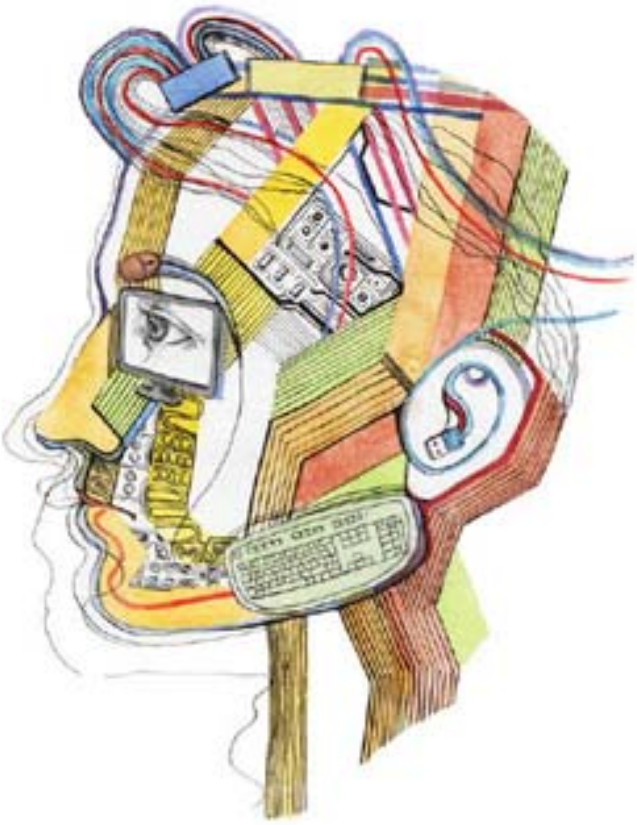
华章科技

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

InfoQ软件开发丛书

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 Email：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 punches）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。

塗　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

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

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[译]。

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

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

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 PepcidTM（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：

- 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。
- 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。
- 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。
- 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。

- 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。你在耍我呢？”

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：比五香熏牛肉鋈还好”

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。

> 鎏五香熏牛肉（Pastrami）是一种加入各种各样的香料熏制成的牛肉。——译者注

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“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 Email。

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。

Email：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。

请参阅本书第 5 章的“复审一下这个——审查”栏目。

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。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。真是太捧了！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 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 XX

温度 XX

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——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”（AxiomaticDesign）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 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。