绪论——软件开发的四个时期

理论上，成功的软件开发似乎并不困难：首先必须深入了解顾客的需求和在市场上的角色定位，然后设计出软件产品的蓝图——它必须比别的产品更能切合顾客的需要。然后组成一个开发小组制作这个产品，完成之后将它推出上市。之后再用各种营销方法告诉潜在顾客这个产品特有的价值，顾客就会高高兴兴地买下它，用起来非常满意。然后你的公司就成为软件产业的个中翘楚，会跃上《财富》（Fortune）、《滚石》（Rolling Stone）、《成功》（Success）等专业杂志的封面，成为世人争相传颂的典范。

理论上，准时完成软件产品看似容易，但事实上，大部分的公司都失败了。

听起来真不错，但事实可差得远了，也许是因为始终没有一套真正有效的软件开发工作手册之故。一旦有了成功的软件开发模式，人们会期望按照这些步骤进行，然后获得成功。在后面我们会说明如何设定并维系整个组织的这种期待，但首先得声明，按进度做出成功软件是世界上最困难的事情之一，虽然我们多么希望它每次都实现。您不能完全倚赖公式，每一个软件都有它的独特性，在开发软件的任何时刻，都需要特别的付出才行。但相对地，软件开发的成功是世界上最令人快慰的事情：必须深入了解顾客的需求，必须组成高效率的开发团队，对软件产品的定义必须切中市场利基，同时开发程序必须正确无误。然后，伟大的产品终于诞生了，在市场上造成震撼，媒体不厌其烦地一再介绍您的产品，顾客带着大把钞票抢着购买。

是的，这是有可能的。本书的目的就是告诉您如何使软件开发成功的技巧，这些技巧是以一条条原则的方式表达，这样比较便于记忆，也比较实际——小箴言大妙用——对软件产品的定义、发展、行销各方面都有重大的影响。要准时完成优良的软件产品非常困难，但不是不可能。本书可能改变您的观念，在过去您认为重要的因素可能事实上不怎么重要。开发方法、程序、优越的技术能力与开发工具，以及项目管理的技巧，固然对软件开发项目的成败有重大的影响，但管理者是否能充分实践本书介绍的观念，才是决定性的因素。

软件开发是一项多元化的工作，而本书强调众多原则的整体性。我认为，与其说软件开发是可依进度或功能切割的项目，不如说是一种第六感。我绝不是轻视组织、协调合作之类的重要性，而是要指出那潜藏在井然有序组织之下的各种复杂的心理活动，包括创意、群体的互动、本能、科技时尚等各种力量，会造成一种骚动，而这正是创造力的来源，这是开发团队的发光发热的能量，将它适当地导入软件产品，会使软件更有特色、与众不同。

在看似平静的表面下，软件开发的内涵其实是充满骚动不安的。

软件开发的省思

软件是一种智能财产，软件开发基本上是一种智能的投入，磁盘或 CD 上的每一个位元，都是开发团队的智能结晶。在软件中的智能愈高，软件的价值就愈高，能被市场接受的价格就愈高；好的软件不需要太多的广告或行销成本，它本身就能吸引大量的顾客，为创造它及使用它的人带来利润。

对于一位领导或监督软件开发团队的经理人，正确的观念是——将软件

开发活动的重心放在把智能从人脑转移到计算机的过程。大部分的软件开发经理人对自己的领导任务都没有正确的认识，以为他们的工作就是设计和测试软件、制作系统文件、行销软件这类的事宜，而把焦点放在软件开发过程的管理上。

这种误解常常是造成进度延误的主要原因。不论软件的品质好或坏，它总是延误进度，即使极严重的落后也已经习惯成自然。这种进度严重落后的软件被我们戏称为“泡沫软件”（Vaperware），因为它随时可能消失，倘若不是它所背负的那种心血付诸流水的心痛，倘若不是它所代表的人力、时间、金钱、社会资源的浪费，这个名词也许称得上是幽默吧？

软件开发管理的真正任务是将团队成员的智能充分而适宜地发挥，并有效地投注在创造软件的活动中。这种智能指的是抽象的人类脑力，例如创造力、聪明才智、理性、效率和高贵的人性等等。创造智能财产的重点是，你必须将一群人的智力，用很聪明的方式结合起来，这也是整个软件开发过程中最辛苦的部分。按时完成软件虽然做起来困难，在观念上却很容易理解，作为一个清楚的目标，是 每一位关心这个问题的人都期望达成的事。

想如期完成伟大的软件，那是需要每个人全力投入的。

假定财务支持不成问题，那么剩下来惟一重要的事，就是团队是否全心地投入、各组员的智能是否有效地结合。对于愿意思考的人，本书中所有的观念和基本原则都是浅显易懂的道理，可以说是常识而已。要整合团队智能，只有三件事是真正重要的：激发团队思考、掌握正确的思考方向、有效地思考。如果您曾经参与过这样的智能型团队，您可能已经发现这三个（也许更多）重要的诀窍了。

那么问题究竟在哪里？既然软件开发的管理原则不过是这么简单的常识，为什么成功的案例少之又少？

让我们先思索一下大多数企业内人力资源运用的模式。一般来说，企业的人力资源大都投入两类领域：一是创造性的智能活动，一是重复性的机械活动；前者像是设计或规划，后者像是生产或施工。到目前为止，大部分的企业中都是将人力投入在重复性的机械活动。虽然一部车、一栋房屋或是一条公路，都有智能的投入，但相对于整体的人力资源，智能的比例相当低。

让我们再来思考，如何组织一个以重复性机械活动为主轴的企业？“效率第一！”是的，所有的附加价值根源在此。自亨利·福特倡导“生产线”的观念，开创了 20 世纪工业发展的历史之后，所有的生产活动都像是一部无情的、公式化的、不断重复工作的大机器：原料从一端源源不断地灌入，产品从另一端持续地产出。为了管理这个庞大的机器，阶层式的管理组织应运而生。每一个人在这个组织中只是一个小螺丝钉，工作内容是狭小范围内精确定义的角色。只要每一位成员在小小的岗位上克尽职责，整体的效率就可以达到最高。

然而，软件的生产全然不是这么回事。因为真正产出的是无形的智能财产，而不是有形的商品，重复性的机械活动所占比例急剧缩小，而以创造性的智能活动来扮演决定性的角色。

当然，将软件包批量生产的拷贝工厂不容易，但是大概也不会比一般的消费性产品（像是照相机之类）的批量生产困难。毫无疑问，软件公司最大的挑战是第一份成熟的软件，它的创作过程复杂而充满不确定性，只要原型做得出来，其他的事情就容易多了。

软件生产的复杂性和不确定性主要是来自创造性的智能活动本质。大部分的企业组织不是为鼓励思考而设计的，您必须对企业组织做点修改，现在流行的说法是叫作企业再造（re-engineering），才能建立适合的软件开发环境。您必须找出所有阻碍人们思考的原因，把这些阻碍因素去除，而不是把这些人去除。

在本书中，我们企图掌握成功软件公司的整体心理型态（gestalt），这是无法整齐划分成各个功能片断的。对企图掌握软件开发日程的经理人而言，它非常重要。对于程序开发者而言，充分了解日程的关键意义也是同样重要。同时，如果任何一位团队成员对软件开发计划缺乏全盘性的了解，他/她的贡献将只局限于直接指派的工作，变成仅只是程序产生器而已。这是人力资源与智能的绝大浪费。如果您要软件准时完成，就必须使每位成员的心力都投注于此，而且保持这样的向心力直到软件完成，这是软件开发经理人的首要任务，也是本书的主题所在。

软件开发的四个时期

在本书中，我将软件开发的过程，分为四个时期：

布局时期（Opening Moves）包括拟定合理的市场策略、产品设计、开发计划和一切序幕的工作。在本阶段中最重大的挑战，就是如何在一切百废待兴的混乱中，组织您的开发团队。就像是一盘棋局的第一步棋，奠定整个项目的基础。我把项目开始的那一天，称作“行动日”（Moviny Day），这是布局时期的最高潮，项目于是开始。

中程时期（The Middle Game）大部分的时间都花在这个阶段，也许会在第一次进度落后时无疾而终，也许会在下一个推出时期画下完美的句号。中程时期可比喻成圣经所述 40 年漫长的劳苦岁月，无数信仰坚定的奴隶和贱民，在耶和华指引下将沙漠打造成为美丽的国度。软件开发的项目通常不需要 40 年，但其中的艰辛和血汗，会让人们感觉仿佛真的过了这么久。坚持到最后的一组人马会是个奇迹似的生命共同体，除了坚定的信念和奉献之外，大概只会想早点把这该死的软件写完，忘掉这场恶梦吧！

推出时期（Ship Mode） 这是您脱离苦海的惟一出口。就像婴儿出生一般，充满痛苦和恐惧，但却是无法避免的。必须经过这个阶段，软件才能诞生问世。最令人振奋的就是出生时的喜悦，软件的完成是本阶段的最高潮，也是最后一次必经的痛苦。

发布时期（Launch）这是营造气氛的时刻，软件上市发布了，您必须在顾客心中留下深刻而永恒的印象，至少到下一次新版本发布前都还记得您。经过这个阶段，软件才算是真正交到使用者手中。

如果您不是软件包从业人员

您可能不是软件包从业人员，本书对您还是很有参考价值。本书虽然以软件包的开发为主轴，但内容是来自作者本身的经验，其中最基本的原理原则，是可以适用于任何其他领域的，相信您可以很容易将本书的内容想象成您实际面临的工作环境。

第一篇　布局时期

很多探讨软件开发的书籍都假设自己存在于理想的状态下：团队本来就应该非常专心且克尽职责地完成工作，并完全掌握任务的本质；他们收集得到完整的需求情报，建立设计规格，并导引软件原型（prototype）一次又一次的蜕变；他们请求使用者参与，使用者便会全力配合，而且非常认真地与开发团队共同完成深入的需求分析。所有的事情都是那么完美顺利。很不幸地，理想状态是不存在于现实世界的，在真正的软件开发项目中，您可能看不到一丝一毫属于完美顺利的景象，而且看到的恐怕只有一堆的问题。

不只是问题，还有可怕的挑战。优秀的开发团队在布局时期必须做的事情多如牛毛、广如大海，我们大略分为五个范畴：组织、竞争、顾客、设计、开发。布局时期的工作是多维的，而且必须在每一个细节中都能综合、兼顾所有预期的结果。

组织开发团队

我所谓的组织，是指集结适当的人选分别担任下列角色并参与设计：

项目管理（Program Management）——负责制定开发日程、与外界沟通、寻求技术方面的后勤支持。

软件品保（Quality Assurance）——测试与评估软件的品质。

程序开发（Development）——写程序、抓错虫。

产品管理与行销（Product Management/Marketing）

——负责整个产品的形象定位，传递正确的产品信息给顾客，以及产品的上市发表、与传播界的沟通。

系统文件与使用者教育（Documentation/UserEducation）——负责以文字表达正确的产品使用方法。（这里所谈到的设计，是指软件产品的设计，不是指写程序或程序设计。读者耳熟能详的“程序设计师”，译自 Programmer 一词，虽然这种工作有相当的设计成分，但这种设计比较不是艺术性的，而是工程性的。原作者甚至鲜少使用 Programmer 一词，而以 Developer 表达。概略地说，软件产品设计是厘定目标，事先定义出软件要做到的功能，本产品预定要满足什么需求，目标顾客是什么样的，主要的硬件或操作系统环境等等。——译者注）

您不一定要将自己的团队成员划分成这五种角色（虽然我认为这是最有效率的做法），但务必要确定这些工作都有适当的负责人选。请注意每一种角色都有参与设计，如此每一位成员对项目都有整体性且清楚的认知，使每一位成员的目标是一致的。

如果您的开发团队无法合作无间，对于目标老是有不同的意见。那么，首先要做的是找出不团结的真正原因。

品保人员（QA）是少数民族？

如果品保人员认为他的工作是测试程序，而开发人员认为他的工作是写程序给品保人员测试，那就得小心了，这是一个警讯。这种情况会造成开发人员和品保人员之间的疏离，开发人员的优越感会使品保人员感觉自己是被歧视的少数民族，当然会影响到软件的品质。品保人员的最主要功能，是不断鉴定和评估产品的现状，是否在品质上和功能上确实遵行产品目标，而让其他的人员专心投入他的职务。

品保人员的评估工作是一项整体性、持续性软件开发活动中的一环，而不是偶尔来点缀一下。好的评估报告在本质上应有客观的分析和衡量标准，如此才能导引软件产品符合现实的需要。这种导引的重要性是不容轻忽的。因为在开发过程中，开发人员可能因为一些偶发的小事或某种无关的灵感而不知不觉地偏离了现实的需要，暂时忘记了什么才是产品最该有的功能。品保人员的职责就是为软件的品质把关，以现实、客观而市场导向的眼光，不断地检视软件产品。

谁来设计产品？

如果项目经理、产品经理、开发人员不断争论谁有权设计产品或是各执

一词，这是一个愚蠢的开发团队，只会关心自己的权威；然而，真正的权威是来自于对现实状况的精确掌握。产品设计的目的是将最好的想法列为产品的基础，每一位工作人员都应该为此努力。至于什么是最好的想法，应该在项目真正开始之前就通过实地检验。市场调查是很好的方法，不需要太多的时间或成本，就可迅速平息大家对于产品设计的争论。我们在下文中会讨论如何增进彼此之间了解，这也是解决方法之一，精确缜密的思考通常能使问题柳暗花明。权力争夺战会迫使人们心胸狭窄互相竞争而不合作。

产品设计的争论是权力的争夺战，会使人们心胸狭窄，互相竞争而不合作。可能有人主张产品在这项功能上有所不足，结果焦点变成了功能而不是产品本身，一位有智能的领导人应该将争论视为组织出现问题的症状，而去发掘问题的根源，这比仲裁这项争论要有用得多。在这方面我的经验与传统所谓的智能刚好相反，我认为用职权裁定谁有产品设计权是毫无用处的，虽然改变正式的产品设计权或许有点帮助。

老实说，谈这个谁来掌权、谁来负责的主题并不有趣，而且解决这个问题非常浪费时间。组织健全的团队是适才适任，各司其职的；此外，每个人的除了各自专司的角色之外，还担任一般的角色（generic role），这一点也受到管理者的肯定，因此角色与能力之间的关系能够取得和谐。一般角色会使团队更能融为一体，互相支持。团队是一种平衡的生态，人与人之间的职权界线像是和睦邻人的关系，于是，对每一位单独个体都有足够的尊重，整体的效能也达到最佳化。这种平衡的组织生态是自然生成的，强制性的外力干预只会造成负面的影响。如果团队成员彼此的关系无法做到亦邻亦友的和谐，领导人必须分析原因。通常如果利益分配不公平，或有多人争夺同一项利益，或是提供的报酬不够齐全，不平衡就会发生；有时候管理方面欠缺弹性也会造成资源必须用人为的方式重新分配，效果不一定会好。

在任何项目开始之前，项目经理必须弄清楚团队中有哪些地方需要特别的注意，尤其是团队的变动性（teamdynamics）。就像团队中的每一份子必须清楚要做什么工作、有什么资源、目标在哪里，项目经理也必须设计自己所要带领的团队该有什么样子，自己该如何“激活”这个团队。因此，我们导出了法则 1。

法则 1

Establish a shared vision 建立一个共同的目标

乍看之下，这何需赘述，但这却是相当困难的事。团队中每一位成员都必须非常清楚我们要做什么、成品会是什么模样、基本的产品策略是什么、什么时候必须完成。凡是与共同目标相冲突的看法都必须转化成一致，而不是把它消音。和谐的共识是绝对必要的，否则软件不可能做得好，很多事会复杂化而难以收拾。

建立共同目标的方法从独裁式的极端到放羊式的极端，中间有无数不同的方法，但在我们讨论这些方法之前，得先说明何谓目标，毕竟目标的建立已是现今衡量领导能力的指针之一。简单地说，目标是共同的希望，对未来的事情所描绘出来的景象。

目标一词常被政治家或经济学家使用。任何一位好的领导者，都有责任为其追随者创造目标。以我的看法，领导者对其组员心理状态的掌握与认知，

是目标的开始，然后领导者对于组员之间复杂的心理状态认知、修正、建立共同点，融入领导者的个人特质，使得领导者与组员、组员彼此之间的防卫界线逐渐消失。有共同的目标才能建立团队的认同感和归属感，大家会觉得我们是一个整体，这是团队气氛的心理基础。

团队的认同感和归属感，能消除个体的自利行为。

通常，共同的经验可以作为领导者的感情移入，领导者用各种不同的形式告诉追随者：我们过去是那样，我们现在是这样，而我们将来会是什么样。

不论是否有共同的历史，“ 成功的领导者 ”能在团队 中营造共鸣，而“ 群众煽动者 ” 则不能。当组员对领导者意见相左时，最常见的反应不是直接反对，而是：“ 是啊，我们也是这么认为，但我们该怎么办呢？ ” 这时必要的努力、鼓励和妥协都可能发生。领导者应该自问：“ 如果换他们来领导，他们会怎么做？我该如何将他们复杂的情绪转化成行动力？ ” 这些问题的答案能帮助领导人解决大部分的问题。

有远见的领导者会构思一个美好的前景，需要大家共同努力创造；而群众煽动者则是对现状不满，意欲去之而后快。有远见的领导者会带领不同心态的人朝共同的目标努力，有时为了长远之计而放弃近利；而群众煽动者则是逞一时之快、急功近利的人。

领导能力与目标是来自领导者与组员之间的心理共鸣，若非如此，目标会流于虚幻，也无法带动团队迈向成功。

微软程序语言部门的故事

1992 年的微软程序语言部门，情况实在不乐观。那年的四月，微软发表了 C7。C7 花了超过两年的时间开发，经历无数次的进度落后和品质恶化，似乎是一场永无休止的死亡进行曲。而微软的竞争对手已经抓住这个机会，在 Windows 版的 C 和 C++上面获得很好的评价。C7 在很多地方都不足以与之竞争，虽然它有精美的包装、动人的价格、优异的类函数库（classlibrary）MFC1.0，和微软素以见长的最佳化执行码编译器（compiler）。这些优点只能使微软勉强维持不被淘汰出局，但很显然下一版非得大幅扭转劣势不可。

微软的 C/C++开发团队经历了好几年的流年不利，光是项目经理就换了三次。丹尼斯·吉伯特（Denis Gilbert）是新任的资深开发部经理，而我是新任的行销兼使用者教育部经理 ;虽然我们都从未领导过这么庞大的团队(超过两百人),但我俩都充满斗志,企图有一番作为。杰夫·赫伯（Jeff Harbers）管理 AFX 小组，这个小组集结了微软的精英工程师，负责为 C/C++的产品提供“ 所见即所得 ”（What You Seels What You Get， WYSIWYG）的类函数库工具。他们的经验极为宝贵，虽然两组人马的文化差异刚开始难免造成点小摩擦。

程序语言部门是微软中最古老的部门，第一个产品就是比尔·盖茨与保罗·爱伦（Paul Allen）合写的 Basic 直译器。当微软进入操作系统与应用软件领域之后，程序语言的重要性相对降低许多。微软的重心转移之后，尽管微软的程序语言产品有占先的优势，但开发者的辅助工具方面则跟不上竞争者了。

在 1992 年，大部分的微软高层主管都希望重新振作这个部门，也许是被忽略得太久了吧，我无意批评他们，但我猜多少也是微软扩张得太快太广所致，无论如何，这使得我和丹尼斯肩负着更重大的责任，更强烈地希望把事情做好。

对我们的压力与期望不只来自微软高层，还有来自专业杂志的负面评价和使用者的责难，甚至微软内部同仁也轻视这个部门。大家都说我们缺乏目标，是一群笨蛋，要不然就笑我们 C7 品质不良，完成日期老是拖延。就连比尔·盖茨的电子邮件也会带上一句——程序语言部门是全微软最笨的一群人。这一切不只是没面子而已，我们实在该做点什么。

回想过去这段时日的发展和本部门堆积如山的工作，我们（主要是丹尼斯）发现本部门一直被以下的几个错误所蒙蔽：

·我们无法完成任务，因为我们没有足够的能力。

·即使我们每件事都如期完成，也一样会失去市场。我们没办法叫媒体不批评，或叫使用者不要抱怨。

·我们的人力素质和数量可以做一个项目，最多两个跨出重要的第一步

丹尼斯征询过所有的经理人的意见，要求他们将各项项目作个评比，发现每个人都认为 Caviar 是最重要的项目。Caviar 就是 Visual C++ for Windows 3.1 的计划代号。我们的结论是只要将 Caviar 做得好，又能在适当时机推出，就是打赢了一场重要战役，甚至会带来最后的胜利。第二重要的项目是 Barrauda，基本上就是 Caviar 的 Windows NT 版。

我们必须立刻加强这两个项目，幸运的话两个都能完成，但若做不到，至少要集中火力在 Caviar。我们把所有的资源都集中在 Caviar，期望它赢得 Windows 3.1 的市场胜利。

于是我们把手上大大小小的项目逐一列出，要大家票选自己认为最重要的项目，以期形成初步的共识。但丹尼斯并未加强这个共识，他甚至并未再邀请组员发表意见，因为他已做了决定。我们明白 Caviar 一定要赢，虽然希望渺茫，大家都愿意尽可能贡献自己的力量。

丹尼斯召来了全体组员，向大家说明我们要如何打败竞争对手。我们必须将不重要的项目放弃，并且重新整编改组。

这是一个好的开始，但只不过是开始而已，其他重要的条件诸如团体共识、决断能力、清楚的目标、组员心理上的满足感等都还差得远。丹尼斯注意到整个团队都为无法打败竞争对手而感到愤怒而沮丧。他自己也有一股驱动力，想把这一切好好整顿。他感受到只要有人把胜利的希望带进团队，全体组员都会发誓：“我们要赢！”

Caviar 会成为 Windows 3.1 的 Visual C++。

丹尼斯的决心点燃了整个团队，Caviar 非赢不可，虽然大家对于部分项目必须牺牲放弃不免情绪反弹。显然丹尼斯已经创造出成功的必要条件，但他还缺少充分条件。

问题接踵而至

这群人从来不曾形成团队。我们也没有跨部门的协调组织，而且当初所期望的充分授权管理方式（consensus－style management）还处于很幼稚可笑的阶段。我们没有跨版本的产品计划，也没有详尽的技术规划，过去也从来没有成功的经验作为开发程序的参考。几乎每个人都高估了我们如期完成的能力，即使对于我们刚刚设立的新目标也是如此。但无论如何，弃车保帅的策略和改组行动确实为部门注入了新的动力，但结果还未见分晓，我们可不知道是否能一举奏效。

几个月过去了，事实告诉我 Caviar 实在不太可能如期完成，事实也告诉我非得如期完成不可。我们已决定在 1993 年的 Software Development ’93

West 展览上发布 Visual C++ 1.0，同时铺货在各种通路上，要让竞争者来不及反应。这是我们成功的关键，也是我们士气和信心的关键，最后的期限是 1993 年 2 月 22 日。

然而，这时的开发工作只能说是一团糟。我们仍不断添加零星的功能；好的一面是我们还在不断提升系统效能，坏的一面是几乎无法将模块建构成完整的产品，甚至无法每天产生一次完整的执行档。开发人员把软件品保当成是黑箱作业，每天丢一份雏型给品保就继续埋头苦干了，根本不管品保人员为时数周的测试。

大约在推出前的四个月，我受到了赫伯的严重警告，他很坦诚地告诉我，如果我真要做好这个产品，必须改变一些做法。他使我了解到除了我自己，没有人能阻碍我做该做的事。我明知道该做些什么，却往往因为某些因素（大多是个人的因素）而退缩不前。我现在是行销经理，虽然过去在微软的 R&D 部门曾经领导过 15 个项目，但从未加入过 C++的团队，如果我再不好好表现，我大概就是第三个卷铺盖走人的经理了。

赫伯的忠告使我明白我太消沉了。我可以清楚看到事情是如何一塌糊涂，我有责任改变它，而不是逃避它。赫伯向来以毫不遮掩、实话实说闻名，而且总是能够一语中的。我终于被他的话点醒了。

于是我和丹尼斯讨论了眼前的困境。我们其实无法确定该怎么做才对，但都明白非得有一番彻底的改变不可。当天下午我们便召开了一次小组会议，这件事本身就极不寻常，抓这么多人过来只为宣布一件小事？是的，我们要在组员的心理上制造一个转折点。

我们为了求胜已经尽可能减轻不必要的负担，而我们仍然在市场上失利，在同类产品评比中输给对手，我们已经受够了失败的滋味——总是在进度严重落后的情况之下做出平庸的产品——而我们明明知道自己可以做得更好。只要我们同心协力，我们一定可以在期限内做出最优良的产品；没有人能阻挡我们，只除了我们自己（当然也有些不可知的因素）。我愈思考眼前的处境，以及它对微软公司、对我们的意义，就愈觉得斗志高昂。

我们有士气、充满活力，一切成功团队所需的条件我们都有了，只除了共同的目标。我个人有个想法：我们不但要追上日程，也要做出最伟大的产品 Visual C++1.0，要令所有的人都刮目相看，尤其是那些一向对我们避之惟恐不及的营销人员。我们要扭转战局，让对手俯首称臣。

我的求胜心很强，但矛盾的是，我却不敢告诉我的同事。没有人阻止我说出心中的想法，倒也没有人要求我说，这种时候个人的感觉似乎已经不重要。我害怕自己被嘲笑：“他是谁啊？新来的嘛，这搞行销的家伙想教我们写软件？”

然而我再也无法沉默，再不做点事情我在微软就混不下去了。那天下午我终于告诉全体组员我的感受，也问问他们的感受：“我受够了挫败，你们也是吧？作为全微软最差劲的项目经理，我恨透了，外界的批评令我觉得丢脸，你们也是吧？Visual C++是最伟大、最值得骄傲的产品，你们难道不这么认为吗？我知道我们可以如期完成它，而且用它来给对手一个迎头痛击，不只我一个人这么想吧？我们将创造微软的明日世界，我们会大有作为的，不是吗？”

结果，几乎每个人都有类似的感受，纷纷发言为我补充。我终于整合了共同的目标：我们要准时完成 Visual C++1.0，没有什么事情比它更重要。

所有的资源都投注在这个清楚明白的目标；不必要的枝节可以牺牲，其他的项目（即使是 NT 版的 Visual C++）都可以暂缓，全部的人、钱、机器，都押这一注，而且得立即行动！

会后我们紧接着要求每一个工作小组至少提出五项具体务实的建议，只要对 2 月 22 日完成 VisualC++有帮助的想法，请大家务必提出来。我获得了许多宝贵的建议，本书的第三篇“推出日期”中，就有许多很棒的观念是来自这次的建议。而此时此刻，大家的心理上和情绪上都有完成目标的强烈的决心，因此会主动告诉管理者该做什么，然后确实去做。而我和丹尼斯则负责提供工作环境，凡是对目标有帮助的，我们都全力支持，甚至包括一些背离传统却实际可行的构想，这就是员工领导的管理风格（manage-ment-led style）吧。

我个人觉得，在任何项目的执行过程中总有些“关键时刻”，使大家在心理上和情绪上因此凝聚成对目标的共识，对共同的目标产生共鸣，对事情的优先级形成清楚的认知。我无意夸口 Visual C++的团队有多伟大或领导多卓越，我的目的是指出在大家一片士气低落时促成大家态度转变的关键因素。

首先，有人察觉到了团队发生问题，并勇敢地指出来。赫伯以他多年丰富的经验，独特的真知灼见，看出了我们在胡乱挣扎；虽然我们的做法基本上是正确的，但却不够。赫伯没有任何义务帮助我们，但他仍然不吝直言。

赫伯在这件事情上，完全克服了人性的弱点。一般人很少会主动帮助并未求助的人，尤其是软件开发这种智能财产的工作。赫伯首先必须对自己的看法有足够的信心，并且在心理上要冒着很大的风险；然后，赫伯确信自己的行动对事情有所助益；最后，他必须不介意是否失去我的友谊。为了帮助整个团队，他必须冒着得罪我或整个团队的风险。

他的忠言刺伤了我的自尊心，激起了我的防卫心理，并且刚开始时不断和他争辩。他说我搞砸了一切，我应该彻底改变做法。我用尽自己的脑力和心力反击他的“帮助”。而他则以关怀和冷静的事实击退我的不理性。直到凌晨四点，我还特别记得他冷冷地说：“别管我的语气，别管是谁告诉你这些，忘掉你的骄傲，只要你听得进去我的话。”

渐渐地，我想我可以接受他的劝告了，我的情绪平息，思考也变得更有建设性。我开始觉得不必因为接受他的建议就感到自己不如他，我太骄傲了。想通了这一点，我的创造力和旺盛的活力又恢复了。

这个事件的教训是，我们应该坦诚将事实告诉那些当局者迷的人，我们应该告诉他们问题在哪里、他们的潜力在哪里。如果你的心胸够开阔，当你犯错时就会有人及时纠正你，甚至提供更有用的建议。如果你的忠告被曲解成太骄傲或自以为什么都知道，或者“你是谁？凭什么批评我？”，别管他们的防卫心态，直接告诉他们的情绪正在混淆事实。如果他们说：“什么，你以为只有你懂软件？”你只要在被拒绝前简单告诉他们，你只不过提供一些想法，又没有恶意，何不试着了解呢？

单纯的沟通的效果可能会昙花一现,你必须真的以彼此的情谊为赌注(有时甚至冒着失去工作的危险）来讲出实话，如果不是这样，可能很难突破对方的心防，而打从心底接受你的建议。

这种实话实说需要高度的技巧与智能，但却是建立共同目标的基础之一。经过不断发掘出最真实的情况，才是鼓舞团队士气的金钥匙。

然而，真话也得有人听才行。事实是最残酷，实话最难听，但不论是你自己发现或别人告诉你，你终究得接受它；缺乏自信心或安全感的人比较难接受事实，而抗拒事实或情绪反应会使人丧失创造力，或是发生错误的直觉。

这是两个阶段的过程，你不只要学会接受事实，还得学着传达真实的信息。传播事实的最佳媒介是情感，这是你用之不竭的，多付出你的关怀和好意，让事实被正面地接受。即使你被拒绝，你也可以达到鼓舞对方的目的。

我们终能如期完成 C++的开发，而且它确实是一件伟大的作品；微软的行销部门也全力支持我们。我们终于打败竞争者，虽然压力和威胁还在，但我们毕竟扭转劣势，赢了这一仗。

往后的日子里，我们逐渐建立了版本控制的规则，我们可以每隔固定的期间发表新的版本，以及共享版本（请参阅法则 3：建立开发多版本的技术规划），但最艰苦又最令人难忘的、团队精神最炽热的，还是那第 1 版的经验。

法则 2

Get their heads into the game 使大家主动投入

如果每个人都在认真思考如何使团队更有效率，这个团队自然就比较容易表现出高效率，反之亦然。听起来又是老生常谈，但事实上，让每个人的思考都动起来是一项管理上的重大成就，这需要有人用智能和努力营造出这样的环境，才能让它发生。

这种全体一致为整体设想的思维，其重要性绝不致被高估，但其困难性却常常被忽视。在团体思维之外，每一个人都有自己个人的想法，不能说不对，却常常是问题发生的根源。

每个人都有自己的思想、价值观、信念，也许不会表达出来，当然也不是每件事都非得说出来不可，但是团队中大部分的事都值得花时间沟通，特别是有人的想法非常敏锐时（请参阅附录）。每个人的想法都值得注意倾听，而且其所隐含的信念都值得花时间了解，尤其是有流言、断章取义或误会发生的时候。

人们的行为来自信念，人会为行为改变信念，也会为信念改变行为。

创造力的可贵

在一个需要创造力的环境中，好点子是愈多愈好。你只要鼓励大家有这种共识，很容易造成深远的影响。假设大家在讨论一个问题时遇到瓶颈，没有好点子，你的做法是，规定每个人至少提出一个或两个好点子，然后你就会发现气氛动起来了。

有好点子是令人愉快的事，尤其当两个看似无关的点子，凑在一起突然变成一个崭新的、更复杂更丰富的好主意时，大脑中的愉悦中枢会受到刺激，所以创造力总是伴随着喜悦。

有人倾听或了解自己的想法也是令人愉快的事。最棒的是你的好点子被大家广为使用或接受，这本身就是心理上的很大的快慰与满足。想出好点子是内在的脑部活动，但当别人恭贺你或肯定你的点子，你会禁不住开心地微笑。

好点子是具有感染性的。真正的好点子应该像病毒般广为散播，每个听到的人都会感受到创造性思考的快乐。

好点子具有相乘的效果。当好点子打破某些错误的假设时，会有更多更好的点子被激发出来。

创造性思考能培养你的洞察力和头脑敏锐度，以及对新想法的成本、价值等的评估能力。

创造性思考只有好处没有坏处。虽然培养某种特殊的创造力不是件容易的事，但“想出好点子”本身很单纯，你多用它就会磨亮打光，一旦时机到来，创造力就会像核反应一样，绵延不断地增生扩大。

为什么组员会怠于思考或是不敢说出想法？

·因为他们认为没有人会重视他的想法。

·因为他们认为该由别人告诉他该做什么事。

·因为他们认为这样做没有什么好外，只会使老板皱眉头罢了。

·管理者只管发号施令而已。

如果人们的想法被采纳，他们就会愿意思考和表达想法。如果他们的新点子能够被团体接受，就表示值得实施，团体的力量会更强化好的想法，这就叫做“授权共决”（empowered consensus）。或许有更适切的名词表示这个抽象的观念，我选用这个名词是因为它表达出决策形成的两个重要过程：主动思考与尝试表达。

法则 3

Create a multi-release tech- nology plan

建立开发多版本的技术规划

组员如果乐于参与我所谓的“技术规划”（technologyplan），会对未来更有信心。授权共决是所有信赖的基础，技术规划是它的结果之一。在民主式的授权共决中每个人都有投票权，参与技术规划的决策。更重要的是，每个人的想法都受到重视，被认真地评核。技术规划是团队行为的基础，在最理想的情况下，它包含了全体组员最好的想法。

技术规划是组员对团队工作的契约，也是团队的宪章。一年大约修改一到两次，以适时反映每二位组员的意愿，这么做会让组员更有安全感，更愿意信赖彼此和未来。技术规划勾勒出了大家信赖的未来的蓝图，因此，假定有一位组员对目前分配到的工作觉得不太喜欢，没关系，下一次改版时他可以挑个比较有兴趣的工作；或者有一位组员不喜欢现在的产品方向，可以暂时退出，等下次技术规划修改时再加入。

技术规划对于日程的掌握也有帮助：开发人员总是想把一些了不起的功能加入，常常和进度不能两全其美。如果有技术规划，开发人员会信赖未来，他有的是表现机会，不必急于一时。另一方面，开发人员很清楚该在适当的时间做适当的事，既不会负荷过重，工作品质也更能掌握。

当每个人都很明白在这个产品中自己该做什么，事情就很容易被精确掌握。其中的关键就在这种“割爱”的艺术。

一个技术规划的实例

有一次我受邀担任一个软件开发团队的顾问（不是微软），协助他们建立技术规划。他们的资深项目经理出身该公司主要的技术部门，决定集中力量在一个五年计划（即使两三年的时间对科技来说就已经是全面改观了，我一般不建议定这么长的计划日程）。当时，他们正在讨论各种新技术能为工

作带来的好处。

在讨论中有一个主题很有意思，是他们年久失修的关键程序。这个程序庞大而脆弱，有很多错虫，跑得很慢，其中有一部分的程序代码甚至已经超过 10 年。组员中没有一个喜欢它，很多地方根本不符合现代的设计原则，他们甚至不愿去碰这个大麻烦。

我想每个公司或多或少都有这类的程序：程序逻辑纠结交错、缺乏弹性、冥顽不灵，简直令人头疼。这种程序最后会变得无法维护，更别提加入新的功能了，结果是大大限制了开发团队创造优秀软件的潜力。

我把这种情况称之为“软件开发的恶性循环”，你不可能在推出下一个版本的同时，彻底翻修旧程序。谁能承担得起重新经历一次软件开发的整个过程，只为了将程序改写，而这段期间很可能降低或中断客户服务，在程序还没有变得更有弹性而能搭配新功能之前，客户可能得等上好几个版本的时间。

比恶性循环更深一层的问题是：为什么会有这种怪兽程序？是技术总监失职，没有好好监控软件的品质？项目经理坐视程序败坏至此，竟不能防微杜渐？在你没有弄清楚真正的病灶并矫正它以前，就算重新修缮了现在的麻烦程序，同样的问题还是会一再发生。不要在损坏的地基上重新修筑倒塌过的房子，那是白费力气！

在这一组开发人马中有一群新成员，新的资深经理带着一群新的工程师，负责这段伤脑筋的程序代码；当然，这也为这个团队和这个程序带来新的活力和希望。他们打算重新架构这段程序，以便融入更好的技术，提高产品的使用价值。

这个小组经过一番研究之后，倾向外购一项全新的技术，作为未来开发的基石。当他们向决策当局提出这项建议时，反应是两极化的：赞同的人认为即使外购也可能有兼容性的风险，但旧程序实在太难清理不如放弃；反对的人则认为团队有能力重整旧程序，只要管理当局给他们机会。

项目经理也搞得不知如何是好。这位资深经理在情感上倾向放手让属下好好努力，去清理他们数年来不断奋斗的怪兽程序，但她又实在不放心这群属下是否真的准备好迎接这项重量级挑战，尤其是想到他们过去不怎么样的记录。她不认为这是原先管理者的“错”，虽然是他们的疏忽以致程序腐化到这般田地。

一般而言，如果管理当局对技术是压榨而非培养，优秀的人会离开，苟且的人会留下，最后组织和技术一起走下坡。

在本实例中，有不少的新人（包括那位项目经理）觉得这套旧技术实在活得太久，该被淘汰了。也许大家所缺乏的只是管理当局的承诺，包括技术生涯规划、创造力的突破和员工福利等等。这位项目经理希望她的努力可以改变这种状况。

最后，她决定做一次技术规划，然后冒着被开

除的危险充分授权给开发团队，果然不只使得怪兽程序浴火重生，而且只花了一次改版的周期时间，也扫清了许多地雷暗礁。成功的背后，她也失去了几位主张使用外购新技术的人员。

下一次她修订技术规划的时候，她不只是让开发小组参与，还设法训练主管们也能参加并了解这个决策，让负责准时完工的技术人员梦想和希望有机会呈现在技术规划之中。她也鼓励其他的主管们参加这些讨论，这样做事

的时候会比较容易沟通，也更不会错失任何好主意。

有很多方法可以制定技术规划，但最重要的原则只有一个。优良的技术规划有很多好处，也许“割爱”是最重要的。技术规划会让您知道将会走到哪里，更容易掌握该做什么，采取什么方法，如何就会更接近目标。而且，惟有完善的多版本技术规划，才能帮助您准确详估工作量。

修订技术规划

另一个我所观察到的技术规划实例也是加强共识的最佳典范。由开发小组和项目经理中派人组成的技术观察员，花了一两个月的时间起草大致的技术计划（请参考法则 5：刺探敌情）；然后这份草案被全组人分别传阅、加注意见、讨论激辩，最后被交给负责执行的开发人员（大约 80 位）。

同一时间，这份草案也在高阶主管之间传阅，他们会参照许多其他的技术规划后，定出多版本的产品计划。他们通常不会修改技术规划，因为他们知道这是所有组员的心血结晶，也是得来不易的团队共识。然而，他们发现了一个问题：这份规划看起来像是一大串互不相干的产品特色（ feature），这样蔓杂的目标怎么能激励开发人员呢？怎么可能向全世界描述他们有这么样的产品——有几十个不显眼特色的产品？

然后经理们归纳出来，这些产品特色分属于五大类型：策略、竞争、顾客满足、投资，以及他们称作“典范性”（paradigmatic）的功能诉求。典范性的功能是指有计划地放在 n 个连续版本中的产品特色，最终目的是改变使用者的工作方式。这套分类当然不是说典范性的功能不属于竞争或策略，或是策略性产品特色与顾客满足感无关，这么划分只是为了比较精确地描述。

“策略性”产品特色与最基本的环境和限制有关。譬如说是本产品支持什么操作系统？采用何种对象模式（object model）？产品需要什么等级的中央处理器（CPU）？支持哪种程序语言？此外，策略性产品特色也涵盖了公司发展其他产品线的全面性策略。例如公司卖的打印机，就必须写驱动程序去支持。“竞争性”产品特色基本上是为了对抗竞争者而设计的，特别是竞争者有而我们没有的功能。虽然不必每一项竞争者的功能都要放进来，但倘若其中有特别巧妙或特别吸引媒体注意的功能，就值得投资。这一类的产品特色是数目最少的。

“顾客满足性”产品特色基本上是大家已经耳熟能详，且顾客大都需要的。多去听听顾客的声音，产品特色就会更能搔到痒处。这部分的产品特色数目最多，但开发成本相对较低。

“投资性”产品特色是技术方面的前瞻性投资，其效益可能在未来的一或数个版本中都未必能明显看出。毋庸置疑，投资性产品特色是未来大放异彩的潜力，它能带来未来版本中的典范性产品特色。这个项目的重点是使开发人员善用每一分钱的投资。

“典范性”产品特色可以改变使用者的工作方式。基本上这是整体开发人员梦寐以求的目标，在每一次改版中放进刻意设计的巧妙功能，逐渐引导使用者。通常这是行销部门持续性对外宣传的产品特色，它甚至能改变整个游戏的竞争规则。我们可以说：如果典范性产品特色成功的话，没有人能跟你竞争——除非他也有这项特色。

在本例中，项目经理将技术规划中所有的特色加以整理，分别归入这五大类，并决定在这五大类中分别要投入多少的资源、占整体投资的比重，并

且做出相对应的技术需求分析，当然要符合内部资源和技术的分配。这一部分我就留给读者当成家庭作业，您自己思考一下，以您的条件、产业竞争型态等因素来看，五大类产品特色中，其所占资源比例应该如何分配呢？

然后，整个开发团队在接下来连续大约五天的时间，每天开会约两个小时，逐步讨论出下一版产品的技术规划的细节，整个计划就此拍板定案。这五天是大伙儿发表意见的最后机会（当然啦，事实上计划永远是可以变更的，这样做是为了强化计划的效力）。因为计划的制定是由下而上，不会有任何人跌破眼镜；因为管理阶层负责综合各方意见而形成完整的、有系统的产品特色，组织目标会被大家接受；整个过程开放而民主，每位成员都获得充分的授权，最后的争议也会最小，很能共识凝聚，热忱也会很强烈。没错，整个团队会深刻感受到共同的目标。

这是我所见过最理想的规划过程。我写本书的时候，他们的产品已经上市了，真的是很棒的产品呢！

每个人对技术都应该有一份使命感，都有追求进步的内在欲望，然而新点子可能因为缺乏适当的分析或时机不成熟而被排除于计划之外。团队成员必须明白真正的工作内涵，并且能够理解有时候新点子虽好，但总得经过慎密的分析并证明实际可行又有足够的效益才行，而好的想法若能被团队审核过关、形成共识，最终还是会实现的。如此可以减少未经深思熟虑的提案，对整个产品的稳定性颇有助益。

最后，将产品特色分为策略性、竞争性、顾客满足性、投资性、典范性这五大类的用意是，给经理人一个管理上的指针，便于监督各项资源的运用以及诊断问题。比方说，如果失去了技术上的领先地位，管理者就应该加强典范性的投资，而相对降低其他各类的投资比重。

法则 4

Don't flip the bozo bit

别做笨蛋

我再说一次，软件是智能财产。必须运用智能，才能得到软件产品。若能用更快的速度结合更多的智能，软件的智能财产价值就愈高。这是显而易见的事实。曾经有人问我：“ 在软件产业中最重要的事情是什么？ ”

我毫不犹豫地回答：“ 让大家思考。 ”

信不信由你，大部分的人都不愿意思考。他们认为自己乐于思考，但事实上并非如此。保持脑袋空空很容易，在微软我们把这种人叫作 bozo，意思是笨蛋。永远没有人会注意笨蛋的所作所为，即使他真的有贡献，他也不会有任何份量。笨蛋当然是不可信任的，你对笨蛋惟一的期望是但愿他不要搞砸事情。

然而每个人都有可能是笨蛋。你自己反省看看，是不是知道自己在做什么，是不是觉得自己一点能力都没有？小心，笨蛋可能就是你。

在我的部门里，这种德行是不允许的。我要每一个人都全心全力地投入，每个人都得有贡献，每一个人都可以侃侃而谈我们的产品——如何在市场上竞争、何时出新版本等等，而且每个人对产品的看法都一致，不会众说纷云。

判断一个人是否在思考，最简单的指针是看他是否专心倾听别人的看法，并且立即给予直指核心的响应。面对优于自己的看法，我们必须平息一

开始的那种竞争心理或防卫反应，别人当然是经过一番智力淬炼才能想得出这些，我们应该公正地评判这些新的、可能很有价值的信息。

懂得思考的人当听到诸如批评或别人比较优秀等不顺耳的话时，会把自高自大的心理干扰过滤掉，并从沟通中接收正确的信息。他们能够避免下面两种心理现象：

第一种心理现象是防卫心理，让接受信息的人无法忍受别人的批评。在创造智能财产的工作中需要很多的情绪和创意投入（像是自己心血结晶的宝贝孩子，有一份特殊的情感），别人对产品或制程的意见往往会听起来像是讽刺。懂得思考的人在三思之后，会将自我的主观意识排除，然后接受信息的真实内涵，不过这种人并不多见。

将自己的意见强行加诸于他人者，其实是笨蛋。

不懂思考的人不但不会请求别人赐教，反而在别人好意提供信息时过度防卫，而导致正面的冲突，所以无法对信息作出正确的判断，对事情毫无建设性。如果这种现象不断地发生，信息接受者会有两种可能的反应：一是这项信息确实很重要，二是这个笨蛋在将自己的意见强行加诸他人。结果呢？当然是后者。

第二种心理现象是相互排斥，发生的可能性更高。如果向别人提出建议，但对方却因为恐惧或其他负面情绪而排拒，久而久之，这位好意的建议人就会认定对方是无法沟通的笨蛋。

团队中有人无法沟通是极危险的事情，这会导致团体中人际关系的恶性循环，对团体能力造成莫大的伤害，而且几乎无法弥补。而且一旦管理者认为某人是无法沟通的笨蛋，团队中的其他人都会跟着这么认为。

对这个问题的药方是加强每个人正面的沟通能力，能够虚心接受别人的意见。如果你正试着传出信息，而对方似乎无法接受，那么换一种比较轻松的方式做做看，至少，试着向对方解释他的封闭令你难过。相反地，如果有个家伙不断灌输你“差劲的”想法或是“恶意”的“攻击”，放下自尊心彻底反省，是不是原始本能的防卫心理蒙蔽了你的判断力。如果你能在团体中实践这个准则，当你不小心犯错时就会有人及时纠正你，团队才能在和谐中进步。

死亡进行曲

在大多数的软件开发项目中，一开始多半都在做前一个项目的收尾工作。如果你的项目是以这种方式展开序幕（很不幸几乎所有的软件项目都是这样的开场），这个过程 被比喻为“死亡进行曲”（death march）。

为了赶时间，产生了太多没人能懂的程序代码。

当前一个项目拖得太久（也许是为了更前面的一个项目而晚了几个月才真正开始进行），也许因为项目经理忽略了该注意的地方，或是因为客户的强烈抱怨你食言而肥，或是前一个项目的到期压力太大而影响了软件的品质，或是虽然前一个项目还算顺利但人员筋疲力尽想暂时歇会儿，这一切的理由都埋下了延迟和品质不良的种子。为了如期完成软件，工程师明知其中有不少有错虫，明知程序写得多松散，甚至没有把握程序能够正常执行，为了赶时间只好牺牲自尊心，放弃对“我的作品”理想的执着，他们无法以自己的作品为傲，以通过软件争霸战的残酷考验而自豪。在符合进度且产出稳定的开发团队中，最后每个人都会是团队的英雄，牺牲自己无私奉献终能完成任务，事实也确是如此，软件开发是多少人绞尽脑汁的成果啊！

然而现在他们认为自己应该得到一些适当的休息，有适当的奖励或充电的机会，做些自己有兴趣的事情，或是玩玩他们心爱的计算机。

重点是，他们无论是用爬的、用滚的，还是被鞭子赶的方式，毕竟如期完成了软件，也许不是那么漂亮完美，总是达成了目标。任何有开发时间限制的软件，到最后工程师大概都是除日程之外什么都顾不了，这是非常巨大的压力。然后紧接着又一个项目开始，不知道又要投入多少脑力，这造成各种有形无形的员工的反弹，其中最可怕的是那种江郎才尽（burn-out）的感觉，这将是团队最迫切也最严重的危机。

江郎才尽

软件工程师那种“江郎才尽”的感觉，就好像巴拿马运河的建筑工人染上疟疾一般，一发不可收拾。“江郎才尽”是你再也无法承受压力的感觉，那是一种极度的疲惫和沮丧，只有软件从业人员会染上，其症状包括：

- 确信这个软件正在榨干所有的人的精力。
- 觉得这个项目管理简直乱得无可救药。
- 一想到要出下一个版本，就觉得头晕想吐。
- 对于任何企图解决问题的作法抱着愤世嫉俗的态度。
- 完全无法沟通。
- 对计算机失去兴趣。

工程师染上这种“病”时，不论《PC 周刊》（PCWeek）或《信息世界》（Infoworld）（或是 Dr. Dobb’ s 和 Midnight Engineering）等杂志看都不想看，觉得科幻小说荒诞可笑，虚拟实境（Virtual Reality）不过是人工智能的游戏，新版的 MFC 手册懒得去翻，甚至对最新款的计算机毫无兴趣。基本上，就是软件工程师那种做出最好的软件的狂热，已经消耗殆尽或是转移到别处，只剩下颓废。

其实管理者也可能发生这种“病”，这是必须特别防范的，因为发生在管理者身上的症候特别容易传染到整个团队。

对一位软件开发人员而言，计算机的狂热是他最重要的动力泉源，对某些人而言这是一种终极的自我实现。就像是笔之于诗人，颜料之于画家，程序编译器是软件开发人员心灵之所系，用以发挥才情的工具。当热情燃尽时，将自己的智能倾注于软件开发的那种无怨无悔，也就随之化为一堆肮脏的灰烬。

我和每一位软件从业人员一样，非常害怕染上这种综合症。如果不加以防范，这就像是艺术家的陨落一般，会使软件从业人员的职业生命骤然消失。

法则 5

Use scouts 刺探敌情

在开始下一个项目之前，先刺探敌情吧！

“侦察员”（scout）是军队中先派出去侦察敌情的“探子”。他们负责先了解四周的环境和敌我界线、我方资源、寻找安全的驻扎地点、确认最好的前进路线，并且随时注意任何敌方行动的迹象。长久以来，当一群人在一个危险的旅程中，通常都会先派出侦察员去了解一下前方的路况。倘若软件项目不是大批精英的危险道路，就没什么好谈的了，所以当然得派个先知先觉者去为项目事先“侦探”一番。

倘若没有人事先研究出最适合的路线，各版本的开发过程就像在沙漠中胡乱游荡。

“侦察员”必须敏锐地嗅出任何软件产业的蛛丝马迹：软件开发团队与大环境之间的依存关系、正在开发中的操作系统新版本、与本项目有关的技术发展现况（也许还得另外找人来研究这项新技术）等等。“侦察员”能建立或修订多版本的技术规划（请参阅法则 3），他们访问顾客、学习有竞争力的新技术（至少要有基本的了解），并且研究决定项目该以何种方式或路线进行。

“侦察员”必须能够提出硬件最低需求的建议书，分析所有的使用者需求及各项软件开发规范，准备原型产品，草拟未来的产品计划，以及建议下一版产品的重要特色。

“侦察员”的任务和贡献可以说是无限的。倘若没有人决定好最适合的路线，你的开发过程就会像在沙漠中迷失了方向，版本与版本之间毫无组织和明确的方向，注定了悲惨的命运。虽然有好的侦察员不能保证情报够充分，至少“侦察员”会让你不致盲目乱走。

侦察员的重要性

最近我与一家大公司的 MIS 部门有接触的机会。他们在过去三年内成功地将原本在大型主机上的终端机全部（有上千部之多）换成 PC 及 Windows 的作业系统与网络环境。可想而知这是非常艰巨的任务，需要大量的人力与金钱，和一段很长时间，整个公司在这个转换的过程中，受尽各种煎熬。

完成了 Windows 和 PC 的转换之后，MIS 部门开始发展一套分布式的应用软件，以期充分发挥新科技的效用，这是他们第一个重大的投资回收项目，预计降低的成本和增加的利润可达每年数百万美元。

这个工作团队大约由 100 个人所组成，主要是软件工程师与品保工程师，大部分都是公司的新人，或对技术还不是很熟悉，再加上对公司而言也是第一次开发这种关键性任务（mission－critical）软件，可想而知他们一定得克服无数的困难。在我与他们接触之初，这个项目的进度已经严重落后，并且大大超出预算，更糟的是眼看期限就要到了。

这项 PC 软件的项目自开始至今已经有三年了，当初他们开发软件时规划的硬件、网络协议等环境早就已经过时，目前所使用的软件开发工具和操作系统竟然落后了两个版本！这三年来，硬件价格大幅滑落，PC 效能大幅提升，所要求的硬件运算能力也提高了几倍，他们使用的 PC 已经在市场上绝迹，但运行在上面的应用软件却无法淘汰。

我实在无法帮助他们解决这个难题，只能说这个实例证明了“侦察员”的重要性。当年他们几乎是凭着直觉和谨慎保守的作风来决定软硬件环境和工具策略：“我们尽量不要改变任何东西，反正它能用就好。”倘若他们曾经好好调查一下计算机的发展趋势，倘若他们曾经考虑过兼容性的问题，他们就会知道必须使用最新的工具和最新版的操作系统，并且尽可能将未来的发展也估算在内，多花一点钱也是值得的，也不会因为当年决策错误而造成今日难以挽回的局面。

不过还好，这支勇敢的 MIS 团队毕竟最后还是圆满达成了任务，成功地完成了新的软件。这次他们学乖了，事先派了两位最优秀的组员担任“侦察员”，做了一次彻底的技术调查和完善的规划，终于在危机爆发之前将之化解。

老实说，对于类似上例的公司，我只能感到同情。计算机科技进步太快太剧烈，冲击着组织不够健全的公司，旧的系统难以割舍，新的技术又逼着你不得不进步。是的，使用第 0 版的软件也许会让你惶恐，但从第 7 版升级到第 8 版可能意味着重大的进步。如果你做决策之前，好好做过“侦察员”和完善的规划，而且知道这些对系统的重要性，你就会乐于接受科技带来的改变。

本书强调的基本观念是，我们并不企图减缓科技进步的速度，也不是要建立僵化的系统，而是要在改变中获得益处，将不断变化的科技管理得宜。行动总比停滞好，行动力高的组织会有较高的适应环境能力。“侦察员”就是为科技的改变而准备的，如果你决定永远停着不动，那你不需要“侦察员”。

当然“侦察员”也可能带来问题。如果他对多版本计划没有非常清楚的认识，他可能不知道要侦察什么；他们也可能不懂这项任务的重要性而随便做做。“侦察员”必须要非常了解自己肩上所担负的责任：一旦选错了一个操作系统或开发工具，可能会害死整个组织。

另一方面，“侦察员”本身也可能因为这项任务的重要性而过度膨胀自我，看不起其他的人，自以为是“定义未来的人”。团队中的其余成员会因此而觉得沮丧，对未来的方向完全没办法掌握，觉得自己不受重视；甚至以为事情已经多得做不完，管理者竟然把人调出去研究无关的技术：“我每周工作 70 小时，而那个家伙悠闲地在看书！”这种抱怨在派最顶尖的人去当“侦察员”时尤其会发生。

团队成员对“侦察员”的工作不免嫉妒。当“侦察员”比当开发者好——比较光鲜亮眼、比较先进、比较酷。当几个月的“侦察员”似乎真是不错，你可以观摩别人的公司、跟厂商谈谈、玩玩软件原型、趁机多充实自己的技术能力、有权力影响未来的方向。但是“侦察员”不参与软件开发，只负责做初步的规划、培养团队对于目标的共识。如果“侦察员”能够让团队信任，并凝聚出真正的共识，开发人员就比较能够自在工作，相信这些担任“侦察员”的优秀同事是在为大家的未来披荆斩棘地开路。

对开发团队来说，“侦察员”能够运用得愈成功，代表他们愈相信项目经理的领导，团队的共识愈强；不论是否处于技术转换的过程，对团体和个人都是一种很好的现象。

法则 6

Watch the ratio

注意人员的组成比例

项目经理经常犯的错误之一，是以为只要雇用软件工程师就好，其他的人都不必要，或是让软件工程师占整个团队很高的比例。也许是认为开发人员愈多，写出来的程序就愈多，这是错误的观念，项目的目的是完成软件，不是完成很多程序代码。在开发团队中，事实上有一些工作是不适宜交给软件工程师的。

在我的小组中，比例通常是 6 位开发人员，2～3 位品保人员，一位项目经理，以及两位技术文件撰写人。在微软的各个部门中，这个比例会稍有不同，也许和您的人员比例也不同；但基本原则是开发人员和品保人员的比例不超过 2：1。其实真正负责软件如期完成的是品保人员。当进度落后时，我

们第一个要看的是品保人员：人数够不够？有没有充分授权？有没有确实参与设计？进度上能不能跟开发人员配合良好？能不能一有问题出现就立刻提出警告？品保人员和开发人员的理念一致吗？是不是跟开发人员过度亲密而放水？

对于人员的组成比例应该着重于有效的人数比，而不是实际人数比。一个健全的软件开发团队一定要符合上述的人数比例原则，平均每一位品保人员所支援的开发人员不超过两位：前者是思考并监督软件的状况是否达到预期水准，后者专心写程序和抓错虫。注意人员的组成比例，可以帮助（但不保证）团队的运作取得平衡，记住，平衡才是你真正的目的。

法则 7

Use feature teams

运用特色监督小组

在微软的 C++开发小组中，我们就用过特色监督小组（feature team）的横向组织。我觉得这是非常好的做法，对整个团队的工作品质产生了绝佳的影响。

如果你问一个品保人员他的工作是什么，他会回答：“ 监督软件开发的进度，确保如期完成，并且确保品质达到预定的目标。 ”他绝不会说：“ 测试程序。 ”那是肤浅的答案。品保人员的工作是如期完成软件，是开发“ 产品 ”，是去了解顾客，而且还知道技术规划中每一件大大小小的事情进行得如何；可以说，品保人员必须要管我们的产品、我们的市场和我们的整个事业。

我们的团队组织像一个二维矩阵，传统的组织模式为经，特色监督小组为纬。以经理、品保、开发、文件四种角色，虽然是一个传统的阶层式组织，但还算相当扁平化的；除此之外，这四种角色必须各派代表参加特色监督小组，每一项产品特色都有专属的特色监督小组，以确保每项特色都能照日程做出来，这个小组必要时可自行开会。（特色（feature）一词，也有人为了听起来较大众化而译为功能，这是因为中文里的功能一词，意义非常广的缘故。事实上特色是指完成某项功能的独特方法，特别是与竞争者不同的地方，它适用的范围很广，您可以说，采用 Control－C.Control－X.Control－V 来操作“ 剪贴簿 ”是微软产品一贯的特色，您也可以说，某软件具有随插即用的特色。——译者注）

特色监督小组运作模式有几个重要因素：分别是充分授权、赋予责任、融入任务、建立共识和地位平等。特色监督小组是我所经历过最神效的组织方法，我将它的五项重要因素讨论如下：

充分授权（Empowerment）像特色监督小组这样的编制似乎比较非正式，无法像阶层式的组织那样容易管理，但它需要被充分授权才能发挥作用。譬如像开发人员，他们是“ 专才 ”，几乎专职负责某一特定的技术领域，若是让他们完全主导一项产品的设计显然太不智，如果加进一些别的角色，形成“ 通才 ”的局面，就平衡多了。单靠经理人的决定，恐怕不够全面，我看过太多好的（或坏的）决策，一下子就被另一位经理完全推翻，我倒是没见过特色监督小组把事情搞砸过。也就是说，只要特色监督小组能被充分授权，并且发挥它的作用，就可以确保决策的品质。（当然特色监督小组的必须挑

选适合的人参加，本书附录中将说明这一点。）

赋予责任（Accountability）我认为软件开发问题中最有趣的就是赋予责任这个主题。我们讨论过的题目是：“你负责什么？”我的理论是，大多数人的思考都没有好好利用到，因为人们并不把提出新想法当成自己的责任之一（请参考法则 4：别做笨蛋）。

首先，人们可能认为：“这不是我的责任范围”，因此少管闲事，不提任何有建设性的提议。如此一来会有很多不良的后果，尤其是当事情没有用语言沟通，会引起许多误解、猜忌、行动力的相互抵销，而最后只有用行动来抗议了。

只要单纯地请对方表示看法，这样的姿态很容易解除对方的心理防线。

其次，如果建议的来源真的说出了他的想法（可能要冒很大的风险），被建议的对象很可能产生防卫心理（又不是你的事情，管什么管嘛！），导致争端。有时候，防卫心理是非常隐讳的，特别是当这个组织的文化将防卫心理视为缺点时。讽刺的是，愈是成熟的团队或个人，愈会执着于现有的优良管理制度，愈会提出各种反驳的理由来证明现有的做法才是最好的，不需要什么建议。

第三，除非建议者克服自己的攻击心理，否则这个建议可能永远到不了对方的心中。就像防卫心理是普遍存在一样，攻击心理亦然。通常提出建议的人都有某种攻击性或包含价值判断的态度，这也是绝大多数的人都有的弱点，而受到批评的人会感觉到威胁而更强烈反弹，于是提出建议的人立刻下结论：这人是个笨蛋，沟通的大门立刻关上。

有很多建设性的想法就此消失。我所想到惟一的解决办法，就是所谓的研讨会（workshop）。在研讨会中人们主动上前邀请同事提出批评指教，这样单纯的提出、接受、反馈的循环中，每个人都会受到批评、提出建议，这是比较容易让人卸除心理武装的气氛，比较容易对事不对人，如此单纯的信息传达让每一个人都轻松接受建议而愿意改善。令我惊讶的是，由于研讨会的练习使得组员熟悉沟通技巧，最后反而不需要经常开研讨会了。

我有过多次运用研讨会的方式解决软件开发问题的经验。方法像是个案研究（case study），由一位志愿的组员主持；这个案例当然是现在发生的真实情况，通常是关于组内的人际关系，主持人先描述案例的大致情况，有时故意不说明当事人的名字，比方说：

——我实在无法接受这种想法。

——没有人看我的电子邮件（指建议）。

——某人根本没有把心放在项目上。

然后其他的人开始讨论，询问主持人关于此案更详细的信息，并理清一些盲点，或是询问主持人尝试过那些方式，之类的问题。最后事情会在讨论中澄清，问题也就差不多解决了。

前述的两种简单方法会产生两项很重要的结果（从来没有让我失望过），其一是这样的讨论会造成非常多的新点子伴随而生：你会看见主持人不断地说：“这个想法太棒了！”然后当场记在笔记本上；其二是这种案例研讨的结论通常可以应用在其他类似的情况，人们会发现，这就是我上个月讨论过的某案嘛！当类似的过去案例累积到相当的数量，人们就会发现本案也不是什么特例，然后就会发现处理这类事情的通则，把它记在笔记本上作为日后的参考。

在横向的特色监督小组中，一个人的成功或失败是完全透明的。

“赋予责任”有很大的好处，很少有应用上的限制。如果一个组织健全的团队成员，对于整个任务的过程——设计、开发、除错、品质、期限——彼此都负有责任，就会自然地互相给予建议和建设性的批评，而且用正面的态度接受。因为每个人都有相同的责任。一旦建立团队的责任感，这种对责任的认知会分享、传递到团队中的每一个份子。

融入任务（Identity）充分的授权和责任感，使人具有控制权和影响力，愈能使自己与任务融为一体。控制权愈大，融为一体的一致性愈高。这种融为一体的一致性是开发任何好软件的先决条件，它会将个人的心理状态健全与否，表现在软件作品中。譬如说，有一个人心中有挫败感或是自我毁灭的倾向，就会在他所负责的软件功能中表现出来。

在横向的特色监督小组中，每个人会将产品当作他的任务，而非狭隘的技术而已。没有什么人可以推诿塞责，一个人的成功或失败是完全赤裸而透明的。你不能责怪管理不良，因为你自己就是管理者，你不能责怪其他的功能支持人员没有好好配合你的产品特色，因为你们是相互负责。因此，你有责任自己找到答案，你有责任克服自己的障碍。

建立共识（Consensus）共识是特色监督小组的气氛。因为大家聚在一起的原因是“产品特色”而非功能，由于责任是互相的，敞开心胸是安全也是必要的。我见过有些特色监督小组自动地重新组织他们的关系，建立共同目标、重新规划资源、适度修改日程，而没有发生严重冲突。即使有冲突，由于不是对人的或是出于私心，通常都很容易解决，不需要主管以职权介入。

地位平等（Balance）由于特色监督小组的每一位成员都是不同的背景、专长，不同的工作角色和不同的观念，没有谁比谁优越的情形，所以每个人的地位都是平等的。通常人们对于不属于自己专业的领域，反而会有一些全新的看法，刺激彼此打破过去的观念限制或盲点，激发更好的创意。由于各人的意见来自不同的着眼点，就可以使产品的开发顾全更广的层面，而更周全、完美。不同角色的人会关心不同的问题：

项目经理：团队目前的状况如何？开发过程顺利吗？领导是否有效而得宜？我们走到开发周期的哪个阶段？进度是否控制得当？需要别部门协助的地方是否已经取得支持？我们的目标明确吗？还有什么尚未完成的工作？本周的工作主题是什么？

品保人员：我能不能依照预定的时间自开发人员手中取得阶段性程序代码？程序有什么样的错虫出现？这些错虫对付得如何？有什么功能尚未开发出来？程序的执行效能如何？本周的产品状况是否合格，需要对谁提出警告吗？项目经理知不知道程序的稳定程度？我们团队的沟通如何？是不是每一个人对产品状况都有相同的认知？本周合理的短期目标应该是什么？

开发人员：本周我能不能完成预定的工作进度，将阶段性程序代码交给品保人员和文件人员？这一段程序代码写得够好吗？使用者需不需要这个功能？这个程序容易使用吗？程序是否执行够快体积够轻巧？我清除了所有的错虫吗？有没有人因为在等待我的工作进度而耽误进度？我是否完全了解本周的短期目标？我的工作是否符合策略上的方向？而且工作内涵对技术规划有无贡献，或只是个早晚会被丢掉的垃圾？

产品经理/行销：产品特色是否吸引顾客（包括潜在顾客）？我该如何生动地表达这个产品的特色？它能否牵动顾客心中的心理或情感反应？我们应

该如何修改产品，才能加强顾客的购买欲？当顾客听到我们的产品时心里会有什么样的反应？顾客在什么时候会使用这个产品？产品特色是否能加强我们与顾客之间的关系？媒体是如何报导我们的产品？开发这项产品特色背后的动机是什么？我是否完全了解产品特色及其重要性？准时完成的可能性有多大？我是否应该向公司说明产品的不确定性有多高（或是非常确定）？产品成功推出的机会点在哪里？

技术文件/教育训练 这项功能是否以更容易的方式使用？我对它的说明够清楚吗？使用者接口能不能设计到不必学就能操作？如果我现在还没有完成文件，会不会太迟？这项产品特色是否已经通过品保人员的核准？我对这项产品特色有什么感觉？我能不能用更简洁的方式来说明？其他的组员认同我写出来的文件吗？

虽然特色监督小组是非常的重要，但要成功地组织起来可不是件容易的事。在传统的阶层式组织中加入横向的特色监督小组是非常困难的转变，这种转变上的困难可能

来自团队的内部或外部。

在特色监督小组的内部，本身就会面对相当大的不确定性，大家不知道自主权的范畴是什么，只知道我们是一组人，应该在一起做事，但不确定的该用什么样管理方式。特色监督小组的本意是突破传统角色的藩篱，但是组员却会很自然地以原来的传统角色来为自己的定位，如此特色监督小组的作用就会明显受限。开发人员常常会主导小组的运作，导致使其他的功能不容易发挥。如此一来，特色监督小组的横向组织功能大概只能改善沟通，这似乎与它的高成本（组织小组、召集开会、物色人选、行政作业等等）完全不成比例。

在很多人的心中，所谓的团队只不过是虚伪矫情的共识和假装一团和气罢了。

而且，人们总是要等到问题已经很严重时，才会成立特色监督小组。如果这个时候管理阶层放手让他们自行决定命运，组员反而会有被抛弃的沮丧感。向来不习惯自治的组员突然要组织起高难度的特色监督小组，一定会觉得非常不安。特色监督小组的功能在这样的情况之下很难发挥，问题反而很可能雪上加霜。

特色监督小组本身也充满了矛盾和冲突，因为在很多人的心中，所谓的团队只不过是虚伪矫情的共识和假装一团和气罢了。

特色监督小组惟一不可磨灭的贡献是创意，代价是无数不眠的夜、对拒绝的害怕、对个人勇气的磨炼。于是，“ 冲突 ” 便成了特色监督小组的标志，虽然它也是成长的动力。

所以，项目的初期往往看不出来特色监督小组的重要性，因为没有什么挑战性，特色监督小组看起来像是管理阶层在搞的流行玩意。

最后特色监督小组还是会发挥很大的效用，只要它能被适当的组织和赋予足够的权力。它有资源、有创意、有管理者支持，能够发挥惊人的力量，在关键时刻，他们会发现自己竟然能够搞定这么多、这么重要的事情。

当然在特色监督小组内也有它的问题。很多头脑优秀而充满创意的人不敢提出他们的构想，有一种内在的负面力量阻止他们将自己的天份表现出来。大多数人都会害怕在团体中被排斥，这种心理源自于做个乖小孩的童年经验：由于父母很少会限制小孩的发展，因此小孩对于 “ 不要这样 ” 的信息

特别敏感，为了保护自己，尽量表现得与大家一样，服从于大多数平庸者的价值体系，而不太敢表现自己的才能。然而，与众不同才是软件开发的成功要素。

这种自我设限的负面行为，虽然在一般的生活环境中是正常现象，但在正常的软件开发环境中却是病态。一般人都会为了在阶级系统中求生存而学会“善伺上意、察颜观色”，自然而然地会去迎合管理者的态度而改变自己的行为。然而，管理者也会不知不觉地发出“停止！不要这样”的信号（也许是出自盲目的策略或单纯的固执，就像父母对孩子的行为教育一般），导致属下害怕功高震主、得不偿失。管理者应该率先鼓励人们尽量发挥，对于优异的组员表示肯定和支持，至少对于不同的意见采取开放的、正面的态度。

另一方面，每个人对于这种意见自由也会有不同的反应。每个人感受到自己被充分授权，一肩挑起一件大事的成败责任，会自然地把这种情绪传达给团队中的其他成员。这种个人自由（personal liberation）在不同的时机会以不同的样貌呈现，也会以不同的方式强化，每个人都不一样，但这并不容易被观察出来。每个人都需要被挑战、被鼓励、被栽培；弹性和耐性使个人形成团体，而以自己的脚步共同成长。这种成长过程虽然艰辛，却是成为优秀智能工作团队的先决条件。

在特色监督小组中容易发生奇怪的对话，像是谁能决定什么事，除了控制之外到底该有什么样的管理角色是管理者应该担当的。

我也同时注意到，每个人对横向组织（特色监督小组）的参与程度，恰好与他原功能组织（例如系统文件等角色）的成功程度成反比。如果整体组织的功能表现不彰时，其中某个功能部门的表现还算不错，致使这个部门的表现显得突出，那么在整个组织效能改善之后，原来的疏离感反而使得这个部门的人较不容易融入横向组织中。

我们再来谈谈特色监督小组可能面临的外部问题。参与者本身所隶属的功能部门管理者（如开发经理、品保经理等等之类的各式经理）有他原来的部门目标，现在要抽调人力参加特色监督小组，当然不免有些挣扎。也许其原属的功能部门不是那么有创意，但在这“疯狂的特色监督小组”成立之前，他们对组织也有一定的贡献，或多或少都对特色监督小组抱持着对立的态度，毕竟，他们已在原来的专门领域建立起一定的责任感和权威性，现在要他们加入一个陌生的组织，做一些自己不擅长事情，而且在原本未被授权的环境中，他们凭者能力取得一席之地，但现在不需要了，却在新环境中被授权却更多，使得他们对原有的成功价值产生怀疑，因而使他们更难适应新的环境。

管理者知道如何以传统的方式推出软件产品，当他看到特色监督小组刚成立时所作的尝试，会觉得太幼稚了，会感到非常地担心：“他们这样做是错的，应该由我来做，或是由我来教他们如何做。”这真的是善意的关心，管理者无法忘记自己对产品的责任，会怀疑特色监督小组是不是疯了（我自己就曾经这么想），竟然让这些没有经验的笨蛋决定技术与产品的发展方向，并且管理者从前犯过的错误他们现在一样照犯！

管理者学习在“鼓励属下”与“控制属下”之间取得平衡，于是发生奇怪的对话，像是谁能决定什么事？除了控制之外到底该有什么样的管理角色？如果不是管理者该做的决定，那么他要如何负责？如何自处？如何定位？

当年我也曾经怀疑过，和其他多位资深经理怀疑我们为什么要大费周章地成立特色监督小组，我曾经在深夜醒来，自问我们究竟为什么要这样自找麻烦，想出这种疯狂的主意使每个人都全心投入。渐渐地，我们了解到管理者的角色应该是教导、挑战、鼓励，并肯定这个创意思考的过程。如果我们的观点是对的，事实自然能够证明它。在授权给特色监督小组的同时，经理挑战组员假设的前提，让他们能够自我检视自己的动机和行为，协助他们达成共识和化解冲突，并让他们明白管理者了解他们，会支持他们的决定。我们训练组员自己去追求效能，这是非常基本的要求，因为我们要让组员自行负责，让组员有能力决定，并且以自认最理想的方式去实现。

权威是来自学识，而非职位。

当然，这整个过程是循序渐进的，就某种意义来说，它仍然不断地在发展。有很多时候，小组希望由经理直接替他们做决定和设立目标；也有些时候是经理不当地命令小组做事。传统性的功能组织和横向特色监督小组之间，没有全然的界线。但是在这一切都在转变中，我清楚地感觉到大家都朝着正确的方向在进步。

在一个理想的项目中，基本上有两种角色存在：创造者（creator）和推动者（facilitator）。创造者是一些专业人员，例如开发程序、行销、软件品保和文件撰写；而推动者则负责凝聚团队共识和维持最佳的开发环境。在这里可以无忧无虑地尽情发挥创意，有充足的资源解决问题和实现理想，组织的运作非常有效率。推动者就是项目经理或一般的管理者，他们的能力不直接显示在产品中，但却是推动产品成功的保姆。

推动者必须像创造者一样对最后的成果负责，而创造者也必须像推动者一样对团队共识负责。让两种角色互相负责是很好的做法，可以互助互补。传统阶层式组织的重要性逐渐降低，权力来自于知识，而非地位，这是一项革命性的突破，值得所有的软件开发组织深思。

法则 8

Use program managers

项目经理的职责

项目经理是软件开发团队的一份子，他的职责是：

- 领导大家定义出一个成功的产品。
- 引导大家对产品注入深切的期望和信念。
- 带领团队将理想实现，变成可预见的产品诞生。

要定义“项目经理是什么”，倒不如从定义“项目经理不是什么”，还比较简单些。一位项目经理实际所扮演的角色并不是一般人直觉上想到的工作性质而已。

项目经理并没有（或只有很少的）正式的权力或地位，至少刚开始时是如此，所以项目经理通常会为此感到非常地焦虑。笨蛋项目经理可能以为他的工作是写出软件规格，让其它人去写程序、测试程序、撰写文件等等，最后完成项目经理心目中理想的成品。在最好的情况下，别人会认为他天真；最坏的情况下，别人会认为他愚蠢、成事不足败事有余。在项目经理可以对团队有任何的价值之前，不应该有任何直接的控制权；幸好，一个健全的开发团队不会让这种事情发生，组员会适时阻止项目经理不当地直接控制。

当然这类事件会导致在项目经理心中有些气愤和挫折，因而要求上级授予更多的权力，如果上级愚蠢到真的这么做，就会导致更严重的愚蠢事件（请参考

法则 9

要权威，不要霸权）。

通常是等到情况已经坏到无可救药时，项目经理才会明白“政治”只是合法的借口。

对于这类项目经理的挫折感，我见过很多不同类型的反应。最常见的一种是项目经理开始对他真正的角色——领导——做出负面的、反叛性的行为，这种行为通常是通过对“政治垃圾”（political bullshit）的厌恶来表现。除了对科技要有一定的知识之外，软件开发团队的领导人还得具备对人性高度敏锐的观察力，必须能够看出在团队外在行为背后那些隐藏着的情绪因素。可想而知，有些专案经理为了避免那些剪不断理还乱的人性问题，开始寻求比较“清爽”的自我形象，而在软件公司中免不了的科技挂帅文化，很容易视“政治”为畏途。当然，驾驭技术要比掌握人性要简单多了。然而，当人们抱怨政治的奸诈诡谲时，其实是对不良领导的反感所致。不幸的是这种不满的情况比比皆是，健全的组织却少得多。于是失败的项目经理把过错推到“政治”这个字眼。然而，大概要等到情况已经坏到无可救药时，专案经理才会明白“政治”只是合法的借口，他应该用领导代替控制，才懂得为自己的偏执负责。有谁愿意身处于乌烟瘴气的组织环境？但是当项目经理或任何人伤害组织的健全性时，那就是走向失败。

虽然刚开始时项目经理对自己的角色非常不确定，不能肯定自己对项目的贡献度，还对政治嗤之以鼻，不过当他了解到应该以领导代替控制时，就会觉得一切都得心应手。项目经理应该是以栽培和教育的全方位领导，来引领软件的发展。

项目经理应该要有技术的的背景，而且必须在两种层面非常专精：一是对开发产品所使用的技术很熟悉，二是拥有建构软件的技术领导能力，而后者是本书主要的讨论范围。项目经理必须精于哄骗、驱策、鼓励、要求他的团队做出最好的软件和表现出最好的工作效能，他清楚知道软件制作过程中每一项的投入和产出细节，他必须懂得用最好的方式定义产品和维持健全的技术。最后，项目经理还必须是团队的发言人，面对媒体、客户、以及整个组织。

项目经理是维系团队灵魂的关键人物，他应该是擅长沟通和倾听的人，具有设身处地为他人着想的本领。总之，项目经理是软件开发的核心人物。

团队的精神

在此我强调一个观念，在后面也会再重复提及：软件代表着创造它的团队。你想了解这个团队？看看他们的软件就知道了，反之亦然。我特别强调这个观念是因为这是软件开发管理的基础。在某一个时间点或是从某一段描述中，或是从这个团队的行为，常常无法准确判断这是个什么样的团队，但软件不会说谎。软件会忠实地展现创造它的团队：一切优点和缺点，天赋和天谴，从潜伏期的小病到最极致的团队合作。若对这个团队有任何疑问？他

们的软件作品就是答案。如果你的询问对象是人，你还得注意他的表情和肢体等等隐性语言，但是如果你仔细研究他们的软件作品，你不必问任何人就知道他们团队有没有问题。软件会自我表达，它绝不说谎，也不隐瞒。

这项理论是软件开发管理的基础，可以用一个恒等式表示：

团队= 软件

（team= software）

基本的原则是：如果你对团队有任何疑惑，去看他们的软件；如果二者一致，你可以相信自己看到的团队现状。相反地，如果软件未能表现出应有的水准，表示团队有问题，不论团队看起来多么健康，你都应该去分析、去发掘，找出问题解决掉。

团队等于软件，不错，但团队是什么呢？你是以什么方式或风格领导这个团队？我相信软件开发的关键是与“团队精神”（group psyche）保持密切的连系（Psyche 是小爱神丘比特所爱的美少女的名字，原意是灵魂或精神，这里的灵魂不是宗教上的含义，那是 soul，精神也不是强调团队的合作协调，那是 teamwork， group psyche 是指一群人所共有的中心思想和心理状态）。这样说很抽象，下面是“团队精神”的具体描述：

1.一群人同心协力，集合大家的脑力，共同创造一项智能财产。

2.个人的创造力是一种神奇的东西，源自于潜在的人类心智潜能，它被情感丰富，而被技术束缚。

3.一群人全心全意地贡献自己的创造力，结合成巨大的力量。结合的创造力由于这一群人的互动关系、彼此激荡，而更加复杂。

4.这种复杂的情况之下，领导变成像是人际互动的交响乐指挥，辅助并疏导各种微妙的人际沟通。

5.在团体中的沟通和互动是正确而健康时，能够使这一群人的力量完全结合，会产生相加相乘的效果，抵销互斥。沟通顺畅能使思想在团队中充分交流传达。

6.团队工作的品质比时程更重要，而作品的伟大是需要对“团队精神”特别加强，才能达成。“团队精神”可视为个别成员精神的平均值，而个人的精神（psyche）则是使他能感觉、能思考、能推论的内在力量。

7.倘若忽视了“团队精神”，则只会有平庸的成果。

那么，如何照顾‘团队精神’呢？我的答案就在本书中。Be an authority，not anauthority figure

要权威，不要霸权

在组织中，大多数人都会寻求一种权威，而不在乎这种管理风格好或不好。这种心理现象可能是因为人们希望重建来自家庭的权力结构：父母亲控制并保护孩子，而孩子依靠父母。无论这种需求来自何处，人们迫切地需要权力和依靠的对象，以致将领导者投射成权威的化身——也许另一方面管理者也有迫切的掌权欲望吧。

权威的目的是让每一位团队成员都有自己的专业权威，和团队的专业自信，这才是管理者真正的权威。

在任何社会中，人们总是会将群体投射到自己所熟悉自在的心理结构，这种倾向在工作团体中特别明显。人们会很自然地将工作中所遇到的他或她，比拟成过去相处过的人物，特别容易将管理者当成父母，更有甚者，团队中最优秀的人可能会将组织中的高层人物比拟成某位权威人士。这种奇怪

的不理性现象必然造成管理者有一种“霸权”的形象产生。

就像在家里一样，一个健康的组织一定会经历三个阶段：孩童期、青春期和成熟期。在团队成立之初的孩童期，组员通常会不知不觉地将管理者当成无所不能的权威人物：决定事情的人、保障资源的人、主宰奖赏和惩罚的人。而组员则早已预设了自己的角色：服膺阶级。由于他们对权威的信赖，在组员心里会投射出一大堆“应该如此”的模式到管理者的形像上，他们会说：“事情应该是这样。”并把团队与管理者之间的关系看得单纯而天真。这种管理者“应该如此”的模式包括了领袖特质、权威和成就等等，其实没什么实质上的意义，睿智的管理者应该避免浪费时间去探究在每个人心目中的权威形象。人们倾向将管理者当成像父母般的权威。

如果管理者接受了这种幼稚的权威形象，和不健康的“领导者-团队成员”关系，整个团队就永远停留在孩童期。千万记住我们的目标是让团队中的每一个份子都有权威，而不是把权威集中给管理者。管理者也许要花一点时间才能让大家接受这个观念，也许在过程中会有人抱怨，但团队一定要有对权威的正确观念才能成长。对于一个不成熟的团队，为它描绘目标比凝聚它的共识容易，你可以召集所有的人在一起，告诉他们你为他们拟定的目标是什么，这是短期的做法，长期来看，一定会有技术或产品的专家脱颖而出，主导大家对产品的目标，而管理者必须相信并支持这个目标，并且将它传递给整个团队。

充分授权

我很想找另一个字来表达“充分授权”（empower-ment）的涵义，因为这个字现在已经被用滥了。不论它用什么样的名词表达，这个观念永远是创造智能财产团队的中心价值。人们经常分不清楚准许（permissiveness）和授权（empowerment）的差异，前者是“让组员去做任何他认为最好的事”，后者是“让组员能够思考，然后尽全力去做”，完全是不同的两回事。

充分授权是让组员能够全力发挥，无所阻碍，是为他们清除各种不同的障碍，让他们以自己的力量达到某种成就。自由是充分授权的基石，让他自由判断和实行判断，自由地思考和表达他认为应该思考和表达的事情，自由地冒险尝试，而不必怕受到额外的惩罚。充分授权是充分学习的结果，而不是忽视他，让他胡搞瞎搞。当管理者对属下说：“这是你的决定”时，他必须已经提供完善的支持——训练、信息、资源——让属下能够做出正确的决定，这样才算是充分授权。否则的话，让属下做决定等于是弃他于不顾。

如果每个人都能获得充分的授权，那么在冲突发生时该怎么做决定？这是一个理论上的问题，实际上不会发生。在充分授权的环境中，不是无政府的混乱而是事实和才能领导一切，每个人都有安全感，因此而会放弃因为骄傲所造成的愚蠢、去除自我的狭隘心态，于是软件的设计开发以及决策问题就变成是单纯的资源分配运用。一个充分授权的团队有能力分析各种方法的优点和缺点，并且能够选择对目标最有利的方法，决策没有对或错，而是在产品功能、资源与时间三者之间的权衡（trade-off）。

你知道如何组织团队，辅导团队成长，由于你提供好的技术和程序等等环境，充分授权团队如期完成软件产品，不断成长再成长，这一切造就了你个人的权威。你的见识是“团队精神”、“团队作品”以及二者之间的关系。你真正的权威是来自这些事情，而不是公司赋予你的阶级或是地位，也不是属下的权威依靠需求所投射出来的虚象。但威权是人们对权力形象的需要，

那是向人们保证有人关心他，有人保护他的利益，有人让他尊敬崇拜，而不是真正的权威。

那么，真正的权威会有什么影响力？那是你知道自己在做什么，让组员了解你在做什么，并且让组员和你一起努力，大家一起为目标创造价值。

一个孩童期的团队自然不太懂得接受充分授权，有人会抱怨没有目标，缺乏明确的目标，并且不愿去做困难的决策等等。这种时候，要激发团队成长最好的方法是集中心力在软件如期完成的目标上，管理者只要对软件产品进度和品质负成败责任，让每一个人都卷起袖子干活儿，不要去理会什么阶级和分工的界线，专心使软件如期完成，将目标单纯化，排除一切干扰。管理者以这种姿态融入团队的工作，很自然会打破霸权迷信，引领团队自我成长。这是“权威”，是行动，是一起创造软件的工作，绝不是谁决定什么、谁决定奖惩的“霸权”。

竞争

在任何市场中，都会有下列四种情形之一：

没有竞争对手。

- 与竞争对手不相上下。
- 落后竞争对手。
- 领先竞争对手。

无论是那一种情况，都必须对市场未来的方向、竞争对手可能的动向以及自己的竞争地位与走向做出正确的判断。

人类学缩影

在我们开始分析这四种市场状况的不同性质和适当对策之前，让我们先花点时间思考天然竞争的经验。我想由人类学专家来谈会更好，但我觉得商场上的竞争环境在本质上是人类竞争的缩影，还可能更激烈些。在竞争中，只有胜利者和失败者两种角色。威胁、梦想、恐惧和希望萦绕在参赛者心中，在内心复杂而对立的情绪冲击之下，激发出创造的能量，使他能一心致力于创造伟大的成就。

商业竞争源自于人类对战争的原始渴望，其本质就是征服敌人、掠夺其财产和毁灭其未来，和野蛮人比起来说也许只是形式上比较美观吧！竞争就是将他人的资源据为己用，以敌人后代子孙的生活环境为代价，换取自己后代子孙的更优越的生活环境。

虽然现在的人类竞争已经不再以生命厮杀，但却发展出团队对团队的资源竞争，而且不再以猎物、财产或领地为标的。我们也许可以获致一个结论：现存的人类，是那些倾向以合作方式共同追求某项目标的人群，我们把这种合作的倾向（genetic predisposition，原意为遗传体质，是作者的譬喻）称之为“团队合作”（teamwork）。对某些人来说这个名词意味着软弱的仁善和无效的好意，但一个比较夸大的比喻是：一群邪恶的狼正在分食一只羔羊，狼群不会为争食羔羊而自相残杀，只会残杀所捕捉到的猎物。

团队侵略具有毁灭性的力量，听起来不怎么令人舒服。事实上商业的竞争行为只是千百年来人类演化活动的现代版。我们可以合理地论断，人类之所以选择团队合作能力的基因存活下来，因为这是非常有效的成功方程式，比任何其他非团队合作的策略都更有效。因而我们不得不相信这样的竞争和演化过程仍然在持续进行着。

我并不想对这项人类特质作任何的道德判断，但我个人倾向相信这种团

队合作的特质在人类演化过程中扮演着极重要的角色。文明、沟通、爱、忠诚、信赖，甚至狡滑和想像力都根源于这种特质。征服对手、弒杀猎物和夺取生存空间的欲望在人类活动中展现无遗，我们在思考群体组织和竞争情势时必须将这些因素都列入考虑。在你的团队中的所有的人们，都是曾经合作并取得胜利的人类后代。

如果我们将竞争者视为敌人，那么市场就是大家争夺的猎物，或是给企业后代的丰沛资源的土地。如果我们夺取了市场，就等于是掠夺了敌人的财产，将敌人赶走或歼灭。

我们必须了解，任何有利可图的市场都会有竞争者出现，只是时间早晚的问题。竞争者会组织他们的团队和你抗衡，争夺你的市场，拼命地想将你逐出摧毁，你 也是一样。

软件竞争

在你展开任何项目之前，都必须对竞争态势做彻底仔细的评估。在这个软件开发生存日益困难的行业中，竞争是比较私人性的，不像在传统产业中整个企业处在一个较大的领域中竞争。软件的创造是个人的思考和创意，而竞争则是个人的智能对个人的智能，其竞争态势的变化也比任何一种传统产业快得多，并且游戏规则也不太一样。传统产业是对每一张订单，比比看谁的处理行动最快最敏捷，但软件业就复杂多了。我们已经站在信息科技的尖端，某一些软件产品、技术可能在一年之内就完全改头换面。因为在产品的生命周期还没到尾声时，市场就可能已经发生根本的变化，所以很难将软件的竞争当成单一的市场来看待。因为在产品的生命周期还没到尾声时，市场就可能已经发生根本的变化，所以很难将软件的竞争当成单一的市场来看待。

因为软件业变化得太快，软件产品的全球性行销策略就必须既新颖又深入。高科技的进步是以令人窒息的加速度在奔驰，每天都有新的变化。2000 年的版本当然要比 1999 年的版本功能更强大,而向全世界宣传新版本有什么新的、高价值的特色，就成为重要而紧急的任务。

如果你无法时时掌握时代的脉搏，如果你怠于响应周围迅速而剧烈的变化，特别是竞争者的行动，如果你不能持续地创新原有的技术，永远保持领先，那么别人马上就会趁虚而入，取代你而成为市场上的优胜者，掳获顾客的心和他们的荷包。

确定了你的情况之后，应该先考虑采取标准步法。

软件的竞争

很显然，在任何竞争环境中都会有无数的方法反击对手，但我们仅讨论前面所提的四种典型。就像棋局一样，有一些开场的标准步法（classic moves），就像是对招一样，有什么样的攻击招式，用什么样的对应招式，就看你所面对的情况了。

法则 10

Alone？A market without acompetitor ain＇t

没有竞争对手？未必是好事

我们常听到某家公司认为自己在市场上没有竞争对手，因为他们有独特的产品特色、渠道，或是他们的定位独树一帜，没有人能与之相提并论。当然，在某些情况下，上述的说法可能是真的，尤其是一些新开辟的市场，或是非常专门、具有独占本质的市场。不过，在一般的情况下，这种定位并不恰当，理由如下：

首先，这种说法代表了一种错误的价值观，认为“没有竞争对手”是一件好事，有些公司夸耀自己没有遇到竞争者，并且认为这代表了灿烂的景象。我认为这个观点不一定对，如果一个市场够健康，它一定会吸引竞争者。事实上，竞争者通常会采取和你有相同功能的行销策略，“帮忙”你形成一个市场，和你同时开拓市场。

第二，如果竞争者紧追不舍，抄袭你最成功的产品特色时，你要领导市场就非常困难。但是竞争者的抄袭动作也等于是肯定了你的市场领导地位。经验证明，成为市场中的惟一会导致某种不安，不一定是好事。如果你决定投资开发一项划时代的新技术，你需要有一个可比较的标杆来证明自己有长足的进步，而且必须造成市场上的某种冲击。如果没有人想要你的市场，这个市场还有什么价值？

第三，如果你是市场中的惟一，就没有办法在顾客心中建立鲜明而独特的形象。诚然，在顾客心目中你的公司和你的产品功能会是同义词，但他们对你的技术特点还是模糊不清的。顾客既没有选择，也就没有认同。顾客如果经过比较才选择购买你的产品，他们才能识别你的特点（identification），也才能够解释他为什么做这样的选择。然而若你是市场中的惟一，顾客在心理上就没有必要特别去认识你的特征，去了解你和别人不同的地方。很多非常棒的想法之所以没有被采纳，是因为本来就已经是惟一了，没有可比较的对象来显示它的优异。每一项采购都是一种大胆的冒险。虽然在导入软件的初期，厂商可以和顾客密切地合作，共同努力，并建立良好的关系。但大多数的情况是：顾客如果在市场真正起飞之前购买惟一的产品，最后都会感到失望和幻灭。

市场中的独占者，反而很难向顾客传递独特的产品信息。

当然，我们前面提过了，市场中的惟一有两种情况：一种是新兴市场（nascent market）。为了成为市场先驱，快速扩张市场占有率是每一个软件公司的梦想；然而很不幸地，要在一片处女市场中开疆拓土需要庞大的投资，风险也高，大概只有最大也是财务最健全的公司才会考虑，或者是那些恰好早就已经在顾客心目中占有一席之地的公司。

开拓新兴市场至少会有两个层面的问题:首先,如何将信息传达给顾客？新功能和新技术通常很复杂，很难解释清楚，必须做过相当大幅度的浓缩和摘要，才有可能清楚地告诉顾客。除非你有一位沟通天才，再加上顾客正好有明显且紧急的需要，而你的产品正中红心，否则你的行销费用将和效果将不成比例。

第二个层面的问题是，通常新产品都需要大量的行销活动，时间拖得愈长，胜算就愈低。在你经过包装、塑造产品形象、定价、建立渠道和一般性的沟通之后，你已经没有多少预算了，也许撑不到第二回合。

对于独占的竞争态势，不需要再多谈了，如果你没有竞争对手，你应该知道，而且你大概也晓得该如何维护你的市场。

法则 11

Dead beat？ Break outof a feature shoot-out

竞争者紧追不舍？推出创新的功能特色

在白热化竞争的市场态势中，尤其是当第一名尚未确定时，大家都会竞相推出创新的产品功能，以显示与对手的差异并吸引顾客的青睐。此时评估产品优劣的重点是看谁的产品有什么功能特色，以及品质的高下程度。你大概听过 check-box（条列出数个选项，使用者在需要的选项打上一个小叉）的勾选方式吧，类似这种的产品功能特色，常常用来当作评比的项目，看看市场中的产品谁有/谁没有这个特色。结果是免不了开发一些花俏的产品特色，只是为了满足媒体评论者，或是某些经销商的要求，让他们觉得这个产品不落人后。

这种竞争方式的成本很高，也没有效率。

愈来愈多的产品功能特色，会使软件庞大而脆弱。

如果撇开定价、渠道、服务等因素不谈，纯粹就软件本身来看，有两种主要的策略可以赢得产品特色的竞赛（feature shoot-out）。一种方式是你推出遥遥领先竞争者的产品特色，使他在一两个产品周期内都无法赶上。这是蛮力的方式（brute-force approach）。这个策略的困难是，由于你的创新必须有大幅度的跃进，产品的复杂度会明显飙升，很难以简明而有力的方式向顾客说明这个产品特色如何使用、有何好处等等，而且功能特色会一次比一次更难做，软件产品会因此庞大而脆弱，稳定性不易维持，软件愈庞大，改版就需要愈长的时间，这是无法避免的。

当你采取这种大幅跃进的功能特色作为竞争策略时，必须是你的竞争者也采行相同的策略，而且你能确定他的特色无法跟你相比，而且在短时间内也无法赶上，这种方法值得一试，但不是最好的方法，而且这样做并不能创造出伟大的软件。

另一种能够赢得产品功能特色竞赛的策略是大量投资在典范性的功能特色（paradigm-shifting features）上，而且功能特色要多到让对手无法在这个典范性领域中与你一较长短，通常至少要有三项才够安全。一旦你建立了在这个典范性领域中的王朝，你的对手就会暂时被逼退，即使对方原本略胜你一筹。缺少几个竞争者有的（也许华而不实）产品特色无所谓，只要你真的改变了使用者的工作方式，赢的何止千百倍？最重要的是确定真的能改变使用者的工作方式，否则只不过是在产品特色竞赛中平添一笔罢了。典型性功能特色是你决胜的一击

法则 12

Behind？Ship moreoften with new stuffs

落后竞争对手？加大投入，更快推出新版本

软件业中没有两个完全相同的失败。产品失败有数不清个可能因素，但是最常见的莫过于新版本无法跟上对手的脚步。如果对手领先你一两个版本周期（release cycles），那你就非常危险了，而且非得彻底检讨、重新定位不可。

如果上帝对你够仁慈，让你活着，你还有机会继续奋战。

集中火力强化产品本身需要大量的成本，而且你可能因为在短时间内要开发这么多新功能，而使得团队乱了脚步，这个结果的危险更大，绝对足以使任何种类的软件产品被三振出局。因为落后对手，团队的士气已经痛苦不堪。但这剂猛药的风险非常高，就像化学治疗一样，治疗可能比疾病更伤人。

发现自己处于下风实在令人沮丧，媒体、上级、同事、顾客、全世界都知道你的表现不佳。这是非常可怕的感觉，而且结果也不晓得会如何。

如果上帝对你够仁慈，让你活着，你还有机会继续奋战。但痛苦的煎熬还在后头。特别是你必须向全世界宣示你打死不退的决心。这一点非常重要，因为没有人会买软件开发公司可能会放弃的产品。

省下和媒体说明的精力，就让他们批评吧！反正无论如何辩解，他们都是照批不误。不如专心集结所有的资源，好好准备下一场战役，你只是输了一着，并没有输掉全局。保留实力，你还是有赢的机会。

不要提前宣布计划。虽然你需要宣传，需要挽回面子，也需要给老顾客一线希望，但是绝对不要在产品完工前放话出去。这个举动太危险，等于是降低了产品推出时的震撼效果。

对自己和对团队内部都要承认失败的事实，不要粉饰太平（对别人就不必提了）。确实找出最重要的毛病，把它实时修正，然后重新出发。

如果问题是软件的错虫太严重，当然得在下一版之前全力大扫荡，至少一定要消灭掉最主要的错虫。这是非常艰巨耗时的工作，可能会吸干你的资源。即便如此，也尽量不要向顾客收取除虫版的费用，虫虫危机不是顾客的错。

我认为取得市场领导地位的要诀只有一个词：残忍（relentless）。你必须在追求卓越时残忍，也在展开攻击时残忍。最有效的战术是比对手更快速地推出新产品。这个观念是：让你的顾客对你有信心，显示你旺盛的企图。如果软件竞争是一场比赛，顾客就是观众。观众喜欢看精彩的对决，而且他们有权决定谁输谁赢。所以，站起来吧！即使战局不利，向顾客展现你的实力，看你如何扳回一城。

如果你的团队能够持续地比竞争对手更快推出新产品，并且将投入的成本大约控制在合理的程度，那你终究会赢！产品推出是最困难的事情，如果你做得比竞争者好，那你很有希望在别的地方也表现得比他好。准时地、经常地推出新产品是软件开发产业中最大的金科玉律。

法则 13

Ahead？Don＇t everlook back

领先竞争对手？不要回头

已经领先竞争对手？恭喜你终于可以称霸群雄，现在你遇到的困难是继续领导这个产品领域的方向，开拓新的技术以及伴随而来的新机会。自满是你最大的敌人，请牢记下面几件事：

你无法永远保持领先。随时有人在伺机攻击你，你不知道敌人是谁？用什么方法？在什么时候？

你必须与自己竞争。现在你该把典范性功能特色的投资增加为原来的三倍，每一次改版的内容要更丰富（虽然你已经拿下冠军宝座），建立你更高层次的冠军地位。

让每一个人都知道你勇往直前，绝不回头。

将改变的速度提高到别人无法赶上的程度。身为市场领袖，你有机会自由决定自己的日程和更新版本的速度。照理说，你的获利要比别人都高，你应该增加投资更稳固你的领先地位。打下江山固然困难，守住基业更不容易。记住，如果让你的竞争者拿下推出速度的控制权，那你很可能就会丧失领先地位，至少被迫做出相对的响应。高科技产品的生命周期是以月来计算，而不是以年来计算，而产品的世代交替是这么地快速，随时都可能一夕之间山河变色，所以你一定要保持自己迅捷的行动能力。

要有破釜沉舟的决心。巩固现有的地位之外，你必须继续朝更高的目标挑战。让每一个人都知道你勇往直前，绝不回头。你不能被现状限制了向前的脚步，兼容性是重要的，在你向前直奔的同时，保证产品的兼容性，让顾客跟着你一起向前走，而不是被顾客或其他人拖住。

法则 14

Takethe Oxygen along

保持新鲜

当你决定为个人计算机开发软件产品，特别是在微软的视窗坏境之下时，你必须有一种认识：你不只是在开发软件，你在选择一种生活哲学。它改变的速度快得令人窒息，就像是科技和社会的变迁一样。快到在你还没有适应过来时就得再更新。分析、开发、建置、维护的过程再也不是一套操作化的公式。

操作系统的全球化快速改版已经是信息时代的基本特质，这驱使了软件不停地改变。不只是软件，包括硬件、外围设备和大众通信的厂商都因为操作系统的版本更迭交替而受到很大的影响。这种快速变迁的节奏是信息社会的常态，也是你不能与之对抗的。

快速变迁的节奏是信息社会的常态，你必须快速前进，否则就落伍了。

对全球化的信息科技作出迅捷的反应是组织健全的基本条件，而不是特殊功勋。软件包厂商将他们的技术包装在产品中卖给顾客，而产品本身就是最主要的沟通工具。为了解决现在的问题，厂商开发下一个版本，这是持续性的购买行为，不是买过一次就算了。

操作系统厂商频频推出新版本，并不是为了自己的好处（虽然这样做对他很有利），最主要的目的还是将新的科技纳入产品，让顾客都能使用。而如果在 PC 上开发应用软件的厂商跳过一个操作系统版本,顾客就得不到使用新版操作系统的好处，比如说速度无法提升或是觉得用到了次级产品等等。

不要忽视潮流的力量，不要错过潮流。想想看你对汽车、时装或是流行音乐的追求和感受，就能明白潮流是不可抵挡。你的顾客也会喜欢最新、最方便的科技。所以，将潮流纳入你的产品开发计划中吧！吸收最新鲜的氧气顾 客

由于大部分的软件购买行为都是顾客对既有软件购买新版本（repeat business），因此绝大部分的软件公司都必须维持与顾客的长期良好关系，然而这并不容易。我们不讨论买过一次就了事的那种软件购买行为。基本上，身处在这快速进步的信息科技之中，顾客总会需要再去更新软件的。这种顾客一供货商之间的“ 持续关系 ”（ongoingness），在软件业中特别密切、复

杂，也特别重要。

在软件业中的顾客-供货商关系之所以复杂到惊人的程度,主要是因为顾客花很长的时间在使用软件，而且研究得很深入。就拿我的 Visual C++来说，我敢说我的顾客每天与它相处的时间超过 8 小时，而且我的顾客待在虚拟世界的时间也比在现实世界要长。他与我的产品相处的时间超过他与家人相处的时间，这是很值得注意的事情！

第二个使顾客-供货商关系非常复杂的原因是：软件不只是耗费他的时间，还对他的潜力造成限制。软件为了做到自动化和执行效率，必须事先定义很多的限制条件，而这也就同时限制了使用者的表现自由。

被软件俘虏的顾客

如果顾客迷恋你的产品，请对以下的讨论特别注意。你的顾客选择你的软件，是基于心理和情绪性的因素，而不是基于做事的需要。顾客购买你的产品时，好像跟自家人买东西一样，就像是对偶像的崇拜一般。这时候你得特别小心，这种顾客-供货商关系特别容易出问题。

顾客是非理性地选择你，并没有事先比较过其他的产品。一旦热情消退，他就会觉得挫折和憎恨，而且会不公平地感受你的软件的优缺点。你必须补偿他们因为缺乏选择所造成的遗憾；你必须让他们心甘情愿地选择你。即便现在是被你的软件俘虏，但不是已经误上贼船，不得不选择你。在这种情况下，一开始你是处于负面情况，你得让顾客觉得不后悔他的决定。

软件的一切动作都是经过设计的（多多少少都找过专家咨询）。设计者是找出最正常的程序，归纳出一组标准动作，然后选择要自动化的功能，限定它的输入和输出应该遵循什么规则，应该在什么机器上执行，要用什么软件来做。设计者的目标使用者是大多数的普通人，设计者希望能让最多的人获得最大的满足。这种做法是没错，但却令少数人觉得受到挫折。

虽然使用者花在软件上的时间那么多，但他对设计的影响力却少得可怜。软件的开发就像是黑箱作业，有一大堆的信息放进去，建筑师（产品设计师）从其中找出人数最多数的类型，考虑其住房需求，而不直接与其中的住户对谈。这种产品是为了服务市场，而不是服务顾客，这种做法实际上剥夺了个人无穷的潜力。

这种不平衡是必要的，但确实使得客商关系复杂化。没错，每个人都可以说，软件公司不去开发足够的软件工具（也许是技术和时间因素），而专注在开发自己的应用软件，软件公司总是做一个自己认为的典型，却不让顾客有使用上的自由。自制软件的变化性，对照于市售软件的笨拙，使个人更增添挫折感。虽然每个人都喜欢以自己的方式做事，但大部分的软件都要求使用者用特别的方式才能满足一点个性。好吧，只要有办法做得到，使用者勉强觉得比较满意。应用软件厂商当然可以对于这一点加强努力，让那些不满足设定标准的人，有自己的弹性空间。

没有它万万不行，但有了它我还是不能做我想做的事。

然而，因为个人化的差异而在软件工具或应用软件中加入多样化的组态，也可能反过来使软件复杂到无法被大多数人接受的地步。无法接受的复杂度不一定是人工操作组态所造成，但二者常常伴随发生。因为组态是很难设定的，人为的尝试设定常常失败。说得更具体些，就是提供弹性的结果，通常是让很多人在有意或无意间碰到了组态的问题，然后搞得是一团乱，大家都不满意。“ 容易使用 ”和“ 便于修改 ”两者之间在本质上就存在着冲突

和对立。如果你能舒缓这个问题，就可以满足更大范围的顾客需求。让软件容易使用是一大学问，而要让软件能够动态运用更是难上加难。如果你能让它变得简单些，就很容易吸引那些“改变需求就是我的需求”的顾客。

在这个时间和资源有限的世界中，软件厂商必须选择适当的功能特色来开发，如此就必然限制了产品能够给予使用者挥洒的空间。愈来愈多的使用者对软件上的限制觉得懊恼，也许可以应付每天例行性的工作，但绝不能算是得心应手。软件使用者最后只能很痛苦地迁就现实，每天都在用，这么拘束难受，但不靠它根本行不通。就好像每次穿鞋时都用鞋拔硬挤一下把脚塞进去——姑且称它为“鞋拔现象”吧。

我经常以软件开发为题发表演讲，每当谈起软件与顾客的关系时，总是会放一张投影片，上面写着：“大部分的软件都烂毙了”（Most Software Sucks），每一次都使听众大笑不已，最后掌声不断。我不得不承认，现在的软件虽然令人如此不满，但总比没有好，所以才会创造出数十亿美元的财富吧！但是有多少人称赞它呢？

我买软件，也喜欢它们（刚开始的时候），只因为有总比没有好。但是愈用愈觉得恨透了这些限制，比我自己写还不适用。我自己重新安排一下设定，还是不行，于是我很自然地决定更换软件，结果全世界都改变了，这个新软件变得更糟，我却得倚靠这个无名孤儿般的软件。我真是烦死了这些！亲爱的软件厂商们，求你们让我有机会升级吧，没有它我做不成事情，但有了它我还是不能做我想要做的事情。

顾客购买模式

从顾客的角度来看，一个简化的软件购买心理和情绪过程分为几个阶段，大致上是依序的：首先，他们会收到厂商的信息，引起对产品的注意（attentive），然后开始对软件发生兴趣（interested），想对它多了解一些，最后他相信（convinced）买下它是个正确的决定。当然啦，如果他一开始就对产品信息充耳不闻，他也就不会有后面的心理变化。但即使顾客在心理上已经相信，但他还不一定会买，他一定要对产品有渴望（desire），才会促成购买行为。

你要让顾客渴望你的产品，从一开始设计时就必须考虑这一点。想象一下你的软件有什么特质“properties”是能够让使用者觉得满足，把它们具体化一些，你自己先“渴望”它们，把它们鲜明地表现出来。个别的功能可以妥协，但这个中心特质一定要强化。对这些特质思考再思考，回顾一下软件的各个版本对这些特质做到什么程度，设法使它深入到程序的每一个角落，一定要让顾客能够明显感受到这些特质才行。你得让任何人在初次使用时都能体会它，并且说得出这个特质是什么。

软件的美学

让我们在这里稍微离题一下，谈谈美学（esthetics）。大部分的人都常常听到这个字，但是只有很模糊的概念，觉得这大概是有关艺术或设计的一种知识。也许从反面来谈美学（anesthetic，麻醉之意）会比较容易掌握它的基本精神，那些让我们提不起劲或想睡觉的东西，就不是美的东西。所以，美的东西会让人的精神为之一振，感觉变得敏锐，或是神清气爽。

什么东西会使人振奋呢？答案是容易感受得到的东西，如形状、颜色、声音、动作、和谐感等等，这些特质在无形中都会吸引人们的注意力。设计时加入了美感的软件，能够让人的心情愉悦，让人在使用时感受比较舒适。

美感的提振作用虽然是纯心理方面的，但会影响到使用者实际的行为。让你的软件创造出美的感受，会有更大的使用价值和市场潜力。

法则 15

Enrapture the customer

给顾客惊喜

大部分的顾客会购买既有软件的新版本，经过一段长时间与软件相处，他知道这个软件一切的优缺点。而且由于行销部门不断地宣传你的产品和服务，市场非常了解你的公司。总而言之，你的顾客认识你。

如果顾客对于自己必须倚赖你那不太能满足他的软件，他会感到愈来愈不舒服，他每天的软件生活都是处在“鞋拔现象”中，他一定殷殷盼望你在下一版能够改善并解救他，哪怕鞋子只是宽一点点也好。

成功地符合顾客的期望是和那些怒发冲冠又穷凶极恶的顾客沟通，了解他们对每一版的不满在那里。如果你对顾客忠实，愿意倾听他们的声音，采纳他们的意见，他们就会愿意与你走在一起（虽然他们可能仍怀着愠怒的心情）。譬如说，如果你的系统中有一个执行太麻烦，使得顾客痛苦不堪，那就在下一版把它改善。顾客最低的希望是你能够理解他所感受到的痛苦经验，并且你必须在下一版中表现出来。

顾客最低的希望是你能够理解他所感受到的痛苦经验。

前面所述是正常的成功，伟大的产品当然不止于此，而是应该把技术的轴心放在顾客心底最深处的需要，不是他们最低的期望。你必须用创新的方式，满足顾客那些难以表达的需求，用你的产品告诉他们你深深了解，包括那些萦绕在潜意识里的念头，给他们惊喜和感动，扫除那些令人愤怒的缺点，让软件完美地配合他们的需求。

伟大的软件在市场上会得到证明。伟大的软件一版又一版地淬炼她的完美，你会藉由她告诉顾客你多么了解他们，她会吸引无数顾客的忠心。你会被顾客当成力量的来源，顾客因此而狂喜。

问题自然又来了，你如何知道是什么令顾客困扰？更重要的是，如何了解他的内心深处的困扰？答案很简单，直接去问他们，任何时候任何地方，以任何形式，通过任何渠道。

市场调查和统计数字会告诉你市场的偏好，帮助你决定未来的产品方向，给你明确的数据。然而，如果你想要和顾客建立良好的关系，是比较艺术性的，而不是科学性的，而且你想知道重要的事情，那么以下的讨论会告诉你几个比较容易的方式可以了解顾客。

套句广告词儿，就是“just do it”。寄信给你的顾客问他们喜欢什么或讨厌什么；打电话给他们；在信息展的会场与顾客聊天；组织一个“使用者联谊会”（focus group）。没有一种方式极复杂或很花钱。使用者联谊会只要有个场地，邀请一些不太远的顾客，就可以坐下来谈很多事，你做点笔记整理一下，就可以得到深入的意见。做一次市调，研究一下我们需要知道什么问题，用问卷做调查，就可以得到广泛的意见。

你不必完全照上述方法执行。在现实世界中的软件开发团队，一定对收集到的顾客意见争论不完（到底顾客的实质意义是什么，我们如何将顾客意见落实在软件中等等）。这很正常，也是好现象，可以鼓励他们这么做。你

可以继续追踪顾客的需求，记住市场是愈来愈复杂的，了解顾客需求是一件持续性的工作，不是做完就结案的短期任务。

了解顾客对软件的需求是需要技巧、创意和持续不断的努力。你必须认识市场的核心需求，将技术和沟通等资源集中来满足它。了解核心需求对软件产品有下面的效果：

·产品将带给顾客安全感。

·产品会让顾客能掌握得更多。·如果你的产品在其他方面不如竞争者，但在核心需求上绝不辜负使用者，你的产品还是很有希望会赢。

·你的产品诉求信息非常清楚。

·你的产品在使用上更简单易学。

法则 16

Find the sweet spot

寻找靶心

我相信任每一个市场都有它的靶心，靶心就是各方人士价值观的交集。靶心永远随着市场的成长而移动，有些时候还移动得飞快。如果你的产品能够抓住大部分人所认定的重点，也就是正中红心，因此只要发表得当，就一定会大受市场欢迎。

当我们开发 Visual C++ 1.0 时，大家都很顺理成章地认为 C++一定会成为市场主流。因为主要的经销商都把 C 和 C++一起搭售，所以无法在市场的统计数字中得知：纯就 C++而言它被市场接纳的程度；我们只能假设，由于这一组产品主要是以 C++编译器做诉求,因此这个庞大的销售数字代表了 C++被大量的顾客采用。

有趣的是，当时的分析师、媒体和顾客都一致告诉我们，市场需要我们把“模板”（template）和“例外处理”（exception handling）放在产品中。这在当时对 C++而言还是相当新的功能特色，有非常多的理由显示这是很不容易设计，而且我们在那样的时间限制之下，几乎没办法满足这项要求，而且我也怀疑顾客是不是真的需要我们做这些。我不断问自己：“为什么我要把这么复杂的特色纳入产品中，而我敢打赌几乎没有人会用到这么复杂的功能？”事实上这正是行销的理论基础，追求独特，如此才能摆脱传统的束缚。

在收集到的市场情报中，找出奇特或不甚合理的地方，由此切入，摆脱传统思维的束缚。

我们做了一些研究，结果是只有不到 15%的人从 C 转移到 C++，虽然有 80%的人计划要，或是想要，或是希望在明年转移。所以我们开始与顾客面对面沟通，企图了解他们真正的需要。通常顾客不会告诉你他真正想要什么，特别是答案与传统背道而驰的时候。因为他们欠缺安全感，他们会告诉你他以为他要的。这也是行销的理论之一：顾客恐惧新科技，而开发人员也一样恐惧新科技。

如果你走进软件开发者的工作室，问他们：“你们之中有多少人学不会 C++？”我想一定没有人敢举手站起来大声说：“我！我学不会，它实在太难了。”他们一定会这样说：“哦，我们还在等待模板和例外处理。”或是其他的借口。所以，你必须更深入挖掘顾客内心真正的需要，而不是这种表象

式的纯技术方面的功能特色。

真正有效的做法是，私下问他：“是不是在使用 C++时遇到了困难呢？”

这时候顾客可能会说出心底话：“是啊，我实在没空学 C++，这玩意儿太复杂了，我真的是被它给打败了。我连 Windows 程序都不是很会，虽然我已经试着读一两本这方面的书，我也看过入门录像带，那太粗浅了，我还是没办法对它有概念；但是所有的人都会这东西，我落后了，怎么办？我真的好害怕。”

法则 17

It's a relationship，nota sale

与顾客建立关系，而不是卖产品

在当时，因为 C++的观念太新，显然不太可能用这么复杂的程序语言来把我们的市场整个开展起来。C++实在太难了，顾客不知道该从何着手来学习它，而 Windows 程序也是，很多人都在为此挣扎。他们觉得真要弄出一个不错的应用软件，非得付出大量的努力不可。

于是本小组中有一位聪明人士，想出了一个聪明点子：精灵（wizard）。现在看来它也没什么了不起，它不是什么仙女棒，只不过是一个可以产生一小段程序代码的按钮罢了。在概念上来说，精灵差不多是我们所做过的功能特色中最容易的（虽然它的基础 MFC，相当困难）。但是精灵却是正中靶心，它让无数的程序设计师只要按一下按钮就产生出一堆基础程序代码。

我们把精灵放进产品之后，Visual C++的市场占有率明显窜升了数十个百分点。由 Visual C++的例子可以看出来，高深的技术应用并不是重点，重点是帮助你的顾客，找出他们真正的需要，他们的内心在为什么事情挣扎，然后把所有的注意力转向这里。不必管模板和例外处理，不必管其他的高科技名词，只要全心全意开发大多数人真正需要的东西就够了。

顾客需要被了解，虽然他很可能说不出口，你应该体会他真正的需要，并且以软件产品表现出你对顾客的了解。将技术的轴心放在这里，做出顾客梦寐以求的东西。在 Visual C++的例子中，顾客的愿望是：“我想成为 C++的程序设计高手，成为年薪 9 万美元的高级工程师。如果你让我实现这个愿望，而不必学得那么累，我一定花钱买你的产品，并且叫我的亲朋好友一起用你的产品，我一定要得到它！如果公司不买我就自掏腰包，如果我的另一半不同意我就跟她争论到底，我会不惜任何代价买到它。”

“我一定要买这个软件！甚至不惜和老婆吵架，我愿意付出一切代价买它。”

身为软件厂商，一定要牢牢记住顾客需要安全感，要能够控制软件，要软件帮助他事半功倍。顾客不愿意承认他无法跟上新科技，顾客不会说：“你的软件烂毙了。”他会想：“一定是我太笨了。”一般人除非相信自己聪明到足以掌握科技，否则对科技都怀有畏惧和逃避。你不能等着竞争者使顾客聪明。

你和顾客的关系就像是舞伴。你向前一步（推出新产品和传达新信息），然后顾客响应一步，然后你跳下一步；你必须注意整体发展，而不是刚刚那一步。这就是我在法则 3 中提到的多版本计划。如果顾客知道你会按时出新版，而且在这漫长的科技旅途中你都会忠实地伴随他，那他就会在下一步时

比较能够配合你，跟着你的脚步舞得更流畅优美。

你和顾客的关系也像情侣。如果你忽略他们太久，或是表现出兴趣缺乏或者是拒绝，当然你们的关系就没法子融洽。你的忽视会使顾客觉得受到背叛、伤害，觉得生气，恨不得给你一点报应。所以，一定要对顾客“一路走来，始终如一”。

我一定是太笨了

不久前我在飞机上与一位女士聊天，她问我在哪儿工作。通常人家听到我在微软工作，不外乎两种反应，一是露出很崇拜的样子，一是直接问我认不认识比尔·盖茨。

这位女士问过比尔·盖茨的问题之后，开始和我谈计算机，告诉我她学计算机的计划。她说她要去一间社区大学上课，要学所有的计算机软件，包括数据库、电子表格和文字处理器。谈到这些，她脸上流露着兴奋的光芒：“计算机实在太棒了，我要学会所有的东西，Windows 啦、RAM 啦，反正所有的东西，要花几年也无所谓。”

听到这位女士的梦想，我对她面对计算机革命的态度有两种感想。我钦佩她的勇气、决心和远见，她不愿被科技抛在后头，错过这本世纪最重要的文化与技术核心。她愿意投注大量的时间、金钱和努力去学习计算机，不怕承认自己不会计算机的事实。

咦？等等，这位女士真的认为计算机很难，计算机真的那么难吗？她需要回到学校，才能够学会操作 PC 的软件。而事实上这些软件实在非常地容易。这么说，这位女士一定很笨啰，还得重新回去做老学生，忍受数不清的挫折，只为了学计算机。嘿！这可真不错，这就是我们软件厂商的大生意，把软件弄得很难表示我们智商高人一等，然后顾客会付我们白花花的银子来学软件。即使我们的软件很烂，顾客也只会觉得自己太笨或计算机太难。

既然你是本书的读者，应该已经具备相当程度的专业知识，而且你的亲朋好友都知道你懂计算机。请回想一下，你是不是常常遇到有人困窘羞怯地告诉你自己实在是计算机白痴呢？这是相同的现象。大部分的人都觉得不懂计算机一定是自己太笨的缘故，而感到极度不安。人们的错觉是：计算机是对的，自己是错的。

法则 18

Cycle rapidly

加速产品推出的周期

我们已经谈过了准时推出软件产品，兼顾进度和品质。除此之外，你还得加快新版本推出的速度，这么做的理由有几个：第一，无论你做得多快，市场、科技的进步和竞争者都会比你更快。第二，你与顾客之间的关系是维系在你多勤快地与他们沟通，沟通的主要媒介就是产品的新版本。你要告诉顾客的每一件事都包含在软件产品之中：你对顾客的细心体贴、你的热情、你的信念、你对顾客的看法，往往都由软件表现，软件是不会说谎的，她是你（和你的公司）的代言人。

我从来没有听说过有任何的关系会被太多的诚恳沟通而破坏。借着频频推出新版，不只是使你在市场中的曝光率增加，也等于是回馈顾客，拉近与顾客之间的距离。钞票也不会说谎，常常推出新版会使你与顾客的交易机会

增加，也使得双方的关系更密切。

Visual C++团队具有在短时间内推出产品的丰富经验，所以我们在这方面做得很成功，以后还要继续。这是我们强大的竞争武器，也表示我们能与市场维持良好的互动关系。现在我们甚至能够用预订的方式卖软件，一年推出三次新版本。我认为这是贩卖软件的最好方式，但是有两个问题：第一是很少有厂商能够保证自己规律而快速地推出软件；第二是软件更新往往需要厂商付出大量的成本，而且很难管理得当。不过我相信现在的软件厂商愈来愈能够处理这个问题，再加上 PC 和网络的进步成熟，我认为这种方式会逐渐普及的。

站在顾客的立场，他不一定每一个版本都得使用，他可能已经习惯于旧版本的环境，或是采购预算的限制，或是已经在旧版本上做了一些投资（比方说开发了一些应用软件）。不论理由如何，顾客永远有权说要或不要。顾客需要做自己的技术、预算或时间规划，他需要预估你何时推出新版本，而届时他就可以使用市面上最先进的技术。不幸的是，现在的软件大多是非常不定期地推出新版，而在新版中会有重大的变革或是大幅度的除虫。

经常而准时地推出新版（修改规模比较小），比不定期的一次出个革命版本效果要好得多，而且利润也比较高。开发软件最重要的是对市场能够迅速反应，而不见得要有革命性的突破（responsive but revolutionary）。定期推出新版让你有机会直接响应顾客的需求，同时很快地扫除令顾客烦恼的障碍（也许只是个小瑕疵），而顾客会对你心存感激。我发现消除顾客的烦恼（Annoyance fix）实在是一本万利，通常不需要太多的技术难度，有时候只不过是一点点使用习惯的问题。这会让顾客相信你会替他解决问题的，大大增加了他的安全感。

我发现顾客比较喜欢新产品摆在他眼前，而不喜欢厂商的承诺，有少部分的顾客是两者都要。但一般来说，持续推出新产品才是对顾客最好的保证。送顾客纪念品只能让他高兴一下，不会让他想买你的软件。

完成很多个小成就，比完成一个大计划，还要受人欢迎。因为这种方式比较容易切合使用者的需要。按时推出新版软件会让你设定比较小的目标，对这个目标更清楚而且容易掌握，比起很大而不确定性甚高的目标来说，也比较好管理。设计

对于一个伟大的软件而言，最重要的是在正确的时机，推出正确的产品。也就是说，你必须知道如何准时推出软件，而且能够抓住顾客内心深处的需要，这就愈能够体会到顾客的内心，这个软件就愈伟大。软件的设计——每一位团队成员都必须参与——这表示团队整体对功能需求的了解程度。总而言之，软件设计的第一要诀是：将全团队中最好的想法组织起来，去满足顾客内心最深处的需要。

在设计过程中，最困难的是让每一个人最好的想法或建议、最棒的创意或灵感表现出来。但是这样做的代价是绝对值得的。想想看：只有两三个人做产品设计的话，加起来的智商最高只有 230，或是 250，有 10 个人的话，结合的智商也许可以到 1300。算一下你的团队有多少人，你能结合的智商愈高，做出伟大软件的潜力就愈高。在设计或开发过程中的每一个阶段，让创意充分发挥是非常重要的。一个伟大的教堂，只需要一位伟大的建筑师，但是软件的设计却需要上百人或上千人的智能来为它创造价值。

然而，让每个人都充分发挥创意而又不牵绊住其中的天才，是一件相当

复杂的工作，也是领导者所面临最大的挑战，当然这不是软件业中惟一的挑战。教育可说是领导者或管理者最主要的角色：带领团队做案例研讨，带领大家思考如何解决一切的疑难，让每一件事都在该做的时候做好。

法则 19

Go for greatness

追求卓越

如何让组员生活在美的环境中，陶冶他们对美的素养？让组员阅读什么样的书籍或给他们什么挑战，可以加强他个人的成长？管理者应该培养组员对美学的概念，可以把千百年来人类的文明当作美学的来源，而重点则放在软件的设计。人类对美的感受是非常复杂的，几个世纪以来不断有人研究这个主题。因此，不妨利用前人探索的经验当作基础，以下我将引用一些前人的理论，来引导本章的讨论。

除了美学之外，我们也可以利用历史的角度：暂时忘掉这个信息时代，想想看历史上有什么人物或事迹（时间较近的排在前面）令你觉得很崇拜或很感动，改变了你的想法或态度？你团队的工作是什么？两者有没有类似的地方？这个历史故事能不能激发人们的潜能，而开启另一扇创意之门？这个创意是不是完全没有人想到过，而且会导向一个没有人想象过的未来？你可以思考这一类的问题，用历史的眼光来分析一下你的工作，也许会有意想不到的结果。

法则 20

State your tyeme

设定主题

有几件伟大的著作影响了我对伟大软件的观念：乔治·桑塔亚那（George Santayana） 1896 年的经典名作《美感》（The Sense of Beauty），以及较近的乔治·史汀尼（George Stiny）与詹姆士·吉普（James Gip）于 1978 年合著的《美学法则》（Algorithmic Aesthetics，University ofCalifornia Press 出版）。尤其是史汀尼的著作，对于我们现在所讨论的软件美学有很大的关系。史汀尼对美学的定义如下：

美学所关心的问题是如何描述、诠释和评价一件艺术作品（work of art），以及如何创造一件美的艺术作品。

在德惠克·帕克（DeWitt H.Parker）于 1926 年所著的《艺术分析》（The Analysis of Art，Yale University Press 出版）中提到六项美学的准则（Criteria of esthetic form），很适合作为分析软件设计之用，而我要求我的团队把它们当作设计时的标准。

德惠克·帕克的六项美学的准则是：统一、主题、变奏、演进、平衡、层级。

根据帕克的看法，统一性（unity）是伟大艺术作品最主要的原则。而我也曾经在许多件伟大的软件作品中看到这项特质。如果我们将帕克对统一性的解释套用在软件设计上，我们可以说，一件具有统一性的软件，每一个小组件对于整体的美感价值都非常重要，而且每一个美感元素都应该出现在作

品中，缺一不可，而且不应该出现的东西、对美感没有贡献的东西，绝对不要出现。因此，软件设计应该是把顾客想要的东西全部纳入，而顾客不要的东西统统排除，由于软件中所有的东西都是需要的，顾客对于软件的使用不会被干扰，而去注意不必要的东西。所以，追求软件的统一性，是软件设计的首要目标。也许你在别的领域（也许是创造一件艺术作品）的工作中，也曾经由于直觉或别的理论而运用统一性，而我认为帕克的六项原则对于软件的设计非常有用。

软件的主题（theme）会主宰设计的基础观点，也是软件价值的主要根源。因此，你必须在团队中明确地传播这个主题，让开发人员和行销人员对主题有非常透彻的了解才行。软件的主题事实上是目标的同义词。目标愈明确，造成的的冲击就愈大，因为你可以将模糊降到最低，而目标在每个人心目中造成的感受与解读会更一致，整个设计过程就愈平顺。但是主题决定之后，你还得注意与主题无关的部分都要删除掉，即使开发人员认为这一部分很重要，或者这是你一贯的信念，都还是得忍痛牺牲。

产品的销售信息会由主题衍生。精明的观察家只要看到主题就能抓住信息。信息只是补充说明主题的意义，如果主题模糊或是不只一个，再好的销售信息也没用（大家都明白这个浅显的道理：产品若没有鲜明而惟一的形象，广告再多也没用）。产品的主题根源于你的对市场的观念，以我 VisualC++的例子来说，我们对市场的观念是：大家都觉得 C++太难学，于是我们设计的主题是：让使用者容易学习 C++，我们的信息自然就是：Visual C++把 C++变容易了。

重点是产品的功能特色不能像是一袋子随便抓过来的东西，应该把与主题无关的东西都删掉，而且你的目标也必须符合统一性（unity of purpose）才行，这一点是与主题互为一体的两面。将资金投注在这个目标上，让所有的人都完全明白这个目标，并且为这个目标努力，做得到这些的话，你的产品就会完全包含这个目标。

专心致力于主题

我不知道以下的故事是真是假，但我很喜欢它，而且经常说给我的组员听。

有一家电子表格公司做了一项研究,结果发现到使用者大约每打 20 个数字就会想用这些数字画张图表。这项研究发现，大多数的人在用电子表格时都有这样的行为模式。

所以他们研究再研究，开发再开发，全心全意地努力让产品有这样的功能：可以利用很少量的数字，很容易地产生图表。那些用电子表格绘图的使用者看到这项功能特色都非常高兴，认为只要有这个功能就值得买下这套软件。

另一个类似的故事（我也不知道是真是假）是一个做家庭财务软件包的团队。他们研究过市场之后发现：这项产品一定要让顾客立即得到好处，否则就卖不出去，而且消费者不会再买升级版。他们决定要让任何一位完全陌生的使用者，都能够在打开包装后十分钟之内得到结果，从安装、操作、输入、输出，一定要又简单又快。

所以他们派人到软件商店去，征得顾客的同意，跟着他们回家观察他们的使用状况，巨细靡遗地记录下来：包括他们拆开包装之后会先看什么东西，在安装和执行软件的过程中会遇到什么困难等等之类的，以提供产品改进的

依据。分析研究之后，他们找到了数十种加强顾客满意度的机会，然后他们在往后的几个版本中逐步实现这个目标——十分钟内的满意。产品终于成功了！

变奏（variation）是将主题稍加变化润饰后，重新表现一遍。在主题表现过之后，为了持续吸引使用者的注意力和兴趣，变奏是以另外一种方式来诠释主题，加强使用者对主题的理解和欣赏，使他对主题留下深刻的印象。

演进（evolution）的意思是用前一部分来决定后一部分，就像是学习的过程应该是先入门后进阶一般，由浅入深的变化会让人更容易接受，更喜欢学习。如果软件作品的前后能够如此呼应，通常会有满意的结果。

平衡（balance）是对软件中各项组件都不偏废或过度强调。例如，正好相反的两个对象，应该给予相同份量的说明。

层级（hierarchy）是指软件作品中的各个元素，依照它们的重要性与大小给予合理程度的比重。层级与平衡的概念很接近，层级可以说是建立与衡量平衡的方法。如果主题是在层级的最顶端，则以下各层级的同层组件都应该彼此平衡，同层级的组件对主题的支持力也应该相等，愈近顶端的层级对主题的支持力量愈大，以此类推。

美的特质

1975 年，盖·瑟西罗（Guy Sircello）在他的著作《美学新主张》（A New Theory of Beauty）中，提出了一个有关于美感认知的学说，相当有趣。我们不谈他对物体特性的质与量的详细划分，他提到一些有关美学感受的理论，非常深得我心。瑟西罗认为我们之所以感受到美，是因为这个物体有一种以上的特质很美；瑟西罗进一步解释说，惟有一项特质在作品中被特别强调，才会是有美感的特质，而一个物体惟有包含一项以上的美感特质，才会被人们感受到它的美。足够的美感特质不一定保证让整个物体显得美，但是一个没有美感特质的物体绝对美不起来。

瑟西罗的学说或多或少解释了为什么有一些软件叫好不叫座，这些失败的软件什么都有，想要大小通吃，有一大串的功能特色就是没有主题，结果没有一项特色能够使产品鹤立鸡群，就无法吸引顾客的注意，当然注定要失败。

法则 21

Minimize dependencies

不要倚赖不确定的事

尽量减少团队需要而又无法控制的事情（dependency）。项目开始的时候，决定允许倚赖的事情愈少，最后就会愈顺利。一般来说，程序设计的效率是不会太高的（也许是由于不熟练或是错虫太难抓），即使你努力加班在时限内完成了自己份内的事，别人也可能无法做到这样。

因此，在设计时就得考虑这种必要而又不确定的事，要知道这种依存性有可能会吃掉大量的成本，只有在很重要或是非不得已的地方才允许这种倚赖，让成员们明白其严重的后果，尽量配合协助，并事先评估倚赖之事失去控制的可能性，以及会有什么影响。

法则 22

Propitiate the gods

平息顾客的愠怒

在项目进展的过程中，总有一些依存性或外在的不确定性因素可能会拖垮你。你必须在其中找出最有可能绊住你的因素，事先研究好万一发生问题时你该采取什么步骤。

先检视一下产品的功能特色，其中有没有不明确的、或是对顾客满意度没有太大意义的部分，在时间来不及时可以牺牲掉这些。不错，可能会有顾客不满意你没有做这项特色，但是只要你能如期推出新版，那顾客就不会放弃你，并期待在下一版中见到他要的特色。

法则 23

Portability is for canoes.

软件的可移植性

对于大部分的软件厂商而言，做到跨平台（multi-platform）的支持是相当困难的。即使不考虑每增加一种支持平台所增加的开发成本，在品保方面所增加的工作负担也是呈指数增长，再优秀的品保管理也无法真正解决这个问题。最好的办法是要求系统软件厂商提供工具支持，然后慎重小心地决定你要支持的平台，数目愈少愈好。

但千万别选错了，那会是你的致命伤。

法则 24

Design time at design time

在设计时将时间因素考虑在内

在设计时就将时间因素考虑在内，千万不要先设计好再决定要花多少时间才能做出来。时间是你最大的限制条件。

产品设计的目标一定要完成才能推出，你不能把做到一半的软件给顾客使用。开发人员和管理者在做产品设计时很容易忘记考虑时间因素。正确的做法恰好相反，你应该在设计阶段就把时间当成关键因素，当你在考虑替代方案时，时间短的加分，时间长的减分。通常只要把时间因素纳入设计时的考量重点，你就能够缩减开发的时间。

如果你的设计并不一定要产品如期完成？别傻了，还是如期完成最重要，比什么伟大的理想（可惜实现不了）重要太多了。开发

这里所谈的开发，包括的以下的实际行动：执行计划、进度安排、程序撰写以及品质验证，而不是讨论什么人做什么工作，在团队中每一个人都必须参与开发的活动，因此，每一位都是开发者。

人们使用软件开发（software development）来表示软件产生的过程。这一点很有意思，这表示软件的形成是一种渐渐成熟的过程，经过一组有顺序而相互影响的步骤演变而成。究竟是什么因素推动软件趋向成熟呢？是团队的创意凝聚。从众多个人的智能产物，逐渐组成一套结构紧密而完整的智能财产，那就是将要推出的软件产品。这又使我想起了本书一再强调的主题：软件开发的基本活动就是将一群个人的智能结合成一项智能财产。

软件开发的终点事实上是开发人员脑力的极限。

但软件开发的过程牵涉到很多层面。每一个人、团体互动和科技，三者都在发展，并不是固定不动的。所以，软件开发没有明显的起点或终点，而是很多人在很多不同的层面切入，共同创造。虽然每一次的新版，都可以说是一个里程碑，但是不能把它当成终点，软件开发的终点事实上是开发人员（脑力的极限）。

很多专家以工程的眼光分析软件开发的问题，而软件开发的问题也常常被当成是工程问题。但我却常把软件开发当成社会学或文化现象。和一群人共同创造智能财产，比起工程上的设计方法论或其他的基本理论要困难得太多了，在工程上的情况都是有限的，但在软件开发过程中的问题却是无限的，而且没有绝对的是与非，很多问题都是要靠沟通和取舍来解决。聪明才智是创造软件的绝对必要条件，但光有聪明才智并不够。

处于布局时期的“开发”，就像解联立方程组一样，方程式中的变量有无法预测的时间排序和不明确的产品内容。要知道，在该说的全说完但还没真正动手之前，大概没有人知道软件完成时的模样。在这个阶段，所谓的产品功能特色只是个描述性的名称，和大概期望的功能，而设计通常是到了新版产品出炉了才算真正结束。换句话说，在这个联立方程组中的变量都是不确定的值，而且还会随着开发阶段的演变而改变，这是个极为复杂的问题。

与其说软件开发是一组结构巧妙的程序，不如说是开发人员脑力的密切结合。

另一个在布局时期所面临的难解方程式是人的创意，也就是说你必须在此时就估计出每个组员创造力的效用程度。我们不仅要在本阶段仔细思考功能特色的内涵，还得预测它如何能变成具体明确的工作项目，何时能放到组员的手中加工，并且完成它。

上述的不确定性因素实在是太多太复杂了，以致安排进度几乎是不可能的任务。当然啦，你还是要设定项目的整体目标，然后集中精神在第一个里程碑，也许是项目开始后的一两个月，这大概是你在布局时期时最多所能做到的了。在第二篇孕育阶段中我们会更详细地谈设定里程碑的技巧；这是一个让团队练习在短时间内做出具体成果的好机会。注意每个里程碑的间隔不要超过三个月。

一般而言，软件开发比较像是在赶进度，而不像是在演奏交响乐；交响乐是和谐有序而优雅的，而软件开发却是一堆排山倒海、蜂拥而至的工作。这个比喻太可怕了，把它当作是即兴演奏吧！你必须自己知道什么时候该出来表现，什么时候退后一点，大家都是在动态的情况下，和很多人密切合作，任何两个音符都不要互相抵触，让整体表现出的是一段优美的旋律。在一切都不确定的情况下把整个活动带到最高潮，使其美妙的程度不下于交响乐，而且事实上，它本身就是一种喜乐。

法则 25

Don’t accept dictation

拒绝不合理的命令

我很惊讶居然有软件开发团队接受非专业人士的指挥，尤其是有关进度的事情。在现代科技领域的世界中竟然使用典型旧社会的方法，简直是大开

倒车。有些公司竟然让根本不懂软件开发的人来管理最重要的事——时间、特色和资源是软件开发的金三角——简直是疯了。那些在上位者或是行销人员是假装无所不能的妖魔鬼怪，变个魔术就能拟定进度了。更糟的是，有些专业人员竟然也接受这种缺乏见识、毫无逻辑的领导，还把这种事情当作是标准的作业程序呢。

我曾经接触过数以打计的开发团队，据我私下估计，大约有三到四成经常为不合理的重大命令所苦。合理的做法是由负责做事的人来估计时间。当然啦，如果精确不是目标之一的话，任何人都可以随便乱估。

如果在上位者不让真正执行任务的人来估计所需的进度，那就是专制。

有些时候，没有好好估计时间是团队自己的问题，是开发人员和项目经理没有担负起决定达成目标所需时间的责任。把估时间的责任从执行工作者手上拿走，是最最最反授权的做法。在任何情况下接受这种做法都是极槽糕的，一开始进度就是假的，团队如何会有工作能力呢？

没错，我们很能同情那些上位者需要感到自己能控制能预测，但究竟是什么原因使这种荒谬的决策一再上演，而使软件开发永远是个大灾难？

不管是人或组织，大部分的时候都无法从错误中学习。人有时候很粗心大意，上次做错的事，我试着再做一次能不能做好，完全没有细细思量上回弄错的原因，甚至不去想想看成功软件开发事例给我什么样启示。

这种盲目的现象特别容易发生在进度严重落后的团队中，很多开发人员再也受不了这种死亡进行曲，只好挂冠求去。这些离去的人对于工作要求都很高，才会不容许自己继续和这些官僚大爷们混下去，而且通常都是最重要的人。而留下来的人在怨声载道中，被弃于险地而无人理会。魔鬼在办公室的每一个角落出现。行销人员像个傻子，他们的承诺像是空气，不知不觉流露出愤世嫉俗的态度；高阶管理者充满困惑、窘困和愤怒；顾客觉得自己被骗了。灾难于是开始……

要知道，在疯人院中，头脑清醒的人反而会被当成疯子。

慢慢地，云消雾散，大家又重新燃起一线希望，也加入新的成员，新的（或是被原谅的）管理者来带领大家，而新的技术也吸引了开发人员再度投入热情……上过当却学不了乖的高阶领导者又开始了错误决定，下令软件必须在某月某日做出来，于是灾难的循环又再度开始……

如果是你，该如何对付这种荒谬的情况呢？如果你发现自己身陷在这种组织当中，该如何面对困境？记住！在这种荒谬的环境中，任何正常人都会失去理智，而组织中弥漫着不理性的行为，它的未来必定走向不可理谕，你在这里一点希望都没有，身为惟一头脑清楚的人，必定会被旁人当成是疯子。也许你可以明哲保身地在这种自我毁灭的组织中勉强活下去，但你不可能会有什么像样的作为。

但是无论如何总得解决这个问题。所以你在接受不合理命令的同时，还是要很技巧地提醒你的昏君，这样做是不对的，无法这样做是因为事实的要求。权力不是万能的，他们的做法行不通，你得帮助你的昏君明白，大家都在拼命完成皇上的目标，而不是团体的目标，如果能够多点参与，情况一定会比较好，人们也会因此而做得更快，更愉快。

开发进度表应该由下而上来拟定，每一个人负责自己的工作，也负责设定它的时间表，负责准时完成工作。责任和充分授权是一体的两面，二者兼备才能拟出合理的开发计划。

法则 26

Now go play

把工作当作游戏吧

如果把软件开发当成是一种工作，那实在是一个索然无趣的职业；从另一个角度来看，我建议你把它当成游戏。

在布局时期，每件事情都按步就班进行并不是成功的保证。

如果你引用我的法则来建立你的团队，虽然软件还是同一个，但你会发现他们更自动自发、反应更快、更多幽默、更容易做出令人欣赏的软件。把竞赛当成游戏，你比较容易赢。计算机其实是个充满趣味和挑战的玩具，而开发人员呢？当然是大玩家啰。

在健康的团队中，以游戏的方式来做事是很自然的。管理者不需要做什么特别的举动，只要避免限制组员的发挥，和注意大家是否乐在工作又自动自发就行了。鼓励组员快乐，不要怕他们分心，这可能是创意的泉源。

软件开发这场游戏，会自然而然地产生规则，做得好的人自然会获得奖励，而做得不好的人自然会受到惩罚。管理者不必操心正义和仲裁，游戏本身的奖惩比你做的奖惩要更有效果。

这个游戏能给你什么好处？多玩玩它，练习它，感受它的趣味，并且喜欢它。如此一来，软件创造本身的趣味会使僵硬、骄横和严肃都消失，这样最好。

在布局时期，每件事情都按步就班进行并不是成功的保证。本篇的目的是告诉你如何在发挥团队最大潜能之前，准备好你的团队。本篇并不是成功的公式，而是伟大的先决条件。如果每件事都能在布局时期就照本书的法则进行，那么以后仍然会如此进行，协助你的团队成长起来。

第二篇　中程时期

本书为了帮助读者建立清楚的观念，将软件开发过程划分成四个阶段，然而在实际运作上真的很难这样做，各个阶段彼此都有重复的部分。在本篇中，我不设章节段落，因为这一部分的内容用各项法则来说明比较清楚。因此请不要以为我在每个阶段所用的法则仅适用于该阶段。事实上，我所提出的法则都是观念，适用范围是整个的软件开发过程，好的开发团队应该随时随地注意每一个法则是否被实践。而在起始阶段所提的各项法则，如果你有做到的话，你会发现在孕育阶段更容易实践它们，因为你已经练习过这些方法。

在起始阶段的主要目的是让团队凝聚并建立对团体与产品的目标，而蕴育阶段则是充满对目标的期望、不确定和挣扎，所有的事情都是一片混沌，并且充满对可能失败的恐惧。这时候，毅力是最重要的。暂且不管你的团队是多成熟，暂且不管你经历过多少次这种场面，或是产品出炉时会有多伟大，孕育阶段总是可怕而乱糟糟的，你随时都可能遭遇意外打击或是想象不到的失败，但绝对要坚持下去。

你很可能会有那种心情：但愿有条路让我逃离这一切，每个项目都会有这种时候。每件事情都无法确定，于是组员的心情开始恶劣，冷嘲热讽、消极抵抗都来了，几个月下来不见天日的辛勤工作，使得组员好似得了坏血病。这场灾难似乎永无休止，可怕啊！

深呼吸一下，进入孕育阶段吧！

法则 27

Be like the doctors

用医生的方法

当病人已经药石罔效时，医生通常会对病情有所保留，避免病人太过悲观或恐惧，并且尽量鼓励病人保持希望，最好能让病人有个期望完成的目标。软件开发在这方面也很像医学，它不是完全能用科学来解决一切。只是很不幸，到目前为止，一般人还不了解软件开发是一门艺术——是必须具有技术专长的艺术。

现在，你得学着用医生的方法。医生绝对不会斩钉截铁地断言什么医疗行为一定会有什么样的结果，反而是以一种自在且充满信心的口吻说：“试试看吧，一切都还没有确定呢。”反观软件项目经理人，常常在还不确定是否可行时，就率尔对团队保证产品的特色、时间等等。更糟的是，医生往往是在整体系统大都健全的情况下，对其中的一部分功能做医疗；而软件开发则常常得面对全新的、从来没有运作过的系统，不确定性更高。

当然，任何情况都是有可能的，即使是再简单的医疗行为，都可能有风险。

另外一件应该向医生学习的事情是，即使是再小再简单的医疗行为，都带着几分风险，所以医生会说：“当然，任何情况都是有可能的，治愈率再高我都不能跟你说百分之百没问题。”如果这样说都还不能让组员明白任何未来的事都有不确定性、都有风险，那我也只好束手无策。

我们必须记住，面对软件开发的不确定性，总会有方法可以试试看。就

像医生会试着让病人了解任何医疗行为都有风险一般，管理者也应该让组员了解，任何项目都有不确定性和风险。

法则 28

Remember the triangle：features，resources，times

软件开发金三角：特色、资源和时间

作为一位软件开发的领导人，你得集中注意力在三件事情：资源（人和钱）、特色（产品与其品质）和时间。这三件事是软件开发的核心，其他的都是外围。资源、特色和时间是三角形的三个边，任何一边的变化，都会影响到另外一边或两边。所以如果时间落后了，去看你的三角形，看看对特色和资源的影响；当有人谈到可以增加什么功能特色时，你得立刻谈起时间和资源，以显得你思虑周详反应敏捷。所以，管理者的第一要务是把这个三角形放在心里，随时利用这个模式来思考问题，你会发现答案都在这个三角形内。“好吧，我的时间落后了，我得利用另外两个边来弥补，或是三边都得调整，我可以少一点特色，我也可以加一点资源，或是修改一下时间。”由于人、时间和特色都是你最关心的，所以你对这个三角形有具体的概念，很快地，你就会发现这个三角形对你的思考帮助非常大。

还有一点要记得，时间落后时，你只有四种选择：增加时间、减少特色、增加资源或三者同时进行。但人是不可以随便增加的。

不可加派人手？

你是否听说过，加派人手是个错误的决策？在佛莱德·布鲁克有关软件开发的经典著作《人力资源之钥》（The Mythical Man-Month）中，用了很大的篇幅来阐述他的论点：在软件开发项目中不当地增加人手，反而会使工作进行更慢。布鲁克的观点被证明是非常正确的，但我们应该了解的是，布鲁克是在写一本书，而不是写一部适用的律法。

然而，传统的教育使得人们用太过单纯的理解方式，使得布鲁克真正的睿智多少被曲解——绝对不要在进行到一半的开发团队增加人手，重点是‘进行中的团队”。虽然布鲁克十之八九是对的，但却成了太多管理者的借口，理直气壮地决定不加派人手。是的，增加人手是很难管理（人员应该在项目开始前就规划妥当），通常也不是个好办法，只有在非不得已时才考虑，但不是被禁止的。

法则 29

Don’t know what you don’t know

不懂别装懂

对于不懂的事情，千万别装懂，或是看起来似乎懂，也不要接受别人不懂却装懂。不懂却装懂一定会造成团队管理上的困扰，这一点几乎没有例外，你若犯了这种错误，一定会尝到它的苦果。在每个阶段的每一个人，一定有某些重要的事情是他不知道的，这应该是被允许的。去掩饰你不懂的事情只会造成别人在认知上的偏差，结果反而导致不知道那些事情可以相信，那些不能相信。如果你勇于承认自己不懂的事情，就不会掉进这个泥淖。

人们会觉得对于重要的事情，我如果不知道就很丢脸，这是天性。而在

软件开发过程中，太多东西是大家不知道的，因此，管理者或开发人员就很容易有这种不懂装懂的倾向。好的开发团队应该有一张清单，上面列着我们目前不知道的事情，这样才比较容易掌握到底什么事情会不确定。抗拒“我都知道”的骄傲天性，是需要团队士气和心理上的勇气的。对于管理者而言，称许承认不懂的勇敢组员更加困难，尤其是这个事实已经被文饰过的时候。虚假的文饰是面对未知事情的错误防卫心理。

虚假的文饰是面对未知事情的错误防卫心理。

我非常建议管理者重视组员了解自己什么地方不知道，而不是被迫或自愿去掩饰。要让你的团队明白他们从未检视过自己的未知，但要从现在开始练习承认自己不懂的事情；一般来说，在团队成立的初期或转变阶段会比较容易做到。我们没有时间拐弯抹角。最后，大家会明白成功比较重要，牺牲一点点面子或是受到幻灭的打击又有什么关系呢？

你坚持组员必须面对不确定性，承认自己不知道，最后会将无知变成知识。领导者的任务是让大家全都明白不确定性是绝对的事实，必须强迫大家去面对它、适应它。为了大家好，团队应该学会在不确定的环境中生存，并且兴旺起来。

既然要写在书里，就让我们说这是正式的规定：当有一件事是不确定时，就直接说明这个事实，即使不确定的事情是何时能够推出新版。不必担心，没有项目经理会因为承认自己不知道的事情而被撤换。

不必说大家都晓得，我的属下会因为知道自己不懂什么而获得我的加分，因为知道自己不懂，才会去求知。我宁愿知道什么不足，这样我才知道什么事情可能会绊倒我，我的同事和属下最好也明白这些。

软件开发项目的目标并不是事前做好正确的规划，而是每天都得在事情从未知到已知的时候，做出正确的抉择。如果你明明不知道某件事却假装知道，你就无法在事情从未知到已知的时候得到正确的信息，也就可能会做出错误的决策。当信息证明你错了，你一定觉得非常难过。于是你会更加害怕信息，而别人就以为你在抗拒事实，最后你将陷入恶性循环。

只有当你知道不确定性在那里时，你才有可能解决它；那些没有被发现的确定事情，会把你绊倒。相形之下，承认你不知道是比被击倒要好得多了。

当你不知道事情该如何完成，当你内心有个声音在质问你、在困惑你——面对它吧！不必害怕承认了自己不懂就显得很笨，你一定会因此而学到某件事情的。在你内心不断萦回的声音，事实上是团队还没浮现的怀疑，而你收到了无形的信号。也许当你告诉别人时，得到的回答是：“对啊，我也在怀疑同样的事情。”当然，偶尔别人会认为你是不理性的空穴来风，没关系，那是你在不确定的环境中必须付出的小代价（如果命中率实在太低，也许你得看看心理医生，但至少不会令你延误就医吧）。

表里一致

表里一致是一种平衡与调和。你是否说一套做一套？或是想一套说一套？或是意念和行为不一致？我们的言谈真的是我们想说的话吗？我们是否让别人牵着鼻子走？我们的行为和我们的感受与意愿是否一致？当我们说要做,是不是要去做呢？我们是真的同意,还是假装一下让别人以为我们同意？

一致性是诚实与伪善的区别所在。当我在做某一件事情的时候，我是因为这件事本身值得做，还是因为它对别人似乎有点价值？我是诚实说明自己的看法，还是比较倾向保护自己？当我认为某件事情是个好主意时，我是否

会去做呢？我是真的相信，还是让别人以为我相信？

一致性也是实情之所在。惟一比说出实情更难的是了解实情。当周围所有的人都认定某件事情是对的，但我实在不能证明，只有直觉地怀疑那是错的而团体思考模式尚未建立时，我能不能说出我的看法然后去发掘真相呢？

法则 30

Don ’ t go dark

建立适当的检查点

假设在你接近检查点日期时询问开发人员：“ 情况如何？跟得上进度吗？ ”

他回答：“ 最近六个星期都进行得非常顺利，但是今天的进度呢，是六个月前该做完的事。 ”

不，落后不是一天造成的，落后不是到了项目快结束时才出现的，也不是到检查点前一天才发生的，它每天都在发生，每小时都会发生。每一次你泡一壶香浓的咖啡，每一次你回个电子邮件、组装一部计算机、抓一只难解的虫，都是落后发生的时候。

为了对付进度落后的问题，你必须把一个大型的开发工作切分成细细的小段，每一小段都是一个检查点，每一个检查点都必须能够看出来有没有延误。检查点的间隔周期应该多长呢？如果太长，检查点的效果不够；如果太短，工作可能无法分割得这么细。软件不像堆积木那样简单，程序之间的关系是立体的、动态的，倒是很像庖丁解牛。在我的小组中，我们为此来来回回争论不休：五天？十天？三周？根据我的经验，三周足以使情况失去控制。

每家公司适合的检查周期并不一定。我们的做法是，让组员和组员之间彼此协议，一方应在期限之内做出该做的东西交给另一方，否则对方无法及时开始，如此就形成了适当的工作切割点，而且一有延误，我们立刻会知道。比方说，我们知道这个星期的进度落后了一天，一天可不是小事，这是非常必要知道的，这总比进度落后了半年我们才知道要好得多了，万一到了那种落后程度时，恐怕连计算落后多久都来不及了，全世界最糟糕的事情莫过于此。

全世界最糟糕的事情，是你迷失了，迷失在软件开发的金字塔中。在孕育阶段你很可能有这种完全被搞迷糊的经验：“ 我不知道该做什么才对，我不知道现在是离目标更近或更远，我只知道现在的状况糟透了，我所做的每一项措施似乎只有把事情搞砸，我只能确定每一件事情都搞砸了…… ”这种可怕的感觉是来自纵容黑箱工作——让工作在没有人检查的状况下进行。把你的灯打开吧，让你的组员在透明的环境工作，一天完成一天份量的工作。

迷失在软件中

我不必浪费时间向你说明开发进度落后是多么严重的问题，你完全明白，否则你就不会在本书中找寻答案。随便找个软件开发人员或项目经理问问看，他们有没有经历过进度严重落后的问题，答案只可能有两种：一是有，一是表面上没有但事实上有。表面上没有但进度延误是因为原本估计的进度表过于宽松，所以能补偿落后的进度，或是大幅降低目标作为代价。

更恼人的问题是，有多少项目经理或开发人员感觉自己迷失在软件开发的金字塔中？

大概凡是软件开发从业人员都能理解这个问题的涵义，迷失是极可怕的感觉。空有满坑满谷的理论帮你计算项目进度的指标，却完全没有办法让你摆脱郁闷，每一个人都觉得闷闷的，你开始失去敏锐的判断力，即使是有点疯狂的档案结构方式也被认真考虑，焦虑、畏惧、烦恼、沮丧，偶尔又觉得挺有希望的。你根本已经不知道自己在干什么，事情做得怎么样，问题到底在那里，你的每一个动作都是大失败，你完全束手无策。

于是，你开始有一种信心溃散的直觉，自己一定是能力不足才会搞成这样。接着，每一个人都会发现你的恐惧，对你的信心也因此而消失；组员已经不信任你，你还想如何领导他们？

这些，正是每个项目经理人的锥心之痛。

现在，基于检查点的重要性，你告诉开发人员：“我们要每周做一次进度检查。”有人反映这样的管理太细了。为了让这些人能够真正接受这样的管理方式，你得向他们说明，还有其他的人等着这份程序完成才能开始做事——文件要撰写、品保人员要评核，而且还有其他的开发人员也需要得知本阶段的执行结果，才能撰写或测试相关的程序。团队成员彼此之间的工作时程有上下游和回馈的关系，彼此互相依赖彼此的进度，于是形成了实施检查点的动机，而不是靠管理者的铁腕来强制。

有些产品的功能特色需要长达数个月或数年的时间才能完成，但是时程的掌握是与任务大小无关的，因为任何的进度延误都是一点一滴累积出来。也就是说，工作的切割必须够细，万一其中一小段发生延误，可以在短期内立即赶上。事实上，一个星期的检查周期已经够长了，想想看，如果要在下一个星期内补赶进度，你是否做得到。因此，即使管得太罗嗦，让组员讨厌你，但为了整体进度能够如期完成还是不得不如此；而且如果大家都把时程当作是最重要的，那么大家都会觉得时程表让人有成就感，而比较不介意检查的麻烦。

此外，同事之间在进度上的互相依赖和承诺，也是顺利执行检查点的要素之一。就像产品设计的目标是让每位成员都参与，每个人都贡献出自己最好的想法，检查点的实施也是同样的道理，每个人都实践自己的承诺，这个制度就能成功。比方说，开发人员玛丽答应文件撰稿人乔伊星期五给他写好的程序，于是玛丽个人的信用就成了工作的一部分，这时候玛丽会自动自发地如期完成；另一方面，如果管理者比尔命令开发人员哈瓦德负责的程序必须在周五前完成，那么哈瓦德心中对威权的反感成了工作的一部分，而比尔对自己管理者形象的不安全感，也同样成了工作的一部分。这又是创造智能财产过程中有趣的现象：每个人的心理状态对成果都有直接影响，没有人例外。

我在前文所提的表里一致，对于软件开发的影响就是进度问题，每个人都重视自己的承诺和信用，彼此没有隐瞒，而检查点的实施就会很顺利。对于那成千上万的小段工作，就靠彼此的承诺和信用来维系。这和整体目标的大小没有关系，倒和组员是否表里一致、勇于面对不确定性很有关系。

法则 31

Beware of a guy in a room

留心没有检查点的组员

法则 31 是法则 30 的特例，但有必要独立说明。

曾经有一次（其实是三次，但我实在惭愧得不敢把三个故事都说出来），我把最困难的工作分配给团队里最最优秀的人才威廉，大家都知道他是最有经验、最有创造力、技术能力最强的人，我们拥有他是上帝的恩宠。没有人怀疑他的天份与判断力。

当威廉开始他最具挑战性的工作时，他回房关上门，此时陪伴他的只有水族箱、重金属音乐与莫扎特的旋律、泡泡糖和星际争霸战的海报，以及 6 箱果汁和可乐，一切就绪后，他开始工作了。其他的组员都对他充满信心，非常庆幸我们拥有威廉这样的天才可以扛下这么艰巨的任务。在门外我们听到他快速敲击键盘的声音日夜不停，心中在微笑，虽然没有人知道在关上的门内是多么伟大的程序，但我们都觉得有高手威廉辛勤地为公司的命运打拼，真是大家的福气。

他的前额出现了斗大的汗珠，显然压力很大。

（而这是前所未有的乐观情况，我们的目标很清楚，我们的安全控管做得很好，我们的团队是最优秀的人所组成，虽然我们偶尔来杯即溶咖啡，但我们都深信自己在做一项伟大的计划，别人的批评我们充耳不闻。）

渐渐地，我们开始看出威廉遇到了困难。他渐渐不回家了，鱼也不喂了，音乐也停止了，斗大的汗珠出现在他的额头，泡泡糖愈吹愈小，威廉显然是……进度落后了。

整个工作只有威廉的部分没有检查点，他只要在完成时交出作品就行了，当其他的人都已经如期完成份内的工作，我们不得不敲敲他的门：“ 威廉，做得怎样了？ ”里面是一阵迟疑，然后听到他说：“ 还不错。 ”“ 什么时候可以完成呢？ ”经过更长的停顿，我们几乎听到泪水流下的声音，威廉哽咽着回答：“ 就快了。 ”

我们知道他说的“ 快 ”，其实还早得很。现在全公司的人都只能呆呆站着干等，一点也帮不上忙。而我，身为主管和项目负责人，面临了困难的抉择：我可以开除威廉，但考虑到他是惟一能够解救我们的人，惟一真正了解当前困难之所在的人，所以我不加思索地决定，让威廉继续完成他的工作，他毕竟是团对中最优秀的人才，也是团队中惟一能够掌握这个问题里里外外的人。我惟一能做的，只有替他买更多的可乐。

另一个比较好的方法是给他加点压力，让他搞清楚自己身系团队的危急存亡，所有组员和他们的家庭都指望他不凡的脑袋和飞快的手指，我们得催逼他做出个像样的东西出来。

但我稍微三思，就觉得这也不是个好办法，对生产力没有帮助。我没有办法摆脱这个困境了，我只能往好的方面想：威廉会完成他的工作的，只要他没死或离职。经过这次教训，他大概再也不敢为我工作了。现在我惟一能做的事只有帮他买更多的可乐，其他什么办法都没有！

当然，如果你有一位大天才关起房门埋头苦干，你或许会有更具独特创意的结果，一种是健康型，一种是病态型。病态型的闭关修炼是大天才无法在任何一关做出成品，却能在期限的最后五分钟交出奇迹似的杰作；健康型的闭关修炼是大天才免除一切世俗的干扰，全心致力于工作，由于他的工作创意程度太高，完全得靠他的心理、情绪等内在的心智力量，这时候团队合作对他来说没有什么帮助，反而有打扰之虞。

像这种天才型的特殊程序设计师，为了清静而把自己关在房间里很长的

时间，而没有人知道他的工作进展，也许是有发挥创造力的必要，但这对于如期推出产品而言是冒着极大的风险，无论他多么优秀，领导者都不该把重要的工作交托给他，除非让他与其他的开发人员一起接受定期检查工作进度，而且对时限绝不宽贷。可惜有些天赋才智恣意奔驰的人无法忍受这样的限制。在软件业中，这种人确实存在，他可以做一些无法想象的创举（在实验室里？），掀起一波技术的飞跃，但他绝对不会出现在矢志如期推出产品的开发团队中。

不论是健康的或病态的闭关修炼者，允许一位开发人员关在房间里没有人知道他的工作进度，结果通常会是开发团队的致命伤。所以，一定得竭尽所能地避免这种情况，绝对不要允许“没有检查点的组员”，否则你难逃失败。

法则 32

If you build it， it will ship

软件要经常建构，就能顺利推出

（软件通常是由许多许多程序组成的；程序写好后，经过编译器产生执行码，这个动作叫 compile，通常如果是小而独立的程序，可选直接选择编译成可执行档，但是以软件包或 C++之类的大型软件而言，程序数目都在数百个以上，通常每个程序都是编译成对象文件，即 object code，然后众多不同的程序依一定的结构关系组合成一个软件，这个组合的动作叫作建构，build，产生的结果是真正可执行的完整软件。每一套软件的建构方式多多少少都有些不同，原则上，被呼叫的程序或函数库要先 build，主程序最后 build。——译者注）

将程序建构成软件才能推出，这个自然，总不能卖给顾客一个不可执行的原始程序代码吧！但我的意思是要能够经常且定期地建构软件，并不是到了期限前一天才建构那么一次。你必须随时都让整个软件的现状都能被大家看见才行。

（在很早以前曾经有过一个案例，各个程序都按计划中的时间表进行撰写，个别程序的测试也完全正确，但是到了最后却怎样也无法组合，各个程序就是无法搭配。因为各个程序之间都有相互传递资料或先后连结的关系，或是其中有一个程序内有个无伤大雅的小瑕疵未被注意，但却在别的程序上造成大问题。在写程序的过程中随时要注意我的程序能否跟其他所有的程序相互配合，最好的办法就是常常 build，这样才能做完整的测试。完整测试是 build 最基本的目的，所以作者就不刻意强调，但定期建构软件还有更重要的益处。——译者注）

你能想象几百位工匠蒙着眼睛盖一栋大楼吗？不知道自己在盖大楼的那一部分，不知道这栋大楼现在盖到第几层，只顾盖自己的部分，这种大楼怎会不垮呢？因此，经常建构软件是非常必要的，这个法则的重点不仅是要在整个开发过程中经常地、定期地建构软件，而且要尽可能建构出现阶段最完整又最正确的软件，更重要的是建构出来的结果要放在公开的地方，让每个人都能看到。

软件也许不必每天都做建构，但是一定要经常且定期做，而且不只是程序要做建构，安装程序和线上求助的部分也要包括在内，然后将建构出来的

结果放在公开的地方，让品保人员可以评估每天的软件状况，也可以观察出它发展的情形，或是陷于停滞。规律建构软件是一项最可靠的指标，表示团队的运作是否正常，软件是否能够完成。

软件的建构需要时间，程序也很复杂，值得你为它仔细研拟最适当的策略。在微软内部有很多开发团队采取每日建构的策略，并有专门的小组（build master）负责这项工作。建构程序是很重要的，必须确定软件能够建构得起来，没有失败或中断。在每一只程序置入（check-in）开发环境时，会产生这一只程序需要的建构程序（通常是配合适当选项的 compile），然后整合在整体的建构程序中。于是其中任何一段建构程序发生问题的话，很容易找到由谁来负责，所以组员的责任心和荣誉感会促使自己一定要做出至少能够合格的程序，而每天的软件建构就会非常顺利。当然你可以做得更严格，以我的经验，一天建构一次是最有效率的方式（请参考下一个法则）。

（有些软件公司每次的建构都是把所有的程序都 compile 一遍，然后 build 起来，有些则是将修改过的程序 compile，再 build 上已有的软件。由于全部 build 一次所需的时间可能太长，所以大部分的公司都采取第二种方式。说得更口语一些，就是把其中的一块挖下来改一改再嵌回去。当这段程序被认为 OK 了，就做置入的动作，check-in；每隔一定的时间，就有专人把这些被改过的程序 compile 和 build。——译者注）

另一方面，项目经理应该是拟定最佳化的建构策略，而不是绝对性的每隔几天做一次建构。不一定要每天做，应该是采取对软件最适合又最小的间隔，重点是在能够建构出软件的前题下，用最短的时间间隔建构出软件。

老实说，我并不知道软件“应该”采取多长的建构周期最好，但我非常相信大部分的软件公司在这方面都做得不够。对某些开发团队而言，每天建构一次也许稍多，但我很确定，对任何开发团队而言，每周建构一次绝对是最起码的。

项目经理一定要时常看见软件的整体状况，才能明了现在正在做什么。即使你的工作分配和检查点设置得非常好，各个组员之间的信任与承诺关系也非常理想，但若是无法建构软件，则你对进度的掌握将仅限于想象和猜测，而不是根据事实，所以你还是不能掌握软件开发的真正状况。

每一次你建构软件，你就能算出错虫数目有多少，你就能掌握软件开发的进度，你就能看见功能日益成形。

再者，为了让软件能够建构出来，程序必须维持在一定的品质水准，太夸张的错误要立刻被发现，而且不可能将软件倒回去重来。因此，每隔固定时间建构软件，才能够维持软件有一定的秩序和品质，这一点要让大家都知道。

更重要的是，定期建构软件就像是团队的心跳一样，每天五点（比方说），大家都看到今天整个团队的工作成果，每天都看得到今天的进步，这对团队士气有很大的鼓舞。如果有一天软件建构失败，或是为了某种原因而不做建构，马上就会引起大家的注意和紧张，一定要找出原因来改正。如果建构经常失败，这就是整个项目失败的前兆。

但请不要搞混了，我所谓的软件建构，不是程序设计师在自己的 PC 上做的小规模的建构，而是指正式的、公开的建构，将大大小小的程序结合成软件，而且是每个人都能看见、能执行的软件。品保人员可以对它做初步的测试（sniff test），看看它的基本功能大致上有没有错误。如果你每天都做

软件建构，你就很有把握它至少不会当得惨兮兮，而且大家可以看见它朝着目标一天天地迈进，功能特色一天天地成形；而且每个人都测试着相同的软件，对于软件的现状也就会有相同的认知。

经常建构软件还有其他的好处：

·经常而公开地建构软件可以看出组员彼此之间真正的信赖程度。只要有任何一个环节没有密合，软件就建构不起来，检视建构的过程就像是检视组员之间或是团队对外的关系，在任何地方发生问题都很容易找到。

·建构出来的软件可以显露出在设计时没有考虑到的问题，例如执行效能、对象大小等等，这种问题万一发现得太晚就来不及修改了。

·软件的建构会很自然地让组员的脚步一致。一般而言，建构软件最常见的问题是版本协调，最严重的是各人有各人的版本，谁都不知道别人在做什么版本，有了公开建构的软件，大家的版本就可以同步了。

·建构软件可以促使组员面对他想忽略的问题。团队恒等于软件，所以目前的软件状态就是目前的团队状态。当我遇到产品有问题时，我追问组员：“我们软件建构的情况如何？”答案很多种，意思只有一个：“我们花很多时间才能成功建构一次，有时候长达两个星期，我们希望时间能够缩短，但是因为某些某些原因，我们没有办法照计划来建构。”是的，要频繁而规律地建构软件当然会有很多困难，但是解决这些困难却能够带来健康的团队和顺利的开发。开发人员必须得自律，自己仔细检查所有的功能都完善了才把程序置入，程序不仅要在逻辑上执行正确，还得注意体裁（程序段落清楚）和资源运用（内存用完了要释回），以及执行效能，如果个别的程序组件都很健全，就比较容易建构软件，并且能够大大减少回溯检查的时间消耗。

在开发过程中，很容易对软件的状况产生错觉，但是如果你每天都建构一次软件，你看到的将会是具体的事实。

法则 33

Get a known state and staythere

掌握实际情况

这又是一个很简单又很有用的道理，引伸的意义是“每天都要有可以推出的产品”。

你必须对软件的实际状况知道得非常清楚，特别是要推出的时候，你必须知道它是什么模样，架构、特色、效能特性等等。在你将它当成产品推出时，你必须知道所有能够得知的情况。

如果你询问开发人员软件的状况，他可能答对，但那是碰巧。

现在，假设有人要求你对刚推出的产品加进某一个特色，你能够立刻掌握这个特色对软件的一切意义，那你就会胸有成竹，因为你知道该怎么做才能加入这个特色，会不会对现有架构产生重大的影响或冲突等等。你可以召集一些开发人员和品保人员、文件人员，讨论一下就可以确定大家对这项特色的了解，大家对自己的角色和任务范围都非常清楚，然后你就可以说：“来吧，开始行动！”

也许几天以后，也许一星期左右，那几位组员手上拿着磁盘片来找你，告诉你新的特色做好了，只要安装这个就可以把新的特色加入软件。并不困难，是不是？

这是因为你能够充分掌握情况的缘故。你要知道，推出一个新版本就好像无数次不停地加上新的特色，然后推出。这是最能让你理解组织软件开发活动的方式：将一群各种角色的人组织成一组，负责一项功能特色；关键就在让软件保持在你能充分掌握，又能够推出的状态。千万不要让软件变成一堆凌乱的单元，千万不要让软件成为你不清楚的状况，千万要抓紧对它的掌握，不要放松。

想要掌握软件的状况，一定要有品保人员帮助你，你需要有人专职负责检查软件的状况。如果你去询问开发人员，他的答案也许是对的，但那只不过是碰巧猜对。开发人员虽然是直接创造软件的人，但他们却无法知道软件全面性的真实面貌，这还是得靠品保人员来评估衡量，让专门的第三者来向你报告软件的状况。

而且你每天都得亲自做点小的测试，一部分是自动的，一部分是手动的；每周或每两周要把产品整个测过一遍，确保你的软件随时都在可以推出的状态。

什么叫做“掌握软件的状况”呢？就是在某一特定的时间点，对于每一个产品组件的一切状态都有精确的信息。因为有品保人员将各个组件都测试过，所以你能确知信息是正确的。在你手上必须有一张清单，上面清楚列出各个测试通过和没通过的功能、错虫数目、错虫发现率和清除率，和一些其他重要的数据。

我们要注重数据管理，像变动率（churn）就是很好的数据，你可以看出有多少的程序代码增减，当然还有许多其他的数据可以参考，都挺有用的。但重点是，你决定要看那几项数据，就得每天去测量它们。

预定的目标能够确实完成，就可以推出。

然而，并不是由品保人员去评估决定软件是否可以推出。由于开发是团队全体的责任，软件是否可以推出在每个人的心里都有数，这应该是不会有争议的——原本设定的目标能够确实完成，就可以推出。当你终于在每一项预定功能上面打勾表示完成时，你会拥有那种真正的成就感（还有你的报酬），难怪你愿意为这个勾勾去努力。

但先决条件是你能掌握软件的状况，而且维持有关软件的最新信息。你必须善用你的品保人员来做到这一点。

时时刻刻对软件的状况清楚掌握其实是相当困难的，你必须有一群非常优秀的品保人员。很多软件公司只有很少或没有真正的品保人员，所以永远无法掌握软件的状况，事实上，一个现代化的软件开发团队是不能没有品保的。

记录里程碑

这一段主要是对法则 33 的补充说明。我在此特别要针对这些软件开发的观念作详尽的说明，这些都是我多年来不断摸索、思考、创见、旁征博引以及求来的，是我与多位极聪明的同事，犯过不少大错所换取到的。

你想在软件产业中占有一席之地吗？你想有点不凡的进步，不是吗？所以你必须做个“里程碑”（milestone）的记号，表示你现在到达了某个目标，但你还要前进到下一个里程碑。

里程碑一定要有一种衡量的标准，否则很难达到，所以不要设定一个无法精确定义的目标。你的每一个目标，不论大小，都应该有个专属的档案来做记录，你的资源投入一定不能模糊随便。

法则 34

Use ZD milestones

零缺点里程碑

在发展中的软件本质上是无形的、捉摸不定的。一部分存在于开发人员的脑袋，一部分是在媒体里边看不见的字节，一部分以纸张的形式散见于各个计划中的文字，更有一部分是完全不知道的潜意识，随时都在变化，要到适当的时机才会被激发出来。而团体的潜意识，不论是创造性或是病态性，在无形中都表现在组员的日常活动和每天所做的决定中。正如前文所述，软件恒等于团队，所以团队的一切状况都表现在软件之中，软件就是团队活生生的投影。

零缺点不代表软件中没有错虫，也不表示没有遗漏的功能。

如果想把软件的开发活动管理得当，就得让软件定期“整装待发”，这时候大家对软件所付出一切有形无形的努力就可见分晓，同时也可以分析那看不见的潜意识究竟是创造性或病态性。当以各种形式分散在各处的软件组件建构成一整体之后，你可以得到完整的软件状况信息，看看下回应该改进什么。然而，即便是经常建构软件的好处那么多，这件事背后所需的一切——包括产品的设计、发展程序和团队内心潜藏的各种动力，实在是件复杂无比的工作。

软件推出就是最后一个里程碑。

暂且不管多么复杂，为了让开发工作能够顺利进行，你至少要让组员把手上的工作划出一个成形的句点，并且要有勇气面对一切麻烦的问题，同时也让组员加强对软件现状的了解，让组员更有信心、更有能力预测自己付出什么程度的努力会有什么样的结果。一开始大家都会很痛苦，但慢慢能够培养出这些能力，对这些场面应付自如，开发工作也会顺利起来（请参看法则 3）。

很多微软的开发团队使用“零缺点里程碑”（ZeroDefects milestone）的方式来了解软件现状，简称“ZD”里程碑。所谓零缺点，并不代表软件中没有错虫，也不表示没有遗漏的功能，而是指团队的成品达到了事先规划的品质水准，也经过测试验证，就是零缺点里程碑。在 Visual C++的团队中，我们每一次推出产品通常会预先规划三到四个里程碑，一个产品周期正常是一年左右。

当然，最后一个、也是最高的里程碑就是软件推出，但是其他的里程碑绝对是同等重要。

我们在法则 33 中倡导，软件应该随时保持可以推出的状态，而且每天如此、永远如此，这是个理想，但事实上即使没有进度落后的现象，有时候也会做不到这一点，因为你偶尔会需要倒回头一些，去处理前面的问题（regressive things），像是将前面为了换新功能而暂时替代的东西要移除掉（好像在道路修筑过程中要另辟一条暂时的便道才能维持畅通，确定修好了之后便道就得拆除）。而零缺点里程碑，就像是经过较长的时间后的某一点，要确保软件的状态必须达到预期的品质水准，好似定期大扫除或大检修一样，不要把小缺点久留。

一个里程碑大约是六星期到两个月，当这个日期到了，除非里程目标已

经达成，所有的开发活动都不能跨越这个里程碑往下走。因此，里程目标常常被称为“ 中间产品 ”（deliverables），而且是大家都能看得到的。达成里程目标之后，新的里程随即开始。

这个原则说来容易做来难，如果你有充分授权的开发团队，团队中的各个成员可以共同确定里程目标达成，管理者没什么事要操心。每一次的里程目标都由品保人员事先设计好验证的准则，事先沟通每一个通过验证的品质水准细节，而且大家都同意、有共识。也就是说，这个中间产品的品质水准是由组员协商出来的，而不是管理者命令出来的。每一个中间产品都有事先定义好的验证准则，注明每一项准则应该在什么时候通过，这许许多多的验证准则会由项目经理集合起来，就是里程碑。

采用“ 零缺点里程碑 ”最大的好处是，每当有进度延误时，能够立即发现并在最短的时间内补上，也可以确保问题能够防微杜渐，及时处理。如果你每一两个月就有一次里程碑，你就得把进度延误控制在这个里程碑之内。你在一个里程碑内所发生的进度延误总比整个项目少，也比较容易赶上，这比到了最后推出时才发现要好得多了。每一次里程碑都确实达到，也就给你一个确定的信息；若是预定的里程碑日期到了，软件却又再过了 n 周才达到应有的品质水准，你至少有个立场来催促团队加把劲儿：“ 我不知道我们最后的推出日期会延误多久，但这次的里程碑已经确定我们落后了 n 周。 ”

法则 35

Nobody reaches the ZD mile-stone until everybody does

所有组员一起到达零缺点里程碑

如果有一个工作小组遇到的困难比较多，以致前进的速度比较慢，而其他的小组已经完成份内的工作时，最好让其他人支持比较慢的这一组；团队工作应该平均分配，其中有一组做得特别快或慢都不是很好的现象，毕竟，团队工作是全部都做完才能算完成，只要有一部分尚未完成就等于失败。

有一种很不好的情况是（我真的遇到过这种情况）“ 落井下石 ”：比较快的小组发现比较慢的小组在开发关键性技术时遇到困难，可能会使整个里程碑延后时，反而主张在这一部分加入更多的特色。避免这种情况的办法是，领导者必须让整个团队共同担负达到里程碑的责任，除非每一人都达到了里程碑，否则就等于没有人达到里程碑。

法则 36

Every milestone deserves ano-blame postmortem

完成每个里程碑后，心平气和地检讨

每完成一个里程碑后，紧接着应该做一次检讨，这并不需要花很多的时间，却能让团队下一次更好。当我们认为做到了百分之百，是不是真的没有任何遗漏？好的检讨绝不是指责任何人拖累进度。项目经理可以召集一个会议，每一个特色小组中至少有一人参加，或是请大家有任何心得都不要客气，发个电子邮件告诉项目经理，检讨会最后的结果再由项目经理发电子邮件给每一个人分享。尤其是做得特别好的事情，应该在检讨会中强调，大家以后就更知道怎么做会最好。在每个里程碑之后立即做一次检讨，是一个群体学

习的最好方法。

在里程碑之后责难某人或某小组拖累大家的进度，是愚蠢的自我伤害。讨论进度延误的目的仅限于：研究下次如何避免，加强大家对延误的警觉心，以及万一延误时最好的补救方法。进度延误的不利后果发生在属于大家的软件事业，不是管理者该去判断或核准的。延误势必增加公司的经营成本，团队成员不会天真到以为自己不受影响，所以管理者不必刻意惩罚，只要让团队明白延误对大家的伤害，下次就会特别小心了。

责难任何人拖累大家的进度是愚蠢的行为，就好像责怪叶子不可以从树上掉下来一样没有意义。

对于个别组员而言，延误进度是不好，但也具有某些教育意义。一般来说，只要是健康的团队，不论成员的素质高低或是延误的情形严重与否，发生延误的组员都会觉得不好意思，而主动试着解决它，他会尝试修正对自己的期望，改善自己的工作效率或工作方法。管理者应该帮助他，责难他是愚蠢的行为，就好像责怪某一片叶子怎么可以从树上掉下来一样没有意义。

如果你在每一次的里程碑之后都做了足够的反省，而且你的组员都吸收了里程碑给他们的教育，你的团队一定会逐渐成熟，最后每一次的里程碑都会准确无误地达成。

法则 37

Stick to both the letter andthe spirit of the milestones
把握里程碑的实质意义与精神

里程碑的做法在成本上是比较高的，而且团队得承受比较大的痛苦，但这是惟一能够确保开发成功的方式。团队成员为了协调出里程碑的衡量准则，所付出的时间精力就是里程碑的额外成本；为了能具体标示出里程碑不得不修改特色，团队的理想被打了点折扣，内心多少有点失望心痛。而更多的痛苦是来自于设法使大家都专注在里程碑上，协调大家对里程碑产生一致的价值观，为了里程碑愿意付出额外的时间和努力，即使真正的推出时间还早得很，我们却要劳师动众地练习、经历这个大家都辛苦的中间产品“假推出”。

然而“零缺点里程碑”所带来的好处远超过这一切所付出的代价，每个人所获得的经验和成长也远超过这些痛苦。实施零缺点里程碑可以让软件评核的过程在开发初期就开始，藉此可以带来团队的共识；对事实的了解和包容成了最高的团队价值，因为有太多人对人、群组对群组的信赖和承诺，不是实践就是食言。里程碑内容的定义，就是大家对自己工作的期望。从某个意义来说，里程碑就代表了团队承诺的信赖度。

经过项目初期的几次里程碑之后，在团队中已经培养出对里程碑的使命感，能够准确地判断自己可以实现什么承诺，在这样的时间限制之下，那些事情可以完成，那些不能。我曾经经历过几次“字面上的里程碑”，团队把里程碑当作形式和教条，而不去实践里程碑的实质意义和精神，这是很糟糕的。以我的经验，凡是健康的团队，都会对“一致性”（请参考法则 29）有很强烈的感受力，会极力避免任何欺骗自己或逃避现实的事情，完全自发地希望一切都是真实的，而不是被迫遵守任何传统的教条或规定。健康的团队喜欢里程碑和组员之间有一致性，不要形式化的里程碑，或是过度严格而做

不到的里程碑。说实话这是一个成功开发团队的重要品德，我曾经看过组员之间互相挑战对方，是不是在欺骗自己承诺做不到的事；在健康的团队中，组员能够“嗅”得出来彼此是否在说大话。

我曾经听过组员之间彼此挑战，是否在初期的里程碑目标上太过高估自己。

在项目初期的里程碑，把要求放低一点是可以接受的，比较困难的特色不要求在这个里程碑中完成、品质水准不必那么高、资源可以多用一些，这些都不要紧，可以当作团队成长的代价。反而是忽视已经发生的问题而不予补救，或是对于团队成长的问题自欺欺人，才是致命的大错误。只有在项目初期不断自我检验，团队才能成长，在往后的里程碑中才能学会掌握软件成功的诀窍。

法则 38

Get a handle on“normal”

培养正常的团队运作

在整个发过程中，前前后后的里程碑应该如何定义才适当？有没有一种原则或特征来判断在每一个里程碑中的团队行为是正常或病态？如何看出团队的领导有没有问题？无论如何，一定要先知道什么是正常的团队运作，才能做出诊断、治疗和预防。

诊断

在软件开发过程中最困难的活动之一便是诊断——准确判断出团队行为和产品状态在不同时间所发生的一切问题。因为创造软件的过程本来就是动态的，充满合宜与冲突、愤怒与喜悦，团队的行为是变幻莫测，基本上不能以组员的心情愉快与否、日程落后与否、或是表面上的行为与事件来判断整个团队行为是否正常。这种变幻莫测的本质，也使得领导人很难预测治疗行动的结果。

曾经有一次，我参加的一个团队因为时间表定得太过自信，而且工作目标也定得高到几乎接近完成，结果眼看着就要延误第一个里程碑（以下简称M1），此时大家都很紧张，幸而我们还算是个够健康的团队，我们也知道前面的成功不代表后面一定成功，我们愿意虚心检讨。

诊断有无繁杂

在 M1 阶段，这个团队深受目标过度膨胀所苦。在 M1 阶段的开发工作，使我们发现到有一些关键性的功能特色，特别是典范性的功能特色（请参考法则 3），显然在设计上有不够周详的地方，没有办法充分支持我们所要传达的信息，这样的设计事实上无法达到我们要改变使用者习惯的目的。于是我们激烈地争论、辩证、思考，几乎使团队分裂。最后我们终于得出结论，一个震憾性的超强新设计方案，大家都会为它而振奋——但也大大增加了全体团队的工作负荷。

原本是改良目标，对软件有益的行动，却造成了一个不良的副作用：只剩下三个星期就到 M1 了,组员们还不能确定要不要开发新设计的特色？在旧设计中是不是有些特色要取消？组员们搞不清楚这些特色的优先级，如果要

采用新的设计方案，由谁来做、何时做等等问题。旧设计也许不理想，但毕竟是我们好不容易才凝聚起来的共识，就这样被打破再重新设立目标吗？管理者在这样的情况下是否能坚持授权共决的原则（特别是这些功能特色原本是他们主张的），而且不认为我们是在找麻烦？像这类不确定的问题多得列不完。不难想象，这些问题使我们距离 M1 的目标愈来愈远。

就算没有工作量的增加，这种提高目标的事情本身就极具争议性。在我们讨论新的设计方案时，显然没有想到它会带来这么多的问题，所以不知不觉地增加了许多特色，这等于是破坏了我们对团队的承诺。不错，以团队的纪律来说我们可能很糟，但是我们确实对新设计方案中的每一项特色都做过彻底的分析，研究它对每个人或每件事的影响，并且取得了新的共识，大家都同意这些新增的特色每一个都很重要，没有包括这些的设计将造成产品的缺陷。我们的结论是牺牲掉其他比较次要的特色，重新配置我们的人力，尽量减少对进度的不利影响。

现在，我们面临了翻案的后果：在决定采用新设计的争论之后，我们回归到开发计划的现实，我们的 M1 订错了，而且时间上显然岌岌可危。我们集合公司内所有的项目经理和品保人员，对现状做一个彻底的评估，这次讨论的结果，可以说明在软件开发中的一切事情都无法预知，但却是可以被管理得宜的——我喜欢这一点。

争论不休

有一些人认为，我们其实可以稍微放宽一些 M1 的期望，依照旧设计做出 M1 的目标，在不影响原定目标的前提之下，少做几个特色。毕竟这是第一个里程碑，如果在实质意义上无法达到，至少要在形式上做到。

第二派意见认为，有无达到 M1 太难定论，一开始目标就定得不够明确，那么即使 M1 目标的字面意义也会有不同的解释。虽然基本上我们同意 M1 是应该达成，但实质上我们却用新的设计且违背了原意。

第三派意见认为，我们是因为赶不上 M1 的时间而狗急跳墙，他们认为这是我们不当地允许在 M1 目标中出现多余的功能，现在则企图掩饰这些成本。我们在鱼目混珠，模糊焦点，我们在欺骗自己，自以为有团队纪律，自以为表里一致，事实上我们缺乏控制功能特色的能力，而不敢承认无法达成 M1 的事实。

最后，终于有人指出，我们根本不是在“繁杂”（featuritis），繁杂是不知不觉、毫无目标地乱加特色，而且是小型特色，但是这些特色对团队共识和产品目标没有关连。在我们新的设计中所增加的特色，本来就是产品该有的，只是很不巧，这些特色都很小，以致在原本的设计中被忽略了，但是这些小特色加起来却对产品有重大的影响,而如今在 M1 的开发阶段发现了它们，以致不得不宣告 M1 的延误。然而如果此时没有发现这个问题，最后产品还是会因此而迟延。

就像疾病特别容易侵袭原本不健康的身体，“繁杂”特别会伤害原本就不太健康的团队——也许是市场中的落后者，或是对自身没有清楚认知的团队。在健康的团队中，对于进度的落后自然会产生一种不安，但只有强烈的热忱和信念才会使团队相信这些新增特色的重要性，而且困难终究会被克服。另一方面，太多的不确定性和工作负荷，会削弱团队的免疫系统，开始信心动摇，而不敢肯定这些新增的特色到底是不是产品不可或缺的重心，还

是又一个败笔。而此时若是市场或顾客也产生变化，使得团队疯狂地大量新增特色或是重新定义产品，最后必导致失败。

我们在 M1 所决定的新增特色，都是经过彻底的分析，而且大家都有共识，愿意为了这个理想而承受增加的工作负荷，所有的重要工作同仁都同意我们为了这些特色，势必将进度延后。然而如果没有这些特色，我们就无法改变使用者的习惯，也就失去典范性功能特色的意义。为此我们的进度和人员都得重新调整，但所有的人都同意，发展这些特色是正确的决定。

M_1 的本质

上述的结论对 M_1 是有特殊含义的，我们把这种经验称为“ M_1 的本质 ”（M1-ness），以便与一般的病态性团队行为有所区别。我们不认为这是“ 繁杂 ”（featuritis），因为我们确实地、彻底地、不遗余力地分析过这些新增特色，这是与繁杂最大的差别。此外，这次的设计翻案，确实是对原设计的缺失有重要的改正，而这方面的贡献远超过纪律不良的负面影响。

基本上我们要了解，M1 正好是把这类问题理清的时机，M1 可以让你发现原来设计时忽略或模糊的地方、进度表上的弱点，以及再次凝聚团队共识，对目标会有更清楚的概念。从这次有点惨烈的经验中，我的结论是：从此以后大家对 M1 都特别注意，修订 M1 的目标必须有团队共识，而且确实可以达成。

法则 39

A handful of milestones is ahandful

里程碑不宜太多，才好掌握

有很多人主张，六星期到三个月是里程碑之间比较理想的时间间隔（interval）。如果你的团队比较小（譬如十到二十人左右），或是目标较小，就可以用比较多的里程碑，因为里程碑对小型团队的额外负担（overhead）会比较少。而小型团队如果经历过比较多的里程碑，会迅速成长和凝聚共同文化（请参考法则 7），这是附带的好处。

但是以我的看法，任何团队的里程碑数目如果超过 7 个，就过多了，至少是个不寻常的现象。因为在一般情况下，任何人都无法预测那么远的未来。我的经验是三到四个里程碑，再加上要推出的那一个，应该是最理想的数目（请参考法则 18）。

法则 40

Every little milestone has ameaning（story） all its own

每一个里程碑应有专属的宗旨

每一个里程碑，无论大小，都应该有它成立的理由。记得我们刚刚在谈论 M1 的本质时，提过团队内部发生的激烈争辩吧，我们希望达成 M1 实质上的精神，不亚于我们希望达成 M1 形式上的目标。里程碑的宗旨就是设立它的理由，可以被简短的一两句话表达，同时也能描述达到这个里程碑对团队的意义。就理论上来说，任何对里程碑的争论都可以回归到它的宗旨而获得解决。里程碑的宗旨可以清楚地表达它的目标，我故意不称它为里程碑的目标，因为里程碑的宗旨应该包括了团队与它的互动、团队的士气，以及里程碑所

隐含的信息，也就是包含了它全面的意义。如果单从 M1 的目标来看，大部分的人都无法产生去达成它的动机，或是联想到它所代表的报酬，M1 的目标是一大串的菜单列，是我们要去做的工作（虽然目标也可能是错的，谁教软件这么不确定呢）。一张清单不能引起人们的热情、专注，以做出伟大的软件为理想。人们只有为了有意义的事情才会愿意牺牲奉献，只有里程碑的宗旨才能让人们愿意付出自己的心灵、精力去换取某种结果。

在 Visual C++的开发过程中，有几个不错的里程碑宗旨，值得参考。在典型的第一次里程碑，我们通常会把目标放在“ 狗食理论 ”（dogfooding），意思是要使用开发中的产品，才知道应该如何进一步开发（Dogfooding is usingthe product under development to further the developmenteffort.）。这是源自行销学的理论，原创者是史提夫·巴摩（teve Ballmer），由微软将之发扬光大，他的名言是：“ 狗食工厂应该试吃自己生产的狗食。只有亲身试用自己的产品，才会真正了解自己的产品。 ”对于像 VisualC++这样的程序开发工具，这个概念尤其重要，因为它就是用自己早期的开发工具写出来的。对于我们团队而言，彻底实践狗食理论的意义是，新版本一定要比旧版本更具生产力，即使新版本尚未完工或还有错虫。表面上我们没有特别强调狗食理论：管理者不会每天巡视办公室看看是不是大家都在试用自己开发的软件。然而，全体开发团队都太清楚狗食理论对我们的重要性：如果我们在每一个组件上都发挥狗食理论的精神，我们的产品一定会变得愈来愈好。借着经常使用自己开发的软件，我们找到了所有的错虫和瑕疵，生产力提升了，也更懂得掌握设计的效率。

新版本的试用者通常是精挑细选过的。

另一点是我们在拟定里程碑的宗旨时会特别注意“ 什么时候将什么产品交给什么人 ”。为了保护我们自己的荣誉，将新版本交付出去时会特别小心，通常对象都是选择性的，因为在错误的时机交给错误的人会造成团队额外的工作负担。因此，里程碑可以说是我们准备将中间产品交给某些早期客户来试用。

所以，我们第一个里程碑的宗旨是：“ 所有的产品组件都已经被开发团队试用过，并准备将中间产品交给微软内部的单位试用。 ”简单的说法就是“ 内部试吃狗食 ”。而后面一点的（就算是第三个吧）里程碑，它的宗旨也许是：“ 所有的特色都已经完成，而且软件的使用者接口也不再更动，准备交给一些外部顾客试用。 ”

里程碑宗旨的关键点在于：

·宗旨必须点出里程碑的精神所在。

·判断这个里程碑有没有达到，是看里程碑的宗旨是否被实现。

·在开始工作以前，所有的人都已经充分明白并接受这个里程碑的宗旨。

法则 41

Look for the natural milestones

寻找自然出现的里程碑

虽然很难精确地把软件开发过程依时间顺序通通记录下来，但有一些事情是必定会自然发生，而且对整个开发活动就像是分水岭一样，有划分的作用，这就是自然出现的里程碑。请注意我所谓自然出现的里程碑，是指“ 正

常”的，而不是“常见”的，正常的是指开发过程本身没有重大问题。以下是六种自然出现的里程碑：

1.产品设计趋于稳定。

2.中间产品被明确定义。

3.团队真正了解要花多少时间和努力才能完成目标（通常这会发生很多次，而且多半是进度落后的时候）。

4.产品设计被删减，或是资源增加，或是进度延误，或是三者同时发生。

5.开发活动停止。

6.产品进入除错或稳定阶段。

这六个自然的里程碑不仅在整个开发过程中出现，也会在单一的里程碑中出现。稍后我们会用较大的篇幅详细讨论，但现在请大家注意的是，每一个里程碑所意味的行动，跟所有的里程碑加起来所意味的行动，应该是相同的，跟整个技术计划所意味的行动，也应该相同。

健康的团队在每一个里程碑所表现的行为应该是可预见的，称作“里程碑的自然模式”（metamilestone pattern），管理者应该予以重视和培养。如果里程碑的自然模式没有出现，就代表团队出了问题，管理者应该采取某些治疗的行动。到目前为止，我所观察到的自然里程碑可以归纳为六种事件，但各种事件会持续多久则是无法预测的，比较可以确定的是，前一个事件结束时，下一个事件一定要跟着出现，前后事件之间可能会有重叠，或是几乎同时发生；这些事件会依序发生或是同时发生，是取决于团队的沟通频宽，在沟通性强的团队中，信息传递得像病毒一样快，这六个事件就比较可能同时发生。

第一个里程碑：设计趋于稳定

最开始的时候，由许多特色组成的产品设计是非常抽象的，是一种令人又兴奋又迷惑的东西，兴奋是因为自己的想法或创意即将实现，迷惑是因为不太确定它究竟是什么模样。一切的人事物（技术计划、团队共识在里程碑之前已经形成）都显示出有这些特色是要完成的，但真的问起来，又没有人确切知道该由谁、在什么时候、做什么事情，才能完成这些特色。这种情况我称之为“软件梦想综合症”（the software dream syndrome）。在产品的设计阶段，这种细节不明的现象会不断地重复出现，直到详细的工作分配形成为止。

也许你会听到开发人员说：“喔，是的，我们当然要在第一个里程碑中加入 footbar 的功能；我正在弄出它大概的模样给你瞧瞧。”

而品保人员可能会说：“Widget 的特色确定要在第二个里程碑中开始动工，但我还不知道该由谁来负责开发。”

在传统上，或者说至少是在有制度的公司内，对于程序开发一般都要有几项文件的前置作业，包括需求描述、档案或数据库规格定义、各种程序逻辑图等等，不胜枚举。虽然这些文件有助于将程序要做的事情具体化，容易厘清细节，但是基于以下的理由，我个人反对花时间做这些文件：

·在设计趋于稳定之前必会频频修改，因此使得文件一再重做，结果保持文件的更新变成了一种道德或服从命令，而不是因为事实上真有这个需要。

·遵守很快就会过时的文件来做事，势必限制了弹性，错失宝贵的机会，而且对健康的团队来说是负担的加重。

·文件的作用是在开发工作开始前，把所有的事情都确定下来，结果也会限制了程序的发展。与软件有关的很多事情是在开发过程中，才会确定该怎么做的，而且这时候才能确定怎么做最好，有时候开发人员会有意想不到的创意，为产品增色。所以，如果将软件中未知的部分完全排除，就等于是宣告软件的结束，而非开始。

在软件开发的过程中，最重要的事情不是做出你承诺要做的事，而是在有限的时间、资源限制下，从各种可能的方式中做出最明智的选择，使结果达到最佳化。

产品设计会藉助组员之间各种形式——电子邮件、规格定义、产品模型、交谊厅的聊天、用白板的小组讨论——的沟通、讨论，而由变动渐渐趋于稳定，当产品设计逐渐确定，下一个阶段就可以准备开始。管理者应该注意是不是所有的争议都已经被共识所平息，检查是不是所有的功能特色都包括在设计中，是不是所有的人对产品设计的认知都很类似，如果答案是以上皆是，就可以进入下一个阶段，否则产品设计还不能算是成熟。

第二个里程碑：中间产品被明确定义

如果能用清楚、简明又可信的方式来表达中间产品，没有含糊、复杂或诡异，就可以说“中间产品被明确定义”。在项目正式进入开发阶段时，以及需要检验里程碑是否达成时，我们需要的都是明确的定义：产品究竟该是什么模样、有什么工作要做完、由谁来负责完成、品质的要求到什么程度，这些都需要有明确的估计值，虽然不必要和照片一样的精确程度，但至少要像印象派画作一样的明确。

随着产品设计的确立，品保人员、项目经理、开发人员和文件人员就可以组成特色小组了，特色小组各自将负责开发的特色具体化，更清楚地描述他们的需求和目的。他们开始花大量时间讨论和沟通，因为他们有共同的目标和工作，他们关心的是同一件事，而他们对产品设计的认知也相同，于是，开发的细部工作项目逐渐在组员的心中自然成形，为了将这些工作依时间先后顺序安排，并拟出进度的草案，沟通和协商也许要重新展开。

第三个里程碑：团队真正了解要花多少时间和努力才能完成目标

当然啦，在你完成开发工作之前，你永远无法得知真正所需的时间。当产品设计定案，中间产品被明确定义，组员会有一种幻灭似的失落感，第一、因为产品的外观似乎不如想象中的美丽，第二、真正该做的事永远比想象中的多，第三、团队发现自己需要更多的资源、较小的目标、更多的时间——或者以上皆是。

在这个时候，无论团队多么经验丰富、领导多么卓越，总是有一股失望的气氛在弥漫着，那是一种抑郁而消沉的感受，好像自己被击垮一样。其实这是个好现象，表示组员们对事实有进一步的认识，往后才会因为明白事实而感到自在。幻灭是成长的开始，“软件梦想综合症”于是消失。在健康的团队中，对事实的认识会引发另一波行动的高潮，在我的团队中，我们把它称作“战斗会议”（warroom meeting），我们会连续开好几个小时的会，来检讨我们现阶段的状况，并研究出问题的对策。

团队中弥漫着一股失望的气氛，那是一种抑郁而消沉的感受，好像自己被击垮一样。

在前三个自然里程碑中，最重要的事情是：解决那些应该理清的未知。然而，当事情一旦被弄清楚，工作通常是只会多不会少（模糊的面孔比较美

丽吧），组员可能会被平添的信息吓住，但是健康的团队会面对这种情况，设法克服恐惧。

第四个里程碑：产品设计被删减，或是资源增加，或是进度延误，或是三者同时发生在这个阶段中，团队会利用“软件的三角形”（请参考法则 28）来解决所遇到的冲突。如果一切顺利，这时候时间应该刚好是整个项目（或是某一个人为的里程碑）的前四分之一。我们很有理由相信，愈是健康的团队，前三个里程碑会走得愈快，并且会有充分的时间来解决三角形告诉他们的冲突，或是重新建立共识，这差不多是该对里程碑计划稍加修正的时候了。如果一切顺利，虽然对里程碑的某些期望可能需要调整——也许是有些特色要取消、也许得增加一些资源等等，里程碑的宗旨应该是到现在还维持不变。

在里程碑到达之前，谁也无法保证你能如期到达。

健康的团队会在此时展现出弹性和反应能力，以准确地达到这个里程碑的要求。你不会希望里程碑延期，这是最后的手段；你也不愿意见到资源被无限地要求增加。在这时候，你最需要看见的是组员之间形成了无数的承诺，培养出解决问题的效率，并准备好表现出最优异的里程碑行为模式。

第五个里程碑：开发活动停止

在某一个时间点，所有撰写程序的动作会全部停止，也就是特色全部开发完毕，此时必须禁止一切修改，以便软件进入完成和稳定阶段。开发工作完成后就不必再写或改程序，这听起来好像是废话，但事实上有必要特别标出这一个里程碑，因为原本预估的时间需求可能不对，这个里程碑等于是计算耗用时间的依据。必须确定软件不再需要开发活动，才算到达这个里程碑。

第六个里程碑：产品进入除错或稳定阶段

如果你能估出“净除错能力”（net bug fix capacity），即“错虫清除率”（bug fix rate）减去“错虫发现率”（bug find rate），而且“净除错能力”是正值，那就差不多可以算出第六个里程碑要花多少时间。而且基于本次的经验，对于以后类似的项目，就可以降低“错虫发现率”，并达到更高的软件正确性。这时候你需要找出这些问题的答案：有多少错虫？要多久才能把错虫全部清除？有多少比例的程序代码需要整个重新检讨（regression rate）？

当这些问题被认真地讨论时，你大概快接近里程碑的尾声，而且所有能发现的错虫全都被清除了。你会发现利用错误等级评估法（bug triage）来解决问题，会比随抓随改的方式要轻松多了。

法则 42

When you slip，don ' t fall

如果滑了一跤，别就此倒地不起

进度落后（slip）？不要紧，这并不是世界末日，就像感冒发烧一样，令人不舒服，但也表示身体正在自我治疗。

从另一个角度看，进度落后是团队从软件的梦幻中惊醒，开始面对现实。好像一种醒悟，一种“现在，我懂了”的心情。进度落后像是从一个怪异的、层层包裹着你的梦境，当你从第一层梦境中醒来，想道：“好吧，现在我醒了，我刚才是怎么了？三个月来竟然蠢到完全没发现这个明显的错误——延误了 footbar？啊！我终于醒了。”若把进度落后当成道德上的罪孽，人们

就会全力拒绝面对它。

三个月之后，也许是三个星期之后，你又发现了一个明显的延误，你又想道：“哇，这次我真的是醒了，现在，我懂了。”每一次的觉醒都让你进入一个“真实”的世界，但也许有一天你再次从这个“真实”的世界醒来，发现刚才的“真实”其实不是真正的真实。

你可以把这种不断的觉醒当作成长的过程。当然，每一回觉醒都带给你更多的知识和经验，并让你更接近真实，更丰富你的人生。这个多层梦境的醒悟计数器是每个新项目从零开始，每一次觉醒加 1，如果你已经有了 n 次觉醒的经验，你就是从 n 开始你的醒悟计数器。

任何项目都有可能碰到进度落后和觉醒，但有些觉醒不一定是跟随进度落后而来，这很有可能，但有点危险，并且不是好事。以下是你对进度落后应有的正确观念：·进度落后与道德无关，请切记！这并不是失败或做错事，应该受到责罚，这是创造智能财产过程中无法避免的。千万不要有道德上的罪恶感或良心上的负担，并且要求你的团队也要这么想。因为组员们最怕被威权形象责备，他们会因此而陷入一种很微妙的精神压力中，在讨论时拖拖拉拉不着边际，在恐惧犯错和渴望荣耀之间挣扎，在责难和赞誉之间游移，结果会尽全力摆脱这种压力，会胡思乱想，因此而消耗所有的心力。所以，千万不要让任何人对进度落后有道德上的联想，认为这是不应该或不对的事情。

·不要隐瞒事实。人在遭遇冲突或危险时，一定但愿没这回事，而有逃避的倾向。在进度落后时，你和组员们都难免会不想谈它，但这时候特别要克服这种天性，发挥你卓越的领导力，把事情变得不一样。不要瑟缩在你的办公室中。喃喃自语：“我不敢去看，进度迟延了，……不，我不相信有这种事……”勇敢地走出去面对问题，你要打败问题，而不是逃避问题。

·化阻力为助力，利用进度落后来激发效率。对于进度落后的认知是一种醒悟，组员们开始有重新振作的精神，有最新的信息，有最新的看法，创意和想象力也可能有新的境界，领导者必须得把这些新兴的朝气导向最有效用的地方，不要管传统规矩的包袱，进度落后是考验团队弹性的最佳时机。

进度落后不是问题，被进度落后吓倒才是问题。进度落后并不代表产品的难度太高而无法开发。但是如果进度已经落后却未被察觉，那表示组员们不思考、不观察、不讨论，此时组织可说是濒临瓦解了。

善用你的迟延，这是最能看出你领导能力的时候，此时也是组员最脆弱也最需要你的时候，在这个时候组员最能把你的话听进去，此时组员的学习能力最强。如果你在办公室内激动得大喊大叫，指天骂地，那就错失了赢得组员的心的大好机会。你必须说：“OK，进度落后了，让我们来看看问题出在那里？……今天下午五点在会议室，我们要检讨每一个细节，问题一定要设法解决！”当组员了解到你不是企图卸责或算帐，而是真诚地想解决问题，就会乐意把一切开诚布公地摊开来谈，大家一起研究问题，从各种角度去设法克服问题。“进度落后”反而变成大家宝贵的成长经验。

进度落后

里程碑达到了吗？其实这个问题很难有完全客观的衡量标准。由于产品设计可能会变，而品质本身就是无形的，所以里程碑的到期日也可能会变，因此，里程碑到达与否，变成了一种人为的判断：当需要回头修改的项目愈来愈少、里程碑的宗旨达成、组员对于下一个里程碑跃跃欲试时，你差不多

可以宣布里程碑成功。当然这一刻是很难决定的。对于这个问题，我只能举个反面的例子来说明这个观念：比方说你去问路，人家的回答是：“你一直往前走，若是看到尖塔教堂，就表示走过头了。”如果你已经接近里程碑，但开发活动还在进行，就表示已经走过头了。身为领导者，你必须时时掌握软件的一切状态，必须知道在每一个阶段应该进行那些活动，又该停止那些活动。

曾经有一次（也许不只是曾经），我们接近 M1 了，但一切工作都陷入混乱，我们只剩下两星期，而软件的情况却糟得难以想象：错虫数（bugs count）实在太高了，虽然每个程序设计师的错虫率还在合理范围，但加起来的错误率却高得惊人，从我们每天的错虫增加数目显示出软件距离稳定还差得很远。

我们没有办法每天做软件建构（请参考法则 32），虽然主要的产品组件多半都能建构成功，但就是有那么几个不成的，表面上都能编译和连结（compile＆link），但就是建构不起来。这个问题大大地妨碍了下一步骤工作：“猎犬测试”（sniff test）和程序代码分析。当其中的一个定点问题好不容易被找出来解决，这至少又花掉了一天的时间（请记得，在里程碑的后期，一天就可能耗用是百分之十的资源）。

组员和领导者并没有感到特别紧张，通常在接近里程碑的时候，组员们都知道自己已经尽力，但不知怎地，就是觉得不对劲，里程碑的精神似乎不见了。

事实上，这个问题的主要原因（也是一般常见的状况）是出在沟通和协调不良，起于至微而酿成巨祸。当一只程序置入时，就改变了另一只程序的假设前提，结果另一只程序就遇到了意料之外的状况。A 小组依赖 B 小组的公用程序，A 小组改好也测试过了自己的程序，但前提是 B 小组维持在第 n 版，然而 B 小组却改到了第 n+1 版，虽然改得很少，但恰好是悠关 A 小组的部分，B 小组误以为不会影响 A 小组的程序，所以没有告知 A 小组，结果 A 小组的程序就建构不起来，因为它应该与 B 小组的第 n 版搭配，结果 B 小组已经置入了第 n+1 版的程序。

领导人应该制止这种情况吗？

绝大多数的项目经理一旦发现这种情况，第一个反应是采取补救行动——成立特别小组来接管事务。但是这么做可能会造成一些问题：最严重的是，这种行动的代价甚高，虽然这么做一定会使组员非常注意软件的状态和风险——我把这种活动称作“强烈焦点”（radical focus）——你等于是用领导者的职权来使组员就范，这种敲响警钟的方法不能太常用，否则团队就会发生“狼嚎症状”（cry wolf syndrome），懒得对老板太认真，老板只是叫得凄厉而已。

如果进度落后的情况太频繁，是很危险的。进度落后变成正常的事，此时运用领导者的威严来提高团队的危机感，这种手段只有在士气低落时才适用，这时团队需要一点刺激，就是“强烈焦点”活动：让小组在同一时间内担负几个小项目（大概不超过两个），但是这几个项目的目标是南辕北辙、毫不相干的。

而当“强烈焦点”展开时，即使是健康的团队也多少会反弹，觉得领导人在“反授权”（disempower- ment），于是一切不顺利的牢骚都算在“当权官员”的帐上，把这项措施解释成不尊重团队和团队的决定。防卫心理升

起，而且开始情绪不平，甚至可能因此怨恨起来。即使是最优秀的领导者在采取“强烈焦点”措施时都会遭遇一段抗拒期。在违反对方的意愿之下教育他，虽然做得到，但要辛苦很久。

同时，你要和团队中最优秀开发人员的反抗权威心理作战，他们可能不会很好沟通，而直接把你当成笨蛋（请参考法则 4）。在另一方面，你也有可能误判士气的低落程度，这时候麻烦就大了，健康的团队会对你反弹，如果你死不认错就会丧失组员对你的尊敬和忠诚，要不就是你的自大使你愈走愈远；而团队如果不是那么健康，不敢对你谏言，那你绝对会陷入万劫不复的失败。

在你决定采用“强烈焦点”时，以上所谈的故事都是你的成本和风险。你的“固定成本”至少是几个工作天（大概一星期左右），而你必须抓住组员的注意力，保持他们的兴趣，让组员明白你是玩真的，让他们愿意告诉你心里的感受，让他们开始做点实际的事情。在这个起步阶段所需的时间视团队的大小、沟通的意愿与能力而定。

现在我们回到接近第一个里程碑时的原地踏步，领导者要团队有什么样清楚的认识（awareness）？如何做到这一点？然后会引起什么行为？

在我们的案例中，要团队清楚认知的是 M1 迫在眉睫，而且除非过了 M1 这一关，否则我们不可能完成产品；我们也希望让团队练习使软件“整装待发”、完工交货；最后，我们希望团队经历一下不成熟的团队合作所带来的可怕后果，而从中学到我们每一个人都有等量的责任，也是全部的责任-就是达成 M1 里程碑。

在我们的案例中，“强烈焦点”的施行是不合适的，因为团队的问题不在士气低落，而且事后发现，团队的反弹非常强烈。我们应该为真正的紧急情况保留子弹。

那该怎么办？

很明显地，我们不可能让整个投资都押 M1 这个岌岌可危的一注，我们知道一定不能让问题持续下去。我们现在正处于瓶颈，无法建构软件的不良效应正扩散开来，这一点可以从超高的错虫率可以证明。大家对程序的置入动作更谨慎，必须得仔细检查是否会妨碍到别人的程序，也就是速度慢了很多。错虫数仍然居高不下，为了除虫而做的修改仍然会发生前述版本不同步的问题。当软件无法建构的状态持续太久，会使得更多的程序搭配到别人的旧版本，结果使得软件建构更加成功无望。我们很关心这种情况，组员对危机的感受度似乎无法和工作的进度成正比。

团队合作是相互影响的。为了完成整体的工作，你必须划分出局部的工作。你可能让三人或四人组成一个小组，这一小撮人的影响力就会扩及全部的团队或工作，相对地，这一小组的失败也会扩散到全体，导致软件严重的伤害。此时你的重心集中放在问题根源的小组，这会比企图整顿全体，结果却头痛医头脚痛医脚地到处乱转要有效得多。

我们经过一番诊断和试疗，终于决定：在我们做到每天都建构成功以前，禁止所有的程序置入。有三个主要的原因迫使我们非这么做不可：第一、没有协调妥当的置入动作，会使很多其他的小组受到影响，最后使得大家都无所适从；第二、负责建构的小组经验不足，因为建构的任务不久前才从开发部门转移到品保部门；第三、有一个小组总是通不过测试，它是问题的根源。

好吧，以上这些症状，告诉我们什么？这个团队的心理状态到底出了什

么问题？

看看团队出了什么问题？

在起始阶段我曾经提过的一项理论：从软件能够看出团队的心理状况（ psyche）。现在事情不对头了，团队显然无法发挥应有的工作效能，此时你必须向问自己，团队的这些行为是否透露出什么样信息？

首先，我们来看软件始终无法建构的问题，协调不良的程序置入动作（surprise check-in）代表团队缺乏整体性的自我认知：小组之间缺乏交谈。每天的开发目标和程序撰写都是纯组内的，小组之间的程序开发动作太不协调，彼此相互掣肘。每一个小组单独来看差不多都是朝着 M1 的目标前进，但每个小组的工作成果却无法组成产品。个别组员的贡献在组内都很不错，但到了组外就完全看不出来。

第二个问题是建构小组的经验不足。品保人员没有做过建构，以前都是由最资深的开发人员负责每天的建构。这一群开发人员在技术上有足够的能力解决任何建构失败的问题，因此开发人员已经习惯对于建构小组期望过高。这一群资深开发人员像看门狗一样为软件建构把关，解决任何问题，甚至指导其他人关于程序开发的知识和技巧。而新接手的品保人员没有办法对软件（和其他开发人员）照顾得那么细微，没有技术能力来立刻解决建构失败的问题（例如 debug），而且这也不是品保人员的责任。另一方面，品保人员正试着融入团队文化，过去是开发人员最大，现在开始要在团队中建立品保人员应有的角色及工作模式，品保人员一时还无法坚决地要求开发人员把程序改到什么地步。

因此，开发人员对于建构小组的“ 不良 ”工作表现觉得失望，无形中，开发人员开始认为建构小组都是笨蛋，他们已经习惯有资深高手替自己修缮程序，程序设计设计师（尤其是那些负责核心部分的）已经被宠坏了，忘记去思考自己的程序与整个软件的关系，也不认为自己的程序必须处处配合别的程序才行，每个人都认为我只管写自己的程序，建构成不成功不关我的事。现在资深的保姆不再为大家打点这个清理那个，每天都有高品质的程序是每位开发人员的责任。建构小组的工作是把品质好的程序建构成品质好的软件，维护可预测的建构程序和结果，找出瓶颈所在，并且监督开发人员改正自己的程序。开发人员对于这项责任归属一开始当然是不情不愿的，他们原本对于建构失败的反应是“ 建构小组今天没把事情做好 ”，现在，要把他们的观念转变为“ 是今天的程序不够好而无法建构 ”。

第三个问题是有一组总是做不好，他们的程序总是无法建构成功或趋向稳定，一直通不过测试。这并不意外，这部分程序长久以来受到忽视，没有足够的开发和品保人员照料，项目经理根本没有为它付出关爱，也许每天的建构失败就是它的不平和控诉吧！这一部分程序对产品而言其实是挺重要的，只是不知道为什么它老是被当成二等公民。于是我们重新分配资源，拨出足够的人力来把它弄好，我们得补偿过去对它的忽视和差别待遇。从团队的心理状态或产品本身，都可以清楚看出我们过去没有给这一部分程序应有的重视。

总而言之，团队没有协调好，责任没有划分清楚，而且没有相互为对方的程序负责、全体共同对软件负责的成熟心态，管理者也疏于分配资源到适当的地方。等到我们把问题诊断清楚，收回“ 强烈焦点 ”的错误措施，开始真正对症下药的治疗行动。

让组员学习彼此互相配合

很明显地，由于团队缺乏共同的任务认知，小组与小组之间既不交谈，也不会朝共同的工作目标努力，因此我们必须充分利用团队工作的扩散效应。也就是说，如果你在某处解决了一个问题，就等于是对全体解决了这个问题。项目经理是团队中理所当然的领导者和灵魂人物，他必须有很好的管理者形象，负责找资源、保护团队、带领团队迈向成功，他同时也是执行长，在别人眼中他就等于产品的总括，每一个人的事情都是他的事情。

项目经理或上级主管可以命令团队应该维持某种程度的沟通，也可以用鼓励的方式达到更好的沟通效果。然而，若是一开始组成团队就期望很有效率的整体沟通，这野心太大了，失败的可能性会很高：因为团队中的事情太多，组员与组员之间的关系太复杂，而且随着团队人数的规模而呈指数成长。比较好的做法是，让几位项目经理互相讨论他们的工作状况、目标、希望以及长期或短期的技术计划，这种层次高、人数少、没有隐藏的沟通虽然困难，但并非不可能。然后让各项目经理去维持自己小组的沟通。用这种方式，团队会很自然也很快地建立起整体的沟通，虽然你不能命令或创造团队整体的沟通，但你可以起个头、可以培养。

虽然项目经理会有很多位，各自负责不同的产品领域，但其中有三位的专长和职权可说是综览全局：一是三人中最年轻的，但资历比较丰富，曾经领导过多次产品推出工作，另外两位比较年长，虽然从未担负过如此重任，但也是快速窜红的“新秀”。三人之间的沟通很差，其中一个原因是，最年轻的那位没有给另外两位足够的引领，让他们能更进入情况。

部分原因是这三位项目经理分别辖属于不同的老板。还有一个原因是在M1的早期阶段，此时项目经理们还未能充分掌握自己应该担任的角色，他们会倾向萧规曹随，看看前一位担任此职务的人是怎么做，以为自己照做就够了，或是他们以为有些事已经有前人完成，自己毋需费心，而且三人彼此之间的沟通相当少。这个问题反映在每天的建构上，我们必须将建构失败的问题和三人之间的沟通问题一起分析，深入了解二者的关系。

代罪羔羊

最常导致建构失败的原因是程序置入时，版本发生不协调，结果发生更难建构的连锁效应，另一个无法建构的原因是置入的程序其实尚未完全确定没有错误。项目经理当然不可能不知道本组现在正在修改什么程序，项目经理（或至少是品保人员）应该把不够完美的程序退回去给开发人员修改，并追踪修改的进度。程序置入前的品质没有被重视和把关，当不严谨的程序进入了建构程序，或是开发程序应有的纪律没有被认真遵守，在心态上就已经开始散漫，开始推诿塞责——“都是建构小组没有做好工作”。

推诿塞责是团队合作失败的必然现象，尤其在责任感很狭隘、误解或扭曲的环境中，更容易泛滥成灾。对道德标准高的某个人或某个群组，特别容易在推诿塞责中受到伤害。代罪羔羊存在，就表示团队的心态有问题，这是一种病态性的防卫反应，好像只要有别人可以责怪，自己就可以没事，人们直觉地想找个替死鬼，以便证明自己的表现没问题，而忽略了整体的情况正在急速恶化（那不是我的错！）。推诿塞责是将团队的整体问题与自己隔离，把问题简化、窄化的一种企图。

推诿塞责当然是人类原始的、不成熟的恶习之一，但对于团队的心理状况却有短期的改善功效。这也就是为什么推诿塞责的现象这么普遍，而且是

在短期合作的团队中。然而，推诿塞责毕竟不是长久之计，短期的好处实在没有什么大用，凡是聪明、有自信心的人都不会采用推诿塞责的方式，只有比较年轻、比较怕受伤害的人才会这样做。

很不幸，推诿塞责是会生根的恶习，它会扩散、传染出去，直到整个团队的工作绩效低落到大家都无法忍受为止。区分“推诿塞责”和“合作无间”最明显的指针是看看团队是否对于被当成代罪羔羊的牺牲者视若无睹。大部分的受害者会觉得自己是被罗织入罪，然后这种压力会造成受害者更强的防卫心理和“太极拳给他打回去”的行为。推诿塞责的目的是逃避检讨问题，姑息问题的存在，找个代罪羔羊替自己承担责任。推诿塞责的结果会像滚雪球般愈来愈严重，太极拳大战会从小组打到管理者，再扩大到管理者的管理者，最后会造成组织的致命伤。

推诿塞责对组织的破坏性可以分成两个层面。那只成为众矢之的的倒霉羔羊会被自己的同仁当成坏蛋，被排斥和被轻视的情况使他不得不学习各种保护自己的方法，当然包括把责任踢回去，或是另外找个人来当代罪羔羊的代罪羔羊。无论团队原来的工作效能如何，绝对会因此而急剧下降。一旦第二只倒霉的羔羊（很可能是宰杀第一只羊的人）也面临被指责的危险，就会开始出现两种对组织具有破坏性的后果：第一，因为把责任推给别人是这么的容易方便，所以真正的问题就更不会被人分析研究和解决，而且根本没有人会注意到，人们忙着拼命把责任向外推，结果加速了推诿塞责的病情扩大，更快速地侵蚀组织原本就偏低的工作效能；因此，只要有一个人变成倒霉的代罪羔羊，就开始了相互推卸责任的恶性循环。

第二个破坏性的后果更加严重，因为内部互相踢皮球，同事彼此之间关系的平衡遭到破坏，必然要引来高层主管的干涉（讽刺的是，也许高层主管是为了解决推诿塞责而加入战局的），于是局面变成各执一词的两造双方，而高层主管要做出仲裁。判决的结果很可能双方都不满意，因为实在太难找出适当的方法解开这个结，当然会造成彼此的信任丧失，忠诚度降低，对于主管的敬意也打了折扣。

任何领导人都必须对“推诿塞责”的情况有正确的认知，它是一种团队心理的病征，尤其是正在扩散的推诿塞责，更是团队工作效能的严重伤害，长期下来必会危害整个组织。

在我们的案例中，项目经理应该有责任为建构小组设立明确的目标，与他们一起建立团队对建构小组应有的期望：现在不再有资深开发人员为大家修整程序，将软件建构成功是每一位开发人员的责任，而建构小组的责任则是把不够理想的程序退回去，并协助开发人员找出程序无法建构的问题症结。

到最后我们为了解决这个问题，不得已将 M1 的时程延后几天或一星期。延期已经无法避免，但借着延期的动作，让组员更清楚了解到我们的团队其实很脆弱，即使初期曾经信誓旦旦，即使管理者对组员充分信任而不采取强烈措施；让延期在组员心中刻下难以忘怀的教训。现在，我们不得不采取扼阻的行动，我们停止所有的开发动作，直到软件能建构成功为止。而且除非程序经过确实的测试，不准任意置入。

于是我们让开发人员了解，他们的工作是多么重要，万一有任何瑕疵，将会影响整个软件的建构，所以程序在置入前一定得坚持完全没有错误。最后我们加派人手给那个孤儿程序，让它很快地修改妥善，不再妨碍建构。

沟通方式与效果

事情的沟通方式等于暗示了事情的意义。一小段没说什么特别事情的电子邮件，人们也不会对它特别注意；而当面开会，则是最强力的沟通法，表示所谈的事情非常重要，或意味着事情不是我们这群人加上我们的顶头上司就能决定，得来点组际协调。

有关于程序置入程序的问题，被安排在“特别会议”（special war room meeting）中讨论。以我们公司的文化，特别会议是不同于一般会议那样送个小开会通知，该到的人到齐就行，特别会议意味着：第一、与产品的推出有关；第二、一位以上的资深经理参与；第三、讨论的主题非常重要，而且需要共识；第四、虽然没有形式上的规定，所有的小组领导最好是参加，因为这个决策必定事关重大。通常是由最高项目经理来召开并主持这个会议，但其他的领导人也可以召开这种特别会议。

请注意这和宣达命令的会议是完全不同的，如果资深项目经理说我们要怎么怎么做，然后大家照章行事，虽然也会讨论，但基本上是主持人所许可的讨论范围。既然是宣达命令，已经没有什么讨论的空间，讨论只是更确立决策或更了解决策的用意。如果与会者中有人比决策者更聪明、更有想象力、对问题有更深入的了解，或是有更好的办法，他会觉得挫折和气愤，其他人则因为不能使用更好的办法而感到遗憾。在会议的讨论过程中做出决策，比较容易得到高明的决策，执行起来也会有效得多。通常是有人发现重大问题，而且已经想了又想，询问过专家的意见，也许已经有了初步的行动方针，在会议中邀集众人的看法；通常与会者已经在会前对这个问题有某种程度的了解，甚至已经思考过一些解决方案，即使不在事先排定的议程内，也可以提案讨论无妨。通常在这类会议中，提出的意见或建议都是经过深思熟虑，初步的行动方针被补充、被改良，共识在这种过程和气氛中凝聚。与会者有意见的话，应该让别人都了解，然后在大家都能认同这个解决方案的情况之下，齐心解决问题。

法则 43

Don’t trade a bad date for anequally bad date

不要因为进度落后而更改最后期限

将来有一天我退休了，真希望能到学术界研究出有关进度落后的数学公式。目前我能确定的是，第一次的进度落后往往是最严重的一次，之后因为经验的积累会使你更能掌握进度，所以，当你第一次遇到进度落后时，千万得有耐心。

进度落后的程度是与计划的不确定性成正比。就好像你在翻阅一本名叫未来的书，你无法预料未来是什么，直到你走到那一步；随着时间过去，本来不确定的事会变成已知，于是整个计划的不确定性会随着时间而降低，进度就愈来愈能准确掌握。理论上，进度落后的程度相对于计划的不确定程度，并不是数学上的趋近于零而永不等于零，而是，如果不确定的程度为零，就不会发生落后。

我相信应该有这么一个算法，用时间轴（X）和不确定性（Y），给定有任两点进度落后的值，就可以投射出理论上的项目完成日期。我相信这种算法，因为我没有一次不用到它，虽然我无法提出精确的公式或证明。但请你

睁大眼睛注意，你所经历的进度落后是否一次比一次轻微，如果是，表示你渐入佳境，如果不是，那你离目标渐行渐远。

在你知道自己什么时候能够完成以前，你往往会经历自己即将落后的煎熬。

但是，千万不要为了减轻进度的延误，而将最后期限向后挪，这种交易不划算，你会因此而信用扫地。一般来说，早在你知道确切的可完成日期之前，你会知道项目已经延误，这时候，整个团队乃至于全公司所有的人，包括外界的新闻探子（如果他们知道的话），都会问你：“既然每个人都知道延迟势难避免，何不干脆将到期日延后？”仅管有庞大的压力逼你延期，但原来正式的到期日仍然是项目正式的最后期限，如果轻言更改，团队成员会因此隐约觉得原来的日期设定不妥当，造成人心浮动。项目的正式到期日并不是不能改变，但不可因为进度赶不上而将它延后。

在进度落后时，最最糟糕的办法是估计落后的时间差，而将到期日向后顺延。这等于是将一件你确定已经做错的事情（进度落后），往后再背负下去（进度还会再次落后）；这个方法似乎是很方便的权宜之计（我们再也不必去想它），却是最极致的愚蠢。这时候你最确定的事情就是必须思考为什么会发生进度落后。

即使你现在已知的信息，比起你在项目初期设定期限时要丰富许多，但大概依然不足以让你决定新的时程表。不过话又说回来，要说明在什么情况之下可以重设新期限本来就是极困难的事。比较好的一般性原则是，除非你已经很确定各个组件产生多少程度的落后，还欠缺什么样的内容，发生落后的原因已经确实找出来，并且问题已经在克服中，否则绝对不要轻言更改最后期限。当然，要做到这个前提，必须全体团队的合作，正确的领导，以及全心的投入，在能够精算出期限的数学公式（我预备退休后要研究的）发表之前，你实在无法估算出正确的期限。你应该很务实地将目标放在最近一次的里程碑上，并且你必须做好心理准备，往后的里程还是充满不确定的因素。

珍惜你的时间，把它善用在找出发生进度落后的原因上，而不是计算我们该延期多久，你不会因为花时间解决问题而耽误更多的时间，相反地，你会使团队以后走得更快更稳，早一点达到目的。

法则 44

After a sliP，hit the nextmilestone，no matter what

延误了这个里程碑，就一定要如期到达下一个里程碑

我们必须明白，每一次的延误，就是你和团队信心的一次受挫，所以，延误这个里程碑时，最好的补救办法就是无论如何绝不延误下一个里程碑。团队必须挽回对自己的信心和对理想的承诺；因此，下一个任务必须准时完成的意义更重大，团队需要重建信心。

你知道进度落后的情形有多严重，也知道是什么原因造成，你知道该如何改正问题，然后你建立一个比较近程的、保守估算的下一个里程碑，这次绝不能再失误，然后重新发布这个消息。如果你的时程表是以下三周内不做任何事，那也很好，喔，实在太过头了，是不是？我的意思是你一定得定下一个能够完成的里程碑，哪怕这个里程碑在内容上没什么了不起，并且绝不失误地达成这个目标，这个动作必须铿锵有力才行。绝不能再次延误，因为

团队需要藉这个机会重建自信。如期达成里程碑，这个目标的成功，会带给团队鼓舞的力量，重新充满活力，相信自己有能力达到对时程的承诺。这种心情是非常重要的，如果团队相信自己能够如期达成里程碑，他们就会尽一切努力去达成，这似乎是人类应有的工作方式。

法则 45

A good slip is a net positive

把延误当作宝贵的学习机会

把进度落后当成一种宝贵的学习经验：你曾经不明白的事情，现在明白了，此时正是分析、了解、思考和消化的最佳时机。这时候学到的心得是最深刻的，也许很难用言语描述。如果你只是用刻板的印象去看待进度落后，把它当作是一件坏事，那这个进度落后的事实不能让你进步，进度落后会成为你深怀恐惧却时常发生的事。

虽然有这么多不确定的因素会造成进度落后，延误几乎是必然发生，甚至已经被视为正常。有很多软件开发工作本质上是具有实验性的，新的平台、新的操作系统、新的程序技术，往往使得每一个新项目都充满不确定性。

延误既然不可避免，为了防止酿成大祸，你必须衡量各种可能造成延误的因素。最理想的情况是，你事先已经知道一个以上的不确定因素，并且让所有的工作人员明白进度本身必须包括这些风险，让大家知道我们如何评估这些不确定因素对进度的影响，如何估算风险所占的时间，这项预估能力是团队的知识与技巧，对未来大有帮助的。你同时必须懂得判断人们是不是在做“对”的事情，进度的落后经常的原因是，人们花太多时间在研究一些比较外围或衍生出来的特色，事实上这些工作对产品的核心信息没有什么帮助。

如果延误是你突然的发现，那就表示沟通的渠道不畅通，你必须加强团队的沟通能力，你必须搬出一大堆详细的信息给团队成员看，让他们对这个问题有具体而真实的体会，延误暴露了团队的弱点，但也提供了毫无保留的检讨机会。你必须确定每个人的每个角色都得到了需要的指点。

延误也是重新评估未来相对于什么目标该有什么资源的好机会，日后对该有的产品功能特色，会更加务实地捡选，对于不利于团队工作效率的特色将尽量不予纳入目标，对于资源与特色相互的作用也能更准确估算。

总之，能够让团队学习的延误，绝对是件好事。

法则 46

See the forest

见树亦见林

如果本项目有六个模块，各有 90%的部分已经完成，那么你已经完成了 54%。每个模块完成了九成，听起来是个挺不错的成绩，但不能当成整个项目完成了百分之九十，它们之间不是相加的关系。你必须“见树亦见林”。如果任何一个模块完成比率是零，那么整个项目的完成率也是零。

用这种数学运算去计算整个项目的完成率是非常正确的，因为每一个组件对产品都是相等的重要性，假定你的团队是平均分配工作，那么任何一位

失败就是团队全体的失败。

必须注意的是，没有一件事情是能够有百分之百把握的，如果是，那很可能发生“骄兵必败”的后果。今天的英雄可能明天会惨遭滑铁卢，我每每对曾经跃升的明星和它坠落的速度震惊不已。你必须设法让组员了解声誉得来如此不易，却是如此脆弱易碎，不堪一击。团队必须时时刻刻惕励自己，不要任意把别人当作笨蛋，不要狂妄自大，不要自我膨胀，不要让自己心中的魔鬼打败自己。

法则 47

The World changes，so shouldyou

世界在变，所以你也得跟着改变

成功的软件开发工作有一项重要的特质，就是能够从每天涌进的新信息中，做出正确的决策。你不必过度拘泥于计划和进度，那是人造的事实，难免有失真的时候。改变是机会之母，如果你对学校课堂中教你的知识不能活用，你会失去许多机会，会很难适应这个迅速变迁的环境，你会把改变或机会当成是障碍，当成计划和进度的干扰，而不是充满潜力的转机。

软件开发也是一个不断改变的有机体，通常一个大得不得了的困难，事实上代表着团队对于开发出好软件的强烈欲望。在你作出直觉反应之前，在你把它当成问题去消灭之前，花点时间挖掘这种心理状态背后的玄机，试着把这股能量导入正确的方向。问题本身可能表达着许多意义，你必须运用你的观察力、想象力、第六感，决定如何将软件开发的过程导引顺畅。绝对记住你是在领导一群人的心，共同进行创造性的工作。如果可预测性愈高，代表创造性愈低，所以，每件事情都得保持弹性。

当然，你不会希望轻率改变，任何改变都不可导致你偏离方向，过度的改变会使人无所适从。弹性不代表随便（randomness），弹性是延展性、适应性，是自然地改变，而随便则是突然地、毫无关连的胡乱改变。如果外界没有变化，随机的改变一下也许会有点灵光乍现的好处，但维持原状则更加保险。只有在改变能够加强整体效率时才能接受它，改变的目的是让团队的工作最佳化，而不是导入更多的复杂性。

我们这样举例说吧，也许有一天，你想增加一个里程碑，没有明显的理由非得这么做不可，但是有确定的迹象显示这是目前的发展方向所趋，这时候增加里程碑可能增加了进度落后的风险，但也可能增加产品推出后的机会。我特别提出这个例子，是因为我的团队最近经常遇到类似的困扰，虽然我们一直以准时推出产品为荣，但是环境的变化使我们愈来愈觉得必须与众不同，必须有所突破，在原定计划中加入一个新的里程碑，做一点特别的东西。每一个项目都像是新生的孩子，每个项目都是独特的，你必须顺应项目独有的特质，配合它定出最适当的里程碑，并且充分支持它，用它的语言去表达它，而不是像切蛋糕般，切成你想要的样子。

虽然你想做些改变，你未必有勇气实行。

伟大的软件必定只有一个中心思想，至于品质能够实现到什么程度，依赖领导者能否带领团队融合无数个小而重要的改变。如果你能在混乱中辨识出对项目最有意义的改变，并且引导团队去适应它，将它融入团队的精神中，将来就会在产品中表现出这项改变，呈现在顾客眼前。

抗拒变化是失败的策略，你必须学习找出无法抗拒或是对产品有利的改变，拥抱它、适应它，最后这项改变会带给你丰富的收获。这是困难的心路历程，你需要刚铁般的意志，因为你得依赖自己独有的远见，你所看见的没有任何人能看见，你会很寂寞，每每在深夜被怀疑缠绕而不得安眠。虽然你实在很想做些改变，你未必有足够的勇气付诸行动，你等于是在挑战着人们的期望，而且你自己的感觉也会因此而被左右，你不敢确信这些改变是好的。你会面临艰难的抉择，在改变与不改变之间饱受折磨。

然而，不论改变有多大的阻力，只要是必须的改变，你就得排除万难接受它，否则你会被它摧毁。

对我而言，即使只是把这些想法写下来，都令我感到惊慌害怕，但这实在是我在软件开发的过程中，确确实实的经验体会啊！

ThreePart

第三篇　推出时期

对于软件开发团队来说，没有什么比产品准备推出（Ship Mode）更令人兴奋的了。大多数人认为产品在经过这个阶段后，就可说是诞生了，但我还是倾向于将此一时期细分为三个小阶段：激活、推移和完成（onset，transition，endgame）。这三个小阶段没有明确的间隔，彼此也有重叠的部分，但三者的工作重点却不相同，而每个团队必定都会一一经历，才能将产品推出。每个人对这三个小阶段的区分方式也可能不尽相同，但对于将整个产品推出的催生阶段，每个人都会有类似的看法或感受。

产品的推出：激活

激活产品推出（ship mode：onset）是一段漫长而渐进的过程，就好像婴儿出生前的阵痛。这时候团队开始为产品的推出做一切繁杂的准备工作，就好像分娩前产妇要练习呼吸，身体的机能也会为出生的那一刻作准备一般。在产品推出的那一刻，团队必须全神贯注、倾全力使它顺利出门，实在是个紧张而痛苦的时刻。

假定我们有四或五个里程碑，大概到 M3 时产品就应该大致成形，M3 的到达就是产品推出的第一次“模拟考”。往后的第四个和第五个里程碑，应该更强调产品推出时的练习，不但是产品本身愈来愈接近推出时该有的模样，里程碑的到达也应该更像是产品的真正推出。

通常进入激活阶段的第一个征兆是，一部分认真思考的组员开始焦虑：“我们真的能做到吗？”这并不表示团队害怕自己做不来，而是一些先知先觉者开始思考许多该做的事、可能降临的困难，想到那么多那么杂的事情要做，觉得千头万绪，简直不知如何是好，尤其是开始焦虑的人还是少数时。逐渐地，恐惧甚至惊慌，像是野火般蔓延到整个团队，也许领导者在公开场合也流露出这种内心的情绪。在整个团队都染上这种紧张的情绪时，领导者反而可以松一口气，因为现在每个人的心里都关心着同一件事，产品的推出是每个人现在强烈的心愿，大家有一种携手同心的感觉，我们正在共同为一个伟大的产品而努力的感觉，经过了那么辛苦的开发终于要有成果的感觉。

有些组员会怀疑，领导者大概是乐观过了头。

现在，领导者要做的事是为大家加油打气，为大家灌输自信和热忱，有些组员可能会觉得领导者大约是乐观过了头，但大部分的人都会受到鼓舞，理解在这时候焦虑是正常的，是可以被接受的自然反应，而且是创造智能财产过程中必经的压力。

在激活阶段的团队工作效率会达到巅峰状态，因为大势已定，不需要再揣摩或协商。这时候的团队心理有以下几项特点：

·大势已定 产品的功能特色、团队中每个人的角色几乎是完全定型了。所有犹豫不决的特色不能早点定案到现在都得删除，组员之间的工作负荷很平均，产品架构完全确定而且趋向成熟，测试计划正在进行，文件撰写原本只有纲要，现在要加入细节。此时一切都没啥好争论的，没有任何疑惑，每个人都知道产品就要诞生，剩下来该做的事也很清楚，去做就是了。艰辛的开发工作终会过去，此时，赶紧把它完工是最重要的。

·共同一致的信念 每个人都有一致的信念，我们即将完成产品，时间一到产品就能推出。现在所有的痛苦抉择或权衡都已经过去，大家只要专注于眼前产品的状态就好。产品的催生时期就代表着所有的争议结束。如果现在

还有怀疑或争论（到现在还没解决的话），也只有暂时搁置，到下一版时再说，否则永远没完没了。此时若是还有人对项目有严重的怀疑，产品是永远无法进入催生时期的。·每个人都明白 每个人都明白自己该做什么事，也明白整个团队该做什么事，才能使产品顺利推出。现在已经没有什么事是不确定的了。所以，只要继续把手上的工作做出来即可。延误或是对延误的恐惧，只单纯地和自己的工作有关，不再和别人的工作有太多的依存。在组员心里曾有的焦虑、飘浮无依的不安，现在都可以放掉。我常把这个阶段称作“工作清单”阶段，每个小组都把自己的一大串的工作逐条列在清单上，现在要把清单上的工作逐项完成，打个勾勾，清单上每一项工作都做完时，产品就完全推出了。

开发团队的最高领导人必须身先士卒，率先进入产品推出时该有的心理状态，并且信守一切的承诺，才可能带动所有组员进入这个紧张的时期。对于细节的认知会愈来愈清楚，在工作步调上也许会有点凌乱的危机，但这时候领导人最重要的工作就是带领大家专注于产品的推出，凡是不太相干的事情尽量排除；在团队中的每一个成员都渴望推出，看到自己辛勤耕耘的结晶是多么欣慰，领导人必须将大家对于推出的渴望，转化成完美的推出工作。

产品的推出：推移

推移阶段（ship mode：transition），在时间上可以跨过一个以上的里程碑，视产品性质的需要与产品的推出而定。基本上，产品推出的推移阶段是产品从开发迈向完成的逐渐稳定阶段。开发工作仍在进行，直到最后一个里程碑为止；如果开发工作的重心是增加功能，势必很难同时追求产品的稳定，自然严重影响产品的推出；由于同时追求增长和稳定实在太耗费心力，这时候主要的开发活动是除虫，凡是无法完成的功能特色都暂时放弃。要放弃任何的功能特色，都难免引起组员的不安情绪，所以，下决定之前必须审慎地分析评估。

如果我们把产品推出当成一条数学上的连续曲线，你会发现当产品愈来愈成熟，所需的开发人力愈来愈少，产品推出就进入了完成阶段（我们稍后再详细讨论）。现在，产品推出的推移阶段即将结束，而将进入早期的完成阶段时，会有下列的迹象：·组员的思考更加保守而小心谨慎，避免掉入开发的泥淖。

·程序置入的动作审慎再审慎，通常会成立一个临时的特勤小组，检查每一个程序置入的动作，务求事先防范大幅的不利影响。

·增进执行速度和改善使用者接口成为工作的重点，把产品磨亮打光，以期带给外界深刻的印象。收尾成为最重要的方向。

·修饰的工作完成后，接下来加强产品的稳定度，以达到推出的要求水准，所有的开发工作已经结束，清除错虫的工作也已接近尾声。

产品的推出：完成

在产品推出（激活、推移和完成）的三个阶段中，工作重心发生移转，从准备进入催生时期的心理变化，产品逐渐稳定下来，直到产品推出的完成阶段（ship mode：endgame）时，只剩下一个里程碑了，就是“推出”（shipping）。在概念上，这个阶段的工作相当单纯，就是一张工作清单，当上面所列的项目逐一完成，产品就推出了。工作清单上有几百项，甚至几千项，也不过只是一张清单罢了，很多但不会太困难，你没有时间想或做这张清单以外的事情，每个人只有心思专注在自己该做的事情上。也许会发生

一些状况，使得工作清单上必须增加一些项目，团队会在一阵“适度抗拒”（appropriate resistance）之后接受工作项目的增加；这种适度的抗拒，我把它称之为“谦逊的抗拒”（profound resistance）。

这时候每天召开例行的检讨会是个很不错的方式，如果能常常邀集决策者参与更好。会中的议程不拘泥于惯例或任何形式，但必须是能立即决断的问题，凡是长期性的目标或一时无法解决的问题，都暂时撇开不谈，下次改版时再说。管理者参加这个会议的作用之一就是将议题引导在这个方向上。现在，团队的主要目的是在推出日期时，产品的品质水准要够好。已经进入倒计时阶段，开发人员主要的工作是错虫的更正，而且是重要的错虫——这会影响很多的使用者，或是造成严重具毁灭性的错虫，都必须优先清除。至于使用者界面的美化、执行速度加快、或是新的功能，都不宜在完成阶段讨论。Beta 测试版的测试结果，以及外界的反馈，都以“追求产品的稳定性与避免严重错误”为大前提，作小幅度的修改。而其他的建议案，都只是预警相关人员在销售上可能遇到的困难或不利影响，或是下一版的改进时的建议，但基本上，目前不打算在产品上响应这些项目。

有多少无伤大雅的错虫，存在于你推出去的产品中？

你必须了解，顾客对于软件品质的要求程度，或是说对错虫的忍受程度。在你推出的产品中，有多少错虫因为优先顺位较低，而未及修正就随着产品出门？这些较次要的错虫会不会造成问题？使用者会不会因此而在操作上遇到麻烦？由于在产品的推出阶段，稳定是第一优先考量，所以在决定是否要更正一个错误以前，必须先考虑会不会因此而造成更大更多的错误；有些错误的修正太冒险，错误本身的危害并不大，把它列在产品的读我（readme.doc）档中说明，会比较好。

有时候，就像有些人的阵痛期特别长一样，产品的推出工作如果不太顺利，也可能使上市的日期受到延误，以下有几个法则提供软件开发团队的领导人参考，以期能避免在产品推出时发生延误的情形。

法则 48

Violate atleast one sacredcow

关怀多于要求

身为团队的领导人，你可能巴不得所有的属下都拼命工作。但事实上，虽然在产品推出阶段所面临的挑战可能真的需要大家拼命工作，你却不可能“命令”大家为工作牺牲奉献自己的一切。我曾经见过一些没有人性也不懂人性的管理者，要求他的组员晚上加班、周末来上班，结果组员要不是他的命令理都不理，就是在背后嘲弄这位可恨的管理者。

事实上在产品推出的阶段，管理者的作用并未消失，有些仪式性的动作，虽然很小或很简单，却是意义非凡。此时领导人应该用实际的行动来表达他对组员的关怀，肯定他们对组织的贡献，让每一位组员都感受到自己的每一个角色都是那么的重要，领导者的关怀，可以增强组员的向心力，对产品的顺利推出有莫大的影响。

领导者对组员的沟通应该是具有鼓励性，不是强迫加班，而是提供方便的加班环境。例如，写一张小卡片给某位组员（不能用虚假的客套话、公式话，要真诚的关心）、走廊上餐厅里不期而遇时的几句赞美、提供加班时的

托婴托儿服务、自掏腰包来个点心宵夜，或是让组员可以很方便地在家工作（简单的远距办公室通信设备）；这些小动作花费极少，却让组员体会你的关心与诚意，而感到自己的加倍付出是值得的。还有，这些小动作似乎都是与传统规矩背道而驰，你为了他们而打破这些传统的规矩，也让组员们觉得你对他们的肯定超过公司给你的规定。一般来说，优秀的开发人员多少都有点叛逆心理，喜欢对传统的权威挑战，你用别出心裁的方式鼓舞他们，比用传统的方式要来得有效。

优秀的开发人员多少都喜欢违抗传统。

以行动表示的关怀，同时还告诉组员产品推出是件值得纪念的大事。大家合拍一张照片，写一篇小故事，收集一些开发过程中辛苦的笑料，送个电子邮件恭贺大家产品终于要推出，或是表扬一下杰出的组员或事迹。这些都是领导人在这时候可以做的事情，让大家留下难忘的回忆。

好好享受这欢笑的收割吧！因为大家都曾经流泪播种。产品推出是大家形成一个团队的目的，也是大家同甘共苦情绪的最高点。这时候团队士气是最高昂的，所有曾经犯下的愚蠢行为和几近干戈的争议，现在一律抛诸脑后。大家已经完成任务，产品推出了。

法则 49

Beta is not the time to changeBeta

测试版不是修改功能的时候

几乎全世界的人都有这种误解,以为 Beta 测试版就是邀集外界对产品提出设计方面的改进建议，然后让软件公司对功能做加强或修改。这是完全错误的。（Alpha 测试又名内部测试，Beta 测试又名外部测试，Beta 测试版是把即将推出的产品提供给部分顾客试用，另一方面也作为对市场的试探。）Beta 测试的目的是确定产品是否能在预计的各种硬件平台与操作系统中正常运作。虽然 Beta 测试的反馈意见很有参考价值，但除非 Beta 测试中发现产品有重大问题，否则不应对功能再做修改，顶多只是更正错误。所有的建议和反馈都留到下一版时再考虑纳入。

这么做绝不表示忽视顾客的意见，相反地，如果你要把 Beta 测试的意见纳入，那你永远没有办法推出产品。你与顾客之间的关系应该是更亲密、更持续的（请参考本书第一篇中的顾客关系）。

法则 50

The Beta is for spin develop-mentBeta

测试是暖身活动

顾客对产品的第一个印象往往决定了他对产品的评价。基本上，Beta 测试者的反应，也会是大部分顾客的反应。行销人员应该把握 Beta 测试的机会，了解测试者对产品的感受，分类并记录测试者的反应，以便在产品的包装、信息传达等方面抓对方向。即使在这时候主要的信息已经发布给媒体，敏锐的行销人员仍然能够从 Beta 测试的结果,得知那些应该强化或淡化的信息。行销人员应该建立一套顾客心理学的模型，用来了解顾客在使用产品时的心理状态及其变化，以及产品信息应该如何根据试用顾客的心理反应来做

调整。

毫无疑问，你一定会在 Beta 测试中得到一些负面的评价，这很正常，而且应该在你的意料之中。如果负面反应很强烈的话，你得设法找出其他的产品优点去平衡它；如果负面反应很普遍，每个测试者都有，那你得让大众降低对产品期望的水准，让这个负面反应在消费者的意料之中，而不致产生吃惊受骗的感觉，这样负面的反应就不会那么强烈；最后，你得设法让这个产品的缺点看起来不那么严重，比方说提供解决的操作方式等等。

就算 Beta 测试活动不是由行销人员所主导，也应该让行销人员密切参与。在这个时候，与外界的沟通就可说是产品成败的关键了。产品信息比产品本身还重要，现在已经不是用开发活动来创造产品的形象。

法则 51

Triage ruthlessly

急救术

如果软件代表开发团队，那老实说，它一定有数不清的缺点。欠缺完美是所有智能财产的必然性质。世上所有的事物都不完美，所以问题不在于判断这个产品好或不好，而是决定应该修改那一部分，使它比较能被顾客接受或喜爱。我们把这个判断并修改的过程称之为“急救术”（triage）。

急救术，当然是个医学名词，在急诊室中医生必须非常迅速地查看所有的问题，采取最急迫必要的急救医疗措施，然后再依优先级分别施救。软件急救术是非常类似的观念，先分析各项错误与瑕疵，以及它的严重性，以下是判断是否施救的准则：

·错误的严重性：如果错误严重到必须回收所有已推出的产品时，那么就必须立即改正这项错误。

·明显程度：错误会很明显而立刻被使用者发现吗？会不会影响到产品的品质？会影响到产品的形象，而成为竞争者攻击或嘲笑的把柄吗？

·影响范围：有多少比例的使用时间，会遇到这个错误？这个错误是大部分的使用者都会遇上的吗？

·修正错误的风险：如果要修正这个错误，会不会造成软件的不稳定？这需要非常资深又对产品了如指

掌的开发人员来判断。

·团队动态：为了修改这项错误是否需要动用大批的人力，抑或是一小部分人员投入即可？会不会使已经忙碌不堪的开发人员更加人仰马翻？

·修正错误后所需要的测试成本：为了任何理由修改任何部分都需要测试，团队有没有足够的人力来执行测试工作？时间上允不允许测试？

一般而言，急救小组的任务是选择和修改产品中最重要的错误，尽量让大部分的使用者在大部分的时间都能使用愉快。对使用者而言，这项不完美的产品是帮助还是灾难？他是用这个不完美的产品比较好，还是宁愿不用比较快乐？这个复杂且严肃的问题，需要工作人员不断设身处地去设想使用者的情况，揣摩使用者的心情，与使用者双向沟通，才能探得正确的答案。

法则 52

Don’t shake the Jell-O

小心保持软件的稳定

处于完成阶段，你必须灌输组员一个观念：修改软件是一件危险的事。软件马上要推出，这时候稳定绝对是第一要务。错误修改的成本或风险如果太高，宁愿不要修改；不必要的修改必须极力避免。

产品的推出，就像一盘特大号的、结构脆弱的果冻，你一摇动盘子，果冻必定会颤动、摇晃，然后逐渐静止。你摇动愈用力，果冻晃得愈厉害，需要变回静止的时间就愈长。同样地，你修改任何一个错误，都不可避免地影响到整个软件的稳定性，等到你的软件像果冻一样恢复静止，恭喜你，那就是你推出产品的时机了！

然后，它会再度震动。

第四篇　发布时期

本书的主要目的虽然在阐述如何以团队方式开发伟大的软件，然而，我们无论如何都不能忽略与大众的沟通。软件如果无法让一般大众很容易感受它的伟大、它独特的优点，又有什么用呢？我的看法是，软件的伟大必须要经过与大众沟通的过程，让世人都认识它，既然是伟大的软件，当然要让它在历史上记下一笔，让它永垂不朽。因此，软件产品完成后，必须要有一个产品发布会。

产品发布会就是让产品留名青史的时刻，顾客、产业记者和评论家都在发布会上睁大眼睛看你的软件作品，这是他们了解产品的开始。即使你的软件不是商业性的，也不打算特别包装，但仍得利用产品发布会来强调它的技术性，至少是在一个研讨会之类有很多重要人物聚集的场所发布你的新技术。无论采用什么方式或强调什么重点，你的目的是昭告世人新软件产品出炉了，希望大家对它有正确而清楚的认识。这是软件首次登台亮相，你得让它吸引众人的目光，尽可能使它的信息清楚地传递给所有的人，这是你的开发团队和顾客建立关系的大好时机，也是对组员很大的鼓励。

产品发布会是一个充满成功气氛的活动。

已发布的信息是无法收回的。信息虽然可以重新发布，但结果大都不理想，人们心中对产品的印象不是新的信息，而是一片混淆，所以第一次信息发布格外重要。一个规划妥善的产品发布会，所有信息与气氛的设计都应该以产品为中心，一切安排都是为了抓住人们的注意力，让产品在明确的信息烘托之下，漂亮登场，配合精心设计的各种沟通媒介，务必让每个人都对产品有鲜明而正确的认识。把产品发布会做成功的好处是说不尽的，此处择要概述如下：

·确定的产品发布会，能够促使团队为确定的目标同心协力。如果每个人都认为并期待某一件事应该在某一天发生，就像是一个共同的目标，无形中促使开发团队把它当作一项有意义的挑战而同心协力。

我建议每一组参与产品发布活动的人，都有详细的战略规划，因为发布会和产品本身，就像一场战役。

产品发布团队（launch team）应该与开发团队一样，应用本书的 54 条法则，应该有共同的目标、懂得运用策略、有团队精神，发布团队必须把任务当成一种全心全意的承诺。全体工作人员会感受到产品发布那种成功的气氛，辛辛苦苦孕育的产品，如今捧在自己手中，那是非常令人振奋的感觉，这种心情会感染给会场上的每一位来宾，使产品发布会更加热烈。

·成功的产品发布会能够提高产品的接受度。有些意见领袖对产品发布会特别敏感。很多人参加产品发布会纯粹是为了对科技的狂热，或是产品对他个人有重大的影响，再加上产品发布会本身多少带有戏剧性，所以，会有很多人带着热情而来，他们会希望认识你、你的团队和你们的努力、你的软件和你对这个软件的感情，因为使用你的软件，他们开始认识你，你和团队的言行气度所表达出来的形象，也会是他们对新产品的印象之一。事先应该和行销人员研究好，如何确切地表达这个产品在历史和技术上的重要性，这和如何发挥产品发布会的戏剧效果是同样重要。·发布会是产品与大众见面的最重要的活动。就像电影的首映会一样，发布会是产品问世时的第一次见面；发布会的前置作业与后续的活动都有逻辑上的顺序，应该事前安排妥当。

在 Beta 测试时，你就应该邀请媒体、重要的顾客、相关领域的权威人士、你的朋友，试用 Beta 测试版，这样的话，他们会对产品有清晰的概念，也会比较乐意提供自己的感想，为产品做见证。而在发布会前夕，你私下寄给这些重要试用者一段小小的摘要，稍微提醒一下产品的特点，免得他们太忙而不记得产品试用时的感受。在产品发布会上，这些人就会很自然、很清楚地向记者宣布他们对产品的看法，等于是为产品背书一样的效果。发布会后接踵而至的是一大堆的广告活动，包括传单、电子邮件、产品目录、现场展示等等。你必须密切注意与会者的现场反应，尽可能给予足够的支持和服务。这一切细节都是发布会的一部分，也是产品信息的一部分，不论这些事情要花你多少功夫，无论你和顾客熟到什么程度，你都必须将这一切细节都照顾好，确定产品发布会尽善尽美。

法则 53

Compete with the superiorstory

伟大的软件应该有一个伟大的故事

每一个伟大的软件，背后都有一个伟大的故事（平庸的产品当然只有平庸的故事）。信息架构就像软件中的位元一样复杂交错，因为信息必须以不同的层次传达给不同的对象，信息必须和接受者发生情感上的共鸣，必须随着时间和环境而演变修正，信息必须够简单，这样才能强而有力，但又必须够复杂，这样才能引发接受者更大的兴趣。很显然地，信息所达到的沟通品质，与产品销售成功与否有莫大的关系。请切记，你应该有适当的投资放在信息传递上，就像开发工作一样重要，因为信息和产品是密不可分，二者都是团队所创造出来的智慧财产。

传统智慧经过许多年的传颂，容易让人很快接受。

你必须事先仔细思量，以产品的沟通策略来设计的信息结构应该是什么样子？如何传递出去？应该如何引导人们的思路，帮助他们开始懂得欣赏产品？你的信息应该包括这些，才能让无形的智能财产被世人看见。

当你站在听众面前，别只顾着示范产品特色，先表达一下产品所赋予的内涵，把产品的故事说给团队、高级主管、记者、产业分析师、顾客或投资人听，这个故事必须具有传统智能的色彩。凡是带着传统智能色彩的东西，都会被自然地传颂，而且在任何专业领域都是共通的语言。

传统智能必定是个小故事大道理。你的故事必须至少有一条主轴是与科技趋势或市场情况（或是二者）有关，这样才能让听众觉得你的故事有价值，乐意倾听。故事的内容必须简明扼要，同时又能引发听众更深入的思考，让头脑灵活的听众觉得其中有些内涵正是我所想过的（共鸣！），太玄的言论无法引起在座专家的兴趣，而且绝不能落于俗套。请注意故事要既简单又有内涵，不要太过简化而流于空洞，你的内涵一定要让听众觉得非常有价值才行。

要让听众深有同感：“对！这正是我过去一直在思考的。”

要把产品信息用一个简明、有见地的故事表达出来，本来就是一个很大的挑战。你必须能够打动听众的心弦，要让他们深有同感：“对！这正是我过去一直在思考的。”这不是在宣传你的观念，而是在无形中与听众的思想交流，引发他去想你提示的问题，让听众用自己的心去体会你的信息，而不

是被你灌输信息。这些信息必须听起来像是常识，被你说出来，又在他心中产生了共鸣，就会让听众觉得这是真理。

让我们举个例子来说明信息的内涵，你可以这样开场：“使用者不爱用footbar，因为它就是这么难用。”这好像是直说国王没穿衣服一样的反传统——但绝不激进。用一个大家都知道或愿意相信的事实（也许大家都不好意思宣之于口），从这个事实紧密地引申出另一个与产品有关的事实（你要表达的重点，也许直接说会不容易被人接受），而且记住不要陈述已经被别人提过的事实（人们会混淆这话到底是谁讲的）。

当我们发布 VisualC++1.0 的时候,我们强调的事实是大部分的程序设计师还没有从 C 换到 C++。我们调查结果的数据资料显示，有 80%的 C 语言程序设计师打算、即将或是希望能从 C 换到 C++，但只有 10%的人真正能做到。媒体和产业分析报告都表示 C++的时代已经来临，但我们的资料却显示着相反的情形，这是我们要直言说破的皇帝新衣。

直接说出产品信息，未必是最有效的表达方式。

我们的直言不讳很快打动了人们的心坎，因为这是千真万确的事实，而且大部分的人都有同感。就算我们不说，也许大家终究会知道事实的真相，只不过要花多一点时间和疑惑吧。

然后，第二步是紧密地引申出另一个事实，也就是产品信息，非常简单：即使是对于精通 C 语言的程序设计师，C++还是太难了——直到 VisualC++的出现为止；我们不追究为什么 C++那么难，我们只知道 VisualC++把它变容易了。

传统智慧通常都以俗谚表达，用一两句简单又响亮的词句，让人朗朗上口、易颂难忘。政治家都是精于发明口号的高手，例如：“我们需要的是五分钱的雪茄”、“除了畏惧本身，没有什么好畏惧的”、“不要问国家为你做了什么，要问自己为国家做了什么”。这些口号之所以深入人心，因为它们包含了传统智慧，简明响亮，又提供了指导的功能，没有冗长复杂的理论。尽管喊出这些口号的人早已故去，但这些口号如今依然存在，而且已经变成了智慧的明灯，因为世人迷惑时它们提供指引，如果这些口号没有智慧的内涵，流行过后必定会消失无形，被人淡忘。

好的故事会让人终生难忘，因为它有内涵、有见地，会让听的人感受到其中的睿智。好故事必须让人们容易联想，一旦它触动了听众内心的共鸣（因为是很简单的事实），人们会很自然地打开心扉准备接受下一个联想或引申出来的事实，而且接受程度和你打动他们的程度成正比。例如，“某教练的球队总是无法赢得重要比赛”这是第一个事实，但人们听到之后在自己的内心会这样重述：“这教练不好”或是“要不是他输掉了关键球赛，可能还是个不错的教练”这是一般人必然的反应，就是把接收到的信息加上自己的观感，用自己的话重述一遍。

你诱导听众自己想出来的事情，肯定比由你来告诉他更有说服力。

所以，如果你能将产品故事用个有情节的方式来述说，那是再好不过了：有背景、有气氛、有情节起伏、有好人也有坏人、有刺激、有舒缓、有场景变化等等。找一位故事专家协助你把产品故事说得生动活泼，又能打动听众内心，并且教会团队所有的人都能说这个故事。

一个好的故事必定以显而易见的事实为依据。但是绝对不要太多的事实，一般人常犯的毛病是有太多的信息急于让听众知道，结果流于杂乱，缺

乏重点，每位听众的内心印象都不相同，这是不好的沟通。关键事实不要超过三项，给听众他记得住的事实，又简单又重要，而且要确定你的关键事实与你的产品故事密切不可分离。

故事必须让很多重要人物重复地说，让听众的印象极深刻，甚至让听众自发性地跟自己的朋友说。所以，故事必须以几个重要的事实为根据，流露着感性。以人性为着眼点，这样的故事必定让人一听难忘。

有趣的是，最好的方法是别提重要的事，你要做的是让听众自己从内心导引出你要的正确结论。比方说，我们发布 VisualC++时，最重要的事是让媒体、产业分析家和大众了解：微软要重回 C++的市场，带着复仇的决心要夺回盟主地位，我们决心扭转乾坤。但是我们在任何行销信息中都没有提到这件事，甚至从产品故事中也完全看不出丝毫端倪。这个道理很简单，我们如果到处向人炫耀我们的产品有多伟大，既无法使人信服，反而惹人生厌。相反地，我们让这个信息从听众自己内心推想出来，在产品故事中提供足够的线索让人们导出这个结论。你诱导听众自己想出来的事情，肯定比由你来告诉他更有说服力。

法则 54

Create a winning image

建立赢家形象

除了做出好产品，巩固市场地位，基于以下几个理由，你还得建立赢家形象：

·顾客需要对软件的认同感。软件这东西就像汽车之类的商品一样，顾客需要对自己买的商品有认同感。大概是因为顾客花在与软件相处的时间实在很长的关系，他们希望自己被视为是“某软件的专家”或“某软件的玩家”，从这些名词可以看出来，顾客并不是“买”软件，而是“采用某软件”(adopt)，这一点值得所有的软件从业人员深思再三。

·顾客希望自己拥有的东西是大多数人渴望或羡慕的。软件象征了顾客的身份地位或是他的“工作伙伴”（with-it-ness）。在这个计算机科技日新月异的时代，顾客用的东西象征了他个人先进的程度，最新的东西（不论是软件或硬件）表示是最流行、最尖端的，追求最新最好是一种强烈的动机。如果你自认不受流行的影响，请想一想你在挑选领带、西装，或购买车子时，你内心浮现的欲望，你就知道流行实在是无法抗拒的（请参考法则 14）。

·顾客知道厂商支持的重要性，不希望自己成为孤儿使用者。顾客不是单纯地买软件，而是与你建立长长久久的关系，他们知道决定使用你的软件之后，会有很长的时间依赖你在新版本中的进步，他们要投入大量的时间金钱去学习使用你的软件，他们不希望买下软件之后厂商倒店，自己成为科技孤儿，用也不是不用也不是。

对外界的沟通应该不是只有产品信息，还包括塑造人人称羡的赢家形象。嘘……别忘了我们的法则 53，别用直接说明的方式，胜利者不会四处宣扬自己是胜利者，更不会泄露自己对胜利的渴望，他就是每次都胜利！

结束语

如果一定要用一句话来做总结，我会说：软件开发造就开发人员（Development develops developers）。

也许罗曼·波兰斯基（Roman Polanski）在电影“唐人街”（Chinatown）中的情节，正好为我表达了软件开发的结论：剧中的主角杰克（由杰克·尼克逊饰演），经历着相当大的内心挣扎，几乎崩溃，他的同伴总是用戏谑的语气揶揄他：“撑着点吧，这是唐人街。”

当然，软件开发是没有什么结束语的，你必须从本书的开头再经历过一遍又一遍永难超生的轮回，一版又一版…

亲爱的朋友，撑着点吧，这就是软件开发！

附录善用人才

对于智能财产的创造而言，最重要的当然是人类聪明的脑袋，所以项目经理无不希望雇用最聪明的人，好好照顾他们，让他们做出最聪明的软件，并且组成所向无敌的优异团队，不断成长、创新，永远能给顾客最新最好的东西。

这些话听起来像是老生常谈，要做到可就是一门大学问了。

雇用聪明的人

要去寻找并网罗聪明的人并不容易，但这方面的努力是值得的，因为聪明的人才能生存。在软件的职业生涯中，一个人必须承受许多负面的经验，像是没眼光的老板，错误的设计决策，浪费时间去做没有用处的软件，分崩离析的团队，进度严重落后的项目。这些对于任何一位软件工程师而言都是很大也很痛苦的挑战，但这却是软件业的正常现象；“聪明”（being smart）是他最有利的条件，靠着聪明，他可以无视这些困难，甚至能解决团体危机，带给他有满足有成就的软件开发生涯。当然，智能财产的创造原料就是聪明才智，所以你必须寻找最聪明的人，网罗他加入你的团队。

在你面试一位聪明的人才时，请注意以下几种人格特质：

一、留心言语无法表现的各种智能象征

所谓“大智若愚”，智能的展现方式会因人而异，有时候，真正的聪明看起来却不怎么样，你必须留心每个人的独特性，绝不要被“聪明”的刻板印象所束缚，注意观察他自然流露的聪明，虽然或许不是你预料的那种形态。

仔细观察应征者的各方面反应，他如何表现自己的个人特质？这一点透露出什么信息？从他的面部表情揣度他内心的情感变化，把速度放慢点，分析他，判断一下他有多少深度和潜力。练习观察人，你会发现很多隐性的信息，比显性的表征更能让你正确判读眼前的陌生人。

用你自己的感情和智力反应来衡量他，这位应征者是否能挑战比你想得更多更深，探求更微妙的奥义，或是更快地评估出正确的情势，他的要求是否超过你所能够提供的？整个面谈的感觉如何？他是否有值得你学习的地方。

面谈的时间并不长，彼此的角色是固定而有点僵化的。应征者的反应是否如你预期？他是否有些言语行为令你感到意外？他是否很自然地主动打破求职者的传统角色和面谈的僵硬气氛？如果不然，你是否能打破彼此僵固的界线，使他卸下求职者和面试者之间习惯使然的刻板形象？他看来是否够弹性够开通，不会无谓地固执己见？有没有适当、发自内心的求知欲，或只是因为害怕被淘汰而追求新知？如果应征者够聪明，那他大概会使面谈过程发生一些变化，如果他有能力影响面谈过程，他也会有能力影响开发团队，或是更复杂的事情。

二、用难题考验应征者

你给他一个真实世界的问题，提供许多有关该问题的背景信息，请他发表看法。当然应征者知道自己正在被接受考试，所以你可能会得到准备过的标准答案，然而，重要的不是答案，而是看应征者推理的过程，可以从中发掘他的工作与思考问题的态度（working style）。

应征者是否很快地对问题作出错误的假设呢？几乎每个人，在一开始的时候都是如此。要不然，应征者可能会对考题要求更多的信息，从中界定问题的架构，然后试着导出一个看起来面面俱到的答案。有些时候，我会故意给一个绝对无法找出答案，但我好像确定有答案的问题，看看应征者如何反应。聪明的人不会掉进自我辩白的陷阱，因为无论他们的答案是什么都一定是错的，而会反过来问我，究竟该怎么办。当然，大部分的人都太固执、太有防卫心，而且就是不懂求助正是最好的办法。

试试看如果由你主动提出帮助，应征者会如何反应，有意思的是，会有无数种大异其趣的结果，我比较欣赏的是先问问接受帮助是不是弱者的表现，再决定要不要接受帮助的人。

三、试着教导他们

谈谈你的专业领域，看看应征者的反应，他们是不是能很快地吸收你所讲的东西？会不会询问更深入的问题？接受它或是抗拒它？应征者是否能够提出更好的看法？

请记住你的目标是雇用你将来要赋予重任的人，将来要对他充分授权，他要替你管理许多技术，甚至决定你的软件的一部分未来。应征者如何与权威形象（也就是担任主试者的你现在的形象）建立关系，是他未来表现良窳的重要关键。你要的人，必须能把主观的自我心态放在一边，而专心去解决问题或学习新知。

四、排除审查人才的盲点

我常见到在甄选人才时最大的错误，就是管理者过度重视某一项技术。从征人广告上就可以看出这种倾向：“ 征求资深软件工程师，必须具备 X 程序语言技术及 Y 年以上的经验 ”，诚然，拥有技术是考量的重点，但绝非一切，更重要的是技术之外的东西；即使我们只看技术，这项技术可能在未来一年之内就会淘汰，我们对技术的观察重点是此人如何在相关的其他技术中，积累他今天对这个技术的精熟。我要看他的生涯中，技术学习和积累的过程：他在什么时候用过什么技术？他的学习历程是否足以证明他能立刻掌握现在我们要用的技术？适才适任

如果你已经雇用了一位聪明的人才，你一定希望他的聪明才智能够充分发挥，为团队带来长期的益处。如果只有你慧眼独具，知道他的潜力，而其他的团队成员却不太能看出他的天赋，那么，他所能发挥的将比较有限。团队成员的工作性格可以分为两大类：勇猛躁进者（racehorses）和眼高手低者（overreachers）。同一个人在不同的群组或不同的时间，可能会有截然不同的工作性格。身为管理者的你，必须知道你的属下是那一种工作性格，他们的个人生涯是怎么回事，你才能给予适当的辅导，让他们适才适任，充分发挥。

勇猛躁进者喜欢快节奏

勇猛躁进者是那种凡是学得很快、成长很快、表现成功的人，所以显然他需要更宽广的驰骋空间，才能维持他喜爱的快速。管理者的任务是肯定他的成功，提供他更多的表现机会，让他担任的角色范围更宽广，导引他把目

标放在自我的超越上。如果管理者有眼不识千里马，或是打压他的速度，勇猛躁进者就会欺负实力较弱的人，或是到处寻觅，试着自行发掘更宽广的挥洒天地。在健全的团队中是不应该有这种情形发生的。

就像我那住在爱荷华的父亲常说的名言：“槽里的食物若是不够，连猪都会互相残杀。”

人呢，不是成长就是凋零，不可能原地不动。如果你的团队推出新产品的周期愈来愈频繁，你的市场占有率愈来愈高，顾客的满意度也逐年提升，或是任何其他有利的结果，都表示你的团队正在成长，否则就是退步。

对于团队来说，成长所带来的挑战是如何能配合团队的成长而适当地补充人才，同时避免人数增加所带来的成长障碍。通常团队人数快速增加的第一个问题是组员开始烦躁不安，成功的喜悦很快被遗忘，人们开始谈论过去的好日子。这种现象的发生，是因为管理者没有给组员适当的挑战。

为了改变这种烦躁不安，得来点刺激的，找出几个关键人物想要做点什么、什么东西会对他们具有挑战性，问问他们，这东西与我们的软件事业有什么重要关连？他们希望有什么样的环境？他们梦想创造的软件是什么模样？然后，你授权给他们去做适当的东西。

成长，代表新的投资，你可能要做全新的东西，让组员扮演截然不同于以往的角色，不要迟疑，每个人都需要成长，从最高领导人到最基层的人，成长都是不可避免的，包括你和我。所以，你必须为组员的成长找寻最适当的出路。

创造力的省思

创造力总是受限于防卫心理，而防卫心理如果不过度的话，其实是健康的。在一个健康的团队中，无论创造力怎么被大家高度重视，总是有些改变会受到抗拒。即使是再小的改变，也得以追求进步为目标，强化或深化原有的思想或行事方法，才可能行得通。因此，改变必须架构在原有的基础或是人们已经接受的事实上，才能顺利推行。可惜的是，即使改变只有两步，第二步是建构在第一步之上，而不是某种事实，都可能被最健康的团队所抗拒。

真正具有创造性的改变，是要在一个具有自我超越能力的环境中方能产生。这种环境，不只是接受一般的改变，把改变视为正常，还要能自发地培养改变的机会、欢迎改变的来临，而且借着改变把自己推向一种全新的动态境界。自我超越的团队会把革命性的剧烈改变当成巅覆过去的思维，以全新面目再出发的契机。健康的团队不等于富有创造力的团队，团队可能很健康但创造力略嫌不足，这并不是最理想的情况；理想的团队应该充满创造的活力，随时能有新的想法，要很有勇气，敢于摸索从来没有人涉足的领域，又有能力取得丰硕的成果。

讽刺的是，愈健康的团队，愈以过去的优秀工作成绩为傲，愈无法接受革命性的改变。这是任何领域中，整体性表现最好的团队所共有的通病，值得我们省思。

眼高手低者企图大有作为

有些时候，有些组员所希望分配到的工作，事实上超过他的能力。这种“贪多”的心理可能起源于他个人早期的工作经验，也许那时候他对工作的渴望没有得到适当的满足，结果这种过去未满足的心理需求转移到现在的工作中，造成病态性的好高骛远，眼高手低：即使现在已经是个完全能充分发挥、各展所长的环境，他还是需要更多的授权、做更大范围的事、企图扮演

更多的角色，永远不能满足内心的匮乏感，因为他潜意识地极力避免在团队中没事干。这种病态的“贪多”，就是想要的超过实际所需要的，而且似乎永远难以满足。由于团队中会自然地设法避免不必要的浪费，其他的成员不会让这种眼高手低者过度扩张。

即使是很优秀的组员，在刚开始学习成熟时也会有这种情形：他可能分配到一个小范围的工作，但并不明白工作范围小些其实并不妨碍自己的表现机会，于是他可能抱怨管理者判断错误，把我大才小用，埋没我的天份。如果他懂得细心照顾自己的生涯，他就会有更正面的态度去面对这件事，藉此机会多学习。很可惜，太多优秀人才在这一关跨不过去，要不就是骤然挂冠求去，要不就是管理者真的给他太多无法消化的工作，最后的结果都是毁了一个人在组织里求发展的机会。管理者应该协助属下安然渡过这一关，这对他个人以及整体团队都是很有利的。

发掘真正的目标

你必须用心发掘眼高手低者内心真正的目标。眼高手低者有一个明显的特征，就是对已经过大的目标优柔寡断，而且常常是互相对立的两个目标（因为他鱼与熊掌都想兼得）；这一点很重要，这是你以下的工作的基础。

大部分的时候，眼高手低者渴望有一种自我形象，他想要利用工作内容或价值来标榜出也许超过字面上的意义。你必须试着发掘出来，他真正想要的或是企图表现的是什么样的自我形象。方法很简单，问他几个不错的问题就行了：这个目标对你而言有什么意义？你觉得这个意义适当吗？你认为自己有多大的把握能够完成目标？如果你顺利完成了目标，你希望你的成果可以如何运用？然后，检视你自己的感觉：这个目标在我看来适当吗？别人的看法大概是会怎样？谁是最佳人选？如果选择由某人来担任，是为某人，还是为团队，还是为我自己？

探求他对软件开发生涯的看法

现在你已经了解他的目标了，你现在要决定这个目标是否超过他个人的能力，或是与团队的目标不合。没有问题当然最好，但如果二者有差距，你得找出真正的原因。为什么他对自己的能力有过高的自信？这是偶然的现象或是此人向来的习惯？你要知道，组员对于别人认定他“能力不够、做不来”这类的事情是极度敏感的，你必须让他相信你是站在他这一边，帮助他达成远大志向的人，而不是阻碍他的人。

你必须了解他个人对软件开发生涯的看法，藉助这些来探索他企图有某种作为的真正理由。也许连他自己都不知道二者之间的关连，但他对软件开发生涯的看法，绝对会影响他今天的企图，你得用敏锐的洞察力去发掘真相。方法一样简单，用一些不带价值判断、轻松又简单的问题直接问他就行。等你相当了解他的看法，了解他如何从自己对软件开发生涯的期望，导出今天的（事实上过度的）志向。你来想想看，这种心理机转模式有没有违背团队的目标？有没有与团队的价值观冲突？他个人的志向是否与团队的终极目标不谋而合？

你的工作是导引他将个人的志向融入团队目标，辅导他懂得配合自己的能力定出合宜的目标。

找最适合的工作给他

现在，你已经完全了解他想要什么，以及他为什么想要这些；然后暂时把这些信息放在一边，想想看这个人最适合担任什么角色，找出三件事是他

做得比任何人都棒的，如果让他做这些工作，他可以成为最有成就的人，请团队的其他成员告诉你他们对此人的期望，如果他本人同意自己确实对这三件事最为擅长，设法把他的志向扭转成大家的期望。

至少在理论上，加强一个人的长处，比纠正一个人的缺点是要容易得多。但是身为领导人的你，必须有勇气丢掉一切对工作角色分配的假设，这绝不像听起来这么容易，很多管理者都失败过很多次。我认为只有最崇高最卓越的领导者才能做到这一点：调整组织，让每一个人的天赋才干都能淋漓尽致地展现。

新天地

在我带过的团队中，就有这么一位天资聪颖的女性。她曾经在别的公司有五年的工作经验，当过极成功的项目经理，由于我们对她过去的经历没有直接的了解，所以并不太知道她的管理风格和能力。她表示想当更高的主管，也就是说，她想以“管理者”为终生职志。

慢慢地，我们看到她不可思议的能力。她能够解决任何人际问题的疑难杂症，不论任何冲突或障碍，她都能建议最好的方法，也就是说，她有分析问题、平息争议的天才，愈来愈多的人找她帮忙提供指导和建议，或请她剖析复杂的情势。这些都是自然发生的，我并没有使力促成。

我仔细观察并思考此一现象，并且与她本人讨论，我愈来愈觉得，一个健康的团队随时需要像她这样的人，立即而有效地针砭团队中的疑难杂症。看起来，有个人专职减少摩擦、平息纷争、提升效能是个好主意，对团队的益处实在很大。如果让她当管理主任（supervisor）似乎并不适合，她不必靠监督别人来展现她的才华，她只要大家对她的支持就够了。

虽然我们并没有谈出确定的细节，我和她已有共识，决定成立一个新的头衔及工作内容来配合她的专长，头衔尚未决定，工作内容则是“效能分析及工作顾问”。她同时会参加功能特色小组的横向组织，她的主要工作是解决任何工作效能的问题；她在“效能分析及工作顾问”的小组中担任领袖，负责建立起这个创新的工作领域，发展适当的工作规则，训练她的助手，为公司培养像她一样的人才，为这个全新的角色奠定日后发展的基础。

为优秀的人才创造适当的职位，可以为组织带来莫大的利益，这是一种“创意管理”。

当你发现一位组员的特殊天份，鼓励他充分发挥时，大部分的人都会表现出有点迟疑的态度。至少我个人和我的几个亲密的工作伙伴多少也会有这种情形。我想可能是因为天赋是个人珍视的东西，自己内心容许引以为傲，但对外则不免戒备恐惧。我们觉得自己因为拥有这些天赋而与众不同，但我们却害怕把这些自我评价拿出来公开检验，因为，这有点风险——万一失败了，那我不是顶没面子，对自己的价值感可能受到打击。

我也需要鼓励

不久以前，我的主管丹尼斯·吉伯特（DenisGilbert）帮助我克服了这种心理障碍。过去很多年来，我常常以软件开发为主题，在世界各地发表自由即兴式（freewheeling）的演讲，评价很不错，我自己也很喜欢，不只是因为这代表对我个人的肯定，也是我对社会上所有栽培过我的人的一种反馈（演说的内容就是本书的前身）。

丹尼斯知道我的演说颇为成功，很多软件开发人员都因此获益匪浅，但是，那是在微软之外呀，我不敢在自己的公司里演讲，要我在自己人面前讲

出种种糗事和内心挣扎，实在太丢人，而且微软内部高手云集我是再清楚不过，万一讲不好，还引起争议怎么办？我相信让同事分享我的经验是对大家的帮助，但我很害怕自己被公开评估，那怕这是我最得意的一项才华，我就是不想冒着被拒绝的危险。

后来丹尼斯不断鼓励我，给我信心，我的演说真的很精彩，而且我的经验座谈会帮助很多人（当然包括微软内部的同事）克服工作上的困难，这是功德无量的（丹尼斯如今大概后悔了，放我去写这本书，而置 VisualC++的工作暂时不顾）。

畏惧和自我设限往往会阻碍一个人的发展，这是个人内在的障碍，而不是来自外界的压力。在充分授权的环境中，虽然这不是管理者造成的障碍，但是管理者应该协助属下突破自我设限的障碍。

人们对自己的天赋有极高的敏感性，如果有主管赏识他的天赋，给他机会，鼓励他表现，他会做得比想象中更好、更卖力。你必须时时鼓励他，给他挑战，支持他的冒险行动，在他表现不错时，适时赞美他。刚开始时，他会低估自己的潜力，主管必须告诉他，他能做到的，他是这方面的天才，主管深信他一定可以表现得令人赞赏，主管可以用各式各样的法子设法让他明白，没有什么事情比他所拥有的天赋更重要，其他的困难都是可以被克服的。

有很多实例可以证明，主管对属下的赏识和肯定，绝对是一本万利的。人们通常在受到主管的慧眼赏识时，都会感到惊喜，会更愿意付出，说是“士为知己者死”当然是太夸张，但他一定能发挥出更大的潜力；他原本就是在做自己擅长的事情，现在表现更好，并且对公司更忠诚，以回报主管的知遇之恩。

对一位专业人员来说，有人能够了解他、重用他，有人赏识他的才干，就是最宝贵的报酬。

一位用心的主管会懂得找出与属下的内心世界沟通的好方法，这是一项困难的挑战，但一定是值得的，只是回报方式可能很难想象。用心的主管会花时间了解属下，也许半小时的谈话却花掉两个小时的事前准备，用心思考你要告诉他什么信息，注意信息是否能确实而清晰地传达，不要使他误会、或是引得他的胡思乱想。

不要企图一次就告诉他太多，但每一次的信息都要很清楚，运用你的沟通才能，这是主管的重要任务，清楚的沟通是双方都需要的。

你可以对自己的信息“再确定”，这个动作能够让属下知道，你不是随便说说，你是真的肯定他的天赋，现在你是真的授权给他了。

不耻下问

当你遇到问题，尽量向周围的人讨教。虽然你自己也很不错，但一定有别人在某方面强过你，能够支持你。所以，不要做骄傲的笨蛋，开口求援才是正确的法门。你不是单兵作战，在你身边有一大群优秀的人才可以帮助你。不必害怕请教别人会使你暴露弱点，相反地，让别人知道你也有弱点，反而使你更受欢迎，属下更愿意对你忠诚。

设定短期目标

给你的组员一个短期目标来展现他的能力，证明他的进步，告诉他你会给他最好的机会和支持，而这是互相的，你也希望他表现优异，如果他表示宁可不要这个互惠协议，那就暂时不谈了；你可以告诉他这是个很不错的机会，在全部的人中，你相信他最适合担任这项工作，而你十分乐意提供属下

最好的，但前提是属下得有意愿表现才行。现在由他来选择，如果他不喜欢，你当然只好把机会让给别的组员。

建立长期计划

分派新任务给一个组员之后，他一定希望对这个任务有比较长远的概念，或是自己的长期发展方向等，他也会想知道自己应该如何掌握什么原则。也许他不好意思问你，但这是你身为管理者的重要工作，别偷懒！

建立评核制度

你如何得知个别组员在团队中的表现孰优孰劣？团队似乎要不就全体成功，要不就全体失败，你如何断定单一个体对团队的贡献（或破坏）程度？

软件开发工作就像现在电子游戏（video game）一般，有很多关（level），难度愈来愈高。每一个团队成员都在同一关，希望能有成就而获得晋级，每一关都有不同的挑战，一开始时大家都会迷惑，觉得好困难，然后开始奋战，甚至几乎发狂。好不容易到了下一关，游戏规则又不一样了，先前的发展出来的合作方式在这一关未必有效，搞不好还有害。一关又一关，团队终于学会如何面对新环境，而且发展出真正有效的合作模式。

评核员工的团队表现，最大的要点是先决定他们到底在第几关，容易的还是困难的（团队的人数、每个人应有的表现水准、这一组人的任务性质），然后再来决定个别成员的表现是优于预期，或是不如预期。这是评核员工绩效的基本原则。

软件开发团队也像电子游戏一般，看你有多少“生命点数”或“火力”，愈好的先天条件当然愈容易打赢，有时候输掉实在是资源太少的非战之罪，主管在给属下打分时也别忘了这一点。

评核，是你重新检讨对属下的期望的时候，看看你有没有忽略谁的天赋、有没有人需要调整脚步、短期的目标是否相对太高或太低。直接而坦白的沟通是最好的方式，虽然也是最难的方式，组员也需要有此机会和你一起检视一下，自己所处的挑战是否需要调整，这一关要不要重打，或是跳到那一关；晋级速度太快并不是好事，也许自己在这一关中还需要复习，或是前一关有东西还没学全。

在每一关中的成熟过程每个人都大不相同，有人是天才早熟型，有人是大器晚成型。每个人都会自己走出自己的路，走过独特的成熟历程，终于绽放出属于自己的色彩光芒。基本上，每一关都是团队的合作，大家互相扶持，才能跃高到另一个层次，然后看得更宽、更广、更远，个人的影响力更大，角色更复杂。最后，培养出卓越的领导能力（可不是监督能力），或是更高的工作效能，而终于成为公司最宝贵的资产。资料来源

很难精确地引述本书的想法是来自何处。基本上，本书的内容主要是我工作多年的心得，我读过的书给我一些特别的启发，很多人给我的建议（我在前言中提过了，再次感谢他们）。为了让读者方便检索更多想法的出处，容我在此简单列出我能想起的一些人物或书籍，希望这对读者有些帮助。

每个受过教育的文明人都听过弗洛伊德（Freud）吧。读一些弗氏的著作，或是他的生平、传记、电影，或是任何一种了解他的方式，你会发现一些心理学的奥秘。如果你有时间，做一点心理学的分析试验，成本虽然有点高，但你可能（抱歉我不保证）对心理学有更深刻的认识，有更新的视界。

达尔文（Darwin），无疑是最伟大的自然探索者，他的理论几乎在任何领域都用得上。花点时间研读他的著作，尤其是“自私的基因”（The Selfish

Gene）一书，你得到的启示会值回千百倍的票价，你会对这个竞争社会的演化本质有更透彻的了解。

欣赏一下莎士比亚（Shakespeare）是个不错的主意，不必花太多时间读完全集，你能抓住美学的要领是最重要的，你可以从中一窥领导、激励的奥秘，就这一点而言，莎士比亚的著作绝对是最好的典范。就像“仲夏夜之梦”，它与软件开发在本质上并没有不同。

林肯总统（Lincoln）、格兰特将军（General Grant）和丘吉尔（Churchill）一直是特别吸引我的伟人。格兰特将军全心投入，不到战争结束绝不放弃战斗的性格，正是最佳的软件开发团队领导人。丘吉尔也是性格坚强的人物，全世界都反对时，他仍然坚持理想，他善于演说，能够激励盟军破败的士气。林肯总统也是一位热情洋溢的演说家，能够激励最自私的人，能够领导最难驾驭的人才。

软件是人类智能的现代式精致产物，所以，你需要多培养观察力。欣赏毕加索（Picasso）的画作，他那突破所有传统的表现方式，超出所有现实的窠臼，也许可以给你一点特别的灵感。其他任何美学的东西，时尚的流行、建筑、艺术、设计，这些都有时代文化的背景，诉说着人们价值观和思潮的脉动。因此，只要你有时间，都不必吝惜学习欣赏美。

关于美学，我在第一篇中大略提过。我建议你读鲁道夫·尔汉（Rudolf Arnheim）的著作，特别是“艺术与视觉的感知”（Art and Visual Perception），你会找到设计使用者接口时所需要的一切理论，它会帮助你设计出极佳的使用者接口。

有一本蛮有趣的书可以充实你的基本历史观念,那是由威尔和爱瑞尔·杜蓝（Will & Arial Durant）合著的《文明的故事》（The Story of Civilization）。如果你想创造出伟大的软件，历史是你的必修课之一，你需要了解人类文明的历程，否则绝对和伟大的软件无缘。

电影《芭比的飨宴》（Babette' s Feast）是领导人的入门功课，从这里你开始了解人们的认知，领导人大部分的时间应该是花在使人们用更多的方式经历更丰富的事情，这部电影不错。

至于艺术家的使命和任务，我建议你观赏《艾迪伍得》（Ed Wood）这部电影。

如果你不想追寻更多的灵感来源，可以继续埋头做软件，但你没有新的源泉注入是别想有杰作出现的。