建模，其中之一涉及了一个新 API 的测试。该 API 是在 Windows Vista 中发布给开发人员的。我们没有编写使用该 API 的测试应用程序，也没有将一系列可能场景写成程序，而是使用了建模来为开发人员可以做出的所有可能的函数调用生成状态机。这就意味着，对于某个给定的函数和指定参数的调用，我们可以确定所有其他后续函数调用会有怎样的反应。我们不用为每种有趣场景都编写一次性应用程序，而是通过测试框架就可以测试 API 是否对不同选项/参数/完成架构做出了正确反应。这才是我们测试框架的真正价值。

通过使用模型，我们的测试足迹小了很多，而同时却获得了与旧途径相比更大的覆盖面。

—Jim Liu, Windows Networking 软件测试工程师

不带测试的建模

尽管同建模一样令人兴奋，基于模型的测试和建模并不是同一件事情。然而许多测试工程师都不能将它们区分起来。一个模型对设计或开发组来说可能是一个很强大的工具，即使它并不能生成测试用例。在这个小节，我们来看两个不直接影响测试用例的建模的例子。一个例子显示如何对不确定性或风险进行建模。另一个例子为测试工程师提供一种方法来保证他们的模型像他们所期望的那样工作。

贝叶斯图解模型 (Bayesian Graphical Modeling)

贝叶斯图形模型 (BGM)³⁶和微软使用的其它建模方法³⁷有很显著的不同。BGM 分析的目标是为了衡量和减少测试的不确定性。它是对风险分析的一种建模方法。每个程序员都对他要测试的应用程序的代码有一些程度的信心。很多因素形成了这种信心 (开发员的声誉, 代码的复杂度, 依赖的二进制文件的数量, 测试时间和成本, 以及其它因素)。如果每个测试工程师以及每个应用程序都有同样的假设, 那么我们测试及减少不确定性就有了一个好的起点。

如果一个测试工程师认为要测试的这段代码的质量是好的(特别是在有很强的内部开发控制的情况下), 所有测试工程师最好以回答这个问题为起点: 我有多少信心认为这段代码没有缺陷? 在 BGM 中, 测试工程师表明他对要测试的这些组件的信心, 并且详细描述建模的原因。BGM 将所依赖的组件的不确定性和风险也考虑在内。然后测试工程师运行测试, 当发现缺陷的时候, 他们就去更新 BGM。这个模型和测试在产品周期中形成一个更新反馈回路。BGM 的例子如图 8-12 所示。

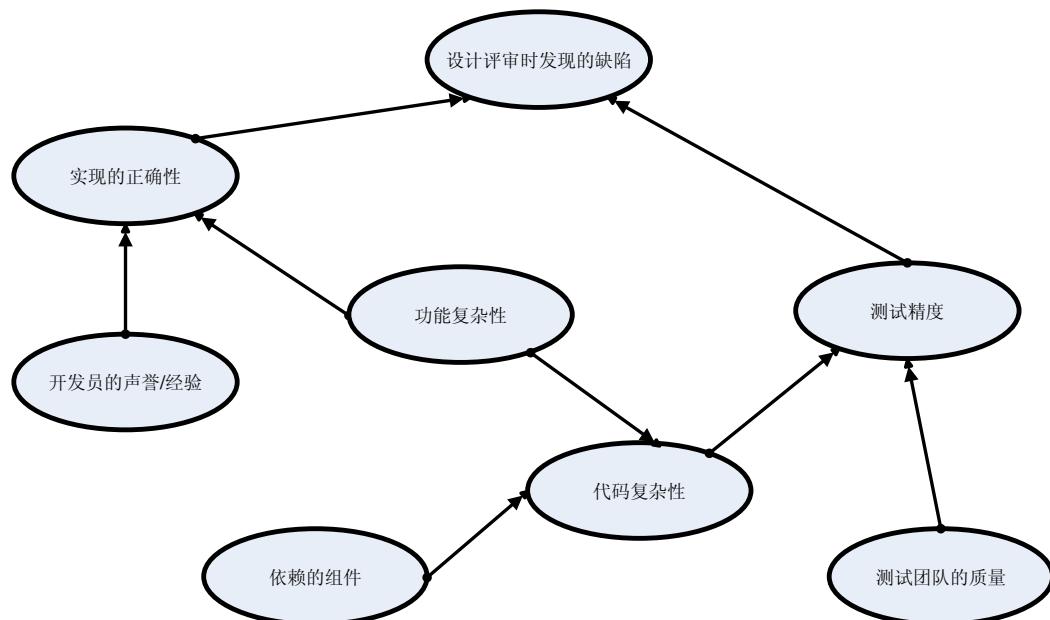


Figure 8-12 应用贝叶斯图解模型评估产品质量

我们测试工程师从来不能做到对我们的工作完全确定, 但我们可以衡量并且采取措施减少任务中的不确定性。在我们必须发布产品的时候, 能知道我们成功的概率毕竟是一件很好的事情。

Petri 网

Petri 网是一个建模工具, 它由库所、转移 (transition) 和连接库所和转移的有向弧三种元素

³⁶ 又被叫做贝叶斯信念网络

³⁷ Heckerman, David, 贝叶斯网络的学习教程, 技术报告 MS-TR-95-06(Redmond, WA:微软公司, 1995 年三月)。

组成。库所也可以包括令牌；每个库所包含的令牌的数量表明模型系统当前的状态。转移用来表示可能发生的活动（也就是被转移所触发的），从而改变系统状态。转移只能在激活状态(*enabled*)下发生，也就是所有前提条件都满足的情况下。当转移发生时，输入库所的令牌被消耗，同时为输出库所产生令牌。使用 Petri 网有点像在和计算机玩棋盘游戏。玩家/测试工程师通过移动令牌，观察系统状态，获得对系统更好的理解。

Petri 网为与软件行为交互提供了一个视觉的方法(加上 Petri 网工具，可以提供交互的方法)。Petri 网可用于多种目的，但主要是针对并发和资源共享的系统来建模。

建模胜利

在很多情况下，即使只是创建模型这个行为都是有影响的。当我在微软教测试课程的时候，有一次我们在讨论建模，学生们在画他们所工作的系统模型。练习开始大概 15 分钟后，一个学生站起来很大声的说：“我刚为我们即将要发布的产品画了个模型，意识到我们以前忽视了的一个场景。我刚测试并发现了一个缺陷！”在过去我一直使用模型作为我理解规格说明的一种方式。一个图有时候确实胜过一千个文字。

微软的基于模型的测试工具

在我看来，微软的测试工程师在基于模型测试方面犯下的最大的错误，大概就是尚未理解建模基础就急于钻研基于模型的测试工具。建模工具通常与测试工程师所熟悉的程序设计语言相类似。但是对很多测试工程师来说，在学习新的工具的同时，要完成将测试作为模型来考虑的思维转变仍然是比较难的。在很多情况下，我的职业生涯中也有许多次，我仅仅首先创建了模型，就获得了成功。画一张我们测试的软件是如何工作的图是十分强大的工具，其价值是不可低估的。

当模型的复杂度、遍历的不同变化、建模对象的范围增长到一定程度时，大多数建模者都需要一定的帮助。在基于模型的测试中，工具的出现恰恰起到了这种帮助。

微软内部最早的基于模型工具大约是在 2001 年左右出现的。产品开发团队和微软研究院都曾参与为基于模型的测试创建工具，其中一些工具得到了广泛的使用。各种各样的工具为不同类型的测试工程师/测试问题提供了解决方案。从技术领域的各个不同角度迎接建模的挑战，使得更多的团队能成功的应用建模。

Spec Explorer

微软使用的一个强大的基于模型的测试工具是微软研究院在 2002 年创建的 Spec Explorer，从 2003 年起这个工具就在微软的产品组中使用了。Hypervisor 团队使用 Spec Explorer 创建了他们的产品的完全模型，开发了 10000 行左右的模型代码，成功的用于测试他们的核心技术。（要想更多地了解 Hypervisor，参见第 15 章“今天解决明天的问题”。）近年来，微软有超过 20 个不同产品组使用 Spec Explorer。

基于模型测试工具家族中的第三代产品 Spec Explorer for Visual Studio 2008，已经从微软研究院移交给微软的服务器和工具事业部。正是在这里，它在其他领域中被用于大量的微软互操作性协议的测试。Spec Explorer for Visual Studio 有一些特性。例如，它可作为微软开发环境的插件运行，并且允许用 Rich 模型状态以 C# 语言创建模型。该工具除了支持面向场景和面向状态两种建模外，还支持基于事件的测试和非确定论（实现有多种反应选择）。

关于 Spec Explorer 最好的消息大概就是它很快就会以 Visual Studio 的 Power tool 的形式面世，应该会先于本书与大家见面。在深入讨论 Spec Explorer 模型之前，让我们先来看一下它的整体设计。Spec Explorer 的目的是什么？能解决什么样的问题？如何解决？

用 Spec Explorer 建模

Spec Explorer 对可生成有限状态机的模型程序的分析和转变能力令人赞叹。使用有限状态机有一个问题，即它的复杂度增长的很快。Spec Explorer 通过仅列出那些能产生不同效果的状态来控制复杂度。

模型程序是一个基于模型测试高级应用程序最重要的部分。在模型程序中，特殊的建模属性通知运行时使用哪些方法来驱动模型逻辑。测试工程师也正是在模型程序中，描述合同以及使模型程序成为模型的合约规则，比如前置条件、后置条件和不变量。

模型程序完成以后，我们手中可能会得到一个非常复杂的状态空间。为了减少模型的安培数，我们需要一个单独的文件，来提取那些会真正驱动测试的特定场景、特定动作或一系列动作。确实，检验模型有效性的一个方法是检查一些已知场景是否在列。如果我们要为某个实现生成一致性测试，那么还需要将模型和它的测试用例与我们要测试的二进制文件绑定。

Spec Explorer for Visual Studio

图 8-13 显示的是 Spec Explorer 在 Microsoft Visual Studio 2008 环境中运行时的状况。

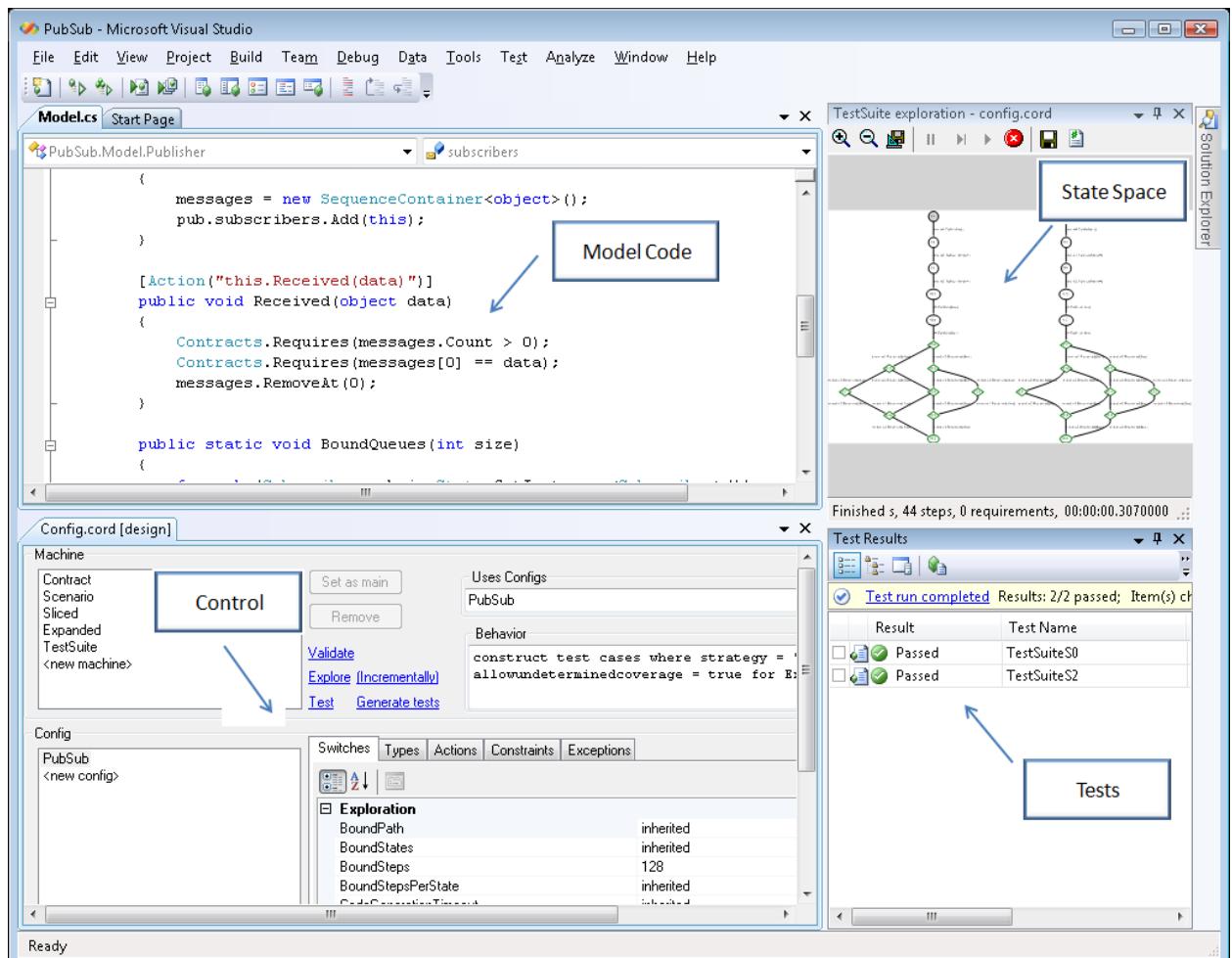


图 8-13 Spec Explorer 运行中

左上角的窗格中包含的是模型程序代码，以常见的 C#语言形式给出，但有一些自定义的属性。左下角的部分是 Spec Explorer 的控制中心，在这里你可以配置各种参数以及选择探索目标。右上角的窗格是探索结果的自动图形化表示。右下角的部分代表从模型生成的测试用例，它们由 Visual Studio 测试工具来管理和执行。

为了更好的了解如何使用 Spec Explorer 来做基于模型的测试，我们来考虑一个简单的秒表应用程序，有如下属性：

- 秒表有两种显示模式：
 - 当前时间
 - 停表
- 表有三个按钮：
 - 转换模式（总是可用）
 - 重置停表（仅在停表模式且停表在走时有效）
 - 开始/停止停表（仅在停表模式有效）

秒表上的每个按钮代表一个动作，我们可以向应用程序询问停表是否在走。为这个秒表程序建模的测试工程师可以创建如表 8-5 所示的动作和代码来描述秒表模型。

```
using System; using Microsoft.Modeling;
```

```

namespace Model
{
    static class Stopwatch
    {
        static bool modeTime = true;
        static bool stopperRunning = false;

        [Action]
        static void ModeButton()
        {
            modeTime = !modeTime;
        }

        [Action]
        static void StartStopButton()
        {
            Contracts.Requires(!modeTime);
            stopperRunning = !stopperRunning;
        }

        [Action]
        static void ResetButton()
        {
            Contracts.Requires(!modeTime);
            Contracts.Requires(stopperRunning);
            stopperRunning = false;
        }

        [Action]
        static bool IsStopping()
        {
            return stopperRunning;
        }
    }
}

```

表 8-5 秒表模型

表 8-5 中有两项内容，大多数读者可能不大熟悉。[Action]（动作）属性是来自 Spec Explorer 库，用于区分模型动作和普通 C#方法；而 Contracts（合同）类（特别是 Contracts.Requires 方法）是用于强制执行前置条件的。换一种说法，在从模型生成有限状态机时，Spec Explorer 会扫描模型的状态，并执行所有激活状态下的动作（Action）方法。

配置文件是使用 Spec Explorer 建模的核心。配置描述了一系列开关和参数，用于控制探索、状态图的显示、以及测试。图 8-14 显示了设计视图下的配置文件。

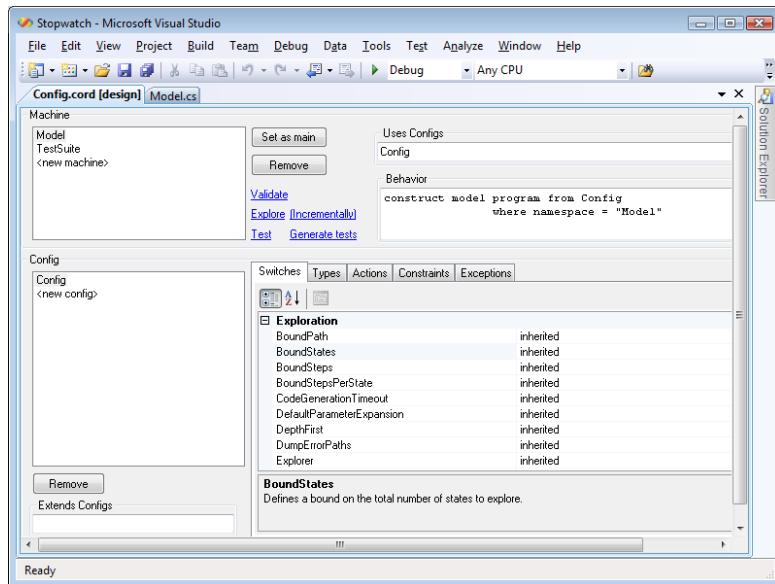


图 8-14 Spec Explorer 的配置

由此测试工程师可以创建模型视图，如图 8-15 所示，或生成遍历模型的测试代码。

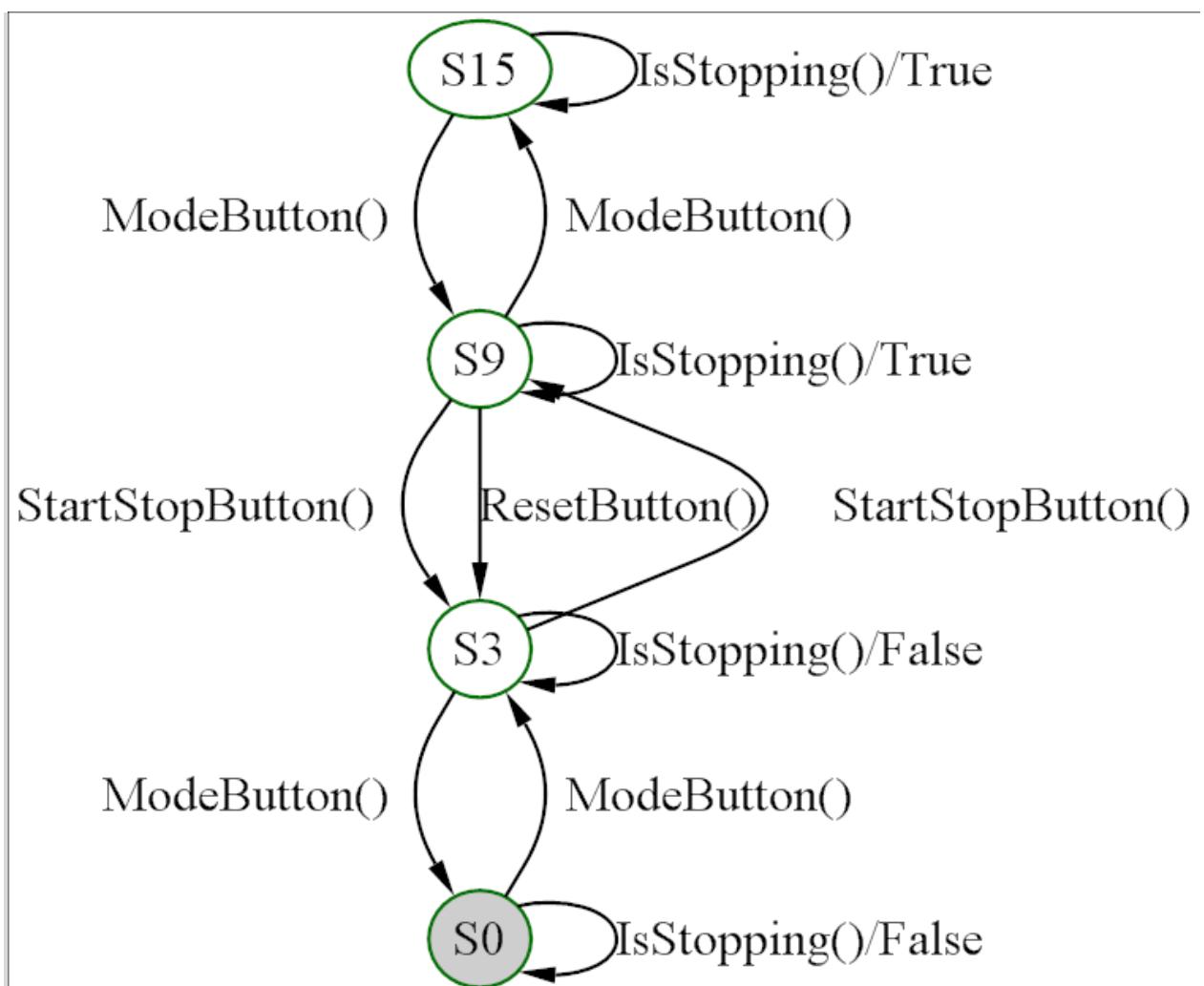


图 8-15 Spec Explorer 生成的秒表模型

测试工程师也可以直接探索和测试模型，或创建场景来从整体模型中抽取“片断”，或继续更改模型的其他众多排列组合。使用模型，测试工程师可以制定测试策略和为关键场景建模

(特别是观察模型的图形显示时这些会更加明显)。

在模型配置文件中，测试工程师可以为测试组指定策略。表 8-6 中显示了 Spec Explorer 策略的一个例子。

```
machine TestSuite() : Config
{
    当策略是"shorttests"时，为模型生成测试用例,
}
```

表 8-6 秒表的短测试策略

接下来测试工程师可以从模型生成（并且运行！）测试，或者继续向模型添加约束条件或其他配置细节。基于模型的测试和 Spec Explorer 确实是与多数测试工程师已习惯的测试方法十分不同，但是迄今为止在微软内已经取得了显著效果。图 8-16 所示的是使用 Spec Explorer 从秒表模型生成的测试的例子。

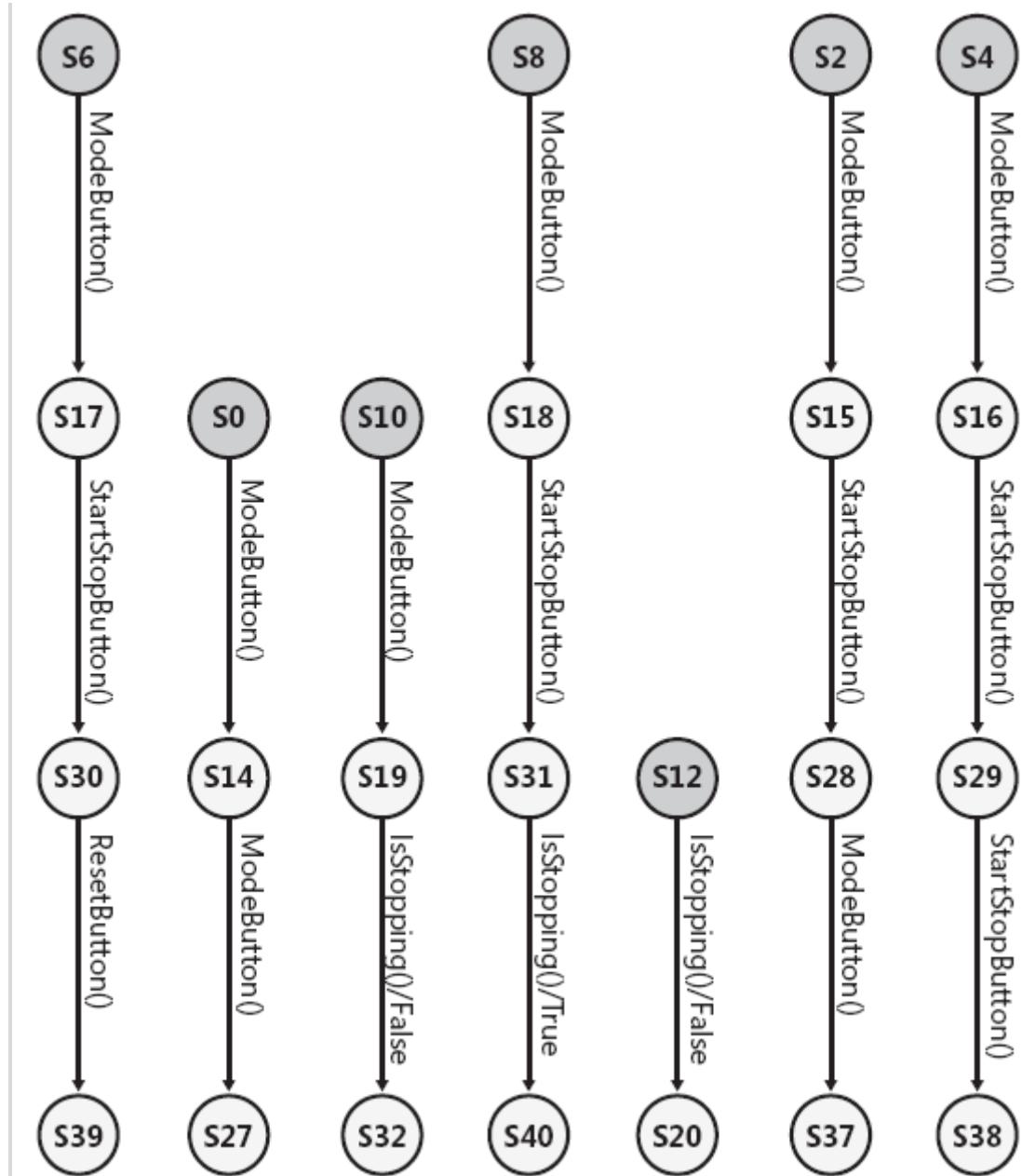


图 8-16 秒表模型生成的测试

语言和引擎

在微软中使用的另一个成功的 MBT 实现也是简单的利用语言的公共语言运行库 (CLR)，和一个在运行时执行整个模型图的引擎。与 Spec Explorer 不同之处在于这里的引擎也使用 CLR 来执行。

在这个简单但强大的解决方案的帮助下，测试工程师可以十分迅速地创建模型。与 Spec Explorer 相同，创建模型类似于大多数测试工程师都熟悉使用的面向对象编程。测试工程师可以同时在多个模型上工作，模型可以继承自其他模型，模型可以嵌套，并且模型可以调用其他模型。编写模型就像在用一种程序设计语言，所以对于熟悉编程的人来说十分自然。这里并不需要额外的编译器或者工具。测试工程师写代码时只需要用自定义的 C# 属性来定义

模型行为，然后运行执行引擎来解释这些注解就可以了。由于模型仅是建立在注解基础上的，所以模型和非模型测试之间可以实现无缝集成了。

使用这个特定建模工具的团队能够迅速的建立一个小规模的模型，然后根据需要通过与其他序列测试集成来扩展他们建模的范围。

表 8-7 显示了一个以 C#语言书写的使用属性来描述的模型，而表 8-8 显示的是运行该模型的一个测试的代码。

表 8-7 用 C#属性建模

```
[Model]
public class AppModel : Model
{
    [ModelVariable]
    public bool _running = false;
    public bool _hello = false;
    public bool _world = false;

    [ModelAction]
    [ModelRequirement(Variabe = "_running", value = false)]
    public void StartApp()
    {
        //start application
        _running = true;
    }

    [ModelAction]
    [ModelRequirement(_running = true)]
    public void SayHello()
    {
        buttonHello.Press();
        _hello = true;
    }

    [ModelAction]
    [ModelRequirement(_running = true)]
    public void SayWorld()
    {
        buttonWorld.Press();
        _world = true;
    }

    [ModelAction]
    [ModelRequirement(_running = true)]
    public void Clear()
    {
        buttonClear.Press();
    }
}
```

```
    _hello = false;
    _world = false;
}
}
```

表 8-8 运行模型

```
class ApplicationTest
{
    class ApplicationTest
    {
        static void Main()
        {
            ApplicationTest test = new ApplicationTest();
            test.MainTest();
        }

        public void MainTest()
        {
            ApplicationModel app = new ApplicationModel();
            ModelEngine engine = new ModelEngine(app);
            engine.Run();
        }
    }
}
```

Spec Explorer与Windows 7

在即将要发布的被称为 Windows 7 的 Windows 操作系统中有一个新功能：服务帐户。这个服务帐户功能解决了在 Active Directory 目录服务中为自动密码管理和简化的 Service Principal Name (SPN) 管理提供新的托管服务帐户类的问题。

该功能实现了一套丰富的 API 来管理这种新型帐户。此外，它支持与安全相关的系统功能，如帐户登录。API 的结果测试矩阵，调用顺序以及参数的排列是非常复杂的，所以如果使用传统的自动化测试，我们很容易漏掉测试用例。基于模型的测试一个很好的解决办法，因为帐户状态和 API 调用可以很容易地用一个有限状态机来表示。

可用于 MBT 的工具选择范围很广。我们选择 Spec Explorer 是由于它的一些功能，例如 C# 驱动的模型开发，与 Visual Studio 的集成，以及支持（使用案例）对模型切片。我们第一个模型是使用与 Spec Explorer 一起发布的范例模型来创建的。随后，模型随着新状态、转换和其他参数排列的增加而扩大。使用 Spec Explorer，您可以添加可执行代码来实现状态转换。我们的模型会生成一个测试用例的 XML 文件，之后可以为传统测试执行引擎所用。

应用 MBT 来测试服务帐户是非常成功的。使用 MBT 生成的测试用例使我们花更少的时间测试，却得到了更好的覆盖率。另外，模型十分容易扩展。它也帮助我们在开发的早期发现有趣的设计问题。

我们学到的一个重要的经验教训就是，作为一个好的建模实践，我们应该首先为最基本的功能建模，然后逐步扩展它；这使得修正建模错误容易得多。相对于传统测试设计，以下是一些我们在使用基于模型的测试中总结出的观点：

- MBT 为功能的设计提供了不同的视角。
- 测试是自动生成的，而且保证用最少的步骤达到对模型的完全覆盖。
- 扩展模型十分容易，而且可以利用以前生成的测试。

对我来说，使用 MBT 充满了乐趣，是一个很好的学习经历。模型很容易理解，使得我对我的测试覆盖程度很有信心。很多有趣的缺陷被发现，包括设计缺陷、输入验证缺陷和场景缺陷。在测试下一个功能时，我肯定还会选择使用 MBT。

——Sasha Hanganu，Windows Security 软件测试工程师

建模提示

我观察到许多测试团队都将基于模型的测试添加到他们测试技术的武器库中。我看到有的团队获得了成功，也看到有些团队无法成功地采用基于模型的测试。以下是一些我观察到的团队在尝试使用基于模型的测试时最常见的错误：

- **过多的建模：**有些团队尝试对所有东西模型，而最终却什么都没有测试。最重要的是牢记在刚刚开始建模时，从简单功能的小模型开始。请记住，并非所有情况都适合建模，且大型模型是很难维护的。

- **建模并不能代替其他的测试:** MBT 只是众多您可以用来做测试的工具之一。期待 MBT 无所不能的团队很快就会发现事实并非如此。开发一个充分利用测试工具箱中所有工具的测试策略将更加有效。
- **只为能验证的东西建模:** 随机猴子测试十分有趣，但通过这个方法发现的漏洞极难调试和诊断。优秀的模型包括每一个步骤的检查，确保当前的状态符合预期的状态。
- **仔细设计:** 当测试工程师在编写常见的自动化测试时，一两个错误只是造成一两个测试报告错误的结果。但是当他在建立一个模型时出错，则整个系列的测试都可能失败。所以模型的设计是十分关键的，需要格外的小心和审查，尤其是对正在学习建模的团队而言。

本章小结

任何类型的建模都是有益的，而随着模型而来的从模型产生的测试是强大的。模型能帮助测试工程师们理解（并解释）复杂的系统，帮助管理风险，和帮助寻找漏洞。

从模型生成的测试用例可以做一些有意思的事，一些人类测试工程师可能没有想到或没有耐心去做的事。模型意想不到的行为可能令应用程序生产意想不到的行为（对测试工程师而言，这是十分有益的）。例如，由 100 个随机生成的步骤组成的测试可能会也可能不会（但通常会）找到一个深藏在被测试的应用程序内的漏洞。如果您可以像产生这 100 个步骤的随机行走一样彻底，您可能也会发现这个漏洞。但是既然计算机在恰当的指示下就可以不分昼夜的随机探索测试应用程序，直到它找到崩溃为止，您又何必麻烦自己呢？

当然，基于模型的测试能做的不仅仅是在应用程序中随机行走。微软的团队曾经使用基于模型的测试与传统的测试自动化相结合，有效的测试了许多新功能和新应用程序。成功使用建模的团队也都明白建模只是测试工具箱中众多测试工具之一而已。

推荐资料和工具

- 用 C# 做基于模型的软件测试和分析（Model-Based Software Testing and Analysis with C#）， Jonathan Jacky， Margus Veanes， Colin Campbell 和 Wolfram Schulte
- 实用的基于模型的测试：使用工具的方法（Practical Model-Based Testing: A Tools Approach）， Mark Utting 和 Bruno Legeard
- 测试面向对象的系统（Testing Object Oriented Systems）， Robert Binder
- Spec Explorer， <http://research.microsoft.com/projects/specexplorer/>
- NModel 建模工具（NModel Modeling tool）， <http://www.codeplex.com/NModel>

第三部分 测试工具和系统

第 9 章 缺陷和测试用例管理

阿伦·培智

在我加入微软之前，我是一个小型软件公司的第17位雇员，这个小公司就在微软园区（译注：微软把公司的楼和绿地等叫做园区，有个专用名词叫微软园区）附近。在这个小公司里，我第一次体会到了软件测试的挑战性。也就是在这个小公司里，我意识到了我有多大的热情和能力来迎接这些挑战。也是在那里，我第一次接触了缺陷跟踪系统。这个系统解决了我们团队的一些测试问题，但系统本身也有些问题。作为系统的开发人员，多年以后我才发现，当时，也就是1994年，我对很多缺陷管理的知识根本不知道。

记得那时候，我们用很原始的方法来跟踪缺陷。这些方法包括白板上的笔记，不同颜色的即时贴（sticky note）以及各种电子邮件。在这种情况下，用一个系统集中管理缺陷是有很多好处的，但是我们要求这系统能做更多的事情。我们要把缺陷指派给某个工程师，我们需要系统能提供更多关于缺陷的信息，比如重现（reproduce）缺陷的步骤，能重现缺陷的软件测试的版本号。我们还需要知道缺陷是什么时候，和怎样被修复（fix）的，以便于测试团队（也就是我本人）能验证缺陷的确被修复了。除此以外，我们需要报告结果，这样我们能够监测找到的缺陷和修复了的缺陷数目和类型。我设计的缺陷跟踪系统支持所有这些功能，但没有除此之外的功能了。说到底呢，这个系统能提供很有用的信息，能帮助工程师团队软件开发，但是它的运行速度慢而且一点也不灵活。我听说我离开这家小公司加入微软后不到一年，这个缺陷管理系统就被更新替换掉了。

当然，微软有个真正的缺陷跟踪系统。是微软内部某团队开发的，它用了个著名的杀虫剂品牌做为系统名称。该系统用微软SQL Server数据库存储缺陷信息，前端用户界面包括的一些栏目是我当时设计那个缺陷系统根本就没有想到的。我用微软的这个缺陷系统报告了我的第一个缺陷（当然还有第十个，第一百个）。这个系统也有它的缺点，后来我在微软工作五周年后，微软的大多数部门都使用了一个新的内部缺陷跟踪系统：Product Studio。Product Studio比其前一个系统的局限性少多了，而且该系统还可以根据不同产品、团队灵活定制所需的栏目和功能。在我们写本书的时候，虽然有些部门已经开始使用微软Visual Studio的团队系统（顺便说一句，该系统基本上是按照Product Studio开发的），但这个缺陷系统在微软依旧被广泛地应用着。

本章将包括微软广泛应用的缺陷跟踪系统和测试用例管理工具的信息、讨论和实例。另外，还给出了基于成千上万个微软测试团队和测试工程师的工作经验，教训和实践点津。

缺陷工作流程

缺陷和测试用例构成了几乎所有的测试队伍的两个最大的工作成果。简言之，测试用例描述测试过程的意图，缺陷则描述了这些测试用例的结果。当然，其他许多因素构成了这两个词完整的定义。在本章中，我希望能扩展对这些概念的传统认识和理解。

缺陷的工作流程描述了一个缺陷从创建到关闭的过程，所有的参与者，以及缺陷报告的所有可能途径。图9-1描述了缺陷的生命周期所有可能的途径。

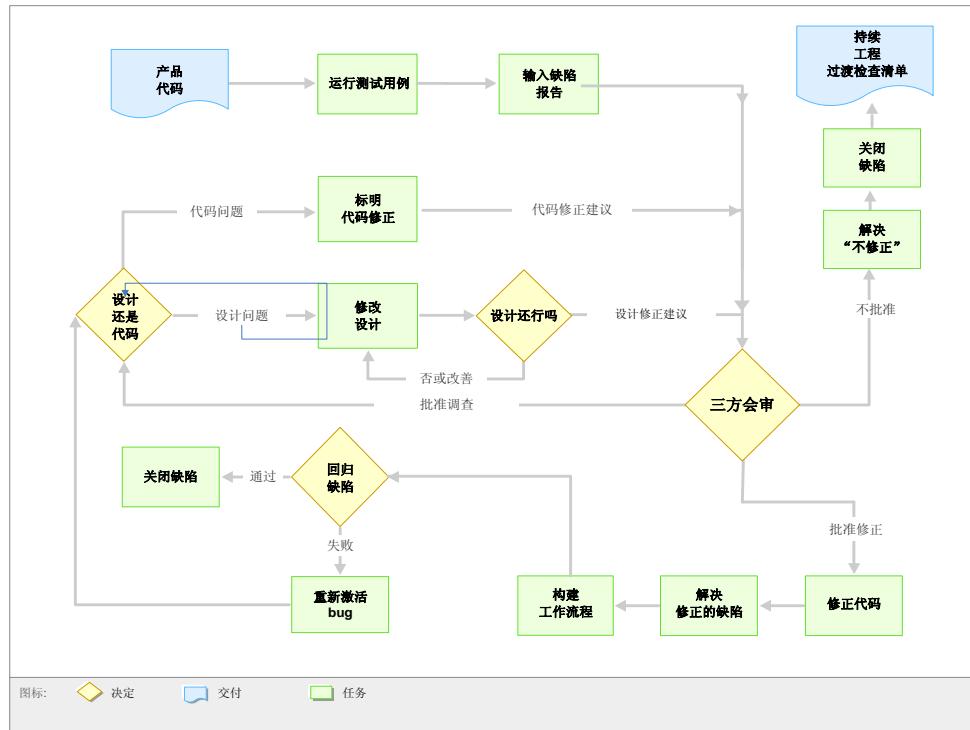


图9-1 缺陷的工作流程

译者注：缺陷的工作流程为：

- 产品代码->运行测试用例->创建缺陷报告->三方会审（triage）讨论缺陷。
- 如果缺陷没有被批准（not approved），把缺陷按照不修正（fix）来解决（resolve）->关闭缺陷。
- 如果缺陷批准了要调查（investigation approved）->研究是代码错误，还是设计错误。
- 如果是代码错误，提议修正代码错误，再进行三方会审->如果修正批准了->修正代码->解决缺陷（resolve 缺陷）->重现缺陷->通过了则关闭缺陷,如果不通过，重新激活缺陷->重新调查是代码错误还是设计错误。
- 如果是设计错误，修正设计直到批准->再进行三方会审。其它后续流程和以前类似。

缺陷的跟踪

软件测试工程师的最大的工作成果之一就是他们发现的缺陷。更加精确一点讲，是软件开发工程师编码时产生了这些缺陷，而测试工程师通过检查这些代码和程序发现了这些缺陷，和重现这些缺陷的具体步骤。在测试产品代码的整个过程中，测试团队能发现上千个缺陷。这些缺陷里，有些是相互关联的，有的则是相同的问题，不过是被不同的人发现的而已。决定

解决一个缺陷的步骤可能会经历很多变化，还可能有很多不同的所有者。对于任何测试团队来说，一个重要的工作任务就是使用的一个缺陷跟踪系统，及使用这套系统的共同的方法和准则。

一个缺陷的生命周期

一个软件缺陷生命开始于软件设计，编程以及其他软件开发的相关过程里。正如格言所说：“如果在一个没有人的森林里，一棵树倒下了，它会发出声音吗？”言外之意是它发出或没发出声音，没有人注意到。同样道理，客户也好、开发工程师也好，或者测试工程师也好，直到有人运行代码，我们才会知道缺陷的存在。

一个缺陷可以用很多种方式发现或记录下来，但如果测试工程师在运行一个测试用例时发现缺陷，他通常要编写缺陷报告，直接输入到缺陷跟踪系统里。三方会审成员定期审查所有的缺陷，确定其优先级别，然后指派给适当的人进行调查或修正缺陷。在修正了缺陷或者设计修改了之后，三方会审成员可能审查所有的改动，并决定是批准还是拒绝更改要求返工。有时，一些变化可能会被认定为风险太大，因此推迟到以后的版本。以下情形并不罕见，有的时候，我们觉得一个问题应该很小，到头来却是一个很大的工作项目，甚至可能破坏整个功能区（feature area）。

要有一个核心小组来审查缺陷，以及缺陷的修正，确保缺陷的修正是按照适当的优先级别进行的。一些缺陷可以确定为“不予修正”，也就是说虽然承认该缺陷，但不会修正，或者决定推迟修正，即该缺陷将被在未来的版本中修正。持续软件工程和产品支持小组以这种方式审查和解决缺陷。许多“不予修正”或推迟修正的缺陷将最终写入微软知识库的文章里。产品支持团队也可以读取他们支持产品的缺陷数据经常利用这些信息。

当缺陷修正被批准后，开发工程师进行相应的修正并整合到应用程序中。这时候，该缺陷在缺陷数据库被设定为已解决，然后是测试人员的工作来确认修正是否修复了原来的问题。如果缺陷仍然存在（即修正没有解决问题），缺陷会被重新激活，重新指派回到开发人员。如果缺陷被修正了，测试人员会关闭缺陷，但缺陷的生命并没有就此结束。修正的缺陷常常会被定期分析根源或其他关系。缺陷数据多年后仍然相当有用。

虽然从一个角度来说缺陷可被视为“出问题了”，缺陷的生命周期却经常体现了工程进程的脉搏。缺陷报告提供了关于剩余工作量，风险和项目总体状况的资料。事实上，许多公司（和微软的团队）把缺陷跟踪系统当作项目管理系统使用，记录缺陷的同时也记录工作事项。这样使用缺陷跟踪系统的优点之一，是可以简单的看到一个特定领域的所有工作（包括功能开发和缺陷修正），或了解指配到某个开发人员或开发组的工作量。表 9-1 和表 9-2 的例子显示出不同的方式去查看缺陷。正如任何包含大量数据的系统，能够以不同的方式查看数据有利于好的产品和工作流程的规划。例如查看指配给团队的每个开发人员的缺陷，和指配给每个功能区的缺陷可以帮助管理，平衡开发团队的缺陷。

注：在 2007 年，微软系统里输入了超过 1500 万的缺陷和项目管理记录。

表9-1 按负责人统计缺陷

缺陷 负责人	活动缺陷数量	已解决缺陷数量
Adam Carter	7	2
Michael Pfeiffer	5	0
Kim Akers	11	0
Chris Preston	4	3
Nader Issa	0	7
Hao Chen	0	4
Minesh Lad	0	2
Ben Smith	0	9

表 9-2 按工作领域统计缺陷

工作领域	活动缺陷数量	已解决缺陷数量
核心引擎	10	7
用户界面	6	3
Web网控件	16	

缺陷跟踪系统的特征

对刚刚成立的测试团队来说，缺陷跟踪系统往往最早被挑选或实现的工具。可以打赌，当你在阅读本书的时候，也许你正在使用某个缺陷跟踪系统。成功的缺陷跟踪系统常有一些重要的特征。这些特征使我们能区分即时贴条、工作表和被工程团队有效使用的成熟的系统。

简单好用是缺陷跟踪系统中最重要的要素之一。测试人员用这个系统来记录缺陷。测试管理层从缺陷数据中挖掘出很多情报，工程管理层经常会从缺陷数据中发掘出不同的情报。数据录入和数据展现必须简单，有效。

可设置性也是缺陷跟踪系统中重要的特征之一。不同团队对缺陷跟踪常常有不同的需求和要求。能自由添加字段来存储更多情报，能选择那些字段是系统必须或可选的，都很至关重要。举例来说，当存入缺陷时，微软有些团队只要求输入标题，描述，严重性和版本修改号。其他团队也许要求或多或少的额外信息。

可靠性是缺陷跟踪系统中另一至关重要的特性。作为工程团队中最常用的软件之一，这些系统必需七天二十四小时不间断的正常运转。微软内部IT部门负责维护那些支持微软软件缺陷系统的硬件和SQL系统。专用的带有备份电源，数据备份的硬件，实时的技术支持，使缺陷流程得以不间断的处理。

缺陷跟踪系统其他特性包括：

- 缺陷通知 当缺陷被指派给工程师或有变动，系统应该能通知相关工程师。在微软，最常用的通知系统是通过自动电子邮件系统直接从数据库中提取数据。独立系统直接连接SQL数据库进行实时变化报告也常被采纳。
- 互操作性 能够轻松从系统中提取数据并且显示于工作表中，Web应用程序或自定义的控件是非常有益的。
- 外部用户访问 客户或合作伙伴公司有时需要获得查看或修改缺陷，并能够通过代理共享缺陷。

为什么撰写缺陷报告

缺陷报告可以准确的，长期的记录软件的错误和错误处理的决策。对很多团队来说，也许电子邮件或走廊的谈话看似更容易解决缺陷，但用缺陷跟踪系统记录所有的缺陷有着显著的优势。在微软，缺陷根本原因分析是常见的。通过分析缺陷产生的工程过程和发现缺陷的工程过程，从而确定怎样才能在不同的工程过程中更有效的发现缺陷。这种类型的分析，通常被称为缺陷去除效率（DRE）。只有所有的缺陷都记录在案时，根本原因分析和 DRE 才更为有效。

缺陷报告往往成为历史参考资料，这些资料对开发未来版本的产品，持续工程团队，和产品支持部门都很有参考价值。这些部门依靠缺陷报告中的资料去了解产品，影响决策，或帮助客户。

在罕见的场合，缺陷报告可以成为法律辩护的依据。我知道一个案例，一个不是很关键然而严重的缺陷在产品周期后期被发现。几个高层管理人员和法律顾问开会议讨论，但最终决定推迟缺陷修正，并定于产品发布后不久再修正。结果客户发现了这一缺陷，并提出了起诉。幸运的是，缺陷报告中的资料足以证明，微软并没有恶意的意图，案件也因此被驳回。

解剖缺陷报告

不管你跟踪了多少数据，或者你的缺陷跟踪系统中有多少字段要求，几个关键的要素使高度可操作的缺陷报告区别与一堆可能导致你错误方向的数据，或可能会错误地被忽视了的数据。表9-3列出了好缺陷报告的主要特性。

表 9-3 好缺陷报告的特点

属性	说明
标题	<p>在微软，缺陷的标题也许是缺陷报告中最重要的（或最常用的）的信息。系统的临时用户可以扫描标题去了解产品中或某一特定领域的缺陷类型。标题也是缺陷系统中被搜索最多的字段，它使我们能便捷地找到特定领域的类似的缺陷，这些缺陷可能很难用其他字段跟踪。标题还是产品功能或会审小组便于审查的重要信息。一个标题，短短的 80 来字，需要提供准确的缺陷全面总结。它需要很好但不过分的描述，提供恰到好处的信息。微软的不少测试人员在迅速地填补缺陷报告的各种信息之后，往往花上额外的时间来字斟句酌标题。</p> <p>以下是一些缺陷的标题和评价：</p> <ul style="list-style-type: none">• 程序崩溃 – 太短• 在同一时间运行多个同一程序，其中之一对话框崩溃 – 既冗长又模糊• 在低内存条件下，设置对话框中程序崩溃在– 具体、准确，能从报告中了解足够的信息。
说明	缺陷说明回答了标题中不显而易见的问题。它包括一个简要的总结，客户影响的信息和预期的结果与实际结果。表明实际和预期的结果便于各方澄清正确的系统表现。
状态	缺陷状态要么是“活动的（Active）”、“已解决（Resolved）”、或“关闭（Closed）”，它反映了缺陷的处理需求。新缺陷的状态是“活动的”，并维持“活动”状态直到找到解决方案，这时缺陷的状态会被设成“已解决”。缺陷一经解决，测试人员会验证修正，或者重新设置“活动”状态（如果修正没有解决问题）或设置“关闭”状态（如原来的问题确实被修正了）。
版本号	所有的缺陷应注明被发现的软件版本号。在微软，大多数产品是每天重新构建，在某些情况下甚至更频繁。知道缺陷出现的确切版本对重现缺陷或

	核实修正是大有帮助的。
功能区	微软很多团队要求记录缺陷是在产品某个区域或分区发现的。例如，微软 Windows 缺陷数据库可能有一个文件系统领域和分区，包括 NTFS, FAT, 或其他文件系统相关的领域。准确的功能区和分区的信息，有利于纵观产品各领域，以确定哪些区域可能有危险或可能需要更多的时间来了解和解决更多的缺陷。
重现步骤	重现（Reproduction）的步骤往往是包括在说明中，但有些系统单独列出这一缺陷报告的重要组成部分。微软所称的重现步骤，是指可以被缺陷工作流程参与者重新再使缺陷出现的步骤。最令测试人员沮丧和倍感浪费时间的是开发工程师说，“这缺陷并不能在我的电脑上重现。”一套好的重现步骤并不能保证这种情况不会发生，但会大大降低这种可能性。重现的步骤也必须尽可能简明扼要。虽然缺陷可能在 10 个步骤下被重现，花些时间看看是否有些不必要的步骤，有没有可能减少重现的步骤是很重要的。减少重现步骤会有助于迅速隔离缺陷根本原因，从而增加了第一次就能正确修正缺陷的机会。
分配	“指派给（Assigned To）”是微软所有缺陷系统中都需要的字段，但不同团队有不同的用法。许多团队把缺陷指派给负责该领域的开发工程师（如知道是谁的话）。其他团队倾向于指派缺陷为“活动的”（“活动的”是Product Studio 缺陷系统指派的默认栏目值），并依靠缺陷会审小组指派适当的人选。每个缺陷在任何时候都只能指派给一个人。接到被指派缺陷的人负责解决问题，（即修正缺陷）或重新把缺陷指派给其他人。
严重性	严重性描述了缺陷对客户，对开发过程和整个缺陷工作流程的影响。严重性考虑到缺陷影响程度，频率和缺陷重现率，通常是一个 1 至 4 的数值，其中，1 是最高的严重性。微软多数缺陷数据库使用下列严重性定义： <ol style="list-style-type: none"> 1. 缺陷导致系统崩溃或数据丢失 2. 缺陷导致主要功能或其他严重的问题；产品在不明情况下崩溃。 3. 缺陷导致次要功能问题，可能会影响产品最终的完美体现。 4. 缺陷包括错字，不明确的措辞，或低可视性领域的错误讯息。
客户影响	在缺陷报告中记录客户影响说明是很重要的。客户影响应包括该缺陷对用户的影响，以及如何影响到客户的情景和要求。撰写客户影响说明应考虑以下几个方面因素： <ul style="list-style-type: none"> • 确定缺陷影响客户的情景和要求。

	<ul style="list-style-type: none"> 确定客户遇到该问题的频率和或然率。 调整严重性字段，以真实反应缺陷对客户的影响。
环境	<p>重要的是要在缺陷报告里清楚地描述测试环境条件，和必要的步骤来重现这种环境。您甚至可能需要描述哪些前提条件不存在，或没有尝试过什么情况。这使得其他人容易找到并重现缺陷。环境的细节可以包括以下内容：</p> <ul style="list-style-type: none"> 硬件规格和配置 系统、组件和应用程序版本 工具和流程的采用 相关连接和数据配置 角色、权限、以及其他适用的设置 说明已被淘汰的环境因素
决断	<p>当缺陷被解决时需要填写此字段。对决断字段，几乎所有的微软产品都使用以下的选项。</p> <ul style="list-style-type: none"> 已修正 缺陷被修补好。 不能重现 缺陷不能被重现。这通常发生在重现步骤不完整，或测试和开发人员的计算机环境不同。 重复 当两个不同的缺陷描述同样的问题，其中一个缺陷（通常是后来报告的）会被断定为重复。更多重复缺陷的信息请见此章节后侧栏标题为“一个重复缺陷的注记”。 根据设计 有时看似是一个缺陷而实际上并不是。相反，所见行为是有意的设计。例如，当我运行 Windows 计算器，手按进“2/0”，输出窗口的计算器内容显示“不能除以零”（这是预期结果）。这之后，所有数字键都不工作，直到我按 C 键清除数据。有些人可能会认为这是一个缺陷，但它是“设计所导致的”。这种决断已经成为著名的测试人员的口号“这不是一个缺陷，这是一个功能。” 推迟 这项决断是考虑在未来产品版本中修理。

此外，缺陷数据库中普遍使用的字段还有：

- 如何发现 (How Found): 什么测试活动发现的错误?
- 问题类型 (Issue Type): 是编码错误，设计的问题，文档的问题，等等?
- 缺陷类型 (Bug Type): 缺陷类型可能是安全、性能、功能、压力等等。

- 资料来源 (Source): 谁发现的错误? 测试、开发、内部用户、测试用户、或其他人找到缺陷?

图 9-2 显示了一个例证: Visual Studio Team System 软件缺陷跟踪系统。

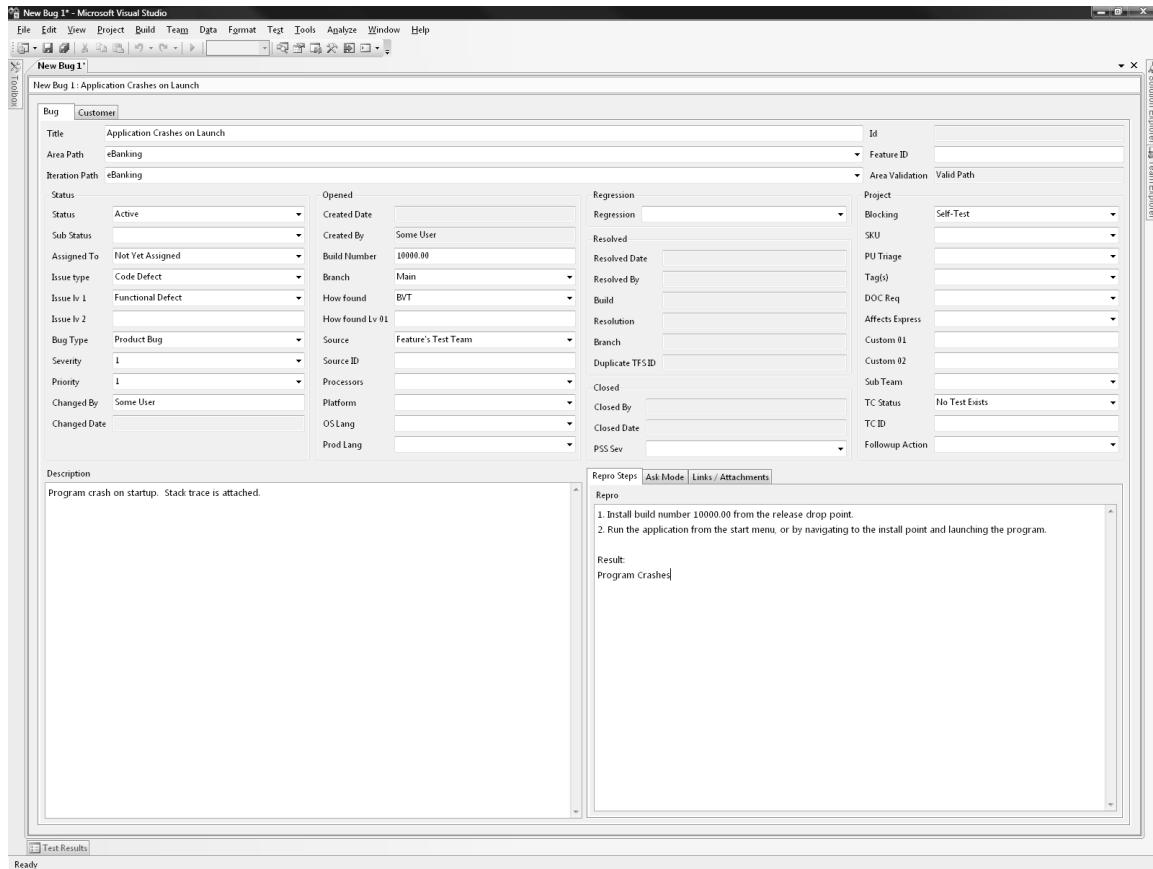


图 9-2 Visual Studio Team System 缺陷跟踪系统。

要避免在缺陷数据库中增加太多的字段。每一个月左右的时间，我们往往想增加额外字段来跟踪“该月很酷的新度量”。保持系统的简单会让团队的每个人，以及其他团队能有一致的系统使用体验。

系统中设置专用字段是有益的，但太多的字段可能会造成混乱，在解决缺陷时也易发生错误。如果有太多的必填字段，工程师可能倾向在缺陷系统之外解决缺陷，这样，就失去了未来分析的缺陷数据。

缺陷会审

缺陷会审是从医疗行业借用的术语，几乎微软每一个产品开发团队都有缺陷会审制度。在急诊室，会审过程是用来判断哪些地方需要首先集中医疗资源。换言之，这种方法是用来确定需要先处理头部受伤，然后是手指骨折，最后才是皮疹。

“这不是一个缺陷，这是一个功能!”

每一个在软件项目工作过的人可能都听说过这一说法“这不是一个缺陷，这是一个功能。”这常常是用以表明，看似缺陷的东西，其实是有意的设计。有时，测试工程师报告缺陷是因为不理解设计。还有时，测试工程师认为设计其实是真的错误了。在多数情况下，这句话其实是以讽刺的口吻来形容除了设计者之外，每个人都认为这是一个缺陷。

有一天，我花了几个小时和一个开发工程师争吵一个被定论为“根据设计”的缺陷。我认为他的决断更多是出于懒惰和冷漠而不是真诚的设计决定。最后，我想我们都放弃了，决定回家过夜。我走过停车场试图记起我停泊的地方（一天工作劳累后，这是经常发生的），当我看到一个老大众甲壳虫（也叫“缺陷”）停泊在其中一个车位时，不禁笑了。这款车是深蓝色的，但引擎上有一个黄色的套，中间有一块板，上面写着“功能”。

在微软，会审概念应用于报告的缺陷而不是接收的病人。缺陷会审发生在产品的各个层次。会审小组的主要责任是充当决策者，并确保缺陷工作流程的预期进行。会审小组由特定的产品或功能区跨专业的决策代表（通常是开发，测试和项目管理）组成。作为缺陷工作流程中缺陷初始路径的决策者，他们可以指定某个开发工程师来修复程序（Fix）、做更多的调查（Investigate）、或定论这是一个重复的（Duplicate）、要推迟的（Postpone）、外部的（External）、根据设计的（By Design），或其他一些决断，而把缺陷返还给报告者。会审小组还可以查看重复的缺陷报告，或把相关的缺陷连接在一起。他们的主要任务还包括完成缺陷的优先排序。有些缺陷需要立即修正，而另一些缺陷可以在产品生产后期去修复，甚至在产品发布后再修正。有些类型的缺陷甚至不会被修正。这些都是缺陷会审小组决定的。

高质量的缺陷报告是至关重要的，它使缺陷会审小组可以做出正确的决定，并确保缺陷适当的处理路径和优先顺序。有些团队要求报告缺陷的人在系统中记录缺陷的优先度，然后由会审小组调整。有些团队选择录入空白优先度字段，依靠缺陷会审小组指派缺陷的初始优先顺序。缺陷的优先顺序通常设定值如下：

1. **必须修正（Must Fix）** 尽快修正这个缺陷。缺陷影响产品在这一领域的进一步进展。
2. **应修正（Should fix）** 此缺陷应该尽快解决，在产品发布前，在这一里程碑末，或在下一轮开发之前。
3. **有时间就修正（Fix if time）** 这是一个小缺陷，可推迟取决于产品开发的阶段。

按三个等级划分缺陷的优先顺序可能有些困难。有些团队使用了附加优先级别或使用缺陷数据库中其他字段来代表更细化的优先角度。只要有一致的缺陷优先评价系统，并采取适当的应对措施，任何优先权系统都可以有效的操作。

当产品临近发布时，大多数缺陷会审发生在项目管理的高层。团队或部门高级管理人员审查新缺陷并作出决定，是否修正每一个处于活动状态的缺陷。越接近产品发布日期，审批修正缺陷的标准更高。因为代码更改总是有一定的风险，所以临近产品发布，会审小组会斟酌再三，只修正非常关键的缺陷。

缺陷会审小组在产品开发晚期的目标是帮助团队成功地达到零缺陷。当然，曾经从事软件产

品的人都知道，所有的软件发布时都有少数缺陷。微软的产品也没有例外。每一件产品发布时都有一些已知缺陷。零缺陷概念的真正意义是，据我们所知，没有缺陷将阻止我们发布这一产品。新的缺陷被会审，然后被指派给开发工程师修正，或决定推迟解决，或决定将不会修正。最后，当缺陷修正的标准越提越高，活动的缺陷数降为零。有时候，这就是所谓的零缺陷反弹。如果仍然有一些收尾工作要做，或最近的修正引起了其他的错误，或发现了新的缺陷，缺陷计数反弹从零到正数。然而最终，缺陷计数达到零、产品也宣布发布了。在这一阶段，产品仍然有缺陷，但会审小组认为这些缺陷都没有严重到可以阻止产品发布的程度。

缺陷报告中常见的错误

一个缺陷跟踪系统如果使用不正确或不适当有可能失去它的价值。虽然具有系统的好处远远大于它的短处，在编写指南时，教育他们如何正确使用该系统时，一定要考虑到这些危害。很多错误的发生只是未遵循表 9-3 中列出的好缺陷报告中特性。在缺陷报告经常发生的其他一些有趣的错误包括一些电子邮件讨论，缺陷的变种，以及在同一份报告中罗列多个缺陷。表 9-4 中列出这些类型的错误的例子。

表 9-4 缺陷报告中常见错误

错误	实例
电子邮件讨论	<p>2007 年 10 月 28 日下午 5 时 38 Jim Hance 报告 同时打开多个实例应用程序时，程序崩溃。不阻碍测试，但应该修正。</p> <p>2007 年 8 月 28 日下午 5 点 53 David Pelton 指派给 Jim Hance 我会尽快解决此问题。错误发生在打开第二个实例，或更多？ 你是否认为这是涉及到新的经理？</p> <p>2007 年 8 月 28 日下午 6 时 15 分 Jim Hance 指派给 David Pelton 哈哈。缺陷发生在第二个实例（您应该先试试）。 跟经理没有任何关系，但也许与办公室搬家有关。</p> <p>2007 年 8 月 28 日下午 6 时 34 David Pelton 指派给 Jim Hance 这*真是*有趣。顺便说一句，如果你打算很快回家，能载我一程吗？</p> <p>2007 年 8 月 28 日下午 6 时 41 Jim Hance 指派给 David Pelton 没问题，15 分钟后在你办公室见。不要忘记把这缺陷返还给我。</p>
	<p>电子邮件和缺陷系统是微软大多数工程师最常用的两个工具。所以不奇怪，工程师们有时将两个工具混用。</p> <p>然而除了测试和开发工程师外，缺陷报告还有广泛的用途。所以和缺陷不直接相关的信息不应被写入报告。</p>
缺陷渐变	<p>2007 年 9 月 17 日上午 09 时 34 分 Don Hall 报告 我无法从我们的应用程序中打印。环境条件和重现步骤如下</p>

	<p>...</p> <p>2007年9月17日上午9时49分 Ann Beebe 指派给 Don Hall 我看了看环境信息，你错误地配置了% PRINT_INFO %变量。纠正这错误应该就没问题了。然而，看过你的笔记后，我发现了一个不同的问题。打印对话框中有一些布局问题，需要加以解决。</p> <p>2007年9月17日上午10时37分 Don Hall 指派给 Ann Beebe 我已经更新了安装脚本去设置那变量，我更改了这个缺陷报告的标题，来反映这布局问题。谢谢。</p>
--	--

缺陷渐变是指在同一缺陷报告中，缺陷从一个问题演变成另一个完全不同的问题。这种现象有时发生很快，有时要过几天或几个月。不论怎样，缺陷渐变绝不是好主意。
对变形的缺陷，通常很难分析其根本原因。产品支持或持续工程部门在审查产品或搜索客户的问题来发现这些问题是否由已知缺陷引起时，这些缺陷还可能造成混淆。如果一个缺陷报告开始演变，要及时停止，并就新的问题重新报告一个新的缺陷。

多个缺陷	<p>2007年8月28日下午6点13分 Jeff Ressler 报告 在布局引擎上运行版本验证测试时，我发现以下错误：</p> <ol style="list-style-type: none"> 1 手动调整应用程序窗口的尺寸小于 200 × 180 时，控件重叠。 2 最大化窗口时，两个控件超越了屏幕的界限。 3 配置对话框中的文字有拼写错误，即“configuration”。
------	---

如果测试人员很忙碌，他们可能会把有些相关的缺陷记录放在一个缺陷报告中。
尽管我们尽力避免这类问题，在一份报告中报告好几个缺陷从来都不是一个好主意。如先前所提，在一个缺陷报告中包括多个错误会给缺陷报告的其他使用者造成问题，例如：

- 缺陷的优先度不能单独设置。
- 缺陷的决议不能单独设置。如果会审小组决定其中一个缺陷应该要推迟到不同的版本，就没有办法注明这一点。
- 虽然缺陷似乎是在类似领域，他们可能需要分配给不同的开发工程师。
- 在分析产品缺陷的根本原因时，同一缺陷报告中的每一缺陷可能有不同的错误根源。

一个重复缺陷的注记

当我谈论缺陷数据或数据的潜在用途时，通常会提到我对重复缺陷的看法。几乎在我所工作过的所有测试队伍里，测试工程师报告重复的缺陷会被认为是“坏事”。有些团队甚至使用各种各样的算法，来计算发现的缺陷，修正的缺陷，和缺陷解决型的比率，希望能从中推算出一个神奇的测试效率数字。

我想重复的缺陷似乎是不好，因为潜在的事实是，这是在浪费别人的时间。

测试工程师 Bobby 报告了一个几天之前 Jane 刚刚报告过的缺陷，于是他神经紧张了：现在要有人花时间来决断说这个缺陷是重复的。然而，如果他们测试的是产品的不同领域，并没有认识到 Jane 报告的缺陷实际上是在一个共同的组成部件。坏 Bobby，坏 Bobby！

我一点都不认为输入重复的缺陷是坏事，事实上有这个担心是不对的。让我告诉你为什么。

如果测试工程师输入重复缺陷有任何消极后果的话，那是说测试工程师会因担心输入重复缺陷而不敢大胆报告缺陷。想想看，我发现了一个重大错误。然后，我要花一些时间来找找是否是已知的缺陷，我感觉有一个可能是相似的。如果输入这个缺陷有负面的影响，我也许更倾向于不输入这个缺陷。理想状况下，我会记下这错误，并验证它解决了我刚才发现的这问题，不然我就要发电子邮件给原来报缺陷的人，听听她的意见，但在许多情况下，测试工程师只是认为，“这可能是重复的，”然后继续其他的事，即使这意味着错过了一个潜在的重要问题。

输入重复缺陷是浪费别人时间的想法是错误的。很多测试工程师很了解自己的测试领域，但可能对系统的其他部分了解不多。当我发现一个缺陷，可能会花上 20 分钟去看看是否有重复缺陷或周围领域是否有类似或相同错误。如果我直接输入缺陷，让缺陷会审小组（或谁负责检查新缺陷）审查，他们很可能在远远少于 20 分钟之内发现这是一个重复缺陷。不然的话，只有我自己浪费了时间。

谁在乎测试工程师是否输入了一个重复缺陷？一个缺陷报告中的信息往往不能提供足够的信息来诊断问题。关于同一问题的另一份报告有可能引导开发工程师的思维而发现缺陷根源或更有效的修补程序。多数缺陷数据库系统有办法注明“相关”（或“重复”）并且保留缺陷之间的联系。在我看来，越多的缺陷数据总是越好，重复的缺陷是提供这些数据的副产品。

数据使用

管理层最喜欢的事情之一似乎就是从缺陷数据库中得出的报告。这些报告发掘和代表了系统中各种各样的信息。没有神奇的公式或查询告诉您这些项目是否已经准备就绪、可以发行、或者是遇到麻烦，但是有无数的方法可以研究数据。微软的团队用数百种不同的排列组合来研究缺陷数据。表 9-5 列举了一些缺陷度量（metrics）及其潜在用途的实例。

表 9-5 度量和用途

度量	度量用途
修复的缺陷/所有解决了的缺陷	缺陷修正和其他决断的比例。在产品开发早期，预计发现缺陷的数字比缺陷得到解决的数字更多；接近产品开发后期，期望得到解决的缺陷数比发现缺陷数要多。这个度量也可以帮助你建立达到零缺陷的预测模型。

每种语言总计缺陷	测试本地化版本成本参照。这个度量可以提供一些线索来促进更加有效的本地化工作。
缺陷发现率	太高或太低都有担心；对峰值应加以解释。
错误修正率	缺陷被修正的百分比。当缺陷会审标准提高时，修正的百分比应逐渐下降。
每个代码区的缺陷数	根据功能排序来列出最多报告的缺陷，这些数据可以影响哪些领域需要更多的测试。
每个功能区发现的缺陷	每个测试团队、内部用户、开发工程师、产品支持、和外部 Beta 测试者发现缺陷数能够影响测试的策略。
不同严重性的缺陷	随着项目的进展，期望看到严重性 1 和严重性 2 的缺陷发现率下降，而严重性 3 和更低的缺陷百分比增加。也就是说，我们一般期待在产品早期发现那些严重的缺陷。
哪里发现	这一度量对于不同测试产品的类型可以有所不同。知道缺陷在产品那里出现可以揭示产品的风险领域。
如何发现	了解缺陷是如何被发现的可以帮助根源分析和实现缺陷防止技术。
缺陷产生时候	知道问题出现在产品开发的哪个阶段（例如，规范，设计，编码，缺陷修复）可以帮助决定缺陷防止技术需要在哪些方面得到实现。
缺陷重新激活率	这是一个衡量程序修正质量的很好度量；在项目接近尾声时，缺陷修复量达到最大，缺陷重新激活率也往往会增多。
每个测试活动发现的缺陷	分析哪些类型的测试会发现缺陷。各种测试活动包括探索性测试、结构化测试、发布前测试、测试用例开发、配置测试、打印机测试、自动化测试、一般产品的使用、测试版测试、每次测试通过（Test pass）、验收测试等等。
平均解决缺陷的时间	跟踪开发团队对输入的缺陷的反应速度。

平均关闭缺陷的时间	跟踪缺陷的平均反应时间,或完成缺陷工作流程所需时间。最理想的状态是,开发人员尽快地修复缺陷,测试人员尽快地验证修复。
-----------	--

当然,如同大多数的度量,缺陷数据用图表表达会更有帮助。

图 9-3 和图 9-4 是两个用简单的图形格式来表达缺陷趋势信息的例子。在微软,这些各种各样的缺陷图表会定期内部发送给产品团队,以便快速传播相关缺陷资料。

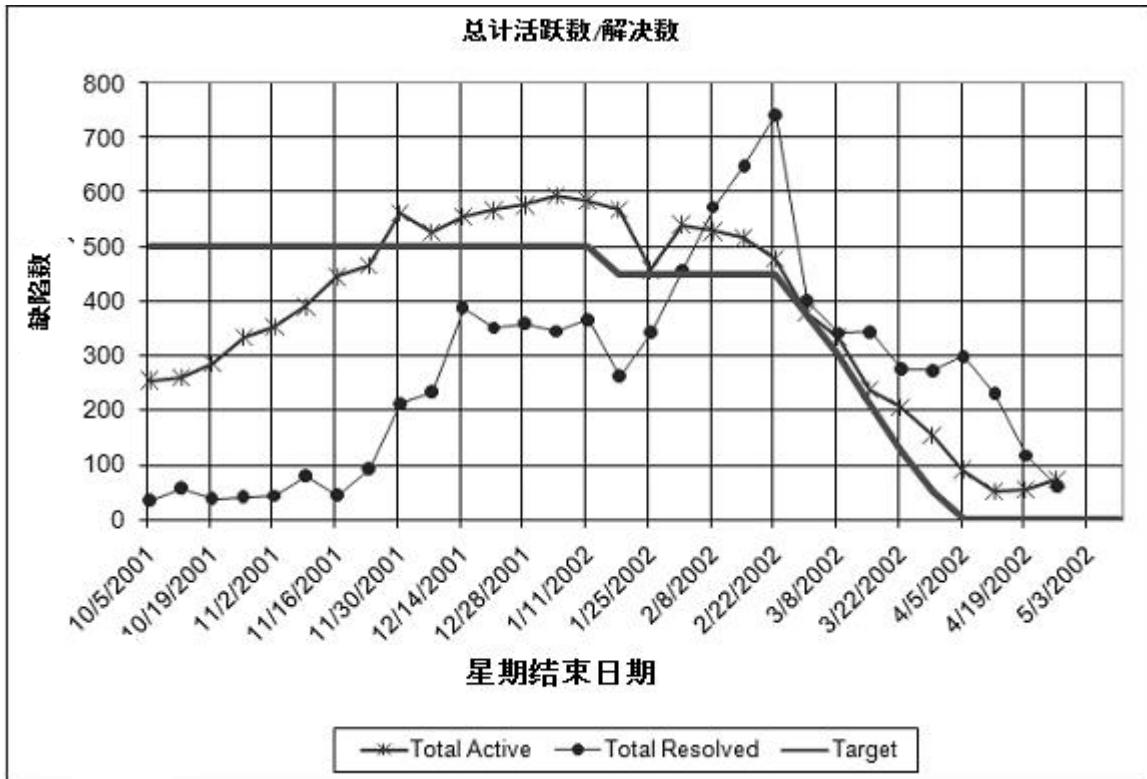


图 9-3 活跃的和解决的缺陷预测趋势线。

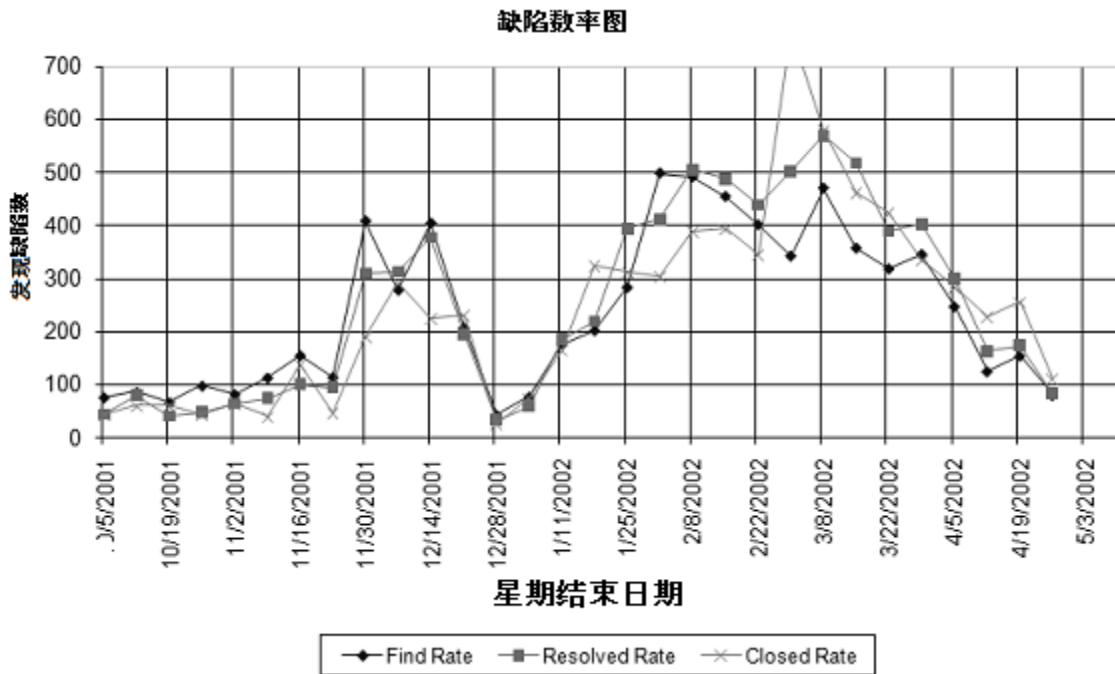


图9-4 一周缺陷发现、解决和关闭率

何时不能使用缺陷数据：缺陷作为绩效度量

使用缺陷数据来测量绩效是诱人的。测试人员是找缺陷的，因此您期望好的测试人员找到很多缺陷。许多管理人员通过收集和跟踪缺陷数据来进行绩效管理。然而，缺陷数量报告仅能对个人业绩提供非常有限的参考。尤其是同事人之间的比较时，缺陷数据具有太多的可变量，比如下面的几方面：

- 所测试功能的复杂性
- 开发人员编程能力
- 规格完整性
- 缺陷预防与缺陷发现
- 报告的及时性

此外，如果有人打算利用缺陷数量数作为一个业绩考核标准，该人必须理解该标准的参数和考虑到诸如如下的问题：

- 报告的缺陷具有什么严重性和优先度，分布如何？
- 功能缺陷与简单的用户界面缺陷一样算数量吗？
- 花费时间（一天或几天）追踪一个关键问题（如数据丢失，内存泄漏）并使之得到解决，这能说明没有达到预期或业绩表现差吗？如果是，什么是团队合作的政策，即协助开发人员排解疑难问题？
- 缺陷质量是一个因素吗？如果是的话，在团队里，这些具体因素是如何决定缺陷的？团队平均值是什么？平均数是目标吗？哪些具体的因素是超过预期的目标？
- 每一次评比，最低的缺陷数量是什么？什么样的缺陷数量是测试人员超过期望的数量？

发现了大量的缺陷可能表明测试人员做的很好，或者它可能意味着开发人员编写的代码很差。反之，如果一个测试人员找到很少的缺陷，这可能是一个迹象：表明他做得不理想，也可能意味着他正在测试高质量的代码，具有较低的缺陷密度。所以关键是怎样解读数据，这也意味着可能需要额外的个案调查。例如，如果一个测试人员没有报告很多缺陷，看一下功能区以确定是什么原因造成缺陷数量低。如果其他用户（客户、开发，Beta测试用户）在该功能区找到缺陷，该测试人员的低缺陷数可能会有问题。如果是测试运行次数（用测试用例或代码覆盖信息为度量），低缺陷数量也是值得调查的。当然，如果进一步的调查后，您确定功能区的测试不错，没有多少缺陷，这当然就不该怪测试人员了。

一个缺陷度量的故事

当我刚开始在微软工作时，对找缺陷数量有特定的要求。我的经理告诉我，团队里的每一个测试人员每星期应找到 10 个缺陷。这似乎像一个合理的要求，所以我努力去工作，并开始寻找缺陷。像大多数微软的员工，我一直想做得更多一点，所以我每星期找出 12 或 13 个缺陷。

幸运的是，我所测试的领域总是在变化，对我来说，完成配额从来没有问题。事实上，有几个星期，我发现 20 个或更多的缺陷！当发生这种情况，我却很担心，我已发现太多的缺陷，下周我将无法完成我的配额。所以，我每周只报 13 个缺陷左右，这样来“节省”缺陷，以防下一周我的缺陷枯竭。

这是另一类典型的例子，它表明你衡量的是什么。我的经理每星期要 10 个缺陷，我给了他所想要的，却不管我是否能发现更多的缺陷。我一再看到有人企图利用缺陷度量来衡量个人业绩，但这些度量很少有效。

缺陷门槛（缺陷 Bars）

在微软，一个日益普及的流程概念叫缺陷门槛（*缺陷 bars*）。最简单的来说，一个缺陷门槛就是一个开发人员在某个特定时间所能接受的一定数量的缺陷。如果分配给开发人员的缺陷数量超过此数量，则开发人员预计将停止功能的开发工作来修改缺陷。根据不同的具体规则，开发人员可能需要将他们的缺陷数接近零或限制在低于某个限度内。

如果能正确使用缺陷门槛，它可以成为一项有效的工程工具，但与任何衡量个人度量的工具一样，它们可能被滥用。肆无忌惮的开发人员可能会要求测试人员，把“这些缺陷放在电子邮件里”，或者可能贿赂（*bribe*）同事“从我的盘子（*plate*）里拿走一些缺陷”，以保持其缺陷数量低一些。当衡量人的时候，无论这种改变是否支持你原有的目标你总会得到你所想要的。

两个开发人员的轶事

Rob 和 Kirk 是在一个产品小组的开发人员。他们的团队使用了缺陷门槛的规定，如果有 10 个以上的产品缺陷被分配给开发人员，他们就需要停止其功能开发工作，只有解决所有的缺陷，才能继续功能开发。

Rob担心缺陷门槛将会减慢自己的开发工作。他倾向于完成所有功能开发工作，然后再一次性改正所有的缺陷，但他也认为，缺陷门槛将有助于保持降低总缺陷数。开发工作如期进行，Rob立即开始开发他负责的功能（features）。虽然有几个代码缺陷，但他的工作进展顺利，并只有6个缺陷分配给他，所以他继续努力在功能的开发上。一个星期后，Rob接近完成他的最新功能开发，他的老板来到他的办公室告诉他，他现在有12个缺陷，他必须停止开发来修补这些缺陷。对不得不停止开发工作，Rob感到失望，当他意识到大多数缺陷是一个月以前的代码时，失望感更强烈，他不得不花时间重新研究以前的工作以修正这些缺陷。

Kirk也担心缺陷门槛（缺陷 bars）会减缓自己的开发工作，但他愿意去试一试。Kirk开始编写代码，当不久后他第一次签入（check-in）代码，两个缺陷在他的代码里被发现。不等缺陷数达到或超过了10，Kirk决定停止开发，看看他能否立即解决这些缺陷。这些有缺陷的代码是Kirk几天前写的，他非常惊讶缺陷是多么容易被修复如果代码仍然记忆犹新。他继续写代码，并继续尽快处理分配的缺陷。他发现，修复刚完成的代码的缺陷比以后再修复要容易得多，他还发现，早期修复缺陷能帮助后来编写更好的代码，尽管他会犯错误，但很少再犯同样的错误。由于他的新的开发和修复缺陷的方式，Kirk成为开发团队最有成效的和受人尊重的开发人员之一。

有些团队把缺陷门槛看作一个过程（process），它可以在任何时间段内限制总活动的缺陷数。例如，如果团队有 20 个开发人员，缺陷门槛是每个开发人员只能有 10 缺陷，数学上来说（理论上）任何时候将永远不会有超过 200 个缺陷。在实践中，这一理论可能正确也可能不正确，但降低整体的缺陷数不是缺陷门槛的目的。它的意图是迫使每个开发人员在最短的时间内修补缺陷。在先前的例子，Kirk 理解其中的含义，但 Rob 就没有。一个（没能真正理解的，英文为 ill-conceived）解决办法是降低缺陷数量。例如，如果缺陷数定为 5，Rob 将不得不早点修正自己的缺陷！虽然这种解决办法可能有效（也可能无效），真正的解决办法是让整个团队知道什么是目标以及修正缺陷的意图。

当然，如果修正缺陷的目标是尽可能地接近缺陷发现的时间，那么缺陷就应该在它刚产生时尽早地被发现。换言之，这样的系统要求测试团队从一开始就参与功能的开发工作。如果没有人在早期发现缺陷，就没有人能早点修复缺陷。

微软的典型缺陷

在微软，每年缺陷追踪系统记载着数以百万计的不同缺陷。许多是产品缺陷，但待做工作项目（work items），在不同的缺陷数据库的重复的缺陷，以及测试工具缺陷和测试代码缺陷也导致问题总数的增加。但是，每年少数问题会落入一个完全不同的类别。几个例子如下。

缺陷 #65889：新的 2% 的牛奶纸盒行不通。因为他们太难打开

由 XXX 报告

新的 2% 牛奶纸盒显然是行不通的。他们不能被正确打开。这似乎由于过时的设计造成的。
35 号楼也有相同的问题。

显然，这是一个优先级（priority）为 1，严重性（Severity）为 1 的缺陷，因为我每天都会遇

到 2 到 3 次这个问题。

微软餐饮部回应：

感谢您与我们联系有关新牛奶纸盒的问题。我们已找到了原因，牛奶盒很难打开是因为牛奶供应商刚刚买了崭新的能生产品脱大小的牛奶盒的机器，他们正在调整目前的机器以免有这样紧的封盒（seal）。谢谢您报告的问题，如果您需要更多信息，请随时与我联系。

建议替代解决方法：

1. 喝水代替。
2. 带自己的奶牛。
3. 使用电梯门打开了封盒（seal）。
4. 冻结牛奶纸盒，让冷冻牛奶撕裂纸盒，然后解冻。
5. 告诉你的经理，你没有牛奶就不能工作，让他解决这个问题。

此缺陷正造成大量波动，这可能酸化我们试图在这个星期回归整合(RI, Reverse Integration)的努力。我希望我们能迅速解决这个题。

据报告，在微软 Sammamish 园区有完全相同的问题。我们发现当地一个替代方法可能有帮助。位于南部的一点三五英里有一个近似替代 2% 奶瓶，夸脱（quart）大小的容器，非常容易打开。缺点是，牛奶太多，一个人无法舒适地喝完。额外的解决方案是找到 2-3 也想在同一时间喝牛奶的人。我不敢肯定，这些方案能降低缺陷的严重性，因为与此相关的警告：(1) 替代源的地理位置相对同楼的厨房冰箱太不方便，(2) 有效率的消费需要额外的资源。

我从我们的奶制品供应商得到最新的消息，他们修复的缺陷将无法立即投入使用，因为新的修复将需要通过广泛的测试。

测试人员拒绝签署该修复测试通过。他们说，他们只是进行了非正式的修复测试，但尚未运行充分的回归测试。目前，只有三个测试人员正在处理这部分，他们一天只能饮用 8 盒牛奶。该小组可以作更多的纸盒打开测试，但牛奶品尝测试，牛奶溢流测试和纸盒压力测试仍然没有做。此外，由于封盒（seal）已较松散，他们一直在观察压力测试的牛奶溢流。

测试需要 3-4 周时间。

缺陷 68648：求爱缺陷

安妮

一年前，你发现一个 Windows 95 的问题，然后与我的经理和产品组共同将我调来这里，看我是否能帮助在 Windows 98 解决同样的问题。你给我提供建议说要从数据得到正确的知名度（visibility）。没想到，其实你给了我更多。您给予我生命的意义，让我笑，让我哭，让我捡杂草。你知道我疯狂地爱上了你。我爱你的一切。如果没有你，我的生命没有任何意义。我知道这不是最浪漫的方式，但如果你愿作我的妻子，让我分享您的生活，我会永远幸福。你愿意嫁给我吗？

重现问题的步骤：

1. 结婚。
2. 生孩子。

对不起，我无法抗拒。

（安妮回复说：）好的！

我真不敢相信你这样做了！

缺陷通常是痛苦地提醒人们出的一些错。随意报告缺陷往往是娱乐性地提醒我们：除了注意到工作要做的事和失败以外，还有更多的生活和更多的软件开发。

测试用例管理

缺陷的发现是通过产品开发过程中各种各样的活动，一个软件产品中发现的大多数缺陷是从基于测试用例的文档而进行的针对性的测试取得的。在微软，即使是最小的项目，也有数以千计的测试用例。一个较大的产品通常会有几十万或更多的测试用例。存储和组织大量测试用例需要一个测试用例管理系统。

一个测试用例管理器（*test case manager*, 简称 TCM）是一个系统，它可以管理测试用例的定义、版本、储存和执行。测试用例管理器（TCM）与缺陷管理系统有着许多相同的属性。这些共同的属性，以及测试用例和错误之间的关系，是我包括这两个主题在这一章的一个原因之一。另一个有趣的原因是，在微软使用最广泛的二个缺陷跟踪系统，即 Product Studio 和 Visual Studio Team System 软件，都是在同一系统中管理缺陷和测试用例。这样做好处是能够将测试用例，缺陷和功能区连接起来。TCM 的使用能使整个团队保持持续性和广泛性，因此属性，如易用性，可配置性和可靠性对 TCM 是必不可少的，就像缺陷跟踪系统一样。

什么是测试用例？

一个测试用例描述了针对某一特定的软件组件及预期的结果的具体操作。该组件可以是一个小的应用程序编程接口（API），一个用户界面（UI）的控制，或设备驱动程序的端口（port）处理。也可以是作为一个多台计算机和应用程序一起工作的大型软件系统。

测试用例可以由一组步骤和预期结果组成，如手工测试用例，或一组比如自动化测试用例的软件指令。自动化测试用例应该自我核查（即能够确定它们自己是否通过或失败）。图 9-5 显示一个简单的测试用例管理表的例子。

图 9-5 测试用例模板例子

测试用例编号: 0000
功能区： 测试用例简要标题

功能区	功能分区
-----	------

--	--

优先级	类别	频率	测试时间（分钟）
1	功能性	每次构建	2

描述
测试目的:
初始条件和背景:
步骤:
1.
2.
3.
4.
预期结果:
注:

大多数 TCM 系统基于 Web，独立的应用程序，或两者兼而有之。Visual Studio Team System 软件测试版包括 TCM，您可以使用它创建测试用例和报告。

图9-6显示一个开放的测试用例。

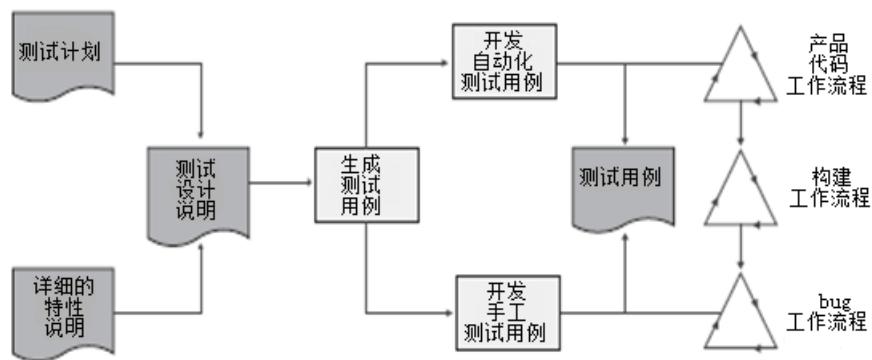


图 9-6 在 Visual Studio Team System 的测试用例管理

测试用例的价值

一个测试用例是一个正式的文件或记录，描述了测试活动是怎样具体执行的。一些测试参考资料指出测试用例的目的就是发现缺陷，但是测试用例的用处远远超出发现缺陷。测试用例可以验证程序功能正常或验证错误能被正确处理。测试用例的其他用处是可以尝试增加代码覆盖或专门地用于覆盖很少使用的路径。

测试用例文档的价值往往在微软和软件测试行业双方有争论。有几个因素对是否选择测试用例文档有帮助。测试用例文档的一些好处如下：

- **历史借鉴** 测试用例的存在要远远超过产品发布。持续工程（Sustained engineering），以及产品未来版本的负责人往往需要借用测试用例来了解测试过什么，以及是如何测试的。测试用例文档，以及一个有组织的储存系统，对长期支持或修订产品的一部分是至关重要的策略。
- **测试进展跟踪** 通过测试用例文档，您可以跟踪一些额外的属性，如测试用例的执行数目，测试用例的通过或失败数目，以及每个功能领域的测试用例总数。
- **可重复性** 好的测试用例文档可以由任何人在任何时候执行。这同样适用于自动和手动的测试用例。重复准确地执行同样的测试对重现步骤或检测回归是至关重要的。

测试用例文档也有缺点。其中包括以下内容：

- **建立文档的时间** 如果建立测试用例文档的时间比运行测试用例所需的时间还长，建立测试用例文档也许就没有意义了。经常有这样的情况，即测试用例只需要在一个单一的环境下执行寥寥几次。
- **功能变化引起测试用例过期** 建立测试用例所需的时间很可能因功能经常变化而增加以至于失去控制。如果测试用例的功能领域变化频繁，建立测试用例文档就不一定明智的。这种场景之一是尝试写测试用例以验证用户界面组件。
- **很难设想读者的知识** 写测试用例的人往往极为熟悉被测试的功能。这些人常犯的错误是在测试用例中使用术语或缩写，而将来运行测试用例的人很可能看不懂这些测试用例。出现这种情况出现时，测试用例已不再能准确地重复，测试用例也失去了这关键属性之一。

测试用例通常用测试用例管理器（TCM）来建立文档，微软大部分团队用测试用例管理器（TCM）记录绝大多数测试用例。重要的是要记住，测试用例并没有定义所有的测试活动。如缺陷大扫除（缺陷 bashes），那是整个团队致力于数小时或数天的时候，专注于使用功能或应用程序，寻找可能被测试用例错过的缺陷，这种测试活动在微软的各团队是常见的。许多团队也在产品周期中花时间致力于客户的使用场景。例如，一些Visual Studio的团队经常性地花一些时间，整团个队除了在Visual Studio开发环境创造和建立各种应用程序外，什么都不做。

剖析测试用例

有几个因素能区分好和差的测试用例。其中的一些因素以下：

- 目的** 找出为什么这个测试用例是重要的以及它要做什么。其目的可能是核查某具体功能，验证错误处理，核实具体情况，或其他特定目的。
- 条件** 确定哪些方面的环境是重要的或不重要的。明确测试是否需要运行在具体硬件或特定的操作系统，是否需要其他软件，或是否需要额外的先决条件。
- 具体的输入和步骤** 列出所需的一切步骤，以准确和反复运行测试用例。
- 预测结果** 让任何运行测试的人能正确的判断测试是否通过或失败。

其他的测试用例属性是指定运行地点和时间。这些属性如下：

•**测试频率** 通常称为测试类型，如版本验证测试（BVT），每夜测试（每天晚上运行的测试通过），或者一个里程碑测试，旨在内部迭代至少运行一次。一些测试用例管理系统用名字列出这些类型的测试，而另一些可能会指定一个优先级以说明测试运行的频率。

•**配置** 指出测试用例将运行在什么样的目标软件的配置。例如，一个测试用例的某属性可以说明，该测试用例只有在PC版的应用程序里运行，而另一测试用例则可在更多受支持的操作系统里运行。

•**自动化** 自动化属性指的是测试的自动化程度。换句话说，是否测试用例完全自动化了，部分自动化，或完全手动。一般自动化设置如下：

•**手动** 手动测试需要测试人员运行所有测试用例步骤，并在TCM里记录结果。

•**半自动化（Semi-automated）** 半自动化试验需要一定的手动操作。例如，网络测试可能需要一定的手动操作建立网络拓扑结构，但实际测试则完全自动化。

•**自动化** 自动化测试（有时也称为完全自动化测试）不需要用户干预。测试安装，执行，及结果报告到TCM完全自动化。

测试用例误区

创建好的测试用例是一个困难的过程。即使一个错误就可以毁掉测试用例的意图。一些易出问题的领域如下：

•**步骤缺乏** 匆忙建立的测试用例或假设测试用例的一些步骤会被执行而未将它们包括在测试用例里是非常常见的错误，它造成不能准确地重复，。

•**太多细节** 虽然提供具体和足够的信息很重要，不必要的字词或冗长的解释，会使测试用例难以遵循。仅需包含足够的信息以便精确地运行测试用例。

•**行话太多** 不要以为运行测试用例的人（包括产品技术支持和持续工程）都知道所有你写的缩略语，代号和缩写。阐明任何对整个产品生命周期有价值和必要的信息。

•**不明确的通过/失败标准** 如果运行的测试后，不清楚测试是否通过或失败，那测试用例是毫无用处的。

测试用例范例

差的测试用例例子

标题：检查加密邮件

步骤：

1. 获得 v3 的证书
2. 发送加密邮件

- | |
|---|
| <p>3。创建 0 RSA / DH 有锁箱子邮件，发送它
 4。创建 1 RSA / DH 有锁箱子邮件，发送它
 5。创建一个有 RSA / DSA 签名的邮件，发送它</p> <p>验证：</p> <ul style="list-style-type: none"> •所有邮件一切正常 •证书服务器产生了良好的证书 •纯文字和文字块正常通过 |
|---|

很好的测试用例例子

标题：有 v3 证书的纯文字

目的：

验证一个加密的使用 v3 证书的纯文字邮件可以被接受人打开和读取

条件/先决条件：

密码概况和 V3 证书（迪菲-赫尔曼 Diffie-Hellman）安装

用户 1 和用户 2 有 Microsoft Exchange Server 邮箱帐户

步骤：

1. 用户 1 启动 Microsoft Office Outlook。
2. 在工具菜单上，单击选项，然后单击安全选项卡。
3. 确定当发送签署过的文字邮件时，发送纯文字签署过邮件选项已选。
4. 确保 v3 的证书设置正确。
5. 单击 OK。
6. 打开一个新的邮件，给它一个主题“foo”和正文 “bar”
7. 填入用户 2 地址并发送邮件。
8. 在另一台计算机上，启动 Outlook 用户 2 帐户。

验证：

- 加密的电子邮件已经到达用户 2 的收件箱中，可以被打开
- 主题和正文中的信息分别是“foo”和“bar”，

注意：每一步都有编号。这是一个最佳做法。

管理测试用例

随着软件产品大小和复杂性的增加，其所需的测试用例的数量也随之迅速地增多。随着时间的推移，产品功能增多了、要支持的其他软件或硬件平台也更多了，需要运行和追踪的测试用例数量会是呈指数级增长。到达某种程度后，唯一能追踪的这种有庞大数目的测试用例的方式，就是通过一个全方位测试用例管理器（Test Case Management,简称TCM）。

TCM系统看起来和感觉上都很像缺陷跟踪系统。微软的主要两个缺陷跟踪系统（Product Studio 和 Visual Studio Team System）都包括了缺陷跟踪和测试用例管理的辅助工具、并且他们两者的很多功能间也共享许多用户界面和应用程序结构。这两者整合了的系统具有几个吸引人的特点。例如，发现的缺陷可以直接与测试用例连接起来，并且缺陷跟踪系统中，重现缺陷的步骤直接地从测试用例连接起来或可以容易地被直接导入进来。更进一步地，使用Visual Studio Team System（简称VSTS）系统还能直接地使测试用例与功能要求直接连

接起来。这种使功能需求、测试用例和产品缺陷的三项连接，有利于几种独特的看法，例如，某种需求条件下，能知道有多少个测试用例，知道哪些需求没有测试用例，和观察在缺陷和需求之间的对应关系。

测试用例和测试点：计算测试用例

测试用例是一系列特定的软件行为，是用于验证某特定功能、或能否正确处理某种出错行为，或其他软件质量衡量的属性（性能，可靠性等等）。一个测试播放音频文件的简单用例也许是 *打开 tada.wav 文件，点击“演播”，和验证它能正确地播放*。这是一个测试用例，但它也许需要在 32 位和 64 位系统都要执行一遍。也许还要求在相关的，但又不同的音频芯片组（chipsets）条件下执行一遍。取决于所需的执行特点，也许测试用例也需在各种各样的地方化的版本系统条件下执行几遍。这时候，许多测试人员可能搞不清，到底应把所有这些不同执行场景、条件的运行算作单一测试用例，还是多个测试用例。微软内部和外部的测试人员经常对这种情况有不同的理解。在微软，有些技术术语被更多地使用的实例之一就是测试点（*Test Point*）的概念。测试用例是为执行一个测试行为设定的一组步骤的一个单一实例（*instance*），而测试点是那个测试用例在某个特殊环境里的一个具体实现（*instantiation*）。众多测试点的结果可以与横跨测试矩阵（*testing matrix*）或以前已有记录的测试点互相进行比较，而此时这些测试点基于的测试用例也许根本就没改变。例如，当我在 Windows CE 团队时，我们在不同的硬件工作台和不同的配置操作系统运行了许多我们的测试。我们一百多万个测试点，有几十万个测试用例，但通过用测试用例和测试点区分测试，我们能迅速确定一个没有通过（*Pass*）测试用例是在所有运行环境都没通过呢，还是在某个或某些特殊的配置或平台下才会通不过。以下是几个有用的测试术语：

- 测试用例：为运行测试的一个计划。
- 测试点：一个测试用例和它的一种运行环境的组合。
- 测试套件（*Test suite*）：相关测试用例或测试点的一个集合。通常，一个测试套件是被测试功能的单位（*unit*），常限于产品的某组件（*component*）或特性（*feature*）。
- 测试运行（*Test run*）：被启动（*invoke*）使用的测试套件作为一个整体单位（*unit*）的集合。通常，一个测试运行是在单一的硬件或软件环境下，所有被启动的测试的一个集合。
- 测试通过（*Test pass*）：测试通过是一个测试运行的集合。通常，测试通过要涉及几种硬件和软件配置，并且受一个具体检查点（*checkpoint*）或软件发布计划的限制。

微软测试用例管理的解决方案

我在微软用的第一个测试用例管理工具是Microsoft Excel。事实上，我成为微软正式员工的第一天，我当时的经理就交给我一份刚刚打印出来的要我负责的测试用例。随着时间的进程，我们推广和修改了我们基于Excel工作表的系统。其实对我们来讲，这种旧式的基于Excel工作表的系统是可以满足我们当时的需要。但我还是认为这样的解决方案不是很有效率，也不适合。我怀疑有任何人能真正执行我们的测试用例（或能找到这些用例）。当我们把新的配置变化加入到我们的测试矩阵（*test matrix*），

我们一般都是把有关测试用例复制后，粘贴到新的Microsoft Excel工作表上。

荣幸地说，今天微软使用的 TCM 要好很多。但更精确地说，微软要面对一个不同的 TCM 系统的问题。这问题就是 TCM 数量好像爆炸似的增长。要在微软公司网上工具库中搜寻“TCM”，你会得到 22 个结果。我记得曾经有发生过的是 Windows 部门至少有 6 个不同的 TCM。大多数测试团队已经开始转向使用 Visual Studio Team System 里的 TCM，但几个主要的比如 Windows、Office，和 Developer 大部门，每个还都有自己的 TCM 系统。当然，在有着 8000 多测试人员的大公司，辅助工具的大幅度增长是不可避免的，但是同时有众多的测试管理系统也会造成其它问题。比如，如果 Windows 部门想要对新版本的视窗平台软件做与 Office 应用程序的兼容性测试。使用不标准化的 TCM 系统，使从一个 TCM 复制测试用例或执行测试用例到另一个 TCM 的难度也加大。

这种状况正在迅速地得到改进。现在几乎所有的 Windows 部门团队都使用同样的一个 TCM 系统，绝大多数新团队正选择使用主要的已有 TCM 系统或直接使用 Visual Studio Team System 来追踪测试用例和测试结果。

追踪和解释测试结果

一个 TCM 系统的主要作用之一就是能追踪相关的测试度量，比如已执行的测试用例和测试通过与失败的测试用例数目。一个 TCM 系统能把测试用例组织成为一个测试通过所需的范围，就是一个能每天、每周、每月按要求执行，或者针对某应用程序的某版本或版本类型的测试套件。当 TCM 系统能和缺陷追踪系统兼容的话，拿 TCM 系统就可以报告进行一个测试通过期间，发现缺陷的数量，或在缺陷数据库中，一个注明“已修复”缺陷是不是的确已修复了。

许多测试人员和测试经理先做的事总是检验有关的度量，然后尝试着去决定度量的含义是什么，这其实是方向性错误。当我们检验缺陷度量的时候，至关重要的是要先确定您试想答复哪些问题。以下的表 9-6 被列出了微软测试常用的度量和各自的用法。

表 9-6 测试常用的度量

度量	用法
通过比率	这大概是最常用的测试用例度量。该度量表示通过的与失败的测试用例数目的比率。计算这个比率常使用测试点而不是测试用例。
通过/失败的数目	在测试用例数目很大的系统里，该度量会有很明显的作用。例如，99% 通过比率通常是非常好（exceptional）的比率。微软产品的系统有百万个或更多测试点（测试用例*配置的数量）是很常见的。在一个有百万个测试点系统中，即使有 10000 个失败，通过比率仍为 99%。

测试用例的总数 / 计划的测试用例的总数 如果事先计划测试用例，并且在整个产品周期中实施，这度量可以为管理人员提供进展度量。

自动化比率 知道自动化测试相对手工测试的百分比是有用的。有些团队也许把增加测试自动化数量作为目标，这可能是针对一个特殊功能的，还可能是横跨整个产品团队。这个度量容易被滥用。不是所有的手工测试都应该或都能够实现自动化。关于自动化的测试类型更多的讨论请见第 11 章：“其他测试：非功能测试。”

测试类型的数目 测试经理可能想知道某个特定测试通过中哪些测试或多少测验被运行了。例如，版本验证测试（BVTs）需要迅速执行并尽可能在广度上覆盖产品。BVT 的数目与所测试的功能一起，能迅速提供关于该功能行为的信息。

发现缺陷的测试数目或百分比 该度量偶尔用在测量测试的有效性。它能显示是否不同种类的测试用例普遍都发现缺陷，还是仅仅某些测试用例或测试种类发现了多数缺陷。

本章小结

用最简单的术语来说，缺陷和测试用例是测试人员的中心世界。无论您是测试大型的、复杂系统或测试为卖旅行宠物辅助部件的网站，一个能允许您跟踪测试创作和执行测试的系统是极为有利的。好的系统是可以高度定制配置的，同时也很简单；它不仅为高技术的工程师也为所谓的市场专家或高层管理所用。并且，最重要的是，这些系统是测试工作的基石。

第 10 章 测试自动化

阿伦·培智

我写自动化程序已将近二十年了。当然，我写自动化测试的时间没有那么长。更确切地说，我在执行一些计算机操作时使用着可以反复执行的脚本或者代码，从而使冗余的任务更有效率，或者减少出错的可能。在早年使用计算机的时候，我使用批处理文件（批处理文件是一种脚本文件，包含着由命令解释器处理的命令序列）把文件备份到软盘上，或者重新配置应用程序的运行环境。现在的我仍然继续使用批处理文件为不同的编译系统配置命令行环境，但是我还使用其他的自动化方法。我书写和使用自动化脚本来安装应用程序，设置提醒，在不同计算机之间同步文件，以及调整我的计算机性能。我并不把这些看作自动化测试，但它们的确是自动化程序。这些自动化程序让我无做重复任务之虞，还确保了每次执行任务的步骤准确一致。

当然，好的自动化测试远远不只是准确地执行一个重复的操作。本章讨论高质量的自动化测试及自动化测试的架构和基础设施所应具备的要素、属性和结构。

读者在阅读本章（和本书其他章节）时要记住一点：测试自动化的策略、工具和技术在微软不同的地方会有很大的变化。本章的目标是讨论微软公司内较普及的一些自动化测试方法。尽管很多讨论到的自动化的办法是习惯做法，这个领域还在持续发展创新中，自动化工具和手段也一直在增长和扩展。

自动化的价值

似乎没有什么能够比对测试自动化的讨论更能联合或者分割整个业界的软件测试工程师了。对有些人来说，自动化测试盲目无情地替代了只有人脑才能完成的那类测试。对其他人来说，如果测试没有做到完全自动化就令人失望。然而在实践之中，具体环境决定了自动化的价值。有些时候，自动化每一个测试都是合理的。而其他时候，完全不用自动化测试却可能是合理的。有些类型的缺陷只有在人眼仔细盯着电脑屏幕上运行的程序时才会被发现。这些缺陷的描述往往是这样开始的：“奇怪——每当我关闭对话框的时候，整个屏幕都会闪一下”，或者“当我移动鼠标指针经过控件的时候，指针就会不停闪烁”。人类发现这类缺陷的能力远好过计算机。然而，对很多其他种类的缺陷，自动化测试则更有能力和效率。

自动化还是不自动化，就是这个问题

你为什么要写自动化测试？为什么该选择用人工测试而不是自动化？什么时候该做这样的选择呢？事实上，几乎所有的测试工程师早晚都要面对的问题就是是否选择自动化和决定自动化测试的程度。如果你只打算执行一次测试，根本没有必要自动化。可是，如果你打算测试两次呢？这也不意味着你应该使用自动化。有些软件在发布之前或者在维护阶段，可能需要执行上百次，上千次，甚至百万次的测试。有些因素有助于在具体环境下准确地评估自动化的益处。如下是其中几个需要考量的因素：

- **投入** 确定创建自动化测试的投资回报率（ROI--Return On Investment）的第一步是确定要花费的投入和成本。有些种类的产品或功能的自动化很简单，而其他的自动化却不可避免得很麻烦。例如，应用程序编程接口（API--Application

Programming Interface) 测试，以及别的通过编程对象的方式 展现给用户的功能之测试，对其自动化往往都能够直截了当。而在另一方面，用户界面（**UI--User Interface**）的自动化测试常会遇到问题而需要花费更多的精力。

- **测试的生命期** 一个自动化测试在变得无用之前将会运行多少次？对测试的长期价值的评估是决定是否对某个特定的场景或者测试用例实现自动化的考量的一部分。要考虑被测试的产品本身的使用寿命和产品开发周期长度。对于短期内就要发布而且将来不打算更新的产品，和对于两年后要发布将来也会有多个更新发布的产品，自动化的选择必须是不一样的。
- **价值** 要考虑自动化测试在其生命周期内的价值。有些测试人员说测试用例的价值是找到缺陷，但是很多自动化测试所找到的缺陷是在测试第一次运行时或者在写自动化测试时发现的。当缺陷被修复以后，这些测试成为了回归测试——这些测试用例的作用是确认最近的改动不会导致以前能够正常运行的功能停止工作。很多自动化测试技术通过改变测试用的数据，或者改变每次测试运行路径的方法，从而在测试的生命期中继续找到缺陷。对一个生命周期很长的产品来说，不断增长的回归测试套件有很大的优势：随着底层软件功能越来越复杂，存在大量的能确认以前工作的功能能够继续工作的测试是极为有用的。
- **切入点** 我目睹的许多成功的测试自动化项目都是测试团队从最开始的时候就参与了。代码写到尾声或者完成之后才开始想到加入自动化测试的项目通常都是失败的。
- **准确性** 好的自动化测试在每次运行后会报告准确结果。企业管理层对自动化测试最大的抱怨之一是自动化测试中误报的数量。（请阅读下面题为“**误报（Positively false）**”的侧栏。）误报是指测试报告中的测试失败是由测试本身的某些问题造成的，与产品无关。项目的有些领域（例如经常变化的用户界面组件）难以用自动化测试分析，且较容易产生误报。

误报（Positively false）？

当被测功能工作正常，而测试用例汇报验证失败的时候，我们通常称其为误报。测试用例报告一个错误（即测试为阳性，positive），但实际上并没有错误存在（即测试的阳性报告是假的，false）。反过来，当一个测试用例汇报验证通过，而被测试的软件功能其实有问题，我们称之为漏报（false negative 或 false pass）。举个在软件测试业之外的例子：法院冤枉一个无辜的人有罪，这就是误报；而法院判一个罪犯无罪，这就是漏报。这些术语来源于统计学。在统计学中，误报和漏报分别被称为一类（Type I）和二类（Type II）错误。

- **支持平台** 测试自动化是应对许多大型项目中测试矩阵爆炸的一个办法。写一个自动化测试用例一次，就可以运行在多个平台或者系统上，从而得到节约大量时间的优势。从小规模来说，我们希望一个自动化测试既能在微软Windows Server上运行，也能在Windows Vista操作系统上运行。但如果一个单独的自动化测试用例可以在八个不同的嵌入式平台或者十个不同的网络浏览器上运行，它就是一个获得巨大成功的测试。
- **复杂度** 与人工执行和验证的测试相比较，自动化测试有多复杂？如果依赖现有的自动化框架或技术，有些测试很难自动化。例如：图形程序可以使用像素比较算法来比较两个图像。图像比较在制造误报上的名声很坏。模糊匹配有时能提供更好的结果，但执行起来往往很复杂，而且需要不断更新自动化测试，以得到期望的结果。
- **其他因素** 你还必须考虑自动化所需要的时间以及写自动化测试的测试人员的技能。自动化是一种投资。它不是在项目的最后一分钟就能搞定，也不是一个没经验的测试人员仅仅凭在学校有过编程的经历就能做到的。记住你的目标就是在给定完成工作所需的时间和资源下，做能达到的最好的测试。还要记住，如果在产品早期的设计中，没有考虑到可测试性和自动化支持这两个因素，编写自动化就会更加困难。最后一个要考虑的因素是管理层对自动化的指导意见。自动化绝不是灵丹妙药，但也不能完全忽视。有些组想什么都用自动化测试，而另外有些组可能很少会用到自动化。自动化测试的正确用量没有一个确定的说法。在决定何时用自动化和用多少的时候，需要注意考虑到上述的所有因素，而不是纯粹根据管理层的指导意见来作出决定。

我们没时间把测试自动化

我用一个叫 Microsoft Test 的程序学会了写用户界面自动化。这个常被称为 MS Test 的程序（后来命名为 MS Visual Test，之后又被出售给了 Rational Software 公司）是一个写用户界面自动化的工具。大概在我去微软工作一年以前，我开始使用这个程序的二期测试版。等到在微软开始工作的时候，我已经对这个程序相当熟悉，用它写用户界面自动化得心应手。我在微软的第一个小组负责测试日文、中文和韩文 Windows 95 下的网络功能。我的主要工作就是核查这些语种的字符在网络属性的用户界面的各部分显示正常，以及包含这些字符的文件名在不同的网络拓扑结构和操作系统下的互操作性。那时候，我对测试非英文版本的 Windows 操作系统所知甚少，更别提 Windows 95 了。我知道写这个领域的自动化是个很有

趣的挑战，我跃跃欲试。

工作的头一两天，我熟悉了一下我的办公室环境，找到测试实验室，跟同事见面，设置电脑，还找到在内部网络上的必备工具。我的主管让我去找他谈谈我的测试职责。我迫不及待想要开始工作。我们简单地谈到了我负责的工作职责，确定我明白自己的测试范围后，他给了我用微软 Office Excel 工作单打印出来的一系列测试用例。我花了几分钟大略看了一下，确定都懂。所有的测试用例都很合理，我自信我很快就能把这些测试用例自动化。

谨慎起见，我特地问了一下我什么时候必须完成自动化。他的回答我至今还记得一字不差：

“唔，不需要，我们没时间把这些测试自动化。你只需要在每个新版本出来的时候，把它们运行一遍。”

当时我就想：“哇！自动化这些测试肯定比我想象的要难多了。”我回到办公室开始运行这些测试用例。这样做了几天，我发现了几个缺陷，也对这个领域了解得更多了一些。有一些测试用例是关于命令行网络选项的。我开始对执行这些测试用例感觉有点枯燥无聊了。结果呢？我发现我有时会忘记一些步骤。我想，“不如用一点批处理文件吧，那样运行这些测试就更一致了。”十五分钟以后，我证实了自己是对的。我还发现了一个办法能够自动化测试更多的配置操作。

几天后，在作用户界面测试的时候，我又犯了类似的错误。于是我想花几分钟时间试试自动化真的会有多难。我打算好了周末去加班，尝试在 Windows 95 上写用户界面自动化测试。可我很快发现，Visual Test 能非常好地自动化我所需要执行的测试。几个小时后，经理给我列的工作表上大部分测试都变成自动化的了。

几个月内，我每天都运行这些测试，它们帮我省下了数天甚至数周的时间。我用这些时间扩大测试的范畴，并查看那些否则我永远都不会去看的方面。如果我每天一直专注于同样的测试脚本，我可能会错过至少一百多个缺陷。回过头来看，我当时还是一名测试新手，我写的那些测试用例可能不是很好，但无论怎样看，在当时的环境下，自动化仍然是正确的答案。

用户界面自动化

面向公众的函数如 Windows API 或任何通过编程界面而面向公众的功能，都很适合做自动化测试。你不一定非用自动化就可以测试编程界面。例如，仅通过运行微软 Office 的套件我就可以测试大部分的核心 Windows API。但通过针对单个的 API 功能的自动化测试，我可以使用测试应用程序有效测试多个参数的组合。

很多非功能测试都适合自动化，如性能测试、负载测试、压力测试和泄漏测试。（详见第 11 章，“非功能测试”。）实际上，自动化测试是执行很多相关情景测试的唯一方法。模拟上千个同时对一个服务的连接，以及为某个操作计时的任务，自动化都是唯一切实可用的解决方案。

提到测试自动化，很多软件测试人员首先想到的是用户界面（UI）测试。软件测试市场提供了很多工具，辅助测试人员在写自动化测试中操作用户界面。相当数量的工具还提供了录制和回放功能：测试人员可以简单地录下人工测试的过程，然后以自动化测试的方式回放。录制和回放的工具常常受到批评，因为它们产生的测试不能抵御用户界面的微小变化。这是用户界面测试中普遍存在的问题，而且许多用户界面的元素的自动化相当困难。

在微软的实践中，用于界面自动化的主要方法是绕过演示层，直接使用底层的对象模型，或者用相似的方法来操纵用户界面的核心逻辑。在有的情况下，用户界面自动化会通过模拟鼠标点击和击键直接与 UI 互动。

```
//以下的C#代码启动Microsoft Office Word，输入文本，选择文本，并将字体加粗
```

```
//为简洁计，删去了错误检查
Process wordApp = Process.Start("Winword.exe");
if (wordApp.WaitForInputIdle(1000))
{
    SendKeys.SendWait("This is input to WinWord");
    SendKeys.SendWait("^a"); //发出Ctrl+a (选择全部文本)
    SendKeys.SendWait("^b"); //发出Ctrl+b (将字体加粗)
}
```

使用击键和鼠标点击写成的代码是重现用户与软件互动行为最相近的自动化测试方法，但也是最不稳定的 UI 自动化测试方法之一。控件会移动，控件的标识符会改变，文本也会更新或者本地化。仅仅使用模拟击键或鼠标点击的方法写出健壮的自动化代码不是不可能，但很困难，也远远不能防止错误的发生。

另一种自动化的方式是自动化那些当用户与用户界面互动时发生的行为。例如，自动化测试不模拟用户点击按钮，而直接调用用户点击按钮会触发的代码。

图 10-1 是对面向对象社区称之为对象模型做简单变化后得到的。对象模型是操作应用程序特定部分的对象（功能）的集合。例如：HTML 文档对象模型（DOM）提供了对 HTML 控件（按钮、复选框等等）、嵌入链接和浏览器历史的访问。有些语言，比如 JavaScript，可以通过 DOM 直接访问这些控件。

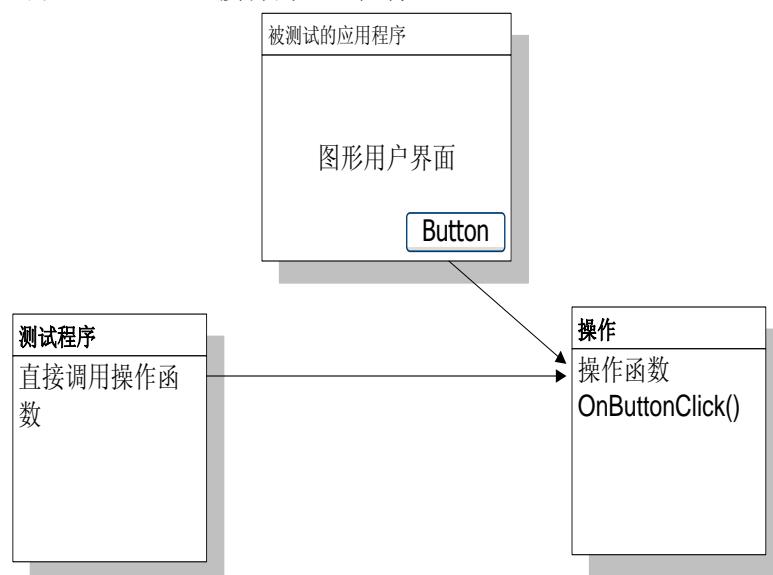


图 10-1 不通过 UI 调用 UI 操作函数

很多 Windows 上运行的应用程序提供对象模型以访问程序的功能。通过使用对象模型，加上用户界面代码（这些代码将程序逻辑从用户界面逻辑中分离），测试人员能够使自动化测试无需直接和用户界面控件互动就能操纵用户界面的任一部分。

这个代码通过使用 Word 的对象模型来执行前面例子中的自动化。

```
Object template = Type.Missing;
Object newTemplate = Type.Missing;
Object docType = Type.Missing;
Object visible = Type.Missing;
```

```

Word.Application wordApp = new Word.Application();
Word.Document wordDoc = new Word.Document();
wordApp.Visible = true;

//以下的一行代码等同于先选择“文件”菜单，再选中“新建”菜单项来创建一个基于Normal.dot模板的新文件。
wordDoc = wordApp.Documents.Add(ref template, ref newTemplate, ref docType, ref
visible);

wordDoc.Selection.TypeText("This is input to WinWord");
wordDoc.Selection.WholeStory();
wordDoc.Selection.Font.Bold = 1

```

此代码通过对象模型来操作 Word，而没有使用鼠标点击或者击键。这比使用 SendKeys() 的 C# 例子繁琐些，但是容易扩展和维护。

使用 Microsoft Active Accessibility (MSAA) 是另外一个类似的自动化用户界面的办法。MSAA 改善了辅助工具如屏幕阅读器在 Windows 上工作的方法，同时也提供了一个写自动化测试的简便方法。MSAA 的核心是 IAccessible 接口。IAccessible 接口支持的属性能让你得到相应用户界面元素的信息。Windows 的通用控件，如按钮、文本框、列表框和滚动条都实现了 IAccessible 接口。许多 Windows 应用程序所包含的自定义控件也支持这一接口。

应用程序（或者测试）通过诸如 AccessibleObjectFromWindow、AccessibleObjectFromPoint 或 AccessibleObjectFromEvent 函数得到一个指向 IAccessible 接口的指针，从而使用辅助功能函数。通过使用接口指针，测试程式使用函数获取关于控件的信息（如文本或按钮状态），或者操纵控件（如模拟点击一个按钮）。

随着 Microsoft .NET 框架 3.0 版本的发布，Microsoft UI Automation 成为 Windows 上新的辅助功能框架，并在所有支持 Windows Presentation Foundation (WPF) 的操作系统上都可用。与 MSAA 类似，UI Automation 提供辅助功能，但是用托管代码实现并且最容易在 C# 或 VB.NET 应用程序下使用。UI Automation 将用户界面的每个组件以 AutomationElement 元素展现出来。AutomationElement 元素又显示了它们所代表的 UI 元素的常用属性，如外观和状态。类似于点击按钮的操作，可以用如下代码实现。

```

// 包含一个InvokePattern对象，然后用它来点击一个按钮
// 注释：为了简洁，略去错误检查
private void InvokeControl(AutomationElement targetControl)
{
    InvokePattern invokePattern =
        targetControl.GetCurrentPattern(InvokePattern.Pattern) as InvokePattern
}
invokePattern.Invoke();

```

微软内部写用户界面自动化测试的大部分团队使用上述这些方法或这些方法的组合。很多用户界面自动化框架包含了一个或多个提到的解决方案。例如，一个应用程序可能有一个不完全的对象模型，所以我们可以对大部分测试使用对象模型，而对余下的用户界面测试使用 MSAA。

使用蛮力的用户界面自动化测试

在大多数情况下，通过模型或者类似的方法来访问控件的 UI 自动化，与通过点击按钮和击键来访问 UI 的自动化是一样的。但有些时候，纯粹通过模型的自动化可能会错过重要的缺陷。

多年以前，一个从 Windows CE 分离出来的小组在做一个叫作 Windows Powered Smart Display 项目。这个装置是一个平面显示器，同时还作为终端服务的瘦客户端。当显示器脱离工作站的时候，它将通过终端服务器客户端继续连到工作站。

该设备的软件主要是由 Windows CE 的核心组件写成，但还包含一个简单的用户界面，用来显示连接过的服务器历史、电池寿命、连接速度和任何本地的应用程序。CE 的组件都已经被认真测试过，指派过来测试该设备的测试小组在一些手工的功能测试之外还做了一些用户场景测试。产品开发快要结束的时候，我花了一些时间和测试小组一起工作，帮助他们开发一些额外的测试用例。

我首先要做的事情之一就是创建几个可以连夜运行的简单测试，以发现普通用户在数天甚至数周内可能都不会遇到的问题。这个程序没有对象模型，也没有其他可测试功能。但因为程序很简单，我就有时间开始做我称之为使用蛮力的用户界面自动化。这个应用程序只有一屏用户界面，我很快就写了些代码以找到该屏幕上的每一个窗口。我记得我本来要查找那个程序使用的特定的 Windows 消息的，但这需要我等待源代码的访问权限。我从来都不喜欢干等，于是我决定写个能将鼠标置于屏幕上某个指定窗口中心并点击鼠标的代码。经过几分钟的调试和测试，我写出一个简单的应用程序，它能随机连接到任何现有的服务器上，验证连接成功建立，然后终止该终端服务器会话。

于是在结束当天工作回家之前，我设置该程序做无休止的运行，启动，运行。与此同时我去办最后几件事情。偶尔我会扫一眼那个程序，很高兴地看到程序每几秒就做一次连接再断开的动作。可是，当我站起来准备离开之时，转身看了我的测试程序最后一眼，我看到程序已经崩溃了。我碰巧是在调试器下运行，并注意到崩溃是由于内存泄漏导致的，而且程序用完了一种特定的 Windows 资源。开始，我以为是我的程序的问题，所以我花了点时间搜索源代码，寻找使用这类资源或者可能滥用 Windows API 的地方。我找不到任何错误，但仍然认为问题肯定出在我这里，也许是早些时候的调试会话中造成了问题。我重新启动设备，设置测试再次运行，然后走出了办公室门。

第二天早上到了公司，我一眼就看到程序在同一个地方崩溃了。到这个时候，我已经有了应用程序源代码的访问权限。大概花了一个小时的调试，问题证明不在连接代码上。每当一个计算机名被选中，该程序就会初始化定制绘画的代码。定制绘画只是一个小小的蓝色闪光，使用户可以知道程序已接受鼠标点击，就像在一个基于 Windows 的应用程序中，点下按钮，按钮就会显得下沉。这个程序的问题是每次测试和绘图代码运行时，发生了资源泄漏。经过几百次连接，资源泄漏就足够大了。当定制绘画代码运行时，程式就会崩溃。

如果我总是写不直接通过用户界面而执行功能的用户界面自动化，我认为我根本不可能发现这个缺陷。我想在大多数情况下，写出健壮的用户界面自动化的最佳方法是不和用户界面交互而直接访问控件。但从这件事以后，我总是会留意那些属于用户界面但要比其代表的功能做得多的代码。

自动化测试中包括什么？

在考虑自动化测试用例时，不仅仅要考虑测试的执行步骤。在运行任何步骤之前，程序必须处在能够执行测试的状态。在测试执行后，至关重要的是知道测试是否通过，并且测试结果一定被保存到某处以待检查或进一步分析。另外，还可能需要清理测试中生成的垃圾（文件、注册表设置等等）。最后，测试一定要易于维护和易于理解，以便任何人在任何需要的时间都能运行或修改。

自动化测试的含义远比自动执行原来人工执行的测试步骤多得多。好的自动化测试利用计算机的能力去执行人力无法有效执行的测试。自动化测试不是测试工程师的替代品。能思考的人执行测试的作用远远超过机器。但是如果有效地利用，自动化测试会节省大量的时间和开支。

Keith Stobie 和 Mark Bergman 在他们 1992 年的论文 “How to Automate Testing: The Big Picture” 中用缩写 “SEARCH” 来描述测试自动化的组成部分³⁸。“SEARCH” 代表的是设置（Setup）、执行（Execution）、分析（Analysis）、报告（Reporting）、清理（Cleanup）和帮助（Help）。

- **设置** 设置指的是将软件准备好，让实际的测试操作可以开始执行。
- **执行** 这是测试的核心，包括检验软件功能的特定步骤，充分的错误处理，或者一些其他的相关工作。
- **分析** 分析是确定测试通过还是失败的过程。这是最重要的步骤，也常常是测试中最复杂的步骤。
- **报告** 报告包括分析结果的显示和传播，例如日志文件、数据库、或者其他分析过程所生成的文件。
- **清理** 清理阶段将软件返回到已知状态使得接下来的测试能继续执行。
- **帮助** 指的是使测试用例在其生存周期中保持可维护性和健壮性的帮助系统。

考虑以下的简单测试用例：

标题

验证计算器程序（Calculator）能够显示大于 32 位的整数

步骤

1. 在十进制模式下，用键盘或计算器程序的控件输入最大的 32 位无符号整数（4294967295）。
2. 给现有值加上任意正整数。

验证

验证加法操作是否正确。例如， $4294967295 + 10 == 4294967305$

这个测试用例描述了确定一个计算器程序能否处理比 2^{32} （最大的 32 位整数）大的数值所需的操作。³⁹

³⁸ Keith Stobie 和 Mark Bergman, “How to Automate Testing: The Big Picture”, 1992 年 3 月,
http://keithstobie.net/Documents/TestAuto_The_BigPict.PDF

³⁹ 没有考虑到大于 2^{32} 的值的程序会“越界”（溢出）到该数据类型里最小的数值。如果是这样，4294967295 之后的数字变成零。 2^{32} 的值是 4294967296。因为计算机从零开始计数，无符号 32 位整数的范围就是从 0 到 4294967295。

表面上看，这个测试很简单。但实际上自动化这个测试所需要的步骤要比上面列出的两个多得多。在执行测试步骤之前，需要先运行计算器程序，而且其状态应该可以接受输入。即使在最快的计算机上，有些应用程序也要花几秒钟的时间启动。所以自动化测试需要把这一因素考虑在内。另一种做法是自动化测试假设被测程序已经启动。无论这里的假设是什么，在执行测试用例的第一步之前，某些条件必须要先满足。

完成上述步骤之后，自动化测试需要检验输出结果以判断测试通过与否。在本例中，自动化测试可以从计算器的输出域中读取结果，也可以验证程序内部变量的值（或双管齐下）。决定测试是否通过以后，自动化测试需要把测试结果记录下来，以后不论谁需要评估测试状态，都可以查看测试结果。如果后续的测试用例假设计算器的输出域为空或清了零，或假设计算器程序没有运行，自动化测试就需要将计算器清零或在自动化测试结束前把程序关闭。

最后，为了测试的长期维护，也为了方便其他（或以后的）团队成员修改或添加测试，测试用例应包含（或测试代码中应集成）关于测试的附加信息。

开发自动化测试时，考虑从设置到维护的整个测试构造是很重要的。自动化程度很高的自动化测试可以把 **SEARCH** 的每个步骤都自动化，但有时只把部分步骤自动化就很有用。例如有些情况下，可以将程序的安装和设置自动化，之后执行关键的、或探索性质的人工测试。或者，你可以采用某个系统把测试结果记录到测试用例管理器，作测试结果的自动分析和报告。大多数情况下，一个成功的方法是自动化大部分的步骤。

很多测试自动化的努力遭遇失败，其原因在于它们只自动化测试执行的环节。一个完整的自动化方案要求自动化不仅仅测试执行这一个环节。在一个自动化策略中，如果没有一个把应用程序准备好到测试可以执行状态的计划，也没有自动的测试结果报告和分析，就很少能带来什么益处。许多人在想到测试自动化时主要考虑测试执行的自动化，其实其他测试阶段的自动化也很有用。用“计算机辅助测试”这个术语来描述对测试各个阶段自动化的概念，可能要比“测试自动化”这个术语更合适。彻底的自动化绝不仅仅是执行一个测试。执行测试需要各种支持性任务，在计算机上开发的工具和实用程序提供了自动化这些任务的好解决方案。

测试工具和实用程序

有些情况下，自动化测试不能作为有效的测试方案。我在 Windows 98 上的一个测试内容是字体绘制。有些方面的字体测试能极大地得益于自动化。性能测试就是一个很好的例子。我们对核心图形引擎显示非西文字符的方法做了重要改动。我的任务是测试不同字体绘制不同长度的随机字符串的时间。因为组合太多，人工测试是不可能的，但我写了一组测试套件能准确地跟踪性能改善的影响。

字体测试的其他一些方面没法自动化，但我仍需要快速、准确的办法确定多重字体设置产生的效果。对很多的测试，我用了一个工具把字体显示在大的网格中，以观察字体的每一个像素。这个工具还可以快速地调整各种影响字体显示的参数。例如，如果有客户报告某种字体的一些字符看起来很奇怪，我可以很快检查字体显示，并确定问题的来源。

我还需要确认一种字体里所有的字符显示都没有问题。那些日子里，在有些颇让人费解的情况下，应用程序在试图绘制某个特别的字符时崩溃，或者在打印某个范围的字符时有问题。当然，对每种字体中每个字符做打印测试很费时间（也费树木），于是，抽查新字体和研究客户所报告的缺陷就变得很重要。为帮助我自己应对这种局面，我写了一个工具可以任一种大小和属性（粗体，斜体，等等）显示或打印任一字体中的所有字符，如图 10-2 所示。我只花了几个小时写这个工具，但第一次使用就发现一个第三方提供的字体在滚动显示字符时会引起程序崩溃。我和小组的其他成员在整个开发周期中经常用这个工具作快速验证，冒烟测试，以及其他调查。我们用这个工具只找到了几个其他缺陷，但这个工具在验证各种字体绘制问题方面的作用却非比寻常。这并不是自动化的测试，但它为我们几个人节省了数十个小时的时间。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
340	止	丙	彘	今	牛	乂	舅	芸	会	玄	吉	夬	𠂔	𠂔	𠂔	𠂔
341	努	彖	䷤	䷤	䷤	䷤	䷤	䷤	䷤	䷤	䷤	䷤	䷤	䷤	䷤	䷤
342	𠂔	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵
343	𠂔	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵	𣎵
344	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔
345	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔
346	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔
347	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔
348	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔
349	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔
34A	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔
34B	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔
34C	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔
34D	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔
34E	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔
34F	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔	𠂔

图 10-2 字体显示工具

微软中的“SEARCH”

微软内部有许多不同的实现测试自动化的途径和方案。并非所有的团队都以“SEARCH”作为自动化以及测试用例生成的方法，但在大多数团队中盛行的方案都会考虑自动化测试的各个阶段。

设置

微软的团队的测试范围之广和测试矩阵的规模之大，突显了对测试设置阶段进行自动化的需要。对这个阶段进行自动化的最简单形式，就是使用脚本或者命令行选项实现应用程序的无人值守安装。通常情况下，测试机在半夜就自动安装好了被测程序的最新版本。测试工程师们早晨来上班以后，就可以开始测试这个全新安装的程序了。这种方式能够节省时间。然而，假如产品的安装很少按照消费者的实际安装设置方式来测试的话，那么这种方式还是存在危险的。倘若所有的安装过程都是自动进行的，测试团队将毫无疑问会漏掉一些在安装设置程序方面的重要缺陷。对此，在微软一种普遍使用的方法是：多数测试团队会继续使用自动化的程序安装。然而，还会有一名专职的安装设置测试工程师（或者测试团队）来专门负责

验证整个安装设置测试矩阵。

当你考虑了操作系统版本和待测程序版本的覆盖要求时,为执行测试所要准备系统的范围就可能会迅速扩大,其中包括安装操作系统(如果有必要的话)和配置待测程序。考虑一下表 10-1 中所示的测试矩阵。

表 10-1 安装设置测试矩阵

基础操作系统	所需测试
Windows XP	完全安装
Windows XP	从版本 1 升级到版本 2
Windows Vista	完全安装
Windows Vista	从版本 1 升级到版本 2
Windows XP	安装应用程序, 然后升级到 Windows Vista

安装和升级的情景是极其复杂的。当测试的基础配置包括安装操作系统,或者前一版本的应用程序时,对测试的设置立刻成为巨大的耗时负担。在这些情况下,非常有必要加速完成基础配置。通常情况下,需要花费数个小时才能手动安装一个操作系统并配置好必要的应用程序。对此,微软的许多团队利用一种使用了镜像(imaging)技术的系统来实现快速基线安装。镜像技术给基线系统拍一张快照并直接写到计算机的硬盘。如果一位测试员需要一份安装了 Office Professional Edition 2003 的 Windows XP 系统的拷贝,她可以直接选择那个镜像,然后在她想要配置的计算机上运行一个脚本或应用程序。几分钟以后,一台安装了需要的程序的待测计算机就准备好了。制作镜像通常分为两个阶段:一是创建必需的软件的基线;二是安装特定于被镜像计算机的驱动程序并配置起必需的用户帐户。

使用这种方法的团队们在手头上握有数百个镜像,并使用这些镜像来配置多种多样的手动和自动化测试。这样的镜像库的另外一个优点是可以集中给镜像中的应用程序或操作系统打补丁,这会减少配置应用程序时的另外一项耗时的部分。

Windows CE 操作系统的安装使用了一种类似的方法。Windows CE 的设计没有采用应用程序或操作系统那样的安装方式。它的系统镜像是被刷进一个诸如移动电话、网络路由器或者支持 Windows CE 的开发主板这样的设备上的。Windows CE 自动化测试系统,能够按照指定的测试和硬件配置自动选择适当的操作系统镜像文件,将操作系统刷进设备,并在执行测试前,使用特定的硬件自动重置(重启)设备。

我没想到你能够自动化那个

有时候,你的某些测试工作看起来很难或者几乎不可能被自动化。我认为非常重要的是,无论何时遇到这些看上去不能自动化的测试,要多集思广益去找找看是否存在途径,可以让其中哪怕是最小的一部分测试变得更有效率。

Windows 和 Windows CE 团队使用多种定制的硬件来协助他们的自动化工作。例如,测试 PCMCIA 卡是一项耗时的工作。操作系统需要支持在任何时间插卡和拔卡。一些类型的测试可能需要运行在不同的 PCMCIA 网卡上,比如网络测试等。在 Windows CE 团队,我们使用了一种叫“PCMCIA 点唱机”的设备来协助测试。这种设备包括六个 PCMCIA 插槽和一个串行连接。从六个插槽中的任何一个,我们都可以用这个串行连接来模拟插卡或拔卡的动作。虽然不是每种外围设备都有其相对应的“点唱机”,但在没有“点唱机”的地方,我们有一些“转换器”去模拟插入和拔出其他使用串行连接的设备,诸如使用 USB 或 1394/火线(Firewire)连接的可移动设备。

Windows 团队非常依赖设备模拟框架来模拟硬件。有了这种框架,一些本来不切实际的测试变得可行了,例如插上数百个设备,或者成千次插拔一个设备。

执行

执行测试用例各步骤是自动化测试的核心，而且执行方法是多种多样的。一种简单的执行形式是编写并运行一段脚本或程序。如下所示的是一个用 Microsoft VBScript (Microsoft Visual Basic Scripting Edition) 编写的简单的测试用例。这个例子打开并且打印指定的 Microsoft Word 文档。

```
Set objWord = CreateObject("Word.Application")
Set objDoc = objWord.Documents.Open("c:\tests\printtest.doc")

objDoc.PrintOut()
objWord.Quit
```

测试的执行也常会以独立程序的形式存在。我写过的一个应用程序包括很大一套针对 Windows 98 的网络测试。在启动这个程序并配置一些选项以后，点击一个按钮就可以触发一大批网络文件共享和拷贝的功能。这并不是一个最优的方案。尽管这些测试本身是自动运行的，但是仍旧需要用户干预来选择和执行这些测试。

在今天的微软，一种更好的几乎被每个团队都采用的解决方案是：使用测试用具(test harness)来运行自动化测试。测试用具就是运行测试所需要的框架。好的测试用具是可配置的、可扩展的，并且能够使自动化测试更加容易。如图 10-3 所示的是一个简单的测试用具结构的例子。

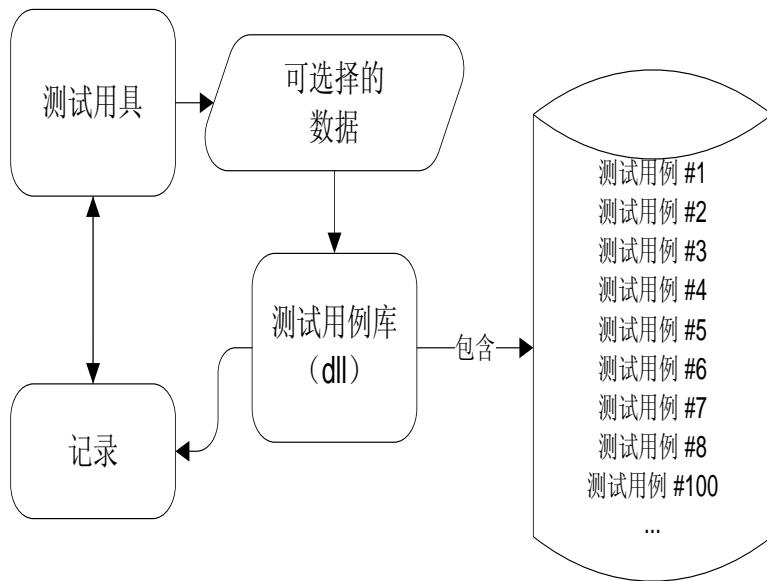


图 10-3 测试用具的设计

在这个例子中，测试用具是一个应用程序，其运行的测试包含在另一个二进制文件中，例如 Windows 动态链接库(DLL)。这个测试用具的工作流程大体如下：

1. 测试用具启动并检查任何传给它的附加数据。这可能包括环境变量、要运行的测试、文件位置、网络地址或者测试需要的其他任何数据。传给测试用具的信息总汇包括含有测试用例的文件名。例如，一个命令行会运行 TestCase.dll 中的测试用例 1 到测试用例 100，并且重复执行 10 次这 100 个测试用例，同时将日

志信息保存到一个文件名为 test.log 的文件。这个命令行看起来是这样的：

```
harness.exe TestCase.dll -Tests 1-100 -Repeat 10 -output test.log
```

2. 测试用具执行测试用例。取决于测试用具或者可选数据，自动化测试用例可能依次运行，也可能随机运行，或按特定的顺序运行。当测试用例是由本机 C 或 C++ 代码实现时，测试用具常用的实现方式是使用一个包含了每个测试所需函数的动态链接库。典型情况下，动态链接库使用头文件中的一张表来跟踪详细的测试信息和每个测试的唯一标示，如下所示。

```
struct functionTable[] =  
{  
    { "API 1 的功能性正测试", 1, FunctionalTest },  
    { "API 1 的边界测试", 2, BoundaryTest },  
    // ...  
};
```

这个例子中的表包括了测试用具使用到的三个项：一个纯文本的测试描述（例如，“功能性正测试”），一个唯一标识符和执行测试的函数的地址。这样就可以运行测试 2 而不必知道（或记住）测试 2 是一个特定 API 的边界测试。

使用这种结构的测试用具工作的时候，首先用 `LoadLibrary` 函数加载动态链接库，然后把函数表中的地址传递到 `GetProcAddress` 函数，进而调用动态链接库中的测试用例。

在这样的测试用具的体系结构中，测试代码看起来像是典型的应用程序中的函数。

```
int FunctionalTest(int parameters)  
{  
    int testResult = 0;  
    //编写测试代码  
  
    if(testWasSuccessful)  
        testResult = 1;  
    return testPass;  
}
```

这种测试用具在微软的自动化测试工作中被普遍使用，效率高而且易于扩展。

在微软，越来越多的自动化使用 C#（读作“cee sharp”）来编写。一般来说，用 C# 开发要比用本机 C 或 C++ 代码开发更快。典型说来，用托管代码开发的测试（或应用程序）比用本机代码开发使用的代码更少。这使得阅读、检查和维护代码更加容易。另外，托管代码也消除了本机代码所可能引起的许多缺陷。例如，变量初始化得以保证，而内存管理是自动的。

当编写测试用例时，测试的细节可以以函数属性（attribute）的形式存在。比起创建在头文件里一张函数表，属性显得更为灵活。使用属性把分类的测试信息写在测试函数源代码前面而不是在一个分开的文件中，让测试信息容易更新。

```
[TestCaseAttribute( "版本验证测试：数学测试", Type = TestType.BVT, ID=42)]  
public TestResult BuildVerificationTest_Math1()
```

```

{
    TestResult.AddComment("执行 BuildVerificationTest_Math1");
    //删去的测试代码...
    return TestResult.Pass;
}

```

这个例子使用了一个定制的属性来描述测试和测试类型，并给测试指定了一个唯一的标识符。测试用具使用反射来检测属性的细节然后执行包含在托管代码中的测试函数。一个测试用具的命令行用法示例如下：

managed-harness.exe managed-test.dll TestType=BVT

使用这个命令行的测试用具使用反射检测 *managed-test.dll* 并运行所有附带 *TestType.BVT* 属性的测试。

Microsoft Visual Studio Team System 里的单元测试框架包含了一些内置的属性，这些属性既适用于单元测试，也适用于功能性测试。托管代码编写的测试用具经常添加一些附加的属性来实现更完整的说明和测试用例分类。表 10-2 包含了一系列常见的托管代码测试属性。

表 10-2 常见的托管代码测试属性

属性	描述
<i>ClassSetup</i>	附带此属性的函数包含执行任何测试之前的行为。 附带此属性的函数可以启动应用程序、设置数据库的值或者设置所有测试之前所需的配置。
<i>ClassTeardown</i>	拆卸行为发生在所有测试运行之后。它们将环境恢复到初始状态。 附带此属性的函数用来移除测试中所有的遗留物，例如文件、数据项或者注册表项。
<i>TestInitialize</i>	附带此属性的函数在每条测试之前运行，包括了为准备测试环境做任何需要的事情，例如拷贝或创建测试需要的文件。这减少了在每个测试函数执行之前可能的重复代码，但其前提是每条测试都需要从同一点开始执行。
<i>TestCleanup</i>	附带此属性的函数在每条测试之后运行。这些函数可能删除测试中创建的文件、恢复一个数据库到已知设置或者恢复其他的系统设置。
<i>TestMethod</i>	附带此属性的函数代表包含在文件中的测试。
<i>Step</i>	这个属性被用来指示函数必须按照特定的次序执行。一个典型的托管代码编写的测试用具能够按照任意顺序来执行测试。如果特定的测试需要按照特定的次序运行，那么就用 <i>Step</i> 这个属性指示这一点。
<i>SupportFile</i>	这个属性指示运行测试所需要的特定的文件。

```

// 这些测试函数总是以如下顺序执行

[TestCaseAttribute("有序测试示例", Step=1, SupportFile="one.txt")]
public TestResult TestOne()
{
    // 使用"one.txt"
}

```

```

    ...
}

[TestCaseAttribute("有序测试示例", Step=2, SupportFile="two.txt")]
public TestResult TestTwo()
{
    // 使用"two.txt"
    ...
}

```

这个例子通过 Step 这个属性指示测试用具：TestOne 总是在执行 TestTwo 之前执行。与此同时，TestOne 和 TestTwo 需要附加的支持文件存在（分别是“one.txt”和“two.txt”）。

3. 测试用例可以记录测试数据（包括测试状态）并输出到一个文件、一个调试流或者别的固定的位置。测试用具检查测试日志或者其他信息以决定测试是否通过。测试日志可以是简单的文本文件、基于 XML 的文件、HTML 文件、三者的组合、或者完全不同的输出诸如 Windows 系统事件、或者数据库里的记录项。对于追踪测试结果和调试测试中的失败，日志文件是必不可少的。表 10-3 列出了自动化测试中每个日志文件里都应该有的几个元素。

表 10-3 推荐的测试日志所必备的元素

元素	注解
测试标识符 (Test ID)	系统的每项测试需要一个唯一的标识符，即使把结果汇总到顶层的报告也不会有标识符冲突。全局统一标识符 (GUID) 经常被用作测试的标识符。
测试名称 (Test Name)	一个容易理解的测试名称。
环境信息 (Environment Information)	包括操作系统的版本、SKU (例如, Windows Vista 旗舰版)、操作系统的语言、体系结构 (x86、32 位、64 位等等)、总共的内存、剩余磁盘空间和计算机名。环境信息有助于追踪那些只发生在一部分测试计算机上的缺陷。
被测程序信息 (Application Under Test [AUT] Information)	包括应用程序或组件的主版本号和次版本号。必要时，还应包括程序的本地化语言和所有程序依赖项的版本信息。
测试结果 (Test Result)	测试结果通常包括通过或失败。本章的后面部分会讨论其他可能的选项。

不管日志的格式或者介质是什么，当记录自动化测试的数据时要考虑以下几点最佳实践：

- 通过的测试应生成尽可能少的日志记录。
- 失败的测试应该提供调试失败所需的足够多的信息，这样就不需要重新运行测试或者连接调试器。日志文件应该包括待测的二进制文件和函数，以及待测功能、预期结果与实际结果。
- 错误代码应该清晰并使用容易理解的文字来写。

- 记录应该无须重新编译就可以配置。测试在运行时可以有多种记录级别，包括“不记录”、“正常记录”或“详细记录”。

下面是一个简单的日志文件的例子，包括有测试信息、测试结果和执行测试时的附加状态信息。

另一个以 XML 格式的简单日志文件如下。

```
<harness harness_client="testclient1">  
<harness_client_machine>testclient1</harness_client_machine>  
<harness_client_ip>157.59.28.234</harness_client_ip>  
  
    <result_information>  
  
        <parameter name="computer_name" value="LAB-09563"/>  
        <parameter name="result_filename" value="1234.LOG"/>  
        <parameter name="result_testid" value="1234"/>  
        <parameter name="result_comments" value="Passed"/>  
  
    </result_information>  
  
</message>  
  
</harness>
```

分析

执行测试中的步骤只是自动化测试的开始。在执行以后，还必须进行一定程度的调查以确定测试的结果。有的时候，分析这一步挺简单，但是用来确定一条测试是不是通过的标准可能会非常复杂。测试准则（test oracle）是测试用例所预期的测试结果的源头。Windows API 里的 `CreateFile` 函数能够创建一个新文件，或者打开一个已存在的文件。如果成功，函数将返回这个文件的句柄（一个唯一整数值）；如果失败，函数将返回一个错误代码。你用一个很微不足道的方法就能测试这个函数，即通过检查返回值来确定测试状态。

```
TEST_RESULT TestCreateFile(void)
{
    HANDLE hFile = CreateFile(...)

    if (hFile == INVALID_HANDLE_VALUE)
    {
        return TEST_FAIL;
    }
    else
    {
        return TEST_PASS;
    }
}
```

这个“测试”实际上仅仅确定 `CreateFile` 函数是否返回了一个值。为了确定函数是否真的工作以确定一个精确的测试结果，还须进行大量的附加测试。测试工程师可以创建一个测试准则（或测试校验函数）来帮助确定测试状态。

```
TEST_RESULT TestCreateFile(void)
{
    TEST_RESULT tr = TEST_FAIL;

    HANDLE hFile = CreateFile(...)

    if (IsValidFile(hFile, ...) == TRUE)
    {
        tr = TEST_PASS;
    }

    return tr;
}

BOOL IsValidFile(HANDLE hFile, ...)
{
    /* 测试准则：
       检查文件的句柄是否为INVALID_HANDLE_VALUE,
       确定文件是否存在磁盘上,
       确定文件属性正确,
       如果文件是可写的, 确定可以写该文件,
    */
}
```

```

    作其他相关的检查。
    如果文件有效，返回true值
    否则，返回false值
    */
}

```

测试准则的困难之处在于精确地预言出它们所验证的操作的结果。准确的测试准则既有赖于对被测功能的丰富知识，也得益于对功能意图描述清晰的文档。至少，这些准则必须验证测试结果是否成功，但它们也必须验证环境和程序的各种变化。这些变化在测试的同时发生或者作为功能测试的副作用而发生。

测试准则的挑战

我在 Windows 98 团队的职责之一，是测试 Windows 操作系统的图形函数，例如在屏幕上绘制文本和图形。在 Windows 里，`SetPixel` 函数是根据指定坐标将像素的颜色改变成一个特定色。相应的 `GetPixel` 函数是获得指定坐标的像素颜色。

我记得一次深夜里关于测试准则和那个操纵屏幕上像素的函数的有趣讨论。这次讨论围绕着我们是否可以放心地使用 `GetPixel` 函数作为准则来测试 `SetPixel` 函数。换句话说，假设我使用 `SetPixel` 函数设置了一个具体像素的颜色，那我可不可以信任 `GetPixel` 函数去检索数据而不是直接地返回 `SetPixel` 函数所调用的那个值呢？

我开始设计并编写了一些代码原型来查询显示驱动程序的数据结构，并尝试不通过调用 `GetPixel` 函数来检测像素的颜色。可一些人显然比我聪明得多，他们指出实际上 `GetPixel` 函数就是这么做的。

他们说得对，我可以放心地使用 `GetPixel` 函数来测试 `SetPixel` 函数。但是，我仍然会时不时地听到人们问这样的问题，它也总是会提醒我思考选取测试准则里面的哲学。

当运行手动测试的时候，通常很直接就能判断出测试用例是不是通过了，也很容易判断出因为测试环境的种种变化而不应该执行特定的测试用例。然而，自动化测试一定要能不加人工干预地做出这样的判断。如前所述，判断出测试是通过还是失败是有挑战性的。通过和失败之间的灰色地带常常会造成困惑。除了通过和失败之外，测试还可以有其他的结果。理解这些测试结果对于准确汇报自动化测试的完整状态是必不可少的。表 10-4 里列举了几种可能的测试结果类型。

表 10-4 测试结果的种类

测试结果	描述
通过 (Pass)	测试通过。
失败 (Fail)	测试失败。
跳过 (Skip)	跳过测试的情况通常会在测试可选功能时。例如，一套视频卡的测试就可能包括了一些仅仅能在支持某些特定功能的硬件上运行的测试。
放弃 (Abort)	最常见的以放弃为结果的情况发生在当预期的支持文件不存在的时候。例如，如果测试附属品（运行测试所需要的附加的文件）存在于一个网络共享位置，但是那个共享位置却是失效的，那么测试就被放弃了。
阻断 (Block)	由于一个已知的程序或系统问题，测试在运行时被阻断了。由于一个已知的缺陷而使测试不能运行的时候，就将其标记为被阻止（而非失败）。这样不会让失败率人为地增高，但是高数目的被阻断的测试用例显示出质量和功能还处在未被测试和未知的状态。
警告 (Warn)	测试虽然通过了，但是指出了警告信息可能需要被更细致地检查。

人们可能使用比表 10-4 更多的测试结果类型，但在另一方面，可接受的分析也许仅包括通过、失败和未知这三种结果类型。在自动化测试中，测试准则不仅负责精确地确定测试状态，还必须对任何没有通过的测试指出其状态和需要采取的行动。

报告

比较小的项目经常直接用测试日志文件作为测试报告。测试日志文件记录着测试用例和测试套件运行通过或失败的结果。只要这些日志文件不是太多，就可以直接用作测试报告。很多测试项目，比如微软产品的测试项目，经常有成百上千的测试套件和数以万计的测试用例。对于这种规模的项目，人工检查其日志文件是不可能的。解决这个问题的办法之一是把测试结果直接记录到数据库，然后利用数据库的数据汇总和分析、报告功能生成测试状况报告。不过在很多情况下数据库的使用并不现实。某些测试场景没有网络连接。即便有网络连接，上传测试结果到数据库的速度远不如将测试结果直接记录在本地存储设备或调试串流上来的快。

另一种比较常用而且有效的办法是测试日志文件的自动解析。日志文件解析程序可在运行测试用例之后独立执行，既可以用来解析一个日志文件，也可以用来解析成批的日志文件。解析程序可简单地记录测试用例名和运行结果，也可以记录可能用得到的重要数据，如测试分类，被测的产品组件，以及可用来分析测试失败结果的相关信息。通常人们用数据库作为后端存放解析得到的数据，用应用程序或网页显示解析结果。表 10-5 是一部分测试用例运行结果的例子。

表 10-5 测试用例结果

产品组件	通过	失败	跳过	阻断	未执行	结果总数	%通过	%完成
组件 1	1,262	148	194	415	0	2,019	69.15	100.00
组件 2	1,145	78	18	28	0	1,269	91.53	100.00
组件 3	872	18	28	4	0	926	97.53	100.00

注意在这里跳过的测试用例并不影响通过率，但被阻断的测试用例却要计算在内。计算公式是这样的：通过的用例数 / (全部结果 - 跳过的用例数)。按照表 10-4 的定义，有“跳过”标记的测试是测试工程师有选择地让它们不要运行。“阻断”则表示这些测试用例应该执行但不能执行。即使我们无法知道阻断用例的运行结果，这些测试仍属于失败一类。测试通过率的计算方法有多种，这里列出的是微软最常用的一种。

清理

在可能的情况下，自动化测试应该尽量将运行环境回归到测试运行前的状态，以保证后续的测试能正常执行。失败的测试反映的应该是产品中的问题，而不是前面测试留下的垃圾数据或环境设置的结果。环境清理对于那些需要很长时间设置测试环境的用例尤其重要。

虽然理想的工作流程应该是在测试执行后清理环境，但现实中却不一定处处可行。例如，如果一个有内存泄漏或内存破坏 (memory corruption) 问题的测试用例每次在运行完之后重设环境，就会掩盖这里的内存问题。反过来，如果所有的测试都不作环境清理，分析失败的测试结果的根源就会很困难。一个折中的处理方式是在所有的自动化测试用例后加上环境清理

的步骤，同时重复运行另一组测试，不作环境清理的步骤。例如，在运行附带环境清理的自动功能性测试同时，在另一些计算机上重复运行一组不附带环境清理的场景测试。

帮助

有时测试工程师会因为经验的增加转去负责更关键的部分，或转到公司内其他的产品组。他原来负责的测试程序就会转交给其他的测试工程师维护、设置、运行和进行测试结果分析。**SEARCH** 的最后一个步骤就是为了确保自动化测试在整个产品生命周期里的运行和维护。

测试代码和产品代码一样，必须是可维护的。除了清晰合理的结构和代码注释，帮助这个阶段还包括建立一系列相关文档。测试目标、已知的限制、设置步骤以及如何解读运行结果等都应包含在文档之中。

对大多数测试工程师来说，建立文档是 **SEARCH** 中最没有吸引力的一个步骤，但这一步可能成为这些自动化测试程序整个生命周期中最重要的一个环节。

提示：很多微软产品的自动化测试代码比产品本身的代码还要多。

好东西真的越多越好吗？

微软的内部工具库里有四十多项测试工具。由于微软产品的多样化和各测试组独特的需要，虽然各部门力求统一测试工具，测试工具的重复开发是不可避免的。不同的测试工具为产品开发组之间共享测试用例以及测试结果造成了困难，同时也增加了测试工程师更换产品组以后的适应和学习时间。在微软这么大的公司要求各产品组使用完全一样的工具也许不是很现实，但太多方案造成的问题很难解决。

幸运的是，微软正努力缩小公用测试工具集，保留几种长期被使用的测试工具，同时许多团队正向 **Visual Studio Team System** 提供的解决方案靠拢。

自动化测试，跑起来！

简单的自动化测试只需要启动一个程序或一段脚本。大型的自动化测试则依赖复杂的构造框架（framework）确保一轮测试（test pass）产生格式一致、通用的测试报告。图 10-4 显示了包含 **SEARCH** 所有内容的自动化测试基础设施（infrastructure）。构造完好的自动化测试系统应该使得完成整个测试套件的执行就像是按一下按钮。

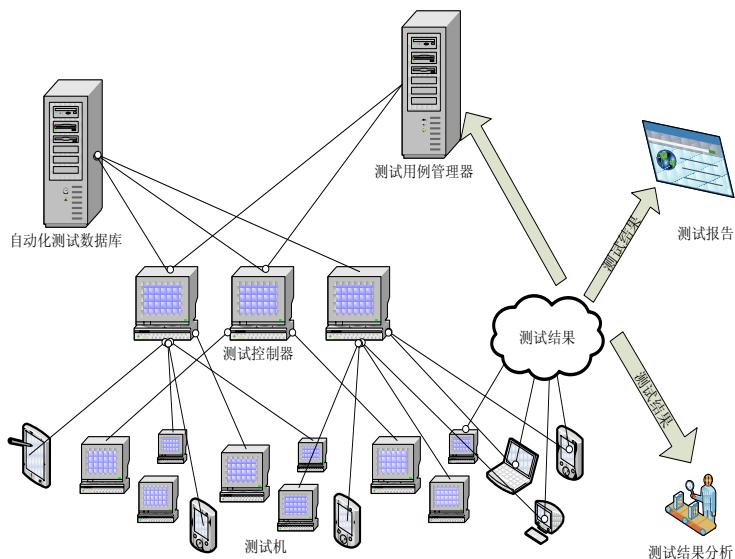


图 10-4 自动化测试基础设施

把一切装配起来

一个自动化测试系统包含很多部分。除了测试用具之外（参见图 10-3），还需要各种机制从测试用例管理系统中读出需要运行的测试，再将这些测试对应到自动化测试二进制文件或脚本。同时测试系统还要包括计算机和各种设备运行自动化测试程序。最后，分析测试结果、生成测试报告并由测试用例管理系统记录在案。

提示：微软专用于自动化测试的计算机超过 100,000 台。

大型自动化测试系统

在微软的大多数自动化测试系统中，一轮测试往往由一条简单的命令启动。这条命令可以来自命令行、网页、或某个应用程序。有些产品组建立了产品版本的自动监控程序，每当新版产品编译完成，这个程序就会自动启动新一轮的测试。自动化测试流程从 TCM（测试用例管理器，是 Test Case Manager 的首字母缩略词）开始。根据每一轮测试的设置，TCM 选择相对应的测试用例集。

之后，相应的代码或脚本文件开始执行测试用例集。TCM 和自动化测试数据库之间的协调由专用标识符，或全局统一标识符（GUID）完成。自动化测试信息存放在另一个数据库的好处是将自动化测试有关的内容与测试用例本身分开，如命令行选项、测试文件以及其他相关文件的存放地点。

运行有些自动化测试会用到额外一些称为“测试附属品”（test collateral）的文件，如为 Microsoft Word 测试准备的.doc 文件或者为 Windows Media Player 测试准备的媒体文件。这些文件通常存放在自动化测试系统的某个数据库服务器中，或是某台计算机的共享目录上。接下来，自动化测试数据库和 TCM 通知测试控制器，由测试控制器（test controller）根据测试用例的配置要求将测试用的计算机准备好。通常每个控制器负责将测试任务分配在 8 到 50 台测试机上（测试机的数量由多种因素决定，包括硬件及网络的局限性）。测试控制器将测试分配给准备好的测试机，等待测试运行完毕（或运行崩溃），然后获取测试机上的测试日志文件。有时测试控制器负责测试日志的分析、处理，但多数情况下，控制器将日志

发送回 TCM 或另一台计算机进行分析处理。

在对日志分析得到测试结果之后，测试系统要备份日志文件（以便于分析失败的测试）并将测试结果记录在案（存入 TCM）。微软很多自动化测试系统直接将失败的测试结果记录到产品缺陷追踪系统里。最后，各种测试报告从不同角度显示测试结果，测试小组对失败的测试作进一步的分析。

测试自动化中的常见错误

自动化测试程序有无数的优越性，但写测试代码也很容易出错。微软测试开发人员的编程能力与产品开发人员相当，但产品代码和测试代码有一个很大的区别：产品代码是检测过的。公平地说，借助反复的测试和测试结果的不断反馈，你可以说是产品检测了自动化测试。不论怎么说，微软各产品组都有一个同样的目标，测试代码应该和产品代码有同样的质量目标。以下列出了编写自动化测试代码中一些常见的错误。

- **硬编码路径** 测试在执行时常会用到一些外部文件。最快、最简单的办法是把文件路径写在程序中。但是路径会变，文件服务器会改甚至退役。一个较好的做法是用 TCM 或自动化测试数据库中的文件记录这些的路径信息。
- **过于复杂** 我们在第七章讨论过的复杂性问题在测试代码和产品代码中同样普遍。正确的目标是尽可能用最简单的程序充分地测试一个功能。
- **难于调试** 测试人员调试测试失败的流程应该快速、简洁，不应该是以数小时计的时间投资。造成调试困难的关键因素是日志记录不充足。一旦测试失败，日志应记录下这个测试的失败原因。例如，“流测试失败：缓冲区大小期望值是 2048，实际大小是 1024”好过“流测试失败：缓冲区尺寸错误”或者最简单的“流测试失败”。如果日志记录足够好，有些时候报告和修复失败的测试可以完全不依赖于调试程序。
- **误报和漏报** 误报指的是测试人员发现测试失败并不是产品的缺陷，而是测试代码中的缺陷引起的。与之相对的漏报的后果则更为严重：自动化测试错误地报告测试通过。通常分析测试结果时，测试人员只检查失败的测试。除非漏报的缺陷在其他测试中出现，或被内部用户在平常使用产品时发现，否则其结果是产品缺陷流通到产品消费者手中。

编写自动化测试是一项足够困难的工作，而编写高质量的自动化测试则是一个艰巨的工程。尽管产品组最初的目标如此，不是所有微软的测试代码都达到产品代码的质量水准——至少现在仍然不是。除了我们上面提到的原因以及其他相关问题，误报和漏报要比测试小组期望的更为普遍。在第十二章，“其他工具”，我们会讲到测试人员如何使用一些工具以避免上述某些问题，写出高质量的自动化测试。

本章小结

显而易见自动化测试在微软举足轻重。如果没有在自动化测试上的投入，要充分地测试庞大的诸如像 Windows、Office 和 Visual Studio 这样的产品是不可能的。再考虑到繁多的本地化版本和十年的产品支持计划，微软确实需要在自动化测试上做大量的投入。

自动化测试的一个很大目标就是扩展测试的影响范围。即便开发一个自动化测试用例需要很大的投入，但当那个自动化测试用例可以在不同的应用程序配置和语言中执行，甚至能被产品维护部门使用十年之久的时候，其价值就会显著增加。将这种自动化测试放入一个集配置测试平台、执行测试、报告结果和登记缺陷于一体、可方便添加测试用例的自动化测试基础设施，是一个优秀的、长期有效的测试自动化方案的基础。

第 11 章 非功能测试

阿伦·培智

这是一个几乎每个人都经历过，或者听说过的故事 ——

产品开发进行的非常顺利。产品团队的测试和开发人员团结协作，工作取得了快速的进展。开发团队每天准点报时般产生一个新的构建（build），测试团队也相应地每天获取构建的更新，然后生成和运行新测试。功能方面的缺陷一经发现，开发团队就马上修复。即使发布的日子快到了，大家也没有什么压力，因为软件工作得很好、测试也都通过了。Beta 版试用者也反馈说软件就像预期的那样，各个方面都正常工作。所以软件在发布之时，初始的反响也是正面的。

但两周过后，第一通电话来了，很快更多的电话来了。该团队开发的软件按照设计是需要在后台一直运行的。可是，连续使用几周以后，软件仍然挣扎着“工作”，但是性能已经降低到不堪忍受的地步了。这个团队对该应用软件的特点和功能过于自豪，以致没想到过（或者忘了）象用户那样，几天、几周甚至几个月地连续运行该软件。每日构建意味着在测试期间他们每天重装软件。最长的运行时间也不过是从周五到周一之间的周末，这么短的时间不足以把资源的每次小泄漏积累到显而易见的地步，除非连续运行近两周才行。

接下来的一周里，开发团队修复了近一打的内存泄漏和性能问题，测试团队也尽了最大的努力来确保这些修复不会影响任何的现有功能。该团队在发布了他们引以为荣的软件二十五天之后发布了第一个紧急补丁。接下去的几周和几个月内，他们又发布了其他好几个补丁。

功能之外

非功能测试是个易混淆的词汇，但在测试界广为使用。在某种程度上，它是有道理的。因为功能测试涉及了软件在功能上正反两面的测试，而非功能测试就是所有其他方面的测试。定义为非功能的测试领域包括了性能、负载、安全、可靠性和其他很多方面。非功能测试有时也被称作行为测试或质量测试。非功能测试的众多属性的一个普遍特征是一般不能直接测量。这些属性是被间接地测量，例如用失败率来衡量可靠性或圈复杂度，用设计审议指标来评估可测性。

国际标准化组织（ISO）在 ISO 9216 和 ISO 25000:2005 中定义了几个非功能属性。这些属性包括：

可靠性

软件使用者期望软件能够无误运行。可靠性是度量软件如何在主流情形和非预期情形下维持它的功能，有时也包括软件出错时的自恢复能力。例如，自动定时保存现行文件的功能就可以归类到可靠性。可靠性在微软内部是一个严肃的课题，是可信赖计算计划(Trustworthy Computing Initiative)的支柱之一(<http://www.microsoft.com/mscorp/twc>)。

可用性

如果用户不明白应该如何使用，那么，即使是零差错的软件也会变得毫无用处。可用性测量的是用户学习和控制软件以达到用户需求的容易程度。进行可用性研究、重视顾客反馈意见和对错误信息和交互内容的检查都能提高可用性。

可维护性

可维护性描述了修改软件而不引入新错误所需的工作量。产品代码和测试代码都必须具备高

度的可维护性。团队成员对代码的熟悉程度，产品的可测性和复杂度都对可维护性有影响。

(可测性在第四章“软件测试用例设计的实用方法”有讨论，复杂度在第七章“使用代码复杂度的分析风险”有讨论)。

可移植性

微软 Windows NT 3.1 曾运行在四种处理器家族中。当时，代码的可移植性是 Windows 部门的一大要求。现在，Windows 的代码则必须能在 32 位和 64 位处理器上运行。许多微软产品运行在 Windows 和 Macintosh 两种平台上。可移植的产品和测试代码对微软的许多部门来说都很关键。

属性测试

除了以上所说之外，属性还包括了好几打的其他质量属性，例如可依赖性、可重用性、可测性、可延展性和可适应性等等。所有这些都能用来帮助评估和理解超越功能以外的产品质量。可扩展性（程序能应付过载的能力）和安全性（系统应对非授权改动图谋的能力）是微软团队内两个接受测试最多的属性。

微软测试团队经常会有专门队伍专责于此类属性。在可用性的例子中，我们甚至分立出一个工程职种，专职于运行测试，以及和革新测试工具和方法。在组织结构上，有两种主要方法来进行非功能领域的测试。大一点的团队可以如图 11-1 那样组建与功能测试团队并行的非功能测试团队，由测试主管或测试经理来管理。

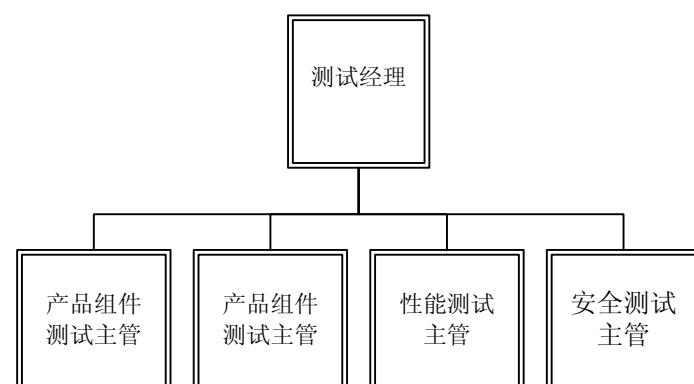


图 11-1 非功能测试的专职队伍

图 11-2 所示是一个被更加广泛使用的方法。它使用虚拟团队来测试非功能领域。虚拟团队并不向同一个经理汇报，但在一起工作，以处理他们在性能工作之外的那部分具体测试。每一虚拟团队都会指定一个组长，以负责该团队的战略、目标、成功度量。

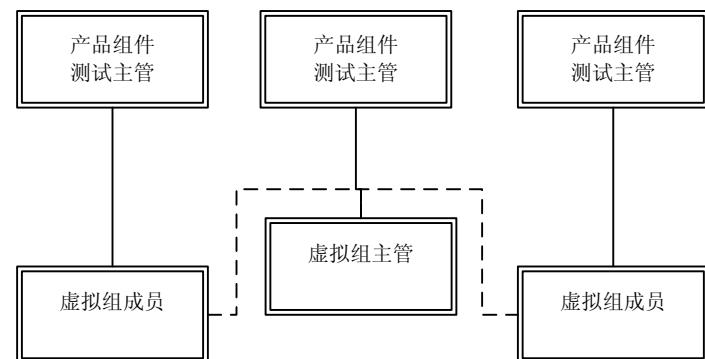


图 11-2 非功能测试的虚拟团队

综合使用上述两种方法也很常见。例如，成立一个专门的团队负责安全或性能测试；同时虚拟团队负责可用性测试和可达性测试。

阐述不同类型的非功能测试的资料已经有很多。这里，我并不打算针对微软使用的每一类型都进行讨论。在微软已有解决方案的非功能测试类型中，我选出在革新、方法或尺度上比较有意思的若干类型来讨论。

性能测试

因为各属性之间在范围上有重叠，很多非功能属性的名字是可以通用的。例如，性能测试经常会跟下面将要提及的压力、负载和可扩展性测试的概念联系起来。在微软的许多测试部门中，同样的测试工程师或测试团队负责所有这些领域。这些领域的测试手段和目标有很多不同点，但也有几个相似点。例如，在有几千个用户联线的情况下测试一个服务器系统的表现是一种性能测试（很多其他人会认为这是负载测试或可扩展性测试）。类似地，软件经过好几周或几个月的连续运转而无需重启的测试也被很多人看作是性能测试的一部分（大多数人都会说这是可靠性测试或马拉松测试）。

最常见的性能测试可以称作读秒测试。很多年前，一些测试工程师确实是坐在屏幕前，用读秒计来测量好些不同功能测试的表现。虽然用这个方法作为性能测试的原始形式是可以理解的，但是这样的方法容易出错，多数情况下它也不会是为软件性能计时的最好方法。

这类性能测试的要旨只是测量某些重要操作(action)的执行时间。这种性能测试是设计单个测试或一组测试去测量软件对不同的用户操作的响应时间，或是测量在受控环境下的产品功能。因为读秒测试方法既不具备可扩展性，又不具备真正意义上的可重复性，大多数的性能测试实际上是通过执行测试用例并记录所用时间的自动化测试来实现的。

性能测试的目标是发现重大的系统瓶颈。你可以想象一个系统由一系列的瓶颈组成；发现并改善一个瓶颈往往会在其他地方产生一个新的瓶颈。例如，我曾为一运行微软 Windows CE 的器件部门工作。我们发现的第一大性能问题体现在某一具体硬件环境下的内存管理中。我们把问题分离出来，改善了内存分配的效率。尔后再次运行我们的测试，又找到了一个新的瓶颈，这次体现在网络吞吐量上（throughput）。解决了这个问题后，我们接着又为下一个瓶颈改善而工作，然后再下一个，直到整个系统都达到了性能目标。要记住的是：关键在于要尽早订立性能目标，否则你可能不知道什么时候该停止性能测试。

怎样测量性能

大概性能测试最难的部分在于决定测量什么。性能测试工程师使用几种不同的方法来帮助他们确立测试方向。有一句话是每一位有经验的性能测试工程师都会告诉你的：在设计过程的早期就积极地介入代码评审和分析性能目标是绝对必要的。事实上，若要满足大多数非功能测试需要，最好的办法是在程序设计阶段就加以考虑。下面是一些在设计阶段能帮助发现潜在的性能问题的技巧：

- 提出疑问：找出有潜在性能问题的地方。对网络交通的拥塞状况、内存管理的效率、数据库设计的合理性、或其他任何有关地方提出疑问。即使你并没有性能设计的解决方案，而只是通过让其他团队成员考虑性能问题，测试工程师也一样能够产生很大的影响力。

- **考虑全局：**不是片面地考虑局部的优化，而是考虑全面的用户场景。你将会在整个开发过程中有相对充足的时间深入性能场景的细节，但是在设计阶段的时间最好是花费在考虑从头到尾的（end to end）场景上。
- **明确目标：**象“响应时间应该很快”这样的目标是不可度量的。应用 SMART (Specific-具体的, Measurable-可度量的, Achievable-可实现的, Relevant-相关的, Time-bound – 有时限的) 标准来设计目标。举个例子，“每个用户操作的执行时间必须不超过 100 毫秒，或上一版本的 10% 的时间之内将控制权返回应用程序”。

还有一个要考虑的技巧是预测哪里可能有性能问题，或者说分辨出哪些操作对用户来说是最重要的，从而是需要度量的。而往往最有效的办法就是在设计阶段就定义这些场景。基于场景的方法对于测试旧代码而言，也同样是个有效的选择。无论何种情况下，以下是一些很有益的性能测试的技巧：

- **建立基线** 早定义、早测量的一个重要方面是建立基线。如果性能测试开始得晚，那么瓶颈发现后，要确定是开发阶段的什么时候引入了该瓶颈就会有难度
- **经常运行测试** 一旦你建立了基线，就尽量经常地测量。当需要精确诊断哪些代码改变导致性能下降时，这些测量就是最好的帮手。
- **测量响应效率** 用户并不关心程序在后面用了多长时间来执行。他们关心的是应用软件响应是否够快。性能测试必须着重于测量用户响应时间，而不管该操作的后台计算需要耗费多少时长。
- **测量的是性能** 我们常常忍不住想在性能测试中加入功能或其他类别的测试。但是，既是性能测试，就要专注于测量性能。
- **充分利用性能测试** 由上一个技巧的另一面可知，性能测试所采用的方案经常在其他测试场合是非常有用的。只要有可能，就要把自动化的性能测试方案用在其他自动测试系列中（例如压力测试系列）。
- **预估瓶颈** 针对延迟会发生的地方做性能测试，比如文件和打印 I/O，内存程序，网络操作，或其他任何不响应行为可能发生的地方。
- **使用工具** 与上一技巧相联系的是，使用工具来模拟网络或 I/O 延迟，以确定该应用软件在非常规条件下的性能特征。
- **合理使用资源很重要** 响应时间和延迟两者都是性能的关键标识，但不要在性能测试中忘记检查 CPU 的负荷，磁盘或网络 I/O，还有内存。例如，假如你正在测试多媒体播放器，除了响应时间，你还要检查网络 I/O 和 CPU 使用率，以确保该软件的资源占用不会导致其他软件不工作。
- **“干净机器”：用还是不用** 让一部分性能测试运行在干净机器上(新安装的操作系统和所测软件)，而其余部分在基于顾客配置的机器上运行。干净机器对于产生一致的数据很有用，但如果性能会被其他应用软件、外接程序(Add-ins)或其他扩展严重影响的话，这些数据就会起误导作用。在干净

机器上运行性能测试可以给你产生最好的数据，但是在一台装满了软件的机器上产生的数据会更加接近你的顾客所体会到的。

- **避免改变** 克制住给你的性能测试小修（或大修）的冲动。长远来看，测试本身改动得越少，所得数据就越精确。

性能计数器（performance counters）经常被用来检测系统的性能瓶颈。性能计数器是揭示应用软件或系统的某些性能指标的细节测量工具，它使得监测和分析这些指标成为可能。所有 Windows 操作系统的版本都包含了一个监测这些计数器的工具（Perfmon.exe），Windows 在很多瓶颈存在的地方设置了性能计数器，例如 CPU，磁盘 I/O，网络 I/O，内存统计，和资源占用率。Perfmon.exe 的使用截图如图 11-3 所示。

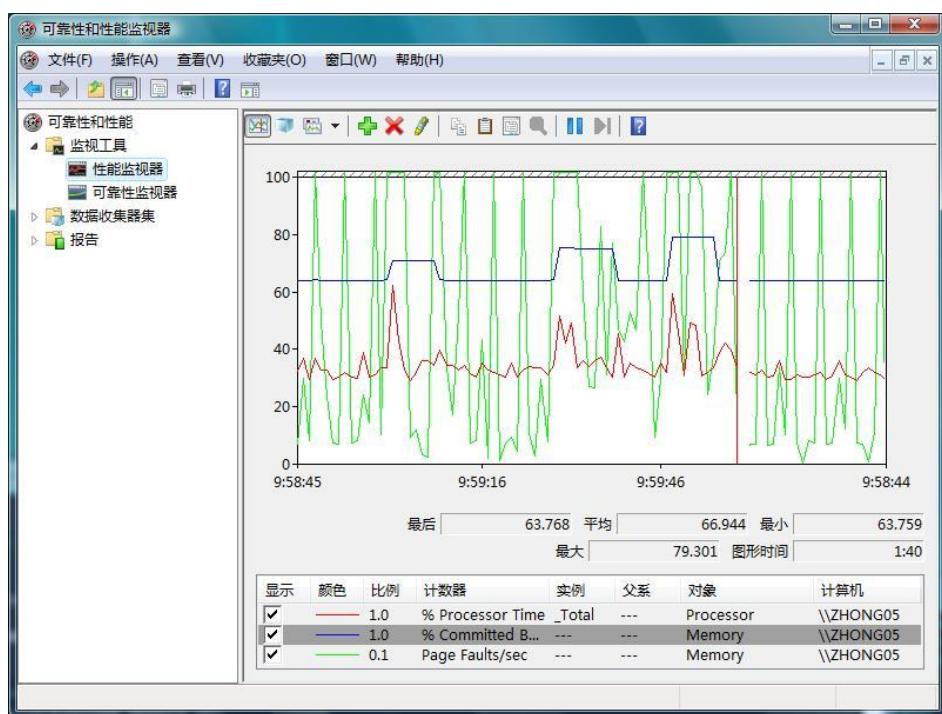


图 11 - 3 可靠性和性能监视器

应用软件可以使用专用性能计数器来追踪自有的对象（object）的使用、执行时间、或跟性能有关的任何东西。在设计阶段就计划好、并且早早实现一组全面的性能计数器，会对贯穿整个产品生命周期的性能测试和分析十分有益。

很多书和网上的参考文献拥有性能测试领域方面大量而深入的资料。patterns & practices Performance Testing Guidance Project 的两个网站 <http://www.codeplex.com/PerfTesting/> 和 <http://msdn.microsoft.com/en-us/library/bb924375.aspx> 是找到更多讨论此题目的资料的好去处。

压力测试

应用软件在预期的和重负载条件下的表现和处理容量增大的能力，通常被归类在性能测试下面。广义上，压力测试经常包括了负载测试、平均无故障时间(MTBF - mean time between failure)测试、低资源测试、容量测试、或重复性测试。这些不同测试的方法和目的之间的主

要区别是：

- **压力测试** 一般来说，压力测试的目的是要通过模拟比预期要大的工作负载来让只在峰值条件下才出现的缺陷曝光。压力测试是要发现软件的弱点所在。内存泄漏、竞态条件、数据库中的线程或数据行之间的死锁条件、和其他同步问题等等，都是压力测试能发掘出来的常见缺陷。
- **负载测试** 负载测试是要探讨在高峰或高于正常水平的负载下，系统或应用软件会发生什么情况。例如，一个网络服务的负载测试会试图模拟几千名用户同时连线使用该服务。性能测试一般都包括测量峰值负荷下的响应时间。
- **平均无故障时间（MTBF）测试** MTBF 测试是测量系统或应用软件在出错或当机前的平均运行时间。这一测试有几个变体，包括平均无错时间(MTTF) 或平均无当机时间(MTTC)。技术含义略有不同，但实践上，这些词汇都是一个意思。
- **低资源测试** 低资源测试是要确定当系统在重要资源（内存、硬盘空间或其他系统定义的资源）降低或完全没有的情况下状况。重要的是要预估将会发生什么，例如为文件存盘而无足够空间、或一个应用程序的内存分配失败时将会发生什么。
- **容量测试** 与负载测试非常相似，容量测试一般是用来执行服务器或服务测试。目的是要确定一台或多台计算机能支持的最多用户数目。容量模型通常建立在容量测试数据基础上。有了这些数据，营运团队(Operations)就能定计划什么时候增加系统容量：要么增加单机资源，如 RAM、CPU 和磁盘空间等；要么干脆增加计算机数目。
- **重复性测试** 重复性测试是为了确定重复某一程序或场景的效果而采取的一项简单而“粗暴”(brute force)的技术。这个技术的精髓是循环运行测试直到达到一个具体界限或临界值，或者是不妙的境况。举个例子，一个操作也许会泄漏 20 字节的内存。这并不足以在软件的其他地方产生任何问题，但如果测试连续运行 2000 次，泄漏就可以增长到 4 万字节。如果是提供核心功能的程序有泄漏，那么这个重复性测试就抓到了只有长时间连续运行该软件才能发现的内存泄漏。通常有更好的办法来发现内存泄漏，但有时候，这种简单“粗暴”的方法也可以很有效。

压力订书机

在 Microsoft Office 开发早期，测试工程师常会用意想不到的方式来使用办公用具。我们的目标是模拟真实用户的键盘输入，让程序的输入缓冲超载。我面临的困难，是要找到尺寸和形状都合适的办公用品，在我去吃午餐的时候能很方便的放在键盘上。最后我们发现订书机最合适不过了。当我吃完午餐回来，我的屏幕上总会有一个 ASSERT 错误，或者是当机。

- Craig Fleischman, 测试主管

USB 死亡之车

在 Windows 2000 项目过程中，我们测试“即插即用”的方式很有意思。我们制造了 USB 死亡之车。我们最开始用了一种两层的小车，就象在图书馆常见的那种。我们把大约十个 8 端口的接线器连在一起，再把每个端口都插上不同种类的 USB 设备。在小车后面加上 USB 方向盘，再用一个 USB 收音机当天线。前面再装上两个照像机。所有的电源都接在一个 USB 不间断电源（UPS）上。整个小车，可以轻便移动，只有两条接线（电源，USB）。这条最终的 USB 线，被接到一块 USB PCMCIA 卡上。我们会把卡插到一台笔记本电脑上，看着操作系统启动它所连接的 50 多个设备，然后（在它结束之前或之后）突然拔掉 PCMCIA 卡。如果出现蓝屏或者别的错误的话，我们就会叫相关的开发人员来看电脑。同时我们会把小车推到下一台笔记本电脑，期待发现不同的缺陷。

- Adrian Oney, 高级开发主管

分布式压力测试

在微软压力测试很重要。大多数产品线都会在上百台甚至更多的机器上做压力测试。有些压力测试部分会延续很长的时间—常常是 3 到 5 天或更多天的连轴转。然而对大多数团队来说，绝大部分的压力测试是在员工晚上回家和第二天上班之间的 12 到 14 小时内完成的。每个人都义务提供他们的计算机来做夜间的压力测试。测试、开发、管理甚至产品支持人员每天晚上都运行压力测试。

当运行压力测试时，失败和当机是不可避免的。对于小团队来说，可以轻松的通过电话，电子邮件或者敲门来报告压力测试失败和找到代码的拥有者。不幸的是，如果当机发生在一台无人注意的计算机，或者是在弗兰克的电脑上，而他正在休假，那么对失败的调查和调试可能永远都不会发生。

大的团队需要更有效的方法来判断哪些计算机有压力测试失败，和谁应当去调查失败。最常见的办法是用普通的客户端-服务器方法。因为压力测试总是隔夜运行，理想状况下，客户端的压力部分总是在待机状态，直到一个特定时间点。夜间压力测试是开发周期的一个重要部分，这样才能保证每个人下班之前都不会忘记启动压力测试。Windows Vista 团队则采用了一个后台程序，它可以用一个在通知区域的图标来配置。

分布式压力架构

一个分布式的压力测试架构比在第十章“测试自动化”讨论的自动化基础架构要稍微简单一些。但它有一些实现上的挑战。图 11-4 展示这样一个系统的基本工作流程。

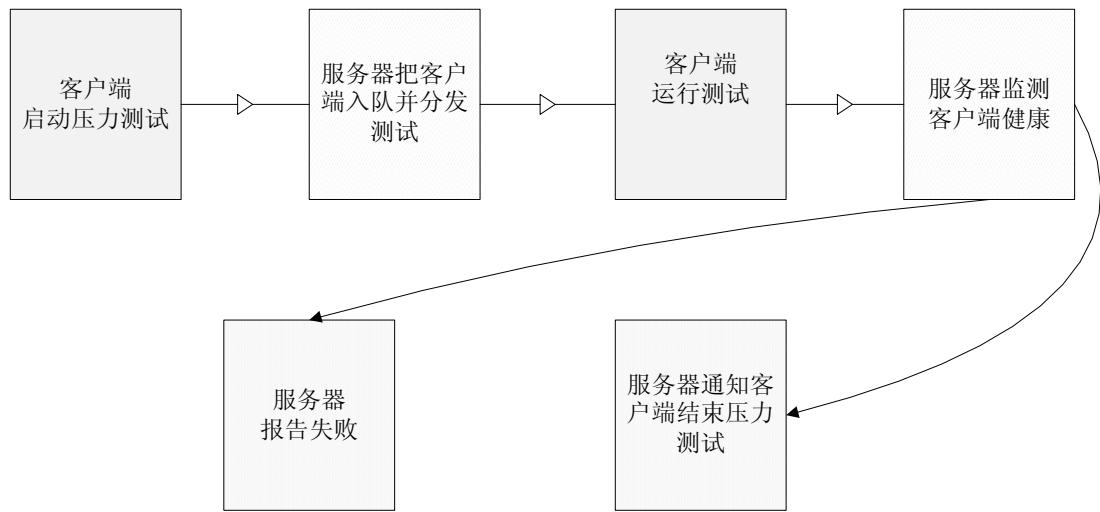


图 11-4 分布式压力系统架构

压力客户端

就象前面提到的那样，在被压力测试的计算机上(压力客户端)，由一个应用程序启动压力测试。虽然也可以手工启动压力测试，但在许多情况下，压力测试都是在预先设定的时间自动启动的。启动过程就是简单的通知服务器，宿主计算机已经准备就绪，可以运行压力测试了。这时候，服务器就把各种压力测试分发到宿主计算机运行，一个混合测试会运行不同的时长，直到设定的时间点，或者压力测试被手工停止了（当员工早上在设定的终止时间前到达了，或者需要收回计算机作它用时，手工停止比较方便）。图 11-5 显示很多压力客户端联结到压力服务器。

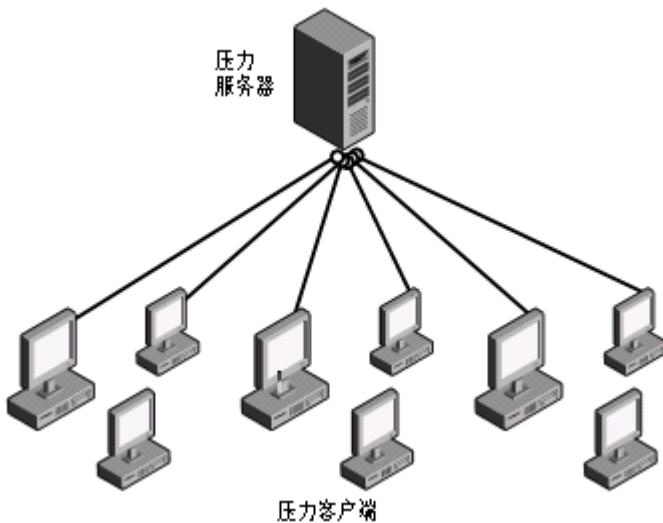


图 11-5 分布式压力测试

对于操作系统的压力测试，如 Windows 和 Windows CE，压力客户端计算机通常都被挂接到一个调试器以帮助调查失败原因。基于应用程序的压力测试套件能在整个测试运行过程中跟 WinDbg 或 Microsoft Visual Studio 这样的调试器挂接。它们也可以启动一个调试器作为实时 (JIT) 调试器。在应用程序与服务器部门，实时方法最为常用，而开发操作系统的团队则主要用“始终挂接” 调试器的方法。

压力服务器

压力服务器的作用是分发一组压力测试（通常叫做压力混合）到所有的压力客户端并且跟踪客户端的状态。当一个客户端联系服务器时，服务器把该客户端加入已知压力客户端名单，然后开始分发测试。客户端每隔一段时间就向服务器发一个心跳信号。心跳对于决定一台计算机是瘫痪了还是死机了很重要。如果在合理的时段内还没有接收到客户端的心跳，该计算机就会被加到需要更多调查和调试的计算机名单里。

在压力测试结束时，服务器发信号给所有客户端以终止压力测试。然后是检查失败并分派给相应的拥有者做进一步调查。

Windows 压力测试团队

Windows 压力测试团队并没有一个有上百台计算机的实验室。相反，他们依靠内部的 Windows 社区义务提供他们的计算机来做压力测试。每天，运行压力测试的上千台计算机报告成打甚至更多的压力测试失败。由于 Windows 压力测试团队并不拥有这些运行测试的计算机，他们的目标是保证这些计算机被迅速的跟踪调试以便计算机的拥有者能够使用它们来做他们的日常工作。

每天早上六点半左右，压力测试团队的一名或多名队员就会到达微软并且开始检查头天晚上（或周末）发现的所有失败，接下来的两到三小时他们跟踪调试问题并且分派给适当的拥有者。这种初期跟踪调试，虽然很花时间，但能帮助他们准确地找到每个问题的责任人。在一个有着上百开发员的团队，把问题在第一时间分派给恰当的人能节省很多时间。

当他们找到问题最有可能的拥有者后，他们给计算机的主人和被分派调查这个问题的工程师发一封电子邮件，描述问题所在。还有可能远程联结到 Windows 团队用的内核调试器 (kd.exe)，这样来解决问题的人甚至不用走出办公室就能查看失败原因。正是因为这样，尽管 Windows 压力测试团队的规模大，人员分布广，但是很多问题在计算机主人上班之前就被解决了。

多客户端压力测试的特点

为一个大型分布式压力系统所编写的压力测试，跟常见的自动化测试有很多相同的质量目标，但也具有一些独特的属性：

无穷运行 压力测试一般会一直运行，或直到被通知终止。标准的实现过程是让测试能够对 WM_CLOSE 信号作出快速反应，以便服务器能够运行不同时长的测试。

内存使用 如果压力测试有内存泄漏，通常会因为资源不足显现为其他测试的失败。理想情况下，压力测试不会有内存泄漏。由于测试需要和其他的测试同时运行，所以最好不要大量利用进程，线程，或者其它系统资源。

没有已知问题 许多团队要求所有测试要在一台私人计算机上跑 24 小时到一周的时间没有失败，才能被加到测试混合中。夜间压力运行的目标，是要确定当有很多不同的行为同时发生，并且以不同的顺序发生时，应用程序或操作系统会怎样反应。如果一个测试在每一台压力测试客户端机器上造成同样的失败，那么整个晚上的压力测试的努力就白费了，因为测试没有发现任何新的错误。

兼容性测试

应用程序兼容性测试(Application compatibility testing)一般注重于应用程序之间或所测试的

目标系统与其他应用程序之间的交互。其他应用程序可能包括内部和外部两种。在微软，Windows 团队在应用程序兼容性测试方面毫无疑问是付出了最大的努力。这是因为 Windows 每个新版本都增加了新的功能，但都必须继续支持为老版本的 Windows 设计的应用程序。应用程序兼容性(英文亦简写为 app compat)也影响大多数其他微软产品。 Internet Explorer 必须继续支持相关的插件或其他添加功能；已具有广泛且活跃的软件开发社区的 Visual Studio 或 Office，也必须支持各种各样第三方开发的功能。就算是在新版本上支持以前的文件格式都是非常重要的。

我现在用来书写本章的 Microsoft Word 2007 就支持 Microsoft Word 旧版本的文件格式以及其他为旧版本开发的外接程序和模板。我使用的几乎每个应用程序都支持打开旧版本的文件，也支持打开其他应用程序产生的文件，还支持增强程序功能的各种各样的外接程序。应用程序兼容性测试是为了保证所测程序和所有的文件格式及组件之间的互用性能够继续正常运作。

应用软件库

微软的许多团队都拥有专用于应用程序兼容性测试的应用程序或组件库。Windows 应用程序兼容性团队的应用程序库包含有数百种应用程序。其他团队例如 Office, Windows CE, .Net Framework 等也都有自己的应用程序库。

应用软件大“领养”

Windows 95 是微软的第一个面向普通用户的 32 位操作系统。当时，Windows NT 是 Windows 的“企业”版本，所以只支持相对少量的应用程序。而 Windows 95，一方面按预期要支持新的 32 位应用程序，另一方面要支持成千上万的 16 位 Windows 3.1 遗留的应用程序。我们当时全力推进对多媒体应用程序的支持，结果在 Windows 95 中运行此类应用程序时，我们发现了“成吨的”缺陷。在进度已经延后的情况下，我们需要马上采取行动。正是在那种压力下，微软历史上的一个经典故事问世了：这也就是最初在 David Cole 的“微软自述”(Inside out Microsoft-in our own words) (Warner Business Books, 2000)中叙述的如何完成这一任务的故事……

“在 1994 年节日期间，大量的运行于 Windows 3.1 上的多媒体应用程序发行出来了。我们发现这些应用程序中，很多与即将发行的 Windows 95 不兼容。这也是我们不得不推迟发行的原因之一。为了不将时间浪费在从供应商的订货的物流过程中，我想了一个歪主意，即驾驶我的卡车直接到本地玩家 (Egghead) 商店，将他们所有的多媒体应用程序每个买一份。

因为我们需要尽快的测试，我们的想法是将这些应用程序分给员工，让他们在家或上班时测试。然后，作为报酬，他们可以把这些软件留作自用。

我们一干人等驾车来到玩家商店，开始从每一种应用程序中抓起一个，然后堆放在收款机前。那里的三名营业员高兴坏了，想知道我们为什么要这样做。听了我们的解释后，他们激动得眼睛都瞪圆了。一个营业员开始为我们结帐，但是收款机总是当机。在连续三次输入所有的物件，连续三次当机后，营业员推测出当机大约发生在 10000 美元左右。因此他决定每次在 7000 美元左右即停止，用我们的信用卡结算，然后再进行下一批，花了很长时间。我想总数大约是 20000 美元。

我们将所有东西装在卡车车斗里，盒子填满了整个车斗。我将卡车倒到了 5 号楼的前门。”

回到主园区，我们召集一些志愿者将像小山样的一堆软件卸载到自助餐厅。然后给整个

产品工程团队送出了电子邮件宣布新的“应用程序大‘领养’”项目。员工们迅速从四面八方聚集到自助餐厅，漫游在桌子之间，设法从桌面上杂七杂八的数百个软件中选择一个。然后要签一个“合同”，答应报告缺陷，或在内部试用新操作系统上运行成功的消息。所有的应用程序在几小时之内就全被“领养”了。

这一活动很快就引出了数十个缺陷和许多成功案例的报告。员工们每天或每星期在更新了他们计算机的操作系统版本后，继续使用他们“领养”的应用程序。当 Windows 95 发行时，绝大多数 Windows 3.1 的应用程序已能毫无问题地在 Windows 95 上运行了。所有这些应当归功于 Beta 测试人员的成功测试和“应用程序大‘领养’”活动的开展。

应用程序检验器

工程师测试应用程序兼容性使用的一个关键工具是应用程序检验器（Microsoft Application Verifier - 微软的代码验证工具）。Application Verifier 是用来在运行时积极检测用户模式下的本机用户模式应用程序（native user-mode applications），以找出由于常见的编程的错误而产生的潜在的兼容性问题。它可以检测出诸如应用程序不正确地检查 Windows 版本，越权使用管理权限及其他数十种微妙的编程错误。Application Verifier 还能用来查出诸如内存泄漏、内存损坏或者无效句柄用法等其它好几个类型的缺陷。Application Verifier 具有可扩展性，常被用于许多其他的故障注入（fault injection）的场景。

Application Verifier 的插件，例如 Print Verifier（打印检验器）是被用于测试和核实子系统、打印机驱动程序和打印应用程序的。Print Verifier 可检测出诸如无效打印机句柄用法、打印函数的不正确使用，以及在打印机驱动程序中错误的函数实现。为其他驱动程序子系统的开发的类似插件也是通常可见的。

Application Verifier 是在调用“真正”的函数前，通过挂钩几个核心视窗函数及加入额外的检查达到检测目的。例如，当被测试的应用程序被加载时，Win32 应用程序接口（API）的 CreateFileA 方法的地址将被一个内部 Application Verifier 函数替换，该函数将触发一系列的测试，如图 11-6 所示。如果其中任何一个测试失败，失败将被记录在日志（log）中，如果调试器已经安装，调试中断点（break point）也许会被触发。关于 Application Verifier 的更多信息可在以下的 MSDN 网页中找到：
<http://msdn2.microsoft.com/en-us/library/ms807121.aspx>



典型的应用程序使用Win32 API的方法。应用程序调用从视窗函数库导出的函数



Application Verifier 产生一个“壳（shim）”截住从应用程序来的指令。在完成一定数量的测试或修改以后，shim 调用原先调用的窗函，然后在返回到调用窗函前，再有一次机会审查返回值。

图 11 - 6 Application Verifier结构图

吃我们自己的“狗食”

“在微软，当我们谈到‘吃我们自己的狗食（ Eating our Dogfood）’时，我是指，‘在你让任何人吃你的狗食之前，你必须自己先尝尝。’” - 史蒂夫·鲍尔默于 2003 年 10 月 21 日 Office 发行大会上

有时，确定用户将怎么使用一种应用程序的最佳的方式是自己去扮演用户。在微软，“吃我们自己的狗食”（每天使用我们正在开发的产品）是每个产品团队的可用性和兼容性战略的一个关键部分。Windows 团队的每个员工每天或每星期都必须安装新的版本，使用正在被开发的操作系统来开发操作系统。Visual Studio 团队用 Visual Studio 开发他们的产品。Office 团队使用 Office 最新的版本编写规格说明，提供客户介绍，甚至收发电子邮件。当我在 Windows CE 团队工作时，我的办公室电话、我的手机和我的家庭无线路由器上运行的全都是测试版本的操作系统。

员工使用测试版本的缺点之一是将来的真正用户可能跟开发员工使用产品的方式不同。例如，如果工程师是 Microsoft Word 的初期版本的唯一的用户，而他们只用它来编写规格说明和设计文件，很可能其他类型的用户会遇到问题。Beta 测试工程师（测试产品预发布版本的外部测试人员）可以部分的解决这个问题，他们对帮助微软的产品开发非常有价值。对于微软 Office 应用程序，公司的一大群各式各样的非工程员工的参与也有很大的优势。众多的微软律师、会计师和许多其他非工程员工在产品发行之前的许多个月都使用 Office 的测试版本。

大量的“狗食”（A lot of Dogfood）

Visual Studio 的团队基础服务器（Team Foundation Server），简称 TFS，是 Visual Studio Team System 的一个综合性的团队协作服务器，它包括缺陷跟踪，项目跟踪，源代码控制和版本管理等特性。多年来，微软员工使用各种各样的工具来完成这些同样任务。但是 TFS 最初在开发部以外的团队的接受速度是缓慢的。开发 TFS 的团队对其他内部团队是如何使用他们的

产品很感兴趣。为此，他们出版了一个 TFS 在全公司内部使用的月度更新报告。

截止 2008 年 3 月，多于 11,000 个不同用户积极地参与了 TFS 项目。这些用户工作在几乎 300,000 个项目上和签入了近 2.4 亿个文件。任何公共的成套工具的使用都是有利于微软的，而 TFS 的客户端和服务器部分的 Dogfood 过程使得这个产品的最终用户体验大大改善。由于 Dogfood 对微软产品质量产生了如此的重要影响，我们将这个概念也延伸到我们的服务性软件。Windows Live Mail 团队为一组客户推出了一套 dogfood 版本的服务性软件，这些客户知道他们得到的软件不如其他用户的稳定，但是他们很高兴有机会能给我们提供反馈，让软件工作得更好，这也就是为什么他们愿意经历试用的烦恼。

可达性测试

“可达性是指消除障碍，让每个人都能享受技术的益处。”

— 史蒂夫·鲍尔默

可达性是指为每个人提供接触信息和工具的相等机会，这些信息和工具是他们完成每天工作所必需的。这包括从复制文件到浏览网页到产生新的文档等所有事情。软件可达性的根本点是让用户有一种感觉，他有能力创造和维护应用程序、网站或者文件，并且有能力与之互动。

微软的最大的客户之一，美国联邦政府，要求其信息技术必须考虑到所有用户的需要。1998 年，Rehabilitation Act (修复法)第 508 部分(www.section508.gov) 成为法律，它为伤残人员消除了障碍，提供了机会。微软作出承诺支持第 508 部分的内容。一个内部的可达性事业部与工程团队、技术支持公司及伤残人保障组织一起工作，以保证所有软件公司开发的软件能被伤残人使用。

可达性是由几层特别的特性定义的--其中每一个都是程序可达性的重要部分。其中任何应用程序都必须测试的特性包括：

- 操作系统的设置：这些设置包括大字体、高 DPI、高反差界面风格、光标闪烁速度、粘滞键(StickyKeys)、过滤键(FilterKeys)、鼠标设置子键(MouseKeys)、串行键设置子键(SerialKeys)、切换键设置子键(ToggleKeys)、屏幕分辨率、自定义鼠标设置及从屏幕键盘的输入，如图 11-7 所示。

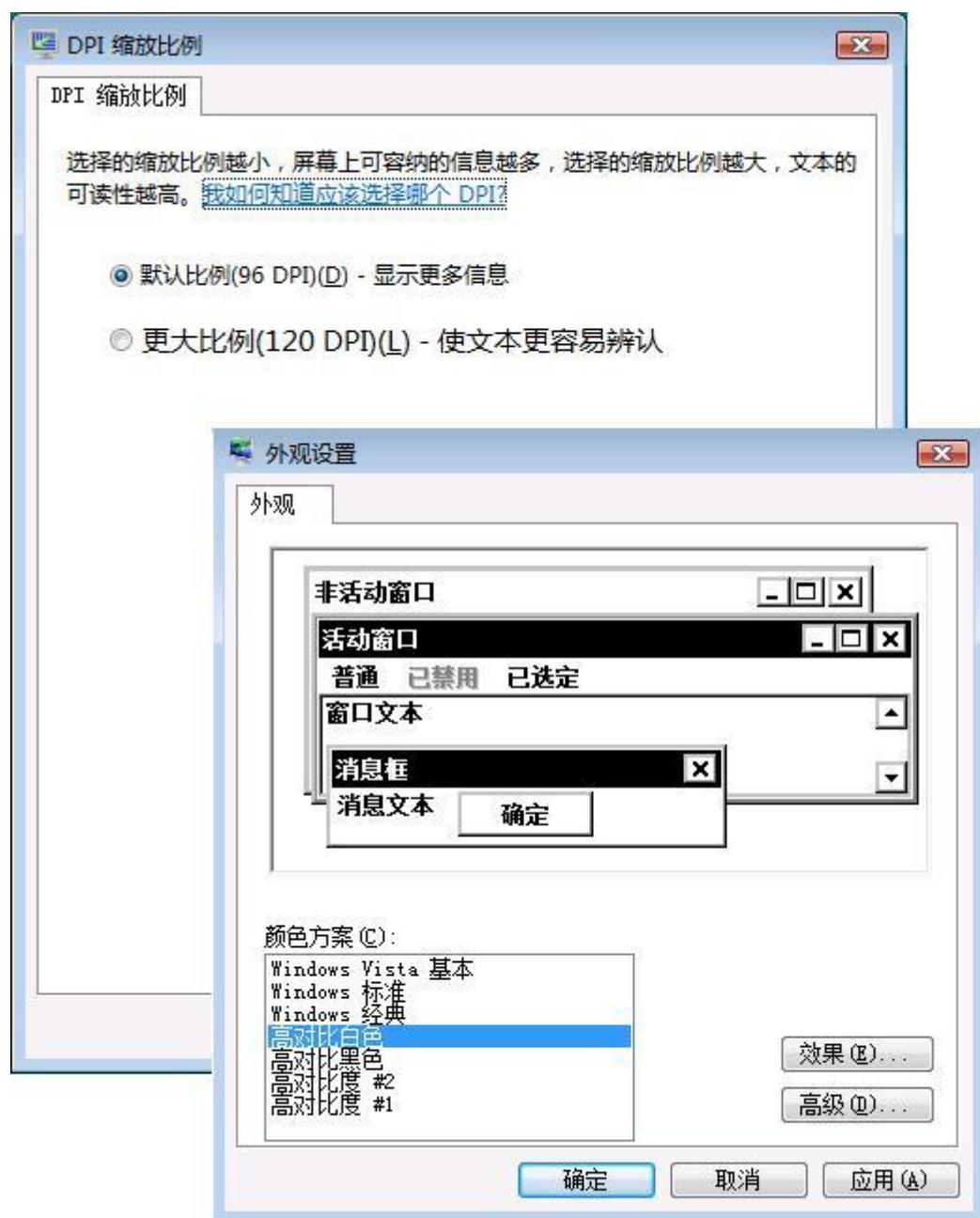


图 11 - 7 在 Windows Vista 选择可达性特性

- “内置”可达性特性：这些特性和功能包括 Tab 键顺序、热键和快捷键。
- 编程访问：这包括实施 Microsoft Active Accessibility(微软可达性技术)(MSAA)，或者使用任何其他能实现可达性特性的相关对象模型。
- 可达性的技术工具：可达性测试的一个重要方面是使用可达性工具去测试应用程序，这些工具包括，屏幕阅读器、放大镜、语音识别或者其他输入程序。微软保持着一个对所有员工开放的可达性实验室，实验室充满了安装着诸如屏幕阅读器和盲人识字系统阅读器等可达性软件的计算机。

可达性角色

角色(Personas)是用来代表某类用户和他们如何使用我们产品的虚构人群。使用角色有利于产品团队集中精力设计和开发适合这些用户的功能。在微软，多数产品团队在产品周期的早期即找出和确定角色，并在产品的整个生命周期中始终参照这些角色。

产品团队也许会为他们的产品创造三个到五个(或更多)的角色，但是许多角色被微软所有产品使用。为了帮助团队更好地理解特殊类型的伤残客户是怎样使用计算机及怎样和软件交互的，我们创造了十多个角色。例如，盲人用户角色中包括了有关屏幕阅读器使用的信息(屏幕阅读器不能阅读位图图像和 custom controls(自定义控制)上的文本信息)和他们期望的导航功能。同样，为聋子和听力受损用户的角色能帮助工程团队记住声音应该能被自定义(customizable)，音量应该是可调整的，并且对来电显示和语音邮件等功能，应该为用户提供替换的方案。

鼠踪消失

在鼠标与键盘之间，我是几乎每项任务都倾向于使用键盘的那些计算机用户之一。对于我，快捷键不是可达性功能；他们是生产力功能。如果我可以让我的手不离开键盘，我工作的速度会快得多。

在我职业生涯的早期，我曾经在测试我们的一个应用程序时遇到了没有鼠标几乎无法使用的产品部分。我唯一能找到的达到几个控制的方法，是仔细地安排一系列的 Tab 键和箭头键组合。我知道这个问题是重要的，因此我提交了一个缺陷然后就下班回家了。

我第二天早晨惊奇地发现负责项目的主管已解决了缺陷。我解释了对可达功能的需求，但是他向我保证它已经是“足够可达了”，并且我们有更大的问题急需关注。我不好意思地点了点头，并且要求他在关闭该问题之前设法至少不用鼠标使用该功能一次。他嘟哝了几句就回他的办公室了。几天以后，其他测试人员也开始报告相似的问题，但是主管仍然未尝试不用鼠标使用该功能。我觉得是尝试一种不同的战术的时候了，因此，在我那天晚上回家之前，我勇敢地走进了主管的办公室，拔去了他的鼠标，并且留言说一旦我们对项目的可达性目标达成协议，我会将鼠标归还。

第二天早晨我来得很早，怕主管并不喜欢我的幽默。我们几个员工偷偷地注视着办公室内的主管，很高兴发现看他微笑着阅读了留言。我们各自回到自己工作，并且等待着。不到 1 小时以后，我与主管会面再讨论可达性问题，并在不用鼠标的前提下把应用程序过了一遍。由于这次讨论，我们最终修复了大多数键盘可达性问题，从而开发了一个质量更好的产品。

可达性测试

使用角色是一种重要方法，也被微软用于测试的许多方面。有些可达性测试方法适用大多数应用程序，应该成为任何测试方法的一部分。其中一些测试方法的要点如下：

- 遵守系统范围的可达性设置：核实应用程序不使用自定义的视窗颜色、文本尺寸，或者其他可由全局可达性设置的元素。
- 支持高反差界面风格：核实应用程序可以用于高反差界面风格。
- 尺寸事关重大：固定的字体尺寸或小鼠标目标是两个潜在的可达性问题。
- 注意音频功能：如果应用程序使用音频报告事件(例如新的电子邮件)，应用程序也应该允许非音频报告，例如光标变动。如果应用程序包括音频讲解或视频介绍，也应提供文本副本。

- 能够对 UI 元素和文本进行编程访问：这听起来似乎是可测性特性(能够用自动化工具)，通过 Active Accessibility 或 .Net 的 UIAutomation 类的编程访问是屏幕阅读器及其他类似可达性功能运作的主要方式。

微软 Active Accessibility (MSAA) 测试工具

Active Accessibility 软件开发工具包(SDK) 包含了几个测试应用程序可达性的有价值的工具，特别是实现了 MSAA 的应用程序或控制。

- Accessible Explorer 程序允许你审查对象的 IAccessible 属性和观察不同的控制之间的关系。
- Accessible Event (AccEvent) 监视工具允许开发员和测试工程师确认应用程序的用户界面(UI)元素在 UI 改变时引发适当的 Active Accessibility 事件。当 UI 元素被调用、选择、或者有状态变换，或者焦点变动时，在 UI 上有变化发生。
- Inspect Objects 工具允许开发员和测试工程师审查应用程序的用户界面(UI)项目的 IAccessible 属性值及导航到其他对象。
- MsaaVerify 工具核实控制的 IAccessible 接口的属性和方法是否符合 MSAA 规格描述的指南⁴⁰。MsaaVerify 的可执行文件和原始代码均可在 CodePlex (<http://www.codeplex.com>) 找到。不管你是为了满足政府的规则，或设法让更多用户喜欢你的软件，可达性测试都是至关重要的。

可用性测试

可用性和可达性很相似，但我们所考虑的这两者有一个很大的区别。可达性是指任何人都能够使用用户界面，而可用性是指用户能不能很容易的理解和与用户界面互动。可达性特性能够带来更高程度的可用性，但可用性包含更多。有用的文档、工具提示、容易找到的功能，还有很多其他的标准都对提高软件的可用性有帮助。

当测试一个应用程序的用户界面时，可用性测试包括确认应用程序的功能可以被找到，并能够象用户期望的那样工作。同样，当测试应用程序接口（API）或对象模型时，可用性测试包括确认所发布的功能在被编程任务调用时直截了当，并且完成所期望的功能。可用性测试还包括确认文档正确和相关。

可用性实验室

很多微软的产品团队都利用可用性实验室。测试工程师通常不直接参与实验过程，但他们确实利用实验的数据来影响他们的可用性测试方法。比如，尽管实验可能会发现设计问题，需要项目经理或是开发工程师来解决，但测试工程师常常利用应用程序的使用方式来建立不同场景，或是根据使用模式来权衡特定领域的测试。当然，还有很多其他的因素来决定关于用户怎样使用一个应用程序的模型建立（其中一些技巧和工具会在第 13 章“用户反馈系统”中讲到）。

在微软正式的可用性实验是在一个格局类似图 11-8 的实验室中做的。参加者花大概两小时使用一个应用程序，并通常被要求完成一些目标任务。

⁴⁰ http://msdn.microsoft.com/library/en-us/msaa/msaapndx_2a05.asp

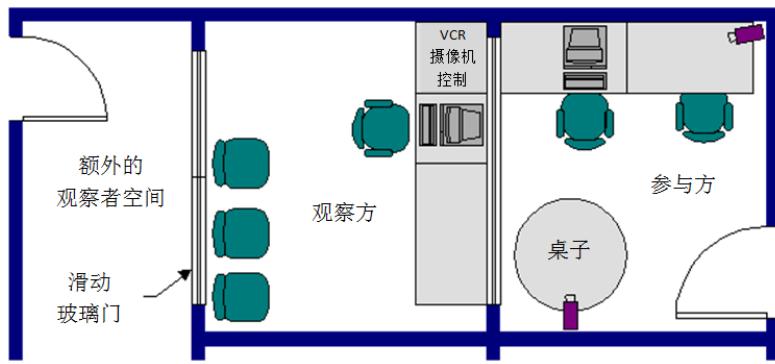


图 11-8 可用性实验室格局

这些实验的目标可能不同，但都试图解答这些常见问题：

- 用户的需要是什么？
- 什么样的设计能够解决用户的问题？
- 用户需要做哪些工作，他们能不能很好地解决问题？
- 用户如何学习软件，以及保持他们对软件的技能？
- 软件用起来是否有意思？

当我们跟团队提到可用性测试时，我们总是分享一个心得，那就是，可用性测试最终总会发生的。你可以选择把它作为测试的一部分来做，要不就得留给客户来做。尽早的可用性测试不仅会让软件在用户眼里看起来更成功，还能对减少你所收到的产品求助电话数目产生极大的影响。

提示：微软在全球有多达 50 个可用性实验室，而且每年有 8,000 多人参加微软的可用性试验。

在微软，可用性实验在持续增长和进步。利用了新技术例如眼球跟踪、网上远程以及游戏“试玩”等等的可用性测试都在进步。

安全测试

近年来，安全测试已经成为微软文化中不可缺少的一个部分。对于恶意软件和间谍软件的反应使得微软的所有工程师都具备了安全观念。安全测试是如此重大的一个主题，它更适合用一整本书来探讨，而不是一章里的一小节。实际上，我在一家网上书店略微一扫就找到不下六七本关于安全测试的书，包括微软员工所写的书，如：Gallagher, Landauer, 和 Jeffries 的“寻找安全缺陷（Hunting Security Bugs）”，以及 James Whittaker（他是知名的软件安全专家）的“怎样破解...（How to Break...）”系列丛书中的两本关于安全的书。另外，很多微软测试工程师的书架上都有 Howard 和 LeBlanc 的“编写安全代码”一书。如果你想知道更深更广，这些或其他书中的任何一本都会有益。

测试工程师在安全测试中的作用不只是找到缺陷，更要决定这个缺陷能否和能怎样被利用。在接下来的几小节里我们将讨论安全测试的一些主要的方法和技巧。

威胁建模

威胁建模审查应用程序结构以找到潜在的安全威胁和弱点，是一有组织的集体工作。威胁建模在微软被广泛应用，测试工程师是威胁建模过程中活跃的参与者。测试工程师一般对输入校验，数据处理，和会话管理比较熟悉，这促使他们在检查应用程序的潜在安全问题时，成为主导力量。

威胁模型—就象本章讨论的其他概念一样—在程序设计时完成最佳。威胁模型就象功能计划或设计文档一样，是一种规格说明。而最大的不同在于威胁模型的意图是找出一个应用程序能被攻击的所有可能的方法，然后根据概率和可能的危害来排优先级。好的威胁建模需要分析和调研技能—这两种技能使得测试在这过程中很适用。关于威胁建模的更多信息，包括实例，都能在 Frank Swiderski 和 Window Snyder 所写的威胁建模一书中找到（微软出版社，2004）。

模糊测试

模糊测试是一种用来决定程序对无效输入数据会怎样反应的技术。一个简单的办法就是用十六进制的编辑器来改变程序所使用的数据文件的文件格式，比如，改变微软 Word 的.doc 文件的字节。在实践中的过程会比这样更具方法性。除了随机改动数据之外，模糊测试通常还包括将数据改变成更易暴露潜在的安全问题的样子，比如引发缓冲区溢出的形式等。模糊测试在以下的场合中也同样适用：数据库测试、协议测试或任何一种必须读取和解释数据的系统或应用程序中。

找出模糊漏洞

在 Windows Vista 的开发过程中，我负责管理微软 Windows Shell（用户界面）的模糊测试。这个问题最难的部分不是创建和运行模糊测试，而是找到 Shell 解析文件的所有不同方式。Shell 天生就具有可扩展性，Windows 的许多团队都对其做扩展，来改善用户处理 Windows Explorer 文件的体验。仅仅是文件解析器的质量和跨组织的代码拥有就很有可能造成测试漏洞。我们也意识到了这一点，为了解决它，我们基于解析器的类型（也就是，属性处理器，shell 折叠器，略图抽取器，等等）制定了详细的测试计划，并基于仔细的手工代码审计编辑了测试事项列表。

尽管我们和整个机构的设计师及测试主管一起评审我们的数据，我们还是意识到会遗漏一些东西。因此，我们开始监视缺陷数据库，看在 Shell 中的所有因为不正确文件格式造成的当机、死机、或大量占用内存。在 Windows Vista 产品周期后期，我们发现了一个在某一非自动测试下造成文件损坏的缺陷。我们立刻研究了这个问题，以了解为什么我们全面的模糊测试没有把它抓到。这才发现在跨组拥有权的一个属性处理器中有一个模糊漏洞！

后来弄清楚了，问题的根本原因在于一个不正确的假设，拥有这个功能的小组认为他们所作的基于 API 的文件模糊测试全面覆盖了他们的属性处理器，但实际上，由于模糊化底层的 API，他们有一小部分的代码没有被覆盖到。我们担心在这段非模糊的代码里可能还潜藏着更严重的问题，我们快速的发起了跨团队的努力，来提供我们漏掉的模糊覆盖。这个特殊的组件解析好几种不同的文件类型。我们中的两个人花了一周的时间，用一个实验室的所有机器来运行测试，来全面填补这个漏洞。这个组件的模糊化最终产生了六个当机缺陷，但在发布前都被解决了。

- Eric Douglas, 高级测试主管。

本章小结

功能测试极其重要，而用来有效的完成功能测试的许多技术和方法也同等重要。很多团队有时会忘记一点，那就是用户并不关心找到了多少个缺陷，或者有多少测试失败了，或是代码覆盖率。它们都是“测试食谱”中重要而有价值的成分，但最终来说，用户关心的是产品非功能性的方面。用户想要的软件必须既安全又可靠，能够轻而易举地完成他们想做的事；还要能方便地找到功能，而且响应速度足够快。

非功能测试与功能测试相得益彰，对于确定产品是否拥有高质量和能否发布起着决定性作用。非功能属性常面临的一个难题是很多方面在开发早期就需要充分地考虑，而真正测量这些属性要一直等到用户使用该软件才能开始。要解决这个难题，关键是要通过角色扮演或其他相似的机制，在各个测试的前沿倾听用户的声音。

第 12 章 其他工具

阿伦·培智

工欲善其事，必先利其器。一个好的木工需要拥有和使用几十种不同的工具，熟知它们各自的性能，然后针对特定的任务选用合适的工具才能有效的进行工作。又如同我们在电视节目中常见的破案人员，他们拥有各种五花八门的特殊仪器，每一种仪器都有其独特之处。针对不同的案情需要，就选用不同的仪器和检测方法，往往很快就能破案。以上这些工作都说明工具非常重要，我认为测试也是一样。只有当你具有出色的专业知识，同时还拥有合适的工具时，才可能成功。

测试工具就是可以辅助测试人员提高测试效率或测试效果的应用程序。微软的测试人员在测试过程中会使用到种测试工具。在这中，既有用于运行测试实例的，也有监测系统环境的，跟踪测试进度的，更有各种各样用于其他测试目的。测试工具有很多种，前一章节已经提到了几种微软常用的工具，本章将介绍更多的实用工具。

代码改动量

改动（churn）是一个特定术语，指的是在一段时间内，一个文件或模块中的代码变化的总量。代码改动有很多种计量方法。以下列出最常用的一些计算方法。

- **修改的次数:** 总计此文件被修改了多少次。
- **增加的代码行数:** 在某一个特定时间点后，总计有多少行代码被添加入此文件
- **删除的代码行数:** 在某一段选定的时间跨度，此文件中总计有多少行代码被删除。
- **修改的代码行数:** 在某一段选定的时间跨度，在此文件中总计有多少行代码被修改过。

微软的 Visual Studio Team System 定义了“改动项总数”（Total Churn）指标，该指标是增加的代码行数，删除的代码行数和修改的代码行数这三者的总和。详见表 12-1.

表 12-1 Visual Studio Team System 代码改动指标实例

文件代码行数	改动总计	修改的代码行数	删除的代码行数	增加的代码行数	改动项总数 (Total Churn)
857	161	0	0	161	161
899	359	3	178	178	161
932	72	2	35	35	161
946	16	0	0	16	177
合计	608	5	213	390	177

类似于第七章“用代码复杂性分析风险”中讨论的“复杂性”（complexity）指标，使用“代码改动”指标可以显示出软件中哪一部分出现缺陷的可能性较高。从某种意义上说，这是一个非常直观的指标。除了为软件添加新功能需要编写新的代码，代码改动主要是由于需要修正已知的软件缺陷。相当比例的缺陷修复工作并不能真正解决问题，或者会引发新的问题。

这两种情况都需要进行再次修复，或者是针对原有的问题，或者是针对新发现的问题。通常，这种情况还会反复发生。当代码非常复杂时，往往需要进行多次的反复才能修正所有已知的缺陷，并且确保没有新的故障发生。（然而从概率上而言，该段代码的代码改动指标将会非常高，这也就意味着其中还可能存在更多的缺陷）。

应该记住，“代码改动”只是一个有警示的指标，如果一个产品具有很高的“代码改动”值，并不总是意味着那个产品存在很多很多问题。它只是提醒您产品某些部分发生了大的变化，您可能需要更仔细地检查那部分的代码。

代码改动的研究

微软研究院的研究人员使用了几种不同的改动指标来研究软件缺陷和代码改动之间的数量关系。这些改动指标包括被修改的代码行数，删除的代码行数以及被修改的文件个数。研究员先计算出微软 Windows Server 2003 源代码的代码改动指标，又从相应的缺陷管理系统中获得了 Windows Server 2003 软件缺陷数据，通过分析，他们发现在这两者确实是相互关联的。

随后，研究人员随机选取了占系统总量三分之二的二进制目标文件，获得其相应的代码改动指标。使用这些数据，他们建立了一个缺陷预测模型，这个模型成功地预测了剩下的三分之一文件中缺陷的数量，这个结论可以说具有很显著的统计学意义。最后，研究人员还试图将模型用于区分“易出错型”和“不易出错型”文件。使用同样的方法，研究员随机选取三分之二的目标文件来建立这个文件分类模型，剩余的三分之一的文件被编入对照测试组。结果表明，当使用文件分类模型来对测试组中的文件进行分类时，百分之九十的文件都可以被正确地归类为“易出错型”或“不易出错型”⁴¹。

基于此研究，很多微软团队开始统计代码改动指标，并使用该指标来判断是否应该重新设计某些组件，或者用它来评估在产品的开发后期修改代码的风险。

一切尽在掌握

源代码控制管理（跟踪源代码的所有变化）对于开发人员是非常重要的，在微软，它对测试人员也同等重要。在软件行业几乎所有的开发团队都要使用某种源代码管理（Source Code Management-SCM）系统，微软开发团队也不例外。在某种程度上，每个微软的测试团队不仅使用了传统意义上的源代码管理，还会把它应用于一些特定的测试任务。

追踪变更

如同通常使用的源码管理系统，测试团队进行源代码控制管理的主要目的是跟踪一些测试工

⁴¹ Nachiappan Nagappan and Thomas Ball, 使用相对代码改动指标来预测系统缺陷密度（Use of Relative Code Churn Measures to Predict System Defect Density）(Association for Computing Machinery, 2005), <http://research.microsoft.com/~tball/papers/ICSE05Churn.pdf>.

具代码的修改，以及跟踪自动化测试代码的修改。一些测试工具只被当前测试团队使用，另一些则会被整个微软公司内很多测试团队使用。当越来越多的测试小组选用同一个工具，对那个工具进行代码修改跟踪就显得越发重要。因为只有这样才可以方便地找到由于这些更改而引起的问题。这里所说的源代码控制管理和通常应用于软件开发的源码管理系统的最大差别就在于“使用者”的不同，后者是由开发人员使用，而前者是由公司的某一个测试人员或测试团队在使用。

在测试自动化或编写测试实例中跟踪代码更改也很有帮助。当测试团队使用了源码管理系统，不仅可以跟踪在整个产品周期内所有的测试实例的变更，还可以在任意一个时间点上创建或者重建出当时源代码的完整状态，甚至包括当时相应的文档信息。最通常的做法是在每到达系统的一个重要开发里程碑时就创建一个完整的系统快照（snapshot）。一个普遍的例子是，当产品完成要发布时，就会创建一个包含所有的源代码和所有的测试代码在内的系统快照作为留档。有了这些测试代码，产品维护小组（sustained engineering team）就可以更放心对产品进行更改。这和产品开发人员编制单元测试代码（unit test）一样，都是使用回归测试来保证新的修改没有破坏现有的产品功能。产品维护小组会使用这些保存的测试代码来验证他们所做的更改没有影响现有的功能，同时评估这些修改是否有可能会在系统的其它部分引发一些新问题。

什么改变了？

在我童年时，我家订的报纸上有连载亨利·博尔蒂诺夫（Henry Boltinoff）的漫画“大家来找茬”。就是两张上下并排的图片，那两张图片看上去好像完全一样，但是图片中间的文字提示读者可以“在上下两幅图中找到至少 6 个不同之处”。通常我可以在几分钟内就找到所有六个不同点。基本上我就使用一招：我通过对第一幅图的每个物品或区域和第二幅图的相应部分进行单独比较，比较它们的形状，大小或者其它一些属性，先比较一个物品然后再比较下一个物品如此循环直到所有的区域都被比较过。有时候，那些物品的某一部分是不同的，又有时候，我则需要观察一个较大的区域来找到不同点。练习多了，我找差异的速度越来越快，准确性也越来越高。最后，我每次都可以在几秒钟内就找到所有的六个不同点。

同样的，作为一个软件测试人员，我常用源码管理系统和文本比较工具（diff tool—用于比较两个文件内容差异的工具）一起来定位出错的代码。图 12-1 是使用文本比较工具比较两个文件的结果。源码管理系统记录了所有代码的更改历史，所以当测试人员发现一个从前没有问题的功能突然有问题了，他就可以通过查找源码管理系统来缩小问题代码的可能发生范围。源代码控制管理可以显示出在任一时间段内，在一个文件、模块、功能或整个应用程序上发生的所有代码变更。如果测试人员发现，在两周之前的测试中，某个缺陷并不存在，那他就可以很容易的找出所有在这两个星期内被更改过的代码，然后逐个检查哪个改动可能会有问题。值得注意的是，在有些团队中，这部分工作也可能是由开发人员来承担。（下文中“测试工作的职权范围”部分对此有更详细的阐述）

The screenshot shows a Visual Studio window comparing two files: 'main.cs' (C#) from '\$/代码实例/测试库/测试库/main.cs;C10 (服务器)' and 'main.cs' from 'C:\测试工具\测试共享库\测试库\测试库\main.cs (本地)'. The title bar indicates the comparison is between the versions from March 1, 2009, at 11:56 AM and 11:58 AM. The left pane shows the original code with several lines highlighted in blue, indicating they have been deleted. The right pane shows the modified code with new additions highlighted in green. The bottom status bar shows tabs for '已删除文本' (Deleted Text), '已更改文本' (Changed Text), and '已插入文本' (Inserted Text), with '第 1 行, 第 1 列' (Line 1, Column 1) selected.

图 12-1 在 Visual Studio 软件中比较两个文件

需要被严密监测的并不局限于源代码的变更，同样重要的是还要对所有规范说明和所有辅助文档进行控制管理。有很多程序可以被用来跟踪文档的变化。比如本书在出版过程中，就是先由作者著书，然后审稿人审阅，最后编辑定稿。每一个阶段，阅稿人都可能会作些修改或给出一些修改建议。所有的这些改动都在微软的文字处理软件 Office Word 中被记录下来，这样大家都可以看见哪些部分被更改了，更改的进程如何，还可以查看到关于为什么要更改某些用词，哪些内容需要进一步修订的各种相关讨论。

即使阅稿人没有打开 Word 中审阅修订功能，Word 还是可以用来比较两个文件的内容，参照图 12-2。这样如果要比较统一文档的不同版本之间差异就非常方便。当有多个审稿人一起修改同一个文档时，审阅过程也变得更便捷了。

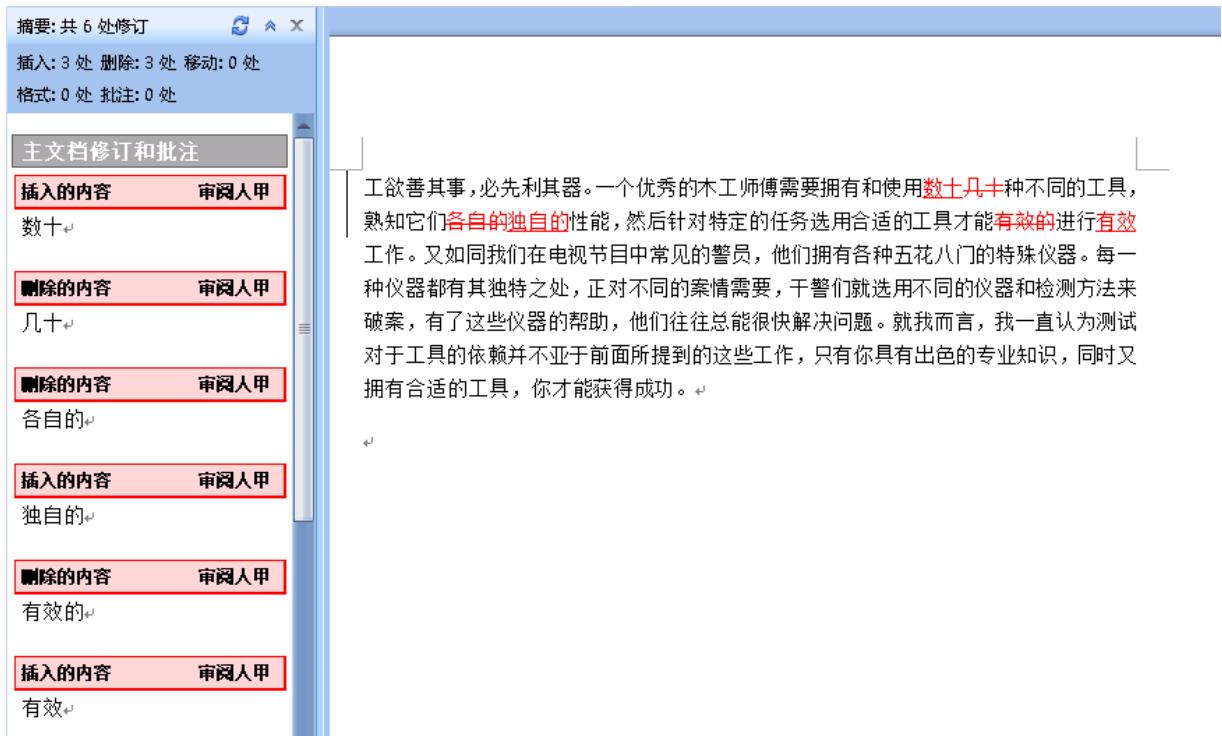


图 12-2 使用微软 Word 来比较文档

测试工作的职权范围

当我在微软讲授“测试入门”和“软件调试”课程时，常常有测试人员问我“关于程序调试，我要不要去做，要做多少？”。有趣的是，开发人员也会问“我应该期望测试人员做哪些调试程序的工作？”。对这两个问题，我的回答都是，“没有绝对的答案，这要看具体情况来决定”

调试程序就好像侦破疑案。调试程序时需要检查被调试程序各个变量的实时状态，检验日志文件，翻阅代码改动的历史情况，这些都是为了找到导致问题发生的根源。对于很多新的测试人员来说（对有些开发人员也适用），在“发现缺陷”和“修复缺陷”中找到一个合理的工作分界点不是一件简单的事。很多测试人员在勘查和查找缺陷来源时并不比开发人员做的差（有时甚至比他们做的还好）。微软的有些团队里，测试人员经常调试程序找到出错的文件，代码行以及那次代码签入（check-in）导致了这个缺陷。而某些团队里，测试人员只要报告软件缺陷，并不需要做很多勘查工作。几乎所有的微软测试人员都拥有深入调试程序的技能，至于他们具体需要做多少调试工作，一般则取决于项目的紧迫度和大家的心理预期。

如果测试小组缺少人手，项目进度落后，或是工作被其它的一些关键资源制约，那就要考虑是否值得要求测试人员花时间在缺陷勘查工作上。换而言之，有时测试团队根本没有时间彻底地调试程序和除错。满足大家的心理预期在这里就显得更为重要。我知道很多开发人员不希望测试人员调试他们编写的代码，我也认识很多测试人员他们不想调试别人写的代码。我个人并不赞同这两种情况。当有人问我到底某项工作该谁来做的时候，我都鼓励相关测试人员和开发人员（有时发生在整个测试团队和开发团队之间）互相回答这两个问题“你觉得你自己可以做些什么”，“你希望我帮你做些什么”。这两个问题对于解决工作关系问题非常有效。对于测试人员和开发人员，理清这两个问题可以澄清从单元测试到测试策略等等很多重

要的议题，当然了，关于缺陷报告和程序调试该由谁来做，该做些什么的问题也可以达成一致共识。

为何改变？

有时候我会比较原来的源代码和修改后的源代码，我可以看见哪里被修改了，可是没有办法理解为什么那段代码要被修改。

```
=====
--- math.cs;9 (服务器) 2008年5月6日 下午 5:24
+++ math.cs;9 (服务器) 2008年5月6日 下午 7:25
*****
*** 20,26 ****
}
else
{
!
    return value;
}
--- 20,26 ---
}
else
{
!
    return value * 2;
}
=====
```

在上述代码中，我们可以很方便地找到被修改的部分--函数的返回值变成了原来的两倍。可是为什么要修改函数的返回值呢？测试人员可以通过源码管理控制系统提供的很多重要信息来顺藤摸瓜。一个有用的信息就是这段代码被哪位开发人员更改过，每次更改的原因描述，以及该开发人员还修改了其它哪些代码。当你对一段代码的改动有疑问时，因为你已经知道是谁做的修改，你就可以直接找他，或通过电话或电子邮件向他请教。

如果是对刚刚改动的产品代码有疑问，或者是对前不久才引入的修改有疑问，最有效的方法是直接联络开发人员。但是如果开发人员已经下班，或者已经转组，甚至于已经离职了呢？通常源码管理控制系统还存储着很多其它和代码修改相关联的信息，比如说开发人员当时添加的注释信息，被修复缺陷的编号或者是某个指向被修复缺陷的链接。在缺陷管理系统中，通过这个缺陷编号或链接，就可以查找出本次代码更改究竟修复了哪个缺陷。当代码被签入（commit）到源码管理系统时，当前的修改的工作代码会被合并到源码管理系统成为正式的代码。在签入表单上，修改代码的人员通常会被要求填入一些相应的信息，比如说该次签入会修复哪个缺陷（如果是添加新功能，那就要填入相应新功能的编号）。同时他还要填入其它一些信息，例如代码审阅人的姓名，再加上一些关于该次代码更改的简短说明，参见图12-3。所有这些信息对于测试人员和开发人员勘查缺陷都非常有用。

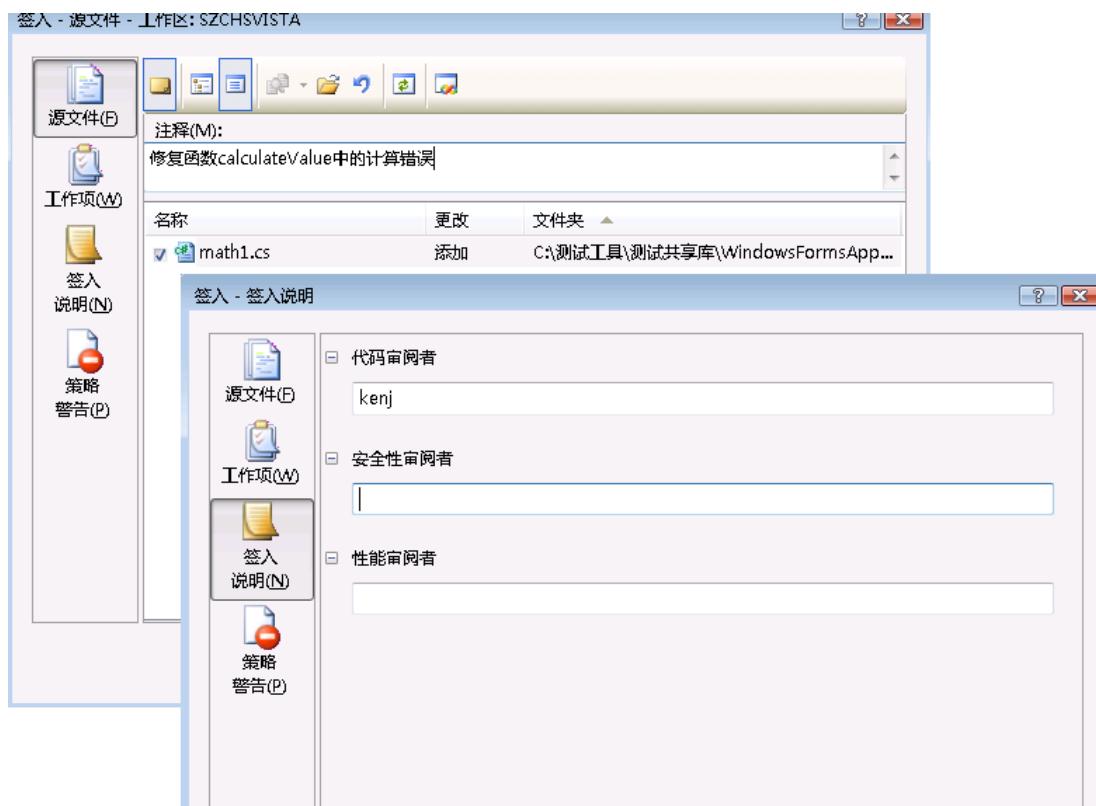


图 12-3 Visual Studio Team System 中的代码签入窗体

集中型的源代码管理控制

在微软公司，各测试团队在源代码控制系统的使用上已经历了各个阶段，也在不断的完善中。我早年在微软的时候，可以说我所在测试团队的源代码控制至少是做得非常不正规的。大多数测试小组都使用源代码控制系统来管理各自的测试代码或数据文件，但是每个小组的不同成员可能使用的是自己单独的源代码控制服务器。使用单独的并由自己维护的源码控制服务器的确可以保证测试源代码被按时备份并且所有的代码修改都有据可查。如果你并不想其他团队查看你们的源代码或也不愿意和他们共享所有的代码，各个小组成员使用自己单独的服务器也是可行的。在我的小组，每个测试人员负责编译自己的测试源代码，同时要负责把编译产生的二进制代码文件复制到同一个共享的网络文件夹中，这样其他的测试人员或者自动化系统需要使用它们的时候都能够找到。大多数情况下，这种方式都不错，但是若复制过程中出错了或是某人不小心删除了网络文件夹中的一个文件，那时就有问题出现了。

渐渐的，越来越多的小组开始把所有的源代码集中起来存放。整个团队的源代码都会存放在一台服务器上，使用同一个系统来管理，并在其上严格定义了文件的存放结构。现在大多数团队都采用把测试源代码和产品源代码存放在一起的方式，使用相同的服务器系统来保存它们。参见图 12-4。接着，“构建实验室”(build lab) -也就是专职负责每天生成当日版本的人或小组，会对所有的产品当前源代码和测试代码进行编译，然后将产生的测试二进制文件自动的分发到服务器上。

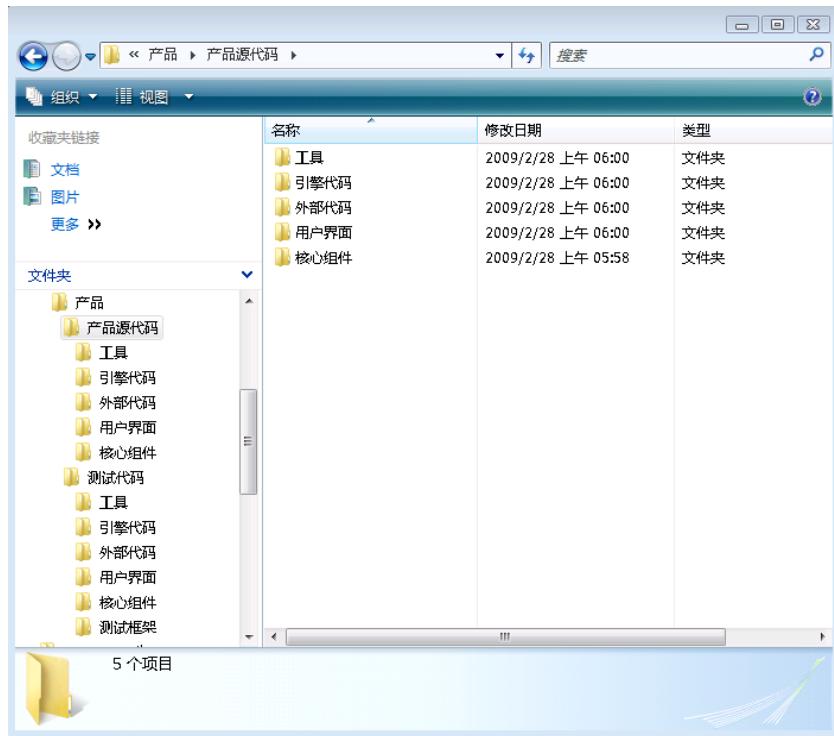
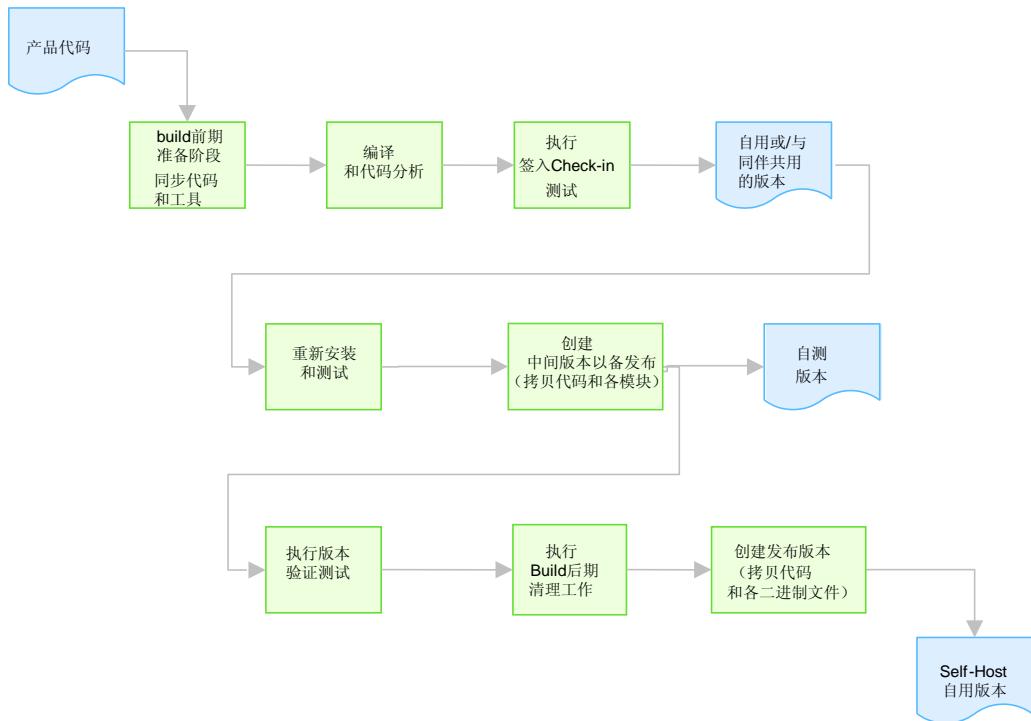


图 12-4 一个典型的源代码管理系统中产品代码和测试代码的存放结构

上图中显示出的代码结构有多个优点，其中最重要的是便于快速查找代码。如果一个开发人员想运行测试小组的某些测试代码，他很清楚去哪里找。同样，如果一个测试人员想更深入地了解他所测试产品的功能和该功能的具体实现，他也可以方便得找到相应的产品代码。当遇到因产品代码和测试代码不同步而导致的代码编译错误或产品生成失败时，“产品生成组”也可以快速的勘查具体原因。最后，如果“产品生成组”在同一个生成过程中既生成产品，也生成测试可运行文件，那么这个产品和测试可运行文件就可以共用同一个版本号。对它们使用同一版本号可以简化对最终生成的产品和测试目标文件的管理，一旦发现了新问题，同一版本号便于测试人员找到该问题最先在哪个版本上发生，从而进行更好更快的分析。

软件构建

软件构建以及由此衍生的很多相关的活动是微软每个小组每天工作的一个组成部分。源代码管理（Source control），缺陷管理和测试都从构建过程开始。



每日构建

对大多数组而言，整个产品的源代码每天至少编译一次。微软每天生成新版本的习惯已延续多年，Agile 社区也大力支持频繁的代码签入（check in），不断地集成，连续地构建。构建过程包括编译（将源代码转换为二进制格式的文件），链接（将许多二进制文件结合在一起），和运行应用程序所需的任何其他步骤：如构建安装程序和部署到新版本发布服务器上。

提示：Windows Live 部门的构建实验室(build lab)每周创建生成 6000 多个构建。

构建实验室的一天

一个典型的编译实验室生活的 24 小时。

下午 3 时左右构建过程启动。自动化的脚本先为构建过程准备好每台机器。所有残留的旧版本都将被删除，源代码将同步到最新的没有问题（last known good）的变化（changes）。对于中型软件产品，构建过程需要一到四小时的时间（大型软件产品，如 Microsoft Windows 或 Office 可能需要更长的时间）。

在初始构建过程中，源代码变成了该产品的二进制文件。接下来的几个小时内，更多自动化的脚本使用这些二进制文件，创建多个 SKU (库存单位的简写，这里指软件的版本类别) 的可安装产品——不同版本如 Professional (专业版)，Ultimate (旗舰版) 或者各种本地化的版本。虽然这是一个自动的过程，但构建团队将指定某人在晚上监视构建。构建监视者可能会定期检查构建的状态以确保他们正常进行，但在许多情况下，构建系统会自我检测，如果遇到错误信息，可以自动发送网页、电子邮件或文本给构建监视者。

有的错误会较早出现，有的错误则会较晚出现。这个过程中不时会发生一些随机事件：比如某台关键计算机意外关机或网络出问题。在整个构建期间，所有出现的错误都被及时处理，记录为用来跟踪的缺陷，发送给产品团队，或被构建监视者标记为已解决问题。

在清晨，构建团队成员开始工作，从前一夜构建监视者创建的摘要中，检查任何悬而未决的问题。如果还有任何阻碍（**blocking**）构建成功的问题，构建团队会找到出问题的地方和相关的负责人，提醒他立即提供相应的修补程序（**fix**）。一旦构建准备好了，生成验证测试（**BVT**）开始。如果发现缺陷，立即联系到相关产品的负责人并记录缺陷。

目标是在中午前生成构建，供各小组测试。如果在这个时候，仍有未解决的缺陷，需要决定是否推迟发布构建或那一天不释放构建。每日构建是软件开发与测试流程中重要的环节，所以微软是不会轻易作出不释放的决定。如果决定是尝试发布构建，会等待所需的修复程序和测试都完成，才能发布构建。当构建准备好发布时，文件从构建计算机复制到发布的服务器上，然后以电子邮件的形式发送到整个团队：包括构建的信息和所有已知的问题。

有时，每天的构建过程还包括一套冒烟测试（**smoke tests**）。冒烟测试是一种简单测试，以确保应用程序的基本功能工作正常。类似于借车试车时，在驾驶它逛整个城镇前，先在附近转转来检查明显的故障。测试小组通常运行一套冒烟测试（**smoke tests**）。更常见的是，这些是大家熟知的构建验收测试（**BATs – Build acceptance tests**）或构建验证测试（**BVTs - Build verification tests**）。这些术语的一些定义表示 **BAT** 要比 **BVT** 范围小一些，但在大多数情况下，这两种叫法是可以互相通用的。好的一组 **BVT** 确保每天的构建是可用于测试的。表 12-2 列出了几个 **BVT** 属性。

表 12-2 BVT 属性

BVT 属性说明	解释
自动化一切 Automate Everything	BVT 首先在每个构建上运行，然后需要每次运行结果相同。如果您的整个产品只有一个自动测试的套件（ suite ），它应是你的 BVT 。
测试一小部分 Test a little	BVT 测试并非包括所有功能测试。他们是用于验证基本功能的简单测试。 BVT 的目标是确保构建可被用于测试。
快速测试 Test Fast	整个 BVT 套件的执行时间应该只有几分钟，而非长达数小时。一个短的反馈循环能立即告诉您构建是否有问题。
报错恰到好处 Fail Perfectly	如果一个 BVT 失败，应该说是构建不适合进一步测试，必须立即修复造成失败的地方。在某些情况下，可以有暂时回避问题的替代方法，但所有 BVT 的故障都表明最新版本中出现严重问题。
广泛测试而非深	BVT 应大致包括该产品的各方面。他们肯定不应该包括每个细

入测试 Test Broadly – not Deeply	小的地方，而是应包括各功能的每个重要部分。它们并不（也不应该）包括一系列广泛的细小变化的输入或配置，应尽可能多地覆盖关键功能的主要使用场景。
可调试和可维护性 Debuggable and Maintainable	<p>在一个完美的世界，BVT 永远不会失败。但如果发现失败，则需要尽快隔离。从发现失败，找到原因，实现修复程序的过程必须尽可能迅速完成，即周转时间必须快。作为 BVT 的测试代码应该是在整个产品中最有效的，最易调试和维护的代码。</p> <p>好的 BVT 可以自我诊断，错误输出常常能指出确切原因。更好的 BVT 会进行自动源代码查找，标识最有可能导致错误的代码的改动（change）。</p>
可信 Trustworthy	你必须能够信任您的 BVT 。如果 BVT 通过，生成的构建必须是可以被用于进一步广泛测试的，而如果 BVT 失败，意味发生了一个严重的问题。这是必须坚持的原则，如果 BVT 不能做到这点，测试团队就无法信任 BVT 测试的效用。
关键 Critical	安排你最好，最可靠和最信赖的测试人员和开发人员创建最可靠和最可信的 BVT 。编写好的 BVT 并不容易，需要花费时间深思熟虑，才能充分满足此表中列出的其他准则。

示例：一个简单的文本编辑器如 Windows 记事本中的一个 **BVT** 套件可以包括以下各点：

1. 创建一个文本文档。
2. 编写一些文本内容。
3. 验证基本功能，如剪切，复制并粘贴到剪贴板。
4. 测试文件操作，如保存，打开和删除。

每天（或更频繁频密）的构建和 **BVT** 过程可以减少大的集成或全面的更改引发错误的机会。保持该产品天天成功的构建和运行是一个健康软件团队的关键。在微软，测试团队和产品团队一样都有每天构建的需要。同样原因，自动化测试的源代码和测试工具也每天构建，通常与产品代码一起构建。

中断构建（Breaking the Build）

每日构建至少确保编译错误（也称为构建错误）在代码签入（check-in）24 小时之内被发现。

编译错误出现的次数很少，但是一旦出现，就会中断工程流程，测试小组会因为没有构建而无法开始测试。最常见的编译错误也是最容易预防的：开发人员代码中的语法错误（syntax errors）。任何人只要编写过任何代码或运行脚本，都会犯过语法错误。缺少分号，错误输入的关键字或一个错误的击键都可以在编译该程序或运行该脚本时导致编译错误。这些错误本身是无可避免，但只有当带有这些错误的代码已 Check-in 至源代码管理（source control）中，问题才会出现。我知道没有任何编程人员会故意 Check-in 有问题的代码，但有时因为粗心，

这种情况也会发生。它们通常都来自于一些小的改动，而开发人员忘记重新编译本地源代码来检验一下。一种简单的预防方法可能是要求开发人员在 Check-in 源代码之前构建，但这可能很难强制实施，再者一些错误仍可能由于其他原因出现。

你中断了构建过程（You broke the build!）

在软件业内一直有个悠久的传统，谁的错误中断了构建过程就惩罚他带一个傻里傻气的帽子（译注：简称傻帽，这很有趣，与我们的俗称有些类似之处），在他的办公室门上贴条标语，让他给整个团队买面包圈，或以其它方式让大家注意到他的错误。我还知道有的团队：半夜给中断构建过程的开发人员家里打电话，让他们赶过来——马上来！其实只要大家更小心些并且在代码签入代码之前认真想清楚，很多麻烦可以避免。但有时大家会不经意犯错误。幸运的是，许多团队开始意识到人难免会犯错误，一些构建中断（build break）是无可避免的。这些团队选择采取措施防止构建中断出现和减少出现后造成的负面影响，而不是把注意力放在惩罚上。

构建中断通常由语法错误以外的其他原因引起。忘记签入某个文件是我看过引起构建中断的最常见的原因之一。当改动大型的复杂系统时，因为有依赖关系的系统的另一部分发生变化造成构建中断也很常见。设想一下：核心 Windows SDK 的头文件（包含 Windows 数据类型和函数的定义文件）。对这些文件中某个文件的一个小的改动可能会引起很不相关的组件的构建中断。类似的一个稍小规模的例子，COM 库中的接口名称或其他共享的组件的变更，很容易会导致依赖的组件中的编译错误。

测试编译

对大型平台如 Windows、Windows CE 和 Office，测试代码大量使用平台的公共函数，使用量甚至多于许多应用程序。当我在 Windows CE 团队时，我们每天的构建过程会先构建操作系统，然后再生成所有我们的测试程序。

因为产品中函数原型或定义的更改，我们的测试代码在编译时偶尔会被动导致中断（编译错误）。构建中断当然不是什么好事，但是其有好的一面，就是可以很早地捕获这些错误，而不是等以后客户使用时碰到。测试代码和该产品代码同时构建使我们能够找到这些缺陷。

停止构建过程的中断（Stopping the Breaks）

我还没见过有人在其团队中工作中没有过经历构建中断。在微软，我们使用几种技术来将构建中断的次数和影响减至最低。两个最受欢迎和最有效方法滚动构建（rolling builds）和签入系统（check-in systems.）。

在其最简单的形式，滚动构建（rolling builds）是基于该产品最新的源代码自动连续的构建。一天可能会有几个构建，因此构建错误会更早被发现。滚动构建系统中的基本步骤包括这些：

- 一个良好，没有问题的构建环境
- 自动同步（synchronization）到最新的源代码
- 构建 整个系统

- 自动报告构建结果：出错误或成功

实现一个滚动构建系统最简单的方法是用简单的 Windows 脚本语言写一个控制文件。我们也经常使用其它脚本语言，如 Sed, Awk, and Perl。表单 12-1 是一个滚动构建系统脚本的例子：

表单 12-1 简单的滚动构建系统的 Windows 脚本语言

```
rem RollingBuild.cmd
rem sync, build, and report errors

rem The following two commands record the latest change number
rem and obtain the latest source changes

:BEGINBUILD
rem clean up the build environment
call cleanbuild.cmd

changes -latest
sync -all

rem build.cmd is the wrapper script used for building the entire product
call build.cmd

rem if ANY part of the build fails, a build.err file will exist
rem notify the team of the rolling build status
IF EXIST "build.err" (
    call reporterror.cmd
) ELSE (
    call reportsuccess.cmd
)

goto BEGINBUILD
```

微软的一些团队的滚动构建系统还包括一些或所有 BVT 和自动报告测试结果。通常每天进行几个滚动构建的团队会选择其中一个 —— 例如，在下午一点之前最新的成功构建 —— 进行进一步的测试和配置，然后释放到小组，作为当天的正式构建。

另一种预防构建中断和提高代码的质量的方法就是使用一个签入系统。几年前，当开发人员作了代码改动(changes)，准备签入到 SCM 系统中时，他们会直接签入到源代码的主目录树中(main source tree)。任何错误 —无论是语法错误还是缺陷— 会立即影响所有构建。这样做，对于小型项目勉强可以，但对于较大的软件项目，选择多阶段的签入系统，对项目很有

益处。

图 12-5 显示了一个签入系统的基本体系结构。此系统：当程序员准备签入改动(changes)时，代码不是直接进入主要的源代码管理系统，而是先签入到一个中间的系统。最低限度，中间系统至少要在一个平台验证代码构建无误，然后才代表程序员把代码签入到主要的源控制系统。大多数这类系统包括多个配置和目标——单个开发人员只靠自己几乎不可能完成。

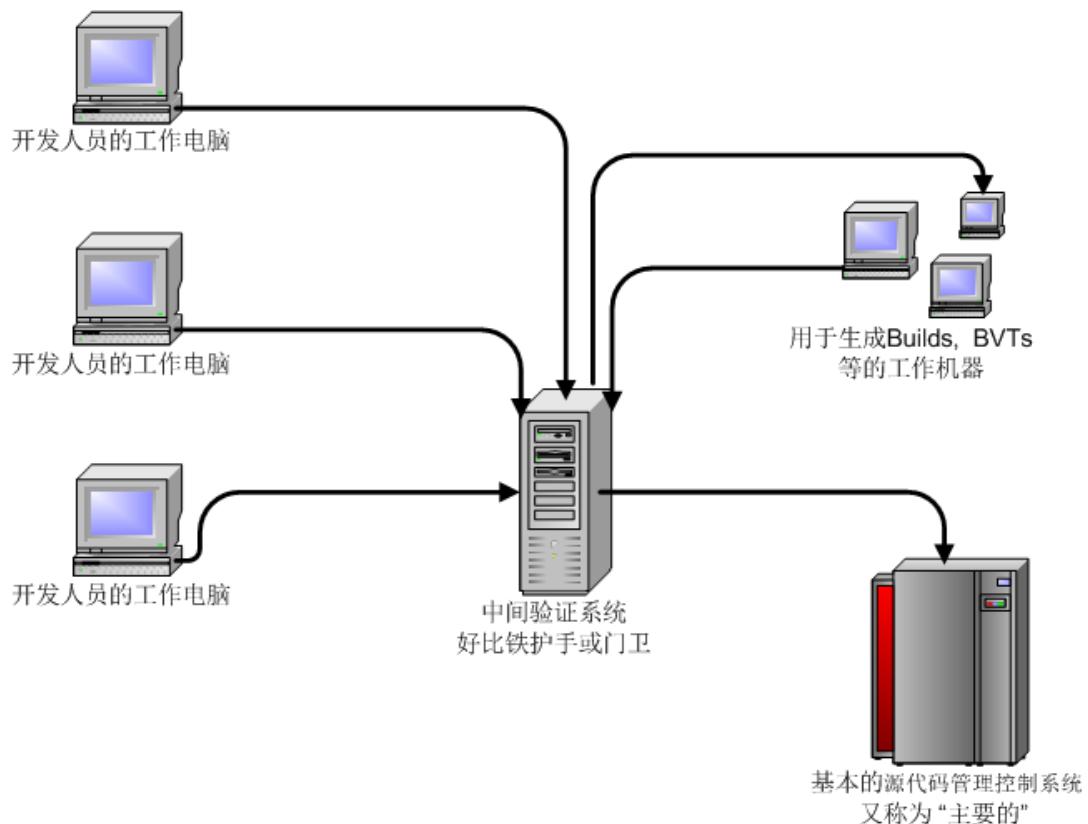


图 12-5 显示了一个签入系统的基本体系结构。

中间系统（有时也称为一个 铁护手(*gauntlet*)或门卫(*gatekeeper*))）也通常针对改动可能造成的回归（*regressions*）来选择运行相关的自动化测试。根据系统的具体实施办法，运行的测试可以是被预先指定好的，也可以是在签入时由程序员自己挑选的，或是基于这部分上次的代码被改动所进行测试动态创建的。

任何构建之前或之后的选择的测试，都可以在任何时间添加到系统中。对于开发人员，应该感觉不到这和直接签入到主要系统（主要的源代码树）有任何不同，真正的不同之处在于：大量的错误会在没有影响整个团队时被提前发现。

静态分析

当测试团队开始写测试软件

当一个测试团队开始编写软件去测试其它软件时，会有一个有趣的问题：“谁负责检验测试

软件？这个问题很容易被忽视但值得一问。大量的测试软件工作是编写测试程序，既然是写程序就很容易出现在正式产品程序上出现的同类型缺陷。在大多数情况下，运行测试和调查测试中的失败是一个检验测试软件的过程。但这种方法仍然可能错过测试软件中的许多缺陷。一个在测试代码（或者任何代码）中寻找缺陷的有效方法是利用工具做自动静态分析。静态分析工具可以分析源代码或二进制文件，发现许多类型的缺陷，而不必实际运行程序。

本机代码分析

一些不同的工具可用于分析非托管代码（例如，用 C 或 C++ 写的代码）。传统的工具包括商业产品，如 PC-Lint⁴²，KlocWork⁴³，和 Coverity⁴⁴，以及包括在 Visual Studio Team System 中的静态代码分析器

微软的每个团队都使用代码分析工具。自 2001 年以来，微软主要用于非托管代码分析的工具，是一个叫做 PREfast 的工具。这个工具也同样用于 Visual Studio Team System 中做非托管代码分析。

PREfast 扫描源代码，一次一个函数，并寻找编码模式和不正确的代码使用，并显示编程错误。当 PREfast 发现一个缺陷时，便会显示缺陷警告，并提供缺陷相关源代码的行数，参见图 12-6。

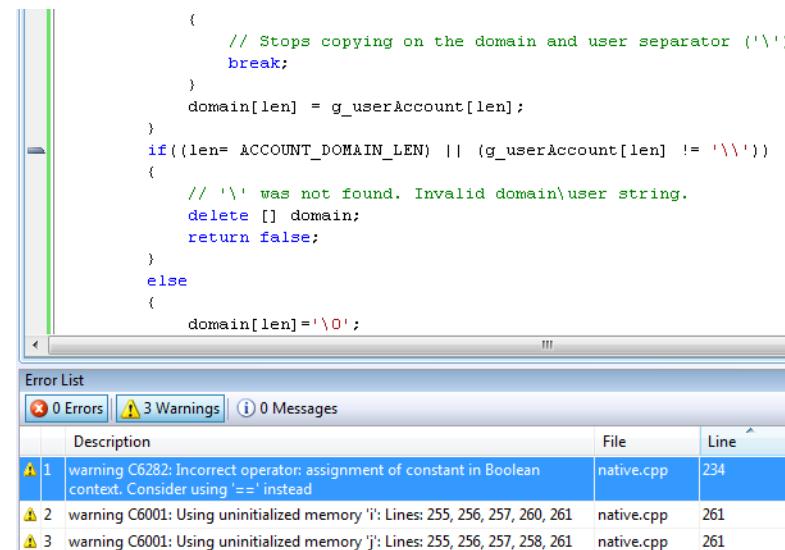


图 12-6 在 Visual Studio 上 PREfast 产生的静态分析警告

警惕破窗 (broken window)

⁴² <http://www.gimpel.com/>

⁴³ <http://www.klocwork.com/>

⁴⁴ <http://www.coverity.com>

在经典的《程序员修炼之道》一书中，作者讨论了破窗理论及它与软件熵（entropy）概念的关系：小缺陷没被改正，会产生更多的缺陷。当一个团队首次运行代码分析工具时，他们会发现大量的潜在问题需要调查，难免不知所措。团队面临的一个巨大的挑战是，即使是最好的工具也会产生一些误报。正如第 10 章自动测试提到的，误报是工具报告中的缺陷或者是测试结果反映的缺陷并不真是程序或代码的缺陷。当代码分析工具报告显示成百上千的缺陷时，这些不准确的报告会起到搅浑水的作用。

几年前，我负责部署我们的静态分析工具和负责报告测试结果。程序量是巨大的，所以我让各级开发团队的经理决定调查哪些缺陷报告。我们知道，最终希望调查和解决分析工具报告中的所有缺陷，但短期策略要由团队的管理层决定。我们改正了所有严重缺陷（忽略误报），并在当年晚些时候，发布了新版本的产品。

几个月后，一位软件开发工程师调查了一份用户问题报告。他需要一些特殊的条件去重新演示这个问题，最终他花了近一整天去调试。我记得那天傍晚他走进了我的办公室，他说：“我有好消息，也有坏消息。好消息是我发现了缺陷，并知道如何修正它。坏消息是我们的静态分析工具也发现了缺陷，三个月以前就报告过了。”

我们请来了其他几位开发工程师讨论这个问题，并查出，在这一特定的中我们的工具发现了几十个缺陷，它们并没被完全忽略。但开发工程师看了几个缺陷后，就把它们划为误报，并忽略了在这一部分的所有同类缺陷。

我们从这个例子中获得了一些经验教训。我们确保开发工程师们知道如何安全地忽略误报，此外要求至少两位开发工程师审查所有被忽略的误报，最后得出这些错误是误报的结论。为了确保程序质量，我们修正了所有的被报告的问题，并在我们软件开发系统中，增加了进一步的检查步骤。现在很难说出有多少潜在的用户问题被避免了，但我们都相信，纠正破窗问题是一个明智的投资。

托管代码分析

FxCop 是一个应用程序，它分析托管代码，报告各种信息，例如对软件设计，本地化，性能和安全性的可能改进，如图 12-7 所示。除了检测许多共同的编码缺陷外，**FxCop** 检测出许多违背编程和设计规则的现象（在微软类库开发人员的设计指导中描述了这些编程和设计规则）。任何人在托管代码上创建应用程序都会发现这个工具的价值。

FxCop 有两种方式，作为一个单独的工具，或作为 Visual Studio 的一部分。测试人员可以使用单独的工具来审查托管二进制代码的缺陷，但大多数的测试人员和开发人员会使用工具的综合版本进行分析。

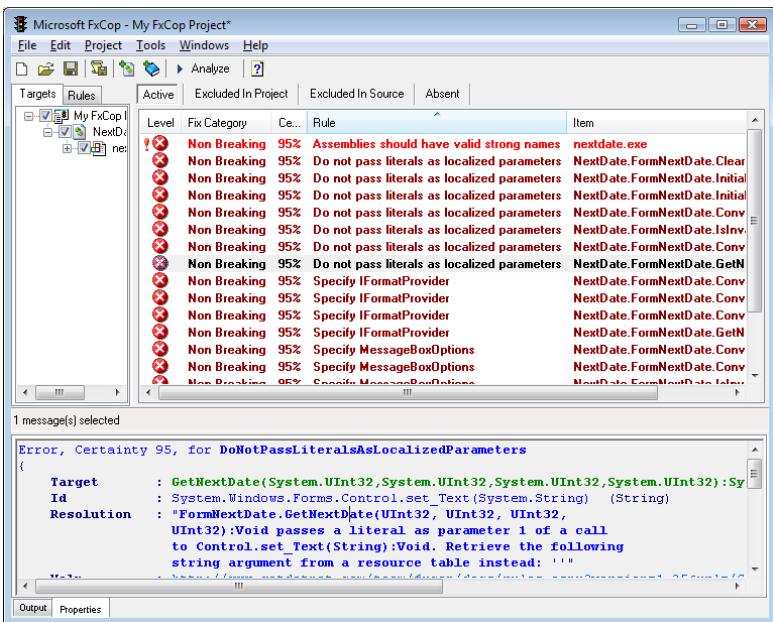


图 12-7 FxCop 分析工具

提示：代码分析团队博客上 <http://blogs.msdn.com/fxcop> 登载了丰富的 Visual Studio Team System 分析工具的信息。

代码分析的开销 (Code Analysis Overload)

使用代码分析工具的最大挑战是在使用入门期。对于一个全新项目，若代码分析工具在早期阶段使用，检测和修复代码缺陷的“额外”工作几乎注意不到。然而在成熟产品上开始使用分析工具时，潜在缺陷最初会带来额外的工作量，可能会把团队吓倒以至于停止使用工具。

几年前，我负责一个大程序库上的技术支持，维护和运行静态分析工具。初步报告显示，我们需要在下一个版本之前调查数以千计的缺陷。虽然在一个约有 100 名开发人员的团队里，人都知道会有错误被发现，但我知道如果要求每个人调查 10-20 个新缺陷，肯定有人会抱怨。

所以我通知大家，将开始在现有代码上运行静态分析工具，开始记录所发现的缺陷。我进一步地解释说，最先记录的是最关键的缺陷（可能是崩溃和安全问题），但随着时间的推移，我会增加额外的规则和检查。最初我记录了少量的缺陷，这之后每星期我添加了更为严格的规则，不断增加缺陷的记录量。有时有一些抱怨，但在短短几个月内，我们修正了数以千计的分析缺陷-其中许多缺陷不在我早期报告的缺陷类型之中。

如水煮青蛙的寓言所表达的，人们不会注意到相当于同样巨大变化量的逐渐改变。随着时间的推移，我们的团队解决了代码分析缺陷的积压任务，许多团队成员甚至经常找到我，要求增加更多的代码分析规则。

工具只是工具

FxCop 和 PREfast 都是功能极强的工具，每个软件团队都应当使用。静态分析工具能在测试团队看到代码之前，有效地发现某些类型的缺陷，但是它不可能取代测试。有些情况当然可能出现（可能性是存在的）：程序在代码分析时没发现任何问题，但仍然存在着缺陷。然而，早期纠正这些缺陷确实给了测试团队更多的时间和更好的机会，以保障在整个产品周期中找到大量更关键的缺陷。

测试代码分析

自动测试代码常有一些特定的缺陷。其中一个方面的缺陷是对环境的敏感性。这是指测试对环境不变的依赖性。它有多种形式的表现，用实例来解释可能更好。比如说：我们需要测试一个叫 **FileSaveWidget** 用户控件。这个用户控件它接受一个输入文件的路径然后储存数据在这个文件里。下面是测试这个用户控件的一个测试程序用例。

```
1 void TestFileSaveWidget()
2 {
3     // this file contains the expected output of the file save widget
4     string baselineFilePath = @"\test-server\ TestData\baseline.txt";
5
6     // the local path for the save file
7     string outputPath = @"D:\datafile.txt";
8
9     FileSaveWidget widget = new FileSaveWidget();
10    widget.SetDataFile(outputPath);
11    widget.Save();
12
13    Try
14    {
15        VerifyDataFile(baselineFilePath, outputPath);
16        WriteTestResult("PASS");
17    }
18    catch (Exception e)
19    {
20        String errorMessage = e.Message;
21        if (errorMessage.Contains("File not found"))
22        {
23            WriteTestResult("FAIL");
24        }
25    }
}
```

这段测试代码看起来挺不错，但里面至少有三个问题。在第四行它用了一个远程文件服务站

叫“test-server”。当这段测试程序在一个和测试人员起初使用的不同网络里或是在不同的使用帐户下执行的话，这个远程文件服务站不见得能被看见。在第七行它用了硬编码(hard coded)的 D drive。大概是这个测试员使用过的所有机器都有 D drive 吧。但这种假设大概会导致以后的缺陷，比如当 D drive 实际上是 CD-ROM 或者根本不存在的时候。

第十九行隐藏了一个较难发现但是很重要的问题，那就是在保存文件失败时，可能出现的缺陷提示信息是英文的“File not found”-它假设了这段程序运行在英文的操作系统上或是英文版本的软件。当我们用它测试西班牙语的版本时，报错信息应该是“Archivo no encontró”(译注：西班牙语，找不到文件)。这样这个测试就没有办法检测到了。许多微软的测试团队开始使用一些扫描工具来对测试程序进行扫描。下表 12-3 就是一个扫描工具的输出例子。

表 12-3 测试代码分析结果

问题	场所	所有人	细节
硬编码的共享路径	test.cs:4	Chris Preston	硬编码的共享路径在一个源程序文件里。它使程序不具备移动性(portable)。
硬编码的本地路径	test.cs:7	Michael Pfeiffer	硬编码的本地路径在一个源程序文件里。它使程序不具备移动性(portable)。请考虑使用配置文件。
硬编码的字符串	test.cs:19	Michael Pfeiffer	硬编码的异常字符串“File not found”。请用 ResouceLibrary 来使用本地化的测试。
无效的用户名	测试用例 ID 31337	N/A	测试用例指定给的用户名不存在。请指定给当前的团队成员

这种分析类型的工具不单单是在测试程序或是其他任何测试人员可能犯错的地方分析代码找出缺陷。它给测试代码带来了很大的改进，比如说：

- 通过找出硬编码的字符串，它改进了我们运行本地化测试过程的成功率。
- 通过查找潜在的源于使用 *Thread.Sleep* 而产生的竞争状态条件以及不成立的配置等，它改进了测试的可靠性。
- 通过寻找那些签入了但却没有被放在测试套件里的测试用例或测试程序，它提高了测试的覆盖度。

分析所带来的更大的价值在于它每天都提醒着我们预先防止这些缺陷，测试人员每天得到测试用例或测试程序的反馈，并以此提高它们的质量，进而提高测试质量，最终提高我们的产品质量。

测试代码也属于产品代码

这些缺陷也许对产品代码来说不那么重要，但对测试的可靠性和可维护性却是决定性的。测试代码的缺陷是业界的测试经理们最担心的事情。就算是在微软这样的环境中，我们有着大批写程序高手的 SDET，仍然需要靠这些工具来帮助我们开发可以可靠地运行十年并汇报准确测试结果的测试程序。

现在你应该很清楚了，微软的测试工程师使用很多传统的开发工具。这其实意味着在微软无论代码是否发布，我们是一样对待源代码的，不管是产品程序代码还是测试程序代码。

更多工具

测试工程师使用的工具的数量是没有止境的。微软的员工总是设法提高效率包括使用辅助软件，并且仔细地确认自动化工具的效用和手工测试的效用的一致性。让我们继续本章开头提到的工具箱的类比，用软件来辅助测试就像是用一个电动工具来代替手动工具。在很多时候，电动工具更快，能达到更好的结果，但是有些工作仍然需要人工来完成。

解决特定问题的工具

前面提到的大多数测试工具可以被大多数测试工程师所用。另外还有很多的工具是为少数人解决大问题的。屏幕记录器、文件解析、自动工具的附件、还有我们在第十章提到的字体显示器等全是用来解决特殊的测试问题的例子。

除了这些工具，测试工程师还习惯在他们的团队里共享其程序库，使得整个团队都可以用它作为一种被检验过的方法来解决类似的问题。比如在 Office 测试部门，他们有共享程序库帮助测试所有的 Office 应用软件中的 Windows 控件、在 Windows 移动部门，他们有共享程序库来模拟 cellular 数据。这些例子都证明了公用的解决问题的方法是如何帮助整个团队更好地工作。

服务大众的工具

微软的员工喜爱创建工具。几千种不同的测试工具从辅助测试程序库到 Outlook 的插件等等都被放在一个大家可以轻易拿到的工具库里。工程师们喜欢使用这些工具来帮助他们更好地工作，很多的人都愿意和本部门以及全公司的人共享他们的工具。

提示：微软的工具库里有员工们写的超过 5000 种不同的工具。

员工们可以在这个工具库里（ToolBox）如图 12-8 所示查找自己需要的工具。也可以预订 RSS 源，这样当新的工具被加进来的时候就可以得到通知。每一个工具都会列出关于这个工具的名字、作者和用途等信息。而且每一个工具都能在工具库的浏览器里被打分以及添加评论以帮助以后的使用者更好地做出选择使用哪个工具的决定。

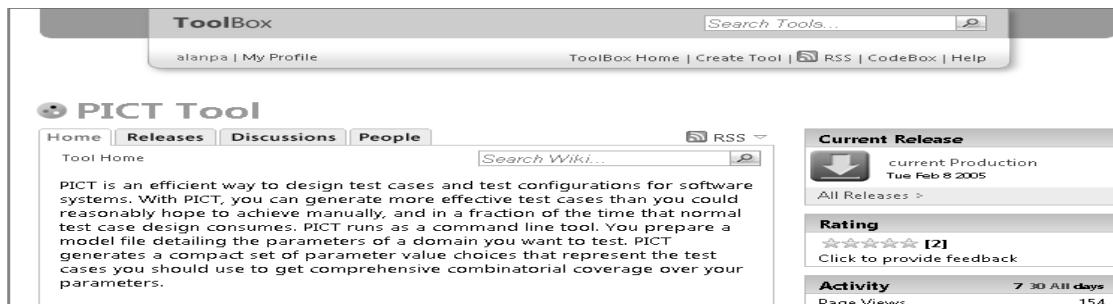


图 12-8 微软的工具库(ToolBox)

本章小结

我注意到一个优秀的测试工程师的其中一个资质就是他们能在测试中保持特别高的效率。他们并不急于采取行动。而总是能清楚地知道软件工具在哪些地方能比他们的大脑更快地解决问题。当软件可能解决当前的问题时，他们能够找到现有的工具或是在现有的工具上增加功能来满足自己的需要，如果必须的话，他们可以自己开发测试工具来解决问题。

测试工程师为了更有效地工作，需要在适当的时候使用新的工具，也应该到工具库里来重新查询。就如同源程序管理控制一样，测试工程师也应该检验他们的工具，看已有的工具是否可以有另外的用法。一个丰富的工具库，加上正确使用工具的知识和技能，是测试工程师拥有的最大财富之一。

第 13 章 用户反馈系统

阿伦 · 培智

大部份的质量难题是与用户相关的。软件公司如微软，是为人编写软件的。软件可以帮助人们提高生产效率或完成任务，除此之外别无它用。本章将讨论微软公司搜集用户和合作伙伴数据的工具和技术，这些工具和技术被用以改善我们的产品质量和改进我们的测试方法。

测试和质量

实际上，用户并不真正关心软件测试。在一定程度上，我猜想他们所关心的是在他们付款之前，产品得到了一些测试就行了，但他们对于测试工程师是如何实际测试软件的并无兴趣。

设想你在浏览商店中的软件时，拿起一个软件包装盒，阅读印在盒子侧面的功能说明的要点：

- 执行过 9000 个测试用例，98 % 的通过率
- 代码覆盖率超过 85%
- 每晚超负荷测试
- 发现了近 5000 个缺陷
- 已经有超过 3000 个缺陷得到了修复！
- 测试用到黑盒方法和白盒方法
- 还有更多……

所有这些数据对工程队伍是很有吸引力的，但用户对此并不关心。他们只关心产品是否解决问题，是否按他们的期望而工作。如果你认为软件质量是提供给用户的价值，那么大多数测试活动并不直接提高软件质量。尽管如此，测试确实有其价值（否则我不会写这本书的）。那么，测试究竟提供了什么？

测试提供了信息

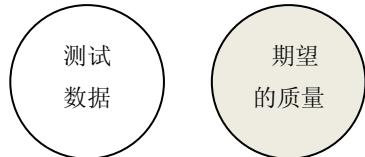
如前所述的要点提供了有关测试活动的信息，某些情况下，它们还反映了产品的状况。这些信息对于评估进展和确定风险至关重要。如果测试团队的最新报告说，他们已经运行了一半的测试，发现了 40 个关键的“严重级别为 1”的缺陷，这比起他们告诉你，他们已经运行了所有的测试，只发现了一个严重缺陷来讲，涉及的风险度量就大不一样了。当然，这还不够。你会想知道做了什么类型的试验，在什么情况下进行了测试，在哪些产品领域进行了测试，发现了多少非关键性缺陷，还有几十个其它的数据点。我从不喜欢把测试作为质量的“看门人”的想法。正相反，我更认为，测试过程提供了信息，使得产品决策者能够根据时间表和风

险对产品作出正确决定。

质量感知

如果一切顺利，测试团队所做的工作减少了风险，提高了质量。在现实中，这并不总是发生。往往是测试团队所提供的资料并不能够反映用户实际使用产品的情况。

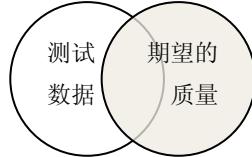
可以把测试数据和顾客感知的质量（我喜欢把这它叫做经验质量）作为两个相关的但不同的质量数据领域，如下图所示的两个圆：



如果有完整的测试信息，我们将能够预测到客户将会感受到的质量。例如，在产品推出的时候，我们会知道 6 个月后客户的满意度将是多少。此时，以上的两种质量圆的面积将接近重叠：



在大多数情况下，这些质量面积有重叠，但很少是我们所希望的有相当多的重叠：



微软正在努力寻找更多的测量方式来预测用户将以何种方式看待质量，以便使得已经获得的经验与测试数据相吻合。如果数据显示了用户正在使用产品的方式，哪些方面没有满足用户的要求，或是能够更好地了解用户更喜欢，或是更不喜欢怎样的产品，这样的数据对于开发高质量软件会很有帮助。

微软的大型软件项目在搜集客户数据上所面临的一大问题是处理各种不同数据集，以准确地反映出客户群的不同需求。如图 13-1 所示，产品支持数据、电子邮件、客户调研问卷和可用性研究结果都提供了宝贵的信息，但最大的困难是确定反馈意见的优先级，以及了解围绕这些数据点的用户的真实行为。而且，我们发现很多时候来自于不同渠道的数据并不一致，并且带有主观色彩，这使得我们经常无法处理和理解所有数据。

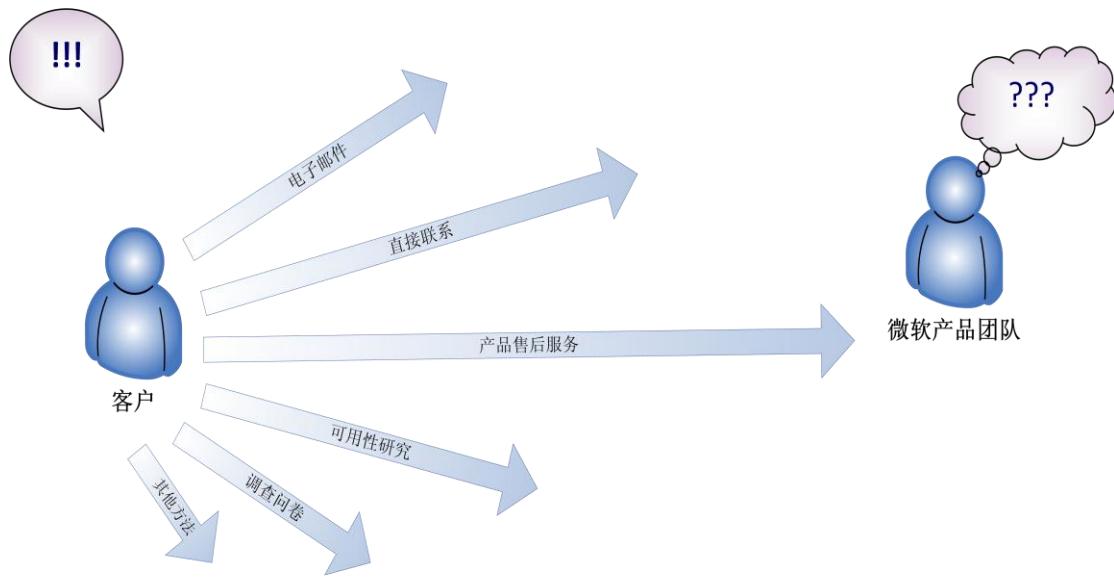


图 13-1 用户反馈

有许多方法可以用来寻找并收集客户反馈信息。本章将讨论大多数微软产品团队所使用的四种方法：客户体验改善计划（CEIP）、Windows 错误报告（WER）、发送微笑，以及 Microsoft Connect。许多这样的工具给我们的合作伙伴，以及微软的企业客户提供了方便。

用户救援

了解你的客户如何使用软件的最好方式，就是去观看他们的实际工作。只要你有几个用户并知道他们什么时候使用软件，而且他们同意你观看他们的工作，事情就好办了。了解你的客户如何使用软件，这已超出了可用性研究过程所能发现的内容。了解用户如何使用软件，这意味着你知道他们如何学习使用程序，哪些功能他们最经常使用，而哪些功能他们从来没有使用过。

在你安装一些微软开发的应用程序时，会有一个如图 13-2 那样的信息出现，询问你是否愿意帮助改进这些程序。如果感兴趣，你就有机会加入“用户体验改进计划（CEIP）”。如果参加了该计划，当你的计算机闲置时，有关你使用程序方式的匿名资料，会上载到微软公司去。在微软内部，我们知道为了做出高质量的产品，需要尽最大的努力去了解有关产品使用的情况。上千万的人们使用微软的产品，对于这些大的产品，我们即使想接触很小比例的用户都不可能。而这些数据提供了巨大信息量，以帮助我们了解用户如何使用我们的产品。除了针对软件外，我更愿意把 CEIP 想象成一种收集电视评分结果的耐尔森（Nielsen）评分盒。

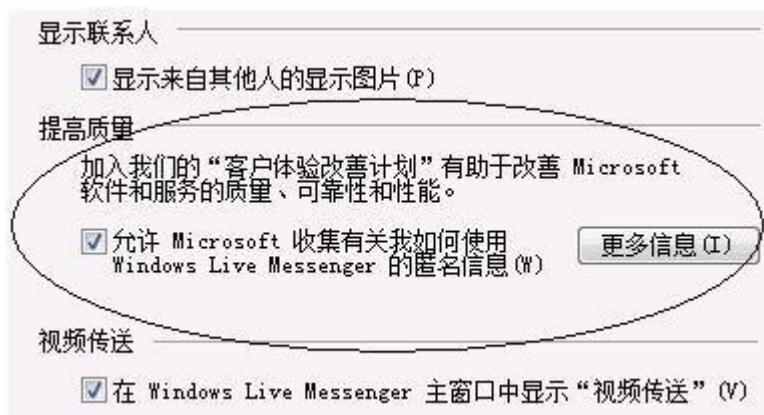


图 13-2 Windows Live Messenger 的 CEIP 选择的项目

一旦客户同意参加这项计划，我们就会搜集匿名的、非可追踪的数据点，它详细说明了软件被使用的方式，包括各种其他的信息，例如该软件是安装在什么样的硬件上，以帮助了解客户的软件产品使用情况（机密或个人身份的数据绝对不会被搜集）。最初使用这些数据的往往是设计师或用户体验工程师。从数以百万计的用户那里搜集到的数据，为了解用户使用产品的方式提供了独特的视角。下面是一些数据实例。

- 使用的应用程序
- 各个命令的使用分别有多频繁？
- 哪些键盘快捷键是最常用的？
- 应用程序的运行时间？
- 特定的功能的使用有多频繁？
- 质量度量
- 运行多长时间应用程序不崩溃（平均无故障时间）？
- 通常出现什么错误对话框？
- 多少客户在试图打开一个文档时收到了错误信息？
- 需要多长时间来完成一个作业？
- 用户无法登录到 Internet 服务的各种故障缘由各占了多少百分比？
- 配置
- 有多少人在使用高对比度色彩模式？
- 最常见的处理器的主频是多少？
- 一般用户的硬盘上还剩余多少可用空间？
- 大多数用户在何种操作系统版本上运行该应用程序？
-

以上这些数据点可以影响产品开发的许多方面。表 13-1 列出的典型 CEIP 数据，能够测量出用户花在程序运行上的时间。产品开发团队的最初设想是，用户在后台不间断地运行程序，偶尔把应用程序切换到前台。基于这一假设，有相当部分的测试场景和研发力量被集中在，随着时间推移的条件下，应用程序资源使用情况的测量上。与此同时，还要观察应用程序在长时间运行下可能出现的其他问题。该 CEIP 数据证明了这一事实，并表明四分之一的用户使用该产品的时间为 8 小时或更长时间，有将近一半的用户使用该产品至少 2 小时。

●

在24小时内应用程序运行的时间	运行次数	百分比	实际运行时间(秒)	平均时间(分)	实际运行时间的百分比
少于5分钟	2496181	24%	17811873		
6-30分钟	1596963	15%	8572986	18	31%
31-60分钟	691377	7%	4997432	45	16%
1小时	757504	7%	6957789	90	10%
2小时	487616	5%	5645785	150	8%
3小时	395007	4%	5514112	210	7%
4小时	347705	3%	5593810	270	6%
5小时	331561	3%	6135420	330	6%
6小时	355017	3%	7785188	390	6%
7小时	457104	4%	12357653	450	6%
8小时以上	2636023	25%	68692969	960	3%

表 13-1 用户体验改善计划 (CEIP) 的样本数据

产品团队意想不到地发现，将近四分之一的用户使用产品的时间少于 5 分钟，而另外 15% 的用户使用产品的时间不到 30 分钟。有相当比例的用户不是让应用程序在后台运行以便快速调用，而是启动应用程序检查变更的数据，检查完毕后马上关掉应用程序。在分析了这一数据后，产品团队重新调整了部分测试场景和性能工作以缩短应用软件启动和关闭时间，其目的是改善用户在这种情况下的体验。

•

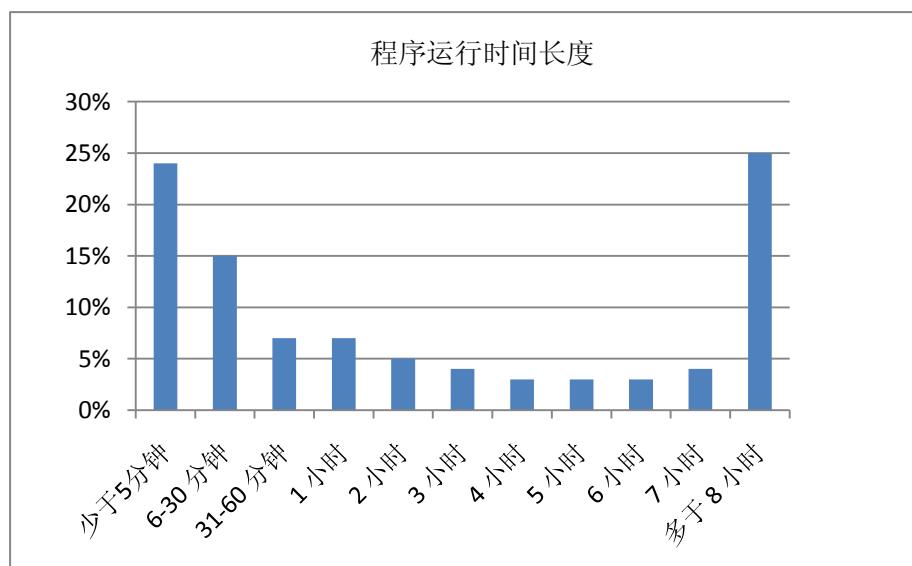


图 13-3 从 CEIP 得到的应用程序运行长度数据

CEIP 数据影响了许多设计决策，但同样对测试团队也很有帮助。从已出品的产品数据可以影响到未来版本的测试方法和战略，也可帮助分析产品出厂后所发现的重大问题的根本原因。在产品生产周期中，经常分析从测试用户和主要合作伙伴

那儿得到的数据，可使测试团队有可能更好地了解用户的使用模式和痛点，不断更新测试场景和测试的优先级。

提示：在 Microsoft Word 2003 版中最常用的前三个命令是，粘贴，保存和复制（来自 CEIP 数据）

在微软工作的许多年中，我曾分析过在产品发布后由顾客发现的许多缺陷，以便确定我们的团队如何改变对今后产品的测试。有些令人惊讶的发现是，只有少数的客户错误是在我们“遗漏”的领域中，例如那些领域我们没有测试覆盖或情景。大多数的错误都在我们已经确定的测试场景中或者有着很高的代码测试覆盖率。但只有当用户的场景与我们的场景不同时错误才明显。近年来，CEIP 数据已经开始帮助许多测试团队缩小了这一差距。

更多的资料：欲了解更多的 CEIP 的资料，请登录
<http://www.microsoft.com/products/ceip> 查看用户体验改进项目

用户为中心的测试

我们在接收到的CEIP数据和从测试用户来的Windows错误报告（WER）数据之间建立了桥梁。我们经常监测和使用客户数据，帮助了解客户看到的错误，并把这种错误与测试中发现的结果联系起来。我们高度重视寻找和解决每星期发现的最严重的10大问题。其中有些是我们的测试没有覆盖的新问题，有些是我们早已发现但无法稳定重现的老问题。分析数据使我们能够增加测试覆盖率和暴露出只有在用户的场景下才能出现的错误。

—克里斯莱斯特（Chris Lester），高级SDET

提示：早期试点的 MSN 订阅服务的注册过程限定为 20 分钟；CEIP 发现许多用户所花的时间超过 20 分钟，在他们完成注册之前已被终止了！

游戏也不例外！

受到 CEIP 的启发，微软游戏系统团队创建了一个应用软件监测工具，叫做“文思”（VINCE）是“消费者最初体会的检测”的简称，又被叫做“Voodoo Vince”，它是首次用于游戏监测的工具。今天，CEIP 能够运行在所有的 Windows 和 Windows CE 平台。“文思”可运行在 Xbox、Xbox 360 平台和 PC 平台，并支持游戏的具体需要。例如，测试版的用户可以通过游戏控制器迅速回答简短的客户调查表，告诉我们指定游戏部分的困难程度。

迄今最成功的监测例子之一是 Halo2。单人战役的 Halo2 有 14 个级别，每个级别包括数十个遭遇战（在游戏中总共有 211 个遭遇战，从 2 分钟到 30 分钟长度不等）。使用监测工具，该团队在每一个遭遇战中获得消费者的反馈信息至少 3 次。总共他们从超过 400 个参与者收集了 2300 小时的游戏反馈，如果没有这个监测工具，这个成绩是不可能取得的。

Halo2 的 Gravemind 级别的设计改进就是一个利用这些数据的实例。Gravemind

级别的出现是在大约 *2/3* 的游戏完成的时候。遭遇战发生在一个充满了敌人的房间里，其中包括邪恶的暴君。这是第一次游戏者与暴君作战，这个房间特别危险。测试结束后，通过查看监测报告，*Halo* 团队能够很快地发现这个遭遇战是异常困难的。对遭遇战的进一步分析表明，由于等离子手榴弹 (*plasma grenades*)，针刺枪 (*needlers*)，和暴君的攻击造成了大量的人员死亡。“文思”的一个有价值的功能是，它可以捕捉那些需要进一步调查领域的视频。看了相关的录像后，发现等离子手榴弹投掷的弧度过于平坦（不给游戏者足够的时间躲避手榴弹），暴君从后面杀害游戏者（游戏者立即死亡了还不知道死的原因），而且游戏者承担了针刺枪武器的沉重打击损失惨重（有些敌人使用两个针刺枪武器，变得极端的凶恶）。

Halo 团队在此反馈基础上做出了一些改变：一个总体的改变是如何投掷手榴弹（删除遭遇战中平坦手榴弹弧线的情况），敌人在房间里的位置（所有敌人从一个地方出现而不是几个地方，以减少游戏者从后面被杀的机会），并全面减少游戏者与敌人战斗的次数。还有其他一些小的变化。通过反复试验，他们发现，设计的变化大大减少了在这个房间里的死亡人数，同时改进了遭遇战的态度反馈（又称娱乐因素）。换句话说，*Halo* 团队能够减少困难程度和死亡人数又不至于使遭遇战过于容易。

用户使用情况的数据可作为一个重要的办法，去影响任何应用程序的产品设计和测试设计。这些数据可以用来定义没考虑到的用户场景，或修改现有的测试场景和配置。这当然不是唯一的办法去影响设计，如果提供反馈的合理监测功能在产品代码中不存在，这种技术不会有帮助。微软团队正是使用了这个重要的技术，设计、实现和测试了软件程序。这些软件程序明确解决客户的痛苦和不满意，支持关键的用户场景。

Windows 错误报告 (WER)

在微软，监测数据对定义软件开发工作的优先级所起的作用是令人惊讶的。我相信你们都看到过，在 Windows 中只要应用程序或系统发生了故障，你就能够向微软公司发送一份报告。

我们获得了很多这样的报告。我们已建立起非常有效的数据管理系统，用于研究这些报告，了解哪些驱动程序有问题。

我们允许任何拥有 Windows 应用程序的人注册，并得到与他们的应用程序有关的错误报告，人们可以到 winqual.microsoft.com 的网址得到这样的服务。

今天，我们已经看到了很多驱动程序制造商所作出的改进，但我们希望在应用程序的层次看到更多的工作，这会促进我们共同努力去改进用户还不满意的任何方面。

——比尔·盖茨，2003 年专业开发者大会

几乎每一个微软用户都看过图 13-4 所示的错误报告对话框。



图 13-4 Windows 错误报告对话箱

每次应用程序崩溃时，大多数 Windows 用户将看到一个类似的对话框。虽然我个人希望少看到这个对话框的出现，但是这个对话框有一个优点，每当它出现时，它会在背后产生数据点，微软使用这些数据点去改进产品质量。以上的对话框是 Windows 错误报告（WER）的基础结构的一部分。注意，在 Windows Vista 中，用户可以设定自动发送错误，而不需要进行交互式对话，。

WER 的工作原理

WER 是一种灵活的基于事件的反馈意见的基础结构，旨在搜集微软 Windows 可以发现的、有关应用程序或操作系统的硬件和软件问题，向微软回报信息，并为用户提供任何现成的解决方案。WER 是一种简单而安全的方法让用户提供应用程序崩溃的信息。WER 提供了一个数据库，它能为微软和软件伙伴供应商自动分类错误和设定优先级。它为微软和供应商们提供工具，以回应状况报告，要求用户提供补充资料或有关具体问题的信息链接。

在 Windows Vista 上，如图 13-5 所示，控制面板上的问题报告和解决方案记录了所有通过 WER 报告的系统和应用软件问题，它还向用户提供了更多有关错误的信息，包括可能的解决办法。



图 13-5 Windows Vista 的问题报告和解决方案

WER 的工作原理很像及时 (Just In Time) 程序调试。如果某个应用程序导致错误，它本身的错误处理程序未能处理，Windows 的错误系统会捕获这个错误。除了显示对话框，该系统在故障点收集数据，包括进程名称和版本，加载的模块，调用堆栈信息等。整个过程是这样进行的：

- 1 错误事件发生
- 2 Windows 启动 WER 服务
- 3 WER 收集崩溃的基本信息。如果需要附加的信息，WER 会征求用户同意获取资料。
- 4 这些数据上载到微软公司（如果用户没有连接到互联网，上载推迟）
- 5 如果应用程序注册为自动重新启动（使用 Windows Vista 上的 RegisterApplicationRestart 功能），WER 会重新启动应用程序
- 6 如果一个解决办法或补充资料存在，用户将被告知

虽然 WER 自动提供崩溃，死机，和内核故障的错误报告，应用程序也可以使用 WER API 来获取用户问题的信息。除了如前所述的应用程序重启功能，Windows Vista 扩展了 WER，以支持多种类型的非关键事件，如性能问题。

企业错误报告

企业错误报告 (CER) 是 WER 的一个版本，当情况发生时允许公司捕捉应用程序的崩溃信息并把报告传到微软。

CER 为企业提供若干好处。由于担心公司的机密资料，很多公司不希望他们的用户自动地转送崩溃数据到微软，（微软公司对收到的崩溃信息已有严格的保密限制）。

企业通过实现 CER 所获得的主要收益是，能够跟踪环境中的崩溃错误。通过记录企业用户所遇到的崩溃事件的确凿数据，企业将能够优先考虑那些对企业环境产生最大正面影响的软件更新。此信息也是一个有价值的数据，为 IT 部门衡量改善用户工作效率提供了依据。

填入存储桶（Buckets）

你可能想知道谁会做这种繁琐的工作，负责调查千百万 WER 事故报告。幸运的是，我们可以充分发挥计算机的作用，后端错误报告系统做了出色的工作，处理了大量的错误报告。

停止工作的软件过程名称，版本，调用堆栈，处理器寄存器，以及其他收集到的数据等会被自动地分析，并被分类放入不同的存储桶中。一个存储桶放有特定错误类别的所有实例。这个错误与特定版本的驱动程序，应用程序，Windows 功能，或其他部分有关。产品团队使用存储桶，适当调整他们如何处理 WER 问题的优先级。换句话说，产品团队根据每一个桶的点击数量来排序存储桶。

为确保优先处理用户最大的不满，所有超过指定阈值的问题会被自动地输入到产品错误跟踪系统中，这使团队能够专注于导致大多数客户抱怨的问题。

清空存储桶

随着越来越多的数据输入，数据的趋势也会变化。分析数据表明，在所有报告的问题中，有时修复报告频率最高的 20% 错误可以解决 80% 的用户问题，有时处理 1% 的错误会解决 50% 的用户问题，如图 13-6 所示。简单地说，用户看到的错误大部分是由少数实际错误引起的。这种现象通常与什么样的应用程序无关。产品团队的目标是首先调查那些导致最多崩溃的缺陷。集中于这些缺陷会用相对小的努力去产生最大的回报。

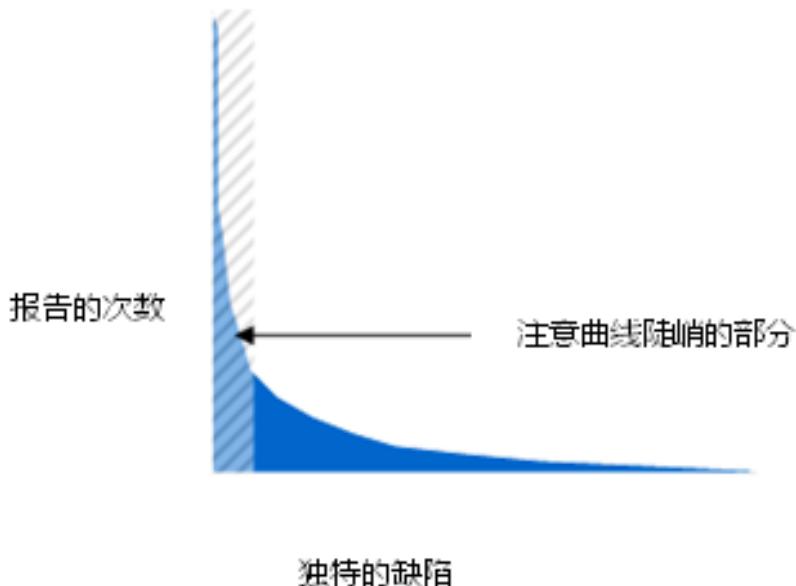


图 13-6 WER 事件分类报告

像其他反馈数据一样，WER 信息总是有帮助的，但是在测试版出厂期间所收集和分析的数据会非常有用。表 13-2 列出应用程序 10 大问题的典型崩溃数据。针对开发过程中搜集的 WER 数据，许多产品团队制定了自己的目标。共同的目标包括：

- **覆盖方法：**当使用这种方法时，团队的目标是调查 N%（通常为 50 %）的应用程序的总访问数。
- **阈值的方法：**如果他们的软件崩溃曲线是非常陡峭或非常平坦，团队可以使用阈值法，如图 13-6 所示。对于平坦或陡峭崩溃曲线，使用覆盖方法可能是不恰当的，因为它可能需要过多或过少存储桶进行调查。这种方法的合理阈值百分比介于 1% 和 0.5% 之间。
- **固定数方法：**这个方法把纠正错误数量固定为 N，而不是把目标定为百分比。

存储箱识别号	应用软件名	版号	模块名称	模块版号	代码符号	被调用次数	类别
231387	app.exe	1.0.0.1	appsup.dll	6.1.0.0	appsup.dll!Erase	1,511,546	Crash
309986	app.exe	1.0.0.1	app.exe	1.0.0.1	app.exe!Release	405,982	Crash
195749	app.exe	1.0.0.1	appsup.dll	6.1.0.0	appsup.dll!Draw	394,517	Hang
214031	app.exe	1.0.0.1	appsup2.dll	6.1.0.2	appsup2.dll!Reset	137,638	Crash
485404	app.exe	1.0.0.1	app.exe	1.0.0.1	app.exe!SetObject	100,630	Crash
390064	app.exe	1.0.0.1	appsup2.dll	6.1.0.2	appsup2.dll!Display	95,604	Hang
208980	app.exe	1.0.0.1	appsup3.dll	1.0.0.1	appsup3.dll!AppPrint	74,997	Crash
204973	app.exe	1.0.0.1	app.exe	1.0.0.1	app.exe!Release	55,434	Crash
407857	app.exe	1.0.0.1	app.exe	1.0.0.1	app.exe!MainLoop	54,886	Crash

229981	app.exe	1.0.0.1	appsupp.dll	6.1.0.0	appsupp.dll!function	51,982	Crash
--------	---------	---------	-------------	---------	----------------------	--------	-------

表13-2 WER 典型数据

测试和 WER

最起码，测试团队应负责监视汇总上来的崩溃数据和针对错误纠正目标的测量进展状况。从用户所发现的造成应用软件崩溃的方法中，我们会发现大量可利用的附加信息。

纠正大多数用户反映的问题至关重要，但是对用户常常看到的错误是否做大的改动，要取决于错误的根源。通常情况下，了解一个错误是如何产生的，了解为什么代码分析工具、测试、或者其它方法，没能发现这个错误。这种深入的调查可能会产生解决问题的方案，它可能防止类似的问题不再出现。只是纠正错误固然能够帮助用户，但是如果我们都能够做任何事去防止这类错误的发生那就更好了。

分析错误的根源也会找出软件崩溃的模式。在软件世界，设计模式是解决反复出现错误的答案。你可把崩溃模式想象为一种共同的、可借鉴的方法，它可造成完好程序崩溃。通过分析软件崩溃的共同原因，人们往往你会发现，产品的其他方面会以同样的方式崩溃。模式也可以被（已经被！）应用到许多其他类型的错误上去。

提示：需要了解更多的 WER 信息，请到微软 M S D N 网站
<http://msdn.microsoft.com/en-us/library/bb513641.aspx>，查找 Windows 错误报告论题

微笑，微软和你一同微笑

微笑着开始每一天并度过每一天。

- W. C. 菲尔德 (W.C. Fields)

我常常在想，当有些人使用我参与开发的软件时，他们的想法是怎么样的。哪部分功能他们喜欢？哪些部分功能他们不喜欢？如何才能了解用户对产品的反馈意见？CEIP 和 WER 给了我必要的数据，找出用户正在使用哪些功能，应用程序是否崩溃或锁死。但我知道，不论使用何种软件程序，每个造成崩溃的错误都可能是由上百种不同的问题引起的。我们希望用户感到满意，享受使用我们的产品。尽管在设计和开发过程中尽了最大努力，但是我们清楚，用户在使用产品时常常会感到不满意。具体来讲产品的哪一部分让用户不满意，哪一部分让用户喜欢呢？

情绪上的反应是难以衡量的：可用性实验室研究测量了基于一组任务的用户使用性能，但由于其结构的性质，这些研究不能稳定地重新产生前后关系，以引起用户的情绪反应。调查问卷测定了使用情况和满意度，但很少捕获到用户对产品的

感觉。实地考察和重点小组讨论会帮助用户回忆起他们的经历，但众所周知用户很难准确地回忆起足够的细节。

情绪上的反应是难于捕捉的，这是由于很难重新产生反应的机理。因此在这个领域里几乎没有历史资料。一种办法是，在软件推向市场时附带一组电极设备，观察心跳，及各种化学和电子信号，以便测量出使用软件时的情绪反应。不幸的是，我所期望的发现人类情绪的方法是难于测量的。（更不用说，大多数的用户可能不会正确地安装好测量设备）

另一种收集用户意见的替代方法是“发送微笑”计划。“发送微笑”是一个简单的工具，测试用户可以通过这个工具提交对微软产品的反映。在安装客户端应用程序后，小笑脸图标和小苦脸图标显示在系统托盘中（见图 13-7）。当用户有负面的经验，希望让微软知道，他们点击小苦脸图标。该工具捕获到整个屏幕的情况，用户输入文字描述他们不喜欢的部分。屏幕截图和评语会被送到微软的数据库，产品开发团队成员会对对其进行分析。



图 13-7 发送笑脸计划的小笑脸图标和小苦脸图标

参加“发送微笑”研究的外部用户开始喜欢上这个工具。他们发现当他们不满意时点击小苦脸图标的过程非常有趣。他们所要做的只不过点击图标，键入评语，点击发送，反馈意见会传送到微软。参与这一计划的用户与信息反馈计划管理员有直接的联系，他们发现他们的反馈意见影响到了新产品的设计、或影响到了对现有产品设计的改进。

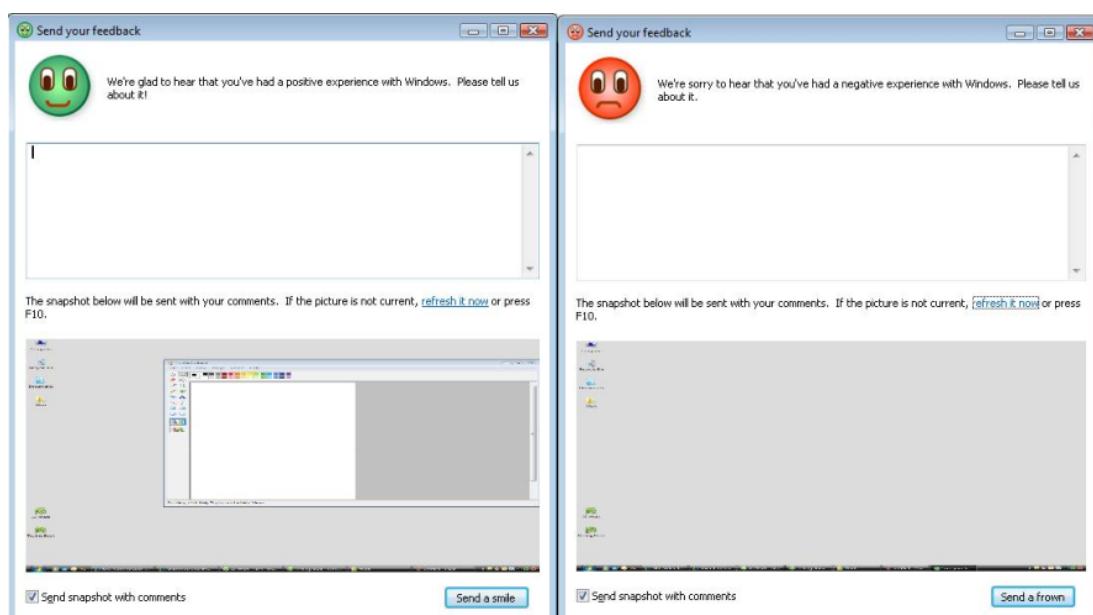


图 13-8 发送微笑用户界面

“发送微笑”计划相对较新，现在并不是所有的微软团队用它。该系统在收集客户意见方面有大量的潜力。它鼓励客户在任何时候都能提供快速的、自发的反馈

意见。设身处地想一想，有时你对应用程序很恼火，除了抱怨不能做任何事情。或者有时你使用一种新的软件，它对你的工作很有用，你感到有点意外。“发送微笑”程序提供了一个快捷的反馈机会，

发送微笑的影响

虽然“发送微笑”是一个相对新的计划，但它的影响是重大的。一些最大的好处包括：

- **对独特错误的贡献：**Windows 和 Office 团队依靠“发送微笑”系统捕捉独特的错误，这些错误真正影响消费者，却没有被其它测试方法或反馈方法所发现。例如，在 Windows Vista 测试版第一个里程碑期间，早期试用的用户通过“发送微笑”系统直接报告了分布在产品的 13 个不同的领域的 176 个独特的错误。在 Office12 的测试版本期间，Office 早期试用计划的用户报告了 169 个独特的错误。
- **提高顾客意识：**“发送微笑”帮助团队理解了用户使用我们产品的实际体会。
- **用户的实际问题影响错误的优先级：**“发送微笑”收集来的用户意见帮助团队确定纠正错误的工作重点。
- **理解用户是如何使用我们的产品：**团队经常猜想用户发现和使用了哪些功能，他们是否注意到了产品中增加的小功能。“发送微笑”使得团队成员很容易地看到用户的评语。
- **加强其他客户反馈：**“发送微笑”截图和评语被用来解释从其他方法收集到的特定用户问题，例如 CEIP 监测，新闻组，可用性实验室研究，以及调查抽样。

连接用户

在早年软件测试中，我从公司的 MSDN 账户下订购并使用了一个测试版的产品叫做 Microsoft Test（后来这个产品变成了 Microsoft Visual Test）。预发布的应用程序工作得相当好，但我也遇到了一些障碍，有一些问题不能解答。作为我公司的唯一测试人员，我的反馈选择是有限的，但我们的总裁有一个办法。用了他的帐号和密码，我登录 CompuServe 并发现了一个微软的论坛，我可以问问题。我从来没有给产品支持服务打过电话，没想到什么时候（或是否）能得到任何答案。第二天我登录时惊讶地看到，我不仅有一个回答，而且是个合理的答案。我被告知，我的建议会被产品团队考虑。在未来几个月内，我问了更多的问题，每次产品团队成员都以同样的态度回答。随着时间的推移，我读过其他测试人员的问题和答案，学到了很多。我甚至记得我也回答过几个问题。

今天，新闻组和论坛继续作为一种客户和微软产品开发团队工程师之间的沟通方法。`microsoft.public.*`的结构包含数百个新闻组，微软的工程师们和他们的用户积极参与讨论。在 forums.microsoft.com 网上论坛上，数以百计的其他讨论也在积极进行中。另一千微软员工在 <http://blogs.msdn.com> 博客上写下有关自己产品和工作的评论。微软认真对待通过任何这些手段得到的客户反馈，用来决定修复错误和增加或改变产品特点或功能。

Microsoft Connect (<http://connect.microsoft.com>) 是为微软员工和客户进行沟通的另一种方法，但它在交流中增加了独特而宝贵的信息。Microsoft Connect 的目标是建立一个围绕产品反馈的社区，向客户提供了与开发微软产品的工程团队直接交流的渠道。客户可以报告错误，提出问题，甚至建议新的功能。使用 Microsoft Connect 的“有趣”方面是，其他社区成员可以表决错误报告和建议。当用户提出了建议或输入了一个错误报告，其他人认为建议的功能将是有价值的，或者他们对特定的问题也经历一些痛苦，可以在反馈报告中简单地写上两个英文词“me too（我有同感）”。随着时间的推移，最流行的建议（和最不受欢迎的错误）上升到顶端，使产品开发团队快速准确地决定下一步需要做哪些工作。所有的错误和建议可被搜索，如果合适，社区成员（或微软员工）可以提供绕行办法或者替代建议。

图 13-9 微软 Visual Studio 和 .Net Framework 的 Microsoft Connect 的用户界面。

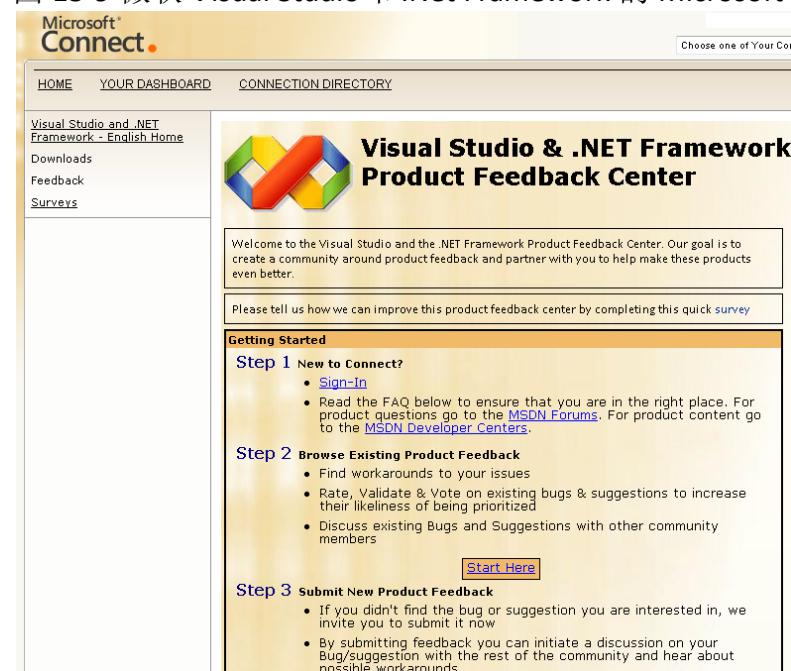


图 13-9 Microsoft Connect

Microsoft Connect 提供了用户一些其他有价值的功能。软件（包括给早期试用者的预发行软件），白皮书，以及其他资料可供下载。调查问卷，也是一种不错的选择，保证客户的声音被听到，并帮助微软收集改善软件的意见。客户连接是建立质量软件的最重要的建筑基石之一。

提示：要了解 Microsoft Connect 的资料，请到 Microsoft Connect 网站 <http://connect.microsoft.com>。

OneNote 用户参与

在 Office 2007 的周期，OneNote 测试团队探索和参与了许多客户参与的活动。该团队与产品规划和可用性工程师紧密合作，访问了七种不同的角色的 24 个客户。这使测试团队与真正的客户进行互动，收集反馈意见，更好地理解现实世界的使用情况。这个过程通知用户我们的测试策略、场景、配置和现实世界的项目。它还在我们中间产生了“同情心”，感受用户使用我们产品时所体会到的痛苦经验。

每个测试人员访问了 2 或 3 个客户，并写下有关工作的笔记，调查 OneNote 结构，寻求反馈意见等。这些数据是用来确认用户角色，功能决定，最终确认我们的测试和真实世界的项目。访问这些客户的其它好处是提高了测试团队的工作满意度。许多人评论说，这次活动是产品研发中最有意义的亮点之一。

测试团队使用博客搜索引擎找出早期试用的用户们对产品的评论。该团队发现了一个博客名为“我恨 OneNote 和 SharePoint”。通过这些评论，我们找到了博客的主人，开始通过电子邮件试图追踪这一问题。我们追踪到一个复杂的 WebDAV 配置，客户在使用和获得 OneNote 同步时失败。测试团队已经看到了这个随机错误，但它无法在内部重新产生。通过与博客主人一起工作，我们找到了这个问题的原因，并找到了解决办法，这就是安装一个特定的 Windows Server QFE。此外，测试团队有了更多的 WSS /Windows Server 测试配置，把这一信息加到测试场景中。

OneNote 也充分利用微软的连接机制 Microsoft Connect。这一网站已被广泛使用在测试阶段，我们能够动员社区参与收集错误，更好地了解配置，以及收集大量的现实世界的文件。我们现在有了专门的数据库。从我们连接网站来的信息包括：

- 18,000 会员
- 报告了 475 产品错误，其中 146 注明是要修正了。
- 来自社区的最好的错误例子：在 OneNote 2007 的最后阶段，通过我们的 Connect 网站，我们收到了两个严重损坏的文件，以及一些没有测试用例的明显错误。
- 超过 2000 用户回答了问题调查

在接近 Office 2007 尾声时的另一重大努力是开展了大规模跨部门的模拟现实世界的项目。来自近 30 个团队的超过 50 位测试人员和项目经理设计了 8 个不同的项目。通过建设多种多样的“现实世界”项目，小到一个在 MySpace 网站上的小建筑公司的网站，充分展示了 Office 2007 的集成软件系统。这个项目是成功的，有许多产品召回等级的问题和安全漏洞被发现并在最后出品之前得到了改正。

通过推进这种新的用户连接，测试团队能够推出用户喜爱的、高质量、高用户满意度的产品来。

麦克·唐福森（Mike Tholfsen） – Office 测试经理

还有很多其他的努力，帮助微软的工程师了解客户如何使用我们的产品。另外一个重要的例子是场景投票。场景投票展示给用户场景的描述，要求他们评级或提供对场景的意见，并自愿地提供他们对满意度的意见和场景的相对重要性。工程团队可以使用这些数据预先获取对这些场景的直观感受，并确定在哪些领域用户可能会遇到困难和在那些领域运作良好。场景投票概念的详细讨论可见《缺陷预防的实用指南（微软出版社，2007）》。

本章小结

测试人员经常把自己比作戴着用户帽子的人们，尽管有这种说法，许多测试人员并没有机会与用户有深入和必要的接触活动。测试要更接近用户情况，测试人员具有独特观察力，知道用户是如何使用和感受我们的产品。重要的是，测试人员在用户有关的活动中发挥积极主动的作用。

通过使用本章介绍的这些反馈机制，微软的测试工程师们能够发现漏掉的测试场景，找到测试漏洞，并直接针对用户的使用模式设计出新的场景测试。开发高质量软件的一个关键因素是，如何平衡使用用户反馈方法还是使用通过功能测试和代码覆盖分析等深入的技术分析方法。

第 14 章 测试“软件加服务”

肯·约翰斯通

2007 年父亲节那天，作为礼物，我收到了一本精装版的《给男孩子看的危险的书》(The Dangerous Book for Boys)⁴⁵。我让我儿子拿着书。他用小手轻轻抚摸着红色的封面，开始追踪封面上那几个金色的大字。我看他的眼睛睁大了，嘴也咧开了。那本书似乎在召唤地说：“快来读我，你会发现好多秘密”。

书的第一章对我们是至关重要的，它教我们怎样准备超级冒险者背包和必备的探险工具。那些关于“打结”、“化石”和“恐龙”的几章也都深受欢迎。不过我不是很确定“地雷拉法线”这章，但我儿子却觉得它是最棒的。我镇静下来和他一起看完了这章。但我非常清楚地告诉他：不能在我的工具室里，或是家里任何一个地方设定地雷拉法线。

小男孩和危险似乎总是很自然地互相般配。不管在什么情况下，它们总能找到对方。《给男孩子看的危险的书》这书是很有用的工具，能够帮助孩子们学会区分小危险和大危险。软件加服务(S+S)和微软也是自然地般配。微软的应用软件和服务器软件有上亿条源代码。这些产品在全世界上百万的计算机和设备上运行。很自然地我们想把这些产品和所有设备的计算能力，集成为一个软件加服务的系统。

有许多关于面向服务的架构(services-oriented architecture – SOA)和软件作为服务(software as a service – SaaS)的书已经出版了。我希望你读过某一本这类的书。对任何一个跟服务打交道的人来说，懂得这些基础是非常重要的。

但是，在现有关于服务的参考资料中，有一项空白：还没有一本书谈到开发和发布软件加服务的风险。我们像小男孩一样需要有一本红面金字的书来告诫我们：测试“软件加服务”的危险，并提供生存技巧。

两个部分：关于服务和测试技术

虽然这一章是讲述如何测试 S+S，但不讨论一些背景知识就没有办法直接进入测试主题。为了您的方便，我们把本章分成两个不同的部分。

第一部分讲述微软在线服务的历史和我们的策略。然后把它和传统盒装软件以及 SaaS 作对比。这一节还介绍微软在数据中心和服务器数量方面投资了多少，以及微软所有的服务的运营概况。

第二部分则全是关于测试。我首先着重讲影响测试方法的要素，然后详细讲述用于测试软件加服务的环境的一些新技术。在最后，我会讲一些我所看到的在运营环境下服务还存在的缺陷，以及当服务已经在线后怎样改进质量。

⁴⁵ Conn Iggulden and Hal Iggulden, *The Dangerous Book for Boys* (New York: Harper Collins, 2006).

第一部分：关于服务

在本节，我们首先回顾微软在互联网和服务策略中的几个主要转折点，接下来，再简述服务的架构元素。本节将帮助那些初涉服务领域的读者理解以下两者的基本区别：传统盒装软件与微软在软件加服务方面正在努力推出的产品。

微软的服务战略

软件加服务 (*Software Plus Services*) 是微软创造的术语，它展示了我们对分布式软件拥有巨大价值的信念。这种分布式软件集成了在线服务的集中和协作之功能，同时又可以利用超过 8 亿台计算机和数十亿台智能设备的处理能力和脱机功能，这些设备是消费者、知识工作者和游戏玩家已经拥有的。

在很短时间内，服务变成了软件生态系统的一个重要组成部分，很多人认为在不久的将来，它将成为软件使用的主流。以此为基础，读者也许会纳闷，我们为什么把关于服务测试的一章放到书的末尾，并且放在“测试的未来”一节呢？理由是：微软正在迅速发展服务测试的方法，随着此种程度的持续进步和创新，这个话题似乎更适合未来的趋势，而不是象前面章节中已介绍的，经过实践验证的做法。

业务重心向 Internet 服务迁移

1995 年，比尔 盖茨发布了一个备忘录，号召微软迎接互联网，将其作为公司的最高重点。在微软内部，它经常被称为“互联网备忘录”，下面段落节选自其中。当时，网景 (NetScape)、美国在线 (AOL) 和其他几家公司互联网潮流中比我们领先。随着明确的前进号令，微软的工程师将他们的目光转向了与这些公司的全力竞争。

互联网浪潮

我曾经几次提高对互联网重要性的展望，现在我赋予它最高级别的重要性。在此备忘录中我想明确指出，以互联网为重心，对我们每个部分的业务都至关重要。自从 IBM PC 在 1981 年问世以来，互联网是最重要的一项进展。

—比尔 盖茨，1995 年 5 月 26 日

在互联网备忘录后的数月内，互联网开始出现在微软各种产品中，例如：微软 Office Word HTML 外接程序，它可以帮助用户生成和保存 HTML 到互联网开发工具 Visual Interdev 中。在一段时间内，微软内最流行的互联网工具是 Notepad，因为它可以让你快速阅读和编辑原始 HTML。

十年后，一个新的备忘录被推到了前台。2005 年 10 月，我们的新任首席软件总设计师，Ray Ozzie 向所有员工发出了“服务”备忘录，如下段所示。从十年前发布 MSN 起，微软就拥有了互联网服务业务。但是这个备忘录谈到了利用互联网上服务的强大功能，以及将各种服务紧密集成的世界。在这个备忘录中，我们的策略重新定位为软件加服务，它代表着我们如下的策略：桌面应用程序和 Windows 操作系统与在线服务完美结合，以及我们对 SaaS 和 Web

2.0 技术的使用。

互联网服务巨变

如今有三种准则在驱动着这个领域的根本转变--它们都在某种程度上与服务有关。在我们的产品和服务中包含这些准则是至关重要的。

1. 基于广告的经济模式的能力。
2. 新的推出和采纳模型的有效性。
3. 令人注目的、集成化的、拿来即用的用户体验。

—雷·奥齐 (*Ray Ozzie*), 2005 年 10 月 28 日

在 S+S 策略中，互联网处于所有实现的中心，但支持的客户列表却比纯在线的 SaaS 方法要长得多。S+S 可以利用用户手边已有的个人电脑和移动设备的计算能力。

到本书发行时以及其后的几个季度，微软将发布一系列云服务产品。互联网上的任何基础服务，如果其他服务可以建立在它之上，就能称作云服务。最近，我们公开发布了我们新的云架构服务平台，此新产品被命名为 Azure 服务平台。（更多信息请参见 www.microsoft.com/azure）。Azure 在博客中又被称为 RedDog，它是提供给开发人员基本功能的云服务，包括基于虚拟机及云存储的计算能力。Live Mesh 是 Azure 在线服务的一部分，它允许用户实现跨越多台机器及设备（如智能手机）的数据同步。

这些云服务正在改变着开发人员编写和运行服务的方式，并将会进一步改变我们测试服务的方式。一些公司诸如 Smug Mug 和 Twitter，在使用 Amazon 简单存储服务（Amazon Simple Storage Service）（S3）后，报告了大规模的成本节约。但它们也不得不面对由于 S3 中断引起的服务中断。⁴⁶

从大规模成长为超大规模

1994 年，微软发布了 Microsoft Network (MSN)。MSN 迅速发展成为美国第二大拨号上网服务。我们甚至有一个广告活动宣传 MSN 是无需培训即可使用的互联网，这当然是在调侃我们最大的竞争对手美国在线 (AOL)。

即使作为业内的第二大服务，MSN 仍然非常庞大，拥有成千上万台生产服务器。我们超出了我们第一个数据中心（名叫 Canyon Park）的容量，并很快填满了第二个数据中心—Tuk1。MSN 团队拥有很多一流的最佳实践，用来开发和发布拨号客户端、服务（如电子邮件）和互联网连接。在我们的专业领域以外，我们还有许多要学习的东西。1997 年，我们收购了 WebTV 和 Hotmail，也使我们认识到有太多东西我们需要去学习。在我看来，这些收购加快了我们从大规模拨号服务向超大规模互联网服务供应商转变的步伐。

⁴⁶ Jon Brodkin, "More Outages Hit Amazon's S3 Storage Service," *Network World*, July 21, 2008,

<http://www.networkworld.com/news/2008/072108-amazon-outages.html>.

我有幸作为许多工程师的一员，飞到硅谷去向 WebTV 和 Hotmail 学习。这些公司带来了许多聪明的工程师，向我们介绍了各种各样运行大规模服务的创新概念。从这些会议中我学习到两个关键概念。

第一个是 WebTV 的**服务组 (Service Groups)** 的概念。服务组是功能基本上相互独立的生产单元。一个服务组是生产规模的单元，同时也是下一次升级的部署单元。服务组也为网站级别的服务中断提供缓冲。如果某个服务组由于某种原因当机，理论上它不会影响到其他服务组。

第二个概念是 Hotmail 的**现场可更换单元 (FRU)**。他们夸耀他们拥有的生产计算机的数量之多，以及他们是如何便宜，事实上也的确如此。这些计算机实际上就是把主板、硬盘和电源粘在一个平板上。在那时候这些计算机运行时就散热少且耗电低，因此数据中心的中央空调系统足以保持机器冷却，不需要用机箱覆盖或引导气流。

在微软内部，服务组的概念演变为规模组（Scale Group）概念，它保留了服务分割的概念，同时也包括了作为一个单一的大订单来购买增加容量。**现场可更换单元**已经变成众所周知的“商品硬件”，它强调一次性购买大量的廉价服务器。

虽然这些概念还在使用，他们也受到我们从未见过的、极高增长率的挑战。仅仅为了满足我们服务的用户不断增长的需求，微软现在平均每月就向我们的数据中心增加 1 万台计算机。即使规模单元（Scale Unit）和“商品硬件”也不足以应付那种程度的购买和运营配置。

几年前，我们开始购买被称为机架单元或机架型号的服务器。这些都是装满了服务器的高大机架。这些机架被从送货卡车推到数据中心，在后面接上各种电缆线，再打开电源，安装就完成了。这项革新极大地提高了我们采购和安装渠道的效率。但难以令人置信的是：机架单元也已经不够快了。

言出即行：微软建立第一个基于集装箱的大型数据中心

集装箱里装满了事先配置好的、马上就可以运行的服务器。它们被兜售为即时扩展数据中心的更快、更模块化的方法。

微软和 Sun 微系统公司可能都会声称自己首先发明了‘盒子中的数据中心’的概念；但微软似乎是第一个大规模推出基于集装箱的系统的公司，安装在其数据中心。⁴⁷

最近，我们建立了一个完全基于集装箱概念的数据中心。一个集装箱型号是一个满载和生产就绪的货柜，它可以由卡车和铁路运输，如图 14-1 所示，由起重机从平板上吊起，再放到数据中心的地板上。从各种面板上接出几个大型的电缆束；当它们被接通后，马上就有几百台服务器随时可以加入生产中去。若干年后当这些服务器失败率开始增高时，集装箱就被从生产中拿掉，运回原厂家，被回收并且用新设备更新。

⁴⁷ Eric Lai, “Microsoft Builds First Major Container-Based Datacenter,” *Computerworld*, April 8, 2008, http://www.infoworld.com/article/08/04/08/Microsoft-builds-first-major-container-based-datacenter_1.html.



照片由 Rackable 系统公司提供

图 14-1 Rackable.com 现在生产一种被称为冰块（ICE Cube）的集装箱

单单是让所有硬件到位来支持我们的软件加服务策略，就迫使微软开发世界级的后勤规划。产品工程师要不断地与采购专家沟通，以保证正确的设备在正确的时间到位。管理订单的开销在减少并最终将完全消除。

表 14-1 中的数字基于最近的历史记录和对近期的预测。我们在不断的处理由收购带来的增长需求。在本文写作时，微软已决定不再购买雅虎（Yahoo），但是如果我们今后要进行一个像这样大规模的收购，我们可以预期这些数字会有显著的变化。

表 14-1 微软服务小档案（Factoids）

服务器数量	微软平均每个月向基础结构中增加 1 万台服务器。
数据中心	微软新建的支持软件加服务的数据中心平均耗资 5 亿美元，有 5 个足球场那么大。
Windows Live ID	WLID（以前叫微软 Passport）每天处理超过 10 亿个用户验证。
性能	微软服务基础结构每天会通过 System Center 收到超过 1 千亿行性能数据（收集了 8 万性能计数器和 1 百万个事件）。
服务数量	微软现有超过 200 个命名服务，并将迅速突破 300 个。即使这也不是服务的精确数量，因为有些服务像 Office Online 是由不同的服务如剪贴艺术（Clip Art）、模板和词典功能组成的。

能源是成长的瓶颈

兴建一个普通的数据中心大约耗资 5 亿美元。多年来我们发现摩尔定律对计算机依然有效：大约每 18 个月处理能力就会加倍。为运行和冷却单一服务器所做的努力也在不断攀升。

新型生产服务器所需的电能开始引起电能消耗不平衡。此时需要实施大的电能升级项目，直到基础结构达到其最大容量。一旦达到这个阶段，唯一的选择就是重新设计基础结构，规划一个时间段来拆除和重建大部分的电能基础结构。

电能对我们服务的运营非常关键，所以我们和英特尔（Intel）和其他原始设备制造商（OEM）紧密合作，来设计在电能消耗和性能之间更优化的服务。例如，一个 OEM 厂家，在优化一个更低成本的服务器时，可能会选择一个标准的电源供应器，它会使用比实际需要更多的电能。虽然这样使得生产服务器的成本降低，但在我们数据中心运行服务器的长期费用增加

了。我们努力保证在服务器中使用更多低能耗部件，并且仔细调节系统使我们不再浪费电能。

生产节能灯泡是省电的好方法，但学会在黑暗中看到则要好得多...

可以把数据中心的整个建造和日常运营花费想象成一种投资，即输送电能到数据中心中存放的设备。在这种模型下，水泥地板的角色是输送电能，中央空调系统（HVAC）的角色是输送电能，保安人员的角色也是输送电能。将设施内的所有费用都想像成用千瓦在数据中心内来记录和输送以，这就是微软如何思考大型数据中心的运营成本，我们花费时间用于创新，以此来降低我们建造的每一代的数据中心的成本。

让我们来想想，现在用数据中心的千瓦时容量，某一个特定技术所完成的工作。微软的主要成本效能矩阵以每千瓦时的工作量成本来计量。我们问自己，在我们将电能转换成在线场景，比如一个查询，一个电子邮件（e-mail）信息，一个页面浏览，或一个视频流时，我们是不是更节约成本？

用比喻的方式讲，我们要求开发员和测试员设计越来越高效的灯泡，同时要求我们的设施建造者让它们在输送电能时更加节约成本。

但是我们如何完全消除这种需求呢？

人们做事时总是基于不同的动机。虽然我们要求开发员和测试员在设计时考虑成本因素，我们认识到不同的人会被不同的事情所激励。最近我们实施了一个有趣的项目，即计算和分配微软各种在线服务所产生的碳量，并将其碳足迹分配到开发在线服务的不同技术部门。现在开发员和测试员能看到他们的技术每月消耗的碳量。这种碳分配模型（CarBAM）在微软内部网上提供给所有员工，其指标也被放在成本报告中。如果这个简单的认知能以独特的方式激发开发员和测试员在数据中心中部署更少的设备以节约能源，我们就实现了纯的需求消除。这种方式现在弥补了对更高效能代码的不停探索。

我们发现生产高效能的灯泡是减少电能消耗的有效途径。以创新的方式迫使我们的工程师跳出传统框框来看待这个问题则更好。

Eric Hautala, 总经理

服务与盒装软件

盒装软件实质上是任何可以通过 CD 或 DVD 介质购买的软件产品。在微软内部我们倾向于把这些产品称为热缩薄膜包装（shrink-wrap）软件。对微软来讲，收缩性薄膜包装软件也包括您计算机上预装的产品版本或您在计算机上运行的可下载版本。

提示：热缩薄膜包装，也称作热缩薄膜，通常作为很多类型包装的外包装：CD、DVD、软件等等。它由聚合物塑料制成，受热后会紧密收缩，包紧任何在里面的物品。

近几年，似乎对不是服务就是薄膜包装的所有软件产品来看，概念有些模糊了。有些 Xbox

的游戏是从零售店里买的，但用户可以在 Xbox Live 上互相对打和下载额外级别。Microsoft Office Outlook 是 Windows Live Mail (WLM) 的非常流行的邮件客户端。Microsoft Expression 是一个可以在零售店或互联网上买到的产品，这个工具可以帮助网站所有者快速地开发多媒体内容驱动的网站，但它也建立在用户连接到互联网的前提下，这样用户就可以下载额外的内容和插件。虽然所有这些产品都有服务组件，它们通常是通过零售商和 PC 厂商来购买，因此我们把它们归为薄膜包装类别。

提示：2005 年微软的几个主要在线服务，如 Hotmail 和 Passport 被改名，并已包含在 Windows Live 品牌中。Hotmail 现在叫 Windows Live Mail (WLM)，Passport 则变成 Windows Live ID (WLID)。拥有 Hotmail 帐户的用户仍然可以保留@hotmail.com 的地址，尽管我们改变了这项服务的称呼。

Web Service 有许多不同的名字。一些常用的术语包括：Web Service、Web Site、Web Property、和在线服务 (*online service*)。通常，在微软内部我们把自己的服务按产品组来称呼，比如：XBOX Live、Search、Windows Live ID (WLID)、Spaces、Sky Drive 或 Office Live Small Business (OLSB)。有些情况下，比如 OLSB 或 XBOX Live，称呼它们为服务集合更合适。OLSB 提供公开网站、私用网站、网站管理、连接手机的短信息、商务邮件和其他作为附件的服务，如联系人管理器和广告管理器。每一个子服务都可以被其他服务使用，比如 OLSB 被打包到 Dynamics Live 和 CRM Online 中销售。XBOX Live 拥有自己的视频点播下载服务以及锦标赛服务。

负责制订 Web 服务架构需求的 W3 工作组在 2004 年 2 月 11 日的说明中是这样定义 web service 的：“一个 Web Service 是被 URI [RFC 2396] 所标识的软件系统，它的公共接口和绑定使用 XML 来定义和描述。它的定义可以被其他软件系统发现。这些系统然后可以和 Web Service 按照其定义规定的方式进行互动，并使用经互联网协议传送的基于 XML 的消息。”

这是一个好的定义，但是有一些被很多人认为是 Web 服务的网站很少有、甚至完全没有 XML。有关定义的最新解释和变更，请访问 <http://www.w3.org/TR/wsdl-reqs>。

软件加服务实例

以 Windows Live Mail 服务为例，WLM 是迄今世界上最大的电子邮件服务，拥有上亿活动用户。WLM 的吸引人之处在于它的可靠性，以及它对多种客户体验的支持。WLM 支持多种网络浏览器、微软 Office Outlook 客户端、Windows Live Email 客户端、移动智能电话和桌面小程序。此种小程序可以在新邮件到达用户收件箱时提醒用户。

低端浏览器体验很象 SaaS 体验。SaaS 主要是指连接到互联网时使用一项服务，它并不具有脱机功能。在 SaaS 中大部分的处理发生在“云”中的服务器，客户端通常仅提供用户体验的呈现。

提示：低端 (downlevel) 浏览器是微软以及一些 Web 开发网站使用的一个术语。低端浏览器的一个严格定义是一个仅支持 HTML3.2 或早期版本的浏览器。这个术语经常泛指旧的浏览器版本，它们缺少一些关键功能比如级联样式表 (cascading style sheets) 或 JavaScript，并且不能在网站上展示最好的用户体验。

Web 2.0 倾向于依赖新版浏览器、Flash 或微软 SilverLight 来提供丰富的浏览器体验，大部分的运算从云中服务器转移到桌面电脑。Web 2.0 客户端也可能拥有脱机功能。

S+S 将运算转移到对用户和用户体验最有意义的地方。对丰富客户端如 Outlook，主要的运算发生在用户的计算机上。类似地，对于智能电话电子邮件（email）客户端，电话则处理主要的运算。S+S 也强调脱机工作的能力以及让客户端连接内部服务和云服务的能力，正如我们在 Outlook Connector 看见的那样。

Outlook Connector 可以提供一个集成的体验，让用户能同时阅读公司电子邮件和个人邮件账户。在不久的将来，服务器会运行在混合模式，将公司服务器、云服务器和外部服务器紧密结合起来。

在 WLM 例子中，低端浏览器是 SaaS 的例子，而 Outlook、Outlook Mobile 和 WLM 多媒体电子邮件客户端则是 S+S 的例子。这个例子的最后一个成员体现了服务不再是独立的。例如，在 WLM 中，用户验证（当用户输入正确的用户名和密码时）由 Windows Live ID 服务提供。这个概念在后面“从独立服务向分层服务的转变”一节中有讨论。

在 S+S 中，兼容性远远超出了只是浏览器版本的不同，它可以包括数十个不同的客户端，如图 14-2 所示。此外，高可用性、可扩展性、强大的安全性以及用于保护隐私的可信任的政策和程序，这些都比企业场景更加关键。从测试角度看，这意味着质量与以下因素紧密相连：功能、架构、最独特的服务以及运行和维护它们所需的规程。在微软，测试占有重要地位，它驱动在产品深层次解决这问题和保证正确的实施。测试甚至会评估，而且在许多情况下会测试为运行服务而设的底层政策。

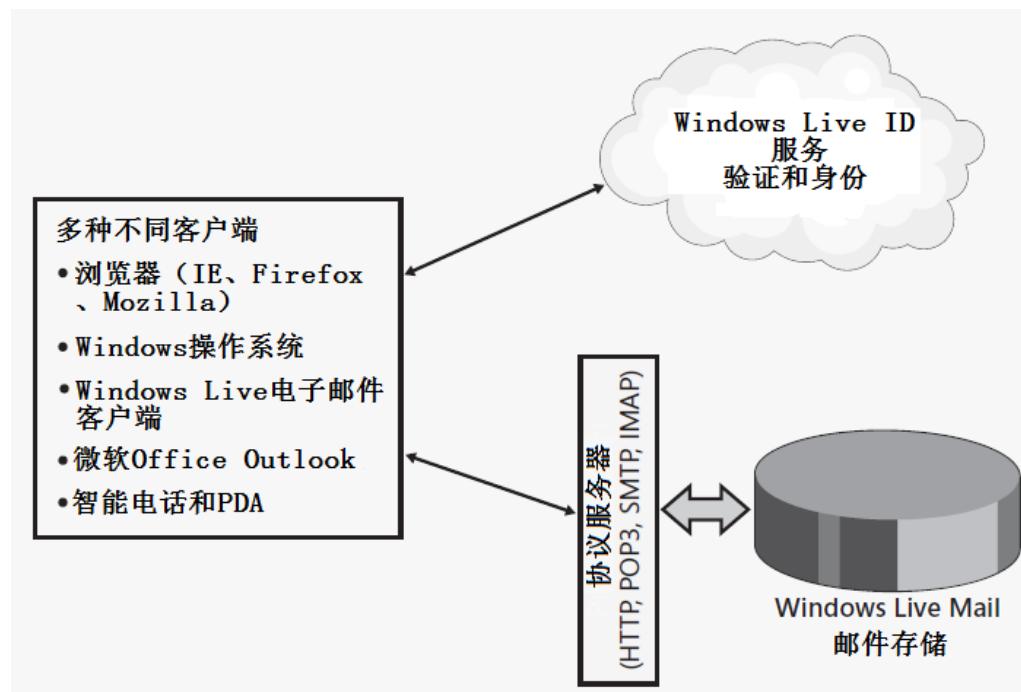


图 14-2 Windows Live Mail S+S 简图

从单机软件到多层次服务

在互联网早期，许多在线网站都基本上是自包含的。信用卡处理服务可能是一个例外，从一开始就有多个零售和在线端点。除此以外绝大多数其他服务都是作为独立服务来开发和运行的。从许多方面，每个在线财产都是一个完整的单家庭住宅。它有地板、墙体、屋顶、电线和管道。每个服务都需要建立所有部件，包括登录、身份验证、定制/个性化、存储、部署和报表机制。

有了多层次服务，创建一个新服务比建立一个完全的“住宅”要简单得多，但仍然具有挑战性。作为对比，在现今的世界发布一个新服务也许可以和卫生间翻新相提并论。微软和竞争对手的平台服务将会继续扩展，为新服务提供更多的基础设施。一个服务甚至可以从多个公司混合匹配基础设施服务，以期构筑最佳解决方案。随着平台服务不断更新，开发和发布一个新服务应当像给一个空卧室刷墙那样简单。

目前，许多消费者认为在线网站比如 Amazon 和 Ebay 很像独立服务，但它们包含许多模块和层次。比如 EBay，在收购 Paypal 服务之后，就把它深度集成到 Ebay 拍卖服务中，同时也允许其他服务使用它。对 Amazon 来讲，偶尔有一些服务，比如客户评论区或其他客户已购买产品的区域没有显示，但是你点击的产品仍然被显示并可以购买。这种情况的发生是由于这些功能中的每一个都是一个单独的模块，允许在不同的服务器上。如果他们不能及时响应，页面仍然会加载。通过层次化，这些服务能保证主要用户体验仍能工作，即使一些子部件不能正常工作。它同时也允许每一层次根据自己的日程创新，不需要建立对其他部门及其功能的依赖。图 14-3 显示了一个服务如何层次化的简化图。



图 14-5 多层次服务模型的简化图

在很多方面，用于测试一个独立服务的策略很容易，测试服务的每一个部分及其子模块，测试所有这些部分的每个结合点，并且测试经过整个系统的端到端的场景。这听起来好像不可能，但是当你拥有整个系统时，你可以控制什么需要更改以及何时更改。由更改引起的风险管理也因此更容易管理。

虽然经过多年，微软才把 Window Live Mail (前身是 Hotmail) 自有的用户验证服务迁移到与微软所有服务共享的验证服务，但它的确实现了。包括 WLM 在内的几乎所有服务现在都使用 WLID 进行身份验证。

层次化和集成使得建立和运行服务更加复杂。随着我们沿着 S+S 路径逐步深入，客户端的数

量以及它们有时与多种服务的交互增大了兼容性测试矩阵。再加上由第三方开发的诸多混搭（mashup），这些混搭与多种服务的公开 API 连接用以创建新用户体验，将使得集成映射变得庞大无比。

提示：混搭（mashup）是一种 web 应用程序，它可以将不同来源的数据和用户界面结合在一起，形成一个单一集成的服务或应用程序。例如：Zillow.com 服务结合了房地产销售的 Multiple Listing Service (MLS)、它本身的由用户产生的数据和用来映射数据的微软 Live Earth 服务。

微软采用层次化（Layering）技术开发其所有服务。许多新兴的服务都采用微软和其他公司的基础架构服务。在本章的第二部分，我会讨论测试软件加服务的技术，并侧重阐述分层的多重服务生态系统的测试。

第二部分：测试软件加服务

本章第一部分涵盖了大量关于服务的信息，包括微软有关服务的历史、服务对于微软来说是一个多么大的赌注、什么是软件加服务（S+S），以及它与我们以前所用模式不同之处。

在这一部分，我先针对服务的一些测试技术作一个综述，然后描述几个用于测试服务和与其相关的客户端的很特殊但不同的方法。

与前面章节中的许多例子类似，这部分中的例子是来自微软的大型软件项目，或大型服务项目。虽然例子来自微软，但技术可以而且应该用于各种规模的项目。

创新的浪潮

在这一章的创作中，我发现几篇文章谈到 PC 计算历史的创新浪潮。以下列出的是微软曾经经历并且参与竞争过的创新浪潮：

1. 桌面计算和网络化资源
2. 客户端/服务器
3. 企业计算
4. 软件作为服务（SaaS）或 Web 1.0 的发展
5. 软件加服务（S+S）或 Web 2.0 的发展

很清楚，我们刚开始这一最新的 S+S 的创新。每次创新浪潮都会带来新的挑战，软件测试必须与时俱进。其中大部分挑战来自于随着用户数量和应用场景的增长，需要支持的不同的客户端的数目和环境也在增长，如图 14-4 所示，每次创新浪潮，测试的复杂性和整体测试矩阵（test matrix）也在增加，测试的挑战在于矩阵将永远不会缩小。每次创新浪潮，都会在原有的测试基础上增加新的测试。

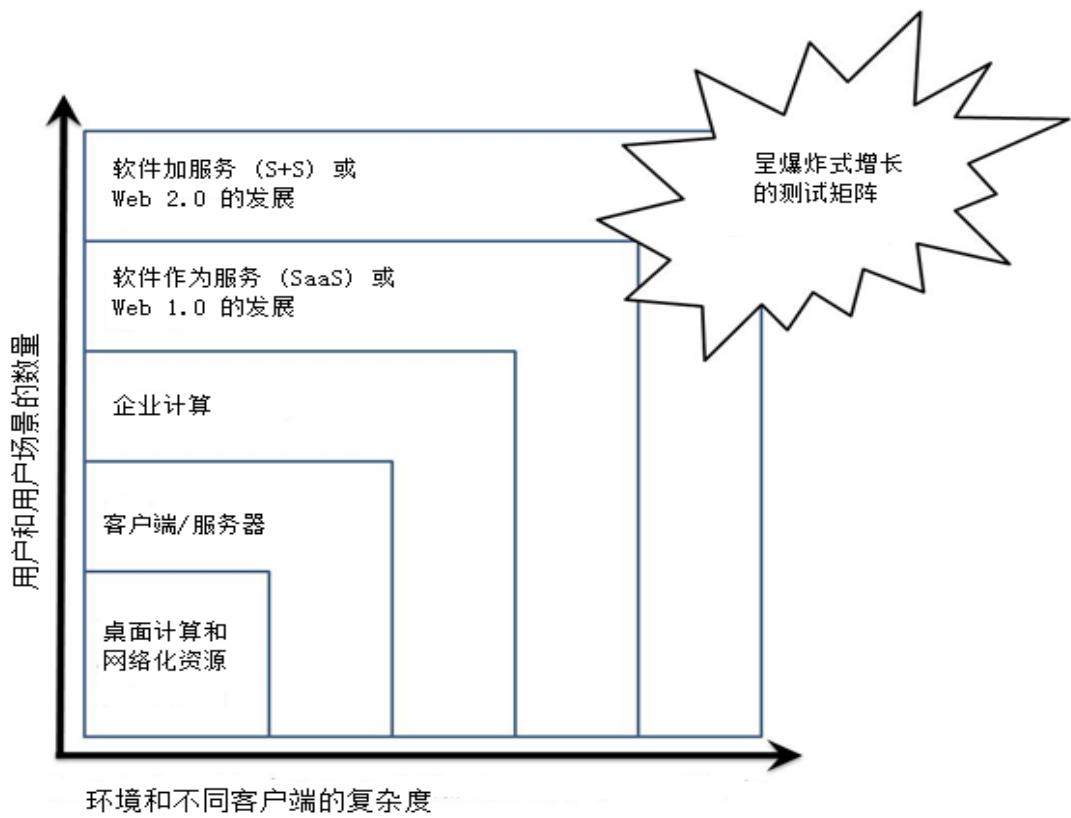


图 14-4 每次创新浪潮，测试矩阵（test matrix）随之增加

微软的桌面创新浪潮始于 MS-DOS 和 Basic 的编程语言。然后基于 DOS 之上开发出 Windows 并进入计算机互联领域。我们的测试也随着每次创新在不断改变。我们开发了许多新方法与流程来帮助我们更好地处理软件作为服务（SaaS）和软件加服务（S+S）的呈爆炸式增长的测试矩阵。

设计合适的软件加服务测试方法

如果写一本名叫《软件加服务的危险之旅》（*Dangerous Book for Software Plus Services*）的书将会很有趣。在我看来，这部书尽管外表精美（如黑色皮革和银色文字的封面），但却像一把没有钥匙的锁。读者如果想了解其中的秘密，需要弄清楚如何打开这把锁。之所以要加安全保护，是因为写出我的心得的同时，我需要共享我在开发、测试和发布服务中所犯的错误。从另一方面讲，测试服务成功的关键在于充分意识到各种危险，测试中要注意避免这些危险。我们在这部分讨论设计出良好的测试服务方法要注意的几个因素。

客户端支持

很多年来，在每个软件发布周期，微软测试人员不得不更新他们在测试中使用浏览器的列表。这些测试矩阵通常包括每个主要浏览器的最新版本和以前仍有相当大的市场份额的早期版本。

在一个 S+S 环境中，您的测试矩阵并不只有一个浏览器，同时也有很多的客户端应用程序。

例如，在测试 Windows Live Mail 中，我们在 Web 接口中包括所有浏览器，但我们也有关于 Outlook、Outlook Express、新的 Windows Live Mail 的客户端版本和各种移动设备。此外，我们测试矩阵中还包括客户端软件相关的因素：例如操作系统版本、语言和区域设置。

在实践中，控制客户测试排列组合（Test permutations）的复杂度，与一个简单的等价类练习不完全相同，但很接近了。审阅所支持的浏览器和客户端软件的市场占有份额，与市场的业务目标相比较，并增加一些操作系统的市场渗透率。产品团队应该可以从排列组合中去掉不支持的部分，然后确定剩余部分的优先次序。对于优先次序高的组合，进行更深入及更频繁的测试，而其余部分则只能进行一套基本兼容性测试。

建造于服务器上

微软在软件加服务（S+S）战略中最大的机会，是在服务领域里使用我们丰富的桌面客户端和企业服务器。在某些情况下，一项服务是从头建立在简化版的 Windows Server 操作系统上，有时候是在一个简单的 Web 服务器上。我们有很多企业服务器产品，如 Microsoft Office SharePoint Server 和 Microsoft SQL Server，都能为服务提供核心技术。

当一种服务建造在服务器产品上时，该产品中已有很多测试，所以我们不需要对其核心功能进行重新测试。但是，有两方面确实需要特别注意：

第一个方面（也是最明显的）是在服务器之上写的新代码，及服务与服务器之间的交互（interaction）的集成。大部分早期的缺陷会出现在基本功能和服务器产品的公共 API 中。在该部分代码稳定后，我们会经常会在新的 Web 服务组件和底层的企业服务器产品性能和可诊断性方面的问题。其中一个例子是在锁定（Lock）一个对象时，该对象并非为处理多个同时读取或写入操作而设计。作为为企业设计的产品，这样的场景通常是符合设计的或是可以被接受的错误，但是此类错误发生时，它所发出的晦涩难懂的报错信息却难以用来诊断错误的根源。

第二个方面是关于测试服务器产品本身在一型数据中心环境中的可管理性和可扩展性。我们在很多情况下，部署和配置企业服务器产品使得它在互联网级别的规模下仍能运行。经常需要额外的工作使得远程服务器的维修保养完全自动化，即无需人进入数据中心，接触实际的计算机。这些方面的改进可以降低运营费用和帮助服务项目盈利。因此，测试不仅仅是着重于发现错误，而且要为下一次升级或发布寻找改善服务器产品的方法。

服务平台与顶级服务

在第一部分，图 14-4 显示了微软的服务架构如何变得更加分层化（layered）。我们知道我们主要的竞争对手也是如此。这种建构服务的方法与操作系统的分层架构相似。甚至与网络开放系统互连基本参考模式（OSI 模型）相似。在较低层中，你会发现关键平台服务（例如，WLID，Windows Live ID）。直接产生收入的品牌服务如 Office Live，则在较高层中。在虚拟地球（Virtual Earth，maps.live.com）服务背后是一系列的基础设施服务，采取所有卫星和低飞鸟瞰的照片，并把它们划分成小块（tile）。

图 14-5 是一个概念性的例子，AuctionCloud.com 的混搭集成了 eBay 和 Zillow 的服务。Zillow 本身就是其他几个服务的混搭。随着层次堆栈的加深，所有权可能横跨多个组织和公司，调试服务中的出现的问题可能更加复杂。

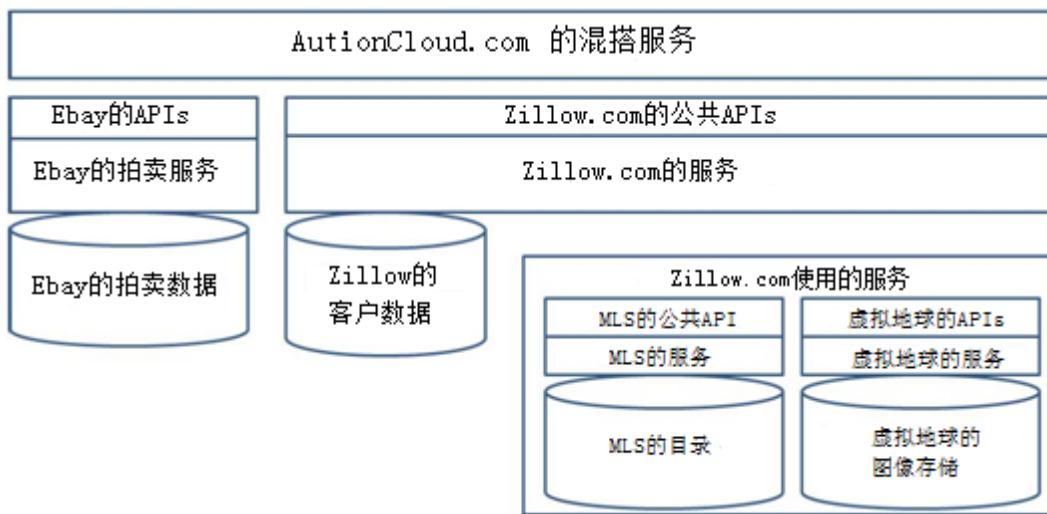


图 14-5 AuctionCloud.com 混搭基于 Zillow 和 Ebay; Zillow 则基于 Windows Live Service (MLS) 和虚拟地球 (Virtual Earth)。

在堆栈 (stack) 的底层我们最终会见到平台 (有时称为基础) 服务。PayPal 和信用卡处理都是电子商务站点中常见的服务平台。身份验证服务如 WLID (Windows Live ID) 也是服务平台。测试一个平台服务需要一个内部的焦点。这很像一个公共 API: 你知道很多开发人员将调用该 API 和使用 API 的几乎每个功能, 您只是不知道他们究竟会如何使用它。测试服务平台着重于每个服务的入口点。在验证了 API 的内部功能后, 一种好的测试方法是选几种现实世界中集成 (integration) 的场景 (scenarios) 来测试, 但不能测试所有可能, 因为那样就算不是根本不可能, 也是不切实际的。

在测试服务平台时, 重点仍是内部的一致性测试, 而不是集成测试。在较上层的服务中, 集成测试可能会成为主要焦点。

在堆栈中较高层的服务中, 测试工作会花相当多的时间集中测试与所有服务平台的特定集成情况, 而不是服务平台本身。上层服务对一个服务平台所需要的测试程度取决于两个服务之间的耦合 (coupling) 的程度。

松散耦合 (Loosely Coupled) 与紧密耦合 (Tightly Coupled) 的服务

这一节本来可以轻松的命名为“松散依赖性或紧密依赖性”, 但是耦合用于服务还将面向对象的设计思想和组织耦合结合在一起。Doug Kay 在他的《松散耦合: Web 服务所缺少的环节 (Rds Associates, 2003)》一书中提及本主题。他在这本书里论述了跨服务耦合对于很多项目管理的影响。

耦合对服务来说既是项目管理, 又是设计元素, 其原因与服务发布的速度直接相关。在每个发布周期, 分层的服务会不断增加新的依赖项和更改现有依赖项。

提示: 在计算机科学中, 耦合 (*coupling*) 或依赖 (*dependency*) 是指每一个程序对另一个程序的依赖程度。在源程序或目标程序频繁更改的情况下, 松散耦合的系统是最好的。

理想情况下, 服务应是松散耦合的。松散耦合是软件设计中经常使用的一个术语, 来描述接口, 组件或系统, 对所依赖的系统做最小的假设。服务之间的依赖越少, 每个服务就可以更

加独立地创新和发布。在这个意义上说，耦合对软件项目的影响是一种依赖性管理挑战。软件项目耦合程度越高，所需要的项目管理的开销就越大，以确保高质量、高价值的用户体验。下面有两个来自 Windows Live ID 的例子。

示例 1：松散耦合：Windows Live ID 经过身份验证的用户

例如，使用 Office Online 时，用户可作为匿名用户拜访网站（office.microsoft.com），仍然能获得大量的价值。一些任务如查找和下载模板不需要任何身份验证。其他功能如注册某些通知，则要求用户是已注册用户。

一个 Office Online 已知的注册用户，是一个已通过 WLID 的身份验证并拥有一个有效的 Cookie 的用户。Office Online 检查加密的 Cookie 以确保用户身份的有效性和可信度。这里，只使用 WLID 进行身份验证，而没有对 WLID 的公开功能的硬性依赖，使二者之间的关系非常松散。WLID 可对其服务做很多更改，Office Online 团队只需要验证阅读加密的 Cookie 的核心功能没出问题。如果 Cookie 的数据结构设计得合理，就算是对该数据结构做扩展，只要不触及基础数据元素，都没有关系。

示例 2：紧密耦合：Microsoft Passport (WLID) 家长控制功能 (parental controls)

当我们发布 MSN 9 时，我们在 MSN 客户端启用家长控制 (parental controls)。为此设置，我们不得不在 Microsoft Passport 实现层次结构帐户的概念。此外，此功能又被另一个个性化设置服务平台所使用。

对这个新的功能，MSN 客户端、Passport 和个性化设置团队不得不紧密合作。他们不得不决定谁会做什么工作，如何调用每个服务，及来回传递信息的格式。在决定哪一队会提供每个组件后，他们不得不在自己的日程和计划中加入足够的时间，以保证每个团队都能完成集成测试。

MSN 客户端团队能够在家长控制界面上工作，但其实不能编写集成或测试的代码，直到 Passport 团队在相关部分作出明显进展。对于测试而言，这是一项重大挑战，因为测试团队要确保他们测试时，使用的服务和客户端的版本都正确。

这两种情形下，对这两个服务，松散耦合明显地会提供更多的灵活性和自主性。即使在微软，紧密耦合的共同开发方案都很难管理。在互联网上横跨多个公司来实施此类项目则更具有挑战性。

我们的许多站点都与外部公司集成。其中一个例子是信用卡处理。我们通过在互联网上的服务进行信用卡处理，我们也会接受 PayPal 付款，我们也可能会与很多零售商的外部网络 (Extranet Web) 服务集成。拥有明确的接口的松散耦合项目，常常能按时发布。而紧密耦合的项目往往会陷入困境，造成产品发布遥遥无期。

在我们自己作努力发展的同时，每个非微软机构也在继续发展其服务。如果没有一个具体实例，紧密耦合的服务平台很难测试。松散耦合的服务可以经常用存根 (stub) 来调用其他服务或使用仿真程序 (请参见本章后面部分“集成测试与测试标志和模拟”)。这将允许在没有一个实例的情况下进行测试。这样，集成测试还是做，但服务的核心功能测试也得以进行。

无状态至有状态（Stateless to Stateful）

无状态的服务是那些不需要存储任何数据的非常简单、快速交易的服务。服务需要维护的状态越少，它的组件就会允许经历更多故障，同时又继续提供良好的用户体验。举一个软件例子，Outlook 发送电子邮件（e-mail message）：如果信息第一次发送失败，Outlook 会试着重新发送，用户很少会注意到。

当服务完成交易的时间很长，并且需要存储特有的用户或重要商业数据，它就更有状态性，因而更容易受失败影响，例如，您用 Microsoft Word 写作文档有好几个小时，然而您正试图保存该文件时碰到崩溃（crash）。可能只是一次崩溃，但对该用户的影响相当大。

无状态服务（Stateless Services）

基本的网络搜索是服务中的一个相对无状态的一个较好的例子。每个搜索查询完成并呈现结果，通常在亚秒级（subsecond）内响应。对于任何大的搜索引擎，用于支持服务的索引每天每小时都在更新，并在多个数据中心之间进行复制。即使索引可能会有状态，失去某个索引的实例（instance），也不会使用户产生无法挽回的损失。至多，一些搜索查询将无法返回结果，直到损坏的索引被替换。

状态服务（Stateful Services）

CRM Online 和 Salesforce.com 是有状态的服务的典范。这些服务有助于中小公司业务在互联网上进行自动化销售和客户支持，允许他们直接与其客户进行交互。他们通常存储关键商业数据：如一家公司的联系人名单、报价的副本和最终合约的副本。服务的用户可以打开一个报价模板，花费一个小时来填表，然后保存并与客户共享。市场营销活动可能会持续几个月。在这些类型的服务中，用户活动和交易会持续很长时间，因此这些数据会在系统中保持很长的时间。衡量服务的质量范围很广，下至最小组件上至灾难恢复的能力都起着关键作用。

发布时间还是功能和质量

对于任何软件产品或 IT 项目，必须在发布时间与质量和功能之间取得平衡。在商业服务领域，先行者（同类服务的第一家公司）可以在一段很短时间内成为事实标准。例如，在电子商务、拍卖、视频和社交网络服务各领域都由少数早期的成功公司主导。他们的名字耳熟能详：Amazon.com、YouTube、eBay、Facebook 及 MySpace。这些公司都得益于最先发布和快速跟进。战胜并取代现存的服务可能比发展和推行下一个可能成功的服务更难。

Friendster 在 2002 年 3 月推出，并被认为是第一个主要社交网络网站。MySpace 和 Facebook 晚来了几年，但当他们推出时，他们改变了这场比赛。在社交网络，自我标榜（self-promotion）被大众接受，这些服务得以跳过 Friendster 作为先行者的劣势而极大地领先。

制定测试策略时需要考虑市场的状况以及明确推出的质量标准（quality bar）。有时更快的进入市场是正确的决定，但根据我的经验，在低质量的服务推出后，要使其在可操作性、可管理性各方面以及整体达到高质量可能需要很长时间，数以年记。如果可能的话，我会尽可能，在推出更多功能的同时尽量把服务更快推向市场。

发布频率和命名

对一个主要发布来说，要决定在什么时候发布，和发布之后做什么。经常一项服务在启动时，会在其名称后附加试用版（*beta*）。一个突出的例子是，Gmail 服务在 Beta 版中的花了三年时间。这允许发现该服务的很多主要错误而不会太大影响客户忠诚度。

提示：服务标题中有试用（*beta*）字样是在清楚的向用户表明：服务中还有缺陷，但现在已有足够功能让用户使用。

团队常用的另一个技术是在一个主要发布后迅速计划下一次发布。每月或每季发布过程意味着当前版本发现的缺陷会在不久解决。作为测试人员，可以允许在发布产品中有并不重要的缺陷，只要你知道不久以后会修复。如果缺陷是已知的，并且在网站发布后几小时内必须修复的（我们通常称为一个紧急补丁 *hotfix*），我建议暂时停止发布，直到修复这个重要缺陷。

软件加服务测试技术

在对影响测试方法的各种因素包括设计、命名以及商业因素等做过仔细考虑之后，是讨论驱动测试流程的整套测试技术的时候了。这一节我们将更深入的发掘软件服务所需的不同测试技术并对软件加服务的危险性问题给出一些答案。

全自动化的安装部署（Deployment）

在第十一章“非功能测试-Non-functional Testing”中我们提到，微软对每个正在开发中的软件，每天都会建立一个供测试用的内部版本。对游戏、服务器、移动和应用软件来说，当这个测试版本结束编译并被放在共享的文件服务器上时，自动化的测试程序就开始工作，把测试版本安装在测试计算机上，并执行成千上百的测试程序。但对软件服务来说，却并非总是如此。

许多负责微软软件服务的测试团队已经开始衡量从测试版本的建立到完成第一次安装部署的时间。我们将在下面的“服务的性能测试度量-（Performance Test Metrics for Services）”一节中详细讨论。

如果你曾经从一个开发工程师手里拿到一个他自己建立的测试版本，并且他说：“复制这个文件到这里，增加此项到配置文件，并且登记这个.DLL”。你就知道这需要额外时间并且是非常易出错的。一个完全的常规安装流程是使测试团队迅速安装并进行测试的关键所在。这在软件服务领域也是一样的。无论是在一台机器（one-box）还是在多台机器上的测试环境中，还是最后在软件服务的运营环境下（production）都需要安装部署过程自动化（请见下面的“一台机器-The One Box”一节）。安装程序的代码无论在哪一个环境中都是一样的，只是通过不同的输入 XML 文件来决定不同的配置。

不得不承认的一个现实是：许多产品开发团队常常认为只需要安装部署一次，那就是到最后，把软件服务的最终版本放到运营环境的服务器上。当一个团队犯这样的错误时，他们把安装部署代码的编写推迟到开发周期的最后。由于这些程序代码不是由数据驱动的，因而它可能是不完全的，并且得不到测试。像对任何一个头等的服务功能一样，应该有专人来开发和测试安装部署程序。

软件服务的运营团队倾向于以下三个技巧。作为测试工程师，我们应该设法在这三个方面发现缺陷。

技巧 1: 如果运营团队不能只双击一下安装程序然后就等候安装部署完毕的话，那在安装程序中一定还有缺陷。也许这些属于设计缺陷，但它们仍然是缺陷。

技巧 2: 如果给运营团队的安装部署指南里，除了“到这里点击安装程序”之外还有别的指示的话，那安装部署指南里一定还有缺陷。

技巧 3: 如果在实际安装部署时，需要有两个以上的工程师的话，那一定是安装程序和安装程序指南中还有设计缺陷。

好的安装部署是卓越运营的关键

每当我的团队开始一个新的软件服务开发时，我会有至少一个开发工程师和一个测试工程师专门负责其安装部署程序的开发和测试。如果做得好的话，这种级别的投资会在未来的发布版本中大大减少。

安装部署行话：一个好的安装程序的关键特征是：零停工期、零数据损失、部份成品更新（或混合模式）、滚动式更新和快速反转机能。

- **零停工期：**早期的在线服务在更新时，经常会有一段时间甚至一个星期以上停止服务。今天大多数的在线服务在软件更新时仍然继续提供服务。
- **零数据损失：**是针对那些拥有很多用户数据的状态服务（*stateful service*）来说的。当需要进行大规模的数据模式变换时，更新状态服务面临着巨大的挑战。
- **部份成品更新：**是对那些只针对部分运营服务器更新的模式来说的。也许只更新 50%，甚至 10% 或 5% 的运营服务器。
- **混合模式：**有些在线服务只更新一部分的运营服务器，但对用户来说，他们能得到同样的高质量的服务不管是用旧的还是新的服务版本。虽然混合模式也许只在更新时才出现，但其实这正是软件服务设计的要点。当用户界面发生很大变化时，混合模式确保那些得以使用新的服务的用户每次访问服务时都能继续体验新服务。
- **滚动式更新：**是指软件服务能够自动更新一部分在线运营服务器，而没有任何的停止服务期。说白一点，这种更新是按计划进行的，可以是每隔几个小时一次，或是每隔几天一次，但全部都自动完成任务。
- **快速反转机能：**当更新服务出现问题时快速反转机能是我们的安全网，比如在更新进行到中途时，发现了一个不能接受的缺陷，不管是完成了 10%，还是 90%，都需要快速返回到上一个服务版本。

我经常提醒我的运营团队，一个大型在线服务的安装部署的时间不能少于三天。这是因为我们需要控制运营风险。我不止一次经历过这样的事情：我们在安装部署一个本属于“小风险”的更新服务时，还没有进行到 10% 时就不得不停止，因为发现了一个不能接受的大缺陷，不得不返回到原来的版本。

部份成品更新提供了一个新版本和现有版本的混合模式来让在线服务能够控制风险。滚动式更新和部份成品更新相似，只不过采取全自动的方式。如果在安装部署的过程中发现了需回收产品的大缺陷时，快速反转机能可以使服务快速回到上一个良好的状态。我们许多的服务都具有这个自动化的流程，所以我们可以同一天里运行多个有交叠的更新服务。实际上，有些服务甚至能够在预定的服务发布期前自动安装部署运营服务器。这样的流程允许服务器同

时拥有新的和旧的版本，能很快地前进（从新的目录里读取）或是后退（从旧的目录读取）。

提示: Windows Server 2008 操作系统现在已经正式发布了，虚拟机的性能比以前有着显著的提高。虚拟机是指在一台 Windows 操作系统上运行一台虚拟的 Windows 操作系统。我们许多的服务团队已经开始使用虚拟机来运营软件服务。在今后的几年里，这大概会成为一种快速和全自动化安装部署的普遍模式。

测试环境

当我们测试服务、终端、或是服务和终端的集成时，有一个适当的测试环境是非常重要的。如果测试环境和最终的服务运营环境差别太大，则可能会漏掉许多缺陷。如果测试环境和运营环境一模一样，那需要昂贵的费用，就算是建立成这样的测试环境，那大概会把所有的测试都集中在这个环境中。

大多数微软的软件服务测试团队都拥有几种不同的测试环境，针对不同种类的缺陷，在不同的环境下测试。有效的定义和使用不同的测试环境需要真正懂得软件服务的构架和集成附属条件。在开发周期的早期定义和建立测试环境的流程，有助于测试团队最大限度的利用他们的测试硬件并通过在最佳环境中做适当的测试来减少测试风险。

一台机器

有时我们把只需要一台机器的测试环境叫做单个机器，单个操作系统，或者单个操作系统的测试环境。大多数时候，我们称它为一台机器。当我们和其他团队一起工作把他们的服务和我们的集成起来时，这些团队常常问我们是否有一台机器的测试环境给他们使用。

据我所知，微软的 MSN Billing 2.0 团队在 2002 年，最早使用“一台机器”作为测试环境的名字。那时他们正把 1.0 版本的数据转移到 2.0 版本上并安装部署 2.0 版本到一系列新的机器上。这是一个很大的项目，而且必须按时发布，要不然 MSN 的更新版本就会错过节日销售的旺季。安装部署程序是在最后开发的。很多安装步骤都没有自动化，所以我们有一个很长的安装部署指南。后来我们把它叫做“纸上的安装部署向导（Wizard）”。

关于纸上的安装部署向导这个奇妙的主意，我只想说这么多：纸（paper）和向导（Wizard）这两个词永远都不要放在一起用。

经过好多天，一边参看安装部署向导，一边加上几百条注释，整个安装部署过程才结束。这不是那种一个人想到的点子，或者是那种只有少数人才有的深刻见识，对参与其过程的人来说，无论是谁都很清楚，我们必须把安装部署程序当作头等产品服务的功能来开发测试，就像我们对待其他的功能一样，每天都测试。

后来 MSN Billing 2.0 团队就专门集中开发了一个基于 XML 数据的安装部署工具。它使得开发工程师、测试工程师和运营工程师都可以在多种环境中方便快捷地安装部署。用这个数据驱动的工具，他们可以定义在一台机器上的各种服务所需的配置。这个配置被大家认作“一台机器”，这就是这个名字的由来。

随着在基于 Windows 的系统上使用虚拟机的普及，我总是需要指出的是其实“一台机器”的测试环境已经不再专指一台机器，而是指一个操作系统的测试环境。开发工程师常跟我说：

他们有时在一台机器上运行十台虚拟机。这样那就不是一台机器的测试环境啦，应该是一个测试集群运行在一台机器上。

对测试工程师来说，一台机器的测试环境可以满足测试软件服务的主要功能的需要。对 WLID（Windows Live ID）和收费系统来说，我们有一台机器的实例环境。实际上它们在运营时由几百台的运营服务器来提供服务。但我们可以把这些服务安装部署在一台机器甚至一台手提电脑上。

一台机器的测试环境可以让开发工程师进行单元测试以及新代码的预前测试。能够同时执行 BVTs 和大规模的自动化测试程序对测试来说是非常重要的。因为能够很快地建立、拆除和重建一台机器的环境，这就增加了项目开发测试的敏捷性。这样的测试环境对一些简单快速的回归测试很有帮助，比如说：对只需改变一个字符串的缺陷的回归测试。

对任何服务来说，一台机器的测试环境是测试中仅次于安装部署全自动化的重要要求。

测试集群

测试的第一个要求就是要有数据驱动的全自动化的安装部署。利用数据驱动方式，就可以把对大规模运营服务器的安转简化成对一台机器的安装。下一步就是自动安装部署在测试集群上。相对运营集群来说，测试集群是相对小规模的，但每种角色的服务器都应该有一台，如图 14-6 所示。

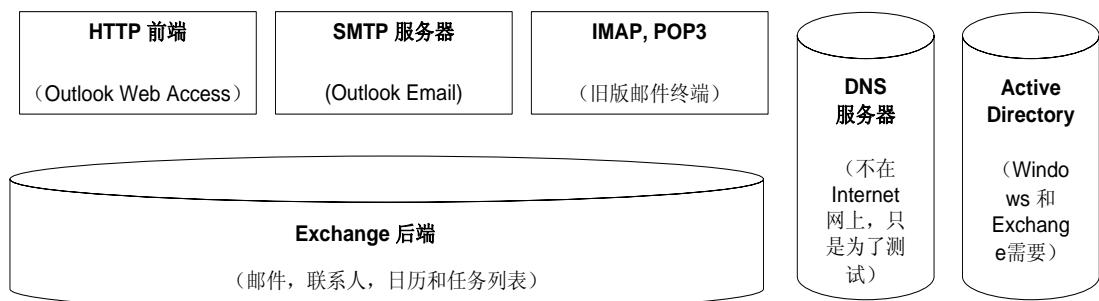


图 14-6 一个建立在 Exchange 服务器基础上的软件服务的测试集群例子

服务器的角色（machine role）是指各种服务器在软件服务中担当的作用而言的。一个简单服务如网络搜索，就有很多服务器角色，如搜索服务器：在网上搜寻英特网的内容以供索引，然后索引服务器再根据这些内容建立关键字索引，还有查询服务器，接受用户的搜索要求从索引服务器中找到结果。

在测试集群环境中，我们不是要找服务的核心逻辑的缺陷，而是要找服务器之间交互的缺陷。有些服务，比如 Microsoft Internet Information Server (IIS)，就可以查询一个本地的 Microsoft SQL Server，因为它们都共用一个有管理员权限的帐户。当这些服务配置在不同的服务器上时，我们会发现性能、管理权限和配置设定方面的缺陷。而这些缺陷是在别的测试环境下不能被发现的。重要的一点是：用一台机器的测试环境来测试所有能够测试的方方面面，然后用测试集群环境来测试在一台机器上不能进行的测试。

频繁地共同享用测试集群并不总是有意义

这是我的机会来提起这个有争议的话题，但请听听我的意见。当我参与公司里测试工程师的面试时，我总是会寻找那些和开发工程师不一样的人。每一个杰出的测试工程师的 DNA 里都有着一种不找到好的缺陷不罢休的愿望。我喜欢把软件搞得不能工作了，而且不是一般程度的不能工作。这就是为什么我认为测试工程师是世界上最好的工作的原因。

我不能理解的是：有些团队会建立一个大的测试集群，然后把他们测试时间的 60% 到 70% 都花在那个环境中。这其实是把一个好的测试团队的每个人都固定和限制在那个环境里。这些原本想方设法寻找缺陷的测试工程师突然被很多规则所约束，这些能做，那些不能做等等，全是因为不能把测试集群弄得不工作了。因为如果测试集群环境不能运行了，所有的人就不能测试以致影响项目的进程。

如果你想知道你的团队是否在测试集群上投资过多，你只要停止运行它一天，看是否你的项目会因此拖延一天。如果你会因此拖延一天，那意味着你的测试团队需要多种的测试环境。

即使你的团队有不同的测试环境，测试工程师应该被允许不在共用的测试集群里测试。这样的话，他们可以远离代码自由发挥他们的测试想象力。我更倾向于让测试工程师建立他们自己的测试环境而不是共享测试环境，这样一来，他们就可以控制这个环境，可以对代码使各种坏招。测试集群应该被当作集成测试环境来用，其他的测试则应当鼓励在独立的测试环境中做。所以，一定要小心，不要把一个“捕食者”式的测试工程师变成一个“温顺”地记录缺陷的人，那就把他们的 DNA 改变了。

性能和规模测试集群

像一台机器那样简单的测试环境也可以被用来做性能测评，甚至测试网页加载时间（page load time）测试（在本章后面我们会讨论这个话题）。性能和规模测试集群是用于不同种类的系统性测试。对在线服务来说，当系统的不同组成部分被分散在不同的服务器上时，有一些部分会早于其他部分先垮掉。对服务进行性能测试，是为了能够找到运营时平衡的硬件分配，以产生高效和高性能的系统。

软件服务面临的另一个挑战是在做向上扩展（scale up）和向外扩展（scale out）时，它们可能会成为瓶颈。向上扩展（scale up）通常是指系统的数据规模增长了。一个能处理 10 封电邮的邮箱和一个每天处理几千封电邮的邮箱是不同的。当数据量增大时，很多服务的性能会别变得很慢或者干脆停止工作，因为数据量增大到超出预期值或是超出当初的测试范围。

向外扩展（scale out）是指服务系统能够通过增加服务器来增加系统容量。在本章前面讨论过的规模单元法能够帮助我们建立服务的逻辑购买和运营单元，能够持续向外扩展。规模测试的目的是为了找到一种可以花最少的钱，配置恰当数量的各个类型的服务器，和各个服务器的配置硬件及操作系统，以建立最大容量的服务系统。这种测试对提供初期的容量模型很有帮助。

性能和规模测试都是要大量使用硬件系统的活动。其他的一些测试比如负载测试和能够确定服务失败的平均时间（MTBF）（关于 MTBF，请参看第十一章）等测试，通常能够得益于使用和运营硬件一样级别的硬件系统。因此，很多团队都使用一套共用的硬件来执行负载测试。

集成服务的测试环境

在多层次的服务世界里，总是能够在集成测试阶段发现缺陷。应该在何时以及怎样执行集成测试实际上取决于服务之间的耦合度。两个服务之间耦合越是松散，就越没有必要做跟实际环境一样的集成测试。

建立 INT（集成）测试环境

回想当年 MSN 和 AOL 竞争激烈的时候，我们在业界的评比回顾中老是输给他们。其原因就是 MSN 的整体用户体验落后于人。所以我们制定了一个目标，就是在下一个版本中让所有的 MSN 的服务互相整合顺利，提供一个高质量的整体服务。

MSN 的测试经理们那时经常在服务运营中心（Service Operations Center-SOC）后面的一个会议室里开会。运营中心里布置了一圈的计算机有点像大学课堂，所有的计算机都对着墙，墙上映照着三张大图：网络交通图、警戒信号图和需求票图。透过会议室的玻璃窗，我们可以清楚地看到这三张图。

在会议室里我们正激烈地讨论怎样才能把各种服务更好地整合集成起来。我们有一些新的服务比如家长控制机能以及共用的跨服务和客户端的联络服务等。Friedbert 和 Monte 是我们中间最资深的测试经理，他们都提议建立一个集成测试环境（Integrated Test Environment – INT）。这个共用的 INT 测试环境要求我们每一个团队在一个共享网络中放置一个测试集群，这样我们就可以用彼此最新的服务版本来测试。

这是很让人头疼的一件事，但我们最终还是建立了 INT 测试环境，在那年的大多数服务的回顾评比中 MSN 胜过了 AOL。

TellMe 在 2006 年被并入微软。他们也用了集成测试环境，这样他们的集成伙伴可以开发和测试与 TellMe 服务之间的整合。他们把它称作“绿区”，这样的方法在软件服务开发初期阶段行之有效，因为初期会有许多服务同时处于开发阶段，变化很快。

尽管 INT 测试环境对一个项目早期的多服务的整合测试起关键作用。但它同时带来了两项问题。

第一个问题是：INT 测试环境不是运营环境，也永远不会和实际的运营环境完全一样。很多测试工程师说：他们需要在 INT 环境里测试，因为它像运营环境。这就是问题所在，因为它像运营环境，但却不是运营环境，所以有些缺陷就会被漏掉。

当共用的测试集群环境被引入 INT 环境时，第二个问题就出现了。这个共用的测试集群指向 INT 环境，所有测试工程师的测试都在这个环境下执行。当 INT 环境出毛病停止工作时，挑战就来了。因为 INT 环境里安装部署的服务的版本都不是最后的版本，比起运营环境来更不稳定，会出很多的问题甚至停止工作。如果一个测试团队把他们测试的重心放在 INT 测试环境，他们会发现他们会经常被耽搁而赶不上进度。

如果一个项目是高度耦合服务型的、或是需要跟客户终端应用程序深度集成的话，那的确是要 INT 环境的。成功的要点是要确保 INT 环境有明确合理的质量要求，而且在 INT 环境出

现大问题时，所有的团队都应该有 INT 以外的测试环境。

INT 测试环境是一个强有力的测试选择。但成功的 INT 测试却需要你在没有 INT 的情况下，仍然能够对大多数的服务展开测试。

安装部署测试集群

我在前文中已经讨论过：全自动化地安装部署是测试顺利进行的一个重要步骤。一台机器的环境可以让开发和测试工程师独立工作，也是任何服务签入前测试的一部分。但这种测试却并不保证安装部署代码在多台服务器的环境里能够正常工作。这正是为什么需要安装部署测试集群。

就一个规模单元里的机器数量而言，一个安装部署测试集群和运营环境是相似的。因为安装部署的测试并不和硬件有太大的关系，所以它们可以在相对便宜的计算机上，甚至虚拟机上执行。我大力推荐虚拟机，因为它们提供最大的灵活性。

安装部署测试的侧重部分：

- 确保在运营环境下安装部署程序和所有的新功能像预期一样的工作。在这种情况下，软件要么和 INT 环境相连，要么和运营中的服务环境相连。在有些情况下，仿真或是测试标志的方法（后文会详细讨论）虽然会被选择但并不是最佳方案。安装部署测试确保其软件服务的安装部署的自动化以及和平台服务的联接的自动化。
- 安装部署测试环境对缩紧系统所需的各种服务是很理想的。Windows Server（或其他的企业服务器）的默认设定常常会安装很多在运营环境下不需要的服务。你可以在安装部署测试环境中系统性地关闭或卸载不必要的服务。这通常被当作一种安全性测试，但最好是在安装部署测试集群中执行。

如果用商品硬件或是虚拟机来建立安装部署测试集群，不应该很贵。能够不停地用不同的设定来测试安装部署程序对软件服务测试来说是非常有益的。

测试环境的总结：

不同类型的测试程序最好运行在对其最佳的测试环境。我们把测试和测试环境分开如表 14-2 所示，主要是目的是为了能够在最简单的测试环境里最快地找到不同类型的缺陷。大部分的测试主要发生在一台机器和测试集群的环境里，但所有的环境都对开发和发布新服务，以及已有服务的升级更新起着重要的作用。

表 14-2 测试环境列表 测试环境	测试重点
一台机器	签入前测试
	基础功能测试
	自动化测试程序开发
	大规模自动化测试套件的并行运行
	快捷安装部署检验

	网页加载时间
测试集群	全面功能测试
	全面自动化测试套件
	服务组件之间的网络检验
	失败转移测试
性能和规模集群	网络间的核心性能测试
	向上扩展和向外扩展
	初期容量规划验证
INT 测试	多种服务之间的集成测试和系统场景测试
安装部署集群	在大规模的集群或是虚拟机集群上的安装部署测试

集成测试（Integration Testing）与测试标志（Test Flags）和仿真（Emulation）

正如前面所提到的，服务本身已经不再自为一体。这些服务常常建立在服务平台之上，因而必须要借助于服务平台去发挥它们所有的功能。由于对方方面的依赖性，对服务不仅要测试它与其他系统的集成点，而且在测试时还要保持每一个服务或者一个服务层面其自身的独立创新能力。集成测试、测试标志和仿真则是面对这种挑战的解决方法。但是，最终用哪种方法最合适，还要取决于服务的耦合度。

仿真/ 测试标志		集成测试	
松散耦合	主要 功能开发以及 大多数测试	次要 用于产品周期 中后期的集成 测试	
	次要 多数情况下用 于回归测试	主要 功能开发以及 大多数测试	

图 14—7 选择集成测试、测试标志和仿真测试的 2 × 2 矩阵

敏捷灵活的测试标志和仿真

标志测试和仿真都可以不用借助服务平台而达到测试服务的目的。但是，要想使一台机器或测试集群真正达到快速测试或验证内部缺陷的目的，服务就必须能够去除其对服务平台的依赖性。测试标志和仿真就是通过不同的途径来实现这一目的。

测试标志

测试标志从内部掐掉了对服务平台的调用。测试标志是通过在配置文件或在 Windows 注册表中设一个全局变量来设置或取消的。如果这个测试标志被设置为真，原来调用服务平台的代码路径将被跳过，取而代之的是一条新的设计好的代码路径从而达到完成测试的目的。

采用测试标志的代码路径通常都会给予正面反馈，返回一些简单的、静态的格式化数据。当然，也有些测试代码会返回错误和可变数据，但这种情况通常比较少见。测试标志对测试工程师来说是一种相当不错的测试方法，可以在新的版本上对一些简单的缺陷做快速回归测

试，比如对报错信息的订正和对用户界面的修改，实质上就是那些与系统集成没有关系的缺陷。测试标志也可以用于压力测试（**Stress Test**）。

新闻报道组群的压力测试（**Stress Test newsgroups**）

微软的 **Office Online**（不久就会更名为 **Office Live**）是当今互联网中最受欢迎的网站之一，微软 **Office** 的用户到这个网站来寻求帮助，寻找艺术剪贴，文件模板，软件培训和用户经验交流。

微软的 **Office Online** 不但要集成来自新闻报道组群的报道内容，还要维持用户界面。这个功能就叫做 **MSCOM 多线索讨论服务**。它其实是一组 API，可以让 **Office Online** 服务不需要离开 **Office Online** 就可以读取和张贴新闻报道组群里的信息。为了测试其功能，通常会建立一个假的新闻报道组群，这样一来，大部分的用户案例就都可以测试了。

如果要进行压力测试或者负载测试，有一点是很清楚的，那就是靠在真正的或者假的新闻报道组群中狂发消息是不可行的。测试团队通常会设置多种测试标志使其既可以做手工测试也可以通过本地的一个.xml 文件来进行性能测试。

加入测试标志不仅使性能测试不再受阻，而且使得手工测试核心功能变得更加灵活。下面的例子就是我们设置或取消测试标志的方法之一，它是通过在适当的位置简单地放置一个文本文件（**text file**）来实现的。在许多环境里，我们采用一个被叫做 **Jukebox** 的共享服务来全局设置或取消测试标志。

```
///<summary>
///Return thread messages for the given asset.
///If file exists this is a test so use file.  If not, use service.
///</summary>
Private string GetThreadMessages(string strAssetId)
{
    string strPath = Path.Combine(Path.Combine(m_strDataPath, strAssetId),
        "ThreadMessages.txt");
    if (!File.Exists(strPath))
        return null;
    return ProcessFile(strPath);
}
///<summary>
///Get message bodies for the given asset.
///</summary>
private string GetMessageBodies(string strAssetId)
{
    string strPath = Path.Combine(Path.Combine(m_strDataPath, strAssetId),
        "MessageBody.txt");
    if (!File.Exists(strPath))
        return null;
    return ProcessFile(strPath);
```

}

—Marty Riley, 高级测试主管, Office Online

仿真

仿真是大家更喜爱的测试服务的方法,因为它既可以找到系统集成缺陷,又可以保持测试的敏捷性。通过模拟平台服务(platform services),由于xml格式错误而引起的缺陷就可以在系统集成测试之前被发现。

一个典型的仿真方案会先在一台机器上运行仿真服务而不是直接去测试服务。在一台机器(One-Box)测试的情况下,由于没有设置测试标志,仿真测试可以直接在本地进行。仿真服务器应该具备可安装插件的结构以便需要时随时添加其他服务。每一个插件的代码都应该能够接受符合标准格式的请求和产生反馈。这种反馈可以是静态的,也可以依测试数据而定。这种采用测试数据的方法既可以使反馈多样化,又可以进行更多的边界和边缘测试。

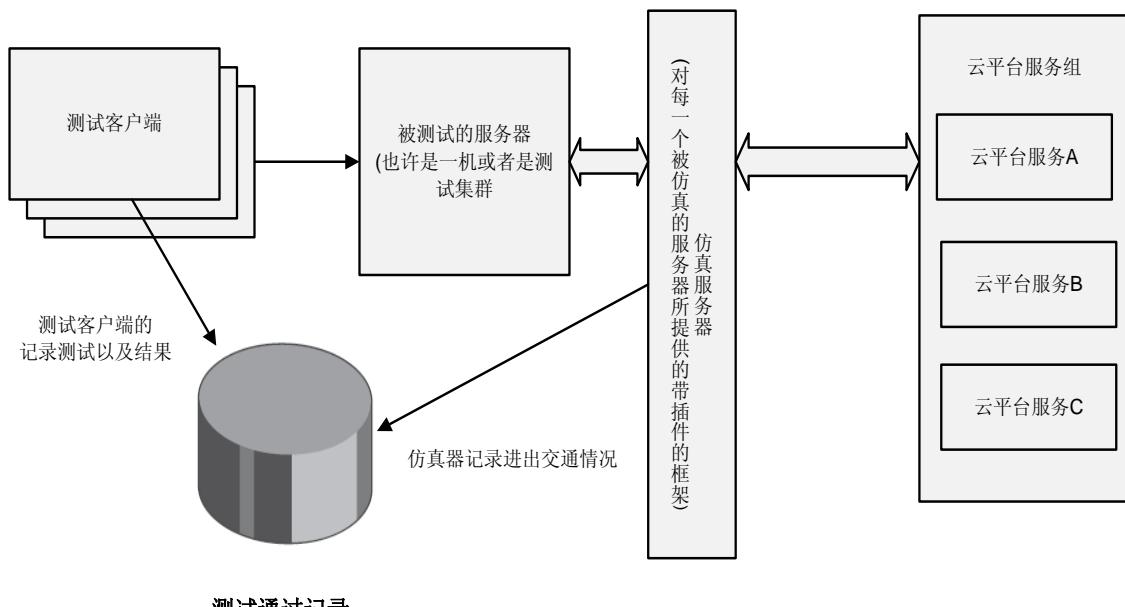


图 14-8 为多个服务器测试用的具有记录功能和安装插件的仿真服务器的框架结构

将测试用例和测试结果记录在案是许多自动化测试的共同特点。好的仿真服务器同样会把进进出出的所有交通情况都记录下来。这样一来,如果一个测试通过了,我们就可以查看发送的数据以及从服务器返回的数据是否正确。我曾经碰到这样一个缺陷,SQL 存储过程(Stored Procedure)两次调用同一个远程服务,但是在数据库里却只存了一行数据。如果没有仿真器将结果记录在案,我们就不会发现这个缺陷。

还有一些更好的仿真方案,它们在设置测试环境允许其调用仿真器的同时,还设置了标志让请求直接通过。这样一来,仿真器仍然记录了所有的交通情况,同时所有的请求会直接进入真正的服务中。

仿真是负载测试的理想方法。由于在仿真器里采用了日志和简化的逻辑，负载测试的测试范围变得更广，而不是仅仅局限于对某一个平台服务的实例的测试。此外，不管实际负载如何，仿真器还可以模仿不同的延迟时间。

直接测试运营服务

对许多运营工程师和产品团队来说，允许另一个服务在实时网站上测试它的新代码是一个荒唐可笑的主意。谁愿意让那些一肚子馊主意的测试人员对程序做些可怕的事情来搞垮你的实时服务呢？大多数运营工程师一想到让测试人员对实时产品胡敲乱碰就会不寒而栗。

尽管如此，通过几年不懈的努力，我已经发掘出了几个论据以帮助大家认识到直接对运营产品进行测试是一件好事，并且不管你是否喜欢这种做法其实已经在进行。

我用的第一个例子是要大家创建一个 WLID 帐户，这个帐户名以单词 *test* 或者 *test account* 开头。你会发现你必须要在这单词之后试过好多数字和字母的组合才能发现一个可用的名字。这就告诉我创建这些的帐户人在进行某种测试，而 WLID 是没有办法制止这种测试的。

所有的服务器都在某种程度上进行运营交易监控，出错时好给 SOC (Service Operation Center) 发出警告。系统每一次升级之后，运营工程师经常会做一些冒烟测试（Smoke Test）以确保所有的主要功能仍然正常运行，这些测试跟那些在一台机器或其他测试环境里进行的测试是一样的。这就说明了运营工程师其实已对运营服务进行了测试。

黑客们一直在不停地寻找运营服务（production services）的安全漏洞（vulnerabilities）。如果一个服务的鲁棒性很好，可以经受黑客们的攻击，它就应该经得起自动化测试的考验。

我要提的最后一个例子是混搭。许多混搭是由分成小组的开发人员采用公开 API 开发出来的。有的甚至采用屏显刮取（screen scrape）技术来建立自己的应用软件。这里关键的一点是，它们必须在互联网上针对运营环境来开发他们的混搭技术。在这种情况下，他们不仅在对运营环境进行测试，而且还直接在运营环境上进行开发。

提示：屏显刮取指的是从全屏显示输出一对混搭来说就是网页—忽略图片等二进制信息而从中提取和分离数据的技术。它并非是最优的方法，因为全屏显示很少有定界符，因而很难从中分离数据。

一旦我们说服了一个团队可以对运营服务进行测试，我们就必须制定测试的指导方针。一些简单的规则，像不能对运营服务进行负载实验，不能直接访问后端服务等，是最重要的。在结构设计上，我们还面临着另一种挑战：把从真正用户来的交通同测试分开。由于许多服务都是有广告赞助的，广告商们是不会愿意为自动测试所产生的广告效应和点击而付钱的。

吃狗食（Production Dogfood）和更大的铁元素（Big Iron）

在本书的第十一章里，我们第一次提到吃狗食的概念，吃狗食仍然是我们测试 S+S 的主要组成部分。对服务吃狗食，指的是用与运营服务相同的硬件设备来设置一个完整的运营服务实例。狗食集群（dogfood clusters）上的用户，有时也叫 Beta 集群，通常都是微软公司的员工、员工家属、合伙人，偶尔也有技术评估者。Beta 集群上的用户也知道在这样的环境里，会有功能不工作的时候。这样对用户的好处是在服务被更广泛地应用之前，用户可以捷足先

登地使用新服务。

狗食集群比正式的运营服务要经历更频繁的版本升级，狗食集群上的用户可以比其他任何人都更早利用新的代码。因而开发团队也期望他们在吃狗食的过程中对产品质量提供反馈。

狗食集群的其他用途还包括运行全套的自动化测试以帮助抓到回归缺陷和做可用性（Usability）测试。通常为本服务发行所进行的公众演示也是在狗食集群上运行的。

运营数据太诱人了但是也充满了风险

微软的 Office Online 有一个资料库存储了成千上万的用户给我们用的文件。这些文件都是现实世界的真正文件，里面有各种各样的宏（Macros）和 VBA 代码以及希奇古怪的用户化格式。我们就用这个资料库来测试软件对旧版本的后向兼容性和用户端对服务器的调用次数（往返）。往返调用将降低服务的性能。

对于那些状态服务来说，总是能得到大量的现实世界里的数据。所以必须在隐私权声明中非常清楚地声明这些数据会不会被用于运营环境之外。如果数据中包含个人鉴别资料信息（personally identifiable information-PII）的话，这一点将尤其重要。

将运营数据应用于测试中将是一笔巨大的财富。我们发现随着时间的推移，有些数据元素的应用被扩展了，而有些元素则被遗忘因而变成了遗留数据，因而运营数据结构也在不停地变化着。

不管运营数据包含个人鉴别资料信息与否，在将其用于测试之前，必须进行净化。我曾经看到过有些团队忘记清理来自运营数据的真正网站的信息而不小心造成对运营网站的狂轰烂炸。因此，应用运营数据的第一条规则就是要净化数据。

提示：数据遮罩（Data Masking），在微软通常指的是数据净化（data sanitization），就是把数据库表里的数据加以遮蔽以确保敏感的运营数据不被意外地泄露出去。清理算法被用来产生可以取代原数据的随机但又具备同样功能的替代值。

应用运营数据的第二个关键就是要做分析。把数据拿来，净化处理后再用于测试，这种做法在很多团队里都很常见。通过分析数据，我们可以寻求边界条件。如果你想知道最长的城市名，那就对一大堆个性化数据加以分析，然后去找到它。这对于不需要运营数据也可以进行运营测试，是非常有价值的信息。

服务的性能测试度量（test metrics）

在本书里，我们已经讨论过许多用来驾驭微软产品质量的度量，所有这些都适用于对软件加服务（S+S）的测试。幸运的是，本章还要讨论一些新的度量来衡量和跟踪 S+S 的测试，如表 14-3 所示。

这里的大多数度量来自于设计最佳实践方案。最佳实践方案的好处在于这通常意味着我们需要去测试以证实它是最棒的。

表 14-3 基于网络浏览器的服务器的测试度量

测试度量名称	定义
--------	----

网页加载时间 1 (Page Load Time 1) (PLT1)	用来衡量从用户第一次发出请求到最后一位数据加载到新的网页中，浏览器所花的时间。所谓的新网页，指的是任何一个此浏览器未曾访问过的网页并且这个网页的内容未被缓存过。
网页加载时间 2 (Page Load Time 2) (PLT2)	用来衡量用户第一次访问过后，每次访问该网页所花的加载时间。它应该比 PLT1 快，因为该网页的内容应该已经被缓存过。
网页重量 (Page Weight)	指的是网页含多少字节。通常含字节多的网页要比重量轻一些的网页加载的要慢一些。
可压缩性 (Compressibility)	衡量文件和图片的可压缩性。
失效时间设置 (Expiration date set)	测试验证相对静态的内容有一个迟于今天的失效期。
往返次数分析 (Round Trip Analysis)	评估任一请求的往返次数，并找出减少往返次数的方法。

当然还有许多其他的度量是测试服务所特有的或者是某一个特殊类型的服务所特有的。比如，搜索领域中一个很重要的衡量指标是相关度，相关度是用来衡量满足搜索条件的网页内容在多大程度上满足用户的需求。

本节中，我列出了在性能方面的可能的度量。这些度量是所有在线服务所共有的。我也挑选了一些质量好的免费测试工具来测试这些度量。

- 本书发表时在 MSDN (<http://msdn.microsoft.com>) 上将会得到的 Visual Round Trip Analyzer (VRTA)
- 在 <http://www.fiddler2.com> 上可得到的 Fiddler，MSDN 上已经登载了几篇介绍它的文章

网页加载时间 1 (Page Load Time 1) 和网页加载时间 2 (Page Load Time 2)

网页加载时间 1 (PLT1) 和网页加载时间 2 (PLT2) 是衡量用户满意度的两个关键指标。研究表明，如果一个网页加载时间超过两三秒钟，它的点击率就会比具有类似内容但加载速度快的网页低。一些搜索引擎甚至用 PLT 来衡量搜索结果的权重。如果一个网页加载速度较慢，即便它的内容和另一个加载较快的网页一样好，它的排名在搜索返回的网页清单中也会很低。

在微软内部，我们非常注重测量我们自己的服务器的 PLT 值以及竞争对手的 PLT 值。我们不仅分析 PLT1 和 PLT2，而且也根据国家来分析，如图 14-9 所示。基于国家的 PLT 值经常会受到进入这个国家的网络带宽和数据到达用户的传送距离的影响。网络工程师的口头禅是他们受到光速的限制。

在某些方面，这有点像观看有线台的广播，现场记者在世界的某个地方，新闻主播向他们提问问题。当新闻主播向他们提问时，现场记者不停地点头。等新闻主播提问完毕，我们看到记者在开始回答问题之前，有那么一秒钟还在继续点头。同样地，数据在从服务器传输到

浏览器时受到光速的限制。

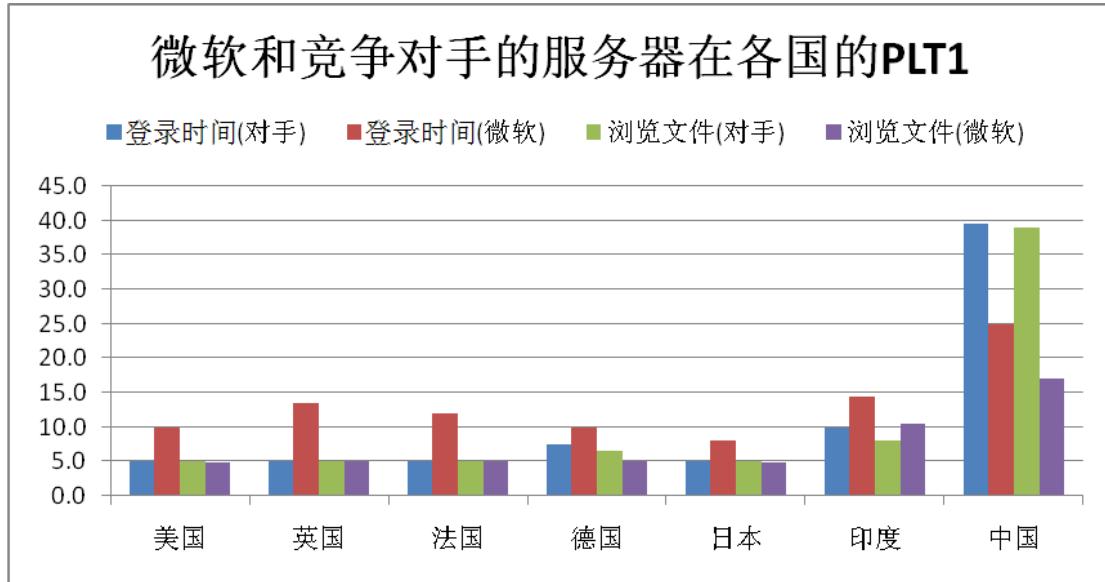


图 14-9 对微软和竞争对手的两种用户交易的分析—登录和浏览文件。除了中国，大多数情况下，微软比较慢。

网络仅仅是影响 PLT 的一个因素。若想最大程度地提高 PLT，可以采用各种优化技术。本节后面我们将要提及的度量被我们用来优化代码质量和网页内容以优化网页加载时间。

网页重量 (Page Weight)

简单地说，所谓网页重量，指的是网页有多少字节。由于网页内容通常都是动态的，由许多元素组成，比如脚本文件 (script files)，图片，甚至 ActiveX 控件，所以，大多数网页重量的测试是在实验室场景里进行的，变化的因素都被固定了。

如果网页重量值很高，测试员就会创建一个缺陷纪录。开发人员拿到这个缺陷后就会去想办法减少总的字节数。最大网页重量值的目标视产品和访问频率而不同。

可压缩性

可压缩性是用来分析一个文件，比如 XML、HTML、图片文件、级联样式表文件 (CSS) 或者 JavaScript (JS) 文件等，在多大程度上可以被压缩。很多发出的 JavaScript (JS) 文件含有空格和注释，虽然不会影响脚本文件的运行，但却会影响文件大小和网页加载时间。VRTA3 是用来分析可压缩性的一个相当不错的工具。

一个网页的许多元素，比如 JavaScript 或者级联样式表 (CSS)，可以被压缩到原大小的四倍甚至更多。这样一来，就可以节省需要在网络服务器和浏览器之间所要传递的数据量。在用户计算机上解压缩是可忽略的。去除像空格和注释这样的空白部分可使网页大小再减少 20%。VRTA3 对文件中的所有元素进行分析，然后确定可压缩的最佳候选元素以及可压缩多少。

表 14-4 列举了从一个压缩报告来的数据。如果可压缩值是 1.0，表明此文件已经被压缩至最佳值。如果可压缩值是 4.1，表明此文件可以被压缩至现在的大小的 4.1 倍。

表 14-4 微软网站的几个 JavaScript 脚本的可压缩性分析

URI	长度	可压缩性	如果被压缩了	时间
http://www.abcxyz123.com/	52, 771	4.1	12, 871	2.766
http://abcxyz123.move.com/fah/hp.js	16, 981	3.2	5, 307	1.35
http://abcxyz123.move.com/fah/commans.js	163, 201	3.1	52, 645	5.447
http://abcxyz123.move.com/fah/tracking.js	15, 851	3.0	5, 284	1.292
http://abcxyz123.move.com/cbrdc/org.css	13, 657	2.7	5, 058	1.207
http://abcxyz123.move.com/cbrdc/org.js	2, 297	2.4	957	1.176
http://static.move.com/abcxyz123/js/nc/s.js	24, 407	2.2	11, 094	1.145
	289, 165	3.1	93, 216	

失效日期

在网页设计中一个易犯的错误是，没有对相对静态的内容设置最大失效日期。这对于减少 PLT2（用户第二次访问同一网页）是非常重要的。如果该文件没有设置缓存，浏览器就会向服务器请求更新文件。即使此文件不曾改变过并且也没有重新下载到浏览器，但是时间还是浪费在向服务器发出请求上了。

Fiddler 是微软的 Eric Lawrence 开发的一个工具，在 <http://www.fiddler2.com> 可以找到它。这个工具在微软内部经常被测试工程师们用于安全测试。它同时也有一些性能测试的功能。

在性能测试的详细结果标签里，Fiddler 会显示该网页所下载的所有文件以及文件的缓存设置。这样一来，就很容易发现可以被缓存的文件。这个工具还可以让用户看到不同内容所占的网页加载时间的百分比。

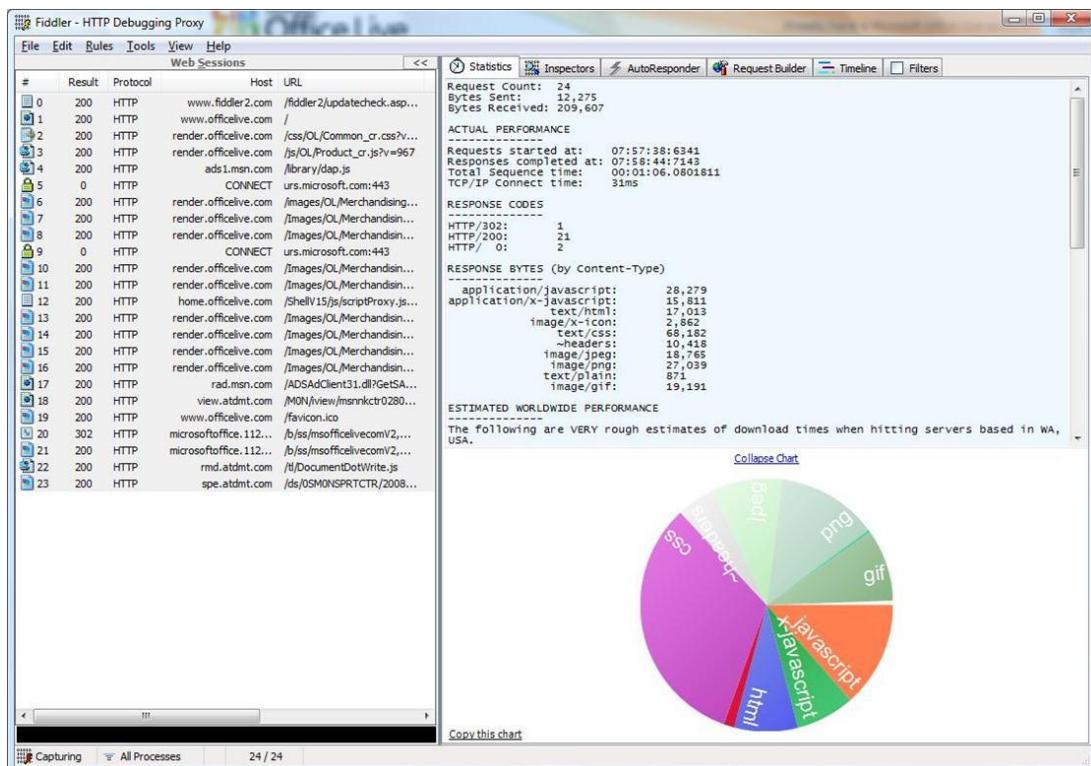


图 14-10 Fiddler 加载 www.officelive.com

往返程分析

网页加载时间的最大影响因素就是网络随着从来源到用户的距离的增加，网络对网页加载时间的影响也随着加大。尽管减少网页重量会减少数据传输时间，但是真正最影响 PLT 的还是往返传输所花的时间。

VRTA3 和 Fiddler 可以帮助对此做分析，并且 VRTA 还有非常棒的图形来帮助分析。

用 VRTA 发现 Internet Explorer 的缺陷

VRTA 的设计可以帮助工程师对网页下载视觉化。在过去的四年中，我在微软一直用这个工具。这个工具把网页下载用视觉化的方法呈现在工程师面前，工程师就可以看到什么东西被依次顺序下载，这样一来，有助于提高各服务器的网页加载时间。

这个工具还帮助发现了浏览器的一些非常难于分析的问题。在 Microsoft Internet Explorer 7 中我们发现的一个问题是 JavaScript 的阻塞行为，由于 JavaScript 的阻塞行为，同时加载的文件数量就会受到限制，并行的 TCP 口也被限制为两个。VRTA 也被积极地用于测试微软新的 Internet Explorer 8 浏览器。但是截止到写本书为止，Internet Explorer 8 浏览器还没有发行，因而我还不能与你分享它的缺陷。

—*Jim Pierson, 性能架构师, MSN 和 Windows Live*

微软的在线办公系统（Microsoft Office Live）团队设置了每一个网页的往返（未被缓存）时间的最大值。程序开发工程师把代码在源码库存储登记时会运行一组性能测试，任何超过往返时间阈值的网页就会被标记出来。

另一些关于软件加服务的重要思想

在这一章节，我想继续和大家探讨一下如何测试软件加服务，虽然这些观点不太适合在测试技术章节里讨论，但是每一位从事软件加服务的测试工程师都应该率先执行并大力提倡。

持续质量提高计划

在第 3 章“工程生命周期”中，我们讨论了里程碑 Q（也称之为 MQ 或 M0）的概念，对于规模较大的项目而言，此里程碑通常是指做好上一个版本的扫尾工作，着手下一个重要的版本的准备工作，团队致力于改进基础设施，使下一代的软件开发和推出更快捷顺利，软件开发工程师致力于研究新技术，开发新原型。而在软件加服务的世界里，在今后的几年内，团队可能每月或每季都要推出新版本，所以通常不设 MQ 做一时的改进，而是致力于持续的改进。

我们所有的运营服务都是数据驱动的，大多数使用与六西格马（Six Sigma）非常相似的过程，我们称之为服务质量（QoS-Quality of Service），来推动持续的质量改进。此 QoS 不应与计算机网络中授予某些应用程序优先网络访问权混为一谈，我们所指的 QoS 是寻觅有助于提高客户满意度的独到的见解。

一个成功的 QoS 计划必须从三大领域摄取数据，我们也可以参考其他领域的数据，但是这些数据常常会导致我们偏离提高客户满意度这个目标。这三大领域是客户意见、产品质量以及运营质量，如图 14-11 所示：

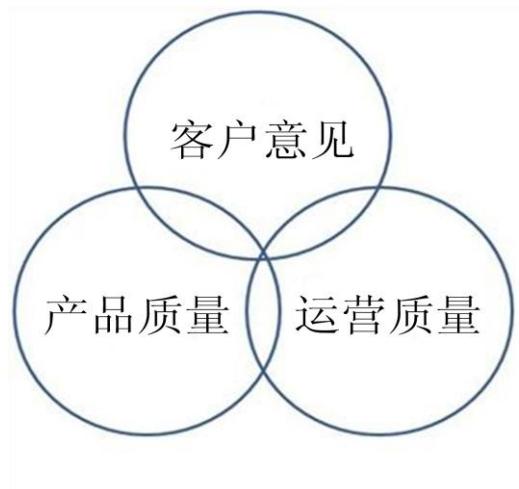


图 14-11 QoS 数据的三大主要来源

客户的意见可以从多个渠道来收集，最常见的做法是对客户满意度进行直接调查，许多服务团队已不沿用此策，而是用净推荐值（Net Promoter Score）。另一种获得客户对产品或服务的反馈的途径是对博客和 Twitter 的数据进行挖掘。

净推荐值是一种管理工具，可以用来衡量客户的忠诚度，能够代替传统的对客户满意度的调查方法。

对于那些有客户支持的服务，客户热线中心的数据是了解客户意见的至关重要的渠道。无论是通过电话帮助还是在线帮助，客户热线中心对用户的要求都加以分门别类。通过挖掘这些数据，服务团队可以找出客户的热点问题，并优先解决这些问题以提高客户的满意度，与此同时，团队也考虑如何提高产品的质量来降低支持费用。

产品质量的主要衡量度是产品的缺陷和性能。在许多情况下，我们知道这些缺陷的存在，但是哪些缺陷需要首先修正却往往难以确定。提高产品质量的一个主要方面是了解竞争对手，不断测量和改善页面加载时间。今天，产品的良好性能是提高客户满意度的关键，但将来却不一定如此。

来自运营质量领域的数据可以用来提高内部效率，但在 QoS 方案中，这些数据用于识别潜在的不满（Dissatisfaction-DSat）因素，我经常和运营工程师开玩笑说，对他们而言，提高客户满意度的机会甚少，降低客户满意度的机会却很多，这个事实说明运营这个角色并不轻松有趣，而是一个具有挑战性的角色。下面是一些主要运营指标以及我们如何用这些指标来改进 QoS：

- 对运营中断错误做根本原因分析（Root Cause Analysis—RCA），能够发现造成中断的不是流程问题，就是潜在的错误或是体系结构的缺陷。
- 不能在短时间内检测到运营问题，能够表示监测系统有漏洞。

- 不能在短时间内解决问题，能够表示日志记录不足或是诊断工具不够。
- 问题券通过第一级或自愈系统解决的百分比，可以显示对大量的简单的运营问题的处理效率。问题如能在第一级或自愈系统得到解决，就能加速总体的解决问题的时间。
- 误报错误的百分比能够表示监测系统有错误或是警报阈值有错误。
- 对问题券加以分类，能够很有效地找出服务可改进之处。这非常类似于客户热线中心使用的编码系统，不同之处是问题券是从运营角度来看问题，而不是从客户角度来看问题。任何具有高计数的领域都意味着服务大有改进之处或自愈自动化。

综合分析这三大领域的数据，团队能将问题按轻重缓急排序解决，最大限度地提高客户的满意度。下面这个例子就能说明在多层次的服务中这个方法是如何行之有效的。6月23日晚间，客户热线中心接到了大量的用户抱怨电话，用户无法处理 PayPal 付款，当客户热线中心接到相当数量的抱怨电话后，就通报了实地运营团队，经过调查，运营团队发现 PayPal 更新了服务证书 (certificate)，因此要求与其联网的服务运用新的服务证书，服务证书实质上是一种密钥服务用于辨认可信赖的伙伴。客户热线中心、运营团队和产品工程团队对这一事件进行了分析，找出了服务监测系统的漏洞，决定实现一个 PayPal 监测方案和警报系统，当证书将要过期时，警报系统会对运营团队发出警报，这样就能防止服务的中断。即使是同样的事件再次发生，有了监测系统，就能大大缩短发现问题和解决问题的时间。

此方法行之有效的关键就是要召集所有的关键人物，分析所有的数据，齐心协力找出问题的根源和最佳的改进方案。

每周只需运行一次，客户不会介意的...哎呀！

早在多年前我们推出新版本的计费平台 (billing platform) 时，我就对服务质量 (QoS) 产生了巨大的热情。下面讲述的是一个很尴尬的故事，不过这件事发生在很久以前，为了保护那些涉及此事的无辜者，我就避免用真名。

新的计费平台是一个内部服务，它的作用是让我们跟踪所有参加订阅服务的用户，并使我们能够通过信用卡公司正确向用户收费。这个计费系统会在夜间做收费的批处理，并非系统设计如此，而是我们历来都这么做。

系统推出后两个星期，我们聚集在战争室 (war room) 会诊缺陷，稍后，运营人员也加入了我们，他们描述了在新系统上看到的问题，Zach 告知我们夜间收费的批处理作业大约要五天才能完成，到目前为止才成功地完成了两次批处理作业。

在座的都大为震惊，我们知道信用卡号码加密会增加一些开销，但也不至于如此，修正这个错误需要重写批处理作业，这无疑会使下一个主要版本的推出推迟几个星期。

当时不知是谁说，“我们为何不每星期只向客户收费一次，比如他们原来是要六月一号付费的，我们就等到六月六号收费，谁会在意呢？我知道这么做会使微软损失一点利息，但至少能使客户满意。”

这果真像一个绝妙的主意。我们离下一个主要版本的推出只差九个月，批处理作业可以等到那时候再重写。我们当时决定在服务质量记分卡上定下目标，批处理作业必须在 168 小时内完成，因为超过七天确实难以接受。

两个月后，我们聚集在一起进行服务质量月终总结，到会的有软件开发工程师、软件测试工程师、项目经理和运营人员，客户热线中心通过电话扬声器参加会议，他们总是在电话上显得更自在。每个团队开始依次谈论记分卡所负责的部分。

产品团队强调指出他们本月修正了 15 个缺陷，注册页的页面加载时间改善了 200%，大家一致认为这样的改进肯定会对客户满意度带来正面的影响。

接下来轮到运营团队发言，Zach 谈论了系统的可用性数据和问题券的数量，当他谈到最后一个数据时，也就是收费批处理作业的完成时间，说道：“收费批处理作业的运行时间越来越长，但是还没有超过一百三十小时，根据目前客户加入系统的速度，大约一年后才会超过一百三十小时。”大家听了后都很高兴，因为这给了我们足够的时间推出下一个主要版本。

随后是客户热线中心发言，他们首先谈论了客户来电的数量和问题得到解决的时间，当谈到来电分类时，他们提到了一个新的客户抱怨，临时命名为“你们使我的支票跳票”，客户打电话埋怨我们透支其银行账户，客户热线中心对此有点不知所措。

Chris 说：“这怎么可能，我们只是从信用卡收费，不可能导致支票跳票。”

Bharat 向前挪动了一下，提高了嗓音，以便在电话上人能听清楚，说道：“我们有可能影响到银行账户，当我登录我的测试账户时，我用的就是银行提款卡的号码。”

Bharat 说得没错，人们逐渐开始用银行提款卡来取代信用卡，但这仍然不能解释为什么客户埋怨我们透支他们的银行账户。

Brett 问道：“客户是不是说我们多收他们的钱？”

“不是的，”电话里的声音说，“客户说我们从他们的账户里取钱，不是导致支票跳票，就是当我们真正收费时，账户里已经没有资金了。”

在座的大多数人开始恍然大悟，Ben 首先说：“就是那个收费批处理作业导致的。”是的，许多客户已习惯于我们在每月特定的日子收费，之后，他们可以随意花费剩余的资金。我们这才认识到修正批处理作业刻不容缓。

如果没有服务质量这个过程，我们就不会将所有的数据和所有的关键人物聚集在一起，也无法认识到不好的设计和错误的决定会怎样引起客户的不满。

我们对批作业进行了重写，运用了大量的并行处理以及使用专用的设备来运行解密过程，几个星期后，新的批作业投入了运行，新的批作业在几个小时内就完成了收费批处理。我们也将服务质量记分卡上的目标改为“连续多天收费批处理作业都在五小时内完成”。

不久后我就离开了这个团队，我最后听说他们已经正常运营一千多天，快要接近两千天了。

我见过的被忽略的常见缺陷

我在微软有一个视频博客系列名为“战争故事—运营中的缺陷”，此系列旨在介绍那些在测试中被遗漏的缺陷。这些并不都是无足轻重的缺陷，而是造成运营中断的严重错误。

有些测试人员不太愿意谈论漏掉的缺陷，似乎这是由于他们的失职所造成的，在此我要指出缺陷并不源于测试人员，而是源于开发人员所写的代码或者是项目经理的错误设计。测试人员找出了成千上万个缺陷，最终有一个缺陷从所有人的身边溜掉了。我认为测试人员提高自己的最佳途径之一，就是解剖分析那些被遗漏了的缺陷。

旧版本客户端可以导致新的服务彻底瘫痪。当一个服务推出有相当一段时间后，大概总有十几个或更多的老客户端仍然在使用这个服务，例如至今仍然有人使用九十年代中期推出的 MSN 客户端版本。我曾多次见过的一个缺陷就是服务升级导致了旧客户端的行为出现变化，这些变化通常出自那些以前从未被运行过的重试逻辑，比如因为客户端不能识别服务返回的新的数据结构，或者是服务器的响应太慢触发了客户端发出新的请求，由于客户端没有设置重试次数的阈值，客户端发出请求的数目持续增加直到运营中的服务彻底瘫痪，犹如遭到了拒绝服务（denial of service-DoS）的攻击。

我喜欢提的另一个错误是“禁止服务器级联”。这种情况是指建立于平台服务之上的服务，会受到其底层组件的可靠性的影响。高层的服务设有服务器禁止列表，当某个平台组件中的服务器未能及时响应时，就被列入此表，这能使高层的服务有机会对平台云中的另一个服务器重新发出查询，然而当平台服务超负荷时，服务器总是不能在预期的时间内响应，查询不断被重新发到另一台机器上，于是服务器被逐个列入禁止列表，直到所剩的服务器寥寥无几，导致系统停滞，所有高层的服务失败。

我想举的最后一个例子是海啸或浪潮。我第一次谈到海啸效果是在 1995 年 MSN 身份验证服务中断后。那时，拨号连接身份验证服务中断了几个小时，中断的时间虽然不长，但时间碰巧是东海岸工作日将要结束时，虽然用户无法登录，但他们仍然在脱机模式下给朋友写电子邮件，同时客户端在后台不断地尝试连接，一旦服务恢复到联机状态时，所有的用户都同时连接上了，于是无数的电子邮件洪流般地涌向互联网，很多电子邮件被送往美国在线公司（AOL），几个小时后，我们的电子邮件服务被浪潮般地从 AOL 返回的电子邮件击垮了。

这种海啸效应在当今的互联网上仍然存在，我们最近就经历了一次，当时，连接北美和日本的洲际网络中断了几个小时，当网络恢复时，拥有大量用户的 MSN Messenger 服务收到了大量的身份验证请求，堵塞了处理请求的 WLID 服务器，这个巨大的冲击使 WLID 服务几个小时不能缓过劲来。

确实，大多数严重的错误是设计缺陷而不是简单的编码错误，但无论是设计缺陷还是编码错误，运营中的缺陷是前事之鉴，后事之师，对这些缺陷进行分析解剖追根究底是团队提高自己的很好的方法。

本章小结

服务是微软下的一个很大的赌注，自九十年代初，微软就一直致力于为互联网开发产品和服务。比尔·盖茨的备忘录“互联网浪潮”和雷·奥齐（Ray Ozzie）的备忘录“互联网服务巨变”都重申了互联网和服务对微软长期成功的重要性。微软每年投资数十亿美元资助服务的开发，同时与业界紧密合作寻找降低电能消耗和提高资源利用率的方法。

用于台式计算机和服务器的传统的盒装软件产品与服务有许多相同之处，这些共性使我们得以使用已成熟的测试技术对服务进行测试。

但是，盒装软件产品与服务也有许多不同之处，这些差异需要微软的测试工程师开发出许多新的测试技术。完全自动化的安装部署和一机测试环境是保证测试得以顺利进行的关键。测试集群使用测试标志、仿真或服务集成测试来查找每一笔服务交易中的缺陷。当测试环境越来越接近于实际运营环境时，我们可以开始对新的平台服务在运营环境中进行测试，甚至将来自于实际运营的数据净化后用于测试。

用于衡量传统软件的所有度量都适用于衡量服务以及软件加服务，但服务需要特别注意页面加载时间，客户不喜欢等待网页加载，对此，有许多简单的改进措施，例如减轻页面的负荷，对文件进行缓存以及减少往返行程，我们应该在测试期间不断衡量这些指标以改善页面加载时间。

要切记的是，绝不能认为服务投入运营后就万事大吉了，从多种角度看，这才是最终测试的开始。测试环境总是有别于实际运营环境，有些缺陷难免会被遗漏。关注实际运营也意味着要实施一个可靠的根据客户的意见、产品质量以及运营质量对服务不断改进的进程。高质量服务也意味着不断地从错误中学习以及从遗漏的缺陷中吸取教训。

第四部分 关于未来

第 15 章 今天解决明天的问题

阿伦·培智

软件测试是一个发展行业，还需要很多年来继续成熟。测试的很多创新和新方法都是对测试团队所遇到的问题作出的被动反应。软件测试工作是在程序员发现他们没法找到他们自己的所有缺陷后才产生的。而很多测试自动化方案的实现，也是在管理人员发现他们不是需要更多的测试工程师，就是需要更有效的方式来进行某些部分的测试后才发生的。

在软件测试中似乎总是又有下一个障碍需要去克服。大多数情况下，测试工程师会一直等到问题大到不得不去解决的时候。为了使测试的艺术，工艺和科学继续进步和扩展，我们需要在问题压到我们不能承受之前，预见到他们。这一章我们讨论很多微软正面临的测试问题，以及我们为了解决它们而选择的方向。

自动失败分析(Automatic Failure Analysis)

如果测试工程师运行了 100 个测试，它们中 98% 的通过了，那么他们可能只需要几分钟的时间来调查剩下的两个失败，然后要么把缺陷记入缺陷跟踪系统，要么纠正测试中的错误。现在，试想一下如果测试工程师有 1000 个不同的测试，需要在 10 种不同的配置下和 5 种不同的语言上运行。在这个“爆炸”到有 50,000 个测试点⁴⁸的测试矩阵中，同样的 98% 通过率会导致有 1000 个失败去调查。随着可用的产品配置数目的不断增多，一个小团队有一百万个测试点已经很常见，一个小小的失败率就会导致足够多的调查而引起“分析瘫痪”——一个测试团队需要花跟运行测试同样多的时间来调查测试失败。

克服分析瘫痪

就象测试自动化是解决测试无穷多的产品配置的一种方法一样，自动失败分析 (AFA - Automatic Failure Analysis) 是一种处理有很多测试失败的解决方案。防止瘫痪最有效的方法是预见到它，也就是说，不要等到你有了多得不能再多的失败需要去调查时，才想到调查对测试团队会造成的影响。测试团队很容易陷在制造自动测试程序和调查失败的无穷循环里（或者，象我有时听到的，制造失败和调查自动测试程序）。处于这种循环里很少能产生测试得好的软件。

在最好情况下，分析成百上千的失败只是要花时间。在其它情况下，会发生更坏的事情。考虑一下下面一个经理和他的员工约翰的对话：

经理： 约翰 - 你调查这些测试失败进展怎么样了？

⁴⁸ 就象在第九章《管理缺陷和测试用例》提到的，测试点是指一个测试在特定环境下的实例。

约翰: 我调查了一些结果，找到了产品的四个缺陷，其它的我还没来得及处理。但我知道其中有几个是因为已知的问题才失败的，那些问题会到下一个版本才会被解决。

经理: Ok，我想让你接着做下一个新功能的自动化测试。

…两个月过去了…

经理: 约翰 - 我们的发布麻烦大了，客户反映有很多严重的问题。

约翰: 是啊…我没有调查的那些失败，跟以前失败得不一样，其中有一个大问题。虽说很不幸，但我没有时间去看每一个失败，而且当时我们也觉得没必要进一步调查…

在这个例子中，约翰因为好几个原因没有完成分析。他忽略过了一些调查，因为他头脑里认为他已经理解了那些失败，而且，他的经理也认为约翰的时间最好花在建立新的自动化测试上。

如果你运行一个测试两次，它两次都失败了，你不能想当然地认为两次失败是一样的。同样，如果你在 5 种不同的配置下运行同样的测试 5 次，这 5 次都失败了，你不知道它们失败的是不是一样，除非你调查所有的 5 次失败。手工地进行分析是很费力的，容易出错，而且让测试工程师不能做他们该做的事--测试软件！

成功的 AFA 需要好几个重要的部分。图 15-1 显示一个 AFA 实现的基本架构。

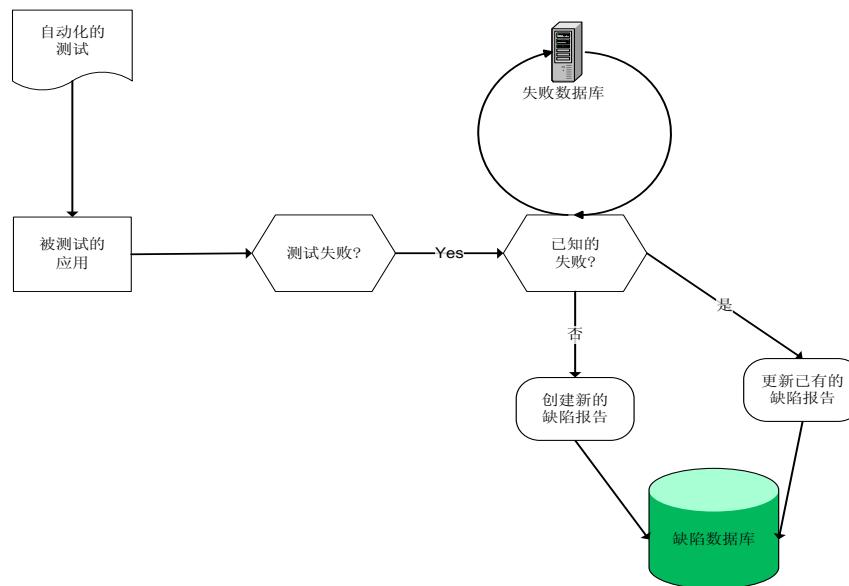


图15-1 失败分析架构

匹配游戏

AFA 系统最重要的部分是失败匹配。如果我在运行一项手动测试(脚本化的或探索性的)时失败了，我记录下导致这个缺陷的跟环境和场景相关的信息。差的自动测试可能只会报告“测试 1234 发生了失败”然后就进行到测试 1235 去了。为了使 AFA 能工作，自动化的测试需要报告导致错误发生的关于环境、场景、和步骤的一致性的信息，好的记录实践（下节会讨论）是一个可靠的失败分析的脊柱，如果测试日志是非结构性的，或者没有包含足够信息，就没有可能进行失败匹配。

在后台，失败数据库包含每一个已知的测试失败的信息。当一个测试失败时，就会和已知失败进行对比，然后系统要么创建一个新的缺陷报告，要么更新已有的缺陷报告。分析引擎的实现很复杂，最起码，它要能对比日志文件、堆栈跟踪或是两者。更全面和可靠的实现会包括复杂的匹配算法，以使系统更灵活和有发展性。举例说，表 15-1 的两个日志，尽管有一些小的差别，应该被匹配为同一个失败，而不是显现为两个截然不同的失败。

日志文件 1

测试用例1234：
系统信息 (MyDevBox)
日期时间。现在
测试输入界限[int foobits (int)]
测试下限[0]
下限测试通过。
测试上限[32768]
期望结果-1
实际结果0
结果： 测试失败。

日志文件 2

测试用例1235：
系统信息 (MyTestBox)
日期时间。以前
测试输入界限[INT FOOBITS (INT)]
测试下限[0]
下限测试通过。
测试上限[32768]
期望-1
实际0
结果： 测试上限失败。

列表15-1 日志文件匹配举例

聪明灵活的失败匹配算法容许在日志信息中有小的变动，以及由于数据驱动变化或测试生成技术而造成的变动。

好的日志记录实践

好的日志记录对分析和匹配失败来说是必不可少的。然而日志常常是测试工程师加到他们自动化中的随机和不稳定的“税务”。尽管是一件很细小的事情，然而高质量和一致的日志实践能区别“垃圾”自动化测试和能运行十年甚至更长时间的测试。表格 15-1 列出了一些写日志文件时要考虑的事项。

日志实践	描述
日志在成功时要简短，失	在实践中，“噪音”测试经常是写得很差的测试。记录在

失败时要极其详细	日志中的每一条信息都应该以诊断最终的测试失败为目标。当测试失败时，测试应当有足够的跟踪信息来诊断失败的原因。
当测试失败时，记下所看到的失败前的成功操作。	知道最后成功操作的状态可以帮助诊断失败路径开始的地方。
日志应该跟踪产品信息	日志应该跟踪记录关于产品的信息，而不是测试的信息。尽管在自动测试中嵌入跟踪语句能够帮助调试，但这些语句不该在测试结果日志中。
跟踪足够的和有帮助的失败上下文	知道更多的关于失败是怎样发生的，可以帮助诊断底层的缺陷。不要记录： 测试失败了 而是记录以下这些： 测试失败了 Win32BoolAPI 参数 Arg1, Arg2, Arg3 返回 0, 期望 1。 或... 测试失败了 Win32BoolAPI 参数 Arg1, Arg2, Arg3 返回 0 并且设置最后的错误为0x57, 期望 1 和 0x0。
避免记录没有必要的信息。	日志不需要列举测试和其测试的软件中执行的每一个动作。记住上面所说的第一条规则，只为失败情形做详细的日志。注意 - 为调试目的，有一个能区分用户级的日志还是有好处的，可以容许最少的或最多的跟踪日志。
如果结果已经被确认或证实了，每个测试点都应该记录这个结果。	把失败聚集在一起的测试往往会掩盖缺陷。如果测试处于“失败并进行”的模式，知道每一个失败是在哪产生的是很重要的。可以用来诊断哪些后来的失败是相关的，而哪些是独立于以前的失败的。
跟随团队的命名标准	标准能帮助确保读日志文件的一致性。所有的对象，测试，和过程名字应该有意义和非滥用的（一件事一个名字）。

表 15-1 日志最佳实践

日志文件剖析

日志文件中哪些信息是有用的呢？表格 15-2 把列表 15-1 的日志文件细分如下：

日志条目	目的
测试用例1234	独特的名字
系统信息 (MyDevBox)	放环境数据的地方
日期时间：现在	上下文数据（和其它测试不可能匹配）
测试输入界限 [int foobits (int)]	指出

测试下限 [0]	它的值是什么，如果崩溃发生，我们将没法在这之后跟踪它的值
测试下限通过	(最后的) 已知的好运行
测试上限 [32768]	记录当前输入值
期望结果 -1	我们想要看到什么？
实际结果 0	我们实际看到什么？(失败状态察看)
结果： 测试失败	测试状态的正式总结

表 15-2 日志文件注释

开始AFA

技术上的欠债把我引向了自动化的失败分析。

我们已有的自动化很难维护而且建造得很差。我们需要花三天时间去分析每周一次的单个配置的测试执行。由于我们的旧自动化体系的开销，我们没有办法处理新的任务。因为追求目光短浅的目标和指示（象100%的自动化，而不顾及自动化的类型或适用性）我们处在了分析瘫痪的状态。

AFA牵涉到我们做测试的基本假设- 我们怎样报告我们的结果和分析那些结果。没有来自上面的声音，每个团队都会很高兴的用自己的行话，而让相邻组不知所云。

AFA能处理我们在写很弱的自动化时累积的技术债务。高质量的AFA要求清理整个过程，也绝不是大多数团队想要找的快速解决方案。我们发现一咬牙花三个月时间实现一个解决方案付清我们的技术债务很困难。如果从头到尾你都做的对，那么你能在几周之内把AFA跑起来。如果你做得差不多，你需要一个月，如果从头到尾你都做错了，会需要好几个月，但这只是因为AFA需要你在开始前要为技术债务付一大笔款子。**Geoff Staneff - 软件测试工程师**

集成 AFA

大型测试自动化要想有非常好的投资回报需要集成自动化的每一个阶段，检入系统，和缺陷跟踪系统。一个成功的方案能够极大的减少手动干涉分析测试结果和失败的需要。AFA 的另一个用途是分析测试失败的走势。例如，分析可能指出在过去六个月内参数验证错误占测试错误的 12%，或者在过去六个月内用户界面的计时问题占测试错误的 38%。这种数据能被有效地用于了解和定位产品的风险区域。

AFA 在行动中：三个短故事

AFA 系统还有很多其它的好处。以下的场景描述了在这种系统中有那些可能。

开发工程师 Bob 刚检入了缺陷号 4321 的纠正。当他检入代码到 SCM 时，缺陷 4321 在缺陷数据库中被自动标为已解决了。失败分析系统发现这个缺陷被标为已解决了，就会自动的把因为这个缺陷而失败的所有测试移到自动队列的前面，这样在测试运行时对这个纠正的验证就会早些执行。

在过去几周，由于缺陷 7734 很多测试用例都失败了。在今天的测试执行中，这个测试又失败了，但是失败分析系统注意到有两个测试跟以前失败的不一样。一个新缺陷被自动建立了。这个缺陷包含跟失败相关的信息，和到缺陷 7734 的链接。最后，对最近代码改动的自动扫描启动了，缺

陷被分派给最有可能造成这个错误发生的开发工程师。

今天早上，测试经理 Jane 看到最后的测试执行有 48 个新的失败。AFA 系统标记了所有 48 个失败的根本原因并且在缺陷报告中记录了这个失败模式。

AFA 使团队能够在新问题出现时专注于处理它们。测试结果分析的风险主要在于由于期望的变化的干扰而丢失重要的变化。在调查一系列的失败时，我们会自然的（普遍的）作出估计说“这个测试点上周失败了，这周的大概跟上周一样”。AFA 系统能够在决定一个失败是否以前出现过时，通过运用分析规则，消除测试执行分析的繁琐，和消除可避免的分析错误。而且，这些规则不会因为对测试的熟悉度或是测试点失败出现的频度而被放宽。

机器虚拟技术

微软的测试团队使用的大型实验室，堆满了桌面计算机和一排排的阵列系统。但机器始终是有限资源，要被用作构建机器，或用来做每晚或每周一次的手工和自动测试执行。微软的 Hyper-V（以前叫做 Windows Server Virtualization）正很快的成为一种替代方法，能让测试团队利用虚拟机器。（虚拟机器 VM 的最简单定义，就是把计算机的实现做成程序，运行在宿主系统（host system）里）。基于 Hypervisor 的虚拟（如 Hyper-V）比宿主式的虚拟（VM 在宿主操作系统中作为程序运行）有更好的性能和安全性。

虚拟的好处

用虚拟机做测试正很快的被人们采用。它对测试工程师的最大好处是方便：维护一台物理机器和一组虚拟机，比维护多台物理机器要简单多了。另一大好处是一个测试工程师可以在一台机器上跑成组的虚拟机。传统上测试工程师可能需要在办公室放很多机器，或者能使用大型实验室。虚拟机提供的另一大好处是节约成本。成本减低是由于使用少一些的机器，和 VM 的并行优势导致已有的硬件得到更好的利用。这些节省对于单个的测试或开发工程师办公室和测试实验室都有好处。

在办公室

测试工程师和开发工程师办公室里经常需要多台机器。比如，他们可能需要同时并在不同的硬件架构或多台机器上做测试。与其在他们的办公室里堆满机器，（在我的测试工程师生涯中，我曾在办公室里有不下十台测试机器），每个办公室可以只要一台 Hyper-V 服务器，在上面运行几个不同的 VM。虚拟技术能让测试工程师和开发工程师在同一时间生成很多不同规格的 VM，配置也相对快捷和简单。这种灵活性让测试工程师和开发工程师不用在办公室增加硬件就能得到更高的测试覆盖率。

测试实验室的节省

测试实验室主管能利用 Hyper-V 的服务器整合应用来更有效的利用已有硬件和最充分的

利用新投资。微软的绝大多数测试组都有大型实验室，装满了用来做自动化测试，压力测试，性能测试和构建的机器。这些实验室是整合的主要对象。测试实验室很少能满载运行。利用虚拟机，实验室管理者能用少得多的机器来完成同样的任务。这能节省可贵的实验室空间和电费。

减少服务器数量还能节省实验室主管的时间。每台实验室的计算机都有时间成本，因为要对其进行安装，上架，和配置。一台机器还有管理开销，例如升级，硬件找错等任务。有些开销总是会有，但用 VM 能极大地减少这些开销。虚拟技术减少了需要这些时间承诺的物理机器的数目。尽管在虚拟机器上很多同样的任务还是必须的，由于能够利用自动化，这些工作会容易一些。一台计算机的物理设置，在有些方面是不可能做成自动的，但是用 Windows Management Infrastructure (WMI) 脚本能够对虚拟机作同样的工作。比如，虚拟网络能让实验室主管用程序动态的修改网络结构，而不是手工的去拔接线。既然 Hyper-V 能在同一台物理机器上运行多种不同的虚拟机，就没有必要在实验室中放很多不同的服务器了。当然，放置代表密传硬件的机器还是有价值的，但虚拟机能很容易的代表简单不同的，如处理器和核的数目，32 位或 64 位，以及内存配置。因为这方面的原因，我们不会推荐用虚拟机测试来完全替代物理机器测试。

测试机器配置的节省

使用虚拟技术节省的不止是金钱，供电，和空间。开发时间是非常有价值的资源，虚拟技术能够减少两大工程时间开支：测试机器的安装时间，和测试恢复时间，从而让开发工程师和测试工程师工作更有效率。

开发工程师和测试工程师都需要一大笔时间来安装机器以测试和验证他们的代码。虚拟方案让用户只用生成测试映像 (test image) 一次，然后部署多次。比如，测试工程师能生成一个虚拟机，然后存放在文件服务器。当要运行测试时，测试工程师只要把 VM 拷贝到宿主服务器上，然后运行测试就行了，而不用花时间安装操作系统和其它软件。测试工程师和管理员经常为此目的生成一整套不同配置的虚拟机。测试工程师和开发工程师然后能选择他们正好想要的虚拟机，而不用手动的安装一台机器。想要在预装了 Office XP 的 Windows Vista 的德文构建上运行测试？只需拷贝一个文件你就能得到用于运行这测试的环境了。

测试机器在测试时经常会进入一种不可恢复的状态。毕竟，大多数测试是为了找到缺陷，而且复杂的应用和系统软件中的缺陷会造成机器失败。虚拟机为这个问题提供了两种不同的解决方法。第一种，也是最简单的，就是 VM 不是物理机器。当 VM 失败时，物理硬件并没有受损害，父分区的数据没有丢失，其它 VM 也没有受影响。灾难性的失败造成的影响被极大的减少了。

虚拟技术的另一个节省时间的好处是能够对系统照快照 (快照 – *Snapshot* 是微软 Hyper-V 用的名词。其它虚拟方法可能会用不同的名字来命名这项特性)。快照是 VM 的一个静态的“冻结的”映像，能够在任何时候照。在照了一个快照之后，VM 继续运行，但照快照时的状态都被保留了。快照能帮助开发工程师和测试工程师很快的从错误总恢复过来。在测试之前照一个快照，测试工程师能快速简便的把 VM 回退到失败之前的一点。他们能够重新运行测试，或进行其他测试，而不用重新生成 VM 或重装它的操作系统。

在我们还没有滥用快照之前要记住几件事情。尽管我们可以生成很大数量的快照，这样每一个可能发生的 缺陷都有一个“重现用例”，这点子看起来完全符合逻辑，但会在

Hyper-V 中造成很差的性能。每一个快照都会增加一层对 VM 的磁盘的间接访问，经过几小时的频繁快照它会变得非常慢。还值得注意的是快照不是便携式的拷贝，而是最初的 VM 和快照之间的一组不同之处—也就是说快照在它所属的 VM 之外就没法用了。

虚拟机测试场景

使用虚拟机在很多不同的测试场景中都有优势。对于一些常用场景的讨论会在下几节中讨论，但使用场景是不受限制的，还有很多其它的测试场景能受益于虚拟机，如 API 测试，安全测试，以及安装和卸载场景等。

每天的构建测试

很多测试工程师都要负责测试各种各样的维护包，升级，和产品的版本。例如，考虑一个可以在 Windows 的多个版本上跑的应用。产品的第一个版本刚发布，测试团队在开始测试第二版本。表格 16-3 列举了这样一个系统的典型的硬件/操作系统矩阵。

	要测试			
	32位	64位	单处理器	双处理器
Windows XP SP2				
Windows Server 2003 SP2				
Windows Vista SP1				
Windows Server 2008 ⁴⁹				

表15-3 宿主操作系统测试矩阵

正常情况下测试工程师会有三个或更多的测试来支持这个矩阵（取决于测试用例的数目）。测试工程师当然可以用一台机器来测试这些变化，测完一个操作系统再测另一个；可是这样会效率很低而且安装和配置测试环境会花大量的时间。安装操作系统和升级，以及相应的重启，都是相当费时间的工作。由于三个或更多的虚拟机能在一台物理机器上运行，虚拟技术能把三台测试机器转换成九台或更多台虚拟测试机器而且把测试机器安装时间减半。利用快照和 Hyper-V 提供的脚本界面等特性这些都成为可能。

在上面的矩阵中，测试工程师一般都会在测试执行过程中，在物理机器上安装八个不同版本的 Windows 操作系统。虚拟化的解决方案能让测试工程师在三台（或更少）物理机器上安装所有的八个版本。一个自动化的测试执行的设置只需一眨眼工夫就完成，而手动的测试覆盖率会上升，因为测试工程师能够集中精力测试产品，而不是花时间设置环境以进行到下一个测试用例。兼容和升级测试也能因使用虚拟机而极大受益。

⁴⁹ 由于 Windows Server 2008 调整多处理器和单处理器之间的内核，为多处理器重装操作系统就没必要了。但对于 Windows Vista 及其之前的操作系统就是必须的。

网络拓扑测试

Hyper-V 能帮助建立复杂的网络拓扑结构，而没有配置一大堆接线和物理交换器的麻烦。图 16-2 and 16-3 展示了机器虚拟能怎样被用来建立复杂的网络环境。图中的整个的网络拓扑结构都能在一台物理计算机中建立。

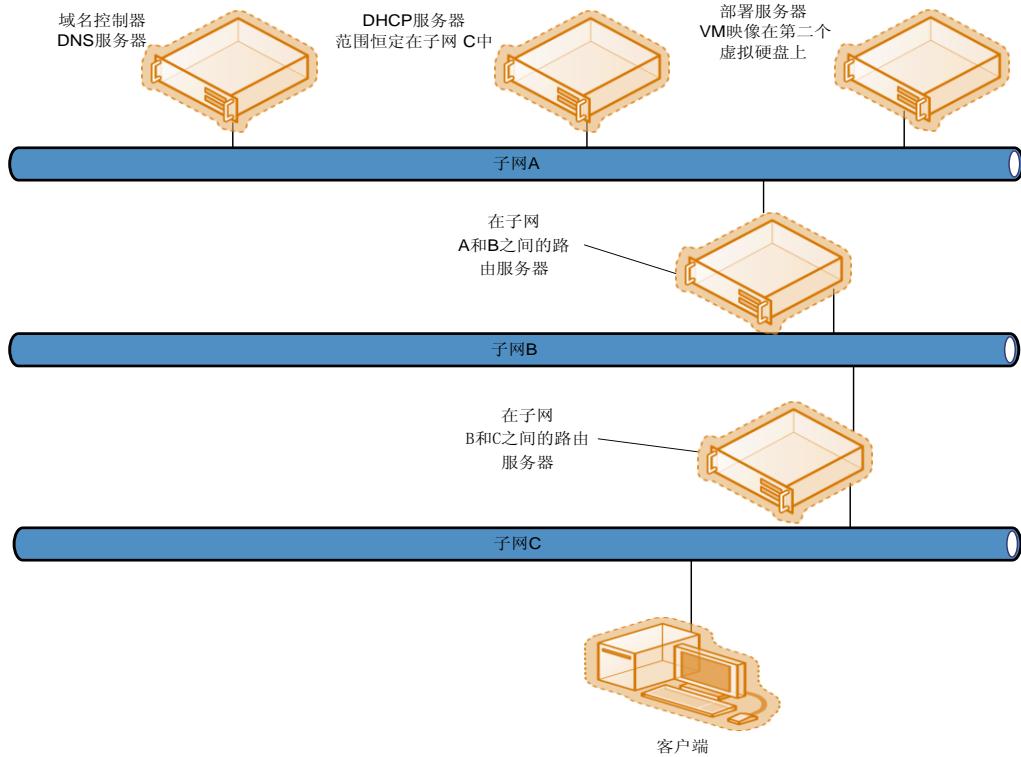


图15-2 虚拟的网络拓扑

在这个场景中，三个子网被建立，并且由两个作路由器的服务器来桥接。子网 A 包含所有典型的人们赖于上网的基础设施。子网 A 还有一个部署服务器能够部署 Windows 映像。在子网 C 的客户端代表一台被期望能从网络启动并安装最新的 Windows Vista 版本的机器。

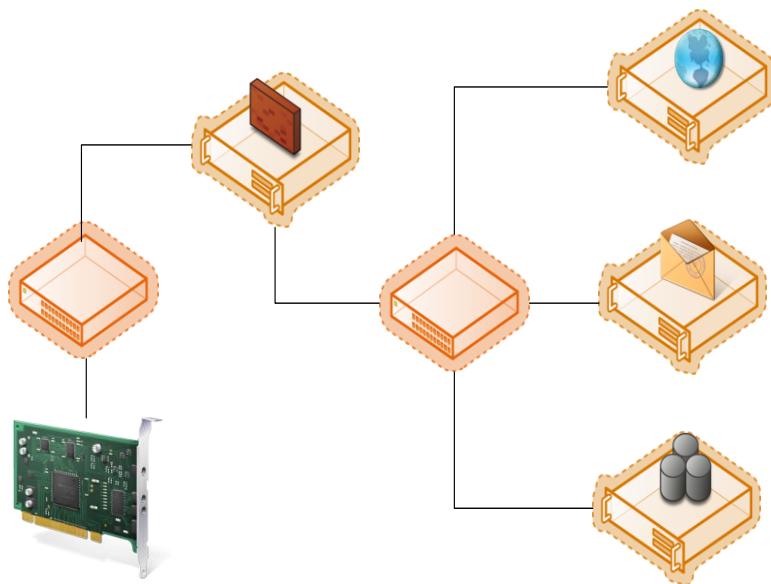


图15-3 防火墙拓扑

在图 15-3 中，三台虚拟服务器在作为防火墙的第四台服务器之后。防火墙 VM 通过一个虚拟交换机与宿主机器的物理网络界面相连，但其它三台服务器没有。它们与防火墙服务器通过第二个虚拟交换机相连，那个交换机与外界网络并没有直接联结。

这个场景的有意思的部分是所有这些虚拟机，交换机，和子网的生成能够通过自动化产生，而且运行在一台物理计算机上。在虚拟技术之前，像这样的实验场得需要好几台物理机器和所有的布线，路由才能建立网络拓扑—做这些事所需的人力就更不用提了！

当测试时发生失败

测试时发现的失败对测试工程师来说是很花时间的障碍。时间经常耗费在等待开发工程师调试问题或对错误给出解答。如果他们需要保留一台机器好几个小时或好几天来等待回应，就会耽搁测试工程师完成更多的测试。

有些失败很不寻常，可能会要好几个小时甚至几天来重新产生。这会花费测试工程师和开发工程师两者的时间，尤其是开发工程师想观察失败发生前机器的状态时。

以下的两个例子是测试组织时经常发生的。利用“导出（export）和导入（import）”以及快照，虚拟技术有能力帮助工程团队提高效率。

导出和导入

在用虚拟机时，测试工程师不需要等待开发工程师调试失败。相反，测试工程师能保存虚拟机并把它导出到一个网络共享。保存虚拟机会使虚拟机的所有状态被保存到磁盘上。导出会把虚拟机的配置打包并保存到指定地点。

开发工程师能在之后在方便的时候导入虚拟机。由于虚拟机在导出之前被暂停了，虚拟机在打开时也处于暂停状态。开发工程师要做的只是让虚拟机恢复运行，然后调试失败，就像坐在一台物理机器前一样。这样一来，相比让测试运行在宿主机器上，测试工程师能更快拿回测试机器的控制权。它也能让开发工程师正确排列调查问题的优先级，和针

对“难于重现的错误”建立和保存多个快照，以便于进一步的调试。

快照(snapshot)

快照能帮助减少重新产生需要运行几小时甚至几天才会发生的缺陷的时间。例如，测试工程师能够编写脚本每小时给虚拟机照一个快照。当脚本运行时，测试会在虚拟机中运行。当失败发生后，测试工程师能够调查并找到失败发生前的快照。然后，测试工程师能够回到那个快照，并从那一点启动虚拟机，在 60 分钟之内遇见那个缺陷。快照使得测试工程师能反复的这样做，而不用等待几小时甚至几天来重新产生缺陷。用这种方法有些非确定性的缺陷可能不会马上重现，但对于过了特定的一段运行时间就出现的缺陷就管用。

快照是拯救者

在测试Hyper-V时，快照成了很重要的特性。我们用快照是因为每当有新的构建可以测试时，我们就不用总是在虚拟机中重新安装子操作系统。我们会在每个虚拟机中安装子操作系统，然后拷贝从构建到构建中不变的工具。当虚拟机准备好之后，我们关掉虚拟机并建立虚拟机的一份快照。在这之后我们才启动虚拟机的备份，安装每天的构建，和开始我们的测试。我们这样做的原因是我想把虚拟机很快的恢复到干净的状态。我们只需重新应用快照并且立即安装应用程序的下一个构建。这样我们日复一日的唯一开销就是安装每日的构建，只要花5分钟，而不是以前需要的几个小时安装时间。

我真希望我们在测试服务包（service pack）的时候有快照就好了。我记得我不得不安装RTM OS，然后安装服务包，然后测试。而且我每天都要做这些。如果我有快照，我就只用安装RTM

OS一次，然后照个快照，然后安装服务包。当我测试完服务包，我只要重新应用快照，就会有一个保证干净的RTM OS来安装维护包的下一个构建。这肯定会节省我好几个小时。

Shawn McFarland - 高级测试工程师

不建议使用的测试场景

尽管虚拟技术有很多优势，在有些情况下不建议使用。如果宿主机器只是用来容纳一两个虚拟机，成本效益就小很多，因为它相对硬件使用并不提供很大的效率提高。不过，在有些情况下快速部署及快照的好处会大于虚拟带来的开销。

由于 Hyper-V 给予操作系统提供它自己的设备驱动，在虚拟机里无法测试设备驱动。同样，需要特定硬件或芯片的测试，将不能访问宿主机器安装的硬件。Hyper-V 的视频驱动是为兼容性而设计，并且为远程桌面使用而优化的。所以，高性能视频和 3D 生成在 VM 上将不能工作。Hyper-V 根据需要动态的为 VM 分配处理器资源，所以底层的电源管理软件在 VM 中就不起作用。

关于虚拟技术，微软 Hyper-V 及技术的资料能在这找到：
<http://www.microsoft.com/virtualization>

代码评审和检验

代码评审是工程过程中不可或缺的一部分。我每完成你们所读的一章的草稿，我会让几个同事和题材专家来评审我所写的，然后再提交给编辑做进一步评审。这是找到文章“缺陷”和数据、示例错误的最佳办法。代码评审给代码提供同样的服务，对早日发现缺陷极其有效。

代码评审是微软每个团队开发过程中的一部分，但也是一个有效性有待改善和提高的领域。

代码评审的类型

代码评审有轻量级的（“你能扫一眼我的代码吗”）直到正式的（团队开会，有设定的角色和目标）。[配对编程](#)，敏捷方法之一，两个开发工程师共用一台工作站是另一种代码评审的方式。不同的方法有不同的有效性，以及牵涉到的参与者的舒服度。微软的很多测试工程师都很积极的参与评审产品代码，很多团队对测试代码和产品代码有着同样的评审要求。

正式评审

最正式的检验类型叫做 *Fagan* 检验（以这个过程的发明人 Michael Fagan 命名）。*Fagan* 检验是团体评审，有着严格的角色和过程。评审员被分派角色如读者，或主持人（代码作者也参加这议程，但不担当其他任何角色）。主持人的工作是确保参加评审议程的每个人准备完全，并安排和执行这个会议。评审员被期望在与会前花相当的时间预审代码，并经常用核查清单及指导大纲来集中精力做他们的评审。*Fagan* 检验需要大量的时间投资，但对找到代码的缺陷非常有效。微软的一个团队用 *Fagan* 检验把测试组和客户找到的代码的缺陷数量从每千行（KLOC）10 个缺陷减少到每 KLOC 不到一个。尽管有找到错误的潜能，阻止团队用 *Fagan* 检验的最大障碍是它们所费的时间（检验率是大概每小时 200 行）-还有就是开发工程师不愿意花 25-30% 的时间在正式检验会议上。因为上述原因，尽管它很有效，*Fagan* 模式的检验在微软没有被广泛应用。

非正式评审

要开发有效的代码评审方案，挑战在于要找到一个正式级别，能在编程阶段既省时又有效的找出重大问题。“站在身后”的评审很快，但通常只能找到小的错误。“email 传送”评审有多个评审员的好处，但结果会因为谁看了 email，他们花多长时间评审，以及他们看变动有多仔细等等而变化。

看起来最佳解决方案是要找到一个既有合作又省时的过程。程序员需要有多个指定角色的同行评审的好处，而没有正式会议的开销。像聪明熊软件（SmartBear software）这样的公司曾做过和发表过有着这些前提想法的案例分析⁵⁰并成功的建立了轻量型的评审过

⁵⁰ Jason Cohen, ed, *Best Kept Secrets of Peer Code Review*, <http://www.smartbearsoftware.com>

程，能跟正式检验的有效性相近。微软的内部实验得到的结果也类似，很多团队都在试验以找到做代码评审的介于正式和有效之间的完美平衡。

核查清单

我认为当人们知道他们要做什么时，大多数人都做得更好这个说法没错。当员工上班时，如果他们有一个要处理的任务列表（不管是他们自己制定的，还是上级为他们制定的），相比较被告知“去做事吧”，他们可能会完成更多。然而，很多代码评审都只是首先要求“请看看我的代码”。有些评审员能够不用任何指导就能评审得不错，但对于大多数来说，一个核查清单或指导大纲会很有帮助。

核查清单会在代码评审时引导评审员找到最容易检测到的缺陷类型，也会在代码评审时找到最关键的缺陷。核查清单的例子可能有：

- 功能核查（正确性）
- 可测性
- 检查错误并正确处理错误
- 资源管理
- 线程安全（同步，重入，计时）
- 简单性/可维护性
- 安全（中断溢出，缓冲溢出，类型不匹配）
- 运行时的性能
- 输入校验

其它类型的核查清单可能会集中在单个的区域如性能或安全，这样多个评审员能够集中评审代码的不同方面。

其他考虑

关于代码评审有趣的是，相对来说很少有团队会直接监测评审的有效性。这看起来很傻：为什么会不知道投资回报就花时间去做？一个间接的有效性的衡量是监测被跟踪的缺陷，也就是，如果在实施了代码评审的政策后，测试队伍和用户找到的缺陷少了，你可以说代码评审起作用了...但是它们到底有多有效呢？如果你一定想知道，你当然得测量所花的努力和得到的效果。要精确测量代码评审，或为之所做的改进过程的投资回报率，需要去测量时间投资，以及用某种方法跟踪在评审时发现的缺陷。

活动

表格 15-4 列举了两个大小和范围相似的假设产品。在产品 A 中，只有测试队伍发现的缺陷或是发布之后（被用户找到）的缺陷才被跟踪。产品 B 还跟踪开发工程师测试过程中和代码评审找到的缺陷。

产品A	产品B
-----	-----

开发工程师测试过程中发现的缺陷	?	开发工程师测试过程中发现的缺陷或返修	150
评审发现的缺陷	?	评审发现的缺陷或返修	100
测试队伍发现的缺陷	175	测试队伍发现的缺陷	90
用户发现的缺陷	25	用户发现的缺陷	10
发现的缺陷总数	200	发现的缺陷总数	350

表15-4 按活动发现的缺陷

产品 A 有 200 个已知的缺陷，产品 B 有 350 个。产品 B 用户发现的缺陷要少一些—但没有足够的数据来说一个产品要比另一个“虫子多”。然而关于产品 B 我们确实知道的是，在那些类型的活动中发现了缺陷。跟计划阶段的数据，或是任何这些任务所花时间的跟踪相结合，我们能开始知道哪些早期检测技术最有效，或是在那些地方需要做改进。

采取行动

除了按活动所找到问题的数目，在代码评审中找到了哪些种类的问题也有帮助。比如说，对用户和测试队伍找到的缺陷作简单轻量的分析，可能会揭示这些缺陷的很大部分应该通过代码评审检测到，说明应该采取行动更新核查清单或执行更严格政策。

此外，按照解决问题所需的返修的类型来分类问题，能够找出在哪需要额外的早期检测技术。表格 15-5 列举了代码评审时找到的一些常见的返修项目示例，和可以用来在代码评审之前检测问题的技术示例。

问题类型	检测/防止技术
重复代码 - 比如，重新实现公用库中已有的代码	告诉开发队伍已有的程序库及其用法；每周进行库的功能讨论及展示报告
设计问题 - 比如，实现的设计不是最优，或是尽可能最快解决编码问题	群体的代码评审能发现这些错误，并且让所有评审参加者都学到好的设计原则
功能问题 - 比如实现有缺陷，或欠缺局部功能（省略错误）	功能缺陷会导致新的指导大纲或技术的实现，被用于开发工程师测试中
拼写错误	在 IDE 实现拼写检查

表15-5 代码评审问题和相关的防止技术

时间在我这一边

对代码评审有效性的精确测量需要精确的知道花在评审上的时间。你可以直接问团队他们都花了多少时间评审代码，但得到的答案，就像你所想的，将会是很不精确。这就是为什么那些想测量代码评审价值的团队会在所有代码评审中都用某种“代码评审工具”。如果评审是在评审工具的框架下进行的，就能更容易记录评审工作所用的时间。当然，要测量花在使用应用程序的时间这样的评审会有些困难。一种精确的方法是监视和应用

程序的交互（敲键、移动鼠标和焦点）来决定评审人是在积极评审还是仅仅打开了代码评审窗口。

跟其它指标放在一起比较，代码评审时间能变成一个有趣的指标，可以回答以下的疑问：

- 我们在代码评审上花的时间跟代码实现比较各占多大比例？
- 我们每评审一千行代码（KLOC）要花多少时间？
- 我们在这一版本和上一版本在代码评审上所花时间的各自的比例？
- 所花时间和所发现的错误的比例（每评审小时找到的问题）？
- 每评审小时找到的问题和测试工程师及客户找到的缺陷的比例？
- 其它...

以上的疑问只是举例而已。对一个特殊的情形要决定正确答案，你要问你自己，你们组代码评审的目的是什么。如果目的是在检入之前现场检查代码，那么任务测量的时间可能就没多大意思，但如果你想提高效果和效率，这当中的疑问就可能帮助你决定是否迈向你的目标了。

更多的评审附属工作

除了要找到需要返工的代码段，代码评审中还有更多事情要做。评审中有一个经常会因事过境迁而丢失但十分重要的部分，这就是对一段代码的讨论和注释。比如说，当我写好了代码，可以被评审时，我不会说“这是我的代码，看一看吧”。相反，我会写一个介绍性的 email，用几句话或一段话描述代码在做什么（如：纠正一个缺陷或加一个功能），而且我可能会描述我的一些实现决策。这些信息会帮助评审人做得更好，但一旦评审结束，这些信息就丢掉了。如果考虑到其后的对话，不管是发生在 email 里、在小组评审、还是面对面的，都丢掉了，那情况就更糟。随着时间的推移，丢掉的信息会造成知识传承的困难和维护等一系列问题。

评审工具能帮助代码评审的另一种方法是跟踪这些附属的数据。代码评审工具可以包含问题、注释、对话以及元数据，并把它们和代码相连系。小组的任何人都可以查看源文件，看变动，和关于任意变动的解释和对话。如果小组的某人需要对一个文件或组件很快上手（比如，小组新来的开发工程师），用一个工具可以把所有的变动和相关的评审注释排成一列，并且用电影或幻灯显示的方式回放。这电影拿不了奥斯卡，但对于帮助新成员（或是老员工负责新的任务）很快上手肯定是有着巨大的潜力的。

评审的两面性

对大多数人来说，评审的主要好处是早点发现缺陷。评审实际上在这方面很在行，但对于任何严肃对待评审的团队来说，还有另一个好处。评审对团队的每个人都是非常棒的教育工具。开发工程师和测试工程师都能用评审过程来学习提高代码质量、学习更好的设计方法、编写更易维护的代码等技术。常规的进行代码评审给每个参与者都提供了机会来学习不同的和可能更好的编程方法。

工具、工具、到处都是工具

在一个拥有这么多程序员的公司工作的最大好处是，不管你面临什么样的问题，会有一大堆员工编写的软件工具来帮助你解决。

在一个拥有这么多程序员的公司工作的最大坏处是，你不得不在一大堆员工编写的软件工具中找到能解决你所面临的问题。

微软内部的工程和生产率工具网站多年来一直是一个巨大的资产，可用的工具数量每年都大量增长。增长的一个弊端是解决一个问题有很多种不同的工具，然而只是解决方法有微小的不同。在网页上查询“测试用具”和“测试框架”分别返回 25 和 51 个结果。虽然很多这些工具有独特的性能和用途，但在功能上有相当多的重复。

提炼、重用、回收

代码重用的概念（重复使用部分代码或组件）一直是软件工程的一个课题）。软件库，如随 Windows 一起发布的公共对话框库（common dialog library）就是代码重用的很好例子。这个库（comdlg32.dll）包含了打开、保存文件、打印、选择颜色及其它用户交互任务所有的对话框和相关函数。程序员不用编写他们自己的函数或生成他们自己的 UI 来打开保存文件；他们可以直接用公共对话框库中的函数。

几年前，当 Office 从仅仅是为需要表格程序和字处理的人而设计的一组应用程序，转型到为商业用户的联合应用程序套件，他们发现在不同的应用程序中很多函数都有重复。由于这个原因，mso.dll – 共享的 office 库诞生了。共享库使得 Office 团队的程序员能很容易的访问公共函数和在应用程序之间实现一致的功能和用户界面。一个更大的好处是测试团队只用在一个地方测试这些函数，所有人都受益。

在 Windows 和 Office 共享库很成功是因为他们都是开发平台- 也就是说，他们的设计企图都是让程序员能用所提供的功能来增加或增强基准平台结构。在 Office 代码重用成功也是因为它是一条产品线。要更好的利用工具和公共库的代码重用，挑战在于大多数部门和产品线都开发他们自己的方案，而不知道其它的方案。在大多数情况下，既没有动力，也很少有好处来共享代码。

问题在哪？

从很多层面上讲，没有问题。相比公司的其它更严重的问题，在库中拥有太多的轻量型 XML 解析器不算什么。而且，工具是在一个中央库中共享的，公司的每个人都能搜索工具库并选择他们喜欢的。更多的选择应该能提供给每个人更佳的选择，但反过来说也一样。在《左右为难的选择》（*Paradox of Choice*）⁵¹一书中，Barry Schwartz 讨论了为什么考虑的选项过多，最终的决定就更难—这只是问题的一部分，还有更多的。

微软还有一个很大的好处是每个产品组都能设置他们自己的目标、自己的愿景、和决定他们处理他们每天面临的工程问题的方法。除了为节省时间而采用或重用工具和工具库的代码外，就没有别的动力了。

⁵¹ Schwartz, Barry The Paradox of Choice, 2004

但是大家还是觉得代码重用的应用还不够，能够改进的主要目标领域有重复用工、和“不是在这发明”（NIH – not invented here）症状等。为了能让工具在像微软这样的大公司被多个部门采用，光共享工具是不够的，代码也得共享。

开放式的开发

在小组之间共享代码和工具，需要满足每个小组的独特的需求。如果一个小组或个人不能定制代码或工具，或是拥有者不能作他们所需要的改变，他们就只能求助于生成他们自己的拷贝，或是重新做起。除非是任何人都能在代码中作增加或改动。

在 2007 年，微软启动了一个内部网站叫做 **CodeBox**。**CodeBox** 使得微软的工程师能够创建、存放、管理合作式的项目。**CodeBox** 是由工程卓越组（Engineering Excellence team）建的内部共享应用程序，从外形到感觉都像 **CodePlex** (<http://www.codeplex.com>)。**CodeBox** 包括有源代码控制，可以让任何人对任意项目作增加和改进。项目拥有者可以完全决定接受哪些改动，而作改动的人，如果他们需要保留一些还没有被接受的特殊的功能，可以在一个分支（branch）或分叉（分支或分叉是最初源代码的一个特定的拷贝，保存有同样的历史信息）中继续自由工作。



图15-4 微软CodeBox

CodeBox 的使用发展很快。很多以前工具网站上的受欢迎的工具都已经被迁移到 **CodeBox** 的共享源模型。从 2007 年 7 月到 2008 年 3 月，每周加入的由 50 增长到 400。除了共享工具和工具库，大的部门用 **CodeBox** 作为开发能成为微软的新产品的应用程序的工作空间。

现在评价 **CodeBox** 对提高代码重用的长期效果还为时过早，但最初的预测很不错。随着采用和使用的增长，共享的代码和工具可以在开发成本的减少、更高质量的工具、把不停的重新发明转换为建立在整个工程社区的知识和经验上会对整个公司有好处。

本章小结

软件工程在不断进步，其复杂性也在增长，随之而来不断有更大的新挑战涌现。对很多软件工程师来说，这些挑战是这个职业的兴奋点和吸引之处的很大一部分。在注意今天的问题的同时，也企盼和对将来会出现的问题采取行动，是微软和软件产业的技术领头人的重要特质。在失败分析、代码评审、虚拟机的应用、和代码重用方面的改进只是微软工程师所面临的众多的巨大挑战中的四个。这些挑战，和不断改进软件工程的努力，是微软文化中令人激动的一部分。

第 16 章 建设未来

阿伦·培智

与软件开发比较，软件测试仍是一个相对较新的领域。像计算机应用公司（Computer Usage Corporation – CUC）和应用数据研究（Applied Data Research – ADR）这样的公司在二十世纪五十年代就成立了，并给计算机制造和商业用户提供系统软件和应用程序。那时候，测试和调试是一个意思，而且都是程序员的活。在过去的几年里，测试分立出来成了单独的活动，由“寻找缺陷”和验证软件符合要求的行动转变为寻找错误和衡量质量的活动。

在大多数情况下，软件测试仍然是为了检验软件的功能，并试图在产品到达用户手中之前找到最严重的错误。当今的软件程序更大、更复杂、被应用于更多变的场景中、拥有大量的用户。尽管在软件公司和 IT 部门里，有更多的测试发生在产品周期的更早期，我们还不清楚这样的投资对于现今越来越复杂的软件有没有起到很大的作用。

就算我们在复杂度日益增加的情况下取得了技术进步，提高了软件质量，我们还得问这个明显的问题：测试下一步该向何处走？

前瞻性思考的需求

在软件测试的艺术（*The Art of Software Testing*）一书中，Glenford Myers 率先讲到了测试从调试到验证的发展⁵²。在 1988 年，Gelperin 和 Hetzel 详细阐述了软件测试的发展和成长⁵³，并提出测试活动正朝着预防活动的方向发展。二十年之后，测试还在朝着预防活动的方向发展。我们总是会需要验证和分析，但软件质量的最大幅提高可能还是要靠预防技术才能实现。

“通过追本溯源来进行前瞻性思考”

就像这个故事讲到的，有一天一个村民在村子边的一条河边上走，看到有个人在河里快要淹死了。他跳进河里，把这个人救上来了。还没等他休息，他看到还有一个人，他一边喊着快帮忙啊，一边又回去接着救人。河里出现了更多的溺水者，更多的村民也都被召集来加入救人的行列。在一团混乱中，有个人走开了，沿着河边的一条小道朝上游走。一个村民叫住他问：“你去哪儿啊？我们需要你的帮助。”他说道，“我要找出是谁在把这些人扔进河里”。

在软件测试周期里，我们常听到“追本溯源求质量”这个说法，但在大多数情况下，都没有充分被做到。现今的大多数软件都太复杂、太庞大，仅靠测试来提高质量也太昂贵。几乎每一本我拥有的计算机科学的书都有一张图显示一个缺陷在

⁵² Myers, Glenford J. (1979). 软件测试的艺术（*The Art of Software Testing*）

⁵³ Gelperin, D.; B. Hetzel (1988). “软件测试的成长（*The Growth of Software Testing*）”， ACM 通讯

不同时间的成本，然而软件产业还是在依靠软件工程的最后环节—测试，来保证产品质量的很大部分。表 16-1，摘自代码大全（Code Complete）⁵⁴，就是一个根据缺陷出现的阶段来估测成本增加的例子。

检测时间					
引入时间	需求	体系结构	建设	系统测试	发布之后
需求	1	3	5-10	10	10-100
体系结构	-	1	10	15	25-100
建设	-	-	1	1	10-25

表16-1 根据缺陷的引入和检测时间，修正所需的平均成本

努力培养质量文化

Joseph Juran 因为对质量问题的考虑的贡献而声名远扬，他在质量问题中加入了“人”这一维度。Juran 认为文化方面的阻碍是质量问题的根本原因。

文化是社区或组织的成员表现出来的共同的信念、价值观、态度、制度和行为模式。质量文化就是社区成员在质量方面享有共同的价值观、态度和信念，并且每天都以这些为驱动力来对待日常工作。如果质量没有像这样被刻画进文化中，质量实践就会被看作工程拼图中可以“稍后”再拼的一块。

不幸的是，世上没有“即时”质量这种事，把质量放在产品周期的最后，或期待质量发生在产品周期的最后是愚蠢的。如果没有把质量放在工程过程的最前线，最后质量就不可能达到能接受的程度。在最近的一次会议上，Watts Humphrey 把这个长期存在的问题重新表述为：“如果你想从测试中得到高质量的产品，首先要有一个高质量的产品可以测试。”而我们经常回避的难题，是怎样才能得到质量较高的产品来测试。

测试和质量保证

Machiavelli 曾经写道“在疾病早期很容易治愈，但很难检测到，随着时间的推移，如果开始时没有被检测到或治疗，它就变得很容易检测，但很难治愈。”⁵⁵ 在软件领域这个说法也很明显；早期的检测和预防对于防止以后很难“治愈”的问题最关键。

在微软（在软件产业的其他地方也一样）用于提高质量的最常见方法是测试。测试是在用户发现软件的缺陷之前把它们找到。实际上，因为有行业中最优秀的软件测试员，和在产品周期的早期就包括测试这一常用实践，微软在产品周期的早期发现缺陷方面效率非常高。尽管这已经很不错了，但最主要的方法还是用测试来取得产品质量。

测试经常被当作质量保证的同义词；然而，它们是两种不同的工程活动。测试是一种被动方法—通过多种的检测和调查技术来找到潜藏在产品中的缺陷，而质量

⁵⁴ Steve McConnell, 代码大全（Code Complete），2004

⁵⁵ Nicolo Machiavelli, *The Prince*

保证是一种主动方法—建造一个预防和质量文化固化在开发周期中的环境。质量保证的方方面面在所有领域中都能见到，但在测试领域中最常见。

微软的卓越工程部（Engineering Excellence group）设计了一门课程，并每隔一段时间就提供给微软的高级测试员，而其中很大部分的材料是关于质量保证技术，而不是测试技术。在很多情况下，高级测试员的职能更像是质量保证而不是测试，主要的职能是提高会影响到一个组或部门内的所有工程领域的质量实践。

质量该谁管？

很多年前我会问这个问题：“质量该谁管？”而回答几乎总是“测试团队管质量。”今天，当我问这个问题时，回答则通常是“每个人都有份。”虽然这在有些人看来是更好的回答，SEI 的 W. Mark Manduke 则写道：“当质量被宣称为每个人的责任时，没人会真正被指派为之负责，质量问题就会蜕变成一团混乱，每天面临危机。”他还下结论说“...当管理层真正在质量文化上下了决心，每个人才会切实为质量负责。”⁵⁶一个每个人都要管质量的系统需要质量文化。没有这样的文化，每个团队在质量上都会打折扣。开发队伍可能会为了节省时间而不做代码评审，项目管理可能会在规格说明上走捷径，或是在“完成”的定义上打马虎眼，测试队伍会在产品周期进展很深的情况下改变测试执行的目标或覆盖率。尽管我们多方努力让质量保证的过程到位，在实践中工程团队常常会为了赶完工期限或别的目标而在质量实践上破例。虽然为了赶上发布日期或其他截止期限而灵活处事很重要，但质量经常因为缺乏真正的质量专管而受到损害。

整个测试团队可能会拥有质量保证的某些方面，但他们很少会是提倡或影响采纳质量文化的最佳人选。高级主管可以当质量倡导者，但他们的工作重点在于也应当在于管理团队、发布产品、和成功运行企业。虽然他们可能也会考虑质量目标，但他们也很少会是质量文化的倡导者。在大多数组里，管理领导团队（通常有开发、测试和项目管理等的机构领导人）会负责承担质量管理。这些领导人负责和驱动团队的工程过程，也处于能评价、估计、和实现基于质量的工程实践的最佳机构位置。不幸的是，在整个产品工程周期的过程中，他们主要关心的并不是质量软件和质量软件工程实践。

有高级管理层对质量文化的支持，这还不完全够。在质量文化中，每个员工都会对质量起作用。在制造业很多的重要质量改进都是从工人的建议中来的。在汽车产业，每个日本汽车员工平均每年会提供 28 个建议，而这些建议的 80% 都被实现了⁵⁷。

理想情况下在微软各个领域的工程师都提出建议来提高质量。当一个团队没有质量文化时，建议会很少，其中极为稀少的会被实现。对质量文化的淡漠会造成团队成员在激情和承诺方面的其他挑战。

⁵⁶ STQE 杂志. Nov/Dec 2003 (Vol. 5, Issue 6)

⁵⁷ 有远见的领导人（The Visionary Leader），Wall, Solum, and Sobul

质量成本

质量成本，是一个广泛使用的术语，但也是一个被广泛误解的词。“质量成本”不是生产优质产品或服务的花费，而是“未能”生产优质产品或服务的代价。每次需要返工的时候，质量成本就提高了。显而易见的例子包括：

- 重新编写或重新设计一个组件或功能
- 因测试失败或代码回归而重新测试
- 重新设计一个工具用作工程过程一部分
- 对一个服务或过程，比如签入系统、构建系统或评审策略进行反修换而言之，任何如果质量完美就不应该发生的成本，都会加入到质量成本中。这又被称为不良质量成本（COPQ--Cost of Poor Quality）。

在《质量免费》一书中，Philip Crosby 将质量成本划分为三类：鉴定成本、预防成本和失败成本。鉴定成本包括薪资（包含付给测试员的费用），和发布所需的费用。预防成本是跟实现和维护预防技术相关的开销。失败成本是返工的花费（或如上所述的 COQP）。我们很少会衡量预防成本，也更少会去奖励它们。相反，我们更愿关注于英雄行为——工作一整夜去修复最后几个缺陷，或者在最后关头调研怎样绕开一个严重问题等。

想一想下面摘自《不停地在产品开发中救火》⁵⁸的一段话：

偶尔会有一位工程师或经理级的超级明星，能够接手这些最后的改动中的一个，迎难而上，披荆斩棘，开辟道路并到达成功。然后我们就尊奉他为英雄。并且其他每个想成为英雄的人都会说：“噢，在这儿那才是让人看重的。”先前几个月所做的日常工作被认为不重要，测试并在问题发生之前消除它们也被认为不重要。被认为重要的却是在最后时刻做改动再强行把它们弄通。

这段话在很多读者看来很熟悉。我们不需要英雄行为；我们需要防止对它们的需求。

测试的新角色

有些人也许会问：假如质量在设计阶段被充分考虑并集成进去，并且开发人员生成了少得多的缺陷，那么测试团队还能干什么呢？事实的真相是，现在寻找缺陷可能太容易了。在一个每个人都把质量当成习惯的组织中，测试角色的着重点就从找到一大堆缺陷转移到了集中模拟用户场景、同事评审和校验方面。还是会有缺陷去找，但是要找到它们要比今天难得多。测试员将会有时间去调查复杂的集成场景，并且，由于不用操心基本的功能性问题，可以找到很多原本会被用户在产品发布后发现的缺陷，从而免除大部分的由补丁、更新和维护包带来的巨大开销。

测试角色（以及其他领域）的其他人可能开始担任质量保证角色并发挥作用。这些

⁵⁸ http://web.mit.edu/nelsonr/www/TippingV2_0-sub_doc.pdf

个人将会分析和实施过程改进、缺陷预防技术及基础设施，以及在他们的机构中担任质量思考和质量文化的大使。这个角色的成长和发展会成为在全公司建立和宣扬质量文化的关键。

关于缺陷预防的更多想法，包括根本原因的分析，和质量文化的需求可以在《实用缺陷预防手册》(The Practical Guide to Defect Prevention, 微软出版社, 2007)一书中找到，这本书是我们在微软从事测试的同事写的。

测试领域的领导力

本书出版时微软有超过 9,000 名全职的测试工程师，随之而来的是，需要找到一条途径来联结、分享和提高这个专业领域的能力。对一个有上百测试人员的组织而言这是一项巨大的挑战，而对于拥有众多遍及世界各地的测试人员的微软而言这个挑战是前所未有的。领导力是开拓出一个生机勃勃的测试工程专业领域的关键，也是微软解决上述问题的办法。

微软测试领导团队

在微软的每个工程部门中都存在领导团队，微软测试领导团队 (MSTLT, 全称是 Microsoft Test Leadership Team) 则是这些领导团队中最活跃最成功的一支。这支团队的首要目标是为全公司范围内的测试员提供共享测试知识和实践的支持。这个目标可以从微软测试领导团队的使命 (参见侧栏) 反映出来，也反映在他们今天所取得的成就上。

微软测试领导团队愿景

微软测试领导团队的使命是：创建一个跨事业部门的论坛，来提交和解决测试专业普遍面临的挑战和问题。微软测试领导团队将会推进培训教育，并推动事业部门测试团队采纳最佳实践以解决常见的挑战。微软测试领导团队将区分并尊重事业部门的不同，据此进行本地最佳实践的优化或差异化。

微软测试领导团队由代表微软各个产品线的约 25 名最高级别的测试经理、总监、总经理和副总裁组成。领导团队会员资格的选举产生过程对于确保它的成功至关重要。会员资格的评选是基于资历级别，也会基于测试领导团队的主席和产品线副总裁的提名或认可 (参见第二章，“微软的软件测试工程师”)。如果没有适当的构成，领导团队不可能以平衡的有代表性的方式为他们的社区发言。此外，他们还需要机构内领导的支持来充分而公平地代表他们的机构。

测试领导团队主席

Grant George 是（并且还在担任）首任测试领导团队的主席。从 2003 年八月起，他出任微软测试领导团队主席一职并且培育了团队最初的成长。2004 年，微软卓越工程机构开始支持这些虚拟团队，正因如此，测试领导团队主席成为卓越工程机构的紧密合作伙伴。这一虚拟整合也使得微软测试领导团队在评审诸如测试工程师培训、卓越工程奖励评选和测试工程师职业发展路线规划（由微软测试领导团队和人力资源联合开发）并为它们提供反馈方面起着积极作用。

启动微软测试领导团队

发件人： Grant George

发送时间： 星期五，10月17日，2003 年下午1:11

主题： 微软测试领导团队

时间： 星期三，12月17日，2003年 下午12:00-下午2:00 (GMT-08:00) 太平洋时间 (US & Canada).

地点： 17/3002

诚如您可能知道的，正有联合的努力来组建微软跨产品组的“专业领域领导团队”，通过创建社区和协作来解决我们事业中各个领域所面对的一些重大挑战。在前方 “第12波”（下一个2-3年）的时段上，会有许多复杂而充满挑战的产品周期在等待着我们，我们应该更规律性地聚会来应对那些挑战，平衡和协调领导力和方法，然后传授给各个测试团队，用以交付高质量的产品——即使交付不了，我们也会商议怎样来衡量微软测试工程持续的增长、扩大的影响和提升的效率。

我们新近组建的领导团队的启动会议将于11月12日从中午到下午2点在 17/3002举行。

在这第一次的会议中我们将讨论会议的频率和机制，以及该如何作为一支领导团队工作，该如何沟通/代表各自所代表的事业部门的测试团队的民意，和该如何划分在将要来临的产品周期中（重复一下，下一个2到3年），我们的测试业务所面临的问题集合的优先级别。在这次会议上，我们将审阅针对这些问题的特定建议，同时也会审阅其他出现在我们共有测试工程雷达上的其它问题。有许多我们能做也应该做出协调改进，并引领我们的事业前进的方面——测试工具、测试自动化环境方法和策略、跨产品组的测试和技术交付、测试人员的成长和表现的管理实践，都这些话题中。

在此我提前对您帮助和参与这个新论坛表示感谢。

-Grant

测试领导力在行动

每年，微软测试领导团队（MSTLT）与卓越工程团队联合，选择 3-6 个意义重大的、涉及微软测试增长与提高所必需的领域，作为研究或调查的方向。过去的工作包括：高级测试员的职业道路、自动化测试经验分享、实验室管理和测试工程师职业阶段概况。

MSTLT 的会议每月举行。会议的议事日程每个月会有变化，但会关注几个主要的方面。它们包括：

- **年度倡议的更新** – 至少由一名MSTLT的会员来负责MSTLT的各项倡议，并且负责做每年四次的工作进展陈述。
- **人力资源的报告** – MSTLT与公司的人力资源部门建立了紧密的关系。会议为人力资源提供了传播信息给MSTLT的机会，同时也给了他们从MSTLT收集反馈的机会。
- **领导评议的其他议题** 工程要求的变化或其他影响到工程的企业政策的变化会先提交给领导团队的会议，然后再传递给所有的测试人员。拥有了这些背景信息，MSTLT 成员可以将有关信息和准确的事实及适当的背景材料一起分发到各自的部门。

测试架构师团队

测试架构师团队 (TAG) 由测试领域的，不从事管理工作的资深领军人员组成。当 Soma Somasegar（索玛·索马塞加尔，他当时是视窗部门的测试主任）创建测试架构师这一职位时，他鼓励最初的测试架构师们定期举行会议，讨论各自部门所面临的挑战和成功的经历。这个团队开始只有六位测试架构师，大多属于视窗部门，但现在已经增长到 40 多位测试架构师，几乎涉及微软每个主要部门。最初，团队成员都是拥有“测试架构师”头衔的测试人员，随着微软更多地采用标准的职位头衔（例如 SDET，资深 SDET，和首席 SDET，在第二章中详述），这个团队现在包括了所有履行测试架构师职能（而不论他们的具体头衔）的资深测试人员。

测试架构师团队 (TAG) — 你们就是它！

发信人: S. Somasegar

发信时间: 2001 年 1 月 11 日, 星期四, 10:46

主题: 测试架构师

我们正在创造一个新的被称为测试架构师的岗位。我们有很多机会提供强有力的技术领导来解决怎样测试及测试什么的课题，这样我们可以确保不断将质量控制向上游推进到我们的研发流程中去，智能地实现测试自动化以提高效率，在产品中构造测试接口 (testability hook) 等等。我们需要进行一些很大的投资来解决在不断增大的硬件基础上测试日益复杂的产品所带来的可扩展性问题。

测试架构师这一岗位提供了一个机会让我们投资于测试团队中的资深个人技术贡献者从而把重点放在这些关键问题上。

创造测试架构师这一岗位的主要目标是：

的资深个人技术贡献者去解决Windows开发团队面临的测试难题和全球化测试课中的个人贡献者创造一条技术方面的职业发展道路

测试架构师需要注重的一些关键问题包括：

的研发流程，使质量控制向上游移动
高效工作方法、归并和领导，提高测试流程的吞吐量

测试架构师的特征：

去解决测试团队所面临的最具挑战性的问题
个人贡献者
式做法和产品的研发流程有坚实的理解
立工作和跨团队协调来开发和部署测试解决方案的能力

测试架构师将由副总裁（V P）提名并留在其目前的工作团队。他们将集中精力解决测试团队面临的各方面的关键问题。测试架构师将组成一个虚拟团队，定期召开会议来相互协作并同包括微软研究院在内的其他团体合作。每个测试架构师将负责代表所在团队面临的独特问题，负责处理跨团队的问题以及实现和驱动团队中的关键举措。

团队成立近 10 年后，TAG 成员仍然几乎每周开会。测试架构师们主要致力于处理各自产品组的技术问题，但 TAG 发现定期召开会议交流想法及微软测试和质量问题的最佳做法带来了许多好处。事实上，许多会议的议程集中在分享目前的挑战和成功的事例这一原始使命上。

让微软最资深的测试人员定期审查、集体讨论、剖析复杂的测试问题的解决方案的价值是无法估量的。近年来，TAG 已成为测试中的新思维、新方法、新工具的“共鸣板”。来自微软各部门的测试团队关于不同构思及相关实现的介绍和演示，排满了许多 TAG 会议的议程。测试架构师团队所提供的反馈的价值和深度受到尊重和追求。每年有几个会议被预留为“TAG 业务”，其中包括讨论 TAG 推动下的全公司范围的倡议，以及讨论其他以 TAG 为重要贡献者的项目（例如 MSDN 测试人中心）。也许定期会议的最大好处在于联络交流的价值。广泛的同行讨论和对于会议中展示的公司内各种工作的看法给了 TAG 成员作出影响整个公司的决策所需的很多知识和信息。

测试架构师团队为微软提供价值的另一种方法是保持开放态度，愿意同任何人就任何议题进行非正式讨论。电子邮件讨论和临时会议请求是一个开端。这方面最成功的倡议之一是“和测试架构师共进午餐”活动。每个月中公司各个部门的测试架构师会和拥有不同资历级别来自不同产品组的测试人员进行几十个午餐谈话。

和测试架构师共进午餐

测试架构师内部网站上有如下申明：

测试架构师团队带来的最大福利之一是联络交流的力量。因为团队的成员来自整个微软的各个部门，我们能听到不同的观点，分享技术，并减少影响跨团队协作的“六度分离”，这一切真是非常棒。我们希望找到一种方法，将这一福利扩展到所有的测试人员，使我们测试架构师团队更易被接触和访问，在全公司内建立更多的关系来分享最佳测试做法和

提升对每个人的要求。为此，我们想和你谈谈。你想和其他产品组的测试人员谈你所面临的有关技术问题吗？你想要进行集体讨论研究你正在考虑的解决方案吗？想更多了解测试架构师的工作吗？想进行一次关于职业发展的非正式讨论吗？

我们也是这样！我们很愿意共进午餐，同时分享有关疑难问题的信息，更好地了解你们团队所面临的问题来决定我们是否能帮助或想到有谁能帮助。这样做还能增长我们的联络名单来发展微软的测试团体。如果你有兴趣与我们中的一员坐下来谈话，请随时与下面名单中的任何人联系。

我们的电子日历都是最新的，我们的兴趣话题和相关经验在下面会被列出。简单地发送一个午餐会议的请求——你甚至不需要来到我们办公室附近的食堂！我们期待着与您交谈。

注：微软研究院有一整个团队研究软件测试和验证。他们的研究项目的描述在 <http://research.microsoft.com/srr>。

卓越测试

微软在 2003 年创造了卓越工程（E E）团队。图队的宗旨除了技术培训（团队的核心是之前的技术教育小组）外，是发现和分享整个公司的工程最佳做法。此团队作为一个整体包含了所有的工程领域，它其中的卓越测试团队代表测试领域。卓越测试团队认为微软的每一个测试人员都是他们的客户。从新员工到副总裁，测试团体整体上的成功是这个团队的驱动力。卓越测试团队的主要任务可以概括为共享、帮助、和交流。

共享

共享与卓越工程共享最佳做法的目标相配合。就卓越测试而言，共享包括了共享做法，工具和经验。

- 做法** 卓越测试团队发现有可能被微软不同小组或部门使用的做法或方法。目的不是让每个人都以同样的方式工作，而是要确定可能被他人采纳的良好的工作。

- 工具** 以上理念同样适用于工具。在大多数情况下，卓越测试团队提供的核心培训与工具无关，这就是说培训侧重于技术和方法，而不是提倡某一特定工具。然而，当一个特定的工具是被确定为解决某一特定问题或应用某一技术的最佳方案，它可以在核心技术培训课程中被使用和推广。这方面的一个例子是在第 5 章“功能测试技术”中讨论的 PICT 工具。这是一个用于配对分析的完美工具，被广泛使用在这一技术的教学中。
- 经验** 微软的团队有很多不同的工作方式，有些团队无法知悉其他团队的可被借鉴的经验。卓越测试试图通过案例研究、报告会（“Test Talks”）、和访谈收集这些经验，然后和不同的团队分享这些经验。

注： 每年会举行20多个Test Talks（“测试讲座”，介绍有效的测试做法的报告会）。Test Talk对微软所有的测试人员开放（包括为Redmond 以外的测试人员准备的实况在线转播和录音录像）。

卓越工程论坛（The Engineering Excellence Forum）

在微软，卓越工程论坛是卓越测试团队和整个卓越工程团队与所有工程师交流工作经验的最明显的方式。这个论坛是一个一年一度的活动，每年六月举行五天。该论坛充满了专题报告，专家小组讨论，演示，当然，还有免费的午餐和小吃。卓越工程论坛相似于一个一年一度的巨型软件开发或测试的会议，但不同的是，它针对的是微软员工。数以千计的出席者几乎都是微软员工，同时数百名微软员工参与了策划，准备，和专题报告。这是员工们学习微软其他团队的工作经验和技术工具的大好机会。

帮助

卓越测试团队在整个测试社团担任着促进者和专家的角色。他们以这些角色与其他测试工程师建立和保持联系。事实上，卓越测试团队的标语是“我们把不同点与质量联系在一起”（*We connect the dots . . . to quality*）。这个品牌声明驱动了卓越测试团队的大部分工作。这个团队帮助测试工程师的一些方式包括促进，提供答案，和关系：

- 促进** 卓越测试团队成员经常协调高管简报，产品线的战略会议，和小组事后总结讨论。卓越测试团队以在产品组外旁观者的角度，提供战略见解和看法是被需求与看重的。
- 答案** 微软的工程师知道卓越测试团队了解测试，有什么问题就问他们。在许多情况下，卓越测试团队成员知道答案，当他们不知道时，他们

的关系能使他们能够很快找到答案。

有时，卓越测试团队成员称自己是测试治疗师，并一对一地与测试工程师见面讨论有关职业发展，管理方面的挑战，或工作与生活相互平衡的问题。

- 关系** 也许卓越测试团队的最大价值是他们的关系——他们与 TLT，TAG，微软研究院，产品线领导层的互动确保他们可以减少任何微软工程师之间的隔离度，而帮助他们迅速及有效地解决问题。

沟通

另一个卓越测试团队的关键价值是交流沟通他们所知道的和所发现的。在一个大型社区分享有关信息有着巨大的价值。一些来自卓越测试团队的通讯包括以下内容：

- 每月通讯简报为微软所有测试工程师提供了即将举行的活动，MSTLT 倡议的现状，和有关测试领域的通知的信息。
- 有关大学关系的讨论，包括测试和工程课程的评论，以及与大学系主任、开设测试和质量课程的教授的一般通讯。
- 微软测试中心（<http://www.msdn.com/testercenter>），就像这本书，提供了微软内部测试工程师所采用的测试方法和途径。这个在 2007 年下半年开始运行的网站，正在迅速成长。微软员工现在创造了大部分内容，但行业内的测试工程师提供了越来越多的网站内容，而且预计在将来会成为更大的贡献者。

一起工作

卓越工程团队最近改造了办公室空间，为以不同工程领域为重点的卓越工程团队增加了团队房间。微软为工程师提供个人办公室的政策是众所周知的，在 DeMarco 和 Lister 的关于软件工程师生产力的影响性的著作《Peopleware》⁸书中解释了为什么个人办公室会帮助工程师提高生产率和个人办公室是怎样帮助工程师提高生产率的。

微软为什么要改变自己的方式和制造一个“办公室小隔间的农场”？卓越工程团队的决定几乎是纯粹的实验。敏捷软件开发工作者会谈论团队房间所提供的合作的必要性，但我也遇到了许多宁愿在树屋工作也不想与别人合用办公室的

人。我们团队的工作带有些合作性质，但肯定不能被视为像一个软件产品的合作性质。然而，在 2006 年 5 月，在办公室改建的前几天，我们搬到一个临时办公室，几个月后，我们搬回到我们的崭新的团队房间。

我不知道我们之间任何人有没有完全知道应该期待些什么，但总体来说，我认为我们都感到惊喜。我认为最大的好处是，我们六人分享有近 1000 平方英尺的空间。由于我们的工作性质，地方看起来更大。因为除了小组会议，我们都在同一时间在办公室是极为罕见的。我们也有灵活性，如果我们要增加一个人，我们可以很容易地做些小改变来为此人腾出空间。我们有沙发，椅子，装在天花板上的投影机，Xbox 游戏机，以及咖啡机。几乎每一面墙都可作为一个白板用，我们还在大部分不是白板的墙上挂了白板。大部分工作空间沿着有窗的一堵墙壁。

图 16-1 显示了我们房间的现有配置。

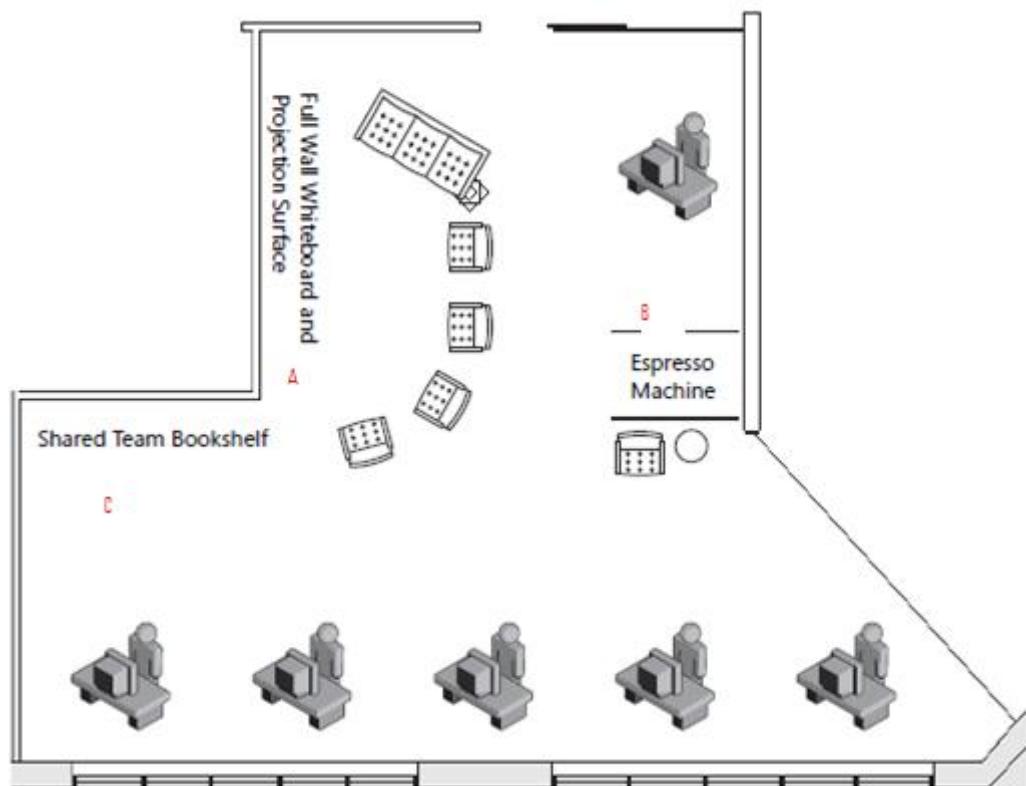


图 16-1 卓越工程小组房间

(图内文字翻译:

- A.白板墙和投影机表面
- B.咖啡机
- C.小组书架)

房间里有时候也有“忙”的时候，有时我们都不得不用耳机来减少分心，但大部分时间我们对我们的房间都感到满意。我们有自己的空间，但我们需要的时候可以面对面交谈。我们的安排是“办公室小隔间的农场”和个人办公室中间的一个折中。

我们房间现在有 6 人，将来可能还可以多容纳几个人，但我认为团队房间概念在规模接近两位数时注定要失败。共享空间的最大好处是，我们都比以往任何时候更了解我们的队友们正在做的工作，还有，我们有一个有趣的，现代的工作场所。

也许我们的新房间的成功的最有效的指标是当我开始领导卓越测试团队时，我选择把我的写字桌放在团队房间里。我也有一个办公室被我们用于 1:1（读作“一对一”）会议，面试，和其他非公开会议，但我喜欢把大部分工作时间花在这个被我们称为“测试大厅”的地方。

关注未来

也许卓越测试团队最关键的角色是预计微软测试工程师未来的需求，并积极主动地确定关于未来软件测试的长远规划。这种长远规划是每年的规划和卓越测试的各种倡议的基础。长远规划的目的，像大部分好的远见规划一样，是给我们这个工程领域指引方向和指导需要做的工作。

微软公司卓越测试主任

微软最独特的工作职位之一是卓越测试主任。从某种意义上来说，这个职位仅仅是负责卓越测试团队的经理，但这个职位也对整个微软的测试工程师有着很大的影响。

毫不奇怪，这本书的三位作者都担任过这个职务。目前，测试主任的职位与卓越测试主任的职位联系在一起，但原来这只是一个资深的负责推进微软测试职业的测试职位。

以下是先后担任该测试主任职务的人：

- Dave Moore (测试开发主任), 1991–1994
- Roger Sherman (测试主任), 1994–1997
- James Tierney (测试主任), 1997–2000
- Barry Preppernau (测试主任), 2000–2002
- William Rollison (测试主任), 2002–2004
- Ken Johnston (卓越测试主任), 2004–2006
- James Rodrigues (卓越测试主任), 2006–2007
- Alan Page (卓越测试主任), 2007–至今

因为这是个管理一个小的战略团队的全职职位，此职位也起到联系 TLT 和 TAG 的作用，并注重与这两个组织主席的密切合作。

三方面的领导

微软测试的领导团队，测试架构师集团，和卓越测试这些组织为微软测试文化的发展和维护填补了重要的角色，如图 16-2。每个组织有一个领导职位负责面向整个专业、跨越不同团队的工作。至于 TLT 和 TAG 的主席，这些是虚拟的团队，主席的职务是处在该位置的个人的额外职责。卓越测试主任则是一个全时的工作岗位。

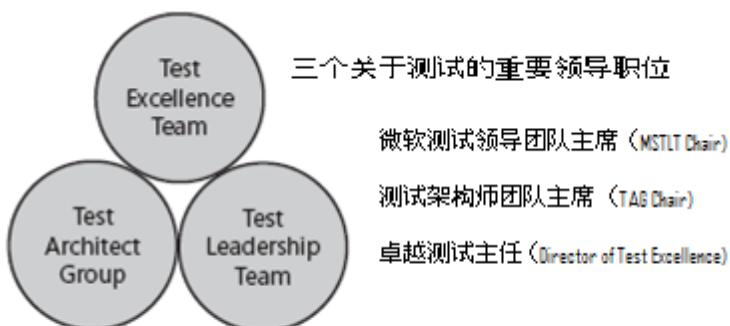


FIGURE 16-2 Three key leadership positions in the three test support organizations.

图 16-2 三个关于测试的重要领导职位

为未来创新

所有的微软新员工都会参加新员工大会（NEO）。微软的新员工把进公司头两天

的大部分时间都花在学习有关政策，组织，和其他我们认为所有新员工应该知道的专题上。我经常作有关创新这个专题的报告。微软肯定会继续在我们的新产品，以及包括 Zune 播放器和桌面计算的新领域中创新，但我试图传达的重要信息是：创新可能在我们制造产品的方式方法中，在我们如何让客户用我们的技术去创新的过程中，更加重要。

当我想到未来的软件，或当我看到科幻电影中描绘的软件时，我总是注意到两件事。第一，软件将无处不在。像今天的软件那样流行，未来的软件将影响我们生活的几乎每一个方面。第二，软件会正常工作。我想不出任何一次当我观看侦探或科学家在未来使用软件来帮助他们解决案例或问题时，该软件系统没有完美地为他们工作，而且我非常肯定从未看到他们所使用的软件崩溃。这是我对软件的愿景——软件无处不在而且总是正常工作。

你读这本书到现在就会意识到，达到这个愿望是一个困难的过程。单靠我们测试工程师是不够的。如果我们想实现这个愿景，我们软件工程工业需要继续挑战自己，创新我们制作软件的过程和工具。这是一个我全心接受，并期待的挑战，我希望所有的读者将和我一样。

如果您对这本书的作者们有任何问题或意见（或要报告错误），或想知道我们对这本书里提到的课题的新思考，请访问 <http://www.hwtsam.com>。我们很想听到您想要说的话。

索引

英文	中文翻译
(BC) formula	(BC) 公式
Abort result (automated tests)	“中止”结果（自动化测试）
access (external) to bug tracking system	对 bug 跟踪系统的访问权限（外部）
accessibility testing	辅助功能测试
accessibility testing – See also usability testing	辅助功能测试 – 参见 可用性测试
Accessible Event Watcher (AccEvent) tool	辅助功能事件查看器（AccEvent）工具
Accessible Explorer tool	辅助功能浏览器工具
accessible technology tools	辅助功能技术工具
accountability for quality	质量可靠度
accuracy of test automation	自动化测试的精确度
Act phase (PDCA cycle)	“行动”阶段（PDCA 生命周期）
action simulation in UI automation	行为拟合用户界面自动化
active (bug status)	“活动”（bug 状态）
Active Accessibility software development kit	Active Accessibility 软件开发工具包
activities of code reviews monitoring	代码复审活动监督
ad hoc approach combinatorial testing	临时解决方案组合测试
ADDIE model	ADDIE 模型
Adopt an App program	“领养应用程序”项目
AFA (automatic failure analysis)	AFA (自动错误分析)
Agile methodologies	敏捷开发方法学
all states path (graph theory)	全状态路径（图论）
all transitions path (graph theory)	全变换路径（图论）
alpha release defined	内测发布 定义
analysis (test pattern attribute)	“分析”（测试模式属性）
analysis phase	分析阶段
analysis phase (SEARCH test automation)	“分析”阶段（SEARCH 自动化测试）
analysis tools for analytical problem solving competency	分析工具 分析式问题解决的能力
analyzing parameter subsets	参数子集分析
API testing with models	带模型的 API 测试
application compatibility testing	应用程序兼容性测试

application libraries	应用程序库
application usage data collecting	应用程序使用方式数据 收集
approving big fixes. See triage for managing bugs	核准大型修正. 参见 bug 管理中的会审制度
architecture for	架构
as test case attribute	作为测试用例属性
assignment of bugs	bug 的指派
assumptions about test case readers	有关测试用例读者的断言
attributes of tracking systems	追踪系统的特征
audio features accessibility of authentication with WLID	听觉特性 辅助功能相关
Automate Everything attribute (BVTs)	Windows Live ID 验证
automated notification of automated service deployment	Automate Everything 属性 (BVTs)
automated tests	自动通知
automated tests . See also test automation	自动化服务部署
automatic failure analysis (AFA)	自动化的测试
automating	自动化测试. 参见测试自动化
automating models	自动错误分析(AFA)
automating models	自动化模型
automation (test case attribute)	模型自动化
AutomationElement elements	“自动化与否”(测试用例属性)
backlog of bugs	AutomationElement 元素
base choice (BC) approach	Bug 回溯日志
combinatorial testing	基础选择(BC) 途径 组合测试
baseline performance establishing	性能基线 建立
basic control flow diagrams (CFDs)	基本控制流图 (CFDs)
basics of modeling	模型的基础
basis path testing	基本路径测试
BATs (build acceptance tests)	BATs (构建验收型测试)
Bayesian Graphical Modeling (BGM)	贝叶斯图解模型 (BGM)
BC (base choice) approach	BC (基础路径) 途径 组合测试
combinatorial testing	行为和研究性测试
behavioral and exploratory testing	行为相关的及探索性的测试
behavioral and exploratory testing	行为测试. 参见 非功能性测试
behavioral testing. See nonfunctional testing	

best guess approach combinatorial testing	最佳尝试途径 组合测试
best practices	最佳实践
BGM (Bayesian Graphical Modeling)	BGM (贝叶斯图解模型)
bias in white box testing	白盒测试的偏差
Big Challenges (company value)	巨大挑战 (公司价值)
big company Microsoft as	大公司 微软公司规模
big picture performance considerations	性能考虑的大方向
BIOS clocks	BIOS 时钟
black box testing	黑盒测试
block testing	分块测试
boundary condition tests	边界条件测试
boundary testing of	边缘测试
boundary value analysis (BVA)	边界值分析 (BVA)
Boundary Value Analysis Test Pattern (example)	边界值分析测试模式(举例)
branching control flow testing. See decision testing	控制流分枝 测试. 参见 决策测试
breadth of Microsoft portfolio	微软公司投资组合的广度
breaking builds	中断构建
broken window theory	破窗理论
browserbased service	基于浏览器的服务性能指标
performance metrics	
browserbased service	基于浏览器的服务性能指标
performance metrics	
bruteforce UI automation	强力用户界面自动化
buckets for errors	收集错误的散列桶
bug	缺陷
bug backlog	bug 回溯日志
bug bars (limits on bugs assigned)	bug 标准 (bug 分配的界定)
bug bars (limits on bugs assigned)	bug 尺度 (bug 指派的局限性)
bug data as metrics of	缺陷数据作为指标
bug lifecycle	bug 生命周期
bug management. See managing bugs	Bug 管理。参见管理 bug
bug metrics	bug 指标, 度量
bug morphing	bug 变体
bug notification system	bug 通知系统
bug prioritization	缺陷划分优先次序
bug reports	bug 报告
bug severity	缺陷的严重性

bug workflow	bug 工作流
build acceptance tests (BATS)	构建验收型的测试 (BATS)
build labs	构建(建造)实验室
build process	版本过程管理
build verification tests (BVTs)	构建版本确认测试 (Build Verification Tests (BVTs) : 也叫做“冒烟测试”
building services on servers	运行于服务器上的构建服务
built-in accessibility features	内建的辅助功能特性
BVA (boundary value analysis)	BVA (边界值分析)
BVA Test Pattern (example)	BVA 测试模式 (举例)
BVTs (build verification tests)	BVTs (构建验证测试)
by design (bug resolution value)	“设计如此” (bug 解决状态值)
C# development	C#开发
calculating code complexity. See code complexity	计算代码复杂度. 参见 代码复杂度
call center data collecting	来自呼叫中心的数据收集
campus recruiting	校园招聘
capacity testing	能力测试
Carbon Allocation Model (CarBAM)	碳素分配模型 (CarBAM)
career paths in Test	测试的职业发展道路
CBO metric	CBO 指标
CEIP (Customer Experience Improvement Plan)	CEIP (客户体验改善计划)
CER (Corporate Error Reporting)	CER (企业错误报告)
CFDs (control flow diagrams)	CFDs (控制流图)
chair of test leadership team	测试领导团队主席
changes in code	代码变更
Check phase (PDCA cycle)	“校验”阶段(PDCA 生命周期)
checkin systems	代码签入系统
checklists for code reviews	代码审阅者的检查表
churn	改动
CIS (Cloud Infrastructure Services)	CIS (云计算基础服务)
CK metrics	CK 指标
classbased complexity metrics	基于类的复杂度指标
classic Microsoft bugs	经典的微软 bug
ClassSetup attribute	ClassSetup 属性
ClassTeardown attribute	ClassTeardown 属性
clean machines	干净机器(新安装的操作系统)
cleanup phase	清理阶段
closed (bug status)	“已关闭” (bug 状态)
cloud services	云服务
CLR (Common Language Runtime)	CLR (公用语言运行时系统)

CMMI (Capability Maturity Model Integrated)	CMMI (能力成熟度综合模型)
code analysis. See static analysis	代码分析. 参见 静态分析
code churn	代码改动
code complexity	代码复杂度
code coverage	代码覆盖率
code reuse	代码复用
code reviews	代码复审、审核
code snapshots	代码快照
CodeBox portal	CodeBox 门户网站
collaboration. See communication	合作. 参见 沟通
collecting data from customers. See CEIP (Customer Experience Improvement Plan)	向用户收集数据. 参见 CEIP (用户体验改善计划)
combination tests	组合测试
combinatorial analysis	组合分析
common bugs	常见 bug
comparing documents	文档比较
compatibility testing	兼容性测试
competencies	胜任能力
compilation errors	编译错误
complexity of code	代码复杂度
complexity of test automation	测试自动化复杂度
compound conditional clauses testing	复杂的谓词分表达式 测试
compressibility (metric)	可压缩性 (指标)
computerassisted testing	计算机辅助的测试
computing innovations waves of	计算创新 波浪
condition testing	条件测试
conditional clauses testing	条件分表达式测试
conditions (test case attribute)	条件 (测试用例属性)
confidence (competency)	自信心 (胜任能力)
configurability of bug tracking system	bug 跟踪系统的可配置性
configuration data collecting	配置数据 收集
configurations (test case attribute)	配置 (测试用例属性)
conformance in test time estimation	符合度 测试时长估计
Connect site	Connect 网站
connecting with customers	与客户建立联系
containerbased datacenters (container SKUs)	基于容器的数据中心 (容器 SKUs)

context bug	上下文 bug
continuous improvement. See process improvement	持续改进. 参见 过程改进
continuous quality improvement	持续的质量改进
contrast display	对比度 显示
control flow diagrams (CFDs)	控制流图 (CFDs)
control flow graphs	控制流图解
control flow modeling	控制流建模
control flow testing. See structural testing	控制流测试. 参见 结构化测试
control testability	控制可测试性
Corporate Error Reporting (CER)	企业错误报告 (CER)
cost of quality	质量代价
cost of test automation	测试自动化代价
counters (performance)	计数器 (性能)
counting	计数
counting bugs	Bugs 计数
counting test cases	测试用例计数
coupling between object classes (metric)	对象类间耦合 (指标)
coupling services	耦合 服务
coverage method (WER)	覆盖率方法 (WER)
Creative discipline	创新原则
credit card processing	信用卡处理流程
criteria for milestones	里程碑标准
Critical attribute (BVTs)	Critical 属性 (BVTs)
crossboundary collaboration (competency)	跨边界合作 (胜任能力)
crossreferencing test cases with automation scripts	使用自动化脚本在测试用例间建立交叉索引
crossreferencing with automation scripts	自动化脚本交叉引用
culture of quality	质量至上的文化
Customer Experience	用户体验改善计划 (CEIP)
Improvement Plan (CEIP)	
customer feedback systems	用户反馈系统
customer focus . See also customer feedback systems	关注用户 . 参见 用户反馈系统
customer impact of bugs	bug 造成的用户影响
customerdriven testing	用户驱动的测试
customerfocused innovation	关注用户的创新
cyclomatic complexity	圈复杂度
daily builds	每日构建

data coverage. See code coverage;	数据覆盖率. 参见 代码覆盖率
equivalence class partitioning (ECP)	等价类划分 (ECP)
data sanitization	数据消毒
DDE (defect detection effectiveness)	DDE (缺陷检测有效性)
debug/checked builds	调试用/已审阅构建
Debuggable and Maintainable attribute (BVTs)	Debuggable and Maintainable 属性 (BVTs)
debuggers exploratory testing with	调试器 利用之进行探索性测试
debugging after automated tests	自动化测试后的调试
debugging scope	调试范围
decision testing	判定测试
decomposing variable data	变量数据解耦
dedicated teams for nonfunctional testing	非功能测试的专门化团队
defect detection effectiveness (DDE)	缺陷检测有效性 (DDE)
defect removal efficiency (DRE)	缺陷移除有效性 (DRE)
defining boundary values	定义边界值
déjà vu heuristic	似曾相识的启发性
Deming W. Edwards	Deming W. Edwards
dependencies services	相依性 服务
dependency errors	相依性错误
deploying services with automation	自动化部署服务
deployment automation	自动化部署
deployment test clusters (services)	用于部署测试的服务器集群 (服务)
depth of inheritance tree (metric)	继承树深度 (指标)
descriptions for test patterns	测试模式的描述
descriptions of bugs (in bug reports)	bug 描述 (bug 报告中)
design (test pattern attribute)	设计 (测试模式属性)
design importance of	设计 其重要性
design patterns	设计模式
designing models	模型设计
designing test cases	测试用例设计
developing	开发
Development (SDE) discipline	开发 (SDE) 准则
development models	开发模型
device simulation framework	设备模拟框架
devices. See hardware	设备. 参见 硬件

diff utilities	diff 工具
Difficulty metric (Halstead)	“困难度”指标 (Halstead)
Director of Test	测试总监
Director of Test Excellence	卓越测试总监
Director Software Development	总监 软件开发工程师 (测试) 职称
Engineer in Test title	
disallow server cascading failure	“不允许服务器级联”失败
disciplines product engineering	准则 产品工程学
display contrast (accessibility)	显示对比度 (辅助功能)
distributed stress testing	分布式压力测试
DIT metric	DIT 指标
diversity of Microsoft portfolio	微软投资组合的多样性
divisions at Microsoft	微软的分部门
DMAIC model	DMAIC 模型
Do phase (PDCA cycle)	“执行”阶段 (PDCA 生命周期)
document comparison tools	文档比较工具
documentation of test cases. See entries at log; test cases	测试用例文档. 参见 日志 测试用例条目
documenting code changes. See source control	代码变更文档化. 参见 源代码控制
documenting with automated testing	使用自动化测试进行归档
dogfood clusters	Dogfood 群集
dogfooding	吃狗食
dogfooding (being users)	吃自己的狗食 (作为用户)
downlevel browser experience	低版本浏览器体验
DRE (defect removal efficiency)	DRE (缺陷移除有效性)
duplicate (bug resolution value)	重复 (bug 解决状态)
duplicate bugs	重复的 bug
E&D (Entertainment and Devices Division)	E&D (娱乐和家用设备部门)
each choice (EC) approach	逐个选择 (EC) 途径 组合测试
combinatorial testing	
ease of use bug tracking system	易用性 bug 跟踪系统
eating our dogfood	“自食狗食”
ECP (equivalence class partitioning)	ECP (等价类划分)
edges (control node graphs)	边 (控制结点图)
education strategy	教育战略
EE (Engineering Excellence) group	EE (卓越工程) 部门
effectiveness of code reviews	代码复审的有效性 度量
measuring	
elements of testing	测试元素
email discussions in bug reports	bug 报告中的电子邮件讨论记录

emotional response	情绪化反应
emotional response from customers	来自用户的情绪化反应
employee orientation	职员导向
emulating services	仿真服务
ending state (model)	“终止”状态 (建模)
engineering career at Microsoft	微软中的工程职业
engineering careers	工程师职业
engineering disciplines	工程准则
Engineering Excellence (EE) group	卓越工程 (EE) 组
Engineering Excellence Forum	卓越工程论坛
engineering life cycles	工程生命周期
Engineering Management discipline	工程管理准则
engineering organizational models	工程组织模型
engineers types of Entertainment and Devices Division. See E&D features	工程师 职种 娱乐和家用设备部门. 参见 E&D
environment bug	环境 bug
environmental sensitivity of automated tests	自动化测试的环境敏感
equivalence class partitioning (ECP)	ECP (等价类划分)
estimating (code smell)	估计 (代码嗅探)
estimating test time	预计的测试时间
estimating test time examples for test patterns	测试时长估计 测试模式举例
exception handling block testing for	异常处理 为其进行的区块测试
execution phase	执行阶段
exit criteria for milestones	里程碑的退出标准
expected practices	期望实施的实践
experience quality	经验质量
expiration date set (metric)	过期日期集 (能力)
exploratory testing (ET)	探索式测试(ET)
exporting virtual machines	导出虚拟机
external user access bug tracking system	外部用户访问权限 bug 跟踪系统
Fagan inspections	Fagan 检视法
Fail Perfectly attribute (BVTs)	“完全失败”属性(BVTs)
Fail result (automated tests)	“失败”结果 (自动化测试)
failure analysis automatic	错误分析 自动化
failure count (test case metric)	失败计数 (测试用例指标)

failure criteria in test cases	测试用例判定失败之标准
failure databases	失败数据库
failure matching	失败匹配
false alarms (metric)	假值警报 (度量)
false positives	假阳性
falsification tests	篡改测试
fanin and fanout measurements	扇入/扇出度量
fast rollbacks with services	服务快速回滚
feature area bug	特性领域 bug
feature crews (Agile methodologies)	功能团队 (敏捷方法论)
features	特性
feedback systems. See customer feedback systems	反馈系统. 参见 用户反馈系统
Fiddler tool	Fiddler 工具
field replaceable units (FRUs)	现场可替换部件 (FRUs)
film industry product development as	胶片工业 产品开发作为
finite state machines (FSMs)	有穷状态自动机 (FSMs)
finite state models	有限形态模型
finite state models building	有穷状态模型 构建
fix if time (bug priority)	“若时间有余即修正”(bug 优先级)
fix number method (WER)	修正数目方法 (WER)
fixed (bug resolution value)	“已修正”(bug 解决状态值)
fixedconstant values	固定常量值
font size (accessibility)	字体尺寸 (辅助功能)
for nonfunctional testing	非功能性测试
for performance	性能的度量
for performance bottlenecks	性能瓶颈
for performance testing	用于性能测试
for quality of services (QoS)	服务重量的度量(QoS)
for services	针对服务的
for services testing	服务测试
for test automation	测试自动化
forgotten steps in test cases	测试用例中被遗忘的步骤
formal code reviews	正规代码复审
forums Microsoft	论坛 微软
forward thinking in testing	测试中的先思
foundation (platform) services	基础 (平台) 服务
frequency of release services	发布频率 服务
frequency of testing	测试频率
Friday the th bug	周五的第个 bug
Frink Lloyd	Frink Lloyd
frowny icon (Send a Smile)	“皱眉”小图标 (发送微笑程序)

FRUs (field replaceable units)	FRUs (现场可替换部件)
FSMs (finite state machines)	FSMs (有穷状态自动机)
full automated service deployments	完全自动化服务部署
fully automated tests. See automated tests	完全自动化测试. 参见 自动化测试
functional testing	功能测试
fuzz testing	模糊测试
fuzzy matching	模糊匹配
FxCop utility	FxCop 工具
game data collecting	游戏数据 收集
gatekeeper (checkin system)	把关人员 (签入系统)
gauntlet (checkin system)	阻止者 (签入系统)
General Manager of Test	测试部门总经理
getting started	开始进行
getting to done (Agile methodologies)	接近完成 (敏捷开发方法学)
gimmicks test techniques as	无聊的勾当 测试技术作为
glass box testing	玻璃箱测试
grammar models	语法模型
graph theory	图论
gray box testing	灰盒测试
Group Test Managers	测试部门经理
grouping bugs in bug reports	在 bug 报告中将 bug 分组
groups of variables in ECP	ECP 中的变量分组
Halo game	Halo 游戏
Halstead metrics	Halstead 指标
Halstead metrics	Halstead 度量
Halstead metrics	霍尔斯德指标
happy path testing	愉快路径 测试
hardcoded paths in tests	测试中的硬编码路径
help phase	帮助阶段
help phase (SEARCH test automation)	“帮助”阶段 (SEARCH 测试自动化)
helping testers team for	协助测试工程师 小组
heuristics for equivalence class partitioning	启发式等价类划分
hidden boundaries	隐性边界
hidden boundary conditions	隐性边界条件
highcontrast mode	高对比度显示模式
hiring testers at Microsoft	微软里聘用测试工程师
historical data in test time estimation	历史数据 在测试时长估计中
historical reference test case as	历史参考数据 测试用例作为

hotfixes	特制补丁
how found (bug report field)	“如何发现”(bug 报告项目)
how to measure performance	如何衡量性能
how to use metrics for	指标使用方法
humorous bugs	幽默的 bug
HyperV	HyperV
IAccessible interface	IAccessible 接口
IC Testers	IC 测试工程师
ICs (individual contributors)	ICs (个人贡献者)
identification number validation	身份证号码校验
imaging technology	镜像技术
impact	影响
impact (competency)	影响 (能力)
impact of bug on customer	bug 给用户造成的影响
importing virtual machines	导入虚拟机
in Agile methodologies	灵便性方法学
in automated tests	在自动化测试中
in test management	测试管理
in test management	测试管理
incubation	孵化
industry recruiting	社会招聘
industry recruiting	面向业界的招聘
influence (competency)	影响力 (能力)
informal code reviews	非正规的代码复审
initial build process about	构建的初始阶段 关于
initiation phase (stress testing)	“初始化”阶段 (压力测试)
innovation	创新
innovation in PC computing	PC 计算中的创新
inputs (test cases)	输入 (测试用例)
Inspect Objects tool	Inspect Objects 工具
installation testing	安装测试
INT environment	INT 环境
integrated services test	集成服务测试环境
environment	
integration testing (services)	集成测试 (服务)
interactions within systems	系统内交互
International Project Engineering (IPE)	国际项目工程 (IPE)
internationalization	国际化
Internet memo	Internet 备忘录
Internet services as Microsoft focus	作为微软用户关注的 Internet 服务
Internet Services Business Unit (ISBU)	Internet 服务业务部门 (ISBU)

interoperability of bug tracking system	bug 跟踪系统的互操作性
interpersonal awareness (competency)	人际关系意识 (胜任能力)
interpreting test case results introduction (test strategy attribute)	测试用例结果的解释 介绍 (测试战略属性)
invalid class data (ECP)	非法类、无效类数据 (ECP)
involvement with test automation	测试自动化的参与
IPE (International Project Engineering)	IPE (国际项目工程)
ISBU (Internet Services Business Unit)	ISBU (Internet 服务业务部门)
ISO program	ISO program
issue type (bug report field)	问题类型 (bug 报告项目)
iterations Agile methodologies	迭代 敏捷开发方法学
jargon in test cases	测试用例中的术语
JIT debuggers	实时调试器
job titles for software test engineers	软件测试工程师的职别
justintime (JIT) debuggers	实时调试器
key scenario (test strategy attributes)	关键方案 (测试策略属性)
keyboard accessibility. See accessibility testing	键盘访问性. 见访问性测试
keystrokes simulating	键击模拟
knowledge testability	知识的可测试性
largescale test automation	大规模自动化测试
largescale testing	大规模测试
Last Known Good (LKG) release	最近已知的成功发布
layered services	层次化的服务
Lead Software Development Engineering in leadership	领导软件开发工程 领导力
Leads. See SDET Leads	组长. 见开发测试组长
learning how to be	学习如何
legacy client bugs	客户遗留 bug
legal defense bug reports as	法律保护 bug 报告
Length metric (Halstead)	长度计量
length of program (lines of code)	程序长度 (代码行数)
libraries	库
libraries of applications for	兼容性程序库

compatibility

lifecycle bugs	生命周期 bug
lifetime of automated tests	自动化测试的寿命
limitations of test patterns	测试样式的局限
linearly independent basic paths	线性独立基本路径
lines of code (LOC)	代码行数(LOC)
Live Mail service	Live 邮件服务
Live Mesh	Live 网孔
LKG (Last Known Good) release	最近已知的成功发布
load tests . See also performance	负荷测试也见性能
log file parsers	日志文件分析器
log files generated with test automation	自动化测试生成的日志文件
longhaul tests	“艰巨”测试
lookandfeel testing	“观察和感觉”测试
loop structures	循环结构
loosely coupled services	松散配对的服务
love bug	爱情 bug
low resource testing	低资源测试
lowresource and capacity testing	低资源和容量测试
Luhn formula	Luhn 公式
machine roles	机器的角色
machine virtualization	机器模拟
maintainability of source code complexity and	代码的可维护性和复杂度
maintainability testing	可维护性测试
managed code analysis	托管代码分析
managed code test attributes	托管代码测试属性
management career paths	管理者的职业发展道路
managing bugs	管理 bug
managing failures during tests	在测试中管理失败
managing test cases. See test cases	管理测试用例. 见测试用例
mandatory practices	必须的实践
manual testing . See also exploratory testing	手工测试 也见探索测试
mashups	信息集合
matching failures	失败比较
MBD (Microsoft Business Division)	MBD(微软企业部门)
MBT. See modelbased testing	MBT. 见基于模型的测试

mean time between failure	失败之余的测试
(MTBF) testing	
measuring	度量
measuring code complexity. See	衡量代码的复杂度
code complexity	
measuring performance	性能的衡量
memory usage attribute (stress tests)	内存使用属性
message loops	信息环
Metrics	度量
metrics for services	服务的指标
Microsoft Active Accessibility (MSAA)	Microsoft Active Accessibility (MSAA)
Microsoft Application Verifier tool	微软应用程序验证工具
Microsoft CIS (Cloud Infrastructure Services)	微软 CIS (云构造服务)
Microsoft OneNote customer connections	Microsoft OneNote 客户联系
Microsoft Passport parental controlled	Microsoft Passport 家长控制
Microsoft services strategy	Microsoft 服务战略
Microsoft Test Leadership Team (MSTLT)	微软测试领导组
Microsoft Tester Center	微软测试中心
Microsoft UI Automation framework	微软界面自动化框架
Microsoft Visual Studio Spec Explorer for	Microsoft Visual Studio 样式书浏览器
Microsoft Visual Studio Team Foundation	Microsoft Visual Studio Team Foundation
microsoft.public.* newsgroups	microsoft.public.* 新闻组
Microsoftspeak for	Microsoft 说话
milestone criteria	里程碑完成条件
milk carton bug	牛奶盒 bug
miscellaneous	杂项
missing steps in test cases	测试用例中缺少的步骤
mission statements Microsoft	微软的任务声明
mistakes in test cases	测试用例中的错误
mixed mode service upgrades	混合模式服务升级
mod checksum algorithm	模十检验和算法
modelbased testing (MBT)	基于模型的测试 (MBT)
modeling control flow	建模控制流程
modeling threats	建模的风险

modeling without testing	没有测试的建模
models for engineering workforce	工程劳动力的模型
models for software development	软件开发的模型
monitoring code changes (churn)	监督代码改变
monitoring code review effectiveness	监督代码审查的效率
monitoring performance	监督性能
monkey testing	猴子测试
mouse clicks simulating mouse target size (accessibility)	鼠标点击 模拟 鼠标目标尺寸(可访问性)
movie industry product development as	电影行业产品开发
moving from SDEs to SDETs	从软件开发工程师 (SDE) 转为软件 开发测试工程师 (SDET)
moving quality upstream	逆流提高质量
MQ (quality milestone)	MQ (质量里程碑)
MSAA (Microsoft Accessibility)	MSAA (Microsoft Accessibility)
MsaaVerify tool	MsaaVerify 工具
MSTLT (Microsoft Test Leadership Team)	MSTLT (微软测试领导组)
MTBF testing	MTBF 测试
multiclient stress tests	多客户端压力测试
multiple bugs in single report	单个报告中多个 bug
must fix (bug priority)	必须修复 (bug 优先级)
names for software test positions	软件测试职位的名称
names for test patterns	测试模式的名称
naming service releases	服务发行的命名
native code analysis	本机代码分析
negative testing	负面测试
NEO (New Employee Orientation)	NEO(新员工定位)
Net Promoter score	净推荐值
network topology testing	网络拓扑测试
newsgroups Microsoft	微软新闻组
nodes (control flow graphs)	节点 (控制流图)
known failure attribute (stress tests)	不明错误属性 (压力测试)
nonfunctional (behavioral) testing	非功能 (行为) 测试
nonfunctional testing	非功能测试
not repro (bug resolution value)	不可重现 (纠错方式值)

notification of bug assignment	缺陷分配通知
number of (code churn)	(代码改动)数量
number of passes or failures (test case metric)	通过或失败数量 (测试用例指标)
number of tests	测试的数量
nwise testing	n 维测试
object model	对象模型
objectoriented metrics	面向对象的度量
observable testability	可观察的测试性
Office Online	Office 在线
Office Online newsgroups	Office 在线新闻组
Office Shared Services (OSS) team	Office 共享服务 (OSS) 团队
OLSB (Online Live Small Business)	OLSB (在线小型企业)
on defect detection. See DDE	缺陷检测的度量. 见 DDE
on emotional response	情绪化反应的度量
onebox test platform (services)	单机测试平台 (服务)
OneNote customer connections	OneNote 用户联系
online services. See services	在线服务参见服务
open development	开放开发
open office space	开放办公空间
operating systems. See Windows	操作系统参见 Windows 操作系统
operating systems	
Operations (Ops) discipline	操作规则
operators in programs counting.	程序中的操作符计数参见霍尔斯特德指标
See Halstead metrics	
oracle (test pattern attribute)	神谕 (测试模式属性)
organization of engineering	工程团队的组织
workforce	
orientation for new employees	新员工指导
orthogonal arrays (OA) approach	正交数组 (OA) 方法组合测试
combinatorial testing	
OSS (Office Shared Services) team	OSS (Office 共享服务) 团队
outdated test cases	过时的测试用例
output matrix randomization (PICT tool)	输出矩阵随机(PICT 工具)
overgeneralization of variable data	变量数据过分概括
ownership of quality	质量责任
packaged product vs.	打包产品
packaged product vs. services	包装产品与服务
page load time metrics	页加载时间指标
page weight (metric)	页重 (指标)
pair programming	结对编程
pair testing	结对测试

pairwise analysis	结对分析
parameter interaction testing. See combinatorial analysis	交互参数测试参见组合分析
parental controlled with WLID	使用 WLID 控制
Pareto principle	帕累托原则
Pareto Vilfredo	帕累托·维尔弗雷多
parsing automatic test logs	解析自动化测试日志
partial production upgrades	部分产品升级服务
services	
partitioning data into classes. See decomposing variable data (ECP)	将数据分类参见分解变量数据 (ECP)
Partner SDETs	SDET 合伙人
Partner Software Development Engineer in Test titles	软件开发测试工程师合伙人职称
pass count (test case metric)	通过数 (测试用例指标)
pass rate (test case metric)	通过率 (测试用例指标)
Pass result (automated tests)	通过结果 (自动化测试)
pass/fail criteria in test cases	测试用例的通过/失败标准
passion for quality (competency)	对质量的热情 (能力)
path testing. See basis path testing	路径测试参见基本路径测试
patternsbased testing approach	基于模式的测试方法
PC computing innovations waves of	个人电脑计算创新浪潮
PDCA cycle	PDCA 周期
percentage of false alarms (metric)	假警报的比例 (指标)
percentage of tickets resolved (metric)	票证解决的比例 (指标)
perception of quality	质量感知
perf and scale clusters	性能和规模簇
Perfmon.exe utility	perfmon.exe 应用程序
performance counters	性能计数器
performance measurement	性能度量
performance test metrics	性能测试指标
performance testing	性能测试
personas for accessibility testing	辅助功能测试的角色
pesticide paradox	杀虫剂悖论
Petri nets	Petri 网络
PICT tool	PICT 工具
pitfalls with test patterns	测试模式的陷阱
Plan phase (PDCA cycle)	计划阶段 (PDCA 周期)
planning	计划

Platform Products and Services Division. See PSD	平台产品和服务部参见 PSD
platform services	平台服务
platform vs. toplevel	平台 顶层
platforms for test automation	测试自动化的平台
play production shipping products as	玩产品发布产品
point of involvement with test automation	测试自动化介入时机
portability testing	可移植性测试
postbuild testing	建造后测试
postponed bugs	推迟（修复）的缺陷
power growth and	能源增长
practical baseline path technique	合理的基准线路径技术
practical considerations	从实际考虑
prebuild testing	建造前测试
predicted results (test case attribute)	预期结果（测试用例属性）
predicting quality perception	预期质量感知
PREfast tool	PREfast 工具
Principle SDETs	首席 SDET
Principle Software Development Engineer in Test titles	首席软件开发测试工程师职称
Principle Test Managers	首席测试经理
Print Verifier	打印验证
prioritizing bugs	把缺陷区分优先次序
proactive approach to testing	积极测试方法
problem (test pattern attribute)	问题（测试模式属性）
Problem Reports and Solutions panel	问题报告和解决面板
problem solving competency	问题解决能力
process improvement	过程改进
processing power for services	服务的处理能力
processing power requirements	处理能力需求
product code. See entries at code	产品代码参见代码条目
product engineering disciplines	产品工程职位
product releases. See releases	产品发布参见发布
Product Studio	Product Studio 产品工作室
product support	产品支持
product teams	产品团队
Product Unit Manager (PUM) model	产品部门经理（PUM）模型
production testing against	测试产品

program decisions counting. See cyclomatic complexity measuring	程序决策计数。参见圈复杂度衡量
program length measuring	程序长度衡量
Program Management (PM) discipline	程序管理 (PM) 职位
programmatic accessibility	编程辅助功能
progress tracking	进度跟踪
Project Atlas	项目自动测试和负载分析系统
project management	项目管理
prototypes	样品
PSD (Platform Products and Services Division)	PSD (平台产品和服务部)
PUM (Product Unit Manager) model	PUM (产品部门经理) 模型
purpose (test case attribute)	目的 (测试用例属性)
QA (quality assurance)	QA (质量管理)
QoS (quality of service) programs	Qos (服务质量) 计划
quality assurance (QA)	质量管理
quality cost of	质量代价
quality culture	质量文化
quality gates	质量把关
quality metrics	质量指标
quality milestone	质量里程碑
quality of service (QoS) programs	服务质量(QoS)计划
quality passion for (competency)	对质量的热情 (能力)
quality perception	质量感知
quality service	质量服务
quality tests. See nonfunctional testing	质量测试参见非功能测试
quantifying	量化
Quests	请求
quotas for finding bugs	寻找缺陷配额
rack units (rack SKUs)	机架单位 (机架的 SKU)
random modelbased testing	基于随机模型的测试
random selection combinatorial testing	随机选择组合测试
random walk traversals	随机游动遍历
random walk traversals (graph theory)	随机游动遍历 (图论)
ranges of values in ECP	ECP 的值范围
RCA (root cause analysis)	RCA (根本原因分析)
reactive approach to testing	被动方法测试
reasons for code change	代码变化原因记录
documenting	文档

reasons for code changes	代码变化原因
recruiting testers	招聘测试员
RedDog. See CIS (Cloud Infrastructure Services)	RedDog （云基础设施服务）
regression tests	回归测试
regular expressions	正则表达式
Rehabilitation Act Section	康复法案 款
related test patterns identifying	相关测试模式确定
release frequency and naming	发行频率和命名
release/free builds	发布或自由构建
Releases	发布
reliability of bug tracking system	bug 跟踪系统的可靠性
reliability testing	可靠性测试
repeatability test cases	重复性测试用例
repetition testing	重复测试
reporting phase	报告阶段
reporting phase (SEARCH test automation)	报告阶段(搜索自动化测试)
reporting user data. See customer feedback systems	报告用户数据。 用户反馈系统。
reproduction steps (repro steps)	重现步骤
Research discipline	研究学科
resolution (in bug reports)	解决方法(bug 报告中)
resolved (bug status)	解决了的(bug 状态)
resource utilization	资源利用
response from customers. See customer feedback systems	用户的响应 反馈系统
responsibility for managing	管理的责任
responsiveness measurements	响应测量
result types for automated testing	自动化测试的结果类型
reusing code	重用代码
reviewing automated test results	自动化测试结果审核
risk analysis modeling	风险分析模型
risk estimation with churn metrics	有流失率指标的风险评估
risk management with services deployment	有服务部署的风险管理
risk with code complexity	代码复杂性风险
riskbased testing	基于风险的测试
role of testing	测试职责
rolling builds	滚动建造
rolling upgrades services	滚动升级服务
round trip analysis (metric)	往返分析（指标）

run infinitely attribute (stress tests)	无限运行属性(压力测试)
running the tests	运行测试
S+S testing approaches	S+S 测试方法
S+S testing techniques	S+S 测试技术
S+S. See Software Plus Services (S+S)	S+S 软件加服务
SaaS (software as a service) . See also Software Plus Services (S+S)	S+S 软件加服务
sanitizing data before testing	测试前数据消毒
scalability testing	扩展性测试
scale out (processing power)	超过尺寸范围 (处理能力)
scale up (system data)	扩大 (系统数据)
scenario voting	场景投票
scheduling	调度
SCM. See source control	代码控制
scope of debugging	测试范围
scripted tests	脚本化测试
scripted tests code coverage of	脚本测试代码覆盖率
SDE. See Development (SDE)	软件开发工程师
discipline; Test	
SDET IC	软件开发测试工程师独立贡献者 (SDET IC)
SDET IC	SDET 独立贡献者
SDET Leads	SDET 主管
SDET Managers	SDET 经理
SDETs (Software Development Engineers in Test)	软件开发测试工程师
SEARCH acronym for test automation	搜索自动化测试缩写
Section (Rehabilitation Act)	款 (康复法案)
Security	安全
security testing	安全测试
selfhost builds	自己运营的构建
selftest build	自己检测的构建
selftoast builds	自己烤糊的构建
semiautomated tests	半自动化测试
Send a Smile program	发送一个微笑计划
Senior SDET Leads	资深 SDET 主管
Senior SDET Manager	资深 SDET 经理
Senior SDETs	资深 SDET
Senior Software Development Engineer in Test	资深软件开发测试工程师
Server (TFS)	服务器(TFS)

server builds	服务器建造
servers building services on	服务器构建服务
service groups	服务组
services and processing power	服务和处理能力
Services memo	服务备注
services metrics for	服务指标
setup phase	安装阶段
setup phase	设置阶段
setup phase (SEARCH test automation)	安装阶段
Seven Bridges of Königsberg problem	Königsberg 七座桥问题
severity bug	严重性 bug
shared	共享的
shared libraries	共享库
Shared Team model	共享的组模型
shared teams	共享团队
shared test clusters	共享测试群
sharing test tools	共享测试工具
Shewhart cycle	修哈特循环
shipping software	发行软件
shirts ordering new	衬衫订购新的
shortest path traversal (graph theory)	最短路经遍历
should fix (bug priority)	应该修的 (bug 优先级)
shrinkwrap software	用收缩胶模包装的软件
simple testability	简单测试
simplicity. See code complexity	简单。代码复杂性
simplified baseline path technique	简化的基线路径技术
simplified control flow diagrams (CFDs)	简化的控制流模型
single fault assumption	单一故障假设
Six Sigma program	六西格玛项目
size issues (accessibility)	大小问题 (辅助功能)
size of Microsoft engineering workforce	Microsoft 工程师人员规模
Skip result (automated tests)	跳过结果 (自动化的测试)
SMEs as testers	学科专家测试
smiley icon (Send a Smile)	微笑的图标(发送一个微笑)
smoke alarm metrics	烟雾警报度量
smoke alarm metrics	冒烟报警指标
smoke tests	冒烟测试
snapshots of code	代码的快照

SOCK mnemonic for testability	套接字记忆测试
software as services. See services	服务式软件
software design importance of	软件设计
Software Development Engineer in	软件开发测试工程师
Test Manager title	经理级别
Software Development Engineer in	软件开发测试工程师
Test titles	级别
software development models	软件开发模型
software engineering at Microsoft	Microsoft 的软件工程
software features	软件特性
software libraries	软件库
Software Plus Services (S+S) . See also services	软件加服务
software reliability. See reliability testing	软件可靠性 可靠性测试
software test engineers . See also titles for software test engineers	软件测试工程师
sound features accessibility of source (bug report field)	稳定特性辅助功能
source control	来源(bug 汇报字段)
Spec Explorer tool	源代码控制
special values in ECP	游览说明书工具
specifications for test design	ECP 的特殊值
specified in bug reports	测试设计说明书
spiral model	bug 报告中言明的指派
SQEs (software quality engineers)	螺旋模型
standalone and layered services	软件质量工程师
standalone applications	单机和多层服务
standalone services	单机应用
stapler stress	单机服务
starting state (model)	订书机压力
starting the test process	开始日期
statebased models. See modelbased testing(MBT)	启动测试进程
stateless vs. stateful services	基于状态的模型
statement testing	无状态的 有状态的服务
statement vs. block coverage	语句测试
statement vs. block coverage	语句和代码块的覆盖率
static analysis	表达式覆盖率 vs. 区块覆盖率
status bug	静态分析
Step attribute	状态 bug
steps in test cases	步骤属性
STEs (Software Test Engineers)	测试用例步骤
stopwatch testing	软件测试工程师
	秒表测试

strategic insight (competence)	战略眼光（能力）
strategy test	战略测试
stress testing	压力测试
structural testing	结构化测试
subject matter experts as testers	学科专家测试
support for test automation	测试自动化支持
support product	支持产品
SupportFile attribute	SupportFile 属性
Surface	表面
switch/case statement testing	switch/case 语句测试
syntax elements in programs	程序中的语法元素
counting. See syntax errors	语法错误
systematic evaluation approaches	系统评价方法
systems interactions	系统交互
systems test. See test tools and systems	系统测试
systemwide accessibility settings	全系统无障碍设置
TAG (Test Architect Group)	测试设计组
TAs. See Test Architects	测试架构设计师
TCMs (test case managers)	测试用例管理
TDSs (test design specifications)	测试设计说明书
Team Foundation Server (TFS)	
team organization	团队组织
technical excellence (competency)	技术卓越（能力）
Technical Fellows	技术院士
Techniques as gimmicks	技术诀窍
templates for sharing test patterns	测试模式共享模板
Test (SDET) discipline . See also SDETs	测试开发工程师
Test Architect	测试架构师
test automation	自动化测试
Test Broadly Not Deeply attribute (BVTs)	广度而非深度的测试
test case design	测试用例设计
test case managers (TCMs)	测试用例管理
test cases	测试用例
test cases (continued)	测试用例（续）
test cleanup	清除测试设置
test clusters	测试群集服务器
test code analysis	测试代码分析
test code snapshots	测试代码快照

test collateral	测试素材
test controllers	测试控制程序
test coverage. See code coverage	测试覆盖率。参见代码覆盖率
test data creating with grammar models	使用语法模型创建的测试数据
test deliverables (test strategy attributes)	可交付的测试结果（测试策略属性）
test design	测试设计
test design importance of	测试设计的重要性
test design specifications (TDSs)	测试设计文档（TDSs）
test environment	测试环境
test environment for services	针对服务的测试环境
test excellence	卓越测试
test flags	测试标志
test frequency	测试频率
test harnesses	测试工具
test innovation	测试的创新
test leadership	测试领导团队
test logs (automated testing)	测试日志（自动化测试）
test management	测试管理
test matrix for services testing	服务测试的测试指标
test oracles	测试名言
test pass defined	测试通过已定义的
test patterns	测试模式
test points	测试重点
test run defined	测试运行已定义的
test scenarios	测试场情
test strategies	测试策略
test suite defined	测试套件已定义的
test therapists	测试分析人员
test time estimating	测试时间预计的
test tools and systems	测试工具和自动化系统。参见自动化
automation. See test automation	测试
testability	可测试性
TestCleanup attribute	清除测试属性
Tester Center	测试人员中心
tester DNA	测试人员 DNA
testing	测试
testing against production	对产品测试
testing approaches	客户支持
Testing at Microsoft for SDETs class	SDETs 课程“在微软做测试”
testing coverage	测试覆盖率
testing daily builds	每日版本测试

testing future of	测试未来
testing good and bad	正面和负面测试
testing S+S	测试 S+S
testing techniques	测试技术
testing techniques as gimmicks	测试技巧
testing the tests	对测试进行测试
testing vs. quality	测试与质量
testing with model	用模型测试
testing with models. See modelbased testing(MBT)	使用测试模型进行测试。参见基于模 型的测试 (MBT)
testing with virtualization	在虚拟化环境中测试
testing; stress testing	测试; 压力测试
TestInitialize attribute	测试初始化属性
TestMethod attribute	测试方法属性
text matrix for automated testing	自动化测试的测试指标
ThinkWeek	思考周
threat modeling	威胁建模
threshold method (WER)	临界值方法
tidal wave bug	波动的 bug
tightly coupled services	高耦合的服务
time for code reviews monitoring	审阅代码的时间监控
time investment. See scheduling	投入的时间。参见制定计划
time to detection (metric)	
time to document	写文档的时间
time to market services	上市的时间服务
time to resolution (metric)	解析 (指标) 的时间
timetomarket considerations	上市时间考虑
tips for modeling	建模的提示
titles for software test engineers	软件测试工程师的职称
titles of bugs (in bug reports)	Bug 的标题 (在 bug 报告中)
tool sharing	工具共享
tools for accessibility technology	用于辅助功能技术的工具
tools for in practice	工具 实践中采用的
toplevel services	顶级服务
topology testing	拓扑测试
tracking and interpreting results (metrics)	跟踪和解释运行结果 (指标)
tracking bugs. See managing bugs	跟踪 bug。参见 bug 管理
tracking changes in	跟踪修改
tracking changes in	在某范围里跟踪变更
tracking code changes. See source control	跟踪代码修改。参见源代码控制
tracking code review data	跟踪有关代码审阅的数据
tracking test cases	跟踪测试用例

tracking test progress	跟踪测试进度
traditional models	传统模型
training as SDETs	SDETs 培训
training strategy in test design	测试设计的培训策略
transitions (in models)	转换（模型间的）
trends in test failures analyzing	测试失败的趋势分析
triad (Test Development Program Management)	三元体系（测试开发项目管理）
triage	三方会审
triage for managing bugs	Bug 管理的类选法
triangle simulation (Weinberg's triangle)	模拟三角形（Weinberg's 三角形）
Trustworthy attribute (BVTs)	可信性属性（BVTs）
tsunami effect	海啸效应
usability labs	易用性实验室
usability testing	可用性测试
USB cart of death	USB 死亡之车
user interface automation	用户接口自动化
using data effectively	有效的使用数据
using ECP tables with	与 ECP 表合用
using for failure matching	用作失败比较
using for failure matching	用于失败匹配
using test patterns	应用测试模式
using triangle simulations	使用三角模拟
vs. SaaS (software as a service)	测试方法
war room	战争室
watching customers. See CEIP (Customer Experience Improvement Plan)	观察客户。参见 CEIP（客户体验改进计划）
Windows Error Reporting	Windows 错误报告
workflow	工作流

