Eric Steven Raymond

The Cathedral and the Bazaar

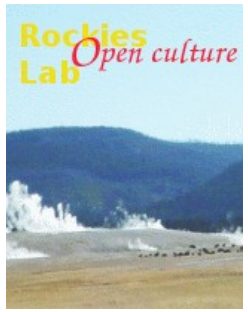
ABSTRACT

I anatomize a successful open-source project, fetchmail, that was run as a deliberate test of the surprising theories about software engineering suggested by the history of Linux. I discuss these theories in terms of two fundamentally different development styles, the ``cathedral" model of most of the commercial world versus the ``bazaar" model of the Linux world. I show that these models derive from opposing assumptions about the nature of the software-debugging task. I then make a sustained argument from the Linux experience for the proposition that "Given enough eyeballs, all bugs are shallow", suggest productive analogies with other self-correcting systems of selfish agents, and conclude with some exploration of the implications of this insight for the future of software.

$Date: 2002/08/02 09:02:14 $
Source: http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/

艾里克·斯蒂芬·雷蒙

大教堂和市集



摘要

Linux 的发展史促生了一些关于软件工程的惊人理论。我有意的在一个成功的开源项目 fetchmail 中测试了这些理论，并在此加以剖析。这里讨论了两 种根本上不同的开发模式：大多数商业项目使用的"大教堂"模式和 Linux 世界的"市集"模式。我们将看到，这两种模式源于对软件调试工作的本质的两种彼 此对立的假设。我接着从 Linux 的经验出发，对"只要眼球足够多，所有臭虫都好捉"的定理作了一个站得住的论证；建议它与其它由自主成员组成的自纠错系统之间富有意义的相似之处。最后，我探讨了这个发现对未来软件业的启示。

洛基开放文化实验室中译本 v1.1
译文按原文使用的 OPL v2.0 发布。中译本的主页在
http://rl.rockiestech.com/node/101
欢迎批评改进。

译序

开源软件和开放性内容兴起的背后是社会信息结构的变革。技术和知识在公共领域的畅通促进发展、公平和机遇，破除与经济和政治权力绑结的知识垄断。然而草根能量需要一个健康的进化机制来真正推动社会的进步。其中的核心是知识生产和传播的可靠性、可信度。

这个《大教堂和市集》的新译本就是洛基开放文化实验室所作努力的一部分。像 RL 的所有项目一样，欢迎每个人的参与和批评。这个版本由 habpi 主译；根据最新的英文版本，比网上流传的中文版本增加了一些内容，也作了很多修正。原文中的长句很多，我们不得不在中文里作了一些结构调整，力求在准确表达作者原意的同时保证句子通畅。然而现在的版本还是不够通俗；而且限于水平和时间，其中的错漏之处是难免的。我们真诚希望通过各界朋友的批评指正来提高，同时也欢迎大家就相关的主题进行讨论。这个版本没有翻译注释、文献和致谢部分，希望有人力把它们在未来的版本中完成。感谢网友 feiyue999、虎子、lawrence、bingo 等人（恕不一一提及）的指正。

有关讨论、核对、反馈和版本升级，请访问这个项目的永久网址 http://rl.rockiestech.com/node/101。

~~~~~~

[http://rl.rockiestech.com]

洛基开放文化实验室，使用开源方法来推动社会文化进步

目录
~~~~~~

Linux is subversive. Who would have thought even five years ago (1991) that a world-class operating system could coalesce as if by magic out of part-time hacking by several thousand developers scattered all over the planet, connected only by the tenuous strands of the Internet?

Certainly not I. By the time Linux swam onto my radar screen in early 1993, I had already been involved in Unix and open-source development for ten years. I was one of the first GNU contributors in the mid-1980s. I had released a good deal of open-source software onto the net, developing or co-developing several programs (nethack, Emacs's VC and GUD modes, xlife, and others) that are still in wide use today. I thought I knew how it was done.

Linux overturned much of what I thought I knew. I had been preaching the Unix gospel of small tools, rapid prototyping and evolutionary programming for years. But I also believed there was a certain critical complexity above which a more centralized, a priori approach was required. I believed that the most important software (operating systems and really large tools like the Emacs programming editor) needed to be built like cathedrals, carefully crafted by individual wizards or small bands of mages working in splendid isolation, with no beta to be released before its time.

Linus Torvalds's style of development—release early and often, delegate everything you can, be open to the point of promiscuity—came as a surprise. No quiet, reverent cathedral-building here—rather, the Linux community seemed to resemble a great babbling bazaar of differing agendas and approaches (aptly symbolized by the Linux archive sites, who'd take submissions from anyone) out of which a coherent and stable system could seemingly emerge only by a succession of miracles.

Linux 是颠覆性的。就是五年以前(1991)，谁能想得到散布在全球各地的几千名开发者的业余敲打，仅靠细细的互联网网线连接，能够魔术一般地铸成一个世界级的操作系统呢?

反正不是我。在 1993 年初 Linux 引起我的注意的时候，我已经在 Unix 和开放源代码开发领域做了十年了。我是 80 年代中期最早的 GNU 开发者之一。 我已经在网上发布了相当一部分软件，正在开发或协助开发好几个直到今天都在广泛使用的软件（nethack，Emacs 的 VC 和 GUD 模式，xlife 和 其它）。我觉得我很懂行了。

Linux 颠覆了许多我以为我懂的东西。多年来我一直在宣扬小型工具、快速建模和进化式编程的 Unix 福音。但我也相信一个项目到了一定的复杂程度后就需 要更集中地按事先计划管理。我相信最重要的软件（操作系统和 Emacs 之类的大型工具）需要像大教堂一样来搭建：遗世独立的圣人巨匠们牵尺引斤琢之磨之； 时候不到 beta 版不出。

林纳斯·托瓦兹（Linus Torvalds）的开发风格令人惊讶：尽早尽多的发布，委托所有可以委托的事，开放到了泛滥的程度。这里没有建造大教堂的安静和虔诚；Linux 社区更像一个充满不同议程和方法的嘈杂的大集市（Linux 归档站点们就是一个绝好的例子，任何人的作品都接收）。一个统一稳定的系统若是从这儿产生看来只能依靠一系列的奇迹。

The fact that this bazaar style seemed to work, and work well, came as a distinct shock. As I learned my way around, I worked hard not just at individual projects, but also at trying to understand why the Linux world not only didn't fly apart in confusion but seemed to go from strength to strength at a speed barely imaginable to cathedral-builders.

By mid-1996 I thought I was beginning to understand. Chance handed me a perfect way to test my theory, in the form of an open-source project that I could consciously try to run in the bazaar style. So I did—and it was a significant success.

This is the story of that project. I'll use it to propose some aphorisms about effective open-source development. Not all of these are things I first learned in the Linux world, but we'll see how the Linux world gives them particular point. If I'm correct, they'll help you understand exactly what it is that makes the Linux community such a fountain of good software—and, perhaps, they will help you become more productive yourself.

结果这种市集风格的确有效、非常有效——真是一个绝大的震撼。在我摸索的过程中，我不仅效力于个别的项目，而且努力去理解为什么 Linux 世界没有在混乱中分崩离析，而是以大教堂的建造者们难以想像的速度茁壮成长。

到１９９６年中，我想我开始理解了。我有了一个测试我的理论的完美机会，一个我可以有意识的用市集风格来运行的开源项目。我这样做了——结果非常成功。

这里讲述的就是这个项目的故事。我将借它来提出一些开源软件有效开发的精髓。它们并非全部源自 Linux 世界，但我们会看到它们如何在 Linux 世界中得到印证。如果我是正确的话，它们会帮助您准确理解什么使得 Linux 社区成为优秀软件的源泉——或许，它 们还会帮助您变得更加高效。

The Mail Must Get Through

Since 1993 I'd been running the technical side of a small free-access Internet service provider called Chester County InterLink (CCIL) in West Chester, Pennsylvania. I co-founded CCIL and wrote our unique multiuser bulletin-board software—you can check it out by telnetting to locke.ccil.org. Today it supports almost three thousand users on thirty lines. The job allowed me 24-hour-a-day access to the net through CCIL's 56K line—in fact, the job practically demanded it!

I had gotten quite used to instant Internet email. I found having to periodically telnet over to locke to check my mail annoying. What I wanted was for my mail to be delivered on snark (my home system) so that I would be notified when it arrived and could handle it using all my local tools.

The Internet's native mail forwarding protocol, SMTP (Simple Mail Transfer Protocol), wouldn't suit, because it works best when machines are connected full-time, while my personal machine isn't always on the Internet, and doesn't have a static IP address. What I needed was a program that would reach out over my intermittent dialup connection and pull across my mail to be delivered locally. I knew such things existed, and that most of them used a simple application protocol called POP (Post Office Protocol). POP is now widely supported by most common mail clients, but at the time, it wasn't built in to the mail reader I was using.

I needed a POP3 client. So I went out on the Internet and found one. Actually, I found three or four. I used one of them for a while, but it was missing what seemed an obvious feature, the ability to hack the addresses on fetched mail so replies would work properly.

邮件必须通过

从 1993 年以来，我在负责宾州西切斯特的一家提供免费网络服务的小公司 CCIL 的技术工作。我协同创建了 CCIL，并写了我们独家的多用户论坛软 件——您可以用 telnet 连接 locke.ccil.org 来试一下。今天它在三十条线上支持近三千名用户。这份工作允许我通过 CCIL 的 56K 的线路 每天二十四小时上网——其实，这份工作事实上要求这一点！

我已经习惯于使用即时的互联网邮件。我发现不时地要 telnet 登录上公司服务器"locke"检查邮件很烦人。我想要的是把我的邮件传送到我家里的机器"snark"上，这样我可以在邮件到达的时候得到通知，使用本地工具来处理它。

互联网的原装邮件输送协议 SMTP 不适用，因为它最好在机器全时在线的情况下使用，而我的个人机器并不总在网上，也没有一个静态的 IP 地址。我需要 一个程序在我拨号上网的期间连到服务器上去，把我要下到本地的邮件取回来。我知道有这类东西存在，多数使用一个简单的应用协议 POP。现在多数的常用客户 端邮件软件都支持 POP，但那个时候，它并不在我用的邮件阅读器里。

我需要一个 POP3 的客户端软件。所以我就跑到网上找了一个。事实上，我找到了三四个。其中的一个我用了一段时间，但它少了一个看起来很明显的功能：提取到达邮件的来信地址以便正确回信。

The problem was this: suppose someone named `joe' on locke sent me mail. If I fetched the mail to snark and then tried to reply to it, my mailer would cheerfully try to ship it to a nonexistent `joe' on snark. Hand-editing reply addresses to tack on <@ccil.org> quickly got to be a serious pain.

This was clearly something the computer ought to be doing for me. But none of the existing POP clients knew how! And this brings us to the first lesson:

1. Every good work of software starts by scratching a developer's personal itch.

Perhaps this should have been obvious (it's long been proverbial that ``Necessity is the mother of invention") but too often software developers spend their days grinding away for pay at programs they neither need nor love. But not in the Linux world—which may explain why the average quality of software originated in the Linux community is so high.

So, did I immediately launch into a furious whirl of coding up a brand-new POP3 client to compete with the existing ones? Not on your life! I looked carefully at the POP utilities I had in hand, asking myself ``Which one is closest to what I want?" Because:

2. Good programmers know what to write. Great ones know what to rewrite (and reuse).

While I don't claim to be a great programmer, I try to imitate one. An important trait of the great ones is constructive laziness. They know that you get an A not for effort but for results, and that it's almost always easier to start from a good partial solution than from nothing at all.

问题是这样的：假设"locke"上一个叫"乔"的人给我发了信。如果我把信取到"snark"上，然后试图回复，我的邮件程序会高高兴兴地努力把回信发送给"snark"上一个并不存在的"乔"。通过手工修改回信地址给邮件重新导向很快就成了很痛苦的事。

显然这该是电脑替我做的事。但是现有的 POP 客户端软件没有一个会做！这给我们带来了第一个教训：

1）每一个好的软件的起因都是挠到了开发者本人的痒处

这或许应该是很显然的（一直有箴言道是"需要是发明之母"），但软件开发人员太过经常地在那些他们既不需要也不喜欢的程序上消磨时日、换取工资。但在 Linux 世界不是这样子的——这或许解释了为什么 Linux 社区中产生的软件平均质量这么高。

那么，我立马儿投入到了一轮疯狂的编码来写一个和现有 POP3 客户竞争的软件了吗？打死你都不会！我仔细检查了我拿到手的那些 POP 程序，自问"哪一个离我要的最接近？"因为：

2）好的程序员知道写什么。伟大的程序员知道改写（和重复使用）什么。

虽然我不自封为伟大的程序员，但我努力模仿伟大的程序员。伟大者的一个重要特点是建设性的懒惰。他们知道你需要的是结果不是过程，而且从一个好的部分方案开始总比从零开始要容易得多。

Linus Torvalds, for example, didn't actually try to write Linux from scratch. Instead, he started by reusing code and ideas from Minix, a tiny Unix-like operating system for PC clones. Eventually all the Minix code went away or was completely rewritten—but while it was there, it provided scaffolding for the infant that would eventually become Linux.

In the same spirit, I went looking for an existing POP utility that was reasonably well coded, to use as a development base.

The source-sharing tradition of the Unix world has always been friendly to code reuse (this is why the GNU project chose Unix as a base OS, in spite of serious reservations about the OS itself). The Linux world has taken this tradition nearly to its technological limit; it has terabytes of open sources generally available. So spending time looking for some else's almost-good-enough is more likely to give you good results in the Linux world than anywhere else.

And it did for me. With those I'd found earlier, my second search made up a total of nine candidates—fetchpop, PopTart, get-mail, gwpop, pimp, pop-perl, popc, popmail and upop. The one I first settled on was `fetchpop' by Seung-Hong Oh. I put my header-rewrite feature in it, and made various other improvements which the author accepted into his 1.9 release.

A few weeks later, though, I stumbled across the code for popclient by Carl Harris, and found I had a problem. Though fetchpop had some good original ideas in it (such as its background-daemon mode), it could only handle POP3 and was rather amateurishly coded (Seung-Hong was at that time a bright but inexperienced programmer, and both traits showed). Carl's code was better, quite professional and solid, but his program lacked several important and rather tricky-to-implement fetchpop

以林纳斯·托瓦兹为例，他实际上没有试图从头来写 Linux。相反，他开始于再用 Minix——一个小小的在 PC 机上的类 UNIX 系统——的代码和主意。最终所有 Minix 的代码都被拿掉或重写了——但在起步的阶段，Minix 提供了那个最后成为 Linux 的新生儿成长的脚手架。

遵循同样的精神，我出发去寻找一个已有的、写得过得去的 POP 程序来作为开发的基础。

UNIX 世界里的源代码共享传统一直对代码再用很友好（这是为什么 GNU 项目尽管对 UNIX 很有成见，还是选择了 UNIX 作为基本操作系统）。LINUX 世界几乎把这种传统发挥到了技术上的极限；有上万亿字节的开放代码可供获取。所以花点时间在 LINUX 世界里找个别人“差不多够好”的程序，是 比其它任何地方都更有可能找到的。

我就找到了。加上我以前找到的，我的第二次搜索有了九个候选对象：fetchpop，PopTart，get－mail，gwpop，pimp，pop-perl，popc，popmail 和upop。我第一个选用的是欧松宏（音，Seung-Hong Oh）的fetchpop。我把我的改写邮件头的功能加了进去，并作了其它一些改进。作者后来把这些加进了他的 1.9 版本。

然而几个星期以后，我碰到了卡尔·哈里斯的“popclient”代码，发现我遇到了一个问题。尽管 fetchpop 有一些很好的新主意（例如它的 后台 daemon 模式），它只能处理 POP3 协议，而且程序代码写的比较业余（松宏当时是个聪明但是缺少经验的程序员，这两个特点都有显示）。卡尔的代码 好一些，很专业和稳固，但他的程序缺几个重要的而且难实现的 fetchpop 里的功能（包括我自己写的那些）。

features (including those I'd coded myself).

Stay or switch? If I switched, I'd be throwing away the coding I'd already done in exchange for a better development base.

A practical motive to switch was the presence of multiple-protocol support. POP3 is the most commonly used of the post-office server protocols, but not the only one. Fetchpop and the other competition didn't do POP2, RPOP, or APOP, and I was already having vague thoughts of perhaps adding IMAP (Internet Message Access Protocol, the most recently designed and most powerful post-office protocol) just for fun.

But I had a more theoretical reason to think switching might be as good an idea as well, something I learned long before Linux.

3. ``Plan to throw one away; you will, anyhow." (Fred Brooks, The Mythical Man-Month, Chapter 11)

Or, to put it another way, you often don't really understand the problem until after the first time you implement a solution. The second time, maybe you know enough to do it right. So if you want to get it right, be ready to start over at least once [JB].

Well (I told myself) the changes to fetchpop had been my first try. So I switched.

After I sent my first set of popclient patches to Carl Harris on 25 June 1996, I found out that he had basically lost interest in popclient some time before. The code was a bit dusty, with minor bugs hanging out. I had many changes to make, and we quickly agreed that the logical thing for me to do was take over the program.

继续用 fetchpop 还是转换到 popclient 上来？如果转换的话，我是扔掉我已经写好的那些代码来换取一个好一些的开发基础。

一个实用的转换动机是对多种协议的支持。POP3 是服务器端 POP 协议中最常用的，但不是唯一的。fetchpop 和那一个竞争对手都不支持 POP2、RPOP 或 APOP，而我已经有了为了好玩添加 IMAP（最新设计的、最强大的 POP 协议）的模糊想法。

但我还有一个更理论上的原因来认为转换也是个好主意。这是我远在 Linux 之前就学到的。

3）“计划扔掉一个；无论如何你都会扔掉一个的。”（弗里德·布洛克《人月神话》第 *11* 章）

或者换句话说，直到你第一次实现一个方案之前，你常常并没有真正理解你的问题。第二次呢，或许你已经学到了如果把它做对。所以你要是想把事情做对的话，准备好至少重来一次。

好吧（我对自己说），对 fetchpop 做的修改算我的第一次吧。于是我转换了。

在 1996 年 6 月 25 日我给卡尔·哈里斯发送了我写的第一批 popclient 的补丁后，我发现他一段时间之前就基本上对这个项目失掉兴趣了。项目的源代码有些陈旧了，小臭虫们流连不去。我有很多要修改的东西；我们很快同意我把整个项目接手过来是理所当然了。

Without my actually noticing, the project had escalated. No longer was I just contemplating minor patches to an existing POP client. I took on maintaining an entire one, and there were ideas bubbling in my head that I knew would probably lead to major changes.

In a software culture that encourages code-sharing, this is a natural way for a project to evolve. I was acting out this principle:

4. If you have the right attitude, interesting problems will find you.

But Carl Harris's attitude was even more important. He understood that

5. When you lose interest in a program, your last duty to it is to hand it off to a competent successor.

Without ever having to discuss it, Carl and I knew we had a common goal of having the best solution out there. The only question for either of us was whether I could establish that I was a safe pair of hands. Once I did that, he acted with grace and dispatch. I hope I will do as well when it comes my turn.

在我没有觉察的时候，这个项目升级了。我不再是试图给一个现有的 POP 客户端程序做点儿小补丁。我负责起了维护整个程序，而且我知道我脑子里冒着的新主意可能会导致一些主要的变动。

在一个鼓励代码共享的软件文化中，这是一个项目进化的自然方式。我在实践这一个原理：

4）如果你有正确的态度，有意思的问题会找到你。

卡尔·哈里斯的态度甚至更重要。他懂得：

5）当你对一个项目失去兴趣时，你的最后的职责是把它交给一个称职的继承者。

尽管卡尔和我从来没有必要讨论过这一点，我们知道我们的共同目标是作出一个目前最好的程序。我们唯一的问题是我能否证明我的可靠性。一旦我作到了，他优雅而迅速地作了交接。我希望当这一天轮到我的时候，我也能做得同样出色。

The Importance of Having Users

And so I inherited popclient. Just as importantly, I inherited popclient's user base. Users are wonderful things to have, and not just because they demonstrate that you're serving a need, that you've done something right. Properly cultivated, they can become co-developers.

Another strength of the Unix tradition, one that Linux pushes to a happy extreme, is that a lot of users are hackers too. Because source code is available, they can be effective hackers. This can be tremendously useful for shortening debugging time. Given a bit of encouragement, your users will diagnose problems, suggest fixes, and help improve the code far more quickly than you could unaided.

6. Treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging.

The power of this effect is easy to underestimate. In fact, pretty well all of us in the open-source world drastically underestimated how well it would scale up with number of users and against system complexity, until Linus Torvalds showed us differently.

In fact, I think Linus's cleverest and most consequential hack was not the construction of the Linux kernel itself, but rather his invention of the Linux development model. When I expressed this opinion in his presence once, he smiled and quietly repeated something he has often said: ``I'm basically a very lazy person who likes to get credit for things other people actually do." Lazy like a fox. Or, as Robert Heinlein famously wrote of one of his characters, too lazy to fail.

用户的重要性

就这样，我继承了 popclient。同样重要的是，我继承了 popclient 的用户群。拥有用户是件美好的事情，不仅因为他们证实了你满足了一种需要，而且你把事情作对了。在适当培养下，他们可以成为共同开发者。

UNIX 传统中的另一个强项，Linux 把它发展到快乐极致的一个，是很多用户也是黑客。因为可以得到源代码，他们可以是有效的黑客。这一点对缩短调试时间会非常的有帮助。有一点点鼓励，你的用户们会诊断问题，提出建议和补丁，并以你一个人不可企及的速度帮助改进代码。

6）把用户像合作者来对待是通往快速改进代码和有效调试的最佳通道

这一点所蕴藏的能量很容易被低估。事实上，直到林纳斯·托瓦兹给我们演示了之前，我们开源世界里的几乎所有人都严重低估了它如何随用户数目而增长，不论系统多么复杂。

事实上，我认为林纳斯的最聪明、最有影响的手笔不是建设 Linux 核心本身，而是发明了 Linux 的开发模式。当我一次在他的面前表达了这个观点 时，他微笑了，安静地重复了他经常说的一句话：“我基本上是一个很懒惰的人，喜欢在其实是别人做的事情上领取荣誉”。象狐狸一样懒惰。或者象 Robert Heinlein 著名地描写他的一个角色：太懒惰而不会失败。

In retrospect, one precedent for the methods and success of Linux can be seen in the development of the GNU Emacs Lisp library and Lisp code archives. In contrast to the cathedral-building style of the Emacs C core and most other GNU tools, the evolution of the Lisp code pool was fluid and very user-driven. Ideas and prototype modes were often rewritten three or four times before reaching a stable final form. And loosely-coupled collaborations enabled by the Internet, a la Linux, were frequent.

Indeed, my own most successful single hack previous to fetchmail was probably Emacs VC (version control) mode, a Linux-like collaboration by email with three other people, only one of whom (Richard Stallman, the author of Emacs and founder of the Free Software Foundation) I have met to this day. It was a front-end for SCCS, RCS and later CVS from within Emacs that offered ``one-touch'' version control operations. It evolved from a tiny, crude sccs.el mode somebody else had written. And the development of VC succeeded because, unlike Emacs itself, Emacs Lisp code could go through release/test/improve generations very quickly.

The Emacs story is not unique. There have been other software products with a two-level architecture and a two-tier user community that combined a cathedral-mode core and a bazaar-mode toolbox. One such is MATLAB, a commercial data-analysis and visualization tool. Users of MATLAB and other products with a similar structure invariably report that the action, the ferment, the innovation mostly takes place in the open part of the tool where a large and varied community can tinker with it.

回头来看，Linux 的方法和成功的一个先例是 GNU Emacs 的 Lisp 库和 Lisp 代码档案。与 Emacs C 核心和大多数的其它 GNU 工具的大教堂建造风格相反，Lisp 的代码群的进化是活跃的、多由用户驱动的。主意和草稿模型经常要重写三四次才能达到一个稳 定的最终形式。象 Linux 那种通过互联网的松散的协作也很频繁。

确实，我自己在 fetchmail 之前最成功的一次编程可能是 Emacs 的 VC（版本控制）模式。那是与其他三个人通过电子邮件象 Linux 一样的 一次合作。三个人中我至今只见过一个（Richard Stallman，Emacs 的作者、自由软件基金会的创始人）。VC 是 Emacs 中 SCCS，RCS 和后来 CVS 的前台；Emacs 借此以提供“单击 式”的版本控制操作。它是由一个别人写的小小的、粗糙的 sccsl.el 模式演进而来。VC 开发的成功也是因为 Emacs Lisp 代码不象 Emacs 本身那样，可以快速地通过多轮“发行／测试／改进”的循环。

Emacs 的故事不是唯一的。其它的软件也有这种双层的构架和双层的用户群：核心用大教堂模式；工具箱用市集模式。其中的一个是 MATLAB，一个 数据分析和呈现的商业性工具。MATLAB 和其它类似架构产品的用户一致报告说，产品的开放部分——有一个巨大多样的用户群可以推敲的地方——才是动力、 热情和创新的所在。

Release Early, Release Often

Early and frequent releases are a critical part of the Linux development model. Most developers (including me) used to believe this was bad policy for larger than trivial projects, because early versions are almost by definition buggy versions and you don't want to wear out the patience of your users.

This belief reinforced the general commitment to a cathedral-building style of development. If the overriding objective was for users to see as few bugs as possible, why then you'd only release a version every six months (or less often), and work like a dog on debugging between releases. The Emacs C core was developed this way. The Lisp library, in effect, was not—because there were active Lisp archives outside the FSF's control, where you could go to find new and development code versions independently of Emacs's release cycle [QR].

The most important of these, the Ohio State Emacs Lisp archive, anticipated the spirit and many of the features of today's big Linux archives. But few of us really thought very hard about what we were doing, or about what the very existence of that archive suggested about problems in the FSF's cathedral-building development model. I made one serious attempt around 1992 to get a lot of the Ohio code formally merged into the official Emacs Lisp library. I ran into political trouble and was largely unsuccessful.

But by a year later, as Linux became widely visible, it was clear that something different and much healthier was going on there. Linus's open development policy was the very opposite of cathedral-building. Linux's Internet archives were burgeoning, multiple distributions were being floated. And all of this was

早发布、常发布

早发布和频繁发布是 Linux 开发模式中关键的一部分。以前多数开发者（包括我）都认为这对象点样子的项目来说是个坏办法，因为早期版本几乎是问题版本的同义词，你不想消耗完用户的耐心。

这个观点也促使人们普遍采取建造大教堂式的开发。如果首要的目标是尽量让用户少遇到臭虫，那么你应该六个月甚至更久发布一个版本，在两次发布之间象 狗一样拼命工作调试。Emacs 的 C 核心就是这样开发的。Lisp 库实际上不是——因为在自由软件基金会所辖之外还有其它活跃的 Lisp 存档，提供独立于Emacs 的发布周期的新的和测试程序版本。

其中最重要的是俄亥俄州立大学的 Emacs Lisp 档案，已经超前具有了今天的 Linux 大型档案的精神和许多功能。但是我们中很少有人深度思考过我们在做什么、俄亥俄档案的存在本身说明了自由软 件基金会的大教堂开发模式的哪些问题。在 1992 前后，我认真地努力要把一大批俄亥俄代码合并到 Emacs Lisp 的官方库里去。我碰上了政治性的麻烦，非常的不成功。

但是到了一年以后，当 Linux 已经引起了广泛注意的时候，显然他们有什么不同的但是远远更为健康的东西。林纳斯的开放性开发政策正与建造大教堂的方式相反。Linux 的互联网档案枝蔓繁衍，多个发行种类在坊间流传。而所有这些都由核心系统的前所未闻的发放频率而驱动。

driven by an unheard-of frequency of core system releases.

Linus was treating his users as co-developers in the most effective possible way:

7. Release early. Release often. And listen to your customers.

Linus's innovation wasn't so much in doing quick-turnaround releases incorporating lots of user feedback (something like this had been Unix-world tradition for a long time), but in scaling it up to a level of intensity that matched the complexity of what he was developing. In those early times (around 1991) it wasn't unknown for him to release a new kernel more than once a day! Because he cultivated his base of co-developers and leveraged the Internet for collaboration harder than anyone else, this worked.

But how did it work? And was it something I could duplicate, or did it rely on some unique genius of Linus Torvalds?

I didn't think so. Granted, Linus is a damn fine hacker. How many of us could engineer an entire production-quality operating system kernel from scratch? But Linux didn't represent any awesome conceptual leap forward. Linus is not (or at least, not yet) an innovative genius of design in the way that, say, Richard Stallman or James Gosling (of NeWS and Java) are. Rather, Linus seems to me to be a genius of engineering and implementation, with a sixth sense for avoiding bugs and development dead-ends and a true knack for finding the minimum-effort path from point A to point B. Indeed, the whole design of Linux breathes this quality and mirrors Linus's essentially conservative and simplifying design approach.

So, if rapid releases and leveraging the Internet medium to the hilt were not accidents but integral parts of Linus's engineering-genius insight into the minimum-effort path, what was he maximizing? What was he cranking out of the machinery?

林纳斯在以最可能的有效的方式以合作者来对待他的用户：

7）早发布。常发布。听取用户的意见。

快速发布、采纳大量用户反馈，并不怎么算林纳斯的创新（Unix 世界很久以来就有这种传统）。他的创新之处是把这个办法升级到了与他开发的系统的复杂性相 匹配的规模和强度。在早期的时候（1991 左右），我们不是没听说过他一天发布不止一个新的内核版本！因为他比任何人都努力地培养合作开发群体、促进网上 合作，他的办法生效了。

但是它怎样生效的呢？这是我能够仿制的，还是只有林纳斯·托瓦兹的独特天才才能实现的?

我想不是的。林纳斯当然是个骨灰级黑客。我们有几个人能从头建造一整个工业级的操作系统核心呢？但是林纳斯并没有作出巨大的概念性突破。林纳斯不是（至少 还没有成为）象 Richard Stallman 或 James Gosling (of NeWS and Java)那种设计创新的天才。在我看来，林纳斯更象是工程和执行的天才，有着避开臭虫和死胡同的第六感官、找到从 A 点到 B 点最快通道的真本事。确实，整 个 linux 透露着这种特质，反映了林纳斯本质上的简约的设计方法。

如果快速发布和淋漓尽致的利用互联网媒介不是偶然的，而是林纳斯对最快通道的工程天才洞察力的有机部分，那么他的资本是什么呢？他在这个机制中依靠的是什么呢?

Put that way, the question answers itself. Linus was keeping his hacker/users constantly stimulated and rewarded—stimulated by the prospect of having an ego-satisfying piece of the action, rewarded by the sight of constant (even daily) improvement in their work.

Linus was directly aiming to maximize the number of person-hours thrown at debugging and development, even at the possible cost of instability in the code and user-base burnout if any serious bug proved intractable. Linus was behaving as though he believed something like this:

8. Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone.

Or, less formally, ``Given enough eyeballs, all bugs are shallow." I dub this: ``Linus's Law".

My original formulation was that every problem ``will be transparent to somebody". Linus demurred that the person who understands and fixes the problem is not necessarily or even usually the person who first characterizes it. ``Somebody finds the problem," he says, ``and somebody else understands it. And I'll go on record as saying that finding it is the bigger challenge." That correction is important; we'll see how in the next section, when we examine the practice of debugging in more detail. But the key point is that both parts of the process (finding and fixing) tend to happen rapidly.

In Linus's Law, I think, lies the core difference underlying the cathedral-builder and bazaar styles. In the cathedral-builder view of programming, bugs and development problems are tricky, insidious, deep phenomena. It takes months of scrutiny by a

这样一问，答案一目了然。林纳斯在不断地激励和奖掖他的黑客／用户们——激励来自于在参与中得到的自我实现，奖掖来自于看到他们自己的工作的持续（甚至每天）进步。

林纳斯直接瞄准了调试和开发中人力的最大化，即使代价是程序的稳定性，甚而某个修正不了的严重问题会疏离用户。林纳斯的做法似乎像是他相信：

8）如果 *beta* 测试者和合作开发者的群体足够大的话，几乎每个问题都会快速显形，会有人轻而易举地把它解决。

或者通俗一点，“只要眼球足够多，所有臭虫都好捉”。我称之为“林纳斯法则”。

我最早的表述是每个问题“都会有某个人搞明白”。林纳斯有异议：理解和解决问题的人不一定甚至一般不是第一个发现问题的人。“一个人发现问题”，他说，“另一个人把它搞明白。而且我会作证说发现问题更困难一些”。这是个重要的纠正；在下一节我们具体研究实际调试时会看到为什么。但是关键一点是，发现和解 决问题这两个步骤一般都会很快完成。

我认为林纳斯法则中包含有大教堂模式和市集模式的关键区别。在大教堂式的编程观念中，臭虫和开发上的问题是复杂、困难和深度的。要几个人全身全力几个月的钻研才有

dedicated few to develop confidence that you've winkled them all out. Thus the long release intervals, and the inevitable disappointment when long-awaited releases are not perfect.

In the bazaar view, on the other hand, you assume that bugs are generally shallow phenomena—or, at least, that they turn shallow pretty quickly when exposed to a thousand eager co-developers pounding on every single new release. Accordingly you release often in order to get more corrections, and as a beneficial side effect you have less to lose if an occasional botch gets out the door.

And that's it. That's enough. If ``Linus's Law'' is false, then any system as complex as the Linux kernel, being hacked over by as many hands as the that kernel was, should at some point have collapsed under the weight of unforseen bad interactions and undiscovered ``deep'' bugs. If it's true, on the other hand, it is sufficient to explain Linux's relative lack of bugginess and its continuous uptimes spanning months or even years.

Maybe it shouldn't have been such a surprise, at that. Sociologists years ago discovered that the averaged opinion of a mass of equally expert (or equally ignorant) observers is quite a bit more reliable a predictor than the opinion of a single randomly-chosen one of the observers. They called this the Delphi effect. It appears that what Linus has shown is that this applies even to debugging an operating system—that the Delphi effect can tame development complexity even at the complexity level of an OS kernel. [CV]

One special feature of the Linux situation that clearly helps along the Delphi effect is the fact that the contributors for any given project are self-selected. An early respondent pointed out that contributions are received not from a random sample, but

把它们清理干净的信心。所以需要长长的发布周期；一旦等候已久的版本不够完美，失望是不可避免的。

另一方面，在市集式的观念中，你预设臭虫都是简单的问题——至少在上千个共同开发者热心地琢磨每一个新版本的情况下，它们会很快就变简单了。相应地，你频繁发布来得到更多的纠错。作为一个附加效应，偶尔出个大勺子的后果也没有那么严重了。

这就是了。这也就够了。如果“林纳斯法则”是错误的，那么象 Linux 内核这样复杂的系统，经过了那么多人的敲打，应该在某一时刻已经在不曾预见的 恶性互动和深藏不露的问题的重压下崩溃了。如果另一方面它是正确的，它足以解释 Linux 相对较少的问题，和数月甚至数年以上的持续运行时间。

或许这不该是如此一个意外。社会学家们多年前就发现了一大群同样内行（或同样白痴）的观察者的平均预测要比其中随机选择的一个人的预测可靠得多。他们称之 为“神庙效应”。看来林纳斯显示了这一点甚至适用于调试一个操作系统——甚至在一个操作系统内核的复杂程度上，“神庙效应”可以简化开发。

Linux 情形中对“神庙效应”有帮助的特殊的一点是，任何一个项目的参与者都是自我选择的。一个早期评论指出，对 Linux 的贡献不是来自于一个随机的人群；他们都有足够

from people who are interested enough to use the software, learn about how it works, attempt to find solutions to problems they encounter, and actually produce an apparently reasonable fix. Anyone who passes all these filters is highly likely to have something useful to contribute.

Linus's Law can be rephrased as ``Debugging is parallelizable". Although debugging requires debuggers to communicate with some coordinating developer, it doesn't require significant coordination between debuggers. Thus it doesn't fall prey to the same quadratic complexity and management costs that make adding developers problematic.

In practice, the theoretical loss of efficiency due to duplication of work by debuggers almost never seems to be an issue in the Linux world. One effect of a ``release early and often" policy is to minimize such duplication by propagating fed-back fixes quickly [JH].

Brooks (the author of The Mythical Man-Month) even made an off-hand observation related to this: ``The total cost of maintaining a widely used program is typically 40 percent or more of the cost of developing it. Surprisingly this cost is strongly affected by the number of users. **More users find more bugs**." [emphasis added].

More users find more bugs because adding more users adds more different ways of stressing the program. This effect is amplified when the users are co-developers. Each one approaches the task of bug characterization with a slightly different perceptual set and analytical toolkit, a different angle on the problem. The ``Delphi effect" seems to work precisely because of this variation. In the specific context of debugging, the variation also tends to reduce duplication of effort.

的兴趣来使用这些软件、研究其机理、试图解决所遇到的问题，而且真正给出显然可行的解决办法。经过了这些筛选的人一般都会有可以贡献的 真才实料。

林纳斯法则也可以表述为“调试是可并行的”。尽管调试者们需要一个人来通讯协调，调试者们之间并不需要多少的协调。添加开发人员带来的平方级的复杂度和管理成本在这里不适用。

理论上因为调试者重复做功而导致的效率损失在 Linux 世界的实践中似乎从来都不是一个问题。“早发布、常发布”策略的一个后果就是通过快速公布反馈修补来把重复做功最小化。

布洛克（《人月神话》的作者）甚至作过一个相关的非正式评论：“一个广泛使用的程序的维护费用一般是它的开发成本的 40％以上。令人惊奇的是，这个费用受到用户数目的强烈影响。**用户越多，发现问题越多。**”［黑体是另加的］。

用户越多、发现问题越多是因为检验程序的角度也越多。当用户同时是合作开发者时，这个效应放大了。在检测问题的过程中，每个人都有一些不同的观察方 法和分析工具，从不同角度逼近同一问题。“神庙效应”似乎正是因为这种多样性而有效。在调试程序的特定环境下，这种多样性也利于减少重复做功。

So adding more beta-testers may not reduce the complexity of the current ``deepest'' bug from the developer's point of view, but it increases the probability that someone's toolkit will be matched to the problem in such a way that the bug is shallow to that person.

Linus coppers his bets, too. In case there are serious bugs, Linux kernel version are numbered in such a way that potential users can make a choice either to run the last version designated ``stable'' or to ride the cutting edge and risk bugs in order to get new features. This tactic is not yet systematically imitated by most Linux hackers, but perhaps it should be; the fact that either choice is available makes both more attractive. [HBS]

所以从开发者的角度来讲，增加更多的 beta 测试者不见得会减少当前最大问题的复杂程度，但会增加某个人的工具箱正好适用于该问题的几率——这样对这个人来说，这个问题就是小的。

林纳斯在此之外还留有一招。如果可能存在大的“臭虫”，Linux 内核的版本编号允许潜在用户选用老一点的“稳定”版本，或冒“臭虫”之险以求前沿 版本的最新功能。多数 Linux 黑客还没有系统地模仿这一招；但他们或许应该去模仿。给出这个选择使得两种版本都更有吸引力。

How Many Eyeballs Tame Complexity

It's one thing to observe in the large that the bazaar style greatly accelerates debugging and code evolution. It's another to understand exactly how and why it does so at the micro-level of day-to-day developer and tester behavior. In this section (written three years after the original paper, using insights by developers who read it and re-examined their own behavior) we'll take a hard look at the actual mechanisms. Non-technically inclined readers can safely skip to the next section.

要多少个眼球来驯服复杂度

在整体上观察到市集风格很大地加速了调试和代码进化是一回事，在微观上、日常工作的层次上、开发者和测试者的操作上来准确理解怎样和为什么是另一回 事。在这一节（写在原始文章的三年以后，采纳了读了原文、又对照了自身的开发者们的意见），我们来实打实地看一下真正的机制。不喜欢技术的读者可以安全跳转到下一节。

One key to understanding is to realize exactly why it is that the kind of bug report non–source-aware users normally turn in tends not to be very useful. Non–source-aware users tend to report only surface symptoms; they take their environment for granted, so they (a) omit critical background data, and (b) seldom include a reliable recipe for reproducing the bug.

The underlying problem here is a mismatch between the tester's and the developer's mental models of the program; the tester, on the outside looking in, and the developer on the inside looking out. In closed-source development they're both stuck in these roles, and tend to talk past each other and find each other deeply frustrating.

Open-source development breaks this bind, making it far easier for tester and developer to develop a shared representation grounded in the actual source code and to communicate effectively about it. Practically, there is a huge difference in leverage for the developer between the kind of bug report that just reports externally-visible symptoms and the kind that hooks directly to the developer's source-code–based mental representation of the program.

Most bugs, most of the time, are easily nailed given even an incomplete but suggestive characterization of their error conditions at source-code level. When someone among your beta-testers can point out, "there's a boundary problem in line nnn", or even just "under conditions X, Y, and Z, this variable rolls over", a quick look at the offending code often suffices to pin down the exact mode of failure and generate a fix.

Thus, source-code awareness by both parties greatly enhances both good communication and the synergy between what a beta-tester reports and what the core developer(s) know. In turn,

理解的一个关键是究竟为什么不关心源代码的用户所递交的臭虫报告一般倾向于帮助不大。不关心源代码的用户倾向于只报告表面症状；他们把运行环境当作理所当然的了，所以他们（一）漏掉了关键的背景数据，（二）极少包括一套能复制臭虫的步骤方法。

这里深层的问题是测试者和开发者脑中对程序的模型的不匹配；测试者从外往里看，而开发者从里往外看。在源代码封闭的开发模式中，他们都被卡在各自的这种角色里了，往往个说个的话，觉得对方相当恼火。

开源开发打破了这种束缚,使得在实在的源代码的基础上建立一个共享的模型、就之进行有效的交流对测试者和开发者容易的多。在实践中，那种仅仅描述外观症状的臭虫报告和直接联系到建立在开发者的代码上的抽象程序模型的报告，给予开发者的帮助是大不相同的。

如果有一个在代码层的对出错条件的描述，甚至不必完整，只要有所指向，大多数臭虫在大多数时间都是容易捉到的。当你的 beta 测试人员中某个人能指 出，“在第 nnn 行有一个边界问题”，或者只是“在 XYZ 条件下，这个变量溢出”，对问题代码的一个快速扫描常常足以锁定出错的准确模式、搞定一个修补办 法。

所以，如果 beta 测试者和核心开发者都对源代码心里有数，双方的交流和合作会得到大大增强。结果，这意味着

this means that the core developers' time tends to be well conserved, even with many collaborators.

Another characteristic of the open-source method that conserves developer time is the communication structure of typical open-source projects. Above I used the term "core developer"; this reflects a distinction between the project core (typically quite small; a single core developer is common, and one to three is typical) and the project halo of beta-testers and available contributors (which often numbers in the hundreds).

The fundamental problem that traditional software-development organization addresses is Brook's Law: ``Adding more programmers to a late project makes it later." More generally, Brooks's Law predicts that the complexity and communication costs of a project rise with the square of the number of developers, while work done only rises linearly.

Brooks's Law is founded on experience that bugs tend strongly to cluster at the interfaces between code written by different people, and that communications/coordination overhead on a project tends to rise with the number of interfaces between human beings. Thus, problems scale with the number of communications paths between developers, which scales as the square of the humber of developers (more precisely, according to the formula N*(N - 1)/2 where N is the number of developers).

The Brooks's Law analysis (and the resulting fear of large numbers in development groups) rests on a hidden assumption: that the communications structure of the project is necessarily a complete graph, that everybody talks to everybody else. But on open-source projects, the halo developers work on what are in effect separable parallel subtasks and interact with each other very little; code changes and bug reports stream through the core group,

核心开发人员的时间会节约下来，即使合作者人数很多。

开源方法另一个节约开发者时间的特点是典型的开源项目的通讯结构。我在上面用到了“核心开发者”一词；这反映了项目核心（一般很小；一个核心开发者很平常，一到三个很典型）和 beta 测试人员、协助人员组成的项目外沿（经常上百人）的区别。

传统上软件开发的组织结构的基本问题是布洛克法则：“在延期的项目添加程序员只会延期更久”。普遍来讲，布洛克法则认为，随着开发人员数目的增加，项目的复杂程度和通讯成本按平方增加，而业绩仅以直线增加。

经验表明，臭虫大多集中在不同人写的代码的界面上；而一个项目的通讯协调的成本一般按照人的界面的数量增加。这是布洛克法则的基础。也就是，问题随 开发者之间通讯路径的数目增加，而后者与开发者数目是平方关系（更准确地说，遵从公式 N*(N – 1)/2，这里 N 是开发者的数目）。

布洛克法则的分析（以及它引起的开发团体中对人数过多的恐惧）是基于一个潜在的前提：项目的通讯结构必须是一个完整的图、每个人都与其他所有人交接。但是在开源项目中，外沿的开发者做的实际上是平行分离的子项目，彼此交接甚少；代码变动和臭虫报告都流经项目的核心，只有

and only within that small core group do we pay the full Brooksian overhead. [SU]

There are are still more reasons that source-code–level bug reporting tends to be very efficient. They center around the fact that a single error can often have multiple possible symptoms, manifesting differently depending on details of the user's usage pattern and environment. Such errors tend to be exactly the sort of complex and subtle bugs (such as dynamic-memory-management errors or nondeterministic interrupt-window artifacts) that are hardest to reproduce at will or to pin down by static analysis, and which do the most to create long-term problems in software.

A tester who sends in a tentative source-code–level characterization of such a multi-symptom bug (e.g. "It looks to me like there's a window in the signal handling near line 1250" or "Where are you zeroing that buffer?") may give a developer, otherwise too close to the code to see it, the critical clue to a half-dozen disparate symptoms. In cases like this, it may be hard or even impossible to know which externally-visible misbehaviour was caused by precisely which bug—but with frequent releases, it's unnecessary to know. Other collaborators will be likely to find out quickly whether their bug has been fixed or not. In many cases, source-level bug reports will cause misbehaviours to drop out without ever having been attributed to any specific fix.

Complex multi-symptom errors also tend to have multiple trace paths from surface symptoms back to the actual bug. Which of the trace paths a given developer or tester can chase may depend on subtleties of that person's environment, and may well change in a not obviously deterministic way over time. In effect, each developer and tester samples a semi-random set of the program's state space when looking for the etiology of a symptom. The more subtle and complex the bug, the less likely that skill will

在小小的核心团体中全面的布洛克成本 才有效。

还有其它的原因使得源代码层次上的臭虫报告往往更有效。一个核心问题是单独的错误常常可以产生多个不同的症状，在用户使用习惯和环境的细节不同时有 不同显示。这类错误一般正是那些复杂和微妙的臭虫——那些最难故意复制或用静态分析捕捉的、那些在软件中制造长期问题的祸根（比如动态内存管理错误或窗口随机干预后果等）。

一个送进这样一个多症状臭虫的非正式源码层描述（例如：“我觉得在第 1250 行附近象是有一个信号处理的窗口”或者“你在哪里把那个缓冲清零的？”）的测试者可以给一个“当局者迷”的开发者通往半打不相关症状的关键线索。在这样的情况下，哪一个外观的错误来自于哪一个具体的臭虫会是很难（如果可能的话）发现的——但是在频繁发布下，这是不必去发现的。其他合作者们一般会很快发现他们的臭虫是否已被修复。在许多情况下，源码层的臭虫报告会导致外观错误在归因于任何修复之前消失。

复杂的多症状错误也常常会有多个从表面症状联系到内在臭虫的跟踪途径。一个特定的开发者或测试者所能追寻的跟踪途径可能取决于这个人的具体环境细 节，也很可能随着时间的改变发生不便预测的变化。实际上，每一个开发者和测试者在寻找一个症状的病原的时候都是在检查该程序的状态空间的一个“半随机”的 集合。臭虫越微妙越复杂，单

be able to guarantee the relevance of that sample.

For simple and easily reproducible bugs, then, the accent will be on the "semi" rather than the "random"; debugging skill and intimacy with the code and its architecture will matter a lot. But for complex bugs, the accent will be on the "random". Under these circumstances many people running traces will be much more effective than a few people running traces sequentially—even if the few have a much higher average skill level.

This effect will be greatly amplified if the difficulty of following trace paths from different surface symptoms back to a bug varies significantly in a way that can't be predicted by looking at the symptoms. A single developer sampling those paths sequentially will be as likely to pick a difficult trace path on the first try as an easy one. On the other hand, suppose many people are trying trace paths in parallel while doing rapid releases. Then it is likely one of them will find the easiest path immediately, and nail the bug in a much shorter time. The project maintainer will see that, ship a new release, and the other people running traces on the same bug will be able to stop before having spent too much time on their more difficult traces [RJ].

靠技能就越难保证找到那个相关的集合。

对于简单的容易复制的臭虫，那么，重音要放在“半”上面而不是“随机”上面；调试的技能和对代码、框架的熟悉是最重要的。但对于复杂的臭虫，重音就要放在“随机”上面。在这种情况下许多人同时追踪要比少数人持续追踪有效的多——即使这少数人的技能水平高的多。

要是从不同的表面症状挖掘到臭虫的跟踪途径难度不一、难以从观察症状来预测的话，这一效果就会非常之明显了。一个持续追踪这些路径的开发者一开始可能会遇 到一个简单的路径也同样可能遇到一个复杂的路径。另一方面，试想有许多人在快速发布下平行地来检查这些追踪路径。那么其中的某一个人可能会马上发现最容易 的路径，在短的多的时间里搞定这个臭虫。维护项目的人会看到这个，发行一个新版本；其他在更困难的路径上追踪同一个臭虫的人们就可以在花费太多的时间之前 停下来。

When Is a Rose Not a Rose?

Having studied Linus's behavior and formed a theory about why it was successful, I made a conscious decision to test this theory on my new (admittedly much less complex and ambitious) project.

But the first thing I did was reorganize and simplify popclient a lot. Carl Harris's implementation was very sound, but exhibited a kind of unnecessary complexity common to many C programmers. He treated the code as central and the data structures as support for the code. As a result, the code was beautiful but the data structure design ad-hoc and rather ugly (at least by the high standards of this veteran LISP hacker).

I had another purpose for rewriting besides improving the code and the data structure design, however. That was to evolve it into something I understood completely. It's no fun to be responsible for fixing bugs in a program you don't understand.

For the first month or so, then, I was simply following out the implications of Carl's basic design. The first serious change I made was to add IMAP support. I did this by reorganizing the protocol machines into a generic driver and three method tables (for POP2, POP3, and IMAP). This and the previous changes illustrate a general principle that's good for programmers to keep in mind, especially in languages like C that don't naturally do dynamic typing:

9. Smart data structures and dumb code works a lot better than the other way around.

Brooks, Chapter 9: ``Show me your flowchart and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowchart; it'll be obvious."

画虎莫类犬

研究了林纳斯的作法后我形成了一个它何以成功的理论；我有意识地决定在我的新项目（当然没有 Linux 那么复杂和宏伟）里测试这个理论。

但我做的第一件事是把 popclient 重组和简化了许多。卡尔。哈里斯的代码实现的很好，但是有一种在 C 程序员中常见的多余的复杂。他把代码放在了中心位置，数据结构作为辅助。结果代码很漂亮，但是数据结构设计得潦草甚至丑陋（至少根据我这个 LISP 老手的标准来看）。

然而，除了改进代码和数据结构设计以外，我的重写还有另一层目的。那是把它进化成一个我完全理解的东西。要是你不完全理解一个程序，维护起来可就不好玩了。

于是在最初的一个月左右，我只是在按照卡尔的章程做事。我作的第一个重要改变是添加了 IMAP 支持。我实现这点的方法是：把协议部分的机制重组成了 一个通用的驱动和三个方法表单（分别针对 POP2、POP3 和 IMAP）。这和以前的变动都示范了一个程序员们应该记住的通用原则，尤其对于像 C 这种本身 不支持动态数据类型的语言：

9）聪明的数据结构和愚蠢的代码要不反过来好的多。

布洛克的第九章：“给我看你的流程图而隐藏你的表单，我会继续糊涂着。给我看你的表单，我一般就不需要你的流程图了；事情该是明显的了”。经过三十年的文化和术

Allowing for thirty years of terminological/cultural shift, it's the same point.

At this point (early September 1996, about six weeks from zero) I started thinking that a name change might be in order—after all, it wasn't just a POP client any more. But I hesitated, because there was as yet nothing genuinely new in the design. My version of popclient had yet to develop an identity of its own.

That changed, radically, when popclient learned how to forward fetched mail to the SMTP port. I'll get to that in a moment. But first: I said earlier that I'd decided to use this project to test my theory about what Linus Torvalds had done right. How (you may well ask) did I do that? In these ways:

- I released early and often (almost never less often than every ten days; during periods of intense development, once a day).
- I grew my beta list by adding to it everyone who contacted me about fetchmail.
- I sent chatty announcements to the beta list whenever I released, encouraging people to participate.
- And I listened to my beta-testers, polling them about design decisions and stroking them whenever they sent in patches and feedback.

The payoff from these simple measures was immediate. From the beginning of the project, I got bug reports of a quality most developers would kill for, often with good fixes attached. I got thoughtful criticism, I got fan mail, I got intelligent feature suggestions. Which leads to:

10. If you treat your beta-testers as if they're your most valuable resource, they will respond by becoming your most

语的变迁，这是同一个道理。

这时（1996 年 9 月初，大约开工后六个星期），我开始想这个程序大概该换个名字了－它毕竟不再仅仅是一个 POP 客户端软件。但是我在犹豫，因为在设计上还没有什么真正的新东西。我的 popclient 版本还需要发展出它自己的特征。

当 popclient 学会了怎样把取到的邮件转发到 SMTP 端口的时候，这一点迅速改变了。我过一会儿再细谈这个。但是首先：我说过我决定用这个项目来测试我对林纳斯。托瓦兹的成功之处的理论。（您也会问）我是怎样做的呢？在以下方面：

- 我早发布和常发布（几乎从未低于十天一次；高强度开发的时候，一天一次）。
- 我把每个和我联系 fetchmail 的人加进了我的 beta 测试名单。
- 每当我发布一个版本，我给 beta 名单发送一个家常式的通告，鼓励大家参与。
- 我听取 beta 测试者的意见，在设计上征求他们的看法，当他们送交补丁和反馈的时候给予鼓励。

这些简单的办法立时就见效了。从项目的开始，我就收到多数开发者梦寐以求的那种高质量的臭虫报告，经常还附带了好的补丁。我收到了深思熟虑的评论、粉丝的邮件、高明的功能性建议。这指向了：

10） 如果你以“最有价值资源”来对待你的 *beta* 测试者，他们会以成为“最有价值资源”来回应。

valuable resource.

One interesting measure of fetchmail's success is the sheer size of the project beta list, fetchmail-friends. At the time of latest revision of this paper (November 2000) it has 287 members and is adding two or three a week.

Actually, when I revised in late May 1997 I found the list was beginning to lose members from its high of close to 300 for an interesting reason. Several people have asked me to unsubscribe them because fetchmail is working so well for them that they no longer need to see the list traffic! Perhaps this is part of the normal life-cycle of a mature bazaar-style project.

fetchmail 的成功的一个有意思的方面是项目的 beta 测试名单（fetchmail-friends）的庞大。在这篇文章的最后一稿的时候（2000 年 11 月），它有 287 名成员，而且每个星期在增加两三名。

实际上，当我在 1997 年 5 月下旬改写的时候，我发现这个名单由于一个有意思的原因，从它近 300 的巅峰开始流失成员了。一些人要求我把他们从名单中去 掉，因为 fetchmail 对他们来讲运行完美、他们再也不需要阅读这个邮件列表了！或许这是一个成熟的市集风格的项目的正常生命周期的一部分。

Popclient becomes Fetchmail

The real turning point in the project was when Harry Hochheiser sent me his scratch code for forwarding mail to the client machine's SMTP port. I realized almost immediately that a reliable implementation of this feature would make all the other mail delivery modes next to obsolete.

For many weeks I had been tweaking fetchmail rather incrementally while feeling like the interface design was serviceable but grubby—inelegant and with too many exiguous options hanging out all over. The options to dump fetched mail to a mailbox file or standard output particularly bothered me, but I couldn't figure out why.

Popclient 变成了 *Fetchmail*

这个项目的真正转折点是哈利·浩赫海斯（Harry Hochheiser）把他的转发邮件到客户机 SMTP 端口的草稿编码发给了我。我几乎马上意识到这个功能要是可靠地实现出来，会让其它所有的邮件递送模式几近成为往事。

许多个星期以来，我更像是在一点点地摆弄 fetchmail；一边感觉到界面设计还可以用，但是不够干净漂亮——太多不足道的选项，比比皆是。把收取的邮件扔在一个邮件箱里或转到标准输出里的选项尤其让我心烦，但我想不出为什么。

(If you don't care about the technicalia of Internet mail, the next two paragraphs can be safely skipped.)

What I saw when I thought about SMTP forwarding was that popclient had been trying to do too many things. It had been designed to be both a mail transport agent (MTA) and a local delivery agent (MDA). With SMTP forwarding, it could get out of the MDA business and be a pure MTA, handing off mail to other programs for local delivery just as sendmail does.

Why mess with all the complexity of configuring a mail delivery agent or setting up lock-and-append on a mailbox when port 25 is almost guaranteed to be there on any platform with TCP/IP support in the first place? Especially when this means retrieved mail is guaranteed to look like normal sender-initiated SMTP mail, which is really what we want anyway.

(Back to a higher level....)

Even if you didn't follow the preceding technical jargon, there are several important lessons here. First, this SMTP-forwarding concept was the biggest single payoff I got from consciously trying to emulate Linus's methods. A user gave me this terrific idea—all I had to do was understand the implications.

11. The next best thing to having good ideas is recognizing good ideas from your users. Sometimes the latter is better.

Interestingly enough, you will quickly find that if you are completely and self-deprecatingly truthful about how much you owe other people, the world at large will treat you as though you did every bit of the invention yourself and are just being becomingly modest about your innate genius. We can all see how well this worked for Linus!

（如果你对互联网邮件的技术细节不感兴趣，可以安全跳过下面的两个段落。）

当我考虑 SMTP 转发的时候，我看到的是 popclient 在试图做太多的事情。它被设计成了既是一个邮件传输工具（MTA），又是一个本地投递工 具（MDA）。有了 SMTP 转发功能，它就可以摆脱 MDA 的负荷，专心只作 MTA，把邮件的本地投递留给 sendmail 之类的程序来做。

当几乎每一个有 TCP/IP 支持的平台上都预留了 25 号端口的时候，为什么还要去折腾 MDA 的复杂配置或邮箱的“锁定－添加”呢？尤其是这意味着收到的邮件看起来几乎保证和正常的人工发送的 SMTP 邮件一样——正中我们下怀。（返回到高一层次上……）

即使你没有去读上面的技术术语，这里有几条重要的经验。首先，这个 SMTP 转发的点子是我有意模仿林纳斯的方法的最大的收获。一个用户给了我这个巨棒的点子——我需要做的仅仅是理解它的含义。

11）仅次于拥有好的主意的是认识到来自于用户的好主意。有时候后者更好一些。

有意思的是，如果你完全坦诚和谦虚地承认你欠了别人多少，你很快就会发现外面的世界会把你放在这样一个地位上——好像你自己做了发明的每一部分、只不过对你生来的天才一味谦虚而已。我们都可以看到林纳斯如此受益了多少！

(When I gave my talk at the first Perl Conference in August 1997, hacker extraordinaire Larry Wall was in the front row. As I got to the last line above he called out, religious-revival style, ``Tell it, tell it, brother!''. The whole audience laughed, because they knew this had worked for the inventor of Perl, too.)

After a very few weeks of running the project in the same spirit, I began to get similar praise not just from my users but from other people to whom the word leaked out. I stashed away some of that email; I'll look at it again sometime if I ever start wondering whether my life has been worthwhile :-).

But there are two more fundamental, non-political lessons here that are general to all kinds of design.

12. Often, the most striking and innovative solutions come from realizing that your concept of the problem was wrong.

I had been trying to solve the wrong problem by continuing to develop popclient as a combined MTA/MDA with all kinds of funky local delivery modes. Fetchmail's design needed to be rethought from the ground up as a pure MTA, a part of the normal SMTP-speaking Internet mail path.

When you hit a wall in development—when you find yourself hard put to think past the next patch—it's often time to ask not whether you've got the right answer, but whether you're asking the right question. Perhaps the problem needs to be reframed.

Well, I had reframed my problem. Clearly, the right thing to do was (1) hack SMTP forwarding support into the generic driver, (2) make it the default mode, and (3) eventually throw out all the other delivery modes, especially the deliver-to-file and deliver-to-standard-output options.

（当我在 1997 年的第一次 Perl 大会上发言的时候，黑客大亨 Larry Wall 正坐在前排上。当我讲到上面的那句话的时候，他喊了起来，宗教复活式的口吻，“说出来，说出来，哥们！”全场都笑了，因为他们知道这一点对 Perl 的发明者也不例外。）

在我发扬这种精神把项目运行了几个星期以后，我开始得到类似的赞扬——不仅来自我的用户们，而且来自于其他有所耳闻的人们。我把一些邮件收藏了起来；要是什么时候我开始疑惑我生命的意义的时候，我就拿出来再看看:-)。

但是有两条基本的、非政治性的经验对各种设计都适用。

12）最有突破和创新的方案常常来自于意识到你把问题的模型弄错了。

当我继续把 popclient 开发成一个带有七七八八的本地递送模式的 MTA/MDA 时，我就是试图在解决错误的问题。Fetchmail 的设计应该作为一个纯粹的 MTA——正常的 SMTP 互联网邮件传输路径的一部分——来重起炉灶。

当你在开发中碰到死胡同时——当你绞尽脑汁要超越下一个补丁的时候——一般来讲你该问的不是你的答案对不对，而是你的问题对不对。或许你的问题需要重新定义。嗯……我重新定义了我的问题。显然，正确的路子是（1）把 SMTP 转发支持加入到通用驱动里去，（2）把它设置为默认模式，（3）最终把其它的传递模式都去掉，尤其是传递到文件和标准输出的模式。

I hesitated over step 3 for some time, fearing to upset long-time popclient users dependent on the alternate delivery mechanisms. In theory, they could immediately switch to .forward files or their non-sendmail equivalents to get the same effects. In practice the transition might have been messy.

But when I did it, the benefits proved huge. The cruftiest parts of the driver code vanished. Configuration got radically simpler—no more grovelling around for the system MDA and user's mailbox, no more worries about whether the underlying OS supports file locking.

Also, the only way to lose mail vanished. If you specified delivery to a file and the disk got full, your mail got lost. This can't happen with SMTP forwarding because your SMTP listener won't return OK unless the message can be delivered or at least spooled for later delivery.

Also, performance improved (though not so you'd notice it in a single run). Another not insignificant benefit of this change was that the manual page got a lot simpler.

Later, I had to bring delivery via a user-specified local MDA back in order to allow handling of some obscure situations involving dynamic SLIP. But I found a much simpler way to do it.

The moral? Don't hesitate to throw away superannuated features when you can do it without loss of effectiveness. Antoine de Saint-Exupéry (who was an aviator and aircraft designer when he wasn't authoring classic children's books) said:

13. ``Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away."

这第三步让我犹豫了一段时间，担心会影响那些依赖于另类模式的 popclient 的老用户。理论上，他们可以马上用 .forward 文件或 sendmail 的替代程序来获得相同的效果。在实践中，这种转换可能会让人头大。

但是一旦我这样做了，好处非常明显。驱动代码中毛病最多的地方消失了。配置容易了太多——再也不需要围着系统 MDA 和用户信箱打转，再也不需要担心背后的操作系统是否支持文件锁定。

而且，唯一可能丢失邮件的途径不见了。如果你指定递送到文件而磁盘满了的话，你的邮件就丢了。这在 SMTP 转发中不会发生，因为除非邮件成功传送或至少缓存了，SMTP 的聆听端不会给以确认。

而且，性能提高了（尽管不是你单独运行一次就能感觉到的）。改变后另外一个不是非同小可的好处是说明手册简化了许多。

后来，我为了对付一些涉及到动态 SLIP（Serial Line Internet Protocol，串行线互联网协议）的晦涩情形，曾经不得以把用户指定的本地 MDA 递送功能加回来。但是我找到了一个简单的多的办法来做它。

说明了什么道理？在不损失效率的情况下，不要犹豫把过了气的功能扔掉。埃克絮佩利✲（他不在写作经典儿童图书的时候是个飞行员和飞机设计师）曾说过：

13）“设计达到完美的时候，不是增加得不能再增加了、而是减少得不能再减少了”。

✲译注：Antoine de Saint-Exupéry，《小王子》的作者

When your code is getting both better and simpler, that is when you know it's right. And in the process, the fetchmail design acquired an identity of its own, different from the ancestral popclient.

It was time for the name change. The new design looked much more like a dual of sendmail than the old popclient had; both are MTAs, but where sendmail pushes then delivers, the new popclient pulls then delivers. So, two months off the blocks, I renamed it fetchmail.

There is a more general lesson in this story about how SMTP delivery came to fetchmail. It is not only debugging that is parallelizable; development and (to a perhaps surprising extent) exploration of design space is, too. When your development mode is rapidly iterative, development and enhancement may become special cases of debugging—fixing `bugs of omission' in the original capabilities or concept of the software.

Even at a higher level of design, it can be very valuable to have lots of co-developers random-walking through the design space near your product. Consider the way a puddle of water finds a drain, or better yet how ants find food: exploration essentially by diffusion, followed by exploitation mediated by a scalable communication mechanism. This works very well; as with Harry Hochheiser and me, one of your outriders may well find a huge win nearby that you were just a little too close-focused to see.

当你的代码变得既优良又简单的时候，这时你知道它上了正轨了。在这个过程中，fetchmail 的设计获得了它自己的特色，脱离了上一代的 popclient。

到了该换名字的时候了。新的设计和老的 popclient 相比，更像是一个 sendmail 的对手；二者都是 MTA，但 sendmail 是发出去再投递，新的 popclient 是接过来再投递。所以在动工两个月后，我把它重命名为 fetchmail。

在这个 SMTP 转发功能如何进入 fetchmail 的故事里，有一个更普遍的经验。那是不仅调试是可并行的；开发和搜索设计空间（在可能令人吃惊的程度上）也是。当你的开发模式在快速循环中，开发和改进有可能成为调试的特例——修正软件模型的原始设计中的“不足的问题”。

即使在高一层次的设计上，有许多共同开发者在你的产品的设计空间附近随机行走可以是很有价值的。想象一下一摊水怎样发现下水口的，或者更恰当一点，蚂蚁怎 样发现食物的：本质上以分散来搜索，然后以一个可扩展的通讯机制来利用。这一点很管用；就像浩赫海斯和我一样，你们随行中的一个很可能发现一个宝藏——你 只不过太专注了一点而看不到。

Fetchmail Grows Up

There I was with a neat and innovative design, code that I knew worked well because I used it every day, and a burgeoning beta list. It gradually dawned on me that I was no longer engaged in a trivial personal hack that might happen to be useful to few other people. I had my hands on a program that every hacker with a Unix box and a SLIP/PPP mail connection really needs.

With the SMTP forwarding feature, it pulled far enough in front of the competition to potentially become a ``category killer", one of those classic programs that fills its niche so competently that the alternatives are not just discarded but almost forgotten.

I think you can't really aim or plan for a result like this. You have to get pulled into it by design ideas so powerful that afterward the results just seem inevitable, natural, even foreordained. The only way to try for ideas like that is by having lots of ideas—or by having the engineering judgment to take other peoples' good ideas beyond where the originators thought they could go.

Andy Tanenbaum had the original idea to build a simple native Unix for IBM PCs, for use as a teaching tool (he called it Minix). Linus Torvalds pushed the Minix concept further than Andrew probably thought it could go—and it grew into something wonderful. In the same way (though on a smaller scale), I took some ideas by Carl Harris and Harry Hochheiser and pushed them hard. Neither of us was `original' in the romantic way people think is genius. But then, most science and engineering and software development isn't done by original genius, hacker mythology to the contrary.

Fetchmail 长大了

现在我有了一个整洁新颖的设计；我知道代码工作良好因为我天天都在使用；beta 测试名单繁荣热闹。我慢慢明白了我不再是在作一个可能对几个人有用的琐碎的个人编程。我在主持一个所有拥有 Unix 机器和 SLIP/PPP 邮件接口的用户都需要的程序。

带有 SMTP 转发的功能的 fetchmail 在竞争对手面前表现强劲，潜质上可能成为一个“类型杀手”——那种在它的功能类型里如此称职以至于对手们不仅被放弃了而且几乎被遗忘了。

我觉得这种结果是有点可遇而不可求的。你必须要有强大的设计构想，能把你整个吸引进去，而产出的结果就像是不可避免的、天然的、甚至命中注定的。追求这种构想的唯一方法就是要有很多想法——或者有工程眼光能够把其他人的好主意推进到他们都想不到的地步。

安迪·塔内保（Andy Tanenbaum）有了一个原始主意，作一个针对 IBM 兼容机的简单 Unix，作为教学工具来使用（他称之为 Minix）。林纳斯·托瓦兹把 Minix 的概 念推进到了安迪可能想都想不到的地步——演变成了一个奇妙的东西。与此类似（然而在小一些的规模上），我从卡尔·哈里斯和哈利·浩赫海斯那里借来主意并努力 推进它们。我们都没有人们对天才的浪漫想象中的那种“原创性”。但是说回来，多数科学、技术和软件开发都不是由原创的天才完成的，而是相反，来自于黑客一 派。

The results were pretty heady stuff all the same—in fact, just the kind of success every hacker lives for! And they meant I would have to set my standards even higher. To make fetchmail as good as I now saw it could be, I'd have to write not just for my own needs, but also include and support features necessary to others but outside my orbit. And do that while keeping the program simple and robust.

The first and overwhelmingly most important feature I wrote after realizing this was multidrop support—the ability to fetch mail from mailboxes that had accumulated all mail for a group of users, and then route each piece of mail to its individual recipients.

I decided to add the multidrop support partly because some users were clamoring for it, but mostly because I thought it would shake bugs out of the single-drop code by forcing me to deal with addressing in full generality. And so it proved. Getting RFC 822 address parsing right took me a remarkably long time, not because any individual piece of it is hard but because it involved a pile of interdependent and fussy details.

But multidrop addressing turned out to be an excellent design decision as well. Here's how I knew:

14. Any tool should be useful in the expected way, but a truly great tool lends itself to uses you never expected.

The unexpected use for multidrop fetchmail is to run mailing lists with the list kept, and alias expansion done, on the client side of the Internet connection. This means someone running a personal machine through an ISP account can manage a mailing list without continuing access to the ISP's alias files.

Another important change demanded by my beta-testers was support for 8-bit MIME (Multipurpose Internet Mail Extensions) operation. This was pretty easy to do, because I had been careful

结果都一致是那种很眩目的东西——事实上，正是每一个黑客毕生追求的那种成功！而且这意味着我将不得不把我的标准设得更高。为了让 fetchmail 达到 我这时预期的水平，我必须不仅为自己的需要编程，而且要包括和支持他人必需的然而在我的轨道之外的功能。这样做的同时要保持程序简单结实。

意识到这点之后，我所写的第一个也是绝对最重要的一个功能是集体收发——从一群用户的集体信箱里把累积的所有邮件取来，然后把每一封分发给单独的收信人。

我决定添加集体收发功能部分上是因为用户们吵着要，然而主要是因为我觉得它会迫使我在完全普遍的条件下处理地址解析问题，从而甩掉单信发送模式中的臭虫。 结果如我所愿。我花了奇长的时间才把 RFC 822 的地址解析搞定，不是因为它的哪一部分很难，而是因为它涉及了一堆相互关联的烦人的细节。

然而集体收发也成为了一项优秀的设计决定。我是这样知道的：

14）任何一个工具都应该达到预期的用处，但是一个真正棒的工具会带来你从来预期不到的用处。

fetchmail 中集体收发的不曾预期的功能是邮件列表可以在网络连接的客户端保存列表和别名扩展。这样，使用个人机通过 ISP 上网的一个人不必持续访问 ISP 方的别名扩展文件就可以管理一个邮件列表。

我的 beta 测试员们要求的另一个重要变化是支持 8 位的 MIME 操作。这个很容易，因为我已经很小心的保持了 8

to keep the code 8-bit clean (that is, to not press the 8th bit, unused in the ASCII character set, into service to carry information within the program). Not because I anticipated the demand for this feature, but rather in obedience to another rule:

15. When writing gateway software of any kind, take pains to disturb the data stream as little as possible—and never throw away information unless the recipient forces you to!

Had I not obeyed this rule, 8-bit MIME support would have been difficult and buggy. As it was, all I had to do is read the MIME standard (RFC 1652) and add a trivial bit of header-generation logic.

Some European users bugged me into adding an option to limit the number of messages retrieved per session (so they can control costs from their expensive phone networks). I resisted this for a long time, and I'm still not entirely happy about it. But if you're writing for the world, you have to listen to your customers—this doesn't change just because they're not paying you in money.

A Few More Lessons from Fetchmail

Before we go back to general software-engineering issues, there are a couple more specific lessons from the fetchmail experience to ponder. Nontechnical readers can safely skip this section.

The rc (control) file syntax includes optional `noise' keywords that are entirely ignored by the parser. The English-like syntax they allow is considerably more readable than the traditional terse keyword-value pairs you get when you strip them all out.

位代码的洁净（就是，没有强迫 ASCII 字符集中没有使用的第 8 位比特去携带程序中的信息）。不是因为我预料到了这个功能要求，而是遵循了另一个规则：

15）在写任何关口软件的时候，花点功夫尽可能不要干扰数据流——除非用户强迫你，永远不要扔掉任何信息！

要是我没有遵守这个规则，8 位 MIME 支持会很困难且毛病不断。事实上，我所需要做的仅仅是读一下 MIME 标准（RFC 1652），添加一条小小的文件头生成规则。

在一些欧洲的用户的要求下，我添加了一个选项来限制每次连接能下载的邮件数目（这样他们可以控制他们昂贵的电话费）。我对这件事抵制了很长一段时间，直到现在也不是完全满意。但是如果你给外边的世界写程序，你不得不聆听你的顾客——就算他们不付你钱也是这个道理。

Fetchmail 带来的其它几条经验

在我们回到普遍的软件工程问题之前，fetchmail 的经历中还有几条经验值得细想。非技术性的读者可以安全地跳开这一节。

rc 控制文件的语法中包括了一些完全不解析的、可选的“噪音”关键词。它们所允许的类似英语的语法比起你把它们全部梳理掉之后所剩下的传统的简洁的“关键词－对应值”要可读得多。

These started out as a late-night experiment when I noticed how much the rc file declarations were beginning to resemble an imperative minilanguage. (This is also why I changed the original popclient ``server'' keyword to ``poll'').

这些开始于一个深夜实验——当我注意到 rc 文件的定义开始看起来多么像一个微型的指令语言。（这也是我为什么把 popclient 原有的“server”关键词换成了“poll”。）

It seemed to me that trying to make that imperative minilanguage more like English might make it easier to use. Now, although I'm a convinced partisan of the ``make it a language'' school of design as exemplified by Emacs and HTML and many database engines, I am not normally a big fan of ``English-like'' syntaxes.

在我来看，努力把这个微型指令语言做得更像英语可能会使它更容易使用。尽管我现在是一个“把它做成一门语言”设计流派——就像 Emacs 和 HTML 和许多数据库引擎展示的那样——的信徒，我一般不是特别热衷于“类似英语的”语法。

Traditionally programmers have tended to favor control syntaxes that are very precise and compact and have no redundancy at all. This is a cultural legacy from when computing resources were expensive, so parsing stages had to be as cheap and simple as possible. English, with about 50% redundancy, looked like a very inappropriate model then.

传统上，程序员们倾向于选用简洁紧凑、完全没有冗余的控制语法。这是计算资源昂贵的时期的文化遗留，那时解析过程不得不尽可能的廉价和简单。那时，大概有 50%冗余的英语看起来像是一个非常不合适的模型。

This is not my reason for normally avoiding English-like syntaxes; I mention it here only to demolish it. With cheap cycles and core, terseness should not be an end in itself. Nowadays it's more important for a language to be convenient for humans than to be cheap for the computer.

这不是我一般避免英语式语法的原因；我在这儿提起它正是为了打破这个看法。有了便宜的循环和核心，简洁不应该为了简洁而简洁。现在一门语言对于人的方便比对于计算机的廉价更重要。

There remain, however, good reasons to be wary. One is the complexity cost of the parsing stage—you don't want to raise that to the point where it's a significant source of bugs and user confusion in itself. Another is that trying to make a language syntax English-like often demands that the ``English'' it speaks be bent seriously out of shape, so much so that the superficial resemblance to natural language is as confusing as a traditional syntax would have been. (You see this bad effect in a lot of so-called ``fourth generation'' and commercial database-query

然而我们还有需要小心的原因。其中之一是解析过程的复杂性成本——你不想把它提高到富产臭虫和困惑用户的程度。另外，试图把一门语言的语法做得像英 语经常迫使它所使用的“英语”严重扭曲变形，以至于对自然语言的表面模仿变得像传统语法一样令人困惑。（你可以在许多所谓的“第四代”和商业数据库查询语 言中看到这个坏效果。）

languages.)

The fetchmail control syntax seems to avoid these problems because the language domain is extremely restricted. It's nowhere near a general-purpose language; the things it says simply are not very complicated, so there's little potential for confusion in moving mentally between a tiny subset of English and the actual control language. I think there may be a broader lesson here:

16. When your language is nowhere near Turing-complete, syntactic sugar can be your friend.

Another lesson is about security by obscurity. Some fetchmail users asked me to change the software to store passwords encrypted in the rc file, so snoopers wouldn't be able to casually see them.

I didn't do it, because this doesn't actually add protection. Anyone who's acquired permissions to read your rc file will be able to run fetchmail as you anyway—and if it's your password they're after, they'd be able to rip the necessary decoder out of the fetchmail code itself to get it.

All .fetchmailrc password encryption would have done is give a false sense of security to people who don't think very hard. The general rule here is:

17. A security system is only as secure as its secret. Beware of pseudo-secrets.

fetchmail 的控制语法似乎避免了这些问题，因为它的语言空间极端有限。它和一个普通用途的语言更本接不上边儿；它所说的事情压根儿不复杂，所以在英语的一个微小子集里和实际上的控制语言之间进行脑力转换而发生迷惑的可能性很小。我觉得这里可能有一个更普适的经验：

16）当你的语言离图灵穷尽还差得远的时候，，给语法加点风味可以有帮助。

另一条经验是关于隐藏带来的安全性。一些 fetchmail 的用户要求我改一下软件来把加密的密码储存在 rc 控制文件里，这样入侵者就不会在无意中看到它们。
我没有照办，因为这实际上并不会添加保护。不管怎样，任何一个取得了权限来读你的 rc 文件的人都可以像你一样来运行 fetchmail——如果他们真的来找你的密码，他们可以从 fetchmail 代码本身中剥离出必要的解码器来得手。
所有.fetchmail 密码加密能做的是给那些不怎么用心思考的人一种虚假的安全感。这里的一般规则是：

17）一个安全系统的安全性取决于它保守的秘密的安全性。小心伪秘密。

Necessary Preconditions for the Bazaar Style

Early reviewers and test audiences for this essay consistently raised questions about the preconditions for successful bazaar-style development, including both the qualifications of the project leader and the state of code at the time one goes public and starts to try to build a co-developer community.

It's fairly clear that one cannot code from the ground up in bazaar style [IN]. One can test, debug and improve in bazaar style, but it would be very hard to originate a project in bazaar mode. Linus didn't try it. I didn't either. Your nascent developer community needs to have something runnable and testable to play with.

When you start community-building, what you need to be able to present is a plausible promise. Your program doesn't have to work particularly well. It can be crude, buggy, incomplete, and poorly documented. What it must not fail to do is (a) run, and (b) convince potential co-developers that it can be evolved into something really neat in the foreseeable future.

Linux and fetchmail both went public with strong, attractive basic designs. Many people thinking about the bazaar model as I have presented it have correctly considered this critical, then jumped from that to the conclusion that a high degree of design intuition and cleverness in the project leader is indispensable.

But Linus got his design from Unix. I got mine initially from the ancestral popclient (though it would later change a great deal, much more proportionately speaking than has Linux). So does the leader/coordinator for a bazaar-style effort really have to have exceptional design talent, or can he get by through leveraging the design talent of others?

市集风格的必要前提

这篇文章的早期审阅者和试验听众们持续就成功的市集风格开发的前提提出问题，包括项目领导人的素质和他开放项目和开始搭建合作者社区时的程序代码状态。

很显然的，在市集风格里你不能从零开始编程。你可以在市集风格里检测、调试和提高，但是以市集模式孕育一个项目会是很困难的。林纳斯没有这样试过。我也没有。你的新生的开发者社区需要能运行和测试的东西来展示身手。

当你开始社区建设的时候，你需要能够呈现一个可行的前景。你的程序不一定要工作的非常好。它可以是粗糙的、问题多多的、不完整的、缺少文档记录的。它一定不能失败的是（1）能运行，（2）说服潜在的合作者它可以在可预见的将来进化成真正漂亮的东西。

Linux 和 fetchmail 开放的时候都带有强劲、吸引人的基本设计。许多像我的描述里那样来思考市集模式的人正确地认为这一点很关键；于是进而断定在项目领导人身上，高度的设计灵感和聪明不可或缺。

但是林纳斯的设计来自于 Unix。我的最初来自于先前的 popclient（尽管它后来变化很大，按比例来说比 Linux 的大的多）。那么一个市集风格的项目的领导人／主持人真的一定要有杰出的设计天才，还是他可以四两拨千斤、通过带动他人的设计才能来述职?

I think it is not critical that the coordinator be able to originate designs of exceptional brilliance, but it is absolutely critical that the coordinator be able to recognize good design ideas from others.

Both the Linux and fetchmail projects show evidence of this. Linus, while not (as previously discussed) a spectacularly original designer, has displayed a powerful knack for recognizing good design and integrating it into the Linux kernel. And I have already described how the single most powerful design idea in fetchmail (SMTP forwarding) came from somebody else.

Early audiences of this essay complimented me by suggesting that I am prone to undervalue design originality in bazaar projects because I have a lot of it myself, and therefore take it for granted. There may be some truth to this; design (as opposed to coding or debugging) is certainly my strongest skill.

But the problem with being clever and original in software design is that it gets to be a habit—you start reflexively making things cute and complicated when you should be keeping them robust and simple. I have had projects crash on me because I made this mistake, but I managed to avoid this with fetchmail.

So I believe the fetchmail project succeeded partly because I restrained my tendency to be clever; this argues (at least) against design originality being essential for successful bazaar projects. And consider Linux. Suppose Linus Torvalds had been trying to pull off fundamental innovations in operating system design during the development; does it seem at all likely that the resulting kernel would be as stable and successful as what we have?

A certain base level of design and coding skill is required, of course, but I expect almost anybody seriously thinking of launching a bazaar effort will already be above that minimum. The

我认为主持的人能否想出杰出灿烂的设计不是很关键，但绝对关键的是，主持的人能够慧眼识别出他人的优秀设计想法。

Linux 和 fetchmail 的项目都显示了这方面的证据。林纳斯，（像前面讨论过的）尽管不是一个特别有原创性的设计者，却展现了识别优秀设计并把它 集成到 Linux 内核里强大才能。我也已经描述了在 fetchmail 里的最有力的一个设计思想（SMTP 转发）怎样来自于另一个人。

这篇文章的早期听众捧我的场说我容易低估市集项目里的设计原创性的价值，因为我自己有很多，因而就想当然的习惯了。这话大概有一点点的真实性在里面；设计（而不是编码或调试）确实是我的强项。
但是在软件设计里表现聪明和创造力的问题在于它形成一种坏习惯——当你应该保持事情稳固和简单的时候，你开始放任地把它们搞的好玩和复杂。我曾经因为犯了这个错误把项目搞砸过，但是我在 fetchmail 里做到了避开这个错误。
所以我相信 fetchmail 项目的成功部分上是因为我克制住了自作聪明的习惯；这（至少）反驳了设计的原创性是成功的市集项目的关键。而且想一下Linux。假设林纳斯·托瓦兹在开发中试图整出操作系统设计的根本性创新；作出来的内核压根儿可能会像我们现在的这么稳定和成功吗?

当然一定的设计和编码技能的基本水准还是必要的，但是我预期几乎每个认真考虑发起一个市集型项目的人已经超

open-source community's internal market in reputation exerts subtle pressure on people not to launch development efforts they're not competent to follow through on. So far this seems to have worked pretty well.

There is another kind of skill not normally associated with software development which I think is as important as design cleverness to bazaar projects—and it may be more important. A bazaar project coordinator or leader must have good people and communications skills.

This should be obvious. In order to build a development community, you need to attract people, interest them in what you're doing, and keep them happy about the amount of work they're doing. Technical sizzle will go a long way towards accomplishing this, but it's far from the whole story. The personality you project matters, too.

It is not a coincidence that Linus is a nice guy who makes people like him and want to help him. It's not a coincidence that I'm an energetic extrovert who enjoys working a crowd and has some of the delivery and instincts of a stand-up comic. To make the bazaar model work, it helps enormously if you have at least a little skill at charming people.

出了这个基本要求。开源社区内部的声望机制给人们一种微妙的压力：［如果你没有几把刷子，］不要发起自己不能胜任的开发项目。迄今为止，这一点似乎工作得很有效。
另外有一种和软件开发一般无关的技能，我认为对于市集型项目来讲，和设计才能一样的重要——甚至可能更重要。一个市集项目的主持人或领导者必须有良好的人际、交流技能。

这应该是显而易见的。要建设一个开发社区，你需要吸引人群，让他们对你做的事情感兴趣，并且让他们对自个儿的工作量舒心。要做到这一点，巧妙的手段会起很大的作用，但远远不是故事的全部。你所展现的人格也很重要。

林纳斯是一个平易的人，让人们喜欢他、想帮助他——这不是巧合。我是个活泼外向的人，喜欢和人群打交道，有着一些现场喜剧演员的直觉和本事——这不是巧合。要使市集模式运行起来，你至少有一点点让人们喜欢你的本领是非常重要的。

The Social Context of Open-Source Software

It is truly written: the best hacks start out as personal solutions to the author's everyday problems, and spread because the problem turns out to be typical for a large class of users. This takes us back to the matter of rule 1, restated in a perhaps more useful way:

18. To solve an interesting problem, start by finding a problem that is interesting to you.

So it was with Carl Harris and the ancestral popclient, and so with me and fetchmail. But this has been understood for a long time. The interesting point, the point that the histories of Linux and fetchmail seem to demand we focus on, is the next stage—the evolution of software in the presence of a large and active community of users and co-developers.

In The Mythical Man-Month, Fred Brooks observed that programmer time is not fungible; adding developers to a late software project makes it later. As we've seen previously, he argued that the complexity and communication costs of a project rise with the square of the number of developers, while work done only rises linearly. Brooks's Law has been widely regarded as a truism. But we've examined in this essay an number of ways in which the process of open-source development falsifies the assumptionms behind it—and, empirically, if Brooks's Law were the whole picture Linux would be impossible.

Gerald Weinberg's classic The Psychology of Computer Programming supplied what, in hindsight, we can see as a vital correction to Brooks. In his discussion of ``egoless programming'', Weinberg observed that in shops where developers are not territorial about their code, and encourage other people to look for

开源软件的社会语境

这句话写到了实处：最好的程序开始于作者日常问题的个人解决方案，因为一大批人正好都有这个问题而流行。这把我们带回了第一条经验的内容，用一种或许更有用的方式来表达是：

18）要解决一个有意思的问题，首先找到一个你觉得有意思的问题。

卡尔·哈里斯和先前的 popclient 是这样的，我和 fetchmail 也是这样的。但是这点大家已经明白很久了。有意义的一点是，Linux 和 fetchmail 的历史看来要求我们关心的一点是，下一个阶段————在用户和合作者形成了庞大活跃的社区时的软件的进化。

在《人月神话》中，弗里德·布洛克表述了程序员的时间是不能用钱币购买的；添加开发人员的做法只能使得已经延期的软件项目更为延期。像我们前面提到 的，他论述了项目的复杂程度和通讯成本按开发人员数目的平方增加，而业绩仅以直线增加。布洛克法则被广泛的认同为真理。但在这篇文章里，我们已经探讨了开 源的开发过程破除它背后的预设的几种途径——而且事实证明，如果布洛克法则统领一切，Linux 就不可能发生。

以事后之明看来，杰拉德·委恩伯格的经典《计算机编程的心理学》提供了一个对布洛克的关键修正。在他对“无私编程”的讨论里，委恩伯格注意到一些地 方的开发人员不对自己的源码画地为牢，而是鼓励他人在其中寻找错误和改

bugs and potential improvements in it, improvement happens dramatically faster than elsewhere. (Recently, Kent Beck's `extreme programming' technique of deploying coders in pairs looking over one anothers' shoulders might be seen as an attempt to force this effect.)

Weinberg's choice of terminology has perhaps prevented his analysis from gaining the acceptance it deserved—one has to smile at the thought of describing Internet hackers as ``egoless". But I think his argument looks more compelling today than ever.

The bazaar method, by harnessing the full power of the ``egoless programming" effect, strongly mitigates the effect of Brooks's Law. The principle behind Brooks's Law is not repealed, but given a large developer population and cheap communications its effects can be swamped by competing nonlinearities that are not otherwise visible. This resembles the relationship between Newtonian and Einsteinian physics—the older system is still valid at low energies, but if you push mass and velocity high enough you get surprises like nuclear explosions or Linux.

The history of Unix should have prepared us for what we're learning from Linux (and what I've verified experimentally on a smaller scale by deliberately copying Linus's methods [EGCS]). That is, while coding remains an essentially solitary activity, the really great hacks come from harnessing the attention and brainpower of entire communities. The developer who uses only his or her own brain in a closed project is going to fall behind the developer who knows how to create an open, evolutionary context in which feedback exploring the design space, code contributions, bug-spotting, and other improvements come from from hundreds (perhaps thousands) of people.

进的余地——在这些地方，改进进展得比别处明显快很多。（最近，肯特·贝克的“极度编程”技术——把编程者配对让他们互相关照——或许可以看作是强制这一效果的尝试。）委恩伯格在用词上的选择可能阻碍了他的分析获得应有的认可——把网络黑客形容成不讲个人英雄主义的一群，光是这样的念头就足以令人莞尔。但是我认为他的争论在今天看起来比以往任何时间更让人信服。

市集模式，通过借助“无私编程”效果的极致动力，强烈抵制了布洛克法则的效果。布洛克法则背后的原理没有被推翻，但是有了一个庞大的开发者群体和廉 价的通讯，它的效果可以被否则不可见的对立的非线性因素所淹没。这就像牛顿式的和爱因斯坦式的物理之间的关系——旧的系统在低能量下仍然有效，但当你把质 量和速度变得足够大的时候，你就得到了如同核爆炸或 Linux 那样的惊奇。

Unix 的历史应该使得我们对研究 Linux 的结果（和我在小规模上有意拷贝林纳斯的方法所实验确认的结果）有了心理准备。这是说，虽 然编程基本上仍是一种个人封闭的活动，真正高超的程序来自于借助整个社区的注意力和脑力。一个在封闭的项目中只使用自己脑力的开发者，将会输给一个知道怎 样创造一个开放的、进化式的环境——从中吸收成千或上万人的探索设计空间的反馈、编码贡献、臭虫检测和其它的改进——的开发者。

But the traditional Unix world was prevented from pushing this approach to the ultimate by several factors. One was the legal contraints of various licenses, trade secrets, and commercial interests. Another (in hindsight) was that the Internet wasn't yet good enough.

Before cheap Internet, there were some geographically compact communities where the culture encouraged Weinberg's ``egoless'' programming, and a developer could easily attract a lot of skilled kibitzers and co-developers. Bell Labs, the MIT AI and LCS labs, UC Berkeley—these became the home of innovations that are legendary and still potent.

Linux was the first project for which a conscious and successful effort to use the entire world as its talent pool was made. I don't think it's a coincidence that the gestation period of Linux coincided with the birth of the World Wide Web, and that Linux left its infancy during the same period in 1993–1994 that saw the takeoff of the ISP industry and the explosion of mainstream interest in the Internet. Linus was the first person who learned how to play by the new rules that pervasive Internet access made possible.

While cheap Internet was a necessary condition for the Linux model to evolve, I think it was not by itself a sufficient condition. Another vital factor was the development of a leadership style and set of cooperative customs that could allow developers to attract co-developers and get maximum leverage out of the medium.

But what is this leadership style and what are these customs? They cannot be based on power relationships—and even if they could be, leadership by coercion would not produce the results we see. Weinberg quotes the autobiography of the 19th-century Russian anarchist Pyotr Alexeyvich Kropotkin's Memoirs of a

但是有几个因素阻止了传统的 Unix 世界把这个方法发挥到极致。一个是各种执照许可、贸易秘密和商业利益的法律限制。另一个（回头来看）是互联网还不够好。
在便宜的互联网之前有过一些地域性的紧密团体，在文化上鼓励委恩伯格的“无私编程”、一个开发者可以容易地吸引到一批有水平的“军师”和合作者。贝尔实验室、麻省理工的人工智能和计算机实验室、伯克利加州大学——这些成为了传奇性的和依然强劲的发明的家园。

Linux 是第一个作了有意识的、成功的努力来把全世界当作智囊库使用的项目。我不认为 Linux 的孕育期与互联网的诞生重叠是一个巧合，Linux 在 1993－1994 之间网络服务业起步和对互联网的主流兴趣的爆发的同期脱离了它的婴儿时代也不是巧合。林纳斯是第一个学会按照普及的互联网 连接所促成的新规则来运作的人。

虽然便宜的互联网是 Linux 模式进化出来的必要条件，我觉得它自己不是充分条件。另一个关键的因素是一种领导风格和一套合作制度的建立——使得开发者可以吸引合作者、在这个媒介中获取最大程度的收益。

但什么是这种领导风格、什么是这些制度呢？它们不可能是基于权力关系的——就算是可能，强制性的领导不会产生我们所看到的成果。在这个题目上，委恩 伯格很恰当的引用了 19 世纪俄罗斯无政府主义者 Pyotr Alexeyvich

Revolutionist to good effect on this subject:

> Having been brought up in a serf-owner's family, I entered active life, like all young men of my time, with a great deal of confidence in the necessity of commanding, ordering, scolding, punishing and the like. But when, at an early stage, I had to manage serious enterprises and to deal with [free] men, and when each mistake would lead at once to heavy consequences, I began to appreciate the difference between acting on the principle of command and discipline and acting on the principle of common understanding. The former works admirably in a military parade, but it is worth nothing where real life is concerned, and the aim can be achieved only through the severe effort of many converging wills.

The ``severe effort of many converging wills'' is precisely what a project like Linux requires—and the ``principle of command'' is effectively impossible to apply among volunteers in the anarchist's paradise we call the Internet. To operate and compete effectively, hackers who want to lead collaborative projects have to learn how to recruit and energize effective communities of interest in the mode vaguely suggested by Kropotkin's ``principle of understanding''. They must learn to use Linus's Law.[SP]

Earlier I referred to the ``Delphi effect'' as a possible explanation for Linus's Law. But more powerful analogies to adaptive systems in biology and economics also irresistably suggest themselves. The Linux world behaves in many respects like a free market or an ecology, a collection of selfish agents attempting to maximize utility which in the process produces a self-correcting spontaneous order more elaborate and efficient than any amount of central planning could have achieved. Here, then, is the place to seek the ``principle of understanding''.

Kropotkin 的自传《一个革命者的回忆录》：

> 成长于一个农奴主的家庭，我进入社会后，像我那个时候所有的年轻人一样，很是相信领导、命令、训斥、惩罚等等的必要性。但是在早期我不得不管理重要 的事业和对付［自由的］人们的时候，在每个错误都会立刻导致严重后果的时候，我开始领悟到按指令和纪律的原则行事与按共同理解的原则行事之间的区别。前者 在阅兵式中运行得令人崇敬，然而就真实的生活而言，它却一文不值；而且目标只有通过许多共同意志的竭诚努力才能实现。

这个“许多共同意志的竭诚努力”正是 Linux 这种项目所要求的；在这个我们叫作互联网的无政府主义者的天堂里，对志愿者们行使“指令的原则”事实 上是不可能的。要有效的运作和竞争，想要领导协作性项目的黑客们不得不学会怎样按照 Kropotkin 的“共同理解的原则”里模糊地提出的模式、招募和激 励活动中的相关社区。他们必须学会使用林纳斯法则。

我早先引用了“神庙效应”作为林纳斯法则的一个可行的解释。但是与生物和经济中的可适应系统的更有力的类比不可避免地自我呈现出来。Linux 世界 在很多方面都表现的类似自由市场或生态系统：由自私的成员组成的一个试图把功效最大化的集合，在这个过程中形成了一个自我纠正的自发秩序——它比不管多少 中央计划有可能达到的成就都更广泛和有效。那么，这里就是寻找“共同理解的原则”的地方。

The ``utility function'' Linux hackers are maximizing is not classically economic, but is the intangible of their own ego satisfaction and reputation among other hackers. (One may call their motivation ``altruistic'', but this ignores the fact that altruism is itself a form of ego satisfaction for the altruist). Voluntary cultures that work this way are not actually uncommon; one other in which I have long participated is science fiction fandom, which unlike hackerdom has long explicitly recognized ``egoboo'' (ego-boosting, or the enhancement of one's reputation among other fans) as the basic drive behind volunteer activity.

Linus, by successfully positioning himself as the gatekeeper of a project in which the development is mostly done by others, and nurturing interest in the project until it became self-sustaining, has shown an acute grasp of Kropotkin's ``principle of shared understanding''. This quasi-economic view of the Linux world enables us to see how that understanding is applied.

We may view Linus's method as a way to create an efficient market in ``egoboo''—to connect the selfishness of individual hackers as firmly as possible to difficult ends that can only be achieved by sustained cooperation. With the fetchmail project I have shown (albeit on a smaller scale) that his methods can be duplicated with good results. Perhaps I have even done it a bit more consciously and systematically than he.

Many people (especially those who politically distrust free markets) would expect a culture of self-directed egoists to be fragmented, territorial, wasteful, secretive, and hostile. But this expectation is clearly falsified by (to give just one example) the stunning variety, quality, and depth of Linux documentation. It is a hallowed given that programmers hate documenting; how is it, then, that Linux hackers generate so much documentation? Evidently Linux's free market in egoboo works better to produce

Linux 黑客们最大化的这个“功效方程”不是经典经济学上的，而是他们自我的满足和在其他黑客中的声望这些摸不到的东西。（有人或许可以把他们的 动机叫作“利他性的”，但这忽略了利他主义自身对利他主义者就是一种自我满足的形式这一事实。）这样运作的自愿者文化其实不是不寻常的；我已经长期参与的 另一个就是科幻粉丝族。不像黑客族，他们早就明白的认识到了“egoboo”（ego-boosting，或在其他粉丝中增强个人的声望）是志愿者活动背 后的基本动力。

林纳斯，通过成功的把自己定位为一个主要由其他人来开发的项目的看门人、培养对这个项目的兴趣直到它可以自我维持，表现了对 Kropotkin“共同理解的原则”的敏锐把握。这个对 Linux 世界的类经济学视角使得我们能够看到这种理解是如何应用的。

我们可以把林纳斯的方法看作创造有效的“自我彰显”的市场的一条路子——把单个黑客的自我体现尽可能紧密的连接在困难的、只有通过持续合作才能达到 的目标上。通过 fetchmail 项目，我展现了（尽管在小规模上）他的方法可以复制、产生出好的结果。或许我甚至做得比他更有意识和更系统一点。

很多人（尤其是在政治上不相信自由市场的那些）会以为自我驱动的个人英雄主义者们形成的文化会是支离破碎、占山为王、低效浪费、秘密诡异和充满敌意的。但 是这个设想显然被（只举一个例子）Linux 文档的惊人的广度、质量和深度所证伪。程序员痛恨写文档是众所周知的；那么，Linux 黑客们是怎样生产出这 么多文档的呢？显然林纳斯的

virtuous, other-directed behavior than the massively-funded documentation shops of commercial software producers.

Both the fetchmail and Linux kernel projects show that by properly rewarding the egos of many other hackers, a strong developer/coordinator can use the Internet to capture the benefits of having lots of co-developers without having a project collapse into a chaotic mess. So to Brooks's Law I counter-propose the following:

19: Provided the development coordinator has a communications medium at least as good as the Internet, and knows how to lead without coercion, many heads are inevitably better than one.

I think the future of open-source software will increasingly belong to people who know how to play Linus's game, people who leave behind the cathedral and embrace the bazaar. This is not to say that individual vision and brilliance will no longer matter; rather, I think that the cutting edge of open-source software will belong to people who start from individual vision and brilliance, then amplify it through the effective construction of voluntary communities of interest.

Perhaps this is not only the future of open-source software. No closed-source developer can match the pool of talent the Linux community can bring to bear on a problem. Very few could afford even to hire the more than 200 (1999: 600, 2000: 800) people who have contributed to fetchmail!

Perhaps in the end the open-source culture will triumph not because cooperation is morally right or software ``hoarding'' is morally wrong (assuming you believe the latter, which neither Linus nor I do), but simply because the closed-source world cannot win an evolutionary arms race with open-source

自我彰显的自由市场在产出优良的、协同的行为上优于商业软件厂商的大笔资金驱动的文档工厂。

fetchmail 项目和 Linux 内核项目都表明，通过适当的表彰众多其他黑客的 egos，一个有力的开发者／协调者可以使用互联网来收获拥有许多合作者的好处，而不至于让项目陷入嘈杂的混乱。所以针对布洛克法则，我反过来建议下面的一条：

19）如果开发的协调者有一个至少和互联网一样好的通讯媒介，而且懂得如何不通过强迫来领导，多个头脑不可避免地优于单个头脑。

我认为开源软件的未来会更多的属于那些懂得如何运行林纳斯的游戏的人们，告别大教堂来拥抱市集的人们。这不是说个人的远见和才华不再重要；而是，就我看来开源软件的前沿会属于那些开始于个人的远见和才华、然后通过有效的建设相关志愿者社区来扩展放大的人们。

或许这不仅是开源软件的未来。要对付一个问题，没有闭源的开发者可以比得上 Linux 社区所能驱动的才能之众。极少有人甚至能雇得起那些对 fetchmail 作出了贡献的 200（1999: 600, 2000: 800）多人！

开源文化会最终胜利，或许不是因为合作在道德上正确或软件“劳役”在道德上错误（假设你相信后者，林纳斯和我都不），而只是因为开源社区可以在一个问题上投入多几个数量级的技术工时、闭源世界无法赢得这场进化式的军备竞争。

communities that can put orders of magnitude more skilled time into a problem.

On Management and the Maginot Line

The original Cathedral and Bazaar paper of 1997 ended with the vision above—that of happy networked hordes of programmer/anarchists outcompeting and overwhelming the hierarchical world of conventional closed software.

A good many skeptics weren't convinced, however; and the questions they raise deserve a fair engagement. Most of the objections to the bazaar argument come down to the claim that its proponents have underestimated the productivity-multiplying effect of conventional management.

Traditionally-minded software-development managers often object that the casualness with which project groups form and change and dissolve in the open-source world negates a significant part of the apparent advantage of numbers that the open-source community has over any single closed-source developer. They would observe that in software development it is really sustained effort over time and the degree to which customers can expect continuing investment in the product that matters, not just how many people have thrown a bone in the pot and left it to simmer.

There is something to this argument, to be sure; in fact, I have developed the idea that expected future service value is the key to the economics of software production in the essay The Magic Cauldron.

关于管理和马其诺防线

原始的 1997 年的《大教堂和市集》论文以以上的预见结束——程序员／无政府主义者的幸福网络群体胜出并压倒了传统闭源软件的阶层化世界。

然而，很多怀疑者并不信服；他们提出的问题也值得一场像样的论战。多数对市集模式的反对归结到一个观点：它的鼓吹者们低估了传统管理促进生产率的效果。

老脑筋的软件开发经理经常指责开源世界里项目群体形成－转变－消亡的随意性大大抵消了开源社区对单个闭源开发者在数目上的显然优越性。他们会指出在软件开 发里，真正重要的是长时间里不懈的努力和多大程度上顾客可以预期对产品的持续投资，而不仅仅是多少人往锅里扔一块骨头让它炖着。

没有疑问，这条争论有料；实际上，我在"The Magic Cauldron"一文中就已经展示了预期的未来服务价值是软件产业经济的关键的观点。

But this argument also has a major hidden problem; its implicit assumption that open-source development cannot deliver such sustained effort. In fact, there have been open-source projects that maintained a coherent direction and an effective maintainer community over quite long periods of time without the kinds of incentive structures or institutional controls that conventional management finds essential. The development of the GNU Emacs editor is an extreme and instructive example; it has absorbed the efforts of hundreds of contributors over 15 years into a unified architectural vision, despite high turnover and the fact that only one person (its author) has been continuously active during all that time. No closed-source editor has ever matched this longevity record.

This suggests a reason for questioning the advantages of conventionally-managed software development that is independent of the rest of the arguments over cathedral vs. bazaar mode. If it's possible for GNU Emacs to express a consistent architectural vision over 15 years, or for an operating system like Linux to do the same over 8 years of rapidly changing hardware and platform technology; and if (as is indeed the case) there have been many well-architected open-source projects of more than 5 years duration -- then we are entitled to wonder what, if anything, the tremendous overhead of conventionally-managed development is actually buying us.

Whatever it is certainly doesn't include reliable execution by deadline, or on budget, or to all features of the specification; it's a rare `managed' project that meets even one of these goals, let alone all three. It also does not appear to be ability to adapt to changes in technology and economic context during the project lifetime, either; the open-source community has proven far more effective on that score (as one can readily verify, for example, by

但是这条争论也有一个主要的潜在问题：它的暗中假设是开源开发不能形成持续的努力。事实上，有的开源项目在很长的时间段里保持了一致的方向和有效的维护团 体，而不需要传统管理上觉得关键的那些奖掖性的结构或制度性的控制。GNU Emacs 编辑器的开发是一个极端的、说明问题的例子：它在超出 15 年的时间里吸取了成百上千贡献者的劳动、形成了一个统一的结构设计，尽管人事变换频 繁、一贯持续下来的人只有一个（它的作者）。没有闭源的编辑器或曾比得上这个长寿记录。

这建议了一个质疑传统管理模式下软件开发的优势的理由，与其它关于大教堂和市集模式的争议不相干的理由。如果 GNU Emacs 可以在 15 年里表述一个一致的框架设计，或者一个像 Linux 的操作系统在 8 年多里快速变化的硬件和平台技术中作到了同一点；如果（事实的确如 此）存在许多设计优良的开源项目超出了 5 年的历史——那么我们有权发问传统管理开发的庞大开销给我们买来了什么——如果有的话。

不管是什么，它显然不包括按期限、或预算、或所有指定功能的可靠执行；能满足这些目标中仅仅一条就是一个罕见的“管理好”的项目了，更不用说全部三 条了。它看来也不包括在项目进行过程中适应技术和经济环境变化的能力；在这上面，开源社区证明了远远更为有效（来简单确认这

comparing the 30-year history of the Internet with the short half-lives of proprietary networking technologies—or the cost of the 16-bit to 32-bit transition in Microsoft Windows with the nearly effortless upward migration of Linux during the same period, not only along the Intel line of development but to more than a dozen other hardware platforms, including the 64-bit Alpha as well).

One thing many people think the traditional mode buys you is somebody to hold legally liable and potentially recover compensation from if the project goes wrong. But this is an illusion; most software licenses are written to disclaim even warranty of merchantability, let alone performance—and cases of successful recovery for software nonperformance are vanishingly rare. Even if they were common, feeling comforted by having somebody to sue would be missing the point. You didn't want to be in a lawsuit; you wanted working software.

So what is all that management overhead buying?

In order to understand that, we need to understand what software development managers believe they do. A woman I know who seems to be very good at this job says software project management has five functions:

- To define goals and keep everybody pointed in the same direction
- To monitor and make sure crucial details don't get skipped
- To motivate people to do boring but necessary drudgework
- To organize the deployment of people for best productivity
- To marshal resources needed to sustain the project

点，比方说，可以比较互联网30 年的历史和私有网络技术短短的半衰期；或者比较微软视窗从 16 位转换为 32 位的成本和 Linux 同一时期几乎毫不费力的升级——不仅是围绕英特尔系列 的开发，而且也扩展到包括 64 位 Alpha 芯片的十多个其他硬件平台上）。

很多人觉得从传统模式中购买到的一件东西是有人负法律责任、如果事情出了问题可能找赔偿。但这是一个幻觉；大多数的软件协议是写了来免除甚至商品化 的保证，更不用说性能了——而且从软件性能问题上成功索赔的案例少得近于虚幻。即使这类事很平常，因为有人来打官司而觉得心安是搞错了要点。您不想打官 司；您要的是能工作的软件。

那么这些管理开销都买了什么呢?

要理解这个问题，我们需要理解管理软件开发的经理们是怎么想的。我认识的一个似乎很优秀的女经理说软件项目管理有五项功能:

- 明确目标并保持大家向同一个方向努力
- 监测并确保关键的细节不被漏掉
- 动员人们做枯燥但是必要的苦力活儿
- 组织人员分配来达到最佳生产力
- 监护项目持续所需要的资源

Apparently worthy goals, all of these; but under the open-source model, and in its surrounding social context, they can begin to seem strangely irrelevant. We'll take them in reverse order.

My friend reports that a lot of resource marshalling is basically defensive; once you have your people and machines and office space, you have to defend them from peer managers competing for the same resources, and from higher-ups trying to allocate the most efficient use of a limited pool.

But open-source developers are volunteers, self-selected for both interest and ability to contribute to the projects they work on (and this remains generally true even when they are being paid a salary to hack open source.) The volunteer ethos tends to take care of the `attack' side of resource-marshalling automatically; people bring their own resources to the table. And there is little or no need for a manager to `play defense' in the conventional sense.

Anyway, in a world of cheap PCs and fast Internet links, we find pretty consistently that the only really limiting resource is skilled attention. Open-source projects, when they founder, essentially never do so for want of machines or links or office space; they die only when the developers themselves lose interest.

That being the case, it's doubly important that open-source hackers organize themselves for maximum productivity by self-selection—and the social milieu selects ruthlessly for competence. My friend, familiar with both the open-source world and large closed projects, believes that open source has been successful partly because its culture only accepts the most talented 5% or so of the programming population. She spends most of her time organizing the deployment of the other 95%, and has thus observed first-hand the well-known variance of a factor of one hundred in productivity between the most able programmers and

显然是有价值的目标，所有这些都是；但是在开源模式下，并且在它周围的社会语境中，这些会奇怪地开始显得不相干。我们来按逆序讨论这几条。

我的朋友报告说许多资源监护基本是防卫性的；一旦你有了你的人员机器和办公空间，你不得不防卫其他经理竞争相同的资源，和防卫上级调用有限资源中最高效的部分。但是开源开发者是志愿的、在兴趣和对所参与项目的贡献能力上自我挑选的（甚至在他们领了薪水为开源项目编码的情况下这条也一般适用）。志愿者的特点会自动解决资源监护的“攻击方”；人们把自己的资源带到桌面上来。而且对管理者来说，没有或几乎没有必要来作传统意义上的“防卫”。

不管怎样，在一个便宜电脑和快速互联网连接的世界里，我们很一致的发现真正唯一的稀缺资源是有技术的努力。开源项目本质上从不会为了争夺机器或网络或办公空间而成立；它们只在开发者自己失掉兴趣的时候消亡。

在这点之外，加倍重要的是开源黑客们通过自我选择组织达到最大生产率——社会环境也无情的对能力作选择。我的朋友，对开源世界和大型闭源项目都熟 悉，认为开源的成功部分归功于它的文化只接受编程人员中最有才华的 5% 左右。她花费了她大多的时间来组织部署其他的 95%，于是第一手观察到了那著名的、 最优秀的和仅仅及格的程序员之间一百倍的效率差别。

the merely competent.

The size of that variance has always raised an awkward question: would individual projects, and the field as a whole, be better off without more than 50% of the least able in it? Thoughtful managers have understood for a long time that if conventional software management's only function were to convert the least able from a net loss to a marginal win, the game might not be worth the candle.

The success of the open-source community sharpens this question considerably, by providing hard evidence that it is often cheaper and more effective to recruit self-selected volunteers from the Internet than it is to manage buildings full of people who would rather be doing something else.

Which brings us neatly to the question of motivation. An equivalent and often-heard way to state my friend's point is that traditional development management is a necessary compensation for poorly motivated programmers who would not otherwise turn out good work.

This answer usually travels with a claim that the open-source community can only be relied on only to do work that is `sexy' or technically sweet; anything else will be left undone (or done only poorly) unless it's churned out by money-motivated cubicle peons with managers cracking whips over them. I address the psychological and social reasons for being skeptical of this claim in Homesteading the Noosphere. For present purposes, however, I think it's more interesting to point out the implications of accepting it as true.

If the conventional, closed-source, heavily-managed style of software development is really defended only by a sort of Maginot Line of problems conducive to boredom, then it's going to remain

这个差别的巨大程度总是引发一个尴尬的问题：不管单个的项目还是整个行业，甩脱了那50%以上最差的是不是会更好一些？肯动脑的管理者很久以来就懂得，如果传统软件管理的唯一功能是把最差的一帮人从净损失转为微盈利，这事儿恐怕就不值得折腾。

开源社区的成功，通过提供硬性证据显示从互联网上招募自我选择的志愿者经常要比管理整栋楼的“身在曹营心在汉”的人们要便宜和有效的多，在相当程度上尖锐化了这个问题。

这恰好把我们带到驱动力的问题。一个同等的、常见的方式来提出我朋友的观点是：传统开发管理是对缺乏动力的程序员的必要补充，不然他们做不好工作。

这个回答一般携带着一个说法：只能指望开源社区做那种“眩目”的或技术上好玩的工作；任何其它的会被拉下（或敷衍其事）——除非金钱驱动的坐隔间的人在经 理们的鞭策下把它搅出来。我在“Homesteading the Noosphere”里解释对这个说法怀疑的心理学和社会学原因。然而就当前的目的，我觉得指出如果假定它是正确而衍生出的推论更有意思一些。

如果传统的、闭源的、冗肿管理下的软件开发真的只是在枯燥引发的问题造成的一条马其诺防线的守卫下，那么它

viable in each individual application area for only so long as nobody finds those problems really interesting and nobody else finds any way to route around them. Because the moment there is open-source competition for a `boring' piece of software, customers are going to know that it was finally tackled by someone who chose that problem to solve because of a fascination with the problem itself—which, in software as in other kinds of creative work, is a far more effective motivator than money alone.

Having a conventional management structure solely in order to motivate, then, is probably good tactics but bad strategy; a short-term win, but in the longer term a surer loss.

So far, conventional development management looks like a bad bet now against open source on two points (resource marshalling, organization), and like it's living on borrowed time with respect to a third (motivation). And the poor beleaguered conventional manager is not going to get any succour from the monitoring issue; the strongest argument the open-source community has is that decentralized peer review trumps all the conventional methods for trying to ensure that details don't get slipped.

Can we save defining goals as a justification for the overhead of conventional software project management? Perhaps; but to do so, we'll need good reason to believe that management committees and corporate roadmaps are more successful at defining worthy and widely shared goals than the project leaders and tribal elders who fill the analogous role in the open-source world.

That is on the face of it a pretty hard case to make. And it's not so much the open-source side of the balance (the longevity of Emacs, or Linus Torvalds's ability to mobilize hordes of developers with talk of ``world domination'') that makes it tough.

在每一个应用领域里的寿命就只有指望没有人发现 这些问题真正有意思、而且没有他人发现绕道而行的方法。因为一旦一种“枯燥”的软件出现了开源的竞争者，顾客们会知道终于有人因为关注这个问题本身而选择 来解决它了——这一点，对软件业像对其它任何的创造性工作一样，是比单独的金钱远为有效的驱动力。

仅仅为了驱动力来要一个传统的管理结构，或许就是一个好的计谋、坏的策略，短期的利益、长期的必然亏损。

说到这里，传统式开发管理和开源相比，现在在两点（资源监护，组织）上看起来不是明智之选，而且在第三点（驱动力）上朝不保夕。可怜的受困的传统项 目经理也不会在监测一项上得到任何帮助；开源社区的最强大的一个证据就是非集中化了的同行评议在试图保证细节不被忽略上胜出了所有的传统方法。

我们可以把目标的定义留下来作为接受传统软件项目管理的成本的理由吗？可能吧；但是这样，我们需要好的理由来相信管理委员会们和公司路线图比开源世界里的扮演类似角色的项目领导和部落长者们在定义有价值的、广泛共享的目标上更为成功。

就面上来看，这很难讲得通的。困难没有多少是来自于对峙的开源一方（Emacs 的长寿，或林纳斯·托瓦兹以“统领世界”的说辞动员大群的开发者的能力）。而是来自于传

Rather, it's the demonstrated awfulness of conventional mechanisms for defining the goals of software projects.

One of the best-known folk theorems of software engineering is that 60% to 75% of conventional software projects either are never completed or are rejected by their intended users. If that range is anywhere near true (and I've never met a manager of any experience who disputes it) then more projects than not are being aimed at goals that are either (a) not realistically attainable, or (b) just plain wrong.

This, more than any other problem, is the reason that in today's software engineering world the very phrase ``management committee" is likely to send chills down the hearer's spine—even (or perhaps especially) if the hearer is a manager. The days when only programmers griped about this pattern are long past; Dilbert cartoons hang over executives' desks now.

Our reply, then, to the traditional software development manager, is simple—if the open-source community has really underestimated the value of conventional management, why do so many of you display contempt for your own process?

Once again the example of the open-source community sharpens this question considerably—because we have fun doing what we do. Our creative play has been racking up technical, market-share, and mind-share successes at an astounding rate. We're proving not only that we can do better software, but that joy is an asset.

Two and a half years after the first version of this essay, the most radical thought I can offer to close with is no longer a vision of an open-source–dominated software world; that, after all, looks plausible to a lot of sober people in suits these days.

统机制在定义软件项目目标上表现出来的尴尬。

软件工程里一个最出名的大众定理是60%到75%的传统软件项目要么从没完成过，要么被目标用户否决了。要是这个范围真的和事实沾边（我从没有遇到过一个有经验的管理者否定过这点），那么占多数的项目都瞄向了（a）现实里达不到的或（b）错得离谱的目标。

在今天软件工程的世界里，“管理委员会”这个词会让听者背上直冒冷气——甚至（或者尤其）当听者是管理者的时候；上述这一点比其它任何的问题都是更主要的原因。那些只有程序员们咒骂这一现象的日子早已过去了；迪尔伯特的卡通✲如今挂上了管理层的案头。

我们对传统软件开发经理的回答，那么就很简单——如果开源社区真的低估了传统管理的价值，为什么你们这么多人表现了对你们自己工作的轻蔑?

开源社区的例子再次把这个问题尖锐化了——因为我们做事乐在其中。我们创新的游戏已在技术、市场占有和观念的成功上以惊人的速率得分晋级。我们在证明不仅我们可以开发更好的软件，而且欢乐是一种宝贵财产。

在这篇文章第一版的两年半后，我能用来结尾的最激进的观点不再是一个开源统领的软件世界；那毕竟，在今天很多穿西服套装的人看来也是有可能的了。

✲[译注]迪尔伯特是美国著名卡通系列的人物；该系列的主题是技术人员对管理层的揶揄。

Rather, I want to suggest what may be a wider lesson about software, (and probably about every kind of creative or professional work). Human beings generally take pleasure in a task when it falls in a sort of optimal-challenge zone; not so easy as to be boring, not too hard to achieve. A happy programmer is one who is neither underutilized nor weighed down with ill-formulated goals and stressful process friction. Enjoyment predicts efficiency.

Relating to your own work process with fear and loathing (even in the displaced, ironic way suggested by hanging up Dilbert cartoons) should therefore be regarded in itself as a sign that the process has failed. Joy, humor, and playfulness are indeed assets; it was not mainly for the alliteration that I wrote of "happy hordes" above, and it is no mere joke that the Linux mascot is a cuddly, neotenous penguin.

It may well turn out that one of the most important effects of open source's success will be to teach us that play is the most economically efficient mode of creative work..

相反，我想提出一个或许更广泛的、对软件业的教训，（或许也是对于任何一种创造性的或专业性的工作）。人们一般在一项任务处于一种适当难度范围的时 候享有乐趣；不要太简单了至于无聊，不要太难了不好实现。一个快乐程序员是一个既没有被浪费也没有被错误制定的目标和烦人过程摩擦所压倒的人。乐趣通往效 率。

以畏惧和厌恶来谈论你自己的工作过程（即使通过悬挂迪尔伯特卡通这种改头换面的讽刺性方式）因此本身应该被看作一个过程失败了的信号。欢乐、幽默，和趣味 是真正的财富；我在上面写的关于“快乐的一群”主要不是为了押韵，Linux 的吉祥物是一个可亲的、稚气犹存的企鹅也不仅仅是玩笑。

结果很可能是，开源的成功带来的一个最重要的影响会是教育我们乐趣是创造性工作的经济上最有效的模式。

Epilog: Netscape Embraces the Bazaar

It's a strange feeling to realize you're helping make history....

On January 22 1998, approximately seven months after I first published The Cathedral and the Bazaar, Netscape Communications, Inc. announced plans to give away the source for Netscape Communicator. I had had no clue this was going to happen before the day of the announcement.

Eric Hahn, executive vice president and chief technology officer at Netscape, emailed me shortly afterwards as follows: ``On behalf of everyone at Netscape, I want to thank you for helping us get to this point in the first place. Your thinking and writings were fundamental inspirations to our decision."

The following week I flew out to Silicon Valley at Netscape's invitation for a day-long strategy conference (on 4 Feb 1998) with some of their top executives and technical people. We designed Netscape's source-release strategy and license together.

A few days later I wrote the following:

Netscape is about to provide us with a large-scale, real-world test of the bazaar model in the commercial world. The open-source culture now faces a danger; if Netscape's execution doesn't work, the open-source concept may be so discredited that the commercial world won't touch it again for another decade.

On the other hand, this is also a spectacular opportunity. Initial reaction to the move on Wall Street and elsewhere has been cautiously positive. We're being given a chance to prove ourselves, too. If Netscape regains substantial market share through this move, it just may set off a long-overdue revolution in the software industry.

后记：网景欢迎市集

与历史同行，那是一种奇特的感觉……

1998 年 1 月 22 日，大概在我发表了“大教堂和市集”七个月后，网景通讯公司宣布了开放网景浏览器源代码的计划。在这个宣布之前我一点都不知道有这件事的迹象。艾瑞克。邯，网景的执行副总和首席技术长官，在那之后不久发给我这样一封电邮：“我希望代表网景的每一个人感谢您带头帮助我们走到了这一步。您的思考和写作对我们的决定是关键性的启迪。”

接下来的星期我接受网景的邀请飞到硅谷，和他们的高层管理和技术人员进行了一个整天的战略性会议（1998 年 2 月 4 日）。我们一起制定了网景的代码开发计划和发放执照。

几天后我写道：

网景将要给我们提供一个商业世界里的对市集模式的大型的、现实的测试。开源文化现在面对一个危险；如果网景的操作失败了，开放源代码的概念会信用扫地，商界在之后的十年内都不会再碰它。

另一方面，这也是一个绝好的机会。华尔街和其他地方的初步反应是谨慎的赞成。我们也得到一个证明自己的机会。如果网景通过这一举措夺回一定的市场份额，或许会触发一场软件业等待已久的革命。

The next year should be a very instructive and interesting time.

And indeed it was. As I write in mid-2000, the development of what was later named Mozilla has been only a qualified success. It achieved Netscape's original goal, which was to deny Microsoft a monopoly lock on the browser market. It has also achieved some dramatic successes (notably the release of the next-generation Gecko rendering engine).

However, it has not yet garnered the massive development effort from outside Netscape that the Mozilla founders had originally hoped for. The problem here seems to be that for a long time the Mozilla distribution actually broke one of the basic rules of the bazaar model; it didn't ship with something potential contributors could easily run and see working. (Until more than a year after release, building Mozilla from source required a license for the proprietary Motif library.)

Most negatively (from the point of view of the outside world) the Mozilla group didn't ship a production-quality browser for two and a half years after the project launch—and in 1999 one of the project's principals caused a bit of a sensation by resigning, complaining of poor management and missed opportunities. ``Open source,'' he correctly observed, ``is not magic pixie dust.''

And indeed it is not. The long-term prognosis for Mozilla looks dramatically better now (in November 2000) than it did at the time of Jamie Zawinski's resignation letter—in the last few weeks the nightly releases have finally passed the critical threshold to production usability. But Jamie was right to point out that going open will not necessarily save an existing project that suffers from ill-defined goals or spaghetti code or any of the software engineering's other chronic ills. Mozilla has managed to

接下来的一年会很有指导意义也很有意思。

确实是这样。当我写在 2000 年中期的时候，后来命名为 Mozilla 的开发项目只算是及格的成功。它达到了网景的最初目标——阻止微软在浏览器市场的垄断锁定。它也达到了一些显著的成功（尤其是下一代 Gecko 转换引擎的发布）。

然而，它还没有召集到网景之外的、Mozilla 创办者们起初所期冀的那种开发规模。这里的问题似乎是，在很长的一段时间里，Mozella 的发布实际上 破坏了市集模式的一条基本规则；它没有发放一个潜在的参与者可以轻易运行和眼观其效的东西。（直到发布后一年多，编译 Mozzila 需要一个非开放的Motif 库的执照。）

最消极的是（从外部世界的角度来看），Mozilla 团队在项目开始后两年半里没有发布出一个工业质量的浏览器——而且在 1999 年，一个项目骨干的辞职引起了不小的影响。他抱怨管理不力，错失良机。“开源”，他正确地评论道，“不能点石成金”。

确实不能。现在（2000 年 11 月）Mozilla 项目经过长期恢复，比起当初杰米。赞维斯基辞职的时候看起来有了戏剧性提高——最近几个星期的连夜发布 终于跨过了生产性使用的关键门坎。但是杰米正确地指出了走开源路线并不一定会挽救一个目标错乱、或编码堆面条、或患有其它软件工

provide an example simultaneously of how open source can succeed and how it could fail.

In the mean time, however, the open-source idea has scored successes and found backers elsewhere. Since the Netscape release we've seen a tremendous explosion of interest in the open-source development model, a trend both driven by and driving the continuing success of the Linux operating system. The trend Mozilla touched off is continuing at an accelerating rate.

程的慢性病的已有项目。Mozilla 成为了一个同时展示开源如何成功和如何失败的案例。

然而与此同时，开源的理念已经在其它地方获得了成功和支持。自从网景的计划公布以来，我们目睹了对开源开发模式的兴趣的爆炸式增长——Linux 操作系统的持续成功既是驱动方也是收益方。这个由 Mozilla 触发的潮流正在加速前进。*

* ［译者按］尽管网景后来难敌微软垄断性的重压，几年后从 Mozilla 中浴火重生的 Firefox 再次证明了开源社区的能量。Thomas Friedman 在他 2005 年的畅销书《The World is Flat》中把网景列为全球化的十大动力之一，因为网景为互联网的普及作出了奠基性的贡献。

Notes

[JB] In Programing Pearls, the noted computer-science aphorist Jon Bentley comments on Brooks's observation with ``If you plan to throw one away, you will throw away two.". He is almost certainly right. The point of Brooks's observation, and Bentley's, isn't merely that you should expect first attempt to be wrong, it's that starting over with the right idea is usually more effective than trying to salvage a mess.

[QR] Examples of successful open-source, bazaar development predating the Internet explosion and unrelated to the Unix and Internet traditions have existed. The development of the info-Zip compression utility during 1990–x1992, primarily for DOS machines, was one such example. Another was the RBBS bulletin board system (again for DOS), which began in 1983 and developed a sufficiently strong community that there have been fairly regular releases up to the present (mid-1999) despite the huge technical advantages of Internet mail and file-sharing over local BBSs. While the info-Zip community relied to some extent on Internet mail, the RBBS developer culture was actually able to base a substantial on-line community on RBBS that was completely independent of the TCP/IP infrastructure.

[CV] That transparency and peer review are valuable for taming the complexity of OS development turns out, after all, not to be a new concept. In 1965, very early in the history of time-sharing operating systems, Corbató and Vyssotsky, co-designers of the Multics operating system, wrote

It is expected that the Multics system will be published when it is operating substantially... Such publication is desirable for two reasons: First, the system should withstand public scrutiny and criticism volunteered by interested readers; second, in an age of increasing complexity, it is an obligation to present and future system designers to make the inner operating system as lucid as possible so as to reveal the basic system issues.

[JH] John Hasler has suggested an interesting explanation for the fact that duplication of effort doesn't seem to be a net drag on open-source development. He proposes what I'll dub ``Hasler's Law'': the costs of duplicated work tend to scale sub-qadratically with team size—that is, more slowly than the planning and management overhead that would be needed to eliminate them.

This claim actually does not contradict Brooks's Law. It may be the case that total complexity overhead and vulnerability to bugs scales with the square of team size, but that the costs from duplicated work are nevertheless a special case that scales more slowly. It's not hard to develop plausible reasons for this, starting with the undoubted fact that it is much easier to agree on functional boundaries between different developers' code that will prevent duplication of effort than it is to prevent the kinds of unplanned bad interactions across the whole system that underly most bugs.

The combination of Linus's Law and Hasler's Law suggests that there are actually three critical size regimes in software projects. On small projects (I would say one to at most three developers) no management structure more elaborate than picking a lead programmer is needed. And there is some intermediate range above that in which the cost of traditional management is relatively low, so its benefits from avoiding duplication of effort, bug-tracking, and pushing to see that details are not overlooked actually net out positive.

Above that, however, the combination of Linus's Law and Hasler's Law suggests there is a large-project range in which the

costs and problems of traditional management rise much faster than the expected cost from duplication of effort. Not the least of these costs is a structural inability to harness the many-eyeballs effect, which (as we've seen) seems to do a much better job than traditional management at making sure bugs and details are not overlooked. Thus, in the large-project case, the combination of these laws effectively drives the net payoff of traditional management to zero.

[HBS] The split between Linux's experimental and stable versions has another function related to, but distinct from, hedging risk. The split attacks another problem: the deadliness of deadlines. When programmers are held both to an immutable feature list and a fixed drop-dead date, quality goes out the window and there is likely a colossal mess in the making. I am indebted to Marco Iansiti and Alan MacCormack of the Harvard Business School for showing me me evidence that relaxing either one of these constraints can make scheduling workable.

One way to do this is to fix the deadline but leave the feature list flexible, allowing features to drop off if not completed by deadline. This is essentially the strategy of the "stable" kernel branch; Alan Cox (the stable-kernel maintainer) puts out releases at fairly regular intervals, but makes no guarantees about when particular bugs will be fixed or what features will beback-ported from the experimental branch.

The other way to do this is to set a desired feature list and deliver only when it is done. This is essentially the strategy of the "experimental" kernel branch. De Marco and Lister cited research showing that this scheduling policy ("wake me up when it's done") produces not only the highest quality but, on average, shorter delivery times than either "realistic" or "aggressive" scheduling.

I have come to suspect (as of early 2000) that in earlier versions of this essay I severely underestimated the importance of the "wake me up when it's done" anti-deadline policy to the open-source community's productivity and quality. General experience with the rushed GNOME 1.0 release in 1999 suggests that pressure for a premature release can neutralize many of the quality benefits open source normally confers.

It may well turn out to be that the process transparency of open source is one of three co-equal drivers of its quality, along with "wake me up when it's done" scheduling and developer self-selection.

[SU] It's tempting, and not entirely inaccurate, to see the core-plus-halo organization characteristic of open-source projects as an Internet-enabled spin on Brooks's own recommendation for solving the N-squared complexity problem, the "surgical-team" organization—but the differences are significant. The constellation of specialist roles such as "code librarian" that Brooks envisioned around the team leader doesn't really exist; those roles are executed instead by generalists aided by toolsets quite a bit more powerful than those of Brooks's day. Also, the open-source culture leans heavily on strong Unix traditions of modularity, APIs, and information hiding—none of which were elements of Brooks's prescription.

[RJ] The respondent who pointed out to me the effect of widely varying trace path lengths on the difficulty of characterizing a bug speculated that trace-path difficulty for multiple symptoms of the same bug varies "exponentially" (which I take to mean on a Gaussian or Poisson distribution, and agree seems very plausible). If it is experimentally possible to get a handle on the shape of this distribution, that would be extremely valuable data. Large departures from a flat equal-probability distribution of trace difficulty would suggest that even solo

developers should emulate the bazaar strategy by bounding the time they spend on tracing a given symptom before they switch to another. Persistence may not always be a virtue...

[IN] An issue related to whether one can start projects from zero in the bazaar style is whether the bazaar style is capable of supporting truly innovative work. Some claim that, lacking strong leadership, the bazaar can only handle the cloning and improvement of ideas already present at the engineering state of the art, but is unable to push the state of the art. This argument was perhaps most infamously made by the Halloween Documents, two embarrassing internal Microsoft memoranda written about the open-source phenomenon. The authors compared Linux's development of a Unix-like operating system to ``chasing taillights'', and opined ``(once a project has achieved "parity" with the state-of-the-art), the level of management necessary to push towards new frontiers becomes massive."

There are serious errors of fact implied in this argument. One is exposed when the Halloween authors themseselves later observe that ``often [...] new research ideas are first implemented and available on Linux before they are available / incorporated into other platforms."

If we read ``open source'' for ``Linux'', we see that this is far from a new phenomenon. Historically, the open-source community did not invent Emacs or the World Wide Web or the Internet itself by chasing taillights or being massively managed—and in the present, there is so much innovative work going on in open source that one is spoiled for choice. The GNOME project (to pick one of many) is pushing the state of the art in GUIs and object technology hard enough to have attracted considerable notice in the computer trade press well outside the Linux community. Other examples are legion, as a visit to Freshmeat on any given day will quickly prove.

But there is a more fundamental error in the implicit assumption that the cathedral model (or the bazaar model, or any other kind of management structure) can somehow make innovation happen reliably. This is nonsense. Gangs don't have breakthrough insights—even volunteer groups of bazaar anarchists are usually incapable of genuine originality, let alone corporate committees of people with a survival stake in some status quo ante. Insight comes from individuals. The most their surrounding social machinery can ever hope to do is to be responsive to breakthrough insights—to nourish and reward and rigorously test them instead of squashing them.

Some will characterize this as a romantic view, a reversion to outmoded lone-inventor stereotypes. Not so; I am not asserting that groups are incapable of developing breakthrough insights once they have been hatched; indeed, we learn from the peer-review process that such development groups are essential to producing a high-quality result. Rather I am pointing out that every such group development starts from—is necessarily sparked by—one good idea in one person's head. Cathedrals and bazaars and other social structures can catch that lightning and refine it, but they cannot make it on demand.

Therefore the root problem of innovation (in software, or anywhere else) is indeed how not to squash it—but, even more fundamentally, it is how to grow lots of people who can have insights in the first place.

To suppose that cathedral-style development could manage this trick but the low entry barriers and process fluidity of the bazaar cannot would be absurd. If what it takes is one person with one good idea, then a social milieu in which one person can rapidly attract the cooperation of hundreds or thousands of others

with that good idea is going inevitably to out-innovate any in which the person has to do a political sales job to a hierarchy before he can work on his idea without risk of getting fired.

And, indeed, if we look at the history of software innovation by organizations using the cathedral model, we quickly find it is rather rare. Large corporations rely on university research for new ideas (thus the Halloween Documents authors' unease about Linux's facility at coopting that research more rapidly). Or they buy out small companies built around some innovator's brain. In neither case is the innovation native to the cathedral culture; indeed, many innovations so imported end up being quietly suffocated under the "massive level of management" the Halloween Documents' authors so extol.

That, however, is a negative point. The reader would be better served by a positive one. I suggest, as an experiment, the following:

- Pick a criterion for originality that you believe you can apply consistently. If your definition is ``I know it when I see it'', that's not a problem for purposes of this test.
- Pick any closed-source operating system competing with Linux, and a best source for accounts of current development work on it.
- Watch that source and Freshmeat for one month. Every day, count the number of release announcements on Freshmeat that you consider `original' work. Apply the same definition of `original' to announcements for that other OS and count them.
- Thirty days later, total up both figures.

The day I wrote this, Freshmeat carried twenty-two release announcements, of which three appear they might push state of the art in some respect, This was a slow day for Freshmeat, but I will be astonished if any reader reports as many as three likely innovations a month in any closed-source channel.

[EGCS] We now have history on a project that, in several ways, may provide a more indicative test of the bazaar premise than fetchmail; EGCS, the Experimental GNU Compiler System.

This project was announced in mid-August of 1997 as a conscious attempt to apply the ideas in the early public versions of The Cathedral and the Bazaar. The project founders felt that the development of GCC, the Gnu C Compiler, had been stagnating. For about twenty months afterwards, GCC and EGCS continued as parallel products—both drawing from the same Internet developer population, both starting from the same GCC source base, both using pretty much the same Unix toolsets and development environment. The projects differed only in that EGCS consciously tried to apply the bazaar tactics I have previously described, while GCC retained a more cathedral-like organization with a closed developer group and infrequent releases.

This was about as close to a controlled experiment as one could ask for, and the results were dramatic. Within months, the EGCS versions had pulled substantially ahead in features; better optimization, better support for FORTRAN and C++. Many people found the EGCS development snapshots to be more reliable than the most recent stable version of GCC, and major Linux distributions began to switch to EGCS.

In April of 1999, the Free Software Foundation (the official sponsors of GCC) dissolved the original GCC development group and officially handed control of the project to the the EGCS steering team.

[SP] Of course, Kropotkin's critique and Linus's Law raise some wider issues about the cybernetics of social organizations. Another folk theorem of software engineering suggests one of them; Conway's Law—commonly stated as ``If you have four groups working on a compiler, you'll get a 4-pass compiler''. The original statement was more general: ``Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.'' We might put it more succinctly as ``The means determine the ends'', or even ``Process becomes product''.

It is accordingly worth noting that in the open-source community organizational form and function match on many levels. The network is everything and everywhere: not just the Internet, but the people doing the work form a distributed, loosely coupled, peer-to-peer network that provides multiple redundancy and degrades very gracefully. In both networks, each node is important only to the extent that other nodes want to cooperate with it.

The peer-to-peer part is essential to the community's astonishing productivity. The point Kropotkin was trying to make about power relationships is developed further by the `SNAFU Principle': ``True communication is possible only between equals, because inferiors are more consistently rewarded for telling their superiors pleasant lies than for telling the truth.'' Creative teamwork utterly depends on true communication and is thus very seriously hindered by the presence of power relationships. The open-source community, effectively free of such power relationships, is teaching us by contrast how dreadfully much they cost in bugs, in lowered productivity, and in lost opportunities.

Further, the SNAFU principle predicts in authoritarian organizations a progressive disconnect between decision-makers and reality, as more and more of the input to those who decide tends to become pleasant lies. The way this plays out in conventional software development is easy to see; there are strong incentives for the inferiors to hide, ignore, and minimize problems. When this process becomes product, software is a disaster.

Bibliography

I quoted several bits from Frederick P. Brooks's classic The Mythical Man-Month because, in many respects, his insights have yet to be improved upon. I heartily recommend the 25th Anniversary edition from Addison-Wesley (ISBN 0-201-83595-9), which adds his 1986 ``No Silver Bullet'' paper.

The new edition is wrapped up by an invaluable 20-years-later retrospective in which Brooks forthrightly admits to the few judgements in the original text which have not stood the test of time. I first read the retrospective after the first public version of this essay was substantially complete, and was surprised to discover that Brooks attributed bazaar-like practices to Microsoft! (In fact, however, this attribution turned out to be mistaken. In 1998 we learned from the Halloween Documents that Microsoft's internal developer community is heavily balkanized, with the kind of general source access needed to support a bazaar not even truly possible.)

Gerald M. Weinberg's The Psychology Of Computer Programming (New York, Van Nostrand Reinhold 1971) introduced the rather unfortunately-labeled concept of ``egoless programming''. While he was nowhere near the first person to

realize the futility of the ``principle of command'', he was probably the first to recognize and argue the point in particular connection with software development.

Richard P. Gabriel, contemplating the Unix culture of the pre-Linux era, reluctantly argued for the superiority of a primitive bazaar-like model in his 1989 paper ``LISP: Good News, Bad News, and How To Win Big''. Though dated in some respects, this essay is still rightly celebrated among LISP fans (including me). A correspondent reminded me that the section titled ``Worse Is Better'' reads almost as an anticipation of Linux. The paper is accessible on the World Wide Web at http://www.naggum.no/worse-is-better.html.

De Marco and Lister's Peopleware: Productive Projects and Teams (New York; Dorset House, 1987; ISBN 0-932633-05-6) is an underappreciated gem which I was delighted to see Fred Brooks cite in his retrospective. While little of what the authors have to say is directly applicable to the Linux or open-source communities, the authors' insight into the conditions necessary for creative work is acute and worthwhile for anyone attempting to import some of the bazaar model's virtues into a commercial context.

Finally, I must admit that I very nearly called this essay ``The Cathedral and the Agora'', the latter term being the Greek for an open market or public meeting place. The seminal ``agoric systems'' papers by Mark Miller and Eric Drexler, by describing the emergent properties of market-like computational ecologies, helped prepare me to think clearly about analogous phenomena in the open-source culture when Linux rubbed my nose in them five years later. These papers are available on the Web at http://www.agorics.com/agorpapers.html.

Acknowledgements

This essay was improved by conversations with a large number of people who helped debug it. Particular thanks to Jeff Dutky , who suggested the ``debugging is parallelizable'' formulation, and helped develop the analysis that proceeds from it. Also to Nancy Lebovitz for her suggestion that I emulate Weinberg by quoting Kropotkin. Perceptive criticisms also came from Joan Eslinger and Marty Franz of the General Technics list. Glen Vandenburg pointeed out the importance of self-selection in contributor populations and suggested the fruitful idea that much development rectifies `bugs of omission'; Daniel Upper suggested the natural analogies for this. I'm grateful to the members of PLUG, the Philadelphia Linux User's group, for providing the first test audience for the first public version of this essay. Paula Matuszek enlightened me about the practice of software management. Phil Hudson reminded me that the social organization of the hacker culture mirrors the organization of its software, and vice-versa. John Buck pointed out that MATLAB makes an instructive parallel to Emacs. Russell Johnston brought me to consciousness about some of the mechanisms discussed in ``How Many Eyeballs Tame Complexity.'' Finally, Linus Torvalds's comments were helpful and his early endorsement very encouraging.