程序员最可信赖的求职帮手

译 者 序

2008年底，当我拿到300页的*More Joel on Software*时，并未料到，翻译此书竟然需要超过9个月的时间。我生活中的每件事，都因为它而延后了。打字的疲惫、尽快交稿的压力、单调工作引发的烦躁以及苦思冥想依然不解其义的愤懑，都不必提了。如果早知道要过9个月这样的日子，我不会答应翻译这本书。

不过，这确实是一本好书，一定会广为流传，许多年后还有人阅读。所以不管怎样，我可能还是会答应翻译它。因为有时你会头脑发热，希望能够参与到某种不平凡的事件之中，希望自己的名字和这样的东西联系在一起……谁知道呢。

在翻译过程中，我得到了很多帮助，在此表示感谢。

感谢出版社的责任编辑，在我一再延迟交稿的情况下，给予我的宽容。

感谢台湾地区的译者。他们无私地将Joel的许多文章译成中文，放上了网络上（local.joelonsoftware.com）。我参考了他们的译文，并从中获得了启发。

感谢给我留言指出翻译错误的网友。他们每个人的留言，都保存在我的网志（ruanyifeng.com/blog/it/mjos/）上。

感谢Google字典（www.google.com/dictionary）和维基百科（en.wikipedia.org）。没有这两个工具网站，我的译稿不可能是现在的样子，许多地方我永远也不会看懂。我认为Google字典是目前最强大的电子词典，而维基百科则是我能想到的人世间最美好的东西。

最后，感谢你的阅读，但愿你喜欢我的翻译。

我为这个中译本做了一个主页，网址是http://www.ruanyifeng.com/mjos/，你可以在那里找到更多关于本书的信息，还可以提交反馈。

中 文 版 序

欢迎阅读*More Joel On Software*的中文版。

毫无疑问，全世界的贸易壁垒都在消退，但是全球软件行业彼此隔离的现状却仍然十分惊人。其实我们大多数人都在使用同样的工具和技术，比如UNIX、互联网、C#、Windows、面向对象编程等。中国程序员用来解决问题的工具基本上与世界上其他地方的程序员使用的工具是一样的。

因此，我很高兴，我的一些疯狂的想法能够被远在中国的你读到。这要归功于图灵公司以及中文版的译者和编辑，由于他们的辛勤劳动，我们之间的语言隔阂才得以消除。

你可以把这本书送给你的老板，但是我认为这主意很糟糕。因为许多国家的老板都在怒气冲冲地讨伐我，我一天到晚都在收这样的电子邮件，我可不想再收到更多了。这本书是纸质的，用起来比网站方便多了，你随时都可以把书撕了，用来垫鸟笼或者包裹东西。我向你保证，这是你能找到的最有成效地使用这本书的方法了。如果你一拿到书就这样做，你就不会受我的胡说八道的影响了。

希望你喜欢这本书！

Welcome to the Chinese edition of *More Joel on Software*.

There is no question that, even as trade barriers fall all around the world, many parts of the software industry are still surprisingly isolated from one another. We use, mostly, the same tools and technologies. Unix, the Internet, C#, Windows, Object-Oriented programming: the tools a Chinese programmer uses to solve problems are mostly the same as any programmer in the world.

So I am very happy that thanks to the Turing Book Company, and the hard work of the Chinese translators and editors, you now have the ability to read some of my crazy ideas without a language barrier.

You will be able to give them to your boss to read, which, I assure you, is a very bad idea: I constantly receive angry email from bosses already in many countries and don't need to add another. Also, the convenient paper format means that, unlike a website, you will be able to tear these pages out and use them to line bird cages or to wrap fish, which, I assure you, is the most productive thing you could possibly do. If you do that first, you won't have to read any of this nonsense.

Enjoy!

Joel Spolsky　2009年11月于美国纽约

Joel、Apress[①]、网志[②]和网志书

“很久以前，在一个很遥远、很遥远的星系中，……”[③]好吧，实际上没有那么久啦，那是在2000年接近年底的时候，Apress出版公司正式运营刚满一年。当时，我们只是一家非常小的计算机书籍出版商，毫无名气。那一年，我们计划出版的书籍只有很少几本，大概只相当于Apress现在一个月的出版量。

那时，我苦苦学习如何成为一个出版商，可能花费了过多的时间，忙于浏览网站和编写程序。有一天，我偶然来到了一个叫做“Joel谈软件”（*Joel on Software*）的网站，发现网站的主人是一个观点鲜明的家伙，他的写作风格有点不寻常，很聪明并且还有意挑战一些传统观念。最特别的是，那时他正在写一组系列文章，批评大多数软件的用户界面是多么糟糕。总的来说，这主要是因为程序员们对用户的实际需求几乎毫无所知——用Joel和我经常使用的话说，这叫做“bupkis”（近乎没有），这是一句来源于意第绪语[④]的纽约土话。我同许多其他人一样，被Joel的这组系列文章以及其他几篇随笔吸引住了。

然后，我就冒出了一个想法：我是出版商，我喜欢读他的文章，那么为什么不出书呢？我给Joel写信，自我介绍了一番。虽然他起初有些怀疑，但是我不知怎地就说服他相信，如果他将那组用户界面的系列文章写成一本书，会有很多人购买，我和他都会赚到很多钱。（当然，那是发生在很久以

① Apress是本书原版的出版者，一家著名的计算机书籍出版商，总部在加州的伯克利。——译者注（除非另有说明，本书中所有注释均为译者所加，下文不再一一声明。）

② 本书中“blog”一词统一译为“网志”或“网络日志”。

③ 原文为“A long time ago in a galaxy far, far away ...”，这是美国经典科幻电影《星球大战》（*Star Wars*）的片首字幕。20世纪六七十年代出生的美国人，几乎人人耳熟能详。

④ 意第绪语（Yiddish）是犹太人的民族语言，本书作者Joel Spolsky是犹太人。

前的事情，那时FogBugz[1]还没有变得像今天这样成功，Joel也还不是一个令人羡慕的收入颇丰的演讲者。不过，那时我们都比现在年轻，并且正如你想的那样，比现在穷得多。）

闲话少说，Joel后来又为新书加入一些新内容，使得它更具吸引力，我觉得也更有销路了。突然之间，Apress就必须考虑如何出版它的第一本全彩书籍了。*User Interface Design for Programmers*（《面向程序员的用户界面设计》）正式出版是在2001年6月21日。现在，它被公认为有史以来第一本“网志书”（blook）[2]。令计算机图书行业和我本人有些震惊的是，按照当时的畅销标准，它竟然成了一本很优秀的畅销书。顺便说一句，直到今天，它仍然在不断重印，仍然卖得非常好，仍然值得一读。（不过，作为Joel的出版商，而不是作为朋友，我想对他说：你是不是该考虑出个修订版了？）

不过，还是有人出来说，*User Interface Design for Programmers*并不是一本纯粹的“网志书”，因为加入了“太多的”网站上没有的新内容，使得这本书看上去更像一个混合体——我的看法是，这正同它的先锋地位相适合。

短短几年之后，“Joel谈软件”成了全世界程序员中最著名的网志，原因当然是Joel一直不停地写作那些非常有趣的文章。其中最著名的大概是那篇经典文章《微软公司如何在API战争中失利》（*How Microsoft Lost the API War*）。据我所知，这篇文章着实把微软的开发部门折腾得够呛。

这样，我就有了另一个想法：将Joel最好的那些文章收集起来，再出一本书，不做大的变动，除了加上一篇字数很少的前言，只要Joel觉得合适就可以。这样一本书的名字就叫做《Joel谈软件》（*Joel on Software*）。即使书中98%的内容都能在互联网上找到，即使人们认定Apress出版这样一本书一定是疯了，它还是在2004年底出版了。今天，这本书已经印刷了10次，而且依然是一本畅销书。

为什么呢？人们的阅读习惯并没有改变，在像品尝美味的巧克力糖果一样品味Joel的文章时，很多人仍然习惯于看书而不是看浏览器。

但是，Joel并没有因此停下来，他依然在努力思索如何才能更好地编程，

① FogBugz是Joel后来创业时的一个商业软件产品。

② “网志书”（blook）是一个20世纪90年代出现的新词，词源为bl(og)+(b)ook，意即由网络日志的文章中编辑而成的书籍。

或者怎样招聘到优秀的程序员，他也没有停止用自己的观点挑战传统看法。所以，我说服他，现在可以出一本续集，收录2004年底上一本书出版之后的那些“Joel的精华文章”。

结果就是你现在手里拿的这第二本文集，Joel的观点、随感以及偶尔的夸夸其谈都浓缩在了他才华横溢的文章之中。除了少量的编辑加工，原文几乎毫无变动，但是同显示器屏幕或者Kindle阅读器[①]相比，你确实以一种非常不同的形式拥有了最新的“Joel的精华文章”，现在这被称为“网志书”。（我要对Joel说，我很希望你像中意第一本集子里那些文章那样，中意这本集子里的文章。）

这本书同第一本一样，有着不同寻常的封面和副标题。这是因为Joel和我都是藏书爱好者（好吧，Joel才是藏书爱好者，我是藏书狂人）。17世纪和18世纪那些经典著作的印刷商，为了让他们的书变得生动，往往会做一些特别的设计，我们两人都非常喜欢这一类东西。在第一本《Joel谈软件》的封面上，我们向伯顿（Burton）的《忧郁的剖析》[②]（*Anatomy of Melancholy*）致敬；这一本的封面上，我们向霍布斯（Hobbes）的《利维坦》[③]（*The Leviathan*）致敬，它的封面很著名，一个巨人由许多个小人组成。Joel和我都感到这个隐喻很不错，可以暗示程序是如何编写完成的：宏伟的整体由个体组成，并且个体是关键。

最后，是一点很个人化的说明：尽管现在Joel的名气很大，但他依然是一个很朴实的人，或者再一次用我们共同的土话说，是一个真正的“mensch”（好人）。我非常骄傲，我有这样一个好朋友。

Gary Cornell

Apress出版公司创始人

① Kindle阅读器是Amazon网上书店于2007年底推出的一种手持电子阅读设备。

② 罗伯特·伯顿（Robert Burton，1577—1640）是17世纪英国的著名学者，他的代表作《忧郁的剖析》（*Anatomy of Melancholy*）出版于1621年，表面上是一本心理医学的教科书，但实际上是英国文学的经典作品。

③ 托马斯·霍布斯（Thomas Hobbes, 1588－1679）是17世纪英国的著名政治学家和哲学家，他的代表作《利维坦》（*The Leviathan*）出版于1651年，从哲学上讨论国家的起源、性质和作用，是古典学术名著。

第一部分 人员管理

1 我的第一次BillG审查

2 寻找优秀的程序员

3 寻找优秀的程序员之实战指南

4 三种管理方法

5 军事化管理法

6 经济利益驱动法

7 认同法

1

我的第一次BillG审查

2006年6月16日，星期五

早先，Excel[①]有一种没有名字的很蹩脚的编程语言。我们管它叫做“Excel宏语言”（Excel Macros）。这是一种功能很弱的编程语言，它没有变量（你不得不将值存在电子表格的单元格中），没有局部变量（local），没有子例程的调用（subroutine call）；一句话，它的程序几乎无法维护。不过，它也有一些高级语言的特性，比如任意跳转语句Goto，但是label标签实际上是看不到的[②]。

这种语言存在的唯一一点合理性，在于比起Lotus[③]中的宏语言，它看上去很强大。Lotus宏语言编程则仅仅是在单元格中敲入一个长字符串。

1991年6月17日，我进入微软公司的Excel开发小组工作。我的头衔是“程序经理[④]”（Program Manager），安排给我的职责是为Excel宏语言找到一个解决方案。实际上，所谓解决方案就是要让它与Basic[⑤]编程语言联系起来。

---

① Excel是微软公司的电子表格软件，第一个版本是在1985年发布的，从1993年第5版起，被并入Microsoft Office软件套装中。

② Goto语句通常和label配套使用，使程序直接跳转到label处。一般认为，这种跳转功能对结构化编程是一种破坏，不提倡使用。作者在这里说：“label实际上是看不到的”，意即Goto语句在“Excel宏语言”中，只能直接跳转到行号，比label还不如。

③ 这里Lotus指的是电子表格软件Lotus 1-2-3，这是20世纪80年代最流行的电子表格软件。微软开发Excel的直接目的，就是与Lotus 1-2-3进行竞争。

④ 微软的程序经理是个很特殊的职位，选择标准是：技术水平是程序员队伍中的最高级别，能做最多且最难的工作，有人格魅力。比尔·盖茨将程序经理描述为程序员队伍中最聪明的那个家伙。——编者注

⑤ Basic语言是高级编程语言之一，最早是在1964年出现的。它是微软公司起家的编程语言。

Basic？一点没错！

我花了一些时间，与不同的开发小组进行沟通。那时，Visual Basic 1.0[①]刚刚发布，它真是太酷了。其中包括一个方向错误但还在开发中的项目，代号是MacroMan。另外，还有一种面向对象（Object-Oriented）的Basic也在开发，代号是Silver。Silver开发小组被告知，他们的产品将有一个客户：Excel。Silver的销售经理是Bob Wyman（是的，就是Bob Wyman那个家伙），他只需将他们的技术推销给一个人：就是我。

就像我前面说的，MacroMan的方向错了，开始有人还不信，不过最后它还是停止了开发。Excel开发小组说服Basic开发小组，我们真正需要的是针对Excel开发的某种Visual Basic。我设法在Basic里加上了4样好东西。我让他们加上了变型（Variant），这是一种可以储存任何类型数据的数据类型，否则每次要储存一个单元格中的变量值，你就不得不用switch语句先进行一番判断。我还让他们加上“后期绑定”（late binding），后来它被叫做IDispatch接口，又称COM自动化，因为Silver的原始设计对理解类型系统（type system）的要求很高，而Excel宏语言的各种编程者根本不需要懂这个东西。另外两个我得到的很棒的语法结构是从UNIX的shell语言csh中借鉴的For Each结构，以及从Pascal语言中借鉴的With结构。

然后，我开始坐下来，写Excel Basic的规格说明书。这个文档真是巨大，有几百页。在我的记忆中，写完的时候，长度是500页。（“瀑布开发法[②]”，有人在偷笑了。是的，就是这样，别笑了。）

那个时候，我们有一档子事叫做“BillG审查”。基本上，比尔·盖茨会审查每一个重大的功能。我被通知将规格说明书复印一份，送到他的办公室，准备接受审查。那差不多用掉了整整一包打印纸。

我急忙将规格说明书打印了出来，送到了他的办公室。

那天晚些时候，我有了一点儿空闲时间。于是，我就开始琢磨，Basic的日期和时间函数是否足以完成所有能在Excel中完成的任务。

---

① Visual Basic是微软公司基于Basic语言开发的第三代事件驱动的编程语言，主要特点是完全在图形界面上进行编程。Visual Basic 1.0是在1991年5月发布的。

② 瀑布开发法（Waterfall），是一种软件开发方式，按照顺序从头到尾一步步完成，就好像垂直的瀑布一样，完成上一个阶段后，再前进到下一个阶段。现在一般认为这种方式已经过时。

程序员最可信赖的求职帮手

在大多数现代编程环境中，日期都是以实数形式存储的。实数的整数部分是从过去某个公认的日期至今所经过的天数，这个公认的日期在Excel中叫做“纪元”（epoch）。比如，今天是2006年6月16日，存储的值是38884，而对1900年1月1日来说，存储的值就是1。

我开始看Basic和Excel里的各种日期和时间函数，东试试西试试，结果我注意到Visual Basic的文档有一个问题：Basic的纪元不是1900年1月1日，而是1899年12月31日，但是奇怪的是，当天日期的值在Basic和Excel里居然是相同的。

原因何在？

我找来一位Excel开发人员，他资历很老，应该记得这些事。艾德·弗莱斯[①]看来知道答案。

“哦，”他告诉我，“检查一下1900年2月28日。”

“存储值是59。”我说。

“再去看3月1日。”

“存储值是61！”

“60是哪一天？”艾德问。

“2月29日，1900年是闰年！它能够被4整除！”

“思路正确，不过猜错了。”艾德说，我愣在那里。

噢，可恶。我又想了想。所有能够被100整除的年份中，只有能够被400整除的年份才是闰年。

1900年不是闰年。

“Excel里有bug！”我惊呼。

“嗯，其实不是啦，”艾德说，“我们不得不这样设计，因为我们需要导入Lotus 1-2-3的工作表。”

① 艾德·弗莱斯（Ed Fries），后来转为开发游戏，成为微软游戏部门的副总裁，并且是Xbox的早期主要开发人员之一。他已于2004年1月离开微软。

“你是说，这是Lotus里的bug？”

“对，不过可能这是故意设计的。Lotus占用的内存不能大于640KB。这是很有限的空间。如果忽略1900年，那么就可以根据最右面的两位数字是否为0，判断任意一年是否为闰年。那就简便多了。Lotus的那些家伙可能觉得，过去的所有时间中只有两个月受到影响，不是很重要的事。但是看起来，Basic的那些家伙不想放过这两个月，所以他们将纪元向前推了一天。”

“天啊！”我发出了一声感叹，转而继续研究，为什么一个名为“1904日期系统”[①]（1904 Date System）的选项对话框（options dialog）中有一个复选框（check box）。

第二天就是令人紧张的BillG审查日。

1992年6月30日。

那个时候，微软还没有那么多官僚机构。今天，整个微软公司的管理层一共有11到12个层级吧。那时不是这样，我向Mike Conte报告，他向Chris Graham报告，后者再向Pete Higgins报告，后者再向Mike Maples报告，后者再向比尔·盖茨报告。从上到下，一共6层。我们曾经取笑通用汽车那样的公司，因为它们有8个管理层或者天知道是干什么的层。

在我的BillG审查会上，上面提到的那些人都到场了，每一个人还带着一大堆听众，我怀疑他们将表兄表妹、七大姑八大姨都带来了。另外，还有一个家伙是和我一个团队的，他在会议期间的所有工作，就是负责准确记录比尔爆了多少次粗口。比尔说Fxxx这个词的次数越少，就代表审查的结果越好。

比尔进来了。

我觉得很不正常，他几乎和普通人一模一样，也是两条腿、两只手、一个脑袋。

他的手里拿着我写的规格说明书。

他的手里拿着我写的规格说明书！

① 1904日期系统是微软公司为了解决1900年闰年问题设计的另一个日期系统。它支持的日期纪元是1904年1月1日。

他坐下后，同一个我不认识的经理说了几句俏皮话，我没听懂。一些人倒是哈哈大笑。

比尔转向我。

我注意到，规格说明书的页边空白处写有评语。他看过第一页！

他看了我写的规格说明书，并且在空白处留下了几句评语！

因为我们是在大约24小时前交给他这份文件的，所以他一定是在昨天晚上看的。

他提问，我回答。那些问题很容易，但是后来我极尽所能，也想不起来他当时到底问了些什么问题。因为我一直目不转睛地看着他快速翻动那份说明书……

他在翻我写的规格说明书！[淡定，难道你是没见过世面的小姑娘？]

……我还看到，所有的空白处都写着评语。文件的每一页都是如此。天哪，他居然从头看到了尾，并且在空白处写了评语。

他看了所有内容！[我的老天啊，怎么可能！]

他的提问越来越难，也越来越细了。

那些问题似乎不太有条理。到这个时候，我已经把比尔当成自己人了。他真是不错！他看了我写的规格说明书！他想问我的，大概都跟那些页边上的评语有关吧！我要在错误提交系统中，把他的每一条评语都放进去，并且确保得到处理，*一定要快！*

最后是一个很要命的问题。

“我不知道，诸位，”比尔说，“你们有人真地看过有关实施的所有细节吗？比如，所有这些日期和时间函数。Excel有那么多日期和时间函数，Basic也要有相同的函数吗？能保证它们的行为都一样吗？”

“是的，”我说，“只有1900年的1月和2月除外。”

一片寂静。

那个粗口记录员和我的上司惊讶地对视了一眼。他们一定奇怪，我怎么会知道那个？1月和2月是什么鬼东西？

“行了。好，做得不错，”比尔说。他拿起那份写满评语的规格说明书。

……别啊！我要那个……

他离开了。

“4次，”粗口记录员宣布，每个人听了都说：“哇，这是我记忆中的最低纪录。比尔随着他的年龄增长变稳重了。” 那一年，你们都知道，他36岁。

后来，我自己是这样想的：“比尔并不是真地想来评论你写的东西，他只是想确定你对实现那些目标是不是有把握。他的标准做法是不断地提问，越问越难，直到你答不上来，承认自己不知道为止。然后，他会冲着你吼‘为什么没有准备好’那个他准备好的最难的问题，如果你答出来了，没人知道会怎么样，因为还没有人答出来过。”

“如果提问题的是Jim Manzi呢？他会提什么问题？”有人问。“他大概会问你，‘什么是日期函数？’”

Jim Manzi只知道MBA（工商管理硕士）。Lotus在他的领导下，走了下坡路。

这是很重要的一点。比尔·盖茨对技术的了解令人惊叹。他理解可变数据类型、COM对象、IDispatch接口以及Automation与虚表有何不同，他明白这种不同可能会导致双重接口（dual interface）。因此，他担心日期函数并非心血来潮。如果他信任那个干事的人，他就不会干涉软件。但是，你不要糊弄他，哪怕是一分钟，因为他也是一个程序员，一个真正的、现实的程序员。

不懂编程的人管理软件公司，就好像不懂冲浪的人硬要去冲浪。

“没关系的！我请了非常棒的顾问，他们在岸上告诉我怎么做！”那些人会这样说。但话音未落，就会一头从冲浪板上摔下来，而且乐此不疲。这

是那些MBA的标准说辞，他们从心底里相信，管理是一种通用职能。史蒂夫·鲍尔默[①]会不会成为第二个约翰·斯考利[②]？后者几乎让苹果公司破产，原因仅仅是，那时的苹果公司董事会相信，知道如何卖百事可乐就可以管理好一家计算机公司。迷信MBA的人们总是愿意相信，懂不懂公司业务没关系，只要懂管理就行。

多年以来，微软公司逐渐变得庞大，比尔·盖茨的精力被分散了，一些道德上有瑕疵的决策使公司的管理层不得不将大量的精力转向与美国政府抗争[③]。史蒂夫·鲍尔默接任CEO[④]，在理论上可以让比尔将更多的时间花在他最擅长的事情上，也就是管理软件开发组织。但是，这看上去好像无助于解决某些因为特殊原因引起的内部问题，比如像11层的管理结构，永无止境的开会文化，一种要将所有可能的东西都创造出来而不管这样东西是什么的顽固倾向（想一想吧，他们决定做一个网络浏览器，而且还要免费发布，结果在研发、打官司、公司名誉上面损失了几十亿美元），以及长期以来使得中层干部素质下降的匆忙、草率的招聘机制。（就像Douglas Coupland在*Microserfs*[⑤]一书中所说的："他们在1992年雇用了3100人，其中并不都是人才。"）

好了，不说了。聚会要到其他地方开了。后来，Excel Basic成了微软Visual Basic应用程序语言Excel版（Visual Basic for Applications for Microsoft Excel），里面的注册商标标志™和权利保留符号®多到我都不知道怎样才能将它们都放进去。我在1994年离开了微软，我觉得比尔已经彻底将我忘了。

---

① 史蒂夫·鲍尔默，微软公司CEO。1977年，他从哈佛大学本科毕业，专业是数学和经济学。1980年，他从哈佛商学院退学，应比尔·盖茨的邀请，加入微软公司，担任微软的第一任商业事务经理（business manager）。他是微软历史上的第24名员工。

② 约翰·斯考利，一位美国专业经理人。1977年至1983年，他担任百事可乐公司的总裁。1983年4月，他应创始人斯蒂夫·乔布斯的邀请加入苹果公司，担任CEO，一直到1993年离职。1985年，他与乔布斯之间的矛盾升级，迫使后者离开苹果公司。此后，在他的领导下，苹果公司的业绩一落千丈，几乎破产。

③ 1998年，微软公司发布Windows 98操作系统，其中捆绑了IE浏览器。这遭到了美国司法部起诉，原因是"滥用垄断"（abusive monopoly）。2000年4月3日，微软公司一审败诉，被判一分为二。这个判决后来被联邦上诉法院部分推翻。2001年，微软公司与美国司法部达成和解。

④ 鲍尔默2000年1月被任命为微软公司的CEO。1998年，他被任命为总裁，比尔·盖茨自己担任董事会主席和CEO。

⑤ 该书由HarperCollins出版社在1995年出版，是一本书信体小说，讲述Windows 95发布前的计算机行业的状况。

直到我在《华尔街日报》上看到一篇不长的比尔·盖茨专访，他在谈到招募优秀员工是多么困难时，顺带举了一个例子，说比如一个优秀的Excel软件经理，他们不会自动从树上长出来，诸如此类的话。

他会不会在说我？不会，可能是在说其他人吧。

都过去了。

程序员最可信赖的求职帮手

2

寻找优秀的程序员

2006年9月6日，星期三

优秀的程序员都在哪里

这是你第一次公开招募雇员。如同大多数人一样，你会发布广告，可能也会浏览一些大型的网上论坛，然后你就收到了一吨的简历。

一份份看下去，你会想："嗯嗯嗯，这人应该可以。"或者："这人差远了。"或者："我要知道他能不能下决心搬到布法罗[①]来。"但是，我保证有一件事绝对不会发生，那就是你对自己说："哇，这家伙太聪明了！这种人，我们一定要得到！"事实上，当你看完足足几千份求职简历之后（假定你懂得如何看简历，那可并不容易），老实说，你从中没有发现一个优秀的程序员。一个也没有。

下面我就来说说为什么会这样。

很简单，就同所有行业中最好的人才一样，那些优秀的程序员是不会出现在招聘市场上的。

通常优秀的程序员在整个职业生涯中，可能会有4次求职。

那些最优秀的大学毕业生，他们会从教授那里得到实习的机会，而教授

① 布法罗（Buffalo），美国纽约州西部伊利湖东岸的港口城市，西与加拿大隔河相望，城外有世界著名的尼亚加拉大瀑布。华人常把这座城市叫做水牛城。

跟业界有不少联系。这样，他们会早早地就从实习公司得到机会，根本不用去找其他工作。如果他们离开那家公司，那可能是因为同朋友一起去创业，或者因为他们跟着一个了不起的老板一起跳槽到另一家公司，或者因为他们决定一定要换个工作方向。比如说Eclipse[①]，因为Eclipse很酷，所以他们想去BEA[②]或者IBM找一个Eclipse的工作，然后他们肯定会得到这份工作，因为他们是优秀人才。

如果某一天，你遇到了这样的人出现在招聘市场上，那么你很幸运，你真地非常幸运。可能的情况是，他们的配偶决定到安克雷奇[③]当一个实习医生，他们就会发出简历，给少数几个他们认为自己愿意在里面工作的位于安克雷奇的公司。

但是大多数时候，优秀的程序员（我几乎是在重复了）是那么优秀（对，我就是在重复），未来的雇主通常会一眼看出他们的优秀，这意味着，这些程序员基本上想去哪里工作，就能去哪里工作。所以，老实说，他们不会发出许多份简历，到处找工作。

听起来，他们就是你想雇的那种人？当然。

这条规律（优秀的人才从不在市场上求职）有一个推论，那就是在人才市场上找工作的，大部分都是一些水平很差、完全达不到要求的人。他们一年到头都在被解雇，因为他们不能完成工作。他们所在的公司也会完蛋，因为这些人水平太糟糕，以致于整个公司都会被他们拖垮。是的，这种事真地会发生。（公司完蛋的另一个可能的原因是，既然雇用了一个不合格的程序员，就可能雇用一大堆不合格的程序员，累积起来，就导致了最终的失败。）

谢天谢地，这些那么糟糕的人很少能够求职成功，但是，他们总是不断地发出求职信。他们找工作的时候，就去Monster.com[④]，将所有的职位翻看一遍，300个或者1000个，试图中奖。

从数量上说，优秀的人才很少，而且从不出现在招聘市场上，而那些不

① Eclipse是一个由Java语言编写的开源集成开发环境（IDE），由IBM公司在2004年发布。

② BEA系统有限公司（BEA Systems, Inc.），成立于1995年的一家美国软件公司，已于2008年1月被甲骨文公司（Oracle）以大约72亿美元的价格收购。BEA是Eclipse的重要开发者和支持者，具有领导地位。

③ 安克雷奇（Anchorage），美国阿拉斯加州最大的城市，属于美国的边远地区。

④ Monster.com，北美最大的求职网站之一。

称职的人，即使数量也同样少，却在整个职业生涯要申请几千份工作。所以，老兄，现在让我们回到你从Craigslist[①]上搞到的一大堆简历上来。你对他们中的大多数都看不上眼，有什么好奇怪的？

我猜想，聪明的读者读到这里，会指出我遗漏了最大的一类人：那些可靠的、称职的程序员。人才市场上这一类人在数量上多于优秀程序员，但是少于不称职的程序员。不管怎样，总的来说，在你收到的1000份简历中，他们的比例是很小的。在绝大多数情况下，几乎所有此刻桌子上堆着1000份简历的Palo Alto[②]人事经理们都会发现，其中970份简历是出自同样的那少数970个不称职的程序员，内容都一样，这些家伙针对每一份工作都发求职信，也许会这样干上一辈子。1000份简历中仅仅只有30份是值得考虑的，其中可能偶然会出现一份优秀程序员的简历。OK，可能一份也没有。请想一想，如何在一堆干草中找到一根针。我们下面将会看到，这是可以做到的，但是比较难。

我能得到他们吗

你能！

好吧，你也许能！

实际上，这要看情况而定！

请不要将招聘看作一个“收集简历，过滤简历”的过程，你必须将它看作一个“追踪优胜者，设法结识他们”的过程。

我有3个实现这个目的的基本方法。

(1) 走出去。
(2) 实习生。
(3) 建立自己的社区（community）。*

（“建立自己的社区”这一条后面有一个星号。它的意思是，这一条有难

① craigslist.org，美国最大的分类广告网站，允许用户自由发布各种各样的广告。
② Palo Alto（帕洛阿尔托），美国加州的一个城市，共有人口6.2万，位于旧金山湾区南部。著名的施乐公司PARC研究中心和斯坦福大学就位于这里，它被认为是硅谷的中心，也是HP公司的诞生地。

度，难得就好像数学家乔治·丹茨格[①]解决的那道数学难题一样。他之所以能解决是因为，他上课迟到了，以为黑板上的题是课外作业，而没有听到老师说那道题是无法解决的难题。）

在这方面，你可能有自己的想法。我只是想谈一谈这3个对我自己有效的方法。

走出去，伙计

设想一下那些你想雇用的人会出现在什么地方。他们会去参加哪些会议？他们住在哪个地区？他们属于什么组织？他们上什么网站？不要像撒大网一样在Monster.com上搜索求职者，你可以浏览“Joel谈软件”网站上的求职讨论区，将搜索范围缩小到那些阅读我的网站的聪明人。你还可以去那些真正有趣的技术会议上寻找合适的人。优秀的Mac程序员会参加苹果公司的WWDC大会[②]。优秀的Windows程序员会参加微软的PDC大会[③]。此外，开源软件也有许多会议。

你要关心正流行的热门新技术。去年是Python语言，今年轮到了Ruby语言。你去参加它们的会议，在那里你会找到这些技术的早期接受者，那些人对新事物充满好奇心，而且永远对如何进一步改进有兴趣。

你要在走廊里到处走走，同遇到的每一个人都攀谈一番；去参加技术环节的小组专题讨论，将发言者邀请出来，一起喝杯啤酒。当你终于发现聪明人的时候，立刻进入全力套近乎和吹捧模式。“啊哈哈哈哈，那真是太有趣了！”你说，“哦，我真不敢相信你这么聪明。还这么帅！你刚才说你在哪里工作？真的吗？那个地方？哎呀呀呀呀。你会不会觉得，你能做出更大的成绩？我想我的公司可以雇……”

① 乔治·丹茨格（George Dantzig，1914—2005），美国数学家，线性规划（linear programming）的创始人。

② WWDC是苹果公司全球开发者大会（Apple Worldwide Developers Conference）的缩写，每年在美国加州举行，用来发布苹果公司的新产品和新技术。最近一次的WWDC 2009于6月在旧金山举行。

③ PDC是微软公司举办的专业开发者大会（Professional Developers Conference）的缩写，面向Windows程序开发者。该会议只在微软公司有新产品和新技术发布的年份举行。下一届将于2009年11月在美国洛杉矶举行。

这种方法的引申含义就是，不要在大型的求职论坛发布没有针对性的招聘广告。有一年夏天，我一不小心在MonsterTRAK[①]上发布了一个暑期实习的招聘广告。只要出很少的钱，MonsterTRAK就能使你的招聘广告有机会被位于美国各地的所有学校的学生看到。结果就是我收到了几百份简历，但是其中没有一份能够通过第一轮筛选。我们花了钱，却得到了一堆几乎毫无机会被我们雇用的人的简历。一连好几天，MonsterTRAK 源源不断地送来简历，这样的事实使我觉得我们大概不会找到想要的人了。同样的，当Craigslist刚成立的时候，用户都是互联网产业的业内人士，我们通过在上面发招聘广告，找到了优秀人才。但是今天，会用计算机的人差不多都在上Craigslist，结果就产生了太多的简历，在其中发现人才简直比在一堆干草中发现一根针的概率还要低。

实习生

抢到那些永远不会在招聘市场上出现的优秀人才，有一个好办法。那就是当他们还在学校里的时候就出手，那个时候他们甚至还没有意识到世界上有人才市场这件事情。

一些人事经理非常反对雇用实习生。他们认为实习生不成熟、水平不够。一定程度上确实如此。实习生不如资深雇员有经验。（且慢，真的如此？！）你不得不在他们身上多付出一些，这需要时间，然后他们才会全速前进。对于我们这个行业来说，好消息是，一些真正优秀的程序员往往在10岁的时候就开始编程。当其他同龄的孩子正在玩足球（这是一种许多不能编程的孩子喜欢玩的游戏，就是用脚去踢一个叫做球的圆形物体——这样说听起来很怪，我知道的），他们却在爸爸的书房里试着编译Linux的内核（kernel）。他们不去操场上追求女生，而是在Usenet[②]上大打口水战，宣称某些编程语言糟糕透顶，没有实现Haskell[③]风格的类型推断（type inference）。他们不在车库里组建乐队，而是动手完成一个很酷的黑客程序，对付那些企图偷用别人

① monstertrak.com是一个主要针对大学生找工作的门户网站。
② Usenet，一种全球性的供用户交换文件和讨论问题的大型网络。
③ Haskell是一种纯函数的编程语言，为纪念逻辑学家Haskell Curry而以他的名字命名。

Wi-Fi带宽的邻居，当那些人一打开网页，所有里面的图片都会倒过来。

所以，在软件开发这一块，同其他行业（比如法律或医学）不一样，当那些孩子进入大二或大三的时候，他们已经是好得不得了的程序员了。

天底下有一个工作，几乎所有人都会投求职信，那就是每个人的第一份工作。大多数孩子都觉得，犯不着为这种事担心，到大学最后一年再说吧。事实上，大多数孩子都不会太主动，他们要等到校园招聘开始以后，才会觉得应该要投简历了。在第一流大学读书的孩子们，单单从校园招聘中就会有足够的好工作可以挑选，他们很少会再去考虑那些懒得来学校的雇主。

你可以赶去参加校园招聘，虽然乱哄哄的，但是别误会，这是一件好事。你也可以在孩子们毕业的一二年前，就把事情搞定。

在这方面，我自己的Fog Creek软件公司有许多成功的例子。每年9月份，我就开始行动，我动用所有的资源追踪这个国家最好的计算机专业的学生。我给全国二三百所学校的计算机系写信。在离毕业还有两年的时候，我就搞来主修计算机科学的学生名单（为了搞到名单，你通常需要在系里有认识的人，老师或者学生）。然后，我就给我找到的每一个学生写一封信。不是电子邮件，是真的纸质的信，上面有Fog Creek软件公司的抬头。我还用墨水笔，醒目地签上自己的大名。很显然，如今这种事情不多见，所以我的信可以得到足够的关注。我在信里跟学生们说，我们公司有实习的机会，我以个人名义邀请他们来申请。另一种情况是，我给计算机系的教授和校友发电子邮件，请他们将我的邀请信转发到计算机系的邮件列表中去。

慢慢地，我们就会收到许多实习职位的申请信，我们就从中挑出最后的入选者。在最近几年中，我们每个实习职位都有200个人申请。我们通常从这么多申请表格中筛选出10份（每个职位），然后给这些人打电话考察。在通过电话考察的人中，我们可能会邀请2到3个飞到纽约，对他们进行面试。

到了面试这一步，面试者就很有可能被我们雇用，所以现在是时候启动全场紧逼式的招募了。豪华轿车在机场等着他们，穿制服的司机帮他们提行李，并且将他们送到酒店，那里很可能是他们迄今见过的最酷的酒店，坐落在城中最时尚、随时都能看到模特在街上走来走去的地区，卫生间里有各种

复杂的设备，简直可以放进当代艺术博物馆中当作藏品（祝你好运，希望你能在这样的卫生间里弄清楚怎么刷牙）。当面试者住进酒店以后，我们给他留了一个大礼包，里面有一件T恤、一本由Fog Creek员工编写的纽约观光手册、一张包含2005年暑期实习全过程的纪录片的DVD。屋里有一个DVD播放机，所以他们当中许多人都看到了上一年的实习是多么有趣。

一天的面试结束以后，如果他们想看看纽约，我们出钱让他们再待上二天。最后，用豪华轿车接他们离开酒店，送到机场，搭上返家的航班。

即使到纽约参加面试的人中，三个里面只有一个能够通过全部的考核，但是让那些最后通过的人对我们有一个正面的印象，这是非常重要的。就算那些没有通过的人，也会认为我们是一流的雇主。当回到学校时，他们就会告诉所有的朋友，这次住在豪华酒店的纽约之行是多么好玩。他们的朋友听了，就会在明年夏天申请我们的实习生，哪怕只是为了有机会到纽约来玩一次。

就暑期实习本身来说，在这个过程中，学生们通常会想："OK，这次暑期实习还不错，我得到了一段美好的经历。也许，只是也许，我可以获得一份全职工作。"我们在学生前面就已经想到了这个问题。在暑期中，我们会决定想要哪些人留下来作为全职雇员，学生们也要利用暑期决定是否愿意加入我们。

正是因为如此，我们会把真正的实际工作交给实习生。那些都不是轻松的活。我们的实习生总是接触会在最后产品中使用的代码。有时候，他们做的是整个公司里最新潮的东西。这会让我们的正式雇员有一点点嫉妒，但是这就是生活啊。有一年夏天，我们用4名实习生组成一个开发小组，从零开始做一个全新的产品。他们用几个月的时间就做成功了。公司获得的收益完全超过了实习的开销。实习生做的即使不是新产品，也总是真实的、会投入生产阶段的代码，并且他们个人要对软件功能中的某个主要方面承担完全责任（当然，资深雇员会作为导师帮助他们）。

为确保他们的实习生活过得愉快，我们会举办晚会，邀请他们到家里做客，免费为他们提供条件很好的宿舍，在那里他们可以结识来自其他公司和学校的朋友。我们每周还有一些课外活动或者郊游，比如听百老汇的音乐剧

（今年的实习生都迷上了音乐剧《可爱大道》[①]）、参加电影首映式、参观博物馆、围绕曼哈顿岛划船、观看扬基棒球队的比赛。不管你信不信，本年度最受欢迎的活动之一是登上落基山顶。我的意思是，爬到曼哈顿中区一幢高楼的顶楼平台。你大概不会觉得这种活动好玩，但是它真的受欢迎。除了实习生以外，Fog Creek的一些正式员工也参加了每次活动。

当夏天结束的时候，总是有一些实习生让我们确信，他们就是那类我们非雇用不可的真正的优秀程序员。不是所有的实习生都是这一类，请听好，有些实习生确实是优秀程序员，但是我们愿意让他们离开，还有一些实习生可能在其他地方会成为优秀程序员，但是不是在Fog Creek。举例来说，我们是一家很强调自治（autonomous）的公司，没有很复杂的管理层级，我们希望所有人能够主动工作。从过去来看，有些时候，如果有人督导，一些暑期实习生就表现得非常好，但是一旦他们来到Fog Creek这种没有很多硬性管理的环境，他们的表现就不太好。

无论如何，对于那些我们确定要雇用的人，等待是没有意义的。我们会很快就提供一个正式职位，条件是他们毕业以后过来工作。这个职位的待遇很丰厚。我们就是要他们回学校后，跟朋友们交换意见，然后他们会意识到自己的薪水比别人都要高。

这是不是意味我们给的报酬太多了？完全不是。你要这样看，第一年的薪水一般来说总是要打一点折扣的，因为必须考虑到新人没有好的表现的风险。但是我们已经考察过这些孩子了，可以毫无风险地认定他们就是优秀人才。我们很清楚他们的能力。所以，当我们雇用他们的时候，我们比其他雇主有更多的信息，后者仅仅面试过他们而已。这意味着我们能够付更高的报酬。我们有更充分的信息，所以我们愿意比那些没有这些信息的雇主付出更高的工资。

如果我们做好了份内的事情（我们通常都能做好），那么到了这个时候，我们选定的实习生就会做出决定，到底接受还是放弃我们提供的职位。不过有时，还需要一点额外的说服工作。有些实习生不愿意马上给出明确答复，

① 《可爱大道》（Avenue Q）是一部讲述年轻人生活、事业和爱情的美国音乐剧，2003年首演，并且获得了当年美国戏剧最高奖托尼奖。Avenue Q是纽约的一条街道（现实生活中纽约从Avenue A~Z都有，唯独缺少Avenue Q），因为这里的房租便宜，所以住着各种没什么钱的年轻人。

程序员最可信赖的求职帮手

他们还想等一等，看看有没有更好的机会。但是，一个来自Fog Creek公司的有效的工作机会，会起到这样一种作用，那就是当他们第一次在早晨8点不得不起床、穿上套装、去参加甲骨文公司的面试的时候，当闹钟响起的那一刻，他们很可能会说："我干嘛一定要在该死的早晨8点起床、穿上套装、去参加甲骨文公司的面试呢？不是已经有一个很好的工作机会在Fog Creek等着我吗？"我的期望是，他们会觉得太麻烦，就不去参加那个面试了。

在我继续往下讲之前，顺便说一句，我必须对计算机科学和软件开发行业的实习生问题做一点澄清。在如今这个时代，在这个国家，对实习生支付报酬被认为是理所当然的，而且他们的报酬往往很不错。虽然不付报酬的实习生在很多其他行业依然很常见，比如出版业和音乐业，但是我们的工资标准是每星期750美元，还提供免费的住宿、免费的午餐、免费的地铁交通费，更别说来回纽约的飞机票和其他各种福利了。如果单看钱数，我们的工资比平均水平要低一点，但是如果将免费住宿考虑在内，我们的工资就比平均水平要高一点。我想之所以我要说这个，是因为每次我在我的网站上谈到实习生的时候，总有人搞不清楚，认为我占了别人的便宜，好像我在搞奴隶制似的。那边的谁——你们这些自以为是的毛孩子！给我拿一杯冰冻橙汁，要手挤的，别磨磨蹭蹭！

实习生制度创造了输送优秀人才的管道，但是这根管道比较长，而且一路上的损耗很大。根据我们的计算，基本上每雇用一个全职雇员，我们就必须请两个实习生，以便挑选。如果你请的实习生离毕业还有一年，那么从你开始确定实习人选的时候算起，到他正式第一天上班，会有两年的时间间隔。这意味着，每年夏天我们的办公室有多少空位置，我们就要请多少实习生来。头三个夏天，我们试着将实习生限制为离毕业还有一年的学生。但是今年夏天，我们终于意识到错过了一些更年轻的优秀学生。所以，我们将实习生计划向高校中所有年级的学生开放。说出来你可能不信，我正在琢磨怎样将高中生弄进来，也许让他们课后组装电脑，为上大学攒一点钱。我们的目的就是开始着手建立与下一代优秀程序员之间的联系，即使这样一来，管道会长达6年。没关系，我有长远眼光。

建立社区*

这里的基本思想是创造一个大型的网络社区，让观点相近的优秀软件开发者以某种形式聚集在你的公司的周围。然后，每当你有职位空缺的时候，总会有天然的候选者。

告诉你实话，Fog Creek里这么多优秀的程序员，就是通过我的网站joelonsoftware.com发现的。那个网站上主要的文章的读者多达100万，其中大部分都是有能力的程序员。有了这么庞大的、自发聚集起来的读者群，无论何时，只要我在主页上提到我正在招聘人，通常就会得到很多优秀人才的简历。

我用一个星号表示这种方法很“困难”，因为我感到，好像我给你的是这样一个建议：“要赢得选美比赛，第一步是要变美，第二步是参加比赛。”我有这种感觉是因为，我真不太清楚为什么我的网站变得这么流行，或者为什么浏览这个网站的人都是一流的程序员，我不知道自己做了什么。

我真心希望我能在这个地方给你更多帮助，但很抱歉我没做到。在这方面，Derek Powazek写过一本好书*Design for Community: The Art of Connecting Real People in Virtual Places*。许多公司尝试过不同的网志策略，但是很不幸，其中的大多数最终都不会有任何读者群。所以我只能说，对我们有效的方法，对你可能有效，也可能无效。我不确定你到底该怎么做。我在自己的网站上搭建了一个求职区（jobs.joelonsoftware.com）。你只要出350美元就可以在那里发布招聘广告，该网站的所有读者都会看到。

员工推荐：小心陷阱

寻找优秀程序员有一个所谓标准建议，就是询问你现在的雇员。怎么说呢，它背后的理论是，聪明的程序员一定认识其他聪明的程序员。

他们可能确实认识，但是他们的亲密朋友中也会有人不是非常优秀的程

序员。这种方法会制造大概100万颗地雷。所以，真实生活中，我一般认为员工推荐是最不可靠的招聘新人的方法之一。

一个无法回避的巨大风险，就是竞业限制合同[①]（noncompete agreement）。如果你不以为然，请想一想Crossgain公司的遭遇，这家公司不得不开除了四分之一员工，因为那些人以前都在微软工作，微软公司威胁起诉他们。在正常理智的情况下，没有程序员会签署一份竞业限制协议，但是大多数程序员确确实实签署了，因为他们觉得这种合同不可能会实施，或者因为他们没有习惯去阅读合同里写了什么，或者因为他们已经接受了雇主提供的职位，穿过半个美国，举家搬到新的城市，在上班的第一天第一次看到了合同的全文，这时再要协商具体条款为时已晚。不管什么原因，总之他们签署了。但是实际上，这是雇主让人最讨厌的做法之一，而且这个条款可以强制执行，并且真的会被执行。

如果你不当心竞业限制合同，并且很依赖员工推荐，后果可能是，你雇了一堆人，他们的前雇主都是同一家公司（因为就是在那个地方，你的雇员第一次结识了其他明星程序员），那么你将承担非常巨大的风险。

另一个问题是，如果你不能保证应聘者百分之百被接受，那么当你要求你的雇员推荐人选时，他们甚至都不会考虑向你说出他们真正的朋友。如果自己推荐的朋友还可能被拒绝，就没有人愿意说服朋友来应聘自己的公司。这样多多少少会破坏友谊。

因为他们不告诉你谁是他们的朋友，你可能就不能雇到以前同他们一起工作的人。那么，剩下来的他们推荐的人，可能就不是很有能力了。

但是，“员工推荐”的真正问题在于，如果负责招聘的人事经理懂一点经济学，决定为每一个有效的推荐提供奖金，麻烦就来了。提供奖金的做法很常见，原因是这样的：如果通过猎头公司或者外部招聘渠道雇人，每个职位的中介费用大概是3万美元到5万美元。如果我们向“员工推荐”支付奖金，比方说，每次成功推荐一个人，他们可以拿到5000美元的奖金，或者成功推荐10个人，他们可以得到一辆昂贵的跑车，或者其他的奖励方式，那样的话，想一想可以省下来多少钱。即使对于一个工资丰厚的雇员来说，5000美元听

① 竞业限制合同，又称竞业限制条款，指雇用合同中规定，雇员在离职后一段时间内，不得到与原雇主有竞争性业务的公司就业。

上去也像是一笔小财，的确是笔不小的收入。总之，这样处理看上去是一种双赢的、对各方面都很周到的方案。

麻烦是，突然之间，你发现每个人的心眼都活了，雇员们开始将他们能想到的每个人都拖来面试，他们强烈希望他们找来的人能被雇用。他们教外面的人如何应付面试，在会议室中同应试者说悄悄话。转眼之间，你手下的整个劳动大军都在设法让你雇用某人的一个大学室友，而此人对你毫无用处。

这样是不行的。ArsDigita公司曾经轰动一时，因为它买了一辆法拉利跑车，将车陈列在停车场上，并宣布不论是谁，只要能成功推荐10个人进公司，他就能拥有这辆跑车。无人完成这项任务，甚至连接近完成的人都没有，但是新进员工的质量却不断下降。该公司后来倒闭了，不过原因可能不是这辆法拉利，因为后来被曝光该车是租的，不过只是公开作秀而已。

如果Fog Creek的雇员提议雇用某个可能对我们很合适的人，我们愿意省去第一步的电话筛选，但是仅此而已。我们仍然要求应聘者通过剩下的所有面试，我们坚持高标准不变。

寻找优秀的程序员之实战指南

程序员会怎么找工作？如何使一份工作对程序员更有吸引力？你怎样才能变成他的雇主？请继续往下阅读！

3

寻找优秀的程序员之实战指南

2006年9月7日，星期四

你是一个雇主。你在所有正确的地方刊登了招聘广告，你有一个完善的实习生制度，你面试了所有你想要的人。但是很不幸，如果优秀的程序员不愿意为你工作，你就无法把他们拉来工作。不要着急，现在就介绍如何使优秀程序员愿意为你工作。我将谈谈优秀的程序员想要什么，还有在工作环境中他们喜欢什么和不喜欢什么，以及如何使你的公司成为顶尖程序员的头号选择。

私人办公室

去年，我到耶鲁大学参加一个计算机会议。一位发言者是硅谷老手，创建和领导过一长串公司，无一不是风险投资加盟的著名创业公司。他拿起一本名叫《人件》[1]（*Peopleware*）的书，该书作者是Tom DeMarco和Timothy Lister。

“你们一定要读这本书，”他说，“这是如何管理软件公司的圣经，是这方面目前最重要的书。”

我不得不说，我很同意《人件》是一本伟大的书。书中最重要和最具争议性的观点之一就是，如果你要让程序员高效工作，你就必须给予他们安静和宽敞的工作环境，比如说私人办公室。该书作者不厌其烦地强调这个观点。

① 该书主要内容是软件公司的内部管理和运作，被视为这一领域的经典著作。

他的发言结束以后，我走到他的面前。“我同意你对《人件》的评价，”我说，“请告诉我，在你所有的创业公司中，你的程序员有没有独立的办公室？”

“当然没有，”他说，“风险投资家永远不会这么激进。”

原来如此。

“但是，这可是那本书中头号最重要的观点。”我说。

“说是这样说，但是你必须有所侧重。对于风险投资家来说，那样做看上去有点像是乱花钱。”

大量证据表明，给予程序员独立的办公室会显著提高他们的工作效率。我在自己的网站上也不断重复这一点。但是尽管如此，硅谷的流行做法却是要求创业者将大量程序员塞进一个巨大的开放空间中。我无法真正地说服大家，虽然我觉得应该还是可以说服的，但是因为程序员多多少少喜欢和大家待在一起，即使这样会降低效率也无所谓，所以说服工作不是那么容易的。

我甚至听到过有的程序员说这样的话：“对，我们都在小隔间中工作，但是所有人都是这样，甚至包括CEO在内！”

“CEO？他真在小隔间中工作？”

“是这样的，他的确有一间小隔间。但是既然你问到了，我就都交待了吧。实际上，他还有一间大会议室，所有重要的会他都到会议室里面开……”

没错。这种大模大样的作秀在硅谷相当常见，CEO装得像平头老百姓一样，在一间小隔间里工作，但是某处还有那么一间会议室，他往往会占为己有。（虽然他解释“只有讨论不能公开的事情，才会使用会议室”，但是当你经过那间会议室，有一半的时间，你会看到你的CEO在里面，独自一人，通过电话与他的高尔夫球伙伴侃侃而谈，把他穿着Cole Haans名牌皮鞋的脚搁在会议桌上。）

但我在这里不想老生常谈了，不想讨论为什么私人办公室可以让程序员更有效率，为什么戴上耳机隔绝噪音被证明会降低程序员的工作质量，以及为什么为程序员提供私人办公室并不会真地花费特别多的钱。这些事情我以前都谈过了。今天，我要谈的是招聘，还有私人办公室对招聘的影响。

不考虑效率问题，也不考虑工作环境中的平等问题，有两件事是确凿无疑的：

(1) 私人办公室意味着更高的地位；
(2) 小隔间和其他共享的空间会带来某些尴尬和不便。

正是因为存在这样两个事实，所以基本判断是：如果提供私人办公室，程序员就有更大的可能接受你的工作邀请。如果这间办公室有一扇可以关上的门，有一扇窗，有良好的视野，就更好。

但是在现实中，很不幸，这些可以让招聘变得更容易的事情不是你的职权所能控制的。如果CEO和创始人听命于风险投资家，那么甚至连他们也无法拍板提供私人办公室。大多数公司每隔5到10年才搬一次家，或者重新安排办公空间。那些小型的创业公司也许根本就没有能力提供私人办公室。所以，我的经验是，除了那些最开明的公司，其他所有公司都会搬出一大堆的借口，使得程序员几乎毫无可能得到一间私人办公室。即使是在那些最开明的公司也有可能会出现这种情况：这些公司有一个由行政经理秘书和来自一家大型建筑事务所的初级合伙人组成的决策委员会，他们每隔10年就要开会，决定公司新的办公地点和办公条件，而且这些人更乐于相信课本上的童话，什么开放性的空间意味着开放性的公司之类的。程序员或者开发团队的意见基本上对这些人是零输入。

这种事情有点太不像话了，我将继续为了自己的信念而战。但是与此同时，私人办公室不是不存在的，我们就设法做到了这一点。在绝大部分时候，我们所有的专职程序员都有自己的私人办公室，而且是在纽约城，全世界办公楼租金最高的地方之一。毫无疑问，这使得人们更愿意在Fog Creek工作。所以，要是你们所有人还是坚持老一套的做法，随你们的便，我就让这一点成为我的竞争优势好了。

工作环境

比起私人办公室，工作环境有更多的话题可供谈论。当一个应聘者在面试的那一天来到你的公司，他们肯定想多看看里面的人是怎么工作的，并且设想他们自己在这个环境中工作会怎样。如果工作环境让人感到很愉快，有

一种生机勃勃的感觉，办公楼坐落在一个良好的社区，所有东西都是崭新干净的，那么他们就会产生一些愉快的畅想。如果工作环境很拥挤，地毯脏兮兮的，墙壁也不干净，到处贴着赛艇队比赛的图片和大幅的“团队精神”的标语，那么他们就会有一种看呆伯特[1]漫画的感觉。

许多技术人员对自己办公室的基本状况麻木得惊人。事实上，如果办公室稍作改进，有些人马上就能从中受益，但是即使如此，那些人可能也已经对自己办公室的某个明显缺点熟视无睹了，因为他们早已司空见惯。

请你站在求职者的位置，设身处地地想一想。

- 他们会怎么评价我们所在的城市？布法罗听上去同其他城市（比如奥斯汀[2]）相比怎么样？应聘者愿意搬到底特律吗？如果你本人住在布法罗或者底特律，你会不会不愿意在9月份进行大量面试？
- 当他们进入办公楼后，会有怎样的感受？他们看到了什么？是不是一个干净而且让人感到兴奋的地方？大楼有没有一个漂亮的中庭，里面有生意盎然的棕榈树和喷泉吗？还是让人觉得像一个贫民区中的公立牙科诊所，到处是枯黄的盆栽植物和过期的旧杂志？
- 办公室看上去怎么样？是不是每样东西都是崭新明亮的？或者签到的时候，你们有没有用那种一整张的、发黄的、很不正式的员工资料登记表格？那种表格是用点阵式打印机打印在折叠式的连续打印纸上的，这些东西让人觉得恍若隔世。
- 办公桌看上去感觉怎么样？程序员用的是多屏幕的液晶显示器还是一个大块头的带显像管的CRT显示器？程序员的椅子是Aerons出品的名牌电脑椅还是Staples品牌打折时的便宜货？

关于Herman Miller设计的著名的Aeron牌电脑椅，请让我在这里多说一句。这种椅子的价格是每把900美元，比办公用品连锁超市Office Depot里卖的便宜货或者Staples牌贵出800美元。

Aeron牌电脑椅比那些便宜货舒服得多。如果尺寸正确，并且调节到最合适的位置，那么大多数人坐一整天都不会感到不舒服。靠背和坐垫都被设

① 呆伯特（Dilbert）是一套由Scott Adams创作的美国漫画，内容都与办公室生活有关，主要讽刺了官僚主义和办公室政治。

② 奥斯汀（Austin）是美国南方得克萨斯州首府。

计成网状形，空气可以自由流通，所以坐着不会捂汗。这种椅子包含了第一流的人体工程学设计，尤其是那些带有腰部支撑的新型号。

它们比便宜的椅子更经用。我们公司开业已经6年了，每一把Aeron牌电脑椅差不多都还是崭新的，我赌你分不出哪些是我们在2000年买的，哪些是我们在3个月前买的。它们的使用年限可以轻松地超过10年。而那些便宜货在买了几个月后，上面的零件就开始松动了。一把Aeron牌电脑椅的寿命至少抵得上4把那种100美元的椅子。

所以结果是，买一把Aeron牌电脑椅，每10年只多出500美元，也就是每年多出50美元，相当于在每个程序员身上每星期多支出1美元。

一卷上等卫生纸的价格大约是1美元。你的程序员每人每星期大概会用掉一卷。

所以，将电脑椅升级到Aeron牌，多出的花销与你花在程序员的卫生纸上的开销大致相等。但是，我可以向你保证，如果你把卫生纸的支出拿到预算委员会上讨论，你一定会被严厉地呵斥不要捣乱，还有许多更重要的事需要讨论。

很不幸，Aeron牌电脑椅同奢侈铺张的坏名声联系在了一起，尤其是对于创业公司来说。它甚至成了某种象征，代表了在互联网泡沫时期所有被胡乱花光的风险投资。这真是可惜，因为考虑到经久耐用的寿命，它其实不是非常贵，尤其是你要想到，每天有8个小时你坐在上面。即使是带有腰部支撑和后部平衡装置的顶级系列的型号，也可以说是便宜得要命，你购买它们实际上是赚到了。

玩　具

同样的逻辑也适用于其他程序员的玩具。就是因为这个原因，所以没有理由不给你的程序员配备顶级电脑，至少两块大（21"）液晶屏幕（或者一块是30"），不给他们设置上限，让他们可以自由地在Amazon.com上订购任何他们想要的技术书籍。这些都会带来很明显的效率上的提高，但是更重要的是，对于我们在这里讨论的问题来说，它们是招募优秀程序员的杀手锏，

特别是在这样一个大多数公司都将程序员视作随时可以替换的齿轮或者打字机的世界中。真的，他们会质疑为什么程序员需要这么大的显示器，15"的CRT显示器有什么不好？他们甚至会说起当年他们小的时候……

程序员的社交生活

程序员与普通人并不是真地有那么多不同之处。当然，我很清楚，如今很流行将程序员看作埃斯柏格综合征[①]患者，行为刻板、呆头呆脑，在人际交往中表现得很不协调。但是，这不是事实，哪怕有些程序员真患有埃斯柏格综合征，他们也很关心工作场所中社交性的一面，这包括以下内容。

程序员在组织中如何被对待

他们被当作明星，还是被当作打字员？公司的管理层是不是由工程师和曾经的程序员组成？程序员外出参加会议时，坐的是不是头等舱？（我不在乎这样是否像在浪费钱。明星坐的就是头等舱。请习惯这种做法。）他们飞过来面试的时候，有没有豪华轿车在机场等着接他们？还是说，他们得自己想办法到公司？如果其他条件都相同，程序员就会选择去一个对待他们像对待明星一样的公司。如果你们公司的CEO以前是干销售的，并且什么都看不顺眼，他就很不理解那些娇滴滴的程序员，为什么一直要求发放诸如腕垫、大屏幕显示器、舒服的座椅这一类的东西，这帮人以为他们是什么？如果你们的公司是这种样子，那么就需要调整态度了。如果你不尊重程序员，你就不会得到优秀的程序员。

谁是他们的同事

面试的那一天，程序员会密切关注他们遇到的那些人。他们待人是否友善？以及更重要的，他们是否聪明？曾经有一年夏天，我在贝尔通信研究所当实习生，那是贝尔实验室的一家子公司，我遇到的每一个人都不断地、一

① 埃斯柏格综合征（Asperger syndrome）：1944年，德国儿科医生Hans Asperger发表《关于儿童自闭性人格障碍》的论文，提出某些患有自闭症的儿童，有一些特殊的症状，比如与人交流时缺乏面部表情、不断重复同一问题、社会交往中表现出自我倾向、对他人没有情感性的共鸣、走路姿势很特别或不稳定等。1981年，英国的儿童精神病学家Lorna Wing将具有上述临床特征的人格障碍命名为Asperger综合征。

遍又一遍地跟我说同一件事："在贝尔通信研究所工作，最美好的事情是这里的人。"

这表明，要是你手下的程序员中有人脾气火暴爱挑剔，如果你不能摆脱他们，至少不要让他们在面试场合出现；要是你的程序员中有人性格活跃、喜欢社交和组织集体活动，一定要让他们在面试中现身。你要不断提醒自己，当求职者回到家中时，他们必须做出决定去哪里工作，如果他们遇到的每个人都面色阴郁，他们对你的公司绝不会有一个好印象。

顺便说一句，Fog Creek软件公司原先的招聘理念是从微软公司抄袭来的，只有两点：聪明，并且能够完成工作。但是，甚至在我们开始运作公司前，我们就意识到应该再加上第三点"不收怪人"。回想起来，在微软公司的时候，是不是怪人其实不是录取新员工的必须要考虑的事。虽然，我相信微软公司在口头上一定说，与其他人融洽相处是多么多么重要，但是实际情况是他们从来不会因为某人很古怪就否决了他的录用资格。事实上，有时候，古怪反而是进入微软公司高级管理层的先决条件。虽然从公司运营的角度看，这不会造成太大的影响，但是从招聘的角度看，它确实会产生负面影响：谁愿意在一家需要忍受怪人的公司里工作呢？

独立和自主

回想1999年的时候，在创办Fog Creek软件公司之前，我辞职离开Juno软件公司，人力资源部约我进行一次很标准的离职谈话。我不知怎地就落入了陷阱，将我对公司管理上的种种不满都告诉了人事经理。虽然我很清楚这样做对我绝无好处，实际上只有坏处，但是我最终还是做了。我对Juno公司最大的不满是那种抽风式（hit-and-run）的管理风格。你们瞧，大多数时候，管理层对程序员完全不闻不问，将程序员扔在那里，让他们静悄悄地完成工作。但是，偶尔公司经理本人也会介入，追问一些极其微小的细节，坚持一定要百分之百按照他们的方式做出来，不许有任何借口。接着，他们又转向其他方面的细节问题，每个问题关注的时间都很短，短到来不及看到按照他们要求做出来的可笑结果。举例来说，我记得有两三天让人特别恼火，从我的上级经理一直到CEO，都跑来跟我说，怎么才能符合要求地填写Juno公司员工资料登记表格的日期栏。他们没有受过用户界面设计师的专门训练，又不肯花时间同我就这个问题交换意见，搞清楚为什么说在那个特殊情况下我恰好是正确的。不过这些都不重要，重要的是管理层根本不愿意屈尊同属下

讨论问题，甚至不愿意花时间听取我的论点。

基本上，如果你要雇用聪明人，你就必须让他们在工作中发挥技能。管理层可以提出建议，而且这样做是受欢迎的，但是他们必须极端小心，不能让他们的“建议”被视为命令，因为不管是什么技术问题，经理们知道的很可能不如在壕沟里干活的工人们，尤其是正如我前面说过的，你雇用的都是聪明人的话。

程序员希望自己之所以被雇用，是因为自己的技能，希望被别人当成专家那样对待，有权力在自己的专业领域中做出决定。

不搞政治

老实说，只要有两个以上的人待在一起，就会有政治。这很自然。我说“不搞政治”的真正的意思是“不搞恶性的政治”。程序员早就练出了对公正有非常良好的判断力。代码要么能运行，要么不能。坐在那里争论代码是否有问题，这是毫无意义的，因为你可以运行代码，答案自然就有了。代码的世界是非常公正的，也是非常严格有序的。许许多多的人选择编程，首要的原因就是，他们宁愿将自己的时间花在一个公平有序的地方，一个严格的能者上庸者下的地方，一个只要你是对的就能赢得任何争论的地方。

如果你要吸引程序员，你就必须去创造出这样一个环境。当一个程序员抱怨“人际关系复杂”时，他们的意思明白无误，就是指任何个人因素超过技术因素的环境。程序员在完成手头任务时，不被允许使用最合适的编程语言，而是被命令只能使用另一种特定的语言，原因仅仅是老板喜欢这种语言——没有什么比这更让人气愤的了；晋升的原因不是成果，而是人际关系——没有什么比这更让人抓狂的了；程序员被迫去做技术上落后的东西，仅仅因为上级或者得到上级支持的人坚持这样——没有什么比这更让人发火的了。

没有什么比因为技术原因赢得一场由于政治原因本来要输掉的争论更让人心满意足了。当我在微软公司刚开始工作的时候，有一个正在开发中的大型项目走入了歧途，项目的代号是MacroMan，目标是创造一种图形化的宏语言。真正的程序员遇到这种语言会很有挫折感，因为图形的特性让你真地没有办法完成循环和条件判断功能。此外，对于那些非程序员的用户，这种语言也不会有很大作用，因为我觉得那些用户不会习惯算法思维，没有办

法很快地理解MacroMan。当我说出对MacroMan的负面评价时，我的老板告诉我："火车跑起来就刹不住了。算了吧。"但是，我还是不放弃，一再地不断地争论。那时我刚走出学校，在微软公司中差不多跟谁都没有利害关系，所以，渐渐地，人们开始倾听我的核心观点，MacroMan后来终止开发了。我是谁并不重要，重要的是我是对的。非政治性的组织就应该这样，这种组织才会让程序员感到高兴。

总的来说，关注你的组织的社交动态变化，对创造一个健康的、令人愉悦的工作环境是很关键的，这样可以留住程序员和吸引程序员。

我干的是什么活

一定程度上，让程序员干有趣的活是吸引优秀程序员的最好方法之一。但是，这可能是最难改变的事情。请试想，如果你很倒霉地在为沙土行当写软件，跟石头和沙子打交道，这就是你的行业，那么你无论如何也没法装得像某些互联网创业公司一样酷，靠这个吸引程序员。

另一类程序员喜欢干的活是开发一些非常简单或者非常流行的东西，这种东西足以让他们在感恩节那一天向艾玛婶婶[①]解释清楚。而艾玛婶婶当然对沙土行当中的Ruby编程并不怎么懂，因为她是学核物理的。

最后，许多程序员也会关注他们服务的公司的社会价值。在社会化网络公司（social networking）和网志公司工作，可以帮助人们交流沟通，看上去也不会造成污染，所以这种公司受欢迎。军火公司和那些不道德、充斥着会计欺诈的公司就非常不受欢迎。

很不幸，在这方面，我真不确定我有办法给那些一般的招聘经理出主意。你可以尝试改变一下产品线，制作某些很"酷"的东西，但是这样不会长期有效。不过，我看过一些公司在这方面采取的举措。

让一流的新员工挑选他们自己的项目

多年以来，甲骨文集团有一个叫做MAP的计划，也就是"多选择性计划"

① 艾玛婶婶是美国情景喜剧IT Crowd中的角色。该剧讲述的是一家大公司IT部门员工令人啼笑皆非的日常生活。——编者注

（Multiple Alternatives Program）。这个计划针对各个班级中甲骨文认为最好的高校毕业生。计划中的安排是让他们来甲骨文，花上一到两个星期到处看看，访问所有缺人的开发小组，然后让他们选一个自己想进去工作的小组。

虽然也许对这个计划的效果我没有甲骨文的人清楚，但是我觉得这是一个好的计划。

使用非必要的热门新技术

纽约的那些大型投资银行被认为是相当艰苦的程序员工作环境。那里的工作条件很可怕，大量的连续加班，嘈杂的环境，咆哮的上司。程序员是千真万确的三等公民。而与此同时，一群狂热的类人猿在那里操盘买卖金融工具。这群类人猿是公司里的皇室，拿着高达3千万美元的分红，公司里所有的汉堡包他们都可以吃（经常是让碰巧在旁边的程序员递给他们）。不管怎么说，这些都是陈规陋习，所以为了留住最好的程序员，投资银行有两个策略：一个是给程序员发一吨的钞票，另一个是给予程序员完全的自由，允许他们使用自己想学的任何最新热门编程语言，不断地一遍又一遍重写每件东西。想把整个交易程序用Lisp语言重写？随你的便。帮我再拿一个该死的汉堡包过来。

一些程序员固执于他们正在使用的编程语言，但是，大多数程序员很高兴有机会使用令人激动的新技术。现在的热门大概是Python语言或者Ruby on Rails，三年前是C#，再以前是Java。

在这里，我不是让你不要用最好的工具完成工作，我也不是让你每两年就用热门语言重写一遍程序，我只是在说，如果你能找到办法让程序员有接触新的语言、框架和技术的经历，那么他们会感到更开心一些。即使你不敢为了学习的目的用一种新语言重写核心程序，那么有没有可能重写你们使用的内部工具，或者其他不关键的新应用程序呢？

我能够认同公司吗

大多数程序员工作不是为了谋生，他们要的不是一份“朝九晚五”的工作，他们要的是工作所能带给他们的意义。他们想要认同他们的公司。年轻

的程序员尤其会被有理想有抱负的公司所吸引。许多公司与开源运动或者自由软件运动（两者不是一回事）都有一些联系，这使得它们能够吸引那些具有理想主义倾向的程序员。另外一些公司与非营利性的社会事业有关系，或者制造的产品被视为和用于造福社会。

作为一个负责招聘的人，你要做的是找出你的公司中理想主义的一面，确保招聘对象了解它们。

一些公司甚至努力在创造它们自己的理想主义运动。芝加哥的创业公司37signals就强烈地认同简单的东西，所以他们开发像Backpack[①]那样简单、容易使用的应用程序，以及像Ruby on Rails那样简单、容易使用的开发框架。

对于37signals来说，简单已经成为了一种主义，实际上是一种国际政治运动。简单不仅仅是字面上的含义，哦，不，它是夏日的时光，它是优美的音乐，它是和平，它是公正，它是幸福，它是头发上插着花的漂亮姑娘。Rails开发框架的创造者David Heinemeier Hansson说，他们的故事是“一个关于美、幸福和激励的故事。从你的工作和你的工具中享受乐趣，并为它们感到自豪。这个故事并不仅仅是一种时尚，而是一种趋势。这个故事使得像激情和热诚这样的词不用找借口就能成为程序员自己所认同的词汇。你再不用为喜欢自己的工作而感到尴尬了”（www.loudthinking.com/arc/2006_08.html）。将一个互联网编程框架上升到某种“美、幸福和激励”，可能看起来有点像说大话，但是这确实非常有感染力，的确使得他们的公司与众不同。他们把Ruby on Rails说成是一种幸福，并且向外推广这种观念，这实际上保证一定会有某些外部的程序员想来找Ruby on Rails方面的工作。

但是，在这种自我认同的管理方法（identity management）潮流中，37signals资历尚浅。如果比起苹果公司，他们连在旁边（为明星）举蜡烛的资格都没有。1984年的美式橄榄球超级碗决赛时，苹果公司播出了一支广告。[②]从那时起一直到今天，它一直在加固自己反对传统文化的形象：追求自由，反抗独裁；追求自我，反抗压迫；追求色彩，反抗单调。就像广告里的内容一样，苹果公司是一个穿着明亮的红色运动短裤的漂亮姑娘，奔跑着

① Backpack是一个互联网个人信息管理系统，网址是http://www.backpackit.com/。

② 这支广告可以在http://www.youtube.com/watch?v=OYecfV3ubP8看到。另外，“1984”在这里同时指的是英国作家乔治·奥威尔的著名小说《1984》，所以后面会说这里面有“奥威尔式”的反讽。

穿过身着制服被洗过脑的人群。但是，我不得不说这里面的含义其实是奥威尔式的反讽。巨型公司用一种不合理的方式操纵它们的公众形象——嗯，比方说，他们是一家计算机公司，那么与反抗独裁有什么关系呢？真是活见鬼——成功地创造出一种自我认同的文化，使得全世界各地购买计算机的用户感觉他们买的并不仅仅是一台计算机，觉得自己通过购买而参加到了一场运动中。当你购买一台iPod时，你当然是在支持甘地[①]反抗大英帝国的殖民主义统治。每一台被卖出的MacBook都表达了一种反抗独裁和饥饿的立场！

好了，不说了，深呼吸……这一部分的真正用意是，思考你的公司代表了怎样的追求，这种追求是怎么形成的，又是怎么才能被别人感受到。管理好你的公司的品牌不仅对营销很重要，对招聘新人也同样重要。

程序员不在乎的一件事

他们实际上不在乎钱，除非你在其他事情上搞砸了。如果你开始听到有人在抱怨薪水，而以前并没有出现这种情况，这经常就是一种信号，表明人们并不真地喜欢他们的工作。如果你想雇的新人提出高得离奇的薪水要求，并且不愿意降低，那么你可能遇到的是这样一种情况，那些人心里想："好吧，如果不得不接受这份糟糕透顶的工作，那么我至少应该有一份优厚的报酬。"

我们说程序员不在乎钱，并不意味你可以向他们支付低工资。因为程序员对公正公平是在乎的，如果他们发现同工不同酬，或者他们发现自己公司每个人的薪水都比街对面同样的公司低20%，他们会被激怒的，然后，突然之间，钱就将成为一个大问题。你必须支付有竞争力的报酬，但是让我们这样说，当程序员决定去哪里工作时，在他们考虑的所有因素中，报酬的位置低得让人吃惊，前提是薪水必须基本合理。同样让人吃惊的是，如果你的公司里有这样的问题，比如程序员用的是15"显示器，整天都有销售人员对着他们吼，他们的工作是猎杀小海豹、制造核武器等，那么向他们提供高薪水并不是一个有效的工具，并不足以克服这些问题，吸引他们加入你的公司。

① 甘地（1869－1948），印度独立运动领导人，带领印度人民反抗英国的殖民统治，建立独立的国家，被视为印度的"国父"。

4

三种管理方法

2006年8月7日，星期一

如果你要领导一个团队，或者一家公司，或者一支军队，或者一个国家，那么你面对的主要问题是如何“使得人们去做你要他们做的事”，更文雅的说法是如何使得所有人都向同一个方向前进。

让我们从下面的角度思考这个问题。一旦你的团队中有不止一个成员，那么你马上就得面对各种各样的人，每个人心里都有自己的一把小算盘。他们想要的东西不同于你想要的东西。如果你是一家创业公司的创始人，你想要的东西可能就是赶快挣到一大笔钱，早早地退休，将剩下的几十年余生都用来参加与女性网志作者有关的会议。因此，在风险投资大街Sand Hill Road[①]上开车跑来跑去以及同风险投资家交谈这些事大概会花掉你的大部分时间，那些风险投资家可能会买下你的公司，然后再转手倒给雅虎公司。但是，在你的雇员之中有一个叫做贾尼丝的程序员，她不关心公司是否能够被雅虎收购，因为这不会让她多赚任何钱。她关心的是用最新最酷的编程语言写代码，因为学习一种新东西很有趣。与此同时，你的首席财务官（CFO）脑子里想的都是怎样才能搬出共享的小隔间而不用跟系统管理员待在一起办公，那个家伙是电视剧《星际迷航》（Star Trek）的超级发烧友。所以，你的CFO忙于炮制一份新的预算提案，表明如果搬到更大的办公空间你可以省下多少钱，而这个新地点离他的家只有2分钟路程，真是巧合啊！

让人们朝着你的方向前进（或者，至少让他们保持向相同的方向前进），

① Sand Hill Road，美国加州的一条公路，靠近斯坦福大学和硅谷，以聚集大量风险投资公司而闻名。

这个问题当然并非创业公司独有。当一个政治家承诺消除政府中的浪费、腐败和欺骗并因此当选后，这个问题也是他面对的一个主要的根本性问题。一方面，市长希望确保新的建筑计划能够轻易地得到批准。另一方面，市政府中主管建筑的官员则希望还能不断得到贿赂，他们已经变得习惯灰色收入了。

军队指挥官也面临同样的问题。他们要求一队士兵向敌人发起冲锋，但是所有的士兵都宁愿畏缩在大石头后面，让其他人去冲锋。

这里有三个你或许会采用的一般性方法：

- 军事化管理法
- 经济利益驱动法
- 认同法

在现实中，你肯定会发现其他管理方法（比如奇特的“时尚女魔头[①]”法、个人崇拜法、无定规法），但是在接下来的三章中，我将只详细讨论上面这三种流行方法，研究它们各自的优缺点。

下面介绍本系列中的军事化管理法。

① “时尚女魔头”（Devil Wears Prada）是一部2006年上映的美国电影，根据同名小说改编，内容是关于一个女大学毕业生在一家纽约时尚杂志的实习经历。

5

军事化管理法

2006年8月8日，星期二

士兵应该害怕他们的长官，甚于害怕任何他们将要面对的危险……亲密永远不会让普通士兵面对危险时挺身而出，只有恐惧才会让他这样做。

——腓特烈大帝[①]

命令和控制式的管理源于军事管理。大致上这种管理方法的思想是，人们只做你告诉他们去做的事情。如果他们没有做，你就对着他们吼，直到他们做了为止。如果他们还是不做，你就关他们的禁闭。要是他们依然没有吸取教训，你就让他们去潜艇里负责削洋葱，住在空间不到一平米的双人房内，并且室友是一个从来不知道刷牙的楞小伙。

这方面有100万条可以使用的伟大的技巧。看看电影《天才大兵》[②]（Biloxi Blues）和《军官与绅士》[③]（An Officer and a Gentleman），你会有所收获的。

一些经理用这种管理方法，因为他们就是在军队里学到了这一套的。另一些经理在专制家庭或者专制国家中长大，认为让手下人听话，这是很自然的方法。还有一些经理对这种方法根本没有深入的了解：嗨，既然它能用来管军队，就能用来管理互联网创业公司！

---

① 腓特烈大帝（1712—1786，Frederick the Great），即腓特烈二世，普鲁士国王（1740～1786年在位）。统治普鲁士时期，他大规模发展军事，扩张领土，赞助文化艺术活动，使普鲁士在德意志民族中取得霸权。腓特烈二世是欧洲历史上最伟大的军事将领之一。

② 《天才大兵》，1988年的美国电影，讲述一个高中毕业生如何在新兵营中成长为男子汉的故事。

③ 《军官与绅士》，1982年的美国电影，由理查·基尔主演，内容是一个新兵如何通过严酷的海军航空兵训练。

但是，事实表明，用这种方法管理高科技团队，有3个缺点。

首先，人们并不喜欢被这样管，尤其是那些对智商很自负的程序员。这些人实际上确实非常聪明，习惯于认定自己比别人知道得更多。要是这种自我认定恰恰是正确的（很大程度上肯定如此），那么当他们被“出于各种理由”命令去做某事时，他们会非常非常反感。但是，也没有足够好的理由废除这种方法……这里，让我们试着保持理性。高科技团队有许多目标，但让每个人都高兴这个目标很少排在第一位。

军事化管理法的另一个缺点是操作层面的，就是说，没有足够的时间用在微观管理上，原因很简单，因为经理的人数不够。在军队中，同时向一大群人发布一道命令是可行的，因为军队的通常情况就是每个人都在做同一件事。你可以向一个排的28个人大吼一声“擦枪”，然后打个盹，再去军官俱乐部，拿着一杯冰茶在阳台上慢慢品尝。在软件开发团队中，每个人干的活都不一样，所以如果想进行微观管理，就会变成“打了就跑”（hit-and-run）的抽风式管理。那就是，有一阵子你每件事都管着程序员，然后突然从他的生活中消失了几个星期，你跑开去管理其他程序员了。抽风式微观管理的问题在于，你无法坚持足够久来看到为什么你的决定行不通，或者你无法将整个过程的每一步理顺。从效果上看，你起到的所有作用只不过是每隔一会儿就将你手下的可怜程序员敲打一番，让他们像火车一样脱轨，然后下一个星期，他们不得不花上所有的时间找回每一节列车车厢，将它们放回到轨道上，将所有一切重新安排好，这种经历会让他们有一点点受伤。

第三个缺点是，在高科技公司中，负责干活的个人总是比“领导者”有更多的信息，所以他们其实是做决策的最佳人选。两个程序员在争论压缩图像的最好方法是什么。他们已经争论了两个小时，这时正好老板走进了办公室，听见了争论。那么在这三个人中，信息最少的那个人就是老板。所以你绝不要去做任何技术上的决策。我记得当我还在微软公司的时候，应用程序部门的负责人是Mike Maples，他是我的大老板，但是他坚决拒绝在技术问题上发表意见。渐渐地，程序员们明白了，他们不应该找他裁决技术问题。这迫使程序员在内部展开争论每个方案的优缺点，最后，问题总是按争论中占上风的人的意见解决，嗯，我的意思是，问题总是用现实中的最好方案解决。

如果军事化管理如此不利于团队运作，那么军队为什么用它呢？

这可以用我在军校里的经历解释。1986年，我在以色列伞兵部队服役。现在回想起来，我可能是以色列军队中有史以来最糟糕的伞兵。

士兵有几条要遵守的规矩。规矩一：如果发现周围有地雷，就要立刻静止不动。听上去很合理，对吧？在基础训练时，你被反复地灌输这一条。每隔一会，教官就大叫“地雷”，每个人只好静止不动。久而久之，你就养成了习惯。

规矩二：遇到敌人袭击时，就要一边开枪，一边冲向敌人。开枪使得敌人必须寻找掩护，所以他们就不能向你开火。冲向敌人可以使得你更接近他们，因此更容易瞄准，也就更容易灭掉他们。这条规矩听上去也非常合理。

好了，下面是一个面试中会遇到的问题：你们发现周围有地雷，这时有人开始朝你们射击，你们应该怎么做？

这并不是一个假设中才存在的情况。遇到这样的埋伏真的是很棘手。

标准的正确答案是，不要去想地雷，一边开枪，一边朝敌人冲过去。

这是因为，如果静止不动，那么敌人会一个接一个地把你们打死，直到所有人都死光。但是，如果你们发起冲锋，那么只有一部分人会触雷而死。两害相权取其轻，所以正确的做法是后者。

问题在于，如果一个士兵有头脑，他就不会在这种情况下发起冲锋。每个士兵都有巨大的动机作弊，自己保持静止不动，让其他更加英勇无畏的士兵去冲锋。这有点像囚徒的困境[①]（Prisoners' Dilemma）。

在生死关头，军队必须保证，一旦命令下达，士兵都会服从，即使是自杀式的命令。这意味着，必须让士兵养成服从命令的天性。但是，并不是天下所有事情都需要士兵完成。对于软件公司来说，服从命令就不一定有那么重要。

换句话说，军队使用军事化管理，因为这是唯一的办法，可以使得18

① “囚徒的困境”是经济学分支“博弈论”（Game Theory）中的一个经典问题，1950年首次提出。它的经典形式是，警察抓住两个罪犯，但是证据不足。如果两人都不认罪，就都将被拘留6个月；如果一人认罪，另一人不认罪，那么前者将被释放，后者将获得10年刑期；如果两人都认罪，将各获得5年刑期。

岁的年轻人在地雷阵中发起冲锋，而不是因为军方认为这是适用于所有情况的最佳管理方法。

尤其要指出的是，软件开发团队中的优秀程序员可以去任何他们想去的地方工作。在这种前提下，如果被人当成士兵一样对待，他们会感到相当扫兴，因此你要是这样做，最后就只能成为“光杆司令”了。

下面介绍本系列中的经济利益驱动法。

6

经济利益驱动法

2006年8月9日，星期三

先讲一个笑话。19世纪的时候，在俄国的一个小村庄里，住着一个贫穷的犹太人。有一天，他遇到了一个骑着马的哥萨克人[①]。

“你用什么喂鸡？”哥萨克人问。

“就用一点面包屑。”犹太人回答。

“你好大的胆子，竟敢用这么低等的饲料喂俄国鸡！”哥萨克人说，拿起棍子打犹太人。

第二天，哥萨克人又来了。“现在，你用什么喂鸡？”他问犹太人。

“报告大人，我给它上三道大菜，分别是新割下的鲜草，上等的鲟鱼鱼子酱，还有一小碗鲜奶油，上面还洒着进口的法国松露巧克力作为甜食。”

“白痴！”哥萨克人说，拿起棍子打犹太人，“你好大的胆子，竟敢在低等的家禽身上浪费这么好的食物！”

第三天，哥萨克人又来问：“你用什么喂鸡？”

“不喂了！”犹太人禀告，“我给它一个铜板，它想吃什么就自个儿去买。”

（哄堂大笑时间）

① 哥萨克人（Cossack），特指俄国历史上聚居在南部地区的一群游牧农民。哥萨克人以骁勇善战闻名，在沙俄时期，沙皇收买哥萨克人作为俄国军队的重要兵力来源，因此哥萨克人在当时的俄国社会中有相对较高的社会地位。

（没人笑？）

（哇啦啦）

（还是没人笑）

（哦，随你们的便）

我在这一讲的标题中用了一个词“经济学101”[2]（Econ 101），这个词只是正好在嘴边。对于我的非美国读者，我来简单解释一下。大多数美国高校中，任何领域的基础导论课程都用数字101表示。“经济学101管理法”指的是有些管理者所理解的经济学理论之片面，简直到了一种危险的程度。

“经济利益驱动法”假设每个人的行为动机都是金钱，让人们听命于你的最好方法就是给他们物质奖励或者物质惩罚，以此创造行为动机。

比如，如果美国在线（AOL）公司的客服人员能够成功劝说想要退订服务的顾客取消退订，那么也许每说成一个，AOL就会奖励他们一笔钱。

再比如，一个软件公司给编程错误最少的程序员发奖金。

如果这种方法能起作用，那么你的鸡就能拿着钱自己去买食物了。

这种方法的一个重大问题是，它将内部激励（intrinsic motivation）变为了外部激励（extrinsic motivation）。

内部激励是指你内心想将事情做好的天然愿望。人们开始干事的时候通常都怀着许许多多的内部激励。他们想做出优异的工作，他们想帮助人们理解一个月付给AOL公司24美元的费用是符合用户的最佳利益的，他们想要写出错误更少的代码。

外部激励是指来自外界的激励，有人付钱让你干某事就是外部激励。

内部激励比外部激励强得多。人们会为那些他们真正想做的事格外努力地工作。这一点并没有太大争议。

但是当你出钱让人们去做那些无论如何他们都想做的事情时，他们就会

① 这一章的英文原题是“The Econ 101 Management Method”，直译就是“经济学101管理法”，现在的中文题目“经济利益驱动法”是意译。

受到一种叫做“过度合理化效应”（Overjustification Effect）的支配。“我要写出没有bug的代码，因为我喜欢钱，我想要奖金。”他们会这样想。外部激励就取代了内部激励。因为外部激励是一种弱得多的激励，所以最终结果就是，你实际上降低了他们做出优异工作的愿望。当你停止支付奖金或者他们变得不太在乎钱时，他们就不再关心自己写出的代码是否没有bug了。

经济利益驱动法的另一个大问题是，人们有追求局部利益最大化（local maxima）的倾向。他们会想出办法将你支付给他们的报酬尽量最大化，但是实际上却没有达到你真正想要的结果。

举一个例子，你的“客户挽留专家”每成功挽留一个客户，就可以得到一笔奖金，他渴望获得更多的奖金，因此发疯一样地不放过客户，最终甚至连《纽约时报》都在头版发表长篇报道，抨击你的“客户服务”是多么卑鄙。虽然他的行为将你付给他的奖金最大化了，但是他这样的做法并没有真正最大化你想要的东西——利润。然后，你打算换一种方法，将他的行为与公司利润挂钩，比如给他13股的公司股票，但是你终究会认识到公司利润并非他可以控制的，所以那样做是浪费时间。

如果你使用经济利益驱动法，你就是在鼓励程序员与制度博弈。

假定你决定给代码错误最少的程序员发放奖金。这样一来，每当测试人员发现一个程序错误，都会演变成一场巨大的争论。通常情况下，程序员会让测试员相信那并不是一个真正的错误。或者测试员同意在向“错误追踪系统”正式提交记录前先与程序员“私下”解决。表面上，错误的数量下降了，但是实际上代码的质量并没有得到提高。

在这方面，程序员是非常聪明的。不管你用什么标准来评估他们的表现，他们都会找到办法将评估值最大化，所以你永远也得不到你真正想要的结果。

Robert D. Austin写过一本书《组织绩效评估与管理》（*Measuring and Managing Performance in Organizations*）。书中提到，当你引入新的绩效测量方法时，会有两个阶段的发展。第一阶段，你实际上得到了你想要的东西，因为还没人想出作弊的方法。但是，到了第二阶段，你实际上让事情变得比原来更糟，因为每一个人都想出了如何将你测量的指标值最大化的对策，即使代价是毁掉公司，他们也在所不惜。

更糟糕的是，信奉“经济利益驱动法”的经理们认定，他们只要不断地调整指标就可以避免上述情况。Austin博士的结论是，这样的想法是行不通的，它最终不会起作用。无论你多么努力地试着调整指标，力求让它们正确地反映你想得到的结果，最终一定事与愿违。

“经济利益驱动法”的最大问题是，它其实根本不是一种管理，更像是管理的退位，或者说是一种设计精巧的推卸责任的方法，不愿承担责任找到办法将事情做得更好。它是一个信号，表明管理层根本不知道如何引导人们做出更好的工作，所以他们强迫每个雇员在制度框架下自己想办法将事情做好。

你的目的是让程序员写出可靠的代码，但是你不是去训练他们，而是付钱让他们自己想办法完成，你逃避了自己的责任。这样一来，所有程序员不得不靠自己来找到办法。

对于那些更普通的工作，比如星巴克的柜台服务员或者AOL的电话客服人员，一般的职员几乎不可能靠自己想出改进工作的方法。你随便走进一家乡间咖啡馆，点了一小杯的加热焦糖豆奶拿铁咖啡。你将会发现，你不得不一遍又一遍地重复这个要求，你必须跟冲咖啡的人说一遍，当他们忘记的时候你还要再说一遍，最后你还要跟收银员说一遍，这样他们才能算出结账的金额。如果没人告诉职员如何改进工作，结果就是这样。在星巴克出现之前，没有一家咖啡馆想过如何解决这个问题。在星巴克，他们有一整套标准的训练程序，内容包括如何给每一种咖啡命名，如何在咖啡杯上做标记，如何大声报出订单，这样就确保了顾客只说一遍他想喝什么就可以了。这一整套方法是由星巴克总部发明的，效果惊人，但是连锁店里的员工永远都不会有办法靠自己将它创造出来。

你的客服人员要花大部分时间与客户交谈。他们没有足够的时间、兴趣或者没有受过相应的培训来想出如何更好地完成工作。在“客户挽留团队”中，没有人能够做出统计、分析数据、确定哪些挽留措施是最有效的以及怎样做才能不触怒客户，免得他们在网志中攻击公司。客服人员并不是特别关心这个，他们也没有足够的知识和足够的信息来关心这个，并且他们眼前的工作就已经把时间都占用光了。

作为一个经理，设计一个有效的系统是你的职责。这就是你拿到高薪的

原因。

如果你在童年时读过大量安·兰德[①]的小说，或者如果你只上了一个学期的经济学，还没有等到老师讲解效用[②]无法用美元衡量就不学了，那么你可能认为，建立一个简单的奖惩制度或者绩效工资（Pay For Performance）就能够方便地、完善地解决管理问题。但是，这种制度是不会起作用的。开始履行你自己的职责吧，别再把铜板给你的鸡，别再让它们自己去买吃的！

"Joel!"你大喊道，"在上一讲中你告诉我们，程序员应该自己做所有的决定。但是，今天你却告诉我们管理层应该做所有的决定。这是怎么回事？"

嗯，不完全是这样。在上一讲中我说过，程序员在基层，最了解情况。如果你们试图在微观层面进行管理，或者实行大声喊口令的军事化管理，很可能会导致不太理想的结果。而在这里，我正在告诉你们的东西是，当你创造一种制度的时候，你不能放弃自己的职责，不能通过给你的员工发钱的方式来训练他们。原则上，管理需要制度，这样人们才能完成工作。你们应该避免用外部激励取代内部激励。使用恐惧进行管理或者使用大声喊口令进行管理都不会很有效。

到目前为止，我已经放弃了军事化管理法和经济利益驱动法，还剩下一个管理方法，它可以让人们向正确的方向前进。我把它叫做"认同法"，将在下一讲中详细讨论。

① 安·兰德（1905—1982，Ayn Rand），俄裔美国女哲学家、小说家，著有《源泉》、《阿特拉斯耸耸肩》等畅销小说。她在作品中倡导一种个人主义、理性的利己主义以及完全自由放任的资本主义。

② 效用（utility）是一个经济学概念，指的是消费者感到的满意程度无法用客观指标衡量。

7

认 同 法

2006年8月10日，星期四

你想让一个团队中的所有人朝同一个方向齐心协力地工作。我已经在前面的文章中指出，军事化管理法和经济利益驱动法在高科技的知识团队中效果很差。

这样一来，只剩下一个办法了，我把它叫做“认同法”。这种管理方法的目标是，使得人们认同你希望达到的目标。它实施起来比其他方法难得多，而且还需要一些很不简单的人际沟通的技巧。但是，如果你真地做到了，它的效果就比其他方法好得多。

前面说过了，经济利益驱动法的问题是，它将内部激励变成了外部激励。认同法的作用恰恰就是设法创造出内部激励。

为了实行“认同法”，你必须动用所有的技巧，使得你的雇员认同公司的目标，这样他们才会感到极大的激励。然后，你还需要向他们提供必要的信息，使得他们向正确的方向前进。

怎样才能使雇员对公司有认同感？

如果公司的目标确实在某种程度上是高尚的，或者至少在别人看起来是高尚的，那么肯定有助于人们产生认同感。1984年，苹果公司在超级碗决赛时，播出了一则广告[①]。就是从那个时候起，苹果公司几乎创造出一种狂热的认同，它靠的差不多就是不停地讲：我们反对极权主义。这种话看上去是不是好像夸夸其谈？但是它的确起作用！我们也会说，在Fog Creek软件公

① 参见本书的第3讲。

程序员最可信赖的求职帮手

司，我们勇敢地站出来反对残杀小猫。耶！

我自己非常喜欢的一种做法是大家坐在一起吃饭。我总是提出，一起干活的人要一起吃饭。在Fog Creek软件公司中，我们每天都为整个团队提供午饭，大家一起围着一张大桌子吃饭。我想，不管怎么强调这种做法的巨大效果都不过分，它让人们感到公司就像一个大家庭，而且是一个和谐的大家庭。6年过去了，没有人离开。

我下面要说的话可能会吓到我们的一些暑期实习生。我的意思是，我们实习计划的目标之一就是让人们喜欢上纽约，觉得自己也是纽约的一份子，这样他们才会感到更放心，大学毕业后愿意搬到纽约来，同我们一起全职工作。为了达到这个目的，我们的暑假课外活动多得让人筋疲力尽：两出百老汇音乐剧，一次登高爬楼，一次环曼哈顿岛的划船，一次扬基棒球队的比赛，一次能够结识更多纽约人的室内晚会，一次博物馆之旅。室内晚会在我和迈克居住的公寓举办，目的不仅仅是欢迎实习生，也是一种手段，让他们对纽约的公寓生活有一个实际的印象，不要局限于我们安排他们住的宿舍楼。

一般来说，认同法要求你创造一个有凝聚力的、像胶水一样粘在一起的团队，就好像家庭一样。这样一来，人们就会对他们的同事产生忠诚感和义务感。

“认同法”的第二部分则是向人们提供必要的信息，使得公司向正确的方向前进。

今天的早些时候，程序员Brett走进我的办公室，同我讨论FogBugz 6.0的发布日期。他倾向于在2007年4月发布，我则倾向于在2006年12月发布。当然，如果在2007年4月发布，我们就会有更多的时间，可以在产品的各个方面进行修补和改进工作。如果在2006年12月发布，我们可能就不得不砍掉一大批很不错的新功能。

尽管如此，我还是要跟Brett解释：明年春天公司打算再雇6个人，如果FogBugz 6.0不早一点儿推出，我们负担新雇人手的费用就有相当大的困难。所以，我与Brett的谈话结束的时候，我让他明白了我之所以希望早一点发布新软件，完全是因为财政上的动机。现在他知道了这一点，我很有信心他会做出正确的决定……不一定就是附和我的观点。一种可能的情况是，即使不发布FogBugz 6.0，我们现有软件的销售也出现了很大的提升，那么现在Brett

已经了解了我们基本的财务状况，他就会想到，原有软件销售的上升意味着我们可以再延迟一段时间发布6.0版，为它再加上一些功能。我这样做的意义在于，通过分享信息，我能够使得Brett根据外部环境的变化相应地做出最有利于Fog Creek软件公司的决定。要是我换个方法，企图用奖金来打动他，我告诉他只要在4月之前，每提前一天发布软件，他就能得到现金奖励，那么，他想做的事情就会变成，当天晚上就把在公司内部公开的、包含错误的现有开发版本毁掉。如果我用军事化管理法，命令他必须及时发布没有错误的代码，他可能会服从命令，但是他会因此痛恨他的工作而选择离开。

结　论

天底下有多少个经理，就有多少种不同的管理方法。我在这几讲中，归纳出了三个主要方法。其中两个简单易行，但是效果不好，还有一个做起来比较困难，但是管用。不过实际情况是，许多软件开发团体会根据时间和对象的变化灵活运用各种管理方式。

第二部分

写给未来程序员的建议

8 学校只教Java的危险性

9 在耶鲁大学的演讲

10 给计算机系学生的建议

8

学校只教Java的危险性

2005年12月29日，星期四

如今的孩子变懒了。

多吃一点苦，又会怎么样呢？

我一定是变老了，才会这样喋喋不休地抱怨和感叹“如今的孩子”，不理解为什么他们不再愿意，或者说不再能够做艰苦的工作。

当我还是孩子的时候，学习编程需要用到穿孔卡片（punched card）。那时可没有任何类似“退格”键（Backspace key）这样的现代化功能，如果你出错了，就没有办法更正，只好扔掉出错的卡片，从头再来。

回想1991年我开始面试程序员的时候。我一般会出一些编程题，允许用任何编程语言解题。在99%的情况下，面试者选择C语言。

如今，面试者一般会选择Java语言。

说到这里，不要误会我的意思。作为一种开发工具，Java语言本身并没有什么错。

等一等，我要做个更正。我只是在本书特定的环境中不会提到作为一种开发工具Java语言有什么不好的地方。事实上，它有许许多多不好的地方，不过这些只有另找时间来谈了。

我在这里真正想要说的是，总的来看，Java不是一种非常难的编程语言，无法用来区分优秀程序员和普通程序员。它可能很适合用来完成工作，但是这个不是今天的主题。我甚至想说，Java语言不够难其实是它的特色，不能

算缺点。但是不管怎样，它就是有这个问题。

如果我听上去像是妄下论断，那么我想说一点儿我自己的微不足道的经历。大学计算机系的课程里有两个传统的知识点，但许多人从来都没有真正搞懂过，那就是指针（pointer）和递归（recursion）。

你进大学后，一开始总要上一门“数据结构”课，然后会有链表、散列表以及其他诸如此类的课程。这些课会大量使用“指针”，并且经常起到一种优胜劣汰的作用。因为这些课程非常难，学不会就表明学生的能力不足以达到计算机科学学士学位的要求，这些学生只能选择放弃这个专业。这是一件好事，因为如果你连指针都觉得很难，那么等学到后面要你证明不动点定理（fixed point theory）的时候，你该怎么办呢？

有些孩子读高中的时候就能用Basic语言在Apple II型个人电脑上写出漂亮的乓游戏[①]。等他们进了大学都会去选修计算机科学101课程，那门课讲的就是数据结构。当他们接触到指针那些玩意以后，就一下子完全傻眼了，后面的事情你都可以想象：他们就去改学政治学，因为看上去法学院是一个更好的出路[②]。关于计算机系的淘汰率，我见过各式各样的数字，通常在40%到70%之间。校方一般会觉得学生拿不到学位很可惜，我则视其为必要的筛选，淘汰那些没有兴趣编程或者没有能力编程的人。

对于许多计算机系的青年学生来说，另一门有难度的课程是有关函数式编程（functional programming）的，其中就包括递归程序设计（recursive programming）。麻省理工学院将这些课程的标准提得很高，还专门设立了一门必修课（课程代号6.001[③]），它的教材（*Structure and Interpretation of Computer Programs*，作者为Harold Abelson等，中文版书名为《计算机程序的构造和解释》）被几百所高校的计算机系采用，充当事实上的计算机科学导论课程。（你能在网上找到这门课的视频[④]，应该看一下。）

① 乓（Pong）游戏是美国雅达利电脑公司在1972年推出后的一款投币式街机游戏，其游戏名称的英文Pong来自球被击中后发出的声音。《乓》是世界第一款街机游戏。——编者注

② 在美国，法学院的入学者都必须具有本科学位。通常来说，主修政治学的学生升入法学院的机会最大。

③ 在麻省理工学院，计算机系的课程代码都是以6开头的，6.001表明这是计算机系的最基础课程。

④ 网址是http://groups.csail.mit.edu/mac/classes/6.001/abelson-sussman-lectures/。

这些课程难得惊人。在第一堂课，你就要学完Scheme语言[①]的几乎所有内容，你还会遇到一个不动点函数，它的参数本身就是另一个函数。我读的这门导论课是宾夕法尼亚大学的CSE 121课程，真是读得苦不堪言。我注意到很多学生（也许是大部分的学生）都无法完成这门课。课程的内容实在太难了。我给教授写了一封长长的声泪俱下的Email，控诉这门课不是给人学的。宾夕法尼亚大学里一定有人听到了我的呼声（或者听到了其他抱怨者的呼声），因为如今这门课讲授的计算机语言是Java。

我现在觉得，他们还不如没有听见呢。

这就是争议所在。许多年来，像当年的我一样懒惰的计算机系本科生不停地抱怨，再加上计算机业界也在抱怨毕业生不够用，这一切终于造成了重大恶果。过去十年中，大量本来堪称完美的好学校，都百分之百转向了Java语言的怀抱。这真是好得没话说了，那些用grep命令[②]过滤简历的企业招聘主管，大概会很喜欢这样。最妙不可言的是，Java语言中没有什么太难的地方，不会真地淘汰什么人，你搞不懂指针或者递归也没关系。所以，计算系的淘汰率就降低了，学生人数上升了，经费预算变大了，可谓皆大欢喜。

学习Java语言的孩子是幸运的，因为当他们用到以指针为基础的散列表时，他们永远也不会遇到古怪的“段错误”[③]，他们永远不会因为无法将数据塞进有限的内存空间而急得发疯，他们也永远不用苦苦思索，为什么在一个纯函数的程序中，一个变量的值一会儿保持不变，一会儿又变个不停！多么自相矛盾啊！

他们不需要怎么动脑筋就可以在专业课上得到4.0的成绩。

我是不是有点太苛刻了？就像电视里的“四个约克郡男人”[④]那样，

① Scheme语言是LISP语言的一个变种，于1975年诞生于麻省理工学院，以其对函数式编程的支持而闻名。这种语言在商业领域的应用很少，但是在计算机教育领域内有着广泛影响。

② grep是UNIX/Linux环境中用于搜索或者过滤内容的命令。这里指的是，某些招聘人员仅仅根据一些关键词来过滤简历，比如本文中的Java。

③ 段错误（segfault）是segmentation fault的缩写，指的是软件中的一类特定的错误，通常发生在程序试图读取不允许读取的内存地址或者以非法方式读取内存的时候。

④ 《四个约克郡男人》（Four Yorkshiremen）是英国电视系列喜剧At Last the 1948 Show中的一部，于20世纪70年代播出。内容是四个约克郡男人竞相吹嘘各自的童年是多么困苦。由于内容太夸张，所以显得非常可笑。

成了老古板？就知道在这里吹嘘我是多么刻苦，完成了所有那些高难度的课程？

我再告诉你一件事。1900年的时候，拉丁语和希腊语都是大学里的必修课，原因不是因为它们有什么特别的作用，而是因为它们有点被看成是受过高等教育的人士的标志。在某种程度上，我的观点同拉丁语支持者的观点没有不同（下面的四点理由都是如此）："（拉丁语）训练你的思维，锻炼你的记忆。分析拉丁语的句法结构是思考能力的最佳练习，是真正对智力的挑战，能够很好地培养逻辑能力。"以上出自Scott Barker之口（http://www.promotelatin.org/whylatin.htm）。但是，今天我找不到一所大学还把拉丁语作为必修课。指针和递归不正像计算机科学中的拉丁语和希腊语吗？

说到这里，我坦率地承认，当今的软件代码中90%都不需要使用指针。事实上，如果在正式产品中使用指针，这将是十分危险的。好的，这一点没有异议。与此同时，函数式编程在实际开发中用到的也不多。这一点我也同意。

但是，对于某些最激动人心的编程任务来说，指针仍然是非常重要的。比如说，如果不用指针，你根本没办法开发Linux的内核。如果你不是真正地理解了指针，你连一行Linux的代码也看不懂，说实话，任何操作系统的代码你都看不懂。

如果你不懂函数式编程，你就无法创造出MapReduce[①]，正是这种算法使得Google的可扩展性（scalable）达到如此巨大的规模。术语"Map"（映射）和"Reduce"（化简）分别来自Lisp语言和函数式编程。回想起来，在类似6.001这样的编程课程中，都提到纯粹的函数式编程没有副作用，因此可以直接用于并行计算。任何人只要还记得这些内容，那么MapReduce对他来说就是显而易见的。发明MapReduce的公司是谷歌，而不是微软，这个简单的事实说出了原因，为什么微软至今还在追赶，还在试图提供最基本的搜索服务，而谷歌已经转向了下一个阶段，开发Skynet，我的意思是，开发世界上最大的并行式超级计算机。我觉得，微软并没有完全明白在这一波竞争中它落后了多远。

① MapReduce是一种由Google引入使用的软件框架，用于支持计算机集群环境下海量数据（PB级别）的并行计算。

除了上面那些直接就能想到的重要性，指针和递归的真正价值在于那种你在学习它们的过程中所得到的思维深度，以及你因为害怕在这些课程中被淘汰所产生的心理抗压能力，它们都是在建造大型系统的过程中必不可少的。指针和递归要求一定水平的推理能力、抽象思考能力，以及最重要的，在若干个不同的抽象层次上同时审视同一个问题的能力。因此，是否真正理解指针和递归与是否是一个优秀程序员直接相关。

如果计算机系的课程都与Java语言有关，那么对于那些在智力上无法应付复杂概念的学生而言，就没有东西可以真地淘汰他们。作为一个雇主，我发现那些100% Java教学的计算机系已经培养出了相当一大批毕业生，这些学生只能勉强完成难度日益降低的课程作业，只会用Java语言编写简单的记账程序，如果你让他们编写一个更难的东西，他们就束手无策了。他们的智力不足以成为程序员。这些学生永远也通不过麻省理工学院的6.001课程，或者耶鲁大学的CS 323课程。坦率地说，为什么在一个雇主的心目中，麻省理工学院或者耶鲁大学计算机系的学位的份量要重于杜克大学的，这就是原因之一。因为杜克大学最近已经全部转为用Java语言教学。宾夕法尼亚大学的情况也很类似，当初CSE 121课程中的Scheme语言和ML语言几乎将我和我的同学折磨至死，如今已经全部被Java语言替代。我的意思不是说我不想雇用来自杜克大学或者宾夕法尼亚大学的聪明学生，我真地愿意雇用他们，只是对于我来说，确定他们是否真地聪明如今变得难多了。以前，我能够分辨出谁是聪明学生，因为他们可以在一分钟内看懂一个递归算法，或者可以迅速在计算机上实现一个线性链表操作函数，所用的时间同在黑板上写一遍这个函数差不多。但是对于在学校只学Java语言的毕业生，看着他们面对上述问题苦苦思索却做不出来的样子，我分辨不出这到底是因为学校里没教，还是因为他们不具备编写优秀软件的素质。Paul Graham将这一类程序员称为“Blub程序员”[①]（www.paulgraham.com/avg.html）。

大学里只教Java语言，无法淘汰那些永远也成不了优秀程序员的学生，这已经是很糟糕的事情了。但是，学校可以无可厚非地辩解，这不是校方的错。整个软件行业，或者说至少是其中那些使用grep命令过滤简历的招聘经理，确实是在一直叫嚷，要求学校使用Java语言教学。

① Blub程序员（Blub programmers）指的是那些企图用一种语言解决所有问题的程序员。Blub是Paul Graham假设的一种高级编程语言。

但是，即使如此，学校的教学也还是失败的，因为学校没有成功训练好学生的头脑，没有使他们变得足够熟练、敏捷、灵活，能够做出高质量的软件设计。（我不是指面向对象式的“设计”，那种编程只不过是要求你花上无数个小时重写你的代码，使它们能够满足面向对象编程的等级制继承式结构，或者说要求你思考到底对象之间是“has-a”从属关系，还是“is-a”继承关系，这种“伪问题”将你搞得烦躁不安。）你需要的是那种能够在多个抽象层次上同时思考问题的训练。这种思考能力正是设计出优秀软件架构所必需的。

你也许想知道，在教学中，OOP（Object-Oriented Programming，面向对象编程）是否是指针和递归的优质替代品，是不是也能起到淘汰作用。简单的回答是“不”。我在这里不讨论OOP的优点，我只指出OOP不够难，无法淘汰平庸的程序员。大多数时候，OOP教学的主要内容就是记住一堆专有名词，比如“封装”（encapsulation）和“继承”（inheritance），然后再做一堆多选题小测验，考你是不是明白“多态”（polymorphism）和“重载”（overloading）的区别。这同历史课上要求你记住重要的日期和人名的难度差不多。OOP不构成对智力的太大挑战，吓不跑一年级新生。据说，如果你没学好OOP，你的程序依然可以运行，只是维护起来有点难。但是如果你没学好指针，你的程序就会输出一行段错误信息，而且你对什么地方出错了毫不知情，然后你只好停下来，深吸一口气，真正开始努力在两个不同的抽象层次上同时思考你的程序是如何运行的。

顺便说一句，我有充分理由在这里说，那些使用grep命令过滤简历的招聘经理真是荒谬可笑。我从来没有见过哪个能用Scheme语言、Haskell语言和C语言中的指针编程的人，竟然不能在两天里面学会Java语言，并且写出的Java程序的质量竟然不能胜过那些有5年Java编程经验的人士。不过，是无法指望人力资源部里那些平庸的懒汉听进去这些话的。

再说，计算机系承担的发扬光大计算机科学的使命该怎么办呢？计算机系毕竟不是职业学校啊！训练学生如何在这个行业里工作不应该是计算机系的任务，这应该是社区高校和政府就业培训计划的任务，那些地方会教给你工作技能。计算机系给予学生的理应是他们日后生活所需要的基础知识，而不是为学生第一周上班做准备。对不对？

还有，计算机科学是由证明（递归）、算法（递归）、语言（λ演算[①]）、操作系统（指针）、编译器（λ演算）所组成的，所以说那些不教C语言、不教Scheme语言、只教Java语言的学校实际上根本不是在教授计算机科学。虽然对于真实世界来说，有些概念可能毫无用处，比如函数的科里化（function currying）[②]，但是这些知识显然是进入计算机科学研究生院的前提。我不明白，计算机系课程设置委员会中的教授为什么会同意将课程的难度下降到如此低的地步，以至于他们既无法培养出合格的程序员，甚至也无法培养出合格的能够得到哲学博士学位[③]、进而能够申请教职、与他们竞争工作岗位的研究人员。噢，且慢，我说错了。也许我明白原因了。

实际上，如果你回顾和学术界在“Java大迁移”（Great Java Shift）中的争论，你会注意到，最大的议题是Java语言是否还不够简单，不适合作为一种教学语言。

我的老天啊，我心里说，他们还在设法让课程变得更简单。为什么不干脆用匙子把所有东西一勺勺都喂到学生嘴里呢？让我们再请助教帮他们接管考试，这样一来就没有学生会改学“美国研究”[④]了。如果课程被精心设计，使得所有内容都比原有内容更容易，那么怎么可能期望任何人从这个地方学到任何东西呢？看上去似乎有一个工作小组（Java task force）正在开展工作，创造出一个简化的Java的子集，以便在课堂上教学[⑤]。这些人的目标是生成一个简化的文档，小心地不让学生纤弱的思想接触到任何EJB/J2EE的脏东西[⑥]。这样一来，学生的小脑袋就不会因为遇到有点难度的课程而感到烦恼了，除非那门课里只要求做一些空前简单的计算机习题。

① λ演算（lambda calculus）是一套用于研究函数定义、函数应用和递归的形式系统，在递归理论和函数式编程中有着广泛的应用。

② 函数的科里化（function currying）指的是一种多元函数的消元技巧，将其变为一系列只有一元的链式函数。它最早是由美国数学家哈斯格尔·科里（Haskell Curry）提出的，因此而得名。

③ 在美国，所有基础理论的学科一律授予的都是哲学博士学位（Doctor of Philosophy），计算机科学系亦是如此。

④ “美国研究”是对美国社会的经济、历史、文化等各个方面进行研究的一门学科。这里指的是，计算机系学生不会因为课程太难被淘汰，所以就不用改学相对容易的“美国研究”。

⑤ 参见http://www.sigcse.org/topics/javataskforce/java-task-force.pdf。

⑥ J2EE是Java 2平台企业版（Java 2 Platform, Enterprise Edition），指的是一整套企业级开发架构。EJB（Enterprise JavaBean）属于J2EE的一部分，是一个基于组件的企业级开发规范。它们通常被认为是Java中相对较难的部分。

程序员最可信赖的求职帮手

计算机系如此积极地降低课程难度，有一个最善意的解释，那就是节省出更多的时间去教授真正属于计算机科学的概念。但是，前提是不能花费整整两节课向学生讲解诸如Java语言中int和Integer有何区别[①]。好的，如果真是这样，课程6.001就是你的完美选择。你可以先讲Scheme语言，这种教学语言简单到聪明学生大约只用10分钟就能全部学会的程度。然后，你将这个学期剩下的时间都用来讲解不动点。

唉。

说了半天，我还是在说要学1和0。

（你有1？真幸运啊！我们那时所有人得到的都是0。）

① 在Java语言中，int是一种数据类型，表示整数，而Integer是一个适用于面向对象编程的类，表示整数对象。两者的涵义和性质都不一样。

9

在耶鲁大学的演讲

2007年12月3日，星期一

本文是2007年11月28日在耶鲁大学计算机系演讲的第一部分。

1991年，我大学毕业，得到了计算机科学学士学位。16年过去了。今天，我想讲一讲我的计算机系本科经历对我的职业生涯产生了怎样的影响。我的职业生涯开始是开发软件，写代码，后来又开了一家软件公司。当然，我这样的演讲主题有一点儿不合理。我记得，麻省理工学院“计算机导论课程”教材的开头有一个著名的段落，Hal Abelson[①]提出并且详细解释了“计算机科学”这个学科其实主要不是研究计算机，也不是传统意义上的科学。所以，我在这里谈论计算机科学专业，应该比其他专业，比如媒体研究专业或者文化人类学专业，更多地训练学生未来如何从事软件开发职业。我这真是有点自以为是。

但是不管怎样，我还是要讲下去。我要讲几门我在大学里上过的最有用的课程，其中一门我只听了第一次，就再也不去了。另一门Roger Schank教授讲的课，被系里的其他老师认为毫无用处，甚至都不给学分，但是我觉得很有用，一会儿我会详细谈。

第三门课是一门很不容易通过的课程，我们那时叫做CS 322，不过现在已经改叫CS 323了。在我读书的年代，CS 322有很多课后作业，所以它的学分是1.5分。但是，耶鲁大学的规定是，另外的半个学分必须从同一个系的

① Hal Abelson，也就是Harold Abelson，为麻省理工学院的计算机系教授，自由软件基金会和创作共用许可证的共同创始人之一。这里指的是他同Gerald Jay Sussman合作编写的*Structure and Interpretation of Computer Programs*（SICP）一书，该书是麻省理工学院计算机科学导论课程的教材。

课程中获得才有效。问题是，计算机系还有其他两门1.5学分的课程，但是必须同时选修，不能单独选。所以，通过这样巧妙的设计，CS 322的那半个学分完全无用。不过，那门课设置1.5学分是合理的，因为它每周的课后习题需要40个小时才能完成。多年来，学生一直抱怨个不停，所以现在这门课做了调整，改成了1个学分，编号调整为CS 323，每周40小时的课后作业却没变。除了这一点以外，现在的这门课同原来完全一样。我热爱这门课，因为我热爱编程。CS 323有一个最大的优点，那就是它让许多人明白了原来自己不是编程的那块料，永远也成不了程序员。这是一件好事。如果不是授课老师Stan Eisenstat教授让一些人明白自己其实不具备编程的能力，他们就会有悲惨的职业生涯，一生中忙于复制和粘贴大量他人编写的Java代码。顺便说一句，如果你在CS 323课程中得到了A，欢迎来我的Fog Creek软件公司，我们有非常棒的暑期实习生项目。你可以在讲座结束后来找我。

就我所知，耶鲁大学计算机系的核心课程一直没有变过。同16年前我在学校的时候相比，课程201、课程223、课程240、课程323、课程365、课程421、课程422、课程424、课程429看上去没有什么不同。自从我进入耶鲁大学以后，主修计算机的学生人数一直在上升，虽然中间有一段日子好像出现了下降，但是那是因为早先互联网泡沫时期入学人数暴增、基数抬高所致。现在，可供选修的有趣课程比我那时多得多。所以，这就是进步。

那时有一阵子，我认定我会去读博士。我的父母都是教授。他们的朋友中有那么多大学老师，所以我从小到大一直觉得所有的人成年后都会去读博士。这么说吧，我认为本科毕业以后申请攻读计算机科学的研究生是理所当然的。后来我在耶鲁大学计算机系，也就是此地大家所在的这个系，选修了一门叫做“动态逻辑”（dynamic logic）的课，我才改变了想法。那门课的任课教师是Lenore Zuck，她现在已经不在这里了，去了UIC（伊利诺伊大学芝加哥分校）。

那门课我只听了一次，课里面讲的东西我也没听懂多少。从我接触到的内容判断，动态逻辑与形式逻辑（formal logic）很相像：苏格拉底是一个人，所有人都会死亡，所以苏格拉底也会死亡。两者的不同之处在于，在动态逻辑中，真值（truth value）会随着时间发生变化。以前苏格拉底是一个人，现在他是一只猫，诸如此类。从理论上看，如果要证明与计算机程序相关的问题，动态逻辑不失为一种有趣的方法。因为在计算机程序中，状态值（即

真值）也会随着时间发生改变。

在第一讲中，Zuck博士先讲了一些公理和变换规则（transformation rule），然后就开始证明一道非常简单的题目。她有一个计算机程序，内容是 `f := not f`，其中f是一个布尔值，也就是一位（bit）每一次进行取反运算。题目的要求是，证明这个程序运行偶数次后，f的值同程序运行之前相同。

证明过程非常冗长，一直在往下推导。如果我没有记错，当时就是在现在这一间教室里，现在的地毯看上去就是那时的那一块，没有换过。我身后的黑板，当时写满了推导的步骤。Zuck博士的证明中用到了归纳法（induction）、反证法（reductio ad absurdum）、穷举法（exhaustion）。那堂课上到很晚，单单这个证明就已经讲了40分钟。到后来，Zuck博士一度无法证明下去，都变得绝望了，幸亏有一个研究生帮了她。事情是这样的，她说："我真不记得这一步应该怎么证明了。"一个坐在前排的研究生说："没错，就是这样，教授，你的思路是对的。"

当所有的推导过程都讲解完、写在了黑板上、这一次漫长的证明总算要结束了时，她却发现，得到的结果与要求证明的命题正好相反。没有人知道怎么会这样。直到后来，又是那个研究生发现，在倒数第63步，有一位的值写反了，原因是黑板上的那个位置碰巧沾上了一点脏东西。改正过来以后，就万事大吉了。

她给我们布置了课后作业，要求我们证明这道题的逆命题：如果程序 `f := not f`的运行次数为n次，并且f所代表的位等于程序运行前的值，请证明n一定是偶数。

我花了好几个小时做这道题。Zuck博士的原始证明我都抄了下来并放在面前，我按照她的思路去做。经过严格的检查，我发现她的证明实际上还是太"简略"，缺少了各种各样微小的中间步骤。但是对我来说，这些步骤没有一个是显而易见的，它们都成了我的"重大"障碍。我把贝克顿中心[①]（Becton Center）里有关动态逻辑的所有书都找了出来，全部读了一遍。我苦苦思索这道题，直到深夜。但是即使如此，我依然一点进展也没有，我对计算机理论越来越绝望。我开始觉得，如果那么琐碎的一个命题都需要长篇累牍的证明，那么在证明中犯错的可能性会远远大于证明成功的可

① 贝克顿中心是耶鲁大学工学院所在地，该幢建筑中有一个图书馆。

能性。我认为，使用动态逻辑这种艰深枯涩的理论来证明计算机程序这样实际的、有趣的玩意，真不是一种有成效的方式。因为，你凭着直觉就能确定`f := not f`这个程序会产生什么样的结果，但是如果你要证明它，反而非常可能出错。所以，我就放弃选修这门课。幸亏有试听制度（shopping period），允许学生试听后再做决定。而且，我还当场决定，我不适合去读计算机系的研究生。正是这个决定，使得“动态逻辑”成了我上过的最有用的一门课。

说到这里，就引出了我职业生涯中的一个重要发现。周而复始地，你会注意到，当程度员遇到问题的时候，他们会把问题重新定义，使得这些问题可以用算法解决。这样一来，问题转化成他们可以解决的形式，但是实际上，那些问题是一种“琐碎”问题。也就是说，程序员解决的只是问题的某种外在形式，而并没有解决真正的问题，原因是这些问题非常难，不是表面的算法可以概括的。下面我就给你们举一个例子。

有一种说法，你们以后会经常听到，那就是软件工程或多或少正在遭遇质量危机。我本人是不同意这种说法的，同生活中的其他商品相比，大多数人用到的大多数软件质量好得出奇。但是，这种说法指的不是这个意思，它所称的“质量危机”涉及许多观念和研究，目标就是如何才能生产出更高质量的软件。从这个角度看，计算机界可以分成技术派（geek）和务实派（suits）两大类[①]。

技术派想要把质量问题用软件自动处理。为了这个目的，他们发明了单元测试[②]（unit testing）、测试驱动开发方法（test-driven development）、自动测试（automated testing）、动态逻辑等，目的只有一个，就是“证明”程序中没有错误。

务实派并不真地关心质量有没有问题。只要有人愿意出钱购买软件，他们才不想关心代码中有没有错误。

当前，在技术派与务实派的大战中，务实派是获胜的一方，因为他们控制了公司的预算。老实说，我不觉得这是一件很糟糕的事情。务实派认识到，

① 原文中，作者用的词是the geeks和the suits。前者的原意是指动作反常的人，后来特指电子技术方面的高手。后者的原意是指西装的套装。

② 单元测试是一种用程序自动检验源代码是否正确的软件测试技术，这里的“单元”是指软件中可供测试的最小单位。

消灭软件代码中的错误是一件边际报酬递减的事情[①]。一旦软件的质量达到了一定的水准，能够用来解决特定的问题，那么就会有用户从这个软件中获益，用户也会因此愿意出钱购买。

同时，务实派对于“质量”有一个更广义的定义。你尽管大胆地想象，这个定义完全符合利益原则。所谓软件的“质量”，就是看它能为大家带来多少奖金，奖金越多，也就表明软件的质量越高。出人意料的是，“质量”的这种定义有深得多的内涵，远不止于写出没有错误的代码。举例来说，这个定义很看重为软件添加更多的功能，使得它能为更多的人解决更多的问题，而技术派很可能会嘲讽这种类型的软件为“膨胀件”（bloatware）。这个定义还看重将软件做得更美观，一个好看的软件就是比一个难看的软件销量更好。这个定义还看重程序是否能使用户感到更愉快。基本上，这个定义就是让用户自己来表达什么是软件的质量，让用户自己来决定某一个程序是否符合他们的需要。

现在回过头再看技术派，他们只是对狭义的技术方面的质量感兴趣。他们只关注从代码中能够看出来的东西，而不关心用户将会怎么判断。他们是程序员，所以会想让生活中每一件事情都自动完成，因此很自然地，他们也想自动完成QA（Quality Assurance，品质保证）过程。这就是单元测试的来历。不要误解我，单元测试本身并不坏，当你运行完所有的单元测试，就可以机械式地“证明”一个程序是“正确的”。但是，这样做的不利之处是，任何不能被自动测试的东西就会被排斥在质量的定义之外，变成与软件质量无关。即使我们明知用户更喜欢看上去很漂亮的软件，但是因为没有办法自动评估一个软件看上去到底有多漂亮，所以软件的美观就被排斥在自动化QA过程之外。

事实上，如果你观察死硬的技术派，你会看到他们倾向于放弃所有人为的有效质量评估手段，大概只保留那些他们可以用机械方法证明的手段。换句话说，他们最终证明的只是软件的运行过程是否同设计规格一模一样。这样一来，我们就得到了一个非常狭窄、非常技术化倾向的质量定义，即软件的质量反映在与设计规格上的标准有多接近，软件越符合设计规格，就代表质量越高。如果输入是给定的，程序能够产生给定的输出吗？

① 边际报酬是一个经济学概念，指下一个单位的产品所带来的利润。文中的意思是，随着软件中的错误越来越少，消灭每一个错误所带来的收益也在变小。

这里就产生了一个非常重要的问题。为了用机械方法证明程序符合某种规格，规格说明书本身就需要写得非常详尽。实际上，设计规格说明书上不得不因此定义清楚与程序相关的每一件事，否则的话，自动的机械化证明就什么也做不了。好吧，就算设计规格说明书将程序运行的每一个方面都定义清楚了，那么，你有没有发现，设计规格说明书中其实包含了编写程序需要的所有信息！到了这个时候，某些技术派会站起身走到一个非常昏暗的地方，开始动脑筋思考如何将设计规格说明书自动编译为程序。他们会觉得自己发明了一种方法，不用编程就制造了程序。

这么说吧，这就是永动机理论在软件工程中的等价物。只有狂想家才会不知疲倦地想把它们制造出来。不管你跟他们说多少次这样做行不通，他们还是会去做。如果设计规格说明书详尽定义了程序的一切行为，详尽到可以从本身直接生成程序，那么这就会有点像循环论证[①]（begging the question）。你应该怎么写设计规格说明书呢？写出如此详尽的设计规格说明书与直接写出相对应的程序是完全一样的，因为设计规格说明书的作者需要考虑的细节与程序员需要考虑的是一样多的。用信息论的术语来说就是，程序本身包含多少香农熵[②]（Shannon entropy），设计规格说明书中就会包含同样数量的香农熵。每一个单位的熵都需要规格说明书作者或者软件作者做出决定。

所以，总而言之，如果真有一种机械方法可以证明软件程序的正确性，那么实际上你所能证明的只是现有的程序是否同某些其他程序完全一模一样。也就是说，后一个程序同前一个程序一定必须包含同样数量的熵。否则的话，证明对象中一定有一些行为没有被定义，因此也就无法证明。这样就会导致写出一个设计规格说明书等价于写出一个程序。你的行为的所有意义仅仅是把一个问题从这里搬到了那里，你实际上什么也没有做。

这样的例子看上去有点太残忍，但是虽然如此，许多程序员还是热衷于寻找如何验证软件质量的圣杯[③]（holy grail），许多人白白走了许多弯路。微软公司的Windows Vista开发小组就是这样的一个例子。根据网络上流传的消

① 循环论证，指的是一种逻辑错误（logical fallacy），即用来证明的前提中已经假设将要证明的命题是正确的。

② 香农熵是一种对随机变量中包含的不确定性的度量，1948年由Claude E. Shannon首先提出。

③ 圣杯是基督教传说中耶稣在最后的晚餐时使用的餐具，具有奇迹般的魔力。

息和现实中的传闻，微软公司显然有一个长期政策，所有不会写代码的软件测试员都要被替换掉，改成微软所称的SDET人员，也就是“软件测试开发工程师”（Software Development Engineers in Test），这种工程师是专门负责写自动测试脚本的程序员。

老式的微软公司测试员要做很多事情，包括检查字体是否协调和清晰，检查对话框中控制按钮的位置是否合理和对齐，检查用户操作时屏幕是否闪烁，观察用户界面的流程，考虑软件是否容易上手，判断用词是否合适，评估软件的表现，检查所有错误信息的拼写和语法。他们花费大量时间确保软件的不同组成部分中用户界面都保持一致，因为协调的用户界面会让用户更容易上手。

所有这些事情，没有一件可以用程序自动完成。微软公司向自动测试靠拢的政策导致的后果之一就是，Windows Vista发布出来的时候，这个产品是极端不协调的，很多细节处都没有处理好。许许多多非常明显的问题没有解决，最终产品就上市了。这些问题中，没有一个是自动化脚本进行软件测试时定义的“错误”，但是它们中的任何一个都会让用户产生一种总的看法，即Windows Vista不如Windows XP。在这里，技术派的质量定义压倒了务实派的定义。我完全相信，在微软公司内部，现在所有的自动化测试脚本都显示软件100%合格。但是，这又有什么用呢？此刻，差不多每一个技术评论家都建议用户，只要有可能，就继续使用Windows XP。看上去没有人写过这样一个自动化测试，就是去测试Windows Vista有没有给用户提供一个令人信服的从Windows XP升级的理由。

我批评微软公司不代表我恨它，真的，我一点都不恨它。事实上，我走出校园的第一份工作就是微软给的。那个时候说出来在那里工作，没有人会觉得那是一个好地方。感觉上有点像你找了一份马戏团里的工作。别人看着你会感到很可笑。这是真的吗？你是在说微软公司吗？那个时候，尤其在学校里，大家都把微软看作一家典型的法人组织，非常地无趣，没有独创性，专门制造劣质软件供会计使用。哦，我其实不了解会计工作，但是微软就是制造那些让会计可以画电子表格或者干其他只有会计才会干的事情的软件。真是悲惨得没话说了！而且，这样的软件都运行在一个可怜的、名为MS-DOS的单任务操作系统之上，这个操作系统中充满了莫名其妙的愚蠢的限制，比如文件名的长度不能超过8个字符，并且没有Email，没有Telnet，也没有

Usenet。好吧，MS-DOS早就被淘汰了，但是UNIX用户和Windows用户之间的文化隔阂反而变得更深了。这是一场文化的战争。两者之间的分歧既是错综复杂的，又是有着根本冲突的。在耶鲁大学的学生看来，那时的微软公司是一家生产小儿科操作系统的软件公司，使用的还是30年前的计算机技术。而在那时的微软公司里，“计算机科学”并不是一个好词，而是专门用来取笑新进公司的毕业生的，这些毕业生竟然会莫名地认定Haskell语言是除C语言之外最重要的开发语言。

我举一个很小的例子，向你们说明UNIX和Windows之间的文化战争。“UNIX文化”倡导将用户界面和功能分隔开来。一个正宗的UNIX程序，第一步总是会提供一个命令行界面。幸运的话，有人会出来为它写一个漂亮的前台界面，包含阴影、透明和3D效果。这个漂亮的前台界面所做的只是在后台调用命令行界面。如果出于某种不可知的原因，命令行界面在后台无法运行，因此无法得到正确的结果，那么前台界面就会挂掉，在那里等着永远都不会得到的输入。

不过，好消息是你可以在脚本中使用命令行界面。

形成鲜明对照的是，在“Windows文化”中，第一步总是先写出GUI（图形用户界面），所有核心功能与用户界面的代码联系之紧密足以让人感到绝望。你因此得到了像Photoshop那样的巨型应用程序，它用来编辑照片绝对第一流。但是，如果你是一个程序员，你有一个文件目录中包含了1000张图片，你想用Photoshop重新设定它们的大小，每张照片都改成200像素乘以200像素，那么你是无法写出代码的，因为软件的所有功能都紧紧地与某一个特定的用户界面绑在了一起。

不论从什么角度看，这两种文化的差别大致上就是阳春白雪与下里巴人之间的差别。事实上，它很精确地反映在全国各个计算机系的课程设置上。在常春藤联盟[①]（Ivy League）的院校中，只教授UNIX、函数式编程、状态机[②]（state machine）理论。当你顺着排名的顺序往下看到那些越来越少人问津的学校，你就会看到Java语言开始在课程中出现了。你再往下看，大

① 常春藤联盟指的是美国东北部的8所顶尖高等学府，这8所学校有着许多共同的特点，都属于美国最顶尖、最难考入的大学。

② 状态机在计算机理论中指的是一种行为模型，由一系列状态和描述状态之间转变规则的函数组成。

概就会看到有一些课程起了这样的名字：Microsoft Visual Studio 2005的101课程，3个学分。等到你看到2年制学院的部分，会发现有些课程的名字，同“21天SQL Server证书课程”差不多，简直如同周末有线电视中的广告一样。想找好工作吗？现在正是你最好的时机开始学习（另一个声音出来说）Java Enterprise Beans[①]！

2007年12月4日，星期二

本文是2007年11月28日在耶鲁大学计算机系演讲的第二部分。

我在华盛顿州的雷德蒙[②]（Redmond）待了几年，越来越不适应那里的环境。我做了一个匆忙的决定，重新回到了纽约。开头几个月，我待在微软的纽约办公室工作，作为微软咨询部门的一个客户顾问。我在那里干得糟透了，是一场彻头彻尾的失败。于是，我离开了微软，去了维亚康姆[③]（Viacom）公司。那时是20世纪90年代中期，互联网刚刚开始大规模使用。维亚康姆是一个巨型的大集团，拥有MTV有线电视网、VH1有线电视频道、Nickelodeon有线电视频道、Blockbuster租片连锁店、派拉蒙电影公司、Comedy Central电视频道、CBS电视网以及其他许多娱乐业公司。就是在那里，我第一次体验到了大多数程序员是如何谋生的。有一种恐怖的东西叫做“内部软件”（inhouse software），它简直让人毛骨悚然。你永远都不会想编写内部软件。让我随便来举个例子，你是一家大公司雇用的程序员，这家公司的产品，嗯，就算是铝制易拉罐吧。市场上没有该公司所需的生产工艺，该公司只能自己完成相关软件，所以公司就会雇用内部程序员（in-house programmer）或者向埃森哲[④]和IBM这样的公司聘请高价的外部程序员。为什么这种工作非常可怕？有两个原因。首先，它令人感到非常压抑，原因我稍后会一一说明。其次，世界上大概有80%的程序员是内部程序员，如果你从学校毕业的时候不是非常非常小心，你可能会发现不经意间你已经在开发内部软件了。让我告诉你，这种工作会把你榨干。

① Java Enterprise Beans就是EJB（Enterprise JavaBean），指的是Java企业级应用中部署在服务器端的一种组件架构。

② 微软公司总部所在地。

③ 维亚康姆是全球最大媒体娱乐业的公司之一，跨电视、电影、广播、娱乐、出版等行业，名列世界500强。

④ 埃森哲是一家跨国管理咨询公司，提供技术服务和外包服务。它是世界500强公司之一。

好的，那么为什么做一个内部程序员就糟透了？

第一点。你永远无法用正确的方法做事。你总是被迫用最保险的方法做事。聘请程序员的花费是非常昂贵的。一般来说，请埃森哲或IBM这样的公司提供服务，每小时的收费是300美元，而给你提供服务的不过是几个刚刚毕业的耶鲁大学政治科学系研究生，这些人只上过一个6星期课时的.NET编程课程，每年的薪水是4.7万美元，心里盼着混上几年资历再去商学院深造。不管怎样，聘请程序员太贵了，因此无论Ruby语言有多好，也不管Ajax效果有多炫目，他们都不会被允许使用Ruby on Rails开发软件。你只好用微软的Visual Studio，单击软件中的向导，在窗口中将小小的Grid控件拖来拖去，将它与数据库挂钩。不一会儿，你就做完了。既然你做出来的程序已经可以运行了，那就离开这个项目，转向下一个项目吧。这就是为什么不能当内部程序员的第二个原因。因为一旦你的程序可以用了，你就不得不停止开发。只要核心功能具备了，主要问题解决了，在程序上再投资就绝对不值得了，没有商业理由再进一步开发这个软件了。所以，所有那些内部程序看上起就像给狗吃的早餐，只要狗能吃饱就行了，何必再多花钱让食物变得色香味俱全呢？忘掉所有你对自己掌握独门技术的自豪感吧，也不要幻想你有机会操练在CS 323课程中学到的本领。你辛辛苦苦做出来的只是一些令人难为情的次品，然后，你还必须十万火急地为去年制造的次品打补丁。那些次品一年不到就不能正常工作了，原因是程序的编写思路从一开始就不正确。你干了27年这样的活，等到退休的时候，得到的只是一块纪念金表。哦，我说错了，现在的公司连金表也不给了。27年的日子，带给你的只是长期敲击键盘引起的腕管综合症[①]（carpal tunnel syndrome）。现在，换一种情况，如果你在一家专门开发某一种软件商品的公司里工作，比如在专业的软件公司，或者像Google和Facebook那样的互联网公司。你是这家公司的一个程序员，那么你的软件写得越好，它们的销售就会越好。这里的区别是，内部软件只要“能用了”就会停止开发，但是如果你开发的是一个商品，你就可以不断对它加工、打磨、优化和改良。比如，假设你为Facebook工作，你就可以用整整一个月时间优化人名选择部分的Ajax效果，使它变得更快速、更美观。你付出的努力都是值得的，因为这使得你的产品比竞争者的更优秀。所以，开发内部程序不如开发软件商品的第二点原因就是，你做不出优秀的产品。

① 腕管综合症是一系列症状的集合，症状有拇指、食指、中指和半边无名指阵阵麻木、刺痛、灼热等。长期敲击键盘很容易患上这种病。——编者注

第三个原因。如果你在专业的软件公司中编程，你的工作与公司的主营业务直接相关，是能够为公司直接带来收入的。这至少意味着一件事情，就是管理层会想到你。也就是说，你能得到最好的福利、最舒适的办公室和最佳的晋升机会。一个程序员永远都不会有机会成为维亚康姆公司的CEO，但是他很可能会成为技术公司的CEO。

回到原来的话题。离开微软公司以后，我去了维亚康姆公司。原因是，那时互联网刚刚兴起，我想更多地了解它，但是当时微软公司完全不重视互联网。进了维亚康姆公司以后，我就成了一个内部程序员，我做的事与维亚康姆公司任何一个主营业务都毫不相干。

我能看出来，不管互联网业务对维亚康姆公司的未来是多么关键，但是一旦到分配办公桌的时候，内部程序员总是被安排三个人挤在一个小隔间里，而且还是在楼层中阴暗的角落，照不到阳光。而那些“制片人”（说实话，我真不知道制片人到底在干些什么）有点像电视剧*Entourage*中Turtle的角色[①]。制片人都拥有自己独立的宽敞办公室，里面有落地玻璃窗，可以俯视哈德逊河[②]。有一次，在维亚康姆公司的圣诞晚会上，我被引见给集团负责互动业务或类似业务的主管，这个职位是非常高的。他只是泛泛而谈，说与用户有良好的互动对于公司是多么重要、这是未来的方向之类的话，可以发现他对实际业务毫不精通。这使我相信，他毫无新思想，他不知道世界上正在发生什么事情，不知道互联网是什么，也不知道我这样一个程序员到底在干什么。在内心中，他对所有这些事情实际上是有一点恐惧的，但那又能怎样？他一年获得的报酬是200万美元，我到底是一个打字员还是一个“HTML操作员[③]”或是其他什么人，他并不感兴趣。无论我做了什么，无论我的工作有多艰苦，他都不会关心，他会觉得他正在上中学的女儿也能干我的工作。

所以我就辞职了，进入了一街之隔的Juno在线服务公司。这是一家早期的互联网服务提供商，它向用户发放免费的拨号上网账户，但是这种账户只

① *Entourage*是HBO电视网2004年开播的一部电视剧，中文名为《明星伙伴》，内容是关于一个崭露头角的年轻演员如何成为明星的故事。在电视剧中，Turtle是主角Vince儿时一起长大的伙伴。当Vince成名以后，他只顾着在Vince的豪宅享受奢华的生活，不想去工作。

② 哈德逊河（Hudson River）横穿过纽约城。

③ “HTML操作员”在原文中是HTML operator，其中operator一般是指操作机器的人员。但是，HTML实际上是写作网页的语言，只能被编写，不能像操作机器那样被操作。作者使用“HTML操作员”的意思是讽刺对方对互联网完全无知。

能用来收发电子邮件。它与Hotmail或Gmail都不一样，这些当时还不存在。因为你不需要连入互联网，只需要拨一个专门的电话号码就能收发电子邮件，所以它是真正免费的。

Juno对外宣传自己的经费完全靠广告支持。但是，Juno的客户都是每月付不出20美元AOL[①]上网费的客户，因此向这样的客户打广告并不是很吸引人。实际上Juno靠的是有钱投资者的支持。不过，对我来说，Juno至少是一家软件公司，程序员在这里得到了尊重。我很欣赏这家公司的目标，即让每一个人都用上电子邮件。我在那里干了三年，职务是C++程序员。说实话，我过得很愉快。但是，慢慢地，我开始发现Juno的管理哲学很老式。那里的做法就是让经理来告诉每个人该做什么。这是从上到下式的管理，与美国西海岸高科技公司的通常做法正好相反。我在西海岸生活的那段日子让我习惯了这样一种看法，即管理只是一种不得不做、让人厌烦的杂务活，之所以公司需要管理，就是为了不影响聪明人的工作，让他们把事情做完。在一个研究型大学里，系主任实际上是一种负担，没有人真想干，大家都更愿意去做研究。硅谷式风格的管理正是这样。经理存在的唯一理由就是把家俱的位置摆好，不要挡道，只有这样，真正的天才才能做出优秀的成果。

Juno的创始人非常年轻，非常缺乏经验。公司的主席只有24岁，这是他的第一份工作，而且第一份工作就是管理层的工作。他不知从哪里，或许是从一本书、从一部电影或从一部电视连续剧里，得到了这种看法，即经理的任务就是做出各种决定。

有一件事我是非常确定的，那就是管理层获得的技术问题的信息是所有人中最少的。经理绝不应该试图对一切问题做决定。当我还在微软公司的时候，应用软件部门的负责人是Mike Maples，有些程序员在技术问题上争执不下时就去找他做评判。他会玩一会儿杂耍，两只手同时抛接三个球，讲一个笑话，再告诉找他的人立刻离开他的办公室，自己去解决他们那些该死的问题，不要来找他，因为实事求是地说，他是最没有资格做技术决策的人。我认为，管理非常聪明、高度适任的人才就是这样，没有其他方法。但是，Juno公司的经理层把自己看得像美国总统一样，一个个都是决策家。如果你企图决定一切，那么需要做决定的事情实在太多了。所以，那些经理形成了一种我称之为“打了就跑”的抽风式管理方法。突然间，他们不知从何处冲

① AOL（美国在线）是20世纪90年代美国最大的互联网服务提供商，提供拨号上网服务。

了进来，评判一些非常微小的细节问题，比如对话框中的日期应该怎么输入，完全不顾所有开发人员的意见，那些开发人员都是非常合格的技术人才，而且已经在这个问题上工作了几个星期。评判完以后，他们就消失了，对后面的事情不管了。这就是为什么我把这称为“打了就跑”，因为他们想管的事情太多了，在他们眼里到处都是火苗，所以他们不得不充当忙碌的救火队员，东奔西跑地指点工作。

所以，我又辞职了，而且没有真正想好要去干什么。

2007年12月5日，星期三

本文是2007年11月28日在耶鲁大学计算机系演讲的第三部分。

有没有一家公司，在那里程序员被当成天才，而不是被当成打字员？我觉得找不到这样的地方。所以，我决定开一家自己的公司。当时，我目睹许多真地非常傻的人拿着真地非常傻的商业计划书在开互联网公司。我想，嗨，如果那样的人能够开公司，那么我也能开，只要我比他们少傻10%就行了，这好像不是很难。在我的公司里，我们将改变做事的方式。我们尊重每一个程序员，我们制造高质量的产品，我们决不让风险投资家或24岁的总经理指手画脚。我们关心我们的客户，只要他们打电话来，我们就为他们解决问题，而不是将所有责任都推到微软公司头上。我们让客户自己决定是否向我们付款。在Fog Creek软件公司，我们将无条件地同意任何客户的退款请求。这样将使我们保持诚实经营。

那时是2000年的夏天。我开始酝酿Fog Creek软件公司的商业计划书，在准备期间，我经常会给自己放假，去海滩边散步。我开始写一些文章，总结自己在过去工作中的经验教训，写完后放在一个网站上，我给它们起名为“Joel谈软件”（Joel on Software）。那时，网志这种概念还没有被发明，一个名为Dave Winer的程序员开发了一个网站EditThisPage.com，任何人都可以在上面发文章，样式同后来的网志差不多。“Joel谈软件”的访问量增长很快，它成了我的一个讲坛，我把自己对软件开发的想法都写了出来，一些人开始关注我的文章。这些文章里写的很多都不是原创的想法，还包括一些搞笑的内容。这个网站很成功，原因是我用的字体比一般网站大一些，所以读起来比较舒服。网站的准确读者数量总是不太容易统计，尤其是我怕麻烦，

没有放置计数器。但是，通常情况下，我的一篇文章的读者人数在10万到100万之间，具体数字取决于标题是否吸引人。

“Joel谈软件”写的都是与技术相关的内容，这种做法就是我在这里的这个计算机系中学到的。我来告诉你们背后的故事。回到1989年，那时的耶鲁大学在人工智能方面相当强，Roger Schank是这方面最著名的教授之一。有一次，他来到Hillel社团[①]做了一个很短的讲话，内容是他本人提出的人工智能理论以及他开创的一些概念，比如script（脚本）、schema（模式）、slot（槽）等。如今，说实话，我认为他在过去20年中都在发表同样的演说，因为前不久我读了他的书。他花费了自己职业生涯中的20年的时间来根据他的理论写出了一些小程序，目的可能是为了检验理论的正确性，但是那些程序却无法运行，而他也不肯放弃它们。不过，那时他看上去确实像是一个非常聪明的人，我想去上他的课。可是困难是，大家都知道他不喜欢计算机系本科生。所以，只剩下一个办法，那就是去上一门叫做“算法思想”（Algorithmic Thinking）的课程，代码是CS 115，那是他为人文类专业的学生开的选修课，难度不高，很容易通过，根本就是为耶鲁大学学生中的诗人设计的。虽然从课程设置上看，这门课属于计算机系，但是系里规定本专业的学生不能选修。不过，这条规定在报名审核时完全被疏忽了，我才报上了名。它是整个计算机系中学生人数最多的课。每次，当我听到我那些主修历史的朋友们将这门课称为“计算机科学”时，我就会感到非常尴尬。它的常见课后作业是让你写一篇小论文，内容关于机器能否学会思考。所以，你们明白了吧，为什么计算机专业的学生不允许选修这门课。事实上，今天如果系里追查以前的资料，发现我上了这门课并因此收回我的学位，我也不会感到很惊讶。

“算法思想”这门课最大的好处是，你必须写很多东西。一共要写13篇小论文，每周一篇。但是，你不会拿到成绩。等一等，你其实是有成绩的，这里面有一个故事。Roger Schank教授为什么非常不喜欢本科生，原因就是本科生只看重成绩。教授想讨论计算机能否学会思考，但是所有的本科生只想讨论为什么他们的小论文只得到了B而不是A。在学期开始的时候，教授大谈了一通为什么成绩具有误导作用，然后决定学生能够得到的唯一成绩就是一个小小的勾，表示担任助教的研究生已经看过了这份作业。过了一段时间，教授想找出那些真正优秀的作业，所以对于优秀作业，在勾旁边有一个

① Hillel社团是耶鲁大学校内犹太学生的组织，作者Joel就是犹太人。

程序员最可信赖的求职帮手

加号，对于那些实在糟糕的作业，在勾旁边有一个减号。我记得我得到过一次勾旁的加号。但是，具体的成绩从来没有过。

尽管CS 115课程不是为计算机系学生开设的，也无助于拿到学位，但是这门课在我毕业以后却成了对我最有用的课程，因为它要求学生写大量关于技术内容的文章，我因此得到了训练。能不能清晰地写出技术内容的文章决定了你是一个口齿不清的程序员还是一个领袖。我在微软公司的第一份工作是Excel开发小组的程序经理，我为一种叫做“Visual Basic应用程序语言”的庞大编程系统撰写技术文档。这个文档厚达500页，每天早上，几百人上班以后的第一件事情就是阅读我写的文档，根据文档决定下一步该干什么。这些人中包括程序员、测试员、市场部人员、其他文档的作者以及世界各地将程序本地化的人员。我注意到，微软公司中那些真正优秀的程序经理都是具有优秀写作能力的人。微软公司的副总裁Steve Sinofsky曾经写过一封电子邮件《康奈尔大学已开始使用互联网》（Cornell is Wired，www.cornell.edu/about/wired/）。就只靠这封雄辩的电子邮件，微软的公司战略发生了180度大转弯[①]。那些决定游戏规则的人都是善于写作的人。为什么C语言是最流行的语言，原因就是创始人Brian Kernighan和Dennis Ritchie写了一本伟大的书《C程序设计语言》。

最后，我来总结一下前面所讲的内容，就是我在计算机系到底学到了什么。在CS 115课程中，我学到了如何写作；在一次动态逻辑课中，我学到了不要去读研究生；在CS 322课程中，我学到了如何进入UNIX这座宏伟的教堂，熟悉了它的礼仪和教规，花了大量的时间写了大量的代码。当学生拿到计算机科学学士学位毕业的时候，他还没有学到的东西主要是如何开发软件。虽然你们可能在智力上已经得到了很好的锻炼，非常聪明，但是这些还不够，如果你们将来决定以开发软件作为职业，你们在学校里学到的东西会帮到你，但是你们还需要学习别的东西。如果你想学习如何开发软件，方法之一就是把你的简历寄到jobs@fogcreek.com，申请一个Fog Creek软件公司的暑假实习机会。如果你获邀请来实习，到时候我们会教你几招开发软件的方法。

非常感谢各位抽出宝贵的时间坚持听我讲到了最后。

① 1994年，Steve Sinofsk在访问母校康奈尔大学时，看到了学生使用互联网。之后他就发出了一封电子邮件，提醒比尔·盖茨，微软面临着互联网的挑战，只关注桌面软件是不够的。

10

给计算机系学生的建议

2005年1月2日，星期日

大概在一两年前，我还在高喊，有着良好用户体验的Windows图形界面式客户端（rich Windows GUI client）将是未来的潮流。尽管我这样说了，但是时不时地还是有大学生写信给我，问我对于找工作有何建议。既然现在又到了招聘季节，我想我还是把我的标准建议写下来，让那些大学生读一读，笑一笑，然后忘掉。

大多数大学生都很自以为是，从不会虚心向前辈求教，他们觉得那样太麻烦。但是，很幸运，在计算机领域，这样做是对的。因为他们的前辈很可能会说一些不靠谱的话，比如“到2010年之前，对纸带打孔员的需求将超过1亿人”，还有“目前Lisp语言的相关人才非常抢手”。

我也不能例外，当我在给大学生提供建议时，我完全不知道我在说些什么。我已经无可救药地属于过时人物了。我真地搞不清楚AIM[①]是什么，我仍然在使用一种老掉牙的叫做Email的东西，真是太恐怖了。那玩意流行的年代，人们听音乐还是用一种又扁又圆叫做CD的小圆盘。

所以，你最好直接漠视我在这里说的话，将时间用来开发某种可以使其他学生找到约会对象的在线软件上。

尽管如此，我还是会说出我的建议。

① AIM是AOL Instant Messenger的缩写，也就是AOL出品的即时通信软件。它的首次发布是在1997年5月。据统计，在2006年，它占据了美国即时通信市场52%的份额。

如果你喜欢编程，那么你真是受到了上天的眷顾。你是非常幸运的少数人之一，能够以自己喜欢的事谋生。大多数人没有这么幸运。你认为理所当然的观念“热爱你的工作”，其实是一个很现代的概念。通常的看法是，工作是一种让人很不开心的事，你为了拿工资才不得不去上班。你工作的目的是为了攒下钱去干那些自己真正喜欢干的事，但是前提是你得等到65岁退休之后才行，而且还有不少条件。条件一，你的积蓄必须足够多；条件二，你没有老到走不动，你还有体力去干那些事情；条件三，你喜欢的事情不需要用到脆弱的膝盖、昏花的视力，也不要求你走上一里地不喘气，等等。

我刚才说到哪里了？对了，我要提建议。

二话不说，下面就是Joel针对计算机专业学生的7条免费建议。（绝对超值哦。）

(1) 毕业前练好写作。

(2) 毕业前学好C语言。

(3) 毕业前学好微观经济学。

(4) 不要因为枯燥就不选修非计算机专业的课程。

(5) 选修有大量编程实践的课程。

(6) 别担心所有工作都被印度人抢走。

(7) 找一份好的暑期实习工作。

我会一一解释这7条建议。如果你头脑简单到我说什么你就做什么，那么你就不必读下去了。在这种情况下，我还要加上一条：

(8) 寻求专业人士的帮助，培养你的自信心。

毕业前练好写作

如果不是Linus Torvalds不断地散布福音，请问Linux操作系统会成功吗？虽然他是一个非常聪明的计算机天才，但是Linux吸引来全世界一大批志愿者的真正原因却是Linus Torvalds的表达能力。他通过电子邮件和邮件列表用书面形式传播自己的想法，最终引起了所有人的注意。

你听说过现在风靡一时的“极限编程[1]”（Extreme Programming）吗？我在这个地方不谈我对极限编程的看法，我只说如果你听过这个词，那么原因就是它的倡导者都是一些非常有才华的作家和演说家。

即使我们缩小范围，将目光局限在任何一个软件开发团体中，你也会发现该团体中最有权势和影响力的程序员正是那些表达能力强的程序员，他们无论是做书面表达还是做口头表达，都能够清晰、自如、具有说服力地传达观点。此外，长得高也有助于提升影响力，不过这个不取决于你。

一个普通程序员与一个优秀程序员的区别，不在于他们懂得的编程语言谁多谁少，也不在于他们喜欢用Python语言还是喜欢用Java语言，而在于他们能否与他人交流思想。如果你能说服其他人，你的力量就可以得到放大。如果你能写出清晰的注释和技术规格说明书，其他程序员就能够理解你的代码，因此他们就能在自己的代码中使用，而不必重写。如果你做不到这一点，你的代码对其他人就没有价值。如果你能为最终用户写出清晰的使用手册，其他人就能明白你的代码是用来干什么的，这是唯一让别人明白你的代码有何价值的方法。SourceForge[2]上有许多优美的、有用的代码，但是它们都像被埋葬了一样，根本没人来用，原因就是它们的作者没有写好使用说明（或者压根就没写）。这样一来就没有人知道他们的成果，他们杰出的代码就衰亡了。

如果一个程序员不会用英语写作、没有良好的写作能力，我就不会雇他。如果你能写，不管你去哪家公司工作，你很快就会发现写作技术文档的任务会落到你头上，这意味着你已经开始在放大自己的影响力了，管理层正在注意到你。

大学里有些课程被公认为“写作密集型”（writing intensive）课程，这就是说为了拿到学分，你必须写作多得可怕的文字。一定要去上这样的课程！不要管学科，只要这门课每周甚至每天都要你写东西，你就去上。

你还可以动手写日记或者网志。你写得越多，写作就会变得越容易。写

① 极限编程是一种软件工程的方法论，之所以称为“极限”，是因为这种方法提倡将一些公认的软件开发的“最佳原则”都发挥到极限，追求软件开发的最佳效果。

② SourceForge.net是一个代码仓库，任何个人或团体都可以在上面免费发布开放源码的项目，访问者可以免费得到这些代码或软件。

起来越容易，你就会写得越多。这是一个良性循环。

毕业前学好 C 语言

第二点我要讲的是C语言。请注意，我说的是C语言，而不是C++。虽然在实际使用中C语言已经越来越罕见，但是它仍然是当前程序员的共同语言。C语言让程序员互相沟通，更重要的是，它比你在大学中学到的“现代语言”（比如ML语言、Java语言、Python语言或者其他正在教授的流行垃圾语言）都更接近机器。你至少需要花一个学期来了解机器原理，否则你永远不可能在高级语言的层次写出高效的代码。你也永远无法开发编译器和操作系统，而它们恰恰属于目前程序员能够得到的最佳工作之列。别人也永远不会放心将大型项目的架构设计交给你。我不管你懂多少延续（continuation）、闭包（closure）、异常处理（exception handling），只要你不能解释为什么`while (*s++ = *t++);`这句代码的作用是复制字符串，或者不觉得这是世界上对你来说再自然不过的事情，那么你就是在盲目无知的情况下编程。在我看来，这就好像一个医生不懂得最基本的解剖学就在开处方，他看病的根据完全是因为那些娃娃脸的医药厂商销售代表说这种药有用。

毕业前学好微观经济学

如果你没有上过任何经济学课程，那么我首先来做一个超短的评论：经济学是这样的学科之一，刚开始学的时候轰轰烈烈，有许多有用的、言之有理的理论和可以在真实世界中得到证明的事实，等等；但是，再学下去就每况愈下，有用的东西就不多了。经济学一开始那个有用的部分正是微观经济学，它是商业领域所有重要理论的基础。跟在微观经济学后面的东西就不行了。你接下来学的是宏观经济学，如果你愿意，尽管跳过去，也不会有什么损失。宏观经济学开头的部分是利息理论，内容比方说是利率与失业之间的关系，但是怎么说呢，看上去这部分里面还没有被证实的东西多于已经被证实的东西。学完这部分，后面的内容越来越糟糕，许多经济学专业的学生实

际上都变成在搞物理学，因为这样才能在华尔街上找到更好的工作。但是不管怎样，你一定要去学微观经济学，因为你必须搞懂供给和需求，你必须明白竞争优势，你必须理解什么是净现值（NPV），什么是贴现，什么是边际效用。只有这样，你才会懂得为什么生意是现在这种做法。

为什么计算机系的学生也应该学经济学？因为，从经营一家公司的角度来看，比起那些不懂的程序员，一个理解基本商业规则的程序员将会更有价值。就是这么简单。我无法告诉你有多少次我是那样地充满挫折感，因为我看到了太多的提出一些疯狂的想法的程序员，这些想法在代码上也许可行，但在资本主义世界中毫无意义。如果你懂得商业规则，你就是一个更有价值的程序员，你会因此得到回报的，但是前提是你要去学习微观经济学。

不要因为枯燥就不选修非计算机专业的课程

想提高GPA[①]绩点的一个好方法就是多选修非计算机系的课程。请千万不要低估你的GPA的重大意义。千千万万的人事经理和招聘人员在拿到一份简历的时候，第一眼就会去看GPA，包括我也是这样。我们不会为这种做法道歉。为什么？因为GPA不反映单个的成绩，而是代表了许多个教授在一段很长的时间中，在不同的情况下，对你的表现的一个总的评估。SAT[②]成绩难道不够吗？哈，那只不过一场几个小时的测试罢了。GPA中包括了四年大学期间你的小论文、期中考试和课堂表现，总数有几百次之多。当然，GPA也有自己的问题，不是百分之百准确。比如，这些年来，老师对学生的打分越来越宽松，学习成绩有通货膨胀的趋势。再比如，GPA无法反映课程的难度，没人能够看出你的GPA是来自无名社区大学家政系的轻松课程还是来自加州理工学院针对研究生的量子力学课程。渐渐地，我形成了一套自己的做

① GPA（Grade Point Average的缩写）是高校中衡量学生平均成绩的一种手段，将各种课程的成绩换算为一个绩点，然后再求加权平均。常见的GPA为4分制，A（90~100分）为4分。有的学校会将A+算为4.3分。

② SAT（Scholastic Aptitude Test和Scholastic Assessment Test的缩写，“学习能力测试”）是美国的一项标准化考试，通常用来评估高中毕业生的水平。美国高校采用这个成绩作为录取参考。

法，首先我会过滤掉所有来自社区大学、GPA低于2.5的简历，然后我会要求剩下的人给我寄成绩单和推荐信。我再从中发现那些成绩一贯优秀的人，而不是那些仅仅在计算机系课程中得到高分的人。

为什么我要关心某人的“欧洲历史”课程成绩呢，毕竟作为雇主我要找的应该是程序员啊？何况，历史是那么枯燥，不得高分很正常。哦，这么说来，你的意思是我应该雇用你，而不用考虑一旦工作变得枯燥你会不会努力工作？别忘了，在编程工作中也有很枯燥的东西。每一项工作都有枯燥难耐的时刻。我不想雇用那些只想干有趣事情的人。

在大学里，我选修过一门叫做“文化人类学”（Cultural Anthropology）的课程。因为我很好奇，想知道这门课会讲什么，想学一点关于人类学的知识，看上去这很像一门有趣的概论类课程。

这门课有趣吗？连有趣的边都沾不上！我不得不阅读那些乏味到极点的书籍，内容有关巴西雨林中的印第安人和特洛布里安德[①]（Trobriand）岛上的居民。恕我直言，我对这些东西一点兴趣也没有。有一次，听课听到一半，实在是太无聊了，我渴望干一些更有意思的事情，比如看着窗外青草在长高。我对这个学科已经毫无信心了。完全地，彻底地，不想学下去了。我的眼睛酸疼，课堂上正在无休止地讨论堆山药，我彻底厌倦了。天知道为什么特洛布里安德岛上的居民要花那么多时间来堆山药？其他的事情我都已经忘记了，总之，这门课超级乏味。但是，那天讲的东西期中考试时肯定会考。所以，我忍住了，继续往下学。我逐渐下定决心，就把文化人类学当成我的抗无聊免疫剂，这是我的个人万米障碍跑的训练场，专门训练如何对抗无聊。如果我想在这门课的考试中得到A，那么就连印第安人在冬庆节[②]（potlatch）中使用的毯子，我都必须知道得一清二楚。如果我做到了，那么我就无敌了，以后天底下再无聊的东西我都能够对付。后来，出于偶然我坐在林肯中心[③]听完了整整8个小时瓦格纳的歌剧《指环》[④]（*Ring Cycle*）。幸亏我在文化人类

① 特洛布里安德群岛是南太平洋中的一个由珊瑚礁构成的群岛，面积为170平方英里，靠近新几内亚的东海岸。

② 冬庆节是北美洲太平洋沿岸的一些印第安部落的重大节日之一，人们在这个节日中往往互赠礼物。

③ 林肯艺术中心是纽约最著名的文化表演场所之一，建于20世纪70年代。

④ 歌剧《指环》全名为《尼伯龙根的指环》（*The Ring of the Nibelung*），是德国音乐家瓦格纳（Wilhelm Richard Wagner，1813—1883）的巨作。全剧共分4部，全部演完需要4天，每天8个小时。

学中已经尝过研究夸扣特尔人[1]（Kwakiutl）的滋味，两相比较，我觉得坐着听8个小时歌剧可谓愉快经历。

最后，我在那门课中得到了A。如果我能做到，你也一定能做到。

选修有大量编程实践的课程

我依然清楚记得我发誓绝不读研究生的那一刻。那是在一门叫做“动态逻辑”的课程上，教师是活力十足的耶鲁大学教授Lenore Zuck，她是计算机系那些聪明的老师中最聪明的人之一。

如今，由于记忆力糟糕，我已经差不多把这门课的内容忘光了，但是不管怎么说，在这里我还是想要对付着说一下。大致上，形式逻辑的意思是说，如果条件成立，你就能证明结论也成立。比如，根据形式逻辑，已知“只要成绩好，就能被雇用”，然后假定“Johnny的成绩好”，你就可以得到一个崭新的结论“Johnny会被雇用”。这完全是经典方法。但是，一个解构主义者（deconstructionist）只需要10秒钟就能破坏形式逻辑中所有有用的东西。这样一来，留给你的只是一些趣味性，而不是实用性。

现在再来说动态逻辑。它与形式逻辑其实是一回事，但是必须再多考虑时间因素。比如，“你打开灯之后，就能看见自己的鞋子”，已知“灯以前是亮的”，那么这就意味着“你看见了自己的鞋子”。

对于像Zuck教授那样聪明的理论家，动态逻辑充满了吸引力，因为它看上去很有希望让你在形式上证明一些计算机程序的相关理论问题。这样做说不定很有用。比如，你可以用它在形式上证明，火星漫游车的闪存卡不会发生溢出（overflow）问题，不会因而整天一遍又一遍地重启，耽误了它在那颗赤红色的星球上漫游寻找火星人马文[2]（Marvin the Martian）。

在第一堂课上，Zuck博士写满了整整两面黑板，甚至黑板旁边的墙上都写上了很多证明步骤。需要证明的问题是，有一个控制灯泡的开关，现在灯

① 夸扣特尔人是太平洋东北海岸的美洲印第安人，主要生活于温哥华岛上。

② “火星人马文”是美国华纳兄弟公司著名动画片Looney Tunes的一个角色。

泡没有亮，这时你打开了开关，请证明灯泡将会点亮。

整个证明过程复杂得不可思议，处处都是陷阱，必须十分小心。保证这个证明不出错太困难了，还不如直接相信打开开关灯就会亮。真的，虽然证明过程写满了许多块黑板，但是还是有许多中间步骤被省略了，因为如果要从形式逻辑上完整证明所有步骤，那就琐碎得无法形容了。许多步骤是用各种经典的逻辑证明方法推导得到的，包括归纳法、反证法等，甚至有些部分还是由旁听的研究生证明的。

留给我们的课后作业是证明逆命题：如果灯原来是关着的，现在却亮了，那么请证明开关的状态一定同原来相反。

我动手开始证明，我真地去证明了。

我在图书馆里待了很长时间。

我对照着Zuck博士的原始证明想依样画葫芦。研究了几个小时之后，我在其中发现了一个错误。可能我抄写的时候抄错了，但是这使得我想通了一件事。如果花费3个小时，写满了一块又一块的黑板，每一秒钟都可能出错，最后能够证明的却只是一个很琐碎的结论，那么这种方式有多大的实用性呢？在活生生、充满趣味的现实世界中，你永远都不会有机会使用它。

但是，动态逻辑的理论家们对这一点不感兴趣。他们看上它不是因为它有用，而是因为它可以为他们带来终身教职。

我放弃了这门课，并且发誓绝不会去读计算机科学的研究生。

这个故事告诉我们，计算机科学与软件开发不是一回事。如果你真地非常幸运，你的学校可能会开设很像样的软件开发课程。但是另一种可能是，你的学校根本不教你在现实中如何编程，因为精英学校都觉得，教授工作技能最好留给职业技术学校、犯人重返社会的培训项目去做。你到处都能学怎么写代码。别忘了，我们是耶鲁大学，我们的使命是培养未来的世界领袖。你交了16万美元的学费，却在学循环语句的写法，这怎么可以？你以为这是什么地方，难道是机场沿途的酒店里临时拼凑起来不靠谱的Java语言培训班？哼哼。

麻烦在于我们没有一种真正教授软件开发的专门学校。你如果想成为一

个程序员，你可能只能选择计算机科学专业。这是一个不错的专业，但是它同软件开发不是一回事。

不过，你运气好的话，计算机系会开出许多有大量编程练习的课程，你可以选修它们。这就好比历史系里有许多课程要求你大量写作，你因此学会了如何写作一样。它们是你能找到的最佳课程。如果你喜欢编程，可是学校里上的都是λ演算或者线性代数这种连电脑的边都摸不到的课程，假定你不明白这些课程的意义，你也不要因此心情恶劣。在那些400等级的课程代号中，去寻找名称中带有“Practicum[①]”这个词的课程。不要被这个拉丁语单词吓倒，这些都是有用的课程，之所以起这种名字，只是为了让那些文绉绉、装腔作势、满嘴胡说八道的公司经理们觉得高深莫测。

别担心所有工作都被印度人抢走

我首先要说的是，如果你本身就已经在印度了，或者你就是印度人，那么你真地毫无必要去想这件事，根本不用琢磨所有的工作机会是不是都跑到了印度。那些都是非常好的工作，好好地享受吧，祝你身体健康。

但是，我不断听说计算机系的入学人数下降得很厉害，已经到了危险的程度。根据我听到的说法，其中的一个原因是“学生们不愿去学一个工作机会都流向印度的专业”。这种担心大错特错，有很多理由可以反驳。首先，根据一时性的商业潮流决定个人的职业选择，这是愚蠢的。其次，即使编程工作无一幸存地都流向了印度和中国，但是学习编程本身依然是一种第一流的素质训练，可以为各种超级有趣的工作打下基础，比如业务流程工程（business process engineering）。再次，不管是在美国还是在印度，真正优秀的程序员依然是非常非常短缺的，这一点请相信我。不错，确实有相当一批失业的IT从业者在那里鼓噪，抱怨他们长时间找不到工作，但是你知道吗？即使冒着触怒这些人的风险，我还是要说，真正优秀的程序员根本不会失业。最后，你还能找到更好的专业吗？你觉得什么专业好？主修历史学？如果那

① Practicum指大学中供学生实习的课程。

样，你毕业的时候就会发现，根本没有其他选择，只能去法学院。不过我倒是知道一件事：99%的律师都痛恨他们的工作，痛恨他们当律师的每一分钟。可是，律师每周的工作时间偏偏长达90小时。就像我前面说过的：如果你喜欢编程，那么你真是受到了上天的眷顾。你是非常幸运的少数人之一，能够以自己喜欢的事谋生。

不过说实话，我不觉得学生们真地有上面的想法。近年来，计算机系入学人数的下降只是回到了历史上的正常水平，因为前些年的互联网狂热使得入学人数出现了大泡沫，抬高了基数。由于这种泡沫，许多并不真地喜欢编程的人也来读计算机系。他们心里想的是，只要进了计算机系，将来就能找到诱人的高薪工作，就能获得24岁当上CEO、进行IPO的机会。谢天谢地，这些人现在都离计算机系远远的了。

找一份好的暑期实习工作

精明的招聘负责人都知道，喜欢编程的人高中时就将牙医的信息输入了数据库，进入大学前就去过三次电脑夏令营，为校报做过内容管理系统，有过软件公司的夏季实习经历。招聘负责人就是要在你的简历上找这些东西。

如果你喜欢编程，就不要随便什么工作都答应，否则你会犯下最大的错误。不管是暑期工作，还是兼职或者其他性质的工作，只要与编程无关，就不要轻易接受。我知道，其他19岁的孩子都想去购物中心里打工，在那里折叠衬衫。但是你与他们不同，你19岁时就已经掌握了一门非常有价值的技能。将时间浪费在折叠衬衫上是很愚蠢的，等到毕业的时候，你的简历上本应该写满了一大堆与编程相关的经历。就让那些财经类的毕业生去租车公司“帮助人们满足他们租车的需要”吧，你要干的是别的事（在电视中扮演超人的Tom Welling[3]除外）。

为了让你的生活变得更容易一些，也为了强调这整篇文章完全是为了满足我的个人目的，我要告诉你，我的公司——Fog Creek软件公司——提供

③ Tom Welling是一个美国演员，在电视剧《超人前传》（*Smallville*）中扮演超人。该电视剧讲述还没有成长为后来超人的克拉克·肯特少年时代的故事。

软件开发方面的暑期实习机会。我们非常看重简历。“比起其他公司的实习工作，你在Fog Creek最有可能学到更多的编写代码、软件开发、商业运作方面的知识。”这是去年夏天我们的一个实习生Ben说的。他会这样说，并不完全是因为我派了人到他的宿舍让他这样说。我们接受实习申请的截止日期是2月1日。一起来吧。

如果你听从了我的建议，你还是有可能落得一个悲惨的下场，比如很早就卖掉了微软公司的股票，再比如拒绝了谷歌公司的工作机会，原因是你想要一间自己的可以关上门的独立办公室，或者做出了其他生命中愚蠢的决定。但是，这些可不是我的错。我一开始就告诉过你，不要听我的话。

第三部分 设计的作用

11 字体平滑、反锯齿和次像素渲染

12 寸土必争

13 大构想的陷阱

14 别给用户太多选择

15 易用性是不够的

16 用软件搭建社区

11

字体平滑、反锯齿和次像素渲染

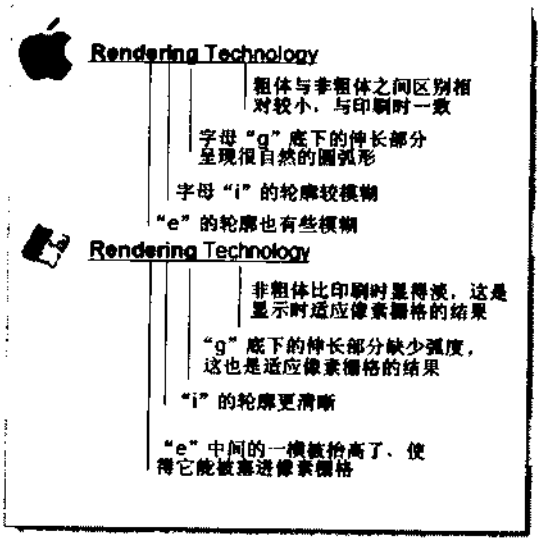
2007年6月12日，星期二

对于如何在计算机屏幕上显示字体，苹果公司和微软公司总是有不同的看法。目前，这两家公司都使用次像素渲染（subpixel rendering）技术，使得字体在低分辨率的屏幕上也能显得很清晰。这两家公司的根本不同之处在于指导思想。

- 苹果公司通常认为，字体算法的首要目的是尽可能多地保持原始设计的样子，即使有损屏幕显示的清晰性，也在所不惜。
- 微软公司通常认为，字体的形状一定要适应像素的限制，要保证屏幕显示不模糊、容易辨识，即使字体的形状因此背离原始设计，也在所不惜。

现在，Safari[①]的Windows版本已经发布了。这个软件克服了许多困难，才将苹果公司的字体渲染算法用到了Windows操作系统上。这实际上给了你一个机会，在同一台显示器上直接比较两种不同的字体哲学。这有助于你理解我下面要举的例子。我想，通过比较，你会注意到两者的不同。苹果系统的字体给人有一点毛茸茸的感觉，边界不是很清晰。但是，从计算机屏幕上看，在不同的字体族（font family）之间，它会显示出更多的变化。原因是苹果公司的渲染算法比微软公司更忠实于字体的原始设计，能够像印刷品那样在高清晰度状态下呈现出字体设计中的细微差别。（关于字体在屏幕上的表现，请访问http://www.joelonsoftware.com/items/2007/06/12.html。）

① Safari是苹果公司Mac OS X操作系统的内置浏览器。



这种差别的源头来自于苹果公司的历史传统，苹果公司一贯非常重视桌面出版和平面设计。它的算法有一个好处，就是当你在计算机中打开针对印刷品的设计稿时，你在屏幕上看到的与最终印刷出来的样子很接近。当你判断一个文本区域的颜色深浅时，这个特性尤其有用。微软公司的处理方法是让字体尽量适应像素的分布，这意味着微软其实不在乎让字母毛茸茸的边缘被一根根细线替代，即使因此文本在屏幕上显示的颜色比印刷时浅也无所谓。

微软的处理方法也有一个好处，就是有利于对着屏幕阅读。微软公司信奉实用主义的观点，认定字体的原始设计并不是神圣不可侵犯的，锐利的、便于阅读的屏幕显示更为重要，不一定非要拘泥于字体设计师对文本颜色深浅的原始安排。这就是说，微软公司设计出的字体主要是为了屏幕显示，比如Georgia字体和Verdana字体，它们完全按照像素的位置设计，在屏幕上确实很漂亮，但是印刷出来后却乏善可陈。

苹果公司正好相反，它选择风格化的路线，将艺术性置于实用性之上。原因很简单，就是苹果公司的创始人乔布斯很有品味，而微软公司宁愿选择一条风险较小的路线，这条路线采纳实用主义观点，只要能够达到使用目的就行，完全没有让人眼前一亮的闪光点。这么说吧，如果苹果公司是Target

连锁超市，那么微软公司就是沃尔玛[①]。

好了，现在到了讨论关键问题的时候：用户喜欢哪一种处理方法？Jeff Atwood写过一篇文章，对这两种字体技术进行了逐一比较（www.codinghorror.com/blog/archives/000884.html）。不难想像，他的文章引起了热烈的反响。苹果公司的用户喜欢苹果的方法，微软公司的用户喜欢微软的方法。这并不完全都是粉丝情结（fanboyism）。它反映了一个事实，如果你问人们喜欢什么样的风格和设计，除非这些人受过专门训练，否则他们一般会选择自己最熟悉的品种。在最常见的品味这个问题上，如果你做一个偏好调查，你会发现大多数人根本不知道应该怎么选择，他们只好选一个自己最熟悉的答案。这种现象到处都是，不管是字体应用或平面设计，还是选购银器（人们挑选小时候用过的样式）。除非受过专门训练，明确知道自己想要什么，否则人们挑中的就是自己最熟悉的。

这就是为什么当苹果公司的工程师们将苹果机上的软件移植到Windows系统上时，他们可能会觉得自己正在为Windows用户做出巨大贡献，将自家“优越的”字体渲染技术提供给其他操作系统的用户使用。这也解释了为什么Windows用户通常认为Safari上的字体有点模糊，看上去怪怪的。这些用户不知道原因，他们只是不喜欢这样。实际上，他们心里想的是：“哇！这和我使用的系统不一样。我不喜欢有差异。为什么我对这些字体没有好感？嗯，当我靠近仔细一看，这些字体有点模糊。这肯定就是我不喜欢它们的原因了。”

① 按照销售额计算，沃尔玛是美国最大的零售商，以购物方便、商品齐全而著称。Target连锁超市则是美国第五大零售商，经常会搞一些别出心裁的推广活动。比如，2006年感恩节前，Target请来著名魔术师大卫·布莱恩，把他吊在时代广场上空的一个圆球里旋转两天，让他表演如何自己解脱，这吸引了成千上万的游人，而且被电视、报纸、互联网等各类媒体广为报道。

12

寸土必争

2007年6月7日，星期四

“谁把收音机忘在浴室里了？”我问杰瑞德。远远地传来一阵古典音乐。

“没有。声音是外面传进来的。前几天你不在的时候就有了，每天晚上都不例外。”

我们生活在纽约的一幢公寓里，上下左右都有邻居。我们都习惯了听着楼下传来的电视声。我们头顶上的那家人有一个小孩，他最喜欢的游戏就是将一大把玻璃弹子扔到地板上，然后砰一声把自己也往地板上一摔。我不知道你会不会记仇，但我会。就在我写下这些话的时候，那个孩子正从一个房间冲到另一个房间，一路上发出各种碰撞声。我肯定忍不到他长大学会玩彩弹枪（paintball）的那一天。

不过，深夜传来古典音乐，以前从来没有发生过这种事。

更糟糕的是，音乐的内容是某一首充满浪漫主义精神的“狂飙突进”派[①]（Sturm-und-Drang）作品。在我要入睡的时候播放这样的作品，真是让人生气。

终于，音乐声停止了，我慢慢进入了梦乡。但是第二天晚上，音乐声又在午夜时分响起了，这一次我真地精疲力竭了。音乐的内容换成了更自以为是的瓦格纳的作品，充满了华丽的渐强式（crescendo）旋律，每当我刚刚睡

① 狂飙突进运动是18世纪60年代到80年代的一场德国思想运动，主要发生在文学和音乐领域，强调艺术要表达激烈极端的情感，在表现形式上以慷慨激昂而著称。

着，旋律就骤然升高，一次又一次将我惊醒。我别无选择，只好起床，坐在客厅里，看着搞笑猫[1]的图片发呆。音乐不再响起的时候——这个时刻最终还是到来了，已经是凌晨1点了。

次日晚上，我觉得自己受够了。当音乐在午夜响起时，我穿好衣服，开始侦查整幢公寓楼。我在走廊里慢慢移动，在每扇门口倾听，想搞清楚音乐到底是从哪里发出来的。我从每一扇窗户探出头，终于发现有一扇通向楼内通风井（airshaft）的门没有锁，那里的音乐声响得惊人。我在通风井中的逃生楼梯上爬上爬下，在每一层平台上，我把耳朵贴近每一个窗口，直到我确信问题出在亲爱的C老太太家中，她住在2B号房间，在我们房间的正下方。

程序员最可信赖的求职帮手

C老太太大概已经有60多岁了，我觉得她不可能那么晚还醒着，独自一人听音乐。一想到本地的古典音乐电台播放歌剧《尼伯龙根的指环》之类的作品时C老太太就精神矍铄、如痴如醉、夜夜不眠地一人独自熬夜欣赏的情景，我自己都被逗乐了。

不可能发生这种事。

我注意到，音乐声每天晚上从午夜12点响起，到1点结束。这使得我怀疑那可能是一台闹钟型收音机，出厂时默认的响铃时间设置是午夜12点。

程序员最可信赖的求职帮手

把楼下的一位老妇人叫醒，告诉她我只是怀疑不明音乐声出自她的屋里，我做不出来这种事。我垂头丧气地回到了自己的房间，闷坐着看网上漫画*xkcd*[2]。我很沮丧，也很生气，因为我没解决问题。接下来的一整天，我都心情恶劣，郁郁寡欢。

第二天晚上，我敲响了C老太太的房门。大楼管理员事先已经告诉我，她明天就要离开了，整个夏天都不会待在纽约。所以，如果问题真地出在她的房间里，我最好能够快速确定。

“不好意思，打扰您了，”我说，“我发现每天晚上，大概接近午夜的时候，

① 搞笑猫指在猫的照片上配上各种幽默的话，形成滑稽效果。它的英文单词是lolcat，即lol（放声大笑）和cat（猫）的结合。

② *xkcd*是一个由Randall Munroe创作的流行系列漫画，只在网站上发表，网址是http://xkcd.com/。

我们房间后面的通风井里都会传来声音很响的音乐声。这让我无法睡着。”

“哦，你搞错了，不是我发出的！”听到我对她的怀疑，她坚决否认。当然她没说谎，肯定不是她发出的，她可能早早就上床睡了，不会把收音机开得响响的，干扰邻居的睡眠！

我猜想她的耳朵不是很好，可能永远不会听到每天午夜她的备用间里有大声喧哗的声音。另一种可能是，她是那种睡得很死的人。

我们争了几分钟，最后我说服她检查一下斜对着我的窗户的那个房间，看看里面是否有类似闹钟型收音机的东西。

结果证明真有。就放在一扇打开的窗下，那扇窗正对着我的卧室。我拿起收音机，发现它的收听频率设在FM 96.3兆赫，那正是纽约播放古典音乐的WQXR电台的频率。我知道我找到罪魁祸首了。

“啊，这个东西？我不知道怎么用，我从来没有用过它，”她说，“我以后要完全切断它的电源。”

“没必要这样做。”我说。我关掉了闹钟功能，将音量设为零。并且由于我是迟发型强迫症（late-onset OCD）患者，我就顺便帮她将钟调到了正确的时间。

C老太太感到万分抱歉，但这真不是她的错。为了搞懂如何操作这该死的闹钟型收音机，我——请看清楚，操作的人是我！——琢磨了半天。嗨，老弟，让我来告诉你，我对闹钟型收音机有意见。它的用户界面太可怕了。一个普通的老太太根本不可能搞懂。

那么这是不是闹钟型收音机的错呢？一定程度上是这样的。它太不好用了。它的铃声会每天自动响起，即使前一天根本没人碰过它，这种设计思想可不怎么高明。而且，根本没理由将重启后的响铃时间默认设在半夜12点，清晨7点才是一个合情合理的设置。

不知怎么回事，最近几个星期我变得特别挑剔。我从每一件东西中寻找缺陷，一旦发现，我就不可抑制地想要纠正它。这是一种特殊的心理状态，实际上，当软件开发者进入新产品的最后测试阶段，他们就是这样一种心理状态。

最近几个星期，我一直在写FogBugz下一个大版本的所有文档。我一边写作，一边试用软件的新版本，目的是为了确认软件能够按照预想的方式运行，也为了抓取截屏图片。几乎每小时警报都会响起。“等一下！怎么回事？不应该有这样的结果啊！”

不过，幸好这是软件，我总是可以找到改正错误的办法。哈哈！我在开玩笑。我已经完全看不懂代码了。我只是提交一个bug，然后让相关人员修正。

Dave Winer说过：“创造一个有使用价值的软件，你必须时时刻刻都在奋斗，每一次的修补，每一个功能，每一处小小的改进，你都在奋斗，目的只是为了再多创造一点空间，可以再多吸引一个用户加入。没有捷径可走。你需要一点运气，但是这不取决于你是否幸运。你之所以会有好运气，那是因为你寸土必争。”（www.scripting.com/2002/01/12.html）

商业软件——那种你卖给别人的软件——就是一种寸土必争的游戏。

每天你前进一小步，将一件东西做得比昨天好一点点。你把闹钟的默认时间定在清晨7点，而不是半夜12点。这样的改进几乎看不出可以让谁获益，几乎没有变化。但是，你前进了一小步。

有无数个要做的这样微小的改进。

为了发现可以改进的地方，你必须有一个思维定势，始终如一地用批判的眼光看世界。随便找一样东西，如果你看不出它的缺点，那么你的思维转型还没有成功。当你成功的时候，你身边亲密的人会被你逼得发疯。你的家人恨不得杀了你。当你步行上班的时候，看到一个司机漫不经心地开车，你几乎用了所有的意志力才勉强忍住不冲上去告诉那个司机，他这样开车差点儿要了旁边坐在轮椅上的那个可怜小孩的命。

当你改正了一个又一个这样的小细节后，当你磨光、定型、擦亮、修饰你的产品的每一个小边角后，就会有神奇的事情发生。厘米变成分米，分米变成米，米变成了千米。你最后拿出来的是一件真正优秀的产品。它第一眼就让人觉得震撼，出类拔萃，工作起来完全符合直觉。就算100万个用户中有一个用户某天突然要用到一个他100万次使用中才会用到一次的罕见功能，他发现了这个功能不仅能用，而且还很美：在你的软件中，即使是看门

人的小屋都铺着大理石的地板，配有实心的橡木门和桃花心木的壁板。

就是在这个时候，你意识到这是一个优秀软件。

我要向FogBugz 6.0开发团队表示热烈祝贺。他们在这种寸土必争的游戏中是优秀得惊人的选手。今天，他们提交了第一个beta版本，一切都在按部就班地进行。夏天结束的时候，他们将发布正式版本。这是他们迄今为止的最佳作品。它将震撼你。

13

大构想的陷阱

2007年1月21日，星期天

本文是对Scott Rosenberg的《梦断代码》（*Dreaming in Code*）一书的评论。

眼睛的工作方式同页错误[①]（page-fault）机制有类似之处。它运作起来如此完美，以致于你都不会觉察到它是怎么运作的。

看东西的时候，你的视力只是在视野中很小的一块区域是高分辨率的，而且视野的正中央还有相当大的一个盲点。但是，你依然想当然地认定你能够超清晰地看清视野中的每一个角落。为什么会这样？因为你的眼睛能够非常快速地移动，通常情况下，你的眼睛能够按照你的意愿瞬息之间就将视点从一处移到另一处。你的神经系统将这个眼睛移动的过程完全抽象掉了，这使得你产生了一眼看清一切的错觉。实际上，只有很小一块区域在你眼中才真的是高分辨率的，其他更大的区域在你眼中的分辨率是极低的。你拥有能够快速处理视觉过程中页错误的能力。它的处理速度非常之快，快到你从没有怀疑过自己看到的是整个图景，你觉得所有的画面都被投射到你大脑中那小小的屏幕上。

这个机制非常非常有用，类似的过程还发生在其他许多地方。你的耳朵在谈话的重要部分会自动变得更加灵敏。你的手指在接触到其他东西（比如一件上好的美利奴羊毛衫）的时候，或者伸进鼻孔中的时候，会自动抚摸一下，因此你从局部获得了对整体的感觉。当你做梦的时候，你的意识会让你问出同醒着时一样的问题。（这是什么？让我来看一看！）但是这时，你的理

① 页错误指的是应用程序读取物理内存的某段地址时，其中没有包含数据，导致出错。

智处于暂时关闭状态（毕竟你是在睡梦中），所以你得到各种奇奇怪怪的答案，进而在你的大脑中，它们组合成一个荒诞的故事，也就是人们所说的“梦境”。第二天早上，当你试图向别人复述梦境时，即使那些梦看上去完全像真的一样，你也会突然意识到，实际上你根本不知道梦里到底发生了什么，醒来后你说出的只能是一些胡话。如果你在梦里再待上一两分钟，你的意识也许会问出这种问题：同你一起在玫瑰花丛中畅游的是什么哺乳动物？然后，你就会得到形形色色的弱智答案（一头鸭嘴兽）。等到你醒来后打算把这个梦讲一遍时，你才发现为了叙述的连贯性，你必须知道在玫瑰花丛中和你待在一起的到底是什么。但是这是你绝对无从知道的。所以劳驾请不要把你的梦告诉我。

这个机制有一些不好的副作用，其中之一就是你的头脑养成了一种坏习惯，会高估自己清晰判断事物的能力。它总是觉得自己看到了全部，即使事实不是如此。

在软件开发中，这是一个特别危险的陷阱。你在头脑中形成了一个整体的设想，想好了下一步要做什么，一切看上去都清楚无比，都不用你再设计什么东西了。你马上就一头扎入了工作，开始落实你的设想。

举一个例子，你的设想是重新开发一个基于DOS的老式“个人信息管理程序”，那个程序真的非常优秀，但是推广得一点都不好。重新开发看上去不是很难。这个程序运行的所有过程似乎都一目了然，你甚至都不用重新规划……你只需要雇用一批程序员就可以着手快速写出代码了。

这时候，你就已经犯了两个错误。

错误一，你掉入了一个很老套的陷阱，对自己的设想过于自信。“耶，我们完全清楚怎么做！整件事我们全都明白了。不需要写详细的说明书了。直接写代码就行了。”

错误二，你在做产品设计之前就开始雇用程序员。因为世界上只有一件事比你自己设计软件更困难，那就是一个团队一起设计软件。

我都说不清楚有多少次，我和其他程序员在一起开会，有时甚至不超过两个人，我们试图讨论软件应该怎么运作，但是总是毫无成果。我只好回到自己的办公室，拿起一张纸，一个人把它琢磨出来。同其他人互动总是使我

难以集中注意力，无法全神贯注地设计出我想要的功能。

最让我抓狂的就是有些开发团队养成了一个坏习惯，每次需要做决定的时候就要开会。你有没有试过一边参加会议讨论，一边写诗？这就像一批肥胖的建筑工人一起研究怎样写一出歌剧，但是他们同时还躺在沙发上看着电视剧Baywatch。坐上沙发的胖子越多，他们就越不可能写出歌剧。

至少要把电视关掉！

现在，我就要非常不自量力地来猜测一下Chandler[①]开发团队中到底发生了什么事。为什么他们花了几百万美元和好几年的时间，却只取得了现在这点成果[②]。Chandler现在还只是一个充满错误、半成品状态的日历应用程序，与其他58个去年一窝蜂冒出来的Web 2.0日历相比，毫无出众之处。更糟糕的是，那58个程序中，每一个都是由两个在校大学生在课余时间完成的，其中的一个人大部分时间还是用在画标志性的吉祥物上。

Chandler连标志性的吉祥物都没有！

就像我说的，我不可能假装知道什么地方出了问题。可能根本没有地方出错。也可能他们自己还觉得一切都开展得很顺利。Scott Rosenberg写的这本新书堪称优秀，理应成为这个十年期中那些最热门的开源软件创业公司的*Soul of a New Machine*[③]。但是，这本书的结尾却充满了挫折，Scott Rosenberg不得不删掉了一些内容，因为Chandler 1.0版本根本无法很快就开发完成。（我猜测，Scott Rosenberg不想冒风险，不想等到读者根本不需要的时候再出书，要是等到那个时候，读者获得知识的方式就不是读书，而是服药丸了[④]。）

不过，它仍然是一个很好的机会，让我们一窥某些特定的软件项目。那

---

① Chandler是一个开源的个人信息管理软件，网址是chandlerproject.org。*Dreaming in code*一书讲的就是Chandler。

② Chandler的开发很不顺利，项目从2002年启动，直到2007年*Dreaming in code*一书出版时还没有完成，正式发布是在2008年8月。*Dreaming in code*一书的副书名就是“20名程序员，3年，4732个错误”。

③ *Soul of a New Machine*是一本关于计算机产业的名著，曾获普利策奖，作者是Tracy Kidder，出版于1981年。该书记录了一个真实的故事，一个计算机开发团队按照紧张的日程安排，顶住巨大的压力，完成了下一代计算机的设计。

④ 此处，作者是在讽刺Chandler迟迟无法完成。要是等到它真的完成的那一天，人类的科技也许已经进化到只需吞一颗药丸就可以掌握知识的地步。

些项目不停地转动轮子，转啊转，但是却一步也没有前进。原因是项目的设想太宏伟了，但是细节的设计没有跟上。就我看到的而言，Chandler的原始设想基本上就是要开发一个“革命性”软件。这么说吧，我不知道别人的情况，但是我写不出“革命性”的代码，除非你告诉我更多的软件细节。无论何时，只要设计报告用形容词来描述产品（“该产品将酷毙了”），而没有提及细节（如“它的标题栏将是拉绒铝的颜色，所有的图标将带有一点反光，好像被放在三角钢琴上一样”），那么你就知道你有麻烦了。

就我从Rosenberg的书中看到的来说，唯一比较具体的开发设想是“点对点”（peer-to-peer）、“杜绝信息孤岛”（no silos）和“日期的自然语言解释器”（natural language date interpretation）。可能由于书籍篇幅的限制，作者没有展开讨论这些内容，但是原始的设想看来肯定是非常模糊的。

“点对点”是Chandler的raison d′ être[①]。……为什么你要买微软公司的Exchange Server来进行客户端日程的同步呢？“点对点”同步就行了。但是，结果却是“点对点”太难实现，或者还有其他原因，最后这个功能被砍掉了。现在，Chandler里有一个叫做Cosmo的服务器。

“杜绝信息孤岛”的意思是，在其他软件中，你的电子邮件以一种形式（silo）管理，你的日程表以另一种形式管理，你的备忘录（reminder note）则是用第三种形式管理。但是在Chandler中不是这样，只有一种统一的形式，所有数据都用这种形式管理。

一旦你开始思考如何实现“杜绝信息孤岛”，你就会意识到这个设想是做不到的。如果你将电子邮件和日程表放在一起，那应该怎么放？按照收到电子邮件的日期进行排列吗？那样的话，假定星期五我收到200封广告邮件，请问怎样才能不让它们掩盖当天有一个非常重要的股东大会的安排呢？

渐渐地，“杜绝信息孤岛”这个想法被设计者改成了“邮票”（stamp）。举例来说，你能为任何文档、备忘录和日程表贴上“邮票”，这样你就能把它们以电子邮件的形式发送给任何人。你能想到吗？这个想法大概10年前就在微软公司的Office软件中实现了。最后，微软公司在Office 2007中去掉了这个功能，因为根本没人用。用电子邮件给别人发东西，世界上有很多方法可以很容易地实现，犯不着这么麻烦。

① raison d′ être是法语，意为“存在的理由”。

说实话，我觉得“杜绝信息孤岛”对架构太空人[①]（architecture astronaut）最有吸引力。那些人看到子类（subclass），就会想到抽象的基类（abstract base class），他们喜欢把功能从子类移到基类中，但是又说不出实际的好处，唯一的理由就是这样符合软件架构上的美学。这通常是用户界面设计中很糟糕的技巧。这样一来，你实际上是让用户在猜谜，猜对了才能理解你的程序的架构。如果你让程序中的对象与真实世界中的对象看上去很像，感觉很像，更重要的是行为也很像，那么用户就会更轻松地明白如何使用软件，你的程序也会变得更易于使用。如果你想尝试将两个在真实世界中差别极大的对象（电子邮件和日程安排）合并在同一种用户界面中，那么你的软件的可用性将一塌糊涂，因为软件中的数据关系在真实世界中根本不适用。

另一个很酷的东西就是Mitchell Kapor[②]见人就说的：在软件中输入日程安排时，你可以输入自然语言，比如输入“下个星期二”，软件就会很神奇地将你的日程真地列入了下个星期二。这个功能确实很花哨。但是，过去十年中，即使是第二流的日历软件，都有一半已经实现了这个功能。这可不是革命性的突破。

Chandler开发团队估计过高的另一件事是自己能从志愿者那里得到的帮助。开源软件的开发方式同他们想的不一样。如果你想模仿实现其他软件的功能，那么志愿者真的干得不错，因为你想复制的软件原型提供了一个进一步工作所需的设计规格说明书。如果你想对软件功能做一个小小的扩展，那么志愿者干得也不错，比如我需要为EBCDIC[③]加一个命令行参数，那么我就自己动手加了，然后我让软件的维护者将它加入正式代码中。但是，如果你的应用程序还根本没有写出来，什么也不存在，那么别人怎么为它扩展功能呢？这个时候还没有一个用户呢。所以，不会有志愿者为你贡献代码的。

---

① “架构太空人”是Joel Spolsky本人发明的一个术语，用来指某些热衷于将具体事物不断抽象的理论大师，他们将事物抽象到了太空的高度，站得像太空人一样高，那里都已经没有了氧气，模型已经失去了实用价值，他们还不肯停下。请参见Joel的*Don't Let Architecture Astronauts Scare You*一文（http://www.joelonsoftware.com/articles/fog0000000018.html）。

② Mitchell Kapor是Lotus软件公司的创始人，Lotus1-2-3软件的设计者。2001年，他创建了开源应用程序基金会（Open Source Applications Foundation），致力于开发Chandler软件，是开发的实际出资者。

③ EBCDIC是扩充的二进制与十进制交换码（Extended Binary Coded Decimal Interchange Code）的缩写，是IBM公司于1963年至1964年间提出的一套字符编码规则，性质与ASCII码类似。

到了最后，Chandler开发团队中的几乎每一个程序员都是花钱雇来的。

我要再说一遍，如果Rosenberg的书没有完全说出事实，或者他的叙述使得读者对那些阻碍开发的实际问题产生了完全错误的印象，或者我本人显露出了偏见，不分青红皂白，一股脑将各种问题都归咎于软件规划过程中的失败，那么我必须郑重地向Chandler开发团队致以深深的道歉。

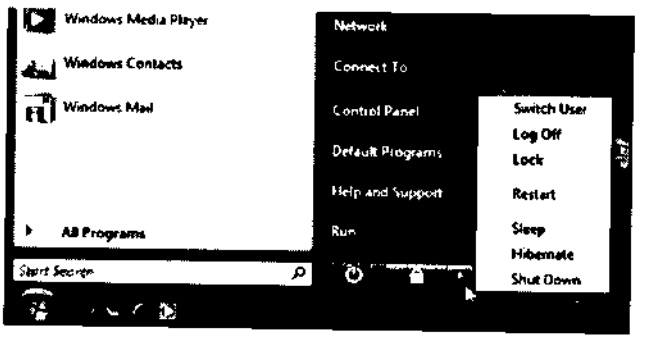
综上所述，Chandler项目的的确确做了一件好事，那就是产生了一本令人着迷的书，这本书的风格同其他两本名著（*Soul of a New Machine*和*Show Stopper*）完全一样，前者讲的是一个开发团队赶在预定日期前完成项目的故事，后者讲的是一个软件开发项目最后无法将各个部分合为一体的故事。最后，向你强烈推荐*Dreaming in code*这本书。

14

别给用户太多选择

2006年11月21日，星期二

我确信，微软公司有一个完整的团队在非常努力地开发Windows Vista里面的那个Off按钮，开发成员包括用户界面设计师、程序员、测试员等。但是说真的，难道你们这些人能够拿出的最好成果就是现在这样？



每次用户要关掉计算机的时候，就不得不在9个选项中选择，看清楚，是9个选项，其中2个是图标，另外7个是菜单项。我认为，那两个图标应该是某个菜单项的快捷方式。我猜想，锁的图标对应了菜单中的Lock项，但是我不确定开关（Off/On）图标对应了菜单中哪一项。

在许多笔记本电脑上，还有4个FN+功能键的组合键，可以实现关机、休眠（hibernate）、待机（sleep）等功能。如果把它们算进去，用户就有13个选择。哦，笔记本电脑本身还有一个开关键，那就有14个选择。你还可以直接把笔记本电脑合上，这是第15个选择。关闭笔记本电脑居然一共有15种不同的方法！

你给用户的选择越多，他们就越难选择，就会感到越不开心。随便举个

例子，Barry Schwartz写过一本书*The Paradox of Choice: Why More Is Less*。让我来引用一段《出版者周刊》的书评："Schwartz的著作在社会科学领域进行了广泛的探讨。他指出，大量令人眼花缭乱的选择在我们的大脑中泛滥成灾，令人精疲力竭。太多的选择最终限制了我们的自由，而不是解放了我们。正常情况下，我们美国人认为选择越多，我们就会越幸福（宽松式的裤子还是休闲式的裤子）。但是Schwartz告诉我们，事实正好相反，他认为太多的选择实际上损害了我们内心的幸福感。"

每一次关机的时候，你都不得不在开始菜单的9个不同选项中选一个，这还不包括选择按下笔记本电脑上的实体开关按钮，以及选择直接合上笔记本电脑，这多少会让你每一次都感到一些不愉快。

可不可以改进？肯定可以啊。iPod甚至连开关按钮都没有。下面是一些改进的思路。

如果最近你跟不懂技术的人士说过话，你可能注意到，他们连待机和休眠之间的差别也搞不清楚。这两个选项完全可以合并在一起，无碍大局。现在只剩下8个选项了。

"切换用户"（Switch User）和"锁定"（Lock），这两个选项也可以合并，因为只有在系统锁定的时候，才允许第二个用户登录。有一点是肯定的，这样就免去了强迫用户退出系统的许多麻烦。现在又少了一个选项。

一旦你将"切换用户"与"锁定"合并，你还需要"退出登录"（Log Off）选项吗？"退出登录"的实际作用只有一个，就是帮你退出所有正在运行的程序。但是，"关闭电源"（Power off）也能做到这一点，所以，如果你很想退出所有正在运行的程序，只要关闭电源，然后再打开就行了。这样一来，选项又减少了一个。

"重启"（Restart）选项也能被去掉，如果你用到这个选项，95%的可能是你安装了一个新软件，它提示你需要重启，如果你确定，它就会帮你自动重启。对于其他情况，你只需要关闭电源，然后再打开就行了。于是又少了一个选项。选择越少，麻烦也越少。

理所当然地，你应当将图标和菜单合为一体。这样又可以再减少两个选项。最后，在我们面前的只剩下：

待机/休眠（Sleep/Hibernate）
切换用户/锁定（Switch User/Lock）
关机（Shut Down）

要是将“待机/休眠”和“切换用户/锁定”合并成一个选项会怎么样？当你选择这个新选项的时候，计算机就自动弹出“切换用户”的屏幕。如果30秒内没有用户登入，计算机就自动待机。过了几分钟后，就自动转为休眠状态。在整个过程中，它一直是锁定的。那么现在我们就剩下了两个选项：

(1) 我现在要走开一会儿。
(2) 我现在要走开一会儿，并且我想将电源完全关闭。

为什么你要关闭电源呢？如果你很关注电源的使用，就让电源管理软件接手就行了。这方面，它可比你在行。如果你担心打开电脑的时候你会被电击，那么关闭操作系统的电源并不能彻底保证你的安全，你必须拔掉电源插头才行。另一方面，如果你的Windows操作系统可以保存内存中的数据，也就是系统在闲置时自动将数据从内存复制到闪存盘上，那么你关不关电源其实毫无影响，就算你处于“离开状态”，拔掉了电源，你也不会损失任何数据。市场上的那些新型混合型硬盘[①]（hybrid hard drive）能够以极快的速度完成这类任务。

那么，现在我们不多不少只剩下了一个“退出登录”（log off）按钮。就把它叫做“再见”（byebye）按钮吧。当你单击“再见”按钮时，屏幕就被锁定了。内存中还没有来得及保存到闪存的数据这时都被写进了闪存。你退出了登录，如果其他人想登录，他们会得到自己的桌面环境（session）。如果没有人要登录，你就可以拔掉整台计算机的插头。

对于我上面的精简过程，你不可避免地会列出一个长长的清单，上面写满了高明的、合理的原因，解释为什么每一个选项都是绝对必需的、不可缺少的。别那么麻烦了。你要说的东西我都知道。多出来的每一个选项肯定有其完全的合理性，但只要你不向你的叔叔解释在15种不同的关闭笔记本电脑的方法中他应该如何选择，一切就没问题。

① 混合型硬盘指的是一种新型硬盘，它在硬盘中封装进了一定容量的闪存，作为内存与硬盘之间的缓存桥梁。这种硬盘可以大大节省电力，并极大地提高了开机速度。

这个例子突显出现实中存在这样一种软件设计风格，微软公司和开源软件运动中都存在这个问题，那就是程序员受到一种愿望的驱使，渴望方方面面都照顾到，让每个人都感到满意。但是，这种愿望的基础其实是一个不正确的认识，更多的选择会不会让用户感到更幸福，我们需要重新思考这一点。

程序员最可信赖的求职帮手

15 易用性是不够的

2004年9月6日，星期一

好多好多年以来，许多自封的（self-fashioned）权威（比如鄙人）一直在不断地唠叨软件的易用性（usability），鼓吹让软件变得易用是多么多么重要。Jakob Nielsen[①]有一个数学公式，使用这个公式可以算出网站易用性的具体数值。如果你愿意出122美元，他就让你看一眼。（如果网站易用性的期望值大于122美元，那么我猜你赚到了。）

我在这方面写过一本书——*User Interface Design for Programmers*，价格要便宜许多。我在书中讲了一些如何设计出具备易用性的软件的原则。我没用到数学，不过你的收获肯定将超过书价。

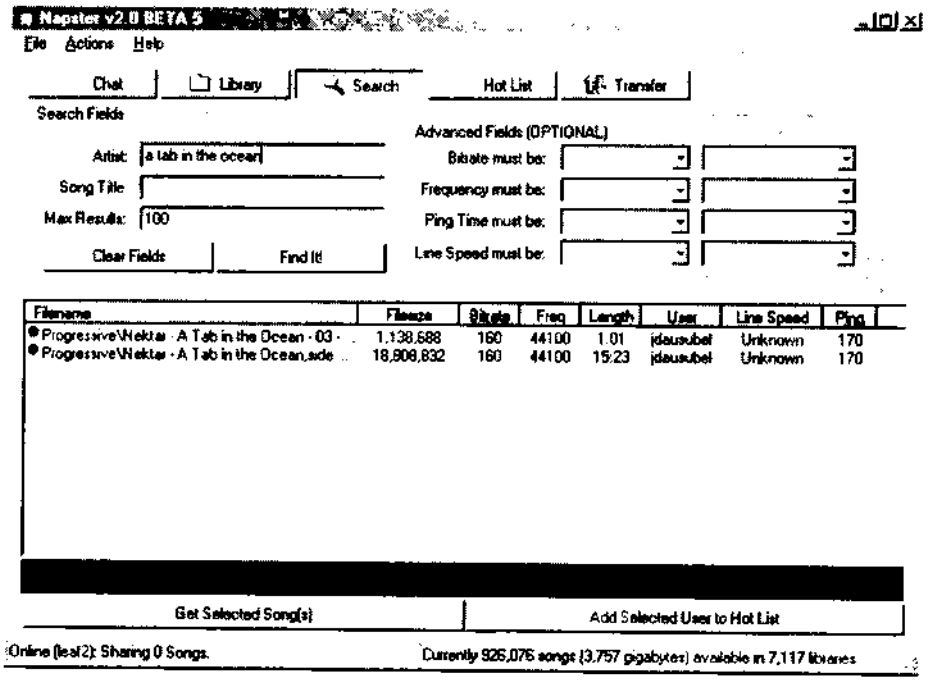
在那本书的第31页，我举了一个例子，对象是当时地球上最流行的应用软件Napster。它的主窗口采用按钮，在5个不同的界面之间切换。（图见下页）界面设计有一个原则叫做“功能的可见性[②]”，根据这个原则，就不应该使用按钮，而应该使用选项卡（tab）。我以前主张的观点就是这样。

但是即便如此，Napster依然是当时地球上最流行的应用软件。

在这本书的手稿中，我实际上还写过这样的话：“这表明软件的易用性并不是决定性因素。”在一本宣传易用性的书中出现这种话其实是挺别扭的。所以，后来当书籍印刷时，排字工人跟我说这一段太长必须压缩，我不由感到如释重负，就删除了这句话。

① Jakob Nielsen，丹麦人，著名的网站易用性咨询专家。

② 功能的可见性（affordance）指的是在设计时必须明显体现出设计对象的使用方法。



但是，必须承认，这件事反映出真理残酷的一面，至少对用户界面设计专家来说，这是残酷的。如果一个应用程序具备确实非常重要的功能，而且用户真地非常需要这个功能，那么即使这个程序难用得令人感到可悲，它仍然会大受欢迎。反过来说，有一个应用程序，被做成是世界上最容易使用的东西，但是如果它对任何人都毫无用处，那么它照样会完蛋。用户界面设计专家总是属于防守方，不得不与一些很不实际的投资回报率公式对抗。那些公式告诉客户，花7.5万美元进行易用性改造到底能取得多大的回报。之所以会发生这种事，完全是因为在许多人心目中，易用性属于可有可无的“可选项”（optional）。在许多情况下，这种看法确实说出了事实。现实中，这样的事情比比皆是。一个用户界面设计师就无法帮助CNN网站多赚到一分钱。它的界面再难用，也依然会有人访问。我愿意担风险，在这里断言，世界上所有以内容为主的网站，没有一个会单单因为易用性的改善而多赚到哪怕一美元。因为，以内容为主的网站（请注意我的言下之意，那些不以内容为主的网站依然属于应用程序的范畴）不管再怎么难用，都已经可以用了，真是太该死了。

不去管它了。

我今天真正要讲的东西不是抱怨易用性如何不被重视……事实上，在其

他条件相同的情况下，易用性设计就是决定性的。有许多例子证明，恶劣的易用性设计会导致小型飞机的空难，造成人员伤亡，还导致饥荒和瘟疫以及别的灾难。但是，这不是我今天要讲的东西。

我今天真正要讲的东西是关于软件设计的下一个层次的问题，也就是当你做好用户界面以后，你会遇到的问题——设计社会化界面（social interface）。

我想，这里需要我来解释一下。

20世纪80年代，“易用性”这个概念被“发明”出来了。当时的软件全部都是人与机器之间的互动。目前还有许多软件是这个样子的。但是，互联网的兴起致使一种新的软件诞生了，它可以实现人与人之间的互动。

论坛，社会化网络，分类广告网站，……嗯，还有电子邮件。所有这些软件都属于人与人之间的中介，而不是人与机器之间的中介。

当你在编写充当人与人之间的中介的软件时，做好易用性设计以后，下一步你就必须做好社会化界面的设计。而且，社会化界面比易用性设计更重要。如果社会化界面一塌糊涂，那么就算你有世界上最好的用户界面，你的软件也活不了。

最好的讲解社会化界面的方法就是看一些成功的例子和失败的例子。

几个实例

首先，请看一个失败的社会化界面。每个星期，都有陌生人给我发电子邮件，邀请我加入他们所在的社会化网络。既然我不认识那个家伙，我就感到有点不悦，直接把邮件删除了。后来，我从别人那里知道了这是怎么回事。某个社会化网络软件公司提供一种工具，这种工具能够自动获取你的电子邮件地址本，然后向其中的每一个人发送邀请加入的邮件。这种工具是第一步，第二步是有些电子邮件客户端软件每收到一封信，都会自动将发信人的地址保存下来；第三步是如果你正好订阅过*Joel on Software*的邮件列表，你

就收到过用我的邮件地址发给你的一封确认邮件，问你是不是想加入，这时你的Email软件就自动把我的地址记入你的地址本。这三者加在一起，一切就这样发生了。所有我根本不认识的人一打开那个社会化网络的工具软件，它就会没头没脑地向我发出一封邮件，要我确认我是这些人的朋友。谢谢你订阅我的邮件列表，但是不要有幻想，我不会把比尔·盖茨介绍给你认识的。我当前的政策是不加入任何社会化网络，因为它们给我的感觉就是与人类关系网的真实运作方式完全格格不入。

现在再来看一个成功的社会化界面。许多人在面对面交流时比较胆怯和拘谨，但是如果不见面用文字交流，就比较放得开。青少年尤其如此。手机上的短消息使得他们更容易将对方约出来。所以，短消息软件在社会化方面做得非常成功，极大地改善了几百万人的爱情生活（或者说，至少改善了他们的社交生活）。即使短消息的用户界面糟糕得可怕，但是它依然在孩子们中疯狂地流行。这件事的可笑之处在于，每一台手机上都有一个比短消息好得多的用户界面，专门供人与人之间的交流，这个精巧的界面叫做打电话。你拨一个号码，然后你说的每一个字对方都能够听到，他说的每一个字你也能听到。就是这么简单，可是在某些人群中，打电话就是不如操作麻烦的短消息流行。你不停地按键盘，简直把大拇指都按断了，只为了发出一句话“天啊，你真漂亮”。为什么你宁愿按断大拇指都不愿直接打电话说？原因是这一长串的按键能让你得到一次约会，如果是直接打电话，你永远不会有勇气说出“天啊，你真漂亮”。

另一个社会化软件的成功例子是eBay。当我第一次听到eBay的时候，我说：“瞎搞！这绝对行不通。你在网上随便遇到一个人，就把钱送给他，寄希望于那个家伙良心未泯会把商品寄给你。这不是太可笑了吗？”抱有类似看法的还有许多人。我们全都错了，完全错了，大错特错了。eBay对文化人类学意义上的人类下了大赌注，并且赌赢了。eBay了不起的地方在于，它的巨大成功完全是因为它的模式当时被看作根本不可行，没有其他人来做这件事，只有eBay做了，等到它依仗着网络效应（network effect）锁定领跑者的优势（first-mover advantage）时，其他人再想追赶就晚了。

除了本身的成功和失败，社会化软件还有一些副效应。社会化软件运作的方式在很大程度上决定了围绕它所形成的用户社区的类型。Usenet用户有

一个叫做“大回复”（big-R）的命令，在回复时可以引用别人的发言，并且在引用的每行左边最前面加上一个简洁的“>”。早期的Usenet客户端，也就是各种新闻阅读器（newsreader），不能列出同主题所有发言，所以如果你想针对某个人的发言做出有连贯性的回复，你就不得不用“大回复”命令引用原始发言。这导致了一种Usenet风格的特殊讨论方式，也就是逐行驳斥（line-by-line nitpick）。这种事做起来感觉很爽，但是对他人来说根本不值得阅读。（顺便说一句，后来在互联网上出现了一些重新使用了这种技巧的政治类的网志，那些作者认为这是他们发现的好玩的新做法，并取名为fisking，具体的原因我这里就不深究了。不过不用担心，这不是脏话。）你看，即使人类已经争论了几千年，但是只因为一个软件产品的小功能就产生了一种全新风格的讨论方式。

软件的小变动就能导致其所支持的（或不支持的）社会化功能的大变动。Danah Boyd为社会化网络软件写过一篇精彩的批评文章*Autistic Social Software*（www.danah.org/papers/Supernova2004.html），抨击当前的一些社会化网络软件强迫用户像自闭症患者那样表现。

> 请认真审视当前的潮流，人们对于将用户通过“朋友的朋友”关系互相连接起来的社会化网络显示出浓厚的兴趣。这样的网络包括Friendster、Tribe、LinkedIn、Orkut等。它们做了一些技术尝试，试图将人们构建和管理自己的人际关系网的方法模式化。它们假设你能够评判谁是你的朋友。在某几个软件中，它们用既定的程序指导人们结识不认识的人，它们不给你提供其他选择，只有唯一的途径，让你可以结识它们提供给你结识的人。
>
> 这种方法肯定有它的优点，因为这样一来，一切都能够通过计算得到。但是，一想到要是人们将这类模型同社交生活等同起来，我就感到恐怖。它们太过于简化了，人们被迫按照程序提供的模式进行交往，好像都得了自闭症一样，好像一定就得按照预先的设计进行互动一样。这种方法对于那些需要这类系统的人肯定有帮助，但是它并非一个普遍适用的模型。而且，用技术来机械化地指导人际交往到底会有怎样的后果？我们是否真地需要一种具有自闭症倾向的人际交往模式呢？

如果一个软件部署社会化界面时，不考虑文化人类学，那么这个软件就

很难使用，令用户恼火，不会真正地发挥作用。

设计社会化软件

让我给你示范一下如何设计社会化界面。

假定你的用户正在做某些他们不应该做的事。

根据易用性设计的思想，这个时候你应该告诉用户哪些事是不对的，应该怎样纠正。专家为这种做法起了个专门的名称，叫做“防御性设计”（Defensive Design）。

如果你把这种做法照搬到社会化软件上，那你就太天真了。

举例来说，用户的不当行为是在一个论坛中贴出伟哥的广告。

你的做法是向这些用户显示警告信息：“对不起，本论坛不欢迎伟哥广告。你的帖子将被拒绝发表。”

猜猜看，那些用户会怎么反应？他们会换一种形式，照贴不误。（或者他们会发动一场冗长而乏味的抗议，跟你辩论审查制度和宪法第一修正案[①]。）

根据社会化界面的工程学（social interface engineering），这时你必须考虑社会学和人类学的因素。在社会中存在各种各样的人，包括想占便宜的人、搞诈骗的人以及其他为非作歹的人。反映到社会化软件中就是，总有人会为了自己的利益滥用软件，并不惜损害他人的利益。听任其发展就会导致经济学家所称的“公地的悲剧[②]”（the tragedy of the commons）。

① 美国宪法的第一修正案保证公民享有言论自由和出版自由。

② “公地的悲剧”是1968年美国生态学家Garrett Hardin（1915—2003）首先提出的。他指出，在公共草地可以无偿放牧，所以牧羊人一定会过度放牧，尽可能多地占用公地的资源，因为如果他不这样做，其他人也会这样做，最终就会导致公地的荒芜。在经济学上，这特指公共资源会因为过度使用而枯竭。过度砍伐的森林、过度捕捞的渔业资源及污染严重的河流和空气，都是“公地的悲剧”的典型例子。之所以叫悲剧，是因为每个当事人都知道资源将由于过度使用而枯竭，但每个人对阻止事态的继续恶化都感到无能为力，而且都抱着“及时捞一把”的心态加剧事态的恶化。

程序员最可信赖的求职帮手

用户界面的设计目标是帮助用户能够成功操作。而社会化界面的设计目标是帮助人与人之间的社会关系能够成功运作，即使这意味着必须要冒犯某个特定的用户。

所以，一个优秀的社会化界面设计师这时候会说："我们不要显示错误信息，假装系统接受了用户张贴的伟哥广告。这样使得张贴者感到达到了目的，就转到其他论坛去贴广告了。我们要做的就是不让他的广告被任何其他人看到。"

说真的，避开攻击最好的方法之一就是让它看上去好像获得了成功。这是装死战术在软件中的表现。

当然，这种方法不会100%有效。不过，95%的情况下它是有效的。所以，它将你遇到的麻烦减少到二十分之一。就像社会学中的其他事情一样，这方面没有最优算法，只能做到大致上的改进（fuzzy heuristic）。在很多情况下它都是有效的，所以就值得尝试，即使不能保证肯定会成功。今天我收到的垃圾邮件中，90%简陋到令人摇头叹息，就连Microsoft Outlook内置的那么低级蹩脚的垃圾邮件过滤器都能把它们拦截住。你只需要随便凑几个很简单的搜索关键词就能成功拦截这些不中用的垃圾邮件。

推广社会化界面

几个月前我意识到，我们Fog Creek软件公司开发的所有软件有一个共同的主题，那就是我们执着地、一心一意地要把软件的社会化界面做好。比如，FogBugz的设计中有许多特定的功能和更多小细节上的安排，使得有效追踪软件中的bug实际上成为可能。不断有客户告诉我，他们以前的bug追踪系统从来没有被真正使用过，因为它不适合团队开发的环境。但是，当他们改用FogBugz后，bug追踪系统就开始真正投入了使用，而且很受欢迎，它改变了团队开发中的一些做法。我知道FogBugz有效，是因为每当我们推出一个新版本时，旧客户选择升级的百分比非常高，这表明FogBugz并不是"架上软件[①]"（shelfware）。甚至有一些客户已经购买了许多张软件

① "架上软件"指的是不被使用的软件，就好像一直搁置在货架上一样。

使用许可证，但是还是经常回到我们这里购买更多的许可证，因为这个软件在他们的组织中使用得越来越广，越来越多的地方需要用到这个软件。这真地令我感到非常骄傲。团队内部使用的软件通常很难推广，因为它要求团队中的每一个人都同时改变使用习惯。如果你学过人类学，你就知道这不太可能做到。由于这个原因，FogBugz在设计时就做了很多安排，即使团队中只有一个人使用这个软件，它也依然是有用的。同时，FogBugz的设计中还有一些有利于推广的特性，能够鼓励其他用户使用，直到团队中每一个人都用上它为止。

在如何正确部署社会化界面这个问题上，我自己的网站上使用的论坛软件做得更完善，我很快就会把它并入FogBugz并作为一个功能上的卖点。我的论坛软件做了各种安排，包括特定的功能、各种细节、设计上的安排等，加起来一共有几十项。这些功能使得我的论坛成为一个可以进行非常高水平的趣味对话的场所，在我去过的论坛中，它具有最好的信噪比（signal-to-noise ratio）。在下一讲中，我会详细来谈它。

这几个月来，我越来越认同“良好的社会化界面设计会为我们创造更多的价值”这一观点，我变得更加专注于实现这个想法。我们聘请了好几个像Clay Shirky这样的专家（他是这个领域的先驱），在不幸的“Joel谈软件”的论坛用户身上进行了大胆的实验（许多变动都很细微，实际上不会被察觉，比如在你回复的时候，我们不向你显示你的留言所针对的原始帖子，目的是防止出现长篇累牍的引用，影响到整个页面的可读性），我们还不惜耗费大量资源，采用许多高级算法，减少论坛上的垃圾帖子。

一个新领域

社会化界面设计是一个全新的领域，还处于婴儿期。据我所知，还没有出版过这方面的书籍，只有少数几个人做过一些相关的研究，但还没有系统的理论来指导如何设计社会化界面。回想起来，在易用性设计诞生的早期，软件公司招募人体工程学（ergonomics）专家和心理（human factor）专家帮助设计具备易用性的产品。人体工程学专家对办公桌的正确高度所知甚多，但是不知道如何正确设计文件系统的图形界面，所以一个全新的领域就这样

诞生了。渐渐地，用户界面设计开始形成了自己的体系，确定了一些基本的概念，比如界面的协调性（consistency）、功能的可见性（affordability）、反馈性（feedback）等。它们成为了用户界面设计学的基石。

展望下一个十年，我期待软件公司会雇用受过人类学家（anthropologist）和人种学家（ethnographer）训练的人，请他们参与社会化界面的设计。他们采用的方法不是兴建易用性实验室，而是走向室外，开展田野调查，写出人种调查报告（ethnography）。但愿我们会找到社会化界面设计的一些原则。那一定非常迷人……就像20世纪80年代用户界面设计刚刚兴起时那样有趣……所以，敬请期待。

16

用软件搭建社区

2003年3月3日，星期一

社会学家Ray Oldenburg在他所著的*The Great Good Place*一书中谈到，人类除了工作地点和住家之外，还需要第三个场所，在那里会见朋友、喝啤酒、谈天说地，享受人际交往的乐趣。比如，咖啡馆、酒吧、美发店、露天啤酒店、台球厅、俱乐部等，对你来说，这些你经常去逛的地方是同工厂、学校和公寓一样重要的。但是，资本主义制度一直在侵蚀这些第三场所，社会生活正在变得贫瘠苍白。在*Bowling Alone*一书中，作者Robert Putnam举出了许多证据，用非常详细和精彩的细节证明美国社会已经丧失了第三场所。过去25年以来，美国人“更少参加聚会性的团体，更少了解自己的邻居，更少与朋友见面，甚至与自己的家人的沟通也变得更少了”。对许多人来说，生活就是去上班，然后回家，然后看电视，生活就是这三件事的单调循环。不过，根据我的观察，这种现象用在程序员身上，尤其在生活于像硅谷和西雅图郊区[①]那样地方的程序员身上，非常不准确。年轻的程序员刚从学校毕业，横跨整个国家，搬到一个没有熟人的新地方，出于孤独，他们只好每天工作12个小时以上。

所以毫不奇怪，那么多的程序员都非常渴望多一点儿人际交往，他们涌向在线社区，比如聊天室、论坛、开源项目和网络游戏。创建一个在线社区，从某种意义上说，就是在创建一个第三场所。在设计时做出正确的决定是至关重要的。以建筑项目为例，开酒吧的话，如果里面的声音很大，人们就不会进行长篇谈话。这样就使酒吧成为一个非常不同于咖啡馆的地方。开咖啡馆的话，如果只放置很少几把椅子，就像星巴克那样，人们只好把咖啡带回

① 微软公司总部位于西雅图郊区。

自己的小屋里，而不能在咖啡馆里休闲地坐着享受社交生活，无法像在电视剧《老友记》（*Friends*）中那家迷人的咖啡馆[①]里一样。虽然我们可能做不到电视里的那种程度，但是有一个类似的地方总是聊胜于无。

软件项目同建筑项目一样，设计规划非常重要，它能够决定在线社区的成败和它的类型。如果你让某个功能很容易操作，人们就愿意使用它。如果你让某个功能很难操作，人们就会避免使用它。通过这种方式，你能够暗中鼓励人们按照预想的方式表现，这决定了一个社区的特性和品质。比如，它是不是让人感到友善？是不是有高质量的发言，仿佛置身于一个欧洲沙龙，里面满是怀有新思想的知识分子？还是它像一个荒芜的遗址，地上到处都散落着肮脏的广告宣传单，没人愿意屈尊将它们捡起来看上一眼？

你可以找几个在线社区，观察一下，你立刻就会注意到它们不同的社交氛围。更仔细地审视以后，你会明白这种差异往往是不同的软件设计规划导致的一个副产品。

在Usenet上，帖子的生命力会长达几个月，讨论内容往往离题十万八千里，你根本无从预料会得到什么结果。只要有贸然闯入的菜鸟问出一个与主题相关的问题，老鸟们就会高声喝止他，让他自个儿去读FAQ。在Usenet上回复的时候，可以用>号将原发言引用出来，这是很大的弊端，如果将所有同一个主题的发言在一个页面上展开，你就会发现根本无法阅读。除非你愿意干一件无聊透顶的事情，愿意将这个系列的帖子逐一地从头读一遍，才有可能搞清楚。但是那样的话，你就会发现几秒钟前读过的东西，又在第二个人的引用中出现，然后是第三个人的引用，接着是第四个、第五个、第六个。你简直在用刷油漆的方式读帖子，那样就成了不折不扣的傻瓜了。

在IRC上，你实际上没有自己的用户名，也没有自己专用的频道，因为只要你一离开其中的一间聊天室，后面进来的人就可以接管你留下的一切。这就是IRC运作的方式。它导致的社交结果就是，第二天你再回到原来的地方，往往不太可能找到你的朋友，因为其他人可能已经锁定了原先的聊天室，而你的朋友也可能被迫选择了不同的用户名。如果有人打算在你睡觉的时候

① 在美国电视剧Friends中，六位主人公经常在一家叫做Central Perk的咖啡馆里坐着聊天。

占据你所在的频道，那么对付他的唯一方法就是创建一个软件机器人，一天24小时地运行，守住这个频道。许多IRC的参与者实际上将大部分的精力用在干一些蠢事而不是交谈上，比如发动复杂的脚本大战（bot war），企图控制某个频道——这往往破坏了其他用户的使用体验。

在大多数商业性的讨论区中，实际上做不到从头到尾完整地读完一个主题的发言，因为主题中的每一个帖子都有它自己的页面，上面布满了广告，一页一页读完整个发言会把你逼疯。那些闪烁夺目的商业垃圾无所不在，使你感到自己好像站在纽约时代广场之上，想要同身边的陌生人结识，但是刺眼的霓虹灯却把你所有的注意力都分散了。

在Slashdot[①]上，每条消息都有几百条读者评论，其中许多都是雷同的，所以那儿的留言让人觉得乏味和愚蠢。在下文中我会解释为什么Slashdot有那么多雷同的回复，而“Joel谈软件”的论坛中就没有这个现象。

在FuckedCompany.com[②]上，讨论区完全是一点儿价值也没有，上面大多数的帖子都是不相关的咒骂和一般性的胡扯，感觉就像同行之间在比赛谁更粗鲁（fraternity rudeness contest），没有一丝的友爱[③]（fraternity）。

说到这里，我们就发现了在线社区的第一公理：

> 软件实现上的小细节会导致在线社区发展、运作、用户体验上的大差异。

因为IRC不允许用户拥有频道，所以用户自己组织起来，忙于进行脚本大战。因为Usenet兴起的时代，上网设备是300波特[④]的调制解调器，所以第一个新闻组阅读软件“rn[⑤]”设计成一次只能读取一个新帖子，不能获得整个主题的所有帖子，目的就是为了节省带宽。这样造成的后果就是，如果你想驳斥前面某人的发言，你就不得不引用他的帖子，否则没人明白你在说什么。如果所有人都这样做，整个主题的发言就变得极为冗余。

---

① Slashdot.org是一个著名的科技新闻站点，以用户留言踊跃而闻名。
② 这里的意思是“某个欠揍的公司”。
③ 作者在这里用了双关语，“同行”和“友爱”在英语中都可以用fraternity表示。
④ 波特（baud）代表了调制解调器中每秒钟通过的位，300波特相当于每秒网速不足0.3KB。
⑤ rn是Read News的缩写，是历史上最早的新闻组客户端软件之一，于1984年发布。

在这个认识的基础上，我来回答一些关于“Joel谈软件”的论坛的最常见问题，比如为什么它设计成现在这样，它是怎么运作的，又是怎么改进的。

Q. 为什么你们的论坛软件那么简化？

A. 我们的论坛刚开出来的时候，我们需要尽快地让发言的数量和质量超过临界点，这很重要，否则就会发生空酒馆效应（没人会去空的酒馆吃饭，而宁愿去隔壁生意兴隆的酒馆，哪怕那家酒馆其实难吃得要命）。我们的设计目标就是，消除发言的一切障碍。这就是为什么我们不要你注册，也几乎不提供其他功能的原因。这意味着你不需要事先了解和学习什么。

从商业上看，这个论坛的目的，是为我们Fog Creek软件公司的产品提供技术支持。论坛就是为这个目的服务的。为了达到目的，没有什么比让论坛变得超级简单易用、任何人都能愉快使用更重要的了。论坛的每一个功能都再明显不过了。据我所知，来到论坛的每一个人都立刻领会了该怎么使用。

Q. 你们能否增加一个功能，当有人回复我的帖子时，自动用电子邮件通知我？

A. 这个功能很容易实现，所以对程序员很有诱惑。但是，它是扼杀新兴论坛的最好杀手。如果有了这个功能，你的论坛可能永远不会有人气。Philip Greenspun开发的LUSENET[①]就有这个功能，但是你能够看到新兴论坛的生命力都被它慢慢吸走了。

为什么？

人们来到你的论坛是为了提问。如果你提供“通知我”功能，那么访问者贴完帖子后，选上这个功能，就永远不会再回来了。他们只读那些发到他们邮箱的回复。一切到此为止。

如果你不提供这个功能，他们就没得选择，只能每过一会儿就回来看看。而当他们回来的时候，也许就会去读其他引起他们兴趣的帖子，然后可能就会为那个帖子贡献一点什么。论坛刚刚起步的那段时间是很关键的，你要设法增加论坛的“粘度”，让更多的人愿意经常在论坛上晃悠，这会大大缩短论坛达到人气门槛的时间。

① LUSENET是一个免费论坛软件，从1995年开始开发，到2005年停止开发。

Q. 好吧。那么你们能不能在一个主题中提供分支功能？如果某些人离题了，那么就让他们在分支中进行讨论，你可以选择跟随他们，也可以选择继续在主干上进行阅读。

A. 分支功能对于程序员来说是很合乎逻辑的想法。但是，真实世界中的会话不是这个样子的。在讨论中另外开辟分支会使阅读变得不流畅，令读者分心。你知道分心是怎么回事吗？有时，我打开我的开户银行的网站，因为网速很慢，我点了几次以后，都不记得刚才想要干什么了。这让我想起一个笑话。三个老太太在一起唠嗑。第一个老太太说："我的记性太差。有一天我站在家门口，手里拿着包。我想不起来我是准备出去扔垃圾，还是刚从杂货店回来。"第二个老太太说："我的记性也很差。有一天，我开着车在自家的车道上，就是想不起来我这到底是回家，还是要去犹太教堂。" 第三个老太太说："感谢上帝，我的记忆力还不错，就像时钟那样清楚。但愿老天保佑我。"她用手敲桌子[①]，哨，哨，哨，然后突然说："请进，门是开着的！"分支使得讨论偏离主题，变得让人困惑和不自然。如果有人想离题，更好的方法是强迫他们另开一个主题。这倒是提醒了我以后开发的方向……

Q. 你们论坛的讨论主题的排列顺序是错误的。应该将最新回复的主题排在第一位，而不是按照话题的发起时间排序。

A. 你说的方法是可行的，网上许多论坛也正是这样做的。但是，如果实行这样的排序，某些主题往往会永远出现在论坛的最上方，因为人们总是愿意讨论H1B签证[②]，或者大学中的计算机教育有什么问题，直到宇宙末日也不愿意停下来。每天都有100多个新人第一次来到这个论坛，他们往往会从话题列表中最上方的主题开始看起，满怀热情一头扎了进去。

我现在的安排有两个好处。首先，讨论主题的更新换代很快，所以列表最上方的话题总是保持相对新鲜。而且，到了某一个点，人们就会停止对一个话题进行争论。

其次，话题的顺序在列表上是稳定的。所以，如果日后你想寻找一

① "老天保佑"在原文中的用词是"knock on wood"，这是一个成语，直译过来就是敲木头，因为在美国有一种风俗，说这句话的时候，就用手敲桌子。

② H1B签证是美国政府发给希望在美国工作的外国人员的签证，允许美国雇主暂时性雇用某些特定职业的外国雇员。

个自己感兴趣的以前的讨论主题，就会比较容易，因为它相对周围话题的位置是固定的。

Q. 为什么你们不提供某种功能，让我看到哪些帖子我已经读过了？

A. 我们的系统最符合分布式、可升级的模式，我们让用户的浏览器自己来记录看过哪些帖子。如果一个链接你已经打开过了，浏览器会自动将链接的颜色从蓝色变为紫色。我们做的唯一一点变动就是，每当一个主题中有新的回复时，我们就对URL做一点小小的重构，将总的回复数加入网址中。这样一来，这个链接就会在浏览器中重新显示“未读”的颜色。

除此以外的其他实现方式做起来都很麻烦，而且毫无意义地使用户界面变得更复杂。

Q. 该死的“回复”命令在页面的最最底下。这样用起来很讨厌，你不得不将页面一直滚动到最底下，然后才能回复。

A. 这是故意设计的。我更希望你看完所有帖子以后再回复，否则你的发言可能会重复他人的话，或者你会说出同最后一个帖子不相干的内容。说实话，我恨不得能够抓住你的眼球，强迫它们从上到下将所有内容都看完，然后再让你贴出自己的发言。如果将“回复”命令放在除了页面最下方以外的任何其他地方，实际上会鼓励人们在看完现有内容之前就迫不及待地发表自己那一点智慧。这就是为什么Slashdot上每条消息有500条回复，但是只有17条回复是值得一读的。这也是没有人愿意一条条地看Slashdot上的讨论的原因，看它们就好像来到一间教室中，听着里面所有的小孩同时一起高喊同样的答案。

Q. 该死的“发起新话题”的命令在整个页面的最最下方……

A. 嘿嘿，理由同上。

Q. 为什么你们不在用户确认发表之前，增加一个预览步骤？这样人们就能避免说错话或者打错字。

A. 实践表明，这种看法是不对的。它不仅是不对的，而且正好说倒了。理由一：如果你增加一个预览步骤，大多数人可能看也不看直接就

点确认了。很少有人会认真重读自己的帖子。如果他们真地想重读自己的帖子，他们在编辑的时候就会这样做，但是事实上，他们对于自己的帖子已经感到厌倦了，就好像在看昨天的报纸一样，他们只想赶紧发表出来了事，这样就可以去干别的事了。

理由二：没有预览反而会使得人们变得更仔细。好像有研究表明，没有护栏的盘山公路更安全，因为人们因此感到恐惧，所以开车时会更小心。如果反过来，即使公路旁边架着不牢固的铝制防护栏，也防不住一辆2吨的SUV车以每小时50英里的速度一头冲下山崖。统计数据表明，当那些司机吓得魂飞胆丧、只敢用每小时2英里的速度在狭窄的盘山公路上爬行时，出事的概率会小很多。

Q. 当我在写回复的时候，你们为什么不显示我要回复的原始帖？

A. 原因是那样会引诱你在自己的回复中引用那个帖子。我想尽一切办法，就是要减少引用的数量，因为这样能够增加整个对话阅读起来的流畅度，让话题变得更吸引人。引用前面某人的发言就会迫使读者连续读两遍同样的内容，这样是毫无意义的，而且铁定让人生厌。有时，人们就是想要引用别人的话，只因为他们要回复隔着3个帖子的某句话，或者因为他们不顾一切地想要逐句驳斥某人的发言，以及分布在不同帖子中的12个谬论。他们并非想搞破坏，他们是程序员，编程工作要求一个人在每个i上面点上一点，在每个t中间画上一横，所以他们形成了一种思维定势，就是绝不能不回应争论，就像在编译器中绝不能留下一个错误一样。但是，我就是不要他们活得这么累，即使我因此被骂死也无所谓。为了达到这个目的，我用尽了各种方法，甚至想设法将帖子从文字转成图片格式，这样你就无法复制和粘贴了。如果你确实需要对隔着3个帖子的某句话做出回复，那么麻烦你多花一点时间，写一句像样的英语句子就可以了，而不要使用一大堆<<<>>>[1]，像随地乱扔垃圾一样。

Q. 为什么有时帖子会找不到了？

A. 这个论坛是有管理员的，这意味着一些人具备删除帖子的权力。如果他们删除的帖子正好是该讨论主题的第一个帖子，那么因为没有

① 在早期的网上讨论中，经常在一行的最前面加上小于号或大于号，表示这是引用他人的话。

办法将剩下的帖子单列为一个主题，所以整个主题都会被删掉。

Q. 但是那样做就是审查发言！

A. 不，我们做的只是从公园中捡出垃圾。如果我们不这样做，信噪比将急剧放大，论坛将变得一团糟。有人发垃圾帖和诈骗帖，有人针对我贴出反对犹太人的内容，有人纯粹是在写毫无意义的废话。一些理想化的青年人也许梦想一个完全没有审查制度的世界，在那里智慧的思想可以得到自由交流，每个人的智商都提高了，仿佛身处理想中的牛津辩论社或演讲角一样。我是一个实用主义者，在我眼中，一个完全没有审查的世界就好像你的个人邮箱一样，其中80%是垃圾邮件、广告邮件和诈骗邮件。如果论坛成为那样，那么很快就会将少数那些有趣的人逼走。

如果你想寻找一个地方，在那里能够自由表达你的想法，没有任何审查，我建议你自己创建一个论坛并让它火起来。（向Larry Wall道歉[1]。）

Q. 你怎么决定哪些帖子应该被删除？

A. 首先，我会删除那些完全离题的帖子，以及那些在我看来只对很少几个读者才有意义的帖子。如果某人谈论的东西完全不符合“Joel谈软件”这个网站的主题，那么可能只有某些特定的读者会莫名其妙对其发生兴趣，而不可能引起大多数访问我的网站的人的兴趣，这些人的目的只是想看到一些跟开发软件有关的内容。

过去，我所定义的“离题”还包括任何对于论坛本身的讨论，比如讨论论坛的设计或论坛的易用性。禁止讨论这些话题的原因同上面略有不同，几乎单独就能算作一条公理。每一个论坛、邮件列表、讨论组、BBS等，每过一个星期或两个星期，都会遭遇到同样性质的关于论坛本身的讨论。大概一星期一次吧，某个人逛到了这个论坛，然后开出一张清单，宣布他认为论坛软件应该做出的改进，声称这些事情应该立刻执行。接着，另一个人说：“老弟，别搞错了，你不花钱就在使用这个论坛，Joel这是在为我们大家做一件好事，你不喜欢用可以不用。”第三个人回应说：“Joel开设这个论坛可不是为

程序员最可信赖的求职帮手

① Larry Wall是前文提到的第一个Usenet阅读器rn的作者。Usenet基本上可以看作一个毫无任何审查的空间。

了公益事业，他完全是为了推广他的Fog Creek软件公司。”整个过程真是太太太太没劲了，因为这种事情每个星期就要发生一次。这就好像大家都找不到别的东西可谈，只能聚在一起谈天气一样。对于第一次来论坛的新人来说，这种事情可能是令他兴奋的，但是这与软件开发没有什么关系，那么我的反应正是卡通人物Strong Bad的口头禅“删你没商量”。可是，不幸的是，我最终认识到，试图阻止人们谈论论坛本身就好像试图阻止河流流动一样。不过我还是要说，如果你正在读这篇文章，并且你想到论坛上讨论这个话题，那么我恳求你行行好，千万千万不要那样做，克制住你想要说话的愿望吧，就算帮我一个大忙好了。

我们还将删除那些非公共性的、只是针对个人的人身攻击。我最好把定义说清楚，我所说的人身攻击，指的是针对某个人本人，而不是针对他的观点的攻击。如果你说“这种想法很愚蠢，因为……”，OK，没有问题，你可以说。但是，如果你说 “你这个人很愚蠢”，那就不行，这就属于人身攻击。只要你的发言是恶意的、粗鲁的、诽谤中伤的，那么我就会删除。不过有一个例外，因为这个论坛是“Joel谈软件”的论坛，所以这里是批评Joel的最佳地点，任何针对Joel的恶意的和粗鲁的帖子都被允许发表，但是前提是它们必须至少包含哪怕一丁点儿有用的理由或观点。

任何对上一个发言者的拼写或语法进行评论的帖子也会一律删除。请想象一下，我们正在谈论面试，一个家伙突然插进来说：“你这种拼写水平还能找到工作，真是奇迹啊。”这种话真是太无聊了，居然谈论别人拼写对了还是拼写错了。超级超级无聊。

Q. 为什么你们不公开管理方针，却像守住秘密一样不说出来？

A. 有一回，我从纽瓦克国际机场搭火车回曼哈顿。置身于破烂失修的车厢中，我唯一可干的事就是阅读墙上的一张大大的告示。它用很严厉的语气非常详细地列出了所有被禁止的行为，一旦你行为不端，在下一个停靠站点你就会被押下列车，警察会被叫来处理你。我心想，阅读这张告示的人中，99.99999%都不会有不端的行为，那些真正的不法之徒才不会在乎告示上怎么说呢。所以，这种告示的真正效果是，一方面让那些守法的好人感到自己正在遭受某种起诉，另一方面又根本无法阻止那些它想要阻止的破坏行为。它起到的作用

只是无休无止地提醒那些新泽西州的守法公民，别忘了这里是纽瓦克——犯罪之都[1]！当不法之徒登上列车时，他们就会干出非法行为，制造令人震惊的结果，被押下火车，接着警察赶来。

几乎每一个来到“Joel谈软件”的论坛的访问者，多多少少都没有什么天生的智力缺陷，大脑都是健全的，能够分清帖子中恶意的个人攻击是不文明行为，明白在一个软件论坛上不要问怎么学习法语，或者很清楚不必发动讨论批评某人的拼写水平很低下。但是，除此之外的0.01%的访问者对论坛的规矩根本不屑一顾。所以，把管理方针公开贴出来只是一种对大多数遵纪守法访问者的侮辱，而对心智不健全的家伙毫无阻止作用。那些家伙目空一切，认定自己的屎都是香的，贴出的每一句话都是有道理的，不可能违反任何规定。

当你将制造麻烦的人公之于众时，其他人会觉得你太苛刻了，或者会很生气，因为他们什么也没有做错，却好像也在被你责备。这就像又回到小学里一样，一个笨手笨脚的小孩打碎了一扇玻璃窗，然后全班的小孩都不得不坐在教室里听老师对每一个人训话，内容是为什么你不可以打碎玻璃窗。所以，对于管理方针的任何公开讨论，比如为什么删除某一个特定的帖子，都是应该忌讳的。

Q. 除了删除帖子之外，你们为什么不提供评分功能，让读者根据自己的喜好对每一个帖子投票，并且还可以选择只读那些评分高的帖子？

A. 这其实就是Slashdot的运作方式。我打赌，你们这些经常阅读Slashdot的人中，大概有50%从来没有搞清楚过这套系统是怎么回事[2]。我不喜欢这种做法的原因有三个。一，它会让用户界面变得更复杂，用户需要花时间学习怎么使用这个功能。二，它让论坛内部的管理和人际关系变得复杂，使得一个拜占庭帝国看上去就好像一个三流学校的管理当局。三，如果你打开Slashdot的留言过滤功能，只读那些评分高的帖子，那么整个话题就给人一种支离破碎的感觉，你得到的只是一些随机的、不连贯的表达而已，没有上下文可以参照。

Q. 为什么你们不搞用户注册呢，这样就能消除那些粗鲁的发言者？

① 纽瓦克市的犯罪率排在美国前5位，单单是杀人案，2006年就有63起，2007年上半年又发生了60起。

② Slashdot以拥有一个强大的留言过滤系统而著称，它的投票和打分系统的运作方式非常复杂，涉及高级的数学算法。

A. 正如我前面解释过的，这个论坛的目标就是让你的发言变得更容易。（别忘了，这个论坛主要是用来提供技术支持的。）一旦只有注册用户才能发言，就会让至少90%的有发言意愿的人却步。如果需要技术支持，那么这90%的人就会转而拨打电话，而所有这类电话都是要由我付费的。

此外，我不觉得注册会对论坛管理有所帮助。如果有人要搞破坏，注册系统也赶不走他们，注册对他们来说只是举手之劳。通过注册来改善社区品质这种想法太老土了。我认为，只有那种发邀请函，然后对方同意来参加的会议，注册才是有效的，你能用这种方式创造一个人际网络，大家在一起讨论问题，并且你还可以向每个参加者收费。

注册既不会改善讨论的质量，也不会提高参与者的总体水平。如果你仔细观察“Joel谈软件”的论坛的信噪比，你也许会注意到发出噪音最多的人（即那些发言最多、却贡献思想最少的人），往往是这个论坛的一些核心的老用户，他们每过十分钟就到论坛看一眼。这类用户感到有必要说出“我同意你”，以示自己的存在，并且对每一个主题都会回复，即使他们没有什么新东西要说。如果论坛有注册系统，他们肯定会注册。

Q. 未来有什么计划？

A. 开发和完善论坛软件不是我和我的公司的首要工作。这个论坛已经很不错了，能够正常运行，已经创造出了一个大家都喜欢来的氛围。在这里，我们一起谈论与计算机有关的一些难题，从那些属于世界上最聪明者之列的人身上获得新的想法。我还有许多更有意义的事情可做，创造论坛系统易用性的下一个飞跃还是留给其他人去做吧。前不久，我刚刚创立了一个纽约城市论坛。我想看看，以地理区域为基础的论坛能否鼓励人们像增进线上交往那样增进线下的在现实生活中的交往。根据我自己的经历，地区性的网上社区往往会发生质的变化，从一个简单的网站，变成一个真正的小社会，一个真正的第三场所。

无论在什么情况下，创建在线社区都是一个值得追求的目标，因为我们中有非常非常多的人都极其需要它。为了做好它，让我们一起来努力。

第四部分

管理大型项目

17 火星人的耳机

18 为什么Microsoft Office的文件格式如此复杂（以及一些对策）

19 要挣钱，就别怕脏

17

火星人的耳机

2008年3月17日，星期一

在互联网上的各个论坛中，网站开发工程师之间即将爆发一场规模空前的口水战（flame war），它的根源就是你下面要看到的东西。这场口水战的规模之大足以使斯大林格勒战役黯然失色，让后者看上去就如同你的小姑子在奶奶家喝完下午茶着急忙慌地出门，结果开着福特野马一头撞在了大树上。

这场即将来临的大战的罪魁祸首就是Dean Hachamovitch。他是一个老资格的微软人，现在正在领导一个开发团队，致力于为你带来下一个版本的Internet Explorer浏览器，也就是8.0版。目前，IE 8开发小组正面临做出抉择的时刻，他们正正好好、不偏不倚、一丝不差地来到一个断层的正中间，两边是两种截然不同的世界观。这两种世界观的差别大得就如同保守派和自由派、理想主义者和现实主义者之间的差别。接下来将是一场浩大的全球性圣战，让同一个家庭的成员针锋相对，让工程师与计算机科学家互不相让，让凌志车与橄榄树之间的冲突[①]再现。

根本就没有化解这场战争的办法。说实话，旁观这场大战真是非常、非常好玩，因为99%的参与者其实不明白自己在说什么。但是，事情并不只是

① “凌志车与橄榄树之间的冲突”指的是《纽约时报》专栏作家托马斯·弗里德曼（Thomas Friedman）的一本书，书名为《凌志车和橄榄树：理解全球化》（*The lexus and the olive tree: understanding globalization*）。在该书中，弗里德曼认为全球化正在与发展中国家的传统生活形态和价值观发生冲突，他用凌志车和橄榄树来比喻这种冲突。“凌志车”是全球化的典型代表，象征更高的效率、更好的产品、更激烈的竞争、更自由的资本流动、更广泛的合作、更尖端的科技以及更大的生活变化。“橄榄树”则代表家庭、民族、地区性、传统文化等更为传统的因素，它们为人提供内心需求中稳定的一面。

好玩而已，每一个需要设计协同[1]系统的程序员都应该认真审视这个事件。

口水战的核心是一种叫做“Web标准”的东西。我们就来听听Dean Hachamovitch自己是怎么解释“Web标准”这个难题的（blogs.msdn.com/ie/archive/2008/03/03/microsoft-s-interoperability-principles-andie8.aspx）：

> 所有的浏览器都有一个“规范的”模式，称为“标准模式”。它提供对网页标准的最好支持。因为对网页标准的支持一直在不断的改善中，所以每一种浏览器的每一个版本的标准模式都不一样。Safari 3有自己的标准模式，Firefox 2有自己的标准模式，IE 6有自己的标准模式，IE 7有自己的标准模式，它们都不一样。我们的目标是让IE 8的标准模式大大优于IE 7的标准模式。

程序员最可信赖的求职帮手

所有麻烦的起源是一个不起眼的小决定。IE 8开发小组需要决定，如果遇到一个声称自己符合“标准”、但是实际上可能只在IE 7下才显示正常的网页，那么应该怎么办？

你也许会问，“标准”究竟是一个什么玩意？

难道不是所有工程领域都有标准吗？（对，都有标准。）

难道标准是不起作用的吗？（这个嘛……）

为什么“Web标准”会乱成这种样子？（这不单是微软的错，你也有份。Jon Postel[2]也逃不了干系，我在后面会解释的。）

上面的问题没有解决办法，如果有，也是错误的，而且错得很厉害。Eric Bangeman在*Ars Technica*一书中这样写道：“IE开发小组就像在走钢丝，要么选择严格地支持W3C标准，要么选择向那些只能在早先IE版本下浏览的网站妥协。”（arstechnica.com/news.ars/post/20071219-ie8-goes-on-an-acid2-trip-beta-due-in-first-halfof-2008.html）。这句话有一个地方说错了，IE开发小组脚下的并非钢丝，而是什么也没有，根本就无处可走。他们往前走会摔死，往

① 在软件领域，协同性（interoperability）指的是不同的程序之间，能够采用一套标准的格式交换数据，也就是说它们采用的是同一套协议。缺乏协同性，就意味着软件设计的时候没有考虑到标准。详情请参见http://en.wikipedia.org/wiki/Interoperable。

② Jon Postel（1943～1998）是美国计算机科学家，为互联网的发展做出了极其重要的贡献，尤其是在制定网络标准方面，著名的Request for Comment系列标准的编辑工作就是由他完成的。1998年，他死于心脏病发作，年仅55岁。

后走也会摔死。

这就是为什么我不能在这个问题上表态，而且我也没打算这样做。但是，每一个业内的程序员至少应该理解标准如何生效、标准应该如何生效、我们又怎么会沦落到今天这种一团糟的局面。所以，我今天打算在这里解释一下这个问题，你会看到正是出于同样的原因，Microsoft Vista才卖得这么糟。我以前介绍过的微软公司中Raymond Chen[①]阵营（实用主义者）与MSDN阵营（理想主义者）之间的冲突，也是同一个问题。后来，MSDN阵营获得了胜利，后果就是如今没人能够搞清楚Office 2007中自己最喜欢的菜单去了哪里[②]，也没人真地想用Windows Vista。这些问题其实都是同源的，关键就在于你到底是一个理想主义者（红方），还是一个实用主义者（蓝方）。

让我从头说起。请先想一下，如何让不同的东西搭配在一起工作。

不同的什么东西？随便什么东西，真的。铅笔和卷笔刀，电话机和电话网，HTML网页和网络浏览器，Windows图形界面应用程序和Windows操作系统，Facebook和Facebook应用程序，立体声耳机和立体声音响。

在它们的结合点，所有相关的事宜都必须取得一致，否则两者无法一起工作。

我用一个简单的例子来说明。

假设你来到了火星，发现那里的火星人没有便携式音乐播放器，它们仍然使用手提式录音机。

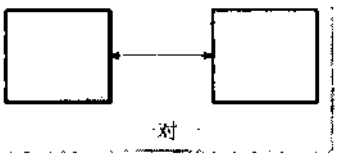
你意识到这是一个巨大的商机，于是就在火星上开始贩卖便携式MP3

① Raymond Chen是微软公司的一个资深程序员，他从1992年就加入了微软。关于Raymond Chen阵营与MSDN阵营之间的冲突，参见Joel写于2004年6月13日的*How Microsoft Lost the API War*一文（http://www.joelonsoftware.com/articles/APIWar.html）。大致上，Joel是说，Raymond Chen阵营认为吸引程序员为Windows系统开发应用程序的关键在于保证系统的兼容性，使得程序员为一个版本的Windows写出的程序在另一个版本的Windows中也能正常使用；而MSDN阵营认为，吸引程序员的关键在于向他们提供更新更强大的组件，让开发工作变得更容易，不论“新技术有多复杂、安装起来有多麻烦、学起来有多难”。

② Office 2007采用新的Ribbon界面，将过去版本中的菜单栏和工具栏一并抛弃，取而代之的是使用标签页将图标、选项和下拉菜单都集合在其中。结果，很多老用户都找不到自己常用的命令在哪里。

播放器（唯一的不同是，火星人将其称为Qxyzrhjjjjukltks）和配套的耳机[①]。为了连接MP3播放器和耳机，你发明了一种灵巧的金属插头，看上去就是下面这样。

因为播放器和耳机都由你来决定，所以你能保证它们能够组合在一起工作。这是一个“一对一”（one to one）的组合：一个播放器，一个耳机。



你也许会写一个设计规格说明书，希望第三方供应商[②]会做出不同颜色的耳机，因为火星人对插在耳朵上的东西的颜色非常敏感[③]。

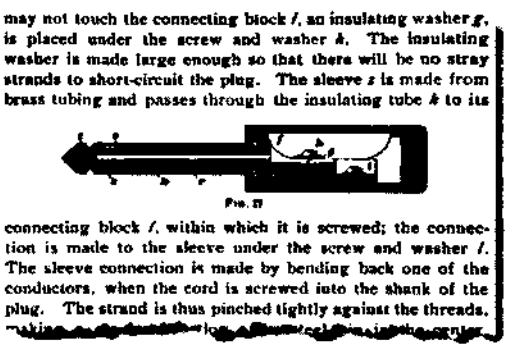

may not touch the connecting block *f*, an insulating washer *g*, is placed under the screw and washer *h*. The insulating washer is made large enough so that there will be no stray strands to short-circuit the plug. The sleeve *s* is made from brass tubing and passes through the insulating tube *k* to its

FIG. 2

connecting block *f*, within which it is screwed; the connection is made to the sleeve under the screw and washer *f*. The sleeve connection is made by bending back one of the conductors, when the cord is screwed into the shank of the plug. The strand is thus pinched tightly against the threads.

但是在写设计规格说明书的时候，你忘了写上电压应该在1.4伏左右。你只是疏忽了。等到第一个热心的厂商制造出100%兼容的耳机，他的扬声器电压大概只有0.014伏。所以当他测试原型产品的时候，结果要么是耳机爆掉，要么是听者的耳膜爆掉，就看哪一个先发生。然后，他做了一些调整，慢慢就得到了一个能够正常工作的兼容耳机，区别是他的耳机产生的音频比

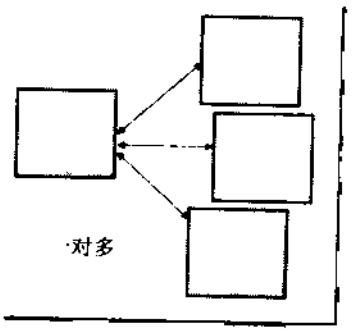
① 作者在这里用MP3播放器和耳机来比喻浏览器及网页。当然，Qxyzrhjjjjukltks指的就是IE。——编者注

② “第三方供应商”指网站开发人员。——编者注

③ 比喻用户希望看到不同内容和外观的网站。——编者注

你的耳机会高出几十埃[1]。

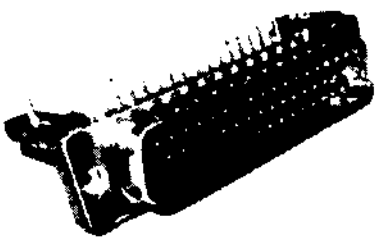
越来越多的厂商加入了这个市场，他们也开始生产兼容耳机。于是，很快地，就形成了一个“一对多”（one to many）的局面[2]。



到目前为止，一切进展得很顺利。耳机插头存在一个事实上的标准。虽然书面的设计规格说明书的内容并非全面完整，但是任何制造兼容耳机的厂商只需要将产品插入你的播放器中测试一下，看看它能否正常工作就行了。如果没有问题，厂商就可以出售耳机了，因为耳机已经能用了。

直到你决定推出新版本：Qxyzrhjjjjukltk 2.0。

Qxyzrhjjjjukltk 2.0中还包括电话（因为火星人自己还造不出手机），所以耳机中就必须再加入一个内置的麦克风。这就要求主机与耳机之间再增加一条数据交换线路，因此你重做了一个同以前产品完全不兼容的插头，样子非常丑陋，但是为升级预留出了足够的余地。



Qxyzrhjjjjukltk 2.0在销售上一塌糊涂，彻彻底底失败了。没错，它有很不错的电话功能，但是没人在乎这一点。用户在乎的是他们收集的数量巨大

① 埃（angstrom）是长度单位，等于1米的100亿分之一，也就是0.0001微米，用于表示电磁波的波长。

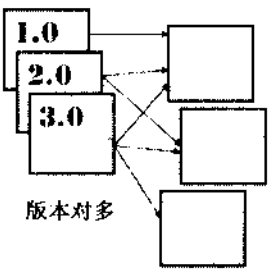
② 也就是IE浏览器一统天下，成为事实标准的时代。——编者注

的耳机[1]。我前面说过，火星人很在意戴在耳朵上的东西的颜色，这证明了我不是瞎说的。眼下，大多数时髦的火星人都有一个大衣柜，里面整整装满了一柜子的漂亮耳机。对你来说，它们看上去都是差不多的（都是红色的），但是对于火星人来说，它们是非常非常不一样的，每一种红色都有非常非常细微的差别，这是你永远不会明白的。火星上最新的高级公寓在促销的时候都打广告说公寓里面附带一个耳机柜。我不骗你。

既然新的插头不成功，你很快就想出了一个解决办法。

看清楚，你做的就是将耳机插头的金属轴分成三段，这样一来，有一段就可以用来传送麦克风信号。但是问题是，你的Qxyzrhjjjjukltk 2.1版完全没法知道插进去的耳机到底带不带麦克风，而它只有知道这一点才能决定是否激活电话功能。你只好又发明了一个小协议……新版本的设备在麦克风的针脚上输出一个信号，然后从插头上寻找这个信号。如果找到了，那么这必定是一个三段式插头了；否则耳机上就不带麦克风，机器就打开向后兼容模式，只能播放音乐。这个步骤很简单，但是它是一个协议磋商过程（protocol negotiation）[2]。

这时，市场就再也不是一对多了。虽然所有的立体声设备还是由同一家公司制造，但是这家公司推出了多个版本，因此我把这称为“版本对多”（SEQUENCE-TO-MANY）市场。



① 即用户经常会光顾的各种网站，可能保存在“收藏夹”里面。——编者注

② 意指新版本浏览器既要增加新功能，又要做到向后兼容，结果导致复杂性增加。——编者注

下面是一些你已经知道的“版本对多”市场：

(1) Facebook对约20 000个Facebook应用程序；
(2) Windows对约1 000 000个Windows应用程序；
(3) Microsoft Word对约1 000 000 000个Word文档。

这样的例子还可以举出几百个。关键的一点是要记住，当设备有了新版本时，一定要保证它自动向后兼容以前版本的附件，保证老附件能够在新版本的设备中正常使用，就像老附件能与老设备协同工作一样。因为在设计老附件的时候不可能考虑到新设备。火星人买的耳机早就被生产出来了，你不可能将它们一一收回再换成新的。更容易和更合理的解决办法就是你在设备的新版本中做出变化。你可以让新版本设备在老耳机插入的时候自动切换成像旧版本那样工作[①]。

由于你想革新，想在设备中增加新的功能和特性，所以你需要为新版本的设备专门开发一个新协议。这样做的意义在于，当设备与附件一起工作的时候，它们首先会进入协议磋商过程，看看是否彼此都理解最新的协议。

微软公司就是在“版本对多”的环境中成长起来的。

但是，现实中还存在一个更麻烦的“多对多”（MANY-TO-MANY）市场。

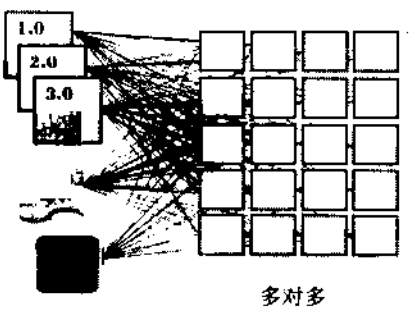
这样说吧，几年以后，你还在疯狂地出售Qxyzrhjjjjukltk播放器，但是市场上已经出现了很多克隆产品，比如开源的FireQx。除此以外，市场上还有很多耳机。你一直在为播放器增加新功能，耳机的插头也因此不断发生变化。这让整个耳机市场怨声载道，因为耳机厂商不得不在每一种Qxyzrhjjjjukltk克隆机上测试新版本的耳机，希望新耳机对克隆机也保持兼容。但是这样做成本很高，而且很费时间。坦白说，大部分的厂商没有那么多时间，于是他们就只保证耳机在最流行的Qxyzrhjjjjukltk版本（比方5.0版）下能工作。如果能做到这一点，他们就很高兴了，但是要是这样，当你将他们的耳机插入FireQx 3.0的时候，就免不了会发生问题，耳机在你的手里炸掉了。原因是在设计规格说明书中有一段叫做hasLayout

① 其实，这正是IE 8中添加“兼容性视图”按钮的初衷。——编者注

的文档[1]写得非常晦涩，没有人能够真正全部理解，大家多多少少都有一点误解，每一个厂商按照自己的理解做出了产品。打个比方，根据设计规格说明书，如果下雨，hasLayout就为真，电压就要上升，驱动车辆挡风玻璃上的雨刷。这个部分大家都看懂了，但是对于冰雹和下雪是否属于下雨，大家存在争议，因为设计规格说明书中根本没提到这一点。FireQx 3.0的处理是，把下雪看成是下雨的一种，理由是下雪的时候你也需要打开雨刷将挡风玻璃上的雪刮走。但是，Qxyzrhjjjjukltk 5.0没把下雪认定为下雨的一种，原因是开发这个部分的程序员住在火星上的热带，那个地方不下雪，而且他从没拿到过驾照，也就从来没开过车。对，你没看错，火星上也要有驾照。

渐渐地，一个无聊的讨厌家伙在她的网志上写了一篇很长的文章，详细地解释了一种破解方法，让Qxyzrhjjjjukltk 5.0能够像FireQx 3.0那样运行。她的做法是利用Qxyzrhjjjjukltk 5.0中的一个漏洞，让Qxyzrhjjjjukltk将融化的雪花误判为下雨。这种做法破坏了原来的系统，但是每个厂商都采用了，因为这样就能解决hasLayout不兼容问题。可是，Qxyzrhjjjjukltk开发小组在6.0版本中补上了这个漏洞，于是你又有麻烦了，你不得不寻找其他可以利用的漏洞，才能使得耳机在每一种设备上都能工作。

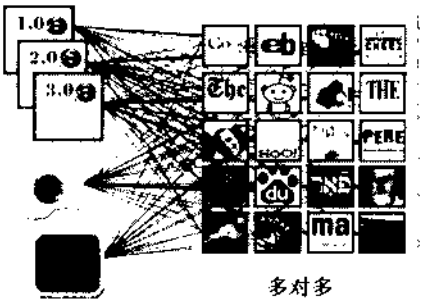
这就是“多对多”市场。下图中，左边是很多互不协调的厂商，右边是无数互不兼容的配套产品。两边都在犯错误，但是又能说什么呢，毕竟，人非圣贤，孰能无过？



多对多

① hasLayout是IE 5及其后续版本中出现的一个非标准属性，该属性的值决定了IE会将哪些元素看作布局元素，以确定显示方式。可是，开发人员不能直接将该属性设置为真或假。这个属性的值要通过某些IE开发人员认为合理的CSS声明来间接修改。由于大多数CSS设计人员搞不清哪条CSS规则会触发该属性的值发生变化（由真变为假，或反之），甚至压根儿就不知道有这么个属性存在，因此导致了很多网页在IE中无法正常显示。最终，只能依靠某些知情人士搞出的破解方法来修正这些问题。由此可见该属性给Web设计造成了多大的混乱。——编者注

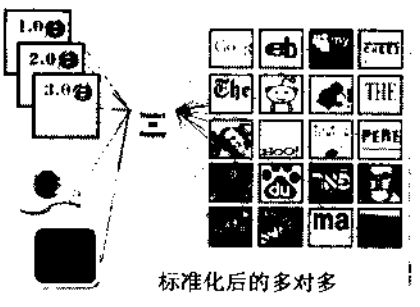
毫不奇怪，这就是我们在浏览HTML网页时面对的局面。许多种浏览器，对上大概数十亿张的网页。



多对多

程序员最可信赖的求职帮手

许多年来，“多对多”市场上发生的事情就是，一直有人呼吁建立“标准”，这样“所有的厂商”（其实是指小厂商）都有平等的机会，能够正确展示全部80亿张网页。而且更重要的是，依据“Web标准”，那80亿张网页的设计师们只需要在一种浏览器中测试，就会知道自己的网页能否在其他浏览器中正确显示，而不用在每一种浏览器中测试每一个网页。



标准化后的多对多

你看，这里的想法就是，不进行“多对多”状态下的测试，而是进行厂商的标准测试和配套产品的标准测试，总的测试数量就会急剧减少。更不用说，你的网页从此不用部署针对某一种特定浏览器的代码，不用担心哪种浏览器本身的错误会影响网页正常显示，因为在这种简单的情况下，浏览器根本不会出错。

但是，这种想法太理想化了。

现实中，对于互联网来说，还有一个难题没有解决：实际上，你没有办法测试一个网页是否符合标准，因为没有可靠的测试工具能够保证，只要通过它的测试，网页就能在所有浏览器上正常显示。这样的测试工具根

本不存在。

所以，你只好在自己的头脑中“测试”，完全用个人的思考来判断网页是否符合标准。你的依据就是标准的技术文档，但是你可能以前从没读过这些文档，或者就算你读了，你也做不到完全理解。

标准文档极其难以读懂，里面都是这样的句子：“如果一个插入区块（run-in box）后面跟着一个同级（sibling）的区块（该区块的位置不是浮动的，而且没有被绝对定位），那么插入区块变为其所在行中的第一个区块。一个插入区块不得插入另一个块，如果后者已经以一个插入块开始，或者本身就是一个插入块……”每当我读到这样的文字，我就很纳闷，怎么可能有人能够正确地遵循这样的标准呢。

你在现实中找不到方法测试你写的网页是否符合标准。你可以用校验器（validator），但是它们不会告诉你网页在浏览器中会显示成什么样。就算你的网页通过了“校验”，依然可能发生文字重叠在一起或者各个部分都没有对齐的现象，所以“校验器”对你而言并不是非常有用。人们实际上的做法是在一两种浏览器中测试自己的网页，看看能否正确显示。如果网页存在错误，但是碰巧在IE和Firefox中都显示正常，那么作者根本无从知道错误的存在。

等到将来新的浏览器问世时，这些网页可能就会出现问题。

犹太人中的极端正统派（ultra-orthodox）完全同意并且坚决地实践犹太宗教经典中的每一个字。但是，如果你去过耶路撒冷的那些极端正统派的犹太社区，你就会发现，你根本找不到任何一个社区的拉比[①]愿意去吃另一个社区的拉比家中的食物，尽管犹太人对于什么是“清洁的”食物并没有分歧。网页设计师现在的处境，其实住在Mea Shearim区[②]的犹太人早在几十年前就体会到了，那就是所有人都同意遵守同一部经典并不能保证所有人的行为都符合同样的规范。因为法条太复杂、太深奥、太难懂了，想要完全正确地理解法条几乎是不可能的，难免会落入陷阱或遇到地雷，所以比较安全的做法就是，你到别人家中做客的时候只吃水果。

① 拉比（Rabbi）是犹太宗教学者的尊称，是一个犹太社区中的宗教精神领袖。

② Mea Shearim区是耶路撒冷西部历史最悠久的犹太人居住区。

标准当然是很重要的，但是你不能迷信标准，你必须理解由于人会犯错，所以标准有时候会引发误解、困惑，甚至是争议。

回到我们的话题，这里的问题准确地说是这样的，因为无法准确测试网页是否符合标准，所以这个标准有点儿像是假冒的，不是一个真正的标准。它太理想化了，存在着许多误读，因此这样的标准无法达到预期的目标，无法减小“多对多”市场中庞大的测试数量。

DOCTYPE是一个神话。

一个平庸的网页设计师在网页头部加上DOCTYPE说明，然后就声称“这是一个标准HTML网页”。这是盲目自大的行为。他们无法得出这样的结论。他们唯一能说的就是，这个网页的目标是要符合标准的HTML规定。他们真正能做的就是，在IE、Firefox，也许还有Opera和Safari中测试这个网页，看看它是否能够正常显示。否则，他们就只是按照书上说的将DOCTYPE说明复制在页面的头部，对这样做的意义一无所知。

现实世界中的人是不完美的，单单一个设计规格说明书无法作为标准。必须还有极其严格的参考实现，并且每个人都必须使用这个参考实现来进行测试。否则，你会得到17种不同的“标准”，这还不如一种也没有呢。

这个问题产生的根源可以追溯到1981年，当时Jon Postel提出了鲁棒性原则（robustness principle）：“对于己方的行为要保守（conservative），对于他方的行为要宽容（liberal）。”（tools.ietf.org/html/rfc793）他想说的意思是，让一个协议健壮地运行的最好方法就是，每个人都要非常非常小心地让自己的行为符合规范，而且同时在与合作伙伴进行信息交换时，又要采取极端宽容的态度。也就是说，只要你能猜出对方的意思是什么就可以了，不用苛求对方的行为一定要符合规范。

举一个例子，一个段落用小字体显示，在技术上的正确写法是<p><small>，许多人写成<small><p>，那样写是错的，而且大多数人都没意识到为什么是错的。但是，浏览器放过了这些错误，自动将文本显示为小字体，因为浏览器认为作者本来的意图显然就是这样的。

现在，网上到处都是有错误的网页，原因就是所有早期的浏览器开发者制造出了超级宽容、超级友善、无比包容错误的浏览器。它们太爱你了，根

本不在乎你犯了错。许许多多的错误就是这样出现的。Jon Postel的鲁棒性原则并没有真正起作用。这个问题许多年来都被忽视了。直到2001年，Marshall Rose终于写出了下面的话（tools.ietf.org/html/rfc3117）。

> 与人们的直觉不同，Jon Postel的鲁棒性原则（“输出行为要保守，接受行为要宽容”）经常导致部署上的难题。原因是新的软件最初发布后，它很可能只需要与既有软件的一些子集交换数据。如果那些软件遵循鲁棒性原则，那么很可能发现不了新的软件中存在的错误。所以，新的软件会得到一些部署，虽然不一定很广泛。类似地，其他新软件也必将得到一些部署。最终，那些不怎么正确的软件一定会遇到没有最早的子集那么宽容的软件。读者应该能够想像接下来会发生什么。

我们应该向Jon Postel致以敬意，他为互联网的发明做出了巨大贡献。我们没有理由苛责他，要求他为鲁棒性原则的声名不佳负责。1981年相当于史前时代。如果你跑去对Jon Postel说：未来将有9000万没有受过专业训练的人在建设网站，其中没有一个是工程师，他们会写出各种各样可怕的网页，要是在这些人身上给予不适当的仁慈，就会使得早期的浏览器开发者认可网页中的错误并将网页显示出来。Jon Postel听了你的这番话，可能就会明白鲁棒性原则是一个错误的原则，想通了在“Web标准”这个问题上坚持理想主义实际上是正确的，互联网本来就是应该按照非常非常严格的标准来建造，任何一种浏览器就是应该丝毫不留情面地向网页作者指出所有错误，对于那些不能搞清楚如何做到“自己的行为要保守”的人，就不能允许他们把网页放上网，直到他们身体力行为止。

但是，不得不说，如果历史变成那样，那么互联网的迅猛发展也许永远不会出现，我们所有人今天使用的也许还是AT&T运营的庞大无比的Lotus Notes网络。光这样想想就让人不寒而栗。

有人可能会感叹，早知如此，何必当初呢！但无论如何，一切已经这样了。我们无法改变过去，我们只能改变未来。见鬼，其实我们也几乎无法改变未来。

如果你是Internet Explorer 8.0开发小组中的实用主义者，那么Raymond Chen的话也许已经烙印在你的大脑皮层上了。他介绍了Windows XP如何被

迫继承旧版本Windows的错误（blogs.msdn.com/oldnewthing/archive/2003/12/23/45481.aspx）。

> 请从用户的角度来考虑这个问题。你先购买了软件X、Y和Z，然后才升级到Windows XP。但是从此你的计算机就开始莫名其妙地死机，并且软件Z根本无法运行。你就会对你的朋友说："别升级到Windows XP，它会莫名其妙死机，而且同软件Z不兼容。"试问在这种情况下，你会自己去研究系统，确定死机是由软件X引起的吗？你会自己去发现软件Z无法运行是由于它使用了Windows内部一条隐藏的消息吗？当然不会。你只会去将Windows XP退货。（你购买软件X、Y和Z是在几个月前，已经过了30天的退货期。你唯一能够退掉的就是Windows XP。）

你看懂了这段话，对吧？根据现在的状况，对这段话做一下更新。

> 请从用户的角度来考虑这个问题。你先购买了软件X、Y和Z，然后才升级到Windows ~~XP~~Vista。但是从此你的计算机就开始莫名其妙地死机，并且软件Z根本无法运行。你就会对你的朋友说："别升级到Windows ~~XP~~Vista，它会莫名其妙死机，而且同软件Z不兼容。"试问在这种情况下，你会自己去研究系统，确定死机是由软件X引起的吗？你会自己去发现软件Z无法运行是由于它使用了Windows内部一条~~隐藏~~不安全的消息吗？当然不会。你只会去将Windows ~~XP~~Vista退货。（你购买软件X、Y和Z是在几个月前，已经过了30天的退货期。你唯一能够退掉的就是Windows ~~XP~~Vista。）

在微软公司内部，理想主义者战胜了实用主义者，我在2004年已经谈过这个问题了。这是Windows Vista恶评如潮、销售惨淡的直接原因。

这种情况是怎么发生在IE开发小组身上的呢？

还是从用户的角度考虑这个问题。你每天浏览100个网站。当你升级到IE 8以后，一半的网页都不能正常显示了，Google地图索性就不能用了。

你会对你的朋友说："别升级到IE 8。在IE 8中没有一个网页是正常的。Google地图根本不能用。"试问在这种情况下，你会自己去查看网页源码，确定网站X用了不标准的HTML代码吗？你会自己去发现Google地图不能用是

因为它使用了IE旧版本中从未被标准委员会认可的JavaScript对象吗？当然不会。你只会将IE 8卸载了。（你控制不了那些你想看的网站。它们的开发者中有一些已经死了，没有办法再更改网页了。你唯一能做的就是回到IE 7。）

如果你是IE 8开发小组的成员，你首先着手要做的正是在这种“版本对多”市场上总是有效的事情。那就是你先要做一点协议磋商的工作，然后针对那些不明确表示接受新处理方法的网站，就继续用以前老的处理方法，这样一来所有现存的网页都能够继续正常显示。你将新的处理方法只用于那些挥着旗帜、高声呐喊“嗨，我与IE 8心心相印！快把IE 8的所有好东西都用在我身上”的网页。

事实上，这就是IE开发小组做出的第一个决定，1月21日对外宣布的。IE 8浏览器无需任何设置就能兼容现存的网页，所以网页设计师无需调整自己的网站。这样，IE 8就不会像错误多多的旧版本IE 7那样遭到网页设计师的痛恨。

实用主义工程师就此得出结论，认为IE开发小组的这第一个决定是非常正确的。但是理想主义的年轻“标准”爱好者却气得简直要发动核战争。

他们说，网页上不应该有特殊标签，不应该有特殊的“嗨！本网页已经过IE 8测试”之类的说明，IE 8就是应该直接向访问者提供符合网络标准的体验。他们痛恨不通用的标签，痛恨每个该死的网页都必须做过37种丑陋的破解（hack），然后才能在五到六种流行的浏览器中全都正确显示——我们已经受够了那些丑陋的破解，让80亿张现存网页见鬼去吧！

IE开发小组的态度就这样出现了180度的大转变。他们做出了第二个决定。我不得不认为这还不是最后决定。他们的第二个决定是采用理想主义的方式，只要任何网站声称自己是“符合标准的”，IE 8就认定它们是为IE 8设计并测试过的网站，直接将它们显示出来。

几乎每一个我用IE 8访问过的网站，多多少少都不是完全正常的。那些使用大量JavaScript的网站通常会彻底死在那里。许多页面只是显示问题：内容出现在错误的位置，向上跳出的菜单改为向下跳出，页面的中间出现了神秘的滚动条。另一些网站出的问题则不是很明显，它们看上去还算正常，但是当你想进一步使用的时候，你发现关键的表单根本无法提交，或者提交以后就直接跳转到一个空白页面上去了。

这并不是那些网页的错误。它们通常都被仔细地编写过，保证符合网页标准。但是IE 6和IE 7本身并不是真正地按标准处理网页，所以这些网页使用了一些小小的破解手段，比如“遇到Internet Explorer时，……将内容向右移动17个像素，修补IE本身的bug”。

IE 8也是IE，但是它没有了IE 7的bug，不再把内容从符合网页标准的位置向左移动17个像素了。所以现在，那些早前的代码不再正常工作，这是完全合理的结果。

IE 8不能正确显示大多数网页，除非你最后决定放弃并按下“模拟IE 7运行”按钮。但是，理想主义者不在乎这个。他们一心一意要强迫那些网页做出调整。

有些网页是没法调整的。它们要么被烧进了CD-ROM，要么作者已经不在人世了。大多数网页的所有者根本摸不着头脑，不知道发生了什么事情，为什么他们四年前雇用设计师做出的网页如今却不能正常显示了。

理想主义者对这样做感到满意了。好几百个这样的人突然涌进了微软公司的论坛，生平第一次对微软说出了一些真正的好话。

我看了一下表。

滴答，滴答，滴答。

就在这几秒的时间里，你就看到论坛中有人跳出来，说了像下面这样的话（forums.microsoft.com/MSDN/ShowPost.aspx?PostID=2972194&SiteID=1）。

> 我已经下载了IE 8，现在遇到了一些错误。我访问的网站中，有一些很难看清，比如惠普公司的网站，字体变得非常非常小……在某些情况下，我上网的网速也变慢了。当我使用Google地图的时候，页面都重叠在一起，没法用了！

嗯嗯嗯。你们所有这些自以为是的理想主义者，都在嘲笑这个菜鸟/白痴。但是，消费者不是白痴。她是你的爱人。所以，请停止嘲笑。世界上98%的人会去安装IE 8，然后说：“它有错误，我不能浏览想看的网站。”他们不会为你们那愚蠢的信仰叫好，不会理解为什么你们一定要网络浏览器符合某些神秘的、柏拉图式理想化的、实际上根本还没有实现的“标准”。他们不

想听你们讲那些丑陋的破解。他们只想要网络浏览器能够正常浏览现实中的网站。

你们看到了，这是一个很好的例子，说明了两大阵营之间存在巨大的鸿沟。

Web标准的理想主义阵营看上去有点像托派分子[①]（Trotskyist）。你觉得他们是左翼，但是如果你写的网站声称符合网页标准，但是实际上不符合，那么那些理想主义者就会摇身一变，成为“美国最严厉的警长”Joe Arpaio[②]：“你犯了错误，你的网站就活该出问题。我根本不关心你的网站中是不是有80%的网页都不工作了。我要把你们这些人全都关进监狱，让你们穿上粉红色[③]的囚衣，吃15美分的三明治，戴着脚镣做工。我不在乎哪怕整个县的人都被关进监狱。法律就是法律。”

另一个阵营中的人都是实用主义的工程师类型，多愁善感，待人热情，愿意通融：“难道我们不能将IE 7模式定为默认模式吗？一行代码就可以做到了……马上动手！解决了！”

这种事情会秘密发生吗？下面是我对未来的一点预测。IE 8开发小组将会向所有人公告，IE 8默认启用网页标准模式。在此后非常漫长的测试期中，他们恳求用户用IE 8来测试自己的网页，请用户自己设法使得网页在IE 8下能够正常工作。但是随着正式发布日期的临近，整个互联网上只有32%的网页能够被IE 8正确呈现。他们将会说：“伙计们，你们都看到了，我们真的是非常抱歉，我们确实想要让IE 8的标准模式成为默认模式，但是我们不能发布一个不能正常浏览世界上大多数网页的浏览器。”于是，他们转向了实用主义立场。另一种可能是，也许他们不会转变立场，因为实用主义阵营在微软公司内部已经失势很长一段时间了[④]。如果是这样，IE将失去很多市场

① 托派分子指的是赞同托洛茨基理论的人。这里可以简单理解为赞同不断的渐进式革命方式。

② Joe Arpaio是美国亚利桑那州凤凰城Maricopa县的警长，自称是“美国最严厉的警长”，因为他对监狱管理和预防犯罪采取严厉措施。在Maricopa县每个街区都有监视器，买辆自行车也要到警察局登记，关在监狱里的犯人都要戴脚镣，不得抽烟和看电视，只能吃廉价食品。

③ “粉红色”在英语中的单词是pink，这个词的另一个意思是“略带左翼政治观点的”。

④ 2008年3月19日IE 8正式发布（作者写下本文后仅两天），结果采取了折中的立场，即默认启用最符合标准的模式，同时提供了“兼容性视图”按钮。——编者注

份额，但是理想主义者不会感受到丝毫打击，Dean Hachamovitch的巨额年终奖金可能一分钱也不会少。

你明白了吗？根本就没有正确答案。

像往常一样，理想主义者在大原则上是百分之百正确的。同样，像往常一样，实用主义者在现实中也是正确的。他们之间的口水战将持续许多年。这场争论将精确地把世界分成两半。如果你有办法找到一种从事互联网口水战事业的股票，那么现在就是买入的好时机。

18

为什么Microsoft Office的文件格式如此复杂（以及一些对策）

2008年2月19日，星期二

上个星期，微软公司公开了Office的二进制文件格式的规格说明书。这些文件格式简直是彻底疯狂了。Excel 97-2003的文件格式用了349页的PDF文件来说明。且慢，这还不是全部！这个说明书中还包括了下面这一句有趣的注解：

每一个Excel工作簿都储存在一个复合文档[1]中。

你看到了吗，Excel 97-2003文件实际上是OLE[2]复合文档，也就是说，在单个文件中就包含了一个文件系统。这真是太复杂了，你不得不再读一份9页的规格才能搞清楚。这些“规格说明书”看上去更像C语言中的数据结构，而不像我们传统上认为规格说明书应有的样子。这些文件格式就是一个完整的层级文件系统（hierarchical file system）。

如果你开始阅读这些规格说明书，心里盼着利用一个周末写出一些漂亮的代码，让你的网志系统能够输入Word文档，或者把你的个人财务数据整

① “复合文档”的全称是“复合文档二进制结构”（Compound File Binary Format）。这是微软公司开发的一种文件格式。在这种格式中，文件内部模拟成磁盘文件系统，也采用目录式的树状结构，每一个目录里面可以存放子目录和其他复杂的二进制对象。详细介绍参见http://en.wikipedia.org/wiki/Compound_File_Binary_Format。

② OLE是Object Linking and Embedding（对象连接嵌入）的缩写。一个复合文档通过连接和嵌入其他对象，诸如文本、日历、动画、声音、视频、控制组件等，就好像是一个能够包含各种视觉组件和信息组件的显示桌面。

理成Excel格式的电子数据表。那么，这份规格说明书的复杂和冗长可能很快就能让你打消这些念头。一个普通的程序员对Office的二进制文件格式将得出如下结论：

- 故意搞得那么复杂，不想让人看懂；
- 设计者简直是一个疯狂的伯格人[①]（Borg）；
- 设计者是一个水平极差的程序员；
- 不可能读懂它们，也不可能做出正确的产品。

这4条罪状你都说错了。我来告诉你一点内情，让你明白这些文件格式怎么会复杂得如此难以置信，为什么它们不能说明微软的编程水平很糟糕，并告诉你一些对策。

首先，你要理解，二进制文件格式的设计目的完全不同于非二进制文件格式，比如HTML。

为了能够在很老式的计算机上快速打开文件，二进制文件格式才被设计出来。早期的运行在Windows系统上的Excel需要的内存大小一般是1MB，而且必须要保证它在20MHz主频的80386机器上能够流畅地运行。在文件格式方面，必须做许许多多的优化工作，才能更快速地打开和保存文件。

- 因为文件格式是二进制的，所以读取一个记录往往就是单纯地从磁盘向内存复制（又称“快速传送”）一些字节而已，你直接就得到了一个C语言下的数据结构，可以拿来就用。在装载文件的过程中，没有任何的词义分析（lexing）或解析（parsing）步骤。这两个步骤比“快速传送”要慢上几个数量级。
- 这些文件格式中还有一些特别的设计，在必要的时候，可以使常见操作的速度变得更快。比如，在Excel 95和Excel 97中，有一种叫做“简单保存”（Simple Save）的功能，它可以将文件保存为OLE复合文档格式的一种更快的变体，OLE复合文档格式本身对主流应用来说有点慢。Word也有一种叫做“快速保存”（Fast Save）的功能，可以使长文档的保存速度变快，大概可以快14倍到15倍。它只是将文档的变动追加在文件的末端，而不是从头改写整个文档。按照当时硬盘的速度，这意味着保存一个长文档不用再花30秒了，只要1秒就

① 伯格人是美国科幻电视剧《星际旅行》（Star Trek）中一种半机械化的生物。

够了。（这也意味着文档中被删改的部分仍然保留在文件中。后来证明这样的设计并不理想。）

这些设计是以库（library）为基础的。如果你想从头写一个二进制文档的输入器，你就必须要支持诸如Windows元文件格式（Metafile Format）（做图功能要用到）和OLE复合储存（Compound Storage）这样的东西。如果你的系统是Windows，那么这样的功能都有库支持，开发起来就是小菜一碟……使用这些库是微软开发团队的捷径。但是如果你想自己从头写出所有东西，这些库的作用就不得不由你自己写出来了。

Office大量使用复合文档，比如，你可以在Word文档中插入一张电子表格。一个完美的Word文件格式解析器必须能够智能地处理内嵌的电子表格。

这些设计没有考虑到数据的可交换性（interoperability）。设计者假定Word文件格式只能被Word读写。这在当时是一个相当合理的假设。这意味着，无论何时一个Word开发小组中的程序员想要对文件格式做出某种变化，他只要考虑两点：这种变化是否加快了速度？怎么才能对Word代码库中的代码做最少的修改？像SGML、HTML这种可交换的、标准化的文件格式，它们的设计思想一直没有真正得到认同，直到互联网的出现第一次使得可交换文档成为可行，情况才发生了变化。这要比Office二进制文档的问世时间晚了十年。Office设计者一贯的假设是，你可以用输入（importer）和输出（exporter）功能进行文档交换。事实上，Word中确实有一种格式被设计成易于与其他软件进行信息交换，这种格式叫做RTF。几乎Word软件一问世，它就在那里。直到今天，它依然百分之百被支持。

这些设计必须能够反映应用程序所包含的所有复杂性。每一个复选框，每一个格式选项，Microsoft Office的每一种功能，都必须在文件格式中有所反映。你记不记得，Word的“段落”（paragraph）对话框中，有一个叫做“与下段同页”（Keep With Next）的复选框？它的作用是在必要时让一个段落移到下一页去，与后面的段落同时显示在一个页面上。文件格式中必须对这个复选框做出安排。这意味着如果你想做出一个完美的Word克隆品，让它能够正确地读出Word文档，你就必须把这个复选框也考虑在内。假定你正在写一个同Word进行竞争的文字处理软件，其中包含读入Word文档的功能，你就写了一段代码，使得你的软件能够从文件格式中读取这个复选框，这可

能只花掉你一分钟，但是改变软件中页面布局的算法，以便显示出“与下段同页”的效果，却要花掉你几个星期。如果你不愿意费这个功夫，那么客户用你的软件打开他们的Word文件，就会发现所有的页面都乱成一团。

这些设计必须能够反映应用程序的历史。在这些Office文件格式中，大量的复杂之处反映了许多过时的、难懂的、不受欢迎的、少有人使用的功能。为了同以前版本兼容，它们依然被保留在文件格式中，反正留着这些代码不会让微软多支出一分钱。但是如果你真地想完整彻底地解析和生成这些文件格式，你就不得不把所有的工作都重做一遍，其中就包括15年前微软的某些实习生留下的代码。我的意思是，几千名程序员多年的工作量才积累成了今天的Word和Excel，如果你真想把它们完整地克隆出来，你就不得不自己完成这些工作量。一种文件格式就是某种应用软件所支持的所有功能的简明摘要。

说点儿有意思的事，让我们仔细来研究一下一个小例子。Excel工作表是不同种类的BIFF记录[①]的集合。我想看规格说明书中第一个BIFF记录类型是什么。这个记录类型叫做1904。

Excel文件格式的规格说明书对此写得非常模糊。它只是说1904记录类型表明“是否使用了1904日期系统”。啊，多么经典的毫无用处的说明书啊。如果你是一个正在使用Excel文件格式的程序员，在文件格式规格说明书中读到了这样的文字，你也许会很有信心地推断，微软一定把有些内容藏起来没说。规格说明书中的说明没有提供足够的信息。你还需要额外的知识，下面我就来为你补上这些知识。Excel工作表共有两种类型：一种的日期元年是1900年1月1日（这种格式中包含一个闰年的计算错误，这是为了与Lotus 1-2-3兼容而故意留下的。要把这个事情说清楚未免太枯燥了，这里就省略了）；另一种的日期元年是1904年1月1日。Excel同时支持这两种日期类型的文件，原因就是Excel的第一个版本是为Mac系统开发的，当时直接使用了操作系统的日期元年，这样做比较容易。但是，Windows系统上的Excel必须能够输入Lotus 1-2-3文件，后者是用1900年1月1日作为日期元年的。这足以让人欲哭无泪。回顾整个历史，程序员无时无刻不想做出正确的事情，但是有时候你不得不向现实屈服。

---

① BIFF是Binary Interchange File Format（二进制可交换文件格式）的缩写，这是微软为Excel开发的一种专用文件格式。

1900和1904这两种日期类型的文件在实际应用中都很常见，往往取决于文件是出自Windows系统还是Mac系统。在后台自动将一种格式转换成另一种格式会出现数据完整性方面的错误，所以Excel不会主动为你改变文件的类型。为了解析Excel文件，你就必须自己来处理这两种类型的文件。这可不是从文件中读取一个数据位这么简单。它意味着你必须重写所有你的日期显示和解析代码，这样才能处理两种日期元年。我想，你大概要花好几天才能写出来。

程序员最可信赖的求职帮手

说真的，在克隆Excel的进程中，你会发现所有关于日期处理的微妙细节。Excel何时将数字转化为日期？如何进行日期的格式化？为什么1/31被当作本年度的1月31日，而1/50被当作1950年的1月1日？要将所有这些Excel行为的细微之处完全写清楚，就必须写出一个与Excel源代码包含同样信息量的文档。

这只是你不得不处理的几百种BIFF记录类型中的第一种。它们中的大多数都极其复杂，足以让一个成年的程序员绝望地哭出来。

唯一可能的结论是这样的：微软公司公开这些Office文件格式，这对其他人会很有帮助，但是并不会使得输入或生成Office文件格式变得哪怕容易一点点。Office软件是复杂得不可理喻、功能极其丰富的应用软件。你不可能只实现其中最常用的20%的功能，然后指望80%的用户会感到满意。这些二进制文件格式的规格说明书，顶多只是让你在对一个极其复杂的系统进行反向工程（reverse engineering）时，节省了几分钟而已。

别着急，我答应过提供对策。好消息是对于你要开发的几乎所有常见的应用程序，试图去读入或生成Office二进制文件格式都是错误的选择。有两种可行的替代方案，你可以认真考虑是否采用。一种是让Office自己去处理这些问题，另一种是使用更易于生成的文件格式。

让Office为你干脏活。Word和Excel都有极端完整的对象模型，可以通过COM自动化（COM Automation）进行调用，因此允许你通过编程完成任何任务。在许多情况中，使用Office内部的代码要比你自己从头写起轻松得多。下面是几个例子。

(1) 你有一个互联网应用程序，它的一个功能是将现有的Word文件转化成PDF格式输出。我的实现方法是用若干行Word VBA代码，先读入这个文

件，然后用Word 2007内置的“PDF输出”功能将其保存为PDF格式。你可以直接调用这些代码，甚至在IIS环境中直接用ASP或ASP.NET语言调用也可以。第一次调用的时候，代码会启动Word，这要花掉几秒钟。第二次调用的时候，Word已经在内存中了，因为COM子系统会把Word在内存中保留几分钟，以便你再次使用它。这样对一个访问量适中的互联网应用程序来说，速度就足够快了。

(2) 接着上一个例子。但是你的网站架设环境不是Windows，而是Linux。那么，你可以去买一个Windows 2003服务器，然后在上面安装一份完整授权的Word，再架构一个专用的Web服务来提供这个功能。这只需要用C#和ASP.NET语言，半天就可以写出来了。

(3) 还是这个例子。现在你需要扩展这个功能。那么，你可以添置任意多台第(2)步中建立起来的服务器，然后在它们前面做一个均衡负载（load balancer）。不用写任何代码。

你在自己服务器上想要完成的所有一般性Office应用，上面这种方法都可以做到。比如下面几种情况。

- 打开一个Excel工作簿，在输入单元（input cell）储存一些数据，然后重新计算，再将数据从输出单元（output cell）中取出。
- 用Excel生成GIF格式的图表。
- 直接从任何种类的Excel工作簿中取出任何种类的信息，完全不考虑文件格式。
- 将Excel文件格式转为CSV表格数据的格式（另一个方法是通过Excel ODBC驱动，使用SQL查询语句将数据取出来）。
- 编辑Word文档。
- 填写Word表格。
- 在Office支持的许多种不同的文件格式之间互相转换（Office提供的输入器一共支持几十种文本格式和电子表格格式）。

在上述所有的情况下，都有办法告诉Office对象程序不是运行在互动模式下。这样一来，Office就不会主动去刷新屏幕，也不会提示用户输入。顺便说一句，如果你选择走这条路，那么要注意一些容易出错的地方（gotcha），而且微软并没有提供正式支持，所以在你开工之前，要读一下微软知识库中的相关文章。

用一种更简单的格式生成文件。如果你只是想利用程序生成Office文档，几乎一定可以找到比Office二进制格式更好的格式，它能够被Word和Excel顺利打开，不会出任何差错。

- 如果你要生成的只是在Excel中能够使用的表格数据，那么考虑使用CSV格式。
- 如果你真地需要CSV不支持的工作表计算功能，那么WK1格式（Lotus 1-2-3使用的格式）绝对比Excel简单多了，Excel打开它一点问题也没有。
- 如果你真地、真地一定要生成Excel特有的格式，那就去找一个老掉牙的Excel版本（Excel 3.0就很不错），里面不存在任何与复合文档有关的东西。然后，你生成一个最小化的文件，里面只包含你想要使用的那些功能。你用这个文件观察那些你的输出中必须包含的BIFF记录类型能被简化到什么地步，接着就去仔细研读规格说明书中的这些部分。
- 对于Word文档，可以考虑生成HTML格式。Word能够很好地支持这个格式。
- 如果你一定要生成带有花哨格式的Word文档，你的最佳选择是生成RTF格式。Word能做的每一件事，RTF格式都有办法表达出来，而且RTF是一种文本格式，不是二进制格式，所以你能对RTF文档做出修改，Word还依然能够打开它。你可以先在Word里创造一个格式化的漂亮文档，里面包含了占位符，然后将它保存为RTF格式，再用简单的文本替代，实时地将占位符换成你想要的文字。这样就得到了一个RTF文档，所有版本的Word都能够顺利地打开。

总地来说，不管你要解决的到底是什么问题，最浪费人力的方法可能就是直接读取或生成Office二进制文件格式了。只有一种情况例外，那就是你确实想要写出自己的软件，它能与Office竞争，能够完美地读取和生成所有的Office文件。但是那样的话，就足有相当于几千年的工作量等着你去完成了。

19

要挣钱，就别怕脏

2007年12月6日，星期四

我小时候在一家面包厂里做工，生面粉最让我头痛。它很粘，到处都会沾上，很难去除。放工回家，我的头发里都是一小团一小团的面粉。每次上工时，我都要花几个小时把机器里的面粉刮下来。我的裤兜里随身就放着面粉刮刀。有时候，一大块面粉不知为何出现在不应该出现的地方，于是所有事情都变得一团糟。面粉成了我的噩梦。

我在工厂的生产车间工作，另一边就是包装和发货车间。他们最头痛的事情是面包屑。到处都有面包屑。发货工人回家的时候，头发里都是面包屑。每次上工，他们都要花几个小时把机器里的面包屑清除。他们的裤兜里都揣着一把小刷子。我相信面包屑也是他们的噩梦。

很有可能的是，不管别人雇你干什么工作，你都会遇到某种很不顺心的麻烦事。就算你不用跟干面粉或者面包屑打交道，只要你在刀片厂工作，回家的时候，你的手指上可能就都是小伤口。如果你为VMware公司[①]工作，你的噩梦就是视频游戏所依赖的高端显示卡总是出错。如果你的工作是开发Windows操作系统，你的噩梦就是代码中小小的变化就会引发几百万的旧程序和老硬件停止工作。它们就是工作中让你不顺心的麻烦事。

我们眼下的麻烦事之一就是，让FogBugz运行在客户的服务器上。37signals的创始人Jason Fried对原因做过一个很好的总结（www.37signals.com/svn/posts/724-ask-37signalsinstallable-software）：“……（如果那样）你

① VMware是一家开发虚拟机的软件公司，它的产品VMware可以在一种操作系统中模拟运行另一种操作系统。

程序员最可信赖的求职帮手

就不得不与无穷无尽、各不相同的操作环境打交道，它们完全不受你的控制。因为你不能控制操作系统，也不能控制第三方软件和硬件（它们都会影响到你的软件的安装、升级或总体表现），那么万一哪里出错了，想要找到原因就相当困难。如果是远程服务器安装，而目标服务器上运行着各种版本的Ruby、Rails、 MySQL等，那么一切就会变得难上加难。”Jason的结论就是，如果他们的产品是需要安装的软件，那么他们的日子“绝对不好过”。他所说的让你“日子不好过”的工作，同我所说的“麻烦事”是一个意思。

但是，实情却是如果你为“麻烦事”找到了解决方法，市场就会向你支付报酬。解决轻而易举的事情是拿不到钱的。就像电视剧《四个约克郡男人》中说的：“要挣钱，就别怕脏。”

我们提供两种FogBugz，一种托管在我们的服务器上，另一种由客户安装在他们自己的服务器上。选择后者的客户是选择前者的4倍。对我们来说，FogBugz的可安装版本使销售额提高了5倍。我们增加的成本不过只是新增一两个人手（薪水按照技术支持人员的标准）。这也意味着我们不得不使用Wasabi[①]，相比常用的编程语言，它有一些很严重的缺点。但是我们发现，它能提供成本最低、效率最高的方式，使得我们将现有的代码变成分别在Windows、Linux和Mac平台上可安装的版本。说到这里，请不要误会我的意思。不会有什么事比抛弃可安装版本的FogBugz并用我们自己的服务器提供一切服务更让我感到高兴了。我们有一架子又一架子的、运作良好的Dell服务器，空余的容量有的是。如果由我们的服务器提供所有的服务，那么技术支持成本就会降到零。我们的日子就会好过多了。但是，我们挣的钱也会少得多，我们将会倒闭。

目前，许许多多样子可爱的创业公司都有一个共同点，那就是他们所有的产品就是一个小小的站点，背后的技术就是一些Ruby-on-Rails和Ajax，不解决任何“麻烦事”，而且别人很容易做出复制品。这类公司中相当一部分，给人的感觉就是不实在和空洞，因为他们没有解决实际中迫切需要解决的任何困难问题（整间公司就是3个小孩，另外加上一条宠物大蜥蜴）。只有等到他们解决了困难问题，他们对用户才是有用的。用户只向解决困难问题的公司付钱。

① Wasabi是一种跨平台的开发工具。

编写一个设计优雅、易于使用的应用程序，同解决“麻烦事”有相仿的效果。虽然看到成品的时候，你会觉得不难做，但实际上是很难的。就好比你在看精彩的芭蕾舞演出，你觉得演员跳起来很轻松，实际上换了你就困难无比。Jason和他的37signals致力于做出优秀的设计，他们得到了回报。优秀的设计似乎是最容易复制的东西，但是做起来却又不是那么容易，你看看微软试图仿效iPod的例子，就会明白这一点。做出优秀的设计本身就是一件“麻烦事”，实际上能够提供牢固得令人震惊的竞争优势。

说实话，Jason做出目前成果的一个可能原因就是他在设计方面有很高的天赋，所以他选择解决设计领域的“麻烦事”，因为看上去这对他来说不难做。同样地，多年以来，我一直是一个Windows程序员，所以用C++从头写出一个FogBugz的Windows安装程序，与所有那些烦人的COM组件打交道，看上去对我来说也不难做。

但是，如果你想保持增长，不管是个人，还是公司，那么唯一的方法就是扩张自己擅长处理的业务的边界。在将来的某个时刻，37signals开发团队可能会决定雇用一个人来写安装程序，允许客户将软件安装在本地的机器上，这样就会获得额外的收入和更高的利润，足以抵消成本。所以，除非他们故意要保持公司的小型化（这是一个再正常不过的愿望），否则将来他们可能就会逐渐下定决心，去干一些别的“麻烦事”。

当然，他们也可能决定不那样做。只干那些自己感兴趣的活，这一点错也没有。但是，要是换了我，我一定受不了。你决定只为一小部分特定的人群解决某些特定的问题，这肯定是无可厚非的。Salesforce.com坚持只在自己的服务器上提供服务，不为客户提供可安装的版本，最终业务也做得非常大。还有许许多多更小型化的软件工作室，它们毫无变成大公司的意愿，它们只想为自己的工作人员提供一种美妙的生活方式。

但是，重要的一点是，每当你新解决了一件“麻烦事”，你的业务和市场都会有巨大的增长。优秀的推广、优秀的设计、优秀的销售人员、优秀的服务，再加上你为客户解决了许多“麻烦事”，这些因素加总在一起，会创造出一种互相放大的结果。刚开始的时候，你有的只是一个优秀的设计，然后你加进了一些有用的功能，解决了许多“麻烦事”，你的客户数量就增加了3倍；接着，你搞了一些推广活动，你的客户数量又增加了3倍，因为现在许多人知道了你可以为他们的“麻烦事”提供解决方案；后来，你雇用了专

门的销售人员，你的客户数量又增加了3倍，因为销售人员帮助那些听说过你的产品的人们下定决心，让他们掏出钱来购买你的产品；再后来，你加进了更多的功能，为更多的人解决更多的问题，你就这样慢慢地得到了将业务做到足够大的机会，让你的软件能够影响到足够多的人，有可能使得世界变得更加美好。

再附上一笔。我在这里不是说，37signals只要提供Basecamp[①]的可安装版本，它的销售量就可以增加4倍。原因有好几个，首先，我们卖出去那么多FogBugz可安装版本的理由之一就是，在某些客户看来，可安装版本似乎更便宜。（从长期来看，可安装版本其实不便宜，因为你必须自己支付服务器和系统管理员的成本。）其次，对于可安装版本，我们的维护成本非常低，因为我们80%的客户都选择在Windows系统上运行FogBugz，而Windows系统的不同版本都很类似，所以我们只要支持不同版本之间的一个最小公约数就可以了，这会使工作变得简单许多。我们大多数技术支持的成本都是由于UNIX平台的多样性而引起的。UNIX平台的版本只占到销售额的20%，但是我估计，它们导致了我们80%的技术支持的工作量。如果Basecamp的可安装版本必须使用UNIX平台，那么相比Windows平台的可安装版本，技术支持成本就会高得不成比例。最后，我们的经验可能无法照搬到37signals的另一个理由是，我们销售可安装版本已经7年了， 销售托管版本刚满6个月，所以使用可安装版本的客户基础非常大。如果只看FogBugz的新客户，那么可安装版本与托管版本的销售之比就会降到3∶1。

① Basecamp是37signals的主要产品。

第五部分

编程建议

20 循证式日程规划

21 关于战略问题的通信之六

22 你的编程语言做得到吗

23 让错误的代码显而易见

20

循证式日程规划

2007年10月26日，星期五

软件开发者并不喜欢做日程规划。一般来说，他们总是想避开日程规划。“到了该完成的时候，事情就完成了！”他们嘴里这么说，心里盼望着这句又新颖又好玩的警句能逗得他们的老板张口大笑，然后大家就其乐融融，没人记得还有日程规划这档子事。

你见到的大多数日程规划都是马马虎虎，敷衍了事。它们被放在某个共享目录中，完全被忘记了。过了两年，开发小组完成开发任务后，某个怪里怪气的家伙翻出办公室里的文件柜，看到了多年前打印出来的日程规划。与现实情况两相一对照，这份日程规划惹得所有人都哈哈大笑。“嗨，你们看！我们居然将用Ruby语言重写代码的时间定为两星期！”

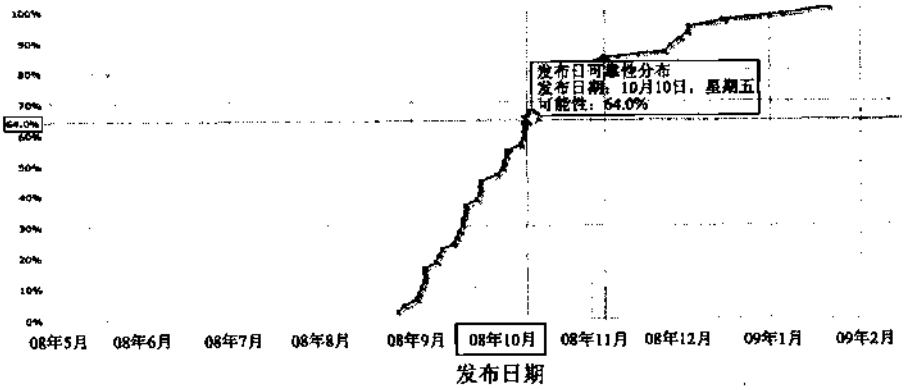
好笑吧？如果你还待在这个行业中，你一定看得懂这个笑话。

你的目的是最有效率、最物有所值地使用你的时间。但是，如果你不知道每项任务所要花费的时间，你就不可能找出最经济的工作方式。比如，现在有两项任务，一项是做一个卡通回形针的动画版，另一项是编写出更多的财务函数，你必须在两项任务中挑一项，这个时候你就需要知道每项任务所要花费的时间。

为什么软件开发者不太愿意做日程规划呢？有两个原因。第一个原因是做起来比较麻烦；第二个原因是，没人相信日程规划是可行的。如果一件东西可能是不对的，那又何必费事把它做出来呢？

大概是从去年开始，我们Fog Creek软件公司开发出一套系统，非常好用，就连我们中那些最挑剔的程序员都愿意使用它。在我们看来，它能生成

极其可靠的日程规划，就叫做循证式日程规划（Evidence-Based Scheduling），简称EBS。你先收集历史数据，主要来源是过去的工作时间记录单[①]，然后你将这些数据导入软件。这时，你所得到的就不仅仅是许许多多的数据，更重要的是，你得到了一条概率分布曲线，显示在任一个给定日期你完成工作的可能性有多大。它看起来就像下图这样。



这条曲线越陡峭，所得结果的可靠性（置信区间）就越大。

下面就是具体的实现方法。

分解时间

如果日程规划是以“天”为单位，甚至以“周”为单位，我就认定它是没用的。你必须将日程规划先分解成一些非常小的任务，这些任务能够在以“小时”为单位的时间段中完成。不能有任何任务所需的时间超过16小时。

这实际上迫使你想清楚自己要干什么。写一个取名为foo的子例程，做一个对话框，解析一个名字为Fizzbott的文件……这一个个独立的开发任务所需要的时间很容易估计，因为你以前就写过子例程，做过对话框，解析过文件。

如果你马马虎虎，没有仔细想过你要做什么，没有在细节上想清楚详细的工作步骤，就在日程规划上为一个大型任务（比如“完成Ajax照片编辑器”）分配了3个星期，那么你其实不知道它究竟要花费多少时间，因为你没有仔

① 工作时间记录单（time sheet）是一种用来记录每项工作开始时间和结束时间的方法，参见http://en.wikipedia.org/wiki/Timesheet。

程序员最可信赖的求职帮手

细想过你要做什么。

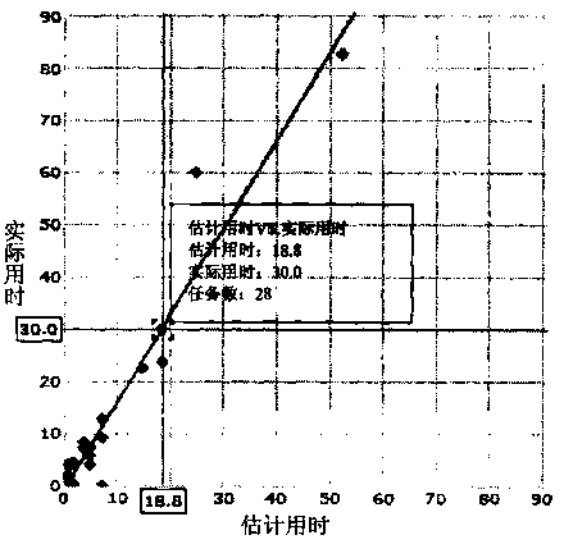
将每个任务所需的时间控制在16小时之内，这就会迫使你好好地去规划你要完成的那个任务。如果你只是很笼统地定下3个星期，用来完成一个复合的大功能，比如“Ajax照片编辑器”，而没有认真地规划细节，那么很遗憾，我不是帮你进一步细分日程的那个人，这个活命中注定是要你自己来干的。在此之前，你从来没有想过怎样一步步地做出这个功能，现在一时之间让你说出来，你肯定会忘掉很多东西。

追踪时间的用途

很难非常准确地估计完成单个任务所需要的时间。你怎么可能预测到会发生什么意外的事情呢？各种莫名其妙的错误，一场接一场的项目讨论会，Windows每半年就找你一次麻烦，让你不得不在工作主机上重装一次系统，这些事情你怎么想得到呢？退一步说，就算这些事情都不存在，你又怎么才能准确估计完成一个特定的子例程所需要的时间呢？

你不能，真的。

所以，请保留好那些工作时间记录单吧。让你用在每项任务上的时间长度留下可以追踪的痕迹。以后，你就可以回过头来参考这些数据，估计一下新的任务相对需要的时间。对于每一个程序员，你收集到的数据就像下图这样。



图中的每一个点代表一项已经完成的任务，横轴代表完成任务的估计用时，竖轴代表完成任务的实际用时。当你用估计用时去除以实际用时，就会得到完成速率。经过一段时间的累积，你就收集到了一系列数据，代表了每一个程序员完成速率的历史数据集合。

- 传说中完美的估计者，只存在于你的想像中，这种人的估计用时总是等于实际用时。所以，他们完成速率的历史数据就是{1, 1, 1, 1, 1, …}。
- 一个典型的糟糕估计者，他的完成速率的历史数据散布在图中的各个角落，比如就像这样{0.1, 0.5, 1.7, 0.2, 1.2, 0.9, 13.0}。
- 大多数估计者都属于估计不足，但是相对值是稳定的。每一次都是实际用时超过估计用时，因为估计的时候没有考虑到修正错误、小组会议、咖啡时间等的用时，也没有考虑到疯狂的老板一直会来打断你的工作。通常来说，这样的估计者的完成速率是比较稳定的，总是小于1.0，就像这样{0.6, 0.5, 0.6, 0.6, 0.5, 0.6, 0.7, 0.6}。

当估计者逐渐地变得富有经验，他们的估计精度就会提高。所以这时候，陈旧的历史数据，比如6个月前的数据，就可以丢弃不用了。

如果你的开发团队中加入了一个新成员，你没有他的历史数据，你就假设他的估计值是最不准的，你用一些假设的数字让他的完成速率起伏很大。等到他完成了六七件真实的任务以后，他的数据就会逐步变得准确了。

对未来情况进行模拟

有了各种任务完成时间的估计值以后，你要做的并不是将这些估计值简单地加起来，得到一个项目的最后完成日期。这样做听起来貌似正确，但是最终的结果却错得离谱。你真正应该使用的是蒙特卡洛方法[①]（Monte Carlo method），用它来模拟未来各种可能的结果。在蒙特卡洛模拟中，你可以假

① 蒙特卡洛方法是一种基于“随机数”的计算方法，即先确定一个随机事件，然后重复进行多次的随机实验，对结果进行统计平均，以随机事件出现的概率作为问题的近似解。这种方法源于美国在第二次世界大战中研制原子弹的“曼哈顿计划”。该计划的主持人之一、数学家冯·诺伊曼用驰名世界的赌城——摩纳哥的Monte Carlo——来命名这种方法。

设未来存在100种可能发生的情况，每一种情况发生的可能性是1%，这样你就能画出一张图，里面是项目在任意一个日期完成的概率分布。

为了计算任意一名程序员在未来任意一种情况下的实际完成时间，你就用每个任务的估计用时去除以这个程序员历史数据中完成速率的一个随机选择的值。这些历史数据是你在上面第(2)步中得到的。下面就是对未来一种情况的模拟：

估计用时：	4	8	2	8	16	
随机选取的完成速率：	0.6	0.5	0.6	0.6	0.5	总计：
期望用时：	6.7	16	3.3	13.3	32	71.3

对100种可能发生的情况，你都重复做一下上面的计算。每种情况发生的可能性是1%，接下来你就能估计出在任意一个日期完成项目的概率。

现在来看看得到了怎样的结果。

- 先看传说中完美的估计者，这种人的所有完成速率都是1。估计用时除以等于1的完成速率，不会发生任何变化。因此，每一轮的模拟都会得到同样的完成日期，而且这个日期的概率等于100%。就跟童话一样！
- 糟糕估计者的完成速率分布在图上各个角落，从0.1到13.0都有可能。每一轮的模拟都会产生一个非常不一样的结果，因为当你用估计用时去除以随机选择的完成速率时，每次的结果肯定差别非常大。你得到的概率分布曲线会非常平坦，向你显示项目在下一个工作日完成，或者在遥远的未来完成，两者的可能性是相等的。顺便说一下，你还是可以从中得到一点有用的信息，那就是它告诉你不可以对预测出来的完成日期有信心。
- 普通估计者的完成速率大多数都非常接近，比如就像这样：{0.6, 0.5, 0.6, 0.6, 0.5, 0.6, 0.7, 0.6}。当你用估计用时去除以这一组完成速率时，得到的期望用时都会比估计用时大出一块。比如你的估计用时是8小时，在一次模拟中实际可能需要13小时，在另一次模拟中则需要15小时。这是对估计者根深蒂固的乐观主义的一种补偿。这种补偿的程度非常准确，因为它是准确基于程序员一贯的乐观主义的，已经从历史数据中得到证明。因为所有根据历史数据得到的完成速率都非常接近，都在0.6附近，所以你每完成一轮模拟，你的数字都差不

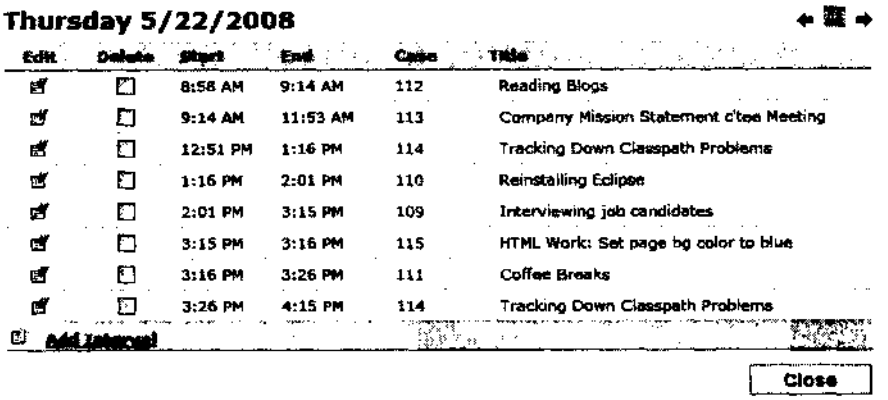
多，最后你就得到了一个非常窄的项目完成日期分布区间。

当然，在每一轮的蒙特卡洛模拟中，你必须将以小时为单位的数据转化成具体的日历上的日期。这意味着你必须考虑到每一个程序员的工作日程、他们的休息日和假期等。此外，你还必须考虑每一轮中哪一个程序员是最后完工的，因为那是整个开发小组的最后完成时间。这些计算很繁琐，但是幸运的是，解决繁琐的计算正是计算机拿手的本领。

不要有强迫症

如果你的老板时不时就来骚扰你一下，同你大谈他的又长又曲折的钓鱼故事，你该怎么办？如果你不得不参加根本与你无关的销售会议，你该怎么办？喝咖啡的茶歇时间呢？用在帮助新手搭建开发环境的半天时间呢？

Brett和我设计这个内部系统的时候，我们非常担心，有些无法预料的事情会占据大量的时间。有时候，这些事情加起来比写代码所花的时间还要长。你是不是也应该对这些事情做出预判，在工作时间记录单上一一记录下来？



对，如果你想做，你肯定能做到这一点。循证式的日程规划一样能够发挥作用。

但是，你其实不用这么做。

实践表明，循证式的日程规划非常可靠，不管是什么任务，你只需要按照自己的安排去做就行了，不用管会发生什么干扰。虽然这么说听上去很不

靠谱，但是只要你这样做，循证式的日程规划就会产生最佳结果。

让我向你展示一个简单的例子。为了让这个例子通俗易懂，我假定有一个叫做约翰的程序员，他的一举一动都非常有规律。他的工作就是写只有一行代码的赋值函数和取值函数，那些糟糕的编程语言就会要求你这么做。这就是一整天他干的所有事：

```
private int width;
public int getWidth () { return width; }
public void setWidth (int _width) { width = _width; }
```

我知道，我知道……这个例子未免举得太笨了，但是你很清楚你就是遇到过这种人。

不扯远了。约翰写每个赋值函数或取值函数要用两小时。所以，他对自己的任务完成时间是这样估计的：

{2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, …}

现在，这个可怜的家伙碰到了一个多事的老板，每过一会儿，老板就同他大谈一通如何钓马林鱼，要谈上两个小时。这个时候，约翰当然可以在日程上加上一项任务，就叫做“痛苦的马林鱼谈话”，把这个也记录在工作时间记录单上。但是，从公司内部关系的角度判断，这样做未必明智。其实，约翰有另外的办法，他完全不用改变日程。也就是说，他实际的任务用时就像下面这样：

{2, 2, 2, 2, 4, 2, 2, 2, 2, 4, 2, …}

他的完成速率就相应为：

{1, 1, 1, 1, 0.5, 1, 1, 1, 1, 0.5, 1, …}

这样一来，请思考会发生什么。在蒙特卡洛模拟中，估计用时被0.5除的概率不多不少正是老板打断约翰工作的概率。所以，循证式日程规划就生成了一个正确的日程规划！

事实上，哪怕是事无巨细、一切都做好记录的程序员，在估计各种工作干扰的置信区间上，准确度都远远不如循证式日程规划。这一点正是循证式日程规划的效果如此之好的原因。下面就是我向别人做的解释。当程序员的工作被打断的时候，他们有两种选择。

(1) 对此深感不满，将这些干扰写入工作时间记录单，反映在完成任务的预估时间中。这样一来，管理层就会看到谈论钓鱼这件事浪费了多少时间。

(2) 对此深感不满，但是拒绝将它写入工作时间记录单中，宁愿让手头正在从事的任务无法按照日程规划完成。因为你预估的时间非常正确，只是由于那些愚蠢的不请自来的钓鱼类谈话才让任务无法按期完成，所以你拒绝更改自己的时间估计。

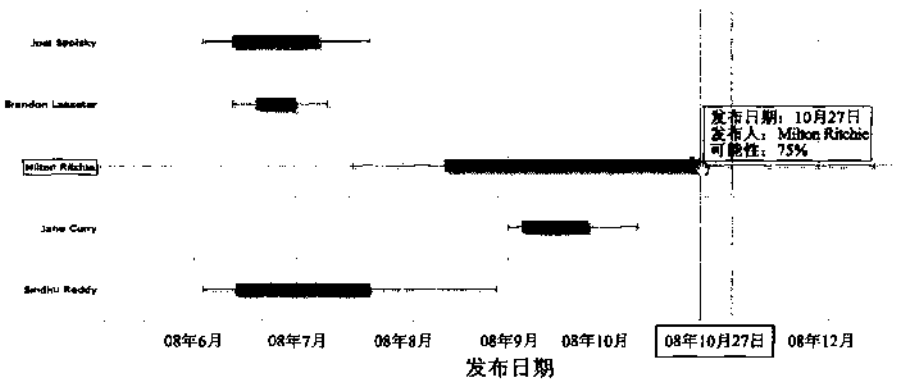
不管是上面哪一种选择，循证式日程规划都会生成同样的、完全正确的结果，与你的程序员的行事风格是主动型或被动型没有关系。

积极管理你的项目

一旦你将这个系统搭建起来，你就能够积极地管理你的项目，让它按时完成。比如，你可以将要实现的功能划分为不同的优先级，这样就可以很容易看出砍掉那些低优先级的功能会让完成日期提前多少。

Priority	50% Date
1 - Must Fix	5/30/2008
2 - Must Fix	6/30/2008
3 - Must Fix	10/2/2008
4 - Fix If Time	10/10/2008
5 - Fix If Time	10/19/2008
6 - Fix If Time	12/4/2008
7 - Don't Fix	3/1/2009

你还能看到每一个程序员的完成日期的概率分布，如下图所示。



一些程序员（比如上图中的Milton）可能会出问题，因为他们的完成日期太不确定了。他们需要好好学习如何改进估计水平。另一些程序员（比如

程序员最可信赖的求职帮手

Jane）对完成日期的估计非常准确，但是完成得太晚，他们需要减去一些任务。还有一些程序员（就是我啦）对项目如期完成影响不大，这类人就让他们去吧。

范围渐变[1]

假定你将所有的细枝末节都规划好了，你开始动手干活了，循证式日程规划就像预期的那样发挥了作用。这个时候，你突然有了新的想法，或者你的销售人员向客户兜售了某种你还没有开发出来的功能，再或者董事会的某个成员想出一个很酷的新点子，决定要在高尔夫球场中的电瓶车上安装GPS（全球卫星定位设备），以便在心脏病患者玩高尔夫球时外界能够知道他们在球场中的实时位置。所有这些事情都会导致项目的完成日期出现延迟，当你在做原始规划时，它们都是无法被预测到的。

在理想的情况下，你在规划的时候会为这些事情留出一段缓冲期。事实上，在原始规划时，你应该为以下的情况留出缓冲。

(1) 新的功能设想。

(2) 对手的新动向。

(3) 整合中出现的问题（也就是将大家的代码合在一起的时候出现的问题）。

(4) 解决各种错误。

(5) 易用性测试（要将这些测试的结果合并进最终产品）。

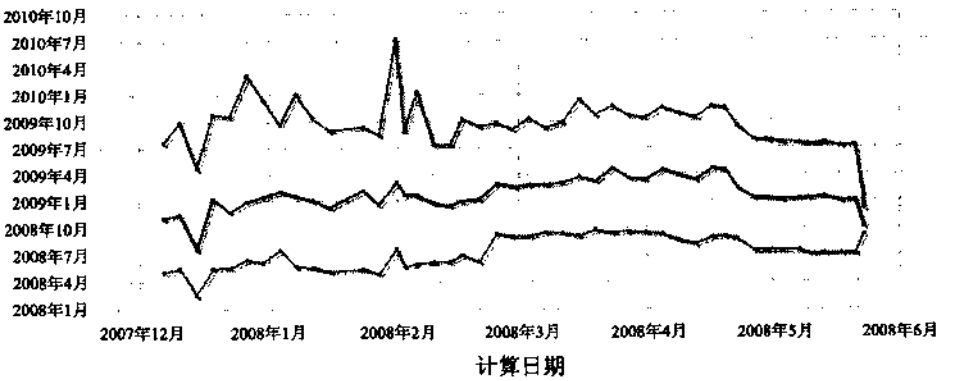
(6) 软件Beta版的测试。

有了这样的安排，一旦你需要实现新的功能，你就把缓冲时间用掉一块，用于开发新功能。

要是你不断地有新的功能需要开发，缓冲时间都用完了，那该怎么办？那样的话，你从循证式日程规划中得到的完成日期就不准确了，会不断地延后。你就要每天晚上记录下完成日期的概率分布，以便进行追踪。

---

① 范围渐变（scope creep）是项目管理中的一个专用名词，指的是在开发过程中涉及以前没有考虑到的问题，导致项目的范围越来越大。



横轴表示进行计算的日期，竖轴表示估计的项目完成日期。上图中一共有3条曲线，最上方的曲线代表95%的可能性，中间的曲线代表50%的可能性，最下方的曲线代表5%的可能性。因此，如果这3条曲线互相靠得越近，那么项目完成日期的可能性区间就越狭窄。

如果你看到曲线不断向上升，这就表示完成日期一再推后，你有麻烦了。如果每过一天，曲线都在往上升，这就表示你加进新任务的速度快于你完成原有任务的速度，你就永远做不完这个项目了。另一方面，你还可以从中看出完成日期的置信区间是不是变窄了（也就是3条曲线是不是逐渐汇合了），如果你真会在某一个日期完成一个项目，那么图形就是这个样子的。

我们的做法

我在多年的日程规划实践中还得出了另外几条经验。

(1) **只有第一线的程序员才能提出完成日期的估计值。**任何由管理层制定然后交给程序员去执行的日程规划，都注定会失败。只有真正负责开发的程序员才能估计出完成任务需要哪些步骤。

(2) **一发现错误就立即修正，将用时算入原始任务的用时之中。**你不能事先规划一个独立的错误修正任务，因为你不知道会出现什么样的错误。当新代码出现错误以后，就将解决问题的时间算进你原先写出错误代码的那段用时之中。这有助于循证式日程规划预计出得到可靠代码所需的时间，而不是得到可以运行的初步代码所需的时间。

(3) **防止管理层向程序员施加压力，要求加快开发速度。**许多没有经验

的软件项目经理认为他们可以“激励”手下的程序员，为程序员制定更好、更“紧张”的工作日程（时间短得不现实），达到更快地完成项目的目的。我认为这种“激励”方式是不动脑子的。如果我的进度落后于日程，我就感到惶惶不可终日，非常沮丧，这时再大的激励也是没用的。如果我的进度快于日程，我就感到欢欣鼓舞，工作效率非常高，这时根本不需要激励。日程规划并不是一个玩心理游戏的地方。

为什么管理者会想这样做呢？

当一个项目启动以后，负责技术问题的项目经理就会四处拜会产业界的人士，列出一张所需要的功能清单，并且估计3个月就可以完成。实际上，真正完成需要12个月。如果你写代码的时候没有想好要采取的所有步骤，那么假定起先估计的用时为n，最后在现实中花掉的时间总是可能会超过$4n$。如果你要做一个真正的日程规划，就需要将所有的任务加总起来，意识到这个项目的用时可能会比原先预料的长得多。产业界的人士就会感到不快。有一些愚蠢的项目经理为了解决这个问题，就想让手下的人加快干活速度。这实际上不太可能做到。你也许可以多雇几个人，但是新雇员需要一段时间才能赶上别人的速度，开头的几个月他们的效率可能只有别人的50%（他们还降低了指导他们的人的效率）。

如果你让手下的团队整个一年都超负荷工作，最终写出来的源代码可能会比正常数量暂时性地多出10%。这并不是很大的收益，而且有点类似于你在吃种子粮。当然，这还没有包括其他因素，比如你用来修改错误的时间翻了一倍、后面的项目也受到影响等。这真是冥冥中自有安排啊。

但是不管怎样，你的实际用时就是无法少于$4n$。如果你觉得你可以做到，请发电子邮件告诉我贵公司的证券代号，这样我就能卖空你们公司的股票了。

(4) 一份日程规划就是一个装满木块的盒子。如果你有一堆木块，你无法将它们装入盒子，你就有两个选择，要么找一个大一点的盒子，要么拿掉几个木块。如果日程规划上写着12个月，但是你希望项目在6个月里完成，那么结果就是你的项目出现延迟，或者你不得不删去了几个功能。你就是无法让木块变小，如果你假装相信自己能够做到，那么你自己就放弃了一个掌握未来的好机会，因为你对自己撒谎，自欺欺人地说未来就是我设想的这个样子。

既然我提到了，那么我就接着说，有效的日程规划有许多很大的好处，

其中之一就是你会被迫删去一些功能。这为什么是好事？

假定你想实现两个功能。其中一个非常有用，会使你的产品变成真正的优秀产品。另一个很容易实现，程序员迫不及待地想把它写出来（“快看！别眨眼睛！”），但是这个功能并非很有用。

如果你不搞一个日程规划，程序员就会首先将容易的/有趣的功能做出来。然后，他们剩下的时间就不够了，你别无选择，只好推迟日程来开发有用的或重要的功能。

如果你确实搞了一个日程规划，那么甚至在你开始工作之前，你就会意识到你必须砍掉一些东西。因此，你砍掉了容易的或是有趣的功能，全部精力投入开发有用的或重要的功能。正是这种迫使你砍掉某些功能的压力，使得你最终做出了一个更强大、更优秀的产品，它包括了更好的功能组合，而且能够在较早的日期完成。

回想很久以前，我还在Excel 5开发团队的时候，我们最初的功能清单十分庞大，完成日期远远落后于日程规划。“啊！老天！”我们心想，“这些全部都是超级重要的功能！如果没有一个宏编辑向导（macro editing wizard），我们还怎么活呀？”

最后，事情很明显，我们没有第二条路，只能把许多功能都砍掉，砍到不能再砍的地步，“只剩下了骨架”，这样才能如期完工。所有人都为这件事感到非常不开心。为了让大家感觉好受一点，我们安慰自己说，被砍掉的功能并不是被抛弃了，而是仅仅被推迟到Excel 6中实现了。

当Excel 5的开发工作接近尾声的时候，我和同事Eric Michelman开始着手编写Excel 6的设计规格说明书。我们坐下来，详细审阅从Excel 5的日程规划中被刷下来、准备放进Excel 6的功能清单。猜猜结果怎样？这份功能清单比你能想到的最糟糕的清单还要糟糕，上面没有一个功能是值得开发的。我想它们之中的每一个功能都从来没有过开发价值。为了赶上日程，我们砍掉了这些功能，现在看起来这是我们做过的最有价值的一件事情。如果我们当时没有这样做，那么Excel 5的开发时间会延长一倍，然后做出来的产品中包含了50%无用的垃圾功能，并且未来我们还不得不维护这些功能，直到Excel生命的最后一天，都要让当前版本向后兼容这些功能。

总　结

使用循证式日程规划是非常容易的，它只需要你在每个项目开始的时候花一到两天的时间做一个详细的规划，然后每天花几十秒在工作时间记录单上记录下你何时开始完成一项新任务。你的收益是非常巨大的，那就是你将得到一份有效的日程规划。

有效的日程规划是创造优秀软件的钥匙。它强迫你首先完成最重要的功能，让你做出正确的选择，思考要开发一个怎样的软件。这会使你的产品变得更出色，使你的老板感到更高兴，使你的客户感到更满意，以及最重要的一点，那就是使你下午5点能够准时下班。

附　言

循证式日程规划的功能包括在FogBugz 6.0中。

21

关于战略问题的通信之六[1]

2007年9月18日，星期二

IBM公司最近发布了一套开源Office软件，叫做IBM Lotus Symphony，看上去大概是根据StarOffice[2]修改的。但是，我怀疑起这个名字的真正目的，可能是想清除人们对最早那套Lotus Symphony[3]的记忆。那套软件在发布之前，被吹得天花乱坠，简直就像耶稣要复活一样，但是在发布之后一败涂地。它就是软件业中的Gigli[4]。

20世纪80年代后期，Lotus公司努力地想找到下一步的方向，升级他们的旗舰产品——电子表格和作图软件Lotus 1-2-3。他们很自然地就想到了两条路。第一条路是往软件中加入更多的功能，比如文字处理功能。这就是Symphony这个产品的由来。第二条看上去很显然的路是做一个3D电子表格软件。这就是后来Lotus 1-2-3的3.0版本。

这两条路一开始就遇到了一个大麻烦：在古老的DOS环境中，内存占用不得超过640KB。那个时候，IBM已经开始少量出售配备80286芯片的个人电脑，这种新的芯片能够提供更多的内存。但是，Lotus公司觉得，为这种售价高达10 000美元的电脑开发专用软件市场不会很大。所以，他们一个字节、一个字节地压缩，花了18个月才将Lotus 1-2-3的新版本塞进了640KB的内存

程序员最可信赖的求职帮手

① 作者网志上有一系列的战略问题阐述，本文是其中的第6篇。——编者注

② StarOffice是Sun公司发布的一套Office软件，它的源代码在2000年7月开源，成为了后来的OpenOffice的基础。

③ Lotus Symphony是Lotus公司在1984年发布的一套Office软件，在DOS环境下运行。1995年，IBM公司以35亿美元的价格并购了Lotus公司。

④ Gigli是一部2003年上映的美国电影，主演中包括Ben Affleck、Jennifer Lopez、Al Pacino等大明星。由于Ben Affleck和Jennifer Lopez在拍摄过程中爆出绯闻，这部电影在上映之前被大肆宣传，但是上映之后，口碑极差，被认为是有史以来最滥的电影之一。

中。但是，最终在白白浪费了许多时间之后，他们不得不放弃了3D功能，因为内存不够了。Symphony的遭遇也差不多，许多功能都被砍得干干净净。

这两条路都走错了。当Lotus 1-2-3的3.0版本上市的时候，每个人家中都已经有了一台80386芯片的电脑，配备了2MB或者4MB的内存。至于Symphony，它的电子表格功能很弱，文字处理功能也很弱，所有其他功能都很弱。

“够了，老同志，”你们会说，“谁如今还关心那些老掉牙的、只能在字符模式下运行的软件？”

请暂且忍耐我一分钟，因为历史正在以3种不同的方式重演。那么，最聪明的应对策略就是押宝在同样的结局上面。

低速CPU和小容量内存的环境

从最早的时候一直到大概1999年，程序员都极其关注软件的效率问题。在这段时期中，内存空间总是不够大，CPU主频也不够高。

到了20世纪90年代后期，一些像微软和苹果这样的公司开始注意到摩尔定律[①]（其实它们只比别的公司早了一点点）。它们认识到，不必太在意软件的效率问题和内存占用……只要把很酷的功能做出来，然后等着硬件升级就可以了。微软公司首次发布Excel的Windows版本的时候，80386电脑还非常贵，实际上没什么人买得起，但是微软公司很有耐心。只过了几年，80386SX[②]出来了，兼容机的价格下降到1500美元，你只要买一台，就能运行Excel。

由于内存的价格直线下降，CPU的速度每年都在翻番，所以作为一个程序员，你就面临选择。你可以花6个月用汇编语言重写程序的内循环（inner loop）。你也可以休假6个月，去一支摇滚乐队当鼓手。不管是哪一种选择，6个月后你的程序都会运行得更快。实际情况是没有程序员喜欢用汇编语言编程。

---

① 摩尔定律（Moore's Law）是指大约每隔18个月，芯片的晶体管容量要比先前增加一倍，同时性能也提升一倍，而价格下降一半。这个定律描述了硬件的发展趋势，由Intel公司的共同创始人Gordon E. Moore在1965年提出。

② 80386SX是80386芯片的一个低价版，后者在1986年上市，前者在1988年上市。

所以，我们都不怎么关心软件的效率或优化问题。

不过有一个例外，那就是在浏览器的Ajax应用程序中使用的JavaScript语言。因为这是当前几乎所有的软件开发工作的方向，所以这是一个重大的问题。

眼下的许多Ajax应用程序有一百万行甚至更多的客户端代码。现在的瓶颈已经从内存和CPU，转移到了带宽和编译时间。你必须想尽办法进行优化，这才能使复杂的Ajax程序有良好的表现。

但是，历史正在重演。带宽正在变得越来越便宜，即使这样，还是有人在考虑如何对JavaScript进行预编译（precompile）。

有一些程序员将大量的精力投入优化工作，要将程序变得更紧凑、更快速。某一天，他们醒来后将发现，这种努力基本上是白忙一场。如果你喜欢用经济学家的口吻夸夸其谈，那么你至少可以说，这种努力“不会带来长期的竞争优势”。

从长远的观点来看，那些不关心效率、不关心程序是否臃肿、一个劲往软件中加入高级功能的程序员最终将拥有更好的产品。

跨平台的编程语言

程序员最可信赖的求职帮手

C语言的原始设计目标就是为了让编写跨平台的应用程序[illegible]它很好地实现了这个目标，但是并不是100%跨平台。所以，后来又出现了[illegible]Java，它的通用性甚至要超过C语言。历史大概就是这样啦。

眼下，在跨平台这出连续剧中，正出现又一个高潮，那就是——没错，你猜对了——客户端JavaScript的兼容性问题，尤其是浏览器DOM（文档对象模型）的兼容性问题。编写一个网络应用程序，让它在所有不同种类的浏览器上都能运行，这简直是一场可怕的噩梦。你根本找不到其他方法，只能精疲力竭地在Firefox、IE 6、IE 7、Safari和Opera上一一测试，猜猜发生了什么事？我没有时间在Opera上测试，所以只好不管Opera了。这意味着，新兴的互联网浏览器根本不会获得立足的机会。

未来会怎样？当然，你可以在心里企盼或祈求，微软公司和Firefox能够制作出更具备兼容性的产品。祝你好运。不过，你还有另外一个选择，就是使用p-code虚拟机[①]或者Java虚拟机模型，你在底层系统之上建立一个小小的沙箱（sandbox），再将软件的运行建立在沙箱之上。这样做的不利之处就是，沙箱有很多缺陷，它们非常慢而且错误百出，这就是Java applet[②]都死光光的原因。建立一个沙箱，你就等于走上了一条不归路，你能得到的运行速度只有底层系统的1/10，你也无法利用任何一个只有某个底层系统支持、而其他底层系统都不支持的特性。（直到今天，我都在翘首期待，有人能向我展示可以在智能手机上使用的Java applet。它能利用手机的所有功能，比如拍照、读取地址本、发送短消息、与全球卫星定位系统互动等。）

沙箱在过去行不通，在将来也不会行得通。

那么，未来会怎样？获胜的一方所采取的策略正是贝尔实验室在1978年做出的决定，那里的科学家决定开发一种跨平台的、高效的编程语言，这就是后来的C语言。这种语言可以将程序编译成不同平台、不同系统可以理解的“本地”码（各种不同的JavaScript和DOM就是本地码）。至于怎么编译，那是编译器作者需要解决的问题，与你无关。代码编译后的运行效果与“本地的”JavaScript直接运行完全一样，能够以一种统一的方式获取DOM模型的全部潜力，能够自动地和跨平台地与IE和Firefox的核心代码融合在一起。对的，它还完美地支持CSS，能够以一种令人惊骇、但是事实证明是正确的方法让你玩转CSS，所以你永远都不必为CSS的不兼容问题发愁。再也不会这样了，永远不会了。哦，等这一天到来的时候，该是多么美好啊。

完善的互动性和用户界面标准

IBM 360大型机（mainframe）使用一种叫做CICS[③]的用户界面，你今天在飞机场还能看到这个系统，你只要在办理登机手续的柜台上弯下身子就能

① P-code是软件编译过程中产生的一种中间代码，不同于最终的机器码，可以使得编程语言不依赖于特定的平台或硬件。

② Java applet是用Java语言编写的、镶嵌在网页的小应用程序。它需要计算机安装了Java虚拟机以后才能运行。

③ CICS是Client Information Control System（用户信息控制系统）的缩写。

看到。这种界面是80字符宽、24字符高的绿色屏幕，只有字符模式，没有图形界面，这是肯定的。主机发送一个表单给“客户端”（一台IBM 3270智能显示终端）。这个“客户端”是智能的，它知道如何将表单呈现给你，允许你将数据输入表单，在这个过程中，根本不与主机通信。这就是IBM大型机如此强大、远远超过UNIX系统的原因之一，因为CPU根本不需要处理你的行编辑，这种任务由智能终端承担了。（如果你做不到为每个人都配置一台智能终端，那么你就去买一台System/1小型机，充当主机和哑终端[①]之间的中介，为你承担表单编辑的任务。）

不管怎样，只要你填完了表单，按下“发送”键，你输入的所有数据就被送回服务器端处理。然后，服务器端又给你发来一个新的表单。整个过程周而复始。

一切都很棒。但是，如果想在这种环境下使用文字处理软件，你该怎么办？（你无法如愿了。在大型机上从来都没有过一个像样的文字处理软件。）

这就是历史上的第一阶段。它与互联网时代的HTML阶段正好对应。HTML语言就是带有字体变化的CICS。

等到历史进入第二阶段，所有人都在写字桌上配备了PC。于是，突然之间，也不管程序员本人愿不愿意，他就是具有了在任意时间、任意场合随意操弄屏幕上任意角落的任意文字的能力。实际上，你可以获取用户打字时的每一次击键，因此你就能做出一个很好很快的应用程序，不必等到用户按下“发送”键，CPU就能提前介入，做出相应的处理。比如说，你可以开发一个文字处理软件，一旦当前行快要写满了，软件就会自动换行，将结尾的最后一个词移到下一行的行首。一切都在瞬间完成。哦，我的老天，你能做到这一点？

第二阶段也有自己的问题，那就是缺乏一个明确的用户界面标准……程序员具备了空前强大的决定权，几乎可以随意按照自己的偏好来制作软件，因此每个人都用不同的方式写软件，这就给用户带来了困扰，如果你会用X软件，这并不代表你就会使用Y软件。WordPerfect[②]和Lotus 1-2-3有截然不同

① 哑终端（dumb terminal）就是连接主机而不做任何计算处理的终端机。

② WordPerfect是Coral公司拥有的文字处理软件，在20世纪80年代末和90年代初流行一时，是事实上的文字处理软件标准。后来，被微软公司的Word取代。

的菜单设计、截然不同的键盘接口和截然不同的指令结构。根本没有可能在程序之间复制数据。

这也正是我们今天在Ajax开发中面对的局面。当然，不可否认，Ajax应用程序的易用性比第一代DOS应用程序有了很大的提高。因为从那时开始，我们已经学到了不少经验。但是Ajax应用程序没有规范的标准，如果想要协同工作，就会非常麻烦。你完全没有办法将对象从一个Ajax应用程序中剪切和粘贴到另一个中。举例来说，我就不太确定，你怎样才能将Gmail中的图片传到Flickr中。拜托，老兄，剪切和粘贴在25年前就发明出来了。

在历史上的第三个阶段中，出现了配备Macintosh操作系统和Windows操作系统的PC。一个统一的、标准的用户界面诞生了，包括多窗口和剪贴板这样的标准功能，这使得在多个程序间进行协同工作成为可能。这种崭新的GUI（图形用户界面）带给我们易用性和实用性的飞跃，导致了个人电脑爆炸式增长。

因此，如果历史会重演，我们就可以期待总有一天，Ajax程序的用户界面会出现某种程度的统一，它的诞生方式就如同Windows的诞生方式一样。总有人会写出一个具备压倒性优势的SDK（软件开发工具包），其他人就可以用它来开发功能强大的Ajax应用程序。不同的程序员使用同样的用户界面组件，使得开发出来的程序可以协同工作。那种赢得最多程序员认可的SDK就具备了垄断性的竞争优势，堪称可与微软用Windows API获得的竞争优势媲美。

如果你是一个互联网开发者，你不想用别人都在用的主流SDK，那么你将发现没有用户使用你的程序。原因其实你知道得很清楚，那就是你的程序不支持剪切和粘贴，无法进行地址本同步，也可能不具备其他所有在2010年流行的新奇的互动功能。

比如，请想象一下，假定你是Google公司的负责人，你为自家有Gmail这样的产品感到沾沾自喜。但是没过多久，某家你从来没有听到过的公司（很可能是一家桀骜不驯的初创公司，背后有Y Combinator[①]的资助）开发出了一种NewSDK，销售状况好得难以置信。这种NewSDK使用一种性能优异的

① Y Combinator是一家创业投资公司，专门为创业者提供种子资金。该公司由Paul Graham等人在2005年创立。

跨平台编程语言，可以直接编译生成JavaScript，而且更出色的是，它还配备了一个大型Ajax库，能够执行所有种类的智能性的互动功能。不仅仅是剪切和粘贴，还有一些很酷的聚合（mashup）功能，就像同步和单点身份管理（single-point identity management）。有了单点身份管理，用户就不必将自己正在干什么告诉Facebook和Twitter了，只需要在网上任意一个支持这个功能的地方输入就可以了。你对这一套NewSDK嗤之以鼻，因为它的大小居然高达惊人的232MB！232MB啊！编译生成的JavaScript，单单载入一个页面就需要76秒。所以你认定自家的应用程序Gmail不会流失任何用户。

但是就是从那时起，就当你在Google总部里，坐在Google式座椅上，细细品味Google味咖啡，感到洋洋得意、沾沾自喜、高枕无忧、踌躇满志的同时，新版本的浏览器发布了，支持缓存编译后的JavaScript。于是，突然之间，NewSDK的载入速度变得很快。Paul Graham又及时地向这家初创公司补充了6000包方便面，让他们饿的时候有东西吃。这样一来，这家公司又可以继续运营三年，将产品不断完善。

你手下的程序员，不管是张三还是李四，都有相同的看法，那就是Gmail太庞大了，无法移植到那个呆呆的NewSDK上面去。那样的话，我们就必须改变每一行的代码。这接近于完全重写整个程序，太可怕了。整个系统模型会一团混乱，充满了嵌套。NewSDK使用的跨平台编程语言用到的括号多得连Google也无法承受。几乎每一个函数的最后一行都是一个包含连续3296个右括号的字符串。你因此不得不去买一个特殊的编辑器，否则无法数清到底有多少个右括号。

后面的事情是，NewSDK的工程师又发布了一个相当不错的文本处理软件以及一个相当不错的电子邮件应用程序，还有一个杀手级的Facebook/Twitter式的事件发表器，能够将网上与你有关的所有事情都进行同步。人们开始使用他们的产品。

就在你不知不觉之间，所有人都开始编写基于NewSDK的应用程序。这些程序的表现非常好，一转眼，产业界点名只想用基于NewSDK的应用程序。所有老式的纯Ajax应用程序看上去都变得很寒酸，它们做不到剪切和粘贴，不能够聚合和同步，互相之间无法很好地协同工作。Gmail就这样成了遗迹，好比Email程序中的WordPerfect。未来的某一天，你对孩子们说，曾几何时当你得到2GB的空间储存Email时，你是多么激动。孩子们全都嘲笑你，他

们的指甲油都不止2克[①]。

你是不是觉得这个故事太荒诞不经了？那你就将“Google Gmail”替换成“Lotus 1-2-3”。NewSDK将是微软公司Windows传奇的重现。整个过程完全是Lotus公司如何丢失电子表格市场的重演。在互联网上，这种事情将再发生一次，因为现在所有影响市场的因素和背后的动力同当年完全一样。我们唯一不知道的就是，它到底发生在何时、何地、何人身上，但是它一定会发生。

① 在英语中，mail（邮件）和nail（指甲）发音相近。这里的意思是，孩子们将2G空间的电子邮件听成2克（2g）的指甲油，因此嘲笑说他们的指甲油都不止2克。

22

你的编程语言做得到吗

2006年8月1日，星期二

某天，你正在查看自己写的代码。你注意到有两大块代码几乎完全相同。实际上，它们就是相同的，唯一的不同是，一块与“意大利面”（Spaghetti）有关，另一块与“巧克力慕思”（Chocolate Mousse）有关。

```
// 一个小例子：

alert("I'd like some Spaghetti!");
alert("I'd like some Chocolate Mousse!");
```

上面的例子是用JavaScript写的，即使你不懂这种语言，你也应该能理解我的话。

这种重复的代码当然看上去很不顺眼，所以你创造了一个函数：

```
function SwedishChef( food )
{
    alert("I'd like some " + food + "!");
}

SwedishChef("Spaghetti");
SwedishChef("Chocolate Mousse");
```

好，这是一个很小的例子，但是你不难从中想到更现实的情况。用函数取代重复的代码是更好的做法的理由太多了，类似的话你听到过100万次了。什么更容易维护啊，可读性更好啊，代码更容易重复利用啊，总之就是好啦！

现在，你又注意到，还有两块代码看起来几乎相同。唯一的区别是，一块代码调用了BoomBoom函数，另一块调用了PutInPot函数，除了这一点，这

些代码完全一样。

```
alert("get the lobster");
PutInPot("lobster");
PutInPot("water");

alert("get the chicken");
BoomBoom("chicken");
BoomBoom("coconut");
```

于是，你需要找到一种方法，这种方法能够将函数作为一个参数传入另一个函数中。这种功能很重要，因为这样一来，你将重复性的代码封装在一个函数体中的可能性就变大了。

```
function Cook( i1, i2, f )
{
    alert("get the " + i1);
    f(i1);
    f(i2);
}

Cook( "lobster", "water", PutInPot );
Cook( "chicken", "coconut", BoomBoom );
```

看到了吗，我们将一个函数作为了参数。

你用的编程语言做得到吗？

等一等……假定你还没有来得及定义函数PutInPot或者函数BoomBoom。那么将函数定义直接作为参数不是比单独在其他地方再写一个函数更好吗？

```
Cook( "lobster",
      "water",
      function(x) { alert("pot " + x); } );
Cook( "chicken",
      "coconut",
      function(x) { alert("boom " + x); } );
```

哇，这可真是方便。请注意，我在调用处实时创造了一个函数，甚至都不用费事地为它起名字。我只要提着它的耳朵把代码甩进函数就可以了。

一旦你开始意识到可以将匿名函数（anonymous function）作为参数，

你也许就会发现这种方法有普遍意义。比如，需要对数组的每一个元素进行某种操作。

```
var a = [1,2,3];

for (i=0; i<a.length; i++)
{
    a[i] = a[i] * 2;
}

for (i=0; i<a.length; i++)
{
    alert(a[i]);
}
```

遍历数组元素是一种很常见的操作。你可以写一个函数，让它来做这件事。

```
function map(fn, a)
{
    for (i = 0; i < a.length; i++)
    {
        a[i] = fn(a[i]);
    }
}
```

这样你就可以将上面的代码改写为：

```
map( function(x){return x*2;}, a );
map( alert, a );
```

另一种常见的操作是将数组的所有值以某种方式进行累加。

```
function sum(a)
{
    var s = 0;
    for (i = 0; i < a.length; i++)
        s += a[i];
    return s;
}

function join(a)
{
    var s = "";
    for (i = 0; i < a.length; i++)
        s += a[i];
    return s;
```

```
}

alert(sum([1,2,3]));
alert(join(["a","b","c"]));
```

函数sum和函数join看起来很像，你可能想着要把它们抽象一下，把具有共性的代码写成一个通用函数，将数组的元素结合成一个单一的值：

```
function reduce(fn, a, init)
{
    var s = init;
    for (i = 0; i < a.length; i++)
        s = fn( s, a[i] );
    return s;
}

function sum(a)
{
    return reduce( function(a, b){ return a + b; }, a, 0 );
}

function join(a)
{
    return reduce( function(a, b){ return a + b; }, a, "" );
}
```


许多陈旧的编程语言根本没有办法实现这样的功能。还有一些编程语言虽然可以做到，但是做起来很麻烦（比如，C语言确实有函数指针，但是你必须在其他地方先声明和定义函数）。至于面向对象的编程语言，它们根本不觉得应该把操作函数的所有权力都交给你。

如果你要像处理第一类对象[①]（first class object）那样处理一个函数，Java语言要求你创造一个只有一个方法的对象（叫做functor）。除此以外，许多面向对象的编程语言，要求你为每一个类创造一个文件。这两个因素加起来对运行速度有巨大的拖累。如果你用的编程语言要用到functor，你就不会得到现代编程环境所带来的全部好处。你最好去查一下，看看是不是能把这种编程语言退货，让卖家还给你一部分钱。

那么，一个这样微小的函数，只能对数组中的元素进行遍历，到底能给你带来多大好处？

① 第一类对象指可以不受限制使用的对象，得名于比喻“像第一等公民那样”。

让我们回过头再来看map函数。当你需要对数组中每一个元素依次进行处理时，那么实际情况很可能是，到底按照哪一种次序进行遍历是无关紧要的。你可以从头到尾进行遍历，也可以从尾到头进行遍历，都会得到同样的结果，对不对？事实上，如果你正好有两个CPU可以用，也许你可以写一些代码，使得每个CPU遍历一半的元素，于是在一瞬间map函数的运行速度快了一倍。

让我们进一步设想，你一共有几十万台服务器，分布在世界各地的几个机房中（只是假定）。然后，你还有一个超级庞大的数组，包含了互联网上的所有数据（再次重申，这只是假定）。于是，你可以在这些服务器上运行map函数，每台服务器负责处理很小的一部分运算。

那么，举例来说，你写出了一些运行速度很快的代码，用来搜索整个互联网的内容。这件事就变得很简单了，简单到你只要调用map函数并将最基本的字符串搜索器（searcher）作为参数就可以了。

我在这里想提醒你注意一件有趣的事情：如果你将map函数和reduce函数看作任何人都可以调用并且确实每个人都在用的函数，那么，你只要找到一个超级天才，让他写出能够在全世界庞大的服务器阵列上分布式运行的map函数和reduce函数，所有你以前在单机上能够进行循环操作（loop）的代码现在就依然能够使用，并且速度提高了几十亿倍，这意味着同样这些代码现在可以在很短的时间内处理完巨型的问题。

让我再重复一次。你可以将“循环操作”这个单独的步骤抽象出来，用任何你想用的方法完成这个步骤，其中就包括用额外的硬件将运算过程轻松地扩展（scale）到多台服务器上。

不久前，我写过一篇文章，抱怨有些计算机系只教给学生Java语言，其他什么也不教[①]。现在，你应该明白我的意思了：

> 如果你不懂函数式编程，你就无法创造出MapReduce，正是这种算法使得Google的可扩展性（scalable）达到如此巨大的规模。术语“Map”（映射）和“Reduce”（化简）分别来自Lisp语言和函数式编程。回想起来，在类似6.001这样的编程课程中，都提到

① 请参见第8章。

纯粹的函数式编程没有副作用，因此可以直接用于并行计算。任何人只要还记得这些内容，那么MapReduce对他来说就是显而易见的。发明MapReduce的公司是谷歌，而不是微软，这个简单的事实说出了原因，为什么微软至今还在追赶，还在试图提供最基本的搜索服务，而谷歌已经转向了下一个阶段，开发Skynet，我的意思是，开发世界上最大的并行式超级计算机。我觉得，微软并没有完全明白在这一波竞争中它落后了多远。

好的，我希望到了这个时候，你已经深信不疑了。你确信，那些具备了“第一类函数”功能的编程语言，能够让你更容易地完成进一步抽象代码的任务。这意味着你的代码体积更小、更紧凑、更容易重复利用、更方便扩展。谷歌的许多应用程序都使用了MapReduce技术，所以一旦有人对底层的并行计算程序进行优化或消除bug，那么所有这些应用程序都会受益。

说到这里，我有点动感情了，忍不住想站起来说，最有生产效率的编程环境是那些允许你在不同层次上进行抽象的编程环境。老掉牙的FORTRAN语言完全不让你写函数，C语言倒是有函数指针，但是它们太太太太太丑陋了，根本无法实现匿名函数，你不得不在使用之前先在其他地方将它们写好。Java语言允许你使用functor，但是这个功能的实现方式甚至比C语言更糟糕。正如Steve Yegge所指出的，Java语言是“名词王国”（steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html）。

更正 我上一次使用FORTRAN语言还是27年以前的事情。很显然，那个时候已经可以在它里面写函数了。我一定是把它和GW-BASIC语言[①]搞混了。

---

① GW-BASIC是微软公司开发的BASIC语言的一种分支，随MS-DOS操作系统一起发售。在MS-DOS 5.0发布后，逐渐被QBASIC取代。

23

让错误的代码显而易见

2005年5月11日，星期三

回到遥远的1983年9月，我有了第一份真正的工作，在一家叫做Oranim的以色列大型面包厂里干活，每个晚上使用6个像航空母舰那样大的巨无霸烤箱生产大概10万个面包。

我第一次走进面包厂的时候，那里的脏乱让我简直无法相信自己的眼睛。烤箱颜色发黄，机器上锈迹斑斑，到处都是油腻腻的。

“这里原来就这么乱吗？”我问。

“什么？你在说什么？”经理说，“我们刚刚大扫除过，这是几星期以来最干净的一天。”

哦，老天啊。

有两个月的时间，每天早上开工前，我都要把烤箱清洁一遍。直到有一天，我终于弄懂了他们嘴里的“干净”是什么意思。在面包厂里，所谓“干净”就是指机器表面不能沾上面粉团，垃圾中不能有发酵后的面粉，地板上也不能有面粉团。

把烤箱内部擦得亮堂堂并不属于“干净”的范围。烤箱大概每十年清洁一次，而不是每天清洁一次。油脂也不属于“干净”的范围。事实上，许多机器需要定期加油，或用油脂擦拭。一层薄薄的油脂层往往表明机器最近刚刚被清洁过。

你只有身处这个行业，才会知道面包厂里“干净”的全部意思。对于外人来说，一走进面包厂，就想做出是否干净的判断，那是不可能的。一个外

来者永远不会想到去看一看面粉搓圆机（dough rounder，一种将面粉块搓成团的机器）的内壁是否被刮干净了。他只会抓住一个事实不放，那就是旧烤箱的外壳已经斑驳脱色。但是，这些烤箱如此巨大，有一些地方脱色难道不是很正常吗？一个面包师傅根本不关心他的烤箱外壳上是否有一些油漆开始泛黄了。因为烤出来的面包不会受到影响，还是一样美味。

在面包厂里待了两个月以后，你就会懂得如何看待“干净”了。

程序的代码也是一样的。

当你还是编程菜鸟的时候，或者当你阅读用一门新语言写成的代码的时候，程序的源码简直就像天书一样。只有等到你理解了这门语言，你才会看出什么地方有明显的语法错误。

在学习的第一阶段，你开始想要找出那种我们通常称作“代码风格”（coding style）的东西。因此，你注意到有些代码的书写风格不统一，不遵守缩进规则，大写字母的使用方式很奇特。

程序员最可信赖的求职帮手

在这个时候，你通常的反应是：“真见鬼，我们必须有统一的代码风格。”第二天，你花了一整天的时间为整个编程团队写出了一份“代码风格规范”（coding convention）。接下来的6天，你一直在同其他人争论是否要采用“单括号结尾法[①]”。然后的3个星期，你都在修改旧的代码，想确保它们都符合“单括号结尾法”，直到有一个经理终于忍不住了，把你逮住，责备你不应该将时间都浪费在那些不能够赚钱的事情上。于是，你决定暂且就做到这里，以后看到的时候再当场修改，这样也不是不可以接受的。所以，你手里的代码只有一半改成了“单括号结尾法”。没多久，你就把这件事忘得一干二净，满脑子想的都是其他与赚钱不相干的事情，比如将一个字符串类（string class）换成另一个字符串类。

随着你对于在某种特定环境中的编程越来越熟练，你开始有能力注意到其他东西了。那些东西百分之百符合语法规则，也完美地符合“代码风格规范”，但是就是让你担心不已。

---

① 单括号结尾法（One True Brace Style）是一种关于代码缩进的编程风格，最早出Kernighan和Ritchie在《C程序设计语言》一书中使用。现在的UNIX内核和Linux内核源码使用的都是这种编程风格。它的主要特征就是表示开始的括号不需要单独的行，与前面的代码在同一行中，表示结束的括号则需要单独的行，两个括号之间的代码有相应的缩进。

举例来说，在C语言中，

```
char* dest, src;
```

这行代码是合法的代码。它可能完全符合你的“代码风格规范”，甚至可能是故意写成这样。但是，如果你有足够的C语言代码的编写经验，你会发现它将dest声明为一个字符指针（char pointer），而将src仅仅声明为一个字符。这可能是你想要达到的结果，也可能不是，但是不管怎样，这种代码总是有点让人感到不顺眼。

甚至还有更隐蔽的例子：

```
if (i != 0)
    foo(i);
```

在这个例子中，代码是100%正确的，也符合大多数代码书写规范，真的是没有一点错。但是，不可否认，条件语句if后面跟的是一个单语句的语句体，它没有使用大括号包住。这一点可能会让你感到不舒服，因为你的脑袋里有个声音在说，不行，有人可能一不小心会在这里插入一行语句：

```
if (i != 0)
    bar(i);
    foo(i);
```

他也忘了在外面加上大括号，导致foo(i)在不经意间成了不经过条件判断、直接运行的语句！正是因为有这种担心，所以每当你看到代码块没有用大括号包住时，你就会多多少少感到那么小小的一丝不爽，仿佛代码的纯洁性遭到了一点玷污，这让你不舒服。

好了，到目前为止，我已经提到了一个程序员的3种境界。

(1) 你分不清什么是干净的代码，什么是不干净的代码。

(2) 你对干净的代码有一个肤浅的认识，主要看它们是不是符合代码书写规范。

(3) 你开始能找出那些很隐秘的不干净代码，在干干净净的表面之下发现蛛丝马迹。但是，发现和修正不干净代码要花去你很长的时间，把你折磨得够呛。

在3种境界之外，还存在更高的第4种境界，这也是我下面要详细谈到的内容。

(4) 你精心构建代码，发挥洞察力，将它们写得清晰易懂，不容易出错。

这是一门真正的艺术，即通过人为发明一套书写规范，使得错误在显示屏上变得显而易见，从而让代码更健壮。

接下来，我要带你看一个小例子，再告诉你一条万能的规则，让你学会用它制定自己的健壮代码书写规范。最后，我将为某种“匈牙利命名法[①]”进行辩护（虽然可能不是让人们头晕的那种），并且将批评某些条件下的异常（exception）处理（虽然可能不是你大多数情况下会经历的那种条件）。

但是，如果你深信不疑，坚信匈牙利命名法**不是好东西**，而异常处理则是除了巧克力冰淇淋以外的最佳发明，同时你也不想听到任何反对意见，那么你大可不必待在这里。继续读下去的话，你可能也不会有任何收获，我推荐一组优秀的漫画（www.neopoleon.com/home/blogs/neo/archive/2005/04/29/15699.aspx），你可以去那里看看。事实上，一分钟以后，我就要开始举例讲解真实的代码了，你很可能一看到它们就会睡着，连感到不爽的机会也没有。对啊，我想这就是我的计划，先把你哄得快睡着了，然后再将“匈牙利命名法=好东西”、“异常处理=坏东西”灌输进你的大脑，反正你睡意朦胧，没有心思同我大辩一场了。

一个例子

不说废话了，在下面这个例子中，我们假定你正在开发一种互联网应用程序。这样假定没有特别的意思，只是因为这阵子小朋友们都流行写这种玩意。

现在有一种叫做“跨站点脚本攻击”（Cross-Site Scripting Vulnerability）的安全漏洞，又称为XSS。我在这里不详细解释了，所有你需要知道的就是，当开发一个互联网应用程序时，你一定要小心，绝不能信任用户通过表单发送的任何字符串，绝不能直接就拿来使用。

举例来说，你做了一个网页，上面写着“你叫什么名字？”，下面是一

① 匈牙利命名法（Hungarian notation）是一种对变量的命名方法，在变量名中显示其数据类型。在23.3节还有对它的详细介绍。

个输入框，用户填写后递交数据，就会自动进入另一个网页，里面写着“你好，Elmer!”（假定用户输入的名字是Elmer）。这里就有一个脚本攻击的漏洞，因为用户输入的可能不是“Elmer”，而是各种奇形怪状的HTML和JavaScript代码。这些代码会做一些非常龌龊的事情，比如读出你存放的cookie信息，将它们发往一些带有邪恶目的的网站。更糟糕的是，这些龌龊的事情看上去就好像是你干的。

我们可以用假想的代码，来演示这个过程。假设

```
s = Request("name")
```

可以从HTML表单中读出用户的输入（POST方法的一个参数）。如果你像下面这样写代码：

```
Write "Hello, " & Request("name")
```

那么你的网站就已经有可以被XSS利用的漏洞了。只需要这样一句代码就够了。

正确的做法是，在将用户输入的内容再输出前，你必须进行编码。所谓“编码”，就是把"替换成&quot、把>替换成&gt，等等。所以真正安全的写法应该像下面那样：

```
Write "Hello, " & Encode(Request("name"))
```

所有用户输入的字符串都是不安全的。任何不安全的字符串都必须经过编码才能输出。

让我们尝试制定一种代码书写规范，确保当编程错误时，代码会看上去就是错的。如果代码看上去是有错的，那么它至少有相当大的机会被作者或检查者发现。

可能的解决方案#1

第一种方案是当用户输入的数据一传进来，就立刻对所有的字符串进行编码：

```
s = Encode(Request("name"))
```

所以，我们的书写规范就是，如果你看到Request外面没有被Encode包围，那么这行代码就是错的。

下一步，你就开始训练自己能够敏锐地发现单独出现的Request，因为这违反了书写规范。

这种方案是可行的，只要你遵守这种书写规范，你的代码就不可能有XSS漏洞，但它却不一定是最佳方案。比如，要是你想将用户输入的字符串储存在某个数据库中，那么就应该直接储存，而不是储存编码后的HTML代码。编码后再储存是没有意义的，因为数据可能被用于非HTML代码的场合。就像在一个信用卡处理程序中，如果输入的是编码后的HTML代码，进一步处理就非常麻烦。大多数的互联网应用程序在开发的时候遵循的原则就是内部使用的所有字符串都是非编码的，直到最后一刻，要把数据发往HTML网页的时候，再进行编码处理。这样的做法可能才是正确的架构。

我们确实有必要将字符串以不安全的格式保存一段时间，不用急于编码。

那么，我再来试试看。

可能的解决方案#2

要是我们制定另一种书写规范，规定只有输出字符串时才进行编码，那会怎么样？

```
s = Request("name")

// 隔开很多行
Write Encode(s)
```

那样的话，你一看到一个孤零零的Write，在它后面没有Encode，你就知道出问题了。

可是，这也不是太管用……有时候，在你的代码中会有一些HTML的小片段，你是不能对它们进行编码的：

```
If mode = "linebreak" Then prefix = "<br>"
// 隔开很多行
Write prefix
```

根据我们的书写规范，像上面这样写就意味着有错。书写规范要求我们在输出字符串时必须进行编码：

```
Write Encode(prefix)
```

但是，"<br>"的意思是换行，一编码就变成了
，完全以字面的形式
输出到了用户端，就失去了换行的作用。所以，这样的做法也不对。

因此，在一些时候，读取的字符串不可以编码，在另一些时候，输出的字符串不可以编码。这意味着我们前两个方案都行不通。如果找不到一套有效的书写规范，我们就仍然有可能遇到下面这样的风险：

```
s = Request("name")
...好几页之后...
name = s
...好几页之后...
recordset("name") = name // 把名字储存在数据库的姓名栏中
...好几天之后...
theName = recordset("name")
...好几页或者好几个月之后...
Write theName
```

我们还会记得要对字符串进行编码吗？任何一句单独的语句都看不出问题。如果你有大量像这样的代码，你就根本无处入手进行查找，不得不追踪每一个输出的字符串的来源，确保它是被编码过的，这种工作量简直能把人压死。

正确的解决方案

好了，让我来公布一个真正可行的书写规范。我们只需要遵守以下规则就可以了。

> 所有来自用户的字符串都必须储存在变量（或数据库）中，而且变量名必须以“us”（unsafe string，不安全的字符串）起首。另一方面，所有已经经过HTML编码的字符串，或者已知来源安全的字符串，都储存在变量名以“s”（safe string，安全的字符串）起首的变量中。

现在我来重写上面的代码，使它符合我们新的书写规范，除了变量名以外，其他都不用改变。

```
us = Request("name")
...好几页之后...
usName = us
...好几页之后...
recordset("usName") = usName
```

程序员最可信赖的求职帮手

```
...好几天之后...
sName = Encode(recordset("usName"))
...好几页或者好几个月之后...
Write sName
```

请注意，有了新的书写规范之后，如果你不小心在不安全的字符串上面犯了错误，你总是可以在同一行代码中发现这个错误，前提是你一定要遵守书写规范：

```
s = Request("name")
```

上面这行代码，可以被推定为错误的。因为你发现Request的结果被赋予一个变量名以“s”起首的变量，这是违反书写规范的。Request的结果永远都是不安全，所以必须被赋予一个变量名以“us”起首的变量。

```
us = Request("name")
```

肯定是正确的。

```
usName = us
```


肯定是正确的。

```
sName = us
```

肯定是错误的。

```
sName = Encode(us)
```

肯定是正确的。

```
Write usName
```

肯定是错误的。

```
Write sName
```

肯定是正确的，等同于下面的代码

```
Write Encode(usName)
```

每一行代码都可以通过自身判断正误。如果每一行代码都是正确的，那么整个的代码块也是正确的。

渐渐地，有了这个书写规范，你能够一眼发现Write usXXX并且马上可以判断出这行代码错了。你也会马上反应过来，知道应该如何改正。我承认，一开始的时候，发现错误的代码不是很容易，但是做上3个月，你的眼睛就

会产生自动识别能力，正如有经验的烤面包工人只要对一个大型面包厂看上一眼，就会立刻说："老天啊，里面根本没人打扫过！这算哪门子干净的面包厂，你们到底在干什么？"

事实上，我们可以将规则再延伸一点，重命名（或封装）Request和Encode函数为UsRequest和SEncode……就是说，如果一个函数的返回值是不安全的字符串，那么函数名就以Us起首，反之就以S起首，同变量名的命名规则一致。那么，再来看代码：

```
us = UsRequest("name")
usName = us
recordset("usName") = usName
sName = SEncode(recordset("usName"))
Write sName
```

看出我为什么要这么干了吗？现在，你只要看看等号两边是否以同样的前缀起首，就能知道是否出错了。

```
us = UsRequest("name")      // 很好，两边都是Us
s = UsRequest("name")       // 出错
usName = us                 // 正确
sName = us                  // 一定出错
sName = SEncode(us)         // 一定正确
```

再来，我还可以再加以改进，将Write重命名为WriteS，将SEncode重命名为SFromUs：

```
us = UsRequest("name")
usName = us
recordset("usName") = usName
sName = SFromUs(recordset("usName"))
WriteS sName
```

这样就使得错误更明显。你的眼睛将学会"看出"可疑的代码，这有助于你在正常的代码书写和阅读过程中就发现隐藏的安全漏洞。

让错误的代码显而易见是一种很好的实践，但未必是所有安全问题的最佳解决方案。总有一些漏洞或错误会漏网，因为你可能做不到每一行都检查一遍。但是，有这种机制比起没有这种机制，绝对要好得多得多。我强烈赞成制定代码书写规范，最起码这会让错误的代码更容易被发现。你立刻就能从中受益，每一次程序员的眼睛扫过一行代码，就能检查和防止某些特定的错误。

程序员最可信赖的求职帮手

一条通用规则

让错误的代码能够被一眼看出，这种做法有一个前提，即让[illegible]显示屏上必须紧挨在一起。为了得到正确的代码，每看到一个字符串，我就需要知道它是否安全。如果我所需要的信息包含在另一个文件里，或者相隔了好几个页面，我不得不去拖动滚动条，那么它们对我就是毫无帮助的。我必须当场就得到信息，这样才能立即做出判断，这就意味着要有一套为变量起名的规范。

有大量例子可以说明，只要你把相关内容放在一起，就能改善代码的质量。大多数编程规范都有类似的规则。

- 尽量将函数写得简短。
- 变量声明的位置离使用的位置越近越好。
- 不要使用宏（macro）去创建你自己的编程语言。
- 不要使用goto。
- 不要让右括号与对应的左括号之间的距离超过一个显示屏。

所有这些规则都有一个共同点，就是让与一行代码相关的所有信息尽可能地靠拢，缩短它们之间的物理距离。这将大大增加你的机会，让你一眼就能看出程序内部是怎么回事。

总的来说，我必须承认，对于那些有隐含内容的语言特性，我总是有点害怕。当你看到下面这行代码：

```
i = j * 5;
```

如果这是C语言，那么你至少知道代码的意思是j乘以5，再将结果保存在i中。

但是，如果你是在C++语言中看到这行代码，你就得不到任何结论。你什么都无法知道。在C++语言中，想要知道到底发生了什么事，唯一的方法就是去确认i和j的数据类型，而这两个变量很可能是在完全不一样的地方声明的。之所以要去确认数据类型，原因是j的类型可能定义了乘法的运算符

重载[①]（operator overloading），在乘法运算时，自动进行某些附加的非常巧妙的处理。另一方面，i的类型可能定义了等号的运算符重载。此外，i和j很可能分别属于两个不兼容的数据类型，所以进行赋值运算时会自动发生类型强制转换（type coercion），相应的函数会被调用。理清这一切的唯一方法，不仅包括搞清楚变量的类型，还包括搞清楚实现这些类型的代码。要是碰巧某个地方用到了继承（inheritance），那么你只好祈祷上帝来帮你了，因为现在你需要将所有类的树形结构自己从头到尾整理一遍，才能真正明白代码是怎么运行的。如果程序中还用到了多态[②]，你就有大麻烦了，因为单单知道变量声明中i和j的类型已经不够了，你还必须知道程序运行的每一个阶段i和j的类型，这涉及你需要检查的代码数量真是不知道有多少。由于还存在停机问题[③]（halting problem），你就永远无法百分之百确定看完了所有地方（哎呀呀呀呀呀！）。

要是你在C++语言中看到i=j*5，老兄，你真的是孤立无援。所以在我看来，这样写代码就降低了你通过肉眼发现问题的能力。

当然，你可以说这样写代码不算是什么大问题。当你自以为很巧妙，使用像运算符重载这样学术味十足的东西时，你的目的肯定是想用它建立一个完备的抽象层（waterproof abstraction）。你看到j是一个Unicode的字符串，不由得眼前一亮，一个Unicode的字符串乘以一个整数，不是正好可以定义一个乘法的运算符重载吗？这样就能方便地完成字符串转换，比如将繁体中文转成简体中文，对不对？

可惜的是，你觉得这是一个完备的抽象层，但是事实上它不是。我在*The Law of Leaky Abstractions*（收录在*Joel on Software*一书中，Apress于2004年出版）一文中，已经深入地探讨过这个问题了。这里我就不重复了。

Scott Meyers从各个角度向你展示了，至少在C++中，为什么这样做不会

① 运算符重载是面向对象编程中特有的概念，属于多态（polymorphism）的一种。一旦使用了运算符重载，部分或所有的运算符就有了完全不同的定义，可以依据运算子（argument）的不同而进行不同的运算。C语言不支持运算符重载，而C++支持。

② 多态是面向对象编程语言的一种语言特性，指的是A类型在行为上与另一种B类型相同，这通常表明A类型继承了B类型，或者A类型部署了B类型的接口。

③ 停机问题属于计算机理论中的一个逻辑难题，大意是无法找到一个程序，可以用来判断任意程序（函数）在有限输入的情况下是否会在有限时间内结束运行。1936年，阿兰·图灵证明了这一点。

取得好的效果，反而会对你造成伤害。他用这个主题都开创出了一番事业。（顺便说一句，他的*Effective C++*第3版已经出版了，所以你应该立刻去找一本来读！）

差不多了。

书归正传，现在让我们来总结一下，到目前为止这篇文章主要的观点。

> 寻找一种代码的书写规范，让错误的代码变得容易被看出。让代码中的相关信息在显示屏上集中在一起，使你能够当场发现和改正某些种类的错误。

匈牙利命名法

现在，我们要来回过头来讲声名狼藉的匈牙利命名法了。

匈牙利命名法的发明人是微软公司的程序员Charles Simonyi。他在微软的主要项目之一就是Word。事实上，他领导了这个项目，创造出世界上第一个“所见即所得”（WYSIWYG，What You See Is What You Get）的文字处理软件。这种成就即使在施乐公司的PARC研究中心[①]也会得到大声喝彩。

在“所见即所得”的文字处理软件中，程序的界面窗口是可以滚动的，所以每个坐标既可以解释成相对于当前窗口的坐标，也可以解释成相对于整个文档页的坐标。这两者有很大的差别，所以对此进行妥善安排是很重要的。

我猜想，这就是Charles Simonyi发明自己的变量命名法的主要原因之一。该变量命名法后来就被称为匈牙利命名法，因为用它命名的变量看上去就像匈牙利文一样，而Simonyi又来自于匈牙利。在Simonyi版的匈牙利命名法中，每个变量的起首是一个小写字母组成的标签，表明变量中包含了什么种类（kind）的数据。

---

① PARC的全称是Palo Alto Research Center（帕洛阿尔托研究中心），它是美国施乐公司于20世纪60年代末在硅谷设立的中央研究所，各大学、公司的研究人员可以在此自由进行研究。它取得了一些意义极其重大的研究成果，包括发明个人计算机、以太网、操作系统的图形界面、鼠标等，堪称是信息技术领域历史上最重要最伟大的一家研究机构。

我故意用了“种类”（kind）这个词，因为Simonyi在他的原始文档中错误地使用了“类型”（type）这个词，导致好几代程序员都误解了他的意思。

如果你仔细地阅读Simonyi写的原始文档，他的意思同我前面举的那个例子中的命名法是一样的。在那个例子中，变量名的前缀us代表“不安全的字符串”，s代表“安全字符串”。这两者都属于字符串类型。如果你将其中一个变量的值赋给另一个，编译器是没有办法帮你区分的，“智能感知技术[①]”（IntelliSense）也不会给你任何提示。但是，这两个变量在语义上确实是不一样的，它们需要区别对待，在赋值时，需要调用某种类型转换函数，否则你就应该得到一个运行故障（runtime bug）。当然，前提是你有那么幸运的话。

Simonyi的匈牙利命名法的原型在微软公司内部最初被叫做“应用型匈牙利命名法”（Apps Hungarian），因为它是在“应用程序部”（Applications Division）中使用的，也就是用在Word和Excel身上。在Excel的源码中，你可以看到大量的rw和col。如果你看到它们，你就知道它们指的是行（row）和列（column）。是的，它们都是整数，但是它们之间的赋值运算毫无意义。我听说，在Word的源码中，你能看到大量的xl和xw，前者代表“相对于页面的横坐标”（horizontal coordinates relative to the layout），后者代表“相对于窗口的横坐标”（horizontal coordinates relative to the window），它们都是整数，但是互相之间的赋值运算也是毫无意义的。在这两种应用程序中，你还会看到大量的cb，意思是“字节个数”（count of bytes），对，这也是整数，但是仅仅通过观察它的变量名你就会知道许多其他事情。它代表占用了多少字节，也就是占用内存中缓冲区的大小。一旦你在代码中看到xl = cb，那么就可以拉响空袭警报了，这显然是错误的代码。虽然xl和cb都是整数，可是将以像素为单位的水平位移赋值给字节的长度绝对是发疯。

在“应用型匈牙利命名法”中，前缀同时用于函数和变量。说实话，我从来没有看过Word的源码，但是我敢用美元同你的面包打赌[②]，源码中一定有一个叫做YlFromYw的函数，它的作用是将相对于窗口的纵坐标转换成相

① “智能感知技术”IntelliSense是微软公司开发的代码书写的辅助工具，具有自动完成功能，最常见的使用场合是在Visual Studio开发环境中。

② 原文是I'll bet you dollars to donuts，这是一句英语成语，字面意思是我以美元为赌注，你以面包为赌注，言下之意就是我百分之百肯定我会赢。

对于页面的纵坐标。“应用型匈牙利命名法”用的类型转换函数名字都是TypeFromType形式，而不是传统的TypeToType形式，目的是确保每个函数名都以返回值的类型作为前缀。这同我在前面的例子中的做法是一样的，当时我将编码函数的名字改成了SFromUs。事实上，在严格的“应用型匈牙利命名法”中，编码函数的名字根本没有其他选择，只能起名为SFromUs。在为这个函数起名的问题上，“应用型匈牙利命名法”没有留给你第二种选择。这其实是一件好事，因为你又可以少记住一样东西，而且你也不必看着Encode这个词，想不清这到底是哪一种Encode[①]。现在的函数名比原来的名字精确得多。

“应用型匈牙利命名法”的作用是极其巨大的，在C语言盛行的时代尤其如此，因为C语言的编译器几乎不提供类型识别。

但是，就在那个时候，却出了一些问题。

消极因素逐渐在匈牙利命名法中占据了上风。

似乎没人知道为什么这件事情会发生，或者这件事情到底是怎么发生的。但是，看上去Windows团队中的文档作者在不经意间就发明出了后来被叫做“系统型匈牙利命名法”（Systems Hungarian）的东西。

不知什么人在什么地方就读到了Simonyi的原始文档，Simonyi在其中用了“type”这个词，那个人以为Simonyi指的是数据类型，就像编程语言中数据类型的那种类型以及编译器所做的类型检查的那种类型。但是，Simonyi不是这个意思。他很详细、很准确地解释了他所说的“type”到底是什么意思。可惜于事无补，危害已经酿成了。

“应用型匈牙利命名法”的前缀是非常有用、非常有含义的，比如“ix”表示数组的索引值（index），“c”表示一个计数器（count），“d”表示两个数量之间的差额（difference）（例如“dx”就表示宽度），等等。

“系统型匈牙利命名法”的前缀就差远了，比如“l”表示长整型（long），“ul”表示无符号的长整型（unsigned long），“dw”表示双精度值（double word），这实际上也是一个无符号的长整型。在“系统型匈牙利命名法”

① 这句话指的是，前面的例子中编码函数最早的名字是Encode，后来改成SEncode，最后改成SFromUs。

中，名称的前缀能起到的唯一作用就是告诉你这个变量到底是哪一种数据类型。

这种差别很细微，但是完全误解了Simonyi的意图和做法。如果说这件事有什么教训，那就是如果你写出艰涩的学术性文章，没人能够理解，那么你的思想就会被误解，人们会把误解后的思想当作你原来的思想加以嘲笑，根本不管你原来的思想是什么样的。在“系统型匈牙利命名法”中，你会看到许多变量的名字类似dwFoo[①]，意思是“double word foo”（双精度变量）。看到这样的变量名时你简直就想骂街。告诉你一个变量是双精度值对你几乎没有任何用处。所以，人们抛弃“系统型匈牙利命名法”就毫不奇怪了。

“系统型匈牙利命名法”传播得又远又广，在Windows编程文档中，它是标准的变量命名法。还有许多书籍对它广为宣扬，比如Charles Petzold的*Programming Windows*（微软出版社于1998年出版）一书，该书堪称学习Windows编程的圣经。因此，它很快就成为匈牙利命名法的主导形式。即使在微软公司内部，除了Word和Excel开发团队，也只有很少人知道这样理解匈牙利命名法是错的。

接下来，大暴动（Great Rebellion）就发生了。那些从一开始就没有搞懂匈牙利命名法的程序员接触的都是那种被误解的形式，他们不断地发现这种命名法真是烦得要死，而且几乎完全没用，于是就决定反抗了。当然，“系统型匈牙利命名法”在发现代码错误方面也并非一无是处。最起码，如果使用这种命名法，你在使用变量的时候就能当场知道变量的数据类型。不过，“系统型匈牙利命名法”就是远远不如“应用型匈牙利命名法”有用。

大暴动在.NET第一次发布的时候达到了高峰。微软最终做出了表态，“不推荐使用匈牙利命名法”。那时真是欢声雷动啊。我根本不认为微软会花心思解释原因。那些人只是翻到文档中“命名指南”这一部分，然后在每一个条目中加上“不要使用匈牙利命名法”。那个时候，匈牙利命名法实在是太不受欢迎了，根本没有人会抱怨微软的这个举动。在这个世界上，除了Word和Excel开发团队，每个人都感到如释重负，终于可以不再使用这种蹩

① 在程序设计中，foo通常代表变量名的通配符，类似的词还有bar。在这里，dwFoo就代表dwXXX。

脚的命名法了。他们心想，只要编译器包含强制类型检查，只要有智能感知技术，就足够了。

但是，“应用型匈牙利命名法”仍然是极其有价值的，它加强了代码之间的联结，使得代码更容易阅读、书写、除错和维护。最重要的是，它让错误的代码容易现形。

在结束这个话题之前，我还要再做一件答应过的事，那就是再骂一次异常处理（exception）。我上一次开口骂的时候，惹来了一大堆麻烦。那是在Joel on Software的网站上，我写了一篇即兴评论，大意是我不喜欢异常处理，因为它本质上是一种无形的goto，我指出为什么它比有形的goto更糟糕。不出所料，几百万人跳出来反驳我。同样不出所料的是，全世界唯一一个站出来支持我的人是Raymond Chen。顺便说一句，他是世界上最好的程序员，所以必须要出来表态，对不对？

下面就本文所涉及的主题，我来谈谈异常处理。你的眼睛有一个学习过程，只要一看到代码，就会试着去找出错误，这能够防止程序出现问题。为了使代码真正强壮和可靠，当你查看代码的时候，你需要有一个好的代码书写规范，允许相关信息集中在一个地方。也就是说，出现在你眼前的关于代码行为的信息越多，你就能够越轻松简单地发现错误。假定你看到了下面的代码：

```
dosomething();
cleanup();
```

你的眼睛告诉你，看不出有什么问题啊？我们总是会做清理工作（clean up）的！不过，dosomething这个函数有可能抛出一个异常，这意味着cleanup函数就不会被调用。这个问题很容易解决，使用finally[①]或类似方法就可以了。我想说的不是怎么解决这个问题，而是我们没有办法知道cleanup函数是否一定会被调用，如果要搞清楚，唯一方法就是查看整个dosomething函数调用的树结构，看看是否某处有某些代码会抛出异常，这样就行了，此外还有像可控异常（checked exception）这样的方法可以使整个过程变得比较容易。但是，我的真正意思是异常处理使得代码关联（collocation）消失了。为了知道某一行代码是否运行正常，你不得不到其他地方去寻找答案，你因此就

① 在异常处理的标准语法中，finally关键字代表不管是否发生异常都要执行的语句。

不能利用人的眼睛一目了然的优势学着看出错误的代码，因为根本没有东西可看。

要是我只是写一个小小的脚本程序，每天收集一次数据，然后把它打印出来，那么异常处理真是好用得不得了。我的日子不可能过得再轻松了，我不需要关心任何可能发生的错误，只要把所有烦心的代码用一个大的try/catch包裹起来，如果发生错误，就向我发送Email，这样就一切OK了。异常处理对于那些随手写写的软件项目和脚本程序很好用，只要不是非常重要和关键性的代码，它都是适用的。但是，如果你写的是一个操作系统，或者核电厂的管理软件，或者一个专用于心脏手术的高速电锯的控制程序，那么异常处理是极端危险的。

我知道大家会认定，我的编程水平很低下，理解不了异常处理的真正含义，完全不知道如果全身心拥抱异常处理，我的生活质量就会出现全方位的提升。但是……我要对你们说，实在是太遗憾了。写出真正可靠代码的方法就是使用那些考虑了人类常见缺陷的简单工具，而不是使用那种含有隐藏的副作用、产生有瑕疵的抽象层、假设程序员绝不会出错的复杂工具。

推荐阅读

如果你还是对异常处理痴心不改、信心万丈，请阅读Raymond Chen的文章*Cleaner, More Elegant, and Harder to Recognize*（blogs.msdn.com/oldnewthing/archive/2005/01/14/352949.aspx）："异常处理用的是否正确，非常难以从代码中看出来。……异常处理的难度实在太大，我的智商不足以用好它们。"

Raymond Chen还写过另一篇文章*A Rant Against Flow Control Macros*（blogs.msdn.com/oldnewthing/archive/2005/01/06/347666.aspx），畅谈了滥用宏语言的危害。这是另一个很好的例子，再次证明了没有把相关信息都集中在同一个地方会导致代码无法维护。"当你看到那些使用[宏]的代码，你就必须钻进许多的头文件，搞清楚那些宏到底在干什么。"

如果对匈牙利命名法的历史渊源感兴趣，应该从Simonyi的原始文档*Hungarian Notation*（msdn.microsoft.com/en-us/library/aa260976(VS.60).aspx）看起。Doug Klunder把匈牙利命名法介绍给了Excel开发团队，他的文章

Hungarian Naming Conventions（www.byteshift.de/msg/hungariannotationdoug-klunder）相对清晰好懂一些。如果你想知道更多关于匈牙利命名法的故事以及它是如何被其他文档作者毁灭的，请阅读Larry Osterman的帖子（blogs.msdn.com/larryosterman/archive/2004/06/22/162629.aspxf），并且特别留意Scott Ludwig的评论（blogs.msdn.com/larryosterman/archive/2004/06/22/162629.aspx#163721），还可以阅读Rick Schaut的帖子（blogs.msdn.com/rick_schaut/archive/2004/02/14/73108.aspx）。

程序员最可信赖的求职帮手

第六部分

开办软件公司

24 *Eric Sink on the Business of Software*的前言

25 *Micro-ISV: From Vision to Reality*的前言

26 飙高音

24

*Eric Sink on the Business of Software*的前言

2006年4月7日，星期五

从Joel on Software这个网站开办的早期，Eric Sink就是常客了。他是Spyglass浏览器[1]的创造者之一，开发了开源文字处理软件AbiWord，现在是SourceGear公司的程序员，专门制造源代码控制软件。

但是，我们这里大部分的人之所以认识他，是因为他是网上讨论组The Business of Software的版主。这个讨论组已经成了软件创业者的聚集地。他发明了“软件个体户”（micro-ISV）这个名词，有好几年的时间，他一直在自己的网志中写关于软件业的话题，他还为MSDN写过一组非常有影响的系列文章。最近，他出版了一本内容丰富的纸质实体书，名字是*Eric Sink on the Business of Software*。他要求我写一篇序言，下面就是我的这篇序言。

我跟你们说过我的第一笔生意吗？

让我试试能不能把整件事回忆起来。记忆中，那是我14岁的时候。有人在新墨西哥大学（University of New Mexico）办了一个TESOL[2]暑期研究班（summer institute），他们雇我当复印员。我坐在一张写字桌后面，只要有人需要杂志上的文章，我就负责复印。

① 此处，Joel说得不准确，Spyglass实际上是一家公司，而不是浏览器。Spyglass公司是伊利诺伊大学控股的一家上市公司，它继承了Mosaic浏览器的代码和所有的权利。Eric Sink是Spyglass公司中浏览器开发团队的负责人。

② TESOL是Teaching of English to Speakers of Other Languages的缩写，意为“对英语非母语的人的英语教学”。

我的写字桌旁边有一个很大的咖啡壶。如果想喝咖啡，你就自己去倒一点，然后将2角5分的硬币扔在一个小杯子里。我自己不喝咖啡，但是我那时很喜欢吃甜面包圈。因此我想，好吃的甜面包圈应该很适合就着咖啡一起吃。

我那时还是小孩，年纪太小不能开车。在我能够步行到达的距离中，没有面包店，我被有效地同阿尔伯克基[①]这个城市中的甜面包圈隔离了。不知怎么地，我居然就说服了一个研究生同我合作，他每天早上从外面买几个甜面包圈带给我。我架起了一块手写的招牌，上面写着“甜面包圈：25美分一个（便宜！）”，然后，钱就这样赚到了。

每天经过这里的人们看到了这块小小的招牌，就在杯子里扔一些钱，然后自行取走一个甜面包圈。我们开始有了回头客。甜面包圈的销售量稳步上升。甚至那些不必经过休息室的人都特地改变每天的路线，就为了拿一个甜面包圈。

我当然有权决定分发免费的样品。说来奇怪，这样做竟然对利润几乎没有影响。甜面包圈的成本大概是1美元12个，可是有人却会支付1美元而只取走1个甜面包圈，仅仅因为他们懒得从放钱的杯子里一个个地数该拿走多少找头。我真地无法相信会有这种人！

到了暑假结束的时候，我每天的销售量是两大盘……大概100个甜面包圈。累积了一大笔钱……我不记得具体金额了，但是肯定是几百美元。要知道，那是1979年，这可是一大笔钱啊，足以把全世界的甜面包圈都买下来了。遗憾的是，那时我已经吃甜面包圈吃到有点恶心了，而开始宁愿去吃非常非常辣的奶油墨西哥馅饼了。

那么，我用这笔钱干了什么呢？回答是什么也没干，语言系的系主任把这笔钱都取走了，他用这笔钱开了一个大晚会。系里所有的员工都能参加，只有我不能去，因为我的年纪太小了。

这个故事的教训是什么？

嗯，没有教训啦。

我只是想说，当你亲身经历新生意的慢慢成长，你会感到一种难以置信

① 阿尔伯克基（Albuquerque）位于美国新墨西哥州中部，是该州最大的城市，也是新墨西哥州立大学所在地。

的激动。那是一种快乐，看着每一项健康的生意都像有机物那样地成长。我所说的“有机物”，字面意思是“与碳化合物相关的东西”。不过，等一等，这并不是我的意思。我真正的意思是像植物一样，一步步地成长。上个星期你赚了24美元，这个星期你赚了26美元。明年的这个时候，你可能赚到了100美元。

人们喜欢一项不断成长的生意，同喜欢园艺是一个道理。这是一种真正的乐趣，将一个小小的种子埋在地里，每天浇水、除草，眼看着一个微小的新芽长成枝繁叶茂、华美耐寒的大株菊花（如果你幸运的话），或者长成大荨麻（如果你不明白这种长得像野草一样的植物有什么用，请不要失去希望，你可以把荨麻用来做茶，只是要小心，别碰到它们）。

看着得到的收入，你会说：“哇，现在只是下午3点，我们已经有了9个顾客了！今天将是有史以来最美好的一天！”到了第二年，一天9个顾客听上去就像笑话一样。等再过了几年，你猛然发现内部报表显示，上周的销售额已经大到无法有效管理的地步。

有一天，你会关闭销售系统的Email通知功能，你再也不需要系统在每次有顾客购买软件时都通知你了。这是一个巨大的里程碑。

最后你会发现，在你雇用的暑期实习生中，有一个人每个星期五早上都带一些甜面包圈来把它们卖掉赚钱。我唯一的希望是，你不会拿走他的利润用来办一个晚会，却不邀请他参加。

25

*Micro-ISV: From Vision to Reality*的前言

2006年1月11日，星期三

下面的文章是我为Bob Walsh的新书*Micro-ISV: From Vision to Reality*写的前言。

我什么时候变成了“软件个体户”运动的招牌人物？

有这么多人，干嘛选我呀。受不了。

我创立Fog Creek软件公司的时候，根本没想过要当“个体户”。我的计划是创立一家大型跨国软件公司，在全球120个国家设有办事机构，在曼哈顿岛上有一幢摩天大楼当作总部，楼顶上还有直升飞机停机坪，可以快速前往长岛最东边的汉普顿。这一切也许要花上几十年才能实现，毕竟我们是白手起家、自力更生，作计划时总是非常保守和谨慎，但是我们的抱负一点也不小。

唉，我甚至不喜欢“软件个体户”这个词，它的原意是“微型独立软件供应商”（Micro independent software vendor），其中的“独立软件供应商”是微软公司发明的一个新词，指“微软公司以外的软件公司”，或者说得更明白一些，就是指“由于某种原因，我们还没来得及并购或消灭的软件公司，可能因为他们从事的都是一些花里胡哨的生意（比如布置婚礼仪式上的桌子），所以像我们这样高级的公司没必要自降身价去跟他们争这些小生意，就让他们逍遥自在去吧。但是你们这些小公司一定要记得使用.NET！”

与这个词在一起的还有另一个微软公司发明的词，叫做“历史遗留下来

的”（legacy），用来指所有那些不是微软公司开发的软件。比如谈到Google的时候，他们就说这是“历史遗留下来的搜索引擎”，好像就在暗示Google只不过是“一家古老、蹩脚的搜索引擎，之所以还有人使用，完全是因为历史原因，用户迟早会不可避免地投向MSN的怀抱。” 反正就是诸如此类的意思。

我宁愿老老实实使用“软件公司”这个词，创办一家小企业并不是见不得人的事。“软件创业公司”，这就是我们称呼自己的名字，我看不出有任何必要要从微软公司的角度来定义我们自己。

我猜想你会读这本书，是因为你也想开一家小型的软件公司。这是一本这方面的好书，但是既然让我在这里说话，我就提供给你我的3点个人意见，你应该先做到这3点，然后再去做你的“软件个……”，咳咳，“软件创业公司”。除了这3点，还有一些别的事情你也应该知道，Bob会在这本书里详细地谈到它们。不过在你开始读正文之前，还是先听听我给你的意见。

第一点。如果你说不清楚你的软件解决了什么棘手的问题，就不要去开软件公司。你要问自己，它解决了什么问题？谁需要这个软件？为什么它能解决这个问题？为什么客户愿意付钱让这个问题得到解决？有一次，我去参加一个推介会，6家高科技创业公司做现场宣讲，没有一家想清楚了它们到底要解决什么问题。我看到，第一家公司做的是帮你找到约朋友喝咖啡的时间；第二家公司要求你在浏览器中安装一个插件，这样就能追踪你浏览网页时的每一个动作，就能在历史记录中删去某些动作；第三家公司可以帮你在特定的地点为朋友留下小纸条，比如，假定你的朋友路过了某个酒吧，他们就会收到你留下来的消息……这些公司的共同特点是，它们中没有一家解决了真正迫切需要解决的问题。所有这些公司都注定成为“被困在放满摇椅的屋子之中的长尾巴猫[①]”。

第二点。不要独自一人创办公司。我知道很多人单枪匹马创业成功，但是失败的例子更多。如果你无法说服任何一个你的朋友，他们都不觉得你的主意是可行的，那么也许它就是不行的。除此之外，一个人创业还十分孤独和压抑，没有任何人与你交流思想，为你出谋划策。一旦陷入困境，如果公

① “被困在放满摇椅的屋子之中的长尾巴猫”（a long-tailed cat in a room full of rocking chairs）是一句西方谚语，用来比喻上蹿下跳、神经过敏。

司只是你一个人开的，你很可能就会卷起铺盖、关门大吉。要是有两个人一起创业，你就会感到对你的合伙人负有义务，就会努力撑下去，渡过难关。顺便说一句，不要把你的宠物猫也算作创业伙伴。

第三点。一开始不要抱太高期望。当产品上市的第一个月，没有人知道未来会赚多少钱。我记得5年前，我们开始销售FogBugz的时候，心里一点底也没有，不知道第一个月的销售额会是0美元还是5万美元。在我看来，这两个数字发生的概率完全相等。如今，我已经同足够多的企业家做过交谈，收集了足够的数据，可以告诉你一个标准答案，适用于你的软件创业公司。

没错，我有一个水晶球，我现在就要告诉你一个你最需要知道的事实，也就是在你的产品上市的第一个月你会赚到多少钱。

准备好了吗？

我要说了。

在第一个月，你会赚到……

大约……

364美元，前提是你把每一件事都做对了。如果你开价太低，那么只能赚到40美元。如果你开价太高，那么一分钱也赚不到。如果你心里的期望值大于这个数字，那么你会感到极其失望，也许就放弃了创业，在比尔·盖茨手下找一份工作，把我们这样还在创业的人称作“历史遗留下来的软件个体户”。

364美元，听上去令人沮丧，但其实不是，因为不久以后你就发现，你的软件中存在一个关键缺陷，将50%的潜在客户拒之门外，阻碍了他们掏出钱包。然后，咔嚓！你就把这个缺陷弥补上了，于是一个月就可以赚到728美元。接下来，你会非常非常努力地投入工作，慢慢吸引了一些外界的注意力，你也学会了如何有效应用Google的AdWords（在线广告系统），本地的婚庆策划人在业务宣传册（newsletter）上专门写了一篇关于你的报道。然后，咔嚓！你每个月赚到了1456美元。下一步，你就准备推出2.0版了，内置了垃圾邮件过滤器（spam filtering）和通用Lisp语言[①]解释器，你的客户在

① 通用Lisp语言（Common Lisp）是Lisp语言的一种方言，是为了标准化此前众多的Lisp分支而开发的。它本身并不是一个具体的实现，而是各个Lisp实现所遵循的规范。

互相交流使用你的软件的心得。咔嚓！你每个月赚到了2912美元。你顺势调整了定价，增加了客户支持合同，发布了3.0版，就连Jon Stewart[①]都在他的脱口秀节目*The Daily Show*中提到了你。咔嚓！你每个月赚到了5824美元。

好了，现在我们终于可以吃上热饭了。展望未来的几年，不管你的起步是多么地微不足道，只要你拼命地工作，每过12~18个月，你的收入就没有理由不翻一番（此处省略详细的数学计算过程——作者按），你很快就能在曼哈顿岛上盖起你自己的摩天大楼，楼顶上停着一架直升飞机，让你只用30分钟就能飞到占地20英亩的南安普顿农庄。

总结一下，我所认为的创办软件公司的真正乐趣就是，创造一些东西，自己参与整个过程，悉心培育，不间断地劳作，不断地投入，看着它成长，看着自己一步步得到报偿。这是世界上最带劲的旅程，无论如何，我都不想错过它。

① Jon Stewart是美国电视主持人，他在节目中主要用搞笑的形式讽刺新闻事件和人物，在年轻人中广受欢迎，获得过9次艾美奖。

26

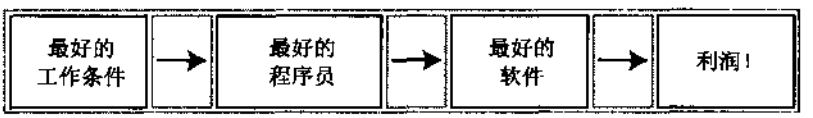
飙高音

2005年7月25日，星期一

2000年3月，我的网站Joel on Software开张了。我在上面大言不惭地说，大多数人都有一种错误的想法，他们认为必须先有一个好的点子，然后才能开办一家成功的软件公司（www.joelonsoftware.com/articles/fog0000000074.html）。

常见的看法是，你在开办软件公司之前，必须先找到一个好点子，认清你的目标市场是什么，想清楚你到底要解决前人没有解决的什么问题，然后再动手实践，去挣大钱。我们把这种想法称为“先做好老鼠夹，再抓老鼠”（build-a-better-mousetrap）。但是，开办软件公司的真正目的应该是将资本转化为有用的软件。

过去5年中，我一直在现实中验证我的理论。2000年9月，我同Michael Pryor一起创办了Fog Creek软件公司，我们的理念可以总结为4个步骤。



这样的总结很合乎我们的情况，尤其反映了我们创办Fog Creek的真正目的，那就是创造一家我们愿意为之工作的软件公司。那个时候，我断言好的工作条件（或者，说得啰嗦一点，“开一家世界上最好的程序员想要为之工作的公司”）将带来利润，这是很自然的事情，就好像巧克力导致发胖、色情电子游戏导致犯罪率升高一样。

不过，今天我不想深究这个话题，而只想谈一点，因为如果这一点不成立，我的整个理论一瞬间就灰飞烟灭了。这一点就是，谈论“最好的程序员”

到底有没有意义？换句话说，程序员之间的能力差别真有这么重要吗？

这个问题的答案，对我们来说显而易见，但是可能对其他许多人来说，它依然需要被证明。

几年前，一家大软件公司考虑并购Fog Creek。那家公司的CEO说，他不是很认同我的观点，不觉得雇用最好的程序员有那么重要。我一听这话，就知道并购没戏。他用了《圣经》中的一个隐喻：你需要的只是一个大卫王[①]，然后再配上一支执行命令的军队就够了。没过多久，那家公司的股价快速地从20美元跌到了5美元，因此我们此前拒绝他们的并购要求看上去就很明智。当然，股价下跌不能归咎于那个CEO崇拜大卫王。

事实上，对于那些喜欢抄来抄去的财经记者，以及那些自己懒得思考、依赖高价请来的咨询公司代替思考的大公司，常识似乎就是，如果想要赚到钱，最重要的莫过于压低程序员的*劳动力成本*。

在有些行业，价格比质量重要。沃尔玛是世界上最大的公司，它的发家秘诀就是卖便宜货，而不是卖好货。如果沃尔玛只卖高质量的商品，商品的成本就要上升，那么沃尔玛整个的价格优势就将丧失。以直筒袜为例，如果沃尔玛想出售经久耐穿的直筒袜，能够经得起各种各样的损耗，比如说，被洗衣机一直洗也不会破，那么这样的直筒袜的各种成分都必须使用最好的原料（比如棉花），才能制造出来合格的成品，每双袜子的成本就必然上升。

既然如此，那么为什么低成本的软件供应商就不能在软件业中生存下来呢？我们只雇用最便宜的程序员为什么就是不行？（说到这里，我想起了Quark软件公司，我很想去问问他们，将整个开发团队都解雇，用低工资的程序员全部取代原来的高工资程序员，会有什么结果[②]。）

让我来告诉你原因，根本的一点就是软件的复制成本为0。这意味着，

① 根据《圣经》的记载，大卫是以色列王国的第二任国王，也是一位英勇的战士。他率领以色列击败了腓力士，杀死了巨人歌利亚。

② 这里的“Quark软件公司”指的是QuarkXpress，它曾经是世界上排名第一的排版软件。原创办人出售公司后，新的出资方解雇了整个开发团队，将开发工作外包到了印度。结果，QuarkXpress 6.0版延迟了两年才发布，实际成本远远高于预算。更糟糕的是，软件中还存在许多错误。6.0版发布的时候，QuarkXpress的市场份额已经从90%降到了50%，之后更是一路下跌，排版软件市场的老大被Adobe公司的InDesign抢走。关于此事的详细情况，请参阅http://discuss.fogcreek.com/joelonsoftware/default.asp?cmd=show&ixPost=86852。

程序员的劳动力成本分摊在你销售出去的所有软件中。对软件来说，如果销售量很大，质量的改进并不会造成单位软件成本的上升。

本质上，软件质量的改进会创造出新价值，而且价值创造的速度要快于成本提升的速度。

或者，换一种不太严格的说法，如果你想压低程序员的工资，那么你就会得到质量很垃圾的软件，而这实际上也不会为你省下很多的钱。

这个现象在娱乐业中也存在。如果你想拍摄一部大片，花钱去请好莱坞大明星布拉德·皮特主演还是物有所值的。虽然他的开价很高，但是因为他是当红大明星，拥有巨大的票房号召力，几百万人会因为他而来看你的电影，你就能把他的报酬分摊到这几百万人头上。

或者，改成这样说，如果你想拍摄一部大片，花钱去请好莱坞大明星安吉丽娜·朱莉主演还是物有所值的。虽然她的开价很高，但是因为她是当红大明星，拥有巨大的票房号召力，几百万人会因为她而来看你的电影，你就能把她的报酬分摊到这几百万人头上。

不过，我到现在依然没有证明任何东西。“最好的程序员”到底是什么意思？不同的程序员开发的软件在质量上真有重大差异吗？

让我们从传统的生产率开始看起，这个指标很直观。但是，衡量程序员的生产率是很困难的，几乎所有你能想到的衡量标准［错误修正涉及的代码行数，功能点（function point）的个数，命令行界面中的参数个数］都很不准确，无法采用。而且，在大型项目中，想要得到具有可比性的数据，难度很大，因为很少会让两个程序员去做同样的事情。

我依靠的数据来自耶鲁大学的Stanley Eisenstat教授。每年他都开设一门需要大量编程的课程，课程编号是CS 323。这门课的作业主要是5道编程题，每一题都要用2星期左右才能完成。这些题对于本科生来说真是相当艰巨，比如开发一个UNIX操作系统的命令行shell程序，写一个ZLW压缩格式的解压程序等。

学生们对于这么重的作业负担怨声载道，以至于Eisenstat教授开始要求大家将做每道题用了多少时间反馈给他。他很用心地收集这些数据，持续了好几年。

我花了一些时间研究这些数据。它们记录了几十个学生在同样的时间用同样的技术做同样的题，这是我知道的唯一一个这样的数据集。所有外界条件都被很好地控制了，就像在做试验一样。

数据集一共包括12道题，我做的第一件事就是计算学生在每道题上花费的平均小时数、最小小时数、最大小时数和标准差。下面就是结果。

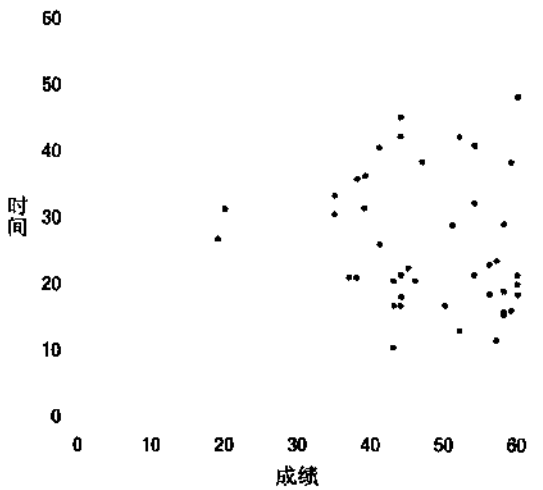
题　目	平均小时数	最小小时数	最大小时数	标准差
CMDLINE99	14.84	4.67	29.25	5.82
COMPRESS00	33.83	11.58	77.00	14.51
COMPRESS01	25.78	10.00	48.00	9.96
COMPRESS99	27.47	6.67	69.50	13.62
LEXHIST01	17.39	5.50	39.25	7.39
MAKE01	22.03	8.25	51.50	8.91
MAKE99	22.12	6.77	52.75	10.72
SHELL00	22.98	10.00	38.68	7.17
SHELL01	17.95	6.00	45.00	7.66
SHELL99	20.38	4.50	41.77	7.03
TAR00	12.39	4.00	69.00	10.57
TEX00	21.22	6.00	75.00	12.11
所有题目	21.44	4.00	77.00	11.16

你从中注意到的最明显的东西就是数据的变动幅度非常大。最快的学生做题的速度比普通学生快3到4倍，比最慢的学生快10倍。标准差之大非常惊人。嗯，我因此想到，可能其中有些学生是在敷衍了事，马马虎虎地做作业。他们只花4小时就完成了作业，但是写出来的程序却无法运行，我决定把这样的学生排除在外。于是，我对数据进行了筛选，只包括那些代码质量最高的前25%的学生。我必须在这里说，Eisenstat教授的评分是极端客观的。学生交上来的代码都通过自动测试进行打分，通过的测试个数决定了分数的高低，分数就像公式那样被计算出来，不考虑任何其他因素。不良的编码风格，或者迟交作业，都不会被扣分。

下面就是代码质量最高的前25%的学生的数据：

题　目	平均小时数	最小小时数	最大小时数	标准差
CMDLINE99	13.89	8.68	29.25	6.55
COMPRESS00	37.40	23.25	77.00	16.14
COMPRESS01	23.76	15.00	48.00	11.14
COMPRESS99	20.95	6.67	39.17	9.70
LEXHIST01	14.32	7.75	22.00	4.39
MAKE01	22.02	14.50	36.00	6.87
MAKE99	22.54	8.00	50.75	14.80
SHELL00	23.13	18.00	30.50	4.27
SHELL01	16.20	6.00	34.00	8.67
SHELL99	20.98	13.15	32.00	5.77
TAR00	11.96	6.35	18.00	4.09
TEX00	16.58	6.92	30.50	7.32
所有题目	20.49	6.00	77.00	10.93

看到了吗，几乎没有差别！前25%的数据得到的标准差，几乎与整体标准差完全一样。事实上，仔细审视这些数据你就会很清楚地看到，时间和成绩之间不存在明显的相关性。下面就是其中一道题目的散点图，很具有代表性。我选择的题目是COMPRESS01，那是一道2001年的题目，要求学生开发一个Ziv-Lempel-Welch压缩器。我选择这道题的原因是因为它的标准差很接近总体的标准差。



从图中根本看不出什么东西，不过这就是我想说的。作业的质量与所花费的时间基本上是不相关的。

关于此事，我问过Eisenstat教授，他指出还有另外一个因素。因为每次都规定交作业的时间（通常是在某一天的午夜之前），迟交会受到很严厉的批评，所以许多学生眼看时间要到了，就停下来将还没做完的作业交上来。换言之，部分因为布置作业和交作业之间的时间是有限的，所以学生花在这些作业上的最长时间是有上限的。如果学生们有无限的时间做作业（这更接近现实世界中的情况），那么数据之间的极差（最大值与最小值的差）还会更大。

当然，这些数据并非完全可靠，可能存在谎报。有些学生也许会夸大用来做题的时间，希望以此博得一点同情，下次就会换来更容易的题目。（祝君好运！今天CS 323的作业同我在20世纪80年代上大学时是一样的。）另一些学生可能会少报，因为他们忘了记录时间。但是，就算存在这样对数据的扭曲，我依然相信这些数据说明了，程序员的生产率有5倍或10倍的差距。

等等，我还没说完

如果程序员之间仅有的差别就是生产率，那么你也许会想，你可以用5个平庸的程序员代替1个优秀的程序员。显然，这是行不通的。还记得布鲁克斯法则（Brooks' Law）吗？——向一个已经延误的软件项目增加人手，只会使它更加延误——这就是原因。一个优秀的程序员独自完成一项任务，就不需要额外的沟通和协调。如果同样的任务让5个程序员一起完成，他们之间就必须沟通和协调。这会花掉大量时间。开发团队越小，就越能获得额外的收益。人力与工时的互换真的是一个神话[①]。

① 此处指的是Frederick Brooks所写的软件项目管理名著《人月神话》（*The Mythical Man-Month*）。所谓“人月”就是一个人在一个月内所能完成的工作量。假如有个项目预估需要12个月，那么派4个人来处理这个项目，理论上只要3个月就能完成。但是，Brooks认为这种换算机制在软件业中行不通，是一个神话，因为软件项目是交互关系复杂的工作，需要大量的沟通成本，人力的增加会使沟通成本急剧上升，反而无法达到缩短工时的目的。在本质上，软件项目的人力与工时是无法互换的，当项目进度落后时，光靠增加人力到该项目中并不会加快进度，反而有可能使进度更加延后。

等等，我还是没说完

用许多平庸的程序员取代少数优秀的程序员，这种做法的真正问题在于，不管平庸的程序员工作多长时间，他们做出来的东西都无法像优秀程序员做得那样好。

5个Antonio Salieri[①]也写不出莫扎特的《安魂曲》。永远也写不出，埋头写100年也没用。

5个Jim Davis［他是一点都不好笑的卡通人物加菲猫的创造者，“加菲猫”里面20%的笑话是在讲星期一有多糟糕，剩下的笑话则是那只猫有多么喜欢意大利面（而且这些话都被当作妙语）］，哪怕5个Jim Davis把余生都用来写喜剧，也永远写不出电视连续剧*Seinfeld*中*Soup Nazi*那一集的剧本[②]。

Creative公司的音乐播放器Zen的开发团队即使再花上许多年，对他们那个丑陋的iPod仿制品进行美化，也永远造不出像苹果公司的iPod那样优美雅致、令人陶醉的播放器。他们不会对苹果公司的市场份额造成任何影响，因为他们的团队中没有那种神奇的设计天才。他们就是缺少设计的天赋。

一流的歌唱演员不管在什么时候，都可以很轻松地唱出高音，而平庸的歌唱演员就是永远做不到这一点。莫扎特的歌剧《魔笛》中有一段著名的咏叹调*Queen of the Night*，音高必须达到F6[③]才能唱好这首歌，世界上能达到这个标准的女高音都快要绝迹了，而飙不到著名的F6，你就是不能表演*Queen of the Night*。

软件真地与歌唱家的高音有关系吗？“可能只是某些方面吧，”你说，“不过应该同我关系不大，我为医疗废物行业开发应收账款的用户界面。”说得不错。我针对的是生产最终产品的公司，它们的成败完全取决于产品的质

① Antonio Salieri（1750—1825）是意大利作曲家。传说中，他的才能不及莫扎特，在嫉妒心的驱使下他毒死了莫扎特。

② *Seinfeld*是一部美国经典喜剧电视连续剧，讲述了4个好朋友在纽约的生活。1989年至1998年在NBC电视网播出，共9季。*Soup Nazi*是第七季的第六集，在1995年11月2日播出。

③ F6是女高音的最高音高。

量。如果你开发的软件只是为了在公司内部使用，开发的目的是为了配合公司的运营，而不是销售，那么这种软件对你来说，可能只要够用就行了，而不需要特别优秀。

在过去的几年中，我们已经看到了非常多的优秀软件，它们是真正的高音选手，它们的音高是平庸的程序员所无法企及的。

回想2003年的时候，Nullsoft软件公司发布了Winamp的一个新版本，他们的网站上贴出了如下的告示：

- 华丽的新外观！
- 绝妙的新功能！
- 大部分功能确实能用！

最后一句话——大部分功能确实能用——让每个人都笑出了声。用户因此觉得很开心，对Winamp感到兴奋。他们使用它，并且告诉朋友他们打心眼儿里认为Winamp真是棒，这都是因为Winamp的网站上写着“大部分功能确实能用”。这难道不是很酷吗？

如果你将一大堆多余的程序员塞进Windows Media Player的开发团队，他们能不能唱出高音？想也别想，一千年都没有这种可能。因为你向开发团队中加进去的人越多，其中就越可能有一个坏脾气的家伙，他认定在网站中写上“大部分功能确实能用”是一种不专业、不成熟的表现。

更不要提了，网站上还有另一句话：“Winamp 3：差不多与Winamp 2一样新颖！”

就是这样的东西，让我们爱上了Winamp。

当AOL时代华纳集团的那些猪头们将黑手伸向Winamp[①]，网站上好玩的东西从此就消失了。这些人看上去简直就像电影《莫扎特传》（Amadeus）中的反派角色Antonio Salieri，怒气冲冲、抱怨不满、哭哭啼啼，一心一意要把所有创造力的标志都摧毁，因为这些标志可能会吓到明尼苏达州的某位老太太，他们为了这个，不惜将所有使得人们喜欢这个产品的因素都清除得

① Nullsoft成立于1997年，同年推出Winamp后一举成名，Winamp成为最受欢迎的MP3歌曲播放软件之一。1999年6月1日Nullsoft被美国在线（AOL）购并，成为AOL旗下的子公司。

干干净净。

换个例子，再来看看iPod。它是不能换电池的。所以，如果电池坏了，那就太糟糕了。你只好去买一个新的iPod。当然，你事实上可以把它送回苹果公司，他们会帮你拿到工厂里换个电池，不过收费是65.95美元。好贵！

为什么不允许你换电池？

我的理论是，苹果公司要把iPod做成一个优美、性感的播放器，不希望破坏它无比光滑、天衣无缝的表面，不希望它像其他很便宜、很垃圾的电子类消费产品一样，背后有一个巨大丑陋的电池盖，因为电池盖的闩锁总是很容易断掉，而电池盖的细缝中总是嵌满了口袋布料的绒毛和其他各种各样恶心的东西。iPod是我见过的表面最光洁无缝的消费类电子产品。它是那样优美。你能感觉到这种优美，仿佛河流中一块光滑的鹅卵石。一个电池盖的闩锁能把整个鹅卵石效果全毁掉。

苹果公司的决定完全出于风格（style）的考虑。事实上，iPod里面到处都考虑到了风格。这种风格不是微软公司的100个程序员、或者Creative公司的200个工业设计师（这家公司真是叫错了名字[①]）所能达到的，因为这些公司没有Jonathan Ive[②]，而外头能找到的Jonathan Ive并不多。

很抱歉，我一谈起iPod就停不下来。那个漂亮的拇指转轮，还配上了轻轻的咔嗒声……苹果公司花了额外的钱，在iPod中装了一个喇叭，就为了让拇指转轮的咔嗒声听起来好像就来自转轮本身一样。他们原本可以省下这几分钱——几分钱啊——将咔嗒声通过耳机传出来。但是拇指转轮让你觉得你在控制。人们喜欢控制的感觉。控制的感觉让人们感到开心。拇指转轮操作的反应非常快速流畅，还带有声音，这一点就会让你感到开心。它与世界上其他6000种便携式消费类电子废物不一样，那些产品的启动时间是那样漫长，你按下on/off开关，等上一分钟才能知道是否有事情发生。你在控制它吗？谁知道呢？想一想，上一次你使用按下电源开关后立刻就能通话的手机是什么时候的事情了？

风格。

① Creative的意思是“创新”，公司的中文名也是这个词。

② Jonathan Ive是苹果公司工业设计部门的高级副总裁，也是iMac、MacBook、iPod和iPhone的主要设计者。

开心。

情感的诉求。

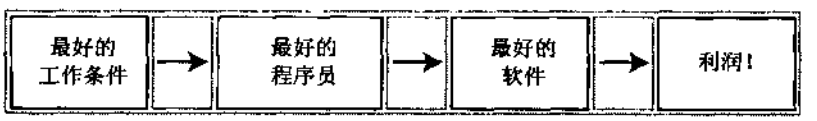
这些就是大热门产品的成功要素，不管这种产品是软件还是电影还是消费类电子产品，都一样。如果你没有把这些要素做对，虽然你可能依然能够解决问题，但是你的产品不会成为排名第一的热门商品，也就无法让你公司中的每一个人致富，无法让你们都开上很有风格、很让人开心、很打动人的汽车（比如法拉利的Spider F1），无法让你们还剩下足够的钱在后院盖一所隐居的小屋。

这不仅仅是“生产率高10倍”的问题，而是“普通”程序员永远都唱不出开发优秀软件所需要的那种高音。

令人难过的是，对于非商业性的软件开发，这一点并不完全适用。内部使用的软件很少重要到需要雇用巨星来开发。没人会花钱请Dolly Parton[①]在婚礼上演唱。这就是为什么最令人满意的职业生涯（如果你是程序员）往往是在真正的软件公司里，而不是在银行里担任IT技术人员。

眼下的软件市场有点“赢家通吃”（winner-take-all）的味道。除了苹果公司，没有其他人能在MP3播放器市场上赚到钱。除了微软公司，没有其他人能在电子表格和文字处理器市场上赚到钱。没错，我知道，他们做了破坏竞争的事情才得到今天的市场地位，但是这改变不了这是一个“赢家通吃”市场的事实。

在市场上排名第二，或者有一个“还不错”的产品，对你来说就意味着失败。你的产品必须非常优异，我的意思是，好到大家愿意谈论它。想要开发优异的软件，你的唯一希望就是依靠那些真正优秀的软件天才，只有他们才能为你创造出来。下面就是整个的计划：



① Dolly Parton是美国著名流行歌手。

第七部分

经营软件公司

27 仿生学办公室

28 他山之石，不可攻玉

29 简化性

30 揉一揉，搓一搓

31 组织beta测试的十二个最高秘诀

32 建立优质客户服务的七个步骤

27

仿生学办公室

2003年9月24日，星期三

言归正传。

这事耗费的时间远远超过了预期。

最终，我们还是正式地把Fog Creek搬迁到了纽约第八大道535号。这时，距离我开始踏破铁鞋、寻找一个新办公场所，已经过去了10个月。在此之前，我们一直在我祖母的老房子里工作，在那儿我们度过了公司创立后的头几年。我们就在卧室和后花园里办公。

大多数软件经理心里都知道什么是好的办公空间，但是他们也知道，这样的办公空间不会实现，也不可能实现。办公空间似乎成了没人能做对、没人能干涉的一件事情。找个地方，签下十年租约，然后公司就可以搬家了。但是不管怎样，最终拍板办公空间如何布置的人总是软件经理。他决定如何放置搁板，如何将空间分割成一个个好像用来圈养牲口的小隔间，如何让整个办公室看起来好像一个大养牛场，以使得搬家后的星期一，所有人第一次来上班时，能够各就各位、待在自己的位置上。

但是，这家公司是我开的，决定权在我手里。既然如此，我就决定做一点符合自己心意的事情。

我可能对建筑风格过于挑剔。比起一般的程序员，我也许对周围的物理环境投入了过多的注意力。也有可能，我太把这当回事了。但是，有3个理由使我觉得必须认真对待这件事。

□ 许多证据表明，良好的办公空间——尤其是单独的办公室——能够

提高程序员的生产率。

- 非常漂亮的、有窗的个人办公室，会使得招募明星程序员变得容易许多。要知道，明星程序员的产出比普通的优秀程序员要多出10倍。我在纽约开软件公司，支付的是纽约水平的薪水，我的对手在印度班加罗尔开软件公司，支付的是印度水平的薪水，如果我想在竞争中获胜，就需要那些明星程序员。他们来面试的时候，看到公司里优越的办公条件，吃惊得下巴都要掉下来。我要的就是这种戏剧效果。
- 嗨，这也是我办公的地方，我会长年累月地待在这里，远离朋友和家庭，所以这里最好条件好点。

一个开窍的CEO，再加上建筑师Roy Leone，以及一大片空间（人均建筑面积40平方米）。就这样，我开始着手创造一个软件开发的终极办公环境。

在建筑师口中，“客户简报”（brief）这个词同程序员口中的“系统要求”（system requirement）是一个意思。下面就是我给Roy Leone的“客户简报”。

(1) 个人办公室，带有可以关上的门，这是绝对必需的，不能协商。

(2) 程序员需要许多电源接口。他们应该可以将各种新颖的小玩意插进与电脑桌齐高的插座，而不用在地板上拖电线。

(3) 我们需要可以方便地转接各种数据线（电话线、网线、有线电视线、警报器线等），装修结束后就再也不想在墙壁上打洞了。

(4) 办公室可以用来“结对编程[①]”（pair programming）。

(5) 整天对着一台显示器工作就需要通过注视远方，使眼睛得到休息，所以显示器不应该靠着一整堵墙摆放。

(6) 办公室应该是一个窝，一个能够很愉快度过时间的地方。如果你想在下班后与朋友一起吃饭，你应该会想要把他们叫到办公室来一起聚餐。还是Philip Greenspun[②]说得直截了当：“你的公司的成功依赖于程序员真正以办公室为家的程度。如果要让办公室成为程序员的共同选择，最好让它比普通程序员的家更舒适。有两个办法可以做到这一点。一个是只雇用那些住在条件极端恶劣公寓中的程序员，另一个是创造一个真正舒适的办公室。”（ccm.redhat.com/asj/managing-software-engineers/）

① “结对编程”是软件开发中的一种方法，指两个程序员在一起编程，一个人负责编写原始代码，另一个人负责查看代码。

② Philip Greenspun是photo.net创始人。

Roy Leone的工作非常出色。你花钱请来建筑师，要的就是这样的服务。我预测，在设计程序员办公室方面，他会成为世界级专家。下面就是他如何将我的“客户简报”转化成了具体的三维空间设计。

个人办公室。我们不仅得到了宽敞的、有窗户的个人办公室，而且那些公共的办公区（供非程序员使用）也被很聪明地隐藏在墙壁之间不规则的角落中，这样一来，每个人就都有了自己的个人办公空间，一眼望去不会看到其他人。

办公室和各个办公区域之间的隔板是用高科技的半透明丙烯酸材料做的，能够发出柔光，为室内提供了自然光，同时又不会减少私密性。

电源。每张电脑桌有20个（没错，就是20个）电源插孔，其中4个是橙色的，与放在一个中央机柜中的不间断电源UPS相连，所以你就不需要在每一间办公室中都放一个UPS。

电源插孔都位于桌面之下的一条特殊的槽中，这条槽与桌面等长，6英尺深，6英尺宽。你可以将所有的各种线缆都恰到好处地藏在这条槽中，它上面还有一个很方便的盖板，合上后完全与桌面融为一体。

布线。在机房中接近天花板的地方，我们安装了一套桥架系统[①]（Snake Tray system），从这里开始贯穿整个办公空间，每间屋子里都有。因为这套系统非常好用，所以如果你想用任何线缆（低电压的）将A点和B点连起来，你就能很容易地做到。我们是星期五搬入新址的，只用了半个小时就重新布置好了办公室内部的局域网，这说明桥架系统已经证明了自己的价值。每间办公室里还有一个独立的八口交换机，你可以把自己的笔记本、台式机、Macintosh都插进去，你还可以再插入一台老式电脑，专门在主力电脑升级本日Windows Update后需要重启时用于访问Joel on Software网站。这样的话，还剩下3个口。（各位数学天才请注意，不要给我发电子邮件，多出来的那个口用于插上行线。）我对笨笨的大楼经理表示嘲笑，他居然还以为每间屋子一个局域网插口就够了。不过，也许对于律师事务所确实是够了。

结对编程。通常，办公桌是L型的，许多程序员都将这种桌子靠着墙角放。当他们需要暂时性合作项目，或者结对编程，甚至只是想向另一个人展

① 桥架系统就是一套悬空的线缆托架，具体情况参见http://www.snaketray.com/。

示屏幕上的内容时，第二个人就没有选择。他要么斜靠着桌子，探出身子来看，要么就是在第一个人身后，隔着前者的肩膀看。为了避免这种情形，我们把所有的办公桌都设计成长条形，因此无论第一个程序员坐在哪里，他的身边总是有空间，可供第二个人再拉一把椅子过来，并排坐下。

缓解视力疲劳。虽然办公桌是对着墙放的，但是墙上有一个对内的窗口，可以很巧妙地看到隔壁办公室的一角，以及通过那间办公室的窗口向室外看。由于设计得非常巧妙，这扇窗户并不会影响到私密性，因为即使通过它可以看到隔壁房间，但是视角是受限的，在大部分的位置，你实际上所能看到的只是隔壁房间的一个小小角落以及里面对着室外的一扇窗。这样设计的最终结果就是，每间办公室的三面墙上有窗户，其中有两扇可以看到外部，这就符合建筑学上“两面采光”（Light on Two Sides of Every Room）的模式。这是一项相当可观的成就，你实际上提出了一整套的方案，关于如何在传统大楼中为每一个员工提供一个小小的角落实现独立办公。这从另一方面证明了聘请一个优秀建筑师是多么值得。

舒适的小窝。我们在整个办公室中还配上了一个小厨房和一个休息区。在休息区里有沙发和一个巨大的等离子高清电视以及DVD播放机。我们计划再放一张台球桌和电子游戏的主机。另外，独享的个人办公室意味着你可以不戴耳机在里面听音乐，将音量开得很大，没人会来管你。

我的评估

如果人员全部到位，整个办公室的月租金分摊到每个人头上，大约为人均700美元。装修的费用没有超出预算，几乎全部由房东来承担。在全世界的软件公司中，人均700美元的月租金应当属于偏高的。但是，如果这能够使我们雇到的优秀程序员的排名从前1%缩小到前0.1%，那么这样的支出就是值得的。

28

他山之石，不可攻玉

2000年12月2日，星期六

到昨天为止，FogBugz的使用许可证还是这样写：你不可以对程序进行逆向工程（reverse engineer），不可以试图阅读源代码，也不可以用任何方式修改程序。各种各样的老实人曾经问过我们源代码许可证要多少钱，他们想买一张以便修改程序。

嗯嗯嗯。为什么我们的许可证不许你改变源代码呢？我一点理由也想不出来。事实上，我倒是想出了一大堆理由允许你改变源代码，所以我立刻就把许可证换掉了。就是因为这事儿，你现在必须坐下来，听我来讲一件陈芝麻烂谷子的往事。

时光倒流，回到1995年，我还在维亚康姆公司工作。在那儿，我们一小群人堪称是互联网时代艰苦的先锋，为维亚康姆旗下的各项赚钱事业制作网站。

那时还没有应用程序服务器（application server），只有一家很无能的Sybase公司。如果你想在互联网上使用Sybase的数据库，Sybase就会让你去购买一个许可证，与你的网站连接的每一个客户端收费150美元。Netscape公司的网络服务器只是刚刚推出1.0版[①]。

另一家很勇敢的公司叫做Illustra，它向外界声称，自己的数据库管理系统非常适合在互联网上使用。你看，Illustra被设计成能够很容易地添加新的

① Netscape公司是在1994年12月推出1.0版网络服务器的，当时叫做Netscape NetSite Web Server。但是，直到1996年3年，该公司发布2.0版的时候，这个产品才能正式投入使用。

数据类型，只要你写一些C代码，然后再把它与他们的DBMS（数据库管理系统）连接就行了。（任何用过DBMS的程序员都会告诉你，这样做听起来过于冒险。C代码？连接？哎呀。）这个功能最初是针对一些令人激动的数据类型开发的，比如经度/纬度、时间序列等。但是，那时正赶上互联网兴起。Illustra公司就写了一个叫做“Web Blade”的东西，将数据库管理系统与这个东西连接在一起。Web Blade不是一个很成熟的产品，据称它可以从数据库中取出数据，实时生成动态网页。在1995年，每个程序员面对的最大难题就是如何生成动态网页。

我在维亚康姆的一个同事被叫去负责开发一个电子商务网站，主要出售最流行的大热门商品。我不跟你开玩笑，那个网站就是卖CD的。（因为那时人们一听到大热门商品，就会想到CD，对不对？）闲话少说，那个同事认定Illustra非常适合完成这项工作。那时，Illustra的价格大概是12.5万美元左右，要让维亚康姆这棵摇钱树吐出12.5万美元这笔巨款，简直难于登天，所以好久也没有搞定。有一段日子，我的同事折了一个纸杯，放在他的办公室小隔间里，上面写着“Illustra基金”，好心人士路过就往里面扔一点钱，就这样总算搞到了一点儿美元。CTO与Illustra进行了长时间的艰苦谈判，最终达成了协议。我们终于装上了Illustra，可以开工了。

不幸的是，灾难降临了。Illustra的Web Blade根本不是成熟的产品，完全不能满足需要。它几分钟就宕机一次。即使没有宕机，情况也没好到哪儿去。它所用的编程语言是唯一一种我所见过的不符合“图灵等价”（Turing-equivalent）的语言[①]。软件自带的“许可证管理器”（license manager）总是提示你未经许可，因此你的网站就会无法访问。用它来建站真是太可怕了，我的那位同事算是倒霉透了。所以，当他们来找我，对我说：“Joel，你来为MTV[②]做个网站。”我的反应是：“啊，不会吧。”

“能不能不用Illustra？”我央求道。

“既然你这么说，就不用吧。但是你用什么代替呢？”那个年代真没有任何其他应用程序服务器可以选择。没有PHP，也没有内置TCL语言的

① 如果一台计算机与“图灵机”的行为完全一致，就被称为“图灵等价”。在不严格的意义上，所有的现代计算机都拥有“图灵机”的计算功能，因此Joel在这里说这种语言不是“图灵等价”，等于在讽刺这种语言无法在现代计算机上使用。

② MTV（Music Television）是维亚康姆旗下的一个专门播放音乐节目的有线电视网。

AOLserver[①]，即使使用Perl，也必须自己修改源代码，然后重新编译。总之，青霉素还没有发明出来，我们的生活是多么悲惨啊。

我的名誉眼看就要毁于一旦。在这千钧一发的时刻，我想通了一件事。Illustra最可怕的地方在于，它崩溃的时候你根本无计可施。但是，我想如果你有它的源代码，那么就算它崩溃了，你至少还可以在笔记本电脑上进行调试，然后把出错的地方改好。你也许会整整一个星期都熬通宵，不断调试别人的代码，但是你至少有机会解决问题，不会无计可施。要是你没有源代码，那么借用一句成语，他山之石，你就无法攻玉了，只能把它当作石头。

通过这件事，我学到了软件开发中重要的一课。那就是，对你最重要、最关键的部分，你一定要使用更原始的工具。举例来说，假定你正在写一个很酷的3D射击游戏（就像同样在那个时期出现的《雷神之锤》[②]），你排在第一位的卖点就是生成最酷的3D图像。为了达到这个目的，你就不能使用任何你能找到的3D库。你一定要写一个你自己的3D库，因为这是达到你目的的基础。那些使用诸如DirectX这样现成的3D库的公司，只是因为他们的卖点不是软件的3D表现（也许他们有一个很好的故事）。

一旦我认识到了这一点，我就决定不再信任任何其他人写出的糟糕的应用程序服务器，而要自己重新用C++和Netscape Server的底层API写一个。因为我想通了，这样的话，假定某个地方出错了，那就是我自己的代码出错了，我能加以处理，并且能最终解决问题。这就是开源/自由软件的最大优点之一，即使你能够负担12.5万美元，从Illustra买来一块石头，但是在这一点上，它还是比不过开源/自由软件。至少出错的时候，你多多少少有机会解决问题，这样就不会被解雇，MTV电视台里那些只有多动症发作的时候才会待人和善的家伙才不会对你大发雷霆。

当我坐下来构建一个软件系统时，我必须做出决定，到底使用哪些工具。一个好的软件架构师只会使用那些“可以被信任”或“可以被维护”的工具。一种工具“可以被信任”，并不意味着它是由像IBM那样可以被信任的大公司制造的，而是意味着你百分百确信它会正常运作。比如，我觉得现在的大

① AOLserver是AOL公司推出的一款开源的Web服务器软件，1995年推出。它是世界上第一个内置脚本语言的HTTP服务器，所内置的语言就是TCL语言（Tool Command Language，工具命令语言）。

② 《雷神之锤》（quake）是一个第一人称视角的射击游戏，在1996年6月22日上市。

多数Windows程序员对Visual C++都很信任。另一方面，他们可能不怎么信任MFC（微软基本类库），但是MFC 提供源码，因此当你发现异步套接库[①]（async socket library）写得糟糕透顶时，你不信任它，但至少你可以维护它。所以，把你的职业生涯赌在MFC的前途上也不会有大问题。

同样地，你可以把你的职业生涯赌在Oracle DBMS上，因为它有效运行，大家都知道它。你也可以把职业生涯赌在Berkeley DB[②]上，因为就算它把一切都搞糟了，你也能查看源代码，解决问题。但是，你可能就不能把职业生涯赌在那些既不开源也不知名的工具上面。你可以把它们用来测试，但是它们不是值得押上你的未来的工具。

于是，我自然就想到了如何才能使得那些聪明的工程师放心地在FogBugz上面押注。一个几乎完全偶然的因素，促使我下定决心公开FogBugz的源码，那就是那时我正好看到ASP网页正是这样运作的，它的源码就是公开的。我做出这个决定后，一点都不担忧。FogBugz是一个程序错误跟踪软件，里面并没有什么奇妙的、秘密的算法。它根本不是高深莫测的东西。（实际上，在任何一种软件中，都极少有奇妙的、秘密的算法。因为分解可执行文件、搞清楚它是怎么运行的，这是一件相当容易的事，没有什么了不起的，并不像版权法律师说的那样关系重大。）对我来说，别人看到源码，或者为了个人使用而修改源码，都是无关紧要的。

如果你自行修改厂商提供的源码，就会担风险，因为如果厂商对程序进行升级，你就要花一大把时间将你的修改转移到新版本上。不过，在这方面，我能帮你减轻一点负担。如果你发现FogBugz有缺陷，你可以把你对源码的修改发给我，我会把你的修改加入软件的下一个版本。这样做是为了让用户更放心：FogBugz确实能用；就算它在某些很关键的场合不能用，用户也可以自己动手修改，而不是眼睁睁看着它，无计可施；如果用户的修改是有效的，而且比较重要，那么就会被加入源代码树（source tree），包括在下一个版本中，我们的生活都将因此变得更舒服一些。

说到这里，开源软件和自由软件的支持者几乎一定会大叫，我都能听到他们的声音了："你这只笨鹅！只要把软件开源就行了，就没事了！开源会

① 异步套接库是MFC的一部分。

② Berkeley DB是一种开源的嵌入式数据库。

解决你上面说的那些问题！”你们说得好听，可是我的这家小公司要养3个程序员，每月的运营费用高达40000美元。所以，我们的软件需要收费，我们不会为收费而道歉，因为它值这个价。我们不会自称自己是开源软件，但是这不妨碍我们借鉴开源社区中一些好的做法。我们能够保证，FogBugz是一个安全的选择。

29

简 化 性

程序员最可信赖的求职帮手

2006年12月9日，星期六

Donald Norman有一个观点，他认为简化的重要性被夸大了。（在欣赏了一番简化是多么重要之后，）记者们聚在一起，评点他们从各处收集到的简化的产品。他们纷纷抱怨，这些产品缺少他们眼中的一些‘关键’功能。那么，当人们声称他们喜欢简化的时候，他们心里到底在想什么？所有的操作只用一个按钮就能完成，他们觉得好吗？当然好啊，但是他们最喜欢的功能可一样也不能少。”（www.jnd.org/dn.mss/simplicity_is_highly.html）

很久以前，我写过下面的话（摘自*Joel on Software*）。

许多程序员都深受古老的“80/20”法则的诱惑。这条法则看上去似乎意义重大，80%的用户使用20%的功能。你因此说服自己，只要开发20%的功能就行了，软件的销量依然会保持在八成左右。

不幸的是，那个20%是一个变量。每个人需要的功能都不一样。过去十年中，许多公司下定决心，坚决不从其他人的教训中获取经验，一定要尝试发行精简版的、只实现20%功能的文字处理软件。单单我知道的这样的公司可能就有几十家。这种事情早就不新鲜了，个人电脑的历史有多久，它的历史就有多久。大多数时候，事情是这样开始的。这些公司先把软件交给一个记者，让他去写报道。那个记者就使用这种新的文字处理软件写报道，发表对它的评论。写完后，记者想使用“字数统计”功能，这个功能对记者很重要，因为大多数记者都需要精确知道自己写的字数。可是，找不到这个功能，原因是它属于“没人使用的80%功能”之中的一种。那个记者最后写出的报道，一方面声称软件发行精简版是好事，庞大臃肿

的软件是坏事，同时另一方面，又说他没法使用这一种软件，因为它不能统计字数。这简直就是自相矛盾，这两个方面完全是对立的。

简单说，生产只实现20%功能的产品是一种出色的自力更生战略（bootstrapping strategy），因为你用有限的资源生产出了产品，同时还赢得了用户。这就好像柔道中的战术，扬长避短，将自己的劣势转化为优势。电影*The Blair Witch*就是一个例子。几个学生想拍一部电影，但是根本没有钱，只有一部手持摄像机，这是他们仅有的设备。他们急中生智，想出了一套情节，使得缺乏设备反而成为了一种优势[①]。表面上，你在出售一种“简化”的产品，好像很理想的样子，但是凑巧的是，这是你手头的资源能够生产出来的仅有的产品。这样的巧合真是太好了，你只能做成这样，但是这已经是一种理想的产品！

在我看来，自力更生时迫不得已的选择并不是一个好的长期战略。因为你的产品太简化了，两个人办一家创业公司就能复制你的产品，你根本无法防止这种事情发生。而且，逐渐地由于人类天性的原因，你的产品也会被抛弃。Donald Norman就说过：“用户想要更多的功能。”你不能因为手持摄像机很适合*The Blair Witch*这部片子，就认定所有的好莱坞大片都要用手持摄像机拍摄。

简化性的拥护者会举出37signals和苹果公司iPod播放器作为津津乐道的证据，证明简化的产品十分畅销。我会争辩说，在这两个例子中，商业成功来自于许多因素的共同作用，比如培养追随者、传播理念、简洁明快的设计、感情上的诉求、美学成就、很短的反应时间、及时的用户反馈、与用户行为（user model）相对应的程序模型（program model）、整体上高度的易用性、让用户感到控制权都在自己手中等。所有这些因素都是用户非常喜欢、愿意为之付钱的优点，从这个角度看，它们都可以被视作产品的某种“功能”。但是，它们中没有一种可以被看作是“简化性”的表现。比如，iPod是一种优美的产品，这也算是一种“功能”，Creative公司的Zen Ultra Nomad Jukebox播放器就没有这种功能。所以，我选择购买iPod，你说对不对？还是以iPod

① 这是1999年公映的一部美国电影，由三个大学生自费拍摄完成。该片是一部恐怖片，大部分场景都发生在黄昏和夜间的树林中，因此很大程度上掩盖了制作的粗糙。该片的成本只有3万美元，电影公司用了100万美元收购，上映后取得了极大的票房成功，总票房达到了2000多万美元。

为例，它的美的外在表现形式正好通过一种干净简单的设计传达出来，但是它不一定非得采用这种设计不可。悍马汽车也有非常动人的美学效果，但恰恰是通过它的粗犷和结构复杂而表现出来的。

我认为将iPod的成功归因于简化的功能是不正确的。如果你相信这种错误的观点，你就会相信在其他产品上，也要减少功能才能让产品变得更成功。根据我6年来创办Fog Creek软件公司的经验，我可以告诉你，在我们做过的所有事情中，最能增加收入的就是推出一个带有更多功能的新版本。这一招最管用。一个新功能的软件新版本对我们财务报表的影响，是绝对无法否认的。它就好像地心引力一样，能把钱吸过来。我们试过Google的在线广告系统，也试过各种业务推广分成计划（affiliate scheme），或者看到新闻媒体上出现介绍FogBugz的文章，它们对财务报表的影响几乎很难被察觉。但是如果推出新版本，其中包含了新功能，我们就会看到公司的收入出现了迅猛的、确凿的、重大的、持续性的增长。

如果你把“简化性”这个词理解成与用户行为紧密对应的程序模型，以及由此产生的产品的易用性，那么没问题，你会看到“简化性”发挥威力。如果你把“简化性”这个词理解成简洁明快的视觉呈现，把它仅仅描述为一种美学上的定义，那么没问题，你会看到“简化性”发挥威力。时下，最简主义（Minimalist）美学相当流行。但是，如果你把“简化性”这个词理解成“不提供大量功能”或者“只提供一种功能，并把这种功能完美实现”，那么我会为你大胆说出心里话而喝彩，但是你不可能会有很大的发展，一种故意减少功能的产品是没有前途的。甚至就连iPod都附带一个纸牌游戏，而号称最简单的代办事项清单网站tadalist.com也提供对RSS的支持。

行了，今天就写这么多，我的手机该升级换代了，我要一个能够高速上网、收发电子邮件、抓取podcast内容、播放MP3音乐的手机。

30

揉一揉，搓一搓

2002年1月23日，星期三

有时，人们想要重写整个的基础代码（code base）。原因之一就是，他们本来的基础代码并非为所用的场合而设计。那些代码的设计目的可能是创造一个原型（prototype），也可能是为了做实验，或者是完成课后作业，或者是为了在9个月里从零开始创造一家上市公司而赶写出来的，或者就是做一个临时的demo。但是，不管由于哪种原因，总之那些代码被保留了下来，并且发展成一大团乱糟糟的东西。整个体系变得非常脆弱，再也不可能添加代码。所有人都牢骚满腹，老程序员绝望地辞职。新程序员被招进来，但是根本弄不懂这些代码，他们只好设法说服管理层放弃原有代码，趁着被微软公司收购的时候从头来过。今天，就让我来讲一个故事，告诉你还有另一种选择。

6年前FogBugz诞生时，它只是我自学ASP编程的一个练习。很快，它就变成了我们在公司内部使用的软件故障追踪系统。几乎每天，我们都把自己需要的功能加入其中，直到它变得非常完善，所有我们想要的功能都已经加进去了为止。

许多朋友问我，能不能把FogBugz复制一份给他们的公司使用。这很麻烦，因为太多的代码都是针对我们公司的情况编写的，不具备通用性，它只部署在我们那台开发用的电脑上，如果要让它在其他电脑上跑起来，将是一项很痛苦的工作。我大量使用了SQL Server的存储过程（stored procedure），这意味着运行FogBugz就必须配备SQL Server。对于两三个人的小团队来说，购买SQL Server的开销太大了，而且没必要用这么高级的数据库软件。我就告诉我的朋友们："行啊，不过要付给我5000美元的劳务费，让我用几天来

修改代码，把数据库从SQL Server改成Access[①]，这样它就能在你们的机器上运行了。”一般来说，我的朋友们都觉得我的开价太高了。

这样的事情发生了好几次以后，我就得到了一点启示：如果我把同样的程序卖给好几个用户，比如3个用户，我只要每人收取2000美元，就可以获利了。如果卖给30个用户，每人只要收200美元就行了。卖软件就是这么简单的一件事。所以在2000年底，Michael[②]就放下手头的工作，将程序的数据库接口改成可以选择使用Access或SQL Server，把所有与特定使用场合相关的东西都放进了一个头文件。我们就开始销售这个软件了。我对它真没有太大的期望，不觉得会得到很多回报。

那时，我的想法是，天啊，世界上有数不清的程序故障追踪软件，每个程序员都写过一个用来除错的小工具，为什么别人会来买我们的产品呢？有一件事我是知道的，那就是程序员在创业的时候，通常有一个坏习惯，认为其他人都是与他一样的人，有与他一样的需求。所以程序员创业就会出现一个不健康的倾向，即喜欢出售编程工具。这就是为什么你会看到，世界上有那么多可怜兮兮的小公司专门兜售奇奇怪怪的源码生成工具、奇奇怪怪的错误捕获和通知工具、奇奇怪怪的除错工具、奇奇怪怪的代码分色显示编辑器、奇奇怪怪的FTP工具，嗯，以及故障追踪系统工具包。所有这一类的玩意，只有程序员才会喜欢。我才不要掉进这个陷阱呢！

当然，事情的发展与我的预料完全不同。FogBugz很畅销，真地很畅销。它在Fog Creek的收入中占据了很大的一块，而且销售额还在平稳增长中。看来这个软件的市场不错。

于是，我们就开发了2.0版，主要加入了一些最明显需要加入的功能。这次的开发人员换成了David，他和我们其他人一样，都确实认为不值得在FogBugz上面投入太多的精力。所以，他编写代码的方式往往是一种可能被别人称之为“权宜之计”的方式，而不是一种牢靠讲究的方式。毫不奇怪，原始代码中的设计漏洞也被听之任之。软件故障代码的编辑页面居然有两种生成方式，而且还是各自使用两套基本相同的代码。SQL语句散布在HTML

① Access是微软公司的桌面级数据库产品，属于Office套装的一部分，它的使用接口包括在Windows操作系统中，可以免费使用。SQL Server则是企业级数据库，必须向微软购买昂贵的许可证后才能使用。

② Michael H. Pryor是Fog Creek的共同创始人。

代码中，这一句那一句，前一句后一句，到处都是。我们的HTML代码写得也很糟糕，不是按照规范编写，而是针对一些老式的浏览器而编写，那些浏览器本身就漏洞百出，载入空白页面也会发生崩溃。

是啊，那时候，FogBugz工作起来确实很出色，有一段时间，我们已知的软件错误数为零。但是软件内部的代码，用技术术语说，就是“混乱无序的”，根本无法解读。再想添加新功能就会像得痔疮那样痛苦。如果想为故障追踪的主表增加一栏，也许需要修改50个地方。即使时间已经过去了很久，连你们家购买飞行车去火星上的海滩度周末都已经是多年以前的事了，你还是会不断发现代码中有些地方你以前忘记了修改。

如果一家比较差的公司遇到了这种情况，并且它的经理可能是从快递公司挖过来的，那么这种公司也许就会决定放弃源码，推倒重来。

我有没有提过我不相信推倒重来？我猜我大概已经说了许多次了吧。

总之，我就是不想推倒重来，我决定拿出生命中的3个星期，彻底把代码整理一遍。揉一揉，搓一搓[①]。根据代码重构（refactoring）的精神，我为本次练习制定了以下规则。

(1) 不添加任何新功能。

(2) 无论何时向源码库提交代码，都要保证程序依然能够完善地运行。

(3) 我所要做的只是一些合乎逻辑的变换（logical transformation），几乎都是机械性的，而且能够立刻确定不会改变代码行为。

我审核了每一个源文件，一个一个地看，从头到尾研究代码，思考怎样才能更好地组织这些代码，然后用尽量简单的方式修改代码。下面就是我在这3个星期中做的一些事情。

- 将所有的HTML代码修改成XHTML。比如，将
改为
，所有的属性都用引号括起来，所有嵌套的标签都一一对应，所有网页都通过了校验。

① “揉一揉，搓一搓”（Rub a dub dub）是一首英语童谣，内容为“Rub-a-dub-dub,/ Three men in a tub, / And how do you think they got there?// The butcher, the baker, / The candlestick-maker, /They all jumped out of a rotten potato, / I was enough to make a man stare.//”（揉一揉，搓一搓/三个男人，一个澡盆/为啥会这样？//杀猪匠，银行家/再加上蜡烛商/都从烂土豆中蹦出来/可是够稀奇。//）

- 移走了所有的格式标签（<FONT>等），将格式都放入一个CSS样式表。
- 从表现层代码中移走了所有SQL语句和所有程序逻辑（市场营销人员喜欢把它们叫做商业规则）。这些东西都被写进类。不过，我没有好好设计类。每当我发现有必要为类增加一个方法时，我就应付一下，把这个方法加上去。（在世界的某个角落，某人正在摆弄一大堆分类标签的粘纸。他削尖了铅笔，恨不得把我的眼睛挖出来。你说你没有好好设计类，这是什么意思？）
- 如果发现有重复的代码块，我就将其改写为类、函数或者方法，以便简化代码。我还把大型的函数分解成几个小型的函数。
- 将所有的英语说明文字都从代码中移走，单独放入一个文件，以便建立多国语言支持。
- 重构ASP站点，做到全站只有一个入口，而不是可以从多个文件进入。这使得以前很难做到的事情现在变得容易了。举例来说，如果你错误地填写了表单，现在我们就可以将错误信息显示在你填写出错的那个表单的那一栏。如果你正确地编写代码，那么这种事情本来是不难的，但是我以前没有正确地编写代码，因为那时我只是刚刚开始学习ASP编程。

经过了这3个星期，内部代码变得越来越理想了。但是，对于最终用户来说，表面的区别并不是很大。只不过由于有了CSS样式表，某些字体更小巧秀丽了。其实我任何时候都可以停下来，因为不管何时，我的代码总是100%可以运行。（而且每次我向源码库提交代码的时候，我都把代码上传到公司内部正在运行FogBugz的服务器上，检验我的修改是否正确。）事实上，在整个过程中，我从不曾有过任何的苦思冥想，也从不曾被迫做出新的设计，因为我所做的一切都是很简单的、合乎逻辑的变换。偶尔，我会遇到一些奇怪的代码块。它们通常是这些年中为了修改错误而加入的代码。幸运的是，我可以把它们完整保留下来。我认为，在一般情况下，如果将所有代码推倒重来，我会犯下和以前同样的错误，可能长年累月都不会注意到。

基本上，我现在完工了。和计划的一样，这件事花了我3个星期。现在，几乎每一行代码都不一样了。不是吹牛，我读过每一行代码，修改了其中大部分的行。整个程序的结构已经完全不同了。所有的错误追踪功能与HTML的用户界面已经完全分离了。

下面是我采用的代码整理方法的所有优点。

- 它的耗时大大少于一次彻底的重写。假设一次彻底的重写需要耗时一年（依据是我们到目前为止开发FogBugz所需要的时间），那么，这就意味着我节省了49个工作周。这些被省下来的时间来自于那些我没有改动的代码，它们包含了很多程序设计上的思考，我没有改动它们，就省去了这些思考。我从不曾被迫停下来想一想，“哦，我需要在这里新增一行。”我只是随手将
改成
，然后再看下一行。我不必考虑文件上传功能的多个部分如何协同工作。这些代码本来就能运行。我所做的就是把它们整理一下。
- 我没有引入任何新的错误。当然，不能保证一点小错误都没有。但是从头到尾，我都没有做过那些会引发错误的事情。
- 在任何时间，如果有必要，我都可以停下来，把程序交出去。
- 工作进度是完全可以预测的。最初的一个星期过去以后，你就能够准确算出每小时中你整理的代码行数，然后对剩下的工作量就可以做一个相当可靠的估计。Mozilla基金会的人们，建议你们试试这种方法[①]。
- 向现在的代码中加入新功能实施起来容易多了。一旦我们开始部署下一个重大的新功能，节约下来的时间很可能就能将修改代码所用的3个星期赚回来。

关于代码重构的许多文献都与Martin Fowler[②]有关，当然不可否认的是，清理代码的主要原则多年前在程序员中就广为人知。目前有一个有趣的新领域，那就是代码重构所使用的工具——“重构工具”，这个新奇的词语指的是那些能够自动清理代码的程序。我们需要优秀的工具，但是距离实现这个目标还有很长的路要走。在大多数的开发环境中，你甚至无法完成一个简单的变换，比如为一个变量改名字（难点在于所有引用这个变量的地方也要自动把变量名改掉）。但是，重构工具正在不断进步中，如果你也想加入，开办一家可怜兮兮的小公司，专门出售奇奇怪怪的编程工具，或者想为开源运动做出有用的贡献，那么这个领域是一片广阔天地，大有可为。

① Mozilla基金会的主要工作是开发Firefox浏览器。这个项目的原始构想是在Netscape Navigator源码的基础上开发Firefox。但是后来，Mozilla基金会认为Netscape Navigator源码过于复杂，因此就抛弃了它，从头开始重写Firefox的源码。

② Martin Fowler所著的《重构：改善既有代码的设计（英文注释版）》已由人民邮电出版社图灵公司出版，该书中文版也即将推出，敬请期待。——编者注

程序员最可信赖的求职帮手

31

组织beta测试的十二个最高秘诀

2004年3月2日，星期二

下面是一些关于如何组织一次软件的beta测试的秘诀。需要注意的是，这里所说的“软件”指的是面向大量用户的软件，也就是我所称的“包装盒软件[①]”（shrink-wrap）。这些秘诀对商业项目和开源项目都适用。不管你开发软件的目的是获得报酬，还是获得眼球效应，或是提高在同行中的知名度，都可以参考这些秘诀。但是，我关注的只是有大量用户的软件产品，而不是公司内部的IT项目。

(1) 开放式的beta测试是没用的。要是你那样做，只可能有两种结果。一种结果是你有太多的测试者（就像Netscape那样），这些人向你反馈了大量的意见，你从中根本不可能得到有用的数据。另一种结果是，现有的测试者根本不向你反馈他们的使用情况，导致你无法得到足够的数据。

(2) 要想找到那些能够向你反馈意见的测试者，最好的方法是诉诸他们“言行一致”的心理。你需要让他们自己承诺会向你发送反馈意见，或者更好的方法是，让他们自己申请参加beta测试。一旦他们采取了某些主动行为，比如填写一张申请表，在“我同意尽快发回反馈意见和软件故障报告”的选项上打勾，许多人就会发送反馈意见，因为他们想要“言行一致”。

(3) 不要妄想一次完整的beta测试的所有步骤能够在少于8~10周的时间内完成。我曾经试过，结果是除非老天帮忙，否则根本不可能做到。

① “包装盒软件”指的是在商场中上架销售、有独立包装、外面用热收缩塑料膜密封的软件商品。

(4) 不要妄想在测试中发布新的软件版本的频率能够快于每两周一次。我曾经试过，结果是除非老天帮忙，否则根本不可能有效。

(5) 一次beta测试中计划发布的软件版本不要少于4个。我从来没试过少于4个版本，因为太明显了，那样不可能达到测试目的。

(6) 如果在测试过程中你为软件添加了一个功能，那么哪怕这个功能非常微小，整个8个星期的测试也要回到起点，从头来过，而且你还需要再发布3个或4个新版本。我犯过的最大错误之一就是，在CityDesk 2.0的beta测试接近尾声的时候，我向软件中加入了一些保留空格的代码，这产生了一些意想不到的副作用（如果我们可以这样说），测试的时间不够了，我本应该将测试时间加长、进一步收集数据的。

(7) 即使你有一个申请参加beta测试的步骤，最后也只有五分之一的测试者会向你提交反馈意见。

(8) 我们制定了一条政策，所有向我们提交反馈意见的测试者都将免费获赠一份正版软件。不管你的反馈意见是正面的，还是负面的，只要你提交给我们，就能获得赠品。但是，在测试结束的时候，那些不提交反馈意见的测试者什么也不会得到。

(9) 你需要的严肃测试者（即那些会把反馈意见写成3页纸发送给你的人）的最小数量大约是100人左右。如果你独立开发软件，那么这是你能够处理的反馈意见的最大数量。如果你有一支测试管理团队或专门的beta测试经理，那么设法分别为每个处理反馈意见的人找到100个严肃测试者。

(10) 根据第(7)条，即使你有一个参加beta测试的申请步骤，最后也只有五分之一的测试者会真地使用你的产品并将反馈意见发送给你。那么，假定你有一个质量控制部门，里面一共有3个测试管理人员，这就意味着你必须批准1500份参加beta测试的申请表，因为这样才能产生300个严肃测试者。批准的数量少于这个数目的话，你就不会得到充分的反馈意见；批准的数量多于这个数目的话，你就会被许许多多重复的反馈意见淹没。

(11) 大多数beta测试的参与者只是在第一次拿到这个程序的时候才会去试用一下，然后就丧失了兴趣。此后每次你推出一个新的版本并发送给他们，他们也不会有兴趣重新测试它。除非他们每天都在用这个程序，但是对于大

多数人来说，这是不可能的。因此，你需要错开不同版本的测试对象，将你的所有beta测试参与者分成四组，每次发布一个新版本的时候，就把一个新的组加入测试，这样就能保证每个版本都有第一次使用这个程序的测试者。

(12) 不要混淆技术beta和市场beta。我上面谈的这些都是针对技术beta，它的目标是发现软件中的错误和得到及时的用户反馈意见。市场beta则是软件正式发布前的预览版本，对象主要是新闻媒体、大客户和那些写入门教程的家伙（该教程必须在软件上市的同一天问世）。对于市场beta，你的目的并不是得到反馈意见。（虽然无论你怎么做，那些写书的家伙很可能都会滔滔不绝地告诉你一大堆意见。如果你置之不理，这些意见就会被复制粘贴进他们自己的书里。）

32

建立优质客户服务的七个步骤

2007年2月19日，星期一

Fog Creek是一家自力更生的小公司，在创业的头几年，请不起专门的客服人员，Michael和我只好兼任客服。我们用来帮助客户的时间占用了改进程序的时间。但是，我们从客服经历中学到了许多东西，现在Fog Creek的客户服务已经十分出色了。

如何提供优质的客户服务？下面就是我们学到的7点经验。我用“优质”这个词，是取它的字面意思，因为我们的目标就是要让客户对我们的服务交口称赞[①]。

每件事都有两种做法

几乎所有技术支持方面的问题都有两种解决方法。一种是表面的、快速的解决方法，只求把问题解决了了事。但是只要你深入一点思考，就会发现还有另一种方法，能够防止类似的问题再次发生。

有时，这意味着要为软件或安装程序加入更多的智能判断。目前，我们的FogBugz安装程序就包含了许多针对特殊情况的处理代码。有时，你需要修改错误信息的提示文字，或者最好写一篇文章放进知识库（knowledge

① 作者在这里使用了双关语。原文中，“优质”使用的词是remarkable，这个词的词根remark是“谈论、说话”的意思。所以，作者说“优质”的客户服务就是可以被人们称赞、谈论的客户服务。

base）中。

我们对待每一个需要技术支持的求助电话，就好像NTSB（美国国家运输安全委员会）对待飞机坠毁一样。每当有飞机坠毁，NTSB就派出调查员，搞清楚发生了什么事，拟定新的政策，防止类似的问题再发生。这一套机制真是行之有效，我们看到美国的飞机坠毁事件是极其罕见的，即使发生，也是因为非常特殊的情况而导致，并且同样原因的事故不会再发生第二次。

这样做有两个含义。

第一个含义。技术支持团队必须能够与开发团队直接沟通，这很关键。这意味着你不能把技术支持人员外包，他们必须与开发人员在同一个地址办公，必须有途径让问题得到彻底解决。许多软件公司直到今天还相信，将技术支持服务搬到印度的班加罗尔和菲律宾，或者整个外包给另一家公司，是一种很“经济”的选择。没错，解决单个问题的成本也许可以从50美元降到10美元，但是同一种类型的问题会不断出现，使得你不得不一次又一次地付出10美元。

如果我们把一个技术支持问题交给一个纽约的合格程序员来处理，那么很可能这就是我们最后一次遇到这种问题。所以，我们一次性为解决这个问题支付了50美元，然后就把这一类的问题都永久消除了。

某种程度上，电话公司、有线电视公司和ISP并不理解这种关系。它们把技术支持服务外包给报价最便宜的提供商，一次又一次地支付10美元，一次又一次地解决同样的问题，而不是一劳永逸地通过修改源代码彻底解决这个问题。那些廉价的电话呼叫中心根本没有办法使问题得到永久性解决。而且说实话，他们也没有动机让问题得到永久性解决，因为他们的收入依赖于来电量，问题一次次重复发生，他们就能不断获得收入。没有什么比能够一次次用同样的答案回答同样的问题更让他们感到欢欣鼓舞了。

第二个含义。如果每次发生问题，你就寻找方法，永久性地解决它，那么长此以往，会发生什么结果？那就是所有常见的和容易的问题都被解决了，留下来的都是一些非常罕见的和奇特的问题。这是很好的结果，因为问题越罕见，就代表着你遇到它们的机会越小，所以你就能把常规性技术支持的费用节省下来。不利的一面是，由于常规性技术支持能解决的问题都解决了，所以以后要么不遇到问题，要么一遇到就必须对程序进行重大的调试和

修改。你培训新的技术支持人员时，就不能只教给他们10种常见问题的解决方法，而必须教会他们调试程序。

“永久性解决问题”的信念使得我们取得了很好的回报。我们的销售额增长了10倍，提供技术支持的成本只增长了一倍。

建议吹掉灰尘

微软公司的Raymond Chen讲过一个小故事（blogs.msdn.com/oldnewthing/archive/2004/03/03/83244.aspx）。有一次，客户打电话来抱怨键盘失灵，结果不出所料，原因是键盘没插好。但是，如果你直接问客户键盘有没有插好，他们会觉得受到了侮辱，怒气冲冲地回答：“它当然插好了！难道我看上去像一个笨蛋吗？”而实际上，他们并没有检查过接口。

“你可以换一种说法，” Raymond Chen建议，“改成这样说：‘别着急，有时候键盘接口会有灰尘，导致接触不良。你能不能拔掉键盘插头，吹掉上面的灰尘，然后再把它插回去？’”

“然后，客户爬到桌子底下，发现自己忘了插好插头（或者把插头插入错误的接口）。他们把灰尘吹掉，插入插头，回复你说：‘嗯，很好，问题解决了，谢谢。’”

很多时候，我们要求客户去检查某样东西，都可以这样表达。不是直接要求他们去检查某个设置，而是告诉他们先改变这个设置，然后再改回去，目的是“确保这个设置被保存进了软件”。

让客户迷上你

每次我们需要购买印有公司logo的纪念衫时，我们就去找服饰经销商Lands' End公司。

为什么呢？

让我告诉你一个故事。有一次，为了参加一个交易会，我们需要一些T恤衫。我给Lands' End打电话，订购了两打。在此之前我们已经从他们那里订购了一批背包，所以就同他们说好，T恤衫使用的logo设计与用在背包上的相同。

T恤衫送到后，我们沮丧地发现，logo看不清楚。

原来，背包的底色比T恤衫的更明亮。所以，棉线的颜色在背包上很醒目，但是在T恤衫上就显得太暗，辨别不出来。

我打电话给Lands' End。与往常一样，电话铃还没响，有人就接起了电话。我很肯定，他们有一个系统，随时都有一个客服人员在待命。因此，顾客打进电话以后，根本不需要等待，电话铃响起之前就有客服人员接起电话，与顾客交谈。

我在电话里解释，我把事情搞糟了。

他们说："别担心。你可以全额退货。我们会用一种不同颜色的线重新为你制作T恤衫。"

我说："交易会两天后就要开了。"

他们说，他们会把新的T恤衫用FedEx快递过来，我明天就能收到。至于前一批T恤衫，我可以在我方便的时候退还给他们。

他们付了两次运费，我一分钱没出。一大堆印有Fog Creek的logo并且字迹模糊的T恤衫对他们不可能有用处，但是即使这样，他们还是同意承担成本。

如今，我只要遇到需要制作宣传纪念品的人，就会把这个故事讲一遍。事实上，每次我们谈到电话应答系统或者客户服务的时候，我都会讲这个故事。Lands' End通过提供优质的客户服务让我记住了它，会主动谈到它，为它做宣传。

客户遇到问题，你帮他解决了，客户实际上变得比没有问题时还要满意。

这肯定与期望值有关。大多数人对于技术支持和客户服务的期望值来自于他们同航空公司、电话公司、有线电视公司、ISP打交道的经验。总体上

看，那些公司的客户服务都很糟糕，糟到从今以后，你都不想自寻烦恼，再给它们打电话，我说的对不对？但是，你给Fog Creek打电话，就会得到完全不一样的待遇。你的电话打进来以后，立刻就有人接听，迎接你的根本不是语音留言或者自助式电话菜单，而且接电话的人非常耐心和友好，确实能够解决你的问题。两相一对比，你往往就会给予我们很高的评价，而且比起那些没有给我们打过电话、对我们持中性看法的客户，你对我们的评价更高。

不过，我还没有变得那么激进，故意制造一些问题迫使客户打电话给我们，以便我们有机会向他们证明我们的客户服务是多么卓越。要知道，许多客户遇到问题的时候根本不打电话，他们就是在那里生闷气。

记住，要是有客户打来电话，你要把这当作一个千载难逢的机会，一个可以培养出死心塌地的忠实客户的机会。如果你做得好，客户逢人就会唠叨你的服务是多么出色。

承受责备

有一天早上，我想为自己的公寓再配一把钥匙。上班的路上，我就顺路去找了街角的锁匠。

我在纽约公寓已经住了13年，早就学会了绝不要对锁匠有信心。他们复制的钥匙有一半不能用。所以，我特意回家试验新钥匙，你瞧，果然不能用。

我带着钥匙回来找锁匠。

他又做了一把。

我再次回家，试验新钥匙。

还是不能用。

于是我发火了，绷紧了脸。距离上班时间已经过去了一个半小时，可是我还要第三次去找锁匠。我很想算了，不去找他了，以后找其他锁匠试试，但是最后决定再给这个废物一次机会。

我冲进店里，准备把我的怒火都释放出来。

“还不行？”锁匠问道，“让我看看。”

他看着钥匙。

我气呼呼地琢磨，怎样才能最好地把我的愤怒表达出来，发泄我对整个早晨都被迫来来回回地做无用功的不满。

“啊，这是我的错。”他说。

突然间，我的气就全消了。

真是神秘啊，简简单单的一句“这是我的错”就能让我怒气全消。只用这一句话就够了。

他第三次做好了钥匙。我不再生气。钥匙也能用了。

我的感想就是，真不敢相信，我在这个世界上活了40年，居然才发现“这是我的错”这五个字有这么大的威力，能够在数秒之内改变我的情绪。

纽约的大多数锁匠都是另一种类型，不愿意承认自己出错了。要他们开口说出“这是我的错”比登天还难。但是不管怎样，我遇到的这个锁匠就做到了。

学会说软话

我心想，好吧，既然早上的时间已经泡汤了，那么我就找个小馆子吃点早饭吧。

那是一家纽约典型的小馆子，同电视剧*Seinfeld*中的差不多。菜谱有30页，厨房却同电话亭的大小差不多。这实在是不可思议。店家一定有*Star Trek*里的技术，才能把所有的原料都塞进这么小的一个厨房中。也许他们在里面改变了原料的原子排列吧。

我坐在收银机的旁边。

一个老年妇女走过来结账。她一边付钱，一边同店主说："我来这里吃饭已经好多年了。服务员对待我的态度真是很粗鲁。"

店主火冒三丈。

"你什么意思？这是不可能的！他招待客人的态度很好！我从来没有收到过投诉！"

那个顾客简直不敢相信自己的耳朵。她这样的忠实顾客，想要给店主帮忙，让他知道有一个服务员需要改善一点礼貌，而店主竟然为此同她吵架！

"好吧，没关系。我来这里许多年了，所有人都对我很周到，只有那个人对我很粗鲁。"她耐心地解释道。

"我不关心你是不是天天来这里。我的服务员根本不粗鲁，"店主继续对着她吼，"我们这里从来就没遇到过麻烦。你为什么要制造麻烦？"

"要是你这样同我说话，我就不会再来你们这里了。"

"我不在乎！"店主说。把小餐馆开在纽约有很多好处，其中之一就是城里的人太多了，即使你得罪了每一个曾经走进你店里的客人，你依然还会拥有大量的顾客。"你不要再来了！我不要你这样的顾客！"

干得漂亮，我心想。一个六十多岁的店主，击败了一个小老太太，取得了一场巨大的精神胜利。你是不是很自豪啊？你是不是觉得自己很像男子汉大丈夫？精神胜利让你感觉很好吗？你是不是不赶走忠实顾客不死心啊？

难道说一句"真抱歉，我会跟他说的"，就会让你感觉自己像窝囊废吗？

客户投诉的时候，你很容易就变得感情用事。

解决方法是记住一些重要句子，不断练习把它们说出口。等到你需要用到它们的场合，你就会把雄性激素自动抛在了脑后，转而让顾客高兴。

"对不起，这是我的错。"

"对不起，我不能收你的钱。这一顿饭算我们的。"

"真是糟糕，请告诉我事情是怎么发生的，我要确保不会再有类似事件。"

不习惯把“这是我的错”这句话说出口是很自然的。人的天性就是这样。但是，这几个字能够使得发怒的顾客变得很高兴。所以，你必须学会这样说，必须让你嘴里发出来的声音听起来就像是你真这样想一样。

这就开始练习吧。

每天早上淋浴的时候，把“这是我的错”重复一百遍，直到你忘了它的意思，只记住了这一连串音节为止，这样你就能在需要的时候张口就来了。

补充一点。你可能有一种想法，认为绝不能承认过失，否则会吃上官司。这纯粹是无稽之谈。避免被别人起诉的方法就是不要让他人对你火冒三丈。要做到这一点，最好方法就是承认错误，并且把该死的问题都解决掉。

学会做木偶

很显然，那个大发雷霆的餐馆店主将顾客的意见当成了人身攻击，这与那个锁匠的处理方式截然相反。确实，当一个生气的顾客面对着你不停地抱怨或者发泄时，你很容易产生对立心理。

你永远不可能赢得这些争执。如果你把它们当作人身攻击，结果更是会糟糕100万倍。就是在头脑发热的时候，你会像那个店主一样说出“我不要你这种混蛋客人”。这种话说起来很爽，很让人振奋，让你有一种艰苦战斗后的胜利感。哇，难道不爽吗？你只是一个小店主，却能够一脚踢掉自己的顾客。真是太爽了。

实际结果是，这样做对你的生意一点好处也没有，对你的心理健康也不利。你把顾客踢出门，觉得自己赢得了胜利，但是你的心情依然焦躁和愤怒。而顾客会要求信用卡公司无论如何都要停止把他们的钱付给你，并且顾客还把他们的不满告诉许多朋友。就像Patrick McKenzie所写：“同顾客争吵，你永远不会是赢家。”（kalzumeus.com/2007/02/16/how-to-deal-with-abusive-customers/）。

面对愤怒的顾客，你只有一条路可选。你必须明白，顾客不是因为你这个人而生气，而是因为你的企业而生气，你只不过碰巧是代表企业的一个活

靶子而已。

既然他们只把你当作一个木偶，当作一个代表真实企业的活靶子，你就也需要把自己当作一个木偶。

在顾客面前，你要假装成一个木偶，真正的你则是躲在后面操纵木偶的人。顾客对着木偶喊，不是对着你喊。他们只是在对木偶发脾气。

你的任务就是想出办法，“啊，我能不能让这个木偶开口说一些话，使得对面那个人变成一个开心的顾客？”

你的真实身份是操纵木偶的人，而不是争吵中的一方。当顾客说“你们这伙人的脑子是不是有毛病”时，你就当作他在演戏，所以你也要演戏。“对不起，这是我的错。”总之，你要找到办法，让木偶的举动能使得顾客开心，而不要把所有一切都当作针对你个人的攻击。

贪婪让你一无所获

最近，我与去年负责大部分Fog Creek客服工作的人员交谈。我问他们，对付愤怒的顾客什么方法最有效。

“坦白说，”他们回答，“我们的顾客脾气都很好，我们真还没有遇到过愤怒的顾客。”

好吧，我们的顾客确实脾气很好，但是接听一年的电话居然没有遇到任何一个生气的人，这未免也太不寻常了。我一直觉得，在电话服务中心工作，本质上就是整天与愤怒的人打交道。

“没有啦，我们的顾客都是好脾气。”

下面就是我的解释。为什么我们的顾客脾气好，我觉得原因是他们不担心。为什么他们不担心，原因是我们的退货政策慷慨得离谱，“如果你使用我们的产品，不觉得异常欣喜，我们就不要你的钱。”

顾客知道他们没有什么可担心的。在交易关系中，他们是握有权力的一方，所以他们不会被粗暴对待。

90天无条件退款保证是我们在Fog Creek做过的最佳决策之一。你可以试试下面的做法。使用我们的Copilot[1]服务整整24个小时，然后3个月以后，你打电话给我们："嗨，哥们，我想买杯热咖啡，但是缺少5美元。请把3个月前我付的Copilot一日通（day pass）使用费退还给我。"我们就会把钱退还给你。即使你在第91天或第92天或第203天打电话来，依然会得到我们的退款。如果你感到不满意，我们真不会要你的钱。Joel on Software的网站上有发布招聘广告的服务。如果你付了钱，却没招到人，你就能从我们这里把钱拿回去。我很肯定，市场上只有我们一家这样做。这是前所未闻的，但是这也意味着我们发布的广告比别人多得多，因为顾客不会有任何损失。

在过去6年左右的时间中，因为我们接受顾客的退货而导致的成本只占总成本的2%。

对，就是2%。

而且你知道吗？大多数顾客使用信用卡支付，如果我们不退款，很多人就会给银行打电话。这叫做"退单"（chargeback），也就是投诉我们有欺诈行为，使他蒙受了损失，从而要求赔偿。最后，他们还是能拿到钱，而我们却不得不支付一笔"退单"费。如果这种事情发生得太频繁，我们的管理费用就会上升。

想知道发生退单的交易占我们所有交易的比例吗？

0%。

我没有开玩笑。

如果我们把退款政策制订得很严格和苛刻，那么我们唯一可能得到的结果就是把一些顾客惹火。他们会在自己的网志上大喊大叫，发泄一通。从此，我们休想在他们身上再拿到一分钱。

我知道，有些软件公司在网站上写得很清楚，任何情况下，你都无权向他们要求退款。但实际上，如果你给他们打电话，他们最后还是会把钱退还给你的，因为他们知道如果不这样做，你的信用卡公司也会把钱从他们的账上划走。这真是两头吃亏。一方面，不管怎样，钱最后都退给了顾客；另一

① Copilot是Fog Creek公司推出的一种远程桌面登录软件，具体介绍参见www.copilot.com。

方面，因为网站上写着不退款，这使得潜在的顾客无法产生一种温馨的感觉，反而会觉得不踏实不放心，所以他们在购买之前就会犹豫再三。很可能，他们因此就彻底打消了购买你的软件的念头。

（再附送一条）为客服人员提供职业发展道路

我们在Fog Creek学到的最后一条重要的经验就是，你需要非常称职的人与顾客交谈。Fog Creek的销售人员需要对软件开发流程有丰富的经验，需要能够解释为什么FogBugz采用现在的运行方式，解释为什么FogBugz能使得软件开发团队更有效地运作。Fog Creek的技术支持人员不能只会用标准答案回答一些常见问题，因为通过改进软件，我们已经把常见问题都消灭了，所以技术支持工作实际上变成了处理疑难杂症，这经常意味着对程序进行调试和除错。

许多合格的人觉得前台的客户服务工作很无聊，我也同意这一点。作为补偿，我雇用这些岗位的人员时就为他们提供了一条明确的职业发展道路。在Fog Creek软件公司，客户服务工作只是3年期的管理培训项目的第一年，这个项目中包括公司出钱让他们去攻读哥伦比亚大学技术管理的硕士学位。这样的安排使得我们可以招聘到有抱负的、聪明的技术人才。他们与顾客进行交谈，帮助顾客解决问题，与此同时自己还有一条很棒的职业发展道路。我们最终支付的薪水确实要比市场上这些岗位的平均薪水水平高一些（还要考虑每年2.5万美元的学费），但是他们也为我们创造出了比一般水平多得多的价值。

第八部分

发布软件

33 挑选发布日期
34 软件定价

33

挑选发布日期

2002年4月9日，星期二

制订详细的进度表有很多理由，其中最好的理由之一就是，它给了你一个取消软件功能的借口。如果软件准时发布与部署《跟着Bob唱歌[1]》的MP3跟唱功能不可兼得，那么你很容易做出选择，把跟唱功能砍掉，反正Bob也不会生气。

所以，我有几条软件开发周期的基本规则。

(1) 确定发布日期，这个日期可以根据客观情况也可以根据主观愿望进行选择。

(2) 列出软件要实现的功能，然后按照优先顺序排序。

(3) 每当你落后于预定进程时，就把排在最后的功能砍掉。

如果上面的每一步你都做到了，那么你很快就会发现，那些被你砍掉的功能根本不会让你感到后悔。那些功能通常都不是很实用。如果它们确实很重要，那么你可以在开发下一版的时候把它们加进去。这有点像编辑文章。如果你要写一篇750字的杰作，你可以先写出1500字，然后再编辑。

顺带说一下，如果你忘了按照优先顺序部署功能，你就会把一切搞糟。只要你一不注意，你手下的程序员就会按照趣味性的顺序开发各种功能，你就将因此无法按时发布软件，或者无法取消部署某个功能，因为所有的程序员都忙于写卡拉OK的混音功能，也不管菜单选择系统还不能用。你就会陷

① 《跟着Bob唱歌》（Bob's Sing-Along）是美国2000年发行的一张CD。Bob是电视系列节目《芝麻街》中的人物，这张CD就收录了Bob在该节目中演唱过的歌曲。

入很被动的局面，离你预定的软件发布日期已经过了6个月，可是你能拿出来的只是一大堆该死的“复活节彩蛋[①]”，软件真正的功能却一点都没有实现。

所以，很显然，问题就变成了你怎么来挑选一个发布日期？

不难想到，有时候软件发布日期是定死的，外部限制使得你没有其他选择。某一天，证券市场的报价从分数改成了小数[②]，如果到时你没有准备好新软件，你的公司就会被迫退出这个领域，你本人也会被拉到码头后面，脑门上挨一枪。或者换一种情形，Linux的内核不久后就要推出新版本，其中部署了另一套全新的数据包过滤（packet filtering）系统，你的所有用户都会使用它，而你现有的应用程序无法在它上面运行。OK，对于这些用户，你的发布日期无疑是定死的。你现在就可以停下来了，不用再往下读这篇文章了，回去为你的亲爱的用户好好做一顿大餐吧。

再见，走好！

但是，对于其他情况，我们这些剩下来的人怎么来挑选发布日期呢？

你可以考虑采用三种方法。

(1) 经常发布稍作改进的版本。这属于极限编程中的方法，对于客户数量较少的小团队开发项目（比如公司内部的IT项目）最适用。

(2) 每12到18个月发布一次。上架销售的商业软件和桌面应用程序等就是使用这种方法的典型例子。它适用于有大型开发团队和成千上万名顾客的软件。

(3) 每3年到5年发布一次。这种方法常见于超级庞大的软件系统和平台，它们本身就是一个个包罗万象的世界。操作系统、.NET框架、Oracle数据库以及某种意义上的Mozilla都属于这种软件。它们各自的开发人员可以达到几千名（Visual Studio .NET的安装程序就有50个程序员在开发）。它们与其他的软件之间存在极其复杂的互动关系，必须要妥善处理。

① 在软件业中，“复活节彩蛋”（easter egg）指的是制作者隐藏在软件中的好玩的信息或功能。

② 在历史上，美国股市的报价一直采用分数方式（比如1/2美元、1/4美元、1/8美元、1/16美元），已有200多年的历史。据说这一习惯源于西班牙商人，因为当时他们的货币实行八进制。美国证券交易委员会2000年6月13日发布命令，要求美国主要的股票市场在当年9月5日之前开始以10进位方式交易部分股票，在2001年4月9日之前完成向小数报价制的过渡。

当你决定软件的发布频率时，下面是一些你需要考虑的因素。

较短的发布周期会较快地得到顾客的反馈。有时候，与顾客配合的最好方式就是给他们看源码，让他们试用，并且把反馈立即体现在第二天你交给他们的新版本中。这样做的优点是，你不会浪费一年时间开发出一个非常复杂的系统，里面的功能却没有人要用。因为你忙于实现顾客提出的最新要求，这确保了你的软件是符合顾客需要的。如果你的顾客人数较少，那么你最好经常性地发布小幅修改的新版本。只要你在软件中实现了某种有用的改进，哪怕只改进了一个地方，你就应该发布一个升级程序。

好几年前，有一项任务派到我头上，要我为MTV频道的网站做一个内容管理系统。具体的要求包括数据库作为后端，前端用模板动态生成，并且还要做一个完整的工作流系统（workflow system），使得MTV频道遍布全国各个高校的特约通讯员能够在网页上提交本校俱乐部活动、唱片店销量、电台节目、音乐会信息等。“你们现在的网站是用什么做的？”我问。

“哦，我们就是用BBEdit[①]一张张手工做，”他们告诉我，“当然，要做几千张页面，但是BBEdit的多文件查找和替换功能很好用……”

我估算全部工作要用6个月完成。“不过，我建议还有另外一个选择可供考虑。让我们先把模板系统做完，我3个月就可以完成，交给你们。这样一来，马上就可以省掉你们手工编辑网页的巨大工作量。一旦新的系统开始工作，我们就立刻着手部署工作流的部分。在最后完工之前，你们一直可以用电子邮件代替工作流系统进行信息交换。”

他们表示同意。我的想法听起来好像是一个很不错的点子。猜猜真正实施以后得到了什么结果？等到我把模板功能做完交给他们以后，他们觉得不是真地那么需要工作流的部分。而且，事实证明模板功能对于那些不涉及工作流的网站非常有用。所以，我们就再也没有去开发工作流的部分，这就省下了3个月的时间。我把这些时间用来继续增强模板功能，这个功能被证明是最有用的。

某些类型的用户不喜欢像这样当“小白鼠”。一般来说，那些购买现成

① BBEdit是Mac平台上的一款HTML网页编辑软件，官方网站是http://www.barebones.com/products/bbedit/。

软件的人不会想要成为软件开发实验的一部分，他们只想要能够满足他们需要的东西。作为一个顾客，你提出要求，想要某个功能，然后这个功能很快就被开发出来，这无疑让人很满意，但是有且只有一种情况会让你更满意，那就是不用你提要求就能立刻得到自己想要的功能，因为那些功能已经被精心设计在软件中，有着很全面的良好的易用性，并且在公开发售之前已经经过了beta测试。如果你已经有了（或者想要有）大量的付费用户，那么你最好不要太频繁地发布新版本。

假定你发布了一个功能不足、没有特色的商业程序，你这样做的目的只是为了把这个程序扔到世人面前，让你能够“开始倾听顾客的意见”，那么你将会听到顾客这样说：“它的用处不大。”听到这话，你也许心想，OK，没问题，这只是软件的1.0版。但是等到4个月后，你发布2.0版的时候，所有的顾客都会想：“不就是那个废物软件吗？凭什么每过4个月，我就该测试它一次呢？我有必要知道这个软件有什么变化吗？！”事实上，就算过了5年，用户也依然记得对1.0版的印象，要让他们重新评估、产生新的印象，几乎是不可能的事。你不妨回忆一下可怜的Marimba[①]。他们创建公司的时候，正逢对Java的吹捧达到登峰造极的时候，于是他们有了无穷的风险投资资金的支持，用于引诱Sun公司Java开发团队中主要成员跳槽。他们的CEO是Kim Polese，此人非常善于搞好公共关系。她推广Java的时候，就拉来Danny Hillis做演讲，鼓吹Java语言是人类进化的下一个阶段。George Gilder[②]写了许多篇让人目瞪口呆的文章，讲解Java将如何彻底地改变人类文明的本质。比如，我们要深信不疑，与Java语言相比，一神论在人类历史中只是一个很小的亮点。Kim Polese的工作是那样出色，所以，等到Marimba公司发布Castanet软件的时候，它不费吹灰之力就得到了可能比史上任何产品都多的炒作，但是Marimba的程序员用来开发它的时间总共只有……4个月。我们所有人都下载了这个软件，结果发现——它只是一个用来下载软件的列表框而已。（只有4个月的开发时间，你还能期待什么？）太让人震惊了。到处都弥漫着浓厚的失望，厚到你可以把它们用刀切好，再涂上黄油。结果就是，即使过了6年，如果你随便找一个人问什么是Castanet，他们都会告诉你，那只是一个下载软件的列表框而已。几乎没人愿意对它进行重新评估。这6年来，

① Marimba是一款网络管理软件，2004年3月31日被BMC软件公司以2.39亿美元的价格收购。

② George Gilder是美国技术类畅销书作家。

Marimba一直在改进代码，我肯定它现在已经是世界上最酷的东西了，但是老实说，谁会花时间去发现这一点呢？让我告诉你一个小秘密，我们自己的产品CityDesk的推广策略，就是在2.0版发布之前，避免做大规模的宣传。2.0版才是那个我们想要让地球上每个人留下第一印象的版本。与此同时，我们将静悄悄地、小规模地进行营销活动，每一个找到这个软件的人都会惊讶地发现它是一个非常出色的软件，能够解决很多问题，而且历史学家阿诺德·汤因比（Arnold Toynbee）不必改写任何他的著作。

对于大多数的商业软件，在开发流程中，你会发现设计、原型制作、整合、调试、开展完整的一轮alpha测试和beta测试、制作文档等步骤要花掉6到9个月的时间。事实上，如果你计划每年推出一个全新的版本，那么你只有大约相当于3个月的时间用于开发新代码。那些每年都升级的软件通常并不包含足够多的新功能，并不让人感到升级是必要的。（在这方面，Corel公司的Photo-Paint和Intuit公司的QuickBooks堪称典型的反面教材。他们每年都推出一个“重大的”新版本，但是很少值得购买。）结果就是，如今许多人都学乖了，不是每个版本都升级，而是每隔一个版本才升级。你不能让你的顾客养成这种习惯。如果你把进度表的时间放宽，每个版本之间的周期拉长到15或18个月，你用于开发新代码的时间就会从3个月变成6个月，这将使得你的软件的新版本变得更吸引人。

好的，如果15个月是一个合适的时间长度，那么24个月会不会更好呢？也许吧。有些公司是本领域中的领导者，他们使用这样的时间长度可能会获得成功。Photoshop看上去好像就很成功。但是只要感到某个应用软件过时了，人们就会停止购买，因为从那时起，他们每天都在期待新版本的推出。此外，这样长的软件发布间隔会给软件业带来很严重的现金流问题。当然，另一个要考虑的因素是，你的竞争者也许紧跟在你的脚跟后面，苦苦追赶。

对于大型的平台式软件（比如操作系统、编译器、网络浏览器、数据库管理系统），开发过程中最困难的部分就是怎么才能与现存的几千种或几百万种的软件和硬件保持兼容。当一个Windows的新版本出来的时候，你几乎听不到它有向后兼容的问题。微软公司做到这一点的唯一方法就是开展数量多得吓死人的测试，整个测试过程的庞大和复杂，使得建造巴拿马运河就像一个你个人的周末DIY项目一样。Windows的主要版本之间的发布周期通常是3年，期间几乎所有的时间都用于枯燥的整合和测试，而不是用于写出新

功能。短于这个时间长度的发布周期都是不现实的，会把人逼疯的。如果一个操作系统经常发布新版本，而且每个版本都只有很少的改进，那么第三方的软件和硬件开发者就不得不一次又一次地进行测试，他们最后只会有一个反应，就是甩手不干了。对于那些有几百万用户和几百万整合点（integration point）的软件系统，最好偶尔才发布新版本。在这方面，你可以向Apache学习，上一个版本在互联网泡沫出现的初期发布，下一个版本则是在互联网泡沫结束的时候发布。真是完美啊。

如果你已经做了大量的验证（validation）和单元测试，而且编写代码的时候很仔细，那么你每天工作结束后编译出来的版本可能已经质量很高了，随时都可以对外发布。这无疑是值得奋斗的目标。即使你计划中下一个发行的版本是在3年后，但是市场上的竞争也可能风云突变，为了应对竞争者的挑战，你有充分的理由快速发布一个临时版本。当你的妻子就要分娩、等着被送往医院的时候，你还在修理汽车引擎，那就太不妙了。相反，你应该赶紧做出一个新版本，不必等一切工作都准备好了才把软件交付市场。

但是，也不要高估你每天编译高质量版本的重要性，不要高估它能为你带来的帮助。即使你的软件永远不存在任何代码错误，但是如果被迫匆匆忙忙地投入市场，你就不可能发现所有的兼容性问题、Windows 95系统中的特殊错误以及在大字体设置下软件不工作的问题。只有你做过几次beta测试后，才会发现这些问题。而一次完整的beta测试至少需要8周，少于这个时间长度都是不现实的。

最后一个想法。如果你的软件的交付形式是互联网上使用的服务，就像eBay或PayPal那样，那么理论上没有什么可以阻止你经常性发布小幅修改的版本，但是这未必是你的最佳选择。你要记住易用性的基本原则：一个应用程序的行为方法与用户期待的方式一致，那么它就是易用的。如果你每个星期都变一变，那么你的网站就缺乏可预测性，所以它的易用性也不会很高。（别以为在屏幕上加一段文字“注意！用户界面已改变”，你就能避开这一点。没有人会去读它，而且它还让整个页面显得乱糟糟。）从易用性的角度看，更好的做法可能是降低发布的频率，把多处修改一次发布，而不是多次发布。而且，在发布的时候，你要努力使得整个网站的视觉效果发生变化，让网站看上去有点怪怪的，使得用户凭直觉就知道网站发生了重大变化，他们使用的时候要小心一点。

34

软件定价

2004年12月15日，星期三

你刚刚发布了一个相册软件，通过某些途径，你让人们了解到了这个产品。至于具体是哪些途径，作为习题，请本文的读者自行思考。也许你有一个很受欢迎的网志或者其他网络渠道，也许沃尔特·莫斯伯格[①]在《华尔街日报》上为你写了一篇充满溢美之词的评论。

接下来，你面临的最大问题之一就是“怎么来为我的软件定价呢？”你向专家咨询，但是他们看上去也不知道答案。定价是一门很深奥、很难以琢磨的神秘学问。专家告诉你，软件公司犯下的最大错误就是定价太低，那样它们就没有足够的收入，不得不关门歇业。但是，还有更大的错误（对，就是比最大的错误还要大的错误），那就是定价太高，那样你的公司就没有足够的顾客，不得不关门歇业。关门歇业不是好事，因为所有员工都会失业，你也不得不去沃尔玛当一个拉门的，挣最低工资，被迫整天穿着一套脏兮兮的涤纶制服。

所以，如果你喜欢穿全棉的衣服，那么你最好不要定错价格。

找到正确的价格很复杂。我想先从一个小小的经济学理论开始讲起，然后我会把这个理论的各种细节都剖示出来，等到我讲完了，你就会对定价有深入的理解。但是，你依然不会知道你的软件应该定在多少价格，这就是定价这门学问的本质。如果你怕麻烦，不愿意读完本文，那么你就为你的软件

① 沃尔特·莫斯伯格（Walt Mossberg）是《华尔街日报》的著名科技专栏作者，各大科技公司往往会在新产品发布之前就告诉他，请他写评论。

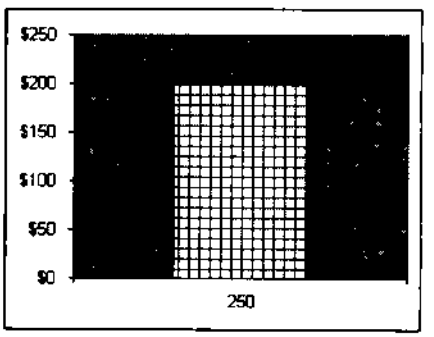
定价0.05美元，除非你的软件用于故障追踪[①]，这种情况下，你就为它定价为3000万美元。

好了，我扯得太远了，书归正传。

一些经济学理论

现在假定你的软件售价是199美元。为什么是199美元？这个嘛，因为我总得有个原始价格。我们很快就会在后文试试其他数字，但是眼下就假定你为软件开价199美元，并且获得了250个顾客。

让我把它画成图：



我画的这张小图的意思是，如果你定价199美元，就有250个人购买你的软件。（正如你所看到的，经济学家是非常奇怪的人，他们喜欢用横轴表示销售数量，用纵轴表示销售价格[②]。如果250个人购买你的软件，那么就必定意味着你的价格定在199美元！）

如果你把价格提高到249美元，结果会怎样？

有些人愿意出199美元购买你的软件，但是他们觉得249美元太贵了，所以不再购买了。

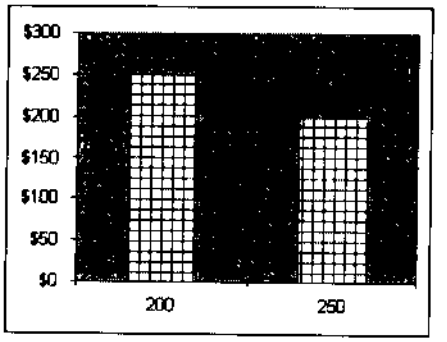
很显然，那些连199美元的价格都不能接受的人，肯定不会以更高的价

① Joel自己的软件FogBugz就是用于软件故障追踪的。

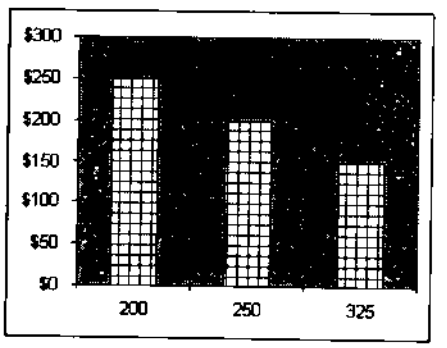
② 在数学中，横轴通常表示自变量，纵轴通常表示应变量。作者这里的意思是，他认为价格是自变量，销售量是应变量，但是经济学的这种作图方式使得看上去销售量好像成了自变量，而价格成了应变量。

格购买。

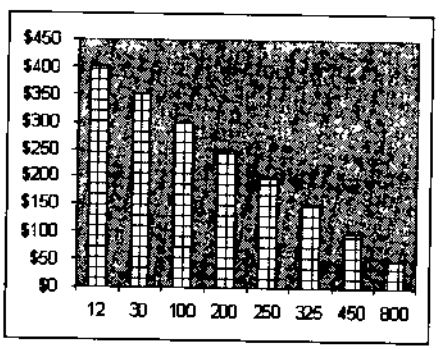
如果在199美元的价格，有250个人购买，那么我们必须假设价格为249美元时，购买者少于250个人。哦，就不妨定为200个人吧：



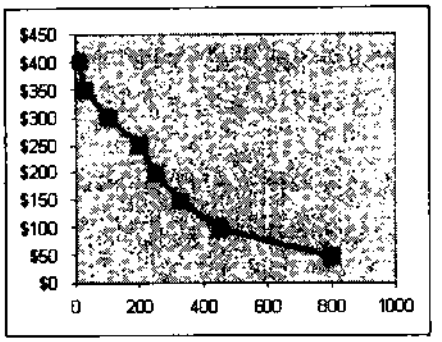
如果定价低于199美元，会怎样？比如说，149美元的价格会有多少人买？那些能够接受199美元价格的人，肯定会在149美元价格时购买，而且可能会有更多的人觉得149美元的价格是可接受的，所以让我们假设价格为149美元时，我们能够销售出去325份软件：



推而广之，就可以得到：



事实上，我们不用在这张图里画出一个个分散的点，而可以用一条完整的曲线，把所有的点都包括在里面。当我这样做的时候，我需要修正一下横轴，让它符合比例变化：



现在，只要你报出在50美元和400美元之间的任意价格，我就能告诉你有多少人会以该价格购买你的软件。我们得到的就是一条经典的需求曲线，而需求曲线总是向右下方倾斜的，因为你定价越高，愿意购买你的软件的人就越少。

当然，这些数字都是虚拟的。到目前为止，我只要求你相信一件事，那就是需求曲线是向右下方倾斜的。

（如果你直到现在还是觉得很困扰，搞不明白为什么我把横轴表示销售数量，纵轴表示销售价格，销售数量明明是销售价格的函数啊，为什么要反过来画图呢？请你去问奥古斯汀·古诺[①]。他要是活到现在，可能也会写写网志。）

那么你应该把价格定在多少美元呢？

“哦，应该是50美元，因为这个价格的销售数量最大！”

不不不。你需要最大化的并不是销售数量，而是利润。

让我们来计算利润。

① 奥古斯汀·古诺（Augustin Cournot，1801—1877），法国经济学家，数理经济学的奠基人。

假设你每销售一份软件，边际成本是35美元。

也许你为了开发这个软件，迄今已经投入了25万美元。但是，它属于沉没成本。我们再也不要去关心它了，因为不管你销售出去的软件是1000份还是0份，这25万美元总是保持不变的。它已经沉没了。我们同它吻别了。不管你制定什么价格，这25万美元总是已经花出去了，因此就再也不用考虑它了。

此时，你所要担心的只是边际成本而已，也就是每多销售出去一份软件所增加的成本。其中也许包括装运和发货成本、技术支持成本、银行的服务费、CD的制造费用、热封塑料膜的费用等。我在这里只是举例，把35美元作为我的边际成本。

现在我们就要用到我们的法宝——电子表格：

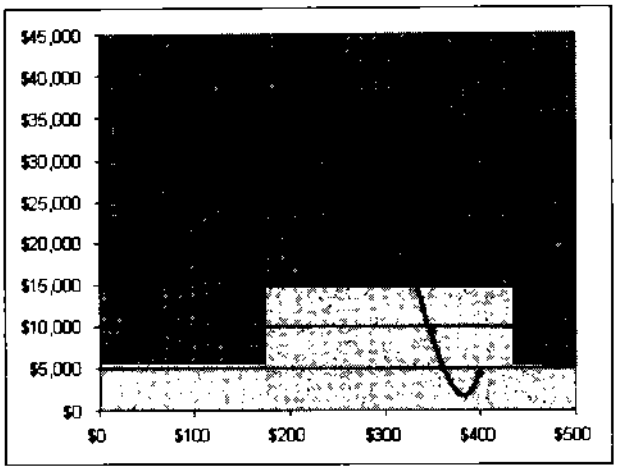
	A	B	C	D	E
1	Quantity	Price	Incr. Cost	Unit Profit (Price - Incr Cost)	Total Profit (Unit Profit x Quantity)
2					
3	12	$399	$35	$364	$4,368
4	30	$349	$35	$314	$9,420
5	100	$299	$35	$264	$26,400
6	200	$249	$35	$214	$42,800
7	250	$199	$35	$164	$41,000
8	325	$149	$35	$114	$37,050
9	450	$99	$35	$64	$28,800
10	800	$49	$35	$14	$11,200
11					

怎么来阅读这张表格呢？首先，每一行都代表一种情景。以第3行为例，如果我们定价为399美元，那么就将卖出12份软件，每份的利润是364美元，总共获利4368美元。

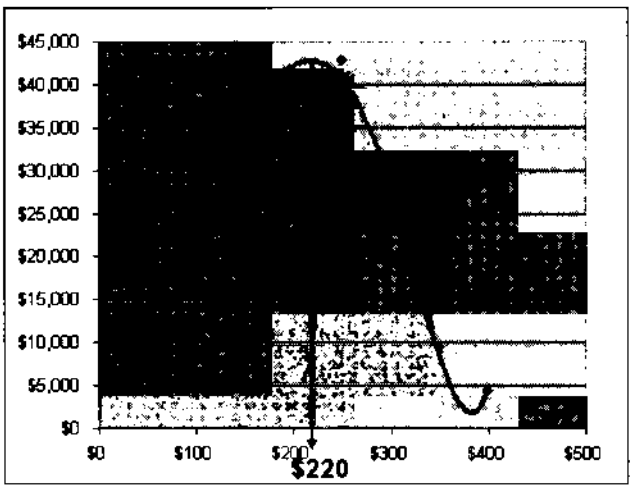
好，我们正在得到一个重要的结论！

这真地很精彩。关于如何解决软件定价问题，我想我们已经摸到答案的边了！我太激动了！

我如此激动的原因是，如果你画出一张价格和利润一一对应的散点图，你就会得到一条很漂亮的曲线，而且在这条曲线的中部还有一个隆起的部分！我们都知道隆起部分的含义！它表示这是一个极大值！或者，我们也可以把它叫做驼峰。但是在这里，它的意思就是极大值！



在这张图中，实际的数据点用小的菱形表示。我已经让Excel根据这些点画出一条光滑的趋势线。那么，接下来我只需要从隆起的顶点向下画一条垂直线就可以了，目的是找出为了获得最大的利润我应该把价格定在多少：



“啊，多么美妙的一天！太棒了！”我开怀大笑。我们已经发现了最佳价格，那就是220美元，这就是你应该为你的软件收取的价格。各位读者，非常感谢你的阅读，本文到此结束。

咳咳。

非常感谢你的阅读！下面没有什么东西可看了！请移步！

你没有离开。

我知道的。

我的有些读者很敏锐，他们察觉我想说的可能并不只是“220美元”。

好，也许吧。还有一个很小的部分，我没有接上，如果你们这些人都还在等，那么我不妨就把它接上吧。OK？OK！

你们看，我们通过把价格定在220美元，就能卖出大约233份软件，总获利43105美元。这非常好，但是有一件事让我感到很不舒坦：许多顾客完全愿意付给我们更多的钱，比如有12个高尚的灵魂就愿意全额付给我们399美元，但是我们就像对待其他顾客一样，只收了他们220美元！

399美元和220美元之间的差异（即179美元），被叫做“*消费者剩余*”（consumer surplus）。这是那些富裕的消费者从购买中获得的额外价值，它来自于消费者愿意支付的价格与实际支付的价格之间的差额。

举例来说，你打定主意要买一件美利奴羊毛衫，你以为它的价格是70美元，觉得很值得。但是，你到服装专卖店Banana Republic一看，发现那里的美利奴羊毛衫只卖50美元！这样一来，你就等于额外赚到了20美元，如果Banana Republic的羊毛衫也卖70美元，你肯定会很乐意地掏出70美元买下！

怎么能把便宜留给消费者！

这个问题让优秀的资本家感到非常困扰。真该死，如果有一笔钱，你本来就不打算留着，那么请把它给我！我将财尽其用，比如买一辆SUV、买一间高级公寓、买一架私人专机、买一艘游艇，或者买其他任何一样资本家会买的东西！

用经济学家的行话来说，*资本家想要获取消费者剩余*。

让我们来试试看。不把价格定在220美元，而是一个个地去问我们的顾客，了解他们是富裕的还是贫穷的。如果他们说他们是富裕的，我们就向他们收349美元。如果他们说他们是贫穷的，我们就向他们收220美元。

这样做可以让我们赚到多少钱？还是回来看Excel：

	A	B	C	D	E
1	Customers	Price	Qty Sold	Unit Profit	Total Profit
2	Rich	$349	42	$314	$13,188
3	Poor	$220	191	$185	$35,335
4				TOTAL:	$48,523

请注意销售数量。我们仍然卖出去了233份软件，与此前保持相同，但是最富有的42名顾客愿意支付349美元或者更多，所以我们就向他们收取了349美元。我们的利润因此出现了上升！从4.3万美元上升到了4.8万美元！太棒了！

能够这样获取消费者剩余，真是太让人激动了！

但是且慢，这还不是全部。在将上面的233份软件全部卖出去以后，我的感觉并不太好，因为还有一些顾客只愿意花99美元购买我的软件。不管怎么说，要是我以99美元的价格向这些顾客再多出售一些，我还是能够再挣到一点钱，因为我的边际成本只有35美元。

对于所有那些面对220美元的价格说出“不，谢谢”的顾客，我们打电话给他们，向他们报价99美元，会有什么结果？

价格为99美元时，我们一共有450名潜在顾客，但是不要忘记，其中的233人已经支付了更高的价格，剩下的217名顾客对我们的产品并不是那么心动，所以不愿意支付比99美元更高的价格：

	A	B	C	D	E
1	Customers	Price	Qty Sold	Unit Profit	Total Profit
2	Rich	$349	42	$314	$13,188
3	Poor	$220	191	$185	$35,335
4	Callback & Beg	$99	217	$64	$13,888
5				TOTAL:	$62,411

我找到致富门路了，我想我们的利润已经上升到了6.2万美元！总体而言，我们比最早的时候多赚了2万美元，这可都是最可靠的现钞啊，你一直梦寐以求想买一艘出海钓鱼的小船，在这条追求梦想的漫漫长路上，你现在终于可以支付头期款了。所有这一切都归功于**市场分割**，即根据你的顾客愿意支付的价格将他们分成不同的组，从每个顾客身上攫取最大的消费者剩余。市场分割真是太了不起了，简直就像蝙蝠侠那样神奇，如果我们要求每一个顾客告诉我们他们愿意支付的最高价格，然后就向他们收取这个价格，我们能赚到多少钱啊？

	A	B	C	D
1	Qty Sold	Price	Unit Profit	Total Profit
2	12	$ 399	364	$ 4,368
3	18	$ 349	314	$ 5,652
4	70	$ 299	264	$ 18,480
5	100	$ 249	214	$ 21,400
6	50	$ 199	164	$ 8,200
7	75	$ 149	114	$ 8,550
8	125	$ 99	64	$ 8,000
9	350	$ 49	14	$ 4,900
10			TOTAL	$ 79,550

简直像做梦一样！利润达到了将近8万美元！比起我们原来只有一个价格时几乎翻了一倍！获取消费者剩余显然非常有利可图。即使有350个令人讨厌的顾客，每人只愿意向我们支付49美元，但是他们依然对利润有所贡献。而且所有的顾客都是满意的，因为我们问他们收取的都是他们本来就愿意支付给我们的，所以这一次我们没有欺骗任何人。好吧，就算稍微有点吧。

下面是一些你可能比较熟悉的市场分割的例子。

- 老年人优惠价，因为老年人往往依靠固定的退休金生活，他们愿意支付的价格，要比在职人员愿意支付的价格更低。
- 下午场电影的优惠票价（只对那些没有工作的人有吸引力）。
- 各种奇怪的飞机票价，似乎每个人支付的飞机票价格都不一样。它的秘密就在于，那些商务乘客的旅费是公司支付，所以他们根本不在乎票价的高低，而普通旅客是自己支付旅费，所以飞机票价太高的话，他们就不会坐飞机了。

  当然，航空公司不会来问你是不是商务乘客，因为其中的奥秘很快就会泄露出去，所有乘客为了得到更低的票价都会撒谎。但是，商务乘客几乎永远都是在工作日出行，他们痛恨周末远离家庭去工作。所以，航空公司就想出了票价策略。如果你的来回机票中包含星期六晚上在外过夜，那么你很可能就不是商务乘客，航空公司就会给你非常低的票价，前提是你星期六晚上在外过夜。

还有更微妙的分割市场的方式。你还记得你在报纸上看到的那些购物优惠券吗？只要你把它们剪下来，记得买东西的时候带到商场去，就可以获得优惠，每盒洗衣粉便宜25美分。这很好，但是这些优惠券的麻烦在于，它们要求许多手工劳动。你得把它们剪下来，然后整理好，记得在什么时候使用哪一张，再根据你现有的优惠券选择具体的商品品牌等。最终的效果就是，如果你剪下优惠券，那么你实际上在从事一种每小时报酬大约为7美元的劳

动。

好的，如果你退休了，依靠社会保险金过日子，那么每小时7美元听起来还不错，所以你就干了。但是，如果你是美林证券公司的证券分析师，你的工作是将许多垃圾的互联网公司吹得天花乱坠，每年的报酬高达1200万美元，那么去干每小时7美元的事简直就是一个笑话了。所以，你不会去把报纸上的购物优惠券剪下来。真该死，有这么一点空闲时间，就够你对十家垃圾互联网公司发出“推荐买入”的评价了！所以，购买优惠券本质上是消费品公司用来区别顾客、实现差别定价的工具，它有效地将市场一分为二。回邮折扣[①]（Mail-in rebate）的性质同购物优惠券很类似，并且还包含了其他一些小伎俩，比如一旦你回邮，你的地址就暴露了，在未来你就成了直接营销（direct-market）的对象。

分割市场的方法还不止于此。你能够用不同的品牌区分你的产品（Old Navy、Gap，还有Banana Republic[②]），然后希望富人能够很轻松地记住买衣服要去Banana Republic的专卖店，而穷人则是会去Old Navy的专卖店。为了防止有人忘记不同品牌的含义，去了错误的专卖店，Banana Republic的专卖店就很合适地坐落在满是价值200万美元公寓的街区中，而Old Navy的专卖店则是坐落在火车站旁边，当你结束了一天艰苦的体力劳动，拖着疲惫的身躯，准备坐火车返回新泽西的时候[③]，就会经过那里。

在软件业中，你可以将你的产品分成“专业版”和“家庭版”，两者只有很细微的差别。你希望那些公司用户（重复一遍，即那些不用自己出钱的用户）会感到不好意思上班时用Windows XP家庭版工作，而去选择购买专业版。上班时用家庭版？感觉上有点像穿着你的睡衣上班！太恶心了！

小提示。如果你也想尝试市场分割的做法，那么对某些特定用户提供折扣可能比要求一些用户付出额外费用更好。没有人喜欢被索要高价。人们更喜欢以99美元购买标价为199美元的商品，而不是以79美元购买标价为59美元的商品。理论上讲，人类应该是具有理性的。79美元少于99美元。但是在

① 回邮折扣是一种促销手段，消费者先全额购买商品，然后将收据或产品外包装上的商标寄回厂商，厂商再将一部分购物款以支票的形式寄回给消费者。

② 这三个服装品牌都属于同一个厂商的产品，Old Navy和Banana Republic是Gap的子品牌。

③ 新泽西州坐落在纽约的北面，房价比纽约便宜很多。很多美国人都是工作在纽约，住在新泽西，每天依靠火车往返。

现实中，人们不喜欢上当受骗。买到一件便宜货的感受要比被漫天要价的感受好多了。

但是……

我所说的还是只是比较容易的部分。

真正的难点在于我上面告诉你的这些事情都不能算是完全正确的。

回顾我的说法，难道定价的奥秘就是分割市场吗？这样做会把顾客惹火的。人们希望他们支付的是一个公平的价格。他们不希望只因为他们不够聪明，没有找到购物优惠券，就要付出额外的费用。航空公司非常非常善于分割市场，几乎做到了同一架飞机上每一个乘客支付的票价都是不一样的。但是，结果却是大多数人感到他们的交易并不合算，因此对航空公司一点好感也没有。如果低成本的航空公司进入这个市场，顾客有了新的选择，他们就会毫不犹豫地同以前的航空公司说再见，因为他们觉得以前的航空公司这些年来一直试图偷他们的钱。

如果一个热门的网志作者发现，你兜售的高价打印机同低价打印机配置几乎相同，唯一的差别就是你打开了打印速度加速芯片，那么你就祈祷上帝保佑你吧。

程序员最可信赖的求职帮手

所以，虽然市场分割是一种“获取消费者剩余”的有用工具，但是长期来看，对你的产品的形象也会产生显著的负面效应。许多小的软件厂商取消优惠券、折扣价、优惠协议、多版本策略、分层定价以后，反而发现他们的收入上升，讨价还价的顾客减少了。某种程度上，如果你向每个人都报价100美元，那么顾客似乎也愿意支付和其他人一样的价格，但是如果顾客知道别人收到的价格是78美元，那么他连79美元也不愿意支付。别忘了，通用汽车公司为了贯彻统一价格原则，避免顾客讨价还价，不惜再开出一家公司Saturn[①]。

假设你愿意接受负面效应，愿意接受由于过度的价格歧视而引起的消费

① Saturn是通用汽车公司最年轻的品牌。1982年，为了抗衡日本中小型轿车对美国国内市场的冲击，通用汽车公司投资近20亿美元，在田纳西州设立了新的Saturn子公司，它完全独立于母公司，使用独立的品牌、独立的组装厂、独立的零售网络，而且统一销售价格，避免用户讨价还价。

者好感的丧失，即使那样的话，市场分割也不是你想做到就能做到的。首先，只要你的顾客发现你有价格歧视行为，他们就会想方设法得到更低的价格。

- 那些频繁出行的商务乘客就会调整他们的机票，同时购买两套来回机票，每套都包括星期六晚上在外过夜。举例来说，有一个咨询顾问周末住在匹兹堡，平时（从星期一到星期四）在西雅图工作。他就会购买两套往返机票，一套是两个星期行程的从匹兹堡到西雅图的往返机票，另一套是在这两个星期之间的、周末从西雅图到匹兹堡的往返机票。这两套往返机票都包括星期六晚上在外过夜，所以价格要比分开买便宜得多。更妙的是，即使没有优惠价，这些航班也是他本来就要乘坐的。

- 你的产品有校园价？所有与学校根本不相干的人最后都会拿到这个价格。

- 如果你的顾客互相之间有交流，他们就会发现你提供给其他人更低的价格。这将使得你最后不得不向所有人都提供最低价。那些大的集团客户尤其如此。虽然它们理论上应该“最不在乎价格”，因为它们很有钱，但是实际上它们都是砍价的高手。大集团都设有专门的采购部，该部门职员的专职工作就是砍价。这些人参加各种研讨会，了解怎样才能拿到最好的价格。他们整天在镜子前面练习，一遍又一遍地说：“不行。价格必须更低。”你的销售人员在他们面前不会有任何机会讨到便宜的。

有一些软件公司自以为聪明，搞出了两种市场分割的变体。它们其实也不是什么妙计。

坏主意之一：网站许可证

这种方法与市场分割正好相反。我的某些竞争对手就这样做，他们对小客户按照人头进行收费，但是同时又以固定价格出售一种“无限制”许可证。这实在是很奇怪的做法，因为你在将最大的价格优惠不偏不倚给予那些最大的客户，而他们实际上是愿意付给你最多钱的人。你真希望IBM购买你的软件，让它的40万名雇员使用，却只付给你2000美元吗？你希望这样吗？

一旦你有一个“无限制”使用的价格，就相当于你立即将巨大的消费者

剩余送给那些对价格最不敏感的顾客。你本应该从他们身上挤下来源源不断的现金才对，现在全泡汤了。

坏主意之二："你有多少钱"定价法

那些由离开甲骨文的职员创立的软件公司就经常使用这种方法。你在他们的网站上找不到任何地方标示出软件的价格。不管你找得多么努力，你最终所能找到的只是一张表单，你被要求提供姓名、地址、电话号码、传真号码等信息。这样做当然是有原因的，不过肯定不是因为那家公司想给你发传真。

很明显，整个计划是这样的：你提交联系方式以后，销售人员就会给你打电话，判断你的承受能力，然后决定向你开价多少。这真是完美的市场分割！

这种方法也不会很有效。首先，低端的客户会直接选择离开。他们会觉得，如果价格没有在网站上明示，那么他们肯定没有能力购买这个产品。其次，那些不喜欢被销售人员骚扰的人也会直接选择离开。

更糟的是，一旦客户发现你的价格是可以协商的，你的市场分割策略就会失败，最后反而取得反效果。为什么会有这样的结果？你把软件销售给大公司，它们本来是愿意向你支付最多的钱的，但是它们的采购人员老练得令人难以置信。他们知道你的销售人员拿的是佣金，只有把产品销售出去，你的销售人员才有收入，他们也知道你的销售人员每个季度都有一定要完成的销售指标，他们还知道每当一个季度要结束的时候，你和你的销售人员都非常急于要把产品卖出去（因为销售人员想得到他们的佣金，而你是因为不想被风险投资家或华尔街打破头）。所以，大的集团客户总是等到每个季度的最后一天才出手，然后总是能够得到一个便宜得出奇的价格，而且其中还涉及很多奥妙的会计处理方法，使你能够把许多从来没有真正得到过的收入记入你的公司的销售收入之中。

所以，不要使用网站许可证，也不要想着先判断客户的经济实力，然后再根据对象的不同决定价格。

但是，且慢！

你是否真想要利润最大化？我前面并没有和盘托出全部实情。你其实并

不需要关心这个月的利润是不是达到了最大化。你真正需要关心的是将你所有时期、包括未来的利润的总和最大化。从技术上看，你要的是所有未来利润的净现值（NPV）最大化（当然，前提是保证现金储备不出现负值）。

小插曲：什么是净现值

今天的100美元和一年后的100美元，哪一个价值更高？

很显然，答案是今天的100美元更值钱。因为你能够将这笔钱用于投资，比如投资于债券，然后到了年底你就能连本带息地收回来，假定可以拿到102.25美元。

所以，当你比较一年后的100美元和今天的100美元，你必须考虑到利息因素，要将一年后的100美元扣除利息。假定利息率是2.25%，那么一年后的100美元就相当于今天的97.80美元，这就叫做一年后100美元的净现值（NPV）。

如果考虑更遥远的未来，你就需要扣除更多的利息。5年后的100美元，按照今天的利率计算，只相当于今天的84美元。所以，84美元就是5年后100美元的净现值。

有两笔钱，你更愿意挣哪一笔？

选择一：未来三年的收入分别为5000美元、6000美元、7000美元。

选择二：未来三年的收入分别为4000美元、6000美元、10000美元。

虽然前两年的收入低，但是选择二应该更好一些，即使扣除了利息也是如此。如果你采用选择二，那么就好像第一年投资了1000美元，第三年就可以净赚3000美元，这真是一笔非常可观的投资！

我提这个问题的原因是，软件的定价有三种方式，分别是免费、廉价和高价。

(1) **免费**。开源软件等就采用这种方式。这与我们现在讨论的问题毫不相干，如果你选择这种方式，那么这里就没有你需要看的东西，你可以走开了。

(2) **廉价**。价位在10美元到1000美元之间，以低价位面向庞大的顾客群，不设置专门的销售人员。大多数上架销售的家用和小企业用的软件都属于这

一类。

(3) 高价。价位在7.5万美元到100万美元之间，仅仅面向一小部分有钱的大公司销售，并且配备一支干练的销售队伍，一年中有6个月在做紧张的PowerPoint展示，只为了能够做成一笔该死的交易。这就是甲骨文公司的模式。

这三种方式都是有效的。

你有没有注意到价位不是连续的？没有软件定价在1000美元和7.5万美元之间。我来告诉你为什么。如果你的软件定价高于1000美元，那么你立刻就会遇到非常严重的大公司内部采购政策的权限问题。你的软件就会在那些大公司的采购预算报表中占有单独的一行。在你的软件最后被购买之前，必须得到采购经理和CEO的批准，还必须经过竞争性招标以及大量的文书工作。所以，你必须派出一个专门的销售人员，到客户那里做PowerPoint展示，你还必须负担他的飞机票，出钱让他陪客户一起打高尔夫球，出19.95美元让他陪客户一起到Ritz Carlton酒店看限制级电影。干完这些事情以后，成功完成一次交易的成本就上升到平均为5万美元以上。如果你派出一个专门的销售员去见客户，软件的要价就不能低于7.5万美元，否则你就要亏本了。

这件事情的荒唐之处就在于，大公司为了防止买到漫天要价的软件，就制定了这样一套机制，设置了这么多麻烦的步骤，最后的实际效果却是大大抬高了软件的成本，从1000美元抬高到了7.5万美元。这其中的大部分钱都用在了为了完成交易所必须通过的重重障碍上，而这些障碍都是那些大公司为了保证交易不出错而设下的。

现在，你很快地浏览一遍Fog Creek的网站，就会发现我坚定地站在第二号阵营，也就是廉价软件阵营这一边。为什么？以低价销售软件就意味着我能立刻得到几千个顾客，其中一些是小客户，一些是大客户。所有这些顾客都会使用我的软件，并把它推荐给他们的朋友。当他们的业务不断成长后，就会购买更多的使用许可证。当他们的雇员离开原公司、加入新公司后，也会把我的软件推荐给新公司。实际上，我把软件的价格定得很低就是为了换来草根用户的支持。我把FogBugz的低价看作一种广告投资，期待在长期中会带来好几倍的回报。到目前为止，这种策略很成功，FogBugz的销售连续三年每年增长100%以上，我们完全没有做市场推广，依靠的都是口碑以及现有用户向我们购买更多的使用许可证。

作为对比，再来看看BEA软件公司。它属于大公司、高价软件的那一类。仅仅因为它的软件价格如此之高，几乎无人有使用它的产品的经验。高校的毕业生创办互联网公司，不会选择BEA的技术，因为他们读书的时候根本用不起BEA的产品，所以对它一无所知。许许多多其他优秀的技术，就是因为价格太高，而注定了黯然退场的命运。苹果公司的WebObjects始终无法被应用程序服务器采用，就是因为它高达5万美元的价格。谁在乎它有多优秀啊？根本没人使用！Rational软件公司[①]生产的所有软件都是这种命运。让用户开始使用这样昂贵的软件只有一种方法，就是使用非常昂贵和强大的推销力度。在这样的价格级别，你的推销对象必须是企业的决策者，而不是技术人员。但是如果软件的技术水平很低劣，那么即使有很强的销售力量，即使管理层强制技术人员使用这种技术，技术人员也可能做出积极的抵制。我们有很多FogBugz的顾客，他们花了远远超过10万美元的代价买来了Remedy软件公司、Rational软件公司或者Mercury软件公司昂贵的产品，但是买来以后，却把它们束之高阁，因为这些软件实际上不是很优秀，并不那么有用。然后，他们购买了价值2000美元的FogBugz，这才是他们真正地在日常工作中使用的产品。Rational软件公司的销售人员对我大加嘲笑，因为我的银行账户多了2000美元，而他的银行账户多了10万美元。但是，我的顾客数量要比他多得多，而且我的顾客都在使用我的软件，并且向他人推荐，向他人传播。相比之下，Rational软件公司的顾客要么根本不使用买来的软件，要么就算使用，也对这种产品感到难以容忍。但是，那个销售人员仍然对我大加嘲笑，因为他已经享受上了40英尺长的游艇，而我还只能躺在浴缸里玩廉价的橡皮鸭。就像我上面说的，软件的三种定价方式都是有效的。但是，软件廉价出售就像你在贴钱做广告或者在对未来进行投资。

好了。

我说到什么地方了？

啊，想起来了，在我唠唠叨叨说了这么一大堆东西、嘴巴起泡之前，我正在引导你对需求曲线有一个全面的认识，并且在需求曲线的推导逻辑中寻找瑕疵。你可能心里会有疑问："我怎么知道人们愿意付给我多少钱呢？"

没错。

① Rational软件公司成立于1981年，在2003年被IBM公司以21亿美元的价格收购。

这是一个难题。

你不可能发现需求曲线真实的样子。

你可以把调查对象分成不同的组，去询问他们，但是他们会对你撒谎。有些人撒谎是因为想炫耀他们的财富和慷慨。“嗨，我眼睛也没眨，就买了一条400美元的牛仔裤！”另一些人撒谎是因为他们很想买你的产品，他们觉得如果告诉你一个很低的数字，你给他们的报价就会低一点。“网志软件？这个嘛，我最多只想出38美分。”

然后，第二天你询问第二组人。这次，头一位接受询问的人士暗恋本组的一位漂亮女士，希望给那位女士留下深刻印象，所以他大谈他的车是多么豪华，让所有人都觉得他是一位大亨。过了一天，在休息时间，你请全组的人去星巴克喝咖啡，就在你上厕所的时候，其他人就互相攀谈，谈论这个地方的咖啡要4美元一杯，所以等到你回来询问他们愿意为软件支付多少钱的时候，他们一个个都变得锱铢必较。

最后，你终于找到有一组人，他们同意你的软件每个月值25美元。然后你问他们，如果买一张永久性使用许可证他们愿意付多少钱，这些人居然表示价格绝对不能高于100美元。这年头，真有人一点算术都不会。

另一种情况是，你找到一些飞机设计师，问他们愿意付多少钱。他们心里觉得99美元就是最高价格了，完全没有意识到他们现在日常使用的软件每个月就要花掉大概3000美元。因为他们不管软件采购，所以对软件的价格一点概念也没有。

所以，日复一日，你不停地询问每个人愿意为某种商品付多少钱，得到的都是截然不同的回答，注意，我说的是截然不同。真正能找到正确答案的方法只有一种，那就是将软件出售，看看有多少人会购买它，这样你才能知道人们愿意为它付多少钱。

然后，你再不断地调整价格，测试在不同价格的情况下市场的反应，试着画出需求曲线。但是，只有等到你的顾客规模达到100万左右，并且能够绝对保证顾客A不会发觉你提供给顾客B更低的价格时，你才会得到在统计学意义上正确的需求曲线。

在实践中往往存在一种强烈的倾向，那就是你会假设，针对大规模人群

的试验应该像高中实验室里的化学试验那样准确和可以重复。但是，每一个做过社会试验的人都知道，每次试验得到的结果差异大得惊人，而且不可重复。如果你想要对得到的结果充满自信，那么唯一的方法就是千万不要把同样的试验做两次。

事实上，你连需求曲线向下倾斜都无法保证。

我们假设需求曲线向下倾斜的唯一理由就是，我们自欺欺人地认定“如果Freddy愿意出130美元购买一双运动鞋，他肯定愿意出20美元，购买同样的运动鞋”。真地如此吗？哈，只要Freddy不是美国的青少年就行！美国的青少年要是穿上20美元一双的运动鞋会觉得没脸见人，他们是死也不肯穿的。嗯，有点像被判死刑？你有没有穿运动鞋？价格是20美元一双？穿着去上学？

我在这里可不是开玩笑。价格是一种信号。在我的老家，我记得看一场电影需要11美元。可是老天啊，以前曾经有过一家电影院，票价只要3美元。是不是所有人都去那里看电影了？我不认为是这样。很明显，那家电影院只是一家垃圾电影的倾倒场。某人现在正长眠于纽约的伊斯特河底，脚上穿着一双灌满水泥的20美元运动鞋，因为他竟然胆敢告诉消费者哪些电影是行业内公认的烂片。

你看，人们总是相信物有所值，或者通俗地说，就是便宜没好货。上一次我需要大量硬盘空间的时候，我买了一些非常便宜的硬盘，据说还是由Porsche先生亲自设计的，折算下来，1GB储存空间的价格大约为1美元。结果不到6个月，我买的4个硬盘全部坏掉。上个星期，我用Seagate公司的Cheetah SCSI硬盘把它们全换掉了，新硬盘的价格大约是1GB储存空间需要4美元。我选择它们的理由是，4年前我创立Fog Creek软件公司的时候就用了这种硬盘，一直用到现在，一点差错都没有出过。请把“物有所值”这句话抄下来。

许许多多的例子都表明，现实世界中“物有所值”确实存在，当消费者对商品所知有限时，消费者通常就会认为比较贵的商品就是质量比较高的商品。想买咖啡机吗？只要质量优异的咖啡机？你有两种选择。你可以去图书馆，找到《消费者调查报告》（Consumer Reports）杂志中有咖啡机的那一期，或者你可以去家居零售商店Williams-Sonoma，从它的货架上选择价格最高

的那个品种。

当你制定价格时，你就在发出信号。如果你的竞争者的软件价格大约在100美元到500美元之间，你就想，嗨，我的产品的价格取它们的中位数就行了，所以我就以300美元的价格销售我的软件了。好，我问你，你觉得你正在给消费者发出怎样的信息？你正在告诉他们，你觉得你的软件只是一个“不过如此”的产品。我有一个更好的点子，你就开价1350美元。现在，你的顾客就会想：“哦，你看，这玩意一定很有用，因为厂商的开价简直疯了！”

然后，顾客决定不购买你的软件，因为在AMEX[①]上市的公司不能购买价格超过500美元的软件。

真是太悲惨了。

你对定价这件事情知道得越多，看上去就对它越不了解。

我已经在这个话题上唠唠叨叨说了这么多，但是我真不觉得我们已经得到了任何结论。在这一点上，我和你们都一样。

有些时候，好像还是当一个出租车司机更容易一些，因为乘坐出租车的价格是政府规定好的。或者卖糖也可以，就是普通的白糖。对，当你感到痛苦不解的时候，你至少还可以尝一口糖，它会给你带来一点甜蜜。

记住我的建议。我再把它说一遍，你可以把书翻到前几页去看一看：如果你不知道怎么定价，就为你的软件定价0.05美元，除非它用于故障追踪，在这种情况下，正确的定价是3000万美元。感谢你一直读到了最后。我很抱歉，读完此文后，你甚至比读此文之前更不知道如何为软件定价了。

① AMEX是指美国证券交易所（American Stock Exchange），只有不满足纽约证券交易所和纳斯达克市场上市条件的小型公司才在这个交易所上市。

第九部分

修订软件

35 五个为什么
36 确定优先顺序

35

五个为什么

2008年1月22日，星期二

2008年1月10日的凌晨3点30分，一阵急促的嘟嘟声惊醒了我们的系统管理员Michael Gorsuch，当时他正在位于布鲁克林的家中睡觉。那是网络监控系统Nagios自动发出的一条短消息，报告系统不正常。

Michael Gorsuch从床上爬起来，不小心踩到了正在床边酣睡的狗，把狗也弄醒了。那条狗气呼呼地窜到客厅里，在地板上撒了一泡尿，然后又回来继续睡觉。这个时候，Michael Gorsuch已经在另一间房间里打开了电脑，发现在他负责的三个机房中，有一个位于曼哈顿闹市的机房连不上去。

这个特别的机房位于曼哈顿繁华地区一幢很安全的大楼里，面积很大，由PEER 1公司负责管理。它配备了发电机，以及足够使用好几天的柴油，还有大量的蓄电池，保证在自备发电机启动前能够提供几分钟的电力，防止出现访问中断。它还有巨大的空调设备、多个高速互联网出口，以及非常务实、负责的工程师，他们总是以一种单调、稳重、井然有序的方式进行工作，而不会用花哨、刺激、时髦的方式做事，所以那是一个非常可靠的机房。

像PEER 1这样的ISP喜欢使用一个叫做SLA（Service Level Agreement，服务级别协议）的术语，承诺保证他们服务的正常运行时间（uptime）。典型的SLA往往这样写："保证99.99%的时间正常运行。"你可以做一下算术，地球围绕太阳公转一圈的时间是525 949分钟（或者以一年365天计算，就是525 600分钟），这就允许他们每年有52.59分钟的故障时间。如果故障时间多于这个数字，SLA通常会写清楚提供某种形式的赔偿，但是说实话，赔偿的金额往往是微不足道的……比如，你购买了一年的服务，他们就按照发生故

障的分钟数，把相应的使用费退还给你。我记得有一次，某一个机房发生故障，整整两天都不能访问，给我们造成了几千美元的损失，结果我从ISP那里得到的唯一赔偿就是10美元的退费。所以，SLA保证条款其实是没用的。而且正是因为赔偿金额微不足道，许多ISP开始打广告，声称正常运行时间高达100%。

过了十分钟，Michael Gorsuch连上了曼哈顿的机房，一切似乎都恢复正常，他就又回去睡觉了。

大约到了清晨5点，这个问题又出现了。这一次，Michael Gorsuch打电话到PEER 1在温哥华的网络运营中心NOC。对方开始调查，做了一些测试，但是没有发现任何异常。5点30分，一切好像又恢复了正常。但是，Michael Gorsuch还是不放心，不敢再回去睡觉了。

清晨6点15分，曼哈顿机房彻底无法连通。PEER 1在他们那一边找不到任何异常。Michael Gorsuch穿好衣服，搭地铁进入曼哈顿。服务器本身好像没有出问题，都在正常运行。PEER 1的网络连接也是好的，问题出在交换机。Michael Gorsuch取下了交换机，将我们的路由器直接连到了PEER 1的路由器。真是奇妙，我们的服务器又可以重新访问了。

这时，天已经亮了，我们在美国的客户开始上班了，他们会感到一切正常。但是我们在欧洲的客户却已经发来了电子邮件，抱怨不能连上我们的服务器。Michael Gorsuch花了一些时间做事后分析，发现事故原因是交换机上一个简单的设置。交换机能够使用多种网速进行工作（10Mbit/s、100Mbit/s或者1000Mbit/s），你可以手动设置网速，也可以让交换机自动选择与所在网络相适应的网速。我们这台交换机就设在了自动选择档，问题就出在这里。通常情况下，交换机的这个设置会正常工作，但不能保证一定不出故障，1月10日的凌晨，它就发生故障了。

Michael Gorsuch其实早知道这个设置可能会引发故障，但是安装交换机的时候，他忘了手动设置网速，所以交换机上的设置一直是出厂时的自动档。这个设置也能正常工作，直到出了问题为止。

Michael Gorsuch并没有因为解决了问题而感到高兴，他给我写了一封电子邮件。

程序员最可信赖的求职帮手
程序员最可信赖的求职帮手

程序员最可信赖的求职帮手
程序员最可信赖的求职帮手

程序员最可信赖的求职帮手

我知道自从推出“FogBugz在线服务”以后，我们并没有一个正式的SLA条款，但是我觉得应该拟定一个供内部使用（至少如此吧）。这样我就能衡量我自己（甚至整个系统管理团队）是否达到了业务管理目标。我本来已经开始着手写了，但是一直没有完成，经过今天早上的事故后，我会尽快完成它。

SLA条款通常用“正常运行时间”来定义，所以我们需要定义“FogBugz在线服务”的“正常运行时间”到底是多少。确定以后，我们就要把它转化成一系列政策，然后再把政策转化成一整套的监控/报告脚本，每隔一段时间就检查一次，看看我们是否达到了业务管理目标。

好主意！

但是，SLA也不是包治百病的良药，它也存在一些问题。最大的问题就是缺乏制定标准必需的统计资料，因为服务中断的情况很少发生，所以你得不到足够的数据，无从知道多长的运行时间才算是“正常的”。如果我没有记错，自从6个月前我们推出“FogBugz在线服务”以来，包括这一次在内，只发生了两次服务的突然中断。其中只有一次是由于我们的过失而造成的。互联网上大多数正常运行的在线服务网站，一年中只发生两次、也许三次服务中断。正是因为服务中断很少发生，所以每次中断的时间长度就开始变得很重要，这才是网站之间出现巨大差异的地方。突然之间，你正在谈论的问题就变成了，如何才能最快地派一个人找到发生故障的设备，然后替换掉它。为了得到非常高的正常运行时间，你等不及派人去换掉出问题的设备，也等不及工程师去检查哪里出了问题，你必须事先就考虑到每一个可能出错的地方，但是这实际上不太可能做到。能够将你置于死地的不是意料到的突发状况，而是意料之外的突发状况。

要想得到真正高的服务稳定性，成本是极其高昂的。俗称“6个9”的服务稳定性（99.9999%的时间正常运行）意味着每年下线时间不能超过30秒。这真有点接近荒谬了。即使有人声称他们已经投资了上千万美元，建立了超一流的大型超冗余“6个9”系统，也无法排除出现严重故障的可能性。某一天当他们醒来的时候，我不知道是哪一天，但是会有这么一天，他们发现一种异乎寻常的故障以一种异乎寻常的方式发生了，比如三个机房都遭到了电子脉冲EMP炸弹的袭击，他们只能拼命捶自己的脑袋，眼睁睁看着服务中断

14天。

请这样想，如果你的“6个9”系统由于某种神秘原因突然下线了，然后你花了一个小时找到了原因并将其修复，即使这样，你也已经把直到下个世纪的服务中断时间额度都用光了。即使是公认的最可靠系统，比如AT&T的长途电话服务，也会出现长时间的服务中断（1991年中断了6个小时），这意味着它们的服务稳定性只能到达一个令人羞于启齿的“3个9”水平（99.9%的时间正常运行）……AT&T的长途电话服务被认为是“电信级”（carrier grade）的系统，是服务稳定性的黄金标准，你的系统会比它更稳定吗？

提高服务稳定性的最大困难，就是“黑天鹅难题”（problem of black swans）。这个名词是由Nassim Taleb提出来的（www.edge.org/3rd_culture/taleb04/taleb_indexx.html），他这样定义：“黑天鹅代表外来因素，是一个超出正常预料的事件。”几乎所有的互联网服务中断都来自于意料之外的突发事件，属于极其小概率的非主流意外。这类事件是如此罕见，以至于常规的统计方法（比如“故障间隔平均时间”）都失效了。“请问新奥尔良市发生特大洪水的平均间隔时间是多少？”

测量出每年服务中断的分钟数并无助于预测你的网站第二年会中断服务多久。这让我想到了今天的民用航空业。美国的全国运输安全委员会（NTSB）的成就非常卓越，所有常见的飞机坠毁的原因都已经被消除了，结果就导致如今每一次发生飞机坠毁，事故的原因看来似乎都属于非常令人意外的、独一无二的“黑天鹅因素”。

服务稳定性有两个极端，一个是“极端不可靠”，服务一次又一次地中断，简直愚蠢至极；另一个是服务稳定性“极端可靠”，你花了几百万美元，终于将每年的“正常运行时间”增加了一分钟。在这两个极端之间，存在一个服务稳定性的最佳位置，即所有被预计到的突发情况都已经事先做好了准备。你预计到硬盘可能会发生故障，就事先做好了准备，所以当硬盘真地发生故障的时候，你的服务就不会中断。你预计到DNS服务器可能会发生故障，就事先做好了准备，所以当DNS服务器真地发生故障的时候，你的服务就不会中断。但是，意料之外的突发情况也许就会让你的服务出现中断。这种局面就是我们能够期望的在两个极端之间达到的最佳位置了。

为了达到这个最佳位置，我们借鉴了日本丰田公司创始人丰田佐吉

（Sakichi Toyoda）的思想，他提出了五个为什么。当某个地方出错的时候，你就问为什么，一遍遍地追问，直到你找到根本性的原因为止。然后，你就针对根本性的原因开始着手解决问题，你要从根本上解决这个问题，而不是只解决一些表面的症状。

我们早就提出过“解决问题有两种方法”[①]，“五个为什么”同我们的提法很接近，所以我们就决定开始采用这种方法。下面就是Michael Gorsuch列出的思考过程。

我们与PEER 1纽约机房的连接中断了。

- 为什么？我们交换机里的网线接口好像不工作了。
- 为什么？与PEER 1的网络运营中心交换意见后，我们判断这个问题很可能是由于网速/双工模式不匹配（speed/duplex mismatch）造成的。
- 为什么？交换机的网速开关设在了自动调节档，而没有被手动设置在一个固定档。
- 为什么？许多年前，我们就清楚地知道有可能发生此类故障。但是，我们始终没有写出一份书面的技术说明文档，用于指导和检查交换机在生产环境中的设置。
- 为什么？我们总是很狭隘地看待技术说明文档，觉得只有在找不到系统管理员的情况下才需要去看它，或者觉得只有运营团队中那些不负责系统管理的成员才需要看它。我们没有认识到应该把它作为技术操作的标准和确认清单。

“如果我们事先就写好一份书面的标准操作流程，安装完交换机后，再根据书面流程一一核对安装步骤，这次的服务中断事故就不会发生。”Michael Gorsuch写道，“或者假定我们已经有了一份书面的操作流程，但是写得不够完整，那么等到事故发生以后，我们就需要对这份文档进行相应的补充升级，确保类似的事故以后不再发生。”

经过几次内部讨论以后，我们所有人都同意不为服务稳定性设置一个静态值作为目标，那是毫无意义的。我们觉得，如果有人希望通过测量某些无意义的指标来改进工作，那肯定是没用的。我们真正需要的是一个能够不断

① 参见第32讲。

改进工作质量的流程。所以，我们决定不向我们的顾客提出一个SLA条款，而是搭建一个网志。在这个网志上面，我们将实时记录每一次的服务中断，提供完整的事后分析，询问五个为什么，找到根本性的原因，告诉我们的顾客为了防止类似故障再次发生我们所采取的举措。就拿这一次的交换机事故来说，我们采取的变化就是，在内部文档中写入详细的操作步骤和检查清单。以后再在生产环境中安装交换机的时候，所有操作步骤都必须严格按照文件中写好的步骤完成。

我们的顾客可以访问这个网志，看看故障的原因到底是什么，以及我们正在怎样改进我们的服务。我们希望，我们的顾客能够因此增强信心，相信我们的服务品质正在稳步提高。

与此同时，如果我们的顾客感到我们的故障对他造成了影响，他就可以向我们要求补偿，客服人员会给他的账户延长使用期限或者退款。我们让顾客自己决定到底该补偿多少，最多可以延长使用期限一个月，因为不是每个顾客都会注意到发生了服务中断，更不要说遭受损失了。我希望我们的这些做法能够提高我们的服务稳定性，到达一种我们可以接受的程度，即我们的目标就是：我们遇到的所有引起服务中断的故障都是真正由于极其罕见的、无法预料的“黑天鹅因素”而引起的。

另外，对，我们需要再招聘一名系统管理员，以免深更半夜再发生故障的时候，只有Michael Gorsuch一个人能被叫醒。

36

确定优先顺序

2005年10月12日，星期三

现在是时候结束与FogBugz 4.0版本的纠缠，开始向5.0版本进军了。不久以前，我们刚刚发布了一个4.0版本的大型升级包，解决了一大堆没人会注意到的小bug（也引进了几个没人会注意到新的小bug）。接下来，该是为这个软件增加几个真正的新功能的时候了。

动手写代码之前，我们规划增加的功能，多到足够1700个程序员干上几十年。不幸的是，我们一共只有3个程序员，而且我们希望新版本软件能在明年秋天上市。因此，必须确定开发的优先顺序。

在我讲述我们如何确定各种任务的优先顺序之前，让我先列出两种不应该被采用的方法。

不能采用的方法一。你发现你开发某个功能只是因为你答应过一个顾客，这时你的大脑中就应该亮起红色警报了。如果你的工作只是服务于某个特定顾客的需要，那么就有两种可能，一种可能是你有一个无法无天的销售员，另一种可能是你正在走向开发“个性化软件”这条危险的道路。“个性化软件”本身并没有错，而且做起来以后，你会觉得很舒服，但是它的盈利性就是不如面向整个市场销售的上架软件。

面向整个市场销售的上架软件采用“要么接受、要么放弃”的开发模式。你把软件做出来，用塑料膜封装好，送进商店，放上货架，顾客要么购买，要么不买。他们不会说，你再开发一个功能我就买，也不会打电话与你讨论应该开发哪些功能。正如你也不会打电话给微软公司，对那里的程序员说：

“嗨，我喜欢Excel里的BAHTTEXT函数[1]，它能把阿拉伯数字改写成泰国文字的形式。我很想要这个函数的英语版本，如果你们能够开发出来，我就会购买Excel。”就算你真把电话打到微软公司，下面就是你会得到的回答：

“欢迎您致电微软公司。如果您有四位数的促销码，请按1。如果您需要微软产品的技术支持服务，请按2。预售许可或相关信息，请按3。人工服务请按4。重听请按*号键。”

看到了吗？并没有一个选项是“购买前要求增加功能，请按5”。

“定制软件”是一个暗无天日的世界。先是顾客提出他想要什么，你问他是否确定，他回答是的。你就写出一份绝对漂亮的软件规格说明，问他：“你要的是不是这样的软件？”他回答是的。你要他在文件上签名，并且用人头担保，他也照做了。然后按照他的要求，你快速准确地把软件做出来了。但是，顾客看到软件的时候被吓坏了，简直惊恐万分，你就只好把这个星期剩下的时间都用来研究你向保险公司购买的“错误和遗漏险”（E&O insurance）能否弥补打官司的费用，或者是否足够用来与顾客达成和解。要是你运气真的很好的话，顾客看到软件的时候会勉强挤出一丝惨淡的笑容，然后就把你的软件锁在抽屉里，不再拿出来，也不再回你的电话了。

“个性化软件”与“定制软件”并没有本质区别。虽然表面上你好像在制作面向整个市场销售的上架软件，但是实际上还是在为顾客定制软件。下面就是个性化软件的开发过程。

(1) 一家制鞋公司雇用你为它开发软件。

(2) 这家公司需要擦鞋软件。

(3) 你用VB 3.0写出了擦鞋软件，并且还使用了一点JavaScript和Franz Lisp，以及一个在老式Mac电脑上运行、用AppleScript连上网络的FileMaker数据库。

(4) 用过的人都说好，你开始梦想开一家自己的软件公司，没准能成为比尔·盖茨或者Larry Ellison[2]也行。

(5) 你向原先服务的公司买下了擦鞋软件1.0版的版权，在风险投资的支

① Baht是泰国的货币单位铢。

② Larry Ellison是甲骨文公司的创始人兼总裁，以生活奢侈著称。

持下开了一家擦鞋软件有限公司，专门推销这个软件。

(6) 但是beta测试的时候，没有一个参加者能运行这个软件，因为它始终离不开AppleScript，而且还在源代码中把IP地址写死了，解决这些问题花了一个月。

(7) 你发现很难吸引顾客购买你的软件，因为安装成本非常高，需要专门购买一台运行System 7[①]的Macintosh IIci型号计算机[②]，这种计算机是如此古旧，以至于只能在拍卖网站eBay上向计算机博物馆购买，这意味着你的软件十分昂贵。投资给你的风险投资家们开始变得非常紧张。

(8) 他们向销售人员施加压力。

(9) 销售人员发现，有一个你的潜在客户需要的不是擦鞋软件，而是熨裤子的软件。

(10) 销售人员使出看家本领，让那个客户花10万美元用了一套熨裤子软件。

(11) 接下来6个月，你只好为那一个客户写出了定制的“熨裤子”模块。

(12) 没有任何其他客户需要这个功能。

(13) 实际上，你的这一年就相当于，先从风险投资家那里搞到一笔钱，然后被一家生产裤子的公司雇用，为它写出一个定制软件，一切又回到了步骤(1)。

这可真是太折腾人了，我必须建议，你最好尽一切可能坚持制作面向整个市场销售的上架软件。因为上架软件增加顾客的边际成本为零，所以你就是在把同一件东西一遍又一遍地卖出去，赚到多得多的利润。而且你还能够降低价格，因为你可以把开发成本分摊到大量顾客头上。低廉的价格使得你得到更多的顾客，许多人会突然发现你的软件是那么便宜那么物有所值。生活从此变得美好甜蜜。

所以，如果你猛然察觉自己开发某个功能只是因为答应了某个顾客，那么你正在滑向“个性化软件”和“定制软件”的不归路。如果你自己感觉不错，那么那个地方也没什么不好，只是它不如现成的上架软件有那么大的获利空间罢了。

---

① System 7又称Mac OS 7，是一种用于Macintosh计算机的单用户图形界面操作系统，发布于1991年5月13日。

② Macintosh IIci是苹果公司生产的一种个人电脑，发布于1989年9月20日。

当然，我不是说你不应该听取顾客的意见。站在我的角度，其实我也认为微软应该开发BAHTTEXT函数的其他语言版本，为那些还没有加入全球经济圈并使用泰国语的人和使用其他国家的货币开支票的人提供方便。事实上，如果你认为最好的分配资源方法是让大客户“投标”决定需要哪些功能，不错，你可以这样做，但是你很快就会发现，那些富有的大客户所需要的功能与市场所需要的功能是不一样的。你加入软件的那些处理泰铢的功能也不会有助于你将Excel卖到亚利桑那州Scottsdale[①]的温泉疗养中心。事实上，你听命于大客户的实际效果，只是让销售人员指挥开发人员，达到他们个人销售佣金最大化的目标。

沿这条路走下去，你会成为比尔·盖茨吗？不可能的。

现在，让我告诉你决定部署哪些功能时不能采用的第二种方法。不要因为有些事情不得不做，你就去做。不得不做并不是一个足够好的理由。请听我解释。

Fog Creek创立后的第一年，有一次我在整理文件，发现蓝色的文件夹用完了。

我整理文件的规则是这样的：蓝色文件夹存放客户资料，棕色文件夹存放员工资料，红色文件夹存放收据，黄色文件夹存放其他文件。我现在需要一个蓝色文件夹，但是已经没有多余的了。

我就对自己说，“反正我总是需要蓝色文件夹的，那么我现在不妨就到文具店去一次，买一些回来。”

当然，我这样做就浪费了时间。

事实上，我后来回想这件事，意识到长久以来，我一直蠢透了。只要我觉得某件事是摆脱不掉的，我就想那不如现在就把它做完。

我以此为借口，拔去花园中的杂草，修补墙上的小洞，整理MSDN光盘（根据颜色、语言和编号），等等。我本应该把这些时间用来写代码或者卖代码，它们才是创业公司真正需要做的仅有的两件事。

① 在美国，Scottsdale被认为是一个高档次高消费的旅游目的地。

换言之，我发现自己一直在假装，好像所有必须要做的事情都是同等重要的。既然是同等重要，而且又迟早要做，那么按什么顺序做就无所谓了！现在就搞定！

而实际上，我只是在耽误时间。

那么我应该怎么做呢？好的，首先，我要克服自己的偏好，文件夹没必要一定要蓝色的，颜色是无所谓的。文件不一定要用不同颜色的文件夹分类。

嗯，那些MSDN的CD-ROM呢？全部扔进一个大箱子。完美解决。

更重要的是，我早就应该想到“重要性”并不是二进制的数字制式，而是模拟制式。不同的事情有不同程度的“重要性”，而不是只有“重要”和“不重要”的两个选项。如果你想把所有事情都做完，最后只会一事无成。

所以，如果你想把事情做完，无论何时，你一定要想清楚什么是眼下最重要的、必须马上做好的事。如果你不做这件事，你就不能以最快的速度取得进展。

渐渐地，我摆脱了做事拖拉的毛病。我的方法就是，不去理会那些相对不重要的事，把它们留在那里。某一位保险公司的热心女士纠缠了我两个月，要求我提供某些数据更新我的保单，直到她第十五次打来电话，我才把数据找出来给她，因为她警告说保险合同将在三天后失效。我感觉这样还不错。我已经发展到认为，被收拾得干干净净的办公桌可能是一个信号，表明你的工作效率不高。

这种想法说出来，真是太不好意思了！

所以，不要开发那些销售员无意间答应某个客户的功能，也不要因为“反正迟早要做”而先去开发那些不重要尽管有趣味性的功能。

好了，回到为FogBugz 5.0版本选择功能的话题。下面是我们怎样定出初步的优先顺序。

首先，我拿出一叠卡片，每一张上面写下一个功能。接着，我召集整个团队。根据我的经验，这种方法大概最多在不超过20个人的时候是有效的。另外，不同观点的人越多越好：程序员、设计师、客服人员、销售人员、管理人员、文档作者、测试员，甚至还可以包括客户。

在开会之前，我要求每个人都想好他们自己想要实现的功能。会议开始以后，第一件事就是很快地让每个人把自己想要的功能展示一下，确保大家对不同功能的含义大致上有一个非常粗略的共同理解。每个功能都有一张对应的卡片。

在这个阶段，主要目的不是争论各种功能的优劣，也不是对功能进行设计，甚至不是开展讨论，而是让会议参加者对不同功能有一个模糊粗略的印象。FogBugz 5.0版本的部分功能如下：

- 个性化首页；
- 轻松安排软件开发进程；
- 追踪资金到账时间；
- 允许对bug分叉（fork）；
- （46个其他功能……）

这时，它们都还很模糊。记住，我们此时不需要知道如何实现每个功能，或者每个功能会牵扯到什么问题，因为我们的唯一目标是得到一个粗略的优先顺序，用来作为基础，启动软件的开发。通过这个阶段，我们得到了一份50种重大功能的清单。

在会议的第二阶段，我们审查所有这些功能，要求大家对每个功能进行投票表决，简单地表示“赞成”或“反对”即可。不要讨论，也不要做其他事情，只要很快地对每个功能表示是不是赞成。这个过程显示，大概有14种功能没有什么人支持。我把所有那些只得到一票或两票的功能都拿掉，剩下了36个功能。

下一步，我们为每个功能设定一个成本，用1到10表示，1表示这个功能可以轻而易举地部署，10表示这个功能是超级的庞然大物，部署起来很困难。这时，一定要记得，这一步的目标不是对功能拟定开发的日程表，而是区分小型任务、中型任务和大型任务。我一个功能一个功能地问程序员，要求他们说出这是小型任务、中型任务、还是大型任务。对于程序员来说，即使不知道每个功能要花多少时间来开发，也很容易看出“允许对bug分叉”是一个小型任务，而庞杂模糊的“个性化首页”是一个大型任务。根据大家的共同估计以及我个人的判断，我们在每一个功能后面写下它的价格：

功能	费用
个性化首页	$10
轻松安排软件开发进程	$4
追踪资金到账时间	$5
允许对bug分叉	$1

再说一遍，这种方法确实有点乱七八糟，也不够准确，但是没有关系。你今天所做的不是制定开发日程表，而只是排出各种功能的优先顺序。你唯一必须大致正确把握的事情就是，粗略知道在基本相同的一段时间内，你可以完成二项中型任务，或者一项大型任务，或者十项小型任务。这种换算关系不需要十分精确。

下一步，就是把所有36项提出来的功能和它们的“费用”做成一张菜单，参加会议的每个人发一份。同时还规定每个人有50美元，可以用来“点菜”。你可以用任何你想要的方式分配这50美元，但是你只有这50美元可用。你可以买半个功能，也可以同样的功能买两个。如果你喜欢“追踪资金到账时间”功能，你可以为它花掉10美元或15美元，如果你不是很喜欢，也可以只为它花掉1美元，留给其他人为它投入足够资金。

然后，我们把每个人在每项功能上花掉的钱加总起来：

功能	费用	销售额
个性化首页	$10	$12
轻松安排软件开发进程	$4	$6
追踪资金到账时间	$5	$5
允许对bug分叉	$1	$3

最后，我用“销售额”除以“费用”：

功能	费用	销售额	
个性化首页	$10	$12	1.2
轻松安排软件开发进程	$4	$6	1.5
追踪资金到账时间	$5	$5	1.0
允许对bug分叉	$1	$3	3.0

以这个值排序，找出最受欢迎的功能：

功能	费用	销售额	
允许对bug分叉	$1	$3	3.0
轻松安排软件开发进程	$4	$6	1.5
个性化首页	$10	$12	1.2
追踪资金到账时间	$5	$5	1.0

完成了！这是一份你可能部署的所有功能的清单，根据大家对最重要功能的大致判断进行排序。

现在你可以开始做一些调整，将那些相关的功能关联在一起，比如“安排软件开发进程”可以更容易地“追踪资金到账时间”，所以这两个功能也许应该都部署，或者都不部署。有时候，顺着这张优先级清单往下看，有些地方很明显搞错了，你就把它们改正过来！没有什么东西是不能修改的。即使在开发过程中，你也还可以改变不同任务的优先顺序。

不过，最让我惊讶的是，对FogBugz 5.0版本来说，我们最后得到的任务清单确实是非常好的优先顺序，真实地反映了我们这个开发团队对各种任务优先顺序的共识。

优先顺序确定以后，我们就会基本按照这张清单从上往下进行开发，一直到明年三月份为止。届时我们计划停止增加新功能，开始整合和测试阶段。在部署每一个（不是凭直觉就会用的）功能之前，我们就为它写好规格说明书。

（在软件开发的BDUF方法/敏捷方法[①]的选美大赛中，唠唠叨叨的计分员这一次彻底糊涂了。“你的这一票到底算是投给BDUF？还是投给敏捷？你到底选谁？能不能一次只选一个？”）

整个优先顺序的决定过程花掉了三个小时。

即使你很幸运地拥有非凡的能力，能够比我们更频繁地发布软件（参见第33讲），你依然需要根据优先顺序清单，从上往下逐项开发软件的新功能。但是，你能够更经常性地停下来，更频繁地发布新版本。这样做的好处是，你可以根据顾客的实际反馈经常重新调整各种开发任务的优先顺序，但是并不是每个产品都有机会得到这样的待遇。

Mike Conte教会了我使用这种方法。那是在Excel 5的规划阶段，虽然会

① BDUF方法和敏捷方法是两种对立的软件开发模式。BDUF（Big Design Up Front）方法强调设计先行，要求在做出完备可靠的设计之后，再开始进行软件的具体实现。而敏捷方法认为软件开发是一个逐步的过程，必须经常做出调整，应对外界快速变化的需求。与BDUF方法相比，敏捷方法更强调程序员团队与业务专家之间的紧密协作、面对面的沟通（认为比书面的文档更有效）、频繁交付新的软件版本。

议室里有几十个人，但是也只花了几个小时就决定了要开发哪些功能以及它们的优先顺序。好玩的地方在于，其中大概有50%的功能，我们后来没有时间开发，但是事实证明，它们都是很蠢的功能，Excel没有它们反而更好。

这种方法并不完美，不过我告诉你，它比去文具店买蓝色文件夹要好多了。

程序员最可信赖的求职帮手