HZ BOOKS
华章科技

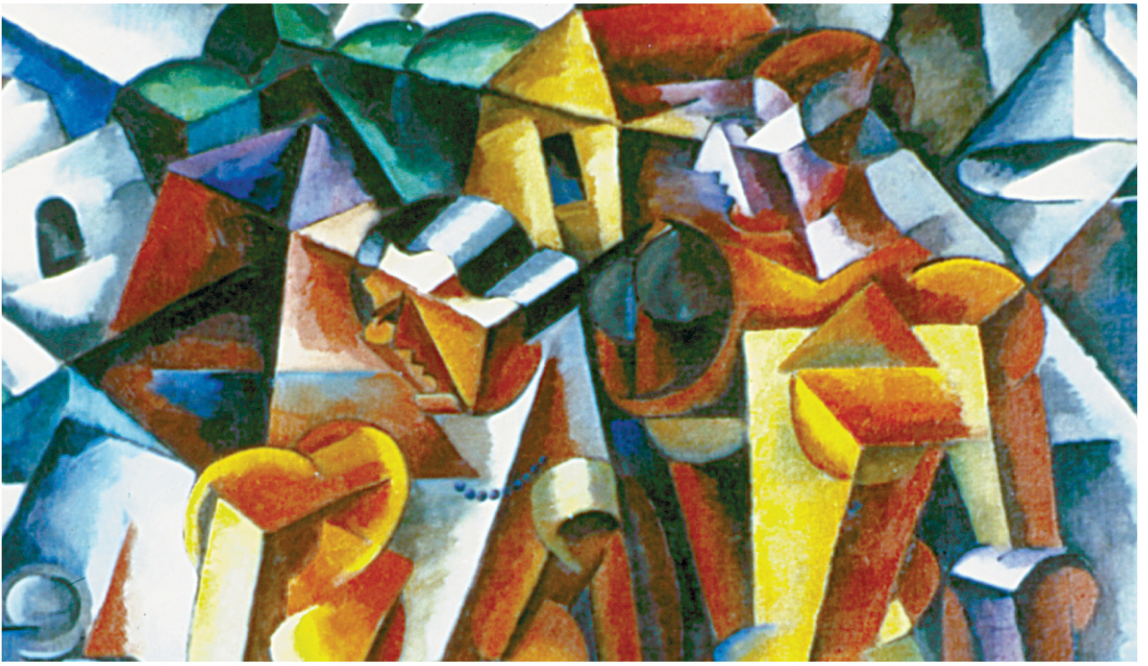
“这是当代软件技术领域最重要的著作，
其影响力远远超出编程范畴。”
——Guy Kawasaki
Garage(车库)技术风险投资公司创始人、董事局主席

最新版

大教堂与集市

THE CATHEDRAL & THE BAZAAR

MUSINGS ON LINUX AND OPEN SOURCE BY AN ACCIDENTAL REVOLUTIONARY



ERIC S. RAYMOND 著
卫剑钒 译

O'REILLY®
机械工业出版社
China Machine Press

BOB YOUNG RED HAT公司董事长兼CEO作序推荐

大教堂与集市

THE CATHEDRAL AND THE BAZAAR

HZ BOOKS
华章图书

大教堂与集市

THE CATHEDRAL AND THE BAZAAR

（美）Eric S. Raymond 著

卫剑钒 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权机械工业出版社出版

机械工业出版社

图书在版编目（CIP）数据

大教堂与集市/（美）雷蒙德（Raymond, E. S.)著；卫剑钒译. —北京：机械工业出版社，2014.1

（O'Reilly精品图书系列）

书名原文：The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary

ISBN 978-7-111-45247-8

I. 大… II. ①雷… ②卫… III. 软件产业－产业发展－研究 IV. F407.67

中国版本图书馆CIP数据核字（2013）第309451号

北京市版权局著作权合同登记

图字：01-2011-7140号

书　　名/　大教堂与集市

书　　号/　ISBN 978-7-111-45247-8

责任编辑/　吴怡

封面设计/　Edie Freedman，张健

出版发行/　机械工业出版社

地　　址/　北京市西城区百万庄大街22号（邮政编码100037）

印　　刷/

开　　本/　147毫米×210毫米　32开本　6.625印张

版　　次/　2014年5月第1版　2014年5月第1次印刷

定　　价/　59.00元（册）

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991; 88361066

购书热线：(010) 68326294; 88379649; 68995259

投稿热线：(010) 88379604

读者信箱：hzjsj@hzbook.com

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即瞬的机会，尽管大路也不错。”

——Linux Journal

HZ BOOKS
华章图书

译者序

“如果你有正确的态度，有趣的事情自然会找到你”，这是Eric在“大教堂与集市”一文中给出的一条经验。很有意思，2012年3月，机械工业出版社的吴怡编辑通过互联网，在我的博客《“大教堂与集市”读后感》下留了一条评论，问我是否愿意翻译这本书。没有太多犹豫，出于对开源世界的敬仰和（本书所提及的）egoboo的驱动，我决定花费时间和精力翻译这本书。

Eric S. Raymond在开源运动和黑客文化中有着很高的地位，正如他自己所言，他有着开源文化“代言人”、“宣传家”和“大使”等称谓，一个更正式的头衔则是：互联网黑客的观察者/参与者人类学家（observer/participant anthropologist，anthropologist是指“someone who scientifically studies human beings, their customs, beliefs and relationships”）。作为黑客文化的第一理论家，他通过观察和参与，对黑客这个群体的习俗、信念及关系有着深入而广泛的研究。他极力为“黑客”（hacker）一词正名，强调“黑客”并不是媒体报道中的计算机违法分子，而是那种着迷于计算机技术并通过编程提供极具价值软件的人。

软件是人类历史上最奇特的产物之一，它和其他事物截然不同，以至于拿任何事物来比喻软件，都给人以不够贴切的感觉，软件的自动化运行能力和几乎零成本复制能力给人们带来了前所未有的便利，而它带来的问题（比如版权问题、消费者权益问题、是否开源问题、开发管理模式问题等等）也是多少年来让各界人士争论不休的。

作为软件的核心内容，源代码无疑是软件“王冠上的宝石”，因为软件的核心在于设计，而所有设计都会体现在源码之中，拿到源码，你就几乎拿到了软件的一切。出于对商业利益和市场竞争的考虑，软件制造商本能地希望把源码保护起来，而黑客出于分享、贡献和不重复劳动的考虑，下意识认为开放源码是更道德的事情，他们考虑的是如何更好更快地做事，考虑的是如何创建一个自由自在、为所欲为的软件世界，而不是如何把源码藏起来牟利。

软件设计是一件需要创造力的事情，天才式的软件必然来自天才式的设计，很多优秀软件的最初版本都是由顶尖黑客独自设计和编码的（正如Ken Thompson之于UNIX，Linus Torvalds之于Linux），但软件膨胀到一定程度，由一个人或几个人继续开发维护就不太现实了，对大型软件来说，多人合作似乎是一种必然，但到底多少人合适，如何分工和组织，如何调动程序员的积极性，如何让软件不会因规模和复杂性而失控，从来都有着不同的方法和认识。Eric在本书中向大家展示了两种最为经典且截然不同的模式：大教堂模式和集市模式。传统大型软件公司的开发模式就像是艰难而缓慢的大教堂建造工程，它有着严密的管理和封闭的集中式结构，但在创新上、生产力上和Bug控制上却落后于集市模式。集市模式是一种并行的、对等的扁平化开发结构，其参与者大多来自于互联网上的志愿者，结构松散，来去自由，就像是一个乱糟糟的集市，但就是这样的组织形式，却取得了像Linux这样令人惊叹的成功。

开源会走向怎样的未来？我们可以看到，互联网和移动智能终端已

经日益影响着每个人的日常生活，而你每天访问的互联网网站，绝大部分基于开源的操作系统、Web服务器和数据库，你所使用的智能手机多采用Andriod或iOS系统，Andriod源于Linux，iOS源于开源的Darwin（Darwin则基于开源的Mach和FreeBSD开发），可以说，只要你上网或使用智能手机，你就在不知不觉中使用了开源软件。开源对软件业和互联网带来了巨大影响，并正在和将会对人们的工作和生活产生更显著的影响，正如Eric在前言中所说的，对于任何一个对计算机有所依赖的人，对于任何一个要在未来工作和生活的人，了解一些开源文化，都是很有意义的。

本书的翻译进度超出预期，一方面是我工作本身比较繁忙（正如Eric所言，IT这个行当有什么时候是不忙的），只能在周末不加班时抽出点时间翻译。另一方面，Eric写文章喜欢用很长的句子，并夹杂很多典故和背景知识，如何准确而通畅地表达原文，并不是一件轻松的事。

翻译本着忠实、通达和易懂的原则开展，我尽量忠于原文并使用平实和简洁的语言翻译，如果同一概念对应多个中文词汇，我会选择最常用和最不容易误解的词汇。比如“noospher”有“智域”、“心智层”、“智力圈”和“人类圈”等等译法，考虑到可以和“大气层”相类比而且最容易望文生义，我采用了“心智层”一词；本文出现最多的词汇“hacker”，则当仁不让地要翻译为“黑客”，这里的“黑客”指的是其本意，即那些着迷于技术并充满才华和理想的人（涉嫌计算机犯罪的cracker则译为“骇客”）；“hackerdom”一词有人译为“黑客道”，但“道”的含义很广泛，容易让人摸不清其内涵，本文将其译作“黑客圈”，因为dom后缀是指具有共同特征或社会地位及角色的群体；“peer”是指在地位上同等的人，如同级别、同类别或同年龄段的人，“同辈”、“同行”、“同类”、“同群”等词汇都不够精确，本文最终选用了虽然不常见但较为精确的“同侪”一词，但将peer-review翻译为“同行评审”，因为软件工程学中已大量使用这一译法。

这本书除了讲述开源及黑客文化外，还给出了很多充满智慧的观点和非常有趣的概念，如命令体系、礼物文化、以少成多、内部市场、竞次、委员会设计、模因、SNAFU现象、软件是服务行业、组织结构决定产品结构、准入门槛越低稳定性越高、程序员是资产而非成本，等等，相信这些内容会给读者带来一些有益的启示和思考。

值得说明的而是，书中很多URL已经失效而无法打开，以后能打开的概率也不大，之所以仍然保留，一方面是为了忠于原著，另一方面，也许你可以从URL中获得一些有用的信息。

最后，感谢我的家人尤其是我妻子于林月的支持与付出。感谢吴怡编辑的耐心和鼓励。感谢译言网上“翻译即反逆”、“nc”、“异议”等人的无私帮助。感谢开源。

谨以此书纪念Robert Anson Heinlein

他教导我：要尊重能力，要珍视和捍卫自由，

特别是：昆虫才讲究技能专一。

✦ ✦ ✦

HZ BOOKS
华章图书

目 录

HZ BOOKS
华章图书

序

✦ ✦ ✦

自由不是一个抽象的商业概念。

任何行业的成功几乎都直接和这个行业供应商及客户所享有的自由度相关，对比美国电话业在AT&T失去垄断地位前后的创新步伐，就能知道用户享有选择的自由是多么重要。

计算机硬件行业和软件行业的对比，是体现自由给行业带来益处的最好示例。在计算机硬件行业，供应商和消费者在全球范围内都享有很高的自由度，所以该行业在产品和客户价值方面的创新速度，是人类前所未见的。而在软件行业，其变化则几乎以十年为单位，办公套件是20世纪80年代的杀手应用，其地位直到90年代才受到浏览器和Web服务器的挑战。

开源软件给软件行业带来的自由，可能要比硬件行业制造商和客户所能享受的自由更广阔。

计算机语言之所以被称为语言，是因为它们确实是语言。掌握编程语言技能的社会成员（如程序员）可以使用该语言构建和交流思想，从而做一些有利于其他社会成员（包括其他程序员）的事。在现代社会，人们越来越

依赖于软件服务，而对这些软件内在知识的法定获取限制（如软件行业长期以来使用的专有软件[⊖]许可制度），导致自由更少、创新更慢。

在软件行业自认为所有基础架构都已经定型了的时候，开放源码这种革命性的理念横空出世。和以往限制用户获得源代码并以此控制用户的做法不同，开源使用户能够控制他们所使用的技术。将开源工具推向市场需要新的商业模式，相比那些仍然试图控制消费者的公司，开源能给用户带来极为独特的好处，能开发出开源商业模式的公司，将取得非同寻常的成功。

要想对世界做出实质性的改变，开源需要做到这两点：一是要让人们广泛使用开源软件；二是要让用户知道并理解这种软件开发模式能给他们带来的益处。

Eric Raymond把这种革命性软件开发模式的好处解释得如此清晰、透彻、准确，对开源软件革命的成功、Linux操作系统的广泛采用、开源软件供应商及开源用户的成功，都起到了至关重要的作用。

——Bob Young, Red Hat公司董事长及CEO

⊖ 专有软件（Proprietary software），又称专属软件、私有软件或封闭性软件等，指通过法律或者技术上的手段，允许用户在一定条件下使用软件，但限制用户对软件进行修改、再发布或逆向工程等，最常见的技术限制方式是封存人们可以读懂的源代码，只发布计算机才能读懂的程序（如二进制格式）。——译者注

前 言
——为什么你应该关心这些

✦ ✦ ✦

你手头这本书是关于黑客[⊖]行为和文化的，本书收集了一系列文章，最早是写给程序员和技术管理人员的，你可能很自然（并且再正常不过）要问：“我为什么要关心这些？”

最显而易见的答案是：计算机软件在世界经济和商业战略决策中扮演着越来越重要的角色。不管你是由于什么原因打开这本书，大概你对那些关于信息经济、数据时代、互联世界的说法都听得太多了，我不想在这里复述那些。我只是想说，如果我们能对如何编写更优质、更稳定的软件有更深入一点的认识，都将会产生滚雪球般的深远影响。

本书并没有给出基础理论上的创新，只是描述了这样的事实：开源软件系统性地利用开放式开发和分布式同行评审（peer review），不仅降低了成

⊖ 黑客（hacker）一词，原指喜欢通过智力和创造性方法挑战难题的人,尤指那些热衷于计算机技术的编程高手。由于媒体报道中出现的黑客事件往往和计算机犯罪相连，导致人们常常误认为黑客是利用网络入侵他人系统的破坏者。事实上，这些破坏者应该被称为cracker，即“骇客”。——译者注

本，还提高了软件质量。开源软件并不是一个新概念（这种文化可以追溯到30年前互联网刚诞生的时候），但直到最近，才在技术和市场的合力下，从小众的圈子中走了出来。今天，开源运动正积极投身于定义21世纪的计算基础设施，任何一个对计算机有所依赖的人，都很有必要去理解它。

我之所以提到“开源运动”，是因为它有着其他更有意思的原因值得读者关注。30多年来，互联网原生的一群充满活力的倡议者团体，一直在追求、实现并珍爱着开源思想。这些人以自称“黑客”为荣，这里所说的“黑客”并非是记者们滥用的电脑犯罪分子的代称，而是指对某种事物的狂热爱好者、艺术家、古怪的天才发明家、问题解决高手和技术专家。

数十年来，默默无闻的黑客团体不仅要奋力解决技术难题，还要忍受来自于社会主流的冷漠和排斥，直到最近他们才迎来了自己的春天。他们创建了互联网、创建了UNIX、创建了WWW，他们还正在创建Linux和开源软件，经历上世纪90年代中期互联网爆炸式的发展后，人们才终于明白这个世界原本早该好好对待黑客们。

黑客文化及其所取得的成功，对于研究人类动机、工作组织方式、专业主义的未来、公司形态等一些基础性问题，以及这些内容在21世纪信息充裕的后稀缺经济时代中如何变化和演进，都提供了一个很好的研究范例。此外，黑客文化还颇具说服力地预示了人类在适应和重塑经济环境方面将会发生的一些深刻变革，因此，对任何一个要在未来工作和生活的人，懂一些黑客文化都是很有意义的。

这本书是我早前发布在互联网上的一些文章的合集，“黑客圈简史”最早写于1992年，此后多次被更新和修订，其他文章写于1997年2月至1999年5月间，在1999年10月做了一些修改和补充，并在2001年1月本书第二版修订时又做了一次更新，但并没有删掉其中比较技术化的部分或者让文章变得更通俗易懂一些。在我看来，能让读者产生一些疑问和思索，比起让读者感觉厌烦或者被低估智商更有礼貌。如果你在文中遇到一些特定的技术话题、历史典故或者偏僻的计算机术语缩写，尽管跳过去好了，整本书是在讲一个故事，读到后面的时候，也许你就理解了前面没弄明白的地方。

读者还应了解，这些文章是不断更新的，我周期性地把读者的评论及纠错整理出来并融入其中，当然，我本人对书中的任何错误负责。这本书受益

于同行评审（类似于对软件代码的评审）过程，采纳了不胜枚举的意见和建议。这里印刷的并不是最终版本，而更像是一个持续研讨的报告，文中所述黑客文化的很多成员，都积极参与了这些研讨。

最后，我必须要表达一下对很多人以及对一连串机缘巧合的欣喜、惊讶和感谢，也正是这些才导致本书的诞生。

特别感谢那些长期以来的友谊和对本书创作的支持，感谢Linus Torvalds、Larry Augustin、Doc Searls、Tim O'Reilly，很骄傲和你们既是朋友又是同事。最要感谢的是Catherine Raymond，我的挚爱、妻子和最长久的支持者。

我是一名黑客，20年来，作为本书所描述的黑客文化的一分子，我有幸和这个世界上一些最有趣、最杰出的人一起共事，一起解决那些让人着迷的问题，并有幸获得几次珍贵的机会，去做出一些真正创新和有用的东西。有太多的人教给我有价值的东西，教给我黑客技术和其他东西，我无法一一列举他们，在这里谨以本书中的文章作为回馈。

这些文章记录了我在不同阶段的发现和体会，在这个迷人的发现之旅中，我学会了以更新、更深刻的视角来看待我长期以来熟悉的工作。我一直惊讶的是，这样一个简简单单的随笔，居然在开源软件融入主流世界的过程中起到了持续催化的效果。希望读者能够在这些随笔中捕捉那些令人兴奋的精彩片段，在主流商业及客户迈入这一旅途之际，一同感受那些展现在我们面前的令人赞叹不已的美妙前景。

第二版修订注记

受益于本书第一版的读者，第二版做了一些实质性的补充和修改，大体如下：

多少双眼睛才能驯服复杂性、要命的最后期限，关于分支和伪分支更准确的定义，进化不利条件理论、孔雀、牡鹿和开源开发者动机的关系，开源的经济学动力，信息不对称效应，用开源做竞争武器。在“黑客的反击”一文中的一些预言已经在一年后被验证了，这次又增加了一些新的。在附录中新增了fetchmail项目的成长记录。

HZ BOOKS
华章图书

1. 黑客圈[⊖]简史

✦ ✦ ✦

本文探索黑客文化的起源，探索那些“真程序员”的史前故事，探索MIT黑客们的光辉岁月，以及早期的ARPAnet是如何孕育了第一个网络部落。本文描绘UNIX早期的辉煌以及最后的停滞，描绘来自芬兰的希望，描绘“最后一个真正的黑客”如何成为新一代开源程序员的领袖，以及Linux和互联网主流如何把黑客文化从公共意识的边缘引领到如今的显赫地位。

⊖ 黑客圈（hackerdom，或译为“黑客界”）是指整个黑客团体（dom后缀在英文中是指具有共同特征或社会地位及角色的群体）及其文化，有人曾将其译为“黑客道”。——译者注

HZ BOOKS
华章图书

1.1 序：真程序员

最初，有一种人叫“真程序员”（Real Programmer）。

他们并不这样自称，也不自称为黑客或者别的什么。据他们中的一位回忆，“真程序员”这个称呼是在上世纪80年代以后才出现的。自1945年以来，计算机技术吸引了世界上最睿智和最有创意的人，从Eckert和Mauchly发明的第一台ENIAC开始，就有一批编程爱好者，并或多或少伴随着一种他们自己能意识到的技术文化，他们编软件和玩软件只是出于乐趣。

“真程序员”通常具备工程学和物理学背景，并常常是业余无线电爱好者。他们穿着白色袜子、涤纶衬衫，打着领带，带着厚厚的眼镜，使用机器语言、汇编语言、FORTRAN或者其他一些已经被人们遗忘了的古老的编程语言。

从二战结束到上世纪70年代初，是批处理操作和俗称“大机”（big iron）的大型机（mainframe）盛行的年代，“真程序员”主宰了计算机世界的技术文化。从这个年代起，一些令人敬慕的黑客文化开始流传，包括种种版本的墨菲定律以及仿德语风格的“Blinkenlights”提示语，后者至今还被张贴在很多机房中。

一些在“真程序员”文化下成长起来的黑客，直到上世纪90年代都仍然很活跃。Cray系列超级计算机的设计者Seymour Cray就是其中的佼佼者，据说他通过机器的前面板开关将他自己设计的整个操作系统以八进制形式导入他设计的计算机之中，并且没有任何错误，这可真是大师级的神作。

“真程序员”文化和批处理计算（尤其是批处理技术）密切相关，随着交互式计算、大学和网络的兴起，“真程序员”文化逐渐衰落，另一个工程师文化诞生并最终演化成今天的开源黑客文化。

1.2 早期的黑客

我们今天所了解的黑客文化，其起源时间大致可定位于1961年，那年，MIT（麻省理工学院）有了第一台PDP-1。MIT技术模型铁路俱乐部（Tech Model Railroad Club）的信号和动力委员会（Signals and Power Committee）把这台机器当作他们最喜欢的科技玩具，并由此发明了一系列的编程工具、俚语以及直到今天仍然依稀可辨的文化氛围。这些早年轶事可以在Steven Levy写的《黑客》[⊖]（Hackers）一书中觅得踪迹（Anchor/Doubleday 1984, ISBN 0-385-19195-2）。

“黑客”一词大约就起源于MIT的计算机文化。技术模型铁路俱乐部的黑客们，大都成为MIT人工智能（AI）实验室的核心成员，直到上世纪80年代早期，该实验室在AI领域的研究都一直处于领先地位， 1969年后，他们的影响逐渐扩展开来，因为在那年，APRAnet诞生了。

APRAnet是第一个横贯美国大陆的高速计算机网络，它从一个由国防部出资兴建的实验性数字通信系统，逐渐成长为一个连接大学、国防部承包商及研究实验室等数百个节点的大网，使得位于各地的研究者能够以前所未有的速度和灵活性交换信息，这极大地促进了合作交流，推动了科学技术的突飞猛进。

ARPAnet的好处远不止这些，这些电子高速公路把散落全美各地的黑客聚集到一起，构成了产生黑客文化的关键力量。这些黑客原先只是在各自隔离的小团体内发展他们短暂的局部文化，现在，他们发现自己已经俨然是（或者说他们已经把自己打造成）一个网络部落了。

黑客文化的第一批产物——第一个俚语列表、第一篇讽刺作品、第一次有意识地对黑客道德的讨论——开始在ARPAnet上传播开来，尤其是1973年到1975年间通过网络合作完成的第一版“黑客行话”（Jargon File，*http://www.tuxedo.org/jargon*）。这本俚语字典成为黑客文化的一个定义性文档，并最终在1983年出版为《黑客字典》（*The Hacker's Dictionary*）。现在第一版已经停印了，其修订版和增补版是《新黑客字典》（*The New*

⊖ 《黑客》中文版已由机械工业出版社引进出版，书号是：978-7-111-35840-4。——编辑注

Hacker's Dictionary），由MIT出版社于1996年出版（3rd edition，ISBN 0-262-68092-0）。

黑客圈在那些联网的大学中——特别是（虽然不全都是）在计算机科学系中——开始发展壮大，上世纪60年代后期，MIT的AI实验室和LCS实验室首当其冲，斯坦福大学的人工智能实验室（SAIL）和卡内基－梅隆大学（CMU）紧随其后，作为当时最繁荣的计算机科学和AI研究中心，这些实验室吸引了大量的优秀人才，不论在技术上还是文化上，这些人都为黑客文化做出了伟大的贡献。

为了更好理解后面发生的事情，我们需要了解一下计算机自身的发展，因为AI实验室的兴盛和衰落，都是由计算机技术的发展变革而导致的。

从PDP-1时代开始，黑客文化的命运就和DEC（数字设备公司）的PDP小型机系统交织在一起了，DEC率先推出了交互式商业计算和分时操作系统，由于其机器灵活、强大且相对便宜，很多大学都购买了DEC的小型机。

廉价的分时系统成为黑客文化成长的媒介，在整个ARPAnet的生命周期中，大多数时间都是DEC小型机的天下，其中最重要的是PDP-10，这款发布于1967年的机器，几乎是这之后15年内黑客圈的最爱，TOPS-10（PDP-10的操作系统）和MACRO-10（其汇编语言）至今仍被黑客在一些俚语和传说中充满怀旧地提及。

同样也是使用PDP-10，MIT却有些与众不同，他们完全摒弃了DEC为PDP-10写的软件，而是自己写了一个操作系统，即传说中大名鼎鼎的ITS。

ITS即“不兼容分时系统”（Incompatible Time-sharing System），这充分体现了MIT黑客们的态度，他们就是要走“自己”的路。好在MIT人的智慧配得上这种自负，虽然ITS古灵精怪，时不时出点bug，但却充满了才华横溢的技术创新，并似乎仍然保持着单个分时系统的最长连续使用时间记录。

ITS是用汇编语言写的，其应用大都是用AI语言LISP写的。LISP比当时的任何编程语言都要强大而灵活，事实上，25年过去了，它的设计仍然比如今大多数语言都要好。LISP让使用ITS的黑客能解放出来，以一种与众不同和充满创意的方式思考问题，它是MIT黑客们取得成就的主要因素，并且直到现在仍然是黑客圈最喜欢的编程语言之一。

ITS文化的一些技术产物至今仍然被人们使用，EMACS编辑器可能是其中最广为人知的。ITS的很多传说仍然活在黑客们心中，感兴趣的话，可以看一下“黑客行话”（*http://www.tuxedo.org/jargon*）。

SAIL和CMU当然不会自甘落后。在SAIL的PDP-10周围成长起来的黑客们，后来大多成为个人计算机及窗口/图标/鼠标软件交互界面的关键人物。CMU黑客们的工作则引领了专家系统和工业机器人技术的大规模实际应用。

另一个重要的黑客文化节点是XEROX PARC，即著名的Palo Alto研究中心。从上世纪70年代到80年代中期，PARC产生了大量极具突破性的软硬件发明，其数量之多令人震惊。目前被广泛使用的鼠标、窗口和图标式软件交互界面，以及激光打印机和局域网，都是在那里发明的。比起80年代才出现的个人计算机, PARC的D系列机器足足早了十年。然而可悲的是，这些天才的先知们，并没有在他们自己的公司内获得荣耀，以至于有个经典笑话说PARC就是专为别人开发绝妙创意的地方。毫无疑问，PARC对黑客圈的影响是普遍而深远的。

ARPAnet和PDP-10文化在上世纪整个70年代得到了迅猛而多样的发展，电子邮件列表（mailing list）除了促进一些专题兴趣小组（special-interest group）在全美范围内的合作外，也越来越多地应用于社交和休闲领域。DARPA对这种“非授权”行为故意睁一只眼闭一只眼，因为它知道付出这点额外流量，就能吸引一整代的聪明年轻人到计算机领域中来，那真是太划算了。

最广为人知的社交类ARPAnet邮件列表应该算是科幻迷们的SF-LOVERS列表了，时至今日它仍然相当活跃，当然，现在是在ARPAnet演化而成的互联网上。当时还有其他一些开创性的网上交流方式，后来被一些营利性分时服务商推向商业化，如CompuServe、 GEnie和Prodigy（后者现在仍被AOL掌控）。

作为这段历史的描述者，我就是通过早期的ARPAnet和科幻迷圈子，从1977年开始接触黑客文化的，我有幸见证和参与了本文所描述的黑客文化的诸多变迁。

1.3 UNIX的兴起

上溯至ARPAnet还远未普及的1969年，在新泽西郊外，一股新生力量开始成长并不断发展壮大，最终使PDP-10文化变得不再重要。这一年，APRAnet刚刚诞生，而贝尔实验室的黑客Ken Thompson，也正在这年发明了UNIX。

Thompson参与了分时操作系统Multics的开发工作，Multics和ITS有着共同的渊源，它是一个验证一些重要观念的试验床，这些观念的着重点在于如何将操作系统的复杂性隐藏在系统内部，不仅让用户看不到，甚至让大多数程序员都看不到。它使得人们可以更简单地使用Multics（以及为它编程！），可以更多去做那些真正有价值的工作。

当Multics逐渐显露出成为“白象”[⊖]这种庞大而又无用之物的迹象时，贝尔实验室从这个项目退出了（这个系统后来被Honeywell公司推向市场，但从未获得成功）。出于对Multics环境的怀念，Ken Thompson开始尝试将Multics的一些理念和自己的一些想法融合起来，在一台废置的DEC PDP-7上开发一个新的系统。

贝尔实验室另一名黑客Dennis Ritchie为还处于雏形阶段的UNIX发明了一种新的语言：C语言。和UNIX一样，C被设计为好用、限制少和灵活方便的语言，很快，这些工具在贝尔实验室流行起来了，1971年，Thompson和Ritchie赢得了开发一个内部系统（类似我们现在所说的办公自动化系统）的投标，更大地刺激了UNIX和C的内部传播，而Thompson和Ritchie的雄心远不止于此。

操作系统在传统上都是用汇编语言精心编写的，目的是充分利用机器的效能。Thompson和Ritchie是最早意识到当时硬件和编译技术都已经好到能让整个操作系统用C语言编写的那批人之一。到1978年，整个UNIX环境已经可以成功地被移植到多种不同型号的机器上了。

这是史无前例和影响巨大的。如果UNIX能够在多种不同型号的机器上提供相同的人机界面和相同的功能，它就能成为一个通用的软件环境。机器

⊖ 白象是一种罕有的白色亚洲象，常作为宗教或王室权利的象征，用来形容昂贵、无用且需要很高代价来维持或经营的事物。——译者注

更新换代时，用户就可以不再购买那些为新机器而重新编写的软件，黑客们则可以在不同机器上使用相同的工具，而不是每次都去做类似发明轮子和钻燧取火的事。

除了可移植性，UNIX和C还有其他的重要优势，它们都是KISS（Keep It Simple, Stupid）哲学下的产物。程序员可以很容易地在脑海中记忆并掌握整个C语言的逻辑结构（这可不同于之前或之后的大多数语言），而不需要去频繁地查看手册。而UNIX则拥有一系列灵活方便的工具程序，每个工具都被设计为可与其他工具组合运用，以方便地实现特定目的。

UNIX和C的组合，很快被证明适用于极为广泛的计算作业，其中很多完全超出设计者的预期。虽然缺乏正式的支持和推广，它仍然在AT&T内部迅速传播开来。到1980年，它已经扩散蔓延到很多大学和研究机构，而数以千计的黑客们则开始考虑在家里使用它了。

早期UNIX文化中的主力机器是PDP-11及其后代VAX。但由于UNIX的高可移植性，使得它基本上不用改动就能运行在比整个ARPAnet上范围更广的机器上，没人再用汇编语言了，C语言被迅速移植到了各种机器上。

UNIX甚至有了自己的网络——UUCP：低速、不太可靠但便宜。任意两个UNIX机器可以通过普通电话线路，点对点地交换电子邮件，而且这种功能是系统自带的，不需要额外安装。1980年，第一批Usenet（Usenet最初运行在UUCP上——译者注）站点开始交换广播消息，由此形成了一个巨大的分布式电子公告板，并很快在规模上超过了ARPAnet。围绕Usenet，UNIX站点开始逐渐形成自己的网络部落。

由于一些UNIX站点运行在ARPAnet上，PDP-10文化和UNIX/Usenet文化开始在各自的边缘交汇，但它们并不能和谐相处，PDP-10的黑客们倾向于把UNIX团体看成是一群暴发户，与LISP和ITS具有巴洛克式令人着迷的复杂性相比，UNIX使用的工具看上去原始得可笑，“就像拿着石刀和穿着兽皮！”他们嘟囔道。

除此之外，另外一股势力也开始成长，第一台个人电脑在1975年开始进入市场，苹果公司于1977年成立，技术变革在随后几年以令人难以想象的速度发展，微型计算机的发展势头越来越清晰，并吸引着新一代聪明的年

轻人，他们的语言是BASIC，这种语言是如此简陋，以至于PDP-10信徒和UNIX爱好者都认为这简直不值得去蔑视。

1.4 远古时代的终结

1980年发生了很多事，三种文化在边缘互相交叠，却各自形成截然不同的技术体系，ARPAnet/ PDP-10文化紧紧围绕着LISP、MACRO、TOPS-10、ITS以及SAIL[⊖]的发展；UNIX和C主要使用PDP-11、VAX以及慢得让人心烦的电话连接；而一群没有组织的微机爱好者则下决心让普通大众都享受到计算机的威力。

这其中，ITS文化仍占据至尊地位，但是MIT的实验室里已乌云密布，ITS所寄身的PDP-10已经开始过时，实验室随着人工智能的首次商业化尝试而四分五裂，受一些新成立公司的高薪职位吸引，实验室里最优秀的人才正纷纷出走（SAIL和CMU的实验室也一样）。

1983年，ITS文化迎来了致命一击，DEC取消了PDP-10的后续项目“木星计划”，以集中精力研制PDP-11和VAX系列。ITS没有未来了，因为它没有可移植性，而且也没人能把它搬到新机器上，在VAX上运行的Berkeley版UNIX成为最出类拔萃的黑客系统。同时，任何一个有点远见的人都能看到，微型计算机风头正劲，看上去似乎要横扫一切。

就在这个时候，Steven Levy写就了《黑客》一书，其中一个最重要的受访人是Richard M. Stallman（Emacs的发明人），作为MIT AI实验室的标识性人物，他坚决反对将实验室研究成果商业化。

Richard M. Stallman（人们更熟悉他的名字缩写RMS，这也是他常用的登录名）离开实验室，创建了自由软件基金会（Free Software Foundation），献身于生产高质量的自由软件。Steven Levy称赞他为“最后一个真正的黑客”，幸好没被他说中！

RMS的宏大计划是黑客文化在上世纪80年代遭遇变迁的典型例证——1982年他开始用C语言重新构建整个UNIX的克隆，并免费发布，这就是广为人

⊖ 这里的SAIL指的是SAIL实验室（Stanford Artificial Intelligence Laboratory）推出的SAIL语言（Stanford Artificial Intelligence Language）。——译者注

知的GNU（Gnu's Not UNIX，这是一种递归式的缩写）操作系统，GNU迅速成为黑客活动的焦点，而ITS的精神和传统，作为以UNIX和VAX为主的新一代黑客文化的重要组成部分，籍此得到了保全。

事实上，在其后大约十多年里，RMS的自由软件基金会在很大程度上定义了黑客文化的公共意识形态，Richard M. Stallman本人则毋庸置疑地成为了整个黑客文化部落的唯一精神领袖。

1982到1983年间，集成电路和局域网技术对黑客圈产生了重要影响，以太网和摩托罗拉68000微处理器成为一个很有潜力的组合，一些创业公司纷纷成立，设计开发第一代我们现在所说的工作站。

1982年，一批来自Stanford和Berkeley的UNIX黑客创立了Sun Microsystems公司，他们认为，UNIX操作系统配以相对便宜的基于68000微处理器的硬件，将会被证明是一个可用于多种场合的无敌组合。他们是对的，他们的洞察力为整个产业提供了范例。虽然工作站的价格对大众个体来说还是太贵，但对企业和大学来说已经很便宜了，工作站之间组成的网络（每个用户一台机器），迅速取代了那些过时的VAX机器和其他分时系统。

1.5 “专有UNIX”时代

1984年，当Ma Bell[⊖]被拆分后，UNIX第一次成为AT&T支持的产品，黑客圈形成了两大阵营，一边是围绕Internet和Usenet而形成的相对有凝聚力的“网络部落”（他们中绝大多数使用运行着UNIX的小型机或工作站级别的机器），一边则是没有网络的分散在各个角落的微机爱好者。

这一阶段，一些严重的计算机破坏（cracking）事件开始被主流媒体报道，记者们误用黑客（hacker）一词来形容那些破坏者，这种不幸的误用一直延续至今。

Sun和其他公司生产的工作站级别计算机给黑客们打开了一片新的天地。其设计目的是实现高性能图形运算和共享数据的网络传输。20世纪80年代

⊖ Ma Bell是Bell System公司的口语化称呼，它以AT&T公司为母公司，下属众多子公司和研究所，由于长期垄断美国电信业，1984年因“反托拉斯法”被拆分。——译者注

期间，黑客圈殚精竭虑，开发了一系列可以充分利用工作站特性的软件和工具。Berkeley UNIX提供了对APRAnet协议的内置支持，解决了由于UUCP点到点连接较慢而带来的网络问题，促进了互联网的进一步发展。

在设法充分利用工作站图形能力的若干尝试中，最流行的当属X Window系统，它由MIT开发，吸纳了十多家公司数百名员工的贡献。X Window成功的关键在于其开发者愿意遵守黑客道德免费提供源码，而且是通过互联网发布。X战胜专有图像处理系统（包括Sun公司自己的）成为了这种改变的一个重要标志，在几年后，它深深地影响了整个UNIX。

ITS和UNIX之间仍然会时不时爆发一些派系之争，且大多数情况下都是ITS一方挑起的。但随着1990年最后一台ITS机器的关机停用，狂热分子们也不得不放下立场，伴随着不同程度的抱怨，他们大多融入了UNIX文化。

在20世纪80年代已经连上网络的那些黑客群体间，最大的对立来自于Berkeley UNIX和AT&T UNIX的爱好者。偶尔你还可以发现那个时期的招贴画：卡通化的“星球大战”X翼战机从画着AT&T标识的死星的爆炸中疾驶而出。Berkeley黑客们喜欢把自己比做是反抗军，矛头直指那些没有灵魂的商业帝国，而AT&T UNIX虽然在市场份额上从来没有超过BSD/SUN组合，却赢得了标准之战。1990年，AT&T UNIX和BSD UNIX已经很难区分，因为彼此都吸纳了对方的很多新特性。

随着20世纪90年代的到来，已经有十多年发展的工作站技术，受到了明显的威胁，基于Intel 386系列芯片的廉价且高性能的个人计算机出现了，历史上第一次，黑客个人有能力购买一台家用机器，而且其性能和存储能力可以媲美十年前的小型机！UNIX则有能力提供运行于其上的整个开发环境，并能连上互联网。

MS-DOS世界仍然无知并快乐着，早期的微机爱好者们很快扩张成一支庞大的队伍，DOS和Mac黑客们的数量已经超过了“网络部落”，但他们没有产生一种有自我意识的文化，其间有五十多种技术如蜉蝣般生死交替，但从来没有稳定到可以发展出俚语、传说和轶事这类的公共传统文化。另外，由于一直没有出现类似UUCP或互联网这种真正能流行起来的网络技术，他们也没能发展出自己的网络部落。

CompuServe和GEnie这类商业连线服务此时已经分布很广了，但由于非

UNIX操作系统并不会随系统附送开发工具，因此微机爱好者们很少能从网上获取源代码，更不要说发展出合作开发的风气了。

这一阶段黑客圈的主流，有组织或没组织地围绕在互联网周围，他们中的大多数认同了UNIX的技术文化。他们不怎么关心商业服务，只希望有更好的工具和更便利的网络条件，这一时期的厂商们则承诺便宜的32位个人电脑可以让每个人实现这一切。

但软件呢？动辄数千美元的商业UNIX仍然太贵了。20世纪90年代初，很多公司致力于将AT&T或者BSD UNIX移植到PC级别的机器上，但一直不太成功，价格也没怎么降下来，而且最糟糕的是你拿不到可以修改和重新发布的操作系统源代码，传统商业模式是无法满足黑客这种需求的。

自由软件基金会（FSF）也没有做到，RMS许诺已久的HURD——这个给黑客开发的免费UNIX内核，多年来一直停滞不前，直到1996年才拿出一个可用的内核（尽管在1990年，FSF就解决了UNIX类操作系统除内核外几乎所有的难题）。

更糟糕的是，20世纪90年代初，人们清楚地看到，十多年来对专有UNIX的商业化努力显然已经失败了。UNIX曾经承诺的跨平台可移植性，在多个专有UNIX版本的争吵声中看不到任何希望，这些专有UNIX商家表现得如此沉闷、盲目和没有市场能力，以至于微软凭借Windows操作系统，从他们手中抢走很多市场份额。

1993年初，对黑客文化不怀好意的评论者，也许有足够理由认为UNIX和黑客部落就要玩完了。计算机行业媒体上更是不乏这种人的言论，自20世纪70年代末以来，他们几乎每隔六个月就会旧调重弹，说UNIX马上就要灭亡了。

那些日子里，一种常见的观点看上去好像颇有道理：技术上的个人英雄主义时代已经终结，软件工业和新生的互联网，将会越来越多受微软这类大型企业控制。第一代UNIX黑客看上去已经垂垂老矣（Berkeley计算机科学研究组已经动力尽失，并于1994年失去了资助），多么令人沮丧的时光。

幸运的是，在行业媒体的视野之外，甚至大多数黑客视野之外，一些东西正在慢慢成长，并将在1993年年底到1994年取得令人吃惊的发展，最终，它将引领整个黑客文化进入一个全新的方向，并取得做梦也想象不到的成功。

1.6 早期的自由UNIX

FSF一直未能完成的HURD使得Helsinki大学一名叫Linus Torvalds的学生有了施展才能的空间，1991年，他开始为386机器开发自由UNIX内核，使用的正是FSF提供的软件套件。Linus很快获得了成功并吸引了互联网上的黑客们，他们帮助Linus一同开发Linux：一个全功能的UNIX，源代码完全免费，而且可以再发布。

Linux并不是没有竞争者，1991年，在Linus Torvalds早期尝试的同时，William和Lynne Jolitz正试着把BSD UNIX源代码往386上移植，大多数评论者在比较BSD技术和Linus早期简陋的成果之后，都认为BSD移植将会成为PC上最重要的自由UNIX。

Linux最重要的特点不是技术上的，而是社会学上的。在Linux被开发出来之前，所有人都认为，如果软件复杂到操作系统这样的程度，就必须要有一个精心协作的团队，团队要比较小，而且紧密互动，不管是以前还是现在，这都是很典型的开发模式。商业软件、FSF在20世纪80年代开发的如大教堂般宏伟的自由软件以及从Lynne Jolitz最初的386BSD分裂出来的freeBSD/netBSD/OpenBSD这些项目，都是使用这种模式开发的。

Linux几乎从一开始就发展出一条完全不同的路，其开发更像是仅通过互联网合作的大量志愿者的随意之作。在质量方面，没有严格的标准也没有一个强有力的机构来管理，他们只是执行一个简单得有点幼稚的策略：每周发布，并在接下来几天内获取数百个用户的反馈。他们创造了一种类似达尔文“物竞天择”的选择机制，被选择对象则是开发者们所做的种种软件修改。让所有人吃惊的是，这种方式工作得非常好。

1993年年底，Linux在稳定性和可靠性上已经和很多商业UNIX不相上下，并能支持比商业UNIX要多得多的软件，一些商业应用软件甚至开始考虑移植到Linux上。Linux间接导致多数小型专有UNIX供应商的关张——因为再没有开发者和黑客买他们的东西了。BSDI（Berkeley系统设计公司）作为少数几个幸存者之一，之所以仍然活跃，是因为他们提供整套的BSD UNIX源代码，并和黑客社团仍保持着紧密的关系。

对于当时的这些新情况，黑客内部并没有做过多评论，外界媒体就更没有

什么声音了。黑客文化置那些不断预言自己死亡的言论于不顾，开始以自己的观点重塑商业软件世界，再过五年多，这种趋势会更加明显。

1.7 Web大爆发

和Linux早期发展相互促进的是：公众发现了互联网。20世纪90年代早期，ISP(互联网服务提供商)行业开始逐渐繁荣起来，普通大众一个月花不了多少美元就可以连上互联网，WWW发明以来，互联网本来就很快的增长速度更是加速到了不可思议的地步。

1994年，Berkeley UNIX开发团队正式关闭了，若干种不同版本的自由UNIX（Linux和386BSD的后裔）成为黑客活动的主要焦点。Linux开始被商业公司刻录在CD-ROM上发布，并且非常畅销。1995年年底，主要的计算机公司开始大张旗鼓地宣传他们的软硬件可以很好地连接互联网。

20世纪90年代后期，黑客圈的活动中心是开发Linux和宣扬互联网，WWW使互联网成为大众媒体，许多20世纪80年代和90年代早期的黑客，开始收费或者免费向公众提供ISP服务。

互联网成为主流后，黑客文化开始受到尊敬，并有了一定政治影响力，1994年到1995年间，黑客的大规模强烈抗议，使得试图将“强加密”算法置于美国政府控制之下的Clipper提案无疾而终。1996年，黑客动员起广泛的同盟，导致所谓的“通信合宜法”（CDA）被废止[⊖]，阻止了政府对互联网的审查。

伴随着CDA的胜利，我们从历史迈进了现在，接下来，我将更多以行动者而非观察者的身份出现（我自己都没想到），更多的故事将在“黑客的反击”里一一展现。

⊖ 1996年2月，为限制和阻止网上色情内容对青少年的影响和危害，美国总统克林顿签署了《通信合宜法》（Communications Decency Act，CDA），很快，美国公民自由联盟（ACLU）以该法侵害美国宪法第一修正案赋予公民的言论自由权利为由，对美国政府提出起诉。1997年6月26日，美国最高法院终审裁定CDA违背美国宪法，并宣布即刻废止。—— 译者注

2. 大教堂与集市

✦ ✦ ✦

Linux有一套令人吃惊的软件工程理论，fetchmail作为一个专门对这些理论进行实验的开源项目，所取得的成功让我感到惊讶。我将在本文讨论这些理论，并对比两种完全不同的开发模式：绝大多数商业公司所采用的“大教堂”模式和Linux世界采用的“集市”模式。两种模式的根本不同点在于他们对软件排错有着完全对立的认识。我从Linux的经验出发，证实了这样一个命题：“只要眼睛多，bug容易捉。”这和那些由利己个体组成的自纠错系统有着异曲同工之妙。在本文的最后，我探讨了在这种观念的影响下，软件可能拥有的未来。

HZ BOOKS
华章图书

2.1 集市模式的成功

Linux是颠覆性的，就在5年前（1991年），谁能想到，几千名散布在全球各地的开发者们，利用业余时间，仅仅是通过Internet这种脆弱的合作，就鬼斧神工般地造就了一个世界级的操作系统？

我肯定想不到。在1993年初Linux进入我视野的时候，我已经在UNIX和开源领域有10年开发经验了。我是20世纪80年代中期GNU最早的贡献者之一，当时我已经在网上发布了一些开源软件，而且还正在开发或者与人合作开发一些程序（如nethack、Emacs的VC和GUD模式、xlife等），这些程序直到现在仍然被广泛使用着，我想我懂这个。

Linux推翻了很多我以为我懂的东西，多年以来，我一直在宣扬“小工具”、“快速原型法”以及“演化式编程”等UNIX信条。但我也相信，如果超过了一定的复杂度，更集中式的管理和更严格的流程是有必要的。我相信大多数重要软件（操作系统和真正大型工具如Emacs编辑器）需要像建造大教堂那样，在与世隔绝的环境下，由天才式专家或几个行家里手精心打造，不成熟时绝不发布beta测试版。

Linus Torvalds的开发风格是：早发布、常发布、委托所有能委托的事、开放到几乎是混乱的程度，这些都令人感到惊讶不已。在Linux社区里，没有建筑大教堂那样的安静和虔诚，倒更像是一个乱糟糟的大集市，充满了各种不同的计划和方法（Linux的文件服务器就是个很好的例子，这里可以接受任何人的代码和文档提交），而既稳定又一致的一个操作系统就这么诞生了，这真是奇迹中的奇迹。

而事实上，集市模式真的管用，而且非常管用，这让所有人震惊。我开始以自己的方式去了解这种模式，除了在我的个人项目中努力探索外，我也

试着去理解为什么Linux世界没有在混乱中四分五裂，反而以大教堂建筑者们难以想象的速度变得越来越强大。

1996年年中，我慢慢开始理解了，而且有幸拥有了一个可以测试我的理论的机会，这个机会使我可以有意识地在集市模式下尝试一个开源项目，我这么做了，更有意义的是，它成功了。

我要讲述的就是这个项目的故事，通过这个故事，我将引出一些在开源开发中很有用的格言警句。虽然对我来说，这些不都是从Linux中学到的，但我们可以看看Linux是怎样淋漓尽致地运用这些理论。如果我是对的，这些格言警句会帮助你准确地理解到底是什么让Linux社区能够源源不断地产生这么多好软件，而且，也许这些格言还能帮助你成为一个富有成效的人。

2.2 邮件必达

“切斯特互联”（Chester County InterLink，CCIL）位于美国宾夕法尼亚州（Pennsylvania）的西切斯特郡（West Chester），是一家小型的免费互联网服务提供商。1993年以来，作为联合创始人，我曾一直负责CCIL的技术部分，并编写了我们独创的多用户论坛程序——你可以通过telnet连到locke.ccil.org来试试看。时至今日，这个程序通过三十多条线路支持近三千名用户访问。这份工作使我能够使用CCIL的56K线路保持每天24小时连在网络上，当然，这是工作需要！

我已经习惯了使用电子邮件，但每过一会儿就telnet到locke上查一下邮件，是一件比较烦人的事。我很希望邮件能够自动地递送到snark（我家里的机器）上，这样，邮件来的时候就会通知我，然后我就能用本地工具来处理它。

互联网原生的邮件转发协议SMTP（Simple Mail Transfer Protocol，简单邮件传输协议）并不适用，因为它最适用于机器一直在线的情况，而我家里的机器并不总是在线，而且也没有一个静态IP地址。我需要这样一个程序，它可以在我时断时续的拨号上网期间，把我的邮件取到本地。我知道有这种软件，它们大多使用了应用层协议POP（Post Office Protocol，邮局协议）。现在绝大多数常见邮件客户端都支持POP，但在那个年代，我所用的邮件阅读器并不支持。

看来我需要一个POP3客户端。于是我便到网上找了一个（事实上我找到了三四个），使用了一段时间后，我发现它有个明显的缺陷：它不能正确地解析取回邮件的邮箱地址，导致我不能正确回复。

问题是这样的：比如locke上有个叫joe的人给我写信，我把信取到snark上并试图回复时，邮件程序会傻乎乎地想把它发给snark上并不存在的joe。所以我不得不每次把回复邮件地址修改为*@ccil.org*的样子，这实在有点痛苦。

很明显这种事应该由计算机搞定，但没有任何一个现成的POP客户端能做到这点！这给我们上了第一课：

1. 好的软件作品，往往源自于开发者的个人需要。

按说这是显而易见的（正如老话说“需要是发明之母”），但太多的软件开发人员并不需要也不热爱他们正在开发的软件，他们把编程当差事，为的只是拿薪酬。Linux世界里可不是这样——也许这可以解释为什么Linux社区里原创软件的平均质量是如此之高。

那么，我是不是应该立即投入到疯狂的编程中，做出一个全新的POP3客户端和其他程序一较高下？千万不要！我把手头的POP程序看来看去，认真思索“哪个最接近我想要的”，因为：

2. 优秀的程序员知道写什么，卓越的程序员知道改写（和重用）什么。

我不敢说自己是卓越的程序员，我只是模仿他们。卓越程序员们有个很重要的特征是“建设性懒惰”，他们知道人们要的是结果而不是勤奋，而从一个部分可行的方案开始，明显要比从零开始容易得多。

以Linus Torvalds为例，他并没有尝试从零开始写Linux，而是以重用Minix（一个用于PC机的迷你型UNIX类操作系统）的代码和理念作为开始，虽然Linux中所有Minix代码最终都被移除或重写，但它在Linux成长初期确实起到了类似脚手架的作用。

基于同样的理念，我试图找到一些代码写得还不错的POP程序，作为我开发的基础。

UNIX世界共享源代码的传统使得代码重用变得很便利（这正是GNU选择UNIX作为基础OS的原因，且不论它对UNIX本身的保留意见），Linux世界则把这个传统发挥到了技术上的极限。相比其他地方，从Linux世界多达数T字节的开放源码中，找到一些他人写的“足够好”的代码要可行得多。

不出意外，连同上次找到的，我一共找到九个备选程序：fetchpop、PopTart、 get-mail、gwpop、pimp、pop-perl、popc、popmail和upop。我首先选择了Seung-Hong Oh写的fetchpop。我给程序加上了“邮件头重写”功能，并做了一些其他改进，这些改进被作者接受并在1.9版本中发布。

几周以后，我偶然发现了Carl Harris写的popclient代码，同时带来的是一个难题：尽管fetchpop有一些很好的创意（比如使用后台进程模式），但只能处理POP3协议，代码也略显业余（Seung-Hong Oh在那时是个聪明但缺乏经验的程序员，这两点都看得出来）。Carl Harris的代码要好一些，专业而稳健，但缺乏fetchpop中一些重要而精巧的特性（包括我写的那部分）。

继续完善fecthpop还是转换到popclient？如果转换，我写的那些代码就可惜了，但同时能换来一个更好的开发基础。

一个更实际的转换动机是popclient对多协议的支持。POP3是最常用的邮件服务器协议，但并不是唯一的。fetchpop以及其他备选程序并不支持诸如POP2、RPOP或AROP这些协议，而我还有点想加上对IMAP（Internet Message Access Protocol，互联网消息访问协议，这是最新设计的、功能最强大的邮局协议，*http://www.imap.org*）的支持，仅仅是为了好玩。

另外，还有一个理论上的原因让我决定转换，这可是早在Linux之前我就学到的：

3. “计划好扔掉一个吧，迟早你会这么做的。”（Fred Brooks，《人月神话》第11章）

或者可以这么说：在你第一次把问题解决的时候，你往往并不了解这个问题，第二次你才可能知道怎么把事情做好。所以，如果你想做对事情，至少要再做一次。[1]

好吧，我对自己说，改写fetchpop是我的第一次尝试，现在我可以换了。

1996年6月25日，我把自己对popclient所做的第一批补丁发给Carl Harris，发现他基本已经失去了对popclient的兴趣。由于popclient代码有点陈旧，还有些小bug没有解决，很多地方都值得改进，我和Carl Harris很快达成一致：由我来接手这个程序。

项目在不经意间升级了，我不再是对现有POP客户端做一些小打小闹的补丁工作，而是开始维护整个程序。新的想法不时冒出来，我意识到自己也许可以对程序做些大改动了。

在一个鼓励代码共享的软件文化中，这是一种很自然的项目演化方式。我见证了下面这条：

4. 如果你有正确的态度，有趣的事情自然会找到你。

当然，Carl Harris的态度更重要，因为他明白：

5. 当你对一个程序不再感兴趣时，你最后的责任就是把它交给一个可以胜任的接棒者。

尽管并没有明确提及，但Carl Harris和我都知道，我们的共同目标是做出最好的解决方案。唯一的问题是我能否证明自己是可靠的，一旦我做到这点，Carl Harris就优雅而利落地把程序交接给我。如果有一天轮到我时，希望我也能交接得这么棒。

2.3 拥有用户的重要性

我就这样继承了popclient，同样重要的是，我继承了popclient的用户群。

拥有用户是一件很美好的事，这不仅表明你正在服务于某种需要，表明你做对了某些事，如果发展得当，他们还会成为你的开发合作者。

UNIX另一个传统强项也被Linux发挥到美妙的极致：很多用户本身就是黑客。因为可以拿到源代码，这些黑客能极为有效地缩短排错时间，只要给他们一点点鼓励，他们就会帮你查找问题、给出建议并帮助改善代码，这些比你自己做要快得多得多。

6. 把你的用户当成开发合作者对待，如果想让代码质量快速提升并有效排错，这是最省心的途径。

这种做法的效力很容易被低估，事实上，连我们这些在开源世界里的人，都极大低估了这种做法的效力，也就是用户越多就越能有效对抗系统的复杂性，直到Linus Torvalds向我们明白地展示这一点。

我想，Linus最聪明和最有价值的成就其实不是构建出一个Linux内核，而是他发明的这种Linux开发模式。有一次我当面向他表达了这个看法，他笑了，平静地重复了他常说的话："我基本上是个很懒的人，别人做事，我得名誉。"像狐狸那样懒，或者像Robert Heinlein曾经描绘的一个很有名的角色，太懒以至于无所不能。

回顾以往，GNU Emacs Lisp库和Lisp代码资源库可能要算是Linux这种成功方法的先例，和Emacs用C语言写的核心以及其他GNU工具不同，Lisp代码池是不断更新的，并且在很大程度上是用户驱动的，大多数新想法和原型在达到最终稳定状态前，都会被重写3到4次，和Linux一样，其频繁的"松耦合"合作都是通过Internet实现的。

说实在的，在fetchmail之前我最成功的作品大概要数Emacs VC（版本控制）模式了，当时我通过邮件和其他三人采用了类似Linux的合作方式。直到今天，我只和其中一人见过面（即Richard Stallman，Emacs的作者，自由软件基金会的创立人，参见*http://www.fsf.org*）。Emacs的VC模式提供了SCCS、RCS以及后来CVS的前端功能，使得用户可以完成"一键式"的版本控制操作。它是从某人写的一个简略而粗糙的sccs.el演化而来的，之所以能取得成功，是因为和Emacs本身不一样，Emacs的Lisp代码可以迅速地完成"发布/测试/改进"循环。

不只是Emacs，还有其他一些软件产品也使用了两层架构和两级用户群，内核使用大教堂模式开发，工具箱(toolbox)使用集市模式开发，比如数据分析和可视化展现的商业化工具MATLAB就是这样， MATLAB和其他类似产品的用户们发现，创新、酝酿和行动最频繁发生的地方总是在产品的开放部分，而这部分的改进也总是由庞大而多样化的用户群完成。

2.4 早发布，常发布

尽早和尽量频繁发布是Linux开发模式中至关重要的一部分，绝大多数开发者（包括我）都习惯性地认为：除非是很小的项目，这么做有害无益，因为软件的早期版本几乎都是问题版本（buggy version），如果早早发布，恐怕会耗尽用户们的耐心。

这种观念使人们更倾向于支持大教堂开发模式，但如果最重要的目标是给用户提供bug尽量少的软件，为什么你只是每六个月（或者更长间隔时间）才发布一个版本，并且在版本发布的间隔里忙得喘不过气来呢？Emacs用C写的内核是按照这种方式开发的，而Lisp库却不是，因为真正起作用的Lisp代码资源库并不在FSF的控制之下，在那里你可以找到新的和不断发展中的代码版本，而且它们都独立于Emacs的发布周期。[2]

这里面最重要的要数俄亥俄州（Ohio）的Emacs Lisp代码资源库，该库早早就拥有了如今大规模Linux资源库所具备的精神和特性，但当时我们几乎没人认真去想想我们在做什么，或者想想资源库的这种存在，是否意味着FSF大教堂式的开发模式有点问题。1992年左右我做过一次认真的尝试，我从这个代码库中下载了很多代码，并将其正式融入Emacs的官方Lisp库中，很快我遇到了类似政治上的麻烦，尝试非常不成功。

但是一年后，Linux越来越广为人知，它明显与众不同而且要健康得多。Linus开放式的开发策略简直就是和大教堂模式对着干，Linux的Internet资源库生机勃勃，多个不同版本同时流传，而这完全是由Linux内核那前所未闻的频繁发布所驱动的。

Linus把他的用户当作开发合作者看待，并以一种尽可能最有效的方式：

7. 早发布，常发布，倾听用户的反馈。

Linus的创新之处，并不完全在于大量采纳用户反馈并快速发布系统版本（这也是UNIX世界多年来的传统），而更多在于将这种做法强化到一种能和系统复杂度相匹配的强度。我们知道他在早期（1991年左右）发布内核的频率会超过一天一次！这要归功于他在发展合作开发群体方面的努力，Linus比其他任何人都更在意如何利用Internet杠杆促进合作，而且他真的做到了。

他是怎么做到的？我能否加以复制？还是说只有Linus Torvalds这样的天才才能驾驭？

我想不是。Linus无疑是一个顶级黑客，想想有多少人能从零开始建造一个完整的具有产品级质量的操作系统内核？但Linux并没有展现出多少令人赞叹的概念性突破。和Richard Stallman或者James Gosling（NeWS和Java的作者）相比，Linus不是（至少现在还不是）一个富有创造性的设计天才，他更像是一个工程实施上的天才，他具备一种避免bug和防范开发走入死胡同的第六感，而且有一种能发现从A点到B点最省力路径的真本事，事实上，Linux的整个设计，都透露着这种特质，并反映了Linus那种本质上保守而简洁的设计取向。

所以，如果快速发布和利用互联网杠杆效应不是碰巧而为，而是Linus慧眼发现的最省力路径，那么他最想利用的是什么？什么是他最想从这种开发机制中获取的好处？

这样一问，答案就显而易见了。Linus在持续不断地激励和回报着他的黑客/用户，用自我满足感激励他们，用持续改进（甚至每天都有改进）回报他们。

Linus的直接目标就是将投入排错和开发的“人时”（person-hour）最大化，即便这样做可能导致代码不稳定，或者可能因为一些难以消除的严重bug导致用户群流失，Linus也在所不惜，他相信：

8. 如果有足够多的beta测试者[⊖]和合作开发者，几乎所有问题都会很快显现，然后自然有人会把它解决。

或者说得更通俗一些：“只要眼睛多，bug容易捉。”我把它称为“Linus定律”。

最初我的表达是“每个问题都会有人弄明白”。Linus提出异议，他认为那个弄明白并修复问题的人往往不是也没有必要是那个首先发现问题的人，“有人发现问题，”他说，“另有人搞定问题，我可以公开地说，发现问题更具挑战性。”这个改正很重要。我们会在下一节仔细考查排错过程到底是怎样的，但关键在于，Linux模式下排错的两个部分（发现问题和修复问题）通常都很快。

Linus定律道出了大教堂模式和集市模式最关键的区别：在大教堂建筑者看来，bug是棘手的、难以发现的、隐藏在深处的，要经过几个人数月的全心投入和仔细检查，才能有点信心说已经剔除了所有错误。而发布间隔越长，倘若等待已久的发布版本并不完美，人们的失望就越发不可避免。

对集市模式而言则完全不同，在上千名合作开发者热切钻研每个新发布版本的情况下，你可以假定bug是浅显易找的，或者至少可以很快变得浅显易找。所以你会频繁发布以获取更多的修正，其副作用是良性的：即便发布中有些小问题，你也不会损失太多。

就是这样，这就足够了。如果Linus定律是错的，那么任何一个像Linux内核这么复杂的系统，经过如此多黑客的改动，在无法预见的不良交互影响以及难以发现的“深度隐藏”bug的重压下，应该已然在某个时刻轰然倒塌了。如果Linus定律是对的，它可以很好解释Linux为什么bug相对较少，且连续运行时间能够超过数月甚至数年。

⊖ beta测试即“β测试”，和“α测试”相对应。对于一个即将面世的软件产品，α测试是指软件公司组织内部测试人员模拟各类用户行为对产品(此时为α版本)进行测试。随后的β测试是指软件公司组织各类典型用户在日常工作中实际使用（此时产品为β版本），并要求用户报告错误及异常情况。最后软件公司再对β版本进行改错和完善。——译者注

也许人们不该为这个定律而惊讶，社会学家早在多年前就发现，一群专家（或一群无知的家伙）的平均观点要比一个随机选择的人的观点更有预见性，这就是“德尔菲效应”(Delphi effect)。看来Linus的做法表明这个理论甚至也适用于操作系统排错——“德尔菲效应”可以驯服软件开发的复杂性，甚至是操作系统内核开发这样的复杂性。[3]

Linux对“德尔菲效应”提供支持的一个特别之处在于，给定任何一个Linux软件项目，其成员都是自发选择参与的。早前有读者指出，对Linux做出贡献的人并不是没有规律的，他们是这样的个体：他们有很大的兴趣使用软件、了解其工作原理、尝试解决遇到的问题并实际给出一个明显合理的解决办法。具备这些特点的人很有可能贡献出有价值的东西。

Linus定律也可被表述为“排错可以并行”，尽管调试人员在排错时需要协调开发人员并与之交流，但调试人员之间并不怎么需要协调，也就是说，增加调试人员并不会带来增加开发人员那样的二次方复杂性和管理成本。

理论上，并行排错会由于重复劳动导致效率损失，但从Linux世界的实践来看，这似乎从来不是一个问题。“早发布、常发布”策略的一个效果就是快速传播反馈回来的修复，从而使重复劳动最小化。[4]

Brooks（《人月神话》的作者）曾经在非正式场合说过：“对于一个被广泛使用的软件，其维护成本通常是开发成本的40%或者更多。令人惊奇的是，这个成本受到用户数的严重影响，用户越多，发现的bug就会越多。”

“用户越多，bug越多”是因为增加用户就会增加程序检验的方式。当用户是合作开发者时，这种效应会被放大，每个着手去发现bug的人，都会有不同的视角，并使用各自略微不同的捕捉方法和分析工具。“德尔菲效应”似乎就是因为这种差异而变得有效。在排错这个特定场合下，差异也使得重复工作倾向于变少。

从开发者角度来看，增加更多的beta测试者似乎并不能减少当前“隐藏最深”bug的复杂度，但这种做法会使bug发现的概率变大：某人的那套环境正好匹配某个问题，使得那个bug容易被他发现。

为防范严重bug给用户带来的影响，Linus有这么一招：在Linux内核版本号上加以标识（可以从版本号看出系统是否稳定——译者注），潜在用户要

么选择上一个被标识为“稳定”的版本，要么冒着有bug的风险使用最新版本以获取新特性。这种策略还没有被Linux黑客们系统性地加以模仿，也许他们以后会这样做。事实上，给用户以选择使得两种版本都更具吸引力。[5]

2.5 多少只眼睛才能驯服复杂性

从宏观上观察到集市模式能极大加速代码排错和演化是一回事，但从微观上，结合开发者和测试者的日常行为，完全理解这是怎样做到的以及为什么会这样，就另是一回事了。本节（在本文首版的三年后写就，采纳了阅读本文首版并对照了他们自身行为的开发者们的观点）我们将仔细审视一下其真正机制。对技术不感兴趣的读者可以直接跳到下一节。

理解这个问题的关键在于要弄清楚这个现象：如果报告bug的用户对源码不关心，则其报告通常不会很有用。对源码不关心的用户，往往报告的都是表面症状，他们把自己的运行环境当成是理所当然的，他们不仅省略了重要的背景数据，而且很少给出重现bug的可靠方法。

这里隐含的问题是开发者和测试者对程序有着不匹配的思维模式，测试者是从外往内看，程序员是从内往外看。对于不开放源码的软件开发，开发者与测试者往往局限于自己的角色，各说各话，都对对方倍感沮丧。

开源开发打破了这种困境，由于大家都有真实的源码，开发者和测试者很容易发展出一个共享的表达模式并进行有效的交流。事实上，一个仅描述外部可见症状的bug报告，和一个直接关联到源码的分析型bug报告，对开发者而言简直是天壤之别。

只要能有一个对出错条件在源码级别上的提示性描述（即便不完整），大多数bug在大多数时间里就很容易被发现。如果你的beta测试人员中有人指出“在第n行有一个边界问题”，或者仅仅指出“在条件X、Y和Z下，这个变量会溢出”，你扫一眼那部分代码，往往很快就能准确找到出错模式并得出修正办法。

所以，如果beta测试人员和核心开发人员都能意识到源代码的作用，就能极大增强双方沟通和合作的效果。相应地，即便在合作者很多的情况下，核心开发人员的时间也会节省很多。

开源方法之所以能节省开发者的时间，另一个原因是开源项目所常采用的沟通模式，以前我习惯使用“核心开发人员”这个术语，主要想区分一下项目核心人员（通常很少，最常见的是一个人，典型情况是一到三人）和外围人员，外围人员通常由beta测试者和潜在的贡献者组成（通常会达到数百人）。

传统软件开发在组织结构上的根本问题由Brooks定律一语道破：“在一个已经延期的项目上增加人手，只会让项目更加延期。”更为一般地讲，Brooks定律指出，随着开发人员数目的增长，项目复杂度和沟通成本按照人数的平方增加，而工作成果只会呈线性增长。

Brooks定律是建立在经验基础上的，人们发现，bug很容易集中在不同人写的代码的交互接口上，沟通/协调的开销会随开发者间接口数的增加而增多，也就是说，问题规模和开发人员间的沟通路径数相关，即和人数的平方相关（更精确地讲，应该是$N(N-1)/2$，N代表开发者数目）。

Brooks定律（以及随之而来对开发团队规模的恐惧）建立在这样的假设上：项目的沟通结构是一个完全图[⊖]，即人人之间都沟通。但在开源项目中，外围开发者实际工作在分散而并行的子任务上，他们之间几乎不交流；代码修改和bug报告都会流向核心团队，只有在那个小的核心团队里才会有Brooks开销。[6]

源码级bug报告非常有用的理由还有很多，但都围绕着这个事实：一个错误可能会导致多种症状，因用户使用方式和环境不同而有不同表现。这种错误往往正是那种复杂而狡猾的bug（比如动态内存管理错误或者非确定中断窗口的产物），这些bug很难重现，也很难通过静态分析找到根源，导致软件中长期存在一些难以解决的问题。

如果测试者能给出一个源码级的bug报告（这个bug可能会引起多种症状），尽管只是一个不确定的试探性描述（比如“看上去第1250行有一个信号处理窗口”或者“你在什么地方把这个buffer清零的”），也会帮助开发人员找到解决那些多症状（症状看上去也许并不相关）问题的关键线索，要知道开发人员往往离代码太近而不能发现问题。这种情况下，要想

⊖ 完全图是一个有n个顶点的图，每两个顶点之间都有且只有一条边，也即共有$n(n-1)/2$条边。——译者注

精确说出是哪个bug导致了哪个外部可见问题，通常很难甚至不可能，但如果经常发布的话，就没有必要去知道了，其他合作者会很快去查看他们发现的bug是否被解决了。很多情况下，人们会关心导致问题消失的源码级bug报告，但很少关心是哪次补丁修复了它。

对于复杂的多症状错误，要想从表面症状追踪到实际bug，往往有多种路径，开发者或测试者追踪时经由何种途径，取决于千差万别的个人运行环境，并且该环境会随时间产生不确定的变化。实际上，每个开发者和测试者在寻找病状症结的时候，都是在“半随机”（semi-random）的变量集合上对程序状态空间进行采样。越是隐蔽和复杂的bug，就越难从技能上保证采样的对症性。

对于简单和容易重现的bug，重点要放在“半”而非“随机”上，此时，调试技能以及对代码和架构的熟悉程度会大显身手。对于复杂的bug，重点就要放在“随机”上了，这种情况下多人共同追踪bug远比少数几个人循序追踪要有效得多——即便这几个人的平均技能要高很多。

当从表面症状追踪到bug的难度不一且难以预测时，并行纠错的效果会更加显著。一个开发人员会循序选取追踪路径，可能一开始就选了一条困难的路径，当然也可能是容易的路径。而在快速发布模式下，如果多人同时尝试对bug进行追踪，很可能某人立刻就发现了最简单路径，然后在比别人短得多的时间内逮住bug。项目维护人员看到后会发布一个新版本，这样在其他更难路径上追踪该bug的人就可以停下来，以免浪费更多的时间。[7]

2.6 何时名不再符实

在研究了Linus的做法并得出他为什么成功的理论后，我决定在我的新项目（当然没有Linux那么复杂和艰巨）里测试一下这个理论。

但我首先要做的一件事，是大幅度重组和简化popclient，Carl Harris的实现很好使，但表现出了一种不必要的复杂性，这在C程序员中很常见。他把代码作为最重要部分，而将数据结构置于辅助地位。结果就是，代码很漂亮，但数据结构设计得有点随意和潦草（至少从一个LISP老手的标准来看）。

除了提升代码和数据结构的质量之外，我重写的另一个目的是想把它弄成一个我完全理解的东西。如果你对一个程序不能了如指掌，而又要负责他的bug修复，那可就不好玩了。

最初一个月左右，我只是继续遵循Cral Harris的基本设计。对程序所做的第一个重大改变是添加对IMAP的支持，我把协议处理部分重新组织成一个通用的驱动和三个方法表（POP2、POP3和IMAP）。这次改动（以及先前的）再次说明了一个程序员应该铭记在心的一般原则（尤其对于C这种并不天然支持动态类型的语言）：

9. 聪明的数据结构配上愚笨的代码，远比反过来要好得多。

Brooks在《人月神话》的第9章里说："让我看你的流程图但不让我看表，我会仍然搞不明白。给我看你的表，一般我就不再需要你的流程图了，表能让人一目了然。"历经30年的术语/文化变迁，这个道理依旧没变。

这个时候（1996年9月初，即从零开始的六周后），我开始考虑给软件换个名字，毕竟它已不再只是一个POP客户端。但我有点犹豫，因为在设计上还没有做出什么真正新的东西，popclient还需要有点我自己的东西。

当popclient学会如何把取来的邮件转发到SMTP端口后，软件就有了根本的变化，我们一会儿再谈这个。先说说前面我提到的用这个项目验证我所发现的关于Linus成功的理论，（你可能会问）我是如何做的呢？有这么几个办法：

- 我尽早发布并频繁发布（几乎从来没有低于10天一次的频率，在高强度开发阶段会一天一次）。
- 我把每一个因fetchmail联系我的人都加到beta列表（是指beta测试人员邮件列表——译者注）中。
- 每次发布新版本时，我都向beta列表发送朋友对话般的通知，鼓励他们参与。
- 我听取beta测试者们的意见，征求他们关于设计决策的看法，当他们发来补丁和反馈时给他们以热情回应。

这些简单措施立刻收到了回报。从项目一开始，我就收到一些质量高到

让程序员们垂涎欲滴的那种bug报告，而且常常还附带了很不错的修复方法；我还收到经过深思熟虑的批评；收到粉丝来信；收到很有智慧的软件特性建议。这一切都表明：

10. 如果你把beta测试者当做最珍贵的资源对待，他们就会成为你最珍贵的资源。

衡量fetchmail有多成功的一个有趣指标是beta列表（fetchmail-friends列表）的规模，在本文最新一版时（2000年11月），列表成员达到了287名之多，并且每周还增加2到3名。

实际上，列表成员最多时接近300名，1997年5月底我修订这篇文章时发现，成员开始流失。一些人要求我把他们从邮件列表中去掉，而原因很有趣：他们觉得fetchmail已经足够好了，他们不想再看到关于这个项目的邮件往来！对于一个成熟的集市模式项目，也许这是其正常生命周期的一部分吧。

2.7 popclient变成了fetchmail

这个项目真正的转折点是在Harry Hochheiser发来了他的代码草稿之后，看完这段将邮件转发到客户端机器SMTP端口的代码，我立刻意识到，如果能可靠地实现这个特性，其他的邮件投递模式都可以废弃了。

曾经有好几周，我都在一步步对fetchmail进行微调，我觉得fetchmail的界面设计虽然可用但有点乱，不够优雅，选项琐碎且遍布各处，尤其是那个将已收取邮件导出为邮箱文件或打印到标准输出上的选项让我感到很烦，虽然我也说不出为什么。

（如果你不关心互联网邮件的技术细节，尽可以跳过下面两段。）

当考虑SMTP转发的时候，我发现popclient想做的事太多了。它被设计为既是一个邮件发送代理（MTA）又是一个本地的邮件投递代理（MDA），有了SMTP转发，就应该把MDA去掉而成为一个纯MTA，由它将邮件再发给其他诸如sendmail之类的本地投递工具。

在所有支持TCP/IP的平台上都已缺省保证25号端口打开的情况下，为什

么还要折腾复杂的邮件投递代理配置或者去设置邮箱的“加锁并添加”（lock-and-append）选项？而采用SMTP转发带来的另一个特别好处是：取回的邮件看上去就像原发的SMTP邮件一样，这可正是我们想要的。

（返回到非细节层面……）

即便你没有读上面那些技术细节，下面还是给你准备了一些重要经验。我特意模仿Linus方法后所得到的最大收获是“SMTP转发”概念，是用户给了我这个极妙的想法——我所做的只是去理解其意义。

11. 仅次于拥有好主意的是，识别来自用户的好主意，有时后者会更好。

很有趣的是，如果你发自内心地谦逊，并承认你欠别人很多，你将很快发现世界会这样对待你：他们认为是你发明了整个软件，而且你对自己的天赋有着得体的谦虚。我们可以看到这一点在Linus身上体现得有多好！

（当我1997年8月在Perl大会上第一次说出这些的时候，黑客泰斗Larry Wall在我前一排，他以一种宗教复兴般的状态大声喊道：“说下去，说下去，兄弟！”所有的听众都笑了，因为他们知道这一点在Perl发明人Larry Wall身上也适用。）

我以这样的精神将项目运行几周后，开始收到同样的赞扬之词，不仅仅来自于用户，也来自一些对此有所耳闻的人。我把这些邮件收藏了起来，如果什么时候我开始怀疑人生，我就把它们拿出来看看:-)。

接下来是两个更基础的、非政治性的经验，适用于所有类型的设计。

12. 通常，那些最有突破性和最有创新力的解决方案来自于你认识到你对问题的基本观念是错的。

我曾尝试解决错误的问题，那时我总是想把popclient做成一个MTA和MDA的结合体，并支持各种各样古怪的本地投递模式。而事实上，应该彻底重新思考fetchmail的设计，它应该是一个纯粹的MTA，成为Internet邮件常规SMTP对话路径的一个部分。

当你发现自己在开发中碰壁时，当你发现自己苦思冥想也很难做出下一个补丁时，通常你不该问自己是否找到了正确答案，而是该问你是否提出了正确的问题，因为也许问题本身需要被重新定义。

于是，我重新定义了我的问题。显然，正确的做法应该是：（1）将SMTP转发做到通用驱动里面；(2)将它设为默认模式；（3）最后，将所有其他投递模式都扔掉，尤其是投递到文件（deliver-to-file）和投递到标准输出（deliver-to-standard-output）的选项。

对第3步我犹豫了一段时间，主要是害怕影响那些依赖其他投递模式的popclient老用户们。理论上讲，他们可以立刻使用.forward文件（如果使用sendmail——译者注）或者其他非sendmail软件的类似功能来获取同样的效果，但在实际操作中，这个转换可能会让人有点头大。

但当我这样做了以后，好处非常明显，驱动代码中最让人厌烦的部分不见了。配置得到根本上的简化——不再需要低声下气地围绕MDA和用户邮箱打转了，也不再担心底层的OS是否支持文件加锁。

并且，信件丢失的唯一途径也不见了——如果你让程序投递到文件而磁盘满了，信件就会丢失。而使用SMTP转发就不会有这种问题，因为SMTP监听程序只在信件被正常投递或者至少被缓存后，才会返回确认消息。

性能也得到提升了（尽管你不是运行一次就能感觉到的），另一个不太明显的好处是，用户手册也变得更简洁了。

后来，为了处理一些涉及动态SLIP的棘手问题，我不得不恢复了对“通过用户指定的本地MDA投递”的支持，但这个做起来容易多了。

这件事寓意何在？在不损失效能的前提下，不要犹豫，扔掉那些过时的特性吧。Antoine de Saint-Exupéry[⊖]（在不写经典儿童读物的时候，他是一名飞行员和飞行器设计师）说过：

⊖ Antoine de Saint-Exupéry（1900年6月－1944年7月），法国作家、诗人和飞行员，是经典小说《小王子》的作者。——译者序

13. “设计上的完美不是没有东西可以再加，而是没有东西可以再减。”

当你的代码变得既好又简单，你就知道你做对了，在这个过程中，fetchmail有了自己的特点，和它的前身popclient不再一样了。

现在是时候改名字了。和老的popclient相比，新的设计看上去更像是sendmail的搭档，两个都是MTA，sendmail是先“推”(push)后投递，新的popclient先“拉”(pull)后投递。所以，在开工两个月后，我把它重命名为fetchmail。

这个故事还告诉我们一个更通用的道理，不仅是排错过程可以并行，开发和设计也可以并行（而且能达到让人惊讶的程度）。如果你采用快速迭代开发模式，开发和改进过程就可能成为排错过程的一个特例——修复软件原先在功能或概念上的“疏漏型bug”（bug of omission）。

即便是高层次的设计，如果能有很多合作开发者在你产品的设计空间周围探索，也是很有价值的。设想下一滩雨水是怎么找到下水口的，或者说蚂蚁是怎么发现食物的。探索在本质上是分散行动，并通过一种可扩展的通信机制来协调整体行为。这很有效，就像Harry Hochheiser和我，一个外围的游走者可能会在你旁边发现宝藏，而你可能有点过于专注而没能发现。

2.8 fetchmail长大了

现在我有了一个简洁和新颖的设计，由于我自己每天都用，我知道代码还不错，并且有了一个不断繁荣的beta列表，我逐渐明白我不再仅仅是做一些微不足道的个人编程，做出来的东西也不再只是对少数几人有用。我现在做的程序，是每一个使用UNIX和SLIP/PPP收发邮件的黑客都需要的。

有了SMTP转发功能，它远远地走在了其他竞争对手的前面，并逐渐成为这个领域内的杀手应用，由于它在同类程序中如此优秀，以至于其他对手不仅被抛弃，而且几乎都被遗忘了。

不过，你一定不要一开始就设定这样的目标和结果。你必须要有一个非常强大的设计创意并完全投入，以至于这个结果就像是不可避免、自然而然

和预先设定的。要做到这一点，唯一的途径是你有很多创意——或者有一种采纳别人好主意的工程上的决策力，并将这些创意发展到超越其作者所能想象的地步。

Andrew Tanenbaum写了一个简单的用于IBM PC的原生UNIX，其原意是将其用做教学工具（他称之为Minix）。Linus Torvalds将Minix的概念发展到Andrew可能无法想象的地步——并成长为让人赞叹不已的产物。同样（尽管规模较小），我吸收并努力推进Carl Harris和Harry Hochheiser的创意。我们都不是那种有浪漫原创精神的天才式人物，但是，大多数科学、工程以及软件开发都不是天才完成的，在青史上留名的往往是黑客。

取得这样的结果真是让人感觉太好了——事实上，这正是黑客们所追求的成功！这意味着我必须要把标准设置得更高一些。现在看来，为了让fetchmail有更好的发展，我不应该只是为了自己的需求而写程序，而应该加入和支持一些他人需要的特性，同时还要让程序保持它的简单性和健壮性。

意识到这点以后，我发现首先要实现的最为重要的特性是对集体邮箱(multidrop)的支持，即有能力从集体邮箱（这里保存着一组用户的邮件）中把邮件取出来，然后将其路由到相应的收件人那里。

我之所以决定增加集体邮箱支持，部分原因是一些用户一直在强烈要求，但更主要的原因是，这可以强迫我以一种完全通用的方式去处理邮件地址，从而借此机会清除掉单邮箱实现中的一些bug，事实上这的确奏效了。弄明白RFC 822（*http://info.internet.isi.edu:80/in-notes/rfc/files/rfc822.txt*）中描述的邮件地址解析规则着实花了我很长时间，不是因为其中有什么地方很难懂，而是里面牵扯了太多相互关联而又繁琐的细节。

从结果看，对集体邮箱的支持是一个很英明的决定。因为：

14. 任何工具都应具备预期内的功能，但一个伟大的工具能给你带来预期外的功能。

fetchmail的集体邮箱模式就有一个用户预期外的功能：它能在客户端（译者注：指连往ISP邮件服务器的终端）上保存地址列表并实现别名扩展，

这意味着用户只要有一台个人电脑和一个ISP账号，就能够维护一个邮件列表，而无需总是读写ISP端的别名文件。

我的beta测试者们要求的另一个重要改动是支持8位MIME（多用途Internet邮件扩展）操作。这很容易做到，因为我已经很小心地保持着第8位的纯洁（也就是说，没有染指ASCII字符集中没用的第8位去让它携带程序信息），并不是我预先想到会有人提这个需求，而是我遵从了另一个规则：

15. 写网关类软件时，尽可能不要干扰数据流，而且绝不要扔掉信息，除非接收方强迫你这么做。

如果我没有遵循这条规则，对8位MIME的支持就会变得困难且容易出错。现在，我要做的只是阅读MIME标准（RFC 1652, *http://info.internet.isi.edu:80/in-notes/rfc/ files/rfc1652.txt*），并稍稍增加一些邮件头部生成的逻辑。

一些欧洲用户一直要求我增加一个选项限制每次连接取回的邮件数（以便他们控制昂贵的拨号上网费用）。我抵制了很长时间，而且直到现在对此也不太开心。但如果你是为大家写程序，你就不得不倾听客户意见——虽然他们并不付钱给你。

2.9 从fetchmail学到的其他经验

在回到一般性的软件工程话题前，还有几个fetchmail的特定经验值得思索。对技术细节不关心的读者完全可以跳过这一节。

rc控制文件（fetchmail的配置文件——译者注）的语法包括了一些不会被解析的“噪声”关键字，相比传统配置文件只剩下简洁的“关键字－值”配对，fetchmail这种类似英语的语法使文件更易读。

这个想法来自于某个深夜，当时我注意到rc文件的声明越来越像一个微型指令式语言（这也是为什么我把原先popclient中的“server”关键字换成了“poll”）。

在我看来，如果把一个微型指令式语言弄得像英语的话，会更方便人们使用。虽然我很崇尚“让它像语言那样”的设计哲学（正如Emacs、HTML以及一些数据库引擎），但我并不痴迷于“让它像英语那样”。

传统程序员倾向于喜欢那种非常简洁、紧凑和没有一点冗余的控制语法。这是计算资源昂贵年代的文化遗留，在当时看来，解析过程应尽可能简单和节省资源。英语大约有50%的冗余度，很不适宜作为控制语法的模型。

我并不是因此而不喜欢英语语法，相反，提及它正是为了打破传统观念。有了更便宜的计算资源，简洁就不该成为最终目标。对现如今的计算机语言来说，是否便于人类使用要比是否节省计算资源更重要。然而顾虑是有理由的：一方面，解析过程有着复杂性成本——你当然不想让它复杂到容易出错和困惑用户的地步；另一方面，试图让语言拥有类似英语的语法，往往使这个所谓的“英语”严重走形，以至于这种对自然语言的表面模仿并不比传统语法让人更易懂一些（你可以在很多所谓“第四代语言”和商业数据库查询语言中看到恶果）。

16. 当你的语言还远不是图灵完备（Turing-complete）的时候，语法糖[⊖]会让你受益良多。

还有一个关于信息隐藏的安全话题。一些fetchmail用户要求我改动软件，使rc文件的密码能够以加密形式保存，这样能防范一个窥探者轻易看到密码。

我没有答应，因为实际上这并不能增强安全性。任何人如果获取了你的rc文件的读权限，都有能力以你的身份运行fetchmail。此外，如果他们想要你的密码，他们能从fetchmail代码中剥离出一个解码器，然后获取密码。

给.fetchmailrc中的密码加密，只会给那些没有认真思考的人一种安全假象，一般而言：

17. 系统的安全性只取决于它所拥有的秘密。谨防虚假的秘密。

2.10 集市模式的必要条件

这篇文章的早期读者和评论者一直不断提出这样的问题，即集市模式成功

⊖ 语法糖（Syntactic Sugar，或译为语法糖衣）是英国计算机科学家Peter J. Landin发明的术语，是指为计算机语言添加某种不会影响语言功能的成分，但却使其更易用一些，目的是增强代码可读性，避免出错的机会。——译者注

的前提是什么，包括项目领导人应该具备什么资格，项目首次发布并建立合作开发社区的时候，代码应该达到什么状态等。

显然，你不可能从零开始实施集市模式[8]。可以用集市模式测试、排错和完善项目，但以集市模式从零开始一个项目是非常困难的。Linus没有这么试过，我也没有。开发者社区从成立伊始，就需要一个可以运行和测试的东西。

当开始建设社区的时候，你需要拿出一个像样的承诺。程序此时并不需要特别好，它可以简陋、有错、不完整，文档可以少得可怜。但它至少要做到：(a) 能运行，(b) 让潜在的合作开发者相信，这个软件在可预见的未来，能演变成一个非常棒的东西。

Linux和fetchmail在首次露面时，都有着很强和很吸引人的设计。很多人都已正确意识到这点非常重要，并进而得出结论：项目领导人必须要有高度的设计直觉和聪明才智。

但Linus的设计来自于UNIX，我的最初设计则来自于popclient（虽然后来做了很多改动，而且改动比例要远大于Linux）。所以，一个尝试集市模式的项目领导人或协调人，是否真的需要他本人是一个超群的设计天才，抑或他能借力于其他人的天才设计？

我想，一个协调者是否拥有卓越的原创设计能力，并不是项目成败的决定性因素，但他是否能识别出别人的优秀创意，则一定是最关键的。

Linux和fetchmail都证实了这一点。Linus虽不是一个让人惊叹的原创设计者（前面说过），但他表现出了能识别优秀设计并将其集成进Linux内核的出色才能。而我也介绍过，fetchmail中最强大的设计（SMTP转发）来自他人。

这篇文章的早期读者曾经很抬举地指出，我之所以低估集市模式的设计原创性，是因为我自身有很多原创，所以视之为理所当然。这可能有点道理，相对于编码或调试而言，设计的确是我的最强项。

但问题是，在软件设计上表现得聪明而有原创性，容易养成一个习惯——在应该保持软件健壮性和简单性的时候，你往往下意识把它弄得既华丽又

复杂。我曾经就因为这样的错误把项目搞砸，所以在fetchmail中，我尽量避免再犯同样的错误。

fetchmail项目之所以能成功，相信部分原因是我限制了表现自己聪明的倾向。这（至少）反驳了设计原创性是集市模式项目成功关键的论点。再来看看Linux，如果Linus Torvalds在开发过程中努力尝试在操作系统设计上表现出基础性创新，那做出来的内核还能像现在这样稳定和成功吗？

当然，一定水准的设计和编码能力还是需要的。但我认为，如果一个人真的想要启动一个集市项目，那么他的能力应该已在最低水准以上。开源社区内在的声誉评价机制会给人们施加微妙的压力，使那些不能胜任项目发展的人，不会去发起一个开发项目。至少到目前为止，这个机制非常有效。

还有一种才能，人们通常不会把它和软件开发联系在一起，但我认为，在集市项目中它和设计才能一样重要——甚至更重要：集市项目的协调人或领导人必须要有很好的人际交往和沟通能力。

这应该是显而易见的，为了建立一个开发社区，你需要吸引人们，让他们对你做的事感兴趣，让他们乐于看到自己的贡献。一些技巧可能有助于实现这些，但远远不是全部，你的人格特征也很重要。

Linus是个好人，人们都喜欢他并愿意帮助他，这（和他的项目成功）不是巧合。我精力充沛、性格外向、乐于社交、有一些脱口秀演员般的说话风格和临场反应，这也不是巧合。为了让集市模式运转，哪怕有一点点的人格魅力，都会对你大有裨益。

2.11 开源软件的社会语境

此言不虚：最好的程序一开始只是作者对自己每天遭遇问题的个人解决方案，程序流传开来则是因为作者遇到的问题成了一大类用户的典型问题。这将我们带回规则1，并以一种可能更有用的方式来重申：

18. 想要解决一个有趣的问题，先去找一个让你感兴趣的问题。

Carl Harris和先前的popclient是这样，我和fetchmail也是这样。这个道理

已经早为人知，而Linux和fetchmail的发展历史让我们关注到更有趣的一点，那就是下一阶段——由用户和共同开发者们组成庞大而活跃的社区，共同促进软件的进化。

在《人月神话》中，Fred Brooks发现程序员的时间是不可替代的，增加开发者进入一个已经延迟的软件项目，只会让项目更加延迟。像我们前面看到的那样，他指出，项目复杂度和沟通成本与开发人员数目的平方成正比，与此同时，工作完成量只会随人数线性增长。Brooks定律已经被广泛地视为真理，但在本文中我们已经通过多种方式论证了开源软件的开发过程不满足这个定律背后的一些假设——并且从实践上看，如果Brooks定律普适于所有开发项目，Linux是不可能完成的。

作为一种后见之明，Gerald Weinberg在经典之作《程序开发心理学》（*The Psychology of Computer Programming*）中提出了对Brooks定律的重要修正。在他关于“无私编程”（egoless programming）的讨论中，Weinberg观察到，在某些工作场所，开发人员不将代码看作是自己的“领土”，而是鼓励别人发现其中的bug和潜在改进点，这些场所中软件改善速度之快，与别处相比是不可同日而语的。（最近，Kent Beck在其“极限编程”（extreme programming）技术中提出的结对编程——两个程序员肩并肩共同完成编程——可以看作是一种效仿。）

也许是Weinberg在用语选择上的问题，导致他的分析未能获得应有的认可——用“无私”来形容互联网上的黑客，这可能会让某些人感到可笑。但我认为他的论点在今天看起来比以往任何时候都更有信服力。

集市模式，运用“无私编程”效果的充足能量，强有力地化解了Brooks定律的影响。Brooks定律背后的原理没有失效，但如果有一个大规模的开发群体和一个低成本的沟通机制，Brooks定律的效果将会被其他非线性因素带来的效果淹没，而后者是大教堂模式下看不到的。这很像是物理学上牛顿理论和爱因斯坦理论的关系——在低能量条件下，老的系统仍然有效，但如果把质量和速度推进到足够大的地步，你就会震惊于核爆炸或Linux。

UNIX历史为我们从Linux中获取经验早已做下了铺垫（为验证其有效性，我专门在一个较小规模的项目上拷贝了Linus的方法[9]），那就是，当编码

在本质上仍然是个体行为时，真正了不起的作品来自于对整个社区注意力及脑力的有效利用。一个在封闭项目中只靠自己的开发者，将远远落后于这种开发者：他们知道如何创建一个开放的、有改进能力的环境，在这个环境中，上百人（甚至上千人）反馈并提供设计空间拓展、代码贡献、bug定位以及软件的其他改进。

传统的UNIX世界中，有一些因素阻止了把这种方法推进到极致。一方面是各种许可证(license)的法律限制、商业秘密和市场利益，另一方面（现在看来）是当时的Internet还不够好。

在互联网变得便宜之前，有一些地域性的协作社区，他们在文化上鼓励Weinberg的“无私编程”，开发人员可以很容易地吸引到很多高水平的评论者和合作开发者。如Bell实验室、MIT的AI实验室和LCS实验室、UC Berkeley大学——这些地方都是创意的家园，它们充满传奇并依然强劲有力。

Linux是第一个有意识并成功将整个世界作为其人才库的项目。Linux孕育之时万维网（World Wide Web）刚刚诞生，Linux蹒跚学步之时（1993－1994）ISP产业开始起飞，与此同时，主流对Internet的兴趣也开始爆发，我认为这都不是巧合，Internet普及之后，Linus是学会如何运用新规则的第一人。

廉价的Internet是Linux模式得以发展的必要条件，但我认为它还不足以成为充分条件。另一个非常重要的因素是领导风格的形成和协作机制的建立，这是吸引合作开发者加入项目的关键，也是充分利用互联网媒介作用的关键。

但应该是什么样的领导风格和协作机制呢？它们不应建立在权力关系上——即便可以这样，强制型的领导风格也无法产生我们今天所见的成果。

Weinberg引用了19世纪俄国无政府主义者Pyotr Alexeyvich Kropotkin所著《一位革命家的回忆》（Memoirs of a Revolutionist）中的一段，很好地诠释了这个问题：

> “我成长于一个农奴主家庭，在投入积极生活之时，像那个年代所有年轻人一样，我非常相信命令、指示、斥责、惩罚等行为的必要性。但当我早期不得不管理重要事业并和（自由）人打交道时，在任何错误都会立刻导致严重后果时，我开始感悟到按“命令与纪律原则”行事和按“共识原则”行事之间的重要区别。前者在军队检阅时的作用令人钦佩，但在真实生活中却一文不值，想要达到目标，必须要靠众人的齐心协力。”

“齐心协力”正是Linux这种项目所需要的——对Internet上（可以看成是无政府主义者的天堂）的志愿者们使用“命令原则”是根本行不通的。如果某个黑客想领导一个协作项目，想要项目有效地运作和竞争，他就不得不学会如何施行Kropotkin所提出的“共识原则”，招募和激励有兴趣的成员形成有效社区。他还必须学会如何使用Linus定律。[10]

前面我提到的“德尔菲效应”也许可以解释Linus定律。但我常常注意到，生物学和经济学中的自适应系统是更好的类比，Linux世界的运转，在很多方面像一个自由市场，或者像一个由很多利己个体组成的生态系统，系统中每个个体都追求自身效用的最大化，在其共生的过程中，能够自然建立起一种具备自我纠错能力的秩序，这种秩序比任何集中式规划都要精妙和高效。这里，正是“共识原则”达成的地方。

Linux黑客们致力于最大化的“效用函数”，其目的并不是经典意义上的经济价值，而是自我满足和黑客声望这些无形的东西。（有人把这种动机称为“利他”，但他们忽视了一个事实，即“利他”本身是“利他者”自我满足的外在表现。）按这种方式运转的志愿者文化其实很常见，除了黑客圈，我还长期参与在科幻迷圈子中，不像黑客，科幻迷们早就清楚认识到“egoboo”[⊖]（个人在团体中声望的提升）是志愿者活动背后的基本驱动力。

⊖ “egoboo”是ego boosting的口语化简称，是指参与志愿工作得到公共认可而获得的快乐，这个术语大约出现在1947年，最早用于科幻迷圈子。egoboo原本是描述人们看到自己名字出现在出版物上的感觉，由于做到这点比较可行的方式是做一些值得被别人提及的事情，该概念很快用到了志愿者活动中。——译者注

Linus成功地将自己置于项目看门人的地位——大多数开发工作是他人完成的，他不断培养大家对这个项目的兴趣直到它能够自我维持下去，这表现出他对Kropotkin“共识原则”的敏锐领会。用这种“准经济”（quasi-economic）视角来观察Linux世界，有助于我们理解“共识原则”是如何应用的。

可以把Linus方法看成是创造一个有效率的“egoboo”市场——把一个个黑客的利己动机尽可能牢靠地牵系到一个艰巨的任务目标上，而这个目标只有在众人持续的合作之下才能达成。正如我在fetchmail项目上所展示的（虽然项目小了点），Linus方法可以被复制并取得很好效果。也许我做得比他更有意识和更有计划一些。

很多人（尤其是那些在意识形态上不相信自由市场的人）把个人导向为主的利己主义文化看成是碎片化的、本位主义的、浪费资源的、不共享和不友善的。这里仅给出一例，就能很有力地证伪这个认识，那就是Linux文档有着令人震惊的广度、深度和质量。程序员痛恨写文档似乎已经成为一个不争的事实，那为什么Linux黑客们还要写出这么多文档？很明显，Linux自由的“egoboo”市场比那些有重金投资的商业软件公司，能够产生更有道德、更利他的行为。

fetchmail和Linux核心项目都表明，如果对参与者的“自我”做适当奖赏，一个优秀的开发者或协调者可以利用Internet获取多开发者的好处，而不会让项目陷入混乱不堪。所以对Brooks定律，我有如下的对立意见：

19. 如果开发协调者有一个至少像Internet这样好的沟通媒介，并且知道如何不靠强制来领导，那么多人合作必然强于单兵作战。

我认为开源软件的未来会越来越属于那些懂得如何玩转Linus定律的人，属于那些离开大教堂并拥抱集市的人。这不是说个人眼光和远大志向不再重要，相反，我认为冲在开源软件最前沿的人，正是凭借自己的眼光和才华而发起项目，并通过构建有效的志愿者社区将之发扬光大。

可能这不只是开源软件的未来。但没有任何闭源开发者可以发动像Linux社区那样庞大的人才库来解决一个问题，也很少有人能雇得起对fetchmail做出贡献的那200多人（1999：600，2000：800）！

可能最终导致开源软件取得胜利的，不是因为“合作是道德正确的”或“软件闭锁[⊖]是道德错误的”（也许你相信后者，但Linus和我不这样认为），而仅仅是由于闭源世界不能赢得一场与开源社区之间的不断演化的军备竞赛，因为后者可以在一个问题上投入比前者多几个数量级的熟练技术工时。

2.12 管理与马其诺防线[⊖]

1997年首版的“大教堂与集市”文章结束于上面这个展望：由程序员和无政府主义者组成的快乐的网络部落，战胜和压倒了等级森严的传统闭源软件世界。

然而，很多怀疑者并不信服，应该对他们提出的问题给予公允的回应。大多数对集市模式的异议都归结到这一点：集市模式支持者低估了传统管理方式带来的生产率乘数效应。

抱有传统观念的软件开发管理者经常会反驳说，开源世界中项目团队形成、变动和解散的随意性，极大抵消了开源社区在人数上相对闭源单人开发者的明显优势。他们说，软件开发真正需要的是持久的努力和客户预期在产品上持续投资的程度，而不是到底有多少人在锅中扔入一块骨头然后让它慢慢炖着。

这并不是没有道理，事实上，我在“魔法锅”（The Magic Cauldron）一文（本书后面的一章）中提出了这样的观点：未来软件产业的经济关键是服务价值。

⊖ 软件闭锁（software hoarding）是指某人或公司试图阻止别人使用或共享他们的软件源码。——译者注

⊖ 第一次世界大战后，为防范德军入侵，法国陆军部长马其诺提议沿法国和德国边境建设一条叫做马其诺防线的防御工事（1929年开始建造，1940年基本建成，长达390公里）。马其诺防线曾被认为是牢不可破的，但德军从侧面包抄并随后占领了这条防线。马其诺防线用来形容表面看似坚固而实际没有价值的东西。——译者注

但这个论点有一个重要的潜在问题，它暗藏了这样的假设：开源开发不能提供“持续的努力”。事实上，有一些开源项目长期保持着连贯的方向和有效的维护社区，而没有任何传统管理认为不可或缺的奖励机制或管控机构。GNU Emacs编辑器是一个极端的、有启发性的例子，尽管该项目的人员流动率很高，从始至今持续起作用的只有一人（Emacs的作者），但15年来，仍然有数百名贡献者投入努力，他们合作造就了一个统一的架构体系，还没有哪个闭源编辑器能匹敌这样的长寿记录。

这倒是给了我们一个质疑传统开发管理有什么优势的理由（与大教堂和集市模式的争议无关）。GNU Emacs能够在15年内保持一致的架构体系，像Linux这样的操作系统在硬件和平台技术不断变化的8年来亦复如是，很多设计优秀的开源项目发展都超过了5年（事实上确实如此）——我们当然有权利质疑，传统开发管理的巨大花费，究竟给我们带来了什么。

不管是什么，它肯定不是这三项目标的可信履行：最后期限、预算和需求书中的所有功能。能达到其中一个目标，就已经是很少见的管得不错的项目，更不用说三个全达到了。它也不是在项目生命周期内适应科技和经济变化的能力，这方面，开源社区已被证明远远更为有效（这很容易被核实，比如说，比较Internet至今30年的历史和专有网络技术很短的半衰期；或者比较微软将Windows从16位过渡到32位的成本以及同时期Linux完成同样升级的毫不费力——不仅仅是在Intel产品线上，而且包括64位Alpha处理器在内的十多类硬件平台）。

一些人认为购买传统模式产品会带来这样的保障：如果项目出错，有人会负责，并为可能的损失买单。但这只是一个幻觉，大多数软件许可证连对商品的保证都没有，更不用说履行责任了——因软件质量差而成功获得赔偿的案例几乎没有。即便很常见，也不要因为可以起诉某人就觉得心安，你想要的不是官司，而是能用的软件。

那么所有这些管理开销能带来什么？

为弄明白这点，我们需要了解软件开发管理者是如何看待自己工作的，我有位朋友看上去在这方面做得很好，她说软件管理有五个功能：

- 明确目标并让大家朝同一个方向努力。
- 监督并确保关键细节不被遗漏。

- 激励人们去做那些乏味但必要的“体力活”。
- 组织人员部署并获得最佳生产力。
- 调配项目所需的资源。

显然所有这些目标都是有价值的，但在开源模式及其所在的社会语境中，人们会惊奇地发现这些目标毫无意义，我们按颠倒过来的顺序分析。

朋友告诉我，很多资源调配基本上是防守性的；一旦你拥有人、机器和办公空间，你就不得不防备同级管理人员对资源的竞争，以及上级对有限资源中最有效部分的调用。

但开源开发者是志愿者，是因为兴趣和能力（能否对项目有所贡献）自主选择的（即便他们因开源工作领取薪水，这也依然适用），志愿者精神倾向于自发去关心资源问题的“解决”，他们会把自己的资源带到工作中，这里几乎没有传统意义上“防守”的必要。

无论如何，在一个廉价PC和快速Internet连接的世界里，我们发现始终如一真正有限的资源是技术人员的关注，开源项目如果失败了，根本不会是因为机器、网络或办公场地，它们死掉的唯一原因就是开发者们不再感兴趣了。

这种情况下，开源黑客通过“自我组织”来最大化生产力就显得加倍重要，自愿者自主选择项目，社会环境则无情地选择能力。我那个朋友对开源世界和大型封闭项目都比较熟悉，她相信开源之所以成功，部分原因是开源文化只接受编程人员中那最有才华的5%。她将自己的大部分时间都花在了组织部署其他的95%，并因而第一手见证到那广为人知的差异：最有才华的程序员和那些刚刚及格的程序员之间，生产率能相差100倍。

人们常常会因为这种差异提出一个棘手的问题：对单个项目或者整个行业来说，撤离出那50%能力较差的程序员会不会更好一些？思考已久的管理者们早就明白，如果传统软件管理的目的仅仅是把能力最差那部分人的净损耗转变成微弱盈余的话，那问题就简单多了。

开源社区的成功使得这个问题更加尖锐，强有力的证据表明，从互联网上招募自主选择的志愿者，通常更便宜、更有效。而传统方式管理的那些写字楼里的人，其中有很多可能宁愿去干点别的。

这很直接把我们带到“动机”问题上，一个经常听到的说法和我朋友的观点等效：传统开发管理是对缺乏激励的程序员们的必要补充，否则他们不会干得很好。

这个答案常常伴随着一种观点，认为开源社区只会去做那些很吸引人或者技术上很好玩的东西，其他“无聊”部分则会被丢在一边（或半途而废），等那些受金钱激励的坐在隔间里的苦工们在管理者的严厉鞭策下机械地将其生产出来。我将从心理学和社会学角度，在“开垦心智层”（Homesteading the Noosphere）一文中对这种观点进行质疑。

如果传统、闭源、严格管理模式的软件开发真的想靠这种由“无聊”部分组成的马其诺防线来防御，那么它之所以在某个应用领域能继续生存下去，只是因为还没人发现这些问题是真正有趣的，并且还没人发现迂回包抄的路径。一旦有开源力量介入这些领域，用户就会发现终于有人是因为问题自身的魅力而去解决它的，就像其他所有需要创造力的工作，若论激励效果，问题自身的魅力比单纯的金钱要有效得多。

单纯为解决动机问题而设立一个传统的管理架构，可能战术不错，但其战略是错误的，这一套在短期看会有效，但长远来说一定会失败。

目前来看，传统的开发管理相对于开源，至少在两方面（资源调配和组织）都没有胜算，而且似乎在第三点（动机）上也快要玩完了。可怜的传统管理者，现在是四面楚歌，在“监管”这个话题上也无法获取一丝安慰，开源社区最强大的一个长项就是非中心化的同行评审，所有致力于细节不被疏漏的传统方法，都无法和它相比。

那么，“明确目标”的作用是否可以作为传统软件项目管理值得存在的正当理由？也许是，但是，比起开源世界由项目领导人和部落长老决定目标的方式，我们需要有好的理由来相信管理委员会和公司路线图所定义的目标在价值上以及在让成员广泛接受上能做得更成功。

但我们很难得出这个结论。并不是因为开源这边（Emacs经久不衰，Linus

Torvalds号召大量开发者们“统领世界”[⊖]）表现太优秀，而是传统机制在定义项目目标方面做得实在太糟。

软件工程中最广为人知的一条大众定理是：传统软件项目中的60%到70%，要么是从未被完成，要么被他们的用户拒绝。如果这个比例还算靠谱的话（我还没见过任何一个有经验的项目管理者对此提出过异议），那么大多数项目把目标设定得要么太不现实，要么完全错误。

这正是如今软件工程领域中，一听到所谓“管委会”就让人后背发凉的原因——即便（或尤其）听者是一名管理者。以前仅仅是程序员抱怨这种模式，现在，呆伯特[⊖]玩偶已经出现在主管的办公桌上。

所以，我们对传统软件开发管理者的答复就很简单了——如果开源社区真的低估了传统管理的价值，那为什么你们中的这么多人都表现出对你们自己流程的不屑？

又一次，开源社区的例子把这个问题变得更为尖锐——我们做这些是为了乐趣。我们创造性的游戏已经在技术上、市场占有率上、观念认同上以令人震惊的速度获得了增长，我们不仅证明了我们可以做出更好的软件，而且证明了快乐也是一种资产。

本文第一版发布已经两年半了，我能提供用来作为结束语的最激进的观点，已经不再是“开源统领软件世界”这样的愿景了，毕竟那些很严肃的西装革履的人，也认为这种观点有些道理了。

更进一步，我想给出一个更普遍的关于软件的经验 （可能适用于所有创造性或专业性工作），人类通常会从一种位于“最佳挑战区”的任务中获得乐趣，也即它不是太容易而让人无聊，也不是太困难而无法完成。一个

⊖ Linus Torvalds于1999年在“The Linux Edge”一文中提到：“Linux现在有数百万用户、数千名开发者和正在增长的市场，Linux用在嵌入式系统中，用在机器人系统中，用在航天飞机上，我想说我早知道这些都会发生，这些都是统领世界计划的一部分”。——译者注

⊖ 呆伯特（Dilbert）是美国作家和漫画家Scott Adams创造的广为流行的职场卡通人物。他是一名计算机工程师，热爱科技、待人友善、憨厚老实，但很不成功，在公司里人微言轻，常常被主管过分要求和利用。——译者注

快乐的程序员是一个既没有被浪费也没有被压垮（由于不适当的目标或过程中充满压力与冲突）的人，乐趣预示着效率。

如果你在工作过程中感到恐惧和厌恶（即便你以自嘲的形式来表达——比如悬挂呆伯特玩偶），就应该意识到过程已经出了问题。快乐、幽默和玩兴是真正的资产，前面我之所以写“快乐部落”（happy horde）并不是为了首字母押韵，而用一只憨态可掬的企鹅作为Linux吉祥物也绝不仅仅是为了搞笑。

现在看来，开源成功的一个最重要成果，就是告诉我们，“玩”是创造性活动中最具经济效能的工作模式。

2.13 后记：网景拥抱“集市模式”

感觉自己正在帮助创造历史的感觉实在很奇妙……

1998年1月22日，大约在我首次发布“大教堂与集市”一文七个月后，网景通信公司宣布了开放“网景通信家”[⊖]源代码的计划（*http://www.netscape.com/newsref/pr/newsrelease558.html*）。此前，我一点消息也不知道。

很快，网景的执行副总裁和首席技术官Eric Hahn给我发了一封电子邮件：“我代表网景公司所有员工，感谢您帮助我们走到这一步，您的思考和写作，对我们的决定有着至关重要的启发意义。”

接下来的一周，我受网景之邀飞抵硅谷，和他们的高管层以及技术人员开了一个长达一整天的战略会议（1998年2月4日），我们在会议上设计了网景源码的发布策略及许可声明。

几天后我写道：

⊖ 网景通信家（Netscape Communicator）是网景通信公司互联网套件（在版本4.0到4.8时）的总称，内含网景导航者（Netscape Navigator）浏览器、电子邮件客户端、新闻组软件、HTML编辑器、多用户通信客户端以及地址簿等辅助工具，在此之前，网景导航者既是整个套件的名字，也是浏览器的名字，容易引起混淆。

> “网景准备在商业世界中给我们提供一个大规模的、真实的集市模式的测试。开源文化现在面临一个危险：如果网景此举失败，那么开源概念将受到严重怀疑，商业世界将在未来十年中都不会再碰它。
>
> 另一方面，这也是一个绝好的机会，华尔街以及其他一些机构，对这件事的初步反应是审慎乐观的。我们也获得了一个证明自己的机会，如果网景通过此举能够重新收复市场份额，那将会引发一场早该到来的软件产业革命。
>
> 接下来的一年将会非常有启发性，也会非常有趣。”

事实上确实如此。2000年年中我修订此文的时候，项目（后来被命名为Mozilla）发展的情况只能说勉强成功，它达到了网景的最初目标——阻止微软在浏览器市场上的垄断锁定。当然它也取得了一些引人注目的成就（特别是下一代浏览器内核Gecko[⊖]的发布）。

然而，它还没能像Mozilla创立者最初希望的那样，聚拢大规模的来自网景外部的开发力量。问题似乎是，在很长一段时间内，Mozilla的发布实际上违反了集市模式的一条基本规则：它的发布缺少能让潜在贡献者可以轻易运行和查看效果的东西（发布后的一年多时间内，编译Mozilla源代码都需要Motif私有库的许可证）。

最消极的是（从外界来看），Mozilla组织在项目开始两年半内都没能发布一款商品级的浏览器。而且在1999年，一位项目核心成员的离职引发了不小的影响，他抱怨项目糟糕的管理和机会的错失，“开源，”他很正确地评价道，“并不能点石成金。”

确实不能。现在（2000年11月）看来，比起Jamie Zawinski写离职信的那个时候，Mozilla的前景有了戏剧性的改善——最近几周的每夜版（nightly releases）已经终于在产品可用性上通过了关键性门槛。但Jamie没有说错，对于一个已有项目，走向开源不一定就能让项目免受目标错误、代码

⊖ Gecko浏览器内核（Rending Engine，又译为排版引擎、渲染引擎）是Mozilla火狐浏览器（FireFox）采用的内核，由于Gecko代码完全公开，因此受到许多人的青睐，使用Gecko内核的浏览器也很多。——译者注

杂乱以及任何其他软件工程慢性病的困扰。Mozilla已经同时示例了如何让开源成功和如何让开源失败。

与此同时，开源理念在其他很多地方已经获得了成功和支持。自网景发布源码以来， 我们看到人们对开源模式的兴趣有爆发式的增长——这种趋势由Linux操作系统推动，并推动Linux继续获得成功，Mozilla引发的潮流将继续加速前行。

HZ BOOKS
华章图书