站在巨人的肩上筑梦
解析微软成功之路

Windows 2000
开发组总负责人
倾力之作

微软经典

微软研发致胜策略

Debugging The Development Process

(美) 史蒂夫·马魁尔 著
苏斐然 译

机械工业出版社
China Machine Press

Microsoft Press

致　谢

有太多太多微软出版社（Microsoft Press）的同仁们，虽然我不知道每一位的名字，但他们贡献的心力让许多信息从稿件变成了书本，让读者们分享，实在功不可没。首先谢谢 Mike Halvorson（他是这本书的主要支持者）和 ErinO ’ Connor（她是本书的手稿编辑，和我一起工作了将近一年，合作非常愉快，这本书也等于是她的）；Jeff Carey 对本书内容的热烈反应使我受到鼓舞；Deboran Long 一流的文字功力让我省掉无数的时间，还有其他担任审校和编辑的同仁，Alice Smith、Therese McRae、Pat Forgette，让我可以心无旁骛地写作；负责排版的 Peggy Herman 不厌其烦地一再尝试不同版面，务求最佳效果；Kim Eggleston 调整她为《零错误程序》所做的漂亮设计以求切合本书的旨趣；还有 Judith Blocn、Wallis Bolz、Barb Runyan、Jeannie McGivem、Sandi Lage、Shawn Peck、John Sugg、Geri Younggre、DeamHomes 等多位微软出版社的同仁，多亏了你们辛苦的工作。

我想在此特别感谢我的两位良师 Anthony Robbins 和 MichaelE.Gerbe，他们指导我如何带领团队，对我影响很大。Anthony Robbins 常主持训练高级主管有效推动业务的研讨会（也许他的个人成就训练班更让人耳熟能详，那也是很棒的课程），对 Anthony 熟悉的读者应该可以在本书中看到他的影响力；Michael E.Gerber 发表过“ 为美国企业带回梦想 ” 的演讲，也写过《E 的传说：企业失控与拯救之道》一书，虽然似乎与软件开发没有一点关连，但是 Gerber 有不少见解完全改变了我对软件项目本身、以及如何执行软件项目的看法。

我非常幸运，能和这么多优秀的同仁一起工作，并完成这本书。他们慷慨地将经验和见解让我分享，他们大都曾担任过多年的程序设计师或项目经理，完成过许多策略性项目。他们是本书的审阅委员，以保证本书的建议都能切中时弊，是正确的解决问题之道。他们是 Paul W.Davis、Melissa Glerum、Eric Schlegal、Alex Tilles；在这里我还要特别感谢 Dave Moors，他以多年担任程序设计师、项目经理、开发部总监，以及最近的总裁经验，为本书润饰。同时，还要感谢 Ian Cargill 在本书撰写初期所提供的宝贵意见。

史蒂夫 · 马魁尔

作者序

这本书是发生在微软的真实故事，也许您读过之后会觉得微软真是糟。

至少这是我在写本书时的疑虑之一。我曾经考虑要不要把一些地方轻描淡写带过，或是干脆跳过不写，但是最后我决定除了人名都换掉以外，一律老老实实地保留所有事件的原貌，好让这些故事对读者更加实用。除此之外，我相信读者不会因为本书就看扁了微软，微软既能够在产业中独领风骚，当然不可能都是笨蛋在当家。

本书大部分的事例，都是我在微软负责为遇到麻烦的团队实施在职训练时亲身经历过的。所谓“问题团队”，就是进度严重落后、程序的品质无法达到公司要求的水准，或是程序设计师一天到晚拼命工作而成绩却极为有限的团队。从与这些问题团队的相处，我很明显地感觉到，他们都在犯同样的错误，而且一再重蹈覆辙，不仅如此，我还发现连那些做得很成功的团队也犯着同样的毛病，只不过发生得比较少、或是及时弥补过来罢了。

每一次研究团队的问题，我总是发现项目经理几乎不去思考项目本身，而只管写程序。项目经理不去注意控制进程，不去预防可能发生的问题，不去保护组员免受杂事干扰，不去把项目目标进一步详细划分，也不去拟定创新或积极的计划。总之，项目经理应该去思考，但却把时间花在做事上了。

其实说起来也不是项目经理的错，他们不是被训练来当主管的。他们原本是程序设计师，某天忽然不知怎么的变成了主管，他们知道如何把程序写得呱呱叫，但不知道如何带领团队，只好继续做他们最专长的事情——写程序，然后让项目自生自灭。

很不幸的是，大部分的程序设计师根本不认为自己需要知道如何推动项目：“我又不是项目经理，何必考虑这些？”他们认为真的当上了项目经理再说。可是到时候再学就有点太迟了。

我写的另一本书《零错误程序》（Writing Solid Code），是把被证明有效的技巧与策略提供给程序设计师，教他们如何使程序错误变得更少。而这本书则是写给项目经理和程序设计师的，它告诉您如何让项目顺利进行，避免进度落后、拼命加班等软件产业中常见的苦难。

要想如期推出高品质的软件，又不需长时间加班，并能做得有声有色有乐趣，这不是梦想，本书所提供的技巧和策略就能帮您做到。

译者序

您是否经常听说“某某软件的推出时间将比预定延后……”？这可能是您在报章杂志上看到的讯息，它告诉您，你还得再等一段时间才能用到新版的产品，也可能，您就是这个项目的领导人，正在为进度延误的问题头痛不已……或者，您正在使用的软件出现了某种怪异的现象，把您辛苦工作的东西全毁了，您最多只能打电话给软件公司的总裁大骂一番，但就是美国总统也无法把您的东西还原回来……

自电脑问世以来，软件开发项目要如期完成、软件的品质要够好，就一直是所有开发团队的梦想，也一直是大部分开发人员的梦，因为它实在太难达到了。最大的原因，当然是软件本身的性质，它是无形的、错综复杂的智慧财产；第二个原因是它几乎没办法由一个人独力完成，必须凭借非常密切的团队合作；还有许多其他数也数不完的原因，不幸的是，这个问题没有公式、没有解药，甚至没有绝对的答案。

本书为所有的软件开发人员，回答这个软件开发领域最大的问题：“如何在限期之内，完成高品质的软件”，更理想的是人员不必超时工作。作者是软件的资深项目经理，曾经参与过无数重要的项目，现担任教育训练的工作，负责辅导微软内部的开发团队，教他们的项目经理该如何领导组员、程序设计师应该如何发挥最高的工作效率。本书正是作者多年工作心得的精华，在软件界，任何人要升项目经理之前，本书是必读圣经。由于作者担任教育训练工作，本书结构完整，深入浅出，读起来特别轻松，而且其中穿插软件的真实故事当做实例说明，让人印象深刻。

虽然作者是微软人，说的都是微软事，但对于任何一位软件从业人员，本书的经验都很值得分享，作者并不歌功颂德，而是真真实实地道出软件开发所有的秘诀，作者对于任何问题都力求抓住重点，直指核心，可以说是从实务面出发的一本软件开发教学手册。

中国虽然纯粹的软件业比较少，但是很多 MIS 部门的经理深受项目难以控制的困扰，也有很多承揽项目的系统整合业者，面临项目无法了结就拿不到尾款的压力。不论是不是软件公司，只要是设计软件的人，多少都尝过项目落后的痛苦感，应该都能从本书中找到一些解答。东方文化中的主管，比较不敢放手让员工发挥，比较爱开会、讲究规矩程序，对属下的工作成绩也倾向要求多于协助和鼓励，本书对于软件从业人员，应该是一个新的视角吧。

衷心期盼本书能带给读者新的思潮，帮助更多的软件从业人员，促进软件业的蓬勃发展。

苏斐然

1999 年春天

出版者的话

微软出版社（Microsoft Press）是由软件业的巨擘——微软公司（Microsoft Corporation）于 1983 年成立的出版社。自成立至今，微软出版社每年约出版两百种的新书与多媒体产品，从一般使用者的学习教材到专业信息人员的参考书籍都有。而《微软经典》是一套关于软件产业经营管理的书籍。这些书籍的内容分别从为整个软件产业、微软公司内部、各种经营或管理角色扮演等不同的层面，来详细说明整个微软的工作模式与管理风格。这有别于以往由第三者的立场或角度来阐述微软。这一套丛书的作者都具有丰富的实践经验，有些现在仍是微软的资深员工，从他们的亲身经历现身说法，保证让您可以真正一睹微软的庐山真面目。

微软是举世公认的计算机软件巨人，他所制作出来的产品，不论是操作系统、商业应用软件、服务器软件，甚至一些计算机外设产品，例如符合人机工程学的键盘、鼠标等等，都广泛地获得使用者的喜爱。在这些风光的背后，微软是如何制作出这些产品的？他如何去了解消费者的需求？如何根据消费者的需求来设计产品？在产品制作过程中如何照进度进行？如何创新产品的功能？如何创造出新的技术继续保持领先？我们都知道：信息技术的进步一日千里，而信息产品的更新周期也越来越短。如何在这瞬息万变的信息产业中占一席之地，这是所有信息从业人员最关心的话题。

“罗马不是一天建成的”，微软的成功经验，绝对是我们一个绝佳的学习典范！

推荐序

就在 1999 年 4 月初，IDC 发布对世界各主要国家和地区关于 1998 年信息社会指数 ISI（Information Society Index）所做的研究报告结果。这份报告指出，美国的信息国力仍保持世界第一的地位，而中国的排名则居于世界第 53 名。

事实上，IDC 对于信息国力的评比指针主要分为计算机基础建设、信息/通信基础建设及社会基础建设三大项。换句话说，信息国力的重点不在于一个国家的信息产值有多少，而在于一个国家于信息应用的投入到达何种程度。

比尔·盖茨在其《数字神经系统》一书中，一开始就强调，过去 80 年代、90 年代，企业成功的因素是品质、是企业再造；而展望未来，企业胜出的关键则是“速度”，这一点从它的书名以《与思考等快的明日世界》（Business @ The Speed Of Thought）为副标题即可看出。

数字神经系统并不是一项新的科技架构或是新的软件代号，它所强调的是一种数字基础建设，除了在硬件方面，将个人计算机透过网络连结起来之外，更重要的是利用目前已经成熟的网络技术来建造企业的管理模式。

未来，企业的成长与淘汰都取决于速度，企业中的每一个工作者都要能够适度地“决策”，如果所有的决策都还要经由层层的关卡才能决定，这样的企业岂能取得速度上的优势？因此企业的工作者必须能在正确的时间、正确的位置撷取正确的信息，以帮助他们做出正确及关键的决策。这就是建立数字神经系统的重点所在。

既然企业建立数字神经系统是如此重要，面对 21 世纪数字时代的来临，企业该如何适应整体经济环境的冲击？企业若仍因循过去的管理经验及技术，在面临未来以信息为管理基础的竞争时，这样的方式将无法满足企业的需求，企业需要的是以未来竞争为根本的数字神经系统。

软件与硬件其实只是实现数字神经系统的材料，真正实现数字神经系统的是企业里的管理团队及信息部门。如何驱动和改变企业内部这些重要成员的动力，发挥其效率将会是数字神经系统能否成功的关键。

除了需要具备信息技术相关知识之外，信息团队的领导者或管理者更需要好的方法与经验。就像企管系的学生在学校中学习管理知识之外，案例研读除了可以帮助学习者更了解这些知识的实质内涵，更可以在未来多变的环境中，拥有更强的应变能力。而这些具典范价值的经验与方法何处可寻呢？在《微软团队：成功秘诀》、《微软研发：致胜策略》以及《微软项目：求生法则》这三本书中，经由微软自己企业发展过程中的经验与法则，可以获得最佳的解答。虽然这些内容是架构在微软软件开发的特质之上，但却都是“放之四海而皆准”的知识与原则。

当你的企业正彷徨于未来信息时代的挑战，而没有胜算的把握时，相信这几本书将会是帮助你前进的最佳材料。

李开复
微软中国研究院院长

前言

卓越的领导者从不同的角度看世界。若是公司被大火烧得精光，他非但不为丢饭碗惊慌，反而利用火焰来烧烤一顿大餐。当每个人都摇头离去，卓越的领导者仍有充分的信心保持乐观，对每件事都从正面角度来思考。就因为凡事都看光明面，卓越的领导者并不把失败当失败，反将其当作学习克服障碍的经验。正因如此，卓越的领导者乐意尝试各种稀奇古怪的想法，并从中获得重大的突破，即使不成功，他只把这次经验当成获得信息的方式之一。这种领导人不一定要有经验，而是需要强烈的进取心和明确的理想，能够将理想与他人沟通，鼓舞他人共同追寻理想的能力，再加上一点机会，这就是能将理想实现的卓越领导者。

也许有人认为这种领导者是天生的，其实，任何人都可以学习做个卓越的领导者，不过并不容易，他必须学习排除成见和习惯，从各种不同的角度思考问题。你可能会说，那不就等于彻头彻尾换了个人？那几乎是不可能的，就算做得到也是一种虚伪和矛盾。我想这就是大多数人无法成为卓越领导者的原因之一，人们大都不愿意将自己改变到这种程度。

一般的领导者

幸而大部分的软件开发主管不需要是卓越的领导者，不必找一片处女地开疆拓土、披荆斩棘，开一家打败传统的新公司；大部分的软件开发主管只要把某个软件的新版本开发完成就可以了。软件开发主管大都已有明确的目标，项目内容是每位组员基本上都同意的，不必彻底改变传统或是鼓舞人们去做极困难的事；基本上软件开发主管最重要的任务就是提高工作效能，这是可以学习的，不必彻底改变自己的性格，只要懂得开发软件的技巧和策略，就可以让软件叫好又叫座，而且不必每周工作 80 小时。

追求效能的主管很明白，成功的项目维系在每位组员身上，才能如期完成高品质的软件。本书所提供的技巧和策略，不是软件开发主管会用就够了，每位组员都必须充分了解。除非每个人都懂得如何在合理的工作时间内做出高品质的软件，否则项目仍是不会成功的。

写出好的程序

软件从开始构想到开发完成、上市，这中间要经历的步骤非常多，每一步骤都可能会出错，这已经没什么稀奇了。本书希望程序设计师试着把开发程序想成写程序：开发程序像程序一样可能有错，程序有错误是一定会造成伤害的，最后得浪费非常多的时间和精神去找出错误，所以最好是第一次就把它做对，或者说，愈少错误、愈高效能。

本书的重点是提高软件开发的工作效能，将人力的浪费减到最少，并兼顾软件的品质。前 3 章主要谈的是让团队不必大量加班的基本观念和策略，后面 5 章是慎防流程的僵化、开发进程的掌握、开发人员的训练、正确的工作态度以及长时间工作的症候群上。

简介微软的软件开发制度

本书大部分的例子都是我在微软的工作经验，因此先了解一下微软的软件开发制度，会比较容易抓住本书正文的内容要旨。

微软的开发项目一般都会包括三种不同的主管，其中：

◆项目经理（ProjectLead）：他是项目的主要负责人，同时负责拟定进程，监督工作确实按进程进行，确保所有的工作都走上轨道，不出纰漏，训练程序设计师，负责向高级主管报告本项目的状况。通常是由团队中最资深的程序设计师担任，偶尔他也写点程序，但那是次要的工作。

◆技术经理（TechnicalLead）：技术经理是团队中对程序最熟悉的程序设计师，负责软件内部的整合性，确定所有的开发活动都符合设计规格，而且没有互相掣肘，他通常也负责让技术文件都确有更新，包括档案格式、内部设计图等等。通常也是由团队中最资深的程序设计师担任。

◆程序经理（ProgramManager）：程序经理负责与行销人员协调，使得产品的开发、文件、测试与顾客支持等事宜能配合行销方面的动作。简言之，程序经理的工作是监督每件事都确实做到，而且做得符合公司的期望。程序经理还常和产品支持小组共同合作 Beta 测试的种种事宜,并根据最终使用者的反应，研究产品如何改善。程序经理也可以是程序设计师，但是他们写程序的工作很少，而且仅限于产品的宏语言（如果有的话），或是像精灵之类的东西。程序经理是对产品未来适用性的主要负责人。

程序经理的原文是 Manager，听起来好像比较大，事实上三种经理角色是不分大小高低的，也许更正确的名称是“ 产品经理 ”（ProductLead），因为他的责任是使整个产品（而不只是程序）要跟上进度，而且要保持良好的品质。

在一个典型的项目中，程序经理（如果这个项目规模比较大，会有不只一位程序经理）要带头与行销、开发、产品支持等小组合作，共同拟出一张清单，上面列着本产品可以改善的地方。然后，程序经理着手撰写产品规格，详细描述每个功能要如何具体展现，包括细节的执行步骤；比方说，决定要开一个新的对话窗，那么产品规格中必须绘出这个对话窗的模样，用文字描述它如何操作，能引发什么功能等等，如果要加一个新的子程序或宏，就得把它的所有参数都定义好。产品规格定好后，必须给所有相关的工作小组复审，完全确定所有的细节后，开发小组才正式开始工作。

在拟定产品规格的同时，程序经理还必须进行一些“ 使用难易度研究 ”（usabilitystudies），确定所有的功能都跟想像中的一样容易使用，没有始料未及的障碍。如果实验结果是有些地方操作上怪怪的，或是容易引起使用者误解，程序经理就得提出改进的建议。当然，这些操作的环境、范例资料、相关文件等等，程序经理都必须事先准备妥当。最后，程序经理要对每项功能或特色逐一审查，特别是对那些改变幅度较大的更要仔细，完全确定产品规格能够符合项目的目标，产品的规格才算完成。

开发工作进行到比较后期时，会进入一个“ 视觉冻结 ”（visualfreeze）的阶段，意思是使用者界面就固定不动了，这样做的目的是要让使用手册等文件能够定稿。所以从这时候起，开发动作要特别小心，各个画面及其彼此的逻辑关系都不能再受到影响，这样手册上的画面才会跟实际执行的画面完全一致。程序设计师当然希望程序全部完工后再来排画面做手册，但是手册的编撰需要比较长的时间，还要排版印刷等等，为了让软件推出时手册也同时就绪，“ 视觉冻结 ”是绝对必要的措施。所以，在“ 视觉冻结 ”以前，一定要把画面确定，功能尚未齐备的部分稍后再进行。

一旦所有的功能都完成，软件就进入了“ 程序完成 ”（codecomplete）

阶段，意思是程序不再作功能上的修改，只要进行抓错除错（debugging）和必要的改进。等到产品确定可以推出了，项目经理或技术经理负责准备好“母片”（goldenmasterdisk），也就是即将大量复制的原型，和手册登录卡等包装成盒，再做好出货的登录号等管理工作，这个产品就可以交给使用者了。

我略去了许多细节，以上所述仅只是让读者有一个基本的概念，让读者比较能理解正文中所提的案例，不然这些案例会让人觉得隔靴搔痒。

还有一点我必须提到的，就是电子邮件是微软内部沟通的命脉，电子邮件让我们工作时不被电话打扰，这在开发人员来说尤其重要。开发人员彼此之间大部分的讨论也通过电子邮件，只有必要时才开会。微软这种防止干扰的做法等于让每个人都有了一间私人的办公室，如果你想专心工作，不要任何干扰，把门关上就行了。

说来容易做来难

我最后要提醒读者一点，本书也许会让您觉得，只要照书上的每一个建议去做，就能立刻将一个濒临失控的项目起死回生。当然本书中有些建议是能立竿见影地看到效果的，但有一些技巧和策略则是需要时间的，比方说训练方法就是。所以，如果您的团队遇到问题，您不能期望照着本书做就可以在一星期后让团队改头换面。以我的经验，让项目起死回生大概要二到六个月，前两个月会有大幅的改善，而以后是比较缓慢而渐进的改善。

第 1 章　奠定基础

您是否曾经暂时停下手边的工作，思考一下如何能使项目的进行更有效率？您想到的是需要高深学问才能解决的方式呢？还是只要利用一些经验法则就能化腐朽为神奇呢？

但愿这个答案像猜谜游戏一般简单，这样我的训练工作就容易多了，可惜事实不然。要让项目提高效率，需要长时间、一点一滴地累积非常多的知识、技巧和信念，尤其新手更是如此，不幸的是，这种能力的培养需要很大的耐心和毅力，而大部分的人都是用尝试错误的方式来学习，这样代价既高，功效也不大。

尝试错误的方式会耗用很长的时间，可能得到经历很惨痛的教训，如果能善用前人的经验和智能，学习前人已经归纳出来的知识，避免犯下同样的错误，不就快得多了吗？

本章首先来介绍前人的经验。就我所知，对于一个希望不加班就能如期完成任务的团队，必须把握好的原则，这是软件开发部门的基本观念，也是往后几章的基础。

专心改善产品

公司付薪水给程序设计师，是要他们在合理的时间内做出品质精良的软件，但是程序设计师的时间却经常被其他的事情占用掉了。这样的程序设计师或他们的经理们，就是因为不了解软件开发的真理：

任何不能改善产品的工作，都是浪费时间或是偏离方向。

如果您一时不了解这个观念的重要，请想一想以下两个极端典型的比较：一位程序设计师一天到晚开会、写报告、阅读和回复电子邮件，另一位程序设计师则专心研究、设计和测试新产品的功能，试问谁比较容易脱颖而出？毫无疑问是心无旁骛的那一位，他甚至可能提前完成工作呢。

我经常发现，团队出问题的原因之一，往往是因为程序设计师们都在做他们不该做的事：他们花太多的时间准备开会、参加会议、读写开会记录和进度报告，以及回复电子邮件。这些不能改善产品的工作，固然一部分是程序设计师自己主动做的，但更大的一部分是主管下的命令。

曾经与我共事过的一位经理,要求每一位组员要用 e-mail 交一份工作周报，每周开一小时的会讨论目前各人手上的工作内容，以及其他突发性的事务，开完会后，提出意见的人负责写出书面报告，交给经理。

这位经理的动机只是想管理每一个细节，但并不了解这会让团队被无意义的工作压得无法喘息。这些进度报告真的那么重要吗？那些后续报告的用意又是什么呢？如果开会时经理自己做个简单的笔记，不就省了这些报告所占用的时间和精神？

很显然，这个问题的答案必须视您身处的企业环境而异，但从我刚才所举的实例来说，其实只有最初制定进度的那份报告是有价值的，其他的报告都是可有可无，甚至进度检讨会都没有必要召开，而且每次那位经理要求后续报告时，我都很纳闷，心想：“我刚才不是已经告诉他我的想法了，为什么我还得再写一遍？”

我不过是偶尔去参加他们的定期进度会议，所以对我的时间损失并不大，不过，我常在想，不知道公司里有多少不必要的例行工作正在加重员工

的负担？这位经理本意是尽力把每件事情做到巨细靡遗，但是却违背了身为软件开发领导者的基本守则：

领导者的任务是努力消除程序设计师工作上的一切障碍，让程序设计师全力专注在真正重要的工作——改善产品。

这并不是震惊世界的大发现，而是极简单的道理，但是，有多少软件开发主管是真的把“ 消除程序设计师工作上的障碍 ” 当作积极追求的优先目标，而且确实做到呢？

如果刚才提到的那位经理真的用心去减少组员不必要的工作，我确信他可以想出更简单有效的方法掌握工作进行的状况，而不必一再浪费组员的时间开会和写报告。

千万不要把程序设计师的时间浪费在改善产品以外的工作上。

请不要从字面上解释我的话

我所谓的不要把程序设计师的时间花在改善产品以外的工作上，请不要从字面解释成程序设计师只许写程序。事实上，思考如何设计、测试程序和接受需要的训练等等，虽然不是直接投入在改善产品上，但对产品品质却有重大深远的影响。如果程序设计师在动手写程序前，仔细思考过产品的设计，把缺点改正，当然会比一味地埋头苦干要好得多了。

有些团队的活动，用意是让团队成员在愉快的环境中工作，提高程序设计师的生产力和士气，虽然看似与产品无关，最后还是对产品品质与工作效率有正面的帮助。

排除干扰

如果您希望团队能在期限之内完成好的软件，就必须尽可能排除一切不必要的工作，特别是您打算分派工作给全体组员之前，请等一下，问问自己，这件工作真的有必要叫大家做吗？能不能由您自己做呢？比方说，如果您要准备向上级报告项目概况，非得要所有的程序员停下手边的工作，为每个程序写一份摘要吗？我倒不这么认为。身为经理，您应该平时就对项目的进度及一切的状况非常清楚，不必靠人帮忙就能做出切中要点的演示文稿，而且信息已经存在脑海中，比起再去汇总整理一堆人的报告应该是快得多，也更好组织。或许这要花掉您几个小时，但总比打搅整个团队去做一件与产品无关的工作要好。

我通常会做得更彻底一点。如果我发现一位程序设计师总是被不能不做、却与产品无关的工作绊住，我会主动解除这件工作，由我来做好了，这样程序设计师就能完全专心在软件上面。除非是为了训练，否则没有任何理由要求程序设计师本人回复 e-mail，询问项目的 e-mail 交给适当的经理来回答就行了。凡是能由主管出席的会议或能由主管执笔的报告，都不应该丢给程序设计师，最好能把这些开会、报告都废除。

我知道这些建议与大部分的管理课程或教科书上所指的授权（delegation）有所冲突，我并不是说这些课程和书籍错了，而是您在实际工作中应该放聪明点，对于工作的分派应该更有选择性。如果把工作分出去的目的仅仅是为了减轻个人负担，实质上您做的是伤害团队生产力的事情。别人“能”做某件事，并不代表他“必须”做那件事。

您看过整个房子的搬迁吗？我不是指搬家，而是整个房子从地基上拔起，用车拖着移到别的地方（美国的房子多是木造的，可以这样搬移）。我们可以把项目比喻成那间房子，把经理想成开路先锋，他的责任是把前面所有的障碍物排除，让房子能一路不停地稳定前进，而不是走走停停，停停走走。

房子前进的时候，领导者不希望每到一个路口就停下来处理红绿灯，或协调其他的车子让路。他必须事先就规划好最理想的路线，在房子抵达路口之前，就协调好工程人员卸下悬挂着的标志灯或是电线等会阻碍行进的东西，等房子经过了再装回去。他应该避免将拖车停下来缴过路费，或是等待他跟工程人员开完协调会。

搬迁房子的开路人员明白一件事，如果要房子一路不停地向前行进，一定要找出所有的障碍，并且排除掉。过路费当然也可以由拖车司机来交，可是这样一来就得停下车子，如果由开路人员在前面交不是更好吗？很不幸，有太多的软件开发经理在不该授权时授权，让组员为了与产品无关的事情疲于奔命，被外界的杂事干扰正常的进度，以致项目进度不断延误，甚至停滞不前了。

保护程序设计师不受任何阻碍和干扰

如果我带的不是程序设计师，而是主管…

前面的讨论中，我假定您是程序经理，带的是程序设计师，但是如果您带的是测试人员、文件撰写人员或是其他类型的小组，您的原则还是一样的。基本上，身为主管的重责大任就是让属下专心

做他该做的事，不论他该做的是写程序、测软件还是写文件。

就算您带的全是主管，您也应该决定哪些工作是不必要的，尽可能免除。开个会了解各位主管们手上项目的进度，和开个会了解各位程序设计师们手上程序的进度，都一样是浪费时间，特别是独立进行项目的主管，没有必要让一位主管了解与他项目无关的概况。主管的时间对产品影响也很大，浪费主管的时间，他就没有办法做他该做的事——排除程序设计师的工作障碍。

一定有更好的方法可以减少干扰

身为经理，我在项目的每个阶段都会问自己一个问题：

“我努力的目的究竟是什么？”

我总是不断用这个问题提醒自己，因为太容易被不重要的工作拉偏方向了。如果您经常把报告或备忘录东修西修的，换个字型或样式，结果美化文件的时间比写的时间还长，您就明白我的意思。因为在那一刻，这件工作占住了您全部的思绪，似乎是最重要的事情，但只要您随时以整个项目的眼光来看事情，您就不会陷入细枝末节了。

我们谈过了把时间浪费在进度报告和检讨会议的案例，现在您应该可以回答这个问题：

“我举行进度会议和要求进度报告的目的究竟是什么？”

我想，主要的目的是了解项目进行的状况，以便及早发现任何进度上的延误，避免项目发生进度失控，不是吗？假定没有发生延误，每一个项目都如期完成，也没有人需要加班，那还有必要报告进度吗？当然不必了。

如果您开进度会议和要求进度报告目的，是想确定项目进度有没有落后，有必要动员全体去收集这些信息吗？我不这么认为。我从来不开进度会议，不论我是带新的项目或是接手其他人带了一半的项目，我第一个废除的就是这类没有意义的流程，我就是不相信非得用开会写报告的方式才能掌控项目的进行状况。

至于进度报告呢，它究竟有多重要？我认为进度报告虽然令人厌恶，但却是必要的，经理总得了解项目是否出了问题。但请别忘了，进度报告虽然必要，但是它绝对不会对产品有任何帮助。所以，当您遇到一件必要但对产品没有帮助的工作，您得寻求这个问题的答案：

“我该怎么样保有这项工作的好处，消除它的缺点？”

进度报告确实有重要的目的，但得花时间去写，而且令组员心生反感——至少在微软曾经有不少团队深受其害。

如果一位程式设计师要花几个小时写报告，交代自己做了什么工作、解释这件工作为什么花的时间比预计的长，那的确会对他造成过度的压力，并且让每个人预定项目一定会延误。更常见的现象是，程序设计师明明每天卖力工作至少 12 小时，而且没有磨洋工，整整工作了 80 小时之后即发现他只完成了本周日程表上 27 小时的工作。

进度会议（StatusMeeting）的用意

我想，每家公司的进度会议都不完全一样，我所指的进度会议，是指每个人坐在一起轮流讲讲自己做了什么，或是没做什么，只有这个用意。

还有一种进度会议，其用意不是讲述各人做了什么或是没做什么，而是同一项目中不同小组的

协调，只有多小组项目（multi-teamtoject）才有这个需要。每一个小组的组长并不报告各事项进行的细节，而只提出会影响其他小组的事项，例如上游工作进度比预期的慢，或是目前已经完成可以交付下游工作小组，或有一件事希望别的小组配合或支持等等。这种进度会议是为了协调小组之间互相依赖的工作事项。任何需要等待别的小组做完才能开始工作的下游小组，都需要预先知道上一小组的工作情况如何，而且每一位应该尽早知道，这对于制定下游小组的工作进度是非常重要的，不然会临时手忙脚乱，或是影响整个项目的进行。

但是真理依然不变，程序设计师是不该被这些事情干扰的，小组的组长去开会就够了。

如果您没有过这种痛苦的经验，请试着想像一下：您已经是拼尽全力加班了，但永远赶不上进度，目标离您仿佛愈来愈远，那会多么令人沮丧！更别提您不敢浪费一分一秒，每天都累得像狗。假如这种情况是日复一日、年复一年，您还能每天早上精神抖擞地跳下床，满怀热忱去上班吗？或者更可能的是，您心中充满愤怒、挫折、沮丧，愈是努力工作，工作积压愈多，结果进度愈落愈远，天——啊……！

我恨进度报告，因为它迫使程序设计师想着自己没做的事，而不是强调自己完成的事。组员无法感受到逐渐推进目标的快感，反而随时被提醒：进度落后了喔，项目不晓得会不会莫名其妙地搞砸。他们只知道自己工作确实很尽心尽力，但实在不知道如何跟上进度。

团队也和个人一样。如果团队觉得整体不断往前进，那么就会一直保持前进，而如果团队觉得自己一定会落后进度的，那么，惯性和自我预示的心理就会使它相信自己是永远的失败者，导致士气非常低落。

请不要误解我的意思：如果一位程序设计师明明工作了 80 小时，却只能完成进程表上 27 小时的工作，那一定有什么地方搞错了；也许他最近做了太多人员面试的工作，或是开了太多的会，或是最近 e-mail 太多，或是他正在调整自己的情绪，主管应该和他一起找出问题。但是，即使是他个人不懂得善用自己的时间，也不该用进度报告使他难堪。我们待会儿有更好的方法解决这个问题。让我们回到前面提出的问题：如何保持进度报告的好处，又不必忍受它的缺点？其中一个答案是设计一种新的进度报告，非常容易写，既不花时间，且内容是正面性的。

以下是我的方法（我相信您一定可以想出更多更好的方法）：

◆每当有人完成了一项新的功能或特色，就发个 e-mail 给大家。

◆每当进度快要落后了，就到我的办公室私下讨论原因，我们一起开动脑筋寻求解决之道。

就这么简单。依我的方法，典型的进度报告可能是这样的：

“我已完成 Faxmangler 的搜寻与取代功能。Frank”

主管应该看一下结果，然后回一个：

“我检查过 Faxmangler 的搜寻与取代，不太顺畅，请再修改。Hubie”

或是：

“很好，继续加油！Hubie”

想想看，如果大家经常收送这类正面的 e-mail，一定会觉得充满干劲，这和可恨的进度报告多么不同！程序设计师会很乐意看见这类的好消息，当自己送出完成工作的信息时，也会很有成就感；没有人会觉得这是很讨厌的报告。当某位程序设计师觉得自己可能要落后了，我会和他谈，研究将来如何避免这种事情。是我们在制定进程时疏漏了某一个重要环节吗？或是时间

表定得太乐观了？是不是有个错虫（bug）在作祟，害得程序很难写或无法测试？不论问题是什么，我们一定想办法解决掉，并且预防它将来再发生。

我只要靠这两种方式，就可以完全掌握项目进行的概况。上级要求的话，我也可以轻松写出项目演示文稿，根本不必劳驾别人，所以我带的程序设计师不会因此而被打扰。

更棒的是这两种反馈方式产生了双重好处：第一，团队成员经常感到项目又向前迈进了一步；第二，这对主管和属下都是很好的学习机会，我们不会耸耸肩，无所谓地说：“ 反正进程一定会落后的，没啥大不了。 ”因为我们会认真面对问题，商讨对策。

为了强调进度报告的正式性，而把它弄成一件沉重不堪的工作，这就是主管过度重视进度而忘了自己真正的目标。结果陷入了流程的陷阱，倒把产品丢在一边了。

只有当您非常清楚您和您的团队该做的是什么，您才能用最少的时间和最少的挫折去完成项目目标。回头检讨一下遭遇过的困难，有没有办法可以改善？有没有更好的方法，让工作愉快些？至于那些对产品没有帮助的杂务，如果可能的话，还是免了吧——至少不要把它丢给程序设计师。

永远记得自己真正的目标，然后让团队用最有效又最愉快的方法把它完成。

被成功的喜讯围攻了？

您可能会有这样的疑问，如果每个人都把自己完成某件事的讯息发给大家，会不会塞满了每个人的电子信箱。实际的情形是，每天的 e-mail 并不多，因为信息不会送给每个人，而是只送给同一组的四到五位程序设计师而已。

微软比较大的团队可能有多达 50 位的程序设计师，但是都会分成小组，每一个小组大约有五到六人。每一个小组负责项目的一部分，有一个明确定义的目标，有一位组长，也有自己的进程。小组内的程序设计师当然也属于整个团队，但是以每天的程序开发工作而言，只与四五位程序设计师有关而已。

事实上，即使是 50 位程序设计师的大型团队，每个人收到的“ 完成××”的 e-mail 也不多，数量稳定，刚好够让每个人感觉到工作的进行，绝无爆炸之虞。

明白说出目标

您所认识的人中，有多少位一早起来突然发现奇迹，选了几门神奇的课程就拿到了计算机硕士学位？有多少人随便收拾一下东西就搬到了您的隔壁？听起来不太可能吧。没错，一般人不会兴之所至就去修个学位或搬家，这些都是计划过的，他们会决定：“ 我要成为程序设计师 ”或是“ 我要住在迪士尼乐园旁 ”，然后筹划一番，再采取行动，才会达到目的。

可是，生活中的确有些事情是不经意就发生了，您可能靠运气就得到一份好工作，或是福星高照在股市捞了一笔。很不幸，推出一套软件的目标，并不比“ 我们要完成 WordSmasher ”具体。

不错，大部分的时候您都可以完成目标，但问题是，在这之前，有多少时间被浪费掉了？虽然您运气好，得到了一份理想的工作，但之前也许有好几年频频跳槽；或者您先细细思量自己适合什么样的工作，然后找到符合自己条件的几家公司寄履历、面谈，会比较稳当？

我所辅导过的案例中，有六个总是在失败边缘挣扎的团队，他们都有一个共同的特征，就是目标太模糊。其中一个案例的情况是，小组的任务是做出使用者界面的函数库，给 20 个左右的其他小组使用，他们做得人仰马翻，还被别的小组抱怨函数库既笨重，错虫又多，很不好用。

当我和这位组长共同检讨工作清单时，我问他项目的目标是什么。“ 为 MS-DOS 的应用程序提供像 Windows 环境的使用者界面函数库。 ”我问他还有什么。“ 您的意思是？ ”

我说，他刚才的答案太模糊，能不能说得更具体些？“ 嗯，函数库应该做到零缺点。 ”

我点点头，再问：“ 还有呢？ ”

他想了一下，耸耸肩，答道：“ 我想不出来了。 ”接着，我说明任何函数库的主要目的是存放每位需求者必要的公用程序，不是大家非得用到的东西是不该在里面的。他很同意，认为这一点是显而易见的。但当我继续下面的讨论，我开始不确定他是否真的懂得这一点。我指着工作清单上的一项，问道：“ 这是做什么用的？ ”“ 那是 Works 小组要求的，是用来…… ”

“ 对其他小组有用处吗？ ”

“ 没有，只有 Works 小组会用到。 ”

我指着清单上的另一项，问道：“ 这个呢？ ”“ 那是给 CodeView 小组的。 ”

“ 那，这边这一项呢？ ”

“ 是 Word 小组要的。 ”

我们一路看下去，我发现他对任何要求都来者不拒。我想他也知道函数库中应该只有大家都要用的程序，但他在做决策的同时并没有确实把握这个原则。他的目标只是“ 为 MS-DOS 的应用程序提供像 Windows 环境的使用者界面函数库 ”，有没有办法说得更详细些？

目标是为 MS-DOS 的应用程序提供像 Windows 环境的使用者界面函数库，只包含对每个小组都有用的程序。

如果把目标定义精确些，就可以发现很多程序其实是不应该放在函数库中的。在我们检讨完工作清单后，我开始研究另一个问题：“ 很多小组抱怨，函数库中每次一放入新版的程序，就会发生链接失败，这是怎么回事？ ”“ 喔，那很简单，他们只是忘了修改自己程序中的函数名称而已。 ”

这我就不太懂了，于是我请他给我一个实例。他的小组修改过函数库中的 bug 了，或是加强了该函数的功能，然后就连名称也顺手改了，用另一个更能表达其功能的函数名称；或者是做了一点小小的修改，然后把函数名称一道改了，以强调所改的差异部分。

这位组长不了解其他小组怎么会如此大惊小怪，不过是改个名字而已，非常简单啊。他从来没想过，有 20 个小组要因此检查所有的程序，把所有用到函数库的地方都改过来；他也没有想到，经常链接失败的函数库是很不好用的，函数朝令夕改，别的小组会无所适从，根本不知道程序应该是什么模样才对。

如果这位组长稍微花点时间，从别的组的角度来看看这个函数库，他就会明白放入新版程序时，与旧版能兼容是多么重要。大家都希望既用到新的程序代码，希望所有的程序都能链接成功，没有任何失败。

于是，我们把项目的目标再更具体化一些，那是：

目标是为 MS-DOS 的应用程序提供像 Windows 环境的使用者界面函

数库，只包含对每个小组都有用的程序，而且必须和其他部分的程序版本一致。

现在我们理清了影响这个函数库的重要相关因素，我和组长一起定出了清楚而明确的项目目标。我们发现，只要用心考虑各因素之间的关系，考虑一项行动会造成什么影响，很多细节就会变得显而易见，而且是可以事先建立正确的规范，避免事后再来花更多时间收拾残局。只要在做出决定之前想一想，“我要这个函数库能做到什么？项目的目标是什么？”很多问题都能迎刃而解，组长可以轻易决定什么是该做的，以及该如何做。

如果做得更彻底一些，组长可以花几个小时，甚至几天来订出详细的项目目标，这些目标不一定要博大精深，只要目标能够写下来贴在容易看见的地方，随时提醒、随时指引，让项目朝着正确的方向进行就可以。

如果函数库中都是全部小组需要用到的，函数库就会体积小些、精简而标准化，新增的功能特色就比较容易做出来，开发小组就不必拼命加班做一大堆只有少数人用得到的函数。想想看，只要把目标稍微理得清楚些，整个项目的方向就会有惊人的改变。

理清详细的项目目标，可以避免在不必要的工作上浪费时间。

相互依赖

项目失控最有可能的原因是小组之间工作上的相互依赖：一方必须依赖另一方把工作做完才能进行自己的工作，而且若是彼此的配合或沟通不够好，整体工作必然受到不利的影响，依赖愈多，愈容易失控。使用共享函数库有很多好处，这是人尽皆知的事。但是身为组长的人，必须衡量是否该把这部分的程序代码放进函数库，所获得的好处与必须忍受的缺点——让其他的工作依赖于它，孰轻孰重。任何领导者，必须牢记并且实践以下的座右铭：

尽可能减少项目中小组彼此间的依赖。

想想看，万一函数库的开发工作进度落后了，又未能及时采取改善措施，对其他的小组有多大的伤害！负责函数库开发的小组会拖累大家的进程，使大家不由自主地一起延误，若不能及时挽救，项目就真的失控了。

另一方面，如果依赖函数库的小组听说开发函数库的小组无法在规定时间内完成要求的工作，也不该强迫他们。因为函数库的开发是很多其他工作的基础，绝不容许发生丝毫错误。如果强迫开发函数库的小组必须在某个期限内完成过多的工作，等于是不恰当地依赖开发函数库的小组。

以上所谈的道理似乎简单浅显，但是做起来可不然，我看过很多重复发生的错误，有许多小组真的耗费了大量时间在函数库的依赖性问题上纠缠。

努力要有代价

有些管理学的书喜欢把设定目标看成一种神秘的哲学，让您读起来会有这样的感想：“我不知道究竟为何要设定目标，只知道经过研究证实，有明确目标的团队，会比目标模糊的团队更有生产力。”

我不知道这些管理学书籍为什么把设定目标这件事弄得这么玄，设定目标只不过是把“你要完成的事”用清晰的语言描述出来，让每个人都有明确的概念。如果您的目标是买一栋这样的房子：20 世纪初期的维多利亚式三彩建筑，有四间卧室两间浴室，后院还要一尊法兰西斯雕像，您大概会先看过一大堆的房子，然后才确定自己想要的是哪一栋。目标越是明确，达成目标的过程就会越有效率，因为您能马上拒绝不符合脑中未来景像的事物；明确

的目标之所以带来效率，正是由于它能帮助您挡掉别的项目丢过来的不相干的垃圾，让您专注在本项目的策略性事项上。

不幸的是，在软件开发过程中，没有任何事件能迫使领导者停下来想一想，却有庞大的压力迫使领导者没有时间思考，反而忽略了设定目标这么重要的事。当项目已经落后了一大截，都快失控了，谁还有时间去设定详细目标呢？有些主管不设定目标则是基于完全不同的理由：别人都没有设定目标，我们为什么要这样做？不论理由是什么，凡是没有明确目标的小组，必定遭遇额外的挫折。

如果您希望项目进行顺利，就一定要花点时间设定详细的目标，这不是什么有趣的事，但这一两天的功夫让您的项目不会偏离方向，相对的报酬是非常值得的。组员的努力应该有代价，而长期加班、饱受压力的小组，多半是工作漫无目标的后果。

不要因为制定目标需要花很多时间，或是别人都没有做，就省略了目标的制定。制定明确详尽的目标所花的时间，绝对会让团队得到更大的好处。

程序设计的优先考虑

如果您请三位朋友顺路到超级市场帮您买些芦笋、绿豆、玉米，结果其中一位买了罐头蔬菜因为最便宜，第二位买了冷冻蔬菜因为最容易煮，第三位买了新鲜蔬菜因为最健康也最美味，您会觉得惊讶吗？至少，您觉得这样的事情会发生吗？

这三位朋友买了不同的蔬菜，因为在他们心目中，强调的优先考虑不同，程序设计也是一样的道理：即使同一个程序，由三位程序设计师写出来的程序代码必定不同，第一位程序设计师强调程序代码应该越简炼越好，第二位认为容易使用最重要，第三位则喜欢追求执行速度。

假若您的产品必须快得让人目不暇给，而程序设计师却认为写程序的第一要点是轻薄短小，那么做出来的成品不太可能会使用特别的加速方法，执行速度也不太可能令人惊喜。假若您的主要目的是在最短的时间内做出高稳定性的软件，但是您的程序设计师却喜欢追求执行速度，而不太在乎稳定性，结果一定无法令人满意。如果程序设计师心目中理想的考虑顺序和项目目标并不一致，想得到满意的产品就好比缘木求鱼。

项目目标和程序设计的优先考虑并不相同，但二者常会发生部分重叠，主要是因为项目目标能帮助考虑顺序的确立，以下是重要的基本观念：

◆项目目标引导项目进行的方向。

◆程序的考虑顺序影响程序设计的过程。

假定项目的目标是做出世界上最快的 Mandelbrotplotter（一种画几何图形的程序），效率就是程序设计时的第一优先考虑。

暂时不谈程序设计的优先考虑有何重要性，在我的经验中，主管很少注意到这个问题。程序设计究竟该以什么为最优先考虑？速度？体积？稳定与坚固性？可移植性？可维护性？每一位程序设计师对于程序设计的优先考虑看法不同，并且反映在他们的作品中。如果大家各自随意，就很容易出现不一致的情况：有些人为了容易维护而坚持写干净漂亮的程序，另一些人认为速度是最重要的，所以写了一大堆别人看不懂的程序代码或汇编语言。

如果您希望小组工作有效率，能够精确地达成项目目标，您就得建立最

适当的程序设计优先考虑顺序，并且让所有的程序设计师确实遵守。最低限度，您应该为以下的程序设计考虑点排出一个优先级表：

◆体积大小（size）
◆速度（speed）
◆坚固性（robustness）
◆安全性（safety）
◆可测试性（testability）
◆容易维护（maintainability）
◆简洁（Simplicity）
◆再用性（reusability）
◆可移植性（portability）

这张顺序表中唯一需要解释的是安全性，如果您认为安全比速度重要，您就得选择重安全而轻速度的方式，优先要求程序中没有 bug。比方说，程序有两种方法可以选择，一是表格扫描（table-scanning）、一是逻辑判断（logic-driven），前者比较慢，但也比较安全，既然安全比速度优先考虑，那么除非有别的重要理由，您应该选择表格扫描的方式。

除了程序的优先考虑顺序之外，您还应该建立各项考虑点的质量规范指导。例如您认为坚固性是第一优先考虑，那到底要多坚固才算合格呢？最低限度，程序在输入资料正确时绝对不该发生错误，但万一输入资料不正确时，程序会怎么办？程序应该很聪明地处理错误的输入资料，在某程度内自动更正它（哪就得付出体积和速度的代价）？或是在程序中加入检查输入的动作，尽可能督促使用者输入正确的资料？或是就让错误的资料进入处理，干脆一路错到底？这个问题没有标准答案，全看您对品质的要求。

大致上，如果是操作系统，在不死机的前提之下应该尽可能接受输入值，如果是应用软件，应该尽量能更正不合法的输入资料，对无法更正的输入应该提出警告。但是，如果纯粹就程序的质量而言，也许该考虑强调性的失败信息，例如一个函数库中的程序，遇到部分不正确的输入值时，虽然可以做下去，但无法预料会有什么结果，此时不如当成错误的状况处理而中断执行，以免错用这个函数的程序误以为一切安好。总之，在一般情况下都应该对不正确的输入作某种程度的处理，只要别让程序代码过度庞杂就好。

这里的重点是，如果事前就决定了最合适的优先考虑顺序，以及各考虑点的质量规范，团队就不会浪费太多时间把程序改来改去，程序的整体风格比较容易一致。

事前决定最合适的优先考虑顺序，以及各考虑点的质量规范，能够指引开发团队的工作。

安全性或是可移植性？

以我个人的观点，我通常把安全性放在比可移植性优先的地位，也就是说，我喜欢安全的程序甚于可移植的程序，这听起来有点含糊，那是因为可移植性高的程序通常表现出的安全性也高，事实上，这两种特性没有真正的关连，只是可移植性高的程序碰巧也是安全性高的程序罢了。

写 C 程序时，宏（macro）是很好用的，它可以用来当作子程序的定义，但事实上它跟函数（function）不同，每一次启用宏，会被展开成一段一般

的程序代码。但如果写得不够小心（例如同一名称重复定义），它会造成潜在的 bug，这一点 C 语言程序设计师都非常清楚。用宏的方式来做函数，很好用但容易有危险。

如果您愿意利用某些 C 语言编译器特别提供的 inline 指令,就可以充分享用宏的好处，而不必冒潜在的风险，惟一的问题是，inline 不具有可移植性，在不同的编译器之间未必兼容。对我而言，安全比可移植性重要，所以我宁愿选择用 inline 的方式来保护宏函数。

当机立断

您可能听过很多极为成功的人士都有当机立断（snapdicision）的倾向，事实上，一般人如果过快做决定，大都是颜面尽失的悲惨下场。差别是在于，那些成功人士早就已经设定好清楚的目标和优先考虑的顺序，遇到任何问题，或是要裁决某个建议时，只要把它放到脑海中的决策“尺”一量，立刻就会有答案。另外，成功人士不会轻易改变主意的，改变主意等于是背叛了自己的信念。

事实上，成功人士并不是未经思考就草率决策，只不过他们把目标和优先级订得非常清楚，所以他们不必想太久，也不必为无关紧要的事情浪费时间；结果是：他们不必花太多时间来做决定，而是花时间去做已经决定的事。

严守基本原则

回顾本章的讨论内容，我们可以总结出软件开发的基本观念：“确定您要达成什么样的目标及如何去做，让每位组员都明白目标，并专注地朝这个目标努力，设定程序的优先考虑顺序，以及相对的质量规范指导。”这些都是很基本、很简单的观念。

现在,回头看看您公司内的各个部门,有多少个部门有清楚的项目目标？有多少程序设计师接到关于程序优先考虑顺序的明确指示，以及质量规范指导？然后问问自己：“程序设计师真的全力投入在改善软件的工作中吗？”

再看看您公司内的项目主管们，他们是否习惯在会议中讨论一些芝麻小事，或是在讨论真正重要的事情？他们是否常常扔些与产品无关的工作给程序设计师，例如写报告等等，或是认真为程序设计师清除所有阻碍工作的障碍呢？

本章所讨论的全都是相当基本的概念，但在我的经验中，很少人对基本的东西认真思考。而我相信正因如此，您随手抓起一本《信息世界》（InfoWorld）或《Mac 周刊》（MacWEEK）之类的专业杂志，里面就充斥着“某个项目已再度落后六个月”之类的报导，或是某位程序设计师为了工作连家都不能回的故事。

重点提示

公司聘请程序设计师，是为了开发高品质的软件，但如果经常被杂事打扰、分心，就无法保持专注在真正该做的事情上。主管必须确定程序设计师能专心投入在具有策略价值的工作上，而不是打杂，凡是会阻碍软件开发的东西，主管应该毫不犹豫地把它排除。

然而，有很多杂事其实是无法避免的，大公司尤其如此，那就只好将它的负面效应尽量减少，方法是不断自问：“我到底想要完成什么？”“我该怎么做才能既保持这件工作的好处，又能避免它的坏处？”要满足实质上的需求，而不是表面上的作业程序。

拥有明确目标所带来的好处虽然不是立竿见影，但没有明确目标所造成的混乱绝对是显而易见。没错，建立明确目标是一件费时又无趣的工作，但比起项目延误或失控的危险，肯定是值得付出

的。请记得使用者界面函数库的实例，项目目标只要稍微改好一些，就会明显地减轻压力，项目目标再修正一次，问题就几乎都迎刃而解了。

每一位成员都必须有一致的程序优先考虑顺序，程序的可维护性是最重要的吗？可移植性？体积？速度？为了让软件产品符合项目的目标，必须让程序设计师明白本项目的程序优先考虑顺序，他们在程序设计时才知道该如何取舍。同时，您还得对每一项优先考虑点事先建立质量规范指导，以避免到时候质量不合格又得重写部分程序，导致时间浪费和项目延误。愈早定出质量规范指针，愈能省时省力。

第 2 章　策略性的作业方式

虽然我从事软件设计已达 20 年，但我写技术文件或文章时，从不使用文字处理器，听起来很意外吧？我仍然用最传统的纸和笔做最初的草稿，然后再输入计算机去编辑。

我肯定自己没有计算机恐惧症，而且我很清楚用纸笔绝对比不上计算机方便，但我仍然这么做。

很久以前我就发现，当我使用文字处理器时，每写完一句我就忍不住用编辑功能东修西改的，最后一天下来写不了多少东西。由于在文字处理器上编修文字远比撰写文字简单，我常常沉溺其中不可自拔，太分心在编修上，忘了写作内容才是本旨。反正早晚得修饰嘛，现在编修并没有什么不对，只是我拖延了自己的工作效率。

有一天我终于领悟到，原来我一直在破坏自己写的作品，我必须寻找提高写作效率的方法，我试着在用文字处理器时专心写作，不去使用编修功能，却没有什么成效。我需要的是撰写比编修容易的东西，那只有纸和笔了，于是我不再使用文字处理器来写作，我只用它来编修那些已用纸笔写好的几页文稿。

我的新方法解决了我在写作时分心编修的恶习，因为它让我只管写作。

从这里我们可以看出，在程序上的一点小小改变，可以造成非常不同的结果。以前我写五段的时间，现在我可以写出五页，这样的进步是因为我成了一位有经验的作家吗？还是因为我的工作时间加长了吗？都不是。我的生产效率提高，是因为我了解到我使用的工具有个缺点，而我找出了另一种比较有效率的方式。

当您读完本章后，您会看到更多小改变大收获的实例。一旦您掌握了这个概念，把它应用在项目上，您可以大声说自己确实是在聪明地工作，而不是辛苦地工作（workingsmart，notworkinghard），并且您的工作能够事半功倍，再也不用熬夜加班，也能如期完成项目。

浓淡合适的咖啡

在咖啡店里常常遇到的难题是记住哪位客人喝什么咖啡：有些人要喝无咖啡因的，有些人要喝一般的咖啡。于是有些咖啡店经理不惜花大笔金钱送员工去接受超强记忆术（KevinTrudeau ’sMegaMemory）的训练，在那里他们训练学员如何把看似无关的事物作视觉联想：把客人点的咖啡和他的衣着、领带等联想在一起，比方说点蓝山咖啡的这位小姐正好有个特征是戴蓝框眼镜。而大部分的经理会用更简单的方法解决问题，比方说规定某种咖啡要用某种特征的杯子，这样服务生只要一看到杯子，就知道这位客人喝哪种咖啡，续杯时就不会弄错。

常见的问题，也许只用一个微不足道的方法就可以解决。

除了咖啡店的例子之外，我们来想像一下，还有没有别的地方也可以运用这个原理解决问题。现在我们来看另一个例子。

我家附近有两家咖啡屋。他们用一模一样的咖啡壶、咖啡豆，连服务生都一样用的是大学生，怪的是，煮出来的咖啡却大不相同：其中一家的咖啡一直是浓淡相宜的，另一家却时浓时淡，有时甚至烧焦，简直令人难以下咽，您根本不知道端上来的咖啡会是什么样的味道。

这两家咖啡屋的一切设施几乎完全一样，除了一个小小的细节：质量稳定的那家，他们的咖啡壶上有一条横的浮雕花纹，这就是简单而关键的质量体系（qualitysystem）的秘诀，靠着这条细细的横纹，他们可以提供品质稳定的好咖啡。每次新的服务员到职，经理就指着这条横纹给他上课，告诉他：

“你倒咖啡时，只要发现水位在这条线以下，就要立刻煮新的一壶，要立刻去煮，不要被任何事情耽搁。”

“万一那时候店里很忙怎么办？”

“即使一整队的芝加哥公牛刚刚打完球，全挤到这儿来也不管，还是先煮新咖啡要紧，那怕你已经倒好一杯咖啡要端给美国总统，只要看到水位在这条线以下，就要停下所有的事，先去煮新咖啡再说。”

接着，经理开始解释，从一个空壶、放进磨好的咖啡豆、到开始煮，总共要花 15 秒钟，这样虽然会让客人多等 15 秒钟，但却可以避免这壶咖啡全空之后、新咖啡未煮好前，让客人多等 7 分钟。

但是如果您走进另一间咖啡屋，坐下来，您可能会看到服务生伸手拿咖啡壶却发现里面是空的，于是您得等上 7 分钟。有时候，服务生为了让客人少等些时候，就会看看从咖啡机滴到壶里的已经煮好的咖啡，如果能凑成一杯的话，就把这杯倒给您。但是好的咖啡不是这样煮的，应该等到这一泡咖啡完全从咖啡机里滴到壶中，热水和咖啡的混合比例才会恰到好处，如果太早把壶内的咖啡倒出来喝，就会太浓而难以入口，而后段的咖啡又太淡没有味道。这就是这间咖啡屋质量不稳定的原因。所以有时候倒出来的咖啡，虽然看起来很像咖啡，味道却是咖啡渣加白开水，有时候喝到很正常的咖啡，有时候咖啡竟煮焦了——为了快点煮好咖啡，咖啡机内只放了一杯份量的咖啡和水，由于水太少，加热器不容易调到适当的火力，结果咖啡就焦了。

这两家咖啡屋唯一不同的是，一家等到壶内的咖啡全空了才再煮新的，一家却是壶内的咖啡到了低水位时就再煮新的一壶。两家的作业方式几乎完全一样，就只有这点小小的差异，结果竟然如此天差地远，而且这和人员的技术毫无关系。

我举咖啡屋的例子，当然是因为这个原理和软件开发很有关连。

如果我问您，软件开发过程中，正确的除错时机是什么，您会怎么说？等到所有的功能开发完毕后再一起测试、除错，或是一发现有错误就立刻除掉它，或是无所谓，反正花的时间都是一样的。

如果您认为何时除错都一样的话，那可就错了。这就像咖啡屋经理误以为什么时候煮新的咖啡都无所谓，是一样的错误观念。对于项目经理而言，最坏的情况莫过于被错误整得团团转，根本无法追求项目目标；如果您想要控制好项目的发展，最好是不要有任何的错虫，忽略了这个目标就等于是注定失败（我在《零错误程序》一书中有详细的说明）。当我刚加入 MicrosoftExcel 工作小组时，他们都喜欢把错虫留到后头再来清除，我指出这种作法可能带来的种种问题，最糟的一个结果是，到时候会无法决定产品究竟可不可以推出。因为实在太难估算一个错虫要多少时间去把它逮到、消灭，再说，除错的动作可能带来新的潜在错虫，这是测试小组无法确定的。

如果只管开发，把错虫留到最后再来解决，会让项目的完成比率被高估。看起来好像已经完成开发的项目，高层主管却惊异地发现还要用六个月的时间除错，而只有真正在拼命除错的程序设计师才晓得为什么，因为到处都是错虫，这个产品不能推出。

在好几项软件因为错虫太多而不得不宣告失败之后，微软决定好好自我检讨一番，以下是这次检讨的摘要：

◆错虫愈晚清除，时间花得愈多。毕竟，您得知道程序是怎么写的，才能判断那里出了错虫；刚写完的程序记忆犹新，一年前写的程序可能早就忘了。

◆在开发的过程就立即除虫，可以让您早些学到经验，然后就不会再犯同样的错误；相反地，如果到了项目后期才发现，您可能已经犯过多次同样的错误而不自知。

◆发现错虫而立即除错是一种缓冲器，提醒那些讲求快速而不够谨慎的程序设计师，以后多加小心。如果您坚持错虫全都清除了才能开发新的功能，就可以防止所有的程序处于半完成状态，因为错虫太多而使项目延误乃至无法推出；相反地，如果您允许错虫的存在，等于是埋下了项目失控的地雷，最后看似完成的项目，其实已经失控。

◆若能保持没有任何错虫，您就能比较准确估出项目的完成时间。不必猜测 32 项功能和 1742 个错虫共要花费多少时间,只要估算 32 项功能的工作时间就行了。更重要的时，万一到时候有些功能做不完，您可以做多少算多少，因为软件一直保持在无错误状态。我经常提醒每一位程序设计师，假若您发现了错虫，而不打算立刻除掉它，请想想微软的惨痛教训。从别人的经验中学习比较上算，要不然您想自己走一遍错误的道路吗？

一发现错虫就立即清除掉，别拖延。

无法忍受的慢

在微软，有些小组把“ 错虫 ”的传统意义扩大解释，包括产品的性能缺陷也算是一种错误；例如软件的执行速度太慢了，即使所有的功能在逻辑上都是正确的，仍然被认为是一种错虫。

如果他们有确实采取“ 一有错误、立即更正 ”的策略，他们就得事先定义何谓“ 无法忍受的慢 ”，也就是必须早点定出质量规范指导，这样的话，程序设计师一开始就知道标准是什么，免得到时候又得重写部分程序代码。

这样做的缺点是程序设计师可能过度追求速度而写出很复杂的程序，因为在写完之前他无法知道整体的执行效率会如何，有时候会品质过高，不过一般来说这种缺点是很容易被发现并改正的。

程序设计师应该把找错虫当成一件很重大的事，必须立即采取相应的行动，不为任何理由而耽误，就像咖啡壶里的水位一般。要求错虫随时发现随时改，等于是在开发过程中引进一个小小的质量管理机制，多方的防微杜渐，保护产品的正确性，这种策略还有其他的好处：

◆以事实向程序设计师耳提面命，错虫是一件严重的事情，不可轻视忽略，也无法逃避，还是勇敢面对它、尽早解决它最好。

◆自己的错虫，自己负责清除。不该是小心翼翼的人去帮助散漫随便的人收拾残局。老是出错虫的程序设计师，会因为必须清除错虫而十分辛苦，而没有错虫的程序设计师则进度稳定，这样才公平。一方面也是鼓励程序设计师，要非常小心谨慎，一个错虫的代价常常是很高昂的。

◆如果一发现错虫就立刻清除，就不会漏掉重要的错虫，也不会到头来赫然发现好长一串错误清单，以致到了期限前才发现项目的延误。也就是说，打击恶魔要趁早，趁它还没有长大到无法收拾以前就把它消灭掉。

◆最后，也是最重要的一点，如果您要求程序出现错虫时立刻清除，那么程序设计师的功力高下便可立见分晓，如果有人的进度一直落后，这等于是警告您——他需要加强训练了。

软件开发过程中，有非常多的小事情会影响整体项目进行的顺利与否，以及产品的品质，绝对不可因为事情“小”就不予重视。咖啡壶上的那一条小小横线就是例子，您可以看出来它的影响有多大。您可以想出更多让项目顺利进行的方法，抓对要点的小小改变可以带来大效益，好好运用这个原理，您会受用无穷。

妥善运用可以促进开发成效的策略性工作方式。

电子邮件的陷阱

e-mail 实在是个很棒的工具，我简直无法想像没有它的话，工作效率会变成什么样子。但是我不得不说，水能载舟亦能覆舟，如果 e-mail 被不当使用，它也会伤害生产力。

我常发现新进的程序设计师喜欢让 e-mail 打断他们的工作，我不是指他们发了太多的 e-mail，而是只要有新的 e-mail 进来，他们就停下手边的工作，看看有什么新鲜大事发生了。新进人员一般不会有太多 e-mail 必须回复，大部分的 e-mail 都是被动性的信息，像微软股票的收盘价、同业的重大讯息、当天的头条要闻等等。

新进人员常常每隔 5 分钟就看看 e-mail 信箱，这样他们一天下来可能一件事也做不成，因为写程序的工作是无法分割成 n 个 5 分钟的片段去完成的。为了解决这个问题，我只能不断告诫新进人员，回复 e-mail 要分批做，不需要实时就做：早上一进公司时看、中午休息时看或是下班前看一下，都可以。e-mail 的目的就是要让程序设计师思绪不被打断，所以不必有事没事就瞧瞧它。

每一位程序设计师看的信息数量都是相同的，这是固定的。您可以把 e-mail 处理得很有效率，也可以让工作进行得更有效率。

学习前人的经验

我经常用很多例子，向程序设计师或主管们说明策略性工作方式的重要，小小的要领可以带来重大的益处，有些人总是不相信，他们认为工作方式是拐杖：“你是在欺骗那些人去倚赖经验，一旦他们换个角色，他们就没有学习能力了。”

由于我深信策略性工作方式的好处，故而我也非常在乎这些人的反应。我相信心存怀疑的人在读完本书之后，也会迫不及待地想要运用更好的技巧来改进工作效能。我认为本书和书中的经验法则并不会阻碍一个人的学习潜力。

最美妙的是，如果有完整设计的工作方式，任何小组成员（不论他是新进人员或由他组调来的人）都能立刻用最有效的方式工作，不必费神去了解或摸索这种工作方式背后的意义。我的建议是，对于每一项工作方式都详细解释它的用意，您为什么如此规定，您期望组员做到什么。适当的时候，组员自然会明白并感激您的巧妙安排，而且更服膺这些道理，甚至自己也开始思考如何改进工作效能。因此，您应该鼓励您的团队成员用心了解与改进工作方式。

不要把策略性工作方式当作训练的教条，应该向组员解释这些工作方式的内涵与用意。

好方法要让大家分享

设计完善的工作方式是很有价值的，因为它很自然地促使人们做对产品最有贡献的事情。策略也是非常重要的，因为它是许多经验和思维浓缩而成的计划，让每个人都了解它，并且付诸行动。将这样的策略或方法集合起来，能够让个人的生产力和工作质量提升到更高的境界。

身为一位主管，您应该鼓励组员提出改进工作效能的建议。以我来说，我对于软件的第一优先考虑点是没有错虫，当然，我们都知道这是说来容易做来难的。即便如此，我还是认真观察那些较少出错虫的程序设计师，研究他们为什么比较能避免错误，我的结论是他们比较懂得在容易出错虫的地方特别小心，懂得避开程序设计时常见的盲点或陷阱，也比较懂得用有效的方法抓虫。换言之，他们懂得写出无错虫程序的方法。

为了鼓励程序设计师学到避免错虫的策略，每次他们在追查错误原因时，我都会问这两个问题：

如何避免这个错误？

我如何以更简单（或自动）的方法发现这个错误？

如果程序设计师经常思考这两个问题的答案，他渐渐就能学会更高明的程序设计技巧。有些程序设计师是同样的错误一再地出现，显然就是主管没有鼓励他认真思考这两个问题，从错误中汲取教训。

当然，您的询问是否切中要害，也引导着组员的思考方法是不是问题的要害。您可以经常问自己这类的问题：

为什么进度总是一再落后？

有什么办法可以避免将来再发生进度落后？

虽然这两个问题很类似，都在问进度，但我想您的答案不会相同吧。第一个问题着重在原因：互赖性的工作太多，工具太难用，老板是个白痴，老在阻碍工作的推动等等。第二个问题在问未来的预防方法：减少互赖性的工作，请购更好的工具，与老板加强沟通，争取他的配合等等。这两个问题的方向不同，第一个是探究原因，第二个是未来的改进方法，因此，这两个问题的回答也一定不同，第一个导引出抱怨，第二个才导引出解决方法（attackplan）。

即使您的问题非常好，完全正中核心，也未必能导出正确的策略。项目的目标会因为定义精确而获得更好的效果，问题也是一样，问得愈精确、问题愈有力，让我们来看看这个问题：

如何保持每次都如期完成软件？

有些主管问这个问题的后果是逼迫组员加班；有些则是以分红或主管掏腰包买晚餐，或是安排一场热门的夜场电影外加爆米花（别只管笑，真有这样的事儿）来贿赂组员加班。

但是，主管可以把问题问得更精确、更有建设性：

如何在不加班的前提下，而能如期交差？

这回的答案可不能一样了，因为逼迫加班和利诱加班都被排除在外，既然不能增加工作时数，于是主管不得不想法子加强效率，去找寻更有效的解决方案。为了不加班，也许得雇用更多的人手，这是解决方案之一，但公司绝对不会喜欢，至少这个方案会摆在最后再来考虑。这样吧，我干脆再把问题说得更精确些：

如何在不加班、也不增加人手的前提下，依然如期完成软件？

这可就真的迫使主管来点真正有创意的思考和认真检讨工作本身值得改进的地方了。也许主管不认为所有的程序一定要自行开发，他可以雇用一位短期的顾问，或是运用公司内别的小组已完成的程序代码，或是买一套文件完整的函数库，这都能够大幅减少开发程序的时间。也许，可以把产品中较无价值的功能特色从目标中删除，这也是个办法。

理想的问题

您读完本书之后，会发现不必每周工作 80 小时就能如期完成项目的方法还真是不少。当您用自问自答的方式来引导策略性思考时，请不要忘了第 1 章所提的重点：“我到底要完成什么？”没有一位主管喜欢自己的组员成天加班，事实上，大部分的主管都希望大家能在愈短的时间内完成愈多的任务。要提出最佳问题最简单的技巧是：想像一下您理想中的项目应该如何运作，然后在您的问题中反映您的理想。您理想中的项目，应该是对进程估计准确、每一个里程碑都准时到达、没有人需要加班、每个人都乐于工作，不是吗？有太多值得自我检讨的问题，同一个问题有太多问法，您必须记得，在问题中反映您所希望的理想项目，就比较容易得到接近理想的答案。

我们要强调的是，愈精确的问题，愈能促使人们朝向更接近理想的答案思考，剔除那些不够好的答案，因为第一个想到的往往是最简单、不够理想的解决方案，我们不能这样就算了。一次比一次更精确的问题，可以刺激思考过程，激发出更有创意的答案。

提出精确详尽的问题，可以引导出真正有效的策略性工作方式，帮助项目目标顺利完成。

不要死守规则

当您提出并推动策略时，同时也要提醒开发团队，不见得要 100%地遵循策略，最重要的是灵活运用。做事情要用脑筋思考，理性判断，而不是盲目地人云亦云，一味地死守规则。规则也有不太适用的时候。

有一项几乎是铁定的程序设计策略，就是不要使用 goto。但是，有经验的程序设计师一般都认为，在某些特殊情况下——大部分是复杂的错误处理，用 goto 反而使程序代码清楚明晰。如果我看到程序设计师在写错误处理程序时，极力避免使用 goto，我通常会拿着程序代码和他一起讨论，我会问道：

“你有没有想过用 goto 来写这段程序？”

“什么？当然没有！goto 是程序设计的祸源，会使程序既不稳定又难读懂，只有低能的程序设计师才用 goto 吧。”

我解释道：“嗯，说得对，不过在某些情况下 goto 是合理的，复杂的错误处理就是。让我们拿用 goto 做的程序和你的程序作个比较，你觉得哪一个的程序代码比较清楚？”我拿用 goto 做错误处理的程序代码给他看。

通常程序设计师都会不太情愿地承认是 goto 版比较好。

“那么，你将来打算用哪种方式来写程序？”

“我自己的，因为它不用 goto。”

“等等，我以为你刚才同意了 goto 是比较好的方式，程序会比较好读，不是吗？”

“它是比较好读，但是用 goto 的话，编译器会无法产生最优的执行码。”

“让我们假设你的论点是对的，编译器所产生的执行码的确差些。你想，

执行这一段程序机率大不大呢？”

“很少会执行到，我想，因为它是错误处理。”

“既然不常执行到，那么对于这段程序而言，执行速度和程序代码的可读性，那个比较重要呢？”

“程序代码的可读性。”

“所以，哪种方式是比较符合项目的优先考虑的顺序呢？”

这时候通常会有一段较长的沉默。然后程序设计师终于勉强吐出了一句痛苦的抗议：

“但是，用 goto 不好嘛。”

首先我得说明，该用 goto 的情况其实是很少的，大部分的时候只要一看到 goto 我就有点紧张，生怕这里有问题。我虽不会对所有的 goto 去之而后快，但我个人绝不赞成用 goto；大部分的 goto 都是散漫的程序设计师，一边哼哼唱唱、一边在键盘上随意敲打的低劣程序。然而，我虽反对使用 goto，我更反对的是对规则盲从，产品才是最重要的。

这就是策略性工作方式的主要缺点，规则太明确了，有时候反而让组员把它视为不容打破的定律，而不去活用策略。僵化地死守策略无异于做傻事。

我知道学校里的老师在教程序语言时强调不要用 goto，基本上是出于善意；他们是对的，但我希望程序设计师必须明白，尽量少用 goto 并不表示绝对不要用。我更希望学校里会示范少数应该使用 goto 的时机，证明在那些情况下使用 goto 是合理的，因为不该用 goto 的实例已经看得太多了。我想这是因为很多老师自己主张绝对不该使用 goto，在教学时自然特别强调 goto 不可用，只要一看到学生的作业中有 goto 就认为这是可怕的程序，使得很多毕业出来的程序设计师畏 goto 如蛇蝎,就好像电影中只要有裸露镜头就判定这是不道德的一样。

程序设计中，只有很少数的策略应该被视为规则，规则该被遵循，但不是死守，主管必须教育组员明白这一点，并教导组员应该如何灵活运用策略，否则属下若是只晓得盲从，就是主管的危机。策略不是死的定律，要灵活运用，不要死守，这是采用任何策略时都应该注意的，当然包括本书中所提的所有策略。

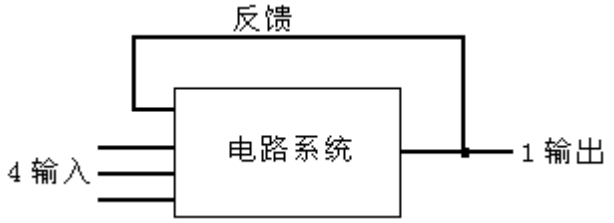
策略不是死的定律，要把它当作指导原则来活用。大部分的时候都应该遵循，但也有例外的时候。

给我看原始程序代码

关于是否该用 goto，教科书、参考书、杂志等已经都讲烂了。所有关于赞成和反对使用 goto 的文章中，麦康奈尔（McConnell）所著的《代码完成》（CodeComplete）一书中第 16 章可以说是最完整的了。他除了举出许多用 goto 确实可以增加执行效率的实例之外,更进一步告诉读者他赞同 goto 的理由，也证明了部分反对 goto 的理由是太牵强了些；麦康奈尔也整理出很多参考资料，包括爱兹格·迪杰克斯拉（EdsgerDijkstra）引发这场世纪大论战的信件，以及大师唐纳·克努斯（DonaldKnuth）那篇举证繁多的“用 goto 的结构化程序设计”（StructuredProgrammingwithgotoStatements）。正如麦康奈尔所言，虽然是否使用 goto 的千古难题至今依然在程序设计师的生活中一再上演，但教科书上的争论，也都还是 20 年前的东西吵来吵去罢了。

反馈回路

电子工程师会利用一种“反馈回路”（feedbackloop）的电路系统，将输出的讯号再当成输入讯号，反馈给系统本身。图标如下：



由于输出不断反馈给输入，这样的电路系统会产生两种结果：反馈讯号与输入讯号相加，称为正反馈，输出讯号愈强，得到的反馈愈强，因而导致输出再放大、再放大；第二种结果正好相反，称作负反馈，反馈讯号会与输入讯号相减，不断相减的结果，最后会达成一个较稳定的输出值。从这样相当简单的描述看来，似乎正反馈很了不起，因为它会自我加强能量，而负反馈则不好，因为无论它输出的起始值多大，最后都会缩小。事实呢？正好相反，负反馈远比正回馈有用。

您在听演讲时，可曾听过麦克风尖锐的叫声？足以吓醒所有的瞌睡虫，这就是正反馈的现象（译注又称作“反受放大”，在唱 KTV 时麦克风绝对不可指向喇叭，就是这个道理）。麦克风除了接收演讲者的声音外，还接收到喇叭的放音，不断地反馈循环，最后使喇叭负荷超载，发出尖锐刺耳的声音，而且频率还愈来愈高。负荷超载（overload）就是正反馈常有的问题。

相反地，负反馈则是以输出来抑制回路未来的输出。其实我们开车就是一种负反馈的系统，先踩点油门，慢慢加油，发现太快了就带点刹车来减速，如果刚起步就是油门一踩到底，那就得重踩刹车才能达到正确的速度。也就是说，输出愈强，需要负反馈的抑制力就要愈大，但也不是让反馈的力量决定一切，负反馈只是调整用的，好让输出维持稳定。停车只是刹车踏板的作用之一，让车速保持稳定也是刹车（负反馈）的作用。

除了电子工程，您还可以发现有很多各式各样的反馈回路，包括人际关系和软件开发。有些反馈回路是刻意设计的，有些则是无意间自然形成的，不论是有意无意的回馈，都有助于加强对项目的控制，所以，您必须明白反馈回路的道理，并且善用它。

错虫，就是程序设计的输出产物之一，我们把软件开发当成电路系统，如果有个负回路可以让错虫导回去抵销下一个潜在的错虫，那有多好！我们前面谈过的立即除错就是一个负回路的观念：

要求程序设计师一发现错虫就立即清除。

如果一发现错虫就立即消灭它，它就没有机会影响到程序设计师的心情，大家可以高枕无忧地继续做下一项工作。但是如果有个伤脑筋的扩散型错虫，清了这里却错了那里，再去修改那里却发现还有某处情况不妙，就得坚持要求这位程序设计师除错到底，把所有的错虫都确实清除干净了，才可以继续开发工作，否则会造成野火燎原之势，错虫一发不可收拾。错虫愈多的程序设计师，愈要加强督促。“立即除错”就是一个最好的负反馈系统，让软件永远保持无错状态。当然，还有我们前面提过的种种立即除错的好处——刚生成的错虫比较容易清除、让程序设计师学习速度加倍、更能准确估算项目的完成日期等等。

反馈回路有利亦有弊，运用不当的话也是会有问题的。记得我在第 1 章中谈过的实例吧，那个项目经理总是要求他的组员写进度报告、开进度检讨会、再做后续报告，没完没了。这位主管是希望藉此获得项目进度的信息，很不幸，他的负回路却阻碍了他真正需要的产出，他想了解组员对于如何解决问题的见解，可惜方法不对，他要求组员写报告，结果让组员心生反感，根本不想发表意见，他的制度使人噤声——讲得愈多，你必须写的报告就愈长，没有人想写报告，所以只好闭上嘴。这正所谓适得其反。

负反馈不是惩罚

惩罚是一种心理上的负向强化作用（negative reinforcement），惩罚是对员工责骂、训斥与威胁，就像鞭打马匹使它服从主人的命令。发现有一位组员进度落后了，不得了！叫过来骂一顿，这就等于是给了他一帖重剂量药物，逼使他以后不敢对进程掉以轻心。

这种管理手段是该受谴责的，我绝对不鼓励任何人这么做。想一想我们前面刚讨论过的负回馈的观念，要求程序设计师立即除错，但程序设计师不需要对除错感到焦虑不安。由于立即除错的策略，他必须花费好几天的时间解决这个问题，当然不是他所喜欢的结果，但主管不应该让他因此而感受到威胁。我们希望任何事情都很自然，没有必要加重组员的苦恼，绝不是强调谁是老板谁是奴才，谁必须服从谁。

在微软曾经有几位主管，每次一遇到项目进行不顺利，就把组员叫出来骂，说他们是最差劲的程序设计师，不配自称是微软的程序设计师，以及等等之类的无聊话。我不能确定这几位主管究竟用意何在，但是如果他们的目的是让组员工作更努力的话，那他们的方法可就大错特错了。我相信您完全可以想像，这种责骂只会激起组员心中的愤恨、羞恼和沮丧。更糟的是，就我所知这些项目的问题事实上都出在管理方面，目标不够明确或是野心太大，这些项目的程序设计师只是倒霉遇上了差劲的主管，其实他们的能力并不比公司内其他的程序设计师差，因此责骂他们只会让项目更糟，绝对没有改善的效果。

您必须小心注意，别设置了不当的负回路系统，例如以程序新增的行数、或新加入程序的数目来决定组员的奖金，把程序改好则不记入功劳，那么，程序员肯定只爱写拼拼凑凑的初稿，程序不求精巧，而且不愿认真改善现有的程序。您原本希望奖励的是生产效率，结果却造成公司里大而无当的拼凑程序到处泛滥。

我希望您能从以上的讨论中，体会到两点原则，第一，当您设计一个新系统时，利用负回路的观念来帮助项目进行顺利；第二，要考虑这种回馈对员工的长期影响，确定不会造成不良的副作用。

在您的软件开发活动中，小心谨慎地运用负回路的观念，让项目顺利进行；但务必要注意避免反馈回路的不良副作用。

愈简单愈好

最后，请您确定您的策略性工作方式够简单明白，让 人容易了解和遵循。

请回想一下我刚才举过的实例：用纸笔写的稿子、咖啡的浓度、整批阅读 e-mail、立即除错，这些都是细微的小事，不会对作业流程发生重大的影响。

人都希望用简单的方式解决复杂又耗时的问题，因为人们常常被工作之外的程序性事情绊住。如果问一位程序设计师：“如何避免错虫？”这个简

单，但若是要求所有的程序设计师针对他所遇到过的每一个错虫写出一份报告，说明如何避免这只虫，那就是另外一回事了，简单的事情立刻变成大麻烦；如何避免错误的报告应该是主管去汇总撰写的，组员只要口述检讨就好。这种麻烦像荆棘一般，会自动蔓延生长，您必须时常大力扫荡一番。

请切记，我们的目的是整体生产力的提升，而不是去填满繁杂行政程序的各步骤。我们要获得工作方式的好处，丢弃它的缺点；妥善的策略性工作方式可以达到这个目的。

重点提示

小小的改变可能产生惊人的效果，所以，请仔细观察您现有的作业方式，会很容易发生问题吗？耗费很多时间吗？矫枉过正或防弊重于兴利吗？会不会让人员心生挫败，而造成生产力低下？如果是的话，请找出一种简单又有效的方式改善这些情况。

当您决定采用任何一种策略性作业方式，请解释您的用意，让组员充分了解是什么方面应该改善。这种开放的做法会在无形中教育组员，让组员学会思考，也许，时间久了之后，他也能想出很不错的点子。

当您针对问题寻求解决方案时，一遍又一遍地修正您问自己的问题，培养自已能够提出精确的问题，想出更好的答案。但光是精确还不够，精确的问题也可能是错的问题，让您得到没有帮助的答案。您必须注意，问题是否切中要害，是否是您真正想达到的目的，是否是您的理想状况。不要自问：“如何叫程序设计师加班？”要问：“如何增强工作效率？”策略愈是吸引人，愈会有多人认同它，甚至把它当成牢不可破的定律。请提醒您的组员，再好的策略也不能应付每一种情况，“避免用 goto”是公认的好的程序设计策略，它让程序可读性提高，但是当不用 goto 的结果是可读性反而更低时，您得教程序设计师如何权衡取舍。

每当您建立一个反馈回路时，请务必考虑它的副作用和长期使用的效果。最好的反馈回路不但可以随着时间增强效益，也能同时减少负面的作用。

第 3 章　保持进度
ChapterThree

我们都会希望项目按照事先规划好的进程来进行，但是事实总是无法尽如人愿，有时候项目会有一点超前，有时候会落后些，但不会与计划差太多。项目的进行大都是以迂回的方式前进。

即使是进行最顺利的项目，也无法完全按照计划执行。但是如果您放任项目随意进行，有一天您猛然发现项目脱轨太远，无法把方向扭过来，剩下的时间也不够，也就是说，项目完蛋了。项目就像是一枚瞄准月球的火箭，只要有一点点不够精确，到时候就无法命中目标，差之毫厘，失之千里，实在不可不慎。所以，绝对不要让项目有一点点脱轨，不论是多么小的偏差，倘若您没有立即修正错误，它很快就会愈跑愈远、无法抓回。

聪明的主管懂得这个道理，他们会经常注意项目的进度，随时修正方向，保持项目不偏离计划进行。本章将介绍一些很有效的策略，帮助项目保持进度。

高架道路

我一直相信，项目之所以脱轨，主要的原因在于人们并未认真思考如何使项目保持进度、顺利前进。如果没有未雨绸缪，只是坐待问题发生，到那时候就太迟了。一个月前没有花 30 分钟思考这个问题，现在得浪费几小时或几天的时间去修正。这就是所谓的“被动式行动”（workingonreaction），而很多主管都是如此。

解决这种被动式行动的方法，就是化被动为主动，发掘潜在的问题，并设法避免。以我们在第一章的搬移房子的比喻来说吧，假定拖车原本缓慢而稳定的行驶，突然在转角边上动弹不得了，原来是有个高架道路拦住了去路，这下子只好掉头绕道行驶，结果前方又多了一些电话线，真是气死人了。本来在地图上看来都是平路，上路了才发现是这么崎岖，不仅要爬坡，还加上坑洞，怎么办呢？事先没有勘察好路况，没有准备好应付路况的辅助工具，临时手忙脚乱不说，还得老天保佑车子别翻了。要是主管用心做好“向前看”的工作，事先把路铺好，把障碍扫除掉，这些麻烦都不会发生。

主管不愿意认真地“向前看”，因为不看似乎比较轻松。您可曾听说过某个主管在面临一个始料未及的障碍时说：“啊，要是我事先多花点时间想想，就可以避免这件事了。”以我的经验，很少有主管会这样承认的，反而是在意外出现时不以为意，他们认为，意外难免发生，很正常嘛。

您得驱逐这种被动式的想法，要积极防范意外才行。有很多方法和技巧可以训练自己“向前看”（workproactively），但总结起来不过是一句简单的要诀：

定期暂停手边的工作，然后往前思考，随时做必要的修正，以避免未来的大障碍。

只要您懂得向前看，要事先避免大障碍并不困难。比方说，Windows 版的软件完成后，自然得继续开发 WindowsNT 版，或是准备一些示范用的软件。这些事看似小，但是如果不及早做出相应措施，后面的事情就很难收拾。就好像开车进入山区之前先把油箱加满是一样的道理，小小的动作可以避免在山区里加不到油而必须跋山涉水的窘况。

我已有十年以上的习惯，每天花 10 到 15 分钟思考下面的问题，并且列出答案：

有什么事情是我今天能做，而且可以帮助项目在未来几个月内顺利进行的？

这是一个十分简单的问题，但是如果主管定期用它来检讨、思考，必定能想到许多保护项目不受意外打击的妙方。请不要把这个问题的答案想得太复杂，事实上，答案应该简单到能在几分钟之内写完，通常是这样的：

◆订购 MIPS 和 Alpha 处理器的技术手册,以便 Hubie 需要时,随时可得。

◆发个 e-mail 给 Word 的工作部门，提醒他们如果有任何功能要加在这次的新版里，请在下周一前提出需求。

◆发个 e-mail 给负责 Graphics 的各位经理，我们需要的函数库预计在 3 个星期后用到，确认他们届时可以交货，没有问题。

没有一件事占掉我很多时间，但是却能省掉我将来可能面对的麻烦。订购 MIPS 的技术手册显然是件小事，但是手册若是 3 星期后才送来，万一这中间 MIPS 发生了问题，没有手册，工作就无法继续，那项目就延误下来了。订购手册要花多少时间，10 分钟够了吧？现在花 10 分钟让手册早点有得用，比起急需要用时再花 10 分钟订，再等 3 星期，那事情都甭做了。

经常用这种方式思考，您就会想到许多也许会发生的情况，也许 Graphics 小组会晚两周，也许 Word 小组不知道该早点告诉您需求，因为他们以为不急于一时嘛。如果没有经常“ 向前看 ”，到时候可能突然发现 Graphics 小组延误了。这下可惨了，所有的工作都会因此而赶不上进程，或者是 Word 小组最后一分钟才发现他们的需求因为告知太晚而来不及做出来。

当然，在理想的情况下，Graphics 的主管们会事先告诉我他们要延期完成，但是他们真的会这样做吗？根据我的经验，几乎是从未有过，因为主管通常会在项目已经明摆着延误了才会跟别人说——通常是到期的前三天。

每天都要问自己：

“ 有什么事情是我今天能做，而且会帮助项目在未来几个月内进行顺利的？ ”

错误的问题

在 WordforWindows 的开发过程中，我曾被要求检查一个函数库。这个函数库不是 Word 小组的人写的，是一个对话窗管理程序，目的是把应用软件和操作系统区隔开来，让应用软件的程序设计师在需要对话窗时，只要调用这个函数库就行，不必担心操作系统是 Windows、Macintosh 或是其他的系统。

我被交代找出这个函数库为何速度这么慢——Word 的项目经理们无法忍受每次调一个对话窗要等那么久，写对话窗函数库的程序设计师已经做了一些最优化的措施，也监测过程序的执行，但是 Word 小组还是不满意，认为这样简直是使公司声誉扫地，双方大吵了一架，而其他原本打算使用这个对话窗函数库的小组也开始迟疑了。

我去找 Word 的程序经理谈，以便更深入了解速度的问题，还要找出什么速度才可以被 Word 小组接受。他递给我一张清单，上面表列着很多项目，每一个项目旁注记有可容许的最长显示时间，也就是他们的质量规范指导。然

后我们开始测试，我们一只手按鼠标调出交谈窗口，一只手同时按下秒表计时，等待对话窗出现。“看！”他一边指着秒表说：“这个等太久了吧。”我倒觉得没那么可怕，所以我再测一次，咦，这次快多了，几乎一下就蹦出来，我说，第二次的速度一点儿也不慢，很明显符合可接受的质量水准。

“没错，第二次总是没问题，我们是针对第一次要求改进。因为只要停了一段时间没有使用，再调出对话窗时，速度又变慢了。”

等我了解了实际状况后，回到我的办公室查看原始程序代码，结果使我大吃一惊，原来问题的症结是 Word 的优化功能。因为 Word 的优化功能会取代 Windows 的程序代码置换算法（code-swappingalgorithm），当 Word 发现某段程序久未使用，就会自动把它从内存中删除，以充分运用宝贵的内存。因此，就算对话窗程序再快也无法通过测试，因为它必须重新自硬盘加载。

所以，让 Word 小组抱怨了好几个月的速度问题根本不是程序的问题，而是内存运用的问题。Word 小组抱怨对话窗太慢，却没有深入了解原因。单独测试函数库，似乎是够快了，同样的程序实在没道理在 Word 里面就跑得特别慢。

猜猜错虫在哪里？

很多程序设计师并未下功夫研究如何清除错虫。很多人都是直接去看原始码，随便修改某个地方，再执行一次看看错虫是否还在，如果还在，就回来改另一个地方，猜猜看是不是这儿错了，如此猜了一回又一回……直到错虫消失。

我之所以知道程序设计师用猜的方式找错，是因为当他们实在找不出错虫在哪里时，就过来问我，“接下来该怎么办？”这种心态好像是在游戏结束前“不玩了”，并不是真正在寻找错虫出现的原因。

以我的经验，最有效的除错方式是用 debugger（除错工具）设定一个暂停点，当程序执行到暂停点时，一一检视变量的值是否正确，如果不对，针对这个变量往前找它的变化，如果变量全对，就往下再设一个暂停点，继续执行到那里再来看变量对不对。即使程序极复杂，要在庞大的数据结构中转来转去，用这种方法总可以找到错虫，比起用猜的方法要有效多了，如果靠猜测来找错，即使猜中也不过是运气好，未必能找出错虫出现的真正原因。

我同时要提醒那些用看程序的方式来找错的人，应该去读一下安德鲁·科恩尼（AndrewKoenig）的《C 语言陷阱》（CTrapsandPitfalls）一书，里面有很多 C 程序的范例，看起来都完美无瑕，但都有微妙的错误。要不然，有些杂志定期会介绍一些附范例的 C 程序除错的技巧，也不妨看看。

用看程序的方式找错，是既懒惰又无效率的方法，用 debugger 来找错虫是最快又最方便的，观察各变量在程序执行过程中的变化，是非常有效的方法，绝对不要用猜或用看的办法来找错。

在这个案例中，Word 小组为了对话窗的速度问题抱怨了好几个月，使得做对话窗函数库的小组花了好多时间，辛辛苦苦地把程序最佳化，希望这是最后一次修改，可以满足 Word 小组的需求。其实，只要有人去看一下 Word 是怎么处理对话窗函数呼叫的，那问题早就迎刃而解了。

当然，不应该要求函数库的制作小组来测试应用程序，因为调用这个函数库的应用程序总有数十上百种。然而，函数库小组自己也该积极点，应该做一个专门测试本函数库的假应用软件，对函数库单独测试，就能为函数库本身的质量建立信心指导。在本例中，Word 小组已经大声地抱怨了很久，而且函数库小组也找不出 Word 在调用时有没有弄错，我相信在我之前一定有人发现了只有第一次特别慢的现象，却没有认真研究这一点，而把问题的焦点

放在把函数库最佳化上面，使得函数库小组无形中背了个黑锅。

身为主管的您，得随时睁大雪亮的眼睛，看看是不是有个悬而未决的问题，一定要有个人（或是由主管自己）来负责研究到底哪里出错，也许这种研究既花时间又无聊，但总比灾难发生之后再来花好几个星期收拾残局要好得多。

不要浪费时间在错误的问题上，一定要先确定真正的问题在哪里，然后才去改正它。

荒谬的菜单

有一次，Windows 使用者界面的技术经理惊慌失措地跑来找我，告诉我麻烦来了，应用软件的小组提出了一项需求，需要好几个星期才能做出来，可是目前离期限却相当短，真不知道如何来满足这个需求。于是我问他，到底是什么样的需求。

“他们希望我修改下拉式列表框（drop-downlistboxes）的格式，因为他们要能够在对话框外面使用列表框，而且可以去掉滚动条（scrollbars），还要能够把列表框中的某些项目弄成暗色以表示不可选取，他们还希望在鼠标移到列表框的某些选项上时，自动弹出一个新的列表框，而在鼠标移开时自动消失。”天哪！

听完他的陈述后，我必须承认，如果真的要把这些需求全部完成，我们自己的进程就完蛋了；但我并不太担心，因为根据我们的原则，不是每个小组都用到的程序是不应该放在函数库中的。我最初的想法是为他们提供标准化的列表框，然后要什么花俏的玩艺儿就让他们自己去写，但我同时也很好奇，应用软件的小组为什么要这些奇奇怪怪的东西？我想一定是个前所未见的、很特别的用户界面吧，所以，在我答复他们做还是不做以前，我请技术经理去了解一下这个大需求的原因。不一会儿，他们笑嘻嘻地回来了。

“他们要用列表框来仿真级连式菜单（hierarchicalmenu），就像在 Windows 和 Macintosh 一样。”

现在我知道他在笑什么了：因为我们其实已经有新的函数库，可以支持阶梯式菜单，而其他的小组还不知道这件事。

我写出这个故事，因为大部分的小组在提出需求时，都不解释原因，这种情形太普遍了，即使在工作以外也很常见。住旅馆时，我有时候会去餐厅吃早午餐，偶尔会有客人跑进来，看到大家都在吃早餐，就对服务生说：“你们的早餐时间到几点才结束啊？”我看到这位饿慌了的客人急着向后转，喃喃自语：“我真想吃午餐。”然后在服务生有机会向他解释现在也可以点午餐之前，已经飘然离去。午餐的菜单明明已经摆在钟旁。

这位仁兄明明想要吃午餐，却问着早餐的结束时间，他该问的是“现在供应午餐吗？”也许是思绪偏离了主题，因而问错了问题。这种事经常发生的。即使是我，多年的职业训练已经使我懂得问正确的问题，我还是会问我太太贝丝，她什么时候看完足球赛回来，而我真正想要知道的是今晚几点开饭。

问了错的问题，而导致错的答案，这种事情似乎已经屡见不鲜。现在您知道了，每个人都有这种倾向，您可以试着让自己养成习惯，先找出对方的目的，再来了解他的问题。如果从他的请求中无法看出他的目的，您可以反

问他，在还没弄清楚他究竟想要做什么之前，不要贸然答应他，宁可拒绝他的要求也不要浪费这种时间。

人们开口要求的东西未必是他真正想要的。处理他的要求之前，请务必确定他究竟想要做什么。

明确定义需求的范围

如果您能够先明确定义自己的需求，再向别人提出，这是个避免在沟通上发生误会的好方法。假定应用软件的小组用这样的 e-mail 提出需求：

在这次的新版本中我们需要阶梯式菜单。因为与下拉式列表窗很类似，我们认为应该可以利用列表框来仿真级连式菜单，所以想拜托您修改列表框，好让我们……

如果我们的技术经理收到的 e-mail 是这样，他就不会吓得跑来找我，而能立刻告诉他们我们已经有了阶梯式菜单，不需要用列表框来仿真。更重要的是，我差点就直截了当地拒绝了他们，若真如此，他们得多花上好几个星期去做我们已经完成的东西。

如果您能很清楚告诉别人，您想要的究竟是什么，这样别人才能给您真正需要的帮助，而不是做一些似是而非的虚工。

就是说不

倘若我懒得追问为什么需要那怪异的列表框，而是一口回绝了他们的需求，您想他们会不会说：“好吧，我们了解。但是无论如何还是谢谢你们。”也许会。但更可能的是一场争辩，他们会说，函数库就是为了应用程序的需求而设计的，我们有责任维护函数库，不断添加新的功能，让它随时满足应用程序的需求。

当然，平息这场争论最简单的办法是忍气吞声地同意他们的需求，这也就是函数库开发主管们最常碰到的烦恼。负责函数库的主管们宁可息事宁人，也不愿为整个产品或自己的工作团队坚持最佳的选择。

有时候，对方的请求其实是非常合理的，您也很想同意，但因为您的日程排满了，实在爱莫能助，您也只好对他们说“不”；然而，在我的经验中，很多主管为了避免冲突，仍然会同意这样的请求，只是不知道该如何如期完成这些过多的工作，只想到时候再说，也许船到桥头自然直。事实上事情很少这么容易——船上若是载了太多货，就是船身直了也过不了桥啊。

这些主管不了解，勉强自己接下不可能完成的任务，实在是以长痛代替短痛的做法，而且长痛的是整个团队。此外，到时候无法如期完成，倒是害得需求小组因此而做了错误的日程安排，所以，最好的做法还是老老实实地拿着您的日程表，对需求小组说明自己心有余而力不足的情况，设法安排一个折衷的日程或工作内容。

请想一想其中的差异。当别的小组向您提出他们的需求内容时，大概都会把期限排在未来一段时间之后，如果您没办法满足对方的需求，至少在这一段时间内您可以和对方商量出其他的解决方法。只有两种情况会让您成为坏人，一是直接拒绝而不试着想别的办法，二是无条件答应请求而最后食言。与其现在心存侥幸，到了时限却实在做不出来，以致连累需求小组一起延误，倒不如现在想个折衷的解决之道。

我们这样想吧：如果您想向银行申请房屋贷款，其中一家银行马上拒绝您的申请，另一家则是一口答应，但等到您已经签约成交时却又反悔。您觉

得哪一家比较可恨？

我并不鼓励您在进程排不出来的情况下立即回绝，我只是强调绝对不要答应别人自己做不到的事情。也许您很想说自己做得到，但那只是希望。通常主管是眼看日程表排得满满的，或是已发生延误，就够紧张了，如果再加接下一件工作，就等于确定会延误，其焦虑可想而知。

对身处前线的主管而言，协调好这些彼此矛盾的工作与日程，实在很不容易。但若是只顾一时的面子，几个月后公司的大老板就会满脸怒容坐在您的桌子上质问，为什么广告都上了书报摊了，你才承认进度落后。

别放弃任何可行的方法

要协调原本就对立的双方，有一个要诀，就是寻求真正的解决之道。如果您对需求小组说“不”是因为您很确定您的小组人员实在无法在他们的期限之内完成所要求的，您务必得协助他们寻求真正的解决之道。也许需求小组自己也可以做这些工作，也许有一部分您的小组可以帮忙，也许他们可以请求公司里其他的部门支持，说不定已经有类似的程序代码已经完成，稍加修改就可以运用。不试试看的话，谁知道呢？

说“不”也许令人不快，但这才是勇敢面对问题的态度。说完“不”之后，就是设法解决问题的开始；明知不可行而答应，就是问题发生的开始。

绝对不要答应别人自己做不到的事情，这样对双方都有益无害。

我无法说“不”

有一回，word 小组请求我的使用者界面函数库小组做一些成本很高的工作，当时我们的日程排得满满的，如果决定满足他们的需求，势必使我们进行中的工作发生延误，这样就会影响到 20 个以上的小组。我向 Word 小组解释我们无法做到的理由，以及我们能够做的，但就是没有办法符合他们对期限的要求。我向他们建议，如果一定要有这些功能，其实他们也可以自己做，然后由我们帮忙写文件、测试以及日后的维护。Word 小组不太高兴，认为我们本来就应该做使用者界面，况且这些功能是大家都会需要的，这一点并没有错，但这并不能改变我们无法在期限内完成的事实。我们为了这个问题僵持了两个月，我终于屈服了，答应他们完成这些功能，并且从我带的另一个项目中抽调一位程序设计师来帮忙。

很不幸，我实在无法找到合适而足够的人手，后果真是不堪设想。Word 小组的期限过了几个星期，我才完成工作，他们都气得要杀人了。我也延误了正常的进度，有 20 个左右的小组受到波及，到处都怨声载道。我非常后悔当初没有坚持说“不”，现在搞得每个人都不高兴，唉！

你无法让每个人都满意

身为主管，您一定会面临各种要求，为了工作的效能，您得学会在适当的时机，适当地说“不”。无论您说得多么委婉，对方都不会喜欢被拒绝，他们可能会认为你错了，然而，您必须了解自己无法让每个人都满意的事实，您要做的是协调，而不是完成每一件事，那是做不完的。

如果您负责做共享函数库，某天某个小组要求增加一些函数，这些是只有他们需要的，如果您说“不”，他们会很难过，如果您说“好”，其他的小组群起抱怨函数库怎么体积变这么大。这是永远无解的矛盾问题，谁教您负责的是共享函数库呢。

当您碰到互相冲突的需求时该怎么办？有没有比较有效的办法？这就是我在前面强调详列项目目标的用意之一了。您的目标既然是提供对每个小组都有用的函数，不符原则的需求就应该婉拒。当然您一定会受到抱怨，您不

妨向需求小组解释原因，万一您今天破例做了“专用”的函数，大家都会来这样的要求，最后函数库成了大杂烩，就失去了函数库的意义，这不花您太多时间的，对方再怎么不悦，您还是要耐心解释，问题总得交代清楚。

每个人都不愿意被别人讨厌，这是人类的本性。但是身为软件开发的主管，您就必须掌握这个道理：如果您希望每个人都满意，最后您会焦头烂额，什么事都做不成。

以我的经验，人们虽然不喜欢自己的需求被拒绝，但如果您有充分的理由，他们还是会了解的，并且感谢您的用心。

不要为了讨好别人而伤害双方的工作进程，您永远要根据自己的目标，做适当的决策。

如果不是函数库项目

我以函数库的实例讨论互相冲突的需求，您可能不是负责开发函数库的，无论您负责的是什么项目，我们所讨论的观念和做法几乎都可适用。基本上，您难免会遇到外部的要求，提出需求的人甚至可能是行销小组，或是来自使用者的反应，即使是极端机密的软件开发项目都可能有人来插上一手，您必须学会摆平冲突。

上级的建议

一位主管在做任何决策时应该以项目目标为最大的考虑，不要企图讨好别人，尤其不要盲从上级提出的建议。我不是主张反抗权威，而是强调上级也是人，一样可能犯错，他们的建议不一定是好的，尤其是他们可能不了解您的目标、决策优先级，以及您所必须面对的技术挑战。如果您想做一位出色的主管，您必须非常认真地衡量所有的建议，不论是谁提出的，您都得确定其符合项目目标才能采纳。

如果上级要求您做一件事，而您认为不妥，那您应该在着手进行之前向上级说明您的想法。也许，上级会同意您的想法而放弃他的建议，也许，上级会赞许您的想法，但仍请您考虑他的意见。不论结果如何，起码经过沟通对彼此都有帮助。

有一次，我查阅一位资深程序设计师的程序。对其中有一些很明显属于设计上的缺点我很惊讶，我不太相信这么一位优秀的程序员会写出此等程序，于是我问他为什么这样设计。

“是柯比设计的，我只是照做而已。”柯比是他的项目经理。

我觉得好奇的是他自己的看法：“你自己觉得这样设计好吗？”

“如果是我设计，不会用这种方式。”

“你在写程序的时候，应该能感觉到这一点吧？”

他轻描淡写地说是的，并且耸耸肩道：“但我刚到微软不久，柯比是我上司，我认为他应该比我有经验，他这样设计也许有特别的用意。我可不想把事情搞砸。”

事实上，柯比并没有比这位程序设计师有经验，他只是比较幸运，在微软待得比较久罢了。

对于这个观念，我还有另一个很有意义的实例。当时我领导Microsoft680x0的交叉发展系统（在Macintosh和PC之间使用），我的顶头上司，也就是有权变动我计划的人，名叫莫特。每隔一段时间，莫特就会

到我的办公室来问我有关 680x0 的 C/C++编译器项目进展的问题。他每次来必定会问一个问题：

“ FORTRAN 进行得怎么样了？ ”

莫特很清楚我们根本不做 FORTRAN 的编译器，但是他喜欢 FORTRAN，而且他相信一个好的 Macintosh 版 FORTRAN 编译器一定很有销路。事实上，如果把我们做好的 C/C++编译器修改成 FORTRAN 编译器并不太难，我和莫特都知道，微软开发编译器都是遵循大部分教科书所描述的三阶段方式：

1.前端处理（frontend）：把高级语言（如 C/C++，FORTRAN 等）解析成中间代码。

2.优化（optimizer）：对中间代码做执行时间的优化处理，诸如程序代码搬移、调整，同次表示式消去等等。

3.后端处理（backend）：从经过优化的中间代码，产生最佳的执行代码。

当然，实际工作比这三点略述要复杂得多。但是由这些您可以看出来，要推出一个 Macintosh 用的编译器，只要重新写一个后端处理，就可以把英特尔 80x86 的编译器转成摩托罗拉 680x0 的编译器。

从理论上说，一旦我们完成了 680x0 的后端程序，我们就可以用它来把所有的英特尔 80x86 的 C/C++、FORTRAN、Pascal 的后端处理替换掉，这样就做出摩托罗拉 680x0 的 C/C++、FORTRAN、Pascal 的编译器了。这就是莫特对 FORTRAN 特别有兴趣的原因。然而事实上，我们要做 FORTRAN 编译器的话，必须把后端处理完全做好才行，当时我们只完成了 95%的 C/C++编译器后端处理。

每一次莫特问我有关 FORTRAN 的问题时，我的回答也是千篇一律：“ 我们还没有开始。 ”然后再补充道：“ 主要是因为 FORTRAN 编译器的关键组件——后端处理，尚未完成的缘故。 ”

莫特也许是对的，FORTRAN 编译器在 Macintosh 上也许会有很好的市场，但是他忽略了项目最优先的目标。就算是有再好的市场，也实在没道理要把做了一半的项目停掉，更何况他自己也认为 C/C++编译器的市场潜力应该比 FORTRAN 更好。要不是莫特的个人兴趣，我不会和他讨论这么多次，他的个人兴趣已经蒙蔽了他身为主管的智能。

您必须保护项目不受外界的左右，尤其是当这种操控来自特权人物之手。像莫特这样，想法明明是错的，您却得服从上意。如果是我刚开始担任主管时，我会向压力低头，服从上级的意思，现在我学会了独立判断，不论是谁的建议或要求，我一定会先问自己：“ 这样做对产品有没有帮助？对于目标的完成有没有策略上的价值？这样做是否会使我忽略了更重要的事情？这样做的成本和风险会不会太高？ ”您必须好好想这些问题，答案如果对项目无益，您就不应该照做。

是您在为项目负责。

不要让任何人的建议阻碍项目的进行，包括上级的建议。

真正的成本

为什么莫特会认为 Macintosh 上的 FORTRAN 编译器值得开发呢？是因为很多人想买吗？还是因为 FORTRAN 对 Macintosh 有特殊意义呢？其实都不是，真正的原因是他自己对 FORTRAN 情有独钟，他希望能够得到一份免费的

MacintoshFORTRAN 编译器，如此而已。

我对能附送那些副产品的兴趣，绝不下于任何人。那是一种欣慰的感觉，因为自己是一位优秀的软件开发专家，才能有这么多精彩的创意，额外做出这些副产品。但是这些副产品对公司或产品都没有策略上的价值，充其量只是一种消费者回馈。如果它们有策略上的重要性，早就放在产品设计的计划里面了。

有趣的是，我们也可以把 C/C++编译器的前端处理换成 Pascal 的，那么我们又多了一个 Macintosh 的 Pascal 编译器了。但我从来没打算这么做，虽然我们都知道 Macintosh 曾经有很长的一段时间只有 Pascal 程序,苹果计算机所有的手册和范例程序都是采用 Pascal，而且苹果计算机的 Pascal 编译器市场并没有太大的竞争。现在的情况则完全改观了，苹果计算机如今已是 C/C++语言的天下。但是如果我们要在 C/C++编译器产品之外附送一种语言的编译器，那 Pascal 绝对比 FORTRAN 要适当。

莫特对 FORTRAN 编译器有兴趣，纯粹因为它免费，而不是基于任何策略性的考虑。但是 FORTRAN 编译器真正的成本可未必那么便宜，如果我们要把 FORTRAN 编译器引进市场，我们要做的事情是：

◆完成后端处理，目前剩下 5%尚未完成，这大概要花几个月的时间。

◆想办法让 FORTRAN 和 Macintosh 的操作系统交谈，因为 Macintosh 向来是用 Pascal 写的，而 Pascal 的记录，FORTRAN 无法支持，所以这方面有个鸿沟，需要另外设法跨越。

◆撰写手册和线上求助的文档。

◆完整地测试 FORTRAN 编译器，包括它的链接能力，除错工具，以及其他的各种辅助工具。

其实还有一些额外的行政工作也是不可或缺，例如训练产品支持小组等等，以上只是几项开发性质的工作。现在您觉得 FORTRAN 编译器很便宜吗？好吧，即使前面三项可以利用 C/C++编译器的东西过来改一改，但测试工作没有快捷方式，它可是完全无法打折扣的，任何产品都要有同样严谨的测试工作，FORTRAN 编译器必须通过和 C/C++编译器同样一丝不苟的测试程序。

所以 FORTRAN 编译器绝对不是白吃的午餐，虽然比起完全从头开始的 C/C++编译器，成本低廉得多，但是“ 便宜 ”可能仍然是高价——只要问问最近买过二手波音 747 飞机的人就知道了。

免费的软件或功能其实并不存在。想想看，您每次听到广告中的‘ 免费 ”，内心的反应恐怕是抗拒多于接受吧，感觉上，这就像是听到“ 参观某工地表演就可以免费参加夏威夷六日游大抽奖 ”一样。大部分时候，这类的免费软件都没什么重要性，只有极少极少的例外会中大奖，如果您希望确实掌握项目朝正确的方向进行，就请专注在有策略价值的工作上，不要被一些花俏的噱头分心。

天下没有真正免费的软件

炒鱿鱼宏

不只是上级会提出不合理的请求，行销人员也可能会。有时候为了争取一张大订单，行销人员会为这位大客户要求一些匪夷所思的修改。您必须捍卫您的产品，避免被这些不正常的杂音所扭曲。

当我负责 MicrosoftExcel 时，行销人员要求开发一个炒鱿鱼宏（LAYOFFmacro），而这个宏的内容您大概已经猜到了，就是一份名单，用随机的方式挑出其中一些人来遣散，因为大公司想要裁减人员，又不想造成反弹或不公平之议，干脆用计算机随机选取，反正遣散谁都一样，正好利用 Excel 来推卸责任。

如果您熟悉 Excel，您就知道并没有这个宏。这个宏我做不来，所以我拒绝了这个要求，因为我认为这会伤害我们的产品。而我的上级也同意我的看法，所以我们成功地阻挡了这个要求。但行销人员持续要求了好几个月，他们认为要有这个特色，才能让顾客更需要 Excel。

这个炒鱿鱼宏成了我们开发小组的大笑话。“这样好了，我们就来做这个宏，里面做点手脚，凡是遇到我们的名字就忽略过去，这样我们就永远不会被炒鱿鱼了！不，还可以更好，我们把程序写成让行销人员的名字优先中奖，这样一来，行销人员永远最先被解雇！”当然我们不会真的这样做。最后的结果是，行销人员另外写了一个使用者自订的宏，去满足那位大客户，而我们始终没有在产品中加入这个讨厌的宏。

就我多年的经验，这类荒唐爆笑的要求其实很少见，因为行销人员一点儿也不想伤害产品，相反地，他们和开发人员一样渴望有最好的产品，只不过有些时候他们并不那么清楚怎么做才是对产品最有利的，因此会要求一些也许不应该开发的功能特色。这种不该加入产品的功能特色有两类，一是不符合产品的未来发展方向，仅仅因为这项功能属于杂志上所列评比清单中的一项；二是特殊客户的要求。有时候，功能齐全不一定是最好的，有自己独特的风格更重要，在产品中加进了太多枝枝节节的东西，可能使产品过度膨胀，也花费了太多开发人员的时间精力，未必是值得的。

遇到这种情形的话，您该怎么办呢？以我的经验，您应该探究这个需求背后的动机。好好想一想。如果行销人员跑来对您说：“惠普公司（HP）的 HP12c 商业用计算器有五项功能是我们的电子表格所不能提供的，希望你们把这些功能加上去。”对产品而言，加入这些功能有没有策略上的价值？能不能真正改善产品？或者只是因为行销人员这么想：“嘿，别人都有这个，就我们没有，须得加紧赶上。”这些功能也许真的很重要，而前一版中来不及加入，也许是当时觉得不值得开发，现在您仍然要坚守原则，不值得开发的功能就不要做。

策略性行销

我希望上面的例子不会造成您对行销人员的负面印象，因而对他们的建议不予理会。有时候他们的要求并不妥当，但是他们通常都很有道理。至少我的经验是如此。

有时候，行销人员要求增加某个功能，由产品的观点来看是没有策略上的意义，但是由行销的观点却非常有必要。比方说，以产品的观点而言，实在没有必要支持 23 种不同的档案格式，使用者只需要一种格式就够了，即使是要与别的软件互读档案，也只需要业界标准的几种档案格式；但是就行销的观点，如果产品没有非常足够的档案兼容性，就会降低顾客购买的意愿，或者，如果他过去所使用的档案格式是本产品不兼容的，而他不能失去过去的资料，那本产品对他就没有什么吸引力了。

如果您认为某些修改产品的要求应该予以拒绝，因为不能改善产品，那么请考虑是否能大幅增加销售量。炒鱿鱼的宏是应该拒绝的，因为它对产品有害，只对少部分大客户有用，对大部分的小客户是无用的，而且对产品形象颇为不利。

如果行销人员常常阅读杂志，很在意上面的评比清单，您可能免不了这类的问题：对于功能特

色的要求，是为了在评比的项目上领先别人，而不是产品本身策略上的需要。行销人员很可能会看到对手有一个很受注目的特点，或是很红的卖点（尤其是对手的销量因此增加时），就强烈要求开发人员跟进，这时候您的产品方向可能会因此转变，您得特别小心，产品的长期目标不可因此而偏废。

应该开发策略上具有重要性的功能，而不是把媒体的评比项目都做齐全。

很酷，但并不重要

在第 1 章中我提过一位使用者界面函数库的组长，我和他一起检视函数库的工作清单。清单中有一项六星期内要完成的工作，是要容许协作厂商所开发的独立小程序直接挂入微软的非窗口应用软件。其目的是将小算盘、小作家、时钟等 Windows 和 Macintosh 必备的附属应用程序，也能在非窗口应用软件（主要是 MS-DOS）中用到。我认为这项功能很有趣，但对于微软内部 20 个使用本函数库的小组而言，并不重要。

于是我问这位组长，是谁要求加入这项功能，他说没有人要求，这是因为前任组长觉得它很重要，才会列在工作清单上。我再问是否有哪位组员对它很有兴趣，这位组长说不知道，不过他加了一句，如果我把它删掉，被前任组长发现的话准会跳起来。

我猜想如果前任组长对这项功能如此坚持，应该是有某个小组真的想要，只是现任组长不知道罢了。所以在我删掉它之前，我先问过使用本函数库的 20 个小组，有没有人对这项功能很需要，或是对它很有兴趣，结果我得到的回答几乎都是清一色的：“噢，我们听说过。是某某人一直说服我们这个东西非常重要。”

大部分的应用软件小组根本就不想要这个功能，有许多更有趣的工作等着他们。况且，要完成这个功能的话，附属应用程序集就会为了能够与协作厂商的程序交谈而变得复杂，那么，应用软件与附属应用程序集之间，就需要大量的沟通。他们并不想要有个桌钟或计算器摆在画面上，他们要的是真正能加强应用软件功能的东西，例如文法检查或是其他的重要工具。提供一般用途的使用者界面固然不是坏事，但这项功能本身要花六星期来做不说，它还连带使其他的部分也复杂化，我们没有时间做这项工作，也不愿把现有的程序弄复杂。

到此为止我几乎已经决定把它删掉了。但我还是先找了前任组长谈，了解他的看法。他对于这项功能将被删掉的事颇感失望，但也别无他法。他主张开发这项功能的理由是，这是一项有趣的挑战，很能磨练写程序的技巧，而在 MS-DOS 的环境下使用弹出式的附属应用程序是一件很酷的事。他说的不错，但除了酷之外，没有其他更具说服力的理由。

至于协作厂商呢……

协作厂商也许会乐意见到这个界面。如果非视窗软件可以让一些小而有趣的应用程式弹出执行，那么协作厂商就可以利用这个“利基”，做出许多特别的小型应用程式，外挂在微软的应用软件。这一点始终没有实现，因为我决定不开发这个界面，但我是在审慎考虑过协作厂商所带来的利益之后，才这么决定的。

倘若这些外挂式小型应用程序真的这么吸引人，协作厂商一定会开发许许多多的新奇作品，使用者也会喜欢，最终会增加产品的需求量。但是，小算盘？记事本？时钟？没有人会单为了拥有这些

小东西，这决定购买微软的文字处理器、除错器等应用软件吧？

更简单地说，使用者并不需要这些外挂式小型应用程式，因此产品也不需要它，为了一项使用者并不在乎的功能要花费六个星期的工作时间，那是浪费。

事实上，这项需要六星期的工作并没有任何策略上的价值，对我们的使用者界面函数库的目标没有帮助，对于我们的 20 个需求小组没有用处。这件工作之所以列在清单上只有两个理由：一是有趣，二是够有个性，对于在 MS-DOS 的文字模式中待习惯的使用者而言，拥有 Windows 和 Macintosh 的附属应用程序确实很有个性——说服他们改用 Windows 不是更有个性吗？

软件产品的开发，不能只为了有趣、挑战性，或是够有个性够令人眩目。

这样比较好吗

有时候日程表中会突然插进一件工作，因为它似乎非常重要，但是如果您再从策略的观点考虑一下，也许就会改变。举例来说吧，Excel 有一个剪贴板（clipboard），总是让我看了很不舒服，因为它的处理方式与众不同，不会留在内存。这个剪贴板一点也不难用，没有 bug，也不是说它处理方式很怪异，但这就是非标准模式。因为它的做法和所有的 Windows 和 Macintosh 的剪贴板都不一样，所幸操作上是完全相同的，使用者鲜少会注意到其中的差异。

我的原则是遵循标准重于一切，特别是关于使用者界面的部分。所以您想想看，如果我是 Excel 的程序经理，我一定会认为“把剪贴板标准化”是一件很重要的工作，而把它排进日程表里。是的，我确实认为它很重要，但是从策略上来看就不重要了。此外，修改剪贴板的程序代码会有一个不良的副作用，就是可能会破坏使用者自己写的宏。

如果我是 Excel 的程序经理，我会非常想把剪贴板标准化，但是考虑到使用者的感受——可能会为了需要修改旧的宏而恼火，可能会搞不清楚剪贴板到底怎么回事，我还是不能改剪贴板。反过来说，如果使用者对现在的剪贴板不满意，觉得很难用的话，我就会把修改剪贴板列为第一优先考虑。然而，现在的 Excel 剪贴板对使用者而言，和其他的剪贴板完全相同，所以并不是非改不可。

还有一个很重要但无关策略性的工作，就是写程序的风格（programstyle）与命名原则（namingconvention）。写过程序的人都知道，程序的风格有点类似文章的文体，每个人喜欢的分段方式不完全相同，对于文件名称、变量名称和函数名称的命名原则也不一样。这没有绝对的标准，但是同一套的程序最好有一致的程序风格和命名原则，写和读程序比较方便，维护时也较不易弄错。一般而言，软件在刚成形时源代码并不太多，就是有点乱也无妨，随着程序代码的增长，程序风格和命名原则应该统一，所以会有需要把过去随便取的名字改成统一的原则，有时候是因为统一的命名原则在后期才被确立下来的缘故。

假定有一位项目经理决定所有的函数都要有一段标注，说明这个函数的功能和各参数的意义。这是非常合理的。我所质疑的是接下来的行动，程序设计师就得停下所有的开发工作，花几天或几周的时间，把所有的函数加上这个标注。如果项目经理决定更改命名原则，那就更严重了，要更改所有的

函数名称、变量名称成本是极高的。这些工作对于日后程序的维护很重要，但是没有策略上的重要性，因为它对改善产品完全没有帮助。

不错，您可以把这类一时没有生产成果的工作当成对日后程序可维护性的投资，这些努力终究有助于改善产品的，但是要停下所有的开发工作代价实在太高。如果您想取其优点、去其缺点，我的建议是不要停下开发工作，而要求程序设计师在这次用到的函数上面，顺手补齐它的标注。如此一来，程序设计师一周下来，多花的时间也许只有一个小时，当然如果欠缺标注的函数很多，时间就会花得多些。这种方式固然无法非常彻底，但无论如何，是比较符合投资效益的。

然而，如果您要停下所有的开发工作，在全部的程序中加进除错用的程序代码，那就是另一回事了，这是能改善产品质量的，而且成效立见。

偶尔我会看一下乌斯奈特（Usnet）的文章，我曾经看过一位程序设计师写道：“我们正在把所有的 C 程序改写成 C++，但我无法了解 C++如何做某事某事……”我看到这里，忍不住战栗了一下，我希望这位程序设计师——事实上是他的主管不要浪费大量的时间重写程序，这样太伤害程序设计师的生产效率。

同样的道理，您可能认为把汇编语言的程序用 C 全部重写，会使程序代码更具可移植性，日后也比较容易修改和维护。但是，用 C 重写 Pascal 程序，或是用 C++重写 C 程序，并不能直接翻译，有很多地方是要从地基再开始盖到顶楼，所以我个人非常怀疑这种做法。我猜想，这类的重写工作大都是来自沉迷于科技潮流的主管所下的命令。当 C++刚开始蔚然成风时，微软内部也有一些程序设计师很想用 C++来写程序，而忽略了原来的程序代码是否能继续运作的问题，他们固执地认为，就是要用最新的技术，重写所有的程序。所幸大多数的微软人是很冷静的，这种主张重写所有程序的声浪终于逐渐平息，公司最后决定的政策是对象导向只用在新开发的程序，必须是真的有策略上的需要才考虑重写。

不要把时间浪费在无法改善产品的工作上，即使这么做在将来会有潜在的利益，也要与现在投入的时间成本做个衡量。

生产效率的迷惑

用 C++重写 C 程序的理由中，我最常听到的是对象导向的方式可以提高程序设计师的生产效率，他们也许是对的，但是忽略了一个很重要的细节，那就是所有的时间都要耗在重写，用 C++重写 C 程序不是逐行翻译，而是完全彻底重写。

如果您带的程序设计师中有人向您建议把 C 转成 C++，您不妨了解一下，他们是真的认为本产品有必要利用 C++特有的优点，还只不过对 C++有兴趣，想借此机会学习一种新的技术。如果确定要重写，您必须慎重衡量二者之间的利弊得失，重写所耗费的大量时间成本，是否能得到相对的产品改善和获益的结果。

避免干扰

到目前为止，想必您已经很清楚什么样的工作才是应该全力投入的：与目标一致的策略性工作。但是这样还不足以让您保持进度，您还得尽量撇开“免费”的附送软件，克制大家追求“酷”的欲望，尽量减少对产品没有改善效果的工作。如果您无法学会说“不”，或无法了解别人真正需要的是什

么，您就会发现自己深陷泥沼，净做不该做的事情。

要想确保项目依照计划进行，没有脱序或偏离，其关键就在领导者完全明白该做什么，并且不让该做的事受到不当的干扰。这也就是为什么详细的项目目标是那么重要了。

重点提示

不要让意外出现的问题打乱项目的脚步，如果您要项目顺利进行，您得花点时间思考未来。今天做个小小的动作，可以防范许多意想不到的问题，即使真发生了无法避免的灾难，您也能在风雨中稳稳掌舵。如果您随时问自己：“有什么事情是我今天能做的，而且可以帮助项目在未来几个月内顺利进行？”您就会知道该采取什么行动。

在您准备解决一个问题之前，先确定您找到了问题的症结。还记得对话框函数库的例子吧，Word小组的抱怨不小心误导了问题的症结，使得函数库小组极力设法优化，却徒劳无功。因此，在您企图解决任何问题之前，请务必确定已经对问题有了彻底的了解。

在投入大量时间于任何一件工作之前，请想一想这件工作是否能满足真正的需求。您还记得那怪异的下拉式列表框，其实应该是个级连式菜单的例子吧。当您接获任何一项要求，最好了解一下背后的原因，提出这项要求的人究竟想要做什么。这样可以节省很多宝贵的时间。

基于非常多的原因，有些主管很难对提出需求的小组说“不”。在比较严重的情况下，主管会“知其不可而为之”，答应对方自己做不到的承诺。如果您发现自己常常不好意思说“不”，请将心比心替对方想一想，万一到时候做不出来，是不是会造成更严重的后果？如果您是需求小组，您对该到货的东西迟迟不见，是不是焦急又恼怒？您必须对其他的小组负责，就像您希望他们也能对您要求的工作负责一样。

每当您接到一项请求，要您在产品中加入某一项功能特色，请先想一想这项工作在策略上重不重要，如果不，就不要开发它；至于这个功能特色是否免费、是不是很酷、竞争对手有没有，都不是重点。特别是有些整组的功能，它们看起来很重要，因为它给您一种没有它就不够完整的感觉。您必须牢记，产品的策略性比完整性重要。如果您不敢确定这项功能特色是否有策略上的重要性，只要想一想这项请求的动机，就可明白大半了。

第 4 章　走极端的狂热
Chapter Four

在您读完前 3 章后，您或许会对我有点误解，以为我是那种专门压榨属下，唯生产力是图的主管。的确，我非常重视生产力，但我的目标绝不是榨取属下，而是尽力营造愉快的工作气氛，让属下很自然地发挥最大的生产效率。

您是否有过顺利完成项目的那种无法言喻的成就感？或者至少有过一次，您带着万分愉快的心情走出办公室，觉得自己今天好充实好满足。试着回忆那个愉快的日子，那天是在开会、写报告、访谈、传 e-mail，还是一整天不受打搅，终于想出一个绝佳的点子，或是完成了一件很棒的程序设计杰作？不必说，您我都知道答案。到目前为止，我从没见过一个热爱开会或写报告的程序设计师。

以我身为一位软件开发团队的领导者来说，我极力想达到的目标就是创造一个理想的工作环境，让所有的人都能尽情发挥，创造出足以自豪的产品。为了做到这一点，我很努力地让程序设计师免除不必要的报告和会议，尽量不是为了赶进程而压力过大，任何与改善产品无关的杂事我都尽量排除；我希望程序设计师安心工作，专心开发产品。在本章中，我会告诉您为什么这些杂事会伤害生产效率（尽管这些杂事随处可见，似乎每家公司都是如此），我也会告诉您如何用比较简单有效的方式达到同样的目的，而不要让这些杂事淹没了程序设计师以致影响了他们的生产效率。

无人理睬的报告

有一次我出差回来后，我的上司找我到他的办公室，问我有关这趟公务的细节。我们谈完之后，他要我把所讲的每件事都一五一十地写成一份出差报告。对我而言，这实在是非常浪费时间的事情，所以我问他报告真的非写不可吗？他很肯定地说是的，所以我花了一整个下午的时间写报告，项目的事情只好暂时搁在一边。

不到一个月，我的上司又把我叫去，口头询问一个问题，而这是我在书面的出差报告中早就详细说明过了的，我能体谅他日理万机，无法记住报告中的每一项细节，但我忍不住怀疑他到底有没有看过我写的报告，所以我就单刀直入地问他。哦，答案可真是令我心碎，他一个字也没看，我把报告交给他的下一秒钟，报告就进了档案柜。

我忍不住有点恼怒：“既然你根本没打算看这份报告，为什么一定要我写呢？”

我没想到他露出惊讶的表情说：“每个人都要写出差报告，这是公司的政策。”

随便哪个理由都比这个好。他可以告诉我，研究显示人们对自己写过的东西记忆会深刻些，那我也就认了；他也可以告诉我，他想要把我的报告呈给上一级的主管或别的小组看，把报告中的心得分享给别人，那我也可以理解。但是他明明就不想看，却仍然要我写出报告，再丢进档案柜，这未免太荒谬了。这就是过度死守规则的错误示范，规则是该遵守没错，但把规则当作铁一样的法律就太蠢了。任何公司都一样，凡是没人看的报告就不该写，

除非是要作者加深自己的印象，即便如此，也不是每一次出差都该写报告。奉劝各位明智的主管，绝对不要像我的上司一样，只知道墨守陈规，而从未考虑对正经事的伤害。

炖肉必须切掉两端

有很多事情因为长久以来大家都这么做，后来就忘了究竟为什么，结果即使在原因消失之后还是习惯这么做。有一个故事把这种情形描绘得非常贴切：

有一个小男孩，看到妈妈每次炖肉时都要把肉的两端切掉，觉得很好奇，就问妈妈为什么。妈妈回答说：“喔，那是因为我妈妈教我炖肉要这样做。”但被孩子这么一问，妈妈自己也开始疑惑，于是就去问外婆，外婆说：“说实话我也不知道，我妈妈以前都这样做，所以我也是有样学样。”这就奇怪了，于是妈妈就再去问曾外婆，曾外婆说：“喔，那是因为我那个年头的锅子都很小，除非我把肉的两端切掉，否则肉放不进锅子啊。”

我的上司就是如此，只一味地墨守陈规，不去了解背后的原因，所以就永远不知变通。要我写一份没有人看的报告，很显然不合理，但他仍然这么做，因为他只在形式上遵守规定，而不肯去了解写报告的理由。事实上，我后来发现，并不是每次出差都硬性规定要交报告。

有一些出差报告确实值得一写，像是 COMDEX 这样的信息商展盛会，公司派去参加的同仁应该一回来就把所见所闻写出来让大家分享。这种报告中应该充满对信息产业的敏锐观察和明智的见解，像某一种新趋势逐渐成形，竞争对手出了什么招数,造成了群众什么样的反应……每当 COMDEX 信息展结束后，出差报告就会塞满了电子邮件信箱，而这都是非常宝贵的信息。

但并不是所有的出差报告都有价值。也许您去了一趟印第安纳州的科克莫（Indiana，Kokomo），解决一个您在办公室里已经抓到的错虫，这种事情实在不值得写报告；如果您刚好去总公司，在经过大厅时顺便帮别人抓到了一只错虫，难道这样也要写报告吗？我觉得不必。如果您出差到顾客的公司时，顺便帮顾客把他们的错虫抓到了，要不要写个报告？至少在微软是不要的。大部分的情况是，如果您开车两小时到市郊，替顾客解决一个问题，那您大概不必写报告，但如果您坐 20 分钟的飞机到另一个城市，您很可能必须交出一份报告才能证明公司没有白费机票钱。这好象是和业务无关，只和报账有关。

我曾经大力主张，主管应该全力扫除那些阻碍目标而不必要的工作，像这种没有意义的报告就是其中一种。除非我认为这个会议重要到可以打断程序设计师的工作，否则我尽量不召开。同样地，除非很有必要，我也不会要求他们写报告。我宁愿他们去做开发工作、和其他的同仁互相切磋技术，我也不要他们写一份其实不需要的报告。我认为，除非有充分的理由，我不应该剥夺他们写程序的时间，打断他们进行到一半的构想。

我很少要求我的组员交报告，因为我认为大部分的报告都不值得中断他们正常的工作。即使我觉得需要报告，我也倾向口头的报告，一方面是双向沟通，我可以反问他们问题，一方面是不用花太多时间，5 分钟到半小时就够了，比起写报告快得多。

有些人一听说要写报告就头大心情坏，您过了三个小时再去看他，准会发现他在文字处理器前面发呆，只写出寥寥数语。对于某些人，写报告就好像是对着一大群人演讲一样令人浑身不自在。

写报告的另一个问题是，如果您不说清楚究竟要的是什么样的报告，交

上来的东西可能是拉拉杂杂的一大篇您根本没兴趣的文字。很多人觉得报告一定要篇幅够多，才看起来有报告的模样，或者是一定要文诌诌的，才会感觉到这份报告够水准，结果明明很简单的一句话可以表达清楚，却弄得啰嗦一大串，这类的报告不但是写的人累，看的人也觉得很累。这种报告就像是电视机的测试报告，架势十足，但是看不懂。

当我请属下写报告时，我都要求愈短愈好，而且表达清楚就行，不要什么正式的文体。尤其对于那些想到要写报告就心烦的人，这是个大好消息。我希望写报告的时间愈少愈好，给属下的负担愈轻愈好。

“写得愈短愈好？”他们会说：“没问题！”

当然，如果有人想在书面报告中多发挥一点个人观点，我也很欢迎。有些人比较喜欢书面报告，因为用写的比用说的容易理清思绪，尤其是当报告的目的是说服别人时，用文字比用语言有效。书面的报告可以帮助作者更审慎地思考他的建议，也可以帮助读者更清楚报告的内容。总之，我的目的是让属下用最少的时间和最少的打搅，让我获得我真正需要的信息。

写报告和开会一样，对开发工作都是干扰真的有必要，尽量不要打断属下做该做的事。

确定您所要求的报告真的值得属下暂停工作，花那么多时间去写。

好报告、坏报告、束之高阁的报告

有一种很值得花功夫写读的报告，就是项目的检讨报告，也就是开发小组在产品刚完成不久所写的分析报告。分析报告主要是针对“我们能从刚完成的项目中学到什么？”这类的主题进行探讨，并且检讨我们哪里做对了（就继续这样做），哪里做错了（下次如何避免再犯）。项目的检讨报告很重要，因为这会促使组员主动而认真地思考如何改进开发程序。

我很喜欢阅读项目的检讨报告，因为里面有许多宝贵的信息。但是非常可惜，我经常看到很好的见解，却没有实际的效用，因为少了下一步——如何做到。有一回，我读了同一项目的检讨报告发现，每一份的开头都是“我们应该在项目一开始时就在程序里加入除错用的指令”，接下来是“我们应该早点开始抓错虫，而不是等到项目快结束时才进行”。当然这是很好的意见，但是很不幸，这两个观点一再出现在每一份检讨报告中，表示这个教训并没有被大家放在心上。

如果您正准备写一份项目检讨报告，我诚恳地建议您，除了指出您发现的问题，提出您的看法和意见，还要建议如何防范这个问题，把详细的步骤一并写出来，这样才能让下一次的项目进步，不重蹈覆辙。这样的报告就很有建设性。我想前面所提的小组一定是把写出来的项目检讨报告束之高阁，再也不去看它，因此不论报告写得多详细，同样的毛病仍然一犯再犯。

有时候检讨报告确实有包括改进事项的实施步骤，但是不够明确具体，因而流于不切实际。举例来说，有几份检讨报告是这样写的：

问题：Mandelbrot 软件包的 Beta 版试用者觉得他们的测试报告好像都没有人注意，因为同样的错虫每一版都出现。这主要是因为我们没有建立一套系统的方法去追踪外部的 Beta 测试报告。所以，我们将来应该更加小心地追踪外部的 Beta 测试报告，并加强后续处理。

大部分的情况下，检讨报告这么说过就算了。偶尔会有人比较认真，提

出更明确的做法：

解决方案：企划部门应该草拟较佳的办法，追踪外部的 Beta 测试报告。

这样还是太笼统了，很难得会有小组真正做出切合实际的检讨报告：

解决方案：由于对 Beta 测试报告的疏忽，不仅影响了 Mandelbrot 项目，也影响了 Biorhythm 和 Morph 两个项目。Hubie 经理已经同意重新考虑三个追踪错虫的系统——BugControl、Programmer ’sDatabase 和 FixIt！，我们将在三者中择一使用，以便追踪 Mandelbrot 项目的测试报告，我们还要把错虫和清除错虫的行动记录下来。

您觉得最能够改善软件开发程序的是那一份报告，是描述问题的报告，提出意见的报告，或是拟出详细改善步骤的报告呢？

无庸置疑，第三份报告才是最有效用的，因为它提出了清楚的解决方案、详细的执行步骤、由谁负责、什么时候该完成、应用在那几个项目。这份报告同时还建议了追踪错虫成效的检讨方式。第二份报告虽然说明了由谁主导，企划部应该拟出追踪办法，但没有说明由谁负责，而且也没有说应该什么时候完成什么样的系统、用在哪些项目中、是否在实施之后进行成效检查？少了这些，解决方案就没有说服力。

除此之外，检讨报告应该包括本项目在开发过程中，值得记录的心得与收获。报告中可以这么说：我们在程序中加入维护叙述（assertion）和除错指令之后，竟然发现程序中到处都是错虫，连那些应该是完全正确的程序中也有错虫。或者是：刚开始使用除错工具一步步查看程序的进行时，觉得很无聊，但是习惯了之后就觉得很有意义，可以抓出很多错虫，而且事实上也不影响进度。检讨报告中也可以有这样的感想：我觉得有一套详尽的项目目标对工作帮助很大，让我们对真正该做的事保持专注。这些都是非常不错的内容，我只是举例说明，当然报告中是不止这些的。

除了将观察或体会到的心得写出来，检讨报告中还要进一步指出将来应该如何利用这些经验。光是发现那些方法有效是不够的，大家更应该学习如何充分发挥这些方法的最大效用。如果有部分组员习惯程序刚写完就立刻用除错工具检查，并且发现这个方法对预防错虫很有用，就应该在报告中详细说明做法和相关注意事项，让所有的小组成员都能够学习这种减少错虫的技巧。

最后，报告中不妨加入作者认为谁应该分享这些信息，让本报告得以发挥最大的作用，比方说：我们基于某个理由，将于某日期提供本报告给某某经理。将原因、时间、人交待清楚。

读和写这样的报告是需要相当的心思和时间的，当然多少会占用软件开发的时间，但是，其中的教育效果非常卓著，所以非常值得投资。当然，前提是主管很重视项目检讨报告，而且确实吸取其中的经验，应用到将来的开发工作上。如果检讨报告只进入档案柜，却没有进入每一个人的心中，写得再好的报告也不过是废纸。

顺便一提的是，最好不要等到项目完成了再来回想有什么值得写（搞不好已经忘了一大半），您应该在每一次遇到问题、并发现好方法可以解决时，就顺手记下这项发现，您何必等项目结束时再来学习？经验的累积是随时的。

利用项目检查报告来改进软件开发的工作程序。为了使报告发生作用，报告中必须确实描述我们这次解决问题的每一个详细步骤，以及将来应该如何运用这项新发现。

避免召开会议

在第 1 章我曾提过，如果您已经得知项目目前进行的状况，就不必再召开进度报告会议了。我极力主张应该避免的周期性会议不只这些，进度报告会议只是其中之一。“ 周期性会议 ”（recurrentmeeting）是定期召开的会议：您一早进到办公室，想到今天是星期二，可别忘了每周二下午 3 点的那个某某汇报。

我很少召开会议，因为那会打断组员的工作，影响工作的顺利进行。我也很不喜欢周期性的会议，因为这种会议大都没有明确的动机。您去开会是因为真有什么事需要用开会解决，还是因为“ 时间到 ”？有些人认为每周一次的进度报告会议是不能少的，但是我已经好几年没有举行，因为我认为开会本身不重要，重要的是您需要项目现况的信息，如果这些信息可以用更好、更有效率的方式取得，何必非开会不可？我在第 1 章已经说过，我的小组用一个简短的 e-mail：“ 我已完成…… ”，对我而言，这样就足够了。

当然有时候开会是很不错的方法,在什么情况下,开会是利多于弊的呢？以下是我的看法：

◆当某个人必须把信息传达给很多人时。

◆信息需要双向或多向沟通时，人们必须立即反问发言人或与会人才能了解信息。

◆必须要亲眼目睹或亲身经历，信息才能传达给接受者，例如产品示范等。

◆有些事情用 e-mail 很难表示清楚，而要大家一起讨论时，例如组织再造等。

符合上列条件之一的会议，就是值得召开的会议，但是如果有更有效率的方式达到同样的目的，还是避免开会为宜。在古时候，发布信息最有效的方式是大家集合在一起，由一人宣读羊皮纸上的皇帝诏书，但科技发达的今日，我们有复印机、e-mail、电子布告栏，有很多方法让信息的传达更有效率，又不会打断手边的工作。当然，如果您有很重要的事情，面对面开会的方式，会比用 e-mail 的方式更让与会者感受到这项信息的重要性，而且也能保证每个人都收到信息；况且，如果您是一位很好的演说家，会议还可以达到激励的效果。无论如何，在您决定召开一次会议之前，请花一分钟问自己以下的问题：

“ 这个会议的结果是否非常重要，即使是中断这么多人的工作都值得？ ”

“ 还有没有比较不影响组员工作的方法，可以让我达到同样的目的？ ”

团队精神

我曾听过某些主管说，他们认为每周一次的进度报告很重要，因为可以借机让小组面对面齐聚一堂，有助于团队精神的培养；我甚至见过有些主管开会的主要原因只是把大家聚集在一起。这些目的是对的，但是我认为开会是最差的方法。以我的经验，这种会议事实上都在报告谁手上的什么工作

还没做完，这种会议对团队精神有害无益。

如果您的组员不喜欢聚在一起头脑风暴，您就得为他们制造一些共同活动的机会。如果是为了培养团队精神，来个全体聚餐，或是安排大家喜爱的活动，会比报告进度的会议更有效果。

当您考虑是否召开产品设计会议，那么思考过前面的两个问题之后，您就会发现召开产品设计会议是很值得的，即使不得不打断组员的开发工作。产品设计会议确实能改善产品，大家能借此机会充分讨论各种意见，对各种设计上的论点做最适当的取舍权衡，对产品产生更清楚的共识；虽然暂时让大家离开手边的工作，但对日后的项目进展有非常大的帮助。用 e-mail 来讨论的话，效果就不会这么好，e-mail 并不适合用来做头脑风暴。

但是，如果把产品设计会议定成每星期二下午 3 点，定期召开呢？我可不认为这是个好主意。除非您已经定好了一系列的议程主题：本周讨论内存管理，下周讨论档案处理，再下周讨论使用者界面……即使如此，我仍然不觉得定期的产品会议有什么意义，它很可能是这样的开场白：“ 这个星期有没有新的设计问题需要讨论？ ”假设有这样的问题需要讨论，我想也不该等到了定期会议时才说出来，应该在发现设计问题时立即提出，如果严重到需要大家讨论才能解决，您可以召开临时性的特别会议。您应该把开会时间保留给真正需要的时机，而不是先定期开会再看有没有问题要讨论。

请注意定期会议的价值，确定它值得每个人放下手上的工作。

适当的开会时间

如果您实在非开会不可，请尽量不要中断做了一半的工作。不要把时间定在上午 10 点或下午 3 点，这样会把上午或下午的时间切割得太零碎，最好排在一清早、快下班前，或是下午工作时间的开始。换句话说，把开会间排在时段的最前面或最后面，不要把这个时段切割，就可以尽量减少工作的中断。

另一个方法是把所有的每周会议集中在同一个时段，例如星期一上午或星期五下午，因为这是工作效率最差的时段，因此不妨把所有的会议集中，让其他更有生产力的时段保留给更重要的工作。

让会议有效果

即使我很讨厌召开会议，也不喜欢参加会议，我还是得承认有一些会议是必要的。既然是必要的，我们必须尽量让会议有效果，去其弊取其利。我们如何从会议中获得最大的效益，又将它的缺点降到最低呢？

会议的目的在于获得结论，但必须耗费时间成本。有时候因为开会的目的对与会者而言不够明确，以致无法得到共识，只是白白浪费时间，所以，为了让会议有效果，您必须一开始就明确定出究竟要做到什么，并拟出一串计划。

一旦您确定某一会议是必须举行的，在发出开会通知前，请先问自己这样的问题：

这次会议的目的是什么？我希望在这次会议中获得什么结果？我该怎么做，才能确保我的目的能够达到？

如果您能清楚回答以上的问题，那您就比较能掌握开会的效率，不会让会议变成漫无目的的讨论。

记得我在第 3 章中房屋搬迁的例子吧，这也是同样的道理。房屋搬迁的

主管没有事先考虑过最恰当路径，结果房屋搬进时困扰不断。开会也一样，您必须事先想清楚自己的目的是什么。

当您问自己：“我希望在这次会议中获得什么结果？”您等于是强迫自己向前看，瞧瞧前方有没有障碍物需要我们绕道而行。如果您对未来的目标有清楚的认识，而且知道要如何达成这个目标，您就会成竹在胸，该请谁来开会、讨论什么议题、如何导出结论。很多会议之所以无疾而终，就是因为该做决定的人根本没有出席，或是该提供重要讯息的人未受邀列席。

当然，无论再怎么努力，会议也有失败的可能：您可能会开到一半才发现有一项重要信息无法取得，因此无法做出决策。这时候只好用一句话草草结束：“乔治，看看你能否在两周内完成 Anagram 的功能，其他的事我们下次再讨论。”这样的做法无疑是浪费大家的时间，每个人聚在一起，却讨论不出个所以然来。如果您的目的是做出决策，那就起码有个决策才能散会。您可以做个有条件的决策，总比用以下这句话来结束会议好得多：“假定乔治对 Anagram 的功能所预估的时间完全正确，大家是否都同意为了这项功能而推迟我们 Wordsmasher 的推出日期？”

如果您这样作出总结，到头来就会发现，没有人真的觉得 Anagram 的功能会重要到我们可以为它延后推出日期，要不然就是认为 Anagram 的功能太重要了，即使因此延后推出日期也无所谓。最好是能得到某种程度的结论：“假设因为 Anagram 所造成的延误在两周以内，我们就这么办吧！”

这样的结论虽然不如“是”与“不是”那般铿锵有力，但是总比把决策延到下次不知何时召开的会议好得多。如果您开会的目的是做出决策，就一定要做到，如果您的目的是别的，也务必让您的会议有效果。

召开任何会议之前，请确定本次会议的目的是什么，达成这个目的的条件是什么，然后，务必达到开会的目的。

会议价值的衡量标准

我一再大力强调每一个会议都必须得到某种程度的结论。如果开了会却没有结论，就是浪费时间的会议。

我们再来回头看看进度报告的会议。这个会议是为了导出某种结论吗？不，它的目的是传递信息。那么产品设计会议呢？是的，它是为了决定如何设计产品，也许实际上是一场头脑风暴，但是它的目的是决策性的，开会的结果就是产品设计，决议出大家都同意的设计方式与内容。

至于高级主管所主持的项目审查会议呢？它会在结束前导出任何决策吗？这就不一定了。有两种报告的形态，会有不同的效果。第一种是项目经理分别报告去年以来，本项目进行的过程，目前进程计划进行的概况，表达颇有信心能够如期完成；另一种则是省略过去的故事，把重点放在目前项目进行的决策问题，分析方向的选择，为什么决定这个解决方案而舍弃另一个，以及本项目如何配合公司的长期目标，需要高级主管们什么样的支持。

第一种类型只是不加思索地倒出信息；第二种类型的最终目的是争取高级主管的支持，报告与分析项目的方向是铺路的工作。您认为自己的公司比较适合哪一种类型呢？是报告性的，还是决策性的？

有一些集会，像是公司的年会或是鼓舞士气的集会等，开会的目的本来就不是要得到什么结论，当然也是有其意义的，但是更重要的是，这种会不是每周或每月定期举行。

避免会后工作

开会的缺点之一，就是伴随而来的后续工作。有些时候实在无法避免，因为没有这些后续工作的话，工作无法继续进行，像是前面的例子，乔治必须估出 Anagram 所需的时间。但是有些后续工作只是在浪费时间，加重组员的工作负担。记得我曾提到过的一位主管，要求组员在会后把自己的发言再发个 e-mail 给他。后续工作成了会议之外的更大伤害。

当您在总结一次会议，摘要重述本会之决议，并且分派工作给适当组员时，请注意每一件后续工作是否确有必要。我认识几位经理，每次开会时都要让每个人都有新工作才甘心，总喜欢在结束会议之前来个大点名，咦？突然发现某位组员成了漏网之鱼，他停顿一分钟，抓抓头，然后终于想出一项工作：“乔治，你何不去做……这件事情？”

如果您发现自己有这种倾向，请赶紧纠正过来，改用另一种方式：当您在会中对出席者在心里默点名时，请评估指派给他的工作是否值得他花时间去做。典型的对话是像这样的：

“下一位是乔治，你应该评估 Anagram 功能需要的时间，老实说，你有没有较好的方法，让整个进程不要因 Anagram 而延误两个礼拜？”

乔治答道：“事实上，我在刚才的 20 分钟里已经想过这个问题，我认为 Anagram 功能应该要完整才有效用，合理的开发时间应该要 3 周。”

“好吧，Anagram 并没有重要到我们非为它延迟推出日期不可，既然如此，我们把它放在下一版的目标如何？大家都同意吗？很好。乔治，我们就这样决定你的事了。下一位，黎佩佳，换你，你要做的是……。”

只要把握住以上的原则，您会惊讶地发现，很多事情在讨论中已经决定了，其实并没有那么多非做不可的后续工作。

试着排除不必要的后续工作。

逃避工作？

您在开会时一个一个点名重新评估分派给他们的工作，企图减轻组员不必要的工作负担，您的苦心会不会反而在无形中造成组员懒惰的机会呢？乔治真的认为他需要多一个星期，或者只不过是把时间延长好让自己做得轻松些，甚至延到下次再做？

任何组织中都会有人为了减轻自己的工作负担而撒谎，并且毫无愧疚之意，这是职业人生的一部分。但我相信绝大多数的员工都是诚实而善良的，不愿意来这套尔虞我诈，您只要揪出那一小撮人，多注意他们就行了。

除此之外，我相信组员如果认为这个功能很重要，绝不会因为来不及完成就轻言放弃，如果 Anagram 真的很重要，不但乔治不会放弃自己做重要事情的机会，其他的组员也不可能同意把这个功能留到下一版再做。

铲除障碍物

如果您希望您的组员保持高昂的工作情绪，就让他们专心致力在产品的开发，不要被一大堆会议、报告及莫名其妙的填表或行政程序等等之类没有生产效率的东西缠身。不过，很不幸的是，公司总是为了芝麻绿豆的小事而兴师动众地开会，而且还像膝盖反射动作一般要求立刻交份报告：“我现在很忙，待会儿尽快把报告交给我”。

您也许觉得这只不过是小小的障碍物，算不上什么，但是您要知道，在赛车场上只要遇到了一点点小颠簸，就会车毁人亡。软件开发就像高速疾驶

的赛车，在它加速的过程中，砰！遇到了要开会的路障，咚！掉进了写开会报告的坑洞，咻！天上飞来再写一份工作绩效报告的落石，霹雳啪啦！接不完的电话……行政程序……无聊的表格填写……而那些要求定期开会、自己不想做笔记和演示文稿就叫属下起草的主管，就是始作俑者。

您虽然不能完全控制所有的路况，但是您一定能铲除大部分的障碍，所以，拿出您的利斧，用力劈开前面的险阻，为您的软件开发工作开路吧！

重点提示

尽量不要让组员写没有用处的报告，即使非写不可，也要尽量减少对开发工作的干扰，务必让每一份报告的价值超过它的成本。

项目检查报告是很有价值的报告，您应该善用。但是检查报告必须清楚陈述解决问题或提高工作效率的方法，而且其中的建议能够确实被执行，否则用处十分有限。

召开会议之前，请确定会议的结果够重要，值得为此打断程序开发的工作，占用组员的时间。特别留意定期召开的例行会议，通常定期的开会最后只不过是因循的习惯而已，并不值得参加。

如果您准备召开一个会议，请将时间安排在一个时段的最前面或最后面，尽量减少工作的中断与时间的切割。

每次开会之前，务必确定您开这个会的目的是什么，而在开会时一定要达到某种程度的结论，即使是有条件的决策也比完全没有要好。

第 5 章　进度狂

在第 2 章里我强调过，一发现错虫就得立刻清除，但我必须承认当年我们在开发 Excel 项目时，并没有遵照这个原则，相反地，我们被迫忽略错虫，直到进程表上的工作都做完了，再一起进行除错，原因是如果一发现错虫就立即去除错，手上的工作进度就必定落后。在当时的微软，没有人在乎您是否进度超前，但是任何进度落后都是不允许的。要是您的错虫不断增加，不算严重问题，但是只要您的工作没有在排定的时间内完成，您就罪孽深重了。“进度”取代了项目目标和软件质量，变成了首要之务，每一个人都在疯狂地赶进度。

微软在当时就是用这种方法开发应用软件，在理论上似乎合理但事实上问题百出的进度控制法，虽然可以使产品准时出货，但是开发人员承受非常大的心理压力，程序设计师为求进度不得不放弃品质。在当时并没有发现明显的问题，直到几年后，微软才尝到过度重视进度的可怕后果。

后来问题变得很严重了，微软才不得不放弃这种做法，改用比较人性化的进程控制管理方式。无论如何，这是微软代价极高的经验教训，我将在本章中叙述微软这段教训的历程，好让其他的开发部门不要重蹈覆辙，我也将告诉您，在微软中被证实真正有效的进程控制管理方式。

很久很久以前，有一个项目……

那是 1986 年的事了，我加入了微软的工作行列，主要的原因是我想开发最好的 Macintosh 应用软件。我被派去 Excel 的工作小组，虽然是最后一套进军 Macintosh 市场的应用软件，从任何一方面来说，能参加这个项目都是令我兴奋雀跃的事：Excel 是 Macintosh 上很重要的软件，深受大众喜爱，能够躬逢其盛实在是我个人生涯中的大好机会，我不仅能获得 Macintosh 上的软件开发经验，更何况这项明星产品将包含我的智能结晶呢！

然而事实没有想像中的美好。刚开始时，Excel 的工作的确很令人兴奋，但是几个月后就变得无聊又痛苦，令人厌倦。怎么回事？Excel 应该是很多程序设计师梦寐以求的项目，没有道理让我觉得恼恨，不只是我和 Excel 的同事，别的项目的程序设计师也是一样的心情。原来问题不在员工，也不是工作环境（微软是我在这一行干了 10 年所经历过最好的的工作环境），问题是在微软所采用的进度控制管理方式。

在加入微软之前，我所带领的工作小组都非常乐在工作，看着项目一步步地迈向目标，产品一天天地成形，大伙儿心中都有一股兴奋喜悦的心情。但是 Excel 项目就完全没有这种快乐，虽然我们每一天都在努力改善产品，而且工作也依照计划稳定地推进，但我们随时收到一种威胁的信息——我们可能要落后了，我落后了、他落后了、每个人都在落后、整个项目都要落后了！焦点总是集中在进度，而不是更重要的软件质量。

在第 1 章中我提过了每周的进度检讨会议，这就好像定期在大家脸上掴耳光，看看你这个星期欠了多少事没做完。每周的进度检讨会议加上报告，这就是微软当时用来控制进度的主要手段。除了这些，开发人员还得每周和测试与文件人员共同检讨原因和落后的情况。可想而知，程序一旦落后，文件编辑小组和测试人员只好暂时没事做，所有的目光和谈话，都集中在你的程序进度落后了。

可怕的事情不只是进度检讨的会议和报告呢。每星期 Excel 的经理们会用交上来的进度报告来更新项目的总进度表，然后再分发给每位组员。这种做法自然是无可厚非的，但是总表的封面就让您一眼看见自己本周落后了多少，整个项目会因此而落后多少，即使您是因为处理其他多重要工作而耽误下来，也没有人同情您，落后就是落后。高级主管看到这份报告，当然会查问原因，压力就来了。落后！落后！落后！那种日子真不是人过的。

心痛之余，您翻到后页看看还有多少未完成的工作。唉，真是糟透了！上周是几百项，现在还是几百项。我们在这里拼了老命做事，结果似乎毫无进展。这就像我听过的一个笑话：“ 如果你一次咬一小口……要多久才能啃完一只大象？ ”这张进度总表就是一只大象，我们一辈子也无法吃完。

进度实在是太被过份强调了，以致于无论我们做多少了不起的事情，都没有半点儿成就感。我们被那种落后的威胁淹没了，再怎么努力，也看不到工作有任何进展，这不是工作本身的问题，实在是那种绝望的无力感所致。

我个人在做 Excel 项目以前，从没想到进度表能够把士气伤害到这样的程度。原本是我梦寐以求的工作，现在觉得像是一场没完没了的恶梦。虽然尽了全力工作，我们仍然不断地落后，而且从来没办法把进度补回来。事实上，进度表定得太不实际了，它是依照以下几个假设前提编定的：

◆为期两年的项目中所有该做的事情都列在总进度表中，没有任何遗漏。

◆每人每周的实际工作时间是 40 个小时。

◆对于每件工作的时间估计完全准确。

◆所有的程序第一次写出来就是最完美的状态，没有错虫，不需修改。

如果进度表是这样定出来的，那么即使是最优秀的软件开发部门、每人每周工作 80 小时，也无法保持进度。计划中的工作事项是很难事先列齐全的，可能会开发到一半发现当初漏列了某一项很重要的工作；每人每周虽然上班 40 小时（以上），但不会时时刻刻都在写程序；一个程序需要的时间很难准确估计，尤其是编日程表时可能不很确定这个程序的功能，再高级的程序设计师都有手不顺的时候；程序不需要修改？真是天方夜谭了，不但可能有错虫，功能需要增加，也可能事后发现还可以更快、更精简。别说是这些了，还有随时冒出来的会议、报告、会谈、e-mail 以及其他杂七杂八的事。更离谱的是，这份日程表竟把法定假日当成工作日，也忘了考虑员工每年两周的休假。这种时间表，怎能不落后！

不要利用进程表来驱使项目的进行，这对小组的士气伤害太大了。

照着规矩办事

我必须澄清一点，Excel 的项目经理们绝不是故意制造出这样令人沮丧的情绪，他们只不过是照着规定的日程控制进度。后来他们也对每周 40 小时的工作时间，在认定上做了点调整，把会议、报告等也算是在上班。我相信微软绝不是有心在程序设计师身上压榨出 80 小时的工作量，但事实就是如此，这也就是微软的员工以工作勤奋出名的主要缘故。

我绝对相信日程表的编定是为了准确追踪项目的进度。毕竟，要推算出完成日期，没有比“ 加总所有的工作时间 ”更合理的方法了。当然，没有人相信日程表可以把所有的事情都算到、算准，但是就有人（尤其是高层管理者）要把“ 推估的完成日期 ”当成“ 实际非完成不可的日期 ”。好在现在的微软人几乎都已十分了解理想和现实的差距，后来的工作就逐渐顺利了，这些我们稍后会谈到。

适当的压力

您大概听人说过：“当经理的人要想属下拼命工作，就得把日程排得紧迫逼人。”我相信绝大部分的主管多少都相信这个论点，我自己也不例外。但问题是，要多紧才是适当的呢？如果日程表能激发属下的潜能又不致摧残他们，使项目以最合理的高速进行，那就很不错；但如果日程是如登天一般不可能的任务，那就只会打击组员的士气。

日程表必须有点紧，让程序设计师有一种急迫感，逼使他们把注意力集中在最重要的事情上。假设您明天就要休假去，一定会急着把手上的工作告一段落，因而加快了做事的步调，不愿浪费时间在茶水间闲聊，也没有时间搞那些 e-mail，不必要的会就别开了，您一定把精神放在最重要的事情。

最有压迫感的时候，就是期限快到的时候，我想任何项目都是如此。在微软，主管大概会发出类似这样的 e-mail：

我们的期限快到了，请大家好好运用所剩无几的宝贵时间。现在每个人的时间都很珍贵，我们必须全力朝向目标冲刺。要开会或讨论的话，请三思而后行，为了向人求教而打断对方的工作之前，请想一想能不能自行找到答案。如果有额外的问题发生了，请别期望别人来解决，大家都和你一样忙、一样紧张。请不要私下保留一份待完成的工作清单，以为自己“终究”能够做完这些工作，你没有“终究”的机会，所以请把所有未完成的工作让经理知道，至少我们可以决定它们的优先级。如果自己工作都完成了，可别以为你可以闲着没事干，除非是整个小组都完成了工作，你才算是大功告成。我会提醒你们，但我知道你们都够聪明够勤奋，你们都不是小孩子，我相信你们不会浪费自己的时间。

每当我看到这样的 e-mail，都忍不住纳闷：难道是项目期限快到了才开始紧张，工作小组不是平常就该保持这样的工作态度吗？

我这话听起来怪吓人的，您一定觉得可怕：我是不是那种把员工榨干压扁的恶魔主管？当然不是，我一再倡导的观念是：不要把规则当法律，要多想想如何聪明地工作，不要浪费时间在没有价值的工作上，也不要浪费别人的时间，要用积极的态度推动项目；这个观念在本质上与这封 e-mail 并没有太大的差异，只不过，在期限将至的时刻看到主管发的这封 e-mail，感觉就有点刺痛了。

如果您觉得时间非常紧迫，我想您在开会总结时一定不会说：“乔治，关于这个问题，我们……以后再来研究好了。”您和组员们一定无法容忍事情拖拖拉拉，要不就干脆删掉这项工作，要不就立刻把它完成。

如果组员没有那种急迫感，您想他们做事会全力冲刺吗？如果时间多得很，多到大家早上闲闲散散来上班，翘起二郎腿，一边看窗外的白云，一边慢慢想项目，这样的沉思当然不是白浪费，也许会有极具突破性、非常震撼的创意迸出来，但是这样不符合软件开发的快节奏，项目就不那么刺激、兴奋了。好比是乡间小路的漫步和坐空中飞车的对比，若有充裕的时间钻研和交流大家的创意当然是不错，但是紧张、有压力的头脑风暴，也能激发很好的点子。我相信要让大家的创意保持活跃，适当的压力是必须的，而时间的急迫性就是一种激发能量的动力。

让日程表维持适度的紧迫，但又是可以做到的，好让组员振奋、不松懈，

专心致力于项目的推进。

紧逼盯人的最后防线：质量

时间的急迫性必须适度，如果日程表定得太紧，紧到不太可能完成，组员就会为了赶进度而做出愚蠢的事情。我曾经共事过的程序设计师中，有人就因为程序实在太多、时间实在太少，程序就只好不测了——写完、编译过关、第一次执行没有发生错误，这样就算是完成了，然后继续赶另一个程序。他自己很清楚这样的工作毫无质量可言，但是迫于时间的压力，他别无选择，只能祈祷测试小组能够找到所有的错虫。

身为主管的您，一定要时时注意组员有没有为了赶时间而做出蠢事。您必须随时提醒他们，日程固然重要，但是为此而牺牲品质是划不来的，与其勉强交出一个设计不良、没有测试完整、拼拼凑凑的程序，不如认真把它写好，即使需要多花点时间。超过期限对项目不好，质量不合格的程序却造成产品永远的困扰——除非将来有人不计代价地逐行检查、修正这些草率的程序。

绝对不要草率定出不可能的期限，导致组员为了赶进度而损害产品的质量。

草率的期限

在我的经验中，大部分的期限都是上级主管随意指定的，也许他会对您说：“这件工作应该在某月某日完成吧？”如果您同意了这个期限，到时候就得做出来。事实上，您和主管虽然彼此同意了一个日期，但这个期限可能是草率定出的，绝不表示它比质量重要。想想看，如果您晚了一个月交差，会对公司的长期发展有什么影响吗？六个月后还有人会在乎这件事吗？但是如果您交出的是错误百出、表现恶劣的程序，您想那一种对产品的伤害比较大呢？是稍微迟延，或是外界无情的抨击？

除非您的程序对产品有决定性的影响，而且这个程序的完成日期不容更改，否则您大可不必为了遵守期限而牺牲一切（话又说回来，如果您的程序如此重要，那么万一有错虫岂不更糟）。软件产品的推出就像航天飞机发射一般不容发生错误，如果航天飞机有一个零件尚未就绪，您不会为了准时发射而牺牲造价奇昂的设备和航天员的生命吧，您一定宁愿延期一下的。

当然，我这样的论调也许会让您觉得要想把工作做好，似乎需要很长的时间才行，其实不然，就我的经验来说，如果方法对了，花的时间应该会比较少。在项目刚开始的时候有一些基础工作不能忽略：设定明确的目标和优先级，预先思考设计上和开发上的问题并设法预防，建立测试计划，设定质量水准规范，这些工作在日后可以节省很多时间。您想一想，在项目进行中一发现错虫就立即除错，和项目快结束时才一并除错，哪一种用的时间比较短？答案很简单。当其他的小组每周工作 80 小时，仍然为大量的错虫焦头烂额时，您的小组早已清完了错虫，而在宝贵的最后数周内，再做彻底的测试，把可能存在的最后一个错虫揪出来。

稳操胜算的日程表

稳操胜算的日程表能够兼顾公司和员工的利益，正如我所说过的，必须有点紧迫，又不会太紧迫，要让产品及时完成，也要让组员觉得时间刚好够完成最棒的产品。另一个日程表重要的观念是，好的日程表必须能够肯定小组的工作进展，让大家有“赢的感觉”。

我在叙述 Excel 的开发历程时，以大象作比喻的日程表，一周又一周的辛勤工作，然而上面的待办事项始终没有减少。将近两年的时间，我每天一

大早踏进办公室，第一件事情就是在这张超级大表中找出最重要的几件事情来做。对于完成日期在两年之后的项目，您想我会有时间紧迫的感觉吗？老实说，没有，直到期限将至的几个月前，我才开始觉得火烧眉毛。

也许您听过这么一句谚语：“没有期限的目标只不过是梦想而已。”期限会促使您赶紧去做最重要的事：开发产品，期限会迫使您尽量丢开无聊的行政程序，有期限，您才会设法用最有效率的办法做事。当然我们 Excel 项目也有完成的期限，但是两年似乎还太遥远，因而没有激励的作用，我们当时只觉得：“别急，反正总有一天我们会完成 Excel。”

我所参与过的每一个令人兴奋的项目，期限都没有超过两年的，无一例外。并不是这些项目的规模比较小，需要的时间较短，事实上这与项目的规模并无关连，我们是把整个项目分割成好几个较小的项目，每个小项目的完成期限大约是两个月。每一个小项目的期限在两个月后，必须完成一项短期目标，这样既能创造适当的紧迫气氛，也能让小组有完成目标的成就感。换言之，我们并不是整个项目结束时才完成产品，而是每两个月就“出货”一次。

还好，在 Excel 之后，微软的项目大部分都是采用这种阶段式的日程控制法，而把每一个小项目称之为一个里程碑。若是单纯把日程表切割成每两个月一块，其实不够，除了每隔两个月要来一次推出虚拟产品之外，并没有改变什么，最后的期限仍在两年以后。并不是要多加几个期限来制造压力，重点是每一个小项目就是一个目标的完成，让大家感觉到工作的推进，提升整个团队的成就感和工作热情。下面我们来谈谈项目应该如何切割成小项目，才会有效果。

“把工作项目中最优先的全都做完”也许是很重要的，但是工作清单上最优先的未必能构成一个完整的小项目，分属各大类中的最优先项目，可能只是一堆彼此不怎么相干的工作。

然而，“完成绘图子系统”就是一个完整的小项目了，所有的工作项目都属于同一主题，各工作项目之间需要紧密协调。您不妨用一个独立的日程表和工作清单来提醒组员该做的事情，但是总体来说仍然是属于大项目的开发路线。小项目并不是分头完成 352 件毫不相干的工作，而是完整地完成一个子系统，而是可以被当成即将出货的产品。

假设您打算在家里请客，所以去超级市场大采购，您大概不会只买请客用的东西，只要是需要的都不妨顺便买齐，您也不会到了超市才开始想我要买什么，或是我忘了买什么，或是我一定还有什么地方没想到。您一定是把请客采买当作目标，先拟好一张购物清单，买好了就打个勾勾，而且同类的东西一起买。很简单的常识，因为这样做最快嘛。

我前面引述过的那件 e-mail 中，那位主管强调每件事都得设想周全，每一件事都要好好收尾，不要因此而“为山九仞、功亏一篑”。组员在审视自己的工作清单时，重心会放在自己，我还应该做什么；审视小项目时，重心则在整个软件，去找出所有相关的工作，还有什么是该完成的。

任何没有主题的阶段性工作就得依赖工作清单来推动，因为没有主题，您当然没法子找出与这主题相关的工作，这样的话，整体工作的进行就比较零零散散；如果是分成各有主题的小项目，整体工作的进行就会比较系统化。

把长期的大项目，分成几个完整而独立的小项目，各小项目必须有一个

主题。

哇哟！真棒！

同样是从大项目切割出来的小项目，想想以下哪种较令人兴奋？是完成某一主题的全部工作、还是完成许多不相关的工作？我想用装璜房子的比喻来说明其中的差异。您有一间新房子正在装璜，工作进行到了一半时您去看看，下面那一种情况是您会感到很明显的惊喜？情况一是某间卧房的墙壁粉刷好了、起居室的灯饰安装好了、客厅的地毯铺好了，情况二是客厅全部完成，其他部分则是都还没开始，我想答案是后者。同样的，当您完成了一个小项目，最好能带来一种“哇哟！”的惊喜气氛，让每一个人——包括整个开发小组、Beta 测试参加者、高层主管——都明显感觉到项目又向前推进了一大步。相反地，如果每一个小项目都是这里改点东西、那里加点东西，整体看起来似乎没有明显的进步，就没有令人振奋的效果。

当然，唯一的难题是如何将大项目切割成小项目，才不会在各小项目之间造成掣肘或冲突，不过以我的经验来说，这个难题是可以解决的。

想办法创造一些“哇哟！真棒！”的欢呼吧，因为这是激发小组创意的动力之一，对于士气的鼓舞是非常有效的。

加强“哇哟！”的效果

如果小项目的目标是“把工作项目中最优先的全都做完”，也就是依照总工作表的优先级划分小项目，那么，小项目只不过是一大堆互不相干的工作混杂在一起。如果小项目的时间紧迫，项目经理可以悄悄调整总工作表中的优先级，这就掩饰了这个问题，也许可以让项目经理觉得舒服些，但却埋下了隐忧。

如果小项目的目标是“完成所有关于画面的程序，好让文件小组确定手册中的执行画面”，这就是有明确主题的目标，而且根据这个目标，很容易可以判定哪些工作该做，哪些是不该做的。对于任何一件工作项目，任何人（哪怕他是笨蛋）都可以立即判断它是否会影响使用者界面，应该现在完成，或是延后到下一个小项目。

您可以采用各种不同的方式来将项目切割成小项目，只要记得您的原则是：小项目应该有一个明显的主题，能够产生令人惊喜、振奋士气的效果，这样就对了。当我们在开发 Macintosh 的 C/C++ 交叉发展系统（MacintoshC/C++crossdevelopmentsystem）时，我们的各个小项目的目标如下：

把编译器中所有英特尔 80x86 的专属指令区隔开来，以使编译器能够支持其他类型的处理器。

◆在编译器中加入 MC680x0 的程序代码产生器。

◆撰写 MC680x0 的汇编程序码表列功能。

◆撰写 MC680x0 的对象文件（objectfile）产生器与相关的支持工具。

◆连结单段（single-segment）应用程序，让它能够执行。

◆连结多段（multi-segment）应用程序，让它能够执行。

◆在编译器中加入程序代码优化的功能。

◆……

我之所以定出以上这几项小项目的目标，是因为它们都可以在两个月内完成，且容易理解，同时，除了第一个目标“隔离 80X86 专属指令’之外，其他各小项目都有明显可见、令人兴奋的结果。当我们第一次看到编译器能

够正确产生可执行码，以及第一次看到屏幕上印出正确的汇编程序码时，全组都跳起来疯狂欢呼。当我们第一个测试程序完成正确的链接而执行成功时，我们确定这个产品已经合格了，而优化功能确实可以发挥作用时，我们相信自己的作品已经能与市场上的其他两个编译器竞争了，我们的兴奋真是笔墨难以形容！

不要忘记细节

前面列出的阶段性目标是为了说明上的简洁扼要，事实上并不那么简短。“连结单段的应用程序，让它能够执行”是一个概念，并没有表达出足够的细节，真正的阶段性目标还要更明确：

我们应该能把任何一条 Macintosh 程序，拷贝到一个目录之下，更名为 test.c，然后键入 make 就能编译器程序，并正确地链接函数库，而我们在直接联机到 Macintosh 的开发用 PC 上，对此执行文件的图像双击左鼠标键，就能执行这个 Macintosh 程序，执行结果必须完全正确。

从这段详细的描述中，您就可以了解，我们必须处理编译器中所有尚待“收尾”的工作：包括支持 Macintosh 特有的 C 语言扩充功能，而这些扩充功能设计上的用意是使它能与 PASCAL 的程序兼容，像是\P 可以支持 PASCAL 的字符串、call-back 的呼叫方式、ROMtrap、占 4 个字节的长整数（long）资料类别等等；还有修改 80x86 的连结器，使其支持 MC680X0 指令，并能产生在 Macintosh 上的可执行码；此外，我们还得编写开始执行的激活码（startupcode），补充一些 C 的函数库，还要写在开发用的 PC 与测试用的 Macintosh 之间控制传送的程序……有一堆事情要做呢！

您不一定要设定如此积极的目标，实际上有些细节看起来虽小，其实是很不容易做到的，尤其是可能影响别组的工作，所以这方面可能难免需要妥协。我想最重要的是目标要详细清楚，同时得注意不要忘掉细节部分。

我当然也可以照优先级来分割小项目，把比较重要的工作排在前面，但是如此的安排就不可能产生那种惊喜的效果，亦没有激励的作用，也就失去分割小项目的主要用意了。

为了让小项目既能振奋人心，又够实际，我们不采用简单的测试程序（像“helloworld”这样的），而是在每一个小项目中做出真实、完整的应用程序，也许规模小些，但该有的功能绝不能少。由于做的是真正的程序，我们完全可以看出编译方面的搭配表现以及一切细节，而这是在测试性的或是简化版的程序中看不出来的。在工作清单上也能完全反映出真实性，凡是事先无法规划到、临到头来却又非做不可的工作（这是必然的，很正常），都会在小项目中适时冒出，不至于快到了期限才发现事情怎么也做不完，日程表却无法推进。

如果您的上级主管不了解分割“主题式”小项目的这些益处，他也许会怀疑您是不是用射飞镖的方式随意安排日程，因为传统的观念里，优先级高的事情应该先做才对。

为了保持创意的活力和团队士气，必须让每一个小项目都有令人兴奋的结果。

理想的期限

人们常常忘记一点：日程表上的完成期限是推估出来的，这个期限未必最符合成本效益，事实上也未必合理，因为工作项目都是估计出来的，不能代表实质的工作项目。简言之，日程表上的完成期限是个理想，有很多情况可能使这个理想的期限变得不合理，因此它并不是真正的完成日期。大部分

的上级主管都不会乐意听到这些，但这却是千真万确的事实。使用阶段性的小项目可以让事实更接近理想，但也不是完美的做法。

身为主管的您，一定要时常提醒组员：产品的质量远比遵守期限重要。请牢记本章所提供的经验教训：

最会误导项目发展、伤害产品质量的事情就是过份重视进度，这不仅打击人员士气，还会逼迫组员做愚昧的决定。

我相信尽力在期限前完成工作仍然是您的理想目标，但请记得我强调过预定期限是理想而非现实，如果您发现自己为了期限而得做不当的决定，请悬崖勒马，以免造成后悔莫及的损害。

重点提示

如果您定的日程表使组员产生“落后恐惧症”，为了赶上期限而牺牲了产品的质量，那么该检讨的是这个日程表而不是组员；如果您定出的日程表是个无法达到的目标，只是为了从组员身上压榨出更多的工作时间，那只不过是打击团队士气，对产品毫无帮助。一旦组员发现自己身处绝境，那您永远无法让他们表现出最佳状态，等到项目结束（也许更快），他们就会另谋高就，找个是人做的工作。

将项目分割成数个小项目，各有阶段性目标的做法，可以让组员更加投入，并且营造出“赢”的气氛，让组员受到项目有进步的鼓舞。理想的小项目期限大约是两个月，这样给组员适当的急迫感，而促使他们积极地工作，特别是当小项目有一个明显又令人振奋的主题时。您应该试着把小项目设计得令人兴奋又期待，使小组在完成后有股冲动想说：“哇！看看我们完成的工作！太棒了！”随着每一个小项目的完成，小组会有愈来愈强的信念，相信自己的工作是非常重要的，对使用者而言是非常有价值的。觉得自己的工作有价值、有贡献，这是一种很大的成就感，这种感觉最能鼓舞组员凝聚团队的力量，共同创造出最优秀的产品，而且会很快地做出来。

第 6 章　学无止境
ChapterSix

在今年的冬季奥运中，花样滑冰赛的电视节目带给我很大的感触。节目中回顾 25 年前的赛事，早期金牌得主只要靠几个后仰、蹲立、旋转、腾跳和高雅的舞步就可以夺金，但是相同的表演在今天恐怕连个小镇的滑冰冠军都拿不到，现在的参赛者要想在大赛中脱颖而出，至少得做到三次三周跳跃、好几个花样跳跃、高难度旋转、以及充满美感的舞步，更重要的是，整体的表现要有“个人风格”，否则就只能拿到普普通通的分数，更何况赢得金牌。

有一次电视转播时，记者提到卡塔瑞娜·威特（KatarinaWitt）小姐打算以她六年前赢得卡加立（Calgary，加拿大西南部的城市）奥运金牌相同的舞步出赛，他随即残酷地评论道，就算威特小姐的表现和六年前一样好，她也不可能有好名次，因为六年前的东西在今天已经没有竞争力了。

其实，仔细想想，现在的花样滑冰选手真的比 25 年前的选手条件更好吗？当然是，但绝对不是人类体能的进化。最主要的原因是滑冰选手每年、每季都在提高自己的标准，因为他们都想胜过上一届的全国或世界冠军。25 年前的选手当然也可以做到今天的高难度动作，但是以当时的标准来看不必这样辛苦，所以他们也就没有把自己的潜能开发到这种程度。

相同的情形也出现在程序设计的领域。在 People-ware 一书中，作者汤姆·德马克（TomDeMarco）与蒂莫斯·李斯特（TimothyLister）曾经比较过不同公司程序设计师功力的差异。这两位作者举办了一场“程序设计大战”，邀请不同的公司各派两位程序设计师参加。他们发现程序设计师之间的表现可谓天壤之别，所需时间甚至有 1 比 11 的差距。更令人惊讶的发现是：来自同一家公司的程序设计师功力会差不多，如果一位很差，另一位也好不到那里去，如果一位很棒，另一位也会表现不错，即使这两位程序设计师分属不同部门也是一样。两位作者指出，公司培养程序设计师的环境当然是因素之一，但是我认为高达 11 倍的差异，最主要还是因为各公司对程序设计师的平均技术要求不一致使然。

您上一次听到组长对组员说：“我对你很失望，你只是做完了份内的工作，并没有追求进步。”是多久以前的事？程序设计师是否功力到达了公司的平均要求水准之后就停滞不前，即使还有很大的成长空间，也因为学习的压力已经减轻，就不再求进步了？公司有没有察觉到这种现像？程序设计师就像 25 年前的花样滑冰选手，觉得自己够好了，但其实还有精益求精的空间。而另一方面，项目经理只关心那些功力还不够的程序设计师的学习状况，不太有时间对已经合格的程序设计师加强训练。

如果程序设计师只是完成份内的程序设计工作，那还不够好，一位讲求效率的管理者应该不断提高对属下的要求标准，就像花样滑冰选手的教练一样，当您提高了对组员程序设计功力的要求，也会带动整个公司的程序设计功力水平向上提升。

五年资历的笨蛋

有时候我看到有五年（或更久）资历的程序设计师，一直都在同一个团队，做同一个产品，做得久不是问题，问题是他还在做同样的工作。如果他当初被分派到 Excel 的项目，负责开发 Macintosh 版特有的功能，那么他就

一直做下去，五年下来成为此特定领域的专家；如果他当初被分派到负责编译器的优化工作，那么多年后他还是与世隔绝地做着同样的工作，当然已经变成专家了。

从项目的角度来看，为了发展最精良的产品而把旧人留在原工作小组，是个不错的主意，任何人都是做自己熟悉的事情时速度最快，但是如果没有适当地教育他们，结果就适得其反，您等于是剥夺了他们扩展新视界、学习新技术的机会，对程序设计师不利的话，最后也等于是对项目、甚至公司不利。

假定有一位新聘的程序设计师，在第一年时成为档案转换的专家，于是在以后的四年里专门为各种产品的档案格式写转换的程序，这件工作绝对是重要的，但是他的技术只有在第一年里大幅提升，其余的四年都在重复旧的工作，没有学新的技术，事实上他是停滞不前了。他有五年的工作资历，但不是五年的工作经验，他只是用五年的时间重复第一年的经验罢了，他的五年，其实是五个一年。

如果他在后面的四年里接触应用软件的其他部分，他的技术范围就会比较宽广，如果他一直在开发某一个 Windows 或 Macintosh 的应用软件，每年都在一个主流领域中负责不同的工作，那么他五年下来可能具备完整的历练，他会知道以下的各种技能：

◆如何制作使用者界面函数库，包括菜单管理程序（menumanager）、对话管理程序（dialogmanager）、窗口管理程序（windowsmanager）；并且利用这些函数库来作为使用者界面的基本组件。

◆如何使用线上求助函数库（helplibrary），为应用程序中的各个对话框提供适宜的辅助说明。

◆如何运用绘图函数库（graphicslibrary）在屏幕上显示各种不同的形状，位对映图（bitmap），处理调色板，控制各种不同的显示装置等。

◆如何自打印机打印输出结果，并让每种打印机都发挥最高质量，充分运用每一种打印机的独特功能，例如支持 PostScript 的打印机，就能绘出水印和极细的线。

◆如何处理国际版中不同语言的字码问题，例如双位字节（double-byte），某国特有的时间和日期的格式、文字排列方向等等。

◆如何处理在网络环境中执行应用软件所可能发生的各种问题。

◆如何与别的应用软件交换资料，从最简单的剪贴簿，到极复杂的 Windows 动态数据交换函数库（DynamicDataExchangelibrary）或对象链接与内嵌的函数库（ObjectLinking＆Embeddinglibrary）。

◆如何撰写跨平台的程序，让本软件能够在市面上流行的各种操作系统——MS-DOS、Windows、WindowsNT、OS/2 与 Macintosh——上执行。

◆……

您大概可以看出来了，以上是一位在微软的 Windows 或 Macintosh 的应用软件有五年开发经验的人应该养成的技术——如果他在不同的领域都锻练过，新的要求、新的工具都会促使程序设计师学习与成长。

请比较以下的两种方式：当您在建立一个新的部门时，有一位专才和一位通才让您选择，两位都是五年经验，您会比较倾向用谁？

通常项目经理在分派工作时，很自然会让最擅长这件工作的人去做这件事。他会让最精通档案转换的人去做档案转换，这位仁兄做了五年的档案转

换，肯定没有人比他更行了。除非所有的档案转换专家都扬言再不让我做自己感兴趣的东西，我就跳槽，否则项目经理是不会改变主意的。

项目经理的理由是：“如果不让最擅长做档案转换的人来做，反而派一位新手负责档案转换，不就慢得多吗？”再不然就是异曲同工的：“如果没有把每件工作都派给能做得最快的人，那不就徒然增加开发的时间吗？”

如果您把项目当成一件临时任务，着眼点是追求时效，那么，这样的想法并没有错。但是如果您把项目当成一种长远的理想来追求，那么，您就应该培养组员各方面的专长，固然目前的工作速度会慢些，然而几年后您就拥有一个阵容坚强的团队，任何一位组员，都有能力处理任何一种问题。万一出现一个难缠的错虫时，您不必去找那唯一的一位专家，任何一位程序设计师都能解决这个错虫。如果您要在产品中增加一项跨领域的功能，任何一位程序设计师都能做，而且所有的程序设计师都能了解如何共享子系统和其他人的程序，就可以避免重复写同样的程序，并且有能力改善整体的产品设计。最终来说，通才的培养对整体效率还是有利的。

您的组员在新的领域中摸索时，难免会多花点时间，也因此才会学到足够的经验，花在学习的每一分钟，将来可以节省更多时间，因为他们可以在各种不同的领域应用学到的技巧。所以，持续性的训练、培养是必要的投资，以后会带来不可限量的回馈。

不要让程序设计师的学习停滞不前，要让程序设计师有机会磨练不同领域的技术，培养十八般武艺样样精通的组员。

受用无穷的技术

在微软，一位新聘用的程序设计师刚加入一个项目时，通常会派给他基础性的工作，例如追踪错虫、修改程序等等。渐渐地，这位新人比较熟悉这个软件了，就可以增加工作的难度，直到他能够一人承担一个程序或全权负责完成一项功能。这种循序渐进的方式很合理，因为新人对这个软件完全不了解，不可能让他做太重要或太困难的事情。对于这种训练新人的模式，我唯一不同意的是，微软分派工作是以难度来区分，而不是以新人学习范围的广度来区分。当您分派工作时，千万要把“在工作中学习”的观念牢记在心，设法让每个人都能在所分派的工作中学习到新的技术，即使要让一位新手做困难的工作也无妨。这种做法不单是为了培养人才，对公司也很有益处。

例如电子表格的应用程序，工作的范围可以从一个对话框，到重行运算（recalculation）的核心程序，范围是相当广的，而程序设计师从这两种截然不同的工作中，可以学到两极化的技术：“对话框”是简单的、到处都用得到的使用者界面，讲求亲和力，而“重行运算”则需要对整个软件的程序结构非常了解，讲求效率和逻辑性。对程序设计师而言，“重行运算”是一种很好的训练，颇具挑战性，对于本项目的重要性也很高，但是这种技术只有在电子表格应用软件中有价值，在其他的产品中就没有重行运算这类的程序；而对话框则是几乎每一种应用软件都用得到的。

提高对程序设计师的要求水准，不仅能提高本组程序设计师的技术层次，还能提高全公司开发人员的平均素质。您可以随意分派给程序设计师不同的工作，让他们随时保持学习，但您也可以采取更好的办法，在分派工作之前仔细分析每项工作所需的技术，然后把它分派给最需要学习这项技术的

人。一位有经验的程序设计师应该已经知道如何写对话框、控制字号、多窗口切换等等，一般用得到的技术他已经学全了，比较难的电子表格的宏指令处理他也做过了，为了让他持续不断地学习，就应该派他做一些很特殊的东西，例如写一个效率极高的重行运算程序。

一位新进的程序设计师则不妨从最简单的对话框开始，再慢慢磨练多窗口切换处理之类较难的工作，让他不间断地学习，等到一般性的技术都纯熟了，再来考虑让他做特别的工作，若是此时他正好被改派到别的项目去，这些技术还是非常有用的。

这又是一个方法简单，用处却很大的例子。其实您无论派给一位新人什么工作，对项目的进行来说都没什么差别，所以您不妨给新人磨练各种不同技术的机会，而且一开始就培养他一般性的技术，使他不论到哪个项目都能有贡献，这样您也是替公司培养有价值的人才。

训练新进程序设计师时，先培养他对整个公司所有项目都有价值的技术，然后才培养本项目独有的技术。

让专家再重新学习

如果您让组员尽量接触不同的程序，学习新的技术，总有那么一天，他会把本项目所有的程序全都摸熟，就暂时没有成长空间了。虽然您可以就让程序设计师停滞不前，让项目享用他们的专长，但是为了公司的利益着想，您应该把没有新东西可学的程序设计师推到别的项目去。如果您让程序设计师的学习停滞，等于是伤害到公司程序设计师的平均素质。把所有的程序设计师放在他们可以成长、可以进步的地方，这是您对程序设计师和公司的义务之一。我在开玩笑吗？绝不。

任何主管都舍不得既优秀又有完整专长的程序设计师离开，即使他们留在本项目也没有成长的机会。我为什么主张反其道而行？是我疯了不成？请容我说明。

在第 3 章中，我谈过受到 Word 小组抱怨的对话框函数库小组，那时我并不是这个函数库的组长。不久后我当了组长，而当时的首席程序设计师对我表达意愿，他的学习似乎到了高原期，再待在同一个项目的话，恐怕学不到什么新的东西，而且，他本人对于一直做同一套软件已经感到厌倦，他需要扩展到新的技术领域。

我问他有兴趣的是什么工作，有没有想去哪个项目，他向我表示，微软内部有一个新使用者界面的实验室有空缺，如果他能加入，一定能做出创新的作品。看起来这似乎是他梦寐以求的工作，所以我去找那位实验室的主任谈，确定这是让他学习的好机会，我就慷慨放人了。在一个星期之内，我手下最强的程序设计师离开了我的项目，这下我怎么办呢？

很多主管突然失去最倚重的手下时都会惊慌失措，我正好相反，我觉得这是一个大好机会，让别人能学习成长以填补他的重要性，我也可以让别的组员学习承担他以前扛起来的责任；果然，有一位组员自告奋勇，接下他的工作。之后偶有机会遇到那位实验室主任时，我问他项目进行得如何。他很高兴地说：“我真是做梦也想不到，我们做得比想像中的好太多了。”原来他只能得到一位初级的程序设计师，但是却有一位经验极丰富的高手乐意加入，使得他的项目出乎意料地顺利。

您可能会以为失去最棒的程序设计师会对项目造成难以弥补的伤害，事实上我绝少看到这样的情况。短期内难免要挣扎适应，长期来说对公司却有很大的好处，少了一位停滞不前的程序设计师，多了两位积极学习的程序设计师，和一个进行十分顺利的项目。这样的结果其实不令人意外，只要员工成长，就是公司资产的增加。

不要舍不得放您最优秀的程序设计师到别的项目去。如果他在您的项目已经没有新的东西可学，为了公司和他个人的前途，您应该把他推荐到别的项目，让他的成长永不间断。

交叉传授论？不灵啦！

偶尔我会听到这种说法，认为公司应该让程序设计师交叉轮换，好互相传授技术、学习技术，就像是植物的自花授粉（cross-pollination）一样。

这套理论听起来挺吸引人的，因为它的目的是促成程序设计师的彼此学习、改善项目开发程序。但是经验却告诉我，这不是个好办法，它很难达到预期目标，理由很简单：忽略了基本人性。赞成这个理论的人认为，一位新人加入一个团队，这个团队就会教他最多的东西，好让他能够很快进入情况。但是有多少人在刚进入一个全新环境时不会紧张的？有多少程序设计师在繁忙的工作之外，还愿意对新人倾囊相授？就算这些都不成问题，反过来说，新人可能带来他原属部门的看法，认为事情应该如何做才对，但是这个环境中的旧人一定很难接受，即使新人是来当组长的，为这个项目带来全新的看法也不错，如果只是个基层程序设计师，那就得要数年的功夫才能让其他人潜移默化吧。无论如何，双方都得很辛苦地互相适应，冲突可能在所难免，也许很久以后才能渐渐突破各人心中的藩篱。

赞成交叉传授论的人认为新人会带给小组新的知识，实际的状况却正好相反，新人无法带给小组新知识，反而是从新加入的团队中获得新知识。新人发现自己身处陌生的环境，不得不加倍学习，尽快跟上。这大概是交叉轮换的唯一好处。但是这个人既然能在原属的团队中继续学习，又何必非把他调离呢？所以，让已经学无可学的程序设计师离开团队就好了，不是所有人都该调来调去，您最好不要期望交叉轮换可以带来什么真正的交流，事实上是弊多于利的。

“新年新希望”症候群

项目经理想要让组员有系统地学到新技术，并不是分派项目的工作就行了。比方说，要想学习当个好组长，就得有决心、有计划按部就班地追求这个目标。这应该是主动而积极的学习，和在工作中“顺便”学习的态度是不同的。如果您希望组员追求高度的自我成长，每天都要有进步，您就得要求他们主动地设定更高的目标。

传统的做法是把这些目标列出来，当成每年的考绩评量项目。这样做的结果是，除了少数特别自觉的人之外，大部分的人都在一星期内就忘光了自己所立下的目标，等到一年结束，主管就会很失望地发现达标率实在非常低。我想大家都有这种经验吧，这叫作“新年新希望”症候群，只不过，时间未必在每年的新年罢了。

这种目标之所以迅速被遗忘、或是终告失败，主要的原因是没有一套完整的实施计划，或是就算有计划，却像我在第 4 章提过的检查报告一样不够明确、实际，而无法有效地执行。就好像是随便说说今年的新希望是“我要发大财”一般，只不过是对着流星许愿，没有具体的目标、没有计划、没有

期限、没有决心为目标付出血汗努力。

有一个方法倒是可以确保组员的成长，那就是把个人的目标和项目目标结合起来，每两个月阶段性的小项目，必须加入至少一项个人目标，一起事先规划，一起届时检验。这种方法让组员每年都有六次的进步，完成六项以上的个人目标。

每一项目标不必全面兼顾，可以只针对特定领域：阅读一本科技或商业的好书，或是一步步培养追踪错虫之类的良好工作习惯，或是改正工作上的不良习惯（例如不要摸来摸去、写程序之前先构思好等等），都可以。

最近读过什么好书吗？

我个人经常读书，充实自己的知识，不断了解新的观念。您拿起一本好书，只要花几天的时间阅读，就能吸取别人多少年来从错误中获得的体会，这不是很划算吗？如果每位组员一年至少读过六本好书，那么他就能学到六份别人的经验智能，对他会有多大的影响啊。我特别喜欢读与执行策略有关的书，这也是我撰写《零错误程序》一书的动机之一。不过我的书并不是创举，在我之前有伯瑞恩·科奈汉和普劳格（BrainKernighan & P.J.Plauger）所著的《程序设计风格之要素》（TheElementsofPogrammingStyle），1974 年出版，至今仍然风行，价值不减，乔恩·本特利（JonBentley）的《设计高效率程序》（WritingEfficientPrograms）也是一本相当精彩的策略书籍，安召·克宁格（AndrewKoenig）的《C 语言陷阱与危机》（CTraps＆Pitfalls）是专为 C/C++程序设计师所写的，值得一读再读。

除了这些策略性书籍之外，有关软件开发方面也有无数的好书等您来品味，葛纳德·温因伯格（GenaldWeinberg）的经典之作《计算机程序设计心理学》（ThePsychologyofComputerProgramming），以及较晚的史蒂文·麦克奈尔（SteveMcConnell）所著《完整的程序》（CodeComplete），里面有一章很详尽地介绍“哪里有更多的资料可供参考”，为意犹未尽的读者指引下一步，告诉您哪里有软件产业最好的文章。

不过可别因为自己从事软件这一行，就把阅读范围局限在软件这个领域了。马克·麦科曼克（MarkMcCormack）所写的畅销书《在哈佛学不到的管理知识》（WhatTheyDon’tTeachYouatHarvardBusinessschool），就是一本适合任何一位现代人阅读的好书。迈克·格伯（MichelGerber）的《E 神话》（TheE-Myth）讲的是如何建立连锁店的方法，以软件开发单位的眼光来看，书中有很多可以援引的技巧。也不要以为这类的书只是给管理者看的，即使是最嫩的程序设计师也可以从中得到启发。

为了确保个人的兴趣和项目目标契合，我鼓励组员们自由选择想追求的技术，而我只要求大家的个人目标必须符合以下的条件：

◆这项技术必须是对程序设计师本人、项目、公司三方面都有用的。学 LISP 也许是个人的兴趣，但是对于微软这种公司而言没有任何用处。

◆这项目标必须是可以在大约两个月的时间内完成的。每个人都能在两个月之内读完一本技术方面的书籍，但是想要“精通 C++”就不是短时间内做得到的。

◆这项目标必须有一个可供评量的指针。“成为一位更好的程序设计师”就不是一个可以评量的目标，但是“养成习惯运用除错工具来随时追踪程序，以便早期发现错虫”，这种目标就比较容易评量是否能做到。

◆最理想的情况，我希望目标是对项目立即派得上用场的。如果程序设计师没有立刻使用刚学到的技术，等到几年后他真正要用时可能已经忘了一大半了。符合以上条件的目标，可以很自然地达到程序设计师、项目和公司

的三赢局面，既是个人需要用到的，也对他的晋升有帮助。如果程序设计师想不出来自己可以专攻哪一项技能，或是没有明显的兴趣，项目经理就不妨这样为他选择：“ 如果我想让他晋升，那他还需要学些什么才行？ ”

确定每位组员、每两个月都有一项技术上进步。

训练您的接班人

除非这位程序设计师有充分的理由相信自己能够跻身管理阶层，否则一般的程序设计师不太愿意主动学习管理方面的技巧。因此您必须找到一位对管理工作比较有兴趣的人，协助他获得担任管理者所需的一切知识和技巧。除非您打算永远带领这个团队，否则您一定会需要一位接班人。如果您没有接班人，您就会发现自己被困住了，永远没有办法换一个新的或是更有趣的项目，因为没有人能接替您目前的工作。

实时的学习

有一个很好的方法，可以让您帮助组员找到最适合他们学习的技术，就是每当您看到某一个问题或是机会时，立刻为他们设定一个成长目标。每当我发现程序设计师抓错虫的效率不高时，我立即指点他们用比较高明的方法，同时要求他们在未来几周之内熟练这个方法；每当有一位程序设计师表示他想学习如何写程序才会有比较快的执行速度，我就给他一本乔恩·本特利所写的《设计高效率程序》，并且要他保证会认真阅读，而且和我讨论；每当我发现某人的程序风格（programstyle）很容易出错时，我会立即告诉他我的看法，并且要他答应我一定要改正，以后不再用这种风格写程序。

我坚持发现错误时必须立即改正。因为实际上，这种学习效果会更快更好。想想看，您把程序设计师今天写的程序拿给他看、要求他改进，或是拿他几天前写的程序指出其中的缺失，那一种比较能令他印象深刻，并且感受到这项学习是很严重、很认真的事情呢？

我曾经训练过的组长中，有一位每次一有问题就跑来找我。他会问这样的问题：“ 那个 WordSmasher 小组没有时间做 Anagram 的功能，他们想知道我们能否帮忙，我该怎么办呢？ ”由于他来找我咨询的次数太频繁了，我认为他并没有独立思考，我把这一点告诉他，他的回答是，他每次来问问题之前都把可能的解决方案彻底想过了，但是他不想犯错，所以来问我比较保险。我对他说这种方式太依赖了，我们应该一起解决这个问题。

我了解这位组长需要别人的肯定，所以我告诉他只要他感觉有任何问题，还是可以尽量找我讨论，但是并不是把做决策的责任推给别人，因此，我要他在提出问题时，做到下面几点：

◆向我解释整个问题。

◆有系统地描述他所想到的解决方案，包括他赞成的和反对的。

◆告诉我他选择的解决方案，并解释理由。

在这位组长照着我们的“ 约法三章 ”做之后，我对他的印象马上大幅扭转，其实他的思虑还挺周详的。十之八九我只要对他说：“ 很好，就这么做。 ”就行了，偶尔我们会有不同的意见，我就把我的看法解释给他听，两人讨论一番，他就会有全新的见解，有时候是我会有全新的看法。有时候我会有不同的建议，纯粹只是因为我俩观点上的不同，其实我的建议执行起来未必最有效，这时候我们就决定还是照他的方法做。

这种新方法对我们两人都没有增加什么负担，但是我们的结果和两人的关系却是比以前改善太多了。以往我总是觉得他要别人代他做决定，现在我欣赏他自己做的决定，我对他的态度也有 180 度的转变，从“这家伙太依赖别人，自己从来不去用脑筋”变成“这小子真是考虑周到，决策都很正确”。他自己的态度也有相当显著的转变，从非常害怕做决定、害怕做错，变成相信自己深思熟虑后的决定。渐渐地，我们连那种“我该怎么办”的讨论都变少了，最后完全消失，现在他只有遇到真正无法独力解决的难题才来找我。

是什么造成这样大幅的转变呢？我并没有对他施予什么训练，他也没有刻意培养什么才能，只不过是沟通方式上的一点小小调整，只因为我发现他过度依赖时立即着手改善。这就是小小的改变、大大的效果。

一发现某处需要改进，就立即采取更正的行动。

马后炮式的管理

请注意我是在发现那位组长的问题时立即把我的看法告诉他，而不是把我的意见反映在他的年终考评上，我从不认为年终考评对于个人的成长计划有什么帮助。我的经验是，主管对属下的表现有任何意见都应该立即表示出来，迟来的纠正通常毫无效果——除非年终考评不只指出缺失，还能够提出务实有效又详尽的改进计划。年终考评的另一个缺点是，主管大都很难在这么长的时间后，对组员的成长作出正确的衡量。

我们都听过许多有关考评争论的故事。主管在年终考评时给某一位程序设计师很差的成绩，但是在一整年中从来没有提过对他什么地方不满意。受到考绩的意外打击，程序设计师震惊得连说话都结结巴巴了，向主管问道：“您能不能举出一个例子，告诉我，这是什么道理，我表现得这么差吗？”主管愣了好几秒钟，终于想出一件这位程序设计师做错的事情，是的，那件事的确是他的错，但是就为了这件事情就给这位程序设计师的“全年”考评这么差的分数，听起来颇为荒唐，连主管自己也觉得说不过去，于是再搜索枯肠、用力想想这位程序设计师做错过什么事，但是通常都因为时间过去太久而根本想不起来。

程序设计师眼看着和主管也沟通不下去了，只好黯然走开。稍微有时间思考刚才的对话之后，他就会气愤不已：“为什么他从来不告诉我什么地方做错了，等到年终考评再来放马后炮？如果我根本不知道自己哪里不好，我怎么可能改进？”

这种故事我听过不知道多少遍了。

如果足球教练也这样做，那结果不知道会有多惨，难道要等赛季结束、战绩为零时，教练才告诉球员哪里不对，需要改进吗？

“疯狗，我决定下一季要你坐冷板凳。”

“什么？我不懂，我觉得我踢得很好呢。”

“是踢得不错，但是每次球要传给你时，你接球的反应都不够快。”

“我是这样吗？”

“没错。而且你因此而漏接到不少机会球，在你改进这一点之前，我必须罚你不准上场，当然，这也就是说，你的薪水将从 520 万美元掉到不足 2 万美元。不过别担心，你还享有各种福利，赛季中有免费的饮料和热狗，买纪念品时还可以打折。”

疯狗这时候可真是疯了，向他的教练咆哮道：“既然你看见了我接球不够敏捷，你为什么不早点告诉我？我可以改的啊！”

“喂，我现在就在告诉你啊，而且赛季的检讨报告就在这里，写得不够清楚吗？拿去，这是你的新合约。”

听起来很蠢，不是吗？但是很多管理者事实上就是这样做，而且做了很多年。我刚才说的那位从依赖中学习独立的组长，如果是在别的公司，不幸遇到马后炮型的主管，结果会怎么样呢？他的考评表上会写着：

太依赖别人替他做决定，自己无法独立思考。

经过一阵交换意见后，对问题的改进计划可能是这样：

我不再依赖别人为我做决定，我以后会彻底地思考每一个问题。

这种改进计划太含糊，不可能有任何效果，更谈不上改进后的衡量指标。计划中没有说明要这位组长做什么、怎么做、如何衡量成效，这样的计划根本行不通。在明年的年终考评会上，同样的戏码将再度上演一次。

我所见过的年终考评大都毫无用处，所以也不必管其中的评语和新目标了。立定目标何必一定要在年终呢？主动找出需要改进的地方，把它和项目的阶段性目标结合，这就是最有效的学习方式。而正式的年终考评是回顾并记录员工一年之内的成长，这才是高级主管应该要的。如果只是把员工在一年内需要改进的地方，一条条列给高级主管看，并没有表达出什么实质意义，反之，记录下员工确实学习到的重要技术、以及他们如何运用这些技术帮助项目的进行，既能具体描述员工的贡献，也让高级主管看出员工的进步，好让高级主管正确评定是否调薪、核发工作奖金、或是升迁。

不要用年终考评来订立学习目标，要利用年终考评来记录个人的成长。

全方位的才能

我在微软面试过的程序设计师大都是即将毕业的大学生，但是偶尔我会面试到想跳槽到微软的在职程序设计师。我发现已经进入社会工作的程序设计师中，即使工作资历相当，来自小公司的人，几乎都比来自知名大公司的人能干得多。起初我觉得很惊讶，后来我发现其中的缘故，是由于大公司的人往往专注于某一个领域，因为公司有足够的财力让他们分成三十组，每一组只有一个专长，而小公司就没这么奢侈，一个人要顶好多人用，因此每个人都得学习多方面的技术才能应付得来。这也就是我在本章提过的重点：全方位学习和随时学习。

身为一位组长，即使您有充足的预算聘用专家，您也得创造学习的压力和气氛，不论是由您亲自教授，还是让组员去上课、读书或参加研讨会，都是非常好的学习方式。只要学习是持续的，让组员保持不断的进步，您就会逐渐提升公司内“一般程序设计师”的技术水准，就像冬季奥运会的花样滑冰选手一样，不断挑战极限、突破极限。这样的话，不管是对您的组员、项目或是公司，都是极大的好处，最后，您的顾客也是赢家之一。

重点提示

绝对不要让组员一直做同样的工作，这样是限制了他的学习，使他停滞在原来的领域。一旦程序设计师精通了某一个领域，就让他换别的领域做做看，永远让他们学习新的技术。

各种技术的用途范围有所不同，有的技术在一般的项目都用得上，有的技术只有在特定性质的

项目才用得上。当您训练您的组员时，必须让他们的技术能在公司发挥最大的用处，最好的办法就是，把应用范围最广的技术放在训练的最前期，应用范围最小的技术放在最后训练。

优秀的程序设计师是项目经理最需要的，所以经理们通常舍不得让自己手下功力最强的人到别组去，但是如果这位第一高手在本组内再也没有新东西可学时，经理就应该让他到别的项目去，一方面他个人可以重新开始另一次的成长，一方面让接替他的人学着承担重要的工作，最后公司的平均程序技术水准因而提升，对大家都很有好处。

为了确保每位程序设计师的技术都在稳定地进步，一定要让每个人有个努力的目标，最好的方法是把个人的成长和项目每两个月的阶段性目标相结合，这样一年就有至少六次的进步了。假定一位组员在公司待了五年，那么他就学了 30 种新技术、或是读了 30 本好书、或是 15 项技术加 15 本书，对他的工作能力影响多大啊。

最好的成长目标是出于当时的需要。如果您发现有位组员工作缺乏效率，或总是在犯同样的错误，最好抓住机会立即为他立一个目标，并且要求他立刻开始改进。这种当时设立的目标让人印象深刻，又是马上寻求改善，效果通常会非常好。比起年终考评那种模模糊糊的建议，更能引起程序设计师的重视。

第 7 章　态度问题
ChapterSeven

在第 6 章中，我主要在强调学习的重要性，组员的技术和知识应该精益求精。让组员接触他没做过

的新工作会促使他学习，鼓励他多看书、养成良好的程序习惯会有更好的结果，但是还有一种最深度的改善，发自程序设计师内心的，就是开发软件产品的良好态度。

错误的态度

我在第 1 章中提到过一个基本概念，就是专业的程序设计师应该在合理的时间内，开发出有价值，而且零错误（bug-free）的程序代码。但是很不幸的是，要写“零错误”的程序实在很难，否则每个人都能写程序了，何必要专业。

在软件业有一个公认的观念，就是错虫是无法避免的，而且除了发现错误时把它清除之外，没有什么好的对策。虽然这个观念如此普遍，却是大错特错。零错误程序是可以达到的目标，只不过需要额外的努力，而通常程序设计师不愿意为这个理想付出，除非他们真正了解“零错误”对软件产品开发的重要性。

有一个非常简单的方法，我经常用，就是在编译程序时，选择预警功能（warningoption），也就是让编译器来帮您找到比较明显的错误，编译器会警告您这里可能有错虫，并且把原始程序代码列出给您参考。例如很多 C 语言的编译器都抓得到这种错虫：

```
if（ch=tab_char）/*注意这是单等号，应该是==才是比较的意思*/
```

虽然这一句指令是完全符合 C 的语法，但却隐含了一个错误，程序设计师的本意是比较这两者，结果却变成把 ch 的内容设定为 tab_char 的值。正确的写法应该是：

```
if（ch==tab_char）/*注意现在已改成双等号*/
```

让编译器来替您做初步的纠错，显然是很方便的做法，但是我就看过很多程序设计师拒绝使用这个功能，他们觉得这些警告的信息看起来很恼人，会干扰自己的思路，因为有时候就是要在 if 条件式中放单等号，编译器也会不分青红皂白地当作警告列出来。例如这样的 if 条件式：

```
if（ch=readkeyboardo())
  {……}
```

这段程序代码有两个作用，一是执行 readkeyboard()，把传回值留给 ch 变量，一是若传回 null 字符，则 if 条件不成立。它是正确的，但编译器却会给程序设计师警告，看起来有点烦人，于是就得改成这样：

```
ch＝readkeyboard（）；
if（Ch！＝nul_char）……
```

或是更简洁的：

```
if（（Ch＝readkeyboard（））！＝nul_char）
```

这两种方式都会编译出一样大小的可执行码，差别只是在明显表示 ch 与 nul_char 的比较关系，或是隐含表示二者的比较关系。对于大部分的程序设计师而言，这几种写法都一样容易理解，而如果要快速浏览程序的话，又

以第三种（只有一行）最快。

话虽如此，就是有顽固派的程序设计师不肯使用编译器的预警功能，而认为“我要怎么写程序，是我的自由”或是“编译器根本就不应该对语法正确的程序代码发出警告”。如果您看到这些程序设计师对我的建议所表现出的态度，您真会觉得我是在劝他们扔掉 PC，回头去使用打孔卡片的古董计算机来写程序呢。

这就是程序设计师面对错虫的态度问题了。以我个人来说，因为我使用编译器的预警功能已经成为习惯，我并不觉得警告讯息会困扰我，除非我不小心留了错虫在程序里，编译器也不会用警告讯息来烦我，因为我写程序的习惯不会使用 if（ch＝readkeyboard（））这样的方式。对我来说，只要能找到程序中的错误，程序风格上的修改是微不足道的小事。在我看起来，那些拒绝使用预警功能的程序设计师关心个人的表现远甚于程序的正确性，如果他们连这一点小小的改变都不肯，又怎能期望他们做到重大的改变呢？如果全部门或全公司从现在起规定新的命名原则，或是固定的程序风格，怎能期望他们遵循呢？他们能不能放弃自己喜欢、但容易出错的程序风格？他们会肯用除错工具来一步步地追踪程序的执行步骤，以期及早发现错误吗？

是的，撰写“零错误”程序的确需要下功夫，而程序设计师大都不愿下这些功夫，除非程序设计师有正确的态度：没有任何理由，有错虫就是不行。在我的项目中，我会仔细追查错虫清单上的每一个错虫，看看这个错虫是不是应该早在单元测试（unittest）阶段就应该发现，或是在用除错工具追踪时就该被找出来。如果是程序设计师粗心，让这个早在开发时期就能发现的错虫留到整个系统建置完毕时才被发现，我就会认为他的工作质量不及格，应该再重新训练。

有些比较笨蛋的程序设计师会太早认为自己已经完成工作，觉得编译能够过就表示没问题了，因为他们的基本态度还没有认识到错虫是多么严重，程序无论如何就是不能有错虫。他们会说：

我认为程序已经完成，因为编译都没有任何错误，而且执行起来似乎蛮正确的。

笨蛋程序设计师会有这种态度，因为他们没有被各种难缠的错虫搞得七荤八素过——溢出（overflow＆underflow）错误、有号型别与无号型别（signeddatatype＆unsigneddatatype）错误、一般性错误、运算子优先级的错误、逻辑的错误以及特殊状况的错误。有太多种“地雷”，笨蛋们还不懂得如何把它们嗅出来，也不懂得它的可怕。

除错要趁早

我严格要求程序中绝对不允许有错虫：要程序设计师在写程序的当时，就必须用除错工具一步步地追踪程序的执行步骤、确实执行单位测试，因为如果让任何一个错虫留到整体测试时才被发现，那不知道要花多少倍的时间才能找到它并清除它。

一旦错虫进了产品，不只是伤害产品的质量，还会赔上大量的人力和时间。负责除错的程序设计师一时不能投入开发工作，要追踪到错虫的藏身之处、加以修正、重新测试（找错的话再重来）……测试结果正确之后，报告错虫已经清除，再做一次整体测试，确定清除错虫的动作没有伤害到其他正常的程序代码，然后再来一次回归测试（regressiontest），回归测试是不能自动化的，必须由人工执行并核对所有的功能，完全正确才算真正过关。清除错虫是多么浩大的工程！

比较起来，在写程序的时候就确定没有任何一个小错虫，已经是相当单纯的事了。如果程序设

计师认真使用除错工具、如果在严格的单元测试之后，才把程序置入系统，我上面所说的浩大工程都可以避免。所以，怎么说都是愈早除错愈划算。

凡是经验丰富、甚少出错的程序设计师都知道，那怕赌注再高，他们也不敢赌自己的程序毫无错虫。和笨蛋不同的是，他们绝对不会觉得自己的程序没有错虫了，即使再熟练，他们还是要经过测试才相信程序的质量，他们会认为：

除非我已经完全测试过，没有错虫了，否则程序不能算完工。

有这种态度的程序设计师会不会测试过度？我从来没见过这种事发生，因为是自己写的程序，一定知道什么样的测试是不必要的。能当程序设计师的人，一定聪明到知道什么样的行动是浪费时间，但常常没有聪明到知道什么样的测试才够彻底。其实，不做重复的测试，这很容易，反而是要保证测试工作毫无疏漏、每一种情况都测到，颇为困难。

要让每一位程序设计师都明白，写出零错误程序是很不容易的，所以应该多花功夫用各种方法做最彻底的测试。

不愿下功夫

每当我审核产品设计和原始程序代码时，我总是自问：“这个设计或程序出错的机率有多少？”我找出缺点，并试着评估修改的代价有多大，一旦发现缺点，要么修正它，要么就是在程序中加入除错码（debugcode）来特别监视这一部分的程序。

有一次我在审查一个利用一组庞大的数字矩阵来控制的功能，其实用数字矩阵来执行这个功能是一个很好的主意，但是我恐怕它容易出错，因为数字矩阵的内容可能根本就是错的，这个程序恐怕经不起这种错误，所以我要求程序设计师加入除错码来防止。程序设计师想也没想，脱口而出道：“写除错码太浪费时间了。”

听到这句话的刹那间，我脑中警铃大响……

因为这位程序设计师完全没有想到“写除错码的要求是否合理”，也没有想到“这样做能否符合项目的目标及工作的优先级”，而只想到要花太多时间，或者是他要付出多少辛苦。

等这位程序设计师冷静下来后，我告诉他我反对他这种态度，并解释我的理由，然后请他依次以下列问题来评估我的要求：

◆在程序中加入除错码有没有道理？

◆如果有，增加除错码是否符合项目的目标与工作的优先级？

◆最后，增加除错码的重要性是否值得花时间去做？

在我们依序讨论过这三个问题后，程序设计师同意加入除错码，不过还是不太情愿。

30 分钟后，他拿着加了除错码的程序到我办公室来，给我看被除错码突显出来的三个潜在的错虫，其中两个很明显是程序的错虫，只是没有除错码的话看不太出来，第三个则令人一头雾水，我们两人都看不出来怎么会有这个错虫。一开始我们猜测也许是除错码本身错了，使得除错码产生的错误报告也不正确，但如果是这样，那前面两个错虫就无法解释了，错误的除错码怎么能正确地找出错虫来呢？我们研究了半天，终于发现原来是数字矩阵中

的资料有错，这个错虫实在太难找了！如果不是利用除错码，我们几乎不可能找到它。

这位程序设计师学到了宝贵的一课：第一，即使您已经觉得程序不可能有错，还是值得加上除错码，第二，对于加上除错码这件事，永远不能认为“太花时间”或是“太困难了”，这都是借口。

纠正程序设计师以为加除错码会花太多时间的观念，应该训练程序设计师第一个反应是考虑加上除错码是否有道理，第二是考虑加除错码是否符合项目的目标与工作的优先级。

凡事不能的态度

我和很多程序设计师、程序设计师的小组长、项目经理共事过，他们甚少想出新点子或尝试新的开发策略，因为他们的想像力在还没开始以前就先封闭了。您有没有看过那种可怜虫，提出一个新点子但是遭到众人们的重炮攻击：行不通、上级不会同意，甚至直叱道：“你不能这么做！从来没有人这么做过！”

这就是“凡是不能的态度”——对于解决问题与创造力是极大的伤害，我总是尽一切的可能消灭这种消极的态度。我的部门里有一个不成文规矩，所有的人都不准说“某件事不可能做到”，他们可以说很难或很花时间，但是不准说“不能做到”，我的理由是：

当某人说“某件事不可能做到”时，他往往是错的。

很久以前我就学会了不去管别人说什么“不可能做到”之类的话，说这种话的人大都没仔细想过究竟可能不可能。当然，您可以举出各种千奇百怪的“做不到”：明天中午以前修正 2704 个错虫……之类。但是一般人说“做不到”时，说的通常是正常的事情，千奇百怪的事情不会有人当问题来问吧？

当您听到别人说“某件事不可能做到”时，请注意他有没有仔细思考过您的问题，如果他有，考虑一下现在的状况是否和他所想像的不同，因为世界在变，尤其是软件产业的世界，也许去年做不到的事情现在只要动几下手就行了——特别是在内存空间和速度方面，变化更是惊人。几年前也许有人会认为图形界面行不通，因为太占内存，会拖垮整个执行速度，在过去这话是对的，但现在早已完全被推翻了。

有的时候会有政策上或管理上的规则是您“不可能”打破的。微软的管理者一定会告诉您，不可能有人连续升级或加薪超过最大限度，但是两项我都例外地做到了，当然非常不容易，我必须能够证明我的要求是为了公司的最大利益；因为我的要求合理，所以公司的规则才能为我例外。这并不是“不可能”，只不过“很困难”罢了。

我认为，人们之所以会说“某件事不可能做到”，只是由于这件事超出了他们的经验范围。

在 1988 年，我们即将完成 Macintosh 版的 MicrosoftExcel1.5 产品时，高层管理者已经在计划 2.0 版了，他们的目标是凡是 Windows 版 Excel 能做到的，Macintosh 的 Excel2.0 版也要完全做到，尽可能沿用 Windows 版的程序代码，不能抓来用的就另外写，但是要看起来都一样。已经在 Excel 项目做过两年的我并不害怕这个任务，我只是觉得这个计划本身有很多问题。除了外观上的相似性之外，Windows 的 Excel 和 Macintosh 的 Excel 基本上是

完全不同的软件，同时我也觉得 Macintosh 的 Excel 其实比不上 Windows 家族的软件，当然后者的团队也比较庞大，增加的功能也比较多。

微软的 Macintosh 版软件还有一个很严重的问题，就是它无法利用 1MB 以上的内存，这是由于最初在设计时的一项决策所致。因此，软件必须加载到最前面 1MB 的内存才能正常运作，使用者非常抱怨，我明明还有 7MB 的内存，为什么它偏偏只肯用最前面那 1MB？这算哪门子道理嘛！

苹果计算机公司的程序设计师在开发名叫 MultiFinder 的多任务操作系统时，发现 Excel 只认得那最前面 1MB（低地址）的内存。当时的苹果计算机操作系统对于软件的加载是采用从高地址到低地址的做法，但是他们发现除非把 Excel 放在内存的最底部，否则无法执行，最后只得来个削足适履：为 Excel 保留“特别座位”，刻意安排操作系统把 Excel 加载时放到内存的最底部，所以他们就笑 Excel 是“有恐高症的软件”。他们也要求微软改善，尽快治好 Excel 的恐高症。事实上，微软已经在彻底翻修 Windows 版的 Excel，程序几乎是逐行地重写，所以不论在质量、功能或是维护上，都远胜过 Macintosh 版的 Excel。

当我看到 Macintosh 版的 Excel2.0 开发计划：大刀阔斧地修改原始程序代码，解决它的“恐高症”，并且把 Windows 版 Excel 上所有的功能和特色统统做出来。我觉得要我的组员重做一遍 Windows 的 Excel 小组早已做过的事，这是重复的工作，况且依照这个计划来看，做好的程序代码与 Windows 小组的并不会兼容，对我来说这是时间的浪费。

我的看法是，我们何不从现有的 Windows 版 Excel 中修改，开发出跨平台的 Excel 呢？在我加入微软以前，我已经做过好几年跨平台的软件，所以我知道挑战会在哪里，而且我也确信为 Excel 软件撰写踏平台程序代码是可行的。如果我们让 Windows 版的 Excel 可以跨平台，那么 Macintosh 中自然就有 ExcelforWindows 的一切功能，不必为这些功能重复地写程序，而且 1MB 内存的问题也会消失，将来的维护与更新也不必做两套，这样不是太完美了吗？

我向上级主管提报了我的建议：放弃 2.0 版的设计计划，改为采用跨平台的设计方案，上级要我花一个星期的时间评估 Windows 版和 Macintosh 版的 Excel 原始程序代码，并且就此写出一个规划草案。

一个星期以后，我向上级主管与两组 Excel 的部门报告我的计划，使我惊讶的是，大家都持强烈反对的意见，一致表示这是不可能做到的事情，虽然我的报告相当简单明了，但是大家都只去看必须克服的困难，我就在那里被他们用“凡事不能”的态度（can’tattitude）围攻。

“马魁尔在作梦。”“Windows 版和 Macintosh 版差异那么大，怎么可能跨平台？”“即使我们做成了跨平台的 Excel，那也会毁了 Excel，程序代码会太大、太慢，最后 Excel 的每一项优点都发挥不出来。”“我们没有足够的时间了，要做也是下一版再做。”难道下一版就会比较有时间吗？甚至有一位同仁表示，如果有人威胁他非这么做不可，他就辞职。

我本来预期大家会热烈拥抱这个计划，结果我却是踢到铁板了，这件事给我一个宝贵的教训：即使是最优秀的团队，也会对未知的事情恐惧和排斥。

几天之后，Excel 小组与高级主管开会，他们做了微软内部有史以来第一次的投票表决——我的计划被否决了，不做跨平台的 Excel，原来的 2.0 版设计计划照旧进行。

不过事情终于有了转折，微软总裁比尔·盖茨看过我的规划草案后，认为跨平台的观念很有道理，所以我们最后——就像各位读者如今看到的决定要做跨平台的 Excel 产品。

接下来的八个月，全体小组的努力有了成绩，跨平台的 Excel 出炉了，原本反对者所持的理由和顾虑，没有一件发生。跨平台的 Excel 的确比较慢，但那是为了突破 1MB 限制的结果，并不是跨平台的罪过。

Excel 小组非常以跨平台的成就为荣，并且与微软的其他应用软件小组分享跨平台的经验，协助其他的小组开发跨平台的软件。

不要让凡事不能的态度阻碍了创新。

别给我找问题，给我解决方案

“凡事不能”的态度本身就是问题，它会阻碍人们创新，或者更糟的是当他们听到某一个需要解决的问题时只会觉得很烦，非但不去想如何解决，反而怨恨那些提出问题的智者。您是否有过这样的开会经验，提出建议的人被老板吼回去：“不要提出你解决不了的问题，简直在浪费我的时间！”

非常不幸，这样会导致没有组员敢提出问题，除非他有把握解决。程序设计师在开发时即使遇到严重的问题，也不敢提出，以免最后丢脸又自讨没趣——而且问题还是没被当作问题。

身为组长或管理者的人必须明白，无论有没有办法解决，有问题时一定要让组员乐意提出来。您想想看，如果在核电厂工作的人发现了反应炉边流出绿色的粘稠物，他会因为自己无法解决就只好闭口不说吗？他也许不知道怎么办，但是有别人知道啊，至少应该尽快想办法处理吧。

核反应炉是如此，开发部门又有什么不同呢？即使提出问题的人不知如何解决，但其他的人也许知道，即使都没有人知道，也可以一同想出办法。如果问题连提都没有提出来，那才是永远无法解决的。

这样已经够好了

有的时候主管要求修改某个地方，程序设计师却认为这对于使用者而言是不必要的，程序现在的样子就已经够好了。

有一回，我要求一位程序设计师示范一下他的新改版。由他执行软件程序表现的功能，看起来一切正确，只是速度稍微慢了。

我想也许是因为加入了除错码的关系，所以我问他这是不是正式版。

“是的，这是正式版。”说着他仍然继续示范。

“你有没有想过要加快速度。”

这位程序设计师很疑惑地问：“您这话是什么意思？”

“我是说，难道你不觉得它的速度有点慢吗？”

“嗯，我也不喜欢这样，但是这对使用者来说应该已经够快了吧。”

我非常惊讶：“你觉得你和使用者有什么不同吗？特别是现在，使用者就是你，而你就是开发这个程序的程序设计师啊！”

我从来都无法理解，为什么程序设计师会以为使用者——不论是其他的程序设计师、牙医或糕饼店老板并不在意执行速度或是软件质量上的问题，程序设计师以为这些问题只有程序撰写人才会知道吗？

我认为使用者才更会在乎速度和质量的问题，因为他们是真正与软件利害相关、休戚与共的人，程序的撰写人反而不是。您认为撰写 FORTRAN 编译器的人会每天使用 FORTRAN 语言吗？撰写 Word 中“邮件合并”功能的程序设计师，会一天到晚在用这个功能吗？Excel 的宏语言呢？这么多年来有数十

位程序设计师每天的工作是强化 Excel 的宏，但是他们之中有几位曾经去写使用者定义的宏？我的意思并不是说他们都很坏，长期漠视使用者，而是要强调，这些每天在开发软件的人，自己很少去当使用者，写的都是自己没有机会使用的程序，想想您的项目，您的组员是否真正去使用自己写的软件？

程序设计师自己不使用软件，当然就拉长了他们与使用者之间的距离，偶尔，会有程序设计师觉得使用者都是白痴，哪儿懂得速度和品质的问题？

为了时时提醒自己为使用者设想，程序设计师应该以这句话作为工作准则（也许您可以做成一张海报，贴在办公室入口）：

使用者和程序的撰写者一样关心速度和品质的问题。

当然是有一部分的使用者并不在乎软件执行速度稍微慢些，只要求工作正确完成，但是如果您想做出最好的产品，就不能以这种要求水平，来作为本产品所要满足的目标，您应该把要求水平最高的顾客当作本产品的目标使用者，他们会关心软件的执行速度、稳定、有无各种造成死机的错误，使用者的要求高，表示本产品有进步的机会。

不要让程序设计师以为使用者并不在乎软件的质量。

易用性

在 80 年代末期，微软开始进行易用性研究，企图找出使软件容易使用的法则。他们找了十位从未使用过该产品的人来做测试，结果有六到八位受试者不懂如何操作使用者界面，而且大部分的功能都不会用。当他们看到了研究结果，有些程序设计师的反应竟然是：“ 我们是打哪儿找到这么笨的使用者？ ”他们没想到笨的不是使用者，而是他们的使用者界面。

如果您的程序设计师有意无意间流露出使用者是笨蛋的想法，请尽快纠正这种态度。把使用者当成笨蛋的程序设计师，不会认真倾听使用者的抱怨，不会从其中试图找出改善产品的方向；那些认为使用者很聪明、有分辨能力的程序设计师，才能做出真正适合使用者的产品。组员对使用者的基本评价，会对产品产生重大的影响。

小心次功能

我过去总以为，给使用者一个慢得不像话、限制一箩筐的功能，也总比完全没给的好，至少在我们下一版做得更好以前，急需的使用者可以将就应急一下，没鱼虾也好，这是我的理论。但是后来我才发现，我错了，使用者其实不能体谅我“ 有总比没有好 ”的用意，当他们拆开包装、执行软件后，发现这项功能运作远逊于预期，心中反而升起不满和抱怨：“ 为什么总是要做两次，才能做好？ ”。

由于经常看到使用者的这种反应，我决定凡是雏型已具、未臻理想的功能，我都不把它放进正式版中。使用者既然从来未曾拥有，也就谈不上失去。如果我给他们质量次的功能，反而造成使用者对产品质量存疑的印象，对产品的信心开始打折扣，也许就想换别家的来用用看。

我很不愿意这样说，当您看到一个功能质量做得还不够理想，即使它再怎么有用，您都不应该把它放进正式版。等到下一版，这个功能已经完全符合质量标准了，再让它出门。万一这项功能非常重要，甚至可以考虑为它延后推出新版的时间，但是，一定要把它做到好，绝对不要有次品。

不要给使用者次品，宁愿延期交货，务必追求质量完美。

程序设计师要懂得顾客心理

在第 1 章里，我提过一位负责开发窗口函数库的组长，他从来没有把函数库的使用者当作“顾客”看待，所以他从未想过版本不兼容的话会对其他的组员造成极大的困扰。我看过太多类似的事例，因此，我认为程序设计师都应该有这样的思考能力——把程序的使用者当作“顾客”，去了解他们的心理，使程序更能满足他们。

当 Windows 版的 Excel 小组打算重写一部分的程序代码，使它也能在 Macintosh 上执行时，有一位程序设计师采用键盘操作的方式来做菜单（keyboard-drivenmenu），这是很多企业用户强烈要求，但是在 Macintosh 上却做不到的，对于 Macintosh 使用者来说，这一部分向来是用鼠标来操作的。由于在 Macintosh 上没有标准的“键盘驱动菜单”可以遵循，所以这位程序设计师采用了 Windows 的按键惯例来写键盘驱动菜单，这样的话，程序会最好写，因为这样就不必为 Macintosh 另外设计一套操作的流程和逻辑。他完成了这一部分的程序，就到我的办公室来示范给我看，他的 Macintosh 版键盘驱动菜单和 Windows 版的看起来完全一样，做得非常精彩，我不禁暗中叫好，我操作把玩他的程序时，忍不住发出赞叹。但是当我稍微冷静下来，我想起了一个问题，我问程序设计师道：“我现在不想使用 Windows 界面的话，该怎么做？”

他显得非常纳闷：“你为什么想这样做？用键盘来操纵菜单并不干扰鼠标啊，没有理由非把 Windows 界面移除不可吧。”

他的反应让我惊讶，因为在那个时候，随便找一本苹果计算机方面的杂志，上面都充满了对 Windows 的憎恨，苹果的使用者对于 Windows 的“超人气”非常反胃，他们认为 Windows 不过是个靠模仿起家的三流产品，Macintosh 才是有看头的好东西，只可惜世人眼光多短浅，竟把它当成古怪的玩具。对所有的 Macintosh 迷来说，Windows 是个不折不扣的大坏蛋。

所以，我对这位程序设计师说：“如果我们的 Excel 采用 Windows 习惯的菜单操作方式，势必造成 Macintosh 迷的强烈反感，如果我们把 Macintosh 上的 Excel 做得跟 Windows 完全一样，注定会被杂志批评得体无完肤的。”

这位程序设计师不太愿意修改他的程序，他认为自己已经完成了这一部分的工作，而且他急着去做下一项工作。后来我们找了几个比较有关的程序设计师来讨论这个敌对性的使用者界面问题，结果大家意见一致：Macintosh 版的 Excel 不但要看起来像是 Macintosh 的应用软件，它还要打从骨子里流着苹果的血统，要让别人看着它时脑中浮现苹果那六条鲜艳的颜色。于是这位程序设计师只好回去改他的程序了。

过了一会儿，他从隔间里跑出来，给我看他的最新作品。这次我真的是大惊喜，他不只是为 Windows 式的按键菜单做了开关，他还做出了 Macintosh 式的按键菜单，还有，他做了一个相当聪明的设计，程序在预设情况下以 Macintosh 式的按键菜单出现，使用者可以用热键的方式在 Macintosh 式和 Windows 式之间切换，还不只这些，他甚至把对话框也比照办理，使用者切到 Macintosh 式的按键菜单时，连对话框也一并自动变成 Macintosh 式，使用者切到 Windows 式的按键菜单时，对话框立即自动变为 Windows 式。这回我们可以“通吃”啦。程

程序设计师必须经常以使用者的观点来看自己写的程序，程序设计师必须能体会使用者的感受。

产品的整体观

长久以来，微软的“程序语言部”里，把编译器、除错工具（debugger）、链接器（linker）等等程序语言的软件分别当作完全不同的产品。这样做是符合开发的观点，但以使用者的观点来看则未必合理。因为对于使用者来说，编译器、除错工具、链接器是他同时要用的，是同一个产品，这一点实在显而易见，如果我不说，您可能根本不会想到这些组件是属于不同的部门在制作。

很不幸，在微软的程序语言部并没有这种共识。全世界的程序设计师都在要求改善除错工具，但是除错工具小组没有足够的人力完成这个需求，而同时，编译器小组却兴高彩烈地进行优化（codeoptimizations）设计，而优化设计只有少数使用者需要。他们脑中想的是“我们要不断改进本组的作品”，而不是“我们要不断改进整个产品”。

也正因为如此，微软的链接器长久以来都做得很差，速度非常慢，而竞争者的链接器却强得不得了。微软内的程序设计师全都知道自己的链接器是乌龟在爬，但是没有什么改善的行动。公司虽然派了一位程序设计师来为链接器想点办法，但是他还有其他的职务在身，虽然他已经尽力了，但由于他没有很多时间来改善链接器的速度问题，所以实际作用也有限。除此之外，程序语言部里似乎有种观念，认为编译器才是最重要的，链接器只不过是个辅助工具。但是使用者肯定不会这样想，因为他不会刻意去分编译器或是链接器，对他来说，所有的东西都是同一个产品：微软的××语言开发环境。

微软内部至少就有一组开发部门放弃本公司的链接器，干脆去用竞争者做的链接器；而在应用软件部门，大家也被功能低下的链接器弄得苦不堪言，被逼得宁愿自己另写一个阳春型的链接器，还好用些。最终程序语言的部门开始正视这个问题，改良链接器，让它和编译器搭配成一整体。

最后，经过几次高层管理的人事变动，程序语言部终于获得整合，把自己的产品定位为“程序开发环境”，而不是编译器。结果 VisualC++，一个令世人耳目一新的产品，一个令微软的程序语言部改头换面的产品，才在这样的理念中诞生。

在包装盒里的每一件东西，都是产品的一部分。

重复就是浪费

前两节中我提过的那位程序设计师，在为 Excel 开发“键盘驱动菜单”（keyboard-drivenmenu）时，另一位程序设计师也正在为 Macintosh 版的 Word 写同样的程序，两者相距仅十步之遥。虽然我向 Excel 的程序设计师提过这件事，也分别向 Excel 和 Word 的项目经理商讨过，希望避免性质重复的开发工作，但是没有实际效果。两边的程序设计师仍然分别写自己的键盘驱动菜单，且都正确执行，两个都很完美，但使用者界面就是截然不同。我认为这是非常可惜的,我觉得我们错失了一个让 Excel 和 Word 的使用者界面互相整合的机会，这样可以减少一半的开发时间，也可以建立起一套菜单的函数库，顺便可供其他的应用软件小组使用。因为他们的态度是“这东西不是

本组写的”，所以一定不适用，以致没有人在意我们的程序设计师在做重复的工作，结果是无法产生标准的微软菜单惯例。

我个人对开发工作有一个原则：凡是别人写好、测试通过，而我可以使用的程序，我一定二话不说就把他的程序抓过来用。基于这样的态度，我在开发自己的程序时，总是尽可能为潜在的使用者设想，写出一个别人可以使用的程序。我虽不能把程序做到完全的可移植性，但我尽量做到“让程序代码可以重复使用”（reuse），在其他条件相同时，我永远选择让别人分享我程序的做法。

在 Excel 的最早期版本中，有一位程序设计师做出了一个很棒的功能：打印预览（printpreview），让使用者在打印之前，预先瞧瞧一下大概的样子。这个功能设计得非常具有直效性，可以一页页往前或往后翻，而且连图形也能显示出来，还有整份文件的预览。

这个打印预览受到使用者的极大好评,因此 Macintosh 版的 Word 小组奉命在 Word 里也做一个打印预览的功能。他们做得比 Excel 的打印预览精致得多，相形之下，Excel 的打印预览反而显得粗糙，不够精良。所以我也奉命将 Excel 的打印预览改得更完美，要像 Word 一样好得令人刮目相看。

我的第一个想法是放弃 Excel 的打印预览程序,而把 Word 那一套拿过来用，这样不但减少开发的人时，而且可以使 Excel 和 Word 两种产品的使用者界面更统一。当我把这个想法告诉一位 Word 的程序设计师，他告诉我他们的打印预览程序与 Word 主程序是紧密结合的，只适用于 Word，别的产品恐怕很难引用；他说他可以修改这个打印预览程序，但是他从来没想过我们会想要这个程序，毕竟 Excel 已经有了另一个打印预览程序。真是令人伤心，我不能用他们做得这么棒的打印预览程序。

最后虽然我把 Excel 的打印预览修改得更精致，功能更强，但还是 Word 的效果比较好。更令我沮丧的事情是，由于 Excel 的打印预览具有可移植性，反而其他的应用软件都是使用 Excel 的打印预览程序,而非 Word 的打印预览程序。

我曾说过，撰写新功能最好的方法是引用别人已经写好又测试过的程序，也就是程序代码的重复使用（code-reuse）。虽然大部分的程序设计师都很认同这一点，但是他们潜意识里觉得别人写的程序很难拿来使用。

为了增加程序代码的价值，程序设计师应该培养新的态度，注意自己写的程序是否具有“可再利用性”，要满足这一点，程序设计师就会尽量减少子程序与主程序之间的关连性，让子程序独立，才能被别人使用。这个问题就和避免直接使用 global 变量一样，有时候的确无法避免，但还是有技巧可以排除这个问题。

程序设计师应该随时自问：

这段程序代码是否对别的（或是未来的）程序有用？

或是反过来问：

这段程序代码是否确实只有我一个人需要，别人不太可能引用？

如果“可再利用性”的答案是肯定的，这个程序就应该写成很独立的方式。像键盘驱动菜单或打印预览这类的程序，当初就应该做成独立的、可再利用的程序。也就不会造成现在 Word 和 Excel 得做重复的工作，但没有双倍的效果。

将程序的可共享性当作优先考虑的目标之一，否则程序设计师将经常做重复的工作。

杠杆的效应

如果您的部门想要比别人更成功，就得学会善用“杠杆原理”，就是设法使投入的有限力量得到更大的报酬，也就是事半功倍的意思。每位组员都应该牢记这个原则：

充分利用现有资源或创造新资源，以便从每一项工作中获得更大的价值。程序代码的再利用，就是很好的例子，当然，还有其他的地方可以运用“杠杆原理”。

我在第 6 章提过训练新兵的原则就是一种“杠杆原理”的运用，您让新加入的程序设计师优先学习公司内使用最广的技术，最后再让他接触只有本项目会用到的技术，如此一来，这位程序设计师对公司的价值就会被提升了。对您的项目来说，新加入的程序设计师学习的先后顺序都无所谓，但是万一这位程序设计师被调到其他的小组，这个顺序就很重要了，因为结局只有两种：程序设计师很快进入状况、或是完全重头开始学习。

只要您用心发掘，大部分的工作里都有杠杆存在的。举例来说吧，有一回，我在审核使用者界面函数库时，技术经理拿了一份加强该函数库功能的提案列表，这些功能几乎涵盖了所有小组的需求，看起来很不错。我对他说：

“这看起来很不错，但是某些函数库的界面似乎和 Windows 的不太一样，你有没有参考过 Windows 手册所列的函数库？”

技术经理似乎有点冒火：“马魁尔，这个函数库可不是给 Windows 用的，只要我们的函数库好用，谁在乎 Windows 怎么呼叫函数库，我觉得要我去看 Windows 参考手册是在浪费时间。”

他说得很对，我这才明白我根本没有向他解释过，为什么我认为我们的函数库界面应该和 Windows 的一样。于是我说：

“对不起，我想我懂了。你的意思是说，只要我们的函数库做得好就行了，至于函数库的界面是怎么样似乎是次要的，可以和 Windows 的函数库界面相同，也可以不同，对吧？”

他点点头：“对呀。”

“请让我问你一个问题，现在有 20 个小组在用你的函数库，你想他们会不会调去做 Windows 的软件呢？”

“没错，是有可能。”

“我想也是。那么请你告诉我，那些程序设计师被调去 Windows 的项目时，会很容易学会使用 Windows 的函数库吗？”

他脸上露出恍然大悟的表情：“只要我们把函数库的界面做得和 Windows 的一样，那就非常容易了！等等，你的意思是说，我们顺便教他们 Windows 的程序设计？”

“完全正确，而且你想，这样做公司会需要付出什么额外成本吗？”他想了一下。

“我想，什么都不用，只要我去读 Windows 参考手册就行了。”

“没错。另外有件事也请你回去时顺便想想：你会永远做当这个函数库的技术经理吗？包括你，也可能调到 Windows 的项目呢？”

您可能从来没想过，像“为函数命名”这样简单的事情竟然也能有杠杆

原理，虽然是这样小的事情，却也能导出不少连带的利益。

人们之所以无法创造出新的杠杆，是由于这需要对未来有洞烛先机的远见和很大的信心，相信我，现在您所创造的价值，将来必定有所回报。即使您现在创造的资源未必能在预料中的将来派上用场，但是现在的世界变化这么快，所以，您应该保持乐观和健康的心态，随时为未来做好准备，如果您有各项齐备的资源，需要时便立即可用。我一向相信这个真理：

如果您创造了一项资源，并且让别人知道，那么总有一天会派上用场。

当我开始做 Macintosh 的交叉发展项目（Macintoshcrossdevelopmentproject）时，应用软件和程序语言这两个主要的需求部门都将这个项目视为仅限内部使用，但我的目标却希望将这个发展系统架构在微软已上市的 80x86MASM 产品上，也就是拿 80x86 版的软件来修改，这样做的好处是，我们交叉发展系统就能随着 80x86MASM 做同步改良。这又是一个杠杆的实例，而我打算把它发挥得更淋漓尽致，因为我相信其他公司的程序设计师如果有了我的交叉发展系统，就可以一边写 Windows 的应用软件，一边顺便做出 Macintosh 版。当时大部分的人都认为我疯了，但是我非常地肯定，交叉发展系统不应该只限微软内部使用，而应该是个可以上市的产品，如果我们不在一开始时就认清这一点，将来我们的决策都很可能方向错误。

所以我在设计会议中常常主张，某个方法确实适合内部使用，但如果我们想把交叉发展系统当作正式的产品，就不应该这么做。

“但是我们本来就没把它当成正式产品啊。”有人会这么说。

我会说：“没错，如果你从来没想把它当成正式产品，它就永远不会是。但是如果它的设计和品质够好，市场上有需要，为什么它不能既当作内部使用、也同时被当成产品卖？”

事实证明，大部分的情况是我们不但能够想出一举两得的方法，还能做出更好的设计、花更少的时间去开发。目标提高迫使我们必须思考得多一点、难一点，有时候反而想出了更好的设计方式。也有些时候，我们会发现如果仅供内部使用的话比较容易开发，坚持把它做成产品的话时间上会花得太多，那我们就暂时做个内部使用版，等将来要将它商品化时再来修改这一部分。

每当上级主管关心我们的项目状况，我除了演示文稿项目进度外，都会向他们说明我的策略是尽量以将来要商品化为目标。上级主管唯一关心的，也是我所认同的，就是不要花时间去做将来用不到的东西。

没有人相信交叉发展系统终有一天会商品化。但这一天还是来了，微软宣布了“四海之内皆视窗”（WindowsEverywhere）的计划，就在刹那间，我的项目就成了微软在非 80x86 平台的 Windows 解决方案，Macintosh 的交叉发展系统正式宣布将成为一项产品，并列入非常优先的地位，公司加派更多的程序设计师进来。

从您的每件工作中创造最大的资源，不管是利用现有的杠杆，或是创造新的杠杆。

善用资源的态度

我一直鼓励您在任何时候，都尽量以创造资源的态度做事，这是我在本章中所倡导的观念。只要稍微改变一下态度：不是把上头交办的工作完成就算了，而是去思考哪里可以产生杠杆，您的工作成果是不是可以创造出更有用的资源，这种投资报酬率比我所知道的任何一种训练方法都高。

持续不断的改善是件好事，这样做的话，您就可以经常保持领先竞争对手的优势，但是若希望您的小组更上一层楼，就必须培养他们自动自发的心态。前面说过的那位技术经理，一开始不愿意读 Windows 函数库手册，只不过是因为我没有解释理由，一旦他了解我的动机是创造资源，他一点都不需要我催促他看 Windows 函数库手册，他已经有了自动自发的态度。

重点提示

新进的程序设计师必须了解，写出“零错误程序”并不是容易的事，如果他们有这种认知，就不会轻易坚持自己的程序已经完成，没有错虫。有经验的程序设计师知道，写出“零错误程序”很困难，但是并非不可能，那是需要多下点功夫才能做到的，程序设计师应该在把程序送交测试小组之前，彻底用除错工具追踪过程序的执行。由于写“零错误程序”是这么困难，有错虫的程序一旦被置入软件，那就会造成极大的损失，要大量的时间、人力才能大海捞针似地挖出这个错虫，所以程序设计师务必审慎再审慎，用一切的办法侦测和预防错误，即使要自己改变程序风格也无妨。

小心那种“太难了”、“太花时间”或是“太麻烦”的反射性反应。当您遇到别人有这种反应，请先问自己他有没有认真思考过这件事的重要性、以及是否符合项目目标，如果您认为他其实未经深思熟虑，只是直觉的反应，那您就应该把您的想法告诉他，请他重新评估，也许就会有公平的答案。

人们遇到经验范围之外的事情，多少有恐惧感，就会认为“这完全不可能”而强烈反对。试着消除这种习惯性的反应，设法给组员灌输“只要花时间想想看，大部分的事情都做得到”的观念。您不妨以这个问题来对付那种“凡事不能”的态度：“我了解这是做不到的，但是‘如果’做得到，那你会怎么做？”然后您就会发现惊人的转变，您马上就会听到组员七嘴八舌地说应该这样做、那样做，说的是他们刚刚坚持做不到的事情。这个“如果”把他们带离直觉的反应，带到全新的思考模式，这才是他们应该做的。

把使用者当作什么都不懂的外行人，是非常不好的观念。每当您发现有人表露出这种心理，一定要立即纠正，提醒他们使用者才是真正受产品好坏影响最深的人，他们和程序设计师一样关心软件的执行速度和质量。

教导程序设计师以使用者的角度来看产品。程序设计师必须对产品有整体性的认知，包装盒内一切有形无形的东西都是产品的一部分，使用者并不在乎其中一部分好或不好，也不会想知道里面有多少不同的小组各负责几个组件，也不在乎究竟用什么语言写成，他不在乎软件是怎么做出来的，这只对软件公司有意义，他们只知道产品是那家公司出的。程序设计师当然不会每个程序都参与，但是他们必须了解产品是一个整体，任何一部分不符品质标准，都得研究对策，而不是只做自己被分配到的程序。只要大家都关心产品中比较弱的组件，自然那一部分就会被设法改善。杠杆原理是您最有用的观念，找到您工作中的杠杆，您可以为小组、项目、公司、甚至软件业创造无可限量的价值。无论如何，尽量利用资源并创造资源，这个原则是绝对错不了的。在您写程序的时候注意程序代码的共享性、训练组员的时候注意到他对公司的价值，即使是像函数命名这种小事，都有杠杆的存在。不管做任何事，都要想到“善用资源”，为未来做好准备。

第 8 章　沉船的感觉
Chapter Eight

当项目的进度开始落后时，主管第一个想到的策略不外是：增加人手，或是要求组员加班。看起来似乎很合理，但是对于扭转项目的劣势来说，是最差的对策。

让我们想象一下 16 世纪，有一艘大商船正横越大西洋，朝向新世界展开冒险的旅程。商船航行在大洋中时，大副发现船进水了，立刻向船长报告，于船长命令船员将船内的水舀出去，但是不论船员如何努力，水还是渐渐升高。船长眼看情形不妙，又叫来更多的船员一起舀水，依然没有用，这下船长慌了，只好命令全体船员以轮班的方式来舀水，但海水还是继续渗进船里，船渐渐往下沉……

这时船长也知道他已经用尽了所有的人力，可是船还是不断地进水，所以他只好下令所有的船员日夜不停地舀水，累倒了之后醒来，再继续工作，这下真的发挥了作用，船果真不再往下沉，进水线降低了。船长因为这次的成功而沾沾自喜，想必是自己对人力资源运用得宜的缘故。好歹一个星期是撑过去了。

但是很快地，船员就因为过度劳累而舀不动水了，舀出去的水愈来愈少，船又下度下沉。大副试着说服船长，如果要船员工作有效率，就必须让他们休息。但是因为船仍然在往下沉，所以船长拒绝讨论让船员休息的事。船长咆哮着：“ 我们快沉船了，所以船员不准休息，在船没有停止下沉以前，没有人可以休息，你没看见吗？我们就要沉船了！！ ”

进水线仍然残酷地升高，愈来愈高，于是船终于沉入海底！

大家不妨想一想，如果您是船长，除了命令所有的人拼命舀水，有没有更好的办法呢？如果您正在一艘漏水的船上,四周都是一片汪洋,您怎么办？我的想法很简单，我可以告诉您我怎么办：“ 找出漏水的地方，把它堵上。 ”您不也是这么做的吗？

这实在是显而易见的道理，但是为什么总有那么多项目经理不懂得用这一招来挽救他们的项目呢？我只看见一大堆项目经理一遇到进度落后就二话不说地要求组员加班，组员加班若还是解决不了问题，就寻求更多的人手投入。就像那位船长，水都到腰际了，仍然不肯停止舀水、先去“ 找漏 ”、“ 补漏 ”，直到所有的人跟着陪葬。所幸，大部分的项目还不至于沉船，因为组员舀水的速度，勉强可以比进水的速度快一些，所以项目最后大都是硬干蛮干也把它赶出来了，只不过，浪费了无数的人力，折损了无数程序设计师的青春血汗。

在第 1 章我谈过的那个使用者界面函数库小组，他们一年来每周工作 80 小时，但项目看起来一点进展都没有，情况只是愈来愈糟，却没有人想到停下来找出问题症结，一味埋头继续苦干，祈祷自己的辛勤能感动上苍，他们一天工作 12 小时,每周工作 7 天,他们还能怎么样？就像我在第 1 章描述的，这个小组把大部分的时间都花在他们不应该做的工作上，他们并没有认清项目的目标是：“ 提供对每一个需求小组都有益的使用者界面函数 ”，这就是漏水的地方，问题的症结所在。

在第 3 章中，我也举过另一个对话框函数库的例子，他为了 Word 小组而改写程序代码，企图加强执行速度，却始终达不到 Word 小组的要求；而事实

上根本不是对话框程序不够快的问题，而是由于 Word 本身的置换（swapping）功能会把所有暂时不用的程序代码移出内存，要用时再重新加载（reload），是这段重新加载的时间使得对话框看起来很慢，和本身的速度无关。但是当时却没有人注意到这个问题，双方都把焦点放在对话框的执行速度，要求将对话框的程序代码优化。

在第 5 章，我也提过 Excel 小组曾经每周工作 80 小时，试图赶上那不合理的日程表。

在这些事例中，“ 加班 ”本身就是一个危险讯号，明白告诉您一定有什么地方出了问题，很不幸地，大部分的主管都不明白这一点，而忽略了这个严重的警讯，反而在项目发生进度落后时，直觉地采取两个最愚蠢的步骤：增加人手，或是要求组员加更多班，而不是去找进度落后的原因。

过正常的生活

我在微软的七年，最重要的工作是让那些挣扎中的项目起死回生。在每一个案例中，小组的成员都是每周工作七天，天天加班，只为了赶上预计的完成日期，团队的士气非常低落，程序设计师开始憎恨自己的工作。

每次我正式进入一个小组成为他们的新主管，我做的第一件事就是停止加班，并寻找进度落后的真正原因。我会在每天的傍晚，到程序设计师的办公室把他们都赶回家去：“ 赶快下班去！过个像人的生活！ ”

程序设计师会抗议道：“ 我不能走，我的进度落后了。 ”

我会说：“ 没关系的。整个小组都已经加班工作了整整一年，仍然无法使项目赶上进度，可见加班绝不是解决问题的方法。一定有其他从根本上就出错的问题，我们必须把问题找出来，再针对它改正。加班是没有用的。回家吧！好好休息一下。我们明天第一件事就是找到出错的地方。 ”

一开始大家都以为我在开玩笑。这种话似乎从来没听主管说过，主管们向来是要求他们再多一点努力、再多一点加班，现在太阳还没下山呢，而我竟然叫他们回家，他们以为我疯了。项目目前的进度已然严重落后，如果大家都停止加班的话，后果岂不是会更糟？

但是接下来的几个星期，我会用本书前面所介绍过的策略来改善项目的体制。我会停止不必要的报告和会议，我会排除不必要的杂务；我会把条例式的工作进度表丢掉，换成我在第 5 章建议的阶段式小项目来取代；我会为组员建立第 7 章所述的正确心态，例如一发现错虫就立刻清除；我把项目的目标定得非常清楚，并且让程序设计师明白，我身为主管的目标之一就是保护他们有一大段完整、不受干扰的时间，专心去开发程序。大约经过一两个月的辛苦，小组学会了正确的方式，能够完成小项目的目标了，大家打胜了第一仗，而且不必每周工作 80 小时，小组开始有信心，接下来的几个月，大家已经很能掌握正确的工作方式，把我传授给他们的技巧变成工作的习惯，渐渐地，愈来愈容易完成各个小项目，而最后整体进度终于得以完成。

如果进度发生落后，那表示有个地方出错了。

您应该找出问题，并加以解决，不要一味要求组员加班，在问题没有解决之前，加班是没有用的。

鞠躬尽瘁的迷惑

有的小组加班的原因不是为了赶进度，而是因为高层主管们认为程序设计师不加班的话是无法如期完成产品的，因此他们非加班不可，每周工作必须超过 80 小时才行。这种笃信加班的主管看到有某个小组每周“才”工作 40 小时而已，就会以为这个小组对公司没有尽心尽力。也许高层管理者认为如果小组每周工作 40 小时就能完成进度，那一定是进度表定得太宽松了，他会把每周工作 80 小时的小组拿来作例子，说道：“这样才是对公司鞠躬尽瘁的表现！”要求所有的小组也每周工作 80 小时，不然的话就是对公司不够鞠躬尽瘁。我当然完全不同意这种看法。如果我赞成这种论调，前面所说过的那些故事的主角：使用者界面函数库、对话框函数库、Excel 的部门，就会变成应该模仿的英雄偶像，而那些拥有具体目标、不断学习、坚持工作效率的不加班团队，不就应该经常进度落后了？事实正好相反。

我这样说，读者一定觉得很蠢，但是且听一位高层主管对不加班的小组说的话：“公司雇用他们并不是让他们在数时间混日子的，告诉他们工作时间必须长一点，我要看到他们为公司尽心尽力！

真是没道理，高层管理者竟然把没有效率的小组拿来表扬，而把有效率的小组说成愉懒。还好有另一种高层管理者，当他看到有一个小组不加班却能完成工作，就会去请教这个小组如何做到的，并要求其他的小组效法。

为什么同一件事却有截然不同的反应呢？简单的说，就是观念问题。

同样是不加班而能完成进度的情况，有的管理者认为是罪恶，有的管理者认为是典范。聪明的读者，您觉得如何呢？当然在某些特例之中，这两种管理者可能都是错的，但是我们应该从正面去思考这个问题，对吗？

就像是有些项目经理在遇到进度落后时，第一个反应是要求组员加班，而不是寻找哪里出了问题，有些高层主管也是一样的直觉式反应，因为他们在观念上相信加班对项目有好处，对公司勤奋的文化也有好处。他们都忘了一件事，在软件业，开发小组贡献给公司的是智能的价值，这是重质不重量的，有很多比加班更好的方式增加小组对公司的贡献：像是提高产品的质量（却不一定以投入更多的工时为代价）、写出可共享的程序代码供别的小组利用等等。但是对于笃信加班神话的主管来说，他只在意程序设计师是否把自己所有清醒的时间加上部分的睡眠时间完全付出给公司，以为这样就可以增加开发小组对公司的贡献，这是完全错误的。

“加班加得愈多，代表总投入工作时数愈多，产品就能愈快完成”，在理论上看来似乎没什么不对，在传统制造业或许成立，但是在软件业是行不通的。软件业的投入工时和产出价值没有必然的正比，多投入三小时，未必得到三小时的工作成果。总之，软件开发项目的问题，绝不应该是靠着加班来解决的。

不要责怪程序设计师

虽然我一再把使用者界面函数库、对话框和早期的 Excel 几个项目当作错误示范，但事实上这些项目的问题都不在程序设计师。大部分的时候程序设计师工作非常辛苦，在各种令人沮丧的环境中仍然得尽最大的努力。但是在项目进行不顺利时，程序设计师很容易变成各方责怪的对象，这是不对的，项目有问题的话，整个部门都要检讨，也就是说，这是管理上的问题。

如果高层管理者强行规定程序设计师每天必须工作 12 个小时，从早上 10 点做到晚上 10 点，那么程序设计师待在公司的时间确实多出了 3 个小时，

但是这多出来的时间是用来做什么呢？

我们来仔细分析一下程序设计师的一天：每天在公司的时间是 12 个小时，扣掉 1 小时午餐和 1 小时晚餐，因为 10 点回去的话所有的餐厅和家里的厨房早打烊了。因为每天都得在公司待足 12 个小时，所以自然有些私人的活动必须加进来，再扣掉运动 1 小时，也许是慢跑或上健身房，这样就剩下 9 个小时了。

还有许多私人的琐事因为下班后没时间做，只得在办公室里处理，我曾经看过程序设计师在整理一堆帐单，签支票、粘信封等等，我也见过程序设计师拿着他们放在办公室的电子琴键盘练习钢琴指法，甚至有的程序设计师在大厅里和其他的组员一起玩，从回力球到室内高尔夫，什么都有。

每天工作 12 小时的人，他的工作量和正常工作时间（早上 8 点到下午 5 点）的人比起来，不见得比较多。虽然他在公司多待的 3 个小时里，会完成某些正经事，让管理者误以为生产能力有所提升，但其实这只不过是补晚餐和白天处理私事的时间而已。

当然也有些时候，程序设计师留在公司是因为看到一个错虫，不除之而后快的话，晚上会睡不着觉，或是就快完成一条程序了，没写完的话舍不得走，这种加班是有生产效率的。但是请注意这是来自程序设计师个人的意愿，不是上级主管的压力，而且这种情形是属于偶发现象，并不是常态。

项目经理的工作之一是保护小组不受“ 加班迷惑 ”的伤害。我想许多公司多少都会有笃信“ 不加班就是不够努力 ”的高层主管，要说服他们并不容易，因为他们职位比较高，但是您一定要站稳立场，向他们解释加班对项目没有帮助，所以这样的命令还是请上级收回吧。这种时候，项目经理处于两面不是人的为难状况，要不就是对上级抗命，要不就是把压力往属下身上掼。如果是我，我宁愿冒着得罪老板的危险，也不会要求组员做一件我根本就反对的事情。所幸，我从来不至于因此而与高层主管开战。不论是微软或其他我呆过的公司，我都还没遇过对加班着迷不已的高层主管，基本上，他们都相信加班对于生产能力和效率并没有提高的作用。

别误信加班等于增加生产能力，长期的加班只会伤害生产能力，对项目没有帮助。

成功的人不都是拼命工作的吗？

我这样强力主张不加班，也许有少数人会不以为然：“ 那些极为成功的人，不是每天都拼命工作，才得以完成梦想吗？如果他们坚持不加班，怎么能有如今的伟大成就呢？ ”

可是如果您再深入探究他们成功的原因，您就会发现，那并不是由于他们拼命工作的缘故，至少不是主要的因素。成功的人和一般人最大的不同是他们心中有一个理想，为了早日实现它，心中有一股内在的动力使他们拼命工作，而不是因为上级或公司的规定而必须日夜加班。有更多的人日以继夜工作却一事无成，只是我们不会津津乐道他们的故事。拼命的工作并不是成功的关键，成功的关键是有一个明确的目标，具体而切合实际的计划，以及每天不断向这个目标迈进。

周末悍将

也许您已经成功说服了高层管理者，强迫组员加班对于生产能力没有任何帮助，应该设法让组员把上班时间利用得更有效率才是正途。但是上级也可能用您的话来反驳您：“ 好啊，你说你的小组不必每天工作 12 小时，也能

维持工作效率，很好，但我要他们周末来上班总行了吧，你别跟我说他们周末来工作不会提高生产能力。”有时候这种论点是对的，特别是组员的工作效率本来就不错，而他们的私人时间也很空闲的话，请他们周末加班确实能提高生产能力，至少短时间内不会有问题。

但是，高层主管必须明白，要求组员在周末工作可能造成小组与管理阶层之间的关系恶化。因为组员知道周末应该是私人的，不是公司的。所以他们被迫工作的周末愈多，就会愈憎恨公司这样利用他们的劳动力，因而降低对公司的向心力，于是开始有人离职，替补的程序设计师对项目并不那么熟悉，也比较缺乏经验，当然生产能力一时跟不上来，公司因此蒙受损失，这种损失往往超过组员周末加班所挣得的利益。您想当项目完成时已经损失了 1/4 的程序设计师，那对团队而言是多么大的伤害！这种事情真的发生过，但那些短视的高层主管反而拍手称庆：“很好，爱抱怨和工作不卖力的人走掉了。”

我还听过另一种论点，是说软件业竞争如此激烈，倘若公司希望维持竞争优势，开发小组就“必须”在晚上和周末加班。这个“必须”也是您应该警觉的字眼。开发小组“必须”在晚上和周末加班，这是“除非开发小组加班，否则我们无法打败竞争对手”的另一种说法罢了，这种观念对吗？除了让开发小组加班之外，难道我们就没有更聪明的策略了吗？难道打败竞争对手的责任，就落在开发小组的加班上吗？我很希望本书能为软件业带来正确的观念：有许多很好的方法让我们事半功倍，这才是我们努力的方向。

周末是属于组员私人的时间，不是公司的。

公司不应该以打败竞争对手为理由，要求员工周末加班。

新兵训练

有些人坚持开发人员必须长时间工作，除了提高生产力之外还有一种理由：就是每天长时间的相处有助于培养团队那种生死与共的感觉，特别是新加入的程序设计师应该受过这种磨练，使他们以身为团队的一份子为荣。

我们假定这种论点是对的，长时间的相处确实有助于培养团队一体的感觉，但是，长时间工作就是训练新兵最好的方法吗？

各位都知道，在程序设计的工作中，“思考”才是最重要的，所以，为什么一定要这么重视长时间的工作呢？难道训练新兵用心思考不是更重要的事吗？一位新人需要学习的是认真、聪明、有效的思考。在设计阶段，他们要想得很慎密，确定这样的设计没有疏漏，写程序时要用脑筋，要懂得思考怎么测试这个程序才能确保没有错虫；一位新进程序设计师在遇到错虫时，更要懂得不是胡猜瞎猜，要思考如何有系统地搜寻错虫藏身之处，要学会判断是否有相关的错虫尚未现身。他不仅要学习对付错虫，还得思考如何在一开始写程序时就防范它的发生。同时要了解这一行里新知识是不断蜂涌而至的，他必须不断地阅读、学习，才能跟得上产业的脚步，并且积极地提高个人的技术层次。

这些都是很艰难的训练，很不容易学习和遵循，真的很难，因为每一项训练都要靠脑力，而且要练到精熟才行。这些才是新兵训练应该有的课程，长时间的工作绝对不是重点，因为工作时间的长短和程序的好坏是无关的。

强调思考的重要性，而不是长时间工作。

我会拿不到奖金的！

当我在下班时间吆喝程序设计师离开办公室时，有些人会抗辩：“那奖金怎么办？我要是不加班，年终考核时就拿不到奖金了。”

我会向他们说明，我从来不把加班与否当作工作奖金的考核依据。超时工作对我来说，只是团队出问题的显示，我要改正它，而不是把它当成对程序设计师的奖励指导。

我接着告诉程序设计师：“如果你想拿大笔的奖金，就去想办法让我们的产品做得又快又好。或者，找出我们在那里浪费了人力、有什么地方我们可以引用别组的程序代码。或者，你有什么自动测试程序的好主意、更好的侦错方法，请尽快提出来。或者，你知道市面上有我们急需的工具，那就更棒了。或者，你想到了一个更能符合使用者直觉的操作界面，那非常好，我希望你的点子可以和产品充分结合。想拿奖金吗？照我的话去做，这些并不需要加班的。”

我继续说：“想大幅加薪吗？赶快积极学习新的技术、养成良好的工作习惯，做事更有效率，把握有限的时间，增加你个人对公司的价值。想成为微软之星吗？那你最好养成每年都拿高额奖金的习惯，就是不断想出让产品做得更快更好的方法，有这种习惯的人每年都能拿到高额奖金和大幅调薪。”

我希望程序设计师工作表现更好，而不是工作时间加长。

扭转乾坤

您的小组长久以来每天都在加班，而您突然决定要停止这种现象，把他们从几乎窒息的工作中拉出来，去解决真正的问题，您最好先有心理准备：刚开始的时候，几乎没有人能把工作完成，这是挺可怕的，但却是让团队起死回生的必然过程。就像人并不是天生就会念书，程序设计师也不是天生就懂得如何有效率地工作，这些技巧必须经过学习和培养，所以，程序设计师马上要开始进行工作训练。

若是程序设计师在每周工作 40 小时的情况下，无法完成合理的工作量，而且我确定的日程表并未过度乐观，我就会要求他把某一天的活动记录下来，分析一下他个人在时间利用方面该如何改进：

为人事部门进行面试，并写报告。

与 CodeView 小组的某位程序设计师闲聊 30 分钟。

浏览 comp.lang.c 和 comp.lang.c++等新闻讨论区的最新消息。

阅读《个人电脑周刊》（PCweek）杂志。

午休两小时，吃午餐和处理一些个人杂事。

审阅使用手册的部分草稿。

参加另一个小组的项目进度会报，并且报告他们想知道的某一项功能的开发进度。

在健身房里玩 30 分钟的回力球。

阅读 27 件 e-mail，并回复其中的 15 件。

这就是他在办公室里七到八小时的活动内容，没有写一行程序。我在开玩笑吗？这是特例吗？都不是，根据我的经验，每天工作 12 小时的程序设计师常见的活动就是这样。

当然程序设计师不是每天都看《个人电脑周刊》，但是整个一星期里他每天都在看这看那，也许是公司的业务通讯，或是他自己订阅的 InfoWorld、MSJ、PCmagazine、WindowsSources 以及 SoftwareDevelopment 等等。电子

邮件也经常打断他的工作，每周还要协助人事部门进行一两次的面试，并浏览网络上的讨论区，还要午休、办杂事和运动。

弹性上班？

微软和很多高科技公司一样，采取弹性上班制。您可以在任何时间到办公室，也可以随您工作多久。这就是为什么程序设计师可以午休两小时，再玩个回力球，却一点儿也不会良心不安的缘故。在规矩严格的公司里，这种行为足以让您被革职，但是在微软不会，只要您将工作完成就行了。

弹性上班多棒啊！如果您与牙医师有个约诊，不必请假，如果您女儿参加学校的话剧公演，您尽管去观赏，如果您碰巧是个棒球迷，想看下午的现场球赛，跳进自己的车里开走就行了。弹性上班制可以大幅提高员工的生活品质，因为每个人可以完全针对个人需要，安排自己的时间表。

但是弹性上班也有缺点，这是人事部门在招募人才时不会提到的。依照弹性上班的定义，员工既然没有固定的工作时间，因此考评程序设计师是否认真工作的办法，就是看他有没有如期完成工作，如果您再深入想想这一点就会了解，一旦有某位程序设计师落后进度，就表示他工作不卖力。当然，没有人会立刻告诉他这一点，但是上级会很自然地希望他把工作完成再下班，他们根本不管程序设计师其实已经辛劳工作了一整天。

如果您发现程序设计师必须延长工作时间才能完成工作，就表示有问题。也许这位程序设计师滥用弹性上班制来掩饰工作的拖延，也许是小组发生了我在前面几章讨论过的各种毛病。总之，不要忽略问题，它的确存在。

对于一位每天工作 12 小时的程序设计师而言，这样的一日活动是有理由的，不然他什么时候去办私事和看杂志呢？这就是高层管理者忽略的一点。他们强迫程序设计师超时工作，结果程序设计师为了适应这项要求，不得不把私人活动排进上班时间。

我让程序设计师把一天的活动用白纸黑字写清楚后，我开始问他们一些问题。“如果你从现在起每天五点下班，你会需要两小时来午休和处理私事吗？你会让电子邮件随到随处理，或是固定时间一并处理？如果杂志留到回家时看，你觉得怎么样？和别的小组谈某一项功能的进度问题，是否可以交给项目经理？”

我会和程序设计师共同研讨出一份合理的活动表，让他在上班时间内完成份内的事，并且准时下班。这并不太困难，只是看主管怎么做罢了。

训练开发小组懂得在正常工作时间内掌握好工作的效率，不要让他们超时工作，因为超时工作只是浪费时间的假面具。

我无法在白天工作

程序设计师常常抱怨无法在白天工作，看看前面那位的活动日志就很清楚，这种抱怨绝非空穴来风。程序设计师必须协助面试、阅读和回复 e-mail、审阅使用手册等等，这些都是正当的公事，程序设计师无法将它们推掉。

问题在于这些正当的公事扰乱了程序设计师真正的工作——开发产品。例如电子邮件随到随处理，程序设计师的时间就被切割得太零散，没有一个比较长的时段让程序一气呵成，也没有办法对处理次要公事的时间做妥善的规划，如果每一件次要的正当公事都是一放到程序设计师桌上就立刻处理，难怪程序设计师永远没有时间写程序了。

我曾经听过许多管理者建议，事情一来就立刻处理，要不就是马上解决

掉，要不就是不做任何处理，把它打发掉；这样才能保证绝不拖延，并且把注意力保持集中在最优先的事情上面。基本上我同意这样的观点，但是对于程序设计师而言需要一点小小的修正，程序设计师如果一味地盲从于这个建议，每天的干扰仍然使他们太过分心而无法在白天工作，而要等到晚上夜深人静时才能写程序。

上述的管理者的建议，重点是在事情一发生便“即刻处理”，而程序设计师虽然无法预料事情什么时候会发生，还是能设法避免受其干扰，e-mail 就是最佳例子。程序设计师不必随时查看邮件信箱，可以把所有的 e-mail 集中在一天的某一个时段（或是两三个时段）处理，把它变成每天固定时间的工作，而且不妨挑选自己工作效率比较差的时段来看 e-mail，看完 e-mail 就立刻决定要处理（回复邮件）或不处理（删除邮件）。

也就是说，程序设计师应该利用 e-mail 作为缓冲区，让临时突发的次要公事全都暂时搁在邮件信箱，到了固定时间再一起处理，处理的时候要快刀斩乱麻，绝不拖延，立刻解决。

同样的道理，也可以运用在其他不定时出现的公务上，使它变成可预期的工作，不再打搅正在开发的程序工作。程序设计师应该拟出一份时间表，安排最优先的工作，避免让随时插队的工作喧宾夺主。以我的每日时间表为例，看看一天的时间应该如何分配：

仅专心进行开发工作（3.5 小时）
午餐并稍微休息（45 分钟）
第一次阅读并处理 e-mail（15 分钟）
继续努力进行开发工作（2 小时）
第二次阅读并处理 e-mail（10 分钟）
处理突发的其他工作（1.5 小时）
第三次阅读并处理 e-mail（10 分钟）

我将午餐前的时间，也就是一天工作效率最高的时间，完全专注在开发工作上，这段时间我尽量不接电话，绝不打开 e-mail 信箱，因为那是最会令人分心的，我将这三到四小时的时间完全只用来写程序。用完午餐后我才会去看 e-mail。

在我处理完第一批电子邮件后，我再度投入开发工作，这时若有突发的工作进来，我会先置之不理，直到我排定处理次要公务的时间，我再决定是立即处理或不予处理。如果我今天实在无法把这些次要公务完全解决，我就留到明天同一时间继续。

您看，利用这样的时间安排方式，e-mail 和一般的杂务就不会打搅到我的开发工作了。我的诀窍就是在我排定的时间内处理这些次要公务，而不是让这些次要公务来决定我的时间表。也就是我把无法预料何时来临的事情，留到一定的时间内处理，这样我就有足够的时间而且能专心地从事开发工作了。

很不幸的是大部分的程序设计师都让他们的时间表被各式不定期出现的杂音给弄乱了，他们没有办法把程序放在第一位，反而被突然跳出的事件搞得团团转，因此直到下班根本没碰过一行程序（正经事），却处理了大量的：电子邮件、会议、报告（杂事）。若是照这种做法，怎么能不加班呢？不加班就无法完成产品了。

如果您很确定项目的日程表是合理的，并没有过度乐观之嫌，而程序设

计师却总是超时工作，那么可以确定必然有问题，您最好检查一下这些可能的根源：

◆程序设计师没把临时出现的事情安排在固定的时间，而占掉了写程序所需要的时间和专心。

◆程序设计师让次要公务抢占了比写程序更优先的处理顺序。

前述的时间表对我个人非常适用，当然并不是每个人都得和我一样，我的时间表对某些人来说可能不太适合：“为什么一定要在午餐之后才能看 e-mail？我可没办法，我就是喜欢一清早来看 e-mail。”当然，如果您的工作是以阅读 e-mail 为主，那大可不必规定自己什么时候看 e-mail，但程序设计师最主要的工作是开发产品，所以，任何时间表的安排，最大的原则都应该是：让程序设计师能够专注于开发产品。您不妨把新邮件到达时的提示声音关掉。

与程序设计师共同研拟出一份每日活动的时间表，把无法预期的临时公务变成固定时间处理的事情，并且把程序开发的工作放在最优先的地位，不要让其他次要的事情干扰到写程序。

仅专注于开发工作？

我所谓的开发工作是比较广义的，不是只有关在办公室里写程序、其他啥事也不做。开发工作除了写程序、测程序外，还包括与其他的程序设计师头脑风暴、讨论、程序的审核等都算在内。

激情过后

在某些特别的情况下，加班是有道理的，比方说在期限前的那个周末对所有的程序再做一次彻底的审视，或是在 COMEX 商展前的几天全力准备一个别出心裁的示范程序等等。但是我强调这是“短期”的、特例的，而不是常态的。加班只能在刚开始的一两个星期增加生产能力，因为那时候组员有强烈的危机感。如果您要求组员每周工作 80 小时，他们一开始时会努力工作，但是危机感不久后就会消失，每日活动表的内容就会流于公私不分，反而变成正常该有的生产效率都打了折扣：午休两小时、在大厅里聊天等等，就像我在前一节说的那样。

还有一种很特别的例外情况，就是组员对于项目非常投入，因为这个项目令组员兴奋异常，使他工作做得意犹未尽，下了班还舍不得离开。这种情况看在项目经理眼中当然是令人欣慰的，这种组员无论在吃饭、睡觉或呼吸时都在想着项目。我希望每位程序设计师都至少有一次这样的经历，做的项目不只是公司的产品，同时也是自己的梦想。但这毕竟不能经常发生的，您不太可能经常遇到宝贵的项目，如果是，那就不稀奇了，兴奋的感觉不可能占生活的大部分，所以我对这一种特殊的情况，倾向比较保留的意见。

在我早期的职业生涯中，大约有五年的时间，做了无数的项目，都是令我兴奋不已，使我除了吃饭睡觉之外，几乎全在写程序，和我同组的程序设计师们也都如此，我们没有什么休闲和社交生活。我们整天和计算机泡在一起，生命全部的意义只有程序，每天工作到凌晨两三点，回家睡六七个小时再继续写程序。而我们喜爱这样的日子，我们燃烧着对软件的热情，渴望早日见到作品完成的样子。

做过那些项目之后，我后来的项目虽然一样令人兴奋，但是我不再让程

序填满自己全部的生活。我每天工作八小时，留下时间来追求生命中其他的美好事物——参加宴会、40 英里（1 英里=1.6093 公里）的脚踏车越野赛、观赏歌剧、认识新朋友……我觉得自己像是重生了一样。如果从前有人告诉我每天只顾着写程序会错失生命中的许多美好事物，我一定会大笑，就好象您劝一位老顽固更换速度快 100 倍的计算机时他也会大笑：“ 我现在就用得这么愉快，干嘛换什么机器？ ”但是那天他的老计算机坏掉而不得不更换时，态度就截然不同了，他会瞪大眼睛，脸上出现难以置信的表情：“ 我的天，我竟然到现在才换计算机，我真不敢想像我怎么会对以前那部老古董感到满意！ ”

我就像这种使用者，我不知道我那根筋不对，那么长的时间里除了写程序以外什么社交生活也没有，而我并不感到任何缺憾，可是等到我在工作之外同时也过正常人的生活之后，我才体会到均衡生活的重要。对正常生活的追求欲望也使我更积极利用上班时间的每一分钟，在上班时间内把所有想做的事全做完，才有时间过很充实的私生活，两边都兼顾得很好。

现在，我回想起从前那段完全只有程序的日子，我真希望有人能令我醍醐灌顶，告诉我生活不是只有工作而已。也许当时的我听不进去这种话，但我还是希望有人试着对我说过。因此，现在我如果看到组员像从前的我一样废寝忘食，我会对他们说：“ 回家去吧，好好享受你的人生！ ”

重点提示

经常加班就是项目出问题的明显信号，也许是因为主管的观念错误或是目标不够清楚，不论是什么原因，项目经理绝对不能忽视这种现象，要对这个问题正确处理，项目经理必须协助组员在每周 40 小时的工作时间里，把事情做得更有效率。

我经常听到高层主管称赞组员每天为公司工作很长的时间：“ 您对公司的贡献值得嘉奖，做得很好！ ”这绝对是错误的信息，员工应该是因为工作做得好而受到称赞，而不是因为在办公室待得久，管理者不该把“ 生产效率 ”和“ 工作时间 ”混为一谈，有的人也许可以用更少的时间，完成两倍的工作呢。

为了让组员把办公时间用在正确的地方，并提高部门的工作效率，项目经理不但要为他们排除任何不必要的会议、报告和杂事，还要协助他们好好运用宝贵的上班时间。您应该协助组员安排适当的每日活动表，用一些小技巧，让组员有长段又不受干扰的时间，适合进行开发工作。

如果您关心组员的生活，就不要让他们把全部的时间都投入在工作。每天只要确定他们卖力工作了八小时，就可以把他们赶出办公室了，当然这样做也许会有老板看不顺眼，但是如果您像我一样相信均衡、健康的生活是一切创意的原动力，就坚持这份理念吧！

每周工作 40 小时并不是金科玉律，只不过是美国的传统，而软件开发项目大都以此为前提编定日程表。所以如果一个项目需要程序设计师每周工作 40 小时以上才能赶上进度，就表示有问题，也许是日程表定得太乐观，也许是程序设计师需要再训练。不管怎么说，这个问题一定要认真解决，绝对不应该让程序设计师加班来弥补这个漏洞。

给领导者的话

偶尔，我会听到某一位经理说，当了经理后，就再也不是小组的一员了，和其他的组员就会有些距离，这是无法改变的事实。以我的经验来说，事实并非如此。我曾经加入过数十个小组，不论是担任程序设计师或项目经理，我都和所有的人一样，并无特殊地位。我当经理的时候，他们一样对我大呼小叫的，从来不觉得我高人一等，对他们来说，经理只是小组中一位不专门写程序、肩负其他工作的人而已。

对于很多不懂美式足球的人来说，四分卫是全队中最风光的人物，全场中只有四分卫能指挥每一位球员，俨然就是控球的焦点，获胜的时候，被所有的球员抬起来欢呼的也是四分卫。

看起来四分卫好像比其他的人地位高些，但是如果您真的了解美式足球，您就知道四分卫只是球队中担负特别责任的人，并没有高人一等。一位优秀的项目经理也是如此，虽然项目由他领导，但并不表示他就比其他的组员享有优越的地位。

项目经理和其他人一样是团队的一员，不同的是他肩负着与其他组员不同的责任。

一位优秀的管理者非常明白，团队中每个人都扮演着一份角色，有些人负责输入界面的设计，有的负责打印功能，也有些人负责多国语言版本或是档案转换等。项目经理除了一部分的程序工作外，还必须定出工作的优先级、沟通协调相关的部门、营造有效率的工作环境、培养组员技术更精湛以增加他在公司的价值，做这些事情并不需要比别人高的地位。

倘若有一位项目经理自认为高人一等，比组员优越，接踵而来的就是以下的伤害，我举的是比较极端的例子：

◆项目有问题时，主管只会责怪组员，但是如果项目顺利成功，主管又认为自己功劳最大。

◆主管并不关心组员，组员只不过是工作机器，谁在乎他们是不是每周工作 80 小时？他们唯一关心的只是万一进度落后，自己会脸上无光。

◆主管要求组员对命令绝对服从，从来不敢质疑主管的权威，“ 我说做，你就得做 ” 是惟一的信条。

◆主管永远无法忍受有人在某些方面比他强，只要有人似乎在知识或技术上超过他，他就本能地采取否定和攻击的态度。

◆因为主管永远是对的，所以他从来不肯认错。

◆任何组员对产品开发方式有改善的建议时，主管只把他当作找麻烦而命令他闭嘴。

◆主管把自己当成项目不可或缺的灵魂人物。

老实说，也不是所有的主管都自以为高人一等到这种地步，只不过，还是有那么一点凌驾他人之上的气氛。如果当属下的人是您，您喜欢“ 为上级主管工作 ” 还是“ 和一位职称叫经理的同事合作 ” 呢？您只要稍微观察一下主管流露出来的态度就知道了。

如果一位项目经理把自己当作小组的一员，工作起来必定更有效率，因为他不必用斗争来巩固自己的位子，而且本来就没这必要。如果主管和大家是一样的，那么，当小组里出了一位超级巨星的时候，他就毋需过度防卫自己的崇高地位，而与超级巨星互相掣肘，反而会感谢上苍赐予自己如此能干

的同事呢！

您用什么态度面对自己“身为主管”这件事，影响着您做的每一件事。如果您的属下对于考绩有意见，您会怎么做？您会坚持自己是“对”的吗？或是尝试从另一个角度来解决这个事件呢？如果您的属下仍然不同意，您会不会修改考评报告的内容，加列双方的意见，让上级在读这份报告时能有评估的空间？

再回头看看那些自高自大的主管，他表现出来的行为有可能和一位平易近人的主管一样吗？您希望和哪一种主管共事呢？是自以为高人一等的主管，还是对每个人同样尊重的主管？将心比心，您就能扮好主管的角色了。

主管应该把自己视为团队中的一分子，与其他人平等，而不是高高在上。