# 一切从你的思维开始。

## chp1 焦油坑

### 2 职业的乐趣

- 创建事物的快乐
- 开发对其他人有用东西的乐趣
- 将可以活动、相互啮合的零部件组装成类似迷宫的东西，这个过程所体现出令人神魂颠倒的魅力
- 面对不重复的任务，不断学习的乐趣
- 工作在如此易于驾驭的介质上的乐趣-纯粹的思维活动-其存在、移动和运转方式完全不同于实际物体

### 3 职业的苦恼

- 将做事方式调整到追求完美是学习编程最困难部分
- 由其他人设定目标，并且必须依靠自己无法控制的事务（特别是程序），权威不等同于责任
- 实际情况看起来要比这点好一些，真正的权威来自于每次任务的完成
- 任何创造性活动都伴随着枯燥艰苦的劳动，编程也不例外
- 人们通常期望项目在接近结束时可以收敛的快一些，然而 情况却是越接近完成，收敛得越慢
- 产品在完成前，总面临着陈旧过时的威胁；只有实际需要时，才会用到最新的设想。

## chp2 人月神话

- 1. 缺乏合理的时间进度是造成项目滞后的主要原因，它比其他所有因素的总和影响还大。
- 2. 良好的烹饪需要时间，某些任务无法在不损害结果的情况下加快速度。
- 3. 所有的编程人员都是乐观主义者：一切都将运行良好
- 4. 由于编程人员通过纯粹的思维活动来开发，我们期待在实现过程中不会遇到困难。
- 5. 但是我们的构思是有缺陷的，因此总会有bug
- 6. 围绕成本核算的估计技术，混淆了工作量和项目进展，人月是危险性和带有欺骗性的神话。因为他暗示着人员数量和时间是可以相互替代的。
- 7. 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
- 8. 关于进度安排：1/3计划 1/6编码 1/4构件测试（单测） 1/4系统测试

- 9. 作为一门学科，我们缺少数据估计
- 10. 我们对自己的估计技术不确定，在外部压力下，缺乏坚持下去的勇气
- 11. brooks法则：为进度落后的项目增加人手，只会使项目更加落后
- 12. 中途向项目增加人手额外增加的工作量：任务重新分配和造成的中断；培训新人员；额外的相互沟通

## chp3 外科手术队伍

- 1. 同样有两年经验且受到相同培训的程序员，不同程序员间的生产率可能相差10倍
- 2. S E G 数据显示，经验和实际表现之间没有关系（持怀疑态度）
- 3. 小型、精干队伍是最好的-思绪尽可能好
- 4. 两个人的队伍，其中一个是领导者，常常是最佳的人员使用方法（参考婚姻设计）
- 5. 对于真正意义上的大型系统，小型精干的队伍太慢了
- 6. 实际上，绝大多数大型编程系统的经验显示，一拥而上的开发方法时高成本、速度缓慢、低效的，开发出的产品无法进行概念上的集成。
- 7. 一位首席程序员、类似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，还彻底减少了沟通的工作量。

## chp4 贵族专制、民主政治和系统设计

- 1. 概念完整性是系统设计中最重要的考虑因素
- 2. 功能与理解上的复杂程度的比值，才是衡量系统设计的最终标准。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）
- 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. 对于非常大型的项目，将体系结构方面的工作和具体的实现相分离，是获得概念完整性的强有力方法（同样适用于小型项目）-自顶向下设计，逐个子模块替换实现
- 5. 如果要得到系统概念上的完整性，就必须有人控制这些概念。这实际上是一种无需任何歉意的贵族专治统治。
- 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。
- 7. 概念上统一的系统能更快的开发和测试

## chp5 画蛇添足

- 1. 尽早交流和持续沟通能使结构师有较好的成本意识，使开发人员获得对设计的信心，并且不会混淆各自的分工。
- 2. 结构师如何成功影响实现

- 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

- 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
- 4. os/360是典型的画蛇添足的例子（windows NT 似乎是20世纪90年代的例子）
- 5. 为功能分配一个字节和微秒的优先权值是一个很有价值的规范化方法

## chp6 贯彻执行

- 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。
- 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。
- 3.处于精确性的考虑，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。
- 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；他们都可以作为表达的标准。
- 5. 设计实现，包括模拟仿真，可以充当一种体系结构的定义；这种方法有一些严重的缺点。
- 6. 直接整合是一种强制推行软件的结构性标准的方法。（硬件上也是如此-考虑内建在ROM中的Mac WIMP接口）
- 7. 如果起初至少有两种以上的实现，（体系结构）定义会更加整洁和规范
- 8. 允许体系结构师对实现人员的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（电子邮件现在是可选的介质）
- 9. 项目经理最好的朋友就是他每天要面对的对手-独立的产品测试机构/小组。

## chp7 为什么巴比伦塔会失败

- 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
- 2. 功能的不合理和系统缺陷纷纷出现。由于存在对其他人的各种假设，团队成员之间的理解开始出现偏差。
- 3. 团队应该以尽可能多的方式进行相互之间的交流：非正式地进行简要技术陈述的常规项目会议，共享的正式项目工作手册[以及通过电子邮件]
- 4. 项目工作手册不是独立的一篇文档，他是对项目必须产生的一些列文档进行组织的一种结构。
- 5. 项目所有的文档都必须是该工作手册结构的一部分

- 5. 项目所有的文档都必须是该工作手册结构的一部分
- 6. 需要尽早和仔细地设计工作手册结构
- 7. 事先制定良好结构的工作手册“可以将后来书写的文字放到合适的章节中”，并且可以提高产品手册的质量。
- 8. 每个团队成员应该了解所有的材料（工作手册），换句话说，每个团队成员应该能够看到所有的材料，网页即可
- 9. 实时更新是至关重要的。
- 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这些变更重要性的评述上。
- 11. os/360 项目工作手册开始采用的是纸介质，后来换成了微缩胶片。
- 12. 今天（即使在1975年），共享的电子手册是能达到这些目标的、更好的、更加低廉的、更加简单的机制。
- 14. parnas强烈地认为使每个人看到每件事的目标是完全错误的，各个部分应该被封装。从而没有人需要或者被允许看到其他部分的内部结构，只需要了解接口。
- 15. parnas的建议的确是灾难的处方，（parnas让我认可了该观点，使我彻底地改变了想法）
- 16. 团队组织的目标是为了减少必要的交流和协作量
- 17. 为了减少交流，组织结构包括了人力划分和限定职责范围
- 18. 传统的树状组织结构反映了权利的结构原理-不允许双重领导
- 19. 组织中的交流是网状，而不是树状结构，因此所有特殊组织机制（往往体现为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。
- 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很大区别，需要不同的技能。
- 21. 两种角色的任意组合都可以是非常有效的。
  - 1. 产品负责人和技术主管是同一个人
  - 2. 产品负责人作为总指挥，技术主管充当其左右手
  - 3. 技术主管作为总指挥，产品负责人充当其左右手。

## chp8 胸有成竹

- 1. 仅仅通过对编码部分时间的估计，然后乘以其他部分的相对系数，是无法得出对整项工作的精确估计的。
- 2. 构建独立小型程序的数据不适用于编程系统项目
- 3. 程序开发随程序规模的大量增长而增长
- 4. 一些发表的研究报告显示，指数大约在1.5

- 4. 三发表的研究报告显示，指数大约在1.5
- 5. Portman的ICL数据显示，相对于其他活动，全职程序员仅将50%的时间用于编程和调试。
- 6. Aron的IBM数据显示，生产率是系统各个部分的交互的函数，在1.5k代码行/人年到 10k代码行/人-年的范围内变化。
- 7. Hall的Bell实验室数据显示，对于已完成的产品，操作系统类的生产率是0.6k LOC/人-年，编译类工作的生产率约为2.2k LOC/人-年。
- 8. Brooks的OS/360数据与Harr的数据一致，操作系统0.6-0.8kLOC/人-年，编译器2-3kLOC/人-年。
- 10. 在基本语句级别，生产率看上去是一个常数。
- 11. 当使用适当的高级语言时，程序编制的生产率可以提高5倍。

## chp9 削足适履

- 1. 除了运行时间以外，程序所占据的内存空间也是主要开销。特别是对于操作系统，它的很多程序都是永久驻留内存的。
- 2. 即便如此，为内存花的钱也是物有所值的，比其他任何在配置上的投资效果都要好。规模本身不是坏事，但不必要的规模是不可取的。
- 3. 软件开发人员必须设立规模目标，控制规模，发一些减少规模的方法-就如同硬件开发人员为减少元器件所做的事一样。
- 4. 规模预算不仅仅在占据内存方面是明确的，同时还应该指明程序对磁盘的访问次数。
- 5. 规模预算必须与分配的功能相关联；在指明模块大小的同时，确切定义模块的功能。
- 6. 在大型团队中，各个小组倾向于不断的局部优化，以满足自己的目标，而较少考虑对用户的整体影响。这种方向性的问题是大型项目的主要风险。
- 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。
- 8. 从系统整体出发以及面向用户的态度是软件编程管理人员最重要的职能。
- 9. 在早期应该制定策略，以决定用户可选项目的粗细程度，因为将他们整体打包能够节省内存空间（常常还可节约成本）
- 11. 为了取得良好的空间-时间折衷，开发队伍需要得到特定某种语言或者机型的编程技能培训，特别是在使用新语言和新机器时。
- 12. 编程需要技术积累，每个项目需要自己的标准组件库。
- 14. 精炼、充分和快速的程序往往是战略性突破的结果，而不仅仅是技巧上的提高。
- 15. 这种突破常常是一种新型算法。
- 16. 更普遍的是，战略上的突破常来自于对数据或表的重新表达，数据的表现形式

是编程的根本。

## chp10 提纲挈领

- 1. 前提：在一片文件的汪洋中，少数文档成为了关键的枢纽，每个项目管理的工作都围绕着他们运转。他们是经理们的主要个人工具。
- 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。
- 3. 对于大学科系，关键文档类似于目标、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、教室和研究生助手的分配。
- 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。
- 5. 因此，即使是小型项目，项目经理也应该在项目早期对上述的一系列文档进行规范化。
- 6. 以上集合中每一个文档的准备工作都将注意力集中在思索和对讨论的提炼上，而书写这项活动需要上百次的细小决定。正是由于他们的存在，人们才能从令人迷惑的现象中得到清晰、确定的策略。
- 7. 对每个关键文档的维护提供了状态监督和预警机制。
- 8. 每个文档本身就可以作为检查列表或者数据库。
- 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. 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内得到交流。
- 11. 只有一小部分管理人员的时间-可能20%-用来从自己头脑外部获取信息。
- 12. 出于这个原因，广受吹捧的市场概念--支持管理人员的‘安全信息管理系统’并不基于反映管理人员行为的有效模型。

## chp11 未雨绸缪

- 1. 化学工程师已经认识到无法一步将实验室工作台上的反应过程移到工厂中，需要一个实验性工厂（pilot plan）来为提高产量和在缺乏保护的环境下运作提供宝贵经验。
- 2. 对于编程产品而言，这样的中间步骤同样是必要的，但是软件工程师在着手发布产品之前，却并不会常规性地进行实验性系统的现场测试。（现在，这已经成为了一项普遍的实验，beta版本不同于有限功能的原型，alpha版本同样是我倡导的实践）
- 3. 第一个开发的系统对于大多数项目并不合用。它可能太慢、太大，而且难以使用。或者三者兼而有之。
- 4. 系统的丢弃和重新设计可以一步完成，也可以一块块的实现，但这是必须完成的步骤。
- 5. 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称。

- 6. 因此，为舍弃而计划，无论如何，你一定要这样做。
- 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）。
- 8. 用户的实际需要和用户感觉会随着程序的构建、测试和使用而变化。
- 9. 软件产品易于掌握的特性和不可见性，导致他的构建人员（特别容易）面临着永恒的需求变更。
- 10. 目标上（和开发策略上）的一些正常变化无可避免，事先为他们做准备总比假设他们不会出现要好得多。
- 11. 为变更而计划软件产品的技术，特别是拥有细致的模块接口文档的结构化编程广为人知，但并没有相同规模的实践。尽可能的使用表驱动技术同样是有所帮助的。（现在内存的成本和规模使这项技术越来越出众）
- 12. 高级语言的使用、编译时操作、通过引用的声明整合和自文档技术能减少变更引起的错误。
- 13. 采用定义良好的数字化版本将变更量子化（阶段化）是当今标准的实践。
- 概述（14-19）为变更计划组织结构。
- 14. 程序员不愿意为设计书写文档，不仅仅是因为惰性，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。
- 15. 为变更组建团队比为变更进行设计更加困难。
- 16. 只要管理人员和技术人员的天赋允许，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；特别是希望在技术和管理角色之间自由的分配人手的时候。
- 17. 具有两条晋升线的高效组织机构存在着一些社会性的障碍，人们必须警惕并积极的同他它做持续的斗争。
- 18. 很容易为不同的晋升线建立相互一致的薪水级别，但同等威信的建立需要一些强烈的心理措施；相同的办公室、一样的支持以及技术调动的优先补偿。
- 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（20-27）
- 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，或者是使用环境或配置变换引起的调整。
- 21. 对于一个广泛使用的程序，其维护总成本通常是软件开发成本的40%或更多。
- 22. 维护成本受用户数目的影响。用户越多，所发现的错误也越多。
- 23. campbell指出了一个显示产品生命期中每月bug数的有趣曲线，其先是下降，然后上升。

然后上升。

- 24. 缺陷修复总会以20%-50%的几率引入新的bug。

- 25. 每次修复之后，必须重新运行先前所有的测试用例，确保系统不会以更隐蔽的方式被破坏。
- 26. 能消除、至少是能指明副作用的程序设计方法，对维护成本有很大的影响。
- 27. 同样，实现设计的人员越少、接口越少，产生的错误也就越少。
- 前进一步，后退一步--系统熵随时间增加（28-29）
- 28. Lehman和Belady发现，模块数量随大型操作系统（OS/360)版本号的增加呈线性增长，但是模块随版本号的指数增长而收到影响。
- 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（熵）。即使是最熟练的软件维护工作，也只是延缓了系统退化到不可修复的混乱状态的进程，以致必须要重新进行设计。（许多程序升级的真正需要，如性能等，尤其会冲击它的内部结构边界。原有边界引发的不足常常在日后才会出现。

## chp12 干将莫邪

- 1. 项目经理应该制定一套策略，并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。
- 2. 开发操作系统的队伍需要自己的目标机器，进行调试开发工作。相对于最快的速度而言，它更需要最大限度的内存，还需要安排一名系统程序员，以保证机器上的标准软件是及时更新和实时可用的。
- 3. 同时还需要配备调试机器和软件，以便在调试过程中，所有类型的程序参数可以被自动计数和测量。
- 4. 目标机器的使用需求是一种特殊曲线：刚开始使用率很低，突然出现爆发性的增长，接着趋于平缓。
- 5. 同天文工作者一样，大部分的调试工作总是在夜间完成。
- 6. 抛开理论不谈，一次分配给某个小组的连续的目标时间块是最好的安排方法，比不同小组的穿插使用更为有效。
- 7. 尽管技术不断变化，这种采用时间块来安排匮乏计算机资源的方式仍能够延续20年（1975），这是因为他的生产率最高。（在1995年仍然如此）
- 8. 如果目标机器是新产品，就需要一个目标机器的逻辑仿真装置。这样，可以更快地得到辅助调试平台。即使在真正机器出现之后，仿真装置仍可提供可靠的调试平台。
- 9. 主程序应该被分成：（1）一系列独立的私有开发库（2）正处在系统测试下的系统集成字库（3）发布版本。 正式的分离和进度提供了控制
- 10. 在编制程序的项目中，节省最大工作量的工具可能是文本编辑系统。
- 11. 系统文档中的巨大容量产生了新的不易理解的问题【例如，看看unix】，但是它比大多数未能详细描述编程系统特性的短小文章更加可取。

- 12. 自上而下、彻底地开发一个性能仿真装置。尽可能早地开始这项工作，仔细听取“他们的表达意见”
- 概述（13-16）高级语言
- 13. 只有懒散和惰性会妨碍高级语言和交互式编程的广泛应用（如今，他们已经在全世界广泛使用呢）
- 14. 高级语言不仅提高了生产率，还改进了调试：bug更少，而且更容易寻找。
- 15. 传统的反对意见-功能、目标代码的尺寸、目标代码速度，随着语言和编译器技术的进步已不再成为问题。
- 概述（17-19）交互式编程
- 17. 某些应用上，批处理系统绝不会被交互式系统所替代（依然适用）
- 18. 调试是系统编程中较慢和较困难的部分，而漫长的调试周转时间是调试的祸根。
- 19. 有限的数据表明，系统软件开发中，交互式编程的生产率至少是原来的两倍。

## chp13 整体部分

- 1. 第4.5.6章所意味的煞费苦心、详尽体系结构工作不但使产品更加易于使用，而且使开发更加容易进行且bug更不容易产生。
- 2. 许许多多的失败完全源于那些产品未精确定义的地方。
- 3. 在编写任何代码之前，规格说明必须提交给外部测试小组，以详细地检查说明的完整性和正确性。开发人员无法自己完成这项工作。
- 4. 十年内（1965-1975），Wirth自上而下地进行设计（逐步细化）将会是最重要的新型形式化软件开发方法。
- 5. Wirth主张，在每个步骤中，都尽可能使用级别较高的表达方法。
- 6. 好的自上而下的设计从4个方面避免了bug。
- 7. 有时必须回退，推翻顶层设计，重新开始。
- 8. 结构化编程中，程序的控制结构仅由支配代码块（相对于任意的跳转）的给定集合所组成。这种方法很好的避免了bug，是一种正确的思考方式。
- 9. Gold实验结果显示，在交互式调试过程中，第一次交互工作取得的进展是后续的3倍。这实际上获益于在调试之前仔细地调试计划。
- 10. 我发现对良好（对交互式调试做出快速响应）系统正确使用，往往要求每两小时的终端会话对应于两小时的桌面工作：1小时的会话后清理和文档工作，1小时为下一次变更和测试。
- 11. 系统调试（相对于单元测试）所花费的时间会比预料的更长。
- 12. 系统调试的困难程度证明了需要一种完备系统化和可计划的方法。
- 13. 系统调试仅仅应该在所有部件能够运作之后开始。（这既不同于为了查出接口

- 13. 系统调试仅仅应该在所有部件能够运作之后开始。（这就不同于为了查出接口bug所采取的“合在一起尝试”的方法，也不同于在所有构件单元的bug已知但未修复的情况下，即开始系统调试的做法。对于多个团队尤其如此）
- 14. 开发大量的辅助调试平台和测试代码是很值得的，代码量甚至可能有测试对象的一半。
- 15. 必须有人对变更和版本进行控制和文档化，团队成员应该使用开发库的各种受控拷贝来工作。
- 16. 系统测试期间，一次只添加一个构件。
- 17. Lehman和Belady出示了证据，变更的阶段（量子）要么很大，间隔很宽；要么小且频繁。而后者很容易变得不稳定。【微软的一个小团队使用了非常小而频繁的阶段（量子），结果每天晚上都需要重新编译生成增长中的系统】

## chp14 祸起萧墙

- 1. 项目是怎么被推迟了一年的？A：一次一天
- 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易弥补。
- 3. 根据一个严格的进度表来控制大型项目的第一个步骤是制定进度表，进度表由日期和里程碑组成。
- 4. 里程碑必须是具体的、特定的和可度量的事件，能进行清晰的定义。
- 5. 如果里程碑定义得非常明确，以至于无法自欺欺人时，程序员很少会对里程碑的进展弄虚作假。
- 6. 对于大型开发项目中的估计行为，政府的承包商所做的研究显示：每两周进行修订的活动时间估计，随着开始时间的临近不会有大的变化；短期内对时间长短的过高估计，会随着活动的进行持续下降；过低估计直到计划的结束日期之前大约三周左右，才会有所变化。
- 7. 慢性进度偏离是士气杀手，如果你错过了一个最终期限（deadline），那就确保完成下一个最终期限。
- 8. 同优秀的棒球队伍一样，进取对于杰出的软件开发团队是不可缺少的必要品德。
- 9. 不存在关键路径进度的替代品，使人们能够辨别计划偏移的情况。
- 10. PERT图的准备工作是PERT图使用中的最有价值的部分。它包括了整个网状结构的展开、任务之间依赖关系的识别和各个任务链的估计。这些都要求在项目早期进行非常专业的计划。
- 11. 第一份PERT图总是很恐怖，不过人们总是不断努力，运用才智来制定下一份PERT图。
- 12. PERT图为那个使人泄气的借口-其他的部分反正会落后--提供了答案
- 13. 每个老板同时需要采取行动的异常信息以及用来分析和早期预警的状态数据。
- 14. 状态的获取是困难的，因为下属经理有充分的理由不提供信息共享。
- 15. 老板的不良反应肯定会对信息的完全公开造成压制；相反，仔细区分状态报

告、毫无惊慌的接收报告、绝不越俎代庖，将能鼓励诚实的汇报。

- 16. 必须有评审机制，使所有成员可以通过它了解真正的状态。出于这个目的，里程碑的进度和完成文档是关键。
- 17. Vsy: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（最基层经历的日期）的日期。项目经理必须停止对估计日期的怀疑。
- 18. 对于大型项目， 一个对里程碑报告进行维护的计划和控制小组是非常可贵的。

## chp15 另外一面

- 1. 对于软件编程产品来说，程序向用户所呈现的面貌--文档，与提供给机器识别的内容同样重要。
- 2. 即使是完全开发给自己使用的程序，描述性文字也是必需的，因为它们会被用户-作者所遗忘。
- 3. 培训和管理人员基本上没有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客服懒惰和进度的压力起起促进和激励作用。
- 4. 这样的失败并不因为都是缺乏热情或说服力，而是没能正确地展示如何有效和经济地编制文档。
- 5. 大多数文档只提供了很少的总结性内容。必须放慢脚步，稳妥地进行。
- 6. 由于关键的用户文档包含了与软件相关的基本决策，因此它的绝大部分需要在程序编制前书写。
- 7. 每一份发布的程序拷贝应该包括一些测试用例，其中一部分由于校验输入数据，一部分用于边界输入数据，另一部分用于无效的输入数据。
- 8. 对于必须修改程序的人而言，他们需要程序内部结构文档，同样要求一份清晰明了的概述，它包括了5项内容
- 9. 流程图是被吹捧的最过分的一种程序文档。详细逐一记录的流程图是一件令人生厌的事情，而且高级语言的出现使它显得陈旧过时。（流程图是图形化的高级语言）
- 10. 如果这样，很少有程序需要一页纸以上的流程图
- 11. 即使的确需要一张程序结构图，也并不需要遵照ANSI的流程图标准。
- 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。
- 13. 最小化文档担负的三个关键思路：
  - 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。

- 14. 程序修改人员所使用的的文档中，除了描述事情如何，还应阐述它为什么那样。对于加深理解，目的是非常关键的，即使是高级语言的语法，也不能表达目的。
- 15. 在线系统的高级语言（应该使用的工具）中，自文档化技术发现了它的绝佳应用和强大功能。

## chp19 20年后的总结

- 1. 概念完整性和结构师
- 2. 开发第2个系统所引起的后果-盲目的功能和频率猜测
- 3. 图形界面的成功
  - 通过类比获得概念上的完整性
- 4. 没有构建舍弃原型-瀑布模型是错误的
- 5. 增量开发模型更佳-渐进地精化
  - 构建闭环的框架系统
    - 它也许就是一个划分好模块功能的一个空的框架，没有实际的功能实现（仅由空函数）
    - 按模块一个个的开发，在每个阶段，都有一个可运行的系统。
    - 功能基本可以运行之后，再一个个的精化或重写模块，增量地开发（growing）整个系统。
  - 增量式开发和快速原型系统
    - 原型：仅仅反映了概念模型准备过程中所做的设计决策的一个程序版本，它并为反映受实现考虑所驱使的设计决策。
    - 核心区别：功能-增量式开发每次加入的功能模块都是和后续上线保持一致的。
- 6. 关于信息隐藏，Parnas是正确的
  - Parnas：代码模块应该采用定义良好的接口来封装，模块内部应该是程序员的私有财产，外部是不可见的。编程人员被屏蔽而不是暴露在他人模块的内部结构面前。
- 7. 人月到底有多少神话色彩？Boehm的模型和数据
- 8. 人就是一切（或者说，几乎一切）
  - 推荐书籍：《人件：高生产率的项目和团队》
  - 项目转移
- 9. 放弃权利的力量
- 10. 最令人惊讶的新事物是什么？数百万的计算机
  - 改变了每个人使用计算机的方式

- 改变了每个人使用计算机的方式
- 改变了每个人开发软件的方式

11. 全新的软件产业-塑料薄膜包装的成品软件
    - 传统软件产业
    - 操作系统世界已经统一了
    - 塑料薄膜包装的成品软件产业
12. 买来开发-使用塑料包装的成品软件包作为构件
    - 元编程
    - 它处理的确实是根本问题
13. 软件工程的状态和未来