人月神话(二十周年纪念版)

作者：弗雷德里克·布鲁克斯

关于作者

Frederick P. Brooks，Jr.是北卡罗来纳大学Kenan-Flagler商学院的计算机科学教授，北卡来罗来纳大学位于美国北卡来罗来纳州的查布尔希尔。Brooks被认为是"IBM 360系统之父"，他担任了360系统的项目经理，以及360操作系统项目设计阶段的经理。凭借在上述项目的杰出贡献，他、Bob Evans和Erich Bloch在1985年荣获了美国国家技术奖（National Medal of Techology）。早期，Brooks曾担任IBM Stretch和Harvest计算机的体系结构师。

在查布尔希尔，Brooks博士创立了计算机科学系，并在1964至1984年期间担任主席。

他曾任职于美国国家科技局和国防科学技术委员会。Brooks目前的教学和研究方向是计算机体系结构、分子模型绘图和虚拟环境。

1975年版献辞

致两位特别丰富了我IBM岁月的人：

Thomas J. Watson, Jr.，

他对人们的关怀在他的公司依然无所不在和Bob O. Evans，

他大胆的领导使工作成为了探险。

1995年版献辞

致Nancy，

上帝赐给我的礼物。

二十周年纪念版序言（Preface to the 20thAnniversary Edition）

令我惊奇和高兴的是，《人月神话》在20年后仍然继续流行，印数超过了250,000。人们经常问，我在1975年提出的观点和建议，哪些是我仍然坚持的，哪些是已经改变观点的，是怎样改变的？尽管我在一些讲座上也分析过这个问题，我还是一直想把它写成文章。

Peter Gordon现在是Addison-Wesley的出版伙伴，他从1980年开始和我共事。他非常耐心，对我帮助很大。他建议我们准备一个纪念版本。我们决定不对原版本做任何修订，只是原封不动地重印（除了一些细小的修正），并用更新的思想来扩充它。

第16章重印了一篇在1986年IFIPS会议上的论文《没有银弹：软件工程的根本和次要问题》。这篇文章来自我在国防科学委员会主持军用软件方面研究时的经验。我当时的研究合作者，也是我的执行秘书，Robert L. Patrick帮助我回想和感受那些做过的软件大项目。1987年，IEEE的《计算机》杂志重印了这篇论文，使它传播得更广了。

《没有银弹》被证明是富有煽动性的，它预言十年内没有任何编程技巧能够给软件的生产率带来数量级上的提高。十年只剩下一年了，我的预言看来安全了。《没有银弹》激起了越来越多文字上的剧烈争论，比《人月神话》还要多。因此在第17章，我对一些公开的批评作了说明，并更新了在1986年提出的观点。

在准备《人月神话》的回顾和更新时，一直进行的软件工程研究和经验已经批评、证实和否定了少数书中断言的观点，也影响了我。剥去辅助的争论和数据后，把那些观点粗略地分类，对我来说很有帮助。我在第18章列出这些观点的概要，希望这些单调的陈述能够引来争论和证据，然后得到证实、否定、更新或精炼。

第19章是一篇更新的短文。读者应该注意的是，新观点并不象原来的书一样来自我的亲身经历。我在大学里工作，不是在工业界，做的是小规模的项目，不是大项目。自1986年以来，我就只是教授软件工程，不再做这方面的研究。我现在的研究领域是虚拟环境及其应用。

在这次回顾的准备过程中，我找了一些正工作在软件工程领域的朋友，征求他们的当前观点。他们很乐意和我共享他们的想法，并仔细地对草稿提出了意见，这些都使我重新受到启发。感谢Barry Boehm、Ken Brooks、Dick Case、James Coggins、Tom Demarco、JimMcCarthy、David Parnas、Earl Wheeler和Edward Yourdon。感谢Fay Eard出色地对新的章节进行了技术加工。

感谢我在国防科学委员会军事软件工作组的同事Gordon Bell、Bruce Buchanan、RickHayes-Roth，特别是David Parnas，感谢他们的洞察力和生动的想法。感谢Rebekah Bierly对16章的论文进行了技术加工。我把软件问题分成"根本的"和"次要的"，这是受NancyGreenwood Brooks的启发，她在一篇Suzuki小提琴教育的论文中应用了这样的分析方法。

在1975年版本的序言中，Addison-Wesleys出版社按规定不允许我向它的一些扮演了关键角色的员工致谢。有两个人的贡献必须被特别提到：执行编辑Norman Stanton和美术指导Herbert Boes。Boes设计了优雅的风格，他在评注时特别提到："页边的空白要宽，字体和版面要有想像力"。更重要的是，他提出了至关重要的建议：为每一章的开头配一幅图片（当时我只有"焦油坑"和"兰斯大教堂"的图片）。寻找这些图片使我多花了一年的时间，但我永远感激这个忠告。

Soli Deo gloria－愿神独得荣耀。

查珀尔希尔，北卡罗来纳 F.P.B., Jr.

1995年3月

第一版序言（Preface to the First Edition）

在很多方面，管理一个大型的计算机编程项目和其它行业的大型工程很相似--比大多数程序员所认为的还要相似；在很多另外的方面，它又有差别--比大多数职业经理所认为的差别还要大。

这个领域的知识在累积。现在AFIPS（美国信息处理学会联合会）已经有了一些讨论和会议，也出版了一些书籍和论文，但是还没有成型的方法来系统地进行阐述。提供这样一本主要反映个人观点的小书看来是合适的。

虽然我原来从事计算机科学的编程方面的工作，但是在1956-1963年间自动控制程序和高级语言编译器开发出来的时候，我主要参加的是硬件构架方面的工作。在1964年，我成为操作系统OS/360的经理，发现前些年的进展使编程世界改变了很多。

管理OS/360的开发是很有帮助的经历，虽然是失败的。那个团队，包括我的继任经理F. M. Trapnell，有很多值得自豪的东西。那个系统包括了很多优秀的设计和实施，成功地应用在很多领域，特别是设备无关的输入输出和外部库管理，被很多技术革新广泛复制。它现在是十分可靠的，相当有效，和非常通用的。

但是，并不是所有的努力都是成功的。所有OS/360的用户很快就能发现它应该做得更好。设计和实现上的缺陷在控制程序中特别普遍，相比之下，语言编译器就好得多。大多数这些缺陷发生在1964-1965年的设计阶段，所以这肯定是我的责任。此外，这个产品发布推迟了，需要的内存比计划中的要多，成本也是估计的好几倍，而且第一次发布时并不能很好地运行，直到发布了几次以后。

就象当初接受OS/360的任务时协商好的，在1965年离开IBM后，我来到查珀尔希尔。

我开始分析OS/360的经验，看能不能从中学到什么管理和技术上的教训。特别地，我要说明System/360硬件开发和OS/360软件开发中的管理经验是非常不同的。对Tom Watson关于为什么编程难以管理的探索性问题，这本书是一份迟来的答案。

在这次探索中，我和1964-65年的经理助理R.P.Case，还有1965-68年的经理F.M.Trapnell，进行了长谈，从中受益良多。我对比了其他大型编程项目的经理的结论，包括M.I.T.的F.J.Corbato，Bell电话实验室的V.Vyssotsky，International ComputersLimited的Charles Portman，苏联科学院西伯利亚分部计算实验室的A.P.Ershov，和IBM的A.M.Pietrasanta。

我自己的结论体现在下面的文字中，送给职业程序员、职业经理、特别是程序员的职业经理。

虽然写出来的是分离的章节，还是有一个中心的论点，特别包含在第2-7章。简言之，我相信由于人员的分工，大型编程项目碰到的管理问题和小项目区别很大；我相信关键需要是维持产品自身的概念完整性。这些章节探讨了其中的困难和解决的方法。后续的章节探讨软件工程管理的其他方面。

这个领域的文献并不多，但散布很广。因此我尝试给出参考资料，说明某个特定知识点和指引感兴趣的读者去看其他有用的工作。很多朋友读过了本书的手稿，其中一些朋友给出了很有帮助的意见。这些意见很有价值，但为了不打乱文字的通顺，我把它们作为注解包含在书中。

因为这本书是随笔不是课本，所有的参考文献和注解都被放到书的末尾，建议读者在读第一遍时略去不看。

深切感谢Sara Elizabeth Moore小姐，David Wagner先生，和Rebecca Burris夫人，他们帮助我准备了手稿。感谢Joseph C.Sloane教授在图解方面的建议。

查珀尔希尔，北卡罗来纳 F.P.B., Jr1974年10月

焦油坑（The Tar Pit）

岸上的船儿，如同海上的灯塔，无法移动。

- 荷兰谚语

Een schip op het strand is een baken in zee.

史前史中，没有别的场景比巨兽在焦油坑中垂死挣扎的场面更令人震撼。上帝见证着恐龙、猛犸象、剑齿虎在焦油中挣扎。它们挣扎得越是猛烈，焦油纠缠得越紧，没有任何猛兽足够强壮或具有足够的技巧，能够挣脱束缚，它们最后都沉到了坑底。

过去几十年的大型系统开发就犹如这样一个焦油坑，很多大型和强壮的动物在其中剧烈地挣扎。他们中大多数开发出了可运行的系统--不过，其中只有非常少数的项目满足了目标、时间进度和预算的要求。各种团队，大型的和小型的，庞杂的和精干的，一个接一个淹没在了焦油坑中。表面上看起来好像没有任何一个单独的问题会导致困难，每个都能被解决，但是当它们相互纠缠和累积在一起的时候，团队的行动就会变得越来越慢。对问题的麻烦程度，每个人似乎都会感到惊讶，并且很难看清问题的本质。不过，如果我们想解决问题，就必须试图先去理解它。

因此，首先让我们来认识一下软件开发这个职业，以及充满在这个职业中的乐趣和苦恼吧。

编程系统产品

报纸上经常会出现这样的新闻，讲述两个程序员如何在经改造的简陋车库中，编出了超过大型团队工作量的重要程序。接着，每个编程人员准备相信这样的神话，因为他知道自己能以超过产业化团队的1000代码行/年的生产率来开发任何程序。

为什么不是所有的产业化队伍都会被这种专注的二人组合所替代？我们必须看一下产出的是什么。

图1.1：编程系统产品的演进

在图1.1的左上部分是程序（Program）。它本身是完整的，可以由作者在所开发的系统平台上运行。它通常是车库中产出的产品，以及作为单个程序员生产率的评估标准。

有两种途径可以使程序转变成更有用的，但是成本更高的东西，它们表现为图中的边界。

水平边界以下，程序变成编程产品（Programming Product）。这是可以被任何人运行、测试、修复和扩展的程序。它可以运行在多种操作系统平台上，供多套数据使用。要成为通用的编程产品，程序必须按照普遍认可的风格来编写，特别是输入的范围和形式必须扩展，以适用于所有可以合理使用的基本算法。接着，对程序进行彻底测试，确保它的稳定性和可靠性，使其值得信赖。这就意味着必须准备、运行和记录详尽的测试用例库，用来检查输入的边界和范围。此外，要将程序提升为程序产品，还需要有完备的文档，每个人都可以加以使用、修复和扩展。经验数据表明，相同功能的编程产品的成本，至少是已经过测试的程序的三倍。

回到图中，垂直边界的右边，程序变成编程系统（Programming System）中的一个构件单元。它是在功能上能相互协作的程序集合，具有规范的格式，可以进行交互，并可以用来组装和搭建整个系统。要成为系统构件，程序必须按照一定的要求编制，使输入和输出在语法和语义上与精确定义的接口一致。同时程序还要符合预先定义的资源限制--内存空间、输入输出设备、计算机时间。最后，程序必须同其它系统构件单元一道，以任何能想象到的组合进行测试。由于测试用例会随着组合不断增加，所以测试的范围非常广。因为一些意想不到的交互会产生许多不易察觉的bug，测试工作将会非常耗时，因此相同功能的编程系统构件的成本至少是独立程序的三倍。如果系统有大量的组成单元，成本还会更高。

图1.1的右下部分代表编程系统产品（Programming Systems Product）。和以上的所有的情况都不同的是，

它的成本高达九倍。然而，只有它才是真正有用的产品，是大多数系统开发的目标。

职业的乐趣

编程为什么有趣？作为回报，它的从业者期望得到什么样的快乐？

首先是一种创建事物的纯粹快乐。如同小孩在玩泥巴时感到愉快一样，成年人喜欢创建事物，特别是自己进行设计。我想这种快乐是上帝创造世界的折射，一种呈现在每片独特、崭新的树叶和雪花上的喜悦1。

其次，快乐来自于开发对其他人有用的东西。内心深处，我们期望其他人使用我们的劳动成果，并能对他们有所帮助。从这个方面，这同小孩用粘土为"爸爸办公室"捏制铅笔盒没有本质的区别。

第三是整个过程体现出魔术般的力量--将相互啮合的零部件组装在一起，看到它们精妙地运行，得到预先所希望的结果。比起弹珠游戏或点唱机所具有的迷人魅力，程序化的计算机毫不逊色。

第四是学习的乐趣，来自于这项工作的非重复特性。人们所面临的问题，在某个或其它方面总有些不同。因而解决问题的人可以从中学习新的事物：有时是实践上的，有时是理论上的，或者兼而有之。

最后，乐趣还来自于工作在如此易于驾驭的介质上。程序员，就像诗人一样，几乎仅仅工作在单纯的思考中。程序员凭空地运用自己的想象，来建造自己的"城堡"。很少有这样的介质--创造的方式如此得灵活，如此得易于精炼和重建，如此得容易实现概念上的设想。（不过我们将会看到，容易驾驭的特性也有它自己的问题）然而程序毕竟同诗歌不同，它是实实在在的东西；可以移动和运行，能独立产生可见的输出；能打印结果，绘制图形，发出声音，移动支架。神话和传说中的魔术在我们的时代已变成了现实。在键盘上键入正确的咒语，屏幕会活动、变幻，显示出前所未有的或是已经存在的事物。

编程非常有趣，在于它不仅满足了我们内心深处进行创造的渴望，而且还愉悦了每个人内在的情感。

职业的苦恼

然而这个过程并不全都是喜悦。我们只有事先了解一些编程固有的烦恼，这样，当它们真的出现时，才能更加坦然地面对。

首先，必须追求完美。因为计算机也是以这样的方式来变戏法：如果咒语中的一个字符、一个停顿，没有与正确的形式一致，魔术就不会出现。（现实中，很少的人类活动要求完美，所以人类对它本来就不习惯。）实际上，我认为学习编程的最困难部分，是将做事的方式往追求完美的方向调整。

其次，是由他人来设定目标，供给资源，提供信息。编程人员很少能控制工作环境和工作目标。用管理的术语来说，个人的权威和他所承担的责任是不相配的。不过，似乎在所有的领域中，对要完成的工作，很少能提供与责任相一致的正式权威。而现实情况中，实际（相对于正式）的权威来自于每次任务的完成。

对于系统编程人员而言，对其他人的依赖是一件非常痛苦的事情。他依靠其他人的程序，而往往这些程序设计得并不合理，实现拙劣，发布不完整（没有源代码或测试用例），或者文档记录得很糟。所以，系统编程人员不得不花费时间去研究和修改，而它们在理想情况下本应该是可靠完整的。

下一个烦恼--概念性设计是有趣的，但寻找琐碎的bug却只是一项重复性的活动。

伴随着创造性活动的，往往是枯燥沉闷的时间和艰苦的劳动。程序编制工作也不例外。

另外，人们发现调试和查错往往是线性收敛的，或者更糟糕的是，具有二次方的复杂度。结果，测试一拖再拖，寻找最后一个错误比第一个错误将花费更多的时间。

最后一个苦恼，有时也是一种无奈--当投入了大量辛苦的劳动，产品在即将完成或者终于完成的时候，

却已显得陈旧过时。可能是同事和竞争对手已在追逐新的、更好的构思；也许替代方案不仅仅是在构思，而且已经在安排了。

现实情况比上面所说的通常要好一些。当产品开发完成时，更优秀的新产品通常还不能投入使用，而仅仅是为大家谈论而已。另外，它同样需要数月的开发时间。事实上，只有实际需要时，才会用到最新的设想，因为所实现的系统已经能满足要求，体现了回报。

诚然，产品开发所基于的技术在不断地进步。一旦设计被冻结，在概念上就已经开始陈旧了。不过，实际产品需要一步一步按阶段实现。实现落后与否的判断应根据其它已有的系统，而不是未实现的概念。因此，我们所面临的挑战和任务是在现有的时间和有效的资源范围内，寻找解决实际问题的切实可行方案。

这，就是编程。一个许多人痛苦挣扎的焦油坑以及一种乐趣和苦恼共存的创造性活动。

对于许多人而言，其中的乐趣远大于苦恼。而本书的剩余部分将试图搭建一些桥梁，为通过这样的焦油坑提供一些指导。

人月神话（The Mythical Man-Month）

美酒的酿造需要年头，美食的烹调需要时间；片刻等待，更多美味，更多享受。

- 新奥尔良Antoine餐厅的菜单

Good cooking takes time. If you are made to wait, it is to serve you better, and to please you.

- MENU OF RESTAURANT ANTOINE, NEW ORLEANS

在众多软件项目中，缺乏合理的时间进度是造成项目滞后的最主要原因，它比其他所有因素加起来的影响还大。导致这种普遍性灾难的原因是什么呢？

首先，我们对估算技术缺乏有效的研究，更加严肃地说，它反映了一种悄无声息，但并不真实的假设--一切都将运作良好。

第二，我们采用的估算技术隐含地假设人和月可以互换，错误地将进度与工作量相互混淆。

第三，由于对自己的估算缺乏信心，软件经理通常不会有耐心持续地进行估算这项工作。

第四，对进度缺少跟踪和监督。其他工程领域中，经过验证的跟踪技术和常规监督程序，在软件工程中常常被认为是无谓的举动。

第五，当意识到进度的偏移时，下意识（以及传统）的反应是增加人力。这就像使用汽油灭火一样，只会使事情更糟。越来越大的火势需要更多的汽油，从而进入了一场注定会导致灾难的循环。

进度监督是另一篇论文的主题，而本文中我们将对问题的其他方面进行更详细的讨论。

乐观主义

所有的编程人员都是乐观主义者。可能是这种现代魔术特别吸引那些相信美满结局的人；也可能是成百上千琐碎的挫折赶走了大多数人，只剩下了那些习惯上只关注结果的人；还可能仅仅因为计算机还很年轻，程序员更加年轻，而年轻人总是些乐观主义者--无论是什么样的程序，结果是勿庸置疑的："这次它肯定会运行。"或者"我刚刚找出了最后一个错误。"

所以系统编程的进度安排背后的第一个假设是：一切都将运作良好，每一项任务仅花费它所"应该"花费的时间。

对这种弥漫在编程人员中的乐观主义，理应受到慎重的分析。Dorothy Sayers在她的"The Mind of the Maker"一书中，将创造性活动分为三个阶段：构思、实现和交流。书籍、计算机、或者程序的出现，首先是作为一个构思或模型出现在作者的脑海中，它与时间和空间无关。接着，借助钢笔、墨水和纸，或者电线、硅片和铁氧体，在现实的时间和空间中实现它们。然后，当某人阅读书本、使用计算机和运行程序的时候，他与作者的思想相互沟通，从而创作过程得以结束。

以上Sayers的阐述不仅仅可以描绘人类的创造性活动，而且类似于"基督的教义"，能指导我们的日常工作。对于创造者，只有在实现的过程中，才能发现我们构思的不完整性和不一致性。因此，对于理论家而言，书写、试验以及"工作实现"是非常基本和必要的。

在许多创造性活动中，往往很难掌握活动实施的介质，例如木头切割、油漆、电器安装等。这些介质的物理约束限制了思路的表达，它们同样对实现造成了许多预料之外的困难。

由于物理介质和思路中隐含的不完善性，实际实现起来需要花费大量的时间和汗水。

对遇到的大部分实现上的困难，我们总是倾向于去责怪那些物理介质，因为它们不顺应"我们"设定的思路。其实，这只不过是我们的骄傲使判断带上了主观主义色彩。

然而，计算机编程基于十分容易掌握的介质，编程人员通过非常纯粹的思维活动--

概念以及灵活的表现形式来开发程序。正由于介质的易于驾驭，我们期待在实现过程中不会碰到困难，因此造成了乐观主义的弥漫。而我们的构思是有缺陷的，因此总会有bug。也就是说，我们的乐观主义并不应该是理所应当的。

在单个的任务中，"一切都将运转正常"的假设在时间进度上具有可实现性。因为所遇

的延迟是一个概率分布曲线，"不会延迟"仅具有有限的概率，所以现实情况可能会像计划安排的那样顺利。然而大型的编程工作，或多或少包含了很多任务，某些任务间还具有前后的次序，从而一切正常的概率变得非常小，甚至接近于无。

人月

第二个谬误的思考方式是在估计和进度安排中使用的工作量单位：人月。成本的确随开发产品的人数和时间的不同，有着很大的变化，进度却不是如此。因此我认为用人月作为衡量一项工作的规模是一个危险和带有欺骗性的神话。它暗示着人员数量和时间是可以相互替换的。

人数和时间的互换仅仅适用于以下情况：某个任务可以分解给参与人员，并且他们之间不需要相互的交流（图2.1）。这在割小麦或收获棉花的工作中是可行的；而在系统编程中近乎不可能。

图2.1：人员和时间之间的关系--完全可以分解的任务

当任务由于次序上的限制不能分解时，人手的添加对进度没有帮助（图2.2）。无论多少个母亲，孕育一个生命都需要十个月。由于调试、测试的次序特性，许多软件都具有这种特征，

图2.2：人员和时间之间的关系--无法分解的任务

对于可以分解，但子任务之间需要相互沟通和交流的任务，必须在计划工作中考虑沟通的工作量。因此，相同人月的前提下，采用增加人手来减少时间得到的最好情况，也比未调整前要差一些（图2.3）。

图2.3：人员和时间之间的关系--需要沟通的可分解任务

沟通所增加的负担由两个部分组成，培训和相互的交流。每个成员需要进行技术、项目目标以及总体策略上的培训。这种培训不能分解，因此这部分增加的工作量随人员的数量呈线性变化1。

相互之间交流的情况更糟一些。如果任务的每个部分必须分别和其他部分单独协作，则工作量按照n(n-1)/2递增。一对一交流的情况下，三个人的工作量是两个人的三倍，四个人则是两个人的六倍。而对于需要在三四个人之间召开会议、进行协商、一同解决的问题，情况会更加恶劣。所增加的用于沟通的工作量可能会完全抵消对原有任务分解所产生的作用，此时我们会被带到图2.4的境地。

图2.4：人员和时间之间的关系--关系错综复杂的任务

因为软件开发本质上是一项系统工作--错综复杂关系下的一种实践--沟通、交流的工作量非常大，它很快会消耗任务分解所节省下来的个人时间。从而，添加更多的人手，实际上是延长了，而不是缩短了时间进度。

系统测试

在时间进度中，顺序限制所造成的影响，没有哪个部分比单元调试和系统测试所受到的牵涉更彻底。而

且，要求的时间依赖于所遇到的错误、缺陷数量以及捕捉它们的程度。理论上，缺陷的数量应该为零。但是，由于我们的乐观主义，通常实际出现的缺陷数量比预料的要多得多。因此，系统测试进度的安排常常是编程中最不合理的部分。

对于软件任务的进度安排，以下是我使用了很多年的经验法则：

1/3计划

1/6编码

1/4构件测试和早期系统测试

1/4系统测试，所有的构件已完成

在许多重要的方面，它与传统的进度安排方法不同：

1. 分配给计划的时间比寻常的多。即便如此，仍不足以产生详细和稳定的计划规格说明，也不足以容纳对全新技术的研究和摸索。

2. 对所完成代码的调试和测试，投入近一半的时间，比平常的安排多很多。

3. 容易估计的部分，即编码，仅仅分配了六分之一的时间。

通过对传统项目进度安排的研究，我发现很少项目允许为测试分配一半的时间，但大多数项目的测试实际上是花费了进度中一半的时间。它们中的许多项目，在系统测试之前还能保持进度。或者说，除了系统测试，进度基本能保证2。

特别需要指出的是，不为系统测试安排足够的时间简直就是一场灾难。因为延迟发生在项目快完成的时候。直到项目的发布日期，才有人发现进度上的问题。因此，坏消息没有任何预兆，很晚才出现在客户和项目经理面前。

另外，此时此刻的延迟具有不寻常的、严重的财务和心理上的反应。在此之前，项目已经配置了充足的人员，每天的人力成本也已经达到了最大的限度。更重要的是，当软件用来支持其他的商业活动（计算机硬件到货，新设备、服务上线等等）时，这些活动延误出现即将发布前，那么将付出相当高的商业代价。

实际上，上述的二次成本远远高于其他开销。因此，在早期进度策划时，允许充分的系统测试时间是非常重要的。

空泛的估算

观察一下编程人员，你可能会发现，同厨师一样，某项任务的计划进度，可能受限于顾客要求的紧迫程度，但紧迫程度无法控制实际的完成情况。就像约好在两分钟内完成一个煎蛋，看上去可能进行得非常好。但当它无法在两分钟内完成时，顾客只能选择等待或者生吃煎蛋。软件顾客的情况类似。

厨师还有其他的选择：他可以把火开大，不过结果常常是无法"挽救"的煎蛋--一面已经焦了，而另一面还是生的。

现在，我并不认为软件经理内在的勇气和坚持不如厨师，或者不如其他工程经理。但为了满足顾客期望的日期而造成的不合理进度安排，在软件领域中却比其他的任何工程领域要普遍得多。而且，非阶段化方法的采用，少得可怜的数据支持，加上完全借助软件经理的直觉，这样的方式很难生产出健壮可靠和规避风险的估计。

显然我们需要两种解决方案。开发并推行生产率图表、缺陷率、估算规则等等，而整个组织最终会从这些数据的共享上获益。