Understanding the Professional Programmer

理解专业程序员

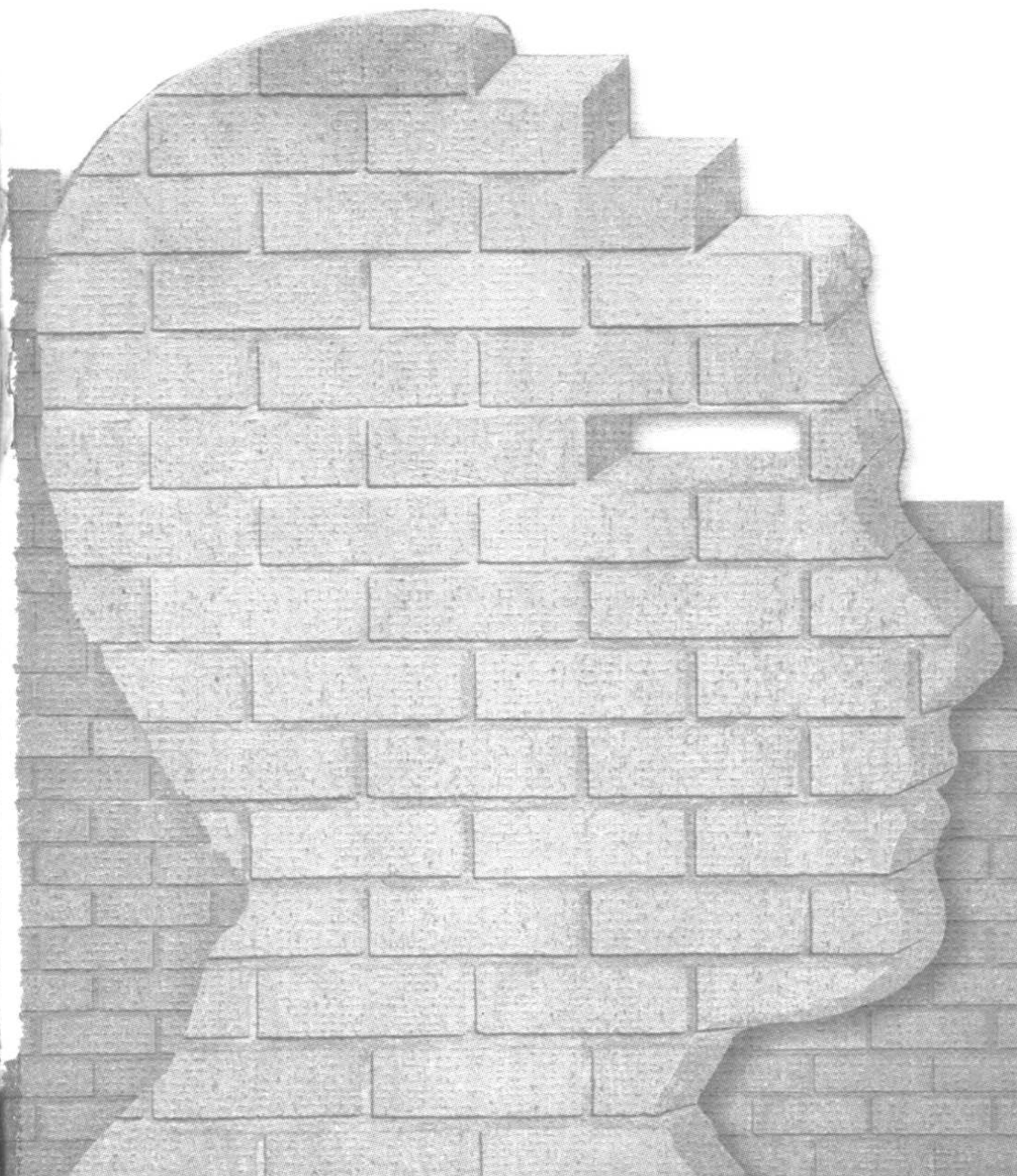
(美)杰拉尔德·温伯格 /著
刘天北 /译

清华大学出版社

Understanding the Professional Programmer

理解专业程序员

(美)杰拉尔德·温伯格 /著
刘天北 /译



清华大学出版社
北京

Understanding the Professional Programmer
By Gerald M. Weinberg
EISBN:0-932633-09-9

图书在版编目(CIP)数据

理解专业程序员 / (美)温伯格(Weinberg, G. M.)著;刘天北译. —北京: 清华大学出版社,2006.7
书名原文:Understanding the Professional Programmer
ISBN 7-302-12994-0

Ⅰ. 理…　Ⅱ. ①温… ②刘…　Ⅲ. 程序设计—文集　Ⅳ. TP311.1-53

中国版本图书馆 CIP 数据核字(2006)第 051059 号

出 版 者: 清华大学出版社　　**地　　址**: 北京清华大学学研大厦
http://www.tup.com.cn　　**邮　　编**: 100084
社 总 机: 010-62770175　　**客户服务**: 010-62776969
责任编辑: 贺　岩
封面设计: 李尘工作室
印 刷 者: 清华大学印刷厂
装 订 者: 三河市李旗庄少明装订厂
发 行 者: 新华书店总店北京发行所
开　　本: 148×210　**印张**: 6.125　**插页**: 1　**字数**: 177 千字
版　　次: 2006 年 7 月第 1 版　2006 年 7 月第1 次印刷
书　　号: ISBN 7-302-12994-0/TP·8247
印　　数: 1~4000
定　　价: 25.00 元

代译者序

认识你自己

可以用一句略带美国味的话来总结你捧在手上的书："它让你笑，它让你叫，它让你跳。"

和所有的温伯格作品一样，这本书首先是一本易读的作品——毋宁说，是一本可读性很强的作品。我们这些从十多岁就开始接受理工科教育，多年来一直善于和机器打交道的程序员，常常给外人（乃至我们自己）留下一些神秘，甚至古怪的印象——当你每天穿着牛仔裤和旅游鞋上班，每句话带上一两个三字母缩写词，趴在电脑前一整天看着那些稀奇古怪的符码，外人很难不把你看做一个与众不同的"geek"。在geek的圈子里，温伯格就算得上是咱们大家的老前辈、老祖师了，听他讲起那些几十年前的geek们的笑话，文化认同极度匮乏的中国同行们该有一种加倍的亲切感。

但是，我得提醒你注意，这本书的名字叫《理解专业程序员》。要照温伯格老先生的说法，15年时间恐怕还不够造就一个专业程序员。于是你也就不难想到，在他老先生的眼里，你、我，以及咱们身边几乎所有的程序员或多或少都带着些"不专业"的味道。所以你就得做好准备了，这位老先生讲的笑话一多半是在讽刺"不专业"的程序员，这些带着刺的笑话极有可能句句刺着你的心窝子。比方说吧，要是你刚离开学校没多久（甚或还没离开学校），现在就翻开"想打板球的蟋蟀"和"想打棒球的蟋蟀"这两个小故事看看，我敢打赌你脸上的表情不会太优雅——那种泛着红晕的苦笑，我可是太熟悉了。

很多时候，被人讽刺不一定就是坏事，尤其是这

些来自专业程序员的讽刺。对专业程序员的误解来自两方面。于其外,那些西装革履、衣着光鲜的外人不仅不理解程序员的“专业”,他们甚至常常就是故意制造着误解——用温伯格的话来说,“程序员们成了各种行业媒体寻常而方便的靶子”;于其内,众多的程序员就像那只可敬的蟋蟀一样找不到“专业”的门径,再加上形形色色的 FUD(Fear, Uncertainty, Doubt)不停地在耳边萦绕,很多人自己就开始相信……唔,点到为止吧,在网络上你不难找到他们相信的东西。这本《理解专业程序员》首先就是要厘清误解,把一个“专业”的目标、一种职业的认同感还给程序员。

和温伯格所有的作品一样,这本书一直在强调着、渲染着程序员的文化。谈起“文化”,温伯格举得最多的例子就是医生——既然医生可以用一种内生的文化为自己筑起藩篱,以便提升自己的地位和价值,程序员为什么不可以?(有心的读者不妨留意书中和医生有关的故事,看看他们是如何借助来自希腊文的生僻词汇营造文化壁垒的,这算得上是一个典型的传播学案例了。)而营造一种文化,第一件要务无疑是区分“专业人士”和“业余爱好者”(或许还应该加上“不那么专业的从业人员”)。既然包括 Fred Brooks 在内的众多专家都赞同程序员水平的差异会对生产率造成几个数量级的影响,那么这种专业/非专业的划分就该是对所有人——而不仅仅是程序员自己——有利的。

说实话,对于你在阅读本书时即将体验到的复杂心情,我非常理解——因为我也刚刚体验过。上一分钟,你也许还在为一个与己无关的笑话而捧腹;下一分钟,被嘲笑的对象好像又变成了你自己。我衷心希望本书的读者能珍视这种难得的阅读体验。特尔斐神庙的先知告诉人们:“认识你自己。”作为一个有志于在专业程序员之路上探索跋涉的旅行者,能够看到这条路的方向,能够有人(尽管有些尖锐地)指出自己的不足之处,这该是弥足珍贵的。

透　明

2005 年 3 月于杭州

著者序

1944 年，我 11 岁，我读到了《时代》(*Time*)杂志上一篇谈计算机的文章！在那个年纪，我的脑子是一块干净的石板，等着它的粉笔，所以那篇文章给我留下了很深的印象。我记得自己坐在起居室那把高背椅上，《时代》放在膝头，而整个“时代”都在我手中，当时我就决定，我要成为一个“计算机人”。1944 年以来，计算机发生了巨大的变革。由于计算机的应用，我们的生活也发生了巨大的变革——我自己的生活就更其如此。但至少一件事没有变化。显然，从那以后几乎过去了一生的时光，我也从计算机文章的读者变成了作者。但是即使是今天，仍然没人理解“计算机人”是种什么人，他们究竟做什么，他们应该做什么。

不能说“没人知道”。这里的问题是：**人人**都知道——但每人知道的都不一样。我在计算机界度过近乎一生之后，尘埃还未落定。计算机人士仍有**选择**他们做什么的自由——而依据这个自由，他们也就能选择究竟让计算机做什么。也许，再过 50 或 100 年，计算机人士将被塞入界线清晰的鸽子窝中，度过一生。可是今天，我们的命运似乎还在自己手上。

迄今为止，计算机产业一直建筑在思想的自由交流上。从前，你每月都能参加一次会议，在会上能和——比如说洛杉矶市内的——每

一家大公司的某个代表谈话。现在呢，怀旧派还会去 NCC[①] 大会，想在来自世界各地的 75 000 名同行之中找回过去的那种氛围。可那其实已经大不相同了。

确实，永远也找不回旧日子了。我们从前凭借直觉（有点儿类似于动物本能）取得的经验，现在只能通过更加明确的结构体系来获得。比如说，在那些大型会议上，已经没有留给“小”主意的空间了。而在那些小型会议上，更没有留给小人物的空间——大家都是来听大人物咳珠唾玉的。

从前呢，我们的工作进程特别依赖于会议，因为很少有书面材料。当 1956 年我开始为 IBM 工作的时候，我用了不到一周的时间，就读完了旧金山分部的每一篇技术文献（包括公共库中的所有文件）！这也大概就是 IBM 在 1956 年拥有的所有技术文献了。

今天，我们简直淹没在技术文献当中，很少再有时间进行面对面的交流，至于“非正式读物”，那似乎就完全没时间看了。就这样，各处的计算机人士失去了彼此的联络，失去了思想交流的机会——尤其是那些算不上特别“技术性”的思想，因为它们的模糊特质，也很少能被收入各种使用手册或教科书。而一旦不再交流这些思想，我们也就失去了对于自身现实中很重要的一部分的接触，我们的工作也因此受到了损害。

我定期为几种刊物撰写专栏文章（专栏标题分别是“Stateside”，“Phase 2”以及“From Eagle, Nebraska”[②]），会收到大量读者来信，当阅读这些来信时，我就能特别强烈地感受到上面说的那种“缺乏对现实的接触”。这些读者都是计算机人士，分别在杜布克、都柏林和丹尼丁[③]工作。虽然他们彼此有那么多相近之处，但却不能有机会面对面交谈。如果他们真能遇上，他们肯定能马上认出对方就是“计算机人

---

① NCC 大会，全称是“全国计算机业大会”（National Computer Conference），美国计算机业曾经举办的系列年会，现在大概已经停办了——作者为中译本做的注解。

② 这些专栏刊载在全球多种著名专业杂志上，包括英国、澳大利亚、新西兰的 Computerworld 杂志，以及日本的 BIT 杂志——作者为中译本做的注解。

③ 杜布克、都柏林和丹尼丁（Dubuque, Dublin, Dunedin）是作者按照字母顺序随机举出的 3 个城市名，足见读者遍及天下。

士”——就像他们从我的文章中认出自身那样。但是我说的“计算机人士”是什么意思呢？是那种人，他读着这些文章，会不时说“是呀，我就是这样做的”，或者是“我也这么想”，或者是“但愿我的经理能懂这个”，或者是“我可以用上这个技巧”，甚或是“这个叫温伯格的家伙已经弄不明白我工作的这种环境了”。

所以，这一组文章，是写给那些想要更好地了解计算机人士的读者的——所谓计算机人士，也就包括程序员、分析师、经理、设计师、培训师、测试员、维护人员、操作员、系统管理员、架构师、企业领导，或者其他各种头衔下的那些人。大部分计算机人士的时间都很宝贵，为了不浪费这宝贵的阅读时间，我也力图把文章写得有趣。只要你有 5 分钟空闲时间（比如你在等着 TSO① 响应的那一会儿工夫），你就可以打开本书的任何一页，读完其中的一点想法。这可能不是世界上最伟大的想法，但还是会有一定益处。即使没有益处，大概也能为你带来一点儿消遣——这本身也算是一种益处吧。

① TSO：即 Time-Sharing Option，时分复用。

前言

“专业的”(professional)这个词,既包括几层明晰清楚的“显义”,也带有好些含混模糊的“隐义”。我想先考察一下本书标题中的“专业的”是取自哪些意思,这样读者也就能更好地理解,本书究竟涉及了哪些范围。

我的《美国传统词典》说,“专业的”的意思是:“从属于、关联于、介身于或适合于一项职业”,这又把该词的意思抛给了“职业”一词的定义。这样,也会引发下列争论:编程是一种“职业”吗?它应该是一种职业吗?它又怎样才能成为一种职业?可在本书中,我并不打算让读者投身于以上争论,虽然可以肯定,书中的多篇随笔将向上述“职业”主题抛去少许光亮,或是些微暗影。

词典里的第二和第三个定义强调了工作的报酬。但是我所说的“专业的程序员”,可能为编程工作收取报酬,但并不一定收取。而且,还有很多人,他们确实为编程工作收取报酬,但我却**不**会把他们当成“专业的”,我的理由呢,读者看了以下随笔后自然就能明了。

跟我的想法最接近的定义是这个:**在特定的活动领域里,具备了不起的技艺或经验**。

这本书,讲的就是在计算机编程领域具备了不起的技艺或经验的那些人。

归属于“计算机编程”这个题目下的内容非常丰富,与我早前写的《程序开发心理学》(*The Psychology of Computer Programming*)那本书不同,本书并未试图囊括所有那些内容。这里的重点,是那些技艺高超、经验丰富的工作者——在程序开发这样一个波谲云诡、变动无常的环境中,我们怎样才能成为这样一个人呢?我们又怎样才能保持这样的水准呢?

我的老朋友、老对头 Phil Kraft 曾经为了《程序开发心理学》的书名狠批过我一回。按我的理解，他想说的意思是：从字面上讲，不存在一项"活动"的"心理学"，只存在个人的心理学，或是人群的心理学。《理解专业程序员》(Understanding the Professional Programmer)这个书名，想必会让 Phil 好过一些(不过我怀疑，本书的不少内容又会让他火冒三丈)。在书名里，我用的是"程序员"的单数形式，这是因为本书中我重点考察的是作为个体的程序员。

《理解专业程序员》的首要目标，是为专业程序员提供一种自我考察、自我测试的方法。一位非常机智的咨询顾问 Eugene Kennedy[①] 说过：

> 有些人严厉、粗疏的自我考察，往往包含某种混杂的动机，半是出于担惊受怕，半是由于自己具有某些欲望，又不愿被人发现。正如大多数传教士和募捐者知道的那样，让人们感到自己有罪，这实在并非难事……可是，由于自我考察往往不必要地让人自认有罪，所以在很多人眼里，自我考察也担上了恶名；他们或逃避，或拖延自我考察，因此也就从未意识到它的价值。

我的意图，不是在专业程序员中间引发罪责感。恰恰相反，我认为，程序员们成了各种行业媒体寻常而方便的靶子，这些行业媒体主要关心的是兜售硬件；在那些缺乏认真思维习惯的记者们看来，程序员，对于提高硬件销量和广告收入来说，仅仅是一块绊脚石。

Kennedy 接下来的这段话正好可以描述我的观点：

> 有另一种看待自我的方式；专业人士——无论是医生还是运动员——更多地就是这么做的：为了解放自我，持久地提升自己的表现，个人需要接受某种特定的规训[②]。换句话说，健康的专业人士自我考察，不是为了惩罚自身，而是为了获得自我提升。

如果你想成为上述意义上的"专业程序员"，这本《理解专业程序

---

① Eugene Kennedy，美国著名的非虚构类畅销书作家。所著《怎样成为咨询顾问：非专业咨询顾问指南》是该领域的经典读物。

② 规训，原文 discipline，一般译为学科、训练、纪律、惩戒等。中文中很少有现成语汇能涵盖上述多个义项。该词也是法国后结构主义哲学的一个核心概念，大陆学术译著中通译为"规训"，现从此译法。

员》就恰恰是为你而作。

致谢

我想向以下同行致谢，我在书中引用了他们和我的通信：David Coan, Jo Edkins, Bob Finkenaur, David Flint, D. A. Martin 和 Barbara Walker。我也想感谢其他的很多通信者，他们激发了我的灵感，不过很难在这里一一列出姓名了。

在准备本书时，我搜集了想归入这里的多篇随笔，有几位同事通读了它们，并帮我划分了内容的级别，在此我对他们表达特别的感激。他们的评语和建议又一次告诉我，评论对于创造性工作来说有多么重要。我的这个“专家组”包括：Jim Fleming, Mason C. Gibson, Tim Gill, Bill Hetzel, Roger House, Bob Marcus 和 Paul Mellick。没有他们，本书就没法完成。

当然，要是没有我家里的那个“内部专家组”（包括 Dani[①] 和 Judy），本书也不可能完成。她们从不惮于把想入非非的我带回现实。

① Dani 是本书作者 Gerald M. Weinberg 的太太，一位人类学家。

目录

第1章

对专业人士来说，有哪些重要问题

成为一个程序员要花多长时间

对于有些事情，似乎每个人都是专家。教学就是一个好例子。任何人，只要智商超过80，又懂得一点儿什么东西，似乎都可以当老师。至少美国的教育体系就是建立在上面这个理论上的。在美国，但凡你敢对一个教授说，他的课堂教学还有可改进的地方，那他就会感到羞辱、恼怒，还很可能采取法律行动。

还是在美国，每个人都是当招待的专家。在欧洲，一个侍者可能要经过10年，甚至20年的训练，才能获准在一个一流饭馆服务。在美国，只要按照广告应征，在小臂上搭一条毛巾，那就是侍者了。

编程是另一个不缺乏专家的领域。按照标准看法，6个星期的“培训”就足以把一个人提升到“专家”层次，该人不必再学习任何新的知识，即具有设计在线生命救援系统的资格。如果你看到一条广告招收“有经验的”程序员，那意思往往就是一年或者两年经验。实际上，如果谁有15年的编程经验，人们倒会觉得这人简直是个智障。如果他真有一点点智力的话，那总应该在14年前就学会了全部编程知识。在此之后，他就早该做腻了这一行，去换个管理呀，销售呀之类的职位了。

先别忙着嘲笑持这种观点的人，首先我们还是应该承认，15年的经验，就其自身而言，在编程方面不一定就能教会你任何东西。我认识一些有“15年经验”的美国侍者，甚至不知道餐前如何在餐桌上放盘

子。我也知道一些有“15 年经验”的美国大学教授，甚至教不会小狗摇尾巴。同样，我也认识一些有“15 年经验”的美国程序员，他们仍然会在一个多程序访问的系统中，在更新直接存取主文件(master file)之前，就给事务文件(transaction file)排序。

如果说这个例子还太难懂，那我就来列举几个前两天读“有经验的”程序员写的代码时发现的问题：

1. 在做整数除法时，有些人不懂“余数”是什么东西！

2. 为了把一个取值在 0～5 的变量转化成取值在 1～6 的变量(用于 FORTRAN 语言的下标)，有人用了 5 个 IF 语句，再加上 5 个赋值语句！

3. 在写 COBOL 程序的时候，有些人不用“ELSE”子句，原因是“这不一定管用”。

4. 在写 PL/I 程序的时候，有些人从来不用变长字符串，原因是“这个不够高效”。

5. 有些人根本不写子程序，原因是“这太复杂了”。

这个单子能够无限地写下去。这里的要点不是在于，居然有这么多看似专业程序员的人在四处丢人现眼，而在于，没有几个管理者知道，正在和自己打交道的到底是“他们”中的一员，还是“我们”中的一员。

这和美国侍者的处境特别相似。在美国，很少有人曾经享受过专业侍者的服务，所以即使人们真正遇到了一个专业侍者，他们也无从辨别。或者这样说更好，他们根本无法意识到，他们心目中的“标准”侍者其实还处于“亚专业”层次。

同样，除非你自己就是一个胜任的程序员，否则也就很难衡量一个程序员的工作质量。世上有很多可怜的企业，这些企业中从来没能长期留住一个真正胜任的程序员，因此他们也就没有一套标准来衡量程序员的专业性。这些企业的标准就是把庸人当成奇才。而这样的标准也千奇百怪，各地均不相同，甚至同一公司中的不同部门也不相同。

每次我到一家新公司去做咨询顾问的时候，我都提前让经理给我看一些典型代码。经理们往往都不敢相信我真是要看代码，我总得坚

持索要好几次才能得手。只要看一小段代码，我通常就能对该公司的工作环境具有相当准确的了解。有时候我说得特别准，管理层听了都大吃一惊，以为此前我跟员工们私下谈过话。

经理们自己永远也不看代码。代码之于经理，如同脏盘子之于领班侍者。一旦你从那个垃圾堆里提升出来，你就再也不碰那些垃圾了——开玩笑碰一下都不成。

有一回，在大学里的时候，我们学生提议，教授们也应该和学生一起参加硕士生考试，好给学生们做个榜样、立个标准。2/3 以上的教授对此满是惊恐，敬谢不敏。他们自己也经过 20 多年的考试折磨，再也不愿意回到考生的位置上去——这会让他们想起从前卑微的地位。

同样，在我们的行业里经理不愿意编码，这说明写代码这个职业在人类等级体系中的地位略高于盗墓者，低于管理层。对于这样的思考方式来说，编写代码不可能构成一种独立的技艺，不可能是一种天分，也不可能是一种有着自身地位的体面职业——所谓体面，就是说不必和盗墓呀，管理呀之类的在同一个尺度下衡量。只要这种态度在数据处理行业还处于主导地位，那就仍然会有 6 个星期培养出来的专家，也还会有那些经理——他们甚至不愿倾听公司高薪聘请的、有 15 年经验的程序员说话。

当老师、当侍者、当程序员，这 3 件事有什么共同之处吗？为什么人人都觉得自己能够像专业人士一样做这 3 件事？首先，这些工作似乎是容易理解的，因为很多挺普通的人都有过相关的经验。每个人都或多或少曾经教过别人。每个人都做过把盘子放在桌上，或者收拾脏盘子的事。但是不是每个人都曾经在一个活人大脑上做过手术，也不是每个人都曾经在陪审团前为一个案件辩护。

但是编程序又是什么情形呢？当然了，并不是每个人都写过程序，对不对？也许不是每个人都写过，但是似乎每个经理、会计、工程师，或者其他大学毕业的专业人士都写过程序。编程课程在大学里相当风行，在很多职业教育中，这也是必修的课程。比如说，IBM 在 20 年来，在行政人员培训班中就设置了一定的“编程经验”。

我不太清楚现在 IBM 的行政人员培训班的具体课程内容，但是有好多年这门课程中包括了那个著名的“曼哈顿问题”，作为唯一的编

程练习。在美国，数据处理课程的主流入门教科书大多会讲到这个“曼哈顿问题”，如果读者中有人不巧没学过这个，我就按照教科书上的写法，在这里重复一遍：

问题是这样的：据说在 1627 年，白人们用 24 块钱买了曼哈顿岛。如果这笔钱被存入一个银行户头，按年利率 4.5%计算，今天会有多少钱？

（如果 4.5%的年利率偏低的话，那是因为这道题是 1956 年出的，从那时起就被一代代的作者在不同的教科书中抄来抄去。）

这道题的“解法”，如果抛开一些无关紧要的细节，按照 FORTRAN 语言编写，那就是这样一个循环：

```
      I = 1627
      PRINC = 24.00
2     PRINC=PRINC * 1.045
      I = I + 1
      IF(I-IYEAR)2,1,1
1     WRITE (3,601) PRINC
```

至少有三四百万名学生学会了这个“解法”，这之中包括从行政人员到大学新生的各种人。对于其中的一些人，以上代码就是他们“写过”的唯一程序，但是这就让他们有足够资格判断编写一个操作系统、一个劳动力部署系统、一个零件需求管理模拟器、一个在线处理控制器，或者无论什么你想得出来的系统的复杂度。而且，当然了，在行政人员的课程中，每个学生还有一个专业程序员作为辅导，“好帮助他们处理细节问题”。

其实呢，曼哈顿问题确实可以作为一个出色的工具，教给行政人员关于编程行业他们应该知道的最重要的一课。假设让他们编写了以上那么一段程序，也对他们承认这确实是问题的一个“解法”。然后你就问问他们，编这个程序花了多少时间，运行该程序又要多长时间，再问问他们，觉得这些数字“好不好”。

当他们交了作业，也总结了感受，你就让他们看看下面这个程序，告诉他们这样的代码就能获得同样的结果：

```
PRINC=24.00 * (1.045 * * (IYEAR-1627))
WRITE(3,601)PRINC
```

对他们比较一下编程时间和运行时间。你大概能够发现这后一个程序只需要 1/5 的编程时间，和 1/100 的运行时间，当然具体的比例在不同的环境下不一样。然后你就问他们：“如果对这样一个最简单的程序，两种不同的代码之间能够具有 5 倍，甚至 100 倍的差别，那么，如果一个专业程序员和一个业余程序员编写同样一个操作系统的话，又会产生多大差别呢？”

如果给行政人员上了这样一课，那么这种给他们扫盲、让他们理解编程是怎么回事的课程也许能够利大于弊。但是目前这一类课程的主要目的，虽然从来没有明言，但其实是这样的：“编程并没有那么复杂。练习几个星期，哪怕是我也能成为编程专家。”

为了把编程当成一种正规职业对待，公众——也包括程序员自己——都应该通过某种方式受到教育。他们必须懂得这样一个道理：即使是 15 年的经验，对于学习编程知识来说也不一定就够用——除非这位学习者特别一心一意。

残障人士能成为成功的程序员吗

只要编程人才的市场变成了卖方市场，雇主们就会试图扩大供应源。那些死板的灰头发老经理们，发现最近的毕业生要求实习期间工资高过管理人员，所以转而也开始面试女性了。盎格鲁-萨克逊血统、信仰新教的经理们也开始面试少数族裔的应聘者，虽然他们的皮肤并不是粉红底色带雀斑①，而他们所信的宗教呢，经理们连怎么念都不知道。而那些身强体壮，早年在橄榄球场上曾经屡战屡捷的经理们，现在也来向我咨询："雇佣残障人士"是不是"安全"。

当特权阶级暴露出对弱势阶级的无知时，往往是很让人恼火的事情。（所谓"非特权阶级"这样的词，本身就是特权阶级为了粉饰太平发明的，所以我不会允许使用这一类说法。）当然，捡起一种明显的弱势标签也是一件容易的事，因为这样一来就没人敢质疑你的正当性。但另一方面，如果在严格的商业意义上（而不是在"商业人士的社会责任感"的意义上）说"雇佣残障人士"，人家就会指责你简直是赤裸裸的商业主义、完全缺乏情感。然而，对于残障程序员问题，我想进行一个出于理性的讨论。如果这种看待问题的方式让你恼火，那么对于残障人士（至少是被你标为"残障"的那些人）来说，你很可能是一个糟糕的雇主。

残障，说到底，是一种标签——它标明的与其说是身体状态，不如说是精神状态。我这个观点大概有点儿陌生，那么让我举一些例子，看看在编程领域之外——具体地说就是体育领域，残障人士的工作成绩到底怎样。

一天晚上，我们看了一个电视节目，介绍的是一个捷克的手枪射击冠军，他在第二次世界大战中失去了右臂，大家当然都会同意，这对于一个右撇子射击选手来说就是一种残障。但是，这个人相当热爱射击，并不因为失去了右臂就放弃了这项运动，所以他自己学着用左手

① 盎格鲁-萨克逊人种是北美的主流族裔，血统纯正的该种族后裔往往金发碧眼，并有"带雀斑的粉红色皮肤"。因为大多数为英格兰移民，所以该种族也以新教徒居多。

射击。而且最后得了一块金牌！

另外一个电视节目讲的是美国短跑选手 Wilma Rudolph 的故事。在小时候，她病得很重，人家说她永远都不能走路了。我忘了她最后得了多少块奥运金牌，但是当然比你我得的都要多了——可是从来没人跟我说，我永远都不能走路了。

当然了，这个故事也许只能说明当时美国的医疗水平让人不敢恭维，我想起还有一个打破 1 英里长跑纪录的运动员，小时候得过小儿麻痹症，人家也说他不能走路了，这可能确实是医疗水平不高的证据。（另一方面，美国橄榄球界最近出了几个好手，要么是先天脚部畸形，要么是被切除过脚趾，还有一个是用的义腿，所以也许美国的医疗水平其实也没那么糟糕。）

其实，我以为这些故事与医疗水平关系不大，相反这与雇佣残障人士的问题有很大关系。根据这些故事（还有其他很多故事），根据我对一些失明、失聪、截瘫、四肢瘫痪，以及其他各种残障的程序员工作表现的观察，我可以得出这样的结论：成功在于精神，而不在于眼睛、耳朵、手臂或者双腿。当一个残障者决定对整个世界证明她（他）与旁人没有什么区别时，她（他）的努力经常不仅能够弥补先天的不足，还可以大大超出，令她（他）在编程工作中获得冠军——就像在赛跑中夺冠的那些残障人士一样。

克服一种看似无法克服的残障，这种能力对我来说是一个重要的启示。成为一名橄榄球冠军所需要的因素中，如果有 1 份在于双腿的话，那么就有 100 份在于个人的内在动力。说到底，腿是能够训练出来的，可是我们怎么训练自己的“内在动力”？按照上面的故事，如果切除了几个脚趾，也许这个运动员的“动力”就能被调动起来，可是我们真的甘愿这么做吗？你愿不愿意切除部分大脑，以获得成为一个优秀程序员的“动力”？如果你真地愿意，那么也许你已经动力满满了，大可不必再去做那个手术。

想想你认识的那些程序员吧。有多少人，看似天赋不薄，有一副好脑子、好身体，却从来没能从平庸中脱颖而出？你想怎么做才能让他们意识到自己的幸运？对大多数人来说，只有到了失去这些天赋和幸运的时候，才会知道它们的可贵。

说起我自己，我一直认为，自己有双手是理所当然的事——直到我患上了关节炎，打字都会觉得痛苦。当时我抱怨个不停——想想看，这种关节炎对于一个作家来说有多糟糕——直到有一天我遇见了一个程序员，他根本就没有手。我还没有像他那样，在头上绑着一支笔来打字。

对于编程来说，有一种特殊的残疾，无论个人怎样有动力，似乎都没法克服这个。我指的显然是缺乏做编程工作必备的智力。当然，我倒不是说，“低智商”也是一种精神状态——也许我就是这个意思？好吧，我得承认，世上就是有那么一种人，根本就缺乏智商来理解什么是计算机，什么是程序员，所以也就没有成为计算机程序员的智商。但是我以为，这一类人的数量并不像很多读者想象的那么多。

对计算机编程确实需要智力。事实上，这个工作需要太高的智力，以至于没人真正能做得特别好。当然，有些程序员比别的程序员强，但是在做这项工作时，大家都像是总是误打误撞的老小孩。为什么会这样呢？因为计算机编程是人类以前从来没有遇到过的、最难的智力工作。

根据你的宗教信仰不同，你可能会相信上苍造人是为了(1)两臂挂在树枝之间摇来晃去吃香蕉；(2)生活在伊甸园中吃苹果；(3)什么也不为。据我所知，还没有人相信上苍造人就是专门为了编写正确、紧凑、高效、可维护，而且还廉价的计算机程序的。所以，如果有谁失去了某个肢体，或是某种感官，其实在编程能力上他与我们其他人没有太大区别。也许雇佣残障者就需要添置某种特别的辅助设备，而这种设备可能并不昂贵，但是我们在装修办公室时，总要安上很多五花八门的小玩艺儿，只要我们觉得这些东西能够提高工作效率。所以，为了残障辅助设备花钱，精打细算的管理者也能算清这笔账，明白这其实是划得来的，因为与那些自以为健全的程序员相比，身患残障的程序员往往工作业绩更大。

可是有些管理者担心，有一个残障者在办公室里，会对其他工作者产生某种影响。根据我的见闻，事实上这种影响主要在于，让人们意识到自己是多么幸运，而付出的努力又多么少。这样一种意识确实会使编程工作产生变化，但是如果一个管理者连这样的变化都应付不

了，那他也就根本不是在管理了。

在编程这件事上，我们全都身有残障，虽然我们可能还并不自知。因为我们平时只用上了很小一部分的潜能，所以，当我们想要向世界证明什么东西的时候，我们也总能超越自我。所以，承认、接受自己身有残障的事实，这应该算是一个优势。如果我们承认、接受了“每个人在精神上都存在残障”这个事实，那么也许编程也能成为奥运项目。

专业程序员有哪些范式

1978 年的图灵奖(Turing Award)授给了斯坦福大学的 Robert W. Floyd。他授奖演说的题目是《编程的种种范式》,刊登在 1979 年 8 月号的《ACM 通讯》(*Communications of the ACM*)杂志上。我推荐每个读者 ——也就是与编程有关的每个人—— 都去读一读 Floyd 的演说稿。我也很乐意把他的一些想法导入到另一个方向上去。

一种范式是一种模式—— 对于编程来说,也就是一种思考模式。Floyd 仔细考察了若干范式,包括

结构化编程

动态编程

递归协同程序

基于规则的系统

非确定系统

编程语言

他谈到了“计算机编程技术目前的发展水平”(照我看就是一团糟),并且说:

> 我们最大的心愿就是提高我们自身的能力。我相信,为了提升计算机编程的总体实践水平,最好的方式就是更多地关注本行业的各种范式。

和其他计算机业内人士一样,Floyd 自己也致力于发明新的范式,提炼现有的范式,以及将各种范式明白无误地传授给下一代程序员。我相信他的这些业绩足以使他获得图灵奖。我尤其乐见的是,他利用领奖的机会向大家展示了自己内心的工作机制——也许这能叫做一种“元范式”,因为这就是 Floyd 如何能够创造出这么多饱含智慧的业绩的奥秘。

我从这篇文章里提炼出来的“元范式”包括以下内容:

1. 使用类比,把计算机的处理过程与人类机构中的处理过程联系起来,对比研究;

2. 在解决复杂的问题时，并不是仅仅得到一个解法就满足了，而是反复追溯自己的思路；

3. 从特殊案例中推出普遍规则，并且利用其他特殊案例检验这些普遍规则；

4. 阅读其他人的范式；

5. 阅读别人的程序，尽量扩展自己的理解能力，专门去读懂那些“写法奇特”的人的代码，从而在读程序时取得更多的收获；

6. 尝试跟别人交流范式，或者把自己的范式教给别人，以此进一步澄清自己的思想；

7. 积极地调查别人此前的工作成果，而不是重新发明一切；

8. 把别人的工作看做一个出发点，就此对自己提问：“我会怎样发明这个东西？”

正如 Floyd 自己总结的那样：

> 我想对那些严肃的程序员这样说：在每个工作日中，花一部分时间来考察、精炼你的工作方法。虽然程序员们总是要赶在最后期限之前完成工作（当然也可能已经过了最后期限还没完成工作），但在方法论上的提取和精炼将是一笔明智的长效投资。

如果你是一个严肃的程序员，你就应该留意这个建议。你可能还会把“范式”或“方法论上的提取和精炼”的意义加以扩展。从本质上讲，Floyd 的所有例子都涉及这样一个场景：当一个程序员接到一个定义清晰的问题时，他或她应该怎样做。知道怎样处理定义清晰的问题，这非常重要，但还并不足够。为什么呢？因为在实际工作中，专业程序员花在定义不清的问题上要多得多，比如在以下情况下：

1. 一个用户不知道问题是什么；

2. 一个用户知道问题是什么，但是其实问题在另外的事情上；

3. 一个用户知道问题是什么，可程序员不理解、不相信，或者不听从那个用户；

4. 有不止一个用户，其中的每个用户都处在上面 3 种情况的一种或多种之中；

5. 用户的数量随时间变化；

6. 问题随时间变化；

7. 所有人都在大喊大叫，都丧失了理智。

在那些定义不清的情况下，程序员往往会自作主张，定义一个他们自己愿意解决的问题。Floyd 提到，在斯坦福大学一个研究生办公室的墙上，有这么一句涂鸦："与其写程序，不如写帮我写程序的程序。"我们能看到，本来"方法论上的精炼"是一种健康行为，但到了这里却濒于病态——它成了一种逃避我们自己不愿意处理的问题的方式。

对于那些专门用来逃避问题的范式，专业程序员当然自有一整套丰富的方法论说辞。如果有一个笨头笨脑的客户，敢于拿着一个定义不清或者沉闷无趣的问题踏进我们神圣的办公室，那么我们就会把那一套说辞兜售给这个客户。我相信，总有一天某个程序员会对客户这么说："只要我们找到那个合适的范式——像 Floyd 教授教给我们的那样——我们就能完成这个程序了。在这段时间里，别拿你的问题来打扰我们。"

在我看来——我想 Floyd 本人也不会反对——提取用于解决那些定义清晰问题的范式，这只不过是程序员们的部分职责。还另有一些范式，更深刻、更模糊，与编程工作自身的特性紧密相关，对这类范式的提取至少与前者同样重要。对我而言，所有范式里最有普遍性的一个就是，"专业程序员是为其他人解决问题的人——无论问题有多么麻烦"。

专业程序员可能会用到的方法包括：

1. 如果有这样一个定义清晰的问题，我相信根本就不应该去解决它，那么我应该怎样处理为这个问题编程的任务？

2. 我怎样才能确定究竟是否应该使用计算机？

3. 计算机程序总是要在人与机器构成的总体环境中运行，那么，我应该怎样设计这一总体环境？

4. 我应该怎样设计计算机程序，才能避免它对其运行的总体环境产生副作用？

5. 总有一些人，他们带着定义不清的问题，他们对计算机的了解不如我多，但他们对其他很多事情都懂得比我多，而且他们的工作动力也和我很不一致——我怎么才能更有效地与他们协作？

6. 如果在某个情况下，我作为一个专业程序员不能有效地工作，那么我怎样才能改变这样的处境？

7. 未来充满了不确定因素——问题本身会发生改变，人员会发生改变，我也不一定会一直在场，那么我如何保证自己创造的程序在未来仍然能够胜任工作？

8. 对于一个特定的时刻，我怎样确定采用哪种层次的方法论才最能适合当下的工作？

9. 在我的个性、我解决问题的方式中有一些方面，我自己由于身在其中，很难察觉到它们，但它们甚至可能是决定程序员工作效率的最重要的因素。那么我又应该怎样来调节这些因素呢？

也许以上内容都与“宁愿写自己编写程序的程序”的“计算机科学家们”无关。如果确实如此，那么这与谁有关呢？似乎还没有太多的人蜂拥着去解决这些问题——惟其如此，严肃的程序员才应该自己着手考虑这些问题——作为一笔明智的长效投资。

一个专业人士能从这个职位中感到快乐吗

不少程序员和分析师都为数据处理专业人员所负有的职责深感苦恼，我就经常遇到这样苦恼的同行。今天我们在计算机上的操作，可能会影响到今后的多年里亿万人的生活。而且，其中的大部分人都不会把他们生活中的不愉快归结到我们今天的操作上。他们可能会温顺地接受这样的解释："计算机只能这么工作"，或者另一种更阴险的解释："事情就是这样的"。

我认识一些专业人员，尤其是一些在那种缺乏监督机制的企业工作的程序员，他们会蓄意破坏雇主的信息系统，以此来减轻良心的不安。有些情况下，很难判断这到底是有意的、还是无意的。但是有些情况下，毫无疑问是有意的。

很多程序员和分析师都对我抱怨过，他们觉得自己的工作对个人来说毫无意义。他们写了一段程序，或是一份规格书，但不知道这到底有什么用，要么就是知道有什么用，但他们并不赞成这个目的。对此，他们的对策就是继续目前的工作，继续领薪水，而且一有安全的机会就说雇主的坏话。

我认为，是到了我们挺身而出、仗义执言的时候了。对于那些生活受到我们创造的系统影响的人，我们负有重大责任。如果我们不相信自己的雇主正在进行的工作，或者，如果我们不能理解这种工作，那么为什么我们还要在他们那里工作呢？难道只是为了领一份高薪？如果真是这样，那我们又成了什么人？

若干年前，我写了一篇文章，其中列出了一些原则，指导那些正在找一份新工作的程序员和分析师。很多人看了这篇文章之后，都说这些原则不仅对找新工作有帮助，而且对于正在考虑离职的人，对于想要改变受雇条件的人，都有帮助。

考虑这些问题的人似乎越来越多。为了帮助这些人，我把那些原则改编成以下问题，这可以用来评估一份新工作或者一份旧工作。

1. 这家企业的目标与我的个人信条是否一致？

2. 我在企业中的职责清晰吗？我是否能够认同它？

3. 管理层是否能为我的专业发展提供足够的时间和资源？这种时间和资源的提供，究竟是偶尔如此，还是一种明确的、长期的承诺？

4. 管理层是不是故意促使我和同事们进行竞争，并以此对我进行评估？还是鼓励我与同事相互协作，帮助他人完全发挥自己的潜能，并且也鼓励别人这样对待我？

5. 我理解别人交给我做的差使吗？我理解别人为什么让我做这样的事情吗？企业是否鼓励员工理解自己的任务？

6. 我和其他人的工作是不是都能够公开让同级评审？我是否希望参与评审别人和被人评审？

7. 我对这家企业、这个项目投入的程度，能否达到人们的期望？

那些用这些问题扪心自问，并且认真回答的人，就不会陷入那种糟糕的工作中。

用这些问题扪心自问不是一件容易的事。有时候你会得到你并不喜欢的答案，这时你就不得不做出选择。这当然不一定就是坏事，但是确实会非常困难。我的一个同事发现，他对自己项目的目标不能认同，却为了高薪接受了那份工作。因此他离职找了一份薪水低一些的工作，他的一家人，虽然比以前穷了一些，但是都满心欢喜，因为这个新工作确实让他们觉得幸福。

有那么几次，人们发现自己的经理们看了其中的某个问题之后暴跳如雷，因此他们意识到，经理们生气，说明今后肯定要有麻烦。于是他们通过种种办法为自己找到了新的经理。

当我把其中的一些问题发表在一篇专栏文章里的时候，我收到了很多感谢信。有一封信，是一名叫 Jo Edkins 的读者来的，特别让我惊讶。我想让大家也看看这封信，我来一个部分一个部分地引用，再加上自己的评述。

开始的部分是一些恭维的话，作为炮弹上的糖衣，然后 Jo 就开始了正题：

> 我完全同意你的态度，但是我这个人比较玩世不恭，所以觉得这里的细节有点儿不切实际。认识一个企业或者一个项目的目标，对于一个 20 人以下的企业这是个不错的想法。我所在的

公司是一家跨国企业，我不太相信有任何人知道它的目标是什么。

我也同意，在有些情况下确实很难，甚至不可能知道一家企业的目标，但还是让我们从能知道目标的那些情况开始考虑。我和烟草公司的员工们有过几次深入的交谈。他们也许不知道自己企业的真实目标，但是他们确实对企业在世界上的作用有些了解，这个让他们不太舒服。在那样一种环境下，他们在企业内部的任何作为都没法让自己好过一些。当然了，并不是这种公司里的每个人都这么想。很多人认为烟草是对人类的一大福祉——碰巧我太太就属于这种人。问题不在于谁的价值观是正确的，而在于，如果你的努力工作却会效力于你所不相信的价值观，那么你还能不能在这种情况下工作。如果你离职了，这可能是对你的个人安乐的最大贡献，而且也有助于推广你自己的价值观。最终，如果没人愿意为烟草公司工作，那么这个行业的劳动力成本就会居高不下，这些公司就会关张。

也许，对于很多目标并不明确的情况，答案都是“这目标并不要紧”。但是，如果你觉得，“自己参与的是哪种价值观”确实对你至关紧要，那么也许加入一家“20 人以下的企业”倒是最佳选择了。这也就是我自己的选择，我还认识很多程序员，他们也是这么做的——恰恰是出于以上原因。

有些公司确实太过巨大、太过笨重，以至于已经不知道自己的目标了，如果有足够多的人从这些公司离职，那么也就能提高这一类公司的劳动力成本，这对于庞大和颟顸无疑是一种惩戒。最终，这能够调节我们社会的形态和结构。当然这不会有太大的作用，但与在那种大公司工作相比，这可能还是要好些，因为在那样的工作中你甚至会怀疑，你干的越努力，世界反而会因此越糟糕。

但是，也并不是所有大公司都要么颟顸昏庸，要么居心叵测。这也就是 Jo 的下一个反对意见：

再换一个低一些的视角考虑，有这样一条管理学的规律：“一个知识的层次与我们想把它告诉给其他人（尤其是卑下者）的愿望成反比。”这是生活中的一个现实。当然了，我们每个人都可以自己开办软件公司，但是，现在雇佣我们的公司有没有良好的目

标呢——他们会把这个目标告诉我们吗？

我完全同意。我的很多朋友都对我夸口自己的目标多么纯洁，但他们受雇的公司却会制造有毒气体，或者会吓唬那些善良的老太太——人家只是看不懂他们的煤气账单，交费晚了而已。

即使你是软件行业里，或者是在培训行业里，你还是要考虑自己工作导致的后果。如果你考虑得比较深远，就可能会发现一些你并不喜欢的事情。出于这种原因，我就拒绝了不少生意，但是我确实接受来自中央情报局或者类似机构的学员上我的课，虽然我从不为这些机构本身做咨询。我的理由是：来上我的课程的客户，只代表个人，而不代表他们的机构。

实际上，我的一些学生是由公司支付的学费，但他们学完我的课程之后就立刻离开了该公司。有意思的是，虽然在这种情况下我总会提出要偿还学费，到现在为止，每一家这样的公司倒是都不以为意，他们觉得这样心怀不满的员工不要也罢——无论在技术上这个人多么优秀。（另外有一点儿并非完全离题的话，我不喜欢听到有人说自己“属于”一家公司，也不愿意听到用“高等的、卑下的”这样的词来描述两个工作者之间的关系。我相信，这样的说法会对我们的思维产生某种影响，会使我们真的开始认为自己——或别人——是卑下的，也就会让我们或多或少地把自己——或别人——看成奴隶。）

在下一段里，Jo 开始讨论大学的话题——这也是我最喜欢抨击的目标，他说：

> 我们可以去做学院派，但是这也不一定就肯定问心无愧：我们负责维护的计算机会不会被用来保存学生的政治材料？学校会不会把这些材料交给公司？（这当然是计算机的一种该死的用途。）我还是愿意在我能自得其所的环境下工作。

我还是完全同意。所谓学院派，其实尤其会屈从于某种危险的自我幻觉，他们不知道或者不想知道自己雇主的实际所作所为。我本来在一所美国大学里有终身教职，恰恰是因为学校在用计算机做 Jo 提到的那些事情——以及更为糟糕的事情，我才从这所大学离职的。计算机技术人员的地位得天独厚，最便于发现雇主们冠冕堂皇的说法和实际目的、实际作为之间的不一致。但是，如果你确实一开始就不相

信那些公共目标的价值，那又另当别论了。就像甘地曾说过的那样："如果你根本不认为酒吧间有价值，那么再去为酒吧间里的种族隔离争吵也就没什么意义了。"

如果你确实愿意基本认同企业的目标，那么 Jo 的以下说法就很有道理：

> 以离职作为威胁——即使是在编程人才短缺的情况下——也不是一个好办法，而且往往可能完全事与愿违。

我觉得有点遗憾，因为原来那篇文章（像理查德·尼克松常说的那样）"没把话说清楚"。我当然不建议任何人把离职当成"威胁"。如果你觉得目前的工作简直难以接受，你应该要么努力改变它，使它能够让人接受，要么干脆离职。不要用离职来威胁。威胁不仅仅"可能"会事与愿违，它几乎保证会事与愿违。在最好的情况下，雇主会以为你只是想要更多的薪水而已；而在大企业里，这样做则肯定会让你明白一个道理：没有谁是不可取代的。

> 如果你离开一个地方，那么这个地方会因为你的离开而更加恶化（因为一个很积极的批评者走了）。如果你留下，抱怨、叫喊、批评、建议、劝告，那么你会最终有所收获，你会让别人也意识到这里存在的问题，而且有可能（当然也只是可能而已）你能给环境带来些许改观。

上述意见说得再好不过了，但是我还是要指出，Jo 建议的这条路可能会非常难走，非常孤独。而且，如果你最后也没能造成一点点改善，那么你可能会特别灰心，也许以后做任何事情都再也鼓不起干劲。正因为如此，你一开始就应该作出这个判断：到底能不能心安理得地与该企业的价值观念共处——这对你自身至关重要。如果你突然发现自己一直努力工作，却无意中为一些你痛恨的东西——比如有毒气体，或者黑材料——为虎作伥，到那时再寻求对此做出改善，这当然不会是你愿意遇到的处境。

另一点至关紧要的是，在你致力改善局面的时候，至少该有影响总体规划的一点可能性。无论那个总体规划看起来多么冠冕堂皇，如果你自己已经对此失去了成功的希望，那么继续为此工作，就会扑灭你内心仅存的一点火花——如果没有这火花，到了下一次战斗的时候

你也就无法点燃自己了。

我写下这一串问题的一个主要原因是，在过去的 20 年里，我见过很多优秀的男男女女因为某种蹩脚的目标而葬送了前程——他们自己从一开始就知道那个目标很蹩脚。这么多年了，我见到这种情况还是不能心中释然、一笑了之。每次见到这样一个故事，还都会像第一次遇到那样让我难受。

在一开始的时候，所有这些勇士对他们的工作，对整个编程行业，都是激情澎湃——就像 Jo 在这封信中说的那样：

> 现在，很多人觉得计算机工作最担心的是两件事，一是失去个人私密空间；二是失业。有时候，我自己最担心的就只是编得不好的程序，或者设计得不好的系统。至于斯旺西的 DLVC[①] 会对整个国家的技术进步发生什么影响，我才不愿意考虑呢。但是，如果所有最棒的人都离职了，事情肯定不会变得更好。唉！直到最近两年，在商业公司的编程人员才慢慢懂得了一个“革命性的概念”：在写一个程序之前，先要做设计。之所以有这样的变化，还是大家在行业内部致力变革的结果。

我自己也见过这个行业里很多灾难性的情况了，我坚信，对于其中的大多数事例，如果企业中最棒的人能够离职，那么结果会好得多——离职越早越好。我见过很多构思拙劣的项目，因为最棒的人都走了，所以项目最后垮台了——因此也就没有对公众造成恶劣影响。我认为，如果在企业内部无论怎样努力都没法改变全局，那么你最好的选择就是“用脚投票”——这是你对于整个世界能做的最大贡献了。

你应该这么看问题：在这个注定失败的项目中，那些经理们可能对技术上的状况一无所知，只能通过“卑下者”的提议才能略微了解实情。有一个下级告诉他们，整个公司绝对是走在一条大难临头的路上了，可是经理们根本无动于衷，所以那个下级也就继续工作，照领工资。在这么一个情况下，经理们会相信那个“卑下者”说的话是真的

---

① 斯旺西是英国的一个城市，至于 Jo 说的 DLVC 指的是什么，我也不太敢肯定。也许这是“Driver and Vehicle Licensing”（司机和车辆执照）的缩写，不过本书出版这些年来，这个术语恐怕也发生了很大变化——作者为中译本做的注解。

吗？若要是你，你会相信吗？

Jo的下列意见我仍然同意：

> 我认为，应该教育公众不接受质量低劣的计算机相关产品，这才是保护公众不受这样的产品侵害的最好办法。当然，凭着我们微薄的影响力，我们也能做出自己的贡献，但是对于一个行业来说，安全保障最终还是应该来自这个行业外部……这当然不会免除分析师或程序员自身的责任。一个分析师或程序员应该尽可能做到最好，但是，单单他一个人不可能解决整个问题。

说到底，如果我们自己都接受了低劣的计算机相关产品，那也就没法“教育公众不接受这样的产品”。所以，我们自己必须学会识别那种几乎必然会产生这种产品的局面，这样才能避免在经济上、情感上陷入这种局面，不能自拔。

当然，没人能够完全确定地预测未来。我们所能做的最好的，也就是回首过去，从往事中提取出一些普遍的原则 —如果我们当时贯彻了这些原则，就能够避免一些麻烦的。我们每人都必须犯一些错误才能成长，但是人生苦短，如果每件事情都要通过痛苦的经历才能学会，也未免太过艰难。此外，在计算机行业，还有太多真正值得去做的事情等着我们，所以对于那些愚蠢的、痛苦的、完全错误的工作，我们也实在不是非要尝试才行。比如说，为什么我们不致力于让公众更加尊重（而不是像现在这样惧怕和嘲弄）我们的行业呢？正如Jo所说：

> 我确实同意你的以下看法：数据处理专家们不应该仅仅盯着自己的工资收入和晋升机会，更应该关注他们的公共责任。我知道你一直在倡导他们这样做。祝你好运！

也祝你好运！

没耐心的心理分析师：一个寓言

一个心理分析师，一整天遇到的都是非常难办的病人，但是这个叫 George 的病人尤其不合作。心理分析师都开始怀疑自己的专业能力了，他搜索枯肠，想找一些本行业证明行之有效的办法来对付这个不驯服的 George。

他决定采用“自由联想”来试一试。“George，”他说，“我要你清除所有杂念，让脑子一片空白。然后，当我敲这支铅笔的时候，我要你告诉我这时你的第一个念头。”

当分析师做出了敲铅笔的信号后，George 说，“我想让你吻我，大夫。”

分析师十分恼火，但是气急败坏当然显得很不专业，所以他还是强压怒气。“这个不行，George，”他回答说，“你又在重复老一套了，我们知道你应该尽量抹去这些东西。我要你再试一次。清除你刚才那个杂念，然后告诉我你在想什么。”

George 试着做了，然后很抱歉地说，“对不起，大夫。无论我多么努力地试着做，我想到的却全是要你吻我。”

这一回，大夫的怒气再也忍不住了，带到了他的回答中：“那样不行，George。想点儿别的。”

“我想不了别的。只想让你吻我。”

心理分析师完全失控了，“我跟你说过了，跟你说了一千次了，我不会吻你的。所以，打消这个念头吧。”

“但是为什么你不愿意吻我呢？”George 哀求说。

“为什么我不愿意吻你？嗨，我都不应该跟你一起在这个躺椅[①]上！”

教训：当你们已经一起在躺椅上了，那担心接吻的事情也许太晚了。

换句话说：当你发现自己总在专业工作中为一些小问题恼火，也许这是因为，从前有某个重要问题你已经回答错了。

---

① 躺椅：这是接受心理分析的病人常用的一种陈设。

第2章

专业程序员是怎样达到专业性的

不能把程序员的教育完全托付给计算机:他们太珍贵了

选择一种教学语言

每隔4年,伴随着县选举,本地的计算机科学教授们也会重新开始争论:哪种语言是编程教学的正确之选。跟县选一样,这里也会有"把流氓扔出去"的叫嚣,有很多顿午餐专门用于选战,选举前夕信心百倍的暴发户最终也会一败涂地。最后,FORTRAN语言总都和老县长一样重登大位。也许他们腐败;也许他们无能;也许他们老态龙钟;但至少他们还都算是熟面孔。

就像县选的选民一样,教授们也很乐意为这个结果找些理由。1956年以来这么多年过去了,教授们的论据,一个接着一个地,都枯萎衰败了。不过还有一个论据幸存下来,每一年都会被当成各种保守主义的主心骨被提出来。咱们听听一位正牌教授的说法:

"我之所以选择WATFIV[①]语言作为本课程的基础,不仅是出于对现实世界应用情况的认识,也是出于对我们机构中计算机成本的实际经济考虑。"

---

① WATFIV,即WATerloo Fortran IV,是FORTRAN语言的一个变种,由加拿大滑铁卢大学开发。

“现实世界”和“实际经济”，这些可都不是废话呀——不是那些象牙塔里出来的胡说八道。那就让我们看看，在现实世界中，对于培训用的计算机成本，究竟有什么样的实际经济考虑。

20 年前，计算机确实昂贵——昂贵到了 FORTRAN 作为编程课程的教学语言都不能被考虑的地步。实际上，当时即使再敢想，如果提出把一台计算机用于编程课程，那也只能被斥为轻浮夸大。

对于县长的一生来说，20 年是一段漫长的岁月了，对于编程职业来说，甚至还要更长一些。现在很少有人记得当年的艰辛了，甚至大家都不会相信：没有计算机怎么教授编程？在不久的将来，随着个人计算机的普及，也不会有人记得原来学习计算机还要一个老师的。在人们做出那个关于 WATFIV 的决定之后的几年内，实际经济情况已经发生了很大改变，我们几乎可以给每个学生一台个人计算机了。在这个摩登时代，用单数说“我们的系统”已经是落伍了。哪个像样的大学，没有十几台，甚至更多的小型机，至于微型机，那就更是四处散布在校园里，简直像是发给每家每户的小广告。

近年来硬件成本下降了这么多，但支持 FORTRAN 的论据还是幸存了下来。现在呢，是因为微型机不能运行别的，就认——天杀的——BASIC[①] 语言。这种保守主义是从哪儿来的？到哪儿才能打住？它要传播多远？这么多语言设计者、语言实现者，付出了一次次的努力心血，就是为了打垮它，为什么它还是能负隅顽抗呢？

使用软件工具

编程语言本身就是最古老、最为人熟知的软件工具，同样，大学也是最古老、最为人熟知的为社会变化服务的工具。对于新工具的保守主义态度，在所有其他机构中，针对所有其他工具，其实都有体现。所以，如果我们能回答上面关于保守主义的提问，它的终结也许就为期不远了。

不使用新工具有很多原因，其中最明显的是对培训缺乏重视。用于软件工具研发的金钱简直是庞然大象——目前大概每年超过 10 亿

① 我们知道，BASIC 语言也是 FORTRAN 的一个变种。

美元。与这头大象相比，花在培训人们使用这些工具的投资，则渺小得像微生物。

作为研发和培训之间这一隔阂的后果，研发出来的大部分工具都很少——甚至从来没有——被使用过。人们不仅还继续使用 FORTRAN 语言，甚至在使用时也从来都不运用一些便利技巧，比如说，程序变量的交叉参考列表。即使他们的 FORTRAN 编译器支持交叉参考，他们所在的机构也不允许使用，这种禁令往往是一个“标准”，理由是“代价太大”。即使这个交叉参考列表按常规制作出来了，90%的程序员也不会看上一眼；可是，这种列表是最简单的软件工具之一——使用起来最直接、最方便、最古老的工具之一。

即使是在很多已经抛弃 FORTRAN，选用“高级”语言的机构中，情况也不见得就好。我的客户和学员们研究了不少已经出版的程序代码，作为正规评审技巧的一种练习。这些程序代码似乎是被当成让新手效法的样板的。我们对两位计算机科学教授写的程序作了一次标准的评审，研究其对 PL/I 语言的使用。我们发现，PL/I 语言中的以下便利技术在这一程序里都没有用上，虽然该语言的规范中已经给出了应用所有这些技术的典型场景：

1. 动态存储分配
2. 截面记法
3. 数组表达式
4. 属性分解
5. 下标表达式
6. 类型转换控制
7. 位串

其实完全可以把这些代码中的分号去掉，那样简直就能用 FORTRAN 编译器编译了。

这些程序是一个非常典型的情况，我们接下去的考察证明，这么多年来对于编程风格的讨论，居然没有在这些程序中留下任何影响。我们发现，程序中有以下特别令人遗憾的做法：

1. 错综复杂的分支，其中包括循环的入口、出口。
2. 对于前面语句的操作理解不够透彻，导致了不少多余语句。

3. 在使用变量的循环出口处又初始化这些变量。

4. 使用单字符变量名，比如 K、R。

5. 在一个没有存储限制的程序中，使用，并且重用无符号变量。

6. 同一个名称却有两种不同意义，一种意义出现在程序中，另一种出现在解释性的注释中。

7. 在非常混乱的语境中，使用关键字 PTR 作为数据名称。

8. 各处变量命名不一致。

9. 在一个“为了高效”而大量使用指针的程序中，却低效地把运算放在循环内部，而且使优化器无法对之优化。

在设计层面，这些程序也丝毫没有体现出业界最近的讨论，更别说用上设计工具和设计概念了。我们发现以下问题：

1. 没有对输入有效性做检查——既不检查下标，也不检查值。

2. 用未被检查的输入数据来控制计算，比如程序分支。

3. 输入格式完全未经设计，很容易导致错误。

4. 使用的算法，只在很少几种输入的情况下还过得去，在其他情况下都非常低效。

5. 对算法的性能没有监控，这可能导致使用者的性能损失。

6. 难以理解的错误提示。

7. 一些错误提示虽然可以理解，但内容却是错误的，或包含误导。

在我们的考察中，一共评审了来自几十家机构的几百份程序。所有程序都体现了大致相同的问题，所有机构都在差不多的程度上忽略工具的使用。至少有 75%的机构，通常使用无格式的 16 进制转储数据(dump)来做调试(debug)。至少有 90%的机构都从来没用过预处理器。程序库倒是渐渐被用上了，但还是有 50%以上的机构没有采用。很少有人生成测试数据，更少有人把测试数据存档，即使是最简单的数据集比较也几乎没人用到。

所谓“被用上的工具”，我们指的不是那些买来以后放在架上，空惹灰尘、浪费金钱的工具。我们说的是对工具的实际使用，即使是功能中的一小部分被用上就算。所以，这种“用上”离真正的专业应用还相差很远，我们的数字也就还有高估的嫌疑。既然我们在培训上作了

投资，那么这又能把什么教给程序员呢？

作为教师的计算机

谈到对昂贵工具的不当使用，很显然，我们不是要教给程序员怎样在编程工作中恰当使用计算机。这个结论显而易见——对于课程，我们应该在计算机上多做——而不是少做——投资。

显而易见吗？似乎如此，除非我们看看课堂上计算机是怎么使用的。实际上，我们根本没有在教学生——计算机替我们负起了这个重担。结果是，我们发现自己同时站在一道篱笆的两边。这个问题的两面都成立：(1)在程序员教育中，计算机没有被充分利用。(2)在程序员教育中，计算机被过度利用了。

让我们考察一下，在一个典型的大学编程课上是怎么使用计算机的。也许，为了理解这种应用方式的本质，最好还是打个比方。假设，我们专门给高中英语课开发出了一套“阅卷判分器”程序，而且让学校通过以下方式使用这个程序：

1. 教师给 50～500 个学生讲上一课，课程主题可任选。

2. 教师布置作业：写一篇文章。

3. 学生写文章，并且要严格遵照规则：不准帮人写，也不许让人帮着写。

4. 我们的计算机程序给文章打分，判分依据是拼写错误和语法错误的多少。

5. 把判了分数的文章发回学生。

你觉得学生能从这里学到什么？

这样一个比方之所以能切中要害，是因为对于学生们到底学到了什么，我们无需枉费猜测。我们都已经知道了这个结果。即使没有用计算机判分，很多高中的英语教师也恰恰是如此上作文课的，其结果自然是极其糟糕的。很多学生学会了拼写；一些学生学会了不出语法错误；但根本上没人从中学到如何交流。（幸运的是，少数学生从其他经历中学到了交流的手段。）

在编程教学中，编译器本身就起到了“阅卷判分器”的作用。而且，由于编程作业要在这个“判分器”中多次往返，直到所有“拼写和语

法”错误都被改正，所以对这些方面[①]的强调比作文课来得还要突出。在很多班上，教授既没有时间，也没有心思实际阅读程序。有时候，连输出结果也没有人看。在这些班上，学生即使交了错误的输出结果，教师也不会帮他们指出来。学生即使交了假的输出结果，教师也永远不会发现。

为了解决这个“输出结果无人关注”的问题，一些老练的学校开发出了“判分器”程序，自动把测试数据输入到学生的程序中，然后根据输出结果判分。当然了，“判分器”提升了“计算机辅助教学”的水平——但是受益者更多的是那些不胜其烦的教师，而不是学生本身。是呀，毕竟很多班上都有上百个编程新手，如果不用上“判分器”，又怎么才能多快好省地应付这一群群的血肉之躯呢？——不谈多快好省，也许连是否能够应付也是个问题。

公正地说，与多数学校现在还在采用的单靠编译器排错的方式相比，“判分器”确实算得上一种进步。但是，还是那个问题，这样能教什么？如果你评审一下这些班上教出来的学生（无论是在课上，还是以后多年的实际工作中）编的代码，你就会明白计算机在编程教学中实际教了什么：

1. 在一个需要批处理作业的环境中[②]，精确地对卡片打孔；
2. 在一个联机环境中[③]，利用文本编辑器避免打字错误；
3. 关键字的拼写；
4. 对该程序员善用的一些词，做到拼写一致；
5. 对特定编程语言句法的一个子集有一定掌握。

但是在这些重要的教学内容之上，计算机——如果按上述方式应用——还教了远为深刻的一课，这一课的内容会被学生牢记，哪怕是到了他们把 WATFIV 句法全都忘了的时候。这就是：

人家教给我们，通过测试来修正程序。

---

① 这些方面：指“拼写和语法”。

② 在“批处理作业环境中”，需要作业卡片作为系统的输入、控制。

③ “联机环境”与上面的“批处理作业环境”相对，是指使用者可以通过终端实时操作的系统。

这是奇怪的一课。让我们听听 Edsger Dijkstra 关于结构化编程的正宗训诫：

> 保证程序的正确性不仅仅是程序外部规范、行为的一个功能，而是与程序的内在结构密切相关。

简而言之，结构化编程是自从汇编语言发明以来最重要的一场编程方法论革命，而计算机教给学生的东西，与结构化编程的主要训诫却完全背道而驰。每一天，在这个国家的每个学校里，计算机都在向成千上万的现任程序员与未来程序员传授这个。

另外还有重要的第二课——这其实是一个“课中之课”①——关于“如何学习”的一课：

> 我们把垃圾扔进计算机，看看会出来些什么——这样我们也就学会了编程。

这一课，与我们青春期的偏见类似，铭心刻骨、难以改变。更重要的是，既然当今个人计算机已经如此普遍，这一课也就越来越早、越来越深地在学生心中留下烙印。那么，我们对下一代程序员还能做些什么呢？

把本属计算机的归给计算机②……

计算机在教学上是有优势的——这一优势主要在教给学生一些计算机相关知识方面。比如对于句法错误来说，讲课很难在大多数学生心中留下哪怕是最淡漠的印象——也许除了一点儿消极影响之外。可是，一个编译器却能耐心地、无情地把语言句法教给一个又一个桀骜不驯的学生。如果运用得当，作为教师的编译器甚至能激发学生去搞懂句法背后的设计原则，虽然他们不得不另找老师学习这些原则。

> 一些工具制造者说，句法和拼写教学无关紧要，因为我们可以让

---

① “课中之课”：原文 meta-lesson 是作者自铸新词。所谓 meta-，也就是“元-”，比如 meta-language（元语言）就是“关于语言的语言”，所以 meta-lesson 也就是“元课程”，关于课程本身的课程。

② “把本属计算机的归给计算机”，原文为“render unto the computer…”，是对《圣经》名言“凯撒的物当归给凯撒；上帝的物当归给上帝”（马太福音 22:21）的戏拟。此处是“让计算机负责适合计算机教学的部分，也应让教师负责适合教师直接指导的部分”的意思。

工具来更正任何错误。我只能部分地同意他们，因为永远不会有哪个系统能够更正所有类型的错误。所以，即使是最出色的程序员，也必须花些时间学习句法、拼写的艰苦课程——不然就会在实际调试时，把几百个小时的时间浪费在着急生气上。

而且，即使是结构化最完善的程序，再用上最纯熟、最仔细的校对技巧，计算机也还有可能漏过两种错误。第一种，简单的校对错误——这种错误像瘟疫一样充斥着最高级的数学杂志，也充斥着最普通的程序。第二种，则是有可能完全误解了问题本身。

尽管一个程序员可能证明，程序正如她（他）所想的那样运转，但她（他）永远也没法证明，她（他）所想的和用户（或用户们）想要的完全吻合。在某种意义上说，其实没有“错误”的程序，只有“不同”的程序。要想证实，是否程序解决了正确的问题，唯一的办法只有让原本提出问题的人考察这个解决方案。

考虑以下典型情况。一个考古学教授正在教一门入门课程，他有一台计算机，能按照随机法则从题库中抽取、打印不同的试卷。这个打印试题的程序是一个计算机科学系的高班学生写的，可是程序有一个毛病。同一份试卷会有同一道题出现两次，甚至三次，而且这种情况还相当频繁。问题的原因也很简单，程序本应该对题库作“无退还抽样”，但却作了“退还抽样”。

教授发现了这个错误，找学生质问。学生却说，程序是可以作无退还抽样的，但那样一来，就会变得“非常低效”。他明白，教授一定不会愿意付出这种效率降低的代价，所以不妨这样解决：比如说，每次打印 14 道题，从而保证肯定有毫无重复的 10 道题。学生答题时，只要告诉他们做“前 10 道没有重复的题目”就成了。

读者肯定会感到奇怪，这个学生程序员考虑选用的都是什么破算法。其实，他的道理从概念上说挺简单。如果决定了采用“无退还抽样”，程序就要生成一个“暂定试卷”，然后再测试一下，看看其中是否包含重复试题。题库里一共有 20 道题，每次抽取 10 道题，所以如果用程序测试这种“暂定试卷”的话，每 30 份试卷才会有 1 份是没有重复题目的！

我们必须留意，才能从这个故事中得出正确的教训。很多计算

机科学系的教授都会批评这个学生对算法全然无知。虽然这样的无知是可悲的，但在“现实世界”中，我们也能学着和它共处。但这个学生对程序员在实际生活中的角色也全然无知，恰恰是这种无知才是让我们没法与之相处的。

教授到底愿意为试卷付出多大代价，这不是程序员能够决定的。他对教授意愿的推测，让他的程序打印出了几千份最后作废的试卷。错过了最后期限，结果还得重新启用原有的试卷生成系统。除非是为了取乐而编程，否则永远轮不到计算机来决定程序是否正确，也轮不到程序员，因为这个决定只有最初提出问题的人有权做出。

那么另一个相关的意见，也就当然是：程序员对程序是否错误也没有哪怕最小的发言权。他甚至也无法从性能观点评判程序是否错误。无论是编程课教授，还是计算机，都不会教给他这样一课。

最终，要成为一个真正的专业程序员，一个人必须学习很多课程。其中每一种课程的习得，都有其自身的特性。在设计编程教学课程时，我们要确保每一种课程都是通过最有效的方式教授的，而不仅仅是让一群学生在整个学期里都有事可做，在结束时不要求退回学费就行了。

如何利用/不用计算机讲授编程课程

那么，我们怎样设计这么一套有效的课程呢？其实到现在为止，主要原则应该很清楚了：

1. 用计算机教授那些只有计算机才能教授的内容。

2. 用人来教授只有人才能教授的内容。

3. 仅仅对于计算机和人能达到同等效果的那些方面，按照经济原则做出选择。

毫无疑问，在一套入门课程中，开始时必须有几次上机的机会，这样才能教授以下方面：

1. 创建程序的大体过程。

2. 计算机的挑剔特性。

3. 人类求精能力与计算机挑剔特性之间的不匹配。

当以上内容（至少是在入门层次）教授完毕，课程就应该转向由真

人教师来教授难度更高的内容。典型的任务可以包括：

1. 教师为了几种特定的教学目的，充当“用户”提出一个问题。
2. 每个学生在纸面做出一种试验解决方案。
3. 每个学生评估另外一人的解决方案。
4. 学生分成小组，往往由教师督导，每组达成一个综合解决方案。
5. 上机试验综合解决方案。
6. 小组之间交流方案，提出正式评审（有时由教师在全班课堂前指导进行）。

在这样一个班上，能给学生提供一份更丰富、更有意义的编程知识食谱，避免了在句法和拼写上浪费时间。因为这些方面都能在正式评审中被顺便订正，也可以在上机时由计算机教授——只有通过评审来教授句法、拼写特别经济、特别高效时，这样做才是合理的。从教师到学生，从学生到学生，传播着关于代码风格、编程语言、算法、设计、工具以及上百种小的编程技巧的知识——这些技巧尤其重要，因为正是它们才把专业程序员和业余爱好者区分开。而且（如果这么说不会显得自以为是的话），偶尔甚至还会有学生把知识传播给老师的情况。

但是“实际经济考虑”怎么办？我们对此也考虑过了。我们对此也做过实验，实验结果甚至超出了我们最大胆的考虑。当然，首要的节省来自以下事实：只有五分之一的学生作业要通过上机完成。即便如此，每个学生学到的内容也远远多于旧有的课程体系，因为每个学生都能看到多种（而不是唯一的）解决问题的途径。

其次，为了产生一个正确程序（不是一个仅仅“能转”的程序），需要的运行次数有了显著的下降。下降的数量级在一定程度上取决于问题的规模，一个典型的数字大概是从平均每人运行 20 次降低到平均每人运行 2 次。实际上，大多数程序都会在第一次上机时就正确运行——而且学生还能做出演示。他们知道，最好要达到以上效果——否则另外的小组就会把这个程序批得一无是处。

第三，一旦注意力集中在设计而不是语法和拼写上，往往程序就能够运行得更有效率。上面考古学教授的例子里，30 倍的效率损失或许也并非出格，不过让我们假设只会发生 2 倍的损失。把上面 3 项中提到的数字相乘，则总机时损失了 100 倍（5×10×2）。毫无疑问，这

已经足够满足“实际经济考虑”，允许我们按照教学效果（而不是教学成本）来选择编程工具。

但是以上方法的收益还不仅仅停留在机时利用上。当我们想要教一种语言，却没有编译器时，甚至当连编译器还没有发明，课程也没有出现时，我们都可以成功应用这种方法。因为，如果程序在第一时间能够成功运行的概率非常高，那么在很多情况下，实际运行该程序其实也没有多大教学效果。所以此时甚至可以摆脱计算机，至少是可以控制计算机导致的种种稀奇古怪的变数，这就能让我们对教学计划更有信心，随着学期的进行稳步完成计划。确实，一旦我们从机器的约束下解放出来，课程的经济考虑就主要取决于有现成可用的教师和教室。而只要渡过了最初的入门阶段，任何一组程序员都能按照这样的方法自我教学，无论是脱产还是不脱产。

实际上，不脱产培训的经济情况与学院中的一般假设完全相反。学生很可能比老师工资更高，如果考虑班级整体，则肯定是如此。显然，如果把这些专业程序员全体送到大学校园里听编程课程，肯定不如用更少的时间通过互相评审代码学习编程来得划算。

如果专业教师不想踏进失业线，也就必须面对这样一种隐含的挑战。要是有合适的人引导，一个程序员小组能够通过评审技术大大提高学习效率。引导者能够帮助他们把从学费、机时费上节省下来的钱用在更有用的地方，比如，安排多次测试运行，这样可以进行设计、算法上的比较——而这种比较单靠凭空分析是很难完成的。

引导者可以让他们在同样时间内处理更多问题，或者探讨同一问题的多种解决思路。这样的课程，能够推着学员们在设计的路上越走越远，而不是没完没了地摆弄语言的细枝末节。学员们也能评估各种工具的实际用途，这样也就有可能给我们在工具上的亿万投资带来回报。但是对于这样一种教学环境而言，教师必须费力奔跑，才能保证自己一直在队伍前面。

有些教师认为自己的工作就像 Olivier[①] 在一大排昏睡的学生面

① Laurence Olivier，国内有译为劳伦斯·奥立弗，1907—1989，英国著名演员，曾出演经典影片《哈姆雷特》(1949)。

前表演哈姆雷特，上述新环境可能不适合这些老派的教师。有些计算机操作员凭着闭目塞听、一味用功也能在通常的班级中拿到很多"A"[①]，上述环境也不适合这样的学生。不过，如果真的能够促使这几类人放弃编程职业的话，那也算是这个教学体系带来的另一项收益吧。

一段未来的历史

如果我们考察一下其他高科技领域，比如电气工程、机械工具、电话、蒸汽、印刷的发展，就能发现计算机行业并非独一无二。我们也许能够更快地通过一些阶段，但这些阶段终究还是需要经过的。我们的第一个四分之一世纪步伐很快，几乎完全服从于技术的推动。硬件销售员是我们这个狭小世界的始和终[②]。任何不能促进硬件销售的东西，都被技术丛林的生存竞争抛在了后面。

在其后的几年中，由于 IBM 的"解绑政策"[③]，销售经理们逐渐明白了软件也是一种产品，就像硬件一样——但软件有一个甚至比硬件还巨大的潜在市场。软件热现在正处于青春期，它正在以更快的步伐重复着硬件的历史。

但今天，在对硬件和软件做出了大概 2000 亿美元的投资之后，成绩相对来说却还很小。与什么相对？如果历史有任何指导作用的话，当然就是与将要发生的未来相对了。我们现在还缺乏任何真正意义上的编程行业，这也可以当作衡量我们的不成熟度的一种指标。今天程序员的职业生涯选择是"往上爬，往外走"。大多数程序员在掌握了语法和拼写之后，就学无可学，所以如果正好有一个刚刚完成培训的新手只需要很低的工资，对于公司来说，从经济角度考虑也就没有太大理由留住工资较高的"有经验的"程序员。

而且，如果哪个人超出了这种初步培训的水平，管理层中也没有

---

① A 是最高分数。

② 始和终，原文为 alpha and omega，分别是希腊字母表的首尾。见《圣经》启示录 1:8："I am Alpha and Omega, the beginning and the ending."

③ 解绑政策，指不再绑定销售，而是对不同产品分别交易。

人能够认识到这样一位专业人才有多大价值。毕竟，经理们也都上过一回“编程课”。他们知道编程是非专业的、浅薄的、难以管理的。他们知道在培训上花钱是浪费，还不如投资在某些新硬件，或者某种声称能取代若干程序员的软件工具上。

我们所有人都被一种蛊惑人心的销售口号催眠了，这口号里包括漏洞百出的“实际经济考虑”呀，想人非非的“现实世界情况”呀之类的货色。我们在工具上花费了上十亿美元，可是在弄清“如何培养使用工具的专业技术领导者”这个问题上，却从来一毛不拔。我们在“学校教育”上花费了上百万美元，但从来就忽视真正的学习。结果呢，我们的工具躺在架子上，被误解、被闲置。我们的系统似乎开销太大，但是每次开会，那句销售口号都会被一再重复——再买更多的东西！我们的系统不让人满意，但我们听到的却是“人们不理解计算机的挑剔特性”——下一代就能解决所有这些问题！

当我们从青春期的玩具爱好中长大成人，开始培养出成年人兴趣时，下一代也就要登场了。到那时，我们也就会像其他技术那样，开始掌握社会的、心理的力量——这些力量才是成功技术背后的真正动力，而且也是技术存在的首要理由。作为其中的一个重要起步措施，人们会把计算机培训从计算机手中解救出来，然后也许会把它放进人们的大脑中。

训练随机应变的能力

我有一个很令人振奋的通信伙伴，叫 Jo Edkins，他最近又给我写了一封信，指出了以下问题——这问题对于今天的很多人来说都挺有意思：

> 在我的日常工作中，培训是一个非常重要的部分；眼下，我正在把一些毕业生培训成 COBOL 程序员……现在学校和大学都有了各式各样的形状不一、大小不一、构造不一、可靠程度不一的计算机。最常用的语言是 BASIC，FORTRAN 排第二，PASCAL 和 ALGOL 也挺时兴。16 个毕业生中，只有 2 人从来没有写过任何计算机程序。拿这跟以前比比！但是只有 3 人写过 COBOL 程序，没人写过 PL/I。本次教学的目的，是让他们“适应计算机世界”，可是他们觉得自己的知识简直是一道障碍、一个缺陷、一种混乱。那些从来没有计算机经验的人，似乎能够正常地接受概念，而其他人则要把新概念放回到已经学过的旧概念中去。当然了，我们尽量调和他们各种不同的知识，但那样你就得跟 BASIC，FORTRAN，ALGOL，PASCAL，APL 等语言同时打交道！我自己都不懂最后那两种呢！
>
> 我担心的是，其中有好几个人都难以理解“数据”和“文件”的概念。很多人抱怨说：计算机工业似乎更想要有经验的程序员，为什么不用那些“学过”计算机的人呢？这是因为在培训这种人时，不仅要进行必要的学习，还得先让他们忘了此前学的那些知识，可是大多数公司都付不起这么大的代价。我并不是要责怪这些毕业生本人（我可不敢这么干——他们会剥了我的皮）。在学校里，他们为了学习计算机知识竭尽全力，可是到了公司，为了学习不同的概念，还要首先忘掉原来的知识。
>
> 你对此怎么看？

接到这封信，我不由得想起了 20 年前对这类问题的一个解决办法。Herb Leeds 和我写了一本极其成功的书，《计算机编程基础》

(Computer Programming Fundamentals)，内容是关于汇编语言编程的。当要推出该书第二版时，出版商问我们能不能写点儿“关于 FORTRAN 的内容”，因为它在当时是很有前途的语言。想到要屈尊纡贵，在“最好的语言”——汇编语言——之外教授其他语言，我们两人都不由得心生怯意，可我们最后还是想到了一种不让自己双手生锈的办法。我们对比介绍 FORTRAN 和汇编语言，每一个例题，都用两种语言分别做出答案。我觉得，当时我们私心指望，任何人看了这两种解决方法后，都会理所当然地选择汇编语言，而不是那个异端的 FORTRAN。

写那本书的经历让我们学到了很多东西——我把这些经验教训带到了课堂教学当中。最后，我们的那种思路发展成了一种相当完善的教学方法——这种方法不仅能够避免上面 Jo 的问题，另外还有几种额外的大优势。

这一方法的本质，就是不仅仅教授一种语言，而是教授两种判然有别的语言。最早我们教的是汇编和 FORTRAN，但后来还试过很多其他的组合，而且效果良好。最后，我固定在 APL 和 PL/I 这一对上，因为它们之间的对比再大不过了。编程领域的大多数重要概念都可以通过这两种语言图解出来。在连 APL 和 PL/I 都没法描绘一个概念的时候，至少它们让学生清楚地理解了一个道理：没有任何一种语言能成为编程的“唯一”途径。

从一开始，每一个课堂练习都要用两种语言完成。做完了练习，学生还要附加一个讨论：为什么这两种语言编写的程序会有不同。通常，在写讨论的时候，学生就能自己“发现”编程的新概念，而这些概念在两种语言中都没有出现。

每一次练习后，每个学生都要评审另外一人的解答。通过评审，学生们不仅学到了语言，而且也学到了多种编程风格。

让一个学生做课堂演示，然后邀请其他同学给出该同学的解决方案的变体，或者完全不同的解决方案。如果人们光盯着自己的工作成果，大多数人都只能缓慢地、少量地学习；但是，如果参照别人的成果，大家就能快速、深刻地学习。

当我们在大学里使用这种“双语”教学法时，也就避免了 Jo 提到

的那种“语言沙文主义”。人很容易学习第一语言。你要是不信这个的话，可以去听听中国 3 岁小孩儿滔滔不绝的说话[①]。学第三门语言也是很容易的，第四门、第五门简直就是小菜一碟。但是第二语言却能要了我们的命。但是，如果我们采用了这种双语策略，那也就没有什么第二语言了，这个问题也就自动取消了。

能够在商业环境中采用这种双语策略吗？是的，如果学生们都是新手的话。但是，对那些已经学过一门语言的人该怎么办呢？要做的第一步，就是让这些学生相信，他们已经懂得的那些知识还不足以应付新的环境。

有很多种办法能达到这个效果，但我自己最偏爱的是用一种挑战的办法。我们这么说：一个真正专业的程序员，应该能够很快学会一门新的编程语言，通过两周的自学就能出活儿。我们的双语学生在工作中就能达到这个程度。如果你的学生也确实能做到这样，那么就有理由不采用正式的课堂教学（这种教学方式也正是 Jo 的问题所在）。如果他们不能达到这个程度——可以让他们试一试，失败了就知道了——那么当然也就应该进行双语教学。

要想成功地使用这种策略，教师自己最好能够熟练使用多种语言。事实上，教师最好精通学生们可能知道的所有语言。（没错，Jo，找本 APL 书看看吧，再找一本 PASCAL 的。）不然，学生可能会炫耀自己在某种语言上的“高手”水准，以此对你构成威胁。不过一个老练的教师能够不受这种威胁，反而把个别学生的既有知识变成大家的优势。首先承认这一点：编程是一件难事。然后向学生们解释：程序员尽可能地需要一切帮助，所以任何编程语言带来的洞见，对课堂教学都是极为有益的。鼓励他们在课堂上分享不同的思维方式——这些思维方式可能是学习其他语言时获得的。每次他们这样做时，都及时指出掌握了多种解决方法有多么重要。

一开始，一些学生可能会抱怨，在 COBOL 里面怎么没有 BASIC 的一些“好用的”功能呀。这种情况下，就要向他们保证，虽然 BASIC

---

① 西方人认为汉语很难学，所以连中国 3 岁小孩儿都能滔滔不绝地说汉语，这对他们来说也是很了不起、很需要解释的事情。

能够给 COBOL 带来洞见，但是学生们来这里是要学会成为“专业人士”的。专业程序员用项目要求的语言写程序，而不是把语言的差异作为借口一味抱怨。一旦发生这种情况，我总会用那句谚语来挑战学生：工匠蹩脚怨工具。

教师首先要树立的，就是“专业”的形象。你不是在教学生们 COBOL，你是在教他们成为以后使用 COBOL 工作的“专业程序员”。如果他们已经是专业人士了，那么这种“第二语言综合症”根本就不会存在，教给他们必要的 COBOL 知识简直易如反掌。

从业余爱好者到专业人士的这种转变，就好比是一束光亮，教师借此也能发现很多解决其他问题的办法，比如“数据”、“文件”概念的问题。学生们在告别业余身份的同时，也在进行着从小系统过渡到大系统的转变。这个差别最容易从数据设计和文件的概念上表现出来，但也表现在其他很多方面。比如，存在着不同开发者之间的程序可读性、可交流性问题。很多“学院派”语言恰恰是因为这些方面的欠缺，所以没法用于构建大型系统。它们无法处理不同来源的大量数据。当系统要由很多人共同完成时，这些语言也会捉襟见肘。

如果你的班上有几个学生学过一种“学院派”语言（比如 BASIC 或 PASCAL），你可以让他们用一个相对大型的案例来进行对比实验。拿一个被投入实际使用的程序，再让他们翻译成自己最喜欢的语言。他们会从中学到很多事情。最重要的是，他们会懂得，一个专业程序员必须比业余爱好者更能够随机应变。

想打板球的蟋蟀[①]：一个寓言

一只蟋蟀，年纪尚轻，挺有干劲儿，有天晚上坐在炉边[②]，听一些人谈论国际板球锦标赛中英格兰对澳大利亚的比赛。他呢，不明白什么叫英格兰，更甭提澳大利亚了，因为地理课他总是开小差逃掉，但是听到“板球”一词在谈话中晃来晃去，他也就很是用心地听了一回。他推断出，板球是某种很受尊崇的活动，所以他也就立刻决定，这板球必是他的宿命。他还听说，那场伟大的比赛将于下周四在博内茅斯举行，虽然他懂得什么叫周四，但“博内茅斯”就全然不晓得了，所以他马上动身去咨询地理教授。

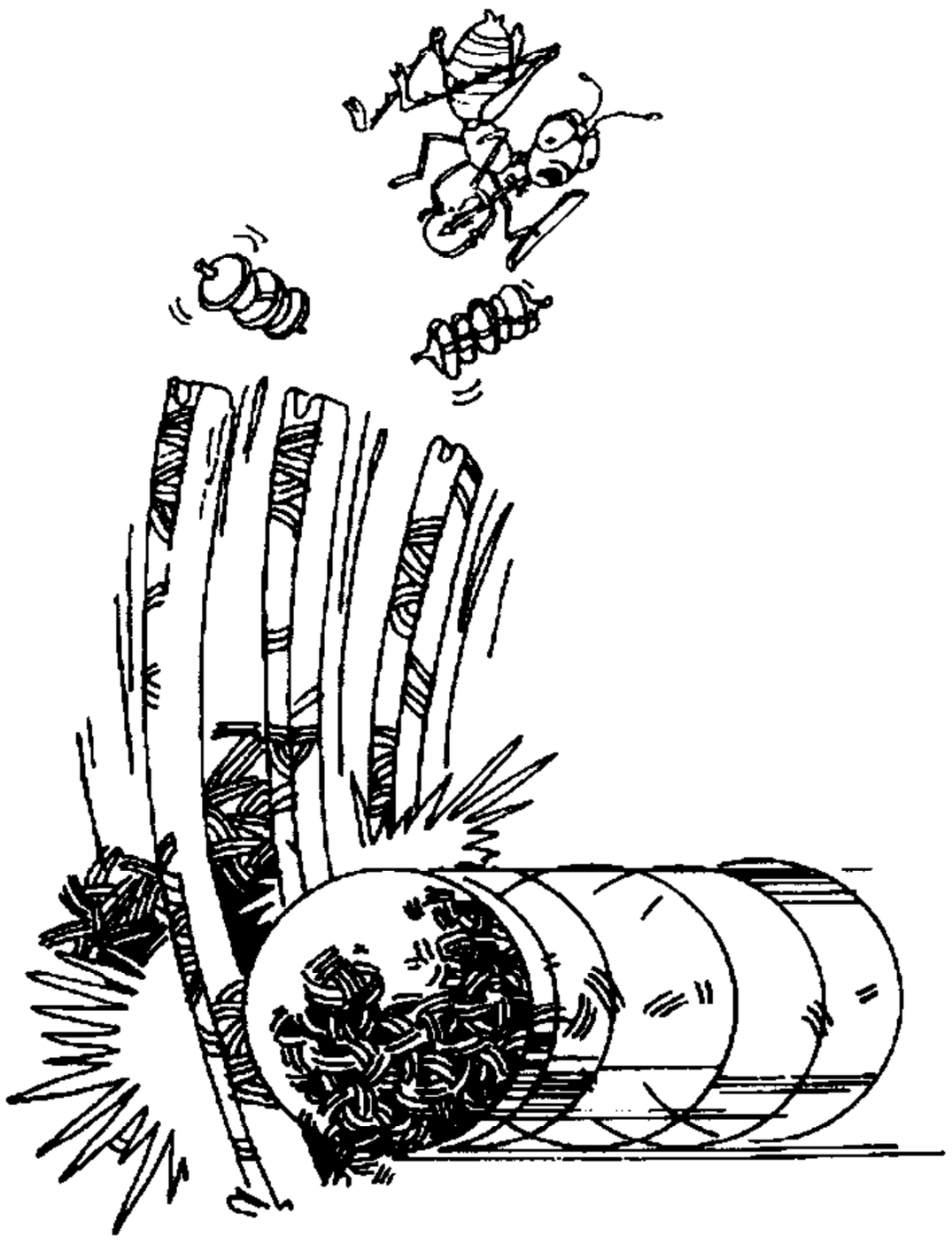
自己最差的学生眼中居然闪出了兴趣的火花，地理教授看到这个也很高兴，但是当他得知蟋蟀对地理感兴趣的原因，往最好里说他也是失望透了。“看这儿，小伙子，”他说，声调里带着那种抹不掉的、跟学生说惯了话的、盛气凌人的劲儿——这种腔调也是蟋蟀不爱上学的原因之一。“看这儿。如果你以为你碰巧名叫‘蟋蟀’就天生会打板球，那你可就犯了一个严重的学术错误。虽然‘板球’(cricket)一词的词源还不能确知，但它很可能是‘三柱门’(wicket)这个词的一种讹误形式。而你的名字呢，却不是一个词源学(etymology)问题，倒是一个昆虫学(entomology)问题。由此可知，你对打板球的渴望纯属一个语义错误，如果你再受点儿教育，这个念头就肯定会无疾而终。为什么不呆在这儿呢？你在这儿有权获得免费教育呀。这肯定对你更有好处，也比在球场上冒险花费更少。”

可是蟋蟀对这一类的冬烘学究一点儿耐心也没有。所以，从教授那里挤出了他渴望的信息后，他马上出发，前去观看下周四的盛事(这么早走是因为蟋蟀走路不快)。确实，当他抵达赛场，比赛已经开始了，澳大利亚正在击球。蟋蟀却没有在观众中找座位坐下，反而立刻

---

① 板球、蟋蟀在英语里都是“cricket”。作者的文字游戏。

② 英美诗文似乎很喜欢把蟋蟀和壁炉联系在一起的。比如狄更生甚至做过一篇《壁炉上的蟋蟀》(*The Cricket on the Hearth*)。



动身跑到了赛场中央，那里有不少木桩子，立在场地中间。因为他体格微小，所以跑到木桩上也没有干扰比赛，但也因为他身形太细，赛况也就看不太明白。他对板球根本一无所知——尤其是连什么叫“三柱门”都不懂——他决定爬到木桩顶上，以便看得更清楚些。正在他坐在三柱门顶端的时候，英格兰的投球手把澳大利亚最好的击球手杀出了局[①]。这时三柱门滚下了木桩，不走运的蟋蟀也被带着重重地摔在了地上，同时人群中响起一片叫好声。蟋蟀当然觉得这“好”是给他叫的，但他也没陶醉太久，因为膝盖破裂的疼痛马上就让他晕了过去。当他醒过来，天都黑了，当日的比赛结束了。在漫长、寒冷、痛苦的归途上，蟋蟀自己这么想着：“当然了，人群都为我的果敢一摔叫好，这是非常美妙的感受，但是我想还是放弃板球比赛算了，既然我这就已经达到成就的顶峰了。”

教训：“狠摔学校”的学费很贵，但有些人就是不能从任何廉价的学校里学到东西。

---

① 板球规则中有这样一条：如果柱子被撞倒，击球者要出局，这叫一个“stump”。

想打棒球的蟋蟀：一个寓言

蟋蟀在板球比赛里受了伤，康复期无事可做，只好去听地理课。课上他学到了美国，那边的国球可不是板球，反倒是板球的一个变种，叫做棒球。他倒没有任何学术偏见，也不顾自己之前吃的苦头，立刻就产生了一种对远方的憧憬；同时，忘了膝盖的疼痛，却记起了球场观众的轰动，他马上动身前往美国——就在他刚刚恢复跳跃能力之后。

“我相信，我将为勇士队比赛，”他在漫长的旅途中自己这么琢磨，“因为在英格兰的板球场上，我可是展现了自己的勇气。另一方面呢，我得当心红雀队，因为谁都知道红雀遇上蟋蟀会干什么。”这么想着，终于他发现自己到了勇士体育场，这是一个下午了，场内正在进行勇士队和巨人队的比赛。

他还是渴望加入到比赛中间，因此不在大看台上找座位，却直接往本垒跑去。“这一回，”他想，“我得离木桩远点儿，离地近一点儿。虽然可能没法看得太清楚，但是所谓‘先知先戒备’，有了上回的经验，还是谨慎些的好。”这样，他专门挑了一个最平坦的座位，这碰巧就是本垒。在那儿他待得挺安生，直到第九局末尾，双方还是平局，这时，一个跑垒员想从二垒跑上本垒得分。胜负特别接近：在一阵尘土的云雾中，球、捕手、跑垒员几乎同时到了本垒上，裁判员喊道“安打”，顿时人群雷动。几个小时之后，那喊声还在蟋蟀的耳朵里回荡，那时他躺在空空的体育场当中、本垒的上面——这一回两个膝盖都破裂了。“我选择了一种艰苦的生活，”他想，“但是说到底，有了这样的喝彩，难道还不值得吗？”

教训：经验不一定能教会人懂得任何东西。

第3章

为什么程序员如此做事

个人化学[①]和健康身体

很多数据处理领域的专业人士，渴望成为管理人员，但企业在任用管理层时却往往不考虑他们；每当此时，他们都会大惑不解。也许，问题出在他们的“个人化学”上。

我有一份剪报，上面是 J. Gerald Simmons 的一段话，此人在一家叫“汉迪集团”的猎头公司担任总裁。（有些人喜欢把名字缩写，而把中名完整地写出来，好比 J. Edgar Hoover 或是 G. Marvin Weinberg[②]，这里你大可以放下对这种人的偏见，听听此人到底说了什么。）Simmons 说，在从其他方面都不相上下的候选者中挑选人才时，领导者应该重视“个人化学”。所谓“个人化学”，其成分包括：外貌，个性，风格，口齿清晰度，精力，态度，是否好动脑，是否从容镇定，火花[③]，兴趣广度，另外就是某种领袖气韵。一下子要求这么多品质，简直像童

---

① 化学：原文 chemistry，既指一般意义上的“化学”，也另有“神秘的过程”、“融洽的关系”等比喻意。本文的“个人化学”，指的是专业人士个人应具备的、某种不乏神秘的气质或素质。此处保留字面意思，望读者体察。

② J. Edgar Hoover，1924 至 1972 年任联邦调查局局长，一个令人生畏的名字。G. Marvin Weinberg，本书作者本人。英美姓名惯常的写法，是缩写中名（middle name），而保留名字（given name），所以上面的写法显得特别。

③ 火花：原文 sparkle，本义为“火花，火星儿”，它既是“活力”、“光彩”的隐喻，又可指该人眼神中确实闪出的“火花儿”。故保留该词本义。

子军纲领了，不过咱们还别急着抛下这个人，再听听他还说了什么。

Simmons 称，个人化学是能够培养的——这给想成为管理者的那一大帮人留下了希望的余地。不过，他关于如何培养个人化学的建议，听上去往往像一枚伪币发出的似是而非的声音。举点儿例子：

外貌：痴肥、干瘦都要扣分。（建议：视情况减肥或增重。）

精力、动力、野心：培养自己，平时走路大步、快步，仪表清新，从话音中体现出过人的健康。

从容镇定：爱咬指甲的，爱捻头发的，爱用脚打拍子的，爱一根续一根地抽烟的，爱面部抽动的那些人，很少能通过面试。（建议：改掉那些让人分神的习惯。）

领袖气韵：身形笔挺，头部高昂，风度和善，直视对方，再加上适度的自信——这些体现出了领袖的素质。

当我读到这些要诀，我想起了总在念叨“坐直了”的妈妈和总在怒吼“别嚼口香糖”的老师。以上建议，在我看来简直不仅无益，而且有害。要是一个人老想着“我是不是正散发着领袖气韵呢?”这人也就很容易咬指甲、鼓嘴巴，甚至干脆就搞坏了脑子。

可是，毫无疑问，Simmons 说的是对的。我们都愿意身边的人相貌宜人，有活力而又很镇静，并且待人和善。问题是，如果我们不能做到这些，那又该怎么办。如果我们对生活感到颓丧，那么旁人的帮助最容易的莫过于这样一条建议：“别灰心。”要么，你太胖了？那么看看这条建议如何：“别吃太多。” Ambrose Bierce① 把“建议”定义为“面值最小的流通硬币”。建议，在价格上往往是免费的——而且在价值上也很低菲②。以我所见，真正的幸福和成功只需要寥寥几点要则，而种种关于“个人化学”或“成功秘诀”的建议，多数都反而遮蔽了这几点要则。伯特兰·罗素③在他的经典著作《征服幸福》(Conquest of Happiness)中已经写过这些了，我每年都要试着重读一回这本书，以确保自己走在正道上。我说“试着”，是因为我手头的这本书常常被人借走，

---

① Ambrose Bierce：美国作家，著名的幽默作品《魔鬼词典》的作者。

② 指提建议的人一般无需代价，而建议对接受者也帮助不大。

③ Bertrand Russell：罗素，英国哲学家。

而借者往往成心不还。眼下我在家里、办公室里就都找不到它，不过，不看原书我也记得罗素讲的首要原则。

罗素——他可能是我们这个世纪最伟大的哲学家——说，要是一个人足够健康，不用其他办法，健康本身就“制造”幸福。反过来说也一样，身体要是不健康，其他“成功秘诀”都不会管用。想想 Simmons 的“个人化学”：要是你健康，你的外表也就不会痴肥或干瘦。你的步伐自然就很快。不用喷那包含一千种念不出名字的成分的香水，你的外貌和体味都一样清新。你也不会咬指甲，捻头发，用脚打拍子，或是敲手指头。你也不太可能会经常抽动面部，也许根本就不抽烟。你会坐得笔直，也不会显出午餐消化不良的样子——因为你确实消化得很好。

一个很有意思的巧合是，关于健康的现代观点是：健康在很大程度上与人体化学有关（你吃什么你就是什么），还与人的行为有关（你做什么事你就是什么）。如果这个观点成立，那么“个人化学”也许就跟“实在的”物理、化学差不多了——讲的似乎都是最直白的道理。不过，我还是放下说教，先来讲几个故事。

咱们都知道那个叫 Gary Gulper[①] 的程序员，特别专注——专注于用垃圾食品填满他的胖脸。他觉得，在午餐时间都不停手地工作，只是从自动售货机买个点心糖和一筒可乐充饥，这就能给老板留下深刻印象了。还有 Susan Sitter[②]——她的成功秘诀是一天工作 16 小时，盼着如此就能引起老板的注意（当然还要在她死于缺乏身体锻炼，或是因臀部肿块失去活动能力之前）。

我对这一类人非常熟知，因为我自己就具备所有这些品质。我们共有的一个特点，就是一种对工作的诚挚的专注——如此专注，以至于在需要时，我们会虐待自己的身体以完成工作。当然了，诚挚的专注本身无可厚非，但如果极端到了损害身体，妨害我们今后更有效地工作的地步，那就大为不当了。如果我们加班又加班，却因体质下降

---

① Gary Gulper：作者仿照好莱坞明星 Gary Cooper 虚构的名字。Gulper 有“大吃客”的意思。

② Susan Sitter：同样是作者虚构的名字。这里 Sitter 也就是“长坐客”。

导致工作质量受损，那又有什么意义呢？如果我们吃一顿体贴的午餐，就能让头脑获得养分，身体获得放松，从而快速完成任务，那么我们为什么还要省去这顿午饭，仅凭点心糖的能量挣扎在这个任务上呢？

成了，咱们还是说老实话吧。仅仅一次、仅仅一会儿虐待自己的身体，这甚至还挺有趣。谁能拒绝一个糖果宴、一次饮料狂欢，或者说，一次放纵自己整夜纵酒狂欢的机会？所以，虽然我们力称我们是为了专注工作才这么做的，其实这不过是一个完美的借口，用以满足我们内心隐秘的放纵欲望。而且呢，一点点过火儿不会伤害任何人，对吧？而且有时候工作确实需要全力以赴、自我牺牲的投入。

好，现在让我们引出教训。很多专业人士——包括我自己——都养成了为工作牺牲身体的习惯。他们养成这个习惯时，人还年轻，身体也还皮实。但终有一天，他们发现自己的身体不像从前那样能够迅速、轻易地复原了。而当他们发现这一点时，往往已经太晚了，因为以下原因：

1. 习惯已经根深蒂固，很难改掉。

2. 身体上的亏欠已经持续了很长时间，直到情况足够恶化，他们自己才觉察到。可是旁人早就觉察到了，所以他们的事业已经受到了损害。

3. 他们年岁已经太大了，很难轻易地再养成好习惯。

4. 他们身体上的各项问题很可能会互相促进，而我们都知道，如果系统中同时有两个交叉影响的错误，这个情况会有多难对付。

5. 即使他们立刻开始放下过多的工作负担，恢复正常饮食、锻炼，也不会马上就产生成效——这也就意味着，他们的工作效率不会马上提升，可是工作业绩却会有明显的下降，因而有可能失去工作或职位。

上述言论听起来只不过是教训。20 年前，人家跟我说这些的时候，我当然就把这当成教训。现在想来，所谓“教训”，也就是防止为了一时快乐牺牲长期幸福的一种智慧。因此说到健康，我们还真需要几点简单有效的“教训”，因为当我们的健康受损，大脑就是第一个开始衰败的器官。到了那时，不仅我们身体不适，而且甚至没法集中脑力面对这个“不适”的问题，因为大脑也受困于这种健康滑坡，自身难保。

当大脑衰败的时候，大脑的主人总是最后一个才觉察到这个衰败的人。可是有时候教训不管用，因为教训会凝结成规矩，而规矩的初衷又常常被大家忘记。在这种意义上，教训简直就像数据处理中的一些“标准”。我认为，大家不应该不考虑理由就盲从这些“标准”，无论是“外表要健康！”，还是“吃菠菜！”，或是“避免用 GO TO 语句 ！”。缺乏理由的规矩只关注事物的外表，忽略了实质。我从不建议别人从外表做起改善“个人化学”，也就是出于这个原因。

相反地，我提倡从深层次考虑问题。在所有表面细节之下，其实只有一个简单的论断。很多专业人士过于敬业，以至于损害了健康——而健康才是“个人化学”的实质。只要明白了上述概念，其实你能发现很多条通往健康的路；可我不会为你个人作诊断的。我不是那种医生。我是给企业、机构治病的医生，对于一个机构而言，可以采取一些方法，为了员工健康建设一种良性的气候。也正出于此，我建议数据处理业的经理们采纳以下建议，为员工们做出榜样：

1. 在**大多数**时候按时上下班，不加班。

2. 拿出时间好好吃饭，不要弄出一个不吃/匆忙吃中饭（或晚饭，如果你违背了第一条的话）的坏榜样。

3. 永远不奖励那些在额外时间工作、忽略吃饭的人；相反，奖励那些井然有序，在正常工作时间好好完成正常工作的人。

“为工作而工作”、“为了别人看着表现好而工作”，我们很容易落入这样的俗套中。但如果你时时提醒自己身体才是第一号工作装备，你就能打破这些俗套。如果身体垮了，国王的全班人马也没法救起它，让它运转如新。而我甚至怀疑 IBM 的外派工程师能对此有什么帮助。①

---

① 这是作者的俏皮话：“国王的全班人马”是英国童谣中一种著名的“天降神兵”。Lewis Carroll 在《艾丽丝镜中奇遇记》里引用过这个：“矮梯胖梯坐在墙头，矮梯胖梯摔个跟头。国王的全班人马，也没能把矮梯胖梯扶回墙头。”作者又拉上了现代的“天降神兵”——IBM 现场工程师，可是对于程序员受损的健康来说，他们的本事也都无能为力吧。

为了应变，程序员需要什么

温伯格双胞胎定律

从前，我写过一本书，题目是《程序开发心理学》。虽然这本书让我发了财，成了名，但题目中的“心理学”一词却让好些人把我当成了心理学家。我最好还是更正一下这个糟糕的错误。

我没获得过任何心理学家的资格认证。我没有心理学学位。我没法治愈一个抑郁症患者，甚至都不能治愈一个轻微的偷窃癖病例。其实呢，在我上大学的时候，我成心避开任何心理学课程，也不让人看见我和心理学教授们在一起。

开诚布公地说，我一直怀疑，整个心理学领域是由50%的错误和50%的虚假构成的。可是，随着年龄的成熟，我开始越来越尊重一些心理学家的工作了。到了最后，我发现他们为了在公众面前推销自己，确实做了很困难的工作，我也学会了欣赏他们的努力。毕竟，社会上能人有的是，就连每一个酒吧招待，用不上任何证书、学位、课程、书籍、培训，也都天生就是人类行为方面的专家了。

其实呢，人类的多数行为都非常容易预测——容易得简直可笑。气象学家教给我们，预测明天的天气有一个简单有效的办法，在2/3的情况下都是正确的：只消说明天的天气和今天一样就是了。这样一来，每个人都算得上气象问题的专家——66%的正确率。请想象一下，如果我们也能轻易地预测99%的人类行为，那么会造就多少心理学的专家呢——只要一条简单的定律就足够，这就是温伯格双胞胎定律。

你的心理学教授从没教给你温伯格双胞胎定律，不过你还得原谅他们。没人愿意泄漏专业秘密。要是你知道心理学课程只能覆盖1%的例外情况，剩下的99%都可以轻轻松松在一分钟内掌握，那你还会去上心理学课程吗？

就像许许多多真正伟大的定律一样，我这条定律也有一个极为卑

微的起源。那是一个冬日阴沉的交通高峰期间，我们温伯格家的两口子坐在 44 路公共汽车上，往百老汇方向驶去，这时一位憔悴但不失漂亮的年轻女子，拉扯着 8 个小孩上了车。

“车费要多少钱?”她问司机。

“大人 35 美分，5 岁以下的小孩免票。”

“那好，”她说，倒腾了一下手上抱着的两个最小的孩子中的一个，这样才能取出钱包。她往钱箱里投了两枚硬币，然后就带着那一大班人往过道那边走去。

“等一下，女士!”司机发话了——也就纽约公车司机才有这个权威。“你不会是指望我相信，所有这 8 个孩子都在 5 岁以下吧?”

“可确实如此呀，”她生气地说，“这俩 4 岁，这些女孩儿 3 岁，刚走路的这些 2 岁，这两个小的 1 岁。”

司机又惊又愧。“天啊，女士，真抱歉。你一直就在生双胞胎吗?”

“噢，那倒不是，”她说，一边还把一缕棕色的秀发捋直了。“绝大多数时间，我们什么也不生。”

就在这一刻，我们两人脑子里突然灵光一现。绝大多数时间，对于世界上的绝大多数系统，都不会有任何重要情况发生。没错儿，如果你比较这一分钟和下一分钟，绝大多数情况下你会发现二者几乎完全相同。对于各种各样的系统中的绝大多数来说，你能做出的最佳预测就是，下一刻它们的工作情况就和上一刻完全一样。

不消说，我们兴高采烈。我们的这个发现，简直是一切定律里最重要的一条，超过了所有学科中(数学啦，甚至哲学啦)的发现，我们的英名，当然也就因这一发现而永垂青史。可也正在此时，一阵忧郁也随着高兴劲儿，突然降临下来。我们的定律如此重要，当然配得上“温伯格定律”这个名字，可是此前已经有了一条关于双胞胎出生频率的“温伯格定律”[①]了！这样一来，新的定律就不能合适地命名为“温伯格双胞胎定律”，因为这个名字已经铭刻在科学史上了。

我们伤心透了，在此后的多年里，都没有把我们的定律公布给别人。当我们最终振作起勇气，一吐为快时，朋友却笑了。“嘿，别傻

① 似乎指遗传学上著名的哈代-温伯格定律。

了，”他们责备我们，“这样的情况早有先例，想想牛顿第一运动定律和牛顿第二运动定律不就得了。”

这样一来，我们升入科学圣殿的机会又重现了。在这圣殿的大理石宝座上，刻着温伯格第一双胞胎定律和温伯格第二双胞胎定律。抛开奇怪的数学表达，用一般系统的术语来讲，第一双胞胎定律说的是：

在所有的分娩中，双胞胎出生是比较罕见的，三胞胎更罕见。

第二定律——也就是我们的定律——说的是：

虽然双胞胎出生在所有的分娩中算得上罕见，但分娩本身就远比不分娩要罕见得多。

就像牛顿的运动定律一样，我们这条第二定律，虽然名字里的“第二”有种种不利的暗示，但是——如果一定让我们来说的话——却是两条定律中远为重要的一个。它重要，因为它确实预测了99%的人类行为，并且对其他各种系统的行为也能作出相同精度的预测。考虑一个标准的美国女子，她大概活70.7年，生2.1个孩子。这意味着，随机抽取她生命中的任何一天，她分娩的概率小于0.01%。（当然，对于男子来说，这个几率还会更小。）另一方面，如果她在某一天确实分娩的话，那么生出双胞胎，或多胞胎的机会是1.5%左右。所以，第二定律的预测能力高于第一定律100倍。

不过，出于某种原因，我们并没有像自己期待的那样有名。似乎很多人都认为他们已经知道这条定律了，虽然没人能找出哪本知名杂志上对此有过引用。也许这是因为人们对一条定律期望过高了。人们不止要一条普适的定律，还希望对社会中的连续性做出解释。这个可不像看上去那么简单。如果我们想解释一下，为什么绝大多数时间里什么都不会发生，这个要花整整一本书，也许接着还得要好多本才行。

Dani Weinberg在我们的论著《论稳定系统设计》(On the Design of Stable Systems)中完成了这个任务，所以这里我也就不做重复了。我想考察的是，很多人想改进专业程序员开发计算机程序的方式，对于这些人的工作，“社会连续性”具有重要意义。

第一惯性定律

一旦你明白了，“不变”是一条重要规律，“变化”则是一种罕见的例外，你就会开始考虑了：既然“不变”的情况如此普遍，那么为什么还会发生变化呢？对于理解变化来说，这个问题是一个很好的起点。你必须把变化视为一种奇妙的、罕见的、值得观察的东西。如果你想掌握高效变化的艺术的话，就不能再把变化视为理所当然。

可是人们确实把变化视为理所当然。那么，如果“不变”是规律，“变化”是例外的话，又怎么会出现这样的情况呢？显然，变化也不是太罕见，没到让我们完全陌生的地步。如果理解了变化的最普遍原因——也就是为什么变化不那么罕见的原因——我们就确实踏上了掌握变化的正路。

设想你是一个职业高尔夫运动员——最棒的选手之一。好比说，你在奖金排行榜上排名第 14，一年挣了大约 187 000 美元，外加跟这差不多的签约、赌彩收入。再设想，你对技术的掌握已经得心应手，总能保持稳定的成绩。一个巡回赛接着一个巡回赛，一场接一场，4 轮下来你总能以接近标准杆数的总分完成比赛。

这样，如果你研究最近几十年的高尔夫比赛获胜得分，你会发现获胜得分越来越低[①]。一些体育作家认为这是球场路线越来越容易的缘故。一些人则认为，这是因为装备改良的缘故，要不就是技术提高了，要不就是奖金提高了，这样就能让好手有更多机会留在比赛中，完成一个成功的职业生涯。事实上，所有这些因素都起了作用，也许还有很多其他潜在因素的参与。

无论是何原因，得分下降的趋势还在继续。这样，如果你的成绩始终一成不变，那么你的职业生涯会发生什么呢？

这一年你挣了 187 000 美元。下一年会挣多少？很显然，如果获胜得分一直在降低的话，你的平均排名也会越来越低。除非大赛奖金提高，不然你的收入会减少。同样，你赢的赌彩也会越来越少，签约也会减少。简而言之，无论你自己如何稳定不变，如果你的环境在变，你

① 我们知道，高尔夫比赛得分（杆数）越低越好。

个人的处境可能也会变化。

如果你开车沿街而下，由街道开到了码头，如果这时你不改变速度或方向的话，你的环境则会帮你改变的。我们把这个称作惯性定律："在变化方面的无能最终将导致戏剧性的或灾难性的变化。"

第二惯性定律

如果"不懂变化"终将导致灾难，那么在这样一个世界中，没有学会优雅地变化的系统也就不能长久。存活下来的那些系统，都发展出一套娴熟的技术，用以抵御变化，或者把变化引往更可接受的轨道上去。

举个例子，还是设想一下，一个人想把一种新技术介绍给另一个人，可后者已经具有相当稳固的技术了。为了不触及什么真实情况，还是拿那个职业高尔夫球员举例吧。这是赛季结束后的时间，你刚刚挣了 187 000 美元。你正在与一个体育用品公司的销售代表共进午餐，他正向你兜售一套全新的、完全不同的球杆。用了这套球杆，这个销售代表声称，你每轮比赛平均能再降低一杆。而且不仅如此，如果你和他们公司签约，你都不用付给他们钱。实际上呢，这家公司为了让你用这球杆，还会付给你 10 000 美元。你会怎么反应？

如果你就是个平常人的话，你会听一会儿，然后拒绝他的建议。他们以为自己是谁？居然来教你怎么提高技术！知不知道，你是奖金榜上排名第 14 的选手？

或者，你确实是一个开放的、善于创新的人，很乐意尝试任何新生事物。你就要来了这么一套球杆，一大早就带着它们到了你的常用场地，来试试手风。一上来你就发现，这些球杆手感很怪，完全不平衡。你的挥杆变形了，你开始打歪了。打了几个洞之后，你想校正这种偏差，却又矫枉过正，一开球就打进了小河里。你在场上慢慢走着，暗自庆幸时间还早，没人看见你的臭球。在球场会所，你把得分卡片加起来算了算，总分居然是 83![1] 比标准杆数超出了 11 杆！比这更糟糕的是，你的自信心动摇了，你有点儿疑心，就算现在赶紧换回常用的球

① 场地标准杆数一般是 72。

杆，你的准心也会受到影响。于是你给那个销售代表打了电话："多谢，不过实在用不上。"

销售代表争辩说，一开始的时候，有一点儿下降是意料之中的事情。可能有一个赛季，或者只是半个赛季，你会比原来的成绩差那么一点儿，不过你很快就又能回到低于标准杆的分数。为了进一步支持论点，这位销售代表排出了上一年的巡回赛结果，让你看看，如果每轮都减少一杆，你会收到怎样的效果。有 3 场巡回赛，这就满可以让你夺冠了。总的算起来，这样你的直接收入就会从 187 000 美元上升到 430 000 美元。这样看来，一开始费些周折难道不划算吗？

但其实你根本没在听他说话。相反，你在盘算，如果每轮都增加一杆，又会怎么样：你的收入会从 187 000 美元跌到 45 000 美元。更糟糕的是，有两个巡回赛你本来赢了的，那样一来就赢不了了，这样你参加的大多数比赛都会一败涂地。说实话，要是你当时没赢得那两次巡回赛的话，这个销售代表甚至都不会来找你谈话。要是你真用了那套破烂儿球杆的话，肯定什么也赢不了。

如果这个体育用品公司了解他们的买卖的话，其实一上来根本就不该找你。相反，他们会去挑一个刚刚因为成绩不好退出巡回赛的职业选手，要么就挑一个从来没打过巡回赛的年轻选手。这些人可没赚到 187 000 美元。事实上，他们这一年的净收入可能是个负数。他们会真的感激那 10 000 美元的签约费用，不过或许他们更会感激人家的关注。但最重要的是，每轮成绩再糟糕一些，他们的收入也不会再下降，所以他们尝试新球杆，或其他革新时也不怕失去什么。相反，如果他们的成绩确实提高了几杆，那就会是球员和球童之间的差别了。

还用得着我再把这个故事还原到计算机编程行业吗？或者还原到我们引进新技术的情况下？让我来总结一下，以上所说的关于变化的两条定律：

1. 环境变化，迫使人们改变。唯有此时人们才会改变。这是第一惯性定律。

2. 如果被迫变化时，人们总要挣扎着保住对他们最重要的东西，牺牲不那么重要的东西。这是第二惯性定律。

罗麦法则

惯性第一定律和惯性第二定律，哪一个是变化中更强大的力量？那些最巨大，持续最长久的变化是怎样产生的？以上问题的答案很有意思，一位古生物学家阿尔弗雷德·罗麦[①]为了解释化石记录中的一些变化，第一个提出了这个问题。罗麦问道，既然海洋生物对水如此适应，而大气对它们又颇有毒害，那么为什么它们还会离开水面，开始呼吸大气，最终到陆地上生活呢？

罗麦想象出一幅水中过于拥挤的情景。也许是因为气候变干燥，一个湖泊范围收缩。也许是因为某种新的鱼类在对食物的竞争中击败了一些老的变种。无论出自什么原因，水中的食物现在有限了，所以哪个物种能够从其他来源找到别的食物，比之其他不能如此的物种，它们的状况也就更为有利。在这样的情况下，可能有一个物种，为了啃咬长在水边的植物——比如说——偶尔屏住呼吸，爬出或跳出水面，而且逐渐把这当成了习惯。从其他鱼类的观点看，这个物种简直是移到第四维空间去了。从它自己的观点看，它只不过是暂时离开了有利环境，本意还是为了在这个环境中长久存活的。

长话短说，第一批爬上陆地的人类先祖，其之所以离水登陆，只不过是为了留在水中。但是，一旦它们走出了微小的第一步，就好比骰子抛出、覆水难收，它们的一些后代最终把陆地，而不是水面，当成了首要的生存环境。所以这就是罗麦法则：

> 最巨大、持续最长久的那些变化，往往最初只是试图去保留一些东西，而恰恰是这些东西最终受到了最大的改变。

在数据处理界，也尽是罗麦法则的例证。很多公司开始采用计算机，本来都不想改变他们自身的业务，而是为了让业务还跟从前一样，一成不变。比如说，当交易规模上升时，在负载之下，旧有的交易处理方法就开始崩溃了。除非能开发一种应付这一增长的方法，否则这项业务就到了生死攸关的时候。这样就引入了一台计算机——但它又引发了一连串不可逆的变化，就像那第一条爬出水面啃蕨菜的鱼

---

① 阿尔弗雷德·罗麦：Alfred Romer，1906—1998，美国古生物学家。

一样。

由罗麦法则可知，想控制变化进程的人，也不用和系统保护自身的努力去较劲，而是应该把这些努力引导到预想的变化轨道中。“不愿变化”的决心，往往是一种最强大的可加以利用的力量。如果你直接跟它较劲，那你几乎肯定要落败。如果你把它向你所想要的变化方向引导，让系统自己去为保护它认为最有价值的东西而奋斗，这样你就很有可能获得成功。这是过去的历史给我们上的重要一课，对于国家、公司、同事都适用。

在编程生产力方面的变化

还是让这些想法进一步贴近我们关心的事情。在过去的 30 年中，高尔夫比赛的获胜比分下降了几杆，在总分的几百杆中，这大概占了 1%或 2%，但这已经让整个职业比赛获得了革命性的转变。在同一个时段里，计算机的速度和能力改变了几百万，甚至几十亿倍。这些变化对于职业计算机人——程序员——又有何影响呢？

从表面上看，并没有那么大变化，这也就是我们常听说的人的生产力没有明显提高的原因。1979 年，我用我的 IBM 5110 做了一次试验，证明所谓“生产力并无改变”的话纯属幻觉，至少在一些情况下是如此。1956 年，我离开学校，到旧金山的 IBM 去工作。人家让我上新型的 650 计算机玩玩，自学编程，因为那时还没有编程课程。解决了一些实际问题之后，我被任命给我的同事们上课——这可是很有教育意义的经历——即使他们没获得教益，我自己可确实获得了。然后，我终于有机会写了自己第一个真正的应用程序。我和一个年轻的土木工程师——Lyle Hoag——一起工作，写了一个分析水压网络的程序（这个系统用来满足城市的给水需要）。那时计算机是个新生事物，这个应用是一种了不起的革新，甚至值得出版——这也是我的第一篇文章，发表在《美国水力工程协会会刊》（*Journal of the American Water Works Association*）上。

从那时以来，我写了上百个程序，辅助完成了上千个程序，几乎所有的都被我忘记了。不过既然这第一个程序最后形成了一篇文章，我也就能在 20 多年后重读到它，从中得到了可靠的数据，能够了解当时

我的生产效率。

用上了我的5110计算机和APL编程语言，我重写了那个程序，看看这么多年以后我编程序的效率有没有提高。1956年的时候，我们两人用了4周全日工作时间，还加了不少班，才写好、测试完了这个程序。1979年，我用了两个半小时，写完了这个程序的第二版，生产力大概提高了200倍。这样算起来，每年大概增长了25%，据我所知，这个速度超过了任何其他的劳动力密集型行业。

当然了，就整体而言，编程技术可没有跟上这25%的年增长率，这主要是因为我们总在把效率最高的人才从编程岗位上挪走或者轰走。另外，即使是这200倍的生产力增长，也没跟上计算机本身在速度上、能力上的提高，所以程序员岗位大大膨胀了——于是，即使尚有一些高水平人才留在这一岗位上，他们带来的生产力提升也被大批涌入的新人中和了。人越多，所需的管理也就越多，交流带来的损失也就越大，成本也就越高——所有这些，都使生产力的增长受到阻碍，所以表面看来，编程生产力似乎变化不大。最后，我们在旧有的软件和软件工具上的投资过于巨大，形成了沉重的负担，每个程序员在提高自身的工作质量、数量的同时，都会被这样的负担拖住后腿。

但是，如果从另一个角度考虑程序员们，以上种种都不应该构成问题。有一次，一个程序员对我这么说过："为什么我的生产力每年提高5%，可我的经理还不满意呢？她自己的生产力10年都没有任何提高。"

我对他解释说，一个经理的生产力是按另一种方式衡量的——按照她管理的那些人的生产力衡量。于是他又问了一个非常尖锐的问题："那么，为什么我的生产力不能按照同样方式衡量——通过我写的程序的生产力来衡量？"是呀，为什么不能呢？

就像那个经理一样，那个程序员也不是一个直接的生产者，而是间接生产者。如果你还没明白这个论点的推论的话，就问问你自己，你觉得下面两个程序员哪个更好：

1. Dorothy，她从前用来写100行代码的时间，现在可以写110行，不过她的程序质量完全与以前一样。

2. Herbert，他还是用同样时间写100行代码，但是他的程序现在

能帮助 1000 个使用它的职员，让他们提高了 1%的生产力。

当我在 1979 年重写那个水压网络分析程序时，我做了大量的改进，尤其是在输入、输出格式上，这些改进还没有算进我对生产力增长的评估中。但是有上百个土木工程师使用那个旧程序——这些人都是高技术、高薪水的人才，可就因为旧有的输入格式准备起来很麻烦，输出格式很难读，他们会多花费很多小时的时间，还弄出好些代价不菲的错误。

可是，传统上呢，大多数数据处理企业都把程序员考虑成普通职员，他们的生产力都是直接计算，而不是根据他们的产品导致的生产力计算的。结果呢，好多程序员只好屈从这种压力，也就按照这种衡量方法隐含的方式工作——提高代码数量，而不是代码质量。

事实上，真正的编程工作是文书工作和创造性工作的一个混合体，就像真正的管理工作一样。在某一个时间、某一个机构重要的事情，可能在另一个时间、另一个机构就是次要的了。所以，不但要求程序员多产出，也要让程序员们来决定，这个要产出的“多”指的是什么。而很多负责数据处理的管理人员，却简单地把上层管理者施加的压力转嫁给程序员们，只不过加上了数据处理管理者常用的一套俗语外壳。这样做的结果，就是程序员们虽然认为自己的生产力之高前所未有，却感到管理层：

1. 不知道他们想要什么。

2. 一直施压要求“生产力”的提高，却连“生产力”这个词是什么意思都不知道。

3. 不赏识已经取得的成就。

4. 设立了重重阻碍，使程序员难以取得任何意义上的生产力提高。

程序员需要什么

通过第一惯性定律，我们知道程序员应该会感知压力。通过第二定律，我们知道程序员对压力怎样反应，取决于程序员认为应该保住哪种重要的东西。这样，我们就要问了，“变化会威胁到程序员珍视的什么东西吗？”

心理学家[①]告诉我们，人类的需要分为不同的等级。这意味着，一些需要被置于另一些的前面——除非它们已经被满足了。如果被满足了，那它们也就不再是需要，就会被下一级的需要所取代。比如说，最重要的是生理需要。如果你没有空气可以呼吸了，那对于获得经理的尊重也就不会太感兴趣。总的来说，在今天的发达社会里，特别是在程序员中间，空气、水、食物都算不上能鼓动人心的动力。计算机能力的巨大提升，并没能真地威胁任何程序员的饭碗，而且法律也不允许经理们做出这样的威胁："你要不快点儿编码的话，我就憋死你！"在其他领域，如果生产力不能快速提升，管理层可以用另外一个等级的需要来做出威胁——安全层面[②]。这样说当然是合法的："如果你不快点儿编码，我就开除你！"但是这种方式能够提供必要的变化动力吗？

在今天的商业环境中，对于大多数程序员来说，开除的威胁并不特别有效。首先，在很多大型机构中，这个威胁实质上是空洞的，因为在这些企业，把人放到一个犄角旮旯继续付给他们工资，都比开除来得容易。但即使开除确实可行，看一眼"招聘程序员"的广告，你就能明白为什么这种威胁缺乏信服力了。

一直有关于"编程行业之死"的预告，我都听了有 20 年了。经理们喜欢听这样的预言，因为如果编程职位变得稀缺的话，再用"开除"威胁程序员，就能让他们听话一些。由于篇幅限制，我没法证明自己的预言，不过我要对看重实际的经理做出以下建议："别非等到编程行业衰落了再解决你的问题。"

实际上，虽然招聘广告铺天盖地，"开除"的威胁倒还是有点儿作用，但这并不是由于程序员对自身的安全担忧。还有另一个层次的需要，那就是对归属于某个社会团体的需要。失去工作对这一需要威胁最大。很多经理气恼、抱怨办公室里的"社会生活"[③]，却不明白对于很多员工来说，在本企业和另一个薪水更高的职位之间，恰恰只有"社会

---

① 人的需要层次理论，由美国心理学家马斯洛(Abraham Maslow，1908—1970)提出。

② 安全，security，这里指"衣食来源、生活保障"，而不是"人身安全"。

③ 办公室里的"社会生活"，the "social life" at the office，指员工们在上司眼皮底下拉帮结派。

生活”这一点儿阻碍。

获得了一种相对安稳、相对舒适的生活之后，普通程序员会去寻求满足更高的需求，心理学家把这些需求列出如下：

归属感——成为某个团体的一员

尊重——被他人高度评价

自我实现——被自身高度评价

以上领域，也就是依照第一惯性定律程序员会感到压力的地方，所以这些也就是依照第二惯性定律，程序员会奋力保住的东西。让我们看看，它们对于我的几个客户是怎么起作用的。

案例 1：金钱作为衡量尊重的一种尺度

Manuel 编程已经有 5 年了，现在他能感到自己编程能力确有提高。当一个学员过来问他一个问题时，他发现，要是他编那个程序，只需要该学员所花的 1/3 的时间。他对“归属感”很满意，因为学员来找他问问题，这让他觉得自己在扮演“一家之主”的角色。同样，这种角色也满足了他对“尊重”的需要，因为很显然，学员很尊敬他的技术。最后，他对自己能力的感知也满足了他的最高需要——“自我实现”的需要。他在做一件很值得做的事情，而且他知道自己擅长这个。

几天之后，Manuel 碰巧得知那个学员挣 30 000 美元，与此相比，他自己的薪水是 40 000 美元。这 40 000 美元满足他的物质需要已经足够，而且他的职位当然也足够安全，但他却觉得不太自在。“为什么，”他想，“我的生产力是别人的 3 倍，却只多挣了 1/3？ 也许我的经理对我的技术不够赏识。”他的尊严受到了威胁，所以他有动力对此做点儿什么了。

他看了看报纸，来验证自己的感受。他得知，没有人会仅仅因为产出是别人的 3 倍就拿 3 倍的报酬，这部分地补偿了他的尊严。他明白了，工资差异与其说是直接反映人的价值，不如说是一种象征。

但是他也看到人才中介的广告说，要招聘一个有他这样经验的人，薪水远远超过 40 000 美元，所以他的焦虑没有完全打消。他倒是不想辞职，因为他所有的朋友都在这儿工作，但他决定在年终考评时提出工资问题。

在考评时，Jean，Manuel 的经理，挺不高兴。上边给了她压力，让她引入新方法，提高生产力，控制人力成本。她计划给 Manuel 略微提一次薪——虽说数额比她本心想的要少——她还想，为了给这么少的数额找个借口，应该提醒一下 Manuel，他还没有采用新方法。另外呢，Manuel 以前到底有多能干，或者说，如果给 Manuel 一个施展的机会，他可以表现得多么能干，她也实在没有办法衡量。

她身上压力不小，因为随着计算机能力的巨大提高，管理层的期待也大大提高了。只要我们编点儿什么程序，门就开了，用户就带着那些他们真心想要解决的问题排着队进来了。结果呢，Manuel 每年 25%的生产力提升，其中很大的一部分都被吸收进了那种种高度复杂的应用，而对于这些应用，是很难归功于他一个程序员的。

更糟糕的是，Jean 没有时间去亲自考察 Manuel 的工作，看看他到底是不是采用了新方法。即使她考察了，她的技术能力也远远不够格了，虽说她从前就是因为技术过硬被提升到管理层的。Manuel 明白这一点，所以要是 Jean 批评他的工作，他就觉得很不公平。但即使她尽力去表扬他的工作，这对于他的自我荣誉感也只有打了折扣的价值。Manuel 当然知道，Jean 已经不再是他所属的程序员团体的一员了。她操起了管理工作，这也就表明，她不再看重原先同甘共苦的情谊了。

这么一个年终评审会，无论是 Manuel，还是 Jean 都不太可能感到满意，也许因此，他们两人都会避开下一次评审。如果 Manuel 能够尽量回避不跟 Jean 接触的话，他也许还会对职位保持满意，尤其是如果他跟同僚相处还算融洽的话。如果不能，他就会马上响应某一条招聘广告，或者给一个招聘者打电话（此人从 Manuel 的一个 3 个月前离职的同事那里听说了他）。有了一个老朋友牵线搭桥，Manuel 对新工作的最大担心就消除了，因为在新环境中，他可以马上归属于另一个团体。

但是，这个故事中最重要的教训是，Manuel 从来没被调动起“通过新方法来提高生产力”的积极性，至少管理层的措施没有给他这样的动力。事实上，管理层每次试图接触 Manuel，给他提供动力，结果都适得其反。Manuel 倒是有了动力，不过不是去提升生产力。相反，为了自己的工作状况，他会选择以下出路中的一个：

1. 离职，找一个新工作。

2. 尽可能避开他的经理的眼睛。

3. 为了弥补自己的“低报酬”，减少工作量。

而且，如果 Manuel 真的感到需要更多金钱时，他也不大可能通过提升生产力来赚取。相反，他可能尝试以下途径：

1. 力争一个管理层职位。

2. 晚上干点儿私活儿，给本地某个会计师事务所的计算机写点儿程序。

这两种方式都会降低 Jean 所在公司的生产力——第一种会损失一个有经验的人手，第二种则会损失这个人的时间和投入精力。

案例 2：用技术评审调动积极性

这一次，让我们看另一种调动积极性的方法。Muriel 和 Manuel 一样，是一个具有 5 年经验的、相当不错的程序员，虽然她对自己有点儿缺乏信心。因此，听到她的经理 Claude 宣布他要引入一套正规代码评审体系时，她非常不安。因为 Muriel 是这个项目上最有经验的员工，所以她的代码被抽中第一个评审。她特别紧张，前一天晚上都睡不着觉。辗转反侧之际，她真的想告病不去了——然后花一天时间按招聘广告上的几个电话号码打电话。

最后，Muriel 还是去上班了，主要是因为不想让她的朋友们认为她是个胆小鬼。评审一开始，她最担心的事情就发生了。在第一轮审议中，就发现了 3 个相当严重的 bug。她想要自辩，但她知道人家是对的。最后，她提出能不能结束评审，给她一个修正代码的机会。

评审组长——他在此前可是做了很好的准备——和气地、但又坚决地解释说，是不是结束评审不能由 Muriel 来决定。只有评审组才有决定的权力，不过如果她真的希望如此，他会询问组员的意见。她确实希望这样，所以评审组就开始讨论此事。让她吃惊的是，她听见一个评审员说，如果只有这么 3 个 bug，那在他评审过的同一开发阶段的程序中，这就是最好的了。当剩下的组员也同意这个意见，而且表示从 Muriel 那里学了很多新技巧时，她更惊讶了。

然后，评审组长温和地批评了组员们：不应该在一开始评审时作

出完全否定的表态，并且提醒他们，Muriel 是整个项目上第一个被评审的人。他说，很自然，她会对批评意见比较敏感，因为不知道人们期望她写出多好的程序。最后，评审组表决继续评审。又过了 20 分钟——其间 Muriel 的技巧多次受到表扬——他们结束了评审，决定采纳这些代码，只要那 3 个 bug 被更正，再成功运行两个测试就行了。

这次经历一个月之后，Muriel 作为学员，参加了我的一个研习班，就是在那时，她给我讲了她的故事。她说，她永远也不会忘记那一天，评审之后发生的事情，虽然她已经根本记不起来具体的细节了。她记的最多的，用她的话说就是一种“光彩”或一种“丁铃铃的感受”，好几个人在她的隔间旁边停下来，祝贺她，说他们听说她完成了一个杰作。

Muriel 的经理 Claude 也在这同一个研习班上，他告诉我，之所以选择 Muriel 第一个评审，是因为他知道她的工作在全公司都是最好的，但每次这么对她说，她从来不信。所以他要来个突然袭击，这样也就成功地满足了她的 3 种最高需要：归属感、同事的尊重、自己的尊重。

在我们的讨论中，Muriel 显得是个正规评审的坚定支持者。她向班上的学员介绍自己公司的一个又一个例子，看到此情此景，真难想象在第一次评审的时候她居然都害怕上班。

福特的基本反馈公式

在这两个例子中，经理们都试图达到同样的效果——激励程序员采用新方法，从而获得更高的工作效率，或许还帮助别人高效工作。可是一个案例是彻底的失败，另一个则是完全的成功。为什么会有这样的差异？

我相信，第一个经理失败了，是因为她认为，为了操纵变化，你必须直接跟员工交流。虽然直接干涉在有些时候是一个正确的手段，但高效的经理却有一整套广泛得多的工作方式。具体地说，Muriel 的经理就特别高效，因为他引导办公室里的社会力量，达到了为整个企业的生产力目标服务的目的。

为了理解这里的机制，咱们再用一个隐喻。生产力低下，就像是污染——在人们的输出中有某种不受欢迎的东西，就像是工厂往河水里倾泻的有毒化学物质似的。早在 20 世纪 20 年代，国会就在调查河流污染了，他们邀请了工业领袖亨利·福特[①]前来作证。福特到了讲台上，就批评国会浪费宝贵的税收，花在种种复杂的反污染法规上。“你们需要的，”他说，“只是一条简单的法律，就能清理所有受污染的河流。”

委员会的成员们听是听着，可根本就不相信，最后也没有采纳福特的建议。但是如果他们真的采纳了，这条“福特法”肯定会管用。你可以自己来判断一下，他的建议如下：

> 任何人，出于任何目的，都可以从任何河流中取走任何数量的水，只要他们最后把水再送回取水处的**上游**。

你知道，人可能情愿喝别人的污水，但他们永远不会情愿喝自己的污水。如果他们必须把水送回上游，你就可以肯定，他们会尽其所能，净化自己的输出。

我相信，这条原则是管理者的武器库中最微妙的一件武器，所以我给了它一个名字，用来向一切时代里最微妙的一位经理人致敬——福特基本反馈公式。简单地说，福特公式说的是：

> 如果你想让人们改变他们正在做的，那么你就应该确保他们所作所为的结果都会反馈给他们自己。

我相信，正是因为管理者没能应用这条原则，我们提高编程生产力的工作才造成了很多问题。很多程序员——也许是**绝大多数**程序员——工作在这样一个环境中：他们从来没有收到过自己工作成果的真实反馈。因为缺乏反馈，所以他们也就缺乏谋求变化的积极性，而且他们也缺乏信息，指导他们做出**正确的变化**。

明白了福特公式，我们也就能理解 Manuel 案例与 Muriel 案例之间的真正区别。Manuel 的经理，首先就不确切地知道 Manuel 在干什么，所以她做出直接反馈的企图就注定要失败。即使 Manuel 真地相信她说的话，他也无法从中获得任何真正的信息，用来提高生产力。

---

① 亨利·福特，Henry Ford，1863—1947，美国汽车业巨头。

她倒是对他该做什么，不该做什么颇有见解，但是说实话，如果他真地依从这些意见，倒很可能会把事情搞糟。

Jean 的方法在程序开发管理者那里很常见。Jean 自己就是从编程职位上一层层爬上来的，在正式成为经理之前，人家交给她 3 个学员，管了 1 年。跟学员打交道的时候，她用这种直接方法取得了很大成功——成功到了获得提升的地步。但是作为一群有经验的程序员的经理，她不可能还用对学员那一套来取得成功。

Jean 以为自己使用的是同样的方法，但她已经没法再向 Manuel 反馈信息了。她不可能像全力负责 3 个学员那时一样，检查每个人的工作。既然没有时间检查实际工作，她也很快就失去了检查工作的能力，即使她有时间，也力不从心了。而且，她从来都没有完全靠自己检查编程老手工作的能力，让她指出每个好的地方，每个坏的地方也根本不可能。但这并不是 Jean 的一个缺点，因为在整个企业中都没有谁比别人强那么多。

在 Muriel 的公司，情况非常相近。但是，Claude 通过整个评审过程汇集了技术知识，这也就胜过了任何单个人的贡献。而且，负责评审的人有充足的时间。每个评审员都直接从每次评审的内容中受益匪浅，这足以弥补他们所花费的时间，而且还能让优秀的新技巧像野火一样四下传开。通过评审得到的反馈，比起 Muriel 经理的反馈——或者任何经理的反馈——都更加切中要害，更令人信服，更加全面。

技术评审有很多形式，对经理来说，另外还有很多种依据福特基本反馈公式利用社会力量的方法。我选了一个评审的例子，因为我相信，评审是任何经理**今天**就能采取的，最立即、最有效的步骤之一。更重要的是，如果不做评审，任何程序开发管理者，对任何提高生产力的手段的实施，都不可能有踏踏实实的信心。我到过几十家公司，全都在吹嘘自己实施了结构化编程呀[①]，特殊的测试工具呀，全新的文档形式呀，新标准呀，或者无论是什么的花样。对于它们中的大多数，经过了一次实事求是的调查后，都能够发现，虽然经理们有这样那样的幻

① 当时结构化编程还属于比较前沿的方法论。

觉，程序员们大体上都不怎么采用那些新花样。而且，由于缺乏评审，经理也就没有任何可靠的方法来打消幻觉。

总结

1. 绝大多数时候，不会发生变化。

2. 之所以不会发生变化，是因为很多力量都在积极地保持事物稳定。

3. 为了把变化往高效的轨道上引导，你必须理解那些保持稳定性的力量。

4. 当人们变化时，那是因为他们的环境发生了变化，所以即使他们一成不变，也还是变化了；他们的环境变化了，所以他们为了保住某种重要的东西，就必须做出改变。

5. 最剧烈的变化，恰恰是因为要在变化中保住某种东西而引起的。

6. 人们要保住的东西在需求等级中排列如下：

生理需要

安全

归属感

他人的尊重

自我尊重或自我实现

7. 在今天的环境中，几乎没有程序员会受到生理需要或安全需要的激励。

8. 如果把人们的工作结果反馈给他们，他们就能知道哪些是好的，哪些是坏的，这样他们就能确保在变化中增加好的，减少坏的。

9. 管理者的任务，并不一定是直接给出这样的反馈信息，而是去安排这样的反馈，并使之正规化、可靠化。

10. 因为程序员的生理、安全需要早就得到了满足，所以满足他们的高层次需要——特别是社会需要——就成了一种成功的管理策略。

11. 有些策略能够引导人的高级需要，对程序开发效率给出可

靠的反馈信息。程序员之间的同级技术评审就是这样一个突出的例子。

12. 技术评审有很多种形式，可以因地制宜，适合任何机构使用；但如果缺乏技术评审，你在其他改进生产力技巧上花费的大部分金钱很可能就被浪费了。

狎弄规则

“什么东西是绿的，有轮子，而且长在房子周围？”

“猜不出来。答案呢？”

“草……轮子是我瞎说的。”

Poul Anderson[①] 书中的一个主人公给一个坏巨人出了以上脑筋急转弯问题，巨人勃然大怒，乱了方寸，后来在猜谜大赛中败下阵来。你会被这么一个问题击败吗？这难道意味着你就是一个坏巨人吗？还是说，这意味着你是一个标准的程序员？

如果让程序员们玩一个游戏，却不让他们知道全部规则，他们总会非常困惑。如果告诉他们这里**确实有**规则，但其中一些未经言明，很多程序员就会开始努力——努力去发现那些潜在的规则。但是，一旦他们感到这里根本就没有规则，或者规则能够任意改变，程序员们就会对此产生敌意，并退出这个游戏。

心理学家会说，这种行为模式体现了一种对“控制感”的强烈需要。程序员们寻求和维持哪种类型的工作，他们对担任管理职务的热衷或冷淡，他们对不同管理风格的整体态度，他们对团队编程和技术评审的看法，还有程序员生活方式中的其他很多方面，都受上面所说的“控制感”影响。程序员喜欢在规则明确的环境下工作，原因之一是：不能在这种环境中成功工作的人，一般也就不会做程序员了。如果你正管理着一些程序员，而且希望你的员工高高兴兴，那么你就得建设一种环境，其中规则明确，而且众所周知。无论你明确地设置怎样的规则，他们都肯定会照章行事。但是，你也得当心。如果你对某些规则并未认真考虑，那么也就很可能收到奇怪的结果。比如说，如果你要求，谁写的代码行数多，谁就多拿薪水，程序员的反应就会像松鼠见了干果一样，开始攒起一堆堆的代码。如果规则改为谁的代码少谁的薪水就高，程序员们也会高高兴兴地再一改编码风格，使劲儿往

---

① Poul Anderson：科幻小说家。

少里写。

有一回，我和一个程序员聊天，按我的看法他的上司为人特别糟糕，可这个程序员却安之若素。我就问这个程序员，那个上司行为毫无法则可言，而且为了往上爬，能对任何人做任何事，作为属下他是怎么忍过来的。那程序员就解释说，恰恰是上司的这种不择手段才让他觉得自在的。其实这个上司并非（像我想的那样）做事全无法则，而是有一个最简单的规则：他不会在别人后面捅刀子，除非这么做能帮他升迁。这样，如果你离他的青云路远远的，他不会因捅你而获益，那么他就永远不会找你麻烦。这样，该人的行为也就完全可以预知——至少对那个程序员如此，这程序员可从来没想过要爬到管理层上去。所以这个属下觉得挺自在——因为他对这里的规则洞悉透彻。

最近，我太太和我受到邀请，到一家大型程序开发机构，去研究新的软件工具在该机构中产生的心理影响。关于软件工具中的这个或那个缺陷，我们听到了许许多多、大大小小的抱怨，但是独有一个问题，每个人都在声讨，而且大家的怨气最强烈：负责维护这些软件工具的系统部门，在任意的时候都可以更改这些工具的工作机制，而且完全不做事先通知。

说起这种做法，程序员们怒气冲天。如果我们想到，程序员需要而且渴望在一个规则明确的环境下工作，他们的愤怒就很容易理解了。只要他们在规则变更之前得到了通知，这种变更对他们来说就是可以接受的。只要人家通过某种更高层次的规则解释眼下规则变更的原因，那么他们甚至会觉得变更还挺不错。但是，就我们面谈过的每个人来说，系统部门那种任意的、没常性的变更方式是完全不可接受的。

随着程序员在事业上的进步，他们关于“什么算是规则”的认识也在逐步深化、成熟。比如说，当客户们在项目进行到中段时要求变更需求，年轻程序员们往往会怒向胆边生。随着他们经验的增长，程序员们就开始推导出一条更高层次的规则：客户们**永远**会在项目进行到中段时要求变更需求。

这样，通过这一个更泛化的规则，程序员也就抓住了事情的实质，因而也就具备了对付那些朝令夕改的客户的本事。每当那种情况下，

他们会觉得讨厌，但不会发怒。当程序员们用新的规则约束住各种可能出现的异常情况时，他们也就具备了很多专门对付这些情况的手段。而如果这种异常情况落到规则范围之外，将导致工作效率的下滑。

就像小孩儿有时会试探家长的反应，宠物会试探主人的反应一样，当新规则刚刚引入时，程序员们也许会发些脾气。但是，如果规则足够明确，程序员们也就会随之迅速调整，并且找到在这些新规则范围内获取成功的模式。打个比方，如果标准手册中突然虚张声势地加入了一大堆新标准，程序员们一开始会大喊大叫、吵吵嚷嚷。但是，如果他们发现这些标准确实被强制执行了，多数程序员也就会遵从。不过，如果让他们发现标准没有被强制，那么他们会推导出一条新规则：这些规则（标准）不算数（“轮子是我瞎说的”）。很容易忽略那些不算数的标准。大家又都觉得挺舒服了，因为一切又都回到规则之下，井然有序。多少世纪以来，程序员们从生活的其他领域学到了种种规则，而且也发明了不少自己的规则。下面是我最喜欢的 10 条——也可以称作“温伯格珍贵编程法则”：

1. 标准不够标准。

2. 总会剩下 1 个 bug 的，你找到了这个剩下的，那还会有另 1 个。

3. 任何有可能发生的事都将会发生，除非你的测试计划中已经算到了它会发生。

4. 要是有人伸出 1 指，点出代码中的错误就在某处，那么就在该处之外的其他地方去找这个错误——很可能错误就在该人的另外 3 个指头指的地方。

5. 1 盎司“预防”顶得上 1 磅“事后补救”，可是管理层不会在“预防”上花哪怕 1 个便士。

6. 需求、设计、编码能以任何速度完成——只有排错花时间。

7. 再长、再乱、再复杂的代码，维护者都能把它弄得更糟糕。

8. 人人都谈“文档”，可从来没人为它作任何事。

9. 你可能缺少硬件，但永远不会缺少硬件销售人员。

10. 每个程序员至少都有 10 条法则，但 1 000 个人里只会有 1 个愿意劳动大驾记下其中的哪怕 1 条。

通过与几百名程序员的谈话，我知道这 10 条法则正确无误。程序员们都有些规则，因为他们喜欢规则。他们特别喜欢“元规则”——也就是关于规则的规则，比如法则 1。元规则构成了程序员最喜欢的一种游戏，但是就像任何递归游戏一样，它也必须有个头儿。有一回，我拜访的一个客户正在搜索某种管理困境的根源，我也从他们那里学到了那个“终极”的元规则。在那个公司四处一团糟的情况下，有个程序员却显得安安静静、高高兴兴。我就问他，他是怎么幸免于难的，他能不能帮别人对付这个困境。

“噢，这挺容易，”他说，“这里边儿只有一条规则。”

“那是什么？”我问。

“这唯一的规则就是：这儿根本就没规则。”

有了这么一条规则，一切也都得到了解释，所以他能惬意自在。也许他这个主意是对的。要么，也许那根本就是草——没轮子的那种。

我要的只是一点儿尊重而已

在《计算机世界》(*Computerworld*)的一次访谈中，James Martin[①]讲了一个国营航空公司的故事。他说，要是你是一个乘客，并且……

> ……对空中小姐们特别失礼的话，他们就会把你放进一个黑名单。这黑名单的作用是：下次你再向该公司订票时，电脑会自动拒售座位给你。
>
> 在一台终端前，他们向我介绍怎样显示这个黑名单。有趣的是，黑名单上的好多名字正巧是大家都熟悉的名人。我怀疑一家以盈利为目的的航空公司会不会也这么做。

这个轶事中首先抓住我眼光的，是 Jim[②] 在国营航空公司跟以盈利为目的的公司之间所作的对照。最近，很多私营航空公司都报告了亏损。我住在瑞士的时候，得知瑞士航空(Swissair)——它是完全国营的——是世界上持续盈利时间最长的一家航空公司。

我在瑞士时学到的另一件事，得自一个朋友，他经营着日内瓦最好的一家餐馆。有天晚上，我们正像往常一样享用美食，却让相隔两张桌子的一帮子吵吵嚷嚷的醉鬼坏了雅兴。我的朋友先是用几句最客气的劝告让他们安静些，无效之后，也只好放弃了。他以个人名义向其他就餐者道歉：对这帮小丑实在没辙。当天晚些时候，餐馆打烊后，他跟我交心地说，餐馆这生意，最让人腻味的就是这些无行的醉鬼了。他先作领班，后当老板多年，可是一旦醉鬼们坐上桌来，他也全然无能为力。但是，他又跟我透露，我可以放心，今晚闹事的这批流氓永远也不会再受他招待了。他太太负责预订事宜，也留着一个黑名单，其中记录了所有这些坏胚。如果他们再打电话订座，餐馆就会客气地说：没座位了。

瑞士人具有一种极发达而根深蒂固的感受力，在公共场所事关他

---

① James Martin：著名 IT 作者，被视为“CASE 工具之父”。Computerworld 杂志曾评选他为“计算机界最具影响力的 25 个人”之一。

② Jim：上述 James Martin 的昵称。

人安逸的问题上，这种感受力使他们总是知道如何举止得体。英国人也具有这种感受力——至少在英国国内的时候是如此。可是他们一旦离开了自己那个紧绷狭小的岛屿，甚至会变得比美国人还粗鲁。这也就是为什么我从来不坐英国或美国航空公司的航班（只要航班的选择由我说了算）。

但是，用这样一抹简笔就为这么两个伟大国家的所有公民画了像，也许还有欠公平。Jim 确实提到了“黑名单上的好多名字正巧是大家都熟悉的名人”。也许这里的作用参数不是国籍，而是某种国际地位。也许“名人们”觉得，他们表现得粗鲁些，其他旅客反而不会觉得那么突兀或不快？

Jim 隐瞒了名字的那个神秘公司可能就是瑞士航空。我还没见过任何瑞士人遇到名人就五体投地。我敢肯定，即使从旅客名单里干掉几个名人，瑞士航空也不会担心自己的盈利受损。我开饭馆的朋友肯定不会因为把一些日内瓦有名的蠢货写上黑名单而担心的，虽然这些人大可以是联合国权贵！别人来他的餐馆，不是为了看名人，而是要在宜人的环境中品尝珍馐美味。

所有这些都导向一个问题：如果有些人让其他人无法安生，那么利用电脑记录这些人的黑名单又何错之有？Jim 的弦外之音是：无论这个做法道义上是对是错，这总是个愚蠢的商业手段——尤其是因为，多数粗胚还都是名人。也许在瑞士住得太久，把我惯坏了，但在长途飞行中，我并不在乎名人不名人的。我要的是这样一种邻座：安安静静，不在非吸烟区抽烟。

当然我的认识既有局限，也不乏偏见。对于那些一辈子只过一次假期的旅客，要是隔着过道就坐着一个好莱坞明星或一个计算机界伟人，也许他们会狂喜不已，相比之下，忍受一下这些名人的无礼行径自然是小小不言的代价。要是这些人的行径不够无礼的话，旅客们还没法知道他们的邻座到底是怎么个有名法儿呢！

也许我不该对名人们太狠。我们本来就生活在一个粗俗的社会里。在相对拥挤的环境中，似乎很难获得起码的尊重——这尊重本来是任何常人应有的权利。因此，我们中的很多人，都已经出离地愤怒，一味努力获取专业领域中的尊重，作为社会尊重的一种替代物。

我小的时候，经常被当成“低于人类”的某个东西对待。这种待遇的结果，就是我的思维方式在好几个方面都不乏扭曲。我老是想着通过自己的专业工作获得尊重。“无论如何，”我这么推算，“如果我是周遭最棒的程序员，他们就会体体面面地对待我了。”我也确实获得了一些专业上的成功，但这些从来没能弥补童年受到的挫折。首先，无论我怎样优秀，我也不可能做到完美，做到不可或缺，所以没人非得尊重我不可。

但是当人们确实欣赏我的杰作，体体面面地对待我时，我还是不满足。我要的是起码的个人尊重。实际上，我自己嘀咕：“他们并不尊重作为个人的‘我’；他们只是因为我做好的工作而尊重我。”

最终，我才明白，我真心想要的是**自我**尊重。其他任何人都没法给我“自我尊重”，虽然一些人曾经能够夺走我的“自我尊重”。当我最终能够善待自己时，我发现在航班上我也能和其他旅客和睦相处了——而他们对我的专业成就甚至一无所知！

我还感到，现在我对无礼行径也有了一点儿理解——即使这些行径是针对我本人的。那些无礼的人，当他们举止无礼的时候，相当缺乏“**自我**尊重”。当我们憋在座位里忍受长途飞行的时候，大家都有时会产生这样的感受。但是有些人几乎全程都表现如此，那么这些人，也就登上了我的黑名单。

名人不尊重自己，这也很有一些原因。也许他们——和我从前一样——寻求成功只是为了弥补童年的挫折。或者是在成名之后，他们感到崇拜者们的梦想和他们内在现实之间的差距太大，他们敏感的内心时时受此煎熬。

如果你在自己选定的职业中干得不错，你也将不得不如此面对关于自身的一些事实。我们的社会—经济系统自有一套回报专业成功的方式，由于这种方式的影响，我们很容易认为外在成功也就等同于内在价值。但是你内心中的某一部分永远不会受此愚弄。内在价值和外表成功的分歧越来越大，这里产生的张力，也就使你很容易把无名怒火引向航班上的空姐，或是其他任何碰巧看见了你困窘的秘密的人。

所以还是先别操心你是否在航空公司的电脑黑名单上。操心一下：你在自己的黑名单上吗？

蝴蝶和毛茛[①]：一个寓言

很久以前，一只蝴蝶遇上了长在草山上的一棵毛茛。蝴蝶说了："你是我见过的最美的蝴蝶，你柔美的双翅，真让人叹为观止。"

毛茛回答："我从没见过其他花儿有你这样的自由精神，你的飞舞，简直不受微风的左右。"于是二者相爱了。

一只甲虫，在旁边的叶子上歇脚，正好目睹了这桩如火如荼的恋情，不由得对恋人们喊出话来："你们这些笨瓜，难道就不知道异种通婚没法繁衍生息？睁大眼睛瞧瞧，做事儿也得讲点儿理性吧。"

可是蝴蝶和毛茛全没听进去甲虫的劝告——他俩爱火正炽，早对外界闭上了耳目。所以，在夏天剩下的日子里，他们幸福地——也是幼稚地——共渡爱河。另一方面，那甲虫则产下了 38 312 个蛋，其中 8 279个在后来的几年里陆续孵出。最终，一个小姑娘爬上山来看这蝴蝶和毛茛，甲虫却让她一脚踩死了。

教训：与其做一只出色、高产的甲虫，也不如做一只不育的蝴蝶，或一棵憔悴的毛茛。

换句话说：让一心求产量的人理解那些一心求质量的人，近乎不可能——反之亦然。

① 蝴蝶(butterfly)和毛茛(buttercup)：又是作者的一个文字游戏。

第4章

我们能更有效地思考吗

为什么人们根本不思考

伦敦的读者 David Flint 给我来了一封信，非常启人深思。他是这样开头的：

> 我怀疑，SP（结构化编程）目前进展不够顺利，可能跟 Dijkstra 最初那封信的标题有关。[①] 从那以后，人们以为只要避免使用 GOTO，那就是在做结构化编程了，代码也自然就改善了。当我给其他的程序员和邮政系统的分析师介绍结构化编程时，他们的类似偏见让我大吃一惊。
>
> 在我看来，SP 的要点并非在于“一个程序需要有一个结构”，因为任何事物或多或少都会有一个结构，相反，这个要点应该是“程序应该有良好的结构”。人们不赞成这个意见，因为“良好”看上去比较主观，还因为实现“良好的结构”要比单纯排除 GOTO 语句难得多。程序员们，至少是商业公司里的程序员，都太急着编码，太不愿意思考——也许你该写一篇文章，解释一下为什么会这样子，又应该怎样解决这个问题。（实际上我觉得这个原因

---

① Dijkstra：著名荷兰计算机科学家。他给《ACM 通讯》编辑的一封信极大地推动了结构化编程运动。该信的标题是“Go To 语句有害论”（Go To Statement Considered Harmful）。——附带说说，据资料记载，这封信本来是 Dijkstra 的一篇投稿，以信件形式刊出是编辑的决定，这个题目似乎也是编辑所拟。

是显而易见的。)

我完全同意 David 的意见,只是到最后的“显而易见”上才有所保留。我学习过数学,从中知道一个道理:应该当心“显而易见的”或者“显而易见地”这样的词。这个词是专门设计出来,用以麻痹你的思考,让你昏昏欲睡的,而这,恰恰是我们要避免的。这样一来,你就可以定出一条规矩——也许比“排除所有 GOTO 语句”要高明:

无论何时,只要发现一个“催眠词”,就要立刻清醒起来!

在《走查、审查与技术复审手册》(Handbook of Walkthroughs, Inspections, and Technical Reviews)一书中,Daniel Freedman 和我花了好几页的篇幅,专门列举“催眠词”。在阅读/复审规格说明书时,我们把这种“催眠词”看做触发器。这里还真有好些挺可爱的东西,包括“所有的”、“永远”、“每一个”、“可能”、“只有”、“同一个”、“应该”、“也”、“将要”、“当然”、“所以”、“很明显”、“显而易见地”,或者“傻子都能看出来”。我们的语言中存在这些催眠词,从这个事实就可以引出为什么人们不思考的一些原因——David 的信中其实已经暗示这一点了:

1. 有人不愿意让人们思考,所以就以这样的方式写作、说话,从而扼杀思考。

2. 没人在乎自己是不是思考,但是我们的语言习惯,很容易就扼杀了思考,因为在其实并不确定的时候,我们却会说得很确定。

为什么语言会扼杀我们的思考?嗯,绝大多数场合(当然是编程领域之外的场合),思考会给你带来很大麻烦。在大多数工作环境中,你要是想得太多,那就会冒丢掉工作的危险,因为工作完全成了惯例。思考被当成是管理层的一种特权,就像桌子旁边摆一小盆热带树也是这么一种特权一样。

即使在我们的私生活中,思考也是昂贵的事情。要是你每吃一口食物都得思考一番,每次过马路也得思考一番,那样你就没什么时间干别的了。所以,就必须把生活惯例化,从而避免思考,节省时间来做其他事情——比如把省下来的时间用于思考必须思考的东西。

这对编程也适用。如果我们确实发现了什么简明的规则,遵照它就能百试不爽、避免思考,那么我们就应该采用这样的规则。还是那

句话:把思考留给必须思考的东西。

比如说,PL/I 和 COBOL 允许对于一些关键字使用缩写形式,但是是否确实使用缩写则是可选的。这也就意味着,每次你写代码写到这样一个关键字的时候,你可能都要想一想,是用缩写好呢,还是不用缩写好。虽然你可以列出两种理由的大量论据,其实,只要你对于同一个关键字的做法保持一致就行了(无论是否缩写)。因此,我总是一开始就一劳永逸地决定对某个关键字是否缩写,这样也就避免了不必要的考虑。我从来不能完整拼出 CORRESPONDING 或者 ENVIRONMENT。但是对于 CHARACTER 和 PICTURE 则总能拼出来。这是很蠢的习惯,但是这么做能达到节省时间的目的。

当然,如果我是为一个大系统的某个部分编码,而整个系统开发中规定了关键字的标准写法,与我自己的惯例不一样,那么我也甘愿改变我的习惯。不过这可要费一些事情了。我编码的时候就要分神考虑,到底要遵循哪一种体系。所以,如果有这样一种体系能够节省我这方面的考虑,我自然会感恩戴德,但现在我也不会把宝贵的时间用在抱怨这种体系的缺席上。

这也引出了人们不爱思考的另一个原因:

3. 思考会让人疲倦,因此也可能妨碍其他的思考。

当一个程序员第一次开始用结构化方式编程时,他/她从前的一些节省思考的办法就都不管用了("我从来就是这么写循环的","我从来就是这样处理一个 3 叉分支的"等)。以前,这些习惯能够节省思考,而用其他习惯取代它们则会花费思考。在这种转变过程中,我们会感到大量的思考被花费了,很少思考被节省了。

而那个"不要 GOTO"规则,实际上却是另一种节省思考的习惯(至少对我是如此)。当我发觉自己开始要写一个 GOTO 的时候,我把它当成一个触发器,而不是一个催眠词。我会对自己说,"温伯格,从前你在类似情况下不假思索地写了 GOTO,结果导致了不少多余的思考。因此现在干吗不多想想,这样也许会在以后必要时节省不少思考呢。"

结构化编程的所有其他规则——以及任何优秀编程风格的规则——都是这种类型的触发器规则。它们说:"当你发觉是这种情形

时，就别再昏昏欲睡了，动起脑子来。”

你不可能所有时候都一直保持清醒意识，即使是在做开车这样危险的事情时也是如此。所以，你能慢慢积累起若干“触发器”，一遇到危险，立刻就把自己唤醒。比如，在你驾车的时候，这些触发器就包括红灯、黄灯、警笛、路上滚过来的皮球、嬉戏的小孩，或者前面左摇右摆的车等。

在编程时，代码中也有一些触发器，这包括 GOTO 语句、混乱的 IF-THEN-ELSE 判断、相近但又不一致的段落、意思含混的接口，以及任何注释要比代码还长的代码。

在我们的研修班上，我们设计了一些“触发器按钮”，发给学员们，这样就能帮助他们对一些特定的说法提高警惕——这些词往往能够给他们带来麻烦，因为说出这样的词，要么是自我催眠，要么是催眠别人。比如说，我们最喜欢的一个按钮是：

没办法了。

显而易见(!)，你这么说的时候，其实意思是，“既然已经不可能了，那思考这个问题也就没用了。”每次我们听到“没办法了”，我们就立刻给犯规的这个人戴上这么一个“按钮”，直到他学会不再说这个“催眠句子”——或者，至少学会听见后立刻抓住这个句子，并且把它当成一个思考的触发器。

我们很喜欢写在“按钮”上的另一个催眠句子是：

这怎么可能有错呢?

这也就和 David 在来信第二部分讲的内容刚好吻合：

> 也许我可以推荐一个信条：“怀疑论的设计”。这里的要点是，在进行设计时，首先应该检讨自己的前提和假设。作为这种信条的一条圣言，罗素的说法很合适：“无论你相信什么，都别完全信它。”当然，罗素在说这句话时考虑的还是更普遍、更深刻的问题。

对于这样一种新的编程信条来说，“怀疑论的设计”是个了不起的名字，不过如果已故的罗素知道他的那句话被当成一句“圣言”，说不定在墓穴里都要暴跳如雷了。下一次，如果你对自己刚刚完成的设计特别满意，那就别这样问：“这怎么可能有错呢?”而要问“什么地方有

可能有错?”

当然,这也就把我们带入了人们不思考的最重要原因:

4. 要是我想得太多,没准就会在这里发现错误,那我又该怎么办?

同样,这也不是一个彻头彻尾的笨想法。通过多年的学校生活,我们学会了这个道理:“如果老师没发现,那就不是错误。”所以,如果我们知道老师其实没有时间看我们的作业,那我们又为什么要自寻烦恼地排除错误呢?

但问题是,计算机编程与学校作业很不一样。这里的“老师”——也就是计算机——会查看所有的作业,任何一处错误,都会被上百倍,甚至上千倍地放大。在学校里,采取鸵鸟政策,把头埋进沙子,大多数时候都能够蒙混过关——所以人们在编写计算机程序时,往往也会同样行事,这也就不奇怪了吧?

总结一下,我能得出人们不思考的 4 点“显而易见”的原因:

1. 有人不愿意让人们思考。

2. 没人在乎自己是不是思考。

3. 过度思考会让大脑疲倦,因此也可能妨碍其他的事情。

4. 在学校里我们没学会思考。

“显而易见地”,这张单子不可能有错误或不完整的地方。难道你想接受挑战,提提意见?

你是哪种类型的思考者

有这样一个问题，叫做“岛屿测验”，很多讨论者都把它当成一种测试不同思维类型的办法。最早它是由 Robert Karplus 在 1970 年左右发表的，培训教师的时候常会用到它，帮助他们识别不同的解决问题风格。对于大学生来说，这道题还真有点儿挑战性，但是对于专业的数据处理工作者来说，就应该是小菜一碟。是不是这样呢？你也来试试吧，但是要留心。把你的答案和推理过程都写下来，然后读我的笔记，看看你自己会有什么样的反应。

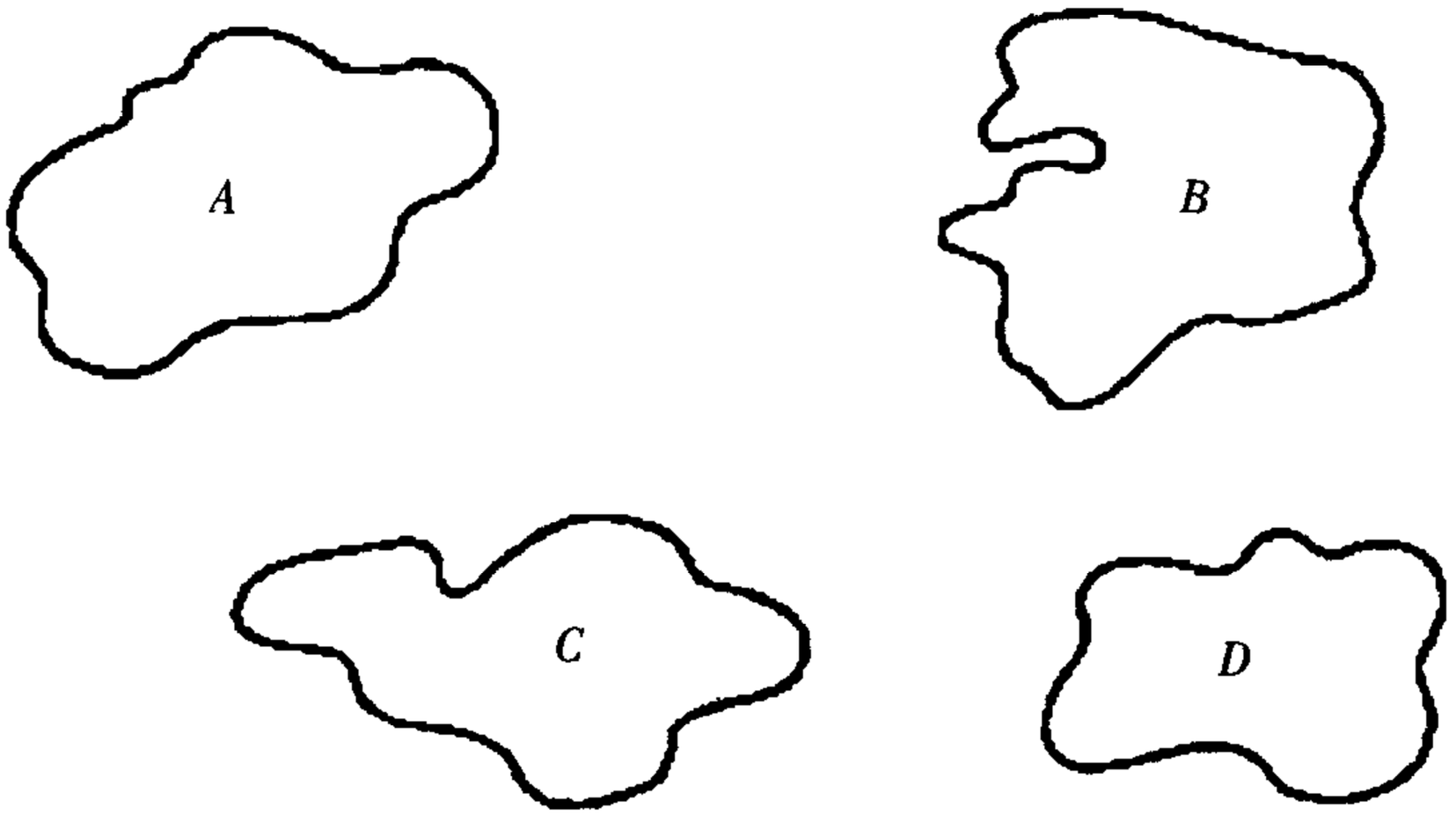
岛屿测验

这个问题讲的是大洋中的几个岛屿 A，B，C 和 D。从前很多年，人们在岛屿之间过往都要借助船只，但最近开了一条飞机航线。请仔细阅读以下关于目前可能路线的线索。路线可能为直达，也可能包含

在一个岛屿上停顿和换机。如果一条航线可行，那么也就在两个岛屿之间双向可行。为了更好地利用线索，你可以记一些笔记，或者在图上作记号。

第一条线索：人们能够在岛屿 C 和 D 之间通过飞机往来。

第二条线索：人们不能在岛屿 A 和 B 之间通过飞机往来，即使间接往来也不行。



利用这两条线索回答问题 1。但现在不要读下一条线索。

问题 1：人们能在岛屿 B 和 D 之间通过飞机往来吗？

能____不能____无法从这两条线索中判定____

请解释为什么如此回答。

第三条线索：（现在别修改你对问题 1 的答案了。）人们可以在岛屿 B 和 D 之间通过飞机往来。

问题 2：人们能在岛屿 B 和 C 之间通过飞机往来吗？

能____不能____无法从这三条线索中判定____

请解释为什么如此回答。

问题 3：人们能在岛屿 A 和 C 之间通过飞机往来吗？

能____不能____无法从这三条线索中判定____

请解释为什么如此回答。

岛屿测验笔记

1. 问题的叙述中说，一条路线“既可以是直达，也可以包含停顿。”显然这个问题中所用的术语跟一般民航术语不一样，因为我们平时说的“直达”航班也可以停顿。所以我最好还是留心——也许还有其他不容易理解的内容。

2. 其实我对航空旅行有不少知识。那么，如果在解题中，术语方面出现了困难，我能不能用上这些知识呢？也许我在航空旅行上的知识比出题的人还要多，那么我甚至可能得到那个人都无从想见的答案。其实我在跟旅行社打交道时就常常发生这种事。旅行社应该知道的不少了，但是我航空旅行的次数比他们都要多，而且还去过世界上很多不同的地方。

3. 第一条线索挺直截了当——你查一查《航线指南》，就能发现 C 与 D 之间的航班。但是第二条线索就令人生疑。你怎么就能看出，你从一个地方到不了另一个地方？这样一条信息是从什么来源获得的？这里总会有一些推理过程，比如：

a. 任何人都不能飞往岛屿 A，要么因为那儿没有机场，要么因为有法律限制之类的东西。

b. 旅行社核对过了，没找到路线。

c. 可能其实是有一条路线的，但是旅行社认为那样中途停留时间太长了（我常常碰到这种事），所以干脆就说这样“办不到”。

d. 可能存在某种限制，比如检疫隔离之类的，因此不允许任何人从 B 坐飞机到 A。

4. 还有很多其他的可能，但是看了 3(d)，我不由得又重新读了一遍那条线索：“人们不能在岛屿 A 和 B 之间通过飞机往来，即使间接往来也不行。”听上去，这非常像是一种传染病，要么就是一种政治隔离。如果确实如此，那往往就能找出通融的办法——行贿呀，领个疫苗注射证书呀，让政府高层开封介绍信呀，同意对方搜身呀，等等。所以，虽然我们还没有完备的信息，我们也很有理由质疑那条线索的来源。也许提供线索的人并不知道这些办法；也许他们压根就搞错了情况；也许他们存心误导我们。可能问题并不是表面上的那样。

5. 至于问题 1，在目前的线索中当然没有充分的信息。我们确实知道，飞机可以到 D，但是也许飞机根本到不了 B。当然，这可能性也不大。有一回，人家告诉我飞机到不了塔斯马尼亚的玛丽亚岛，但我们租了一架飞机，就飞到了。B 看上去是 4 个岛中最大的。可能岛上有很多山，可玛丽亚岛也是如此，我们当时找了一块平地降落。不过最好要保证安全，所以还是回答“从这两条线索中看不出来”，即使除了明显的“线索”之外，还有很多其他线索——岛屿大小，面积与距离的比例关系，关于曾有船只通航的知识（在那些礁石岛屿之间，我们则通过浮舟往来），关于“我们的信息来源缺乏知识”的信息，等等。

6. 让我们看看第三条线索。啊哈！这意思是说，B 和 D 之间一直能够通行，还是说最近开设了新业务？如果是前一种情况，提供信息的人不早告诉我们这个，说明他并非知无不言。如果是后一种情况，我们就能得出一个结论：这岛屿航班的业务变化很快。所以这条线索信息量很大。

7. 第三条线索可能暗示，航班计划经常变化，所以回答问题 2 风险不小。如果航班计划不变的话，很可能你就能够从 B 经过 D 再到 C，但是可能还会有检疫隔离之类的问题。我以为，最好还是回答“无法从这三条线索中判定”。我甚至不知道，所谓的“三条线索”指的是哪些线索，也不知道先前的线索是否有效（就算是它们本来还有效）。

8. 至于问题 3，我们仍然连 A 是否有飞机都不知道，不过我们倒还可以租一架飞机。如果线索 2 绝对真实，而且在航班变化后（或者旅行社知道更多信息后）依然有效，我猜你会得出这样的结论：从 A 到 C 不可能有航空旅行。但是这与我周游世界的经验完全不符，所以我肯定不会说“不能”的。按照概率推算，“能”似乎可能性更大。如果答不对就要我的命，那我只能说“无法判定”。但是，如果必须从 A 岛到 C 岛去才能不丢工作，按自己的经历来说，我也一定不会放弃。我会给某家包租飞机的公司打电话，也许还会给另一家旅行社打电话。如果经验能起作用的话，我也许就可以找到一条从 A 到 C 的航线，而且没准还能节省些费用。

反应

这里实际的问题是:“你觉得我的分析怎么样?”

我觉得你在打岔。

我觉得你没看准问题。

我觉得我没看准问题。

我觉得你的结论是对的,但推理是错的。

我觉得你的结论和推理与出题的人所设想的推理一样有道理。

我觉得所有上面所说的都既是对的,又是错的。

我觉得上面所说的没有一条是对的。

我觉得在这道愚蠢的问题上我得不了分。

我从不让自己有什么“感觉”,尤其是在多选题和智力题上。

我以为,写下这些话的主要作用在于,让我自己和你们都略略受到一点震动,不再把某些事情视为理所当然。尤其是,我们总把学校当成生活的一种理想模型,但其实生活比学校里那些测验要复杂得多——即使是那些专门设计的,用来教育我们如何在真实生活中思考的测验。

有人说过,如果你在岛屿测验中“答对了”,你可能是一个很好的程序员,而如果你的回答跟我的差不多,那你可能是一个很好的系统分析员(当然并不一定就是个坏程序员了)。至于我自己,从 20 多年的经历中我得出了这个结论:无论分析员也好,程序员也好(还包括他们的经理),最大的问题就是他们太爱想当然。尤其是,他们总以为他们知道自己面对的问题是什么性质——总以为那是一个“测验”,而不是一个“问题”。

到底是集中还是强迫

每个行业都有自己的秘密。虽然对我们自己的行业秘密非得守口如瓶不可，但是任何其他行业的任何秘密当然都会吸引我们的好奇心。在所有秘密当中，医疗秘密似乎最为诱人。与大多数人不同，我有幸有一个当医生的大舅子，Marvin，而且还是个相当玩世不恭的医生。Marvin 跟我保证，医疗行业的最大秘密之一，就是 90% 的病都是自己好的——根本不用医生动手。正因为有了这样一个了不起的秘密，医生要做的一切，就是避免伤害病人。嗯，近乎一切吧。一个成功的医生还得让病人相信，为了治好他的病，他可是做了一些事情的——而且这些事情只能出自医生那个净是秘传医药知识的聚宝盆。如果这个行不通，那么救济站前很快就会有医生排队了。

90% 的病都是自己好的，这里的原因在于身体本身的智慧。虽然“智慧”这个词听起来很神秘，这不过是一种富有诗意的总结，其背后则是千百代的破坏性测试，作用在亿万次的并行复制上，而每次这样的测试都要持续几十年。这些世代和测试是如此古老，基本上都没有机会从现代医学中受益，所以人体中的任何构造，如果缺乏自我治疗的智慧，都肯定会在进化过程中被淘汰。说到底，我们每一个人都是这条几乎无穷无尽的幸存长链上的直接后裔。

不幸的是，这个小小的医学秘密似乎对计算机人士帮助不大。正如我们所知，程序中很少有——如果不是没有——“智慧”。一个计算机程序很少有——如果不是没有——自己的祖先。它如果经过了什么测试的话，也仅仅局限于自身生命周期之内，所以也就很难有空间和时间，进行上百代的、亿万次的测试。这也就是为什么计算机编程这个行业要比医疗行业要求高得多。这也就是为什么我们永远不用在救济站前排队。

在这样一个高要求的行业中，只要可能从其他行业取得帮助，程序员们当然都很需要。所以，即使那个医学秘密帮不上什么忙，Marvin 还教给我几个其他的秘密，没准你也能够用上。首先，我们有盘尼

西林。所以对于那 10%不能自我治愈的病，盘尼西林和其他的几种抗生素又能对付其中的 90%。但是与一般人相信的正好相反，单单是用了抗生素还不够。还得用得恰当，所以这时就是医生登台施展的时候了。

比如说，如果你每次感冒都不分青红皂白地吃盘尼西林药片，那可能会造成真正的损害。首先，感冒属于那些能够自我治愈的 90%的疾病，所以盘尼西林其实作用不大。可能有一些心理上的辅助作用，但是那种没有药效的安慰剂就能起这个作用了。盘尼西林无论怎么说都不是安慰剂，这也就意味着，虽然对感冒没什么作用，它却另有其他的作用。至少，它会影响你的身体对盘尼西林的敏感性，所以如果有一天你真需要用上盘尼西林，它可能不太管用，甚至根本不管用了。更糟的是，据说它还可能产生副作用。

与盘尼西林有关的另一个常见医学问题，正好跟不分青红皂白地服用相反。也许是因为关于乱服药的警告太多了，人们常常太早就开始停用抗生素。他们不完成整个疗程，而是一旦最明显、最难受的症状并不马上消失就停止服药。在细菌感染的情况下，即使受感染的组织已经恢复了控制，症状还是会延续一段时间。但是如果太早停用抗生素，病情就又会反复。而且这一回，可能就会对抗生素产生抗药性了。

你看出这与编程之间的关系了吗？当你在对付一个难缠的问题时，有没有这样的情形：过早地放弃了一个很有希望的方案，只是因为如果采用这样的方案，在最后结果出现之前你要全力投入得太多？对这种苦涩的经历，大多数程序员都很熟悉。这也就是为什么程序员们总能够在某一个问题上集中全部注意力，不顾其他琐事的原因。你告诉我哪个程序员从来没有在别人喝咖啡的休息时间继续工作过，我就能告诉你哪个程序员是平庸之辈。你告诉我哪个程序员从来没有抱怨过别人打断了他的重要思路，我就能告诉你哪个程序员简直就是行尸走肉。

但是此刻，还得让我打断你一下，先别考虑“集中注意力”的问题了。我给你讲讲 Marvin 透露的另外几个医学秘密。实际上，Marvin 不仅仅是个医学博士，还是一个心理治疗专家。（哦，这还不是那个秘

密。)在所有与医疗有关的话题中,当然是疯子们的故事最吸引人了。不过我这个故事可与疯子没关系,只与心理治疗专家有关——我们知道,这种专家当然不可能是疯子,但是也不一定就很聪明。

每个月,Marvin 都要开车到州立精神病院,去给那里的医师做咨询,帮助他们治疗那些最棘手的病例。这种咨询最容易不过了,Marvin 告诉我,因为其实他根本就不需要任何医药知识或者心理学知识。每次他们拿出一个难缠的病例,Marvin 就问他们,正在用什么疗法。如果他们说疗法 A,他就告诉他们换成疗法 B。如果他们说 B,他就说换成 A。当然了,他这么说的时候还得包上一层那种特别晦涩的术语(医生最吃其他医生的这一套把戏了),但是主要原则很简单:无论他们正在用什么疗法,都让他们立刻停用,换某种其他的办法。

很容易看出来为什么 Marvin 这一套管用。他是一个咨询顾问,所以要他来过问的病例,都是本院的医生自己治不好的那种。所以他就得出了结论,有一点是不会错的:无论他们在对这些病例用什么疗法,这疗法都肯定不对。他们被某个思路绊住了,没法脱身。所以,正因为他们付给 Marvin 一大笔钱做咨询,这一套旁观者清的办法才能让他们脱离原来的思路——如果 Marvin 身价这么高,他肯定知道自己在干什么!

如果你是一个好的程序员,你也会看到,你其实已经知道这个秘诀了。困在一个问题上整整两天,却被一个同事的偶然提议——或者是一个临时的打岔、一个偶然事件——给解救了:如果哪个程序员没有这样的经历,那他的灵魂一定僵死得可以。是呀,如果各行业之间能够互相传授秘诀的话,它们真的能够彼此受益呢。

但还有一个更能让人受益的办法,那就是把某个行业的“秘诀集合”都拿过来,然后做对比考察。比如说,考虑一下我们刚刚讨论的这两条:

1. 不要太早地放弃一种疗法。

2. 不要让一种疗法耽搁太长时间。

嗯,也许说到底,医生拿那么多钱是有道理的。他们这个行业秘诀中的秘诀,不在于这些“秘诀”本身,而在于知道何时应用哪个秘诀。也许我们付给他们这么多钱,也不仅是为了他们“知道怎样做”,而且

是为了他们“知道何时怎样做”。

同样，在编程行业中，我们的行业秘密也是一些两两互补，或者两两矛盾的法则。如果我们只按照其中的一个办，那另一个就会坏了我们的大事。程序员们失败的一个原因，是因为不能在一个问题上集中注意力，直到把问题解决。但另一个失败的原因，则是对一个问题太过沉迷，以至于没看到某个“显而易见”的解法。

我还从 Marvin 那里学到了另外一课——这一课讲的是“疯狂”是什么意思。心理治疗专家必须研究“正常”人，才能明白疯子是怎么回事。似乎对于大多数情况，“疯狂”的行为，只不过是“正常”行为做过了头。

我相信，这一课对于程序员，对于所有有志于解决问题的人，都适用。我们的最差解题方法，其实就是我们的最佳解题方法做过了头。如果做过了头，集中注意力也就变成了强迫性精神病。面面俱到也就变成了优柔寡断。把最佳解法和最差解法分开的那一条线，也许比我们想象的还要细微，这也就能鼓励那些目前最差的人，并且让最佳的那些人保持清醒。就像 Will Rogers① 有一次提到猴子时说的那样：“我们觉得它们滑稽，是因为它们太像我们了。”

---

① Will Rogers(1879—1935)，美国著名艺人。

大脑会变得不健康吗

我可不是那种泡夜总会的人。按我的记忆，上一次我去夜总会，还是 1957 年在迈阿密滩，参加 IBM 百分百俱乐部大会[①]的时候。那天晚上可有不少值得记忆的事情，不过能在这里印出来的只有一个——其实我也不敢保证这个就一定能印出来，但是为了引出我的论点，还是得讲讲这个故事。

我记得在聚光灯下说笑话的那个喜剧演员，穿着一身再完美不过的燕尾服，让人都看不出他是个小丑。他先是对观众说了几个有伤大雅的段子，暖了暖场，然后突然停下来，全身笔直地站着摆了个姿势，开始给大家讲他自己的生活故事。“我从前可不是这样的，”他哀叹说，“但是很早我就学到了一句名言，从此这句话不离我的左右：‘头脑好，身体好’……这可是你们要自己选择的事！”

这句话呢，让他这么一说简直挺可笑，但是我们中的大多数人确实很早就做了这么一个决定。或多或少，我们有这么个印象，运动员都很蠢笨，而程序员都没什么力气——我们或是做前者，或是做后者。如果你在学校里选择了那条“没力气”的路，那么业余时间就得全用在图书馆、自修室，或者计算机中心里。如果你觉得有点动摇了，同事们谈到运动员时那种贬损的口气又会让你恢复自信——当然，他们敢这么说还得等酒吧间里没有运动员出现的时候。

我在前面写到 J. Gerald Simmons 所说的成功者的“个人化学”时，强调了人的身体有一些极限，超过了这些极限，你的大脑就不可能不受影响。我们可以用图 1 表示这种关系。从该图我们可以看出，身体好，就能提高工作效率，这样也就产生出更多的空闲时间，可供工作者采取更健康的生活方式。这样一幅“直接效果关系图”可以根据每个方框对下一个方框的影响加以分析（参见拙作《论稳定系统设计》，Wiley，1979）。通过这样的分析，我们能发现，贯穿整个循环的是一种

---

① IBM 百分百俱乐部大会（IBM Hundred Percent Club），是该公司的一个著名的年会。在风景名胜、度假佳所进行，用来表彰业绩卓著的员工。

积极的趋势，这样一来，比如说健康的身体也就能让身体更加健康。

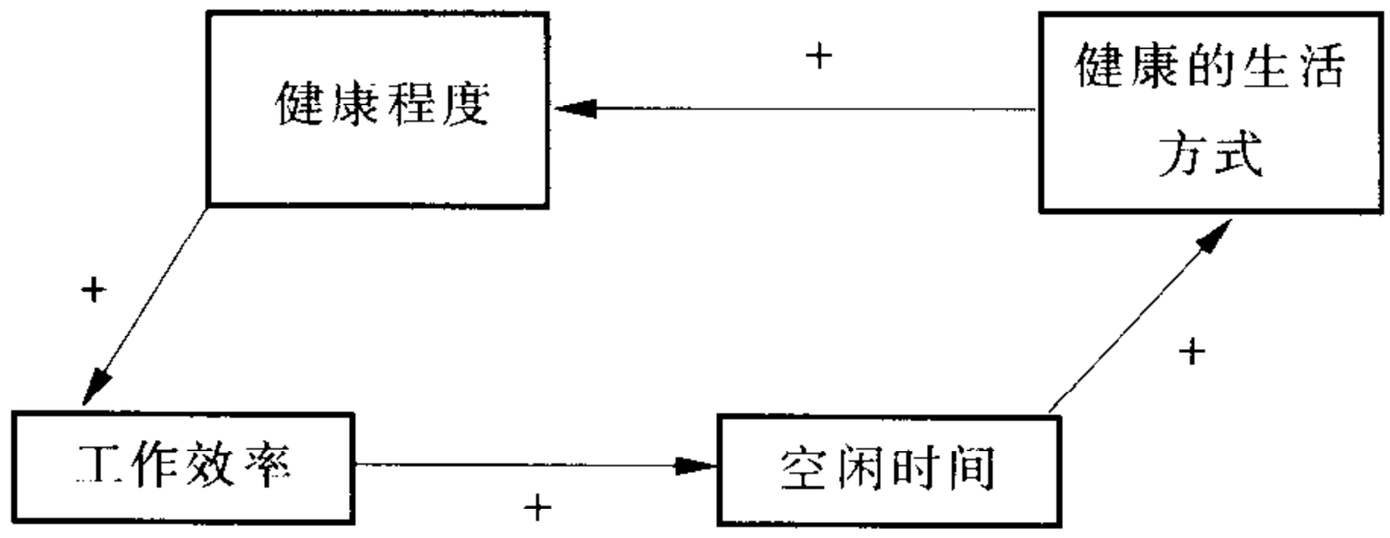


图 1　身体健康和工作效率之间的正反馈循环

可是，这样一种“正反馈”循环也能往另外一个方向发展。如果工作者的身体本来就不好，如此循环一番，那么肯定会导致更差的健康状况。为什么呢？如果分析一下这个循环，我们就能发现，身体不好，这样工作效率就不高（部分地因为大脑思考能力下降），这反过来就会使工作越积越多，因此我们就要加班工作，匆匆忙忙地吃垃圾食品，并且在各方面都忽视自己的健康。最后，这个循环像螺旋似地一圈圈转下去，我们的身体甚至比以前还要糟——除非我们能够适时地休一次假，或者管理层能够善解人意地干涉一下，才能打破这个循环。

但是这一类大脑功能紊乱还只是最表面的现象——就像脑壳被钢琴腿撞了一下的后果。大脑并不是那种载重的牲口，所以，它的负载能力也不能用一个简单的数字衡量。事实上，大脑是一种专门解决问题的复杂装置，对于它的作用，我们只有一些很模糊的认识。我们知道钢琴腿会让大脑停止工作，就像饥饿也会起到类似效果一样，但是这只不过就是“如果我们拔掉电源线，或者用锤子砸 CPU，计算机就会停止工作”那一类的知识。

Simmons 的那张个人化学清单上，列举了一些元素，能够体现大脑的一些多样性特征：

1. 表达性：至少能够用你的母语流利地写作和说话。
2. 思维性：能在做出反应之前几秒钟对问题加以衡量。
3. 智慧、通晓事理、火花[①]：“很难定义，但是如果一个人不具备这

① 火花：见“个人化学和健康身体”注③。

样的素质，就能非常明显地看出来——这个人也就会显得特别沉闷无趣。”

4. 兴趣的广泛性：能够巧妙地与人交谈，不会因为对话题没有兴趣，或者缺乏教养而造成冷场。

Simmons 的意思似乎是说，你可以在自己本来很无趣的本性上抹上一层“个人化学”的清漆。比如，“做出反应之前先沉思片刻，会给别人留下很有判断力的印象”——注意，不是判断力，而是判断力的“印象”。在这个分析层面上，所谓大脑的“化学”就由一些特定的规则构成，比如“在回答一个问题之前先数到 3，这样别人就会认为你深思熟虑”。

哦，是呀，几条这样的规则确实能让你在鸡尾酒会上展现不错的“个性”，甚至可以让你找到一份更好的工作。对于酒会来说，下一次不一定还会遇上同样的人，但是如果你找了一份工作，你就得多动动脑子了。因为在真实的工作岗位上，那层清漆很快就会销蚀掉，也就会暴露出下面不平和空洞的地方。如果你真地想变得更会表达、更加深思熟虑、更有智慧、更通晓事理、更才华四溢，那么你还是得在工作中多投入一些时间和努力。而你又从哪儿找这些时间呢？我的一些熟人，不是程序员，他们告诉我在他们认识的人里，程序员最沉闷无趣了，我自己当然不相信这个说法。每次我们开办一个“成为技术领导者”的研修班时，在星期日都会举办一个晚宴，这时的气氛比我自己参加过的任何艺术家酒会都更活跃、更才华四溢。不过，你只要听听程序员们实际都谈了些什么，你就会明白那些外行是什么意思了：从程序员们说话的音量和热情来判断，你会以为他们说的是伊特鲁里亚大理石[①]，或者是怎么种好甘蓝菜之类的话题，但其实他们说的都是计算机——愚蠢、乏味、过时的计算机。

我们自己当然知道，计算机并不愚蠢，也不乏味，当然更没有过时。计算机是一个永远迷人的话题，充满了光彩夺目的细节，也并不缺乏让谈话者做出指点江山的大判断的机会。不过，咱们最好还是现实一点吧，生活中有比计算机多得多的内容。而且，我们的大脑也还

---

① 伊特鲁里亚：Etruscan，一种已经消失的意大利古文明。

有很多功能，远比我们平常用于计算机工作的那些要多。其实，如果我们能够定期锻炼大脑的这些部分，让它们保持健康，那么它们也肯定会给我们很大帮助。是呀，只要我们沉重的工作负担允许，我们当然会这样锻炼的。这句话听起来挺耳熟？请看图 2 中的循环，它与图 1有很多相似之处。图 2 体现的不是大脑的生理健康，而是它的精神健康——考虑的也并不是它受过多少锻炼，而是它的那些素质受到了锻炼。

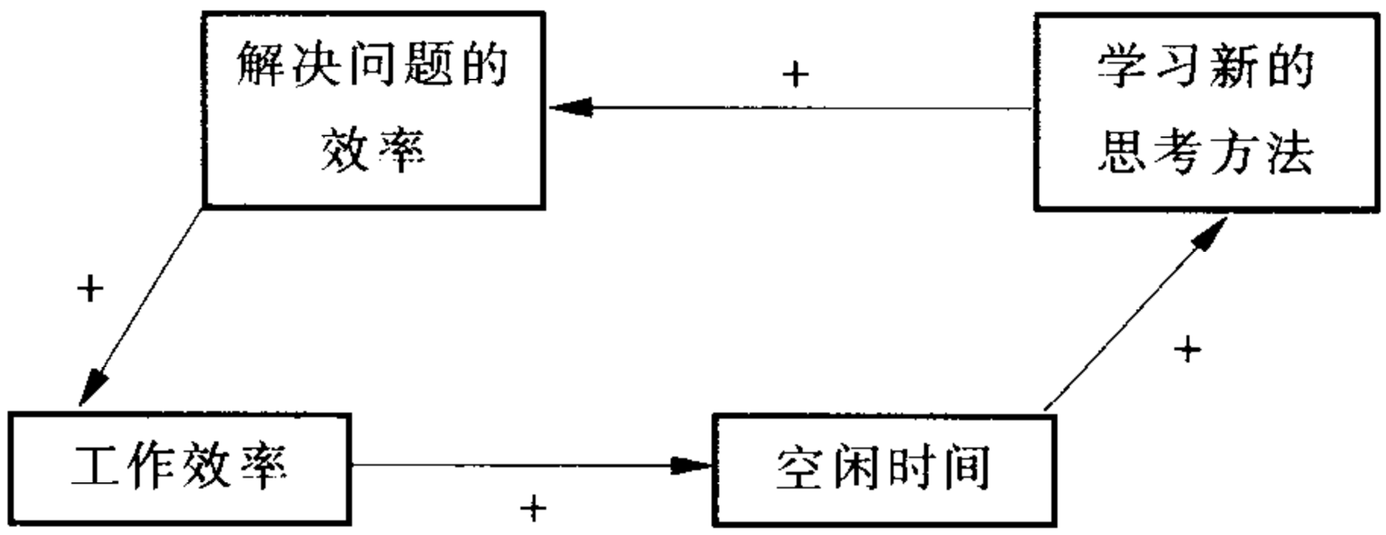


图 2　精神健康和工作效率之间的正反馈循环

在我们的研修班里，我们不断地注意到这样一个事实：如果一个人总在一个封闭环境下工作，他解决问题的方式也会变得特别单调死板。这是因为，如果人们发现在自己的环境中有一两种技巧特别有效，那也就会总是使用这几种技巧，忽略其他的所有方法。所以我认为，一个研修班之所以能提高工作效率，是因为每个参与者在其中都能看到别人解决问题的方式和风格。不过在实际工作中，图 2 的这种反馈循环是最为常见的。

月复一月，计算机行业的问题越来越复杂，所以，如果我们解决问题的效率仍然停留在同一水平上，那么很快我们的工作就会堆积如山。空闲时间越少，我们出外活动的机会也就越少，因此平常工作中不太用得上的那些大脑机能也就很难受到锻炼和刺激。这样，我们解决问题的效率只在一个狭窄的领域内增长，我们也都成为了非常狭窄领域的专家——这样我们遇到新问题就往往无从下手。

大脑需要刺激。如果你把自己锁进了一个“工作工作再工作”的模式里，你的大脑很快就会习焉不察——就像你不会注意到钟表的滴答声一样。所以，如果你想提高工作效率，你就要（听上去有点儿矛

盾)在工作中不那么一心一意地投入。你做的任何事情——任何能够刺激大脑的其他机能的事情——都能够提高你作为程序员或分析师的工作效率。我担保如此,无效可返还现金。

但是请注意,大脑必须得到刺激。在传统上认为能改善思维的方法中,并不是每一种都能够刺激大脑的。其实恰恰相反。很多程序员和分析师,为了让大脑得到新的刺激,就去大学里注册了课程。虽然一些人确实得到了课程的激发和刺激,但另一些人却没有得到,原因有二:

1. 课程很乏味,但他们还是坚持上完了——也许是因为他们的雇主为他们付的学费,而他们要是中途退出会很尴尬。

2. 课程与他们的本职工作太过相关——也就是说,跟他们上班时干的差不多。这本身并不是坏事,而且还能够教给他们很多有用的东西,但就是不能提供对大脑的刺激。

总体来说,最好还是在正规教育系统之外寻求这种刺激。比如说,试试改变你看电视的方式——并不一定非得看那些知识性的节目,但无论如何,看一点儿跟你平时看的不一样的东西。如果你根本就不看电视,那么你倒可以试试去看一次,这也算是一种刺激,虽然有可能是消极刺激。如果你平时除了工作手册什么书都不读,那么请在今晚回家的路上随便买一本廉价小说,读一读。但是如果你觉得乏味,就放下别读了。如果你平时读书,那就读点平时不读的东西,要么就在几天时间之内什么也别读,只是用你的眼睛、耳朵、鼻子感知周围的整个世界。另外,如果你非去上课不可,那就去学一些你的雇主永远也不会帮你付费的东西。这样一来,如果你觉得乏味就大可以退出,用不着害怕为此丢了工作。而且,即使你觉得课程很乏味,它至少还提供了你和同学之间的共同语言。这些同学里是不会有太多的计算机人士的。

我为什么总有主意

人家经常问我:“你每周都要写一个专栏,哪儿来的这么多主意?”我通常的回答是:“从你这样的读者那儿来的呀。”——这也确实是真相的一个主要部分。举例来说,本文就是受一个读者激发而作的,该读者写信给我,问了作为标题的那个问题。

主意是怎么产生的?基本上呢,主意来自 3 个源头:

1. 错误
2. 偷窃
3. 交配

那些彻头彻尾的新主意来自错误。我常常会打错很多字。有一次,我把“变化”(change)打成了“机会”(chance),这却刺激了灵感,产生了我正写的那本书里的整整一章内容。另一回,我发现一个客户在黑板上写字时,把“立即可用的系统”(turnkey system)写成了“火鸡[①]系统”(turkey system)。我把这个笔误放在了一次演讲的开头,这与演讲的主题——“现成可用的系统”的危险性(这一类系统有时与“脱出常规的系统”恰相对照)[②]——正好配合上。

但是这一类创造性的错误非常罕见。或者说,实际情况是我们很少主动利用自己的错误,也许正是因为与此相比,偷窃新主意还要容易得多。作为一个专栏作者,我当然有几千名读者,都在为我提供好主意。每隔一段儿时间,这些读者中的某个人读到我的一篇专栏文章,就会说:“嗯,这写得不错,可我自己在这个问题上的想法要比这好上一百万倍。我想我该给这个笨作者写封信,教教他怎样真正思考问题。”也不是所有的来信都像写信者认为的那样全是好主意,但很多信

---

① 火鸡(turkey)另有“无用、废物”的双关意思。

② “现成可用的系统”(off-the-shelf systems)是常见的一个说法,指该产品从货架(shelf)上拿下来就能用上。“脱出常规的系统”(off-the-wall systems)是作者仿照制造的讽刺性短语。英语“off-the-wall”有“奇思怪想”、“不合常法”的意思。现在的译法只是一种中文解释,若保持原味,或可作“下架系统”和“下墙系统”。

确实如此。这样一来，我就有了一个稳定的素材来源。

作为一个咨询顾问，我也能从客户处获得同样的收益，只是总不能带着打字机去客户处随听随写。我拜访的每一个客户，都热切地告诉我正在投入实践的所有新主意（作为交换，我也得给他谈自己的主意）。拜访了几个客户之后，我从他们那里获得的切切实实的主意，就比我整整一年用在客户那里的主意还多。而且，虽然我“偷了”他们的主意，我却一点儿也不内疚，因为我做的工作永远对得起他们付给我的费用。

当然了，我不会谈客户们视为“专有”的那些主意的。我总是以完美的服从心尊重保密协议，哪怕有一点儿违背客户意愿的可能，我都不会泄漏那项内容。我偷的那些主意，客户们总是认为小小不言、理所当然，甚至根本不把那当成“主意”。它们在这家公司里可能确实“小小不言”，但要是移植到合适的环境里，它们却能茁壮成长、开花结果，形成业务上的重大突破。

我的另一项有利因素，是我误解某个偷来的主意的本事，通过这“误解”，也就引入了一个“错误”。常常发生的是，这种“错误”却构成了最富创造力、最有价值的内容。实际上，有时我会把这个改头换面的主意用在它原先的主人那儿，这主人还会觉得它价值连城呢。我记得有一回，一群经理告诉我，他们计划用他们的那台大型计算机给刚刚买来的微型机编译程序。我以为他们说的是，用微型机给大型机编译——至少是在微型机上输入程序，再用微型机完成一些辅助、查错的工作。我把这个主意讲给同一公司的另一个小组，他们听后热情高涨，决定也应该给每个程序员配备一台微型机。由于程序员的打字能力都极差，配备了用于查错的微型机后，在大型机上调试就不再是整个研发工作的瓶颈了。

但是我们最终做到的比这还要好。我得知他们用微型机在模拟终端上训练数据录入员。于是我天真地问，他们为什么不用同一套软件来培训一下程序员们，提高他们的打字水平呢？他们听了这话，简直是大叫大笑，而且还揪头发，我赶紧制止了他们——因为我没多少

头发好揪了[1]。最后，我说服他们采用了这个主意，结果是他们的一些程序员打字水平大为提高，足以使用大型机上的在线功能了。（我是怎么说服他们的——这可是个商业机密，在这里不便透露。）

那么，归根结蒂，偷窃和犯错是两种优秀的主意来源，但大多数确实优秀的主意却来自**交配**——把两个独立的主意合成一个新的主意（而且新主意要比两个老主意都好）。鸡蛋也许让人喜欢，白糖也许让人喜欢——但一个好的蛋白酥则让人**爱**。

在生命系统的遗传学中，错误、偷窃和交配的重要作用得到了最好的例证。遗传提供了下一代生命赖以构建的信息（主意）。有时候，突变（错误）会引入一个新主意，但在大多数时候，染色体的复制都是非常精确的。也就是说，我们从父母那里偷来了大部分遗传材料，这也无可厚非，因为父母的遗传材料则是从他们父母那儿偷来的。而且，正如其他对“主意”的偷窃一样，遗传信息的偷窃并不会使原主失去自己的“主意”。

但是，除非你是单细胞生物（有时候连单细胞生物也概莫能外），你的大多数“原创”的遗传信息（主意）都是通过重组窃自父母的遗传材料而成的。也就是这个重组过程使你独一无二——就像其他每个人都独一无二一样。（先不考虑同卵双胞胎的情况。）

父母是如何组合遗传信息，形成了那个绝妙的新人（也就是你）的，这里的细节也无须过多考察了。指出一点就已经足够：很多人都认为这个组合过程是整个生命循环中最令人愉悦的部分。但是，唉，不是每个人都这么认为。也许这也就能解释，在想新主意的时候，为什么我们常常遇到困难。我们在学校的时候（至少是**我**在学校的时候），老师灌输给我们一整套戒律，好些事情都被标明了是邪恶的，一定不能做，但其实往往就是这些事情才能创造出好主意。

我们交作业让老师打分数，于是发现任何错误都被惩罚以低分。如果为了避免错误，我们从书中剽窃，或抄袭其他同学，那只会使惩罚加重。一个拼写错误，或读音错误可能让我们丢掉 10 分，或是让我们

---

① 没多少头发好揪：为了验证这个事实，读者可以查看任意一幅 Weinberg 成年后的照片。

放学后留下来把正确答案在黑板上写 100 遍。但要是抄袭的话，老师就会说我们是“骗子”，而且还会送去见校长。

至于“交配”，这么说也许就足够了：要是学校为这个抓住你，那么上述对欺骗的惩罚与此相比简直像是诺贝尔大奖。

但别搞错。我可不是那种怯生生的自由派，不相信惩罚有任何效用。惩罚是最有效的教学方法之一。它教给你如何避免惩罚。那些在学校里老是因为错误、偷窃或交配被惩罚的人，不太可能想出了不起的主意——因为他们不会从对惩罚的恐惧中总结。

说到我自己，我在学校的时候当然也犯了该犯的那些错误，甚至偶尔也骗过几回人。这样，我至今仍对出错感到恐惧，对抄袭他人工作也总是放不开手脚。

但是，唉，那时我真是个害羞的小子。我不曾有幸和弗里达、瓦莱丽、厄丽思、丽碧，或与我同校的那些可爱女郎中的任何一个钻进灌木丛温存。更别说为这个被抓到了！结果呢，对“把两个好主意放在一起合成一个了不起的主意”这回事，我也就从来不害怕了。

所以我对任何以创意为生（为死）的专业人士有如下忠告：少年时要过得清白、健康——或者，至少在做事时不被人抓住、惩罚！

着急的海狸和聪明的刀子：一个寓言

今天不是海狸干活儿的日子，至少那只海狸自己是这么想的，因为他正在修一道新的水坝，但是那最后的一棵树落进小溪时也掉到了相反的方向上。[①] 他把树上那些旁逸的枝干都去掉了，但是他也知道，即使是那个光秃秃的树干，他也没力气把它从陷进去的烂泥里拖出来。他必须做点儿事情，但是该干什么呢？再伐倒另一棵树？但是在建水坝的周围，只有这一棵树的大小合适，所以，即使他咬断了另外一棵树，还得要别人帮忙才能把它拖到小溪的这个位置。为了找人帮忙，他就得跑去找其他海狸，等找到了可能天也就黑了。可是到了明天，他想，就会有很多水从坝上流过去了，上游可能会有一个月都不下雨。他决定四处跑跑，尽可能地找人来帮忙。

他正在匆匆忙忙地在林子里面跑，突然觉得喉咙被什么东西抓住了，只好一侧身子，才免得被活活勒死。揉着自己酸痛的脖子，他这才发现那是一条牵住帐篷的绳子——有一些野营者跑到这一带来了。“这些人呀，”他一边这么想着，慢慢也恢复了镇静，想要继续忙自己的事情。

突然，从上方传来了一个调门挺高的声音：“当心一点儿好不好！你差点儿把那条绳子割成两段，如果真那样的话，整个帐篷都要垮掉——也得把我埋进去。”

海狸抬头一看，原来在帐篷的柱子上挂着一个闪闪发光的金属物体。“你是谁呀？”他问得挺有礼貌，但是也挺不耐烦。

“显而易见，”那个东西尖锐地回答说，“我是一把刀子。那么你又是谁呀，这样忙忙叨叨地乱撞，险些把我们的帐篷都给弄坏了？”

“我是一只海狸，但是我刚才确实没认出你，刀子先生。我的眼神不太好，另外呢，我恐怕以前还没见过刀子。你是做什么工作的？”

“哦，我要干的可不太多，”刀子打了个呵欠。“大部分时间我就在

---

① 海狸这种动物很会把树咬断，使之在河流中形成水坝。

这儿晃悠着，什么也不干。你知道，我可是很敏锐的[①]，要是也像你那样跌跌撞撞地乱跑，那这儿的所有东西都得让我切成碎片。哎，就拿这条帐篷上的绳子来说吧。如果我要像你那样撞上它，它就会断成两截、从此报废。所以可不能乱跑乱撞，得谨谨慎慎的。”

“恰恰相反，”海狸回话说——他想要赶紧结束这次交谈了——“不应该整天晃悠，什么都不干。总是有工作要做的。如果我不抓紧时间工作、工作、再工作，那就什么也干不成。”

“好吧，随你的便，”刀子斩钉截铁地说，“但是如果你换个地方忙乎的话，我会很感激的。我可没想对你不礼貌，跟你聊天挺不错，可是我确实要小睡一会儿了。”

海狸一点儿也没觉得人家怠慢了他——实际上，他对能够终止谈话，继续去忙那件大事，高兴还来不及呢。即便如此，他还是忍不住要来一句临别赠言：“你要是不改掉这个晃悠的毛病，开始努力工作，那你也就永远切不开芥菜[②]。”而且，他自己还琢磨着，游手好闲总会惹是生非。看看这个刀子给我捣了多大的乱。我在这儿跟他荒废了这么多时间，都没法找到足够的帮手来修水坝了。

他这么想着，就跑回到小溪边，因为即便是无事可做，他也闲不下来。快到岸边的时候，他跑得太快了，以至于都掉进了水里——溪水已经涨上来了，他眼神不好，又没看见。他从水里游到水面上，四下看看，想找那根不服管的树干，可是树干已经不在泥里陷着了。就在他跟刀子聊天的那一会儿工夫，水塘里的水也涨了起来，那根树干也浮出了淤泥。小溪中的水流都争着往水坝的缺口处流，这样就正好把木头送到了合适的位置上。工作干完了，海狸也没有道理不享受一下、游个泳——既然无论怎样他也已经在水里了。

教训：锋利的牙齿和锋利的刀刃都是很好的工具，但是如果不假思索地使用，它们也会变得很危险。三思而无为，总比不思而妄为要好。

换句话说：知道什么时候让事情自己水到渠成，这是最高的智慧。

---

① 敏锐：原文为 keen，刀子说了这个双关语，既表示刀刃很“锋利”，也还有“热心”、“热衷”的意思。

② 切不开芥菜：这是海狸的俏皮话，“切开芥菜”原文是 cut the mustard，本意是“达到标准、符合要求”。

第5章

为什么不是人人都能理解我

输出过载

在我们研习班的第一次“如何解决问题”讲座上，我遇到了一个很罕见的情况。一个学生——非常聪明，非常能言善辩的一个人——对那个待解决的问题给出了一个特别确切，表达特别清楚的解法。在这样一个班级环境中，如此确切的解法实在罕见，而能讲出来就更罕见了。

提出解法的那个学生——就叫他 Mack 好了——是一个“大组”的成员——全班分成两个大组，互相竞争。他知道，如果两组人合作进行的话，他的解法就能够“获得最高的成果”。经过一段讨论，他说服了本组的几个人，他们相信他可能掌握了这个问题。因此他们表示，如果他能够说服另外一组的人也按他的办，那么他们也都会服从他调遣。

在这个情况中，难得的不是 Mack 认为自己有了一个完整的解决方案——每次这样的模拟竞赛都有很多人这么想。难得的是，他确实有这样一个解决方案。而且，他还能用精确的数学概念表述这个解决方案。看着他往黑板上写出那些公式，我才觉得有点不妙，因为他用的是 APL 的标记法。有些学生懂 APL，但是剩下的就不懂了。虽然大家都是资深程序员，Mack 要是解释一下 APL 标记法也很容易，但是他因为自己的想法太过兴奋了，以至于居然没有意识到他已经丢掉了 3/4 的听众。

他一边往下说，听众们一边提了一些小疑问，但是他却没有真正停下来回答，让大家满意。这道题目是我出的，所以我当然能明白他的解法，但是我也成心不告诉他和其他人这个解法到底对不对。毕竟，在生活中不太可能有一个全知的裁判坐在边线上，等你拿出了正确答案他就欢呼。

既然没有裁判，那么 Mack 的解法的成败就全靠他自己的演示了。结果失败了。而且随着这次失败，全班人对 Mack 的信任也就泡汤了。这一周的后几天，他都在尽力拯救大家对他的信任，可是无济于事。他越是失去信任，他就越是努力在每一个练习中讲解自己的好想法。可是他越是努力，他说得就越快；说得越快，听的人也就越少。

我想告诉 Mack 一些东西，帮助他重获全班同学的接受，但是我也想不出合适的办法。课后，有几个学生坐在一起，在讲从前的黄金岁月里的亲历故事。不过他们中间没有哪个人在“裸机”上工作过——连帮助处理 I/O 之类的杂务的操作系统都不带，所以我就来跟他们解释解释，在这么一种环境下编程该多有乐趣。

我给他们讲自己在 20 世纪 60 年代早期，用 PDP-1 型计算机[①]做心理学实验的事。为了让输出的磁带能够在 IBM 环境下读取，我就得控制从这台计算机中输出的每一比特数据。我得编程序来计算每个字符的校验位，还有每个记录的校验位。学生们听了很感兴趣，但是当我告诉他们，我还得编程控制每一个字符传送到磁带的时间时，他们就不由得惊叹了。

换句话说，为了保证字符的间距符合 IBM 的标准，我的程序必须精确控制循环次数，这样磁带才能恰好移动那么多距离。不仅如此，在记录的结尾处，还得再多延时一会儿，才能满足校验位的正确间距。然后，再空上记录间距的一半距离。当下一个记录开始时，我的代码再控制机器写下记录间距的另一半。

有一个学生就问了：“如果你的时间控制出问题了，又会怎么样呢？”我解释说，如果字符输送速度不够快，那么数据记录就会间隔太大，在 IBM 环境下就读不出来，但是他考虑的还不是这个问题。“我

① PDP 系列是 DEC 公司在 20 世纪 60 年代生产的小型计算机。

的意思是，如果你的字符输送得太快，磁带跟不上，又会怎么样？”

我想找一个最好的解释办法，这时 Mack 的形象出现在我的脑子里。“嗯，”我说，“那就像星期天晚上 Mack 对全班讲他的解法时的样子。输出的字符会一个叠着一个，把磁带搞得一团糟，但是程序也感觉不到有什么问题。直到你想用这个磁带，再现上面的信息时，你才会发现当时输出设备过载了。”

“就像 Mack 不知道我们没跟上他的思路一样？”

“一点儿不差。”我很遗憾，没能像这样轻松地跟 Mack 本人讲清楚。

Mack，和很多计算机程序员、分析师一样，智商非常高，对计算机以及其他知识都所知甚多。他很喜欢参加智力测验，也许这能够证明他是一个天赋很高的人——虽然他从同事们那里得到的验证可不一样，因为人家都不听他的。智商高，就像 CPU 的运算速度快一样。对于解决问题，这是一个非常了不起的资源——只要这个问题不涉及多种输入输出就可以。

但是，当我们把一个解决问题的想法转化成实际的解决方案时，有时必须要和他人交流，这时太高的“内部运算速度”会导致过载。在一生中，我们大部分人都有遭遇“过载”的经历。这常常发生在学校里，当某位教授认为自己站在讲台前的目的就是让大家瞧瞧这个教室里谁智商最高的时候。

我们遭遇过载的其他场合包括在书本中，尤其是当书的主题与我们的主业不太一致的时候。有一次艾尔伯特·爱因斯坦解释了为什么总会发生这种情况：

> 大多数号称是写给外行的科学著作，其实主要追求的是给读者留下深刻印象（“真吓人！”“我们的进步多大呀！”，等等），而不是把科学的基本目的和方法给他解释清楚。所以，一个比较聪明的外行读了这么几本书之后，就会完全失去信心。会做出这样的结论：我的头脑太差，最好还是别看这个了。

如果那本书的作者不是一个“权威”，那么我们可能会给自己个台阶下，反而得出这样的结论：那位作者，而不是我们自己，头脑太差了——这也就是我们读使用手册、计算机学术刊物时常常产生的感

受——这也就是我们听 Mack 这样的人讲话时常常产生的感受，其实 Mack 的“纯粹智商”特别高，但是他的表达能力要比这个“纯粹智商”低好几个数量级。

如果 Mack 遇到了一个计算机系统，有上面说的那种 CPU 太快的毛病，他肯定马上就能知道怎么解决这个问题。他当然不会花时间让程序运转得更快，但是当他自己就是这么个系统时，他却恰恰会越弄越快。

Mack，还有很多特别聪明的年轻计算机工作者，所需要的，是要改善他们的“输入”、“输出”能力，从而与环境更加平衡协调。在谈话时，在写作时，他们可以减少输出的数量，然后把他们简直过剩的运算能力用在提高质量上。但是我也不是说，在别人说话的时候，你就应该静静地坐在那里一遍一遍地琢磨“对方闭嘴以后我该怎么说才最好”。

要利用运算能力改善提高输出质量，一个办法就是把这种能力用来处理输入。我们一般称此为“倾听”。有时候，很难知道一个人是不是在听——还是仅仅在那里等着夺取谈话的控制权。Mack 常常不让别人把话说完，所以别人就知道他并没有听人说话。他似乎是在说，既然我已经知道你要说什么了，而且早就准备好下面的好几个步骤了，那么干吗还要让你说完呢？

当然，很可能 Mack——像爱因斯坦说的那样——只是想给同学留下深刻印象。如果事情确实如此，那他还是失败了，因为开头几天之后就没有多少人听他说话了。他留给他们的印象，就像那台 PDP-1 给我的印象一样：要是我没见过别的，我也许会以为它很了不起。

“如果你想让别人喜欢你，”俄罗斯有一句古老的谚语这么说，“那就让他们帮你个忙。如果你想让他们恨你，那就帮他们个忙好了。”要想让别人对你的智力产生印象，其实也跟上面的情况类似。如果你想让人认为你聪明，那就认真听人说话，仔细理解人家的意思。如果你想让人认为你傻，那就不妨经常用你的高见打断别人的话。

无论如何，我还是希望你会打断自己正做着的任何事情，来听我的高见。

重写和 H 配方测试[①]

不少人，其中还包括 Edsger Dijkstra[②] 这样的业界权威，都曾经断言，母语的能力——也就是读和写的能力——对于程序员来说是一种最重要的素质。虽然我并不相信哪一种单项素质能称得上“对程序员最重要”，但是我确实认同他们对语言能力重要性的评价。

不幸的是，没有多少出版商赞同这一点，他们对编程图书作者的语言能力并不看重。如果我们的教科书能够更关注语言明晰的重要性，也许编程领域就会吸引更多的好作者。如果程序员们能从作者那里看到语言明晰的好榜样，也许他们中更多的人也就会花些力气，使自己的程序也变得明晰易懂。

写英语文章和写 COBOL 程序是非常相近的活动，但这并不是因为 COBOL 语言“很像英语”的缘故。其实一切优秀写作的秘诀都是**重写**。英语如此，COBOL 也是如此。甚至连 APL 也是如此。除了天才，没有人第一稿就能写得完美无缺。所以，作家或者程序员需要的，就是欧内斯特·海明威[③]称作“内置大粪探测器”的一种东西。

让我们来看看，海明威说的是什么意思。试读以下自然段。然后，如果你愿意的话，用原文一半的字数重写这段话。

> 正如我们所知，所有 COBOL 程序都有 4 个层次。其中前 3 个层次——“标识”，“环境”和“数据”——确定了程序的多个方面，但并没有描述程序如何完成处理操作。只有在“过程”这个层次的指令中，才实际确定了在程序的执行过程中，各个处理步骤如何发生。所以，也只有通过“过程”层次的指令，程序员才能与计算机沟通，指定计算机完成哪些操作。

下面是这段话用一半的字数重写的结果：

> 在 COBOL 程序的 4 个层次中，前 3 个——“标识”，“环境”和

---

① H 配方，原文是 Preparation H，是至今仍然流行的一种痔疮药。

② Dijkstra(1930—2002)，荷兰最著名的计算机科学家。

③ 海明威(1899—1961)，美国作家，诺贝尔文学奖得主，以文笔洗练著称。

“数据”——并不描述处理操作。“过程”层次，也只有“过程”层次，才包含实际指令，确定各个处理步骤。

我重写这段话费了 5 分钟时间，我希望大家看了这第 2 段话，也会同意这 5 分钟花的值得。不妨这么考虑：如果出版了一本书，其中包括的是前面那个样本，而不是我修改的那一段，有 12 000 个读者每人就会在这段上浪费一分钟时间。所以我投资了 5 分钟，就能节省大家 200 个小时。这还不包括如果拙劣文笔导致技术误解，那样损失的无数小时。

所以关键的问题就是：…**种读物将来会让人们花多少时间阅读？**

在过去，我们曾经把汇编程序重写十多遍，就为了节省 3 次机器循环。现在我们一般不会这样做了，因为现在的主要成本是人力，不是机器。那么，如果重写一下各种佶屈聱牙的文献（比如下面这一段“解释性”的注释），又能节省多少“人力循环”？

THE FOREGOING DEVICE WAS NECESSARY IN ORDER TO PERMIT THE SAME PROGRAMMING TO CALCULATE THE STATIONARY AND THE STABLE POPULATIONS. ON THE FIRST ROUND THE GIVEN VALUE OF R WAS ENTERED. ON THE SECOND ROUND THE VALUE IN R WAS SAVED IN RR, R WAS SET EQUAL TO ZERO, AND THE CONTROL TRANSFERRED TO 15. THE PROGRAM RECOGNIZED THAT IT WAS ON THE SECOND ROUND BY FINDING ZERO IN R, AND IT THEN TRANSFERRED TO 23, HAVING PUT THE STABLE POPULATION IN THE ARRAY VKKA AND THE STATIONARY IN VKK, FROM WHERE THE STATIONARY IS TRANSFERRED IN THE LOOP AT 25 TO VLL. ①

我就放你一马，不让你读以上注释试图解释的那个程序了。我敢

---

① 参考译文：为了使本程序能够计算静止人口和稳定人口，下面的策略必不可少。在第一循环中，输入变量 R 的值。在第二循环中，R 的值保存在 RR 里，再设置 R 等于 0。然后控制跳转到第 15 行。程序判断 R 等于 0，因此知道这是第二循环，然后跳转到第 23 行，把稳定人口保存在数组 VKKA 中，静止人口保存在 VKK 中，然后在第 25 行，再把静止人口从 VKK 转到 VLL 中。

保证，你可以想象得到，一个负责维护的程序员为了读懂它要花多少时间。

别上当——不要重写这样的注释。所谓庆父不死，鲁难未已——只要还有牛，就有 B. S. [1]。这么误人误己的注释，倒还不是病症本身：它们只是症状而已。如果我们打开代码看看，我们就会发现那简直是令人作呕的一团糟，根本没法印到任何体面的出版物上。这也就是那种“探测器”会让我们重写的东西。

一个程序、一段注释、一篇描述如果需要重写，那会有很多症状做出提示的。大多数症状都落在“效率”名下：如果读它花的时间要比写它还长，那就应该重写，要么干脆扔掉，从头再来。

不妨用这个“效率”原则考察以下这段：

> 为了排除程序在初始化一种 IO 操作时需要周期性查看操作是否完毕的费时过程，也为了提高整体运行效率，引入了中断机制。

这是一段从使用手册上抄下来的话，如果你是这种手册的一个典型读者的话，你就要来回读上好几遍，才能理清这里的意思。到你弄明白的时候，花的时间肯定要比作者写这句话多得多了。只要用同样多的时间，你就能把它改写成这么一段一目了然的话：

计算机之所以需要中断机制，有两个主要原因：(1)使程序不再需要周期性查看并行过程是否结束；(2)提高整体运行效率。

当然，当你把一句话改得更加易懂时，可能就会发现这句话根本就不对。比如上面这段话吧，我们可能就不同意它对引入中断机制原因的分析，或者也可能会认为第二点其实就是对第一点的一个重复。

“不对”当然也是需要重写的一个标志。有时候，拙劣的文风直接导致了错误，比如下面的例子：

> 但是，就像人类记忆一样，数据可以放入计算机的内部存储，并且可以在以后的某个时间被重新召回。

---

① B. S. 是“牛粪”(bullshit)的缩写。作者为了雅驯故意不写全文，译者就没这么好运气了。这个词在英语里也有“胡说八道”、“废话”的意思，还可以跟上面海明威的名言互相对应。至于为什么说“只要牛不死”云云，当然是指，代码之于注释，正如牛之于 B. S.。

这本使用手册的作者的意思，也许并不是说“数据”就像“人类记忆”一样，而是说，计算机的内部存储就像人类记忆一样。所以，或许最好还是把人类记忆的神秘现象留给生理学教科书，把这一段简单地改成：

计算机的存储介质的特点，就是能够先接受数据，并使数据在此后的时间里随时可用。

在程序中，就像在英语文章中一样，如果行文不够直截了当，净是兜圈子，那也是需要重写的一种迹象。程序中的兜圈子可以按照以下症状判断：

1. 引入多余的临时数据元素。
2. 循环中包含笨拙的异常，或者包括复杂的终止条件。
3. 反复给同一些数据项目赋值。
4. 重复的语句。
5. 开始或结束处过多的辅助操作。

在日常英语中，我们也能发现类似的标志，比如说滥用代词，用语重复，不善于开场白或者不善于结束。不过，在英语里有一种句法结构几乎肯定是“兜圈子”的特征——被动语态。上述有几个例子，我们就能够通过被动语态一眼看出毛病。当然了，有时候也是成心用被动语态，比如说在需求规格书或者销售文档中，因为那些作者不愿意承担任何责任。

在广告里，我们发现了好些了不起的被动语态例子，尤其是电视广告里。下面这个例子是西北大学的 Bob Finkenaur 首先发现的。“H 配方”在美国很流行，尤其是那些专业人士，比如程序员呀，分析师呀，每天都会在椅子上坐太长时间，所以往往需要这一类治疗。H 配方的一条最受欢迎的广告词是这样的：“H 配方经医生测试，有助于迅速暂缓直肠组织的不适。”

对于患者，这听上去不错，但是仔细分析，我们就发现这个 H 配方：

1. 经过医生测试，但不一定经医生认可。
2. 有助于，但自身不直接起作用。
3. 迅速，但不是立刻。

4. 治疗不适，但并不治疗带来不适的生理问题本身。

5. 给予缓解，但不治愈。

6. 暂缓，而不是永久解除。

下次要是有人向你推销硬件或者软件，你也可以对它做一次“H 配方测试”。其实，为什么不把所有这些测试标准放在一起，就称作“H 配方测试”呢？每次你读到什么写给别人看的东西，都先问问自己：

1. 它是不是读下来比重写一遍还费劲？

2. 它是不是不正确？

3. 它是不是容易误导？

4. 它是不是会让你感到一种只有 H 配方才能缓解的不适？

如果以上任何问题的答案是“是”，那就重写一遍。也许这会影响 H 配方的销量，但是整个世界都会因此爱你的。

但是，别对我这本书作这个测试。如果有什么问题的话，毫无疑问都是编辑错误，或者是排字错误。

说你所想，要么想你所说

“那么你就应该想什么就说什么，”三月兔继续说。

“没错呀，”艾丽丝连忙回答，“至少——我说什么就想什么——这是一回事，你知道。”

“根本就不是一回事！”帽匠说，“不然，你大可以说‘我看见了我吃的东西’和‘我吃了我看见的东西’也是一回事了！”①

艾丽丝就是个标准的程序员。我在读代码的时候，常常就想到艾丽丝在疯帽匠的茶会上说的那些天真幼稚的话。很少有程序员懂得“说其所想”和“想其所说”的区别。比如，有一次我在一段 PL/I 程序中发现了这样的代码：

```
        I=1;
XYZ:A(I)=B(I);
        I=I+1;
        IF I<21 THEN GO TO XYZ;
```

这段代码想说什么？说到底，问“这段代码想说什么”又是什么意思？一段代码最明显的“意思”，就是它引发执行的指令——也就是它想对计算机说的东西。但是比这远为重要的，则是它想对别人说的东西——总会有人是第一次读这段代码，正费力去理解它。

要花多长时间才能理解一段代码，我对这个问题特别感兴趣。在教学中这是个重要问题，但是对于日益增长的程序维护工作，它还要重要得多。代码有自身的生命周期，首先是被一次写下，而后可能会被上百次地阅读。所以，如果代码所说的和它想说的不一致，那么累积起来，浪费在误解中的时间可能要以年来计算。

上面那段代码难懂，有好几个原因，但是大多数原因都可以放在“为什么就不能换个方式写”这个问题下讨论。或者，还可以问，“如果想说的是别的东西，那么为什么不直接说别的东西？”

---

① 这是英国著名童话小说《艾丽丝漫游奇境》中的一段。在英语里，三月兔和帽匠都用来指“头脑癫狂”。

比如，这段代码似乎构成了一个循环，循环变量从 1 累加至 20。但是 PL/I 正好就有一种循环控制结构，也就是 DO 关键字，它就是特别为了这一类循环来设计的，就像

```
DO I=1 TO 20;
    A(I)=B(I);
    END;
```

如果程序员没有选用这种特定的形式，读者就肯定会问，“是不是还有别的用意?”也许行号 XYZ 还用于其他目的，比如在循环之外的某个程序分支。如果是那样，则采用这样的结构还是合理的，因为在 PL/I 中，程序分支不能跳转到 DO 循环的内部。或者，也许还有其他微妙的原因，但是如果不加以帮助，我们就看不出来。但是如果那个程序员想说的就是和那个 DO 语句一样的意思，那么她可就“说非所想”了，这样也给我们带来了不小的麻烦。

甚至那个 DO 循环可能也是“说非所想”。那个数字 20 有什么特别含义？是不是程序员想说的其实是

```
DO I=1 TO N;
    A(I)=B(I);
    END;
```

其中 N 就是要赋值的元素个数？在这种形式中，代码说的就是：把数组 B 的第 1 至 N 号元素的值按顺序赋给数组 A 的相应元素。这相当清楚，但是也许，程序员本来还有别的意思。

或许，程序员想说的是

```
DO I=1 TO HBOUND(A,1);
    A(I)=B(I);
    END;
```

这里的意思则是，按照数组 A 的元素个数，用数组 B 给数组 A 赋值，从第 1 号元素开始。这样我们就不用绞尽脑汁来理解 20 或者 N 的特别含义了。而那个程序员写的程序，既说的太多，又说的不够。

但是等一下！这个数字 1 是怎么回事？在 PL/I 里，数组下标不一定从 1 开始，所以也许那个循环还另有用意。如果负责维护的程序员不够留意，这简直是一个陷阱，但是也许它的意思就是“A 中元素的

序号”。如果它想说的就是这个，那么那个程序员本来应该这么写的：

```
DO I=LBOUND(A,1) TO HBOUND(A,1);
    A(I)=B(I);
    END;
```

这样读者就确定无疑了：这就是按照 A 的元素个数来给该数组赋值，无论该数组的上下界究竟是多少。

但是还有问题。这么一个循环本可以写成：

```
A=B;
```

为什么不这么写呢？读者大可以问，“如果程序只不过是在数组之间赋值，为什么作者不直接使用数组赋值语句呢？”我们是不是要假定，作者想的和说的不一样？如果我们在这里这样假定了，那么在别处为什么不也这样假定呢？

甚至连这个简单的语句

```
A=B;
```

也还给误解留下了空间。如果我们是为了并行处理写这个程序，那么是不是允许同时给所有元素赋值呢？PL/I 的定义说“不是”，因为 B 中的元素一定要按一定顺序给 A 赋值。我们可能不在乎这个顺序，但是也许我们还是要在乎的——如果发生了中断的话。在这里，PL/I 并不能真地让我们说我们想说的意思——这意思本来应该是

```
A=B,无需特定顺序;
```

同样，似乎数组 A 与 B 很可能有同样的大小，但是如果只考虑这个语句的话，这也不是必定如此。只要 B 的上下界能够包容在 A 的上下界里，那就允许赋值，所以也许那个程序员的本意是，一般情况下 B 要比 A 小。为了验证这一点，我们还是去看看源代码的其他地方，比如声明 B 和 A 的部分。

在原本的这个程序中，A 和 B 声明时的维数是一样的，但是分别使用了两个语句，大致如下：

```
DECLARE A(N) FIXED;
DECLARE B(20) FIXED ...
```

N 的定义在这段过程之外，但是当这个过程被调用时，N 被赋值为 20。这样就都吻合了，但是只要其中哪一个地方变了，如果负责维

护的程序员不留意，这些程序就会变成一个陷阱。

实际上，我们很有理由相信 N 的值是会随时间变化的，而如果它真是如此，则程序中至少有 3 处应该保持与它变化一致。A 其实是一个传递给这段过程的参数数组，它的维数是在调用时传入的。所以它的声明本来应该是

DECLARE A(*) FIXED;

但是，B 又是怎么回事呢？我研究了声明 B 的整个语句，才发现是这样的：

DECLARE B(20) FIXED INITIAL ((20) 0);

这意思就是说，B 里的元素是 20 个 0。啊哈！看来 B 的唯一用途，就是在过程一开始的时候，给数组 A 的所有元素赋值为 0，这样，大可以完全取消这个 B，用以下形式更好地表达：

A＝0;

这样写出来，代码就消除了大部分其他问题。还不算十全十美，但是你可以看出，它已经多么接近矮梯胖梯[①](他可是最了不起的一个程序员)说过的那个理想："当我使用一个词，它的意思就恰恰是我想要让它说的——不多也不少。"

如果我们有了更多的矮梯胖梯，也许废铜烂铁似的程序就会更少。

---

① 我们在《个人化学和健康身体》一章里已经见过这位矮梯胖梯了。在《艾丽丝镜中奇遇》中，这是一个身材像鸡蛋的人物，很有些自命不凡的隽语。

误诊病理学

虽然我们可以自以为摩登，自以为免于一切迷信玩意儿，但其实我们确实有一些古怪的礼仪。这里最奇怪的一个，是建立在一种偶然情况上的：人有十个指头。所以这么多年以来，这个偶然情况也就导致我们的先辈发展出了一套基于数字 10 的计数系统。

正因为有了这套计数系统，每过 10 年，年份都会以 0 结尾，另外倒数第 2 位数字也要增加 1。这种变化就让一些系统设计者遇到了麻烦，比如，当 1979 年转到 1980 年时，我的几个客户就陷入了困境。我猜，当 1999 年转到 2000 年时，会有更多的人要焦头烂额——如果到那时我们的计数系统还没有变化的话。

我不太相信计数系统会变化。我们已经对它很熟悉了，而我们不喜欢那些陌生玩意儿。另外，大多数人——除了系统设计者之外的大多数——都听说过怎样递增倒数第二位数字。事实上，普通人对这样的变化很感兴趣，满怀期待。

数字 10 的这种神奇的重要意义在我们的一生都有体现，因为对重要事件（出生、结婚、毕业等）的纪念，一般都按这个数字来安排。而且，因为 10 碰巧能被 5 整除，每隔 5 年的纪念似乎也沾上了一些灵性。比如说，每隔 5 年或 10 年，大家都要礼仪性地纪念高中毕业或者大学毕业，这就叫“班级重聚”。虽然这种仪式很难让人忍受，不过大部分人都不会在同一年里赶上两回，因为按照美国的教育体系，大多数人在高中毕业之后要上 4 年大学。虽然传统常常很无聊，但是在我们不注意的时候它也会做些好事。在“班级重聚”时，我们往往会碰上飞黄腾达的校友，跟他们一比简直要自惭形秽，幸好有数字 5 帮忙搭救，这样我们在同一年里不太可能有一次以上这样的重聚机会。

唉，对我来说就不是这样了，我上完大学用了 5 年，所以每隔 5 年，我就要与两次“重聚”奋战。今年我的 5 年债期又到了。我知道，如果真的参加了这两次纪念会，我不大可能活着回来，但是为了表示我对大家这种可敬习俗的亲和，我也确实觉得自己应该做点儿什么。

我决定花 15 分钟，回想一下自己在学校里学到了哪些东西——这也算是为了班级重聚写的一篇很特别的专栏文章了。可我实在想不起来自己在学校学到什么了。也许这已经深深埋藏在我的体内，甚至织入了我的每一根纤维，可总的来说，我没法举出任何一样东西，说："这是我在学校学的！"只有一个例外！虽然我挺不愿意承认，但是我记得很清楚，有这样一个课程，我现在每天还能用上其中的教诲，我也觉得自己上了这门课是再走运不过了。

不，这说的不是我第一个计算机课——吾生也早，我上学的时候，连计算机课都没有。不，这说的也不是基础数学课程。（对，你们这些促狭鬼，我生下来的前几年人家就已经发明数学了。）也不是对那些不朽文学作品的研究。也不是我上过的那一种心理学课程，因为我第一节课听了 20 分钟就逃走了。不，我现在满怀感激回想的，可不是这些冠冕堂皇的课程，而是一种简简单单、普普通通的，名字就叫"科学希腊语"。一个挺和善的老绅士发明了这门课程，他本人原来是古典学系的，可是那时古典作品无人问津了，所以为了维持生计，只好想出了这么个主意。每个星期，老师都给我们一些希腊词根，大家都要记熟，然后通过每周的考试复习。这些词都是用英语拼写的，所以我们连希腊语的字母表都不用学。

现在回想起来，从教育学的观点看，这门课居然没有用上任何分析能力或创造能力，这实在有些让人惊讶。只需要记忆力就行，纯纯粹粹、简简单单。你记住的越多，分数就越高。确实，每次教我们一个词根，那位老师都会引出好些迷人的小故事（他教了一辈子古典学，当然不缺这个），但是考试的时候我们可记不起什么故事了。确实，这些故事可能有助于记住词根，但是，我现在虽然还记得很多词根，但连一个故事也想不起来了。

这些词根时时浮现在我的脑子里，有时候它们真的价值连城。事实上，它们不止一次救过我的命。比如那一回吧，我躺在医院的病床上，因为牙齿感染，连脖子都肿到 40 号了。[①] 一个勤杂工推着一辆盖着布的小车走了进来。在那块布下面，我听见金属器具的碰撞声，于

① 脖子肿到 40 号，这指的是衬衣的领围。

是我也竭尽全力地问了一句:“那是什么?”勤杂工看都没看我一眼,只是机械地说了这样一句话来宽慰我:“哦,没事。这只不过是我们用来作气管切开术的东西[①]。”

他推着车又离开了这个房间,这时我的科学希腊语浮上了脑海,顿时起到了救命作用。这之前我都没听说过“气管切开术”这个词,但是根据希腊词根,我立刻猜到,他们是要切开我的喉咙!不用说,既然我知道了这事,也就做好了准备,当那个大夫过来要做这件坏事的时候,我早就想好了怎么劝阻他别作这个手术。而且我确实成功地劝阻了他!我猜,单单因为我猜到了他要干什么,这一点就让他措手不及。一般的医疗程序,要求病人在最后一刻之前保持无知,到了那一刻才让病人签一个“知情同意书”——这个文件的作用是,如果手术进行得并不顺利,那么凭借这一纸同意书,就给医生多加了好几层法律保护,使他不至于因为不当治疗而被起诉。当然,如果手术顺利但是病人死了那也一样。

有了这希腊语知识,在签这个同意书之前我就能够准备好几个问题来询问医生。也是因为有了希腊语知识,我能听懂他们在回答时的一些用语,这往往是用神秘的医学术语表达出来的。根据他们的回答,我就能够冒险决定,先不让他们切开我的喉咙,无论按照医生的时间安排来说,马上做这个手术是多么方便适宜。幸运的是,经过抗生素治疗,我已经开始消肿了,所以至今我的喉咙还安然无损。

最近,我得了一种神秘的小病,难受了好几个月,怎么治都没用。我的大夫都有点束手无策了,只好把我送进医院,做一些测试和观察——在圣诞节期间留在医院里实在够温馨。经过了5天的刺探、调查、侦询,他们显得比此前更困惑了。这两个大夫,手里拿着图表,在我的床脚开了一个小会。其中一个对另一个说,“可能是医原性的[②]。”

“我也正要这么想呢,”另外的那个严肃地说。

“很可能就是医原性的。”

---

① 气管切开术,英语为 tracheotomy,来自希腊语 trakheia(动脉、气管)和 tome(切口)。

② 医原性的,英语为 iatrogenic,来自希腊语 iatros(医生)和拉丁语 genus(产生)。指因医生的治疗而引起的症状。

我相信，他们以为这个对话会让我觉得情况不妙，因此会服服贴贴地接受更严格的治疗。但是还没等大夫们继续说下去，我的科学希腊语就再一次出马挽救我了。

“好吧，”我说，“如果我的病是你们造成的，也许你们就应该完全停止治疗，让我回家。”他们看上去有点儿吃惊，因为我居然明白他们的意思，这样一来，我也就占了上风，还让他们退了款。我一出院就换了医生，而且还预订了一种医疗服务，以保持自身健康，而不是病了再去治。从那以后，我感觉身体好得多了。

这多好呀，你会说，可是这与计算机编程有什么关系？没什么直接关系，我想，因为我们程序员说的不是科学希腊语。但是我们确实在说一种黑话[①]（比如 JCL，前端-数据库-微处理器，分布式-智能-循环-网络-构架，COBOL-SNOBOL-SPITBOL，APLGOL-DAMMITOL[②]，等等），这些东西的作用，也和医务人员的那些希腊语啦，拉丁语啦，看不懂的草书啦差不多。

那么这又起什么作用呢？为什么我们也要对那些令人同情的客户说些类似于“医原性症状”的鬼话，从而隐藏实情呢？因为我们和医生一样，都要想方设法隐藏错误，但又不能真在错误上面盖上土、种上花儿。在医学中，“医原性”字面意思是“由治疗过程产生的”——简而言之，就是医生们自己引起的病。你不得不承认，“医原性”听上去、感觉起来就要好得多。咱们程序员也应该给自己找个类似的词，可惜希腊人里没有程序员，那么我们该用什么语的词根呢？

对了，如果你还不明白为什么我用了 5 年才上完 4 年的大学课程，我就解释一下：有一年花在了医原性疾病上。我当时还不知道这个——这是在我学希腊语之前一年——直到后来我才推测出来。在那时，我可没意识到是大夫们的误诊让我得了病，我还以为他们救了

---

① 黑话，原文是 Pig Latin，指的是把每个单词开头的辅音移至词尾并附加上另外的一个音节而成的一种行话。作者借用这个称呼描述一些计算机术语，也是因为这里的 Latin（拉丁文）和上面的希腊语略呈对应。

② 这是一串技术黑话。JCL 即 Job Control Language，大型机常用的一种语言。SNOBOL，SPITBOL，APLGOL 也都是编程语言的名称。至于 DAMMITOL，则是一句似是而非的骂人话（相当于 damn it），由于读音与上面的几种语言相近而被收入。

我的命呢！我对此铭心刻骨，甚至还学了一年医学预科，后来明白过来才又回去学计算机。

我很高兴自己还是回去学了计算机。在我们这个行业里，即使一个用户既不懂希腊语、也不懂计算机，他也不会把灾星当成救星。如果我们强调清晰明白的沟通，这类暴行就永远不会发生！

统计数字如何导致误解

只要我拿定了主意，我就爱听支持我的意见。我买了一辆“内燃兔”[①]之后，凡是称颂内燃机、兔子和“内燃兔”的文章我都搜集。自从我饭量下降之后，对表彰苗条的文章我也一篇一篇、读了又读。同样，自从我拿定主意，认为 HIPO[②] 无关大体，我也就开始留意所有支持“程序代码和普通英语表述作为文档的价值”的证据，并且乐意再三品味。

所以你可以想象我的高兴劲儿——当《计算机世界》(Computer World)1979 年 5 月 21 日那一期的第 33 页上有这么一个煽动性的标题：“文档研究证明了程序代码的实用性”。

我狼吞虎咽地读着这篇文章，一直想看看“证据”到底在哪儿，不放过咀嚼每一个段落。该文首先介绍了研究的背景。这是一家研发中心，负责为美国海军陆战队维护“部队专用的，实时的，战术性的计算机程序”。然后文章列出了被考察的软件文档工具：

1. 数据库设计文档(DBDD)，其中包括了所有程序数据结构的描述和图形表示。

2. ANSI 流程图(FLOW)，数据处理中的一种常用工具。

3. 等级关系图(HIER)，显示程序的调用等级，类似于一种组织结构图。

4. 分层结构-输入-处理-输出图(HIPO)，数据处理中的一种常用工具。

5. 计算机程序代码(LIST)，在这家研发中心里包括用于各处的源代码和目标码，还包括所有程序元素中的交叉参照信息和引用信息。

---

① 内燃兔，Diesel Rabbit，是大众汽车公司的一款家用车型。

② HIPO，一种用于程序功能设计、开发和文档编制的图解工具，具体含义下详。

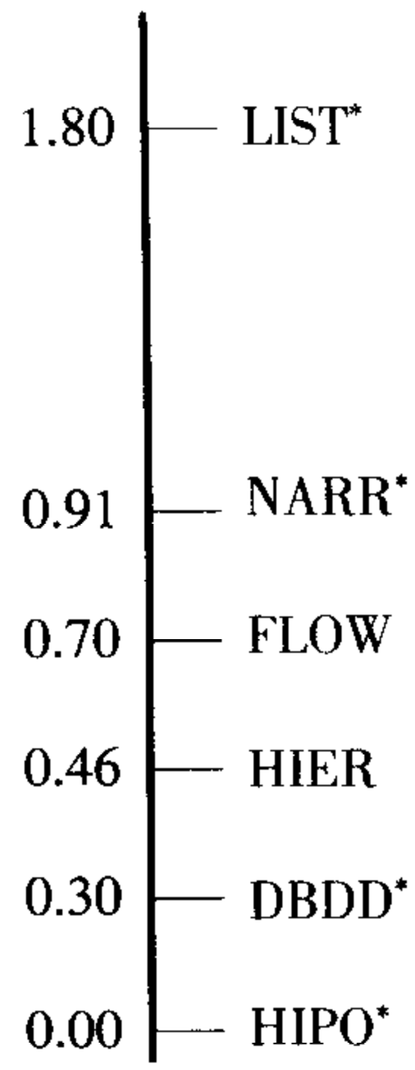


图 3　各种不同类型的文档对程序员的作用

同样在第 33 页，就在标题旁边挺显眼的位置上，有一幅如图 3 的插图。在按照“接下页”指示翻到第 37 页之前，我认真地研读了这幅插图，希望也许从中能够看出“证据”的实质。但我发现，HIPO 的得分是 0.0，旁边还标着一个星号，这个星号的意思是“该数据的有些样本与平均值之间的差异显著性高于 0.90 级别(are significant beyond 0.90 level)”。看到这里，我就开始担心了。

为什么要担心呢？首先，我对“显著”(significant)[①]这个统计学专用词有一种由来已久的别扭。对于统计学家，“差异显著性高于 0.90 级别”大概意味着：

“如果我们做 100 次同样的实验，各项参数都不作变动，我们将有 10 次获得这样的结果。”

---

① 作者写作本篇的主要目的之一，就是探讨 significant 一词的 3 种不同含义。在统计学里，一般译为“显著”，表明特定情况偶然发生的概率极小，所以不能归于偶然因素；在数学计算中，一般译为“有效”或“数据精度”，即我们常说的“取几位有效数字”；在日常用语中，则译为“重要”。那篇《计算机世界》杂志的文章恰好没有明确说明这个词的用法，引起了作者的不满。希望读者细心体察。

而对于读者，“significant”这个词还意味着小数省略上的操作。一般我们会谈到 0.90，0.95，0.98，0.99，也许甚至是 0.999 级别的精度(数字有效性)。另外，很显然，实验最后给出的数据精度肯定也就是统计者能给出的最高精度。也就是说，如果文章中说精度在 0.90 级别，那么你就可以肯定，这不会达到 0.95 的精度。

对“significant”的这些定义其实都没什么问题——虽然它们与平常人对“significant”这个词的理解毫不相干。对于不是统计学家的平常人来说，这“significant”简直是另外一个词，只不过碰巧跟前面那个词的写法、读法一样罢了。根据我的字典，这个词的意思是：“具有一个含义；有意义；充满意义，重要，值得注意。”

你看，如果保证参数不变，进行 100 次相似的实验，得出的结果是有不到 10 次的情况 HIPO 得分为 0.0，那么这个数据**可能**是“重要”或“值得注意”的。可能是如此，但是在第 33 页上没有任何提示确实如此说明。事实上，第 33 页根本也没说明这些数字都是怎么来的。

这也是我开始担心这个 0.0。我自己碰巧相信，在绝大多数情况下使用 HIPO 都不太划算，但我从来没说过它的价值是 0.0。事实上，任何人只要用过 HIPO，或者看过别人用它，都会承认这个技术包含某种价值。作为一个咨询顾问，人家常常问我的却是另一个问题：“使用 HIPO 合算吗？”

回答那个问题的时候，我往往这么说，“我不知道是不是合算，不过如果你希望用 HIPO 做个实验的话，那就先停止使用一个别的什么方法——某个给程序员们造成一定负担的方法。”当我的客户们如法炮制之后，有时候他们确实觉得 HIPO 还是合算的。而且有时候，他们觉得“停止使用另外的那个方法”其实还要更合算。

所以我觉得这个 0.0 不太令人信服——直到我发现图中还有一个词：“标准化”。这时我才明白了。这种图还有个别名，叫“哎哟图”[①]。把得分“标准化”，这个意思就是说，把最低分归化到 0，这样就

---

① “哎哟图”(Gee Whiz Graph)，一种整理数据的常用办法：把坐标上移或右移，使数据更明显。Gee Whiz 是感叹语(哎哟，天呀)，这里是戏称，说明这种图表的表现效果之强烈。

能使差别的效果更明显。

我们不知道，而且那篇文章也没说，原来的分数到底是多少。可能是 HIPO—3.0 分，LIST—4.8 分。或者，也可能是 HIPO—497.1 分，LIST—498.9 分。在统计学上，这可能无关紧要，但是对于想要理解这个图表的重要意义的人来说，这当然是很要紧的。

不消说，看到这里我很有点儿灰心，但是我对自己的意见还是相当投入，所以就继续翻到第 37 页，继续读这篇文章的结论。我这才发现，这些数字并不是采用不同的文档工具，进行了实验之后得出的结果，而是对 18 个程序员做了“问卷调查”的结果。原来是一个民意测验！

作者们说，最后的结果呢，表明“LIST 显然**被看做**一种很优越的工具。”（标宋体是温伯格加上的。）

“看做”？我的字典里，“看做”一词的 3 个定义是（当然还有其他的定义）：

1. 严密、接近地观察某个对象。
2. 用一种特别的方式看待或考虑某个对象。
3. 对某个对象有极大的好感或赞赏。

简言之，虽然该文的标题表明这里的“看做”使用的是第一种定义，但事实上，第二，甚至第三种定义才最符合实际情况。在我看来，这项研究证明的也就是：那 18 名程序员对 LIST 的好感大概要比 HIPO强。

我想公平对待该文的作者们，所以也先提个醒：我这篇文章未必就能说明人家的整项研究。但是，我猜这已经相当接近实质了。这个实质就是：我们选择软件工具的方式，跟选择香烟、除臭剂是一样的。

我可以举一些例子，“老资格程序员十有八九喜欢用子程序。”“我一开始编程用的就是 FORTRAN，所以我宁愿战斗到底，绝不转移阵地。”“当个真正的程序员——别用虚拟机！”

想想吧，我们不正是一直如此吗？是呀，不过现在我们有了统计数字、实验心理学支持自己的偏见了。谁说软件工程没有未来？

来自大学的一课

在 10 年当中，我的大儿子 Chris 在多所大学上过不同的课程。一开始，他是从高中逃课，到大学里去听感兴趣的课。后来，他干脆从高中退学，也不在大学注册，就去大学里上课。

过了几年 Chris 参加了考试，拿到了高中文凭，开始当一个正式的大学生。但是不知怎的，他觉得这样的正规生活不适合自己。每个学期他都要注册 5 门课程，然后再退掉 3 门。有时候他根本就懒得退课了，但是也不去上课，结果就落了个不及格。最后，他也发觉自己这样浪费了好多钱，而且认为自己当正式大学生是没出路的。

Chris 一直对园艺感兴趣。他在大学的维修组找了份差事，这样就能在户外劳动，挣一点钱，甚至还能付得起一两门课程的学费。这个新办法对他好像很合适。他晒得挺漂亮，肌肉也练得挺瓷实，而且几乎每天都能学到点儿什么东西。不过让我说，他学到的东西，主要还是来自在维修组的工作，而不是那两门课程。

前两天，Chris 开始抱怨自己的头头，这人已经在维修组呆了 40 年，马上就要退休了。我问 Chris，难道这头头这么优秀，他们为什么把他留到退休年龄之后？“才不是呢，”Chris 说，“我们想干什么就干什么，他呢，很少盯着我们，甚至在我们干活儿时很少露面。”那么，难道这个人会用什么办法恐吓学校当局，所以他们不敢让他离职？“也不是那样的，”Chris 说，“虽然这么说也许还跟实际情况更接近一点儿。实际情况是，只有他一个人干了这么久，所以知道所有的地下管道都埋在哪儿。很可能他几个月什么都不干，但是他们某一天要挖开校园里的某个新地段，所以就要找他，让他说说管子埋在哪儿。他记得住所有的管子，所以呢，也就给学校省了一大笔钱——不然要是在挖沟的时候挖错了地方，撞上了水管，那花费就大了。我敢保证，就因为他脑子里记得的那些东西，他做出的贡献远比工资多好多倍。”

自然，我不会让这一课白白溜走的，总要从中得出点儿道理。我跟 Chris 说，这就是万物之道。人家付你工资，更多地是因为你所知道

的东西，而不是因为你干的事情。Chris 就是撞破一千次脊梁，也不会从人家那里获得他头头那样的安全感，而这，只是因为后者知道管子埋在哪里。

这种情况在计算机界也一样。不幸的是，很多计算机企业的管理者似乎没看出这个道理：最重要的文档是保存在人脑子里的。有这么一类人，他们的唯一功用，就是唯有他们才记得事情到底是怎么回事，或者事情怎么会变成这样的；对于这类人的这一点功绩，居然就要付给高薪，管理者们总是很气恼。他们更愿意为“代码行数”这一类看得见摸得着的成绩付费。

这些数据处理经理之所以没法好好和这类员工相处，是因为存在一种概念上的误区：他们把“文档”和“文件”搞混了。在这方面，编程和管线维修之间有非常严格的对应。对于管线维修，最基本的“文档”在土层里。如果图纸上说某个地方没有管子，但是你的铲子碰上了管子，你就得相信铲子。如果图纸跟老 Fred 记得的不一样，那也得相信老 Fred。绝大多数情况下，Fred 都会比图纸正确。当然铲子才是终审法官。

在编程时，代码就是土层。当然四处还会有很多其他文件，但是在绝大多数情况下，它们都不如代码本身正确。每个优秀的负责维护的程序员都知道这一点，所以他们实际上很少查看这些文件，虽然说起来，这些文件就是来描述这个待维护的系统的。当出现了一个程序员自己不能直接解决的问题，那么信息的首要来源是老 George——那个几年前编写这个系统的人。10 次里有 9 次，George 能给出你需要的信息。

10 次里的那另外一次，George 也往往能回答这个问题：“另外还有谁会知道这个？”也许他会提到 Sally，而 Sally 回答这个问题也有十有八九的把握。在回答过程中，George 或 Sally 可能会引用一种文件，这文件可能已经被另外的文件覆盖了，但是那后来的文件虽然内容新，却没有你要找的那个信息。

当然，上面所说的这些并不是什么社论或编者按，而只不过是一种描述。程序开发的管理者可以仔细研究一下以上描述，然后对比一下自己机构中发生的实际情况。管理是一件困难的事情。有些管理

者不敢基于现实考虑如何在机构中行事，那么对于他们，管理就还要更困难一些。如果你想要改善自己工作环境中的文档状况，那么，首先了解清楚目前文档是什么状况，难道不是一个好办法吗？

为了改善文档，下一个步骤应该去看看，哪些办法可以对现有的自然流程起到辅助作用——而不是引入那些人为的、武断的方法。比如说，既然代码本身就是所有文档的根基，那么为什么不着手提高代码作为文档的质量呢？所以，在这样一个文档改进计划里，第一步就是要确保，代码在放入程序库中之前，首先要让一个以上的人读过了、理解了。如果代码本身就是一种文献，那么说“测试代码的‘文档功能’的唯一办法就是读代码”就是合情合理的。一旦你的企业开始以一种正规流程审读代码，那么就会自然而然地想出很多办法来提高代码的可读性。

在改进代码的文档功能的同时，很可能你也想找办法来改进“个人”的文档功能。首先，你应该意识到，当这些个人离开机构时，他们作为“文档”也就几乎不可用了，就像文件本身那样。所以，任何有助

于降低离职率的措施，都能够提高这些“活文档”的质量。

在某种程度上，你可以用一个办法来避免离职造成的损失，那就是把这些“活文档”录制下来。录音带就不错，可录像带还要更好。让系统每个部分的设计者到公司的录像室坐几分钟，记录他们的设计思路，这难道不是很好的办法吗？——那些思路很可能在几个月之后就会被忘掉了。在很短时间之内，你就能积累起一个不错的视频文档库，这至少在新人加入项目时会有用处。他们不仅能够学习到整个项目的思路，而且也能了解到设计者的个性，而这些设计者本身就是“活文档”，他们此后少不了要打交道，所以这种了解非常有益。

如果你们没有录像设备，或者说，即使你们有录像设备，也还有其他整理文档的方法，能让“有问题的人”今后找得到“知道答案的人”。比如说，每份代码都有一段“序言”，列出所有参与了这段代码编写的人。另一种常用的办法，是在每一行代码上都注明最后一个编写者的名字缩写字母。这样，如果负责维护的程序员在某一行代码上遇到了困难，就可以很容易地找到对此有最新、最直接的知识的那个人。

即使是其他正式文件，也应该包括一份列表，逐步记录下谁编写、修改、使用过它。在我说的那所大学里，还留着 70 年前的地层图纸。除非跟那么一个老资格的人一边看一边解释，不然这些图纸就根本没用了，因为只有那些老人才记得，比如说，1937 年他们用这张图时某处有个错误。而且毫无疑问，他们那时做出了正确的维修操作，却从来没有改正图纸上的错误。实际上，文档系统越是宏伟，那么使用这些文件的那些凡人修改、更新它们的可能性也就越小。1910 年时设计师画出的漂亮图纸，又有哪个维修工会在上面加上脏乎乎、臭兮兮的随手修改呢？

同样，我们的神圣系统的设计者们留下的文件，也没有哪个程序员敢轻举妄动。不过，任何程序员，只要使用过这些文件，都会清楚地记得弄懂某一段话费了多少时间。所以这样一个程序员也就很乐意帮助后来的新手理解那一段话。如果在每份文件上都留一处地方，让曾经阅读过、理解过的人签上名字，那么后来的读者也就能通过这样一种参照，找到对他们可能最有帮助的人。

关于怎样改进“活文档”系统，我还能提出好多例子，不过最好还

是由你来认识、研究自己企业的这个系统，然后做出自己的改良建议。只有这样，我的这些建议才能完全对你自己的问题起到作用。我在这个问题上的想法，也是来自很多企业，因为它们都做了这些事情——研究它们自己的非正规的信息系统，然后再寻找改进的办法。

所以呢，谁要是说大学从来就不能教什么东西，这可就是一课。大学能教很多东西。如果你回想自己的校园时代——假设你上了大学——而且，如果你对自己真诚的话，你会发现，你自己在大学里也学到了不少。可能你没有意识到，因为你只注意到在那些正规场合的“学习”了——同样，往往你也只注意到整个文档系统中的那些正规部分。但是大学中的大部分学习经验都发生在教室之外，这也正如在软件企业中，大多数文档都存在于正规文档系统之外一样。

老鼠和熨斗：一个寓言

一个星期二，一只老鼠在厨房里探险，偶然爬到了熨衣板上。同在这板上，还有一个电熨斗；这熨斗亮闪闪的，老鼠都能看见自己的影子了。熨斗稍微有点儿倾斜，所以那影子看上去是一只比他自己小一点儿的老鼠。他呢，还是单身汉一个，所以就从这影子大小推断，这只新老鼠是一位女郎。

他羞羞答答地往陌生老鼠那边蹭了一点儿。她，带着同样的羞涩劲儿，也向他靠拢了一点儿。他慎重地微微一笑；她，也同样慎重，用微笑回报。可以想象，遇见这么一位投合的伙伴，他当然立刻地、从头到脚地坠入了爱河。

他们二位共坐片刻，四目相对，如痴如醉，直到房子的女主人打开了厨房门。“快跑，”他对他的甜心下令，于是他们就匆匆逃离了那只熨斗。瞬息间他不禁回望，发现她也确实逃走了，不过路线是相反的方向。“她一定住在那一带，”他这么想。

其后的整整一周，他都在厨房的另外一侧寻觅她的芳踪，可她却杳无影迹。不过，又到了星期二，又架起了熨衣板，放上了熨斗，插上了电源。女主人一离开厨房，他就溜上了熨衣板。没错儿，在熨斗上又是他的真爱。他跑向她，没法掩藏自己的真情；他也看出她的真情流露。他向她伸出手，她也伸手过来。他们两爪相触。他们接吻。他简直激情澎湃，好在还有足够理智，能看出她也一样激情澎湃。他又吻了她，发现似乎她的香吻越来越火热。

他对她吻了又吻。这时她的热乎劲儿已经不容错认。说实话，他已经有点儿热得受不了了，只好退后一点儿冷却一下。他面露微笑，自述生平。虽然她并未开口答话（他喜欢那些能当好听众的女人），从她的微笑中，他也能看出她对这番话的反应。最后，他再也没法抵挡她的魅力，冲上前去，再吻她一回。

可是这一次，没的说，熨斗已经达到了最高温度。“哎哟！”他叫道，连忙跳了开来，但嘴和爪子都被烫伤了。“你干吗要这样？”但还不

等她回答，房子的女主人进来了，所以他们又得逃之夭夭。

整整一周，他都在琢磨他哪一点得罪了这位朋友。可能他太过热衷了。要么，可能他不够热衷。另一方面呢，他花了太长时间谈论自己。没准这就是她觉得冒失的地方。“下一次，”他决定，“我会让她谈谈她自己。我会请求她的原谅。”

下一个星期二，熨衣板又架了出来。他冲上去见自己的意中人，一颗心跳得厉害，生怕她不会来了。她还是来了，而且这一回熨斗没插电，她迎接他的态度就显得挺友好，又带点儿冷淡。俩人度过了一个愉快的钟点，亲嘴儿、握爪儿，对此前的误会只字不提。他倒是正想道及此事，可厨房的门开了，他们又得分别整整一周。

这一周时间，又让这老鼠好生琢磨、思忖。虽说她不计前嫌又接纳了他，他也高兴着呐，但回想这一回的相聚，他总觉得她比以前都要冷淡。他最后决定，下一次碰面自己一定要留心。

下一次会面推迟了一个多小时，因为女主人一直留在厨房里，熨斗也一直在加热，直到熨完了衣服才离开，熨斗呢，则拔下了电源。她刚刚离开，那老鼠就跳上了熨衣板，冲向所爱——她也正伸开双臂等着呐。“哎哟！”他撞上了烤干的熨斗，不由得大叫起来。“你干吗要这样？”

他向她质问，向她恳求，甚至向她表白，承认自己的所有缺陷；但她还是不肯说他究竟做错了什么。不过，最终他还是从她的态度上看出了一点儿变化，即便她一言不发。“也许，”他猜测，“她看到我已经被罚得够惨了，所以就原谅了我。”他走上前去与她相拥，当然啦，她也以第二次会面时的温情，回吻、回拥了他。

就这样，一个又一个星期二，这桩好事准要开演，有时热情，有时冷淡。很快，可怜的老鼠就被熨斗烙了一身伤疤。更糟的事，为了猜透女友的心思，他完全不务正业、不思饮食、日渐消瘦。最后，有一个星期二，他跟她会面回来时，恍恍惚惚、饥肠辘辘，以至于撞上了一个老鼠夹子。这样也终于结束了这场运气不佳的恋爱。

教训：如果你傻到了认为熨斗是为了你才时冷时热，你也就活该挨烫。

换句话说：所谓交流，总要有两个活人。一个活人加一个熨斗则不成。

第6章

我怎样在官僚体系下生存

米德市的三角职位轮换

你听说过所谓的“永恒三角”——那项理论说的是人类性欲与婚姻安排之间的永久纠缠和交织[①]。你听说过百慕大三角——那个关于鬼船呀，鬼水手呀的没完没了的神秘传说。但是迄今为止，有一个东西只有我一人知道，那就是在美利坚合众国的米德市[②]里发生的，程序员神秘循环的大三角。

我也是第三次，或第四次到米德市的时候，才发觉此前见过的很多该市程序员，都有一个特点。米德市有 3 家主要的数据处理公司：合联吸蛋，中央甘蔗测试，还有联合雪茄出租[③]。这 3 家公司总共雇佣的米德市内 70%～80%的程序员，大概一共有 600 人。在我遇到的那些程序员中，有 10 年或以上工作经验的，似乎在以上 3 家公司都工作过。其中有些人呢，已经开始在这个循环中玩第二轮了——我也就开始管这个叫做“米德市三角”。

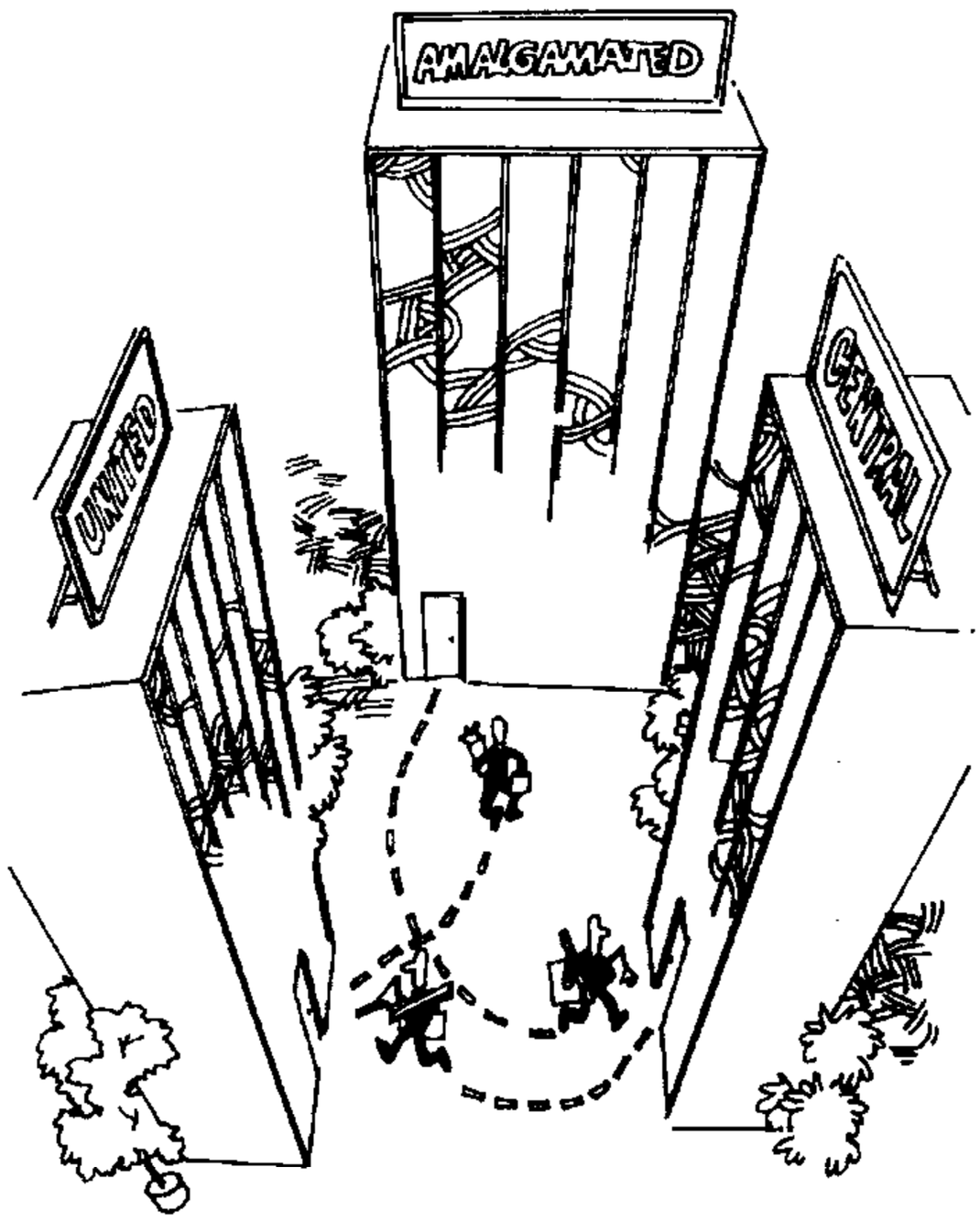
在一个更大些的城市，也许我就不会注意到这种现象了，因为大

---

① 永恒三角：Eternal Triangle，通常就是指“三角恋爱”，但在心理学上，这也是指两性、婚姻关系中的复杂构造，其中亲密度、情欲冲动（利比多）、决断/承诺三种因素相互交织。

② 米德市：Mid-City，直译就是“中等城市”。这是虚构的城市名，含义见后文。

③ 合联吸蛋，中央甘蔗测试，还有联合雪茄出租：从业务的荒诞程度判断，这当然是作者杜撰的 3 家公司。



城市里程序员的工作选择会更加多样。在一个更小些的市镇呢，那就往往只有那一家老爹公司，要么给他干活儿，要么走人。可是在米德市，这种循环显得纯粹、完美。

一般这事会如此进行：Brent Bleary[①] 从大学毕业，加入了“合联”的新人培训计划，当然了，如果他开头进的是另外两家也是一回事。上完了 6 个月的 COBOL 课程，人家就让他写一个小应用程序。他觉得有点挑战，感觉不错。每天他都忙着写代码，提交任务，对着 bug 深思，再把变更的代码合并到程序中。最后，仅仅在预定时间的几周之后，他就提交了自己的杰作，老板深感满意。

作为回报，Brent 得到了一个甚至更有挑战性的程序任务，他自己吹着口哨，就又干了起来。但是在新项目开始几周之后，一个小小的烦恼打断了他——有个人需要对第一个程序略加变更。Brent 一蹴而就，做好了这个变更——虽然文档工作已经让他很头疼了——然后立刻回到了第二个程序。此后的几个月里，每隔一段时间似乎就会发生

① Brent Bleary：作者虚构的人名，字面意思是“迷眼的黑雁”。

一次这种小干扰，但还好，它们顶多也就是挺讨厌地让人从眼前的主要任务上略分心思而已。

第二个程序仅仅推迟了一周就提交了上去，Brent 因此获得了一千次表扬，被加薪一千美元。他甚至奖励自己，去附近的湖边小屋渡了一个短假。让人生气的是，假期也被从中打断了。一个老大爷给他送来一封城里来的加急电报：他的一个程序崩溃了。

假期泡汤了，Brent 当然不高兴，但即便这样，当他回去修正问题时，他发现自己成了大家注意的中心，这种“重要人物”的感觉就抵过了些许的不快。而且，当第三号程序的任务交给他时——这可是他一直想要的任务——所有关于假期的念头——过去的、现在的、未来的假期——都烟消云散了。

第三号程序进行得很顺利——至少是 Brent 不被第一、第二号程序的修正、增强、问题、澄清、服务支持等任务打断的时候。当经理询问第三号程序的进度时，Brent 反过来问了一个问题：要到什么时候，Brent 才能从这些恼人的维护工作中脱身，集中精力做手边的重要任务？不会太久的，经理对他保证。只要新员工完成了 COBOL 培训课程就行了。

课程结束了，却没有新人要来的意思。有紧急任务要处理，再说，也没时间让 Brent 把第一、第二号程序交给生手呀。确实，要让 Brent 把那些程序的微妙之处教给一个刚通过培训的生手，那还不如 Brent 自己处理来得省时呢。Brent 只好同意了这个意见，因为第三号程序的工作和维护的杂务已经太忙了，真的没时间再带新人。

就这样，又过了一年，现在 Brent 已经是骄傲的五胞胎之父了——第一、第二、第三、第四、第五号程序。他手头还在做第六号程序，可是这一回的产期越来越长，因为 Brent 在一个星期里也找不出几个消停的小时，用来编这第六号程序。他的经理好几次都信誓旦旦，要把他从维护工作里解放出来。有一回简直差点儿就实现了，可是就在关键时刻，经理却被提升了，新来的经理在把一切搞定之前不敢担风险，所以只好作罢。

于是，一天晚上，在“计算机与人协会”本市分会的一次聚会上，Brent 听说“中央甘蔗”为了进行一个了不起的新开发项目，正急着招

聘有经验的程序员。他打听了几回，安排了一次面试，完成了程序员智力测试。那个公司给了他一个职位，薪水有所增加，但最重要的好处是：如果他给“中央甘蔗”干活，就不用再负责维护任何“合联”的程序了。他要做的，只是开发一个绝妙的新项目。这就像把老爷车换成了一台新跑车——按揭的月供还比原先少！

但是 Brent 最大的特点就是忠诚，所以他给了他的经理最后一次机会。说实话，这经理对 Brent 的情况满心同情，满口保证几个月之后一定把 Brent 解放出来，还满是好消息：经理刚刚找上司谈判，给 Brent 一次大大的提薪，这样就能确保他在这几个月过渡期也高高兴兴。Brent 为这次提薪谢过了经理，回头就给“中央甘蔗”的人事经理打了电话，接受了那个职位。当他的原经理发觉他已经离职时，简直完全糊涂了：我对他这么同情，还给他这么大的提薪，他怎么会还要走人呢？

得了，长话短说，Brent 在“中央甘蔗”的经历，跟“合联”那边如出一辙。和他的处女程序度过了一段光荣的蜜月之后，这些早期的胜利就成了后来的负担，维护工作越长越胖、絮絮叨叨，好比是给他套上了一副沉重的鞍子。最后，他听说“联合雪茄”正在扩展业务，又经过了一番手续，他就来到了城市另一边，一间粉刷一新的办公室里，永远不用再见“中央甘蔗”那些叫人难受的程序、经理了。

可是，又过了两年，当这个循环转了一圈时，Brent 在米德市简直无处可去了。如果是哥谭市[①]的话，他还可以没完没了地继续换公司，可是在这儿——他该怎么办呢？时间一个月一个月地过去，他开始绝望了，甚至想要换个其他领域的工作，比如套牛[②]什么的。就在这时，碰巧他遇上了从前在“合联”时的经理。

Brent 担心人家对他还有什么残存的敌意，但是毕竟过了 5 年了，大家早就忘记了、宽恕了。另外，“合联”正在对几种大型应用做自动化，所以那位经理也无意去冒犯任何有编程经验的人。没用多少工夫，Brent 就打点起自己的那些模板和作业控制卡片，带着它们回到了

---

① 哥谭市：Gotham，纽约市的一个绰号。

② 套牛：calf-roping，牛仔的一项本事。

“合联”。但不是原来的那张桌子，因为现在他是一个资深程序员了，能够享有一间半私人的办公室。但还有最重要的好处：6 年过去了，没人还记得他和那第一号程序有什么瓜葛。至少是出于礼貌，没人会提这回事了。

我在米德市多待了几天，专门采访了和 Brent 同时代的一些人，比如 Sue、IIarry、Betty Anne、Irma、Wolfgang，还有 Thelma。大家的故事大同小异。Wolfgang 事实上已经在这米德三角里转了两圈，正在商谈第三次循环呢。什么时候是尽头呢？在回来的飞机上，我自己琢磨着。怎样才能击破这个铁三角对芸芸众生的控制呢？就我的回忆而言，在采访中，我发现这些程序员都很快乐，快乐程度可能远超出了一般水准。我在琢磨，现在又是谁被那些维护工作套牢了呢？

（关于这个重要话题，我本想多说两句，不过要说的太多，还是留给下一本书吧。如果你现在等不及了，我建议你去看看 Girish Parikh[①] 的《程序和系统维护技巧》（*Techniques of Program and System Maintenance*），了解一下别人是怎么避免这种要命的米德市三角综合症的。）

① Girish Parikh：技术作家。《程序和系统维护技巧》是他的名著。

大型机构、小型计算机和独立程序员

我总是让自己受一些伟大作品的影响——《圣经》、《释梦》(The Interpretation of Dreams)、甘地的自传,两部艾丽丝历险记[①]和《猎蛇鲨》(The Hunting of the Snark)[②]。《猎蛇鲨》教给很多东西,但其中最重要的是这个道理:任何事情,只要听人讲过 3 次以上,就一定是真的。凭借这样洞察,我足不出户,就能够为整个国家,甚至整个世界把脉。

谈谈我是怎么做到这点的吧。如果在任何一周里,我收到了 3 封信或者 3 个电话,谈论的都是同一种趋势,我就会知道这就是正在发生的实情了。这样,我就能知道什么时候大街上的裙子下摆升高了,什么时候股市指数降低了。我用不着看报纸、看电视,因为这些媒体在捕捉新闻上总要比我慢好几天,甚至好几个礼拜。

最近,我发现了一种新趋势,很想与读者们分享。第一个线索来自这周一的一封信,是我从前的一个学生 Claude 写给我的,他目前在印第安纳波利斯的一家大型食品公司作程序开发。快到这封信的结尾处,他写道:“我太太和我最近买了一台 Radio Shack 计算机[③],开心极了。我太太是一个退下来的程序员,所以,当孩子们不在的时候,她就用所有的时间自己编一些挺棒的程序。我一回到家,她就去厨房做饭,这样我就接管了这台机器。当孩子们睡觉的时候,我们可以一起工作。我们已经开发了 3 个程序包,并且要把它们卖给本地公司,另外也希望进行全国范围的销售。你哪天来我们这儿,一定得过来看看我们的产品线。如果销售得好,我可能也会从公司退下来。就这样,我白天在办公室里应付那些日常工作,只是到了晚上回家才忙自己的事。在家里,没人能够指手画脚——在这儿,他们没法对我颐指气使,

---

① 两部艾丽丝历险记:《艾丽丝漫游奇境》和《艾丽丝镜中奇遇》,都是化名 Lewis Carroll 的英国数学家 Dodgson 所作的著名儿童小说。

② 《猎蛇鲨》也是 Lewis Carroll 所作的长诗。其中“蛇鲨”是一种想象动物,其名称(snark)的名字是作者用“蛇”(snake)和“鲨”(shark)凑出的文字游戏。

③ Radio Shack 公司当时以制造家用计算机著称。

让我忙个不停了。”

就在收到这封信的同一天下午，我又接到了一个老朋友的电话。Joanna 在南加利福尼亚，为一家大型制造商作了 16 年程序开发。我们正安排为她的公司作咨询。“你来这儿的时候，”她说，“一定要陪我买计算机。我下决心要买一台家用计算机，但是还要花工夫来选定最好的一种。”

我可是天生的保守主义者，所以就问了，“你可想清楚了，你在公司里用的计算机都是大家伙，功能那么多，都快把你宠坏了，你买回来这么一个小机器，就那么点儿功能，未必适合你用吧？”

“用不着担心。我的朋友们有这样的机器，我都用过了，所以我早就体会到了哪些工具、哪些性能是家用计算机不具备的。在家里，我能够完全主宰一切，这就能弥补功能上的损失了。其实，家用计算机跟我最早开始工作的时候那种简陋的配置大概是差不多的。”

“是呀，那时候生活多简单呀……”

“我只担心一件事：要是我不用去办公室，就在家里这么干活上了瘾，那还要公司干什么？如果在家就能满足工作需要，那也许我就不用离开家门了。不过无所谓。在家干活儿多有意思！”

不过，两次这样的情况还不能构成一个“趋势”，尤其是，其中还有一人是南加州的，那儿可净是怪人怪事。直到星期四，我又接到了一个电话，这一回是 David 从华盛顿打来的。

在我们的整个教学历程中，David 都是那种不多见的高材生。他参加了我最早的一个培训班，在我们的脑子里和心灵上都留下了无法磨灭的印象。每隔几年，只要我一去首都办事，我都会去拜访他，但是他主动给我打电话，这可是头一回。他已经从程序员高升为系统开发经理了。我一直以为，像他这样身居高位，一定是忙得不亦乐乎，根本没时间打电话闲聊。但是很显然，他打电话来，就是为了闲聊一会儿，交换些新鲜话题、流言蜚语。

办公室里的“新鲜话题”也还是那么不可救药地乏味沉闷，但是当 David 谈起自己的新投机时，他的声音就活跃了起来。“每天晚上，我都尽早溜出办公室，到本地的五六家小企业，为他们做咨询性质的系统分析。我按照他们的业务需求，给他们做好系统需求规范，然后再

把这些规范交给一家事务所，这家事务所已经有两台微机了，马上还会有第三台。”

“我还以为，你忙了一天，下班还要做这一类事情，一定会很疲乏无味呢。”我说。

“恰恰相反。现在，在班上我都没有系统分析工作可做了。即使我有了一个好点子，要实现它也会花很长时间、很多麻烦才能完成，简直把所有乐趣都耗尽了。但是和这些小企业打交道时，我就能处理全局的问题，他们对我言听计从，而且几天后我就能见到结果。这实在让人高兴——就跟从前那时候一样。”

“那你的公司对你这么干怎么想？”

“噢，他们可不知道这事。我就告诉了你一个人，因为我觉得这事也只能告诉你。我猜，如果他们知道我拿着这样的薪水还在干私活儿，一定会要我好看。不过我做这个可不是为了钱——我平时工作挣的钱足够了。但是在咨询工作之中，有某些东西是班上的工作已经没法满足的。我真的觉得自己又有了生气。”

“那你会不会干脆辞职，专做这个？”

“我想不会，虽然那个事务所一直在求我多给他们一些任务。他们已经有了很多潜在客户，但是做不了高水平的系统分析——除非我和另外几个处境相似的人出手帮忙。我觉得冒那么大风险不划算。为什么我要辞职做这个？我在这里有一份终生工作，薪水也挺高，他们也没那么多要求。我想，如果他们真地发现了我的私活儿，而且大惊小怪一番的话，我就只好停下这份咨询差使，但是这也不大可能发生。无论如何，他们不会解雇我。他们从来没解雇过任何人，而且我的工作成绩要比他们期望的还要好上一倍。”

David 放下电话没多久，我还在想他这个情况，这时又来了一个电话。这一次是 Sawyer 打来的，他在硅谷的一家计算机公司工作。Sawyer 是我在大学里的一个学生，现在已经是非常成熟老练的程序员了。离开学校以后，他曾经为 3 个制造商工作过，两年以前才到了目前的这家公司。

Sawyer 来电话的目的，是想打听一些信息：他和两个朋友想开一家公司，这件事该怎么办？“我们正在接合同开发程序，”Sawyer 解释说，

“钱越挣越多，以至于我们觉得，最好还是把这个办成一个真正的企业。”

“我还以为你有一份固定工作呢。”

“是呀，我确实有固定工作，但是他们给了我一台终端，这样我就可以在家里工作，每天我干他们的工作只需要不到一个小时。剩下的时间——只要我们不去骑车锻炼——我用来完善软件包，这些软件包都是针对本地企业购买的微机开发的。”

“你们外卖软件包吗？”

“不卖，我们只做委托开发。制造商卖了硬件之后，就对那些买主不闻不问了，这就给我们留下了空间，我们按照他们需要的方式定制软件，让计算机对他们切实可用，每小时收取 50 美元。”

“这么干你们永远也发不了大财。制造商的做法更可取。批量制造，获得销售额，然后就溜之大吉。这才是利润最大的地方。”

“好了，我们也没想发大财，但是我们的做法也并非不可取。首先，我们都有固定工作，这就能确保基本收入了。而且，我们的客户喜欢我们。我们要多少他们就给多少，因为制造商抛弃了他们，他们也不知道找别人怎么解决问题。另外，我们还在开发一些能卖给多家客户的小东西。我们每次按照从头开发的成本收取费用，但其实只需对以前完成的工作略作修改就可以了。”

“听起来挺好，”我说，“只要你每次都能办到就成。”

“一点儿都不费力。他们太想要程序员了，根本不过问我是怎么干的。嘿，你愿意过来跟我们一块做这个吗？”

你知道，我可是确实动了心。我想，要不是我自己就是一个独立资本家，要不是我也有一份固定工作，拿固定的薪水和福利的话，我真会接下他们这个邀请。想想吧，为什么我要坐在打字机前，像奴隶一样卖力工作？“住在阁楼里的饥寒交迫的作家”在现在这个硬件廉价、人工值钱的时代，早就不时兴了。要是我真地追求自由自在的话，首先我就会去跟一个那种诱人的招工广告联系——你知道，就是那种用绝大部分篇幅夸耀度假啦，员工福利啦，公司附近的娱乐场所啦之类的广告。这样我就能进驻办公室，拿到终端和操作手册，钉上我的卡通画，然后就开始盘算，怎么干一份新的私活儿而不被发现。对了，这就来找找，我把那些广告放在哪儿了？

从“月光”[1]中看世界：管理者的一种视角

不久之前，我发表了上一篇文章，谈了谈专业程序员使用微型计算机干私活儿的趋势。那篇文章讲了 Claude，Joanna，David 和 Sawyer 的故事——当然这些都是化名，不过他们确实是我的真实朋友，每个人都出于不同的原因干起了私活儿。我倒不担心那篇文章给他们哪个人带来麻烦，因为它是在英格兰发表的。但是当我决定把它在更大范围发表时，我先给 Claude，Joanna，David 和 Sawyer 他们每人寄了一份，确保这样不会暴露他们的身份。他们觉得很好玩儿，甚至受宠若惊。但是他们也有一点儿担心。就像 Claude 说的那样，“你是不是要冲我们吹哨子[2]？”

我不太相信是我第一个吹了这哨子的，因为这个现象肯定已经传播很广，不可能这么长期避开人们的注意力。也许管理层早就知道这个了。我决定找几家与 Claude，Joanna，David，Sawyer 所在的企业相似的公司，开展一次调查。我想看看管理者是不是知道了这个。如果他们知道，我想看看他们会对此做些什么。

这些公司有两种反应。一种是恼怒和恐惧，比如，有一个经理是这么说的：“你可不能发表这篇文章。恐怕，有些人会把这个故事当真的！”

我或多或少有点儿盼望这种忧心忡忡的反应。但是还有另一种反应，老实说，这可是吓了我一跳。Claude 公司的另一个部门的经理是这么说的：“我察觉到自己的员工有这一类活动，已经有 3 年了。当然我们不能公开承认这种事，但是我们的政策很简单。只要这些员工的名字不在公开场合出现，只要他们做完该做的工作，我们就不去干涉。事实上，我觉得自从有了这样的活动之后，似乎大家反而比以前高兴得多。我们是个大企业。我们很官僚。我们自己没办法把工作变得更有生气，让员工中最棒的 10%满意。

---

① “月光”(moonlight)也指干“私活儿”。

② “冲我们吹哨子”，意即向人们警示这个现象。

“但是如果这些最棒的人觉得腻烦了，我们也就失去了他们。我们需要这些人的专长，但不是每天、每个小时都不能放过。所以我们扭开脸，假装不知道他们在干什么。如果上级管理层找我交涉的话，我会说，这些人如果在办公室里干私活儿，也算得上是一种培训。而且这是比我们能安排的任何培训都强得多的一种培训。有一个新上任的经理，就是给一家别的办公室干了两年私活儿，然后在我们这里得到了提升。在这段时间里，我发现他在很多方面都有了长足进步。我们在这儿交给他的任务可没法给他这样的经验。这肯定是受益于他在外面的私活儿。”

第三位经理跟我解释，他自己是怎么开始鼓励干私活儿的。他为一家相当稳定、相当保守的金融公司工作。“我的壁球同伴要给他的汽车配件公司买一台计算机，让我出点儿主意。我告诉他，需要有一个人长期担任咨询顾问的角色。而且让他当心那些刚入行的、没经验、不可靠的小孩儿。但是我也没法向他建议怎么做更好。同一天下午，我手下最资深的程序员要来见我。他说另有一家公司要给他更高的工资，可是公司的政策在那儿，我实在没法跟他们比了。他挣的都要比我多多了！当然他值这个价钱，但是我们公司绝不会让一个技术人员挣的比他的经理多的。”

“我提议让他升职当经理，之前他已经拒绝过好多次了。但管理确实不是他该干的事，他能够直接拒绝，我也实在佩服。就在他要走出我办公室的时候，不知为什么，我脑子里突然一闪，想起了打壁球时的谈话。我给那个朋友打了电话，问他如果我手下最棒的一个去给他做咨询，他愿意付多少钱。他愿意付的钱可不少，比起我能给那个程序员的工资和另外那家公司给他的工资之间的差距，还要多一些。我打电话叫那个程序员回到我的办公室，相当谨慎地问，他愿不愿意考虑这样一个安排。他很高兴。其实他真地喜欢这里的工作，也不愿意为了新工作搬出本地。这样，我就保住了手下最好的技术人员，同时也满足了朋友的需要。当然了，如果办公室里的其他人也干私活儿的话，我就没法反对了。但是我会更留心关照，保证他们在办公室都能完成工作，而不是磨洋工。而且，我确实希望公司能破格给技术人员更高的薪水，但是这不一定能办到。所以，在很长一段时间，也许我们

还得用这么个权宜之计。”

另一个经理告诉我，他自己对这件事的反应，是怎么从暴怒转向接受的。“当我第一次发现公司里有这样的现象时，我简直想直接走到那个家伙的桌前，当场把他给解雇了。我当时一定都气得红头涨脸了，不过幸运的是，碰巧已经下班了。我可以用整个晚上冷静下来，根本没怎么睡觉。到了早晨，我才意识到自己把这件事当成对我个人、对我的权威的一种挑战了。我决定先拖几天。然后我和自己的老板谈了谈这个问题——作为一种假设的案例。她只用了一个问题就把我拉回了正轨：‘你觉得，员工干私活儿这件事，说明你为自己的部门创造的工作环境有什么问题？’我大吃一惊，深感挑战。

“结果，我做出了一系列改革，改善本部门的工作环境，纠正自己对待本部门员工的态度。我们扭转了此前多年蕴积的不良趋势。我不是说，没有人再干私活儿了——我知道他们还在干——但是数量上比原来少得多。这也让我始终保持着警惕。干私活儿是一种症状，不是病源本身。我们管理本部门的方式才是这个病源。”

我告诉他，我觉得他还是把这个当成对他个人的挑战了。“其实不是，”他说，“我是经理，我的工作就是对这一类事情负责。确实有一个人为了全职做他的计算机业务离职了。在那以后的几天里，我把他的离职当成了自己的一种失败，但是每次有人离职，我都会这么想。可是，当我回头考虑这件事时，我明白了，一个公司不可能面面俱到，满足所有人的所有要求。我相信，那个人在自己的小公司里一定非常快活，在我们这里根本没法让他高兴到这个程度。不是每个人都一定要在一家跨国公司里当程序员的。”

这最后一句话实在有道理，我也没法说得更好了。

生产力的衡量：也许我们搞反了

所有帮着别人排除过错的程序员都知道，我们总倾向于做出种种未经检验的假设，如果我们真地着手检验它们，那么我们也能从中找到此前从未被发现过的大千世界。

在美国，有一个很普通的文字游戏："假设"这个词(assume)也就是把你("-u-")和我("-me")变成一头傻驴("ass-")。每次别人拿出程序，然后对我们说："那一处你就不用看了。我**知道**那个部分是对的。"我们也就清楚地知道该从哪儿找问题了。

可是，未经检验的假设影响的不只是我们生活中的这一个小节。很多我们赖以为生的假设，都是用从没被置疑过的话语表达出来的，可是其中的很多假设不单是错了，而且恰恰处于真相的反面。

有一回，我听到收音机里一位传教士又在重弹那句烟火气十足的老调："罪孽的报应是死亡"。突然灵光一闪，我想到了这个未经检验的论断的反面："罪孽的报应是生育"。啊哈！我们在编程时能有这样的急转弯吗？

从自己一直以来的经验中，我知道我们会糟糕地把两个正相反的术语搞混。比如说"浮点数"和"定点数"吧，每次要想清楚哪个是哪个，我都要花一番该死的工夫。

这样有一回，在一节课的中间，突然灵光一闪，我就对认真听讲的学生们说："通过这么一个方法你们就能想清楚谁是浮点数，谁是定点数了：如果小数点的位置在这个数字中**固定**住了，永远不动，那就是一个浮点数；而如果小数点位置在数字中，按照运算类型和运算数的位置**浮动**，那么它就是**定**点数。"从那一天起，我对此就再也不会提出问题了。

前两天，我和人讨论估价程序员工作的问题。一个话多的经理自告奋勇，提出了自己的方法："从我的座位上，我能看到通往每个程序员办公隔间的入口。那么，我只要看看到底是哪些人老是到别人的隔间去问问题就成了。"

这话一说完，周遭一片寂静，大家都在琢磨这种估价手段中的潜台词，过后一个人说了，“你知道，我真觉得你这个作法挺有道理。编程是种特别复杂的营生，谁要是以为自己知道每个问题的答案，从来不查书、不问人，这人弄出来的麻烦肯定比他的贡献还要多。”

那个多话的经理突然显得无话可说了——在那整个研讨班的过程中这还是头一回。最后他终于开了尊口，“唔……嗯……你还没明白我的意思。我想说的是，要是他们不**知道**所有问题的答案的话，我干吗要给他们付工资呢？”由此产生了一场白热化的讨论，但即使是到了最后，那个多话经理还没质疑过自己的假设：他总觉得“万事通”型的程序员价值才高。

还是这次研讨班上，稍后的时候，我们讨论了更多的量化标准，用来衡量程序员的生产力。似乎大家钟情的办法还是计算每天产出的代码行数，而争论的主要焦点，在于什么才算得上是一行“代码”。最后，我告诉他们，近期研究显示，对于给定问题，一个解决方案的代码越长，其质量也就越低（这个道理基于 Halstead[①] 在软件科学方面的成果）。

他们迷糊了。“要是不能数代码行数，我们还能数什么？”“即使代码行数和质量成反比，数一数不是也比什么都不数强吗？”

我没有急着回答那个问题，而是考虑了一下：一个陈旧的假设可能会蜕化成一个崭新的错误。“也许，”我说，“你们可以为每天编码量少的程序员发奖。”他们听了这话，简直是活灵活现地打了一个机灵儿，我连忙接着推论下去，把他们带出苦海。

“但是那也会很快变成一个错误的假设。为什么我们不这样测量生产力：对于标准难度的问题，完成 1 000 行质量恒定的代码所用的天数（这其实是生产力的倒数）。”

“但我们没法衡量质量呀！”“难度也没法衡量！”

“噢，”我答应着，又被绊住了。突然我又想起了那个“反面”原则。“要是那样的话，你们就应该先去搞清楚‘质量’、‘难度’等问题，不要拣‘代码行数’这样容易对付，却不相干的地方下手。”

这回轮到他们说“噢”了。

---

① Maurice Howard Halstead，名著《软件科学基础》的作者。

幽默能提高生产力吗

英格兰，曼彻斯特，1923 年那些昏暗惨淡的日子里，我父亲当时 19 岁，整理好很少的几样盘缠，告别故乡，移居英国的前殖民地。虽然他后来几乎完全改掉了英国口音，但却从来没有丢掉英国式的幽默感，这在美国也给他带来了没完没了的麻烦。资本家说他是激进派，激进派说他是资本家，就因为人家从来不把他说的笑话当成笑话听。

我呢，有一个英国父亲，一个地道的美国母亲（她同样从来听不懂他的笑话），于是生来就有一种根本没人理解的幽默感。我总是发现，当我在剧院里哈哈大笑时，别人要么在哭，要么睡着了。虽然没人当真地说过我是激进派，或者资本家，但是我确实经常被误解，尤其是当我要写点儿好玩儿的东西的时候。

显然，上一篇文章我又玩过了火，因为一个叫 D. A. Martin 的读者产生了这样的印象：我提倡用代码行数来衡量程序员的生产力。事实上，我想当时我要说的意思是，计算代码行数根本没有任何道理，除非：(1)该任务所有部分的完成质量都完全一致；(2)所解决的所有问题都有完全相等的难度。而且，既然没有人知道如何衡量质量和难度，我就提议，也许我们应该首先研究这些问题。

如果我正确地理解了 Martin 的来信的话（当然，有可能我也误解了他的幽默），他大致同意我的意见，但是着重讲了进行以上两种衡量的可能方法。他说：

> 两个程序员可能在相同时段内实现同一个程序。一个人的程序有 200 行，另外一个则有 600 行。那么谁的生产力更高呢？Halstead[①] 的意见是，那个 200 行的解法可能“质量”更高——这大概没错，但却是一种误导。因为如果要让“质量”这个词具有确定意义的话，那就必须还要同时考虑到可靠性、可维护性和性能。这话说的非常准确——我只是不同意上文中的那个“可能”罢了。

---

① Halstead，见前一篇文章的注①。

另一个读者 David Coan 看了那篇文章也写了封信，谈了他对“质量”的看法，深获我心。他说，所有衡量判断程序的标准，都应该与下面的问题发生关联：“对于它的用途来说，这个程序是否足够好？”

这很像我们在每次正式技术评审时会问的问题：这个产品是否像人们设想的那样工作？以上问题既可应用于代码，也可用于任何我们想衡量其“质量”的产品。

我相信，这是一个最高层次的问题，比 Martin 提出的那些衡量标准（比如“可维护性”）还要高一个层次。我一直在为“可维护性”辩护，即使在很久以前，人们把我根本不当回事，说我不懂得“程序可和硬件不一样，完全不需要维护！”的时候就是如此（那是 1961 年，IBM 在巴尔匹克召开的编程会议上）。但是对于某些程序来讲，“可维护性”确实不属于“人们设想它该做的工作”。

有些用户爱说“这程序只需要运行一次”——不，我还没有傻到相信他们的地步。但是每次过来这样一个用户，我都会为程序定两种价格：如果程序运行以后由我亲自销毁，那么定价 x 美元；如果要求程序运行两次或者以上（由我或者其他人），那么定价为 3x 美元。如果用户真地不需要“可维护性”，那么为此收费也毫无道理。如果没有对可维护性的需求，却要把程序编写的颇可维护（而且相应提高几倍成本），在我看来，这也不是一个高质量的程序。

换句话说，质量与所处理的问题有关。一辆劳斯莱斯是公认的高质量汽车。但是如果我在内布拉斯加州的林肯市拥有一辆这种车，这儿却没能满足我的需求，因为

我车库里放不下

配件、服务都有困难

它的配件都太独特，很容易遭到盗窃或侵扰。

（如果哪一位读者在考虑送给我一辆劳斯莱斯当生日礼物的话，千万别沮丧。我总能修一个新车库的。）

我相信，Martin 在来信的稍后部分谈到以下内容时，他已经相当接近我的观点了：

> 关键在于，代码行数是对解法的一种衡量，而不是对问题本身的衡量。而评估者的目的，却应该是对于特定问题，在可接受

> 的完成质量的前提下，衡量该问题所需要的人力。为了达到这个目的，他应该去量化问题本身，而不是解法（代码行数）——就像在后一个例子中，那个不可避免地要对“最佳”解决方案的本质做出种种假设的人一样。

当然，上面谈的是衡量工作量，而不是评估员工绩效。如果我们对衡量工作者的生产力感兴趣，我们还得比较他的实际工作量，和基于问题“难度”估计出的工作量。

反过来说，如果我们不去给特定任务设定标准工作量，而是预计实际工期，那么也要参照员工的实际生产力。超出现有的工作能力来估计工期肯定是毫无意义的。也许这也就是为什么我们经常在估计自己编程任务的工期时大错特错的原因吧。

有一个很简单的事实：对于任何真正有意义的问题，我们都不知道该怎样预估它的内在“难度”。我们进行过很多次编程实验，让几个或几组程序员对“同一个”问题进行工作，结果总是发现工作绩效会出现 10 比 1，甚至更高的差别。典型的反差常常是 30 比 1，甚至 50 比 1。从中我自己得出了这样结论：在很多情况下，“问题难度”的概念作为一种衡量尺度是没有意义的。所谓“难度”似乎是问题本身和解决方法之间的一种关系。

当然了，从统计学角度考虑，有些问题在绝大多数时间比另一些容易。但是这是一种有用的衡量尺度吗？想想“从程序中找一个错误”这样的问题吧。我们的实验发现，有的人要 15 分钟找出错误，另一个人要 15 小时，还有的则只要 15 秒。用 15 分钟左右时间发现错误的人比较多，但是我们也发现，对于有些人，无论搜索多长时间，就是找不出错误。我们的各种预估很难以这样一种巨大差异为根据，虽然平均的，甚至标准的工作时间是 15 分钟左右。

有些作者考虑了以上这一类差异，因此建议只雇佣那些“最好的”程序员——也就是能在 15 秒钟找出错误的那种人。如果不可能只雇佣最好的，那就尽量排除那些“15 小时人”和那些永远也找不到错误的人。但是，虽说确实有些人明显不胜任编程职业，但是“能在 15 秒钟内找出错误的程序员”却根本不存在。对于这个错误，一个人可能只要 15 秒钟，对于下一个错误没准就要 15 分钟，对于第三个，甚至就根

本找不出来!

人性的这种巨大可变性,也是我最为重视,一直谨记的事实。所以 Martin 的下列责难肯定与我无关:

> 温伯格这种"把问题翻个个儿"的策略,似乎是一种曲线救国式的思维方法论。这也会导致一种挺诱人的,过度简化问题的做法。程序员的生产力,并不是一个"不相干的问题",也不是一个碰巧很容易回答的问题——恰恰相反,对于软件制造业中的所有人来说,这都是一个核心问题。

别人说我"挺诱人",我当然乐意听,但是我以为他还是误会了我的意思——事实上,正好是我原意的反面。我说的"不相干的问题"是"一个程序员一天能拿出多少行代码?"虽然也带有一点儿小小的麻烦,但是这个问题相对来说还是很容易回答的——但是,我以为,它实在是"不相干"的。我并没有把这个问题和"一个程序员有多高生产力"等同起来。这也就是我整篇文章的核心问题。"把问题翻个个儿"并不是一种"曲线救国"的思维方法,而是说,如果你一直就把问题搞反了,那也就得准备被嘲笑。那篇文章里,我只不过是想让读者看看我们多么容易犯下 180°的错误——我当时的例子就是计算代码行数。

但是,唉,我的英美混血幽默对于一些读者还是没有效果,比如 Martin 吧,其实他的意见跟我百分之百一致,但是还是没明白我的玩笑。但是,我反省之后却觉得这可能并不是我自己的错(什么时候错是出在程序员身上的?)。Martin 是 NCR 公司的一个质检经理,所以肯定每天上班都要跟这个"生产力问题"纠缠不休——这倒不像那些数据处理经理,后者只有到天塌下来的时候,才会把这个问题当作救命稻草。因此,Martin 在这个问题上的考虑也就比一般数据处理经理要更深刻、更切中要害。

不幸的是,无论是 Martin,还是天塌下来的时候的那些数据处理经理,在考虑生产力问题的时候,一点儿开玩笑的情绪都没有。我对此相当理解,也相当佩服。但是我可不推荐这种解决问题的办法。如果我们对这样一个困难的问题太过严肃,那么也就很可能不屑于一些看似"荒谬"的提议,比如"把问题翻个个儿"。

为什么那些看似"荒谬"的提议重要呢?其实这里也没有什么神

秘的。如果一个“有条理”的办法能够解决“生产力衡量”的问题的话，那么 Martin 和我，还有其他的几百个人，在多年之前早就把这个问题解决了。这么多年问题还没解决，这就说明它具有一种根深蒂固的困难。要不，就是说明了在我们所应用的方法中有一种根深蒂固的困难。

所以，当我拿生产力衡量这回事开玩笑时，并不表明我不认真对待这个问题。这只是表明，我认为这个问题太困难了，太重要了，以至于我们没法一直绷着脸对待它。

让我们这样考虑，也许衡量一个程序员能力的最好标准不是代码行数(LOC)，而是幽默感(SOH[①])。我们的业务非常艰巨。不仅工作艰巨，而且衡量也艰巨。也许不用很多 LOC，你也能幸存下来。但是如果 IMS 在你的 TSO VDU 上加了一个 OC3 SVC，而且 IBM 又说没有 ZAP，所以你得要一个 RPQ[②]，那你最好还得有充分的 SOH 供应。不然过不了多久你就得 DOA[③]。

① SOH：sense of humor 的缩写。

② 在这句话中，作者故意模仿计算机行业滥用缩写术语的恶习，以示不可卒读。当然 IMS、TSO、VDU、OC3、SVC、ZAP 和 RPQ 在特定语境下也都有自己的意思，但作者建议中译者把这个句子留给读者，作为一次有趣的读解练习，以及关于如何使用/滥用缩略语的生动一课。

③ DOA：Dead On Arrival(到达前死亡，出师未捷身先死)的缩写。

玛丽亚·特雷莎勋位

Paul Watzlawick[①] 就恐怖分子绑架人质问题作过一些幼稚的建议，我曾经对此颇不感冒。作为补偿和平衡，我乐意对他的《"真"有多真?》(How Real Is Real)一书的第一部分内容做些发挥、引申。说老实话，这第一部分远胜于第二部分——也许他草草结尾，没有工夫审查批判自己的观点。不过，一个错误观点和一个迂腐观点之间可是有天壤之别。只要一个观点能够诱发生成新观点，那就总有望从中产生好东西。《"真"有多真?》这书，就充满了引人入胜的观点，其中一些是对的，一些是错的。

这里，我想谈的观点，体现在一种勋章上，这是玛丽亚·特雷莎女皇统治奥地利时创设的一种勋位——玛丽亚·特雷莎勋位。Watzlawick 这么描述这一奖励：

> 直到第一次世界大战结束，它都是奥地利的最高军事勋位……但这里有一种沁人心脾的荒谬劲儿：它只颁发给那些在战争中力挽狂澜的军官，而且这些军官还必须成心违抗命令，主动把责任揽到自己手中。当然，如果事情办糟了，他们也就不会被授勋，而是会因抗命受到军事法庭审判。

在奥地利这个世界最著名的官僚[②]国家中，居然设立了这么个善解人意的奖项，Watzlawick 在简述事实之后也忍不住要发几句高论：

> 玛丽亚·特雷莎勋位也许是政府当局的一次最典型的"自反矛盾"[③]，体现了奥地利民族特有的一种气质。面对箭矢般扑面而来的厄运，他们的态度往往可以是一句格言：这情况没救了，但不算严重。

---

① Paul Watzlawick，励志类畅销书作家。

② 官僚：bureaucracy，该词既有中性用法，也常包含贬义，与汉语想象中总是"尸位素餐"的官僚还略有不同。下文大多译为"官僚"，但在明显不含贬义的用法中则译为"科层制度"，还望读者体会。

③ 自反矛盾：counterparadox，心理学术语，根据 paradox(悖论、自相矛盾)加上前缀 counter-(反)构成。姑译为"自反矛盾"。

也许 Watzlawick 本来大可以不必多嘴发这番议论的。玛丽亚·特雷莎勋位固然是官僚制度下的一项奇迹式的发明，但却未必够得上“自相矛盾”这个称呼，更不用说什么“自反矛盾”了。每一个成功的组织——国家也好，企业也好，街道风筝俱乐部也好——都有一些规则，这些规则是专门用来反对本组织的其他规则的。玛丽亚·特雷莎勋位的唯一不同寻常之处，在于它竟被成文地确立下来，而且被官方正式认可了。

当杰弗逊草拟美国宪法时，他很自然地写下了一个关于宪法修正的条款。但是，有人让他在宪法中声明，人民有权完全推翻宪法，从头再来。这时杰弗逊拒绝了。他——我认为是正确地——争辩说，无论这种话写不写在宪法里，人民都有此种权利。这是一种废除任何政府，任何官方成文法规的权利。实际上这是“政府”一词的应有之意(tautology)，因为如果不受被统治者(the governed)的认可，那么“政府”(government)也就无从谈起。那样的政府也许是一种幻影，但算不上“政府”。

上述考虑对任何现代科层制度都同样有效。制定规则并不是专门为了让人违抗的——当然也不是专门为了让人不违抗的。之所以制定规则，是为了让一个组织更有效地运营。因此有一条规则就会凌驾在所有其他规则之上：“怎么运营更有效，就怎么干。”从根本上说，如果惩罚你，那也是因为你妨害了有效的运营，而不是因为你违反了规则。

在我看来，官僚制度的问题在于，这枚硬币的“正面”不够闪亮。如果你遵守规则，事情却搞糟了，那么往往不会被惩罚。战争中这甚至会更简单些，因为如果事情真地搞糟了，你没准就送了性命，根本就用不着再向女皇复命了。在 XYZ 肉汁黄豆厂[①]，你当然性命无虞——虽然每当买主用刀叉把你的产品往嘴里送时，都可能因为你的错误而丢掉性命。

玛丽亚·特雷莎勋位的那些获奖者们，在获此殊荣之后生涯如

① 作者虚构的企业名。

何，我倒真有兴趣去钻研一番。我知道，美国的荣誉勋章[①]也常常授给类似情况下的违命者，而很多获奖者此后都泯然众人，甚至会把勋章送到当铺，换上一两文小钱。即使在最好的情况下，荣获勋赏也不过是一种转瞬即逝的荣誉罢了。

当我动笔写作本文时，我本打算这样收尾的：建议每家机构都设立一个“玛丽亚·特雷莎勋位”，鼓励冲破循规蹈矩的束缚——即使在管理得最好的机构中，循规蹈矩的倾向也往往像流行病一样，四处传染蔓延。不过随着写作思路的发展，我才意识到，“勋章”还不是这里的答案。如果你就处于一个大机构的金字塔尖儿上，却想要避免人们错误地因袭执行你的命令，那么恐怕你还得比玛丽亚·特雷莎更卖力才成。

首先，你可以向大家保证，没人会仅仅因为议论你的一条命令有何价值就受到惩罚。即使你不因议论惩罚属下，你也还需要很长的一段时间才能克服人心中固有的畏惧；因为这种对“不当议论”的畏惧，可能是你的属下们在其他机构里多年历练习得的，很难马上克服。不过等这么长时间还是会收到成效的，因为一旦员工们克服了畏惧，开口提意见，你就能放下从前那种自以为“十全十美”的重担了——这“十全十美”可远非什么好事，没有哪个人、哪个国家能承受得了这个。

你可能想到，下一步就可以鼓励大家违抗他们认为不对或愚蠢的命令了。其实，只要言论的风气一开，也就用不着鼓励了。至少其中的一些人，用不着鼓励就会主动不遵成命、另辟蹊径；另一些人则会在旁边关注，看看这些打头阵的会有什么结果。所以，当有些人确实违抗命令，最后又没办成事时，你的第二个问题就来了。当然了，这时候你很容易对他们说：“看看，我不是说过……”但是如果你真地那样做了，从此也就没什么玩头了。所以你必须让那个失败的违命者说说，为什么要违抗命令。也许，这是一条很蠢的命令，即使依命行事也会失败的。也许，大家根本就误解了你的命令——这往往是最可能发生的情况。

一旦你明白了违命的原因，就整个儿放下这件事！每个人都应该

① 荣誉勋章：Medal of Honor，美国军队的最高荣誉。

有偶尔犯点儿错误的权利。如果谁都不犯错，那就说明大家从不尝试、从不思考，这才是科层制度中可能发生的最坏的事情。只有当某个人多次重复违规时，你才应该采取行动——到那时，你的措施也自然就“出师有名”、无懈可击了。

但是，如果多次重复出现失败，那不也会引起一种灾难吗？也许吧，不过那样你就可以用上那句奥地利格言安慰自己了：这情况没救了，但不算严重。

在程序开发行业中，有一个优越的地方：我们的所作所为都有很大的安全缓冲。我们犯了很多错误，但我们也有一整套机制，用来监测、排除错误，不让它们产生太大危害。如果这套机制得力，就为我们犯错误留下很大的余地——而这种余地对于学习来说也是必需的。所以在这种条件下，鼓励违抗愚蠢的命令，不一定非要勋章才行。

胡(狐)狸和山鸡:一个愚(寓)言[①]

一天,天气不错,一只狐狸搬到了山鸡家附近做邻居。山鸡们站得远远的,看着东西从货车上搬下来,这时一只山鸡说了,“这可是豪(好)事呀[②],跟胡(狐)狸做街坊。”

“可不是嘛,”另一只山鸡也同意,“只要胡(狐)狸在这儿,大甲(家)伙儿就别想安生。”

“用不着海(害)怕,”Philip 说了,他是最小的一只山鸡。“只要大家保持团结,就没有哪只胡(狐)狸能伤害我们。我想大家应该开个会,看看我们在胡(狐)狸面前能不能痛(同)仇敌忾。”

虽然大家不太情愿被这个小家伙指挥,但是所有的老大哥也都觉得他说的在理,所以当天晚上大家就开了一个会。会上,大家一致同意,应该坚决彻底地警告狐狸,严禁把“玻璃罩山鸡”[③]放在菜单里,否则就让他当“土罩狐狸”[④]。可是,此后问题又来了:该怎么告诉狐狸呢? Philip 提议,应该让大家都签名,然后作为山鸡大会的决议附件交给狐狸,这样狐狸就能看到究竟是谁支持这个决议了。

“你看,”他对大家解释说,“咱们必须公开表态,拿出人数来,这样胡(狐)狸才能知道我们并不海(害)怕。不然的话,他就会每次抓走咱们一个,把咱们做成一顿山鸡答(大)惭(餐)。”

可是年长的山鸡并不信这一套。比如有一个就说了:“小年轻的华(花)言乔(巧)语当然豪(好)了,但是我要的却是成熟的智慧。”在这

---

① 本篇的标题和内容中,很多字都采用了奇怪的拼写方法。这是作者再现动物语言的一种尝试。中译也用特殊汉字,并加括号补正。

② 山鸡的英文名字是 pheasant,所以每次说到“f”的音,都会因为自己的口音变成“ph”。比如“狐狸”(fox)--“胡狸”(phox),“好”(fine)—“豪”(phine)。

③ 玻璃罩山鸡(pheasant under glass),高档餐馆的一道名菜,山鸡做好(往往填馅)后用大玻璃罩盖好。

④ 土罩狐狸(fox under ground),山鸡们仿照上面的菜名想出的威胁,大概是指让狐狸魂归西山、入土为安。但其实狐狸洞经常挖在地下,这可能是山鸡们的见识尚不及的地方吧。

种场合下，所谓成熟的智慧，就会说狐狸看见每只山鸡的名字，肯定会勃然大怒，这样就会针对签名的人采取过激行动。“最好，”老山鸡说，“还是避免面对面的冲突。只要大家通过这个决议，就能向胡（狐）狸展示我们的纷（份）量了。人（如）果我们的年轻碰（朋）友愿意面对那只胡（狐）狸，我提议就让他去送这份决议。”

这个提议和决议一起通过了，大家也并没有签名，虽然那只年轻山鸡最后还是辩解了一番，希望大家都公开表态。当他拿着决议去找狐狸时，他知道人家交给他这份危险的任务，是为了惩罚他公开挑战年长者的智慧和勇气。“当然，我也海（害）怕，”一边儿敲着狐狸的门，他还一边对自己说，“但是，我也虽（随）时准备微（为）兹（自）由而粘（战）。”

狐狸开了门，看见这么一个客人十分惊讶。“天呀，”他叫出声来，“是我见了鬼了，还是真有一只扇（山）鸡[①]来拜访我？”

“我叫 Philip，”客人说道，对主人的怪腔怪调[②]倒没在意。“山鸡大会派我来照（找）你，交给你这份决议，这样我们就能在相互谅解的基础上建立睦邻友好关系。”

“我叫 Freddy，”狐狸说。“干吗不进来坐坐，让我读完你们的决议，给你一个答复？坐下听听溜（留）声机怎么样？”

狐狸读着那份决议，Philip 在一边儿坐着，尽量掩饰他在发抖。狐狸读完了，就说，“好吧，Filip[③]，这里全是很好的花（话），但为什么没有你们的个人签名呢？”

“这我不能说，Phreddy[④]，”Philip 回答，但是 Freddy 猜到了原因。他也想到了，如果这个山鸡大会不敢让他们每人的名字引起他对个人的注意，那么如果他从中抓一只当点心的话，这个大会也不会反对。除了 Philip——那只年轻山鸡——之外，因为虽说他显然挺害怕，但是

---

① 狐狸的英文名字是 fox，所以每次遇到字母“ph”的组合，都会因为自己的口音变成“f”。比如“山鸡”（pheasant）—“扇鸡”（feasant）。

② 指狐狸的上述口音。

③ 我们看到山鸡和狐狸的名字 Philip、Freddy 也和各自的口音有关，所以狐狸念不好山鸡的名字。

④ 同样，山鸡也念不好狐狸的名字。

如果抓他，也一定会还击的。Freddy 计议已定：他可不愿意让人把眼睛啄出来，尤其是，还有那么多山鸡很容易就抓得到。

于是，山鸡住处这一带的日子也就这样开始了。Philip 管这个叫“吴(五)十只蛋(胆)小的山鸡和一只匪(肥)胡(狐)狸。”

教训：公开表态总要好过不战斗而被吃掉。

换句话说：害怕并非不专业。做胆小鬼才是不专业。

第7章

程序员职业向何处去

一百年后编程会变成什么样

像大多数人一样，我也对未来挺好奇。也像很多人一样，我对过去并不特别好奇。但随着我对未来的好奇逐日增长，我也不得不开始熟知历史：我认为，对未来的好奇心，只有研究过去才最能满足——因为过去也就是“现在”还是“未来”的那段时间。

举个例子说明我的意思。我的一个朋友给我看了一本 1884 年出版的书，这书是在内布拉斯加州的 Prarie Home 地方的一间阁楼上找到的。该书致力于探讨女性就业问题。导论部分综述了一个全新的时代，谈到了女性将摆脱传统的家庭主妇角色——这个变化将“肯定永远不会再逆转”。

全书描述了对于女性最有前途的工作，每一章都介绍一种最重要的工作岗位，比如医药业。奇怪的是，书中没有突出地谈到秘书工作。秘书当时还被称为“文书”，一个商业办公室还没有被视为适合女性工作的环境。

当然了，打字机改变了这一切，不是吗？嗯，这个变化倒不是立竿见影的，因为在打字机刚刚发明的时候，主要当然还是男性来操作。只有男人才能够用机器工作——这个成见根深蒂固，以至于在那本书中没有提到任何“机械性”工作。而且，理所当然地，只有男人才能当文书。

两种主要的工作岗位吸引了我的注意力。第一个是当时被认为

有前途的图片着色员——这个工作是由于当时刚刚萌芽的摄影技术而创生的。摄影术的发展非常普遍，以至于摄影工作室急需年轻女子，着色员都到了供不应求的地步。任何人，但凡有一点儿美术天分，都能学习从事这项职业，人家都会保证终身雇佣她。

另一个有前途的职业是电报员。这种高科技工作确实与机器有关，但是该书作者花了不少笔墨指出，很多现代发明创造，把使用电报机器的任务大大简化了。在近期测试中，女士也能用和男士一样敏捷的手法使用发报机了。

当然，如果该书作者对她那个时代的技术发展了解更多一些的话，她也就不会对与电报相关的职业寄予太高希望了。但是，她对这些发展的理解，很有可能也类似于此前 10 年西联公司[①]的一个官员的意见。当时的科学实验表明，有可能通过电线直接传送人类声音。《科学美国人》(A Scientific American)杂志的一篇文章中(1874 年)，引述了上述官员的意见：

> 有了这样的新发明，就不再有操作仪器(比如发报机)的必要了……业务员用不着再发报，而是通过电线直接传送自己的声音，并且和远端的对方直接谈话。

这样说来，如果我们所说的那位作者是一位未来学家的话，倒应该让当时的年轻女士参加声音训练，以便为今后的电报员工作做准备了。那样一个预测，肯定会被视为大胆而惊世骇俗的，富有很强的未来气息。但也很可能被当作一条深刻有理的建议接受下来。

但是假设，她或者那位西联官员也想到了，今后每人家里几乎都会有一台电话。甚至更糟一些，假设他们预见到了，每个家庭中的电话都不需要专门的操作员就能使用。那样就没人相信他们了，他们会被抛弃，被遗忘在那块专门留给未来学家的大陆上。

而且，抛弃他们是有道理的！为什么呢？因为其实人们并不想知道 100 年后，甚至 50 年后，到底是什么样。他们想知道的是，下个礼拜去哪儿能找到工作，或者这一年里把钱投资到什么地方。1884 年的时候人们并不想了解未来，1984 年的时候人们还是一样不想了解。我

---

① 西联公司，经营汇款业务的著名美国企业。

还可以竖起脖子说，2084 年的时候他们也还是一样。

实际上，预测未来再容易不过了。以下是你可以应用的一些规则：

1. 对于短期来说，一些事情会基本与现在保持一致。

2. 对于长期来说，一些事情会与现在完全不同，以至于根本无法预测。

3. 可是，对于长期来说，我们也都会死去，所以我们当然也就对别人 100 年前的预言不感兴趣，无论对错。

4. 对于短期来说，我们都得谋生，当然这个“短期”，有些人的会短一些，有些人的长一些，但总之都是短期。

以上规律允许你随心所欲地做长期预测。比如，我就预测，2084 年的时候将不再存在“计算机”的概念——除非是像“文书”这么一个古词那样存在。我们可能会有“感情机”取代“计算机”。感情机是一种半生物的，人造生物学系统，这种机器不会“解决”问题（在“解决”这个词当今的意义上），而会让我们不再为问题烦恼，从而消除了那些问题。

当然了，那时也没有什么程序员了。他们会像“照片着色员”一样消失（是，我知道现在还有一小撮人在为照片着色，来制造那种怀旧图书。那么也许未来也会有一小撮 PASCAL 程序员在主题公园里工作——那时一定还会有主题公园，对吧？）

取代程序员的，将是“游泳员”。他们的工作，是潜入装满红花油的大桶中（“感情机”就在这些桶里工作），用身体擦拭感情机的全身，以此与这些机器交流。（哦，对了，我能听见你们中的一些人会问：“分布式感情机怎么样？ 就像电话一样，每家都有的那种？”但是，拜访这种“感情机大桶”是一种宗教朝圣，所以不能在家里安置个人感情机。另外，这东西气味也很难闻。）

嘿，这种预言做起来还真有意思！ 我可以写上好多页——但是你不会觉得好玩的。因为你，和我一样，还都是在短期谋生的。

要是你并不想考虑自己的短期前景，那你还真地会对此感兴趣的。这也就是为什么在飞往澳大利亚的 27 小时航班上，我要读科幻小说。但是如果你考虑自己作为一个程序员的专业前景，那么我要是

建议你去练练在红花油里游泳的话，你就会不高兴的。

好吧，关于最近的未来——比如说，你的职业生涯——我能告诉你什么呢？以下是一些想法：

1. 虽然在短期内，事物会线性发展，但如果有人告诉你，按现在的速度发展，到 2015 年美国会有 5 亿程序员，那你也大可以忘掉这样的结论。

2. 事实上，恰恰因为类似推论的存在，好多企业家都被吓坏了，他们正在积极投资，用以降低对程序员的需求。在电话接线员身上也发生过同样的事情。如果现在还要接线员处理电话的话，以我们现在的通话量，每个美国人都得变成接线员——那还有谁打电话呢？事实是，如果没有发现削减接线员的方法的话，现在的电话系统都不可能存在。

3. 如果现在的趋势延续的话，同样的事情也必须发生在程序员身上。一些削减程序员的方法会起到效果——虽然现在很难肯定哪些会起效果。但是请注意，今天的电话接线员还是比 50 年前要多，当然比 100 年前就更多了。

4. 今天，还有一些公司招聘 1401 Autocoder[①] 专家，而且此后的 20 年里，可能还会有比这更多的公司需要 360/370[②] 汇编语言的专家和 COBOL 专家。但是可能你自己却不会再想做这种工作。

我还能做其他一些预测，但是我觉得主要概念已经很清楚了。在你的整个编程生涯中，都会有充裕的编程工作，但是将来你特别想做的那些工作，却很可能与你现在的编程工作大为不同。处于为这样的将来做准备的目的，你必须准备应变。

那么，怎样准备应变呢？我会在其他地方讲这个，不过，现在不妨停留在抽象层次上，再加一条预言：

5. 如果我们考察历史，我们从没发现有谁能超越自己的生活。相反，我们有时会发现人们活得“不够”。

---

① 1401 是 IBM 在 20 世纪 60 年代早期生产的一种计算机，Autocoder 是在其上运行的一种编译器，能把指令语句编译成机器码。

② 360/370 也是 IBM 生产的著名计算机。

哪些人“活得不够”呢？大多数都是那些并不实际生活，却去梦想未来生活的人。这样说，为了应对未来的变化，就应该充分地生活在现在！

所以，不要问“一百年后编程会变成什么样？”相反，应该看看你现在正在做什么。问一问自己，“今天我学到了什么？”如果回答是“很少或根本没学到，”那么你作为一个专业程序员的未来，就不一定会比现在更光明。难道这就是你把梦想投射到无法触及的未来的理由？

程序生涯能有多长时间

有一种“典型化”的思维方式，认为调查统计中的“平均值”是一个现实的个体，这一结果往往惨不忍睹。比如说，假设我们调查一下周末日场舞会的参加者。碰巧儿，票卖给了当地的共济会，这些会员们就带着小女儿参加舞会，让她们见见世面。对于这样一个群体来说，女儿们大多是5岁，3英尺高，50磅重；老爹数量相仿，大多是35岁的汉子，6英尺高，200磅重。如果这样调查下来，我们就能轻而易举地得到以下结果：舞会的“标准”爱好者，大概是一个20岁的胖侏儒，高4英尺6英寸，重125磅——而且还彻头彻尾地雌雄莫辨！

——引自温伯格著《论稳定系统设计》

关于“程序员的江郎才尽”，不少人也写了不少“典型化”的废话。这个假想的现象，要表达的是生产力和经验之间的一种关系，如图4所示：

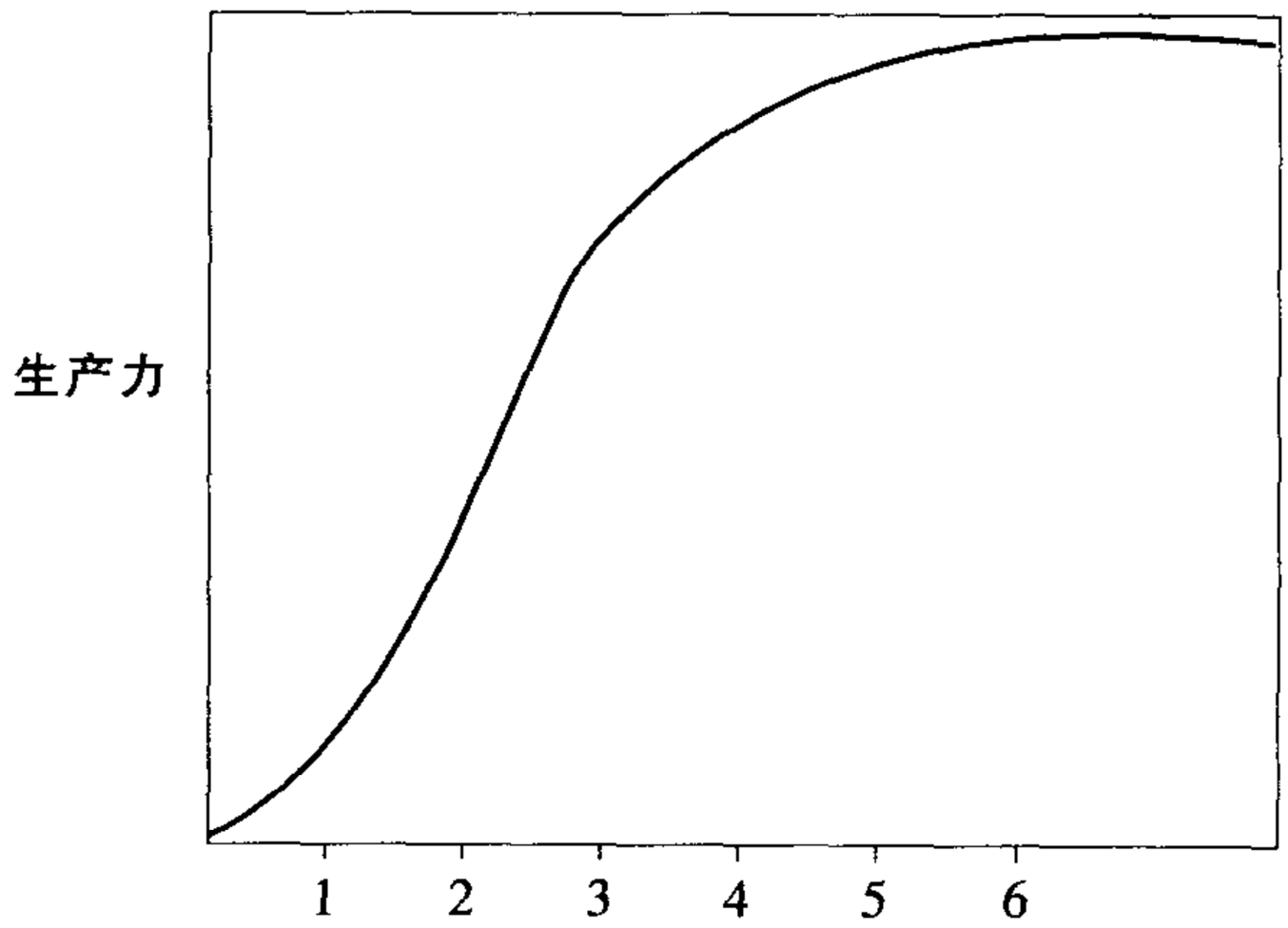


图4　生产力与工作经验之间的假想关系

根据这幅图，工作3年以后，随着年份增加，程序员的生产力并不

会明显地提高。所以，相信这个模型的管理者，也就当然不愿意为多年工作经验多付工资了。通常，他们会从人才市场上找一些有一到两年经验的程序员。

这个模型的一个问题，是它并没有考虑问题的难度。在大多数公司，平均来讲，会让经验丰富的程序员写难度更大的程序。所以，如果生产力的衡量标准不考虑问题的难度，那么经验丰富的程序员的生产力自然就被低估了。在很多情况下，一个更符合现实情况的模型应该如图 5 所示。

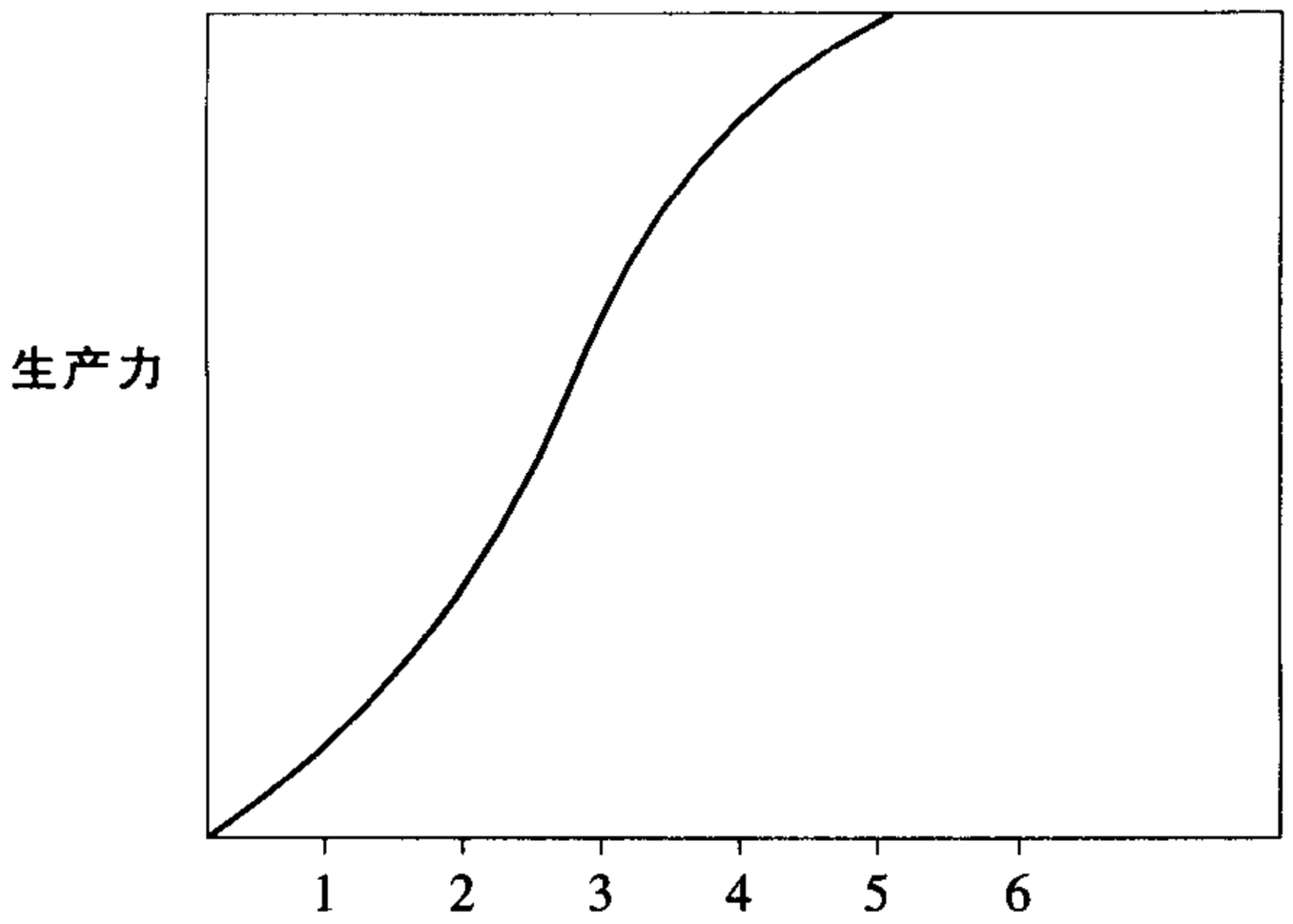


图 5　考虑了问题难度之后的生产力/经验曲线

这个曲线就稍微乐观一些了，它也已经被程序开发心理学的大量研究成果所证实。当我们要求经验不同的程序员完成同样的任务时——其中包括编码，寻找 bugs，编写测试数据——平均的绩效/经验曲线往往就和图 5 吻合。即使这样，一个管理者研究了图 5，还是会采用雇佣无经验程序员的策略，这样才能更好地利用曲线中最陡峭的一段上升部分。相反，为了换取此后按资排辈的待遇，程序员们却能够忍受工作最初一二年的低报酬。

图 4 和图 5 表现的都是普通程序员的模型。在那些真正的大型公司里，要雇佣成百上千的程序员，人事政策会迫使管理者在雇佣和付酬时遵循“平均”的标准。可是，在小型企业中，管理者就能够做出

更多不同的决定。如果他们有眼光的话，他们就能区别每个程序员个体和那个假设的“平均值”，并由此获益。

给出图 5 的那一组研究，还支持另一个观点：管理者不光要看经验，还要看经验本身的“质量”。一个程序员真地有 10 年经验，还是说，只不过是 1 年经验，重复了 10 次？如果我们把每次研究中的程序员分成两组——高产出的和低产出的——我们大致能得到如图 6 的经验曲线。

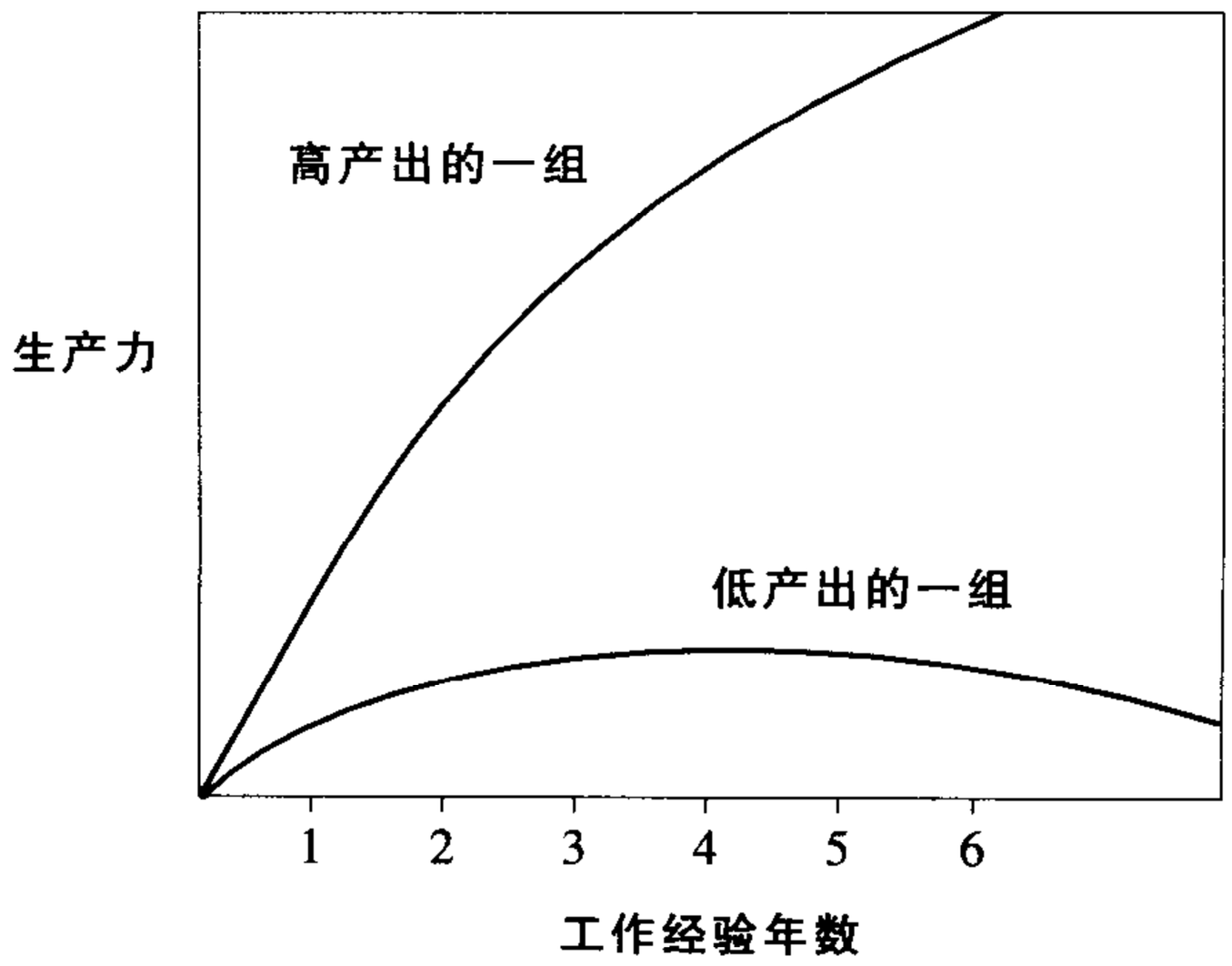


图 6　按产出能力区分的生产力/经验曲线

图 4 和图 5 的曲线，可能都是把两种曲线混合成单一曲线的结果。

这两种曲线体现了真实情况吗？还是说，这只是对数据的人为操纵？我自己对于类似研究的经验和我的工作经验都告诉我，这里面包含了一条重要的真理。为了理解这个真理，假设我们能够跟踪 100 个程序员的职业生涯，再假设他们都是 6 年前开始工作的。每一年，都有一些人会放弃编程工作，因为不喜欢这个工作，要么就是因为不能胜任。一开始离开的人很多，但此后会随着工资的增长逐渐下降，因为涨工资能克服工作中不满意的地方。

一年或者两年之后，管理者就会提出对有些表现更好的程序员加以提升，任命编程之外的工作。一些人会接受。另一些人则更愿意作

他们喜欢而且擅长的编程，所以会拒绝提升。不过，总的来说，提升的机会往往落在比较优秀的程序员的身上——无论他们是不是分析师、经理、数据库管理员的最好人选。

下面的表，显示了在这 6 年中，这个假想的 100 名程序员人群，会发生怎样的成分变化：

年	留在编程工作中的人数	放弃编程工作的人数(累计)	因提升而离开编程工作的人数(累计)
0	100	0	0
1	85	15	0
2	75	20	5
3	65	20	15
4	45	20	35
5	35	20	45
6	30	20	50

以上数字都并未力求准确，只不过是体现趋势而已。

让我们再把这个表拆成两张表，每一张表包括 50 个程序员。首先是“低产出”的 50 人：

年	留在编程工作中的人数	放弃编程工作的人数	因提升而离开编程工作的人数
0	50	0	0
1	37	13	0
2	32	17	1
3	30	17	3
4	26	17	7
5	24	17	9
6	23	17	10

然后是“高产出”的 50 人：

年	留在编程工作中的人数	放弃编程工作的人数	因提升而离开编程工作的人数
0	50	0	0
1	48	2	0
2	43	3	4
3	35	3	12
4	19	3	28
5	11	3	36
6	7	3	40

换句话说，依照这个模型，6 年之后留下的 30 个程序员中，有 77%的人都来自能力不强的一组。最差的几个人一开始就放弃了。另一方面，能力强的那一组里，大部分人都被提升了，这样才能满足非编程工作岗位上对资深人才不断增长的需要。

如果我们假设，程序员的能力不是这里唯一的或最关键的因素，仍然可以采取同样的建模方法。无论考虑哪一种因素，都必须避免把人员构成视为静态。否则也就会让我们陷入“典型化”的误区。

在人员构成中，你能看到多种选择因素作用的结果，所以当你认为自己在衡量一个因素时，很可能其实衡量的是两种或更多因素的混合作用。比如说，我们衡量的可能不是工作经验年数，而是错过提升的年数和拒绝被提升的年数的一个混合物。

如果管理层犯下了这种“典型化”的错误，那一定是有害的，但是往往还算不上是致命错误。可是，如果某人犯了这样的错误，那就可能导致个人悲剧。所以，个人应该不再相信“典型化”的神话，把个人的选择建立在实际“个人”的基础上。

你会在 3 年之后江郎才尽吗？别看那个“标准”程序员！你该去找找，有没有 10 年老兵还很厉害的。然后再调查一下，为什么人家这么厉害。再看看其他人，学习他们每人身上的长处。无论如何，别把自己跟“标准”程序员看齐——否则你也就成了那个雌雄莫辨的胖侏儒！

我该做多长时间程序员

我写了一篇文章，讨论程序员职业生涯的长度、生产力，以及在考虑这些问题时“典型化思考”的危害。发表之后，好几个读者都来信，为我揭穿了“3 年江郎才尽”的谬误而高兴，但是他们也表达了对文章的不满。有一个人是这么说的：“你其实根本没有回答题目中的问题：程序员生涯能有多长时间？”

我确实没回答，因为我也不知道。或者说，我也不知道怎么回答这样一个大而化之的、社会学的问题。我更乐意从个人角度，通过故事来考虑问题，所以我情愿把问题改一下：我该做多长时间程序员？

这样一个问题让人从特殊情况开始思考。如果我的脑筋沉浸在特殊情况中，往往就会有些什么东西浮现出来，然后我就能整理它，打磨它，最后把它包装成一条普遍原则。

不知为什么，这个关于编程生涯的问题，却让我想到了我自己的阅读强迫症。我肯定是很小的时候就养成了这个习惯，因为我能记起自己 4 岁的时候，就一边吃早饭，一边认盒子上的字了。在旅行的时候，我最害怕的是被晾在机场，书店又关了门。可是，最近我克服了对机场的这种恐惧，因为我明白了读电话目录的乐趣。

我相信，我对电话目录的兴趣是从冰岛开始的。因为冰岛面积很小，地理上非常孤立，所以在这个国家中过去和未来产生了一种奇妙的结合，而这一点对美洲和欧洲的旅游者来说也特别有乐趣。比如说，整个首都从前都是通过地热采暖的——当然现在也是如此——早在地热流行之前很久冰岛人就已经充分利用它了。

冰岛还有一个现代的电话体系，其中包括电话目录，但是目录中有一些来自过去的、惊人的东西。如果你要从目录找一个人，你很快就会发现人名是按照字母顺序排的——不过是按照名字排的！

14 世纪之前，很少有欧洲人有姓。随着人口的增长，随着城市人口越来越多，流动人口越来越多，为了达到管理目的，名字就不够用了。但是，在冰岛，这种现代化还处于缓慢的发展中，我们自己的古老

欧洲的人名系统还被保留下来。

当大家开始使用姓，也就用到了好几种不同的体系。在北部欧洲，比如冰岛，两个人如果名字一样，就通过父亲的名字来区别。所以当指定姓的时候，这个体系就被永久固定下来，因为政府没法容忍一代一代人从 Olaf Jonson 到 Sigmund Olafson 到 Ivar Sigmundson 变个没完（别问我政府为什么能容忍妇女婚后改名）。

在冰岛，这套体系却没有固定下来。在电话目录上也有父名，但不是按照字母顺序排的。干吗要按字母顺序排父名呢？既然父名只是用来区分相同的名字的。另外，不同的职业也可以用来区分电话目录上的人，这和其他地方在普遍采用姓之前的情况一样。

在美国的电话目录上，有几百个姓其实都是职业名称，即便我们很少想到这一点也是如此。对于有些姓，比如 Smith 和 Fuller[①]，那些职业都已经消失了。我发现，在林肯城的电话簿里，有家公司叫 Waggoner[②] 汽车修理店。我怀疑，这 Waggoner 一家都没想过这个姓是怎么来的。也许这一家就是车夫世家，一代一代传下来，最后从马车工作转行进入了汽车工作。

有些人争论说，编程不会存在太久，也就不可能产生“世袭程序员”。我在电话目录里也没有看见“Programmer”的字样，虽然我确实发现了不少“Coder”[③]。难道编码行业也已经走到了顶峰，现在要像铁匠一样消亡了？

当人们不再使用马匹，当然也就不需要那么多铁匠了，这样“Smith”这个名字成了一种虚弱的回声。程序也就是今天的马匹，虽然程序员可能还不必担心自己会遭受铁匠那样的命运。

实际上，传统行业不只有一种命运。从中世纪以来，人口有很大增长，所以今天我们还在大量地消费面粉。那么，从前每个村子都有的姓 Miller[④] 的人现在去哪儿了？今天我们每人使用的灯光比起古人

---

① Smith 的本意是铁匠，Fuller 的本意则是漂洗工。

② Waggoner 的本意就是车夫。

③ Coder 这个名字的本意是“编码员”。

④ Miller 的本意是磨坊主。

要强几千倍。难道姓 Chandler[①] 的人，也比从前多了几千倍吗？

这个蜡烛商的例子正好可以拿来与编程作对比。正因为我们现在需要的灯光更多，所以我们也就不能再依靠蜡烛了。灯光的成本越低，需求也就越大，需求增长了，灯光的成本也就进一步降低。

同样，编程工作的市场也取决于编程的成本。如果成本太高，那么市场就会萎缩。或者，更可能发生的是，市场会去寻找更廉价的替代品。

对于传统行业来说，这样的变化往往需要一个人一生的时间完成，有时候还会跨越几代人。但是对于编程行业，我们在一生中就看到了许多次这样的变化，这种情况也注定给个人带来巨大的压力，就像从前传统行业的变化给那些家族带来的压力一样。

很多程序员反抗这种压力的办法是自己逃避压力，要么去找那种还在使用“传统”编程方法的职位，要么干脆放弃编程工作，寻求管理、行政之类的非技术工种。

一代一代的人都留在同一个行业中，这显然表明人类有某种很强的内在力量驱使他们这样做。虽然好些先知都在预言技术乌托邦即将来临，但是改变工作方式可能比他们料想的要难得多。至少，这会让人失去个人认同感，就像从前转行的家族失去家族认同感一样。另一方面，也许这种损失是“进步”所必需的小小代价吧。

无论如何，“3 年就改行”的说法是一种谬论。更糟的是，这样的预言会自动实现自己。也就是说，如果你相信这样的预言，你就会不再努力向前，也就会开始琢磨，怎么样才能不再编一行行的代码（这样最后就真地实现了这个预言）。

也许应该从现在就开始做个调查，研究一下人身上的哪些潜在动力决定他们做多长程序员，研究一下“正确的”职业观念是怎样影响程序员的工作的。为了取得第一步的近似结论，我们应该考虑一下两者之间的微妙平衡：(1)当人们觉得无论是否学习都很安全时，他们就会停滞不前；(2)当人们觉得他们不会在这个行业干太长时间，所以也不值得去费力学习，这时候他们也会停滞不前。

---

① Chandler 的本意是蜡烛商。

前一种倾向在大学教授那儿最明显，尤其是那些终身教授。后一种倾向在程序员中间特别普遍。但是还有很多教授，很多程序员在不断地努力进步。他们与其他人有什么不同？

如果我们能回答这个问题，我们也就能回答“该做多长时间程序员”的问题。首先，我们把问题改成如下形式：我怎样知道何时就应该不做程序员了？

这个问题的答案很清楚。我把它称作“职业到期测试”：如果你不学习了，那么你也就到期了。

说了一圈儿，我又回到了“读电话目录”的话题上了。如果你能把机场上“损失”的时间充分利用上，那么你也肯定就能充分利用上在大学里、编程工作中“损失”的时间。如果你没有这样的动力的话，那还是溜之大吉算了。

我如何为未来做准备

“请问，你能告诉我，我从这儿该走哪条路吗？”

“这可要看你想去哪儿了。”那只猫说。

“我去哪儿都无所谓——”艾丽丝说。

“那样你走哪条路也都无所谓。”猫说。

“——只要能到某个地方就成。”艾丽丝又解释了一句。

“你肯定能走到某个地方，”猫说，“这只要走得足够远就成。”

——Lewis Carroll：《艾丽丝漫游奇境》①

几十年来，计算机行业的变动如此剧烈，任何人只要起步得早，也就几乎一定到达了“某个地方”。今天，虽然机会仍然很大，但是发展趋势也越来越清晰了，所以要对未来的程序员略加指引，总还是能比柴郡猫给艾丽丝的建议更具体一些。

艾丽丝相当幸运，因为她的家教很好，这样就为在奇境中历险做了不错的准备。每当她遇到什么奇怪的情况，她都能找出一条普遍原则来对付过关——至少是让自己免于陷入太多的麻烦。

比如说，当她一掉进那个兔子洞，就碰到了一个奇怪的小瓶子，上面的标签写着：“喝了我”。② 当时艾丽丝非常镇定，做出了正确的举止：

> 说“喝了我”倒确实不错，可是聪明的小艾丽丝也不会那么忙着照办。“不，我得先看看，”她说，“观察一下它是否标着‘有毒’”；因为她读过几个不错的小故事，是讲一些小孩儿被烫着、被野兽吃了，或者遇上了其他的倒霉事，都是因为他们不记着朋友们教过的简单规则：比如说，要是你拿着一个红红热热的拨火棍

---

① 我们知道，《艾丽丝漫游奇境》是化名 Lewis Carroll 的英国数学家 Dodgson 所作的著名儿童小说。内容荒诞不经，但似又饱含哲理。这段 Alice 和柴郡猫的对话，是各行业的作者尤其爱引用的。

② 兔子洞是 Alice 进入奇境的入口。

太长时间，那就会被烫着；再比如，如果你用刀子把指头深深割一下，往往就会出血；所以呢，她也不会忘记以下这一点：如果你从一个标着“有毒”的瓶子里喝了太多，那简直一定会对你不利的——或迟或早。

对于那些在“编程奇境”中寻求历险的朋友，艾丽丝的规则当然是很有帮助的：别拿着红热的拨火棍太长时间；别用刀子把自己割得太深；而且，也别从一个标着“有毒”的瓶子里喝得太多。即便未来还云山雾罩，我们也能通过观察过去得出下面的推论：

1. 从来没有哪个程序员因为身体好而遭了太大的罪。

提醒你这一点，似乎都没太大必要，但是很多专业程序员对待身体，简直像拿着烧红了的拨火棍的小孩儿一样。所以，如果你想在编程领域“走到某个地方”的话，那么想办法保持一个好的身体，成功的可能性就会大大提高。或者，如果你现在还不够健康的话，就先把身体弄好吧！

从过去中，我们还能采集到哪些智慧的话呢？当我想到自己认识的很多专业人士时，我觉得下面的话很恰当。

2. 没有哪个程序员，是因为自知太多而大大影响了前途的。

这并不是说，如果你毫无自知，那也就不能在编程领域有所长进。要是那样的话，我也就不会在这里写这篇文章了。回想我的早年经历，在自知方面的盲区太大——简直足够遮住一条银河了。我倒宁愿说，我后来的经历是托了自己成熟之后的眼光的福，但是似乎我学的东西越多，我的自知也就越少。到现在，我确实能够明白，如果我当时能站在局外考虑问题的话，早先有很多机会本来能够做得更好的。想一想，如果当时不是被自己的盲区遮盖了双眼，我会省下多少次开夜车的浪费！

我倒是很会观察别人的盲区。每次我终于发现了自己是多盲目时，我也会发现，其实别人也有“见我所未见”的本事。这样一来，我就会请求别人洞察力的帮助，作为回报，我也用自己的洞察力去帮助他们。

获得、索取这一类的眼光往往并不像我想的那么容易，但是这又教给我另一个道理：

3. 没有哪个程序员，是因为善于和人相处而大大阻碍了发展的。

注意，我说的是“和别人相处”，而不是“顺应别人”[①]。这个区别特别细微，但是在计算机行业却特别重要。计算机永远不会“顺应”。有些人老是为了计算机的这种“直脾气”发火，可是这样的人做程序员久不了。程序员们最后会明白，计算机不会把谎言当作论据——因为它的论据就是简简单单的事实。计算机也不会操纵你非按照它的办法做，因为它没有自己的“办法”。

从计算机上人们懂得了正直——因为计算机正是正直待人的。而为了应付不正直的系统，人们也懂得了如何不正直地行事——我们也能从中得出另外 3 条成功的原则：

4. 没有哪个程序员，因为正直对待计算机，或者正直对待正派人，而受到过伤害。

5. 很多程序员都曾因为正直对待不正派的人而受到过伤害，但是他们都很快地恢复了。

6. 也有很多程序员，为了跟不正派的人打交道，自己也就变得不正派，而这种不正派往往是永久性的，成了无可挽回的残疾。

我猜，这些也许还可以归结为“处理不同意见的技巧”或者“在有限时间内承受困难的意愿”。如果你的工作让人实在没法忍受，或者你还有其他选择，可参见这一条：

7. 如果程序员银行账户里不缺存款，或者还有其他的公司邀请，那么在纷争中，或困难中，他也不大会做出个人让步，或个人牺牲。

但是，问题不仅仅是问题，也提供了学习的机会——了解这一点对你也会有帮助。比如说，离职当然是躲开一大堆麻烦的好办法，但是：

8. 对于程序员来说，“一大堆麻烦”也提供了增进自我了解，提升与人相处能力的机会。

如果你不相信，只要读一读艾丽丝的故事，或者其他历险故事里成百上千个历经劫难的主人公的收获，就不难知道了。

我自己在“编程奇境”的历险教给我一个道理：要想为未知的将来

---

① “和别人相处”(get along with)和“顺应别人”(go along with)形义相近。

做准备，与其说秘诀在于学习 COBOL 的关键字，不如说在于掌握在不同情况下处理问题的新手段。我自己作为一个咨询顾问的工作也曾经有过这样一种转变，这主要还是受到了 Carl Rogers[①] 著作的影响。在他的《论个人力量》(On Personal Power)一书中，Rogers 说：

> 个体，而不是问题，是我们关注的重点。我们的目的不是要解决一个特定的问题，而是要帮助个体成长，使他既能应付当前的问题，也能用某种更完善的方式，应付今后的问题。

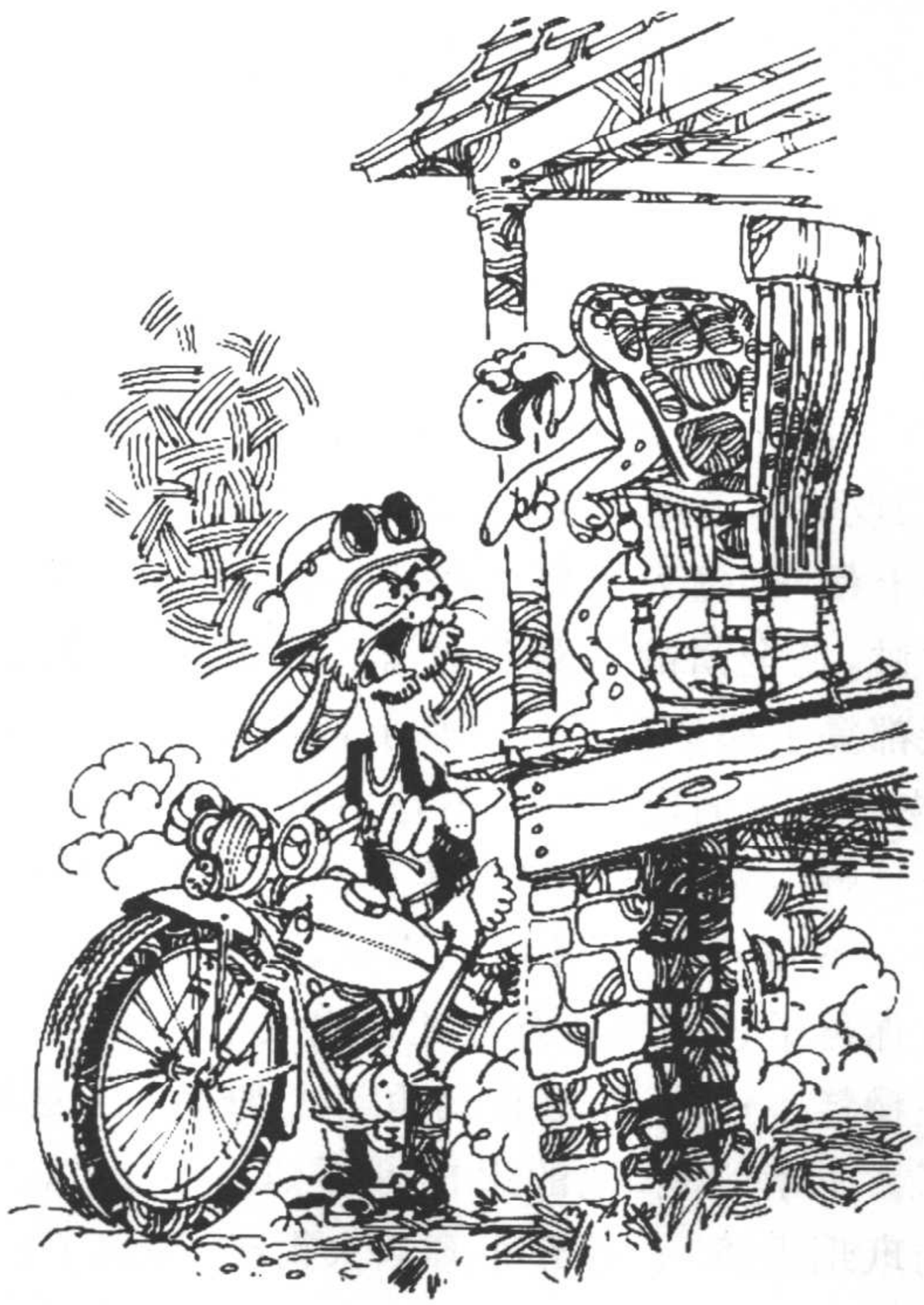
这番话也就引出了我最后的原则：

9. 如果一个程序员一直都在成长、进步，他也就不会害怕未来。

① Carl Rogers(1902—1987)，美国著名心理学家。

乌龟和毛毛：一个寓言

一个秋天的傍晚，一只老龟在自己的前院里欣赏着落日，回忆着自己从前那些了不起的胜利，还用立体声音响听着莫扎特的一首优美的《圆号协奏曲》。突然之间，他的美梦被粗暴地打断了，一只年轻的野兔驾着摩托车轰鸣而至，还开着收音机，听着摇滚乐队的嚎叫。兔子在这条街上上下下开了两回之后，乌龟简直受够了。



“嘿，你这个小混蛋，”他叫道，“干嘛不关掉那些‘耶耶耶’的，听点儿幽静的、有教养的东西？”

兔子一个急刹车，把摩托停在了乌龟的院子前面。“是这么回事，老爹，”他这么回话，尽量显得毕恭毕敬，“我可没有教你该听什么呀，

所以你干嘛要来指教我？另外，说道‘混蛋’，我才不是呢，你大概还沾边儿[①]。我可是野兔，不是乌龟。”

“要我说，你是个野‘毛毛’[②]，”乌龟怒气冲冲地说，他也上岁数了，耳朵有点不好使。“瞅瞅你自己：你可是浑身是毛。你怎么就不能像我一样，把毛修理修理得体面点儿？”

“好吧，乌龟先生，我来解释一下。我是野兔，故有此毛。你是乌龟，所以也秃如一龟。[③] 我可没非得让你去弄些毛再回来，那你干吗不也这么善待我呢？”

乌龟对自己的光秃外壳相当敏感，所以他又换了个话题。“还有一件事——那台摩托！你怎么就不能像所有体面人那样开辆小车呢？”

“我买不起小车。”

“这是废话，”乌龟告诫说，“首先，要是你找个工作，老老实实干点活儿，你就能买得起小车的。”

“我确实有工作呀，只不过挣的不多就是了。”

“……第二点，”乌龟继续讲，想要自圆其说，“你开那个摩托，无非是想在高速路上飙车，撞死无辜的路人。成，我让你见识见识。我像你这么大的时候，是全州的公路赛冠军。打败了你的亲祖父的，就是我！所有报纸都登了那个新闻。你骑上你的小摩托吧，我也把我的车开出来，给你来个真正的比赛——不是小孩儿玩的游戏！”

“嘿，老爹，”野兔看见乌龟当真要开动了，不由得喊了起来，“我可不想跟你比赛。爷爷跟我说过你是当时最热门的车手，我也深信不疑。不过现在你再干这种事就有点儿上岁数了……”

这么说最糟糕不过了，正好让乌龟怒上加怒——这时他已经准备好开车上路了。不幸的是，他的眼神可不如当年了，尤其是还老不愿意当众戴他的玳瑁眼镜[④]。他开车奔向大街时，冲到了路边，撞上了一

---

① 上文乌龟的骂人话“小混蛋”的原文是 whippersnapper，其中后半部分的 snapper 正好有“甲鱼”的意思。所以兔子有此一说。这里只好意译。

② 英语“野兔”(hare)和“毛发”(hair)同音，所以有此一说。

③ “秃得像只乌龟”是一句俗语。

④ 玳瑁眼镜(tortoise-shell glasses)就是海龟壳制品。

辆警车(警车正好是来教训野兔,让他别用摩托打扰佳邻的),虽然没人受伤,乌龟的车可全都撞坏了——不过这倒也不要紧了,因为他的驾照也由于不戴眼镜驾车被吊销了。

教训:年轻人再莽撞,再吵闹,再没教养,头发再难看,再长,说到底,年纪越大还是越走下坡路。

换句话说:专业人士的最终安置应该是体面地退休。

尾　声

所谓“尾声”，就是“文学作品结尾的一个补充段落，或是一段结论，常常要介绍作品中人物此后的下落。”这本书也许算一本文学作品，也许不算，但是其中却只有一个人物——那就是“专业程序员”。另外，或许你已经读完了讲“未来”的那些章节了。那么，为什么还要一个“尾声”呢？

对这个问题，我也说不太准，不过大概有几种可能：

1. 如果结尾没有足够的戏剧性的话，编辑就要对我大吵大嚷了。

2. 可能有些读者不愿意通读枯燥的全书，所以是从结尾开始看这本书的。如果没有这么一个“尾声”，这些读者可能不乐意。

3. 我不知怎么觉得还有话要说。

上面的 3 种可能，都不能算是写一篇尾声的好理由，不过加在一起倒能算得上“一条”过得去的理由了。不过，也许还有某种更强的力量在驱使着我——这大概与下面的故事有关。

有一个午夜，一位了不起的天文学教授突然听见自家前门上传来了一声重击，不由得吓了一跳。他从卧室窗户露出头去看，发现这噪声的来源是一个年轻人——他的“宇宙演化史”课程的一个学生。

“这么晚了，你这是要干什么？”教授抱怨地说。

“扰您清梦啦，实在抱歉，但是我真是灰心极了、不安极了。根本睡不着觉，而且我都害怕自己把自己弄伤了。”

“真糟糕，但这跟我有什么关系？”

“都是因为您讲的课呀，”学生解释说，“您讲到了宇宙的终结，真是太可怕了，让我好生不安。”

“我看不出来你有什么好不安的，”教授反驳说，“我不是跟你们说过了嘛，宇宙在 1 000 亿年以内是不会终结的。”

“哦！我给听成了 1 亿年了！”

那么，为什么在给这本写给专业程序员，谈论专业程序员的书到了尾声时，我会突然想起来这么个故事呢？我自己，不止一次，也有这

样的经历：深更半夜，突然被某个“紧急事件”给唤醒了——结果这“紧急”程度大概就跟那个学生差不多。我相信，很多专业程序员也都有同样的经历，当然另外还有许多别的经历，所有这些加在一起，就很容易让他们高估自己工作的重要性。

在过去的几十年里，编程工作是一种不错的职业。在今后的几十年里，前景似乎也不错，甚至还会更好。任何人，稍微有点儿编程天赋，都算是生活在一个幸运的时代了，不过这至多也就能叫“幸运”而已。50 年前，我们这种工作连一个铜板都不值，而在距今 50 年后还可能更糟糕。即使是听说了 50 年以后程序员都要被吊死，我也一点儿都不奇怪。

但是你作为一个人的自身价值，并不完全依赖于你作为一件劳动力商品的价值。说到底，一个婴儿有什么用？但是，正因为人家对你付以高价、争相求聘，很可能你就会忘记了自己除了编程技术之外的价值。矮梯胖梯[①]如果从墙上摔下来，大概不比煮熟的鸡蛋要好多少，而一旦我们的行业发生了变故，你的下场没准也与此类似。

在 20 世纪 70 年代早期，我就见到自己的一些朋友碰上了这种事情。这不是什么好玩的事情，而且以后还会发生。在这种事情里摔得最惨的，就是从前爬得最高的那些人——倒不一定是职位最好的人，而是各方面自我膨胀最厉害的那些人[②]。

但是，如果你根本没碰上这么一摔，那可能还要更糟糕。你也许会一直自我膨胀下去，直到变得盛气凌人，牛气十足，让人没法忍受，连所有的家人、朋友、邻居都讨厌这个人。每次我的书受到了读者的欢迎，我就会变成这么个人（我自己也知道这一点），而且我也知道，朋友们总是要花天大的力气，才能把我重新拉回地面。

所以，写了这么一本关于专业程序员的书，我倒有点儿担心了，恐怕对于一些程序员已经自我膨胀的个性，这本书又将起到推波助澜的作用。如果这些程序员跟我一样的话，那么这种自我膨胀的感觉（总

---

① 矮梯胖梯：在《艾丽丝镜中奇遇》中，这是一个身材像鸡蛋的人物，我们已经在《个人化学和健康身体》、《说你所想，要么想你所说》这两章里见过他的事迹了。

② 我们记得矮梯胖梯就挺喜欢自吹自擂，而且总是坐在一堵墙上。

觉得我自己的问题是大问题，是重要得多、复杂得多的问题，与普通人遇到的那些不可同日而语），可能会导致不少不愉快。

下一次你再有这样的感觉，就想想那个学天文的学生吧，还有整个宇宙那剩下的 1 亿年，或者 1 千亿年。要么，就想一想这颗星球上生活着的 50 亿其他人。要么，想想宇宙中那 1 千亿个银河系，其中每一个可能都有 1 万亿颗星星，每颗星星上都可能生活着 50 亿人。

其中不少“人”可能都是程序员，也许有那么好几十亿人吧，此时此刻，正在跟你处理同样的一个 bug 呢。每当我在什么问题上一筹莫展的时候，想到宇宙中的那些难兄难弟就会让我宽慰不少。这也就是作为本书的尾声，我想告诉你的事情：如果你对自己在宇宙中的位置有了更合理的认识，你睡觉都能香得多。

译后记

在本书作者温伯格先生精辟、生动的序言和前言之后，在我的朋友熊节兄高屋建瓴的译者序之后，对本书内容再作更多介绍肯定会显得多余——甚至会像蝙蝠对燕子的模仿，拙劣而未得其精髓。

我曾经有一个谬论：对于一本书，作者类似于狠心的父母（很少有作者愿意从头重读自己的作品，他们在生产之后就把孩子推向社会了）；读者类似于宽容的朋友（所谓“好读书，不求甚解”，即使是最认真的读者也不可能了解全书的所有细节；而且“君子之交淡如水”，我们跟绝大部分书本，都是有过一面之交后就天各一方了，虽然常常思念，但也不必朝夕相对）；而译者，则类似于钟情的恋人。翻译，就像恋爱，是一次铭心刻骨的体验；译者与原作一起思想，一起表达，他知道对方的每一处细微的闪光、含混或瑕疵——在翻译完成的那一刻，他甚至成了比作者本人更了解其作品的人。

也正是因此，我就更加折服于作者在本书中体现的睿智、博学和洞察力。在阅读和翻译（有一些诡辩者说，这两者其实是一回事）中，我遇到了许多可口、绝妙的轶事和典故，常常不禁会心击节；另外，作者（作为 20 世纪 50 年代就加入这个行业的老资格程序员）也在书中留下了编程行业各个发展阶段的快照和历史片断（少数古旧的术语，一些难以检索的缩写，等等）。为了复原其中包含的信息，我也动用了一切可能的手段，直至向作者本人求援。带着一个“译者/恋人”谦卑的骄傲，我能够做出以下坦白：读者眼下看到的中译本是译者经过体验，经过历险所赢得的充分理解——当然其中也可能不乏误解，就像

恋人之间通常发生的那样。

原作的风格晓畅、生动，对口语和典故的运用都恰到好处；译本的目标首先是力求忠实于原意。其次，也试图用笨拙的汉语跟随这种风格。在一些技穷的地方，只能放弃逐字翻译，转向解释；有时为了提高中文句子的可理解性，会对原句增添若干字，略去生涩；原作没有注释，为了介绍技术、文化背景，译本中加入了一些译注。

有 3 点特别的考虑希望读者体察：

1. 作者在全书各处都使用了“数据处理”(data processing)这个说法，其实这就是“计算机处理”(computing)的一个古旧表达法。为了保持原作的时代感，在中译本中仍然采用了“数据处理”这个翻译。

2. 作者多次引用了英国作家 Lewis Carroll 著名的艾丽丝系列童话。对于那些想要更好地理解本书的读者，我也乐意推荐这些艾丽丝故事：如果你喜欢本书，你肯定也会喜欢那些故事的——当然，反过来也一样。

3. 在《幽默能提高生产力吗》一节的结尾，作者留下了一个密电码式的段落。我曾想过在注解中给出一种破译，以便读者参考。但与温伯格先生商议后，还是决定把猜谜的乐趣保留给读者。每个朋友，每个恋人，都会对我们保留某个难以理解的秘密，对吧？

在此我愿意感谢清华大学出版社的热忱帮助与合作，感谢熊节兄画龙点睛的译者序，感谢作者 G. M. 温伯格先生的耐心解答和指导。

译稿告竣的一刻，也许注定是个不可思议的时刻。“译者/恋人”在此时既完成了体验、历险，也要面对与“所恋”的离别。请读者评判译本吧。同时，让我用披头士乐队的一句歌词告别这次难忘的翻译：“And if you saw my love/You'd love her too”。

刘天北

理解专业程序员

Understanding the Professional Programmer

如果你是一个程序员，或是程序员的管理者，或者处于任何和程序员紧密相关的位置，这就是你该读的那本书……温伯格的文风明朗、有趣，使阅读本书成为一种享受……作者深入考察了程序员的心理，增进读者对程序员职业所具有的种种独特、微妙特征的理解……任何时间你都能打开本书，从其中任何一篇短文开始阅读——并且发现一个闪光的思想。

——《系统开发》杂志

ISBN 7-302-12994-0

9 787302 129943 >

定价：25.00 元