

软件随想录

[美] Joel Spolsky ○著 杨帆○译

卷1

第15届JOLT大奖图书

被读者誉为堪比《人月神话》的经典软件项目管理文集

Stack Overflow创始人兼CEO切身感悟

Joel on Software



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

软件 随想录

卷1

“这本书自首次出版来已经过去十年了，但是内容常读常新。尽管少数内容略显过时，不过其中的基本原理依然如故。对于软件开发行业中的任何人来说，这都是一本好书。”

—— Keith Hodo
Avanade高级软件工程师

“在这个产品生命期只有六个月的领域，这本书经受住了时间的考验。Joel在这本关于软件开发的书中分享了自己作为程序员、项目经理和软件公司CEO的经验。许多关于编程和项目管理的策略在今天仍然适用。即使不完全同意Joel的观点，大部分程序员、项目经理和软件公司高管也都有必要读读这本书。”

—— Amazon.com读者评论

“这本书风趣幽默，通俗易懂，全是干货，切中肯綮。书中的很多内容虽是常识，但是许多软件团队在现实中所做的却恰恰相反。具有软件开发或管理经验的人可能会在其中发现自己犯过的错误，如从头开始重写软件或是构建过于复杂的架构。我愿意把这本书送给我的工程师和经理，这挺难得的，因为像《人月神话》这么经典的书毕竟不多。”

—— Jonathan Abrams
创业家，曾创办Nuzzel、Socializr、Friendster和HotLinks并担任CEO
Founders Den联合创始人

“Joel把软件工程教科书里的内容变得容易理解，用幽默的文风把枯燥的知识变得引人入胜。如果你觉得读功能规范书让人头疼，或者想在轻松的周末充实自己，那就翻开这本书吧。”

—— Amazon.com读者评论

Apress®

图灵社区: iTuring.cn

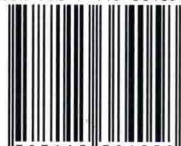
热线: (010)51095186转600

分类建议 计算机/软件管理与开发

人民邮电出版社网址: www.ptpress.com.cn



ISBN 978-7-115-38405-8



9 787115 384058 >

ISBN 978-7-115-38405-8

定价: 49.00元

软件 随想录

中国计算机报·中国软件网



中国软件网·JOLT大奖赛

中国软件网·JOLT大奖赛

中国软件网·JOLT大奖赛

Joelon Software



中国计算机报·中国软件网



中国软件网·JOLT大奖赛

TURING

软件随想录

[美] Joel Spolsky ○著 杨帆○译

Joel on Software

人民邮电出版社

北 京

图书在版编目 (C I P) 数据

软件随想录. 第1卷 / (美) 斯伯尔斯基
(Spolsky, J.) 著; 杨帆译. — 北京: 人民邮电出版社,
2015. 4

ISBN 978-7-115-38405-8

I. ①软… II. ①斯… ②杨… III. ①软件工程—文
集 IV. ①TP311.5-53

中国版本图书馆CIP数据核字 (2015) 第017821号

内 容 提 要

这是一本介绍软件管理的小品文集。全书分为 45 章, 内容十分丰富全面, 小到项目负责人制订进度表, 大到软件执行总裁提出富有竞争性的战略, 都在本书的介绍之列。

本书面向的读者包括软件开发人员、设计人员、管理人员以及从事软件相关工作的其他人员。

◆ 著 [美] Joel Spolsky

译 杨 帆

责任编辑 朱 巍

执行编辑 杨 琳

责任印制 杨林杰

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号

邮编 100164 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

固安县铭成印刷有限公司印刷

◆ 开本: 720 × 960 1/16

印张: 17.75

字数: 312千字

2015年4月第1版

印数: 1—4 000册

2015年4月河北第1次印刷

著作权合同登记号 图字: 01-2013-4003号

定价: 49.00元

读者服务热线: (010)51095186转600 印装质量热线: (010)81055316

反盗版热线: (010)81055315

广告经营许可证: 京崇工商广字第 0021 号

站在巨人的肩上
Standing on Shoulders of Giants



iTuring.cn

版 权 声 明

Original English language edition, entitled *Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity* by Joel Spolsky, published by Apress, 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705 USA.

Copyright © 2004 by Joel Spolsky. Simplified Chinese-language edition copyright © 2015 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由Apress L.P.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。版权所有，侵权必究。

致 谢

首先要感谢我的出版商、编辑兼好友加里·康奈尔，是他让本书得以问世。感谢 Apress 出版社工作人员一直以来的大力支持。Apress 是一家以作者为本的出版社，这在计算机图书出版行业非常罕见，非常荣幸能和这家出版社合作。

本书内容脱胎于我的个人网站，这个网站的存在离不开 UserLand 软件公司的戴夫·维纳的鼓励、宣传以及免费提供的网站托管服务。感谢菲利普·格林斯潘，是他教会我要把学到的东西放到网上和大家分享。还有诺亚·特拉特，感谢他在闲谈中提议我应该把自己的满腹牢骚记录下来，这让我产生了开设博客的念头。

我还要感谢微软 Excel 团队的同事，在我职业生涯的起步阶段，是他们教会我如何开发一个商业软件。

多年来有数百万人访问了我的网站，其中大概有 1 万人拨冗给我写过电子邮件。来自读者的鼓励支持着我坚持将博客写下去。虽然由于篇幅所限，无法列举各位的名字，但我还是非常诚挚地感谢大家给我的留言。

前言

你从没想过要当一名开发经理。和我认识的大多数软件开发者一样，如果能一直坐在那里安静地写代码，你可能会比现在快乐得多。但你是最棒的开发者，自从团队之前的老大奈杰尔发生了那场不幸的事故（与蹦极绳和笔记本电脑有关），作为团队技术明星的你，被顺理成章地提拔为了开发经理。

现在，你有了自己独立的办公室（而不是和那些赖着不走的暑期实习生挤在一个小隔间里），不得不填写每年两次的绩效考核表（而不是每天盯着 CRT 显示器来自毁双目），还得经常想办法满足明星级程序员们的各种古怪要求，和过分亲热的销售人员称兄道弟，和异想天开的“用户界面设计师”们（天啊！他们是以图形设计师的身份被招进公司的）讨论设计稿——他们想让确定和取消按钮闪闪发亮并且带有倒影效果，天啊，“倒影效果”对应的 RGB 颜色值是多少？还要硬着头皮回答高级副总裁提出的各种愚蠢问题，他们对于软件的那点了解全部来自于旅途中翻阅达美航空公司提供的旅行杂志。“我们为什么不用 Java 来代替 Oracle 呢？听说它一致性更好。”

恭喜你荣升管理岗位！你猜怎么着？软件项目管理和编程其实一点关系都没有。如果这辈子除了写代码什么都没干过，你可能会突然发现，人类行为比最普通的 Intel CPU 还难以预测。

其实，之前的团队老大奈杰尔也不是非常称职。“我并不想变成一个整天泡在无聊会议里的经理，”他在发表这种宣言的时候总显得有些虚张声势，“我认为自己还是可以把 85% 的时间花在编程上，剩下的时间用来管理团队。”

奈杰尔其实想说的是：“我完全不懂如何管理项目，但愿只要像被委以重任之前一样老老实实在地写代码，所有事情就都能顺利地完成。”这当然不可能。这种心态可能有助于解释，为什么奈杰尔会在命中注定的那一天，带着一台 IBM ThinkPad 去蹦极。

故事的细节不必多说，总之后来奈杰尔奇迹般地恢复了健康，与一起玩蹦极的朋友合开了一家小公司，出任 CTO。这家公司叫“今夕何夕”(WhatTime-IsIt.com)。他只有六个月的时间，要从头编写一个新系统，而且这次再也不能靠事故来脱身了。

管理的艺术

软件项目管理是一门很少有人了解的艺术。没有人获得过软件项目管理的相关学位，这方面的图书也很少。只有极少数人带领软件开发项目取得了真正的成功，其中绝大多数都在发财后早早退休去钓鱼，来不及把积攒的经验教训传给下一代项目经理。剩下的人燃烧尽了自己的青春，改行去做其他压力稍小一点的工作，比如帮市中心的小混混辅导英语。

这导致了许多软件开发项目都因为种种原因而中途夭折，有的失败得轰轰烈烈，有的则无声无息，原因是团队里没有人知道如何带领一个软件项目走向成功。有太多的开发团队无法交付像样的产品，或者拖了太久才交付，或者交付了一个谁都不想用的软件。但是最让我感到愤怒的，是那些在团队中闷闷不乐，痛恨每一分钟工作时间的人。人生苦短，怎么可以浪费时间去痛恨自己的工作。

几年前我在网站上发表了“乔尔测试”，这份清单包含优秀软件团队必须要做到的 12 件事，包括维护一个 bug 跟踪数据库，让面试候选人现场写代码，等等（不要着急，这本书会拿出大量的篇幅来介绍这些内容）。让我感到惊讶的是，很多人发来电子邮件，说自己所在的团队只能在满分为 12 分的测试中得到 2~3 分。

2~3 分！

这太不可思议了！就好像一群木匠聚在一起做家具，结果谁都没听说过螺丝钉。他们只会用踢踏舞鞋把钉子敲进木头里，因为没人告诉他们世界上还有一种东西叫锤子。

要做好软件项目管理工作，需要一整套和写代码完全不同的工作技能和技术储备。项目管理和软件开发是两个完全不同、几乎没有交集的领域，二者之间的差别几乎相当于脑外科手术和烹饪椒盐卷饼之间的差别。一名脑外科手术医生完全想不到，自己会被传送到一个制作椒盐卷饼的工厂里，这很可能

是时空裂隙造成的结果。即使是接受过哈佛医学院教育的外科医生，也完全对如何烹饪椒盐卷饼没有一点概念。但是人们通常认为，一位顶尖的开发者不经过任何岗前培训，就能自动胜任项目管理工作。

和刚刚提到的脑外科医生一样，你和奈杰尔都被安排到了一个新岗位——管理。这个岗位要求你和人类而不是编译器打交道。如果你觉得现在的 Java 编译器 bug 丛生、行为不可预料的话，说明你还没有作为开发经理和团队中的明星程序员打过交道。管理由人类组成的团队，难度要大大超出编写 C++ 模板函数。

要实现成功的软件项目管理，也并非无迹可寻。这门艺术发展到今天，已经脱离了钉子和踢踏舞鞋的阶段。我们现在有锤子、螺丝刀和双斜面组合条锯。本书的目标就是介绍我知道的所有管理方法和技术，涉及各个层面，包括团队管理者需要预估时间和开发计划，软件公司 CEO 要筹划市场竞争战略，等等。你将学到：

- 如何聘用并激励最优秀的人才——这是软件项目取得成功的最关键因素；
- 如何让开发时间预估和进度规划变得切实可行，以及为什么有必要这样做；
- 如何设计软件的功能，写出能真正指导开发的规格说明书，而不是制造那种“一次编写、永不阅读”的无用文件，只能拿来把办公室隔间的隔板垒得更高；
- 如何绕开软件开发中常见的陷阱，以及为何程序员编程的时候总喜欢犯“推倒重来”的错误；
- 如何组织和激励团队，为什么程序员需要独立的工作空间和关闭的门；
- 在哪些情况下必须从头开始写每一行代码，抵制从网上下载足够好用软件的诱惑；
- 为什么软件项目总是在过了最初的几个月之后，看起来像是陷入了停滞状态；
- 对于软件公司，发展战略意味着哪些思考和行动，为什么 BeOS 从一开始就注定了失败；
- 除了以上几条之外，还有更多内容。

本书的观点带有强烈的主观色彩。为了行文简洁，我无法在每句话的开头加上“在我看来”这几个字，但实际上你要知道，这本书里的每句话都是我个

人的观点。本书也许并不完善，然而是一个不错的开头。

啊，你是我网站的读者……

本书的很多内容都来自我个人网站上的专栏文章，这些文章统称为“乔尔谈软件”（www.joelonsoftware.com），记录了我在这几年中的很多思考。希望这本让你捧在手书中的书，能比网站上松散的文章在形式上显得更为紧凑，这个词的另一个含义是，“可以在浴缸里阅读而不用担心遭到电击”。

我们把本书分为三大部分。第一部分从小处着眼，探讨了软件开发的方方面面：要想开发出不危害人间的软件，每一个开发团队都应该遵循哪些良好的开发习惯。第二部分包含了一系列关于如何管理程序员和编程团队的文章。第三部分的内容则更松散一些，但大致上围绕着“如何长久地在软件开发行业生存下去”这一主题。你将了解到为什么“膨件”（bloatware）大行其道，为什么本杰瑞公司（Ben & Jerry's）和亚马逊公司如此不同，以及为什么软件开发方法论的出现标志着这个行业缺乏足够数量的优秀人才。

本书的内容当然远不止以上介绍的这些，所以闲言少叙，书归正传。

目录

Contents

■ 第一部分 比特和字节：编程实践	1
01 选择编程语言	2
02 回归本原	4
03 乔尔测试	12
04 软件开发者不可不知的 Unicode 和字符集知识（别找借口！）	24
05 轻松撰写功能规格书 第一部分：为什么要写？	34
06 轻松撰写功能规格书 第二部分：什么是规格书？	40
07 轻松撰写功能规格书 第三部分：但是……怎么做？	49
08 轻松撰写功能规格书 第四部分：写作技巧	53
09 轻松掌控软件开发进度	60
10 每日编译是你的朋友	70
11 完美主义者是如何修复 bug 的	75
12 五个世界	80
13 纸上原型设计	87
14 别被架构航天员忽悠了	89
15 干扰射击	93
16 软件开发是一门手艺活	96
17 计算机科学的三个错误观念	100
18 二元文化主义	105
19 自动从用户那里收集崩溃报告	111

第二部分 管理开发者	121
20 非正式面试指南	122
21 奖励有害论	134
22 不设专职测试人员的 5 条（错误）理由	137
23 任务切换有害论	143
24 永远不要做的事情（第一部分）	146
25 揭开冰山之谜	150
26 抽象必漏定律	156
27 帕默斯顿勋爵谈编程	161
28 评估机能失调	166
第三部分 乔尔语录：中心明确的胡思乱想	169
29 里克·查普曼在“寻找愚蠢”	170
30 这个国家的狗要做什么工作	174
31 普通程序员如何改善团队工作方式	179
32 两个故事	184
33 巨无霸汉堡 vs. “原味主厨”	188
34 凡事没有看上去的那么简单	193
35 为 NIH 综合症说两句	197
36 企业发展战略系列一：本杰瑞 vs. 亚马逊	200
37 企业发展战略系列二：先有鸡还是先有蛋	207
38 企业发展战略系列三：让我回到过去	214
39 企业发展战略系列四：膨胀和二八法则	219
40 企业发展战略系列五：开源软件经济学	223
41 祸不单行的一周	231
42 微软是如何输掉 API 战争的	235
第四部分 关于.NET 的闲言碎语	249
43 走火入魔的微软	250
44 我们向.NET 迁移的策略	255
45 能给我一个链接器吗？	258
第五部分 附录	261
“乔尔问答”精选	262

第一部分

比特和字节：编程实践

-
- 01 选择编程语言
 - 02 回归本原
 - 03 乔尔测试
 - 04 软件开发者不可不知的Unicode和字符集知识（别找借口！）
 - 05 轻松撰写功能规格书 第一部分：为什么要写？
 - 06 轻松撰写功能规格书 第二部分：什么是规格书？
 - 07 轻松撰写功能规格书 第三部分：但是……怎么做？
 - 08 轻松撰写功能规格书 第四部分：写作技巧
 - 09 轻松掌控软件开发进度
 - 10 每日编译是你的朋友
 - 11 完美主义者是如何修复bug的
 - 12 五个世界
 - 13 纸上原型设计
 - 14 别被架构航天员忽悠了
 - 15 干扰射击
 - 16 软件开发是一门手艺活
 - 17 计算机科学的三个错误观念
 - 18 二元文化主义
 - 19 自动从用户那里收集崩溃报告

01

选择编程语言

2002年5月5日，星期日

给定一个开发任务，开发者是如何选择使用哪种编程语言的呢？

如果要求运行速度非常快，我会选择纯 C 语言。

如果要在 Windows 平台上运行，为了让安装包尽可能小，我通常使用 C++ 和静态链接的 MFC。

如果软件的图形界面要跨平台在 Mac、Windows 和 Linux 上都能运行，我通常选 Java。即使界面不会很好看，但能达到跨平台的目的。

在需要快速开发出图形界面和整洁的 UI 时，我倾向于使用 Visual Basic，但代价是安装包会很大，而且只能在 Windows 平台上开发和使用。

在开发一个能运行在 UNIX 机器上的命令行工具，而且不需要很快时，Perl 是个不错的选择。

要让代码在浏览器里运行，JavaScript 确实是唯一的选择。开发一个 SQL 存储过程（stored procedure）的时候，你只能选择某个厂商开发的 SQL 衍生产品。

这意味着什么？

我很少根据语法特性来选择语言。虽然我偏爱带有 {}；（花括号和分号）的语言（C/C++/C#/Java），对于什么样的语法看起来“顺眼”也有自己的评价，但我不会为了能在代码里写分号而增加一个 20MB 的运行库。

这不由得让我对.NET 的跨语言策略心存怀疑。我的观点是, 编程语言那么多, 它们的原理也都相同, 你可以随便选一种。

除了语法上的细微区别, VB.NET 和 C#.NET 几乎完全相同。如果别的语言想加入.NET 阵营, 它必须支持某些核心的功能以及类型, 否则会无法和现有语言配合使用。但是, 怎么用.NET 开发一个 UNIX 命令行工具? 怎么用.NET 开发一个小于 16KB 的 Windows 可执行程序?

这样看来, 在大家最不关心的语法特性方面, .NET 提供了一个选择开发语言的“机会”。

使用.NET 开发一个 UNIX 命令行工具, 需要考虑以下几个方面:

1. 编译后的可执行文件大小。2. 编译后的可执行文件在目标平台上的运行效率。

3. 编译后的可执行文件在目标平台上的兼容性。4. 编译后的可执行文件在目标平台上的可移植性。

5. 编译后的可执行文件在目标平台上的可维护性。6. 编译后的可执行文件在目标平台上的可扩展性。

7. 编译后的可执行文件在目标平台上的可测试性。8. 编译后的可执行文件在目标平台上的可部署性。

9. 编译后的可执行文件在目标平台上的可升级性。10. 编译后的可执行文件在目标平台上的可降级性。

11. 编译后的可执行文件在目标平台上的可配置性。12. 编译后的可执行文件在目标平台上的可定制性。

13. 编译后的可执行文件在目标平台上的可集成性。14. 编译后的可执行文件在目标平台上的可分离性。

15. 编译后的可执行文件在目标平台上的可重用性。16. 编译后的可执行文件在目标平台上的可复用性。

17. 编译后的可执行文件在目标平台上的可共享性。18. 编译后的可执行文件在目标平台上的可协作性。

19. 编译后的可执行文件在目标平台上的可竞争性地。20. 编译后的可执行文件在目标平台上的可同步性。

21. 编译后的可执行文件在目标平台上的可异步性。22. 编译后的可执行文件在目标平台上的可并行性。

23. 编译后的可执行文件在目标平台上的可串行性。24. 编译后的可执行文件在目标平台上的可分布式。

25. 编译后的可执行文件在目标平台上的可集中式。26. 编译后的可执行文件在目标平台上的可混合式。

27. 编译后的可执行文件在目标平台上的可异构式。28. 编译后的可执行文件在目标平台上的可同构式。

29. 编译后的可执行文件在目标平台上的可异构式。30. 编译后的可执行文件在目标平台上的可同构式。

31. 编译后的可执行文件在目标平台上的可异构式。32. 编译后的可执行文件在目标平台上的可同构式。

33. 编译后的可执行文件在目标平台上的可异构式。34. 编译后的可执行文件在目标平台上的可同构式。

35. 编译后的可执行文件在目标平台上的可异构式。36. 编译后的可执行文件在目标平台上的可同构式。

37. 编译后的可执行文件在目标平台上的可异构式。38. 编译后的可执行文件在目标平台上的可同构式。

39. 编译后的可执行文件在目标平台上的可异构式。40. 编译后的可执行文件在目标平台上的可同构式。

41. 编译后的可执行文件在目标平台上的可异构式。42. 编译后的可执行文件在目标平台上的可同构式。

43. 编译后的可执行文件在目标平台上的可异构式。44. 编译后的可执行文件在目标平台上的可同构式。

45. 编译后的可执行文件在目标平台上的可异构式。46. 编译后的可执行文件在目标平台上的可同构式。

47. 编译后的可执行文件在目标平台上的可异构式。48. 编译后的可执行文件在目标平台上的可同构式。

49. 编译后的可执行文件在目标平台上的可异构式。50. 编译后的可执行文件在目标平台上的可同构式。

51. 编译后的可执行文件在目标平台上的可异构式。52. 编译后的可执行文件在目标平台上的可同构式。

53. 编译后的可执行文件在目标平台上的可异构式。54. 编译后的可执行文件在目标平台上的可同构式。

55. 编译后的可执行文件在目标平台上的可异构式。56. 编译后的可执行文件在目标平台上的可同构式。

57. 编译后的可执行文件在目标平台上的可异构式。58. 编译后的可执行文件在目标平台上的可同构式。

59. 编译后的可执行文件在目标平台上的可异构式。60. 编译后的可执行文件在目标平台上的可同构式。

61. 编译后的可执行文件在目标平台上的可异构式。62. 编译后的可执行文件在目标平台上的可同构式。

63. 编译后的可执行文件在目标平台上的可异构式。64. 编译后的可执行文件在目标平台上的可同构式。

65. 编译后的可执行文件在目标平台上的可异构式。66. 编译后的可执行文件在目标平台上的可同构式。

67. 编译后的可执行文件在目标平台上的可异构式。68. 编译后的可执行文件在目标平台上的可同构式。

69. 编译后的可执行文件在目标平台上的可异构式。70. 编译后的可执行文件在目标平台上的可同构式。

71. 编译后的可执行文件在目标平台上的可异构式。72. 编译后的可执行文件在目标平台上的可同构式。

73. 编译后的可执行文件在目标平台上的可异构式。74. 编译后的可执行文件在目标平台上的可同构式。

75. 编译后的可执行文件在目标平台上的可异构式。76. 编译后的可执行文件在目标平台上的可同构式。

77. 编译后的可执行文件在目标平台上的可异构式。78. 编译后的可执行文件在目标平台上的可同构式。

79. 编译后的可执行文件在目标平台上的可异构式。80. 编译后的可执行文件在目标平台上的可同构式。

02

回归本原

2001年12月11日，星期二

我在自己的网站用大量篇幅探讨了很多激动人心的宏观话题,如.NET 和 Java 的比较、XML 使用策略、平台依赖 (Lock-In)、竞争策略、软件设计、架构,等等。这些话题是分层次的,就像一张千层饼:最上面一层是软件策略,往下一层是.NET 之类的开发架构,再下面一层是各种独立的软件产品,如 Java 这样的开发产品和 Windows 这样的平台产品。

我们不妨再深入地探讨一下更底层的话题。动态链接库 (DLL)? 对象? 函数? 不,再往下想! 这就到了一行行的程序代码。

这还不是底层。今天我要说的是 CPU,就是那块把字节搬来搬去的硅晶片。现在把自己想象成第一次学习编程的菜鸟。先把你积累的所有关于编程、软件、项目管理的知识统统放在一边,回归冯·诺依曼体系的最底层。把 J2EE 之类的知识统统搁置脑后,先来思考字节。

为什么要这么做? 我认为,人们在最高层面思考时所犯的错误,其中一部分源于对最底层简单事物的认知存在薄弱环节或者缺陷。这好比盖了一座宏伟的宫殿,地基却是一团糟,用的材料不是厚重的水泥板,而是碎石头。虽然宫殿看上去很漂亮,但是不知怎么回事,浴缸偶尔会在浴室的地板上滑动。

所以现在请深呼吸,让我们来做一个小小的 C 语言练习。

请回忆 C 语言中字符串 (string) 的表示方法:许多字节后边跟着一个值为 0 的 null 字符^①。很容易注意到以下两点。

(1) 除非遍历字符串找到末尾的 null 字符,否则不能知道它在哪结束 (也就

^① 更多关于字符串的信息,参阅 <http://www-ee.eng.hawaii.edu/Courses/EE150/Book/chap7/subsection2.1.1.2.html>。

是不知道字符串的长度)。

(2) 字符串内不能有值为 0 的元素。所以, 不能用 C 的字符串来存储大型二进制对象 (blob, binary large object), 例如一个 JPEG 图像文件。

为什么当初会这么设计 C 的字符串呢? 因为 UNIX 和 C 语言诞生在 PDP-7 微处理器上, 该处理器有一个叫 ASCIZ 的字符串类型。ASCIZ 意思是“末尾带 Z (zero, 零) 的 ASCII 字符串”。

这是存储字符串的唯一方法吗? 当然不是, 而且这其实是一种很糟糕的方法。在开发实用程序、API、操作系统和类库时, 你应该像躲避瘟疫一样避免使用 ASCIZ 类型。为什么这样说呢?

我们先来实现一个 strcat 函数, 也就是把两个字符串连接在一起的函数。

```
void strcat( char* dest, char* src )
{
    while (*dest) dest++;
    while (*dest++ = *src++);
}
```

来看看这段代码做了什么。首先遍历第一个字符串找它的结束符 null。找到后, 遍历第二个字符串, 并在这个过程中把每一个字符逐一复制到第一个字符串的相应位置。

柯尼汉和里奇^①觉得这种字符串处理和字符串连接方式足够好了, 但其实不然。比如, 你想把一堆人名连在一起, 变成一个长长的字符串:

```
char bigString[1000];    /* 我永远不知道分配多少空间才够用 */
bigString[0] = '\0';
strcat(bigString, "John, ");
strcat(bigString, "Paul, ");
strcat(bigString, "George, ");
strcat(bigString, "Joe! ");
```

理论上可行, 是吧? 没错, 而且看起来整齐干净。

但是, 这段代码的性能如何? 还能再快点吗? 数据扩展性如何? 你觉得用这段代码连接 1 百万个字符串, 是个好主意吗?

不。这段代码使用了施勒梅尔算法。施勒梅尔是谁? 就是下面这个笑话的主人公。

^① 布莱恩·柯尼汉 (Brian Kernighan) 和丹尼斯·里奇 (Dennis Ritchie), 《C 程序设计语言 (第 2 版)》作者。

施勒梅尔是个油漆匠，他接了一份活，要在道路中央漆出一条分隔的虚线。第一天，他带了一罐油漆，刷了 900 英尺的路面。“干得不错！”老板说，“你做得很快嘛！”然后付给他一个铜板。

第二天，施勒梅尔刷了 450 英尺。老板说：“虽然不如昨天，不过你干得还是挺快的，450 英尺很不错了。”然后给了他一个铜板。

第三天，施勒梅尔只刷了 90 英尺。“只有 90 英尺！”老板大叫道，“第一天你干的是今天的 10 倍！你怎么了？”施勒梅尔说：“我也没办法啊，每天都离油漆桶越来越远，累死我了！”

（题外测试：施勒梅尔后两天实际走了多少路？）^①这个笑话准确阐释了我刚才写的 `strcat` 函数有什么不好。`strcat` 的第一部分每次运行时都必须遍历目标字符串，去查找结束符 `null`，这部分工作其实是不必要的，所以这个函数的性能远远没有达到最优，数据扩展性也不好。很多人每天写的很多代码都存在这个问题。许多文件系统的实现方法都不适合把很多文件放在同一个文件夹下，因为当一个文件夹中的文件数量达到几千的时候，性能会急剧下降。你可以打开一个塞得满满的 Windows 回收站，感受一下：显示全部文件所需的时间可能长达几个小时，而且很显然，时间不是随着包含文件的数量 n 呈线性增长的。一定是油漆匠施勒梅尔算法在作怪。下次你看到本该是线性的算法性能，却表现出像是 n 平方时，找找隐藏的施勒梅尔吧。最常见的隐藏地是引用的库。当你看到一行行整齐排列的 `strcat`，或是在一个循环中出现了 `strcat` 时，你可能不会马上喊出“ n 平方”，但实际上就是这么回事。

如何修复这个问题？有些聪明的 C 程序员这样实现 `mystrcat` 函数：

```
char* mystrcat( Char* dest, char* src )
{
    while (*dest) dest++;
    while (*dest++ = *src++);
    return --dest;
}
```

这段代码做了什么？我们用很小的代价返回了一个指针，指向连接后那个更长字符串的结尾。这样，调用函数之后，就可以继续在新字符串的末尾进行连接，而不用从头扫描一遍字符串：

^① 关于该数学问题的讨论，参阅 <http://discuss.fogcreek.com/techInterview/default.asp?cmd=show&ixPost=153>。


```
char bigString[1000];    /* 我永远不知道分配多少空间才够用 */
char *p = bigString;
bigString[0] = '\0';
p = mystrcat(p, "John, ");
p = mystrcat(p, "Paul, ");
p = mystrcat(p, "George, ");
p = mystrcat(p, "Joel ");**
```

这段代码的性能是线性的，而不是 n 平方，即使要连接再多的字符串，也不会出现性能下降的问题。

Pascal 的设计者注意到了这个问题，并规定在字符串的第一个字节中存储字符串的长度，从而“修复”了这个问题。这样的字符串叫作 Pascal 字符串。它可以包含 0，不需要用 null 作为结尾。由于字节只能存储从 0 到 255 的数字，Pascal 字符串长度上限为 255 字节，但由于没有 null 尾缀，它和 ASCII 字符串占用同样多的内存。它的好处在于，你不需要循环来得到字符串的长度。在 Pascal 中获取字符串的长度只需一个汇编指令，而不是一个循环。这可快多了。

旧版苹果机 (Macintosh) 操作系统广泛使用 Pascal 字符串。许多 C 程序员在其他平台上使用 Pascal 字符串，以提升速度。Excel 内部使用 Pascal 字符串，一方面，这使得 Excel 中很多地方的长度上限都是 255 字节，另一方面，这也是让 Excel 快如闪电的秘诀之一。

在很长一段时间里，如果你想在 C 代码中写一个 Pascal 字符串，必须这样：

```
char* str = "\006Hello!";
```

没错，你必须亲自数出字符串有几字节，然后把长度硬编码到第一个字节里。有的程序员会偷懒，写出的程序非常慢：

```
char* str = "Hello!";
str[0] = strlen(str) - 1;
```

这样你会得到一个以 null 结尾的字符串（编译器干的好事），同时它也是一个 Pascal 字符串。我以前管这叫混蛋字符串，因为称之为以 null 结尾的 Pascal 字符串有点太长了，但这是个文明的网站，所以没办法，名字长就长点吧。

刚才有一个重要的问题我一带而过了。记得这行代码吗？

```
char bigString[1000];    /* 我永远不知道分配多少空间才够用 */
```

今天的观察视角是深入到比特级的，所以我没有理由忽略它。我要搞定的是这件事：算出结果字符串一共需要多少字节，然后分配大小合适的内存。

这不是每个程序员都该做的吗？

如果不这么做的话，某个聪明的黑客就可能会看到我的代码只分配了 1000B 的内存，通过某种方法让我的 `strcat` 函数处理 1100B 的字符串，造成栈帧被复写，从而修改返回地址，致使函数返回的时候执行那个黑客自己写的代码。经常听人说某某程序受到缓冲区溢出漏洞的威胁，其中的道理就在于此。缓冲区溢出曾经是 IT 系统受到黑客或蠕虫攻击的罪魁祸首。不过那是微软推出 Outlook 之前的事了，Outlook 极大降低了黑客行为的技术门槛，现在十几岁的孩子黑个网站都跟玩儿似的。

所以说被黑客攻击的网站站长是为自己的过失买了单。他们早就该算出需要分配多少内存。

但是，这件事在 C 语言中是很难做到的。回到刚才那个小例子：

```
char bigString[1000];    /* 我永远不知道分配多少空间才够用 */
char *p = bigString;
bigString[0] = '\0';
p = mystrcat(p, "John, ");
p = mystrcat(p, "Paul, ");
p = mystrcat(p, "George, ");
p = mystrcat(p, "Joel ");
```

应该分配多少内存呢？我们来算算看。

```
char* bigString;
int i = 0;
i = strlen("John, ")
    + strlen("Paul, ")
    + strlen("George, ")
    + strlen("Joel ");
bigString = (char*) malloc (i + 1);
/* 记得为结尾的 null 分配内存！ */
...
```

好麻烦，我写得眼睛都有点酸了。你大概也看不下去。我不会责备你，但稍微忍一下，好戏还在后头。

我们先扫描一遍所有的字符串，得到总长度，然后再扫描一遍，把它们连接起来。如果使用 Pascal 字符串，至少 `strlen` 操作是很快的。也许我们可以再写一版 `strcat`，自动地重新分配内存。

这又打开了另一个潘多拉魔盒：内存分配器。你知道 `malloc` 的工作原理吗？`malloc` 函数实际上维护了一个叫作自由链的长链表，它由可用的内存区块组成。调用 `malloc` 时，程序会遍历自由链，查找一个大小符合需要的内存区块。然后它把该区块切成两段，一段的大小等于申请的内存大小，另一段是剩下的字节。前一个区块被返回给调用者使用，后一个区块（如果存在）被

还到链表中。调用 `free` 函数时，程序把释放的区块添加到自由链中。这样一来，自由链会渐渐被切成较小的内存碎片，这时如果你申请一块较大的内存，就没有现成的可用区块了。这时 `malloc` 会暂停工作，开始整理自由链，把相邻的小区块合并为大区块。这大概需要 3 天的时间。本来 `malloc` 就不是很快（总需要遍历自由链），而偶尔在做清理工作时，它会慢得难以预料、令人震惊。（真奇怪，拥有垃圾回收机制的系统的性能恰巧也有这个特点。人们声称垃圾回收机制导致程序性能降低，由此看来这并不完全有道理，因为即使使用传统的 `malloc` 函数，也会导致类似的性能损失，只不过程度较为轻微罢了。）

为了尽可能减少 `malloc` 进行碎片整理的次数，聪明的程序员总是申请大小为 2 的幂的内存区块，也就是 4B、8B、16B、18446744073709551616B，等等。玩过积木的人很容易看出这样做的合理性，也就是尽可能减少自由链中奇形怪状的碎片数量。尽管看上去这样会浪费空间，但显然不会浪费超过 50% 的空间。程序使用的内存不会超过所需内存的 2 倍，这其实还好。

假设你想写一个自动重新分配目标缓冲区的智能 `strcat` 函数，它该总是申请与所需相同大小的内存吗？我的导师 Stan Eisenstat 建议，当调用 `realloc` 时，应该把分配的内存大小调高一倍，这意味着对于长度为 n 的字符串，最多调用 $\lg(n)$ 次 `realloc`，这样即使字符串很长，效率也不会差，而且不会浪费超过 50% 的内存。

总而言之，和字节打交道的时间越久，你的生活就越像是一团乱麻。值得庆幸的是，如今大多数情况下已经并不需要用 C 语言来进行开发了，可以从很多不错的高级语言中选择，包括 Perl、Java、VB 和 XSLT 等。在使用这些语言的时候，不需要考虑底层的实现细节，很多问题在语言层面就得到了妥善的处理，不需要开发者操心；但是，偶尔还是会暴露出一些底层的问题，比如，需要在使用 `String` 还是 `StringBuilder` 之间作出选择。编译器还没有那么智能，无法猜出我们到底想要做什么，它只能试着提供帮助，避免我们在无意中写出施梅勒尔算法。

上周我在文章中提到，如果在 XML 中存储数据，要实现 SQL 语句 `SELECT author FROM books` 就不可能很快^①。当时可能没有讲得很清楚，而经过刚才在 CPU 里这一番摸爬滚打，我们再来看看为什么是这样。

关系型数据库是怎么实现 `SELECT author FROM books` 的呢？在关系型数据

① 参阅 <http://www.joelonsoftware.com/articles/fog0000000296.html>。

库里，一张表（例如 `books`）里的每一行都有相同的字节数，也就是说，从行首到每个字段的偏移量都是固定的。所以，如果 `books` 中的每一条记录的长度都是 100B，且 `author` 字段的偏移量是 23，那么 `authors` 数据存储在以 23、123、223、323 等开头的字节中。具体到这个查询，移动到下一条记录的代码就是：

```
pointer += 100;
```

只用一条 CPU 指令，相当快。

然后我们来看 XML 中的 `books` 表。

```
<?xml blah blah>
<books>
  <book>
    <title>UI Design for Programmers</title>
    <author>Joel Spolsky</author>
  </book>
  <book>
    <title>The Chop Suey Club</title>
    <author>Bruce Weber</author>
  </book>
</books>
```

请听题：移动到下一条记录的代码怎么写？

这个……

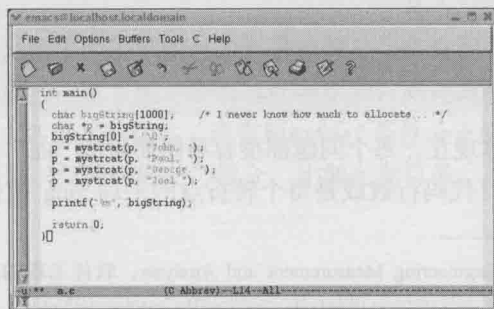
一个优秀的程序员会说，嗯，我们可以先把 XML 解析为内存中的树结构，以便快速对其操作。要完成 `SELECT author FROM books`，CPU 所要做的工作多到让人想哭。写过编译器的人都知道，词法扫描和解析是编译过程中最耗时的环节。这么说吧，词法扫描、解析、在内存中构建抽象句法树的过程，中间涉及很多的字符串操作、很多的内存分配操作，而我们刚才看到，这两种操作都是非常慢的。这里还隐含了一个假定，即内存足够大，可以把所有的东西都一次性装下。对于关系型数据库，在记录与记录间移动的效率是常数级别的，准确地说就是一个 CPU 指令。数据库在设计之初就是这么考虑的。而且借助于内存映射文件，只需要加载那些真正会使用到的磁盘分页即可。再来看 XML，如果你进行预解析，虽然在记录与记录间移动的效率也是常数级别的，但启动时间非常久；如果你不预先解析，移动的效率取决于前面所有记录的长度，至少也有几百个 CPU 指令。

我认为这意味着，如果运行效率很关键，而且数据很多，就不要使用 XML。如果你的数据不多，或者不是很关心运行的效率，那么 XML 还是一个不错

的格式。如果你想鱼与熊掌兼得，就需要找到一种方法，在 XML 的数据边上存储元数据，和 Pascal 字符串中显式存储字符串长度的做法类似，表明数据在文件中的位置，从而免去解析和扫描的过程。但是，这么一来你就不能直接用文本编辑器编辑文件了，否则会搞乱元数据。而且，这样的格式也不再是标准的 XML 了。

能坚持读到这里的读者，你们都是大好人，希望我的文字让你们学到或是重新思考了一些东西。希望思考诸如 `strcat` 和 `malloc` 函数的工作原理等在计算机本科一年级就学过的无聊玩意，能让大家温故而知新，做出最新的、高端的、战略和架构层面的决策，并且知道什么时候该用 XML 这样的技术。作为家庭作业，请思考为什么全美达 (Transmeta) 的芯片总是感觉慢吞吞的；或者为什么最初版本的 HTML 中的表格设计会那么糟糕，导致使用调制解调器的人们总是要等好长时间才能看到网页上很长的表；或者为什么 COM 在进程内部运行很快，而跨进程运行时就慢下来了；或者为什么 NT 的开发者把显示设备驱动放到内核空间而不是用户空间中。

要给出这些问题的答案，你需要对字节有一定的认识，因为字节影响着我们在架构和策略层面所做出的高端决策。所以在计算机技术学习方面，我有一个观点：计算机系的新生必须从基础学起，从使用 C 语言、认识 CPU 开始，自底向上地构建知识体系。我非常反感一些所谓计算机科学课程，把 Java 作为入门语言，因为它“简单”，不需要接触那些令人困惑又无聊的 `string`、`malloc` 等，就能学到很酷的面向对象编程的内容，让你编写的大型程序总是十分模块化。这样的教学思想简直是在养虎贻患，将导致一代又一代的毕业生进入软件开发领域后，到处留下油漆匠施梅勒尔的算法而浑然不觉，因为只学过 Perl 脚本的他们根本不知道，字符串在底层其实是很难处理的。要想真正授人以渔，必须先从最底层的基础知识教起。就好像电影《练空手道的孩子》(Karate Kid) 一样。上蜡、去蜡，上蜡、去蜡……连续苦练 3 个星期之后，击倒对手就易如反掌了。



03

乔尔测试

2000年8月9日，星期三

你有没有听说过 SEMA^①？这是一个非常复杂的评价系统，用来衡量软件团队的优劣。等一下！不要读关于 SEMA 的书！要理解它差不多得花上 6 年时间。因此，我自己写了一套评价软件团队质量的测试，非常业余，仅供参考。但好的一面是，只需要大概 3 分钟就能做完这个测试，省出来的时间足够你读一个医学学位了。

乔尔测试

- (1) 你们用源代码管理系统吗？
- (2) 你们能一键编译吗？
- (3) 你们做每日编译吗？
- (4) 你们有 bug 数据库吗？
- (5) 你们在写新代码前修改以前的 bug 吗？
- (6) 你们的进度表是最新的吗？
- (7) 你们有软件规格书吗？
- (8) 程序员的工作环境安静吗？
- (9) 你们使用了能买到的最好工具吗？
- (10) 你们有测试人员吗？
- (11) 你们面试时会要求应聘人员写代码吗？
- (12) 你们做过走廊可用性测试吗？

乔尔测试的简洁体现在，每个问题都很容易用简短的“是”或“否”回答。不需要你统计每日代码行数或是每个转折点的平均 bug 数量。每回答一个

① 即 Software Engineering Measurement and Analysis，软件工程的度量与分析，参阅 <http://resources.zsei.cmu.edu/library/asset-view.cfm?assetid=19580>。

“是”，就给你的团队加上 1 分。乔尔测试也有缺点，就是它确实不能用于保证软件的安全性，尤其是核电站管控系统之类的软件。

得到 12 分是完美的，11 分尚可，但是 10 分或更低意味着你们的开发团队出了很严重的问题。事实上，大部分软件开发团队的得分只有 2 或 3，真的是亟待改进。相比之下，微软这样的公司任何时候都在以 12 分的水平运作。

当然，决定开发活动成败的因素并不只是以上列出的这些，尤其是当产品本身乏善可陈时，即使软件团队再好，也还是没有人会买你的产品。当然，如果你的队伍中全部都是无招胜有招的“世外高人”，那么无需满足这些条件，也能做出改变世界的惊人软件。但总而言之，做到以上 12 点，你就能拥有一个能够不断交付的可靠团队。



1. 你们用源代码管理系统吗？

我用过收费的源代码管理工具，也用过免费的 CVS^①，根据我的经验，CVS 足够了。然而，如果你不用源代码管理系统，那么就很难使得程序员们协同工作。程序员没办法知道别人做了什么。一旦出现错误，很难恢复到原来正常的状态。使用源代码管理系统的另一个好处是，签出功能使得每个程序员的机器上都存有源代码，我从没听说过用了源代码管理的项目丢失过大量代码的案例。



2. 你们能一键编译吗？

换句话说，从最新版本的源代码开始，到打包成一个可以交付的软件为止，需要几步操作？好的团队只用一个脚本就能从头开始完全签出代码，重编译每一行代码，做出各种发行版本、语言和 `#ifdef` 条件分支组合的所有可执行文件，生成安装包以及最终的交付媒介，比如光盘目录、下载页面等。

① CVS 是 Concurrent Versions System 的缩写，即并发版本管理系统。——编者注

如果整个过程多于一步，就容易忙中出错。离交付日期越近，往往就越会想更快地迭代，修复“最后一个”bug，做出最终版本的可执行文件，等等。如果每次编译代码、构建安装包都要经过 20 个步骤，开发者不仅会被逼疯，还会犯低级的错误。

出于这个原因，我就职的上一家公司放弃了 WISE，转投 InstallShield 的怀抱。我们要求安装进程必须能从一个脚本中自动运行，一夜完成，并且使用 NT 的计划任务。WISE 做不到从计划任务中启动并且一夜完成，所以我们抛弃了它。（WISE 公司的好心人向我保证，最新版本已经支持每夜编译了。）

3. 你们做每日编译吗？

使用源代码管理时，某个程序员有时会不小心签入让编译不能通过的故障代码。例如，某人添加了一个源文件，在自己的机器上可以正常通过编译，但是却忘了把新文件加到代码库中，就锁定机器，高高兴兴地下班回家了。结果其他所有人都无法工作了，被迫回家，相比之下他们可不太高兴。

破坏编译是非常有害（也十分常见）的开发问题，因此有必要实践每日编译，确保及时发现故障代码。对于大型团队，有一个方法能保证故障代码立刻得到修复，那就是每天在固定的时候进行编译，比如午饭时间。每个人尽可能多地在午饭前签入代码。大家吃完饭回来，编译已经完成。如果全部通过，那再好不过了。大家可以签出最新版本的代码继续工作。如果编译失败，那么责任人自行修复代码，而其他人则不受影响，在前一个可用的构建版本上继续工作。

在 Excel 团队，我们有一条规矩，谁的代码破坏了编译，谁就负责构建的维护作为“惩罚”，直到另一个人犯错替代他为止。这对于保持编译不被破坏有很好的激励作用，也提供了一个机制，通过大家轮流负责代码的编译工作，让每个人都熟悉它的过程和原理。

阅读“每日编译是你的朋友”（第 10 章）^①，获取更多相关知识。

^① 参阅<http://www.joelonsoftware.com/articles/fog0000000023.html>。

4. 你们有bug数据库吗?

不管你有何种理由,只要做代码开发,即使团队只有一个人,如果没有一个列出代码中所有已知 bug 的数据库,那么这个团队交付的代码质量一定不高。许多程序员自以为可以在自己的脑袋中维护 bug 列表,真是痴心妄想。我最多能同时记住两三个 bug,可能睡一觉醒来,或者是在临近交付日的忙乱之中就全忘了。你必须正式地记录下每一个 bug,内容至少应包括:

- 重现 bug 的完整步骤;
- 预期应有的行为;
- 观察到的(错误)行为;
- 谁负责修复;
- 是否已被修复。

如果导致你不做 bug 记录的唯一阻碍是相关软件的功能太复杂了,那么就做一张包含以上 5 个关键字段的表,然后开始用它。

关于如何记录 bug,请阅读“轻松追踪 bug”^①。

5. 你们在写新代码前修改以前的bug吗?

微软开发第一版 Windows 字处理软件 Word 时的惨痛经历可以用“死亡之旅”来形容,耗时极长,举步维艰。尽管整个团队都在没日没夜地加班,项目进度却一拖再拖,压力空前。几年后,这个产品终于交付了,微软把整个 Word 项目组送到墨西哥观光胜地坎昆去度假,并且开始进行深刻的反思。

他们意识到问题出在项目经理严格按照进度表推进,而正式的进度表中根本没有预留修复 bug 这一环节的时间,导致程序员们疲于奔命,敷衍了事。没有人关心如何减少 bug 数量,代码质量之差可想而知。据说曾有一个程序员,在写计算一行文字高度的代码时,只是写了 `return 12;`,然后就坐等测试人

^① 参阅<http://www.joelonsoftware.com/articles/fog0000000029.html>。

员提交 bug，告诉他代码结果并不总是正确的。这种进度表其实就是一张功能点的检查表，按照它只能把一个个的功能点转化成一个个的 bug。在事后的调查报告中，这种情形被称为“无尽缺陷方法论”。

为了修正这个问题，微软开始在全公司实行所谓的“零缺陷方法论”。许多程序员都在摇头偷笑，认为管理者是妄想用他们的行政命令手段来压低 bug 数量。实际上，“零缺陷”的意思是无论任何时间，在写新代码前都要优先消灭现有 bug。这么做的原因如下。

通常来讲，一个 bug 在修复之前拖得越久，修复它的成本（时间和金钱）就越高。

例如，修复拼写错误或语法错误这类编译器就能捕获的 bug，基本上不费吹灰之力。

当你首次尝试运行某段代码时发现 bug，可以很快将其修复，因为你还清晰地记得所有的代码。

当你在几天前写的代码中发现一个 bug 时，就得花一点时间才能找到出错原因，然而在你阅读自己写的代码时，还是能回忆起当时的思路，并且在很短时间内修复 bug。

但如果你在几个月前写的代码中发现一个 bug，可能已经忘了代码的很多具体细节，要修复 bug 就更难了。时隔几个月，你还有可能是在修复别人的 bug，而犯错的那个人正在阿鲁巴岛度假。在这种情况下，修复 bug 就更像是一门科学了：你必须放慢脚步、有条不紊、小心翼翼，而且无法估计找到修复的方法需要多长时间。

如果你在已经交付的代码中发现一个 bug，要修复它，你将付出难以想象的代价。

所以，你要及早发现并修复 bug，原因之一是这样耗时更短。另一个原因是，估计写一段新代码的时间，比估计修复现有 bug 的时间要容易。举个例子，如果我问你写一个列表排序大概要多长时间，你可以给出一个相对准确的估计。但如果我说，你的代码在安装了 IE 5.5 的机器上出现错误，你认为大概花多长时间能找出问题所在？你没法猜，因为根本不知道是什么原因导致的错误。可能要花 3 天才能找到原因，也可能两分钟就找到了。

也就是说，如果你的进度表里充斥着各种需要修复的 bug，那么这份进度表

完全不可靠。但如果你已经修复了所有已知的 bug，剩下的工作都是写新的代码，那么进度表会是惊人地精确。

保持 bug 数量为 0 的另一个好处是，你可以在竞争中保持较快的反应速度。有些开发者把这个原则表述为让产品始终保持可发布的状态。一旦对手引入一项新的杀手级功能并有可能从你手中抢走生意时，你就可以快速实现同样的功能，并且立刻交付，前提是你不需要修复以前积攒的 bug。

6. 你们的进度表是最新的吗？

下面我们来说说进度表。如果代码有商业价值，投资者有很多理由要求知道什么时候能写完。而在制定和履行进度表方面，程序员是出了名地不靠谱。“当它该写完的时候就写完了！”他们会对商人大喊。

很遗憾，这样的答案难以令人满意。在代码交付之前，商人需要预先做很多计划和决策：软件功能预览、展会亮相、广告投放，等等。这一切都有赖于你的进度表，而且必须是最新的。

进度表的另一个关键作用是，它会迫使你做出选择，思考哪些功能必须要开发，哪些功能不重要。时间不够就要砍掉不重要的，从而免于患上功能泛滥症^①（也叫边界增生）。

按计划推进其实没那么难。我在第 9 章介绍了一个很简单的方法，教你怎么做好计划。

7. 你们有软件规格书吗？

写软件规格书就像使用牙线：大家都觉得是件好事，但就是没人做。

我不确定导致这种情况的确切原因，但有可能是因为绝大部分程序员讨厌写文档。所以，全部由程序员组成的团队，通常会在代码而不是文档中表述他

^① 功能泛滥症的定义参阅http://www.netmeg.net/jargon/terms/c/creeping_featuritis.html。

们对特定问题的解决方案。面对开发任务，他们更倾向于一头扎进代码中解决实际问题，而不愿事先撰写一份软件规格书。

在设计阶段发现了问题，可以通过修改几行文本来改正。而一旦写成了代码，修复问题的成本就急剧升高，其中既有情绪因素（人们不愿丢掉代码），也有时间因素，所以修复问题会遇到很大阻力。没有规格书的指导，软件的设计最终会很糟糕，进度也会失控。网景公司似乎就遇到了这个问题，某个软件经过头4个版本的迭代，变成一团乱麻，致使管理层作出一个愚蠢的决定，抛弃了所有写过的代码，从头再来^①。同样的一批人后来又又在谋智公司（Mozilla）犯了同样的错误，结果创造出一头失控的软件“怪兽”，耗时几年才使其成熟进入 alpha 阶段。

我有一个小小的理论，要解决这个问题，或许可以让程序员学习写作强化训练课程^②，来减少他们对写作的抵触。另一个解决方案是，聘用有能力的项目经理来写软件规格书。这两种方案都要求贯彻一条简单的规则：无规格，不编程。

阅读第5章～第8章，你可以了解到撰写规格书的方方面面。

8. 程序员的工作环境安静吗？

有大量研究证据表明，通过改善脑力劳动者的工作环境，包括提供足够的空间、安静的环境并确保隐私受保护，能够提高他们的生产效率。经典的软件管理图书《人件》^③中详细记述了这些因素对生产效率的促进作用。

下面介绍一下原因。我们都知道脑力劳动者在“心流”状态（也称为“进入状态”）时做得最好，那时他们一心扑在工作上，模糊了对周围环境的感知。他们忽略了时间，在精神高度集中的状态中创造出伟大的作品。这正是完成一切高效工作的时候。作家、程序员、科学家，甚至是篮球运动员都会有这种“进入状态”的体验。

① 见第24章，<http://www.joelonsoftware.com/articles/fog0000000069.html>。

② 例如，耶鲁大学的“每日主题”（Daily Themes）课程<http://english.yale.edu/courses/daily-themes>就以要求学生每天撰写一篇短文而闻名。

③ 汤姆·德马科，蒂姆·李斯特，《人件（第2版）》。

但问题是，“进入状态”谈何容易。平均起来，需要大概 15 分钟才能进入最有效率的状态。当你感到疲倦，或是当天已经做了许多有创造力的工作时，就再也无法进入状态，在剩下的工作时间里只能消磨时间，浏览网页或是玩俄罗斯方块。

另一个问题是进入状态的人很容易被打扰。噪音、电话、外出午餐、开 5 分钟的车去星巴克喝咖啡，尤其是被同事打断，都会把你从高效工作的状态中踢出来。某个同事问你一个问题，可能只花了 1 分钟时间，而这件事的影响会让你在之后长达半个小时的时间里，都难以回到原来的效率，从而使你总体的生产效率大打折扣。如果你的工作环境像牛栏一样吵——热情高涨的互联网公司都喜欢创造这样的工作环境，程序员的工位旁边就是对着电话咆哮的销售人员——那么软件开发者的效率将急转直下，因为脑力活动会一次又一次地被打断，永远都进不了状态。

对于程序员，这尤其是个难题。编程的效率，依赖程序员在短期记忆中操控大量的细枝末节。就像同时抛接多个小球的杂耍一样，任何打断都有可能让这些小球掉落在地。你重新开始工作时，没法回忆起这些细节（比如正在使用的局部变量名称的含义，或者搜索算法的实现将要进入什么状态），只能从头开始逐个查找这些细节，从而进一步拖慢了回到全速运转状态的时间。

这是很简单的算术题。假定（从我的经验来看是这样）我们打扰某个程序员，哪怕只有 1 分钟，实际上都破坏了 15 分钟的生产力。举个例子，把两名程序员，杰夫和马特，放在相邻的开放式办公隔间里，就好像呆伯特漫画中的场景一样。马特忘了 `strcpy` 函数的 Unicode 版本名。他自己可以在 30 秒内查到，也可以花 15 秒问问杰夫。由于就坐在杰夫旁边，他选择了问杰夫。杰夫因为受到打扰，失去了 15 分钟的生产力（为马特节省了 15 秒）。

现在把他们挪到两个用墙和门分隔开的办公室里。如果马特忘了函数名，还是可以选择花 30 秒查到，或者选择去隔壁问杰夫，不过这回得花 45 秒，而且需要站起来（鉴于一般程序员的身体素质，要站起来可不轻松）。所以他选择自己查。结果马特损失了 30 秒，而我们为杰夫省下了 15 分钟。啊哈！



9. 你们使用了能买到的最好工具吗？

普通家用电脑所不能胜任的任务已经不多，而使用编译语言写程序代码是

其中一件。如果你每次编译所花费的时长超过了几秒钟，请考虑换一台最新最好的电脑，这会节省不少时间。如果编译时间超过 15 秒，程序员就会在等待的无聊中抽空去看《洋葱新闻》^①，继而不能自拔，几个小时的生产力就这么没了。

用单显示器调试图形界面非常痛苦，甚至是不可行的。双显示器会让编写图形界面的工作变得更简单。

大多数程序员在编写图标或工具栏时都需要操作位图数据，而不是每个程序员都有一个好用的位图编辑器。用微软画图程序去操作位图数据就是个笑话，但大部分程序员只能这么做。

我在上一家公司工作时^②，系统管理员总是给我发自动邮件，抱怨说我使用了超过 220MB 的服务器硬盘空间。我指出，就目前硬件的价格来看，这点空间的成本比我用掉的卫生纸的成本低得多。花 10 分钟清理我的工作目录都是严重浪费生产力的行为。

顶尖的开发团队是不会亏待程序员的。不好用的工具造成的挫折感即使很小，但是累加起来也会让程序员感到郁闷、不幸福。郁闷的程序员就是没有效率的程序员。

话又说回来，程序员很容易被最酷、最新的工具打动，要让他们死心塌地为你干活，让他们用最好的工具可比真正付给他们有竞争力的薪水要便宜多了！

10. 你们有测试人员吗？

如果你的团队没有专职测试人员，或者数量没有达到两三个程序员配备一个测试人员，那么你要么会交付有很多缺陷的产品，要么你是在浪费钱，让时薪 100 美元的程序员做时薪 30 美元的测试员的工作。在我看来，为了省钱而不配备测试人员实在是拣了芝麻丢西瓜，很奇怪为什么大家没有意识到这一点。关于这个问题，请阅读第 22 章“不设专职测试人员的 5 条（错误的）理由”。

① 参阅<http://www.theonion.com/>。

② 见第 32 章，<http://www.joelonsoftware.com/articles/TwoStories.html>。

11. 你们面试时会要求应聘人员写代码吗?

你在雇佣一个魔术师之前，难道不让他先变个戏法露一手吗？当然会了。你在为自己的婚礼选定餐饮服务商之前，不先尝尝他们的菜吗？应该都会尝的。（除非遇到玛芝姑姑，如果不用她的“秘制”炒肝蛋糕招待宾客，她会记恨你一辈子。）

但是，程序员经常因为简历写得令人印象深刻或是和面试官聊得很投缘就被录用。有的面试官则喜欢考一些无关痛痒的问题（“CreateDialog() 和 DialogBox() 有什么区别？”），只要查一下文档就能找到答案。面试官应该考察的并不是面试者能否记住成千上万有关编程的技术细节，而是他们写代码的能力。还有一类面试问题更糟，即所谓的“啊哈”问题：如果你知道答案，就会觉得非常简单，而如果你不知道，就几乎不可能自己想出答案。

恳请各位面试官，不要再问这类问题了。面试时无论如何要让应聘者写点代码。（阅读本书第 20 章以获得更多建议。）

12. 你们做过走廊可用性测试吗?

走廊可用性测试就是在走廊随便拉住一个人，然后强迫他（她）使用你刚写的程序。如果你让 5 个人做过这个测试，那么 95% 的程序可用性问题都会暴露出来。

优秀的用户界面设计没有想象中那么难，而且至关重要，它让用户喜爱你的产品，增强购买的意愿。可以读读我写的《面向程序员的用户界面设计》^①，这是一本写给程序员看的简明设计入门书籍。

用户界面方面的问题有一个突出的特点，如果你把程序展示给一群人（其实五六个就够了），你很快就会总结出最不令人满意的地方。雅各布·尼尔森

^① Joel Spolsky, *User Interface Design for Programmers* (Apress, 2001)。免费在线试读，参阅 <http://www.joelonsoftware.com/uibook/chapters/fog0000000057.html>。

的文章^①对此给出了解释。即使缺乏 UI 设计方面的技巧,在强迫自己做过(免费的)走廊可用性测试之后,你的用户界面将会得到很大的改善。

使用乔尔测试的四种方法

- (1) 为你所在的软件开发机构评分,然后告诉我,我可以帮你散布消息。
- (2) 如果你负责管理一个编程团队,可以把乔尔测试当作一张检查表,确保你的团队处在最佳的工作状态。一旦团队评分达到 12,你就可以放手让程序员去做事,自己全职负责屏蔽销售团队对他们的干扰。
- (3) 如果你正在犹豫是否接受某个编程工作,可以询问潜在雇主在这个测试上的分数。如果得分过低,一定要确认自己有改变团队开发习惯的权力。否则,你在那里会十分沮丧、效率低下。
- (4) 如果你是一名投资者,正在做尽职调查,评估某个编程团队的价值,或者你的软件公司正在考虑和另一家公司进行并购,就可以使用乔尔测试,作为一种粗略的经验法则。

22

第一部分
比特和字节：
编程实践

后记：2004 年 6 月 14 日

自从 2000 年 8 月乔尔测试公布起,我就被淹没在来自世界各地开发者的电子邮件中,他们向我报告所在开发团队的乔尔测试分数。虽然结果的分布非常平均,我还是得说,绝大部分开发机构的得分都在 2 和 3 之间,这值得我们思考。还有很多开发者反映,他们受够了粗放松散的“武林高手”式的开发管理,因此在找新工作的时候,会拒绝那些乔尔测试得分过低的公司。我也从许多团队管理者那里收到了正面的反馈,他们按照乔尔测试中的条目,逐步改进团队的开发效率、提升测试的分数。

与此同时,许多开发机构的问题已经不是乔尔测试能够涵盖的了,官僚主义已经病入膏肓。主要表现是人们准备会议的时间比开发软件的时间还要长。

① 雅各布·尼尔森 (Jakob Nielsen), “Why You Only Need to Test with 5 Users”, useit.com, 2000 年 3 月 19 日。参阅 <http://www.useit.com/alertbox/20000319.html>。

某个机构可能在乔尔测试中得到了 12 分，却由于各种办公室政治问题和形式主义搞得一事无成。悄悄告诉你，自从 20 世纪 90 年代我在微软工作的那段时间之后，所有证据都表明，微软因为其庞大的体量、内部的政治斗争以及官僚主义造成的浪费，逐渐步入了滞涨阶段。你要我拿出证据？只有微软的中层经理才会喜欢 Tablet PC 这个产品，它的设计目的似乎就是让华盛顿州雷德蒙德市的项目经理们即使整天泡在会议室里，也能够处理部分日常工作邮件。这并不是微软独有的现象。如果你发现，自己花了很多时间安装和配置大型的软件方法论系统，或是需要名为“可视化某某系统的企业级架构版”的软件才能工作，或者跑来跑去、到处向团队成员讲解极限编程理论和 UML 图，让大家叫苦不迭，根本无法安心工作，在这种情况下，即使团队在乔尔测试中表现得很好，你们可能还是会遇上很大的麻烦。

04

软件开发者的不可不知的Unicode和字符集知识（别找借口！）

2003年10月8日，星期三

你是否熟悉 HTML 页面中的 Content-Type 标签？你是否不确定该在里面填什么值？

你是否接到过来自保加利亚友人的电子邮件，打开它发现主题栏显示为“???? ?????? ??? ?????”？

我发现许多软件开发者的都不太了解字符集、编码、Unicode 等概念，认为它们有某种神秘感，这让我非常担忧。几年前，在 FogBUGZ^① 的 beta 测试期间，有用户提出是否能支持对日语电子邮件的处理。日语？他们还有用日语写的电子邮件？我并不清楚。我们用商业版 ActiveX 控件来解析 MIME 类型中的电子邮件消息，经过调研发现，这个控件在处理字符集上犯了很大的错误，导致我们不得不写很多代码，先撤销错误的转换，然后重新进行正确的处理。我转而研究另一个商业库，发现那个库对字符集的实现同样很差。我给那个库的开发者写信，得到的回复是，他感觉“无能为力”。他的想法和许多程序员一样，认为这些问题不会造成大麻烦，总会过去的。

但事情并没有这么简单。目前流行的网络开发工具 PHP 几乎完全忽视了字符编码的问题^②，草率地用 8 比特表示每个字符，这严重阻碍了网络应用的国际化开发。事已至此，我再也看不下去了。

我宣布：21 世纪的程序员们听好了，你们中要是有什么不知道有关字符、字

① 我们的 bug 追踪产品，参阅<http://www.fogcreek.com/FogBUGZ>。

② 参阅<http://ca3.php.net/manual/en/language.types.string.php>。

符集、编码和 Unicode 的基本知识，如果让我抓到你，我会罚你在一艘潜艇里剥 6 个月的洋葱，我说到做到。

而且话说回来：

这些知识没那么难学。

这篇文章里，我会告诉你们每个写代码的程序员都应该知道的知识。所有那些“纯文本 = ASCII = 所有字符，都是 8 比特”的想法，不仅是错误的，而且错得不可救药。还在这么编程的人，其愚蠢程度简直和不相信微生物存在的医生一样。请不要接着写代码了，先读完这篇文章再说。

在开讲之前，我要说明一下，少数对国际化知识有所了解的人，可能觉得我的所有论述有点过于简化了。我想做的是提高编程人群最低限度的知识水平，否则我们没法处理纯正英语之外的任何语言，就连英语的变种方言都不行。还要指出的是，字符处理只是软件开发走向国际化的第一步，但是路要一步一步地走，文章要一篇一篇地写，我今天就讲字符集。

历史视角

要理解这些事物，最好从历史说起。你可能觉得我要讲非常古老的字符集，比如 EBCDIC，那你就错了。EBCDIC 和你的生活没有关系，我们不需要回溯那么远。

在不那么久远的过去，当 UNIX 诞生、K&R 在写《C 程序设计语言》的时候，一切都是那么简单。EBCDIC 正在衰落。唯一常用的字符集是大家的老朋友，纯正的英语字母，代号为 ASCII，它能用 32 到 127 之间的数字来表示各种字符^①。例如，空格是 32，字母 A 是 65，等等。该字符集可以很方便地存储在 7 比特里。当时的大部分计算机都用 8 比特表示 1B，所以不仅可以存储所有可能的 ASCII 字符，而且还多出 1 比特。如果愿意，你可以利用这 1 比特来达到特殊的目的，比如 WordStar 公司的蠢货们就用首位置 1 来表示该字符位于词尾，致使 WordStar 软件注定只能处理英语文本。编号在 32 以下的字符叫作不可打印字符，是用来骂人的。开个玩笑，它们是用来做控制字符的，

① 更多关于 ASCII 字符的信息，参阅 <http://www.robelle.com/library/smugbook/ascii.html>。

比如 7 让你的计算机发出提示音，12 能控制打印机送出当前的纸页，并填入新页。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	space	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6		a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

对于英语使用者来说，这一切还不错。

由于 1B 最多能装下 8 比特，很多人就想：“为什么不把编号为 128~255 的空间也利用起来？”问题是当时有许多人都有这个想法，而对于这些数字该表示什么意思，大家意见不一。IBM-PC 发展出了后来叫作 OEM 的字符集，它支持了某些欧洲语言中出现的方言字符，以及一堆画线用的符号，包括横线、竖线、右侧带小线条的横线，等等。你可以用这些字符在屏幕上画出漂亮的方框、线条等图像。直到今天，你还可以在干洗店的 8088 计算机上看到这些图像。从个人计算机销售到美国以外的地区起，五花八门的 OEM 字符集就如雨后春笋般被发明出来，每一种字符集对高位 128 个字符的解释都不一样。比如某些个人计算机上编号为 130 的字符显示为 é，同样的字符在以色列出售的计算机上则显示为希伯来字母 Gimel (ג)，所以当美国人把 résumés (简历) 送到以色列时，就会显示为 rsums。经常出现同一种语言中对高位 128 个字符有很多种解释的情况，例如俄文，因而几乎不可能可靠地传输俄文的文件。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
3	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
4	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
5	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
6	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
7	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
9	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
A	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
B	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
C	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
D	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
E	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

这种 OEM 的乱象最终被 ANSI 标准化了。在 ANSI 标准中,人们对低位 128 个字符的含义达成了共识,基本上和 ASCII 一致,但根据地区的不同,对编号为 128 及以上的字符的处理方式也不同。这些子系统被称作代码页^①。举例来讲,在以色列, DOS 使用编号为 862 的代码页,而希腊用户使用编号为 737 的代码页。二者编号在 128 以下的字符是相同的,128 及以上则不同,那些看起来奇形怪状的字符都存储在这里。

MS-DOS 在各个国家一共有几十种代码页,支持的语言从英语到冰岛语应有尽有,甚至有“多语言”代码页,可以在同一台电脑上处理世界语(Esperanto)和加利西亚语(Galician)! 好神奇!但是有的语言,比如希伯来语和希腊语就不能在同一台电脑上使用,除非你写个程序用位图来显示所有字符,因为这两种语言的代码页不同,对高位的编号有不同的解释。

而事情在亚洲变得更疯狂,因为亚洲语言的“字母表”里有上万个“字母”,不可能装在 8 比特里。通常解决的办法用一种叫做 DBCS 的杂乱系统,即双字节字符集(Double Byte Character Set),里面有些字母用 1B 存储,另一些用 2B。在字符串中向尾部移动很容易,而向首部移动几乎是不可能的。最好不要直接用 s++ 和 s-- 在字符串中移动,而是使用 Windows 提供的 AnsiNext 和 AnsiPrev,这些函数知道怎么处理这种混乱的局面。

但是,绝大多数人仍然假装 1B 就是一个字符,一个字符就是 8 比特,只要你永远不在计算机之间传输字符串,或者使用两种以上的语言,这基本上就不会错。但显然自从互联网诞生起,在计算机之间传输字符串就变得越来越常见,混乱一发而不可收拾。幸运的是,Unicode 应运而生。



Unicode

Unicode 是一个创举,它旨在用一套字符集,支持地球上所有自然的书写系统,甚至还支持某些杜撰出来的书写系统,如克林贡语(Klingon)。有些人误以为 Unicode 只是一种 16 位的编码,即每个字符有 16 比特,总共有 65536 种可能的字符。其实不然。这是对 Unicode 最常见的一种误解,所以如果你

^① 更多关于代码页的信息,参阅[http://technet.microsoft.com/zh-cn/library/2x8et5ee\(v=vs.110\).aspx](http://technet.microsoft.com/zh-cn/library/2x8et5ee(v=vs.110).aspx)。

也这么认为，别太难过。

实际上 Unicode 用了一种不同的思路来表示字符，你必须理解这种思路，才能看出其中的玄妙。

在之前的所有字符集中，我们都假设一个字母映射为一些比特，然后存储到磁盘或内存中：

```
A -> 0100 0001
```

在 Unicode 中，一个字母映射为一个理论概念：码点。至于如何在内存或硬盘中表示码点，就是另一件事了。

在 Unicode 中，字母 A 是一个柏拉图的理型 (platonian ideal)。它漂浮在天上，有些抽象：

A

这个 A 的理型不同于 B，和 a 也不一样，但是和其他字体、字号的 A 一样。Times New Roman 字体和 Helvetica 字体中的 A 是一样的，但是不同于小写的 a。这个想法本身没什么争议，但是在某些语言中，某个字母表示什么是有争议的。德语中的 ß 是一个独立的字母，还是书写 ss 的变体？如果某个字母的形状在词尾发生变化，这是否算作一个不同的字母？希伯来文说是，阿拉伯文说否。无论如何，Unicode 工作组的专家研究这些问题已经约有十年了，很多争议带有政治色彩。你不用知道这么多，他们已经把一切都理清了。

Unicode 工作组给所有字母表中所有字母的理型都指定了一个魔法数字，写出来是这样：U+0645。这个魔法数字就是一个码点。前缀 U+ 的意思是“Unicode”，数字是十六进制的。U+0639 是阿拉伯字母 Ain。英文字母 A 是 U+0041。你可以用 Windows 2000/XP 提供的字符映射表 (charmap) 工具查看所有映射，或者访问 Unicode 官方网站^①。

Unicode 能定义的字符并没有真正的上限，事实上 Unicode 的码点数量远超过了 65 536，所以和大家的错觉不一样，不是所有 Unicode 字母都能塞进 2B 里。

好，假定我们有如下字符串：

```
Hello
```

^① 参阅<http://www.unicode.org/>。

在 Unicode 中，对应如下 5 个码点：

U+0048 U+0065 U+006C U+006C U+006F

这只是一些码点而已，更准确地说是数字。我们还没有说明如何在内存中存储或在电子邮件消息中表示它们。

编码 (Encoding)

表示码点的是所谓的编码。

最早人们对 Unicode 编码的实现是每个数字都用 2B 存储，这导致了“所有 Unicode 字符都是 2B”的错误印象。于是 Hello 变成：

00 48 00 65 00 6C 00 6C 00 6F

没错吧？等一下，能不能写成这样：

48 00 65 00 6C 00 6C 00 6F 00 ?

理论上也行得通，而实际上，人们在一开始就用高位补零和低位补零两种模式存储 Unicode 码点，根据机器的 CPU 来选用处理速度较快的存储方式。好家伙，一夜之间已经冒出两种存储 Unicode 的方法了。为了使两者有所区别，人们被迫约定，在每个 Unicode 字符串前面插入字符 FE FF。这叫做 Unicode 字节顺序标记 (Unicode Byte Order Mark)。如果高低位互换，标记变成了 FF FE，别人在读字符串的时候就知道需要把其他字节进行两两互换^①。真麻烦。而且实际使用中的 Unicode 字符串并不一定都带有开头的字节顺序标记。

这样安稳了没多久，程序员们又不满意了。他们抱怨：“看看，有这么多的 0！”身为美国人，他们平常接触的只有英文，很少使用 U+00FF 以上的码点。而且他们是一群住在加利福尼亚州的自由派嬉皮士，最喜欢把“节俭”挂在嘴边（其实是小气）。如果是从得克萨斯州来的人，才不会对多出一倍的字节数斤斤计较呢。但是让这些加利福尼亚铁公鸡用多一倍的空间存储字符串，简直就像割他们的肉一样。况且已经有那么多用各种 ANSI 和 DBCS 字符集存储的文档，谁来转换所有文件的编码？反正我不干。这些原因促使绝大部

^① 关于字节顺序标记的更多信息，参阅 http://msdn.microsoft.com/library/default.asp?url=/library/en-us/intl/unicode_42jv.asp。

分人在很长一段时间里对 Unicode 视若无睹，情况变得越来越糟。

这时，有人想出了一个绝妙的主意，那就是 UTF-8^①。UTF-8 是一套新的编码，即用 8 位的字节把由 Unicode 码点，也就是那些神奇的“U+数字”组成的字符串，存储在内存中的方法^②。在 UTF-8 中，从 0 到 127 的码点存储在 1B 中。只有 128 及以上的码点才用 2B、3B 乃至 6B 来存储。

Hex Min	Hex Max	Byte Sequence in Binary
00000000	0000007F	0vvvvvvv
00000080	000007FF	110vvvvv 10vvvvvv
00000800	0000FFFF	1110vvvv 10vvvvvv 10vvvvvv
00010000	001FFFFF	11110vvv 10vvvvvv 10vvvvvv 10vvvvvv
00200000	03FFFFFF	111110vv 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv

这么做还有一个很方便的作用，就是英语文本在 UTF-8 中的表示和在 ASCII 中的表示完全一致，对此美国人民表示毫不知情。而剩下的地球人就没那么轻松了。举例来讲，字符串 hello 的 Unicode 表示为 U+0048 U+0065 U+006C U+006C U+006F，那么它用 UTF-8 编码的存储形式为 48 65 6C 6C 6F，（注意！）这和它在 ASCII、ANSI 以及其他所有 OEM 字符集中的表示都完全一致。在使用英语方言字母，或是希腊语字母，甚至是克林贡语字母的时候，都需要几字节来存储一个码点，而美国人却不用理会这些麻烦。（UTF-8 还有一个好处，某些旧版的代码会把值为“0”的字节当做字符串的 null 结束符处理，而如果用 UTF-8 编码，就不会错误地截断字符串。）

到目前为止，我一共讲了三种 Unicode 的编码方法。比较传统的方法是用 2B 存储每一个码点，叫做 UCS-2（2 代表有 2B），或者叫作 UTF-16（16 代表有 16 比特），但你还是需要区分高位补零的 UCS-2 和低位补零的 UCS-2。还有一种方法是较新的、广受欢迎的 UTF-8 标准^③，即使你碰巧在使用白痴到除了 ASCII 之外什么都不认识的程序来处理英语文本，UTF-8 照样能工作得很好。

实际上，Unicode 还有很多其他的编码方法。有一种类似 UTF-8 的编码方法 UTF-7，它的最高位始终是 0。某些警方或国家机关使用的电子邮件系统只接受用 7 比特表示一个字符，UTF-7 能保证 Unicode 信息无损地通过这些系统。还有 UCS-4，它用 4B 来存储每个码点，好处是每个码点对应相同的字节数，但是，就连得克萨斯人也没有大方到敢浪费这么多内存。

① 关于 UTF-8 的更多信息，参阅<http://www.utf-8.com/>。

② 参阅<http://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt>。

③ 参阅<http://zh.wikipedia.org/wiki/UTF-8>。

如果你从柏拉图理型的观点出发来思考,就会发现其实可以用任何老式的字符集来编码 Unicode 码点!例如,你可以把 Hello 的 Unicode 串(U+0048 U+0065 U+006C U+006F)表示为 ASCII、老式的 OEM 希腊编码、希伯来 ANSI 编码,或是其他几百种编码中的任何一种,只是有一个问题:某些字母会显示不出来!如果你想表示的 Unicode 码点在某编码中没有对应的字符,你通常会得到一个小问号,或者如果你走运的话,会得到一个方框。

数百种传统编码只能对部分的 Unicode 码点进行正确的转换,对其他码点则显示为问号。比较流行的英文编码包括 Windows-1252 (Windows 9x 系列对西欧语言的编码标准)和 ISO-8859-1,后者又名 Latin-1 (也支持所有西欧语言)^①。但是如果用这些编码来存储俄文或希伯来文,你就会得到一堆问号。UTF-7、8、16 和 32 都可以正确地储存任意的码点。

关于编码最最重要的事实

即使你忘了我前面讲的每一句话,也请记住以下这个重要的事实:不能确定编码的字符串没有任何意义。你再也不能像把脑袋扎进沙堆里的鸵鸟一样,假装“纯文本”就是 ASCII。

所谓的纯文本,根本就不存在。

如果你在内存、文件或电子邮件消息里看到一个字符串,在确定它的编码之前,不可能去解析它,也不可能把它正确地显示给用户。

诸如“我的网站看起来像是胡言乱语”或“她无法读取我用方言字母写的电子邮件”之类的愚蠢问题,背后的罪魁祸首总是某个头脑简单的程序员,他不知道如果不确定某个字符串是用 UTF-8、ASCII、ISO 8859-1 (Latin 1) 还是 Windows 1252 (Western European) 编码的,就不能正确地显示它,甚至不能确定它在什么地方结束。编号 127 以上的码点都有超过一百种可能的编码方式,乱猜是没用的。

如何储存字符串所用编码类型的信息?有一个标准的做法。对于所有电子邮件消息,你需要在表头有这样一个字符串:

^① 关于 ISO-8859-1 字符集的概述,参阅<http://www.htmlhelp.com/reference/charset/>。

Content-Type: text/plain; charset="UTF-8"

对网页而言，以前 HTTP 协议的设计者想和电子邮件保持一致，要由服务器随网页本身返回一个 Content-Type 的 HTTP 报头。也就是说编码信息不存在 HTML 文件内部，而是作为响应消息报头的一部分，先于 HTML 页面发送。

这会导致一些问题。假设你的网络服务器很大，上面托管了很多网站，众多网友用不同的语言制作并上传了大量的页面，所用的文件编码都是微软 FrontPage 软件根据操作系统的环境变量自动生成的。服务器本身并不知道每个文件具体是用什么编码写的，所以它无法在报头中发送 Content-Type。

如果能把 HTML 文件的 Content-Type 放在本文件内部的某个特殊标签里，那就好办了。较真儿的人一听就疯了，你怎么可以在不知道文件编码的情况下就读取 HTML 文件呢？！幸好几乎所有常用的编码对编号从 32 到 127 的区间内的处理都是相同的，因此在不使用任何奇怪字母的前提下，总可以在文件开头写上这些内容：

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

meta 标签必须是<head>内部的第一个节点，因为浏览器一看到这个标签，就会停止解析页面，转而用你指定的编码重新解读并加载整个页面。

如果浏览器在 HTTP 报头和 meta 标签中都找不到 Content-Type 怎么办？Internet Explorer 很有意思，它会根据各字节在不同编码和语言组合中出现的频次，猜测页面是哪种语言、用了什么编码。很多老式的 8 比特代码页都会在编号为 128~255 的位置存储某种语言特有的字母，由于每一种人类语言都对应一个特定的字母使用频次直方图，因此这种方法实际上是可行的。听上去很不可思议，但实际使用这种方法猜中语言和编码的概率很高，导致有的门外汉即使不知道需要在报头中加入 Content-Type，也能写出看上去正常的网页。直到某一天，当他们写的内容的字母频次分布与所用的语言不太一致时，Internet Explorer 会突然将页面误识别为韩语。我觉得，这表明所谓的波斯特定律 (Postel's Law)，即“对输出要严谨，对输入要宽容”，坦白地讲，并不是一个很好的工程原则^①。那么，当保加利亚语的页面被浏览器误判为韩语显示（而且是语法错乱的韩语），用户该怎么办？他需要打开菜单，在“查看|编码”下尝试各种编码（至少有十几种东欧语言），直到页面看起来像

^① 出自 Jon Postel。引自美国资讯科学研究院 (Information Science Institute) 的“RFC791—Internet Protocol”，1981 年 9 月。

话为止。而大多数用户都不知道这个手动指定页面编码的方法。

我所在的 Fog Creek 软件公司开发了一个名为 CityDesk^①的网站管理软件, 在它的最新版本中, 我们决定所有内部字符串都用 UCS-2 (2 字节码) 编码, 因为这是 Visual Basic、COM 组件和 Windows NT/2000/XP 使用的原生字符串类型。在 C++ 代码中, 我们使用 `wchar_t` (“宽字符”) 而不是 `char` 声明字符串, 并且使用 `wcs` 函数族而不是 `str` 函数族 (例如用 `wcscat` 和 `wcslen`, 而不用 `strcat` 和 `strlen`)。在 C 代码中表示 UCS-2 字符串, 只需在前面加一个 L, 像这样:

```
L"Hello"
```

CityDesk 发布网页的时候会自动转为 UTF-8 编码, 各种浏览器对该编码的支持早就很完善了。Joel On Software 所有 29 种语言的版本都是用这种方法编码的, 我还没有听到任何人反馈页面的语言显示有问题^②。

这篇文章篇幅有限, 无法涵盖所有关于字符编码和 Unicode 的内容。读完本文, 我希望你们下次写代码的时候使用抗生素, 别再碰水蛭和咒语了。这就是我给你们的任务。

① 参阅 <http://www.fogcreek.com/CityDesk>。

② 参阅 <http://www.joelonsoftware.com/navLinks/OtherLanguages.html>。

05

轻松撰写功能规格书 第一部分：为什么要写？

2000 年 10 月 2 日，星期一

在乔尔测试^①面世之初，读者反映，其中最难做到的部分就是写规格书。写软件规格书就好像使用牙线：大家都觉得是件好事，但就是没人做。

为什么人们不写规格书？有人说跳过写规格书的阶段可以节省时间。在这些人眼中，写规格书是一种奢侈，只有那些为 NASA 制造太空飞船的工程师或是大型知名保险公司的工作人员才配享有。这真是胡说八道。首先，不写规格书是软件项目开发中最大的不必要风险，其愚蠢程度就像不带装备就横穿广阔的莫哈韦沙漠，心存侥幸能逃过一劫。那些不写规格书就一头扎进代码中的程序员和软件工程师，以为自己是要酷的神枪手，不用瞄准就能拔枪射击。他们是在做梦。他们的效率非常低，写的代码烂，做出的软件更烂，并且为项目平添了非常大的、完全不必要的风险。

我认为，对于任何非玩票性质的项目（开发时间超过一周或人员多于一名），不写规格书会导致项目花费的时间更长，代码的质量更差。原因如下。

规格书最重要的作用就是对程序功能进行设计。即使只有你自己在写代码，规格书也只写给自己看，在写规格书的过程中，通过详细描述程序的工作原理，你会被迫对程序做真正的设计。

让我们虚构来自两家公司的两个程序员。来自急速香蕉软件公司的史比迪小姐，从不写规格书。“规格书？我们才不需要那玩意！”来自好脾气软件公司的罗杰斯先生，在写代码之前总会完成详细的规格书。我虚构了好几位朋友，

① 见第 3 章，<http://www.joelonsoftware.com/articles/fog0000000043.html>。

这是其中两位。

史比迪和罗杰斯有一个共同点：他们都负责维护各自产品 2.0 版本的历史兼容性。

史比迪决定，提供历史兼容性的最好方法是写一个简单的转换器，把 1.0 版本文件转换为 2.0 版本。她立刻开始敲代码。敲、敲、敲，点、点、点，硬盘飞转，尘土飞扬。两周后，史比迪的转换器完工了，但她的客户不满意，因为她的代码强迫公司的所有人都必须立即升级到新版本。史比迪最大的客户 Nanner Splits 无限公司拒绝购买新的软件。Nanner Splits 想让新软件不作转换就能处理 1.0 版本的文件。史比迪决定写一个降级转换器，并把它绑到“保存”功能上。这下乱套了，因为虽然看上去你可以使用 2.0 版本的功能，但在要保存为 1.0 版本的格式时，软件会提示你半个小时前用过的功能在旧文件格式上不可用。写降级转换器不仅又花了 2 周时间，而且效果不好。这样总共花费了 4 周时间。

再来看好脾气软件公司的罗杰斯先生，他古怪偏执的一面表现在没有规格书就拒绝写代码。他花了 20 分钟来做历史兼容功能的设计，想法和史皮迪差不多，规格书大致的意思是：

打开一个用旧版本产品创建的文件，自动将其转换成新版本的格式。

这个规格书拿给客户一看，客户急了：“等一下！我们不想强迫每个人立刻升级！”于是罗杰斯先生又想了想，修改了规格书，如下：

打开一个用旧版本产品创建的文件，会在内存中自动将其转换成新版本的格式。保存文件的时候，用户可以选择把文件转换回旧版本。

又过了 20 分钟。

罗杰斯先生的上司是一个面向对象编程的狂人，他看到规格书，觉得缺点什么。于是他给出了一个不同的架构。

重构代码，分出两个接口：V1 和 V2。V1 包含所有 1.0 版本的功能；V2 继承 V1，增加 2.0 版本的所有新功能。V1::Save 负责处理历史兼容性，V2::Save 用新格式来保存文件。如果你打开 V1 文件时想使用 V2 的功能，软件会发出提醒，用户要么转换文件，要么放弃使用新功能。

20 分钟过去。

罗杰斯先生郁闷了。重构将耗时 3 个星期，而不是他预计的 2 个星期！但是这么做确实优雅地解决了用户的所有问题，所以他接受了方案，开始编程。

罗杰斯先生总共用了 3 周零 1 小时。史皮迪总共用了 4 周，而且代码的质量不如前者。

这个故事告诉我们，瞎编一个例子可以证明任何事情。呃，我不是这个意思。故事告诉我们，当你用人类的语言设计产品时，可以在几分钟的时间里权衡几种方案的利弊，然后修正、改进你的设计。谁也不会为了在文字处理器里删除一段话而难过。但如果你选择在编程的语言里做产品的设计，可能要花数周的时间才能做出可以让人反馈的设计。更糟的是，程序员对于耗时两周时间写出的代码是有感情的，不管这些代码错得多离谱。无论史皮迪的上司或客户怎么说，她都不甘心丢掉她辛苦写出的、漂亮的转换器代码，尽管这段代码在设计上并不漂亮。结果是，最后交付的产品形态介于最初错误的设计以及理想的设计之间。只能评价为“这是我们能做的最好设计，毕竟我们辛辛苦苦写了那么多代码，总不能全都推倒重来吧。”而最好的评价应该是“这是我们能做的最好的设计，没有之一。”

以上是写规格书的第一大理由。第二大理由是节省沟通的时间。通过写规格书，你只需要和别人沟通一次。团队的其他成员只要去读规格书，就能知道程序的方方面面。测试人员通过规格书，可以了解程序有哪些符合预期的功能和行为，也就是该从哪些角度去测试。营销人员以规格书为蓝本，撰写出充满溢美之词的产品白皮书，然后放到网站上去宣传你们还未出世的“杰作”。业务拓展人员经常误读规格书，用天马行空般的想象力把产品塑造成秃顶或是皮肤疣的克星，不过只要他们能忽悠到投资，你就没什么可抱怨的。开发者读了规格书，就知道该写哪些代码。客户读了规格书，就能确信开发者们正在做的产品是物有所值的。技术文档编写者以规格书为参考，为客户写出图文并茂的说明书（就是那些经常遗失或被丢弃的小册子，不过那就另说了）^①。经理们读了规格书，就可以在管理会议上表现得一切尽在他们掌控之中。诸如此类。

如果你没写规格书，所有这些沟通的过程还是要发生，因为这是必需的，不过这种情况下，它们会时不时地发生。测试人员拿到软件时并不知道怎么样

^① 关于为什么没有人阅读说明书，参阅 Joel Spolsky 的《面向程序员的用户界面设计》（Apress, 2001）相关章节，<http://www.joelonsoftware.com/uibook/chapters/fog0000000062.html>。

才算正常工作，只能随便点点看看，一旦发现有奇怪的地方，他们就会一次次打扰程序员，东一句西一句地追问各种愚蠢的问题。这不仅会导致程序员的工作效率被破坏殆尽^①，而且程序员在回答这些问题时会倾向于用代码实现细节来解释，而给不出所谓“恰当的答案”。导致的后果是，测试人员是基于程序来测试程序，而不是基于设计来测试程序，而后者可能，嗯，还稍微更加有用一点。

如果不写规格书，发生在可怜的技术文档编写者身上的事才叫滑稽（带有一丝悲凉的感觉）。他们通常比较弱势，不敢打扰程序员大人们。许多公司的情况是，如果技术文档编写者养成了向程序员提问的习惯，经常询问某个软件怎么样才算正常工作，程序员就会向经理告状，说那些可恶的文档编写者老是麻烦我，让我什么都干不成，能不能让他们离我远一点。经理们为了提高工作效率，只能禁止文档编写者浪费程序员们的宝贵时间。至于这种情况发生在哪些公司，你从他们的帮助文件和产品手册上就看得出来，里面包含的信息比你运行软件时自己猜到的信息还少。当你看到屏幕上出现如下提示信息：

你想开启 LRF-1914 支持功能吗？

然后你点“帮助”，弹出的帮助主题会让你哭笑不得：

让你在使用 LRF-1914 支持功能（默认）和不使用 LRF-1914 支持功能之间切换。如果你想使用 LRF-1914 支持功能，选择“是”或按“Y”键。如果你不想使用 LRF-1914 支持功能，选择“否”或按“N”键。

那个，谢谢了。很明显，编写者在努力掩盖自己根本不知道 LRF-1914 支持功能是什么的事实。他们不能问程序员，可能因为(a)他们不好意思开口，或者(b)程序员远在印度海得拉巴，而他们在英国伦敦，或者(c)管理层禁止他们打扰程序员，或是因为其他难以穷举的大公司通病。但是最根本的问题在于，没有规格书可以参考。

写规格书的第三大理由是，没有详细的说明，就无法制定进度表。如果你在攻读博士学位而且做好了为之奋斗 14 年的准备，可以没有进度表。或者如

^① 参阅<http://www.joelonsoftware.com/articles/fog0000000068.html>。

果你写的程序有望成为下一个《毁灭公爵》，也可以理直气壮地说“我们什么时候把程序做扎实做漂亮了，就什么时候交付”。但是对于几乎所有面向实际业务的开发工作，你必须要知道要做某件事需要多长的时间，因为开发产品是要花钱的。买牛仔裤之前还要看价格呢，何况是负责任的投资者，他们在决定是否开发某个产品时，当然需要先知道开发的时间成本，进而换算成经济成本。关于如何制定进度表，请阅读第9章。

公司常犯的一个重大错误是，当大家对某些设计有争议时，争议从来都得不到解决。Windows 2000 的开发主管布莱恩·瓦伦丁有一句座右铭：“一定要在10分钟内达成决议。”^①

太多的软件开发机构都有这个问题：一旦出现设计上的争议，没有人站出来力挽狂澜，促成一个决议，这通常是出于某种办公室政治原因。这导致程序员们只能就无关痛痒的设计进行开发。随着时间的推移，所有困难的决定都被推迟到了最后一刻。这样的项目是最有可能失败的项目。如果你开了一家研发某项新技术的初创公司，并且发现公司在体制上就无法作出决定，那你还不如趁早关门，把钱还给投资者，因为这样下去公司什么也做不成。

在写规格说明书的过程中，所有那些令人不悦的大小设计决定都会统统暴露出来。如果不写，它们很容易就被掩盖过去了。即使是非常琐碎的决定，也能从规格书中看出其问题的端倪。比如你们在做一个会员制的网站，可能大家都同意，如果会员忘了密码，就通过邮件告知他们。很好。但是这还不足以指导开发。要写出代码，你需要知道那封电子邮件的确切措词。大部分公司都不信任程序员的文字水平，不让他们直接撰写真正要展示给客户看的文案（而这种不信任通常是有道理的）。所以消息的具体措词需要交给某个营销、公关或是文学专业的人去拟定。“亲爱的笨蛋，这是你忘记的密码。下次别这么大意了。”当你强迫自己把规格书写得规范、完整的时候（我马上就会讲这个），就会注意到很多细节的问题，你要么立即解决，要么就在旁边做一个大大的红色记号。

好，现在大家意见一致了吧。就好像第二次世界大战时美国大兵口中的“母亲和苹果派”，我们要为规格书奋战到底。我猜大部分人都明白这一点，而我发的这么多牢骚尽管有趣，却并没有告诉大家什么新东西。问题是为什么

^① 微软新闻稿“Valentine and His Team of 4,200 Complete the Largest Software Project in History”，Microsoft.com，2000年2月16日。参阅<http://www.microsoft.com/presspass/features/2000/02-16brianv.asp>。

人们明知道规格书有用，但还是不写呢？肯定不是为了节省时间，因为恰恰相反，不写规格书并不省时，我觉得大多数程序员都对此深有体会。（在大部分机构里，唯一能算作“规格书”的东西是那种生硬的、篇幅只有一页的文本文档，而且还是某个程序员在写完代码，并向 300 个人解释过那些该死的功能之后，愤然一挥而就的。）

我认为问题在于太多的人不喜欢写作了。面对一个空白的屏幕，很多人都会发怵。我之所以能克服这种恐惧，得益于我在大学里选过一门课，课程要求每周交一篇 3 至 5 页的短文。写作能力是一种实力。你写得越多，就越会写。如果你需要写规格书但写不出来，可以先开始记日记、写博客、参加一个创作培训班，或者干脆给认识的人写信，不管是亲戚也好，你 4 年多没联系的大学室友也罢。不要放过任何一个把词写在纸上的机会，持续改进你的规格书写作技巧。如果你是软件开发活动的管理者，要是你手下的人不愿意写规格书，就把他们送去参加一个为期两周的创意写作培训班，最好是在深山里办的那种。

如果你没有在一个要求写功能规格书的公司工作过，你可能不知道这种文档是什么样的。在下一章里，我会示范一个简短的规格书供你参考，然后再看看一份好的规格书应该包括什么。

06

轻松撰写功能规格书

第二部分：什么是规格书？

2000年10月3日，星期二

本系列文章讨论的是功能规格书，而不是技术规格书。二者经常被混淆。我不知道是否存在更权威的界定，但我对这两个术语的理解如下。

- (1) 功能规格书从使用者的角度，完整描述了产品的行为。其中并不涉及具体的实现方式。它只介绍功能，描述界面、菜单和对话框等。
- (2) 技术规格书描述的是程序内部的实现细节。它涉及数据结构、关系型数据库模型、使用哪些程序语言、工具以及算法，等等。

从头开始设计一个产品时，最重要的是完成对用户体验的设计：有哪些页面，页面如何使用，具有哪些功能。然后，设计者要关注如何从一个页面到另一个页面，也就是设计工作和数据的流程。在确定产品将要做什么之前，没有必要去想应该用什么编程语言。在这个系列中，我只会着眼于功能规格书。

下面是一个简短的例子，来示范一份好的功能规格书该是什么样子的。

概 述

WhatTimeIsIt.com 是一个告诉人们当前时间的网络服务。

这份规格书无论怎么看，都不够完整。所有的措辞都可能需要经过数次修改，才能最终定稿。这里展示的产品图形界面和布局都只是为了表述内在的功能，产品的最终观感将通过长期的开发形成，这需要图像设计师的投入并且经过多次用户反馈才行。

本规格书并不涉及时间计算引擎所使用的算法，这部分内容将在其他文档中进行阐述。本文只讨论用户在与 WhatTimeIsIt.com 交互时能够看到什么。

使用场景

在产品设计阶段，构想真实生活中的若干（具有代表性的）人物会如何使用该产品，是十分有益的。我们现在就来看两个使用场景。

场景1：迈克

迈克是一名忙碌的企业家。他是一家大型重点企业的总裁，该企业面向儿童市场，专做和炸药相关的产品，产品行销全国。通常，迈克每天都参加很多重要会议，和很多大人物见面。有时候银行会派人过来，用逾期 3 月没有支付信用卡利息的小事来叨扰他。有时候另一个银行的人会来向他推销另一套信用卡服务。有时候风险投资人（他们心肠很好，给了迈克很多钱来开公司）会拜访他，抱怨他赚钱太多。“我要的是篝火！”他们会说，“华尔街愿意看到的是篝火！”

如果这些访客事先经过了预约，到时候却见不到迈克，就会很生气。这种情况经常发生，因为迈克不知道现在几点了。在秘书的推荐下，他注册了 WhatTimeIsIt.com 的账号。这样，迈克想知道时间的时候，只需通过用户名和密码，登录到 WhatTimeIsIt.com，就能知道当前的时间。他每天都要访问网站好几次：看看是否到了午餐时间，是否错过了下个会议的时间，等等。在每天快下班的时候，也就是下午 3 点钟之后，他会越来越频繁地登录网站，看看是否到了回家的时间。到了下午 4:45 分，他的鼠标几乎没有离开过浏览器的刷新按钮。

场景2：辛迪

辛迪是一名高中生。她所在的高中是一家很普通的公立学校，而她又很聪明，所以当她下午 2 点钟回到家，大概平均只要花 7 分钟就能写完代数作业。其他老师甚至都允许她不写作业。她的弟弟（同父异母）还是个婴儿，在家里唯一的电视机前一动不动地看动画片《天线宝宝》，所以辛迪整个下午（从 2:07 直到 6:30 左右，那时继母会喊大家吃晚饭）都在网上冲浪，和“美国在线”上的朋

友们聊天。她总在寻找新奇的网站。有一天她不小心在搜索引擎的输入框中输入了“现在几点了？（What Time Is It？）”（本来这句话是要通过即时聊天工具发给朋友的），结果跳转到了WhatTimeIsIt.com，于是注册了一个账户。她选了一个用户名，然后把密码设为“RyanPhillipe”^①，设定好时区，就能查看当前的时间了。

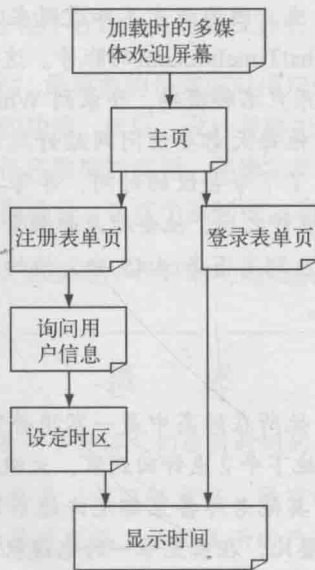
非目标

本次开发的版本不支持如下功能：

- ☐ 设置多个时区（假定所有用户都处于同一时区）；
- ☐ 修改密码；
- ☐ 预约；

WhatTimeIsIt.com的流程图

我们之后有的是时间探讨具体的细节，但是现在，让我们来看一个简单的流程图，以了解整个服务的全貌。这个流程图并不完整，但它展示了使用WhatTimeIsIt.com的“故事板”：



^① 雷恩·菲利普，男子名，你懂的。——译者注

每个页面的功能规格说明

WhatTimeIsIt.com 包含好几个不同的页面。大部分页面都遵循统一的形式，具有相同的观感，这会由图像设计师给出具体的设计方案。本文档更集中于讨论页面的功能和交互设计，而不是具体的页面样式或布局。

所有页面都是用 HTML 写的。（唯一的例外是加载时的多媒体欢迎屏幕，它是用 Macromedia Shockwave 写的。）

WhatTimeIsIt.com 的每个页面都有一个标准化的名称，在本文中以黑体小节标题显示，所以您可以通过标题来知道我们在说哪个页面，例如**主页**。

加载时的多媒体欢迎屏幕

一个让人厌烦、毫无必要的 Shockwave 动画配上愚蠢的背景音乐，总会让人抓狂。加载时的多媒体欢迎屏幕由我们聘请的专业图像动画工作室精心打造。这些艺术家们住在苏活区，每天带着狗上班，喜欢把各种奇怪的物品别在耳朵上，每天午饭前要去 4 次星巴克。

动画播放 10 秒之后，一个“跳过动画”的链接会从右下角淡入。为了避免有太多的人看到并点击它，“跳过动画”必须位于非常偏僻、很多人都看不见的角落。至少要距离页面左边 800 像素、距离页面顶部 600 像素。

点击“跳过动画”会直接进入**主页**。当动画结束时，浏览器会自动跳转到**主页**。

待解决的问题 如果市场部允许，我们应该在用户电脑上存一个 cookie：如果用户点击过一次“跳过动画”，那么将来每次都会跳过动画。频繁访问网站的用户应该只能见到一次欢迎动画。我和市场部的吉姆谈了一下，他打算召集销售、市场和公共关系部门的人一起开会讨论这件事。

主页

Shockwave 动画结束时显示主页，它主要包括三个功能：

- (1) 让用户了解服务内容，促使他们注册服务；
- (2) 让注册会员登录；
- (3) 引导有注册意愿的用户注册账号。

主页如下图所示：



在本页以及其他所有页面上，点击左上角 WhatTimeIsIt.com 的 Logo 会回到主页。

技术注解 因为很多页面之间的高度相似性，所以服务器需要引入某种 `include` 机制，这样当服务名称改变，或是我们买不到想要的域名时，可以在一个地方修改所有的页面。我建议使用 Vignette Story Server。虽然它售价 20 万美元，解决这个问题有点大材小用，但是这比服务端写 `include` 要容易多了！

点击文案为“点此登录”（click here to log on）的链接，进入登录表单。点击文案为“点此注册”（click here to sign up）的链接，进入注册表单。其他 5 个链接指向由管理员配置的静态页面，这些页面不在本文叙述之列。它们的变动不会很频繁。

登录表单

已经注册的会员通过登录表单登录账号，以查看当前时间，如下所示：

WhatTimeIsIt.com
Please enter your email address:

Enter your password:

Forgot your password? Just enter your email address and we'll email it to you.
Not a member yet? Don't worry - membership is *free*! Yes, that's right, *FREE*! Just [click here](#) to sign up, and within *minutes* you'll be able to find out what time it is!
[Privacy Notice](#) | [About Us](#) | [Jobs](#) | [Contact Us](#)
[About WhatTimeIsIt.com](#)

页面右侧和之前介绍的主页一样。

电子邮件地址输入框中最多填入 60 个字符。密码输入框中最多输入 12 个字符。为了阻止黑客行为，用户输入的字符会显示为星号*。

技术注解 使用<INPUT TYPE=PASSWORD>即可。

当用户点击“登录”（Log In）时，服务器会作如下验证。

- (1) 如果用户在电子邮件地址处输入的字符不符合一般的格式（比如没有@或是包含 RFC-822 中不允许在电子邮件地址中出现的字符），服务器会返回另一个和登录表单相同的页面，只不过这个页面在电子邮件地址输入框的上方多了一行红色的错误信息，内容为：“您输入的电子邮件地址无效，请检查。”虽然这行文字是红色的，但是“请输入电子邮件地址”这行仍是黑色的。之前用户输入的错误电子邮件地址将被预加载到编辑框中。
- (2) 如果用户提供了电子邮件地址，但是找不到对应的注册会员，那么服务器返回另一个与登录表单相同的页面，只不过这个页面在电子邮件地址输入框的上方多了一行红色的错误信息，内容为：“您输入的电子邮件地址不是我们的会员，请检查。如果想注册会员，请点击页面右侧的链接。”虽然这行文字是红色的，但是“请输入电子邮件地址”这行仍是黑色的。之前用户输入的错误电子邮件地址将被预加载到编辑框中。（我想问一下开发者。

这种情况下如果用户点击了“注册”，能用 Javascript 把之前填过的电子邮件地址预加载到注册表单的对应域吗？）

- (3) 如果用户提供了电子邮件地址，并且有对应的注册会员，但是没有输入密码，我们就发送一封邮件到电子邮件地址，告知用户密码。电子邮件的主题是：您的 WhatTimeIsIt.com 账号信息。电子邮件的格式是纯文本。至于具体内容，董事会仍然在激烈讨论中，将在软件交付前给出。（开发者们：我建议暂时使用一句脏话来作为邮件内容。这会敦促用户尽快给出合适的内容。）
- (4) 如果用户提供了电子邮件地址，有对应的注册会员，而且输入了密码，但是密码不正确，服务器会返回另一个和登录表单相同的页面，只不过这个页面在电子邮件地址输入框的上方多了一行红色的错误信息，内容为：“您输入的密码无效，请检查。注意密码区分大小写。”如果密码中全部都是大写字符，我们会增加一句提示：“也许您无意中激活了大写锁定键？”密码错误时，登录表单会清空密码框的内容。
- (5) 如果电子邮件地址和密码都正确，直接跳转到显示时间。

待解决的问题 需要解决 2 号用例中的 Javascript 问题。

待解决的问题 需要从 CEO 那里得到发送给用户的密码邮件的具体措辞。

以下是我在每篇规格书中必写的内容。

免责声明 完全出于自我保护的目的。如果你写上一段文字，说几句“这篇规格书并不完整”之类的话，就不用担心有人冲进办公室来冲你大发雷霆了。随着时间推移，等到规格书的内容基本完整了，你可以把这段文字改成“这篇规格书是完整版，但因作者学识有限，如有疏漏，请不吝赐教”。这倒提醒了我，每篇规格书还应该包括以下方面。

作者 一位作者。有的公司认为规格书应该由团队共同撰写。如果你尝试过团队写作，你就知道那是一种折磨。还是把团队写作留给管理咨询公司吧，这些公司供养着一支由哈佛毕业新生组成的大军，他们急需大量工作证明他们对得起自己的高额薪水。你的规格书应该属于一个人所有并由那个人来撰写。如果产品较大，那么把它分成几块，每一块分别交给一个人来写规格书。有些公司认为，只让一个人在规格书上署名并“把功劳据为己有”是一

种自大的、不符合“团队精神”的做法。一派胡言。人们本来就该为自己写的规格书负责，对其有主人翁意识。如果规格书有问题，那么应该有一个确定的责任人，就是在规格书上署名的那个人，他（她）有责任更正其中的问题。

使用场景 设计一个产品时，你需要对真实的使用场景，也就是人们怎样使用这个产品，做到心中有数。否则到头来你会设计出一个和真实世界的需求格格不入的产品（如 CueCat^①）。从你产品的使用者中，选取几类代表性的目标用户群，为每一类虚构一个想象中的、但完全典型的用户。我的用户界面设计手册中的第 9 章^②讨论了如何虚构用户和场景。现在，这就是它们的用武之地。场景越是生动、逼真，你设计出的产品就越适合用户使用，无论他们是虚构的还是真实的，因此我经常编写很多冗余的细节。

非目标 当你带领团队做产品时，似乎每个人都有一些自己偏爱的功能特性，认为非实现不可。如果要实现全部这些功能，将花费无数的时间和金钱。项目管理者必须从一开始就修剪功能特性，最好的方法就是在规格书中加入一节“非目标”，即我们不做什么。非目标可能是你不想实现的功能（“不要心电感应的用户界面！”），也可能是更宽泛的目标（“我们在这一版里不关心性能。产品只要能行就行，慢点没关系。如果有时间做下一个版本，那时再优化性能。”）。这些非目标很可能会引起争议，但是尽快将其公开非常重要。就像老乔治（George Sr.）说的：“就是不做！”^③

概述 这就像规格书的目录。可以是一个简单的流程图，也可以是对基本架构的详细文字描述。人们在了解产品的大致轮廓之后，会更容易理解其中的细节。

细节、细节、细节 最后要详细阐述每个细节。大部分人会略读这部分内容，需要了解某个特定细节时才会查找并细读相关的部分。在设计网页类型的服务时有一个好办法，就是给每一个页面起一个规范的名字，然后分别用一章的篇幅来详尽描述每个页面。

细节是功能规格书中最重要的内容。在规格书示例中，你会注意到我极尽详细地描述了登录页面所有可能出现的错误情形，例如电子邮件地址不符合规

① 参阅<http://www.joelonsoftware.com/articles/fog0000000037.html>。

② Joel Spolsky,《面向程序员的用户界面设计》，参阅<http://www.joelonsoftware.com/uibook/chapters/fog0000000065.html>。

③ 老乔治即老乔治·布鲁斯（George Bluth, Sr.），是情景喜剧《发展受阻》（*Arrested Development*）中的人物。——译者注

范怎么办？密码错误怎么办？所有这些情形都对应到将要实现的具体代码，但更重要的是，这些情形对应着某个人将要作出的设计决定。总要有人来决定当用户忘记密码时网站要如何应对。如果不做决定，就没法写代码。规格书的作用就是把设计决定用文档表述出来。

待解决的问题 第一版的规格书可以留下一些待解决的问题。在写第一版草稿时，我总会写许多待解决的问题，并做出标记（便于之后查找），如果可以的话，还会讨论一下可能的解决方案。所有这些问题必须在程序员们开工之前得到解决。（你可能觉得可以先让程序员做简单的部分，之后再慢慢思考并解决剩下的难题。这不是好主意。程序员在实现代码的过程中会冒出许多新问题，到时候根本没有时间去解决设计阶段遗留的问题。你应该尽早解决它们。另外，决定如何处理某个非细节性问题，可能会对代码的实现方式存在巨大的影响。）

多角度的注解 在写规格书时，要记得阅读的人群有好多种：程序员、测试人员、营销人员、技术文档编写者，等等。在写作过程中的某一段落，你可能只是写给这些人中的某一个群体。例如，我会把写给程序员的部分标记为“技术注解”，内容通常是技术实现细节。营销人员会跳过这部分内容。程序员则会对这部分逐字逐句地照做。我的规格书中经常充斥着各种“测试注解”、“营销注解”以及“技术文档注解”。

规格书需要与时俱进 有些开发团队践行“瀑布式”思维：技术负责人一次性完成所有设计，然后写好规格书，打印出来，把它扔到墙那边的程序员办公室中，然后下班回家。我对此的评价是：“哈哈哈哈哈！”

这种思想的盛行让规格书恶名昭彰。很多人都对我说：“规格书没用，因为没人会按上边写的做，它总是过时的，从来不能反映产品的现状。”

对不起。可能你们的规格书总是过时，不反映产品的现状。但我们的规格书经常更新。随着产品的开发以及不断作出新的设计决定，规格书总在不断更新。规格书总在反映着我们团队对于产品开发和使用的最佳共识。产品代码开发完成之时（也就是产品的功能开发完成了，但还需要进行测试和调试工作），才是规格书定稿之日。

为了减轻大家工作量，我并不每天更新并发布规格书。我通常把规格书的最新版本更新到服务器上，供团队成员参考。在偶尔的产品关键节点，我会打印出带修订标记的规格书，这样大家就不用重读全部文档，而是可以通过浏览带修订标记的部分来查看内容有改动的部分。

07

轻松撰写功能规格书

第三部分：但是……怎么做？

2000年10月4日，星期三

在了解了为什么要写规格书以及规格书应该包括哪些内容之后，我们来看看规格书该由谁来写。

谁来写规格书？

我来讲点微软的历史吧。当20世纪80年代微软刚开始迅猛发展的时候，所有的员工都读过《人月神话》^①这本软件管理的经典著作。（如果你没读过，我强烈推荐这本书。）书中的主要观点是，给一个延期的项目增加开发人员时，会让项目延期更长的时间。原因是当一个团队中的程序员数量为 n 时，每两个人之间的沟通路径数量总和是 $n(n-1)/2$ ，增长速率为 $O(n^2)$ 。

微软的程序员们对如何开发越来越庞大的软件感到担心，因为当时主流的观点认为增加人手会使项目延期更加严重。

查尔斯·西蒙尼在很长时间里都是微软的首席架构师，他提出了所谓编程大师的概念。这个概念的思想是，所有的代码基本上都应该由一个编程大师来负责编写，但是他（她）背后有一群初级程序员做“编程仆从”。编程大师不需要亲自实现并调试每一个功能点，他只负责写出每一个函数的原形，勾

^① 弗雷德里克·布鲁克斯，《人月神话》。

勾勒出大致的轮廓，然后交给某个初级程序员具体实现。（当然，西蒙尼本人的角色是编程大师之上的大师。）“编程大师”这个术语有点中世纪的味道，不够时髦，所以后来微软把这个角色更名为“项目经理”（Program Manager）。

理论上，这个工作模式是用来解决人月神话问题的，因为程序员之间不需要彼此沟通——每个初级程序员只和唯一一位项目经理沟通，所以沟通成本以 $O(n)$ 而不是 $O(n^2)$ 的速率增长。

西蒙尼可能知道匈牙利命名法^①，但他不知道人件^②的原理。没人想要成为编程仆从。这种工作模式无法得到执行。最终微软发现，尽管可能存在所谓的人月神话问题，还是可以通过向团队增派聪明的人员来提高产出，尽管这种做法的边际效用是递减的。我在 Excel 团队的时候团队有 50 个程序员，团队的产出比有 25 个人的时候稍多一些，但肯定没有达到两倍。

后来这种大师/仆从的开发模式被弃用了，但微软公司里头衔为项目经理的那批人还在。一个叫做贾波·布鲁门萨的聪明人重新定位了项目经理这个职位。从此以后，项目经理负责做产品的设计和规格。

之后，微软的项目经理负责需求收集和软件功能的设计，然后撰写功能规格书。通常每个项目经理手下约有 5 位程序员，程序员负责用代码来实现项目经理以规格书的形式定义好的功能。项目经理还需要协调与其他部门人员的合作，这些部门包括营销、文档、测试、本地化等程序员无暇顾及的所有冗赘细节。最后，微软的项目经理还需要对公司的“宏观战略”做到心中有数，这样程序员就能集中精力来搞定眼前的代码。

项目经理的价值不可估量。作为管理者，如果你总是抱怨程序员不懂营销，只专注于技术上的精益求精，那么你需要一个项目经理。如果你抱怨过能写得一手好代码的人写起文章来总是词不达意，那么你也需要一个项目经理。如果你抱怨过产品的发展似乎在左右摇摆中失去方向，那么你还是需要一个项目经理。

① 参见 Microsoft.com 的“匈牙利命名法”条目：[http://msdn.microsoft.com/en-us/library/aa260976\(v=vs.60\).aspx](http://msdn.microsoft.com/en-us/library/aa260976(v=vs.60).aspx)，作者为查尔斯·西蒙尼

② 汤姆·德马科，蒂姆·李斯特，《人件（第2版）》。

如何招到靠谱的项目经理?

大部分公司甚至没有项目经理这个概念,我觉得这太糟糕了。我在微软的时候,项目经理强势的团队通常会开发出成功的产品:Excel、Windows 95 以及 Access 就是很好的例子。相反,有些团队(例如 MSN 1.0 和 Windows NT 1.0)中开发者基本不听项目经理的指挥(这些经理本来就不怎么样,没有号召力也是理所当然的),开发出的产品就不那么成功。

在考虑项目经理人选的时候,需要避免以下三种情况。

- (1) 不把程序员提拔为项目经理。优秀项目经理需要具备的素质(文笔清晰、外交手腕、市场嗅觉、用户视角以及优秀的界面设计能力)在优秀的程序员身上是很难找到的。诚然,有些人能胜任这两种工作,但这样的人才实在是不可多得。通过把优秀的程序员提拔到更高层的岗位来奖励他们,让他们用自然语言而不是 C++ 来写东西,这是“彼得原理”的典型表现:职场中人们经常被提拔到一个不能胜任的职位上^①。
- (2) 不让营销人员做项目经理。我并不歧视任何人,但我觉得优秀的营销人员很少具备设计一个产品所必须具备的技术素养,相信读者们也会同意我的观点。开发过程的管理基本上是一条独立的职业发展道路。项目经理需要对技术十分熟悉,但他们不一定要成为优秀的开发者。项目经理需要研究用户界面设计,和用户打交道,并撰写规格书。他们需要具备和许多不同类型的人群沟通合作的能力,从“很傻很天真”的用户,到那些活在自己的世界里、穿着《星际迷航》制服来上班的不合群的程序员,再到身穿价值 2000 美元的高级西服、满嘴跑火车的销售人员。在某些方面,项目经理是软件团队的粘合剂,其个人魅力非常关键。
- (3) 不让程序员听命于项目经理。这是个很微妙的错误。我在微软当项目经理时,设计了 Excel 对 Visual Basic (VBA) 的支持方案,巨细靡遗地描述了如何在 Excel 中实现 VBA。我的规格书写了大概 500 页。在开发 Excel 5.0 的忙碌时刻,我估计每天有 250 人的工作依赖于我写的那份巨型规格书。我并不知道这些人都是谁,但是事实上,仅仅在 Visual Basic 团队就有十几个人专门给这份规格书写技术说明文档(还不包括 Excel 那边写文

^① 劳伦斯·彼得与雷蒙斯·赫尔合著的《彼得原理》,Buccaneer Books 出版社 1996 年出版。

档的人，以及那些全职工作是维护帮助文件中超链接的人)。然而奇妙的是，我处在报告树的“底部”。你没想错，没有人由我直接管理。如果我想让别人做什么事情，我必须让他们认同这件事值得去做。如果开发主管本·瓦德曼不想做我在规格书中写的某些功能，他可以不做。当测试人员抱怨规格书中的某些功能无法进行完整的测试时，我必须简化它。如果这些人中的任何一个向我汇报工作的话，产品就不会这么棒了。因为那样一来，下属可能不敢去质疑上司的决定。另一方面，出于自负或为了眼前的利益，我可能会用行政手段强制他们按我说的做。而事实是，我管不了任何程序员，要贯彻某个想法，我必须去努力争取大家的认同。这种工作方式确保了团队永远在做正确的事情。

08

轻松撰写功能规格书

第四部分：写作技巧

2000年10月15日，星期日

到目前为止，我们已经探讨了为什么要写规格书，规格书里应该写些什么，以及谁来写规格书。在本系列的第4篇也是最后一篇中，我想和大家分享一些写好规格书的心得。

在那些真正写规格书的团队中，你能听到最多的抱怨就是“没人会去读规格书”。负责撰写规格书的那个人写了很多的东西却没有人愿意读，难免会变得有些愤世嫉俗。就好像是呆伯特漫画中的场景那样，工程师们把10公分厚的规格书沿着工位的隔板垒成一圈，好像在修筑防御工事一样。庞大臃肿的公司在现实中比比皆是，在这些大公司里，每个人花在编写乏味规格书上的时间累计可达数月之久。一份规格书经常是刚一写完就被束之高阁，从来没被翻阅过。而产品开发总是另起炉灶，完全无视规格书的存在。没人愿意读规格书，原因是它们实在是太催眠了。虽然如此，写规格书本身仍然是一个不错的练习，因为至少人们能在写作过程中想想产品设计方面的各种问题。而这无法改变规格书一写完就被束之高阁（没人读，更没人喜欢）的事实，所以人们才会觉得写规格书这种事实在是费力不讨好。

另一个问题是，如果没人读你写的规格书，等到产品开发完成交付的时候会突然冒出很多质疑的声音。有人（可能是管理者、营销人员或客户）会说：“不对啊！你承诺过产品会有蒸贝壳的功能，现在怎么看不到蒸贝壳功能？”程序员如实回答说：“其实没有这功能，你看，规格书第3章第4节的2.3.0.1段明明写着‘没有蒸贝壳功能’。”客户听了当然不会满意，而客户永远是对的，所以倒霉的程序员不得不补充开发，把蒸贝壳的功能加进去（他们对规格书

的必要性更加嗤之以鼻了)。或者一位经理会说：“这个对话框上的文字太啰嗦，而且所有对话框顶部都应该有一个广告区。”程序员会沮丧地说：“我的规格书里早把每个对话框的布局和文字都写得一清二楚，你当初都同意了我才这样开发的！”实际情况是经理并没有真正读过规格书，他看着白纸黑字的时候，大脑就开始神游，惦记着周二要去打高尔夫球。

所以说，规格书如果没人愿意读，内容再好也没用。作为规格书的撰写者，你必须学会使用引人入胜的叙述方法，并且要尽可能写得浅显易懂，免得那些本来脑袋就不够用的人看不进去。

其实只要写作技巧过关，激发人们阅读的兴趣还是很容易的。“只要文笔好”可能并不是你想看到的答案，这对程序员太不公平了。下面我给出 4 条准则，通过遵循它们，你也能写出让人们喜欢阅读的规格书。

第 1 条准则：要幽默

没错，要让人们对你写的规格书感兴趣，最重要的就是要让阅读的过程变得轻松愉悦。别对我说你没有幽默细胞，我不相信。每个人都会时不时冒出一些好玩的想法，只不过通常人们在写规格书的时候会压抑这些想法，认为这么写“不够专业”。然而有的时候，你恰恰需要打破一些条条框框，方能出奇制胜。

如果你通读过鄙人网站上的拙作，你会发现其中有些地方会显得故作轻松幽默，玩笑开得不太高明。往前数 4 个自然段，我就开了一个经理打高尔夫的蹩脚玩笑。我知道自己不是搞笑大师，但至少我努力过。其实努力搞笑的举动本身就挺搞笑的，有一种伤心小丑般让人心酸的喜剧效果。在规格书写作中，举例子的地方总是活跃气氛的好时机。比如，你原本会这样举例说明一个功能的使用场景：

用户输入 Ctrl+N 来创建一张新的雇员表，然后逐个输入雇员的名字。

还可以这么写：

猪小姐^①那娇小圆润的手指不方便按键盘，她借助一支眼线笔，输入Ctrl+N，创建一张新的男朋友表，然后输入唯一的记录“科米蛙”。

如果你读过很多戴夫·巴瑞的专栏文章，就会发现最简单的搞笑方法就是在没有必要写得具体的时候写得具体一点^②。“好斗的哈巴狗”比“狗”要有趣。“猪小姐”要比“用户”有趣。“特殊用户群体”可以用“种植鳄梨的左撇子果农”来代替。“那些不替家犬收拾粪便的人应该受到处罚”这句话如果换一种说法，可以说“那些人该被关进凄苦的监狱，在里面孤独终老”。

另外，如果你觉得搞笑显得不专业，那很抱歉，只能说你缺乏幽默感。（别不承认，缺乏幽默感的人从来不会承认。你哄不了我。）如果在你就职的公司里，人们会因为谈吐风趣、文笔诙谐而看轻一个人，那么请你考虑换一家公司^③。人生苦短，何必待在一个严肃到连玩笑都开不起的地方受罪呢。



第2条准则：像编写用大脑执行的代码一样写规格书

我认为程序员不擅长写规格书的原因如下。

当你写代码的时候，主要是写给编译器看的。我知道代码也是写给人看的，但人们很难看懂别人写的代码^④。对大多数程序员来说，写出能让编译器读取并正确执行的代码已经够难了，而试图写出易于被人类理解的代码，更是难上加难，而且用处不大。举例来讲，你可以这样写：

```
void print_count( FILE* a, char * b, int c ){
    fprintf(a, "there are %d %s\n", c, b);}
main(){ int n; n =
10; print_count(stdout, "employees", n) /* 这段代码是故意写得晦涩难懂的 */ }
```

也可以这样写：

```
printf("there are 10 employees\n");
```

① 猪小姐（Miss Piggy）和科米蛙（Kermit）是《大青蛙木偶秀》中的木偶角色。

——译者注

② 参阅<http://blogs.herald.com/>。

③ 参阅<http://www.fogcreek.com/>。

④ 见第24章。

两种写法的输出结果是相同的。仔细想想，这种思维惯性导致了程序员经常会这么写规格书：

假设有一个函数 `AddressOf(x)`，其定义为从一个用户 x 到一个符合 RFC-822 标准的电子邮件地址（格式为 ANSI 字符串）的映射。假设有两个用户 A 和 B，A 想给 B 发一封电子邮件。于是 A 使用任意（但并非所有）的方法（已在别处说明），初始化一个新消息，然后在“To:”编辑框中输入 `AddressOf(B)` 的值。

同样的内容也可以这么写：

猪小姐想要去吃午饭，于是她新建了一封电子邮件，然后把科米蛙的电邮地址填在“To:”旁边的方框里。

技术注解：地址必须是标准的互联网地址（符合 RFC-822）。

理论上，二者的“含义”相同，但是第一个例子理解起来很费劲，好像在读代码，而第二个例子则很容易理解。程序员写起规格书来，就像是在写艰深的学术论文。他们认为“正确的”规格书只需要“技术上”表述正确就万事了。

这种想法的谬误在于，在写规格书的时候，仅仅做到正确是不够的，还要做到易于理解，用编程术语来讲，就是要让人类大脑能“编译”你写的文字。电脑和人脑的一个重大区别在于，电脑能耐心地等着你定义各种术语，然后加以引用。但是如果一段文字不能引起人们足够的兴趣，人脑是不会注意并试图理解它的含义的。人们不愿意对信息先做解码处理，而是想直接边读边理解内容。对于人类读者，你必须先概述宏观背景，然后再详述具体细节。对于计算机程序，你则可以从头到尾，一直详述各种细节。电脑不会在乎变量名是否有深刻的含义。而人脑则更易于理解生动如画、有故事性的内容，即使只是某个故事的片段也行，因为进化让我们的大脑更擅长理解各种故事。

把一张国际象棋棋盘的布局向经验丰富的棋手展示几秒，如果盘面是经过真实的对弈得到的，他们马上就能记住每个棋子的位置。但是如果你胡乱移动几个棋子，让盘面在真实对弈中不可能出现（比如把卒放在第一行，或是把两个黑象都放在黑格上），棋手们就很难记住盘面了。电脑则不一样，对于能记忆棋盘的计算机程序，记忆可能和不可能盘面的难度是相同的。人脑工

作的方式不是随机存取；大脑中经常使用的神经路径会得到增强，常见的事物往往比不常见的事物更易于理解。

在写规格书的时候，试着想象你面前有一个读者，你要不断地向他们确认是否听懂了每个步骤。请逐句阅读自己写的文字，思考在给定的上下文中，别人是否能深入理解你的思路。如果你的目标受众中有部分人不知道 RFC-822 的含义，你要么给出相关的定义，要么把 RFC-822 写在技术注解里，这样，管理层的读者就不会因为看到过多的技术行话而放弃阅读下去。



第3条准则：写得尽可能简单

不要觉得使用简单的词句会显得不够专业，而故意使用生硬的书面语。用你能驾驭的最平实的语体风格来写。

比如，人们会觉得用“use”显得不够专业，而使用“utilize”这样的词。（关于“不够专业”我想说，当你的某种做法遭到“不够专业”的批评时，你就知道批评者根本没有站得住脚的反对理由。）事实上我觉得人们下意识地轻视写得清楚明白的文字，总是觉得其中的内容缺乏深度和思想。

用短句来说明问题。如果你写长句子的功底不够，不容易把事情说清楚，就把长句拆分成两三个短句。

切忌写出豆腐块式的排版，即整个页面上全是文字。人们会被长篇大论吓住，根本不去读它。你什么时候见过流行杂志或报纸上整页都是密密麻麻的文字？杂志为了避免排版的单调，有时会摘出文章中的一句话，用巨大的字号印在版面的中间。使用前面有数字编号或圆点的列表，或者照片、统计图、数据表，甚至是大量的留白，让版面“看起来”更多样化一些。

要提升一份规格书的排版水平，最好的办法是用大量的截图。一图胜千言。给 Windows 平台上的软件写规格书的人们应该人手一份 Visual Basic，不需要多么熟练的使用，只要会用它搭建出模拟的页面就好。（对 Mac 来说，必须真正的 Basic 语言；对网页来说，用 Front Page 或 Dreamweaver。）然后就可以截屏（Ctrl+PrtSc），把生成的图片粘贴到规格书里。

用巨大的字号印在版面的中间。使用前面有数字编号或圆点的列表，或者照片、统计图、数据表，甚至是大量的留白，让版面“看起来”更多样化一些。

“杂志为了避免排版的单调，有时会摘出文章中的一句话，用巨大的字号印在版面的中间。”

要提升一份规格书的排版水平，最好的办法是用大量的截图。一图胜千言。给Windows平台上的软件写规格书的人们应该人手一份Visual Basic。

第4条准则：重读并修改几遍

我原本打算详细阐述一下这条准则，但是它太浅显了，不言自明。写完规格书之后请重读并修改几次，就这么简单。如果发现某些句子不是非常通俗易懂，就重写。

我用在写第4条准则上的时间太少了，因此我决定再加写一条。

第5条准则：尽量不要套用模板

有一种做法乍一看挺有吸引力，就是先写一份标准模板，然后把所有规格书往里套。你可能会觉得“每篇规格书的格式看起来一致”很重要。其实不然。格式不一致有什么关系呢？你书架上的每一本书都一样吗？你想让它们都一样吗？

使用模板还有一个缺点，如果你认为某件事对所有功能特性都很重要，你就会在模板中为它增加相应的章节。例如，比尔老大宣布从现在起，所有微阮（Microsquish）公司的产品都要有一个互联网组件。于是规格书模板中增加

了“互联网组件”一节。无论谁写规格书，无论写的产品和互联网组件有没有关系，他们都必须填写“互联网组件”一节的内容，甚至微阮键盘的规格书也不例外。（你现在知道键盘上那些“网络购物”之类的按钮是怎么回事了吧）。

随着章节越加越多，模板会变得臃肿不堪。（谁会需要在规格书里罗列参考书目或是词汇表？老天！）这种庞大的模板会让打算写好规格书的人望而生畏。

总之，规格书是你希望让别人阅读的文档。从这个角度看，它和《纽约客》上的散文或是大学生写的学科论文是一样的。你听说过哪个教授会给学生提供作业报告的模板？你读过的哪两篇优美散文能套进同一个模板里？快打消套用模板写规格书的念头吧。

09

轻松掌控软件开发进度

2000年3月29日，星期三

去年10月，一列货运列车的广告轰轰烈烈地席卷了美国东北部地区，这趟列车名叫 Acela，运营路线从波士顿到华盛顿。随处可见 Acela 的电视广告、巨型广告牌以及海报，人们大概觉得 Amtrak 公司的货运服务将供不应求。

也许吧，但 Amtrak 最终没能证实这一点。Acela 的研发多次延期，导致当营销部门开足马力为它做宣传的时候，Acela 服务甚至还无法使用。这让我想起以前听到一位市场部经理说的话，当时他的产品在上市之前一个月就得到了强烈的反响。他说：“宣传战大获全胜！真可惜市场上还买不到这破玩意儿！”

有些狂妄自大的游戏公司喜欢在网站上自我吹嘘，说他们的下一款游戏“将在一切准备就绪”之后发布。具体日程？我们才不会用具体日程来限制自己！我们是最酷的游戏开发者！然而大多数公司并不敢这样理直气壮，不信你看 Lotus。他们开发 Lotus 123 软件的 3.0 版本时，需要运行在当时不是很普及的 80286 电脑上。他们决定把产品延期 16 个月，硬是把这个产品打磨到能在普通 8086 机器上的 640K 内存里运行。等到他们做完的时候，才发现微软通过开发 Excel 软件在技术上领先了 16 个月，并且更具讽刺意味的是，8086 已经不流行了！

我写这篇文章的时候，网景公司几乎已经把网页浏览器 5.0 版本的发布推迟了两年。部分原因是因为他们经常自我否定，把以前的代码全部扔掉然后从头再来。Ashton-Tate、Lotus 以及苹果公司的 MacOS 也犯过同样的错误，这些公司的产品大多数已经进了软件历史的回收站。网景浏览器在这两年的时间里，市场占有率从 80% 掉到了 20%，而网景公司面对竞争无动于衷，因为他们的主产品支离破碎，根本无法支撑任何定向的升级开发。造成这种局面

的因素有很多，其中最主要的是决策失误，就像是导致网景自我毁灭的一颗核弹。（杰米·扎文斯基在著名的文章^①中愤怒地披露了其中的细节，在第24章中将详细介绍。）

所以，制定开发计划是绝对必要的。但是几乎没有几个程序员愿意制定开发计划。根据我的经验，大部分程序员都想不经过计划阶段就直接编码。少部分的程序员在老板的强迫下不情愿地做了计划，但是除了高层管理人员，没有人会相信他们的计划表能够贯彻执行，高层管理人员还相信“没有一个软件开发过程能按时完成”以及不明飞行物的存在。

为什么大家不想做计划？有两个主要原因。第一，做计划真的很难。第二，没人相信它的存在有任何价值。如果制定的计划无论如何都不准确，那还做它干什么？既然计划表总是错的，而且随着项目进展错得越来越离谱，那干嘛要费力不讨好呢？

下面我会告诉大家一个把计划做准确的方法，而且实行起来非常简单、轻松。

1. 使用微软Excel

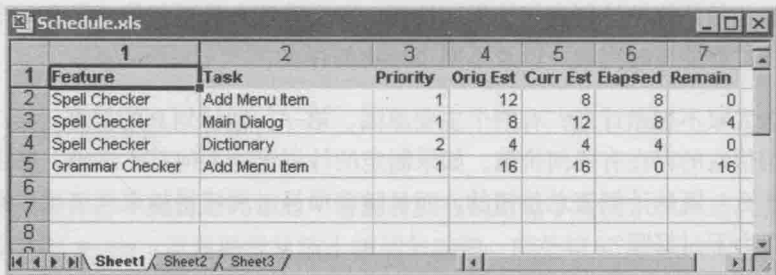
不要用微软 Project 之类花哨的东西。微软 Project 的问题在于它假定你愿意投入大量的时间处理任务的依赖关系。依赖关系是指某两个任务之间的关系，其中一个必须在另一个结束后才能开始。我认为软件开发中的依赖关系太明显了，不值得专门去记录和处理这方面的内容。

Project 的另一个问题是它假定用户希望只按一个按钮，就“重新平衡”整个计划表的工作量。重新平衡不可避免地意味着要重新安排事项，并把事项安排给不同的人。对于软件开发活动来说，这完全不可理喻。程序员的工作可不能随随便便地互相交换。约翰修复丽塔的 bug 需要消耗的时间可能是丽塔自己修复 bug 时间的 7 倍。如果你试着让前端界面程序员解决一个 WinSock 问题，她可能要耗费一周的时间来学习 WinSock 编程的知识。Project 这个软件是为建造办公楼、而不是为建造软件而设计的。

^① 参阅<http://www.jwz.org/gruntle/nomo.html>。

2. 保持简单

我的计划表标准格式非常简单，容易记忆。首先标出 7 栏：



	1	2	3	4	5	6	7
1	Feature	Task	Priority	Orig Est	Curr Est	Elapsed	Remain
2	Spell Checker	Add Menu Item	1	12	8	8	0
3	Spell Checker	Main Dialog	1	8	12	8	4
4	Spell Checker	Dictionary	2	4	4	4	0
5	Grammar Checker	Add Menu Item	1	16	16	0	16
6							
7							
8							

如果团队中有好几个开发者，你既可以为每个人单建一张表单，也可以添加一栏，填上负责某项任务的开发者名字。

3. 每个功能点应该分成几个子任务

举例来说，功能点可以是给程序增加一个拼写检查器。增加拼写检查器的任务由好几个独立的任务组成，可以分别交给几个程序员来做。计划中最重要的部分，就是分解任务列表。下面到了最重要的一条准则。

4. 只有负责写代码的程序员本人才能准确地预估时间

那种管理者写好计划、交给程序员执行的做法注定行不通。只有真正干活的程序员才知道实现某个功能点需要经过哪些步骤。也只有程序员才能估计每个步骤要花多长时间。

5. 适当细分任务，保持合适的颗粒度

这一点对于计划表能否执行至关重要。任务的时间必须以小时数计量，而不是以天数。（一看到某份计划表的任务是以天甚至是以周为单位的，我就知道它肯定无法执行。）你可能觉得细分一份计划表只是让它更精确。大错特错！当你把较粗的任务分解为子任务时，你会发现得到的不是更加精确的工作量估计，而是完全不同的工作量估计。分解前后的差异非常大。为什么会这样呢？

当你做任务细分的时候，你在强迫自己想清楚完成它需要采取哪些步骤。先写子程 *foo*。新建一个对话框。读取某文件。这些步骤需要的时间很容易估计，因为你以前写过类似的子程，新建过对话框，读取过同类的文件。

如果你犯懒，只是泛泛地写出“大致”的任务（“实现语法更正功能”），那你还没有真正想过究竟要做什么。连要做什么都没想清楚，就更不可能估计出需要多长时间来做了。

通常的规则是，每个任务的耗时应该在 2 小时到 16 小时之间。如果你的计划表上有一个 40 小时（一周）的任务，说明你还没有把任务分解得足够小。

做任务细分还有另一个理由：它迫使你和功能点进行设计。如果你拍脑袋想出一个叫“互联网集成”的功能点，然后给它安排了 3 周时间，那你完了，伙计。你必须强迫自己想清楚要写哪些子程，也就是把功能点“定死”。通过在设计层面进行强制性的提前设计，你将避免许多软件项目中的不稳定因素。

6. 记录原始和当前的时间估计

首次把一项任务加到计划表中时，先预估它需要的小时数，把值填在 Org Est（原始预估）和 Curr Est（当前预估）两栏中。随着开发的进展，如果某项任务所需要的时间比预想的要长（或短），就更新它的 Curr Est 栏数据。根据需要，可以多次更新。从以前的失误中总结，不断磨练自己估计任务时间的能力，这是最好的学习方法。大部分程序员都不知道该怎么估计做一件事需要

的时间。没关系。只要你不断地学习，不断更新你的计划表，这份计划表将最终发挥它的作用。（你可能需要砍掉功能点或延期开发，但是计划表还是有用的，因为它不断地提醒你是否需要砍掉功能点或是推迟开发。）我发现大部分程序员只要经过一年的锻炼，都能成为很好的项目规划者。

当一项任务完成的时候，Curr Est 的值会等于 Elapsed 值（实际用时），而 Remain 值（剩余时间）将变成 0。

7. 每天更新Elapsed一栏的值

大可不必在编程的时候用秒表计时。在你回家之前，或者对于极客来说，在你钻到桌子底下睡觉并假装自己工作了 8 个小时（哈！）之前，想想今天都做了哪些任务，然后把 8 个小时按工作量加权分配给各个任务，更新 Elapsed 一栏的值。剩余时间一栏的值会由 Excel 自动计算并更新。

同时，还要更新这些任务的 Curr Est 一栏，以反映最新的实际开发进度。每天更新计划表只占用大概 2 分钟的时间。这就是为什么我叫这种方法为轻松做计划的方法：快捷而且简单。

8. 把休假、节日等也考虑在计划中

如果你的计划表周期为一年，每个程序员都会休假大概 10 到 15 天。你的计划表中应该把休假、节日以及其他任何消耗人们时间的活动都包括在内。最好把剩余时间累加并将这个数字除以 40——即距离交付还有几周的工作量——把软件交付日算出来。

9. 把调试代码的时间也考虑在计划中！

调试时间是最难估计的。想想你做的上一个项目。调试时间很可能是写代码

时间的 100%~200%。调试必须是计划表中的一项内容，也许将是其中耗时最长的一项内容。

举个例子来说明。假设一个开发者在写某功能。Org Est 为 16 小时，但目前已经耗时 20 小时，看起来还需要 10 小时才能完成。于是他在 Curr Est 下填了 30，在 Elapsed 填了 20。

在一个开发里程碑的末尾，总会积攒很多这样的“延期”任务。理论上，为了解决这些延期任务并按时交付，我们需要砍掉一些功能点。恰好有一个叫缓冲的功能点，它已经占用了很长的时间，我们可以先把它砍掉。

原则上开发者应该边写代码边进行调试。程序员在修复已有 bug 之前，绝不应该写新代码。必须时刻压低 bug 数量，原因有二。

- (1) 在当天写的代码中，bug 总是更容易修复的。如果等到写完代码一个月后再来修复其中的 bug 就很困难且耗时了，因为你已经忘了代码的细节。
- (2) 修复 bug 就像做科研。不可能预估什么时候能发现并解决 bug。如果在任何时间点都只有一两个明显的 bug，那么很容易预估产品什么时候能交付，因为之后工作中无法估计的科研成分的比例很小。另一方面，如果 bug 堆成了山，那么就不可能估计何时能把它们全部修复。

如果开发者能做到边开发边调试，那还有必要在计划表中列出调试一项吗？即使你们做到了边开发边调试，在一个开发里程碑的末尾，还是会不可避免地出现很多 bug，这些是测试者（包括内部测试和 beta 测试）发现的真正“难解决”的 bug。

10. 把集成也考虑到计划表中

如果你们的程序员人数多于一，就会不可避免地出现两个程序员的工作不一致，需要解决代码冲突的情况。可能两个人都实现了功能类似的对话框，在不必要的方面出现了不一致。总要有某个人把所有菜单、键盘快捷键、工具栏按钮等再过一遍，对之前每个人随意添加的界面元素进行清理。两个人同时签入代码时可能会出现编译错误，这必须得到修复，而且必须成为计划表中的一项。

11. 把缓冲时间也考虑进计划中

很多时候拖拉是不可避免的，因此需要预留出缓冲的时间。有两种重要的缓冲值得考虑：第一，对于消耗的时间超出预估时间的任务进行缓冲；第二，对于你不知道该做什么的任务进行缓冲，这种任务的出现经常是因为管理层认定某某功能极其重要，必须出现在下一个版本中。

你可能会感到惊讶，休假、节日、调试、集成和缓冲时间加起来，可能要比真正干活的时间还要长。如果你对此感到惊讶，你写代码的时间一定不长，对吧？千万不要低估这些琐事占用的时间，否则风险自负。

12. 永远不要让开发经理压缩程序员预估的开发时间

许多新手软件开发经理觉得，通过“勒紧”计划表（让任务的时间短得不合理）能够“激励”程序员，让他们快点干活。我认为这种激励方法非常愚蠢。作为程序员，如果落后于计划表上的进度，我会感到情绪低落、沮丧且缺乏动力。而当进度比计划超前时，我会轻松愉快，而且非常有效率。总之开发计划表不是玩心理游戏的地方。

作为一个程序员，如果经理迫使你削减自己对任务的预估时间，那你就这么干。在计划表上新建一栏，标题为“里克的估计”（假设你叫里克），然后写下你自己估计的时间。然后让你的经理填写 Curr Est 一栏，他爱写啥写啥，你直接忽略。当项目做完的时候，看看谁的估计更接近实际情况。其实根本不用动手实施，只要这么威胁经理就非常有效，他马上会意识到，这么做其实是在打赌你到底干得能有多慢！

为什么不称职的经理会让程序员削减对任务的预估时间呢？

当项目启动时，技术经理和销售人员接洽后，拟出一个功能列表，以为大概要花 3 个月时间开发，实际上要花 9 个月。当你不考虑具体步骤就估计写代码的时间时，假设估计的时间为 n ，那么实际上更准确的时间可能是 $3n$ 。这

样再来做真正的计划表时，当你把所有任务的时间加起来，意识到项目需要的时间远远超过估计值，项目排期带来的压迫感将渗入你的每个毛孔。

为了赶上进度，不称职的经理会想办法加快人们干活的速度。这是不切实际的。也许可以雇佣更多的人干活，但新员工需要一段适应期，可能几个月的时间里工作效率都只有 50%（并且拖累了那些负责培训新人的员工）。无论如何，在这个行业里，至少要花 6 个月才能增加一个优秀的程序员。

你也许可以暂时地让程序员的代码输出量提高 10%，代价是让他们一整年都满负荷运转。这么做收益不是很大，而且难以长期持续，形同杀鸡取卵。

你也许可以恳求每个人努力工作，不管他们已经有多累，可能会让代码产出量提高 20%。但是这样一来，调试的时间将翻上一倍。这实在是自讨苦吃的愚蠢做法。

但是无论怎么做，你都不可能把 $3n$ 的时间降低到 n ，永远不会。如果你觉得你们公司能做到这一点，请发邮件告诉我，我好做空你们公司的股票。

13. 计划表就像积木

如果你有很多积木，没办法装进一个盒子里，那么你有两个选择：找一个更大的盒子，或者丢掉一些积木。如果有一个项目你以为 6 个月能交付，而计划表中排出了 12 个月的工作量，那么选择有两个：要么延期交付，要么删掉一些功能点。你不能缩小木块，如果你假装你可以，那只是在自我欺骗，同时失去了真正把握未来的好机会。

用这种思路做计划的另一个好处是，强迫你删减功能点。为什么这是件好事呢？假设有两个功能点，其中一个确实有用，它将让你的产品从优秀迈向伟大（比如网景浏览器 2.0 中的表格功能），另一个功能实现起来非常容易，而且程序员会非常乐意实现它（比如 BLINK 标签），但是实际上没有什么用处，也不能促进销售。

如果你没有计划表，程序员会先做简单而有趣的那个功能点。然后时间不够了，你不得不推迟开发那个有用且重要的功能点。

而如果你做了计划的话，甚至在开始工作前，你就将意识到必须要砍掉某些

功能点才能按时完成，于是你砍掉简单而有趣的功能点，只做有用且重要的功能点。通过强迫做减法，你们将开发出更强、更好、功能配比更优秀的产品，而且能更早地交付。

我记得开发 Excel 5 时的情况。我们最初的功能列表非常庞大，时间远远超出计划。这怎么办？这些全都是很重要的功能点！没有宏编辑向导怎么可以！

后来，我们别无选择，为了赶上交付日期，只能删繁就简，保留我们认为是“精简到骨头里”的功能。每个人都对删掉的功能点耿耿于怀。我们自我安慰地说，这些功能点并没有被砍掉，只是被延期到 Excel 6 中再开发，毕竟它们不是最核心的功能。

当 Excel 5 的开发接近尾声，我和同事埃里克·麦克曼一起，开始起草 Excel 6 的功能规格书。我们把所谓的“Excel 6”功能列表从头到尾过了一遍，令我们感到震惊的是，这些被砍掉的功能其实一无是处。实际上，没有一条是值得去开发实现的。我不记得其中的任何一条在接下来的三个发行版本中得到了实现。为了赶上进度而作出删减功能的决定，是我们做过最正确的事。如果我们不这样做，Excel 5 的开发时长将是实际时长的两倍，并增加 50% 的无用功能。（我十分肯定这正是发生在网景浏览器 5.0 和谋智公司的事情：他们没有计划表，没有明确的功能列表，没人愿意砍掉任何功能，所以他们离交付遥遥无期。即使将来软件交付了，他们还是有许多鸡肋功能，比如 IRC 客户端这样的根本不值得花时间去开发的功能。）

你应该知道的 Excel 特性

作为管理软件开发进度的工具，Excel 是一个伟大的产品，原因之一是大多数 Excel 程序员使用它只是为了维护他们的软件开发计划表！（他们中大多数都没有商业策划的能力，是如假包换的程序员！）

共享列表 使用“文件->共享列表”命令可以让大家同时打开并编辑同一份文件。在整个团队持续更新同一份计划表的时候，这真的很有用。

自动过滤 这个功能可以过滤计划表，比如，你可以只看指派给你的所有功能点。通过和自动排序功能配合使用，你不仅可以看到指派给你的所有功能，还能根据优先级排序，这完全可以作为你的“待办事项表”，非常方便。

数据透视表 这是查看数据汇总和交叉分析表的好方法。例如，你可以做一张表，显示每个开发者在每个任务上的剩余时间。数据透视表就像是切片面包和巧克力奶昔。你最好掌握它的用法，因为它让 Excel 如虎添翼。

工作日函数 包含在 Excel 的分析工具包中，它能帮你从一张计划表中得到日历日期。



每日编译是你的朋友

2001 年 1 月 27 日，星期六

1982 年，我们家成为了以色列最早一批 IBM-PC 的购买和使用者。我们甚至到仓库去等着电脑从港口运过来。我成功说服了父亲购买了当时配置最好的版本，有两个软盘，128K 内存，一个点阵式打印机（用来快速打印草稿）以及一个“兄弟”牌的菊瓣字轮打印机，后者打印时候的声音就好像机关枪，甚至比机关枪还响。我记得我们买了当时所有能买到的配件：PC-DOS 1.0、价值 75 美元的技术参考手册，里面有 BIOS 的完整源代码、宏编译器，还有令人惊叹的 IBM 单色显示器，它有整整 80 列宽，甚至还支持小写字母！整套设备的价格大约是 1 万美元，包含了以色列当时高昂的进口税。奢华！

“所有人”都知道 BASIC 是小儿科，你用它只能写出意大利面条似的混乱代码，脑子也被搞得一团糟。所以我们又斥资 600 美元购买了 IBM Pascal，当时使用 3 张软盘才能装下它。编译器的第一遍工作在第一张软盘上，第二遍工作在第二张软盘上，链接器在第三张软盘上。我写了简单的“hello, world”程序，然后编译。总共整整花了 8 分钟。

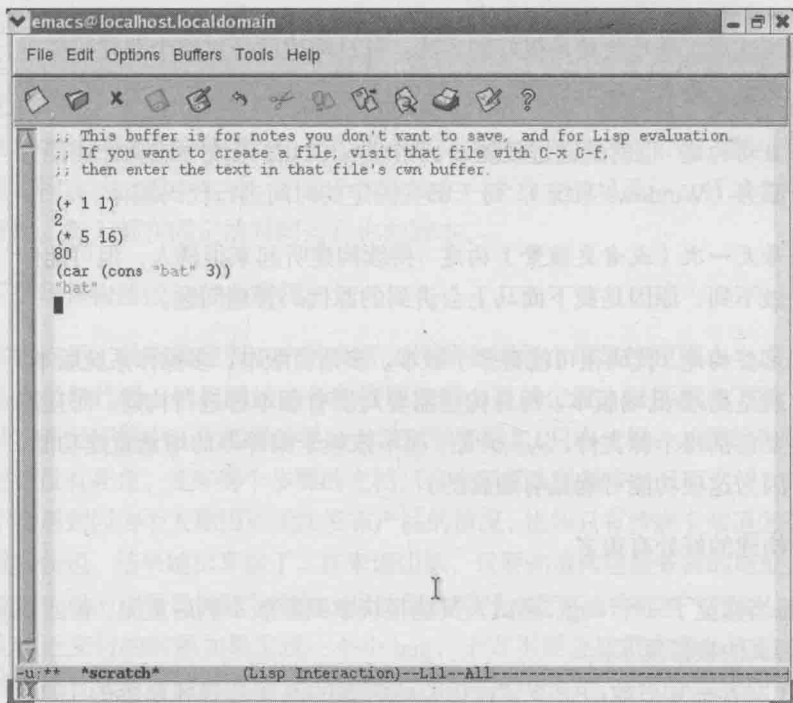
嗯，时间有点长。我写了一个批处理脚本让过程自动化，然后把编译时间降低到了 7.5 分钟。有进步。但是当我试着写一个比较长的程序时，则总是不能完成，大部分的时间里我都在等待编译结束。我曾经听一位资深程序员说：“当年我们在办公室里放了一个仰卧起坐板，写好程序等待编译的时候我们就在上边锻炼。经过几个月的编程训练，我练出了六块腹肌。”

然后突然有一天，在丹麦诞生了一个叫 Compas Pascal 的不可思议的程序，菲利皮·卡恩买下它并更名为 Borland Turbo Pascal。Turbo Pascal 的表现令人震惊，功能和 IBM Pasca 完全相同，但它在运行的时候只占用 33K 的内存，而且它还自带了一个文本编辑器。这实在是太令人惊奇了。更令人惊奇的是，

你可以在一秒内编译完成小型的程序。这给我的感觉，就好像发现一个从来没听过的公司推出了别克 LeSabre 的复刻版，每小时能跑 1 百万英里，消耗的汽油少到一只蚂蚁喝掉都不会感觉不舒服。

突然间，我的编程效率大大提高。

我在这个时候听说了 REP 循环的概念。REP 是“Read, Eval, Print”的缩写，意思是“读取、求值、输出”，它描述了 Lisp 解释器的看家本领：“读取”你的输入，对其“求值”，然后“输出”结果。举一个 REP 循环的例子：我用键盘输入一些字符，Lisp 解释器读取输入，对其求值，然后输出结果。



在写代码的时候，你其实是在实践一个比 REP 循环规模稍大的版本，叫做编辑-编译-测试循环。你编辑代码，然后编译、测试，查看结果是否符合预期。

关键在于，你写程序时必须一遍又一遍地循环这个过程，所以很自然地，编辑-编译-测试循环运转得越快，你的编程效率就越高（当然，这还受到即时编译的实际速度的制约）。所以，计算机程序员有非常正当、科学的理由来要求公司配备真正快速的硬件设备，而编译器开发者会尽其所能地让编辑-

编译-测试循环变得更快。Visual Basic 在用户输入一行代码后就进行解析和词法分析，这样最后的编译就非常快。Visual C++提供的增量编译、预编译 header 和增量链接功能，也是为了加快编译速度。

一旦你开始在大型团队中工作、与很多开发者和测试者配合的时候，你会再次遇到一个更大规模的循环（有点分形的味道）。测试者找到并报告代码中的一个 bug。程序员修复 bug。测试者要过多久才能得到修复过的代码？在某些开发团队，这个报告-修复-复测的循环要持续数周时间，这样的团队效率很低。为了让整个开发过程平滑运转，你的团队需要下功夫，使这个循环加速。

要做到这点，每日构建是很好的方法。每日构建就是对整个源代码树进行自动、完整、每天一次的构建。

自动构建 也就是通过设置 cron 作业（UNIX 系统）或者计划任务服务（Windows 系统），每天都在固定的时间进行代码编译。

每天一次（或者更频繁）构建 持续构建听起来很诱人，但可能你做不到，原因是我下面马上会讲到的源代码管理问题。

完整构建 代码很可能有多个版本。多语言版本、多操作系统版本，或是高端/低端版本。每日构建需要对所有版本都进行构建。而且需要包括每个源文件，从头开始，而不依赖于编译器的增量重建功能，因为这项功能可能是有瑕疵的。

每日构建的好处有很多。

- (1) 每当修复了一个 bug，测试人员能很快拿到新版本然后重测，验证 bug 是否真的被修复了。
- (2) 开发者感觉更放心，不会再担心他们做的改变会弄坏系统的 1024 个发布版本，不需要真在桌上放一台 OS/2 设备，就能测试该平台上的系统表现。
- (3) 开发者在每日构建之前提交代码变更，这样就不用担心会因为“破坏构建”而影响到他人，也就是说，可以避免不恰当的变更导致所有人都不能编译。这就好像是整个开发团队都蓝屏死机了，由于经常有程序员忘了把新文件加到代码库中，这种情况也很常见。这导致代码虽然在他们自己的机器上可以编译通过，但其他人检出代码时就会出现编译链接错误，什么活都干不了。

- (4) 需要提前使用产品半成品版本的外部团队，如营销、beta 测试用户网站等，可以选取一个大家认为相对稳定的构建版本，使用一段时间。
- (5) 如果维护了所有每日构建的存档，当你发现一个非常古怪、毫无头绪的新 bug 时，你可以在历史存档上做二分查找，找出 bug 第一次出现的那个时间点。将这个信息与优秀的代码管理相结合，你也许能追查到该 bug 是哪次签入的代码引入的。
- (6) 当测试员报告了一个程序员认为已经修复过的 bug 时，测试员可以指出问题是在哪个构建版本发现的。然后程序员就可以确认该版本在签入修复补丁后，问题是否解决。

你可以用下面的方法来实现每日构建。你需要一个每日构建服务器，可以是你能找到的最快的计算机。写个脚本，从代码库（你们肯定在用源代码管理系统，对吧？）签出整个代码树，然后从头开始构建，覆盖每个发行版本。如果有安装包或设置程序，也要构建。你交付给客户的每样东西，都应该能从每日构建的过程中生成。把每个构建放在一个独立的目录下，目录名中写上日期。每天都在固定的时间运行你的脚本。

以下是每日构建的一些小窍门。

- 关键在于，最终构建版本中的所有文件都必须能由每日构建脚本生成，从签出最新代码，到把网站的各种资源放到合适的位置，以供用户下载（当然在开发环境中，你部署的是一台测试服务器）。只有这样，才能确保构建过程没有死角，没有哪个步骤的文档只存在于某人的脑中。只有这样，你才不会遇到因为个人原因而无法发布产品的情况，比如只有沙妮卡知道怎么生成安装包，结果她出车祸了。在朱诺团队，只要知道构建服务器的地址，知道怎么双击“每日构建”图标，你就能从零开始完整地构建整个产品。
- 在临近交付的时候如果发现一个小 bug，千万不要直接就在每日构建服务器上修复 bug 后交付产品，这样带来的后果不可预知，会让你发疯。所有交付的代码，都必须通过一次性签出全部的代码，由完整、干净的每日构建脚本来生成，这是一条黄金法则。
- 把编译器的警告通知等级调到最高（微软系工具里是-W4，gcc 里是-Wall），然后设定一旦遇到最小的警告，就停止构建。
- 一旦每日构建失败，就将面临整个团队都无法工作的风险。停下手头的所有工作，不断地重试构建，直到问题解决为止。所以有些日子会存在多次每日构建。
- 每日构建脚本应该把构建失败的信息通过电子邮件发送给整个开发团队。

用 `grep` 获取日志中的“错误”或“警告”，并把这些信息也加进电子邮件中很简单。脚本还可以把状态报告加到某个大家能看到的 HTML 页面里，让程序员和测试员快速了解哪些构建是成功的。

- 在微软 Excel 团队工作时，我们有一条规矩很有效，就是谁的代码造成了构建失败，谁就负责构建的维护工作，直到另一个人犯规来代替他。这不仅巧妙地敦促了大家不要破坏正常的每日构建，也几乎让每个人都轮流担负起构建管理员的职责，在这个过程中，所有人都对构建的工作原理有了更多的了解。
- 如果整个团队的工作地点都在同一个时区，那么可以利用午餐时间进行构建。每个人在吃午饭前签入最新的代码，大家吃饭时代码也在构建中，吃完饭回来的时候，如果构建失败了，就先修复错误。等到构建成功了，每个人就可以放心地签出最新的代码，不必担心会因为构建有问题而无法工作。
- 如果你的团队在不同的时区工作，要小心安排每日构建的时间，避免一个时区的人阻碍了另一个时区的人工作。在朱诺团队里，纽约的同事在当地时间晚 7 点签入代码后下班。如果构建失败，印度海德拉巴的同事一上班（大概是纽约时间晚 8 点）就没法干活，这一天就白费了。后来，我们构建的频率改为每日两次，时间分别在两个团队下班前的 1 小时，从而解决了互相干扰的问题。

拓展阅读

- 一些对每日构建工具的讨论，参阅 <http://discuss.fogcreek.com/jelonsoftware/default.asp?cmd=show&ixPost=862>。
- 实践每日构建的重要性让它跻身提升代码质量的 12 个步骤，见第 3 章。
- G. 帕斯卡·扎克瑞的著作《观止》^①中有很多关于 Windows NT 团队实践（每周）构建时发生的有趣故事。
- 史蒂夫·麦康奈尔关于每日构建的网站，参阅 <http://www.construx.com/>。

^① G. Pascal Zachary, *Showstopper! The Breakneck Race to Create Windows NT and the Next Generation at Microsoft* (The Free Press, 1994).



完美主义者是如何修复bug的

2001年7月31日，星期二

软件的质量，或者说软件质量的缺失，一直遭到大家的抱怨。我开了自己的公司以后，终于决定要对此做出一些努力。最近两周，我的公司Fog Creek一直在忙着发布FogBUGZ的增量版本，致力于清理所有已知的bug（有30个左右）。

对软件开发来说，修复bug是好事，不是吗？难道它不总是好事吗？

并不是！

修复bug这件事，只有在它带来的价值高于其成本时，才值得去做。

修复bug的价值和成本虽然很难用金钱去衡量，但也并非做不到。举个例子，假设你运营一家花生酱果冻三明治工厂。你的工厂每天产出10万个三明治。最近推出了一种新口味（蒜香花生酱加哈瓦那辣酱）之后，产品人气爆棚。工厂每天开足马力只能生产10万个三明治，而需求量估计在接近20万个三明治。产能已经无法提高了。每个三明治的利润是15美分，也就是说，因为产能不足，你每天都在损失1.5万美元的潜在收入。

建一个新工厂的成本太高。你缺乏资金，而且担心辣味/蒜香三明治的流行很快会过去，但还是不甘心放着每天1.5万美元的收入不去挣。

幸好你雇佣了杰森，一名14岁的少年黑客。他曾经黑进你工厂的电脑，并且自称有办法让流水线的工作速度加快1倍，这是他从Slashdot上看到的一种技术，叫超频。而且在测试运行中，超频似乎管用。

然而有一个问题，让你犹豫要不要大规模使用这种技术。有一个小小的bug，每隔一个小时候左右，就有一个三明治会被压碎。杰森想修复这个小问题，他

估计需要3天时间。你是等他修复bug再开工呢，还是让机器带着bug运转？

如果你决定把软件的部署时间推迟3天，你将损失45 000美元。而这一决定将为你节省，嗯，大概72个三明治的材料费。（无论你的决定是什么，杰森都将在3天后修复这个bug。）我不知道来自星星的你是怎么看这个问题的，但是在我们的地球上，这么多三明治的价值远远达不到625美元。

我说到哪了？哦对，有时候不值得去修复bug。再举一个不值得修复的bug的例子：如果bug只出现在打开特别大的文件时，而且只会影响使用OS/2系统的单机用户，而据你了解，这部分用户根本不用大型文件。这种情况下就别修复bug了。比这坏的情况多的是。基于同样的思考，我基本上不再关注使用16色屏幕的用户，或者使用预装的Windows 95系统、7年都没有更新过的用户。这样的人不会花钱购买商业软件产品的。相信我。

但大多数情况下，bug还是值得去修复的。即便是那些“无害”的bug，也会损害公司和产品的名声，长远来看，会对公司收入产生显著影响。产品一旦给人们造成bug成堆的印象，就很难再摆脱这种印象了。所以，在推出产品的.01版本之前，请参考以下步骤来找到并修复合适的bug，或者说那些根据投入产出比来看值得修复的bug。

1. 尽可能收集bug的所有信息

以我们公司的产品FogBUGZ为例，我们使用两种方法来收集bug信息。首先是在免费体验版软件的服务器上捕获bug，尽量收集相关的信息，然后用邮件发给整个开发团队。听起来大家的邮箱会被bug报告淹没，但这是好事。例如，我们通过这种方式，发现有很多人在“Fix for”页面没有按规定输入日期，而我们的系统对此甚至没有错误提示，直接就崩溃了（具体到一个网络应用，这意味着用户将看到一个丑陋的IIS错误页面）。抱歉。

我在Juno团队工作时，我们有一个更酷的系统，能从“案发现场”自动收集bug。我们用TOOLHELP.DLL设置了一个处理模块，每次Juno崩溃时，这个模块会在死前的一点时间里把错误栈输出到一个日志文件里。下次程序可以联网的时候，它会用邮件上传日志文件。在beta测试阶段，我们通过收集日志文件，整理出所有系统崩溃的状况，然后录入bug跟踪数据库。通过这种

方法，我们找到了几百个导致崩溃的bug。如果用户数量达到一百万，会出现五花八门的崩溃原因，通常和内存过低或电脑配置太烂有关。（你听说过帕卡德·贝尔牌电脑吗？）你的代码可能是这么写的：

```
int foo( object& r )
{
    r.Blah();
    return 1;
}
```

这段代码会因为引用值r为NULL而崩溃，这完全是莫名其妙，因为C++的语言机制保证了不可能存在指向NULL的引用。你别不信，经过长时间的堆栈日志收集和分析，在汇总了几百万用户传来的崩溃信息后，我们发现这种地方竟然确实会出错，简直不敢相信自己的眼睛。（而且没有人想修复这些bug。天哪，拜托这些用户赶快换台新电脑，这回不要乱装各种奇怪的共享软件了。）

我们收集bug的第二个方法是把技术支持的每次电话看作出现bug的证据。每接一个电话，我们就努力寻找排除故障的办法。比如，旧版FogBUGZ在安装阶段会假设软件使用匿名账号来连接互联网。这个假设在95%的情况下都成立，只有5%的情况不成立，但属于这5%情况的用户都拨打了我们的服务热线。所以我们修改了安装程序，提示用户输入互联网账号。



2. 衡量修改bug的成本和收益

虽然不太可能精确地衡量修复每个bug的成本和收益，但你可以这样做：要求业务部门对自身原因造成的技术支持服务“买单”。20世纪90年代初微软曾有过一次财务重组，之后每个产品部门都要负担相应产品的全部技术支持费用。所以产品部门开始要求PSS（微软的技术支持部门）定期提供排名前10位的bug列表。当研发团队集中精力解决这些bug时，产品支持服务的成本大大下降了。

这种做法已经和新的流行趋势相左，目前绝大多数的大型公司都让技术支持部门自行承担运营费用。Juno公司的技术支持部门为了实现收支平衡，决定向用户收取服务费。这种做法相当于把bug导致的经济负担转嫁到用户头上，这会让bug的负面影响变得更加难以估量。（而且客户会因为替你的产品bug

买单而心生怨恨，然后告诉身边的朋友。这种负面口碑带来的损失完全是不可估量的。但在此我要为Juno说句公道话，因为它的产品本身是免费的，所以请不要再抱怨了。)

有一种两全其美的办法，如果服务电话中的问题是产品的bug导致的，就不向打电话的用户收费。微软就是这么做的，效果很好，我给微软打电话从没花过钱，哈哈。每一通这种电话的成本（当年是245美元，不知道现在是多少钱）都要记到开发团队账上。只要客户打电话投诉bug，这一份产品就算白卖了（甚至还要亏出好几倍的钱），这在经济上会提高产品团队修复bug的积极性。这让我想起了另一个话题，为什么DOS游戏开发是个很可怕的行业呢？由于为了让游戏画面好看、运行流畅，你需要给视频驱动程序打奇怪的补丁，而只要出现一个投诉视频驱动问题的技术支持电话，20份游戏副本的售卖利润就打水漂了，再加上付给Egghead、Ingram和MTV电视台的广告支出，你卖游戏软件赚到的钱就这样全部蒸发了。



3. 算出修复所有bug的价值

我们的Fog Creek软件公司目前规模还很小（但野心不小），因而开发团队也兼职接听技术支持电话。每天接电话的时间累计大概有1小时，根据我们的咨询服务收费水平，成本大概是每年7.5万美元。我们确信如果修复所有已知bug，接电话的时间能降到每天15分钟。

经过粗略的估计，修复所有bug的净现值大概是15万美元，相当于62天工作的价值，也就是说如果能用低于62人天来修复所有bug，就值得去做。

通过使用FogBUGZ内置的估算工具，我们算出需要20个人天（2个人工作2周）来修复所有bug，相当于“投资”4.8万美元，得到15万美元的回报，投资回报率很高，这还仅是考虑了技术支持方面节省的开支。（请注意，还可以把程序员的薪资成本和系统开销替换为咨询费用，得到相同的1：3的投资回报率，因为前者抵消了后者）。

刚才的计算甚至还没有考虑产品质量提升带来的好处，我们现在就来看看。7月份我们的测试服务器上捕获了来自17个不同用户的55个崩溃记录。可以推断至少有一个用户因此认为FogBUGZ不好，决定不买我们的产品。（对此

我没有确切的统计数据。)考虑到各种情况,客户流失带来的损失折现大概在7000美元到10万美元之间。(如果给予足够重视的话,不难找出准确的数字。)

下一个问题。产品的bug变少了,可以因此提高售价吗?如果可以,这让修复bug的价值又提升了一大截。我猜想在极端情况下,软件产品的bug数量会影响其售价,但是要从商业软件行业找出一个实际的例子,恐怕很难。



请不要断章取义!

难免有些人看了本文之后得出一些错误的结论,比如,称认为不应该修复bug。其实我认为大部分bug还是值得修复的,有着很明确的投资收益。但是,可能存在比修复每一个bug更具经济价值的事情。如果你面临两个抉择,一个是修复OS/2平台的bug,另一个是开发新功能,然后卖给通用电气公司2万份软件副本,那么只能对OS/2用户说抱歉了。如果你蠢到认为修复OS/2的bug比开发通用电气需要的功能还重要,那竞争对手可不会这么想,你迟早会被淘汰出局。

话又说回来,我骨子里是个乐观主义者,我相信提高软件质量所带来的潜在价值是难以用金钱衡量的。你的员工会更加自豪。你不用担心用户会把软件光盘扔进微波炉、用斧子剁碎然后寄回给你。因此,我倾向于在软件质量方面精益求精(我们修复了FogBUGZ中所有已知的bug,而不只是会引起崩溃的那些)并且引以为豪。我们坚信,通过彻底清理测试服务器上发现的所有bug,我们软件的品质将变得坚如磐石。

12

五个世界

2002年5月6日，星期一

我的第一份全职工作是在一家面包工厂。那里有两个巨大的车间：一个用来烤面包，另一个用来给面包装箱。在第一个车间，人们整天和面团打交道。面团黏在机器里，黏在手上、头发上、鞋子上，每个人都随身携带一把画家用的小铲子，用来清理塞住机器的面团。在第二个车间，人们整天都和面包屑打交道。面包屑粘在机器里、手上、头发上，每个人都随身携带一把小刷子。我觉得每份工作都有它的难处，都有某种特有的、令人恼火的小问题。谢天谢地我没有在刮胡刀工厂找到工作。

对于编程这份工作，我们也有自己的难处，而且第一个“车间”的难处和第二个“车间”的难处不同。但是出于某种原因，即使你看遍了市面上讲软件开发的所有书籍，可能也不清楚这些“车间”的区别，更别说里面分别有哪些难处了。

离开面包工厂之后，我的大部分时间都在编写数百万人使用的软件。第一次读到极限编程理论主张把“客户”请进开发团队的时候，我就想，哪些客户？在朱诺公司（美国的一家主流互联网服务提供商）的电子邮件客户端团队工作时，我们有数百万的客户，包括很多可爱的老奶奶，而我们的团队建设材料里并没有怎么提到她们。

我也曾为大型集团的开发者们做过开发咨询。当我口若悬河地讲述自己在微软的时候如何如何，他们就像看火星人一样看着我：乔尔，这里不是微软。我们没有上千万的客户，只有坐在后边办公室的吉姆。

我终于明白了一个道理。软件开发领域可以分为许多不同的世界，不同世界里适用的规则也不一样，这个显而易见的事实被大家忽视了太久。没错，你

我都是软件开发者，但如果对于彼此世界的差异视而不见，就会产生各种无意义的比较和纷争，例如.NET与Java，敏捷方法与软件工程。讲述UML建模的文章绝不会透露这种方法完全不适用于硬件驱动开发。一篇发表在Usenet上的帖子可能会写“.NET程序所需的20MB运行时根本不成问题”，但它不敢指出一个明显的事实：如果代码要在只有32KB存储空间的传呼机芯片上运行，20MB的运行时就当然是个大问题！

我认为软件开发领域可以分为五个世界：

- (1) 盒装软件
- (2) 内部软件
- (3) 嵌入式软件
- (4) 游戏
- (5) 用过即抛的代码



1. 盒装软件

这里我所说的盒装软件是指那种被大量用户自由使用的软件。它可能包裹着塑料封套，在CompUSA中出售；也可能通过互联网供下载使用。它的发行方式可能是以盈利为目的的商业版，也可能是共享软件、开源软件、GNU软件等，但不变的特征是，这种软件会被上百万甚至上千万的用户安装和使用。它甚至可以是一个有上百万或上千万访问量的网站。

盒装软件面临着来自两个方面的特殊问题。

- 由于用户很多，众口难调，用户界面需要比普通软件更易于使用，才能取得成功。盒装软件的开发者需要把一半的精力用在进一步提高软件的易用性上，让德卢斯的利奥叔叔都能够轻松上手。
- 由于在很多电脑上运行，所以代码需要兼容各种型号和运行环境的电脑。开发者在另一半时间里，都在费劲地辨认DLL细小而繁多的版本号。

上周有人通过邮件向我报告了CityDesk的一个bug，它只在使用波兰语键盘时出现，出错原因和使用所谓的AltGr键输入特殊字符有关。我们的测试环境涵盖Windows 95、95OSR2、98、98SE、Me、NT 4.0、Win 2000和Win XP，并且测试了安装IE 5.01、5.5和6.0的系统。我们测试过美国英语、西班牙语、

法语、希伯来语和汉语的 Windows，但是还没来得及测试波兰语。

盒装软件的变种主要有三种：开源软件、基于网络的应用和咨询软件。

开源软件

开源软件在开发过程中通常没有资金支持，这使得开发动力有了很大的变化，例如不够“有意思”的功能通常得不到开发。马修·托马斯生动地指出，这对软件的可用性造成了一定的损害^①。开源软件的开发团队成员通常在地理上比较分散，这使得团队沟通的质量很难得到保证。开源软件世界里，人们很少能够面对面地交流，或是聚集在一块白板前，边讨论边画方框和箭头。需要使用白板这种交流形式来讨论和决策的设计在开源项目中通常都做得很差。因此，地理上较分散的团队更适合去做已有软件功能的复制品，因为需要设计的地方很少，甚至没有。由于绝大部分开源软件都是在“野生”环境中使用的，所以属于盒装软件。

基于网络的应用

基于网络的应用包括 Hotmail、eBay，甚至某个以内容为主的网站，等等。这类网络应用需要让用户轻松上手，并且要兼容多种浏览器。虽然大部分用户都使用“主流”浏览器，但是最喜欢抱怨的用户使用的都是老古董，这些早该进历史博物馆的浏览器连符合 W3C 标准的 HTML 页面都无法正确显示，即使你把杰弗瑞·泽尔德曼^②本人编进程序里都不行，所以，网页开发者经常要花大量的时间搜索“网景浏览器 4.72 CSS 字体 bug”之类的信息。尽管开发者对于“部署”环境（即数据中心的服务器）拥有某种程度上的控制权，但是还需要适配一系列不同的网络浏览器，并且服务于庞大的用户群体，因此我把这类软件看作盒装软件的一个变种。

咨询软件

咨询软件是盒装软件的另一个变种，它需要很多的定制化和部署才能工作，你需要聘请一支价格不菲的专家顾问团队才能安装并运行它。CRM（客户关

① 马修·托马斯，“为什么自由软件通常都不好用”，2002 年，参见他 6 年后的修改版 http://stc-access.org/wp-content/uploads/matthew_paul_thomas-blog_archive-why_free_software_has_poor_usability_and_how_to_improve_it.pdf。

② 杰弗瑞·泽尔德曼是互联网标准的先驱和制定者之一，详情参阅 <http://baike.baidu.com/view/3461152.htm>。——译者注

系管理)和 CMS(内容管理系统)软件通常属于这个类别。它们给人一种实际上没干什么事情的感觉,不过是一个冠冕堂皇的幌子,好让咨询“部队”有理由进驻现场,收取每小时 300 美元的高额费用。尽管咨询软件通常会伪装成盒装软件,但实施的高额成本让它更接近于内部软件。

2. 内部软件

内部软件只需要在一家公司的电脑上运行,这大大降低了其开发难度。开发者可以任意对运行环境进行各种假定。你可以假定目标运行环境安装了特定版本的 Internet Explorer、微软 Office 或 Windows。如果需要一张图表,就调用 Excel 来画,因为可以确保自己部门的每一台电脑都预装了 Excel。(但是如果在盒装软件里这么做,你会损失一半的潜在用户。)

内部软件开发者通常不会承认,但事实上,可用性在内部软件开发中的优先级是很低的,因为只有少数人需要使用这类软件,而且他们别无选择,只能硬着头皮使用公司制定的软件。很多时候,这类软件的 bug 要多于盒装世界中的同类。相对而言,开发速度则显得更加重要。因为这类开发活动只对一个公司产生价值,所以能够正当利用的开发资源变得很匮乏。微软可以斥资 2 亿美元来打造一个操作系统,平摊到每个用户的成本大概只有 98 美元。但是当查坦努加铁路公司想要开发一个火车交易平台的时候,投资的数额就必须合理,因为它只对一个公司有用。为了获得合理的投资回报率,开发内部软件的投入不可能像开发盒装软件的投入那样高。事实上,内部软件和盒装软件的一个关键差异就是,在达到某个临界点之后,再投入更多资金去提升内部软件的稳定性或易用性,其边际效用会有明显的降低;而对于盒装软件,不断追求软件的稳定性和易用性永远都符合经济定律,都是对公司核心竞争力的提升。所以令人叹息的是,很多内部软件虽然能完美地实现既定的功能,但是质量却非常差。年轻、有热情的开发者经常会被管理者泼冷水,劝他们不要再完善某个软件,因为“已经足够好了”。这通常会让他们十分沮丧。

3. 嵌入式软件

嵌入式软件的特征是在芯片上运行,在绝大多数情况下都不可能升级。(即

使在技术上可行，也不会进行升级。相信我，没人会为微波炉下载并刷新 EPROM 软件的版本。) 这是一个完全不同的世界。它对软件的质量有着更高的要求，因为没有改正的机会。你要面对的可能是比普通桌面电脑处理器运行速度低很多的处理器，需要花费大量的时间做性能优化。代码追求的是运行速度，而不是优雅。能用的输入和输出设备可能有限。我上周租了一辆车，车载的 GPS 系统的输入输出设备非常难用。你曾经试过在一个没有键盘的设备上输入地址吗？你曾经试过用一个只有电子表那么大的地图来导航吗？我有点跑题了。

4. 游戏

游戏是一种特殊的软件，原因有二。第一，游戏开发的经济圈是典型的赢家通吃。某些游戏非常热门，剩下的大部分游戏都是失败品。如果你想通过游戏软件来赚钱，必须要对这一点有清醒的认识，不要把鸡蛋放到一个篮子里，只有这样，依靠一两个热门游戏所赚的钱才有可能超过所有失败游戏所赔的钱。这种状况更像是拍电影，而不是开发软件。

开发游戏的另一大问题是，只能有一个版本。玩家如果在《毁灭公爵 3D》里通关，就不会为了 bug 修复和得到新武器而升级到 3.1D 版本。除了个别的骨灰级玩家，一般人在游戏通关之后，都没有兴趣再玩一遍。所以游戏和嵌入式软件一样，对软件的质量有很高的要求，同时面临的经济压力也要求软件尽可能一次到位。与此相比，盒装软件开发虽然总是在费劲地辨认各种 DLL 的版本，但是他们的世界有一个好处：即使产品 1.0 版本不够符合用户需要、卖得不好，但 2.0 版本还有希望。

5. 用过即抛的代码

第五个世界是用过即抛的代码，它涵盖了不属于上述四个世界的所有代码。程序员为了得到某些信息而编写这些临时性的代码，一旦得到了想要的信息就不再需要这些代码了。例如，写一小段 Shell 脚本，把一个输入文件转换

成适应其他目的的格式，并且是一次性的。用过即抛的代码没有特别令人烦恼的共性问题，然而某种商业法则决定了，临时性的代码有一种很快转变成内部软件的趋势。某个 MBA 看到之后，会喃喃自语：“这里面隐藏着巨大的商机……”然后，又一个咨询软件公司的“企业级解决方案”诞生了。

了解你的世界

你可能会说，我现在知道了开发者生活在不同的世界里，但这又有什么意义呢？其实，有一条很重要的启示。如果你读到由全职软件开发大师/顾问所撰写的关于编程方法论的书籍，相信我，里面谈论的全是内部、企业级软件的开发，而不是盒装软件、嵌入式软件，更不可能是游戏。为什么？因为只有大公司才会聘用这些软件大师。大公司出钱赞助了这些书的写作和出版发行。（相信我，id 软件这种游戏开发公司可不会聘用爱德·尤顿（Ed Yourdon）来谈论结构化数据分析。）

最近我学习了许多关于极限编程（Extreme Programming, XP）的知识。在开发许多类型的项目时，与各种编程学派教导的僵化、无聊、与现实脱节的“开发流程”相比，使用极限编程和其他敏捷方法的体验，都像是一阵新鲜的微风。毫无疑问，如果真正用起来，测试驱动开发之类的技术都是很棒的。但问题就在于，你必须真正把它们用起来。极限编程理论中有很多疏漏之处，这可能会对你的开发造成影响，也可能没有影响。比如，还没有人能把测试驱动的开发方法应用于图形用户界面（GUI）开发。我曾经用测试驱动的开发方法完整搭建了一个网络应用，效果很好，但是这个应用服务器所做的只是把字符串变来变去而已。我还是想不到一种方法，把自动化测试应用于任何包含图形用户界面的产品开发中。我能想到的最好结合点就是在后台进行模拟用户行为的自动化测试，但是如果图形化用户界面实现了拖拽释放（drag-n-drop）之类的动作，就完全没办法使用自动化测试了。

极限编程理论还假定重构是一件很容易的事，因为你事先编写了各种测试用例，确保整个过程中功能不被破坏。这对于相对独立的项目可能是成立的，尤其是那些就地开发、不需要和外界交互的项目。但是我们公司的产品 FogBUGZ 就不是这样，我不可能随心所欲地修改它的数据格式，因为很多客户已经基于这些格式进行了二次开发，如果我修改了数据格式，他们以后

肯定不愿意升级了。拿不到升级的费用，我就养不起手下的程序员，整个公司就垮了。要考虑到和其他系统的交互，必须仔细地进行分析和设计。所以我一直在探索如何让敏捷方法适用于我所在的盒装软件世界，尽管已经意识到二者根本不属于一个世界，无法实现完美的契合。

虽然软件开发中的大部分知识和经验都是普适的，但不能认为它们适用于所有类型的项目。当别人告诉你一种方法论的时候，留心想想它是否适合你做的工作。想想知识传授者来自哪个软件开发世界。话虽如此，无论你来自何方，大家通过交流，都能从彼此身上学到很多的东西。每个世界都有自己特殊的问题。如果你在企业软件领域，使用 Java 开发服务端应用，请不要因为不需要应付 DLL 地狱问题而沾沾自喜，并因此觉得 Java 是一门更加优秀的语言。你之所以不用应付 DLL 地狱，是因为你的程序只在一个盒子里运行。如果你是一名游戏开发者，就请不要去抱怨字节码的执行速度太慢。它本来就和你工作完全无关。它是为那些内部软件的开发者准备的，大概只有四个注册账号会“冲击”他们的软件。对于彼此的面团/面包屑问题，请大家多一份尊重和理解，让 Usenet 成为一个更加文明的场所。

13

纸上原型设计

2003年5月16日，星期五

许多年以前，Excel 团队想知道，允许用户使用鼠标拖拽单元格是不是个好主意。他们让几个实习生用当时最先进的 Visual Basic 1.0 “快速搭建一个原型”，然后对这个功能点做可用性测试。原型的搭建工作持续了整个夏天，因为如果不复制 Excel 的很多功能，就无法保证可用性测试的真实可靠性。

可用性测试的结果如何？没错，这的确是一个很好的功能特性！负责开发的程序员只用了大概一周的时间，就完全实现了单元格拖拽功能。这件事被当作是笑话来讲，因为搭建原型本来的意义在于“节约时间”。

一年以后，另一个微软的绝密团队为全新的用户界面搭建了一个完整的原型，这次使用的是当时最先进的产品 Asymetrix Toolbook（老天爷，真不敢相信这个产品现在还活着）。原型搭建花费了大概一年时间。基于原型做出的真实产品是 Microsoft Bob^①，它是软件世界里的 PCjr。又白搭了一个原型。

我基本上放弃了做软件原型。如果原型能够模拟实现产品全部的功能，它其实就是那个产品了。如果原型无法实现产品全部的功能，那它其实没什么用。幸运的是，有一个更好的办法：纸上原型设计，它漂亮地解决了普通原型带来的困境，同时还解决了一个名为冰山^②的问题。

纸上原型设计非常简单，就是用铅笔在纸上画出用户界面。画得越潦草越好。把这幅设计图展示给几个人看，询问他们如何用这个界面完成某项任务。在这个过程中，设计者和使用者完全不会因为字体、颜色等问题而产生争执，因为完全没有字体和颜色，谁也不会指望铅笔线画得有多美。并且，你显然

① 参阅<http://toastytech.com/guis/bob2.html>。

② 参阅第 25 章。

并没有投入过多的精力，人们就不会因为怕伤害你的感情而在提出意见时有所保留。

纸上原型设计比所有软件工具都要便宜得多，而且甚至可以用来做真正的可用性测试，你需要的只是一叠纸、一个好用的橡皮擦、一支铅笔和一副剪刀。当被试者说明自己想做的操作时（“点这里！”），你可以随时换纸或对页面做出修改。需要测试引导式对话框？你只需要在不同的纸上分别画出每一页的界面，然后准备一些索引卡片，写上各种情况下的错误信息。

关于纸上原型设计的更多内容，请参阅卡罗琳·斯奈德的著作^①，里面详细介绍了这种方法。这本书对于所有的用户界面设计者都是必要的参考，写得非常好。

^① 卡罗琳·斯奈德，《纸上原型设计：一种快速和轻松设计完善用户界面的方法》。

别被架构航天员忽悠了

2001年4月21日，星期六

伟大的思想家在思考问题时，首先会透过现象看到内在的模式。思想家看到人们互相传送由文字处理软件生成的文件，看到人们互相传送电子表格文件，然后归纳出这些现象都属于一个具有普遍性的模式：传送文件。在这第一层抽象的基础上，他们又做了一层抽象：人们发送文件，浏览器也“发送”网页请求。仔细想想，其实一段代码调用某个对象的方法，也可以看作是给对象发送了一条消息！同样的模式又出现了！这些动作全都是发送操作，所以我们聪明的思想家引入了一个崭新的、更高级、更普遍的抽象概念，称为消息传递（messaging）。可惜这个概念经过了太多层次的抽象，变得玄之又玄，没人听得懂了。

如果把事物上升到过高的抽象层次，人类思维就无法理解了，就好像一个人如果处于离地面很远的高空，大脑会因为空气稀薄而缺氧一样。而有时候，片面追求思维抽象层次的思想家们，会构想出古怪的、令人困惑的宏观宇宙图景，这些理论虽然看上去很好，但其实言之无物，根本没有任何的实际意义。

我把这类思想家称为架构航天员。很难让他们屈尊去写代码或设计程序，因为他们的脑子时刻在思考架构问题。之所以称呼他们为航天员，是因为他们完全置身于大气层以外，我真不知道他们怎么呼吸。通常大型公司里总是供养着许多这类高学历、低产能、对公司毫无贡献可言的所谓思想家。

举个最近遇到的例子。一个典型的架构航天员听到一个简单的事实，比如“Napster 是一个以点对点（peer-to-peer）形式提供音乐下载服务的应用”，他会自动抽取其中和架构有关的细节，在完全脱离实际场景的前提下，得出“Napster 的成功秘诀在于点对点传输架构”的结论。而实际上这个应用的精髓在于，用户只要输入一首歌曲的名称就能立即欣赏到这首歌曲。

架构航天员整天谈论的都是如何把点对点的理念应用于这个问题、那个问题以及其他各种问题。于是一夜之间，不知从哪儿冒出了点对点会议、点对点风险投资基金、甚至点对点反冲等概念，连毫无判断力的商业记者都开始互相抄袭抨击点对点的文章，起的标题既耸人听闻却又空洞无物：“点对点已死！”

架构航天员会这样思考：“你能想象一个类似 Napster 的软件，用它不只可以下载歌曲，还可以下载任何类型的文件吗？”然后他们会构建出他们心目中更加通用的应用，比如 Groove，却忽视了那个看似没什么技术含量的功能特性，即输入歌名就能听歌，而这才是最核心的用户需求。就像人们常说的“捡了芝麻丢西瓜”。假如 Napster 不使用点对点传输架构也能让用户输入歌名就能听歌，它还是一样的会受到追捧。

架构航天员还喜欢做这种事：发明某种架构，然后声称它能解决某些问题。Java、XML、Soap、XML-RPC、HailStorm、.NET、Jini，天哪，技术架构的更新也太快了，我都跟不上了。而这些架构都是最近 12 个月才开始出现的！

我并没有说这些架构有什么不好，绝对没有。它们是很好的架构。但让我看不惯的是有些人把这些架构吹捧得天花乱坠，进行铺天盖地的宣传。记得微软发布的 .NET 白皮书^①吗？

Windows .NET 是 Windows 桌面平台的次世代版本，它能最大限度地发挥用户的生产力、创造力，支撑用户的管理、娱乐以及其他更多的活动，而且设计以人为本，让用户自主掌控数字化生活。

这是 9 个月前写的。上个月，微软公布了有关 HailStorm 的消息。那份白皮书^②里写道：

人们目前无法控制身边的科技……而 HailStorm 致力于让你生活中的科技齐心协力为你工作，由你掌控。

太好了，也就是说我公寓里的高科技卤素灯再也不会忽明忽暗了。

① 微软于 2000 年 6 月 22 日发布的“Microsoft .NET: Realizing the Next Generation Internet”，目前微软官网上已无法查看。

② 微软于 2001 年 3 月发布的“A Quick Introduction to HailStorm”。参见 [http://msdn.microsoft.com/en-us/library/bb263932\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/bb263932(v=vs.85).aspx)。

自我吹嘘的并不只有微软。Sun 公司的 Jini 白皮书^①里这样写道：

这三个事实（成为新的系统管理员、计算机无处不在、计算机无所不在）将组合发力，在全球推动使用计算机的方式回归正途——消弭计算机的物理边界，通过让计算机无所不在，通过改善使用计算机易用性，让使用计算机就像是把 DVD 光盘推入家庭影院设备中一样简单。

我实在不愿意提起 Java 布道师乔治·吉尔德向世界散播的福音^②：

这是科技史的重大突破……

当架构航天员向你展开强大的忽悠攻势时，有着很明确的信号：长篇累牍的夸夸其谈，史诗、乌托邦般的豪言壮语，自吹自擂，完全架空于现实世界的痴言妄语。然而人们偏就信这一套！商业媒体简直太疯狂了！

为什么人们会对平凡无奇的架构更替感到印象深刻呢？上面提到的那些架构说白了，不过是为远程进程调用（RPC）增加了一种连接方式，或是使用了一种新的虚拟机。这些架构本身可能设计得很好，推广之后会让广大的开发者受益匪浅，但是它们不应该，我再说一遍，不应该被人们捧上天去，一谈起架构革命就像是在宣扬救世主再临、骑着白驴进入耶路撒冷，或是世界和平、宇宙大同。不，微软，计算机不可能突然学会读心术，完全按人们心中的意愿工作，即使世界上的每个人都注册了 Passport 账户也不行。不，Sun，做企业销售数据分析不可能“像是把 DVD 光盘推进家庭影院设备一样简单”。

请注意，架构师们致力于探索他们能解决什么样的问题，而不太关心解决那些有实际用途的问题。SOAP+WSDL 可能是最新的技术宠儿，但是它并不能真正帮你解决任何一个使用以前的技术无法解决的、值得去解决的问题。“使用某某架构，分布式服务将会涅槃”之类的鬼话，架构航天员们已经向我们许诺过好多次了，这么多年来的说辞都差不多，变换的只有许诺的对象，DCOM、JavaBeans、OSF DCE、CORBA 等都是这样。

① Sun.com 于 1999 年发布的“Why Jini Technology Now?”参见 <http://systems.cs.colorado.edu/~grunwald/MobileComputing/Papers/why-jini-now.pdf>。

② 乔治·吉尔德的“The Coming Software Shift”，1995 年 8 月 28 日发表于《福布斯 ASAP》杂志。

现在他们又在说，可以使用 XML 作为线上数据的标准传输格式了。太好了。但我对这个消息完全提不起兴趣，就好像得知附近的超市使用卡车从仓库运货一样。（哈欠）呵呵。架构航天员们，请告诉我一件用以前的技术做不到、现在能做到的事情。如果你们讲不出来的话，那还是请还是待在大气层外，不要再浪费我的时间了。

15

干扰射击

2002年1月6日，星期日

有时候我什么都干不成。

我来到办公室，这转转那看看，每10秒查看一次邮件，浏览网页，做些在线付账单之类的无脑任务。但就是不能进入到写代码的状态中。

这种效率低下的状况，通常会持续一两天。但在我的开发生涯中，曾经有一次连续数周都感觉一事无成。俗话叫不在状态、心猿意马、魂不守舍。

心情的波动人人都有，有的比较平缓，有的则比较强烈，甚至让人无法工作。效率低下的状况有时候确实和低落的心情有关联。

这让我想起，研究者们称人们根本管不住自己的嘴，任何节食的尝试都无法长久坚持下去，体重总会像溜溜球一样反弹到正常水平。也许作为软件开发者的我，真的没法掌控自己的工作效率，只能任凭它忽高忽低，如果平均下来能输出足够多行的代码，保住我的饭碗，那就谢天谢地了。

从第一份开发工作开始，我就发现平均下来，每天只有2~3个小时是在高效率地写代码，这很让我抓狂。我在微软做暑假实习的时候，另一个实习生告诉我他每天真正的工作时间是中午12点到下午5点。5个小时，还不算午餐时间。就算是这样，他的产出还是远高于平均，同事们都爱死他了。同样的情况也发生在我身上。看上去其他人时刻都在拼命干活，而我凭借每天火力全开地工作2~3个小时，还是成为团队中最能干的成员之一。这也许是人件^①和极限编程（XP）的布道者都强调杜绝加班，每周只工作40小时的原因，因为他们确信这么做并不会减少团队的产出。

然而让我担心的并不是“只”工作2个小时的日子，而是那些一事无成的日子。

① 汤姆·德马科，蒂姆·李斯特，《人件（第2版）》。

对此我想了好多。我回忆整个生涯中最高产的时期，也许是在微软刚刚把我的工位换到一个崭新办公室的时候，那个办公室有很大的观景窗，外面是一个樱花盛开的石头院落，非常漂亮。所有事情都一拍即合。我在几个月里写出了 Excel Basic 的详细设计书——用堆积如山的纸，巨细靡遗地阐述了一个庞大的对象模型以及编程环境。我那时像上了发条一样工作，根本停不下来。期间我不得不去波士顿参加 MacWorld 大会，我坐在哈佛商学院里漂亮的天台上，还在用笔记本电脑撰写 Window 类的说明文档。

一旦你进入状态，保持运转就很容易了。我的日常是这样的：(1) 上班，(2) 收邮件、看网页，(3) 决定先吃午饭再开始工作也来得及，(4) 吃完回来，(5) 收邮件、看网页，(6) 下决心我必须开始工作了，(7) 收邮件、看网页，(8) 再次下决心，我真的要开始了，(9) 打开该死的编辑器，然后，(10) 开始马不停蹄地写代码，回过神来已经晚上 7:30 了。

在第 8 步和第 9 步之间的某处似乎有个 bug，我不总是能跨越这道鸿沟。对我来说，开始干活是唯一困难的事情。静止的物体总会保持静止。仿佛我脑中有个特别沉重的东西，很难带着它把速度提起来，而一旦全速前进，继续运动下去就很轻松了。就像一架自行车装上了越野所需的沉重装备^①，要让这样的自行车运转起来是很费力的，但难以置信的是，一旦骑出速度，感觉就和骑没有装备的自行车几乎一样了。

也许这就是高效率的关键：只管开始就好了。也许结对编程起作用的原因，在于当你和同伴一起编程时，会迫使彼此开始工作。

我曾经在以色列做过空降兵，那时听一位将军讲关于战术的心得。他说，步兵战中只要记住一条：进行干扰射击，也就是一边开火，一边靠近敌人。开火迫使对手躲避，这样他就不能朝你开火。（这也是士兵们喊“掩护我”的意思，即“向敌人开火让他躲避，这样我在跑过街道的时候就不会挨枪子儿”。这很管用。）不断移动让你步步为营，而离敌人越近，射击就越有可能打中目标。如果你不动，主动权就在敌人手里，那可不好。如果你不开火，敌人就会抢先朝你开火，让你寸步难行。

我一直记得这条战术，并发现从飞机混战到大规模的海军火力部署，几乎每种军事战术都建立在“干扰射击”这个思想的基础上。又过了 15 年，我才意识到这个原则也是生活中的做事法则。你必须每天前进一点点。即使你的

^① 参阅<http://joel.spolsky.com/biketrip/>。

代码很烂，bug 很多，没人想用，都没关系。如果你不断进取，不断写代码、改 bug，那么时间就会站在你这一边。当竞争者朝你开火的时候要留神。他们是不是想用干扰射击让你疲于应付，无法前行？

以微软推出的历代数据连接技术为例。ODBC、RDO、DAO、ADO、OLEDB、和刚推出的 ADO.NET。这么多的新技术，有什么非用不可的理由吗？还是说微软负责开发数据连接服务的团队特别无能，每过一年就不得不推倒重来（还真有可能这是这样）？我认为这更有可能是一种火力压制，迫使竞争者们用全部精力来跟上微软的节拍，移植现有的功能，从而没有时间开发新的功能。让我们仔细审视软件行业的现状。凡是做得风生水起的公司，都是对大公司技术依赖最少的，这些公司不需要把所有开发周期浪费在紧跟大公司并且多次重写同样的功能上，或是浪费在修复一些只出现在特定平台（如 Windows XP）的 bug 上。而那些做得不好的公司却时刻在求神问卜，关注并猜测微软技术的未来走向。有些人看到 .NET 出来了很焦虑，决心为 .NET 重写整个架构，因为他们认为非这样不可。其实是微软在朝你开火，而且是干扰射击，这样他们可以前进而你不能。没错你被耍了，老兄。你想支持 HailStorm 吗？SOAP 呢？还是 RDF？你做出支持这些技术的决定是因为客户需要呢，还是只是由于干扰射击的存在而被迫做出的回应呢？大公司的销售团队也会玩这一招。他们跟客户说：“不买我们的产品没关系，谁家的产品最好就买谁的。但是你一定要买支持（XML/SOAP/CDE/J2EE）的产品，不然你就落伍了^①。”然后小公司的销售们就会听到客户方的 CTO 们鹦鹉学舌：“你们有 J2EE 吗？”于是这些公司就把时间浪费在构建 J2EE 上，即使这并不能让软件卖得更好，与此同时，也失去了做出自己特色的机会。这叫做勾选功能，开发它的目的只是为了在旁边打个勾说你有这个功能，但是没有人会用到它、需要它。这就是干扰射击的影响。

对于我们这样的小公司，干扰射击意味着两件事情：一是一定要抓紧时间，把开发的主动权掌握在自己手里^②；二是必须每天进步。这样迟早会胜出。我昨天做的全部工作就是对 FogBUGZ 的配色做出小小的改进。挺好，我们的软件总在进步。如果我们的软件每天都变得越来越好，就会有越来越多的客户购买我们的产品，这才是我最关心的事情。在公司的规模膨胀到 Oracle 那个级别之前，我们不需要考虑采取某些“高端”的战术。现阶段我们要做的，就是每天早晨来上班，然后想办法说服自己，尽快打开编辑器。

① 戴夫·温纳，“The Micro Channel Architecture”，2001 年 7 月 6 日发表于 DaveNet。

② 参阅第 36 章。

16

软件开发是一门手艺活

2003 年 12 月 1 日，星期一

开发软件并不像是工厂在制造产品。20 世纪 80 年代，大家惊闻日本在建造“软件工厂”，这些工厂能通过流水线作业批量生产高质量的软件。而这在当时的技术水平下是天方夜谭，即使是现在的技术也还达不到。把一群程序员塞进车间，让他们站成一排，并不能有效地减少 bug 数量。

如果写代码不同于流水线作业，那它更像什么呢？有人提出，软件开发是一门手艺活。当然，这种说法也并不总是成立的，因为不知你怎么想，反正我觉得那个询问你该如何索引帮助文件的 Windows 对话框，从形状到样式，无论从哪个角度看，都和人们常说的“工艺品”差远了。

写代码不是工业生产活动，也不总是一门手艺活（但它可以是），它其实更像是一种设计活动。设计是一个很模糊的概念，其特征在于用较慢的成本增长换取较快的价值增长。《纽约时报杂志》对 iPod 赞不绝口^①，称赞苹果公司是世间少有的知道如何用优秀设计来增加价值的公司。



图片由苹果公司提供，详情参见：<http://www.apple.com/pr/photos/ipod/03ipod.html>

① 鲍勃·沃克，“The Guts of a New Machine”，《纽约时报杂志》，2003 年 11 月 30 日，第 78 页，第一栏。

但是，关于设计这个话题我已经讲了很多，现在想谈谈软件开发的工艺品特性，包括它是什么以及如何辨识。

我想和大家分享一段我正在重写的代码，它实现的是 CityDesk 3.0 的文件导入功能。（插播一条广告：CityDesk 是我的公司发布的一款方便易用的内容管理产品。）

功能规格说明非常简单。用户通过标准的对话框选择一个文件，然后程序将该文件复制到 CityDesk 数据库中。

结果这个例子很好地说明了，有的时候“写好最终 1% 的代码要耗去 90% 的时间”。程序的第一版设计草案是这样的：

- (1) 打开文件；
- (2) 把全部内容装进一个庞大的字节数组；
- (3) 把字节数组储存为数据表中的一个条目。

这个程序在通常情况下运行得很顺畅。对于一般大小的文件，程序基本上一眨眼的工夫就跑完了。但是它存在几个小 bug，容我一一列举。

其中一个较大 bug 出现在压力测试中，当时我试着把一个 120MB 的文件拖入 CityDesk。一般情况下，人们不会在网站上放这么大的文件。这在实际中几乎不可能出现，但也不是没有可能。程序运行了大概一分钟才结束，其间没有任何视觉反馈，界面呈现假死状态，点击哪里都没有反应。这明显不够理想。

从用户界面设计的角度来考虑，我应该在程序进行耗时操作的时候显示某种进度条，以及一个取消按钮。更理想的情况是让文件在 CityDesk 的后台进行复制，不影响用户进行其他的操作。要实现这一功能，比较容易想到的方案有三种：

- (1) 只开一个线程，每隔一段时间检查是否有输入事件；
- (2) 再开一个线程，仔细地做好线程同步；
- (3) 再开一个进程，不需要很仔细地做好进程同步。

我的经验是，方法 1 的效果总是不够理想。当程序正在进行文件复制操作时，很难保证其他所有代码能安全运行。埃里克·雷蒙德的论述让我相信，多线程的方案不如多进程的方案^①，而且根据我多年的经验，多线程编程增加了

① 参阅 <http://www.faqs.org/docs/artu/ch07s03.html#id2923889>。

太多额外的复杂度，会引入不少这种方案所独有的、危险的海森堡 bug^①。3号方案看起来更好，尤其是考虑到我们使用的数据库支持多用户操作，并不介意多进程的并发访问。所以过了这个感恩节假期，我准备用方案3来实现这个功能。

但是退一步想，我们从最初简单的读取文件、保存到数据库出发，最终采取了一个复杂得多的方案：开一个子进程，让它读取文件并保存到数据库；子进程还要有进度条和取消按钮；并且要建立一种机制，让子进程能在文件保存完毕时通知父进程，即时显示处理完毕的文件。另外要做的一部分工作是通过命令行把参数传递到子进程；还要处理窗口焦点的行为，让它符合使用者的心理预期；以及特殊情况的处理，比如用户在程序正在复制文件的时候关闭电脑。我猜最后要写的代码量是原来的10倍，只是为了优雅地处理大文件，而这些工作的成果只有大概1%的用户能看到。

当然了，还有一种程序员会认为，子进程的方案还不如原来的方案。它被“过度设计”了，增加了太多额外的代码行，这也导致程序更有可能出现错误。这叫过犹不及。他们会说，这种做法是典型的 Windows 思维，Windows 在他们眼中并不是一个设计得很高明的操作系统。他们对设计进度条的必要性嗤之以鼻。因为在 UNIX 下，只要敲击 Ctrl+Z 然后用“ls -l”看看文件体积有没有增加就知道程序是否还在运行了！

这个故事告诉我们，修复出现概率为1%的问题，有时候需要花500%的精力。这种情形并不是软件开发活动所独有的，我这样说是因为我管理过一些建筑施工工程。上周，我们的建筑承包商最终完成了对 Fog Creek 新办公楼^②的装修完善工作，包括给正门装上崭新的蓝色丙烯面板，并在门的边缘每隔20厘米包上一条铝片。仔细看下面这张照片，你会发现每扇门的四周都有铝条。在两扇门接触的部分，两条铝片呈竖向对齐排列。在图上可能看得不明显，但这两条铝片中央的固定螺丝并没有完全对齐，大约相差2毫米。木工师傅事先做了仔细测算，但是在安装铝条的时候，门是放在地上的。等到门挂到轴上才发现，坏了，螺丝很明显地没有对齐。

① 详情参阅<http://c2.com/cgi/wiki?HeisenBug>。

② 参阅<http://joelonsoftware.com/articles/BionicOffice.html>。



这种情况绝非个例，我们的办公室里有很多螺丝不是完全对齐的。问题在于打完孔后再矫正位置的成本是很高的。正确的位置只偏那么几毫米，因此不好重新打孔。如果追求完美的话，只能换掉整扇门，太不值了。这个例子再一次阐释了有时候修复出现概率为 1% 的缺陷需要 500% 的努力，也同样揭示了为什么这个世界上有那么多工艺品的完美程度停留在 99%，而不是 100%。（我们的建筑师总是喋喋不休，声称亚利桑那州的某些豪宅中，每颗螺丝都是完全对齐的。）

大部分人都认为，软件具有与之类似的特性，让它带有工艺品的气息。当一群真正具有工匠情怀的人们构建一个软件产品时，他们会想办法对齐所有的螺丝。也就是说对于一些极其罕见的情况，软件也能处理得很漂亮。但要做到这一点，开发者势必将大部分的精力投入到对各种罕见情况的正确处理上，而不是集中精力让主要的业务逻辑正常运转。这也就意味着开发者要用 500% 的精力去处理好出现概率为 1% 的情况。

要达到一定的工艺水平，需要付出巨额的开发成本。软件行业里唯一值得付出巨额的开发成本的情形，是开发面向海量用户的软件。很遗憾，譬如保险公司的内部人力资源系统，就永远不会达到这种完美的工艺水平，因为没有足够的用户数量来摊薄成本。然而对于开发零售软件的公司，这种不断追求并完善软件品质的做法正是用户想看到的，因为那将会为公司提供长期的竞争优势，所以请广大用户耐心等待，我们会慢慢来，因为慢工才能出细活。



计算机科学的三个错误观念

2000年8月22日，星期二

这么说可能会引起某些人的反感，但是坦白地讲，我认为计算机科学中有三个观念是错误的，而人们对此认识不足。所以我写了这篇文章，想给大家提个醒。

不确定是否还有其他的错误观念，但是以下三大观念曾经让我误入歧途：

- (1) 搜索系统的难点在于找到足够多的结果；
- (2) 开了抗锯齿效果的文字看起来更舒服；
- (3) 网络软件应该做到让网络资源的使用就像本地资源一样。

我对这三个观念的评价是：

- (1) 错；
- (2) 错；
- (3) 错！

下面容我一一道来。

搜索

在搜索领域，学术界痴迷于对这类问题的研究：“用户搜索‘土豆’，但是文档里用的词是‘马铃薯’怎么办？”

很多搜索方面的学术研究都基于“词根抽取”（stemming）这个技术，也就

是把用户的查询词做各种词法变形，这样一来，用户搜索“searching”也会找出带有“searched”和“sought”的文档。

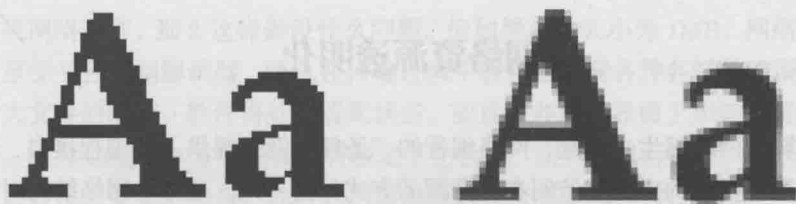
以 Altavista 为代表的大型互联网引擎刚出现时，经常自我标榜能返回海量的查询结果。在 Altavista 里搜索“Joel on Software”返回了 1 033 555 个页面。这显然根本没有任何用处。目前互联网的页面总数接近 10 亿。即使 Altavista 把这个数字缩减到 1 百万，对我又有什么用呢？

搜索领域真正要解决的问题是如何对结果进行排序。当然这不能怪计算机科学家，在互联网出现之前，没有人意识到排序对于索引海量语料的重要性。

但是还是有人注意到了。谷歌公司的创始人拉里·佩奇和谢尔盖·布林意识到，对页面进行正确排序远比抓取尽可能多的页面重要得多。他们发明的 PageRank 算法^①效果就很好，能对海量的页面进行排序，把用户想找的页面排进前十位。实际实验一下，在谷歌上搜索“Joel on Software”，第一个结果就是我的博客。而在 Altavista 里，我翻了 5 页都没有找到自己的博客，然后就放弃了。

抗锯齿技术

1972 年，抗锯齿技术诞生于麻省理工学院（MIT）的架构机器组，这个小组就是后来著名的多媒体实验室的前身。该技术的思路是，如果彩色显示器的分辨率有限，可以在边缘填充各种明度的灰色来制造高分辨率的“错觉”。下面是一张抗锯齿效果的放大图：



请注意左边的文字边缘非常锐利，而右边的文字边缘似乎被模糊了。如果你眯起眼睛或者离远一点看，就会发现正常的文字会因为分辨率有限而呈现出

^① 参阅<http://www.google.com/technology/index.html>。

很奇怪的“棱角”，而抗锯齿的文字则看起来更平滑舒服。

这就是大家如此热衷于抗锯齿技术的原因。现在到处都在用抗锯齿。微软 Windows 系统甚至提供了一个选项，开启后系统的所有文字都将呈抗锯齿效果。

这样做有什么问题呢？问题在于，一段文字在经过抗锯齿处理后，看起来就是模模糊糊的。我对此毫无办法，但事实就是如此。请比较以下两段文字：

Antialiasing was invented way back in 1972 at the Architecture Machine Group of MIT, which was later incorporated into the famous Media Lab.	Antialiasing was invented way back in 1972 at the Architecture Machine Group of MIT, which was later incorporated into the famous Media Lab.
---	---

左边的段落没有做抗锯齿处理，而右边的段落用 Corel 公司的 PHOTO-PAINT 进行了抗锯齿处理。坦白地说，经过抗锯齿的文字确实变难看了。

终于，微软字体技术组^①意识到了这个问题。他们创造出了许多很棒的字体，比如 Georgia 和 Verdana，这些字体“在计算机屏幕上具有很高的易读性”。他们转变思路，不再试图把为高分辨率而设计的字体压缩到很粗的像素栅格里，而是把像素栅格作为“给定条件”，然后设计出这个条件下最适合的字体。然而还是有人没有注意到这一点：微软阅读器开发组为彩色液晶屏幕设计了一种称为 ClearType 的抗锯齿技术，但是对不起，即使是在彩色液晶屏幕上，文字看起来还是很模糊^②。

在图形学专家读者们愤怒回应之前，我想补充说明一下，抗锯齿还是一项很好的技术，它适用两个场合：标题和产品标识（logo），这些场合对整体样式美观的需求高于文字的可读性。此外，对于图片，抗锯齿也是一种缩小照片尺寸的好方法。

网络资源透明化

从计算机网络诞生之日起，网络编程的“圣杯”就是提供一种编程接口，让访问远程资源的方式和访问本地资源的方式保持一致。这样，网络结构对于

① 关于微软字体，详情参阅<http://www.microsoft.com/truetype/default.asp>。

② “First ClearType screens posted”发表于 Microsoft.com，2000 年 1 月 26 日。参阅 <http://www.microsoft.com/typography/links/news.aspx?NID=1135>。写作本文时，ClearType 已经成为微软 Windows XP 系统的标准特征，并得到了极大的改进。

使用者就“透明”了。

实现网络资源透明化的一个实例是著名的远程程序调用技术（Remote Procedure Call, RPC）^①，借助它，你可以用和访问本地资源同样的方式来访问网络中其他计算机的资源。另一个例子是微软推出的分布式 COM（DCOM）^②，它构建于 RPC 之上，能让你用和获取本地 COM 对象类似的方式来获取位于另一台计算机上的 COM 对象。

听起来它很合理，是吧？

错。

访问另一台计算机的资源和访问本地资源之间有三大区别：

- (1) 可用性
- (2) 延迟
- (3) 可靠性

访问网络上的另一台机器时，该机器很可能暂时无法使用，或者网络连接暂时无法使用。受网络速度的制约，一个请求很可能要过一段时间才能有回应，因为你可能在通过速率为 28.8kbit/s 的调制解调器连接网络。还有可能是另一台机器崩溃了，或者通信时网断了（有只猫绊到了电话线）。

要开发出使用网络的可靠软件，必须考虑到这些特殊情形。如果编程接口把所有细节都隐藏起来了，你很可能会开发出质量很差的软件。

举一个简单的例子，假设我想开发一个把文件从一台计算机复制到另一台计算机的软件。在 Windows 平台上，老式的“透明”方法是调用 CopyFile 方法，用 UNC 名称来作为传递的文件名，比如 \\SERVER\SHARE\Filename。

如果网络正常，那么这样做没什么问题。但如果文件大小为 1MB，网络要通过至少一台调制解调器，那么在传输过程中就可能出现各种各样的错误。传输大文件的时候，软件将处于假死状态。而且没办法在界面上添加进度指示条，因为 CopyFile 方法的设计者假定了它会很“快”地执行完毕，所以没有预留开发进度条的接口。此外，网络掉线时无法实现断点续传功能。

① 关于 RPC，详情参阅 http://searchwebservices.techtarget.com/sDefinition/0,,sid26_gci214272,00.html。

② 关于 DCOM，详情参阅 <http://www.microsoft.com/com/tech/dcom.asp>。

在实际使用中，通过网络传输文件通常会使用设计得更好的 API，比如 `FtpOpenFile` 以及与之相关的一系列函数。虽然这与复制本地文件不同，而且使用起来更加困难，但是这个函数在设计上考虑到了网络编程和本地编程的差异，提供了制作进度条所需要的接口，在网络无法使用或突然掉线时会优雅地返回传输失败的信息，并且在读写模式上是异步的。

综上所述：下次要是有人来向你兜售一种编程开发产品，声称它能让你用同样的方式访问网络和本地资源，那就赶快跑吧。

18

二元文化主义

2003 年 12 月 14 日，星期日

今天，Windows 和 UNIX 的大部分功能都是相近的。它们都支持同样的编程范式，包括命令行、图形用户界面（GUI）和网络服务器；它们的系统资源配备如出一辙，有几乎相同的文件系统、内存管理方式、套接字（socket）、进程和线程。两个操作系统提供的核心服务集差异很小，基本不会对你能够开发的应用类型产生限制。

两个操作系统的差异主要体现在文化上。虽然人都要吃东西，但在地球那边，人们用木筷子吃生鱼片和米饭，而地球这边的人们吃的是面包夹牛肉饼。文化存在差异并不意味着美国人的胃消化不了寿司或者日本人的胃消化不了汉堡包，也不意味着没有多少美国人吃寿司或者没有多少日本人吃汉堡。它意味着在一个美国人抵达东京、刚下飞机的那一刻，一种非常强烈的陌生感油然而生。无论怎么强调人们内心深处都一样，都会相爱、工作、歌唱和死亡，也改变不了文化差异存在的事实：美国人和日本人永远不会对彼此的卫生间布置感到舒服。

UNIX 和 Windows 程序员的文化差异表现在哪些方面呢？抛开细枝末节和微妙的差异不讲，基本上归结于一点：UNIX 文化看重编程对其他程序员的价值，而 Windows 文化看重编程对非程序员的价值。

诚然，这样说把事情过分简化了，但我确实觉得，两种文化的差异集中体现在这个问题上：写代码是为了其他程序员还是最终用户？相比之下，其他的差异都显得不那么重要。

埃里克·雷蒙德是一位颇受公众争议的人物，他最近写了一部关于 UNIX 编程的大部头，名叫《UNIX 编程的艺术》，书中深度剖析了他熟悉的 UNIX 编

程文化^①。你可以买一本纸质书来看；如果你由于不喜欢雷蒙德的政治立场^②而不想付钱，还可以免费在线阅读^③，这样就能保证付出辛苦劳动的作者不会收到一分钱^④。

来看一个小例子。UNIX 编程文化对能从命令行调用的程序有很高的评价，这些程序从命令行接收参数来控制所有的程序行为，并且输出格式规整、机器可读的纯文本。程序员们非常青睐这样的程序，因为很容易把它与其他程序进行集成，组成一个更大的软件系统。例如，UNIX 文化中有一个核心价值观，雷蒙德称之为“沉默是金”，即如果程序按人们的意图正常执行完毕，那么它不应该输出任何信息^⑤。你可能输入了一行包含 300 个字符的命令去调用一个程序，它的功能也许是创建一个文件系统，也许是编译并安装一个复杂的软件，甚至有可能是发射载人登月火箭。UNIX 文化认为，如果程序执行成功，那么最好什么都不输出。命令提示符重现的时候，用户就能推断出程序执行完毕，并且一切正常。

这是 UNIX 文化中很重要的价值观，因为你的程序是写给其他程序员用的。对此，雷蒙德写道：“程序们不喜欢和喋喋不休的家伙一起玩耍。”与之形成鲜明对比的是，在 Windows 文化中，程序是写给玛芝姑姑用的，而玛芝姑姑有理由认为，如果一个程序什么都不输出，那么根本无法分辨执行成功还是失败，也不知道程序是否正确理解了用户的请求。

基于类似的原因，UNIX 文化崇尚程序的界面保持文本化^⑥。UNIX 程序员不太喜欢图形化的用户界面，即使有图形界面，也是构建在文本化程序之上的，更像是一种事后的补救措施。他们也不喜欢二进制文件格式，原因是文本化的界面更容易用编程的方式进行交互。相比之下，图形界面如果不提供一些诸如脚本语言之类的内建机制，是很难用编程的方式交互的。这再一次体现了 UNIX 文化更倾向于编写对其他程序员友好的代码，而在 Windows 编程中则很少见到这一诉求。

这并不意味着所有 UNIX 程序都是只为其他程序员而编写的。完全不是这样。

① 埃里克·雷蒙德，《UNIX 编程的艺术》。

② 埃里克·雷蒙德，“Draft for an Anti-Idiotarian Manifesto（第 2 版）”，发表于 Armed and Dangerous，2002 年 10 月 16 日。

③ 参阅 <http://www.faqs.org/docs/artu/>。

④ 埃里克·雷蒙德，“Eric S. Raymond—Surprised By Wealth”，发表于 Linux Today，1999 年 12 月 10 日。参阅 http://linuxtoday.com/news_story.php3?itsn=1999-12-10-001-05-NW-LF。

⑤ 埃里克·雷蒙德，《UNIX 编程的艺术》。

⑥ 埃里克·雷蒙德，《UNIX 编程的艺术》。

但是文化决定了人们更看重对其他程序员有用的代码，这对某些程序的体验造成了某种程度的影响。

假设你分别找来一个 UNIX 程序员和一个 Windows 程序员，让他们开发同一个面向最终用户的应用。UNIX 程序员会先开发出一个命令行或文本化的程序内核，然后在此基础上添加一个对内核进行操作的图形化界面。这样，程序的所有主要操作都能由其他程序员使用命令行来调用，并且所有的输出都可以用文本方式进行获取和解析。Windows 程序员则会先写图形化界面，然后在此基础上添加一个脚本语言，作为操作图形界面的自动化接口。这种做法，在一个 99.999% 的用户都对编程一无所知并且毫无兴趣的文化里，还是很恰当的。

在 Windows 程序员中有一个很重要的群体，他们的工作主要是为其他的程序员服务，那就是微软内部的 Windows 产品组。他们开发软件的方式是先写 C 语言可调用的 API 来实现功能，然后写图形化界面来调用 API。用户能用 Windows 用户界面做的事情，程序员使用任何常用语言通过编程接口都可以做到。例如，微软 Internet Explorer 软件本身只有 89KB，这个小程序封装了几十个灵活而强大的 Windows 功能组件，这些组件本来只能由对 Windows 编程极为精通的程序员使用。遗憾的是，由于外部开发者看不到这些组件的源代码，他们只能以微软组件开发者们预见并且允许的方式来使用这些组件，有时候这远远不够。有时候程序出了 bug，即使犯错的通常是 API 的调用者，但是在没有 API 的源代码的情况下，调试 bug 是很困难、甚至不可能的。UNIX 开发者崇尚开放源代码文化，这使得开发环境变得相对友好^①。任何一个 Windows 开发者都会向你诉苦：他曾经有过花 4 天追踪一个 bug 的经历，最后发现出错原因是误以为 LocalSize 返回的内存大小等于之前用 LocalAlloc 申请的内存大小，如果能看到库的源代码，10 分钟之内就能找出错误的原因并修复 bug。雷蒙德编了一个有趣的故事来阐述这一点，相信任何用过二进制代码库的人都会有共鸣^②。

这引起了一些意识形态上的争论。有人说 UNIX 更好，因为你可以调试库里的代码。有人说 Windows 更好，因为玛芝姑姑能确认她把邮件发出去了。其实没有孰优孰劣，只是两种文化的价值观不同：在 UNIX 世界里，对其他程序员友好压倒一切，而在 Windows 世界里，对玛芝姑姑友好压倒一切。

① 埃里克·雷蒙德，《UNIX 编程的艺术》。

② 埃里克·雷蒙德，《UNIX 编程的艺术》。

我们来看另一个文化差异。雷蒙德说：“经典的 UNIX 文档简明扼要，且十分完备……这种文风要求读者有一定的主动性，能从文中提到的内容推理出明显但没有明说的结论，并且对推理过程的正确性有足够的自信。你必须字斟句酌，因为文档不会把同一个意思叙述两遍。”天啊，我觉得他是在教唆年轻一代的程序员写出更晦涩难懂的 man (manual, 手册) 页面。

要让最终用户能看懂，这种文档肯定是不行的。雷蒙德也许会说最终用户“总把事情想得过分简单、懒得弄清细节”，但是 Windows 文化中的程序员懂得最终用户不喜欢啃手册^①，即使他们阅读你写的文档，也只会阅读尽可能少的部分。因此，你必须重复讲解一些事情。事实上，优秀的 Windows 帮助文件的标志就是，任何一个主题都可以抽出来单独阅读并且被普通读者理解，并且不需要事先对其他帮助主题有任何的了解。

核心价值观的差异是如何形成的呢？这就涉及雷蒙德这本书的另一个优点：追溯了 UNIX 的历史和演进历程，让新一代的程序员也能了解该文化自 1969 年以来的发展脉络。UNIX 刚刚诞生并且开始形成其文化价值取向的时候，计算机软件的最最终用户群体还不存在。当年的计算机非常昂贵，CPU 时间也很昂贵，学习计算机就等同于学习如何编程。UNIX 文化在当年的环境中很自然地形成了对其他程序员友好的核心价值观。与之形成对比的是，Windows 作为一种软件产品，它的诞生只是为了一件事：尽可能地多卖出副本、多赚钱。卖出数不清的副本。Windows 团队明确地提出了“让每个家庭、每个桌面上都有一台计算机”的目标，并据此制定开发计划、产生核心价值观。为了让每个家庭的每个桌面上都有一台计算机，唯一的办法是开发出不会编程的用户也能轻松使用的软件，因此 Windows 文化非常强调软件的易用性。而程序员只占广大用户的极少一部分，Windows 自然不会优先考虑他们的使用感受。

巨大的文化隔阂导致 UNIX 一直没能真正在桌面市场占据一席之地。玛芝姑姑不会使用 UNIX，即使程序员多次努力对 UNIX 前端进行漂亮的包装，玛芝姑姑还是不会用 UNIX，这完全是因为这些程序员受 UNIX 文化的影响太深。举个例子，UNIX 奉行一个叫做把策略和机制分离开来^②的价值观，它最早来自 X 窗口系统^③的设计者。这直接导致了 UNIX 设计者在用户界面上的

① Joel Spolsky, 《面向程序员的用户界面设计》，参阅<http://www.joelonsoftware.com/uibook/chapters/fog0000000062.html>。

② 埃里克·雷蒙德，《UNIX 编程的艺术》。

③ 参阅<http://www.x.org/>。

巨大分歧；对于系统桌面该是什么样的，从来没有达成一致意见，而且大家还觉得这样挺好，这符合 UNIX 文化所鼓励的多样性，但是对于玛芝姑姑来说这样很不好，一个程序中实现剪切粘贴功能的界面竟然和另一个程序中的不一样。自 UNIX 的设计者尝试写出好用的用户界面已经过去了 20 年，而现状是，最大的 Linux 供应商的 CEO 仍然告诉大家，家庭用户只要用 Windows 就好了^①。我听说经济学家称法国不可能出现硅谷，因为法国的文化向来对于失败者有着沉重的惩罚，导致企业家们不敢冒险。也许 Linux 也是这样：受文化价值取向的阻碍，它可能永远也不会成为一个桌面操作系统。OS X 的成功也佐证了这一点：苹果公司之所以能开发出一个玛芝姑姑也会用的 UNIX，恰恰是因为苹果的工程师和产品经理也是最终用户文化的忠实拥趸（我之前把这种文化称为“Windows 式”是不恰当的，其实它最早起源于苹果公司）。他们的想法不同于 UNIX 所秉承的以程序员为中心的文化。他们甚至把 UNIX 核心的目录名都改成了通俗的英语名称（简直是大逆不道），例如“bin”改成了“application”，“lib”改成了“library”。

雷蒙德在书中也尝试比较了 UNIX 和其他操作系统的异同，但我认为这部分内容与这本优秀著作中的其他部分相比写得不是很好，因为他真的不懂自己在讲什么。他每次讲 Windows 的时候就露出马脚，我认为他对 Windows 编程的了解来自于看报纸，而不是实际的 Windows 编程经验。这没什么，毕竟他不是 Windows 程序员，情有可原。作为只对一种文化有很深研究的人，他通常会忽视其中的普适价值观（伤害老太太和程序崩溃总是坏事）以及该文化特有的价值观（生吃鱼片和命令行参数的吸引力因人而异）之间的差别。

只熟悉一种文化的程序员太多了，他们就像从来没离开过明尼苏达州圣保罗市的美国孩子一样，无法区分文化特有的价值观以及全人类共享的价值观。我见过太多的 UNIX 程序员对 Windows 编程嗤之以鼻，认为 Windows 离经叛道、愚蠢不堪。雷蒙德也经常落入俗套，对其他文化的价值观断章取义、妄下评判。相比之下，Windows 程序员则没有这么多偏见，他们总体上就事论事、不会过分理想化。至少，Windows 程序员会承认他们文化中的不合理因素，并且做出务实的解释：“如果你想把文字处理软件卖给很多人，就必须让软件能在他们的计算机上运行，如果这意味着我们必须使用丑陋的注册表、而不能使用优雅的~/rc 文件来存储软件的设置参数，那我们也认了。”与 Windows 世界里更加务实的人们（“随你怎么想吧，我不过是要混口饭吃”）

^① Munir Kotadia, “Red Hat recommends Windows for customers”, ZDNet UK, 2003 年 11 月 4 日。参阅 <http://news.zdnet.co.uk/software/linuxunix/0,39020390,39117575,00.htm>。

相比，UNIX 世界里则充斥着各种自以为是、自命不凡的“文化传播大使”，以及信奉“/.”如神明的宗派主义者。UNIX 文化给人一种四面楚歌的感觉，其发展一直限制于服务器和发烧友市场，而难以进入主流桌面市场。一份源自于自卑的自傲感是《UNIX 编程艺术》这本书最大的败笔，但其实也无伤大雅：综合来看，这本书中有太多关于编程的有趣而深刻的隽语、太多值得借鉴的普适理念，为此我愿意忍受书中极少出现的不切实际的抱怨。进一步讲，我推荐所有程序员都阅读这本书，无论你来自哪种编程文化、使用什么开发平台、有何种开发目的，这本书都值得一读，因为书中有太多的理念是放之四海而皆准的。例如雷蒙德指出 CSV 格式不如 `/etc/passwd` 文件中的格式，他是想说明 UNIX 比 Windows 更优越^①，但你猜怎么着？我觉得他说得对：`/etc/passwd` 的格式确实比 CSV 更容易解析。如果你读了他的书，你就会知道原因，并且因此成为一个更好的程序员。

① 埃里克·雷蒙德，《UNIX 编程的艺术》。



自动从用户那里收集崩溃报告

2003 年 4 月 18 日，星期五

我曾经为美国一个规模很大的 ISP（网络服务提供商）开发过客户端。使用我们软件的用户真的有数百万之多。即使最罕见的 bug，也有可能对几十万人造成影响。然而在按下按钮、交付最终版代码的那一刻，我却感到无比自信。我记得对父亲说：“测试版的效果很好，昨天我们的软件在整个北美只有 12 次崩溃记录。”

12 次？为什么不是 13 次或 14 次？

没错，就是 12 次。拥有这种对任何地方发生的崩溃信息都了若指掌的自信，对于开发出在各种未知环境下都能保证高质量运行的软件来说至关重要。处于消费者软件行业，你不可能指望用户主动报告崩溃信息，因为许多用户都不懂技术，而且大部分很忙，没有时间来提交一份包含足够信息的崩溃报告。所以有必要把整个崩溃报告的收集过程完全自动化。

身为一家软件公司的创始人，我可以自豪地说，公司的所有代码几乎都能把错误自动报告给开发团队。即使是供内部使用的软件，也会在发生崩溃时通知开发者。在从用户机器上收集崩溃报告方面，我们这些年来掌握了很多方法和经验教训，接下来就和大家分享。



收集数据

假设你的软件崩溃了。几乎所有编程环境都在某个核心的位置提供了从崩溃中恢复的方法（在编程开发环境的侧边栏通常有一些相关的代码实例）。利

用这类方法，我们在软件崩溃的那一刻，暂时不关闭程序，而是先弹出一个简明扼要的对话框：



只问一个问题，能增加用户报告崩溃的概率

填写电子邮件地址是可选的，以减轻用户对隐私的顾虑

为保证报告的隐私性和匿名性，需要让用户看到将要发送的具体信息

我从多年编程经验中学到，你问人们的问题越多，他们就越不愿意回答。所以我们尽可能用最少的问题来获取有助于诊断问题的信息，其他的重要信息大部分都可以自动获取，例如使用的操作系统版本、内存大小、等等。

一定要强调提交崩溃报告的行为和内容是匿名的，用户的隐私将会得到妥善的保护。有些人在工作中常与一些保密数据打交道，如果他们怀疑你会上传敏感信息，就不会提交了。为此，需要提供一个链接，用户通过点击就能查看发送报告的具体内容。为避免留下不良印象，最好用恰当的方式，把自动收集并上传了哪些信息也告知用户。

下一个问题是，应该收集哪些信息来帮助开发者找到崩溃的原因？收集程序的设计者常常倾向于眉毛胡子一把抓，抓取所有能找到的系统信息，包括用户操作系统上每一个 DLL 和 COM 控件的版本信息，甚至是完整的内存导出数据（也叫核心导出数据）。

我做了很多年的开发者，一直不知道该如何分析核心导出数据，而且发现收集这类数据其实没有必要。只要知道崩溃代码的具体位置信息，就足以定位并修复几乎所有的崩溃错误。对于剩下的罕见情形，可以通过电子邮件联系相关用户来获取更多有助于调试的信息。

减少收集的信息量,有助于加快崩溃报告的处理过程,减少用户等待的时间。只检测所有 DLL 和 COM 控件的版本,就要花上一段时间,还要考虑通过调制解调器上传数据的时间,况且这些信息很少对调试错误有帮助。即使你发现某种错误和某个版本的微软系统 DLL 有关联,又能做什么呢?只能修改自己的代码,想办法绕过出错的地方。

自动收集的数据

- 产品的具体版本
- 操作系统和 Internet Explorer 的版本 (Windows 的许多功能其实都是由 Internet Explorer 及相关组件提供的,所以它的版本信息对调试图形化界面程序很重要。)
- 发生崩溃的代码所处的文件名和行号
- 字符串格式的错误信息
- 错误类型的独特编号
- 用户对当时所做事情的描述
- 用户的电子邮件地址

如果用户提供了电子邮件地址,开发者在需要更多信息的时候,可以在 bug 跟踪系统中点“回复”按钮,立刻向用户发送邮件询问。bug 跟踪系统会自动把来往邮件的副本保存在 bug 报告里。

打电话回家

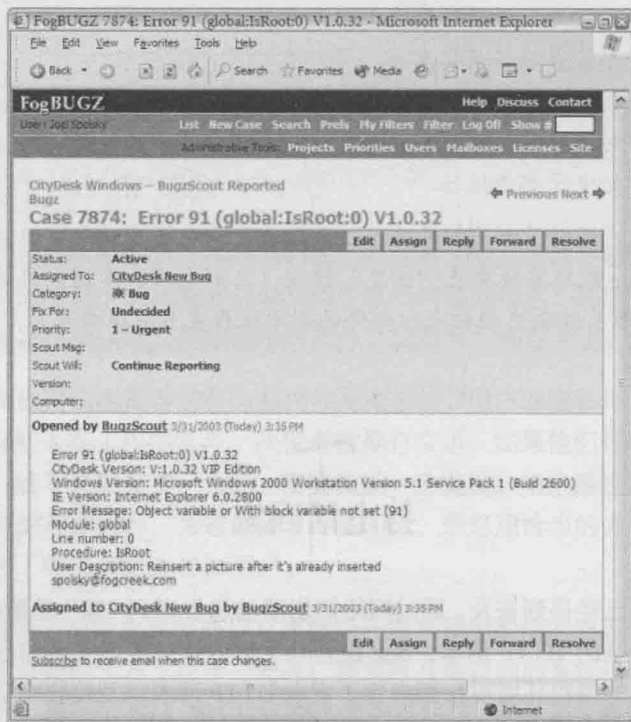
如今互联网已经得到普及,通过网络把崩溃信息发送到“家”是最好的选择。通过发送标准的 HTTP 请求,bug 报告几乎可以绕过客户安装的任何防火墙类型。所有的编程环境现在都内建了发送 HTTP 请求以及接收响应的库。例如在 Windows 上,WININET 库里有一个函数,使用 Internet Explorer 的网络传输代码来发送 HTTP 请求并接收响应。使用这些函数的好处是,即使用户配置了浏览器的代理服务器(这在受防火墙保护的系统里很常见),WININET 调用也会自动使用这些代理服务器,完全不需要你做任何事。

对于 HTTP 请求的响应部分,我们的服务器返回的是一个很短的 XML 文件,表明报告已经收到,并且包含一条显示给用户的消息。如果你的应用是基于

网络的，还有一种更简单的方法：显示一个网页，用上面的表单把错误信息提交到服务器。

对于特定类型的应用，你可以采取另一种策略：不马上发送崩溃报告，而是把它写到一个文件或是注册表里，然后等用户下一次打开程序的时候再发送。我把这种技术称作延迟传输。虽然报告发送晚了一点，但好处是即使错误严重到无法发送 bug 报告，你最终还是能得到这份报告。

所有的崩溃报告通过一个统一的 URL，抵达 Fog Creek 软件公司的一台对公服务器上。我们的 bug 跟踪数据库通过这个 URL 接收 bug 报告。事实上，该数据库唯一的公开访问接口就是这个 URL，其他部分都被隔离开来，这样人们只能提交报告，不能借此进入数据库。下图是 FogBUGZ 的一个 bug 报告：



我们可以通过设置，让 bug 报告自动发送到开发团队的某一个成员，但是最近我们不想打扰程序员，而是设置了一个虚拟的角色，叫做“CityDesk New Bug”。每隔一段时间，当我们想筛查崩溃报告的时候，会搜索指派给这个虚拟角色的所有 bug，然后逐一决定是否修复。最后把那些决定修复的 bug 指派给某个真正的程序员。

鉴别重复的崩溃信息

自动收集的崩溃信息有一个重要特点，就是同样的崩溃可能会在不同的人身上发生很多次，而作为开发者，你不想把每一条重复的崩溃信息都作为新 bug 输入数据库。我们为每条崩溃数据构建一个包含关键元素的特征字符串，将来如果出现相同特征字符串的数据，它就被自动附加到原有数据条目的一个字段里，而不是生成一个新的数据条目。

在生成字符串的时候，我们想让不同用户报告的相同崩溃信息有相同的特征字符串。经过一些试验，我们发现最好的方法是在字符串中包含错误号、文件名、函数名、行号，以及软件版本信息。上图中，这个特征字符串是“Error 91 (global:IsRoot:0) V1.0.32”，意思是编号为 91 的错误发生在名为 global.bas 的文件里，在函数 IsRoot 的第 0 行，软件的主版本是 1.0，构建版本是 32。我们有一个规定，构建版本号为偶数的版本是内部构建版本，构建版本号为奇数的版本是面向用户的发行版本。这样，我一眼就能看出这个崩溃事件发生在开发者而不是用户身上。

特征字符串的设计大有学问。过去，我们把崩溃错误信息的整个文本都加到字符串里，但是很快发现同一个错误信息会被翻译成各种语言。这样一来，每一个崩溃报告都会出现各种版本，有英语、德语、西班牙语、法语和一些我不认识的语言！后来，我们把错误信息放在崩溃报告的正文部分，用作特征字符串的标题只包含错误编号（在各种语言中都一样），从而解决了这个问题，减少了重复记录的数量。

我们还特殊设计了 bug 报告数据的标题字段以便检索。我们的标题格式是“文件名:函数名:行号”（注意冒号分隔符），这样通过搜索“:函数名:”就可以搜索某个函数的 bug。同理，我们在版本号前面加上了字母 V，你可以搜索 V1、V1.0 或 V1.0.32。如果没有前面的 V，查询版本号为 1 的时候会返回所有标题中包含数字 1 的 bug 报告。

当某个 bug 被正确识别的时候，我们把其“继续报告”标记改为“停止报告”，这样具有同样特征字符串的崩溃记录就会被忽略。我们甚至可以设置一个文本消息，发送给将来每个报告同一种 bug 的用户。通过这些消息可以向用户提示一些暂时性的解决措施，例如：“嘿！下次在保存之前别忘了先拍拍脑

袋，揉揉肚子！”

重复的报告还有一个常见的原因，就是处理崩溃的代码崩溃了。这并不一定说明处理崩溃的代码本身有问题，可能只是因为真正导致崩溃的问题非常严重，让任何代码都无法正常运行。



调试

在 beta 测试阶段，我们收到崩溃报告后会马上调查。而当产品即将交付、开发者们忙于完善下一个关键的发行版本时，他们通常没有时间来查看每个崩溃报告。这个时期，可能会等几个月再来查看哪些问题最为常见，然后只调查这些问题。这样做的缺点是无法和用户沟通发生崩溃的细节：毕竟事情已经过去了几个月。但我发现，如果同样的崩溃发生了好几次，总会有某个用户提供足够的信息，让我在开发环境中重现 bug。实际上，如果知道是哪行代码发生了崩溃，很少会出现对原因毫无头绪的情况，即使不知道 bug 的重现方法也没关系。有一次我甚至用算术和应用逻辑学方法，一步一步倒推出了 bug 的重现方法：嗯，如果这里崩溃了，那这个值肯定是负的；如果它是负的，说明这个 IF 语句返回的是 true。这样一直回溯到什么样的输入数值组合会导致崩溃，然后就能找到出错的原因。



优先级排序

自动化的崩溃报告系统建好以后，崩溃报告会源源不断地流入。所以拥有良好的优先级排序技能显得尤为重要，能够决定哪些 bug 最值得修复，然后忽略剩下的 bug。

经过调查和总结，我发现了一些可能永远都不会去修复的 bug，例如：

- 用户的电脑因为内存问题而崩溃；
- 用户试图手动修改数据文件；
- 用户的系统太老，比如 Windows 95，已经病入膏肓；

□ 用户在内存极低的情况下运行程序，可能磁盘和内存都已经满了。

有时候很难找到导致崩溃的原因，特别是那些只发生过一次的崩溃。事实上，我们采取的策略是忽略只发生过一次的崩溃记录。我们有更重要的事情要做。如果一个 bug 在实际使用中都难以重现，那在开发环境中重现的可能性也很小。

这种方法适用于你的工作吗？

经过以上对崩溃报告的介绍，你觉得它适用于你的工作吗？这个问题的答案在某种程度上取决于产品的用户数量。

如果你开发的是供内部使用的企业软件，自动的崩溃报告可能不会给你带来多少价值。企业软件项目的开发通常都是为了解决某个特定的问题，一旦问题得以解决，就不必花更多的钱在项目上了。软件有可能一周崩溃一次，虽然有点烦人，但是不值得花几千美元去让开发者修复导致崩溃的问题。也就是说，修复 bug 虽然是一种良好的习惯，但却无利可图。过于理想化的软件开发者在开发内部软件项目的时候通常会很失望，因为一旦代码“足够好”，经理就会让他们停止开发。当软件已经可以解决某个商业问题的时候，无论再怎样去提高它的质量，投资回报率都是 0。但是话说回来，由于许多企业软件开发团队都没有配备质量分析和测试人员，自动崩溃检测可能是开发者获得 bug 报告的唯一途径。

另一方面，如果你在开发薄膜软件或是消费者软件，过硬的质量必然为公司带来巨大的竞争优势。你的软件会运行在非常不友好的环境中。作为产品的运行环境，消费者的个人电脑（PC）确实很糟糕。没有两台计算机是完全一样的，硬件和软件的配置各不相同。销售商经常会在卖出的电脑里预装你能想到的每一种垃圾软件。很多消费者也经常傻呵呵地下载并安装各种看上去光鲜体面的软件，其中一些会聪明地把自己注入到其他正在运行软件的进程空间里。再有，绝大多数家庭用户都缺乏计算机的知识，无法维护系统的良好运转。所以，在这种充满敌意的环境里，自动的崩溃报告是唯一能让软件质量水平满足市场需要的方法。所以，构建一个健壮的系统，来接收、报告、分类并跟踪来自真实使用场景的崩溃信息，将会让客户更加满意，并且通过对交付产品质量的提升，带来数倍于开发成本的价值。

错误处理

Visual Basic 代码中的错误处理

由于一般 Visual Basic 6.0 事件驱动的程序有太多的入口点（每个事件处理程序都有一个入口点），为了捕获出现在所有地方的崩溃，唯一的办法是在每一个函数中都加入错误处理函数。我们的所有函数都如下所示：

```
Private Sub cmd_Click()  
On Error GoTo ERROR_cmd_Click  
  
    ' 真正的代码处理逻辑 '  
Exit Sub  
  
ERROR_cmd_Click:  
    HandleError "moduleName", "cmd_Click"  
End Sub
```

手写这样的代码太痛苦了，幸好有一个叫 ErrorAssist 的工具（<http://www.errorassist.com>），可以自动为所有的函数添加捕获异常的代码。在所有情况下，我们都调用一个叫 HandleError 的全局函数，这个函数会显示定制的崩溃对话框。

Windows API 代码中的错误处理

Win32 API 包含了一个叫做结构化异常处理（structured exception handling）的机制。发生崩溃时，Windows 先查找并执行当前代码处的未处理异常句柄（unhandled exception handler）函数。如果没找到这个函数，就会弹出用户体验非常差的“该程序执行了一次非法操作”对话框。

要为软件配置一套未处理异常句柄函数，你需要做两件事。第一，实现一个函数，其接口形式为：

```
LONG UnhandledExceptionFilter(  
    STRUC _EXCEPTION_POINTERS *ExceptionInfo  
);
```


第二，调用 `SetUnhandledExceptionFilter` 函数，将一个指针传入 `UnhandledExceptionFilter` 函数。

还有一种方法能用 C++ 代码实现相同的功能，就是把程序的主入口点包裹在一个 `_try/_catch` 语句中。注意前面的两个下划线，它会告知编译器处理来自更底层的错误，例如解引用空指针，而不管那些你自己写的、用 `try` 抛出并用 `catch` 捕获的各种 C++ 异常。

要获得更多详细信息，请在 Windows 平台 SDK 或 MSDN 上搜索如何使用结构化异常处理。

ASP 应用中的错误处理

微软的互联网服务管理器（Internet Services Manager）允许你定制一个错误处理页面，HTML 或 ASP 均可，用来处理没有被 `On Error` 语句处理的脚本错误。特别是当一个 ASP（Active Server Pages）应用崩溃或是出现一个任意种类的未处理错误时，页面就被重定向到这个错误处理页上，错误号为“500;100”。我们所有的应用都有一个 ASP 页面来捕获 500;100 错误。

这个处理页包含如下的 VBScript 关键代码：

```
Set objASPErrors = Server.GetLastError
```

这样，名为 `objASPErrors` 的对象中就存储了许多和刚才错误有关的信息，包括出错的文件名以及行号。

安装了 IIS 的计算机都附带处理 ASP 错误的样例代码：请在 `\windows\help\iishelp\common` 目录中寻找一个叫 `500-100.asp` 的文件。这个文件只是简单地把 ASP 错误的细节呈现给最终用户。以 `500-100.asp` 文件为基础，你可以构造定制的消息页面，然后在表单（form）的隐藏元素里包含与崩溃相关的数据。这个表单应该有一个 `action` 属性，能把所有的错误信息提交到另一个网页。

第二部分

管理开发者

20 非正式面试指南

21 奖励有害论

22 不设专职测试人员的5条（错误）理由

23 任务切换有害论

24 永远不要做的事情（第一部分）

25 揭开冰山之谜

26 抽象必漏定律

27 帕默斯顿勋爵谈编程

28 评估机能失调

20

非正式面试指南

2004年6月4日，星期五

给大家出道面试题。

一群由无政府主义者、自由恋爱倡导者以及香蕉权利激进分子组成的乌合之众从瓦拉塔港劫持了“爱之船”，船上载有 616 名乘客和 327 名船员。他们扬言如果当局在 7 天内不答应他们的条件，就要把船弄沉，并杀死船上所有的人质。条件是什么呢？小面额且未做标记的 100 万美元现金，以及实现一个遵循 GPL 协议的 WATFIV，即著名的滑铁卢 Fortran IV 编译器。（自由恋爱倡导者和香蕉权利激进分子竟然对此达成了一致，实在令人吃惊。）

作为假日游轮公司负责计算机编程的主管，你负责评估能否在 7 天内从头写出一个 Fortran 编译器。你手下有两名程序员。

你们能做到吗？

你说：“嗯，我觉得，得根据具体情况而定。”写文章的一大好处是，我想让你说什么你就得说什么，你对此没有任何办法。

根据什么情况而定？

“那个，我的团队能用 UML 生成工具吗？”

这重要吗？3 个程序员，7 天，滑铁卢 Fortran IV。实话实说，你觉得 UML 工具是决定成败的关键因素吗？

“大概不能。”

那么，关键性因素是什么呢？

“能给我们配 19 英寸的显示器吗？我们有足够的功能饮料吗？”

还是那个问题，这重要吗？咖啡因能决定你们能否做成这件事吗？

“大概不能。等一下，你说我手下有两个程序员？”

没错。

“他们是谁？”

这重要吗？

“当然重要！如果团队相处不融洽，就没法一起工作。而且我知道有少数精英程序员，单枪匹马就能在一周内写出一个 Fortran 编译器，还有很多程序员，给他们 6 个月都写不出启动画面。”

终于说到点子上了！

关于“人是软件项目中的决定性因素”这个话题，每个人都能说得头头是道，但大家都不太确定该如何去提高公司雇员的编程水平。如果你想拥有一支高素质的开发团队，首要任务当然是聘用合适的人选。这意味着你必须能够分辨出谁最合适，而这通常需要一套招聘流程。所以这一章要讲的其实是企业如何面试。

下面介绍企业招聘工作的一般流程。首先收集一大堆简历，然后扔掉看起来不够聪明的候选人简历。我可能有点浅薄，但是我通常会扔掉有一大堆语法错误的简历：连简历都不认真写的候选人，怎么能指望他会认真工作。（有一次招聘专员递给我一份面试 Windows 编程工作的简历，上面写的是“精通 MicroSoft Window”。）在筛选简历时，我更看重的是能暗示候选者曾经过五关斩六将，最终脱颖而出的讯息，也就是说，这个人以前经历了某种非常困难的选拔过程，并且通过了考验。例如候选人毕业于某所录取率很低的大学，这是一个好现象。你可能知道某些以招聘条件苛刻而闻名的公司，如果你看到一份简历的工作履历上出现了这些公司，那么这个候选人可能有点本事。曾经在某些录取率很低的精英军事部队服役也是一个好现象。总结起来，你要找的就是一个毕业于耶鲁大学、在微软工作过、曾经在空降部队服役的人——我。

当然，也不能完全根据简历来决定是否聘用某个人。

下一步通常是一轮电话筛选：给候选人打电话，和他们就某个编程问题聊上半小时，例如，“如何实现一个 XML 解析器？”

通过打电话，你会发现大概四分之三的人不是很聪明。你不得不向他们反复解释同一件事。他们缺乏自己的观点。和他们交谈，你的脑中会不断浮现“傻瓜”两个字。通过电话面试来鉴别并剔除不够聪明候选者，你可以省下真人面试所需的大把时间和金钱。

最后，才是现场、真人的面试。你应该派至少 6 个面试官来面试每个候选人，其中至少有 5 个人是候选人未来的同事（也就是其他的程序员，而不是经理）。有的公司只派资深的经理去面试程序员候选人，并且根据他们的意见决定聘用与否。这种公司招不到好程序员，一次面试太容易蒙混过关了，尤其是由一个不会写程序的人去面试程序员。

只要 6 位面试官中有 2 个认为不应该聘用某人，就不要聘用。也就是说，如果面试某个候选人的前两名面试官都不满意，其实就可以终止面试流程了。这种做法没有错，但为了不让面试者伤心，可以事先不告诉他们总共要经过几轮面试。我听说有个公司在面试时实行一票否决制，这有点冒失了。资深员工的否决意见可能值得参考，但我不会因为某个新员工的一面之词就拒绝面试者。

不要尝试同时面试很多人，这不公平。每场面试都应该一对一地在关好门的房间里进行，并且配有白板以便书写代码。根据我多年的经验，如果一场面试的时间短于一个小时，就无法做出负责任的决定。

在面试中你大致会见到三种人。位于编程水平数轴一端的，是数量众多的门外汉，他们缺乏最基础的编程技能。一般通过两三个简短的问题，就能甄别并且排除他们。而在数轴的另一端，是程序员中绝顶聪明的超级巨星，他们写个 `lisp` 解释器就跟玩一样，使用汇编语言只用一个周末就能完成，而且是在任天堂主机上运行的。编程水平介于以上两种人之间的，是大量“可能适合”的程序员，他们虽然没有什么闪光点，但还是有可能对公司有一定的贡献。面试的窍门就在于如何把超级巨星从一大群可能适合的人选中挑出来，因为招聘的秘诀就是，永远不要聘用那些可能适合的人。

在面试结束的时候，你必须对面试者的表现做出明确的判断。只能有两个选项：聘用或不聘用，别无他选。永远不要说“可以聘用，但别进我的团队”。这种说法非常无礼，相当于说候选人虽然不够优秀，不能和你一起工作，但对于其他部门的傻蛋来说足够优秀了。如果你发现自己想说“可以聘用，但别进我的团队”，直接把这句话翻译成“不聘用”。软件行业瞬息万变，你需要那种具有超强适应能力，随便什么开发任务都能胜任的人。如果通过面试

的交流你发现一个候选人对 SQL 非常熟悉，而对其他的编程领域一问三不知，也是“不聘用”。长痛不如短痛。

永远不要说“可能吧，我说不好”。如果你说不好，那就是不聘用。这其实比你想得简单。感觉“说不好”，那就说“不好”。如果你左右为难，那就是“不聘用”。永远不要说“嗯，我猜可以把他招进来，但是我有点担心……”，这也属于“不聘用”的范围。干脆把所有的犹豫不决转化成“不”，这样就行了。

为什么我在这个问题上会如此固执、不懂变通？因为拒绝一个优秀的候选人远比接受一个差劲的候选人要好得多。差劲的候选人将会消耗公司许多人力和物力，需要浪费其他员工的时间来为他们收拾残局。开除一个本不该聘用的人可能要花好几个月，而且非常困难，尤其是在一切按部就班的时候。甚至有的时候不能随便开除任何人。差劲的员工虽然会影响优秀员工的士气，但他们有可能人缘很好，只是编程水平不高而已，或者他们实在不能丢掉饭碗，无论何种原因你都不忍心开除他们，即使你横下心来开除了他们，还是会让大家都不高兴，等等。总之场面会很难堪。

另一方面，如果你拒绝了优秀的候选人，我猜这可能对他们有点不公平，但是别担心，是金子总会发光的，有很多的好工作在等着他们。别担心会因为拒绝太多人而招不到人，这不是你在面试过程中应该考虑的问题。努力寻找优秀人才固然重要，但当你面试某个人的时候，一定要假装门外还有 900 个人在排队等待面试。无论找到优秀的候选人有多困难，都不要降低选拔的标准。

好，下面来谈最重要的部分：如何决定是否聘用某人？

原则上很简单。你要找的人有如下特质：

- (1) 聪明
- (2) 能干

就这些，你要找的特质只有这两条。把它们记住，每天睡觉前背诵一遍。面试的时间很短，能弄清这两点就不错了，不要浪费时间和其他细节上，比如面试者是不是排解旅途寂寞的聊天高手或者他们是否真的知道 ATL 和 COM 编程。

聪明但不会做事的人经常拥有博士学位，在大公司工作过，即使在那里没什么人听他们高谈阔论。他们会在非常学术性的问题上钻牛角尖，经常不能按

时交付代码。这样的人在面试中很容易识别，他们喜欢指出两个风马牛不相及的概念之间在理论上具有相似性。比如他们会说“电子表格其实只是编程语言的一个特殊例子”，然后花一个星期写一篇白皮书，论述电子表单从编程语言角度来看所具有的理论计算语言学特性。他们的想法是很聪明，但没用。他们还喜欢在临近产品发布的忙碌日子里，手捧一杯咖啡溜达到你的办公室，准备和你好好讨论一下 Java 内省机制对于 COM 类型库的比较优势。

能干却不够聪明的人会不经思考就做出一些愚蠢的事，需要别人替他们收拾残局。这让他们成为公司的“净负债”，因为他们不但没有贡献，还占用了其他优秀员工的时间。这种人宁愿复制粘贴大量的代码，也不自己写一个通用的方法。这样做虽然完成了工作，但是并不高明。

如何在面试中发现聪明这个特质？第一个明显特征是你不需要向面试者反复解释同一件事情，沟通进行得十分顺畅。候选者经常会妙语连珠，显露出独到的见解、深刻的思维或敏锐的直觉。面试官的职责有一部分就是创造一个环境，让面试者充分展现自己的聪明才智。有一种不称职的面试官叫作话痨。他们在面试的整个过程中一直说个不停，让面试者几乎插不上嘴，只能抽空说一句“对，太对了，我完全同意”。话痨会聘用所有的人，因为他们觉得所有面试者都特别聪明，“跟我的想法非常一致！”

第二种糟糕的面试官叫做智力问答节目主持人。这种人觉得聪明就意味着学识渊博。他们会问面试者很多细枝末节的编程问题，答对了才加分。下面这个问题可能是世界上最差的面试题：“Oracle 8i 中的 `varchar` 和 `varchar2` 有什么区别？”这问题太糟了。我真的想象不出，了解这个技术细节对于决定是否聘用某人有什么帮助？谁在乎它们有什么区别？上网查一查，15 秒就能得到答案！聪明并不意味着“上知天文，下晓地理”。软件团队需要聘用有天赋的人，而不是只具备特定领域技术知识的人。因为候选人目前掌握的任何技术知识，都会在几年之内过时，与其聘用一个目前刚好知道某些技术细节（比如用 JDBC 怎样访问 MySQL 数据库）的人，还不如聘用一个能快速学会任何新技术的人。

总的来说，最能了解面试者的方式就是问他们一些开放式的问题，让他们来主导谈话。

那么，该问什么呢？

询问哪些问题

我个人的面试问题列表，源于我就职的第一家公司微软。事实上微软有几百个著名的面试问题。每个面试官都有一套自己的面试问题，你也应该摸索出自己的问题集以及个人的面试风格，来帮助你决定聘用与否。和大家分享一下我用过的几个比较成功的技巧。

面试之前，我会浏览面试者的简历，然后在一张纸上简单写出面试的方案，其实就是我想问面试者哪些问题。以下是面试程序员的方案样例：

- (1) 对面试者进行介绍
- (2) 询问有关面试者最近做过的项目
- (3) 不可能的问题
- (4) 编程问题
- (5) 对自己的表现满意吗？
- (6) 还有什么问题吗？

我非常小心，不让自己对面试者形成先入为主的印象。如果事先得知某人拥有麻省理工学院（MIT）的博士学位，在见面之前就认为他聪明过人，那么在接下来的一个小时，无论他们说什么都不会改变这个印象。如果你得知面试者是从社区大学获得的学位，并因此断定他是个草包，那他无论说什么都不会改变这个印象。面试就像一个非常精密的天平，仅靠1小时的交流就去评判某个人是非常困难的，有时候选择只在一念之间。如果事先知道了一点关于面试者的信息，就好像为天平的一端加上了一个沉重的砝码，这次面试就起不到什么作用了。有次面试前，公司的一位招聘专员进了我的办公室，说：“你肯定会爱上这个人的。”对此我真的很生气。我当时就应该跟她说：“既然你这么确定，那就把他招进来呗，何必浪费我的时间来面试呢？”但由于我当时年少无知，还是面试了他。当面试者的回答显得不那么高明时，我就会想：“哦，大概是马失前蹄了。”我透过玫瑰色眼镜来审视他在面试中的表现。我最后作出了聘用的决定，尽管他的表现并不好。你猜怎么着：其他的面试官一律说“不聘用”。所以，千万不要听招聘专员的话；不要在面试某人之前打听他（她）的情况；在所有面试官独立决定之前，决不要分享对候选人的意见。这样做才科学。

1. 介绍

在面试中，介绍的目的是让面试者放松下来。我会问他们这次航班怎么样。我会花大概 30 秒时间向面试者介绍自己以及面试的基本流程。我总是告诉面试者不必过于担心答案的对错，我关心的是解决问题的方法。

2. 最近做过的项目

接下来询问面试者近期项目的相关信息。面试在校学生的时候，可以问他们的毕业论文（如果有的话）或者在课程中做过的时间较长、比较感兴趣的项目。例如有时候我会问：“你最喜欢上个学期的哪门课程？不一定是计算机相关的课程。”面试工作经验丰富的候选人时，可以讨论上一份工作中的项目。

我重复一遍：询问开放式的问题，然后倾听。在面试者卡壳的时候可以说“请再详细说说那个项目”，以便沟通顺畅进行。

在倾听面试者解答开放式问题的时候，应该寻找哪些特质呢？

第一，激情。聪明的面试者会对做过的项目饱含感情。一谈起相关的话题他们就显得非常兴奋、语速加快、手舞足蹈。即使带有负面的感情也是好现象。“我的前任主管只懂 VAX 计算机，所有的事都得在那上面完成，弱爆了！”职场中有太多的人不带着一种事不关己、好坏都无所谓的态度。这种人的积极性是很难被调动起来的。

糟糕的面试者在面试中表现得满不在乎，整个过程都缺乏热情。如果面试者在谈论起某个项目时表现得非常投入，像是忘了正在进行面试，那是非常好的现象。有时候某个面试者刚进房间的时候会很紧张，这在面试中太正常不过了，我通常会忽略这种表现。而当你引导他谈论起计算机素描艺术的时候，他会表现得极为活跃，紧张的表现一扫而光。非常好，我就喜欢对工作有激情、全身心投入的人。（如果你好奇什么是计算机素描艺术^①的话，试着拔掉你的显示器。）你可以用一些问题来挑战面试者的说法（试试这个：等他们说某事有可能为真时说“这不可能是真的”），然后看他们如何强势地捍卫自己的观点。相比 5 分钟前紧张到出汗的情形，简直判若两人，因为他们确实非常在意自己讲的东西，以至于忘记了自己的命运还

^① 根据 quora 网友的说法，计算机素描艺术是作者杜撰的，其实就是黑屏，在这里它代表令程序员们非常着迷却让外人一头雾水的事物。——译者注

掌握在面试官手中。

第二，无论多么高深的话题，优秀的面试者都会讲得深入浅出、明明白白。我拒绝了好几个面试者，因为他们不能用常人理解的语言来描述做过的项目。主修计算机科学的面试者通常会忘记不是每个人都知道贝茨定理或 $O(\log(n))$ 的含义。如果他们喜欢滥用术语，我就这样打断他们：“麻烦你能用我奶奶听得懂的方式解释这个概念吗，就当是一个小小的练习？”这样说过之后，还是有很多人继续使用各种技术行话，完全不考虑别人能否听懂。这种人不值得聘用，因为他们没有足够的才智来让别人听懂自己的想法。

第三，听面试者讲做过的团队项目时，努力寻找领导潜质。面试者可能会说：“我们本打算做功能甲，上司说要做乙，而客户想要的其实是丙。”我就会问：“那你们最后做了什么呢？”比较好的回答可能是：“我把团队召集在一起，大家写了一个提案……”比较差的回答可能是：“这种情况下我们能怎么办呢？没办法。”记住，你寻找的是聪明又能干的人。判断某个人的办事能力强弱，唯一可靠的标准是以前交给他的事是不是大多数都办成了。事实上，你甚至可以直接要求面试者举例说明，以前是如何克服团队中的惰性，带领大家做成某件事的。

3. 不可能的问题

我的面试问题列表的第三项是不可能的问题。这很有意思。基本思路是，问他们一个很难找到合适回答方式的问题，看看他们如何应对。比如询问“西雅图一共有多少名眼镜验光师？”“华盛顿纪念碑有几吨重？”“洛杉矶市有多少加油站？”“纽约一共有多少名钢琴调音师？”

聪明的面试者会意识到你考察的不是他们的知识，然后积极地尝试，去做一些基本的估算。“嗯，我想想，洛杉矶大概有7百万人口，人均拥有2.5辆汽车……”即使他们的估算错得离谱也没关系。重要的是，他们表现出对于解决这个问题的热情。他们可能会这样估算一个加油站的接待量：“一辆车加满油大概要4分钟，一个加油站大概有10台油泵，每天开18个小时……”他们还可能会从地理面积上进行估算。有些面试者会别出心裁地向你要一本洛杉矶的黄页电话本。所有这些都是标志着面试者聪明能干的好现象。

不太聪明的面试者遇到这种问题，就会脸红或是生气。他们会死盯着你看，好像你是从火星来的。你不得不去引导他们进行思考。“假如你负责规划一

个像洛杉矶这么大的城市，你觉得布置多少加油站合适呢？”你也可以给他们一些小小的提示。“你觉得加满一辆车的油需要多长时间？”尽管如此，不太聪明的面试者还是经常会在整个过程中表现被动，等着你用各种提示来拖着他们前进。这些人是不擅长解决问题的人，不应该雇佣他们。

4. 编程问题

面试的绝大部分时间都应该花在这个环节。我会让面试者用 C（或是其他擅长的语言）写一个小函数。我通常会问这些问题：

- (1) 就地逆转一个字符串
- (2) 逆转一个链表
- (3) 计算一字节有几比特
- (4) 二分查找
- (5) 找出字符串中最长的连续串
- (6) atoi
- (7) itoa（这个很好，因为必须要用到 `strev` 或栈）

不要问代码行数超过 10 行左右的问题，没那么多的时间。

让我们来看看其中的几个问题。第一，就地逆转一个字符串。在我面试过的所有程序员中，没有一个能把这个函数一次写对。他们无一例外地尝试分配另一块缓冲区，然后把字符串逆序输出到缓冲区。问题是，谁来分配缓冲区？谁来释放缓冲区？通过向几十位候选人提问，我发现了一个有趣的事实：大部分以为自己精通 C 语言的人，其实都不懂内存和指针。他们真的不懂。我十分惊讶这么多人是怎么当上程序员的。对于这个问题，你可以从以下几个方面来评判程序员的答案。

函数够快吗？看看调用了几次 `strlen`。我看到过用 $O(n^2)$ 的效率实现的 `strev` 函数，而这个函数的执行效率本来应该是 $O(n)$ ，错误之处是他们在循环里重复调用了 `strlen`。

他们使用指针运算符了吗？这是一个好现象。很多所谓的 C 程序员不知道如何使用指针运算符。通常，我不会因为某个面试者不具备某种特定编程技巧就拒绝他们。然而，我逐渐意识到理解 C 指针不是一种编程技巧，而是一种天赋。在大学一年级的计算机科学课程中，在学期开始时通常有 200 个孩子，所有人在 4 岁的时候就能用 BASIC 写冒险游戏。他们在大学里愉快地学习 Pascal，直到有一天教授开始讲解指针，他们就都突然听不懂了。

这意味着之后的内容他们全都听不懂。上课的学生流失了 90%，有人转到政治学专业，然后告诉朋友，因为计算机科学专业的帅哥美女太少，所以他们才转专业。不知为何，大部分人的脑中都不具备理解指针的关键部分。很多人都不能以一种双重且无向的复杂思维方式思考，因此无法理解指针，而理解指针对于写好代码是非常关键的。许多“脚本高手”通过把 JavaScript 代码片段复制到自己的网页里而走上编程的道路，后来又学习了 Visual Basic 等脚本语言，但他们从来不知道什么是指针，因而无法写出质量达标的代码。

我们再来看问题 3，计算字节中的比特数。你不需要考察他们是否知道 C 语言中的位运算符，因为这是能够查到的信息。如果他们不会，你可以帮他们。最有意思的部分是，等他们写出了计算字节中比特数的函数，再请他们提高代码的运行速度。真正聪明的面试者会建一张查找表（总共算下来只有 256 种情况），这样只需要算一次，之后查找即可。你还可以和这些优秀的候选人谈谈，如何在存储空间和执行速度之间进行交换和权衡，一定很有趣。还可以给他们更多的压力，告诉他们，你不想花额外的时间去建查找表。聪明的面试者也许会提出一种缓存的办法，即第一次计算某个字节的比特数时把值填入查找表，之后在计算 1 字节的时候就可以直接从查找表读取，不用再数一遍了。而绝顶聪明的面试者会取捷径，利用数字出现的某种规律来计算查找表。综上所述，聪明的面试者能够在你的引导下完善、优化代码，并且提出很多改进的方法。

让面试者放心，你了解在没有代码编辑器的环境中编程的困难，因此不会在乎潦草的板书。你还了解在没有编译器的环境中写出没有 bug 的代码十分困难，所以不会苛求。

优秀的程序员通常有如下习惯：他们会遵循某种变量命名规范，即使这种规范比较原始。优秀的程序员通常会为循环的索引变量起一个短小的名字。如果一个人把循环索引变量命名为 `CurrentPagePositionLoopCounter`，很显然他没写过多少代码。对于 C 语言，注意寻找某些良好的习惯，比如把常量放在“=”的左边（例如 `if(0==x)` 而不是 `if(x==0)`），这样在不小心写成“=”的时候编译器会报错，避免了把判值语句写成赋值语句。有些程序员认为这样写很别扭，应该由编译器来捕获在 `if` 中赋值的错误。我并不是推荐大家都这样写，而是说当你看到面试者这样写的时候，可能是个好现象。优秀的面试者会本能地知道 `while` 循环中的变量应该是什么值。使用 C 语言遍历一个数组的时候，他们总是会不假思索地写出 `while(index<length)`，而不是

<=。使用 C++ 语言的时候，他们总会为析构函数加上 `virtual` 标志。

优秀的程序员在动笔之前会先思考，尤其是在写和指针有关的逻辑时。例如，如果让他们写一个逆转链表的函数，优秀的候选人总会先在旁边画个草图，看看每个指针该如何调整。他们必须这样做。以人类的能力，如果不先画出一些小方框和小箭头，就不可能写出逆转链表的函数。不够优秀的程序员则会立刻开始写代码。

5. 你对自己的表现满意吗？

你难免会在他们写的程序中发现一个 `bug`。这就过渡到了面试问题列表中的第 5 项：“你对自己的表现满意吗？”你也许想问：“OK，你觉得代码哪里有 `bug`？”这完全是来自地狱的开放式问题。所有的程序员都会犯错，这本身没什么不好，只要他们能意识到错误就好。大部分大学毕业生都会在写 C 语言的字符串函数时忘记为字符串添加 `null` 结束符。对于所有的函数，他们都有可能犯差一（`off by one`）的错误。他们有时候会忘记写分号。函数可能会在处理长度为 0 的字符串时出错，或者在用 `malloc` 分配内存失败时触发通用保护性错误（`GPF`），等等。很难碰到一个面试者能一次把代码写对。在这种情况下问这个问题就更有意思了。当你声称“代码里有个 `bug`”的时候，他们会立刻仔细检查，之后就看他们能否坚持自己的代码是正确的，以及能否婉转地表达这个意思。

6. 你有什么问题吗？

作为面试的最后一步，询问面试者他们是否有任何的问题。记住，虽然是在面试他们，但优秀的程序员总是有大把的工作机会可以挑选。他们也在借此机会评估你提供的工作机会是否适合自己。

有些面试官会评判面试者的问题是不是“有水平”。我个人并不在乎他们问什么，因为在这个阶段，我已经作出了我的决定。问题是面试者一天里可能要和五六个人交谈，很难每次都问一些不一样的、睿智的问题，所以如果他们说没什么问题好问，那也没关系。

我总在面试结束的时候抽出大概 5 分钟，向面试者宣传我们的公司以及提供的工作机会。即使你并不打算聘用他们，这也是有必要的。如果你运气好，找到了一个真正优秀的候选人，那此时更应该使出浑身解数来把人留住。即使候选人不是特别理想，给他们留下良好的印象也不是什么坏事。

啊，我刚想起来，应该列举一些特别糟的问题。

不要问哪些问题

首先，不要问违法的问题。在美国，在面试中询问有关种族、宗教、性别、祖籍、年龄、服役经历、性取向或身体残疾方面的信息是违法的。如果面试者在简历中说 1990 年在军队服役，你不能问当时是否在科威特，哪怕是活跃气氛的闲谈也不行，这是违法的。如果简历中说他毕业于海法的以色列理工学院，不能问他是不是以色列人，就算你的妻子是以色列人或者你最喜欢吃以色列的油炸鹰嘴豆饼也不行，那也是违法的。

其次，避免让面试者觉得你在依据某个方面或某些标准来区别对待他们，其实你根本没那么想。我能想到的最好的例子是，问某人有没有孩子或者有没有结婚。这可能会给面试者造成一种错误的印象，以为你觉得有孩子的人在工作上投入的时间不够，或者会突然申请长期的产假。总而言之，只问那些和目标职位完全相关的问题。

最后，不要问脑筋急转弯类的问题，比如如何用 6 根等长的棍子摆出 4 个全等的等边三角形。它们大部分属于“啊哈”问题，要么你知道问题的答案，要么不知道，而知道答案只意味着一件事：你以前听过这个问题的答案。作为面试官，你通过脑筋急转弯类的问题，完全得不到有关面试者是否“聪明能干”的信息。

面试结束的时候，如果你确信面试者既聪明又能干，并且其他面试官中有 4 到 5 个同意你的意见，那么可以立刻聘用，一般不会错。但如果你有任何的疑虑，还是把希望寄托在将来出现更好的面试者上吧。

21

奖励有害论

2000年4月3日，星期一

麦克·穆雷曾经是微软公司的人力资源经理，他干过不少荒唐事，对公司最大的“贡献”就是在任职之后不久设立“交付”奖。也就是说，当一个产品交付的时候，团队成员会得到一座大小跟字典差不多的水晶奖杯。发奖的目的是为了鼓励好好工作：如果哪个人工作不够称职的话，就不给他水晶奖杯！我特别想知道，微软在推出这个奖杯之前，是如何交付产品的呢？



交付奖这个制度是在一次盛大的公司野餐活动时宣布的，公司内部曾经对它满怀期待，大家议论纷纷。宣传海报早在活动的几周前就贴满了办公园区的各个角落，上面印着微笑的比尔·盖茨，下面写着：“为什么这个人笑得这么开心？”我不知道这是什么意思。比尔笑得那么开心，难道是因为交付产品的人终于有奖可拿了？在公司揭晓答案的那一刻，员工们都大呼上当，觉得自己被当成了小孩子。台下嘘声一片。Excel 编程团队打出了大幅标语，上面写着：“为什么 Excel 团队的人都在打哈欠？”道格拉斯·库普兰的经典小说《微软奴隶》^①中有一篇（不是虚构的）故事，写的是一群程序员尝试

^① 道格拉斯·库普兰，《微软奴隶》。

用喷枪来毁掉一个奖章^①，由此可见交付奖受大家的鄙视程度。

这种把高精尖人才当幼儿园孩子的做法并不鲜见。几乎每个公司都有类似的、让人觉得带有侮辱和贬低意味的员工激励计划。

在我待过的两家公司里，一年中压力最大的时间就是每年两次的绩效评估期。不知怎么的，朱诺（Juno）公司和微软公司人力资源部门的绩效评估系统如出一辙，好像抄袭了呆伯特的讽刺漫画，整个流程一模一样。首先，员工对直接上级进行“匿名”的绩效评估（就好像没人会偷奸耍滑一样）。然后，员工可以填写一份的“自我评估”表格，不填也行，经理会在准备每个人的绩效评估材料时“综合加以考虑”。最后，每个人得到一个绩效数值表，里面包括许多难以量化的得分项，比如“和同事相处融洽”，得分的值域虽然是1到5分，但其实能得到的不是3分就是4分。经理会向上头提交每个人的奖金分配建议，这完全没用，每个人最后得到的奖金数额几乎完全是随机的。这个系统完全没有考虑到每个人的差异性以及特殊才能的作用，而这些正是良好的团队合作所必需的。

绩效评估会给员工带来很大的压力。我有许多朋友，尤其是那些才华卓著、但无法体现在传统量表中的朋友，通常在绩效评估中的得分很低。例如，我的一个朋友就像是正能量的催化剂、活力十足的领航员，每当团队遇到困难的时候，他就站出来鼓励大家。他是让团队凝聚在一起的粘合剂。尽管如此，他在绩效评估中总是得到不好的评价，因为经理并不理解他的特殊贡献。另一个朋友具有长远的战略眼光，他通过和别人讨论做事的方式和规范，提高了每个人的工作效率。他用比一般人更多的时间去探索新技术，这些探索为团队创造了不可估量的价值。但是仅仅从产出的代码行数上考量，他写得要比一般人少，而经理昏庸到根本注意不到他在其他方面的贡献，所以他也得到了很多负面的评价。负面评价无疑是对员工士气的沉重打击。事实上，即使给某人一个正面的评价，但如果低于他们所预期的程度，同样也会对士气造成负面影响。

评价对于士气的影响是一边倒的：负面评价严重影响士气，而正面评价对于士气或生产效率没有影响。得到好评的员工本来就干得很出色。他们认为，对自己份内的工作尽心尽责、精益求精，本来就是一个公司雇员应有的职业素养，而给他们好评只会让他们觉得自己被当作了巴甫洛夫实验中的狗，工

^① 出自《微软奴隶》。在线阅读地址：http://www.wired.com/wired/archive/2.01/microserfs_pr.html。

作的目的只是为了奖赏，而非追求工作质量。

实际情况更加复杂。因为大部分人都认为自己做得不错（实际上有可能不是这样）。人们通过自我催眠来让生活轻松一点。如果所有人都自我感觉良好，而对他们的评价只能算是公允（做到这点已经很不容易了），那么大部分人都对自己得到的评价感到失望。这对于团队士气的影响是很大的。在那些严格执行绩效评估的团队，通常结果公布之后的一周大家都会非常沮丧、士气低落、无精打采，部分人还会提出辞职。绩效评估还会造成团队成员的隔阂，得到较低分数的人会嫉妒那些得分较高的人，德马科和李斯特^①称这种现象为团队谋杀：人心涣散的团队注定会走向瓦解。

《哈佛商业评论》刊载过阿尔菲·科恩的一篇经典文章，里面写道：

……最近三十年来，至少有几十项研究结论均表明，那些期望通过完成或出色地完成某个任务而得到奖励的人，其表现不如那些不期望得到奖励的人好。^②

他总结“奖励（或贿赂）在职场是行不通的”。德马科和李斯特进一步明确指出，任何类型的职场竞争，任何的奖惩制度，甚至最老套的“发现并奖励好人好事”，都是弊大于利的。给某人一个看似正面的奖励（比如在愚蠢的公司集会上给员工发奖章）就是在向大家暗示，这些人好好工作的目的是为了得到水晶奖章，也就是在暗示，如果不能偶尔尝到一点甜头，他们是不会自觉干活的。这种名为奖励的行径，实为一种侮辱和贬低。

绝大多数软件经理出于无奈，只能遵从现有的绩效评估制度。如果你坐在这个位置上，阻止团队谋杀的唯一方法就是给团队里的每个人最好的评价。但如果你有选择的余地，我还是建议，远离各种形式的绩效评估、激励奖金或是愚蠢的每月最佳员工评选活动。

① 汤姆·德马科，蒂姆·李斯特，《人件（第2版）》。

② 阿尔菲·科恩，“Why Incentive Plans Cannot Work”，《哈佛商业评论》1993年9月/10月号。参阅<http://hbr.org/1993/09/why-incentive-plans-cannot-work/ar/1>。

22

不设专职测试人员的 5 条（错误）理由

2000 年 4 月 30 日，星期日

1992 年微软刚推出 Word 的 Windows 新版本时，科技作者詹姆斯·格雷克写了一系列抨击软件质量问题的文章。格雷克认为 Word 糟透了，并写了一篇长文，刊登在《纽约时报杂志》的周日版上。这篇文章给人一种咄咄逼人的感觉，好像要把 Word 团队放在火上，“烤问”他们为什么对客户需求毫无响应，而且还交付出 bug 这么多的产品^①。

后来，他又想让自己的互联网服务提供商 Panix（我也使用这家公司的网络服务）提供一种邮件自动排序和过滤的方法。UNIX 有一个叫 procmail 的工具能够做到，但是它的界面非常晦涩难懂，即使是 UNIX 的忠实追随者也承认这工具太难用了。

有一天，格雷克先生在使用 procmail 时不小心打错了字，所有的邮件都被删除了。盛怒之下，他决定自己开一家互联网服务公司。他聘用了一个叫乌代·伊瓦图里的程序员，创立了 Pipeline 公司。这家公司的服务理念超前于那个时代，是第一家具有图形化界面的商业互联网服务提供商。

然而 Pipeline 也有自己的问题。产品的第一个版本没有任何的纠错机制，用户经常遇到内容错乱、程序崩溃的情况。就像所有的软件一样，Pipeline 也有 bug。1993 年我去 Pipeline 求职。在面试中，我向格雷克先生询问他写的那篇文章。“现在您来到了围墙的另一侧”，我问，“您是否体会到了，开发一

① 詹姆斯·格雷克，“Chasing Bugs in Electronic Village”，《纽约时报杂志》，1992 年 8 月 4 日。在线阅读地址：<http://www.around.com/bugs.html>

“个好用的软件是多么不容易？”

格雷克非常固执。他否认 Pipeline 有 bug，否认自己公司的产品和 Word 一样糟糕。他告诉我：“乔尔，总有一天，你也会开始憎恨微软的。”我有点吃惊，像他这样不仅有多年软件使用经历，而且还有多年软件开发经验的人，为什么不肯承认开发出没有 bug、易于使用的软件是一件非常不容易的事？面试后我就离开了，没有接受这份工作。（Pipeline 后来被世界上最奇葩的互联网服务提供商 PSI 收购，之后毫无征兆地被拖去枪毙了。）

软件难免出现 bug。CPU 极其挑剔，如果没有明确教给它某项任务怎么做，它会拒绝处理，而且拒绝的方式非常幼稚。当我把笔记本电脑带出家门的时候它就经常崩溃，因为找不到以前配置的家庭网络打印机。这也太孩子气吧。很可能只是某一行代码存在一个极小的 bug。

因此，软件公司必须要有测试部门。程序员和测试员的比例应该达到二比一（如果软件的配置复杂或者支持很多操作系统的话，需要的测试员数量还要更多）。每个程序员应该和单独一个测试员密切配合，尽可能经常地让他们测试私有构建版本。

测试部门应该是独立且强势的，不能让它听命于开发部门。事实上，如果软件质量达不到标准，测试部门的负责人应该有权否决某次软件的发布。

我真正的第一份软件开发工作是在微软公司，虽然它并不以软件质量而闻名，但是仍然雇佣了一大批软件测试者。所以我想当然地认为所有软件公司都有测试员。

尽管大多数公司都有测试员，但不配备专职测试人员的公司多到让我吃惊。事实上，很多软件团队甚至不相信测试的价值。

你可能会以为，经过 20 世纪 80 年代对于产品质量的狂热（包括 ISO-9000 等质检认证、“六西格玛”等热门词汇），如今的管理者会把提升产品质量融入他们的经商理念中。事实上，他们脑中确实有这个概念。但最后还是会编出很多理由，不招软件测试人员。

我希望通过下面的文章向你说明，为什么这些理由全部都是错误的。如果你赶时间的话，请略过下面的内容，直接去招一批全职测试人员，数量和全职程序员达到一比二就好。

以下是我最常听见的不招测试人员的理由。

1. 懒惰的程序员才会写出bug

背后的歪理是：“如果我们有专门的测试员，程序员就会偷懒，写出 bug 很多的代码。而不招测试人员，就可以迫使程序员一次性把代码写对。”

持这种看法的人，要么从没写过代码，要么就是在刻意歪曲写代码的真实情况。根据定义，bug 之所以会出现，就是由于程序员无法发现自己代码中的错误。很多时候，只要多一双眼睛，就很容易发现代码中的 bug。

当我在朱诺公司写代码的时候，通常只用一种自己惯用的方式来测试代码，操作时大量使用鼠标。而我们不可思议的、工作能力远超职责要求的测试员则有自己的一套测试习惯：她更倾向于用键盘（而且不辞辛劳地用各种可能的输入组合来测试界面）。这样做能很快地暴露大批 bug。事实上，有时候她告诉我整个界面都不能用，100%不能用，尽管我用得好好的。我在看她复现 bug 的过程中，突然有一种醍醐灌顶的感觉。Alt 键！你按下了 Alt 键！为什么我没想到测试这个呢？

2. 我们开发的软件部署在网上， 一旦发现bug可以很快修复

哈哈哈哈哈！好吧，我承认与过去的盒装软件相比，网络能让你更快地发布修复 bug 的补丁。但是不要因为开发的是网站就低估修复 bug 的成本，尤其是在项目已经完工，暂时冻结之后。首先，你可能在修复 bug 的时候引入更多的 bug。更坏的情形是，如果你们发布新版本需要经过复杂的发布流程，那么在线发布修复补丁可能是一个代价颇高的提议。再者，让用户使用有 bug 的产品会造成负面印象，这就到了第三个不招测试人员的理由。

3. 用户会帮我测试软件

啊，这就臭名昭著的“网景公司的辩解”。该公司秉持的“测试”理念对其声誉造成了巨大的破坏：

- (1) 当程序员的工作完成一半的时候，通过网络发布软件，不进行测试；
- (2) 当程序员声称工作做完了的时候，通过网络发布软件，不进行测试；
- (3) 整个过程重复 6~7 遍；
- (4) 把软件版本定为“最终版本”；
- (5) 每次 CNET 上提到一个令人感到羞愧的软件 bug 时，就发布 .01、.02、.03 的修复版本。

这家公司率先提出了“大范围 beta 测试”的概念。他们真的让几百万人下载 bug 成堆的半成品软件。在最开始的几年，几乎所有使用网景浏览器的人用的都是某个预发布或 beta 版本。结果大部分人都认为，网景公司的软件就是有很多 bug。即使软件的最终发行版本质量还过得去，但使用有 bug 版本的人太多了，平均下来，在大部分人的印象中，网景软件的质量非常差。

而且，“客户”测试能否生效，取决于能否定位并修复用户遇到的 bug。不幸的是，无论网景公司还是这个世界上的任何其他公司，都没有足够的人力来筛查从 2 百万用户那里收集来的 bug 报告，分析出其中有价值的信息。我在使用网景浏览器 2.0 版本的时候，就发现 bug 报告网站本身屡次崩溃，导致我根本没法提交报告（即使提交成功了，这份报告也没人看）。而且网景公司不懂得亡羊补牢。我在新闻组中看到，当前 6.0“预览”版本的测试人员还在抱怨 bug 报告网站无法提交数据。这么多年过去，问题还没有解决！

我敢打赌，虽然 bug 报告的数量很大，但大部分是 5~10 个最明显的 bug 在不断地重复出现。在 bug 的汪洋大海中，可能有那么一两个有趣、难以定位的 bug，但是没有人会去看这些用户好不容易才上传的 bug，所谓的用户测试根本就是形同虚设。

这种所谓的测试方法的最大缺点是对公司形象造成非常负面的影响。UserLand 公司刚刚发布旗舰产品 Frontier 的 Windows 版本时，我下载了一份，在照着官方教程进行操作的时候，Frontier 崩溃了好几次。我逐字逐句地按

照教程上的指示来做，结果还是不能成功地在软件里待上超过 2 分钟。我觉得 UserLand 公司在产品发布前压根没有做过任何测试，甚至不能保证软件能按照官方提供的教程正常工作。由于产品质量的低下，我在很长一段时间里都对 Frontier 非常抵触。

4. 能胜任测试工作的人都不想当测试员

这条理由切中要害。招到优秀的测试人员确实太困难了。

测试员和程序员一样，最好人选的能力要比平均水平高出一个数量级。在朱诺公司，我们有一个测试员叫吉尔·麦法兰，她一个人找到的 bug 数量，是其余 4 个测试员找到 bug 数量总和的 3 倍。丝毫没有夸张的成分，我真的做过统计。也就是说，她的工作能力是普通测试员的 12 倍还多。她辞职的时候，我给 CEO 发了一封电子邮件，说“吉尔一周工作两天，也比剩下的整个测试团队出色”。

不幸的是，聪明到这种程度的人很快会对日复一日的测试工作感到厌倦，顶尖的测试员通常只会在一个地方干 3~4 个月，然后就会跳槽。

对于这个问题只有一个办法，那就是面对现实，未雨绸缪。以下是一些建议。

- 对于技术支持人员来说，测试可以是职业升级之路的目标职位。虽然测试工作很磨人，但是比起应付愤怒的客户投诉电话来说好多了，并且测试工作做好了，也会减轻技术支持方面的压力。
- 允许测试员学习编程以谋求自身更好的职业发展，鼓励聪明的测试员用编程工具和脚本语言开发自动化测试的工具。这比一次又一次地手动测试同一个对话框有意思多了。
- 接受顶尖测试员经常跳槽的事实。通过勤勉的招聘来保持人力资源的充足。不要因为暂时不缺人手就停止招聘工作，因为黄金时代很快就会过去。
- 发掘“非主流”的测试人员：聪明的青少年、大学在读生和退休人员等，让他们来兼职。只要有两三个顶尖的全职测试员以及大批来自布鲁克斯技术高中（一所位于纽约的重点高中）的学生（让他们暑期实习以赚取大学学费），你就能组建起一支非常强大的测试团队。

□ 招临时工。招 10 个临时工，在几天内对软件进行高强度的测试，你会发现非常多的 bug。这批临时工中可能有两三个人拥有很好的测试技巧，可以考虑和他们续签全职的合同。如果发现某些临时工无法胜任测试工作，就及早把他们打发走，再招新人。很多中介机构都提供这方面的服务。

但是不要用下面这种方法来应对招不到好测试员的问题。

□ 决不要告诉大学计算机科学专业的毕业生，他们可以来工作，但是“在承担开发项目之前，每个人都要做一段时间的测试工作”。这种事我见多了。程序员并不一定能胜任测试工作，你还有可能因此失去一位更加难以替代的优秀程序员，得不偿失。

最后，是我见过所有不招测试人员的理由中最愚蠢的一条。



5. 我雇不起测试员！

这理由再愚蠢不过了，根本站不住脚。

无论招聘测试人员有多难，他们的要价比起程序员还是便宜得多。不招测试员的话，程序员势必要承担部分测试的工作。如果你对测试人员的流失感到非常头疼，那么程序员的流失更会让你欲哭无泪。一个明星级别的程序员年薪可达 10 万美元，如果他们烦透了公司“发布产品前花几周时间进行测试”的不合理要求，就会立马离职，去找一家职业分工更合理的公司。到那时，光是付给招聘机构的服务费用，就足够你在整整一年里雇佣三个测试人员。

通过不招测试人员来节省开支的想法，从经济角度来看完全是错误的，我很惊讶为什么很多人都不明白这个简单的道理。

23

任务切换有害论

2001年2月12日，星期一

作为一名软件开发团队的管理者，首先必须要掌握的一项技能就是分配任务。说白了就是给人们找事做。在希伯来俗语中有另一种表达方式，翻译过来是“倾倒文件”（因为你要做的其实就是把一堆文件倾倒给某人）。一旦管理者对于把哪堆文件倾倒给哪个人有了一定的心得，团队的生产效率就会因为任务得到合理的分配而大幅提升。相反，如果任务分配不合理，整个团队会举步维艰，所有人都完不成任务，大家会抱怨“呆在这破地方什么事都干不成”。

由于本书的读者大多数是程序员，我就来给大家出道编程题热热身。

假设你拿到两个独立的计算任务 A 和 B，它们分别需要 10 秒的 CPU 运算时间。为简单起见，不妨假设你可以使用一块 CPU，并且当前 CPU 的任务队列是空的。

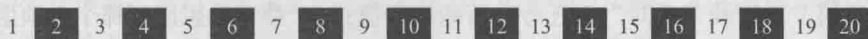
这块 CPU 支持多任务切换。所以，你可以选择串行计算，即先计算 A 然后计算 B。

串行处理



或者，你可以选择并行处理两个计算任务。

并行处理



如果选择并行处理，在这块 CPU 上，单个任务一次运行最多占用 CPU1 秒的时间，在任务之间切换的时间可忽略不计。

你打算如何执行计算任务？大部分人凭直觉会青睐并行计算的方案。无论采用哪种方案，得到两个计算结果的最终时间都是 20 秒。但是请想想看，得到两个计算结果分别要多久。

在两种情况下，计算任务 B（用黑色表示）都是在 20 秒得到结果。但是再看看任务 A 呢？如果使用并行方案，需要 19 秒钟才能得到结果，而在串行处理方案中，只需要 10 秒就能得到 A 的结果。

也就是说，在这个理想化的例子中，串行处理的平均计算时间（15 秒）要低于多任务并行处理的平均计算时间（19.5 秒）。（事实上，这个例子并不是我凭空想出来的，它是根据杰瑞德在工作中遇到的一个实际问题改编而来的。）

方法	计算任务 A 需要	计算任务 B 需要	平均
串行	10 秒	20 秒	15
并行	19 秒	20 秒	19.5

我刚才说“任务切换的时间可以忽略不计”。而实际上，真正的 CPU 在切换任务的时候是要消耗一点点时间的。在这段时间里，CPU 保存当前任务的寄存器状态，并且加载另一个任务的寄存器状态。理想情况下，这个时间是短到可以忽略不计的。但是为了让事情更有意思，假定任务切换需要半秒的时间。这样一来，并行处理的方案变糟了。

方法	计算任务 A 需要	计算任务 B 需要	平均
串行	10 秒	20+1 次任务切换=20.5 秒	15.25
并行	19+18 次任务切换=28 秒	20+19 次任务切换=29.5 秒	28.75

接下来，（就算是让我开心一下，我知道做这种假设很蠢）如果任务切换需要一分钟会怎么样？

方法	计算任务 A 需要	计算任务 B 需要	平均
串行	10 秒	20+1 次任务切换=80 秒	45 秒
并行	19+18 次任务切换=1099 秒	20+19 次任务切换=1160 秒	将近 19 分钟！

任务切换需要的时间越长，多任务并行处理的性能下降越明显。

当然，这个例子本身并不能说明什么问题。我大概很快就会收到愤怒的读者来信，声讨我“反对”多任务并行的编程模式。“难道你想回到那个必须退出 WordPerfect 才能运行 1-2-3 的 DOS 时代吗？”他们会这样质问我。

我不是这个意思。我只是想让你同意如下观点：

- (1) 串行处理的平均速度更快；
- (2) 任务切换需要的时间越长，多任务并行处理的性能下降越明显。

好，我们再回到管理程序员的话题，这可比管理 CPU 有趣多了。和管理 CPU 不同的是，程序员在切换任务时要花很长很长的时间。因为编程是需要同时把很多信息放在脑中的思维活动。一个人能够同时在脑中存放的信息量越大，他（她）的编程效率就越高。一名程序员在开足马力编程的时候，他（她）的脑中同时存放着各种信息，包括变量的名称，数据的结构，重要 API 的用法，自己编写并经常调用的工具函数，甚至是源代码储存的具体路径。如果你把某个程序员送到克里特岛度假三个星期，他们会把所有信息都忘得一干二净。就好像是人脑把这些信息从短期的随机存取存储记忆体（RAM）迁移到了用于长期备份的磁带记录体，这一旦发生，再找回这些信息就需要花费非常久的时间。

到底需要多长时间呢？举个例子，我的软件公司在最近 3 周暂停了手头的工作（开发代号为 CityDesk 的软件产品），所有人都去帮某个客户处理突发情况。当大家回到办公室的时候，我们发现大概需要额外的 3 周，才能让 CityDesk 项目的开发回到正轨。

我们再来看个人层面的并行问题。不知道你注意过没有，根据我的经验，如果把一项任务指派给一个人，这个人能很好地完成任务；而当你把两项任务同时指派给某个人时，通常的结果是两件事都做不好。这个人要么做好其中一项任务，而完全不做另一项，要么会用比蜗牛还慢的速度同时做两项任务。这是因为在编程任务之间进行切换需要的时间太长了。我的感觉是，如果我同时做两项编程任务，那么进行一次任务切换大概需要 6 个小时。这意味着在一个普通的 8 小时工作日里，如果同时处理多项任务，我只能高效地编程 2 小时。这太令人沮丧了。

根据我的经验，当你给一个人同时指派两项工作的时候，如果他（她）选择只做其中一项而“晾着”另一项工作，你应该感到庆幸，因为这样做缩短了任务完成的平均时间，能在同一段时间内完成更多的工作。以上事实对于管理者的启示是，永远不要把多于一项的任务指派给同一个人做。要确保每个人都时刻知道自己要完成哪项工作。优秀的管理者把为下属排忧解难作为自己的工作使命，这样下属就能集中精力，一次只做一件事，并且做到最好。做为管理者，遇到突发情况要首先考虑自己能否独立处理，实在不行才把任务委派给下属，尽量不要打扰全身心投入在项目中的程序员。

~ 24 ~

永远不要做的事情（第一部分）^①

2000年4月6日，星期四

网景浏览器 6.0 终于进入了第一轮公众 beta 测试阶段。5.0 版从来没有出现过。上一个重大发行版本 4.0 是在将近 3 年前发布的。3 年在互联网世界里是很长的一段时间。在这段时间里，网景公司坐视自己产品的市场占有率直线下降。

可能有人觉得我站着说话不腰疼，毕竟网景公司并不是在有意拖延产品的发布。他们也不想这样，不是吗？

恰恰相反，我认为他们自作自受。网景犯下了一个软件公司所能犯下的最严重的战略性错误：

他们决定从头开始重写全部的代码。

网景并不是第一个犯下这种错误的公司。Borland 公司在收购 Arago 并尝试将其整合进 dBase 的 Windows 版本时，也犯了这种错误，在一个注定失败的项目上投入了太长的时间，期间微软 Access 趁虚而入。之后又重蹈覆辙，在 Quattro Pro 的新版本开发中再一次推倒重来，最后发布了一个功能少得让大家惊讶的版本。微软也曾经差点犯了同样的错误，它试图在一个叫 Pyramid 的项目中重写整个 Windows 版的 Word 软件，后来 Pyramid 被终止、抛弃并遗忘了。微软公司幸运地逃过一劫，他们的文化是从不放弃，持续完善旧版的代码，这样做的好处是总有产品可以发布，虽然偶尔会因此蒙受一定的经济损失，但是产品发布的整体战略并不受影响。

我们都是程序员。每个程序员的内心都有一个建筑师，建筑师总是在怂恿我们，把破旧的建筑夷为平地，然后重建一座宏伟的大厦。建筑师对于小打小

① 没有其他部分了。标题是在模仿梅尔·布鲁克斯的电影《世界历史：第一部分》，该电影与软件和本文内容都没有关系。

闹的修缮提不起兴趣，补补院墙、垒个花圃之类的，实在没意思。

程序员之所以总是想扔掉以前的代码从头再来，还有一个难以明说的原因。他们打心眼里觉得以前的代码写得一团糟。有意思的是，我发现他们的看法有时候是错的。程序员觉得以前的代码不好，是因为一条非常基本的编程法则：

读代码比写代码还要难。

这就是为什么代码重用如此困难的原因。解释了为什么在同一个团队里，所有的程序员都不愿意用别人写的字符串分割函数。因为与看懂旧代码的逻辑相比，自己从头写一个函数要简单得多，也有趣得多。

这条公理有一个必然的推论。随便问一个程序员他们正在使用的代码怎么样，他们会告诉你：“糟透了，简直就像是一头长毛怪物。我恨不得把它全删了重头再来。”

为什么代码糟透了？

他们会说：“就拿这个函数来说吧。它竟然有两页长！里面充斥着各种和主体功能八竿子打不着的东西！这些 API 调用有一半都不知道是干嘛用的。”

在 Borland 公司发布 Windows 平台的新版电子表格软件前，该公司富有传奇色彩的创始人菲利皮·卡恩向媒体自卖自夸，说 Quattro Pro 会比微软 Excel 好得多，因为它是从头开始重写的。所有的代码都是崭新的！他这么说就好像代码放久了会生锈一样。

认为旧代码不如新代码的观点在我看来十分荒谬。旧代码经过使用和测试，已经找出很多 bug，并且都进行了修复。有 bug 是很正常的，软件不会因为放在硬盘上放久了，自己滋生出 bug 来。恰巧相反！难道软件像是在车库里放很久就会锈迹斑斑的道奇飞梭吗？难道软件像泰迪熊玩具一样，如果不是用崭新材料做的，就显得脏兮兮的？

回头来说那个两页长的函数。没错，它只不过是一个显示窗口的简单函数，干嘛要长出那么多的“毛”，夹杂了那么多不知所云的代码。我来告诉你它们是什么：修复补丁。以前，当南希试图在没有 Internet Explorer 的机器上安装时程序会崩溃，其中一段看似多余的代码就修复了这个 bug。另一段代码修复了在内存过低时出现的崩溃 bug。还有一段代码避免了用户在软盘正在读写文件时弹出软盘导致的程序崩溃。调用 LoadLibrary 的代码是很丑陋，

但是保证了程序对旧版 Windows 95 的兼容性。

以上这些 bug 都是在数周的实际使用后被发现，然后由某个程序员花费几天的时间在开发环境中复现并且修复。对于大多数 bug，修复的代码可能只有一行，甚至只是几个字符，但是这两三个字符却是大量智力劳动和时间的结晶。

如果你把旧代码扔掉重写，其实是扔掉了所有经过实践检验的知识。扔掉旧代码意味着以前积攒的所有修复补丁都消失了，数年的编程工作都白费了。

扔掉旧代码的公司其实是在抛弃自己的市场领导地位。你们会白白送给竞争对手两到三年的时间，相信我，这么久的时间在软件世界里是致命的。

公司将陷入极其危险的境地，在几年内都只能卖旧产品，完全不能做出任何战略调整或是响应市场需求推出新功能，因为没有可以发布的新代码。与其坐以待毙，还不如让公司在这段时间里暂时歇业。

你们是在浪费大把的金钱，重写已经存在的代码。

还有别的办法吗？大家一致认为旧版网景浏览器的代码库非常差。也许是这样吧，但是你可能不知道，它在现实世界的很多计算机系统上都运行得很好。

当程序员声称公司的代码一无是处的时候（他们经常这么说），通常可能会有三个谬误。

第一，代码可能有架构问题。代码的功能划分不正确。负责网络通信的代码会主动弹出自己的对话框，而这本是 UI 层代码该做的事。这类问题通过仔细的代码移位、重构和修改接口，是可以得到解决的。可以由一位细心的程序员一次性提交所有的重构修改，这样其他人就不会受到影响。即使是非常重大的架构调整，也并不一定需要抛弃所有的旧代码。在朱诺的项目中，我们有次花了几个月时间进行代码重构，包括调整代码位置，清理冗余代码，整理出合理的基类以及在模块之间建立明确的交互接口。我们在重构已有的代码库时非常小心，并没有引入新 bug 或者抛弃已经投入使用的代码。

第二，程序员会因为代码效率不高而否定它的价值。据说网景浏览器用于渲染 html 页面的代码性能很差，但是这只涉及项目的一小部分代码，你可以优化甚至重写这部分代码，但是并不需要重写全部代码。进行性能优化的时候，1%的努力有时候能达到 99%的效果。

第三，代码可能丑陋得要命。我曾经在一个项目中见过一个数据类型，叫“该

死的字符串”。在另一个项目中，开始大家遵循用下划线_开头的名字来表示类成员变量，后来改成了更加通用的以 m_开头的命名方式。这使得一半函数以_开头，另一半以 m_开头，看起来非常别扭。坦白地说，使用 Emacs 编辑器的宏命令，只要 5 分钟就能解决这种问题，完全没有必要重写。

值得注意的一点是，每次重写代码，完全没有理由认为这一次的代码会比以前的代码还好。首先，编程团队可能有人员变动，并不一定对项目有“更深入的了解”。新团队还可能会把以前犯过的错误重犯一遍，然后引入在原有版本中不存在的新问题。

在构建大型商业软件的时候，践行俗语中的“狗熊掰棒子”是非常危险的。如果你在编写实验性质的代码，那么一旦想出一个更好的算法，就可以不假思索地扔掉上星期写的代码。你还可以重构某个类，使其更易于使用。然而，把全部代码一股脑地扔掉是非常危险而且愚蠢的举动。网景公司内部一定缺乏稳重且具有丰富经验的软件行家，缺乏对项目进度有力的监管，才会犯下自毁长城的错误。



后记

曾经的网景艺术家们终于推出了重写的、焕然一新的浏览器产品，取名谋智（Mozilla）。在这段时间里，他们的市场份额几乎全部蒸发了。

25

揭开冰山之谜

2002年2月13日，星期三

“我不知道开发团队出了什么问题”，CEO 暗自想道，“这个项目启动的时候非常顺利。刚开始的几周，开发团队加班加点地赶制出了一个很棒的原型产品；但是自那以后，开发进度变得越来越慢，举步维艰。他们准是在偷懒。”他一边这样想，一边挑了根嘉乐威钛合金球杆，顺便打发球童去取一杯冰镇柠檬汁。“或许我该开除几个懒鬼，给他们点颜色看看！”

与此同时，你和你的开发团队并不觉得事情出了问题。事实上，一切正常。项目完全在按计划进行。

别让这种事发生在你身上！为了让你的生活变得轻松一点，我决定告诉你一个关于非技术出身管理者的小秘密。其实非常简单，一旦你知道了这个秘密，就再也不用担心得罪非技术出身的领导了（除非你和他在高尔夫球杆的弹性系数问题上有分歧）。

显然，程序员和工商管理硕士在思考的时候用的是两种完全不同的语言。我一直在思考如何解决软件管理中双方交流的问题，因为我明白，很少有人能够在程序员和工商管理硕士之间充当翻译官，而这些人将会获得非常大的权力与回报。

我在软件行业摸爬滚打这么多年，几乎所有参与过的软件都可以被称为“投机性”软件。也就是说，软件不是为某一个客户量身定制的，而是为了让尽可能多的人使用而开发出来的。然而，并不是所有的软件开发者都像我这么幸运：有的程序员作为顾问，为客户定制开发某个项目；有的程序员作为驻场程序员，在某个大型机构里做财务会计工作（或者是别的工作，总之，我一直觉得驻场程序员很神秘）。

你是否注意到，这些定制化项目中，导致延期、失败和其他种种令人不悦事情的原因，都归结于这样一个通病：客户根本不知道他们想要什么。

以下是这种通病的三个表现方式。

- (1) “该死的客户老是改主意。刚开始他想要客户端/服务器架构。然后他在三角洲航空的航班上读到了有关 XML 的介绍，于是告诉我们想要用 XML。现在他又要求用‘乐高头脑风暴机器人舰队’把软件重新实现一遍。”
- (2) “我们完全按客户的要求把软件做出来了。签合同的时候，双方把产品的功能规格自顶向下地进行了详细的定义和约束，考虑到了最微末的细节。我们完全按合同上写的规格进行开发并交付。但是拿到产品后，客户显得非常沮丧。”
- (3) “我们公司可怜的销售人员和客户签订了一份固定价格的合同，合同中完全没有写明产品具体的功能细节，而客户的律师精明地在合同中加了一项条款：‘在客户满意之前，无需付款。’这让我们拥有 9 名程序员的开发团队在这个项目上花了两年的时间，最后只得到 800 美元。”

销售顾问需要把一个观念深深植入自己的脑中（必要的话用每分钟 2500 转的德瓦特电钻打进去）。

客户不知道他们想要什么。

别再指望客户知道他们想要什么了。

他们永远不可能知道自己想要的是什麼。面对现实吧。

换一种思考方式，假定你们总要做出点什么来，并且要让客户喜欢，最好还有一点惊艳的感觉。你们必须做足功课。必须拿出一个既解决问题，又让客户喜欢的设计方案。

站在客户的立场思考问题。假设你刚把公司卖给雅虎，赚了 1 亿美元，于是决定把厨房重新装修一下。于是你雇佣了一位室内设计方面的专家，让他“把厨房弄得和情景剧《威尔和格蕾丝》里的厨房一样好”。你自己并不知道具体该怎么做。是该装一个维京风格的火炉呢，还是该配备一个零下冰箱？你甚至都不知道这些词是什么意思。你只是希望室内设计师能替你打造一个最好的厨房，否则为什么雇他。

极限编程理念认为，要做到这一点，唯一的办法是把客户“请进来”，让他们的代表作为研发团队的一员，参与设计过程的每一个环节。我认为，这种做法有点“极端”了。用上面装修厨房的例子来讲，就好像室内设计师在每

个设计细节上都向客户征求意见。我觉得这些细节非常无聊，如果我对装修细节感兴趣的话，就自己去当室内设计师了。

此外，作为开发者，你并不是真的想把客户拉进开发团队吧？客户派过来的代表很可能是某个来自应收账款部门的可怜虫，因为干活太慢、毫无存在感而被发配到这里和程序员一起干活。然后你们全部的产品设计时间都会耗在解答他的各种傻问题上。

首先假定你的客户并不知道他们想要什么。根据你们对于该领域的理解，独立设计出一个产品。当然，在了解该领域的时候可以花一些时间求助于这方面的专家，但是你们要独立拿出软件产品的设计。只要对领域有足够的了解，并且用户界面做得好看一点，就会让客户满意。

现在，我来兑现刚才的诺言，告诉你们在软件的客户（或是非技术出身的管理者）所用的语言和程序员所用的语言之间进行翻译的秘密。

你可能知道，一座冰山 90% 的部分都在水下。其实软件开发也是一座冰山：开发一套漂亮光鲜的用户界面只占全部开发工作量的 10%，也就是说 90% 的开发工作是不可见的。如果考虑到一半的时间都花在修复 bug 上，那么开发用户界面的工作量所占的比重只有 5%。如果把用户界面限定为视觉可呈现的部分，即落实到像素上、能在 PPT 里展示的部分，那么这部分工作量所占的比重还不到 1%。

这还不是我说的秘密。秘密是那些不会编程的人不能理解软件开发的冰山之谜。

冰山之谜有一些非常重要的必然性推论。

重要推论一

如果你给不懂编程的人展示一个完成度只有 10% 的用户界面，他们会以为整个软件的完成度只有 10%。

我在做顾问的时候体会到了这一点，当时我们为客户的行政团队展示了一个基于网络的大型项目 demo。程序的业务代码已经 100% 完成，就等着图形设计师最终敲定字体、配色，做好酷炫的、带有凹凸效果的标签栏。我们在展

示中只用了最普通的字体以及黑白配色，页面排版也比较浪费空间，总之看上去很丑；但是 100% 的功能都是可用的，能做许多不可思议的事情。

然而在演示会议中，客户一直在抱怨屏幕上出现的图形不够好看。他们甚至没有涉及真正的用户界面层级，只停留在图像显示上。项目经理抱怨“它看起来不够漂亮”，他们满脑子想的都是这个。我们无论怎么努力，都无法让他们把注意力集中在真正的功能上。后来，我们只花了一天就把图形设计部分搞定了。看起来好像他们以为自己出钱雇佣的是画师一样。

重要推论二

如果你给一个不懂编程的人展示一个完成度为 100% 的漂亮用户界面，他们就会觉得软件差不多做好了。

不懂编程的人看到软件的时候，只看到一堆像素。如果像素看上去在产生某些变化，他们就会想：“哦，图像显示效果都做得这么完善了，想必离实现真正的功能也不是很远的吧？”

先给客户看做得很漂亮的仿真用户界面有很大的风险，虽然可以引起客户的兴趣，但是他们会以为软件已经快做好了。如果之后的一年里开发团队都在做“幕后工作”，那么其他人不会看见你们的辛苦，不会把这些工作当回事。

重要推论三

那些把站点首页做得漂漂亮亮，却只有四个页面的互联网公司，其估值要高于虽然功能强大、有 3700 年的文档记录，而网站页面却没有经过精心设计的互联网公司。

等一下，互联网公司已经完全没有价值了。别管这条了。

重要推论四

当公司规定不会编程的管理人员或客户要对项目“签字验收”的时候，提供多个版本的图形设计方案，让他们选择。

微调页面的排版方式、视觉样式、字体等，把产品标识改大或改小一点。给他们一些决定无关痛痒事情的权力，让他们觉得自己的意见举足轻重。注意不要让他们意见对项目进度造成太大的影响。优秀的室内设计师会不断向客户展示一些设计小样或是小件装饰品，让他们从中选择，但是绝不会就洗碗机的摆放位置征求客户的意见。无论客户的要求是什么，洗碗机就得放在水槽边上。没必要和客户讨论洗碗机该放在哪里，它只能放在水槽边上，没有商量的余地。可以在一些无伤大雅的方面满足客户参与设计的愿望，比如橱柜台面使用哪种材质：是意大利大理石，还是墨西哥方砖，还是挪威厚木砧板？在这个问题上，让客户说了算，即使改 200 次主意也没关系。

重要推论五

做产品演示的时候，唯一起作用的就是产品截图。一定要让截图 100% 完美。

永远不要抱有幻想，以为可以通过让客户凭空去想象产品有多酷而侥幸过关。不要假定他们会关注产品的功能特性，他们从来不会。他们只是想看到一堆漂亮的像素。

史蒂夫·乔布斯深谙这个道理，更准确地说，他是这方面的大师。苹果公司的工程师掌握了让产品在截图中大放异彩的秘诀，比如在桌面的停靠栏（dock）中加入 1024 像素×1024 像素的崭新图标，即使这是对屏幕空间的极大浪费。而 Linux 桌面开发者对于 xterms 中的半透明窗口趋之若鹜，这种设计在截图中看上去很美，用起来却非常讨厌。每次 Gnome 或 KDE 桌面推出新版产品介绍的时候，我会直接跳读到截图部分，然后说：“哦，他们把桌面背景从木星改成了土星，真酷。”我从来不关心新版桌面的功能有什么变化。



如何绕过冰山

还记得本文开头的那个 CEO 吗？他之所以不高兴，是因为在项目刚开始的时候开发团队向他展示的 PPT 很漂亮，虽然那其实只是假图片，是用 Photoshop 做的，甚至都没用到 VB。而展示过后，当整个团队埋头苦干的时候，CEO 却觉得他们什么都没做。

遇到这种情况，你该怎么办？现在你已经了解了冰山之谜，再遇到这种情况就很容易处理了。要知道，任何在幻灯片上展现的产品都只是一堆像素而已。如果可以的话，在 demo 的用户界面中，把那些未完成的部分画成未完成的样子。比如，在工具栏图标对应功能完成之前，把它画成潦草的手绘风格。在构建网络服务的时候，可以先不把未完成的功能展示在网站首页。这样随着开发的进展，人们就能看到首页的功能按钮从 3 项逐渐增加到 20 项。

更重要的是，要掌控人们对于开发进度的预期。用 Excel 文件的形式提供一份详细的开发进度规划表^①。每周发一封自我表扬式的邮件，告诉大家项目的完成度从 32% 进展到了 35%，并且肯定能赶在 12 月 25 日前发售。要确保提供详细的事实数据，以打消一切对于项目进度的疑虑。另外，不要让你的老板用嘉乐威钛合金球杆——不管你多么想让他赢，美国高尔夫球联合会早就禁用了这种球杆，因为太不公平了。

^① 见第 9 章。

26

抽象必漏定律

2002年11月11日，星期一

在人们日常使用的互联网核心，有一个像魔术一样的事物支持着整个系统的运作，叫作 TCP 协议。如果把互联网比作一座大厦，那么 TCP 就是构建大厦的砖瓦。

TCP 提供了一种可靠的数据传输方法。这句话的意思是，如果你用 TCP 协议通过网络发送一条消息，那么它总会到达，并且内容不会错乱或损坏。

TCP 有很多用途，包括获取网页和发送电子邮件。有赖于 TCP 的可靠性，你才能收到那些贪污公款的东非友人发送的激动人心的邮件^①，内容一字不乱。太好了。

与此形成对比的是一种叫做 IP 的数据传输方法，它是不可靠的。数据不一定会到达，而且到达之前就有可能已经混乱了。如果你用 IP 发送一系列消息，很可能只有一半会到达目的地，到达的消息也已经顺序颠倒或者被其他消息所替代，可能里面带有一张可爱的小猩猩照片，或者更可能只是一堆不可读的文字，就好像那些从台湾发来的垃圾邮件标题一样。

互联网的神奇之处就在于：TCP 是构建于 IP 之上的。也就是说，TCP 负责使用一种不可靠的工具，来实现可靠的数据传输。

为了描绘为什么 TCP 非常不可思议，我们用一个现实世界的例子来类比，尽管可能会有点滑稽。

假设我们用某种方法，能够把演员从美国东海岸的百老汇送到西海岸的好莱坞，就是把他们塞到小汽车里然后横穿美洲大陆。这些小汽车中有一部分发

① 指诈骗邮件。——译者注

生了车祸，里面的演员不幸丧生。有时候演员在路上喝醉，剃了光头或是在鼻子上刺了青，破坏了形象，没法在好莱坞演戏了。演员们到达目的地的顺序也通常和出发的顺序不同，因为选择的旅行路径不同。下面再来想象一种新的服务，叫做好莱坞直通车，它能把演员送到好莱坞，并且保证每个人：(1) 都能到达；(2) 按出发的顺序到达；(3) 毫发无损。不可思议的是，好莱坞直通车并没有运送演员的特殊方法，还是得把他们塞进小汽车，然后驾车横穿美洲大陆。好莱坞直通车兑现承诺的方法，是在每一个演员到达的时候检查其是否毫发无损。如果演员出现任何问题，就打电话给总部，让他们再送一个一模一样的演员过来。如果演员抵达的顺序有误，好莱坞直通车就会按出发顺序把他们排好。如果一个大型飞碟在飞往 51 区的路上在内华达州高速公路上坠毁，导致道路无法通行，那么所有走这条路的演员都会绕道亚利桑那州，而好莱坞直通车根本不会告诉加州的电影导演们路上发生了什么。导演们只会感觉演员到达的时间有点迟，完全不知道路上有飞碟坠毁。

这就是 TCP 的神奇之处。计算机科学家通常把这称作抽象：隐藏内部复杂的机制，提供简化的使用接口。实际上，计算机编程的很多工作都涉及对于抽象的构建。字符串库是什么？它是一种方法，让人们感觉操作字符串就像操作数字一样容易。文件系统是什么？硬盘实际上是一堆高速旋转的磁性碟片，上面的特定位置存储了许多比特的信息，而文件系统隐去了这些细节，让人们以更符合认知的方式管理数据，即操作由一层层文件夹组成的分层系统，最里面的文件夹包含了文件，而文件中存储的是一个或多个字节字符串。

回到 TCP。刚才为了简化叙述，我撒了个小谎，有些读者可能因为这个而怒发冲冠了。我刚才说 TCP 保证了消息会抵达目的地，其实不然。如果你养的宠物蛇咬穿了连在计算机上的网线，导致所有的 IP 包无法通过，那么 TCP 再厉害，消息也绝对发不出去。如果你得罪了公司的系统管理员，他们为了报复你，把你的数据链接放在一个过载的网络集线器上，那么你发出的 IP 包有很多都无法通过。虽然 TCP 能工作，但是一切都将变得异常缓慢。

这就是我在标题中所说的抽象必漏。TCP 试图提供一个严密的抽象，完全屏蔽底层不可靠的网络对用户的影响，但是有的时候，网络还是会从抽象的缝隙泄漏出来，让用户感觉抽象也无法保护自己。上面这个例子佐证了一个定律，我称之为抽象必漏定律：

所有重要的抽象，都具有某种程度的不严密性。

抽象难免会失效。有时候程度较轻，有时候则影响严重。有时候只是很小的

跑冒滴漏，有时候则会把事情搞砸。只要有依赖抽象的地方，就有可能失效。下面举几个例子。

- 有时候即使是遍历二维数组这么简单的事情，也会因为遍历方向的不同（横向或纵向）而产生巨大的性能差异。这取决于“木质的纹理”——某个方向的遍历可能会导致内存页错误显著增多，而内存页错误会降低计算速度。即使是汇编语言的程序员，也可以假定拥有很大一块内存地址空间，但是虚拟内存的出现意味着这种抽象非常不严密，经常会出现内存页错误，导致某些内存读取操作时间比平均时间多出几个纳秒。
- SQL 的设计初衷是对查询数据库的过程性步骤进行抽象，使用者只要说明想要什么，数据库就能自己生成查询的过程性步骤。但是在某些情况下，某些 SQL 查询要比与之在逻辑上等同的查询慢几千倍。一个著名的例子是，某些 SQL 服务器在查询条件为“where a=b and b=c and a=c”的时候，要比查询条件为“where a=b and b=c”的时候要快得多，而两种查询结果的集合是一样的。本来使用者不需要关心怎么查，只要说明查什么就可以了，但是有的时候抽象会失效，导致可怕的性能问题。这时候你必须在查询计划分析器上打断点调试，看看到底是哪步出了问题，然后找出提高查询速度的方法。
- 虽然 NFS 和 SMB 之类的网络库能让你把位于远程机器上的文件“看作”本地文件，但有时候网络连接会变得非常慢，甚至不可用，这时远程文件就不再像本地文件了，程序员必须编写代码来处理这些情况。“远程文件和本地文件一样”这种抽象经常出纰漏^①。再举一个 UNIX 系统管理员遇到的实际例子。如果把用户的主目录放在以 NFS 格式加载的驱动器里（一层抽象），然后创建 .forward 文件，把这些用户的邮件转发到另一个地方（另一层抽象），那么当 NFS 服务器不可用时，新电子邮件将不会被转发，因为无法找到 .forward 文件。由于抽象的纰漏，有些消息实际上被丢弃了。
- C++ 字符串封装类本该可以让你把字符串当作基本数据类型。这些封装类试图使用抽象把字符串很难处理^②这一事实隐藏起来，让你像使用整数一样轻松地使用字符串。几乎所有的 C++ 字符串封装类都重载了 + 运算符，这样你就可以用 s+"bar" 来连接两个字符串。但是，无论 C++ 的设计者如何努力，世界上还是没有有一个 C++ 字符串封装类，能让你在代码里写 "foo"+"bar"，因为 C++ 的字符串字面量总是 char* 类型，而不是字符串类型。这种抽象的漏洞，你想堵都堵不上。（有意思的是，C++ 的演化史可

① 参见第 17 章。

② 参见第 2 章。

以被演绎为一段试图堵上字符串抽象漏洞的历史。我一直想不明白，为什么他们不增加一种原生的字符串类。)

- 下雨的时候不能把车开得太快。虽然你的车拥有雨刷、前灯、顶棚和暖气，你本不该再担心外面在下雨这个事实(这些东西试图通过抽象来屏蔽掉天气的影响)，但是实际上，你不得不担心路面湿滑，而且有时雨势很猛，能见度不是很高，所以雨中行车必须放慢速度。由于抽象必漏定律，天气的影响并不能完全地被抽象屏蔽掉。

抽象必漏定律值得引起大家的重视，原因之一是，它意味着抽象其实并没有按预想的那样简化我们的生活。教 C++ 编程的时候，如果我能跳过 `char*` 和指针运算这些内容，只讲 STL 中的字符串封装类就好了。但如果这样，总有一天他们会写出 `"foo"+"bar"` 这种代码，然后发生奇怪的事情，我还是得停下来，告诉他们关于 `char*` 的所有知识。又或许有一天，他们试着调用一个 Windows API 函数，文档上说要传入一个 `OUT LPTSTR` 类型的参数。如果不对 `char*`、指针、Unicode、`wchar_t`、`TCHAR` 头文件以及其他所有会出纰漏的抽象有所了解，他们就不可能明白怎么去调用这个函数。

教 COM 编程的时候，如果我能只讲如何使用 Visual Studio 的引导式对话框以及所有代码自动生成功能就好了。如果这样，一旦事情出了差错，他们就不可能知道问题出在哪里，更别说如何调试和修复问题了。到那时，我还是得跟他们讲 `IUnknown`、`CLSID`、`ProgID`……天哪！

教 ASP.NET 编程的时候，如果我能只告诉他们，点点鼠标，写一些响应用户点击事件的代码就可以了，那该多好。ASP.NET 通过抽象，屏蔽了处理点击超链接 (`<a>`) 的代码和处理点击按钮的代码之间的差异。但问题来了，ASP.NET 的设计者无法屏蔽如下事实：在 HTML 里，不可能通过点击超链接来提交表单。他们通过生成一段 JavaScript 代码，并挂到超链接的处理句柄上来解决这个问题，但是这种抽象方式是有漏洞的。一旦用户禁用了 JavaScript，ASP.NET 应用就无法正常工作。如果程序员不明白 ASP.NET 屏蔽了什么，就不可能知道出了什么问题。

抽象必漏定律还意味着，每当某种号称能大大提高编码效率的代码生成工具出现的时候，总会有人说：“先学会手动编写代码，然后再使用生产力工具以节约时间。”代码生成工具总是试图用抽象来屏蔽某些细节，但和所有抽象一样，它也存在一定的漏洞。要处理这些问题，唯一的方法是了解抽象的工作原理以及抽象屏蔽了哪些细节。也就是说，我们可以用抽象节省工作的时间，但无法节省学习的时间。

从以上论述中，我们得出了一个看似荒诞却十分正确的结论：随着我们拥有的编程工具越来越先进，抽象层次越来越高，成为一名熟练的程序员也变得越来越难了。

我第一次在微软实习的时候，写过一个在苹果机上运行的字符串库。任务非常典型：实现一版 `strcat` 函数，并且返回指向新字符串末尾的指针。一共只用了几行 C 代码。所有的东西都是直接取自一本 C 编程语言的薄书，作者是布莱恩·柯尼汉和丹尼斯·里奇（K&R）。

如今，在开发 CityDesk 时，我需要了解 Visual Basic、COM、ATL、C++、InnoSetup、Internet Explorer 内部原理、正则表达式、DOM、HTML、CSS 以及 XML。和 K&R 书中的内容相比，这些都是高层的工具，但是我还是得了解 K&R 的理论，不然就完了。

10年前的人们可能幻想，到这个时候，新的编程范式将让编程变得更加简单。诚然，这么多年来人们所做的抽象工作，确实能够让开发者处理复杂度更高的软件开发问题，这些问题在 10 到 15 年前是不存在的，比如 GUI 编程和网络编程。尽管我们拥有了许多优秀的工具，比如现代基于面向对象思想的编程语言，能够让我们很快地完成许多开发工作，但是一旦抽象出了纰漏，用两周的时间才能找到并解决问题。当你需要聘用程序员来做 VB 编程的时候，选择只会 VB 的人是不够的，因为每当 VB 这层抽象出了问题，他们就会束手无策。

抽象必漏定律的存在拖慢了软件开发的进度。

27

帕默斯顿勋爵谈编程

2002 年 12 月 11 日，星期三

曾几何时，你只要读彼得·诺顿的书^①，就能了解所有关于 IBM-PC 的编程知识。在最近的 20 年里，世界各地的程序员孜孜不倦地工作，在 IBM-PC 架构上建立了一层又一层的抽象，让编程变得更容易，使架构变得也更强大。

但是抽象必漏定律^②告诉我们，虽然人们建立各种抽象的目的是降低编程的难度，但要想成为一名优秀程序员，所需要掌握的知识总量却在持续增加。

要在一个编程领域真正达到精通的程度，需要好几年的时间。虽然许多天资聪颖的少年可以一周学会 Delphi，下一周学会 Python，再下一周学会 Perl，然后自以为精通这些语言，但是他们完全不了解自己错过了多少真正重要的知识。

早在 ASP 和 VBScript 问世之初，我就开始使用它们了。VBScript 是世界上最精简的编程语言，ASP 编程基本上只需要掌握 5 个类，常用的只有 2 个。而直到今天，我才真正感觉自己找到了 ASP/VBScript 程序的架构之道。我终于确定了应该把数据库访问的代码放在哪里，了解了使用 ADO 获取数据集的最佳方法，以及分离 HTML 和代码的最佳实践，等等。我终于用正则表达式替代了一次性的字符串处理函数。直到上个星期，我才学会从内存中得到 COM 对象，然后重新编译（在不重启整个网络服务器的情况下）。

Fog Creek 软件公司太小，养不下太多的专业人才，所以在为我们用 ASP 和 VBScript 写的产品 FogBUGZ^③开发定制化的安装工具^④时，我需要凭借自己多年的 C++ 和 MFC、Windows API 经验，以及不错的 Corel PHOTO-PAINT

① 彼得·诺顿，*Inside the IBM PC: Access to Advanced Features and Programming*。

② 见第 26 章。

③ 参阅 <http://www.fogcreek.com/FogBUGZ>。

④ 参阅 <http://www.joelonsoftware.com/news/20021002.html>。

使用技巧,才能在安装引导界面的一角创建一个漂亮的图片。为了能让 FogBUGZ 完美支持 Unicode,我需要用 C++和 ATL 写一个小小的 ActiveX 控件,而这需要多年的 C++和 COM 经验,以及在开发 CityDesk 代码过程中花费大概一周时间学到的字符编码知识。

我只用 3 分钟,就调试并修复了一个只在 NT 4.0 上出现的奇怪 bug,这是因为我知道如何使用 VMWare,在 VMWare 中设置好了一台干净的 NT 4.0 虚拟机,了解如何用 Visual C++进行远程调试,并且知道通过查看 EAX 寄存器获得函数返回值的技巧。换了另一个不知道这些知识的人,可能需要一个小时乃至更长的时间才能解决同样的问题,而我已经掌握了很多知识,学习过程始于 1982 年我的第一台 IBM-PC 机和那本诺顿的书。

漏洞百出的抽象意味着如今的编程学习曲线是呈指数形态的:日常编程知识的 90%可以在一周内学会,但是剩下 10%的知识欠缺可能要花几年的时间才能弥补。这 10%的知识决定了经验丰富程序员的价值,令那些把“无论你让我干啥,我找本书学学就能干”挂在嘴边的人相形见绌。一支开发团队需要很多经验不足但是能产出大量代码的程序员,更需要几个经验丰富的程序员去处理真正困难的问题,否则团队无法完成任何项目。

计算机编程领域可以细分为许多个小领域,每个领域都需要巨大的知识储备才能称得上精通。我个人对三个领域最为熟悉:

- MFC/C++/Windows

- VBScript/ASP

- Visual Basic

可以说,这些领域都属于 Windows 编程的范畴。我也写过 UNIX 代码和 Java 代码,但是不多。我对于 Windows 编程的精通不仅体现在对基本技术的掌握,而且体现在对整体支持架构的了解。我可以自豪地宣称自己精通 Windows 编程,因为我还知道 COM、ATL、C++、80×86 汇编、Windows API、IDispatch (OLE 自动化)、HTML、DOM、Internet Explorer 对象模型、Windows NT 和 Windows 95 内部原理、局域网管理器和 NT 网络安全性 (ACE、ACL 等)、SQL 和 SQL Server、Jet 和 Access、JavaScript、XML,以及一些关于直角三角形弦平方的有趣事实。当 VB 的 StrConv 函数无法如我所愿时,我可以不假思索地创建一个 COM 控件,通过 C++和 ATL 调用 MLang 函数。我花了很多年的时间才对这些编程工具达到如今的熟悉程度。

除此之外,还有很多其他的编程领域。比如在 BEA WebLogic 上进行开发,

需要知道 J2EE、Oracle 和各种我叫不上名字的 Java 相关知识。在苹果机上进行开发，需要掌握 CodeWarrior、MPW、从 System 6 到 X 的工具栏编程、Cocoa、Carbon，甚至还有 OpenDoc 这种已经过时的好东西。

但是，很少有人能精通超过两个领域的编程知识，因为不下几年的苦功去学习，是很难在某个编程领域登堂入室的。

必须加强学习。

编程面试中，因为缺乏某些方面（比如 Win32、J2EE、Mac 编程或者别的什么）的经验而遭到拒绝，总是令人不悦的。还有某些面试官，自己见了 MSMQ 都不认识，就敢问面试者“你有没有 10 年的 MSMQ 经验”。

对 Windows 编程不够了解的人，可能会以为 Win32 只是一个函数库，和其他的库一样，只要翻翻手册就能在需要的时候调用相应的函数。这些人可能认为掌握了编程基础知识，比如专业的 C++ 技能，就掌握了 90% 的知识，剩下 10% 和 API 有关的知识可以在几周内习得。对于持这种想法的人，我想说：对不起，时代变了，现在这个比例倒转过来了。

如今，很少有人直接使用底层的 C 算法操作字节。大部分人在编程时都把全部的时间花在调用 API 上^①。如果一个 C++ 高手对业务 API 一无所知，那么他（她）只拥有进行日常编程工作所需知识的 10%。经济景气的时候，这没什么关系^②。你有一份工作，雇主也乐于付钱让你学习具体业务平台的知识。但是当经济不景气，每个工作岗位都有 600 个人竞争的时候，雇主就有权挑选并优先录用熟悉某个开发平台的面试者，比如那些知道 4 种在 Visual Basic 中用 FTP 收发文件的方法，并且能说出这些方法优缺点的人。

编程领域之间的巨大差异导致人们经常对于哪个领域更好而发生无谓的争执。有人在我博客的留言板上匿名留下了一条自以为是的评论：

这更让我庆幸自己生活在“自由世界”里，我说的“自由”是“言论自由”的“自由”，免于应付安装程序以及注册表这类东西的“自由”。

我觉得这个人想说的是，在 Linux 世界里，他们不用写安装程序。我不想让你失望，但是不得不说，Linux 世界里提供的安装部署方案同样复杂：imake、

① 参阅 <http://www.joelonsoftware.com/articles/fog0000000250.html>。

② 参阅 <http://www.joelonsoftware.com/articles/fog0000000050.html>。

make、配置文件等，发布应用的时候还得附带一个 20KB 的安装说明文件，里面是各种自作聪明的指令，比如“你以后可能需要 zlib（这是什么？）”或者“安装时间较长，你可以到外面弄点 runt 回来”（runt 大概是种糖果吧，我猜的）。至于配置文件，Linux 没有像注册表这样单独的、巨大而有序的键值对存储结构，取而代之的是几千种不同的配置文件格式，各个程序的配置不能通用，命名也无统一规范，从 `.whateverrc` 到 `foo.conf`，散落在文件系统的各种地方。想修改 emacs 的配置，就必须会 lisp 编程，想修改 shell 的配置，就需要懂 shell 脚本编程，而且每种 Linux shell 的脚本还都略有区别，像方言一样。如此种种，不胜其烦。只熟悉一个编程领域的人就像井底之蛙，每当听见别人抱怨另一个领域的艰难，就越发觉得待在自己的领域里无比轻松愉快。其实根本没这回事，你只是太熟悉自己的领域，早就习惯了其中的艰难。这些编程领域经过多年的发展，已经过于庞大和复杂，无法进行比较了。就像帕默斯顿勋爵说的：“施勒斯维格·霍斯坦因问题太复杂了，整个欧洲只有三个人懂。其中一个普林斯·阿尔伯特，死了。另一个是一名德国的教授，疯了。我曾经是第三个懂得这个问题的人，但现在全忘了。”

这就像如今的软件世界，庞大、复杂、多元化。我看到有人在博客里写“微软不擅长做操作系统”之类的言论。坦白地说，这种言论很无聊。对于几十年来上千名程序员编写的几百项重大功能中的上千万行代码，任何一个人都没有足够的心智去理解其中的大部分，怎么可能用寥寥数语概括？我并不是袒护微软，只是说这种井底之蛙的高谈阔论，是如今网络上最没有价值的信息之一。

我一直在思考如何把一个应用部署在 Linux、苹果系统和 Windows 系统上，并且避免为了支持 Linux 和苹果系统而付出不成比例的成本。为达到这个目的，需要使用跨平台的解决方案。

Java 试图解决跨平台的问题，但是 Sun 公司缺乏对图形化界面的认知，无法提供和各个原生应用类似的优秀用户界面。就好像《星际迷航》中的外星人，它们透过望远镜观察地球，了解了人类食物的样子，但是并不知道，食物还应该有其独特的味道^①。Java 程序虽然在正确的位置上有菜单，但是键盘快捷键和其他 Windows 程序完全不同，而且 Java 程序的标签对话框看起来有点吓人。此外，无论你怎么定制，都不能让 Java 程序的菜单栏和 Excel 的菜单栏一模一样。为什么？因为 Java 没有提供一种好的方式，让开发者在抽象失

① “The Squire of Gothos”，《星际迷航》（Paramount Studio，1967 年 1 月 12 日）。完整内容参阅 <http://www.voyager.cz/tos/epizody/19squireofgothostrans.htm>。

效的时候调用原生的工具方法。例如在 AWT 里编程的时候，没办法获得窗口的 `HWND` 句柄，不能调用微软的 API，而且肯定不能截获 `WM_PAINT` 消息然后重写绘图方法。Sun 公司说得很明确，如果你试着这么做，那么你不是好孩子。你被宠坏了，去你的吧。

软件行业曾经出现过许多使用 Java 构建用户界面的著名产品，结果都失败了（比如 Corel 公司的 Java 办公套件和网景的 Javagator）。之后大部分人知难而退，离开了这个软件领域。Eclipse^① 独自研发了完全基于原生控件的窗口库，这样在写 Java 代码的时候，就可以有一个接近原生界面体验的图形化界面。

Mozilla 的工程师们研发了自己的跨平台解决方案 XUL。到目前为止，它给我留下了相当不错的印象。借用前面的比喻，Mozilla 终于做出了一点食物的味道。比如我偏好使用快捷键 `Alt+Space N` 来最小化窗口，Mozilla 的产品就支持这种操作方式。虽然他们早该实现这个功能，但是晚实现总好过不实现。

Lotus 和 Lotus 1-2-3 的缔造者米驰·卡普，决定在他的下一个产品中使用 `wxWindows` 和 `wxPython` 来解决跨平台支持问题^②。

XUL、Eclipse 的 SWT 以及 `wxWindows`，哪个更好？我不知道。每一个都是非常庞大的领域，我没有精力去身体力行地使用然后评判。阅读教程是远远不够的。必须要花上一两年的时间，真刀真枪地使用一种开发工具，才能判断它足够好，或是发现无论怎么尝试，都无法做出原汁原味的用户界面。可惜在大部分项目里，必须在写代码之前就决定使用哪种开发工具，而这恰恰是参考信息最少的时候。在我以前的一份工作中，整个开发团队都必须忍受一套设计非常糟糕的架构，原因是最初的开发者一边写代码，一边自学 C++ 和 Windows 编程。从部分最老的代码中，可以看出编程者当时完全不理解事件驱动编程。核心的字符串类（我们当然自己写了一套字符串类）是不折不扣的反面典型，几乎涵盖了 C++ 类设计的所有错误。后来，我们清理并且重构了很多旧代码，但是其影响还是挥之不去。

目前我的建议是：在启动一个新项目之前，必须保证至少有一名架构师对于决定使用的语言、类库、API 和开发平台拥有数年扎实的功底。如果可以选择开发平台，要使用团队最擅长的平台，即使它不是最流行的或者名义上开发效率最高的平台。当设计抽象或编程工具的时候，要舍得下苦功，保证这些设计没有漏洞。

① 参阅 www.eclipse.org。

② 参阅 <http://blogs.osafoundation.org/mitch/000007.html>。

28

评估机能失调

2002年7月15日，星期一

“感谢致电亚马逊在线，请问您需要什么帮助？”然后“咔”地一声，通话断了。真讨厌。等了10分钟才接通的客服电话，就这样莫名其妙地被挂断了。

我大概知道是怎么回事了。麦克·戴西说过，亚马逊客服代表的业绩是用每小时接电话的次数来衡量的^①。通过提前挂断客户的电话，可以增加每小时接电话的次数，进而提升业绩水平。

你觉得这只是个别现象吗？

杰夫·维特森接管 Gateway 公司后，为了节约客户服务热线的成本，制定了一套新规定。“客服代表如果和同一个客户聊天超过13分钟，就拿不到月度奖金，”卡特里娜·布鲁克写道^②，“结果客服代表开始不择手段地缩短与客户的通话时间：假装线路故障、主动挂断，甚至经常不惜成本地给用户寄送新零件、新电脑。Gateway 公司一度行业领先的客户满意率，就这样毫无悬念地跌落到行业平均水平以下。”

每一次试图对脑力劳动者的工作成果进行定量评估，似乎都会以失败而告终，并且对团队造成巨大的破坏。罗伯特·奥斯丁把这种现象称为评估机能失调。他在著作《机构中的绩效评估和管理》^③中，对此作了出色而细致的研究。管理者喜欢建立各种评估系统，还喜欢根据评估系统反映的绩效状况来实施奖惩措施。但是在缺乏100%监督的情况下，员工通常会“上有政策，下有对策”，只关注提高绩效评估中涉及的数值，而忽略了工作的真正价值。

① Mike Daisey, *21 Dog Years: Doing Time @ Amazon.com*.

② *Business 2.0*, April 2000

③ Robert Austin, *Measuring and Managing Performance in Organizations*.

软件公司通常会奖励：(1) 写了很多代码，(2) 修正了很多 bug 的程序员。要想获得高绩效，最好的方法就是先提交有许多问题的代码，然后再逐个修复，而不是在写代码的时候多用点心，争取一次写对。有的公司还有进一步的应对措施，对引入 bug 的程序员实施惩罚，意在提高代码质量。但是程序员将计就计，隐瞒自己导致的 bug 或者不向测试人员提交新代码，这样又给 bug 的搜查工作增加了难度。总之，绩效评估系统很难奏效。

全球 500 强企业 CEO 的薪资构成通常是基本工资加上公司股权。股权通常价值数千万甚至上十亿美元，相比之下，基本工资完全可以忽略。因此，CEO 会尽力让股票价格上涨，即使这终将导致公司破产倒闭也在所不惜（这个月里与此相关的消息已经上了好几次头条）。他们急功近利的做法会让股价短时间内上涨，然后在高点抛售。公司的薪资调整部门反应迟缓，但是他们最近想出了让执政官离开公司之后才能卖股票的“聪明”主意。太好了，这激励了 CEO 暂时性抬高股价，然后辞职。绩效评估系统还是难以起效。

你可以不相信我的话，去读读奥斯丁的书，你就会明白为什么在无法全面监督雇员行为的情况下（也就是在几乎所有情况下），评估功能失调是不可避免的。

我早就指出员工激励计划并不是个好主意^①，奥斯丁进一步佐证了这一观点，他向我们证明了公司甚至无法评估员工的绩效，所以基于绩效的激励计划就更不靠谱了。

① 参阅第 21 章。

第三部分

乔尔语录：中心明确的胡思乱想

-
- 29 里克·查普曼在“寻找愚蠢”
 - 30 这个国家的狗要做什么工作
 - 31 普通程序员如何改善团队工作方式
 - 32 两个故事
 - 33 巨无霸汉堡 vs. “原味主厨”
 - 34 凡事没有看上去的那么简单
 - 35 为NIH综合症说两句
 - 36 企业发展战略系列一：本杰瑞 vs. 亚马逊
 - 37 企业发展战略系列二：先有鸡还是先有蛋
 - 38 企业发展战略系列三：让我回到过去
 - 39 企业发展战略系列四：膨胀和二八法则
 - 40 企业发展战略系列五：开源软件经济学
 - 41 祸不单行的一周
 - 42 微软是如何输掉API战争的

里克·查普曼在“寻找愚蠢”^①

2003年8月1日，星期五

在我熟悉的所有高科技公司中，都存在着两种势力的对抗：极客和西装族。

在开始阅读这本由西装族中的翘楚、营销奇才里克·查普曼的大作之前，请先听我说说极客是怎么想的。

请配合我一下，好吗？

请根据你心目中对极客的印象，想象一个最为典型的极客人物：喝着功能饮料，吃着中式快餐，玩着电子游戏，看着 Slashdot，把 Linux 命令行玩到极致的家伙。你可以发挥想象力，把他想成小个子或是胖胖的家伙都可以，但是可以肯定，他不是那种趁着感恩节假期回家看望妈妈时，会和当年高中同学一起玩橄榄球的人。由于他只是一个模式化形象，所以把性别假定为男应该没什么问题吧。

我们的模式化程序员会这样想：“微软的产品不怎么样，但是它的营销部门很强大，所以才有那么多人买微软的产品。”

问起他对自己公司营销人员的看法，他会说：“我们公司的营销人员特别傻。昨天我和一个销售部门的蠢姑娘在休息室大吵了一架。吵了 10 分钟后我发现，她根本不知道 802.11a 和 802.11b 的区别是什么，岂有此理！”

年轻的极客，你眼中的营销人员平常在干什么呢？“我不知道，他们每天除了麻烦我修改愚蠢的产品规格清单，大概就是和客户打打高尔夫之类的吧。如果我说了算，就把他们通通开除。”

① 本章是里克·查普曼的新书《寻找愚蠢》(In Search of Stupidity: Over 20 Years of High-Tech Marketing Disasters) 的序言。

一个叫杰弗瑞·塔特的好心人曾发表了一系列叫做 Soft-letter 100 的年度榜单，上面列举了一百家当年最大的个人电脑软件发行商。下面是 1984 年的榜单^①。

排 名	公 司	年 收 入
#1	Micropro 国际	60 000 000 美元
#2	微软集团	55 000 000 美元
#3	Lotus	53 000 000 美元
#4	Digital Research	45 000 000 美元
#5	VisiCorp	43 000 000 美元
#6	Ashton-Tate	35 000 000 美元
#7	Peachtree	21 700 000 美元
#8	MicroFocus	15 000 000 美元
#9	Software Publishing	14 000 000 美元
#10	Broderbund	13 000 000 美元

当时微软排在第二位，但是有好几家公司的年收入和它相差无几。

我们再来看 2001 年的榜单。

排 名	公 司	年 收 入
#1	Microsoft 集团	23 845 000 000 美元
#2	奥多比	1 266 378 000 美元
#3	Novell	1 103 592 000 美元
#4	Intuit	1 076 000 000 美元
#5	欧特克	926 324 000 美元
#6	赛门铁克	790 153 000 美元
#7	Network Associates	745 692 000 美元
#8	思杰	479 446 000 美元
#9	Macromedia	295 997 000 美元
#10	Great Plains	250 231 000 美元

哇！请注意，除了微软，1984 年排在前十位的其他公司都不见了。另外请注意一点，微软的年收入比第二名高出一大截，成为毫无争议的第一。如果把微软的苏打饮料预算发给奥多比（Adobe）公司，那么后者的收入就将翻一倍。

微软独霸了个人电脑软件市场，收入占排名前 100 家公司收入总和的 69%。

① 来源：Soft letter，杰弗瑞·塔特，2001 年 4 月 30 日，17:11。

这才是微软的现状。

微软能取得今天的地位，仅仅是因为它的营销部门很强大吗（就像想象中的极客所说）？还是说它的霸主地位是某种非法垄断行为的恶果？（如果你倾向于第二种解释，我不禁要问：微软是如何取得垄断地位的？用垄断来解释垄断，逻辑上说不通。）

而里克·查普曼提出了另一种观点。他指出，答案其实很简单：微软是当年那批公司中唯一从未犯过愚蠢致命错误的公司。有人说这归功于微软精明的商业头脑，还有人说它只是运气好。无论你相信哪种解释，都无法回避一个事实，那就是微软犯下过的最大错误，仅仅是创造了一个会跳舞的曲别针^①。说实话，它有那么糟吗？我们对 Office 助手嘲笑一番，然后关掉它，继续使用 Word、Excel、Outlook 和 Internet Explorer，几乎每天、每分钟都离不开这些软件。但是看看其他曾经取得市场领导地位，如今却跌落谷底的公司，你都能指出一到两个重大决策失误，好比掌舵者因为错误的判断和操纵最终导致轮船撞上了冰山。Micropro 在重写打印机架构上浪费了太多的时间，没有升级其旗舰产品 WordStar。Lotus 浪费了一年半的时间，非要让 123 运行在 640kb 内存的机器上。等 Lotus 做完的时候，Excel 已经问世，而 640kb 机器已成明日黄花。Digital Research 公司对 CP/M-86 系统漫天要价，失去了成为标准 PC 操作系统的机会。VisiCorp 在法庭上把自己告垮了。Ashton-Tate 则不遗余力地激怒 dBase 开发者，破坏了本来就很脆弱的开发生态圈。要知道，作为一家平台提供商，失去了开发者的支持就等同于失败。

我是一名程序员，所以也倾向于把这些愚蠢的错误推到营销人员头上。这些错误几乎都围绕着一个事实：非技术出身的商业人员对最基本的技术事实缺乏了解。当百事可乐的营销大师约翰·斯库利负责开发苹果公司的 Newton 产品时，他不知道以当时的技术条件，手写识别是不可能完成的功能，而这是一个所有计算机科学专业的学生都知道的技术事实。就在同一时刻，比尔·盖茨在做另一件事：把程序员抓去开会，请求他们开发一套在所有微软产品中通用的富文本编辑控件。如果吉姆·曼吉（那个让工商管理硕士接管 Lotus 公司的西装族）出席了这些会议，他会干瞪眼：“富文本编辑控件是什么东西？”他从来不会有占领技术制高点的想法，因为他不 grok^② 技术，事实上，他可能连 grok 这个词都听不懂。

① 微软 Office 助手中的大眼夹。——译者注

② “grok”就是“懂”。——译者注

也许我个人存在偏见，但我认为一家软件公司要想成功，必须要有一名程序员执掌大权。到目前为止，我的假说和实际情况相符。不过也有很多愚蠢的公司战略决策出自程序员群体。网景公司作出的重大决定，即重写网景浏览器的代码而不是在旧版代码基础上改进，使其浪费了数年的互联网时间，而这段时间里，其产品的市场占有率从 90% 跌到了 4%。这就完全是程序员的主意。当然，网景的管理者不懂技术且经验不足，根本无法意识到为什么重写代码是个坏主意。时至今日，还有大批程序员为网景重起炉灶的做法辩护。“旧版代码确实烂透了，乔尔！”是啊，也许吧。持这种意见的程序员有着对整洁代码的追求，这值得敬佩，但是我绝不让他们参与任何商业决策，因为很明显，在他们眼里干净的代码要好过无法发布的软件产品。

所以，我赞同里克的部分观点：如果你想在软件行业取得成功，必须有一支完全理解并热爱编程的管理团队，他们还必须理解并热爱商业运作。虽然这样跨界的领袖人才打着灯笼也难找，但只有如此，才能避免里克在书中列举的种种致命错误。所以请读读这本书，轻松一下。如果发现你们公司的领导也在犯傻，还是趁早把简历准备妥当，找一个位于雷德蒙德（微软总部所在）的住处为好。

这个国家的狗要做什么工作

2001年5月5日，星期六

我们太天真了。

我们欺骗了自己，认为贝佐斯只是把利润进行了再投资，所以亚马逊的财务报表上才没有这部分钱。

去年的这个时候，新闻中出现了第一波互联网公司倒闭的消息：Boo.com、Toysmart.com……之前“快速成长”的互联网思维已经不管用了。500多个三十出头、身着 Dockers 牌休闲服的互联网公司创始人用亲身经历告诉世人，简单地复制杰夫·贝佐斯已经不再是一种可行的商业模式了。

最近的几周里，Fog Creek 公司的气氛安静得有些奇怪。我们的新产品 CityDesk 即将完成。在谈论 CityDesk 之前，我想先和你们谈谈狗粮。

狗粮？

上个月，萨拉·科贝特发表了一篇专栏文章^①讲述“迷失男孩”的故事，他们是一群 8~18 岁的苏丹难民，背井离乡，被迫踏上了一条长达上千英里的征程——从苏丹到埃塞俄比亚，再到苏丹，最后来到肯尼亚。半数的人在路上死于饥饿、干渴，或命丧短吻鳄之口。时至隆冬，有少数幸存者获救，被安置到美国北达科他州法戈市等地。在从机场到新家的路上，一个男孩问道：“这片灌木丛里有狮子吗？”后来，在超市里还发生了这样的事：

皮特碰了碰我的肩膀。他手里拿着一罐普林娜牌狗粮：“萨拉小姐，请问这是什么？”在他身后，超市货架上的宠物食品几乎堆到了天

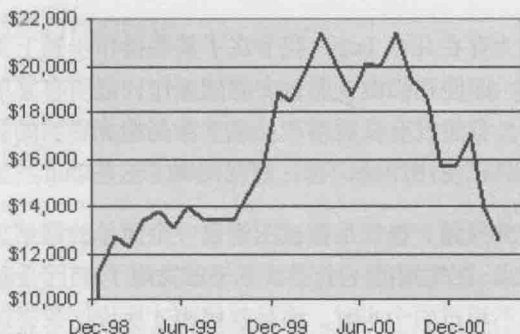
①《纽约时报周刊》2001年4月1日刊登萨拉·科贝特的文章“The Long Road From Sudan to America”。

花板。“哦，那是给狗吃的食物。”我话一出口，就有些担心，一个喝了8年粥的人会对作何感想。“啊，我明白了。”皮特似乎得到了满意的答复，并把那罐狗粮放回了货架。他推着购物车，没走几步就又转过身来，脸上写满了疑惑。他问：“那么，这个国家的狗要做什么工作？”

是啊，皮特，你没听错。法戈市有足够的食物，狗不用干活也有狗粮吃。

这是令人沮丧的一年。

今年刚开始的时候让人很愉快。互联网行业一派欣欣向荣，大家排着队涌入B2B、B2C和P2P等领域，那感觉就好像一家人高高兴兴地去Krispy Kreme采购甜甜圈，为周日的郊游做准备一样。但这还不是最有意思的，最有意思的是，在那些最糟糕的商业计划惨败的时候，人们还在嘲笑这些公司的股价从316美元一下子跌到了3/16美元。这下完蛋了吧，让你再鼓吹新经济！人们在幸灾乐祸的情绪中自娱自乐。这时，《连线》杂志著名的乌鸦嘴再一次应验了：每当某个新鲜事物登上《连线》的封面，它就会在几个月内被证明是愚蠢和错误的。



哦，对不起，你买过《连线》指数吗？

而对于所谓的“新经济”，《连线》杂志的编辑真的有点过分了。他们早该知道，任何技术、公司或流行事物，只要放上这本杂志封面，就像是一记死亡之吻。多年来这类事情时有发生，从smell-o-rama到那些穷途末路的游戏公司，再到PointCast将会取代互联网^①的预言——等一下，PointCast在1997年3月已经取代了互联网。但是《连线》的编辑这次不仅把“新经济”搬上

① 克雷格·比克内尔，“PointCast Coffin About to Shut”，《连线》新闻，2000年3月29日。

参阅<http://archive.wired.com/techbiz/media/news/2000/03/35208>。

了封面，还为它做了一期专刊^①。这让 NASDAQ 股票指数像一只学习飞行的绵羊一样，一落千丈。

建立在别人不幸上的快乐总是那么短暂，如今局势变得更加令人沮丧。虽然我知道经济还没有正式进入萧条期，但还是情绪低落，不是因为那么多愚蠢的初创公司夭折了，而是因为整个时代的面貌变得很低迷。看来我们再也吃不起 Krispy Kreme，要改吃狗粮了。

我们正在把这种心态付诸行动，生活无论如何总得继续。即便每个人走路都垂头丧气，像是下巴被黏在胸口一样，哀叹自己投入了那么多的时间和精力，牺牲了健康和爱情去买 SockPuppet.com 的股票，生活还是得继续。产品的开发周期也必须继续，而对 Fog Creek 软件公司而言，目前产品开发进行到了必须要“吃自己做的狗粮”的阶段。所以这段时间请叫我们 Dog Creek 软件公司。

“吃自己做的狗粮”是我们计算机行业的行话，意思是真正去使用自己开发的产品。一个月前的经历让我再次相信了它的作用。当时我把 CityDesk 的一个构建版本（以为还有 3 周就可以发售了）带回了家，试着用它来搭建一个网站。

结果我发现，因为存在几个 bug，我卡在了某些操作步骤上无法继续，必须先修复这些 bug。即使我们做了那么多测试工作，把所有菜单都逐个点开查看功能是否正常，竟然没有发现对产品基本功能造成影响的拦路虎。通过站在用户的角度去真正使用产品，我立刻就发现了这些 bug。

我还发现了更多的问题。我只是在试图搭建一个简单的网站，甚至没有测试任何特殊的功能点，在短短的一个周日下午就发现了 45 个 bug。我并不勤快，使用软件的时间不超过两个小时。而且就像刚才所说，我尝试的都是产品最基本的功能。

周一早晨我一到公司，就把整个团队召集到厨房里，告诉他们我在周末找到了 45 个 bug（实际上，其中很多所谓的 bug 并不是代码缺陷，而是我在使用过程中觉得不够方便的地方）。然后建议每个人都用 CityDesk 去搭建至少一个真正的网站，以便发现更多 bug。这就是我刚才说的“吃自己做的狗粮”。

接下来举例说明你会有什么收获。

① 彼得·施瓦茨，彼得·莱登，“The Long Boom: A History of the Future, 1980-2020”，《连线》，1997 年 7 月。参阅 <http://www.wired.com/wired/archive/5.07/>。

我猜有很多人会试图通过复制粘贴 HTML 代码，把现有的网页导入 CityDesk。这是可以做到的。但当我试图导入《纽约时报》网站上的一个动态页面时，花了整整一天编辑 HTML 代码：先找到所有 IMG（图片）链接（指向某些外站的图片），然后手动下载这些图片，将其导入 CityDesk，再修改 IMG 链接，指向本站的内部图片。令人难以置信的是，那个网站一篇文章的页面里就包含了 65 个 IMG 链接，指向 35 个不同的图片地址。有些起分隔作用的素材图片只有 1 像素，很难用一般的浏览器下载并保存这样的图片。CityDesk 还会把导入的图片用一个规范化的数字重命名，根本无法判断图片对应哪个数字。长话短说，我花了一整天的时间才把一个页面导入 CityDesk。

这让我感觉很沮丧，于是决定换换心情，去苗圃清除杂草。（真不知道没有这些杂草我该如何减压，谢天谢地我们请不起园艺服务。）接着我突然开窍了。嘿，我可是程序员！导入一个网页并调整所有图片所消耗的时间，足够我写出一个自动导入网页的小程序了！事实上，写这个小程序的时间比手动导入的时间还要短。有了它，导入一个网页的时间从一天减少到了半分钟，而且基本上不会出错。

哇！

这就是吃自己做的狗粮的收获。

当迈克试着导入某些网站时，他发现了 I 无意中写下的 10 个 bug。比如，我们发现某些网站的图片名字非常复杂，不能被转化为文件名，因为其中包含了问号（问号虽然可以在 URL 中出现，但不能出现在文件名中）。

有时你下载了某个软件，发现它非常不好用，即使完成最简单的任务也非常困难。这很可能是因为软件的开发者没有亲自使用过它。

对于不吃自己做的狗粮而导致的失败，我还有一个有趣的例子。你猜朱诺在线服务（Juno Online Services）公司内部使用什么电子邮件产品？（如果没有看过我的文章，你可能不知道我以前在朱诺的客户端团队工作过几年。）

你猜是朱诺，因为毕竟是我们自己的产品，对吗？

事实并非如此。只有几个人，包括总裁在内，在家里使用朱诺邮件客户端。剩下的 175 个人都使用微软 Outlook。

而且理由还很充分！朱诺客户端确实不是很好用，我们在两年里做的唯一工作就是优化广告的呈现方式。团队里有很多人都认为，如果我们使用自己开

发的产品，仅仅为了减轻自己的痛苦，也会让它变得更加好用。公司总裁一度坚持让我们从 6 个位置同时弹出广告，而当他回家打开朱诺，看到弹出 6 个广告时，才改了主意：“还是只留两个广告好了。”

当年美国在线的邮件客户端会员增长势头甚猛，部分原因是它提供了比朱诺更好的用户体验，而我们对此并不了解，因为我们根本不用自己开发的产品。我们之所以不想吃自己做的狗粮，是因为它太难吃了，而且研发管理弊病丛生，根本不允许技术人员去改进产品，让它变得不那么难以下咽。

扯远了。总之，CityDesk 变得好多了。我们修复了之前所有的 bug，又找到了更多 bug，并加以修复。我们正在增加一些以前被忽视但又非常必要的特性。而且，产品离发售更近了！万岁！值得庆幸的是，我们少了 37 个竞争对手，它们每个都拿了 2500 万左右的风险投资，免费提供产品，通过把大幅广告塞到用户眼皮底下来赚钱。在如今这个后新经济时代，所有人都在盘算如何避免付费。如果你够聪明，后新经济时代没什么不好。但是关于“互联网崩盘”的新闻报道没完没了，只能凸显商业媒体编辑们创造力的匮乏。像 fuckedcompany.com 这样的网站，在刚出现的一个多月里可能会让人觉得新鲜，但是现在已经令人生厌。Fog Creek 软件将继续致力于改进产品，努力在这场残酷的行业淘汰战中存活下来。我们不会坐着飞机到处去忽悠更多的风险投资，而是会倾听用户的需求，吃自己做的狗粮，从而实现自身的发展。

31

普通程序员如何改善 团队工作方式

2001年12月25日，星期二

本书主要探讨软件开发管理。但是有时候，你并没有足够的权力去使用行政命令来推动团队做出改变。如果你只是一个最基层的程序员，位于公司的最底层，显然不可能去命令大家开始写项目进度表或是使用 bug 数据库。事实上，即使你当上了开发经理，还是会发现，管理开发者简直就像放牧一大群猫，而区别在于前者一点也不好玩。你说到口干舌燥，他们依然我行我素。

在乔尔测试^①得分很低的地方工作，你会感到非常沮丧。不管你的代码质量有多高，如果同事写的代码质量很差，你还是会羞于承认自己和这个项目有任何关系；或是水平有限的管理者做出非常糟糕的决策，迫使自己浪费青春和才华，比如调试一个在 AS/400 系统上运行、让孩子们做退休规划的游戏。

你当然可以一走了之，但也很有可能因为某种原因无法脱身。比如还没拿到股权，或是在你生活的地方没有更好的公司，或是老板绑架了你最爱的人。无论何种原因，在一个糟糕的团队里艰难度日是非常令人愤懑的。但还是有办法去自下而上地改善你团队的工作方式，接下来我就来和大家分享一下。



办法 1：放手去做

有时候只要有一个人起带头作用，就可以在很大程度上改善项目的开发方

^① 见第 3 章。

式。没有每日构建服务器？自己搭建一个。在你自己的机器上设置一个计划任务，每天晚上进行代码构建，并且用电子邮件把结果发送出去。构建代码需要的步骤过多？自己写一个 `makefile`。没人做可用性测试？自己进行走廊可用性测试，用一张纸或是一个 VB 原型产品，向收发室的人询问改进意见。



办法 2：借助病毒营销的力量

乔尔测试中的许多开发策略，即使没有团队配合，也可以由一个人来实施。如果某些策略在实践中取得了良好的效果，那么其他人自然会纷纷仿效，从而把好的开发习惯慢慢扩散到整个团队。

比如，你们团队的人都不愿意使用 bug 数据库^①，那么别管他们，你自己用就行。记录并管理你在自己的代码中发现的 bug。如果你发现了一个应该由其他人修复的 bug，就在 bug 数据库中把它指派给相关责任人。如果你们使用了一个优秀的 bug 跟踪软件系统，那么指派行为将自动地发送一封电子邮件给相关人员。如果 bug 没有得到修复，你可以一直发送电子邮件。最终他们会认识到 bug 跟踪系统的价值，开始使用这个系统。如果测试部门拒绝向 bug 跟踪系统输入数据，你可以拒绝通过其他渠道接收 bug 报告。如果他们抗议，你就说：“听着，我很乐意修复这个 bug，但是我怕会忘掉。你可以向 bug 跟踪系统录入一条数据吗？”直到他们不胜其烦，就会开始用 bug 数据库了。

你们团队都不想用源代码管理系统？那就自己建一个 CVS 代码库，如果必要的话，建在自己的硬盘上也行。即使没人合作，你还是可以签入自己写的、独立于其他人的代码。然后，当团队遇到一个源代码管理系统能解决的问题（比如某人误把 `rm *` 写成了 `rm * ~`），他们就会向你求助。最终大家会意识到，他们也可以签出自己的代码，避免代码的丢失。



方法 3：营造一方优秀的小天地

团队不写开发进度表^②？不写功能规格书^③？你可以自己写。如果你花一两

① 参阅 <http://www.joelonsoftware.com/articles/fog0000000029.html>。

② 见第 9 章。

③ 见第 5 章。

时间，为将要做的工作制定最简单的功能规格书和开发进度表，没人会抱怨你偷懒。

让更优秀的人进入团队。积极参与招聘和面试^①活动，选择优秀的人才，为团队增添新鲜血液。

找到既愿意也有能力去改变现状的人，想办法让他们站在你这边。即使在水平不高的团队，还是能发现某些聪明的家伙，他们往往因为缺乏经验而写不出优秀的代码。帮他们一把，引导他们自主学习。阅读他们签入的代码，如果这些代码犯了某些愚蠢的错误，不要直接给他们发邮件，用盛气凌人地口吻指出代码的谬误之处。这会激怒他们，让他们产生抵触情绪。你需要做的是报告这次代码签入所引起的 bug，让他们自己追查出错的原因。这样在问题得到解决的时候，他们会有更深刻的体会。

方法 4：干掉害群之马

即使最棒的团队也可能有一两个害群之马。团队中有差劲的程序员是令人苦恼的一件事，因为他们写的拙劣代码会破坏别人的好代码，优秀的程序员不得不花时间去为他们收拾残局。

作为普通程序员，你的目标是尽可能减少差劲程序员对团队造成的伤害，即采取所谓的牵制战术。比如，你发现某个“天才”花了两周才写出的代码拙劣至极，永远都运行不了。你很想 15 分钟内把代码重写一遍，让它正常工作，但是要拒绝这种诱惑。摆在你面前的是一个绝佳的机会，可能让这个白痴在几个月里都无法造成更大的伤害。你需要做的只是不断地报告这段代码的 bug。他们别无选择，可能在长达几个月的时间里都在和这段代码较劲，直到你再也找不出更多的 bug 为止。至少在这几个月里，这些害群之马无法在别的地方造成破坏。

方法 5：屏蔽干扰

快乐的工作环境总是相似的（私人办公室、安静的工作氛围、最棒的工具、

^① 见第 20 章。

尽可能少的打扰，以及大型会议），而不快乐的工作环境各有各的烦恼。

坏消息是，在所有公司里，改变工作环境几乎都是不可能的。有时候积重难返，连 CEO 也无法改变工作环境。所以，软件开发者很少拥有自己的私人办公室。这至少在两方面损害了公司的利益。第一，招募顶尖开发者变得更加困难，这些开发者更青睐提供舒适工作环境的公司（在其他条件相当的情况下）。第二，干扰次数的增多会让开发者的工作效率急剧降低，因为他们不能进入工作状态并长时间保持专注。

你可以寻找逃离这种环境的方法。带着笔记本电脑去公司的餐厅，那里桌子很多，而且一天中大部分时间都空空如也（没人能找到你）。或者预定某个会议室，在那里写一整天代码，要产出比平常更多的代码签入量，以凸显拥有个人空间对于产能的巨大提升作用。下次如果出了紧急状况，经理问你“需要提供什么条件才能在明天把工作做完”的时候，你知道该说什么了吧。他们会给你暂时分配一间办公室。很快，他们就会开始琢磨怎么才能让你一直保持这种开发效率。

工作时间内可以采取早点来、晚点走的策略。其他人都下班回家的那段时间可能是最有效率的时候。如果你们公司的开发者上班都很晚的话，那就早点到，比如 9 点。在其他人员陆续到来并开始打扰你之前，有两个小时的清净时光，这段时间内做的工作可能比你一天剩下的时间里做的还要多。

不要让电子邮箱或即时通讯工具客户端持续运转。如果必要的话，每隔一个小时检查一次电子邮件，但不要随时检查邮件。



方法 6：提升自身对团队的价值

如果你对团队的贡献并不显著，以上方法都不会起作用。如果你写的代码不够好、不够多，还在那里鼓捣 bug 数据库，别人就会认为你不务正业，对你有意见。如果给别人留下了“这个人只关注过程，什么都干不好”的印象，那你的职业生涯就危险了。

有一次，我刚到某家公司做普通程序员，发现那里的开发环境在乔尔测试中大概只能得到 2 分，于是决定改变这种局面。但我也知道，留下良好的第一印象是非常重要的。所以，我每天工作时间的 7 个小时都做好本职工作，

也就是写代码。大量的代码签入是在团队中树立威信的最好方式。与此同时，我在每天下班前的一小时，着重改进公司的开发过程。我用这段时间来修正那些让产品调试变得困难的事物：搭建了一个每日构建服务器和一个 bug 数据库，修复了所有常年遗留下来的、阻碍开发顺利进行的小 bug，为一天中涉及的工作内容撰写功能规格书；我写了一篇文档，详细介绍了如何一步步地从头搭建开发机器环境；为公司内部使用的一种语言补充了翔实的说明文档。开发过程慢慢变得越来越好。只有我（还有我领导的团队）撰写开发进度表和功能规格书，我的开发团队还在乔尔测试中达到了 10 分。

即使你没有行政权力，还是可以推动一些事情向着更好的方向发展。但必须保持低调，把自己看作“凯撒的妻子”，不要引起怀疑，否则在此过程中会树立很多敌人。

32

两个故事

1999 年末，戴夫·温纳开设了一个名为 EditThisPage.com 的在线服务，邀请广大网民创立自己的个人站点，形式为与他的个人网站 Scripting News 相同的博客。那时我恰好在某家公司受了很大的委屈，被那里“打了就跑”的管理方式（经理偶尔会越俎代庖，盯着你做每一件事情，但在剩下的时间里，你完全感觉不到他们的存在）折腾得空有一身才华无法施展。于是我就写下了这篇典型的乔尔式文章，它符合博客文章一贯的伟大传统：伪装成商业管理建议，宣泄个人的苦闷情绪。

谢天谢地，没有几个人看我的网站，所以我可以在博客中随意指名道姓，甚至敢于冒险去得罪西雅图地区的亿万富翁。人家名下的豪车的价值，就比我所有银行户头上的钱外加退休金还要多。啊，多么令人怀念的一段岁月！

2000 年 3 月 19 日，星期日

我想和大家分享职业生涯中的两个故事，它们很好地诠释了不同管理水平的科技公司之间的差别。这个差别最终归结为：是信任员工、给他们把事做好的自由，还是把他们看作快餐店的临时工，需要时刻严密监控，以防其开小差或是搞砸一切。

我的第一份工作是在微软。交给我做的第一个项目，是为 Excel 设计某种宏语言的方案。我很快完成了 Excel Basic 功能规格说明书的第一版草案（后来演变成了 Visual Basic for Applications，暂且不提）。不知怎么的，微软传说中的“应用架构组”听说我在规划一个宏语言方案的消息，因为他们认为宏语言方案设计是自己负责的内容，所以表示关注，想看看我写的规格书。

我到处问同事，应用架构组是何方神圣？我发现没人把他们太当回事。实际上这个组只有 4 个人，都是刚招进来的博士（当年这在微软并不多见）。我把规格书抄送给他们一份，并前去拜访，想听听博士们有何高见。

他们七嘴八舌地神侃一通，我没有听到什么值得采纳的意见。他们非常热衷于把事物细分成很多子类别，似乎认为在 Excel 中使用宏的用户也想定义很多子类别。最后，其中一个博士说：“你写的很有意思。下一步是什么？该由谁来批准你的规格书？”

我笑了。即使刚来微软几个月，我也知道这里从来没有批准规格书这回事。唉，根本没人有时间读我的规格书，更别说批准了。程序员整天缠着我，让我多写几页，好让他们有更多的代码可写。我的上司（以及他的上司）早就向我坦承，其他人都不懂宏，也没有时间去弄懂，所以不管我做什么，最好自己把事情做对。而这个供职于微软奇怪研究部门的博士，误以为这里的开发工作流程会更加正式。

我很快就意识到，应用架构组对宏的了解比我更少。至少我与四五个宏脚本开发者和一些 Excel 前辈交谈过，大概了解人们用 Excel 的宏来做什么：比如每天重新计算一遍电子表格的数据，或根据某种模式来重新排列一些数据。但应用架构组只把宏看作是一个学术上的议题，举不出一个人们编写宏的实际例子。迫于我的压力，其中一个人举例说，目前 Excel 有下划线和双下划线，也许有人想写一个宏来实现三重下划线。好吧，这功能真的有必要吗？从此以后，我按照自己的想法来做，并且尽量礼貌地忽略他们的意见。

但我还是激怒了一个叫格雷格·威登的家伙，他是应用架构组的老大。要知道，格雷格在微软的员工号是 6。他一直都在公司，尽管没人能指出他做过什么事情，但很显然他总是和比尔·盖茨一起吃午饭，而且 GW-BASIC 是以他命名的。格雷格召集了一个声势浩大的会议来抱怨 Excel 团队（也就是我）把宏策略搞砸了。我们追问他具体的反对理由，但是他的答复难以令人信服。请想象一下，我作为一个刚出校门、进入公司的小人物，竟然敢和员工编号为 6 的公司元老叫板，而且竟然占了上风！（你能想象这种情形发生在一个传统公司里吗？）以本·瓦德曼（现在他是微软的副总裁）为首的 Excel 编程团队完全站在我这边，这其实才是最重要的，因为代码是由他们编写的，他们对于功能如何实现有着最终决定权。

如果这件事到此为止的话，我会十分满意。如果应用架构组想找事情做，和我坐而论道，我一定随时奉陪，只要他们放过程序员，让程序员做好自己的

工作。但是接下来发生的事情更加离奇，让我惊呆了。有一天我正在和同事一起吃午饭，在雷德蒙德灿烂的眼光下，我看见皮特·希金斯向我走来。那时皮特是 Office 部门的总经理——我当然认识他，但他不一定认识我。

“乔尔，最近怎么样？”他问，“我听说你和应用架构组之间有点小分歧。”

我说：“没什么，我扛得住。”

他说：“不用再说了，我明白了。”然后就离开了。第二天我听说，应用架构组被解散了。不仅如此，其成员被拆散到微软的各个部门，彼此之间离得不能再远了。之后他们再也没有和我联络过。

我当然大吃一惊。在微软，作为一名负责 Excel 宏策略的开发经理（即使加入公司还不满 6 个月也没关系），你在 Excel 宏策略方面就是神，没有人可以挡住你的路，即使是 6 号员工也不行。就是这种感觉。

这件事给我们造成了很深的影响。首先，每个人都对自己的工作更加负责。你不能“规格书经过管理层批准”来作为遁词，因为管理层并不会仔细阅读规格书。管理层所做的就是把聪明人招公司，然后给他们事做。其次，这使得工作环境变得非常优越。谁不想在自己的领域称王称霸呢？软件生来具有易于分解为更小组件的特性，所以总能够把开发职责进行细分，然后让每个人负责一个领域。这也许是软件工程师喜欢在微软工作的最主要原因。



很多年过去，我在在线服务和免费电子邮件提供商朱诺公司工作。这一次的工作经历几乎和微软的情况截然相反。我手下有两名程序员，但是我的上司经常削弱我（本来就有限）的行政权力，在我毫不知情的情况下，直接给我的下属分派工作。即使像批准或拒绝员工请假这种小事，我的上司也不肯放权。

在朱诺工作几年后，我负责开发一个新的用户注册功能。在 3.x 重大发布版本中，我负责重写整个注册流程。我那时候在技术团队的职位已经很高了，绩效也一直很好，而且高管们似乎都很赏识我。但是他们还是不能信任我，仍然以命令和控制的方式安排我工作。

注册流程中有一个环节，需要用户输入生日。这只是漫长注册流程中的一个

小步骤，整个流程大概有 30 个页面那么长。朱诺会像审讯一样询问你的收入、喜欢的运动、有几个孩子、孩子多大了……有一百多项。生日这一项的格式比较自由，你可以输入“8/12/74”或“August 12, 1974”或“12 Aug 74”等。（你可能用过 Outlook。这项功能和 Outlook 中是一样的，以任意一种格式输入日期都能被识别。）

还没有了解清楚详情，我的经理就决定不喜欢这种做法。这成了关乎他自尊的大事。他先是对正在设计页面的设计师大喊大叫（我毫不知情），然后又对我大喊大叫。接着他每天都提醒我，必须按他的意愿修改设计。之后他找来了公司的 CEO，让他当众批评我的新设计。在朱诺，即使是 CEO 也喜欢经常干预底层的工作。事实上，这已经成了行政惯例。

我当然非常愤怒。这只是一件小事，完全是个人喜好问题。有些人倾向于我的处理方式，有些人则喜欢他的处理方式。无论如何，这件事带给大家的讯息非常清楚：上头怎么说，你就怎么做！这种命令与征服式的心态，让一次普通的用户界面设计分歧，演变成了一决雌雄的斗争。

这件事并不是导致我离开朱诺的直接原因，但反映了我离开朱诺的原因：这家公司让我觉得，无论你多么努力地工作，多么聪明，是否在担任某些“管理”工作，都在最小的方面没有一丁点的自主权。必须把一切让你在职场上有价值的点子、经验、小聪明、大智慧都收起来。而且，朱诺有非常多的经理，大概占有所有员工的四分之一。因此，这帮经理有大把的时间来插手每一个细小的决策，以确保一切都处于掌控之中。与此形成强烈反差的是，在微软，副总裁会专门从 9 层下来，明确告诉员工他们享有职责范围内的充分自主权。

朱诺无可救药的管理模式，在某种程度上是由于它地处美国东海岸的纽约，而不是西海岸，所以现代化的管理还没有渗透进去。导致管理问题的另一个诱因是朱诺的管理层缺乏经验，而且源于高层——CEO 当年只有 29 岁，只在 D.E.Shaw 公司工作过，无论大事小情全部都要过问，包括亲自修改软件错误信息的措辞。CTO 经常在听中层经理报告的时候大发雷霆，只是因为经理胆敢质疑他的智慧；而下属转而拿程序员出气，程序员则只能回家去踢自己的狗了。微软与此完全不同，所有的工作都是在最底层完成的，而大部分经理平常最重要的工作，就是满屋乱跑，把挡道的家具挪开，以便大家能全神贯注地工作。



巨无霸汉堡vs.“原味主厨”

2001年1月18日，星期四

为什么某些最大的 IT 咨询公司提供的服务水平却最差呢？

为什么出色的咨询类初创公司在取得最初一系列非凡的成就，迅速发展壮大之后，会突然退化，最终“泯然众人矣”？

我经常思考这些问题，思考自己的公司 Fog Creek 该如何成长。能找到的最好范例来自麦当劳。没错，就是那家令人肃然起敬的汉堡连锁店。

巨无霸汉堡的成功秘诀在于它并不是很好吃，但这就像每个人并不完美一样。能够接纳生活中不完美事物的人，在吃巨无霸汉堡的时候，绝不会感到丝毫的惊奇。

巨无霸汉堡的另一个成功秘诀是，即使是智商介于“白痴”和“愚钝”之间的人，也能做出合格的巨无霸，它将与世界上所有的其他巨无霸汉堡一样平淡无奇。这是因为麦当劳真正的秘密配方，是它厚厚的操作手册，里面巨细靡遗地描述了制作巨无霸汉堡的每一个步骤，每个分店都必须严格遵守这套操作流程。如果一个巨无霸汉堡在阿拉斯加州的安克雷奇需要油炸 37 秒，那么在新加坡，也必须油炸 37 秒。多一秒少一秒都不行。必须严格按照规定制作每个巨无霸汉堡。

这些规定是由一群相当睿智的人研究制定出来的（在麦当劳的汉堡包大学），蠢蛋可以和聪明人一样，很好地遵守这些规定。事实上，这些规则中包含了针对所有情况的预防措施，例如，如果一批薯条在油中炸得过久，就会响起铃声，从而避免很多的人为疏忽。到处都有秒表和计时系统。还有一个系统专门用来确保清洁工每半小时检查一下卫生间是否干净（每次的检查结果都是不干净）。

在设计上，整套系统考虑到了每个人都会犯各种错误，但其目的是想确保产出的汉堡质量是，怎么说呢，一致的，而且保证点餐员每次都会询问：“您是否要搭配薯条一起享用？”

出于娱乐的目的，让我们来比较一下麦当劳的“厨师”（完全不懂烹饪，只不过是按照一堆操作规程加工食物）和杰米·奥利弗，那位来自英国的可爱“原味主厨”。（如果你选择现在放下书，去他的网站^①观赏拍得如同音乐录影带的《“原味主厨”教你做罗勒蒜香蛋黄汁》，那就祝你好运，我不会怨恨你的。）虽然把麦当劳小时工和美食大厨相提并论实在不成体统，但请暂时忍耐一下，因为这里面有些值得研究和学习的東西。

毫无疑问，“原味主厨”绝不可能照着一本操作手册来做菜，也不会对任何食材或烹饪步骤进行精确计量。他烹饪的时候，你会看到到处都是各种食材，而他在其中游刃有余。“我们还要再加一点点迷迭香，这无伤大雅，反而为这道菜增添了几分复古的韵味。”大厨说，“把它捣碎。太完美了！然后到处都撒上一点。”（看上去他似乎确实是在到处乱撒食材，但如果我照猫画虎也那样做的话，出来的味道就完全不行。）在14秒内，他基本上是在即兴发挥，将海鲑鱼剖开切片，加上草本香料，放在刷有欧芹酱的蘑菇与土豆上烘烤，自创了一道全新的美味。

很显然，“原味主厨”创作的美食要比麦当劳快餐好吃多了。即使这个问题看似很蠢，我还是想问，为什么？这个问题其实并不简单。大型公司拥有难以计数的资源，难以想象的庞大规模，能够聘请到最好的美食设计师。既然从来不缺钱，为什么不能为顾客提供可口的食物呢？

假设“原味主厨”厌烦了“上电视”，开了一家餐馆。因为他是烹饪天才，那家餐馆的食物肯定非常好吃，很快变得门庭若市、财源滚滚。

而当拥有一家利润可观的餐馆之后，老板很快发现，即使每个晚上都顾客盈门，即使一道开胃菜标价19美元，一罐可乐卖3.95美元，利润的增长空间还是有限。因为一个厨师就算有三头六臂，也只能做那么多的食物。必须雇佣更多的厨师，或者开一些分店，进而把生意拓展到其他的城市去。

这时一个问题会日渐明显，我们科技圈的人称之为可扩展性问题。试图复制在一家餐馆取得的成功时，你面临两个选择：要么雇佣一个和你水平相当的大厨（而这些人很可能想获得自己赚取的大部分利润，不划算）；要么雇佣

① 参阅<http://www.jamieoliver.net>。

一个要价较低、没你这么优秀的年轻厨师，但很快顾客们就会发现分店的食物不那么正宗，因此不再光顾。

解决可扩展性问题的通常做法是，雇佣便宜的新手厨师，让他们严格遵守操作规程去做每一道菜，这样就不会搞砸。只要遵守这些操作规程，人人都可以做出美味的食物！

但问题是，烹饪不是一门可以严格规范化的手艺。优秀的厨师在做菜的时候总是有很多即兴发挥、随性而至的地方。他们采购食材的时候看到芒果很新鲜，就会灵机一动，自制一种以芒果、香菜和洋葱为原料的萨尔萨辣酱汁，来搭配那天的鱼类主菜。优秀的厨师如果发现土豆不够了，会用芋头片来代替。在流水线上按照指令进行食物加工的厨师，只能在一切不出差错的情况下做出某道菜，但由于缺乏真正的才华和经验，无法进行即兴发挥。因此你不可能在麦当劳看到凉薯。

麦当劳使用一种特殊的土豆，并在全球广泛种植这种土豆，然后大规模地进行预切削、冷冻等处理，以预防食材短缺的情况出现。虽然经过预切削和冷冻的土豆不如新鲜的好吃，但是能长久保存，并且降低了对操作者厨艺的要求。实际上，麦当劳做了数百件事情，都是为保证产品品质的稳定性。各个分店厨房工作人员的厨艺水平，对于这种稳定性的影响非常小。

简单小结一下：

- (1) 做好某些事情是需要天分的；
- (2) 天分无法复制，具有不可扩展性；
- (3) 人们想出一种复制方式，就是让有天分的人制定规则，让缺乏天分的人去遵守规则；
- (4) 按照规则做出的产品，其品质是很低的。

你会发现，同样的道理也适用于 IT 咨询行业。下面这种故事你听过多少次了？

迈克闷闷不乐。他从大公司聘请了一个大型 IT 咨询团队来构建一个系统。这些咨询专家谈论起“方法论”来头头是道，但是动手能力极差。结果迈克花了上千万美元，到头来却什么都没得到。

幸好迈克碰到了一位天资过人、才华横溢的年轻程序员。这个程序员在一天之内就建好了整个系统，只要了 20 美元和一张比萨饼。迈克高兴坏了。他向所有朋友推荐这位年轻的程序员。

这位程序员赚得盆满钵满。很快工作多到一个人干不过来，于是他雇了一群人帮忙。请不动优秀的人才，他只能雇佣那些更加年轻、刚出校门的程序员，然后经过为期6周的课程“培训”，就正式上岗。

问题是“培训”并不能使员工产出稳定的工作成果，为此，这位年轻的程序员开始制定各种规则和操作规程。几年以后，这些规则越来越多，很快就集结成册，变成了一本6卷的巨著，被冠以“方法论”的名字。

几十年过去，当年那个年轻的程序员已经变成了另一个只会纸上谈兵的IT咨询大师，有一套自己的方法论，还有大批盲目而虔诚的信徒。虽然大师的方法论并不总能奏效，但这些信徒不知道除了相信方法论还能怎么办，因为他们自己并不是真正有才华的程序员，可能只是主修政治学专业的学生，只参加过为期6周的编程速成班。

到这个时候，又一个规模庞大但水平低下的IT咨询集团开始走下坡路。客户的满意度逐渐降低。然后另一位初出茅庐却才华横溢的程序员闪亮登场，抢走所有的生意，如此循环往复。

我不需要指出具体的名字，这种循环已经发生了十多次。所有的IT服务公司都被贪婪蒙蔽了理智，补充人才的速度赶不上公司规模增长的速度。虽然构建一层又一层的规则和流程有助于产出一致性的IT服务，但这种服务的水平实在让人不敢恭维。

况且规则和流程只在一切不出差错的情况下有效。最近几年出现了数家咨询公司，提供的服务针对“数据支撑的网站”，他们向编程菜鸟兜售“构建一个数据支撑的网站需要掌握的14个知识点”（“孩子，用心领会这个select语句，你就能自己建一个网站”）。然而，在互联网泡沫破灭的今天，业界突然出现了对于掌握高端图形用户界面编程、C++编程和传统计算机科学知识的的人才的需求缺口，而这些只领会了select语句的孩子们面临的是一条异常陡峭的学习曲线，很多人都迈不过这道坎。他们不肯轻易放弃，还在试图按照第17章里的规范化数据库来做，但不知为何，这些知识在新世界里不适用了。咨询公司的创始人能够很快适应新的世界，因为他们是天资卓越的计算机科学家，能快速掌握任何新知识。但是他们创立的公司却无法适应这种变化，公司之前赖以生存发展的是各种规则手册，而要想适应新的时代，不能靠规则手册，而要靠个人的才能与努力。

这个故事告诉我们什么呢？不要过分相信方法论。它虽然能让任何人都产出可怜但勉强过得去的业绩水平，但在同时，对于那些真正有才华的人们来说，

它是一种沉重的负担，限制了他们的发展。如果让一个有才华的厨师在麦当劳做汉堡，他肯定不会高兴，这恰恰是因为麦当劳各种各样的操作规定。IT咨询业也是同样的道理，为什么 IT 咨询师总是在吹嘘自己那套所谓的方法论呢？（反正我不知道。）

这对于 Fog Creek 意味着什么？首先，我们从来不想成为一家大规模的咨询公司。我们当初做咨询业务，是为了想成为一个能够持续盈利的软件公司。我们想通过咨询业务贴补一下软件的收入，从而更加接近这个目标。几年以后，当我们的软件收入增长到了一定程度，相比之下，咨询就成为了利润率很低的边缘业务，所以我们现在只做能直接支持我们软件的咨询。众所周知，软件具有很高的可扩展性。每卖出一份 FogBUGZ，我们都能赚到更多的钱，而成本却没有增加。

更重要的是，我们坚持只招最好的人。如果找不到足够的优秀人才，我们完全乐于保持公司目前的较小规模（话虽如此，由于我们每年有 6 周的带薪休假，招聘似乎不是什么大问题）。等到聘用的人才成长到能够成为后来者的导师时，我们才会考虑扩大公司的规模。

凡事没有看上去的那么简单

“极限编程”是一系列理念的统称，其中的很多理念非常好，有几个不是很好，还有一个错误而危险：在软件开发过程中，做计划和设计是在浪费时间。极限编程哲学提供了一套完整的开发方法论，虽然总的来讲是很有道理的，但是实践中，程序员经常把极限编程当作借口，冠冕堂皇地不先做好功能设计再开发代码。他们辩解道：“写代码就是在做设计！”但这并不正确，如果这样开发软件的话，你会发现自己像是圆形转轮里的仓鼠，通过重构来改善软件设计“坏味道”的尝试总是原地打转，完全无法取得任何有意义的进展。

2002年3月4日，星期一

我们公司的软件 CityDesk 有一个可用性问题。

问题是这样的：可以用一个菜单命令（“导入网页”）从网页上导入文件，也可以通过鼠标拖拽将硬盘上的文件导入 CityDesk，但是没有导入硬盘文件的菜单命令。这让用户要么认为软件不支持此项功能，要么尝试用“导入网页”功能来导入硬盘上的文件，而这是行不通的。

一种简单的解决办法是使用分为两页的引导式对话框。第一页询问用户：“你想从哪里导入文件？”如果用户选择“硬盘”，第二页就会弹出打开文件对话框。如果用户选择了“网页”，那么第二页就会提示输入一个 URL。

我差点就开始写代码实现这个解决方法了，但不禁停了下来，转而先写一个最精简的功能规格书。以下是规格书的全部内容：

第一页：

你想从哪里导入文件？（硬盘/网页）

第二页（硬盘）

标准打开文件对话框

第二页（网页）

带有迷你网页浏览器的URL输入框

我突然想到一件事情。能够把 Windows 提供的标准打开文件对话框，嵌入到我们自己写的引导界面里吗？

这是个问题。

我调研了一下。虽然可以这样做，但是一点也不方便，需要几个小时才能完成。可以不使用引导界面来解决这个问题吗？我重写了规格书：

两个菜单项：

1. 从互联网导入网页→弹出URL对话框
2. 从硬盘导入网页→弹出打开文件对话框

这样好多了。通过多花三分钟进行设计工作，我节省了几个小时的编程时间。

只要是做过一段时间编程工作的人，就会同意这条经验之谈：凡事没有看上去的那么简单。

像是复制文件这样简单的事情，也可能会出现各种异常状况。如果第一个参数是目录怎么办？如果第二个参数是文件怎么办？如果目标文件夹已经存在相同名称的文件怎么办？如果没有写权限怎么办？

如果复制中途出错怎么办？如果目标文件夹在可用的远程机器上，但是需要身份验证怎么办？如果文件很大，网络连接很慢，需要显示一个进度条，该如何实现？如果传输速度非常慢，接近于 0，该在什么时候放弃并返回错误消息？

作为面试官，招聘测试岗位人才的时候有一个好方法，就是给定一个简单的操作，请面试者列举所有可能出现的异常情况。一道经典的微软测试类面试题是：如何测试打开文件对话框的功能？优秀的测试者能滔滔不绝地列举几十种奇奇怪怪的测试用例（“在某个文件被选中到点击打开的这段时间里，

该文件被其他用户删除了”)。

很好，我们有了一条公理：凡事没有看上去的那么简单。

软件工程领域有另一条公理：尽量降低风险。特别要注意规避开发进度风险，也就是完成某项工作需要的时间超过预期。开发进度风险危害很大，老板会因为它朝你大吼大叫，让你不开心。如果这还不足以让你重视它，还有另一个经济方面的原因。假如你在决定开发某项功能时，根据前期收集的信息判断需要 1 周开发时间。后来发现已经过了 20 周还是没有完成，那么开发这个功能的决定就可能是错误的。如果当初你知道需要开发 20 周，可能会作出截然不同的决定。你作出的错误决定越多，印有你们公司品牌标志的大袋子越有可能出现在破产清算人的仓库里。那时公司前任 CEO 会幽怨地说：“我们太失败了，甚至没能在网站 fuckedcompany.com 上留下名字。”

将上面提到的两条公理，即“凡事没有看上去的那么简单”和“尽量降低风险”合起来，就得出了一个结论：

先做好设计，再去实现。

如果你感到失望的话，抱歉。没错，我知道你读过肯特·贝克的文章，觉得不经过设计就动手实现功能也没什么问题。遗憾的是，这样做确实有问题。虽然人们总说“修改代码就像修改设计文档一样简单”，但这是绝对不可能的。有人会说：“如今我们都在使用各种高级的工具，比如 Java 和 XML。修改代码逻辑只需要几分钟。为什么不在代码中进行设计呢？”我的朋友，你可以把轮子安在人身上，但不能说他就是汽车。如果你认为重构一个实现有问题的文件复制函数，把它从多线程式改成抢占式多任务处理，能和我写出这个句子的时间一样快，那么你肯定是在自欺欺人。

我并不认为极限编程理念倡导完全不做设计的开发行为。这种理念只是说“不要过度设计，够用即可”，这是无可非议的。但说者无心，听者有意。大部分程序员都在寻找各种理由，来支持他们不做基础设计就着手开发实现功能的行为。于是，他们像飞蛾扑火一样拥抱这个“不做设计”的理念，结果只能是引火烧身。这是一种特殊的懒惰行为，导致最后要做的工作比不偷懒还要多。我太懒了，不想先在纸上把功能点设计好，所以就先试着写了一些代码，然后发觉不对，需要再修改，最后发现比先设计后开发花费了更多的时间。而通常的情况是，我写了一些代码，然后发觉不对，但是太晚了，产品只能以这种低品质的状态发布，我不得不把剩下的时间用来制造各种借口，解释“为什么只能做成这样”。这实在是太懒散、太不专业了。

林纳斯·托瓦兹^①曾经抨击过设计^②，但针对的是大型系统，这样的系统必须逐步演化，否则就会变成另一个 Multics^③。他的话并不针对于文件复制函数这种功能开发。因为对 Linux 有一个明确的发展路线图，所以林纳斯轻视设计是情有可原的。无论如何，林纳斯都比一般人聪明得多，所以对他适用的事情并不一定适用于一般人。

增量式设计和实现很好，频繁发布新版本的产品也不错。（但是对于零售软件或是面向大众市场的软件，产品版本发布太频繁会让客户抓狂，显然不是什么好主意，可以选择只频繁迭代内部开发的里程碑版本。）在设计阶段过分注重形式和规范是在浪费时间，我从没见过一个开发项目由于遵循了某种繁复的设计规范或工具而受益，无论是流程图、UML 图，还是 CRC 图，无一例外。此外，林纳斯讨论的代码量在千万级别的系统，确实只能逐渐演化，因为以人类的心智，设计那么大规模的软件很难一次考虑周全。

但是，当你着手实现文件复制函数，或是构思软件下一个发行版本的功能特性时，一定不要受到塞壬的蛊惑，踏踏实实先做好设计吧。

① 林纳斯自传《只是为了好玩》已由人民邮电出版社出版。这是 Linux 之父唯一亲笔自传，充满各种笑料以及对技术和软件的严肃思考。——编者注

② 参阅<http://www.uwsg.iu.edu/hypermail/linux/kernel/0112.0/0004.html>。

③ Multics 是一套安装在大型主机上的多人多任务操作系统，由贝尔实验室、麻省理工学院和美国通用电气公司于 1964 年共同研发。Multics 旨在让大型主机提供超过 300 个终端机的连线使用，后来因计划进度落后、资金短缺而宣告失败。更多信息请参阅<http://baike.baidu.com/view/1351079.htm>。——译者注

35

为NIH综合症说两句

2001年10月14日，星期日

先做个小测验吧。

1. 代码重用：

- (a) 好
- (b) 不好

2. 重新发明轮子：

- (a) 好
- (b) 不好

3. NIH 综合症^①：

- (a) 好
- (b) 不好

每个人都知道应该尽量利用别人的工作成果。那么，正确答案是 1(a)、2(b)、3(b)，对吗？

先别那么快下结论！

NIH 综合症是一种管理学上的通病，意思是一个团队拒绝使用不是自己创造的技术。患有这种病症的人们明显小肚鸡肠，仅仅因为使用别人开发的工具不能凸显个人工作的价值，就拒绝这样做，从而导致整个组织的利益受到损害（你也是这样想过吗？）。随便走进一家书店，就能在商业史类书架上看

① NIH 综合症即 Not-Invented-Here Syndrome，直译为“不是这里创造的东西（就不用）综合症”。为了行文简洁，下面统一称为“NIH 综合症”。——译者注

到 NIH 综合症的各种实例，例如某个愚蠢的团队花了上千万美元和 12 年时间构建了某个东西，而这个东西在 CompUSA 店里只卖 9.9 美元。只要对这 30 年来计算机编程取得的进展稍有了解，都知道代码重用是所有现代编程系统所追求的终极目标。

我当初也这么想。所以，担任第一版 VBA 的项目经理时，我设计由微软的四个团队（注意，是四个团队）一起合作来开发 Excel VBA 中的定制对话框功能。这个设计十分复杂，代码有各种互相依赖的关系。首先，我们使用 AFX 团队正在开发的一种对话框编辑器。然后使用 OLE 产品组开发的新技术，把一个应用嵌入另一个应用。接着由 VB 团队来提供背后的编程语言机制。经过一周的磋商，我终于让 AFX、OLE 和 VB 团队原则上同意了这个方案。

我去拜访安德鲁·瓦提内兹。他当时是我的上司，我的本事都是蒙他所赐。他听了我的方案，说：“你知道 Excel 团队的座右铭吗？‘找出依赖，然后消灭它们。’所以，那些人绝不会同意一个有这么多依赖关系的开发方案的。”

真有趣，当时我是第一次听到这种说法。我猜这就是为什么 Excel 团队使用自己开发的 C 编译器。

我相信有些读者已经乐不可支了：“微软太傻了，宁可为了一个产品专门开发一个编译器，也不愿意用别人的代码！”

别那么早下定论！正因为 Excel 团队对于自立更生的坚持，他们才能总是按时交付产品，并且代码质量很高。他们早在 20 世纪 80 年代就有了自己的编译器，独立生成 pcode，同时兼容采用 68000 芯片的苹果电脑以及英特尔个人电脑。pcode 将英特尔个人电脑上的二进制可执行文件缩小了一半，提升了从软盘加载的速度，运行所需的随机访问内存也更少。

“找出依赖，然后消灭它们。”当你身处一个非常优秀的团队，身边都是编程大牛，那么坦白地说，其他人写的代码都是滋生 bug 的温床，只有你自己才知道该怎么做才能按时交付产品。一名蓝带级厨师如果需要新鲜的薰衣草，那么他（她）不会去农贸市场采购，而是会自己种，因为农贸市场要么没有新鲜薰衣草，要么会用不新鲜的薰衣草冒充新鲜品出售。

在最近的互联网狂热中，一帮不学无术的商业评论家认为，未来的公司将不存在实体——可能只是一对衣着新潮的夫妇坐在客厅里，一边品着莎当妮酒，一边远程指挥各种外包团队干活。这些兴奋过头的“梦想家们”忽视了一个事实：市场只承认附加的价值。两个雅皮士坐在客厅里从 A 公司购买电

子商务引擎，销售 B 公司的产品，由 C 公司提供仓储和物流，D 公司提供客户支持服务，这种商业行为本身并没有真正创造多少价值。事实上，任何外包过一项关键商业职能的人，都会发现外包糟透了。不能直接控制客户支持服务的公司，其客户服务水平一定差到天怒人怨。经常可以看到人们在博客中抱怨自己的遭遇：拨打客户服务热线，结果连最简单的要求都无法得到满足。如果你选择外包物流服务，经常会遇到对于“按时送到”与一般人有不同理解的物流公司，这样一来，客户一定不会满意，而你对此无能为力。首先是因为换一家物流伙伴可能需要几个月的时间；况且，你其实连客户是否满意都不清楚，因为他们无法直接向你投诉。话说回来，你外包客户服务的目的，不正是为了无视客户的投诉吗？再来说说那个买来的电子商务引擎。它绝对不可能像亚马逊公司自己精心研发的 obidos 一样灵活。（如果随便买来的电子商务引擎都能和 obidos 一样好，那亚马逊就没有竞争优势了。）同理，刚买回来、未经调校的网络服务器不可能像谷歌公司的服务器一样快，因为那可是谷歌精心编码和优化过的。

可惜的是，这个原则看上去和“代码重用好，重新发明轮子不好”的理念相左。

对此，我能提供的最好建议是：

对于公司核心的商业职能，无论多么困难，都要自己做。

选定某些方面作为公司的核心竞争力和成长目标，然后在内部不断地完善和提高。如果你开的是软件公司，代码质量是公司成败的关键，员工餐厅或是光盘刻录之类的事情就可以外包出去。如果你开的是制药公司，一定要亲自写药品研究方面的软件，但是不要自己写财务会计工具包。如果你的公司提供在线财会服务，一定要自己写专业的财会工具包，但是别试着设计投放在杂志上的广告。如果你们有客户，绝对不要外包客户服务。

如果你们在开发一个以剧情见长的电脑游戏，使用第三方 3D 引擎没什么关系。但是如果游戏以炫酷的 3D 特效为卖点，那你们一定要自己写引擎。

我怀疑，唯一不适用于这条规则的情况，就是自己公司的员工比一般人还要无能，这样一来，所有独立研发的尝试都会以失败而告终。没错，很多地方都是这样。对于这种公司，抱歉，我爱莫能助。

36

企业发展战略系列一： 本杰瑞 vs. 亚马逊

2000年5月12日，星期五

作为一名企业家，在创立一家公司的时候，你必须做出一个非常重大的决定，它会影响你做的其他所有事情。无论接下来做什么，你一定要想好站在哪个阵营里，并且据此做出相应的调整，不然很快就会酿成大祸。

这个决定就是选择公司将如何发展：是要慢慢地、自然地发展，从一开始就考虑盈利；还是愿意一炮而红，靠大量资金投入来取得迅猛的增长？

缓慢自然的发展模式先从小处着眼，目标定得低一点、具体一点，然后投入很长的时间，慢慢地把生意做起来。接下来，我会把这个模式称为“本杰瑞模式”（Ben & Jerry's model），因为本杰瑞公司的发展与这种模式非常契合。

另一种发展模式通常被称为“迅速做大”（也叫作“抢占地盘”），即首先募集一大笔资金，然后尽可能快地扩大公司规模，前期完全不考虑盈利。接下来我将称这种模式为“亚马逊模式”（Amazon model），因为亚马逊公司的创始人杰夫·贝佐斯几乎就是“迅速做大”模式的形象代言人。

让我们看看两种模式的不同。首先要问的问题是，你要进入的行业领域，竞争是否激烈？

本杰瑞	亚马逊
存在很多竞争对手	新技术，最开始没有竞争对手

如果你的公司像亚马逊一样，没有任何真正有威胁的竞争对手，那么就有可能在“抢占地盘”中大获全胜，也就是尽快地获取尽可能多的客户，以提高

后续竞争者的准入门槛。但是如果你切入的行业中，有很多老牌竞争对手，那么抢地盘的模式就完全无效，因为必须从竞争对手那里夺取客户群体。

对于进入强手林立的行业，风险投资人一般不会感兴趣。我个人并不惧怕和行业老大竞争，也许是因为我在微软 Excel 团队工作的时候，Excel 正被 Lotus 1-2-3 打得毫无还手之力，后者霸占了几乎全部的市场份额。今天文字处理软件的王者 Word，抢夺的是当年 WordPerfect 的王位，而后者曾经把 WordStar 斩于马下。这些产品曾经都在一段时期内取得了行业的垄断地位。本杰瑞公司并不是第一家售卖冰淇淋的公司，但它如今发展得也很好。如果你全力以赴，打败强大的竞争对手并非不可能（我将在之后的战略系列文章中探讨如何做）。

和同行竞争时面临的问题，还有网络效应以及用户黏度：

本杰瑞	亚马逊
没有网络效应，较低的用户黏度	较强的网络效应，较高的用户黏度

网络效应描述的是这样一种情形：拥有的客户越多，得到的客户也就越多。它来源于麦特卡夫定律^①：一个网络的价值等于它用户数量的平方。

eBay 就是一个很好的例子。如果你想卖掉自己的百达翡丽的古董手表，选择在 eBay 出售能卖个好价钱，因为那里的买家更多；反过来也一样，如果你想买百达翡丽手表，还是可以去 eBay 看看，因为那里的卖家更多。

另一个网络效应极强的例子，是各大公司推出的聊天软件，例如 ICQ 或美国在线即时通讯软件（AOL Instant Messenger）。如果你想和某个人在线聊天，需要使用相同的聊天工具。ICQ 和美国在线目前的用户数量是最多的，你的朋友很可能就在用这些聊天软件，而不是相对冷门的软件，例如 MSN。即使是微软这种技术和经济实力雄厚、营销手段高明的公司，也很难让自家产品打在线拍卖或即时通讯市场，因为这两个领域的网络效应太强大了。

用户黏度即某种商业服务的特性，使得用户一旦使用，就不想更换。即使网络服务差强人意，也很少有人愿意更换互联网服务提供商，因为更换电子邮件地址，再分别通知所有人太麻烦了。同理，如果新的文字处理软件无法读取旧软件生成的文件，用惯了旧软件的人也肯定不愿意更换。

用户黏度有一个更加狡猾的版本，我称之为隐性用户黏度：用户不知不觉地

① 参阅 https://www.princeton.edu/~achaney/tmve/wiki100k/docs/Metcalfe_s_law.html。

就被某种服务“套牢”了。例如，目前出现了一种新服务，比如 PayMyBills.com，它能替你接收各种账单，扫描成电子版方便你在线查看。这类服务通常有三个月的免费期。到期之后，如果你不想付费使用，就必须联系所有寄出账单的服务商，让他们把账单的邮寄地址改回到你家。这种繁冗的重复劳动，通常能阻止一大批人退订 PayMyBills.com 的服务——反正就是每个月多从卡里扣 8.95 美元呗。如果你也这样想，恭喜你，你被“套牢”了！

如果你想进军的行业领域有着天然的网络效应和较高的用户黏度，并且没有成熟的竞争对手，那么你最好采取亚马逊的发展模式。如果你不这样做，将来肯定会有竞争对手这样做，到时候你可能再无立足之地。

下面进行一个快速的案例分析。1998 年，美国在线利用大量的投资，换来每五周一百万用户的迅猛增长。美国在线的产品有很多不错的功能，比如聊天室和即时通讯服务，这些服务都意在培养隐性用户黏度。一旦在聊天室找到一群投缘的朋友，你肯定不想再换互联网服务提供商了，因为这等同于换掉一大批新结识的朋友。我认为，这正是美国在线在众多竞争对手只收取 10 美元月费的情况下，敢于收取约 22 美元月费的关键原因。

我在朱诺公司工作时，管理层不能认同我的这个观点。他们在大批人流开始接触网络的时候，错失了赶超美国在线的最佳时机：朱诺没能下定决心，砸钱换用户，因为他们不想因为融资而稀释了手中股票的价值，没有从战略层面去思考聊天室和即时通讯软件，所以朱诺的产品和服务不具备美国在线赖以取胜的隐性用户黏度。到目前为止，朱诺有 300 万用户，收取 5.5 美元的月费，而美国在线有 2100 万用户，收取 17 美元的月费。这就是差距。

本杰瑞	亚马逊
启动资金很少，很快实现收支平衡	需要大笔启动资金，需要多年才能盈利

本杰瑞公司的启动资金很可能来自某个人的信用卡。公司起步的那几个月甚至几年里，他们必须使用一个能快速盈利的商业模式，而这可能并不是他们最终要实现的商业模式。例如，你可能期望公司成长为年收入 2 亿美元的冰淇淋行业巨头，但是最开始，只能踏踏实实地从佛蒙特州一个小小的冰淇淋店起步，等到这个小店赚了钱（如果能赚钱的话），再用获得的收益去一家一家地扩张分店。本杰瑞集团宣称，他们的启动资金是 12 000 美元。ArsDigita 公司宣称最开始获得的投资是 11 000 美元。这些数额像是万事达信用卡的透支额度。没准真是这样呢。

亚马逊公司融资的速度就好像其他公司花钱的速度一样快，这是有原因的。

他们总是风风火火。在一个没有竞争对手，网络效应异常强大的领域，他们最优的选择就是尽快做大做强，分秒必争。用钱来换时间的方法有很多，通常都很有意思。

- 使用预先建好、装修齐全的行政办公室，而不是传统的办公室。成本：普通办公室的 3 倍；节省时间：几个月到一年，因市场而异。
- 用高得离谱的薪水招贤纳士，或是给程序员配备宝马轿车作为入职福利。成本：技术岗位的薪资上浮 25%；节省时间：在 3 周内招到足够的人才，而通常则需要 6 个月左右。
- 招聘顾问而不是员工。成本：原来的 3 倍；节省时间：公司可以立即进行咨询业务。（顾问没有把足够的时间和精力放在你身上？用现金收买他们吧，直到他们对你忠心耿耿。）
- 不惜血本，做好员工的后勤工作。如果新招进来的明星程序员因为搬家装修等事情而分心，可以聘请高级搬家服务公司来为他们分忧。如果新办公室的固定电话老是装不好，就去买几十部手机。网络连接太慢，拖累了大家的工作效率？多买几家网络服务就好了。还可以设立一个为所有员工服务的杂事总管，帮大家去干洗店取衣服，预订机票酒店，安排机场接送车辆，等等。

本杰瑞公司肯定拿不出钱来做这些事情，所以必须慢慢地增长。

本杰瑞	亚马逊
公司文化非常重要	几乎不可能形成公司文化

当公司以超过每年 100% 的速度增长时，培训导师不可能有时间把公司文化传递给新员工。一个程序员被提升为开发经理后，手下突然多出 5 个昨天刚招进来的程序员，这种情况下，怎么可能有足够的正式培训？网景就是这种情况的极端例子，在短短一年之内，员工数量就从 5 人暴涨到 2000 人。结果就是，公司毫无统一的文化可言，不同的人有不同的价值观，彼此之间还会形成冲突和矛盾。

公司文化对于某些公司并不重要，但却是另一些公司的安身立命之本。本杰瑞公司之所以存在，就是因为创始人具有相同的价值观。他们绝不会放手让公司肆无忌惮地成长，必须要留出足够的时间，让所有员工认同公司的文化。

我们来分析一个关于软件的例子，假设你想进军文字处理软件市场。如今这个市场几乎被微软垄断了，但你独具慧眼，发现有一部分人出于某种原因，绝对无法容忍文字处理软件的崩溃。你决定开发一种超级结实、符合工业强

度标准、永不崩溃的文字处理软件，然后定位这个细分市场，卖给把文字处理软件视为性命的人们。（我承认有点过头了。但是我刚才说了，这只是一个假设，对吧。）

与此相应，你的公司文化应该包含所有和提高代码稳定性有关的因素：单元测试、正式的代码审查、代码规范，以及大型测试团队，等等。要贯彻这些文化并非一日之功，需要很长的时间，才能让这些文化在所有员工的心里扎根。新手程序员在学习如何编写稳定可靠的代码时，需要经验丰富的员工言传身教。

一旦公司的增长速度过快，这种言传身教无法进行，公司的这些价值观就无法传递给新人。新招进来的人没有接受足够的培训，写出不可靠的代码。他们写代码时不会检查 `malloc()` 的返回值，还会以某种无法预料的方式出错。没人有时间去检查他们的代码，教给他们正确的编程方法，于是，你们公司就丧失了和微软 Word 竞争的资本。

本杰瑞	亚马逊
善于从错误中汲取教训	错误通常会被忽略

一个公司如果增长过快，就根本不会注意到过程中犯下的重大错误，尤其是那种因为好大喜功而导致的错误。亚马逊以大概价值 1.8 亿的股票，收购了购物比价网站 *Jungle*，然后才发觉购物比价服务并不利于他们的商业发展，所以关闭了这个网站。如果公司有大把的现金可供挥霍，像这种愚蠢的错误很容易就被搪塞过去了。

本杰瑞	亚马逊
需要很长时间才能把企业做大	企业规模以极快的速度扩张

规模迅速扩大的企业，给人一种成功的印象（或许不只是印象，而是现实）。当求职者看到某家公司以每周招进 30 个新人的速度发展时，他们会产生一种加入一场激动人心的宏大事业之感，仿佛公司上市指日可待。而那些“规模小、发展慢”，只有 12 个人和一条狗的公司，不会给求职者留下什么深刻印象，即使这些公司已经在赚钱，并且有着长远的发展目标。

一条经验之谈是，一家公司要想招揽人才，要么提供优越的工作环境，要么许诺员工将会很快发财，两者必居其一。

你的一部分员工可能会青睐于一家很有可能上市，向潜在员工许诺了很多期权的公司。这些人愿意在这种公司待上三到四年，即使他们憎恨在那里度过

的每一分钟也在所不惜，因为能看到彩虹尽头的宝坛。

如果你选择让公司缓慢地自然生长，那么通向宝坛的路途将更加漫长。在这种情况下，你必须为员工创造一种工作环境，让奋斗过程本身成为一种享受。肯定不能每周工作 80 个小时。办公空间肯定不能是嘈杂的大开间，里面到处都是折叠桌和硬木椅子。必须给员工足够的带薪休假时间。员工之间必须建立起超越一般同事关系的友谊。工作场合也要有适当的社交和社区活动。管理者必须经验丰富，充分信任并支持下属，绝不能像是呆伯特漫画里那些拿着鸡毛当令箭的管理者。通过这些举措，你的公司将会吸引一大批人，他们被“公司上市、一夜暴富”之类的鬼话忽悠了太多次，转而寻求长期稳定的发展。

本杰瑞	亚马逊
可能会成功。肯定不会赔掉太多的钱	有极小的概率成为亿万富翁，极大的概率会失败

如果你遵循本杰瑞的发展模式，并且个人的悟性还不错的话，一般都会取得成功。可能会经历很多的挫折，发展时快时慢，但是除非再来一次经济大萧条，就不大可能赔掉太多的钱，因为投资本来就有限。

亚马逊的发展模式则有一个问题，就是所有人都以为自己是亚马逊。然而，亚马逊只有一个。剩下 95% 的公司花了大笔的风险投资，最后市场反映冷淡，败得体无完肤。如果遵循本杰瑞的模式，至少在用掉一张万事达信用卡的额度之前，就知道大家会不会购买你的产品。

最不可取的发展模式

最不可取的发展模式，就是在本杰瑞公司和亚马逊公司所代表的两种模式之间摇摆不定。

如果你要进军的市场没有强大的对手、较高的用户黏度和网络效应，最好使用亚马逊的模式，否则将重蹈 Wordsworth.com 的覆辙，这家互联网公司比亚马逊早起步两年，而现在仍然默默无闻。更糟糕的是，你的公司可能会变成像 MSN Auctions^① 一样无人问津，看不到任何超越 eBay 的可能。

① 参阅 <http://msnauctions.com/>。

如果你将要进入的是一个较为成熟的市场，采用“迅速做大”的亚马逊模式会让你浪费大笔的资金，就像 BarnesandNoble.com 那样。最佳的策略是细水长流，稳扎稳打、持续盈利。这样，你才有机会在持久战中赶超竞争对手。

还是无法决定？我再来列举一些参考因素。根据你个人的价值取向，你觉得自己更适合成为亚马逊还是本杰瑞的创始人？可以读一读这些公司的发展史，先从亚马逊^①和本杰瑞^②的历史读起。虽然这些读本写得像是名人名言集一样，但是通过阅读，还是能体会出哪种文化更适合你。事实上，本杰瑞发展模式的更好范本是微软公司，市面上有很多关于微软公司发展史的著作。虽然微软早年依靠 PC-DOS 订单取得的进展有“机缘巧合”的因素，但是这家公司一直在赚钱，一直在稳步发展。只有活得够长久，才能等到并抓住机会，一鸣惊人。

还要考虑到你对风险与回报的观念。你是否想过趁着年轻赌一把，争取在 35 岁时当上亿万富翁，即使实现这个梦想的可能性比买彩票中大奖还低？如果答案是肯定的，本杰瑞公司的发展模式无法帮你圆梦。

也许最不可取的发展策略，就是虽然决定公司要成为下一个亚马逊，但在实际运作时表现得像本杰瑞一样（尽管你们自己并不承认）。亚马逊式的公司会抓住一切机会，用金钱换时间。你可能会坚持聘请那些要价符合市场平均水平的程序员，觉得自己聪明又节俭。但实际上你并不聪明，因为这样公司花 6 个月才能招够合适的人，而不是 2 个月，多耽搁的 4 个月可能意味着你错过了圣诞节的购物旺季，也就是说耽误了整整一年。这样一来，整个商业计划可能就泡汤了。你可能觉得在推出 Windows 版产品的同时推出 Mac 版是很高明的商业决策，但是如果这个决策将导致开发者多花一倍的时间才能交付产品，而到头来只多争取了 15% 的客户，你还觉得这么做很高明吗？

两种商业模型都能奏效，但是你必须从中选择一种，并且坚持下去。如果举棋不定或者朝令夕改，做起事来就总会出差错，即使失败了都找不出原因。

① Robert Spector, *Amazon.com—Get Big Fast: Inside the Revolutionary Business Model that Changed the World*.

② Fred Lager, *Ben & Jerry's: the Inside Scoop: How Two Real Guys Built a Business With a Social Conscience and a Sense of Humor*.

企业发展战略系列二： 先有鸡还是先有蛋

2000年5月24日

广告的本质，就是不着痕迹地说谎。大部分企业在开展广告宣传时候，都会针对本行业最不讨人喜欢的特点，把黑说成白（“撒谎”），并且靠不断地重复来加深印象。这就是所谓的“谎言重复千遍就成了真理”。举例来说，乘坐飞机出行通常又挤又难受，空乘人员既无礼又讨厌，我甚至觉得整套商业航空系统就是为了折磨人而存在的。也正因为如此，所有航空公司的广告都在着重强调坐飞机出行是多么舒适和愉快，整个过程中乘客有多么备受呵护。英国航空的一个广告里有这样的情节：一位商务人士坐在飞机里，脑海中浮现出自己变成婴孩，躺在摇篮中的画面。这真是太荒唐了。

一个例子说明不了问题？那就再举两个。造纸公司对于国家森林的破坏是非常严重的，这些古老的森林不是他们的财产，却被其机械夷为平地。所以在造纸公司的广告里，总会展现古老而茂密的松树林海，伴随着“我们十分爱护环境”的解说词。吸烟对健康有着致命的损害，所以烟草公司的广告着重展现的是生机与活力，比如一群笑容满面、健康活泼的人们在户外运动。如此种种，不一而足。

苹果电脑问世之初，没有软件支持 OS X 系统。所以在广告宣传中，苹果公司印发了一本厚厚的软件介绍名册，里面列出了所有支持苹果机的优秀软件。然而半数软件的旁边，都用极小的字体印着“正在开发中”，另一半软件要么功能鸡肋，要么售价不菲。部分软件特别差劲，根本不会有人愿意去买。翻遍这本印刷精美、辞藻华丽的名册（每页只介绍一款软件），你也找不到一款文字处理软件或电子表格软件能够用于只有 128KB 内存的电脑。

NeXT 和 BeOS 平台也有类似的“软件产品指南”。(NeXT 和 BeOS 的粉丝们, 请不要在这里和我讨论你们心爱的系统了, 好吗? 自己去开个专栏吧。) 这些软件产品指南实际上只告诉了你一件事情: 没有软件支持这个系统或平台。以后再看到这种大部头就赶紧跑吧。

Amiga、Atari ST、Gem、IBM TopView、NeXT、BeOS、Windows CE、General Magic……失败的“新平台”太多了。因为顾名思义, 这些平台本身就不是很吸引人, 全靠上面运行的其他优秀软件来汇聚人气。虽然有极少数的例外(我相信, 这篇文章一经发布, 那些神秘莫测、失去大众宠爱的平台——类似 Amiga 或 RSTS-11 这种——的忠实教徒们就会发来大量的邮件), 但是一般情况下, 软件开发者只要有一点常识, 就会选择为用户较多的平台开发软件。比如, 与每天最多只有 10 万人使用的 BeOS 平台相比, 他们更愿意为每天有 1 亿人使用的 Windows 平台开发软件。为各种小众平台编写软件的开发者们用实际行动表明, 赚钱并不是人们写代码的唯一动力。他们对于这些平台的爱只能形容为信仰般的狂热: 干得漂亮, 伙计; 你竟然把 microEmacs 移植到了 Timex Sinclair 1000 上, 太棒了; 我要为你捐 25 美分, 用这笔钱犒劳犒劳自己吧。

扯远了。如果你涉足平台类软件的研发领域, 可能会深受一个俗称“先有鸡还是先有蛋”的问题所困扰。一方面, 只有平台上存在很多可用的优秀软件, 大家才会购买这个平台产品; 但另一方面, 除非一个平台的装机量很大, 否则没有人愿意为它开发软件。这就是问题所在。这有点像是希腊神话中的戈尔迪之结, 或者更形象的说法: 戈尔迪之死亡螺旋。

在企业发展的众多战略问题中, “先有鸡还是先有蛋”问题及其变种是很重要的一类, 你应该对它们有所了解。不过即使不了解也有可能成功: 史蒂夫·乔布斯就是最好的例子, 他向来否认这种问题的存在, 并且在其职业生涯中, 两次战胜了这个问题。但是乔布斯只有一个, 普通人并不具备他那种强大的现实扭曲力场, 所以大家还是虚心一点, 努力学习为好。

“先有鸡还是现有蛋”问题的经典适用领域是软件平台领域, 但其适用领域并不仅限于此, 我们再举一例。每个月, 几百万家信用卡公司会通过邮政信件寄出数不清的账单。人们收到账单后签一张纸质支票, 装在信封里寄回给信用卡公司。大量的信封被装在大箱子里, 运到其他国家, 那里廉价的劳动力打开信封并进行财富处理。但是此项业务流程花费不菲: 我最近一次听到的数字是, 每份账单的平均处理成本超过了 1 美元。

我们这些互联网的弄潮儿会觉得这简直就是个笑话。你可能会说：“用电子邮件把账单寄给我，然后我就可以在网上付钱给信用卡公司了！这样做的成本，连一分钱的十万分之一都不到。”

你说的没错。事实上，有很多公司都在做这种业务，专业名称叫作账单承兑。涉足此领域的其中一家公司是（猜猜是谁）微软。他们提供的解决方案叫 TransPoint^①，它以网站的形式存在，用户登录后，系统显示该用户的账单，然后用户可以直接向网站付费来支付账单。

假设使用微软的这个系统来承兑账单，你要做的就是每隔几天登录一下网站，看看有没有未支付的账单。假如你每月平均有 10 份账单，那还不太麻烦。但是还有一个问题：只有屈指可数的商家使用微软的系统。对于其他商家寄出的账单，你只能到别的地方处理。

最终你会发现，这样做得不偿失。如果能有 1 万人使用这个系统，我就会很吃惊了。微软不得不到每个商家那里，请他们使用在线系统向用户发送账单。商家会问：“好啊！需要花多少钱？”微软说：“每单服务费 50 美分，比 1 美元便宜多了！”商家说：“可以，还有其他的成本吗？”微软说：“有啊，需要安装并配置一套软件，把你们的系统和我们的系统进行对接，还有调试和售后服务费，总共加起来是 25 万美元。”

然而，微软的系统上只有那么几个用户，哪个商家会傻到投入 25 万美元，就为了在 37 位用户身上节省 50 美分处理成本。可恶的“先有鸡还是先有蛋”问题，再一次露出了它的狐狸尾巴！在商家进驻之前，顾客不会上门；而如果没有顾客，又有哪个商家敢吃螃蟹！最终，可能微软要靠雄厚的资本来摆脱这一困局了^②。然而对于规模较小的公司，这是不现实的。那该怎么办？

如何打破“先有鸡还是先有蛋”的怪圈，连“鸡”带“蛋”一勺烩呢？我们来看看自从 IBM-PC 问世以来，个人电脑上软件平台的发展历史，说不定会得到一些启发！

大部分人都误以为 IBM-PC 必须安装 PC-DOS 系统，其实不是这样。IBM-PC 刚出现的时候，消费者可以选择三种操作系统：PC-DOS、XENIX（一个功能孱弱的 8 位 UNIX 发行版本，发行公司是微软——这可不是我编造的）以

① TransPoint 早已不存在了。我想我是对的。

② 我想我错了。

及一种叫 UCSD P-System^①的系统。令人难以置信的是，UCSD P-System 非常类似于 Java：容易上手、运行缓慢、跨平台的字节码，但是比 Java 早了 20 年。

如今大部分人都没听说过 XENIX 或者 UCSD 这些老古董。年轻一代可能会猜想，微软当年是通过强大的营销能力或其他手段拿下了操作系统市场。这完全不是事实，因为当时的微软真的是微软。拥有强大营销能力的是 Digital Research 公司，他们也推出了自家的操作系统产品。那么，是什么让 PC-DOS 在三足鼎立中突出重围的呢？

在 PC 诞生之前，唯一能买到的操作系统是 CP/M，但它的市场非常小，因为安装了 CP/M 系统的计算机每台售价大概 1 万美元。这个系统既不稳定又价格昂贵，用户体验也不好。购买这个系统的人，大概都是为了使用上面的文字处理软件，因为当年 WordStar 的 CP/M 版非常不错，相比之下，苹果 2 代电脑的文字处理功能几乎就没法用（刚开始时连小写字母都不支持）。

关于 DOS 有一条重要的事实：从 1.0 版本开始，DOS 都内置了对于 CP/M 软件的兼容性支持。它不仅有一套自己的新编程接口（资深程序员可能记得，这套接口叫 INT 21），而且完全兼容之前 CP/M 的编程接口。几乎所有的 CP/M 软件，都可以在 DOS 上运行。事实上，WordStar 只修改了源代码中的一个字节，就被移植到了 DOS 平台上。（我早就忘了是哪个字节，也许比我更厉害的程序员还记得。）

我觉得有必要再说一遍。WordStar 在移植到 DOS 平台上的时候，只修改了代码中的一个字节。好好体会一下。

程序员们作何感想？

DOS 平台之所以能很快流行起来，是因为从一开始就支持很多软件。这归功于蒂姆·帕特森的远见，是他规划并加入了 DOS 对 CP/M 软件的兼容性功能。即使在软件行业的蒙昧时期，还是有聪明人懂得如何处理“先有鸡还是先有蛋”的问题。

纵观 PC 平台的发展史，PC 用户只有两次重大的平台转换：Windows 3.x 和 Windows 95。在这两次转换中流失到其他平台的用户数量非常少。如果你喜欢“微软征服世界”的阴谋论，我无话可说。我认为，造成这种局面的是一

^① 参阅<http://www.threedee.com/jcm/psystem/>。

个更加有趣的原因，还是和“先有鸡还是先有蛋”有关。

首先来看 Windows 3.x。这次平台转换事件的重点在于“3”。为什么使用 Windows 1.0、2.0 以及之后 Windows 286、386 的人很少？是因为微软连发 5 个版本才把软件做好吗？不是这样。

真实的原因比这更加微妙，与计算机硬件的一些非常不起眼的功能特性有关。这些功能特性最早在英特尔 80386 芯片中出现，而 Windows 3.0 是第一个强制要求英特尔 80386 的系统版本。

- 功能特性一：旧版 DOS 程序在屏幕上进行显示时，会直接复写和屏幕上字符单元对应的内存位置。这样保证了输出的速度，程序显示流畅。但是 Windows 使用的是图形化的操作界面。在旧版的英特尔芯片上，微软工程师别无选择，只能让系统在运行 DOS 程序时切换到全屏模式。但是在 80386 上，他们可以设置一个虚拟内存区域，在某个程序想直接复写屏幕对应的内存位置时，通过系统中断来通知操作系统。这样 Windows 就可以立刻把对应的文字显示到图形化界面中的某一个屏幕区域里。
- 功能特性二：旧版 DOS 程序假定运行的时候是独占机器资源的。这会导致多个 DOS 程序同时运行时互相打架。而英特尔 80386 芯片拥有创建“虚拟”PC 的能力，每个虚拟 PC 都像是一个独立完整的 8086 机，所以旧版的 PC 程序可以继续假定自己独占机器资源，但从系统来看，多个应用程序是可以同时运行的。

所以基于英特尔 80386 芯片的 Windows 3.x 是第一个能同时运行多个 DOS 程序的操作系统。（其实之前的 Windows 386 已经实现了这一点，但那时 80386 机还非常罕见、价格昂贵，直到 Windows 3.0 面世之后，情况才有所改善。）也就是说，Windows 3.0 是第一个能让大家使用所有旧软件的操作系统。

再来看 Windows 95。它的兼容性也非常好，在推出崭新 32 位 API 的同时，仍然能正常运行旧的 16 位软件。微软对于保证产品的兼容性是非常执着的，当年投入大量人力和财力，测试了市面上能找到的所有旧软件在 Windows 95 上的兼容性。琼恩·罗斯是电脑游戏《模拟城市》的缔造者，他告诉我，在 Windows 3.x 上运行的最初版中遗留着一个 bug：会读取已经释放了的内存。但是游戏在 Windows 3.x 平台上却能正常运行，因为由于系统原因，这块内存并没有被马上释放。下面发生的事情令人称道：在 Windows 95 的 beta 版中，《模拟城市》无法通过测试。微软追查到了 bug，然后在 Windows 95 的代码中为这个游戏增加了专门的处理逻辑。如果当前运行程序为《模拟城

市》，系统会让内存分配器以特殊模式运行，在代码中调用了 free 的内存不会被马上释放。正是这种对于产品兼容性的执着，让人们放心地升级到 Windows 95。

你现在大概有所体会了，要想打破“先有鸡还是先有蛋”的怪圈，只需要提供某种向后的软件兼容模式：要么提供很多的鸡，要么提供很多的蛋（二者其实没什么区别），然后就坐等鸡生蛋、蛋生鸡，财源滚滚而来吧。

再回到账单承兑业务。刚才我们说到，这个领域的“先有鸡还是先有蛋”问题，表现为在线服务只能覆盖全部实体账单中的很小一部分，所以没人愿意使用在线服务。如何解决这个问题呢？微软没想到办法。但是 PayMyBills.com（以及其他六七个硅谷初创公司）找到了解决的办法。他们提供了一种向后兼容模式：如果商家不支持在线系统，就让他们把实体账单寄到帕洛阿尔托市的大学路，公司在那里雇了一批人来拆信、扫描账单。这样，你就可以在网站上看到所有账单了。这个系统适用于所有提供账单的商家，客户一定很愿意使用它。但是仔细想想，这种向后兼容模式非常折腾：首先是 Visa 信用卡联盟的银行把电子版的账单通过打印机变成纸质账单，然后塞进信封，经过 1500 英里的邮政运输送到加利福尼亚州；信封打开后，抽出里面夹带的广告传单（宣传某个“免费”但实际上收费 9.95 美元的电台服务，而这份广告传单最终会被统一丢弃在某个垃圾填埋场里），把纸质账单扫描进电脑，传到网站上；最终这份账单到达了它本来就应该通过网络到达的归宿。但是这种愚蠢的向后兼容模式最终会消失，因为 PayMyBills.com 不像微软，成功地吸引了越来越多的消费者，所以他们很快就可以去 Visa 信用卡联盟的银行谈判了：“嘿，你们的 93 400 个客户都在使用我们的服务。只要直接把账单电汇给我们，每个月就能省下 93 400 美元，何乐而不为呢？”到时候 PayMyBills.com 肯定稳赚不赔，而微软仍在想办法争取第二家签约商户，还不如去乔治亚州碰碰运气，也许能发展得快一点。

对于“先有鸡还是先有蛋”问题认识不足的公司，通常都会做出“精卫填海”式的举动：他们疯狂的商业计划，需要 9300 万人合作才能实现。我听过的最愚蠢的一个想法是 ActiveNames^①。该公司想让每个人都在电子邮件客户端上装一个小插件，这个插件能根据用户名，从该公司的中央服务器上得到真正的电子邮件地址。这样做的“好处”是，你给别人留联络方式的时候，不用说电子邮箱地址是 `kermit@sesame-street.com`，而是说你的 ActiveName 用

① ActiveNames 已经倒闭了。参见帕翠西娅·奥德尔的文章《ActiveNames 关门歇业》，2001 年 4 月 11 日。网址是 <http://www.chiefmarketer.com/activenames-shutters-business/>。

户名是 spolsky,但如果他们想给你发邮件的话,也需要安装这个小插件。这个想法真是蠢到家了,不值一晒。

综上所述,如果你所在的行业存在“先有鸡还是先有蛋”的问题,你最好准备一套向后兼容方案,以打破这个怪圈,否则有可能很长时间都没有起色(甚至有可能永无出头之日)。

还有很多其他公司出色地认识并解决了他们面临的“先有鸡还是先有蛋”的问题。Transmeta 最近发布了一款基于 x86 架构的 CPU,在很长一段时间里,该公司是除英特尔之外,唯一认识到只有支持 x86 代码的 CPU 才能被广泛接受的公司。日立、摩托罗拉、IBM、MIPS、美国国家半导体公司,以及其他不可胜数的公司都试图让市场接受自己推出的新指令集。而 Transmeta 从一开始就认识到,如果装有某块芯片的电脑连 Excel 都运行不了,那么这种芯片肯定卖不出去。

企业发展战略系列三： 让我回到过去

2000年6月3日，星期六

要想从竞争对手那里赢取客户，就必须了解客户如果放弃现有产品转而使用你的产品，会遇到哪些障碍，并且对这些障碍有充分和切实的体会。如若不然，你将无法得到足够的客户，企业也就无法经营了。

214

第三部分 乔尔语录：中心明确的胡思乱想

前几章探讨了两种公司发展模式的差异：本杰瑞式的公司试图挑战存在多年的竞争对手，而亚马逊式的公司则在全新、没有行业竞争的领域“抢占地盘”。20世纪90年代，我在微软 Excel 团队工作，当时的微软完全属于本杰瑞模式。当时，Lotus 123 是电子表格软件领域最强大的产品，它在这一领域几乎形成了完全的垄断。虽然有些新的电脑用户从一开始就使用 Excel，但大多数时候，微软必须从竞争对手那里抢夺客户。

处在这种地位的公司，必须具有直面现实的勇气。有些公司连这点都做不到。我工作过的上一家公司是朱诺，那里的管理者就不肯承认，竞争对手美国在线已经取得市场统治地位。他们总是说“还有上百万人没有使用过网络服务”，还说“每个行业通常都有两大巨头，比如杂志领域的《时代》和《新闻周刊》，饮料市场的可口可乐和百事可乐，等等”。他们唯一不敢承认的是：“我们必须从美国在线那里把用户抢过来。”我不知道他们在害怕什么，也许是不敢“唤醒沉睡的猛兽”。朱诺有一位明星级程序员（不是我）曾经斗胆在公司会议上问了一个简单的问题：“为什么我们不努力促使美国在线的用户转而使用我们的服务呢？”此言一出，他就被请出了会议室，上司劈头盖脸地骂了他一个小时，然后撤销了之前许诺的升职奖励。（猜猜后来谁另谋高就了？）

选择进入一个竞争激烈的市场本身并没有错。事实上，即使是非常新颖的产

品，也会面临来自传统的挑战，比如 eBay，就面临线下跳蚤市场的冲击。不害怕竞争。如果你的产品在某些方面比竞争对手的更加优秀，用户还是很有可能转而使用你的产品。但是你必须采取一定的战略措施，把眼光放长远一些，而不只是看到当下的表面现象。

想让用户改用你的产品，唯一需要采取的措施就是排除转换过程中的所有障碍。假设时光倒流，回到 1991 年，Lotus 123 拥有电子表格市场几乎 100% 的占有率，此时你被任命为微软 Excel 团队的产品经理。你需要思考的是：用户在改用 Excel 的过程中会遇到哪些障碍？哪些因素阻碍了人们尽快成为 Excel 的使用者？以下我列举了一些想使用 Excel 的人可能遇到的障碍：

障 碍
(1) 必须了解 Excel，并且认同它是更好的产品
(2) 必须购买 Excel
(3) 必须购买 Windows 才能运行 Excel
(4) 必须把使用 123 制作的电子表格转换为 Excel 可识别的格式
(5) Excel 无法运行键盘宏命令，所以必须重写
(6) 必须学习一套新的用户界面
(7) 需要一台更快、内存更大的电脑

还有更多的障碍，在此不一一列举。客户更换产品的过程，可以看作参加一场障碍赛，而以上列举的就是途中的重重关卡。出发的时候可能有 1000 名选手，其中一半会在轮胎阵中被绊倒；幸存者中又有一半跳不过障碍墙；剩下的大部分会在绳梯上摔下来，跌进泥潭里……最后只有一到两个人能克服九九八十一难，到达终点。如果障碍的数量多至 8 到 9 个，那么几乎所有人都会拥有一个理由，打消使用新产品的念头。

通过上面的计算还可以看出，如果你想赶超竞争对手，最重要的办法就是尽量减少障碍的数量。只要减少一个障碍，销量就有可能翻倍。再减少一个障碍，销量有可能再翻一倍。当年的微软对于这张障碍清单上的所有条目都采取了相应的措施：

障 碍	解决方案
(1) 必须了解 Excel，并且认同它是更好的产品	广告宣传 Excel，免费发放示范磁盘，在全国巡展，演示产品
(2) 必须购买 Excel	为之前使用 123 的用户提供购买 Excel 的特殊折扣

障 碍	解决方案
(3) 必须购买 Windows 才能运行 Excel	发行一个 Windows 特别版, 免费提供 Excel
(4) 必须把使用 123 制作的电子表格转换为 Excel 可识别的格式	让 Excel 识别 123 电子表格文件
(5) Excel 无法运行键盘宏命令, 所以必须重写	使 Excel 能运行 123 的宏命令
(6) 必须学习一套新的用户界面	让 Excel 支持 Lotus 的键盘快捷键, 让习惯使用 123 的人也能轻松上手
(7) 需要一台更快、内存更大的电脑	等待摩尔定律解决电脑计算能力不足的问题

这些措施的效果非常明显。微软对于清除障碍的不懈努力, 换来了 Excel 市场份额的缓慢提升, 而且这些份额都是从 Lotus 那里“虎口夺食”而来的。

在某个领域的新旧垄断势力交替中, 存在一个神奇的“转折点”: 一夜之间, 你发现自家产品的市场份额从 20% 暴涨到 80%。局势扭转的速度非常快。(从 VisiCalc 到 123 再到 Excel, 从 WordStar 到 WordPerfect 再到 Word, 从 Mosaic 到网景再到 Internet Explorer, 从 dBase 到 Access, 都是这样。)这通常是因为一旦你击溃了最后一个障碍, 大家就会顺理成章地选择你的产品阵营。

首先要清除的当然是那些最明显的障碍, 有了应对方案之后, 就应该找出那些不是特别明显的障碍。这方面的策略不太明朗, 因为在导致人们不愿意更换产品的众多因素中, 有一些是比较微妙的。

举一个例子。今年夏天我在海滨度假, 但所有的账单还是寄到了我在纽约市的住所。我平时也经常出门旅行。有一个不错的网络服务 PayMyBills.com, 它试图简化人们处理账单的过程。客户先把所有账单都寄到 PayMyBills 公司, 他们收到后把账单扫描成电子版, 然后客户就可以随时随地在网上查看或付款了。

PayMyBills 每月收费 9 美元, 还算公道, 于是我打算使用它的服务。但是之前互联网经济类服务给我留下的印象不是很好, 有一个叫 Datek 的服务, 总是算错帐, 而且出具的账务报表非常不专业, 我甚至怀疑他们没有相关资质。所以对于 PayMyBills 的服务, 我的心态是愿意试试看, 要是效果不好, 我希望能够回到原来处理账单的方式。

但问题就出在这里。一旦使用了 PayMyBills, 如果想停止, 就必须打电话通知每家信用卡公司把邮寄地址改回来。因为害怕这种麻烦, 我决定还是不用他们的服务了。之前, 我称这种营销策略为隐性用户黏度^①, 并且表示了赞

^① 参见第 36 章。

赏，但假如你想采取这种策略，要承担的风险是，一旦潜在客户察觉了你的营销手段，你就有麻烦了。因为这对人们决定使用你们的服务造成了障碍。这种障碍并不来自于过高的准入门槛，而是来自于担心一旦“上了贼船”，就很难脱身。

这让我想起了 Excel 4.0 版本时的市场份额暴增。这个转折点之所以出现，最主要的原因是 4.0 是首个支持无缝回写 Lotus 电子表格功能的 Excel 版本。

没错，我说的是写，不是读。微软发现，阻止人们马上开始使用 Excel 的最大问题是，身边的同事还在使用 Lotus 123。没人愿意做第一个吃螃蟹的人，一款软件如果连创建的文档别人都读不了，那就根本没法使用：这又是一个典型的“先有鸡还是现有蛋”的问题^①。如果公司的绝大部分人都在使用 123，那么极少数拥护 Excel 的人也不得不随大流。而如果 Excel 无法输出 123 格式的电子表格文件，那么基本上就意味着没有人可以用 Excel 办公。

在抢占市场份额的战役中，你必须将所有障碍各个击破。遗漏了其中的一个，你的公司就会损失掉 50% 的潜在客户，就是说无法获得 50% 以上的市场份额，也就永远不可能战胜任何业已取得统治地位的竞争对手，从而沦为“先有鸡还是先有蛋”问题的牺牲者（变成炒鸡蛋）。

问题是，很多管理者思考问题的方式都是走一步看一步，就像是蹩脚的棋手，不知道提前多想几招。他们大多会说：“让人们使用我的产品固然很重要，但为什么要浪费有限的预算，去开发让人们可以轻松离开的功能呢？”

这种对于发展策略的思考显得不够成熟，让我想起某些自命清高的独立书店经营者。他们说：“为什么要把书店的环境布置得那么舒适？我希望顾客把书买走，而不是赖在店里看书！”然后巴诺书店出现了，他们的书店配备了松软的沙发和香气扑鼻的咖啡，店员的态度让你觉得他们巴不得让顾客整天赖在书店里，不买也没关系。这吸引了一大批顾客在巴诺书店里一坐就是几个小时，用脏兮兮的手在书架上摸来摸去。事实上，顾客在书店的时间越长，就越有可能找到并购买心仪的书，二者呈线性正比关系。即使是爱荷华市最冷清的巴诺书店，每分钟都能进账几百美元，而与此同时，独立书店则纷纷濒临破产。曼哈顿上西区的莎士比亚书店（Shakespeare and Company）之所以关门，并不是因为巴诺书店采取了低价倾销策略，而是因为巴诺书店的顾客更多。

^① 参见第 37 章。

更为成熟的发展策略是，不让潜在客户感到任何使用产品的压力。试图“黏住”一个还未成为客户的人，不是什么好主意。当你拥有 100% 市场份额的时候，再来和我谈“用户黏度”吧。在那之前，任何试图把客户“黏住”的想法都操之过急，一旦用户有所警觉，就会被你的强势“撵走”。没人愿意为了使用一个产品，就把将来选择的自由统统抹杀。

我们再来看一个当前的例子：ISP（网络服务提供商）。这是一个竞争达到白热化阶段的市场，然而几乎没有 ISP 提供这种服务：当用户不再使用他们的服务时，ISP 会把发到原来地址的电子邮件转发到用户指定的新地址。我很惊奇为什么大家都没有意识到，“各扫门前雪”的观念是非常狭隘的。如果你经营了一家小型的 ISP，正在想办法招揽客户，就应该意识到，人们面前最大的障碍就是要逐个通知朋友们电子邮件地址的变更。一想到换地址这么麻烦，人们试用服务的兴趣就被打消了。即使有人真的尝试了你们的服务，他们在短期内也不会把新地址告诉朋友们：万一发现不好用呢。这意味着新地址不会收到多少邮件，也就是说，他们无法真正体验你们的服务，感受其特别之处。这对于用户和公司“双输”的局面。

假如有一家冒天下之大不韪的公司，敢于作出如下承诺：“放心尝试我们的服务吧。如果你不喜欢，我们将永久保留你的电子邮件地址，并把寄到该地址的邮件免费转发到其他任何 ISP 提供的电子邮件地址。一旦使用，终生享用。用了我们的服务，就可以毫无后顾之忧地尝试各家的 ISP 服务，我们将为你提供终身免费的转发服务。”

当然，这种营销策略会让某些业务经理感到不舒服：“为什么要降低用户离开我们服务的门槛？”只有目光短浅的人才会这样想。很多人还不是你们的客户，如果过早试图把他们“黏住”，只能适得其反。但是，通过郑重承诺“即使不喜欢我们的服务，也可以轻松离开”，人们面前的障碍就又少了一个。正如我们刚才了解到的，哪怕只减少一个障碍，都会对用户转化率造成巨大的正面影响。假以时日，当最后一个障碍轰然倒地，客户会像潮水般汹涌而至，你的公司将开创一个属于自己的时代，直到后来者用同样的方法取而代之。

企业发展战略系列四： 膨件和二八法则

2001年3月23日，星期五

微软电子表格旗舰产品 Excel 的 5.0 版于 1993 年问世。它在当年绝对称得上是大制作：安装所需硬盘空间达到了 15MB。那个时候，大家还对自己只有 20MB 硬盘空间的第一台 PC 记忆犹新（1985 年前后），所以 15MB 已经显得很大了。

而 Excel 2000 推出的时候，所需安装空间居然达到了 146MB，是上一个版本的将近 10 倍！你可能会想，微软的程序员也太无能了吧？

错。

你大概以为接下来我会像网上其他无聊的文章一样，开始抱怨所谓的“膨件”（膨胀的软件）。如今的软件太臃肿了，简洁优雅的 edlin 和 vi 远比 Word 和 Emacs 要好多了，等等。

你想错了！我不会抱怨膨件一无是处，因为我并不认同这种看法。

以 1993 年的硬盘成本来看，微软 Excel 5.0 所占用硬盘空间的价值大概为 36 美元。而以 2000 年的硬盘成本来看，微软 Excel 5.0 所占用硬盘空间的价值大概为 1.03 美元^①。实际上，Excel 的体积变小了！

到底什么是膨件呢？《行话大全》^②对它的定义充满了偏见：“一种提供的功能极少，却需要极大硬盘和内存空间的软件。常见于软件和操作系统的升级

① 成本核算数据来自网络，同时考虑到了通胀因素。

② 参见 www.catb.org/~esr/jargon/。

版本。这个术语指代的事物及其出现原因，通常与 Windows/NT 平台有关。”

很容易看出这帮人非常厌恶 Windows。自从 Windows 386 推出虚拟内存功能以来（1989 年），我已经十多年没有遇到过内存不足的情况了。目前，每 MB 硬盘存储空间的成本低至 0.0071 美元，而且仍在直线下降中，就像一只从树上跳下来，学习飞行的绵羊一样。

也许利努斯·阿克隆德能给出让人信服的解释。他在个人网站上写道：“这些臃肿的程序有一个最大的缺点，就是即使要完成的事情非常不起眼，也必须加载这个庞大的程序。它把所有的内存都榨干了……系统没有得到有效的利用。膨胀阻碍了系统性能的正常发挥，而这毫无必要。”^①

哦，原来膨胀把内存都榨干了。等一下，这并不是真的。自从 1987 年的 Windows 1.0 开始，操作系统就能够按需加载内存页了。也就是说，如果软件体积为 15MB，而运行中代码的体积相当于 2MB 大小的内存页，那么系统只会把这 2MB 从硬盘加载到内存。事实上，在现代的 Windows 版本中，系统会按照加载的先后顺序自动重排硬盘上的页面数据，让程序在下次加载的时候变得更快。

毋庸置疑，计算机发展到今天，已经变得能力过剩、价格偏低。如今，系统加载大型程序的速度，比五年前加载小型程序的速度还快。那么，抱怨膨胀的人到底有没有真正令人信服的理由呢？

一位不愿透露姓名的程序员向我们提供了线索。他用几个小时的时间来剖析一个微软的小工具，对于它占用 1MB 的硬盘空间表示愤怒（在他写文章的时候，1MB 硬盘空间的成本大概是 0.0315 美元）。他认为，这个小工具的体积应该缩小 95%。好笑的是，他剖析的小工具叫作 RegClean。你可能没听说过这个程序，它的功能是扫描 Windows 注册表，寻找并删除那些没有使用的东西。对于清理注册表无用信息的软件大小都如此斤斤计较，只能解释为一种强迫症的表现。我开始怀疑问题出在抱怨膨胀的心态上，而不是软件设计思想上。

另一方面，程序员写出越来越臃肿的软件其实是有很多正当理由的。最重要的理由是，如果程序员不需要过分关注代码的体积，就可以更快地完成软件的开发。这意味着用户将更早获得功能特性更加丰富的软件，而众多功能特

^① 利努斯·阿克隆德，《为什么我不喜欢膨胀：一次有充分理由的咆哮》，1998 年刊登于利努斯的个人主页。

性通常让人们的生活变得更好（如果人们使用这些功能的话），即便不然，也没什么坏处（可以不用这些功能）。软件厂商也可以选择在产品发售前两个月停止开发，努力将代码体积压缩到 50%，但这种行为对于消费者的好处几乎可以忽略不计。如果用户的硬盘总是塞得很满，那么缩小一半软件体积充其量可以让用户多下载一首 Duran Duran 乐队的 MP3 歌曲文件。但是推迟两个月发布软件的新版本给企业带来的损失是显而易见的，毕竟少卖了两个月的软件。

很多开发者都被经典的二八法则误导了。这条看上去有点道理的法则：80% 的人只会使用 20% 的产品功能特性，所以只要实现 20% 的功能特性，还是能卖出 80% 的软件。

可惜它的谬误在于，不同的人使用的 20% 功能特性不尽相同。在过去 10 年里，我就听说过几十家公司（他们就不能从彼此身上吸取教训）尝试推出只实现 20% 功能的“轻量级”精简版文字处理软件。这种想法在个人电脑诞生的时候就有了。通常，软件公司把开发的文字处理软件交由某个记者来评测，而记者评测的方式就是用这个软件来写评测文章。因为绝大部分新闻稿都有严格的字数限制，所以记者需要文字处理软件具有“字数统计”功能，但精简版通常不会实现这个功能，因为“80% 的人都用不上”。所以最后评测文章的基调经常是，一方面承认轻量级软件的想法是好的（顺带贬损两句膨胀的坏处），另一方面则认为该软件根本没法用，因为“连字数统计功能都没有”^①。如果这种事情每发生一次我就能得到 1 美元^②，那我肯定乐坏了。

你在推销“轻量级”产品的时候可能会说：“嘿，它很小，只占用 1MB 硬盘空间。”用户听了先是很高兴，然后就会询问是否具备某些对于他们来说不可或缺的功能特性。如果答案是否定的，那么他们肯定不会买你的产品。

结论是，如果你采取二八法则作为战略，产品很可能会卖不动。现实就是这样。从软件行业诞生之日起，这种想法就一直存在，但从来没有产生过很好的经济效益。然而令人惊讶的是，还是有很多信奉“唯快不破”的企业家把公司的命运寄托在这个战略上。

杰米·扎文斯基对此有一针见血的论述。在谈到具有划时代意义的初代网景

① 大卫·柯西，《寻求微软 Office 的便宜替代品？试试 ThinkFree 吧》，2001 年 2 月。参见 <http://www.linuxtoday.com/developer/2001020700806OPDTSW>。

② 查尔斯·贝尔曼，“Yeah Write”，WashingtonPost.com，1997 年 6 月 27 日。参见 http://washingtonpost.com/wp-srv/tech/reviews/finder/rev_1030.htm。

浏览器的时候，他这样写道^①：“谋智（Mozilla，即网景浏览器 1.0 版）之所以庞大，并不是因为有很多无用的垃圾功能，如果真是这样，那事情就简单了。谋智之所以庞大是因为用户需求是庞杂的，而用户需求之所以庞杂是因为互联网太大了。市面上存在很多轻量级的网络浏览器，但是几乎没提供什么有用的功能。我们创造谋智的时候，并没有把开发出一款尽善尽美的产品作为目标。”

① 杰米·扎文斯基，《复活节彩蛋》，1998 年。参见 <http://www.jwz.org/doc/easter-eggs.html>。



企业发展战略系列五： 开源软件经济学

2002年6月12日，星期三

我在大学时期学过两门经济学课程：宏观经济学和微观经济学。宏观经济学充斥着各种不符合实际的理论，比如“低失业率导致通货膨胀”。而微观经济学里却有很多既有趣又有用的理论，尤其是那些关于供需关系的理论，能很好地解释一些实际现象。比如，如果竞争对手采取降价促销的手段，你就不得不跟着降价，否则消费者对于你产品的需求量就会减少。

在本章中，我将阐述如何运用这些概念来解读一些知名电脑公司的遭遇。经过多年的观察与思考，我注意到了开源软件的一个有趣特性：有些公司斥巨资支持开源软件的开发事业，背后的原因并不是他们突然放弃了资本主义信仰，对“和言论自由一样的自由”^①理念一见钟情，而是出于自身发展的一种商业战略思考。

市面上的所有产品都有替代产品和配套产品。如果你觉得原产品比较贵，可以考虑购买替代产品。鸡肉就是牛肉的替代产品。当牛肉价格上涨的时候，对于鸡肉的需求也会跟着上涨，养鸡专业户就能卖出更多的鸡肉。

配套产品就是你通常会和另一种产品一起购买的产品。汽油和汽车互为配套产品，电脑硬件和电脑操作系统软件是一对典型的配套产品，婴儿保姆是高级餐厅的配套产品：在小型城镇里，每次五星级饭店推出情人节促销买一送一的活动时，当地的婴儿保姆（实际上，连9岁的孩子也成为了其中一员）

^① “和言论自由一样的自由”是自由软件倡导者提出的一个术语，参见 <http://www.gnu.org/philosophy/free-sw.html> 对此的探讨。他们的意思是，你拥有使用、复制、散布、研究、改写和再利用该软件的自由，但软件本身不一定免费。

都会索要双倍的工钱。

在其他条件对等的其前提下，配套产品价格降低会导致原产品的需求上升。

我再重复一遍，以防你因为倦怠而错过了这个概念，它真的很重要。当配套产品价格下降的时候，对原产品的需求会上升。举个例子，如果到迈阿密的机票降价，迈阿密的酒店房间需求量会上升，因为更多的人会乘飞机到迈阿密，他们需要住处。当电脑越来越廉价，吸引了更多人来购买的时候，因为电脑离不开操作系统，所以对于操作系统的需求会上升，这意味着，操作系统的定价可以上涨。

行文至此，自由软件倡导者可能会表示：“等一下！但 Linux 是免费的！”这会导致一定的困惑。首先，经济学家所说的价格是总体价格，其中包括一些不可见的成本，比如设置系统需要的时间，对每个软件使用者的培训，以及修改现有工作流程的成本，等等。我们不妨称之为“要拥有某个软件所付出的总成本”。

其次，自由软件倡导者通过“和免费啤酒一样的免费”的论点，试图规避经济学规律的影响：因为价格为零，所以影响为零。举例来讲。当 Slashdot 网站询问 Linux 开发者莫什·巴，未来 Linux 内核版本是否会向后兼容现有硬件驱动的时候，他说没有必要这样做。“商业软件因为包含税金，每行经过调试代码的均摊成本是 50~200 美元，而这些成本对于开源软件并不适用。”^①莫什进一步称，即使 Linux 内核的每次升级都导致之前所有驱动程序失效也没关系，因为重写所有驱动的成本为零。大错特错。他其实是说，用一点编程时间来让内核向后兼容的代价，和使用大量编程时间来重写每个现有驱动程序的代价相等，因为都要乘以其“价格”，即他认为的“零”。但是很明显不能这么算。重写每个硬件驱动需要上万甚至上千万小时的开发时间，其代价绝不会是零。如果内核版本升级导致现有驱动失效，那么在重写完毕所有驱动之前，Linux 将再一次在市场上陷于被动低位，因为它不能支持现有的硬件。为什么不把这些“零成本”的开发时间，投入到对 Gnome 的完善或是对新硬件的支持上呢？

无论对于商业软件还是开源软件，经过调试的代码都不是免费的。即使你不需要花真金白银来买，它还是有机会成本和时间成本的。开发者自愿为开源事业贡献的编程人力资源并不是无限的，而且开源项目之间对于这些有限的

^① Slashdot 网站对莫什·巴的采访，2002 年 6 月 7 日。参见 <http://interviews.slashdot.org/interviews/02/06/07/1255227.shtml?tid=156>。

资源存在竞争关系，只有最有魅力的项目才能吸引超过所需的编程资源。总而言之，对于那些试图证明“和免费啤酒一样的免费”软件不适用于一般经济规律的观点，我不敢苟同，因为它们会在运行时抛出除零错误。

开源软件并不能摆脱物理和经济规律的限制。我们在这方面看到了很多失败的尝试，例如 Eazel、ArsDigita 以及之前一家叫 VA Linux 的公司，等等。但是开源世界中很少有人能解释清楚，为什么有很多大型上市公司，在背负着广大股权人盈利预期的前提下，还要投入大笔资金给庞大的程序员团队，来支持开源软件的开发事业。这种现象可以通过刚才提到的补充法则来解释。

我再重复一遍：当配套产品价格下降时，原产品的需求会上升。一般来说，公司要实现自身战略利益，应该致力于降低配套产品的价格。理论上最低的可持续性价格是所谓的“商品价格”（commodity price），即很多竞争者提供相同产品时市场所达到的价格。所以，聪明的公司会致力于让其产品的配套产品变得商品化，唾手可得。如果能做到这一点，对该公司产品的需求就会上升，该公司就可以提高定价，获取更多的利益。

IBM 设计 PC 架构的时候，使用的是通用组件而非定制化组件，而且在（具有划时代意义的）IBM-PC 技术参考手册中，详细定义了组件之间的接口。为什么这么做？因为其他制造商可以加入进来。只要你的产品符合接口定义的标准，就能被 PC 机使用。IBM 的目标是让硬件配件市场商品化，因为它是 PC 市场的配套产品市场。他们的做法非常成功。在很短的时间内，就涌现出无数的硬件配件公司，提供了各种内存卡、硬盘、显卡、打印机等。配件价格的下降带来了 PC 需求的上升。

当 IBM 从微软那里获得 PC-DOS 系统的授权时，微软非常精明地没有提供独家授权。这让微软能够和其他公司签订同样的授权协议，如康柏公司等上百家 OEM 厂商。这些公司合法地使用 IBM 的官方文档克隆出自己的 IBM-PC 架构。微软的目标是让 PC 市场商品化。这个目标很快就实现了，PC 价格持续下降，机器性能持续提升，PC 厂商的利润空间被压榨到最低。低价导致了需求的增长，PC 需求的增长又意味着对于其配套产品 MS-DOS 的需求也增长了。在其他条件对等的前提下，一个产品的需求量越大，对公司收入的贡献就越大。这就是为什么比尔·盖茨富可敌国的原因。

今年，微软又试图故技重施：新游戏主机 XBox 采用了标准 PC 硬件作为配件，而不是定制化的配件。理论上，硬件商品的价格会逐年下降，所以 XBox

可以趁势而起^①。可惜这招似乎失效了。显然，PC 硬件商品经过多年的发展，价格已经被压低到趋近于商品价格，导致 XBox 的制造成本下降速度远不如微软预想得那么快。微软对 XBox 的另一个战略是使用自家开发的图像公共库 DirectX，它可以让开发者编写的代码在各种显卡上运行。这一战略的目的是压低显卡的价格，使其商品化，从而促进游戏销量的提升，因为游戏才是利润更为丰厚的领域。那么，显卡厂商为什么不能联合起来，反过来让游戏成为商品呢？这种想法实现起来有难度。如果某款游戏卖得很火，比如《光晕》，它实际上并不存在替代产品。这就像你到电影院观看《星球大战：克隆人的进攻》的时候，绝不会因为想省一点钱就决定观看一部伍迪·艾伦执导的电影。虽然两部片子都有可能很棒，但二者不能彼此替代。如果可以选择的话，你想当游戏发行商还是显卡厂商？

推动配套产品的商品化。

这条战略有着深远的影响，它甚至能够解释为什么许多商业公司乐于为开源事业添砖加瓦。接下来，我们就来看看相关的事例。

新闻标题：IBM斥资百万，助力开源软件

表象：卢·盖斯特纳阅读了 GNU 宣言之后，决定放弃资本主义信仰。

事实：IBM 之所以这样做，是因为它要转型为一家 IT 咨询公司。IT 咨询业是企业软件业的补充市场，所以，IBM 需要让企业软件变得商品化，要达到这一目的，最好的方法就是支持开源软件。事实就摆在眼前，这种对开源的战略投资已经让他们的咨询部门赚得盆满钵满。

新闻标题：网景公司开源浏览器

表象：他们想获得广大草根网民的支持，让那些在新西兰网吧里编程的人们也能免费贡献自己的代码。

^① 迪恩·高桥，《打开 Xbox：揭秘微软的娱乐革命计划》。

事实：此举的目的在于让浏览器商品化。

这其实是网景公司一贯的商业战略。他们的发布的第一篇新闻稿^①就旗帜鲜明地指出，浏览器是“免费软件”。网景公司通过免费发放浏览器，达到卖服务器赚钱的目的。浏览器和服务器的互为天生的配套产品，浏览器越廉价，对于服务器的市场需求量就越大。这个规律在1994年10月表现得尤为明显。（当时MCI公司不请自来，二话不说就为网景投了大笔资金，之后才幡然醒悟：原来浏览器本身也能赚钱^②。但是这类收入和他们的商业模式无关。）

网景公司之所以用开源软件的方式发布网景浏览器，还因为他们看到了一个降低开发浏览器成本的机会。开发成本的降低意味着能以更低的成本享受商品带来的益处。

之后，美国在线时代华纳公司收购了网景。服务器软件本来应该是廉价浏览器的受益者，但它卖得并不好，所以被裁撤了。那么，是什么支持美国在线时代华纳公司继续投资开源事业呢？

美国在线时代华纳是娱乐产业的巨头，娱乐产业则是所有娱乐分发平台的配套产品，其中包括网络浏览器。战略利益促使该巨头推动网络浏览器这一娱乐分发平台的商品化，他们巴不得所有公司都不能对浏览器收费。

有一个事实看似和我的解读不符，那就是微软免费提供Internet Explorer浏览器。推动浏览器商品化也符合微软的利益，这样他们就能多卖出一些桌面和服务器的操作系统。微软还更进一步，提供了一系列组件，让每个人都能开发出自己的浏览器。Neoplanet、美国在线和朱诺都用这些组件推出了自己的浏览器。既然IE这个老对手已经免费了，网景为什么还要继续让浏览器变得“更不值钱”呢？这是一种先发制人的手段。他们想阻止微软独霸浏览器市场，免费的浏览器市场也不可以。如果微软垄断了这个市场，它在理论上就有能力变相提高浏览器的价格，比如提高Windows的售价。

（我对于网景公司战略的解读还有一点站不住脚，因为很显然，在巴克戴尔掌舵的时代，网景根本没有明确的战略可言^③。更有可能的解释是，网景公

① 网景通讯公司新闻稿：《网景通讯公司通过互联网免费发放网络浏览器》，Netscape.com，1999年。

② 网景通讯公司新闻稿：《MCI选择网景通讯作为安全软件提供者，打造新的InternetMCI服务》，Netscape.com，1999年。

③ Charles Ferguson, *High Stakes, No Prisoners: A Winner's Tale of Greed and Glory in the Internet Wars* (Crown Business, 1999)。

司当时的高层对于技术一无所知，只能被动附和底层开发者出的馊主意。开发者都是黑客出身，并不懂经济学，他们的想法只符合自己的利益。不过我还是愿意相信开发者的本意是好的。)



新闻标题：全美达公司为Linux之父林纳斯捐款， 促其改进Linux

表象：全美达只是在吸引公众的注意。要不然它还能怎么上头条？

事实：全美达是一家 CPU 制造公司。CPU 的天然配套产品是操作系统。全美达想让操作系统成为商品。



新闻标题：Sun和惠普公司注资Ximian公司， 促其改进Gnome桌面系统

表象：Sun 和惠普公司支持自由软件，因为他们的公司文化偏爱集市胜过大教堂。

事实：Sun 和惠普都是硬件公司。他们要想在台式机市场上赚钱，需要使台式电脑的配套产品商品化，即窗口化软件系统。他们为什么不用投资 Ximian 的那笔钱，自己开发一个商业版的窗口化软件系统呢？其实他们早就尝试过了（Sun 有 NeWS，惠普有 New Wave），但这些做硬件起家的公司一般做软件的水平有限，况且他们想让窗口化软件成为一种廉价的商品，而不是一项自家的经营项目，所以他们付钱给 Ximian 的优秀软件工程师来做这件事。Sun 收购 StarOffice 然后把它开源的举动有着相同的动机：让软件商品化，然后在硬件上赚钱。



新闻标题：Sun推出Java编程语言， 新的“字节码”系统让开发者一次编写、到处运行 (Write Once, Run Anywhere, 简称WORA)

字节码的想法并不新奇，程序员一直致力于让代码运行在尽可能多的机器上（还是为了让配套产品商品化）。很多年以前，微软就有自己的 pcode 编译器和跨平台的窗口管理层，能让 Excel 运行在 Mac、Windows 和 OS/2 系统上，使用摩托罗拉、英特尔、阿尔法、MIPS 和 PowerPC 的芯片。Quark 公司有一个兼容层服务，能让苹果机的代码在 Windows 上运行。C 编程语言实际上是一种独立于硬件的汇编语言。总之，字节码对于软件开发者并不是什么新概念。

如果软件能在各种平台上运行，就能进一步把硬件商品化了。硬件价格下降拓展了市场需求，也推动了对于软件的需求（而且让客户有更充裕的资金，去购买更昂贵的软件）。

因此，Sun 公司对于 WORA 的热情是不合逻辑的，因为作为硬件公司，硬件商品化绝对不符合 Sun 的核心利益。

这实际上是 Sun 犯下的一个愚蠢错误！

Sun 公司是计算机产业里出了名的暴脾气。他们经常出于对微软的愤怒、恐惧和厌恶，在冲动之下做出不符合自身利益的商业决策。Sun 的两个主要战略是：(1) 通过支持和开发自由软件，推动软件的商品化（StarOffice、Linux、Apache、Gnome 等）；(2) 通过 Java 的字节码架构来实现 WORA，从而推动硬件的商品化。那么 Sun，当音乐停下之时，你打算在何处落座？如果在硬件和软件两个产业都失去了资产优势，到头来就只能接受商品价格，连位于墨西哥瓜达拉哈拉的廉价工厂都承担不起，更别说硅谷的豪华办公室了。

“但是乔尔！”杰瑞德指出，“Sun 公司和全美达一样，试图推动的是操作系统商品化，而不是你想的硬件。”也许吧，但 Java 字节码的跨平台特性在实质上促进了硬件的商品化，如果这是误伤的话，也是非常严重的误伤。

从这些例子中，你会察觉到很重要的一点：软件很容易使硬件商品化（只要

添加某种硬件抽象层就可以了，比如 Windows NT 的 HAL，代码量非常少），但是硬件使软件商品化则极为困难。软件通常具有不可替代性，StarOffice 的营销团队对此肯定有深刻的体会。虽然他们的产品价格为零，但客户放弃微软 Office 转而使用 StarOffice 所付出的成本可不是零。除非让转换过程的价格趋近于零，否则台式机办公软件不可能成为真正的商品。两种零售软件之间的差异再小，也会导致转换过程的困难。例如，虽然谋智浏览器几乎拥有我想要的所有功能（只要它别老是像和我玩打地鼠一样弹出广告），但它不符合我按下 Alt+D 聚焦地址栏的习惯，所以我就不用谋智（不服你去告我啊）。只要产品之间有一点点差异，它们就失去了成为商品的可能。而提到硬件，我曾经把 IBM 电脑的硬盘拔下来，插到戴尔电脑里，系统还是能正常启动，和旧电脑一模一样。硬件就是商品。

Creo^①的 CEO 阿莫斯·迈克尔森曾经告诉我，他们公司的每个员工都要参加一门他称为“经济学思维”的培训课程。好主意。只要对微观经济学中的理念有所涉猎，就可以对如今正在发生的根本性变化有一个大致的了解。

① Creo 公司已于 2005 年 5 月被柯达公司收购。——编者注



祸不单行的一周

2003 年 1 月 25 日，星期六



第一章

承载着我们公司 CVS 代码库（里面有公司全部的源代码）的 Linux 服务器坏了。这没什么，因为我们用 rsync 做了自动镜像备份，远程存储在另一个机房。为了恢复代码，需要几个小时来压缩并传送镜像数据。这时我们才发现忘记在 rsync 的选项中加上“不同步已经删除的文件”，所以备份数据并不全是镜像，还包含了所有已经删除的文件。这些文件需要手动删除。

这一切完成之后，我决定签出整个源代码树，并与本地的代码进行比较，看看代码是否完好。但我的笔记本电脑硬盘空间不足以完成这项操作，应该升级硬盘容量了。我买了一个 60GB 的笔记本硬盘和一根 PCMCIA/硬盘连接线，靠这根线来把旧硬盘上的数据复制到新硬盘上。经过六个小时，在完成进度为 50% 的时候，这个操作失败了，提示“请运行磁盘检查程序 scandisk”。磁盘检查又花了几个小时，然后我重新发起复制操作。又过了六个小时，在完成度为 50% 的时候，操作再次失败了。这回旧硬盘报废了，我悲愤欲绝。之后的几个小时里，我换了好几台电脑重试读取旧硬盘，结果发现它确实坏了。

这其实也没什么大不了的。我们还有每日备份（NetBackup Pro）。我把 60GB 的新硬盘插进笔记本，格式化，并安装了 Windows XP 专业版，然后让 NetBackup Pro 把机器恢复到之前的状态。如果能够成功，那么我损失的就只有一天的工作成果，反正我这天基本上也没干什么。这一天的电子邮件也丢

失了，所以如果这周你给我发了邮件但没有收到回复，请重发。

NetBackup Pro 要运行好几个小时。我决定下班回家，让它通宵工作。第二天早晨，系统坏了，启动画面都无法显示。我猜想一定是因为我试图在 XP 专业版系统上恢复一个 Win2K 镜像。所以我从头开始，新装了一个 Win2K（格式化硬盘：一小时；安装 Win2K：一小时；然后安装 NetBackup Pro 客户端）。然后我再次开始恢复系统。五个小时后恢复还没有结束，我决定回家睡觉。

第二天早晨，系统启动不太正常，还是蓝屏，但可以进入安全模式。鼓捣了半个小时以后，系统总算可以正常启动了。除了几个 Windows 加密（使用 EFS）的文件无法读取之外，一切都恢复到了灾难之前的状态。这几个加密文件的损坏与公钥以及证书丢失有关。我还没有找到这个问题的解决方法。如果你知道如何解决，请联系我，不胜感激。（1月26日：几个小时后，我终于解决了这个问题。）

经验教训

这种情况已经发生了不止一次：硬盘损坏引发一系列其他的问题，最终导致几天的工作付之东流。但请注意，我有非常完备的备份策略，每天都会把所有文件备份一次，存放在其他地点。在我印象中，由于硬盘损坏导致一连串不幸的事件，这已经是第三回了。结论是，仅有备份是不够的。从今以后，我要部署硬盘 RAID 镜像。如果硬盘再次损坏，我希望能在 15 分钟内更换一个新的硬盘，然后从之前中断的地方继续工作。所以我决定，从今以后，Fog Creek 软件公司的所有非笔记本电脑都要有 RAID 镜像。

第二章

你注意到我们的服务器死机了吗？上周五中午的时候，本地的一个 Verizon 基站失火了，所有电话线以及网络连接都断了。Verizon 在几个小时后恢复了电话线，但是 T1 线路还是有点问题。我们是从 Savvis 购买的 T1 服务，而 Savvis 是从 MCI（现在叫 WorldCom）购买的本地环路，而 WorldCom 的环路也不是自己的，他们的本地环路是从 Verizon 购买的。

所以从周五中午到周六午夜，迈克和我轮番上阵，每隔一小时就给 Savvis 打

个电话，询问进展如何。我们向 Savvis 施压的同时，他们也在向 WorldCom 施压，但是 WorldCom 却一直以为问题是某种 SQL Server DDOS 攻击造成的，所以他们想办法敷衍了 Savvis。但当时 Savvis 并没有把详细情况告诉我们，于是我们只能再一次向 Savvis 施压，他们又向 WorldCom 施压。如此往复三遍以后，WorldCom 终于妥协了，给 Verizon 打了个报修电话，Verizon 派出技术团队把机器修好了。我实在觉得整件事情就像是在表演提线木偶戏。这种 T1 线路断了一天的事件，本来只是由于出了一个很不起眼的技术故障，如果不是和这么多愚蠢的公司打交道的话，问题可能很快就被找到并修复了。

经验教训

当你从一家公司购买某种服务，并且那家公司也是通过外包来获得该服务的资质时，很难得到满意的客户服务。你从这种二道贩子手里不会得到有任何质量可言的客户服务。我虽然也不愿意和本地垄断性质的电信公司打交道，但比这更难以忍受的，是和另一家愚蠢、官僚主义、不得不求助于本地电信公司的公司打交道。谢天谢地，我们的新办公室将直接使用 Verizon DSL 提供的线路服务。

顺便说一句，如果戴尔公司能按时把我们的服务器送来，广大网民就不会注意到这次的服务中断了。我们本该在一个月前就用上 Peer 1 Network 公司提供的高冗余度安全主机托管服务。我刚才提到我发烧了吗？每当事情出差错的时候，我都会发烧。



第三章

Fog Creek 软件公司所在大楼的四楼暖气又停了，之前不知道发生过多少次了。暖气的热力是通过埋在墙里的热水管来输送的，而水管被冻成了冰疙瘩。暖气管怎么会结冰呢？哦，那是因为上周锅炉房熄火了，安装锅炉的人愚蠢透顶，可能连相关执业资格都没有，他在水平方向接了一截长达 25 英尺的烟囱管子（目前这截烟囱只供一个租客使用），造成通风不畅，锅炉熄火十多次。最后，热力公司的某个技术员提议，可以安装一个导风管，为烟囱强制排风。他们后来也这样做了，但为时已晚，暖气管已经结冰了。导致结冰的另一个原因是，纽约建筑行业的某个傻瓜没有把水管的隔热保温层做好，

但是如果锅炉不熄火，这个隐患也不会暴露。

经验教训

隐患重重的系统往往会呈现出一切正常的假象，直到与之有关联的系统出了故障，问题才会暴露出来。

患有过敏症和背部疼痛的人，可能几个月都很健康，但在一场花粉症的突袭下，会因为不断用力打喷嚏，继而诱发背部疼痛。类似的情况在系统管理工作中时有发生。一定要利用这些机会，一次性把所有暴露出的问题都解决掉。为所有 PC 都配好 RAID，做好备份；不要使用 EFS；硬盘一定要足够大，以免发生紧急情况时才发现需要升级扩容；仔细检查 rsync 的命令行参数。安好烟囱的导风管，把暖气管的保温层做扎实。把重要的服务器主机托管到安全的机房里，并且改用 Verizon 提供的办公 T1 线路服务。

微软是如何输掉API战争的

2004年6月13日，星期日

如今，流传着这么一种说法：“微软快完了。一旦 Linux 在台式机市场取得一席之地，加之网络应用替代桌面应用，这个强大的软件帝国就要垮台了。”

虽然这种说法有一丝真实的成分：Linux 确实是微软的劲敌，但是预言这家位于雷德蒙德市的公司死期将至，似乎为时过早。微软在银行里的现金储备还很充裕，每天仍在赚取可观的利润。瘦死的骆驼比马大。除非它在十年中持续性地犯各种错误，否则离破产那一天还远得很。况且，说不定他们还会在难以为继的时候大胆转型，变成一家冷饮公司什么的，所以别太急着唱衰微软。20世纪90年代，大家曾经以为 IBM 没有出路了：大型主机服务器已经过时了！罗伯特·X.克林格利曾经预言，大型主机服务器的时代将在2000年1月1日结束，所有用 COBOL 语言编写的应用程序都会失效，而且人们无法修复。因为他认为到时候程序的源代码已经丢失了很久，人们只能用客户端—服务器端架构来重写这些程序。

结果如何呢？大型主机服务器还活得好好的，2000年1月1日时，什么都没有发生，而 IBM 成功转型成为了一家大型科技咨询公司，还兼制作便宜的塑料电话。所以，片面地根据某些数据就推断微软不行了，是非常严重的夸大其词。

然而，另一个现象则很少有人提及，那就是微软已经失去了自己的战略高地：Windows API。

微软近乎垄断的实力主要来自 Windows 和 Office 两大系列产品，它们几乎是微软全部收入的来源，并且辐射带动了很多盈利不佳或刚刚赚钱的边缘产品线。而 Windows API 作为 Windows 和 Office 产品的基石，已经失去了对开发者的吸引力。虽然这只下金蛋的鹅还没有一命呜呼，但它确实已经身患绝症，

大限将至，只是还没有人察觉罢了。

话虽如此，但是上面一段有点耸人听闻、言过其实了，文风几乎和那些穿着体面的社论作家如出一辙，围绕微软的战略资产 Windows API 谈论不休。接下来，我想用几页的篇幅来说明这个论断的真正含义，并且举例论证。在看完我的解释之前请不要下结论。这章会相当长：首先说明 Windows API 是什么；然后需要证明，为什么它是微软最重要的战略资产；接着解释为什么微软失去了它，长远来看这会带来哪些影响。事先说明，本文讨论的是宏观趋势，所以难免会有部分词句显得夸张或者泛化。



开发者、开发者、开发者、开发者

什么是操作系统？它是负责管理计算机的各种资源，以便其他软件能正常运行的软件。人们其实并不关注操作系统本身，他们关注的是操作系统上能跑哪些软件，比如文字处理软件、即时通讯软件、电子邮件、应付账款系统，带有帕丽斯·希尔顿照片的网站，等等。操作系统本身并没什么用。人们购买它，主要是为了能在上面运行各种软件。因此，一个操作系统上能运行的有用程序越多，它就越有用。

由此得出的推论是，如果你是操作系统的销售商，最重要的就是让软件开发者愿意为你的系统开发软件。正是出于这个原因，史蒂夫·鲍尔默才会在演讲时在台上大叫“开发者、开发者、开发者、开发者”^①。微软非常看重开发者，他们之所以不免费提供 Windows 平台的开发者工具，唯一原因就是不想断了其他开发工具提供商的财路（也没剩几家了）。一个平台如果拥有多种开发工具可供选择，可能会对开发者有更大的吸引力，但是微软确实很想把开发工具送出去。通过他们的 Empower ISV[®]项目，开发者可以得到完整的五套 MSDN Universal（这个套装也被称作“除了微软模拟飞行之外的所有微软产品”），价格在 375 美元左右。.NET 的运行库免费附送了命令行编译器。C++ 编译器现在也免费了^③。他们竭尽全力地帮助那些为 .NET 平台贡献代码的程序员，同时避免挤垮 Borland 这类公司。

① 参见 www.ntk.net/ballmer/mirrors.html。

② 参见 members.microsoft.com/partner/competency/isvcomp/empower/default.aspx。

③ 参见 <http://www.microsoft.com/en-us/download/details.aspx?id=41151>。



为什么苹果和Sun公司的电脑卖不出去

这句话有失公允，苹果和 Sun 当然能卖出很多电脑，但是这两家公司在企业级台式电脑和家用电脑这两个利润最丰厚的市场斩获甚小。苹果公司的市场占有率一直徘徊在个位数，而 Sun 公司的电脑呢，只有 Sun 公司自己在用。（请注意我讲的是宏观趋势，当我说“根本没人”的时候，其实指的是“少于 1000 万人”这样的量级，所以请不要跟我抠字眼。）

为什么会这样？因为苹果和 Sun 公司的电脑不能运行 Windows 程序，即使勉强可以，也是在代价高昂的模拟器中运行，效果并不好。记住，人们买电脑是为了用上面运行的程序，而且 Windows 平台上优秀桌面软件的数量比 Mac 多出太多了，要当个 Mac 用户可真不容易。

这就是为什么 Windows API 是微软重要资产的原因。

（我知道，现在那 2.3% 使用苹果电脑的用户正在启动邮箱，酝酿着写一封怒火冲天的邮件，让我了解他们对于 Mac 的爱。我重申一次：这篇文章谈论的是宏观趋势，难免会有些泛化，所以不要浪费时间来纠正我了。我知道你们热爱 Mac。我知道你想要的软件 Mac 上都有。我爱你们，火爆的小辣椒，但是你们只占全世界用户的 2.3%，所以这篇文章针对的不是你们。）

“API”到底是什么？

如果你在写一个程序，比如文字处理器，想显示一个菜单或是写某个文件，就需要通过调用一类特殊的函数，让操作系统帮你完成。这些函数调用因系统而异，它们就叫作 API：操作系统（如 Windows）开放给程序开发者的接口。开发者通过调用这些接口来写出文字处理器、电子表格软件等。这些函数数量众多，调用参数和方式各不相同，开发者可以用它们来让操作系统做一些有用的事情（比如显示菜单、读写文件），或是做一些不常见的事情（比如将日期翻译成塞尔维亚语），甚至是做一些非常复杂的事情（比如在窗口中显示网页）。如果一个程序使用了 Windows 的 API 函数，那么它就不能在 Linux 上运行，因为 Linux 有另一套 API 函数。尽管不同操作系统的 API 函数有时候作用差不多，但任何差异都会导致 Windows

软件无法在 Linux 上运行。如果你想让 Windows 程序在 Linux 系统里运行,需要重新实现整个 Windows API^①,也就是里面上万个复杂的函数,工作量几乎等同于重新实现一个 Windows 系统。微软当初投入了数以千计的人年才把 Windows 做出来。所以,即使是很小的错误,或遗漏了一个函数,都会导致程序的崩溃。

微软内部的两大阵营

微软内部有两大互相抗衡的势力,接下来戏称为:雷蒙德·陈阵营和 MSDN 杂志阵营。自 1992 年起,雷蒙德·陈就在微软 Windows 团队做研发工作,他在自己的博客“旧闻新知”^②上写了很多 Windows 开发的技术秘辛,解释了 Windows 的某些设计决策。即使表面上看起来愚蠢的决定,背后也有非常正当的理由。阅读雷蒙德的博客,给我留下最深印象的,是 Windows 团队多年以来对于支持向后兼容所做出的艰苦卓绝的努力。

请站在客户的角度思考。你之前买了程序 X、Y 和 Z。升级到 Windows XP 后,发现电脑不定时地崩溃,而且程序 Z 根本无法工作。你肯定会和朋友说:“别升级到 Windows XP。它经常崩溃,而且不兼容程序 Z。”作为普通用户,你会对系统进行调试,发现是程序 X 造成了系统崩溃,而程序 Z 不能工作是因为它使用了不可靠的窗口消息吗?当然不会。你只会拿着 Windows XP 要求退款。(程序 X、Y、Z 是几个月前买的,30 天内退款的政策已经失效,所以只能把 Windows XP 退回去。)

让我第一次意识到这种问题存在的,是热门游戏《模拟城市》的一位开发者。他告诉我程序里有一个严重的 bug:会立刻使用刚释放的内存。这个严重的错误在 DOS 中没有被发现,但在 Windows 中暴露无遗,因为 Windows 的内存一旦被释放,就可能立即被另一个运行中的程序使用。当年 Windows 团队的测试人员拿来了很多流行的应用程序来测试其在 Windows 95 上的可用性,发现《模拟城市》总是崩溃。Windows 团队接到 bug 报告后反汇编了《模拟城市》,在调试器里顺藤摸瓜找到了这个 bug,然后在 Windows 中特别加入

① 请参考 Wine,一个开源的 Windows API 项目: www.winehq.com/。

② 雷蒙德·陈,“旧闻新知”。参见 <http://blogs.msdn.com/b/oldnewthing/>。

了如下处理逻辑：如果检测到当前运行程序为《模拟城市》，就让内存分配器以一种特殊模式运行，让程序可以使用被释放的内存。

这件事并不是个例。Windows 有一个庞大的测试团队，最主要的职责就是保证每个人都能安全地升级操作系统，之前安装的软件（即使有未发现的错误，使用了不可靠的 API 函数或是依赖某种在前一个 Windows 版本中有问题、在新版本中得到修复的特性）都能继续正常工作。事实上，浏览注册表的 AppCompatibility 区域，就能看到由 Windows 进行过兼容性处理的全部程序。兼容性处理包括模拟旧版系统中的 bug 或奇怪行为，以便这些程序能正常工作。雷蒙德·陈写道：“当人们指责微软在系统升级时故意破坏旧版程序的时候，我非常愤怒。如果有任何程序在 Windows 95 中不能正常工作，我都把它看作我个人的失败。我曾经通宵达旦地为第三方程序修复 bug，以便它们能在 Windows 95 上继续运行。”

很多开发者和工程师都不认同这种工作方式：如果程序本身有 bug 或是依赖某种不可靠的行为，就应该在系统升级后无法运行。早年开发苹果机操作系统的开发者就持这种意见（但是后来苹果机操作系统的向后兼容性有所改善），所以绝大多数在早期苹果机上出现的程序，如今都无法运行。如果当年推出这个程序的公司倒闭了（大部分公司都倒闭了），那用户只能自求多福了。

与之形成对比的是，我在 1983 年为初代 IBM-PC 写的 DOS 程序，今天还能在最新的 Windows 上完美运行，这要归功于微软内部雷蒙德·陈阵营的不懈努力。我明白不仅仅是雷蒙德一个人，这还是 Windows API 核心团队的集体作风。是雷蒙德通过他精彩的个人网站“旧闻新知”让这段历史流传于世，所以我用他的名字来指代这个阵营。

我们再来看另一个阵营，我称之为 MSDN 杂志阵营。MSDN 这本面向开发者的杂志上充斥着各种技术文章，介绍如何把各种微软产品组合运用到自己的程序中，很多方式非常不靠谱，风险很大。MSDN 杂志的目标就是鼓励大家多用更新颖、更复杂的外部技术，比如 COM+、MSMQ、MSDE、Microsoft Office、Internet Explorer 及其组件、MSXML、DirectX（请使用最新版）、Windows Media Player 和 Sharepoint（Sharepoint！不知道谁会用）——简直是外部依赖代码库的盛装检阅。如果你在开发中使用了其中的任何一个，一旦将产品卖给用户以后依赖库出现问题，就非常不好解决了。这种情形的术语叫作 DLL 地狱。为什么同一款软件，在这台机器上好用，换一台机器就不行了呢？

雷蒙德·陈阵营讨好开发者的方式，就是努力让他们写过的代码能在尽可能

多的平台上运行（当然这些平台都以 Windows 开头）。而 MSDN 杂志阵营讨好看开发者的方式，则是提供功能强大的代码库，开发者如果不介意承担复杂的部署和安装带来的风险，并且愿意攀登陡峭的学习曲线，就可以借助这些代码库的力量提升开发效率。雷蒙德·陈阵营强调稳定压倒一切。他们似乎在说，请不要制造更多麻烦，先让写过的代码都能用再说吧。而 MSDN 杂志阵营则不断拿出各种强大的科技武器，让人眼花缭乱、目不暇接。

这就是为什么我要写这篇文章的原因。

微软丧失了对向后兼容性的信仰

在微软内部，MSDN 杂志阵营已经赢得了这场战役。他们首次大捷是在研发 Visual Basic 的时候，.NET 平台的 VB 不兼容之前的 VB 6.0 代码。这是有史以来第一次，用户购买微软产品的升级包之后发现原来的数据（也就是原来在 VB6 里写的代码）不能无缝、静默地导入到新版本的软件中。这是有史以来第一次，微软的升级没有顾及客户使用旧版产品的劳动成果。虽然 VB6 开发者们当时群情激愤，但这批开发者本来就在日渐消亡：即使没有这次事件，他们也打算转行去做网络应用程序开发了。所以，这次事件的长远影响被掩盖了。

在取得了第一次重大胜利后，MSDN 杂志阵营捷报频传。很快，大家就默许改变既定规范的行为了。IIS 6.0 的线程模型与之前版本不同，让某些旧版程序无法运行。用户在 Windows Server 2003 上运行我们公司的 FogBUGZ 时遇到了问题，当时我大吃一惊。然后是 .NET 1.1 的问世，它并不完全兼容之前的 1.0。受到这种“革新”风气的影响，Windows 操作系统团队终于也决定，不再向旧版 Windows API 增加功能特性，而是要完全重写一套 API 来替代它。开发者被告知，不要再学习 Win32 API，应该做好准备、拥抱下一代的 Windows API：WinFX^①。它基于 .NET 的托管代码、XAML、Avalon 等新技术，与 Win32 API 截然不同。我承认，这听起来比 Win32 高级多了。但这并不是升级，而是与过去的决裂。

外部开发者本来就觉得 Windows 开发过于复杂，这下他们更会一致远离微软开发平台，投入如火如荼的网络开发领域。在互联网爆发早期缔造了雅虎商

① Mark Driver，“微软推出 WinFX，促进 .NET 平台的使用”，高德纳咨询公司，发表于 2003 年 11 月 3 日。参见 www.gartner.com/DisplayDocument?doc_cd=118261。

店的保罗·格雷厄姆对此曾有过精辟的论述：“如今，初创公司有更充分的理由去基于网络来开发应用，因为写桌面应用变得越来越没意思。如果你现在还打算写桌面软件，就要受到微软的胁迫，调用他们的 API，绕过他们在操作系统里挖的陷阱。即使你写的程序大受欢迎，到头来你还是感觉自己只是在为微软进行市场调研。”^①

微软的规模足够庞大，拥有数量众多的开发者，而且一直热衷于从软件升级中获取收益，所以他们完全可以突然拍板决定重写每一个软件，甚至可以重写两次。但是，当年那个保守的、以雷蒙德·陈为代表的微软，在实现新的图形化界面系统 Avalon 时，可能会依托于一系列和应用软件捆绑发布的 DLL 方式，这样所有版本的 Windows 就都能使用 Avalon 了。但是微软需要一个让人们升级新一代 Longhorn 操作系统的理由，这次升级对于用户体验的改变之大，几乎可以和 Windows 取代 DOS 时的改变相提并论。问题是，Longhorn 对于 Windows XP 的改进并不像 Windows 对于 DOS 的改进那么大，人们会觉得为了这个系统而购买新电脑和新软件，理由不够充分。但是微软当然想促成这件事，所以努力增加新系统的吸引力，但到目前为止，我对新系统的优越性还是抱有怀疑。很多人都打赌微软犯了错误。例如，WinFS 在宣传中被描述成一种让搜索更好用的手段，通过关系型数据库来管理文件系统，但微软忽略了一个事实：真正让搜索更好用的方法，就是直接改进搜索^②。与其费力给所有文件添加元数据以便用 SQL 查询文件，还不如务实一点，使用 1973 年就已经十分常见的全文索引以及相关技术，快速地在硬盘上找到用户输入的字符串。

自动挡的胜利

有必要澄清一下，我认为 .NET 是一个很棒的开发环境，Avalon 和 XAML 双剑合璧，比古老的 Windows 用户界面编程方式高级多了。.NET 平台最大的优势就是拥有自动内存管理系统。20 世纪 90 年代，软件行业的思潮是，在面向过程和面向对象编程范式的大战中，面向对象的编程范式将取得胜利，

① 保罗·格雷厄姆的文章“另一条前进的道路”，发表于 2001 年 9 月，详见 www.paulgraham.com/road.html。

② 约翰·尤德尔，“关于 Longhorn 的问题，第一部分：WinFS”，来自约翰·尤德尔的博客 iDiscuss, InfoWorld, 2004 年 6 月 2 日。参见 <http://jonudell.net/udell/2004-06-02-questions-about-longhorn-part-1-winfs.html>。

因为它大大提高了程序员的开发效率。我当时也那么想，如今还是有很多人同意这个观点。但是我们错了。面向对象编程虽然方便又漂亮，但是并没有显著提高开发效率。我认为要想真正提高开发效率，有必要选择一门具有自动内存管理功能的语言；实现方式可以是引用计数或是垃圾回收，语言可以是 Java、Lisp、Visual Basic（即使 1.0 也可以）、Smalltalk 等许多脚本语言。如果一门编程语言能让你获取一块内存而不用考虑如何释放它，那么这种编程语言就具有内存管理功能。使用这种语言，开发效率就会比使用那些需要程序员主动管理内存的语言要高。很多编程语言的倡导者都会自吹自擂，说使用该语言的开发效率有多么高，其实这种高效率大多归功于自动内存管理，即使连他们自己都没有意识到这一点。

为什么要有内存管理？

为什么自动内存管理能提升开发效率？原因有三：(1) 你可以随意写出 $f(g(x))$ 这样的嵌套函数，不用担心如何释放 g 的返回值内存，可以随心所欲地在函数中处理并返回复杂的数据类型，能在更高的抽象层面上进行思考和编码；(2) 你不需要写释放内存的代码，也不用追踪内存泄漏问题；(3) 你不再需要让函数在同一个代码位置返回，因为不用考虑内存是否得到完全清理的问题。

这么说可能会引起赛车爱好者的反对，但根据我的经验，日常驾驶中的绝大多数情况下自动挡汽车都比手动挡汽车方便。软件开发也是这样：在绝大多数情况下，自动内存管理都比手动内存管理方便，更有利于开发效率的提升。

如果你以前做过 Windows 开发，就会知道微软提供了两种内存管理方式：直接写 C 代码调用 Windows API 来自己管理内存，以及使用 Visual Basic 的自动内存管理功能。这两种方式我都很熟悉，从我 13 年来的开发经验来看，使用 Visual Basic 的开发效率要远远高于使用 C 的开发效率。我经常在这两种语言中实现相同的功能，使用 C++ 和 Windows API 写一次，再用 Visual Basic 写一次，而 C++ 的工作量通常是 VB 的三到四倍。为什么？就是因为内存管理。举个最简单的例子：返回一个字符串。那么，对于由谁负责为字符串分配内存，以及如何协商需要分配多少内存这种问题，你可以查查看，已经有很多讨论。通常需要调用两次函数，第一次传参告知它你为返回值分配了大小为 0 的内存，然后它会报错“内存分配不足”，并且好心地告诉你至少需要分配多少内存。如果返回值的类型是字符串列表或是长度可变的数据结构，那么代码将更加复杂。总而言之，即使是像打开文件，写入一个字符串

再关闭的简单功能，使用原始的 Windows API 也需要写长达一页的代码。而在 Visual Basic 中，同样的操作只需要写三行代码。

这就是两个编程世界之间的差异。几乎所有人都认同，托管代码的世界比非托管代码的世界更具优势。Visual Basic 曾经是（可能仍然是）世界上卖得最好的编程语言，Windows 平台的开发者对它的喜爱要远远超过 C 和 C++。虽然名字中带“Basic”（初学者通用符号指令码）这种字眼，让程序员的中坚力量对它嗤之以鼻，但无法否认的是，VB 是一门相当现代化的语言，有很多面向对象编程的功能特性，而且语言中的糟粕部分很少（行号和像呼啦圈一样的 LET 语句）。VB 的另一个问题是部署时需要提供 VB 运行时，这对于通过调制解调器传输的共享软件来说是个大问题。更糟的是，别的程序员能看到你的程序是使用 Visual Basic 开发的（真可耻）。



统一运行时，称霸全平台

然后 .NET 应运而生。它是一个伟大的项目，试图统一 Windows 平台的所有开发环境，从而一举结束之前混乱的局面。毫无疑问，.NET 具有自动管理内存的功能特性。除了保留 Visual Basic，.NET 还支持一门新的语言。它虽然实质上和 VB 相同，却有类似 C 语言的语法，例如使用花括号和分号；最棒的是，这个 Visual Basic/C 的混血儿叫作 Visual C#，所以你可以扬眉吐气地告诉大家，你再也不是一个“Basic”程序员了。程序员再也不用和丑陋难用的 Windows 函数直接打交道，不用理会各种向后兼容性 bug 和难以琢磨的字符串返回语法。所有这些麻烦都被统一的、清晰的面向对象接口所取代，而且只有一种类型的字符串。统一运行时，称霸全平台。这非常完美。作为一种开发环境，.NET 也非常好，它能自动管理内存，有一个丰富、完备、相容的操作系统接口，以及一整套丰富、完备、优雅的基本操作对象库。

到目前为止，使用 .NET 的人还不多，然而少数用户对它却爱不释手。但是，作为一个全新的编程环境，.NET 具有不止一种两种，而是三种（也许是四种？）编程语言，这并不有助于统一 Visual Basic 和 Windows API 使用者的分歧。就好像在两个孩子吵架的时候，用高于他们的分贝大喊：“闭嘴！”这种方法只在电视里管用，现实生活中，这只会让二虎相争转变为三国大战。

（顺便说一句，对于博客聚合服务的使用者来说，晦涩难懂但又纷争不断的

聚合页面格式也面临着这种局面。RSS 日益面临版本分化，功能规格不明确以及种种派系斗争问题的困扰，试图通过创造一种名为 Atom 的数据格式来一次性解决所有分歧，但是完全行不通。通过第三种力量来解决两种力量的对抗，只会导致三种力量互相对抗，什么也统一不了，什么也解决不了。）

因此，.NET 并没有带来开发工具的统一和简化，反而导致了六方相争的混乱局面，每个人都不知道该采取哪种开发策略，是否值得把之前平台上的代码移植到 .NET 平台上。虽然微软市场部门的宣传语是清晰一致的（“相信我们，就用 .NET 吧！”），但大部分顾客仍然在使用 C、C++、Visual Basic 6.0、传统的 ASP 以及其他公司提供的其他开发工具。而那些使用 .NET 的人大多在用 ASP.NET 开发网络应用，这种应用部署在 Windows 服务器上，运行的时候不需要 Windows 客户端。这很关键，之后讲到网络开发的时候会强调这一点。



等等，又有新的来了！

由于微软拥有的开发者太多了，所以重新研发整个 Windows API 还嫌不够，要再来一次。他们在去年的个人电脑开发大会（PDC）上预告了下一代操作系统的重大升级版本，代号为 Longhorn。它拥有诸多特性，其中包括一套全新的用户界面 API，代号 Avalon。Avalon 再起炉灶，充分利用了现代计算机的显卡性能，可以进行实时 3D 渲染。如果你正在使用微软“官方”出品的最新图形化界面开发环境 WinForms，那么两年后就要重头开始，把今天写的软件移植到 Longhorn 和 Avalon 平台上。WinForms 就这样“出师未捷身先死”。但愿你还没有在上面开发太多的代码。约翰·尤德尔（John Udell）发现微软发布了一个名为“如何在 Windows Forms 和 Avalon 之间进行选择”的幻灯片，于是他就问道：“为什么开发者需要在 Windows Forms 和 Avalon 之间进行选择呢？”这是个好问题，但他并没有给出一个很好的答案^①。

目前的状况是，Windows 开发者可以用 Windows API、VB、.NET 平台上的好几种大同小异的语言进行开发，但是请不要在这些语言上花太多的时间，因为微软在做 Avalon。Avalon 只支持最新版的微软操作系统，而这个新系统

^①“关于 Longhorn 的问题，第 3 部分：Avalon 的开拓性使命”，来自约翰·尤德尔的 InfoWorld 博客，发表于 2004 年 6 月 2 日。参见 <http://jonudell.net/udell/2004-06-09-questions-about-longhorn-part-3-avalons-enterprise-mission.html>。

在未来很长一段时间内都不会得到普及。我个人还没有投入足够的时间去深入研究.NET, Fog Creek 软件也还没有把两个主要产品从 ASP 和 Visual Basic 6.0 迁移到.NET, 因为对于我们来说, 这样做的投资回报率是零。我认为这只是干扰射击^①: 微软非常乐意看到我们停止为自己的 bug 跟踪软件和内容管理软件开发新的功能特性, 浪费几个月的时间去把代码移植到新的编程环境中, 而这种做法对于用户没有一点好处, 更不会对我们产品的销量产生任何提升作用, 所以这几个月的工作是徒劳无功的。这正中微软的计策, 因为他们也有自己的内容管理软件和 bug 跟踪软件。他们最愿意看到的事情就是我浪费宝贵的开发时间去追赶当下的潮流, 最好等到 Avalon 出来, 再浪费一到两年, 而在此期间, 他们则紧锣密鼓地为自己的软件增加新的功能特性。实在高明得很。

全职程序员根本没有闲工夫去试用雷德蒙德出品的所有新开发工具, 因为微软做开发工具的程序实在太多了!

已经不是 20 世纪 90 年代了

20 世纪 80 年代到 90 年代, 微软伴随着个人电脑市场的急剧增长而迅速崛起。当时个人电脑的年销量比之前的总装机量还要多。这意味着, 如果你做了一个只能在新电脑上运行的产品, 那么在一到两年内, 即使没有人放弃正在使用的产品而改用你的产品, 你还是能取得行业龙头的地位。这是 Word 和 Excel 能如此彻底地取代 WordPerfect 和 Lotus 的众多原因之一: 微软只要等待下一波硬件升级大潮的到来, 然后把 Windows、Word 和 Excel 卖给进行台式机配置升级的大公司(有些公司甚至是第一次购买台式机)。在很多方面, 微软并不需要学习如何说服使用旧产品的广大用户改用升级版的产品。因为人们得到新电脑的时候, 会很乐意顺便装上微软出品的所有最新版软件。但是用户主动升级软件的可能性就小得多了。这在 PC 产业如火如荼发展的时代完全不成问题, 但是现在世界上的个人电脑数量已经趋近饱和, 大部分电脑的性能都还不错, 于是微软突然感觉到, 推动最新版产品走向市场的时间显著变长了。当他们试图让 Windows 98 “寿终正寝”的时候, 才发现还有太多的用户在使用这个旧版系统, 于是被迫承诺多提供几年的支持。

^① 参见第 15 章。

不幸的是，像.NET、Longhorn 和 Avalon 这样试图改变人们开发和使用软件习惯的大胆革新性产品，在实际推广中遇到了很大阻力，因为很多人在 1998 年买的旧机器还很好用。即使 Longhorn 系统能于 2006 年如期发布（我个人觉得这很难做到），要取得足够数量的用户，成为一个有价值的开发平台，至少还要等好几年的时间。开发者、开发者、开发者……开发者们肯定不会听从微软多重人格般紊乱的开发建议。



属于网络应用开发的时代

写了这么多，终于讲到了网络开发。当今的每个开发者在规划新的软件应用时，都面临两个选择：可以为网络开发应用，也可以为 PC 开发“富客户端”应用。两种选择的优缺点非常明显：网络应用更容易部署，而富客户端的响应速度更快，用户界面可以做得更绚丽。

网络应用之所以更容易部署，是因为不需要用户进行任何安装操作。用户只需要在浏览器地址栏键入 URL，就完成了软件的“安装”。今天，我通过在浏览器里按下 Alt+D，键入 gmail，并按下 Ctrl+Enter，就“安装”了谷歌最新推出的电子邮件应用。基于网络的应用，其版本兼容性问题以及与其他产品的兼容性问题相对较少。每一个用户使用的版本都完全相同，不需要维护一系列历史版本。你可以任意选择编程开发环境，只要能让代码在你自己的服务器上跑起来就行。你的应用可以自然而然地运行在几乎所有性能不是太差的电脑上。与此同时，用户数据可以通过几乎所有电脑进行访问。而这要付出的代价是，这种应用的用户界面不会特别顺畅。以下是网络应用不能很好处理的几个方面：

- (1) 快速绘图；
- (2) 具有实时拼写检查工具，能在错误处绘出红色波浪下划线；
- (3) 警告用户如果点击浏览器关闭按钮，会丢失所有未保存的工作；
- (4) 在不需要访问服务器并获得响应的前提下，根据用户的操作更新一小部分用户界面；
- (5) 做一套不需要鼠标就能操作的键盘快捷键交互接口；
- (6) 支持离线工作。

这些方面其实也不是多么难以克服的问题，有些问题可能很快就会被聪明的

JavaScript 开发者所解决。最近的两个电子邮件应用 Gmail^①和 Oddpost^②，处理方式就相当漂亮。用户似乎不太在意界面上的小毛病或者网络造成的界面延迟。我认识的大部分普通人都欣然接受了基于网页的电子邮件客户端，无论我怎么摆事实讲道理，他们都不愿意使用体验更加丰富的富客户端。

目前，网络应用的用户界面效果已经达到本地富客户端应用效果的 80%。即使不借助新的浏览器，通过努力也可以让这个数字达到 95%。这对于绝大多数人来说都足够好了，而且对于开发者来说也是好事：最近比较出名的新应用，几乎都是基于网络的。

这意味着，微软的 API 突然变得可有可无了。基于网络的应用不需要 Windows 运行环境。

微软对这种局面早有心理准备。他们当然有准备，当局势日趋明朗的时候，还来了个急刹车，紧急叫停了 HTA^③和 DHTML 等颇具前途的新技术。Internet Explorer 团队几乎销声匿迹，几年都没有他们的消息。微软绝对不会让 DHTML 技术比现在更好，因为网络应用已经威胁到了他们的核心业务：富客户端。如今微软内部流传着一句话：“微软把公司命运押在了富客户端上。”从 Longhorn 的宣传幻灯片上就可看出端倪。Avalon 团队的乔·贝达说：“Avalon 乃至整个 Longhorn 就是微软下的一笔赌注，我们相信桌面软件的力量，只有在本地机器上才能实现真正强大的体验，即使在未来也是如此。我们把研发力量集中在桌面软件上，因为这个领域仍然大有可为，希望我们的努力能重新唤起大家对桌面软件的激情……”^④

但问题是，已经太迟了。



我个人对此感到惋惜

实际上，我个人对此感到有些惋惜。网络技术本身虽然很诱人，但我个人认

① 参见 gmail.google.com/。

② Oddpost 公司已于 2004 年 7 月被雅虎公司收购，提供新版雅虎邮箱服务。——编者注

③ 《HTML 应用（HTA）简介》，微软公司。参见 [http://msdn.microsoft.com/en-us/library/bb250409\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/bb250409(v=vs.85).aspx)。

④ 乔·贝达，“Avalon 能够战胜 Web 吗”，Channel 9，2004 年 4 月 7 日。参见 <http://channel9.msdn.com/ShowPost.aspx?PostID=948>

为目前网络应用的整体水平有待提高，界面延迟现象还很严重，设计思想也不够统一，这对于人们的日常软件使用体验是极大的挫伤。我还是喜欢使用本地富客户端应用。如果不得不使用网络版的 Visual Studio、CityDesk、Outlook、Corel PHOTO-PAINT、QuickBooks 等日常软件，我可能会发疯。然而，开发者们就是想实现各种日常软件的网络化。没有人（重申一句，这里是指“少于 1000 万人”）愿意再用 Windows API 开发软件了。风险投资人也不看好 Windows 应用，因为他们害怕和微软进行行业竞争，而且大部分用户对于糟糕网络用户界面的容忍度好像比我要高。

最重要的是，我注意到（并且和一位做招聘工作的朋友确认过）在纽约，懂得 C++ 和 COM 编程的 Windows API 程序员的年薪为 13 万美元，而使用托管代码语言（Java、PHP、Perl 以及 ASP.NET）的网络应用开发者的年薪在 8 万美元左右。差距还是很大的。我把这种情况讲给微软咨询服务团队的朋友听时，他们承认，新一代程序员已经对学习微软的技术失去了兴趣。COM 程序员的年薪之所以高达 13 万美元，是因为最近 8 年来几乎没有人愿意去学 COM 编程，还在使用 COM 技术的公司只能聘用资深的程序员，而这些人通常早已进入管理岗位，很难说服他们回过头来写代码，整天和（天啊）编组、名字对象、单元线程、聚合、tearoff 接口等老古董打交道。这些概念大概只有微软老牌工程师唐·鲍克斯（Don Box）才能真正弄懂，而他本人现在大概也不愿意再看一眼这些代码了。

不得不说，已经有大批开发者转战网络开发平台，并且一去不返。目前大部分 .NET 开发者都是 ASP.NET 开发者，为微软的网络服务器进行开发。ASP.NET 真的很棒，从我将近 10 年的网络开发经验来看，它非常先进，是次世代的开发平台。它还是一种服务器技术，用它开发的软件应用可以服务于使用各种桌面系统的客户。此外，借助 Mono^①，ASP.NET 在 Linux 下也能跑得很顺畅。

这一切对于微软都不是什么好消息，因为目前它营收利润的很大一部分都依赖于 Windows API 的力量。而新的 API 就是 HTML，应用开发市场上崭露头角的赢家，会是那些有能力让 HTML 技术大放异彩的人。

① 参见 <http://www.mono-project.com/>。

第四部分

关于.NET的闲言碎语

43 走火入魔的微软

44 我们向.NET迁移的策略

45 能给我一个链接器吗?

43 走火入魔的微软

过去,微软向程序员出售开发工具。我记得 Microsoft C 有一个非常棒的广告,大概是在推广 3.0 版本的时候,并且发表了一份四页的白皮书,详细描述了 this 编译器所采用的新优化技术。

不知从何时起,微软开发工具产品的营销人员意识到了,IT 公司的大部分产品购置预算是由行政管理人员控制,而不是由程序员控制的。这些管理者大概喜欢听到“在您的 .NET 产品的整个开发周期中,管控效率和可扩展性,以降低风险,减少您拥有产品所需要付出的总体成本”之类的夸夸其谈(没错,这段话就是从 Visual Studio 主页上摘抄的^①。)那些只针对开发者来定向营销的时代已经一去不复返了。我第一次注意到是在 2000 年 7 月,微软大张旗鼓地对 .NET 进行市场宣传,伴随着很多出于公共关系考虑的媒体曝光活动,而这一切都发生在 .NET 真正发售的三年以前,很明显是在混淆视听,用 FUD 战术^②阻碍 Java 的成长。

2000 年 7 月 22 日,星期六

微软最近的产品名为 Microsoft .NET,《财富》杂志之流把它称为一次重大“革命”,实际上完全就是雾件。我认为这次事件表明,雷德蒙德市的那帮人出了非常严重的问题。

所谓雾件,就是对各种功能特性和产品作出空口许诺,但实际上根本拿不出可以卖的东西。.NET 比雾件还要差,一贯高傲的微软这一次甚至连“雾”都没有提供。仔细看看 .NET 的白皮书^③,你会发现里面说得天花乱坠,但实

① 微软 Visual Studio 开发者中心, <http://msdn.microsoft.com/vstudio> (2004 年 5 月 25 日访问并摘抄)。

② FUD (Fear, Uncertainty, Doubt), 括号中的三个词的意思分别为惧、惑、疑。具体参见 <http://baike.baidu.com/view/815678.htm>。——译者注

③ “Microsoft .NET: Realizing the Next Generation Internet”, 微软, 2000 年 6 月 22 日。这份文档在微软的官网上已经找不到了。

上只是 FUD 战术，一点实质内容都没有。无论如何努力，也无法从这本白皮书中挖掘出任何有参考价值的信息。就像是一把沙子抓得越紧，它流失得就越快。

我不是说.NET 毫无新意，而是说从这本白皮书中，什么都看不出来。不信请看下面这段话：

人们普遍认为网络将逐渐演化。要想让这种演化真正惠及开发者、企业和消费者，需要一个全新的梦想。微软把提供这种梦想作为自己的使命，并且努力研发所需的科技，让梦想变为现实。

还有这段：

微软的.NET 梦想，意味着赋予消费者、企业、软件开发者乃至整个产业更强大的力量；意味着释放互联网的全部潜能；还意味着让网络变得更加符合人们的意愿。

这到底是什么意思？我翻遍了这本白皮书，完全找不出能真正落实到某个软件产品中的想法。微软提供的不是一份功能特性清单，而是一份含义模糊的“好处”清单，比如：

网络站点将提供更加灵活的服务，让大家能够进行交互、交换和利用彼此的数据。

这竟然被列为.NET 架构激动人心的“功能特性”之一。这个表述太宽泛、空虚和抽象了，完全没有任何实际意义，而大家对此好像一点也不在意。再来看看这个：

微软.NET 让寻找与之交互的服务和客户成为可能。

太棒了！从 Altavista 上线之日起已经过了 5 年，从拉里·佩奇和谢尔盖·布林创办在本质上更加优秀的搜索引擎公司（谷歌）之日起也已经过去了两年，微软竟然还在假装互联网处于无法搜索资源的蒙昧时代，并且将由他们为我们提供这个难题的解决办法。这份白皮书通篇都是这种论调。

这表明了两件事。首先，微软有很多伟大的思想家。伟大的思想家在思考问题的时候，能透过现象看到内在的联系和模式。思想家看到人们互相发送文字处理软件生成的文件，看到人们互相发送报表文件，然后归纳出这些现象都属于一个具有普遍性的动作特例：传送文件。在这第一层抽象的基础上，他们又作了一层抽象：人们发送文件，浏览器也发送网页请求。这些动作都

是发送操作，所以我们聪明的思想家引入了一个更高级、更普遍的崭新抽象概念，称为消息传递。可惜这个概念经过了太多层次的抽象，变得玄之又玄，没人听得懂思想家在说什么了。

如果把事物上升到过高的抽象层次，人类思维就无法理解了，就好像离地面太远，大脑会缺氧一样。片面追求思维抽象层次的思想家们，有时候会创造出古怪的、令人困惑的宏观宇宙图景，这些理论虽然很完善，但其实言之无物，根本没有任何实际意义。

以下文字正是这种情况的集中体现：

下一代 Windows 桌面软件平台 Windows .NET 致力于促进生产力、创造力、系统管理、休闲娱乐以及生活的其他方方面面，其设计旨在真正让用户掌控自己的数字生活。

这些词句抽象到根本没法对其进行批判。谁不想让操作系统促进生产力？多好的功能特性！我也想要一个具有“促进生产力”功能的新系统！问题是，微软到底要怎么兑现这个承诺？在最近 20 年的软件历史上，对于生产力的促进作用一直是温和渐进的。微软难道发现了一种提升操作系统生产力的全新化合物？我觉得不太可能。我觉得他们在虚张声势，这是典型的 FUD 战术和雾件。

可怕的是，他们是认真的

我了解微软——我在那里工作过三年。我知道是什么样的人写出了这个文档。比尔·盖茨很显然在其中扮演了关键性的角色，他卸下了 CEO 的重担，终于有时间写这种东西了。我并不认为微软发布这种文档的原因是他们需要雾件。他们是绝顶聪明的人。

我认为，他们真的以为自己在创造未来，并且对于如何行动胸有成竹。对于自己旗下的所有产品（从 Hotmail 到 SQL Server），微软都试图赋予他们一个更宏大的新使命。但问题是，还没有人能产出某种革命性的发明。这一点也不奇怪，不是微软太笨（他们很聪明），而是革命性的发明太罕见，任何一家公司里的聪明人都是有限的。放眼世界，也只有一个人发明了 Napster，他不是微软的员工。微软特别想通过自己的努力来制造下一个革命性的产品，

但即使是在互联网的“寒武纪生命大爆发”时代，每年诞生的革命性想法也是屈指可数的。这些想法诞生于比尔·盖茨以及他在雷德蒙德市的幕僚们脑中的概率，更是小到可以忽略不计。如果这个想法来自于某个在微软基层工作的聪明程序员（本职工作是为 Windows NT 开发显卡驱动），那么它被听到并且传播开来的概率，更是微乎其微。

从白皮书里能够得到的唯一具体信息，就是这个软件将通过互联网订阅服务的形式发售，而不是通过光盘来安装。

对于客户来说，通过互联网付费下载而不是购买光盘来获得文字处理软件，是有那么一点好处，但差别也不是很大。这并不能解决客户的任何实际问题。通过互联网来获得 bug 修复补丁？很好，但是这早就可以实现了。这七年来，我一直都在为微软产品下载各种补丁，现在这个过程已经相当自动化了。通过互联网来获得软件的新版本？如果新版本的唯一改进就是让获取新版本更加容易，那这个新版本有什么意义？！Word 在最近的三个版本中几乎都没有增加任何实质性的功能特性，唯一能体会到的变化，就是用一种奇怪的方式让摆放图片变得更“容易”，但是我还是不能把图片放在我想放的位置。

事实上，微软早在 1991 年就注意到，他们的收入越来越依赖于软件升级，而让大家升级软件是很困难的事。最近十年中，微软一直在尝试让用户以会员制的订阅模式来购买软件。但是这种努力一直收效甚微，因为客户不习惯这种购买方式。微软试图以 .NET 为契机，再次推动订阅模式，以实现自身的商业目的。

看起来微软推出 .NET 并不是为了满足客户的需求，而是为了满足微软自己的需求，给一万名程序员在接下来的十年中找点事干。众所周知，他们已经很久都想不出文字处理软件的新功能特性了，那这批闲下来的程序员该干点什么呢？

“梦想”的积极作用

一个老笑话是这么讲的：一个男人去看心理医生。医生向他展示了一张鸟的图片，问：“这张图片让你想到了什么？”男人说：“性。”医生又向他展示

了一张树的图片：“那么这张呢？”男人说：“性。”火车的图片。“性。”房子的图片。“性”。医生说：“天啊！难道你满脑子都是性吗？”男人怒答：“怎么能怪我呢？谁让你老是给我看这些色情图片！”

像是.NET 白皮书这样含糊其词的文件也有其积极作用，可以看作是罗夏墨迹测验。每个人都会用自己先入为主的观念尝试对它进行解读，因为它含糊其词，大家会产生微软与自己的观点不谋而合的错觉。UserLand 软件公司^①总裁戴夫·万纳^②对于软件有很多新颖有趣的观点。他读了微软.NET 白皮书后，产生了一种想法：微软终于认同了他两年来一直反复强调的观点。戴夫，你太抬举他们了。他们和你相比，完全是井底之蛙。他们玩的是心理咨询热线和报纸上星座专栏惯用的把戏：用一些模棱两可、泛泛而谈的话来描述你，你会不自觉地对号入座，以为自己真的被读心术看穿了。“今日的行星排列有助于你逾越障碍，实现夙愿。”戴夫的情况略有不同，他对于软件有真实、具体的想法，可以直接指导程序员去编写代码，而微软仍然停留在痴人说梦的状态，就好像他们六年前宣传 Cairo 将“让信息触手可及”，这个梦想最终被互联网实现了，而 Cairo 则早已被人遗忘。

但愿这些痴人说梦式的话语能启发某个人的创造性思维（比如 UserLand 的那位），最终酝酿出几个真正具有划时代意义的创新性产品，即使这些产品很有可能不会诞生于微软。



后记：这篇文章发表后，微软给几乎所有产品线的下一个版本都加上了.NET。后来，人们终于无法忍受这种命名方式引起的困惑了。在耗费了大量的人力物力后，.NET Server 被改成了现在大家熟悉的 Windows Server 2003，而.NET 的概念最终收敛到指代“托管代码”的新编程环境。顺便说一句，我发现这个编程环境相当不错。C# 和.NET 通用语言框架的双剑合璧，确实给这个新编程环境增色不少。它甚至真的能像.NET 白皮书里形容的那样，“降低风险，减少拥有软件所需要付出的总体成本”，有点“化腐朽为神奇”的味道！

① 参见www.userland.com/。

② 参见www.scripting.com/。



我们向.NET迁移的策略

2002年4月11日，星期四

Fog Creek 软件公司的软件开发工具逐渐迁移到了.NET平台，这篇文章记录了我对此的一些想法。

目前公司产品的开发状态是:CityDesk的代码大部分用 Visual Basic 6.0 编写,少部分用 Visual C++ 6.0 编写;FogBUGZ的代码大部分用 VBScript for ASP 写的,少部分用 C++编写;绝大部分的内部工具和网站互动代码(FogShop、产品讨论区等)都是用 VBScript for ASP 编写的。

我们为什么要迁移到.NET平台呢?简单地说,因为.NET似乎是目前最棒、生产效率最高的开发环境。ASP.NET让创建有用的网络应用变得格外容易。最近几天,我试着用.NET开发一些内部工具,发现开发效率高得难以置信。以前用ASP构建网络应用时的各种脏活累活(比如表单验证和错误报告)通常占整个开发工作量的75%,而.NET使得这些难题迎刃而解。从ASP到ASP.NET对开发效率的提升,几乎和从C到Java一样惊人。

C#借鉴了Java很多优秀的特性,还增加了几处小小的改进,例如自动装箱。我们在过去虽然一直努力遵循面向对象的原则,写出质量不错的ASP和VB6代码,但是大家还是觉得改用C#也非常不错。

还有一个原因,.NET自带的类型库太棒了。从数据存取到网站开发再到图形化用户界面开发,所有函数接口都经过了重新设计,这意味着各个层次的接口都具有极高的一致性。回头再看看老掉牙的Win32 API,就会觉得接口的一致性做得太差了,光是从函数调用的结果中得到字符串的方式就有好多种。对于哪种是正确的方式,Win32 API的设计者差不多每过两年就改一次主意。而.NET的类库把这些繁冗的接口都清理干净了。我非常喜欢ASP.NET

的日历小部件(其功能是生成一段 HTML 代码,让用户从按月份显示的日历上选择某个日期),它返回的结果一定是日期类型(我记得好像是 `System.DateTime`),而且和 SQL Server 接口使用的日期类型完全相同。我们之前在日期转换上浪费了难以想象的时间,包括让日期符合 SQL 的语法或是把 `COleDateTimes` 转换成其他类型的日期类型。此外,.NET 终于统一了字符串类型。上周我还在写 ATL 代码时和 `BSTR`、`OLECHAR`、`char*` 以及 `LPSTR` 较劲,真是不堪回首。谢天谢地,以后这些问题都将不复存在。

诚然,.NET 违反了“永远不要重新编写代码”的规则^①。微软之所以能打破这条规则,原因有二。第一,微软请来了世界上最棒的编程语言设计师安德斯·海尔斯伯格^②,软件开发行业近 20 年来 90% 的生产效率提升都归功于他:他设计了 Turbo Pascal (感谢!)、Delphi (再次感谢!)、WFC (虽败犹荣!) 以及这次的 .NET (又一次全垒打!)。第二,微软在三年内投入了大量工程师来重写代码,而在此期间,竞争对手基本上都没有什么动作。记住,微软能做到的事,你并不一定能做到。微软有能力按自己的决心行事,无法用一般的规则来衡量。

读到这里,大概有些读者又要给我写信抗议了。他们会说还有其他的开发环境,为什么不选择 Java,这样可以一次编写、到处运行(可笑),或者 Delphi (这门语言已经失去了灵魂,而 .NET 就是 Delphi 7.0、8.0 和 9.0 版本)、Lisp 什么的。他们会说:“你上了微软的贼船了!”遗憾的是,我目前没有闲工夫进行宗教信仰式的辩论,这种辩论通常都没什么意思。我不在乎日语是不是比英语要优越,因为它和我的生活没有关系。下面继续讲我们的迁移策略。

第一个问题:我们对 .NET 还不够熟悉,代码质量不高。和所有开发环境一样,完成某项任务通常有好几种不同的方式,而我们目前连一种方式都不了解,更别说是了解哪种方法更好了。因此,我们 .NET 代码的质量还达不到交付软件的标准。在比尔·沃恩的第一本 ADO 图书问世之前,我们还得自己摸索基本 SQL 查询的最佳方法。因此,我们的首要任务是自学 .NET,未来所有的内部软件和基于网络的软件(所有不用来卖钱的软件)都用 .NET 来练手。我们还可以把部分 FogShop 的代码迁移到 .NET,并且逐渐把目前使用的所有内部软件都迁移过去。(今天我用 ASP.NET 写了一个 FogShop 的兑换券生成器。虽然代码有点乱,但效果还不错!)

① 参见第 24 章。

② 约翰·奥斯本,“深入 C#: 微软首席架构师安德斯·海尔斯伯格专访”。O'Reilly.com, 2000 年 8 月 1 日。参见 windows.oreilly.com/news/hejlsberg_0800.html。

第二个问题：臃肿的 CLR (.NET 的运行时公共库) 要占用 20MB 内存。之前 CityDesk 网络安装包的体积是 8MB，其中运行时库和数据存取库就占了 6MB；我们无法强求每个使用 CityDesk 入门版的用户再下载 20MB 的 CLR。希望在两到三年内，CLR 能得到普及（它不支持 Windows XP 实在是太可惜了）。我们会持续关注这件事的进展。

总而言之，CityDesk 和 FogBUGZ 目前还没有办法移植到 .NET 平台上。将来 CLR 的装机量达到 75% 的时候，我们会考虑推出基于 .NET 的新版 CityDesk。我们的迁移计划是：

- (1) 先用微软提供的转换工具把现有代码和界面移植为 VB.NET；
- (2) 集中力量修复 bug，让所有功能恢复正常；
- (3) 用 C# 创建新界面和类；
- (4) 用 C# 逐个重写所有的旧界面和类，可以和大规模的功能升级同时进行；
- (5) 只要还能使用，很多旧界面和类将永远停留在 VB.NET 语言的阶段（可能是因为使用了太多向后兼容的字符串函数，等等）。

FogBUGZ 的迁移同样需要等待 CLR 在服务器上的装机量达到一定比例。我们想做一个客户调研，看看大家能否接受 FogBUGZ 把 CLR 作为安装先决条件。

我们还在研发一款神秘产品，这款产品将与 FogBUGZ 共享很大一部分代码（我们把这部分代码库称为 Dispatcho），所以 FogBUGZ 完成迁移之前，这款产品的代码仍将以 VBScript 和 ASP 为主。

对于 FogBUGZ/Dispatcho/神秘产品，迁移计划如下：

- (1) 等到 CLR 拥有足够多的装机量；
- (2) 把现有的“业务逻辑”类迁移到 C#；
- (3) 现有的网页表单继续使用 ASP；
- (4) 用 ASP.NET 开发新表单。

45

能给我一个链接器吗？

如果我是个偏执狂，就会说微软不想和系统、应用开发者分享好用的工具。因为微软怕他们开发出的产品会和自己的核心业务产品竞争，他们只想给那些构建与微软潜在业务无关的领域内垂直定制化软件的 IT 部门制作开发工具。但这么说太偏执了。先生，能给我一顶锡箔帽吗？^①

2004 年 1 月 28 日，星期三

不知道因为什么原因，微软最好用、最先进的 .NET 开发环境缺少了一个重要的工具。这个工具从开发环境这一概念出现以来就非常普及，大家对它熟视无睹，甚至没有注意到它在 .NET 平台上的缺失。

到底是什么工具呢？链接器（linker）。它的作用是把程序的编译后版本和程序中所有库函数的编译后版本合并起来，然后剔除掉所有不需要的库函数。最后，它生成一个单独的可执行二进制程序文件，用户就可以在计算机上使用了。

.NET 用一个叫作运行时的概念取代了链接器。运行时是一大段繁杂的代码，需要占用 22MB 内存。所有用 .NET 开发的程序，都必须依赖于这个不可拆分的运行时来做动态的代码链接，否则无法使用。

和 DLL 一样，运行时会带来各种麻烦。如果某个程序的版本 1 是在运行时版本 1 上开发的，那么在使用运行时版本 2 时，程序的版本 1 可能会出现各种莫名其妙的问题。比如我们公司内部使用的控制面板软件，在 .NET 运行时

① 偏执狂经常被塑造成一个头戴锡箔帽的形象，据说是害怕别人读取或控制自己的脑电波，参见 <http://baike.baidu.com/view/9331470.htm>。可参考动画片《飞出个未来大电影 4：绿色狂想》以及电影《X 战警》中万磁王的人物设定。——译者注

从 1.0 升级到 1.1 后，因为某种原因，所有的销售数字都被省略到了小数点后四位。兼容性问题通常比这还要严重。

事实上，.NET 包含了一种称为 manifest 的代码扩展技术系统。它看上去非常复杂，作用是确保某个程序使用正确的运行时版本，但我和身边的同事都不太清楚该怎么使用。

这里讲一个小插曲。在 Fog Creek 软件公司的新年晚会上，我们想让大厅的几台计算机显示器上出现新年倒计时。迈克只花了 60 秒，就用 C# 和 WinForms 写出了一个倒计时软件。我说过，.NET 是一个很棒的开发环境。

我的工作是要让这个叫 countdown.exe 的小软件在三台电脑上跑起来。听上去很简单吧？完全不是。双击可执行文件，弹出了一个用户体验很不好的错误提示，说找不到 mscoree.dll 文件，然后把 path 环境变量中的目录都显示了一遍。这其实是因为我没有安装 .NET 运行时，而从这个错误提示上完全看不出问题所在。还好我是个程序员，猜到了是怎么回事。

如何安装运行时？最“简单”的方式是通过 Windows Update 安装系统更新。但是 Windows Update 希望我先把所有重要的更新包都安装一遍，然后才让我装 .NET 运行时。听起来不算无理取闹吧？而这堆“重要更新”中包括一个 Windows 服务包以及新版 Internet Explorer，都需要重启才能完成安装。结果是，在每台需要运行这个 .NET 小软件的机器上，都需要先下载大小为 70 ~ 80MB 的文件（还好我们的网络足够快），再重启三到四次。我们可是软件公司！我清楚地知道这些操作一共花了多长时间，因为一开始下载更新的时候，我就把《上班一条虫》（*Office Space*）投放到大屏幕电视上观看，等到电影结束的时候，安装也差不多结束了。而在观影过程中，差不多每隔十分钟，我就得跑到每台机器前，在某个愚蠢的对话框上点击确定按钮。

对于公司内部使用的小软件来讲，这绝对是充满挫败感的安装体验。我们的产品 CityDesk 是绝对不能采用这种安装方式的。几乎所有的用户在购买产品前，都会下载我们的免费体验版^①。下载的文件大概是 9MB，没有额外的安装需求。这些用户基本上都没有安装过 .NET 运行时。

体验 CityDesk 的用户通常是小型机构和家庭用户，如果要求这些用户用看一部电影的时间去安装我们的软件，就为了试试好不好用，大概 95% 的人会中途放弃。这些人还不是我们的客户，而是潜在客户，我不可能为了一个好用

① 更多信息，请访问 www.fogcreek.com/CityDesk/Starter.html。

点的开发环境，就损失掉 95% 的潜在客户。

你可能会说：“但是乔尔，最终大多数人都会安装运行时，这个问题将不复存在。”

我之前也这么想，但是后来意识到，如果微软每隔半年或一年就推出新版的运行时，那么拥有正确运行时版本的用户数量将不升反降。如果我费了好大的劲才让程序在三个不同版本的运行时上正常工作，到头来只能覆盖安装了这三个版本之一的那 1.2% 用户，还不如不做。

我只想要一个链接器，把所有代码打包成一个可以独立安装和运行的可执行文件，即使文件大一点也没关系。我只想要那些我实际用到的功能函数、字节码解释器和一些运行时的公共库，并不需要无法从运行时中拆分出来的整个 C# 编译器。我可以保证，CityDesk 不需要编译 C# 的源代码。我不需要 22MB 的运行时，最多 5MB ~ 6MB 就够了。

我知道有的公司具有反向工程的技术能力，但是没有微软的许可，他们不能合法地将 .NET 运行时的某些部分（比如字节码解释器）重新打包发售。所以，微软请醒醒吧，把 20 世纪 50 年代就存在的链接器技术还给我们，让我们能够打包单独的可执行文件，不需要外部依赖就可以在 Win 98 以及更高版本的操作系统上运行。否则，对于通过网络下载发售的消费者软件，.NET 开发平台存在重大的缺陷。

第五部分

附录

“乔尔问答”精选

附录

“乔尔问答”精选

“乔尔问答”是我在个人网站上开设的一个讨论区，已经有几年的时间了。在这个版块里，我会和读者互动，并回答他们提出的各种问题。以下摘录了一些我个人比较喜欢的片段。

问：

我所在的公司最近几年项目经理泛滥成灾了，他们对于 polo 衫和卡其裤的品鉴知识比技术素养还要好。他们经常混淆各种技术术语，造成了不小的困惑。对于技术上的各种细微差异，他们更是一无所知。最糟糕的是，他们对于蹩脚工程师出的馊主意毫无抵抗力（也许那些蹩脚的工程师才是真正的问题所在，也许软件公司并不需要项目经理懂太多的技术细节）。

当然，这一切可能只是我个人的偏见。命运坎坷、满腹牢骚的我非常怀念过去的岁月，当年一封怒气冲冲的电子邮件就能把一个潜在的问题扼杀在摇篮之中。但是，该如何以平和的心态看待这种现象？项目经理除了具备应付本职工作的技术以外，到底需要对编程技术了解到何种程度？是我偏见太深了吗？

——匿名读者

答：

啊，polo 衫和卡其裤，真让我想念在雷德蒙德的岁月。周日去 Gap 采购一番，结果隔天上班时发现有一打项目经理和自己撞衫。这种事情在那里屡见不鲜。

我过去经常说，不尊重开发者的项目经理，工作效率一定不高，因为没有开发者的支持，他们什么也做不成。我在 Excel 团队的时候，如果项目经理不

懂技术，那么开发者就完全不听他（她）的。根据你说的情况，可能团队中的双方都不怎么样：项目经理能力不强，开发者也不是很厉害。

事实上，项目管理和软件开发的基因是互斥的。最完美的项目经理应该是那些能体会用户感受，具有组织才能和良好沟通技巧的软件开发者，但是这种人才实在太少了，微软在公司规模达到一定程度后就找不到足够的管理人才了。后来终于有人决定这个职位不再要求候选人具备编程知识，你猜怎么样？在微软，谁来面试项目经理？其他的项目经理。众所周知，A级人才聘用A级人才，而B级人才才会聘用C级人才。

如果说我在项目经理这个岗位上干得小有成就的话，那是因为我有一套（没错，就是两套）完整的基因。全世界就我这么一个。而且，在经历过一次恐怖的撞衫事件之后，我会极力避开 Gap^①，看见撞色条纹毛衣就绕着走，想买 polo 衫和卡其裤的时候，只逛贝尔维尤广场购物中心的另外 27 家店铺。

我曾和微软的项目经理探讨他们的工作职责，结果他们老是在说“我们的价值不在于球玩得有多好，而是在于能率先找到球，让别人没得玩！”之类冠冕堂皇的话。这么说来，程序员和项目经理都认为自己有权决定所有事情。这怎么可能呢？到底是谁真正说了算？我来给你一点提示。在你认识的所有项目经理和开发者中，哪些人的人际关系和沟通技巧更好？什么？大声点，我听不到！对！当然是项目经理了，不服不行。开发者的社交技能连比尔·盖茨在湖畔豪宅举行的暑假实习生聚会都应付不了。开发者们在这方面的意识非常薄弱，他们甚至不知道除了找对象之外，社交技巧还能用来做什么（“我……喜欢……丰满一点的女人……但是我不會甜言蜜语……”），所以难怪他们根本不知道这个秘密，而我今天就要在此公之于众。

为什么优秀的项目经理会让开发者觉得，所有重要的设计工作都是开发者在做，而项目经理只负责打打酱油，和一些非常官僚主义的事情周旋，比如“应付客户”“处理一些营销的破事”或者“顺便编写项目规范”？

其实，一名称职的开发经理最重要的技能，就是学会让自己的想法被开发者们实现，同时让开发者以为这是他们自己的主意。没错，穿 polo 衫是必要的道具，但是一名普通的开发经理一天之内就能成功做到三四次。我曾经看过几个在微软工作过的开发者们写的“自传”，里面详细记载了他们在微软工作的几年时光，其中一整章都在抱怨项目经理是多么愚蠢和多余，完全没有

① 此处的 Gap 是双关，暗指两套基因之间的差别。——译者注

意识到自己被项目经理操纵着——说明这些项目经理非常聪明。聪明的项目经理做这种事的时候，当事人完全没有感觉！其实这也不是很困难，因为程序员对于项目经理的心理攻势毫无招架之力，他们能想到的最厉害的反驳之词不过是：“我的回答是正确的，只不过你的问题不对！”

这些开发经理一边把开发者耍得团团转，一边表现得像个脑袋空空的高级俱乐部服务员，这需要多高超的演技啊！不仅要放下个人的自尊心（哦，你刚才的那个点子真聪明，开发者先生！），还要摆出一副以自我为中心的预科生做派。就是让好莱坞演技派明星肖恩·潘来扮演这么一个角色，都不一定能比微软一名普通的项目经理演得好。

问：

.NET 和 Longhorn 是最近非常流行的技术，你对未来 10 年软件开发行业的趋势有何看法？10 年后软件开发行业将是什么样子的？那时的人们会遵循哪些新的开发理念？你觉得最有可能出现哪些突破性进展？

——诺里克

答：

机器人将在厨房里为我们洗碗！机器人将在电梯里为我们按键！银行不再需要出纳员，取而代之的是一排排超级的现代计算机，能吐出崭新的钞票。（未来普通劳动者赚的钱也会多一些，去银行的时候能一次性取出 10、20 甚至 40 美元！）

人们将乘坐经过压力调校的飞行机器在城际穿梭，也许在云端至上，你也能吃到一顿简单可口的热饭热菜！

世界上最大的百科全书——比学校图书馆的藏书量还多——将通过气动管道或是电缆普及到千家万户。

一种特制的电传照相机器将把教育普及到全球每个角落，让最穷苦的清洁工人也有机会学习经典，提升自我。饥饿和苦难将被一扫而光。

另外，C# 将拥有匿名方法，能在使用委托的代码处直接写处理函数，方便地把处理逻辑代码直接绑定到委托实例。

问：

我们公司快要完成第一款产品的研发了。想法本身并不是特别新颖，但是市

场上只有一两个类似的产品，相信我们可以做得更好，因为它拥有竞品不具备的几个功能特性，提高了产品的附加值。但是，我们刚刚发现一个主要竞品推出了新版本，里面的新功能与我们的不谋而合。

大家有点沮丧。虽然我们觉得自己的产品在很多方面更加出色，但是与我们相似的产品已经经历了市场的考验，有更多的时间变得成熟。

我们下一步该把什么放在首位？开发新的功能特性，扩大竞争优势？还是完善现有的产品和功能集合，靠质量打败对手？希望你能提出宝贵意见。

——萨姆·托马斯

答：

不要关注竞争对手的一举一动。请跟着我念：

听客户的，不要听竞争对手的！

听客户的，不要听竞争对手的！

听客户的，不要听竞争对手的！

我们的 bug 跟踪软件有几十个竞争对手，我根本不理睬他们做了什么，也不知道他们为什么比我们强/弱。我根本不在乎。我只在乎客户的意见，光是应付他们我就忙不过来了。

尽快把软件推向市场，不要理会竞品。一定会有没听说过竞品的客户购买你的产品。也许数量不多，但这只是开始。定期访问客户，在公司网站的所有页面都写上电子邮件地址。你会从客户反馈中逐渐把握一些趋势：那些反复出现的意见能帮助解释为什么某些人没有选择你的产品。对于 FogBUGZ 1.0，问题在于不支持上传附件到 bug 报告里。我们在 FogBUGZ 2.0 中增加了这个功能特性。竞争对手有没有这个功能特性？对不起，我不知道。我觉得哪怕花十分钟去搞明白这件事都是在浪费时间。我关心的是：用户告诉我，只有能上传附件到 bug 报告，他们才会考虑购买我的产品。所以我们增加了这项功能。

第 38 章“企业发展战略系列三”中探讨了如何让客户放弃成熟的竞品，改用你们的产品。但目前，你的首要目标是尽快让软件面世，然后才能听取客户和潜在客户的反馈，找出下一步要做什么。

问：

再问一下，你们用了哪些方法来得到客户的有意义反馈？你们会主动给客户打电话、发电子邮件或者采取别的联络方式吗？

——dir

答：

我们从不主动联络客户。客户的反馈渠道主要有三个。

- CityDesk 中的“发送反馈”菜单项。用户发送的反馈会直接进入 bug 跟踪数据库，光是通过这个渠道就能获得大量（质量极高的）反馈，让我们都忙不过来。如果能有足够的时间来对功能需求进行统计就好了。
- 一般来说，网站所有页面的底部都有公司的电子邮箱地址。
- 在线产品论坛。

这三个渠道给了我们足够的用户反馈。我通常告诉德米特里（他负责处理我们公司绝大多数的技术支持工作）持续关注论坛中这样的帖子：“我喜欢你们的产品，但是因为 X，所以没有购买。”这种反馈要比“为什么你们的产品不支持 X 功能？”更值得重视，因为后者并不一定反映了客户“最终否定的原因”，可能只是一般性的抱怨或是异想天开。例如，有人在 Slashdot 上读了我的某篇文章，想知道 CityDesk 能否在 Linux 上运行，但是他和正准备刷卡购买的潜在客户是有区别的。或者，有人使用我们的产品后突然有了一个“很棒的主意”，想增加一个新功能。虽然这种建议值得听取，但是相比之下，不如“我喜欢你们的产品，但是因为 X 所以不买”这种意见显得更加重要和紧迫。

问：

从物流角度来看，有一件非常简单的事我就是想不通。为什么每个商店都提供两个塑料袋？为什么不把塑料袋做得更结实一点？虽然 Duane Reade 提供的大袋子看上去更结实，但是收银员还是给我套了两层，而我只买了一瓶水而已……有什么我没注意到的原因吗？

——一位好奇的纽约人

答：

因为收银员不能决定提供哪种袋子。

决定塑料袋种类的人是 Gristede 世界总部的会计，他们住在皇后区外围，开着 SUV 去超市购物，根本不了解用手提着塑料袋回家是何种感受。他们采购塑料袋时会选最便宜的那种，即使这样会导致收银员每次都给顾客两个，长期来看成本反而更高。他们从 20 世纪 70 年代的大停电以及由其引发的骚乱事件之后就没什么去过曼哈顿，偶尔前往也不过是为了再看一场《哦！加尔各答！》的演出，而那个剧场在 20 世纪 80 年代也关闭了。他们做梦也想不到，提着 12 瓶玛格丽塔混合酒徒步走过三个街区，然后登上 6 段长长的台阶，走到位于地狱厨房的月租 1700 美元的单间公寓是什么感受。

问：

预估开发时间的时候，应该如何预留缓冲时间，根据百分比还是固定时长，需要考虑哪些因素和标准？

——Vani

答：

如果能基于适当粒度的任务（每个任务大概需要一天时间）预估开发时间，并且有一定的相关经验，考虑到了各种情况（包括休假、节日、生病、集成时间、边开发边发明各种功能的“新功能开发”时间、参加各种管理会议浪费的时间、面试候选人的时间……），那么你只需要 10% 的缓冲时间。

最好不要大笔一挥，把所有的时间都笼统划为缓冲时间，应该对缓冲进行分类，按照各种类别的优先级合理分配时间。以下列举了几种可能的缓冲类别：

- 开发过程中临时想到的新功能；
- 竞争者有某些举动，我们需要立即采取应对措施；
- 把不同开发者的代码集成起来（根据团队熟悉程度的不同，这部分时间可能占 25% ~ 100%）；
- 在测试中寻找并修复 bug；
- 雇员必须履行的与开发无关的任务，例如“每个员工都必须参加的多样化培训”、公司紧急会议、防火演习、给老板送生日蛋糕等；
- 由于时间预估不足而引起的缓冲；
- 某些任务没有提供预估的时间，所以需要缓冲。

像这样把缓冲时间进行细分，以便仔细地对每一项进行跟踪和管理。如果目前任务完成了 80%，而“给老板送生日蛋糕”的时间预算只用了 20%，就可以把这部分时间匀给更加紧迫的任务。

问:

是否有可能为某个封闭环境写出一个安全的软件,让狡猾的社会工程学蠕虫病毒无法散播开来?

我觉得仅仅靠告诫用户“升级杀毒软件”“不要随便打开邮件的附件”是完全没用的,因为用户一直把这些说教当成耳旁风。还有一种选择是,禁止执行邮件中的二进制文件,但还是有人会把文件下载并保存下来,然后执行。况且很多蠕虫病毒还会伪装成安全升级软件或游戏,骗人们去点击它。

问题在于,人们区分不出哪些软件是可以在机器上安全运行的。能不能构建一种“监狱”式的环境,有效地限制蠕虫病毒的活动能力(或者把它明显地標示为“恶意软件”),从而阻止其大规模的传播?

——埃里克·塞伯伦

答:

这其实就是老生常谈的沙盒问题。

简短说就是:大概不能。

我最近把手机服务商换成了 Sprint PCS,因为他们提供的手机支持 Java 编程。我有很多想用 Java Applet 实现的想法。

- (1) 一个同步手机通讯录和 Outlook 通讯录的小软件。
- (2) 一个方便在手机响铃和留言模式之间切换的小软件。这样在健身房的时候,我就可以把手机切换到电话留言模式,告诉来电话的人“请电话留言,我将在一小时后回复”。在地铁里的时候,留言提示则会变成“我正在坐地铁,20 分钟后给您回电话。”

然后我开始研究 J2ME,结果发现:(1)我没法得到手机上存储的电话号码;(2)我没法得到响铃音量等手机设置信息;(3)我没法给别人打电话;(4)我不能利用手机内置的 GPS 来定位;(5)我不能使用手机内置的摄像头,等等。

这反映了 Sun 公司对于沙盒的态度:宁可减少灵活性,也要保证安全性。这导致所谓的手机开发环境形同虚设。J2ME 平台的命运将和浏览器中小软件的命运一样:沙盒唯一允许开发者做的事,就是写一些速度极慢的小游戏(手机版的吃豆人虽然挺好玩,但游戏加载要花 53 秒),别的什么都干不了。

微软对于沙盒的态度则完全不同——危险得多,但也有趣得多。在专业开发

者大会（PDC）上，他们展示了一个基于.NET的手机和一个用.NET写的手机小程序，只有10行代码。使用者用它可以拍摄一张照片，从GPS上获取地理位置信息并转化成街道地址，录制一段音频笔记，然后把这些都上传到某个数据库（就是那个众所周知的“保险精算师的辅助应用”）。这一切都是用J2ME办不到的，而手机的硬件显然完全支持这些功能。

我之所以提到古老的沙盒问题，原因要从Applet讲起。Applet诞生的时候，约翰·吉尔德多次表示，下一个文字处理器将以Java Applet的形式出现，但是Java Applet受沙盒的限制，没有读写硬盘文件的权限。如果Applet想保存某个文件，必须把文件上传到某个云端服务器上。有人认为把数据持久化操作限制在沙盒里也没什么不好，只要数据无法脱离沙盒就行。但这种观点的问题是，人们的数据都在沙盒以外。即使未来把所有的数据都封闭在沙盒里面，数据也还是没有什么安全性可言。你关心的所有数据都在沙盒里。事实就是如此：如果你想要运行任意修改图片的代码（把比尔·克林顿的脑袋安在马基·马克的身上），就必须获得运行任意修改图片代码的权限，而这样的代码很难辨别是否含有恶意。对不起，没有别的办法。

微软和Sun公司都在寻求一个“粒度更细”的安全模型，可以对不同的应用采取不同的权限策略。比如，吃豆人游戏可以读写高分榜文件，但是不能访问互联网。这种安全策略可以做得要多复杂有多复杂。

但是任何对于安全领域有所涉猎的人都会告诉你，这种方法行不通。安全系统越复杂，配置出错的概率就越高。人类所能处理的复杂度是有限的，没人有闲工夫去管理所有应用的各种权限。

总而言之，情况就是这样不容乐观，就像我们无法阻止暴徒袭击老奶奶抢走她们的钱包一样。不过我们可以用刑罚来惩戒暴徒，还可以把干过坏事的人关起来，但是如果一个人一心作恶，那么还是可以得逞的。类似地，如果有人一心寻死，就没人救得了他：如果开着汽车全速冲向悬崖，没人能拦得住。

因为计算机世界比真实世界更加容易控制，人们就希望能通过技术手段“保护人们不受自己的伤害”。但这根本不可能，就像汽车厂商永远也做不出拒绝开下悬崖的汽车，老奶奶不可能全副武装地去买菜一样。生活就是这样，所以我个人觉得，大可不必为此而过于担心乃至失眠。

问：

为什么上市公司把股票记入账务很重要呢？通过给出一部分股权来激励员

工为公司的成功而奋斗似乎是个不错的想法。为什么把股票记入账务会让发放优先认股权变得不那么诱人呢？

——杰森

答：

假设你是一家大型软件公司的总裁，公司股价以每年 100% 的速度上涨。通过分给雇员们一点点股权，你就给了他们价值不菲的美元等价物，而实际上并没有支出一分钱。实际上，你甚至可以降低基础工资，他们根本不会在乎，因为股票的迅速升值早就让他们乐开了花。20 世纪 80 年代，微软的基本工资就比竞争对手的低 30% ~ 40%，而那段时期的微软员工，现在每一个都是百万富翁。

但是，如果不花真金白银，这么多股票是从哪里来的呢？通过发行更多的股票。这会稀释现有持股人所拥有的股票价值。举个简单的例子，如果目前公司股票的发行总量是 100 万股，为了能以低于交易价的价格发给员工股票，就需要再发行 100 万股。现在的发行总量是 200 万股。在公司市值不变的情况下，每股的价值就变成了原来的一半，这等于从股东的口袋里掏钱再发给员工。

这么做合法吗？如果股东不同意，就不合法，但股东一般都会同意，因为他们有两个选择：要么从银行账户里拨款付给员工，这会降低公司每股的价值；要么通过增发股票来给员工发股权，这还是会降低公司每股的价值。

明白了吗？我再举一个例子。这个例子有点难以理解，所以请注意。

假如我是一个软件公司的总裁，公司市值 200 万美元。股票发行总量是 100 万股，每股的价值是 2 美元。

现在我需要给员工发薪水了。总共需要 100 万美元。

场景一：我从银行户头里支出这笔钱。我的公司本来值 200 万，现在只有 100 万了，因为我们刚刚支出了 100 万。一旦这笔钱划出去，公司价值就减少了 100 万，是吧？因此每股的价值就从 2 美元降低到了 1 美元。

场景二：不付给员工现金。我得到了股东的许可，再发行 100 万股票，全部分给员工。这样股票发行总量从 100 万股提升到了 200 万股。公司的市值没变，仍然是 200 万，因为我一分钱都没花；没有任何资金流出公司。于是每股的价值从 2 美元降低至 1 美元，因为总量提高了一倍。

两个场景的相同点是，每股的价值都从 2 美元降低到了 1 美元。那它们之间有何不同呢？

□ 员工都得到了 100 万美元的价值。

□ 股东手中股票的价值都减少了一半，从每股 2 美元降到每股 1 美元。

但最重要的是：

□ 在老式审计规则下，如果不把股票记入账务，那么在场景二中，财务报表上展现出来的利润会比场景一多 100 万美元。

我们刚才一直没有提到利润。利润即收入减去支出。两个场景的收入是相同的，不同的是那 100 万美元的价值是否从公司流出。由于两种场景在意图和效果上是一样的，那么场景二的公司在财报上多出 100 万美元利润其实是不公平的。

换句话说，两种场景对于员工来说是等效的，对于股东也是等效的。那么，场景二多出 100 万的利润，并没有反映出公司盈利能力的实际变化，不可能因为你选择了稀释股票而没有选择减少银行存款，就能让公司的盈利能力有所增长。对于股东来说，这两个场景只是资金运作方式上的差别。

在老式审计系统下，华尔街提出的“每股收益”计算方法是不可靠的。很多公司用每股收益的计算方法来衡量利润，盈利能力都非常惊人，但实际上根本不是那么回事。

投资者在比较公司的投资前景时，会参考所谓的市盈率（每股价格除以每股收益）。股东可以要求公司把股票记入账务，即要求他们把财报中表现出来的利润进行降低，抵消由于发行新股、稀释股权所引起的利润增加。这样，就能让场景一和场景二的公司在财报中表现出相同的利润，使得财报上的利润数额对投资者具有更高的参考价值，更加便于准确比较两家公司的盈利能力。股票分析师就不需要分析到底哪部分利润是由发行新股、稀释股权所导致的，在比较两家公司时更加容易。只要分析公司的盈利情况，就能确认给员工发股票是否会导致公司受到股权稀释造成的冲击。以前在市场还不规范的时候，分析师比较每股盈利的唯一诚实方法，就是仔细核查年度财报，估计出由新股发行而造成的股权稀释对利润的影响是多少，而这部分信息通常都不完备或者有一定延迟。其实大家都心知肚明，股票分析师根本不会这么费力地分析，那些只看 Motley Fool 股市评论的人们只懂得比较市盈率，完全不了解公司给员工发放股票的行为会对收益数字的真实性造成不可预测

的影响。

综上所述，上市公司把股票记入账务是为了保证发行新股所造成的股权稀释不会对公司市盈率的判断造成不公平的影响，向股东展现更加公平和真实的公司盈利情况。

问：

不知道以前有没有人提过这个问题，当所有人的家里和公司都配备了 100MB 带宽网络的时候，你觉得微软的命运会怎样呢？

因为网络足够快的时候，人们就不需要 Windows 或 Office 了，只要运行各种 Web 应用就能满足日常需要。微软的软件就没有存在的必要了。

例如，我现在就在用 Hotmail 的在线服务作为电子邮件客户端。如果本地和远程存储服务之间的网络连接足够快的话，将来我可能会使用在线版本的文字处理或电子表格软件。这比桌面软件要方便多了，因为我随时随地都能得到自己的数据，就像 Hotmail 一样。

——罗伊

答：

等一下，牛仔！虽然你的观察抓住了一点小小的趋势，但你的推论未免太脱离实际了。

首先，我非常怀疑一个网页形式的文字处理软件能有多大使用价值，尤其是考虑到 HTML 的发展现状。为 IE 6.x 设计的用户界面又丑又慢。这绝不是什么偶然现象——微软必然会从中作梗，不会坐视 HTML 的用户体验发展得更接近于本地应用。

其次，现在有很多人的带宽连接速度已经很高了，但是我还没见过一个愿意使用网页版文字处理或电子表格软件的人。完全不用担心微软的 Office 系列软件卖不出去。

即使 Office 的销量会受到基于网络浏览器的生产力工具影响，微软也迟早会进军这个产业。

20 世纪 90 年代初，大家都以为 IBM 会逐渐消亡，觉得它不懂 PC 产业，无法摆脱大型机时代的思维。但是人们忘了，一个大型、成功的企业，其衰败也不是一朝一夕的事情。在此过程中，该企业有很多机会站稳脚跟，遏制下

滑的势头。微软也是如此。无论人们对微软的衰败原因有怎样的预测（Web应用崛起、宽带的普及、Linux、反垄断法、甚至是阴谋论者偏爱的黑色直升机），都无法改变一个事实：即使微软从此刻开始一分钱都不挣，仅靠目前的现金储备也足够维持五年的时间，一名程序员都不用裁。况且他们的收入也不可能突然变成零；即使作最坏的打算，每年收入下降 20%，微软也还能支撑 10 年，然后再考虑进行第一轮裁员。

所以，仅仅提出某种能够威胁到微软命运的技术变革，并不足以预言微软的衰亡，还得假定他们在很多年的时间里一直不采取任何调整措施来应对这种变革。