**人月神话(二十周年纪念版)**

**弗雷德里克·布鲁克斯**

**项目经理如何避免画蛇添足（second-system effect）**

项目经理如何避免画蛇添足（second-system effect）？**他必须坚持至少拥有两个系统以上开发经验结构师的决定。**同时，保持对特殊诱惑的警觉，他可以不断提出正确的问题，确保原则上的概念和目标在详细设计中得到完整的体现。

那么，既然他们具备了所有的这些条件，为什么项目还会失败呢？他们还缺乏些什么？**两个方面--交流，以及交流的结果--组织。他们无法相互交谈，从而无法合作。当合作无法进行时，工作陷入了停顿。通过史书的字里行间，我们推测交流的缺乏导致了争辩、沮丧和群体猜忌。很快，部落开始分裂--大家选择了孤立，而不是互相争吵。**

**里程碑的选择只有一个原则，那就是，里程碑必须是具体的、特定的、可度量的事件，能够进行清晰定义。**

**第1章**

**1.1. 编程行业具有一些内在固有的苦恼**

1. 将做事方式调整到追求完美，是学习编程的最困难部分

2. 由其他人来设定目标，并且必须依靠自己无法控制的事物（特别是程序）；权威不等同于责任

3. 实际情况看起来要比这一点好一些：真正的权威来自于每次任务的完成

4. 任何创造性活动都伴随着枯燥艰苦的劳动，编程也不例外。特别是追溯BUG的时候。

5. 人们通常期望项目在接近结束时，（bug、工作时间）能收敛得快一些，然而软件项目的情况却是越接近完成，收敛得越慢

6. 产品在即将完成时总面临着陈旧过时的威胁

**第2章 人月神话**

**2.1 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影响还大。**

1. 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。

2. 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**。

3. 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。

4. 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。

5. 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。

**2.2. 所有的编程人员都是乐观主义者："一切都将运作良好"。**

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。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。

**2.3. 在若干人员中分解任务会引发额外的沟通工作量--培训和相互沟通。**

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。

**2.4. 关于进度安排，我的经验是为1/3计划、1/6编码、1/4构件测试以及1/4系统测试。**

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。

**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。

**2.5. 当进度落后的时候，削减任务，而不是增加人手**

**削减任务，而不是增加人手。**在现实情况中，一旦开发团队观察到进度的偏差，总是倾向于对任务进行削减。当项目延期所导致的后续成本非常高时，这常常是唯一可行的方法。项目经理的相应措施是仔细、正式地调整项目，重新安排进度；或者是默默地注视着任务项由于轻率的设计和不完整的测试而被剪除。**Brook法则：向进度落后的项目中增加人手，只会使进度更加落后。**（Adding manpower to a late software project makes it later）

向软件项目中增派人手从三个方面增加了项目必要的总体工作量：任务重新分配本身和所造成的工作中断；培训新人员；额外的相互沟通。

**2.6. 避免小的偏差以保证符合进度**

1. 假设任务必须按时完成。

2. 重新安排进度。我喜欢P.Fagg，一个具有丰富经验的硬件工程师的忠告："避免小的偏差（Take no small slips）"。也就是说，在新的进度安排中分配充分的时间，以确保工作能仔细、彻底地完成，从而无需重新确定时间进度表。

**2.7. 第二个系统的危险性**

**一种普遍倾向是过分地设计第二个系统，向系统添加很多修饰功能和想法，它们曾在第一个系统中被小心谨慎地推迟了。**

在开发第一个系统时，结构师倾向于精炼和简洁。他知道自己对正在进行的任务不够了解，所以他会谨慎仔细地工作。在设计第一个项目时，他会面对不断产生的装饰和润色功能。这些功能都被搁置在一边，作为"下一个"项目的内容。第一个项目迟早会结束，而此时的结构师，对这类系统充满了十足的信心，熟练掌握了相应的知识，并且时刻准备开发第二个系统。

第二个系统是设计师们所设计的最危险的系统。而当他着手第三个或第四个系统时，先前的经验会相互验证，得到此类系统通用特性的判断，而且系统之间的差异会帮助他识别出经验中不够通用的部分。

**第3章 外科手术队伍**

**3.1. 最好的和最差的表现在生产率上平均为10:1。**

同样有两年经验而且在受到同样的培训的情况下，优秀的专业程序员的工作效率是较差程序员的10倍。（Sackman、Erikson和Grand）

在该小组中，最好的和最差的表现在生产率上平均为10:1；在运行速度和空间上具有5:1的惊人差异！简言之，$20,000/年的程序员的生产率可能是$10,000/年程序员的10倍。数据显示经验和实际的表现没有相互联系（我怀疑这种现象是否普遍成立。）

得出的结论很简单：**如果一个200人的项目中，有25个最能干和最有开发经验的项目经理，那么开除剩下的175名程序员，让项目经理来编程开发。**

3.2. 小型、精干队伍是最好的--尽可能的少。

3.3 两个人的团队，其中一个项目经理，常常是最佳的人员使用方法。[留意一下上帝对婚姻的设计。]

**3.4. 大型系统的开发，最好采用外科手术队伍的团队框架进行**

一位首席程序员、类似于外科手术队伍的团队架构提供了一种方法--既能获得由少数头脑产生的产品完整性，又能得到多位协助人员的总体生产率，还彻底地减少了沟通的工作量。

Harlan Mills的提议提供了一个崭新的、创造性的解决方案。**Mills建议大型项目的每一个部分由一个团队解决，但是该队伍以类似外科手术的方式组建，而并非一拥而上。也就是说，同每个成员截取问题某个部分的做法相反，由一个人来进行问题的分解，其他人给予他所需要的支持，以提高效率和生产力。**

1. 要特别注意传统的两人队伍与外科医生--副手队伍架构之间的区别。首先，传统的团队将工作进行划分，每人负责一部分工作的设计和实现。在外科手术团队中，外科医生和副手都了解所有的设计和全部的代码。这节省了空间分配、磁盘访问等的劳动量，同时也确保了工作概念上的完整性。

2. 第二，在传统的队伍中大家是平等的，出现观点的差异时，不可避免地需要讨论和进行相互的妥协和让步。由于工作和资源的分解，不同的意见会造成策略和接口上的不一致，例如谁的空间会被用作缓冲区，然而最终它们必须整合在一起。而在外科手术团队中，不存在利益的差别，观点的不一致由外科医生单方面来统一。这两种团队组建上的差异--对问题不进行分解和上下级的关系--使外科手术队伍可以达到客观的一致性。

3. 另外，团队中剩余人员职能的专业化分工是高效的关键，**它使成员之间采用非常简单的交流模式成为可能**。

**第4章 贵族专制、民主政治和系统设计**

4.1 "**概念完整性是系统设计中最重要的考虑因素**"。

4.2 "功能与理解上的复杂程度的比值才是系统设计的最终测试标准"，而不仅仅是丰富的功能。[该比值是对易用性的一种测量，由简单和复杂应用共同验证。]

4.3 为了获得概念完整性，**设计必须由一个人或者具有共识的小型团队来完成。**

4.4 "对于非常大型的项目，**将设计方法、体系结构方面的工作与具体实现相分离是获得概念完整性的强有力方法。**"[同样适用于小型项目。]

4.5 **"如果要得到系统概念上的完整性，那么必须控制这些概念。**这实际上是一种无需任何歉意的贵族专制统治。"

4.6 **纪律、规则对行业是有益的。外部的体系结构规定实际上是增强，而不是限制实现小组的创造性。**

4.7 概念上统一的系统能更快地开发和测试。

4.8 体系结构（architecture）、设计实现（implementation）、物理实现（realization）的许多工作可以并发进行。[软件和硬件设计同样可以并行。]

**第5章 画蛇添足**

5.1 **尽早交流和持续沟通能使结构师有较好的成本意识，以及使开发人员获得对设计的信心，并且不会混淆各自的责任分工。**

5.2 结构师如何成功地影响实现：

.. 牢记是开发人员承担创造性的实现责任；**结构师只能提出建议**。

.. 时刻准备着为所指定的说明建议一种实现的方法，准备接受任何其他可行的方法。

.. 对上述的建议保持低调和平静。

.. 准备对所建议的改进放弃坚持。

.. 听取开发人员在体系结构上改进的建议。

5.3 **第二个系统是人们所设计的最危险的系统，通常的倾向是过分地进行设计。**

5.4 OS/360是典型的画蛇添足（second-system effect）的例子。

项目经理如何避免画蛇添足（second-system effect）？**他必须坚持至少拥有两个系统以上开发经验结构师的决定。**同时，保持对特殊诱惑的警觉，他可以不断提出正确的问题，确保原则上的概念和目标在详细设计中得到完整的体现。

那么，既然他们具备了所有的这些条件，为什么项目还会失败呢？他们还缺乏些什么？**两个方面--交流，以及交流的结果--组织。他们无法相互交谈，从而无法合作。当合作无法进行时，工作陷入了停顿。通过史书的字里行间，我们推测交流的缺乏导致了争辩、沮丧和群体猜忌。很快，部落开始分裂--大家选择了孤立，而不是互相争吵。**

**第6章 贯彻执行**

6.1 即使是大型的设计团队，**设计结果也必须由一个或两个人来完成**，以确保这些决定是一致的。

6.2 必须明确定义体系结构中与先前定义不同的地方，重新定义的详细程度应该与原先的说明一致。

6.3 出于精确性的考虑，我们需要形式化的设计定义，同样，我们需要记叙性定义来加深理解。

6.4 必须采用形式化定义和记叙性定义中的一种作为标准，另一种作为辅助措施；它们都可以作为表达的标准。

6.5 允许体系结构师对实现人员的询问做出电话应答解释是非常重要的，并且必须进行日志记录和整理发布。[电子邮件是一种可选的介质。]

6.6 "**项目经理最好的朋友就是他每天要面对的敌人--独立的产品测试机构/小组**。"

**第7章 为什么巴比伦塔会失败？**

7.1 **巴比伦塔项目的失败是因为缺乏交流，以及交流的结果--组织。 交流**

7.2 "因为左手不知道右手在做什么，从而进度灾难、功能的不合理和系统缺陷纷纷出现。"由于对其他人的各种假设，团队成员之间的理解开始出现偏差。

7.3 团队应该以尽可能多的方式进行相互之间的交流：非正式、常规项目会议，会上进行简要的技术陈述、共享的正式项目工作手册。[以及电子邮件。]

**项目工作手册：使用Wiki进行项目工作手册的管理**

7.4 **“项目工作手册"**不是独立的一篇文档，它是对项目必须产生的一系列文档进行组织的一种结构。"

7.5 **"项目所有的文档都必须是该（工作手册）结构的一部分。"**

7.6 需要尽早和仔细地设计工作手册结构。

7.8 "每一个团队成员应该了解所有的材料（工作手册）。"[我想说的是，每个团队成员应该能够看到所有材料，网页即可满足要求。]

7.9 实时更新是至关重要的。

7.10 工作手册的使用者应该将注意力集中在上次阅读后的**变更**，以及关于这些**变更重要性的评述**。

**组织架构**

7.16 **团队组织的目标是为了减少必要的交流和协作量。**

7.17 **为了减少交流，组织结构包括了人力划分（division of labor）和限定职责范围（specialization of function）。**

7.18 传统的树状组织结构反映了权力的结构原理--**不允许双重领导**。

7.19 组织中的交流是网状，而不是树状结构，因而所有的特殊组织机制（往往体现成组织结构图中的虚线部分）都是为了进行调整，以克服树状组织结构中交流缺乏的困难。

7.20 每个子项目具有两个领导角色--产品负责人、技术主管或结构师。这两个角色的职能有着很大的区别，需要不同的技能。

7.21 两种角色中的任意组合可以是非常有效的：

.. 产品负责人和技术主管是同一个人。

.. 产品负责人作为总指挥，技术主管充当其左右手。

.. 技术主管作为总指挥，产品负责人充当其左右手。

**第9章 削足适履**

9.6 在大型的团队中，各个小组倾向于不断地局部优化，以满足自己的目标，而较少考虑队用户的整体影响。这种方向性的问题是大型项目的主要危险。

9.7 在整个实现的过程期间，系统结构师必须保持持续的警觉，确保连贯的系统完整性。

9.12 编程需要技术积累，每个项目需要自己的标准组件库。

**第10章 提纲挈领**

10.1 "前提：**在一片文件的汪洋中，少数文档形成了关键的枢纽，每个项目管理的工作都围绕着它们运转。它们是经理们的主要个人工具**。"

10.2 对于计算机硬件开发项目，关键文档是目标、手册、进度、预算、组织机构图、空间分配、以及机器本身的报价、预测和价格。

10.3 对于大学科系，关键文档类似：目标、课程描述、学位要求、研究报告、课程表和课程的安排、预算、教室分配、教师和研究生助手的分配。

10.4 **对于软件项目，要求是相同的：目标、用户手册、内部文档、进度、预算、组织机构图和工作空间分配。**

10.5 因此，即使是小型项目，项目经理也应该在项目早期规范化上述的一系列文档。

10.6 以上集合中每一个文档的准备工作都将注意力集中在对讨论的思索和提炼，而书写这项活动需要上百次的细小决定，正是由于它们的存在，人们才能从令人迷惑的现象中得到清晰、确定的策略。

10.7 对每个关键文档的维护提供了状态监督和预警机制。

10.9 **项目经理的基本职责是使每个人都向着相同的方向前进。**

10.10 **项目经理的主要日常工作是沟通，而不是做出决定；文档使各项计划和决策在整个团队范围内得到交流。**

**第11章 未雨绸缪**

11.5. 无论如何，都不要将一个完成度低的系统交付给客户/玩家。将开发的第一个系统--丢弃原型--发布给用户，可以获得时间，但是它的代价高昂--对于用户，使用极度痛苦；对于重新开发的人员，分散了精力；对于产品，影响了声誉，即使最好的再设计也难以挽回名声。因此，为舍弃而计划，无论如何，你一定要这样做。

11.7 **"开发人员交付的是用户满意程度，而不仅仅是实际的产品。"**（Cosgrove）

11.14 程序员不愿意为设计书写文档的原因，不仅仅是由于惰性。更多的是源于设计人员的踌躇--要为自己尝试性的设计决策进行辩解。（Cosgrove）

11.15 **为变更组建团队比为变更进行设计更加困难。**

11.16 **只要管理人员和技术人才的天赋允许，老板必须对他们的能力培养给予极大的关注，使管理人员和技术人才具有互换性；特别是希望能在技术和管理角色之间自由地分配人手的时候。**

11.18 **很容易为不同的晋升线建立相互一致的薪水级别，但要同等威信的建立需要一些强烈的心理措施：相同的办公室、一样的支持和技术调动的优先补偿。**

11.19 组建外科手术队伍式的软件开发团队是对上述问题所有方面的彻底冲击。对于灵活组织架构问题，这的确是一个长期行之有效的解决方案。

**前进两步，后退一步--程序维护**

11.20 程序维护基本上不同于硬件的维护；它主要由各种变更组成，如修复设计缺陷、新增功能、或者是使用环境或者配置变换引起的调整。

11.21 对于一个广泛使用的程序，其维护总成本通常是开发成本的40％或更多。

11.22 维护成本受用户数目的严重影响。用户越多，所发现的错误也越多。

11.23 Campbell指出了一个显示产品生命期中每月bug数的有趣曲线，它先是下降，然后攀升。

11.24 缺陷修复总会以（20－50）%的机率引入新的bug。

11.25 在每次修复之后，必须重新运行先前所有的测试用例，从而确保系统不会以更隐蔽的方式被破坏。

11.29 所有修改都倾向于破坏系统的架构，增加了系统的混乱程度。即使是最熟练的软件维护工作，也只是放缓了系统退化到不可修复混乱的进程，从中必须要重新进行设计。

**第12章 干将莫邪**

12.1 项目经理应该制订一套策略，以及为通用工具的开发分配资源，与此同时，他还必须意识到专业工具的需求。

12.6 抛开理论不谈，**一次分配给某个小组连续的目标时间块被证明是最好的安排方法，比不同小组的穿插使用更为有效。**

13.2 V.A.Vyssotsky提出，"**许许多多的失败完全源于那些产品未精确定义的地方**。"

13.3 **在编写任何代码之前，规格说明必须提交给测试小组，以详细地检查说明的完整性和明确性。开发人员自己不会完成这项工作。**（Vyssotsky）

**第14章 祸起萧墙**

14.1 "项目是怎样延迟了整整一年的时间？.一次一天。"

14.2 **一天一天的进度落后比起重大灾难，更难以识别、更不容易防范和更加难以弥补。**

14.3 根据一个严格的进度表来控制项目的第一个步骤是制订进度表，进度表由里程碑和日期组成。

14.4 **里程碑必须是具体的、特定的、可度量的事件，能进行清晰能定义。**

14.5 如果里程碑定义得非常明确，以致于无法自欺欺人时，程序员很少会就里程碑的进展弄虚作假。

14.7 **慢性进度偏离是士气杀手。**[Microsoft的Jim McCarthy说："如果你错过了一个最终期限（deadline），确保制订下一条deadline。2">

14.10 **PERT的准备工作是PERT图使用中最有价值的部分。**它包括了整个网状结构的展开、任务之间依赖关系的识别、各个任务链的估计。这些都要求在项目早期进行非常专业的计划。

14.11 第一份PERT图总是很恐怖的，不过人们总是不断进行努力，运用才智制订下一份PERT图。

14.12 PERT图为前面那个泄气的借口，"其他的部分反正会落后"，提供了答案。

14.16 必须有评审的机制，从而所有成员可以通过它了解真正的状态。出于这个目的，里程碑的计划和完成文档是关键。

14.17 Vyssotsky：我发现在里程碑报告中很容易记录"计划（老板的日期）"和"估计（最基层经理的日期）"的日期。项目经理必须停止对这些日期的怀疑。"

**第15章 另外一面**

15.2 即使对于完全开发给自己使用的程序，描述性文字也是必须的，因为它们会被用户－作者所遗忘。

**15.3 培训和管理人员基本上没有能向编程人员成功地灌输对待文档的积极态度--文档能在整个生命周期对克服懒惰和进度的压力起促进激励作用。**

15.10 如果这样，很少有程序需要一页纸以上的流程图。[在这一点上，MILSPEC军用标准实在错得很厉害。]

15.12 为了使文档易于维护，将它们合并至源程序是至关重要的，而不是作为独立文档进行保存。

15.13 最小化文档负担的3个关键思路：

.. 借助那些必须存在的语句，如名称和声明等，来附加尽可能多的"文档"信息。

.. 使用空格和格式来表现从属和嵌套关系，提高程序的可读性。

.. 以段落注释，特别是模块标题的形式，向程序中插入必要的记叙性文字。

15.14 程序修改人员所使用的文档中，除了描述事情如何以外，还应阐述它为什么那样。对于加深理解，目的是非常关键的，但即使是高级语言的语法，也不能表达目的。