HZ BOOKS
华章科技
Microsoft

《代码大全》姊妹篇，资深软件开发专家30余年工作经验结晶，微软公司软件工程师必读之书，被誉为“软件行业的财富”

从软件开发流程、技术、方法、项目管理、团队管理、人际沟通等多角度总结出90余个具有代表性的问题并给出了解决方案，值得所有软件工程师研读

华章程序员书库

I. M. Wright's "Hard Code"
A Decade of Hard-Won Lessons from Microsoft
Second Edition

代码之殇

（原书第2版）

（美）Eric Brechner 著
林锋 译

机械工业出版社
China Machine Press

Eric是我个人崇拜的英雄，很大程度上是因为他长期以来一直代表着开发社区的一种声音。

—— Chad Dellinger，微软公司企业架构师

软件工程师很容易就会迷失在代码中，更糟糕的是，甚至会迷失在过程中。因此迫切需要Eric在本书中提出的实用建议。

—— David Greenspoon，微软公司技术战略总经理

"I. M. Wright"写的关于开发时间表的文章真是太棒了！它在我所参与的基础项目上同样适用。

—— Ian Puttergill，项目经理

我们真的很喜欢这些栏目。它们不仅实用，而且很全面！我喜欢它们的另外一个原因是，当我在指导初级开发人员的时候，我可以把这些栏目推荐给他们；他们也会记住这些栏目，因为它们都是那么有趣。

—— Malia Ansberry，高级软件工程师

Eric，干得好！我觉得你在这个专栏中说得非常中肯。我想，该向管理者传递这样的信息："不要害怕尝试"。事情的真实情况跟理想化的理论之间差别是非常大的。

—— Bob Fries，合作伙伴开发经理

Eric，你那篇关于死亡行军的栏目来得正是时候，我们正打算在未来几周内开会讨论功能削减的事情呢！我们以前付出很大代价才学到的那些教训，不知怎么回事，总是很容易就忘记了；Eric的栏目对大家起到了很好的提醒作用。

—— Bruce Morgan，首席开

上架指导：计算机/程序设计
ISBN 978-7-111-41682-1
定价：79.00元

Microsoft
www.microsoft.com/mspress

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

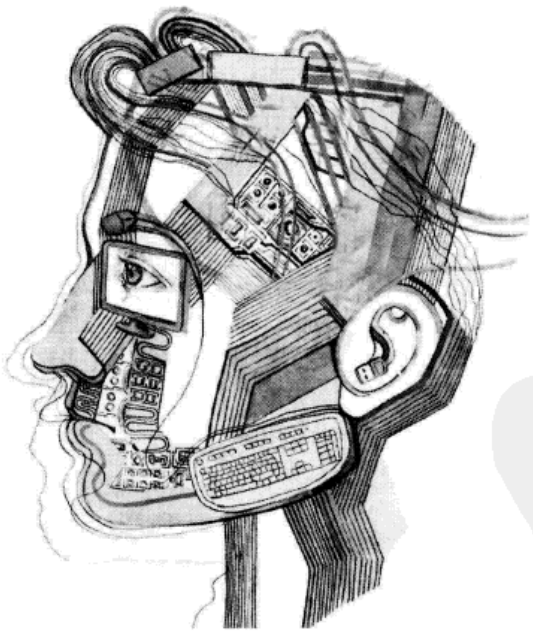
读者信箱：hzjsj@hzbook.com
华章网站：www.hzbook.com
网上购书：www.china-pub.com

I. M. Wright's "Hard Code"
A Decade of Hard-Won Lessons from Microsoft
Second Edition

代码之殇

（原书第2版）

（美）Eric Brechner 著
林锋 译



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

代码之殇（原书第2版）/（美）布莱什纳（Brechner, E.）著；林锋译. —北京：机械工业出版社，2013.4
（华章程序员书库）
书名原文：I. M. Wright's "Hard Code"：A Decade of Hard-Won Lessons from Microsoft，Second Edition
ISBN 978-7-111-41682-1
Ⅰ. 代… Ⅱ. ①布… ②林… Ⅲ. 软件开发 Ⅳ. TP311.52
中国版本图书馆CIP数据核字（2013）第039323号

本书版权登记号：图字：01-2011-7466

本书是《代码大全》的姊妹篇，资深软件开发专家30余年工作经验结晶，被誉为"软件行业的财富"，微软公司软件工程师必读之书。它从软件开发流程、技术、方法、项目管理、团队管理、人际沟通等多角度总结出90余个具有代表性的问题（大多数问题可能会给公司或软件项目带来毁灭性灾难），并给出了问题的解决方案和最佳实践，值得所有软件工程师和项目管理者研读。

本书将这90余个问题分为10章：第1章讨论如何通过管理风险、范围和沟通来保障项目按时完成；第2章介绍消除经验主义的大量过程改进的方法与技巧；第3章讨论消除低效率的策略；第4章主要讨论开发者与其他工种之间的关系；第5章重点阐释软件质量问题；第6章解析软件设计的基本原理和错综复杂的本性；第7章探讨如何规划职业生涯；第8章分析工作与生活中存在的缺点的原因与纠正措施；第9章讨论如何进行有效管理；第10章分析如何成功应对一个软件业务所面临的挑战。

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：秦 健
北京市荣盛彩色印刷有限公司印刷
2013年5月第1版第1次印刷
186mm×240mm · 20.75印张
标准书号：ISBN 978-7-111-41682-1
定 价：79.00元

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

本书赞誉

任何大型组织都会有成为自身文化牺牲品的危险。关于业务模式及行为准则的神话想当然地成为金科玉律。任何组织都会存在这种倾向，但对于需要不断创新才能兴旺的技术公司来说，这却是致命杀手。Eric Brechner 做了件难以置信的事——他亮出了手术刀，深度剖析了存在于组织中的这种谬误。他毫不客气地亮出了利剑——往往一针见血。尽管有一些隐语和例子对于微软内部的员工更具吸引力，但他的智慧和至理名言，大都可以成为整个软件行业的财富。

——Clemens Szyperski，首席架构师

I. M. Wright 写的关于开发时间表的文章真的太棒了！它在我所属项目组参与的基础项目上同样适用。

——Ian Puttergill，项目经理

你不打算遇到死亡威胁这样的事，是吧？

——Tracey Meltzer，高级测试主管

这很可笑——很坦率地说，这类十足的谬论很危险。

——Chad Dellinger，企业架构师

Eric 是我个人崇拜的英雄——很大程度上是因为他长期以来一直代表着开发社区的一种声音。

——Chad Dellinger，企业架构师

软件工程师很容易就会迷失在代码中，更糟糕的是，甚至会迷失在过程中。此时迫切需要 Eric 在本书中提出的实用建议。

——David Greenspoon，总经理

我刚刚读完这个月的栏目……我不得不指出，这是我第一次认为你正在推行一个对公司完全错误并且带有灾难性的想法。

——David Greenspoon，总经理

你太有才了，Eric。几个月之前，我跟产品单元经理以及一些开发主管恰恰进行过一次这样的对话。好主意。

——Scott Cottrille，首席开发经理

我们真的很喜欢这些栏目。它们不仅实用，而且很全面！我喜欢它们的另外一个原因是，当我在指导初级开发人员的时候，我可以把这些栏目推荐给他们；他们也会记住这些栏目，因为它们都是那么有趣。

——Malia Ansberry，高级软件工程师

Eric，干得好！我觉得你在这个栏目中说得非常中肯。我想，该给管理者传递这样的信息——“不要害怕尝试。”事情的真实情况跟理想化的理论之间差别是非常大的。

——Bob Fries，合作伙伴开发经理

我只是想让你知道我有多喜欢你写的文章——它们充满智慧，见解深刻，你还神奇地把本来很严肃的问题变得如此有趣（采用的方法很不错）。

——Niels Hilmar Madsen，开发者传教士

你那篇关于死亡行军的栏目来得正是时候。我们正打算在未来几周内开会讨论功能削减的事情呢！那些我们以前付出很大代价才学到的教训，不知怎么回事，总是很容易就忘记了；你的栏目对大家起到了很好的提醒作用。

——Bruce Morgan，首席开发经理

我想让你知道的是，我真的很喜欢并感谢你在 EE 站点上发表的所有文章。不过，直到今天，当我读了“停止写规范书”这个栏目之后，我不得不说，我强烈不同意你的观点。

——Cheng Wei，项目经理

你到底是谁？你跟 Eric Brechner 都做了些什么？

——Olof Hellman，软件工程师

Eric，我刚刚读完你写的那篇叫“不可攀比”的文章。你不知道我有多感激你！你实际上把这个观点传递给了公司里面成千上万的人……你致力于正确领导和管理团队，并把其中的奥秘跟大家分享，对于你的这种热情我真的非常欣赏！

——Teresa Horgan，商务项目经理

译 者 序

没想过真的可以翻译完成一本书，但天性赋有一颗充满好奇的心，闲时，网游、肥皂剧又非我所好，就喜读读书，摆弄摆弄技术，既然喜欢，与其像他人在网游中找乐，不如一直好奇着读书倒腾技术为乐，而且总有所得。早年看侯捷先生的译作，我就在想，如果有一天我也能翻译，既可借翻译之机广为涉猎，又可担当传播学识、价值观的角色，岂不幸甚？今天，终于有了这样一个机会。

IT 一词，通常让人联想到的是那些技术极客——一群苦大仇深的码农形象，就像本书的书名一样被人误解。其实，软件开发更体现的是一种团队合作、企业文化乃至一人为人处世的态度。本书不是高屋建瓴、看不见摸不着的软件工程理论教科书。作者是微软的一名资深项目经理，本书内容是他从日常工作生活的点滴体会中总结出来的。他不仅把软件工程的一些原理技巧在实践中加以诠释，同时阐述了如何处理团队成员之间的关系，如何面对职业生涯规划，如何树立正确的工作生活态度。

感谢我的挚友，曾晋瑞。他是一个乐于工作，享受生活，激情四射的人；凡事严谨，追求完美，理想主义的人。当大家都在追问中国如何才能出个乔布斯的时候，我只感叹中国没这土壤，否则曾晋瑞就是。当我还未正式翻译此书的时候，他就对我说："林锋，你翻译到哪了，你这书可要好好翻译，起码得从头到尾修改十遍。"我只能表示!!! -_-。他在本书的翻译过程中给予我莫大帮助，一旦我遇到晦涩难懂的内容，只要我求助于他，他就能从亢奋的工作状态中抽出时间，聊上好长时间；也会跑到我家，花上半天的时间，为的只是讨论某段文字的确切意思，甚至用哪个词更为精妙。这里再次表示感谢！

这是本人翻译的第一本书，且原书很多专业术语出自微软内部，译文中可能存在不少错误，误漏之处，请不吝指正，想必，这样的积累日后定能为读者带来更高质量的译本。

最后，分享一句译完本书的个人感悟：你的理想不用很具体，只要你有那份热忱，即使你看不清未来是什么，你也会像我一样，向最初我的景仰人物侯捷先生迈进一步。

如愿与本书译者进一步沟通，请 E-mail：xlfg@ yeah. net，或新浪微博：@ 大黄蜂的思索。

林锋

序

如果你想了解礼仪，你会直接光顾网站 Miss Manners；如果你在恋爱上遇到什么麻烦，你可能就会浏览论坛 Dear Abby；如果你想了解微软到底出了什么状况以及一位名叫 I. M. Wright 的大个头到底是怎么办事的，那么这本书就是为你而写的。I. M. Wright 就是在微软众所周知的 Eric Brechner。

软件开发是项颇具挑战性的工作。我已经把它当成一种开创性的团队运动，这项工作不仅需要你记住之前做过什么，同时要记住没干过什么。我在微软工作的那段时间，Eric 是我的知音。每当我受挫或失意的时候，通常不用我主动向他诉求，他似乎总知道该跟我讲些什么来帮助我；当我遇到难题需要帮助的时候，一个 I. M. Wright 专栏就会出现，它正是我所关心的问题之所需，这些问题在微软及其他软件开发机构都很常见。

一本《代码之殇》（Hard Code）在手就好比你办公室四周角落都呆着个 Eric。在应对变化时你遇到麻烦了吗？Eric 就是答案；你的团队缺乏斗志了吗？我笃定 Eric 对你总有好建议；质量问题困扰着你的代码吗？我知道 Eric 可以帮助你。通过使用笔名 I. M. Wright，Eric 以一种轻松愉快的方式为你解决软件开发问题，他在促使你思考的同时让你笑逐颜开。

但 Eric 并不只写一些你可能遇到的问题，他也萃取一些他所见到的公司的成功经验作为教程，这些公司开发并发布的产品及服务由全球数百万用户所使用。第 2 版囊括了丰富的最新建议及成功案例——它们很有价值，我经常将其作为我客户的必读之物。《代码之殇》是种珍宝，为人人书架之必备。

Mitch Lacey

本书顾问，微软前雇员，现供职于麦切·莱西联合有限公司

2011 年 5 月

前　言

献给当初对我说“为什么不由你来写”的人 Bill Bowlus。

献给当初对我说“好的，我帮你编辑一下”的人我的妻子。

你手上拿着的是一本关于最佳实务的书，它会比较乏味，但也许会有点吸引力，你能从中得到知识，读后甚至对你产生些许影响，但读起来肯定是干巴巴而无趣的。为什么这么说呢？

最佳实务的书是乏味的，因为这个“最佳”是跟具体的项目、具体的人、他们的目标以及偏好紧密相关的。一个实务是不是“最佳”，大家可能看法不一。作者必须把实务列举出来让读者自己选，并分析在什么时候、什么原因选择哪个方案作为“最佳”。但是这种做法是现实的、是非分明的，它会让人感到乏味和厌烦。通过多个案例的研究使原本模棱两可的概念泾渭分明，从而会使文字有味一些，但作者仍必须把选择的机会留给读者，否则作者就会显得傲慢、教条并且死板。

然而，人们喜欢看到傲慢、教条、死板的学者之间的针锋相对。大家喜欢引用学者们的观点片段，与朋友和同事一起讨论。为什么不将对这些最佳实务的争论作为观点栏目来发表呢？唯一的条件就是只要有人愿意将自己扮演成一个思想保守的傻瓜。

本书的由来

2001 年 4 月，在历经了 Bank Leumi、Jet Propulsion Laboratory、GRAFTEK、Silicon Graphics 及 Boeing 等公司总共 16 年的职业程序员生涯，并在微软做了 6 年的程序员和经理之后，我转而加入了微软内部的一个以在公司范围内传播最佳实务为职责的团队。当时这个组正在运作发行一个名叫《Interface》的月刊网络杂志的项目。它很有意义且富有知识性，但同样也是干巴巴而无趣的。我那时建议增加一个观点栏目。

我的上司 Bill Bowlus 建议由我来写。我拒绝了。作为一个半大孩子，我努力使自己成为一个协调员，撮合多方产生成果，而成为一个爱唠叨的实务学者会毁掉我的名誉和效力。因此，我当时的想法是说服一个大家公认的偏执的工程师来写，他可能是我在微软 6 年工作经历中接触过的一位特别固执的开发经理。

但 Bill 指出，我有 22 年的开发经验，4 年的开发管理经验，写作技巧也还行，而且有足够的态度来做这件事，我只需要放下自身的心理包袱。另外，其他的开发经理都忙于常规的工作，不可能每个月来为我们写一些个人观点。最后 Bill 和我想出了一个用笔名撰稿的点子，于是“代码之殇”（Hard Code）栏目诞生了。

从 2001 年 6 月开始，我使用“I. M. Wright，微软逍遥的开发经理”（I. M. Wright，Microsoft development manager at large）这个署名为微软的开发者和他们的经理写了 91 个“代码之殇”观点栏目。这些栏目的标签行都打上了“绝对诚实，直言不讳”（Brutally honest，no pulled pun-

ches）的标语。每个月，有成千上万的微软工程师和经理会阅读这些栏目。

前16个栏目在内部网络杂志《Interface》上发表了。这些栏目都是编辑（Mark Ashley和Liza White）给我分配的。我和《Interface》的美工Todd Timmcke还一起制作了作者的很多搞怪照片。当网络杂志停刊的时候，我才得以有喘息的机会，但也停止了写作。

14个月之后，在我们组的编辑Amy Hamilton（Blair）、Dia Reeves、Linda Caputo、Shannon Evans和Marc Wilson的帮助下，我又开始在内部站点上发表我的栏目。2006年11月，我将所有的栏目转移到一个内部的SharePoint博客上。

2007年春天，正当我打算度过几年前奖励给我的假期的时候，我现在的经理Cedric Coco给了我在休假期间将《代码之殇》出版成书的授权。而微软出版社的Ben Ryan也同意了。同年，本书第1版出版。

除了我已经提及的人，我还想感谢《Interface》的其他成员（Susan Fario、Bruce Fenske、Ann Hoegemeier、John Spilker和John Swenson），其他帮助本书出版的人（Suzanne Sowinska、Alex Blanton、Scott Berkun、Devon Musgrave和Valerie Wolley），支持我的管理层（Cedric Coco、Scott Charney和John Devaan），曾审核过我的栏目并提出过很多主题的团队成员（William Adams、Alan Auerbach、Adam Barr、Eric Bush、Scott Cheney、Jennifer Hamilton、Corey Ladas、David Norris、Bernie Thompson、James Waletzky、Don Willits和Mitch Wyle），以及写下第1版“前言”的Mike Zintel。关于第2版，我想重点提下本书的审核人员及长期读者（Adam Barr、Bill Hanlon、Bulent Elmaci，Clemens Szyperski、Curt Carpenter、David Anson、David Berg、David Norris、Eric Bush、Irada Sadykhova、James Waletzky、J. D. Meier、Jan Nelson、Jennifer Hamilton、Josh Lindquist、Kent Sullivan、Matt Ruhlen、Michael Hunter、Mitchell Wyle、Philip Su、Rahim Sidi、Robert Deupree（Jr.）、William Adams和William Lees）；还有James Waletzky，他在我休假的时候为我的读者写了两篇专栏文章；Adam Barr和Robert Deupree（Jr.），他们发动我为专栏记录了一段播客并发布出去；Devon Musgrave和Valerie Woolley，是他们让第2版顺利出版；我的上司（Peter Loforte和Curt Steeb）对我的努力给予了莫大的支持；Mitch为我写了第2版的“序”；以及我的妻子Karen，当我让编辑加入Xbox. com工作时，她接手了我的专栏编辑工作。

最后，我要感谢我才华出众的中学英语老师（Alan Shapiro），以及那些慷慨给予我反馈的读者们。尤其是，我还要感谢妻子Karen和儿子Alex和Peter，他们让我做任何事情都充满信心。

本书的读者对象

组成本书的91个观点栏目最初是写给微软的开发者及其经理看的，尽管它们也是我过去在软件行业6个不同的公司、32年的工作经验中提炼出来的。编辑和我一起修改了语言表达，注解了那些微软内部的特殊用语，使得本书适合于所有软件工程师和工程经理阅读。

我在这些栏目中表达的观点是我个人的，不代表我现在和以前任职过的任何一家公司，包括微软。我在栏目中的注解以及本简介中的言论同样都是我个人的，与任何公司无关。

本书的结构

根据主题的不同，我把所有栏目分成10章。前6章剖析了软件开发流程，接下来3章重点讨论人的问题，最后1章批判软件业务运转方式。解决这些问题的工具、技巧和建议遍布全书。本书的最后还附有术语表方便大家参考。

每一章的各个栏目均按照当初在微软内部发表的时间顺序排列。每章开头我都给出了一个简短的介绍，随后就是当初我以“I. M. Wright”笔名发表的栏目内容。当将其编辑成书的时候，我还适时在栏目中加上了“作者注”，以解释微软的术语，提供更新内容或者额外的背景知识。

编辑和我尽力保持原有栏目的完整性。我们做的工作仅仅是纠正语法和内部引用。称得上改动的其实只有一处：就是将原来一个叫“你被解雇了”的栏目标题改成了“最难做的工作——绩效不佳者”，因为以前那个标题太容易让人误解了。

每个栏目都以一段激昂的演说开场，然后是问题根源的分析，最后以我对这个问题如何改善的建议结束。我酷爱文字游戏、头韵和通俗文化，因此栏目中充斥着这些东西。特别是大部分栏目的标题和副标题都直接取材于歌词、电影对白和谚语。是的，我自娱自乐，但撰写这些栏目确实给我带来了些许乐趣，同时也使我得到了痛快宣泄。希望你也会喜欢！

微软的组织结构

因为这些栏目最初是写给微软的内部员工看的，因此有必要简要了解一下微软以及我在工作中扮演的角色，这有助于更好地理解这些文字。

目前，微软的产品开发分成七大业务部门，各个部门相应负责我们的主营产品——Windows、Office、Windows Phone、Interactive Entertainment（包括 Xbox）、服务器软件及工具（包括 Windows Server 和 Visual Studio）、Dynamics 以及在线服务软件（包括 Bing 与 MSN）。

每个部门大约包含 20 个独立的产品单元或管理小组。通常情况下，这些产品单元共享源代码控制、创建、安装、工作条目跟踪和项目协调（包括价值主张、里程碑安排、发布管理和持续性工程[⊖]）。除了这些相应的服务之外，产品单元或小组还有高度的自主权，可以对产品、流程和人员做出自己的安排。

一个典型的管理小组通常由三个专职经理组成：项目组项目经理（Group Program Manager，GPM）、开发经理（Development Manager）和测试经理（Test Manager）。一个产品开发单元通过这三个专职经理向产品单元经理（Product Unit Manager，PUM）负责。如果没有产品单元经理，那么他们分别向他们的上司并最终向部门主管汇报。其他工程领域，比如用户体验、内容发布（比如在线帮助）、创建和实施，这些可能单独对某个产品单元负责，也可能在整个部门中共享。

每个工程领域抽出一个或多个代表组成一个虚拟团队，由该团队向这三个专职经理负责并为一个单独的功能模块工作，该团队称为功能团队（feature team）。有些功能团队选择敏捷方法，有些喜欢精益模型，有些采用传统的软件工程模型，有些则根据实际情况综合采用上述多种方法。

微软如何整合所有这些多样化又独立自治的团队并使其朝向一个共同的目标有效地工作呢？这就是部门公共项目协调组所要扮演的角色了。例如，部门的价值主张是为所有的专职管理小组和他们的功能团队设置统一的关键示例、质量尺度和准则。

示范工具和文档

本书提到的称为“在线资料”的示范工具和文档，都可以通过如下地址下载：

http：//go. microsoft. com/FWLink/？ Linkid 220641

⊖ 持续性工程（sustained engineering）指直接发布软件产品而不像通常先出测试版改进后再出正式版，产品发布后根据用户使用情况再改进，然后再次发布新版本，如此反复。——译者注

在线资料列表

工　具	栏　目	章节
SprintBacklogExample. xls；SprintBacklogTemplate. xlt	敏捷子弹	2
ProductBacklogExample. xls；ProductBacklogTemplate. xlt	敏捷子弹	2
Spectemplate. doc；Speccheckliste. doc	糟糕的规范书：该指责谁？	3
InspectionWorksheetExample. xls；InspectionWorksheetTemplate. xlt；Pugh Concept Selection Example. xls	复审一下这个	5
InterviewRolePlaying. doc	面试流程之外	9

系统要求

本书提供的工具都是微软的 Office Excel 2003 和 Office Word 2003 格式的。只要你的电脑上安装有 Word Viewer 和 Excel Viewer，就能使用这些文件。你也可以从如下站点下载这两个应用程序：

http：//www. microsoft. com/downloads/en/details. aspx？familyid = 941b3470-3ae9-4aee-8f43-c6bb74cd1466&displaylang = en

勘误及图书支持

我们尽最大的努力保证本书及其附属内容准确无误。自本书出版以来发现的错误已经发布在 oreilly. com 上的 Microsoft Press 站点：

http：//go. microsoft. com/FWLink/？Linkid = 220642

如果你发现有错误还未公布出来，可以通过这个网站告知我们。如果你需要更多帮助，请发 E-mail 到 Microsoft Press 图书支持部：

mspinput@ microsoft. com

请注意，上面的邮箱地址不对微软的软件产品提供支持。

第 1 版前言

一直以来，我就是 Eric Brechner（以 I. M. Wright 为笔名）的忠实读者。当我第一次遇见他的时候，我花了好长时间才发觉跟我谈话的人似曾相识，因为 Wright 先生之前给人印象是高傲自大的，而眼前的他是一个恭谦、礼貌且友好的人，他看起来更像 Clark Kent[⊖]。

我所喜欢的栏目主要集中于微软内部团队在软件开发的过程中是如何处理技术与人际交流之间的关系的。看到大量的公司内幕被写了出来，我常常会感到吃惊——我不知道还有多少不为人知的故事没有说出来。

大型项目的软件工程管理者面临着 3 个基本问题。第一个基本问题是，程序代码太容易改变了。与机械或土木工程不一样，它们在现有系统上做一次改变总是要付出实实在在销毁某些东西的代价，而软件程序的改变只需要敲敲键盘就行了。如果对一座桥的桥墩或一架飞机的引擎做一个错误的结构性更改，由此产生的后果，即使不是专家也很容易就能明白。然而，如果在一个现有程序上做修改，对于其风险性，即使经验丰富的软件开发者进行了充分的讨论，其结论常常还是错的。

将软件以建筑作为比喻实际上可以给予其很好的解释。基于程序代码在系统中所处的层次，它们可以被比作“基础、框架和装饰”。“基础”代码具有高度的杠杆作用，它们的改动常常会引起严重的连锁反应。“装饰”代码比较容易改动，而且也需要经常被改动。问题是，累积了几年的改变之后，复杂的程序就跟经历了几次装修的房子差不多——电源插座躲到了橱柜的后面，浴室风扇的出风口通向厨房。再做任何改变的话，其副作用或最终的代价都是很难预知的。

第二个基本问题是，软件行业还太年轻，实际上还未提出或建立关于可复用组件的正确标准。龙骨间的间距必须为 16 英寸才能安装 4 英尺 ×8 英尺的石膏板或夹板？我们不仅在这类问题上还没有取得一致意见，甚至还没有决定，是否龙骨、石膏板和夹板这样的组合更可取，还是我们去发明像泥浆、稻草、石头、钢铁和碳化纤维这样的组合更好。

最后一个问题实际上是第二个问题的另一种表现形式。每个项目中重复创建的软件组件，它们也被重新命名了。软件行业里对现有的概念发明新的名字是很常见的，即使用的名字相同，这些名字也以新的方式被重用。行业里有一个心照不宣的秘密：对于如何选择最好的开发方式已经有不少的讨论，这些参与讨论的人使用不同命名，他们之间的沟通完全是一头雾水。

表面上看来，这些都是很简单的问题：建立一些标准，然后强制执行它们。在快速进步的大容量、高价值和低成本的软件世界里，这样做可是一个让你的业务落败的捷径。实际情况是，软件最大的工程障碍，同时也是它最大的优势是无处不在的软件（运行在低成本的个人电脑和互联

⊖ Clark Kent 是“超人”的名字，他具有超强的本领，是一个虚构的超级英雄，美国漫画中的经典人物。——译者注

网上）已经使其以惊人的步伐进行创新成为可能。

随着微软的成长，公司已经不再能在最佳工程实践方法的研究方面大量投入，然后经过深思熟虑，挑选出其中质量最好的方法。个人电脑和 Windows 的成功，已经把公司从按传统方式做些小项目的形态转变为谱写开发有史以来最庞大、最复杂软件的新篇章。

为了能够创建出在风险、效率与创新之间取得平衡的最佳系统，微软面临着持续不断的斗争。考虑到我们的一些项目有着极度的复杂性，这些努力甚至可以称得上“英勇无畏”。在过去的一段时间，我们已经设置了专员，建立了专门的组织，他们都一心一意致力于这个行业里最困难的事情——“软件发布”。我们已经形成了一系列的群体风俗、行业惯例、企业文化、开发工具、流程及经验总结，这些都有助于我们建造和发布这个世界上最复杂的软件。但与此同时，每天都处理这些问题难免让人心惊胆战、意志消沉。Eric 的栏目正是大家一起分享和学习的极好途径。

Mike Zintel

微软公司 Windows Live 内核开发部门总监

目　录

第 1 章

项目管理失当

本章内容：

2001 年 6 月 1 日：“开发时间表、飞猪和其他幻想”
2001 年 10 月 1 日：“竭尽所能，再论开发时间表”
2002 年 5 月 1 日：“我们还开心吗？分诊的乐趣”
2004 年 12 月 1 日：“向死亡进军”
2005 年 10 月 1 日：“揭露真相”
2008 年 9 月 1 日：“我得估算一下”
2009 年 5 月 1 日：“一切从产品开始”
2009 年 9 月 1 日：“按计划行事”
2010 年 5 月 1 日：“敏捷的团队合作”

我的第一个栏目是在 2001 年 6 月微软内部网络杂志《Interface》上发表的。为了进入 I. M. Wright 的人物角色，我需要找到一个真正让我苦恼的主题。而工作时间表和进度跟踪再好不过了。

项目管理的伟大神话至今都让我疯狂，它的威力远胜过其他任何主题。这些神话是：

1. 人们可以按期完成任务（事实上，项目可以按期交付，但项目员工能按期完成各自任务的概率不会高于击中曲棍球的概率）。

2. 有经验的人估算日期比较准（事实上，他们能够较好地估算工作，但不是日期）。

3. 人们必须准时完成各自的任务从而使整个项目按时交付（事实上，员工们不能按时完成自己的任务，所以你若想你的项目能够按期交付的话，你必须进行风险管理、范围管理并通过沟通来减轻人性的弱点可能给项目带来的负面影响）。

在这一章中，I. M. Wright 将讨论如何通过管理风险、范围和沟通，来保障项目能够按时完成。前两个栏目专门讨论开发工作时间表，接着讨论善后事宜的管理（我们称之为“Bug 分诊”）、死亡行军、撒谎以掩饰问题、快速而精准的估算、服务管理、风险管理，以及在一个大型项目中整合各种方法以便协同运作。

不得不提的是，通过我在微软这些年工作期间的观察，在不同的规模、不同的抽象层次中，项目管理有不同的表现形式。这些层次包括：一个团队或功能层次（大概 10 人左右）、项目层次（50 ~ 5 000 人为一个特定版本工作）以及产品层次（由高级主管领导的多个版本开发）。敏捷开发在团队层次很适用，传统方法在项目层次中很适用，而长期的战略性规划方法在产品层次很适

用。然而，人们很少同时在不同层次工作，实际上，每个人总是会分年段地在这些层次间工作。所以人们常常认为某一层次的高效方法应该应用到其他层次上，悲剧就这样产生了。准则就是：小型、紧凑的团队跟大型松散的组织动作方式不同，应相应地选择适合你的方法。

——Eric

2001年6月1日："开发时间表、飞猪和其他幻想"

一匹马走进酒吧，说道："我能在两天内完成那个功能。"开发成本计算和时间表是个笑话。相信它的人，要么是傻瓜，要么是初出茅庐的项目经理。这不是模糊科学，这是瞎编。不错，的确有人相信编码工作可以被精炼成一个可预见进度和质量的可重复的过程，那我儿子至今还相信牙仙子呢！事实上，除非你只需编写10行那么长的代码，或者代码可以直接从以前的工作中复制过来，否则你不可能知道编码会花费多长时间。

作者注：项目经理（Program Manager，PM）有很多职责，其中职责之一是说明最终用户体验和跟踪所有项目的整体进度。这种角色是必要的，但他们常常不讨开发者的喜欢，因而也很少得到开发者的尊重。真遗憾，项目经理是一份很难做好的工作。但是，对于Wright先生来说，项目经理仍然是一个有趣并且容易达到的目标。

译者注：①关于牙仙子（Tooth Fairy）。美国人有个信仰：小孩子换牙时，父母会告诉他把牙齿用信封装好，放在枕头下，早上起来的时候牙仙子会用钱跟他换牙齿。这钱当然是父母给的，用来鼓励小孩子拔牙。牙仙子在美国是人尽皆知的，虽然只是一个"善意的谎言"！②关于飞猪（Flying Pigs）。猪会飞吗？美国人常用此来比喻离奇荒诞之事。

里氏震级估计

译者注：里氏震级（Richter-scale）是地震等级的一种数值标度，分1～10级，每上升一级，强度增加约60倍。

当然，你可以估算，但估算出来的时间是成对数关系的。有些事情需要花费几个月，有些事情需要几周，有些需要几天，有些需要几个小时，有些则只需几分钟。而我跟我的项目组项目经理（Group Program Manager，GPM）一起给一个项目做时间安排时，我们对每个功能使用"困难/中等/容易"3个等级来评估。"困难"意味着一个全职开发人员需要花费整个里程碑时间；"中等"意味着一个全职开发人员需要花费2～3周时间；"容易"意味着一个全职开发人员需要花费2～3天时间。没有其他等级了，也不做精确的时间表。为什么呢？因为我们俩知道，我们无法预测更精确的时间了。

在我的记忆里，除了一系列里程碑、测试版、正式版发布等"项目日期"外，我没有在开发时间表上为各个功能规定交付日期。一个好的开发时间表应该是这样的——它只是简单地列出在

每个里程碑期间需要实现的功能。那些“必须有”的功能放在第一个里程碑期间内并且都是要完成的，如何完成则是根据开发人员的数量和“困难/中等/容易”等级；“最好有”的功能放在第二个里程碑期间内；“希望有”的功能放在第三个里程碑期间内；除此之外的所有功能统统不做。通常情况下，如果到了第三个里程碑期间的第二周，仍然有较多“最好有”、“希望有”的功能没有实现，这时候大家都很惶恐，你就要把所有“希望有”的功能扔掉，并且“最好有”的功能也只保留一半。

作者注：里程碑的设定因团队而异，也因产品而异。典型情况下，一个里程碑跨越 6～12 周不等，是“项目日期”，是组织（50～5 000 人）用于同步工作和复审项目计划的时间点。在里程碑期间，各个团队（3～10 人）可能使用他们自己的方法来跟踪具体的工作，比如简单的工作条目（work-item）清单、产品备忘录及实施进程图。

译者注：产品备忘录（product backlog）是在项目开始的时候，需要准备一个根据商业价值优先级排好序的客户需求列表，是一个最终会交付给客户的产品特性列表，涵盖所有用来构建满足客户需要的产品特性，包括技术上的需求。实施进程（burn-down）图是敏捷软件开发方法 Scrum 中常用的一种图表，用来展示剩余待完成工作与时间之间的关系。时间标识在横坐标轴上，未完成工作标识在纵坐标轴上。

风险管理

这才是我要引出的主题。在开发成本计算和时间安排上不能只盯着日期或时间不放，应该关注风险管理。我们通过软件的功能和特性来取悦客户，不管这是个软件套包还是网络服务。这里的风险指的是，我们未能在合适的时间，将符合质量要求的功能集合交付到客户手中。

一个好的开发时间表通过优先处理关键功能来管理风险。这些关键功能是能让客户满意的最小功能集合。通过“困难/中等/容易”这种评级方法，可以判断出在这个最小集合中包含哪些功能是切实可行的。其他的功能按照优先顺序和一致性原则依次加入。

然后你开始编写代码，并且看着功能实现从困难转向容易，又从容易转向困难。通过集中所有必要的资源，以降低不能按时交付高质量的“必须有”的功能的风险，其他的都是次要的。你还可以将不紧要、但又不失挑战性的项目交给实习生去做。

作者注：具有讽刺意味的是，几乎所有的工程师和经理都赞同优先处理“必须有”的功能，但事实上很少有人真的这么做，因为“必须有”的功能通常是乏味的，比如安装、软件工程创建、向后兼容性、性能优化和测试套件等。然而没有这些功能，你的产品根本就无法发布。因此，产品发布往往是因为这些问题而一拖再拖。

一定要破除“功能交付日期”的神话，因为开发人员专注于这种日期的时候会破坏风险管理。真正要关心的日期只能是“项目日期”，比如各个里程碑、测试版，等等，而绝不应该是“功能交付日期”。项目日期之间一般都有较长时间的间隔，而且这种日期不会很多。管理这几个日期要容易得多。如果要求开发人员在某个日期之前一定要实现某个功能，当他们不能按时完成

时他们往往不会告诉你，而是对你说“我正在加紧做……我会加班……”之类的话。

在软件开发过程中进行风险管理，我们还要特别注意以下几个因素：一个是过度劳累的员工，一个是匆匆忙忙实现的、质量很差的功能，再一个就是你花费几周的时间且动用2～3位甚至更多的高级开发人员去解决一个棘手的问题。如果你的开发人员是在围绕“功能交付日期”付出大量的努力，而不是帮助你在产品的关键功能上降低风险，那么真有可能浪费时间了。

客户赢了

一个产品的成功与否，取决于你对关键功能的风险管理能力。当你给你的开发团队解释清楚这一点之后，情况就完全不一样了。当然，额外的功能可以锦上添花，但最关键的还是要专注于存在风险的地方，充分沟通，并一起努力把它们解决掉。

当所有人都理解了目标，所有人都能比以前工作得更好。每个艰巨任务的完成都能鼓舞士气，即使初级员工也会因为成熟的决议而得到回报。最终，我们的客户是大赢家，因为他们得到了真正想要的功能，并且产品质量也是他们当初所期望的，而不是一些勉强实现的、质量不能保证的垃圾。

顺便提一句，我对开发时间表的所有论述，对于测试时间表同样适用。

2001年10月1日：“竭尽所能，再论开发时间表”

该对我6月份的那个栏目（“开发时间表、飞猪和其他幻想”）的评论做出一些回应了。其实，大部分评论都是恭维之词，这里就不再赘述了，因为没有必要再次证明我有多么正确。我这里要做的是，回应一下那些对那个栏目还在无知中徘徊但又非常热情的读者朋友们。

作者注：这是我仅有的一个“邮包”栏目，收集了我对一些读者来信的回复。我还在持续不断地收到读者对我的栏目的大量“反馈”，但一旦一个栏目很受欢迎，很多新的话题便会涌现出来；讨论那些新话题的价值要远远超过对一个老话题邮件的回复。不管怎么样，当我回顾这个早期的栏目时，我意识到，可能Wright先生应该再次清空他的邮包了。

软件工程绝对是含糊的

我对关于不能也不应该对一个功能的开发做时间安排的论断表示怀疑。文中精确地论述了“编码”活动。遗憾的是，这是初中生干的事情——拼凑一个VB程序来解密信息、相互通信。我们可是软件工程师啊，不是电脑苦工。

——一个充满怀疑的无知者

我经常听到这种说法，但请就此打住。银行经理并不管理银行，软件工程师也不在软件上做工程。他们开发软件，定制软件，通常事无巨细、从头至尾参与其中，并不需要了解操作范围、公差、故障率、压力条件等度量标准。的确，我们的系统有这些标准，但这些标准不是为软件编码准备的。

我曾到一个工程学校进修过。我的朋友当中也有很多是电力、基建、航空、机械等方面的工

程师。这些工程师做的项目，其构造模块和结构流程都经过了很好的定义和提炼，而且都是可预测的。虽然有时候为了达到客户的要求需要一个合适的设计，但只要用一种新颖的方法把各个模块组合在一起就很有创造性了，就算是最标新立异的建筑也会符合一定的公差要求，并且具有严格的可控质量和功能。

但对软件开发来说，情况就不一样了，尽管很多人竭力想让这两者达成一致。软件的各个构造模块太底层了，变数太多。它们之间的交互影响太难预料了。像 Windows、Office、Visual Studio 及 MSN 等大型软件系统的复杂度，已经远远超过了工程的正常范围，以致哪怕只在这些系统中做微小的功能改动，也无法粗略估计出这些改动所引起的“平均失效时间”。

因此无论好坏，还是抛开痴心妄想和崇高理想，回到现实中来吧！我们必须承认，我们是开发者，而不是工程师。我们不能奢望轻易得到传统的工程领域积累了成百上千年的经验才做到的“可预测性”。这无异于我们奢望：不用跟电脑说什么，电脑就能按照我们心里的想法去做事。我们还办不到！

作者注：在我写下这个栏目 6 年后的今天，微软已经对很多软件进行了“平均失效时间”的评估。除此之外，把编程当做工程看待的各种方法也逐渐出现了。这个我会在第 5 章的“软件发展之路——从雕虫小技到系统工程”栏目中再次介绍。纵然如此，我仍然认为本栏目很好地见证了软件开发作为一个专业领域，它已经走过了幼年，但跟它早已长大成人的传统工程兄弟相比，他还只是个十几岁的小朋友。

相信一半你看到的，别信你听到的

如果我在某个功能或者一段代码上依赖于另外一个团队或产品组，我肯定不想听到像“你要的东西应该可以在这个里程碑期间内完成”这样的说法。我需要一个很具体的交付日期。我要有具体细节。

——一个需要日期的人

我想写几个关于依赖关系和组件团队的栏目，也许将来会吧，但眼下我只想讨论依赖方的开发时间表。首先，假设你的依赖方确实有一份开发时间表，你会相信它吗？你也许会说：“当然要信，我有其他选择吗？”建议在你的胃病恶化之前赶紧吃一点 Pepcid™（一种胃酸抑制剂）。不光只是开发时间表，不要相信依赖方所说的任何事情。如果他们坐在隔壁房间，他们告诉你外面正在下雨，你首先要做的是到自己的窗口去看一下。

但我并不是说你不能跟依赖方合作。相反，你应当与他们很好地合作，因为依赖方可能为你的团队、产品和客户带来大量的经验和意外的收获。我只是告诫你要高度警惕当前正在发生的事情。要向他们提出定期多次交付的要求，并对交付的东西进行自动化测试。获取他们对 RAID 进行读和写的 RDQ，观察它们的数量以及存在问题的地方。派你的项目经理去参加他们的分诊会议。加入他们的邮件列表。

作者注：查一下本书最后的“术语表”，以便理解这些用于 Bug 跟踪的词汇。

基本上，你需要像鹰一样盯着依赖方。他们是你的团队和产品的一个扩充。你跟他们接触、沟通得越多，你在规避其短处以及促进其改变方面的能力就越强。至于他们承诺的功能什么时候

能够完成，你必须依赖你在如下3个方面上的影响力：提高优先级，沟通渠道，独立测试（为了知道他们的功能是否真正可用了）。

激励：不能光靠比萨和啤酒

总的来说，你的观点用在项目早期的计划上还行，但对于产品发布前的最后一个里程碑就不那么合适了。时间表怎样提供最后期限和时间约束，让团队遵照执行，使得它能作为一种日常管理工具去激励团队的执行力？你必须要解决诸如此类的问题。

——一个找不到（汽车）油门的人

首先我要重申：如果你坚持让开发人员遵从“功能交付日期”，那他们为了准时交付可能会撒谎。他们会隐瞒自己的工作状态，会在质量和完成度上给你虚假信息。如果你不想你的开发团队这么对你的话，你必须建立起一个更好的激励机制。我用过3种不同的方法，这些方法能使大家互相协调工作并产生良好效果。

第一种，也是最基本的方法，就是应用里氏震级估计。我的开发人员知道，我期望的是每个功能在大致那么多的时间内完成。如果一个原先估计需要2周的任务实际上花了2周半，可能关系不大。但如果花的时间比原先估计的要长得多，那么通常是有实实在在的原因的，那个开发人员必然会让我知道这个原因。如果缺乏充分的理由去延期交付，则足以对开发人员形成一种鞭策。然而，因为没有卡得很死的日期，大家几乎不会去想到隐瞒和欺骗。

第二种激励工具是瞄准里程碑日期。这有招致大家走捷径的危险，但总体的效果是鼓励开发人员从一开始就努力工作，并且让他们对自己是否落后于进度做到心里有数。“功能交付日期”和里程碑日期关键的不同在于，后者是给整个团队设置的日期，需要整个团队一起努力去达到它。因此，个人抄近路的压力就会小很多。然而，这种危险性仍然无法杜绝，逼得我使出最后也是最有效的一招。

> **作者注：**一个自我导向的团队向着一个清晰的共同目标一起努力，这是很多敏捷方法的核心概念，尽管在2001年我还不知道有敏捷方法。

最后一种激励工具是迄今为止我使用起来最有效的。我向团队解释清楚哪些功能是必须要有的，必须优先完成。我告诉他们，必要时任何其他的功能都可能被放弃不做。遗憾的是，这些必须要有的功能常常做起来比较乏味，没有意思，甚至不值得一提。因此我告诉我的团队，如果他们想要做那些很酷的功能，必须首先保质保量地完成之前的这些关键功能。之后，再去做那些不那么关键却要炫得多的东西。这种激励是积极的，有建设性的，并且非常有效。屡试不爽！

在日期上沉沦

继续前面的讨论：时间表在不同的功能单位之间（不只是开发，还有项目管理、测试、用户体验、市场推广、外部合作）同步工作时，绝对是必要的；你还必须要解决这个问题。

——一个出格的人

如果你确实需要具体的“功能交付日期”来同步各个方面和依赖方的工作，那么你的软件永远也没法发布。当然，我们的软件一直在发布——我们甚至准时发布了庞大的Office XP。要知道，

它的发布日期是在两年之前计划的。因此，肯定存在其他的一些关键因素。

其实真正起决定作用的，是要在顺序、成本和方法上达成一致，并且提供及时的状态报告。各个方面之间要协商达成协议，定义好状态汇报的流程，并且避免工作的相互牵制。

- **顺序**：讨论协商各个功能实现的先后顺序已经不是什么新鲜事了，尽管有些部门从来都不能对优先顺序达成一致。
- **成本**：成本的协商通常发生在开发人员和项目经理之间（例如一个开发人员说："如果我们使用标准控件，可以帮你节省 2 周的时间。"）但有时候就只是开发人员做决定。其实，成本的协商也应该让测试和实施人员参与进来。
- **方法**：讨论协商使用哪种方法通常在项目管理规范书中做，但很少在开发和测试的规范书中做——这对他们不利。
- **状态汇报**：至于状态的及时汇报，你务必要把邮件和测试发布文档（Test Release Document，TRD）登记起来，或者两者任选其一，以便让项目经理、测试和实施人员了解项目的进展情况。测试部门需要对阻碍工作继续进行的 Bug 使用警报。项目经理采用类似于"规范书变更请求"（Spec Change Request，SCR）的方式来汇报规范书的更改。（了解更多关于 SCR 的内容，可阅读第 3 章的"迟到的规范书：现实生活还是先天不足"栏目。）

如果各个不同的部门能够对他们的工作顺序进行合理的安排，知道各自的工作需要花费多少时间，对他们使用的方法也很有信心，并且保持着最新的状态报告，那么项目就上轨道了！问题找到了，风险降低了，意外也很少发生。更重要的是，没人顶着因为人为的交付日期而犯错的压力。相反，每个人都朝着同一个目标在努力——为我们的客户交付一个令人愉快的体验。

2002 年 5 月 1 日："我们还开心吗？分诊的乐趣"

如果你没有把这个概念搞清楚，请告诉我……

项目经理希望瞬间得到无穷多个功能，测试人员和服务营运人员希望永远也不要增加新的功能，而开发人员只想在不受外界干扰的环境下编码实现很酷的东西。现在，邀请这几个方面的主管，让他们带着相互冲突的理想去同一间房间，关上门，再给他们点可以用来打架的东西。会发生什么呢？分诊！

作者注：当产品开发问题涌现（比如未完成的工作条目、Bug 或设计变更）出来时，它们都被记录在一个工作条目数据库中。分诊会议就是为了安排这些问题的优先级，并且决定每个问题如何解决而召开的。很多"冲突"（保守的说法）就是源自于这个会议。

译者注：关于分诊（Triage）。这是一种根据紧迫性和救活的可能性，在战场上决定哪些人优先治疗的方法。战地医学的基本原则之一是，如何将伤者划分为 3 类：①无论是否接受医疗都会死亡；②无论是否接受治疗都没有生命危险；③只有及时地接受医治才能保全性命。抛开它的病理本质，分类本身极其重要，只要你希望最大限度地保存有生力量。如果你不进行分类，结果将会比你做分类时要糟糕得多。

很庆幸，鲜血没有从分诊室的门缝中流出来。当然，这正是调解员要做的事情。大部分分诊会议都搞得像大屠杀一样。但必须要这样吗？我在微软见过的几次最激烈的争吵就发生在分诊会议室里。这样很糟糕吗？抑或就该这样子？

战争是地狱

参加过残酷的分诊会议的人，他们都会告诉你这样不好。即使你赢得了大部分的争论，你也会被这粗暴的分诊搞得精疲力竭。

基本上，病态的分诊和病态的团队常常同病相怜。它们在团队成员身上制造坏的“血液”，让他们常常产生报复性的、毫无建设性的行为。

为什么要这样呢？我们这里鼓励激情。我们想要人们为他们的信念而战，站在我们客户的立场上做出正确的决定。带一点点健康的竞争有问题吗？然而，当这竞争不是一点点也不健康时，那就不好了。

这不是个人的事情

不应该认为 Bug 是个人的事情，但事实却是这样的：

- 对于发现 Bug 的那个测试人员来说，Bug 代表着他工作的质量：“什么叫这个 Bug 不够好，无需修复？”
- 对于定义这个功能的项目经理来说，Bug 考验着他当初的设计：“那个 Bug 完全摧毁了这个功能的特色！”
- 对于服务营运人员来说，Bug 意味着实实在在的、可能永无穷尽的工作：“什么，你不关心这个 Bug？反正不要你每天早上 3 点钟进办公室来重启服务器！”

> **作者注：**关于早上 3 点钟重启的趣事。像大部分提供软件服务的公司一样，微软现在正在改变营运模式，不再提供每周 7 天、每天 24 小时的不间断电话服务了。取而代之的是，我们把服务设计成具有自愈能力（重试、重新开始、重启、重新镜像、重置机器）的系统。现在的服务营运人员只需在正常的上班时间，根据自动产生的替换清单对组件进行更换就行了。

- 对于开发人员来说，Bug 代表着一种个人的价值判断：“事情没那么糟糕吧！”

分诊所做的决定应该基于我们客户的利益和微软的利益，而不能光凭个人的感觉。然而，因为各个方面对于 Bug 有着各自不同的立场，分诊讨论转眼之间就会脱离轨道。

分诊的 5 条黄金法则

你怎样才能保证分诊正常地进行并且具有建设性呢？采纳我下面的 5 条黄金法则吧：

1. 关上门。分诊是一个协商的过程，而协商最好在私底下进行。当做决定的过程保密时，大家更容易坦诚相见、相互妥协乃至达成一致。这也便于分诊的与会者把他们的决定作为团队的决定向其他人解释。

2. 所有的决定都是团队决定。当达成一致意见之后，所做的决定就不再是个人决定，而已经

上升到了组织的高度。作为分诊团队中的一员，每个人都要无条件地支持这些决定。分诊的与会者应该具备为每个分诊决定做出辩解的能力，就好像这些决定完全出自于他自己一样。

3. 每个专职领域只派一名代表。分诊必须快刀斩乱麻。遗憾的是，参与的人越多，过程就越漫长；个人情绪掺杂得越多，一致的结论也越难达成。一个人做个决定可以很迅速，但你需要整合各个方面的观点来做出一个集思广益的选择。因此，折中的办法是，每个专职方面各派出一名代表参与讨论，从而兼顾效率的同时，使各方观点得以充分体现。

4. 指定一个可以做最终决定的人。如果与会者不能达成一致，我们就需要有一个人做出最终决定——理想的话，这种情况不会发生。就我个人而言，我更倾向于让项目经理来做这个最终决定，因为项目经理本来就是做协调工作的，之后也是他的职责要去解释这些决定并让其付诸执行。他应该不会滥用这个特权。然而，真正的威胁还在于，可能会有某个工程方面的代表（绕开项目经理!）把他的决定强加给所有的与会者，这样也能让大家达成一致。

5. 所有的决定都应遵从“贵格会”信条。这是最重要的一条法则。通常所说的“一致意见”意味着所有人都同意，但对于像分诊这样艰难、牵涉个人观点的事情，这个要求显然太高了。遵从“贵格会”信条只是意味着没有人反对——所有与会者必须为大家都能接受的解决方案而协同工作，只要不起冲突即可。这样就会产生一种很容易就能实现的、通常也是最理想的结果。（注意：这里说的“贵格会”只是指遵从相同信条的一类人，而不是有宗教信仰的那种。）

遵照上述 5 个法则，你的分诊就会充满热情、具有建设性并且富有效率。不过，下面我还要讲一些细节问题，以使得本栏目更加充实。

魔鬼藏在细节里面

这里有一些细节上面的处理技巧，可以让你的分诊会议开得更加顺利：

- 如果你们的争论针对的是人，而不是 Bug，你就需要把焦点转移到“做什么最有利于客户和长期股票价格”的话题上去。这种方法避开了讨论个人问题，同时把会议的焦点集中到了它本该集中的地方。

> **作者注：**在所有的栏目中，我都在谈论要把注意力放在客户和业务上，而不是个人问题上。你可能想知道，为什么你不能只考虑客户，而不去管长期的股票价格。我对这个观点表示理解，但我也知道，如果我们的业务做得不好的话，我们也就没有客户了。因此，拥有一个商业计划来指引我们的工作，这对于为我们的客户提供可持续性的利益是很必要的。

- 如果你对某个 Bug 或某个修复需要得到额外的信息，有时候你需要通过电话或亲自从分诊团队之外邀请一个人进来。请记住，当你完成提问之后，继续争论你们的决定之前，一定要送走这个访客。否则，分诊的机密性就会被破坏，所做的决定也就不再是分诊决定了。
- 如果你想让你的团队中的某个人了解分诊过程，则可以邀请他加入一个分诊会议，但叮嘱他，务必在你们讨论期间要像趴在墙上的苍蝇那样保持安静，并且向他强调协商过程的机密性。

很难进行下去，不是吗

如果一个或几个分诊团队成员不能就某个问题达成一致，会议无法进行下去，就给他们一些“银弹”吧！游戏规则是，你任何时候都可以用银弹来获取特权，让大家遵从你的意见，但是子弹用掉一颗就少一颗，用完为止。当有人在某个问题上不肯屈服的时候，可以问他：“你想用一颗银弹吗?”如果用，其他人都要支持他的决定。通常这个人会说：“不，不，这事没那么重要。”然后，会议可以继续进行下去。

> **作者注：**几年来，这个关于分诊的栏目引来了大量的争论，特别是上面关于“银弹”的这段。有些人抱怨不应该使用“子弹”这个字眼，而要用“令牌”。更多的抱怨是：一个关键的团队决定可能由某个人通过使用他的“银弹”做出，这个很危险！但实际上，这种事情永远也不会发生。“银弹”是一种稀有资源，它用来帮助它的主人提升问题的重要性。大家不会轻易使用它。因此，如果有人在一个关键问题上滥用一颗“银弹”的话，总会有其他人使用剩下的“银弹”去对抗他。尽管如此，我还从来没听说有这种事情发生呢！

最后，该到数据库中去解决分诊会议讨论过的 Bug 了：

- 记得使用“分诊”标签去表示这是一个分诊会议决定。
- 记得解释清楚分诊团队所做决定背后的考量。
- 不要去“解决”Bug（尤其是外部 Bug），除非你再也不想看到它了。通常情况下，团队把有碍产品发布的 Bug 标为“外部”或“待办”，意思是说，“我们现在不想处理这个 Bug，以后会另行安排。”但如果那个 Bug 被“解决”了，它就落在了“雷达”能够扫描到的有效区域之外，相应的问题也就被隐藏了起来。

> **作者注：**你可以在第 2 章“我烦扰你了吗？Bug 报告”中看到关于 Bug、优先级、分辨率的话题。

谨小慎微

分诊被证明是你需要对团队履行的最重要的职责之一。良性的分诊会议几乎总是跟良性的项目和项目组直接相关。这种关系的真正美妙之处在于，积极、富有成效且愉悦的分诊会议将给你的工作、你的团队带来同样的效果。但跟解决团队和项目的全部问题比起来，解决分诊中遇到的问题要容易得多，牵扯的人也要少得多。

再好不过的是，你们使分诊会议得以改善，并步入正轨，这可能会成为你一整天最快乐的事。当分诊的焦点是 Bug 而不是人，大家意见统一而不是相互攻讦时，那么会议的紧张气氛就会消散，纷争与挫败就会以诙谐的氛围而代之。健康良性的团队在一起工作，他们开的分诊会议常常充斥着俏皮话、玩笑、矫情讽刺和令人发笑的误述。对你的分诊技巧做一些适当的调整吧——你们的笑声可能会传到走廊上，久久回荡！最好总是关着门。

2004 年 12 月 1 日："向死亡进军"

曾经在一个死亡行军的项目组呆过吗？也许你现在所做的项目正是。这类项目有很多种定义，但基本上都可以归结为，"在太少的时间内要做太多事"。因此，你被要求在一段很长的时期内，每天工作很长的时间，以消除两者之间的矛盾。死亡行军因为其漫长、艰辛和伤亡重而得名。（如果我的说法伤害到了在第二次世界大战中真正经历过死亡行军的那些人的亲属，我道歉。不过，遗憾的是，软件中随处可见不当文字的使用。）

很难搞明白，为什么这么多项目组继续进行死亡行军，即使他们几乎肯定会失败，有时候还很悲壮！毕竟，毫无疑问，你在走向死亡。我看不出有任何诱因所在。

> **译者注：**关于死亡行军（Death March），典出第二次世界大战太平洋战场的菲律宾群岛。1942 年夏季，日军相继猛攻巴丹半岛和哥黎希律岛，美菲联军大败，美国远东军司令麦克阿瑟携夫人乘潜艇逃出战场。接替指挥战局的温赖特少将遵照白宫旨意，认为坚持抵抗只会造成无谓的伤亡，便命令全部美菲军队无条件投降。然后，日军派人把温赖特将军押送到中国沈阳，关入监狱。同时下令美菲所有被俘人员做长距离徒步行军，从巴丹半岛的马利维尔斯奔向位于圣费南多的俘虏营，行程长达 1 000 多公里。此时正值炎夏，病疫流行，粮食匮乏，日军对战俘更是恣意虐杀；等到达目的地的时候，死伤人数竟达 25 000 余人。几个月之后，有 3 名美国士兵从日军战俘营中侥幸逃出，越海到达澳大利亚的布利斯坦，揭开了这次死亡行军的秘密。

暗箭伤人

愚蠢的管理层继续热衷着死亡行军，他们暗箭伤人，因此我要拿出其中的几支"暗箭"来曝曝光。

> **作者注：**死亡行军不只在微软才有，也不是只有微软才普遍存在这样的问题。这是我当初加入这个公司时发现的一个令我吃惊的事实。早在 1995 年我加入微软之前，这个公司就已经以工作时间长而闻名了。对此我很担心，因为我有一个两岁的儿子，并且计划再要一个。但我的上司明确地告诉我，死亡行军并不是公司的制度。他的话是对的，但当微软或其他公司的管理层仍然对这种愚蠢而又不可思议的做法抱有幻想时，那就是另外一回事了。

- **管理层神经太大条。**管理者做事情不考虑后果。他们采用傻瓜才用的方法：太多的工作要做吗？那就加油干吧！最起码，管理者可以辩解说，他们正在做着事情，哪怕做的可能都是错事。
- **管理层天真得难以置信。**管理者不知道死亡行军是注定要失败的。不知何故，好像他们在过去的 25 年里都在睡觉，或者从来就没有读过一本书、一篇文章，或没有访问过任何网络站点。他们认为，每天至少多工作 4 个小时，并且每周多增加 2 个工作日，将会使生产力翻倍。数学可以这么算，但遗憾的是，人是非线性的，不能这么简单地加减。

- **管理层不切实际。**管理者认为，他们的团队能够克服难以逾越的难题。规则和纪录将被打破。他们有世界上最好的团队，他们的团队能够应对任何挑战。显然，他们不明白速度超过一头牛（很难）与快过子弹（不可能）之间的区别。
- **管理层没头脑、不负责任。**管理者知道死亡行军必将失败，这个过程会摧毁他们的团队，但他们还是会蛮干，逞英雄。他们通过请客吃饭、评选明星和高分考核来奖励英雄行为。因为管理者知道，我们的客户和合作伙伴在下一次复审结束之前，是不会被我们交付的“垃圾”吓跑的。我觉得这些管理者是最该被拖去让史蒂夫痛斥一顿的。

> **作者注：**史蒂夫是指史蒂夫·鲍尔默（Steve Ballmer），我们敬爱的CEO。他大力提倡工作与生活的平衡，并且以身作则。我曾多次看到他在他儿子的篮球比赛中加油呐喊，或者和他妻子一起在外面看电影。

- **管理层都是些毫无责任感的懦夫。**管理者知道死亡行军很难避免，但他们缺少勇气说“不”。因为，如果他们随大流，他们就不用承担什么责任，这些懦夫也不会因为项目失败负多大责任。当然，项目一旦失败，他们的员工会恨他们，并且离开团队，但至少他们还可以和跟他们一样懦弱、可怜的朋友分享“战争”的故事。

很多人都撰稿论述过在软件项目中死亡行军的无效性，但不知何故还是有人“慷慨赴死”。我无法说服那些不切实际、不负责任的人，但我可以开导那些神经大条、天真的人，并且教懦弱的人一些方法。

对失败的祈祷

给无知者的一些启迪——死亡行军之所以会失败，是因为：

- **一开始就注定要失败。**很明显，你有太多太多的事情要做，但你的时间远远不够。你必败无疑！
- **怂恿大家走捷径。**当你处在压力之下时，没什么比找一些省事的方法逃离工作更自然的了。遗憾的是，捷径降低质量，并且增加风险。这对于小功能或短期之内的项目或许关系不大。但随着项目不断深入，那些风险和糟糕的质量会贻害无穷。
- **没有太多的时间去思考。**项目需要松弛的时间来产生效率。大家需要时间去思考、阅读和讨论。没有这种时间，你就只能凭着仓促的判断去做事情。而仓促的判断往往是错误的，导致糟糕的设计、计划和质量，引来以后大量的返工或成堆的缺陷。
- **没有太多的时间沟通。**你说得对，沟通不畅和误解是所有麻烦之源。很多其他方面运作良好的项目，就是因为沟通上的问题才失败。当人们每天都要工作很长的时间，他们就无暇顾及沟通，效率也很低。沟通不畅成了一个难以逾越的障碍。
- **制造紧张、压力和机能障碍。**当有压力存在时，适意首先消失了。大家都自顾不暇。意外事件不断被扩大和曲解，抱怨声越来越多，甚至出现更坏的情况，大家不再说话。
- **士气受挫、动力受损。**辛酸、压力及长时间与家庭和朋友疏远，都使人的精神和人际关系折损。最终当项目难逃失败，错过了交付日期，也没有达到质量目标，人们常常会抓狂。如果幸运的话，你只是在项目结束之后转到另一个项目组继续工作。如果不那么幸运，你可能要离职、离婚、生病甚至有轻生的念头。

顺便说一下，管理者常常分不清一些员工在工作上长时间的自愿付出和死亡行军之间的差别。死亡行军是完全不同的概念。它们之间的区别在于，死亡行军强制你付出很多的时间。而当人们自愿时，常常是因为他们真的喜欢。这段时间里他们很放松，没有任何压力或理由去走捷径。

> **作者注：**这是常被人们忽略的最关键的一点。自愿的长时间工作跟死亡行军绝对是不同的。

- **信心在进程中消失。**稍微有点智商的人就能意识到，死亡行军是对出现的问题的一种补救。这样做对于我们的员工、客户及合作伙伴来说不是奉献，是个人能力出了问题。只是埋头工作，回避真正的问题，只会更有损于他人对我们合作能力的评价。
- **不妥善解决问题。**工作更长时间不能解决那个真正导致“在太少的时间内要做太多事”的问题。除非这个问题根除了，否则别指望项目除了变得更坏之外能有其他进展。
- **降低你的标准。**当你已经走了捷径，引入了糟糕的设计和计划，产生了大量的返工和缺陷，打乱了你的信息沟通，怂恿大家相互挑刺，挫伤了员工的士气，摧毁了我们对交付能力的信心，结果还是无法达到质量目标和按期交付（以前遗留下来的任何问题都没有得到解决）时，你就别无选择了。通常这会导致降低质量门槛，将计划延后，然后继续死亡行军。“干得好啊!”

转折点

那么，如果你发现自己正面临着“在太少的时间内要做太多事”的情况，你应该怎么办呢？从务实的角度出发，答案非常简单，就是要搞清楚，为什么你会要做这么多的事情，而你却只剩下这么少的时间了。

答案不是“因为那是管理层设定的日期和需求”。为什么管理层设定那些日期和需求呢？如果你完不成某个需求，或者不能在某个日期按时交付，管理层会做什么？他们会将计划延后吗？延后多少？他们会减少功能需求吗？哪些功能会被减掉？你是否还能在过程或方法上做更多的根本性改变，以求力挽狂澜？告诉管理层，你的目标是达到所有的需求并按期交付，但你必须为最坏的情况做好打算。

然后为最坏的情况做打算。按可接受的最迟交付日期及最少功能制定一个计划。如果在有效时间内还是因为任务太多不能完成，就全面拉响警报！你的项目已胎死腹中。如果你为最坏情况所做的计划是完全可行的，那就要全力以赴。告诫你的员工，勤勉者可以在考核时得到3.5以上的分数，但偷懒者的分数必定在3.0以下。

> **作者注：**这些数字源自于微软老一代的评分系统，分数取值范围为2.5~4.5（分数越高，奖励越丰厚）。分数3.0是可接受的，不过大部分人都想追求得到3.5或者更高的分数。

不寻常之路

你所做的努力使得项目避免了死亡行军，并且赢得了宽松的时间来改善。你的团队很可能比最低要求走得更远，但他们做到这些并没有走捷径，也没有做糟糕的决定或自相残杀。你将按时

交付当初承诺的东西，并且使你的合作伙伴和客户建立起信心。

这听起来倒是合情合理，但其实在情感上很难接受。为最坏的情况做打算感觉像要放弃一样。你像是在示弱，好像你无力应对挑战似的。多么具有讽刺意味啊！事实上，情况恰恰相反。

不面对危机是懦弱的表现。假装最坏的情况不会发生，也只是自欺欺人和不负责任。拿出你的勇气来，面对现实！做事机灵一点，别在最后还要把痛苦留给你的合作伙伴、客户和员工。相反，这样体现了你的价值，你的团队、你的生活、你的自尊毫发无损。

> **作者注：**在最近长达9个月的项目中，我的团队在关键的依赖性服务上拖延了3个月时间才完成整个项目，同时我的团队将原本要4个月时间才能完成的主体功能模块缩减为1个月来完成。我们不能延迟时间表也不能削减软件功能模块（我们已将这两者向客户做出承诺）。我们并没有经历死亡行军，相反，我们直接投入到一个与依赖性服务相关的开发环境中同步进行工作，就像这些服务已经完工一样。这种策略不仅补回了之前过多消耗的时间，而且减少了返工次数，因为我们可以在工程创建的早期就将回馈反应给依赖性服务开发团队。在客户相当满意的情况下我们及时发布了软件。事情很艰难而我的伙伴们工作很努力，但是他们同样有宽余的时间并感到快乐，他们只是出于对发布高质量产品的渴望及为他们的同事提供帮助的心愿。在产品发布后，我们没有一个人离开团队。

2005年10月1日："揭露真相"

我不可能说谎——多诱人的话啊，但也只能骗骗小孩。每个人都会时不时地说谎。有时是要故意掩盖一些细节，有时你没有说出你的真实感受，有时就是彻头彻尾的编造。不管是什么原因或者处在什么环境，说谎就是欺骗，含糊不了。

有些人可能想为这种行为辩解，称这是"善意的谎言"，但它仍然没有改变其本质：不诚实。如果有人发现我说谎，不管这个谎言有多么微不足道，我会立即充满懊悔并老老实实地坦白。不过在我小时候，采取的回应却是抵赖。但后来我了解到，抵赖比当初说谎引起的冒犯更具破坏性。大部分人，包括我在内，说谎的目的不是冒犯别人。我们的动机纯粹是为了私利。

不得不指出的是：欺骗其实是逃避问题的一种最容易而下流的方法。那么它跟软件开发又有什么关系呢？因为通过关注你或者你的团队"在什么时候"说谎，以及"为什么"说谎，你能查明从产品质量到人才保持的所有问题，最终提高生产力。

遭受错觉之苦

说谎是少数几个有价值的可以提醒你有麻烦的"过程噪声"之一。为什么这么说呢？因为说谎、时间周期、进展中的工作和不能替代的员工都会掩盖问题。漫长的时间周期和大量进展中的工作掩盖了工作流程中的问题。不能替代的员工掩盖了工具、培训和可重复性的问题。而说谎能够掩盖任何问题。仔细观察这些过程噪声能够发现问题，并且为过程改进创造机会。

> **作者注：**上述每一种过程噪音我都会在后面的栏目中再次介绍，比如第2章的“精益：比五香熏牛肉还好”和第9章的“随波逐流”。至于下文的“5个为什么”，像“精益”一样，这些概念都来自于丰田汽车公司。

关键是要找出说谎的根本原因。有很多好方法，不过我这里要推荐的是应用“5个为什么”，即连问下面的5个问题：

- 为什么你要说谎？你在隐瞒什么苦衷？
- 为什么要隐瞒那个苦衷？有什么危险？
- 为什么危险会发生？有没有办法避免？
- 为什么你没能从一开始就避免危险？你需要采取什么样的措施？
- 为什么你还坐在那里？赶快行动！

接下来，我们拿工作中经常碰到的、人们会说谎的几个例子来练习上述方法。我们将应用“5个为什么”，揭露说谎的根本原因，并讨论其解决方法。以下是谎言家族的邪恶四人组：

- 滥用“做完了”这个词
- 含糊地表达尴尬的考核评语
- 粉饰给客户和上司的进度报告
- 否认机构重组的传言

好好找一下自身问题

假设你的开发团队应该在星期一完成所有的功能开发。到了星期一，你在团队里巡视了一遍，每个人都说，“我做完了。”后来，你发现半数以上的功能都Bug成堆，四分之一的代码没有处理错误情况，没有辅助功能，也没有经过压力测试。你可能会问：“为什么我的团队水平越来越差了？”但更好的问题是：“为什么我的团队要说谎？”让我们来问5个为什么：

- **为什么我的团队谎称已经做完了？他们想隐瞒什么？**他们要赶一个最后期限，如果不能达标，就会降低他们在团队中的地位。所以要达标只要说：“我们做完了。”很简单。
- **为什么只说做完了，但没有具体解释？有什么危险？**没有人想让自己难堪。遗憾的是，说“我做完了”对个人来说没有任何危险。因此他们为什么不说谎呢？危险留给了整个团队。这才是真正的问题。
- **为什么会发生这种事情？你能避免吗？**问题在于，你没有给团队定义一个可验证的“做完了”的标准。这给欺骗留了一扇后门。为了避免它，你需要对“做完了”做一个清晰的定义，并且得到团队的认可，使用客观的手段去验证是否真的做完了。
- **为什么你不对“做完了”做一个清晰的定义？你需要再做些什么？**当你和你的团队对“做完了”的定义和验证手段达成一致时，你就要把工具准备到位。假设这个定义是：单元测试的覆盖率达到60%，并且95%的测试用例能够通过，另外还做了一个三方的代码审查，并找出其中80%的Bug。现在你还需要做的是，提高代码覆盖率，并在创建好的系统中为单元测试配置自动化测试套件，同时在一个合适的时间为审查员安排一次审查过程，然后就是做代码审检。

- **为什么你还坐在那里？**你需要的大部分东西都可以在工具箱中找到——除了首先你必须要有揭开“我做完了”的勇气。关键是要找准欺骗的原因，从源头上解决问题。

> **作者注：**工具箱是微软内部共享工具和代码的仓库。这些工具可以用来评测代码覆盖率，进行单元测试，甚至为代码审查计算 Bug 抓取率。它们中的很多都已经集成到了 Visual Studio、Office 在线模板等产品中。

> **译者注：**Bug 抓取率 = 已经找到的 Bug 数 ÷ 估计的 Bug 总数。关于“代码审查”，可参阅本书第 5 章的“复审一下这个——审查”栏目。

给我个坦率的回答

你的一位员工工作一向很出色，考核分数你准备给她打 4.0，你也这样告诉了她。但你的部门开了一次协调会议，那位原本能得 4.0 的员工只能得 3.5 了，因为相对于同部门的其他同事来说，她的绩效还不够好。你可以很轻易地对你的那位员工说：“我觉得你应该得 4.0，但是你也知道，考核系统是相对的，我不能总是给你该得的分数。”

你在说谎，不是因为你说的话不对，而是因为你没有在这个过程中扮演好你的角色。让我们再来问 5 个为什么：

- **为什么不尽到你的责任？你在隐瞒什么？**你喜欢这位员工，不想责备她。
- **为什么不责备她？有什么危险？**你的员工可能就不喜欢你了，甚至离开你的团队。
- **为什么会发生这种事情？你能避免吗？**你是传话者，你的员工感到无助，你也帮不上忙。为了减轻负面影响，你需要告诉她如何去争取她想要的考核分数。
- **为什么你不早告诉她？你需要再做些什么？**你需要知道为什么她得了 3.5，而那个人却能得 4.0。

> **作者注：**基于绩效的差异化薪水支付，这种做法在高科技行业中饱受争议。就拿这个数字考评系统来说吧，微软已经把它的过程改了很多次，即使这样，它仍然是基于把你的工作跟其他做相同工作的、具有相同职责水平的人去比较的原理。管理者要做的事，就是要理解这些规则，并且清楚地解释给你的员工听，告诉他们如何通过善意的比较去提高自己。

- **为什么你还坐在那里？**找出分别得 4.0 和 3.5 的员工之间的差异，然后告诉你的员工。她会对如何提高自己有个清晰的方向，并且控制那个提高过程。当然，她仍然可能不开心，但至少你帮助了她，她也知道了自己该做些什么。

给猪抹口红

对照时间表，你的团队的进度已经落后了。你有一大堆的 Bug 要去修复，根本来不及。你的客户和上司要求知道项目的状态。但你没有如实反映情况，而是给他们描绘了一张美丽的蓝图，好像你的团队有足够的时间去赶上进度一样。你变成了一个懦弱而无耻的人，你除了对此感觉不佳外，你还应该做些什么呢？下面是 5 个为什么：

- **为什么要孤注一掷？你在隐瞒什么？**你不想难堪或者让别人来干涉。
- **为什么害怕被责备？有什么危险？**你担心你的项目会因为你的无能而被搁置，或者转交给其他人去负责。
- **为什么会有这种事情发生？你能避免吗？**如果你的客户和上司在全无征兆的情况下发现你们要延期了，他们将不再信任你能继续胜任现在的职责。为了避免这个问题，你应该让你的项目状态透明从而避免给他们带来一个冷不丁的错愕，并且给出一个可靠的计划，使项目回到正常的轨道上来，这样才能赢回你的客户和上司对你的信心。
- **为什么你不一开始就让项目状态透明？你需要再做些什么？**经常收集你的团队的状态，然后公布或者通过 E-mail 发送出去，这给你的工作增加了大量的额外负担。替代方案是，你可以毫无掩饰地在你的 SharePoint 站点上公布你的项目时间表和 Bug 数据，并要求你的团队直接在上面更新，以便所有人都能看到。使用图表来清楚地表示进展情况（或者没有进展）。当数据更新之后，通知一下你的团队。所有人对项目状态都有了统一的认识，你就能按照计划把项目带回到正常轨道。
- **为什么你还坐在那里？**没什么难的。项目状态透明使工作有条不紊，同时你也赢得信任，而信任是事关你能否成功的关键要素。

看看所有这些传言

机构重组的传言漫天飞舞。你的产品单元经理曾经交待过你不要声张，但期间你的团队开始胡乱猜测。不可避免，这个话题在你的团队会议上被提了出来，但你对是否知情矢口否认；你告诫大家流言的祸害性，并提醒大家需要把精力集中在他们当前的工作上面。然而，你心存内疚，终日担心有一天整个团队都知道你当着他们的面说了谎。

> **作者注：**产品单元经理（Product Unit Manager，PUM）是微软内部的第一级多领域管理层。产品单元经理通常负责一个大的产品线（比如 Office）上的某个产品（比如 Excel）。他们也可能负责一个大产品中的一个重要组件，比如 Windows 的 DirectX。机构重组（Reorganization）通常从最高级的管理层开始，然后在随后的 9～18 个月内慢慢地向下扩展开来。关于机构重组，我在第 10 章的“我是怎么学会停止焦虑并爱上重组的”栏目做了更多的论述。当微软公司向一个高效的组织结构演化的时候，产品单元经理是一个很稀有的角色，我同样将在第 10 章的“我们是否是功能型?”中论述。

- **为什么否认这些传言？你在隐瞒什么？**主要是因为你的上司交待过了要你保密。你不想你的团队比你的上司更能胡乱猜测。
- **为什么担心对传言的胡乱猜测？有什么危险？**你担心你的团队被流言卷入太深，从而影响到正常的工作。另外，有些团队成员甚至可能因为害怕不称心的改变而选择离开你的部门。
- **为什么会有这种事情发生？你能避免吗？**大部分团队成员，尤其是那些高级成员，知道有时候重组会变得很糟糕。然而，没有人（包括你）知道重组是否真的会发生，也不知道最后会重组成什么样子。因此，你团队的顾虑根本就没有事实依据。

- **为什么你的团队仍然在为传言焦虑？你需要再做些什么？**出现这种情况的话，问题恰恰就在你身上。你对传言太焦虑了，刻意不把你所知道的告诉你的团队。你应该知道，迄今为止，在曾经计划的重组中，真正发生的其实大概只有三分之一。
- **为什么你还坐在那里？**这里的解决方法很简单，也很明显：说真话。“对，我也听到了很多传言。我们在员工会议上讨论过了。不过，在重组真正发生的那一刻之前，最起码，没有任何人知道是否真的会有一次重组。大部分计划中的重组不会发生，而因为我们的胡乱猜测延误了正常的工作，那我们就太愚蠢了。”

我想知道真相

我对人们是否应该永远说真话并没有定论。否则，别人会觉得我很虚伪，就像在我的丈母娘问我对她的装饰的看法时，会给我惹来一身麻烦。

然而，我们为同一家公司工作。你不应该在业务问题上对你的同事说谎。谎言掩盖问题，但问题其实应该暴露出来。如果你觉得你必须说谎，问问自己为什么。然后再次问为什么，直到你找出真正的问题所在。人们一直想知道如何去实现“可信计算”的第四根柱子——“商业诚信”。现在你应该知道了吧！

2008 年 9 月 1 日：“我得估算一下”

尽管“你的任务估算是怎么产生的？”这样的疑问总是列在诸如“不要对你的项目经理或同事抱怨”的话题之首。但当我与刚出道的工程师们讨论问题时，首先的话题不是估算，而是职业发展和一些大众话题。这就是为什么问题总是在高谈阔论中变得无法控制。估算就是在预测未来，有太多的问题是未知而不可预见的，因而想为一个精神错乱的暴君提供一个精确的估算简直不可能。是不是？肯定是这样的，对吧？

错了。估算来自于软件工程师在规范的基础上对最小细节的把握。要身体力行。其实很简单，有非常多看似不同的方法会为你的完工时间提供精确的预测。所有这些方法皆源自于一个简单的概念——上一次用了多长时间，那么这次也是这么长时间。没有比这更容易的了。

通过对比之前的工作你已经明白现在的事是怎么回事了。但这还不够，真正的问题不是任务估算，真正的问题是接受这份估算。估算很简单，让人接受可不容易。

> **作者注：**很多咨询顾问、研究小组及试验项目把精力花在估算上。我敢肯定他们会跟你说，估算很精确，他们都专注于用技术来避免缺陷。一切搞定之后，最麻烦的是让人相信这个估算是对的。这是最难办的事，但这对于精确度却是最重要的。

没有人会接受这样一个计划

让我们假设一下，在执行很多项任务时，你确实记录了一下这些任务要花去你多少天时间（你是这样干了——你邮箱里邮件的日期就说明了这一点）。让我们再进一步设想，你以之前所用的时间作为今天执行类似事情的估算（你已经变得精熟多了）。你的项目经理将会有什么反应呢？

我想，那将是："哦，算了吧。你在要我呢？"

这只是玩笑话。让我们更深一步探讨。设想，你对你的项目经理说你的估算是来自于对之前项目艰难度的总结。他们不相信这种委屈的理由又是什么呢？以下有三条：

- 上次跟这一次不一样。
- 你第二次本该要比第一次快。
- 上次发生了少见的令人痛苦的事情。

让我们逐一驳斥这些强词夺理的说法。

完全两码事

对于来自于上一个项目经验一成不变的计划表数据，你的项目经理拒绝它的第一个借口就是：上次不一样。什么都变了，或许开发环境及工具都变了；或者设计变了，意味着工程程序也得变；或者需求变了，管理变了；或许月亮跟土星的相对位置也变了呢。

撇开这些借口不讲，只有两个因素对你的估算有那么点影响——工具和工程程序的变化，其他因素对这次开发来说都影响甚少，或者说都微不足道。

就算工具及工程程序的变化可能将极其明显地提升你对估算的准确性——工具的变化要必须使实现端对端功能缩减 1/5 的时间，工程程序的变化要必须使一周的工作量减少数天。不然，其对于估算的影响不过算是一种干扰。

看看吧，万变不离其宗。搞定它。

> **作者注**：我们假设一下，一项任务需要花去你两个星期时间，这个时间有一两天的偏差。因此，工具及程序变化如果只节省一天，这对于两个星期来说已经不重要了。

我越来越棒了

第二个拒绝你的计划表数据的理由是：你这次本该会比上一次干得好。但有意思的是，这仅仅是因为你这是第二次。问题在于你不是干同样的项目（尽管希望是）。仅有的相同点是你所用的工具、工程程序、项目规模以及软件工程的一般性任务。

你对软件工程的一般性任务本应该很稔熟，所以你更了解上一次的项目细节对于估算下一次的项目是没什么作用的。当然，如果你是学校刚毕业的新生，那么用在第二个项目上的时间当然比第一次要少。

如果你改变了工具及工程程序，你的进展将会变得很慢，因为你是第一次使用它们。或许它们变化很小，或许带来效益不错，这当然很好。但不要欺骗自己，它们同样带来了难题。

你确实很想以一个相同的模式对当前项目与前一个项目进行比较。比较越到位，这次估算就越精确。但两次估算方法的最大不同在于，它们从哪方面进行比较。

哦，不。不会再发生了

你的项目经理拒绝你的计划表数据的最后一个理由是：令人痛苦的事只在上一次恰巧发生了。比如意想不到的安全补丁，或者功能模块比起预期的复杂得多，组织结构及关联工程的重置，更不用提类似暴风雪、地震这样的事了。对的，地震，你根本没办法预测地震。

就算你有测算恐怖地震这样的本领。还是有出人意料的补丁、功能模块、组织重构及灾难性事故在项目的每一个环节中等着你。通常，随机事件会时而发生，但是它们的影响并非你所想象的不可预测。感谢李雅普诺夫的中心极限定理，随机事件的总体影响服从于统一均值分布。不管上次花了多少时间，这次花的时间也差不多一样。就是说，只要你不自以为这一次不一样（译者注：概率上讲随机事件分布是服从一定规律性的）。

旧瓶装新酒

好了，我们已经论证了你的项目经理会拒绝你提出的估算方案。因此，他们一方面强迫你去设计个你认为根本不可信的可笑方案，一方面责备你动作慢没有早些拿出这样的估算方案。

我们已经知道，精确的估算非常注重细节。重要的问题是，“你怎么将你认为细致入微的精确估算变成让你的项目经理愿意信服的那样?”

假设你的工作时间是可以把控的——这些时间是排除了像地震、查看 E-mail、浴室漏水等事件干扰之外的正常工作时间，那么你将按小时而不是按天数设计你的估算方案。没有了这些琐事的分心，你的估算方案看起来将更精致、更可信，就算它跟原来的还是没什么两样。

按小时估算看似有点难，因为你手头没有这么多详尽的数据。但是，你确实可以通过一些简单的方法，如计划扑克法（或者团队 DELPHI 法，它更精确、更具灵活性），从而快捷、简易又精确地按小时安排你的工作时间。

计划扑克法是由三个以上的工程师围着一张桌子各自私下对同一项任务做出自己的估算。他们同时出示各自的估算从而不会彼此施加不当的干扰。如果大家的估算是相符合的，事情就算完了。如果不一样，最高及最低估算者解释一下他们的原因，在团队中讨论他们的想法，并不断重复直至达成共识。这个过程将各种潜在问题的可能性摆上桌面。

当你们出台了一个可信的按小时计量的估算方案后，还是不能得出结论说完成某项任务要用多少时间，而只是说你在一星期内需要花多少小时用在一项任务上。即便除去假期、培训及大型会议时间，大多数团队只将不到一半的工作时间花在项目任务上，其他的时间他们都在开会、回复邮件、午休等。如果你的项目经理不相信，很简单，只要花两个星期的时间让这个团队记录他们的工作时间，数字是不会骗人的。

作者注：即使除去假期、培训时间、线下会议以及其他安排好的非任务时间，大多数工程师团队只用了大概 42% 的时间在他们的任务上。你可以通过多种方式，如不收发邮件，不开会，或者让功能团队合署办公及自我管理，或者可以使用诸如 Scrum 迅步法加强团队专注度，从而为你的任务增加几天或几个下午的时间。

我目前的团队使用“事务监控点”代替按小时计量任务。这个概念很简单，你可选用一种监控点计量法代表任务平均长度，比如 8 个这样的点。通过这种任务平均长度来估算其他相关任务。大多数团队使用块状长度来估算，而我使用斐波那契数（1，2，3，5，8，13，21，34，…）。几星期后，你计算一下你的团队一个星期可以完成多少个监控点。这就是你团队的速率。你可以在实际操作中更新这种速率并利用它来精确地将事务监控点转换成天数。

你的成果可能各种各样

就像我之前提到的，有一种稳定的随机性或“变异性”贯穿于整个项目的各个环节。虽然它们有平均值，但每个估算都会因为有标准误差而出现状况。标准误差是基于百分比的，它的大小决定于估算范围的大小。因此，为期两天的估算可能左右误差就一两小时，为期两星期的估算误差就在一两天时间了（同样可能多一两天或少一两天），而为期三个月的估算误差就将是一两星期。

只要你不要过于乐观，随机性并不会产生波澜。在项目完成时，整个项目的标准误差跟各个个体任务的标准误差差不多是一样的。如果你太过乐观（说“下次肯定花不了这么多时间”），你的标准误差就会不断增加，而不是平均值了。

我的观点是，估算两天的任务有多少分钟的误差或者是三个月的任务有多少天的误差并不重要。重要的是你需要一个数量级的时间估算，就像在几年前我的第一篇文章中说的：“开发时间表、飞猪和其他幻想。”

我要相信

你应该怎么估算？与同事们专注于你手头上的任务或是通过计划扑克法或团队 DELPHI 法等手段将这些任务按优先等级排序（计划扑克法对理解一般任务有帮助，DELPHI 法则对其他有用）。这只是相对容易的部分。

真正的问题在于最后让人相信并接受你的估算，然后按计划进行。如我之前所述，过多的承诺是愚蠢的，它的要命处在于会导致“向死亡进军”（参看前述内容）。死亡行军是一个重要的风向标，它表示管理是脆弱的、无力的、充满谎言的和不负责任的。

计划安排出现的问题如恶魔一般，但是可以避免的。如果你适当地安排工作的先后次序，先搞清楚什么是最首要的，你就会在计划执行中减少压力。当你基于你的估算做出可行性的承诺时，就会让你的客户及合作伙伴感到可信，从而建立信任，提升你的团队及公司的名誉。要有个好的估算是相对容易的，但相信他们并相信自己是个挑战。

2009 年 5 月 1 日：“一切从产品开始”

说我“老顽固”，但我相信产品决定一切。有个好点子还不够，光努力也不够，几近完成也不够，你必须得最终让产品面市。

微软的面试常常这样开始：“你都做过什么项目？”如果你最近没有什么成功案例，他们就会问：“为什么？”为什么？因为你不能提供产品你就不能向客户提供价值，你不把这次的任务完成，下一次你就不能接着干并不断提升水平，你没有客户你就没有客户回馈。

人们常常抱怨他们的升职机会及回报与他们的付出不符。我说：“是的，就应该是这样。”这样会影响产品质量吗？不，你已经为产品设了最低的质量标准并成功推出了产品；那么这样会影响创新吗？不，创新者通常最初冒着低收入的风险来获得与他们成果等价的高回报。

作者注：一些人报怨在微软对于创新思想并不给予丰厚回报。那是这些人并没有成功地使这些创新思想得以实现，能这样做的人已经成为我们机构及技术领军人物。

一切从产品开始。这点特别适用于服务（译者注：指软件抽象层，如 SOA），我也将在接下来的内容里重点讨论此类话题。评论家们声称在这一个以服务为主导的新世界中，微软已经忘了怎么去开发这些产品了。可能吧，但是确切地说比起其他公司所认知到的，微软不是忘了怎么开发，而是根本没想到要开发这些产品。我们需要一些提醒及培训，特别当 IT 业转而面向服务的时候。

作者注：重在产品会导致死亡行军吗？不。相反死亡行军会拖延产品面市。就像我在“向死亡进军”中所说的：“死亡行军源于缺乏计划及勇气。”这对于理解服务的概念非常重要，在服务领域，产品要不断地推陈出新，这样才能保持长期成功。

作者注：顺便提一下，对于这句话：“比起其他公司所认知到的，微软不是忘了怎么开发，而是根本没想到要开发这些产品。”我受到了很多批评。这句话听起来有些傲慢，往往让人感觉微软并不愿听取一些关于开发新产品的意见。确实，I. M. Wright 很自大而且为此很自豪，就像他声称的，在我们最终赶上对手之前，微软并不想听取这些意见，我们之前很多的竞争者也不认为微软有快速并反复听取这些意见的相关记录。有些人可能会说，“那是因为我们公司的规模（足够大）。”但是我们并不是总这么强大。有些人会说，“这是我们的策略。”但是策略并不是无懈可击的，你必须以正确的方法在正确的时间开发正确的产品。这需要耐心并不断学习。

我为你提供了服务

根据评论家的说法，就提供服务这点，微软忽视或失去了多少呢？也许还不至于让你如此确信评论家的说法，但足够引起你的疑虑。让我们带着些许欣喜来消除这种疑虑与哀叹。

疑虑是：

- 服务让你觉得什么都会变得不一样。
- 服务注重数据，而套件产品注重功效。
- 服务比起套件产品更关心安全性。
- 服务在依赖性上存在严重问题。
- 服务比起套件产品需要更高的质量以及更快的更新周期。

哀叹与欣喜是：

- 服务不是只在单个客户端上运行的，它是存在于几百台机器上的。
- 服务必须有自我扩展功能。
- 服务容易变换，而不像套件产品那么难。
- 服务能即时升级以迎合人们的需求。
- 服务是实时的，多变的。

让我们拨开迷雾，从褐红鲱鱼开始。

那是什么味儿

服务面临的第一条褐红鲱鱼个头很大："服务改变了一切。"就像我在"你的服务"（见第6章）所讲的："完全不是那么回事。"就像其他所有产品一样，服务始终致力于帮助客户实现他们的目标，你得始终关注客户体验及他们所期望完成的目标，不然你就失败了。就是这样。

接下来的三条褐红鲱鱼是——注重数据、安全问题以及依赖性问题。虽然之前已经花了我们不少心思去理解这些问题，但这与实现套件产品是一样的。无论在软件客户端还是服务器端，为了使数据格式的变化不被客户发现，你必须避免这种情况发生；在现今，不能说你的计算机产品或服务足够强大而不会受到攻击——你必须保证它们的安全；最后，如果你认为客户端的外部依赖性不存在问题，你就不会需要过多的驱动程序。我不是说这些不是什么真正的问题——而是说，对于服务来说这些问题不新鲜也不特别。

最后一条褐红鲱鱼是再寻常不过的，服务实现与套件产品生产的不同之处在于其高可用性与互联网响应时间。看吧，套件产品经常会出故障或者当要正式使用它们的时候总要重启计算机，这样太糟糕了，出现像这样糟糕的事情已经有些时间了。服务同样要注重这样的质量要求，虽然很多服务产品也时常失败。

关于互联网响应时间，在10多年前引入Windows Update后，这个问题就出现了。如果你认为这些补丁不过是修补一下安全问题，就不会引起太多关注。不管是服务还是套件产品，不断增加的客户体验问题在客户向我们提供报告之后如果能快速地得以解决，这样对于客户来说他们感觉会非常棒。

然而，不是说逐日逐月地、渐近地提升客户体验就行了。不管服务还是套件产品都需要典型的、打包完整的升级包来为客户提供突破性的价值。Facebook不想像Vista一样逐步地升级为Windows 7，从而也使自己逐步升级为Twitter。因此，你必须关注客户真正要实现的是什么，而有些时候这些改变不是那么迅速的。

> **作者注：**了解发布产品的最好途径就是早发布，常发布。要让每个程序的"构建"都是可发布的"构建"，每天构建，且每星期至少对整个系统重新构建一次。定期发布技术预览版及测试版。定期发布将使产品逐步更新修复。早发布，常发布，生命在于运动。

> **译者注：**褐红鲱鱼，在18～19世纪的不列颠海域，鲱鱼被用做一种鱼饵，在那个年代没有冷藏技术，一般用盐腌制或烟熏后保存，烟熏后的鲱鱼就呈褐红色并有一种刺鼻的味道从而迷惑猎物。

这里有太多问题了

然而，并不是所有与产品实现相关的问题都适用于服务实现，还有心态、过程管理及团队协调需要你特别处理。

首要的是，服务在分散于全球的多个数据中心上的成千上万台计算机上运行。有时候它们的功能与数据会有重复，有时候又不一样。通常，它的规模程度与可靠性亦步亦趋，这样会暴露出

设计与同步性问题，很多书籍已经对此做过相关介绍（再读一次这些书也不会有什么新发现）；其次比较严重的问题是调试与部署问题。

为什么调试一项服务这么困难？计时问题可谓是在多台机器、多个处理器的多个线程上的杀手。哎呀！然而，这还不是最要命的。

在你调试一个问题的时候要做的第一件事是什么？分析栈，对吗？对于服务来说，栈分散在服务器与响应方之间，这使它几乎不可能跟踪一个特定的用户行为。好消息是，有一个工具可以有助于将分布于各个计算机间的行为进行关联；而坏消息是，这同样还不是最棘手的问题，最棘手的问题是你总是在一个实时的环境中调试，你得不到符号、断点或者逐步跟踪代码中每一步的能力。

至此，让我们回顾一下。调试服务意味着在没有栈的、没符号的或在动态代码中没有断点的多台机器上调试繁琐的计时问题。只有一种解决方案——使用测试设备，有很多这样的设备，这样的设备是从一开始就针对某种服务设计好的，它预先考虑到了你以后将在一个无法进行栈跟踪、没有符号、没有断点的动态机器上调试。

他们增长得太快了

解决调试问题给我们带来了另一个重大困难——部署问题。部署应该是完全地自动化及轻便的。就拿安装程序时文件复制来说，要快速地复制文件，就意味着不用注册，不用用户干预，甚至不用指导用户干什么。

为什么部署需要这么快速又简单？有两个原因：

- 你正在将软件安装在运行着的、遍布全世界的成千上万的机器上。安装必须在无须人为干预的过程中进行，稍有复杂就会引致失败。记住，在 1 000 台机器上花五分钟就是三天半的时间。最好不要出什么问题。
- 服务器的数量需要根据负载量动态增加或减少。不然，你就是为了满足高要求而在浪费硬件、浪费电源、浪费制冷以及带宽。因为你的规模决定于负载，它会随时变动。当负载变动的时候，你就需要自动且迅速地扩充系统。

令人庆幸的是，有 Azure 可以为你在部署时分担重担（所以用这个就可以了，不用另外再开发类似工具），不过，你还是需要设计好你的服务以便于安装时快速复制文件。

人生太无常

有很多问题你可以预测，但不能预测的呢？服务在时刻不断地变化着。有些服务固定不变，它们保持着你的数据（像 Facebook 及 eBay），而有些服务却不会（像搜索引擎及新闻）。稍稍几分钟的宕机就会让你失去数千的客户，数据损坏或丢失可能让你失去上百万的客户。他们马上会换地方，我们的竞争对手会很高兴地接受他们。这是把双刃剑，你必须努力去招揽新客户并留住他们。

当你升级你的服务时，客户就会马上体验到最新版本的功能，而不是要过好几年。如果有一个 bug 在一个客户的上千次体验中发生一次，那么意味着这个 bug 也会在几千个客户身上发生（大数定律）。这样你就必须及时解决此类问题或者进行事务回滚。不管怎样，到星期五再更新服务绝对是个坏主意，而是应随时有个应急回滚按钮以应对不测。

最后，应该认识到服务是实时的、变化着的事物。你可能会想，因为服务器是我的，是在我的构想中设立并配置的，那是一个可控的环境——那只是在你把机器开关打开之前。一旦服务器

开始运行，这些都变了。内存使用在变动，数据及其布局在磁盘上也会变化，网络流量发生变化，系统负载也发生变化。服务像条河流而不是块石头，你不能刻舟求剑。必须随时关注它们，为使你生活得更轻松，要始终谨记五“重”自动化——重试、重新开始、重启、重新镜像、重置机器（虽然重新替换有时需要人为干预）。

面对这些让人心烦的事，客户们很期望得到一些宽慰。一个理想的方法是建一个“点子检测”平台，因为你可以看到客户的不同看法并看看他们日常关注的是什么；同时应具备这样的能力：马上加以改进并在以后找出会间歇性诡异发生的 bug（谨记五“重”方针）。

返璞归真

现在你明白了。围绕客户及其目标的传统的稳健代码编写方式是发人深省的。

然而，如果缺少成果面市，这些什么也不是。要成功就要先有成果。是的，我们的质量标准已经有所提升，我们不是要王婆卖瓜，所以我们需要定期改进客户体验质量，不管是长周期还是短周期的；我们必须通过互联网、PC、电话去实践体验。我们必须服务好客户，让他们开心从而使他们更靠近我们。这是个长期的过程，但千里之行始于足下。

2009 年 9 月 1 日：“按计划行事”

我的大儿子会开车了。但也为我的生活平添两份担忧：一是我已渐感年老，二是颇为我的儿子担心。为了减轻这第二种担忧，我和我的妻子设了一个强制性的禁令：我的儿子如果回家晚了，他必须先给家里打个电话。有一天，他回家晚了 20 分钟又没事先通知，我们生气了，我的妻子愤怒是因为他晚了 20 分钟，而我愤怒是因为他没事先打电话通知。

为什么我的儿子没有打电话说要晚点回家？因为，跟我妻子一样，他只把注意力放在计划上。在回家之前，（不打电话）他可以避免纷争。他说：“我已经尽我所能早点赶回家了”——为此都好几次违反交通规则。但我的儿子没有明白要点，规矩的目的是想减少风险，而他对此做出的反应是增加了风险。

软件工程师也经常这么干。从一个开发计划开始，非人所愿的问题就随之发生，然后他们就延期交工。为了避免纷争，他们往往并不向项目经理提及延迟的事，而是匆忙赶工，牺牲质量，草率应付计划，所有这些都使工程陷于不可控也不透明。其结果跟项目经理所料恰恰相反，而那些项目经理是严格按部就班执行计划的。为什么？因为多数项目经理及工程师不能区别这两种计划——兑现承诺及风险控制。

> **作者注：**我很喜欢这个专栏，它囊括了关键性及基础性的问题，虽然这些问题通常被误解。我希望我的家庭、我的朋友、我的同事及我的团队都读一读。

谁懂事物具有两面性，谁又不懂？

的确，有两种类型的时间表及项目管理技术。

- **兑现承诺。**你向客户或者合作伙伴做了承诺，你必须遵守承诺按时保质完成任务。要按时。

- **风险管理。**工作有关键性及期望性之分。人们或许会做出错误的选择从而产生问题，你必须善于进行风险管理以保证关键性工作的完成。

这两种项目时间表及项目管理技术往往会被搞混，为什么？

- **它们通常同时发生。**承诺贯穿于整个工程项目。但它是由许多个小任务组成并需要风险管理。
- **两者都有时间表。**不同的是向客户承诺的计划时间表不可精确确定，但所有人及事情又为其所驱使。而风险管理的时间表可以有很多监测点以保证其踪迹可寻。
- **它们都可称为计划。**很多人不知道二者的区别。
- **人们所被告诫的往往只是承诺。**当适龄儿童入学后，他们就因为有时间表及规矩显得中规中矩。当他们日后学习了项目管理，就只知道甘特表及里程碑，而把风险管理抛在脑后。
- **风险管理往往是自我学习的过程，是非正式的。**只有一小部分人真正学习过风险管理，大多数人只是在大学里为应付大工作量的任务才向同僚们效仿了一下。我们不是将所有事情及时完成，而是成立工作组，只关注关键性工作，以及极力减少有损我们评分的可能。

不能很好理解这些道理的惨痛结果是导致可怕的决策、可悲的工程实施以及计划失败。你必须知道二者的区别以及将正确的计划应用到相应的问题上。让我们从兑现承诺开始。

这是你唯一承诺的事

“兑现承诺”是跟他人合作的基本准则，也是多数商业行为的准则。在需要内部相互依赖、外部相互信任的情况下，如果产品没有按工程日期安排及时到位，你们的工作就无法协调进行。因为承诺是前后关联的，你必须及时完成承诺之事，不然你就会面临惨重失败。

假如你女儿的生日快到了，你承诺说将给她买一款她喜欢的新游戏，如果别人承诺了游戏出品时间而又没完成，你会是什么感觉呢？在开发者、厂商、销售商以及你之间是手手相传的链条，所有这些都及时到位才能确保在你女儿生日的时候让她开心。

当然，这对基于 Web 的产品来说是相对容易的，但是在团队及部门间相互关联的过程中还是会产生同样的问题。承诺并交付产品才使信任得以建立，及伙伴关系得以维系，失信则相反。虽然很多软件项目只需要很少或根本就没有团队协作，而运作大型成熟的项目通常就需要守信以及其他项目管理技术。

> **作者注：**为了有助于做出承诺，一个公司通常使用库存、保留余地以及其他风险管理的形式为实现承诺的每一步骤提供保障。这就是风险管理的精髓。你使用正统的项目管理方法确保整体目标的承诺得以履行，而使用风险管理确保履行承诺过程中的每一个步骤无恙。

你不认为那是一种风险吗

风险管理是关于如何确保关键性工作得以完成的论题，即便是在一个极易变动的环境里。Scrum 及 Feature Crews 就是两个在软件开发中比较突出的例子，它们关注于风险管理，确切地讲，风险意味着你不能高效地将有价值含量、高质量的产品提供给客户。

就像我在第 1 篇专栏“开发时间表、飞猪和其他幻想”中提到的那样，开发计划及测试计划都在风险管理之列。所有的标志性日期都是降低风险的检测点，但只是仅有的几个跨团队的同步

检测点（标志性里程碑）才是向客户承诺的要点。

什么才是风险管理的重点、先后次序以及其表现状态，这还没有确切的说法。只要注意什么是重要的，先把重要的做好并在条件改变的时候跟踪其状态就可以了。注意，紧盯着细节工作日程安排并不重要，你在规定的时间以规定的质量完成最关键的任务才是重要的，其他的可以不用太在意。

这就是为什么你要告诉你的工程师，就像我告诉我的儿子一样："是否按时回家并不重要，告诉我们你不能按时回家了才是重要的。"如果你知道风险的存在，你只要管好这些风险就可以了，工作时间表只不过用来提醒你计划需要更改了。

当然，你不能过了向客户承诺的时间（一个标志性里程碑）而将关键性任务一直向后拖延，就像我的儿子不能在外超过凌晨 1 点钟，不然我们将没收他的驾照。禁令应事先设置，它防止了可能发生的这种灾难，这就是禁令应具有的功效。

作者注：有很多众所周知的风险管理技术。这里顺便列出几个在软件开发中用得到的（好几个在前面的专栏提到过）：

- 开几次常务会议，15 分钟左右，谈谈进展情况、未来的期望及目前遇到的一些难题（称为 Scrum 迷的 Scrum 会议）。
- 为所分配的任务做个备案（在缺乏经验的人与富有经验的人合作的情况下）。
- 留点余地——安排些额外的时间以备不测（就我个人而言，我向来不喜欢这种方法；我宁愿为任务列一个优先次序表也不愿有面面俱到的想法）。
- 轻在承诺而重在成果——也可以说是量力而行。
- 留有退路——始终谨记为有风险的任务设个预案，比如削减功能模块或重新使用老版本。
- 平衡风险——当事务发生变化时，通过增加或移除风险始终使你的项目保持在一种常态：有点担心但并不恐怖。比如，如果一个团队成员的父母过世了，你的风险就增加了，所以你必须削减具有风险性的功能模块或重新将那项艰巨的任务分配给一个高级工程师。

选择一个正确的工具

所以，当你开始对一个计划付诸实施之前，先等一下，思考一下计划都有什么。其中是否有一连串的约定时间及产品需要交付？或者有哪些不同优先级的重要任务你需要跟踪监视并防止其错过的？

就比方说我儿子回家的禁令，不夸张地说，那就是我不容许他破坏的任务。因此，我们需要风险管理以及需要及时关注其状态的重点，而不是说关注什么时候回家这样的事。这样的模式很适用于大多数软件开发项目。你只是想你的工程师及时告知你情况，这样如果他们延时了，你可以进行调整。太精细是不必要的，而且会适得其反。

然而，如果你跟多个团队及合作伙伴实施一个大型项目，那么在一定级别的层次上，你们相互间的承诺及同步是很重要的。在这个层次的时间表上都是一个个里程碑以及经典的项目管理工具。

不要将高层次的工作计划与低层次的任务相混淆，如果你对待低层的相互间约定像对待高层的一样，你的工程师就会抄近路并快速赶工，你可能就会导致他们破坏整个关键性任务，从而破坏了你高层的客户合约，风险管理也就无从谈起。你在合适的层次使用合适的方法，你一晚上都会睡得很好的。

> **作者注：**更多关于传统项目管理技术与敏捷项目管理技术组合的话题，参见下一专栏“敏捷的团队合作”。

2010年5月1日：“敏捷的团队合作”

我用Scrum已经7年了，而后面6年我一直写关于它的文章。Scrum的概念太吸引人了——规则多变、自我管理的团队在短而固定的周期内周而复始地进行一系列小环节（或功能）工作，并不断提升水平。很多微软团队的成功都来自于此。让人犯昏的是高层的项目经理与团队层的Scrum工程师仍存在严重隔阂。

很多高层及中层的项目经理认为Scrum是混乱的、随意的、危险的并毫无意义的，会使大家对计划失去信心；而很多Scrum迷认为项目计划是种浪费，会引起混乱，毫无价值，只是让不切实际的高层管理者设计个看似完美的计划表以自我满足。结果怎样？他们都错了，而认为他人无知就是愚昧者自作聪明，那么就是，双方都无知。

Scrum是按计划加载的，比起我在微软所见到的其他项目管理方法，它更高效并精确地跟踪数据（除了TSP，在少数团队中使用）。同样，高层项目计划对于项目规模的成功把控、协同合作及萌生宏观上的创造力是很重要的。如果你着眼于小处，Scrum就足够。如果你只想比你的竞争对手提供更低质量且不用太顾及客户价值的产品，或者只想在你局部的范围内微观管理所有工程状态，那么只要项目计划足够就可以。如果你着眼全局并想高效地提供高质量且丰富的客户价值，那你就需要在高级项目计划与Scrum间找到平衡。

> **作者注：**功能小组一个有趣的Scrum变体。这种组织方式最先源自于Microsoft Office的项目开发。就像Scrum团队，功能小组是规则多变并能自我管理的团队，在一个很短的周期内他们周而复始地进行一系列小环节（功能模块）工作。虽然他们可能也会开些常务性会议，但他们并不仅仅是照搬Scrum模式——固定周期迅步法及频繁的重复规划。不过，功能小组已经成为了很多微软团队工作的一种主要方式，它们能高效地执行一项简洁的软件工程过程。

我尊重你否定我的权力

为什么在高层的项目经理及Scrum迷之间会存在如此隔阂呢？几年前，在该章对于管理失当的介绍中我已有所提及。

项目管理在不同的规模、不同的抽象概念中有不同的表现形式。有下面几种：团队或功能层次（大概10人左右）、项目层次（50~5 000人为一个特定版本工作）以及产品层次（由执行官领导的多个版本开发）。敏捷开发在团队层次很适用，传统方法在项目层次中很适用，而长期的战略性计划方法在产品层次很适用。然而，人们很少同时在不同层次工作，实际上，每个人总是

会分年段地在这些层次间工作。所以人们常常认为某一层次的高效方法应该应用到其他层次上，悲剧就这样产生了。准则就是：小型、紧凑的团队跟大型松散的组织运作方式不同，相应地应选择适合你的方法。

你不能期望注重过程管理的传统方法很好地应用在小团队上，就像你不能期望动态的应急计划应用于大型机构。如要在其间搭起桥梁，你必须明白各方的目标及它们相互间的需求是什么。让我们来摆平它们。

计划什么都不是，也什么都是

高层次项目计划（愿景、架构、时间表以及风险管理）就是：

- **设定一个共同不变的愿景使大型的机构有序运作。**没有一个共同不变的愿景，你要保持长期的成功只能是碰运气。是的，跟客户进行反复沟通是会带来可观的价值，且可能跟企业长期价值方向一致，但主要还是要依赖于共同不变的愿景。
- **做好团队间对接使愿景具有可行性。**一个共同不变的愿景使企业机构间拥有一个共同的目标。当然，你不用地图或高速公路就可以从西雅图开车至纽约——但是那样会花费你很长的时间才到达那儿。
- **实现对合作伙伴的承诺。**大型、成熟且成功的公司需要有一个与合作伙伴的良好关系才能使承诺得以实现。如果你希望愿景能带来现实价值，你就需要伙伴公司的合作。这就意味着承诺是双方相互的，金钱就存在于你与他之间。
- **发现并解决可能会破坏承诺的问题。**很多基于伙伴关系的大型工程项目包括了互相依赖性、风险性及双方的合作。很多项目可能夭折或最终失败，甚至于已经实施行将大功告成的项目。你必须为成功或失败做好准备，在项目受挫前发现及解决问题确实是项艰难的工作。

项目规划者、构建者以及中间管理者需要 Scrum 团队来有效协调以适应共同的愿景，遵守他们之间的协定（或者相应地调整协定），完成他们间的承诺（或者相应地更改约定），并且在问题发生时把它摆上桌面并解决它。

我会照顾好自己

好的，我们已经有了共同的愿景、相互间的协定、承诺以及一个风险控制计划。现在我们只需要照单行事，是不是？你是来自哪个空间的？在我的世界，变化无时无刻不在，工作进展很可能就在其中一步戛然而止。

在上层的抽象概念中，计划可能达成一致并让人感觉良好，但是在底层，还有很多细节需要处理。当着手于细节的时候差异性及复杂性就来了，项目规划者、架构师及中间管理者就会不断通过召开进展情况及工程设计会议，再不断增加项目经费，发现瓶颈问题等方法从微观层面应对变化，从而在不断的摸索中艰难前行。或者他们就信任工程师让他们自己从细节处解决问题。

敏捷方法如 Scrum 能快速高效调整以适应突发细节并改变状态。这种流程及项目经费的精减有利于工程进展及客户关注的满足工程进展需要的解决方法得以实现。这并不是说 Scrum 团队不制订宏观计划——只是说他们是禀着一种不断努力的态度，步步为营，以实现共同愿景。

Scrum 团队需要项目计划者设定一个共同愿景、相互间的协定以及指定必要的资源，然后就放手让他们干。愿景一旦达成，多方协定得以遵守，资源到位，项目计划者就放宽心了。自我管理的 Scrum 团队可以快速解决大量问题，项目计划者、中间管理者必须学习、接受它，还需要及时包容。

作者注：很多项目规划者及中层管理者对于要信任一个自我管理的团队感到很担心，有三种办法消除这些恐惧：

1）持续跟踪数据——快速浏览一下与估计值相当的数据负载，浏览一下优先级别、工作的进展、遇到的问题、工作的效率及已完成的工作。

2）每星期或每天召开一次 Scrum 会议——只要 15 分钟，与所有的 Scrum 团队主管讨论遇到的问题及预审进展。

3）为 Scrum 团队的决定设置底线，当某项决定影响到两个以上 Scrum 团队的时候，就必须由项目团队或架构团队来审核。

太高兴了

项目计划者与 Scrum 工程师应该互相包容（精诚团结）。他们完成各自的任务。项目计划者为 Scrum 团队设立了方向，Scrum 团队使项目计划者只用专心关注于整体规划，而他们以一种快速的、灵活的、反复的过程提供了高质量的、可用性的、面向客户的具体信息。

项目计划者与 Scrum 工程师在一个大型的具有宏观视野的项目中扮演着重要角色，担心或拒绝对方的工作是不可原谅的。我们对双方相互间的角色及目标理解得越透彻，那么我们就越能为客户带来令人愉悦的新颖体验。

作者注：当你没有权衡好项目规划与自我管理团队之间的关系时，会出现什么情况呢？明摆着的是，苹果及谷歌提供了很有趣的例子。

苹果自顶层开始以项目规划为导向，然后在基层进行微观管理。其结果是他们的产品很有创意，但限制了他们商业的纵深度。

谷歌则全是细碎化的自我管理团队，其结果是快速地提供了类目繁多、品种多样的服务，但这些服务之间彼此脱节，缺乏一致的能凌驾于公司常态之上的战略图景。

虽然这样说有些言过其实，不过还是很有启发性。当然，苹果与谷歌已相当成功，即使他们有这样的缺陷。在不变的项目规划及高效的自我管理团队之间取得平衡，使得微软对于他们更具竞争力。

第2章

过程改进，没有灵丹妙药

本章内容：

2002年9月2日："六西格玛？饶了我吧！"
2004年10月1日："精益：比五香熏牛肉还好"
2005年4月1日："客户不满"
2006年3月1日："敏捷子弹"
2007年10月1日："你怎么度量你自己？"
2010年10月1日："有我呢。"
2010年11月1日："我在缠着你吗？Bug报告。"
2010年12月1日："生产第一"
2011年2月1日："周期长度——生产力的老生常谈"

我错了，好吧？我什么都不懂。现在也请你冷静下来！有些人把个人行事方式提升到了一种信仰的程度，这就是我对经验主义的理解。斯金纳指出，当一种动物，比如鸽子，通过它的一次偶然行为产生了合意结果时，这种"经验"就产生了。人也一样，因为一次偶然的机会通过自己的能力产生了一种满意的结果时，人们就会沉溺于这种非常特定的做法。

译者注：斯金纳（Burrhus Frederic Skinner，1904—1990），新行为主义心理学的创始人之一，操作性条件反射理论的奠基者。他创制了研究动物学习活动的仪器——斯金纳箱。1950年当选为国家科学院院士，1958年获美国心理学会颁发的杰出科学贡献奖，1968年获美国总统颁发的最高科学荣誉——国家科学奖。

这么做并不总是错的。我跟大家一样也会热衷于经验主义。但人们只热衷于运用某种方法（如极限编程）而不会变通时，这时的经验主义轻者是无效率的，重者将使整个项目分崩离析。

本章分析了大量的过程改进的方法与技巧来消除经验主义。第一个栏目摘自于一期《Interface》，这期《Interface》着重探讨了六西格玛法在微软的应用（很多主题文章是由《Interface》的编辑安排的）。接下来的文章是关于精益软件工程、需求描述、敏捷方法实例、依赖关系管理、精益BUG管理方法、服务的持续性部署及缩短开发周期的。

很多优秀的书籍对这些主题的探讨比我在这里说的要深入得多。如果我对这些概念的阐述不能使你满意，请原谅。毕竟我并没有试图让自己完美，而只是追求正确。

2002年9月2日："六西格玛？饶了我吧！"

我很抱歉。如果你跟我谈论另外一种全面的、持续的质量管理改进计划，我可能会抓狂。还好，我们只是想尝试一下六西格玛法。

仅仅需要两个月的时间，你就能被培训成为六西格玛绿带。或者坚持学满4个月，你就能成为六西格玛黑带。我想如果是我就快坚持不下去了。

译者注：六西格玛（Six Sigma）管理方法是为了获得和保持企业在经营上的成功，并将其经营业绩最大化的综合管理体系和发展战略。它比以往具有更广泛的业绩改进视角，强调从顾客的关键要求以及企业经营战略焦点出发，寻求业绩突破的机会，为顾客和企业创造更大的价值；强调对业绩和过程的度量，通过度量，提出挑战性的目标和水平对比的平台；针对不同的目的与应用领域，提供专业化的业绩改进过程：产品/服务过程改进DMAIC流程和六西格玛设计DFSS流程；在实施上，由勇士（Champion）、大黑带（Master Black Belt）、黑带（Black Belt）、绿带（Green Belt）等经过培训的、职责明确的人员作为组织保障。通常，一个六西格玛过程改进项目的完成时间为3~6个月。

我只是不明白，为什么我们要借用玄虚的名词和空手道术语来比喻这些成功的工程实践方法呢？就好像高层管理者把他们的智慧、受过的教育和经验都统统抛在了脑后，而沉迷于现今流行的计量分析这种小伎俩，认为这样将解决我们落后生产力的所有弊病。

作者注：我经常在我的栏目中表示对管理层的不满。除了项目经理外，其他管理者是我最喜欢奚落的对象。由于他们向来广受好评，微软管理者从来不把这种奚落当回事。不仅如此，他们中的很多人还以此为乐。当然，我的上司偶尔也会被问及，是否这些栏目有足够的建设性。不过，我写了6年相当有争议的主题，没有一个栏目受到过管理层的审查，或者被勒令修改。

但我在微软的工作还是要受命于这些管理者，因此无论我喜不喜欢，我都得看看那期《Interface》上讨论六西格玛的文章，及一些六西格玛网站上的资料。我放弃了我的立场吗？没有。充斥于这些文章里既新颖又让人兴奋的思想改变了我们产品的生产方式吗？也许吧。那它到底有点价值没有？当然是有的。

啊！这是什么巫术?!

六西格玛是一个结构化的问题解决系统，它有一大堆的方法用于分析和解释在商务、开发、制造过程中的各种各样的问题。实际方法本身并没有什么创新，比如头脑风暴、5个为什么、因果图、统计分析，等等。这些方法多年来一直被用于工程和商务领域来寻找问题的根本原因。

这种方法论是基于这样一种原则：通过试验，最终找出并解决真正的问题，归结为所谓的六西格玛DMAIC流程：界定（define）、测量（measure）、分析（analyze）、改进（improve）、控制

(control)。这种质量改进的基本循环步骤，其实微软每一个产品部门在稳定化阶段都在用。Bug 的界定（微软通过规范书来描述）、测量（微软会通过测试发现 Bug 并将其存档）、分析（微软有 Bug 分诊会议）、改进（微软称其为 Bug 的修复）和控制（通过回归分析、排定优先级和再一次的分诊控制）。

那么，为什么要设立一个六西格玛的部门呢？为什么要成为一名绿带呢？在公司里面放 20 个全职的六西格玛黑带又有何意义呢？

召集骑兵

通常，我们在危急关头会惊慌失措，会把以前学过的工程知识和积累的实践经验忘得一干二净。同样，当我们被压力击垮时，当我们面对满目的 Bug 觉得事情无可救药时，我们往往会茫然不知所措。

因此你叫来了身边的绿带，或者直接请来了黑带大师。他会提醒你应该马上采取什么措施。然而，因为六西格玛的高度结构化特性，因此，此时不能带有个人情绪化的冲动。

那些搞六西格玛的人能够冷静地观察真实的数据，找到问题的根源，并且采取有效的措施去对问题进行改进，而不是指责、臆测和莽撞行动。然后，他们会给你一个流程用于跟踪你的改进，并且控制它的影响。

为混沌建立秩序

是的，任何具有良好工程背景的人都同样能找到问题并且解决它们。任何人，只要他通过了微软的面试，应该都具有足够的智力去找到一个问题的解决方案。但有时当你陷在问题中太深或者情绪不稳定时，你需要一种冷静的外部因素来给你施加影响，从而帮助你专心做正确的事情。

另外，那些搞六西格玛的人对全公司范围内的各种方法和最佳实践都有接触和了解。他们能够给你的部门带来其他部门的经验，并且提出当时你怎么也无法想到的、非常相关的解决方案。

这么说来，我是六西格玛的支持者了，是吗？不是。我仍然认为绿带、黑带这些玩意儿有点可笑，而它的方法论也是对“全面质量管理”（Total Quality Management，TQM）和“持续质量改进”（Continuous Quality Improvement，CQI）的重复。然而，微软还是选择了六西格玛法——如果在问题失去控制的时候，能有一个部门马上介入并给予帮助，我觉得这也不错。

> **作者注：**尽管六西格玛在微软的产品开发中从未得到过重用，但在紧急关头能有个指导者及相关团队的想法还是挺不错的。我本人也曾是这种组织里的一个管理者。

2004 年 10 月 1 日：“精益：比五香熏牛肉[⊖]还好”

曾经走在一个公共场所，比如机场的出入口通道或公园，是否突然有一群狂徒冲你而来，想要教化你或者恐吓你甚至就因为你貌似傲慢无礼而要揍你。跟他们当中的任何一个人讲理，逻辑和推理都失去意义。在他们眼里，只有疯狂的信仰和不容辩驳的真理。即使你完全赞同他们，也不容你质疑或理论。你必须相信，你不能怀疑，哪怕一点点。

这让我很不爽。我的意思是说我真的巨不爽。我有我自己的头脑，我想完全独立地思考。不是只在聚会或者社交场合，而是所有情境下。问一下

⊖ 五香熏牛肉（Pastrami）是一种加入各种各样的香料熏制成的牛肉。——译者注

“为什么”并搞清楚“怎么做”，这才是人性独立的要义。

你可能会认为，我这样的见解应该成为不明就里而无法进行软件调试的开发者的标准。但就像一些致力于宗教、政治斗争和环境保护的人所具有的狂热一样，软件开发者也强烈地推崇一些新兴的软件开发方法，比如极限编程（eXtreme Programming，XP）、敏捷方法和团队软件过程（Team Software Process，TSP）。

做任何事情都要适中

我非常喜欢这些开发模式所提倡的思想和方法。但面对一个虔诚的追随者，如果我问为什么要这样做，或者当我建议对规则或实践做一个很小的改动，以使它更适用于我的实际工作的时候，那就不一样了！这好比把一只魔戒摆在哈比人的面前——他顿时露出獠牙，发根直立。对于一些开发者来说，极限编程和敏捷方法已经成为一种迷信。而对于另外一些开发者来说，团队软件过程是一种忠诚度的风向标——你跟我们非友即敌。

请原谅我的务实，原谅我使用自己的头脑，原谅我不迷信灵丹妙药，而坚持去做实用的事。我不会因为“你必须得这么做”而去做。我的行为原则是，要对这样做会成功而用其他方法就会失败这两方面都能有相当充分的理由。

> **作者注：**这样的“夸夸其谈”经常使大家笃信我在文章中论及的主题思想，我以前总以此为傲，但这次不会了。只有极端主义者才说“迷信”无害，我可不是极端分子。

俭则不匮

是什么让我关注“精益”呢？是的，本栏目的标题。虽然在极限编程、敏捷方法和团队软件过程中有很多奇妙的东西，但至少有一个概念它们是共同的：减少无为的浪费。这恰恰是精益设计和制造的重点，而精益是源自于丰田汽车公司的一个概念，它比极限编程、敏捷方法和团队软件过程要早30多年。但是，极限编程、敏捷方法及团队软件过程使用不同的路子处理浪费问题，通过了解精益模式，我们可以更好地理解这些模式方法是怎么实现的。

因此，冒着触动一些狂热者敏感神经的危险，听我娓娓道来。精益的精髓就是以最少的投入换取向用户提供最大的价值。它采用一种“拉模式”及“持续改进”的方法来做到这一点。拉模式其实很简单：“不需要，就不做”，这就减少了无用的、不必要的、无价值的工作；而持续改进则重在减少浪费和提供一种平稳的客户价值流。

> **作者注：**非常感谢Corey Ladas，因为是他首先把我引入了精益（lean）、公理设计（Axiomatic Design）、Scrum、质量功能展开（Quality Function Deployment，QFD）、集合设计（Set-Based Design）、Kaizen改善、普氏概念选择法（Pugh Concept Selection）的世界。除了这些理论，他还有很多很多其他出色的想法。我们曾经在一起共事了两年，后来他离开了团队，之后他的位置再也没人能够补上。他现在和另一位出色的前团队成员Bernie Thompson在一起，维护一个精益软件工程的网站。

精益理论对有损于客户价值流的浪费归纳为以下 7 种类型：

- 过量生产
- 运输
- 多余动作
- 等待
- 过程不当
- 库存
- 缺陷

这些显然是制造业的术语，是吗？它们不可能跟软件有关联，是吗？显然你太天真了。所有这 7 种浪费都直接跟软件开发有关，我视其为软件开发“7 宗罪”。以下我将谈谈如何通过极限编程、敏捷方法、团队软件过程及一般性常识来规避它们。

过量生产

第一种浪费是供过于求，但愿这永远也不要发生。是否有一种软件产品符合需求规范而无需删减任何功能模块呢？是否有一种软件产品存在客户从来用不着的功能模块呢？这些问题既复杂又平常，即宽泛又耐人寻味，既似无关紧要又劳人伤神……过量生产太恐怖了，它会导致难以置信的浪费。

极限编程通过短而紧凑的循环周期来解决这个问题。它注重与客户的持续交流，以及开发者之间的持续沟通。这保证了所有人都知道其他人在干什么，并且总是有较高的客户认可度。结果，几乎所有完成的工作都是对客户有价值的。当然，微软的客户是庞大的，因此微软的很多团队都使用敏捷方法。

敏捷方法是各种精益方法的一个总称，它包括极限编程。确切地说，敏捷方法不是指某项具体的技术，而是一种整体的方法论，它为软件开发提供了很多有趣的方法。其中之一称为 Scrum 项目管理方法（Scrum 的命名来自于一个橄榄球术语）。开发团队定期地跟客户代表碰头，通常每 30 天一次，以展示工作进展、重新安排工作的优先次序以及进行过程改进。跟极限编程一样，团队成员也每天开会更新各自的进度和讨论工作中遇到的难题。

通过每月对工作优先次序的重新安排和每日对工作的重新组织，一个 Scrum 团队把自身的关注点锁定在了客户看重的东西上面，几乎没有任何工作是多余的。通过定期性的过程改进，价值流可以不断地被优化。

深入探讨

当然，你也可能只会死板地运用 Scrum 和极限编程：你先在“基础框架”上花费工夫，而让客户苦苦等候着他们想要的功能。为了定期获得客户的反馈，快捷的周期循环要有个基本前提：开发应该“深度优先”，而不是“广度优先”。

确切地说，广度优先是指对每一个功能进行定义，然后对每个功能进行设计，接着对每个功能进行编码，最后对所有功能一起进行测试。而深度优先意味着对单个功能完整地进行定义、设计、编码和测试，而只有当这个功能完成了之后，你才能去做下一个功能。当然，两个极端都是不好的，但深度优先要好得多。对于大部分团队来说，应该做一个高层的广度设计，然后马上转

到深度优先的底层设计和实现上面去。

这正是微软的 Office 功能小组的工作方式。首先，团队对他们需要哪些功能以及如何使这些功能协调工作做出计划。然后大家被分成各司其职的多个小型团队，每个团队自始至终一次只负责一个功能。最终结果是，一个运行良好且稳定的产品快速地交付给用户进行试用。

> **作者注：**当然，功能小组（Feature Crew）的想法并不新鲜。然而，在一个像 Office 这样庞大而活跃的生产环境中，能够找到一种方法去实现精益软件开发已经是一个重大的成就了。你要知道的是，Office 系统现在已经拥有多种桌面应用、服务器应用和大型的在线服务。

深度优先通过把注意力放在将被使用到的工作上，而不是可能永远不会被客户关注或者面面俱到而永远得不到定论的“基础框架”上，这样就能减少过量生产。另一个出色的深度优先开发方法是“测试驱动开发”（Test-Driven Development，TDD），但我想在后面“过度开发”那一节中再展开讨论它。

运输

第二个极度浪费是期望万事俱备。在制造业中，这通常是指零部件的运输问题。对于软件来说，这个“运输”指的是团队之间可交付成果的传递。这里有 3 个令人厌恶的运输问题之源：创建、分支和 E-mail。

- **创建：**创建花费的时间越长，浪费的时间就越多，这是我想让你引以为戒的。极限编程和敏捷方法都坚持每天创建，而这条规则他们很可能是从微软学去的。但对于大型团队来说，每天创建越来越不现实。幸运的是，我们已经安排了优秀的人才来解决这个问题，但这确实是个大问题。这点不用多说。
- **分支：**我喜欢 Source Depot。它对整个公司的价值是巨大的。但它也像一只宠物象一样让人失去兴趣：当它还小的时候非常可爱，但经过几年的喂养后，灵活性就逐渐丧失。建立代码分支是个很好的主意，因此很多大的团队都这么做了。现在假设你在 A2. B3. C1 分支上工作，而你的伙伴在 A3. B1. C2 上实现了一个关键功能或者修复了一个重大 Bug，那他们首先需要把 C2 横向集成到 B1 里，再将 B1 横向集成到 A3，而你必须将 A3 纵向集成到 A2，再将 A2 纵向集成到 B3，将 B3 纵向集成到 C1。天哪!!! 等所有这些集成都做完了，黄花菜都凉了。这种做法还仅仅是你当前周期内产品生产线的一个分支。

> **作者注：**Source Depot 是微软用于管理上亿行源代码和工具的大规模资源控制系统，包括版本控制和分支管理。

- **E-mail：**最后一个运输噩梦是 E-mail 通知单。项目经理告诉开发和测试人员规范书准备好了；开发人员告诉测试人员说编码完成了；测试人员告诉开发人员说他们的代码有 Bug；开发人员告诉项目经理说他们的设计变更有问题；还有我的个人问题：与客户、依赖方或者零售商之间的沟通，特别是越洋沟通。极限编程和敏捷方法通过废除这种形形色色的岗位角色和每日召开会议来解决这个 E-mail 通知单问题。但对于外地零售商和依赖方，这行

不通。我们只有在可行的情况下才使用自动通知功能，在必要时才用 Live Meeting，以及通过 E-mail 给对方一个明确答复来减少 E-mail 的恣意蔓延。

行为

第三个极度浪费是仅仅为了找出问题而花时间。在制造业中，这是对机械和人工行为的一种浪费。在软件行业中，就是把时间花在研究做什么、怎么做及怎么调整原先方案上。糟糕的搜索技术是行为浪费的一个典型例子。不可测试、无法维护、无法管理的代码同样是一种浪费。

设立中断及调试参数有助于快速找出 Bug 及减少浪费。设计复审、代码复审、代码分析和单元测试都能达到减少浪费的目的。极限编程居然建议“结对编程”（Pair Programming），但我个人认为，这样做是对资源的一种浪费（除非开发人员是在一起学习一个新的代码库）。团队软件过程可用于检测你所有工作及弊病，你可以清楚地了解到你的时间是怎样花掉的，因而你也能大大地减少你的行为浪费。

> **作者注：**在一些陌生领域，为了开发一些新鲜的玩意儿，我的团队已经采用了“结对编程”法。它相当有效。

一个特别讨厌但能避免的行为浪费是复制 Bug 修复信息，因为代码注释需要同时变更，Source Depot 需要重新集成，Product Studio 需要重新整合以及收到邮件后要相应地进行工程改进。每个人都浪费劳力去管理 Bug 和项目进度数据的多个副本。有一些工具可以让这类事情变得简单，信息只需输入一次，便能自动被传播到其他所有的地方。这类工具可以有助于减少无谓的行为。

等待

第四个极度浪费是等待。前面谈到的运输问题只涵盖了创建、分支集成和及时沟通方面存在的一大部分等待问题。但等待远远不止这些地方。最常见的盲区是，团队在功能的优先次序上达不成一致，或者即使达成了一致但没按照既定的顺序去实施。也就是说，项目经理如果胡乱地写规范书，开发人员就不得不等待；如果开发人员胡乱写代码，则测试人员只能等待；如果测试人员胡乱测试，那么所有人都必须等待。

极限编程、敏捷方法和团队软件过程都强调要求团队确定一个统一的工作次序，并且得到客户或者负责人的认可，然后再按照这个顺序依次开展工作，直到他们决定重新审定工作次序为止。团队软件过程在这方面尤为严格，但如果没有一个临机应变的带头人，这样做也不会有可持续性。

此外，不稳定的代码也会导致等待。只要代码不稳定，测试团队就不得不等待，就像其他需要等待用户反馈的事情一样。极限编程和敏捷方法特别注重经得起考验的稳定代码，这是深度优先策略的又一个要点。

> **作者注：**其他形式的等待是因为部署新的服务或有其他类似事件的发生，整个服务系统环境需要稳定化或进行同步。一种避免浪费这种等待时间的最好方法是公开操作过程，并逐步更新部署。我将在本章后面的专栏“生产第一”中讨论这个内容。

过度开发

第五个极度浪费是开发过度。你会经常遇到这样一种情况：编制过于复杂的功能，在已经运行良好或并不称得上瓶颈问题之处对性能要求过于吹毛求疵。这种浪费跟过量生产有关，但这里更强调在具体功能的实现上面。

药方：测试驱动开发。这是一种为实现既定设计方案的极限编程及敏捷开发方法，它在实现了单元测试的同时实现了代码全覆盖，一石二鸟。其过程相当地简单：

1. 创建 API 或者公共类方法。

> **作者注**：这是我和敏捷社区的一些成员起争执的一个地方。你是在写单元测试之前还是之后写 API 或者公共类方法？偏执狂说之后写，但我认为要之前写。两者不同之处在于，前期方案设计的工作量，以及你的代码依赖方与你之间的关系。我会在其他栏目再次谈及前期方案设计，我相信，在 10 万行代码级别的项目中适度的前期方案设计是成功的关键要素。

2. 按需求写一个 API 或类的单元测试。
3. 编译并创建你的程序，然后运行单元测试并确认它失败。（如果成功了，则跳过第 4 步。）
4. 不断修改代码直至其通过为止。（同时要保证以前所有的单元仍然能够测试通过。）
5. 重复第 2 步至第 4 步，直到所有符合需求的 API 或者类都测试通过。

很自然，当你掌握了这种方法的窍门之后，你可以一次为多个需求写多个单元测试。但当你刚刚起步的时候，最好还是一次只做一个。这样能够养成良好的习惯。

当你使用测试驱动开发方法时，你就不必写过多的代码。你也自然而然地得到了很容易测试的代码，并且这些代码通常还是高内聚、低耦合、少冗余的——所有这些都是真正的好东西。哦，我曾提到你也能获得代码全覆盖的单元测试了吗？你还有什么不满意的吗？

库存

第六个极度浪费是没有交付的工作产品。这跟削减功能有关，但它也包括那些正在进展中的工作。当你采取宽度优先的开发方法时，你所有的工作同时开展，直到代码编写完成并完成稳定化。所有完成的规范书、设计和等待通过测试的代码都属于库存。它们的价值都尚未实现。

尚未实现的价值是种浪费，因为你不能把价值演示给客户和合作伙伴看。你不能得到他们的反馈。你不能改进和优化客户的价值流。甚至，如果产品计划改变了，这些尚未实现的库存通常就变成了巨大的工作浪费。

精益拉模型强调只做需要做的事情，因此它的结果就是低库存，这在 Scrum 和测试驱动开发方法中得到了很好的验证。Scrum 特别关注正在进展中的工作，时时跟踪并努力减少浪费。Scrum 同时注重于定期改进和优化你传递价值的方式。测试驱动开发方法要求你只实现满足需求的代码，多则无益。

质量缺陷

第七个极度浪费是返工。这是最明显的一个，也是我过去批判得最多的一个（参见第 5 章）。极限编程和敏捷方法通过各种方法来减少 Bug 和返工，这些方法不仅仅包括测试驱动开发、每日

创建、持续的代码复审和设计复审，等等。

然而，极限编程和敏捷方法也以一种更为微妙的方法来减少 Bug——在边干边学中建立起一种框架。在你为整个产品完成设计和编码之前，通过深度优先开发方法，一步步勾勒出整个项目的整体框架。这避免了严重的架构问题：这种架构问题隐藏得很深，等到被发现的时候已经太晚，不容许再调整了。听起来很熟悉吧？

减少缺陷是团队软件过程的专长。使用这种方法的团队能够使他们的 Bug 率下降到行业平均水平的千分之一。第 5 章的“软件发展之路——从手工艺到工程”会详细论述如何通过团队软件过程进行缺陷预测、跟踪、消除。虽然团队软件过程本质上来说不是精益，但它也并不排斥深度优先的开发方法。

合作共生

下面我该激怒极限编程、敏捷方法和团队软件过程的虔诚追随者了。因为没有理由认为对这些方法的综合运用会比简单地将它们合并的效果更差。使用 Scrum 来完成一个精益、深度优先、灵活、优化的开发计划。使用测试驱动开发方法来创建一个精益实现。使用团队软件过程来分析你的缺陷和工作，这将大大减少你的 Bug 及无谓劳动。这些在某些人听起来可能会怪怪的，但在我看来却是再合理不过的了。

现在我如果能找到一些纯正的五香熏牛肉就好了。

> **作者注：**我在纽约长大。在雷德蒙很难找到纯正的五香熏牛肉了。

2005 年 4 月 1 日：“客户不满”

你总是在伤害你爱的人。我们必须真心诚意地爱我们的客户。我们把满是 Bug 的代码发布出去，尽管这不是什么大问题；我们延期交货，这不是什么大问题；我们对客户需求没有一个深入清晰的了解，并做到客户至上，这不是什么大问题；我们没有跟客户进行很好的沟通并达成一致意见，这不是什么大问题；我们没有很好地聆听客户的声音，然后把信息传递给该得到这些信息的人，同样，这仍然不是什么大问题。注意，真正的大问题是，当我们在折磨客户的时候我们常常浑然不知，也不知道我们做得有多恶劣，而到察觉的时候已经太晚了。

> **作者注：**以上陈述我有些夸大其词，不过是为了增加些戏剧性而已。实际上，我们能够认真聆听客户的声音，并且把这种客户的意见集成到了我们的产品中，这方面我们做得很好。这甚至是多年来微软的一个巨大的竞争优势。尽管这样，随着软件市场的日益成熟，我们客户的期望也在不断提高。因此为了保持我们的竞争优势，我们必须继续改进。本栏目将讨论应客户需求的变动而变动代码的好处。

如果我们发布了没有 Bug、高质量的代码，但它跟客户想要的东西不一致，那么客户会不满意。按时发布客户不想要的代码同样会使他们不满意。即使我们对客户需求有了一个深入清

晰的了解，分清其需求的轻重缓急，我们发布的代码仍然必须符合这些需求，否则客户就会不满意。做好沟通与聆听并不够，如果我们最终无法交给客户他们真正想要的东西，一切都没有意义。

不知者无罪

实际上，良好的沟通和聆听反而会有害于我们。假设我们跟客户交谈了，了解了他们真正想要的东西。客户看到我们这么真诚会很高兴，以为我们真正了解了他们所要的东西。两年之后，我们交给客户一个解决方案，但它达不到他们的期望。结果是：

- 客户很失望，因为产品并没有像他们期望的那样发挥功能。
- 客户觉得受到了侮辱，因为我们浪费了他们的时间，我们燃起了他们的希望，最后又亲手把这希望之火扑灭。
- 客户被激怒了，因为我们没有恪守承诺为他们服务，他们可能再也不会相信我们了。

如果我们一开始就不理会客户，至少我们还可以为这个错误找到借口。我们可以声称“我们不知情”。遗憾的是，我们确实知情，我们的确承认了，我们确实也承诺了。即使承诺没有以合同的形式确定下来、不具备法律效力，但承诺是实事，而我们没有守诺。

欲速则不达

你觉得这不会发生吗？错了！我们总是不恪守诺言。令人惊讶的是，我们居然还有客户。我们的销售人员与客户交谈，告诉他们我们的计划。我们的顾问拜访客户，信誓旦旦地说他们会与产品团队一起提供特定的解决方案。我们的市场和产品计划人员成立专门的工作组并告诉客户我们正在开展工作。

> **作者注：**我说的“我们总是不恪守诺言”是指我们达不到理想目标。我们交给客户当初要求的东西，但并不完全符合他们所需要的。客户在看到产品之前并不知道他们想要什么，这也是为何敏捷方法强调迭代客户反馈的原因。我使用“承诺”这个词，是因为它对微软的员工来说有着重要的意义。我们很容易在我们的产品中犯些小毛病，但就是这些小毛病使客户头痛不已。我想让工程师知道这一点。

我们确实是在开展工作。市场机遇决定了我们产品的计划和远景。但我们与客户之间还没有形成一个良好的沟通渠道，除非已经太迟了，我们无法应客户要求再做改变。当客户在运行着的产品上点击某个按钮时，我们才意识到需要花时间去重新思考我们已经做过的事情，此时的天国之门已经关闭。遗憾的是，大部分情况下，在客户接触到我们的产品之前，我们已经完成了所有的编码。

> **作者注：**这里让我来推介一下产品测试版和技术预览版，因为我在原来的栏目中没有提及它们（这是个严重的疏忽）。大部分微软的产品都发布测试版，但只有一两个，并且通常只在开发周期的后期才发布。然而，一些产品已经开始在早期阶段就频繁使用技术预览和测试版——这种做法我很喜欢。

实际上，当我们的产品出货之后，我们跟客户的沟通渠道做得相当好。Watson 和 SQM 会向我们汇报信息，反映我们的客户在我们交付的产品上正在经历的所有糟糕体验。这已经向前迈进了一大步。我们修复这些 Bug，然后在 3 个月或者 3 年之后再次发布产品，接着 Watson 可以向我们证明这些讨厌的问题是否已经消失。

> **作者注：**当你的应用程序在微软的 Windows 操作系统上崩溃的时候，你会看到一个“发送错误信息”的对话框，Watson 就是支持这个功能的一个内部称谓。（常常会有这种对话框弹出，并确实引起了我们的注意。）SQM 也是一个内部技术名称，它通过匿名收集用户对产品的使用模式和体验，来支持 MSN、Office、Windows Vista 和其他产品的用户体验改进计划。（请你在安装我们的软件的时候加入这个计划，它会让我们知道什么工作得很好，而什么不尽如人意。）

但有些问题会导致我们不能守诺，扼杀我们的商业机会，摧毁我们的客户对我们仅有的一点点信任，这会是些什么问题呢？我们怎样才能在产品出货之前就发现这些问题？我们怎么能避免这些问题的发生呢？

敏捷错觉

说到这里，那些敏捷方法的信徒们可能已经在尖叫了：“使用敏捷方法！”好啊，你试试每周或者每月跟 1 亿个客户开个会看。这并没有看起来那么简单。我不是说敏捷方法没用，而是说你可能有点太想当然了。

当然，你可以让项目经理或者产品的计划人员替代这 1 亿个客户，但他们能代表所有这些客户的准确性，跟你买彩票中大奖的概率差不多。也许你真的会中头奖，很多人都玩彩票，但你可不能把你的生意或者你的退休保障当赌注压在这种偶然的东西上呀！

你需要与客户间建立一条直达的通道，就像 Watson 所做的一样。你写的任何代码都应该对应于一个特定的客户需求、市场机遇、商业需要（比如 TwC）或者用户问题（比如一个 Watson 桶）。这样的话，如果一个特定的问题出现了，或者你需要对你的工作进展得到定期反馈的话，你就知道应该找 1 亿客户中的哪个人了。

> **作者注：**实际上，Watson 并没有帮我们跟客户之间建立起一条直达的通道——我们收到的信息是匿名的，并且进行了汇总处理。我们真正建立的，是一条获取用户问题的渠道。每个“Watson 桶”代表了一个用户问题，这个问题上千，有时甚至上万的用户都经历过了。因此对于一个 Watson 问题，我们不知道该找谁去确认，但我们能够了解到他们的问题到底是怎么回事。可信计算（Trustworthy Computing，TwC）——微软在安全、隐私、可靠性、健全的商业实践方面发起的研究课题，已经从 Watson 数据上为我们的用户带来了巨大的收益。

回头想想

你如何才能把一个特定的客户需求跟一行代码关联起来呢？对于 Bug，这种关联我们已经做得差不多了，但对于功能开发呢？为了解决这个问题，你必须回头再想想：

- 为什么你要写那行代码？它的功能需求是什么？
- 那个功能需求因何而来？客户的应用流程是什么？
- 那个应用流程从何而来？市场机遇或者客户协议是什么？
- 谁定义了那个市场机遇或者签了那个客户协议？他的 E-mail 是什么？

如果你回顾一下你所做的工作发现它并不是客户所需要的，那么你所谓的客户需求，很可能是凭空猜测出来的。可追溯性（traceability）是能使我们的客户满意的关键。

一石多鸟

不过这只是个开始。像所有伟大的东西一样，可追溯性解决的不仅仅是与客户相关的需求问题：

- 可追溯性使我们的客户可以检查他们面临的问题的状态及相关解决方案。现今，不管是服务开发还是产品开发，因为我们具备后向追溯的能力，当客户检查一个错误或程序崩溃的状态时，他们基本上可以做到这一点了。至于从产品定义到产品开发的前向可追溯性，则不管是我们还是我们的客户都已获益匪浅。
- 可追溯性有助于我们权衡业务轻重缓急并促进交易达成，就如功能开发时安排优先顺序一样。因为可追溯性将我们与我们的变动对业务产生的冲击联系了起来，我们可以对一项功能或变动所需的资源量做一个明智的选择。
- 可追溯性帮助我们架构解决方案，确定依赖关系，并且安排组织项目。这是最出乎我意料的一个优点。有了可追溯性，你就可以知道什么样的用户应用流程决定了什么样的功能开发。因此你就能知道为什么会有这功能模块。这有助于建立正确的软件架构及依赖关系，从而采取适当的方法来组织项目。太棒了！

当然，如果没有可追溯性，客户就不知道他们的需求是否能够被满足，以及何时能够被满足；产品部门也不知道真正的业务危机将会是什么，因此只能靠猜测去做决定；各个部门都不知道他们为什么要这个或者那个功能，也不知道它们之间是怎样的依赖关系。我们的生活从此变得混乱不堪，危机重重。

> **作者注：**为了引起注意，我又一次夸大其词，但我并没有夸大可追溯性的好处。对此我欣然接受，我非常喜欢可追溯性。

工欲善其事，必先利其器

那么，如何实现可追溯性呢？理想情况下，我们应该有一个工具，我们能用它来跟踪用户应用过程和需求，就像我们现在跟踪 Bug 和程序崩溃一样：

- 销售人员和顾问能够使用这个工具来记录客户需求、应用过程和承诺。
- 市场和产品计划人员能够使用这个工具来抓住市场机遇，并由此与客户达成协议，并且定义关键性的跨产品的应用。
- 产品计划人员和项目经理能够使用这个工具来整合需求，记录重复性，把多个产品之间相关的应用关联起来，并且在产品的层面草拟一项应用的操作过程、需求和功能规范书。
- 产品部门能够使用这个工具来对功能需求进行分诊，在设计和实现的全过程跟踪各个功能的进展情况。

- 测试团队能够运行测试用例并发现 Bug，于是他们很容易就能明白各个潜在问题的破坏性。
- 客户需求的发掘人能够对客户提出的需求实现进度跟踪。产品团队也会联系他们，要求他们说明相关问题或者提供反馈。当情况有变时，客户需求的发掘人也能主动联系产品团队来更新需求。

查缺补漏

有些部门其实正在使用 Product Studio 来实现可追溯性，但这个工具还不算是个完美的解决方案。在我们得到正确的工具之前，若想做到在整个设计过程中跟踪客户需求和应用过程，我们还有以下几个方法：

> **作者注：** Product Studio 是微软内部的一个工作条目跟踪数据库。如今我们已经将它产品化了，并把它集成在 Visual Studio Team System 中。自从本专栏设立以来的 5 年里，大多数微软的工作部门已经采用 Team Foundation Server（TFS）来跟踪项目进展。我们已经近乎实现可追溯性了，就像我之前所述的那样。

- 当你撰写市场分析报告的时候，对相关的客户协议书做个链接，并确认这些协议书具有相应的客户联系信息。同时也在这份市场分析报告中留下你自己的联系信息。
- 当你在创建高级应用范例的时候，对市场分析报告及客户协议书也做个链接，同时也要留下你的联系信息。
- 当你撰写功能规范书、需求列表或者产品应用案例的时候，将相关的高级应用范例、需求分析、市场分析报告和客户协议书都做个链接。哪个功能与哪篇文档相关一定要明确，不要只建一个冗长的链接清单了事。
- 当你创建设计文档的时候，做个到规范书和其他支持文档的链接。再次强调一下，不要创建一个长长的参考列表——事无巨细地将所有信息都罗列出来，甚至你可能将每个联系人的联系方式都列出来。
- 当你面临抉择或复审工作的时候，通过这些链接追踪到相关人员那里，也就找到了问题的根源。

令客户满意

当下，我们的业务运转方式像小孩子的“打电话”游戏一样。每个人把他认为客户想要的东西告诉下一个人。一路传下去，信息在每一步都会被扭曲和丢失。等到我们产品出货的时候，客户甚至不认得这就是他当初要求的东西。（此时你可能想起了那幅经典的卡通图：客户真正想要的是在一棵树上挂起一个轮胎秋千；但他们最后得到的却是树干被挖出一个洞。）

当产品不断演变，开发不断推进，你必须回过头去跟客户谈谈，以确保你做的决定是正确的。同样重要的是，客户在他们的需求或者应用流程改变的时候，也要有办法跟你取得联系。如果没有一个沟通渠道，这是不可能做到的。

可追溯性把这种渠道建立起来了，但你必须谨记你要做什么并努力去做。期间出现的任何差错，都会使你承受毁约的风险。但如果你做对了，也就意味着你每次都能给客户带去他们真正想要的东西。那就是你辛勤努力的价值所在！

2006年3月1日："敏捷子弹"

我很难做出判断。也许你可以帮我。我不能断定以下两种观点，哪种更糟心：一种观点认为使用"敏捷"方法，并且恨不得微软在全公司范围内采用它，用它解决我们面临的所有麻烦；另一种观点认为敏捷是被一些无知的学者鼓吹出来的，它实际上是一种改头换面的愚昧方法，它让开发者不用承担任何责任。这是个两难的决定，两种观点都会使我有种作呕的感觉。

> **作者注：**这是我最喜欢的栏目之一，因为其间爱恨交错不能自已。尽管它并不完美，但我在这个主题上的评论还是相当公允的。

现在让我们来纠正这两种观点：

- 如果你认为敏捷方法解决了产品开发过程中的所有错误，那你真是愚蠢至极。雇用成千上万个人来开发高度复杂、深度集成的软件给上亿客户来使用，这不是件容易的事。这世界上没人比我们对这个任务知道得更多，包括敏捷联盟的那些聪明家伙。并不是我们现在做的所有事情都是错的，也不是所有事情都能用敏捷方法来满足我们的需要。
- 如果你是极端的反敏捷人士，你认为Scrum是System of Clueless Reckless Untested Methods（一个毫无头绪、不计后果、未经试验的方法系统）的缩写，那你也是个蠢蛋，而且更加无知。不假思索就随便以什么理由妄加否定，本身就是带有偏见且不专业的做法。像敏捷这样的草根运动总是有一些事实基础的，它可以使我们的团队和客户受益。那些概念可能并不一定直接适合于我们的业务，但当你停下来理解它们的时候，那些事实基础总是有一些用武之地的。

> **作者注：**敏捷是在微软整个公司范围内，也就是由一小撮人和一些小团队牵头的一次草根运动。

该是破除敏捷方法的神话的时候了。我还将解释如何去使用这些方法背后的创新思维，以构建我们自己的优势。

真理的敌人

首先，让我们来破除下面这些敏捷神话……

- **传说之一：敏捷就等于极限编程（结对编程、Scrum、测试驱动开发、用户需求描述或者其他敏捷方法）。**敏捷方法实际上是软件开发方法的一个集合，这些方法遵从一套统一原则，但除此之外其他方面都没什么关联性，有时甚至还是对立的。你可以从敏捷联盟（www. AgileAlliance. com）了解到关于敏捷的更多真实信息。
- **传说之二：敏捷方法不适合大型组织。**这种说法很荒谬。敏捷是各种方法的一个集合。这些方法中有一些不适合大型组织，但有一些适合，还有一些如果你创新一下的话也可能适合。在做出一个武断的结论之前，你必须先好好研究一下具体的方法。

- **传说之三：敏捷方法很适用于大型组织。**敏捷哲学崇尚“客户合作胜过合同谈判”和“随机应变胜过循规蹈矩”。但跟 1 亿多个客户合作是艰难的。合同谈判在跨团队依赖关系管理方面是至关重要的。（参见第 8 章的“各走各的路——谈判”栏目。）循规蹈矩对于商业承诺来说是必要的，因为合作伙伴在上百万美元面前会变得难以相处。在大规模项目上应用敏捷方法，要求你灵活而有创造力地去处理好这些问题。
- **传说之四：敏捷意味着不写文档。**敏捷哲学崇尚“能用上手的软件胜过面面俱到的文档”。很多敏捷的狂热者看到这个就认为，“啊，不需要文档了!”这么认为的话只能说你是井底之蛙。敏捷哲学声称，“精益求精。”换句话说，能用上手的软件比文档更重要，但必要的文档对客户、合作伙伴和跨部门的依赖方仍然是有价值的。
- **传说之五：敏捷意味着不需要前期设计。**敏捷哲学崇尚“随机应变胜过循规蹈矩”。很多敏捷的狂热者将其误解为，“没必要思考或做计划，设计到时候将突然出现!”突然从哪里出现？一个放射性污水池吗？这里的要点是说，随机应变胜过过分死板地遵循原来的计划——而不是撞了南墙后再回头。
- **传说之六：敏捷意味着没有个人责任。**敏捷哲学崇尚“个体性与交互性胜于过程管理与工具”和“随机应变胜于循规蹈矩”。很多管理者看到这个感到很恐惧，认为这个意味着完全没有责任可言。实际上，敏捷在这个领域收到的效果恰恰与管理者担心的相反。敏捷使个体对团队负责，而团队对管理层负责。责任性大大地加强了，并且这种独特的哲学理念让敏捷团队更加有效、有弹性并且名副其实地“敏捷”。
- **传说之七：Scrum 是个缩写词。**这是个很无聊的传说，但它几乎让我发疯。Scrum 是最有名的并且使用得最广泛的敏捷方法之一，但它绝不是一个缩写词。Scrum 是根据橄榄球术语来命名的，代表球队聚在一起，手挽手，准备争球。它也是 Scrum 团队每日例会的名字。在微软，我们已经使用了一种类似 Scrum 的方法达数十年，比这个术语的出现要早得多。Scrum 是最简单的敏捷方法之一，也最接近于微软的很多团队在现实中采用的方法。稍后再对 Scrum 作更详细介绍。

拨乱反正

抽象地谈论敏捷只会招来漫无边际的争论，但是实际中的应用才是重点。我们已经知道，敏捷实际上是软件开发方法的一个集合，问题是，“哪种方法适合在大规模项目中应用呢?”很多人对这个问题已经思考过或者撰稿论述过，但写这样的栏目的人却不多。在我给出我的观点之前，请看下面的几条基本规则：

- **能不变就不变。**如果一个团队已经在业务注重的标准上表现得很好，那就没有必要再改变了。改变总是有代价的，哪怕它的结果可能会很美好。你改变的目的只能是为了在最后获得改进。因此如果不需要改进，也就不需要改变。
- **不要陷入太深。**如果改变是必要的，也不要一下子改变所有的东西。让功能团队每次只挑一两样改进，并且留意它们的进展状况。不是所有的团队都要一起改变，也不是所有的团队都要做相同的改变。当然，如果你改变的是一个像创建系统这样的中心服务，那么所有的团队最终都必须接受它。但即使是这种改变，我们也可以选择让它对某个团队公开或先隐瞒。推荐的方法是：一点一点尝试，一点一点学习，循序渐进。

- **区分项目级别和功能级别。**人们感到困惑最大的地方，尤其对于敏捷方法，是在项目级别和功能级别上的差异。在项目级别，团队之间需要严格的日期和协议约束。在功能级别，你…好吧，事实上…管它呢。很多管理者都不能理解这个奇异的概念——你的团队可以在你规定的任何期限内完成工作；问题只是在于在这个日期之前要完成多少功能模块而已。只要项目级别的计划能够被跟踪和控制，那么你的功能团队应该选择任何一种能够让他们工作得最有效率的方法。

> **作者注：**这一段话体现这样一个理念，环环相扣。它给我们的警示是，当组织中的几个小团队使用相似的方法时，通常这个组织会工作得更好。这些方法不需要完全相同，但如果团队步调一致的话，他们会在一起工作得非常好。否则，因为团队之间的预期时间各不相同，结果他们之间的协调和沟通就会变得一团糟。

想尝试不一样的感觉了吗

现在你想尝试敏捷方法了。但也许你只是想安抚一下部门里的那几个敏捷疯子，在他们喝着醉人的 Kool-Aid 饮料时，给他们送上一些 Scrum“小吃”。那么，你应该怎么做呢？你又怎样才能把它最好地集成到正常的工作中去呢？眼下有大量的敏捷方法可供选择，因此我在这里只能谈一谈最流行的那几个：Scrum、极限编程、测试驱动开发、结对编程、用户需求描述、重构和持续集成。

> **译者注：**Kool-Aid 是美国本土的饮料，里面含维生素 C，是美国人在成长时最喜爱的饮品，不含咖啡因，口味多变。

首先，有两个方法在微软我们已经用了十几年了，它们是“重构”和“持续集成”。重构只是简单地重新组织你的代码，并不改变它原有的功能。重构用来将复杂的函数（面条代码）打散，或者在现有代码的基础上增加新的功能，就像把一个只能读取 CSV 文件的类改造成一个抽象类，以便能够同时读取 CSV 文件和 XML 文件。持续集成的想法是，让新代码总是定期集成到完整的工程创建（理想情况下是每天）中去，以便所有人都能对它进行测试。

> **译者注：**面条代码（Spaghetti Code）是一种经典的反模式，如频繁使用 GOTO 语句，用来形容看上去很乱、几乎没有软件结构的一段程序或者一个系统。

让他说话

其次是“用户需求描述”，它们就像是应用流程和一页功能规范书的组合。用户需求描述的想法是，提供足够的信息来估计需要执行与测试什么样的功能特性。

比较麻烦的是，用户需求描述应该是由用户来创建的。很多敏捷方法假设用户经常就坐在功能团队的旁边。遗憾的是，当你有 1 亿个目标用户时这就成了问题。

不管你喜不喜欢，我们需要有人代表用户的角色。像市场、产品计划、用户体验、销售和客户支持部门都可以充当这样的角色。他们对用户大量的调研成果可以写入价值主张和远景规

划书。然而，当对那些高瞻远瞩的远景规划和端对端的应用流程进行细化时，我们在功能级别上仍然能够使用用户需求描述的概念，以提供足够的事实依据来对一个功能集合的实现和可行性进行评估。

合作共赢

“结对编程”是指两个人共用一张桌子和一个键盘在一起编码。它的想法是，当一个人在打字的时候，另外一个人则更有全局观，可以发现设计或实现的不足之处。这一对人常常交换角色。虽然两个脑袋比一个脑袋好，但这样的成本也是双倍的。我更愿意把这两个人分别用在设计和代码审查[⊖]上面，这样价值会更高。然而，结对编程对于熟悉新代码库效果显著，这时通常把一个熟悉代码库的人和一个不熟悉代码库的人结成一对。

> **作者注：**在一些陌生的领域，我的团队曾经为创建一些新鲜的事物使用过结对编程。这种方式相当好！

除了重构和持续集成外，“测试驱动开发”和 Scrum 已经被证明是在微软应用的最简易、最有效的敏捷方法。我在我的栏目中描述精益工程的时候（本章的第二个栏目），关于测试驱动开发我是这么论述的：你首先为一个类定义它的功能和函数，然后给公共函数编写单元测试，再然后才是写实现代码。这是个反复的过程，并且每次都只写几个单元测试和一点点代码。这种方法相当流行，因为这样为开发者提供了所有开发所需的单元测试代码，从而为实现代码提供即精简又高效的设计框架。

先写测试也更有趣。当你先写实现代码的时候，单元测试就成了一种花样翻新的痛苦，而且它只会带来坏消息；测试不能真正起到增强代码的作用。当你先写单元测试的时候，编写代码去适应测试会比较容易，而且当测试通过的时候你的心情会非常愉悦。

> **作者注：**很多业内人士都认为，测试驱动开发的真实目的是一种优秀的实现代码设计方法。虽然我同意这种观点，但单元测试的积极作用不应该被夸大。

测试驱动开发可以和结对编程组合起来使用：一个开发者负责写一些测试，另一个负责写足够的代码来通过这些测试。他们两人的角色还可以常常交换。最后，测试驱动开发让开发者对“什么时候才算是真正完成了实现”有了清晰的认识。也就是，所有的需求都经过测试，并且所有这些测试都通过。

有点极端

“极限编程”是一个完整的开发方法论。它把用户描述、结对编程、测试驱动开发、重构、持续集成和另外一些实践组合在一起，其非常适合应用在跟客户紧密合作的小团队中。

极限编程大量依赖团队知识，以及与客户之间的直接交互，而且几乎没有文档。如果你的团队是独立的，并且你的客户就在你的走廊边上，这种方法会很有效，但在微软这种现象并不普遍。

⊖ 请参阅本书第 5 章的“复审一下这个——审查”栏目。

如果我们不能每年赚到数10亿美元，我们的形势就会很悲惨。然而，就像我已经说过的那样，极限编程中的很多单个方法在我们的产品开发中应用得非常好。

准备玩橄榄球！

最后我将讨论的（可能也是被人误解最深的）敏捷方法是Scrum。人们可能把Scrum和极限编程（它实际上不用Scrum）混淆在一起，也可能认为敏捷就等于Scrum（哈?!）。除此之外，Scrum最令人困惑的部分就是下面这些相关的术语了：Scrum大师（Scrum Master）、产品备忘录（Backlog）、burn-down图、冲刺（Sprint），甚至包括猪和小鸡——这些足以吓跑任何一位管理者。真是极大的误会啊！

无论好坏，Scrum是由一个喜欢有趣名字和故事的人发明的。其实施起来既不复杂，也无争议。因此，除了重构和持续集成之外，Scrum是多年来我们内部一直在做的、最接近于敏捷方法的一种实践，并且我们还有一些重大的改进。

接下去，让我们先来澄清那些令人困惑的术语。“Scrum”是指每天的例会，“Scrum大师”是指功能团队的组织者，“产品备忘录”（backlogs）是一些功能或者工作条目的列表，“实施进程”（burn-down）图用于展示剩余的工作，“冲刺”是指小型的里程碑，“猪”和“小鸡”则是指企业家的农场动物（故事很长，但很好笑）。

这些概念没有一个是创新出来的，但Scrum的确带来了一些大的改进：

- Scrum的每日例会组织得非常好，用于收集有用的数据。团队的组织者（Scrum大师）简单地问所有的团队成员3个问题：从昨天到现在完成了哪些工作（并且花费多久时间），现在到明天到来之前准备做什么（并且告知剩余的工作量），阻碍工作进展的问题。

> **作者注：**跟踪每个任务完成所花费的时间，是我的团队对微软Scrum作的一点小小贡献。通过把这些信息加入到实施进程数据中（还剩下多少工作量），你就可以画出美妙的累积流线图，度量花在进展中的任务和工作上的时间，并且更好地估计团队的生产力。典型情况下，产品团队花在任务上的时间大概为42%，花在沟通上的时间为30%，拿我来说，我花在一起协作的功能团队上的时间有60%之多。

- 在Scrum会议上收集到的数据输入到一个电子表格或者数据库中。基于这个电子表格，你可以分析花在任务上的时间、完成日期、进展中的工作、计划变更和许多项目问题。这种情况下很流行使用实施进程图——一种展示时间和总剩余工作量之间的动态关系的图表。

> **在线资料：**冲刺任务清单（SprintBacklogExample. xls，SprintBacklogTemplate. xlt）

- Scrum大师是一个独立于团队之外的力量。他甚至最好不是组织中的一员，但这通常不太现实。Scrum大师有权力排除阻碍每日例会进程的因素，保持会议的简短。
- 功能列表或者时间表称为“产品任务清单”（Product Backlog），而工作条目列表或时间表称为“冲刺任务清单”（Sprint Backlog）。通过分离这两个列表，管理层可以只关注他们想

要完成的工作（产品任务清单），而团队则只关注手头的工作（冲刺任务清单）。为了保证所有事情都在正常的轨道上，Scrum 大师一般每周跟管理层开一次会（比如每周的主管会议），进行状态更新。

在线资料：产品任务清单（ProductBacklogExample. xls，ProductBacklogTemplate. xlt），冲刺任务清单（SprintBacklogExample. xls，SprintBacklogTemplate. xlt）

- 冲刺作为小型的里程碑，它的时间长度是固定的。当一个特定的时间段用完了，一个冲刺也就结束了。典型情况下，一个冲刺大概 30 天。

作者注：写这个专栏 6 年以来，我在一个团队里采用过 1 周冲刺、2 周冲刺及 30 天冲刺。现在，我们的团队都采用 2 周冲刺，这是我最喜欢的。2 周的时间正合适，而且无需额外的经费——所有的进展情况可以描绘在一块白板上。但是，对于完成一项重要任务来说，2 周有些过长。我现在正打算采用卡班（KanBan）法或一种持续冲刺法，但这又是另一个主题了。

- 每次冲刺结束后，功能团队会跟管理层一起复审工作（不错的改变，哈?），总结本轮冲刺做得好的地方，以及下轮冲刺需要改进的地方（这总比等上 1 年或者 10 年、产品最终出货后再做总结好），然后为下轮冲刺制订计划并重新评估工作条目（改变原先的计划和评估？决不!）。

通过使用每天、每周或每月的反馈机制，Scrum 使团队在一个多变的环境中保持高效的工作，并且富有弹性。通过收集一些关键数据，Scrum 使团队和管理层知道团队的运作状况，在问题还没有成为问题之前就把它解决掉。通过分离管理层掌管的功能列表和功能团队掌管的工作条目列表，Scrum 使团队实现自我指导，工作更加投入，使团队内的每个成员和团队外的管理层都很有责任感。

最后你要知道的

不是所有的敏捷方法都适合于每一个人。很多根本就不适合微软的大型项目。但 Scrum、测试驱动开发、重构和持续集成可以被很多微软的团队使用，并且会有显著的效果。结对编程和用户需求描述的适用程度要低一点，但如果条件适宜的话，应用这些方法也能很有效，只要你不是很心急，一下子走得太远，或者强迫你的团队采用敏捷，应用这些方法定会大有收获的。

作者注：几乎在我任职过的所有地方，我都见到过有管理者强迫工程师改变他们的方法论的情况。这绝对行不通，哪怕像 Scrum 这样很受大众欢迎的东西。管理者可以建议、支持、资助行为方式的改变，但绝对不要强制。

如果你想有更多的了解，可在我们的内部网络或者互联网上搜索敏捷方法，同时也请留意关于敏捷方法的课程。如果你的团队方方面面都表现得很出色，那建议你不要做任何改变。但如果你希望看到更高质量，或者更好的基于功能团队的项目管理，你自己思量一下是否要采取一些行动，尝试一下敏捷方法。

2007年10月1日："你怎么度量你自己?"

在微软，我们唯命是从，但是否应该有自己的思想? 当数十亿的美元投入到生产线上，你最好不要对已做出的决定有什么异议。10年前，我们的产品不是猜想出来的，它们是我们在学习竞争对手成功的产品基础上改进的，我们赢在后来居上。

现在我们在很多领域处在领先地位，没有竞争对手，智囊团们只能凭空遐想。他们的格言是：开发些很酷的玩意儿，希望得到客户的认可。结果是：一地鸡毛，毫无价值或无法理解。

万幸的是，明智的团队不会胡乱猜想。他们依照微软的研究数据及客户数据来认定使客户真正开心或烦扰的因素，并由此提升我们的产品。如果没有这些数据或回馈，我们将陷入绝境。因此，记得通过数据决定我们如何生产我们的产品，那么就会一身轻松。

没那么多尝试

如果出色的团队根据数据生产产品从而消除无端猜想，为什么这种猜想还会左右我们如何生产呢?现今的软件开发过程充满着奇思妙想。"最佳实践"是传统的智慧结晶，过程管理是业内共识，一些自命不凡的敏捷开发方法被当成教条从而取代数据。为什么?

> **作者注：**我最喜欢的敏捷方法，如Scrum及测试驱动开发，都使用数据。测试驱动开发则更甚——它需要以测试通过率为基础。

不要告诉我有数据可以证明某种方法最有效。我不是在谈论某某人的数据，我是在说你的。在发现Bug之前，你怎么知道你的团队使用了正确的方法能得到正确的结果?你怎么知道你们今天就会比昨天更好?为什么你们始终不使用数据来查找问题?

或许是因为软件开发是一种开创性的过程或技艺，它是无法度量的；或许度量就是错误的或容易被不当利用；或许由于你没有充分的数据进行判断；也或许怯懦的狂热者们只愚昧地膜拜于他们所迷信的陈规戒律，他们因为太胆怯而不敢（利用数据来）度量，也太蠢不知道怎么度量。

仅仅把心思放在成熟的好方法上是不够的。如果没有这么多资金及相关人员投入到产品线上，且你又对如何以一种正确的方式使用正确的度量方法一无所知的话，你是无法经受住生命的洗礼的，伙计。幸运的是，你不用非得刨根究底先了解它。

有什么问题

我曾听闻："你们这些蠢货，你不知道软件度量很恐怖吗?你不知道脑子锈豆的管理者将利用它们来使你与你的同事们，或者让你的团队与另一个做着不同项目的团队相争吗?你不知道它们只是用来冒险一试，而陷你们的工作及客户于水火吗?"是的，我知道。我们早已知道你们对如何恰当使用度量方法一无所知，但既然你提到它了，那就让我们来打消你的顾虑：

- **软件是一项创新性的工艺，是不可度量的。**就像我在第5章中说的："软件苦旅——从工艺到工程，"工艺对于制造桌子与椅子很管用，但对于一座桥，一个心脏起搏器以及

软件来说就不够了。总之，你忽视了其中的区别。**准则一是：不要想着怎么度量，而是度量什么。**

- **软件度量是错误的，也容易被不当使用。**有些人会这样说："你度量什么就会来什么。"如果你对每行代码都进行度量，人们就会写出更多糟糕的代码；如果度量修复了多少 Bug，那他们就会留下更多的 Bug 需要修复。**准则二是：不要对中间环节进行过多度量分析，只对其期望结果进行度量分析。**
- **有充分的数据能对所有事情做出论断。**计算机能产生非常大的数据量，而软件开发是在计算机上进行的。然而，如果这些数据反映的是更多问题而不是答案的话，那它就是没用的，不管那些图表看起来有多美。**准则三是：不要只是收集数据而已，使用度量分析解决关键问题。**
- **只有管理者使用数据而不是你。**管理者的慵懒众所周知。如果数字告诉了你该做什么，有什么必要把你的想法提交给管理者呢？出色的度量分析工作不是告诉你该怎么做，因为好的度量工作并不是解决怎么做的问题（记得准则一吗？）。**准则四是：不要使用度量分析来做决定，而是通过度量分析来使你明白需要做出个决定。**
- **管理者往往做一些不恰当的比较。**管理者的无知也众所周知。他们"高屋建瓴"，认为软件是软件，Bug 是 Bug，完全不拘细节。只把注意力放在期望结果上或许有用，但这往往会带来不正当的相互比较。**准则五是：不要只对原生度量进行比较，要有一些基准及范例，它们提供了参照细节。**

> **作者注：**我那些在 Google 及新兴网络公司的朋友会说："这很简单，把管理者干掉就是了。"好主意。把"管理者"换成"执行官"或"产权人"，你会面临同样的问题。

现在就让我们按照以上准则按部就班。

怎么回事

准则一：不要想着怎么度量，而是度量什么。

人们都讨厌被迫按某一模式工作。没错，他们会乐于接受一些指点及建议，他们也愿意接受一些约束及要求，但是没人愿意成为一个（模式化的）机器人。

当一个人开始一项任务时，可以肯定他们已经知道有办法可以更好地解决问题。强迫人们按你的方式工作，而不是他们自己的，那当你的方式比他们的差时，这种不良后果就可能发生。这样就会让他们产生挫败感，他们会憎恨你，蔑视你，认为你愚蠢。

如果对你希望事情怎么做进行度量分析就等于说你要求人们怎么做。这样就把你树立成为了一个让人憎恨又蔑视的蠢人了。我可不建议这样。

相反，你要对希望完成些什么进行度量分析而把怎么完成留给明白人。只要说明你希望一段脚本运行良好，而不是度量需求规格完成情况、功能点情况或还剩下多少 Bug（这都是关于"怎么做"的），而是把这些脚本分解成片段再分析有多少脚本片段及这些片段之间衔接起来是否能按预期运行。理想情况下，你需要一个客户来当你的评判，但如果有一个独立的测试者也可以了。

最后，你得感谢我

准则二：不要对中间环节进行过多度量分析，而只对其期望结果进行度量分析。

软件度量被过度利用了。我们都了解软件度量，大多数人都用过。为什么？因为管理者掌控着软件度量标准。如果你没有达到其度量标准，你的头儿就会从地洞里冒出来冲你发火。这样就让目标变成“完成你的度量标准”而不是“完成你的目标”。

如何能避免以软件度量指标为准而不是以目标为准这样的陷阱？有两种方法：

- 不要使用软件度量，愚者自乐。
- 将你的目标与软件度量目标等同起来。

想想你团队的目标。他们真正追求的是什么？当你们作为一个团队要有什么成果？你们要努力完成什么？对期望结果进行度量分析。那么怎么达到那些度量标准（合理的标准）就不成问题了，因为达到这些标准就是你们真正想要的。

> **作者注：**将这一节再仔细读读。可叹的是很多人并不理解这一点。

我现在就想知道

稍等片刻，一名惊恐的管理者有个问题：“如果我只对最终结果进行测评，那如何保证我们总是能得到这样的结果呢?”这个问题问得很好。成功的软件开发没有一次不是步步为营的——先以跬步，并时刻监测你们是否保持在正确的轨道上，再接着走下一步。但如果你只是对结果进行分析，那如何能保证你在正确的轨道上呢?

有两种方法可以始终警示你，使你的工作始终围绕着期望的结果进行：

- **使每一小步都小有成果。**这是敏捷方法的最好方式及基本概念。只要在每次阶段性工作中都能为客户提供其价值，就可以通过客户来经常性地监测你是否保持在正确的轨道上。你的软件度量方法注重度量客户想要的结果（如可用性、完整性及稳定性）。
- **使用一些与期望结果紧密相关的预测方法。**这种方法并不是很好，因为这种相关性从来都不完美。然而，一些“最终结果”直到最后也不能进行准确测定，所以用一些预测方法是需要的（如在第5章中说的“工程质量的大胆预测”）。如果你必须使用预测分析，那么始终把它们当成是对真实结果分析的基础从而确保你正在向你所想的目标前进。

按需索取

准则三：不要只是收集数据而已，使用度量分析解决关键问题。

软件开发，毋庸置疑，会产生海量的数据——软件构建消息、测试结果、Bug、可用性研究、编译警告、运行时错误及断言、时间表信息（包括实施进程图）、源代码控制统计，等等。作为一个软件工程师，你或许已经将这些数据制成报表并产生数据图表。那很好，祝贺你!

确实该祝贺你吗？不，不，过犹不及。在一个人员嘈杂的大商场，你很难听清他人的交谈，事实上你根本没听清。你的大脑都把这些列为噪音并加以屏蔽。对于一堆杂乱的数据来说也是这样。

按需收集数据，不要一整堆铺天盖地地扔过来。相反，关注你真正想要的。你关心哪些关键

属性？有些人称之为关键质量信息（CTQ）或关键性能指标（KPI）。你需要一种浓缩的精华，即关键又通用的 CTQ。

一些 CTQ 对于你的产品是特定的。比如你正在从事一项有关无线网络的项目，其目标是建立一个快速又稳健的网络连接，那么你的 CTQ 就是网络连接时间及平均在线时间。

而另一些软件开发的 CTQ 则是共性的。如果你追求精益（我希望你是这样的），你就关心最小开发周期，你的 CTQ 可能就是完成一段脚本所需要的时间；如果你追求工程质量（我同样希望你是这样），你可能就关心代码的稳定性。你的 CTQ 应该是一种预测指标，像代码返修率或复杂度，及一个更精准的指标，像 Watson 警示。

> **作者注：** 当一个 Windows 应用程序崩溃时你将看到一个错误报告对话框，Watson 是其背后机理的内部命名。

我们有责任

准则四：不要使用度量分析做决定，而是通过度量分析使你明白需要做出个决定。

我想对一些员工最大的担忧就是他们将度量指标仅仅当成是一个个数字。我在一个专栏里阐述过这种误解，“不仅仅是数字——是生产力”（见第 9 章）。如果你的审查过程及成就仅仅归因于某种公式，那问题就严重了。

对于所有的决策来说同样如此。如果这个决策也基于一种公式做出，缺少应有的思考与顾虑，那么我们就成为了工序及工具的奴隶。那是本末倒置。工序及工具只为我们所用，而不是我们为它们所用。

幸运的是，如果你遵从前面的法则，只对你的期望结果进行评测，那么你的管理层就无法用那些指标来进行决策。是的，如果你总是没得出团队的期望结果，管理层可要你好看，这也是你应有之责。然而，因为所有管理者所获得的都是最终结果，你就不必明了为什么你没得出这些结果或者由谁或其他什么对此负责。他们（管理层）就必须进行调查、理解及分析。他们就必须在得出结论前进行必要分析。

完善的度量指标让你知道你遇到了问题。它们无法也不应告诉你为什么（有这些问题）。分析根本性原因需要仔细研究。如果有人说他可以从数据指标中轻易找出答案，那么就是在撒谎。

每个女孩都应有自己的风格

准则五：不要只对原生度量进行比较，要有一些基准及范例，它们提供了参照细节。

等等，一个惊恐的工程师有个问题：“好吧，那我们以分析一次结果是否合意为例，如完工的脚本量，相对于其他团队，我们的功能团队只完成了一半的脚本量，那我们的管理层就会要求我们花更多的时间努力赶工，即使我们的脚本相对要复杂得多。使用‘确切’的度量却使我们更加悲惨！”这是一个很不错的想法。但我有一些好消息及坏消息。

好消息是管理者确实在试图解决问题（正确的度量指标是有益的）。坏消息是管理者并没明白问题出在哪里。他们不会分析问题的根本原因（因为复杂庞大的脚本），他们认定问题出在偷懒的工程师身上。你要提供必要的参照帮你的管理者进行分析。

最方便且最好的参照是一些基准及范例。

- **基准会让你明白从度量指标中你需要什么。**你最初获得的结果的度量指标就是基准，自此以后，通过与这些基准进行比较你就知道你如何会做得更好或更差。如果你的基准本身就很庞杂，你的管理者对你的脚本就不会有所惊奇了。

使用基准对于你不断地提升工作质量是很有用的。

- **范例会让你明白你的工作成果要达到什么程度。**范例是应用软件度量得出的最佳结果。不必关心范例是怎么完成的或是哪个团队完成的。你的成果与它之间的差别就是你需要改进的地方。“但是如果他们盗用范例快速地完成脚本怎么办?”如果脚本符合质量标准，那么他们就没有盗用，而是他们找到了更好的方法。“但是如果我们的脚本比范例更庞大更复杂怎么办?”那么，你应该将它们分拆使之简化。记住，你是在度量一种期望结果，如果你真希望快速地为客户提供价值，你就必须使你的代码块既小又能快速地完成。要使你们工作水准获得最大的提高，使用范例是极其省事的。

标新立异

那么，你现在知道了软件度量是分析什么及怎么分析的，也知道正确的软件度量与错误的软件度量之间的区别，也知道忽视软件度量会使你感觉轻松但会显得很无知。忽视软件度量会使软件开发成为一个猜谜游戏从而使成功成为一种偶然。我认为，对这种轻视的委婉称法就是：愚蠢。

然而，你也注意到了，好的软件度量方法只对期望结果进行了分析，这样的做法并不通用。你不能只是简单地将之应用于每个项目上。确实，你还希望有符合工程质量及效率要求的结果指标（如生产率、可靠性、稳定性及责任度），但其他的指标如性能、可用性及所有有关客户价值的东西依赖于你所期望的脚本质量及客户需求。

这就意味着，将适当的度量方法用在适当的地方并不是多余的。这需要从团队的角度出发来决定某个版本中什么是你们真正关心的问题以及你们怎么知道你们已经达到了目标。因此，从一开始你们要使这些度量工作成为工作中每个步骤及客户回馈的一部分，从而保证你们正在朝正确的方向前行。言过其实吧，是吗？事情确实会向预期的目标前行吗？或许我是个神经过敏的傻蛋。

> **作者注：**以我个人的看法，我相信测试部门在定义并监督软件度量上具有很大的作用。他们按数据说话，并以客户的角度来使用软件。关键是，测试者的工作要放在度量期望结果上，而不是建一张张的图表，制造干扰。

2010 年 10 月 1 日：“有我呢。”

我们正面临着重量级版本发布前的最终决斗，我早已厌烦了人们满腹牢骚：依赖模块不稳定还延期交付。你是从哪个星球过来的？他们不稳定又延期那是当然的。

没错，没错，软件包所依赖的另外一些软件包应该比自己的更稳健（稳定性依赖原则）。我已经不厌其烦地强调这个原则。但是当你为一个雄心勃勃的科技公司工作时，如微软，没人愿意干等着一项其所依赖的技术稳定后再行编码——至少我没遇到过这样的执行官。

这就意味着你的依赖模块不稳定还可能会延期交付。这不是你所依赖的团队的错，而且下一次也不会有所改善。认栽吧——停止抱怨，该怎么办就怎么办。不知道怎么办？我知道会这样。

作者注：停笔3年后，我在过去5个月中发表了4篇有关过程改进的专栏，这是第一篇。为什么停了这么久又突然思如潮涌？因为在这篇专栏发表前7个月，我重新回到了Xbox.com网站的项目组。我曾对开发专员、项目负责人及项目经理进行过培训，而现在我又回到了这个位置。

在回来承担开发经理期间，我的专栏不是谈论迅速适应新角色就是讲一些我备忘录里的话题及一些新想法。当我从事新工作6个月后，效率低下使我不胜其烦。因此，我又重新去关注工程改进。

锦囊妙计

有5种方法可用来应对不稳定的依赖模块：

1. 将强依赖转化为弱依赖或知识依赖。
2. 通过交流及项目管理一举搞定。
3. 尽可能地接近他们。事必躬亲、身体力行、亦步亦趋、互通有无。
4. 吸取他们的成果为你所用。
5. 建立多版本开发计划，构建稳定的接口及实际可行的时间表，要作一个引领者而不是追随者。

等等，最后一种方法有点白日做梦——就只有4种方法可以应对不稳定的依赖关系。我们逐个讨论。

作者注：通过建立多版本计划、稳定的接口、更实际的时间表以及成为一个引领者而不是追随者来避免不可靠还误工的依赖方所造成的麻烦，微软里的（不管哪里的）团队称之为：明天更美好，从而按预期时间表提供完善可靠的用户体验。

这些明智的团队对新兴技术做了些牺牲，但是，记住苹果是一个很注重创新的公司，但是苹果的创新不是利用新兴的技术，相反，他们通过成熟的技术开创全新的用户体验。

我想你脑子没那么死

一个强依赖模块，就是如果没有它，你就铁定不能顺利发布产品了。如果它失败了，你也就失败了。一个弱依赖模块，具有一种回退机制。如果它失败了，你的软件仍然可以通过削减某些功能成功发布。

不稳定的强依赖模块往往使你死无葬身之地。你会希望将它们改造成一个弱依赖模块从而有个备选方案。通常，方案包括使用依赖模块的前期版本，也可以削减功能，或买断依赖模块的版权，或是进行整合。

备选方案往往更具心理作用。它们消除了对失败的恐惧及不确定性。谁都知道接下来将发生什么。产品中不会有新鲜的玩意——表现平平的预览版及乏善可陈的最终版。但此时人们仍然有积极性提供尽可能好的功能，没有了后顾之忧，人们就能更好地合作并解决问题了。

取一段合作方的代码作为参考，试着从强依赖或弱依赖转换成知识依赖。你确实不需要对其他团队有所依赖，除了他们的知识与经验。

知识依赖被轻视了——它们并没有得到应有的重视。因为你的团队或许不想使用一些强依赖模块或弱依赖模块，但并不意味着你不能从一些做过类似事情的人的才智与经验里获益。在第10章的“怀疑论及其他革新毒药”中我对此有讨论。

沟通失败

如果你正面临着交杂繁重的时间表，就如你以往参与的项目一样，你必须尽量与你的合作方沟通并作好项目管理。无论他们有多么可靠，也无论你的协调能力有多强。即使合作意向可以达成，关键的细节却没有定论。你需要经常重复地跟每个人沟通，并时刻盯紧每个预成品。

你可能认为这些额外的沟通会很烦琐，但只要你处理得当就不会烦琐——通过经常性的面对面交谈、项目跟踪（如 Product Studio 或 TFS）及通过 E-mail 交流计划变更。

- 经常面对面交谈（一个星期左右）对于协调小范围的变更、修正发生的问题及对关键事宜作周全的检查非常有用。一次周全的检查只用5分钟就能对合作意向作出确认。（我们仍然能在两星期内提供关键样品，不是吗？我们仍然没被解雇，不是吗？）
- 在工作项目数据库中进行项目跟踪，如使用 Product Studio、TFS 等商业软件包，对于跨团队跟踪 Bug 修复情况及工作项目的进展情况相当有效。与你的合作伙伴共享数据库查询，那么每个人都可以得到相同的状态信息。
- 当你或你合作伙伴的计划变更时，每个人都应当及时知晓。先与各种人（Scrum 负责人、项目经理及直接相关人员）通过 E-mail 联系。如果有些工作项目取消了，改变了或增加了，要马上更新工作项目数据库。在接下来面对面的会晤中，再对什么改变了以及为什么改变作全面阐述。这样做似乎毋庸置疑，但一人的细微改动对于别人就是伤筋动骨，这就是为什么你还得作个周全的检查。

> **作者注：**很显然，这种附加的交流及项目管理是一项额外工作。对于这篇专栏中提到的所有其他方面也是一样。额外的工作通常与项目管理者及测试者高度相关，但对开发者也影响重大。额外工作的分量与依赖类型（强的、弱的或知识性的）及复杂程度有关。要预备相应的计划方案。

你中有我，我中有你

近距离沟通并迅速解决问题的最简单办法是参与到团队中：

- **事必躬亲。**亲自出面了解你的合作伙伴。见个面及一起参与交际从而真正地了解对方。亲密的工作关系对于各个方面都很有帮助，你对双方的成功都至关重要。
- **身体力行。**与你的合作伙伴多些实际接触。整个团队可能没必要，但是专门安排一两个人抽些空跟你的合作伙伴相处，你会对双方存在的问题有一些惊奇的发现。
- **亦步亦趋。**随时与你的合作伙伴保持紧密关系。他们部署，你也开始部署，他们发布 Beta 版，你也发布 Beta 版，他们发布正式版，你也发布正式版。与他们保持同步可以让你省很多事——听我的没错。

作者注：灵活敏捷在这里确实很管用。采用简短的开发步骤并时刻准备着发布成果，这不仅有助于减轻你的工作负担，减少技术投入，同时也有助于使你与合作伙伴的产品版本保持同步。

- **互通有无。**多了解合作伙伴使用的工具、工作项目数据库及源代码。对他们的工作了解得越多，就越有利于你预见、理解及解决问题。

作者注：即使你的合作伙伴的接口还没定型，最先的接口雏形也有助于你尽早地进行开发及测试。你可以自己写个模仿器，可以使用合作伙伴早期版本的接口，在其最终版本出来之前以此进行开发及测试。

自成一套

熟悉合作伙伴所用工具的关键点在于把握好你们两者间的交接环节。在他们部署新版本之前，他们应该运行你们所写的构建验证测试——只有你知道希望从合作伙伴那里得到什么；当他们部署新版本之后，你们应该运行他们写的摄取工具——只有他们知道哪些模块在运作、模块间的复杂关系及需要做哪些特殊处理。

你们写的构建验证测试应该能快速检测新版接口是否在你们的预期下运行。编写这些测试可能会有些烦琐，因为你们必须了解自己的应用模式，你们还必须在他们的测试系统上编写测试版。当然，当每次交接顺利完成，你会觉得所有这些努力都是值得的。

你们应用新版接口时，他们写的摄取工具应符合你们所有的需求。包括安装、库、说明、配置及许可。编写这些摄取工具并不会浪费精力，因为他们对你们或你们的合作伙伴的帮助是相同的。也就是说，当每次交接后，如果他们不用再为了使你们的系统正常运行花上两天时间，那所有这些努力都是值得的。

作者注：我们的团队随时备有这些工具。非常棒。

别嚎了！

即使在最完善的环境中，在开发的过程中也时常会有意想不到的事情发生。保持灵活性，以精短的开发周期快速应对变化，并与你的合作伙伴、客户及在你们的团队内部进行妥善的沟通，这样在处理意外之事时就会游刃有余。

责怪你的合作伙伴要少犯错误是无济于事的。即使他们确实错了。我们有缘相聚，我们共进退同患难。如果你们处理问题不当或没有及时发现问题，那你们同样也是在犯错。你们可以在下一次工作中改善沟通并妥善处理问题——问题自然就少了。

因为不稳定的依赖关系可能会带来麻烦事，这或许还让人高兴，所以就要更具有包容的团队意识，为客户带去更快速、更广泛、更强大的革新产品。

不要成为可怜的失败者。设立备选方案，加强沟通，凝聚团队，并按序对工作进行高品质的交接，勇于面对，必创辉煌！

2010年11月1日："我在缠着你吗？Bug报告。"

有些开发者恨透了Bug。他们认为有Bug说明他们的工作出错了——在Bug出现之前，他们的代码看起来那么完美。这样的开发者可称为"业余"。专业的开发者懂得，他们唯一没有发现Bug的原因是他们没注意到。

我喜欢看到Bug。让我的客户发现它们还不如我先发现。我真正讨厌看到的是糟糕的Bug报告——语焉不详或泛泛而谈的标题，混乱或缺失的修改步骤，夸大其词，杞人忧天，以及条理不清、逻辑混乱、让人费解的解决方案。

为什么大家就不能写一份像样的Bug报告？不是说像样的报告就比蹩脚的报告冗长或者更难写，也不是说在Bug报告中为每个事物做个确切的定义是不可能的。呵呵，团队间的那些定义确实不同而且还互相矛盾。那在Bug报告中什么才是最确切的定义呢？我希望听到你的回答。

> **作者注：**软件的每一小部分都会有成千上万的Bug，这取决于它的规模与复杂程度。有些Bug并无大碍，像"我希望关闭按钮更大一点。"有些Bug则是误解了，像"我不能为我的游戏匿称起个下作的名字。"但有些Bug就是讨厌的臭虫，无论如何都必须加以改进，像泄漏用户信息。因为Bug往往是开发团队的局外人发现的，必须等到所有问题修正为止，Bug报告才告完结——通常使用工作项目跟踪数据库，像Product Studio（微软经典的开发工具）或者Team Foundation Server。

Bug剖析

所有的Bug报告有以下的基本要求：

- 标题。要简略。
- 指派。谁来处理这个问题。
- 重现步骤。问题再次出现的相关步骤。
- 优先级别。问题的紧迫性与重要性。
- 严重程度。问题所产生的后果。
- 解决方案。怎么解决问题。

其他很多方面对修复问题及明白其深层次原因也很有帮助，但以上基本准则简练得多。下面我们来对每条准则逐一展开讨论，消除这些疑惑。

标题及指派

标题应该简明扼要，一句话就详尽说明问题的唯一性，使Bug报告的检索及标识变得简单。"点击取消按钮，屏幕就清空了"是个差劲的标题。"关闭编辑框，清空屏幕"就是个很好的标题。后者简短得多，而且对问题的出处及发生时间提供了具体的信息。

当你要创建一份新的Bug报告时，你必须指定具体人选来解决其中问题。但是，即使你这个团队的每个人都很了解，你也不应该将一个Bug指定给其中某一位，除非你是开发团队的一员。相反，你应该将此任务交给这整个团队。通常的做法是在Bug报告中指定责任方或团队作为默认

选择。默认的选择通常是“主导”或“会诊”团队。不会再有更好的了。要相信这些团队，他们会知道问题由谁来解决。

> **作者注：**有些团队希望将所有 Bug 都指派给团队中的某些个人，这样可保证没有 Bug 被遗漏。但是，他们还是必须确认将 Bug 指派给“主导”或“会诊”团队以确保 Bug 未被遗漏。毕竟，团队外部人员并不知道软件还有其他什么功能。
>
> 作为惯例，所有 Bug 必须指派给能对其进行经常性检查的个人或团队。因为，大多数优先团队会每天开例会，我还是偏好将 Bug 指定给“主导”或“会诊”团队为默认选择。

重现步骤

没什么比一份 Bug 报告没有清晰的重现步骤更让人郁闷了。就像你的亲友对你说：“你知道该怎么办!”，没有给你更多解释。这让我很茫然，不知道怎么办。悲催了。

Bug 重现步骤应是言简意赅——一言中的。同时要包含软件创建编号（通常是单独列出的），你的工作环境（操作系统版本、所用浏览器及其他相关的细节）以及一些先备条件（像先注册个 Xbox. com 金牌账号等）。

有时你不能确定 Bug 是怎么发生的，因为它有时是间歇性的或跟某种特定的状态相关。这种情况下，列出创建编号、运行环境及配置等信息，接着描述下当时的情况，以说明具体的 Bug 重现步骤无法确定。

> **作者注：**我们有些内部工具，如 Watson 与 Autobug，它们可以自动生成 Bug 报告。诚然，用这些工具生成 Bug 重现步骤有其局限性，但是它们通常仍可以提供些堆栈跟踪信息、创建编号、环境及其他相关的信息，且它们对隔离问题有帮助。

在简洁的 Bug 重现描述后，你必须指出什么是你希望发生的（“期望”），及事实发生了什么（“事实”）。所有的重现步骤包括这三方面——配置、期望结果及实际结果。这样当别人在看这份 Bug 报告时就知道到底哪里出错了及怎么重现它。

通常一张图、一段视频顶上千句文字，有很多工具可以对屏幕进行图片及视频抓取。将这些文件附到 Bug 报告中，这些文件就是一份能妥善修复 Bug 的报告与含糊不清的报告之间的区别。

> **作者注：**如果一个问题可以用 4 个步骤讲清楚而你在 Bug 报告里却用了 15 个步骤，这是让人相当恼火的。不仅仅是因为 4 步很简单，容易理解，而且这样可以使开发者及测试者快速找到 Bug。重现 Bug 用的时间越少，在确认 Bug 的原因上所花时间也越少（可能出现 Bug 的步骤少了），同样在确认 Bug 已被修复上所用的时间也越少。

优先级别

对于优先级别意义的讨论一直没完没了，这种级别的范围值通常为 0 ~ 3。说实在的，你可以把时间更好地用到其他地方去。这里还是说些简单的准则，以此为基础阐明优先级别。

- 优先级别一旦设定则不宜再改，除非 Bug 本身角色变换了。如果级别 1 意味着："在目前的冲刺阶段或里程碑期间修复"，级别 2 意味着："到下一个冲刺阶段或里程碑期间再修复，"那么在每个冲刺结束时，你必须更新 Bug 的优先级别，这样不仅很浪费时间，而且改变了 Bug 的"最后一次变更时间"，这会丧失很多重要信息。
- 优先级别必须容易指定并区分。你不会想让你的团队花大量的时间争论每一个 Bug 的优先级别吧。它必须是显而易见的，不管是在写 Bug 报告或读 Bug 报告的时候。
- 优先级别必须易记且易操作。人们不需要问："下一个 Pri 2 是什么?"，人们也不需要问哪种级别需要做什么。

基于以上三条准则，一般普遍接受以下优先级别的定义。

优先级别	描　述	修复时间点
Pri 0	一个需引起严重关注的致命错误。不存在变通办法，是一个不可逾越的 Bug	只有解决了这个问题或找到了变通办法，你才能安心
Pri 1	一个需引起严重关注的致命错误	必须在当前的冲刺阶段或里程碑期间解决
Pri 2	一个严重的错误	必须在产品发布前解决
Pri 3	一般性错误或建议	最好在产品发布前解决

Pri 0 通常有碍测试、部署或其他对时间敏感的工作。你必须给开发者或团队发邮件并电话告知他们，或者直接过去跟他们谈，直到有人解决这个问题。如果有变通办法，Pri 0 就必须改成 Pri 1。

> **作者注：**确实有开发团队对优先级别有非常多的定义。有的从 Pri 1 开始，而不是 Pri 0；有的不遵从我在本章开始时列出的准则，或者在一个单独的区域提示 Bug 信息。
>
> 如果你查看另一个团队的工作项目数据库，确定你使用的是他们的定义。这些定义通常显示在工具提示上或帮助窗口中。

严重程度

严重程度比优先级别简单得多，但是它还是经常被搞混。严重程度指的是问题所产生的影响范围，不关乎"有多么严重"这样的问题。其定义是：

- 严重程度 1。某问题引起系统崩溃或客户数据丢失。
- 严重程度 2。某问题引起的故障阻断了后续操作。
- 严重程度 3。某问题引起操作不便或界面显示不完整。

注意，严重程度与优先级别是相互独立的——换句话说，严重程度与优先级别毫无关系。优先级别 1 的 Bug 比级别 2 的 Bug 更重要，不管其严重程度如何。显示一些不合适的内容就是严重程度 3 但也可能是优先级别 1；系统崩溃后用户强行重启就是严重程度 1 同时也可能是优先级别 3。工程师声称一个未致系统崩溃的 Bug 的严重程度是 1，因为严重程度很高。你完全没必要成为他戏弄的笑料。如果你这样就白痴了。

解决方案

Bug 报告中最重要且经常被混淆的部分是"解决方案"——说明如何解决问题。解决了一个 Bug 意味着你不再关心这个问题。当 Bug 的发现者确认这个方案能修复这个 Bug 时，你也不打算

再作更多的处理。

在你发布产品前，如果对一个问题需要做更多的处理，即使这不是你的团队的责任，那这个Bug还是要引起关注，并指定你团队里的一个人继续跟踪相关事宜。

以下是解决方案部分可能包含的内容，按字母排序：

- **意图**。Bug报告描述了所需处理的细节，按预先意图进行。
- **重复**。这个Bug与报告中先前指出的Bug有相同的起因及非常相似的用户体验。不要像分析一个旧Bug一样分析新Bug——不管这个新的Bug报告看起来会多精美，除非你想与Bug发现人为敌并丧失“首先发现Bug”的机会。
- **外部性**。一个Bug是由你控制能力之外的原因引起的，则你可以在Bug未修复之前发布产品。如果你团队之外的人没有修复这个问题，使你的产品发布不了，那么保持对这个Bug的关注并指定你团队里的某人进行跟踪，找到其他团队中存在的问题。
- **已修复**。Bug修复了。这是我最喜爱的解决方案。
- **不再发生**。你不能让Bug在之前说过的创建版本及环境中再次发生。声称“在我的机子上运行没什么问题”并不代表Bug解除了——随时与Bug发现人保持沟通。
- **延期**。你不想在这个版本中修复Bug。延期是偷懒者的借口，他们总说明天我会写个测试单元。真正的工程师会时刻关注这个Bug并会在Bug报告里留出一个“等待修复”专区来指出下一个改进版本，只要他们真的想修复这个问题。
- **不修复**。你不再修复Bug。这是我第二种得意的解决方法——这说明你有丰富的经验判知哪些Bug不需要修复。通常是因为修复本身会带来比Bug更多的问题。

当你在解决一个Bug时，你必须在解决方案中有段描述。这段描述是很重要的。这样可使解决方案少些争论，Bug重现时就更易理解，使你与你的公司免于因为这个问题成了公众热议的话题。这在我之前的一个团队中曾发生过——我们使这个公司免于千夫所指，因为我们的解决方案中对一个出现不合适内容的Bug作了描述，以说明我们并非蓄意而为。

当一个Bug被解决，它将被自行指派给发现它的人。如果这个人不是开发团队的人员，那这个Bug必须指定给另一个团队中的人，这个人可以跟Bug发现者核实解决方案。但你不能总是指望团队外部的人能及时周到地确认解决方案。当然，如果这个解决方案不怎么令人满意，那么这个Bug应被重新激活。

> **作者注**：我第一次为我的团队制定解决方案是在10年前。回顾之前的邮件，以上定义至今仍然有效。

过犹不及

Bug报告中还有很多其他区域。我说过用“创建”及“环境”两个区域记录Bug相关信息以及用“等待修复”区域来说明什么时候处理Bug。还有一些区域用来跟踪记录底层原因，这个Bug是怎么被发现的，Bug是在产品或服务的哪个方面发生的，潜在的安全威胁以及其他信息。

设定好Bug报告的必要条件，少则缺，多则无益。要求太多人们会怨声四起而拒绝完成Bug报告——两种极端都会对你及你的客户不利。

Bug报告要易写且易读，这样会促使他们在发现问题的时候制定清晰的Bug报告。使用一些

Bug 模板对于一些内容的编写是很有帮助的。对于我们在乎的工程师及客户来说，规范的 Bug 报告使一个问题在用户发现前消灭于萌芽状态，没有比这更好的礼物了。

2010 年 12 月 1 日："生产第一"

我是这么深爱着微软，我们拥有这么多的优越条件，这个公司拥有着一群充满智慧的人，更拥有富有生命力的产品及独到的视角，但时常还是有人那么无知——微软是一个海纳百川的公司，但这些无知足以让你发狂。

再看我们的服务环境，这里有个我们自己的例子。目前我的团队已将服务环境分成软件开发、签入测试、情景测试、压力测试、跨部门整合、合作者整合、认证及生产等 8 个不同的环境——我们还计划在明年增建一个产前环境。但是一句给你泼冷水的黑色幽默是：即使拥有所有这些环境，仍然会有大量的毛病及意外只在生产环境中才被发现。

为什么我们会这样挥霍无度？因为我们够资本（这很好但不是很好的借口），还因为还有很多老古董的企业工程师不明白服务的真谛：生产第一。这些工程师妄想万事俱备再行测试并奢望有一个难得的商业软件集成环境，他们仍这样执迷不悟。闭上眼，一起按下 Ctrl + Alt + Delete 键三次，好好反思："生产第一，生产第一，生产第一。"

> **作者注：**虽然这是我最近才写的一篇专栏，但其已是我在微软所写专栏中最有影响力并被引用最多的一篇。我已收到多个管理级别部门的电子邮件，在其中详尽讨论了这个想法或直言说："生产第一。"这让人倍感温馨。

这事怎么就成了

哪些傻瓜创建并维护着这些没用的服务环境？这些人毁了企业级软件开发。

大型商业企业依赖于企业级软件——它们必须运行正常，否则他们不会购买。一旦他们购买了软件，他们就是所有者。企业级软件不是你想改就改的。是的，即使是打个安全补丁。

记住，企业的支票是视软件运行是否顺畅而定的。软件的所有改动都直接给企业商务带来风险。如果软件出问题或运行欠佳或不能保持持久稳定，企业是不会买的。他们会跟你说："打个补丁？这鬼主意不错。"

微软所有的工程师都明白了这艰深的道理：在代码未经全部测试之前，你是不能发布软件的。企业级软件是不容许"重试"的。

未经在生产环境中检测就发布软件，这是企业级工程师想都不敢想的。如果他们理解了服务的真谛，他们是该幸灾乐祸了。

拜托，你不能认真一点吗

软件服务的真谛是什么？生产第一位。让我们逐个击破这些关于测试与集成环境的神话：

- **如果签入测试系统在某种环境中通过，那么它们在所有环境中都会通过。**好的，这明显是错误的，但这里还有更糟糕的问题。写一些除了在生产环境而在其他地方都会失败的严谨的登录测试系统并不难（像广播测试及数据库镜像测试）。不要自欺欺人了，还是在开发

人员登录之前，写一些完整的自动登录系统，使它们在其开发环境中快速运行。

- **在将代码与合作方整合之前，你需要一个独立的环境测试各种情况。**人们相信这一点有两个原因：他们不希望不稳定的代码破坏合作方的代码，以及他们不希望合作方不稳定的代码来妨碍测试。第一个理由很有道理——你需要一个测试环境来做一些验收及压力测试，特别是对一些关键组件。第二个理由就很可笑了——就好像，无论在什么状况下，你的合作伙伴都将确保你的测试环境无恙。他们不会，也不能（以下将详细说明）。
- **你不想在生产环境中进行压力测试。**为什么不？你担心生产过程会出意外？你不想知道为什么吗？这不是真正的目的吗？明白什么东西在可控情况下出问题并在必要时回退，这不很好吗？你还好吧？
- **你需要在发布之前通过集成环境对跨部门情况进行测试，并提供产前环境对外部合作方进行访问。**假设在进入生产环境前，跨部门运作良好；再假设，发布前外部合作方在另一个环境中已完工。你现在敢保证质量吗？不，都不敢。在生产阶段诸事难料，那时有更多的机器，不一样的负载条件，不同的路由及负载平衡，不同的配置，不同的设置，不同的数据及认证，不同的操作系统及补丁，不同的网路及不同的硬件。你会发现一些集成问题，但并不能说明这样庞大的开销是值得的。

> **作者注：**一个虚拟的云环境，就像 Azure，会有这样的问题吗？不，它仅仅解决了不同的操作系统设置与补丁以及不同硬件问题。它对于其他问题来说也小有益处，但生产就是生产，不可替代。

- 你需要一个受保护的认证环境。为什么你事先要验证一下产品？因为你想确定他们在生产环境中运行正常。哦，等会儿。

我们来回顾一下，生产第一。你需要一个开发环境运行一些自动签入测试系统，一个测试环境来做验收及压力测试以防灾难性错误，以及一个生产环境。再多无益。

> **作者注：**最好是你的合作伙伴为你提供一个 one-boxes 为你的开发及测试环境所用，one-boxes 是一些事先配置好的虚拟机，以其来运行这些你所需要的服务，这些虚拟机是一些压缩镜像。当然，one-boxes 跟生产环境完全不同。

很无助

“等会儿！我们不能把未经测试的代码扔给客户。他们会勃然大怒的！同样也不要让我们公布未正式公开及未认证的合作方代码。你疯了吗？”闭嘴，成熟些吧。生产第一。现在的问题是配置生产环境对未公开代码进行测试与认证。

这种解决方案称为“持续部署”（continuous deployment）。这个概念很简单：将多个创建（build）部署到生产环境中去，使用自定义的路由定向。就像面向规范服务（regulating service）（而不是源码）的源码控制系统。这种系统没有构建进 Azure 及其他云系统是不可想象的。

实现持续部署有很多种方法，这些方法与部署系统及自定义路由的灵活性有本质的不同。但是，持续部署的实现要相对简单。

最困难的部分是数据处理，必须使其在各种创建间保持有效。然而，如果一项服务被设计成在一项新的创建部署后能执行至少一次回滚，即使这项新创建使用了新数据，那么这项服务在一个持续部署的环境中就能运作良好。

> **作者注：**你还需要注意多种创建间设置的多样性问题。这有点棘手，但还不算很糟。理想情况下，你的设置不会时常改变。如果新创建依赖于.NET Framework或操作系统的最新版本，那么这些新创建必须放到新配置的机器上——但你没必要对持续部署做什么更改。对于数据及模式的变动，我强烈建议不要配置多种数据库。相反，在数据行、列及表增加时应用模式的变更而不要在数据行、列及表变更或删除时应用模式。如我之前所说，你必须这样做，即使你不采用持续部署。
>
> 当你必须对模式做重大变更时，成熟的做法是做双版本准备：
>
> - 第一种版本，创建一个能认识老的及新的模式的版本，并进行对应处理。（这并不是个新鲜的重大功能，它只是具有一种处理两种模式的能力。）
> - 第二种版本，当第一种版本稳定后，推出一个依赖于新模式的版本。现在，如果还有问题，你就有了一种安全的回滚机制。

怎样才能办到

如何在集成测试、合作方测试、压力测试及认证环境中使用持续部署？我们来逐个讲讲。

- **集成测试** 你在生产环境中部署了新的创建，但只是设置你们的工程师团队与创建间的自定义路由来指定路径（默认本来是没有路由的），其他所有人都在等候着你最终的公开版本。这种技术叫做“曝光控制”（exposure control）。那现在你的团队可以在一个有着实际生产数据及实际生产负载（采用未向客户公开的创建）的真实生产环境中测试了。

> **作者注：**你需要一个功能强大的诊断工具来分析你在生产环境中发现的错误，不管是不是持续部署都该如此。

- **合作方测试** 合作方将他们的新创建部署到生产环境中，并只在他们的工程师团队与他们的创建间设置自定义路由来指定路径。其他人看不到变更。现在合作方可以在没人（包括其竞争者）知道其工作进展的情况下对你的生产服务进行测试，
- **压力测试** 你在生产环境中部署了最新创建并完成测试。一旦通过验证，你通过“曝光控制”逐步提升最新创建的实时路由负载——先是1%，再是3%，然后10%，30%，100%，你在这个过程中监测服务状态。如果你的服务出现错误信号，就记录下数据及路由负载回到前一个版本中（实时回滚）。
- **认证** 合作方在生产环境中部署他们的最新创建并对它们进行测试。一旦这些创建被验证，合作方使用“曝光控制”将认证团队定向到他们的最新创建上。在客户或竞争者看到他们的最新进展前，这个认证团队在生产环境中对他们的创建进行认证。当这些创建通过认证后，合作方可以选择何时将实时路由定向到他们的创建上。

- **Beta 红利！** 你部署一个 Beta 版创建时，一旦它通过验证，你利用“曝光控制”将 Beta 版用户定向到 Beta 版创建上。
- **改良红利！** 你另外部署了一个当前创建的改良版。当它通过验证时，你使用“曝光控制”将一半的实时路由负载定向到你当前的创建上，将另一半定向到新的改良版上。你利用在使用模式（usage pattern）中见到过的这种差别提升服务的效率。
- **自动回滚红利！** 在你将所有的实时路由负载定向到新创建上后，你将之前的版本留在原处。将你的健康检测系统与曝光控制系统连接。如果你的健康检测系统提示错误，则曝光控制系统将马上自动重定向路由至你之前的版本——无论白天还是晚上。

我们现在不在堪萨斯州了

十多年前，在我们进入企业软件领域后，微软的工程师从中受益颇多。遗憾的是，这些经历对我们最近的面向服务工程却产生误导，迫使我们以服务质量的名义创建了多余的环境。

维护不相干的环境耗尽了我们的带宽、能源及硬件资源，并给工程师带来沉重的负担，而没有带来真正的质量保证。该停止了，庆幸的是当团队迁移至持续部署后这种趋势行将中止。

有了持续部署，你满足了服务质量而无需外加成本。同时，这为你的服务质量的提高，以及你与你内部或外部合作伙伴的合作带来太多好处。

该停用源代码控制系统来进行软件开发了。这个想法可不是开玩笑，也不是非分之想。持续部署为服务带来了相同的效能。日后的某一天，当我们回想今日，会惊奇：人们怎么可以没有它？

> **作者注：** 目前，在微软，我所知道的只有 Bling 与 Ads 平台拥有基于持续部署的产品。亚马逊拥有业界最闻名的系统。
>
> 我的团队目前正在创建一个非常简单的持续部署环境。这个环境使用一种 on-machine 的 IIS 代理服务器提供曝光控制，这样可以在同一台机器上对同一种角色使用多个软件版本。
>
> 从工程团队的角度看，我们仍然一如既往地为同一台机器部署同一种角色。不同的是现在这些机器安装的是这些角色的不同软件版本，并通过曝光控制由我们选择版本自主定向路由负载。

2011 年 2 月 1 日：“周期长度——生产力的老生常谈”

没有什么比浪费时间与精力更使我恼怒了。我这里所谈的不是培训、重组、激励斗志或度假。这些至少对于你的人生来说都是有潜在价值的。我讲的是创建时间、集成时间、多余的规格、不完整的功能、碍手碍脚的错误、满目疮痍的 Bug 及过分讲究工程质量的代码与过程。你知道——时光一去不复返。

几年前，我在专栏“精益：比五香薰牛肉还好”（本章较前部分）中已对所有这些浪费进行了批驳，但是我的例子及一些解决方案只对个人方面进行了评述，我并没有真正为团队指出一条明路。你需要一种途径找出极具危害的浪费问题，促使你的团队解决这个问题，使你的团队安枕无忧。

在制造行业，成功的秘密武器是降低库存。库存掩盖了你的生产问题，当你降低了库存，问

题就显现了。修正问题并继续降低库存，渐渐地，你的浪费被根除，效率骤升。在软件开发领域，成功的秘密武器是缩减开发周期。概念与实现之间的用时越少，你所面临的障碍就会越多，而这些障碍常与工程无关。修正了这些问题，就释放出了生产力。让我来为你支一招，解放你工程的灵魂。

> **作者注：**很多读者没抓住这篇专栏的要点，以为这只对网站与网络服务有用。而网站只是我举的一个例子，缩短开发周期同样对打包产品有效，如 Microsoft Office。唯一不同的是，网站与服务是每月或更频繁地发布最新版，周期长度是两次公开发布之间的时间。对于打包产品或在线服务来说，公开发布是逐年或更长，其开发周期是完成一项功能的时间。
>
> 很多工程师问为什么缩减开发周期、协同定位、早期 Bug 修复以及其他精益概念之间会有不同。我对这样的问题感到奇怪，因为同样这些工程师就不会问为什么优化内部循环，避开磁盘 I/O 访问，捕获他们的代码错误这些会有助于提升软件执行效率。如果你真看不出来这二者之间的联系，你就该出局了。

一应到位

缩短开发周期的第一步要做的是，确定你的开发周期耗时要多长。对于服务来说，是两次公开发布之间的时间。然而，对于打包产品来说，缩短公开发布版之间的时间往往是不被市场所接受的。因此，周期的更好定义是开始拟定具体功能规范到完成功能开发之间的时间。完成功能开发到底是什么意思？这是关键。

你必须对软件功能及服务的发布定义一个“完工”（done）的概念。以下是我的团队所用的定义。我们认为对于每一个功能要有一个四要件，对于每一次发布有另一个四要件。我们每四个星期对 Xbox. com 网站进行一次发布——我们乐此不疲！

每项功能“完工”：

1. 对所有更新过的设计及代码进行了检查。
2. 编写所有的自动化测试并通过。
3. 不存在有碍产品推出的 Bug。
4. 所有监测及健康检测工具到位（打包产品使用回馈工具）。

每次发布“完工”：

1. 所有本地化及全球化工作准备就绪。
2. 全面测试完成。
3. 解决所有质量问题及合作方工作到位。
4. 完成所有必需的发布文档。

如果你想缩短从头到尾工作的时间，通过这 8 项“完工”标准可以找出所有问题。让我来简要地讨论一下常见问题及如何处理。

如果你要创建

“完工”首先要做的是，检查更新过的设计及代码，这只是节省些时间。所以让我们来谈谈自动测试——单元测试、组件测试、压力测试、验收测试、系统测试及错误注入测试，等等。开

发人员及测试人员必须共同参与这些测试系统的编写。由谁来写哪些测试依团队而不同。因为他们希望缩减开发周期，大部分团队对他们的测试套件（test harness）及运行测试的时间纠缠不休。

为缩减开发周期，你必须分清快速又频繁的测试与缓慢又寥落的测试之间的区别。很多测试在此进退维谷并重写或重构。

有一种测试套件就够了，你不必二者兼具——一个用来进行快速并有可靠性保证的签入测试及一个用于全面性测试。如果你在一个非常庞大的团队，即使一次快速签入测试都要耗时 10～20 分钟以上，那么或许应该让这支庞大的团队采用优先及平行测试技术。

同样，如果重新创建一个庞大的代码库需要 10～20 分钟以上，你最好采用高效的平行创建技术及依赖创建（build dependency）法。记住，创建、测试及签入组成了软件开发的内循环。所有有利于加快软件开发内部循环的做法都将使整体生产率成倍提升。

对于代码分支来说，你可不想主干有一条以上的分支。集成成本是很高的，每一个分支都增加了一次集成，要慎重考虑。假设你在生产个人便携式电脑，要使这些功能独到的组件逐一通过海关，这将使你的销售大计毁于一旦。每一个分支层次就如同在软件修复、功能模块及主干间增设了一个海关。

请神容易，送神难

在公开发布前清理大量的 Bug 确实会拖延开发周期。Bug 的修复耗时越长，你就等待越久。在设计与代码审查中加入自动测试将大有帮助（就如重构面条代码及采用测试驱动开发）。不管怎样，重要的是尽早找出并修复 Bug。即时修复 Bug 对短周期至关重要。

无论你怎么做，你还是会有 Bug——我们是人不是神。有些 Bug 非常难找和修复，这些将使你们的工作慢下来。好消息是有一种架构对错误具有一定弹性，它能缓解修复最棘手（经常反复发作）Bug 的迫切性。弹性架构可以让你在这些顽固、偶发的问题上收集数据并进行修复，只要对这些问题有足够的了解。

> **作者注：**在相当多与此矛盾的专栏中，有一篇我对弹性机制做了更多描述，即“碰撞测试：恢复”（见第 5 章）。

我该怎么做

监测与健康检测常常是事后诸葛亮。这些节外动作徒增了跟踪检查客户方问题的时间，拉长开发周期。这跟打包产品中设计与使用客户回馈工具一样。

监测与健康检测需做事先考虑，一开始就得进行设计。想想你的软件为什么要这些功能，再问问你怎么知道它会按预期执行。这样会让你清楚地明白如何对它的使用进行监测并了解它的健康状态。

所有这些数据及快速的开发周期使快速回馈机制及实时改进得以实现。搞清楚在每个周期内所花的时间都用在了你怎样才可以让你的产品做得更好，以及如何能与你的团队相得益彰。

> **作者注：**事前对监测与健康检测做好准备刚刚被加入到了我团队的“完工”条目里。糟糕的监测过程及缺失的健康检测使我们误入歧途，而只能看着我们的合作伙伴贻笑大方。

看我的

我们已经谈过全面测试的自动化，而本地化过程在微软是极其精到而快速的，所以下一个会产生问题的领域是完工（sign-off）阶段。与质量相关的领域（如安全、隐私等）将有所涉及，而所有工程师一起修复 Bug 成为常态。然而，这些领域的完工如合作方完工一样，确实会延长开发周期。

虽然质量领域的问题是每个工程师的责任，但如果在一个质量领域的工程过程中指定一个工程师作为带头人，完工阶段就会达到最佳效果。这些工程师成为这些领域中的团队专家，这是个他们施展跨团队工作能力的极佳职业机遇。

因为团队专家自始至终在解决质量领域的问题，完工的要件及相关活动对于他们比起其他团队的员工来讲就要更快、更容易。另外，团队专家加强了团队与其领域中有关专家之间的关系，这样同样加快了工作进展并使整个团队快速成长。

> **作者注：**我们在 Xbox.com 项目中为缩减开发周期所做的另一件事情是使功能团队协同合作（适当的时候包括相关的人员与销售商），这样使我们的生产率提升了 20%（按已完成功能模块的规模及数量占整个公开发布版的量计）。

能不能再详细些

利用我所述的几种方法，我的团队已尽量将产品发布周期从一年几次增加到每 4 周一次。真是太棒了！在讲这些优点之前，让我来谈谈人们经常问的两个问题。**如果在功能或架构改变上所用开发时间超过 4 周时该怎么办？**有两个基本的方法：横向及纵向。

- 横向方法是一次只在一个协议栈的层面对这次大范围的变更进行改动，比如，先改变模式，再推出新的服务，再写新的模型，再是新的控制器，最后是新的视图。每个层面都可以在 4 个星期内完成。
- 纵向方法是将一次大改动分成很小的一些功能块。每个纵向功能块就可以在 4 个星期内完成。如果这些功能块使用户体验变得不连贯，你就可以暂不将这些更新的功能块提供给客户，直到有足够的功能块得以完成。
- 人们通常交叉使用横向及纵向方法。遗憾的是，横向方法经常引起每个协议层工程量过大并妨碍了互动的客户回馈机制。我更偏向于纵向方法，横向方法只作为最后的一种选择。

你怎么看待持续性工程（Sustained engineering）？从测试到正式发布过程中，每次持续性工程更新的最终验证大概需要一个月。而我们每 4 个星期发布成果一次，持续性工程只是我们日常工作的一部分。没有哪个正式版是按持续性工程发布的，除非在一些特殊的情况下，更重要的是，没有一个团队谈得上持续性工程的。其中五味杂陈——我们在犯错中痛着，并在前进中快乐着。

生活很美好

在过去的 6 个月中，我们都是每 4 周发布一次，我们实实在在地体会到了这样做的好处：

- 工程师们所抱怨的很多开销已经一去不返。我们本应该将这些开销投入到有益的工作中去。

- 遗漏是可控的。如果一项功能在一次发布中错过了一两个星期，那么它仍然会在一个月内再出现在发布中。
- 正式发布版不再是恐怖又疯狂的了。我们一直在发布，在4个星期内你所能发现的问题也就那么多。
- 事半功倍。通过流水线生产频繁地实现发布需求，我们的开发过程变得更快了。
- 我们使客户更满意，他们也注意到了。显著地提高问题响应及回馈的次数，备受客户赞赏。

虽然变更带来了些痛苦，缩短的开发周期使开销更少，出成果更快，使欢快无时不在。团队很开心，我也很开心。

今后，我们期望能在一两天内就出成果，就像我们的对手一样（他们可能正在嘲笑，4个星期这么长的时间）。我们并不想总是这么快地发布产品，但如果能这么做就意味着效率大幅提升。如果照此方法走下去，你将拉近与客户的距离，这对谁都再好不过。

作者注：为什么我们在Xbox com团队中采用4星期开发周期？因为很多领导者，包括我，知道这是提升我们团队生产率及客户质量的最佳路径。缩减周期及工作量是一项精准制造的古老技术，这可以追溯到20世纪30年代。

第3章

根除低下的效率

本章内容：

正如我在第2章的“精益：比五香熏牛肉还好”栏目中所说的那样，浪费和罪恶在工作中常常相依相伴。关于这一点，没什么比组织的沟通（本章的几个栏目都会涉及这个话题）以及项目之间的自由时间的合理使用来得更为明显。这些领域影响的不仅仅是个人，而是整个团队。因此，它们的影响也是成倍于其他领域的影响。

在我的恐怖字典中，规范文档（规范书）和会议始终占据着特殊的位置。我想可能是因为工程师花了太多的时间在会议上，而且常常还是在讨论规范书的原因吧。尽管我很希望这两样东西在我们熟知的世界中消失，但它们之所以存在必定还是有它们的用途的。我们能做的，是要关注那个真实的用途，而把其他多余的东西统统抛弃。

在这一章中，介绍了一些消除无谓的低效率的策略。第一个栏目解决规范文档临阵变更问题；第二个栏目解决了项目之间的空闲时间的合理使用问题；第三个栏目着重讲的是如何减少对会议的厌倦感；第四个栏目尝试彻底取消规范文档；第五个栏目试图尽量缩简规范文档；第六个栏目分析分布式开发；第七个栏目论证如何能正确地优化团队协作；第八个栏目论证如何利用检查列表及单片流方法进行过程改进；最后一个栏目主张当机立断，即使当时情况不明。

其他栏目对团队间沟通进行了大量的讨论——从跨团队协商到处理非技术性问题，无所不谈。另外还有一些讨论个人可以采取的提升自身能力的措施。但这些篇章对于团队该怎么做才能充分利用有限的时间这个问题的核心进行了充分的剖析。

——Eric

2001 年 7 月 1 日：“迟到的规范书：生活现实或先天不足”

你已经达到了“编码完成”（Code Complete）的阶段，你正在全力修复 Bug，这时候看看你的邮箱里收到了什么？啊，太有趣了，居然是一份新的规范书！把它一脚踹开，如何？请稍等，这可是以前的规范书不小心遗漏掉的一个关键功能，或者像我们常说的那样，“代码本身就是规范书。”

> **作者注：**编码完成（Code Complete），是指开发者认为对于某个功能所有必要的实现代码都已经签入到源代码控制系统的一种状态。通常这只是一个主观判断，而更好的做法实际上应该基于质量标准来度量（那时候经常称为 Feature Complete，即功能完成）。

可以想象，测试人员被激怒了。因为他们没有及时拿到规范书，并且他们觉得“他们被排除在了项目开发周期之外”。实在太晚了！代码的表现跟规范书不符，他们也没有对它进行测试。开发人员也感到焦躁不安，因为他们原以为功能已经完成了，但实际上测试人员却在疯狂抱怨他们实现的是一个“错误”的东西，这将导致大量的返工。更糟糕的是，开发人员由于编写了不合规范的功能而被揪住尾巴。但还是有更让人开心的，人们开始讨论这个新的规范文档，找出漏洞，进行修改，而最重要的是必要时将不稳定的代码丢进坟墓。

对于每次变更，搅动，搅动，搅动

也许在极端情况下，变更还不止一次。但这的确有可能发生。即使变更没有那么晚，规范书常常也是不完整的，或者在尚未被开发人员及时复审和检查之前就匆忙交付开发了。

结果怎么样呢？搅动代码，一次又一次地更改以前的实现。开发人员开始编码的时间太早了！规范书本身就有问题，因此代码自然也有问题。当有人指出这些问题的时候，特别会议召开了，但有人被遗漏了，因而缺席了这个会议，代码返工重写之后，那个被遗漏的人发现了其他地方的错误，于是需要召开更多的特别会议，就这样周而复始、永无宁日。

有什么办法可以解决这个问题呢？有些人可能会说：“项目经理是人渣，应该缠着他，直到他把工作做好。”这听起来有点残酷，哪怕是我也有这样的感觉。规范书来得太晚，这是生活的现实，问题在于你处理它的方式。我见到过有如下一些不同的方法。

> **作者注：**我能想象，一些极限编程爱好者在那边嚷嚷了：“给他们一个房间！”（一个团队房间）。我在后面的一个栏目——“停止写规范书，跟功能小组呆在一起”，也会谈到这个观点。然而，微软是一个相当多样化的环境。不是每个团队都能呆在同一个地方的，文档通常是解决团队之间的相互依赖问题的必要手段。因此，也并不是一个解决方案能够解决所有的问题。

走廊会议

第一种方法是走廊会议。当一个开发人员发现手头的规范书存在漏洞，这时候项目经理正好路过，于是一个走廊会议就开始了，一些问题通过这种方式得到了解决。那个开发人员很开心地

回到他自己的座位，想着他终于搞清楚了接下去该做什么。那个项目经理也回到了他的办公室，想着开发人员写出来的代码肯定能够反映他真正想要的东西。也许他们在想同一件事情，也许不是。也许测试和实施人员会同意他们的解决方案，也许不会。也许他们方方面面都考虑到了，也许他们不曾做到。也许这是最好的方式去处理变更，也许猴子会冲出我的……好吧，至少你知道了有这种方法。

委员会议

第二种方法是委员会议。对于这种会议，不同的团队有不同的称呼，但它主要是用于讨论规范书变更的主管级会议。通常这种会议会定期召开，各个主管形成一个组织，他们聚在一起讨论规范书上的漏洞或者问题，并且以组织的名义寻找解决方案。主管的项目经理记录会议结果，并且发邮件告知整个团队。

这种方法的优点是，委员会议把该包含的人都包含了进来，达成了最终决议，记录在档，并且拿这些最终决议跟团队沟通。缺点是，委员会议也是可怕的噩梦。它们通常比较冗长、令人厌烦、使人筋疲力尽。它们占用了大量的关键资源，阻碍了工作进展，成为了最要命的一种瓶颈——难以自拔，永无宁日。

规范书变更请求

我最喜欢的方法是“规范书变更请求”（Spec Change Request，SCR），它还有一个对等的名字叫“设计变更请求”（Design Change Request，DCR）。这种方法是委员会议和走廊会议的组合，同时带有一些关键的改进。假设你现在想去改变规范书或者给规范书增加新的内容，你的这个想法可能是你自己想出来的，也可能源于一次走廊对话，也可能受到了一次主管会议的启发。

不管你是项目经理、开发、测试或实施人员，你都可以把你的想法写到 E-mail 中去，并且 E-mail的标题定为“SCR：＜受影响的规范书＞-＜变更的简短描述＞”。在 E-mail 结尾的地方，你用粗体字写下这么一句话，**“除非有人强烈反对，否则这就是最新采纳的规范书。”**然后，你把这封 E-mail 发给最直接受这个变更影响的项目经理、开发、测试和实施人员。几天之后，当你根据他们的建议做完了必要的修改，你就可以把你的 SCR 发给团队中剩下的所有人，并且把它跟其他 SCR 一起放到 RAID 中或者一个公共目录中。

这里的关键是，规范书的变更现在文档化了，并且得到了相关人员的复审，而且不会阻碍工作的进展。偶有反对的意见，不会有很多。开发人员在任何时候都可以继续工作，以遭遇反对的风险换取时间。典型情况下，开发人员会一直等待，直到 SCR 经过了初始的几次修改后发给团队全体成员的时候才动手做。

预防是最好的治疗

当然，最理想的是规范书从一开始就不会迟到，至少你不能被它蒙蔽。这里就用得着 T-I-M-E 图表了。在 T-I-M-E 图表中，第一份规范书展示了对整个项目的设计。它不是简单的需求文档，也不是数个小型规范书的集合，而是项目的一个高级规范书，很像是开发主管写的那种高级架构文档。这个规范书应该展示项目将具有的功能和用户界面，以及它们怎样在一起协作，而把细节留给以后的规范书。所有后来的规范书和功能都必须参考这个高级规范书。

这样的话，开发、测试和实施人员就可以制订计划去说明未来所有的功能了。他们能够生产出集成得更好的产品，使用户体验更加流畅。项目经理也可以使用第一份规范书去安排剩下的其他规范书，先做优先级高的，而不必担心遗漏什么东西或者做出让别人吃惊的事情来。这种想法终究使 T-I-M-E 产生了（难以抗拒）。

> **作者注：** T-I-M-E（Totally Inclusive Mutually Exclusive）图表是由 Donald Wood 首先提出的，但它从未像我的同事 Rick Andrews 最初预期的那样流行过。然而，微软现在的价值主张、远景文档、跨产品案例和匠心独具的设计原型都能达到同样的目的。

2002 年 6 月 1 日："闲置人手"

你的开发团队两周前达到了"零 Bug 反弹"（Zero Bug Bounce，ZBB），你突然意识到，你又迎来了一个"工作淡季"。任何一个遇到过零 Bug 反弹的开发人员——其工作成果（产品）已打包出售，都了解这个"工作淡季"。但如果你的团队是提供互联网服务的，那么你现在可以停止阅读了。（等一下，你一开始读本栏目的时间哪来的？回去干活！）

> **作者注：** 零 Bug 反弹（Zero Bug Bounce，ZBB）指的是项目中所有的功能都已完成并且每个工作项目都已解决的那一刻。这个时刻很少会长时间持续。通过进一步的系统测试，通常在 1 小时之内就会有新的问题暴露出来，随后团队又必须埋头去工作。尽管如此，零 Bug 反弹意味着项目在可预见的将来就要结束了。
>
> 顺便说一下，我的团队现在做的是一个互联网项目，我们每个月都尽量开展一些激励活动，并分享各自的阅读心得及创意。这是个秘密——精益与敏捷，宝贝！详见第 2 章。

零 Bug 反弹标志着团队的掣肘由开发人员变成测试人员（如果问题在项目经理身上就没有类似的这种转变）。在处理完产品出货后两周内新 Bug 的冲击波后，大部分开发团队进入了"时而赶工，时而空闲"的模式——当有新 Bug 分配过来的时候奋力去解决它，否则就闲着不知道该做什么了。

最可怕的是，"工作淡季"有时候可能从零 Bug 反弹开始，一直持续到下一个版本的第一个里程碑。这在大项目中可能有几个月之多！开发经理手上总是忙着各种各样的事情，因而很容易就会忘了其实三分之二的团队成员都闲在那里。你知道他们怎么说闲置人手的——嗯，不是很好听！

宝贝干了件蠢事

以下是闲置的开发人员经常做的一些坏事：

- **偷猎 Bug。**零 Bug 反弹之后，你的团队应该处于"禁闭"状态，这意味着，所有 Bug 在被修复之前都要通过分诊会议的慎重考量。闲置的开发人员有时坐在他们的位置上，敲击

RAID（现在叫 Product Studio）上的 F5 功能键等待 Bug 的出现。如果通过这种方式没有发现 Bug，他们就会把视线转到正在被分诊讨论的那些 Bug 上，挑一个有趣一点的，然后开始研究它。在你知道之前，他们可能已经有了一个修复方案，并且正伺机悄悄地把代码签入进去呢……这就是偷猎 Bug！一个有自尊的开发者不应该做这样的事情。

> **作者注：**在软件工程中，Bug 通常是指代码中的错误。然而，微软内部使用 Bug 这个词泛指跟产品相关的所有增加、删除或者修改。但大家对外一般称这些为“工作条款”，其中有一些也可能是代码错误。我更喜欢“工作条款”的说法，这样就能把那些真正的 Bug 区分出来。

谁知道分诊团队是否会决定修复那个 Bug 呢？谁知道哪个真正的 Bug 被修复了呢，并且会不会引起相关的另一个或大或小的 Bug 呢？对于潜在的重大问题进行一点调研是可以的，但绝对不要偷猎！

- **修复尚未登记的 Bug。**现在有一个 Bug 通过了分诊，你正在进行修复。这时你注意到，在你修改的代码附近有其他更多的 Bug（通常这些 Bug 是由以前的修复引起的）。但不知怎么搞的，这些 Bug 还没有被人报出来。你看到了这些代码，而其中的错误也尽收你眼底。为什么不一起把它们都修复了呢？喂！就此打住！

开发团队通过代码复审来避免这种可怕的事情。在“可信计算”时代，团队应该在整个项目周期内复审每一次代码签入。当团队处于“禁闭”状态时，要保证有 3 双眼睛（即代码改动者本人和另外两个开发者）同时审查每一次的代码改动。至于开发人员在修复一个 Bug 时发现的其他 Bug，至于你发现的其他 Bug——登记，会诊，再跟踪处理。

> **作者注：**《凯文与霍布斯》（Calvin and Hobbes）连环漫画系列中有这么一个故事：凯文对一只苍蝇慈悲为怀，打开前门让它飞出去。结果呢？这只苍蝇非但没有飞出去，反而另外 3 只苍蝇飞了进来。这就是为什么从项目的开始到结束都要对每一个 Bug 进行研究和分诊的原因了。我的团队曾经在我们的产品发布前一个月的时候改变了一个参数的值，结果一周之后，全公司的测试人员都发现，只要打开 CD 托盘，所有的应用程序都会停止响应。最后，我们往回追踪到那个看起来无关紧要的参数，并把那个改动撤销了才解决问题。这种事情真实地发生在我们的周围，只是你未必知道而已。

- **修复标为“延期”的 Bug。**大家知道，被标为“延期”的 Bug 在产品发布给生产商（Release To Manufacturing，RTM）之前是不能去修复的。那么，是不是应该在计划下一版产品的时候去修复它们呢？不对！当初在项目的进行过程中，产品的相关团队对“哪些 Bug 对我们的客户影响最大，因此必须在发布之前修复”作了判断，但这种判断在产品没有真正发布之前是无法验证的。当产品发布之后，你就没必要再去猜了。“产品支持服务”（Product Support Services，PSS）、Watson 和“微软咨询服务”（Microsoft Consulting Services，MCS）会告诉你的，它们很中肯。那些标为“延期”的 Bug 只具有参考价值，用于理解为什么这些 Bug 当初没有去修复。但是不要猜测客户会怎么想。你要做的是，关注用户反馈，修复真正影响用户的那些 Bug。

- **重写“丑陋”的代码。**开发人员讨厌“丑陋”的代码。这些代码常常麻烦不断，可读性差，难以维护。因此，当开发人员手头有空的时候，他们经常自言自语：“哈，我手头没有规范书，因此不能开发新的东西。我为什么不趁此机会重写那些讨厌的丑陋代码呢?”他们知道，如果给第二次机会的话，他们能够做得更好。他们也的确可以做到。他们可以在第二次的时候，重写出漂亮得多、清晰得多的代码，而且比第一次写的时候少了很多 Bug。

令人遗憾的是，重写的代码实际上将比当前的丑陋代码带来更多的 Bug。因为当前的丑陋代码是在第一次编写的基础上，经过了几个月甚至是几年的测试和修复之后才达到的质量水准。

有时候重写是必要的。重写可以提高代码的性能、扩展性、可靠性、安全性，或者对于新技术的适应能力。在这些情况下，应该把重写当做一个功能来对待；像处理其他功能一样，写一份规范书，然后为它制定时间表。否则，不要做代码重写这种愚蠢的事情，它只会重新引入一大堆令人讨厌的 Bug，而且还对客户价值没有丝毫的贡献。

> **作者注：**我的上述观点同样适用于“重构”（Refactoring），尽管我很讨厌提到它。哪怕重构是在你毫无察觉的情况下由电脑自动完成的。这并不是说你不应该进行定期的代码重构或重写，而是说，你不应该随意地做这些事情。做不做都应该由团队来做决定，直到测试人员将新 Bug 确定在最低程度，然后再重构或重写。

- **在编码风格上争论不休。**谈到最消耗开发团队时间的事情，在空格、括号、匈牙利命名法等问题上的争论必定在前 5 名之列。请记住：使用一种一致的编码风格对你代码库的质量和可维护性大有裨益，而你的团队具体使用哪种风格一点都不重要。你是开发经理，你来选一个并坚持使用它。谁说这也需要民主!

告诉我该做什么

关于闲散时间不好的一面我已经说得够多了。在这个清静时期，你的开发团队可以做些什么有建设性的事情呢?

很自然，测试团队会坚持说，在产品发布给制造商之前的这段时间里，开发人员应该帮着找 Bug。但是大多数开发人员会感到找 Bug 是件很恐怖的事，即使在别人的代码里找。而项目管理团队会坚持说，在产品发布给制造商之后，开发人员应该花时间去阅读和复审规范书。不过，开发团队不会很乐意，也不会有激情与心思干这样的事。

那么，开发人员在“工作淡季”到底可以做些什么呢?下面是我的一些想法：

- **分析 Bug。**分析团队在过去的一个产品开发周期中修复过的所有 Bug，找出其中的规律。哪些是个人常犯的错误?哪些是团队犯的错误?团队中的每个成员下次需要注意点什么，才能开发出更好的产品?
- **为部门开发一些工具。**尽管开发人员通常不擅长发现 Bug，但他们在开发用于帮助发现 Bug 的工具方面的能力却是超强的。他们还能开发一些工具用于使过程更加顺畅，比如源代码签入、安装、构建和支持。给源代码插桩或者开发一个好的测试用具，能够大大地促进开发与测试团队之间的关系。当然，你应该先到工具箱网站上去查一查，看看满足你需要的工具是否早已存在了。

- **讨好项目经理，把他们的设计思想变成原型程序。**开发原型程序是个好主意，只是不要在常规代码库上去做。尝试用另一种语言来写，或者至少要有独立的构建。在常规代码库上开发原型程序的最大问题是，项目经理和高层管理者会很自然地认为，代码已经到了差不多可以发布的程度，而事实上，原型程序通常存在着本地化、平台依赖、徽标、漫游、性能、安全、兼容性等各种各样的问题。混淆原型程序和产品代码会把产品计划和期望混为一谈。相反，用另一种语言来开发原型程序却是一个极好的学习机会。再说还能……

> **作者注：**虽然以前已经说过了，但我还要再强调一下，“不要把原型程序当做产品来发布。”这么做不会节省时间，而只会花更多的时间。千万不要这么做！原型程序是用于学习和沟通的，它的用途就是这么单纯。除了用另一种语言开发原型程序外，我以前常常把 Esc 按键处理为异常结束。这样的话，如果我的上司在观看演示时表现得异常兴奋，我会按下 Esc 键，让程序崩溃，然后解释说，“很显然，我们的程序还没到发布的时候。”关于原型的更多内容，参见第 6 章的“我的试验成功了！（原型开发）”。

- **学习新技术或技能。**人们总是抱怨他们没有足够的时间去学习新技术或技能，抱怨得不到培训机会使他们自身得到提高。好吧，为什么不好好利用项目的这个清静时期呢？不要让机会在你身边溜走！
- **跟研究人员交谈。**在零 Bug 反弹之后是跟研究团队交谈的最好时机。这时候你有足够的时间去采用某个新技术，花时间去学习它，并且了解你能用到哪些东西。到你的产品发布并且开始计划下一个版本的时候，你可能已经把原型程序准备好了，并且解决了所有的风险，这着实让你的团队惊喜不已。另外，你和研究人员也可以为未来的产品策划新的研究领域。这非常有价值，而且做起来很容易。
- **撰写专利申明或白皮书。**还有剩余时间用来反思和记录你已经做过的事情吗？如果你团队中的一位开发人员在项目过程中想出了一个新颖的点子，产品因此增加了不错甚至是重大的价值，那么务必叫他撰写一个专利申明。这做起来很容易，很快，还能极大地鼓舞士气。请访问专利组织的网站，以便了解更多的细节。如果你想把一些信息文档化，或者与其他团队分享某个想法，那就写一个白皮书。相对来说，这做起来也很容易，但能给作者和你的团队带来尊敬和影响力。
- **反思职业生涯。**最后但并非最不重要的一点，项目的这个清静时期是你审视职业生涯现状的最理想时机。你现在处在你理想中的位置吗？你的职业生涯在向正确的方向前进吗？你准备好迎接新的挑战了吗？你需要做些什么，以使自己能够专注并能富有激情？如果通过上述反思，你觉得必须改变一下，那么，越早采取行动越好。

俭则不匮

这种情况差不多已经司空见惯了：两个版本之间的空闲时间白白地浪费了。而实际在这段时间内做的事情常常还是有害的。其实，只要稍加思量，就可以提升自身能力，提高产品质量，拓宽视野，并增强整个团队的水平，而不至自陷囹圄。为你的“停工期”好好计划一下吧，开足马力勇往直前！

2004 年 6 月 1 日：“我们开会的时候”

别浪费我的时间。拜托，请你不要浪费我的时间。有时我真想翻过桌子用胶带封住你的嘴，因为不想眼睁睁地看着你浪费我生命中宝贵的 60 分钟。召开一次会议怎么能让人觉得你是在浪费时间？如果时间是金钱的话，大部分会议都是不景气的市场。我很讨厌那些一下子能把车开下悬崖的人来召开一次会议。

我不想再参加这种会议了。如果你逼着我去一个会议室，请准备好我会质问你能想到的任何“问题”。你浪费我的时间，我也会让你的时间化为灰烬。不喜欢这样吗？别考验我！我会质问你什么呢？听好了，因为下面就是……

为什么我们会在这里

面对有点娱乐式的“聚会”，我要提的第一个问题是：“为什么我们会在这里？”我们聚在一起到底想干什么？有理由吗？如果你没有让所有人清楚这个理由，大家可能会想这次会议肯定很特别，并争先恐后要求分配会议时间，结果却一无所获。我不知道——也许你应该预先发一个议程，或者一些我们将要讨论的文档？拜托你做点什么！

如果你把会议的主旨作了通告，我可能还会提醒你，一定要坚持这个主旨！我不关心是否还有另外 50 个会议，去做另外的 50 个决定，讨论另外的 50 个话题或者更多的信息共享。我现在参加这个会议，我就只想专注于这个会议。如果有人想要谈论其他的什么事情，让他在我们结束之后再上演他自己的秀吧！

> **作者注：**如果有人试图转移话题，你该如何不失礼貌地打断他呢？我喜欢的方法是，对他说：“让我们先把这个话题讨论完，然后再谈你的话题。”通常情况下，当你结束了第一个话题之后，所有人都想离开了。那个会议干扰者将不得不另外安排一个会议，并且邀请合适的人参加（这样要好得多）。如果那个干扰者坚持认为应该先讨论他的话题，那么首先讨论他要这么做的理由（这时候会议实际上仍然聚焦在原先的话题上）。如果理由很充分，那么是你的会议时机还不成熟，因而是你的会议需要另行安排。在这种情况下，没人会介意离开当前的这个会议。

我们正在做什么

接下去，我的问题是：“我们正在做什么？”

- **我们是要做出一个决定吗？**很好，就让我们做个决定，其他的像头脑风暴、状态检查、流言澄清等统统跳过。
- **我们是要共享信息吗（比如一个状态汇报会议）？**很好，把信息列表上的内容过一遍，但不要试图去做什么决定或者解决什么问题。
- **我们是要收集想法吗？**很好，捕捉每个人的想法，无论它们有多荒诞都不要作出批评或论断。最后，挑出其中最好的一个即可作为会议的成果了。

这里的要点是，混合会议是低效的，常常会徒劳无功。要弄明白你们为什么会聚到一起来，会议正在试图达成什么目标。如果你必须改变这些前提，那你要深思熟虑之后，让每个人都意识到，现在规则已经变了。否则，你会浪费所有与会者的时间，问题永无止境地纠缠不清，最后还不得不重新召开会议。如果你要这么做，麻烦你不要邀请我。我不会参加的！

> **作者注：**有个典型案例，就是在 Scrum 会议上提出设计问题来讨论。Scrum 会议是用于共享信息的，不是用于收集想法或做决定的。像设计讨论这样的 Scrum 会议是个特例，它会经常转换主题。然而，因为设计讨论是很值得做的，我们在 Scrum 会议上会把讨论的话题在白板上列出一个清单。等 Scrum 会议结束之后，那些想讨论这些话题的人可以留下来，参与接下去的设计会议。

为什么这些人会在这里

很好，我们已经知道了开会的理由，也知道了我们正在做什么。现在的问题是，为什么他们会在这里？我指那些不该在这里的人。这些人在问一些多余的问题，他们只是在重复别人的观点，而他们只需在必要的时候表示一下同意。那么，为什么这些人会在这里呢？

会议的持续时间跟参加会议的人数是直接成正比的，说不定它们的关系还是线性的。你应该只邀请那些必须出现在会议上的人。

- **试图做一个决定？**邀请可以做决定的人。其他所有人都可以在会议之后，通过 E-mail 了解到会议的情况。需要做决定的人还没有到齐吗？那就取消会议。立即取消！否则，你将不得不在所有人都能参加的时候重新召集会议。
- **状态汇报会议？**邀请那些将要汇报状态的人。其他所有人都可以在会议之后，通过 E-mail 了解到会议的情况。有一些需要汇报状态的人不能参加吗？我猜他们肯定是懒鬼。
- **头脑风暴会议？**邀请一些有创意的、思想开明的人，他们是会议成功的关键。其他所有人都可以在会议之后，通过 E-mail 了解到会议的情况。

有时候，你必须邀请一些其他人来参加会议，他们对会议是否成功起着至关重要的作用，比如策划人、协调员、带头支持的人等。但也就这些人了。如果有太多其他人登记要参加会议的话，你最好取消会议。（你可以预先判断出将有多少人参加会议，因为当有人接受一个转发的会议邀请时，你会收到一个确认信息。）

尽量预定一个小型会议室，这样可以把一些不速之客排除在外。尽量把会议安排在 30 分钟内结束，这样可以让大家准时出席并且保持会议过程的高效进行。你可以声称这是一个“工作会议”，必要的时候甚至可以使用“信息权管理”（Information Rights Management，IRM），以阻止会议邀请信被转发。

为什么我现在才听到这个

对于一些重要的主题，你不想让关键的相关人员措手不及。没人喜欢匆匆忙忙地被拉去做至关重要的决定，也没人希望对关键领域发生的事情丝毫不知。如果你想要会议开起来比较顺畅，那么预先跟关键人员做个沟通。你们可以发现问题，协商折中方案，预先让所有人都取得一定的共识。接下去，会议更多地只是一个形式了。对于要做出决定的会议来说，这是个好方法，只是

它比较费时。但对于一些至关重要的决定，这也是至关重要的一步。

接下去要做什么

现在会议开完了，结束了，一切都过去了，对吗？错了！会议像好莱坞恐怖电影中的鬼怪蛇神一样，它们会重新获得生命，然后吃掉剩下的那些人。决定接下去要做什么，把它们写到E-mail中去。这样才能使已经结束的会议不会死灰复燃。

写 E-mail 的时候，把所有与会者的名字放在地址栏中，并且抄送所有受会议结果影响的人。务必要包含一个对会议所做的决定、共享的信息或者收集到的想法的简短总结。然后列出接下去的安排，指明谁在什么时候做什么事情。此刻，你终于可以放心地继续前进了。

看到了吧，尊重别人的时间并没有那么难！会议在很多方面是有价值的。当然，会议对于加强组织的沟通也是必要的。但如果你要开会，那就把会开好。所有人都会赞成开会，你也能完成更多的事情。

2006 年 7 月 1 日："停止写规范书，跟功能小组呆在一起"

我不是项目经理（Program Manager，PM），我也从来没有担任过项目经理，我将来也不可能成为一名项目经理。这并不是因为我个人对项目经理的抵触，其实，我的朋友之中不乏出色的项目经理。很显然，我没有权利去教导项目经理应该怎么去做他们的工作。

尽管如此，项目经理应该停止写规范书。就这么简单！他们在浪费我的时间，浪费组织的时间，浪费整个公司的时间。你几乎可以听到残留着的、细微的、嘎吱嘎吱的声音，因为规范书像白蚁一样在一口一口咬掉公司和客户的价值。我对此深恶痛绝！

其实不仅仅是项目经理，开发者也必须停止写开发规范书，测试者则必须停止写测试规范书。疯狂必须停止！浪费必须停止！我们必须反省，保持头脑清醒，重新焕发我们的生产力。

> **作者注：**本栏目肯定是我发表过的最有争议的栏目之一。你也可以从接下去的那段文字看出，我当初就猜到了这一点。关于我的观点，大家最大的误解是在"正式文档"和"非正式文档"的区别上面。我认为，跨工种的团队只需要非正式的文档，比如把白板上的内容拍了照片放到维基网站上，并加上少许的注释。而不同地点不同部门的团队则需要正式的文档，比如具体的规范书。

你失去理智了吗

"你肯定不是认真的？"我听到忠实的读者这么说，"你多年来一直鼓吹质量（参见第 5 章的'牛肉在哪里'栏目）和设计（参见第 6 章的'通过设计解决'栏目）。你告诫开发人员在他们拿到规范书之前不要采取行动，在没有彻底理解设计之前不要开始编码。莫非现在你承认以前是被误导了，或者甚至可能你本身的认识就是错误的？"不，当然不是。

功能团队在开发用户体验之前必须首先理解它，开发人员在实现一个内部设计之前也必须首先充分理解它，这样才能面对面地解释给同伴听。但是，这些步骤中都不需要正式书写的文档。

为什么我们需要正式书写的规范书呢？客户不需要它们。市场和产品规划部门也不需要它们。即便是内容发布者和产品支持人员，他们对规范书的使用也是有限的。因此，谁需要这些荒唐无用的“杰作”呢？为了找到答案，我们不妨把规范书扔掉，看看谁会叫起来。

进退两难

如果我们不再有规范书，开发和测试人员会大叫，“我怎么知道代码应该实现什么功能呀？”告诉他们找项目经理讨论去。然后他们接着抱怨，“项目经理又不会整天在我的办公室里转悠。我需要记录下来的规范书。我必须对它们进行复审、修改和更新。”

是的，这里的确有问题。不是开发和测试人员必须有规范书以便复审、修改和更新，而是项目经理没有整天留在附近，一起来讨论用户体验、实现和测试策略。好吧，那如果项目经理这么做了又怎样呢？

如果项目经理跟开发和测试人员整天呆在同一个开放区域里，并且周围摆满了白板，一起为同一个功能集合努力工作，又会怎么样呢？我们还需要正式书写的规范书吗？等等，我听到了更多的尖叫声。

特殊要求

如果没有正式书写的规范书，依赖这些功能的其他团队将会抗议，“如果我们不知道你的代码是怎么工作的，我们怎么知道如何去使用它呀？”这问题问得很好。如果项目经理整天跟功能团队呆在一起，他们也就不可能有时间去应对所有的下游团队，而我们也不可能把所有人都塞到同一间房间里去。然而，下游团队其实不需要规范书——他们需要的是一个小型的软件开发包。不管怎么样，组件团队都得提供软件开发包的，这么做非常有价值。

如果没有正式书写的规范书，“合规警察”（Compliance Police）将会咆哮，“<在这里插入你最喜欢的官僚栓剂>的文档在哪里？”这问题问得也不错。合规警察让我们远离伤害。尽管他们的工作不怎么讨好，但却非常重要。他们常常需要正式书写的文档来完成他们的工作。然而，合规警察同样也不需要规范书。他们需要的是完整的合规文档，跟规范书相比，它常常以不同的形式包含不同的信息。

> **作者注：**这些“合规警察”是谁？他们其实是普通的工程师，只不过他们的工作重点是，确保微软的产品在关键领域的正确性，比如安全、隐私、全球可用（没有不合适的委婉语或引用）和遵从所有适用的法律和法规，等等。举例来说，他们要求的典型文档包括：威胁模型（安全方面的）、隐私声明、专利权使用条款等。

在上述两种情况下，你都不需要正式书写的规范书。你需要的是其他类型的特定文档，而这种文档会比较容易写，因为它不会没完没了。

我不记得了

那么我们还需要正式书写的规范书吗？我“不记得”所有的状况了，因此让我们来理一下：

- 项目经理在团队的房间里度过他们所有的时间，跟功能团队一起讨论用户体验、实现和测

试策略。

- 功能团队为下游团队写一个小型的软件开发包。
- 功能团队填写必要的合规文档。

我把它们都写下来了，看起来很不错。不过，等一下，这里有个问题。

人们常常健忘。你不得不把想法写下来，尤其当你经常在不同的项目之间切换的时候。很自然，如果一个功能从开始到结束要花费几个月的时间，在这期间可能会有人离开团队，那么信息岂不是都丢失了！

坚持做一件事情

但如果你一次只做一个功能会怎么样呢？那花费的时间就不会那么长，你也不会在项目之间来回转换。团队中有人离开的几率会小一点，而把想法记在脑子里也会容易得多。你只是需要用数码相机把白板上的任何内容拍下来，然后放到一个维基网站上或 Word 文档中或 OneNote 记事本中。

这看起来像是规范书，只是没有了让人头脑发麻的长篇大论。它给你留出了更多的时间去思考，以及在白板前合作，而减少了你在座位上摆弄像素和文字的时间。

很好，你把功能团队聚集在了相互靠近的区域，并配备了大量的白板。你一次只做一个功能，直到这个功能做完。你用相机把所做的决定存档。你撰写了对下游团队有价值的专门文档。这听起来像是精益软件开发（你可以在第 2 章的“精益：比五香熏牛肉还好”这篇文章中了解到更多的内容）。妙极了！这就是你停止浪费之后所得到的。

你准备好了吗

可能极少有团队马上停止写正式的规范书。他们还没有接受“功能小组”（Feature Crew）的概念，即一次只做一个功能，并且从头到尾把这个功能做完。他们不能呆在同一个团队房间里面，主要靠白板来相互沟通。

然而，变化已经开始了。各个部门正在调整位置以相互靠近，因为这样做起事情来更快、更容易。各部门也正在组织功能小组，因为这样做可以更快地得到更高质量的功能，并且留下较少的未完成的工作。顺着这个趋势，把它们不断整合，那你可以把规范书永远抛弃了。这不是梦想，而是简单时代的回归，是久经鏖战的智慧结晶。

作者注：我作为 Xbox. com 项目开发经理时接手的第一个棘手事情是协调 6 个 Scrum 团队，包括将立体墙搬到每个团队的房间中。在这次办公室调换之前及之后数个月，这 6 个团队一直一起奋斗在 Kinect 及 Windows Phone 两个项目上。这次 Scrum 团队得出的时间及速率数据是度量协同合作效力的最佳良机。其中 5 个团队的四周平均速率提升 20% ~63%（第 6 个团队发布了 beta 版，因新项目的开发而暂停），提升 20% 就像每星期另增了一天的生产能力，并多出了两天的周末时间。提升 63%（那就是每星期多了 3 天!），同时缩减了工程长度。不要预先分配其他工程项目，让第一个空闲的人自由选择，这样就有助于跟其他人进行跨部门的工程合作。团队唯一要做的文档工作就是后期维护记录。用 OneNote 记下来，并做好基本的“合规事宜”。

2007年2月1日："糟糕的规范书：该指责谁?"

规范书基本上都是可怕的。不仅仅指项目管理规范书，而且也包括开发规范书和测试规范书。我说的"可怕"，主要是指难以撰写，难以使用，而且难以维护。你要知道，这很可怕！规范书往往还是不完善的，组织得很差，并且没有得到充分的复审。规范书永远都是这个样子，也看不到有任何变好的迹象。

为此，我想要指责项目经理，部分原因是因为我喜欢这么做，但更主要的还是因为他们是糟糕规范书的始作俑者。然而，事实不允许我指责项目经理。人人都在写糟糕的规范书，而不只是项目经理。即使项目经理偶尔写出了上好的规范书，但其他大部分的还是很糟糕的。不管谁来写，上好的规范书总是难以撰写和维护的。

如果项目经理不该为那些以次充好的规范书承担罪责，那么该指责谁呢？管理层将是最直接的目标（另一群我喜欢指责的人）。事实上也是，有些组织，比如Office部门，一贯以来写的规范书都比其他部门的要好。因此，管理层显然在其中发挥着作用。然而，Office部门这么多年来调换过很多次管理层，因此，这里的原因必定比主管的人更加深层次。

树立靶子

现在清楚了，我应该去指责写规范书的过程——我们是怎么写规范书的，以及我们使用什么工具。这个过程是烦琐、艰难并且乏味的。规范书模板冗长、吓人，复杂到无法驾驭。所有这一切使得要写成一份上好的规范书，比披着皮大衣、穿着拖鞋想要去赢得一场马拉松比赛还要无望。

一些落后于时代的杞人忧天者可能会说："写规范书的过程如此荒谬紊乱是有原因的。所有的模板元素和过程步骤都是用来避免以前发生过的灾难的。"看到了吧，你不必去担心从公司高层传达下来太多的官僚主义，其实，在下面的基层已经自己累积了很多。

病态的过程总是源自于最好的意图。麻烦出在，这一路走来，最初的目标和意图都不知不觉地迷失了。唤醒这个目标和意图，使用新的、更好的方法去实现将会使它们重归正途。

> **作者注：**我在波音公司工作了5年。在那里，不是所有但绝大部分官僚主义看起来都来自于公司高层。我已经在微软工作了12年。在这里，不是所有但绝大部分官僚主义看起来都来自于基层。我们在最底下的层次，工作独立，很自由。有时，这意味着我们这是在作茧自缚。

沟通分解

所有项目管理、开发和测试规范书的目标，都是为了跟不同时间、不同地点的人进行设计和设计决定的沟通。我们想要使这种沟通变得容易而稳健，并且进行大量的反馈和质量检查。

假设你还没注意到，这里有4个独立的要求：

- **容易**
- **稳健**
- **反馈**
- **质量检查**

每个要求都可以通过一个不同的解决方案来满足。"我们只需在规范书中多加几节来满足所

有的要求”，这种方法跟“我们只需给这个类多加几个函数来实现所有的功能要求”一样的愚蠢。我们还是对这些要求逐个来分析吧。

保持简单容易

规范书必须要容易写，容易懂，并且容易维护。它应该使用标准的符号（比如 UML）来绘制图表，使用通用的术语用于文字表达。它不应该承载太多的内容或者说得过于深入。

格式越简单越好。在“工程卓越手册”（Engineering Excellence Handbook）中的通用规范书模板有 30 节之多，并且还有 3 个附录。而上好的 Office 规范书模板也有 20 节。它们都太复杂了！

一份规范书只需要 3 节，再加上一些元数据：

- **需求**：为什么会有这个功能？（跟应用范例和客户角色联系起来。）
- **设计**：它怎么工作？（图片、动画和图表特别有用。）
- **问题**：做了什么决定？风险和权衡都有哪些？（比如依赖关系。）
- **元数据**：标题、简短描述、作者、功能团队、优先级、成本和状态等。

就这些了！“状态”元数据可以是个工作流，也可以是个检查清单，但不能太复杂。

“但威胁模型放在哪里？还有隐私声明呢？源代码插桩或者性能度量呢？”我可以肯定你会提出这些要求。请你冷静一下！这些条目属于质量检查，我在后面很快就会谈到。规范书结构本身是简单的，比实际需要的多一点或少一点都不行。这样才能容易写，也容易被人读懂。

> **在线资料**：规范书模板（Spec template. doc）。

变得稳健

规范书必须是稳健的。它必须是可被验证的，并且所有定义的功能需求和质量需求都能被满足。你可能会问，“怎么做到？”但“怎么做到？”究竟是什么意思？怎样在第一时间验证需求？那就要编写测试，对吗？对！这就是你怎么写出一份稳健的规范书的方法。在规范书的第一节中，当你列出功能和质量上的需求时，应该包含如下内容：

唯一标识	优先级	功能或质量	简短描述	相关应用范例	用于验证需求的测试

如果你不能指明一个测试来验证一个需求，那么这个需求就不能被满足，因此要把需求丢弃。不能丢弃？那好吧，重写需求，直到它能被测试为止。

> **作者注**：我相信，一个可靠的设计把“测试”和“需求”放在基本同等重要的地位。每个需求都应该有一个测试。每个测试都应该基于一个需求。这将导致，清晰而可被验证的需求；更加全面的测试；一致的完成标准（即所有测试通过等于所有需求得到了满足）；以及更好的设计，因为测试驱动的设计自然要更加简单，更加内聚，具有更松的耦合。

获取反馈

在规范书付诸实现之前，越多的眼睛看到它，它就会变得越好，并且将来要求的返工也会越少。如果你想很容易就能获得反馈，你也想别人很容易就能提出反馈，至少你要将规范书草稿放

到 SharePoint 上，并进行变更跟踪和版本控制。做得更好一点，你可以把规范书放到一个维基站点上去，或者贴在功能团队主要活动区域的一块白板上。

撰写规范书的这个过程、反馈和变更管理需要有多正式呢？像我在前一个栏目讨论的那样（即本章的“停止写规范书”那个栏目），正式的程度取决于沟通是否直接，以及沟通的“带宽”有多大。在同一个共享区域、同一个时间、工作在同一个功能的人们，可以使用很不正式的规范书和过程；而在不同时区、不同时间、工作在不同功能的人们，则必须要有高度正式的规范书和过程。

不管怎么样，你想让规范书随时都可以修改，直到团队认为它不需要再改为止。那你怎么知道规范书不需要再改了呢？答案是，要等到测试团队验证规范书通过了所有的质量检查。

集成质量检查

这是我们目前正在使用的规范书最离谱的地方：各个部门不是把安全、隐私和许多其他问题当做质量检查，而是把它们一节一节单独地写在了规范书中。这是个灾难，原因是：

- 规范书变得更大并且复杂得多。
- 作者必须在各节中复制信息。
- 读者对后面的节次重视不够，导致严重的质量缺口。
- 设计变得难以理解，因为它们的描述分散在多节之中。
- 错误和缺口很容易被忽略，因为没有一个节次对设计作了完整的描述。
- 更新几乎是不可能的，因为一个最新的改动会影响多个节次，牵一发而动全身。

取而代之的是，采用适合于每一份规范书的质量检查，它以一个清单的形式出现，并且每个人都能触手可得。一开始的几个检查对于每个团队都是一样的：

- ✓ 需求清楚、完整、可验证并与有效的应用范例关联了吗？
- ✓ 设计满足了所有的需求吗？
- ✓ 所有关键的设计决议都被解决并存档了吗？

接下去的一套质量检查也相当基本：

- ✓ 所有的术语都被定义了吗？
- ✓ 安全问题解决了吗？
- ✓ 隐私问题解决了吗？
- ✓ 用户界面是否有亲和力？
- ✓ 为全球化和本地化准备好了吗？
- ✓ 对于响应和性能的期望是否清楚并可测量？
- ✓ 源代码插桩和可编程能力定义清楚了吗？
- ✓ 有没有兼容性问题？
- ✓ 故障和错误处理解决了吗？
- ✓ 涉及安装和升级问题了吗？
- ✓ 维护问题解决了吗？
- ✓ 备份和恢复问题解决了吗？
- ✓ 是否有足够的文档用于支持故障检修？
- ✓ 有没有潜在的问题会影响打补丁？

各个团队还可以为他们自己或者他们的产品线增加更多的质量检查，解决他们常常面临的特殊的质量问题。

在线资料：规范书检查清单（Spec checklist. doc）。

这里的关键是，设计节次对功能进行了完整的描述，而质量检查保证了没有东西被遗漏。是的，这意味着设计节次可能会变得非常庞大，以覆盖所有需要涉及的领域。但这些领域不再把每项功能的质量要求再展开赘述（比如对话框的安全、对话框的隐私、对话框的亲和度等）。

取而代之的是，文档中的相关区块都可看成是功能的逻辑性组件（比如应用程序编程接口、对话框、菜单等）。重复消除了。每个功能组件完整地被描述出来，并且所有的质量要求都融合在了设计情境中。

> **作者注：**一个奇怪而有趣的巧合。在本栏目发表之后的第二天，Office 部门把他们的规范书模板做了简化，只使用单独的一个节次来描述设计，并且采纳了我发布的质量检查清单。尽管我不能对这个改变邀功，但我的成就感着实得到了满足。

差别在哪里

如果增加了所有的这些检查和测试，你可能会怀疑我根本就没有对规范书作出简化。其实，这里的最大改动是：

- 节次的数量减少到了只剩下 3 个（需求、设计和问题）。
- 设计在一个节次中得到了完整的描述。
- 所有功能和质量上的需求都可以被验证。

我也已经谈到了，我们还有机会把规范书写得不那么正式一点，并且更容易被理解。

那么谁该为糟糕的规范书负责呢？其实我们都有责任。不过，糟糕的规范书主要还是由不良的习惯和糟糕的工具造成的。只要做一些小小的改变，并且使用大大简化的模板，我们就能改进我们的规范书，改善部门之间的沟通，并且促进工种之间的关系。总而言之，这样做可以让我们在微软工作得更加开心而富有效力。

2008 年 2 月 1 日：“路漫漫，其修远——分布式开发”

如果你是软件极客，就像我，很自然就会成为朋友与家人的技术产品顾问。当你看到你的家人备受软件折磨时，特别是这个软件是你参与编写的，至少你可以告诉他们：“让开一下，我是计算机科学家。”接着解决了所有的问题。是的，当你撤销他们所做的失败“操作”时，你可能会战战兢兢，但是你还是会及时摆平所有问题。但前提是，你就在他们身边不远。

如果你老妈住在千里之外，你们的通话很可能像下面这样：

老妈：亲爱的，他们更改了我的密码，现在我的 E-mail 进不了了。

我：好的。登录你的计算机，打开 Outlook。

老妈：密码是什么？

我：输一个你通常用过的。

老妈：我不是在申请一个新邮箱，是我的老的邮箱。

我：知道。打开 Outlook。

老妈：我从哪里输入密码？

我：在你输入用户名的主页面上，输入密码。

老妈：什么是主页面？

我：等会。你现在能打开 Outlook 吗？

老妈：是的。我已经打开它了。但是你说叫我登录计算机。

[另外又要花一小时在菜单、对话框及按钮操作上]

如果你有个像远程协助（Remote Assistance）这样的工具，那以上情况就不会这么糟了。但使用这种工具在防火墙及路由器后面操纵同样会非常有趣。本来 5 分钟可以处理的事变成了耗时 1 个小时的噩梦，那会怎么样呢？在你工作中的任一个时间点与地点，这种时间消耗与麻烦会成数量级倍增，这样就将我们引入一个话题：分布式开发。

大家不用呆在一个地方了吗

在全球多个地点共同对一个项目进行开发已成为稀松平常的事。但，这样虽然增加了团队开发的多样性，吸收了各界精英，这应该会是个巨大的优势，结果往往是挫折、延期及质量与机能的错位。为什么？因为有三个大麻烦：带宽、地域分隔及大家齐聚一堂。

畅通的沟通及快速访问中心服务所需的带宽还远远不够。分布在不同地点的项目分工界线不明，导致额外的沟通、冲突、灾难及梳理工作。因为不同的团队互相分隔，大家呆在一起一对一面谈，接洽来客，走廊交谈及其他日常交流这样的情况就不会再有了。

有了这些麻烦，你可能会想：为什么我们还要劳烦分布式开发？不，你想得太简单了，这跟钱无关。采用分布式开发的根本原因是源于人才与市场。计算机科学人才在巴西、俄罗斯、印度、中国等地每年增长近 20%，其总和相当于 2010 年美国软件工程师总量的 1.5 倍。他们不会全部跑到雷德蒙去，我们也不想他们这样，因为他们的本土市场对我们至关重要。

所以，如果你现在在带领一个团队，你最好弄明白怎么处理这三个大麻烦。以下让我们对它们逐一进行讲解。

我必须不停地解释，不胜其烦

第一个大麻烦是带宽——人与人及计算机与计算机之间的带宽。人与人之间、团队与团队之间通畅的沟通对于成功至关重要，我之前多次谈到这一点。在同一个房间里两个人之间沟通的信息量要远远大于人们通过电视电话会议（VTC）沟通的信息量，而后者大于通过 Live Meeting 沟通的信息量，Live Meeting 又大于电话（就像上面我跟我妈的电话沟通），电话大于即时通信（IM），IM 大于 E-mail。换句话说，E-mail 是个信息量最少的沟通工具，所以它的应用自然最为广泛。

> **作者注：**使用微软的 Live Meeting（现在叫做 Lync）进行圆桌会议，现在来说是挺好的，虽然其比不上 VTC。如果你还没试过圆桌会议，你就错失了体验很酷的技术的机会。它会使人们自动地把注意力放在言语自然通顺的人身上。

事实上，人们都使用 E-mail，因为它的方便性、异步性并可以跨时区。遗憾的是，E-mail 的带宽很小，通信质量差并且需要几经周折才明白对方的意思。因为时区的不同，一次邮件往返就是一天，因此，通过 E-mail 交流进展踯躅。计算机间的带宽也是相当慢的，这意味着在雷蒙德快速的源代码控制、文件或数据库操作要历经几个小时才到达大洋彼岸。

你怎样才能消除带宽这个大麻烦？这里有两种方法——增加带宽或削减不必要的通信。

- 你可以通过使用高带宽的通信工具，如 VTC 及 Live Meeting 来提高带宽。这些工具甚至可以在低带宽线路上工作。然而，这些工具是实时同步的，所以你必须同分布在全球各地的团队人员预约交流时间。这意味着如果你在雷蒙德而你团队的另一个成员在中国，你必须约定每天太平洋时间下午 4～5 点与你的中国同事进行即时会议。

> **作者注：** Live Meeting 已与 Office Communicator 在微软的新产品 Lync 上集成。用这个可以进行即时通信（IM），非常酷。视频及音频讯息、IP 电话、会议呼叫及 Live Meeting 等都集成在了一个 Office 应用套件中。呵，我看起来像个推销者，但这个确实很诱人。

- 你可以通过书写更细致的 E-mail 来削减必要的通信。可以预先在 E-mail 中提供额外的信息并回答一般性问题来减少问题，消除混乱。这样写 E-mail 会多花些时间，但谈不上几天时间，而这在即时通信上都是要花上几天的。
- 你可以用 SharePoint 发布项目信息来减少类目繁多的 E-mail 及打电话的次数。将主题“怎么做”、讨论目标、检查列表、文档及项目相关邮件、时间表及进度状态放到上面。无论何时何地，新的团队成员到来时，把完善后的上述相关材料及缺少的文档更新到网站。
- 你同样可以对在不同地方进行的工作建立清晰的项目界线。把本地通信分离出来，包括将源代码控制及数据库存储下来。这样你就只需要在人们或信息跨界的时候再使用跨团队通信。这又将我们带入到下一个大麻烦。

你确实不知道你在浪费时间吗

第二个大麻烦是界限——不同工作地点间的界限。

如果你从未在分布式的项目或团队中工作过，你可能会傻傻地认为分布式的团队都是过得很开心的。就是因为这样单纯的想法让项目经理允许团队中的成员进行远程工作。记住，无知、单纯、愚蠢是成为傻瓜的三部曲。

一个分布在三个不同地点的项目团队不是个开心的团队；在一个很可能成功的项目上工作的三个团队才可能是开心的，这样的团队工作目标一致，认识统一。

因为如前所说带宽的问题，你做不到使团队认识统一、步调一致，除非你将你的队员们集中到一个房间里。这就是为什么有一个共同的协作空间是最好的。也是团队只分布在不同楼层一起应对相同的困难而不是分布在不同的大陆的原因。

要使一个分布式项目能够成功，你必须在分布式的团队间设立清晰的界限，从而有足够的空间独立性使他们独立工作而只需要很少的协作交流。界限越分明，成果越显著。我在第 6 章的“美妙隔离——更好的设计”对此有更详尽的讲解。

> **作者注：**在分布式的团队间建立清晰的界限并不是禁止界限之间的沟通。你仍然可以进行跨团队的设计、代码复审与计划安排以及头脑风暴。各团队可以相互帮助解决问题，并使之相互联系更紧密。清晰的界限的作用是使团队间不至于天天相互阻挠与干涉。相反，高带宽的交流是必需的，可能一星期就一次，跨团队的协作只是锦上添花之事。

通常，界限是个相对于空间与地域的概念。但是同样也可理解成版本或责任的界限。你同样可以让分布式团队关注本地市场，这不会增加不必要的费用，这里的沟通费用可忽略不计。相反，它减少了不必要的冲突、灾难及梳理工作。

但是如果你的团队是绝对独立的，你怎能保持团队统一协作并相互共享，向着相同的目标前进呢？这就是下一个大麻烦中要说的。

能让我看到你的脸，肯定非常不错

第三个大麻烦是——互相会面建立起人与人之间的紧密联系使合作与沟通更真实。

最大限度地利用带宽并分清界限将解决分布式开发的大部分难题。然而，你们始终还是在同一个项目上为同一个目标工作的团队。你必须维系团队间、同事间的重要关系，并绘制出这种人员关系结构图，图中的相关人员需用人脸照片标示。互相会面是相当重要的。

VTC 及其他实时交谈工具对此是很有帮助的。你可以经常在一对一交流场合及会议中使用这些工具。并不必每次都用，但至少一月一次。记住，不同的工作地点之间要实现这些功能需预先约定好时间。

当然，什么也无法替代人与人之间的实际接触，所以面对面的实际会谈每年至少要安排两次。在 10 月安排一次公司会议及执行力审查，在 2 月或 3 月安排一次年度财政及预算计划。每次会面不只两三天时间，一次得 1～2 个星期。会面内容包括激励斗志、一对一会谈及培训，跟计划及审查会议一样。你们要抽出尽量多的时间进行接触。

有些团队会进行轮换工作或人员交换。在这种情况下，来自某个地方的团队人员要用 3 个或甚至 6 个月的时间与另一个地方的团队人员共处。这可以是一次人员调换或指派。这种人员轮换可以提供充分的相互学习的机会，从而建立团队间的互谅，使之相濡以沫保持联系。

最后，有时候制造一种幻象对于维护人际关系，加强沟通非常有效。一些公司已经尝试在普通场合持续播放视频片段。一个在微软行之有效、既省钱又有趣的办法是在人员密集场所放一个很大的乃至真实人物大小的画报。当人们结束会议或在大厅里踱步想些有关项目的事情时，他们看到这些画报就会得到一种暗示，在假想中将这个人或团队囊括在会话中。

太阳下山，你在何方

稍花精力，你就可以在你的项目中应用分布式开发，机会稍纵即逝。分布式开发对于你及微软来说将收获颇丰。

这么多种的体验、文化及想法最初可能使我们的团队及你很不适应。但是随着时间的流逝，你会渐渐感觉自身能力的提升，团队水平的提升，以及产品及服务水平的提升。你工作的成绩将在更多的市场、更多的受众中更受欢迎，在此过程中你将有所学，并逐渐成长。花些时间好好琢磨这三个大麻烦，那么你自己、你的团队、你的工作将会给世界来个惊喜。

2008 年 12 月 1 日：“伪优化”

为什么？为什么！为什么项目经理会做这么愚蠢的决定，到处改动代码，结果却毫无起色？不仅仅是项目经理，虽然他们在提升性能方面还算是个能手——构架师、团队负责人以及其他一干人等在肆意挥霍大家的生命，做些无用功，做毫无头绪的工作，却不能让我们提供真正有价值的东西。是什么原因造成这样的闹剧？很简单。我们在不断优化——优化我们本身一塌糊涂的东西。

是哪些白痴在优化他们本身就种下的祸根？就是你们这些人。你，你的朋友，你的老板。这些人都是出于善意却铺就了通向灾难的道路。我们在优化原本错误的做法却成就了最终错误的结果。这样做是错误的，也是可以避免的。但是，同志们，为什么你害怕一点点的努力会造成蓄意的恶行？

你想知道答案

我还是少找你的麻烦，省去些长篇累牍，先给你个答案——向着期望的结果进行优化。这听起来超简单也很显然，但人们总是自以为是地误入歧途，我还是一字一句地逐个讲来。

- **优化** 定量测定一下你们现在做得有多好，分析一下你们怎样能做得更好，改变一下方式，然后再测度一下。微软对于优化很有一手，但是我们只度量易于掌控的事而不是探究问题真正出在哪里。所以，优化后却得到一个错误的结果。可以在第 2 章的“你怎么度量你自己”中了解到更多内容。
- **意图** 意图要明确。为优化而优化是纯粹自我满足的表现——不要这样丢人现眼哦。相反，三思，再三思。弄明白你要做什么。专业一些，清醒一些。
- **期望** 把心思放在你想要什么，而不是你不想要什么。这里有个误区。人们总是围着问题谈优化而不是解决方法。官僚主义及慢速的软件都是由此而来。他们把精力放在如何驾驭人或代码以防止有错误行为。他们应该把精力放在如何使正确的行为更快更容易上，然后发现其中的异常现象。
- **结果** 绝对不要对某个步骤或算法单独分割进行优化。相反，对你期望的最终结果进行优化。我们都体验过局部优化而非总体优化的影响。它扼杀了我们的效率及创新能力，它将摧毁我们的星球。但三两下之后，人们往往就不能摆脱眼前的问题而去顾及他们真正想要的结果。

你能明白什么时候你会伪优化吗？让我们看些例子，再核实一下。

> **作者注：**我意识到，在一个专栏内同时谈论优化代码及优化人的问题会有点混乱。我无法抗拒这么做，因为这二者间千丝万缕的关系正与日俱增。如果对你来说分清二者关系并不难，那就只要关注一下你所在意的问题。

我想，我可以处理

你怎么处理代码的运行时错误？在没有错误出现的情况下你的代码运行有多快？在一次并无错误出现的操作过程中软件运行是顺畅还是经常卡？以最简易最快速的方式完成代码运行有 80% 的可能，而不是 20%。但是，这并不意味着你在错误处理上蒙混过关，你只是没有对它进行优

化。要相信，你的代码将以零错误运行，要让它的运行速度更快。对零错误加以验证，确保结果正确。要相信但也要验证。

同样，你是怎么处理人为的与工作过程中的错误？你对其过程中的每一步都进行检查了吗？是否让别人按你的方式，按部就班，并拿出大量的表单来证明他们并没有在骗人？或者，你相信别人能做正确的事——能清除道路上的重重障碍，并随后能验证他们做得没错？相信他们但是要验证。

我无私的读者，包括项目经理，可能会声称他们很信任他们的同事。真的吗？上一次出错时你是什么反应？你是很快修复病根并继续开展工作，还是马上展开一次为期几天或几星期的调查而打乱了团队的工作进展？你是事无巨细亲力亲为还是放手委托给别人？你是每个步骤都要干涉还是只对结果进行评定？信任没那么容易。幸运的是，好人有好报。

似曾相识

你的代码是松散的还是内聚的？是否所有的类、方法及功能都乱成一团麻，还是它们可以分开进行独立测试，每个代码段可以执行各自的目的及任务吗？

有良好架构及分层的代码是易于测试、维护及改进的。但，它不如紧耦合代码的效率高。要有权衡。如果你纯粹为速度而优化，最终你得到的是一团难以维护的面条代码。如果你只为清晰的架构而优化，你就在执行效率上缺乏竞争力。你怎么把握个中尺度？

大部分团队并没有把握好架构与效率之间的尺度——他们在玩过山车：

1. 团队采用了一个非常好的架构。运行良好，每个人感觉都很好。
2. 他们对它进行了性能优化。现在效率更高了——原来的团队确实不够专业。
3. 代码变得难以管理，难以改进，性能遇到了瓶颈，所以这个团队只好痛苦地重构代码。然后代码又变得易于管理了，每个人都笑逐颜开——之前的团队确实还太嫩。
4. 性能还不够强，所以这个团队又进行性能优化。现在性能又加强了——之前的团队确实没找对路子。
5. 现在，代码又难以维护了，又要跑回到第3个步骤。

还有一种情况——代码混乱不堪，甚至不能再重构。他们的产品周期变得越来越长，代码运行变得越来越慢，需要更多的内存及更快的CPU。这样的事常发生。

正确的做法是优化以得到期望的结果——易于维护及改进的高性能代码。不是度量运行的速度（这很简单），而是同时度量运行速度及代码复杂度。这样你就找到了二者之间最适当的均衡度。如果你确实够精到的话，你还会度量团队健康及用户满意度指标，在这四者间找到均衡。呵呵，这就像在做买卖（讨价还价）。

击鼓传花

让我们举个更贴切的例子——产品团队的结构。在传统的产品开发与新兴的敏捷拥趸者之间形成了一个战场。谁是对的呢？管它呢！但绝对不要单独地在某一步骤上进行优化——向着期望的结果进行优化。

期望的结果可以在最短的时期内带来大量的客户价值。记住，客户价值是不可用功能多少来度量的，是通过提供令人满意的高质量的点对点体验来度量的。

那么你如何快速地提供高质的价值呢？凭借约束理论（Theory of Constraints，TOC）。约束理

论认为一项工程以最快的速度达到目的的方法取决于其中最慢的那一个步骤，比如用户体验、内容发布及团队间共享的可以满足你的需求的操作方法。你的项目经理可能会在规范书中指定一个月内平均完成4项软件功能，开发团队能在一个月内完成2项功能，而测试团队一个月能检测3项功能。要达到项目经理要求的速度，条件还远远不够，对吗？

但是，产品经理们督促项目经理继续写些开发团队无法处理的规范书——优化局部而不是总体。为开发团队添置人力也无济于事（按：《人月神话》与《经济组织》）——同样，眼界太窄了。

正确的做法是，项目经理与测试团队要与开发团队齐头并进。在各个项目功能间留有足够的余地以保持其机动性，但绝不要让项目经理及测试团队的进度快于开发团队。这种约束理论的策略称为：鼓—缓冲—绳子（Drum-Buffer-Rope）。因为很难准确预测开发团队的进度，所以你要限制项目功能间的空间，在情况改变的时候，避免开发团队的队列中有太多的工作。

这就是为什么功能小组能行之有效。向着期望的结果进行优化——工作实例，在功能小组中——Office项目采取的方式——项目经理、开发人员以及测试人员在某一个实例中精诚合作，直到完成测试及集成。他们谁也不能跑到别人前头去。Scrum团队与敏捷编程团队要有相同的工作原则。不是简单的团队组合就完美了（虽然他们间的沟通很容易），是要一起步调一致，不断优化产品，为产品带来完善的、高质量的客户价值。

> **作者注：**有读者指出软件开发中的关键制约因素是通信带宽。增加通信带宽的最好办法是使团队同处一地，使他们基于功能点对点一起工作。这种方法同样对提升工作进度有效，就如我之前所说。一些Scrum团队也会采用看板模式，这是一种以更简单更直观的方式应用鼓—缓冲—绳子策略的方法。

不要怨天尤人

对眼前出现的问题往往很容易就会“一见倾心”并马上进行优化，而不是对你实际应该关心的其他问题进行优化。人们向来如此——我怀疑我们与生俱来就是这样的。这确实是优化错误的做法得出错误的结果这种做法的最好理由，但是这个借口太牵强。

你应该明白这个道理，如果你没有，你就没有权力兑现一张支票。想想你期望得到的结果，考虑周全，权衡各因素之间的轻重，在总体上进行优化。这并没那么难。我们每天都这么做，就像我们为我们的人生寻找一种均衡。关键是要深思熟虑而不是要滑头，要早做计划而不是怨天尤人。你做得到，只要单单地将你的目光盯住终点线。

2009年4月1日：“世界，尽在掌握”

就如当下的经济困难时期，人们不再发出社会不公这样的陈词滥调。很多时候，我感到轻松了许多，因为我再没听到说英格里德的收件箱中有好多邮件，说布赖恩又怎么怎么搞砸了工程创建，说苏雷什负责的服务开发日程拖了大家后腿。

然而，我们正处于困难期，我们还要为恼人的错误打补丁。你要什么时候再打起精神来修复大堆害人的程序漏洞？确实，还毫无头绪。

令人惊奇的是，好多一般性的问题用两种简单的方法就可以解决——单

件流和核对清单。有大量的行为与过程理论可以说明这些简单的办法很奏效。这里的重点是它们很有效，而你与你的团队没有它们的话就会很没效率。

都很简单

打开英格里德的邮箱。跟大多数邮箱一样，英格里德的邮箱都快满了。她花了大量的时间来处理，但是邮箱只会越来越满。她总是在邮箱中迷失，找不着想要看的邮件，或反复查看同一封邮件。很无助。

如果英格里德采用单件流，那就管用了。单件流会提示她一次处理一件（一次邮件消息），直到这个消息处理完成。“处理完成”指的是她再也不用看那条消息了（除非是为了回复与之相关的邮件）。

以下说明单件流是怎么奏效的。英格里德每次查看一条邮件消息的时候，她就采取以下4种步骤之一：

1. 她删除了邮件消息（我最乐意的）。
2. 她把它放进了一个文件夹。
3. 她把邮件转发给了另一个人，然后将其删掉或放到文件夹里。

作者注：你怎样保证能时时跟踪一个非常重要的转发邮件？我采取以下方法之一。

- 我把我转发的邮件放到一个特殊的文件夹里以示谨记。这封邮件一直呆在那里直到我收到回复或再给对方发一次提醒（之后这次提醒的信息也放到这个特殊的文件夹里）。在没有非常特别或重要期限要求的时候我都使用这种方法。
- 我把我的转发邮件移到我的日程表里，这样日程表就能提醒接收者按约定时间回复邮件或处理相关问题。在有非常特别或重要期限要求的情况下我采取这种方法。

4. 她回复了邮件，之后就删除了该邮件或将其存到文件夹里。

就是这样。她从来不用打开一个邮件消息然后还是把它放在收件箱里。每天结束后，以及之后的每一天，她的收件箱都是空的。她从不会错过、丢失或再看一次某封邮件。

单件流是很有效的，因为它节省了管理费用及避免了重复劳动。每次你查看一条新的邮件消息时要进行一次上下文切换，这样就产生一次管理费用，每次当你重新查看一条消息的时候，就是一次重复劳动。有了单件流，管理费用就能降低，而重复劳动就不会再发生。

作者注：是的，“一次过目”的准则也有例外，但是这种情况只占5%。有一天就在我身上发生过。即使那些邮件消息都放到了特殊的文件夹里以示提醒，我还是要用邮箱的过滤功能预先从讨论组中过滤邮件。如果你想知道怎么做，而不想把邮箱里的邮件全部删除，可以看下面的步骤。

1. 及时查看与你手头工作相关的邮件并把它移到特殊文件夹里。
2. 为剩下来的与你工作职责及兴趣相关的邮件建一个文件夹。
3. 在邮箱里检索与前两个文件夹匹配的邮件，再把它们放到相应文件夹里。
4. 浏览一下特殊文件夹，用单件流方法把90%的邮件清除。

现在，你走上正轨了。

似曾相识——一次又一次

布赖恩总是搞砸工程创建。你可以对布赖恩实施惩罚，直到他害怕了经常检查代码。但只是这样做的话会产生其他问题。

一个较好的办法是给布赖恩一个核对清单。人们普遍误解了核对清单，致使开发利用不当。遗憾的是，在布赖恩把代码集成到创建里之前，善意、顶真的人会把所有的东西都放到他要核对的单子里。这不仅浪费时间，而且没效率。

布赖恩的核对清单应该只保留引起创建失败的一般性问题（最好少于一页）。这样在布赖恩提交代码的时候一眼就能看明白。这样目标实现起来就会很快、很简单而且有效率。

太长或太复杂会让布赖恩搞混，太短也会没效率。幸运的是，大多数错误都是一般性的。因此，所有的团队应该做的是收集引起创建失败的一般性或至关重要的问题，再把它们放到核对清单里。这对设计复审清单、代码复审清单以及所有核对清单都是一样的。

记住，在你改变失败的设计模式时要更新你的核对清单，不然它们就走味了。核对清单防止了一般性错误的发生，而且有利于养成好的习惯。一些你所采用的结构化的、自定义的过程都应该有一份简单的核对单——除非你想象布赖恩一样搞砸工程创建。

作者注：你或许会在决定什么应该放到核对清单里存在思想斗争。你不应该这样。记住，你是要把一般性的或至关重要的问题放到清单里，而不是所有一切发生过的问题。

举个例子，几个月前我将核对清单附到了我的 E-mail 签单里，我发出了每封邮件。核对清单里我只列出了几年来我一直在犯的两个错误，但现在一直没有犯了。

- 检查一下抄送与隐藏抄送名单是否准确。
- 检查一下标题是否正确。

动起来

苏雷什的软件小组在他们的日程表中落下了。对于服务项目来说这是个坏消息——或者可以说对所有的项目工程。需要周末加班吗？不。对这不中用的团队要惩戒一下吗？不。要单件流及核对清单吗？是的。

苏雷什小组成员的构成没什么特别——几个项目经理，包括一个服务开发工程师、一群开发人员及一班数量相当的测试人员。项目经理编写规范书，开发人员编写代码，测试人员进行测试。这个小组采用类似的 Scrum 冲刺法，按优先级别安排功能开发备忘录。

遗憾的是，项目经理建立规范书的速度远快于其所能被开发的速度。在黎明到来之前他们为修改或删节规范书浪费了大量时间与精力。当开发人员与测试人员遇到障碍的时候，他们从一项功能转换到另一项功能。没一次是同步进行的。也没完成任何功能。工作日程不断延后。痛定思痛，所有苏雷什的小组成员只是考虑了相关人员都能顺利完工的情况。

尽管，苏雷什的团队采用的是类似 Scrum 的冲刺法，但他们没使用单件流。他们没有将小组分成一个个跨工种的小团队，而是将所有小团队集中在一起一次只进行一项功能开发，直到这项功能完成。他们甚至没有使用核对清单注明已完成的工作。悲剧了。

只要他们分为不同的团队，一次只对一项功能编写规范书，写代码，或测试（有时称为功能小组），并列出核对清单，标明什么已经完成了。这样，工作进展就有戏了。单件流解决了阻碍工作进展的问题，因为每个人都在处理一个相同的问题。核对清单使团队变得更可靠，并激励着他们协同作战，一起完成工作并保持专注，而不是出错了从头再来。现在，这个团队不再为上下文切换而浪费时间，并且可以明确，完成一项功能到底要多长时间，赢得可信的工作日程表以及高质量的产品及服务。

作者注：很多人想知道当功能小组的项目经理完成规范书后，或是开发人员完成代码编写后他们该干什么。对于项目经理，有两种选择——成为多个功能小组的一部分或参与到开发人员或测试人员中解决问题及进行程序开发。对于开发人员来说，与测试人员协同进行工具测试、自动化测试、单元测试及组件测试。

还有一个更明智的选择，是由微软的一个前雇员——科里·拉扎斯发明的。他称为Scrumban，你可以在由他与伯尼·汤普逊创建的“精益软件工程”网站上找到：（leansoftwareengineering. com/ksse/scrum-ban/）。

两手武器

现在，你手握利器——单件流及核对清单。有了这两个功能强大、意义非凡又简单的方法（可惜并未充分利用），就能进行有效的工作进程管理，提供高质量的软件。

单件流与核对清单可能应用于个人、小团队及大型部门。这没什么可争议的，只要用得有的放矢。经过几年的实施案例下来，历史会给其功效以明证。

是的，你可以在草根基层间掀起一次单件流与核对清单的应用狂潮，但是单纯这样想似乎有点过了。或许，你应该只是择机而用。享受你过往的时光，享受你的成果不断提升的喜悦吧，人生最惬意的事莫过于简约与本真。

作者注：詹姆斯·维斯基有一篇有趣的博客提供了一些核对清单作参考，文章名为：“核对单？我们不需要擦屁股用的核对清单！”（http：//go. microsoft. com/FWLink/？Linkid＝219829）。

2011年4月1日：“你必须做个决定”

哪种情况更糟糕——一个不成熟的决定还是没有决定？很简单。工作因有决定而开展。一个不够完善的决定可能会使你的工作不知不觉中走向歧途，但是至少它还是有所进展的。稍作修整，你就会走上正轨。

不做决定会使工作搁置。即使你在延误期间做了个明智的决定，重整旗鼓所花的时间将使你远远落后。就像滚石与火车，由人组成的团队也具有惯性。这需要很大的力气才能使它们动起来，所以最好还是让它们保持前行并进行调节，而不是暂停再重新开始。

遗憾的是，领头人常常没有足够的信息做个确切的决定。当你的领头人在处理事情时优柔寡断，你会称呼他们什么呢？庸俗之辈，无能的冒牌货。他们是失败者，不是领导者。如果你想做

个领导者，你最好知道如何在数据不完整的情况下做个明智的决定。

我是决定者

讨论在数据不完整的情况下做个决定之前，我必须指出做一个莽断者不如做一个无能的冒牌货好。莽断者做决定的速度很快（这很好），但没有清晰的思路或原由（很不好）。这样的结果是随机或倒退的过程，以及一个怨气丛生的队伍。信息太少不是独断专行或愚蠢的理由。

相反，你应该尽可能地时刻提醒自己：要在相关情况及经验的基础上做出决定。你有多少时间？只能是在你的团队遇上困难之前的一段时间。你怎样利用相关情况及经验形成一个统一稳妥的基础架构再行决定？这种基础架构有助于你把注意力放在整体设计模式上，然后依你聪灵的直觉作个最好的选择。

这很容易举例来描述。我们来谈谈招聘决定、产品定位以及会诊决定。

> **作者注：**如果你想要有更多的时间来做决定，那么得预先做好计划。将你的工作日程制成日程表，对相关事态作个预测后，及早动手作出决定；如果你希望有更充分的数据再作决定，选择好的度量方法，招募业务智囊团，并且也要充分了解相关情况。以此两种策略来做决定，就要未雨绸缪。不喜欢吸尘器吗？填满它。

雇用还是不雇用

我作为“胜任者”已经有 10 来年了，这意味着，在一次招聘中，我是剩下的最后一个面试官。我的角色作用是对应聘者作最后的聘用决定。有时候这样的决定是非常容易的，因为之前的面试评语已经在某方面得出一致意向，而这些评语也很符合我个人对这个应聘者的评价。

然而，做聘用决定常常是很艰难的。上一次的面试回馈是混乱且难以使人信服的。一次面试，我最多只有 1 个小时的时间直接从应聘者身上获取信息。再者，招聘团队及人事经理在那等着我做最后的决定，所以我只有 1 或 2 个小时来做出决定。

为做出可能的最适当的决定，在每次面试中我只遵循一种惯例。在面试之前，我事先看了下应聘者的简历，检查一下简历中提及的在线资料，再看一下之前面试环节的回馈信息。我要在评语中找寻一种既定模式——指出我最关心的根本性问题。带着检验这个根本性问题的目的会见应聘者，并决定之前所关心的问题是否可以舍弃。

现在，我已尽所能地收集所有事实依据，不会让这次招聘时间拖得太长。我已经作了充分的分析，我也有了对应聘者的初次印象。我怎么才能对全部这些作个综合判断？回到最初的时点，让我的直觉来为我指引，作了所有的分析，并凭借我多年的经验，哪个决定会是正确的？

就如马尔科姆·格来德威尔在 Blink 网上指出的（www. gladwell. com/blink/index. html），我不是说一眼就能对应聘者做出个判断，我说的是尽我所能更多地了解事实，根据我对这位应聘者的全部了解，回头再对这个应聘者作个整体全方位的审视，并将他与我面试过的其他应聘者作比较，而不是只把目光放在个人的细节问题上。确实很难做到这么理性——你的直觉可以成为你的指引。

作者注：我即使花再多的时间来谋求别人的参考意见，对应聘者进行更多的面试，然后在网上查找他的背景信息，我的决定也不见得会变得有多好。格来德威尔还指出，有时候再次分析与判断只会更差，因为你开始忽略你对应聘者的整体评价以及逐渐积累起来的对这个人的好感。

如果我还是无法对这个应聘者作个论断，你的决定很简单——不要雇用这个人。你总是希望对这个应聘者的成功潜力拥有足够的信心，无论出于应聘者本身的原因，还是团队或公司的原因。

现在设想一下

二选一的产品定位决定是很难达成的。通常都有很多不同的意见，却很少有数据是起决定作用的。遥感数据与 Web 服务数据或许不同，但是这些数据会告诉你——它做不了决定（需要，但并不充足）。你必须依此数据及你积累的经验尽快做决定，不然就有失去你的市场地位的风险。

个人的决定可能是关于用户需求描述、架构设计或甚至是一个开发工具的。不过，每种选择都是有其优劣的。首先，依据你的价值判断、架构原则或其他相关基础架构（framework）明确选项。如果你还有其他更多的选择，你可以用我最喜爱的方法进行分析，快速地做出取舍。这个方法叫 Pugh Concept Selection，名字很有趣，技术却很简单。（我在第6章的“我的试验成功了！（原型设计）”中，对它做了简单的描述。）

在 Pugh Concept Selection 中，你要创建一张表格，表行代表你的选择，表列代表你所做决定的标准（Excel 表很适合做这个）。其结果就是一份优劣评分交叉表。挑其中的一个选择作为默认标准，赋以数值0，然后将其他的选择与这个默认选择作比较，而不是给其中一个最好的选择标准赋予绝对数值。如果一个选择好于默认的标准，就在相应行列中赋值 +1；如果这个选择更差，赋值 -1；如果这个选择与默认不相上下，赋值0。然后对所有列加总，你就得出每行选择的评分。那现在你就知道所有选择的总体优劣了。

作者注：你可以对标准（列）赋予不同权重以区分哪列标准是更重要的。通常情况下，使用权重值（1，3，9）。将相应权重与每个标准得分（-1，0，+1）相乘后，再加总得出每行的总分。Excel 的 SUMPRODUCT 函数在这里就很适用。你可以在在线资料中找到这种电子表格的示例（Pugh Concept Selection Example. xlsx）

将这张表填满再进行分组是很有趣的，而且会有一个非常不一般的结果。你现在知道这么多选择中真正的优劣处在哪，以及你最为在意的标准是哪些。然而，Pugh Concept Selection 不能为你做最后的决定（虽然它淘汰了一批差劲的选择）。但是总有一种选择会得最高分，以及很多得分靠前的选择值得你慎重考虑。

当选择表完成后，你已经对这些结果及其提示给予充分考虑，现在是做决定的时候了，也就是你该把这张表放在一边并凭借你新生的直觉做决定的时候了。这种分析是必需的也是很具建设性的，但只有你的经验以及对所有情况的把握（你的直觉）才能指引你做出正确的决定。

> **作者注：**“但是，你的决定就不会有偏见吗？”你会打赌这是肯定会有偏见的，就像你平常所作的所有决定一样。要做到细致周到，并对不同的合理性选择多加考虑，就能使你尽可能地消除这种偏见。然而，你最后的决定多少还是会反映出你自己个人的独特看法。

告诉我为什么

当你在管理维护着产品的相关问题，直到产品周期结束时，就会有非常多的问题需要你解决，所以必须快速地做出决定。另外，你经常会在一个由三个有着不同兴趣的人组成的小组里做出决定。你们必须达成共识，而无一人反对。

> **作者注：**在微软，这种决定产生过程称为会诊。如果处理不当，这会是开发周期中最易产生分歧的地方。

你怎么能快速地说服三个或更多的人满怀信心地同意你对某一个重要问题的见解？你需要明白三件事情：

- **问题的根本原因。**如果不知道问题的根本原因在哪，你就不能很自信地凭以往的经验来解决问题。
- **问题的严重程度。**如果不知道有多少客户与合作伙伴将受此影响以及对他们的影响程度，你就不能对你的行为带来的效益作出判断。
- **问题的风险。**如果不知道其内在的风险，你就不能在成本与收益间作有效平衡。

你懂得了这三个道理——后继相关问题的报告中都将包括这三件事情，那你的直觉就周全到位了。现在整个团队的决策者就可以依据内心真实的感觉对提议的解决方案作出选择。当然，不能保证说一定会达成一致意见。想对此类富有争端的团队决策有更多了解，看第 1 章的专栏“我们还开心吗？分诊的乐趣”。

等会儿，还有更多

“如果以后有更多的信息会对你的想法产生重大改变，那该怎么办？”没问题。我们只是想做个决定让我们的工作有所进展就可以，不需要完美。能让我们增长见识与经验的办法就是尝试，接受回馈，判断，然后再次尝试。

相反，没理由经常性地改变决定。保持你不断进取的心，你必须不停地反复地在其过程中一点点完善。但是，偶尔，也会有很多新的内部见解与外部的期望。好了——就是这些，有了这些我们的生活才更精彩，我们的工作才更具魅力。

> **作者注：**关于管理者经常性地改变主意会带来什么问题，你可以在第 9 章的“管理馊了”了解更多。

每次对新的信息进行一次分析，你的直觉就逾加灵敏，你就越有经验，眼界也越开阔，你的决定也就越完善。保持开放思维，多倾听客户及反对者的声音，形成一个统一稳妥的架构，相信知进善学的天性，不要坐以待毙。做个决定，再改进，带领你的团队不断向你们的目标前进。

第 4 章

跨越工种

本章内容：

2002 年 4 月 1 日：“现代临时夫妇？开发与测试”

2004 年 7 月 1 日：“感觉性急——测试者的角色”

2005 年 5 月 1 日：“模糊逻辑——君子之道”

2005 年 11 月 1 日：“废除工种——有什么理由搞专业化？”

2009 年 1 月 1 日：“持续工程的鬼话”

2011 年 5 月 1 日：“测试不该不受尊重”

软件开发如果要做得好的话，需要掌握很多技能。你必须真实地理解客户和业务。你需要有很强的用户体验设计技能和可用性知识（即使你是在做应用程序编程接口）。你需要有工程设计技能、软件开发技能、软件测试技能，并且非常熟悉目标平台（它可能是一个企业数据中心）。

当然，你可能具备以上所有这些技能，但是你是否经常自以为是？如果你没有这种与人沟通的技能，你很可能会遗漏产品开发的一些重要方面。如果你经常自以为是，那可能会引发甚至更为严重的问题。总之，你最好感恩一点，掌握与别人和谐工作的诀窍。

在这一章中，我主要讨论开发者和其他工种人员的关系。第一个栏目描述了与测试者的共生关系。第二个栏目对于测试者的角色进行了更为深入的解析。第三个栏目直击大部分工程师的“致命弱点”：如何跟非技术人员交往。第四个栏目讨论了为什么以及何时设立不同工种是必需的。第五个栏目讨论在持续工程中存在的各种争端。讨论了高级测试人员与软件开发以及项目管理是相通相融的。

我花了前半生的时间去搞明白为什么人不能更像计算机，我的后半生一直为那不是事实而感到庆幸。当然，当有人不认可我的做事方式的时候，我还是会感到失落。但我的生活充满趣味，我提供的解决方案更有创意，它是我对周遭各种各样的人的一种回报。这是我热爱“代码之殇”栏目的部分原因，读者的回应永远不乏味，而且总是有出人意料之处。

2002 年 4 月 1 日：“现代临时夫妇？开发与测试”

有比开发者和测试者之间的爱恨关系更经典的关系吗？（好吧，也许开发者和项目经理之间的关系有得一比，但我不谈这个主题。）作为开发者，你怎么看待测试者？唠唠叨叨或坚持不懈，挑剔或细心，令人讨厌或充满热情，不懂世故或关注客户，预言灾难或谨慎小心。

那么，测试者是如何看待开发者的呢？同样可以说上一段：才华横溢或智力超人，尽心竭力或工作努力，有创造力或善于发明——很好，太对了！但你想听实话吗？其实，测试者对开发者常常像我们对他们一样评价低。

很多团队在这些工种之间远没有达到平等和相互尊重。一些吃着类固醇的开发者可能会作出反应：“没关系。领导、跟随或者离开。如果测试团队不能跟上步伐，不能并肩作战，不能收紧他们的肌肉，那么他们应该走开，远离轰轰而来的开发机器。”

一听就知道这是缺少脑筋的人说的大话。该揭穿这个颠倒是非的谣言了，一片一片地，直到揭示真正的开发者应该怎样跟他们最杰出的盟友——测试者——一起工作。

> **译者注：**类固醇（Steroid）是广泛分布于生物界的一大类环戊稠全氢化菲衍生物的总称，它包括胆固醇、胆汁酸、胆汁醇、类固醇激素（如肾上腺皮质激素、雄激素、雌激素）等。此外还有人工合成的类固醇药物。这类药物具有增加肌肉块头和力量的作用，另外还具有雄性激素的作用。

我怎么爱你？让我来数一下有多少种方式

为什么开发团队经常认为他们的测试团队是二流的，不值得尊重和合作？我把其中的原因分解开来，一个一个讲给你听：

- **所有测试者都希望他们能成为开发者，因此开发者必然出色一些。**这样一概而论是不对的，也是荒谬的。不是所有测试者都希望能够成为开发者。当然，有些人确实有这种愿望，但其他人想成为高尔夫高手、赛车选手、产品单元经理、项目经理、父母、牧师——你肯定还能说出更多的。甚至，有些人就想成为伟大的测试者。

  这个观点反映了一些开发者有多么不自信。一个真正以他的工作和成就为荣的开发者，并不需要觉得开发是高人一等的工种。

- **测试者没有开发者聪明。**这是前面那个谬论的推论，常常还有人这样说：“如果测试者跟开发者一样聪明，他们就会转为开发者了。”再说一遍，这种说法基本上就是牛粪。

  然而，即使是公正的人也会指出，测试者通常不需要拥有像开发者那样同等的教育水平。尤其是黑盒测试者，他们受雇的时候常常很少有甚至没有技术经验。对我来说，这完全是一种“鸡—蛋之争”（先有鸡还是先有蛋）——为测试者准备的外部培训实在是太少了。

> **作者注：**黑盒测试（Black-box Testing）把产品当做一个不透明的盒子。对于产品的内部工作，测试者无权介入，也一概不知，因此他们只能像客户一样去探测软件——使用甚至滥用，直到它表现异常。微软已经稳步转向了白盒测试（White-box Testing），从而测试者可以使用暴露的源代码插桩，对产品的方方面面进行自动化和系统化的测试。

公司内的很多人都通过类似微软就绪计划（Readiness At Microsoft Program，RAMP）和测试主管计划（Test Lead Program），努力纠正这种不公平。甚至一些综合性大学和当地的社区学院已经开始设置测试方面的课程。但只有等到学院和大学为这个重要的学科建立起标准化的学士和硕士学位计划之后，测试者才能找到通过高等教育学习更多知识的良好机会。

是不是这种教育机会的缺失意味着大部分测试者都不聪明呢？当然不是。测试者有强弱之分，就像开发者也有强弱之分。道理是一样的。

- **如果没有开发者，测试者就不会有太多事情可做。**没有开发者，测试者就没东西可测的说法，导致有些人认为开发者要出色一些。（当然，开发者经常以 Bug 的形式给测试者制造大量工作的事实，也让很多人认为开发者要差劲一些。）虽然如此，是不是所有的上游工种都应该被认为要强于、依赖于下游配对工种呢？的确，执导一部电影要比开动一架放映机困难，但挖掘出一块钻石却要比把它切割和磨光来得容易。

  这要取决于工作的复杂程度，而且把产品和功能的测试做好毫无疑问跟一开始开发这些东西一样难。编码方面很多困难的理论问题已经得到了解决，但在测试方面，很多困难的理论问题仍然没有找到答案。

- **与强大的开发团队相比，微软没有很多强大的测试团队。**基于这个假设，开发者可能推断出，他们的工种总体上比测试者一定要好。这是一个糟糕的假设，也是很不负责的借口以推托在整个过程中你没有尽力去帮助测试团队成长，从而使整个部门变得强大的责任。测试不像开发是一个成熟的工种。如果一个开发者以此为由，带着很强的优越感与测试者一起工作，其结果是不仅不会提高测试者的成熟度，而且还会使开发者自身的能力降低。

有苦难言还是良药苦口

我们不能发布一个没有经过测试的产品。回想一下测试者在项目的最后时刻在你的代码中发现的 3 个最严重的 Bug，然后你就知道我在说什么了。很多开发者还没有意识到在测试团队的支持下你能多好地发布一个产品。

令人遗憾的是，很多部门还把测试看成是一种难言的痛苦，而不是构成整体开发团队所必需的一个珍贵伙伴。我知道有一些开发者会斥责我说："必需的珍贵伙伴，但愿！测试永远都是这样，像猴子一样猛敲我们的代码，直到完全把它毁掉；然后对代码牢骚不断，直到我们把它修复。"

如果那样说他们，那么你的测试者真的就会是那个样子。不过，还有其他方法可供选择。

每个人都要知道自己的弱点

先从开发者存在不足的 3 个基本领域开始：写出完美的代码，完整地理解 RAID，在客户环境中工作。

没有哪个开发者能够写出没有任何 Bug 的代码；即使没有逻辑上的 Bug，那可能还有特性上的 Bug 呢。没有哪个开发者与 RAID 是惺惺相惜的；他们只是把它当做工具来使用。没有哪个开发者通常在客户环境中工作；他们工作在有大量内存、高处理能力和大磁盘空间的高配置机器上。开发者还拥有很高的特权、快速的网络、最新的操作系统和补丁，并且没有接手他人的代码——

他们都工作在自己的母语环境中。基本上来说，开发者对于把一个产品从他们手上转交到客户那里所需要做的事情漠不关心。

那么猜一猜，当开发者穿越编码完成与产品发布间的钢索时，是谁把持着安全网，又是谁为他们提供平衡棒？对的，就是你的测试者。

测试者同时查找逻辑上的和特性上的 Bug；他们与 RAID 朝夕相处、惺惺相惜；他们知道开发到了什么程度，还剩下多少事情要做；他们的电脑总是在客户环境上运行，并且有各种不同的语言、平台和配置。无需乱猜，测试者可以清楚地告诉你每个星期需要修复多少个 Bug。他们知道产品的问题区域在哪里，并且可以给你提供内部数据和度量让你知道你离目标还有多远。

你是面镜子

开发者可以让你惊喜地感叹说，“哇!”但测试者会在背后默默保护你，让你对开发者的工作感到骄傲。

如果你把你的测试团队看成是垃圾，那很可能你发布的产品也会是垃圾，最终你可能会觉得自己也是垃圾。如果你更愿意让发布顺利一点，想要发布一个出色的产品，那么一定要把你的测试团队当做盟友。

这里的关键是，你和你的开发团队要理解和感激你的测试团队为你们所做的一切。他们做了你们不能或者不想去做的事情。他们保证你的工作按计划进行，并且处在正确的轨道上。他们让你成为客户愿望的忠实代表。

告诉测试者他们对你有多重要。告诉他们多亏他们才使你们一起推出一个出色的产品。尽你的一切所能去支持他们，他们也会让你安然无恙。

> **作者注：**语言已经无法充分表达我对测试这个工种的尊敬和感激了。我写了这个栏目之后的 5 年间，开发和测试在各个领域的平等性已经得到了很大的改善，但我们还有更多的事情要做。至于管理者和项目经理，我其实一样敬佩和尊重他们，但因为我一直生活在他们一个个奇思怪想的阴影之下，原谅我对他们的自负没有手下留情。

顺便说一下，如果你早已把测试团队当成尊贵的伙伴，那么也该同等地看待服务实施人员。开发者和他们之间存在着同样的问题，因此为什么不把你们之间的痛苦和失落变成协作和成功呢？

2004 年 7 月 1 日：“感觉性急——测试者的角色”

最近，我对代码的构建质量抱怨得比较多。在早期阶段消除 Bug 的方法主要有 5 种，单元测试（Unit Testing）是其中之一，其他的还有：设计（Design）、设计复审（Design Review）、代码复审（Code Review）、代码分析（Code Analysis）。（代码分析常常用 PREfast 这个工具。）开发人员要做全面的单元测试，但这给一些人带来了怀疑和困惑，“难道这不是测试者的工作吗？”“如果开发人员写了单元测试，那测试者还剩下什么可做的啊？”

> **作者注：** PREfast 是一个 C/C++ 编程语言的静态分析工具，它能识别出可能导致缓冲区溢出或其他严重编程错误的可疑代码模式。尽管这个工具最初只是内部使用，但它最近已经集成到了微软的 Visual Studio 2005 中。

首先要说的是，单元测试通常关注独立的组件，而测试者关注的事情要比组件测试多得多。他们的测试包括边界覆盖、结构化覆盖、路径覆盖、黑白帽测试，还有许多系统测试和基于场景的测试。总之，他们的任务就是要撕破软件表面的假象。

不对，首先要说的是，如果测试者没事情可做了，你到底在担心什么？从什么时候开始，你已经放弃了检查你自己工作的责任？莫非传递蹩脚代码对你来说是某种形式的报复性施舍？你为什么不直接扔一把飞叉呢？你已经没有了自尊或骄傲吗？

> **作者注：** 我当初在撰写这个栏目的时候，在准备写“其次”段落时觉得难以下笔，于是我回过头去重读了一遍“首先要说的是”那一段。突然，“不对，首先要说的是”这一段内容像潮水一样涌出我的脑海。这往往是好事！

单元测试可以带来更好的实现设计，更可测的代码，更少的回退、创建失败和“创建验证测试”（Build Verification Test，BVT）失败，并且能够得到更好的全局构建质量。

高级保护

在我的一篇叫“牛肉在哪里?”的文章中（参见第5章），我曾指出过开发人员在编写源代码插桩和单元测试方面的责任，如下：

“不对，这不是测试者的工作。测试的职责是保护客户免受任何你遗漏的东西所带来的伤害——而不是盯着你做得最好的东西。测试团队不是你的拐杖，你不能拿它去支撑满是污秽（代码）的双脚的平衡。如果测试发现了一个 Bug，你应该感到局促不安。他们发现的任何 Bug 都是你遗漏的 Bug，而且最好不是因为你的懒惰、冷漠或不周全引起的。”

不过，有些开发人员事实上已经对他们的同伴给予足够的重视，考虑到了单元测试对测试工作的影响，为了公平起见，我应该更多地讨论测试者在保护客户方面的角色，以及有朝一日开发人员都能展开合作之后，测试可能会有怎样的进化。

改变一下对你有好处

在“现代临时夫妇？开发与测试”一文中（参见本章的第一个栏目），我曾指出了测试者保护我们的客户的3个主要方式：

- 查找我们遗漏的 Bug。
- 倚重质量度量（主要利用 Product Studio）。
- 在客户环境而不是在开发环境中运行测试。

即使开发人员做了单元测试，也并没有改变测试者的上述这些角色。

虽然如此，测试还是在改变。几乎所有新招的测试人员都要求知道如何编写自动化代码以及

白盒测试。我们中的有些人可能会冷嘲热讽觉得这是个阴谋，因为所有的测试自动化之后就可以不要测试人员了。然而，加强自动化的真正原因还在于，为了达到验证目的，需要运行所有必要的测试，其总共花费的时间可以大大地减少。这些验证可能针对某个紧急的安全、隐私或可靠性补丁问题，或者甚至只是针对某次创建。如今，很多团队都在力保上乘的质量，而自动化测试对于保证源代码签入之前的构建质量，是一种非常基本的手段。

这给我们带来了下一个重要的转机。随着代码在签入时的质量的提高，测试者如何来对他们的角色进行调整呢？注意，他们的角色没有改变——查找遗漏的 Bug，分析质量度量，并且在客户环境中工作。不同的只是他们如何去做。

黎明时分

现如今，在已经签入的代码中查找 Bug，跟在西雅图寻找一家咖啡店差不多——想要错过它们都很难。（译者注：这说明 Bug 很多，很容易就能找出来。）当然，你要编写测试计划和测试用例来穷举所有不同的可能性。不过，你总会有这样的期望："只需要运行一下程序，Bug 自动就出现了。"

随着团队尽力争取上乘的质量，在代码签入之后查找 Bug 变得更加困难。（译者注：这说明存在的 Bug 已经很少了，很难再找出新的 Bug。）起初，一些团队把测试"密友交付"作为他们代码签入标准的一部分。现在很多团队都采纳了这种方法，包括一些很成熟的部门，比如 Windows 和 Office。

> **作者注：**密友交付（Buddy Drop）是产品的一个私有创建，用以在源代码签入主代码库之前验证代码改动。通过这种方式，不稳定代码对于其他团队的影响降到了最小。开发者只跟他们的"密友"（也就是他们团队中的同事）共享这个私有创建。

但是当开发团队开始使用专业的实践和度量来预测和控制他们的 Bug 数量时，在一个密友交付中发现的 Bug 数量应该骤降至以前的千分之一。从而，一个典型的有 15～20 人的微软开发团队，假设他们以前每年产生 3 000～5 000 个 Bug，那么现在每年只会产生 3～5 个 Bug。当密友交付达到如此强健时，测试就需要学习一些新的技巧了。（我在第 5 章的"软件发展之路"栏目中，将会谈到一些团队是如何通过应用很酷的软件工程原理去达到低 Bug 率目标的。）

充分利用数据

记住，测试的角色是，通过查找开发者遗漏的 Bug、分析质量度量和在客户环境中运行测试来保护客户。如果有一天代码移交给测试的时候只有少数的几个 Bug，接下去的问题是如何把测试的 3 个责任提高到一个新的水平。

在如此低 Bug 率的情况下，测试者必定不再能够在独立的组件上或普通的配置下很容易就发现 Bug。他们必须关注完整的客户系统场景和现实的客户配置，以使他们独有的客户视角起到进一步的杠杆作用。尽管如今的测试团队已经在这方面做了一些工作，但这很可能在将来成为他们的工作重心。

另一个重要改变在于分析质量度量。开发人员为了达到低 Bug 率，他们必须从设计之初到编译和建造一直收集质量数据，然后使用这些数据去了解，他们正在犯什么错误；如何更好地发现错误；以及到什么时候他们可以很有信心地宣布，他们已经找出了其中 99.9% 的错误。在开发之外的某个人必须去检查开发人员的假设，确保数据是可靠而精确的，保证开发人员的诚实以及客户的利益得到保护。显然，做这事的理想人选就是测试者。

这个严峻的开发过程的一个副效应是，测试者将拥有比他们曾经在 Product Studio 中获得的多得多的质量数据，并且有多得多的细节。你拥有的数据越多，你就有越多的东西去分析和发现。你几乎可以听到测试者正在流口水的声音了！

> **作者注：**我所认识的大部分优秀的测试者都对数据很狂热。一个完美的下午对他们来说，就是能把眼球盯在 Excel 表单上用各种不同的方式去分析数据。

尽管开发人员将会做一些分析，但对于测试者来说，那只是些小儿科的东西。他们还要继续做的事情有：

- 从事深入的统计研究。
- 发现关键的薄弱环节和趋势。
- 找到新的方法，进一步提高质量或功效。

在这个全新的世界里，测试者可以成为质量过程和分析之王。

非常酷——我向你保证

当开发者升级为工程师，测试者就可以开始利用全面的数据和过程分析来保证质量了。换句话说，测试变成了“质量保证”（QA，Quality Assurance）。“质量控制”（QC，Quality Control）的测试并没有消失，只是质量保证成熟起来扮演了主要角色。这样的模式已经在其他行业中发生了，软件行业也是迟早的事。如果我们不去推进它，我们就会因为落后而挨打。

当然，大部分的东西现在看起来像是一连串的梦。即使有些团队正在开始了解这个层次的质量工程，但让它成为我们最大系统（比如 Windows）的一部分还有很长的路要走。不过，如果你的团队正在开始取得进展，或者如果你认识一些有远见的测试者，他们准备主导下一波软件质量，建议他们开始学习深入的数据统计和过程分析。那能帮助他们把我们的“游戏”带入到一个全新的水平。

> **作者注：**虽然对于一个大型复杂的项目来说，我们还没做到将 5 000 个 Bug 减少为 5 个，但现在各个部门很看重对质量信誉及数据分析的要求。一个具体的例子就是 Bing，Bing 每天都在更新（我可不是指它的背景图）。如果测试组为了做一件毫无瑕疵的质量控制任务，那测试的度量工作就太令人惊悚了。相反，Bing 的测试只在关键领域进行质量控制，并着重于后续的质量保证，这包括大量的全面数据分析，这也是 Bing 项目工程所追寻的非常重要的一项工作。

2005 年 5 月 1 日：“模糊逻辑——君子之道”

我一生都跟固执的无知者生活在一起——他们可能认为“敲敲木头”是一种愚蠢的迷信，然而对电视是怎么工作的，飞机怎么飞行，或者电话是怎么建立连接的都一无所知。对他们来说，那些都是不可思议的。他们常以神奇的方式来使科技发挥功能。然后这些科技新手会向你诉苦：在你重新启动电脑之前，一定要把房间的灯关掉，否则它就不会工作。

译者注：“敲敲木头”（Knock on wood）是一个典型的英语惯用语，指接触木制的东西可以确保好运、甩掉坏运。这个短语的起源和宗教信仰或迷信有些关联。它很可能起源于古代人的一种信念，即碰一碰树木可以驱邪或找到保护神，当时人们认为保护神是住在树里面的。

在我大学毕业之前，我能很熟练地应付这些天真的白痴。只需避开他们，对他们不予理睬。我喜欢跟技师呆在一起，这让我感到很愉悦。这种情况一直持续到我跟一个“随性者”结了婚——一个受过自由教育的人。突然之间，学着跟技术迷信者沟通成了我自觉的兴趣。到现在，我们已经在一起生活了 20 年，我也逐渐掌握了其中的窍门。

作者注：哦，这个栏目带来了些许伤痛（伴随着许多的称赞）。伤痛来自于一些私底下很随性的技师，和一些热爱技术的随性者。这两种人都讨厌我描绘的人物形象。（既不属于随性者也不属于技师的人，比如美工设计师，他们也很伤心，因为被排除在了本栏目的讨论范围之外。）正如我下面所说的，“很显然，我这里以偏概全了。”对于一些含糊的、有争议的问题，人们通常太斯文或犹豫不决。因此我在每个栏目都做了些过度概括，以突出要点，引出更多的对话。

译者注：在美国，学校可以分成两个大类：一种叫职业学校（Professional College），另一种叫文理学院（Liberal Arts College）。前者的目标是培养有专业技能的人，而后者的目标是造就全面发展的人。文理学院相对于具体的职业教育来说，更加崇尚自由教育或者素质教育，其目标不在于教会学生某些具体的谋生技能，而是从多方面对学生进行教育，使其成为一个高素质、有教养的文化人。

包罗万象

为什么你应该去关心与无知者的交往呢？为什么不让这些白痴继续愚蠢下去呢？我这里有 3 个理由：他们可是我们的“同事”、“管理者”和“客户”啊！

大家知道，我们招聘人才时的参考标准是他们在技术上的悟性，但我们都知道有些人悟性很高，有些人则不是。每有一个高悟性的项目经理或管理者，就有另 5 ~ 10 个悟性还不够的。微软有许许多多的项目经理和管理者。你在产品或实践方面想取得的任何改进都要得到他们的认可。

如果你不能有效地跟项目经理和管理者沟通，恐怕你在他们面前的“低级滑稽表演”，足以给你带来一生的挫折感。

> **作者注：**很多项目经理和管理者都是从技师阶层晋升而来的。他们曾经理解所有的细节，并且认为任何东西都不是理所当然的。然而，随着时间的流逝，他们意识到，忽略细节，多考虑非技术方面，多着眼于全局对他们的工作更有利。于是，他们的世界变得随性起来。是的，我就有这种亲身经历。

至于客户，我们没得选择。客户就是客户，而不能有效地跟客户沟通会成为你职业生涯的短板。正如我在第2章的“客户不满”栏目中所说的那样，跟客户交谈是保证产品关键决策正确的关键。客户不喜欢我们高高在上，那样只会反衬出他们的渺小或愚蠢。我猜想，管理者和项目经理也不喜欢这样。

你必须要理解随性的人。你必须以一种他们认可的方式争取得到他们的最佳评判，让他们感觉到巧妙，并且让他们觉得一切尽在掌控之中。否则你不会如意！

他们跟我们不一样

受过自由教育的人跟我们不一样。不只是他们就读的学校或选修的课程不一样，更重要的是他们会用一种完全不同的方式去看待世界，而这个世界你需要真诚地去“神入”理解（他们可能并不熟悉“神入”这个词）。

有幸的是，我过去20年都在想方设法理解模糊逻辑——随性者的处世哲学。我发现了他们跟我们之间的一些关键差异，你必须要记住：

- **受过自由教育的人相信规则就是规则。**技师相信规则不是用来盲目遵从的，它们应该被分析和理解，然后才是使用或做必要的修改。随性者相信规则是用来保护你的，必须严格服从它们。更糟的是，随性者对规则的理解可能跟你的理解根本就不一样。请记住，如果你打算在一个随性者面前质疑或破坏任何感知方面的规则，你最好准备解释为什么可以这么做，并且找一个权威人士支持你。为什么？因为……
- **受过自由教育的人尊重权威。**技师通常不尊重权威，他们尊重的是成就。因此管理者的赞成对你来说可能没什么大不了的，但对于一个随性者来说却很重要。好消息是，大部分随性者都把技师看成是技术方面的权威，因此你在专业领域说的任何事情，他们大部分都相信。千万不要滥用这个有利条件。随性者可能跟你不一样，但这并不等于说他们都是傻子或笨蛋。

> **作者注：**看到这里，你可能会说：“等等，随性者多年来一直在破坏规则和亵渎权威。”其实，他们只有在觉得正当的情况下才那么做。在日常生活中，随性者更倾向于服从无常。技师则更喜欢依据他们自己的理由和逻辑去判断规则和权威。

- **受过自由教育的人不折腾。**技师酷爱折腾，随性者避免折腾。折腾将会破坏规则，让人感觉有风险、不安稳。这种态度上的差异是微妙的，但很重要。随性者不乐意尝试。他们不习惯鼠标右击、按住键盘不放，或者试一下不同的菜单项，为的只是看一看会发生什么事

情。因此，不要期望随性者去做什么试验，除非他们确信那样做是安全的。同样，不要期望一个随性者能够同意一个改变，除非它很值得去冒险，并且有个安全的退路。

- **受过自由教育的人设想万事皆易。**技师知道任何事情都不简单，因为技师关注在细节上面。随性者关注的是大局，认为万事皆易；如果那不是事实，那它也应该变成事实。两种观点都不错。你做的所有事情，概念上来说都应该是简单的，而且容易在较高层面上解释，否则它自身很可能不堪重负而崩溃。然而，魔鬼藏在细节里面。对于在细节中长大的技术人员来说，“简单”是困难的。但是……
- **受过自由教育的人不关心细节。**技师热衷于细节，但它通常不是项目的最好部分。然而，因为随性者关注的是大局，小细节只会干扰他们手头的事情。因此，如果你正打算向一个随性者讲述你的想法或汇报项目状态，你必须抛开细节，并且把你工作背后简单的、较高层面的概念和需求清清楚楚地告诉他。否则，你最好做好准备，因为你将得不到他的支持；他也有这样做的资本。随性者不是三岁小孩。他们是把你的工作跟其他所有正在进行的工作联系起来的指挥官，你要做的，只是为“你的工作如何适应到其中”提供一个简单而清晰的描绘。
- **受过自由教育的人不关心纯度。**技师热爱纯度。对他们来说，那是优雅，那是美。纯度冲走了所有丑陋的繁琐之物，只留下问题朴素的真谛。遗憾的是，所有这些都超出了典型的随性者能够理解的范围。随性者从一开始就不关心丑陋的细节。他们期望任何事情都是纯粹而且简单的。告诉一个随性者你找到了一个雅致而简单的解决方案，得到的回应很可能是，“是啊，我本来就希望如此，你以前的解决方案是什么，难道不是这样吗？”如果你想说服一个随性者采纳你的雅致架构，不要在纯度上纠缠不清，而要向他渲染由此给客户或业务带来的利益，比如更高的稳定性、更容易维护，等等。
- **受过自由教育的人关注感受和仪表。**技师通常意识不到有感受和仪表这么回事。我妻子花了多年的功夫不断指正我，我才开始意识到这些东西的重要性。随性者非常关注感受和仪表。我知道这看起来很愚蠢，似乎还会招致反效果（只重表面而忽略实质），但是我们别无选择，因此也不必费神去争吵。当你向一个随性者提出一个想法或计划时，一定要考虑人们对它的感受（随性者设想所有人都和他们自己一样随性）。是不是要顾及有些人的面子？你是否正在侵占别人的领土？你是否正在冒犯某个权威人士？你不必解决所有这些问题，但你必须考虑到这些，并且向随性者指出来。他们会因你的善解人意而感动，从而帮助你解决人这方面的问题。

很显然，我这里以偏概全了。不是所有受过自由教育的人都有上述这些特征，也不是所有的技师都有那些对立的特征。不过，你不能假设所有人的思维方式都跟你一样。想要实现有条理的沟通，只需把你预想的主张暂时搁置一下，这将对你大有帮助！

通过安检

这些特征中最重要的含义是，受过自由教育的人倾向于团结在权威人士的周围，并且保护他们，这也使得他们很特别。因为随性者尊重权威，关注感受和仪表（包括他们自己的），他们不能容忍随便一个什么人就去跟高层管理者或者关键客户对话。你必须首先得到他们的认可。偷偷地溜过随性者的“安检系统”可能很好玩，可能第一次会奏效，但当他们意识到你的所作所为时，他们会觉得被冒犯了，并且一直耿耿于怀。

所幸的是，你可以用委婉的方式来表达你的想法。说清楚为什么跟客户或者管理者对话很重要。允许随性者来引荐你，并为你搭建舞台（仪表、规则）。搜集你的问题，把这些问题趁这个机会都抖出来，如此一来，客户或管理者的时间就得到了尊重，并且他们也觉得值了（他们的感受——一种对权威的尊重）。除非你被直接问及，否则把繁琐的细节和绝妙的设计都统统省去（细节、纯度）。至于行动方案，要让客户或管理者明明白白地做选择题（简单）。

结果会怎样呢？管理层会喜欢你，你的客户会喜欢你，曾经被你“蒙骗过关”的随性者也会喜欢你。

着手去改变

如果你想推动团队的工程改进，你必须说服那些随性者。这并不容易，因为改进意味着改变，改变意味着破坏规则，而规则就是规则。然而，你能成为改变的有效推动者，只要你遵循下面这些很简单的策略：

- 首先，描述你正在试图解决的问题。使用统计数据以使问题看起来很恐怖。（你的问题也许真的很恐怖，但随性者尊重的是“数字”权威。）不要欺骗——请使用真实的度量。你要做的是，让问题看起来很恐怖，以证明目前的规则是不安全的，需要改变。
- 其次，描述解决方案必须满足什么条件，才能保证项目和团队的安全——比如说，一个回退策略、条件编译、策略设置、定期复审或者管理者批复程序。不要只是把这些东西拼凑在一起，而要真的考虑到人们会担心什么。你必须在开始描述解决方案之前做这件事情，否则，随性者的忧惧会将你团团包围，使你四面楚歌。记住，改变跟感受总是联系在一起的。
- 最后，描述你的解决方案。谈一谈你的方案是怎样满足安全条件的（那些你已经在前一步骤提出的条件）。然后谈一谈你的方案如何能把项目带入一个更好的状态。建议你使用统计数据来演示结果（比如回退或者一级 Bug 下降的百分比，或者更短的稳定化时间）。切记，保持方案在较高层面上进行沟通，并且让它尽量简单。统计数据特别重要，因为没有其他更权威的方式来证明你取得的进步了。为了避免统计数据成为攻击目标，务必要使用团队度量而不是个人度量。当你的新规则使你的目标达成的时候，记得一定要庆祝一下！

更好地在一起

你一不小心就会去冷嘲热讽那些受过自由教育的人，或者任何跟你思维方式或工作方式不一样的人。不过，不同的方法有不同的价值。最后，我们大家都会受益。有点过于浪漫了，但事实的确是这样。

学会感激随性者给我们的技师世界带来的平衡吧，你会发现你的效率大增！理解你们之间的差异，调整你的方法，感知你的信息，尊重你的听众。最后，你将得到所有的东西，并且不留下任何遗憾。

作者注：自从我离开微软的工程学习与开发部门（Engineering Learning and Development organization）以来，我的妻子就已经在编辑我的专栏，每个月都在做。虽然我们是两种非常不同性格的人，但我们仍然配合得很融洽。我可以真诚地说，自相处 26 年，结婚 22 年后，再也没这样融洽过。

2005 年 11 月 1 日：“废除工种——有什么理由搞专业化?”

我们为什么要测试者? 我们为什么要项目经理? 甚至我们为什么要区分不同的工种? 难道这样不是很没有效率吗? 我们为什么不只设立工程师，让他们做所有必要的事情，发布符合质量要求的产品，取悦我们的客户? 是不是程序员没有能力理解客户的需求，或者执行测试去检验他们的工作? 当然不是!

> **作者注：**我在本章一开始的介绍中谈到了，把拥有各种技能的人组成团队，让他们在一起为一个项目工作，这是非常重要的! 但这并不是说，那些人必须被专门分到不同的工种，然后各自承担相互排斥的职责。

然而，我们不知何故还保留着病态的工种，尽管它们制造着障碍、错误沟通和冲突。像致命依赖、交接地狱、过失黑洞、令人窒息并且毫无意义的批复、单调而让人头脑麻木的会议等，这些听起来更是臭名昭著。我们真的这么缺乏安全感，以致每个人都必须是“特殊的”吗?

这不只是荒谬，而且是反生产。开发者和测试者等待项目经理的规范书，项目经理和测试者等待开发者的代码，项目经理和开发者等待测试运行起来。当然，总有其他的事情可做。但如果每个人都投入到所有的事情当中，那么所有人都关注在其中优先级最高的事情上面，人人都是客户的拥护者，人人都在贡献质量。这就是团队方式。这就是敏捷方式。这是老风格，也是新风格。这是理想境界。何乐而不为呢?

历经未来的日子

微软一开始并没有设立项目经理和测试者。我们从理想境界起步，然而却在天堂迷了路。怎么回事? 我就此请教了一些老前辈。基本情况是，不是每个人都想做某些特定的任务，有些人比其他人更能胜任某些特定的任务，一些重要的任务被遗漏了。于是产生了这样的一种必要，大家开始专业化（专人做专事），以填补理想境界里的裂缝。

当然，那是很久以前的事了。那时的市场不同，我们的形势也不一样，而且当时的经验也比不上现在。但是如果我们回到“每个人都做所有的事情”，历史会重演，大家会再来一次专业化吗? 背后是否有一个基本法则在起作用呢?

遗憾的是，的确有这么一个法则。专业化是无法避免的，也是必要的。

不管怎么样，在你变得过于兴奋之前，让我来先说两点：

1. 专业化的需要是微妙的。它并不总是适用。
2. 我是对的。

考察它的极限

我怎么知道我是对的呢? 因为我在高中数学队时就学会了一个窍门。（好吧，我承认我是个呆瓜。让我们继续。）这个窍门是：当你需要了解一个区间问题时，可以考虑它的极限。回到我

们的问题中来，那就是，什么才算最佳的软件开发角色结构。考察它的极限：一端是编码回答一个面试问题，另一端是编码实现 Windows Server 操作系统。

当编码回答一个面试问题时，你可以自己做所有的事情。你必须了解客户（面试官），理解需求，说明解决方案，编码实现它，然后对它进行测试，最后交货。如果你做不到这些，你就不会被聘用。这个例子传达出来的道理是，在“简单”这个极限，专业化是没有必要的。

编码实现 Windows Server 操作系统需要许许多多的开发者。但必须专业化吗？绝对要的！因为有些代码过于复杂了，以致很多开发人员都理解不了。对于客户设计和测试（质量保证或质量控制）也必须要专业化吗？绝对要的！不过，因为 Windows Server 的范围太大了，不是很容易看明白，还是让我来举个小例子吧：一个 Xbox 足球游戏。

足球是门科学

编码实现一个 Xbox 足球游戏需要不止一个开发者。专业化因为计算机图形组件、人工智能组件、统计组件等这些模块的存在而成为必要。不过，让我们将话题转移到项目经理和测试领域。这里也需要专员吗？绝对要的，不过不是在小的功能层次上。

开发一个足球游戏要求你知道关于足球的所有细节：每个规则、每场比赛、每个信息、每支球队、每个体育场、每个球员以及他们的薪水，等等。然而，不是开发团队中的所有人都要求知道所有这些东西。实际上，大部分人可以对它们一无所知。他们有其他的东西要去关注，比如计算机图形和人工智能。但总得有人要知道呀——游戏设计师。

需要有人去验证游戏的结果。这个人必须是专业的视频游戏玩家，并且会踢足球，对游戏可以怎么玩、应该怎么玩如数家珍。这个工作的复杂度是巨大的，但跟实现相比，它在不一样的细节层次上。编写代码的开发者在一个低得多的细节层次上操作。即使是开发架构师，他们涉及的也是不一样的复杂片断。但总得有人在用户这个层次去验证结果呀——游戏测试者。

Windows Server 比起一个 Xbox 足球游戏要复杂得多。不说成千上万，至少也有成百上千种体验，必须在客户这个抽象层次去完整理解、设计和测试。我非常希望，我们的工程师都有足够大的脑袋，来保证所有细节在每一个抽象层次上都正确无误；我也希望他们都是多路系统、完整业务模式、模块分解和细节实现等方面的精密专家。但这荒谬到了极点！这个例子传达出来的道理是，在“复杂”这个极限，专业化是无法避免的，也是必要的。

> **作者注：**这部分内容让很多敏捷实践者困惑不已。也许是因为没有太多像 Windows 这么大规模的产品来证明专业化的必要性。我要再次重申我所获得的经验：项目在不同的规模和抽象层次需要不同的管理。在有成百上千或成千上万个工程师一起工作的产品层次上奏效的东西，对于 5～8 人的功能团队层次是没有意义的。

两者之间的距离

好了，我们学到了什么？在简单这个极限，专业化是多余的。在复杂这个极限，专业化是必要的。因此，问题不是：“我们应该要专业化吗?”问题是：“在什么复杂或抽象层次我们才需要专业化?”

我认为在具体的功能层次搞专业化是对时间的一种浪费。开发者应该理解具体功能的所有需

求（规范书、应用范例和角色扮演），然后展开设计，进行单元测试，最终编码实现。测试者和项目经理在具体的功能层次不插手规范书和测试。如果开发者在细节层次的工作做得不好，那他们就不称职。

我同时认为专业化在产品层次是必要的。我们需要项目经理去由内而外真实地理解客户，在他们和团队之间协调沟通，并且保证团队时刻关注在正确的目标上。我们需要测试者来确保客户满意。不仅仅是让产品能够工作，而且要让产品以客户需要并且想要的方式去工作。

你深陷其中

然后呢？最大的争论是：在两者之间的哪里去画一条分界线？什么时候你可以不再需要独立的工种？能不能让开发人员设计和测试对话框或者应用程序编程接口？你可以做到多细节的程度，以免过度“杀伤”项目经理和测试者而招致反效果？我个人认为这要看情况。

在哪里划分界线取决于不同的产品。如果那是一款很熟悉的产品，目标也很明确，那么你很可能不需要很多的专员。如果那是一款不常见的产品，目标也很模糊（至少对于工程师来说是这样），那么你需要较多的专员。复杂度很自然会增加负担，也就需要更多的专业。

最后，在任何时候，你都应该避免“只要可能就进行专业化”。这会增加极大的负担，并且导致功能紊乱。如果你有些事情是任何人都能做的，比如检查创建状态或者领取比萨，那么每个人都应该帮忙去做。要不然，你的团队成员就会显得很自私、自我本位、不职业。

当你确实需要专员的时候，接受并拥抱他们吧！给他们机会去学习、成长和领导。我们的成功有赖于此。不过，千万不要让他们骄傲自满。尽力通过使用共享空间、持续沟通、对团队工作和团队主导权的关注，以避免缺陷和障碍。记住：团队第一，比萨不会自己走过来！

> **作者注：**在本章后面，我将讨论这种开发与测试不和谐所带来的影响，见“测试不该不受尊重”。

2009 年 1 月 1 日：“持续工程的鬼话”

管道转接口将废水引向一个又一个更大的管道中去，直至其被排出。这是因为污水是往下游流的，这可以形象地描述测试、实施及持续工程团队接手的中间产品的质量是怎样的。毕竟，他们也位于设计及开发的下游。

我在“感觉性急——测试者的角色”（本章前面部分）中已谈及如何为上流测试提供中间产品的问题，并且在“碰撞测试：恢复”（见第 5 章）中谈论了如何在实施阶段使服务从失败中恢复的问题。就像大多数工程师一样，我曾忽视了持续工程（SE）——也称为工程污水净化中心。乍一想，这种比喻意味着我们发布给客户的产品已经经过“净化”，不是这样的。SE 更类似于漏油事件后的环境恢复过程——不讨好，很艰难也很杂乱。

想象一下，净化小组要想清洗海鸟羽毛上的污油，他们最应该想到的是什么呢？当然，他们会对鸟儿们（客户）充满同情心。当一些不可避免的错误最终导致悲剧（到处有 Bug 的软件）的时候，就会有一种深深的挫败感。这让人很抓狂，恨不得让那些肇事漏油的蠢驴（就是那些让 Bug 丛生的工程师）来代你干这些事。

作者注：你在写一个很有争议的专栏，这很冒险，有人会因此受伤害。现在这就是一个实例。虽然，我找了三个持续工程团队经理事先对本栏目进行审读，以发现讲得不恰当之处及无心之论，但还是有很多持续工程团队的人员受到了深深的伤害。我又重读了本专栏一遍，很快我就发现不仅仅是我质询了那些持续工程师应该如何做的问题，更重要的是我把他们的工作与污水净化与灾后恢复对比了起来。

显然这些观点很像事后诸葛，但人们未得要领。可能审读者们能懂。我是极力支持持续工程的，从没有羞辱他们的意思。我的本意只是对持续工程团队怎么会陷入不讨巧的困境表示同情，而绝不是贬毁他们。遗憾的是，好心不一定有好报。

以下是我写给一位持续工程管理者信中的一段话，他曾对此深表忧虑。

“感谢你对本专栏中令人尴尬但又无意伤人的见解指出不妥之处。但希望你明白我对你的团队及持续工程团队是深怀敬仰的，没别的意思。我只是希望核心工程团队能对他们的设计水准与工作质量担起应有的责任，那么你们的工作与我们的客户就获益良多了。”

委托他人

参与设计、构架及测试的工程师们应该与产品发布后修复 Bug 的工程师是一样的吗？这是持续工程典型的一个问题。

如果修复发布版 Bug 的工程师就是创建发布版的人，那么通常软件修复就会更完善。工程师们感觉到了他们自己错误的可悲性，那么就会将这些修复整合进下一个版本中。但话说回来，可能不会再有下一个版本，因为这个发布版本的工程师并不固定（不是原来那拨人）。

如果有一支专门独立于核心工程团队的持续工程团队，由他们建立代码库的说明，那你应该就可以委托一个供货商来维护老的版本，而你就不用因此在新版本上分散精力，打乱开发进程。

那么就找一个供货商呗，对吗？不，绝对不是。如果两种方式都可以的话，让创建发布版的工程师修复发布后的 Bug 要好得多。只有白痴才相信一个缺乏责任心的人会一直保持高效率、高质量。当然，这世界就是有那么多的白痴。我又跑题了。

必须有人为此负责

是的，有一支专职的持续工程团队很好，但长期这样带给团队成员及客户的只有痛苦。为什么？因为你可以减轻因为发布版的修复工作而分散核心团队的精力，但是却不会因为有专职的持续工程团队减少软件问题。

让我们再次审视一下专职持续工程团队的问题：

- **缺乏热情**　较之核心工程团队，专职持续工程团队体会最深的是什么？灾难。是吗？那天天要面对的是什么呢？永无止境的灾难。换句话说，持续工程团队，想说爱不容易。
- **不经意的修复**　想正确修复，就要充分认识修改后的所有影响，对代码库的影响在哪一部分你必须深入了解。我们假定，这个核心工程团队达到了这样的程度。因为核心团队通常比持续工程团队庞大得多，所以持续工程团队基本不可能对修复引起的影响有真正的了解，事实上只会比核心团队差。你确实可以让持续工程团队与核心团队商议，但老是这么做就会失去持续工程团队本身的意义。

- **重复劳动** 只要有两个团队同时对相同的代码进行修复，显而易见，就是重复劳动了。让两个团队学习相同的代码，在相同的代码中找 Bug，修改相同的代码，再将相同的代码进行测试。毫无疑问，这只会让事情变得更糟，除非你不在乎把这些修复整合进下一个发布版。
- **错误的责任人** 专职持续工程团队的主要目的就是要让核心团队心无旁骛，使他们省去修复的事。而这样核心团队并不清楚他们在老代码中犯下的错，也不知道如何防止这些错误在新版本中再次发生。更糟的是他们无动于衷。有责任心的人也不再写更有质量的代码，而粗心的人也不修复错误，因此，我们也别期望软件有什么改进。这真是对付幸灾乐祸的对手及可悲的客户的良方。

我在干什么

话说回来，当核心团队修复先前犯下的错误时，需要做很多事以避免打乱下一个版本的发布计划。让我们来看看以下大家时不时反应的但又不可逃避的问题，逐个摆平它们。

- **旁枝末节** 你怎么使核心团队的时间不要浪费在与 Bug 无关的问题上，或有一些什么小技巧避免核心团队浪费时间呢？让一个专职团队对问题进行会诊。注意，这个团队不是开发团队。他们纯粹是决定哪些问题是值得修复的评估团队。这样的话，只有认为值得的事才会交由核心团队来干。
- **优先级别** 你如何在之前发布版的 Bug 修复工作与当前新版本开发任务之间取得平衡？设立一个专职评估团队对修复工作进行优先级排序。有 4 种方式：立即修复（很少出现的“马上报告副总统”的情况）、紧急修复（下一次计划更新）、完全修复（下一个 service pack 或更新）及不用修复。这些方法给了核心团队以清晰的信号：什么 Bug 在什么时间修复。
- **不可预知** 由于软件发布后问题的不可预知性，你如何使核心团队的计划安排变得容易？要规范化。一个月部署更新一次，由评估团队每月对紧急修复进行优先队列排序。然后核心团队每月留出必要的时间，将这些修复工作的设计、执行、测试与部署安排成一个可预知的日程表。这不仅对核心团队有效而且对客户也很有效。谁都喜欢事情可以预知。

另外，评估团队可以为了使核心团队方便地修复 Bug 建立虚拟镜像，提升客户的更新体验，并将客户需求及长期可持续性特征反映到以后的版本中。

> **作者注：** 很多读者以为我是提倡去除持续工程团队的。实际上，我是想保留持续工程团队，但要减小他们的规模，让他们专注于问题评估及优先级排序上。这种更小型的持续工程团队还需要提供一种和谐、系统性的支持，这对持续工程团队顺利运行很有必要。他们应该为他们找出的关键问题设计一个以客户为中心的修复方法。他们必须对这些问题优先级进行恰当的排序，以达成一个客户解决方案。然而，小型的持续工程团队成员不应该是修复 Bug 的人。这个责任应该由核心工程团队来承担，他们完全了解他们所犯下的错误，也知道如何避免这样的问题。

毫发无伤

看吧，没那么复杂。你节省了人工。修复更完善。你能预先发现曾有的类似问题。你不用再摸着石头过河。你也明确了只有核心团队才为软件质量负责，并从错误中取得经验。所有的开销不过是通过一个小小的专职团队来对问题进行评估及优先级排序，把握每月的更新进程。从而，这个团队感觉到自身的价值，觉得他们是与众不同的，他们扮演着非常重要的角色，他们是解决客户问题的直接参与者。

是的，污水顺流而下，没人愿意清理。但是，只要加入适当的步骤，你就可以减少污水，而让有责任的拖把来清理污秽。对我来说，这样很公道。

> **作者注：**如果你是专职持续工程团队的一员，并且你觉得这样了无生趣，时常修复一个问题又产生另一个问题，重复劳动，而且核心团队又没责任心，你该怎么办？这里有些小建议：
>
> - ❑ 与核心团队轮换工作。每个人每年要有一或两个月的时间呆在持续工程团队。这不仅仅是个想法，我已经在实施。
> - ❑ 对你的效率及效力进行度量，可以是每种方法解决问题的平均时间、重犯的比率、团队的激情、以及客户对修复的认可度（用记分卡记录）。优化并发布你的成果，告诉核心团队工作完成得有多漂亮。
> - ❑ 每月发布更新一次——每月开展一次表彰大会。

2011 年 5 月 1 日：“测试不该不受尊重”

我爱微软。这些年来我们大家相处得很愉快。如果你们已有了长期融洽相处的友谊，那么你就会知道这意味着什么——就是微软的有些事常遭我唾骂，气得我直跺脚。我已学会了忍耐，但这些事始终让我心有余悸。

首先的例子就是我们对类似测试这样的关键工种极为不屑。测试工种是微软两个最大的工程工种之一，也是工程三人组不可或缺的一部分。为什么我们不能像给予工程三人组中的另两者（项目经理及开发者）一样，给予测试人员一样的尊重呢？

微软始创于开发者并由开发者主导好多年了。我们名为一个前规划工程师所管理，实则项目经理备受令人妒忌的礼遇。开发者不会绘画，对风格设计也毫无感觉，所以设计人员只是锦上添花之用；开发者也憎恶文书，所以内容发布只不过被视为可有可无；本地化及多媒体是个很炫人的事，但只在创建的时候才派上用场。而测试呢？开发者认为测试是很容易的，不过用来修修补补，所以开发者认为测试者低他一等。

> **作者注：**请注意，我并不否定这些工种应得到赞誉。我只是讥讽开发者对他们的这种肤浅认识。

风马牛不相及

因为开发者认为测试相对开发来说太容易了，他们认为他们也可以做测试者的工作。毕竟，敏捷方法不是很酷吗？我们难道不是软件工程师吗？是的。如果我们给和平以机会，所有人都可

以团结在一起了。别天真了。

在独立的环境下（单元测试），开发者可以验证他们开发的个体组件是否在指定的情况下正常工作。但他们的组件是否在独立指定的环境之外以一个整体系统性地运行，这还有很多其他问题要验证。为什么会这样？因为管中窥豹。开发者以一种固定的模式设计编写他们的代码。开发者就是这样看待他们的成果的——他们也注定会这样。

真实的测试应该在有悖常理及主观意愿的情况下，验证代码是否执行正确。想正确地测试代码，你必须彻底忘却怎样及为什么编写这些代码，而不是想着在一种恣意甚至诡异的情况下，这些代码该如何被使用。开发者不会这么想——他们也不应该这么想，而仍然是按自己的意图写代码。测试者会考虑这些问题。这就是为什么我们需要测试者。

很简单

即使这个测试原则对于高质软件是必需的，一些团队仍然想把他们所有的测试者换成开发者，期望这种组合的团队一起编写所有测试代码——这确实是个增加开发人员人头数的好方法。经历这种变换的团队遭遇了以下几个问题：

- **他们失去了测试的主导地位及测试主管** 先前的开发组将主导着这个工程组合团队。（我将在下节中解释原因。）这让之前测试的主导地位削弱成了个体参与的角色。测试主管们由于这种地位的降低而失去了责任感并选择离开。之前优秀的测试者也选择离开——不管将这个组合的管理团队鼓吹得多光鲜亮丽，说测试成员很重要，但毕竟，他们的技能完全没有得到肯定。
- **他们失去了测试者** 在失去测试的应有地位，且测试主管也离开之后，所剩的测试组成员开始顺从于开发者的思维方式。这是因为测试者得不到尊重，他们以开发者为标杆，即使他们使自己表现得不同而更像测试者，他们发觉这再没有测试者的职业发展空间了。因为，毕竟现在的团队中再没有他们独有的角色模式，现在个人向上发展的空间也只有开发（而不是测试了）。
- **团队士气低迷，特别是测试者** 团队成员及主管的离开影响了士气。特别是测试者坚守原则的特征受到了最大的影响，他们失去了他们的角色定位、他们的特性以及他们的好口碑。（就算是最好的苹果在一个橘子林里也是得不到什么好评价的。）
- **代码在一开始时质量很高，而随后逐步降低** 开始时的代码质量得以提高是因为开发者突然意识到他们失去了安全保护网，并真正开始写单元测试，再实施设计及代码复审，并对创建警告加以足够重视。（当然，开发者本应该这样做。）然而，久而久之，系统、溢出错误开始到处滋生。没有人再顾及到这些，所以最终却是由很不恰当的测试者——客户——来发现这些问题。

开发与测试的组合在单元开发阶段（如测试驱动开发）或许还有些用。对于组件层面有严格品质保证要求的团队来说，要开发易于理解的组件，这种组合同样可以适用。然而，开发与测试的组合在系统层面就不适合了。我在本章之前的部分“废除 2 种——有什么理由搞专业化?”有更详尽的讨论。

我不够受重视

缺乏对测试人员的尊重在等级及职业发展方面则更加明显。依据其在公司所处位置的不同，测试人员处于三层级别的最下层，在开发人员及项目经理之下。换句话说，相对于项目经理及开

发人员，测试人员同他们一样在同一项目中拥有相同能力，测试人员的职业舞台却永远处于他或她的同事之下。令人难以置信！

目光短浅的开发者会这样说："是的，但是测试相对开发及项目管理来说并不那么难。"真的吗？你试试看。试着写一个即使开发人员更改了配置及数据时，还能无时无刻都正常工作的自动化系统；试试执行一次穿透测试来弥补漏洞，从而挫败狡黠的黑客；再试试注入测试来找出错误的模块；还有通过系统测试来找出泄密漏洞；或者是通过完整场景测试验证我们下的数十亿美元的赌注。这些例子还不过九牛一毛。

"是的，但是我们的测试团队还没那么先进——他们并没有把所有这些事都干了。"确实是！我们并没有足够地看重测试工种来提升我们的测试能力，从而使测试工种与项目经理及开发人员处于同等重要的层次。

测试真麻烦！我们需要很多人手来干好它，我们还需要工资支出，对他们进行培训并期望他们达到世界一流水准。

是的。测试人员的工作与项目经理及开发人员的确实不同。但是我们不能设计构思一个复杂的系统并认为测试这样的系统却要容易得多。有多少付出就有多少回报。对测试投入得少将产生失衡。我们就牺牲了质量、生产力及效率。质量上的牺牲是很显然的，在生产力及效率上的牺牲来自于不彻底及轻描淡写的测试所带来的高错误率、高返工率以及高额的支持与持续工程成本。

> **作者注：**当我对测试的投入有所抱怨时，我听到的是："是的，但是 Google 及亚马逊的测试人员就很少。"亚马逊对 Bing 及账号管理之类进行了仔细的测试。Google 对它的搜索结果进行反复的分析。而对于微软，我们的业务比亚马逊及 Google 的更广泛。我们是一个平台公司，是一个企业化的公司。客户对我们的期望很多，对质量要求各异而且是必须实现的。是的，产品与服务并不很关键或容易受干扰，当"任务关键型"的产品及服务两次发布时间跨度需几年的时候，期间每个月的更新并不都需要这样的深度测试。我们不需要事事完美，但相应的产品确实需要相应的质量要求。

我们要适应现实

如果我们并不像对待项目经理与开发者一样对测试者寄予厚望，从而放弃对产品质量、生产力、生产效率的要求，就实在是太糟糕了。更糟的是降低这种期望将释放出一种清晰的信号，即被认为：测试者需要变成另一种角色。这种观念意味着不负责任与悲剧。

- 轻视测试者是不负责任的。因为一些重大的工程问题都涉及测试——产品测试、多种内核的及高并行系统的测试、成千上万的机器服务测试、全球化的分布式云服务测试、云系统安全及隐私测试、混合过程及功能的语言测试、普通用户接口测试，等等。
- 轻视测试者是个悲剧。因为我们释放出这样的信号：测试不是一个正规的职业途径，但事实上其有一条成熟的职业路径可以进入高级工程领域，如测试经理及测试个体参与者。如果不是选择这条路径，测试者就放弃了他们的原则而去做其他的事，结果就不可预知。

轻视测试者，就等于说我们在怂恿他们离开他们热衷的职业，而这种职业对微软很重要且充满着机遇，从而引导他们选择并非他们想要的、有碍他们成长的职业领域。这样的做法很拙劣。

作者注：微软最近宣布将改变其薪资计划，增加工程师的基本工资，包括测试者。这很好，我也很高兴。对测试工种的投入已是一种独立的议项了。测试者要成为测试领域的专家，就像在工程管理及开发领域一样。

执行官可能义正词严地问："为什么我们要在测试上增加开销？我们的投资回报是什么？"如果我确信有丰厚的回报，我就把前面这句话反转一下。如果我们想节省开支，为什么我们不从有双倍工资的开发人员身上扣下来一些呢？因为这样我们的产品质量及创新将受到损害。为什么会受损？因为我们的产品是一个广泛的多用途的平台，需要专业的工程师创建提升——以及测试。

告诉我，这跟我有什么关系

当我们想投资测试的时候，我们还是会贬损这个工种。在测试领域我们有非常好的高级别的管理团队，而每年加入测试者队伍的人却很少。甚至测试远远落后于项目管理及开发，测试预算的提升也很少。（预算增加的比例与其他工种一样，但这种比例会被误解，因为项目管理及开发的工资本来要高。）只有一小撮的个人测试者及测试经理达到了其同伴（项目经理及开发者）相同的水平。

我们怎么可以让这么关键的工程工种受如此的轻视及损害呢？是否我们自大愚蠢到认为测试者根本不需要或我们的程序根本就不会出现什么大问题吗？

作者注：正如你所想，这篇专栏对于微软的管理者是一个警醒。他们队伍的两极分化到了什么程度。有人爱有人恨。原因很多，包括了方方面面。讨论这种公开的对话我很高兴，我也很开心，人们已经有所谈论，他们已有所改变或正要改变现状。但对于一个公司及一个行业来说，我们还有很长的路要走。

该是将我们的经费放在紧要的事情上的时候了，也该是把测试提升到一个重要的层面上来的时候了。我们雇用了最好的测试人员，让我们为测试人员铺就一条通往副总裁或技术官的职业道路。让我们开始尽力识得我们拥有的天才们并培养他们。测试应得到我们的尊重——我们的客户、我们的合作伙伴以及我们有赖于其的相关业务人员。

作者注：如果你不是一名测试者或跨领域的领导者，你该做什么？

- ❑ 了解并接受测试人员的理念与技术与开发人员的是大不同的，但是他们面对的问题对于我们的产品质量来说同样是复杂及关键的。
- ❑ 通过设计及代码复审、代码分析及全面的单元测试等方式，在一开始就编写出高质量的代码，这些方式使测试者把他们的注意力放在提供独一的质量保证并发现开发者通常不能检测到的系统问题上来。
- ❑ 鼓励你的测试团队雇用优秀的全职测试人员，这些人能全身心地应对我们面临的测试问题——当这些问题被解决时，将有助提升我们产品的质量，提升我们测试的水平及测试人员的水平。

顺便说一下，对于服务工程师我将写篇类似的文章，这些人与测试人员一样不被理解。

第5章

软件质量不是梦

本章内容：

2002年3月1日："你对你的安全放心吗"
2002年11月1日："牛肉在哪里？为什么我们要质量"
2004年4月1日："软件发展之路——从手工艺到工程"
2005年7月1日："复审一下这个——审查"
2006年10月1日："对质量的大胆预测"
2008年5月1日："碰撞测试：恢复"
2008年10月1日："盯紧标称"

有些人嘲笑软件开发，说如果大楼都像软件一样构建的话，一只啄木鸟就可以毁掉整个人类文明。这个说法非常有趣，或者令人烦忧，但不管怎么样，这是个误导。早期的建筑没有地基，早期的汽车经常抛锚，早期的电视需要不断拨弄才能正常工作，软件也不例外。

起先，微软为早期的用户开发软件，以使他们能够很安逸地更换个人电脑主板。那个时候，产品上市的时间比质量更重要，因为早期用户碰到问题可以绕开，但他们没能力让时钟慢下来。以最快的速度出货意味着快速编码，然后仅仅修复影响产品工作的Bug。

如今我们的市场是广大消费者和企业，他们看重质量胜过对新功能的抢先体验。市场改变是逐步发生的，因此微软最初的回应只是修复更多的Bug。不过我们很快发现，Bug修复花费的时间比编码更长，而且这个过程慢得难以置信。发布高质量软件最快的方法是，设法在早期阶段纠正错误，代码第一次就能编写正确，并且把返工降到最低程度。在我撰写这些栏目的时间里，微软已经转向了强调上游质量的方法。推动这次公司范围内变革的最主要原因，就是2001年后期的一系列互联网病毒攻击。

本章将向工程师们宣扬软件质量。第一个栏目评价了安全问题。第二个栏目分析了为什么质量是必要的，以及你如何获得质量。第三个栏目解释了能够让缺陷戏剧性降低的软件工程方法。第四个栏目讨论了设计和代码审查。第五个栏目描述了在客户开始体验之前可以用来预测质量问题的度量方法。第六个栏目重点讨论了使软件更具弹性的技术。在本章最后强调了软件质量的五个基本点。

尽管所有这些栏目都提出了有趣的观点，然而第二个栏目"牛肉在哪里？为什么我们要求质量"则更突出了一个重要的转折点。我当初写下它的时候，微软内部或外部很少有人相信我们在质量上存在着严重的问题。几年之后，栏目中提到的很多概念都得到了承认。虽然推动变革仅靠只言片语是远远不够的，但呼吁采取行动、促使大家做出反应总是好的。

——Eric

2002 年 3 月 1 日：“你对你的安全放心吗”

我知道安全纯粹就是狗屎。我知道每当有弱智的黑客在系统的一小块代码或者配置中发现了漏洞（他们从来就不敢奢望自己能开发出这样的系统）时，新闻媒体的那帮可恶臭虫们，就会对这个狗屎安全问题大肆渲染、滥加指责，并且告诉我们的客户我们的代码“臭气熏天”——仅仅是因为某个满身痘疮的恶毒小人操控成百万行代码中的两行而妄行不法的勾当。我知道的事实就是这样。

微软拥有成百万行的旧代码、超大的服务器网络群，还有成百上千的外部合作伙伴和依赖方。它们中的任何一个都可能成为卑鄙、满怀报复心理、无所事事的人的牺牲品，或是某些人作呕的便盂，这些人的衣服还得靠他们的妈妈来洗。但是至于安全问题，我们应该首先把精力放在哪些方面呢？

> **作者注：**哈，那时的黑客还只是被误导了的年轻人，他们只是想挑战权威，为自己赢得名声，这种“昔日好时光”早已不复存在。如今的黑客攻击已经形成了一笔不小的生意——无论是·黑客防御和跟踪，还是从事有组织的犯罪。回顾以往，我对早期的黑客可能过于苛刻了。这个世界并没有那么理想，不能期望每个人都那么友好。不过早期的黑客唤醒了我们，让我们回到了编写安全可靠代码的正确轨道上。

当心晃动的钟摆

有些人认为，所有可能的（攻击）弱点都必须被修正。这值得表扬，但也近乎疯狂。我们不能抱有这种妄想，否则我们的产品永远也到不了用户手上。

当初我在波音公司工作的时候，参与“黑色项目”的人在完全没有窗户的大楼里面工作，他们禁止使用网络，并且每个晚上他们的磁盘驱动器都会被卸下来锁在保管库中。尽管应用了所有这些保护措施，这些项目仍然被认定是易受攻击的。

即使是最大牌的安全专家，可能也不认为我们的客户应该被要求堵上窗户，远离网络，并且卸下他们的磁盘驱动器。然而，我们必须比过去大大提高安全门槛，并在我们的组织中自上而下要求责任到位。这里需要记住的关键点是，我们所做的一切都是为了向我们的客户传递可信赖并且令人愉悦的体验。

另外一些人认为，我们只需要集中精力搞好防火墙、协议和通用语言运行库就行了。他们假设只要这些东西安全了，我们就能高枕无忧。其实这很无知，而且完完全全是很危险的想法。

Michael Howard 和 David LeBlanc 写的《Writing Secure Code》（Microsoft Press，2002）一书（微软内部必读之书），其中有一整章用来论述编写安全的 .NET 代码，这一章几乎引用了其他 15 章的内容。那一章已经解释清楚了，软件弱点并不局限于缓冲区溢出、不安全协议和未设防的端口。即使这些东西是唯一的攻击点，防火墙、安全协议和托管代码也不足以抵御来自恶意数据的侵袭。

> **作者注：**在本栏目中，我引用的是《Writing Secure Code》这本书的第 1 版。这个栏目是那一版图书出版后不久写成的。很自然，这本书的后续版本收录了更加有用的信息。顺便说一下，Michael Howard 有着很重的新西兰口音。

做正确的事

当然，正确的做法应该是考量所有可能的软件弱点，然后对每个弱点在对客户可能造成的风险方面进行评级。查找弱点并没有像它听起来那么困难。通过把你的网络或者客户端程序分解成组件和接口，你能很快发现潜在的问题，并使用 STRIDE 模型把它们分类。通过在你的代码中搜索危险的 API 函数调用（参见《Writing Secure Code》一书的附录 A），适当地限制权限，并且检查输入数据，你就能找到一大堆的简易攻击点——黑客通常找的也就是这类攻击点。为了找到更加微妙的问题，你必须做更加深入的评估。尽管如此，这些简单的方法足以给你一个绝妙的开始。

> **作者注：** STRIDE 是一种记忆术策略，用以帮助人们记住不同类型的安全威胁：哄骗（Spoofing）、篡改（Tampering）、抵赖（Repudiation）、信息泄露（Information disclosure）、拒绝服务（Denial of service）和提升特权（Elevation of privilege）。《Writing Secure Code》一书对此有详尽的论述。

接下去，你要去评估每个软件弱点的风险。一个允许黑客发现我们如何给美工图设置关键字的弱点，没有比允许黑客发现客户私有数据的弱点来得危急。而一个只需你从网站上拷贝一份攻击脚本并只需修改其中几个值就能形成攻击的弱点，比必须具体情况具体分析、需要理解代码细节并且必须编写低级语言程序才能形成攻击的弱点发生的概率要更高。攻击造成的影响越大，风险就越高；被攻击的概率越低，风险也就越低。将危险程度除以概率，你就能做出一份“风险评估”。（《Writing Secure Code》一书的表 2-2 详细描述了这个过程。）

安全受制于最薄弱环节

安全风险的计算听起来已经够简单了，但如何在一个像 Windows 这样大型的产品中确定可接受的风险呢？首先，你的团队必须在危险程度和概率等级的定义上取得共识。通常建议它们的取值范围为 1～10。危险程度为 10 表示高度危急，而概率为 10 表示极不可能。

接下去，根据你的部门对重要性的认同感确定这些取值范围。对“产品目录数据库”的只读权限的危险程度评级为 2 还是 8？对“用户数据访问”的只读权限危险程度评级又怎样呢？如果你必须编写定制的 COM 代码才能攻击一个弱点，那么它的概率等级该设成 4 还是 9？如果攻击的代码只需用 VBScript 来编写又怎样呢？（在《Writing Secure Code》一书的表 14-1 中，你可以找到一个对接口进行风险评估的非常有帮助的标准。）

> **作者注：** 这段时间，微软的安全专家已经在 Team Foundation Server（TFS）中编写客户模板，这些模板可能自动将安全 Bug 的弱点类型映射到相应级别及危险值。这些客户 TFS 模板在大型的团队内部使用，用以使 Bug 跟踪常规化及简单化。

当你的部门对危险程度和概率做出标准的定义之后，一个高层次的分诊或者“战争”团队，就可以在整个组织内部设置平衡的风险门槛了。我们已经把这个过程用于解决一个大型产品中不同组件团队之间的差异，也用于设置一致的质量门槛。同样的过程可以（而且应该）应用在安全问题上。

作者注：为了把低层次的产品和功能分诊和高层次的产品线分诊区分开，我们内部使用了很多有趣的名字：战争房间、盒子分诊、超级分诊等。我个人不大愿意使用“战争”这个字眼。

领导、跟随或者离开

你准备好应对安全方面的挑战了吗？你是否讨厌被一群傲慢自私的小子牵着鼻子团团转？

也许你在想，这些亵渎法律的失足青年应该被隔离起来，使他们远离网络，而那些煽情的记者和编辑们也不应该大惊小怪，应该加强庄重、公平和责任感，不要把黑客当做英雄，不要对Windows比对其他所有的平台和系统进行更多的诽谤。也许你在想，如果这些黑客攻击事件都被正确看待了，我们可能就不必总是去应对这种“消防演练”了。遗憾的是，这就是我们赖以生存的世界，公平抑或不公平，真实抑或虚伪。

让我们面对它吧！没人喜欢被攻击，也很少有人会去称赞黑客的创造力及其提供的公众服务。但是让虚伪、自命不凡、下流的人渣控制我们客户的心智和电脑，这彻彻底底是我们的耻辱！

当然，Linux、Oracle、Sun、IBM和AOL跟我们一样，也有很多安全问题。但正如BrianV所说的那样：“我们是这个行业的领头羊，我们必须去领导！”这个承诺言简意赅，少一点都是不可接受的。

作者注：Brian Valentine（BrianV）近10年来一直是Windows部门的高级副总裁。安全问题已经融入到产品及服务中。这种做法我们已普遍接受、了解并力求充分理解。

2002年11月1日：“牛肉在哪里？为什么我们要质量”

本月的《Interface》杂志关注我们从在公司范围内推行的“安全倡议”中学到的经验和教训。我们没有学到的又有哪些呢？我们做得又怎么样呢？

作者注：《Interface》是微软的内部网络月刊杂志，我的首篇专栏就发表在这上面。这份网络杂志的最后一期发表于2003年2月。

安全方面的“消防演练”不仅暴露的是我们软件中的安全漏洞，而且更暴露了我们工作中的“豆腐渣”，以致很多人都想知道下一个“消防演练”将会是什么。他们的猜测包括：隐私、实用性和保障性等。质量怎么样呢？有人听说过质量吗？质量到底出了什么事？

对于大部分开发团队来说，源代码签入时都很随意，一个个都像是业余爱好者。第一次传入代码库的就是像垃圾一样的东西，太可悲了！而以后对它的修复，就像是签署我们自己的死亡证明一样。为什么这么说呢？因为在产品发布之前，我们不会修复那些太肤浅、太复杂或者太含糊的问题，也不会修复那些发现得太晚或者直到我们出货之后才发现的问题。

什么？本来就是这样？我们像这样发布产品已经很多年了。今年和以往有什么不一样吗？我的上帝，要从哪里开始……

作者注：我在9年前写的这些东西很多都发生了变化。这并不是说我们现在已经功德圆满了。我们纠正问题的时机仍然太晚。然而，我们从根本上增加了单元测试、自动化测试、代码复审和代码分析的数量，这些操作在源代码签入进主代码树之前和产品发布之前都会执行。现如今，我们实际上还可以通过发布测试版来评估关键任务和日常工作。

译者注：2002年1～3月，微软在比尔·盖茨的亲自推动下，在开发团队中开展了名为“Security Push”的安全倡议。在此期间，Windows上的所有功能开发都暂停了，为的就是让开发团队能够花时间去分析产品的设计、代码、测试计划和文档上面的安全问题。通过对攻击方式进行研究，得出了180种威胁类型，并对如何防范进行了深入研究，对150种设计进行了修改。通过这项措施，微软将系统遭受攻击的可能性降到了最低程度。

情况变了

首先要说的是，如今我们产品卖向的市场要求“一键式”的解决方案，也就是说，只要你简单地按一下某个键，产品就开始工作了。之所以这些市场要求“一键式”方案，是因为如果相关的客户碰到了问题，他们没有足够的专业技能去绕开问题。因此，如果产品不能马上工作，我们就必须马上修复。

我们已经进入了两个主要的“一键式”市场：消费者产品和企业。如果你反应很快，你现在可能想知道我们的竞争对手是如何在这些市场上获得成功的。

在消费者市场，我们的竞争对手把他们的产品做得简单小巧。那样的话，就没有太多的故障模式；如果确实出了故障，他们的产品能够很快重启和恢复。我们卖的产品要复杂得多，功能也多得多。然而，这也意味着我们有更多的故障模式。为了保持竞争，我们的产品必须做得更好：更少的故障以及更快的重启和恢复能力。

在企业市场，我们的竞争对手提供大量训练有素的支持人员和顾问队伍。对于很多竞争对手来说，这是他们业务运转中最大的一块。实际上，他们靠产品的复杂度和故障来赚钱。当他们的产品崩溃了，我们的竞争对手会立即派遣他们的“骑兵中队”去修复问题，确保企业客户的系统能够正常运行。

我们不采用这种业务模式。我们大批量地卖产品，其单价没那么贵；我们为了将利润最大化，只提供最小限度的支持。然而，这意味着我们不能承受产品经常出故障，而且一旦发生故障，我们必须迅速修复或者恢复。

作者注：因为互联网提供了新的支持模式，所以我们“最小程度”的支持已经得到了极大扩展。不过，微软仍然是一个大批量软件和服务的提供商。

足够好还不行

其次要说的情况变化是，作为一个公司，我们的关键产品现在已经足够好了。事实上，从功能上来说，我们的关键产品（Office和Windows）已经足够好有几年了。

足够好意味着我们已经提供了客户需要的所有功能，或者至少他们认为需要的那些功能。但

这从以下两方面伤害到了我们：

- 人们停止将产品升级到下一个版本。毕竟，当前的版本已经有了他们认为需要的所有东西，并且升级常常是痛苦而昂贵的。
- 任何软件模仿者可以通过抄袭我们广为散发的、在线的、完全自动化的规范（产品本身就是规范），开发出具有竞争力的产品，以维持他们的生计。如果模仿者做得更好，他们把软件做得更加可靠、更加小巧、更加便宜（就像本田汽车曾经对克莱斯勒所做的那样），那么我们的麻烦就大了。

你认为这不可能发生吗？其实这已经发生了。Linux 不是已经敲响了警钟吗！Linux 并没有模仿 Windows。它只是确保拥有 Windows Server 系统所有足够好的功能。如今，还有些开发者在为 Linux 开发像 Windows 那样的外壳，还有像 Office 那样的应用软件。即使他们失败了，你能保证将来某一天不会有人成功开发出更好的产品——只要我们还敞开着质量大门？

我们玩不起追赶我们的潜在竞争对手的游戏。底特律为此奋战了多年，最终难逃失败。我们必须在其他人赶上来之前，把我们的产品做得让别人难以超越。

好消息是，我们还有时间。其他足够大到可以复制 Office 或 Windows 的商业软件公司其运营不佳，在项目管理及测试领域远远落后于我们。而开源团体则缺乏我们的战略性眼光及协同能力。他们依赖于一种强制性的方法期望最终实现他们的目标。如果我们提高我们的质量标准——保证更少的错误、更快的重启及恢复速度，并且只要我们专注于主要用户的需求，我们就可以击败所有的对手。

译者注：底特律（Detroit）位于美国密歇根州东南部的底特律河畔，与加拿大安大略省的温莎隔河相望，是世界上最大的汽车工业中心，号称“世界汽车之都”。“汽车大王”亨利·福特 1903 年创建了第一家大规模的汽车生产厂。作为美国三大汽车公司——通用、福特和克莱斯勒的大本营，底特律早在 100 年前就开始成为美国汽车的同义词，底特律的汽车工业也成为美国经济的一大动脉。通用汽车目前还是美国市场的老大，但随着近几年亚洲汽车厂家在北美的扩张，其市场占有率正逐步下降。这些亚洲对手包括丰田汽车、本田汽车、日产汽车和新近崛起的现代汽车。通用汽车在美国市场的占有率从巅峰时期的 50% 下降到 2005 年的 26.2%，福特汽车则下降到 18.6%。而丰田汽车同期在美国市场的占有率则为 13.7%，高于上年的 12.2%。

艰难的选择

不过谁都知道，没有东西生来就是免费的。如果我们更多地关注质量，其他的什么东西就必然要牺牲掉。在一个较高层次，我们可以控制的变数只有质量、日期和功能。对于有着固定期限的项目来说，追求高质量意味着需要减少功能。而对于有着固定功能的项目来说，追求高质量意味着要延后交付日期。

> **作者注：**实际上，我已经不再完全相信这套理论了。我看到了因减少系统浪费（你可以阅读第2章“过程改进，没有灵丹妙药”的一个栏目“精益：比五香熏牛肉还好”）和早期修复问题而获得的极大效率。尽管它争取到的时间不足以给全公司放暑假，但我相信，它足以在我们追求高质量的同时不牺牲产品的功能或交付日期。是的，要把事情从一开始就做对的话，不可能像以前质量门槛很低的时候一样快，但跟我们最近漫长的稳定化周期相比，如果不说更快，它至少也是一样快的。

在你的思考过程遭遇障碍之前，比尔·盖茨在他的一篇关于“可信计算”的文章中，早已经告诉了我们该如何选择：

> 在过去，我们通过增加新的特性和功能以及使平台富有扩展性来保持软件和服务更加引人注目。我们在那方面做了大量的工作，但如果客户不信任我们的软件，那么所有那些出色的功能都没有任何意义。

唯一的问题是，你打算坚持到底吗？

这里有3个基本方面值得关注，可以用来提高产品的质量：

- 更好的设计和代码。
- 更好的源代码插桩和测试。
- 更好的保障性和恢复。

让我们把它们拆分，一个一个讲述。

终于有足够的时间了

很少有开发者不喜欢有更多的时间去仔细思考他们的代码，然后从一开始就把代码编写正确。麻烦在于，去哪里找时间，并且开发者如何能够自律而精明地使用那段时间。不过现在假设你有更多的时间，你会做什么呢？作为一名管理者，我会花更多的时间跟我的团队成员在一起，讨论设计决议，进行代码复审。

我要强调的两个关键设计问题是：简单和适度分解。

- **简单。**保持设计简单并且重点突出是减少意外结果和复杂故障的关键。
- **适度分解。**这有助于让设计的每个片断都保持简单，并且彼此之间相互分离。它同时也有助于加强数据和操作的独立性，这更易于对代码进行维护和升级。

> **作者注：**在实现设计上，测试驱动开发（Test-Driven Development，TDD）达到了上述两种效果。你可以对组件设计也采用类似于测试驱动开发的方法，尽管测试有时候只是在脑子里面开展的试验。

我也会给开发人员额外的时间，让他们在每个功能任务上都结对工作，目的是：

- 使每个开发人员工作所需的时间翻一番，因为你在制定时间表的时候，每个任务花费的时间是按照单个开发人员来计算的。
- 允许对设计和代码进行同级评审。

- 为每个功能设置一个后备开发人员，为主要开发人员的离职做好准备。

为了帮助开发人员自律，我会：

- 确定“开发规范书”（也称为“设计文档”或“架构文档”）的完成日期。
- 对于功能的质量，让每个后备开发人员与主要开发者承担同等的责任。
- 度量回归率，并且把创建检查故障数作为源代码签入的质量反馈。（当然，这些度量并不完美，但你想要什么呢？每行代码的 Bug 数吗？）

> **作者注：**如今，我已经不再使用回归率，而改用代码搅动和复杂性度量。阅读本章的“对质量的大胆预测”直到结尾了解更多内容。

再检查一遍

你永远也无法保证自己写的代码已经正确无误了。你必须知道这一点。在代码内部使用插桩进行检查，而在外部使用单元测试进行检查。在覆盖率和深度方面，你做的检查越多，你就能越好地保证质量。

注意，这绝对不是测试者的工作。测试的职责是保护客户免受任何你遗漏的东西所带来的伤害，而不是盯着你做得最好的东西。测试团队不是你的拐杖，你不能拿它去支撑满是污秽（代码）的双脚的平衡。如果测试发现了一个 Bug，你应该感到局促不安。他们发现的任何 Bug 都是你遗漏的 Bug，而且最好不是因为你的懒惰、冷漠或考虑不周全引起的。

那么，你要怎么做才能阻止 Bug 流传到测试者和客户那里呢？

- **源代码插桩。**断言、Watson、测试用具钩子、日志、跟踪、数据验证等，所有这些都是在源代码签入之前和之后发现和查明问题的无价之宝（哪怕只是有一点帮助）。
- **单元测试。**测试经常揭示出在优秀与杰出之间、在几乎不能正常工作与坚实可靠之间的最大差异。如今，各种各样不同的单元测试有很多。你、你的后备人员和你的测试同伴应该挑选最适合你的功能的那些测试：
  - “正向单元测试”按照设计意图测试代码，并且检查正确的结果。
  - “逆向单元测试”故意误用代码，以此来检验健壮性和适当的错误处理。
  - “压力测试”把代码推向极限，以期撕开并暴露敏感资源、时间竞争或重入错误。
  - “故障注入测试”暴露错误处理的异常。

你验证的越多，你覆盖的代码路径越多，客户在你的工作上发现故障的可能性就越小。

医生，治好你自己的病

即使你设计做得很好，功能的编码实现也做得很好，给源代码插桩并且测试都做得很好，Bug 仍然会存在。通常那些 Bug 源自于跟其他软件复杂的交互。有时候那些软件不是我们的；有时候因为那些软件很老，或者它们根本就没有被好好设计或测试过。

但纵然 Bug 总是存在，这不能成为你不尽力消除 Bug 的理由，也不能用于在出现 Bug 时帮你逃避责任。你仍然必须帮助客户修复这些 Bug，或者从问题中恢复，并且最好在他们察觉到之前就做好。

一个很好的关于恢复的例子是 IIS 进程循环技术。任何时候，只要有一个 IIS 服务器组件出了

问题，IIS 进程会立即自动重启。在最坏的情况下，客户也只是感到网络突然“打嗝”了一下，而通过一次简单的刷新就能修复。

> **译者注：** IIS 全称为 Internet Information Service（互联网信息服务），是微软公司主推的 Web 服务器，功能包括提供如 HTTP、FTP 等信息服务。

Office XP 也有一个恢复系统，尽管跟 IIS 相比，用户有更多的介入。当一个 Office 应用程序失败了，它会对数据进行备份，报告问题，然后自动重启并询问用户是否进行数据恢复。这些解决方案并不是很复杂，但它们给客户带来的利益是巨大的，并且在客户支持的成本方面节省了大量的资金。

> **作者注：** 本章后面的“碰撞测试：恢复”将对可恢复性有更深入的讨论。

如果你无法让你的产品自动恢复和重启，你至少应该捕捉足够的信息去鉴别和重现问题。那样的话，技术支持工程师就可以很容易、很快地了解问题，并且如果有现成的修复方案就可以马上派上用场。如果一个问题还没有相应的修复方案，那可以把错误信息捕捉下来送到你的团队，然后你就可以把问题重现出来，在第一时间为客户设计出一个相应的修复方案。

捕捉足够的信息用以鉴别和重现问题并不像它听起来那么难：

- Watson 目前在鉴别问题方面做得相当出色。Watson 的未来版本将使客户问题在微软公司内部的重现变得更加容易。
- SQL Server 在捕捉各种各样的客户数据方面做得相当出色。在有错误发生时，它能做到从“精确重现对一个数据库的所有改动”到“简单地输出相关状态”的任何事情。

步步为营

好的，假设你采纳了所有的这些建议。但它值得吗？代码真的变得更好了吗？你曾经遗漏了什么呢？如果一定要说这些“安全倡议”教会了我们什么，那就是：问题不会总是那么显而易见，特殊方法和自动化工具不能找出我们所有的问题。

这些倡议也提醒我们，几个疏忽的问题可能造成微软和我们的客户成百万美元的损失，同时导致我们的客户对产品质量的满意度呈两位数字的下降。

我们能从安全专家那里借鉴到什么技术呢？下面两个立即出现在我脑子里面：

- 把你的产品分解成不同的组件，就像你为风险建模时所做的那样。然后在各个组件内寻找质量问题。自问一下：我们应该怎样来设计每个组件？我们怎样才能给每个组件插桩，使它们可测试、有保障、有恢复能力？请应用更加结构化的工程方法去提高质量，以得到更加可靠、全面的结果。
- 缩小你的产品的故障范围。也就是说，减少产品被误用的途径。抛弃那些允许客户设计意外流程的选项，不要因为你想象中“某人在某个场合下可能想要那个选项”而把它加入到你的产品中。简化每个功能，以使它表现得跟你当初的设计一致，这样就足够了。记住，不必要的复杂只会隐藏 Bug！

太多疑问

到如今，你肯定认为我精神不正常。要你的项目经理、测试和实施团队（姑且不管你的管理层）给你时间去做我建议的所有事情，这肯定不可能。实际上，事情没有你认为的那么糟糕。在这些实践中，很多都是相互关联的：

- 适当设计受欢迎的产品可测试性和保障性。
- 源代码插桩有助于鉴别和重现问题。
- 测试揭露出你必须进行修复的弱点。

为了获取更多的时间把事情做对，应该让提高质量变成整个团队的共识。把保障性和恢复能力当做功能需求来定义。并且让项目经理、测试和实施人员也加入到这个运动中来。

当我们把产品做对了，客户从我们的工作成果上重新获得了信心，我们将绝尘而去，把竞争对手远远地甩在身后。他们无法模仿我们的功能。他们无法抄袭我们的创意和前向思维。如果我们展示了我们是多么杰出的一帮工程师，他们将没有能力再跟我们的质量竞争。

> **作者注：**“牛肉在哪里”是我最喜欢的栏目之一。多年后的今天，当我重读这个栏目的时候，我依然为之振奋。

2004 年 4 月 1 日：“软件发展之路——从手工艺到工程”

该来掂量一下那个争论了多年的开发问题了。不对，不是要把花括号放在哪里的问题（它们呆在自己的代码行上）。争论是，“到底什么叫开发者？”我们是像设计师和艺术家一样需要时间去思考和想象的、具有创造性的那种人吗？开发是手工艺，而我们是工匠吗？或者，正如我们的头衔所示，我们是“软件工程师”吗？这最后一条确实让人愤怒。把它忘掉吧，因为问题已经有了答案。

哦，仍然有很多人认为这事没完，争论还要继续下去。我已经接纳了“开发者”的称呼，看着网站的内容，听着他们的争论。见鬼，我已经表达了我的观点，我曾经强烈声称“我们是开发者，不是工程师”。（参见第 1 章的“竭尽所能：再论开发时间表”栏目。）

开发软件是一个创新的过程。这是一个无法预测的过程，需要处理各种自定义的、难以理解的组件。很多人坚信，在我们的有生之年根本就没希望在这个过程中应用工程实践。他们认为，我们往差了说都是些苦工、牛仔和莽撞的外行，往好了说也就是些有创意的工匠。好吧，但把软件开发当做一种手工艺已经不再恰当了。

工艺制桌子，工程造汽车

不要误解我的意思。我热爱手工艺。没有什么东西能比得上耐用的手工制作的桌子和椅子、雅致的手工制作的座钟，甚至于设计精良、手工搭建的房子——一个抚育你家庭的温馨的家。我只是不想开着我的汽车通过一座手工建造的桥梁。我不想有人在我的胸腔上粘贴手工制作的起搏器。我不想依赖手工制作的软件来运营我的业务、保护我的资产或者指导我的行动。我想要很好

的工程化制造的软件来完成这些任务。我们的客户也是这样。

那么，程序操作员和工匠有什么区别呢？工匠和工程师又有什么区别呢？程序操作员边做边学，先做后想，当有人插手帮助时他扔掉自己所有“乱糟糟的东西”（不以自己的工作为荣）。听起来很熟悉吧？与之相比，工匠做研究，做计划，使用最佳实践和工具，并且为他们的工作感到骄傲。这最好地描述了作为软件开发者的我们。但手工艺没有完全达到工程状态，因为你仍然不知道你将会得到什么。手工艺缺乏确定性和可预测性。你只能尽力做出最佳估计，而不是真实知道。

其实你知道

另一方面，工程本质上就是要“真实知道”，而不是依赖猜测。工程就是度量、预测、控制和优化。工程师不会随意猜忌某事，他会认真对待。工程师不随意估计，他认真计算。工程师不会无畏期待，他会实事求是。这并不意味着工程缺乏创造力或创新意识，只是工程必须强化安全行为的已知边界，以取得可靠的结果。

但是我们都知道，软件是不可预测的。我们怎么可能把结构化的工程实践应用到软件上面呢？其中的奥妙其实很显然，我只恨自己知道得太晚了。不要试图预测软件，而要预测开发软件的人。在软件开发中不变的不是软件，而是人——那些开发者。人是习性动物，而我们的习性是可预测的。这种“觉悟”可能听起来并不深刻，但它改变了一切。

真实面对自己

说你是可预测的可能会伤害到你，但你确实是这样。你稍加反省就将揭示这个真相。你一次又一次地犯同样的错误。你每次写某种类型的函数或对象，花费的时间大致都是相同的。你甚至使用等量的代码去实现它们。这很令人恐慌，但却是真的。更重要的是，你变得可度量、可预测。神奇的鬼东西！

好吧，我承认，我一开始也不相信这些，不过我随后花了几周时间去自我度量，编写程序并且把结果图形化展示出来。跟其他 4 500 个早于我尝试的程序员一样，我很坦率。对于任何给定类型的函数或者类，我大概花费了相同数量的时间，写了一样多的代码行，犯了一样多的同类错误，花了一样多的时间去修复它们（指修复相同类型的错误）。这个发现是令人难堪的，但却很有说服力！

> **作者注：**我在自我度量上花的两周时间，其实是软件工程学院（Software Engineering Institute，SEI）的“个人软件过程”（Personal Software Process，PSP）课程的一部分。个人软件过程是一种用于软件开发的称为“团队软件过程”（Team Software Process，TSP）的工程团队方法的一部分。我对他们演示的结论以及背后的理论印象非常深刻。我的团队也尝试了一段时间的团队软件过程。我很快就会谈到当初我们的进展情况。

如果你刚好为你需要编写的类画了一张图表，就可以很有信心地预测这个任务将花费多长时间，将编写多少行代码，将引入多少 Bug 以及这些 Bug 的类型。你也可以知道会有多少 Bug 分别在设计复审、代码复审、编译、单元测试和团队测试中浮现出来。

数字的含义

那又怎么样呢？它的意义在于，如果你知道你要找到多少 Bug，你就能知道是否要继续寻找，你就能知道到什么时候你才算找了足够多，你就能知道其他人应该找多少，以及他们应该找什么。于是，你可以很有信心地说，“我们已经找到并修复了 99.999 9% 的 Bug。”就这么简单！换句话说，你能真实地知道，而不是靠猜测。恭喜你，你现在是一名工程师了！

好吧，代价是什么呢？为了能做出很有信心的预测，有多少麻烦事要去跟踪呢？下面是你必须收集的度量清单：

- **花在两个检查点之间的时间。**这是你在每两个检查点之间的实际工作花费的时间长度。（检查点可以是：设计完成、设计复审完成、编码完成、代码复审完成、代码构建无错误、单元测试完全通过、代码签入等。）

> **作者注：**如果使用看板，你也应该用这种方式，这些数据的收集就比较琐碎了。你可以通过使用一种累积流量图（一种进度图表）直接从看板表上推演出这些数据。每种过程在看板库中所花的时间就变得显而易见。

- **在以前检查点上所花的时间和返工量。**这是你在先前的检查点已经“完成”的东西上面返工花费的时间，同时使用一行描述来说明发生了什么事情并加以归类，便于以后你能引用到它。（典型情况下，这里指像设计完成之后的设计变更、修复 Bug 而进行的代码改动或者其他类似的任务。）

> **作者注：**如果你单独地从线性流量图中追踪返工量，那么从看板中收集这些数据也是非常简单的。Corey Ladas 在他的文章“计算 Bug 量并返工”中有详尽描述。（LeanSoftwareEngineering.com）

- **你增加、删除或改动的代码行数。**这一条很显然，而且很容易通过自动化工具做到。

就这些了。所有的信息都可以通过一个规范的定时器和记事本去获取，尽管有的团队正在开发工具让事情变得更简单。为了得到精确的结论，你必须坚持不懈。不过，就是这些采样点数据，能让你最终得到梦寐以求的信息。

具体来说，你可以回答这样的一些问题，“这周我花在实际工作上的时间有多少？”“应用程序编程接口被我们修改了多少次，花了多少时间？”“代码复审发现的 Bug 所占的百分比是多少？”“代码复审发现 Bug 的百分比与花在代码复审上的时间有什么关系？”“早期发现的 Bug 主要是什么类型的，晚期发现的又是什么类型？”“修复哪种 Bug 花费的时间最多，它们是什么时候引入的，又是什么时候被发现的？”

各人有各人的习性

那么，得到的是什么呢？数据只对当事人一个人有意义。每个人的习性都不同，你不能把我的数据跟你的相比较，讨论也没有意义。这其实是好事，因为不管怎么样，管理者不应该拿这些数据用于比较。正如我在“不只是数字”一文中所说的那样（参见第 9 章的“成为管理者，而不

是邪恶的化身”），当管理者拿大家的数据相互对比时，度量就会被亵渎。

尽管大家的个人数据不能用于共享或者相互比较，但它们可以聚合成为团队数据。这下子，管理者可以美梦成真了。不费吹灰之力，你可以在团队这个层次做所有的预测和质量管理，而且能够保证相当高的精确度。因为数据的聚合意味着平均化，团队结论的精确性不会低于个人结论。你可能管理100个人，不过你能预测完成日期和Bug数量，其精确度跟个人的预测处于同一水平。太酷了！

大处着想，小处着手

好吧，这里的关键是什么呢？也许工程化做软件在某程度上来说是可能的。也许只要你全身抖擞一下，你就能预测代码大小、Bug数量、开发时间等。接下去，怎样把数据转换成结论呢？微软内部和外部的一些团队，他们像这样使用数据去监控和控制他们的Bug数量，结果是可喜的：他们的缺陷比率从早先的每千行代码40～100个Bug的水平，降到了每百万行代码只有20～60个Bug。换句话说，一个典型的有15～20人的微软开发团队，他们以往每年产生3 000～5 000个Bug（这些Bug通过测试发现），现在降到了每年只有3～5个。

是的，那些低Bug率已经包括了集成时发生的Bug。人们争辩和抱怨：在我们的大型软件系统中发现的Bug都是何等复杂啊！这是事实，你手上的Bug很大一部分都是在进行集成时发现的。但既然这些Bug被找出来了，那么到底是什么类型的Bug呢？它们是在诡异、难以预测的多线程交互中的极其复杂的时间竞争问题吗？也许其中1～2个确实是，但剩下的成千上万的Bug都是弱智琐碎的参数混淆、语法错误、忘了检查返回值，甚至更加普遍的设计错误，而这些设计错误本该在开始编写任何一行代码之前就被揪出来的。

从优秀到卓越

当然，卓越团队通过使用设计复审、代码复审、类似于PREfast的工具和单元测试来控制他们的Bug数量。然而，单独使用这些方法仅仅让开发者变成了工匠，Bug数量也只能大概降低至原来的十分之一，而不是千分之一。这个下降没那么大，是因为你必须猜测什么地方以及怎么应用你的工艺，而不是确切地知道。通过投入少数的一点度量，并且善加利用你的个人习性，你就能确切地知道了。于是你晋升为一名工程师，并且可以得到上千倍的提高！

那是一个很大的进步，也是一个必要的步骤，以保证交付给客户要求的质量和可靠性。你也必须辨别需求，并且创建一个具体的、满足需求的设计，不过那些有趣的话题只有等到下次再说了。眼下，把你的Bug数量降到每年10个左右将是个不错的开始！

作者注：那么，我的团队当初对于“团队软件过程”（Team Software Process，TSP）的试验情况如何呢？我们是否也在Bug（返工）方面达到了1000倍的下降呢？不完全是这样。公正地说，我的团队不是典型的那种团队，我们也没有在团队软件过程中坚持足够长的时间以得到可靠的结果。问题不在方法本身，尽管它偶尔有点不必要的沉重和呆板。真正的问题在于工具——它们不好用，而且不稳定，也不适合大型团队。

从那以后，我们将所有精力放在容易应用的过程改进上，得到的结果也非常不同。这包括Scrum、TDD、计划扑克、Delphi法估算、审查（设计与代码复审的基本方法）、单元测试、代码分析，以及最近开始采用的看板法及注重案例的工程方法。

2005 年 7 月 1 日：“复审一下这个——审查”

当你被邀请参加一个规范书复审会议时，立刻出现在你脑子里的想法是什么？我猜是，“是祸躲不过”，或者“好吧，该看看事情有多糟糕了”。也许你只是无奈地叹口气，就像马戏团里追着大象搞清理的那个人一样。

当你参加一个代码复审时，情况又怎样呢？曾经感到心神不安吗——像是你离家在外已经一周了，但你却记不清是否所有的门都锁了，所有的灯都关了？

在设计复审时，你是不是只能傻坐在那里，任凭另外两个人就一些问题争论来争论去，而你的问题根本没人听？甚至更糟糕的是，居然一半人都没有预先阅读文档。这也许是件好事，因为作者还没有对他设计背后的“瑞士奶酪”理由进行最起码的拼写检查。

> **译者注：**瑞士奶酪在美国称为 Swiss Cheese。这类奶酪最明显的特点就是表面的孔非常多。作者在这里用瑞士奶酪暗指设计文档存在很多漏洞。

好吧，女士们、先生们，你们有麻烦了。很不幸，你的团队患上了一种通病——“对设计不能有效地做到集思广益”。你就该倒霉了！

糟糕的混合

如果你不能区分以下 3 类活动，你就会陷入混乱中：

- 提出想法和解决方案。
- 对进展中的工作收集反馈。
- 对已经完成的工作进行质量评估和问题挖掘。

大部分团队将上述 3 种活动混合在一起，统称为“复审会议”。甚至有些人可能称为“茶话会”。

很自然，这是一种糟糕的实践。如果把 3 个不同的目标混合在同一个会议中，那么每个活动都必将失败，而且你还会让参与者感到失望和困惑，会议组织者收到的信息也是错综复杂的。

如何才能避免这类事情的发生呢？搞清楚你的目标是什么，然后选用最合适、最有效的方法去实现它。这里我推荐的方法有：头脑风暴、非正式复审和审查（Inspection）。

完美风暴

当你想要收集想法和解决方案时，可以使用头脑风暴。你们可以分成多个小组进行，也可以在一块白板前面一对一地进行。当有多于 4 ~ 5 人一起参加时，可以使用归类技巧：给每个人都分配一些便笺贴纸，让他们尽可能多地写下各自的想法，然后将这些便笺按照公共的主题归类。

不管你如何进行头脑风暴，目标都是尽可能多地收集想法。大家提出的建议没有好坏之分。因为最好的解决方案，可能就来源于很多最初看起来不切实际的提议中的某些部分。

头脑风暴非常适合在设计和撰写规范书过程的早期使用，因为这时候你还没有写下太多的东西。把关键的项目关系人和功能团队成员聚集在一起，“榨干”他们的想法！

如果你难以选出一个解决方案，那可以使用“普氏概念选择法”来决定。这种方法使用一张表格来给各个方案针对各个独立的需求评分。分数可以是正的、负的或者零，这完全取决于一个方案对于某个需求的适合度。然后基于重要性，将各个评分乘上一个加权系数。最后总分最高的解决方案胜出。

> **在线资料：**普氏概念选择（Pugh Concept Selection）（PughConceptSelectionExample. xls，PughConceptSelectionTemplate. xlt）。

此外，还有更好的做法，那就是不要选择任何解决方案。保留每一个设计思想，直到发现某个使之不可行的约束或因素为止。最终你手上只会剩下一个方案能够最好地满足所有需求。丰田汽车公司的那伙人称这种方法为“集合设计”（Set-Based Design）。

谁来负责

千万不要把头脑风暴跟“集体设计”混淆在一起。尽管可能很多人都做出了贡献，讨论了他们不同的想法，但设计应该只有一个所有人。这个所有人拥有最后拍板的权利，也只有他的名字才会出现在设计文档上。

只设一个所有人保证了设计的清晰度和一致性。它赋予了所有人责任意识去满足需求，造就了一个理解和贯彻设计抉择精神的勇士。

与之相反，集体设计是专门给懦弱无能的部门准备的——当他们怯懦的多数人意见在极小的张力下崩溃时，他们可以把指责转嫁给别人。

你有什么想法

有时候，你只需要知道你是否在朝着一个正确的方向前进。“单页规范书”复审就是为了达到这个目的。通过 E-mail 或在团队会议上进行的非正式同级评审也有类似的效果。对“进展中的工作”进行的复审起到检查点的作用，以便减少早期的天真或微妙错误引起的工作浪费。

问题在于，人们常把这种类型的复审用在问题挖掘上。非正式复审跟其他方法相比，在问题挖掘方面毫无成功的希望。然而，几乎所有工程部门的规范书、设计、代码复审都采用了“非正式复审”。

试图使用非正式复审来挖掘问题（比如随意说上一句，“嗨，在我签入代码之前，帮我看看这个 Bug 的修复吧！”），会把你、你的团队和我们的产品推向失败的边缘。你得到的只是毫无准备的复审者漫不经心的复审。你的团队一直有这样的感觉：“这些复审真的有用吗？我总觉得我们遗漏了什么东西。”于是，你的产品到处都是 Bug，最终你还得回过头去修复它们——但愿是在产品发布之前。

团队可能不再抱有幻想，或者认为他们必须更加认真地进行复审。可惜的是，非正式复审是用来收集反馈的。它们从来就不是质量控制或质量保证的有效手段。没理由在整个团队中兴师动众，严厉指责他们没有读文档或代码，然后艰难地举行一个低效率的会议，其根本原因还在于不该用这个非正式复审。

如果你需要反馈，那就请求他们提供反馈。收集反馈要赶早。让过程有趣一点、随意一点，

然后谢谢他们的帮助。“过场”（引导部门的同事一起走读你的设计或代码）和非正式复审都是很好的能用来做这件事情的方法。如果你为了质量控制和质量保证需要挖掘问题，那你必须使用审查。

正是这个形式

“审查”就是在已完成的工作上面进行问题挖掘。就这么简单！这种方法不适合于提出想法和解决方案，也不适合于收集反馈。不过，如果你想在产品发布之前找到你的设计和代码中的所有 Bug，你就需要审查。

审查可以列出所有发现的问题，并且对所有尚未发现的问题数给出一个精确的估计。秘密藏在形式里面。审查不会比非正式复审的整个过程花费更多的时间，但它的正式流程规避了有人会不负责的问题。

> **作者注：**我在这里介绍的正式审查，其具体方法源自于软件工程学院的“团队软件过程”（Team Software Process，TSP）。

下面是关于这个流程的简单概要：

- **计划。**确保工作处于完成状态，然后才能安排会议。
- **概览。**给所有审查员（也就是对你的工作进行复审的人）足够的背景知识去理解你的工作，还有一个问题类型的检查清单和工作本身的一份拷贝。
- **准备。**告诉审查员每个人独自列出他们发现的所有问题（基于那个问题类型检查清单）。
- **开会。**把所有审查员聚集在一起汇总问题清单，在重复的问题上面取得一致意见，决定哪些问题必须修复，同时决定那些问题被修复之后是否还要再进行一次审查。
- **返工。**解决所有必须修复的问题。当然，你也可以同时修复一些次要的问题。
- **跟踪。**了解你所修复问题的根源，更新你的检查清单以避免那些问题在将来再次发生。需要的话，重复上述审查过程。

你会在问题类型的检查清单上和问题清单汇总的时候发现审查的魔力。然而，在我们到达那里之前，工作必须已经处于完成状态。

孩子，准备好了吗

审查的要点是在已经完成的工作上面发现所有遗留的问题。如果工作还没有完成，到处都是缺口和低级错误，审查员就会陷入泥潭，对检查清单上更难的问题丧失重点，最后埋怨你拿这些不堪入目的东西浪费他们的时间。

在你安排会议之前，确信你已经有了一份清晰的规范书、设计文档或源代码。确信规范书和设计文档涵盖了所有你想检验的东西。确信源代码能够通过编译、建造，并且发挥正常的功能。这包括你在运行 PREfast 工具时没有遗留任何问题，并且达到了你部门最初设定的质量门槛的方方面面。理想情况下，在你安排一次审查之前，你团队中某个人应该被预先指派去检查所有这些事情是否都已经完成了。

确信工作已经完成的一个好方法是做一次个人审查。许多开发者都已经这么做了。他们通篇

阅读自己的代码或设计，尽力去发现问题。把经常犯的错误列在一份单页的检查清单上，你就能开展个人审查了。检查清单中包含你最常犯的错误方式。不过，每个人的检查清单不都是唯一的，因为每个人犯的不是一样类型的错误。打个比方说，我在资源清理方面做得很出色，但我经常改变参数的位置。在每次审查之后更新你的检查清单，删除你不轻易再犯的问题，当然也要增加你开始意识到的新问题。

再检查一遍

当你的工作完成之后，把它发给所有的审查员（比预定的审查会议时间提前几天）。附上足够的背景信息以帮助他们理解你的工作，比如一个描述和将要用到的检查清单。

团队检查清单像你的个人检查清单一样，应该只有一页。不过，团队检查清单应该从团队的角度出发，填写影响到你团队的问题类型，比如你团队开发的软件最容易犯的特定的安全、可靠性、逻辑或故障处理等问题。像你个人的检查清单一样，团队的检查清单也应该定期更新——删除那些不再普遍的问题（团队已经改进了），增加越来越引人注目的新问题。

审查员然后根据检查清单上的问题类型仔细、独立地审查你的工作，把发现的每个问题的类型和行号都记录下来。通常，审查员会发现一些不在检查清单上的问题。那些问题同样应该被标出来。需要花时间去调查那些很突出但检查清单遗漏的问题类型，以期找到最好的方法在未来排除它们（可能通过更新单元测试、自动化代码分析或审查检查清单）。

审查的方法多种多样。一种有效的方法是，对于每个问题类型都从头到尾检查一遍，按照检查清单上所列问题的次序逐个进行，一次只查一种类型。当然，这听起来很单调，但没有情境切换的干扰，这种方法显得容易而有效率。不管他们使用何种方法，经过一些实践之后，审查员会变得非常善于揪出绝大部分的问题。

通过使用相同的检查清单，审查员很容易就能汇总各自发现的问题，从而也能判断出大家发现的问题中有多少是一样的。

神奇的汇总会议

在审查会议上，发现问题最多的审查员首先把他的所有问题填写到一个电子表格中。通常情况下，需要预约一个对审查流程很有经验的主持人来处理数据录入，并且保证会议向着既定的目标前进。

在线资料：审查工作单（InspectionWorksheetExample. xls 与 InspectionWorksheetTemplate. xlt）。

对于每一个发现的问题，如果有其他人同样也发现了（问题类型和发生地都一样），那么审查组把这些人也标出来。在同一行上可能会发现多个问题。除了对重复发现的问题取得一致意见外，审查员同时也要注明这个问题是否必须修复，还是它可以留给作者本人去决定。

当第一个审查员汇报完他发现的所有问题之后，下一个审查员就开始同样的流程。跳过那些已经被第一个审查员标出的问题。这样一直持续到所有问题都录入电子表格。然后电子表格就会自动计算统计信息，比如缺陷密度、收益、总共发现的问题数、可能仍然潜藏在工作中的问题数等。

那怎么可能呢？因为所有人都在同一份工作上查找相同的问题，而你收集了每个人发现的问

题和遗漏的问题。因此，你能精确地估计出总共有多少问题被遗漏了。你就不会再不知所措了！

> **作者注**：这个数学算法基于古老的用于估计一个湖里有多少鱼的“捕获—再捕获”方法。你首先在湖的几个不同地方捕出 n 条鱼，给它们打上标签，然后把它们放回湖中。过一段时间，你再在相同的地方捕出一堆鱼，记录下其中打标签的鱼占总数的百分比。最后，你用 n 除以那个百分比，就可以很好地估计出这个湖里面鱼的总数了。拿我们的审查来说，第一个审查员发现的问题可以看成是“打过标签的鱼”，其他审查员发现的问题都是在同一个设计或代码“湖”中再次捕到的“鱼”。根据统计的结果，我们就能很容易而且精确估计出问题的总数。

审查的诀窍

很自然，你可以使用很多小窍门，以便最有效地使用审查：

- 尽管你可以跟一群人一起来做审查，但典型情况下，你只需要 2 ~ 3 个审查员就能收到良好的效果（不包括作者本人）。一开始可以邀请 3 ~ 5 个审查员以掌握审查的窍门，等到大家都很熟练的时候就可以把人数减少。
- 正如非正式复审一样，把审查做好也是需要花时间的。如果审查的是文档，准备每小时 3 ~ 5 页；如果审查的是代码，准备每小时 200 ~ 400 行。这是合理的速度。如果太快了你会遗漏问题，太慢了你脑子会不好使。
- 不要一下子审查太多。1 000 行全新的代码大概要花费 3 ~ 4 个小时去审查。温和一点，一块一块地审查。
- 不要在会议上讨论修复方案。审查是为了挖掘问题，不是为了收集想法。你不要自找麻烦！
- 邀请一个在填写电子表格、阻止大家讨论修复方案方面很有经验的主持人是很有帮助的。
- 也可以赋予主持人以职责去验证工作是否已经完成，是否可以开始审查了。在大家掌握审查的窍门之前（包括新加入你部门的同事），你绝对应该邀请一个主持人。
- 你的团队应该设置一个质量门槛，规定存在多少问题是可接受的，可以将东西交付给下游团队进行更为深入的测试。如果审查的结果显示剩下的必须修复的问题数太多了，作者必须在修复了这些已发现的问题之后发出“再次审查”的请求。这里有个不错的指导方针：任何收益（已经发现的问题数除以总共的问题数得到的比率）低于 75% 的工作需要安排再次审查。

走上正道

当恰当使用审查时，它可以在产品发布给客户之前极大地减少问题的数量。微软内部和外部的团队都已经反复证明了这个结果。关键是，你不要搞乱“集思广益”这个目标。

如果你想要收集想法或寻找解决方案，那就开一次头脑风暴会议，注意思想要开放，不要妄加批评。如果你想得到早期的反馈，召集一次非正式复审，给大家一些灵活的空间，并且对有些问题可能被遗漏做好心理准备。如果你想要评估质量和挖掘问题，那就组织一次审查，集中精力去找问题，但不要急于讨论修复方案。记住，你总是可以使用头脑风暴或非正式复审来帮助修复所挖掘出来的问题。

增加一点沟通和透明度可能大有帮助。如果你深谙什么时候应用什么方法来开展部门活动，你将远离糟乱和失败。我们都清楚，那将是多么有益的策略啊！

2006年10月1日："对质量的大胆预测"

最近我在忙着"吃狗食"。这是变成受虐狂的一条理想途径。不管它有多过分，我总可以在一部不错的恐怖电影中得以片刻的喘息。感谢上帝，经过几年的变化，如今投递过来的"狗食"已经有了极大的改善。

> **作者注：**"吃狗食"（Dogfooding）是在日常工作中使用未发布产品的一种实践。它鼓励团队从一开始就把产品做对。通过这种实践可以收集对产品价值和可用性的早期反馈。

几年前，运行一个"狗食"软件的话，其生产力跟拔掉你机器的电源没什么区别。如今，"狗食"软件实质部分的功能已经完全能正常发挥了，不过其他部分的功能仍然不可用、不可靠或者不合理。这引来一个问题，"为什么?"

为什么有些部分是可靠、贴切、迷人的，而其他部分保持着脆弱、费解、面目可憎？怎么会变成那样子的？问一下管理者，他们可能会说："嗯，想要预先判断出什么将变好或变坏是很难的。"听起来好像他们正在和着牛屎咽下我的狗食。

谜？我不这么认为

软件质量是不可预测的吗？不要让我作呕。质量糟糕的软件跟附近的冰淇淋车一样微妙。你知道冰激凌对你没好处，你知道它近在咫尺，然而你却无法抗拒。管理者选择忽略警告符号，买来冰激凌（糟糕的软件），因为他们不想让孩子（上层管理者）失望，也无法抗拒即刻的满足（所谓的工作进展）。

我们对糟糕的软件已经如此习以为常，以致很多人都已经对那些早期的警告符号视而不见了。让我来对本栏目的剩余部分做个概括，让你清楚我要说什么。优秀的软件是可靠的，它起源于完整而关键的用户应用范例。差劲的软件到处是Bug，它源自于某人的个人想法。

邪恶双煞

你如何能在差劲的软件被集成到源代码主分支之前就发现它呢？首先，牢记质量有两面——"工程"和"价值"。大部分工程师关注于质量工程这一面——Bug。然而，在工程实现方面毫无瑕疵的功能可能对于客户来说是华而不实的，因为这个功能可能一开始就是个错误。关于这个，我在第6章的"有时间就做软件设计"的"质量的另一面"栏目中有更多的论述。

我们希望对有毛病的代码和有可疑"血统"的代码都能作出预测。预测有毛病的代码要容易一点，因此我们就从它开始。

嫌疑惯犯

2003年，Pat Wickline研究了Windows Server 2003产品后期Bug的根源。结果跟他2001年对Windows 2000修补程序解决的Bug的研究差不多。简单来说，超过90%的Bug都可以通过设计复审、代码复审、代码分析（使用像PREfast这样的工具）和计划的测试找出来。没有哪一种方法可以找出所有的Bug，但组合使用上述各种方法几乎可以找到所有的Bug。

2004 年，Nachiappan Nagappan 在一个一开始很好但后来发现很多 Bug 的工程系统中研究了可度量属性。那些属性就是“代码返修率”（每个文件增加或改动的代码行所占的百分比）和“代码分析结果”（每行代码中发现的 PREfast 或 PREfix 缺陷数）。

> **作者注：**他已经更正了他的想法，把重点放在了返修率（Churn）和复杂性度量以及组织结构上面。2008 年，Nachiappan 公布了一项发现，即与缺陷高度相关的属性是组织结构。与组件架构相关的组织结构的缺陷是最少的（这些组件只由单一的团队编辑），而多个团队编辑相同组件时，与此相关的组织结构的缺陷是最多的。我相信，这项研究发现是软件管理历史上最重要的发现，但我并不想去真正了解它。

如果你想阻止工程化实现很糟糕的代码进入主分支，一定要让你的创建跟踪代码返修率和代码分析结果。如果那些度量结果不符合代码的质量标准，那么拒绝签入那些代码。如果你的开发人员不喜欢那样，告诉他们去做更多的设计和代码复审，以及编写更多的单元测试。

你可能会问：“可能有这样的情况，就是代码返修确实太多了，但它真的是完整而关键的客户应用范例所需要的功能。怎么办?”请允许我首先恭喜你想到了唯一一个相当好的不把代码完全当垃圾看的理由。然后，请你把原来的代码完全当成垃圾吧！该对那一块代码进行重写了。这是能让那个功能达到你的工程质量目标的唯一方法。

你会喜欢它的

让我们接下去预测可疑功能“血统”。有毛病的代码很容易度量和控制，尽管它确实需要管理层设置一个质量门槛并持之以恒。相比之下，软件的价值就不那么容易度量了，但说到底它也需要同样的东西——管理层设置一个门槛并持之以恒。

你如何才能知道某个功能或源代码签入是否会真实地符合客户的价值呢？很简单！只要它是一个完整而关键的客户应用范例的一部分，用户就会喜欢。那你又如何能知道应用范例是否完整而关键呢？这才是难点。

所幸的是，你不必干那种活。我们花钱雇用市场营销人员、产品计划人员和上层管理者，就是为了让他们去为每次产品发布搞清楚完整而关键的客户应用范例的。没有哪个功能团队或产品部门能够做这件事情，因为完整的应用范例通常会绕过产品部门。注意，工程部的职责是告诉计划人员什么应用范例是可行的，然后从头到尾可靠地实现计划之内的关键应用范例。

停止卖弄愚蠢

当然，不只是项目经理，过分热心的各类工程师都会试图私自做一些功能，而这些功能并不是计划之内的完整而关键应用范例的一部分。尽管这么做可能会治好那些工程师的创造性的“便秘”，但可以预见的是，他们做出来的东西必定令客户讨厌。

为了在稀缺资源被浪费之前设法阻止那些构想糟糕的功能，你必须采取如下两个措施：

1. 确定一个清晰的文档化的远景或价值主张，列出产品发布所需的所有完整而关键的应用范例。程序原型、角色扮演、用户体验设计和高层架构也能非常好地澄清什么才是真正需要的。

2. 召集一个监管会，他们掌控远景或价值主张，并拿它们去审查每一个功能。如果一个功能不能适用于任何一个完整而关键的应用范例，那就要被抛弃。就这么简单！在每一个主要里程碑

的开始时刻，每一位部门项目经理（Group Program Manager，GPM）都要跟监管会复审即将实现的功能清单。尽管监管会可能不会在深入细节的层次复审每一个功能，但他们仍然必须一丝不苟地支持产品发布的质量、价值和完整性。

> **作者注：**微软内部很多出色的部门几年前就已经采用了这个过程。

上述这两个措施正好符合“设置一个门槛并持之以恒”的做法。尽管这个门槛跟工程质量门槛比起来更加主观，但两者都需要管理层执著地承担义务才能成功。

质量就是没有意外

预测质量并不难。实际上，它很简单。然而，各级管理者极少为保证质量表现出必要的严厉。

也许管理者担心保证质量会增加太多的时间开销。似乎从一开始就把事情做对，坚持客户的关键需要会花费更长的时间。实际上，当质量就是客户期望得到的东西时，把精力放在质量上总是产品发布的最近途径。

也许管理者担心工程师不喜欢保证质量。似乎工程师并不以他们自己的工作为荣，或者他们喜欢含糊，喜欢浪费时间。实际上，工程师对他们工作的质量有着强烈的自豪感，更喜欢知道别人的期望，并且厌恶工作浪费。

事实是这样的：质量是人心所向的，质量是基本需求，质量是我们成功的关键。因为我们的客户就是这么认为的。

质量是我们要做的正确事情和正确做事方式。它对于我们未来的生存和繁荣至关重要。质量就是没有意外发生。你能预测并控制它。所有你需要的只是头脑和毅力。赶快行动吧！

> **作者注：**尽管“牛肉在哪里”和“对质量的大胆预测”两个栏目都极力拥护质量，但值得注意的是，它们之间还是有些不同的。前者讨论的是为什么需要质量，以及获取质量的技巧；而后者描述了如何去度量并提炼你的工作，以把质量门槛抬得更高。我们已经取得了重大进展，但质量是一个需要永远警惕的理想。

2008 年 5 月 1 日：“碰撞测试：恢复”

我曾听到过一句看似有点傻的说法，但是当我认真细想后，我意识到这种说法却是绝对的白痴及不负责任。这个说法是这样的：与其捕获异常并修复它不如让软件崩溃后由 Watson 来报告错误。

> **作者注：**很多人要发疯了，因为我认为应该捕捉住异常。很多细心的读者提出对捕获异常的安全性担忧，以及在一个受损的环境下继续运行一项应用的危险性，我并不这么认为。如果错误或异常破坏软件运行，你就不能简单地继续了。我的观点是回退，而放弃对于用户来说是不负责任的。以下我要说的一种方法是回退，报告并重启（重新开始）这项应用，就像现在的 Office 一样。

Watson 是当一个基于 Windows 系统的应用软件崩溃时，弹出发送错误报告对话框背后机理的一个内部名称（一种提示错误的方式，确实引起了我们的注意）。

从技术的角度看，允许这种崩溃产生并予以报告，某种程度上可以说是策略性的。就如断言背后的逻辑——当你意识到情况不妙时，记录下这种情况并选择放弃。这样，当你日后要调试的时候，你就能尽可能地接近问题的根源。如果你不马上选择放弃，则重构环境并标识哪里出错了通常变得不可能。这就是为什么断言备受推崇，是吧？所以，崩溃也合情合理，是吧？

> **作者注：**断言是一种编程构想，它用来判定，是否程序员认为某种联系存在，这种联系就确实存在。如果确定存在，在调试的时候，断言通常中断程序，而在运行的时候断言会记录错误。断言通常用来检查函数参数的格式是否正确，以及检查对象的持续性状态。

哦，行了。断言及崩溃那是 20 世纪 90 年代的事了。如果你仍然那样想，你还是关掉你的随身听加入到 21 世纪来吧，除非你只是为自己或一些老顽固写写小软件。现在，软件并不是等到开发员感觉累了才停止运行，对它的要求是运行并持续运行。永不停息。

与现实搏斗

慢着，有位老古董，我叫他 Axl Rose，想以“现实”说话。“瞧，”Axl 说，“你不能期望糟糕的硬件境况自行消失，也不可能修复每一个 Bug，不管你们在一起工作有多晚。”你是对的，Axl。当我们要尽力设计、测试并开发无错误的产品与服务时，却总是会有 Bug。在这个全新的世纪中我们所认识到的是，很多问题并不在于 Bug——而是我们怎么应对这些 Bug。

Axl Rose 应对的办法是记录 Bug 的相关信息，并期望找出原因。而聪明的工程师则期望发现 Bug 并记录它们，使他们的软件能在错误时迅速恢复。是的，我们仍然期望修复我们记录下来的 Bug，因为错误将破坏客户的体验并造成重大损失。不过，汽车、电视还有网络都会出毛病。它们只是被设计成在错误中迅速恢复而极少会产生灾难性破坏。

> **作者注：**本文是我最具争议的专栏之一。3 年后，读者们基本形成两大阵营：
>
> - 一部分读者不能理解的技术细节是：不可放弃的断言及未捕获异常时重新开始进程。他们不喜欢我的专栏，还质疑我的精神状态是否有问题。
> - 另一部分读者对于这篇专栏的理解不是关于断言与异常而是恢复。他们受到我的专栏的激励，努力提高他们的产品或者服务的客户体验水平。（他们还是我的专栏的拥趸。）

或许，断言越少越好

“但是断言还是有用的，对吗？每个人都这样说。”Axl 说。不，断言对于今天来说就是种罪恶。它们是种罪恶，我认为是罪恶。它们使程序变得脆弱而不是富有弹性。它们产生了这样一种固定思维模式：面对错误我只有放弃而不是回滚后再重新开始。

我们要改变断言运作的方式。断言应该记录下相关问题并触发恢复，而不是放弃。我再次强调——保留断言，但改变它们的运作方式。你可能还期望断言能在早期检测出程序失败的原因。

但更重要的是，你怎么应对这些失败，包括那些溜掉的错误。

> **作者注：**还有一点要提一下——使用断言来预先检测问题的做法很好。但在程序出错时使用断言来逃避修改代码则不适当。

出师未捷身先死

那么，怎样正确地应对程序失败呢？这么说吧，我的意思是，在现实生活中，你怎么面对失败？你是否选择放弃，逃避？如果这是你的态度，我怀疑你在微软的面试流程中就这么干过。

当你经历失败，你要重新开始。理想的状态是，你把你哪里做错了记下来并分析其中原因，但是往往为时已晚。这时，你可能就放下包袱重新再来了。

对于 Web service 来说，有称为“五重”的解决办法：重试、重新开始、重启、重新镜像、重置机器。我们来一一展开讲：

- **重试。**首先，你重试一下错误活动（action）。通常第一次出错的地方第二次也会出错。
- **重新运行。**如果重试没用，重新运行往往有用。对于服务来说，这通常称为回滚并重新运行一个事务或卸载一个 DLL，再重新载入这个 DLL，像 IIS 那样依葫芦画瓢做一次。
- **重启。**如果重新运行没用，就跟用户一样，重启一次机器。
- **重新镜像。**如果重启还没用。就跟客户支持一样，重新镜像恢复这个应用或整个系统。
- **重置机器。**如果重新镜像也没用，那就是该换台新机器了。

> **译者注：**活动（action），指的是用户完成一项任务时，程序背后代码及系统的执行过程。

欢迎来到热带丛林

在数据中心我们的大部分软件并不是以服务的方式运行，与你认为的 Google 的运行方式相反，用户并不想所有的软件都依赖于服务。对于客户端软件，“五重”方式看似与你无关。呵，这样你就太天真、太轻率了。

“五重”方式同样适用于客户端或 PC 及手机上的应用软件。工程师们没搞清楚的关键之处是界定活动（Action）的范围，即重试及重新运行的适用范围。

对于 Web 应用则很容易界定——就是数据库事务或者网页的 GET 及 POST 请求。对于客户端及应用软件（client and application software），你要多想想什么活动是用户或子系统需要重来一次的。

设计优良的软件在每次活动后有一种客户端错误处理程序，就像我在专栏“错误处理的悲剧”中会提到的（见第 6 章）。每次活动后的客户端错误处理将使得应用“五重”方式变得非常简单。

遗憾的是，很多怯懦的工程师，就像 Axl Rose，如我在同一专栏中所述的那样，却是使用错误集中处理程序（RECH）。如果你的代码看起来像 Axl 的，你就会在这些活动之外有相关处理程序，但如果只是少部分的活动隐含了大部分的致命性错误，而你又不能修复最根本的问题时，才应该这样做。

译者注：客户端及应用软件（client and application software）。客户端指的是网络版软件运行在用户机器上的客户端软件；应用软件指的是只在一台独立机器上运行的单机版软件。

重新开始

下面来看看客户端软件应用了“五重”方法的一些例子：

- **重试。**PC 之类的设备比之 Web 服务器更具确定性，所以有一次操作失败很可能就会再次失败。但是，网络连接或数据连接出现问题进行重试时，不一定会再次失败。所以，当保存一个文件时，试着等待一次短暂的超时再接着试一次——这时等待同样的时间或再短些会更好，而不是无限期地等待直至失败。这样的同步性方式要好得多，会使用户感觉无障碍，只是实现起来有点棘手。
- **重新运行。**什么东西在客户端可以重新运行？设置驱动吗？数据库连接？OLE 对象？DLL 载入？网络连接？线程？对话框？服务？还是资源句柄？当然，盲目重新运行你所依赖的组件是很傻的。你必须明白错误的类型，要重新运行整个活动过程来保证你没弄混当时的状况。是的，这并不烦琐。让我发疯的是像用户那样自作聪明，我确实干过重新运行组件这样的事，我遇到的半数问题都能修复。为什么程序代码本身不可以这么干？为什么代码这么死脑筋？少安毋躁，答案很快就有。
- **重启。**如果重新运行组件没有用，或者因为这次错误很严重而不可能这样做，你就要重新启动客户端机器或客户端应用程序。现在大部分办公系统都会自动这样执行。它们甚至会恢复到之前的运行状态。有些手机软件或游戏会特意保存屏幕状态并重启应用或设备以便恢复（只对快速重启有效）。
- **重新镜像。**如果重启应用程序还没用，那么产品支持系统会告诉你什么呢？重装软件。是的，这是个极端的方式，但是，现今安装或修复对于大多数应用程序来说已经完全程式化，可以达到组件级别。你可能很想让用户参与进来（程序安装修复时），甚至必须检索在线修复。但是如果你期望用户这样做，那还是应该你来做。
- **重置机器。**这个我们无能为力。如果我们的软件不能修复错误，客户就别无选择了。现在，当竞争对手绞尽脑汁争夺我们的市场份额时，我们还是保佑已经试过之前所有的办法了吧。

作者注：虽然关于本专栏，我收到了成堆向我唠叨的邮件，但是很多建议已经在 Windows 7 中采用了，这将使正式发布版更具可恢复性。我还是不能妄下保证——这些想法应用起来会很有效，而且能提升我们的客户体验。在提升产品方面我们还有很多要做的。

别着急

Rose 先生还有一个问题：“等等，我们不能单方面地采取这样的行动。必须给客户以警示并得到首肯，对不对？”没错，Axl，但要视情况而定。

当然，有很多例子说明，客户必须提供越来越多的权限来决定重启某一子系统或修复。还有很多情况下，一次活动很耗费时间或者有副作用。不过，大多数活动还是简单明了的，解决问题

也不需要用户太多的介入。尽管这里的关键词是活动。

基本没办法给用户以警示，除非这种警示具有可操作性。只有消息提示。操作失败的时候，如果对修复错误或防止其再次发生毫无办法的话，那显示消息提示又是想做什么呢？为什么不直接告诉我说拿把斧子把屏幕砍了吧？如果有建设性的措施我可以采取的话，为什么代码本身不直接把这事做了？我们竟然有胆量认为客户是个傻瓜？难以置信。

没什么两样

“好吧，虽然这需要多花点力气，”Axl抱怨说，“可谁说软件不会总是重试，重新运行，重启，重新镜像的？毕竟，如果Bug发生过一次，它就会再次发生。”没错，Axl，Bug分两种特性——反复性与随机性。有些人分别称为Bohrbug与Heisenbug。

> **作者注：**说到Bohrbug与Heisenbug，这两个概念要追溯到20世纪90年代，Jim Gray在他的论文中有论及，“为什么计算机宕机以及我们该怎么办？”

采用“五重”方法可以解决随机Bug，即使它们发生也基本无害。然而，记录反复出现的Bug是很重要的，即使程序或服务并没有崩溃，我们还是希望能产生错误报告，这样可以认知这些反复性Bug并修复它们。好消息是最麻烦的Bug——也就是反复性的Bug，却是最容易修复的。

再花点力气，我们就可以使软件面对失败时更具可恢复性，即使Bug没有彻底根除。这样就会看起来像是没有Bug发生并继续一如既往地按原先方式操作。唯一的不同是我们怎么对待错误——预计、记录并处理它们，而不是捕获它们。当软件运行正常时，我们不就得到了家人与朋友的赞誉（同时拥有更多闲暇时间）了吗？欢迎来到新世界。

> **作者注：**我并不奢望明天就实现这些方法。这是次重大改变，特别是在客户端及应用软件领域。以前只有极客才有计算机，所以用户知道怎么去重新运行或修复设备。而今，一切都需要在没有或极少用户干预的情况下运行。部分的解决方案需要很高的工程水准，但也就仅此而已。即使代码没有任何问题，但程序还是会出错。出错后能即刻恢复是必需的。

2008年10月1日：“盯紧标称”

人们总是期盼激动人心的技术变革或过程创新来解决所有问题——提升他们的生活品质，减少他们的腰围，提升他们开发团队的生产力。这就是为什么像敏捷及六西格玛之类的过程管理方法这样令人着迷。只要在你的开发团队身上贴上Scrum标签，然后就会“嘭！”——你的团队能力就马上提高了10倍的功效。

如果这不是一个沉迷于魔力的自恋狂脑子里的想法的话，那还是挺让人高兴的。我很高兴人们乐于尝试新鲜事物。我只是希望他们不要混淆新奇与必需，或者打个小补丁与关键问题的区别。

真正问题在于，及时完成一项高质量的完美软件是没什么可称奇的。（同样，软件服务在这方面也没什么不同。）几千年来，人类通过庞大的行业队伍建造了惊世骇俗的工程。这里没有魔

法。只是精于设计，重在执行。一直都是这样。

> **作者注：**如果你的代码库只有不到 100 000 行，而参与这个项目的也不到 15 人，那你就不需要这么精细的设计与规范的执行。你可以随心所欲——突击设计，只有要求非常宽松的先期设计标准，在客户的催促下才无休止地重写或重构代码。当你的代码库及项目变得越来越大，就必须有精致的设计及规范的执行，不然那就是一团乱码，一个人心涣散的团队，一个失败的计划。

返璞归真

在来微软之前，我在波音公司工作了 5 年。这个航空公司使用术语标称（nominal）和公差（tolerance）来描述可接受的值和精密度。标称类似于支撑梁被置于加热管 2 英尺的地方。公差则表示距离上可以有 1.5 英寸的偏差。

总有菜鸟工程师过于关心公差。他们不断地寻找一个更“聪明”的技术给他们的公差增添更多的余地，好像这非常重要似的。真正的飞机设计建造工程师们知道，这些修正公差的工匠完全摸不着重点，他们已误入歧途，不切实现，还有些天真。

建造一架飞机的关键不在于修正这些公差——而是盯紧标称。并不是说：“钢轨的弯曲度是否在 0.058 英寸内?”而是，“钢轨正好能通过通风口吗?”别再操心那些眼花缭乱的细节了，回归到本源上来，醒醒吧！

> **作者注：**你可能会认为我在批评波音公司，但事实相反。当生命置于平衡棒，我必须清醒地记住真正的目的——那就是把客人安全地送达目的地！如果钢轨偏离了几毫米，你还可以用垫片补上。但如果两个关键航空系统在同一个空间中运行，一个不起眼的事故就足以导致空难。不要让这些细节使你远离客户真正关心的问题。

我想成为名牛仔

倒腾公差而不是盯紧标称对于软件业非常有害。你可以在大多数软件中看到这些结果，不管这些软件是怎么以及在哪儿发售的。软件工程的焦点都放在了林林总总的细微特征上，而不是让这些软件完成它们的基本任务。

或许基本任务不是工程师的主要责任。或许软件工程师就是些修补工匠，而不是那些想要简单又可靠的解决方案的人。但是，我并不这么认为。

我认为注重标称需要精心的设计与规范的执行，两者都需要软件工程师将工程与客户需求放在个人兴趣之上。遗憾的是，这可不是牛仔程序员在乎的，这就是牛仔程序员会成熟或被淘汰的原因。

但是，这很简单

脑子还清醒吧？还相信客户及项目至上吗？还认为只有在你们做到像团队那样合作，把客户放在中心而不是你们自己时，真正的客户价值才能实现吗？好的，让我们来谈谈软件的标称。

如我所说，标称指的就是精心的设计及规范的执行。在意义重大的工程创新过程中也是这样

的。这对软件意味着什么呢？简而言之，精心的设计就是：

- 充分领会客户想要通过你的软件完成什么任务（产品计划、价值主张/视角）。
- 周详考虑端对端体验，包括一些误区（体验设计及范例）。

规范的执行指的是：

- 根据工作的优先级及依赖性进行排序。
- 根据之前项目的经验数据建立时间表。
- 每个步骤都要设立完成标准并贯彻执行。

简单地说，标称就是“三思而后行”并“准确定义完工的概念”。

> **作者注：**很简单，不是吗？“三思而后行”，“准确定义完工的概念”。谁不会这么干？看看大多数项目中菜鸟团队干的事吧。当一些缺脑的执行官没有制定精心的范例、架构设计以及完工标准时，即使久经考验的团队也会不加多虑地匆匆上马。

搞定了吗

之前，对于体验及工程设计，我已经讨论过很多次（特别是在第6章的“质量的另一面——设计师和架构师”）。因此，现在让我们深究一下“准确定义完工的概念”。拿架构中可工程化的模块来说，比如一个特性、组件、API或Web service，那它们的完工标准又是什么？

你可能设置一些Bug限定数或代码测试覆盖率，但这只是评估中间成果。你最后真正关心的结果是什么呢？架构中的工程化模块的最后结果应该是怎样呢？是的。端对端范例中的每一部分应该都让人满意了才行，包括性能目标及避免体验误区，最终结果应该是安全可靠的。

是什么导致软件在范例执行及可靠性方面失败呢？在微软乃至全球投身于此的智者们，为了这个问题已经深究了几十年。你所看到的研究的最终结果都是仅仅源于无法完成以下五项实践——这五项实践应该全部做到。

- **设计归档。**开始前先考虑成熟，并将想法归档，这样你的同事们就可以……
- **设计复审。**这样，在写第一行代码前你就可以找出大半的错误，那么你就可以……
- **代码复审。**如果你执行代码审查，你就可以发现70%以上的错误，虽然有些错误只能通过……找到。
- **代码分析。**通过像lint或PREfast这样的工具，可以找出遗漏掉的特定错误，但不可以完全替代……
- **单元及组件测试。**这可以发现剩下的问题，包括抗压性、性能及极端测试的问题，比如端对端范例检测。

如果没有“准确定义完工的概念”，任务就永远不能完成（似曾相识?）更糟糕的是，牛仔码农会声称完工了，但对工作质量概不负责。想知道为什么优秀的开发人员要为差劲的开发人员擦屁股吗？因为差劲的开发人员只会拿一堆垃圾宣称完工了。如果没有“准确定义完工的概念”，谁能保证让混乱的代码远离用户呢？

> **作者注：**我之所以介绍了这么多不同的系统测试步骤，是因为完工的定义是针对组件而不是整个产品的。显然，在一个产品或服务发布前还有其他很多步骤要完成。

并没这么复杂

注意，为什么五项实践没有一项作为完工特定的或主要指标？它们只是基本步骤。每个人都知道应该这么做。但工程师们却醉心于成为一个经过认证的 Scrum 大师，创建很酷的工具，在他们进行设计复审或编写全面的单元及组件测试前依赖于最新的技术。

在钻研公差之前先要注重标称。没有什么魔法，这不复杂，也不难。只要规范，只要把客户及项目放在你自己之前，只要有专业的要求而不是稀里糊涂。我想你会掌握的。

第6章

有时间就做软件设计

本章内容：

2001年9月1日："错误处理的灾难"
2002年2月1日："太多的厨师弄馊了一锅好汤——唯一权威"
2004年5月1日："通过设计解决"
2006年2月1日："质量的另一面——设计师和架构师"
2006年8月1日："美妙隔离——更好的设计"
2007年11月1日："软件性能：你在等什么？"
2008年4月1日："为您效劳"
2008年8月1日："我的试验成功了！（原型设计）"
2009年2月1日："绿野中长满蛆了"

"在所有的软件开发过程改进中，怎样的改进对我们产品的质量和价值具有最大的影响呢？"这是我目前担当的角色要求我必须回答的基本问题之一。我最初的回答是："对渴望的结果进行广泛而持续的度量。"因为它们能告诉你要做何种正确的改进，以及改进带来的影响。它摒弃了臆测，提供了正面的基于团队的激励。

然而，如果我必须做出决断："度量出来的结果将会给我们带来什么启示呢？"我猜那会是"功能小组"（精益开发）和"设计至上"具有最大的影响。（我们早已谈过了复审、代码分析和单元测试。）我在第2章中谈过了精益开发。现在，该来谈一谈怎样做到设计至上了。

这一章对软件设计的基本原理和错综复杂的本性进行了解析。第一个栏目以基本的错误处理开篇；第二个栏目对代码和数据的复制进行了批判；第三个栏目概括了完整的设计过程，同时推荐了最佳实践；第四个栏目重点论述设计杰出的用户体验以及对它们的实现；第五个栏目揭示了构架的真正价值及目的，而不是让构架师们碌碌无为；第六个栏目以一个用户的角度讨论软件性能设计问题；第七个栏目讨论服务领域中设计的真正价值；第八个栏目对原型进行了试验；最后一个栏目提倡在新的开发领域中的自律性。

在开始阅读这些栏目之前，我想指出工程师逃避设计的两大理由：设计会自然浮现；另外就是没有足够的时间。主张自然发生的设计的人声称，前期设计是一种浪费；作为替代方案，你应该随着工作的进展逐步发现并重构设计。这对于小型代码库（少于10万行）或许很好，因为重构对于小型代码库不是什么大不了的事。然而，大部分产品代码库要庞大得多，严重的返工非常

地昂贵。在你重构或做出其他改动之前如果不把问题彻底考虑清楚，结果你可能会损失惨重。因此，现在就抓紧时间去做设计吧，否则你后面肯定没有足够的时间。

——Eric

2001 年 9 月 1 日："错误处理的灾难"

如果说我们的产品代码在某一方面长期以来一直差强人意的话，那它肯定是错误处理。Office 产品在这个领域已经取得了一些较大的进步。Office 2000 给它的错误对话框加上了 LAME 注册表设置。Windows 2000 也在这方面改进了一些，它极力提供有意义的信息和建设性的指示。Office XP 和 Windows XP 现在会自动地把严重的错误报告给我们，以便我们评估和跟踪。然而，这些努力给我们带来的改变，也仅仅是以前当我们的用户跌倒的时候我们还要踹一脚，现在我们会在第一时间向他们道歉，然后也许递给他们一根藤条，让他们抽打自己以使他们快点站起来。

> **作者注：**Office 2000 内部的 LAME 注册表设置，是在每一个 Office 2000 的错误对话框上加了一个 lame 按钮。如果你不喜欢那个对话框，你可以单击那个 lame 按钮，于是你的"投票"会被记录下来。如今那个按钮已经被替换为"这个信息有帮助吗?"的链接，放在 Office 错误对话框的底部，所有用户都可以访问。

那么，为什么代码不能自动修复错误呢？我研究了我们的代码很多年，明白了有两个主要的原因。第一，代码不能在第一时间知道什么东西出了故障。第二，没有写错误处理代码去修复问题，即使它知道出了什么故障。这两个问题是相互关联、彼此渗透的。让我们来看一个典型的情形。

恐怖，恐怖

一个开发人员写了一堆代码，另一个开发人员再增加一堆代码。然后，他们把其他部门的代码也加入进来，接着，他们自己增加更多的代码，最后，他们意识到必须处理错误，但他们又不想回过头去到处加上错误处理代码，于是他们写了一个"错误集中处理例程"（Routine for Error Central Handling，RECH），把所有错误都传到那个例程去处理。当他们开发下一个版本的时候，也许由完全不同的人增加更多的代码。有些人返回有意义的错误，有些人返回简单的布尔值：成功或失败。有些人喜欢用异常，有些人喜欢用错误数值。但是，RECH 要么处理异常，要么处理错误数值，不可能两样都做。

> **作者注：**我为本栏目发明了 RECH 这个缩写词。我不知道对于它具体所指的糟糕实践是不是另外有个专有的名字。

如果 RECH 例程处理的是异常，那么原本返回错误数值的代码就要包装成当有错误发生时抛出异常；如果 RECH 例程处理的是错误数值，那么原本抛出异常的代码就要使用通用捕捉来把异

常转成错误数值；如果代码块返回的是成功或失败的布尔值，那么要把它转换成通用的异常或错误数值，以便后面再有可能把它转成错误数值或异常。即使你调用一个返回描述性错误数值的函数，像很多 Win32 或 OLE 函数一样，这些调用也会常常被包裹在一个劣质的函数中，它要么忽略丰富的错误描述而只返回成功或失败，要么对错误数值完全不做任何判断。这样一路处理下来，信息就丢失了，永远地丢失了……

使用异常

往伤口上再撒点盐，把异常和错误数值搅和在一起，你将面临的是灾难。如果你不能正确地释放栈对象，你就不能使用异常。这通常意味着，任何需要特殊析构的东西都必须是一个对象。这在 C#中很容易做到，但在 C 和 C + + 中比较难。

举例来说，你必须在可能抛出异常的代码中专门使用智能指针。然而，如果有一部分代码使用异常而其他部分不用，你就常常会遇到这样的情形：一个返回错误数值的函数，它内部调用一个抛异常的函数，而你在往上 3 级的地方才进行异常捕捉。如此一来，一个错误引发更多的错误，纠正措施也失去了意义。如果是在多线程的应用程序中，情况就更糟糕了，因为每条线程都必须处理异常。

别丢弃，用上它

你希望以尽可能最佳的方式使用你所得到的错误信息。那么你该怎么做呢？首先，拿起你的“毒药”并一直带着它：

- 如果你想要使用异常，可以，但务必要在所有的地方都使用异常，并且把需要特殊析构的东西包装成对象（.NET 框架模型）。
- 如果你想要使用错误数值，可以，但务必要以无损耗的方式传递它们，或者在它们发生的源头就进行处理。
- 如果你确确实实必须把异常和错误数值搅和在一起，那么要在一个返回错误数值的函数中，给每一个可能抛出异常的调用加上异常处理程序；而在一个抛出异常的函数中，把每一个返回数值的函数调用的结果传给专门的返回值处理函数。这样的话，至少你不再会丢失数据。

> **作者注：**请注意，这并不是说你就是为了打包异常而打包，这样就太没效率还很怪异。而是说你想通过一个外部函数隐藏一次异常，而这个函数不应当有异常。为了保持外部函数约定的一致性，你应该从内部函数中捕捉异常并将它们转换成一个合适的错误代码，最好不要首先考虑使用混合模式。

接下去，你必须计划错误发生时采取的措施。搞清楚当一块代码出现状况的时候，你最高在哪个级别上还能知道该采取什么措施。这样的级别很少是在最顶层，因此不要采用错误集中处理。

在那个最高级别上加上你的错误处理代码。这将在你的代码中增加不止一个错误处理函数，但这种错误处理函数也不大可能会超过 1 000 个。关键是，你要在栈上尽可能高但又不能再高的地方添加错误处理。当使用错误数值的时候，你可能需要在应用程序对象中缓存一些类似于文件

路径或标记的关键信息，以便于错误处理代码能够使用它们进行修复问题。

把错误报告给用户只能作为最后的手段，或者虚张声势的伪装。最终的效果是，系统看起来永远都是能够工作的，或者至少总能给客户提供必要的关怀。就因为这个，我们的客户会更喜欢我们！

2002 年 2 月 1 日："太多的厨师弄馊了一锅好汤——唯一权威"

Segall 定律说："有一块手表的人知道时间，而有两块手表的人对时间总是不太确定。" 尽管大部分人都能直观地理解这个关于手表的规律，但当同样的规律发生在编程上面时，大部分开发人员显然摸不着头脑了。我们的代码被重复的算法和数据搞得乱七八糟，不仅在全公司范围内是这样，即使在单个应用程序之内也存在这样的问题。

看起来每个人都想动手制作他们自己的手表。我们以各种无法调和的方式存储用户数据或系统数据。我们到处复制、粘贴代码，放任它们失去同步。我们甚至不能在同一个应用程序中共享代码，那么在不同团队之间共享代码的想法岂不是荒唐之极！

一张图说明问题

你不相信我？打开 Windows 文件浏览器，找到一个包含 .jpg 或 .gif 文件的目录。然后在 PowerPoint 中打开一页空白的幻灯片，把一个 .jpg 或 .gif 文件复制一下，粘贴到幻灯片的背景区域。然后再粘贴到文本区域。迄今为止，一切正常。

现在，直接在 Windows 文件浏览器中拖动那个 .jpg 或 .gif 文件，然后把它放到幻灯片的背景区域。接着再拖动一次，把文件放到文本区域。哇！如果你觉得不过瘾，你还可以对格式化的一段文字做同样的试验。你会发现，当你粘贴时会得到一个智能标签，而在尝试拖放方式时却没有。

> **作者注：** Office 2007 似乎已经解决了这个关于图像的 Bug，但格式化文字的问题仍然存在。

我并不是故意针对 PowerPoint。你可以试一下其他应用软件和其他数据。你将会发现，程序对粘贴过来的数据和对拖放过来的数据在处理方式上是如何不一样，而且这并不完全是由数据放置的位置引起的。这些应用场景应该有一样的表现。数据在所有情况下都是一样的。实际上，Word 对于粘贴和拖放的处理是一致的（干得好）。

那么，为什么同样的数据以大致相同的方式插入到一个文档中会产生不同的结果呢？嘿，那是因为程序走了不同的代码路径。具有讽刺意味的是，重复代码路径常常是由于复制、粘贴原始代码造成的。

有人确切知道现在几点了吗

有两条代码路径去做同一件事就好比有两块手表。也许它们有一样的表现，也许它们表现得不同。一旦有了 Bug 修复或规范书变更，保持所有代码路径同步则会是一个噩梦。

当然，解决方案是为做那件事写一个函数，然后所有的代码路径都调用那个函数。这是计算机编程专业第一年就会学的基础课程。然而，我们一次又一次地在犯这种新手错误。

不过，问题不只是发生在函数上面。数据也有同样的问题。在你修复的 Bug 中有多少是由不同步的数据值引起的？这些数据值是不是曾经源于同一个值，但它们却被分开保存了？千万不要让我回想起我天天要被迫重复拼写我的 E-mail 地址！

维护两块手表并且保持它们都精确需要人工同步。这是一个单调乏味而且容易出错的过程。因此，只有一块手表的话，事情就会简单得多。遗憾的是，我们中的很多人都有手表、个人电脑和袖珍电脑，加上我们还和这么多人在一起工作，他们每个人都有自己的时钟。

所幸的是，你的个人电脑可以自动跟网络同步，而网络会同步到一个时钟上。然后，你的袖珍电脑、你的手表和其他人的时钟都可以同步到你的电脑的时间上去。让所有的同步自动化执行，问题就解决了。

只能有一个

这里有一个基本原则，我称为“唯一权威”。每个数据块都应该有一个唯一权威对它的值负责。每一个操作都应该有一个唯一权威对它的操作负责。保持你所有的时钟同步是可能的，因为有一个唯一权威对当前的时间负责。一致的用户界面是可能的，只要有唯一权威对每个操作负责——诸如绘画、数据输入和消息处理之类的事情。

每当你复制、粘贴代码的时候，停下来想一想：“我是在复制呢，还是这确实是一个不一样的操作？对于这个操作如何去做，它的唯一权威是什么？”然后再相应地采取行动。

当你想要创建一个新的注册表项的时候，停下来想一想：“这对于用户或系统确实是一个全新的概念吗，或者我还可以从其他的权威导出这个值？”

如果要求你开发一个自定义控件，想一想：“对于这个控件应该如何工作，是否已经存在一个权威？如果有，我如何使用它呢？”

如果你还顾忌性能问题，请记住，缓存一个计算过的结果常常比重新计算新的结果来得慢，因为处理器速度已越来越快，而缓存命中率受制于门控效应。

万物皆有联系

唯一权威在 .NET 世界显得尤为重要。我们正在努力给我们的用户一个身份的权威、一个日历的权威、一个通信簿的权威，等等。我们笃定，最终这种便利将给客户带去实惠，也会扩大我们的盈利。

不过，这只有在我们使用同一个方法去执行所有相关的操作时才能实现。否则的话，这不仅会给我们带来 Bug，让我们用户感到困惑和混乱，而且还会成倍地增加安全漏洞和实施成本。

> **作者注：**尽管为身份、日历等设置唯一权威的想法在理论上很不错，但从信任、隐私和协同能力方面去考虑，这个概念还达不到市场的要求。

在软件开发过程中，就出现过很多厨师弄馊鲜汤的事。这样的汤会让我们的客户及合作伙伴拉肚子，更不用说厨师们自己了。只要简单地想想，对于每一次操作或查询，哪些代码或数据是

唯一权威，这样我们就可以简化我们的代码，使其更容易修正及维护，并给我们的客户及合作伙伴带来持久、愉悦以及充满智趣的体验。

2004 年 5 月 1 日："通过设计解决"

你坐在一块白板前面，旁边一个"书呆子"就某个设计问题没完没了地跟你纠缠。你试图搁下讨论，把话题转到实际工作上去，但这个神经兮兮的家伙不肯罢休。"如果这发生了怎么办？""我们还没有考虑这种情况。""我们对那还不清楚。""我们还不能开始。"分析瘫痪！这时候，时钟敲响了，你的产品不得不出货了。

我们都知道设计很重要。我们都知道高质量的设计是高质量的产品的关键。但是很可恶，在某个时点你必须开始编码，在某个时点你必须要将你的产品发布。如果你不能及时编写代码将设计实现，那么世界上最好的设计也就只能当做墙纸来看看。我没听过很多管理者会这么说，"你去设计吧，想用多少时间就用多少时间！"

如何才算足够好

然而，另一个极端同样是有害的。刚刚才在白板上进行了短暂的讨论，就根据白板上的预想符号匆忙开始编码，这将导致大量的返工、成堆的 Bug 和不便于后续开发的脆弱框架。大部分"缺陷分析"揭示，所有缺陷中的 40% ~50% 是由设计引起的。跳过设计这一步绝对不是办法，但多少分析才算足够呢？你可以知道你什么时候"编码完成"（Code Complete），但你如何知道什么时候设计足以让开发工作开始了呢？

依我之见，缺少明确的设计过程是最令开发者沮丧的事情。你不可能不知道它的重要性——如果设计清晰而完整的话，编码要容易得多、快得多，而且会极大地减少出错的可能性。没有焦虑、没有困惑、没有惊讶、没人会像臭名昭著的霍默·辛普森一样发出"D'oh!"声。但你在上大学的时候学过设计过程吗？没有，学校不教的。那你在加入微软以后有机会学习吗？可能也没有。大部分部门对设计相当随意，而且很多部门根本就不做设计。嘿，你的好伙伴 I. M. Wright 在此会帮你一把。

作者注：接下来我将列出设计的步骤及方法，包括设计的方方面面。然而，你怎么知道哪个步骤你可以忽略，以及你怎么知道哪个步骤你已经完成了？毕竟，你是想做次充分的设计——不多也不少。答案是，自信——带着充分自信付诸行动。对于某一步骤，你要问一下："我们对这步设计满意吗？"如果是，你就可以跳过这一步，如果不是，就继续加以完善直到满意为止。相信自己与团队，你不需要了解每一个细节，但你必须知道你在做什么。

译者注：霍默·辛普森（Homer Simpson）是美国动画电视剧《辛普森一家》中的一名虚构角色，他是部分美国工人阶级的典型代表：粗鲁、超重、无能、心胸狭窄、笨拙、粗心、酗酒。霍默的口头禅"D'oh!"于 2001 年被收入到了世界上最大的《牛津英语词典》中，解释为"得知发生严重问题时表示沮丧"。

设计完成

一个设计过程最重要的是“完整”。完整说明你的努力已经足够了，你不想做太多，也不想做太少。软件是有多个维度的，有内部工作和外部接口之分，有静态概念和动态交互之分。所有这些维度都必须涉及，以创建出一个完整的设计。另外，有很多不同的工具可以帮助你在不同的领域和抽象层次进行设计。下面的这张表可以帮助你搞清楚我所说的意思。

软件设计的维度

	外　部	内　部
静态	项目管理规范书 应用程序编程接口定义	开发规范书 测试驱动开发
动态	用例 应用范例和角色扮演	顺序图 状态图、流程图、威胁和故障建模

瓦茨·S·汉弗莱有一次在微软访问的时候给我看了这张表。我和我的团队把它应用到了平常的设计实践中，并且依象限顺序按部就班地照做了。

> **译者注：**瓦茨·S·汉弗莱（Watts S. Humphrey）在IBM工作了27年，负责管理IBM全球产品研发。离任后，他受美国国防部委托，加入了卡内基·梅隆大学软件工程学院（SEI），领导SEI过程研究计划，并提出了“能力成熟度模型”（Capacity Maturity Model，CMM）思想。在CMM浪潮席卷软件行业之时，他又力推“个人软件过程”（Personal Software Process，PSP）和“团队软件过程”（Team Software Process，TSP），使它们成为软件开发人员和开发团队的自修宝典。

大部分设计过程都从表中的左下象限以顺时针方向螺旋向内推进。整个过程先从高层次的设计步骤开始：

- **应用范例和角色扮演。**在一个较高的层次上描述客户是如何跟我们的软件交互的。
- **项目管理规范书。**描述一些外部的困扰，包括需求、对话框、菜单、屏幕、数据集合和其他功能。
- **开发规范书。**详细说明类的层次和关系、组件栈和关系以及其他所有需要在较高层次描述实现结构的东西。这份规范书也被称为“设计文档”（用词不当，因为这份文档只涉及设计的一小部分）或者“架构文档”。
- **状态图、流程图、威胁和故障建模。**描述并控制系统中对象与组件之间的复杂交互。

对于这些初始的高层次设计步骤，有些地方需要注意一下：

- 每个步骤都有一个有限而明确的范围。通过将设计分离成4个象限，然后逐一涵盖，你就不必依赖很多文字，也可以避免重复，最后依然能够完整地描述设计。尽量保持简洁，重用前面用过的例子，合适的时候使用一些简单的图表或图片。
- 不是所有步骤在所有设计中都是必要的。有些设计几乎没有外部接口。有些设计不需要状态。这有赖于你具体情况具体分析。

- 外部象限具有很宽的受众（所有的工种）。内部象限则面向单纯的技术人群。
- “统一建模语言”（Unified Modeling Language，UML）为很多领域提供了现成的图表。有些工具可以让创建 UML 图表变得很容易，比方说，微软的 Visio 就有一大堆内建的 UML 图表类型。
- 威胁和故障建模通过重用“组件关系图”，做起来就要容易得多，而且也容易做对（在这之前你应该早就创建了组件关系图）。使用 Excel 电子表格把威胁和故障进行分类和评级，并且指明解决方案。

> **作者注：**我们现在使用一个威胁建模工具来做这部分工作。你也可以从 MSDN 上找到一些。

细节，细节

正如前面的表格所示，在高层次设计中，每一个步骤都顺时针驱动下一个维度的步骤，最终转了一个圈；同样的过程也发生在更加细节的层次上。基本上，你可以通过表格中心的那些步骤螺旋进入实现设计，具体如下：

- **用例。**简单描述参与者是如何使用系统去执行任务的。如果你喜欢图片，你可以使用微软的 Visio，因为它已经为用例集成了各种各样的形状。

> **作者注：**用例（Use Case）这个术语常常用来描述非常高层的应用范例，而不是我这里所说的低层次的简单应用。由此给你带来的困惑我深表歉意！

- **应用程序编程接口定义。**当你有了用例之后，再去正确定义应用程序编程接口（Application Programming Interface，API）就会容易得多。这一步巩固了组件和对象之间的“契约”。接着，你就能真正地开始编写实现代码了。
- **测试驱动开发。**这个实践提供了一个理想的框架，由它可以系统、稳健地创建出简单、内聚、结构合理的实现。测试驱动开发完成的时候，一套完整的单元测试也同时呈现在你眼前。
- **顺序图。**对于特别复杂的函数或方法，使用顺序图来澄清代码的构造。

对于整个设计过程，我还要指出如下几点：

- 这个过程没有不必要的复杂性或多余步骤。每一步都很明确，并且保证进入到下一步时，它所有需要的信息都已经准备好了。
- 跟其他过程一样，为了节省时间，你可能会试图跳过一些步骤（比如在没有创建项目管理规范书、开发规范书或用例之前直接跳到测试驱动开发）。如果那些步骤产生的成果价值不高，也许问题不大；但如果那些成果是需要慎重对待的（事实常常是这样），那你就危险了。
- 你知道了“足够”意味着什么，也就知道了设计什么时候才算完成。

让我看看你是由什么组成的

好吧，我承认，我忽略了两个环节：

- 怎样才能得到正确的应用范例和角色扮演呢？

- 基于那些应用范例和角色扮演的项目管理规范书，跟包含类层次和关系、组件栈和关系的开发规范书之间是存在断层的，怎样才能在这个缺口上架起桥梁呢？

有两种方法可以得到正确的应用范例和角色扮演。其一是通过直接跟客户接触；其二是自顶向下地开始一个更高层次的螺旋式设计过程，具体步骤如下：

- 市场机会和角色扮演（外部—动态）
- 产品远景文档（外部—静态）
- 产品架构（内部—静态）
- 子系统交互图或流程图（内部—动态）

之后，再螺旋进入应用范例和角色扮演这一步骤，就像我们前面讨论过的那样。

> **作者注：**当你有几亿个客户、成千上万个工程师、成百万美元的利润时，我建议你在高层设计上使用严格的方法。如果那让你感到不安的话，我建议你不要参与这种大型项目。当牵涉那么多钱的时候，项目是由政治或过程来驱动的。

当心断层

为项目管理规范书（或者更高层次的产品远景）和开发规范书（或者更高层次的产品架构）之间的断层搭建桥梁，要求将功能需求（比如特性功能）和设计参数（比如类或组件）进行一一映射。这里有两个简单的方法：

- **设计模式。**要知道你以前已经解决过这个问题，然后重用以前老的设计。
- **公理设计（Axiomatic Design）。**如果是全新的设计就使用这种方法。即使是整理、改进以前老的设计，也可以用这种方法。它的步骤相当清晰，如下：

1. 在 Excel 或 Word 中创建一个表格，纵向列出所有功能需求，横向列出所有设计参数。内部的单元格应该保持空白以组成一个矩阵。它也被称为“设计矩阵”。

2. 确保写下来的每个功能需求相互之间都是正交的。这通常意味着你必须把复杂的需求分解成更为基本的片断。增加新的需求只需多加几个新行（很方便）。

3. 对于一个功能需求，列出满足它的所有设计参数；逐个对所有需求重复上述过程。通常一个设计参数（类或组件）有助于满足多个需求。在每个有助于满足相应功能需求的设计参数列的单元格上都要标上一个“X”。最后，设计矩阵的单元格区域会布满“X”符号。

好吧，我们现在来举个例子。简单一点的，为水龙头做个设计。两个正交的功能需求是控制流量和控制温度。我们同时来考虑这两种设计，它们每个都有不同的设计参数：一个是分离的热水和冷水开关，另一个是只有一个控制杆（通过倾斜控制流量，通过旋转控制温度）。这两个设计对应的设计矩阵如下：

水龙头的两个可选的设计矩阵

	热水开关	冷水开关
流量	X	X
温度	X	X

	倾斜杆	旋转杆
流量	X	
温度		X

4. 重新安排设计参数，目标是没有“X”出现在设计矩阵的对角线以上的区域。为了达到这

个目标，你可能需要重新考虑类或者组件如何才能更好地满足更少数量的需求。

从我们的例子中不难发现，分离的热水和冷水开关这个方案有一些“X”出现在对角线以上的区域，而单一控制杆方案的“X”非常漂亮地都落在了设计矩阵的对角线上。按照单一控制杆的设计生产出来的水龙头将好得多。

5. 结果，最后的设计矩阵将记录下一个完美的设计方案。如果所有的“X”都出现在对角线上，那么这个设计完全解耦了，实现起来将非常容易。任何在对角线以下的“X”指出了依赖关系，它们可以帮助你合理安排开发的先后顺序。任何你经过努力还是无法消除并仍然留在对角线以上的“X”，处理起来要特别小心了，因为它们预示着令人厌恶的循环依赖。

成功处方

看到了吧，这就是一个关于如何一步一步得到一个最小但又完整而健壮的设计的指南。你对它可能有点望而生畏，但每个步骤真的都很简单。通过使用系统化的方法，你将不会遗漏任何东西，你可以做出有计划的安排，而在压力环境之下你也不会变得太疲于奔命。

这种类型的完整方法同时治愈了分析瘫痪。遵循每个步骤，你就能高枕无忧。如果人们试图增加额外的步骤或提出额外的需求，你就可以拿出规范文件以阻止他们的进一步动作。

正如我在以前的栏目“软件发展之路——从手工艺到工程”（参见第 5 章）中提到的那样，为了满足我们客户的质量要求，把 Bug 数量降到千分之一是必要的一步。另外要做的是，辨别需求并且创建满足这些需求的具体设计。通过使用一个完整的设计方法，并配以工程化的实现，我们就能超越客户对质量和价值的期望，我们的行业将为之一振，我们的竞争对手会在这个过程中一败涂地……

2006 年 2 月 1 日：“质量的另一面——设计师和架构师”

质量、价值、母性。这些可能是所有人都向往的理想化境界。没有太多人对它们进行争论。但什么是质量？什么是价值？我想同样问什么是母性，但我的妻子声称我永远也无法理解。

当我们被问及如何定义质量及价值的时候，我一筹莫展。这种情况就像要我把“狱火”与“硫黄”说得很动人似的。当然，这样不会有什么结果。如果不是为了得到免费的巧克力甜饼，所用的时间都是浪费。

麻烦的是，质量和价值隐藏在感觉之中。即使我们开发了一个产品，它精确地符合了规范书，而且没有任何 Bug，客户和评论员可能还是会不喜欢它。如果产品没有按照他们认为的方式去工作，那它就是垃圾；如果产品没有按照他们期望的方式去解决问题，那它还是垃圾。

然而，I-Puke 可能产生有问题的塑料片，而且它每天崩溃两次。但它就因为能够使人们想起宠物石而饱受赞誉，因此价钱也卖得很贵。生活是不公平的。市场是不公平的。客户是变化无常的。

没有最好，只有更好

当然，抱怨不公平的市场和变化无常的客户是可悲的。微软传统上有两种方法来处理客户的反复无常：开发必须有的新功能去讨好客户；还有就是参照竞争对手。

遗憾的是，随着软件市场的不断成熟和扩大，客户不再一味地追求功能，电脑的使用变得越来越不可或缺，这使得客户越来越保守。现在，客户经常把更多的功能和更多的麻烦等同起来。他们提出的让人惊叹的建议是："让我们已经有的功能正常工作。"

即使与对手竞争这种策略有它的不足，但因为你知道客户真正的需求，所以这种做法很有效。遗憾的是，当你追上了竞争对手，你又回到了毫无头绪的状态。你费劲赶到前面去，仅仅是再一次落于人后。

改变一下对你有好处

我们怎样才能打破这种循环呢？其实很容易，答案就是设计师和架构师。我说的设计师，是指参与设计用户体验的人——解决 What 的问题。我说的架构师，是指参与设计完整的实现方法的人——解决 How 的问题。客户和业务需要解决 Why 和 When 的问题。

质量有两面：设计和执行。在微软，我们执行得非常好。尽管我们还有一段路要走，但我们无时无刻不在提高我们执行的质量。

> **作者注：**质量是由体验设计和工程执行组合在一起形成的，这个发现是我几年来收获的又一个重要的智慧灵光。人们常常不会把两者区分开来，或者要么只考虑体验，要么只考虑工程。

但如果你开发出来的产品没人关注，完美的执行又有什么价值呢？你需要杰出的设计，也就是设计师和架构师制定下来的规则。

优秀的设计师能够真正地理解完整的客户体验。他们深思熟虑，善始善终，最后设计出一个超越硬件、软件、网络以及其他技术边界的解决方案。他们唯一关心的是如何感动客户。设计师需要关注的关键方面是简单、顺畅、关键案例、与原方案的不同之处、客户的约束和感知价值。

优秀的设计师做了一连串的梦。优秀的架构师一一把它们变成现实。

你误会了

很多开发者认为，架构师只关注图表、抽象和深层的思考，他们对实用性、效率和执行漠不关心。他们见到的架构师大都痴迷于纯粹和典雅，而不会放下身架，混迹于代码中。自鸣得意、脱离群众的架构师还有另外一个名字——"闲人"。

优秀的架构师紧扣现实的脉搏。这也是他们区别于设计师的地方。然而，最好的设计师和架构师都能解放思想，但又实事求是。架构师把设计师的一套想法具体到各种可执行方案，每种执行方案有各自的优缺点。

贯彻执行

优秀的架构师尽可能长时间地将他们所有的方案保持公开。当他们认真考虑如何用当前的技术实现设计师设想的时候，他们的方案就会受到制约——这些制约包括成本、性能、安全、可靠性、法律考量、合作关系和依赖。最后，很少的方案能够保留下来，而一个最佳的高层设计实现（也称为"架构"）也就呈现出来。

为了实现设计师的设想，可能涉及多个产品和组件。架构师的下一个任务是，要清晰地把架

构与它们进行接合。这里有两个重要的因素：接口和各模块之间的依赖图。通常，现有组件和接口必须重构，以避免循环依赖或行为变异。

很自然，最低层次的依赖必须是最稳定的，并且要首先开发。为此想出切实可行的方法是架构师工作的一部分。通常它意味着首先要更新接口，并且不管接口下面的实现再怎么调整，也要保持接口本身的稳定。

> **作者注：**尽管这听起来有点自以为是，我还是要说，“上面三段值得再读一遍。”它们涵盖了卓越架构师遵循的步骤，然而大部分其他的架构师却没有这么做。

接下来，开始雕琢

不管怎么样，在开始写任何一行产品代码之前，设计师和架构师必须预想并描述清楚实现客户价值的路线图。唉，在行动之前，仅仅这么一个想法，我就听到一些傻瓜叫嚷着反对：“废话连篇！”要知道我的想法是建立一个路线图，而不是一个完整的艺术复制品。通过了解你想要实现的目标，并且认真思考容易犯的错误，实际上，你可能已经有机会做出一个切合实际的解决方案了。

因为路线图不是真实的东西，留给产品团队的还有大量的工作要做以对设计进行充实并执行。那么，设计师和架构师在那段时间里干什么呢？接手下一个项目吗？或者画更多的图片？在会议上发表论文？不，别犯傻，那将会是个灾难。

设计师和架构师必须促使产品团队团结，并且解决他们提出来的所有冲突。没人有万能之脑把事情想得面面俱到，你也不应该浪费大量的时间试图做到这一点。优秀的设计师和架构师都能很好地实现平衡，以他们知道的东西作为基础，然后与产品团队一起解决剩下的问题，让答案自然而然地产生。

这意味着，设计师和架构师必须具备很强的沟通技能。他们不仅彼此之间必须很好地合作，而且与产品团队也要密切配合，以传递一致的信息和指导方针。他们的心思也必须足够缜密，以便在复审实现细节的时候能够记得住那张全局图。

关键要有正确的工具

除了对未来的设想，设计师使用的关键工具有完整的应用范例和关键质量（Critical To Quality，CTQ）度量，以保持产品团队跑在正确的轨道上。尽管定义和测试完整的应用范例已经在公司里面得到了大量实质的支持，但关键质量度量才刚刚引起人们的注意。它们是一些易于掌握的度量指标，这些指标能树立或破坏客户对质量及价值的印象。可能是性能或可靠性，比如完成某个任务所需的时间或不间断的连接；也可能是可用性功能，比如单次认证或界面的一致性。任何客户关心的质量问题都是关键质量度量指标。

除了架构，架构师使用的关键工具有接口和依赖图（依赖图可能像方块图那么简单，也可能像整面墙大小的组件图那样复杂），以保持产品团队跑在正确的轨道上。这里的想法是，允许成百上千个小团队能够独立地工作，同时又能保证产品全局的顺畅体验。如果没有正确的接口，模块就不能安装到系统中。如果不考虑层与层之间清晰的边界，工作也不可能保持独立。

作者注：小团队独立工作的想法是如此重要，以至于我在本章的下一个栏目专门来谈论它，“美妙隔离——更好的设计”。

打破壁垒

设计师和架构师扮演着不同的角色，但他们有一方面是共同的：广泛、自始至终的工作范围。他们的工作是跨越功能边界、产品边界甚至市场边界去思考，以抓取机会。那是因为客户看不到任何边界。他们看到的是无限的、尚未被满足的可能性。设计师忽略人为边界来产出价值。而架构师跟边界“格斗”，最终征服它以使价值变成现实。

当然，我们可以继续让客户灰心，玩着追逐游戏，或者像往年那样随风飘摇，直到我们意识到所有东西都不能协调工作了。或者，我们可以利用我们在广度和深度上的优势，向客户传递无与伦比的坚实价值。这并不难，但它确实要求相关工种在我们行动之前进行全面的考量，然后确保自始至终地遵照执行。如果我们做到了，那我们将最终击败市场上最强大的竞争对手——我们自己。

2006 年 8 月 1 日：“美妙隔离——更好的设计”

关于我以前的栏目“停止写规范书，跟功能小组呆在一起”（参见第 3 章），我收到了大量的所谓的“反馈”。根据我的评论，设计文档促使设计的出现，把一个健壮的架构从我们很多代码中分离出来。但不知何故，读者产生了我反对做设计文档的印象。也许是因为那个栏目的标题吧！

那个印象是不对的。我赞成设计，反对的是浪费。我相信，如果你跟一个小型的多工种团队在一个共享的合作空间中一起工作，而且每次从头到尾只做一个功能集合，那么你没必要写正式的规范书。在这种情况下继续写正式的规范书是种浪费，应该停止。

但如果你的功能集合很大，以致你的功能团队人数超过 8 个，怎么办呢？如果你的团队分散在多个楼层，或者甚至分散在七大洲，怎么办呢？如果你首先设计一组功能，然后一起实现它们，最后再一起测试它们，怎么办呢？所有这些情况都要求正式书写的规范书。否则的话，随着时间的推移或距离的拉大，你无法让你的团队保持在同一个状态上。

分解是件难事

架构。现在我可以听到一些天真的反对者开始叫嚣了：“这么说来，你的整个栏目都没有意义。真正的产品中，没有哪个功能会如此小或如此独立，以致它能被少于 8 个人的团队在隔离的环境下完成。”架构。“其实你跟那些敏捷极端分子差不多，”一些死板的“恐龙”们继续说，“他们谈论的只不过是一个不错的游戏，但我们在微软所做的事，是给大客户做大产品，解决的都是大问题。”架构。

有没有一种方法在大产品和小团队之间的缺口上架起一座桥梁呢？答案是肯定的。那就是“架构”。架构最重要的一点，就是它能把难以处理的大问题分解成便于管理的小问题。如果处理架构得当的话，可以带来我们所需要的“隔离”。

正确的做法

遗憾的是，糟糕架构的例子数不清，蹩脚的架构师也随处可见。“你如何才能把架构做对呢?”这很有意思，你也应该这么问。

尽管创建一个健壮、可靠、有弹性的架构很难，但是它的过程本身却很简单：

1. 收集产品架构必须满足的应用范例和需求。
2. 确保那些应用范例和需求是清晰、完整和独立的。
3. 把应用范例和需求都映射到将要实现它们的产品组件上。
4. 确定产品组件之间的接口。
5. 对组件和接口进行故障和弱点专项审查。
6. 以文档的形式把组件和接口记录下来。
7. 当新的需求出现时进行重新设计和重构。

在我展开充满趣味的细节之前，让我们先来谈谈“后勤”。

团队不需要“我”

我认识一些架构师，他们的生活都是失控的。因为架构天性范围宽广，牵涉的人、工作量都非常多。一些架构师把他们的时间全部花在跟“项目关系人”开会上，然后夜以继日，再搭上周末去做实际的架构工作。尽管这种行为让人印象深刻，但其实是不明智的，而且可能也不具有可持续性。

当然，如果架构师闭门造车而不理会项目关系人，结果也不会好到哪里去。一个好得多的模型是采用架构团队。

架构团队由产品架构师牵头，其成员都是各个功能团队的高级工程师。他们定期开会；在项目早期每天都开，之后也不少于每周一次。

架构团队以一个团体的形式完成我上面列出的所有步骤。其中一个成员负责以文档的形式记录收集到的信息和所做的决议。（那个角色可以轮流担当。）以前总是架构师出差到项目关系人那里去，现在除最高级的关系人之外，其他所有人都要过来接近架构团队。这使架构师极大地减少了旅行时间和随机性。

一些产品线也有架构团队。这个团队由产品线架构师牵头，成员是各个产品的架构师。这种团队结构在覆盖最复杂的跨产品案例和需求方面发挥了很好的作用。

循序渐进

好吧，我们已经知道了所有需要完成的步骤，以及执行那些步骤的人。现在就一起进入那些细节吧：

- **收集产品架构必须满足的应用范例和需求。**这一步应该不费什么脑筋。关键是，要让项目关系人跟架构团队在必要的时候一起复审这些东西。在考虑功能需求的同时，务必也要考虑质量需求（比如安全）带来的影响。
- **确保那些应用范例和需求是清晰、完整和可分离的。**获得清晰而完整的需求给架构团队增加了额外的负担，但这绝不能含糊。（不要忘了性能。）可分离的片断更加微妙。你需要可

分离的需求，以避免循环依赖。然而，你常常碰到多个应用范例基于同一个需求。在那种情况下，架构团队必须把相关的应用范例分解成共享的部分和独立的部分。而那个共享的部分变成了一个独立的需求。

- **把应用范例和需求都映射到将要实现它们的产品组件上。**这一步实际上定义了架构、组件之间的层次关系以及它们的职责。在理想情况下，每一个需求都应该由一个（而且也仅此一个）组件去实现。那将带来完全的隔离，并且实现起来也会容易得多。

在现实世界里，通常是很多组件纠缠在大量需求当中。那些纠缠需求数量最多的组件是你最大的依赖对象。它们必须是最稳定的，而且要优先完成。你还要不惜一切代价去避免循环依赖。这一步的映射工作很艰巨，为了实现它，“公理设计”（Axiomatic Design）可以成为你最有力的一个工具。

- **确定产品组件之间的接口。**一旦你完成了映射，确定接口相对来说就比较容易了。如果两个组件协同实现一个需求，它们之间就需要一个接口；否则的话就不需要。总体来说，在架构这个层次，只有需求驱动的接口才应该被定义出来。
- **对组件和接口进行故障和弱点专项审查。**围绕安全、可靠性、易管理性等方面的质量问题，常常在架构这一层就很明显。在项目早期就把它们定位出来并解决掉（或者只是减轻其影响），可以节省你后面无数的时间。
- **以文档的形式把组件和接口记录下来。**是的，我说过了。你应该为你的架构撰写正式的文档。哪怕你的产品部门跟很多小型的多工种团队在一个共同的领域由始至终合作完成一个功能集合，那些小团队也需要隔离以便能够独立地发挥功能。但他们能否那样被隔离，取决于清晰定义的接口和组件职责。架构是跨越团队、时间和距离共享的，因此它必须以文档的形式被记录下来。

一旦组件的职责和接口通过了争论、设计、纠错并最终文档化了，架构也就做完了。但它只是做完，并没有真正地完成。架构只有等到所有的功能团队都完成了实现，它才算真正地得到了充实。

- **当新的需求出现时进行重新设计和重构。**不管你的架构团队有多聪明或多细致，他们难免会遗漏问题，对新出现的需求反应迟钝。那也是他们坚持每周开会的重要原因。当问题被提到架构团队的面前，架构团队就要返回到第一个步骤，需要的话对设计进行重构。因为架构团队本身是由各个功能团队的高级工程师组成的，因此问题总是能被人发现，并且得到很好的理解。

> **作者注：**由 Anders Hejlsberg 牵头的 C#架构团队仍然每周开 3 次会，总计花费 2 个小时，尽管最初的规范书早在 7 年前就完成了。虽然团队成员的人数已经改变了，但 Anders 说：“在理想情况下，架构团队应该由 6 个人加上一个项目经理组成，而那个项目经理负责记笔记，维护议程。”

猫狗不分家

一旦架构文档化了，组件也被隔离了，各个功能团队就可以免于冲突，忙于正事了。如果组件之间出现了一个无法预料的问题，架构团队也能很快地解决。

不过，仍然还剩下大量的“自下而上”的设计要做。但因为这些设计范围缩小了，影响也受到了限制，你就有充足的空间去试验和应用不那么繁重的像测试驱动开发那样的敏捷设计方法了。

如果你试图对整个产品做“自下而上”的设计，你将不断地面临冲突，于是你需要不断地进行更宽的重新设计才能解决。同时发生的种种冲突，使得项目在它自身大量歇斯底里的作用下迅速崩溃。在一个稳定的接口后面重新构思一个组件是一回事，而在百万行级别的代码库上经常改变接口和职责则是另外一回事了。

通过使用“自顶向下”的设计恰恰足以提供隔离，并且使用“自下而上”的设计可以充分优化协作和质量，从而你就能最好地利用这两种设计方法。通过使用架构团队，你协调了工作，你给高级工程师创造了成长的机会，并且实现了美妙的隔离。这是你力所能及的一片多么美好而安宁的天地啊！

2007 年 11 月 1 日：“软件性能：你在等什么?”

在打排球时你伤了肩膀，所以预约了时间看医生。你进入诊所并排了 5 分钟的队仅仅是为了让院方接待员知道你已经到了。他核对了你的联系方式及社保信息，这些信息是好几年不变的，然后再告诉你到候诊室等待。

你在候诊室坐了 10 多分钟，呼吸着人群中夹杂的疾病之气，抱怨待会离开的时候比进来时病得还要严重，直到一个护士把你带到体检室。

在体检室呆了 5 分钟后，另一名护士进来了，对你的体征做了检测，并再一次要你重复最初预约的原因。10 分钟后，医生来了并告诉你，你的肩膀受伤了。

来看看我们软件的用户体验吧，你一直等着就是为了运行一次软件，此时还要提供一次身份验证，即使系统登录时已经有过验证；再接着等着载入个人环境变量，你再点几次菜单项或按钮来运行你所想要的功能；最后，你还要再次等待功能准备就绪执行你刚开始就想让软件运行的功能，这还是假设网络不存在延时的情况。

稍等片刻

等待是非常乏味的，等待会让你沮丧，等待会让人烦躁，等待说到底就是让人难以承受的黑暗时光。无论在什么情况下，没有人喜欢等待，没有人会要求等待，所以，到底为什么我们要让别人等呢?

然而，为什么我们的用户要忍受这样的事情？为什么我要在我医生的诊所苦苦等待？我想因为所有医生的办公效率都很慢。但是如果我的一个朋友告诉我有一个医生办事效率要高得多，疗效还一样，想都不用想我就会换到这个医生那里去。

这就意味着竞争对手的快速行动可以马上击败我们。在对手行动之前，怎样才能使我们的行动更快？我很乐意来回答这个问题。

> **作者注：**如果你的医生足够出色，你可能愿意忍受这种等待而不是换一个人。但是，请给大家一个理由换人是不对的，特别是当换个更好医生是很方便的时候。

欲速则不达

是的，你希望你的软件性能更高，那该怎么做呢？“我知道，我知道！”我们的性能菜鸟Speedy先生说：“检查你的代码，找出其中耗时的原因，再对一些内部循环进行优化，或对其进行并行设计。”

打住吧，Speedy先生，我原以为你聪明得很。让我们剖析一下诊所，可以吗？哦！结果是医生总是很忙，这就是瓶颈。谁能想得到，按Speedy先生的说法，我们要做的就是设法让医生快些，或是找一个动作快些的医生，或是找两个医生来做一个人的事，这对吗？错了！

我的算法没错——我的意思是，我的医生没问题。如果你设法使她更有效率，但她是不会做到这一点的。事实上，她反而会变得更慢。要完成一项工作需要时间，任何改进的方法或许有些许益处，但同时也会带来诸多问题。我也不希望有另一个医生，偶尔地会很快。我喜欢我的医生，我很懂她，她也很懂我。

我也不希望同时拥有两名医生，即使他们是双胞胎，我永远也不知道哪条线程——我指的是，哪位医生——我会求助。可能他们两个人都会尝试为我看病，他们必须相互间不停地沟通以防犯下错误，他们甚至可能因为互相等待而踯躅不前，这样会越发复杂，实际上根本解决不了问题，即使有两个医生看似会快上两倍，而我还是要经历接待员、候诊室、体检室以及护士等这些环节。

> **作者注：**“但是如果使用多核处理器会怎么样呢？”你可能会这样问。看看吧，只是为技术而使用技术，那应该是在这技术确实有用的情况下。如果客户体验需要跨多处理器的线程，我完全支持；如果不是，那你只是寻求个人的刺激而牺牲客户的利益。

我应该有份拷贝吗

“等会，”Speedy先生说，“你需要的是一个缓存——这将提升速率。”你还好吗？在诊所的“缓存”已让人很头疼了，这是我们慢的很重要的原因，我们有过接待排队的“缓存”，候诊室的“缓存”，还有“体检室”的缓存。

好像诊所的每个人都关心自己的办事效率，所以他们也各自设立了自己的“缓存”。接待员有接待员的缓存，护士有护士的，医生有医生的，结果是病人耗尽了时间在等待，从一个缓存转到另一个缓存，而不是直接交由医生来处理。

觉得这些不会在代码中发生吗？显然，你从没有深入了解过数据库、外壳（shell）及系统调用，所有你为了性能而“缓存”的数据已经通过这些方法进行缓存了。有时，有多少分层就有多少缓存。每个缓存都要有一次存取动作及内存消耗，是的，我已经缓存得够多了。

你还不够机灵

让我们回归正题吧，可以吗？而不是想着使现有的诊所加快速度，边际效率是递减的，让我们站在病人的角度来看待问题。作为一个病人，你或我会希望这种体检是怎样的呢？

以下是我乐意看到的。当我打电话预约的时候，核对一下我的联系方式及社保信息，在预约记录里记下我的病征。好吧，让这些由我在网上完成（等等，这有些不可思议）！

当我到达诊所的时候，我可以径直走向我的体检室（只需要一级缓存），这个体检室早有我

的名字，就像在出租车公司一样，有广播提示："请脱掉你的外衣，确认准备好接受检查，并按下'我已就绪'按钮。"

护士看我已按下"我已就绪"按钮，就走进来，检查我的身份证号，观测我的体征，再按下"体检完毕"按钮，从而把我安排进医生就诊队列中。只要我的医生一有空，她就可以过来询问我的病情。这就太完美了！还有，在体检室里可以放一个队列显示屏以提示病人需要等待的时间，在确保每个医生满负荷的情况下，屏幕提示还可以每小时调整一下预约号以使等待时间尽量最小化。

> **作者注：**如果你没有阅读过《约束原理》或使用书中"鼓—缓冲—绳"的方法来进行优化，你就需接受治疗了。每个对性能有所顾虑并要求有所改变的人，无论是软件行业或是快餐行业，都需要阅读一下这些原理。

你可曾以身试法

这实行起来并不困难。按我说的，改造这样一个诊所是很简单的，也花不了多少钱。它不需要更多或效率更高的医生，也确实节省了不少房屋空间。是的，网上预约及等候时间提示屏需要特定的软件，但这些不是更好更快的服务所必需的，必需的是以最小化等待时间为目的，周全考虑用户的体验。

以下是你要考虑的：

- 你的团队最后一次全面讨论完整的用户体验，包括等待时间，是什么时候？
- 除了给客户提供一个"取消"按钮（如果我们的软件及服务有个"取消"按钮就不明智了），想想客户希望怎样解决这种约束性问题（每种工程过程都存在这样的问题）？
- 你怎样将软件错误、网络延时及设备 I/O 访问的影响减少到最低，使用户的体验很自然而不会很突兀？
- 当关键资源被占用的时候，你会使用哪些度量及统计数据来改善用户体验及最小化等待时间？

现在，假设这些性能约束并不存在，让我们来设计一下用户体验。所有的东西都是模态化并同步的，就好比每个函数都会返回值而且用户也不会选择错误的选项；我们一次设计一项软件功能而不是全部；或者如果我们确实要做一个全面完整的设计，我们只考虑最理想的范例，而不是差不多的那种。再假设异常及延时不会经常发生，甚至不发生。这看起来太天真了，甚至可以说是"愚蠢"。

> **作者注：**当然，微软的软件中有很多例子对完整体验作了深入研究。我将在下一节中提及。

你要有所准备

性能优化是有可能的，但函数及服务的规模必须得以扩展。阻塞及死锁问题必须有特殊考量，有真正的性能优化大师可以帮你解决问题。只是这还不是主要问题。

主要问题是最原始的问题，即用户希望完成某项操作。这包括网络及 I/O 访问，这些交互可能导致失败或延时。通常，用户经历过这些问题，也知道它们的存在。处理它们的最好方法是跟

用户沟通并尽可能地了解什么是用户期望发生的。

如果 I/O 访问异步化，客户可能还会很高兴——就是 Outlook 及 OneNote 采用的提升用户体验的解决方案；用户可能更喜欢运行一个本地复本，然后再按需同步——ActiveSync 及 FrontPage 采用的解决方案；还可能，用户希望将他们的请求放入队列再得到一个状态报告——创建系统（build system）及测试套件（test harnesse）采用的方法。

问题在人身上

关键在于以用户的角度审视世界，并为其设计一种用户体验，使其能预测错误的模式并最小化其对用户的影响。性能应该在体验中以特定的度量及导向来确定，而不是心存侥幸。这通常不需要复杂的算法或缓存，这两种方法有些夸张了。这只要求开动脑筋，多思考。

当性能在体验中得以提升后，这样性能从一开始就成了一种内建机制并经过测试。在项目完工的时候，突然间就会发现你已经是名性能专家了，谁也不用为此惊奇，或耗尽心力地去实现这个目标。唯一惊奇的是用户的表情，当用户发现原来一次苦闷的就诊过程变得很开心时。

2008 年 4 月 1 日："为您效劳"

记得有这么句话："微处理器可以改变世界！" 还不至于，没这么夸张。没错，它改变了很多事情，但是人们照样为同样的问题忙得焦头烂额，忙于应付。微处理器只是在增加难题与解决问题方面同样更有效率。那"互联网改变了世界！"呢？不，也没这么夸张。没错，它也带来重大影响，但是人们同样只是在增加新难题与解决问题方面更有能力了。那我们还有"软件加服务来改变世界！"哦，饶了我吧。

你可能会说："有一点你错了，微软确实因为以上这些改变了很多。"我们是有所改变，但不是很多。如果我们改变了很多，我们早就裁掉或换掉了一大批员工（这可要在别的公司跟你的朋友说），相反，我们不断扩充我们的工具及员工数量以使在每一个新机遇上占得先机。

别着急，我无意贬低这些变化的重要性。它们提升了人们的生活品质并使得世界越来越小而触手可及。它们带来了新的商机，提高了质量及生产力，这太美妙了。但不要老是说新技术改变了世界，因为很多事情无法改变：人、人们面临的问题以及他们期望完成的事情，这只要问问那些只专注于技术而不是客户的公司。哦，等等，你做不到了，他们破产了。

> **作者注：** 微处理器的出现就导致了微软的诞生，这是很自然的事。因此，我认为计数器为这个公司带来了重大改变。但是，我还是同样的观点——什么也没变。人们没有改变，他们仍然需要维护人际关系，忙于生计，为人生及事业而奋斗，唯一改变的是人们以什么方式完成这些事情。

好事还是坏事，我有些糊涂了

工程师们说："什么是软件加服务？"没有什么比这更让我恼火了，或是更气人的说法："这是个崭新的世界，我们必须创建服务！"听到这些都要让我吃进的午餐又反刍了出来。停止谈论

技术吧，傻瓜！问题不在技术，从来也不是，问题在于客户以及他们的诉求。

等等，我听到 Ozzite 大声念叨："但是 Ray 说现在到处是服务与云。"听着，当 Ray 谈论服务与云的时候，他本意讲的是人们期望完成什么事情，以及服务与云能给他们提供什么帮助。最先的出发点都是一样的：客户以及他们的目标。

"哦，所以我们应该关注客户想怎么通过服务使用我们的软件，是吗？"不！请搞清楚！客户不是为了使用我们的软件而用我们的软件，你的亲友们把这个概念弄糊涂了。如果客户想做一件事，就比如写一篇学期论文，或分享他们孩子的画作，或是提交一份订货单，如果服务可以为完成他们的目标帮上些忙，我们才需要提供这些服务。

或许举些例子会有所帮助。我们就来谈谈学期论文、孩子的画作及订货单。

水到渠成

完成一篇学期论文对于学生来说是再普通不过而必须完成的事，因此就有相关业务应运而生从而为学期论文提供付费服务。如果我们想为学生们提供学期论文撰写服务，首先设想一下一篇学期论文的撰写过程是怎样的。我们都有过苦涩的撰写学期论文的经历，所以做这个服务并不麻烦。

好了，现在指导老师给学生们布置了一篇关于 2008 年北京奥运会筹备工作的论文，此时我们的一个学生，Stu，打开文字处理软件开始构思一份粗略的纲要。"这么着"，Stu 说，"我先从什么是奥运会开始，然后是奥运城市竞选过程，再或是北京的历史，接着是一些北京竞选方案的细节，再接着是他们筹备的计划，最后以目前的进展情况为结尾。"纲要还不错，Stu。

于是，Stu 点击网页上的搜索按钮，则纲要中的关键词会加亮显示，再点击关键词 Olympic Games，则一个关于这个运动会的搜索结果对话框弹出，Stu 再从头到尾滚动屏幕进行预览，选择了一些他需要的段落，并直接把这些文字插入到他的文档中。然后他对城市竞选、北京及其他部分重复这样的过程。在每一个步骤中，搜索范围不断地从上次结果中缩小，记住 Stu 关心的是 Olympic Games。

接下来，Stu 将对这些文字进行编辑以使其看起来像是他自己原创的，并让人觉得他已深谙论文的内容。语法检查器除加亮显示拼写与语法错误外，还对与原始引文很接近需视为原文处理的段落加亮，当对这篇文章修正了所有错误及太露骨的剽窃后，Stu 把论文发给了他的指导老师。

我很希望我的学期论文就这么简单。那么，微软已经拥有哪些环节的软件功能了呢？我们有文字处理软件，我们有互联网搜索引擎，我们还有语法及拼写检查器以及 E-mail 工具，我们还需要的是一个剽窃检查器以及为 Word 增加一个客户端对话框集成 Live Search，它可以下载经过参数筛选的搜索结果。这并不难，呵呵，软件加服务实在太有用了。

确实，如果 Stu 将他的学期论文放到云端并通过多种设备（包括他的汽车及手机）来访问，那就更方便了。但是你并不能也这样做，因为玩转云与手机并不是那么容易的，你只是得益于 Stu 以及 Stu 想完成的事情。

> **作者注：**耐人寻味的是，可能你并不想就这么简单地撰写学期论文，你的论文应更具开创性并能集思广益，而不是网上搜索一下，然后编辑一下。检查剽窃是有用处的，然而这使得堆砌一篇论文显得过于简单了，或许应该放弃完整范例方式，转由指导老师布置一个富有挑战性及建设性的论文，我日后会把它当做一种练习。

留住你的记忆

我们都知道照片共享。在电脑上接上摄像机，上传照片，然后单击按钮（这个按钮应该有）分享照片。没错，在 Windows XP 上就有个链接可以“将所选项发布到网上”；而在 Windows Vista 上有一个“共享”按钮，但它不是通过 Web 共享的。在一个开放的市场上以及与合作伙伴合作的时候，我知道这种功能很难实现。这是个不错的理由，但不是个好借口。

即使在 Windows XP 中，假设用户可以找到“将所选项发布到网上”的链接，并明白其意思，可你试过吗？这么做只是为了让 Windows XP 完成这次操作而为其提供所需要的信息，却不是为用户出于分享照片的目的而提供信息。这就是为了提供一个直观易用的软件加服务功能所需各种要素的一个范例，但是我们设计的却是为软件服务的体验而不是为用户服务的体验。

> **作者注：**我的真实意图并不是无视创建一个照片共享功能有多么困难，Windows XP 的做法早先是出于处理复杂的 Web 交互过程，因为政府对微软施压要其不能在操作系统中集成过多的功能，合作伙伴也通常互不买账，彼此间也不分享，但关键是不要让这些顾虑影响到我们完整用户体验的设计，只要我们了解了这个动人的体验过程，我们会尽力想办法解决这些现实的约束。

自动完成

现今，人们大多在网上购买生活必需品，在谈论客户购买体验时，先让我们讲讲在线零售商订单完成的过程：

当一个客户提交了他的订单，销售商肯定要经历以下步骤：

- 接收来自银行信用卡的认证信息。
- 将确认页面通过 E-mail 发给客户。
- 核查商品库存单。
  - 如果库存单中有这个商品，则把出货指令发到仓库以将商品寄给客户并更新库存单。
  - 如果库存单中没有这个商品，就从供应商那边订货，同时附上说明将商品直接寄往客户，并跟踪记录交易过程以供日后进货数据分析之用。
- 将发货信息通过 E-mail 告知客户。

这些流程看似很简单，我们同样拥有所有的组件——用于交易的 Web 服务器，用于页面确认的 ASP. NET，用于 E-mail 的 Exchange，用于库存的 SQL Server，以及用于商务逻辑的 Dynamics。可关键是，同时实现这些环节是说起来这么简单的吗？当我们的零售商尝试这么做时，他们必须得应付这么多软件吗？

记住，用户相信微软所有的软件来源于一个地方，而且比尔·盖茨编写了所有的代码。他们不明白为什么不同的应用软件使用起来会这么多不同，他们无法做到像一个专家一样对每个不同的环节进行设置。软件加服务并不意味着更新更高效的代码，而是将重点放在用户希望做什么，并能轻易解决他们面临的种种技术难题上。

作者注：企业用户通常希望由他们自己设计完整的解决方案，或通过咨询能透彻理解他们业务细节的人来解决。然而，我们仍然需要设计那些完整的范例，企业用户及他们的合作伙伴若能更快更方便地了解他们的软件流程，则他们受益就更多，他们的维护费用就会更低，而且他们会因此更感激我们所做的工作。

我们在同一条战线

好吧，你是对的，软件加服务是有些地方会有改动。服务不是刻录在 DVD 上的静态镜像，在数据中心，它是鲜活的，它是伴随着你的每一天的开始而开始的。当你修复补丁、升级或者恢复一项服务，任何其中一件事情发生时，服务仍然有效运行着。Debug 调试是相当困难的，所以你事先需要设计好服务以便在遍布于全球的机器上诊断错误。

所有这些说明设计一项服务以及使用这项服务的软件，必须全方位考虑其可恢复性、可维护性及易于管理。一旦服务得以发布，你就不能全身而退了，开弓没有回头箭，也注定了你必须每天不断维护升级你的服务。

如果我们沉迷于技术与软件功能这没什么，然而却会失去用户的关注。用户想要的不是软件加服务，用户要的是实现目标，而设计优良的软件加服务可以帮他们。

对于软件加服务来说真正的要点在于能跨产品跨集团工作，使用户业务按他们所期望的方式正常运行，而不是我们的公司这样的组织方式。好好学，一切尽在你手中。

作者注：在本章前面的专栏“质量的另一面——设计师和架构师”中你可以了解到更多关于如何在跨产品跨集团的情况下完成完整的业务逻辑。

2008 年 8 月 1 日：“我的试验成功了！（原型设计）”

夏天到了，该去晒晒太阳做个白日梦了，这可以是在放假的时候或是一个周末的晌午。当你的思绪如此放松时，通常一个美妙的好主意就会浮现脑海。如果时间合适，可能是在下一个开发周期开始时，你就会对这些主意作更深入的思考，开始形成这些奇思妙想的原型。没等你反应过来，你的妙想就绽放出了罪恶之花，曝晒而死或是更恐怖的噩梦降临使新生代的工程师诅咒你怎么能来到这世上。

哦，除非那是个童话，多半情况下，奇思妙想的原型往往越变越恐怖，一堆令人发毛的杂碎可并不容易把持、重构或是理解。为什么？这是什么原因？

并不是说你应该把这种原型仔细地用代码写下来，不管是通过单元测试还是其他某种测试——你做不到；也不是说你应该把这种原型一弃了之——即使你会这样做。都不对，问题在于你对原型的所有理念根本上就错了。

放飞梦想

通常，当工程师想到一个主意时，他们会编写一个原型。这是个大错，不应该干这样的事，这会将你的妙思带入不归路。你要明白，你不能只编写一个原型——你应该编写大量参数化的原

型来进行上百次的尝试，所有的原型设计只为解决同一个问题，但是从不同的角度。

其他所有的研究领域都是这样，不只是做一次试验，尝试一种方法，或是就采用你一开始时的猜想，你要做上百次的试验。画家及生产商称为“放飞梦想”。你能想象医学家只对他的想法作一次试验就能治愈疾病吗？你会不会认为那很白痴？你没问题吧？

好玩极了

当然，你不会有用不完的时间来编写成堆复杂的原型，很好，你没有这样。原型设计并不像产品工程，它更像做试验，它应该内含了类似胶带、橡皮泥、铁丝架这样的软件相仿物，如 VBScript、WordArt，以及 Perl，你应该用几个小时而不是几天的时间就能倒腾出一个原型。

如果编写一个能说明问题的原型原本用不了多少时间，而你却花了很长的时间，那你就相当于滑落踏板一头扎入鲨鱼群那般死无葬身之地了。用过多的时间纠缠于一个原型不仅仅分散你寻找另一个解决方案的精力，而且过多地把时间消耗在一个原型于其毫无益处，而你却最终会把它当成产品代码的基础。你就等着失望与无助向你招手吧。

相反，原型应该尽快出笼，不要让人以为它们就定型了。原型是用来尝尝鲜的，是容许犯错误的，是不断修正的，这样才能获得真知。如果你就编写了一个原型，那你什么也没得到，除了证明你是个故步自封、无知又糊涂的傻瓜。

高能发动机部件

等等，说“我无能为力”的人急了，他们说：“没有架构、套件及运行库，你无法很快弄出原型，创建这些工具需要很多时间，在工作进度要求这么急的情况下创建这么多的原型是完全不现实的。”省省吧，如果你受聘为一名码农，你就认命了吧。

话说回来，你要清楚地认识到原型无需照搬产品代码的模式，它们可以用很多种不同的代码编写，并在不同的平台上创建。可以使用些小技巧或快捷方式，或是脚本、动画，或给这东西起个名字，再有个简短的防盗版序列号，这样的原型就很好玩了。

大量的运行库、成堆的套件及架构足以填充整个足球场的资源来帮你快速进行原型开发。你所需做的就是走出这些条条框框，进入满目斑斓的世界。

你还有种选择

像搜索引擎这样优秀的设计就有赖于尝试大量的奇思妙想，同时需要精诚合作，与他人探讨新想法及指导方案。如果需要一个用户界面，你就需要用户体验设计者来给你指引；如果需要运行库或 API，你就需要架构师或客户来为你出谋划策。

> **作者注：**用户体验工程师既是设计者也是软件可用性专家，这些人经过了设计创意培训，恰是原型开发项目的最佳人选。他们中很多人还专于界面美工及软件可用性设计，这些设计对 API 及运行库的设计大有裨益。

当你有了很多想法，将其原型化，不断改进再得真知，这样你就有了很多种选择，你就可以：

- 选一些你心仪的想法。

- 通过一个简单的工具如普氏概念选择矩阵（Pugh Concept Selection Matrix）慎重考虑每个方案。
- 综合各种想法的优点，再进行试验。
- 不要急着做决定直到必须做选择的时候。

尽量不要急着下设计定论，直到时间所限必须将手头讨论方案作为“基础”设计方案。不要放弃你手头的任何一个设计方案，当需要的时候就从中选择一个，因为在认定所有可选方案之前，往往时间表里的进度要求就已到了，你最终还是要重新使用一些类似 Pugh Concept Selection 这样的工具。在你这样做之前，“基础”设计的构思仍有很大的改进空间，这将有助于你选择一个最理想的解决方案。

作者注：普氏概念选择（Pugh Concept Selection）采用一个简单的表格对可选方案按需求的符合程度分级排序，对照你的默认选择，级别可以是正数、负数或者零。需求是按重要性程度评级的，级别最高的可选方案将被采用。

在线资料：你可以在网上找到一份这种电子表格的示例（Pugh Concept Selection Example. xlsx）。

扔掉它吧

当你最终选定设计方案，那些用“胶带”与“箍丝”做的原型就会显得很不牢靠而令人费解，没有人再愿意理它了，最多作为念想。这样的结果就是我们所要的。

把匆匆凑成的代码作为产品开发的基础将导致软件极难维护，也极易失败，只有进行改造以符合质量要求，如自动测试、代码复审、全球化、易用性、安全性、隐私保护、可管理性及性能——不一而足。改造的结果就是我之前所说的令人毛骨悚然的杂碎，而不是一开始就引领你一路而来的美妙创意。

扔掉你的原型，只将它们当做构思与算法的参考，而不是代码的参考。这并不难，因为每一个原型只用了你一点点的时间。

冲动，如影相随

我知道有难以抗拒的冲动使你要以高标准的方法编写可靠的原型码，就如有难以抗拒的冲动你要用又急又糙的方法来快速编写产品代码一样。你必须尽一切办法防止这样的冲动发生。

你在试验期间的行为及编码方式要与产品开发期间的方式不同，众所周知，人们并不理解这一点，这些人被称为“小孩”。

如果你把原型代码当做产品代码来看待，或者相反，那对你这人就要重新看待了。如果你的上司认为你编写原型应当与开发产品一样，或是把产品代码写得跟原型一样，那他该被解雇了，我可没有言过其实。

产品开发的最终结果应当是按时交付的产品，且符合高标准又具有赏心悦目的用户体验，这就是项目经理在产品开发期间应当赋予你的任务。

最终的原型设计应当是见地独到、独具创新的，它将改变你对产品、服务和用户的认识，这就是项目经理在原型设计期间应当赋予你的任务。

善待自己

最后一点是，如果你只为你的创意编写一个原型，那你就对不住你的创意、你的团队、你的公司以及你自己了。如果你不对你的创意的所有内涵进行挖掘，不管是好的或是坏的；你不寻求一种新的方式来实践你的创意或将它延伸开来；你也不挖掘你创意的潜在客户或是深究这个创意与其他的有什么关联之处，则你所有的仅是你最初的一个猜想，你背弃了你自己以及你的创意。

记住，实现多个原型用不了多少时间——只是心态不同。是的，当你要开发产品及服务时，你可没这样的心思。但是，在计划与试验阶段，适时凑合些试验对你“放飞梦想”是很有好处的。谁知道呢？或许，你可以在你的成功经历中略见一二。

> **作者注：**最后一个对于原型的看法是我的一个朋友跟我说的。有些原型是“概念”上的原型。概念原型很特殊，他们在苍穹之上，空中楼阁式的原型意味着在概念化的原型上可以有些作为，它们不代表很快就可以变成一个实在的产品或服务，它们的作用是看看你可以采纳哪些天马行空又切实可行的创意，这或许将为下一个或下下一个版本指明一条可行之路。

2009 年 2 月 1 日：“绿野中长满蛆了”

就如我在第 5 章的“盯紧标称”中说的，一个大型项目成功的关键是“三思而后行”和“准确定义完工的概念”。“三思而后行”是对于设计与计划来说的，“准确定义完工的概念”指的就是设定一个质量标准并遵照执行。但是很多大型项目走错方向，即使人们了解“三思而后行”以及“准确定义完工的概念”，为什么？

经常失败归咎于糟糕的执行决策，这样的决策将其工作日程要求置于产品发布期限之上（但如果如你所愿达到了你的质量标准，这样发布产品确实是要好些——好很多。）然而，失败的另一个更常见的原因是由于工程团队过于深思熟虑及过于注重通用性——想着解决世界性饥饿问题而不是对眼前的小孩给予施舍。

> **作者注：**如果你的工作达标，为什么产品发布总是比在代码上纠缠来得更为重要？必须承认，在最后时刻，执行官可能还认为有些不错的功能要加进去。这会有多糟糕呢？哦，实在是太糟了。
>
> 在产品发布前，任何事情都是未知数——没完没了的 Bug 修复，延期的关键功能开发，以及还没定论的设计决策，这些在产品发布之前都有不确定性。而一旦产品发布，你就会明白——你可以不断改进，否认这种事实而一味胡思乱想毫无益处，这将使你的客户及合作伙伴无所适从，没完没了的争论只会彻底毁了你的代码。

把客户的问题当成试验很具诱惑性，是个非常好玩的智力游戏，但也很自私，你在不断解决问题的同时，将不断地发现更大的问题，不断看到更多的问题，这些问题足够所有的国家，所有

的人，用所有的时间来解决。哦，放了我吧，也放了客户们吧。在广袤的田野上到处播种你的创意不仅仅自私，且这种重功能轻价值的产品与服务的药方，你的客户会吝于善用、懒于理会，而且还会欣欣然地将这些创意弃之于萌芽。绿野中都长满蛆了。

不寒而栗

但是，当我走在走廊上时我仍然能够听到人们在谈论创意、算法或是类的结构，说："这个类可以解决这个问题，这对所有问题都适用。"罪恶啊，罪恶！严重警告！请注意，这样的想法很"罪恶"！

通用性的解决方案有什么罪恶呢？确实，你的代码可以当做地板蜡也可以当做饭前甜点，但它有三大基本缺陷：

- 你很少能在一个产品周期中只采用一个通用的解决方案，未完善的架构不会是完全准确的，但是你已经把产品发布了，你现在已经受制于一个无法正常工作的基础代码了——永远。
- 你带来了更广泛的测试范围及更广泛的安全攻击威胁，任一种情况都不是人们所期望的。
- 你以问题为中心，而不是客户，当客户不是中心的时候，你的代码就失去了灵魂。这样就没了惊喜，只有中规中矩。

> **作者注：**为什么未完善的通用架构不是完全准确的？因为它不可能预见你及你客户的所有要求。毕竟，你只是老老实实地解决一个常规性问题，而不是聪明地解决一个特别的问题——一个不断反复并走上正途的机会。

你让我不再狂热

在我们摆平通用解决方案的这三个缺陷之前，我先要安抚一下我的"敏捷狂热"读者，因为敏捷方法促进了软件与客户的互动，它们极力避免了只想不做的误区。比如，测试驱动开发（Test-Driven Development）及自然设计（emergent design）倾力于使解决方案尽可能简单，并时刻专注于用户的需求。

因为敏捷方法避免了通用解决方案及想法的缺陷，很多敏捷狂热者相信所有事先的设计不足挂齿。他们错了。没错，自然设计具有的定期重构及返工特性对于小型代码库来讲不是什么问题，但是，当你对 100 000 行以上的代码进行大规模返工时，这样的耗费就大了。这就是为什么以客户为核心、事前架构设计对于大型项目很重要的原因。

> **作者注：**Barry 博士在 2004 年对此做过研究。他们发现，在 10 万行代码量以上的项目中返工成本远大于事前设计的成本，项目规模越大，越完善的事前设计越能节省你的成本。

好消息是很多敏捷方法，如 Scrum、持续集成及测试驱动开发，在一个有着以客户为中心的架构的大型项目中非常有效。这些技术可以使团队专注于客户而不是迷失于田野玩着自我满足又毫无价值的智力游戏。

作者注：测试驱动开发是一种极限编程及敏捷开发技术，它使设计得以实施，得到紧凑、稳定的代码，同时，它们能给单元测试带来更高的代码覆盖率。过程很简单：

1. 为 API 或类写一个全新的单元测试。
2. 编译并创建你的程序，再运行单元测试，并确认失败。
3. 写足够多的代码使新的单元测试得以通过（老的也一样）。
4. 如有需要，重构代码去除重复部分。
5. 一直重复，直到所有 API 或类通过测试。

谁来拯救你的灵魂

好的，回到解决方案的三大基本缺陷上来——不成熟、无效的继承代码；扩大化的测试面与安全攻击可能；迷失了你的灵魂。

绿野从通用架构开始，这种架构可以适应各种武器装备，各种风光景致，以及各种内容查看器。堆砌一个游戏或一个网页就跟选择一组你喜欢的武器、景致及内容查看器一样。太棒了！你已经完成了一个游戏或网站，还像模像样。

但这游戏或网站并没有灵魂，因为你把重点放在了架构上而不是内容上；另外，为了检验这个架构你必须慎重考虑每种武器、景致及内容查看器的组合——任一种组合都是黑客的攻击目标。而且，你会傻到把这个架构暴露成为一个“可扩展的接口”吗？所有这些问题都将使问题本身以数量级增长。

作者注：你仍然需要有架构来帮助你组织你的代码及类，但架构不应该是一种通用的模式，它们应该依具体的客户要求而定。

哪位如果玩过《最后一战》（Halo）或是将最新的 SharePoint 与最初的版本比较的话，会知道专注于客户需求的价值。

没那么简单

更糟糕的是当你设计并开发《最后一战 II》（Halo Ⅱ）及 SharePoint 2.0 时，不可否认的是，在前一版本中还有很多顾虑没有解决，这些顾虑使架构的相当一部分未达效果。遗憾的是，这些部分已经创建并发布，因此你只好保留与旧版的兼容性而收效甚微，你还自认聪明，看重通用性问题而不是客户及客户的诉求而洋洋自得吗？

“但是架构及可扩展接口对于我们的平台非常重要！”没脑子的智障小子叫嚷着。是的，当客户是开发者，而客户需求早已内置于我们的平台时，它们很重要。然而，《最后一战》及 SharePoint 的客户根本不是开发者，他们是消费者及企业用户，他们的诉求包括推翻星盟并共享信息。请把心思放在客户身上，而不是架构。

我可以给你讲个故事吗

专注于客户及其诉求是什么意思呢？意思是不要解决一般性问题——解决用户的问题，也就是用户希望解决的问题。

这意味着要领会用户的需求（范例）。用户是谁？他们要达到什么目的？他们是否也曾如愿以偿？我们的软件对他们有什么帮助？在客户看来这软件应该是怎样的？

记住，我们的很多用户都是开发者，（我们所要做的）对于他们来说并没什么不同。他们是谁？他们的目标是什么？开发人员通常是怎么达到这个目标的？我们的软件会有什么作用？如果开发者们使用我们的平台及工具，它们应该是怎样的？

当你把重点放在客户及其诉求上，放在设计、开发、测试及为那些用户实现他们的诉求时——就这些，就会使客户需求变得很吸引人，并且皆大欢喜。

依此而行，一种通用架构出现了，别接近它们，除非它们对你的客户需求有用，只有工程师需要实现客户的需求，所有通用化的东西都是徒劳无功的，因为当你需要这些东西的时候，客户需求又不同了。

> **作者注：**你在软件开发这事上最好信守 YAGNI 的哲学理念——只在你需要的时候再按需而为，不要对下一版本中可能需要的东西付诸实施。YAGNI 意为“你不需要”，或许有些时候这样说有些不文雅。

诱惑总是存在的

因此，先完成一种用户体验模式（场景），然后开始着手下一种，这很可能就是下一个版本。不想包罗万象，八面玲珑吗？不，不，不想。如果包罗万象那你就错了，那样你就无所适从。

而当你开发下一种用户体验时，你就会知道需要加进新的功能、新的设计或是内容查看器。重新设计，加入你现在需要的功能，对于这些功能是什么以及为什么需要你会心中有数——这样就好办了。其结果就是更优良的、更精简的、经过充分测试的代码，因为测试可以把重点放在用户体验上。

注重用户及其体验，在理论乃至实践上都不是难事，但这样做仍需有个度。欲壑难填，要把控好这种冲动，你不可能让所有人都满意，所以尽你所能至少满足客户的需求，投之以桃，即可报之以李。

第 7 章

职业生涯历险记

本章内容：

2001 年 12 月 1 日：“当熟练就是目标”
2002 年 10 月 1 日：“人生是不公平的——考核曲线”
2006 年 11 月 1 日：“职业阶段上的角色”
2007 年 5 月 1 日：“与世界相连”
2007 年 11 月 1 日：“找个好工作——发现新角色”
2007 年 12 月 1 日：“要么带头做事，要么唯命是从，要么赶紧离开”
2008 年 7 月 1 日：“猩猩套装中的机遇”
2010 年 3 月 1 日：“我是很负责的”
2010 年 4 月 1 日：“新来的伙计”
2010 年 6 月 1 日：“升级”
2010 年 9 月 1 日：“辉煌时代”
2011 年 1 月 1 日：“个体领导者”

我认识并敬佩的大部分工程师并非都有满腔抱负，只要能给他们实现想法并给这个世界带来些积极影响的机会，一个工程师这样最简单的头衔以及对他们的些许赞誉，他们就会很开心。确实是这样的，我真不骗你。

遗憾的是，即便是最低微的工程师也必须积极地规划他们的职业生涯，或是正确对待错过机遇的风险，而这种机遇恰是他们一直所追求并能产生积极效应的。这并不是因为世态炎凉，尽管有时候是这样的，而是因为这个世界集聚了太多渴望功成名就的天才人物，如果把你的一生压在命运上，那就真是太傻太天真了。

本章我将与大家分享一个愉快、成功又无可挑剔的职业生涯背后的秘密。第一个栏目是告诫管理者，抱负与价值之间没有联系；第二个栏目揭示了如何在充满竞争的世界中出类拔萃；第三个栏目解释了角色与职业抱负之间的不同；第四个栏目谈论了人际关系对于效率与目标的重要性；第五个栏目论述如何重新寻找一个新职位；第六个栏目论述如何开发战略性眼光；第七个栏目论述如何抓住并利用机遇；第八个栏目论述如何写个虚假合约；第九个栏目将一步一步指导你如何提升一个团队的工作效率；第十个栏目将为你描述所有的职业阶段；第十一个栏目揭示了成为一个执行官要面临的权衡；第十二个栏目引领你成为一名高级人才。

工程师面临的最基本的职业生涯问题是：“我必须成为一个管理者才算成功吗?”简单来说，答案是：“不对。”如果你的管理者对你说这种想法是错的，那就是他错了，不要理他，要想走向成功，

你应该换到一个管理者能力更强的部门去。如果你的上司建议你尝试一下管理，请考虑一下他的建议，管理可能报酬更高，更能实现自我价值，你的上司可能看到了你在那个领域未被挖掘的潜能。然而，它只是个选择，而不是要求。大量的员工以及我自身的经历，还有我众多朋友的经历，都支持我上述的观点，你是你自己职业生涯的主人：由你决定，由你推动，并设定一个你力所能及的目标。

——Eric

2001 年 12 月 1 日："当熟练就是目标"

我们所有人都必须成为超级英雄吗？这是微软的方式吗？当一个开发人员整年夜以继日地为产品的发布而劳碌时，是不是还有人说："做得还不够。""吃人"的职业生涯恶兽到底怎样才能得到满足呢？

很多人，包括我自己在内，都相信并断定你总是能够做得更多，做得更好，达到更高的境界。我们常说，永远也不要满足于现状，要时时鞭策自己和团队，向更高的层次进发。我们把这种态度跟诸如自我发展、力求成效、专注技术此类的"微软特质"（Microsoft Competency）联系在一起，这是不是意味着，如果你满足于你现在所处的位置和正在做的事情，你就是公司的寄生虫了呢？

每个人都要明白自己的局限性

如果你认为出色但安于现状的人是寄生虫的话，你就太傻了。不是所有人都能或是想成为比尔·盖茨的，即使我也一样很喜欢这家伙。我们都有自己的局限性及优势，大多数人如果想要成为一个副总裁都是不现实的。

呵，你们这些夸夸其谈的人肯定会这么想：放低你的眼光也就是降低你的期望，降低期望也就导致平庸。平庸是种毒害，它会在公司内滋长，影响并蚕食这个公司，化成一个毒瘤。说得太对了！

但是，如果说一个具有多年产品发布经验，对产品富有激情，力求成效的开发人员除非成为一名架构师或管理者，否则他就会没有地位，这就不对了。我们确实认识这样一些人，他们热爱这儿的工作，他们热爱编码，热爱软件这行业，热爱日新月异的技术，热爱着为我们的社会带来正面积极的促进作用。

开发者有这样的态度及抱负并没有错，他们是公司的栋梁，他们是鞭策着我们向前并取得成功的基础，我称这些无私的人为熟练开发者（journey developers）。（因为熟练工（journeyman）不够文雅，你知道，我很文雅的。）

> **作者注：**这里的"文雅"是指 Politically Correct，简写为 PC。顺便说一下，本专栏是我写过的最广受称颂的一篇。

> **译者注：**上述"熟练开发者"的英文原文是 Journey Developers，而"熟练工"的英文原文是 Journeyman。这几年来，美国人为了确保 Politically Correct（文雅），纷纷改变了以往对社会上弱势群体的称呼，比如将听起来不事生产的 Housewife（家庭主妇）改为较好听的 Homemaker，将 Janitor（打扫建筑物的人）改成很专业的 Sanitation Engineer（卫生工程师），甚至将 Dish Washer（洗碗工）美称为 Utensil Sanitizer（餐具消毒师）。似乎有点矫枉过正了……

有所成但不坐等

不要把“熟练开发者”与“守株待兔者”混为一谈，坐享其成者已没有了对产品的激情，没有了对客户承诺的动力，以及力求成效的决心。熟练开发者仍拥有所有这些品质，仍在为了生产这个星球上最好的软件矢志不渝，而把对人员、技术和设计的领导权交予了他人。或许有一天他们会改变他们的想法，而不仅仅是作为一种表率或体现一种经验价值，但在这之前，他们对生产力的贡献，他们的奉献精神，以及他们不可估量的价值必须得到支持、回报及鼓励。

> **作者注：**“守株待兔”是对一些员工的内部称呼，这些员工不再努力工作，他们到处游荡，就等着把他们的股票期权套现。自从互联网泡沫破灭之后，这个术语其实就不再那么适用了。

我希望他们给我以正确的定位

要成为一名熟练开发者，就要为开发人员开辟一条成长路径，就如专家/架构师与主管/经理的路径一样。要限制熟练开发者的领导权，势必会限制他们的影响力，意思就是说熟练开发者很少能达到 SDE 63 级以上。

> **作者注：**在美国，一名软件开发工程师（Software Development Engineer，SDE）的入门级别是59 级（不同的地区薪酬水平会有所不同），63 级被认为是“高级”工程师，比这更高的级别就可以成为“首席”（Principal）或“合伙人”（Partner）。

有些人可能会说，这样就足以使人们对这样的职业路径灰心丧气，这样的想法就太看低我们了。我们需要他们的奉献与经验，我们不能把我们最好的开发人员拱手让于一些二流公司，我们的薪酬水平已足够留住我们的熟练开发者了，我们还可以很容易找些更具新意的方法来回报他们的努力，并同时防止强加给他们超越级别的职责。

当然，一些熟练开发者可能会改变他们的职业路径成为我们未来的领导，对于这些人或一些新手，如果希望有这种灵活性的话，熟练开发者的职业路径恰恰按他们的方式为他们的成长提供了时间与尊重，请多些耐心与尊重，因为公司会正视不同的岗位角色以留住出色的人才，提高他们的工作满意度。

我们都牵连其中

可能你早就猜到，“熟练”路径并不只适用于软件开发领域，有很多这样的人在微软的不同部门工作，他们热爱自己的工作并做得很好，但是他们同样没有做好准备成为一名领导者，或是根本没这么想过。我们可以而且应该理解并感谢这群人，他们热爱这个公司，热爱他们的工作，热爱我们的客户，他们只甘于奉献，却不想被迫成为一名领导者。

作为管理者，促使大家脱离一种舒适状态，超越自身的期望值是很重要的，我们不能也不应该对任何人降低这种标准。但是，胁迫我们这些出色的员工去干他们不喜欢的工作是不可原谅的，这样会让他们感到不受重视或认为他工作不够努力，而恰恰是他们让我们得以成功。成为这个世界上最出色的公司里最智慧、最具生产力与激情的一员，那是再好不过了。

作者注：熟练开发者的概念在好几年前就曾提到过，当一个能力出众的高级工程师不再期望向更高层晋升时，不会再有人质疑。

2002 年 10 月 1 日："人生是不公平的——考核曲线"

开发人员 Harley 和 Charlie 具有相同的级别，在相同的部门。Harley 工作很努力也很出色，Charlie 也一样；Harley 很聪明，Charlie 也很聪明；Harley 是名团队成员，Charlie 也是；Harley 为一个首要的软件功能工作，Charlie 则工作于一个次要功能，猜猜谁会在绩效评审会议上位居高位？谁会得 4.0 分而不是 3.5？当然是 Harley。

作者注：这些数字指的是微软老一代的评分系统，分值范围为 2.5～4.5（得分越高，奖励越丰厚）。3.0 分基本合格，大多数人渴望 3.5 分或更高，我们已经多次更改了考核标准，评分系统也是，但是它仍可以用来对两个有着相同工作，具有相同职责的人进行比较与评测。

不知道有多少次，像 Charlie 这样的人来到我办公室或是其他经理的办公室抱怨他们受到了怎样的不公正待遇。"我与 Harley 一样努力，" Charlie 说，"而你却让我去开发这样的垃圾功能，我按你说的做了，却得到这样的狗屁评价，这不公平。"对不起，Charlie，人生本就不公平。

听到这样的咆哮声了吗？到处都是 Charlie 这样的人从座位上跳起来并狂叫，"你到底什么意思，人生本就不公平？这就是你狗屁管理的狗屁借口！你把这样的垃圾扔给我，我是不是应该把你的交待当做放屁，再开辆卡车把 Harley 碾死，然后把他的任务功能抢过来呢？谁把你的脑子搞蓝屏（GPF）了，为什么我还要为你这样一个脑子蓝屏的经理工作？"

回到现实中来吧，Charlie，你这样太唐突了，换个经理也起不了什么作用。

作者注：通用故障保护（General Protect on Fault，GPF）。有些故障会使老的计算机系统崩溃，使它们显示带有崩溃信息的蓝屏。对于 Charlie 的系统崩溃来说，以他的观点，他是别人失败的牺牲品，而事实上，如我后面所述，Charlie 是有机会选择的，但逆来顺受是他失败的真正原因。

我不想再逆来顺受了

确实，管理层也有责任（说句套话），但是那是你的职业，而管理者不得不权衡不同的诉求，总要有人来开发不太讨人喜欢的功能，如果你想接手，好吧，那是你自愿的，我希望你能安心些。

但是没有人告诉你说必须接受它，醒醒吧，成熟些，找到属于你自己的 Harley——踩下油门，驶上职业的快车道。你说："这该怎么做？"聪明些，不然只能尝 Harley 剩下的。

像 Harley 这样的人要么很幸运，要么很机灵，或者二者兼具。因为我们已有定论，你的运气实在太臭了，你还是学得机灵些。这里有个妙招，机灵的开发者善解客户的意思，并深谙业务。

作者注： Harley 和 Charlie 这两个名字最初用的是 Hughie 和 Dewey（我喜欢 Car Talk 这个节目）。当我看到一句“对不起，Charlie”时，我改了这名字，唯一能同 Charlie 押韵的是 Harley Davidson 中的 Harley，这成为了后面专栏中音调与隐喻的基础。

译者注： Harley Davidson（哈雷·戴维森）是一家历史悠久的摩托车制造公司，由 William、Harley 和 Davidson 三兄弟于 1903 年创建。一个多世纪以来，哈雷·戴维森一直是自由大道、原始动力和美好时光的代名词。布什总统曾经在访问哈雷戴·维森工厂时赞不绝口，称赞道：“哈雷·戴维森是美国企业家精神的杰出典范。”

信息就是力量

通常，首要的功能是显而易见的，谁都可以在产品开发周期前期就发现它，然而，总还有些功能看似不那么重要，但日后会变得很重要，而有些看似重要的功能，日后却会发现不那么重要。为了明辨利害，你以及 Charlie 必须充分理解客户的需求及其业务逻辑。通常情况下，你团队中的大部分人（包括管理层）并不明白这一点。

为什么你的同事和管理者中很多都没有去了解客户和业务呢？

- 因为他们认为他们已经了解了，不过他们依据的是陈旧的信息。
- 因为他们认为客户跟他们自己没什么两样，而业务就是“多多赚钱”。
- 因为他们依赖其他人去了解这些事情。

其实，你不必如此愚蠢而无知。

关注业务

为了使你的工作与业务看齐，你必须研究下你们团队的业务模型。记住，你的上司可能并不比你更熟悉业务，所幸的是，你的产品单元经理（Product Unit Manager，PUM）对业务相当地了解，而且产品单元经理的上司也会很精通，约见你的产品单元经理要容易一点，所以先从他这入手。

然而，只是跟你的产品单元经理沟通可能还不够。产品单元经理对于业务模型的理解可能会有失偏颇，因为他会过分强调他自己的产品。为了更加真切地看清楚整体情况，你通常需要去找总监（Director）、总经理（General Manager，GM）或者副总裁（Vice President，VP）。不过这里有一个很重要的礼节，你最好先得到你产品单元经理的同意，再去安排跟他的上司会面。

啊噢，Charlie 可能会问：“到底有什么理由能让我的总监、总经理或副总裁接见我呢？难道他们没有更好的事情去打发他们的时间了？”实际上，假设他们有 30 分钟的空闲时间的话，这些高层人员中的大部分都愿意看到你的出现。

作者注： 在微软，虽然我们还要约见总监或副总裁，但产品单元经理的时间往往是满负荷的。你可以在第 10 章的专栏“我们是功能型的吗？”中了解更多关于微软撤换产品单元的问题。

前进，我的生活我做主

当然，总监、总经理或副总裁的日程总是很满的，他们要处理办公室政治问题，在会议上捍卫他们的部门及决策，到处忙于救火，分析数据以支持他们的立场或找出薄弱环节，还有其他各种从

你干过的工作中分析得出的任务——这就是他们开始工作时就要干的事，相信我，他们喜欢干这事。

这些高层人员喜欢跟基层人员打交道，这有助于他们了解最前线发生了什么，他们喜欢跟这么一群人交谈业务，这群人敬仰他们并问一些他们很容易回答的问题，对于他们来说这些是一小块美味可口的馅饼，而你就是基本调料。

有一点要记住，高层人士每天都在处理问题，他们不想听到你的抱怨，如果你还是发牢骚，他们就会把你的问题归咎于你自己。当你要求与他会面时，先要确定你想要谈论的是，业务的哪些方面是很适合你及你的团队来做的。

> **作者注**：作为一名员工，你完全有理由向你的上司抱怨你所看到的问题，随便什么时候。你不应该只把问题留给自己——你应该找寻解决办法，也就是说，你与你的总监、总经理或副总裁的这次会面所谈论的是有关业务理解的，以及在个人的层面是如何理解自主权的。

行动起来，与人多沟通

通过你跟上层管理者的交谈，当你领会了业务模型的真谛，你就会深受他们的喜爱（这在你要晋升时也很受用）。接下来，你要充分理解客户的需求。

Charlie 说："这不是项目经理的事吗?" 对，没错，只是不要在 Harley 带着你的评分绝尘而去时，你不要被他排出的污气给呛到了。了解客户是每个人的职责，有很多很简单的办法来做到这一点：

- 奉行可用性测试。
- 参加一些客户专访小组。
- 参加一个在线社团。
- 与客户支持服务代表讨论你的产品。
- 阅读产品评论。
- 看看竞争对手产品的评论。

竭尽所能接近你的客户，了解他们的所思所想。

柠檬有了？来杯柠檬水

现在你要开始煽风点火了。如果你了解业务及客户，你可以告诉他们哪些功能很有用，哪些功能不过是种装点，你也知道哪些功能没有得到他们的重视，而这些功能恰恰有助于他们赢得客户的业务。

> **作者注**："肥猪"（Hog）是哈雷·戴维森摩托车的一种可爱昵称。

机会在于你已经为一些不那么起眼的功能而工作，知道这些功能具有使产品得以成功的潜质，所需要的就是提高对它们的重视，以满足你的业务及客户的需求。促使它们按这个方向发展下去，那你将成为团队英雄。

换种心态

如果你发现其他人把所有"甜美"的任务都分光了，或者你不能忍受做些保持兼容性、旧版软件支持及安装配置（它们都很关键但却都很乏味）之类的工作了，还有其他的方法来改变你的

命运。

最简单的办法是在产品开发周期伊始就选择一些最好的功能来做，因为大多数人对业务及客户全无概念，你会发觉，找出一些主要的功能并不是难事，而其他人往往疏忽了。然而，如果你已经在开发过程中感觉被排挤在外了，有两个绝佳的办法能让你重归正途：

- 发现一些新冒出来的功能需求。这是一些听起来类似“怎么能忘了它呢”、“客户在最后时刻提出来的”、“与我们的竞争对手媲美的”，以及更通俗的说法，“这个关键功能都发臭了——我们需要新鲜血液”这样的功能。把你的业务及客户牢记在心以防新发现的是一些功能鸡肋，但是如果某项功能确实不错，那就尽全力完成；如果你原先正在开发的那些无聊又愚蠢的功能是由于新分配的功能造成的，那尽早把它们剔除掉。
- 成为重要功能的后备开发人员。找你的上司并对他说：“嗨，Harley 正在开发一项极其重要的功能，却没有后备人员，如果你同意的话，或许我可以对这项功能作更多了解，或许我可以帮他修复 Bug 或其他什么，那么我们就有个后备，以防意外。”太棒了，你说得很到位，你要么会成为这次开发的英雄，要么在下一次 Harley 去干其他事的时候，你就能接手他原先的任务。

> **作者注：**在本章专栏“猩猩套装中的机遇”中，我会更多地谈到机遇把控的问题。

谁是操盘手

人生确实不公平，好活儿都轮不到你手上。要铭记，你是你自己的老板，你要了解你的业务及客户，并在工作中分享这种财富，但是你不能指望仰仗你上司的认知或大度来将你的职业生涯向前推进。

要掌控自身职业生涯的主动权以及你所从事的工作，好好了解你的业务及客户，善用这些认知，把你的产品打造得更好，使你的职业生涯向前迈进一步。如果你成功了，这不仅仅是你一个人的胜利，我们所有人都将为此获益。

> **作者注：**有三篇专栏在我心目中占据着特殊的位置：“开发时间表、飞猪和其他幻想”（参见第 1 章的第一篇专栏）、“牛肉在哪里?”（走向质量之路的转折点，参见第 5 章）以及本专栏“人生是不公平的”（评论性的话题，不过读起来饶有趣味）。虽然后面也有很多不错的话题、激情说道和文字游戏，但它们的基调都源自于这些早期的专栏。

2006 年 11 月 1 日：“职业阶段上的角色”

我在这坐着，累得不行了。不是因为长时间的工作，不是因为感冒来袭，也不是因为缺少睡眠，而是想着如何对以下问题作第 1 000 次解释：“一个有着三个下属的人怎么会是一位个体贡献者?”或者“架构师为什么没有一份独特的职业阶段简介?”其实，这两个问题的答案都很简单：角色≠职业阶段。明白了吗？我不这么认为。该死！我受不了了。

作者注：职业阶段简介（Career Stage Profile），是在不同的职业阶段，对不同工种员工工作期望的一个详细描述。通常，每个工种都有3个阶段："入门阶段"（对于美国开发者来说，级别范围为59级~62级）、个体贡献者的"技术领导阶段"（63级~69级）以及主管和经理的"组织领导阶段"（63级~69级）。

知道吗？你并不傻，其他那1 000个刚刚被我开导过的人也不傻，角色与职业阶段的关系有些微妙。言多必失，不过，它真的不复杂——角色和职业阶段不是一回事！

一个人同时扮演很多角色

也许给出一些定义会对你有所帮助。我说的角色就像电影中的角色一样，不同的人可以扮演同一个角色，而同一个人也可以扮演一个以上的角色。实际上，你可以在同一个会议或对话中扮演多个角色，这种情况一直都在发生。

举例来说，假设你是一名开发主管，你正在跟一位老朋友讨论一个功能的实现问题，而这位老朋友正好在向你汇报工作，他还跟你的表妹结了婚。在同一段对话中，你可能扮演的角色有：管理者、朋友、架构师、亲戚、同事、导师和对手，那是一个极端的情况，但那完全是有可能的。

角色可以在瞬息之间改变，通常情况下，我们在工作中有一个主要角色，也许还有一个次要角色，以及一些更次一点的角色。比如说，一些架构师有少数的几名下属，他们的主要角色是架构师，次要角色是主管，还有更次一点的，与工作不那么相关的角色是微软员工、同事、测试者、设计者、导师和下属。

搞清楚职业阶段

职业阶段跟角色有很大的不同，它不会在瞬息之间改变。你不会拥有多个职业阶段——你只能有一个。你可能在实际工作中参考很多种"职业阶段简介"，因为它们有助于定位你所扮演的角色，然而，在你的职业生涯的任何一个时点上，你仍然只处于一个职业阶段。（我会在后面解释为什么会这样。）

那么到底什么是职业阶段呢？职业阶段定义了在你所选择的成长路径上目前取得的进展，因为你只在一个职业阶段中，知道自己在什么阶段就显得很重要。为了检验你当前的职业阶段，你可以采取如下步骤：

- 选择你的工种：开发、测试、项目管理及用户体验，等等。

作者注：用户体验（User eXperience，UX）作为一个工种，主要是指设计师和可用性专家。

- 选择你的职业志愿：技术领导者（个体贡献者阶段）或组织领导者（管理者阶段）。
- 和你的管理者一起查看"职业阶段简介"，将你的技能、工种和职业志愿对应到相应的职业阶段。（你当前的级别与能力通常就可以确定你职业阶段的起点。）

我的志愿，先生

记住，这跟你当前的角色没有关系，要看你的志愿。假设你现在的角色是开发主管，但你希望你的技术领导力为人所知，而不是你的管理技能。如果你是一名在雷德蒙工作的64级的开发

者，理论上来说，你的职业阶段要么是“主管 SDE”，要么是“高级 SDE”，然而，因为你的志愿是成为一名技术领导者，你的职业阶段就应该是“高级 SDE”。

> **译者注：**SDE（Software Development Engineer）是指美国的软件开发工程师这个职业。

“但是我是一名开发主管啊!”你说。是的，如果你想继续做开发主管，或者也许有一天你成为了一名开发经理，这都没问题。但是如果你想成为一名技术领导者，那么，要么把你的职业阶段换到“高级 SDE”，要么继续在“主管 SDE”这条死胡同里自娱自乐。

“但是我有 3 个下属啊!”你说。这就对了，因此你的所有角色中有一个是主管，大人物哦!但那跟你所选择的成长路径没什么关系，如果你想成为一名技术领导者，那么你必须选择这条成长路径。当然，你还需要作为一名优秀的管理者以扮演好你的主管角色，而且很多你的实际工作仍然会与“主管 SDE”阶段的相关，但那些不是你职业生涯发展的目标。

资质过高

你说，“如果我是一名 66 级的首席 SDE 会怎样呢，我也很高兴是一名主管?”在雷德蒙，“主管 SDE”最高是 64 级（65 级也属于可接受的范围）。如果你是 66 级，那你在“主管 SDE”阶段的资质就过高了（尽管你肯定还能扮演主管的角色）。

为什么“主管 SDE”阶段以 64 级作为封顶呢？因为在雷德蒙，64 级的你应该拥有主管角色所要求的所有技能和权限。如果你的级别是 66 或者更高，但你实际却不是这种级别是因为你能够很好地管理一小部分个体贡献者；你能做到这个级别，那是源于你的技术领导力。如果你不做技术领导者，而只去扮演一个主管的角色，那么你的作用并不符合 66 级的要求。这就是为什么你的职业生涯路径是技术领导者而职业阶段是“首席 SDE”的原因。

> **作者注：**本栏目中关于主管在 64 级封顶的论断是最具争议的地方，大家（包括我尊敬的一些重要人物）以为我是在说开发主管不能超过 64 级。他们误解了！我的意思是，如果你是 64 级以上的，你能做的肯定不仅仅是管理一个小团队，而是一个很重要的团队，而你肯定拥有高深的技术领导力，这样的话，你的下一步发展潜力就不是小团队的管理能力了，而是你在技术方面的洞察力和指导能力。在作为一名开发主管这次要角色的同时，你必须让你的职业阶段名号跟你的成长源动力相匹配。

我与众不同

你说，“那么，为什么没有一个称为架构师的职业阶段呢?”架构师是一种角色，你需要为不同的职业发展路径而不是不同的角色设置不同的职业阶段。高级技术专家的职业发展路径跟架构师的是一样的，两者都是技术领导者，因此他们有着相同的职业阶段。

同样，架构师的角色跟享誉全球的数据库日志专家的角色是不同的，但是他们必定有相似的技能，并且以相似的路径得以提高。如此的话，他们既是“首席 SDE”也是“合伙 SDE”。

只有一种选择

你说，“好吧，但是为什么你只能处在一个职业阶段呢?”如果你可以扮演多个角色，为什么你不能有多个职业阶段呢？因为你的职业阶段是跟你的职业发展路径联系在一起的，而职业发展

路径又跟你的志愿联系在一起。

大部分具有独立人格的正常人在某个时刻只会有一种职业志愿。是的，随着时间的推移，志愿可能也会改变，但在你生命中的任何一个特定的时间点上，你只有一个主要的职业目标，那个目标决定了你职业阶段的唯一性。

可能你还在热切地期望同时成为出色的技术领导者和组织领导者，这样你会败得很惨。原因不是因为你不够努力，如果你服下足量的兴奋剂，你可以接连几周都每天工作 24 小时。

问题在于，成为领导者意味着领导其他人。那也就是说要跟他们沟通，但并不是所有人整天都会在你的身边，当一天结束的时候，别人是否有空会限制你一天内能完成的工作量。为了成为一名卓越的领导者，你必须把重点放在你的工作上！

你想成为什么

当你长大后你想成为什么样的人呢？那是“职业模型”提出的最核心的问题。你不必提供完美的答案，你只需一个把心思放在你的工作上并使你的工作得以进展的答案。

“择其善者从之”，这是我很喜欢的一条谚言。如果你的志愿改变了，你的职业阶段也可以跟着变，但在此过程中你会学到很多并不断成长。我们不会总是成为一个一开始时想的那样的人，但有趣的是过程！只是，你不该呆在三岔路口犹豫不决。选择你的道路，尽情驰骋吧！

> **作者注：**自从我发表这篇专栏以来，已近 5 年，人们还是困扰于角色与职业阶段问题。为了避免这种混淆，微软已经考虑在工程工种中加入一条技术领导职业发展路径，与 IC 及项目经理并列。以我个人的看法，我想最好把 IC 跟项目经理这两条职业路径都给撤了，而把技术领导与项目经理作为角色分开。开发者就应该走开发者这条路，但是有些开发者还是可以有管理、架构、Web Service 等其他的角色。

2007 年 5 月 1 日：“与世界相连”

对于很多微软的工程师来说，首席（Principal）对于他们很重要。注意，不是关于信念的原则（principle）的意思，而是首席（principal），这个出现在充满妄想的职业阶段的名词。在这个阶段他们有了更大的影响力，获得更多的尊重，同时拥有更多的股权——生生不息。荒谬！天真！实际上，是影响力和他人的尊重使你达到了首席这个职业阶段，额外的股权不过是不错的无心之柳。

但是，那些非首席的如何赢得影响力呢？他们如何能得到他人的尊重？很多工程师都认为，他们被授予的这个头衔源于他们的聪明才智和独特的技能。好吧，他们的能力和智力可能在他们的亲属中是出类拔萃的，但在他们的同事中间却不是。我们希望我们聘用的每一个人都是聪明而且有才能的，因此，你还有其他什么特别的吗？

没错，你可以成为一名像主管或经理这样的组织领导者，把工作做好，你就能从你管理的那些人那里获得影响力和尊重。但是管理有它自身的特点，并不适合所有人。作为一名个体技术领导者，如果单单聪明还不够的话，你又该如何拓展你的影响力呢？答案是通过你的人际关系网络——人际关系是根本！

你认识的人

一些闲着没事的人可能对“你认识的人很重要”这个说法指指戳戳。你的想法是你真正的价值体现吗？是的，它们当然是。现在，回头再想想你是谁？你的想法是什么？与其他27 000名睿智工程师的想法相比，为什么你的想法值得我花20分钟的时间去了解，而不理会这世界上数不尽的优秀工程师的意见？

理解和赞同别人的想法是需要时间的，尤其当那些想法来自于你不认识的人，因为你不知道他们想法的背景。因此，如果你想要你的想法得到他人的赞同，你需要发展与那些可能赞同它们的人之间的关系。你认识的人越多，可能赞同你的想法的人也就越多，你产生的影响也就越大，你得到的尊重也就越多，你也就能进一步做你想做的事情。就这么简单！

一个强大的人际关系网络也能帮助你发现新的角色，从同事那得到更好的评价，对眼前发生的事情有更多的了解。你的人际关系网络让你看起来（以及做起事来）更加潇洒，当有人问你一个问题时，你不必知道它的答案，你只需要把问题转给一个聪明的朋友，即使最终是由那个聪明朋友回答的问题，那个提问题的人还是会把你看成是提供答案的人。

不过，你怎样去扩展你的人际关系网络呢？一旦你有了较大的人际关系网络，你又怎样去维系这些关系呢？让我们直面它。大部分工程师（包括我自己在内）都不是“交际花”，我们也没有时间或没有举办宴会的喜好。所幸的是，交际并没有那么难或者费事，虽然它确实需要花费时间，但不会多于搞销售或办公室政治所费的时间。

养成习惯

为了建立一个大范围的、强大的人际关系网络，你必须养成下面这些习惯：

- **好奇**。对所有事情都要充满好奇。
- **心存感激**。感谢那些帮助过你的人。
- **快速响应**。对那些请求你帮助的人及时做出回应。

这三种习惯对于建立和维系一个强大的人际关系网络至关重要。它们并不复杂，它们花费的时间不会比个人保健所需的时间多。但如果你缺失了它们，你的人际关系网络将迅速瓦解。所以，要养成这种习惯。

难道你不好奇

大部分工程师天生都是好奇心很强的。建立人际关系网络的窍门是把这种好奇心放到别人感兴趣的事情上，因为，那些对他人的工作充满兴趣的人，有谁能不为所动？

当你碰到某个你认识的人，甚至你都不认识那个人，你也可以问询一下他的工作，只要时间允许，尽可能多地发现他愿意讲的东西。始终习惯性地去做这件事情，你将了解到大量的信息，你就会有一个广阔的人际关系网络。

关注他人（不是你自己）很重要，注意，不是你自己，是你认识的人。当然，你要回答你的新朋友或老朋友向你提出的有关你的任何问题，不要偏离你朋友的兴趣太远。因为，谈论你自己的话，除了让你自我感觉良好之外不能给你带来任何价值，谈论你的朋友可以培养亲密感，有了亲密感就有了信任，然后必要的时候你就有机会把话题引到你的兴趣上来。

作者注：关于本栏目，我曾收到过这样一个问题，“如果对方也在试图建立人际关系网络怎么办呢？难道你们当中的一个人不需要谈谈自己的个人兴趣吗？”是的，那是当然，这种对话应该平衡一下。我强调不要谈论你自己是因为那样的话很容易就会让你迷失在自己的兴趣和成绩之中。

那么，下次因为某个原因滞留在公交车上的时候，或排队等候领取食物但队伍却停滞不前的时候，或者等待一个会议或课程但迟迟不能开始的时候，询问一下跟你一起滞留在那里的人他们当前的项目情况，不仅仅是项目的名字，还要包括那个项目具体干的是什么，怎么运作的，有什么窍门，有什么有趣的事，有什么痛苦的事，参与项目的都是些什么人，管理上怎么样，方方面面！这不是客套的对话，而是真正的好奇，目标是跟那个人建立起一种互利的关系。这很有效，而且不管怎么样，花费的时间不会超过你滞留的那段时间。

我们对你充满感激

另一个很微妙却很有效的能把别人拉入你的人际关系网络的方法是欠人家一个人情。我是在阅读关于亚伯拉罕·林肯的文章时学到这一点的——他竟然从住在几英里之外的邻居那里借书。除了读到了稀有图书之外，林肯发现，请求一个小小的帮助，比如借一本书，可以在他和出借人之间形成一个有力的黏合，出借人会发现林肯是一个可信赖的人，并且林肯也心存感激。另外，林肯会出现在出借人的“债务”清单上，这对出借人有利，也有利于建立一个称心的人际关系。

译者注：亚伯拉罕·林肯（Abraham Lincoln，1809 年 2 月 12 日—1865 年 4 月 15 日），美国政治家，第 16 任总统（任期：1861 年 3 月 4 日—1865 年 4 月 15 日），也是首位共和党籍总统。在其总统任内，美国爆发内战，史称“南北战争”。林肯废除了奴隶制度，击败了南方分裂势力，维护了国家的统一。内战结束后不久，林肯遇刺身亡。他是第一个遭到刺杀的美国总统。

既然如此，假设你正在寻找机会，想把一个媒体编解码方面的专家拉入到你的人际关系网络，你知道那个人的地址，但他不认识你。礼貌地就某个编解码问题向那个人寻求帮助，将是开始建立关系的一种理想途径，只要你遵循下面的这些指导方针：

- 问一个需要“专家级”的人帮助的问题。否则的话，只会浪费那个专家的时间，可能还会遭到鄙视。
- 简洁明了地提出你的问题，并且给予一定的时间。这表示你尊重专家的时间，正如我以前所说的那样，这在所有沟通中都很重要。（参见第 8 章的“你在跟我讲吗？沟通的基础”栏目。）
- 好好地感谢那个专家，并且要具体地指出他的帮助给你所带来的价值。你要让那个帮助过你的人觉得整个过程都是有积极意义的，让他以他的专业引以为荣。

你可能会说，“但为什么不去讨好一下专家呢？难道请求别人的帮助不会不好意思吗？”当你去帮助别人的时候，特别是你不请自到的情况下，那个人会欠你一个人情，他和你的关系也会掺杂着歉疚和负担。而当那个专家帮你的忙，你也真切感激他的时候，那就没有负担或歉疚了，相反，你的新朋友只会体会到一种优越感和尊重。

我回头再回复你

很自然，帮助是双向的，这也是人际关系网络的功能所在。人们给你提供帮助，你也要给大家以帮助。如果你想要维系这些关系的正常运作，你必须做到“快速响应”。

那么，快速回应意味着什么呢？好吧，想一想，一个人要有多长时间没有回复你的邮件你会认为他对你的事情不够关注？一天或者两天？你必须在大约24小时之内，对来自于你的人际关系网络的邮件做出回应。就这么简单！

你可以这么回应：“对不起，我现在还不能给你答复。我可以一周之后再回复你吗？”这种回应比根本没有回应要好得多，不作回应意味着你漠不关心。因此，“我现在正忙”这种回答表示了你的关心，只不过你现在有事忙着。

你可能会说：“你是在开玩笑吧！我收到的邮件太多了。现在，我要在一天之内回答所有这些愚蠢的问题吗？”首先，维护人际关系是可以获取大量红利的一种投资，没什么东西是免费的！其次，回应常常是很容易的事情，可以把问题转交给其他人来回答，这跟你自己来回答收到的效果是一样的。为什么呢？因为人们只关心得到答案，他们写信给你，答案得到了，那也就够了，只是一定要记得感谢那些代你回答问题的人。

> **作者注：**我在第8章“有的是时间”中将讲一些高效管理邮件的技巧。

欢迎来到这个世界

哪里可以为你的人际关系网络找到合适人选呢？任何地方！排队的时候，开会的时候，上课的时候，在各类讨论中，在像“代码箱”和“工具箱”这样的协作关系中，在网络中检索，你在微软公司内能去的任何地方。在微软之外，你也可以从各类会议中找到人，还有博客和论坛，如此等等。找到人是容易的，不过把他们拉入你的人际关系网络并且维持与他们的关系则需要些技巧。

当你找到了这些人，记得把他们记录下来。我保留着很多看过的很棒的E-mail和论文，以便我在将来的某个时候当我有机会把它们的作者拉入到我的人际关系网络时，能够很快地去联系他们。

> **作者注：**如今，在网络上出现了一些社交网站，尽管它们很美好、很有用，但你要意识到，布拉德·皮特不可能成为你的朋友，安吉丽娜·朱莉也不可能。这些站点是不错的起点，但不是唯一的方法。

> **译者注：**布拉德·皮特（Brad Pitt）是好莱坞最具票房号召力的新一代性感偶像，是一个不折不扣的影坛“万人迷”。安吉丽娜·朱莉（Angelina Jolie）是布拉德·皮特的妻子，《古墓丽影》的女主角的扮演者，是美国目前最有影响力的女星和演员之一。

一旦你把某人拉入了你的人际关系网络，你还要注意跟他保持联系。不过，不必刻意去那么做，建立和维护人际关系是特别看重机会的，关键是要好好利用机会。当你偶然遇到一个你好久

没有联系过的朋友时，一定要停下来和他谈谈，哪怕只是短短的几分钟。问他最近在忙些什么，表现得好奇一点，这样就能够把他们拉回到你的人际关系网路。

记住，交际未必需要大量的工作——它是个承诺。你因此得到的回报是扩展了自己的思维与影响，你会了解更多当前正在发生的事情，以及机遇在哪里。有更多的人会了解并尊重你和你的才能，你的想法会更容易被人接受，这一切会给你带来更大的影响力和尊重。在这个过程中，你甚至还能交到一些亲密的朋友，这才是最重要的。

2007 年 11 月 1 日："找个好工作——发现新角色"

评审季快要过去，该梳理一下现状，重新审视一下你的职业生涯了。有些人有职业规划，知道他们现在的位置并已经谋划好了下一步的方向，我称这些人为"明智的、成功的但又不安分的"。可能我是出于妒忌，因为，人总得有步骤明晰的职业规划。但是计划跟不上变化，我发现我永远无法预测以后会怎么样。

同样，临时抱佛脚也是不可取的，临时抱佛脚的人也早已臭名昭著——"怨天尤人"。你必须有周全的计划说明你想要什么，怎么做。即使你对现状很满意，但世事难料，因此对你的未来最好早作打算。

当然，你可以现在就着手准备应对之策，但此时也可能感到死神将临，你坐以待毙而不再努力争取；可能你够努力，但你当前的角色使你只能把计划缓一缓；可能你已完全茫然无措——你的职业生涯一塌糊涂，你的上司就像复仇女神，你每天的工作都在痛苦、迷惘及无休止的折磨中度过，最终注定了失败、浪费、碌碌无为。该是为自己换个新角色的时候了！

> **作者注：**我想有人会对他们自己真实的情况过于悲观而茫然无措，或者相反，有人认为改变可能会更加肯定他目前的境况，或是有助于他找到一条更好的出路。

现在我们该怎么办

如果到了该行动的时候，你该怎么办？评审结束时，很多工作机会就冒出来了，因为大家又要开始工作了，因为在评审核定的时候，人事经理还不能确定岗位用人（候选者不会在评审期间给管理者们提供坏消息）。这就意味着有很多活儿可供选择，你该怎么做？

很自然，你希望有一个具有挑战性的、别开生面的、有意思的工作。然而，却有很多因素会左右你的乐趣、你的影响力以及你的成长之路。通常人们用来区分角色的标准有以下 4 种：

- 技术——就是你现在干的。
- 市场——你为之卖力的目的。
- 工作方式——怎么做。
- 团队——你服务的对象及合作伙伴。

大多数工程师首先选择基于技术的工作，他们喜欢做些很酷的玩意儿，毕竟，他们是工程师。这就犯了个错误——技术往往在最后一个环节。对于你目前的工作，起先你大概首先选择了技术，是吧？结果怎么样呢？

众神归位

最为重要的标准是市场。你新建立的团队在忙着一些很重要的策略性工作吗？今天是副总裁决定取消项目以来最心烦意乱的一天吗？这里有个笑话，在一个悲惨的市场开发小组里有个处境堪忧的“棒球明星”，主管跟他说：“有你没你我们都会玩完。”不要加入这样的团队。

第二个重要标准是工作方式。这项工作是否有益于你的学习与成长，还是它也就老样子？职业模式对此有个专门的术语“经历”。你如果希望多方面发展以提升自我及你的职业道路，大型项目组、小型团队、孵化工场、不同工种及远程站点都可以提供不同的成长机会，不要害怕探索不同的道路。

然而，市场与工作方式的标准往往只对缩小选择范围有帮助，真正的区别在于团队。你所为之工作的人及你所共事的人对于你是否乐意接受新的任务分配具有重大影响，至少目前为止是这样。最酷的技术也会因为糟糕的管理而让人发狂，最“恶俗”的技术反而会让你的团队取得光鲜亮丽的成绩，但在你试着创新的时候就不要顾及我这种说法了。

> **作者注：**对于大学课程我也是同样建议，师资力量远比课程更重要。

没什么留给我的了

至此，大半的人可能会说：“这太完美了，但是你怎么找到一个跟你的要求适合的活儿呢？就像项目经理，他们可不会大声宣称他们的无能，而且，在你申请一项喜欢的高级工作前，它早被人定了。”可你不知道吧？人事经理为找到一个合适的人选同样面临这样的问题，方法是一样的——撒大网，利用你的人际关系。

不是所有潜在的职位都会公布的，目前公布的只是一部分，聪明的求职者会在职位发布前找出这些潜在职位。通常，这些潜在职位仍是需要公开的，这会给所有的求职者以机会，但是你越早知道这个工作，你就越有机会得到它，无论在公司内还是公司外。

你怎么能知道有团队正在找你这样的人呢，甚至在他们意识到这点之前？资方也在以同样的方式确定工作人选——利用他们的人际关系网。如我在本章先前“与世界相连”中说的，与你的同事建立稳固的人际关系对于你的成功是很有必要的。当你想要找份新职位，给所有这些人写信，是的，所有的人。理想情况下，潜在的项目经理们会从你的三个朋友那里了解到你，这会给他们留下深刻的印象，于是大把的机会就来了。

如果你的朋友为你找了份工作，你会感到愧疚吗？这视情况而定。会因为你给了他们深刻印象你才有这样的机会吗？天才们面对的都是充分竞争的市场。利用人际关系是施加个人影响力的一种手法，是卓越人才寻找工作的方式，所以，你最好要擅长这一块。

路漫漫，其修远

完美新任务的下一个障碍是人事咨询会议及面试环节，留意你在他们心目中的位置以及这个团队是如何作出决策的，这些可能体现了你将来的处境以及这个团队的运作方式。

要先了解这个团队及其项目，为人事咨询会议做准备。如果你有个强大的人际关系网络，那么你的朋友有些可能也在这个团队或者跟他们有密切的合作关系，在跟人事经理谈之前，先跟他们聊聊，问问在这个团队工作、生活会是怎么样的。

作者注：在微软，你可以在申请职位之前通过人事咨询会议（informational）向人事经理们进行咨询。

你必须坦诚以待、面向未来，阐述自身的情况。但是，不管你目前的状况有多糟糕，你也不要说你有多想离开现在的团队，只能表明自己对加入新的团队非常感兴趣，这很重要，谨记。如果问你为什么要离开，只要说："你们的团队拥有着很多新机遇，这听起来非常诱人。"如果一直追问，你就可以说："没错，我原来团队的内部关系不太和谐，但这不是我现在跟你谈话的理由。"（这也不应该是主要原因——选择离开很少能带来运气）。

你可能会被要求回头再进行一次面试，这样你可能会很紧张；或自上次面试以来可能有很长时间了，他们看起来很恐怖的样子；而有些人就会像重回大学一般死记硬背地回答一些问题。然而，心理上的准备对于树立信心、重拾记忆是很有帮助的，但这还不是成功的关键。

关键在于你自己。如果你试着模仿他人的模样而遭到了拒绝，你永远无法知道他们是否错看了你；如果你学着他人的样子得到了肯定，你会担心有负他们的期望（"他们是否真的需要我呢?"）。做回你自己，你会舒坦得多，自信得多，这样才是对的，你也可以真正确定你与你的团队彼此之间是否合适。

如果离开有些麻烦

如果你目前的团队及项目经理不想让你离开该怎么办？如果你还有很重要的工作未完成怎么办？如果因为你之前的团队人员的水平都不怎么好，还时时翘首以待着你能给他们以"痛苦与伤害"，当面对这些让你充满自豪感的情形时，你该怎么办？稍等片刻，Kersey 先生，我们来仔细分析一下这些问题。

作者注：我引用了死亡之愿（Death Wish）中的人名。好吧，我老了。

首先，没有谁是不可或缺的，即使天灾人祸，生活仍要继续，这比你离开一个团队要悲惨得多。因此，你感到愧疚是没必要的，原来的项目经理谴责你也是没有必要的，如果他们没办法阻止你离开，他们迟早会迎来新成员。

这就是说，职业修养是很重要的。匪夷所思的事会经常发生，特别是在我们这行当，你永远不知道什么时候你可能要重新与之前的团队共事或为他们服务，诋毁老团队或拒绝工作的顺利交接是不妥当的。

这就意味着要做好面试的前期准备工作，安排好交接计划，并尽可能完成未完成的工作。记住，我们所说的是交接计划，不是卖身契，如果你不能将你在老团队中的工作顺利交接给其他人，那么就不要列在这个计划中。

我欲乘风而去

变动是不会这么方便的，就是说从来不会有适当的时间这么做。但是，如果你的职业生涯一直停滞不前，这对你，对你的团队，抑或微软都是无益的。评审结束或产品发布后的这段时间是你好好思考职业生涯，并决定这次变动是否于你有益的最佳时间。

无论你在哪你都要保持乐观的心态，这是没错的，但是该走的时候就要走。出色的管理者将给予你支持与帮助，毕竟，你与他们的团队患难与共对于他们的成功是很重要的。（参见第9章的“随波逐流——人才的保持和流动”。）

不管遇到什么困难，主动寻找学习新技术的机会，不断提升自身价值，再麻烦也是值得的。微软拥有一个充满机遇的广阔空间，我们拥有最出色的人才，但是你不努力争取你就成不了这样的人。偶尔换个地方可能是你保持自身优势、专注与动力的最佳方式。

> **作者注：**15年前，我刚来微软工作的时候，这个公司规模还很小，大家相互间都是朋友也很方便在部门间更换工作，而不需要事先建立个人的威信。现在，一个部门的工程师就有原来整个公司的人那么多，这就意味着当你要换部门的时候，你必须先树立起同外招人员同样高的威信。还有一点要注意，如果部门内部有一个更合适的工作，你应该首先考虑。

2007年12月1日：“要么带头做事，要么唯命是从，要么赶紧离开”

我们又快到半年度职业生涯讨论会了。我始终没有从年中评估的剧痛中苏醒过来，该是把期许与谦恭放手给下一代的时候了。在经历短暂的挫折后，通过这种评估管理者可以向员工们传达信息，而员工们可以设法挽救他们的职业生涯。现在这种评估系统已经由更耗时的CareerCompass所代替，这个CareerCompass很复杂，还不如原来的评估系统来得简单明了。

> **作者注：**CareerCompass是一个内部Web应用，员工们可以将自己的技能水平与某些标准作比较，对自己的职业生涯阶段做出评估；管理者或作为技能水平参照标准的人，同样可以通过这个工具对其他员工做出评估。这个应用很不错，但第一个版本没什么可行性。

别误会，即使我想重新启用年中评估系统，我一开始就知道这种职业性对话并没有什么针对性，虽然，它是有建设性意义的。遗憾的是，虽然人事经理及项目经理的出发点是好的，但他们的建议通常让人搞不懂，举个最突出的例子，对于战略性眼光，大家反应最多的问题是：“我的上司说我应该从战略性层面思考问题。但这到底是什么意思？我应该怎么做？”

问得好。向你最信任的项目经理或执行官请教，如何提升战略性眼光。而他或是她一般会给出类似作足计划、专注重点、充分理解业务及其他一些毫无意义的建议。是的，战略思想贯穿于这些过程中，但是你去看一场职业橄榄球赛不等于你就是四分卫了。

战略不是一项具体活动，是一种思考的方式。它无时不在，不仅仅在计划的时候，如果你想驾驭它而浪费更多的时间在会议上是不可能有帮助的。你对待工作及现实的方式要成熟些，你不再是个无知的流浪者了，该醒醒了，成熟些吧。

盲目跟从还是做名牛仔

有个朋友最近老跟另两个同事作比较——一个只干自己想干的事，而另一个按照业务策略按部就班。我的朋友对我说：“后者更称得上一位战略家。”不，他不是，也谈不上战略性地思考问

题。前者是一位没有牛群的牛仔，而后者不过是遵守战略决策的战术执行者，这么评价更适当，但还不至于说是有战略性眼光。

战略性眼光有其成长路线。收下心，看看你错在哪里，我的意思是，对照以下几点：

- **忽视战略决策** 自我为中心的牛仔。
- **按照公司战略按部就班** 战术执行者。
- **不厌其烦地问什么是战略** 策略性战术执行者。
- **谋划战略决策** 战略决策者。
- **决定公司战略** 战略领袖。

我们来仔细分析一下这几点，看看每个层面的误区在哪里，以及如何沿着这条成长路线一层一层向上走。

天呐，毁灭者来了

自以为是的牛仔会无视战略，认为这是官腔官调，会有碍于他创建很酷的创新性产品。他的英雄情结就如传说中的叛逆者——打破规矩，开发些很诱人的软件功能以便使他们成为“首席”或“合伙人”级别。哦，这名头够唬人的。

> **作者注：**“首席”及“合伙人”级别是微软最高级别的称谓。我认识好些这样的英雄人物，而他们没一个会忽视战略。他们是懒得理这些中层管理人员的，这些中层管理者只会反复强调战略而不会真正理解战略的意图，而叛逆者却做了真正战略性的事并得到了高层管理者的奖赏。这个事实说明一个道理，在你搞糊涂之前，确认你已经对战略有了深入了解。

这就对了，不要回避这些事实告诉我们的道理，我们不是一个创业之初的公司，大多数牛仔们也不是在孵化工场工作，我们是在为一个成熟的产品及服务工作，牛仔往往就会陷于迷茫与无知而不搞明白他们是善于制作吉他呢还是弹吉他。

为什么牛仔们会在大型项目中迷失？因为他们的个人主义行为就像随机性噪声，没有合作精神，一名牛仔所做的工作会抵消另一名所付出的努力。单独来看，他们开发的软件功能或许会很酷，但都不适合整体，因此，谁也不会看中它们，也不会用它们，他们所做的是毫无价值的。

更糟糕的是什么呢，牛仔们开发的这些随意性功能只会给客户带来更多的困惑，他们所做的只会有百害而无一益，他们只会从个人的角度出发想改变世界，这只会给他们自己带来更大的失望。不是因为他们的想法不够好，也不是因为这世界不给他们表现的机会，牛仔的失败只归因于他不懂得与他人合作，将产品推向前进。

反抗是徒劳的

战术执行者会按战略安排办事，信任团体的计划，找出让这些计划付诸实现的建设性方法。很多牛仔们觉得，战术执行者唯唯诺诺，放不开手脚，唯恐伤及集体的团结，只会“哞哞”乱叫博取赏识。然而，成为一名战术执行者的真正理由是出于以下原因之一：

- **“我对战略决策不感兴趣。”**如我在本章前面“当熟练成为目标”中所述，不是所有人都想成为副总裁或技术专家的。有些人首先是乐于奉献，然后才花些时间放在其他个人感兴

趣的地方。理解并追随当前的战略决策是充分发挥个人影响力的最佳途径。

- **“我还没准备好为战略决策担起重任。”**对于那些具有成为技术或团体领袖的人来说，要想超越自我扩大他们的影响力，他需要有个起点。即使你不赞同现在的战略决策，你最明智的起步方式是了解它并追随它，只有傻瓜才试着抗争或发展他们不懂的东西，重复过去的失败。在要改变目前的战略决策前，聪明的人首先会充分理解它的优劣之处。

追随当前的战略决策并不是说要放弃原创的想法或创意，事实上是相反的。世界上很多伟大的建筑及艺术作品是在地理与环境限制的情况下而迸发出的灵感结晶，在符合固定条件限制的情况下完成精美的设计是一项在问题解决的过程中创意灵现的挑战性工作。

战术执行者是极富价值的员工，但是他们还没有作好向战略决策者跳跃的准备。在他们能决定新的战略决策之前，他们必须学会对当前的决策作出建设性的质疑。

这样对吗

策略性战术执行者在质疑战略决策的同时仍会遵照执行，对那些不停报怨又不提供建设性建议的白痴来说，这两者是绝对不同的。策略性战术执行者仍会遵照目前的战略决策，并希望其成功。他只是对战略决策有过充分的了解及思考，会提出些看似似是而非的假设或方法。

有些人会害怕走上这一步，他们不想让人看起来像麻烦制造者，他们相信决策者们知道什么是最好的，弱弱地说一下，以下这些情况可能表明你到了策略性战术执行者的层面：

- 决策者无知得可笑，就像我们一样。他们总是得不到准确或全面的信息，他们会被他们的成见所蒙蔽，他们或许根本没什么能力，虽然在微软这个级别不乏真材实料的天才。
- 麻烦制造者总没什么脑子。深思熟虑的质疑与烦人的质疑是有很大区别的。如果你问的一个问题你自己都能轻易地回答，或者你只是想抱怨就抱怨，你这人就烦人了，这样只会带来很多麻烦。然而，如果你已经对情况作了周全的考虑，确实想把事情办好，你就不会造成麻烦。你指出的是某些需要多加以解释的地方或者有待改进的地方。大多数决策者会感谢你提出的意见并在下一次评审中慎重考虑你的方案，如果你的意见不被采纳，那就该换个项目。
- 当前的情况无时不在变化中。当一项决策出台，它是基于人们对之前情况的认识。现在不一样了，出现了新问题，市场、技术、竞争对手及客户都变了，领导层变了，需求变了，有些时候这种变化难以捉摸。所有这些改变会导致当前的决策不再合适或者不受认同，你该引起注意，检查一下目前的决策是否还有意义，如果没有，就要多作思考。

那么，你怎么知道什么时候该质疑决策呢？什么时候你该发自内心地问：“等等，我们为什么要做这些？”问题一出就要寻找答案。问问你的朋友和牛仔们：“为什么？”一直问，直到你得到一个很好的解释或没人可问只能问决策者的时候，他们会把你当成一个战略决策者而记在心里。这是不是就意味着你就是一名战略决策者了呢？还不算，但是你已经为这种跳跃做好了准备。

我有一个不一样的战略决策

战略决策者会改进决策，对决策进行调整以适应条件的变化或弥合与实现方法间的隔阂。策略性战术执行者与战略决策者的鸿沟在于策略性战术执行者指出问题所在，而战略决策者会设法解决这些问题。一个策略性战术执行者只是同他的管理者一起提出问题，而战略决策者同时会提

出可行性方案来解决存在于当前决策中的问题。

其实，在策略性战术执行者与战略决策者间没有什么太大的差距，你要做的只是承担起改进的责任而不是把它留给别人。人们往往混淆，以为战略性谋划就是打破规则或是重新建立你自己的规则。

牛仔会这么说，战略决策者作出战略决策后，却只会把那些解决方法放在脑子里不断地扩展、调整。这就是为什么战术执行者充分理解决策并贯彻执行是多么的重要。

当一个战略决策者变得成熟起来，他就会开始着手推广战略决策，增强它的影响力。比如，因为他的团队没有按照规范将他们的程序接口完整地文档化，这使他们深受其害。在开发伊始就对它们文档化是很方便很简单的，特别是有相应工具支持的时候，但是没人养成这样的习惯。

一个初级战略决策者可能会建议采用 Sandcastle（blogs. msdn. com/b/sandcastle/）来文档化他团队的代码；一个二级战略决策者可能会把 Sandcastle 放到产品生产线创建中去；一个三级战略决策者可能会将文档完工统计放到创建报告与质量检验关中去；一个高级战略决策者可能会在内部通过 CodeBox 或是外部的 CodePlex（www. codeplex. com）发布他的 Sandcastle 创建（Sandcastle build）及解决方案，并鼓励相关人员在微软都采用这个方案。一个高级战略决策者与一个战略领袖只有一步之遥。

> **译者注：** Sandcastle 根据程序集和对应的 XML 注释文档来生成帮助文档。

我是一名领袖，但不是管理者

战略领袖为他的团队、他的部门或者甚至是整个公司制定决策。战略决策者与战略领袖的区别是战略领袖为所有新兴的领域制定决策。你也可以成为你自己的战略领袖，这有助于让现在的领导层支持你的观点。

> **作者注：** 你并不需要先成为一名管理者才能成为一名战略领袖。架构师是战略领袖，设计师是战略领袖，制定出另一套安全、高性能或高可靠性解决方案的专家也是战略领袖。战略领袖意味着一种方案及思路，而不仅仅是种角色。

通常这可以通过发布全新战略决策观点白皮书，或围绕你的计划建立草根社团，或在你的人际关系网中承担起领导者角色来实现。然而，你必须做好成为领导者的准备，致力于将你的战略决策付诸实施。当决策由你而出，它就成为了你的责任——要贯彻执行。

谁都可以成为一名战略领袖，虽然并不是所有人都想承担这份责任。关键是不要只注重你自己与你自己的兴趣，这样做就太狭隘，也不会吸引他人来追随你。相反，关注你的客户、你的团队、你的公司及这个世界的兴趣所在。如果能运用我们的能力与资源的作用，则对于他们将是多么有益的事啊！

从事于你自己心爱的项目是很容易的，但是你最好穿上牛仔的长筒靴，成为一名精神领袖意味着就要穿着别人的鞋走路，以更高的要求，急人所急。记住这些，你就很可能成为一名出色的战略领袖。

2008年7月1日："猩猩套装中的机遇"

是微软年度评审的时候了。在相同的岗位，高级、一般及低级员工的薪酬是不一样的，我们有着资深的工程师，也有甚至无法跟上其他同事工作进度的人，过去的一年，另有一些人完成了他们大部分的工作，现在该将这些人与前面两者作个比较，调整下待遇了。

> **作者注：**有很多微软内部或是外部的人都在批评差别薪酬制，他们认为这样会导致团队人心涣散，工作懈怠。但我非常赞成把团队成绩作为薪酬的必要评价标准，我不认为差别薪酬有什么问题。（见第9章的"比较的恶果——病态团队"。）

作为一个管理者，这个时候也会有无能之徒向我哭诉，声称他们缺乏锻炼的机会来证明他们是有价值的，就好像管理者是把机会藏起来不给，只会在心软的时候给他们一点似的，也好像机会很少，只会把这些宝藏藏起来留给趋炎附势的一小撮谄媚之徒似的。不，你们这些蠢人，机会从来不少，它们也不会被藏匿起来，它们美轮美奂，芬芳四溢，它们套着猩猩装就站在你的面前，整日捶胸顿足呢。

但是很多聪明的工程师并没在意，他们眼前看到的是硕大、吵闹又发着恶臭的猩猩套装，日复一日，而他们从没注意它。有时候他们的项目经理给他们提供机会，邀请他们开会，或是将他们安排进一个项目，但这些工程师，才艺出众的工程师，没理会这些。他们把这些机会转手交给了别人，他们只是偶尔关注一下，他们只是把它放在一个角落里，直到某天无意间想起。

为什么?！为什么工程师们没有注意到这些机会？为什么把它们扔在一边，而只会在7月到来抱怨是多么缺乏机遇？猩猩套装内的机会无处不在，每天都有，你们却忽视了，为什么？

朋友，我有眼不识泰山

我之前一直认为，人们忽视这些机会是因为他们太懒或是凡事漠不关心，我认为漠不关心是个大问题，但是我不再认为懒惰是主要原因。相反，我认为人们错过机会的原因是出于"心不在焉"。通常，工程师没注意到一些显而易见的机会是因为他们把脑子放在了其他地方。

你可能看到过一段视频，它要求观众们记录一下篮球在球员间传递的次数，在这段视频中，一个着猩猩套装的人走到球队中间，对着摄像机镜头捶着胸狂嚎一阵，然后就离开了人们的视线。视频末尾问观众，你们是否看到了猩猩。不仅仅是观众想不起来了（当时我也是这样），那些喜欢猩猩的人我看也被弄蒙了。然后，他们就再看一遍这段视频，才发现，这个大猩猩就在屏幕中间，好似在嘲笑观众的观察能力。

到处都是

工程师们没有注意到猩猩套装内的机会是因为他们每天只专注于计算传球次数上，他们根本没心思去留意这些机会。不过，你可能就是其中一位，认为机会会被人藏起来。容许我来移开你

的障目之叶，将每天从你身旁溜过的机会罗列一下：

- **极品功能**。没错，你知道有这些东西存在，你可能还知道这些玩意儿到底是哪些，但是你如何得到与它们相关的工作呢？我敢打赌，它们已经设计好了，也编好了代码，但现在在等着评审；可用性测试、单元测试及自动化测试也需要编码或评审；还有 Bug 需要修复。我还敢赌，开发这些功能的人现在只是偶尔来关注它们一下，他们需要后备人员。但是，当然了，你是很忙的。
- **客户主张及业务逻辑**。你认为你了解客户及业务，但是你没有，这就是机会。你与客户的直接沟通越多（如通过产品支持服务，了解可用性及回馈信息），你就越了解业务（跟副总裁谈谈，看看业务规划、业务模型及分析），那么你就会明白极品功能到底是哪些，关键功能又是哪些。但是，这些是别人的工作，对吗？
- **关键功能**。你认为这些无聊的功能如安装配置、补丁、隐私、兼容性、权限及可管理性是很难完成的。太好了，当做这些工作的人以失败下场的时候，你的机会来了。如果你对客户及业务很了解，你就知道如何化腐朽为神奇。但是，现在有很酷的事情可干时，你为什么还要唠叨呢？
- **专题小组、委员会、虚拟项目团队**。你会小心翼翼地说，这些小组委员会是地狱之火。但是，只有当薪资水平比你高的人想解决问题的时候，它们才会被组建，因为谁都讨厌做这样的徒劳之功，这样你就有机会带领这些小组寻找真正有效的方案了。不然，你可以让别人来做这些事。
- **过程与工具改进**。你可能热衷于此，但没人会听你的。不要想着如何改进你的点子，也不要囿于自我瞎倒腾些点子，看看别人的过程与工具，想想你如何才能引进它们对其进行改进，想想如何像一个团队或一个公司一样齐心协力，齐头并进。
- **常规性问题**。你很难在 5 分钟内修复所有问题，只能解决一两个。每个问题都是次机会，这可不是讨巧，这是真的。没错，你永远不能把所有的问题修复，但确实我们能解决些力所能及的问题，不然，你就维持原样。

我好可怜

每天都充满着这样的机会，这让我很晕菜，工程师们还是抱怨缺乏让他们成长、提高他们资历的机会。是的，你很忙，你自己的事都忙不过来；是的，你为之努力的都是很酷的创意，但却不是客户或业务所需要的；是的，做自己的事很利落，因为没有人来打扰你；是的，保持原样就少了不必要的辩争；是的是的，放走机会，碌碌无为，是再容易不过了。

“但是我没有太多的时间放在寻找这些机会上。”无能之辈哀号道。你是在跟我开玩笑吧？就如我在第 8 章的专栏中所说，时间永远是不够的，每一天，每一分钟，你都掰着指头算计，从你手头无数的任务中选几个来做。关键是事情分轻重缓急，扔掉那些即费时又费力又不是“必须做”的事项，而要专注于能使我们的业务，我们的客户，从而使我们的职业生涯与众不同的事上去。

而缩手缩脚的无知之徒会说：“但我们没什么可剔除的了，我所有的东西都是‘必须做’的。”听我跟你说：“那么你总是能完成你所有的需要做的事情吗？确定？你肯定有满腹牢骚要说。”如果你没有完成所有的事，那你肯定做了选择，如果你做了选择，你唯一要做的就是对这些选择作些改动，这就是生活。成功的人通过这种改动把他们有限的时间放在新的机会上。

有句老话："少说多做。"可以概括这种通过选择个人职责创造时间的理念。但是大多数新人都承诺过多，以想给管理层及其同事留下深刻印象，在每天结束时，这些人所做的就少于说的，从而错失机遇，那么他们的评价就会低于那些完成本职工作并有突破的人。

不走寻常路

我并不是说这很容易，我是说应该这么做。没有谁会端着盘子呈给你成功，你的父母不会，你的老板也不会，谁都不会，你该想着如何从每天都在你身边摆着的无数机会中选几个了。

你并不需要抓住所有这些机会，一年中抓住几个就可以。如果你进入了一个你所关心的小组，那就踊跃参与，成为其中真正的有贡献的一员。我不管你是否很忙，时间是挤出来的。这并不意味着工作的时间要更长或更努力，意思是去掉不太重要的事项，这些事项可能目前很重要，但长远来看并不是。

停下手来，关注一下你所敬重的人。不是因为他们做成了多少事，而是他们完成的工作有深刻内涵及影响力，好好掂量你对工作任务的选择，时刻注意可能的机会，创造空间利用它们，那你就会成为别人仰慕的人。

> **作者注：**就如我上面所说，有些人常忘了这点，一年中你只需要选择一到三个机会，选这些能让你热血沸腾的机会。或许过程是曲折的，但是你会因此有所提高；或许那是困扰了你朋友很久的一个问题，而你认为你可以解决；或许那是一个客户或团队忽视的问题，而你想一试身手。拿出你的激情来，但不要盲目单干。如果有项任务你插不了手，该如何说服别人呢？在第8章的"离利于乐，控制你的上司"将有所讨论。

2010年3月1日："我是很负责的"

现在是半年度职业生涯交流时间了。我可以对我们的人力资源管理工具口诛笔伐，但是那就像在前列腺检查时抱怨——肿胀得厉害。相反，每年这个时候扑面而来的是BOGUS合约（commitment）。

你应该听说过SMART原则吧（明确性、可度量性、可行性、以结果为导向、期限限制等），BOGUS合约则是泛滥的（Bloated）、超期限的（Outdated）、普通的（Generic）、非典型的（Unrepresentative）及以自我为中心的（Self-Centered）。

最让我受不了的是BOGUS合约被当成SMART的，它们看起来好像也很明确，可度量，在一定的期限内好似也可以有所成效，但是，这些所谓的SMART合约就是彻头彻尾的BOGUS的。作为一名管理者，对BOGUS合约进行评价是很痛苦的事——"是的，你的12个合约已进展到第11个，但遗憾的是，你的合约让你看起来好似表现非常不错，但你的团队却失败了，其中有9个合约中途有改动，而真正需要改善的合约也是最关键的，但它却被遗漏了。我的天！"

为什么SMART合约会变成为BOGUS的呢？我们来仔细分析一下。

> **作者注：**很显然，人事经理已经在业绩管理工具上得到"反馈"，现正忙着。遗憾的是，反应的问题都是系统性的，是很难在短时间内解决的。如果你作为微软的一员有机会对人事管理制度提出建设性建议的话，那就赶紧行动——为了我们所有人的利益。

我让泡沫拿铁给埋了

最具 SMART 的合约的第一个误区是：泛滥。你已经干了一整年的工作，这些合约内容应当是很明确的，可度量的，也应当参照着管理者的合约做的，但没多久，你的合约数量飙升到了 10 个，很快就有 15 个。

为什么有 10 多个合约不是件好事？我来说明下原因：

1. 编写它们要花很长的时间。我们的目的是干些实事，而不是浪费好几个星期来编写修改合约。

2. 定期对它们进行评审，或在年终绩效评审会议上对它进行讨论，这又浪费很多时间，所以大家不会这么做，因此这么多是没多少好处的。

> **作者注：**绩效评审会议（Calibration）是微软差别薪资体系的组成部分，通过这种评审会议，公司的员工按职业阶段及角色分门别类，对达到工作期望值与未达到期望值的员工进行比较并对其薪资水平进行调整。比如，Office 开发经理定期开个会，将 Office 的开发人员按不同的职业阶段分为三类：高潜质的人员占 20%，中等占 70%，另外的占 10%。通过这样的方法，可以对整个公司内的薪资水平及角色期望值进行调整。如果使用得当，绩效评审会议对提升公司局部效能比整体效能更有好处，但这就是我们采用的薪资体系。

3. 它们的期限规定很明确，所以拖到下半年完成的合约就绝对超出期限了，而安排在第二季度完成的合约也将近过期。

4. 因为没有一个合适的方法来区分关键合约与“项目组成部分”合约之间的区别，你与你的项目经理很难对主要问题给予足够的关注。

5. 同样，由于有太多的合约，你和你的项目经理就很难在对你进行评审时或绩效评审会议上对你出色的工作成绩给予重视。

有什么办法呢？有，最多只能有 4 项合约（如果你是项目经理，可以有 5 项）。前 3 项合约是你这一年中主要的项目职责——这些是你上司会在绩效评审会议上大加吹嘘或极力捍卫的。你的第 4 项合约应该将重点放在个人成长方面上，如果你是名管理者，你的第 5 项合约应该着重于良性团队建设上。就只要 5 项合约，搞定。

> **作者注：**如果你的上司坚持你最好按他的 12 项合约来做，那该怎么办？如果他够大度，你可以向他解释为什么你只用 5 项合约，然后，对照他的合约及时跟进。如果他不这么干，那你就听他的吧，他是头儿，或许你可以把这篇专栏塞到他的办公桌下。

活在过去

一年来，计划总是变化的。你的合约必须反映这些变化，不然它们就过时了。如何将这些变化反映到你的合约上呢？这要看这种变化是哪种类型。

- 你负责的关键项目有一项以上变动很大，从而需要你重写合约。很显然，在职位变动的时候就会发生这样的事，在一次战略决策或机构变更的时候也会发生。

- 一次战略性的决策变更后，你所担当的某一关键项目的预定方案会有重大的变动，这就要求实施计划以及你在合约中的职责也要有所变动。你可以不改动合约而只是对合约的变动作下口头解释，但是重新构思一下你的计划及绩效评估方法才是应做的。因此，趁合约改动的机会重新编制新方案，并增进对它的了解是很有必要的。
- 一次战术性的变动对你在某一关键项目中某一特定职责产生了影响，这样的变动就会迫使你改写你在合约中的职责，但是你还是可以在项目经理的同意下做些简单的口头解释。一定要先确定得到项目经理的同意。

作者注：当有变动的时候，与你的项目经理一起修改你的合约是绝对没错的，特别是在那些工作岗位调整或比较大的机构调整的时候。但也很少有人这样做过，可能是因为每个人都忙于应付变化而无暇顾及合约，而放到下一次评审时再行考虑。这样你就失去了一次与你上司重设期望值以及调整绩效评审指标的机会，但是如果你错过这样的机会，也没什么，很多人都会这样。

这说法有点新鲜

如果评审你的合约及职责与评审其他同事的一样简单，那么这些合约就太普通了。将你的10多个合约缩减至4个（项目经理是5个）可能使你编制的合约没什么亮点，但如果你把重点放在三个关键项目上，就不会发生这样的事，因为这三个关键项目是本年度工作的关键所在，你是不会轻易放弃或转手于他人的。

尝试一下放弃某个项目而只完成其他项目，看看结果会怎么样，你就会知道这个项目是否是关键性的。你的项目经理对你放弃的这个项目无动于衷吗？或者是这样的选择会对你以后的工作埋下隐患吗？

这三个关键性项目每个都有相应的概况描述，有实施计划，年终时也会有明确的可度量的成果。把这些项目编制成合约，你就有了三份主要合约——这三份是你希望你的项目经理在绩效评审会议上进行讨论的。

你会问："如何评审其他人的代码呢？如何编制优秀的单元测试？又如何助我们的产品发布一臂之力呢？"这些工作是成为一名优秀的工程师所必备的。因此，它们就成了实施计划的组成部分及个人开发的职责所在。这样的合约才称得上："我将成为团队及项目组极具决定性及影响力的一员，我将决定最高质量标准。"如此你就能团结所有资深的工程师及所有不俗的团队成员，按照既定的技能提升计划，通过一整年的努力来完成这份合约。

现在，当你的项目经理要对你的合约进行评价时，参照你的关键性职责，他就能很容易看出你是如何来完成它们的，是否这些职责跟你三个最重要的项目相关，是否你是以一个团队成员的身份完成本职工作的，是否你像项目经理一样尽心尽责。

我不认为这样评价就准确了

你应该让你的4项或5项合约可以对你整年的工作做个简单明了又不失公允的展示。签个13份合约没多少好处，只会显得毫无重点。

为什么合约太多会使你一年的工作显得毫无亮点呢？因为不一样的合约却在你的评审中占据

一样的位置，是的，这是没什么关系，你可以使用粗体字或字体加亮来突出显示真正重要的合约，然而，人的思维不会这么敏感，白纸黑字写着，他都会看一遍，结果很可能喧宾夺主。

不是所有的项目都是一样的，为防止你的成果（或努力）注水，你必须把重点放在关键项目上，而把次要的给剔掉。然而，要成为一名优秀的项目经理或团队成员，跟只做一个项目时一样——他们各自都得有自己的合约。

作为一名项目经理，应该确保效率低下的员工不能在一摞的合约中滥竽充数，找出那些你期望这一年里你的团队成员能完成的最典型的关键性项目，力促其顺利完成。

> **作者注：**“在合约中要慎重表述你自己，是什么意思？对我的评审是在绩效评审会议上确定的，可是在绩效评审会议召开之前这评审就已经写入合约中了。”大多数项目组尽量把绩效评审会议安排在员工们在他们的合约做出自我评价之后再行召开。不过，你还是可以在绩效评审会议上对自己能力作个评述，最大限度地发挥个人影响力，甚至这种评述可以超出自己的实际能力水平。你所从事的项目是不是很关键，难度是不是很大，风险是不是很高，这个项目是不是按照团队的计划进行，是不是紧随客户之所需？如果是，你就在合约上把这些写上，这样的话，在你想给合约加上更多评述之前，你的合约就已经对你的工作作出定论了。

只为自己着想

我在看别人的评审时感到非常惊奇，对这些人的评审在合约中已讲述得够清楚了，他们的项目经理还是认为这样的评审惨不忍睹。为什么会发生这样的事？这是因为合约的选择有误，还以自我为中心。

有时候这些合约只强调个人所做的努力而忽视团队的成绩；有时候这些合约所适用的水准并未达到团队对个人要求的水平；有时候这些合约忽视了协作与团队活力的重要性。总之，合约是以个人的角度编写的，而不是从整个团队的角度出发编写的。

如果你想通过一己之力只为自己工作，那么就辞职开家自己的公司；如果你想在这个项目中精诚合作，发挥出比一己之力更大的能量，为大家带来更多的好处，那么在合约中先摆正自己的位置。你的项目经理及你的指导员他们可以也会指导你如何做。我坚信，他们会的。

> **作者注：**马德哈万对我说，用于管理合约的人力资源管理工具给了这些合约一个特有的名字，“个人合约”，而没有更适合一点的名字，如“团队及客户合约”。文字使用有其厉害之处——这些名词应该改掉。

BOGUS，不仅可恨，且绝对是最失败的

谁也不想有泛滥的、超期限的、普通的、非典型的及以自我为中心的合约，它们不能体现真正的你，不能体现你工作的价值，也不能体现你对你团队的重要性。

仅仅是 SMART 的还是不够的，但绝对不能是 BOGUS。将你的合约重点放在四五个同等重要的领域，通过它们使你紧扣团队的共同目标，并对其产生积极作用，这样就能使你尽最大可能地为你、为微软以及为我们的客户提供超乎想象的真正重要的价值。

作者注：每当我跟别人谈起合约的时候，最普遍的问题是如何评测工作绩效及如何看待延续目标（stretch goal），我在第2章的“你怎么评测你自己?”中对工作绩效的评测已有论述。而对于延续目标来说，记住，如果疏忽一项关键合约，那绩效就不算达标，而完成它就算功德圆满；如果放弃延续目标，那你仍然算是成功的，如果也能完成它则是锦上添花。混淆延续目标与关键合约对两者都没好处——你既不会因延续目标而受肯定，也不会因此而被委以关键合约之重任。我建议干脆从合约中去掉延续目标，或是将它们独立放到 meets 或 exceeds 章节。

2010年4月1日：“新来的伙计”

“嗨！你是新手吧！”妙极了。你已经从一个身怀绝技、无可替代的权威人物变成了一个路边货。不管你以前是谁，你曾经多有能耐，你树立过什么功绩，但现在你不过成了一种营销手段。这些新到的牛仔可能表面看似意气风发，实则心里头七上八下的。

听着，大多数人都很友善，他们希望你取得成功。但是，事实上现在你给他们带来了什么呢？什么也没有。几个星期后，你新车的内饰味褪去了，大家也对你失去了新鲜感，他们想要你拿出成绩来。你需要拿出成绩来证明自己，但是你什么也没有。

在你走上正轨，像以前一样做出些成绩之前，大概需要6～12个月的时间，这依不同的工作类型而定。期间，你的工作效率会下降，你的信心会有波动，你的名望也会削弱，很不错吧。但你确实换了个角色使你的职业生涯走上新台阶。你此时会想些什么呢？你只是在想人生积累越多才会越完美。你怎么才能在不断积累的同时，将此期间的副作用降到最低呢？这里有几个很有效的办法。

作者注：在微软，我们总会对新雇员入职之前所做的工作抱过高期望。当发现他们之前的成绩并不是你所想的那么回事时，我们一次又一次经受惨痛的教训后就开了这个刚刚成长起来的人才。一个有经验的工程师的一年全职工资都在20万美元以上，这可是我们花出去的大把大把的钞票。没错，人们不能拿着他们的简历高枕无忧，面对新挑战，我们需要实实在在的成绩证明，不过，汲取一下他们过去的经验并考虑一下他们的建议作为新工作的开始是没有坏处的，我们都该好好利用下这种先发优势。

未雨绸缪

通过以下5个步骤可以使你快速走上正轨：

1. 全面认识自己。要知道你是谁，谁带你来到这个团队的，你的入门槛及长远局限性在哪？

2. 建立你的支持团队。熟悉一下这个团队中的关键人物，以及他们的价值主张。

3. 延长蜜月期。急事急办，你就可以快速取得成绩，树立他人对你的信任，这样你就有了喘息之机。

4. 学一下“绳”原理。好好学一下团队合作的基础原理，融入这个团队。

5. 开始探索。选一个与你密切相关且紧迫的项目，对它进行深入研究，在工作过程中不断磨炼。

计划有了——赶快行动。

全面认识自己

之前，你是个炙手可热的人物，英雄不提当年勇，谦虚让人进步，也让你免于麻烦缠身。在新团队里你两手空空，不懂装懂只会让你显得愚蠢，不牢靠，或者两者都是。你现在已经调到“倾听与学习模式”，这对你只有益而无弊。

自己要承认作为新手的不足，那么其他人也不会在意。当你着手工作的时候，像往常一样把你的长处与短处都展现出来，你已经重新开始，这是你取长补短的最佳时机。（“这件事对我来说不是问题，但不要让我干其他的事。”）当每个人一开始就知道你的分量，那离成功也不远了。

“告诉别人我的不足之处？你疯了吗？”不，我很正常，而你却很愚蠢，不牢靠，不会想到好的一面。大家都知道你是人——这不是秘密，大家在意的是他们给你的工作你是否搞砸了。他们对你了解不多，不知道你的短处何在，事先让他们知道你的长处与短处你就会安心很多，建立彼此的信任，你就能快速上手一个你向往的任务。

> **作者注：**一人之长短乃一人职业生涯的平衡棒，无论何时何地你都要手握不足的一端，这样可以增强你的抗压能力，使你始终持有不俗的眼光，成为公司长期的员工。事先让你的团队知道你的不足之处只是举手之劳，也是你通向成功的另一个关键。

建立你的支持团队

熟悉你团队里的关键人物是你取得成功的关键——人际关系很重要。你的项目经理都很了解团队里的人，那就从他那开始。当你一个一个跟这些关键人物聊的时候，包括你的项目经理，要闭上嘴专心听（无论做什么事都应该这样）。问些问题，用笔记下来。

了解每个人的角色，那样你就知道什么时候可以寻求他们的帮助，跟他们谈什么样的话题。明白他们的目标与问题，你就可以围绕他们关心的话题不断寻求帮助。把这个团队内部及外部的合作关系记录下来，因为这些很可能就是你下一步要联系的关键人物。最后，确认一下在接下来的一段或更长的时期内你可以给他们提供什么帮助，这就是延长你的蜜月期的必要之步，也是你决定参与的第一个项目的基本工作。

> **作者注：**如果你是一位新员工的上司，你可以通过跟他或她聊聊你的角色、目标、存在问题、合作伙伴关系以及近期及长期需求，使之适应新环境，然后给这名新员工一份需要会面的关键人物名单，并要求他们做好面谈记录，待事后汇报。通过记录本，你与你的新员工就可以决定哪些近期及长期的项目是急需接手的。你还可以推荐一位关键人物作为这名新员工的指导老师。

延长蜜月期

跟结婚一样，工作的头几个星期为新员工提供了额外的磨炼与学习的机会。不过，当夕阳西去，很快大家就把你想象成为他们要与之行婚姻之礼的天使。但这是不可能的事，你该如何延长这种蜜月期呢？急事急办。

通过和关键人物的交谈，你了解到近期你可以提供哪些帮助。在你出手相助前只要稍作准备即可，急事急办，你就可以快速取得成绩，树立他人对你的信任，这样你就有了喘息之机。

快速取得成绩对进入全新团队后迈出夯实的第一步是很关键的，利用你过往的技能与学识是你树立个人声望，充分适应新环境并取得一席之地的最有利手段，不管是引入一种过程改进方法或你之前团队使用过的漂亮工具，还是身先士卒，积极参与到项目中来，你很快会取得不俗的成绩。但不管你是怎么做的，千万不要陷于之前团队老套路中——把心思放在新的团队上。

> **作者注：**我的新角色——项目组里的关键人物，包括我的项目经理都说，要增强团队的凝聚力。我所在的新团队刚经历过一次机构重组，他们得重新像个团队那样开始工作。这些事实告诉我，当我担当了一种新角色时，是因为我有排除纷争带领团队前进的能力与经验。我给团队中每个人各起了个昵称，对他们付以期望，设立定期的面对面团队沟通制度，并不断给以激励。这很简单，也很重要。

学一下“绳”原理

在加入到新团队后不久，我就会起个别名，登录 SharePoint 的网站，进入问题跟踪（issue tracking），再打开资源控制（source control），然后在你的计划日程表里把定期召开的团队会议安排进去。如果你这么做了，就把团队的动态及基本工作流程仔细记录下来，这是日后工作的必备信息，即使今后的工作会改变团队动态。

通常，团队还会用一份 Word 文档、维基网页或 OneNote 记事本详细记录团队每一天每个星期的工作进展。阅读这些信息并不断更新是了解新角色“绳”原理的有效办法。

开始探索

当把手头的急事处理之后，你就要开始着手开展长期项目（需要几个月的时间）。跟你聊过的关键人物将给你提供建议，但是你作为旁观者时的想法及个人偏好还是要慎重考虑的。当你可以自由选择时，找个可以真正让你的管理层及同事们所称道的项目，更不用说你的客户了。

你的第一个项目会很吸引人，也让你充满激情，因为它是比较难啃的。你必须学习了解每种新鲜工具、系统、API、过程、依赖方、人际关系及规范。你对任务越感兴趣，你就越想征服它。当你经历了这一切，你就会熟悉人际关系，熟悉代码及工具，这样当你面对下一个项目时就没那么恐惧了。

不管你在这个团队呆了多久，多提问题是成功的关键，当你面对各种匿名抨击的时候，这点是非常重要的，一定要将你在团队维基网站或 OneNote 记事本中找到的答案归档。记住，你不是无所不能的——你不是时空的主宰，也不是人类智慧的主宰，即使是博士也有问题要问的。

> **作者注：**现在我在这个新团队已经有将近一年了，我也是照自己上面建议的做的，这样做相当有效。我一开始有两个项目——一个是有关如何改进开发过程的，另一个是如何调配功能团队的。在项目开发过程中，我采用了上述所有的做法，也就是急事急办，发展与关键人物的关系，收到了不错的效果，最终这两个项目取得了圆满的成功。当你给他人提供的建议同时在自己身上也很适用的时候，那是多么宽慰的事。

通往成功之路

祝贺你进入了新角色，我诚挚地祝愿你成为最出色的人。迎接一次新挑战是很有趣又让人兴奋的事，与新伙伴相处让你焕发生机，可以学习新技术，致力于新业务。为了保持这种积极向上的精神状态，你需要一个计划让你始终保持高效运转。

反思下你是怎样的人，你的长处与短处，与他人相处的时候分享你的信息，与新团队中的关键人物及其团队的合作伙伴要打成一片，通过协助他们解决他们的燃眉之急随时给他们带来快乐，熟悉新团队的基本工作，将它们记录下来留给新晋人员，选择一个与你相关又充满乐趣的项目，通过这个项目让你跟你的团队分分秒秒紧密地联系在一起。我们雇用的都是最出色的人，因此你也必须成为其中一员。多些自我认识，多些诚意，善于倾听，步步为营，加强学习，重在坚持，无往不胜。

2010 年 6 月 1 日："升级"

如果你不是一名微软的工程师，如果你对找个新理由攻击微软不感兴趣，那还是省点时间略过这节。如果你想知道如何提升个人技能水平，稳扎稳打，成为微软真正的工程师，那就看一下。

我作为微软工程师的管理者差不多有 15 个年头了，我为 8 个不同的公司工作过，都记不起曾经担任过多少次管理者的角色。每个管理者与机构对职位晋升都有其各自的看法，但有一些基本原则是他们共同遵循的。这些基本原则从来没有被正式归档过，直到现在——请往下看。

为什么我把这些"秘密"藏起来，而不利用起来以便晋升？因为它们不是秘密。所有与我共事的人都知道这些秘密，我也跟我的员工们谈论过。当他们听了之后，所有人都会问："为什么不把这些秘密写下来？"可以设想，人事经理看到这些道理这么简单会很惊恐的；每个工程师都不一样，没法给他们下统一的定论；另外，微软中的团体笃信这么一个信念——他们都是别具风格、独一无二的。管它呢，我才不管这些。我只是想告诉你……

职业阶段

大部分微软工程师知道有 6 个职业阶段，每个阶段都有各自的"职业阶段简介"（Career Stage Profile，CSP），但他们不知道一个职业阶段与下一个阶段的区别。

- **入门（Entry level，SDE I）**。你刚从大学校门出来，你还没有做好在一个淌着白花花的钱的专业团队中工作的准备，客户还没有对你产生一个专家的印象，你还是个愣头小伙。

> **作者注**：SDE 专为软件开发工程师而设，在其他工程领域有类似的职业阶段。

- **独立开发（Independence，SDE II）**。你可以完成上司交待给你的所有代码工作，仅此而已。如果你不知道怎么做，你知道该去问谁，或者知道可以找谁再问另一个人。这是你可以通过编程水平彰显能力的最后一个阶段。

- **团队领导（Team leadership，Senior SDE）**。你可以影响团队里大概 3～12 名工程师，就像是他们的项目经理或技术领导。与此同时，你产生的影响大大超越了你一己之力。
- **项目组领导（Group leadership，Principal SDE）**。你对整个项目组内大概 12～80 名工程师产生影响，就像他们的部门经理、项目经理、构架师或核心技术领袖。不过，你也已经是组织中的关键一员了。
- **机构领导（Organization leadership，Partner SDE）**。你可以对整个公司机构的 80～500 名工程师产生影响。你是部门总监、总经理、合伙构架师或全球核心技术专家。你在公司里已成为“合伙人”级别。
- **行业领袖（Industry leadership，Technical Fellow）**。你已经可以影响整个部门及整个行业了（500 到几百万的工程师）。你是名卓越的工程师、技术专家或副总裁。钱对你来说已经不是什么问题，除非你嗜赌。

你如何才能从一个阶段提升到下一个阶段呢？让我们一次一个阶段来讲解。

入门阶段

要从入门级别晋升到独立开发级别，你首先要学会独立。切，废话！这可能要花掉你 2～3 年的时间，这要看你的经验。不要总想着拯救世界，不要总想着如何给副总裁留下印象，好好干你的事，如果哪里搞砸了，就把它补上。让你的项目经理及同事看看，你是可以摆平自己的事情的。

自己独立完成任务并不是说就是单干，意思是说，你知道该问哪些问题，谁可以回答，以及如何找到答案。就算独行侠还要通图相伴呢。（我有些倚老卖老。）

> **译者注：**《独行侠》最早是一个电台节目，诞生于 20 世纪 30 年代，之后被改编成电影、电视、漫画和小说等多种形式。故事中，一队得克萨斯骑警遭到匪徒伏击，主角约翰·雷德（John Reid）被印第安人通图所救，之后他化身为骑白马戴面具的“孤胆骑警”，将各种歹徒缉拿归案并交给法律审判，通图则成为独行侠在行侠仗义、除暴安良时的战斗助手。

通常，职业生涯阶段分为两个级别（在雷德蒙，SDE I 有两个级别：59 级与 60 级）。如果你显示出进入独立开发级别的趋势，那在一两年后你就有可能升到第二个级别（60 级）；假设你的团队没有把晋升的机会都给占了，当你真正独立了之后，你就可以提升到下一个阶段（61 级）。

> **作者注：**入门级菜鸟的通病是抱怨，而不是设法解决问题。这个世界是个多元的世界，大家的境况却是一样的，没有谁愿意去解决属于你的问题。如果你遇到难题，就向你的项目经理及同事提出解决方案，最好是付诸实施，否则，生活就会充满困扰，最终你会退出你牢骚满腹的职业。

独立开发阶段

现在你可以独立了，稍加指点，你就可以处理一些单项任务。但你仍然要问很多的问题，你的项目经理一刻也没闲着，你还是名初学者。没有人有义务必须教你该怎么做。

在你进入独立开发阶段之前为什么一定要先独立呢？微软希望每个人都在这个公司获得成功，

如果你的级别提升了却在下一个职业阶段失败了，那么在之前的职业阶段里，微软就失去了一个有价值的员工。不会有降级的可能，因为这样会打击士气及积极性。所以，只有在微软认为你在下一个职业阶段能取得成功的情况下，且你的表现就好像你的级别已经提升了，你的级别才会得以提升。

微软希望每个工程师都成为一名独立开发者或团队负责人，不管是组织管理的或是技术性的。这就意味着，你被期望进入团队领导职业阶段，这通常要在进入独立阶段之后3~5年光景。如果你做不到，你很有可能被开除出这个公司。在微软你不能永远像个独立工程师一样原地踏步——你至少要在团队的层面影响你的同事们。

当你已经完全独立后，不管你的任务是什么，你将进入第二层独立开发阶段。当你能左右你的同事，证明你具有了充分领导能力（这期间需要6~12个月）时，你差不多也就提升到团队领导阶段了。

> **作者注：**更换项目经理及项目组通常对升级所需的时间产生影响，因为新的项目经理不知道你具有这种领导能力多长时间了，在更换项目经理或项目组的时候，有两种方法可以避免你重置升级时间表：
>
> - 请求你的前任项目经理及现任项目经理一起讨论你的领导才能，证明你有这样的才能已有多长时间。这跟请求晋升不一样——这仅仅是促使个人开发职业阶段的转换得以顺利进行。
> - 如果你打算换个项目组或项目经理，那就在晋升后1~2年内换掉，最好能得到现任项目经理的祝福，这样，这次更换就基本上不会对你的下一次晋升产生影响了。

团队领导阶段

太棒了！你已经到了团队领导级别。这并不是说你就是一名管理者了，这意思是你的影响力已超出你自己及你所做的工作。你的影响力左右着团队成员（比如说，给他们以指导，做个范例，提出大家都认可的设计方案，树立大家都认可的软件质量守则等）。

自此，你的编程技能与你的同事们再没什么差别，每个进入到独立开发阶段的人都具有了出色的编程技能。现在，只有你对他们的鼓励与影响力使你与众不同——软实力，这一点我早说过。看来，那些在高中及大学时代折腾得你晕头转向的书面及口头说教现在派上用场了。

> **作者注：**当到达高级职业阶段的时候，很多读者不同意我这样的说法：你的编程技能与你的同事们再没什么差别。他们举了一些例子佐证——真正具有极高编程才能的人早就进入合伙人阶段了，已不跟我们一伙了。我不否认这些事实，我只是想说是他们卓越的编程能力产生的深远影响力驱使他们的职业生涯向前迈进，而不是简单的因为他们开发过多么多的项目。另外我想再说一句，这些人并不是普遍现象。

微软员工未来潜质的评级系统（高层的20%、中层的70%、底层的10%）在这个阶段的作用有所转变。这是因为到了团队领导这个阶段你的事业就进入了一个平稳期，虽然微软还是希望你能努力提升到项目组领导阶段。这两者不同之处见以下表格：

潜　质	入门级别与独立开发者阶段	团队领导及以上阶段
顶层的20%	很可能可以达到团队领导阶段	有晋升可能，依进展情况而定
中层的70%	有晋升可能，依进展情况而定	有晋升可能，但会越来越难以确定，因为高层职位会越来越少
底层的10%	有被解职的可能性，除非他有广阔的提升空间	很危险或到了瓶颈期——征求一下你项目经理的意见，明确一下

当你偶尔在整个项目组显示出了影响力，你很可能具备晋升到团队领导第二层级的潜质。如果你对整个团队有足够长时间的影响力（6～12个月），那你差不多就可以晋升至项目组领导阶段。晋升到这两个级别后都可能要维持几年的时间，没有具体的时间表，也无法保证一定可以再提升，这取决于业务需要与个人抱负，犹如个人能力一样。

> **作者注：**跨团队及跨部门的管理算是什么阶段呢？是项目组领导阶段吗？不，管理多个团队及部门并不是项目组领导阶段的定性工作，即使是入门级的工程师也应该擅长于此。不过，当你对其他团队与部门的决策及组织安排产生影响时，你就完全具有升入到项目组领导阶段的资历。

项目组领导阶段

喔！你已经进入到了项目组领导阶段，成为了一个部门经理、项目组经理、架构师或核心技术领袖。祝贺你，这是意义非凡的一步，你来到这个阶段是因为你突破了你团队的限制，开始考虑团队外部的事情并对其产生影响。

对大型项目组产生影响，必定意味着影响了他们的决策与组织构成。这些方面的论题我已经写过不少专栏，特别是第8章的“寓利于乐，控制你的上司”及本章前面的“要么带头做事，要么唯命是从，要么赶紧离开”。这就要求你运筹帷幄，决胜千里，不再人云亦云，不再刻意指责，或怨天尤人，相反，你将锐意进取，开拓创新。在没有人保护的时候，尝试风险是很危险的，但这就是你成为一名领导的原因。

将自己铸造成一个强大的项目组领导，你就可以晋升到项目组阶段的第二个级别。不过，想进入机构领导阶段要难得多，因为一个公司不需要那么多的机构领导。很少有人顺利通过项目组领导阶段，所以抱负将决定着你继续前进，在一些大规模公司里，这一点确实很重要。

你的很多同事也很有抱负，他们期待在海外生活，与一帮难以相处的家伙开发一个乏味的项目，不停地飞来飞去，有时一有通知就得起身，保持手机24小时在线，就算再不情愿也要做出牺牲。在其他条件相同的情况下，有抱负的人才会成功，因为这种遇到任何事都全力以赴的抱负是进入下一个阶段的必备条件。如果你并不是这样，那就准备好在项目组领导阶段长呆的准备，你或许会有更好的表现，但是抱负是个关键性因素。

机构与行业领袖阶段（合伙人级别以上）

作为一名机构领导——部门总监、总经理、合伙人架构师或全球核心技术专家，你在这个公司已达到合伙人级别了。你会与鲍尔默及执行官开个特别会议，你将享受特殊的股权分配及特殊津贴，因为你不仅为工作成果负责同时要为公司业务发展承担责任。

你仍然要回到公司里来，也就是说，当需要你的专家级技术支持时，你得回来参与到项目中，即使你更喜欢呆在你的本职岗位上；也意味着你将参与到项目组领导、其他合伙人及高级新雇员的领导力发展计划中；也可能遭遇到与你一样有雄心壮志的人。从这方面来讲，你成长了，你应该可以而且也会希望掌控这些情况。

在成为一个行业领袖，即一个卓越的工程师、技术专家或副总裁之后，竞争将越来越激烈。你必定蜚声中外，成为在你帮助下创建的主要业务的中流砥柱。很明显，个人抱负与机遇是迈上这个阶段的关键。只有聪明还不够，到了团队领导阶段的每个人都已足够聪明，如果你想成为一名行业领袖，你还必须要有百折不挠、排除万难、坚持不懈的精神。

设定目标

如果你现在仍旧没有明白其中的道理，那概括地说，职业生涯发展有以下内涵——独立开展工作，感染影响你的同事，从大局考虑问题，以及做好个人牺牲的准备。为了实现你的个人职业目标，你想做多大的努力，愿意付出多少？是你必须扪心自问的首要问题。

这是你自己的职业生涯，它属于你自己。前辈可以指导你，善意的人们可以帮助你，但是最终，自己的事业自己做主。好好想想，你想成为怎样的人，什么会让你快乐，有成就感，能成就你一个完美的人生。充分理解上述内涵，给自己以准确的定位，追逐属于自己的梦想，就这么办。

2010 年 9 月 1 日："辉煌时代"

评审期快要结束了。或许你上升了一个层次，或许你脑子正满怀着成就伟大时代的想法——吹响号角，收获成功，让所有人都听候你发号施令。对于处于入门阶段及独立开发阶段的人来说，这就意味着要成为一名高级或首席工程师（项目经理或架构师）；对于高级工程师及团队负责人来说，这就意味着要成为一个首席或合伙人工程师；对于首席及合伙人工程师来说，这就意味着要成为副总裁或技术专家。

这世界不断地有企业执行官上任，他们功成名就，极具号召力。当我们年轻的时候，我们就陷于对他们的想象与向往当中。他们掌控着一个企业，过着幸福美满的生活，你也向往着能成为他们这样的人，就像电影等媒体中所看到的那样。为了你能有这样的生活，你的父母不知为你倾注了多少心血——让你接受教育，送你去好的学校，激励你要有自己的抱负。冷静一下，生活不是一场电影。

责任感有其利也有其弊，在你追求美好之前，你应该明白有得必有失。在品尝奶酪前你先要清除污物。

时间是个问题

当你的个人影响力不断扩大，成为一个机构或技术领袖时，你已非原来的你。不过，要让自己胜于他们的很重要一点是——要使你能产生长期的影响，需要多少时间？这点难以一言概之，但是很重要。

- 当你还是个实习生，项目经理要你做什么你就做什么时，在明确工作计划到完成所需工作之间的时间是以小时或天数计算的。

- 当你是名团队负责人，吩咐你的下属要完成一个项目时，在明确工作计划到完成所需工作之间的时间是以天数或星期数计算的。
- 当你是部门经理，将任务委任给团队负责人时，等待项目出成果所需的时间就要几星期或几个月。
- 当你是副总裁或技术专家时，完成一个重大的项目则需要6个月到3年的时间。

对于这种曲折缓慢的过程，你束手无策。即使一个公司是扁平化管理的也无济于事，因为沟通障碍重重。你面对的人越多，则需要理解你意图的人就越多，每个人理解并执行的时间就越长。

最好的方法是你要阐明你的意图，不断地重复——每次要以不同的方式阐述，并在过程中倾听他们的回馈。记住，大部分人是安于现状的——你必须不断督促，有些时候还要事必躬亲，拉他们一把。

前途光明

几乎所有的领导阶段都受做出决策与初见成效之间时间长度的影响，时间越长，你的眼光就要看得越远。

这就意味着部门经理与架构师必须着眼于下个月的问题，而不是下个星期的。因此，在工作成果出来之前，他们就要给予团队以指导，并不断地提醒他们需要注意的问题。只看重每天工作进度的部门经理被叫做“微观管理者”，这样的效率是低下的，作用也是有限的，往往会忽视长远利益。

对于副总裁或技术专家来说，提醒人们下星期要干什么是毫无用处的。执行官至少应该把眼光放长远到下一个年度，虽然在本年度对员工反馈做出响应及作小部分的调整是需要的。

上梁不正，下梁歪

随着项目进程深入，你把快速地做出些小成绩当成了大成果。你可以一直这样快速地解决一些小问题，但给你工资不是只做这些小事的——给工资是要收获大成果的，这需要时间。

这就让我们想起责任。如果一个工程负责人做了个糟糕的决定，通常这种不良后果在几个星期后就会显现，问题反馈及纠正也是很快的；如果一个执行官做了个错误的决策，很可能几年内也看不出问题，在此期间反馈会很迟钝，责任也会不明晰。

以我个人看来，我喜欢部门经理或架构师这种诱人的职位。他们的决策在几个月内就会有成效——时间也够快，可以由此进行及时的调整并积累经验，但仍然可以收到不错的成效。相反，执行官为错误的决策负责之前的几年间，他们却可以蒙混过关——假设他们还在同一个部门。

> **作者注：**不管他们所犯错误本来就非常明显，还是他们从一个岗位换到了另一个岗位后，这种错误才逐渐变得不可忽视，不断做出这样错误决策的执行官最终是要为此负责的。

当然，你要找到自己喜欢的岗位。如果你想马上有所斩获，你或许不应该去尝试比高级工程师或领导级别更高的角色；如果你想帮决策层制定集团战略，号令成千上万的员工们——你就不能介意公司政治，并等上几年再收获成果——做一名副总裁或技术专家或许更适合你。

幼小无猜

为什么副总裁及技术专家们与总经理及公司总监们一样，会陷于公司政治呢？因为当决策远未有实效之前，对于所谓的正确决策往往意见不统一。当意见不一致的时候，公司政治就成了主

要手段。这种因素在像微软这样的工程师为主的公司不太明显，因为很多执行官之前就是工程师，他们以事实、数据及逻辑为依据。不过，越依赖于工程师，那么个人主义、压宝式的制胜绝招及拉帮结派就会愈加严重。

“借鉴下前人经验怎么样——你知道的，案例研究及最佳实践?”适应于过去某一特殊情况的案例或实践在当前却往往是不适用的。总是有些特例或反例发生，如果你想成为一名成功的执行官，你就需要玩点公司政治。你必须知道谁会信任谁，谁会对谁有所掣肘，谁掌握议事日程，谁欠谁人情，谁有可能会支持或反对你，为什么，以及什么是这些利益相关者的软肋。

这就是公司政治，又乏味又卑贱。在每个公司每个政府中，当你达到某一层面的时候，这些就会显现，在这样的情况下万事无定数。

作者注：还有比公司政治更糟糕的，懈怠、不负责任给了公司里缺德之人以可乘之机，他们是缺乏恻隐之心与良知的病态撒谎者。Babliak 与 Hare 的书《套服中的蛇》中第 8 章宣称：与大概 200 名具有高潜质的执行者一起工作，他们发现大概有 3.5% 的人符合精神病态的特征，这比全社会精神病发病率的 1% 高多了。

请照办

当然，功成名就后好处就多多了。除了薪酬、威望还有额外津贴等，你还产生了广泛的影响力，你的想法就成了整个公司或部门的想法——如此种种形成了一种方针明确、步骤清晰的公司或部门发展战略。然而，这种影响力权威再高也有它的弊端。

作为一名领导，无论你说什么——即便是随口说说——就会被当成金口玉言。部门经理在他们升任这个职位的时候意识到了这一点——即使一种无心之谈也成了员工们的福音，这就像是海森博格不确定原理的另一个版本——开次讨论会必定对某些人或事产生影响。

当你的影响力越来越大的时候，无心之举也越成为个大问题。执行官的一次偶然评论就会决定一名员工几个月所做努力的生死。你要说什么，什么时候说，对谁说，一定要慎之又慎。为给你周围的人以准确的指导，最好的做法是惜字如金，只说你该说的。

在每句话前加上“我只是随便谈谈”或“如果作为一名普通的客户，我认为……”是没有用的。大家对你说的每一句话都奉若神明，人们总喜欢断章取义，溜须拍马。你能做的也就是把你的意思讲清楚，否则，还是闭上你的嘴。

作者注：如果说执行官们应该少说多做，那这一整年的时间他们是怎么度过的呢?

- 他们大量的时间要花在提供反馈、解决问题，为了达成他们的意图在工作过程中加强指导上。他们提出设想，为之呕心沥血、全力以赴。要知道，人们总是习惯于故步自封。
- 他们还要花大量的时间建立并维护与合作伙伴、同事、上司及员工们之间的人际关系。人际关系让一切皆有可能。
- 当然，一年的计划有时需要因时因事而变，重新安排计划是常有的事，就如为下一次的宏图大业做准备一样。
- 最后，还有些事关大体的细节——预算、人事、日常事务、随时听候上司召唤及个人私事等。

只选对的，不选贵的

现在你对成就丰功伟绩有了一个全面的看法。每个人都有自身的局限性以及对成功的见解。最关键的是要了解自己，要知道什么能让你开心，什么让你有成就感。

就如本章前面“升级”中说的，雄心壮志指的是，要达目的，就要全力以赴。你的目标越远大，你就要为此付出更多的个人牺牲，在你被赋予某种责任之前你首先要明白这一点。

我认识很多执行官，我非常感谢他们做了这样的选择，他们为带领我们的公司走向前进而奉献着自己；我也认识很多高级工程师及技术领袖，他们非常热爱他们现在的工作，也并不想有更进一步的发展。现在，你也了解了这两种情况了，为自己做个正确的选择吧。

2011 年 1 月 1 日：“个体领导者”

你想制造甲烷吗？一群好高骛远的工程师在一种扁平化管理的组织机构中聚焦在一起开了一场半年度职业生涯讨论大会。管理者声嘶力竭地喊着苍白无力的口号激励斗志，很快恶臭的气体即会从个体贡献者那喷射出来予以回应。

为什么满怀希望的人们满腹哀怨？因为技术领导发展路径很不明确，而机构领导发展路径又很狭窄。大多数微软部门都采用扁平化管理，这样一来，工程管理职位又大为减少，这些职位的级别又往更高层次抬升。所以，现在一个开发者可以坐到某方面副总裁的级别，但是要以一个领导级别的岗位为起点却异常艰难。

职能化的组织结构模式造成了大部分的工程师进入个体贡献者行列。好消息是，微软为个体贡献者工程师搭建了一条健全的成长路径，在这种职位上，大多数高级级别及首席级别的开发者都是个体贡献者，不必非到合伙人级别才算得上管理人员。问题是，大多数个体贡献者及他们的管理者对技术领导的定义不同。那你该怎么做？你怎么才能以一名个体贡献者的身份做到高级级别？很高兴你能这么问。

条条大路通罗马

在专栏“升级”中我对微软的职业生涯阶段及升级的通常路径作过详细描述，如何以一名个体贡献的身份从高级级别晋升为首席级别，在专栏“要么带头做事，要么唯命是从，要么赶紧离开”及“与世界相连”中我也对此做过专门论述（这三篇专栏就在本章的前面部分）。问题是，你怎么能在先铸就技术领导力的情况下再成为一名高级个体贡献者？

以下有 3 个方法告诉我们如何升至高级级别，这些方法很明确也很奏效：

1. 成为一名机构领导者（老总），通常是一名工程领袖。如前所说，这样的职位是很少的，特别在开发者这个级别。

2. 成为一名技术领袖（专家），通常是某些领域的专业人士。这是接下来专栏的重点。

3. 成为一尊活化石（贤者），通常是在团队呆了很长时间的人。这条路径是最长的，并且他的成长潜力也是最少的。

每种情况下，你对其他的工程师都有影响——要么作为管理者，要么作为思想领袖，要么出于丰富的经验。当然，很多工程师就是老总、技术专家或贤者。不过，作为一名老总是很累的，作为一名贤者则是个漫长的过程，所以我们来重点讲讲如何成为一名技术专家吧。

向专家求助

成为一名具有影响力的领导者就意味着要对其他工程师的观念及行为产生影响，在科幻小说里，这种影响是通过意念控制完成的，而在现实生活中，要通过专业技能产生影响。

一般情况下，你应该在以下领域成为有求必应者，这些领域是：

- 技术领域，如 LINQ。
- 性能领域，如功率消耗。
- 功能领域，如复制/粘贴。
- 质量领域，如可访问性。

这里的关键是当你团队里的某人在以上领域遇到问题时，他就找你寻求帮助。

> **作者注：**大多数高级及首席工程师都有至少两或三个领域的专业技能，但是为人所知的往往只有一两个。

我知道，我的名声已经臭了

一旦你选择了一个领域，你就要让你的同事们信服你是这个领域的专家。你需要在这个领域中建立个人的声望——称之为你的个人品牌。你可能想成为“LINQ 大家”或“绝对专家”。

你第一步要做的就是精通你所期望的领域的技能，你应该选一个你熟知又热爱的领域。可以通过多读书，参加这个领域的话题讨论，加入到相关的通信组列表，认识公司内的其他专家，积极参与解决这个领域与你的项目组相关的重大问题等方式实现。

如果有你的项目经理的支持那就大不一样了。你的项目经理可以帮你选择正确的工作任务，准许你参加适当的培训计划或会议，并帮助你塑造自我。为能达成这些目标，你必须向你的项目经理表达你想在你所选择的这个领域成为专家的强烈愿望。

下一步就要择机而动。如果你要在会谈中（无论书面或口头）展现你专业领域的技能，表达就要有技巧以显得你很实在：“是的，Eric Meijer 在他的网络广播（webcast）LINQ 主题中对此已有论述，正确的做法是……”你可以在代码评审、博客、通信组列表、维基网、Stack Overflow、MSDN 或其他的论坛采用这种方式，无论何时何地，只要用得着你的专业技能。只要你付出更多，真诚帮助他人，而不是只为作秀，那么受你专业技能“恩泽”的人就会越多。

质量也是个工作

“但是我该选择哪个领域呢?”一些人通常会选择他们感兴趣的领域，但是很多工程师一般会喜欢技术类的。而且，通常这些“热门”的领域里已经有你团队里的专家了，那一个菜鸟专家应该做些什么呢?

> **作者注：**如果真的想在一个已有其他工程师负责的领域成为一名专家，你还是可以作为这个领域的后备人员的。可以要求当前的专家做你的导师，参加这个领域的设计与代码评审会，修正这个领域的 Bug，当这个专家走开时要多多熟悉这个领域，最后，当这个专家离开时，你就成了新的团队专家。

一切为了质量！还有很多地方关怀不够——可维护性、可管理性、可访问性、兼容性、可持续性、有效性、团队协作能力、可靠性、可追溯性、可扩展性、可恢复性、可测试性及可用性，更不用说以下3点：全球化、安全性及隐私。该死，我可能遗漏了一些，其中有一个对你还很有用。

作为一个质量领域专家最荣幸的事是你的才学在所有团队都受用。很多与质量相关的领域未得到应得的重视，质量对于每个产品来说都是很重要的。你要不断提升个人技能水平，乃至整个职业生涯，这样无论你到哪，你的个人品牌将永远跟随着你。

锦上添花

在技能提升的过程中有时会遇上些麻烦。你希望是由你自己选择自己喜欢的技术领域，你的上司或同事可能会给你推荐一个看似很有挑战性，实则并不是你的兴趣所在甚至可能死路一条的领域。远离这些领域，记住，如果你想成为一个领域的核心人物，最好这个领域得是你喜欢的。

> **作者注：**有很多领域，如安装配置及兼容性方面，这些并不惹人喜爱但却是非常复杂且极为重要的。在你拒绝一个领域之前，先对它多作了解，你可能会发现这个领域你还是有点兴趣，而且会真正喜欢上它。

即使你喜欢别人推荐的领域，你也应该从中找些延续任务（stretch assignment）。延续任务是极其有用的——它们有利于快速推进你的职业生涯之路。不过，天上是不会掉馅饼的，如果你在延续任务上失败了，那你也差不多完了，不管你在这些工作上花了多少时间，失败后你就不会得到令人满意的评审，不过，至少你收获了经验。

为什么要干延续任务这样的活？因为你从中得到的经验是非常有价值的，长期来看，这有利于你的职业生涯更快速地发展。另外，如果你够幸运（经验也是运气的重要因素），成功了，那你就会有不错的评审。不过，你必须认清延续任务存在的风险，就像其带来的潜在价值一样。

这是什么意思

成为一个受人尊敬的专家有赖于你如何能施展你的技能，使他人的工作从中获益。如果你没有仔细倾听、理解他人面临的问题，没有做好沟通，表达清楚你的解决办法，那么你再才高八斗也是白搭，这样的话，大家很可能会抓起砖拍死你。

想了解如何倾听与回应反馈，请阅读第8章“我听着呢”；想了解如何进行日常沟通，请阅读第8章“你在跟我讲吗？沟通的基础”。这两处的精髓是，仅有聪明还不够。对照一下你认识的名副其实的专家——他们都是出色的沟通者；再看看那些好高骛远的所谓专家——他们不过尔尔。

你行的

你已经是个聪慧且合格的工程师了，离微软的高级工程师不远了，下一个目标就是成为一个普泽大众的技术领导者，这只需要多付出努力，并没那么复杂。

选择一个你热爱的领域，尽你所能熟悉这个领域，告诉你的上司你的兴趣所在，投身其中，善于沟通，多提建议，跟其他专家打成一片，随时听候调遣。

很快你就会广受欢迎，一路前进。这会让你很有成就感，很有乐趣，这赋予了你工作真正的意义与价值，你已驶入个体领导者的康庄大道——系好安全带，加油！

第8章

自我完善

本章内容：

2002年12月1日："合作还是分道扬镳——协商"
2005年2月1日："最好学会平衡生活"
2005年6月1日："有的是时间"
2005年8月1日："寓利于乐，控制你的上司"
2006年4月1日："你在跟我讲吗？沟通的基础"
2007年3月1日："不是公开与诚实那么简单"
2009年3月1日："我听着呢"
2009年7月1日："幻灯片"
2009年12月1日："不要悲观"
2010年8月1日："我捅娄子了"
2011年3月1日："你也不赖"

在我的职业生涯开始之时，我已结束了研究生学习和几次实习经历，对商界与学术界都有了印象。学术界很政治，而在商界，则有赤裸裸的度量标准摆在那对你的效率作出评测。我不知道我是否够聪明，效率够高，能在这个行业竞争并取得成功。

我用了好些时间才意识到，一旦你进入了入门级别（庸才与懈怠者都已被排除出这个级别），智商和办事效率就难以对人才的优劣做出区分了，长远来看，你的沟通技能与自律能力才决定你的成败。我认识一些头脑非常机灵的家伙，他们一开始很成功，但随着运气的消尽，他们很快就举步维艰。

在这一章中，我是你的私人"服务台"，同时回答了为什么要以及如何纠正你工作与生活中存在的缺点。第一个栏目阐述了在个人和团队之间有效协商的秘诀；第二个栏目讲如何平衡工作与生活之间的关系；第三个栏目提出了一些时间管理的工具与技巧；第四个栏目教你如何影响那些大权在握的人；第五个栏目为高效清晰的沟通清除了障碍；第六个栏目揭示了个人与企业的价值观对于两者共获成功的作用；第七个栏目告诉你为什么以及如何做好一名听众；第八个栏目教你如何应对绩效考评；第九个栏目教你如何有效处理未知且不可逃避的干扰；第十个栏目教你如何知错就改；最后一个栏目教你如何与你周边的无能之辈竞争，包括你自己。

人们每天都要遭遇挫折、糟糕的运气以及不公平，并很轻易地将原因归咎于他人。你要换种心态，反省下自己。你无法掌控什么事情在你身上发生，也无法控制你所担心的东西，但是你总可以决定自己应对的方式。好好学习，天天向上，是你能做出的最有力的回应。

——Eric

2002年12月1日："合作还是分道扬镳——协商"

如今，大家都在跨部门合作上空口说白话。我们的员工调查结果显示，几乎所有人都认为我们应该更好地合作，但几乎没人真的去那么做。哈，问题究竟出在哪里呢？

可能是部门之间有冲突吧？冲突的交付日期，冲突的版本，冲突的功能，冲突的需求，冲突的优先级、目标、客户群、业务模式、行销信息、行政导向、预算约束、资源，等等。没听晕吧？大部分团队在他们自己部门内部的合作都麻烦重重，更别说跟其他的团队一起工作了。

是的，"不过，I. M. Wright先生，"你说，"合作是好事。"Bug只需在一个地方被修复一次，不再有重复的工作；用户也都习惯于某种方式做事，用户界面也只用一种；开发者只需了解一种API；几乎没有漏洞留给黑客去攻击，也几乎不用打补丁；各个部门可以共享资源和源代码。大家可以把更多的精力放在设计上，而放在实现和测试上的精力可以减少。还有……

我们又回到了之前的话题。合作是好事，同时也是棘手的，平时跟他人共事时基本也就这样。关键是要有良好的协商技巧，而我们当中几乎没人意识到这一点，更别谈实践了。那我们该怎么办呢？我知道微软采用了两种基本的合作和协商方法。

做也得做，不做也得做

第一种方式是"无条件合作"！这也是我们这里迄今为止最通用的一种方法。通过这种方法由某个掌权的大人物督促着每个人，直到他们能够在一起很好地工作为止。（通常是一个管理人员撮合项目组之间的合作或其他事宜。）如果还不奏效，管理者就把两个项目合并在一起，这样大家就不用再想东想西了。

这就相当于通过威逼呵斥胁迫大家以你的意图做事。哈，真幼稚！没人喜欢这种粗暴的方法，除了偶尔屈服这种强权。最糟糕的"重组"就是以这种方式来进行的，这令人非常厌恶。优秀的人才会离开他们所在的部门，有时甚至会离开公司。大家的感情受到了伤害，生产力和士气像互联网泡沫破灭时的股票一样一路狂跌，几个月都恢复不过来。

> **作者注：**尽管现在仍然有人采用威逼式的合作方式，但它在微软已经不再占据统治地位了。如今，各个团队相互之间签订了协议（正如我下面将要讲到的那样），或者他们干脆共享源代码，这样要好得多。但人们常常不了解协议的究竟，以致忽略了一些重要的方面。这必然会产生问题，这就是为什么了解如何协商是这么地重要。

更好的方式

第二种方法是一种更成熟的合作方式，不过它也需要仔细琢磨琢磨。你要跟你合作的项目组签订某种协议（是的，我知道，一些自私自利的项目经理把这跟编写得很详细却占人便宜的合同混为一谈，这些合同常使他们的合作方士气受挫，最终离他们而去，不过你不必做得这么过火），你们只是预先达成了某种简单的意向：你的项目组和合伙的项目组各自做什么，你们各自需要从对方那里获得什么，以及如果合作出了问题，你的项目组，同样你的合作伙伴，应该采取什么措施。

这种方法的意思就是说，采用一种有效的协商策略，找出并解决双方存在的意见冲突最后达成共识，从而建立双方的信任及谅解与合作的基础。这是一句寓意深刻的诤言，让我们来仔细分析一下。

阴影与凶兆油然而生

当双方都赞成“通过一起工作来实现互惠互利”的主意时，问题的关键变成了如何消除合作过程中的障碍，而不是合作过程本身。如果双方都像对待我们的竞争对手一样想着如何打击对方，那以上说法就不成立了。

然而，对于微软内部的各个团队，或者在跟我们的合作伙伴一起工作时，合作面临的真正挑战恰是如何清除合作过程中的障碍。

别跟 Windows Messenger 过不去

成功合作的障碍源自于冲突、需求及信任。解决冲突，满足需求，那你们就达成了信任。所有的问题都来自于以下三个方面：

- **冲突**。比方说你想在你的应用程序里采用 Windows Messenger，但是你又担心 Windows Messenger 团队的工作时间表与你的不一致，他们的时间表可能在最后时刻改动，导致你计划失败或项目流产，这就是冲突。
- **需求**。你需要 Windows Messenger 团队保证按你的产品发布日程表为你提供稳定的代码，你需要他们根据你们对 API 的要求检验他们的创建（builds）。

另一方面，你也要满足 Windows Messenger 团队的需求，他们可不想让你也把他们搞得这么痛苦，软件功能不要有太大的变动或要求；不想增加额外的本地化成本；他们希望你们采用他们的配置模式，这样其他使用 Windows Messenger 的应用程序就不会出错了。他们希望事先与你们达成共识原封不动地使用这个组件，以消除不必要的本地化成本并按他们的配置模式放到你们的应用程序里。

> **作者注**：充分理解你们合作伙伴的需求与顾虑对于成功的合作协商来说是非常重要的。要确保你们能及时了解对方的处境与需求，不是妥协就可以的，互相了解是建立信任的长远之计。

- **信任**。你们的协议可以采用非正式的 E-mail 或文档来说明你的团队同意使用 Windows Messenger 的配置模式并省下额外的本地化成本；他们的团队也认可按你们的产品发布时间表为你们提供稳定的代码（用 Source Depot 很容易实现），并用你的 BVT 检验他们的产品；你还可以注明当双方对彼此不满的时候该怎么办。这样，当双方的产品单元经理（product unit manager，PUM）同意了上述内容后，你们就可以成为合作伙伴了。

> **作者注**：“创建验证测试（BVT）”用来验证某个软件是否符合某些需求。在采用其他团队的产品之前利用 BVT 来验证一下是一个非常值得提倡的做法。把你们的 BVT 拿给其他团队让他们自行验证他们的产品就更好了，这样的话当他们满足你的需求时你们就不必为一个无法正常运作的创建劳神了。

自此以后，你就可以定下产品开发的地点与时间表，请求他们实现对 API 的要求。当你们的要求得到满足并取得信任之后，这些就变得很简单了。

皆大欢喜

维持这种信任的关键是要保持一种开明顺畅的沟通。要保证双方的产品单元经理能对开发时间表或功能的变动、存在的问题以及奇思妙想或意外之喜畅所欲言。这样做并不难，但是如果某一方忘了这一点，那就可以跟这个项目说 byebye 了。

> **作者注：**“哈佛协商计划”推荐了一个便于记忆的法则 ABCD：先协商再决定（Always Consult Before Deciding）。也就是说，在未与合作伙伴沟通之前不要做重大的决定。

如果你们按这种方式进行沟通，也即消除冲突、满足需求并取得互信，那么你无论做什么事都会异常有效。人们经常容易犯的错误是在还没认真聆听别人意见的时候就直接提出解决办法。如果你过早地就提出解决方法，没人会理你，他们会担心出什么问题；如果你善于倾听，多提问题，预先解决存在的问题，那大家就会欣欣然接受你的想法。

> **作者注：**在协商完成后一定要让每个人都觉得是赢家，感觉损失就不对了。使大家都觉得是赢家的最简单方法就是让每个人都觉得是胜利团队的一员——比如，“这是‘我们’最完美的设计了。”

这样的合作方法不是天方夜谭，也不仅仅是在团队之间才有用。好的协商技巧在处理家庭关系、邻居关系、自己团队内部的关系及合作伙伴关系中都是很有用的。所以，让“无条件合作”见鬼去吧！

> **作者注：**在第 2 章的“有我呢”中对处理依赖方关系作了更多的讨论。

2005 年 2 月 1 日：“最好学会平衡生活”

警告：这是一个感伤、发人深省的专栏。如果你继续，风险自负。

我之前从不想在微软工作，我很享受我原来的工作，那些工作让我感到很满足。在微软工作就意味着放弃我的生活，我的家庭，没日没夜地工作，把公司放在所有事情的首位。我曾有过很多朋友在微软工作，他们经常叫我加入到他们中间去，我总是说：“不。”

后来我决定要改变一下，于是接受了 Craig Mundie 以前的一个部门提供的一个职位。当时我已不年轻，也不是校园里刚出来的小伙，我有我的妻子，我的家庭，一个 2 岁的儿子，还有一个在肚子里怀着，我根本没打算离开他们。我告诉我未来的老板说，我是个有家庭的人，我要在每天早晨看着我的孩子去上学，在每天晚上与他们共进晚餐，我只能接受这样的职位。让我惊讶的是，他诚恳地同意了。更重要的是，他说到做到。

> **作者注：**Craig Mundie 现在是微软战略研究办公室的主任。

平衡是关键

在我与公司里的开发人员参加的讨论会中，平衡工作与生活的话题经常被提及。真没想到，我随便遇到哪个人都会说起这事儿。很多跟我交谈过的公司员工他的第一感受就是他们必须在微软与他们的个人生活之间做选择，唯一不同的是这种诉求的急切程度。

这实在太悲惨了，这不仅仅是种哲理的反思，是活生生的个人事例。我见到过人们为此离婚，失去了对孩子的抚养权，由此而生病或精神颓靡，失去了友谊，打乱了个人正常的生活状态。自我 10 年前加入到这个公司以来，直到过去的一年，我还是看到这些事在我的朋友与同事的身上发生。

> **作者注：**真正的悲剧。如我下面将要讲的，所有这些悲剧都是不该发生的，微软以及其他类似的公司也不想这样的事发生在他们员工的身上。

我们是人，失去工作生活的平衡将失去自我。通常有三种情况可能使我们自寻烦恼：

- **我们总是把工作放在首位。**如我所言，结果自不用说，崩溃得很。
- **我们对工作有不同的价值观。**价值观决定自己是怎样的人，想成为两种不同的人就非常拧巴。
- **我们总是把工作与家庭分别对待。**以两种方式生活是很压抑的，也不可能处理得好。

光说不练

我们的头儿也曾说过，平衡至关重要。比尔·盖茨在“一起改变世界”的讲话中曾提到，“要让在微软工作的人都有事业取得发展的机会，得到具有竞争力的报酬及工作与生活的平衡。”史蒂夫·鲍尔默也讲过他自己家庭生活的重要性，尤其当他的家庭人员增多之后。但是，显然这些话语并没有成为众多员工的现实。

> **作者注：**如我在第 1 章中所说，史蒂夫·鲍尔默，我们亲爱的 CEO，在工作生活平衡上身先士卒。很多次，在他为他儿子的篮球赛加油鼓劲的时候，或他们跟他的妻子一同出去看电影的时候，我遇到过他。

不断抱怨无法处理好工作生活的平衡是很容易的。虽然几乎每个人都声称希望能取得平衡，但对于管理者来说要安排好生活与其他公事孰先孰后，有时是非常困难的，即使支持工作与生活平衡的管理人员也往往会无意识地发出相反的信号。

比如，在非常忙的时候，一个管理者可能会预先为那些“选择”迟点离开的人安排晚餐。所以早上 10 点到公司的开发人员，呆到晚上 8 点，一天工作有 10 小时。但是他们有孩子的同事会在早上 9 点到，而只工作到晚上 6 点（9 个小时），然后回到家，在晚上 9 点时登录系统一直工作到晚上 12 点，总共工作时间 12 个小时。这些“早些离开”的开发人员就会因为感觉离弃了团队而很有罪恶感，还没有免费的晚餐，而事实上却工作了更长的时间。

这样偶然事例并不一定适用所有人，这只是想举一个令人信服的例子：工作很努力的人会在

无意间被惩罚。如果管理者想提倡一种公平、平衡的工作环境，他们必须要有一种公平、平衡的奖惩体系。

但是工作与生活平衡的责任不应全由管理层来承担。比尔的下一句话是："在一个快速发展、竞争激烈的环境中，这是微软与他的员工们共同的责任。"假如管理层注重结果，那由个人承担工作生活平衡的责任是合理呢还是自毁前程呢？从某种程度上来说，我已找到了使工作与生活平衡既能实现又能各不受侵害的方法，如果你认为我这种说法很新奇，稍等片刻，听我讲讲这种程度是什么。

我的支票簿都快无法平衡了

在你的支票簿上实现平衡不是那么简单的，更不用说你的人生了。以下是我为使工作与生活得以平衡安排的5个步骤：

1. 了解并接受你的生活方式。第一步首先是要了解你自己，你优先考虑的是什么？是事业为先家庭为后吗？你的制约在哪里？你会放弃一个不是学校组织的家长会，以换取你事业的进步吗？你必须了解并接受这些选择，即使这些事很少发生。当你跟你的经理谈论你担当一个职位是发挥你的长处还是遭罪时，这样做将使你有所准备，他或她也会尊重你的选择并支持你。

> **作者注：**根据我以往的经验，这一步是最难的。大部门人从来都没有面临过不得不做生活选择的状况。不要欺骗自己，对工作与家庭之间的关系做出明确的决定确实有难度，但这也确实很重要也很有价值。

2. 跟你的管理者商定基本的规则。每当我向一位新来的上司汇报工作的时候（这在微软是经常有的事），我表达了我对工作的期望："我想每天早晨看着我的孩子去上学，每个晚上都跟他们一起共进晚餐。如果这个要求你不能接受，我想我还是换一个岗位。"我们也承认，凡事皆有例外，但是必须要设立明确的基本规则。没有一位管理者曾拒绝我，而且坚持我的基本规则也没有对我的事业发展有什么影响。不过，经常出差的工作不适合我。很多我以前的上司曾跟我说他们认为我这种是非分明的价值观正是我的长处所在。

3. 不要那么快妥协。偶尔打破常规也是没问题的，但是出差到日本两个星期对我来说确实是个大难题。当遇到这样的情况的时候，我把它当做重申我的条件的机会，通常总会有另外的解决办法，有时或许我也需要去一下。总之，我再一次向我的上司重申我的想法，并提醒他或她当初做过的承诺。如果你很容易妥协，那就是告诉你的上司你对这件事并没那么在意，你的上司很可能会继续向你提出更多要求，直到你最终无法接受，再提出一个新的并不合你意的条件。

4. 需要的话可以用RAS、远程桌面或OWA。当然，每年总有那么几个星期是很忙的。在终端服务器（远程桌面）出现之前，忙就意味着在将我的孩子抱上床之后我还得再继续工作。最近一段日子，在他们睡觉之后我经常在家里登录系统——不是因为我总是很忙，是因为我爱我的工作并喜欢做这样的事。但是我还是花了好几个晚上跟我的妻子看看电视或电影，我要做的事都是以建立一种我与我的家庭所需要的平衡为基础。

> **作者注：**远程访问服务（RAS）是从家里登录到微软的内部网络的一种手段。Outlook网络访问（Outlook Web Access，OWA）是一种AJAX应用，允许你通过互联网连接访问到公司的E-mail、日历和联系人。

5. 抛开造成分离的精神分裂伪装。我游离于工作与家庭间有多年了，我总是被告知就应该如此，不管这样让人感觉多么不适。现在我只有一种生活，只有一种。7 年前家人的一次病危才迫使我揭开这荒唐的伪装——我可以把工作与家庭分开对待。没有人可以很方便地将他们的生活一分为二，除非他们有严重精神分裂症。所以，家庭与工作要同样对待，既要两相适宜又能责任到位。

有平衡才有一切

要明白我们需要什么才能过一个完整的生活，然后按照那些标准生活，这是我们能给自己的一件非常好的礼物。这里的部分含意就是，不要对工作与家庭分得这么清楚，这样我们就不用经常地对我们的价值观及内心思想进行切换。但这并不是说我们要把所有在家里的时间都用在远程工作上，更不是说我们要把在公司工作的所有时间都用在跟朋友和亲人聊天上。这里还有层意思是我们必须尊重同事们的隐私与诉求。记住，微软的价值观是开放与尊重，不仅仅只是开放。

但是如果你真将工作与家庭合二为一，那就两全其美了。你在家庭生活中学到的经验就对你的工作有帮助，你在工作中得到的经验又对你的家庭生活有帮助。在带来幸福感的同时，平衡也大大有利于个人的成长。你可能永远也成为不了那种妄自尊大的工业巨头，但你获得的巨大财富却不是那么轻易就会失去。

2005 年 6 月 1 日："有的是时间"

"用巧力，而不是莽力。"你不想将那些能说会道、废话一堆、自以为什么都懂的家伙推进绞肉机里，让他们看起来更特别一点吗？特别是当这些高调来自于一个中层管理者，而这个人把 18 个所谓"高级别"项目分配给你的团队，然后招集一帮不学无术的莽徒与毫无头脑的蠢瓜加入到你的团队里来，任由他们来打断你的思绪让你无法正常思考。

从来没有足够的时间处理你的待办事宜清单中哪怕一丁点的事情。即使你的工作量并不很大，不厌其烦的干预与会议会让你要好好完成一项工作成为笑话。

然而，这是一个开发者或管理者工作的典型写照。合格的管理者把大半的人挡在了他们的走廊上，成功地度过每一天；优秀的管理者知道怎么管理自己的时间。

直接告诉我要点

有无数种有关时间管理的书或日历本。大多数在我看来都肤浅得让人汗颜，全无可行性，或者是为来自外太空的外星人写的，他们把吹毛求疵当成了高等社会的标志。

在我看来，时间管理可以通过以下三方面做到：

- 消除干预，避免三心二意。
- 将任务委托给其他团队成员。
- 以优先级别及重要程度为出发点，严谨选择工作条款。

原谅打扰

要有效率，你需要集中精力。近期的研究揭示了两个很有趣的发现：

1. 过度的思维切换比吸食大麻更伤 IQ。
2. 当员工只参与两个项目时最富生产力。

第一个发现并不让人吃惊，由于不断地被打扰和思维切换使你不能集中精神，你也就不能正常思考。第二个发现就有点让人费解了，它的意思是你一次只能把精力放在一个项目上，但如果你在第一个项目上遇到了难题或需要暂时放下这个项目，你就有必要换个项目来做。第二个项目就是一种调味剂，也就是说，很有用但并不必需。

有些打扰是很容易控制的。我关闭了所有形式的 E-mail 提醒，并把电话铃声调到尽可能低的状态（电话铃声只设成轻轻的点击声）。这样的结果是，我再不用受 E-mail 或电话的打扰了，当我准备休息一下的时候我再回复，而不是谁点一下发送按钮或拨打我的电话号码时我就回复。跟之前相比，这并没有让我变得响应迟钝，我只是作了控制调整。

当我有了合适的休息时间就会浏览我的 E-mail（通常是每 10 ~ 40 分钟一次），尽力回复每一封邮件。每条邮件信息代表着一次思维切换，这样你就减少了每次读邮件时思维切换的次数（精益的拥趸者称之为“单片流”）。大概超过 95% 的邮件可以删除，或放到文件夹里归类，或转发给另一个人，或通过前台接待直接回复。

> **作者注：**要了解关于 E-mail“单片流”解决方案的更多内容，请阅读第 3 章的“世界，尽在掌握”专栏。

有些人可能会说这样做会在细枝末节上浪费一些不必要的时间，可是，你必须阅读每一条信息以确定它是不是细枝末节。当你读过这条信息后，通常一段时间之后你要花更多的时间把思绪切回再重新看一遍信息，这甚于马上就作个简单的处理。另一个好处是，当你的邮箱几近清空后，要查找一封独特的邮件看看到底有哪些事情要做就花不了多少时间了。

要说 Instant Messenger，我压根没用过它。我相信 IM 是撒旦给我们来诱惑小青年并要毁掉我们生活的。当然，这只是我的个人看法。

> **作者注：**我的长子现在 10 多岁，我支持 I. M. Wright。不过我要对 Messenger 团队中的朋友说，“我无意冒犯。”

找到你的乐土

避免打扰的另一个办法是玩失踪，到一个没人可以找到你的地方完成你的工作。带上一台笔记本电脑或在一个书报亭使用远程桌面，你可以在几乎所有的地方工作，会议室、会客室、咖啡馆等所有你想得到的地方，直到所有的人都认为你只是在开会。你不能总是这样做，这样会伤害到你的团队，但当你很忙的时候这是个很好的办法。

你还可以在身边没人的时候工作。大多数开发人员在早上 10 点左右上班，一直工作到晚上 7 点。如果你在早上 8 点开始工作，你就有了 2 个小时的无人打扰时间，然后在下午 5 点就下班，

如果需要，你可以安顿好所有事情，晚上 8 点后在家里工作。如果我还有另外的工作没完成，我会一直等到我的孩子们睡觉之后再开始。

最后，你可以在一星期中安排一段“专注时间”，这段时间里别人基本上不会打扰你或你的团队，除非十万火急。要使这样的想法得以实现，你必须选择一个可预见性的不太可能有人打扰的时间，一个星期后半段的某天或者一个下午或者一个晚上是很好的选择，因为大部分会议及紧急事件一般发生在前半个星期。

我们谁也不笨

打扰的另一种特殊形式是会议。会议让你不得不放下手头正顺利开展的工作，并把你带入到一个备受折磨，又浪费时间、浪费生命的无底黑洞中去，然后再也无法回复到原先的工作。不过，这里有几个办法可以减少会议带来的影响：

- **不要参加。**你必须要参加的会议很少，一对一会议、项目进度会议及员工会议。几乎其他所有的会议都是某人的一家之谈，如果感觉某次会议可去可不去，那就不要去，如果没什么糟糕的事情发生，也不要去。
- **让别人参加。**委托别人参加这次会议（下一节对此将有更详细的讨论）。
- **开一次高效率的会议。**对于参加的会议，尽量开得有效率些。请参阅第 3 章的“我们开会的时候”中关于召开紧凑会议的内容。
- **把所有的会议安排在一起。**我知道这听起来有些不可思议，但是这个想法可以减少思维切换。首先，在一个星期的前几天安排项目及员工会议，然后把所有的一对一会议紧挨着它们安排。当然，这个星期的前几天会非常痛苦，但是到了周中或周末你就会有一段免除打扰的时间。

有难同担

虽然减少打扰这种做法很好，但要完成工作的最有效办法是交给别人去办。主管或经理们的生活如此之忙是有其原因的——除了要完成项目还要管理整个团队。要取得平衡的很重要一点是委托。

> **作者注：**架构师同样可以通过这种方法利用架构团队，虽然这个团队的成员并没必要向这个架构师负责。

新晋的主管及经理们存在一种倾向，他们并不是让团队来承担工作重担。他们更愿意自行承担压力与责任以免让人看起来能力不足或懒惰。这样做很愚蠢、很怯懦也很自私，这给你的团队带来更糟糕的结果就是你不堪压力，当你不堪压力后，你就会对大家变得很鲁莽，你就没心思细想并倾听，就会做出错误的决定。没多久，你的团队在你的带领下也变得压力重重而无法运转。

你必须是你团队的中坚，在其他人失去理智时你要保持清醒的头脑，在你的能力范围内你只要做到这一点就可以。把所有可以转交的工作交给你的团队去做，记住，他们唯一所希望的就是支持你并讨你欢心。

为什么你的团队要支持你并讨你欢心呢？你傻呀，对他们的评审是你来写的，这决定他们的晋升。他们需要接手那些让他们的评审得分4.0的工作任务来提升他们的职业生涯，你所接手的任务正是他们需要的。为什么把你的下属们排挤在外呢？为什么要大包大揽这些艰巨又关键的任务呢？把这些工作放手给你的团队，满足他们让他们进步。

> **作者注**：4.0是微软旧的评级制度中很高的得分，这种制度的得分范围为2.5～4.5（得分越高，报酬越多）。3.0分算是合格的，大多数人期望并得到3.5分以上。

告诉我应该做什么

当你委托你的工作时，一定要方式得当。委托的诀窍是委托所有权，而不是任务。

这里的区别有些晦涩，所以我来打个比方。比如你的项目是与Media Player相关的，现在安排你与Media Player团队进行一次会晤。你并不想与会，所以你打算让Anil——你团队的一个想成为主管的开发人员去参加。

如果你只是把会议这件事交代给Anil去参加，我知道接下来会发生什么。无论你跟他交待得有多仔细，Anil会被问及一些他毫无准备的问题及做一些他无法做的决定。之后他会找到你跟你重述会议的内容，你也会问各种各样他无法回答的问题，最后，你们两人都会觉得你应该亲自参加这个会议。结果是你会感到很挫败，Anil也会感到很失败。

现在，假设你不只是把这次会议交由Anil来参加，你把所有与Media Player相关的事物都交给Anil，你把Anil叫到你的办公室并对他说：

“Anil，Media Player的相关事宜就全权交给你了。你要确保他们从我们这里获得技术需求，并能保证完成他们的要求。你要自行决定与他们团队间的工作，了解他们的API，并在会晤后自行进行设计与开发。怎么样？哦，对了，第一次会议很快就要召开了。”

Anil会很喜欢这样。他会满怀信心地参加会议，回答问题并达成协定。在身负责任的同时他会有大权在握的感觉，Anil正向着成功出发。期间，你不仅仅与这次会议不相干，你与所有将来的Media Player会议都扯不上干系。太不一样了。

他不过是个小孩

如果Anil不过是个新手，把所有权委托给他会有风险，以下有几个办法可以消除这种风险：

- 跟他一起参加开头的几次会议，但是要管好自己的嘴；最大限度地放权给Anil，迟些时候再向Anil询问有关情况。
- 给Anil找个指导员，这个指导员就是我刚才提及的能担当类似角色的人。
- 找一个你在Media Player团队中的朋友，要他监督一下Anil，如果有什么问题的话要让你知晓。
- 要求两个部门之间来往的E-mail都抄送给你，并且要求Anil提供定期、详细的进度报告。
- 组合使用上面这些方法。

不管你怎么做，Anil拥有了与媒体播放器的关系，他也有机会一展才华，同时你也得到了更多的时间去关注其他的领域。

你应该休息一下

委托所有权的方法几乎对所有类型的任务分配都很管用。任何时候你有任务要转交出去，哪怕只是在你去看牙医的时候帮你打理一下，停下来想想整个任务的来龙去脉，把全部的所有权交出去。

顺便提一下，如果你在如何委托上遇到什么问题，就给自己放两个星期假，你就不得不将你所有的工作给委托了——当你回来工作时，你就可以让你的团队来完成工作了，你还额外多得了两个星期的假期，多好啊！

井然有序

最后一个时间管理的主要问题是你应该选择完成哪些工作条款，以及按什么顺序来完成。首先要列出你当前所有的任务选项，并决定选择哪几个。

工作条款可以归为以下几类（按优先级次序列出）：

- 需要你亲自关照的任务（打理团队、评审、一对一会谈、员工会议、雇员关系问题）。
- 与你的个人发展目标密切相关的任务（培训、关键任务分配、行政或客户约见）。
- 能让你与你的团队并肩作战并对他们进行高效管理的任务（士气激励、项目及团队会议、设计与开发评审、Debug 调试、分诊会议及精选的项目工作）。
- 你偶尔很感兴趣的任务。

第一类的所有工作条款必须列在你的选择列表里。通常有些任务类型是交叉的，这些就是具有高杠杆效力的工作条款，必须放在你的选择列表里。其他则可以忽略或委托给他人。

你的工作列表斟选过后，你要对它进行排序，要考虑我前面所提类别优先级、这个任务的杠杆率以及紧急程度。最后，你应该只有两或三个主要的项目及几个相对次要的工作条款。

记住，一次只全神贯注于一个项目，在你遇到困难或需要休息的时候再换到另一个项目上去。

想怎么干就怎么干

一种你可以选择的具有最高杠杆效应的工作条款是直接性项目工作，如获得、设计和开发一个功能模块。项目工作使你可以深入洞悉员工问题（创建问题、团队个性和跨团队互动）。与时俱进，将有助于你个人职业的发展，使你能以一种直接又紧密的方式与团队或更大的组织机构更融洽地相处。假如你选择了正确的工作任务，你铁定会喜欢它的。

有些人在争论一个开发主管或经理是否应该从事代码工作。对于我来说，直接性项目工作对你以及你的团队是很有好处的，你也无法避免不参与这样的工作，诀窍是选择一个合适的任务。如果你有 3 个以下的下属，你将有足够的时间参与到几乎所有的项目工作中去，然而，如果有 4 个以上的下属，人为及项目相关的问题就会频繁发生，你在其中的作用就变得不可预知了。

工作日程的不确定在项目早期还没什么问题，然而，当接近项目尾声时，那些来自各个方面的同事们就要求你有一个交付承诺，但你却不能给出明确的时间，你就会把未完成的工作交给你的团队成员，而这些成员还在忙于应付他们自己手头的工作，而且，你也没有时间将你的工作完全地转交过去，这会让接手的人感到非常头痛。这样的话，你的同事们会讨厌你，你的团队会讨厌你，你也会讨厌你自己，因为你必须放弃或作出让步。

如果你选择的项目不在关键之列的话，情况就不一样了。具体来说，当有3个以上下属的时候你应该选择这样的项目工作：

- 不需要发布。
- 要取消很方便。
- 比较有风险的，有趣的，很酷的，能给客户带来意外惊喜的。
- 最好是，它能很方便集成到产品中去的。

当你选择了具有这些性质的项目工作后，则那些做其他工作的同事向你要交付承诺时，你可以说："诚恳地讲，我想我可以在你要求的时间内完成，但是我不敢保证。不过，如果我没有及时完成，要取消这项功能很方便，它不会跟产品一起发布。"现在，你的同事们就安心多了，你的团队也不会受影响，你也不会有什么压力，也会感觉很自信。没错，只能这么干。

顺便提一下，要找到这类项目工作并不困难。通常，能方便集成到产品中的这些既有风险又有趣还很酷的功能很少在关键路径上。所以你可以把这些工作分配给自己，并享受它们原本的快乐。做个管理者就是这样好。

运筹帷幄，决胜千里

主管或经理要控制你是很容易的。不断的打扰，让你焦头烂额的承诺，还有跟"实际工作"的脱离，会让即使最有能力的人也不敢奢望当初接手的工作会相当简单。

不过，还是有很多方法可以让你夺回工作的控制权，减少打扰，给帮助你减轻负担的员工创造机会，削减工作量只留那些能为你及你的团队带来最大利益的工作。用这些方法可以让你重新获得自主权，这正是一个积极能干的领导应该做的。

2005年8月1日："寓利于乐，控制你的上司"

你可以摆出一副极高的姿态，以示你对某些自恋自怜的人是多么不屑一顾。你拇指和食指的指尖捏在一起轻轻地拉着，说道："这是世界上最小的一场小提琴演奏，'我心为你而泣。'"

当人们在看起来不可一世的领导面前倾诉他们的无助时，他们给我的感觉就是这样的。"唉，管理层永远不会给我们时间提高我们创建的质量或改善我们的做法。""我希望我们的管理层也来参加这个培训，那是我们唯一能够进行审查的办法。"（见第5章的"复审一下这个——审查"。）"我对我们目前产品的取向及团队组织方式很不爽，但是我毫无办法。"

成熟些吧，小朋友。想以博取同情为借口使自己懈怠，将一事无成。难道你想不到你的上司也会同样跟他的上司这样说吗？如果你不付诸行动，那你期望的改变也不会发生。就这么简单！你与当权者的区别不在于你的权力级别是多少，而是你是否愿意行动。

我没辙了

"没错，但我的上司不会听我的。"这是最常见的反驳。"她有数不尽的理由说为什么我们不能改变一下。"嗯，不错，至少你已经从一个可怜人变为一个愚昧者，你已经在努力试着改变自己，只是有些愚钝。放轻松些，大多数人是这样的。

遗憾的是，没有权威的影响力很少是与生俱来的，这是一种后天的能力。当要出台一项解决方案时，大多数人单刀直入，他们把想法直接告诉他们的上司，唯一的结果就是遭到否决；有些时候人们是做足了功课的，写了长篇大论的白皮书或演讲稿，结果也就是被断然回绝。

你可能不太明白如何做出适当的准备，如何有效地表达你的想法，如何使你的想法付诸实施。我们来仔细分析一下。

> **作者注：**我曾经就这个主题举行了一次内部的讨论，登记表格瞬间就被抢注一空。这说明一个问题，很多人都想更多地了解权力之外的影响力。.

知己知彼

要做适当的准备，我要先列出一连串的步骤，但是它们可以在一天之内完成（有些小问题只用几分钟）。

- **了解你的提议。**你的想法的风险是什么，你怎么化解这些风险？什么东西可以改变你的想法，你又有什么核心原则是不能妥协的？要实事求是。
- **了解你的历史。**是什么原因造就了你目前的这套流程和组织结构？
- **了解你的对手。**谁更喜欢目前的状态？为什么？他们中是否有人足够强硬或足够激进，使得改变难以发生？你将如何安抚他们或者甚至把他们拉入你的阵营？
- **了解你的朋友。**谁对目前的状态感到不满？为什么？他们喜欢你的想法吗？支持你的团体有多强硬、庞大并富有激情？
- **了解你的管理层。**别人如何评价你的管理层？从你的管理层角度看，有什么好处是值得他去冒险的？你能给管理层带来更多好处或者降低风险吗？

听起来有点复杂，但如果你已经注意到你同事们的反应的话，和1~2个朋友通过一次坦率、简短的讨论就能了解一下问题的大概。跟一些人交谈对于了解事情来龙去脉，理解存在的问题，是很有必要的。不管怎么说，多了解是成功的关键。

适者生存

现在你知道要面对的是些什么问题，相应的你也该对你原本的想法作出调整及提炼：

- **想好你要取悦于谁，安抚谁，不在乎谁。**是的，你想让每个人都开心，但是有时候把重心放在少数人身上，让他们满意就可以了，只要不会有人因此崩溃，比如你的上司。有些家伙只要你不伤害他们，就可以与他们相安无事，另外一些人如果你的上司怎么想他们就怎么做，或是他们根本就不关心你的上司怎么想的，那你也不用理会他们。
- **为取悦于人，就要投其所好，消除风险。**用你获取的信息总结出几个好处以加深关键人物对此的印象。如果你的管理层关心效率问题，就谈谈生产力的提升；如果是客户满意度，就谈谈质量及网络连接；如果是按时交付，就谈谈可预测性及透明度；如果要消除风险，就谈谈后备方案，决定什么时候继续做或不做，讲讲备份措施，或清晰的优先级别。

> **作者注：**这是一种消除威胁并满足需求的协商方法，详情参见本章前面的“合作还是分道扬镳——协商”。

- **要安抚他人，必须妥善处置他们发现的任何有威胁的东西。**通过调查了解找出什么是有威胁的，如果有风险，就要化解；如果他们有另一套解决方案，要接受它，并给他们足够的信任及关注（就像“这就是我们的想法”一样）；如果他们还有其他的要求，要么马上满足他们，要么等到下一个版本来完成。

最后，你的计划就会有一大批的拥趸者，没有谁会处心积虑地反对它，你也会满足主要决策者所关心的问题。现在你已准备如何表达了。

把水卖给鱼

如果鱼的信誉度很高，把水卖给鱼就没那么难，你需要做的仅仅是给它们了解一下生活没有水会是什么样子。当你要想有所改变时，事情是一样的，在你提出你的解决方案之前，你必须指出现在存在的问题。

要点在关键人物身上，通常就是管理层，用你调查了解到的信息，把关键人物认为的好处总结一下，以同样的方式，把关键人物关心的问题讲给他们听。那些问题应该放在标题之后的第一张幻灯片上，而且要非常简洁明了。

以下几个重点需要注意：

- **只考虑你自己或你个人的意愿将对提议不利。**提议应围绕着关键人物、团队及客户展开，而不是你自己或你希望得到什么名声、荣誉或高级别的评审得分。
- **想向关键人物表达你的想法，你必须深入体会他们的内心想法。**以他们的方式，多讲讲他们的好处并解决他们关心的问题。如果这个解决方案是因为你关心的问题才产生的，那这个方案也属于这个团队，而不是你，是属于关键人物的，而不是你的，你必须放下对它的哪怕一点点牵挂。

> **作者注：**说实话，深入体会关键人物的内心，而将你自己的感觉搁置起来是很难的，但这对于成功却是很重要的。

- **如果你一开始就讲解决方法，那没人会理睬。**如果你无视存在的问题，那就会失去了前进的动力；如果你先夸夸其谈讲解决方案，然后才是人们关心的问题，人们就会只注意到存在的问题，而忘了你讲的解决方案。所以，应先讲存在的问题，再谈论解决方案。
- **你的陈述必须简短——非常短。**改变会引起讨论甚至争论，争论会浪费很多分配给你的时间。如果你没在两或三张幻灯片中说明你的想法，你就会在论战中失去先机。

势利眼

你应该在第二张或第三张幻灯片上陈述你对未来的展望，这种展望应该简洁明了，并有可行性。依据关键人物的利益，把目标清楚地陈述出来。简单来说，如果你不知道你要走向哪里，也就更谈不上到达目的地了。

你可能还准备了大量的幻灯片，都是与你对未来的展望和提议相关的各种各样的数据和细节，你甚至可能还写了一份 30 页的白皮书。这些材料对于描述你提出的想法的前因后果非常重要，但它们应该作为辅助材料，归入附录幻灯片和资源链接。要成功，你就必须简洁，非必要的其他所

有东西都只能作为参考。

最后一张幻灯片要解决的问题是，你如何从当前情况扭转为以后设想的那样子，或者说你如何达到目的。这张幻灯片只有两个部分：

- **论点部分。**指出如何消除风险和顾虑。这部分要说明你的后备计划、决定什么时候继续还是不继续、备份措施及优先级等事项。
- **后续部分。**指出谁做什么以及什么时候去做。人们大多时候只关注需要做的事情，而对这些事情由谁去做、什么时候去做却忽视了。如果不指定具体的人和具体的目标日期，工作就会举步维艰。

就是这样。一张标题幻灯片，接着就是问题陈述幻灯片、未来设想幻灯片和转变措施幻灯片。现在你已准备好将你的想法付诸实施了！至于一些小点子，你可以通过 E-mail 来完成，不过准备工作是一样的。

> **作者注：**有些人怀疑，区区的 3 张幻灯片无法装载下所有的信息。他们问我，是否能提供一个样例。我确实有个最好的例子，但那是一项微软的机密提议，它把所有信息放在了一张幻灯片上！它上面画了横线和竖线把幻灯片分成了 4 个象限：问题（在左上象限列了 4 条）、解决方案（在左下象限列了 3 条）、论据（在右上象限列了 6 条）和后续安排（在右下象限列了 4 条）。

付诸实施

你现在准备好跟关键人物“交战”了。有很多方法可供选择，但没有哪个能够绝对优于其他的，要具体情况具体分析，在不同的情况下选择不同的方法。总的来说，有如下 3 种基本的方法：

- **跟关键人物逐个交谈。**这种方法适用于所有的关键人物并不都能融洽相处时，或者他们各自有着不同的利害关系的时候。虽然一个一个谈下来会花费更多的时间，但这常常总是一种安全而有效的方法。
- **跟所有关键人物一起开个会。**这种方法适合用来达成共识，以及找出潜藏的问题。这样会快些，不过如果大家已经对问题达成了某种共识，这种方法就能发挥最佳的效力。必要的话，你可以首先使用上述第一种方法去获得初步的共识，然后再通过这么一个会议一锤定音。
- **只瞄准最上层的关键人物。**这种方法适用于当最上层的关键人物特别强势，或者组织结构特别倾向于服从的时候。如果除了最上层的关键人物之外没人真的介意你的想法时，你就可以使用这种方法。

再仔细检查一下，准备好实施你所建立起来的这套流程。记住，它不是事关你个人——它事关团队、产品和客户。让人们尽情地讨论，只要他们把注意力集中在你指出的问题上。如果新的问题或风险出现了，一定要把它们记下并且修改相应方案。

放飞梦想

整个过程可能看起来非常麻烦，尤其是你的解决方案还未必能“存活”下来，更不用说“茁壮成长”。你可能觉得不值得那么做，现状对你来说是可接受的，或者至少你可以容忍。也许你

是对的，但或者是你缺少了点勇气。

那么，请你停止抱怨你是多么无能为力或者管理层对你是多么漠然吧！如果现状是可接受的，那么就接受它并继续前进，如果不可接受，那么就采取行动。不管会发生什么，你至少会让大家意识到问题的存在，并且可能还会促成改变。除此之外，你将获得领导经验，也许还能得到领导的职责。最后，你会变得比以往更加有实力，因为你做到了个人意愿和勇敢行动的统一，你不再只关心你个人，而是更加关心你的团队。

2006 年 4 月 1 日："你在跟我讲吗？沟通的基础"

我读了很多的 E-mail。参加了很多的会议，看了很多的源代码和规范书，参加过很多的复审，看过大量的白皮书，参加过大量的演讲会，除了我的生命已被吞噬殆尽之外，我终于意识到了一个问题：大部分沟通都是对时间的一种可怕而悲惨的浪费。

那很令人吃惊，因为初级工程师和高级工程师之间的主要差别只在于他们的影响力和感染力。因为这里的所有人都很聪明，影响力和感染力最主要的驱动力量是高超的沟通技能。你可能认为大家都能理解这一点，当然，在大部分事情上，我也那样认为。

我已经批判过了冗长的会议（参见第 3 章"我们开会的时候"）、糟糕的规范书（参见第 3 章的"迟到的规范书：生活现实或先天不足"）、糟糕的规范书复审（参见第 5 章的"复审一下这个——审查"）和没有重点的演说（参见本章的前面一个栏目"寓利于乐，控制你的上司"）。然而，沟通对于合作、成长和团队工作的重要性是不言而喻的。毫无疑问，自古埃及时期发展到现在，人们已经学会了如何去有效地交谈。但是证据在哪里呢？我们的 E-mail 和会议都那么臃肿、空洞，我们的源代码和规范书都表达得不充分、晦涩难懂，我们的白皮书和演说尽显出自私和放纵。

> **作者注：**很显然，我在这里又把问题夸大了，在微软内外都不乏很多完美沟通的例子。然而，如果敲响警钟能够带来更加简练的 E-mail、更加清晰的规范书和源代码、更加吸引人的演说和论文、更加简短的会议，那我会全力以赴去这么做。

为什么？什么事情这么难？这是我们多么宝贵的时间啊，它们都在不经意之间被白白地浪费，其问题的根源在哪里？在经过了多年的研究之后，我相信我最终找到了答案：人们没有充分地为"我"着想。

为"我"着想一下

是的，人们没有充分地考虑到"我"——一个要听他们夸夸其谈他们怪诞想法的人。如果有个人是第一次撰写论文或准备一次演说，对他的忠告是什么呢？为听众着想！这是最基本的，但我们仍然有严重的自说自话的迹象。那是因为尽管人们确实为"我"——他们的听众着想，但他们为"我"着想得还不够多。

一般的理解是，为你的听众着想就意味着了解他们是谁以及他们知道些什么，以使你们的沟

通更有针对性。然而，很显然，这太笼统，不充分。你应该为“我”想些什么，这里有一个简明、完整的清单：

- 你具体想从“我”这里得到什么？
- 你什么时候想从“我”这里得到它？是否会请求“我”达到这样的目标？
- 为什么我要关心这些？我是否要一直这样关心，甚至只为了聆听？

> **作者注：**本栏目在谈到沟通时，通篇使用了“我”来代表你的听众，不过“我”可大可小。

告诉我你要什么

良好的沟通从知道你想从我这里得到什么开始，不是你大体上想得到什么，对此我也一样地关心。你想从“我这里”得到什么？

这听起来有点无情吗？绝对不是。这种态度其实是开放的、尊重人的和诚实的，很可能你也这么觉得。我们都有希望、梦想和抱负，这些东西在某些时候边喝酒边聊天会比较有意思，谈谈你的，也谈谈我的。但是现在我正在上班，我很忙，因此要切入重点。

你只是想通知我某些事情吗？省省吧！如果你的信息没有特定的目的，那我没必要听。你说让我知道这个很重要？为什么？你想要我基于这些信息采取什么行动？我应该怎样利用这些信息？

我再认真地问你一遍，你想从我这里得到什么？如果你不知道，那当然我也不知道。如果我们都不知道，那我们纯粹是在浪费彼此的时间。

把你想得到的东西放到最前面，放在一开始的几张幻灯片的最前面的几行上。说得清楚一点，不要遮遮掩掩的，使用粗体和“我”的名字，以突出与我有关的东西。我真的想多些关心，但除非我知道我需要去关心些什么，否则我真的无能为力。

你什么时候想要

劳驾你不要问我要一些我无法回答的东西，这对我是无礼的，对你也没有意义。我对创造奇迹要价很高，因此你还是到其他地方去找那些东西吧。不要只是想着你要得到的东西，你也要想一想什么时候你想要得到它，以及这样的时间期限会带来什么后果。

有时候你给的期限很长，在这种情况下，你不要急于发出请求，因为你将不得不在以后再请求一次。如果是 E-mail，我可以直接把无意义的请求删除；但如果是不成熟的会议、规范书、源代码和演说，那可是实实在在占用了我的时间，不用说，对你倾注的所有努力也是个浪费。

有时候你给的期限很短，千万不要掩饰！我不介意别人请我帮忙，但我介意他们把我想得理所当然。当任务比较艰巨时要说清楚，并且当人们真的做到了之后要记得感谢他们！

有时候你给的期限是个谜。如果你不能预先说出你想要得到它的时间，你最好暂时也不要要求别人去做什么。

抓住稍纵即逝的注意力

现在你知道了想从我这里得到的东西，以及你何时想要得到它，那么我们怎样锁定它呢？让我们正视它！人们能专注的时间都很短，我也不例外，你需要抓住我的注意力，直到我既理解了

你想从我这里得到什么，也引起了我足够的关注去帮你。

这里有一些技巧，能抓住并保持住我的注意力，并引起我足够的关注：

- **简明**。提供必要的背景，但必须指出要点，不要让我的思维迷失方向。如果是 E-mail，把所有东西放在最开始的 3 行（以便它们能够在自动预览窗口中显示出来）；如果是会议，准备一个小范围的议程以使会议简短；如果是源代码，让一个函数体在不用滚屏的情况下就能完全显示；如果是演说，一张幻灯片只阐述一个概念，只提供我关心的细节，而不能只是你关心的东西。

> **作者注**：在写 E-mail 的时候，“把所有东西放在最开始的 3 行”的实例是：“Galt 先生，尽管你的观点很不错，不过你最后的几封 E-mail 过于冗长了。请在下周之前把你所要传达的信息缩减在一段以内。”

- **扼要**。直切重点（警惕离题），不要离开议程的范围。只邀请那些你需要的人参加会议，把他们加入 E-mail 的分发列表，这也意味着，不要简单地“全部回复”，而要尽量缩小分发列表。函数代码、白皮书、规范书和演说要条理分明，如果你想让我感觉心潮澎湃、跌宕起伏，它最好物有所值。只需记住你想从我这里得到的东西，并死死盯住就是。
- **简单**。如果我不得不眯着眼看或者要读两遍，并且你的名字里面不带有“运气”或“安定”的意思，那么你的诉求就失败了。每张幻灯片只列 3 项条目，每篇论文只要 5 页，每个函数只实现一个想法，每封 E-mail 只做一个决定，每份规范书只定义一个功能，使用图片和故事，要把我看成是 5 岁的小孩。如果你不能向一个 5 岁的小孩解释清楚，那么你就注定要失败，你也不应该来浪费我的时间。
- **条理清晰**。E-mail 应该读起来像新闻稿一样——首先是一个概要和悬念，接着才是必要的细节。长一点的文章应该讲一个故事，或者遵循某个其他的常用模式（可以借鉴其他人做得不错的例子）。会议应该有个议程，演说应该先讲概要，接着讲具体内容（一系列紧凑的幻灯片），最后再总结。觉得很乏味吗？你先那样做了再说！然后再增加一些尖锐的话语和一些有趣的图片，你还要充满热情并且适时地开一些玩笑，但务必要条理清晰。我们用这样的叙述模式肯定有它的道理——它们真的很管用！
- **谦恭**。不要问一些只需在网上搜索一下就能知道答案的浅显问题。事先预计一下可能的反对声音和质疑，并且在它们被提出来之前做出回应。不要不懂装懂，特别是在一些法律和专利问题上，如果你不知道，就说“我不知道。”当你直接跟客户、竞争对手、行政人员和有点感情用事的人沟通时，务必谨言慎行；当你演说时，不要所有的时间都只是你一个人在讲，一定要留出时间来给大家提问。不要照读你的幻灯片上的内容，也不必告诉某人他问了个不错的问题——我自己懂，我也知道我的问题还不赖！
- **流畅**。使用恰当的语法和用词，请其他人审核一下你的 E-mail，尤其是在内容比较敏感，或者你本人比较感情用事的情况下。使用清晰的变量名和通用的术语。预先练习一下你的演说——那只是个体力活，在真正开始演说之前，用一些方法让自己放松下来，没有哪个演说大厅没有盥洗室的——用它来感受万籁的寂静，然后再把注意力集中到你想从我这里

得到的东西上，并想方设法说服我。在你演说的最后，以“谢谢，有问题吗?”的方式给出清晰的结束信号，然后享受你绝对应该得到的掌声。

- **以我为主**。沟通不是表达与你自己相关的事，你要知道的东西你自己早就已经知道。沟通的内容应该与我有关——你的听众。把你的信息放在我顾虑和关注的事情上，如果我要求数据，你就提供给我实际的东西，不必讲关于你奶奶的故事；如果我对你的想法“感觉很糟糕”，那就跳过数据展示，直接给我看一个演示，并且让你奶奶来打消我的顾虑；如果我的上司不满意，或者你说的东西不适合于我们的流程，那么你要去说服我的上司或者改变我们的流程，而不要盯着我，浪费我的时间。记住，不同的人关注的事情是不一样的，说服他们的方式也是不一样的。关于这方面更多的建议，请阅读本章前面的一个栏目“寓利于乐，控制你的上司”。

我们做到了吗

沟通是一个很大的论题，而我这里谈到的都是一些基本的规则。你要多花点时间，锻炼你的沟通技能，并且最后做到应用自如。微软到处都是像你一样的聪明人，为了让你真正地有别于他人，你一定要学会有效沟通。谢谢，还有问题吗?

> **作者注**：感谢 Jim Blinn 给出的建议和他在“Things I Hope Not to See or Hear at SIGGRAPH”这篇文章中写的结束语。（你可以在 http：//www. siggraph. org/s98/cfp/speakers/blinn. html 看到一段摘要。）

2007 年 3 月 1 日：“不是公开与诚实那么简单”

我们公司的价值观已然有些势微。我不能否定这些价值观，对正直、诚实、热情、公开、尊重、勇于挑战、自我提升及责任作个准确的评判是很难的。这些都是好东西，这一定没错。

但公司的这些价值观适合微软吗？打个比方，我们与一个关键的依赖方（dependency）有项合作，我们称这个依赖方为“老赖”，他告诉我说一个月前他的团队取消了一项关键功能，因为他们的优先级排序跟我们的不一样。这是否说他们这么“诚实”是对的呢？是否他们的分诊会议很“公开”，我就会有所慰藉呢?

不，“老赖”是我见到过最讨厌的东西。我不在乎他的团队不讲信用是否是真的，他们在公共场合做出一些让人崩溃的决定也没让我觉得怎么样，问题是现在我们的团队已经沉陷泥沼不能呼吸，也没有多余的时间来补救。

借口就是借口

人们总是拿微软的价值观来推脱他们的罪责。老赖说：“我知道我们已经同意推出这项功能，我也尊重你对此的热情，这是个富有挑战性的决策，但是我们的期限有要求，必须削减掉一些功能，我现在对你是够诚实的了。是的，我应该早就告诉你这些，但是分诊会议是公开的，我想沟通对我们都有好处。”

现在，你是否感觉好多了呢？他们只是罗列了一些微软的价值观，得了吧，大家正因为没有进展而饱受指责呢，问题出在哪？

你可能会说，因为责任感不强，但是他们严格遵守期限日期及优先次序，其他方面的责任感也有过之而无不及。

这个问题要复杂得多，并且对我们如何进行软件开发、跨团队合作及业务的开展产生了严重的影响。首先从价值观“正直与诚实”开始谈起。

> **作者注：**在第2章的“有我呢”有关于依赖方处理更有效的方法。

我将坦诚相待

人们认为“正直”与“诚实”是同义词，但它们并不完全相同。诚实意味着不说谎，而正直的意思就是言行一致。

> **作者注：**后来我看到了一句关于正直与诚实的非常绝妙的评语：“诚实就是言必行，正直则是行必果”
> ——Stephen R Covey

你可以做到诚实，但却做不到正直（“是的，我是那个背后戳你脊梁骨的人。”）你可以不诚实但却很正直（“我知道在处理Bug这件事上延期了，我知道这意味着什么。”）以我个人来说，我尊重诚实，但我更看重正直。你可以相信某个人是诚实的，但你更应该将责任托付给一个正直的人。

老赖与他的团队是诚实的，但却缺乏正直。他们答应开发这项功能，但却没有善始善终。当他们不得不放弃这项功能时，老赖却不会跟我们沟通，他没有给我一个备选的解决方案，或是采取措施补救，连一个道歉也没有，他完全不把自己说的话当回事。

没那么简单

诚实相对于正直来说要容易得多了，没人可以指责你的诚实。正直需要勇气与坚持，你必须冒着触怒他人的风险，包括你的上司及你的同事，为的就是坚持你的信念。对你的原则进行妥协要容易得多（假如你有），这就是为什么正直很重要的原因。

要获得大家的赞赏并不容易，但是当你在极端困难的情况下仍能证明你的正直，那就不一样了。人们可能并不赞同你，但是他们会知道你是个有性格有信念的人，一个不能被轻易收买或操控的人，一个值得尊敬的人。

将与他人的合作关系及业务开展寄希望于诚实还是远远不够的，我们还必须正直。可以在价格、计划或细节上讨价还价，但原则不可妥协。如果你放出了话，你就要有始有终，如果你不能说到做到，那就要道歉，然后及时补救。

他们看起来好似开诚布公

但是正直还不足以使跨团队合作无恙，你还要透明，“透明”甚至没有列在公司价值观里。

人们认为公开与透明是一样的，但是并不是这样的。公开意味着你是公众人物，你没什么秘

密；透明的意思是你要把你做的决定与行动的人物、措施、事件、时间及原因讲清楚。

你可以公开但不透明，（“我不知道我们为什么做这个决定，但是我们的会议是公开的。”）你也可以不公开但却可以做到透明，（“协商是关着门进行的，但这是我们一致的决定，你可以了解一下我们为什么这么做以及谁与此相关。”）以我个人看来，相对于公开我更看重透明，你可以与一个公开的人接触，但你可以托付的是一个透明的人。

老赖及他的团队是公开的但并不透明，他们并没有把如何或为什么去除我们需要的软件功能的原因告诉我们。事实上，他们在做出这些决定一个月之前从没告诉过我们这件事，如果一开始我们就知道了这个决定，我们至少可以讨论一下应对之策并对我们的计划做出相应调整。

> **作者注：**当然，在做决定之前先咨询一下你的合作伙伴，或者当时就只是告知一下合作伙伴你的这个决定，这样说比做容易。你必须先准备好合作伙伴的清单，并有一套通知他们的流程，这样不至于引起混乱或停滞。在第3章的“迟到的规范书：生活现实或先天不足”中我推荐过SCR处理方式。

无处遁形

公开比透明容易。在公共场合展现自己要安全得多，因为社会公德可以为你作保护。做到透明则会暴露你的弱点，你不能做作，你必须坦承你的真实情况及厌恶风险的心态。表里不一很容易，“你所要做的就是向别人提出问题”，当你做到透明时，所有人都知道你的实际情况，这就是透明很重要的原因。

当你身陷危机时仍能做到透明是很不易的，但能获得人们的信任。他们可能现在对你不满，但是他们再不会对你有怀疑。你变得众所周知，为人所谅解，最终取得他人的信赖。

或许公开有助沟通，但并不是没它不行，透明则使我们的团队及合作伙伴将所需的内容明白地写在协议上。将你的计划及状况公之于众，与你的合作伙伴分享你的Bug、标准及创建结果，共享进步与成功的欢乐，开诚布公地谈论失败及改进的方法，给那些与你一同工作的人一个相信你的理由。

> **作者注：**最近，围绕着将大家的职业阶段公布在E-mail地址簿上的话题，我们展开了非常有趣的讨论。持积极态度的一方认为，升职、职位模式以及谁是你成绩的评判参照已经变得很公开了；消极的一方认为，这等于暴露等级门次，新手的意见可能不受尊重，而高级别的人会享受特殊待遇。这很难一概而论，但是所有的机构都倾向于透明，这样至少可以通过公开这种等级门次，使不恰当的言行举止得以正确处理。

这不是我的本意

你可能会说：“确实，正直与透明很重要，但是诚实与公开一样重要。你这样小题大做了吧？”不要说得太绝对，记住，我说过问题很复杂，这对我们如何进行软件开发、跨团队合作及业务的开展产生了严重的影响。

当人们只将诚实与公开作为标准时，事情就坏了。我从来不会因为坚持我的正直或透明而后悔过，诚实与公开会惹来麻烦，即使出发点是好的。

诚实可能会显得很鲁莽无情，但更重要的是它可能是种虚伪与无能。因为人们往往不知道真相，而只相信他们所相信的，人们会信誓旦旦却完全误入歧途，这会带来误解，使人感到徒劳无功，并且产生仇恨。

公开同样有这样的问题。你的一言一行决定于你要面对的人是谁，我与家人或我的团队相处时与我在公共场合聚会或聊天时大不相同，我并不是要掩饰什么，我没这么想过，这是因为我的家人及团队跟普通大众不一样。这就是为什么高效的协商必须在私下进行，因为亲情与亲密关系可以让人坦诚相待，毫无顾忌。

以正视听

我对正直与透明的推崇并不意味着我反对诚实与公开，这些都是我们应该推崇的价值观，特别是，公开是一种海纳百川的大度，这样的公开越多，世界将越美好。

但是诚实与公开会是我们成功所需要素的一种短板，而且有时会引起意想不到的麻烦。正直要求我们言必行，行必果，言则守诺，行则坚持；我们对决策及项目需要做到透明，这使得我们在执行与合作的过程中明白我们所处的位置。

千里之堤，毁于蚁穴。在微软，还有人因为透明担惊受怕或缺乏勇气。他们或许该好好想想，要么改变下自己，要么改变下工作的地方。为了微软能成为一个继往开来的伟大公司，我们的价值观不能由一纸繁文来说明。

2009年3月1日："我听着呢"

现在是微软年中职业讨论的时间。或许你已经收工了，但很可能你还在折腾。应该干成什么样子？会干成什么样子？为你自己，还是为你的上司忙活？少安毋躁，要视情况而定。

这跟你或你上司前期工作的进展有点关系，跟反馈意见及如何反馈有点关系，跟你的父母如何把你带大以及你座椅的舒适度有点关系，但是对你的年中职业讨论产生最大影响的还是你及你的上司对反馈意见的回应方式。

且听我娓娓道来。如果你真的不知道如何反馈及如何回应反馈，说真的，一点主意也没有。是我错了吗？你只是在证明我是对的。如果你确实知道怎么反馈及如何回应反馈，那你的回应应该是一句恭敬而有礼貌的"谢谢"。

谢谢你的建议

事实上，对反馈的回应只有两种方式："谢谢"及"继续"。

一句"谢谢"是很简单的，不言而喻的，但糟糕的是大多数人并不会说这个词。大多数人总是为自己辩护，为自己的行为及结果找托辞，对他们是如何做出正确的决定的自圆其说。

省省吧，请你慢慢地、小心地闭上嘴，什么也不要想，只需要听，或许你还可以作下笔记，当你明白你应该慷慨大方时，就说声"谢谢"。

并不是出于礼貌才这样说，你说"谢谢"应该是诚心诚意的。如果人们不再有兴趣以一个旁观者的态度发表他们的看法来帮助我们改进的话，你们团队的关系，你们的生活以及你们的产品

与服务将愈发破败不堪。谢天谢地，他们还是愿意这样做的，为让他们能一直如此，真诚地对他们表达谢意是最基本的。

> **作者注：**如果他人的反馈都是自以为是的，或者可能打个官腔敷衍一下却毫无建设性时，你该怎么办呢？就说“谢谢”或“继续”。说“谢谢”是因为这个人告知你他们对现状的感觉与看法——按玩扑克的说法，他们摊牌了；说“继续”是因为这个人可能不想多提他们的想法，也不想再多说些深层次的东西，因为你可能会反驳他们。

请多提意见

除了“谢谢”，对于反馈的有效回应方式就是“继续”，比如：

- “你能讲得详细些吗？”
- “我还不太明白——你能说得具体些吗？”
- “谢谢，你的建议很中肯，我该怎么做才好呢？”

任何鼓励提出清晰且持续的反馈的方式都是合适的方式。

靠边站，伙计，这我内行

质疑反馈或置之不理的做法都是不可取的，这包括：

- **“我正忙着呢？”**那又怎样？你这是在一错再错，你的工作根本没取得多少进展，不然你就不是现在这样子。无论如何，反馈总是好的，而你反唇相讥既不适当也太过自以为是。
- **“我正要……”**这样就可以做得更好吗？不要把理由与借口相混淆，如果你可以做得更好，你就应该做到，没有借口。
- **“我不这么认为。”**这是新闻吗？你不过在听取反馈，只是一种想法，事实是你只认定你自己对现状的看法，这种看法没什么新意。当否定反馈时，你会错过或误解了某些东西——这些都是很珍贵的意见和建议。

记住，你无须遵照他人的建议去做，你所应该做的就是聆听，认真对待每个建议，并对别人的帮助致以感谢。

> **作者注：**执行官评审会特别重要，在这期间你要保持沉默，多做笔记，且只简单地说“谢谢”。在后面的章节“幻灯片”中，你可以对执行官评审会作更多了解。

轮到我了

现在你知道如何对待反馈了，现在该想想如何要求别人给出反馈意见并同时给出你自己的了。当你要求别人给出反馈或要向别人提出反馈意见时，可以提三个基本问题：

- 什么是对的？比如有关你手头正在做的及你已经完成的。
- 哪些地方可以做得更好？
- 还有什么建议吗？

这样的反馈你可以再想几条，但是最简单、最完整的问题就这三个。当你回应他人的意见建议时，就这三个问题恰是你要回答的。

作者注：最好在行动之前或之后就马上反馈，这样做的目的是在收到别人殷切的诉求后马上给予正面肯定的回应，而在需要的时候再回一个要求对方改进的反馈意见。换句话说，找准反馈的时机是相当重要的。

举个例子，你团队里有个人给你发了封很不错的邮件，但是忘了把相关人员名单复制过来，你马上给他回复："你的来信非常棒——简明扼要。"之后不久，在他可能发第二封新邮件之前，你再回复："记得把相关人员名单也附上。"这样的提醒在这个时候很管用。

我们有一套

当你做出了你的反馈后，要将"什么是对的"作为开始，再讲讲还有哪些需要改进，再加些其他的评述，然后再提醒一下哪里需要改善，最后再重复什么是对的。这个次序很重要。

- 从什么是你认可的开始。你们的对话要建立在一种积极欢快的氛围之中，以防人家失去兴趣。如果你开始就讲哪里有问题，你的听众可能再没心思听哪里是正确的。
- 接下来，你要讲改进的方法。理想情况下，你的听众可能只对其中一种方法感兴趣，大多数人一次只能接受一种方法，选一个最有效的方法，重点讲一下。
- 当然，你有其他很多不那么重要的想法，可以在提及"另外想说的是"时，随意提出来。
- 回到你的中心意思上来——有一种，可能也有两种改进方法是非常有效的。
- 最后以取得了哪些成绩作为结尾。以积极的评述为结束是很重要的。

作者注：一定记住要把重点放在方法或成果上面，而不是某个人身上。人是不能变的，但是他们可以改善他们的方法及成果。

我的时间不多

总之，简明是很重要的，如果你想让你的反馈引起重视，那就应该条理清晰、易于理解、有价值、简洁且以受者为中心。你的反馈跟你自身或你的才识无关，能给受者带来帮助才是要义。

如果在这样的互动反馈中，你是最后一个接受者，那你应该就做到了简明了。反馈意见弥足珍贵，不管它是来自客户、同事或你的上司，不要拒绝它，要鼓励、欣赏加感谢。谢谢。

2009 年 7 月 1 日："幻灯片"

现在接近本财年尾声了。大多数工程师把这个时候视同于绩效评审时间，但是对于首席或更高级别的工程师来说，现在同时是执行官评审会时间。这个时候要费上几个星期制作一些有关执行力的幻灯片，而且这些幻灯片在没有派上用场之前要修改 5 次。

执行官评审会不是在浪费时间——有些时候你需要一个有经验的权威人士来打消你那些想当然的想法，并把你的精力重新放在出工作成果上面。准备幻灯片不是在浪费时间——被迫向别人作解释总是有助于你加强思考，我可不想在一个赞成我的薪酬的人面前显得像个白痴。真正浪费时间的是把精力放在每个季度的幻灯片上，而不是好好想个应对之策上。

聪明、高级别的人仅仅只是不知道如何对待执行官评审会，他们把这样的时间当成个人秀的时间而不是诚心聆听的时间，面对执行官对他们差劲的演说和幻灯片的批评，他们的回应是很不恰当的。我也曾经这么做过——我们的前辈做了个糟糕的模板，而他们又是在他们的前辈们的授意下完成的，乌龙就是这样形成的，以讹传讹。好了，现在该打破这种恶性循环了，弥补这种缺陷，把精力放到重点上去——对于你简明扼要的计划有价值的反馈意见。

明察秋毫

为什么还有那么多脑子灵光的人把执行官评审会搞得一团糟？我觉得有两方面原因——细节与混淆。

- **细节决定成败。**我们必须顾及所有细节而不仅仅只是关注重点，搞糊涂了吧？重点是由谁定义的？执行官。对于执行官来说什么是重要的？问他，一直问到明白为止。不要吸一个助手的二手烟，这对你的健康没好处。
- **将执行官评审与个人述职搞混了。**这二者都使用幻灯片演示，不同的是个人述职时述职者是主宰，而在执行官评审会中，执行官及他们的随同是主宰。没有正确认识这二者的不同将使你的评审失败。

成功的秘密

怎样才能使你的执行官评审会获得成功，并从这些评审会得到所有可能的好处？

有三个步骤：

1. 要了解对于你的执行官什么是重要的？
2. 将这些重要的东西用 3 张以下的幻灯片展示出来。
3. 对所提问题给予精辟的回答，并致以感谢。

就这样。让我们一步一步仔细说明一下。

谜中谜

首先，你必须知道执行官看重什么，这要从项目信息与价值理念两个角度了解。

从信息的角度看，执行官想了解有关你项目的什么信息？你的项目跟其他项目比怎么样？财务贡献度怎样？对市场占有率的影响如何？价值主张是什么？竞争对手的回应会是什么？你必须了解你工作计划的分量。

从价值理念的角度看，哪些原则对于你的执行官来说是最重要的？透明度？服从？忠诚？正直？自信？你必须了解你的分量是多少。

你怎么知道你的执行官对项目信息与价值理念的看法？问问你的同事，谁经历过执行官的评审，问问你的上司及上上级的上司，你甚至可以试图占用执行官 30 分钟的时间。一定不要独信某人，要多渠道听取反馈意见。

> **作者注：**能懂得执行官的心思则更好。但是我们总是希望情绪不要影响执行官的决策，不过执行官也是人，情绪调整还是会带来些影响的。

1，2，3，很简单

现在你可以创建幻灯片了。你只需要三张幻灯片——当前现状、未来期望及实现从第一张向第二张跨越的策略。这就是你的所有计划。记住，评审中你没有控制权——执行官才有。你所能做的就是为这次讨论限定一个范围，所有其他的幻灯片都应该删除，或是作为附录幻灯片以供参考。

> **作者注：**对于更深层次的技术性评审来说，你可能很想做三张以上的幻灯片——要放弃这种想法，把这些更深层次的技术性内容放到附录幻灯片上去。你一定很想把这些附录幻灯片一并给说了，但是执行官是这次讨论的主导而不是你，你只需做好准备应对执行官的问题及意见。

关于现状的幻灯片可以是问题的陈述，对现状的评价，或对目前为止所取得工作进展的概述。什么样的内容及信息取决于什么是你的执行官所看重的，也就是你预先要了解的。

关于未来期望的幻灯片可以是一个解决方案，对期望目标的评价，或是实施策略。这些应该参照第一张幻灯片，解决第一张幻灯片中提出的问题。

策略幻灯片可以是一种时间轴、项目列表或步骤清单，通常还要指出风险所在及相应问题的解决方法。要非常注意幻灯片上的内容，因为这些很可能就是协议（commitment）的内容。

> **作者注：**那张策略幻灯片好似应该涵盖很多内容，当然，这三张幻灯片都应该是这样。你应该弄明白你的执行官喜欢怎样的内容呈现方式。
>
> - ❑ 他是否像喜爱视力检查表一样将所有内容都放在幻灯片上？
> - ❑ 他是否只需要个概要，而希望你将所有细节都放在脑海中？
> - ❑ 他是否更喜欢把细节幻灯片放在附录里以供他参考？

我看过手册指南

很多执行官评审会要求你依据一个事先预置好的幻灯片模板来填充你的内容。有模板确实非常好，它有固定明确的条目。但遗憾的是，大多数的模板非常恐怖，它的幻灯片数量往往是所需数量的4或5倍，而且模板不是太老旧就是出自一个新手之手。

面对有15张幻灯片的模板你该怎么办？挑出3张关键的幻灯片——一张是现状概述，一张是未来期望，一张是第一张向第二张跨越的策略。选准这三张幻灯片——并将其他的幻灯片依次与这三张归类，这样无论执行官从哪张幻灯片开始问话，你仍能抓住重点。可以的话，忽略其他的幻灯片，把重心放在关键的那三张上。

开评审会的目的就是想通过介绍简洁明了的工作计划获取有价值的反馈意见。通过遴选分类你的幻灯片，抓住重点，你可以限定评审会的议题范围，并得到你想要的反馈意见。

> **作者注：**一个小窍门就是在评审会开始三天之前给出一个预览版。用一页纸罗列一下执行官的关键问题，或提供一个参加评审会的人需要事先知道的东西。
>
> 预览版有助于使对话保持住重点，也使执行官的心思放在一个确定的范围内，使他能向你提出明确的问题，这样你的演说就更具目的性。

举止得当

执行官评审会三部曲的最后一步是怎么行动。首要的是，不要屈从于执行官的淫威或受之蛊惑，执行官通常拿起你的工作又放下（不信你可以问问），他们仍旧要洗澡，仍旧要管教他们的子女。为了你自己好，自己搞定这些吧。像你之前所做的那样，行动起来。

在你的评审期间，执行官通常会做两件事——问问题及作评论。大多数时间你应该勤做笔记少说话，如有必要用磁带录下来，阿伯拉罕·林肯说过："不要张嘴，像一个傻瓜一样保持沉默，质疑自除。"

什么时候你应该发话呢？当你有很有见地或至关重要的想法要说时。"我同意"、"我们正这么做"这样的话不要省掉，"当我们修正了这些问题后，在线服务呼叫率下降了67%"，当这样的话非常关键时，或许可以提一下。当你张嘴发话时，一定要确定你说的是有价值的，毕恭毕敬的，并且条理清晰，再来些"是的"，"不"或"我不知道"这样的话就够了，其他时候你只要听。

> **作者注：**如果你需要他人提醒你注意简洁，就让一个参会的朋友在你需要停止发话的时候给你个信号——可以是一束闪光灯信号。

就如我在本章之前"我听着呢"中所说，对所有反馈意见最简单恰当的回应是"谢谢"。而在这种面对执行官反馈意见的情形下，你一定要把评语记下来，对每一条意见都要慎重对待。你不必回答执行官的所有意见，但你有必要重视它们。

记住，执行官评审会不是你个人秀的时间，这是你听取关于你工作计划有价值意见建议的时候，在你按计划发布产品的时候，才是你个人秀的时间。

我该怎么做

当评审会结束的时候，你很可能感觉很沮丧。执行官问了很多苛刻的问题，并作了一堆尖锐的评论，这是世界末日吗？你能说什么呢？

幸运的是，想说评审会有什么效果还不是难事。关键是执行官讨论过的细节的重要程度，而不是他们肯定或否定论断的数量。如果执行官的问题与评论都高瞻远瞩，事关大体，那么这次评审会就不成功，因为执行官只问了些基本策略及前提的问题，如果问题与评论都很具体，那评审会就大获成功了，这样的情况下执行官是赞同你的策略、你的方法的，他们是从细节方面给了你意见建议。

另一方面，执行官评审会并不是为了给你多少信心的，他们的作用是对你的工作计划提出有价值的意见建议。要了解执行官们关心的内容及原则，围绕这些问题你要尽可能简单地表述你的计划，评审会期间要做到很专业，那么你就会得到成功所需的所有意见建议。

> **作者注：**是否执行官评审会对于实施一个项目是必需的呢？很多时候他们扮演的是无组织无计划的救命稻草，当执行官的计划已经出台，相应组织也会应运而生，那么所有执行官应该做的是定期组织召开员工会议及相应的执行官碰面会。如果执行官可以通过博客、一对一会议、吹风会或随意的约谈来与员工们互动的话，这也是很好的，很多微软内部机构现在都是这么干的。

2009年12月1日："不要悲观"

在第9章的最后一篇专栏"管理馊了"中，我提到过管理者们应该如何约束自己不致让他们的员工放任自流。但是，假如你就是受管束的一方会怎么样呢？作为一名员工，一些随机性的请求，或无论什么请求，这些请求跟你当前的任务没有半毛钱的关系，你如何应对？

是的，我们都想成为负责任的小伙或美女却不用回应任何人的请求。种瓜得瓜，种豆得豆——人们都希望成为自己的老板，好像成为一个CEO后你就可以避免来自客户、债主的随意性骚扰。醒醒吧，你是躲不掉的。即使是史蒂夫·鲍尔默的一天也是由这一干人等主宰的。

最要命的是，每个人都觉得他们总是很忙，而其他人都很空闲，杵在那等着你使唤的，这样想真是愚蠢之极。所有人都很忙——非常非常忙，经常是忙得不可开交，不堪重负。所以当有另一项任务请求时，你就不会有多余的时间，但是这样的请求又来自你的上司或你的客户或是你的合作伙伴，他们是不能等闲待之的，你必须做到来者不拒——即使你的脑壳嗡嗡作响，你的邮箱泛滥，你的肚子发疼——你该怎么办？最起码，不要悲观。

你应该说："没问题"

你可以这样回应这些请求："没问题，很乐意为你服务。"口是心非，是不是？因为毕竟，你帮不了什么忙。你没有时间，你的团队成员或你认识的谁都没有，关键是怎么以适当的方式说："没问题。"（之后，我将谈谈在一些特殊情况下你可以说"不"。）

为什么对几乎所有的请求都要说"没问题"呢，而不管你现在的情况如何？因为说"不"于事无补，只会更糟。置若罔闻还不如直接拒绝，最好的方式是"没问题"，可以用不同的方式这样说。

- "没问题，仔细说来听听。"大多数请求是很模糊的，很可能它们毫无内容，也可能它们对你毫无意义。大多数情况下，请求的人根本就没仔细思考过。先给个"没问题，仔细说来听听"只是表示你愿意知道更多细节。当别人的请求说得更具体时，你就找到一个脱身的出口（"哦，就这呀？"）在这个时候，你就可暂时脱身了——有些时候是永远的。

> **作者注：**为什么首先你不问问为什么呢？你可以问为什么，但不是这种方式。"为什么"可能引起别人的反问，而"没问题，仔细说来听听。"就安全得多，也一般会获得你需要的更多的信息。

- "没问题，我给你推荐一个人或网站可以帮上你的忙。"很多请求所问非人，或者找找网上资源就可以解决。适当地将这些请求推诿掉就可解决问题，言简意赅，避免还要回答一次。不要担心你推荐的人会有多忙——他们会像你一样说"没问题"的。

- “没问题，什么时候要给你回复？”弄清楚什么时候要解决这个请求，再安排它的优先等级。如果有充分的时间，你与你的团队可以做任何事；如果有足够的优先等级，你可以马上着手做任何事情。这只是优先等级与时间问题，我们来深入讨论一下。

喧宾夺主

当你问这个请求要在什么时候完成时，回答通常是——“马上！”那该怎么办？这要看是谁的请求。如果这个人对你的日程安排没有决定权，比如这些人不是直接管理者，你就说：“好的，我问问我的顶头上司确认一下什么时候可以完成。”有时候，即使是你的顶头上司急着想要你办事也可以将它推脱掉——如果不行，就该问问你的老板或项目经理了。

在这种情况下，或是你的管理层提出的请求，或是你把别人的请求提交给管理层。换另一种情况，这样的交互是一样的，你要提升这项工作的级别，作为一项当前里程碑或重复任务（iteration）（可以在白板上绘制）将其放入工作时间轴上，你可以问问这项请求应该放到优先等级列表的什么地方，有三种可能：

- 把这项请求放到时间轴下方。你可以回复这个请求，指出它必须至少放到下一个工作周期，如果不行，就把请求者引见给你的上司。这样可以使你及你的团队按原计划工作，并把精力放在最高优先等级的工作上。
- 把这项请求放到时间轴上方。这就意味着最低优先等级的任务项移到了时间轴的下方，在你的管理层做决策的时候你再把它提出来。你要跟受此影响的人沟通，让请求者知道什么时候可以开始这项工作。
- 这项请求放在时间轴的最上方，但是你的上司不想舍弃任何一项任务（听起来似曾相识吧？）你要考虑你现存任务项的规模，可能有些可以简化或削减，可能其他人从你的任务列表里接手几个或由他们来处理新的请求。不要在多方未达成一致意见，大家还不明白这种调整所带来的影响的情况下放任自流。然后马上跟相关人等协商，用书面确定应对之策（这样可以让这种变动确定下来）。

当然，你可能确实想完成这个请求，只要它不影响你的协议，你就可以这么做。不过，不要欺骗自己，如果应承了这个请求要花去你好几个小时的时候，你就要调整你的正式工作日程了。当你的工作遭遇挫折时，没人还会神采飞扬地谈论你所做的工作。

> **作者注：**将提升级别后的待办事项作为里程碑（milestone）或冲刺（sprint）放到时间轴上。如果你使用的是一种类似 Scrum 的方法，这就要很细致。即使是最像样的瀑布式项目管理方法也要使用任务优先等级列表。不过，还是要确定一下你是否使用了这样的列表，如果你的上司也对它很熟悉，那就太好了。

先信任再检验

现在你非常忙，却又接受了一项新任务。可能你会跟你的上司商量让别人来帮你一把，遗憾的是，这帮人都是毛手毛脚的，他们并不想把事情办好，或者说他们并不想按你的方式来做。你该怎么办？

于是你把这个请求的主要部分委托了出去——不是无关紧要的工作条款，是主要部分。你说："这归你了，这是项目需求，这是制约条件，这是相关人员名单，这是我期望的结果及我希望它们在什么时候完成。还有问题吗？"

但问题是别人不会按你想要的方式做事，所以，算了吧。全权委托给他们，让他们自己来找出适合他们的方法，这要比你要求他们采用的方法好得多。这就是如何委托的问题，你要相信别人，这是关键。

自然，你想定期检查一下他们的工作进展及成果，要以信任为先，再则是检查，通常这是委托管理的关键，相信你委托的人，再检验他们的工作。

要知道什么时候说"什么时候"

有些时候你还是要说"不"或对请求不予理睬，但仅在一些特殊的情况下才这样。

- 如果这个请求是向一个大型机构提出的，这样你就可以安心对之置若罔闻了。不过，建立你的人际关系，增加你的影响力的上佳途径就是对这些请求予以回应。看在你头脑还清醒的份上，为了你当前项目的顺利开展，请仔细斟酌你选中的那些请求。
- 如果这个请求来自你的员工，并且没有相应优先等级，你应该说"不"，并利用这次机会重申优先等级的重要性。如果这个团队或团队成员特别热衷于某个请求，提醒他们，他们只要完成他们的协议就可以，不要影响团队的其他人，他们可以自由支配他们的闲暇时间。不过，当他们的工作遭遇挫折时，没人还会神采飞扬地谈论他们所做的工作。

> **作者注：**有时外加入的项目会带来关键性的突破，即使它们没有相应的优先等级。说"只要你达到协议的要求"，意思并不是说"不要做"，而是说"要保证仍然能完成你的协议要求"。这就是闲暇时间很重要的部分原因（另一部分原因是需要时间来思考——千虑必有一得）。要确保你的工作计划是有根有据的，这样就能使闲暇时间得以保证。

- 如果这个请求与公司或部门或团队的策略，或者你的个人原则相悖，你应该说"不"，并趁这个机会重新强调一下这些原则。是谁问的无关紧要——不管是你的老板、总经理或副总裁。这些是一个公司或是你个人所要珍视的原则，如果你没有坚持这些原则，那么你们的工作乃至我们的工作毫无意义。我们可以时常在一个 Bug 或一项功能上妥协，但是如果我们放弃了我们的原则，我们就丢掉了我们的灵魂。

我就丫环命

人生就是索求，要全部满足这些要求是很难的。必须有个轻重缓急，必须做到平衡，在你的门后，你的即时通信软件中，你的墙边，还有你的邮箱里，请求接踵而至，永不停息，但你没法马上处理所有这些请求，重要的是要积极面对，享受这些上天给予的恩赐。

对新来的请求要理解清楚，妥当处置，深思熟虑后再与之前的请求排个轻重次序。如果你能快速、透明并负责任地处理好每个请求，那你就可称为专家了。当你能做到这样还秉持正直——坚守团体及个人的原则，那你可称为受敬重的专家了，请你成为一个受敬重的专家。

2010 年 8 月 1 日："我捅娄子了"

是否总是犯错误？这种错误会使你感觉心中空空——你知道你已经把事情搞糟了？或许你也尝试过努力做好一件事，但是结果往往不经意间就犯了错。这样的事经常在我身上发生，最近也在我的一个朋友身上发生了——我对此感同身受。

更糟的是屋漏偏逢连夜雨。你的压力无以复加，你会不顾一切寻找弥补的办法，你没得睡觉，感觉像犯了罪，忧心忡忡。这样的痛苦会持续好几天，甚或你会变得很另类，受人冷落，因为你就是个渣滓。

亡羊补牢

对于我们这些还有良知，希望亡羊补牢的人来说，我们最急切想要明白的是："我们怎样才可以弥补这样的错误呢？"很高兴你能这么问，以下是你要做的：

1. 负起你的责任——是你犯了错，你就得承认。
2. 深入理解缺漏的原因——不要越搞越糟。
3. 寻求他人帮助改正错误——要知道问题不是在你一个人身上。
4. 确保类似事件不再发生——敞开心扉，跟别人谈谈以后怎样避免此类事件发生。

这时要做到冷静，控制好情绪是很难的。不过，要补救必须先冷静，你捅了大娄子——是没有捷径马上弥补的。让我们来一步一步仔细讲讲。

> **作者注：**无论你说错了什么或做错了什么，已既成事实了，所以不要再想着把这些收回。再看一次邮件只会把心思放到邮件上，掩盖错误只会使事情变得更糟，要成熟些，专业一些——承担起此次错误的责任。

男儿要担当

你已经犯了错，就请再一次证明你是个男人。说"是男人"是理由而不是借口，你确实已经搞糟了，第一件要做的事是承认。

不要指责他人，即使你认为别人犯的错更严重，坦诚一些，指责只会伤害别人，你所要做的就是坦率承认："我错了。"

在工作上犯了错，只说："对不起"，"请原谅"于事无补，在某些敏感的情况下还会引起法律纠纷。为什么说在工作中说"对不起"于事无补？

- 在法律层面上（声明对所有事情负责）你并没有错。你对这个错误表示很遗憾，但你不是有意的。
- 你只是想树立一种你能处理所犯错误的能力的信心——听起来并不像理屈词穷。
- 你只把重点放在以后，以后问题就会解决也不会再次发生——而不对过去铭记在心。

> **作者注：**当然，在人际关系处理中说"对不起"是很重要的，但是，这不是你的私生活，这是工作。真正应该做的是正确认识现在的问题并改正它。

没必要在这问题上啰嗦，直截了当承认："我错了。"接着该干什么干什么。

深刻理解错误

一般人犯了大错后要做的就是急于提出或实施一套解决方案，沉住气，先好好想想，你只有完完全全地理解了这个错误，你才不会再犯这样的错误。别自以为勤能补拙，你必须深刻理解你所犯的错误。

- 谁受此错误影响？
- 是否你的错误以不同的方式影响了不同的人？
- 错误的原因是什么？（人为的、工作计划方面的、资源方面的或其他的什么事情。）
- 人们最想要的解决办法是什么？
- 你如何补救，如果有办法的话？
- 日后如何防止此类事件再次发生？

只有虚心听取那些受此影响的人的意见，多问问题，真切地理解现在发生了什么，你才知道该如何解决。

他山之石，可以攻玉

人犯了错后再犯的第二大错是独自补救错误、没有内疚、狂妄自大或博取同情。一句话"是我错了，后果我会自负。"省省吧，赶快反省一下。没错，错是你引起的，但是问题留给了大家。

当你明白要怎么做才能补救时，要向他人寻求帮助。如果坦诚以待，虚心求教，大家是很乐意帮助你的。一起修复错误有利于重建人际关系，也可以确保所采取的解决方案迎合每个人的需要。

请注意这个名词"人际关系"，就是这么回事儿。当你犯了错，挥之不去的毒害就是在你与合作伙伴及客户之间产生的隔阂，你要竭力维护你们之间的信任关系。

通过寻求帮助，你就不用试图逃避责任或工作了，你仍然是中坚力量并负责问题最终的解决。不过，要想这些最终达到目的——使你的合作伙伴满意，你也必须让他们参与进来，而且是心甘情愿的。

当然，你的合作伙伴提议的解决方案可能你并不喜欢——这正是需要深入理解问题究竟的原因。通过理解分析，可以让你的合作伙伴放心地采用另一个解决方案，同时也有助于你接受他们所希望的解决方案，因为毕竟，你无权支配那些你伤害过的人。

在犯这次错误之前，你们的关系越好，你们一起合作得就会越顺利，当事过境迁之后，你们的关系也会愈加稳定。人际关系决定一切。

> **作者注：**如果你遇到了麻烦，与人事部门或法律部门或者企业综合事务部联系。记住，自己解决问题并不能让你成为英雄或烈士，只会让你显得更白痴。
>
> 如果别人遇到了麻烦，对错误要宽容并帮助你的同事改正，多一个朋友多一条路。

绝不再犯

你承担起了责任，明白了问题根由所在，并努力寻求一种大家都能接受的方案，最后要做的事情就是防止这类问题再次发生。大家都可理解偶然发生的错误，但当再次发生时大家就会怀疑你的真实意图及你与他们之间的关系了。

就如我之前所说，你必须彻底明白如何在今后做到避免此类问题再次发生。所以你必须把你的真实意图解释清楚，只用三个字："向前进"。

> **作者注：**为了彻底明白此类问题如何才能不在今后再次发生，你必须对问题做次刨根问底的分析。在第1章"揭露真相"中我列举了一些例子——即如何使用5个"为什么"分析问题的根本原因。

"向前进，就是在做出最终决定前，我会与我的利害关系者商议。""向前进，就是在对更改进行全面检查之前，我会运行一下全部的自动化测试套件。""向前进，就是在吃最后一块油炸圈之前，我会问问其他人的意见。"

向前进要比向前看好得多。不要指责，不要抱怨，向未来进发，谁都会支持的。

> **作者注：**如我在第3章中所说的，人们很容易一错再错，这就是为什么检查清单会这么有用。逐条看看你的个人习惯，逐条解决，以防止错误再次发生。

一切都会好的

有些时候，人们仅仅是想听到你承认错误，他们会来修正这个问题或者乐于帮助你。他们只是想知道你认识到了你的错误，能正确对待你引起的错误，以及知道如何避免再次发生，迈向前进。

有些时候根本没办法弥补一些严重的破坏，或者这种破坏会被拖延好几年。然而，你仍然可以因承担责任而改善你们的关系，并保证这种错误不再发生。

为什么人们都是这么宽容？因为谁都会犯错误。我们都有过那种搞得一团糟的恐怖经历，我们同舟共济，相信别人会明白道理的，会努力工作弄清楚问题，有一天，这些终会过去。

2011年3月1日："你也不赖"

"我可以跟你谈谈大傻吗？这个人牵动了每个人的神经，他的沟通方式造成很多麻烦，他将整个团队带入了死胡同，他就是个下作鬼。"如果你是名项目经理，之前你很可能就听过这样的话。每个团队都有大傻，你对大傻的看法是怎样的呢？把他调到另一个团队？解雇掉？不，别傻了。

大傻要反思他的所言所行——这一点毋庸置疑。如果你是大傻的上司，你得仔细掂量掂量这个情况，采取适当的行动。

但是毫无疑问，问题不只在于大傻，问题在于整个团队，让我给你搞搞清楚。你就是个大傻，我们都是，没有谁是完美的，没有谁话无错行无过，即使你这样做了，总有人会不待见你。

> **作者注：**大傻就是我小时候抑或恐龙主宰世界的时代，那个主宰儿童乐园的小丑。

好的，坏的，丑陋的

"没错，不能对大傻们一概而论，可我团队的大傻就是毫无用处——简直是灾难。他必须滚蛋。"真的吗？那为什么雇用他？他的评审记录怎样？他是不是一开始就是个大傻呢？

有时候我们确实做了个错误的雇用决定，应该把这个小丑送到另一个马戏团去，我将这种制造麻烦的团队成员称为“负面人员”。如果你把这个大傻从他的团队弄走却没人来顶替他，那他的团队在缺一个人的情况下长期运转会怎么样？如果你相信工作效率会更高而且一直这么高，是因为大傻带给大家的只是更多的工作量，那么大傻就是“负面人员”——他对团队生产率来说是减而不是加。

然而，大多数情况下，大傻虽做错了事但却是个好员工。遗憾的是，他的同事与他相处时没有足够的耐心、宽容心或亲和力。

《微软团队成功秘诀》（Dynamics of Software Development）是关于这个主题中我最喜欢的一本书，这本书是由前微软员工吉姆·麦卡锡写的。其中“不要对大傻吹毛求疵”一文强调宽以待人的重要性。关键是要明白问题的根本原因，并由这个“大傻”与团队一起解决问题。

爱我，就爱我的全部

记住，每个人都是大傻，人非圣贤，我们不能见利就伸手，见弊就驱之不及。你可以不认同，但后果自负，或者你就虚心接受，接受人的天性。

为什么人们总对别人“吹毛求疵”而不接受这个事实呢？因为将你的想象附加于别人身上比之于正视他人的独立人格要显得简单而自然。你相信你的同事，你的爱人，或一位名人总是很完美，那你就会认为他们很完美，就是这样。直到她让你失望，然后就开始吹毛求疵，她是如此不堪入目。

不要再想象，对着镜子照照自己，你并不完美，别人也是。你充其量也就是一个普通人——一个既有优点又有缺点的人，别人也是。接受现实，宽以待人，该干什么干什么。

我会宽待你的

你如何对待生活中的不完美？宽容。

当然，你应该坦率而谦恭地与你的同事一对一地谈谈他们的缺点。相应地，他们会努力改正或至少有助于你们适应彼此。记住，你不能改变所有人的所有问题，无论你多用心，相反，正视自己也正视你的同事——相互适应。

我有过古怪又富创造力的同事，有过睿智却自大的同事，也有过精力旺盛但却狂暴易怒的同事，我是可以规劝那些异类但这样却会让他们失去他们的创造性，我可以威逼那些自恋狂但这样却会让他们失去他们的睿智，我也可以收敛他们的暴躁脾气但这样却会让他们失去他们旺盛的工作能力。相反，我给予这些富有创造力的异类以规矩，给予这些睿智的自恋狂以耐心与指导，再给那些精力旺盛却易怒的家伙以舒心的环境来工作或发泄。

我总是很健忘，所以我就自己给自己发 E-mail，或要求别人给我发 E-mail。每个人都有其工作与处理自己缺点的方式，问问并学学哪些方式对你的同事或你的管理层很奏效，然后再反省反省，虚心接受。

作者注：作为一名管理者，最重要的是要认识到什么时候让员工换个地方找到最好的归宿。有些时候你们团队来了个好小伙但却完全不适合团队，这种情况不常见，但也有。他的到来不是百舸争流，相反是风格相互冲突，找出问题的根本原因后，要鼓励你的员工换个新岗位。要帮你的员工寻找并使之享受一个更开心更灿烂的未来。

我们发挥每人的特长

为什么有的人会妥协并接受你的缺点？那就是说只要可能，你就会改正自身的问题。结果会怎样？你只会面面俱到，事事平庸，并没有充分发挥一己所长。你一定可以做到与每个人都其乐融融，这很好。但是，这不是你被聘用来这个岗位的理由，用迁就来抚平你的缺陷？绝对可以！想一边盯着你的缺陷一边发挥你的特长吗？想也别想！

你之所以为一个独立个体正因为你一己之特长，这些特长正是你该开发并发挥至极致的。总之，你的缺点抑制了你特长的发挥，你必须改正或防止这些缺点，但是，记住你是谁，你成功的根本是什么。换地方，干你最擅长的，让自己快速成长。

宽以待人

很多时候，每个人无论在工作中或是生活中都被别人认为是大傻，好好反省一下，接受自己的缺点，为别人做你可以做的事，然后容忍一下其他的问题。帮你的同事，你的朋友，你的家人，让他们也这样做。

诚心诚意地跟你的同事一对一面谈他们的短处是不够的，要宽容。对他们的缺点多些耐性，发挥他们的长处，则他们也会同样对你。是的，你可以争辩或抱怨，有时候可以把其当做一种发泄，但很少有多少用处。

放下你的架子成为其中一分子，对你，对你的团队，对你所爱的人都有好处。但总有你不可妥协的地方，但通常不多，如果你将他人以常人来对待，那你的压力将小好多，你们的关系将更融洽，而你也将过得更开心，更成功。

第9章

成为管理者，而不是邪恶的化身

本章内容：

2003年2月1日："不仅仅是数字——生产力"
2004年9月1日："面试流程之外"
2004年11月1日："最难做的工作——绩效不佳者"
2005年9月1日："随波逐流——人才的保持和流动"
2005年12月1日："管理我在行"
2006年5月1日："比较的恶果——病态团队"
2008年3月1日："必须改变：掌控改变"
2009年6月1日："奖赏，很难"
2009年10月1日："招人总是后悔"
2009年11月1日："管理馊了"
2010年1月1日："一对一与多对多"
2010年7月1日："文化冲击"

微软文化中有这么个说法："不要抱怨任何事情，除非你已经为它找到了建设性的改进意见。"没错，我常常抱怨我们的管理层，更糟糕的是，我在这个公司的12年时间中有8年是在做管理者。那我对管理的改进有什么建设性的意见呢？你能这么问是挺有趣的。

如今，新任的管理者可以得到很多帮助，但是当我第一次成为一名主管的时候，我与以前的管理者共事的经历成为了我唯一的资本。我做得还行，不过我也从工作中和我的导师那里学到了大量的东西。5年之后，我加入了现在的这个组织，给新任主管和经理提供我曾得到过的有效而具体的帮助，成了我工作中头等重要的事。我写的关于管理的文章中，很多都直接来自于我为新任管理者准备的资料。

在这一章中，我将论述如何进行有效的管理，而不会让你成为恶魔。第一个栏目教导管理者要合理使用度量，并指出一名卓越工程师所具有的特征；第二个栏目谈论面试和招聘；第三个栏目带你认识低效管理的关键性问题；第四个栏目涉及了人才的保持和流动；第五个栏目揭示了成为一名优秀管理者的最低要求，以及如何才能从优秀上升为卓越；第六个栏目揭示了如何从一个失效团队转变为健康高效团队的秘密；第七个栏目一步一步指导如何改善管理；第八个栏目指出了奖赏员工的关键之处；第九个栏目揭示了为什么想要招个完美的人会失去一位出色的应聘者；第十个栏目对漫无目的的管理者提出了批评；第十一个栏目讨论了一对一面谈及

集体活动的作用，以及如何使他们变得更有效率；最后一个栏目告诉管理者如何改变他们团队的文化。

坦率地说，我从来就没想要成为一名管理者。我已经做了 17 年成功的个体贡献者和架构师。我最后成为管理者，是因为我对产品有一些自己的想法，我想了解如何去运营一项业务。令我惊讶的是，我发现管人比编程更加让我着迷和满足。也许其中的部分原因是我当上了爸爸——养育一群孩子跟发展一个团队在很多方面是相似的。但更主要还是因为，跟人相比电脑是可预测的、死板的。哈，当然，电脑能够给你带来惊喜和快乐，但那还达不到人所能给你的。我仍然热爱编程，不过我发现跟人打交道要更加有益得多。

2003 年 2 月 1 日："不仅仅是数字——生产力"

只有我是这样吗？还是外星人接管了我们所有管理者的大脑？让他们深信，数字能够精确地代表一个产品的质量或者一个开发者的价值？我们到底还要忍受多少令人作呕的数据收集、分析和预测，才能让那些被误导的管理者得到足够的满足，以使他们不再来打扰我们，让我们能够做自己的工作？如果我们能够在编码上面集中精力的话，我们其实能够完成比现在多得多的工作，难道只有我一个人内心深处有这种感觉吗？

不过，目前的趋势是，我们对创作的东西及其创作方式做越来越多的"度量"，并且使用这些度量去判断我们产品的优点——还有人的优点，似乎解放我们的大脑是有益的。你是否意识到，有多少自作聪明的管理者很轻易地就对他们最正确的判断表示怀疑，而靠"编码成功计量"给团队成员评分？

> **作者注：**《Interface》的编辑要求我写一篇关于度量的文章。我不认为这就是当初他们脑子里想要的东西，但最后期限摆在那里，我也交出了他们规定字数内的文字。

要小心你所想要的

《Interface》上的文章"度量开发者的生产力"就使用度量的缺陷对多种用于度量开发者的计量分析方法进行了对比。这些缺陷中最主要的是，度量很容易被当做"游戏"，管理者基本上总会得到他们所要求的东西。如果写更多行的代码能够让度量结果更好，那他们得到的代码行会更多；如果更少的代码签入能够让度量结果更好，那他们看到的代码签入会更少。并不是因为代码变好了或者开发者变好了，而只是因为那样做可以使度量的结果更好看。

只要人们相信他们是用数字来判断的，他们就会采取任何必要的措施确保他们的数字很好看，而不会去管那个度量系统的初衷。毕竟，为什么不按常规去做正确的事呢，如果这样的事不能提高你的身价？

那是不是所有的度量都没用呢？不是，只要度量用于跟踪团队和产品的进度，以便达到明确而公认的目标（比如绩效、回归率、发现和修复率、解决客户问题所用的天数等），那它们就很有价值。度量有助于推动团队前进，并且赋予成就客观的内涵，然而，正确的判断绝对不应由它们来做到。

> **作者注：**我后来对建设性的计量方法有了更多的认识。你应该用它们度量想要的结果——在理想情况下，应该是基于团队的结果。结果应注重“什么”，而不是“如何”，这样就给大家的提升以更高的自由度。团队度量为的是共同的目标，而不是恶性竞争。“代码行数”不大可能是人们想要的结果，而“用最少的时间开发出高质量的功能”才是想要的结果，而且这个结果取决于整个团队，而不是个人。

不要用一个数字来说明个体开发者的生产力，管理者更应该回答这样的问题，“是什么成就了一位优秀的程序员？”如果答案跟指定一个数字那么简单，可能很久以前我们就都被定理证明的机器取代了。

各司其职

度量开发者的价值的关键是针对开发团队的，而不是个人。每个人在团队都各有所长，在判定各个开发者的贡献时，要考虑到他们在团队里扮演的角色，最好的开发者往往不是那些代码写得最好或最快的人。

你不希望整个团队都是主管或都是代码工人，也不希望整个团队都是架构师。每个团队都需要有一个人才上的平衡，这样才会有最好的效力和生产力。

优秀开发人员的特质

但是，对于“度量开发人员”这样的话题有很多说不尽道不明。下面，我将列出我认为优秀开发人员应具备的一些特质：

- **他们知道他们正在做什么。**当你问优秀开发者，为什么那里有特定的某行代码或某个变量，他们都能说出理由。有时候他们给的理由不那么得体（“那行代码是我从其他地方抄过来的，它使用了那个结构”），但他们的理由永远不会是，“哈，我不知道，它看起来能够工作。”
- **他们不相信魔法。**这是他们知道在做什么的必然结果。对于黑盒的API、组件或算法，优秀的开发者会感觉不舒服。他们想要知道那些代码是怎样工作的，以免被错误的假设或“有漏洞的”抽象所害（比如一个字符串类，它在实现简单的字符串连接时，隐藏了内存分配故障或所需的 $O(n^2)$ 运行时间）。

> **作者注：**赏你一个我最喜欢的一篇专栏“Joel on Software”，“The Law of Leaky Abstractions”。请访问 http://www.joelonsoftware.com/articles/LeakyAbstractions.htm。

- **他们了解客户和业务。**我在第7章的一篇专栏“人生是不公平的——考核曲线”中详细讨论了这一点。优秀开发者知道事情的实质，他们能够分出轻重缓急并且做出恰当的权衡。
- **他们把客户和团队放在优先于自己的位置。**优秀开发者不会亵渎任何任务，他们认为没有哪个客户是不重要的。
- **他们有不妥协的精神和道德规范。**尽管个人喜好可能会改变，优秀开发者会一如既往地很在意他们如何完成工作，以及怎样去跟其他人交流。不管是他们选择的算法还是他们写的E-mail，他们都会对自己高标准、严要求，始终不会动摇他们的核心价值观。

- **他们有杰出的人际沟通技能。**尽管没有多少开发者能够做一个好的游戏主持，但优秀开发者能够很好地跟别人相处，尊重别人，并且跟人进行清晰、有效和恰当的沟通。他们不会选择欺凌或胁迫（尽管他们可以那么做），而是选择合作。（关于这一点，请同时参考第8章的专栏“合作还是分道扬镳——协商”。）
- **他们有一个广泛的支持网络。**优秀开发者能够认识到别人身上的优秀之处，并且他们相互欣赏。他们能快速发展起一个相互支持的人际网络，使得他们比单一个体要有效力得多。

关于优秀开发者的一般性特质，我还能列出更多：注重质量、善于助人、表现出非凡的设计技能，等等。

这些定义优秀工程师的特质中，没有哪个是可以被很容易地度量的。

你是判官

在迫不得已的时候，作为管理者的你必须尽可能公平地评判你的团队，并且考虑到他们每一个人所扮演的角色。从其他团队拿些例子过来参考，会有助于你正确看待自己团队的开发者；这也是为什么说绩效评审会议如此有价值、有启发性并且痛苦也是值得的原因了。

但是请记住，没有哪项绩效指标或等级可以代表一个人。人实在是太复杂了，即使是最客观的度量也会被主观看法所曲解。作为高等动物的人类，正确理解评价一个人是解放人类所有潜能的关键！

2004年9月1日：“面试流程之外”

招聘像一个巨大的吸尘器，它把我所有的时间都吸到了校园。但在快速聘用到一位高质量的应聘者之后，我就会觉得曾经花费的每一分钟都是值得的。所幸的是，在大部分的招聘过程中我不依赖于任何人，这让大量应聘者向我“滚滚涌来”。不过到了面试流程，情况就不一样了。

嘿，如果我能够跳过面试流程的话，我真想那么做！再不用为日程安排争吵，不用一拖就是几个星期，没有可怕的面试问题，不再杳无音信，不用糊里糊涂地就聘用了，没有绕来绕去的垃圾标准，没有接踵而来的负担，不用在最后时刻选择放弃。

但是，你不能略过面试流程——绝对不允许。如果比尔·盖茨想要到我的团队来任职，我也要让他过下面试流程。我不是想冒犯他！只是想让他走过这个流程，确信他是否能够成功地融入到团队中去——如果他得不到聘用，那我也会打起精神来。

怨天尤人

我说过，在面试流程之前我不依赖于任何人。有些要招人的经理们对此提出了质疑，他们可能会说，“那我的招聘专员呢？招聘专员没给我安排时间，他已经几个月没有给我发送任何简历了，招聘专员是我的瓶颈。”嘿，如果在招聘中我是你竞争对手的话，我会非常高兴的。你这个懒惰、脑子不健全的傻瓜，我会悄悄地抢走你所有优秀的应聘者。

招聘专员是你的合作伙伴、朋友和资源——不是你的佣人。招聘专员找的职位太多太多，除非你的副总裁认为你那空缺的职位很紧急，招聘专员是不可能做你托付的所有工作的。因此，不

要妄想了，主动到招聘专员的办公室去，然后翻看他们手上所有的简历。要不然，你就等我把我这所有的空缺职位都补上之后再说吧。

> **作者注：**如今，所有的简历当然都是在线的了（那时真的是相当地落后）。不过，有一点还是对的，那就是如果你不去找到你需要的人，那他们就会找到其他的工作。

90%是准备

接下去，我们回到面试流程这个话题上。（关于招聘的总体看法，我打算在未来的某个专栏中再讨论。）很多想要招人的管理者误解了面试流程，不过，我马上就会为你拨乱反正。一个成功的面试流程，其中90%是做准备工作，剩下的取决于你的AA面试官。准备工作只要3步：

- 帮面试官做准备。
- 帮招聘专员做准备。
- 再次帮面试官做准备。

任何有资格去面试别人的人，都应该预先做如下的准备：

- 参加面试官培训。
- 精心设计面试问题。

如果不参加面试培训，初级人员就无法学会适当的面试方法，而高级人员也意识不到他们已经养成的坏习惯。如果没有精心设计的面试问题，你也可能把Jason Voorhees招聘进来。

问题

哈，面试问题！怎样才算是一个好的面试问题呢？根据我多年的面试经验，我得出了一个结论，那就是只有下面两种类型的面试问题才是有价值的：

- 能够暴露应聘者性格特征的问题。
- 能够展示应聘者将如何开展工作的问题。

脑筋急转弯是没有意义的，关于个人背景的问题也很无聊，只是问“你将如何做这个工作”这样的问题更是没有大脑。这些类型的问题很少会直接暴露应聘者的性格特征，或者真实地展示他们的绩效。也就是说，除非你对这些问题追问，“为什么？”否则你将达不到目的。不是一次或两次，而是要重复地问“为什么”，直到你了解到应聘者关键的性格特征或绩效水平。

更好的做法是，从一开始就问“为什么”这样的问题。比如说，“为什么你想要得到这份工作？为什么你要抛弃以前的那份工作？为什么你仍然在微软工作？为什么你想要为微软工作？”然后再问一遍。如果你在得到第一次回答之后不再问“为什么”，那么这些问题可能仍然收效甚微。

你寻找的人，是要和我们的关键资质要求看齐的，尤其是激情、坚持、适应能力、正直和职业精神。任何人都可以通过外在行为把这些特征表现出来。坚持问“为什么”，由表及里，通过现象看本质。

白板编译器

除了带“为什么”的问题之外，还有编程问题（或者其他类似的关于解决技术难题的问题），也能揭示应聘者将如何开展工作，这些可以成为强有效的面试问题。问题是，哪里去找到这些合

适的问题？

不用费神去找网上宣扬的、被人过度使用的、无数的糟糕样例。接下去，我会一步一步教你如何得到属于你自己的全新的好问题。

1. 选择2~3个在过去18个月内你或你的团队碰到过的真实问题。它们的解决方案应该只用一块白板就能写出来，并且至少要引用3个不同的变量。这确保你选择的问题具有挑战性，并且解决方案也要求简短而非凡。通常来说，他们是一些小的函数，某个设计问题的片断，或一些特别的测试案例。

2. 把每个难题分解成一个简单的核心问题，并把它用作第一个问题。当应聘者建立起信心，就增加更多的复杂性以提高难度。比如说，让应聘者找出更好的解决方案，引入新的情况，或者要求给出一个更加健壮的“产品级质量”的解决方案。

3. 指出他们在分析的过程中忽略的地方，逼迫应聘者一定要给出答案，这样可以看出他们对熟悉领域之外的问题作如何反应。

4. 多准备几种问题解决方案，并且指出真正难题中核心问题所在。

2~3年前的问题就不要再用了。你总是会碰到新问题，因此可以依次把最老的问题扔掉。这使得面试对你来说更加有趣味，对应聘者来说也更加贴切，这还使得应聘者从互联网上获取答案要困难了许多。

> **作者注：**人们非常喜欢用老问题，因为那样比较容易，而且他们已经了解了怎样去评估应聘者的解决方案。把这个习惯克服掉吧！新的问题会更好。

面试的目的是了解每一位应聘者如何一步一步地解决问题，而不是必须得到正确的解决方案。他们跟你在一起的短短几分钟内未必能找到任何解决方案，哪怕他们是非常优秀的应聘者，而其中的原因是多方面的。你要看的是，他们在解决问题的时候是如何表现出他们的核心能力的，比如：

- 他们是否在一个策略行不通时把它指出来？
- 他们是否问你问题，以帮助他们正确理解问题？他们是否听从你的暗示？
- 他们分析了他们的过程和结果吗？
- 他们是否应用了多个策略？

当应聘者在白板上讲解的时候，你要全神贯注。你要的不是答案，你要知道的是他如何得到答案的过程。

我说的这些很多都可以从“面试官工具包”中找到——那是人力资源部门在许多开发经理（包括我）的帮助下收集起来的一个巨大的资源。你也可以跟你的朋友和开发团队就一些问题进行分享和评论，但小心不要重复他们的问题，否则解决方案很快就会在网上发布出来了。

> **作者注：**如今我常常需要在不借助白板的情况下评估应聘者的能力，因此我使用了角色扮演。其基本依据很简单：如果你想评估某个人代码写得怎么样，那就让他写代码；如果想要评估某个人对他所做决定的信心怎么样，那就对他们的决定提出质疑。

> **在线资料：**如何进行面试角色扮演（InterviewRolePlaying.doc）。

帮招聘专员做准备

在你的面试官为应聘者准备了一些备用问题之后，你还必须帮招聘专员做些准备。把职位描述（包括职位名称和级别信息）发给你的招聘专员，还有一个长长的预备面试官清单，这个清单越长，面试安排起来就越容易。

如果你正在新建一个团队，那从你的伙伴团队中借几个面试官过来，把面试分开安排，以免给任何个人增加太多的负担，记得回报这份人情。不管你是怎样找到那些预备面试官的，由招聘专员从他们中挑选参加面试的人。

把你所有的面试官像菜单一样呈给招聘专员：第一轮面试，从这5个中选一个；第二轮面试，最好在这些当中选一个，如此等等。你可以重复使用面试官的名字并且带一点创意，目的只有一个，就是让面试安排起来尽可能地容易，这样你安排的面试就会更多、更快。

再次帮面试官做准备

最后，在面试到来的前一天，你会收到一封关于面试反馈指示的E-mail，写下你对面试流程的指示，马上回复这封E-mail。把你要招聘的职位解释给你的面试官们听，告诉他们你想重点考察这个应聘者的哪些方面，以及你想要面试怎样进行。

每个应聘者都能给你的空缺职位带来不同的东西，每个应聘者也可能以不同的方式引起你的注意。你必须把这些向参与面试流程的人解释清楚，同时也要说出你对应聘者的优势和弱势的感觉。

接下去，告诉面试官你想让他们中的每一位分别侧重考察应聘者的哪种能力，每个面试官都应该有一个清晰的任务，尽量避免重叠。要为后面的面试官留点空间，让他们追踪当天早些时候提出来的问题，但要保证每个面试官都扮演着自己的角色。这可以避免问题重复、时间浪费和惨痛的失察。

友情提醒

最后，提醒面试官注意下面这些事情：

- 在下一轮面试开始之前，私下跟下一轮的面试官简单地交待几句，谈谈对应聘者的看法，以及你的聘用或不聘用的决定，让后面的面试官知道你已经问了哪些类型的问题，还有哪些类型的问题需要再问一下。

> **作者注：**考虑到面试官们每个人都有一个清晰且几乎没有重叠的任务，你可能想知道，为什么还要把你的印象或你已经问过的问题告诉下一位面试官呢？那是因为世事难以预料，我们都是人，不可能总那么死板。也许你的面试官临场做了些改变；也许面试离题了；也许某个有趣的性格特征值得特别的关注。谁知道呢？灵活一些是件好事！

- 面试之后快速写下反馈，并发送出去。
  - 首先写出你的聘用或不聘用的决定。
  - 接着对你的印象做一个简短的概括，包括你的感觉和真实情况。
  - 对于你的每一个印象，给出面试过程中具体的例子加以证明。应聘者说过的话尤其有用。

- ❑ 写出你问过的问题，以及哪些问题还需要再问一下。
- ❑ 根据你的希望做一个结论。

- ■ 在做聘用或不聘用的决定时，不要使用“两可”的字眼。做出你自己的选择，然后为它辩护。默认情况下，“两可”的聘用就是不聘用。
- ■ 不要担心与前面的面试官有不同的意见，也不用感到尴尬。说出你的真实感觉，以及你真正看到和听到的东西。

最后的难题

有了前面所有的这些准备，参与面试流程的人应该能够跟每一位应聘者进行强有效的面试了，对应聘者职位的胜任度也有了深刻的认识。

最后的难题落在了你的 AA 面试官身上——面试流程中最后一位面试官。像面试流程的其他部分一样，你应该在你的面试流程菜单上列出几位可选的 AA 面试官。AA 面试官应该符合下面的条件：

- ■ 至少跟你的招聘专员一样熟悉你空缺的职位和期望。理想情况下，你应该亲自跟 AA 面试官讨论那个职位。
- ■ 整天忙碌于面试过程，收集最近的反馈，进一步梳理差评，专注于存在麻烦的领域。
- ■ 如果应聘者实力很强，要准备好当说客，说服他选择你的职位和团队。

> **作者注：**在微软，最后一位面试官叫 AA（As Appropriate），因为他的资格比较老，只在适当的时候才做最后一轮面试——也就是说，当应聘者真正有意选择某个职位的时候。

有了一群优秀的面试官，还有各方面充分的准备以及积极的招聘宣传，你的面试流程将会成为你的强项而不是弱点，成为你的“动力燃料”而不是拖你后腿的累赘。你将使你的空缺职位很快地被优秀的人才填补上，你也能够恢复元气，全力以赴，重新投入到产品发布的工作中去。

2004 年 11 月 1 日：“最难做的工作——绩效不佳者”

评审结束了，你可能已经想好是否换个团队会更好。因为你知道哪些团队会更多地给出 3.5 及 4.0 的评分，知道哪里充斥着好高骛远的人，这样你就可以提升得更快，你可能已经决定到哪种团队中去了——一个聪明的决定。然而，你可能会疑惑，为什么会存在这种不公平，这些好高骛远的人是哪来的？他们为什么会在这里？对这样的情况大家都是如何处理的？

> **作者注：**实际上，“跛足”团队在 1～2 年内就会显现出来，但最好所有人都能卷入进去，即使以前他们从来没有那样子过。这也是本栏目所要阐明的观点。

好吧，自己照一下镜子。你是否给你部门的所有员工都打了 3.0 分，让绩效差的人侥幸躲过？你是否让入门级的员工多年来一直保持着入门级的水平？你是否把平庸的员工扔给了另一位管理者或者另一个团队，而不是你去帮助他们？猜到什么了吗？你才是问题！

> **作者注：**在微软旧版的评级系统中，级别2.5及3.0是不合格的，级别4.0与4.5就很优秀。3.5算是合格，司空见惯。

你期望什么

绩效差者不是问题，只要他们还能明辨是非，那就没什么问题。当然，他们并不优秀，但他们通过了考核，那看起来就足够了。你没有对他们的绩效设置更高的期望，结果是，绩效差的人绩效仍然还是差。他们找的是让他们更舒服的部门，他们雇用像他们自己一样平庸的家伙。最后，他们拖垮了所在的组织，然后像沉船上的老鼠一样四处逃窜。

但那不是他们的错，那是你的错！你必须对他们有更多的期望，我知道那很难。绩效差者未必是卑鄙下流之辈，他们常常友善而充满思想。他们有家庭，甘担义务，他们想好好工作，并试图把工作做好。但“想”和“试图”是不够的，不过，要跟他们说明这一点也不是件容易的事。

一定要克服它，也要克服你自己。对平庸的员工太随便其实是对他们的不尊重，也是令人讨厌的。你不是在帮他们，也不是在帮你自己，更不是帮公司。实际上，你把这三方都残忍而自私地伤害了。

为什么呢？因为你在放任绩效差的人走向失败。你的员工每天早上起床，穿戴整齐，然后一头扎进他们终将失败的工作，而这恰恰是你一手造成的局面。你知道那感觉起来像什么吗？你欠他们了，你不够正派，你本应该指出他们所处的位置，并且告诉他们采取什么措施去改进的。

> **作者注：**我这里有些言之过甚了。如果管理者什么也不说，那会放任绩效差者走向失败，然而，在大部分情况下，管理者都会跟员工一起讨论他们的绩效。遗憾的是，他们常常没有表现出应有的明确和坚定，因为他们不想让自己看起来是那么冷漠或残忍。实际上，态度明确而坚定是管理者能做的最友好、最有建设性的事情了。

与虎谋皮

那么，假设你有一些员工，他们在某个地方（或者很多地方）不能达到你的期望，你会怎么做呢？很简单。在每周进行的一对一面谈中，你对他们说出你的期望，指出他们哪里表现得不够理想，而你的期望又是什么样子。告诉他们不要着急做记录，你会发送E-mail给他们。

为什么要把你的期望以E-mail的形式发送给他们呢？是为律师准备的证据，对吗？错了！你写下你的期望，是为了能让他们了然于心。

绩效问题究其原因，信息的错误传达往往难辞其咎，而且是最大的问题之一。如果你的员工很清楚你的期望，那就不会有问题。当他们表现得不够理想时，你的员工必定很清楚其中的原因，他们也知道该做些什么去加以改进。短短的一封E-mail大有帮助，把你的期望一点一点列出来，指出他们不足的地方，同时也指明做好时应该是个什么样子。

寻求专业援助

但如果你看不到他们有改进的迹象怎么办呢？如果你感觉按照目前的状况发展下去，他们只能得2.5分怎么办呢？赶紧联系人力资源部门的专家。专家会协助你，以确保你的员工得到正确

的信息与适当的帮助，如果他们需要的话。

有时候，绩效问题是由一些个人原因引起的。依据具体的情况，微软可能有义务帮助员工度过那些时期，人力资源部门的专家知道有哪些做法，并且可以适当地给你的员工一些应有的帮助。不要在你的员工面前扮演心理学家、医生或律师的角色，把所有那些事情留给人力资源部门的专家吧，只有他们才有资格去做出那样的处理。

失败不是唯一的选择

如果绩效问题依然如故，人力资源部门的专家可以帮你做个最好的选择。尽管在有些情况下，你要对你的员工进行重新评级或改变他们的角色，但在更多的一般情况下，你可以给你的员工如下 3 种选择：

- 主动离开公司。
- 改进。
- 被迫离开公司。

微软在帮助员工实现转型方面做得非常好，这也是让人力资源部门的专家参与进来起到他们作用的原因。通常情况下，当员工面临的选择很明确但还是达不到期望的时候，他们就会释然离开公司，去寻找更有希望成功的新机会。这是次优结果，可能你想不到，这也是常常发生的事情。

> **作者注：**不是所有有绩效问题的员工做的选择都一样。针对每个人的情况，人力资源部门能够帮助你量身定制适当的应对措施，这里我只是列出了常用的几种。

目标就是成功

最好的结果是，员工的绩效得以长足提升，甚至超过了你的期望。优秀员工又回来了！你不再需要解雇谁。你也不必花上几个月的时间去招聘新人，还要让新人跟上大部队。你的员工维护了他们的工作并得到了尊重，他们走向了成功。那是一个巨大的共赢局面。

因此，如果绩效不佳者选择自我改进，你必须相信他们有能力做到，并且他们多多少少会取得一点成功。即使你确信这个人不可能彻底转变他的职业生涯，你也必须相信他总会以某种方式克服眼前的障碍。

为什么呢？因为如果他能成功自然更好，而如果你怀疑他，他一定会知道。当你不相信一个人的时候，他能够自己判断出来，从而他认为，那意味着他在微软已经没有机会取得成功了。记住，如果你不相信你的员工能够改变，你实际上是在把他们推向失败，那可能导致悲剧的结果。

无所求，则无所获

为了把你的员工推向成功，你必须回过头去为可靠的绩效设置清晰的期望。你的期望必须是书面的，以把曲解和误解降到最低的程度。每周复审一下这些期望，面对面地跟员工交谈，同时还要做记录——哪些地方已经有了改进，剩下还要做些什么。

绩效不佳者常常一开始会有个较大的提高。但如果管理者对此过于兴奋，急着告诉员工他们很满意，结果却只会是让员工停止了进步，甚至出现倒退。问题是员工产生了错觉，似乎他们已经超出了期望，而实际上他们只是稍有改善，但仍然还不够理想。

你必须表现出对他们的勉励，但要将关注放在最后的成功上，要这样说，“嗨！我很高兴看到你取得了重大的改进。我真的很欣赏你为此所付出的努力。保持这个趋势，继续提高其他方面，你就能一步一步达到你的职责寄予你的期望了，最后到达成功的彼岸。”这样你就能展示出你的注意和关心，让你的员工打消疑虑，同时又重申了一下清晰的期望水准。

通常情况下，如果一个绩效不佳者本身不具备扭转局面的能力，即使他尽了最大的努力，表现仍然不够理想，他就会意识到自己的短处。如果你想让他们也能成功的话，让他们主动离职，但要保持和睦，并且让他们知道你对待他们很公平，保持着尊重。这样你才给了他们真正的机会。

你不会总能如愿

有些情况下，绩效不佳者会过于顽固或高傲，他们不承认他们达不到你的期望。那你就要为一次被迫辞职（解雇）准备所有必要的文件了。人力资源部门的专家会帮助你和你的员工走过这艰难的一步。

如果你确实需要解雇一位员工，你必须给你的部门发送一封尴尬的 E-mail，“［员工姓名］离开微软，追求其他机会去了。”最难办的一点是，你只字不能提关于那件事发生的缘由。你必须保护你以前员工的隐私，哪怕他跟别人说了一个不着边际的理由。

你部门的员工想知道到底发生了什么事情——这种情况时有发生。他们会问你某位员工为什么会离开。尽管你不允许把关于那位员工的任何私人情况告诉他们，但你可以回答他们真正的问题。当人们问起别人的情况时，实际上，他们几乎总是在为他们自己的工作安全担心。那也许显得有点自私，但其实也是很自然的。

好好利用这个机会，给他们一些反馈，以帮助他们把工作做得更好，并且打消他们关于他们自己地位的疑虑。你也可以让他们恢复信心：“微软对每一位员工都有强烈的期望，因为微软知道，只有公司拥有了非凡的员工，并且他们在一个健康、相互支持的环境中竭尽所能，公司才有希望成功。”这种说法我们任何人都会以之为荣的！

> **作者注：**如果你在超市或者停车场遇到之前你解雇的员工，你会怎么办？是不是很尴尬？不，绝对不是。我也曾有过一批绩效差劲的人，也解雇了他们中的一些人，我也确实偶遇过之前的这些员工。见到他们很开心，因为我们的态度是积极的，也有共同的目标——将他们塑造成为成功人士，这些员工们也像一贯的那样，见到我很开心。我们谈了他们的工作，通常比之前要好得多，这就是我的主要观点。

2005 年 9 月 1 日：“随波逐流——人才的保持和流动”

考核期到了。管理者和员工们都会觉得饶有兴趣，而对于音乐产品部门来说，这才刚刚开始。当考核分数公布的时候，音乐开始响起，一大群工程师从他们办公室的位置上跳起来，然后去别的部门寻找空缺的位置。当所有有趣的位置都被占满后，音乐也就结束了。同样的“游戏”在产品周期结束的时候也会上演，只不过产品周期不是这么确定（那是另外一回事）。

对于那些曾经离开过自己的座位，但又没有在其他地方找到新位置的人来说，日子就难过了。他们常常会被当前的团队疏远，因为他们试图“弃船

而去”。很倒霉？我不这么认为。管理者应该鼓励他们的下属（包括他们喜欢的人）去追求新的机会。否则的话，不说管理者是在搞破坏，欺骗，应该受到严厉的谴责，至少他们也是自私、愚蠢、目光短浅的。当管理者在做关于业务方面的决定时，如果他们把自己的利益放在了微软的利益之前，那他们明摆着就不再为微软工作了。我想，微软也应该停止支付他们的薪水。

不要告诉我，“噢，但这个项目还要靠这个人呢！疏远他并且抑制他的个人发展，对我来说完全是必要之举。这也是为了公司好。”那真是一坛子的粪便！你正在试图告诉我的是，你为人才流动没有做过任何的计划，现在它真的发生了，你却想逃避恢复、招聘、再教育、再分配等所有的这些事情。换句话说，你脑死亡了，你很懒，但是你还要通过自私自利之举来横加掩饰。只有幼稚的猎人才不愿移动，尽想着守株待兔。

> **作者注：**好吧，我一直忍到了第9章，但我现在再也忍不住要说了：“我非常喜欢重读这些栏目。”当初写下它们的时候往往是痛快发泄。为了本书的出版，我有机会重读了它们，这也让我再次体验到向那些我实在很鄙视的行为泄愤所带来的满足。这种感觉很美妙！

我只是想环球旅行

优秀的管理者应该期待每年大概10%的人才流动。糟糕的管理者应该期待更多，不过他们可能意识不到这一点，当然我这里也不去深究它。

如果你是一位优秀管理者，你的部门有20个人，那你应该期待每年有2个人离开你的部门，有时还会多一点，有时也会少一点。即使是拥有5名下属的主管，你也应该期待每两年至少有一个人会选择离开。人们离职的原因是多方面的，而且很多时候都跟你或你的团队无关：其他团队的朋友关系、新技术、环境的改变、工作以外的关系，如此等等。你不应该认为他们是在冒犯你。

不错的水坝

但是，你应该如何去处理人才流动呢？有些管理者走了极端，试图阻止此类事件发生：

- 他们在奢侈的礼物和活动上花费大量的金钱，想鼓舞士气，而实际上他们举办更为频繁、开销低的活动会好得多。

> **作者注：**你可以在本章后面部分“一对一与多对多”中了解到更多关于有效提升士气的例子。

- 他们许诺较大的角色和晋升——那些承诺他们并不能完全控制，也无法兑现。
- 他们不允许团队中的任何人出去参加面试，那毒害了团队士气，使得整个团队感觉起来就像是契约佣工一样。

> **作者注：**微软后来改变了规则，使得管理者不再能够拒绝他们的员工出去参加面试，只有副总裁仍然保留着那种特权。

试图阻止人才流动，就像是筑起一个水坝想要阻止河水流动一样，这样做短时间内可能是有效的，但当水坝破裂或河水溢出的时候，你的团队就会被中转的急流冲走。更糟糕的是，这些做

法会降低士气，使得你的团队对新成员没有太大的吸引力，这给以后的招聘造成的麻烦可不小。

像河水一样流动

取而代之，处理人才流动最好的办法是：期待它、拥抱它。如何做到呢？想一想河流，流动，流动……

把你的团队想象成一条“河流”，而不是一个“湖泊”。湖泊是静止不动的，它没有改变的能量或动力。如果你的部门也像湖泊一样静止不动，他们就会变得平庸和自满，他们会痛恨风险。而河流总是在流动和改变着，并且蕴含着许许多多的能量，你需要的是河流！

河流依赖河水的流动，而你的团队依赖人员和信息的流动。你可以考虑把所有人分成3组：新鲜血液、新任领导者和准备迎接新挑战的老员工。下面解释这3组人应该如何去平衡和流动：

- 最大的组应该是新鲜血液。在他们当中，不是所有人都会成为技术或组织的领导者。
- 有时候你会拥有比老员工更多的新任领导者，有时候也可能反过来，但理想情况下你应该维持一个平衡。
- 至于流动，你要控制稳定的流量，让一部分新鲜血液变成新任领导者，也让一部分新任领导者变成老员工。
- 流动的关键是，新鲜血液流进来，老员工流出去。为了达到这个目标，你希望你的老员工在阻塞流动、破坏给别人的机会之前转出去。

不是所有的技术都会以相同的速度流动。核心引擎（比如Windows内核）流动得很慢；而基于网络的服务（比如MSN搜索）流动得很快。你需要根据你的具体情况做出调整，但即使是最保守的技术也都要改变和流动。

你怎样才能成功地鼓励并维持一个健康的流动呢？

- 持续补充新人进来。
- 灌输信息分享是一种生活方式的思想。
- 塑造组织结构和角色以创造成长机会。
- 为老员工找到新的挑战。

新鲜血液

至于新人的持续补充，没什么能胜过实习生和校园招聘的了。很显然，你也将从社会上招聘人才，以及接纳内部转移过来的员工，但实习生和校园招聘应该成为你的主要选择，因为他们有新鲜的观点和长期的潜力。

每年从校园招聘到的人数，应该在你的员工总数中（包括那些空缺的职位）至少占5%。因此，如果你的团队有20个开发人员，那你每年至少要从校园招到1个人；如果你的团队正在扩张的话，则需要从校园招更多的人。即使在一个扁平的组织结构中，仍然至少有5%的人员损耗，因此，即使你当前没有空缺的职位，你也要去寻找有天分的年轻人。从校园招聘过来的人，有时候要等9个月之后才能正式上班，在那段时间里什么事情都可能发生，因此要提前做计划。

实习生是仅次于校园招聘的最好资源，不过他们需要多一年的时间才能加入你的团队。因此，你可以像校园招聘那样招到一样多的实习生。千万不要计划发布实习生的代码。在最好的情况下，

他们也应该结对编程才能直接参与产品代码的工作，然而，也千万不要给实习生分配一些打杂的活儿。正确的做法是，给实习生指派一个能力强的导师（下一个有望成为主管的人），并且让他们加入到令人兴奋的项目中去（在一些非常酷的功能或孵化工作上以伙伴的形式把他们配备上去）。你要以未来全职雇员的标准去考量他们，并且说服他们：这世上再没有比到微软来为你自己而工作更好的工作了！

分享就是关爱

当有新人加入到你的团队时，你希望他们成长为新的技术上或者组织上的领导者。实现这个愿望的唯一途径是通过信息和知识的分享。分享的方法有很多，下面只是列举了一小部分：

- 维持一个在线的知识仓库，以说明你的部门是如何开展工作的。它可以是一个带有版本控制的 Word 大文档、SharePoint 站点或维基网站——任何最适合于你部门的东西。这里的关键是，要让它容易被访问到，并且内容总是最新的。对于一个新人来说，他在第一个月内的任务应该就是更新它的内容。

> **作者注：**现在我最喜欢的做法是将一个 OneNote 记事本放在团队的 SharePoint 网站上，这种做法太美妙了，因为它比维基还易于编辑，有维基的连接功能，还有漂亮的在线同步功能，而且在 Web 上用新的 Office Web 应用软件查看非常舒服。唯一不足之处是没有维基的自动分页功能。

- 为所有的项目和任务使用伙伴系统。从指导指定的复审者和后备人员，到全面的结对编程，工作安排可以是上面范围内的任何事情。这里的关键是，没有人独自工作，没有信息被隔离。
- 定期把人们聚在一起（理想情况下应该是每天），讨论工作进度以及碰到的问题。没什么能像“高带宽”沟通这样促进信息的分享，哪怕每天只有短短的 15 分钟。

伙伴系统对于新任领导者的成长和老员工的过渡显得尤其重要。如果一位老员工的离职会引起过程中的关键知识和能力的丧失，那样就很危险。通过持续的信息分享，你就能舒缓由于压力引致对老员工的束缚，你也可以使流动和过渡变成一个积极而自然的过程。

成长空间

就跟移植植物一样，你必须给你的员工足够的成长空间。你怎样构建你的组织？你怎样分配任务？你怎样给员工设计并推进他们的成长路径？所有的这些都能促进成长空间的营造。

首先考虑成长路径。新的职业生涯模式是一种美妙又具体的指导方向，成长路径如何应用到你的团队呢？你应该去了解每一位员工渴望的成长路径，以及他们当前所处的职业阶段，然后，你跟你的领导们应该一起讨论一下，看看那些所渴望的成长路径对每个人是否都是现实的，如果不现实，你要如何去调整？

部门的组织形式常常会阻碍员工的成长，所有高级员工可能都在一个团队中，而新手都在另一个团队中。这种情况一定要改变——重新整理、平衡和改组。组织的不平衡状态放任得越久，你就会有越多的麻烦，你承担的风险也会越大。

组织结构的调整可以为成长创造许多新的机会，好好利用它们非常关键。给你的员工分配一些任务，甚至给他们分配一些超出他们的舒适范围之外的新职责。很自然，可以把他们跟更有经验的伙伴搭配在一起，以降低风险并加强学习，但不要总是让相同的人做相同的事。基于个人渴望的成长路径进行任务分配，大家就能共赢。

我必须要旅行

当然，如果最老资格的员工留着不走，没有人能够往上移动。除非你的部门正在扩张，唯一能够给成长创造空间的方法只有请你的老员工离开。所幸的是，那正是他们所需要的。如果他们在你的部门呆了足够长的时间才得到了高级职位，那让他们保持成长的唯一方法，就是让他们到其他地方去接受新的挑战。

因为你已经重视了人才流动，丢失高级员工就不再是什么大不了的事。他们早已经跟大家分享了他们的知识和经验，他们在项目中的伙伴也早已熟悉了他们的工作。现在他们所要做的，就是在你逐步的支持下，到其他地方去为他们找到一个更合适的位置。这种忠诚和支持不仅会得到你的高级员工的感激，同时也会得到整个团队回报过来的忠诚和尊重。

记住，整个团队都在看着你怎样对待最老资格的员工！那也预示着未来某一天你将怎样对待他们。如果在任的员工看到老员工得到了公正而慷慨的对待，看到了老员工是带着荣耀、美好的祝福和光明的前途离开部门的，那就太完美了！它传递的信息是，跟这个团队呆在一起，将得到很好的回报。

放任自流

当你跟人才流动抗争的时候，或者当流动正好碰上了你毫无准备的时候，你的项目和效力以及团队的健康都会经受到风险。而当你直面流动的时候，它只是变成了生活的一个自然结果。没有恐惧，没有担心，只是一个有效团队的健康流动。

除此之外，在你的团队中促进人员和信息的流动，还会给微软的员工创造成长的机会，给微软的客户创造价值。更少的压力，更多的机会，更大的灵活性，复合的知识，更高的士气，还有就是更强的团队。你还能要求更多吗？可以“放任自流”了！

2005 年 12 月 1 日：“管理我在行”

婚礼、旅行和管理者之间有什么共同之处吗？跟任何一个成年人谈论这些话题，你肯定会听到一个可怕的故事。在婚礼上，是喝醉的客人、糟糕的天气或者不合时宜的失言；在旅行过程中，那是丢失的行李、混乱的乘客或者慌乱的转机；至于管理者，那是你以前的一位上司的故事，他可恨、不胜任、毫无头绪、傲慢自大、感觉迟钝、密谋策划、眼睛瞪得瞎亮、没有骨气、自私自利、性情古怪……倒并不是我对以前的上司怀恨在心。

抛开极端的情况，大部分关于婚礼和旅行的可怕故事都可以面带微笑地重述出来，还能给大家带来笑声。但关于管理者的可怕故事却不行。当然，所有人都能对以前上司所做之事的愚蠢或荒谬大笑一场，但当你观察那个曾经遭罪的员工的眼睛时，你会发现那里总是装满了藐视。那个员工还没有原谅或者忘记以前的那位管理者的所作所为。

斩不断，理还乱

为什么人们这么容易忘记婚礼上或旅行过程中的倒霉事，但对管理者的不正当行为却始终耿耿于怀呢？那是因为在婚礼结束的时候，新婚夫妇会甜蜜地接吻；在漫长旅途的终点，你最后回到了家；但糟糕的管理者还在——没有开心的结局。糟糕的管理者呆在那里，每天都在，日复一日，干着一件又一件的坏事。

即使你最后逃离了某位糟糕管理者的魔掌，他对你已经造成的影响却无法挽回：消逝的时间、错失的机会和丢失的成果。他的评语和行为时常萦绕在你心头，阴魂不散，他过去的冷酷伤害了你当前的认知，还有你对价值的真实感觉。你开始对所有管理者有阴影，不再信任他们。过去一年内遭受的创伤，却需要 10 年的功夫去修复——那还只是由一个管理者引起的。

对公司造成的损害就更大了。堕落的管理者造成生产力的丧失、浪费和返工、糟糕的质量、浮躁的行为，还有心境不佳的员工，他们要么满腹牢骚地离开公司，要么继续留在公司消极怠工。堕落的管理者也有可能招来法律诉讼，不过那与本栏目的主题偏离得太远了。

做到优秀就行了

“嘿，让管理者得以片刻的喘息吧，”有些人可能哭诉道，“成为一名优秀的管理者很难啊！”不对，那不难，成为一名卓越的管理者才难，而成为一名优秀的管理者很容易。优秀的管理者只需要关注两件事情——两件非常简单的事情，任何人都能做到：

- 确保他的员工能够工作。
- 关心他的员工。

就这么简单！不需要灵丹妙药，不需要激情昂扬的视频，也不需要每天 24 小时地工作。一名优秀的管理者只需要确保他的员工能够工作，并且他必须关心他们。

草率行事

向最有激情的管理者发问，如何保证他们的员工胜任工作，他们会说，“破除万难。”这是当然的，不是吗？但是再问一下，“什么样的难呢？”然后他们会说，“盯着依赖方，”“咬着项目经理要规范书，”或者“要求测试者提供像样的重现步骤。”真是太误导人了！

为什么对跨团队的问题发难会引起误导呢？

- 当管理者试图成为英雄的时候，他们只会像白痴一样去管理。他们盯着依赖方，结果就是得到匆匆忙忙、价值不高的交付成果，而不是在“建造验证测试”（Build Verification Test，BVT）和 API 设计方面开展合作，并从一开始就考虑到各种可能性，制定切实可行的计划；他们追着项目经理要规范书，而不是在设计的各个方面开展合作；他们要求测试者提供像样的重现步骤，而不是进行源代码插桩，以便对于任何原因引起的故障轻易就能进行调试。换句话说，被误导的管理者他们本身就是问题的一部分，而不是解决方案的一部分。
- 跨部门障碍是难以消除的，而且很费时间。管理者在开始勇敢探险之前，需要给他们的团队找到迂回措施。
- 被误导的管理者允许跨部门障碍转移他们的注意力，他们忽略了其他更为直接、基本的利害关系。

当然，解决跨部门问题也是重要的，但通常有更为简单的方法去帮助解决或绕开问题，况且还有更多基本的问题需要去优先处理。

> **作者注：**我还要指出，新任管理者更关注消除复杂的障碍，而不是那些简单的（下面我会谈到），这是他们最常犯的错误之一。新任管理者另一个最常犯的错误是，他们仍然坚持单兵作战，而不是把自己当成团队的代表，他们没有恰当地把工作委托出去。（参见第8章的专栏“有的是时间”。）

我想要工作

那么，为了确保员工能够工作，管理者应该提供哪些基本的必需品呢？正如我刚才所说的那样，那很容易。工程师们不需要很多，他们只要：

- 一张装有电话的办公桌。
- 电力供应（照明、取暖、电流）。
- 一个带有键盘和显示器的电脑（有些人甚至不需要鼠标）。
- 网络访问权限。
- 一个健康的环境（安全、相对安静、能够呼吸的空气）。
- 工作任务。

谁还能把这个搞糟了？也许给员工找到工作任务要复杂一点，但通常那不成问题。然而，有许多管理者能够容忍员工几天都没有电脑——管理者在一位新的团队成员加入之时，还没有为他准备好办公用品。

最近一次当你的网络瘫痪或者周围噪音太高的时候，有什么事情发生吗？你的管理者有没有放下手中的所有事情，尽他所能去解决那个问题？如果没有，那他作为管理者绝对是不称职的。没什么比保证你的员工能够工作更为基本的事情了。

那什么叫提供一个安全的环境呢？你的团队中有敌对状况吗？你的管理者对此做了些什么？这也是进行必要的“抗骚扰”培训的原因。确保所有人都有机会在一个安全的环境中工作，没什么比这还要更关键、更重要的了。

我不是一件物品

优秀管理者必须做的第二件简单的事情是关心他的员工，这并不意味着热情的拥抱或贺卡。你甚至不需要喜欢他们。你只需要关心你的员工，客观、真实地看待他们——他们是人类中的一员。

可能又有人会问，谁还能把这个搞糟了？那能有多难啊？然而，管理者常常把他们的员工看成是“资源”，而不是“人”。他们给员工打上“好”或“坏”的标签。他们把员工看成是东西，而不把他们当人看。

难道管理者没有亲信吗？那也会带来伤害，不是吗？即使你是亲信中的一员，那是因为你稍不小心就会失宠。管理者如果重用亲信，那他也不再把他的员工当人看了，在他眼里，员工成了收藏品。

花点时间去了解和欣赏你的员工，把他们真正地当人看待，这其实不难，更多的是一个承诺。

你必须抛开你自己的喜好和偏见，让其他人了解你，从而你也能了解他们。我意识到这有些多愁善感，不过优秀的管理者的确把他们的员工当做真正的人来尊重，他们不把员工看成是抽象的人头数。

顺便说一下，你不必刻意说什么或做什么以显示你的关心。实际上，如果你不是真的关心别人的话，哪怕再多的言语或行动都无法使他们信服。人们很容易就能自己判断出来。你可以对他们大叫、表扬他们、批评他们甚至使他们失望——只要你尊重员工，他们都能理解所有那一切，并且仍然尊重你。

记住，为了成为一名优秀的管理者，所有你要做的就是确保你的人能够工作，并且把他们当人去对待。

> **作者注：**那是个“充要条件”——意思是说，如果你确保你的人能够工作，并且你关心他们，那你就是一名优秀的管理者（实际上是较好管理者中的一员）。如果你不能确保他们能够工作，或者你不关心他们，那你就是一名糟糕的管理者，哪怕你在其他方面做出了巨大的努力

从优秀到卓越

如果所有的管理者都至少能够做到“优秀”这个程度，那我们公司乃至整个世界都将更加美好。然而，微软是一个带有竞争性的地方，因此你可能想知道怎样才能成为一名卓越的管理者。

很多图书、杂志和研究所都给出过卓越管理者的定义。在我看来，归根结底就只有如下 3 个方面：

- **成为一名优秀的管理者。**不要从“卓越”起步，先从“优秀”做起。这是基础！如果你把这个基础忘掉了，那你也失去了员工的尊重和效力。
- **个性正直。**这意味着，你的言语和行动要与你的信仰保持一致。如果你相信要有一个严格的工作道德规范，同时也要维持一个清晰的工作与生活的平衡，那就以身作则展示给大家看。如果你相信质量第一，那就优先抓质量，你来设置标准，你来定义团队。
- **定义清晰的目标、优先级和界限。**把你的目标、优先级和界限往上、往下以及在组织内部横向进行清晰的沟通，这是高效团队和个人的本质。目标可以以期望、承诺或远景声明的形式出现。不管采用哪种形式，它们都指出了我们要走向哪里；优先级指出了我们如何到达那里；界限定义了我们可以自由活动的安全边界。如果没有目标，我们可能哪儿也到不了；如果没有优先级，我们很容易就会迷失；如果没有界限，我们一不小心就会绊倒。

成为一名卓越的管理者是有难度的。你必须扛得住对你的信仰妥协的压力。你必须清晰而有效地传递一个一致的信息，说明你的远景，特别是你不可接受的东西。你必须做所有的这些事情，同时不丢掉对你的员工的人性关怀，还要留意到为了保持他们的生产力而必要的一些细小（但很重要）的东西。

为人民服务

那就是成为一名优秀乃至卓越的管理者所要做的一切了，它归根到底就是服务。你不再是那个做实际工作的人了，取而代之，你要使你自己成为你团队成员的仆人，你的目标是让他们都能够成功。如果做得好的话，管理是无私的，这就是最高境界！

作者注：放心将这篇专栏与你喜爱的或痛恨的管理者分享，在过去5年多的时间我再没写过什么，因为关于优秀与卓越管理者的话题已经浓缩得够完整与精练了。

2006年5月1日："比较的恶果——病态团队"

本栏目是写给主管和管理者的，但我猜你一定会将它汇报给某人，那也一起来分享一下吧！

你的员工们相互之间咆哮的频率有多高？他们是否组成了"秘密"联盟？你的团队会议是否笼罩在浓重的紧张气氛中？如果是那样，你的日子一定不好过，而我的团队一直以来都相处得非常好。

你的团队一起吃午饭吗？你们鼓舞士气的活动感觉矫揉造作吗？我的团队每周一起吃一次饭，每个月都举办一次鼓舞士气的活动。我们几乎一直与笑声相伴。

当你的团队处于压力之下，他们是否人心涣散？如果你团队中有一位员工效力不支，其他人会不会利用那种境况？他们是否都觉得事不关己，并且自娱自乐？还是你的团队成员之间能够相互照顾？我的员工总是团结在一起，并且相互支持。

如果将我们俩的这种对比汇报给同一位上司，你对你所做的工作感觉如何呢？我们的上司会说些什么呢？他会告诉你要向我学习吗？他会把我的团队的调查问卷、人才保持和生产力相关的数字扔到你脸上吗？你的感觉又会怎样呢，特别那时正值绩效考核之际？

想要挑起战争

你认为我在挑起战争？你说对了，我正是这么想的。如果我们的上司把我的团队跟你的相比，并且当面向你抛出所有的证据，你会希望我马上死掉，或者至少从老板奖给我的骏马上高高地摔下来。你可能会阴谋破坏我的成功，让我看起来像个傻瓜一样。

这听起来很熟悉吗？如果你的团队是病态的，这也正是你的团队中正在发生的事情。你团队中的成员相互之间阴谋拆台，他们各自为战，竭力提高他们自己的相对地位。团队合作只是为那些对"致命格斗"不感兴趣的人准备的。

译者注：1992年，美国推出了一款叫MORTAL KOMBAT（致命格斗）的街机游戏。跟当时红遍全球的游戏《街头霸王》相比，这款游戏做得更血腥、更暴力、更剧情化。《致命格斗》推出后不久，便在全球拥有了相当数量的忠实爱好者。

那么谁才是罪魁祸首呢？是你的员工太好斗了吗？不可能！我从你的团队中接收了一个非常好斗的人，迄今为止他们都相处得很好。那是绩效考核系统的问题，它使人们相互竞争？不对，我也用了同样的考核系统啊！那是大家承受的压力大小不一样？不见得！我的员工他们越受挑战，智慧的光芒就越耀眼。

差别在哪里呢？为什么我的团队表现得这么好呢？想知道吗？答案就是，我从来不对我的员工相互比较；我把他们跟他们的潜力和我对他们的期望进行比较。就这么简单！那才是奥妙！

继续，再来否定我吧。不可能那么简单，是不是？无论如何，我必须对他们进行相互比较；绩效考核要求那么做，对吧？你错了！实际上它就是那么简单，绩效考核未必会带来破坏性。

> **作者注：**本专栏采用了一个很危险的写作手法，它会引起我的读者的直接对抗，同时也让我自己看起来是那么自以为是。不过当初还是经过了一番深思熟虑的，我想让我的读者再次体验一下内心深处的仇恨，像他们孩提时代面对他们的对手时的感觉。既然 I. M. Wright 是我塑造的一个狗娘养的、傲慢自大的形象，他的名声不会受到损害，况且坦诚的人们能够明辨事理。

这不是竞争

我希望我能宣称这个奥妙是我自己发现的，但我没那么睿智。我是从一本关于“子女养育”的书上学到的——《Siblings Without Rivalry：How to Help Your Children Live So That You Can Live Too》，作者是 Adele Faber 和 Elaine Mazlish，由 HarperCollins 出版社于 2004 年出版。道理很简单，孩子们渴望你的注意和关爱，如果你对他们相互比较，你等于宣布了那是一场竞争。然而，不需要竞争，除非你是一个病态的父母。因此，不要比较！

这个道理同样适用于你的员工，他们都想得到他们上司的钦佩和欣赏。如果对你的员工相互比较，你等于就在他们之间宣布了一场竞争——但那不是你的初衷。是的，员工们为了得到所在部门提供的奖项相互竞争，但那种竞争并不会直接发生在你的团队成员之间。

是的，确实有一些像笨驴一样的管理者，他们坚持认为，他们的员工也应该在团队这个层次为奖励而展开竞争。然而，那是如此不合情理、缺少尊重、毫无理性，以至于我个人想把每一个那样的案例都报告给人力资源部门，像那样的笨驴不应该被允许去管理员工。他们应该被移交，勒令在一个永无止境的圈圈里行走，打扮得像小丑一样，同时还有一群孩子尖叫着拉他们的头发……

当团队成员不再相互竞争的时候，他们想要取得成功的话，最好的机会只有在一起合作。生活也变得美好了！

我会给你个提示

为了维护一个功能高效的团队，关键的挑战是避免团队成员之间的比较。不要给他们任何理由去进行相互竞争。但你要怎样去避免比较呢？请参照下面这些简单的提示：

- **描述出你所想要的。**某位员工就某项工作过来征求你的意见，“跟 Pat 的工作相比，我的这个怎么样？”这是个陷阱，因此不要提及 Pat。正确的做法是，向他描述出你所想要的，“那是个很棒的实现。它清晰而简练，易于测试，并且使用了所有正当的安全保护措施。”
- **对你的员工表示出信心。**某位员工对另外一位员工提出了抱怨，“我的工作被 Joe 阻碍了，除非他马上签入他的源代码。但是他说在代码没有达到完美之前，他不想做代码签入。你能让他改变主意吗？”这位员工（也许是无意的）想要让他自己看起来比较好，而让他的同事看起来比较糟糕，因此不要上当。正确的做法是，表示出你对你的团队的信心，“噢，你被阻碍了，但 Joe 也是在做一个高质量的工作。听起来比较难权衡哦！不过我相信，你

们俩能够坐下来好好讨论，然后找到一个合理的折中方案。”

- **注重满足需求，而不是“公平”。**你提拔了某位员工，但她的同伴产生了嫉妒，“怎么Jane得到了晋升，而我却没有？那不公平！我们俩工作都很努力啊。”很多事情上都会牵扯到公平的问题，但生活是不公平的，不要去比较谁有、谁没有，向你的员工重申要求，并且探讨如何去达到那些要求，“我希望每个人都能马上得到晋升，但那不现实。让我们来探讨一下为了晋升你还需要做些什么吧，然后制定一个计划以帮助你达到那个目标。”
- **对事不对人。**某位你没有提拔的员工表示了不满，他想要知道细节，“为什么Jane得到了晋升，而我却没有？莫非她比我好？”这又是一个陷阱。这种情况下，我们的绩效考核系统可以提供很好的支持。不过要对事不对人，“Jane在设计解决方案方面证明了她对功能团队的领导力，那个方案被用在了一个关键案例中，并且被团队完整地实现了。那种领导力是你还需要培养的东西。你的设计技能很强，但你没有展示出必要的领导力，以使你的设计取得大家的一致同意，并保证它在整个实现过程中落到实处。”

“对事不对人”在评审会议上同样也很管用。你不能说，“Jane比Pat好，”而要说，“Jane已经达到了这个技能水平，而且在这些地方展示出才能来了。Pat没有表现出那些技能。”尽管还是有一个比较，但那不是竞争，Pat将不会去努力超过Jane，Pat所要做的努力是获取Jane那样的技能。这是一个微妙但很重要的差别。

团结在一起

拥有一个有凝聚力的团队你将会得到巨大的回报。他们工作做得更好，更有韧性，具有更高的士气，人才保持得比较好，沟通更高效，整个团队也更容易管理。成为一名优秀的管理者意味着要关心你的员工和其他一些方面，这在“管理我在行”这篇文章中已经谈到了（参见本章的前一篇专栏）。但是，那些并不能保证你得到一个有凝聚力的团队。

你需要避免在你的团队内部的相互竞争。为了达到那个目标，你必须把你的员工都看成是个体，要了解他们的潜力，并且关注他们具体的需要和你对他们具体的期望。关心你的员工，但不拿他们来相互比较和评判。

当你消除了个体之间的竞争，剩下的就只有团队成功了。当然，团队中仍然会有争吵和抱怨，但你的团队会把那些看成是他们共同成功的一种掣肘，而不是为了提高他们个人的地位。一个团队在一起工作，相互支持，最后产生卓越的成果，没什么比那更好的了！

顺便说一下，我有几个孩子，他们在一起相处了10年，至今还没有出现过任何敌对的迹象。

作者注：如果你喜欢行为科学方面的图书，你也可以阅读一下《Don't Shoot the Dog》（Ringpress Books出版社于2002年出版）。这本书的作者是Karen Pryor，她以前是一名海豚训练师。她在书中涉猎很广，从如何使你的狗安静下来，谈到了如何“驯服”你的岳母。如果你想在商业方面获得更为全面的东西，Thomas F. Gilbert写的《Human Competence: Engineering Worthy Performance》（Pfeiffer出版社，2007年出版）对或大或小的企业变更作了全面独到的分析。我将在后面“文化冲击”章节中对Gibert的理论进行更多讨论。

2008 年 3 月 1 日："必须改变：掌控改变"

现在是美利坚政治季，到处都欢呼着一个词"改变"。政治家们为争谁能更好地引领改变而缠斗着，他们都宣称他们是改变的代言人。异想天开的拥趸们上蹿下跳地叫着"改变"的口号，改变，改变，改变，好似改变就是人们所向往的，好似改变就会让人过得更快活。

事实并非如此。这些人都疯了吗？他们不过是一时精神失常。改变这样的理念是很诱人的，如果你想要一位新的领导人上位，那这就是你想要的改变。但是，如果人们真的想要改变，它可能已经发生了，不要得意忘形。事实是，人们厌恶改变，一如我的朋友 U. R. Rong 说的，"一成不变：改变恐惧症。"

> **作者注：**在去年夏天我休假期间，U. R. Rong 代我发表了两篇内部专栏，他也设立了他自己的博客，很有意思："渐进式发展"（blogs. msdn. com/b/progressive development/）。

是的，改变是不可避免的，坦然接受改变是一个正确的决定，但是万变不离其宗。而奇怪的是人们顽固地固守旧观念，循规蹈矩甘受生活之不幸，而不是尝试改变他们的命运。所以，一旦要在工作、家庭或社交中做些改变，人们就不太乐意接受。

变则通

但是如果有一种改变正是你所希望的呢？一成不变可以省很多事，但是毕竟不会让你有所进步。为了让你的生活、团队乃至这个世界增添些光彩，你必须要有所动静，但你如何面对人们对横亘不变的深切希冀呢？

很高兴你能这么问，这并不难，这里有 5 个步骤：

- 有个想法。
- 启动水泵。
- 试下水。
- 跳进去。
- 做出实效。

但是，大多数人自有了想法后，就直接"跳进去"，略过了第 2 步及第 3 步。这是愚蠢的，忽略这些步骤会增加你失败的风险，别跟我胡诌说，你是不想错过机会。改变是很费时而且成本很高的，由于缺乏耐心或愚蠢至极使你没做过多慎重考虑，就扔个骰子来作出决定根本不是勇气的体现。不做则已，做则必须成。

把你的想法说清楚

在你想做出改变之前，你必须将你的想法说给你的同事及管理层听。我在第 8 章的专栏"寓利于乐，控制你的上司"中对此有过详细讨论。重读一下，为你的想法找足理由，那么你的上司及同事就不会对你有什么异议了，他们会与你共进退。

如果一开始你就无法获得支持，换种方式争取，或者及早作罢。如果你未获他人支持就直接

做了，你定将万劫不复。掩耳盗铃，自以为万事大吉，认为别人也是妄想症者，那你这个人也真就是妄想症者了。你不是那些无人理会的怪才，这些怪才才会一直不断尝试直到有人相信他，如果在没人支持的情况下你自行其事，你就是个只会对牛弹琴的白痴。

> **作者注：**每天我好似总听到有人哀怨没人愿意听他的，或是管理层是怎么怎么以“理由不充分”为托辞来阻挠他们尝试改变的想法。但是，深究一下就会发觉，这些人从没有为他们的目标赢得支持，或是他们没有将具体举措说明白以使人们看清楚存在的问题以及由此带来的改善之处。

“寓利于乐，控制你的上司”中我忘了说明的是你要确定你的举措、度量标准及目标获得了一致认同。在你向你的目标努力时却没有说明白你的目的是什么并获得赞同，你就不会得到众人的支持。没有这种明确的表态，你就称不上是成功，你做的谁也不会注意到。在第2章的“你怎么度量你自己”中你可以了解更多关于如何正确使用度量方法的内容。

你准备好了吗

现在你有了一个行动计划，想开始干了，是吗？不！要倾听，这很重要。改变是个痛苦的过程，不断反复。改变跟两种易变的情绪紧紧相连——不安与痛苦。

改变与不安相连是因为它充满了变数。回想一下上一次你换工作的时候，你是否为新工作而焦虑不安？你是否会在新的团队里获得成功呢？你会喜欢这份新工作吗？你是否忧心如果你对新工作不适应你该怎么做？改变的过程中人们往往会易怒、急躁甚至不苟言笑。贸然行动，以为大家都会喜欢实在是愚蠢至极。

改变与痛苦相连是因为，当做出改变后，老路子已一去不返了。大家不熟悉也不适应新的方式，人们按老套路来已经很久了，他们还在怀念那些老套路，这比较痛苦，你应该了解这种痛苦的5个阶段：

- 抗拒。
- 发怒。
- 讨价还价。
- 沮丧。
- 接受。

无视发怒的阶段而贸然行动是很不明智的，无视讨价还价及沮丧的阶段也好不到哪去。你如何能避开这种灾难呢？启动水泵。

> **作者注：**“启动水泵”是源于将液体注入水泵以清除空气使其正常工作的一种说法。通常，其意思是事先做些准备以使你完成既定的目的。

准备就绪

启动是事情发生之前人们整装待发的一种状态，启动使人们能更好地适应新环境，让人们安心。如果你有过彩排或演讲练习，你就有过启动的状态。

启动改变就是事先告诉人们这种改变可以带来什么，在改变之前，要跟大家谈谈以后会是什么样子，具体一些。那在改变发生之前多久就开始启动呢？至少在经过讨价还价阶段之后，越长越好。这要看改变的影响程度有多大，这可能要几天到2个月的时间。

在启动期，你应该多讲讲改变是什么，可以发两封E-mail或跟别人面谈几次，你要以他们的切身利害关系讲讲为什么这次改变这么重要，你也要谈谈这次转变将遇到的不可避免的困难，但是要阐明达到目标的清晰路线图。

事先介绍这次改变，使人们在他们要实实在在干些难事之前让他们先体会其中的悲苦与不安；同样，通过这种方式，在改变产生影响之前，你可以听听他们的反对意见（发怒）及一些建议（讨价还价）。这样，你就有了调整的时间，也给了别人以主动权及主人翁的感觉，所有这些都能使改变平滑过渡。帅呆了！

他们可以有像你一样的领航员

当然，你无法仅仅通过讲就可以指出改变所带来的所有问题，你应该早有准备。在大家放弃你的想法之前你如何把这个问题指出来并加以完善？通过试航，试下水。

试航就是为改变尝试一次试验，成立一或两个小组以新的方式工作。他们的特殊任务就是代表整个团队进行探索，理想的情况下，这些小组是由新员工组成的——一群乐于尝试新事物，灵活机动的人。另外，你还应该在这个试航队伍中放进两个思路开明、善于发现问题的人——他们会发现关键问题，并成为你强有力的支持者。

在你宣称要实施改变之后，你就可以把这些试航者“踢”开了，但是你还是要为团队里剩余的人做启动工作。当这些试航者发现了问题并取消试验后，其余的人就有时间克服阻挠、发怒及讨价还价的阶段了。记住，如同所有优秀的探险队一样，试航员应该将他们的发现记录归档而让其他人参考。试航员其他三个优点：

- **经验**。当这些试航员完成了他们的先期工作后，在这次改变中，你的团队就拥有了一帮经验丰富的专家。现在，你就不用一一答复所有的问题，而且这次改变看起来也不像是你一个人的主意了。
- **所有权**。事实上，当试航团队中的人员经历了那些问题并获得了成功后，他们就会越发觉得他们是这次改变的主人。这次改变变得就像他们的主意一样，如同你的，这恰恰正是你所想要的。记住，要尽量表现出你的支持与赞扬，这样将对你、你的想法及你的团队皆有裨益。
- **修正**。在整个团队进入沮丧阶段时，试航员就可以报以成功的经验。这种快速的成功经验对于你的团队克服沮丧阶段，进入接受改变阶段是相当重要的。

准备好起跳了吗

当团队里的人不再有反对与不同意见的时候，当试航员可以快速解决问题的时候，你就可以开始全面实施了。由于你已有准备，这不再是大难题。大多数障碍及问题已经解决，你也有了试航团队作为开山指路者。

然而，不要期望事事俱全，因为由于影响范围的扩大，很多问题将浮现，特别是在开始的头几个星期。当更多的人参与到这次改变中来，新形势、新情况接踵而来，让你的团队了解这些问

题很可能会发生，那么你及其他人才能提供帮助。再次强调，广泛、经常的沟通——无论是与人面谈还是发 E-mail，都是相当重要的。

> **作者注：**当然，你的改变可能会由于种种原因而失败，这没什么。我宁愿看到人们尝试了又失败而不是什么也不敢尝试什么也没得到。这同样从另一方面说明试航员是多么重要——他们让你早早就知道失败。

往前走，别忘了来时路

当你将改变付诸实施，而人们翘首以盼改变带来的好处时，你很可能早想着庆祝了。这样就太不成熟、太短视了。当你一获得支持时，随之而来的就是要完成既定目标，是否要宣称胜利要看是否达到这些目标而定。事实上，你的改变计划是否能存活就看这些了。

为什么会这样？因为毕竟你提出了想法，启动了水泵，是你试了水并且成功付诸实施。大家好像已经适应了新的工作方式，而且可能正欣喜于他们新获得的成功，为什么还有人想走回头路？哈，你忘了这些支持是他们给的，他们也可以收回。你必须做出实效。

“做出实效”，我的意思就是使这种改变为你的支持者带来真正的价值。他们忘了你的改变所带来的影响之时，也正是他们理所当然让新来者走回老路之日。我打赌你肯定知道在微软及其他地方发生过这样的事。

这就是为什么，设定度量标准、度量方法及目标是非常重要的。整个团队及主导者会因此与时俱进，无往不胜。当然，当你达到甚至超出了你的目标，你、你的团队及主导者就可以额手相庆。

没什么是可以一劳永逸的

改变是生活的一部分。能控制好、掌握好它的人就是成功的人。这并不容易，但是话说回来，它也不那么难，谁都可以办得到，你也可以。善于掌控改变将使你与你的团队共成长，要善于准备、沟通并使过程透明。当你未回过神时，改变已让你获益良多。

2009 年 6 月 1 日：“奖赏，很难”

一年一度的工程师颁奖典礼就要在本周的微软工程大会上举行了，一年一度的评审大会也将随即召开。这些都是奖赏颇具影响力的工作的大好时机，糟糕的是，大多数管理者无知得不行，他们不知道怎么奖赏他们的员工或不知道他们为什么要这么做。

如果你是一名管理者，你可能很期待这样的机会以清理这些糟糕的管理者。猜猜会怎样？要清理的就是你。你不知道如何适当地给予奖赏，你不知道你为什么要这样做。“但是我在赏识我的员工方面尽心尽力，”一位没脑的管理者哭诉道：“我总是以我的团队为荣——我还曾为此剃光了头。”让我把这个管理者称为“鼓噪者”。

鼓噪者以为奖赏就是与信念和士气有关。当然，鼓噪者作为一个团队鼓动者还是有些好处的，但是如果这就是他奖赏的深层次作用，那么结果也就是一片鼓噪，混乱。

为达目的，不择手段

鼓噪者的奖赏目的落空是因为他的奖赏不过是一种督促。督促促成行动，行动得来成果，成果是微软的终极目标。得当的督促在于得到所期望的成果（以及处理预期以外的事物）。

如果你对你想要的成果没有经过深思熟虑，褒奖很容易就会导致一种有害的行为。比如，鼓噪者在他的团队遭遇艰难的里程碑时，他削发明志。为了达到那个里程碑，团队成员就在质量标准上偷工减料，并把结构完善的事一拖再拖，这些问题逐渐累积成“Bug 债务”，拖欠稳定性的债，并降低了成品的质量。更重要的是，鼓噪者的团队成员由此明白鼓噪者对偷工减料予以奖励，却不看重质量问题。

> **作者注：**通过快速开发消除细节方面的顾虑是原型开发与新想法才需要的，这样可以赢得你要的问题改进的空间。

以结果论断方式方法

“没错，但是我们达到了里程碑的要求。我们促成了团队的一致性，这就够了。”鼓噪者说道。到达终点就能评判所用方法得当吗？不，终点不是你们这些鼠目寸光的人理解的那样。

这里的终点对于鼓噪者来说是一个步调一致的团队按一个紧张的日程安排完成任务，然而，终点的另一种含义是圆满完成——解决不断积累的 Bug 债务。如果终点的定义就是一个团结的团队在一个紧张的日程安排中完成任务而没有一点 Bug 债务的话，那这个团队才可以保留这些他们所用的方式方法。

鼓噪者开始说风凉话了，他说：“这么说，如果一个团队去抢银行，拿了钱付给卖主当做还 Bug 债务，这样就行了？”不，不是的，鼓噪者还设了另一个终点——牢狱时光。把终点定义成为一个团结的团队合法地、负责任地、以零 Bug 债务地在紧张的日程安排下完成任务，那么就不会有什么问题了。

要准确定义并表达你所要的终点并不容易。任由鼓噪者的意思来，然后得出一些随机性的成果，也不管会有什么意想不到的结果，这样做要简单得多。但是，有一个清晰的预期，并对你真正想要的结果有充分的认知将有大大的好处：

- 不再需要事必躬亲。
- 发挥团队的创新能力。
- 你将得到期望的结果。
- 你就有更多的时间把重点放在开发上面，并对你的团队做出的出色工作予以褒奖。

大干快上，是时候了

你是否在目标完成后才给团队员工以奖励？在完成一个目标后马上就给予奖赏是非常有效的。但是在达成最终成果的过程中，对中间过程中所做出的成绩也应予以奖赏。

比如，你的团队众志成城，以法定程序，负责任地按紧张的日程安排，而且不带一点 Bug 负责地完成任务的这种过程中，你的项目主管完成了非常出色的设计评审工作，你应该马上对他的努力给予赞赏，要这样说：“今天的设计评审做得实在是太棒了，你们认真听取了大家的意见而

没有半点批评，我很高兴；你对反馈意见作了总结并改进了设计，及早对这些错误进行了修正，消除了误解，这将为我们节省大把的时间，这样我们就可以按既定的紧张日程安排开展工作，并且不带有一点 Bug 债务了。我非常高兴。”

注意，得当的赞赏的关键是：

- **马上**。当天就说，或者最好，当时就说。
- **对你满意的要准确表述**。而不是泛泛而谈“干得好”。
- **要与预期结果紧密相关**。着重强调大家工作的真实价值。

可能，在向你的最终目标行进过程中，你没有时间或机会对其中有积极意义的每一步都一一予以赞赏，但是你应该睁大眼睛盯着，尽你所能，奖励即使一些小小的成就。你的团队会喜欢这么干的，他们也会更乐于接受你对他们的期望，他们也就更有可能达成你既定的目标。

让我们庆祝吧

鼓噪者在想，“天天施以小恩小惠很好，但是这样不会削弱工程完成时庆祝大会的作用吗？我是不是不该剃光我的头？”人靠衣装马靠鞍，庆祝大会是针对重要成果的，它同每天的点滴赞赏是一样重要的。然而，因为庆祝大会是长时间努力后的巅峰时刻，它不是只言片语的赞赏或剃个头那么容易达到的。

首先，赞赏是针对某一准确定义的预期结果而做出的，不是一些模棱两可的“干得好”或“某某人说某某事干得很漂亮”。比如，不要把一项奖励颁给“团队合作”，而是将其颁给“跨部门、跨区域团队合作，它促成了一个完整的客户案例的完成。”

一旦你为奖赏定义了一个准确的标准，就可以着手准备庆祝大会了。以下是上一次成功的经验：

- **人性化**。如果是你定下了目标，那这种赞赏就应该来自于你本人。就要以一种人性化的方式表达出你既定目标的内涵，以及能使个人或团队拥护这种内涵的途径。
- **公开化**。公开是很重要的，公开体现出了工作的意义，建立起社群关系，传播了理念，使大家对目标达成共识。
- **持久化**。相应的奖励应当是实实在在的——厚重、醒目、有形。剃头的举动符合这样的标准，丰厚的物质奖励也一样（像奥斯卡），并在公告板的海报写上精英们的名字。

感谢电影艺术与科学学院

最后一个要考虑的问题是，“你是否奖赏了这个团队或个人？”当你要为整个团队举行一次庆祝大会或颁发一项别出心裁的纪念物（如在团队办公室贴张画报）时，对于这种具有具体期望结果的奖赏，其获奖者只能是少数几个人，通常，这样的人数不超过 5 个，大于这个数目，则奖赏的效果将大打折扣。

为什么多于 5 个人，这种奖赏效果就会被削弱？因为当你只对整个大型团队进行奖赏时，那必然是，具体到个人他们是不会理解你这样做的真正目的的，你殚心竭虑不过就充当了一个司机的角色，不该是这样的。是谁促成了行动？是谁促成了成果？是那些敢作敢当、身先士卒的人，是那些你特别指定的、鼓舞他人奋勇向前的人。

作者注：我好几次都参与了公司内部卓越工程师（Engineering Excellence，EE）的颁奖活动，这些活动对显著改善我们产品及服务设计的方法进行了表彰。评判标准是围绕是否能通过实证表明这些设计方法获得了改善，以及这些设计方法是否适用于他人（能给业务及客户带来好处，并且他们可接受）而定的。获奖的人是那些先有个最初想法，然后付诸实践，最后获得大家认同的人。会有个专门的庆典活动（通常由比尔·盖茨主持），奖励也是非常丰厚、显眼、有形的，并且将使其铭刻终生的。

在过去的几年，卓越工程师颁奖活动已经对我们取得的计算机工程的突破给予了奖赏与鼓励，这些突破带来了更安全更可靠的产品、更满意的客户反馈以及更广泛的语言支持。相关领域的评论家正注意到了这种变化，可以想象，当在更广泛的领域获得更多的突破时，每个人都会注意到这种变化的。

说个花边趣事，几年前，我们把一项卓越工程师奖项授予了一个工具，这个工具汇集了大量人员的心血，被广泛地采用，其自动化的处理大大提升了我们软件的品质。遗憾的是，这个工具本身未经成熟的设计，从而使生产力遭受损害，并对颁奖计划造成不良影响。这就是一个需要对你所要成果标准进行严格定义的典型案例。

好吧，开始评审

现在你知道如何奖赏你的员工了吧？你也知道为什么你应该以及你如何在年度评审会上进行表彰了吧？

- 讲清楚，你员工的哪些成果是你喜欢的（或不喜欢的），以及他们应该如何达成这样的成果。
- 要讲明白小成果与大成就的区别，并把它们结合在一起。
- 对既定目标设定一个严格的标准，这样就可以促使每个个体全身心地投入项目开发中，并为你的业务及客户带来非常优秀的成果。

明确的期望与常态的、表述清晰的反馈紧密相关，这对于产出优秀的产品非常重要，对一个欢快、和睦的团队也很重要。奖赏需要慎思笃行，但这并不是什么难事。远离鼓噪者，你的头发就会平安无事。

2009 年 10 月 1 日："招人总是后悔"

想找一位完美无瑕的应聘者来填补空缺的岗位吗？很好，这意味着你永远不会从我这里抢走一个优秀的应聘者了。这我喜欢，一个死脑筋的傻瓜给了我一个放松的机会，不用天天想着如何竞争了。你不会招聘到完美无缺的应聘者，但请你继续努力，或许半年后你的职位空缺名额就是我的了。

我这样说并不是说完美与优秀不能共存，愚钝的人事经理不知道如何去塑造完美的应聘者——他们只是等着他们出现。就像是一段浪漫的期待，等候着精神伴侣的出现，或像是买房者在寻找他们的"梦幻之屋。"猜猜会怎样？梦不是现实。

如果有人跟你约会了一两天之后就对你说你正是他梦里所寻的那个人——赶快跑吧，他们不

是在跟你约会，他们是在跟他们的梦约会。不用多久，他们就会发现你俩不合适。这样的事同样会在人事经理在寻找他的梦幻应聘者时发生。

赶紧招个人，不要老做白日梦

人事经理和他的团队总对新雇员应该是什么样充满期待，这是很自然的事。你花了大量的时间来查阅简历，打电话，了解情况，然后再面试。不能承受之重，你会担心这样做不知道会是什么样的结果。

遗憾的是，追求完美潜藏着三种危险：

- 你可能一个应聘者都招不到，即使他们符合你的要求，但是他们没有符合你虚构的样子。
- 你可能推迟给一个优秀的应聘者发意向书，因为他们缺乏一种你想象中的品质，这种品质是你要优先考虑的。
- 你可能后悔聘用了一名应聘者，因为不久后他变得并非你想象的那样完美，你反而剥夺了他们成功的机会。

是的，你不会聘用不符合我们所设最高标准的雇员，但是却因为你总是不停的幻想而失去一次聘用优秀员工的机会是很可怕的。我们来深入讨论一下。

> **作者注：**在本章前面“面试流程之外”中我对了解应聘者情况及面试有更多的讨论。

就是你了

如果你对你要聘用的人有了先入为主的想象（更糟糕的是，这个人跟你一样），那么当然你就会在潜意识里以这种想象的模板为基础排除掉应征者。你可能会找到符合你想象模样的应聘者，但是你会失去一大群符合你需求的应聘者。如果一百次中才有一次招聘成功（事实也差不多），你越迟对这一百个优秀的应聘者进行面试，你也就越迟才会聘用到一个你要的人。另外，减少面试条件会带来多样性，这样的团队就有更好的平衡度，会出更好的产品及服务，也会有更好的客户体验。

如果你迟迟不给一个优秀的应聘者意向书，想着下一个或许会更理想，你很可能两手空空。另一个团队会想要你手上的那位优秀应聘者，而你的下一个应聘者总会有瑕疵，人说：“一只鸟在手比在树林里的两只鸟好得多。”除非，你的下一个应聘者要好上一倍，否则，在这位优秀的应聘者成为你的悔恨之前，马上寄出意向书。

说到后悔，你是否在买了某个东西后很快就后悔了呢？购物者的这种悔意同样也困扰着管理者们。你幻想着会有个人可能会适合某个重要岗位，并找到了这个人，然后就后悔了，因为这个人并不是你所想的那样。别动肝火，别当回事。只有电影导演才会编出他们想象的剧情，而当开拍时，这种想象就没了。你已经雇到一位优秀的员工，现在就给他们机会吧。

你的需求是什么

你如何才能知道哪些素质是你的空缺岗位所需要的呢？如何分辨你潜意识里的偏见？首先考虑你想要这个岗位给你带来哪些“必须有”的成果，再想想你要多久出这些成果。那么就坚持等待那些已经完成过类似任务的应聘者出现。你越是需要更快地出这些成果，则对这些应聘者之前

完成过类似工作的要求要更严格。

比如，你需要一个可以开发并发布服务的人，如果你需要他们在9个月内完成这项工作，就雇一些曾经写过类似代码并发布过类似产品的人，这可以是一个游戏或一套理论的算法实现。你所需要的就是这种开发并完成产品的能力，这个人可以是来自各种领域的任一个人——选择很多。

但是，如果你在SQL Server组，需要一个项目主管马上接手开工，这种情况下，一个刚刚成为项目主管的人也可以。可能并不需要他现在就具备SQL Server专业技能，所以这种专业技能并不是必需的，你也不应该有这种坚持。你会很悔恨要了一个具备SQL Server专业技能但很糟糕的项目主管，而放走了一个日后可以积累SQL经验却非常出色的工程师。

关键是把重点放在“必须有”的结果上，以及完成这些成果的必备技能。再找那些可以证明这些必备技能的，可以做出类似成果的人。如果你的需求很紧急，你的要求可以更严格些，但是别唬自己。吹毛求疵可能会浪费人才，而如果你给他们以支持，给他们以机会，他们会在以后好些年给你带来非常可观的成果。

你只要知道他很有原则就行了

有一种素质可能你在应聘者的简历上无法找到，那就是应聘者的原则性。原则在外表、技能或过往成果上看不出来，因此，当你在虚构一个你理想的应聘者时，你往往会忽略原则，而这也是另一个不要浪费时间想象你的理想应聘者的原因。

什么样的原则对于你及你的团队是很重要的呢？正直？坦诚？负责任？大公无私？独立性？多想想。它们跟专业技能与之前做过什么项目一样重要，甚至有些人认为原则更重要。技术可以学，大家也可以共同努力完成任务，而原则却很难符合要求或改变。

不清楚哪些原则对你及你的团队很重要？好好想想——马上。要让你的团队知道，明确原则本身就是一种原则。对于我来说，这是最重要的，也是我在与应聘者交谈时我首先要找的特质。

我想谈谈人生

生活不是写小说，如此种种都是你想象的那样，它是充满着纠结与抉择，并满心期待最终皆大欢喜。招聘组建团队也一样。

你无法知道谁在什么时候看你的空缺岗位，所以不要瞎猜，相反，放开肚量接受可能的选择。记住，你需要的是一个百花齐放、目标一致的团队。

所以你不希望产品发布时间是不确定的，也不希望产品质量是不确定的，你希望的是你的团队思维活跃，互助合作，他们是创新与改进的源泉。你的团队人员越具多样性，而不是一成不变，则你的团队中间就会越乐于相互学习——这可以在更具新意的设计及更好的客户体验中反映出来。

所以，想想你的团队需要什么，不要总设想着谁可能会适合你的团队。放开肚量，把意向书给优秀的应聘者寄过去，除非你知道下一个应聘者会双倍优于他，秉持你的原则，千万不要吃后悔药。

每个人都报怨人力不够——越快为你的空缺岗位找到人选，那你的团队就会越开心，战斗力也就越强。

作者注：这篇专栏很受欢迎，它引起了招聘专员及公司多元化部门的特别关注，他们很喜欢这些建议。招聘是很耗时间的，而且会让人很受挫，所以并不是我喜欢做这样的事。然而，对于我们公司或我们这行业有着这么多的各色人才，我是既惊奇又欣喜。

2009 年 11 月 1 日："管理馊了"

对你来说好的东西，对于你的团队来说就不一定？这很显然，不是吗？你可以称之为本地最优化与全局最优化之争。用一句经典的哲理名言来说就是："少数服从多数。"或者你从实习生的荒诞想法与你上司的荒诞想法之间就可以一见端倪。

举个例子，发挥个体自主性其好处不言而喻，而一成不变只会让人变傻。但是当一个人在管理一个大型的企业时，不确定性会造成大灾难。有很多过来的管理者明白这个道理，但也有很多管理者脑子不灵活，会经常闹些让人崩溃的事，或简单地说，他们尽出馊主意。

我鄙视出馊主意的管理者，他们认为他们反应迅速，头脑灵活，实则现实中他们是团队破败的始作俑者。如果你是一名管理人员，对你的团队说："嗨，我有个好主意。"然后他们回头看了你一下，好似说："你的意思是把你绑在拖拉机上，然后从桥上掉下来吗？"那你出的就是馊主意。

还没那么糟

成为一个尽出馊主意的管理者到底会有多糟糕呢？顽固与一成不变是不是很没趣？答案就是：布朗运动——液体分子随机撞击微尘粒子形成的运动。用微粒子的布朗运动表述飘忽不定是最恰当不过了。

设想一个大型的工程师团队，总经理总是对他们随机性地乱指挥，那摆在你眼前的就是飘忽不定的布朗运动，要让这些工程师们像一个整体那样来完成任务是想也不要想了。

相反，如果这个团队持之以恒，团结一心，朝着共同的目标前行，他们就可以在工作上取得重大进展。因此，一个成功的管理者应该设定一个明确的目标，带领团队朝同一个目标前行，然后不停督促，直到达到这个目标。在团队走上正轨之前，渐进地对团队进行督促，自然就会带领团队走向正途。只是在情况危急的时候，才需要做出重大改变。

我这里说的是战略上的方向，而不是战术上的，战术就要灵活机动。关键是战术上的决定要整体上与战略保持一致，如果你战略上总是变来变去就完了。

作者注："但是我不该让员工们太随性了吧？"是的，是不该，但是这是一种风险管理。并不是说让你的员工肆意妄为，我的意思是要让他们有危机感，而不是让他们惊恐不安。居安思危，才能精益求精。就如我在第 1 章（特别是"按计划行事"）中说的，在情况有变的时候，你就要控制好这种风险。如果遇到一个不牢靠的依赖方，风险就来了，你可能就要考虑加入一些强悍的功能模块。如果团队里的一个关键成员因为休假离开，这也是种风险，你可能就得考虑放弃一个依赖方或是一项重要的功能模块。无论如何，居安思危总是好的，尽出馊主意就愚蠢了点儿。

我是那种看起来出馊主意的人吗，还是顽固不化

如果团队是由一些尽出馊主意、想法飘忽不定的管理者带领的，工作进展又很缓慢，那为什么有这么多出馊主意的管理者存在呢？有三个原因：

- **出些馊主意看起来好像反应很迅速**　这些管理者不会按既定路线前进，他们对外界干扰反应很迅速，这给人感觉他们的管理方法是非常敏捷的，这其实是假象。
- **出馊主意就意味着永远不用完成什么事情**　因为这些管理者的团队方向总是变来变去，所以自我实现的借口成为了一无所成的理由（“不是我说，这样的人大有人在!”）。如果这些管理者不出馊主意，他们就有承担责任的风险。他们必须要费尽心思设定一个可行性的目标，为团队指明路径，并完成这个目标。
- **出馊主意让你有存在感**　一成不变会有什么意思呢？你整天都忙些什么呢？如果总是不厌其烦地重复讲着同一件事，这对你的团队又有什么意义呢？煲电话粥是很累人的。这是很可怕的，你可能要言行一致，干点实事。

当然，因为上层管理的馊主意才滋生了下层管理的馊主意，这是无法逃避的。只是人们不知道一个持续不变的管理思路是多么重要，直到他们经历过了之后他们才明白。

大政方针

当管理者不再乱出馊主意而是始终贯彻如一时，效果就大不相同了。首先，整个团队开始出成绩，管理者不再是制造危机而是预告防止危机出现，而每个团队成员也明白他们每天应该做些什么，也知道他们必须这么做，因为明天还得这么做。

所有这些让大家斗志昂扬，翘首企盼。一开始，整个团队团结一致可能会有些费力，但是很快管理者的工作就会变得简单得多，你不用再一而再再而三地改变你的思路。一旦整个团队精诚团结，剩下要解决的问题就没几个了，一切顺风顺水，成绩很快就会出来。

那留给管理者的挑战是什么呢？首要的是为团队谋划一个蓝图，你要充分透彻地理解这个蓝图，再不断向这个团队重复你的构想。一旦你对未来做好了谋划，当团队成员或外部因素发生变化时，你的挑战就是要高屋建瓴，把持好方向。你的任务就是发现问题，做出相应调整，以防扰乱大局，从而打乱你们的进程。

这并不容易。你必须笃守信念，要对你的战略蓝图的深谋大略充满信心，这是你坚守个人理念并教授他人的一种秉持。这些理念让人们知道你期待的是什么，你是怎么做出决策的，这样也有助于让他们的想法与你的保持一致，从而让整个团队成为你坚强的后盾。

作者注：对高层管理的“诏命”我们有时是很难分清是非的，它们是否准确地体现了客户或业务的变动，以致打乱你团队的进程是值得的，或者这些“诏命”就是捣乱，你应该予以排解，从而不至于打扰到你的团队呢？最好的方式就是问问你公司里的管理者或你信得过的导师。问话要直接，找出这些变动的背景及其深意，即使这些变动很无趣，但可能就是必须做的事；即使这些变动很诱人，但或许你跟着做就错了。

做出正确的选择是你应该掌控的很重要的技能，如果你希望你的事业长长久久，兴旺发达的话。了解这些变动的背景，弄明白你要应付的人是谁，这样做会有多艰难，以及你得失会有多少，然后做出一个正式的决定，是否这样的选择是值得干的。准确的判别力将使你无往不胜。

哦，噪音！噪音！噪音！噪音

“但是，情况不一样了！”出馊主意的管理者提出抗议，“你的团队难道就应该对过时的计划盲目追随吗？聪明的竞争对手将使我们一败涂地。”没错，情况是变了，而计划、架构及设计也要调整，观念与行动也要及时跟进，才能使我们的工作不断前进。但是如果你现今的目标是建立一种大众计算机体验的话，不管是内部或是外部的变化，都不能让你转而去生产拖拉机的轮胎。

优秀的管理者都知道纷杂多变的团队结构、竞争环境、市场变动以及技术变更等都能促使计划及设计变得更周密更精良。然而，这些变更很少会引起大政方针的彻底变更。优秀的管理者对改变是欣然接受的，都会把它们当成是团队的良师益友，从而同仇敌忾，边前进边适应，朝着共同的目标奋勇前进。

> **作者注：**优秀的管理者会直接面对这些骚扰，防止他们的团队受干扰。有些管理者在员工会议中会安排一部分“谣言碾碎机”时间来讨论最近传出的流言蜚语。要把问题放在台面上，尽早戳穿流言，让团队全神贯注，一心一意工作。

小人物才出大成绩

坚守会让管理者看起来像傻瓜，但正是这些傻瓜才是一个团队真正需要的——这样的傻瓜才出成果。虽然坚守听起来很容易，但那需要勇气，要有坚守自己立场的勇气，要有说到做到的勇气，也要有为自己做出的决定负责的勇气。

我这里讲的就是承诺，就是正直，是开天辟地的眼界而不是随波逐流的盲目。

我们的大计应该从客户的需求及其愿望出发，要的是集体的思想结晶，但是最终这样的决策是属于我们作为个体的领导者，我们的成功仰仗于他们将这个决策说出来的勇气，并矢志不渝，一路追寻。你有这样的勇气吗？

2010 年 1 月 1 日：“一对一与多对多”

是否与你的上司一对一面谈让你血脉贲张或是坐立不安？是否集体活动都是一大群的同事扎堆，或只是一些不负责任的人及造谣者来参加？那些适度的你能接受的一对一的面谈才是你需要的，而感情交流这样的活动要么是很少有，要么很多余，或是两者都是。

把时间浪费在一对一面谈及集体活动上都是不可原谅的。它们可能用掉你一个星期或一个月中最有价值的时间，而把时间、金钱浪费在了繁复、无用与自我感伤中，又扰乱了工作节奏，这样的过程是种犯罪。（“我不该享受这样的时光吗？”）对此，你能做些什么呢？是的，你能。让你的上司读下这篇专栏，因为这全是你上司的错。我要跟你的上司谈谈吗？没问题。HI，头儿，你忒恶心了。你做的这些一对一面谈及集体活动太不合时宜，这都是你的错，只有你才能纠正这样的错误。不过还好，还有希望，纠正一对一面谈及集体活动的错误还是简单的，不费多少事，可以把省下来的时间投入到开发一个最出色的工具中去，以提高团队的战斗力。

意味深长

先让我们从一对一面谈与集体活动的初衷开始谈，一对一面会谈不是为了谈些工作进度的，工作进度是要在工作进展会议上谈的，集体活动也并不是为了鼓噪人们自命非凡的。只有通过你的信任、你的指引以及你对员工的个人认知来肯定你的员工，这样他们才会相信自己的价值所在。

> **作者注：**在一对一面谈中你可以谈谈工程进展如何，但只能是你的员工乐意的情况下或你们还有时间剩余的时候。如果你确实需要对某个员工工作的最新进展进行更细致的了解，你可以在你员工的工作日程里安排你俩的时间，毕竟，你是头儿。

- **一对一面谈的真正目的是什么？**为了你与你的员工之间建立起坚实的信任关系，要从更人性化的角度了解他们。管理就是委托，委托就是信任，信任来自于互信与理解，而互信与理解又来自于人们相互间的尊重与情感的培养。同时也需要正直，无论什么这都是必需的。
- **集体活动的真正目的是什么？**是要打破人与人之间、团队与团队之间的藩篱，使人与人之间充满人文关怀。在一起共事的人，不可避免地会产生冲突，商务环境会使人变得冷漠，这使人与人之间的冲突更甚。你应该听过“他们不在乎”或“他就是个白痴”这样的话，要让人们远离商业环境以全新的方式进行交流，你们要以诚相待，营造一种和睦的关系，增进互助与理解。

现在你知道一对一面谈与集体活动的目的了吧。接下来，我们来谈谈如何使这些方式更有效，先从一对一面谈开始。

就你和我

一对一面谈可以在任何地方进行，花个30分钟到2个小时的时间。我一般每个星期要留出1个小时的时间与项目总监交流，每个月要抽出30分钟的时间与我的每个“跳级生（skip-level）”谈谈。在你没搞晕之前，自己算一下，每月为每个“跳级生”花的时间相当于每个星期与项目总监交流时间的一半。没办法，你必须两样都要做。

为什么要这么频繁地进行一对一面谈？因为这非常需要。通常，你每个月都会不可避免地遇上意见相左的时候，如果你一星期内没有进行一对一面谈，那至少每两个星期要跟这个人面谈。然而，两星期一次的面谈虽然不如一个星期一次的面谈好，但是也比一个月面谈一次强，一个月面谈一次对建立你俩的信任关系来说次数就太少了。这样的问题在“跳级生”身上同样适用。

你们该谈些什么？你的员工想谈什么就谈什么。一对一面谈属于员工。什么时候合适？随你的便。你是头儿，你可以随时踱进任何一个人的办公室随意谈。一对一面谈是员工们发表个人诉求的时间。

> **作者注：**是的，浪费一对一面谈的时间不全是管理者的错。员工也确实要认真考虑他们自己的诉求是什么，并主动与管理者交谈，管理者也要欢迎这样的交流，而不要把重点放在开次大会上。

“但是如果有员工总说些不着边际的话题该怎么办？这是否是在浪费时间？”不，不会。记住一对一面谈的目的就是与你的员工建立坚实的信任关系，从个体上了解他们。什么样的方式能更好地了解一个人并建立信任，而不是让他谈些不着边际的话呢？看看你员工的办公室，在墙上，在桌上到处都有可谈之处，退一步讲，你可以谈谈你们的周末怎么过。

> **作者注：**当有些员工很乐于谈他们的周末及个人兴趣，而有些则有些沉默寡言时，则你必须尊重而不是窥探他们的个人隐私，这就是为什么问一些他们桌上或墙上放着的照片会安全一些。这些都是一些可以公开的话题，如果你想谈谈周末的安排，一定要先简单提一下你自己的安排，再看看你的员工是否有兴致跟你交流这样的话题。

自由飞翔

除了可以建立感情外，为什么谈些个人兴趣很重要？有一个事实可以说明问题。

之前我有一个下属，他很喜欢玩直升机航模，每年，他与他的追随者们会在全国航模大赛上进行比赛，所以，每当一对一面谈的时候我们会经常聊聊这些话题。

有一年，将近周末，也是全国航模比赛的时候，我们将要发布产品，问题就来了。这名项目主管也答应不去参加这次航模，不过所有人都已经准备好顶替他的工作了。最终，他去了，很享受那次比赛，而我们的产品也成功发布了。我们事先就知道了这次冲突，并预先做了准备，因为我知道全国航模比赛对他有多重要。你也知道，这位主管会多么感激不尽。

想想如果这位工程师在几个月之前没有跟我谈起这次全国航模比赛的事，那情况会怎么样，谁会愿意就这样让他走掉仅仅是因为一次直升机航模比赛？这位工程师很可能就取消了他的活动，他会明白应该担起自己的责任，但是他会很失落，很可能有怨气。相对于这位主管的感激不尽，态度会多么不同。

这个例子还只是参加一次全国比赛，还不算什么。想想，如果是一些痛苦的事情呢，如家人病了，离婚了或其他悲惨的事。这些把我们当上司看的员工们，他们都是常人，在工作的里里外外都会有影响他们工作的事情发生。不顾他们的生活乐趣及问题会让你及你的团队陷于危难。对他们宽容以待，将建立起忠诚与信任。

我该怎么做

“但什么时候你会讨论职业发展或工作的事呢？”一对一面谈是进行指导与职业发展的大好时机。通常，你不必主动提出跟员工们谈论这些，他们通常也愿意花时间讲讲如何提高他们的技术与职位升迁问题。不过，你应该想尽办法找个机会强调一下重要的决定、战略方针、如何应对环境变化、沟通的方式及其他各种可以提高员工个人能力的事情。

如果在特殊的情况下，一个月中你没有跟你的项目总监谈起职业发展或项目协议的事，你就要自己主动提出来。在有些时候这是非常重要的，没有人愿意突然间听到别人的意见而措手不及，而这些意见本该早就提出来的，这同样也意味着在一个员工求助你之前你必须事先经过深思熟虑，准备好给他们提供意见建议。

要特别注意的是，绩效问题不能等着“指导时间”的到来才去处理，必须马上对它们进行处理，并通过一对一的面谈强调相关事宜。按问题的严重情况不同，你要用文档记下你的意见建议，

以保证你的意见没有纰漏以及记清楚什么时候这问题引起了你的重视。想对这一点进一步深入的研究，请参阅本章前面的部分“最难做的工作——绩效不佳者”

我们正玩得欢呢，不是吗

一对一面谈说得够多了——集体活动又是怎么样的呢？记住，集体活动的目的是通过增进人与人或团队与团队之间的关怀来打破他们之间的藩篱，那这样的活动需要多频繁呢？问得好，要让大家习惯成自然这需要多长时间呢？我想你大概得一个月搞一次这样的活动。

一个月一次的集体活动能使人们互尊互爱。当然，你可以安排一个月举行一次这样针对整个团队的集体活动，接着再举办一次针对更大组织的，再接着每一个季度为整个公司搞一次这样的活动。直到包括你在内，一个月至少有一次大家都以这样和睦的方式共处。

“但是这样是不是太费钱了？”不，因为最好的集体活动通常用不了多少钱。理想情况下，这样的活动就像一种调节剂——没有谁可以鹤立鸡群，也不用那么专业，就像保龄球。以下是既省钱又方便的一些消遣活动：

- 飞碟高尔夫（不用钱）。
- 拼图游戏——Trivial Pursuit、Apples to Apples 以及 Scene It 都是很不错的游戏。
- 扑克牌，特别适合 4 个以上的人玩。
- 边吃外卖边看 DVD。
- 寻宝游戏（geocaching. org）。
- 台球或乒乓球。
- 徒步旅行或其他户外运动。

记住，对于集体活动，多开展省钱的活动总比寥寥几次费钱的活动好。大型的团队可以更多地开展类似保龄球、旋转球或冰壶的活动。关键是要打破平常工作生活的老套路，一起享受一下五彩缤纷的生活。

> **作者注：**最好不要在工作的关键时期开展集体活动，即使这样的活动较往常而言更势在必行，因为这时候大家都有压力，没心情参加这些放松活动，他们正心烦着，放不开手脚。很多团体成员根本就不想参加这样的活动，最好的办法是，要有一个周全的计划及优先级安排以减少这种关键时期的次数及长度。

你要相信我

如果说一对一面谈与集体活动有什么相同之处，那就是花些时间以人性化的方式沟通人与人之间的关系。这样就可以建立起彼此间的理解、友情及信任。良好的关系有利于人与人之间的沟通、合作及团队凝聚力。

这样做的意义并不是说当大家相识的时候就变得更能说会写了，其意义在于能更好地体谅别人犯的错，破除相互间的误解。最好先了解事情的来龙去脉，先要了解别人的真正意思再进行深入交流，当人们秉着诚信与理解一起共事，我们就会成为同舟共济、无往不胜的整体，而不是各自为政。

2010年7月1日："文化冲击"

文化是管理的心魔——这是一个无法掌控的恶魔，一条无法逾越的鸿沟。如果你想看到管理者们如婴儿般号啕大哭的样子，那就让他们解决一些根植于企业文化的顽疾。

当管理者说："这是企业文化的问题。"他们的意思就是："我无能为力。"企业文化成了管理者维持自己光辉形象，逃避责任的最好借口。这种借口不仅仅是差强人意的——而且是幼稚的，不可饶恕的。

文化的概念已与我们的生活水乳交融，还能指望人们为此能改变些什么呢？但是企业文化不会无中生有，它是由企业的创立者所浇注的，并随着这些创立者的改变而改变。因此，你可以影响企业文化，它也可以改变。你所需要知道的就是怎么做，而不是畏畏缩缩。

文化不过尔尔

有数不尽的书刊、杂志、博客等都对如何建立并延续企业文化做过讨论。归根结底的意思就是先有个设想，再模式化，最后毁誉听之于人。

你的父母、同事、邻居及朋友都希望你按着既定的方式生活工作，也为你铺设好了一种模式，你如果没有遵照执行，他们就会帮你指正或是远离你，当你做了个"好榜样"的时候，他们就会很高兴，以此为荣，这就是文化常青的原因。

你可以改变这种文化吗？可以——提出期望，再设定个模式，然后要么引以为荣要么遭人唾弃。没错，就这么简单。

你是否注意到当领导人改变的时候团队或组织的文化是如何改变的？顺理成章——新的领导人期望什么，他奉行的是什么模式，最后种瓜得瓜，种豆得豆。

> **作者注：**新晋领导人所期望的愿景、奉行的模式及最终的结果并不一定是他的本意，他可能说一套做一套。有句老话："听我的话，而不是学我的样。"这对于改变企业文化来说毫无可取之处。

靠边站，伙计，我是名科学家

有一套科学理论可以促成企业行为的改变吗？当然有——这称之为人类绩效技术（Human Performance Technology，HPT）。你可能会认为所有的管理者都懂得这套理论，但事实不是，他们可能更喜欢怨天尤人。

人类绩效技术的基础是B. F. Skinner的理论模型——操作性条件反射。首先，这套理论认为人们随意性的行为由刺激、反应及强化组成。你受到了一次刺激，如你的上司叫你写代码，你的反应是写代码，你就会收到一份奖励（强化），如比萨。

Thomas Gilbert在他的书《人的能力》中对刺激、反应、强化的类型进行了分析，他将这样的类型区分为环境的与人为的。

- 环境指的是企业的影响。如期望（expectation）的刺激会引起一种反应（如他人的指导及绩效考核），体现反应的是工具或过程（如创建系统），以及用奖励来加强这种反应（如

奖金或认同）。

- 人为指的是人的影响，如人们理解刺激的才识，作出反应的能力，以及想要获得奖励的动机（如果你并不饿，比萨就不是奖励了，饥饿才是动机）。

要使行为发生改变，你就要对刺激、反应及强化的环境及人为因素做出改变，即“六格法”。

	刺激	反应	强化
环境	期望	工具及过程	奖励
人为	才识	能力	动机

作者注：将 Gibert 六大论断概括为“六格法”出自于 Car Binder（http：//www. binder-riha. com/sixboxes. html）。

环境因素

期望：假设你希望你的软件开发团队更注重于单元测试，如引入测试驱动开发（TDD）。从环境的角度来看，你必须确保编写单元测试是管理层及团队的意见，这对于该为单元测试设定多少代码量是有帮助的。不过，这样还不够。

工具及过程：一个团队必须有一个很好的单元测试架构——运行简单，开发者也很方便地就可以加入测试代码，而其对测试结果的测定也很方便。当然，这也还不够。

奖励：无论有没有单元测试，如果项目经理给开发人员的“奖励”是可以快速签入代码的话，开发人员就不会再编写单元测试了。要想让他们编写单元测试，那么只有让开发人员必须编写单元测试，这样他们才可以得到赞许或奖励。

作者注：怎样的奖励最有效？一般都不是薪水。当人们的薪资水平不平等的时候，薪水的奖励确实有激励作用，但同时，最近的研究发现，当要完成一项很艰难的工作（自主性、主动性或有成就感的工作）时，薪水的奖励与人们真正期望的奖励并不对等（参见 Dan Pink 的论述，www. youtube. com/watch？v = u6XAPnuFjJc）。要提供更多的责任及机会让人们能一心工作，精益求精，创造出非凡的成果，而不是给予更多的钱。这样的奖励才是工程师的动力所在，同时，很自然地工资也会提高，职位也会晋升。

人为因素

才识：开发人员应该清楚单元测试有哪些类别，以及如何编写；他们需要清楚如何利用单元测试架构，如何对他们的工作进行测评，以及他们的目标是什么。对于这些，我们都差不多合格了。

能力：一般来说，开发人员要具有编写单元测试的能力。最好是，单元测试不要成为他们的难题。不过，你可能希望你的测试团队编写单元测试来测试它的自动化。如果你的测试团队使用的是一个自动化工具，那这个测试团队的成员可能没有写单元测试脚本的能力，或是编写外部测试代码的能力。你应该招一些具备这种能力的测试人员。

动机：最后，开发人员就要把这些期望放在心上，并获得你给予他们的认可与奖励。如果开发人员并没有将此放在心上或被打动，则他们对改变的动力还不够充足。

考虑清楚了吗

我已经对企业文化作了简单大体的概括，你可能会说这有点小题大做。但是，好好想想吧，如果“六格”里任一个因素没有达标的话，企业文化即使有改变，这种改变也不会好到哪里去。说“我知道——我们会对每个人都进行培训的。”是不够的。环境因素对人的影响比人为的影响要大得多，再多的培训也不会解决错误的期望、工具及奖励问题。然而，无知、没能力或无动于衷的工程师同样也无法解决这些问题。

再看下“六格”里的内容。在单元测试中，你会明白，我们需要一个明确清楚的管理者预期、考核标准、考核的目标、单元测试的架构、经理们奖励与认同感上的改变、团队培训以及合适的团队成员——都是能力与动机的体现。

你怎么才能事先就一一将这些考虑周到呢？很简单——你目前的定位是什么，你想完成什么，然后一一解决这些思想上的障碍。

你不能动辄就想找一个一劳永逸的方法（“我懂了——我要用那个最新的工具!”）。你必须先要对你的目标深思熟虑，对实现目标的过程深思熟虑，而不是弄个花哨图表。

如我在第2章“你怎么度量你自己?”中所说，将很多东西的价值罗列在图表上意义并不大，数字没什么意义，那些度量值也不过是摆设。我必须明白你到底要完成的是什么，对你期望的结果慎重度量。

我有些问题要问你

人类绩效技术同样也为绩效提升提供了一种清晰的架构。在提出解决方案之前，人类绩效技术要求你先了解问题原因，明白所要的结果以及度量成功的标准。当下一次有人跑到你的办公室向你建议你的团队应该采用一种新的评估方法时，请耐心听他们说完，并问他们一些问题：

- 你要完成的是什么?
- 目前的完成情况怎样?
- 你如何知道你的目标已经完成了?
- 如何才能获得管理者的支持?
- 哪些过程及工具才是这个团队所需的?
- 管理者怎样才可能认识到你的努力及工作的过程?
- 团队如何才能全面理解新的方式、工具、度量标准及目标?
- 为实现目标，相应的岗位上呆着的是否是正确的人?
- 是否整个团队的人都对此很关切?

齐活儿了

记住，如果你真想做出一番与众不同的事业，那你就得从改变企业文化入手。不论是重大的改变，如换一种服务模式，还是一些微小的变动，如采用代码复审核查表，都需要企业文化的变动。当然，大的改变比小的改变耗时要长得多，这就是为什么方法、度量标准及目标是这么重要。

改变企业文化是可行的甚至是可以测算的。当你提出了关键问题所在，注重结果，充分考虑六大因素的作用，那么一切都不在话下了。

第 10 章

微软，你会喜欢它的

本章内容：

2001 年 11 月 1 日："我是怎样懂得不再焦虑并爱上重组的"
2005 年 3 月 1 日："你的产品单元经理是个游民吗？"
2006 年 9 月 1 日："有幸成为 Windows 的主宰者"
2006 年 12 月 1 日："Google：严重的威胁还是糟糕的拼写？"
2007 年 4 月 1 日："中年危机"
2008 年 11 月 1 日："虚无主义及他的创新毒药"
2010 年 2 月 1 日："我们是功能型的吗？"

管理有 3 个方面：人员、项目和业务。我已经谈过了人员管理和项目管理。现在，来谈一谈微软是如何运转它的业务的。

像微软这样的超大企业及其资本主义，很容易被人冷嘲热讽。微软必须赚取丰厚的利润才能维持它的业务，公司不能单纯基于理想主义或者客户利益最大化来做决策。如何让业务延续必须成为一个考虑因素，否则我们很快就会没有客户。遗憾的是，理想主义者不会在一个大公司呆很长时间。所幸的是，我很注重实效，我愿意容忍一些不完美，只要它意味着我能得到更多的时间和家人在一起。当然，至于这个世界上最大的软件公司，我还是会去奚落它那些不可避免的弱点。

在这一章中，我分析并克服了运转一个成功的软件业务所面临的挑战。第一篇专栏发掘出了"重组"（Reorganization）带来的隐藏价值；第二篇专栏确定了中层管理者的适当角色；第三篇专栏接受了一项指导世界上最大的工程团队的挑战。第四篇专栏比较了微软和 Google 以及其他竞争对手的商业策略；第五篇专栏通过体验微软艰辛的成长过程，给出了公司向中年期平滑过渡的建议；第六篇专栏指出一种对于微软的创新文化的微词，并指出如何纠正这种看法；最后一篇专栏讨论了微软组织结构调整的方法及这些方法对于我们商务决策及执行的意义。

每个月我都要写一篇专栏来咆哮一下，因此大家都误以为我不喜欢微软。大家猜测，我好像是被人收买了，但如果我真的收了别人的好处的话，我会更加无所保留。冒着疏远所有微软抨击者的危险，我必须说实话：我热爱微软！我在很多地方工作过，包括学术界（RPI）、政府（JPL）、小型公司（GRAFTEK）、中型公司（SGI）和大型公司（Boeing），微软是迄今为止我呆过的最好的地方。不管它所有的过失和手段，微软有如下 3 方面最令人惊异的特质：

- 主管的人都诚挚地想使世界变得更美好——通过软件去实现。在我所了解的其他任何一家公司中，情况恰恰相反：先在某个领域取得成功，然后希望它改变世界。

- 主管的人都雇用最好的员工，然后在他们处于最底下的层次时，非常信任他们去做决策并运转业务。很自然，这在过去的几年里导致了一些问题，现在加上了强制检查。不过在微软，“授权”并不只是说说而已。
- 主管的人都鼓励和拥抱变化，然后他们还有耐心和毅力去将它贯彻实施。有太多的公司拼命保持着它们安逸的现状。而微软挑战自我，不断提高，重新去适应它帮着塑造的新的世界。

我热爱微软，每天都为它工作，只想让它变得更好。它并不完美——人造的东西何曾完美过，但如果我当主管的话，我将立足于它的特质并把它发扬光大，而不会根据我自己的不完美映像去重塑它。

——Eric

2001 年 11 月 1 日：“我是怎样懂得不再焦虑并爱上重组的”

这一期的《Interface》谈到了，随着 .NET 的启动，出现了一些技术变革和机会，其能够用于取悦我们的客户。但对于我们中的大部分人来说，变革还没有结束。我只是想说说它的阴暗面。

是的，这种阴暗面似乎在公司内部和部门策略上的每一次变革时都会出现：重组，该死的重组！在时间、精力和动力方面造成了致命的浪费。在这种动荡的过程中，会让你马上跑到桌子边趴在上面紧紧抓牢，害怕它被搬到另外一栋大楼去。

> **作者注：**这个关于重组（Reorganization）的栏目是针对管理者的。其实，《Interface》早期的栏目都是针对管理者的。

沿巴别塔而下

每当关于重组的消息沿着管理的层次逐级往下流传开来，沸沸扬扬，直到最后你的上司发出一封 E-mail 说，“大家继续努力工作，眼下没有任何变化。”关于重组的流言这才稍有停息。

当然，你所在的部门将在 3 ~ 6 个月内发生变化。也许你只是换一个办公室；也许你的管理层会更换；也许你所在的部门将跟另外一个部门合并，或者干脆被削减掉。谁知道呢？你的员工当然不知道，因为哪怕你确实知道，你也不会告诉他们。为什么呢？因为在你通知你的员工之时到变化真正发生时之间，计划可能会改变 5 ~ 6 次；况且提前告诉你的员工的话，还会引起他们的分心和不安。

> **作者注：**其实你不应该对你的员工隐瞒，正如我在“揭露真相”专栏中讨论的那样（参见第 1 章）。任何人问你关于重组流言的事，你都可以告诉他们真相：“我不知道将会发生什么，也不知道是否真有事情会发生。但我知道，如果我们不集中精力做好手头的工作的话，肯定会有事情发生。”

炼狱生活

在这期间，作为一名管理者，你的生活沦落到了活生生的地狱之中。即使你不告诉你的员工即将发生的变化，流言仍然会漫天飞舞。你很快就会感觉到初级员工身上积聚的恐慌，还有高级

员工流露出来的玩世不恭。除了按照平常一样每周做60多个小时的工作，你还需要：

- 让你的初级员工平静下来，打消他们的疑虑。
- 说服你的高级员工去关注变化。
- 为了搬迁到一个好的办公室而去讨价还价。

> **作者注：**关于办公室搬迁，我这里有两条小建议，避开大楼的西南角落，那里太热了；使用完全客观、透明的方法去进行办公室的内部分配。（微软使用自合同签订的那一刻起，到目前你在这个岗位上的时间作为依据，通常也称之为“资历”。）

- 玩弄办公室政治，跟与你的部门有交叉的新部门展开“死亡行军”。

> **作者注：**当两个部门合并时，因为每个部门只能有一位产品单元经理（Product Unit Manager，PUM）、一位部门项目经理（Group Program Manager，GPM）、一位开发经理（Dev Manager）和一位测试经理（Test Manager），一场高度政治性的斗争是在所难免的。

- 和你的管理者共同谋划，重新描述你所在部门当前的计划，使它们跟新上任的上层管理者所要求的相匹配。
- 在你的良心和同伴之间来回摇摆、努力抗争——按照新的公司方向对你的产品做出正确的修改，还是无视那个新方向，只管按时交付你的产品。
- 给你的新上司另外准备一套培训材料，以说明你的部门所做的事情。

为什么我们非要做这些乱七八糟的事情呢？为什么每9~18个月我们必须经历一次组织上的“自我摧毁”呢？我现在已经找到了一个说法。它不是老套的“因为我们的办公室空间不够了，我们必须保证至少有一个大部门能够继续前进”，而是……

不寻常之路

看一看IBM、波音这样的大公司，是什么让它们像药店的出纳员一样敏捷呢？恕我直言，旧组织跟旧习惯一样，它们都倾向于以阻力最小的方式前行，要把新的技艺教给旧组织是很难的。为什么呢？因为与陌生人相比，人们跟他们了解的人一起做事要快得多，也要容易得多。如果一个组织已经老于世故了，不管组织里的人有多精明，也不管谁在负责那个组织，他们仍然倾向于重复地跟相同的伙伴一起工作，而不会去跟新人打交道。

这就是让中层管理者愚笨、效率低下的秘密了。相同的管理者在相同的组织里呆的时间越长，他们就越有可能基于他们了解的人来做决策，而不是基于他们的本分，考虑什么对公司或者客户最有利。这是一种阴险的病毒，哪怕最聪明的人也会被感染上，而且几乎完全检测不出来。他们的生活看起来更加安逸和随便了，然而他们的选择也变得更加受限、刚愎自用，这完全是个灾难！

那要怎样才能使我们的管理者远离旧习惯，去关注新的职责呢？让他们动起来——经常性地流动。除了把所有的中层管理者撤销之外（这看起来很诱人，但实际上不那么通情达理），唯一能让我们这样的大公司像小公司一样行动的办法是，让人们持续流动起来——真正地流动起来！是的，跟被误导的或显然毫无头绪的新的管理层打交道会比较痛苦，但它每次都能打破“老朋

友”网络，从而使我们收获到敏捷、灵活和前向思维。

> **作者注：**另外的解决方案是把组织扁平化——也就是说，减少管理的层数，让更多的人直接给上层管理者汇报。在我写下本栏目之后，微软进行了一定程度的扁平化，有些部门甚至还在试验更为极端的做法。在这个方向上走得太远同样让我感到担忧。我没有养 15 个孩子——没有足够的精力去关心和照顾他们，但作为尝试还是值得的，我们能从结果中学到更多的东西。你可以在最后一篇专栏“我们是功能型的吗?”了解到更多微软最近采用的方法。

当然，重新审视你正在做的事情，尝试去解释它怎样适应公司的远景，这是一种健康运动；认识不同组织中的人，了解他们的项目，这也肯定是件好事。就这样，我学会了停止焦虑并爱上了重组。

是问题还是解决之道

当然，我们正处于一个高度竞争的市场环境中，各种变化时刻都在发生，新的技术、架构和平台要求我们不断调整我们的策略。但如果我们所做的是转变焦点而不是使人移位，那我们的公司就会患上组织“关节炎”——它不断僵化，直到有一天变成一具死尸。

问题是，你或你的部门是否就是问题的一部分？如果你在同一个组织里呆的时间太长了，也许周遭的环境让你感觉有点太安逸了，你要考虑一下做一点改变。如果你的上司在同一个组织里呆的时间太长了，也许这时候你该去焦虑了。

不管怎么样，下一次你收到史蒂夫·鲍尔默的邮件时，如果他说：“我对这些变化将要创造的机会感到异常兴奋，”你要好好理解它，然后对他说，“史蒂夫，我和你一样感到兴奋。加油！把那个新组织运转起来！”

2005 年 3 月 1 日：“你的产品单元经理是个游民吗?”

冒着侮辱一些出色的、我的老上司的危险，我还是要说，大部分产品单元经理（Product Unit Manager，PUM）都是游民。他们在走廊里踱步，一张口就会喷出疯狂而错综复杂的理论，说着事情应该是怎样怎样的——完全脱离现实，然而，他们却还要靠邻近办公室里工作的一群陌生人的仁慈才能生活。他们到底带来了什么价值？设置这样的职位究竟是何用意？

> **作者注：**我拿“游民”来做比较，但我肯定没有侮辱游民的意思。只有一小部分的游民精神上有问题。很多游民只是独立地过着他们所选择的生活。至于产品单元经理嘛，请继续往下阅读……

产品单元经理是第一级真正不做任何实际工作的管理者，他们不写规范书、代码、测试或内容，他们不做本地化、发布、实施或设计，他们只是管理实际做那些工作的人，同时负责他们的预算。管理多个工种和他们的预算是有难度的，但跟这些与世隔绝、精神错乱、苛求的上司打交

道同样也是有难度的。这两件事都不应该设立专门的职位去做。

> **作者注**：非常感谢以前的一位团队成员 Bernie Thompson，因为是他提出建议要我写这个主题的。他现在和另一位前团队成员 Corey Ladas 在一起，维护一个有趣的关于精益软件工程的网站。

有计划的人

当然，你的产品单元经理肯定会对你说，他担当的是一个关键的角色。为了方便起见，我们给他取个名字叫“无头苍蝇”。“无头苍蝇”花很多的时间在会议上，讨论业务的发展策略，他跟合作伙伴和客户培养关键的战略关系。“无头苍蝇”在他的员工会议上讨论这些战略，并且在其他一些场合进行阐释。他每个季度都要向他的上司（副总裁）以及全体团队展示他的3年战略。“战略计划是我们成功的关键!”“无头苍蝇”这么声称，“现在我们所有要做的事情就是执行了。”

然而，不知何故，“无头苍蝇”看起来总是跟现实脱离。也许是因为他没跟团队一起日复一日地努力工作，也许他已经从他的员工那里得到了足够的信息，也许他更痴迷于成为一位产品单元经理和一位领导者，而忽略了他还是要去关心实际工作的。或者也许他酷爱某种抽象的战略，以至于发布产品所需的方法和工具在他看起来都是多余的。“无头苍蝇”以为说一句“现在我们所有要做的事情就是执行了”就万事大吉了。麻烦的是，他把他自己遗忘在了“方程式”之外!

迫不及待付诸实施

每个商业计划都有两个方面：

- **战略方面**——我们打算完成什么事情？客户是谁？什么时候交付？
- **实施方面**——我们打算怎样去完成它？

“无头苍蝇”负责战略方面。对于那些不打算做实际工作的人来说，战略方面看起来更有趣味、更有意义。设定一个战略远景是很重要的，但它不能总是高高在上，它需要走到现实中来。因此当“无头苍蝇”关注战略的时候，他能够感觉到人们的需要。至于实施方面——没问题，他可以把它留给他那些称职的员工们。

但是，没有应用的理论只能用于自我满足。如果在大楼开始建造或软件开始编码的时候，架构师不坚持尽心尽职去解决问题，他们很快就会被边缘化。同样的道理，如果战略家不理睬实施方面，他们就会沦落成为徒有虚名的领袖，他们不理解为什么战略没有按照他们当初计划的那样发展下去，换句话说，他们真的变成了无头苍蝇。

魔鬼藏在细节里面

战略设定了，然而在项目开始了6个月之后，“无头苍蝇”搞不明白为什么事情已经在变坏了。于是，“无头苍蝇”开始参加所有的会议，问一大堆的问题，苛求结果，把他的员工和团队都给彻底逼疯了。

事必躬亲的“微观管理”（Micromanagement）当然也能把事情做好，只要“无头苍蝇”服下足够的兴奋剂，而且不怕惹人厌烦。然而，他肯定会被人鄙视，理由是：

- **没人喜欢被当做小孩。**“无头苍蝇”的员工很优秀，他应该让他们自己去开展工作。
- **所有人都会咒骂他是瓶颈。**当“无头苍蝇”卷入到项目中的各个环节，所有的决定都必须通过他时，于是他成了阻碍工作进度的主因。
- **成长的机会消失了。**“无头苍蝇”负责所有的决策，承担所有的风险。他不给机会让其他人进步，把荣誉统统归自己所有。哎，真是无头苍蝇！
- **大问题放着无人问津。**“无头苍蝇”的精力都放在了细节上，他对团队怎样去运作这样的大问题无暇顾及。整个团队像一盘散沙，大家都感到沮丧，士气消沉，忽略了工程系统的改进和项目在全局上的提高，结果注定是失败。

不过，产品单元经理确实能够找到更好的办法，成功推动团队执行一个战略，那就是所谓的“实施计划”。

道路规则

“实施计划”描述了你的团队打算怎样依据战略远景去执行。作为一名产品单元经理，你不必安排所有的细节，也不必所有的决定都由你来做，不过，你的的确确需要一份实施计划，以便项目在一步一步逼近理想目标的过程中能够不断适应和调整。就跟战略计划一样，实施计划也要由产品单元经理协同他的员工一起去创建，在战略被最终确定下来之前，团队的领导层需要明了他们将如何去实现它。

对于一个产品单元，所要决定的方面涉及人员、过程和工具。下面给出了实例，列举了一些你要去寻求答案的问题：

- **你的组织结构是什么样子的？**谁是主管？谁是功能团队？谁是领域专家、架构师？如果有人员离职，接替计划是什么样子的？
- **你们打算遵从什么样的过程？**这一次你想在哪些方面做得更好？你计划采用什么样的过程？你的质量目标是什么，以及你打算怎样去达到它们？在问题分诊或计划改变方面，相关的决定应该怎样做出？
- **团队将使用什么工具？**你用什么建造系统？你想做哪些方面的改进？你将使用什么语言、什么组件？你将如何自动而客观地度量你的质量目标，并且把状态反馈给团队？

当实施计划得到了应用之后，团队成员很容易就能得到他们所需的信息，以便走上正确的轨道并且不再脱离轨道，产品单元经理也能迅速地了解现实状况，知道要把他的注意力放在什么地方，找到正确的方向。

回到正确的跑道上

战略计划和实施计划都可能需要调整，特别是在一个长期项目进行的过程中。不过就算调整，团队也总是处在一个趋向成功的位置上。那是因为，战略计划提供了远景和优先级，告诉他们要去的地方，而实施计划提供了目标和界限，告诉他们怎样才能到达那里。

做实际工作的人，比如像你，常常抱怨得不到管理层的支持，你们没办法实施改进或一开始就把事情做好。好吧，那些改进写入实施计划了吗？你的产品单元经理让团队去负责执行那个计划了吗？还是团队只是负责按照产品单元经理向他的上司承诺的日期发布所有他所钟爱的功能？不要只是抱怨——找出其中的缺口，然后采取相应的措施。

你的产品单元经理是个游民吗？他是否只提供战略方向，而不给出清晰的目标和界限？当情况不妙的时候，他是否会变成“无头苍蝇”？如果是这样的话，该让你的产品单元经理提高一下了，叫他不仅要定义他所要的东西，而且还要定义他希望怎样把它做好。你要给他点指导，并且献出你的智慧。至于要度量什么团队目标，以及使用什么样的工具，你要帮助他一起来做好这些决定，以推动你们共同的成功。这样的话，你的产品单元经理就不再靠施舍为生了，你也将得到所有你需要的支持，最终把事情办妥！

作者注：6 年后，产品单元经理实实在在成为了过去。过去那些小而相互关联在一起的产品开发组织，现在已经变成单一化的、大型的功能性组织。这些变化从 Office 开始，然后拓展到 Windows、Developer division、Dynamics、Windows Phone、Bing & MSN，以及我最近所在的部门——Interactive Entertainment。唯一保留产品单元经理的大型部门是 Windows Server，不过我猜也会在一年内改变。在后面的章节“我们是功能型的吗？”中我将讨论如何转变成一个功能性组织及其意义。

2006 年 9 月 1 日：“有幸成为 Windows 的主宰者”

Windows Vista 这个项目正在接近尾声。看来那一天迟早要到来了——有人会宣布他们将要转入新的部门，也有人宣告，“我是多么激动啊！因为我马上就会加入［填上你最喜欢的组织］。”

是的，生活是不确定的，变化导致模糊和混乱，但我喜欢“重组”。我甚至为此专门写过一篇专栏（“我是怎么懂得不再焦虑并爱上重组的”，参见本章的第一篇专栏）。但在跟被误导的或显然毫无头绪的新任管理者打交道的时候，你又怎样才能保持心智健全呢？有一种比较有趣的娱乐方式，那就是把你自己想象成那位新任的领导。

如果你成为视窗之王，你会怎么办？你会做些什么？你是否有胆量站出来，说给我们听听呢？什么？你是说我吗？我想，你从来没敢奢望过……

作者注：我上面的高级副总裁，Joe Devaan，调任到了公司核心的 Windows 部门，在那之后不久我写下了本栏目。我曾经问他是否想要复审一下这篇专栏，因为大家都知道我曾是他的下属。Joe 说，“我想我最好还是不去干预你……我能肯定，你不会听我的话，试都不用试☺”自那以后，Steven Sinofsky 接任了 Windows 的总裁，并把这个部门改造成为一个健全的组织结构，就像他在 Office 开发部门中的那样。对于 Windows，Steven 照我谋划的结构做了，但是他没有埋怨产品单元经理，而是应用了很多控制个体团队的工程管理过程。你当然不能对他的成功有什么微词，我也不会。Windows 7 是款绝妙的产品，我可不是为了对付市面上其他的竞争产品而这么说的。

你还有别的要求吗

如果我做了 Windows 的主宰者，我将做下面的这些事情：

- 我肯定会跟每个人交谈，如此等等。是的，没错，肯定会这样。然后……

- ❑ 我会促成组织结构与软件架构之间的匹配。
- ❑ 我会基于用户体验去决定软件架构。
- ❑ 我会基于每种使用类别中的各个关键案例提升用户体验。

- ■ 但在挑选关键案例之前，我要先挑选我的直接下属。
- ■ 我的下属和我将一起分析并确定所有的关键案例。
- ■ 我会派我的下属去领导用户体验和软件架构方面的工作。
- ■ 当上面这些步骤完成的时候，我将建立起组织结构，并且委派领导者。
- ■ 我将把新组织的第一个里程碑放在质量和软件架构定义的工程系统的扩建上面。
- ■ 我将基于案例的优先级和关键链深度为各个功能设置优先级。
- ■ 我将按照优先级顺序，通过尽量缩短功能完成周期的方法管理产品的发布。
- ■ 我将要求已完成的功能达到甚至超过工程系统强制的“狗食”质量门槛。

注意，我这里并没有提到任何具体的方法，比如“测试驱动开发”（Test-Driven Development，TDD）或“功能小组”（Feature Crew）。尽管像这类精益的方法应该在工程系统中被鼓励和支持，但它们不应该是自顶向下的法令。团队动态应该是自底向上驱动的。

现在，我们逐个来讲，因为其中的细节会更加有趣。

准备轮船

跟每个人交谈，如此等等。这并不是对现有员工走的一次毫无价值的“过场”。后面我要从他们当中挑选出我的直接下属的，因此我必须了解当前的情况，以及清楚我有哪些人可以被委任去提供服务。我这里用了一个关键词——“服务”。我并不是在寻找为我或他们自己提供服务的人。那两种人都会毒害我的员工和我们的目标。我希望尽快把他们解雇掉。我要找的是，能够服务于微软和我们的客户的人。

> **作者注：**在微软（或者其他任何公司），如果说我将在管理方面做一些改变的话，那会是用于判断谁可以被委任去提供服务的标准。是的，最好的管理者服务于公司和我们的客户，但公司同时也掺杂着那么一些管理者，他们服务于自己、他们的上司或者他们自己的小团体——这显然是不合理的！

促成组织结构与软件架构之间的匹配。你想让组织结构图看起来跟软件架构的布局图一样。这有利于各个部门按照软件架构定义的那样，在本地进行独立的实施和控制。它也更容易强化架构边界。很显然，你只有拥有了软件架构之后才能做到这一点。然而，你仍然可以认真考虑你所想要的组织结构。我想根据软件架构中的各个主要组件，划分出对应的各个产品部门。每个部门都将设一位产品单元经理（Product Unit Manager，PUM）、架构师（Architect）、部门项目经理（Group Program Manager，GPM）、开发经理（Dev Manager）和测试经理（Test Manager）。

> **作者注：**不再需要三人组，撤掉产品单元经理，保留部门项目经理及测试经理。你也知道 Steven Sinofsky 对 Windows 7 就是这样干的。

基于用户体验决定软件架构。这里有时机和职责的问题。很显然，定义 Windows 用户体验和软件架构这些事情，必须发生在任何产品部门组建或任何实际工作开始之前。否则的话，你不可能基于软件架构来构建你的组织。

然而，即使在定义了用户体验和软件架构之后，它们也会经常性地被重新检查、改进和重构。基于这个原因，我将让“用户体验”（User eXperience，UX）部门直接汇报给我，我还会委派一位总监给这个部门。我将设置一位产品线架构师，让所有各个产品单元的架构师都间接汇报给他；他们这些人作为 Windows 架构团队，需要定期开会。

我也将为项目管理、开发和测试分别设立总监的职位，他们都直接汇报给我。他们将对工程系统负责，维护着一个长长的质量需求的清单，比如安全、隐私、可靠性、性能、可维护性、可到达性、全球化，等等。所有产品单元的工种经理，都间接汇报给项目管理、开发和测试的各位总监。这样的话，我的下属就能跨部门驱动整个工程系统了。

> **作者注：**Steven Sinofsky 将用户体验团队归为项目经理的下属，将产品级架构师归为开发部门总监的下属。而部门项目经理、开发经理及测试经理归于他们各自的总监管理，而总监们从属于 Steven（不是部门项目经理）。这很简单，我敢说，这正是我想要的。不过，他是 Winows 的老大，而我不是。

设置路线

在各细分领域，客户体验取决于关键场景。关键场景是什么？它带来了一种特殊的客户价值，它将新版 Windows 与旧版 Windows 还有竞争对手的产品区别开，通常，市场与技术研究人员会告诉我们各个细分市场特殊的价值取向在哪。得了，这还用说。但是，各位同仁必需押个注：什么才能称为“价值取向”，因为我们都要为此矢志不渝，押上我们的身家性命。

挑选我的员工。招募员工是我必须做的最重要的事，如我之前所说，我要找的是能全神贯注服务于微软及我们的客户的人，而不是服务于我，也不是服务于他们自己。他们或许有职业热忱，但是他们觉得如果他们服务于微软及我们的客户，他们的职业生涯就悬了。对于他们的生活态度，我希望在这方面我的员工与我的理念相同：首先照顾好家庭、朋友，服务于社会，其次是微软及我们的客户，最后才是我个人。

分析并创建关键场景。当工作人员各就各位，我们就要开始为下一个版本的 Windows 定义价值取向。这意味着，激烈的争论及细节分析将旷日持久，最终会有一个清晰的 Windows 构架，成功即在眼前。

起航

委派我的下属去领导用户体验和软件架构方面的工作。“价值主张”定义好之后，用户体验和软件架构方面的工作就要认真开展起来了。这些团队应该在一起紧密合作，彼此交流什么可行、什么不可行之类的信息。最后，我们应该得到关键客户认可的原型程序，以及一个能让我们相信我们的目标能够实现的产品线架构。在正式建立组织结构的时候，用户体验总监和产品线架构师将以这个阶段组建起来的团队为核心，进一步扩展他们的团队。

建立起组织结构，并且委派领导者。软件架构定义好之后，我们就能把组织填充起来了。当

初帮助我一起创建价值主张、用户体验和软件架构的外延员工，他们在关键领导角色的委任上会被优先考虑。之后，我的下属将被分配或者获得剩下的职位。完全的职位过渡将定在 Windows Vista 发布日期之后的 1～2 周之内进行。

把新组织的第一个里程碑放在质量上面，并且扩建软件架构定义的工程系统。软件架构中的关键一块是工程系统的定义——也就是说，能够让工程团队建造和发布我们的产品的一组工具和过程。通常情况下，一个新的架构将需要新的技术和新的方法。如果幸运的话，我们还可以应用在过去所犯错误中学到的经验和教训。第一个里程碑提供了一次去关注尚未解决的质量问题的机会，同时可以扩建改进过的工具、度量和过程。

> **作者注：**微软现在的很多产品线（包括 Windows），都把部门范围内的第一个里程碑放在质量和工程系统的扩建上面。这个趋势早在本栏目发表之前就已经存在了。

导航

基于案例的优先级和关键链深度为各个功能设置优先级。当质量里程碑完成之后，工程人员也将准备好了去创造新的客户价值。按照正确的顺序去创造价值是我们按期发布产品的关键。我将基于案例的优先级（最重要的功能优先）和关键链深度（最关键的受依赖组件优先）来决定那个顺序。这个优先级排序过程和结果必须对所有的工程人员透明。

按照优先级顺序，通过尽量缩短功能的完成周期的方法管理产品的发布。我选择什么样的方法去管理产品的发布，将是我要做的第三重要的事情。第一重要的事是挑选我的下属，第二重要的事是定义工作的“完成”（下面我会谈到这个）。

说到产品发布的管理，我会选择按照优先级顺序尽量缩短功能的完成周期的方法。换句话说，我想让各个功能按照优先级顺序一个一个被完成，并且每个功能从功能团队开始讨论规范书，到那个功能为“吃狗食”做好准备，之间跨越的时间要力争最短。

> **作者注：**“吃狗食”（Dogfooding）是在日常工作中使用未发布产品的一种实践，它鼓励团队从一开始就把产品做对。通过这种实践可以收集对产品价值和可用性的早期反馈。

很自然，我将应用被证实有效的项目管理实践，去跟踪整个 Windows 项目的进展情况。但我们部门快速、可靠地发布产品的能力，跟每个功能团队应用同样有效的项目管理实践的能力是直接相关的。

我不会限定任何功能团队的内部结构，也不会指定他们必须遵循的方法论。那些最好都由团队自己去决定。但我会坚持，我们应该有效地为客户创造价值，确保完整而高质量的功能尽可能快地发布出去。

这将有利于从早期的客户反馈到最后时刻的竞争回应之间的所有活动。如果像测试驱动开发和功能小组这样的精益方法，正好能够产生最快的功能完成周期的话，那就再好不过了！

要求已完成的功能达到甚至超过工程系统强制的“狗食”质量门槛。除了挑选我的下属之外，我要做的最重要的事是定出这样的要求——除非所做的功能达到了高质量的门槛，否则它们就不能被称为“完成”。我选择的质量门槛是，能够用作“狗食”。我将在工程系统中强制执行这

个质量标准，并且我会常常在自己的机器上安装最新的“狗食”软件。通过那种方式，我的下属和我将可以时刻留意到，工程系统是否应用了正确的度量去维持标准。简单来说，如果你的功能够不到质量门槛，它就不应该也不能被签入到主代码库中去。

作者注：典型情况下，功能小组在主代码库派生出来的源代码分支上开展工作。这允许他们在不影响他们自己小团队之外任何人的前提下，使用源代码控制系统去管理冲突和建造。当功能被完全开发和测试过之后，能够用作“狗食”了，功能小组的代码分支就会签入到主代码库中进行合并。当然，其他的功能小组也会把他们的代码签入到主代码库中去，因此，一个功能团队留在自己分支上的时间越长，他们跟主代码库的合并就会越困难。那是另外一个你想要尽量缩短功能的完成周期的原因。

责任

对“完成”进行预防性的定义，是免受责难的责任哲学中的一个关键部分。架构边界得到了支持，因为它们跟组织边界是匹配的。正确的价值、案例和用户体验得到了传递，因为我们就是基于传递那些价值而强化的优先级排序。几乎没有工作浪费，因为我们度量并奖励那些以最快速度传递完整价值的人。产品肯定是高质量的，因为你做完任何事情都可能得不到荣誉，除非所做之事够得上质量门槛。

没有了责难，因为是系统（而不是个人）在强制执行这样的要求——质量、价值和一个健壮的软件架构必须有效地在产品中得到体现。破坏软件架构、价值或质量的唯一办法是，与你的管理层勾结，“光天化日”之下去破坏所有的规则。也许这样做是对的，也许不对。不过两种情况下，责任都是明确的。

下一代 Windows

那就是开发新版 Windows 的 12 个步骤了。没什么复杂或苛刻的——也许有一些时限问题，还有一些工具和度量工作要做，但总的来说，我觉得它是可行的，也是有效的。唉，可惜我不是 Windows 的主宰者。我顶多也就能当个副手。

然而，变化总是伴随有机会，而大量的变化都在进行当中。让别人听到你的声音！如果你不是主宰者，那就把你的想法和建议发给那个主宰的人。谁知道呢？我可能还会掀起一场革命！

作者注：Steven Sinofsky 带来的转变对于很多 Windows 工程师来说都是很艰难的调整，不过，结果告诉他们自从 Steven 接手 Windows 后，Windows 已变得更有效率、更具创新性了，这需要 iPad 以及其他流行的平板设备成为消费者 PC 机的替代品。个人来讲，我不认为 PC 机已过时，是的，有些 PC 机可以成为多用途的设备，它们可以像手机一样放在你的口袋里并接上键盘、鼠标及显示屏；而有些是平板的，也可以以同样方式接上外接设备，但是，有些也还是保持一种便捷的规格形式。我也相信，5 年之后，大多数产品都将运行 Windows 系统而不管它们的尺寸与外形是怎样的。

2006 年 12 月 1 日："Google：严重的威胁还是糟糕的拼写？"

也许我很无知，但 Google 想要跟微软竞争真的很可悲。微软还远远没有达到完美的境地，除非某个公司走近我们，在很短的时间内实实地砍我们一刀，否则想要击倒我们是很难的。那个公司肯定不是 Google。

> **作者注：**这是我写过的最有争议的栏目之一。不管是公司内部还是公司外部，很多人都相信，就像微软当年以个人电脑的服务取代 IBM 的基于大型机的服务一样，Google 的基于网络的服务也将最终取代微软的服务。网络正在改变世界，我们的传统业务（Windows 和 Office）使得我们很难像 Google 一样敏捷，在这些观点上我同意他们的看法。然而，尽管我尊重他们，但我不同意他们所得出的结论。

别误会我！对于 Google 已经做的一些事情，我的印象还是相当深刻的：

- 他们创造了一个有优势的互联网搜索引擎。
- 他们通过两次收购，摆脱了文本驱动的广告服务，创建了一个有效的收入模式。
- 他们营造了善待工程师的工作环境，以吸引和留住优秀的人才。

> **作者注：**我们现在可以说他们（Google）已经开发了适合于开发者的手机以及操作方便的平板设备。

所有那些都是伟大的成就。

然而，从长远来看，那些成就又都算不了什么。如果 Google 幸运的话，它将成为一个像苹果公司那样的成功的隙缝企业。如果 Google 不是那么幸运、不够足智多谋的话，它将加入到 Borland、Netscape、Corel 及 Digital 等这类公司的行列中去。是的，那些公司中有些仍然在运作它们的业务，但它们现在只不过是它们曾经的竞争对手的躯壳而已。

他们步伐踉跄，我们手舞足蹈

我们竞争对手的愚蠢和短视，从来就没有停止过让我吃惊；Google 也不例外。它简直太简单了！也许他们在想，他们集体的无能会麻痹我们，让我们变得傲慢自大、粗心大意、放纵不羁，最后我们忘了自己是谁，忘了我们要做什么。

> **作者注：**好吧，我承认我说过头了。但我之所以那么说，是因为我觉得人们对 Google 的追捧已经远远超出了正常的限度，我必须要把它拉回来一点。顺便说一下，我有我欣赏的朋友在 Google 和其他竞争对手公司里工作。我质疑的是那些公司的业务方向，而不是它们的员工。

我们当然会对自大、粗心和放纵感到心虚。对此我会第一个承认。但谢天谢地，我们没有忘记自己是谁，也没有忘记我们要做什么：

- 我们通过软件赋予每个人以及每台设备以力量。

- 我们持续不断地改进我们的产品。
- 我们持续不断地提高我们自己。

我们遵循这些准则，是因为我们关心每一位客户。我们想要改变世界！

注定要失败

看到微软已经取得的所有成就，你可能认为，我们的竞争对手也会遵循同样的准则啊！但是他们没有。不是他们尝试得不够，而是他们很明确地选择不那么做。

我们的大部分竞争对手都停止了改进他们的产品。他们固步自封，满足于产品的当前版本，就跟我们曾经在 IE（Internet Explorer，互联网浏览器）上所犯的错误一样。这给了我们追上他们的机会，并在我们做出第三版的时候击败他们。现在，Google 正努力纠正着这个错误。他们以前过分强调新的想法，现在他们的高层管理者不得不告诉工程师，在创建新产品上要紧缩开支，要把注意力重新放到他们现有的产品上去。

我们的大部分竞争对手也停止了提高他们自己。在众多的研究和开发领域中，我们投入的经费比所有竞争对手都要多。不过随着老竞争对手的落败，新的竞争对手也会取代他们的位置。

另外，在我们能够满足客户对成熟产品质量越来越高的期望之前，我们已经走过了很长的一段路。我们很幸运，因为尽管 Google 的现状还不错，但他们没有特别重视质量工程。除了他们出色但也缺乏活力的产品之外，他们对我们的最大挑战是，他们营造了一个善待工程师的工作环境。不过，通过我们的持续改进，要不了多少时间，我们最终也将在这两个领域把他们击败。

> **作者注：**当 Google 逐步提升他的搜索引擎、广告、浏览器及安卓系统水平时，他们仍不能显示出他们可以在用户体验方面做到改革性的提升，微软也用了好几年的时间才从开发者用户界面及功能泛滥中解脱出来，进入到了集成的、舒适的用户体验提升中。用户体验是苹果擅长，Google 还远远落后，而微软正逐步赶上苹果，并在我们最新版的产品中有超越之势（苹果的狂热者现在可以傻看着他们的 iPhone 了，而数着念珠按钮也可以让他们安心了）。

聪明人需要智能客户端

但让我们假设 Google 也行动起来了，开始对它的产品和它自身进行持续不断的改进。Google 在吸引并留住优秀工程师方面已经做得非常出色了。他们可能决定把他们的精力重新放到客户价值、产品质量、服务的集成和多次发布计划上面。难道他们不将是一个威胁，特别是他们还有梦幻般的、实际上像摇钱树一样的广告平台的支持？

不是。Google 顶多也就是像苹果公司一样的隙缝企业。为什么呢？因为我们的第一条准则是，我们通过软件赋予每个人以及每台设备以力量——Google 和苹果公司一样，对此选择了漠视。

苹果公司想要对他们选择的人和设备赋予力量。这个策略有它的优势，但也结构性地限制了他们的市场。要不然的话，他们将是一个很难对付的竞争对手；那对微软就太糟糕了！

> **作者注：**我曾经写过一篇关于 Mac 系统的专题文章。我很欣赏苹果公司在设计和用户体验方面已经做出的贡献。一段时间以来，微软在致力于高质量的工程系统的同时，也在致力于高质量的用户体验，而且它已经成为我们自身在新旧领域中获取成功的原动力。

Google 想要通过软件为每个人赋予力量，但不包括每台设备。他们更愿意让服务器保持智能，或者说让客户端保持最简单的应用。那是“网络计算机 + 傻瓜客户端”的陈旧模式。值得注意的是，曾经有多少次，这个失败的策略声称它已经战胜了智能客户端。

> **作者注：**有了安卓，Google 已经进军手机及平板电脑领域了，这很好。但是他们仅用这些设备作为他们的广告平台，他们的业务模式并没有改变。对于安卓，他们并不需要对操作者负多少责任，或者一点责任也没有，却在服务连接及广告上日进斗金。我并不是说这是罪恶的——而是说他们做得还不够。Google 并不是制造最好硬件设备的料，他们在广告推广上才是行家里手。最后，他们差强人意的用户体验将使他们落于竞争者之后，再等几年，你会明白我说的是什么意思。

微软把软件的力量带进了每个人的办公室、家庭和手中。那种力量离人们越近，他们的体验就会变得越是格外地有价值。Google 如果限制自身的涉足范围，它最终将会落败。至于它会落败到何种程度，那还要看他们是否能够成功守住基于服务器应用的少数几片领地。

> **作者注：**有些人可能会争论道，AJAX 给客户端提供了足够的智能啊，再说 Google 也确实发布了智能客户端产品，比如 Google Earth（这个我喜欢）。然而，AJAX 的作用是有限的，Google 的软件开发过程（很多人认为那是它的优势）也只是最适宜于网络发布。Google 如果要聚焦智能客户端的话，它需要做出很大的改变，很大程度上要像微软已经做出的改变那样——敞开心扉去迎接网络服务。

保持警惕

不要仅仅因为 Google 难逃它自身的宿命我们就可以掉以轻心。我们必须不断地提高我们自己，营造一个吸引并能留住有能力的人（包括 Google 员工，如果他们“弃船而来”的话）的环境。我们必须不断地改进我们的产品和服务，提供比 Google 更为引人注目的价值，还有完整的、超乎想象的用户体验。

我们必须保持警惕，因为未来某一天——也许现在在某人的车库中，或者在印度或中国已经开始了，将会出现一个领会我们所有准则的公司。他们不像我们这样有遗留代码和陈旧工程方法的包袱。他们将关注精益、高质量的实践和产品，公司起步规模虽小，但慢慢地在扩大他们的影响。这在每个行业内都会发生，而我们所处的软件行业也不会例外。

一马当先

庆幸的是，我们已经提前感觉到了这种变化。我们知道，市场已经成熟了。我们知道，我们必须转变我们的工程系统和方法，以便凭借更为出色的可预测能力和效率，生产出更高质量的、能够提供更多价值的产品和服务。持续改进我们的产品和不断提高我们自己的准则，也在慢慢地使那种转变变成现实。时间和耐心是必要的代价，但事实上我们非常擅长于花费时间、耐住性子。

我们知道，最后我们一定会赢。你可以去感受那令人兴奋的时刻。你可以为我们的工程和产品的改进做出贡献。无论是每一个人还是每一台设备，你都可以把质量和客户价值放在第一位。

改变是困难的。总有那么一部分人更喜欢现状，尤其是他们更适应于旧习惯。但变化总是伴随有机会，那个机会对于你来说，可能是定义甚至领导一个新的潮流。在像 Google 这样无能的竞争对手给我们机会的时候，一定要抓住它！

2007 年 4 月 1 日："中年危机"

最近，我花了大量的积蓄买了辆跑车——一辆双座的 Tesla Roadster 敞篷车，它可以在 4 秒钟之内完成由 0 至 60 英里的提速。那是一款全电动力的跑车，充电满 3.5 小时之后，总共可以行驶 200 多英里。从现在算起的几个月里，我打算休公假到国外去旅行。我知道你在想什么。你猜我肯定 40 好几了。

作者注：关于 Tesla Roadster，你可以到 http://www.teslamotors.com 了解到更多的信息。我在 2009 年也买了一辆（2008 年款的）。过去两年中，我每天都开着这辆车上班。这是一辆这世上最方便、最经济、最绿色也最可爱的一辆车。你可以在我的博客上了解更多，博客标题是 Tesla，www.teslamotors.com/blog/we-didnt-want-roadster。

是的，我正在经历中年危机，或者按我喜欢的说法，我到了享受生活的时候了。我的妻子和我仍然幸福地生活在一起；我爱我的孩子们，我未曾想过要遗弃他们；我的工作很棒，我跟以前一样对微软充满热情。我只是准备到世界的其他地方去旅行，我也就是买了辆很拉风的电动力跑车。那说明什么问题呢？更重要的是，那又关你什么事呢？

那说明的问题是，我的境况已经变了，我面临着新的机会和隐患。你之所以要关心，是因为微软与我的境况相同。

作者注：为了本书的出版，我要把以前的各个栏目整理一遍；做这件事情我也用到了我的公休假。在微软，在你连续几年对公司做出了杰出贡献之后，你上面的副总裁可能会奖赏你一段时间的公休假。

你已经变了

在这之前，对于我的生活来说，拥有一辆跑车是不可思议的。在我有孩子之前，我买不起这样的东西。而当我有了孩子之后，一辆双座的 Roadster 又不实用。现在，我的孩子都长大了，我的积蓄也跟着多起来了，买跑车也是有生以来第一次变得可行了。

当然，只是可行并不意味着我马上就要买跑车。不过，这可是一款全电动力的、开动起来其劲风能够"吹掉你的家门"、"扭垮你的脖子"的汽车啊！这样的汽车甚至直到去年才在市场上出现。发展到现在，技术已经得到了改进，其他方面的商业气候也有了变化。于是，我拿出了我的积蓄。

同样的道理，在这之前，休公假对我来说并不合适。我花了较长的时间才赢得了公休假。如果我那么做了，对我的孩子和我的工作来说，离开那么长的一段时间都是不可思议的。但现在，

我的孩子不太需要我了，我也把我的下属培养起来了，他们可以为我照料好工作。因此，休公假也变得可行了。

微软也已经长大了。我们过去一直在努力奋斗，为的是把我们的事业创建起来。甚至我们曾经很成功，但多年来我们还不得不坚持不懈地工作，为的是不被别人轻视。现在我们取得了领导地位，银行里也有钱了，却发现自己站在一个十字路口……

日子照过，只不过要掌握一点窍门

那么，这是不是危机呢？对于我的个人生活来说，不是！我的工作和人际关系很稳定。我的财务状况良好。有人愿意，他们也有能力为我照料工作。所谓的“危机”，其实是一次可以纵容我自己的机会。

那微软又怎么样呢？人们把我们的境况描述成危机。他们说，我们已经忘记了怎么去发布产品；我们在“税收”处理方面陷入了困境；我们的管理人员和项目方法已经过时了；我们的竞争对手在人才、产品和服务方面更加敏捷，更加贴近于当今的市场。他们说得对。哪里出问题了呢？在本该稳步前进的时候，我们却在缓慢爬行。

> **作者注：**我不认为微软的境况有像某些批评家说的那么糟糕，但他们的担心还是有道理的。

微软的中年危机有多糟糕呢？我会说，比我的“危机”糟糕，但比我的一位已经经历过这个阶段的老同事要好。他跟他原来的妻子离了婚，他遗弃了他的孩子，然后又跟一个只有他一半年龄的女人结了婚。IBM 的中年危机跟我的那位同事差不多。苹果公司在它的中年危机时离了两次婚，最后还是娶回了它的原配。尽管 IBM 和苹果公司都已经经历过了中年危机，但他们也着实痛苦了一番。

那微软在这个过渡时期又要怎样去避免痛苦呢？我们不去避免，事实证明，我们没能避免得了。我们正在遭受着痛苦，我们也将继续痛苦下去。转变是困难的。然而，我们总能做些事情去把我们的痛苦降到最低的程度，并最大程度地享受变化所带给我们的机会。

不轻易冒险

让我们回过头来再谈一谈我。记住，我要休一个长假，之后才能回到我热爱的电动力跑车、家庭和工作中去。为什么我的妻子不离开我？为什么有人愿意也有能力为我照料工作？我一定碰到了好运气，毫无疑问，但我的境况并不是偶然的，当然也不是我的一厢情愿。

假设我几年前就买了一辆双座的跑车，我的妻子也许能够容忍，但那会给我的家庭带来不幸。车子的购买、保险和维修等费用会给我们的财务状况增加很大的压力。本来下班之后，我和我妻子轮流去学校接孩子回家的，但因为小孩子坐在前排座位上不安全，我就不能履行这个责任了。更糟糕的是，我会错过很多跟我的孩子们接触的机会。

但我并没有那么早买跑车。因此，我跟我的妻子和孩子们的感情越发稳固了，我们的财务状况也更加保险了。就这样，我现在去买一辆跑车就无关紧要了。附加的好处还有，新的技术得到了应用，我现在可以买到全电动力的跑车了。

这里的道理是，当一些令人激动的新兴领域出现时，不要过早地采取行动——你要确信，你

是否安置好了依赖你的人，并且你是否能够管理好风险。等一等的话，你可能发现到那时的技术进步会更加讨人喜欢。

> **作者注：**顺便说一下，我这里并无意去评判那些拥有跑车的人。每个人的家庭和工作情况都不一样。我这里说的是，一个特定的方法是怎样帮助到我的家庭的，它是怎样帮助到我的工作的，我又是如何成功地预测到生活所带来的变化的，以及我的那个方法怎样才能够帮助到微软。

我认为他们还不能胜任

假设我在几年前刚得到公休假的时候，马上就把它休掉，那我的家庭、我的团队甚至我的假期的质量都会大打折扣。家庭方面比较简单。如今，我的孩子们都长大了不少，他们变得更加独立了。这意味着，我的妻子可以为他们少操心一点了，孩子们需要从我这里得到的辅导也比以前少了。附加的好处还有，对于我们去哪里度假以及呆多长时间，我们有了更大的灵活性。

工作方面的影响要大很多。几年前，我的团队才刚刚成立。我还没来得及雇用到合适的员工，也没有时间充分发展他们进入角色。我当时还没有产出可靠的结果，建立起必要的实践和指导方针。如今，尽管形势还不够完美，但在我背后已经存在了一支强大的、经验丰富的团队，他们渴望也能够代替我行使职责。

这里的道理是，当你把精力和时间投入到发展优秀的员工、实践和指导方针时，在让你的人有机会接手做你的工作，并把你的工作当成他们自己的工作的同时，你也获得了自由，从而你可以在没有焦虑或不受连累的情况下去探索新的领域。

不再年轻了

到目前为止，读者中的很多人都开始明白了这跟微软的关系，但假设你来自一个“无知便是福”或者拒绝承认的国度，请允许我细细道来。

微软正在步入中年：

- **我们的市场已经成熟了。**我们已经不再被客户、合作伙伴和政府轻视了。这对我们是好事，但它也对我们发布的东西、怎样发布以及何时发布提出了更高的期望。我们将承受不起因为疏忽或鲁莽而要付出的代价。
- **我们背着好几吨重的陈年包袱。**在没有额外负担的情况下，我们的竞争对手可以比我们更敏捷，他们能够更快地利用新的技术和收入模式。
- **我们的高层管理者、中层管理者和校园新兵足足有几代人。**代沟问题相当地严重。高层管理者从来就不用跟广大工程师一起发布软件产品。中层管理者搞不清楚“敏捷”和“脆弱”之间的差别（译者注：敏捷的英语是 agile，脆弱的英语是 fragile，拼写很相近）。而校园新兵对于产品级的质量和世界级的代码毫无头绪。这导致了对开发时间和工程方法的不一致的期望，同时还有基于错误信息的糟糕决策。

不要惊慌失措

我们该做些什么呢？首先，不要惊慌。我们有钱、有名，而且表面上看起来还不错（从投资组合的角度看）。形势比我们糟糕的公司比比皆是。

其次，记住我们不是人。我们是一家公司。只有在阻止创新，不让下一代取代我们的位置的时候，我们才会变得衰老而死去。如果我们采取了正确的做法，我们就能不断学习，并且使我们自身得以延续。

应用我的第一条道理，我们在采取行动之前，必须先安置好依赖我们的人，管理好我们的风险。我们已经年迈了，对于一些不稳定的基础组件，我们已经无力回旋了；我们在开发 Windows Vista 的时候就已经碰到了这种情况。积极地去研究新技术，创新我们的产品，但不要急着把它们应用到真正的产品中去，除非我们依赖的东西已经稳定了，并且风险也在可控范围之内了。

> **作者注：**我们最后把 Windows Vista 的某些新技术抛弃了，因为它们还不够稳定，还不足以保证高质量的发布。那真是极大的耻辱！当初把那些新技术加进来时我们花了不少的时间，而后来我们要去除它们时浪费了更多的时间。

应用我的第二条道理，我们必须投入到发展优秀的员工、实践和指导方针中去。我们已经年迈了，我们不能再用“死亡行军”去摧毁和挫败我们的员工了（更多信息参见第 1 章的“向死亡进军”栏目）。我们必须要有耐心，允许人们靠他们自己去学习和成长，同时传授给他们我们已经经历过的经验和教训。这意味着，我们要放开手，信任我们的下一代去证明他们自己，同时也要设置好正确的指导方针，以避免灾难的发生。

没有人是完美的

我们在很多方面还不够完美，包括如何建立稳固的依赖关系、管理好我们的风险、发展我们的员工、把正确的实践和指导方针设置到位，等等。转变是困难的，但我们别无选择。

真正能够起到帮助作用的，是在各个层次上的成熟。换句话说，要让我们的举止与年龄相称：

- 没有哪个负责任的成年人会开着轮胎花纹早已磨光的汽车到处乱跑，或者在地质断层线上建造房子。那很愚蠢！我们必须在依靠我们的依赖对象之前，先把它们稳定下来。我们不能在质量上妥协。
- 没有哪个有分寸的父母仍然告诉他们长大成人的孩子要去做什么。那很幼稚！管理层需要为工程师提供知识、经验和指导方针，清楚地说明他们期望的结果，提供危险警告这样的反馈，然后信任工程师以无论哪种他们认为最合适的方法去实施。

采取这些简单的步骤之后，我们将表现出超越我们年龄的成熟。我们将发展出新的领导者，他们可以在老一代人卸任的时候迅速地衔接上。当年轻一代继续我们的工作并以之为己任的时候，我们就可以在没有焦虑或不受连累的情况下去探索新的领域了。

作为公司，“长大”并不意味着像人那样真的“变老”。通过发展、保护和信任我们的年轻人，我们仍然能够保持年轻。我已经等不及那辆电动力跑车的交付了！

> **作者注：**如果当前的行动就是标志的话，我们已经很好地学到了许多教训。我们已经避免去直接告诉工程师怎么做事情；取而代之的是，我们关注我们期望的结果。我们已经改变了我们处理依赖关系和产品发布的方法，这使得我们自己更加精益、敏捷、可靠。当然，微软还有很多事情要做，我也很喜欢致力于为它寻找解决方案。

2008 年 11 月 1 日：“虚无主义及其他的创新毒药”

创新就是创造一些新玩意儿吗（字典上是这么定义的）？或者是依别人的成果再创造？对我来说，微软作为一个公司，她拥有自己的企业文化，她对这个基本问题的认识严重有误。我们每天都要困扰于这样的问题，这让我们的自尊大受影响，并大大削弱我们创新的能力。

正确的答案是，创新是对他人成果的一种提升——没什么新鲜的。想想人们所谓的那些创新产品，iPhone、hybrid 轿车以及 Facebook，这些都是创新吗？你开玩笑吧？如果没有它们背后的那些基础技术，不用说这些创新是不可能的，而且即使拥有这些创新特征的产品也早已有其他类似的存在。创新不过就是将旧的技术凑在一起玩出新的用户体验，并在市场上得到广大用户拥护。

这不是针对创新或是 Facebook 取得的伟大成就的控诉，恰恰相反——这是想让人们知道创新是对他人成果的一种提升。微软的企业文化是一种误入歧途的自残行为，按她的做法就是通过自身努力重新创造一些“新玩意儿”来体现创新。这直接引起三种不良的后果：

- 我们对我们作为一个公司的整体创新精神产生怀疑。
- 我们开发了一些毫不相干的功能模块，而不是诱人的用户体验。
- 我们拒绝他人成果，我们就不能把自己塑造成一个公司、一个团体或是一个人，而如果这样就会削弱我们的生产力、我们的智慧，更讽刺的是，削弱我们创新的能力（这有一个更前卫、更悲剧的用词，“非我发明”，即 NIH）。

为什么拉长脸

我怀疑我们公司的创新精神是因为，我们很多工作是为了提升现有产品与服务的品质，而这些正是要集思广益、博采众长才可以。痛恨微软的人称，这方面正说明了微软并不是一个创新型企业。我们接受这样的罪责，这让我很难受。

说苹果公司不够有创新性是因为其“借助于”施乐的 PARC（帕洛阿尔托研究中心）吗？说 Google 不够有创新性是因为它是以雅虎的模式发展起来的吗？丰田不够有创新性是因为它仰仗于亨利·福特的流水线及精益生产思想吗？会这样说的人就太愚蠢了，简直是种侮辱。

创新其实就是在别人的工作成果上加以改进，我们改进了我们自己出色的工作成果，也仰仗于这世界上各个领域的优秀思想成果。但是，我们却这么不自信，即使每个新发布的产品都蕴涵着创新。

或许可能问题在于我们的创新不如 iPhone 吸引人，为什么呢？

要吸引人还要完美

我们太热衷于改进我们自己个人的小聪明、小点子，却不是创造一种新的更吸引人的用户体验。你不能把一堆零零散散的软件功能推向市场，所以我们的一些个人创新就被忽视了。

当团队意识到客户价值在于设计一些诱人的用户体验时，就该转变一下我们的思路了。微软越来越多的产品及服务都遵从一种一致、完整的用户体验方案。从硬件及 Office 开始，到 Xbox 360，一直是这种做法，最后 Windows 系统（Windows Vista、Windows Server、Windows Live 及 Win-

dows Mobile）也采纳了这种思路。我们可能不是总做得这么完美，其他竞争对手也是，但是第一个新发布的产品都在昭示着一种进步。

悲惨的是，我们公司的企业文化是，个人创新要创造一种完整一致的用户体验才行，要整合这些完整顺畅的体验方案就要大家无私的支持。很少有特立独行的创新案例是来自于工程师一个人的，但是在用户看来，这些创新才是更出色的。

很多艺术家都知道，对制约的突破往往能迸发出极为独特的创意。但是，在工程师们一起致力于创新之前，他们首先必须放下一己之见，服从于大局的约束，就像交响乐演奏会的一员而不是一个独奏者。

我们最成功的案例都是通过很多年金钱与感情的投入，群策群力，才研究出符合用户要求的体验。其中一些创意是通宵达旦才得来的，但托马斯·爱迪生有句话，“天才是1%的灵感加上99%的汗水。”那些Office、Developer Tools及SQL Server团队的工程师们都极有耐心，他们不断努力，以客户为中心，一起造就了完美的用户体验，最终获得了无与伦比的成绩。但是，微软的工程师们还认着死理不放，认为独自白手起家才是成功的秘诀。

我自己来

遗憾的是，微软内部流传的神话都是某某位英雄在一个不眠之夜重写了内存保护模式。独立性、热情与对于陈规陋习的不屑都值得肯定，我喜欢微软的这些特质，但是要有一点改变了。

> **作者注：**有位读者说微软的独立性、激情及对陈规陋习的不屑只是在工程师团队内部才这样，但对于客户我们作为一个整体不是这样的。这是个很有意思的想法，将来可以为此写篇专栏。

但问题在于独立性、激情及那种不屑很可能变成“自己来干。”如果我没有这样的创意，那说明我不需要这样的创意；如果一个团队没有，也只不过会有些麻烦而已；如果微软没有，那最多是微软还没想好要不要。“非我发明”的想法不仅仅是错的、狭隘的，而且会拖我们的后腿，麻痹我们前进的脚步。

在你负责任地完成客户的需求，提升客户期望的体验的同时，你可以独立，有激情，也很不屑；你也可以在吸收其他平台的优势时，保持独立，有激情与不屑。

相反，如果你单打独斗，你的麻烦就大了：

- 生产力会下降。
- 企业的智慧会被削弱。
- 创新能力降低。

或许我可以以他人之长补我之短

如果你拒绝接受“非我发明”的成果，你就要浪费时间和精力，修复Bug，自己创造这样的成果。你会为了一些小功能，为了优化一个小程序而重写一个操作系统吗？当然不。只要用一个现成的操作系统，你就省下了大量的时间和精力。

用一个现成的库、服务、工具及方法也是同样道理。关键是，博采众长而不是通过对他人成果的艰难体验才敢保证这些成果是稳定的。一定要用第三方软件的前一个版本，而不要用最新、

最漂亮的，对于库、工具甚至技术来说也一样。悲剧是可以避免的。

> **作者注：**当要使用微软以外的库、服务、工具及方法时，我们必须尊重许可、版权及专利。一般的情况下，你会仔细研究许可或版权（你看下法律与公司事务条文就会明白），而从来不会研究专利，甚至根本就连考虑都不考虑。对这种取向我曾经感到很困惑，直到我有了一份我自己的专利认证。专利法律声明那段，也就只有这一段，根本没人能看懂，除了专利律师。忽视专利是万幸，我也强烈建议这样做。

我已经够好够聪明

拒绝接受“非我发明”的成果会导致脑力衰亡。在微软，我们聘用了最聪明的工程师，是否5年后他们依然这么聪明呢？那得看情况。他们是否每年都在学习新事物，还是他们一遍又一遍地重复着同样的工作？

如果你换了个团队的时候总要重新开发一个创建系统，你这样就更聪明了吗？如果你换了团队后还要重新开发类库、基础架构或测试用具，你是否就更聪明了？如果你换了团队后，你还要重来一遍项目管理技术、规范书模板及Bug定义，你就更聪明了吗？

不，你没有更聪明了，你变得更傻了。祝贺你，你再怎么重复自己的工作，再怎么重复别人的工作及错误，你也没有变得更聪明，你变得更聪明是因为你依托于之前已有东西。

好了伤疤忘了痛

如果忽视他人已经有的东西，不管好与不好，那将使你的创新无所适从。如我之前讲的，生产力受损——单枪匹马永远不会比依托他人成果来得快。但是，这同样还有一个重大缺陷。

如果你忽视过往的工作成果，你绝对会犯前人已经犯过的错误——这本是可以避免的，却浪费你的时间。是的，你可能想重新回味一下这些挑战，或许可能还会有一项突破，但是放明白点吧，不要犯傻了。如果你无视已有的成果，你绝对会在无意识中就重复了别人已经做过的事。你自我感觉很聪明很自豪，其实就是不折不扣的傻瓜。在此期间，你浪费了时间，结果一事无成。“非我发明”就是虚无主义，拿来主义，这才是我的意思。

> **作者注：**这个栏目广为传阅，有很多人将此误解为我倡导的是被动地而不是主动地创新。因为改进他人的工作成果对你来说并不是那么容易的，也不是说没什么风险。需要想象力、汗水及冒险才能实现突破性的创新，对于当前的技术、市场及客户深层次需求的研究同样也是这样。你并不是一个人在战斗，但是你也不能期望别人为你做好所有的事。

舍我其谁？此时不动，更待何时

微软有很多极富聪明才智、有激情和有创新力的人。我们是一个创新型公司，尽管很多时候我们并不相信。我们不该把创新看成为通过一己之力就能无中生有的，而是要实事求是。创新——是众人拾柴，重在打动客户的一项工作。

个人的观点将体现在哪里呢？它们体现在用户体验需求的大前提下，它们是在团队努力的基础上得以体现的。这样才有突破性的创新，这种创新拓宽了艺术的内涵，能让我们的客户血脉

贲张。

因此，当你将进入一个新的团队、一个新的项目，或有个新的绝妙想法，要明白怎样才使我们既定的客户体验更能动人心魄，并要知道对于你的工作别人已经有的创新。注意我说的是“别人已经有的创新”而不是“如果别人已经创新了”。没什么新玩意儿，如果已经有人干过同样的事，并有相应成果或知识你可以借来己用，那就没什么新鲜的了。要么你可以无视他人成果，慢吞吞地闭门造车，最后有所成就；要么你就借他山之石，进入下一个步骤，来一个实实在在的创新。

> **作者注：**如果你想抢先一步借助他人的工作成果，最好的去处之一是开放资源社区（内部的有 CodeBox，外部的有 CodePlex）。通过使用他人的代码，你可以随心所欲地自定义或改进这些代码或工具，而同时也可以用别人最新最好的来进行更新。你把你的改动提交到程序主体后，这么个美丽的艺术品就因你而改变了，甚至可能会超出你想象。

2010 年 2 月 1 日：“我们是功能型的吗?”

当 Steven Sinofsky 与 Jon DeVaan 进入 Windows 7 管理层的时候，他们给整个企业带来很多重大的变革。其中有两个影响深远的变革，一是建立一个单一的总体规划，二是转换到效率更高的组织机构。拿 Windows 7 来说吧，一些微软工程师会问，“如果我的产品单元经理是个蠢蛋——是该我们让他滚蛋的时候了吗?”

或许吧，但是不要太着急。膝跳反射的蠢蛋们总在他们未想清楚前就匆匆行动，这是注定要在新路上犯老错误的。万物皆有因果，在你让这些蠢蛋滚蛋之前，请想想产品单元经理所扮演的角色，以及如何才能最好地平衡产品与有效需求的关系。忘了？还是从来就没明白这个道理？我来告诉你。

> **作者注：**“你的产品单元经理是蠢蛋吗?”是我早期专栏中最常用的一句话，在本章前面也出现过，产品单元经理一般只对一个独立的功能软件负责，如 Microsoft Excel、DirectX、ActiveSync，虽然现在这些团队没有一个是由产品单元经理领衔的。

我成了产品单元经理了

产品单元经理是一种跨工种的经理，他对战略性及战术性的决策负责，这些决策对于设计微软的产品或主要组件是必要的——至少在理论上是这样。实际中，产品单元经理通常是无用的——因为上有他们的总经理，下有他们的工种经理，他们自己往往被边缘化。然而，如果对他们使用得当的话，产品单元经理通过对业务与执行决策的准确把握，就像对一个刚起步规模的团队，会给创新产品市场带入一点点的敏捷性。

产品单元经理通常出现在产品型组织结构的最底层，而根本不会出现在功能型组织结构中。功能型组织结构及产品型组织结构的企业架构模式并不新鲜——这两种模式的现代起源在 20 世纪 70 年代就开始了。其中是否有一个“正确”的选择呢？让我们来仔细讨论一下。

你是干什么的

功能型组织结构，如 Windows，是分工种设部门的。每个工种都有一个主管（director）向部门总监（division president）负责（如开发主管或测试主管），在这个部门中，没有跨越多工种的主管。

产品型组织结构，如 Windows Server，是按产品分别设置的。有一帮的 GM 向部门 VP 负责，而这个 VP 负责一系列相关的产品。每个 GM 都有一群产品单元经理下属，由他们来各自负责各自的产品。每个产品单元经理都有一群工种经理向他负责。本质上看来，在产品单元经理层级，产品型组织结构就变成了功能型组织结构。

另一方面，要创建产品，工程师就需要跨工种合作，特别是功能团队。因此，这两种组织结构模式都是“混合型组织”——人们跨越功能界限在一起合作共事。

依以上说法，你可以也可能会这么想，“等会儿，我知道的所有公司都是自上为产品型组织结构，自下有些地方就换成了功能型组织结构。”是的，没错。即使是新生代的、前卫的、很酷的以及根本就不起眼的公司他们自上也是产品型组织结构，自下换成为功能型组织结构。问题的根本在于，在什么地方开始转变。Windows 是在部门总监层级开始转变的，Office 是在 GM 层级转变的，Windows Server 是在产品单元经理层级转变的。

> **作者注：**小型的团队中通常有些人扮演多工种角色，所以在最底层功能性对他们来说是看不到的。是的，我的意思就是自我管理的团队会选择摒弃专业分工，从头到尾每个人都干着所有的事，这是跟产品型组织结构最接近的一种模式。

换个管理模式

但在哪个层级组织结构应该从产品型转为功能型呢？理想主义的新手可能会说公司结构应该是扁平的——自上是功能型，或者，最好是，一个产品型组织结构只有一个头儿，在其下面是一帮可互相转换角色的员工，就如一个群居型（crowd-sourced）工程项目。毋庸置疑，这是个美丽的梦，在这里人们分享财富，相互嘉奖，和平共处，永远和谐繁荣。很自然，这是不可持续的。

问题是对业务缺乏管理。当你接手一个要管理 150 个人以上的项目时，你必须建立一种管理机制使每个人朝着同一个目标，按同一日程安排开展工作。有些公司，如 Gore（Gore-Tex 的创始者），它拥有 150 个以上的员工，却没有任何特别的组织架构。然而，你需要一个大型组织团队来创建综合性的操作系统（OS）、一整套的应用程序或是完整的服务。

每个产品都需要有一个明晰的业务趋向，一个远景，这可以是为将来的产品设定一个目标（主题及案例），为其开发的背后设定一个基本的机理（原则），以及为发布产品设定一个日程安排（战略性计划），这同时有助于理解如何达到这样的目标及希望达到怎样的目标。理想情况下，组织中的每个人都是实现这种计划的一员，但有一个人既要协调这一安排又要为其负责，这个人就是业务主管。

在业务主管以下，你希望每个人都服从这样的业务意图。你并不需要更多的业务主管——你只需要一个人组织管理这个机构来执行计划。那在哪个层级，组织结构要从产品型结构转换为功能型结构呢？在业务主管层级。Windows 是一个单独的业务项目——我们不会将这个产品分块发售。这好像是说，自 Sinofsky 以下就是功能型。而 Office 是分模块发售的，这意思就是说 GM 们拥

有这些模块的管理权，而自 GM 以下的组织结构为功能型。

> **作者注：**但 IE 却是与 Windows 分开发布的，猜猜为什么？因为它有一个 GM。

走自己的路

为什么不在业务主管的下面设一个业务分主管呢？因为你不需要多业务的计划——你要的是单一的业务远景。业务主管不需要自己管自己，他们应该做的是设定业务方向，制订计划，做出业务决策，毕竟，他们除了对员工的管理没有其他任何的责任。

> **作者注：**在业务主管下设功能型组织结构还有很多其他好处：
>
> - 各工种主管不需要放弃他们的管理原则，而可以单独作为一个产品单元经理不断成长。这意味着最高级的工程师要解决的是项目工程问题而不是人员、行政或业务问题。
> - 你的组织结构会更扁平化而不需要另设多工种管理层。这就意味着管理人员要少得多，由上而下或由下而上要重复解释或误解的事情要少得多。
> - 组织结构变得更稳定，有更清晰的角色定位与责任，这样可以让人们更方便地一对一进行共事，而不是层层级级上下传达汇报（你知道你的同事是怎样的人以及他们的角色定位是什么）。
> - 你的员工们会有固定且熟悉的指导老师，因为高级工程主管就在功能团队中。
> - 在整个组织中，你可以更方便地分享经验、代码及工具（而不用面面俱到）。

这是否意味着有了单一的产品远景规划后就很难有创新与自主的空间了呢？哦，饶了我吧。我听这些听得都快吐了。产品远景不是一份功能规范书或设计文本，如果这样的话，微软全部的工程师差不多有 125 个人了，还要有巨量的零售商。

要完成以此远景为本的主题与案例，还能按既定的原则与计划完成，这样的方法成千上万。远景只是提供了一种意向——还有很多空间留给灵感。可能一种案例可以有一种新的完成方式，不复杂也不费时；可能有一种原则可以在今后的几年省下上亿的美元；或许可能只用采纳一下客户的意见建议作些稍许改动，却可以讨得某类用户的欢心——而有些人放弃了这样的改动或没有勇气这样做。

没错，功能型组织结构只有一种业务计划，所以大型的功能型组织结构不能在一夜间快速转变他们的业务计划。这对于如 Windows 这样大型的产品来说是适当的，但是对于那些快速变更的领域就不再适合了。你要选择适合你的业务与市场的最佳模式。

> **作者注：**有人可能会尖呼，“但那些平板设备及 App store 呢？Windows 已经落后了，因为它反应太慢。”有时候我们对竞争对手了不起的创新反应有些措手不及，如 app store 及苹果触摸设备，或任天堂的 Wii 无线遥控器。但是我们明白不能武断鲁莽回应，相反，我们要先研究存在的问题，深入理解这个重大机遇，并以我们自己最具创新力的产品予以回应，如令人惊艳的 Windows Phone，她让我们的日常生活大大简化，还有 Kinect，她的用户界面已经完全改头换面。可以期望，微软会继续给大家带来意想不到的产品。

不管你的组织结构是怎样的，在微软有一样东西你一定要有，就是选择与工作的自主权。并不是所有人都利用了这种自主权的好处，人们总把组织安排与限制搞混，如前所说，当有限制的时候，艺术家、架构师及创新想法会发挥出最大的效力。在一个既定现实世界中创造新鲜玩意儿才是真正的挑战。

作者注：提倡个人自主的人忘了这几年微软正身陷麻烦，这常常让我们效率低下，这也让“非我发明”的风气滋生，每个人都重复做着相同的工作，因为大家对利用他人的成果缺乏动力。不过，权衡之下我更喜欢自主，这种信任与微软想通过软件改变世界的设想紧密相关，也让微软独具一格。

寻找一种合适的杂合体

要启动一项大型的项目，你需要有基于产品与功能的领导团队及决策，这种权衡决定了成功与否。太注重功能分工，你就失去了产品吸引力；太注重产品你就缺乏客户黏性、效率及资源的灵活性。

一个成功的工程组织方法是在产品定位与功能型组织相一致的层级上将产品型组织结构转换为功能型组织结构。部门经理应该依业务策略来决定这个层级在哪。

如果我们将 Windows 分成很多模块发售的话，产品单元经理对于 Windows 开发的组织架构来说是很有意义的。但是，如果由我们的合作伙伴各自为政为我们的客户提供成千上万的解决方案，那 Windows 想要成为一个综合性集成平台就歇菜了。所以，我们把 Windows 当成一个单独的平台发售，它用的是一个功能型组织结构。

你的团队应该采纳 Windows 的模式吗？这要依你的业务与市场而定。谁该为业务决策负责呢？在哪个层级要做出这样的决策？你应该将你的组织结构与你的业务策略作对照，如果你的模式不匹配，或许真是该叫这些蠢蛋滚蛋的时候了。

作者注：微软还远远没有达到完美，我们还有很多的工作要做，我也乐于成为解决方案的一分子。我也很高兴我能得到信任来做评述，微软是以公正与忠实的心态出版这本书的，谢谢你能花时间阅读，我真切希望这对你来说是很有趣也是很有帮助的。

术 语 表

2.5，3.0，3.5，4.0，4.5（也称为评分、考核系统、考核曲线） 微软老一代的打分系统，在2006年春天已经做了改变。2.5分和3.0分是不受欢迎的。4.0分和4.5分是让人高度向往的。3.5分是容易让人接受的，而且绝大部分人都得这个分数。

Bill G 或 Bill B Gates（比尔·盖茨） 微软公司首席软件架构师和董事会主席。

black box testing（黑盒测试） 把产品当做一个不透明盒子的一种测试。你不知道产品内部的工作原理，也不需要那方面的知识，你只是像客户一样去使用它——正常使用或滥用，直到发现不对的地方。

Bohrbug 一个可预期且重复发生的软件问题（bug），它跟不可预见的随机的Heisenbug不同。术语Bohrbug与Heisenbug可追溯到吉姆·格里于1985年发表的论文："为什么计算机宕机以及我们该怎么办？"

BrianV 或 Brian Valentine 微软核心的Windows部门的前任高级副总裁。

buddy drop（密友交付，也称为私有建造或密友建造） 某个产品的一个私有建造版本，用以在源代码签入主代码库之前验证代码改动。

Bug 或 work item（Bug 或工作条款） 在微软内部，我们把在产品上增加、删除或修改的任何东西都称为Bug，而大部分人通常称之为"工作条款"。很自然，这包括了代码方面的错误，也就是传统上我们认为的那种Bug。

Build Verification Test（BVT，建造验证测试） 检验一个软件建造是否满足特定的一组要求。

calibration（薪资分类） 是微软用以对薪酬等次进行区分的一种方法。它从职业发展阶段及职业角色可比性的角度出发，对一个团队的预期成果及特别成果进行评定。

CareerCompass（职业罗盘） 是一个微软内部使用的Web应用程序，它可以让员工在他们的职业发展阶段上参照一定的标准对他们的能力及技术进行评价。

Career Stage Profile（CSP，职业阶段剖面） 在不同的职业发展阶段对不同工程领域员工工作期望值的具体描述。它也概括了个体劳动者和管理者各自的成长路径。

CodeBox（也叫 Toolbox 或 CodePlex，代码箱，也叫做工具箱或代码丛） 一个工具和代码的共享库。代码箱是内部的一个代码共享库。工具箱也是内部的一个共享库，不过主要偏重于工具和脚本。代码丛是面向外部的代码共享库。

code complete（编码完成） 开发者认为对于某个功能所有必要的实现代码都已经签入到源代码控制系统的一种状态。通常这只是一个主观判断，而更好的做法实际上应该基于质量标准来度量（常称之为"功能完成"）。

dogfooding（also known as "eating your own dogfood"，吃狗食，也有人说"吃你自己的狗食"） 在日常工作中试用未发布的产品。它鼓励团队从一开始就把产品做好。通过这种试用行为可以收集对产品价值和可用性的早期反馈。

external bug 或 external（外部 Bug） 在本团队不具有所有权的代码中发生的一种 Bug。这些 Bug 不应该被忽略，除非有一个简单的迂回方案绕过它。

feature（功能模块） 可以逐步为产品添加功能特性的独立功能集合。尽管功能模块可能比较庞大，理想情况下，功能模块会被分解成小块的工作，每块工作的设计、开发和测试都不会超过 5 天的工作量。

feature crew（功能小组） 一个小型的跨领域工作团队，他们负责单一的功能，或者密切相关的几个小功能；他们自始至终都在一起工作，包括设计、写规范书、开发以及测试。典型情况下，功能小组是一个虚拟团队：并不是所有的团队成员都要向同一位管理者汇报。

Heisenbug 一种不可预期的随机的软件问题（bug），与可预期的重复发生的 Bohrbug 不同。术语 Bohrbug 与 Heisenbug 可追溯到吉姆·格里于 1985 年发表的论文："为什么计算机宕机以及我们该怎么办?"

informational（信息交流） 一种信息交流过程。通过它你可以在应聘一个职位前与人事经理进行需求沟通。

milestone（里程碑） 也就是"项目日期"，被相关组织（50～5 000 人）用来同步工作和复审项目计划。"里程碑"这个术语也用来指在两个里程碑时点之间的工作时间。里程碑的周期设定因团队而异，也因产品而异。典型情况下，一个里程碑跨越 6～12 周不等。不要把"里程碑"叫做"迅步"，那样对两个术语都有不利。

PREfast 或 Code Analysis for C/C++（PREfast 或 C/C++代码分析） PREfast 是一个 C/C++ 编程语言的静态分析工具，它能识别出可能导致缓冲区溢出或其他严重编程错误的可疑代码模式。尽管这个工具最初只是内部使用，但它最近已经集成到了微软的 Visual Studio 2005 中。

product unit manager（also known as PUM，Group Manager，Director，产品单元经理，PUM，也称为项目组经理、总监） 微软内部管理结构中的第一级面向多领域的管理层。典型情况下，产品单元经理负责一个独立的功能集合，比如 Excel、DirectX 或 ActiveSync。

program management 或 program manager（PM，项目管理或项目经理） 这些工程职位主要负责详细说明最终用户体验，包括决定何时发布那些用户体验的项目全局时间表。

project 或 release（项目或产品发布） 为了发布一个产品的特定版本或者补丁程序包所需的全部工作。

RAID（相关术语还有 Product Studio、Bug 数据库、工作条目数据库） RAID 是一个数据库，通过客户端程序可以跟踪工作条目，内容包括功能实现方面的工作、Bug 报告和设计变更请求。

reorg（重组） 典型情况下，重组首先发生在高层，然后在 9～18 个月内向下扩展开来。

RDQ 或 PSQ 工作条目数据库查询语言，用于确定某个项目的工作状态。

scenario（示例或范例） 最终用户对某个任务操作过程的示例，这些任务未必已经在当前产品中得到了实现。一个示例通常会涉及对多个软件功能模块的使用。

software development engineer（SDE，软件开发工程师） 也就是软件开发者，指的是编写代码、构建客户体验的那些人。

Source Depot 或 source control（代码库或代码控制） 微软内部使用的大规模代码控制系统，用于管理上亿行代码和工具，包括版本控制和分支管理。

specification（spec，规范书） 用于详细说明产品的用户体验，以及产品如何被构造、测试或

部署。

SQM（软件质量度量，客户体验改进计划） SQM 是一个内部技术名字，它通过匿名收集用户对产品的使用模式和体验，来支持 MSN、Office、Windows Vista 和其他产品的用户体验改进计划。（请你在安装我们软件的时候加入这个计划，它会让我们知道哪些让人满意，而哪些不尽如人意。）

Steve B 或 Steve（史蒂夫·鲍尔默） 微软公司的首席执行官（Chief Executive Officer，CEO）。

STRIDE 是一种记忆策略，用以帮助人们记住不同类型的安全威胁：哄骗（Spoofing）、篡改（Tampering）、抵赖（Repudiation）、信息泄露（Information disclosure）、拒绝服务（Denial of service）和提升特权（Elevation of privilege）。Michael Howard 和 David LeBlanc 合写的《编写安全的代码》（微软出版社，2002 年出版）一书对此有详尽的论述。

Test-Driven Development（TDD，测试驱动开发） 是一种敏捷方法，开发者先写测试代码，后写实现代码。

Toolbox（也叫 CodeBox 或 CodePlex，工具箱也称之为代码箱或代码丛） 指一种工具和代码的共享库。工具箱是内部的一个共享库，主要偏重于工具和脚本，而不是代码。代码箱是内部的一个代码共享库，代码丛是面向外部的代码共享库。

triage（分诊，也称为 Bug 分诊或问题管理） 在项目开发周期快要结束的时候定期召开的、对相关问题进行分析的一种会议。典型情况下，这些会议的参加者来自于 3 个工程领域（规划管理、开发和测试）的代表。

Trustworthy Computing（TwC，可信计算） 微软在安全、隐私、可靠性、正确的商业行为方面发起的研究课题。

User Experience（UX，用户体验） 微软设了“用户体验”这个职能，主要包括设计师和可用性专家。

Watson（同时也称为 Crash Watson 或 Windows Error Reporting） Watson 是一个内部名称。在 Windows 上运行的应用软件出现灾难性错误时，你将看到一个错误报告对话框，其背后的机理就是 Watson。

Watson 桶每个“Watson 桶”代表并存储着一个客户问题，这些问题已经由成千上万个用户经历过。微软内部及外部的工程师可以查询哪些桶来自他们自己的软件问题。

white box testing（白盒测试） 通过使用测试工具，自动并系统性地对产品的各个特性进行测试。微软坚决地使用白盒测试代替黑盒测试。

Zero bug bounce（ZBB Bug 零复发） 指一个项目的所有功能已经完成且每个工作条款都已解决的那一时刻。这个时刻很少能保持长久。通常在一个小时内，通过一次系统扩展测试会出现一个新问题，从而工作团队又得回头工作。不过，ZBB 指这种终点是可以预期的。