Eric S. Raymond 五部曲

作者：Eric S. Raymond
翻译：不详
来源：灰狐
整理：MDZ (mdztiger@gmail.com)

September 18, 2008

声　明

1. 本文档内容来源于网络（灰狐社区 <http://docs.huihoo.com/joyfire.net/7-2.html>）。
2. 本人对文档中的部分文字（主要是繁体）和标点作了修改，以更清晰地显示文档的行文。
3. 本文档中的不少链接已经失效，保留原版本文档的链接。
4. 本文档使用 XeLaTeX 排版，其 .tex 文件采用 GNU General Public License 发布。文档内容采用 GNU Free Documentation License 发布。许可请参见 COPYING 和 COPYING.FDL 文件。若要修改本文档内容、传播本文档等，请自觉遵循许可证的要求。
5. 本文档内容源自 Eric S. Raymond 个人网站，翻译由大陆、台湾的多名网友完成。翻译结果若影响理解，本人概不负责。但如条件允许，今后将对难以理解的词句进行修改（保留原译者信息）。
6. 如有任何疑问（比如您知道译者的名字），请发 Email 至 MDZ <mdztiger@gmail.com>，或登录论坛“大师名著” 发帖。
7. 本文档所在项目为“master-zhdoc”。

Contents

1 Hacker 文化简史

1.1 序曲: Real Programmer

故事一开始，我要介绍的是所谓的 Real Programmer。

他们从不自称是 Real Programmer、Hacker 或任何特殊的称号；“Real Programmer”这个名词是在 1980 年代才出现，但早自 1945 年起，电脑科学便不断地吸引世界上头脑最顶尖、想像力最丰富的人投入其中。从 Eckert & Mauchly 发明 ENIAC 后，便不断有狂热的 programmer 投入其中，他们以撰写软件与玩弄各种程序设计技巧为乐，逐渐形成具有自我意识的一套科技文化。当时这批 Real Programmers 主要来自工程界与物理界，他们戴著厚厚的眼镜， 穿聚酯纤维T恤与纯白袜子，用机器语言、汇编语言、FORTRAN 及很多古老的 语言写程序。他们是 Hacker 时代的先驱者，默默贡献，却鲜为人知。

从二次大战结束后到 1970 早期，是打卡计算机与所谓“大铁块”的 mainframes 流行的年代，由 Real Programmer 主宰电脑文化。Hacker 传奇故事如有名的 Mel（收录在 Jargon File 中）、Murphy's Law 的各种版本、mock- German “Blinke nlight”文章都是流传久远的老掉牙笑话了。

§译：Jargon File 亦是本文原作者所编写的，里面收录了很多 Hacker 用语、缩写意义、传奇故事等等。Jargon File 有出版成一本书：The New Hacker's Dictionary，MIT PRESS 出版。也有 Online 版本: http://www.ccil.org/jargon

§译：莫非定律是：当有两条路让你抉择，若其中一条会导致失败，你一定会选到它。 它有很多衍生说法：比如一个程序在 demo 前测试几千几万次都正确无误，但 demo 那一天偏偏就会出 bug。

一些 Real Programmer 仍在世且十分活跃（本文写在 1996 年）。超级电脑 Cray 的设计者 Seymour Cray， 据说亲手设计 Cray 全部的硬体与其操作系统，作业系统是他用机器码硬干出来的，没有出过任何 bug 或 error。Real Programmer 真是超强!

举个比较不那么夸张的例子：Stan Kelly-Bootle，The Devil's DP Dictionary 一书的作者（McGraw-Hill， 1981 年初版，ISBN 0-07-034022-6）与 Hacker 传奇专家，当年在一台 Manchester Mark I 开发程序。 他现在是电脑杂志的专栏作家，写一些科学幽默小品，文笔生动有趣投今日 hackers 所好，所以很受欢迎。 其他人像 David E. Lundstorm，写了许多关于 Real Programmer 的小故事， 收录在 A few Good Men From UNIVAC 这本书，1987 年出版，ISBN-0- 262-62075-8。

§译：看到这里，大家应该能了解，所谓 Real Programmer 指的就是用组合语 言或甚至机器码，把程序用打卡机 punch 出一片片纸卡片，由主机读卡机输入电脑的那种石器时代 Programmer。

Real Programmer 的时代步入尾声，取而代之的是逐渐盛行的 Interactive computing，大学成立电算相关科系及电脑网络。它们催生了另一个持续的工程传统，并最终演化为今天的开放代码黑客文化。

1.2 早期的黑客

Hacker 时代的滥觞始于 1961 年 MIT 出现第一台电脑 DEC PDP-1。MIT 的 Tech Model Railroad Club（简称 TMRC）的 Power and Signals Group 买了这台机器后，把它当成最时髦的科技玩具，各种程序工具与电脑术语开始出现，整个环境与文化一直发展下去至今日。 这在 Steven Levy 的书“Hackers”前段有详细的记载（Anchor/Doubleday 公司，1984 年出版）

§译：Interactive computing 并非指 Windows、GUI、WYSIWYG 等介面， 当时有 terminal、有 shell 可以下指令就算是 Interactive computing 了。 最先使用 Hacker 这个字应该是 MIT。1980 年代早期学术界人工智慧的权威：MIT 的 Artificial Intelligence Laboratory，其核心人物皆来自 TMRC。从 1969 年 起，正好是 ARPANET 建置的第一年， 这群人在电脑科学界便不断有重大突破与贡献。

ARPANET 是第一个横跨美国的高速网络。由美国国防部所出资兴建，一个实验性质的数位通讯网络，逐渐成长成联系各大学、国防部承包商及研究机构的大网络。 各地研究人员能以史无前例的速度与弹性交流资讯，超高效率的合作模式导致科技的突飞猛进。

ARPANET 另一项好处是，资讯高速公路使得全世界的 hackers 能聚在一起，不再像以前孤立在各地形成一股股的短命文化，网络把他们汇流成一股强大力量。 开始有人感受到 Hacker 文化的存在，动手整理术语放上网络， 在网上发表讽刺文学与讨论 Hacker 所应有的道德规范。（Jargon File 的第一版出现在 1973 年，就是一个好例子），Hacker 文化在有接上 ARPANET 的各大学间快速发展，特别是（但不全是）在信息相关科系。

一开始，整个 Hacker 文化的发展以 MIT 的 AI Lab为中心，但 Stanford University 的 Artificial Intelligence Laboratory（简称 SAIL）与稍后的 Carnegie-Mellon University（简称 CMU）正快速崛起中。 三个都是大型的资讯科学研究中心及人工智慧的权威，聚集著世界各地的精英，不论在技术上或精神层次上，对Hacker文化都有极高的贡献。

为能了解后来的故事，我们得先看看电脑本身的变化；随著科技的进步，主角 MIT AI Lab 也从红极一时到最后淡出舞台。

从 MIT 那台 PDP-1 开始，Hacker 们主要程序开发平台都是 Digital Equipment Corporation 的 PDP 迷你电脑序列。DEC 率先发展出商业用途为主的 interactive computing 及 time-sharing 操作系统，当时许多的大学都是买 DEC 的机器，因为它兼具弹性与速度，还很便宜（相对于较快的大型电脑 mainframe）。便宜的分时系统是 Hacker 文化能快速成长因素之一，在 PDP 流行的时代，ARPANET 上是 DEC 机器的天下，其中最重要的便属 PDP-10，PDP-10 受到 Hacker 们的青睐达十五年；TOPS-10（DEC 的操作系统）与 MACRO-10（它的组译器），许多怀旧的术语及 Hacker 传奇中仍常出现这两个字。

MIT 像大家一样用 PDP-10，但他们不屑用 DEC 的操作系统。他们偏要自己写一个：传说中赫赫有名的 ITS。

ITS 全名是“Incompatible Timesharing System”，取这个怪名果然符合 MIT 的搞怪作风 —— 就是要与众不同，他们很臭屁但够本事自己去写一套操作系统。ITS 始终不稳，设计古怪，bug 也不少，但仍有许多独到的创见，似乎还是分时系统中开机时间最久的纪录保持者。

ITS 本身是用汇编语言写的，其他部分由 LISP 写成。LISP 在当时是一个威力强大与极具弹性的程序语言；事实上，二十五年后的今天，它的设计仍优于目前大多数的程序语言。LISP 让 ITS 的 Hacker 得以尽情发挥想像力与搞怪能力。LISP 是 MIT AI Lab 成功的最大功臣，现在它仍是 Hacker 们的最爱之一。

很多 ITS 的产物到现在仍活著；EMACS 大概是最有名的一个，而 ITS 的稗官野史仍为今日的 Hacker 们所津津乐道，就如同你在 Jargon File 中所读到的一般。在 MIT 红得发紫之际，SAIL 与 CMU 也没闲著。SAIL 的中坚份子后来成为 PC 界或图形使用者介面研发的要角。CMU 的 Hacker 则开发出第一个实用的大型专家系统与工业用机器人。

另一个 Hacker 重镇是 XEROX PARC 公司的 Palo Alto Research Center。从 1970 初期到 1980 中期这十几年间，PARC 不断出现惊人的突破与发明，不论质或量，软件或硬体方面。如现今的视窗滑鼠介面，雷射印表机与区域网络；其 D 系列的机器，催生了能与迷你电脑一较长短的强力个人电脑。不幸这群先知先觉者并不受到公司高层的赏识；PARC 是家专门提供好点子帮别人赚钱的公司成为众所皆知的大笑话。即使如此，PARC 这群人对 Hacker 文化仍有不可抹灭的贡献。1970 年代与 PDP-10 文化迅速成长茁壮。Mailing list 的出现使世界各地的人得以组成许多 SIG（Special-interest group），不只在电脑方面，也有社会与娱乐方面的。DARPA 对这些非“正当性”活动睁一只眼闭一只眼，因为靠这些活动会吸引更多的聪明小夥子们投入电脑领域呢。

有名的非电脑技术相关的 ARPANET mailing list 首推科幻小说迷的，时至今日 ARPANET 变成 Internet，愈来愈多的读者参与讨论。Mailing list 逐渐成为一种公众讨论的媒介，导致许多商业化上网服务如 CompuServe、Genie 与 Prodigy 的成立。

1.3 Unix 的兴起

此时在新泽西州的郊外，另一股神秘力量积极入侵 Hacker 社会，终于席卷整个 PDP-10 的传统。它诞生在 1969 年，也就是 ARPANET 成立的那一年，有个在 AT&T Bell Labs 的年轻小伙子 Ken Thompson 发明了 Unix。

Thomspon 曾经参与 Multics 的开发，Multics 是源自 ITS 的操作系统，用来实做当时一些较新的 OS 理论，如把操作系统较复杂的内部结构隐藏起来，提供一个介面，使的 programmer 能不用深入了解操作系统与硬体设备，也能快速开发程序。

§译：那时的 programmer 写个程序必须彻底了解操作系统内部，或硬体设备。比方说写有 IO 的程序，对于硬碟的转速，磁轨与磁头数量等等都要搞的一清二楚才行。

在发现继续开发 Multics 是做白工时，Bell Labs 很快的退出了（后来有一家公司 Honeywell 出售 Multics，赔的很惨）。Ken Thompson 很喜欢 Multics 上的作业环境，于是他在实验室里一台报废的 DEC PDP-7 上胡乱写了一个操作系统，该系统在设计上有从 Multics 抄来的也有他自己的构想。他将这个操作系统命名 Unix，用来反讽 Multics。

§译：其实是 Ken Thompson 写了一个游戏“Star Travel”没地方跑，就去找一台的报废机器 PDP-7 来玩。他同事 Brian Kernighan 嘲笑 Ken Thompson 说：“你写的系统好逊哦，干脆叫 Unics 算了。”（Unics 发音与太监的英文 eunuches 一样），后来才改为 Unix。

他的同事 Dennis Ritchie，发明了一个新的程序语言 C，于是他与 Thompson 用 C 把原来用汇编语言写的 Unix 重写一遍。C 的设计原则就是好用，自由与弹性，C 与 Unix 很快地在 Bell Labs 得到欢迎。1971 年 Thompson 与 Ritchie 争取到一个办公室自动化系统的专案，Unix 开始在 Bell Labs 中流行。不过 Thompson 与 Ritchie 的雄心壮志还不止于此。

那时的传统是，一个操作系统必须完全用汇编语言写成，始能让机器发挥最高的效能。Thompson 与 Ritchie，是头几位领悟硬体与编译器的技术，已经进步到作业系统可以完全用高阶语言如 C 来写，仍保有不错的效能。五年后，Unix 已经成功地移植到数种机器上。

§译：Ken Thompson 与 Dennis Ritchie 是唯一两位获得 Turing Award（电脑界的诺贝尔奖）的工程师（其他都是学者）。

这当时是一件不可思议的事！它意味著，如果 Unix 可以在各种平台上跑的话，Unix 软件就能移植到各种机器上。再也用不著为特定的机器写软件了，能在 Unix 上跑最重要，重新发明轮子已经成为过去式了。

除了跨平台的优点外，Unix 与 C 还有许多显著的优势。Unix 与 C 的设计哲学是“Keep It Simple, Stupid!”。programmer 可以轻易掌握整个C的逻辑结构（不像其他之前或以后的程序语言）而不用一天到晚翻手册写程序。而 Unix 提供许多有用的小工具程序，经过适当的组合（写成 Shell script 或 Perl script），可以发挥强大的威力。

§注：The C Programming Language 是所有程序语言书最薄的一本，只有两百多页哦。作者是 Brian Kernighan 与 Dennis Ritchie，所以这本 C 语言的圣经又称“K&R”。

§注：“Keep It Simple, Stupid!”简称 KISS，今日 Unix 已不 follow 这个原则，几乎所有 Unix 都是要灌一堆有的没的 utilities，唯一例外是 MINIX。

C 与 Unix 的应用范围之广，出乎原设计者之意料，很多领域的研究要用到电脑时，他们是最佳拍档。尽管缺乏一个正式支援的机构，它们仍在 AT&T 内部中疯狂的散播。到了 1980 年，已蔓延到大学与研究机构，还有数以千计的 hacker 想把 Unix 装在家里的机器上。

当时跑 Unix 的主力机器是 PDP–11、VAX 系列的机器。不过由于 UNIX 的高移植性，它几乎可安装在所有的电脑机型上。一旦新型机器上的 UNIX 安装好，把软件的 C 原始码抓来重新编译就一切 OK 了，谁还要用汇编语言来开发软件？ 有一套专为 UNIX 设计的网络 —— UUCP：一种低速、不稳但很成本低廉的网络。 两台 UNIX 机器用条电话线连起来，就可以使用互传电子邮件。UUCP 是内建在 UNIX 系统中的，不用另外安装。于是 UNIX 站台连成了专属的一套网络，形成其 Hacker 文化。在 1980 第一个 USENET 站台成立之后，组成了一个特大号的分散式布告栏系统，吸引而来的人数很快地超过了 ARPANET。

少数 UNIX 站台有连上 ARPANET。PDP–10 与 UNIX 的 Hacker 文化开始交流， 不过一开始不怎么愉快就是了。PDP–10 的 Hacker 们觉得 UNIX 的拥护者都是些什么也不懂的新手，比起他们那复杂华丽，令人爱不释手的 LISP 与 ITS，C 与 UNIX 简直原始的令人好笑。“一群穿兽皮拿石斧的野蛮人”他们咕哝著。

在这当时，又有另一股新潮流风行起来。第一部 PC 出现在 1975 年；苹果电脑在 1977 年成立，以飞快的速度成长。微电脑的潜力，立刻吸引了另一批年轻的 Hackers。他们最爱的程序语言是 BASIC，由于它过于简陋，PDP–10 的死忠派与 UNIX 迷们根本不屑用它，更看不起使用它的人。

§译：这群 Hacker 中有一位大家一定认识，他的名字叫 Bill Gates，最初就是他在 8080 上发展 BASIC compiler 的。

1.4 古老时代的终结

1980 年同时有三个 Hacker 文化在发展，尽管彼此偶有接触与交流，但还是各玩各的。ARPANET/PDP–10 文化， 玩的是 LISP、MACRO、TOPS–10 与 ITS。UNIX 与 C 的拥护者用电话线把他们的 PDP–11 与 VAX 机器串起来玩。 还有另一群散乱无秩序的微电脑迷，致力于将电脑科技平民化。

三者中 ITS 文化（也就是以 MIT AI LAB 为中心的 Hacker 文化）可说在此时达到全盛时期， 但乌云逐渐笼罩这个实验室。ITS 赖以维生的 PDP–10 逐渐过时，开始有人离开实验室去外面开公司，将人工智慧的科技商业化。MIT AI Lab 的高手挡不住新公司的高薪挖角而纷纷出走，SAIL 与 CMU 也遭遇到同样的问题。

§译：这个情况在 GNU 宣言中有详细的描述，请参阅：（特别感谢由 AKA 的 chuhaibo 翻成中文） http://www.aka.citf.net/Magazine/Gnu/manifesto.html

致命一击终于来临，1983 年 DEC 宣布：为了要集中在 PDP–11 与 VAX 生产线， 将停止生产 PDP–10；ITS 没搞头了，因为它无法移植到其他机器上，或说根本没人办的到。而 Berkeley Univeristy 修改过的 UNIX 在新型的 VAX 跑得很顺，是 ITS 理想的取代品。有远见的人都看得出，在快速成长的微电脑科技下，Unix 一统江湖是迟早的事。

差不多在此时 Steven Levy 完成“Hackers”这本书，主要的资料来源是 Richard M. Stallman（RMS）的故事，他是 MIT AI Lab 领袖人物，坚决反对实验室的研 究成果商业化。

Stallman 接著创办了 Free Software Foundation，全力投入写出高品质的自由软件。Levy 以哀悼的笔调描述他是“the last true hacker”，还好事实证明 Levy 完全错了。

§译：Richard M. Stallman 的相关事迹请参考: http://www.aka.citf.net/Magazine/Gnu/cover.htm

Stallman 的宏大计划可说是 80 年代早期 Hacker 文化的缩影 —— 在 1982 年他 开始建构一个与 UNIX 相容但全新的操作系统，以 C 来写并完全免费。整个 ITS 的精神与传统，经由 RMS 的努力，被整合在一个新的，UNIX 与 VAX 机器上的 Hacker 文化。 微电脑与区域网络的科技，开始对 Hacker 文化产生影响。Motorola 68000 CPU 加 Ethernet 是个有力的组合，也有几家公司相继成立生产第一代的工作站。 1982 年，一群 Berkeley 出来的 UNIX Hacker 成立了 Sun Microsystems，他们的算盘打的是：把 UNIX 架在以 68000 为 CPU 的机器，物美价廉又符合多数应用程序的要求。他们的高瞻远嘱为整个工业界树立了新的里程碑。虽然对个人而言，工作站仍太昂贵，不过在公司与学校眼中，工作站真是比迷你电脑便宜太多了。在这些机构里，工作站（几乎是一人一台）很快地取代了老旧庞大的 VAX 等 timesharing 机器。

§译：Sun 一开始生产的工作站 CPU 是用 Motorola 68000 系列，到 1989 才推出自行研发的以 SPARC 系列为 CPU 的 SPARC station。

1.5 私有 Unix 时代

1984 年 AT&T 解散了，UNIX 正式成为一个商品。当时的 Hacker 文化分成两大类，一类集中在 Internet 与 USENET 上（主要是跑 UNIX 的迷你电脑或工作站连上网络），以及另一类 PC 迷，他们绝大多数没有连上 Internet。

§译：台湾在 1992 年左右连上 Internet 前，玩家们主要以电话拨接 BBS 交换资讯，但是有区域性的限制， 发展性也大不如 USENET。Sun 与其他厂商制造的工作站为 Hacker 们开启了另一个美丽新世界。 工作站诉求的是高效能的绘图与网络，1980 年代 Hacker 们致力为工作站撰写软件，不断挑战及突破以求将这些功能发挥到百分之一百零一。Berkeley 发展出一套内建支援 ARPANET protocols 的 UNIX，让 UNIX 能轻松连上网络，Internet 也成长的更加迅速。

除了 Berkeley 让 UNIX 网络功能大幅提升外，尝试为工作站开发一套图形界面也不少。最有名的要算 MIT 开发的 Xwindow 了。Xwindow 成功的关键在完全公开原始码，展现出 Hacker 一贯作风，并散播到 Internet上。X 成功的干掉其他商业化的图形界面的例子，对数年后 UNIX 的发展有著深远的启发与影响。少数 ITS 死忠派仍在顽抗著，到 1990 年最后一台 ITS 也永远关机长眠了；那些死忠派在穷途末路下只有悻悻地投向 UNIX 的怀抱。

UNIX 们此时也分裂为 Berkeley UNIX 与 AT&T 两大阵营，也许你看过一些当时的海报，上面画著一台钛翼战机全速飞离一个爆炸中、上面印著 AT&T 的商标的死星。Berkeley UNIX 的拥护者自喻为冷酷无情的公司帝国的反抗军。 就销售量来说，AT&T UNIX 始终赶不上 BSD/Sun，但它赢了标准制订的战争。到 1990 年，AT&T 与 BSD 版本已难明显区分，因为彼此都有采用对方的新发明。随著 90 年代的来到，工作站的地位逐渐受到新型廉价的高档 PC 的威胁，他们主要是用 Intel 80386 系列 CPU。第一次 Hacker 能买一台威力等同于十年前的迷你电脑的机器，上面跑著一个完整的 UNIX，且能轻易的连上网络。沈浸在 MS-DOS 世界的井底蛙对这些巨变仍一无所知，从早期只有少数人对微电脑有兴趣，到此时玩 DOS 与 Mac 的人数已超过所谓的“网络民族”的文化，但他们始终没成什么气候或搞出什么飞机，虽然聊有佳作光芒乍现，却没有稳定发展出统一的文化传统，术语字典，传奇故事与神话般的历史。它们没有真正的网络，只能聚在小型的 BBS 站或一些失败的网络如 FIDONET。提供上网服务的公司如 CompuServe 或 Genie 生意日益兴隆，事实显示 non-UNIX 的操作系统因为并没有内附如 compiler 等程序发展工具，很少有 source 在网络上流传，也因此无法形成合作开发软件的风气。Hacker 文化的主力，是散布在 Internet 各地，几乎可说是玩 UNIX 的文化。他们玩电脑才不在乎什么售后服务之类，他们要的是更好的工具、更多的上网时间、还有一台便宜 32-bitPC。

机器有了，可以上网了，但软件去哪找？商业的 UNIX 贵的要命，一套要好几千大洋（$）。90 年代早期，开始有公司将 AT&T 与 BSD UNIX 移植到 PC 上出售。成功与否不论，价格并没有降下来，更要紧的是没有附原始码， 你根本不能也不准修改它，以符合自己的需要或拿去分享给别人。传统的商业软件并没有给 Hacker 们真正想要的。

即使是 Free Software Foundation（FSF）也没有写出 Hacker 想要的操作系统，RMS 承诺的 GNU 操作系统 —— HURD 说了好久了，到 1996 年都没看到影子（虽然 1990 年开始，FSF 的软件已经可以在所有的 UNIX 平台执行）。

1.6 早期的免费 Unix

在这空窗期中，1992 年一位芬兰 Helsinki University 的学生 —— Linus Torvalds 开始在一台 386 PC 上发展一个自由软件的 UNIX kernel，使用 FSF 的程序开发工具。

他很快的写好简单的版本，丢到网络上分享给大家，吸引了非常多的 Hacker 来帮忙一起发展 Linux —— 一个功能完整的 UNIX，完全免费且附上全部的原始码。 Linux 最大的特色，不是功能上的先进而是全新的软件开发模式。直到 Linux 的成功前，人人都认为像操作系统这么复杂的软件，非得要靠一个开发团队密切合作，互相协调与分工才有可能写的出来。商业软件公司与 80 年代的 Free Software Foundation 所采用都是这种发展模式。

Linux 则迥异于前者。一开始它就是一大群 Hacker 在网络上一起涂涂抹抹出来的。 没有严格品质控制与高层决策发展方针，靠的是每周发表新版供大家下载测试，测试者再把 bug 与 patch 贴到网络上改进下一版。一种全新的物竞天择、去芜存菁的快速发展模式。令大夥傻眼的是，东修西改出来的 Linux，跑的顺极了。

1993 年底，Linux 发展趋于成熟稳定，能与商业的 UNIX 一分高下，渐渐有商业应用软件移植到 Linux 上。不过小型 UNIX 厂商也因为 Linux 的出现而关门大吉，因为再没有人要买他们的东西。幸存者都是靠提供 BSD 为基础的 UNIX 的完整原始码，有 Hacker 加入发展才能继续生存。

Hacker 文化，一次次被人预测即将毁灭，却在商业软件充斥的世界中，披荆斩棘，筚路蓝缕，开创出另一番自己的天地。

1.7 网络大爆炸时代

Linux 能快速成长的来自令一个事实：Internet 大受欢迎，90 年代早期 ISP 如雨后春笋般的冒出来， World-WideWeb 的出现，使得 Internet 成长的速度，快到有令人窒息的感觉。

BSD 专案在 1994 正式宣布结束，Hacker 们用的主要是免费的 UNIX（Linux与一些 4.4 BSD 的衍生版本）。而 Linux CD-ROM 销路非常好（好到像卖煎饼般）。近几年来 Hacker 们主要活跃在 Linux 与 Internet 发展上。World Wide Web 让 Internet 成为世界最大的传输媒体，很多 80 年代与 90 年代早期的 Hacker 们现在都在经营 ISP。

Internet 的盛行，Hacker 文化受到重视并发挥其政治影响力。94、95 年美国政府打算把一些较安全、难解的编码学加以监控，不容许外流与使用。这个称为 Clipper proposal 的专案引起了 Hacker 们的群起反对与强烈抗议而半途夭折。96 年 Hacker 又发起了另一项抗议运动对付那取名不当的"Communications DecencyAct"，誓言维护 Internet 上的言论自由。

电脑与 Internet 在 21 世纪将是大家不可或缺的生活用品，现代孩子在使用 Internet 科技迟早会接触到 Hacker 文化。它的故事传奇与哲学，将吸引更多人投入。未来对 Hacker 们是充满光明的。

2 大教堂和市集

2.1 大教堂和市集

Linux 的影响是非常巨大的。甚至在 5 年以前，有谁能够想象一个世界级的操作系统能够仅仅用细细的 Internet 连接起来的散布在全球的几千个开发人员有以业余时间来创造呢?

我当然不会这么想。在 1993 年早期我开始注意 Linux 时，我已经参与 Unix 和自由软件开发达十年之久了。我是八十年代中期 GNU 最早的几个参与者之一。我已经在网上发布了大量的自由软件，开发和协助开发了几个至今仍在广泛使用的程序（Nethack, Emacs VC 和 GND 模式，xlife 等等）。我想我知道该怎样做。

Linux 推翻了许多我认为自己明白的事情。我已经宣扬小工具、快速原型和演进式开发的 Unix 福音多年了。但是我也相信某些重要的复杂的事情需要更集中化的，严密的方法。我相信多数重要的软件（操作系统和象 Emacs 一样的真正大型的工具）需要向建造大教堂一样来开发，需要一群于世隔绝的奇才的细心工作，在成功之前没有 beta 版的发布。

Linus Torvalds 的开发风格（尽早尽多的发布，委托所有可以委托的事，对所有的改动和融合开放）令人惊奇地降临了。这里没有安静的、虔诚的大教堂的建造工作 —— 相反，Linux 团体看起来像一个巨大的有各种不同议程和方法的乱哄哄的集市（Linux 归档站点接受任何人的建议和作品，并聪明的加以管理），一个一致而稳定的系统就像奇迹一般从这个集市中产生了。

这种设计风格确实能工作，并且工作得很好，这个事实确实是一个冲击。在我的研究过程中，我不仅在单个工程中努力工作，而且试图理解为什么 Linux 世界不仅没有在一片混乱中分崩离析，反而以大教堂建造者们不可想象的速度变得越来越强大。

到了 1996 年中，我想我开始理解了。我有一个极好的测试我的理论的机会，以一个自由软件计划的形式，我有意识地用了市集风格。我这样做了，并取得了很大的成功。

在本文的余下部分，我将讲述这个计划的故事，我用它来明确一些自由软件高效开发的格言。并不是所有这些都是从 Linux 世界中学到的，但我们将看到 Linux 世界给予了它们一个什么样的位置。如果我是正确的，它们将使你理解是什么使 Linux 团体成为好软件的源泉，帮助你变得更加高效。

2.2 邮件必须得通过

1993 年以前我在一个小的免费访问的名为 Chester County InterLink 的 ISP 的做技术工作，它位于 Pennsylvania 的 West Chester。（我协助建立了 CCIL，并写了我们独特的多用户 BBS 系统 —— 你可以 telnet 到 locke.ccil.org 来检测一下。今天它在十九条线上支持三千的用户）。这个工作使我可以一天二十四小时通过 CCIL 的 56K 专线连在网上，实际上，它要求我怎么做!

所以，我对 Internet email 很熟悉。因为复杂的原因，很难在我家里的机器（snark.thyrsus.com）和 CCIL 之间用 SLIP 工作。最后我终于成功了，但我发现不得不时常 telnet 到 locke 来检查我的邮件，这真是太烦了。我所需要的是我的邮件发送到 snark，这样 biff(1) 会在它到达时通知我。

简单地 sendmail 的转送功能是不够的，因为 snark 并不是总在网上而且没有一个静态地址。我需要一个程序通过我的 SLIP 连接把我的本地发送的邮件拉过来。我知道这种东西是存在的，它们大多使用一个简单的协议 POP（Post Office Protocol）。而且，locke 的 BSD/OS 操作系统已经自带了一个 POP3 服务器。

我需要一个 POP3 客户。所以我到网上去找到了一个。实际上，我发现了三、四个。我用了一会 pop-perl，但它却少一个明显的特征：抽取收到的邮件的地址以便正确回复。

问题是这样的：假设 locke 上一个叫“joe”的人向我发了一封邮件。如果我把它取到 snark 上准备回复时,我的邮件程序会很高兴地把它发送给一个不存在的 snark 上的“joe”。手工的在地址上加上“@ccil.org”变成了一个严酷的痛苦。

这显然应是计算机替我做的事。（实际上，依据 RFC1123 的 5.2.18 节，sendmail 应该做这件事）。但是没有一个现存的 POP 客户知道怎样做！于是这就给我们上了第一课：

1. 每个好的软件工作都开始于搔到了开发者本人的痒处

也许这应该是显而易见的（“需要是发明之母”长久以来就被证明是正确的），但是软件开发人员常常把他们的精力放在它们既不需要也不喜欢的程序，但在 Linux 世界中却不是这样 —— 这解释了为什么从 Linux 团体中产生的软件质量都如此之高。

那么，我是否立即投入疯狂的工作中，要编出一个新的 POP3 客户与现存的那些竞争呢？才不是哪！我仔细考察了手头上的 POP 工具，问自己“那一个最接近我的需要？”因为：

2. 好程序员知道该写什么，伟大的程序员知道该重写（和重用）什么。

我并没有声称自己是一个伟大的程序员，可是我试着效仿他们。伟大程序员的一个重要特点是建设性的懒惰。他们知道你是因为成绩而不是努力得到奖赏，而且从一个好的实际的解决方案开始总是要比从头干起容易。

例如，Linux 并不是从头开始写 Linux 的。相反的它从重用 Minix（一个 386 机型上的类似 Unix 的微型操作系统）的代码和思想入手。最后所有的 Minix 代码都消失或被彻底的重写了，但是当它们在的时候它为最终成为 Linux 的雏形做了铺垫。

秉承同样的精神，我去寻找良好编码的现成的 POP 工具，用来作为基础。

Unix 世界中的代码共享传统一直对代码重用很友好（这正是为什么 GNU 计划不管 Unix 本身有多么保守而选取它作为基础操作系统的原因）。Linux 世界把这个传统推向技术极限：它有几个 T 字节的源代码可以用。所以在 Linux 世界中花时间寻找其他几乎足够好的东西，会比在别处带来更好的结果。

这也适合我。加上我先前发现的，第二次寻找找到了 9 个候选者——fetchPOP，PopTart，get-mail，gwpop，pimp，pop-perl，popc，popmail 和 upop。我首先选定的是“fetchpop”。我加入了头标重写功能，并且做了一些被作者加入他的 1.9 版中的改进。

但是几个星期之后，我偶然发现了 Carl Harris 写的“popclient”的代码，然后发现有个问题，虽然 fetchpop 有一些好的原始思想（比如它的守护进程模式），它只能处理 pop3，而且编码的水平相当业余（Seung-Hong 是个很聪明但是经验不足的程序员），Carl 的代码更好一些，相当专业和稳固，但他的程序缺少几个重要的相当容易实现的 fetchpop 的特征（包括我自己写的那些）。

继续呢还是换一个？如果换一个的话，作为得到一个更好开发基础的代价，我就要扔掉我已经有的那些代码。

换一个的一个实际的动机是支持多协议，POP3 是用的最广的邮局协议，但并非唯一一个，Fetchpop 和其余几个没有实现 POP2.RPOP，或者 APOP，而且我还有一个为了兴趣加入 IMAP（Internet Message Access Protocol，最近设计的最强大的邮局协议）的模糊想法。

但是我有一个更加理论化的原因认为换一下会是一个好主意，这是我在 Linux 很久以前学到的：

3. “计划好抛弃，无论如何，你会的”（Fred Brooks,《人月神话》第 11 章）

或者换句话说，你常常在第一次实现一个解决方案之后才能理解问题所在，第二次你也许才足够清楚怎样做好它，因此如果你想做好，准备好推翻重来至少一次。

好吧（我告诉自己），对 fetchpop 的尝试是我第一次的尝试，因此我换了一下。

当我在 1996 年 6 月 25 日把我第一套 popclient 的补丁程序寄给 Carl Harris 之后，我发现一段时间以前他已经对 popclient 基本上失去了兴趣，这些代码有些陈旧，有一些次要的错误，我有许多修改要做，我们很快达成一致，我来接手这个程序。不知不觉的，这个计划扩大了，再也不是我原先打算的在已有的 pop 客户上加几个次要的补丁而已了，我得维护整个的工程，而且我脑袋里涌动着一些念头要引起一个大的变化。

在一个鼓励代码共享的软件文化里，这是一个工程进化的自然道路，我要指出：

4. 如果你有正确的态度，有趣的问题会找上你的，但是 Carl Harris 的态度甚至更加重要，他理解：

5. 当你对一个程序失去兴趣时，你最后的责任就是把它传给一个能干的后继者。

甚至没有商量，我和 Carl 知道我们有一个共同目标就是找到最好的解决方案，对我们来说唯一的问题是我能否证明我有一双坚强的手，他优雅而快速的写出了程序，我希望轮到我时我也能做到。

2.3 拥有用户的重要性

于是我继承了 popclient，同样重要的是，我继承了 popclient 的用户基础，用户是你所拥有的极好的东西，不仅仅是因为他们显示了你正在满足需要，你做了正确的事情，如果加以适当的培养，他们可以成为合作开发者。

Unix 传统另一有力之处是许多用户都是黑客，因为源优码是公开的，他们可以成为高效的黑客，这一点在 Linux 世界中也被推向了令人高兴的极致，这对缩短调试时间是极端重要的，在一点鼓励之下，你的用户会诊断问题，提出修订建议，帮你以远比你期望快得多的速度的改进代码。

6. 把用户当做协作开发者是快速改进代码和高效调试的无可争辩的方式。

这种效果的力量很容易被低估，实际上，几乎所有我们自由软件世界中的人都强烈低估了用户可以多么有效地对付系统复杂性，直到 Linus 让我们看到了这一点。

实际上，我认为 Linus 最聪明最了不起的工作不是创建了 Linux 内核本身，而是发明了 Linux 开发模式，当我有一次当着他的面表达这种观点时，他微笑了一下，重复了一句他经常说的话：“我基本上是一个懒惰的人，依靠他人的工作来获取成绩。”像狐狸一样懒惰，或者如 Robert Heinlein 所说，太懒了而不会失败。

回顾起来，在 GNU Emacs Lisp 库和 Lisp 代码集中可以看到 Linux 方法的成功，与 Emacs 的 C 内核和许多其他 FSF 的工具相比，Lisp 代码库的演化是流动性的和用户驱动的，思想和原型在达到最终的稳定形式之前往往要重写三或四次，而且经常利用 Internet 的松散合作。

实际上，我自己在 fetchmail 之前最成功的作品要算 Emacs VC 模式，它是三个其他的人通过电子邮件进行的类似 Linux 的合作，至今我只见过其中一个人（Richard Stallman），它是 SCCS、RCS 和后来的 CVS 的前端，为 Emacs 提供“one-touch”版本控制操作，它是从一个微型的、粗糙的别人写好的 sccs.el 模式开始演化的，VC 开发的成功不像 Emacs 本身，而是因为 Emacs Lisp 代码可以很快的通过发布/测试/改进的过程。

（FSF 的试图把代码放入 GPL 之下的策略有一个未曾预料到的副作用，它让 FSF 难以采取市集模式，因为他们认为每个想贡献二十行以上代码的人都必须得到一个授权，以使受到 GPL 的代码免受版权法的侵扰，具有 BSD 和 MITX 协会的授权的用户不会有这个问题，因为他们并不试图保留那些会使人可能受到质询的权力）。

2.4 早发布、常发布

尽量早尽量频繁的发布是 Linux 开发模式的一个重要部分，多数开发人员（包括我）过去都相信这对大型工程来说是个不好的策略，因为早期版本都是些充满错误的版本，而你不想耗光用户的耐心。

这种信仰强化了建造大教堂开发方式的必要性，如果目标是让用户尽可能少的见到错误，那你怎能不会仅仅每六个月发布一次（或更不经常），而且在发布之间象一只狗一样辛勤“捉虫”呢? Emacs C 内核就是以这种方式开发的，Lisp 库，实际上却相反，因为有一些有 FSF 控制之外的 Lisp 库，在那里你可以独立于 Emacs 发布周期地找寻新的和开发代码版本。

这其中最重要的是 Ohio 州的 elisp 库，预示了今天的巨大的 Linux 库的许多特征的精神，但是我们很少真正仔细考虑我们在做什么，或者这个库的存在指出了 FSF 建造教堂式开发模式的什么问题，1992 年我曾经做了一次严肃的尝试，想把 Ohio 的大量代码正式合并到 Emacs 的官方 Lisp 库中，结果我陷入了政治斗争中，彻底失败了。

但是一年之后，在 Linux 广泛应用之后，很清楚，一些不同的更加健康的东西诞生了，Linus 的开发模式正好与建造教堂方式相反，Sunsite 和 tsx-11 的库开始成长，推动了许多发布。所有这些都是闻所未闻的频繁的内核系统的发布所推动的。

Linus 以所有实际可能的方式把它的用户作为协作开发人员。

7. 早发布、常发布、听取客户的建议

Linus 的创新并不是这个（这在 Unix 世界中是一个长期传统），而是把它扩展到和他所开发的东西的复杂程度相匹配的地步，在早期一天一次发布对他来说都不是罕见的！而且因为他培育了他的协作开发者基础，比其他任何人更努力地充分利用了 Internet 进行合作，所以这确实能行。

但是它是怎样进行的呢？它是我能模仿的吗？还是这依赖于 Linus 的独特天才？

我不这样想，我承认 Linus 是一个极好的黑客（我们有多少人能够做出一个完整的高质量的操作系统内核？），但是 Linux 并不是一个令人敬畏的概念上的飞跃，Linus 不是（至少还不曾是）像 Richard stallman 或 James Gosling 一样的创新天才，在我看来，Linus 更象一个工程天才，具有避免错误和开发失败的第六感觉，掌握了发现从 A 点到 B 点代价最小的路径的决窍，确实，Linux 的整个设计受益于这个特质，并反映出 Linus 的本质上保守和简化设计的方法。

如果快速的发布和充分利用 Internet 不是偶然而是 Linus 的对代价最小的路径的洞察力的工程天才的内在部分，那么他极大增强了什么？他创建了什么样的方法？

问题回答了它自己，Linus 保持他的黑客用户经常受到激励和奖赏：被行动的自我满足的希望所激励，而奖赏则是经常(甚至每天）都看到工作在进步。

Linus 直接瞄准了争取最多的投入调试和开发的人时，甚至冒代码不稳定和一旦有非常棘手的错误而失去用户基础的险，Linus 似乎相信下面这个：

8. 如果有一个足够大的 beta 测试人员和协作开发人员的基础，几乎所有的问题都可以被快速地找出并被一些人纠正。

或者更不正式的讲：“如果有足够多的眼睛，所有的错误都是浅显的”（群众的眼睛是雪亮的），我把这称为“Linus 定律”。

我最初的表述是每个问题“对某些人是透明的”，Linus 反对说，理解和修订问题的那个人不一定非是甚至往往不是首先发现它的人，“某个人发现了问题”，他说，“另一个理解它，我认为发现它是个更大的挑战”，但是要点是所有事都趋向于迅速发生。

我认为这是建造教堂和集市模式的核心区别，在建造教堂模式的编程模式看来，错误和编程问题是狡猾的、阴险的、隐藏很深的现象，花费几个月的仔细检查，也不能给你多大确保把它们都挑出来的信心，因此很长的发布周期，和在长期等待之后并没有得到完美的版本发布所引起的失望都是不可避免的。

以市集模式观点来看，在另一方面，我们认为错误是浅显的现象，或者至少当暴露给上千个热切的协作开发人员，让他们来对每个新发布进行测试的时候，它们很快变得浅显了，所以我们经常发布来获得更多的更正，作为一个有益的副作用，如果你偶尔做了一个笨拙的修改，也不会损失太多。也许我们本不应该这样的惊奇，社会学家在几年前已经发现一群相同专业的（或相同无知的）观察者的平均观点比在其中随机挑选一个来得更加可靠，他们称此为“Delhpi 效应”，Linus 所显示的证明在调试一个操作系统时它也适用 —— Delphi 效应甚至可以战胜操作系统内核一级的复杂度。

我受 Jeff Dutky（dutky@wam.umd.edu）的启发指出 Linus 定律可以重新表述为“调试可以并行”，Jeff 观察到虽然调试工作需要调试人员和对应的开发人员相交流，但它不需要在调试人员之间进行大量的协调，于是它就没有陷入开发时遇到的平方复杂度和管理开销。

在实际中，由于重复劳动而导致的理论上的丧失效率的现象在 Linux 世界中并不是一个大问题，“早发布、常发布策略”的一个效果就是利用快速的传播反馈修订来使重复劳动达到最小。

Brooks 甚至做了一个与 Jeff 相关的更精确的观察：“维护一个广泛使用的程序的成本一般是其开发成本的 40%，奇怪的是这个成本受到用户个数的强烈影响，更多的用户发现更多的错误”（我的强调）。

更多的用户发现更多的错误是因为更多的用户提供了更多测试程序的方法，当用户是协作开发人员时这个效果被放大了，每个找寻错误的人都有自己稍微不同的感觉和分析工具，从不同角度来看待问题。“Delphi 效应”似乎因为这个变体工作变得更加精确，在调试的情况下，这个变体同时减小了重复劳动。

所以加入更多的 beta 测试人员虽不能从开发人员的 P.O.V 中减小“最深”的错误的复杂度，但是它增加了这样一种可能性，即某个人的工具和问题正好匹配，而这个错误对这个人来说是浅显的。

Linus 也做了一些改进，如果有一些严重的错误，Linux 内核的版本在编号上做了些处理，让用户可以自己选择是运行上一个“稳定”的版本，还是冒遇到错误的险而得到新特征，这个战略还没被大多数 Linux 黑客所仿效，但它应该被仿效，存在两个选择的事实让二者都很吸引人。

2.5 什么时候玫瑰不是玫瑰?

在研究了 Linus 的行为和形成了为什么它成功的理论之后，我决定在我的工程（显然没有那么复杂和雄心勃勃）里有意识的测试这个理论。

但我首先做的事是熟悉和简化 Popclient。Carl Harris 的实现非常好，但是有一种对许多 C 程序来说没有必要的复杂性。他把代码当作核心而把数据结构当作对代码的支持，结果是代码非常漂亮但是数据结构设计得很特别，相当丑陋（至少对以这个老 LISP 黑客的标准来看），然而除了提高代码和数据结构设计之外，重写它还有一个目的，就是要把它演化为我彻底理解的东西，对修改你不理解的程序中的错误负责可不是一件有趣的事。

第一个月我只是在领会 Carl 的基本设计的含义，我所做的第一个重大修改是加入了 IMAP 支持，我把协议机重新组织为一个通用驱动程序和三个方法表（对应 POP2、POP3 和 IMAP），这个前面的修改指出一个需要程序员（特别是象 C 这种没有自然的动态类型支持的语言）记在脑中的一般原理：

9. 聪明的数据结构和笨拙的代码要比相反的搭配工作的更好

Fred Brooks 也在他第 11 章中讲道：“让我看你的‘代码’，把你的‘数据结构’隐藏起来，我还是会迷惑；让我看看你的‘数据结构’，那我就不需要你的‘代码’了，它是显而易见的”。

实际上，他说的是“流程图”和“表”，但是在三十年的术语/文化演进之后，事情还是一样的。

此时（1996 年 9 月初，在从零开始六个月后），我开始想接下来修改名字 —— 毕竟，它已不仅仅是一个 POP 客户，但我犹豫了，因为还没有什么新的漂亮设计呢，我的 popclient 版本需要有自己的特色。

当 fetehmail 学会怎样把取到的邮件转送到 SMTP 端口时，事情就完全改变了，但是首先：上面我说过我决定使用这个工程来测试我关于 Linus Torualds 所做的行为的理论，（你可能会问）我怎样做到这点呢? 以下面的方式：

1. 我尽早尽量频繁地发布（几乎从未少于每十天发布一次；在密集开发的时候是每天一次）。
2. 我把每一个和我讨论 fetchmail 的人加入一个 beta 表中。
3. 每当我发布我都向 beta 表中的人发出通告，鼓励人们参与。
4. 我听取 beta 测试员的意见，向他们询问设计决策，对他们寄来的补丁和反馈表示感谢。

这些简单的手段立即收到的回报，在工程的开始，我收到了一些错误报告，其质量足以使开发者因此被杀掉，而且经常还附有补丁、我得到了理智的批评，有趣的邮件，和聪明的特征建议，这导致了：

10. 如果你象对待最宝贵的资源一样对待你的 beta 测试员，他们就会成为你最宝贵的资源。

2.6 popclient 变成了 Fetchmail

这个工程的真正转折点是 Harry Hochleiser 寄给我他写的代码草稿，他把邮件转发到客户端机器的 SMTP 端口，我立即意识到这个特征的可靠实现将淘汰所有其他的递送模式。

几个星期以来我一直在修改而不是改进 fetchmail，因为我觉得界面设计虽然有用但是太笨拙琐碎了，到处充满了太多的粗陋的细小选项。

当我思考 SMTP 转发时我发现 popclient 试图做的事太多了，它被设计成既是一个邮件传输代理（MTA）也是一个本地递送代理（MDA）。使用 SMTP 转发，它就可以从 MDA 的事务中解脱出来而成为一个纯 MTA，而象 sendmail 一样把邮件交给本地递送程序来处理。

既然端口 25 在所有支撑 TCP/IP 的平台上早已被预留，为什么还要为一个邮件传输代理的配置或为一个邮箱设置加锁的附加功能而操心呢？尤其是当这意味着抽取的邮件就像一个正常的发送者发出的 SMTP 邮件一样，而这就是我们需要的。

这里有几个教益：第一，SMTP 转发的想法是我有意识地模拟 Linus 的方法以来的最大的单个回报，一个用户告诉我这个非同寻常的想法 —— 我所需做的只是理解它的含义。

11. 想出好主意是好事，从你的用户那里发现好主意也是好事，有时候后者更好。

很有趣的是，你很快将发现，如果你完全承认你从其他人那里得到多少教益的话，整个世界将会认为所有的发明都是你做出的，而你会对你的天才变得谦虚。我们可以看到这在 Linus 身上体现得多明显！（当我在 1997 年 8 月的 Perl 会议上发表这个论文时，Larry Wall 坐在前排，当我讲到上面的观点时，他激动的叫了出来：“对了！说对了！哥们！”所有的听众都哄堂大笑起来，因为他们知道同样的事情也发生在 Perl 的发明者身上）。

于是在同样精神指导下工程进行了几个星期，我开始不光从我的用户那儿也从听说我的系统的人那儿得到类似的赞扬，我把一些这种邮件收藏起来，我将在我开始怀疑自己的生命是否有价值时重新读读这些信。:)

但是有两个更基本的，非政治性的对所有设计都有普遍意义的教益。

12. 最重要和最有创新的解决方案常常来自于你认识到你对问题的概念是错误的。

一个衡量 fetchmail 成功的有趣方式是工程的 beta 测试人员表（fetchmail 的朋友们）的长度，在创立它的时候已经有 249 个成员了，而且每个星期增加两到三个。

实际上，当我在 1997 年 5 月校订它时，这张表开始因为一个有趣的原因而缩短了，有几个人请求我把他们从表中去掉，因为 fetchmail 已经工作的如此之好，他们不需要看到这些邮件了！也许这是一个成熟的市集风格工程的生命周期的一部分。

我以前一直在解决错误的问题，把 popclient 当作 MTA 和具有许多本地递送模式的 MDA 的结合物，Fetchmail 的设计需要从头考虑为一个纯的 MTA，做为一个普通 Internet 邮件路径的一部分。

当你在开发中碰了壁时（当你发现自己很难想通下一步时），那通常不是要问自己是否找到正确答案，而是要问是否问了正确问题，也许需要重新构造问题。

于是，我重新构造了我的问题，很清楚，要做的正确的事是 (1) 把 SMTP 转发支持放在通用驱动程序中，(2) 把它做为缺省模式，(3) 最终分离所有其他的递送模式，尤其是递送到文件和标准输出的选项。

我在第三步上犹豫了一下，担心会让 popclient 的长期用户对新的递送方法感到烦心，在理论上，他们可以立即转而转发文件或者他们的非 sendmail 等价物来得到同样的效果，在实际中这种转换可能会很麻烦。

但是当我这么做之后，证明好处是巨大的，驱动程序代码的冗余的部分消失了，配置完全变得简单了——不用屈从于系统 MDA 和用户的邮箱，也不用为下层 OS 是否支持文件锁定而担心了。

而且，丢失邮件的唯一漏洞也被堵死了，如果你选择了递送到一个文件而磁盘已满，你的邮件就会丢失，这在 SMTP 转发中不会发生，因为 SMTP 侦听器不会返回 OK 的，除非邮件可以递送成功或至少被缓冲留待以后递送。

还有，性能也改善了（虽然在单次执行中你不会注意到），这个修改的另一个不可忽视的好处是手册变得大大简单了。

后来，为了允许处理一些罕见的情况，包括动态 SLIP，我必须回到让用户定义本地 MDA 递送上来，但是我发现了一个更加简单的方法。

所有这些给了我们什么启发呢?如果可以不损失效率，就要毫不犹豫抛弃陈旧的特性，Antonine de SaintExupery（在他成为经典儿童书籍作家之前是一个飞行员和飞机设计师）曾说过：

13. “最好的设计不是再也没有什么东西可以添加了，而是再也没有什么东西可以去掉。”

当你的代码变得更好和更简单时，这就是你知道它是正确的时候了，而且在这个过程中，fetchmail 的设计具有了自己的特点，而区别于其前身 popclient。

现在是改名的时候了，这个新的设计看起来比老 popclient 更象一个 sendmail 的复制品，它们都是 MTA，但是 senmail 是推然后递送，而新的 popclient 是拉然后递送。于是，在两个月之后，我把它重新命名为 fetchmail。

2.7 Fetchmail 成长起来

现在我有了一个简洁和富有创意的设计，工作得很好的代码，因为我每天都用它，和一直在增长的 beta 表，它让我渐渐明白我已经不是在从事只能对少数其他人有用的工作中，我写了一个所有有一个 Unix 邮箱和 SLIP/PPP 邮件连接的人都真正需要的程序。

通过 SMTP 转发功能，它成为一个潜在的“目录杀手”，远远领先于它的竞争者，这个程序如此能干以至于其他的程序不但被放弃简直被忘记了。

我知道你不可以真得瞄准或计划出这样的结果，你只能努力去设计这些强大的思想，以后这些结果就好象是不可避免的、自然的、注定了的，得到这种思想的唯一办法是获取许多思想，或者用工程化地思考其他人的好主意而超过原来想到它的人的设想。

Andrew Tanenbanm 原来设想建造一个适合 386 的简单的 Unix 用做教学，Linus Torvalels 把 Andrew 的可能想到的 Minix 可以做什么的概念推进了一步，成长为一个极好的东西，同样的（虽然规模较小），我接受了 Card Harris 和 Harry Hochheiser 的想法，把它们变得更强大，我们都不是人们所浪漫幻想的天才的创始人，但是大多数科学、工程和软件开发不是被天才的创始人完成的，这和流传的神话恰恰相反。

结果总是执着的原因 —— 实际上，它是每个黑客为之生存的成功！而且它们意味着我必须把自己的标准定高一点，为了把 fetchmail 变得和我所能设想的那样好，我必须不仅为我自己的需要写代码，而且也要包括对在我生活圈外的人们的需求的支持，而且同时也要保证程序的简单和健壮。

在实现它之后我首先写的最重要的特征是支持多投 —— 从集中一组用户的邮件的邮箱中取出邮件，然后把它路由到每个人手中。

我之所以加上多投功能部分是因为有些用户一直在闹着要它，更是因为我想它可以从单投的代码中揭露出错误来，让我完全一般地处理寻址，而且这被证明了。正确解释 RFC822 花了我相当长的时间，不仅因为它的每个单独部分都很难，而且因为它有一大堆相互依赖的苛刻的细节。

但是多投寻址也成为一个极好的设计决策，由此我知道：

14. 任何工具都应该能以预想的方式使用，但是一个伟大的工具提供你没料到的功能。

Fetchmail 多投功能的一个没有料到的用途是在 SLIP/PPP 的客户端提供邮件列表、别名扩展。这意味着一个使用个人机器的人不必持续访问 ISP 的别名文件就能通过一个 ISP 帐户管理一个邮件列表。我的 beta 测试员提出的另一个重要的改变是支持 8 位 MIME 操作，这很容易做，因为我已经仔细的保证了 8 位代码的清晰，不仅因为我预见到了这个特性的需求，而且因为我忠实于另一准则：

15. 当写任何种类的网关型程序时，多费点力，尽量少干扰数据流，永远不要抛弃信息，除非接收方强迫这么做!

如果我不遵从这个准则，那么 8 位 MIME 支持将会变得困难和笨拙，现在我所需要做的，是只读一下 RFC 1652，在产生信头的逻辑加上一点而已。

一些欧洲用户要求我加上一个选项来限制每次会话取得消息数（这样他们就可以从昂贵的电话网中控制花费了），我很长一段时间拒绝这样做，而且我仍然对它不很高兴，但是如果你是为了世界而写代码，你必须听取顾客的意见 —— 这并不随他们不付给你钱而改变。

2.8 从 Fetchmail 得来的另一些教益

在他们回到一般的软件工程问题以前，还有几个从 fetchmail 得到的教益需要思考。

rc 文件语法包括可选的“noise”关键字，它被扫描器完全忽略了，当你把它们全抽取出的时候，关键字/值对更具可读性。

当我注意到 rc 文件的声明在多大程度上开始象一个微型命令语言时，这是一个 Late-night 的体验（这也是我为什么把 popclient 原来的“server”关键字改成了“poll”）。

对我来说似乎把这个微型命令语言变得更象英语可能会使它更容易使用。现在，虽然我对经过 Emacs 和 HTML 及许多数据库引擎所证实的“把它做成一个语言”的设计方式确信不疑，但是我并不是一个通常的“类英语”语法的狂热拥护者。

传统程序员容易控制语法使它尽量精确和紧凑，完全没有冗余，这是计算机资源还很昂贵时遗留下的一种文化传统，所以扫描策略需要尽可能的廉价和简单，而具有 50% 冗余度的英语，看来好象是一个非常不合适的模型。

这并不是我不用类英语语法的原因，我提到这一点是为了推翻它，在更廉价的时钟周期与核心的时代，简洁并没有走到尽头，今天对一个语言来说，对人更方便比对机器更廉价来的更加重要。

然而，有几个原因提醒我们小心一点，一个是扫描策略的复杂度开销 —— 你并不想把它变成一个巨大的错误来源和让用户困惑，另一个是试图使语言表面上的类似可以和传统语言一样令人困惑（你可以在许多 4GL 和商业数据库查询语言上看到这一点）。

Fetchmail 的控制语法避免了这些问题，因为语言的领域是极其有限的。它一点也不象一个一般性的语言，它很简单地描述的东西并不复杂，所以很少可能在英语的一个小子集与实际的控制语言之间发生混淆，我想这有一个更广泛的教益：

16. 如果你的语言一点也不像是图灵完备的，严格的语法会有好处。

另一个教益是关于安全的，一些 fetchmail 用户要求我修改软件把口令加密存贮在 rc 文件里，这样觑探者就不能看到它们了。

我没有这样做，因为这实际上起不到任何保护作用，任何有权读取你的 rc 文件的人都可以以你的名义运行 fetchmail —— 如果他们要破你的口令，它们可以从 fetchmail 的代码中找到制作解码器的方法。

所以 fetchmail 口令的加密都会给那些不慎重思考的人一种安全的错觉，这里一般性的准则是：

17. 一个安全系统只能和它的秘密一样安全，当心伪安全。

2.9 集市风格的必要的先决条件

本文的早期评审人员和测试人员坚持提出成功的市集模式开发的先决条件，包括工程领导人的资格问题和在把项目公开和开始建造一个协作开发人员的社团的时候代码的状态。

相当清楚，不能以一个市集模式从头开发一个软件，我们可以以市集模式、测试、调试和改进，但是以市集模式从头开始一个项目将是非常困难的，Linus 没有这样做，我也没有，初期的开发人员的社团应该有一此可以运行和测试的东西来玩。

当你开始创建社团时，你需要演示的是一个诺言，你的程序不需要工作的很好，它可以很粗糙、很笨拙、不完整和缺少文档、它不能忽略的东西是要吸引哪些人卷入一个整洁的项目。

Linux 和 fetchmail 都是以一个吸引人的基本设计进入公共领域的，许多和我一样在思考市集模式的人已经正确的认为这是非常关键的，然后得出了一个结论，工程领导者的高度的设计直觉和聪颖是必不可少的。

但是 Linus 是从 Unix 得到他的设计的，我最初是从先前的 popmail 得到启发的（虽然相对 Linux 而言，它最后改变巨大），所以市集风格的领导人/协调人需要有出众的设计才能，或者他可以利用别人的设计才能?

我认为能够提出卓越的原始设计思想对协调人来说不是最关键的，但是对他/她来说绝对关键的是要能把从他人那里得到的好的设计重新组织起来。

Linux 和 fetchmail 项目都显示了这些证据，Linus（如同前面所说）并不是惊人的原始设计者，但他显示了发现好的设计并把它集成到 Linux 内核中的强大决窍。还有我也描述了怎样从别人那里得到了 fetchmail 中最强大的设计思想（SMTP 转发）。

本文的早期读者称赞我，说因为我做了许多关于原始设计的事，所以倾向于低估原始设计在市集项目中的价值，也许有些是对的吧，但是设计（而不是编码或调试）本来就是我最强的能力。

变得聪明和软件设计的原始创作的问题是它会变成一个习惯，当需要保持事物健壮和简洁的时候，你却开始把事情变得漂亮但却复杂。我曾经犯过错误，使得一些项目因我而崩溃了，但我努力不让它发生在 fetchmail 身上。

所以我相信 fetchmail 项目的成功部分是因为我抑制自己不要变得太聪明，这说明（至少）对市集模式而言原始设计并不是本质的，请考察一下 Linux 假设 Linus Torvalds 在开发时试图彻底革新操作系统设计，它还会象今天我们所拥有的内核那样稳定和成功吗?

当然基本的设计和编码技巧还是必需的，但我希望每个严肃考虑发起一个市集计划的人都已至少具备这些能力，自由软件社团的内部市场对人们有某些微妙的压力，让他们不要发起自由不能搞定的开发，目前为止，这工作得仍然相当好。

对市集项目来说，我认为还有另一种通常与软件开发无关的技能和设计能力同样重要 —— 或者更加重要，市集项目的协调人或领导人必须有良好的人际和交流能力。

这是很显然的，为了建造一个开发社团，你需要吸引人，你所做的东西要让他们感到有趣，而且要保持他们对他们正在做的工作感到有趣，而且要保持他们对他们正在做的工作感到高兴，技术方面对达成这些目标有一定帮助，但这远远不是全部，你的个人素质也有关系。

并不是说 Linus 是一个好小伙子，让人们喜爱并乐于帮助他，也并不是说我是个积极外向的，喜欢扎堆儿工作，有出众的幽默感的人，对市集模式的工作而言，至少有一点吸引人的技巧是非常有帮助的。

2.10 自由软件的社会学语境

下述如实：最好的开发是从作者解决每天工作中的个人问题开始的，因为它对一大类用户来说是一个典型问题，所以它就推广开来了，这把我们带回到准则 1，也许是用一个更有用的方式来描述：

18. 要解决一个有趣的问题，请从发现让你感兴趣的问题开始。

这是 Carl Harris 和原先的 popclient 的情形，也是我和 fetchmail 的情形，但这已在很长一段时间被大家知晓了，Linux 和 fetchmail 的历史要求我们注意的有趣之处是下一个阶段 —— 软件在一个庞大的活跃的用户和协作开发人员的社团中的进化。

在《人月神话》一书中，Fred Brooks 观察到程序员的工作时间是不可替代的：在一个误了工期的软件项目中增加开发人员只会让它拖得更久，他声称项目的复杂度和通讯开销以开发人员的平方增长，而工作成绩只是以线性增长，这个说法被称为“Brooks 定律”，被普遍当作真理，但如果 Brooks 定律就是全部，那 Linux 就不可能成功。

几年之后，Gerald Weinbeng 的经典之作“The Psychology Of Computer Progromming”为我们更正了 Brooks 的看法，在他的“忘我（egoless）的编程”中，Weinberg 观察到在开发人员不顽固保守自己的代码，鼓励其他人寻找错误和发展潜力的地方，软件的改进的速度会比其他地方有戏剧性的提高。

Weinberg 的用词可阻止了他的分析得到应有的接受，人们对把 Internet 黑客称为“忘我”的想法微笑，但是我想今天他的想法比以往任何时候都要引人注目。

Unix 的历史已经为我们准备好了我们正在从 Linux 学到的（和我在更小规模上模仿 Linus 的方法所验证的）东西，这就是，虽然编码仍是一个人干的活，真正伟大的工作来自于利用整个社团的注意和脑力，在一个封闭的项目中只利用他自己的脑力的人会落在知道怎样创建一个开放的、进化的，成百上千的人在其中查找错误和进行修改的环境的开发人员之后。

但是 Unix 的传统中有几个因素阻止把这种方法推到极致。一个是各种授权的法律约束、商业机密和商业利益，另一个（事后来看）是 Internet 还不够好。

在 Internet 变得便宜之前，有一些在地理上紧密的社团，它们的文化鼓励 Weingberg 的“忘我”编程，一个开发人员很容易吸引许多熟练的人和协作开发人员，贝尔实验室，MIT A1 实验室，UC Berkeley，都成为传统的、今天仍然是革新的源泉。

Linux 是第一个有意识的成功地利用整个世界做为它的头脑库的项目，我不认为 Linux 的孕育和万维网的诞生相一致是一个巧合，而且 Linux 在 1993–1994 的一段 ISP 工业大发展和对 Internet 的兴趣爆炸式增长的时期中成长起来，Linus 是第一个学会怎样利用 Internet 的新规的人。

廉价的 Internet 对 Linux 模式的演化来说是一个必要条件，但它并不充分，另一个关键因素是领导风格的开发和一套协作的氛围使开发人员可以吸引协作开发人员和最大限度地利用媒体。

但是这种领导风格与氛围到底是什么呢？它不能建立在权力关系之上 —— 甚至如果它们可以，高压的领导权力也不能产生我们所看到的结果，Weinberg 引用了 19 世纪俄国的无政府主义者 Kropotkin 的“Memoris of a Revolutionist”来证明这个观点：

“我从小生活在一个农奴主的家庭中，我有一个活跃的生活，象我们时代的所有年轻人一样，我深信命令、强制、责骂、惩罚等等的必要性。但是当我（在早期）必须管理一个企业，和（自由）人打交道时，当每一个错误都会产生严重后果时，我开始接受以命令和纪律为准则来行动和以普通理解为准则来行动的区别。前者在军事阅兵中工作的很好，但是它在现实生活中一文不值，目标达成只是靠许多愿望的聚合的简单后果。”“许多聚合在一起的愿望的直接后果”精确地指出了象 Linux 的项目所需要的东西。“命令的准则”在 Internet 这种无政府主义的天堂中一群自愿者之中是没有市场的，为了更有效的操作和竞争，想领导协作项目的黑客们必须学会怎样以 Kropotkins 含糊指出的“理解的准则”模式来恢复和激活社团的力量，他们必须学会使用 Linus 定律。

前面我引用“Delhpi 效应”来作为 Linus 定律的一个可能的解释，但是来自生物学和经济学的自适应系统的更强大的分析也提出了自己的解释，Linus 世界的行为更象一个自由市场或生态系统，由一大群自私的个体组成，它们试图取得（自己）最大的实效，在这个过程中产生了比任何一种中央计划都细致和高效的自发的改进的结果，所以，这里就是寻找“理解的准则”的地方。

Linux 黑客取得的最大化的“实际利益”不是经典的经济利益，而是无形的他们的自我满足和在其他黑客中的声望，（有人会说他们的动机是“利他的”，但这忽略了这样的事实：利他主义本身是利他主义者的一种自我满足的形式），自愿的文化以这种方式工作的实际上并非不寻常，我已参与一个科幻迷团体很长时间了，它不象黑客团体一样，显式地识别出“egoboo”（一个人在其他爱好者之中的声望的增长）作为自愿者活动背后的基础驱动力）。

Linus 成功地把自己置于项目的守门人的位置，在项目中开发大部分是别人做的，他只是在项目中培养兴趣直到它可以自己发展下去，这为我们展示了对 Kropokin 的“共同理解原则”的敏锐把握，对 Linux 这种类似经济学的观点让我们看到这种理解是怎样应用的。

我们可以把 Linus 的方法视为创建一个高效的关于“egoboo”（而不是钱）的市场，来把自私的黑客个体尽可能紧密的联系起来，达成只能通过高度协作才能得到的困难的结果，在 fetchmail 项目中我展示了（在较小规模上）这种模式可以复制，得到良好的结果，也许我比他更有意识一点、更加系统一点。

许多人（尤其是哪些由于政治原因不信任自由市场的人）会盼望自我导向的自我主义者的文化破碎、报废、秘密和敌对，但这种盼望很明显地被 Linux 的文档的多样性、质量和深度打破了，程序员讨厌写文档似乎已是

圣训，但 Linux 的黑客们怎么产生了这么多？显然 Linux 的 egoboo 自由市场比有大量资金的商业软件产品的文档部在产生有品德的、他人导向的行为方面工作的更好。

Fetchmail 和 Linux 内核项目都表明，通过恰当的表彰许多其他黑客，一个强大的开发者/协调者可以用 Internet 得到许多协同开发人员而不是让项目分崩离析为一片混乱，所以关于 Brooks 定律我得到了下面的想法：

19. 如果开发协调人员有至少和 Internet 一样好的媒介，而且知道怎样不通过强迫来领导，许多头脑将不可避免地比一个好。

我认为自由软件的将来将属于那些知道怎样玩 Linux 的游戏的人，把大教堂抛之脑后拥抱市集的人，这并不是说个人的观点与才气不再重要，而是，我认为自由软件的前沿将属于从个人观点和才气出发的人，然后通过共同兴趣自愿社团的高效建造来扩展。

可能不只是自由软件的将来，在解决问题方面，没有任何商业性开发者可以与 Linux 社团的头脑库相匹敌，很少有人能负担起雇佣 200 多个为 fetchmail 出过力的人！

也许最终自由软件文化将胜利，不是因为协作在道德上是正确的或软件“囤积居奇”在道德上是错的（假设你相信后者，Linus 和我都不），而仅仅是因为商业世界在进化的军备竞赛中不能战胜自由软件社团，因为后者可以把更大更好的开发资源放在解决问题上。

2.11 网友写给作者的感想

你好，Eric：我刚读了你的大教堂/市集的文章，因为你的主页指出你还要继续关于这个问题的思考，我提供一些个人的观察。首先介绍一些背景：当 1990 年出现 BSD Net/2 的时候，Brad Grantham 和我把它移植到了 Mac 平台上，它在几个月之后以 Mac BSD 发布（当然是以市集风格），后来成为 Net BSD/Mac。我作为一个市集协调人学到了一些东西：

1. 人们很快地自愿提供帮助，但是常常很慢，我们收到上百封信说：“我很想帮助，请告诉我需要什么?”这些人没提供什么帮助，不管他们有多么积极，真正有帮助的人那些给我们的第一封信便说：“嘿，我修改了这个，这儿有一个补丁。”最后我们忽略了所有第一种类型的邮件（只是把他们引向工作列表），培养与第二种人的关系，这种情况所有协调人都应知道，来克服看到这么多“志愿者”时的盲目高兴。（注意：他们的动机是好的，他们只是没有认识到他们正在志愿做什么）。

2. 你已经提到了这一点，但我认为它是极端重要的：甚至在你宣布产品以前你必须有一个可工作的系统：例如，我们一直等到有了一个可引导的内核和一个单用户根 shell 之后才把它贴到 Usenet，曾有过（据我所知）四个不同的 Mac Linux 项目，每一个都在 Linux 新闻组中有一大批拥护者，都创建了邮件列表，每个人都很热情，写了 FAQ，还有许多诸如 MacOS 的图标应是什么样的讨论。所有这些项目没有发布一行代码或者一个内核、我挑选了 MkLinux（Apple 开发的）作为一个可工作的 Mac 版 Linux（在一个项目中，MacLinux 假设运转在 68K Mac 上，而邮件列表中所有的讨论都是关于怎样把它移植到 Power Mac 上。68K 版本甚至不能远程工作！），这些项目吸引了上述的第一种“帮助者”，热情高涨但是实际上却没做什么事，杀掉一个项目最快的方法是在你什么都还没有之前就宣布它，我已经见的太多了，尤其是在 Linux 世界里。

我知道这两点看起来相当悲观，但我知道当我们想到“啊，我们做了这么多事了，肯定搞定了不少问题了吧！”的时候，我们太容易失去理智。而那实际上只不过是一些善良的动机罢了（谁说过：“不要把动机和行动混淆在一起？”本·弗兰克林？）协调人需要解散所有那些诸如图标应该是什么样的、FAQ 用 HTML 格式还是 SGML 模式的热情讨论，而把注意力放在取得产品的一个可工作的版本，一旦得到了，人们就真正开始帮助了。

（从正面来看，MacBSD 极大地得益于从它的开发风格，我们得到了代码、设备驱动程序、钱和一些捐赠和借到的测试和开发的硬件设备）。我期望看到对我上述观点的任何评论和你关于这个主题写的任何东西。

3 如何成为一名 Hacker

3.1 为什么会有这份文档?

作为 Jargon File 的编辑和一些其他有名的类似性质文章的作者，我经常收到充满热情的网络新手的 email 提问（确实如此）“我如何才能成为一名出色的黑客？”非常奇怪的是似乎没有任何的 FAQ 或者 Web 形式的文档来说明这个十分重要的问题，因此我写了一份。

如果你现在读的是这份文档的离线拷贝，那么请注意当前最新版本（英文版）在 http://www.catb.org/~esr/faqs/hacker-howto.html 可以得到。

注意：在这份文档最后有 FAQ（常问问题解答）。请在向我提出任何关于这份文档的疑问之前读两遍。

目前这份文档有很多翻译版本：保加利亚语，简体中文，繁体中文，丹麦语，荷兰语，法语，德语，匈牙利语，印尼语，日语，朝鲜语，葡萄牙语，俄语及瑞典语。注意由于这份文档时有修正，所以以上翻译版本可能有不同程度的过时。

3.2 什么是黑客?

Jargon File 包含了一大堆关于“hacker”这个词的定义，大部分与技术高超和热衷解决问题及超越极限有关。但如果你只想知道如何成为一名黑客，那么只有两件事情确实相关。

这可以追溯到几十年前第一台分时小型电脑诞生，ARPAnet 实验也刚展开的年代，那时有一个由程序设计专家和网络名人所组成的，具有分享特点的文化社群。这种文化的成员创造了“hacker”这个名词。黑客们建立了 Internet。黑客们发明出了现在使用的 UNIX 操作系统。黑客们使 Usenet 运作起来，黑客们让 WWW 运转起来。如果你是这个文化的一部分，如果你对这种文化有所贡献，而且这个社群的其它成员也认识你并称你为 hacker，那么你就是一位黑客。

黑客精神并不仅仅局限在软件的黑客文化中。有人用黑客态度对待其它事情，如电子学和音乐 —— 事实上，你可以在任何最高级别的科学和艺术活动中发现它。精于软件的黑客赞赏这些在其他领域的同类并把他们也称作黑客 —— 有人宣称黑客天性是绝对独立于他们工作的特定领域的。但在这份文档中，我们将注意力集中在软件黑客的技术和态度，以及发明了“黑客”一词的以共享为特征的文化传统之上。

有一群人大声嚷嚷着自己是黑客，但他们不是。他们（主要是正值青春的少年）是一些蓄意破坏计算机和电话系统的人。真正的黑客把这些人叫做“骇客”（cracker），并不屑与之为伍。多数真正的黑客认为骇客们又懒又不负责任，还没什么大本事。专门以破坏别人安全为目的的行为并不能使你成为一名黑客，正如用铁丝偷开走汽车并不能使你成为一个汽车工程师。不幸的是，很多记者和作家往往错把“骇客”当成黑客；这种做法一直使真正的黑客感到恼火。

根本的区别是：黑客搞建设，骇客搞破坏。

如果你想成为一名黑客，请接着读下去。如果你想做一个骇客，去读 alt.2600 新闻组，并在意识到你并不像自己想象的那么聪明后去坐五到十次监狱。关于骇客，我只想说这么多。

3.3 黑客应有的态度

黑客们解决问题，建设事物，同时他们崇尚自由和无私的双向帮助。要被他人承认是一名黑客，你的行为得体现出你好像具备了这种态度一般。而要想做得好象你具备这种态度一般，你就得切切实实坚持它。

但是如果你认为培养黑客态度只是一条在黑客文化圈中得到承认的路子，那就大错特错了。成为具备这些特质的这种人对你自己非常重要 —— 有助于你学习，及给你提供源源不断的动力。同所有创造性的艺术一样，成为大师的最有效方法就是模仿大师的精神 —— 不仅从智力上，也要从感情上进行模仿。

或许，下面这首现代的禅诗很好的阐述了这个意思：

> To follow the path:（沿着这样一条道路：）
> look to the master,（寻找大师，）
> follow the master,（跟随大师，）
> walk with the master,（与大师同行，）
> see through the master,（洞察大师，）
> become the master.（成为大师。）

嗯，如果你想成为一名黑客，反复读下面的事情直至你相信它们：

3.3.1 世界充满了待解决的迷人问题

做一名黑客会有很多乐趣，但却是要费很多气力方能得到的乐趣。这些努力需要动力。成功的运动员从锻炼身体、超越自我极限的愉悦中得到动力。同样，做黑客，你得能从解决问题，磨练技术及锻炼智力中得到基本的乐趣。

如果你还不是天生的这类人又想做黑客，你就要设法成为这样的人。否则你会发现，你的黑客热情会被其他分心的事物吞噬掉 —— 如金钱、性和社会上的虚名。

（同样你必须对你自己的学习能力建立信心 —— 相信尽管当你对某问题近乎一无所知，但只要你一点一点地试验、学习，最终会掌握并解决它。）

3.3.2 一个问题不应该被解决两次

聪明的脑袋是宝贵的有限的资源。当世界还充满非常多有待解决的有趣的新问题时，它们不应该被浪费在重新发明轮子这类事情上。

作为一名黑客，你必须相信其他黑客的思考时间是宝贵的 —— 因此共享信息，解决问题并发布结果给其他黑客几乎是一种道义，这样其他人就可以去解决新问题而不是不断地忙于对付旧问题。

（你不必认为一定要把你所有的发明创造公布出去，但这样做的黑客是赢得大家极度尊敬的人。卖些钱来养家糊口，租房买计算机甚至发大财和黑客价值观也是相容的，只要你别忘记你还是个黑客。）

3.3.3 无聊和乏味的工作是罪恶

黑客（泛指具有创造力的人们）应该从来不会被愚蠢的重复性劳动所困扰，因为当这种事情发生时就意味着他们没有在做只有他们才能做的事情 —— 解决新问题。这样的浪费伤害每一个人。因此，无聊和乏味的工作不仅仅是令人不舒服而已，而且是罪恶。

作为一个黑客，你必须坚信这点并尽可能多地将乏味的工作自动化，不仅为你自己，也为了其他人（尤其是其他黑客们）。

（对此有一个明显的例外。黑客有时也做一些在他人看来是重复性或枯燥的工作以进行“脑力休息”，或是为了获得某种技能，或是获得一些除此以外无法获得的特别经验。但这是自愿的 —— 有脑子的人不应该被迫做无聊的活儿。）

3.3.4 自由万岁

黑客们是天生的反独裁主义者。任何能向你发命令的人能够迫使你停止解决令你着迷的问题，同时，按照独裁者的一般思路，他通常会给出一些极端愚昧的理由。因此，不论何处，任何独裁主义的作法，只要它压迫你和其他黑客，你就要和它斗到底。

（这并非向所有权威挑战。儿童需要监护，罪犯要被看管起来。如果服从命令得到某种东西比起用其他方式得到它更节约时间，黑客可以同意接受某种形式的权威。但这是一个有限度的，有意的交易；那种权威想要的个人服从不是你应该同意给予的。）

权威喜欢审查和保密。他们不信任自愿的合作和信息的共享 —— 他们只喜欢由他们控制的所谓“合作”。因此，作为一个黑客，你得对审查、保密，以及使用武力或欺骗去压迫有行为能力的人们的做法有一种本能的敌意。同时你要有为此信念斗争的意愿。

3.3.5 态度不能替代能力

作为一名黑客，你必须培养起这些态度。但只具备这些态度并不能使你成为一名黑客，也不能使你成为一个运动健将和摇滚明星。成为一名黑客需要智力，实践，奉献精神和辛苦工作。

因此，你必须学会怀疑，并尊重各种各样的能力。黑客们不会为那些装模做样的人浪费时间，但他们却非常尊重能力 —— 尤其是从事黑客工作的能力，不过任何能力总归是好的。具备很少人能具备的那些方面的能力尤其好，其中具备涉及脑力、技巧和专注方面能力的当然最好。

尊敬能力，你就会享受到提高自己能力的乐趣 —— 辛苦的工作和奉献会变成一种高度娱乐而非苦差事。要想成为一名黑客，这一点非常重要。

3.4 黑客的基本技能

黑客态度重要，但技术更加重要。态度无法替代技术，在你被别的黑客称为黑客之前，有一些基本的技术你必须掌握。

这些基本技术随着新技术的出现和老技术的过时也随时间在缓慢改变。例如，过去内容包括使用机器语言编程，而直到最近才包括了 HTML。总的来说现在主要包括以下技术：

3.4.1 学习如何编程

这当然是最基本的黑客技能。如果你还不会任何编程语言，我建议你从 Python 开始。它设计清晰，文档齐全，适合初学者入门。它是一门很好的入门语言，并且不仅仅只是个玩具；它非常强大、灵活，也适合做大型项目。我有一篇 Python 评价详细说明这点。好的教程可以在 Python 网站得到。

Java 也是好的入门语言。它比 Python 难得多，但是生成的代码速度也快得多。它同时也是一种优秀的计算机语言，不止是用来入门。

但是注意，如果你只会一两门语言，你将不会达到黑客所要求的技术水平，甚至也不能达到一个程序员的水平 —— 你需要学会如何以抽象的方式思考编程问题，独立于任何语言。要做一名真正的黑客，你需要学会在几天内通过一些手册，结合你现在所知，迅速掌握一门新语言。这意味着你应该学会几种截然不同的语言。

如果要做一些重要的编程工作，你将不得不学习 C 语言，Unix 的核心语言。C++ 与 C 非常其他类似；如果你了解其中一种，学习另一种应该不难。但这两种都不适合编程入门者学习。而且事实上，你越避免用 C 编程，你的工作效率会越高。

C 非常有效率，节约你的机器资源。不幸的是，C 的高效是通过你手动做很多底层的管理（如内存）来达到的。底层代码都是复杂极易出现 bug 的，会使你花极多的时间调试。如今的机器速度如此之快，这通常是得不偿失 —— 比较明智的做法是使用一种运行较慢、较低效率，但大幅节省你的时间的语言。因此，选择 Python。

其他对黑客而言比较重要的语言包括 Perl 和 LISP。Perl 实用，值得一学；它被广泛用于动态网页和系统管理，因此即便你从不用 Perl 写程序，至少也应该学会看。许多人使用 Perl 的理由和我建议你使用 Python 的理由一样，都是为了避免用 C 完成那些不需要 C 高效率的工作。你会需要理解那些工作的代码的。

LISP 值得学习的理由不同 —— 最终掌握了它时你会得到丰富的启迪和经验。这些经验会使你在以后的日子里成为一个更好的程序员，即使你实际上很少使用 LISP 本身。

当然，实际上你最好五种都会（Python，Java，C/C++，Perl 和 LISP）。除了是最重要的黑客语言外，它们还代表了截然不同的编程思路和方法，每种都会让你受益非浅。

这里我无法给你完完全全的指导教会你如何编程 —— 这是个复杂的技能。但我可以告诉你，书本和上课也不能作到（最好的黑客中，有许多，也许几乎都是自学成材的）。你可以从书本上学到语言的特点 —— 只是一些皮毛，但要使书面知识成为自身技能只能通过实践和虚心向他人学习。因此要作到（一）读代码及（二）写代码。

学习如何编程就象学习用优美的自然语言写作一样。最好的做法是读一些大师的名著，试着自己写点东西，再读些，再写点，再读些，再写点……如此往复，直到你的文章达到你体会到的范文的简洁和力量。

过去找到适合阅读的好的代码是困难的，因为几乎没有大型程序的源代码能让新手练手。这种状况已经戏剧性地发生变化；开放源代码软件，编程工具和操作系统（全都由黑客写成）现在已经随处可见。让我们在下一个话题中继续讨论……

3.4.2 得到一个开放源代码的 Unix 并学会使用、运行它

我假设你已经拥有或者能使用一台个人电脑（今天的孩子们真幸福 :-)）。新手们能够朝学习黑客技能迈出的最基本的一步就是得到一份 Linux 或 BSD-Unix 的一种，安装在个人电脑上，并运行它。

没错，这世界上除了 Unix 还有其他操作系统。但它们都是以二进制形式发布的 —— 你无法读到它的源代码，也不可能修改它。尝试在运行 DOS 或 Windows 或 MacOS 的机器上学习黑客技术，就象是带着脚镣学跳舞。

除此之外，Unix 还是 Internet 的操作系统。你可以学会上网却不知道 Unix，但你不了解 Unix 就无法成为一名 Internet 黑客。因此，今天的黑客文化在很大程度上是以 Unix 为中心的。（这点并不总是真的，一些很早的黑客对此一直很不高兴，但 Unix 和 Internet 之间的联系已是如此之强，甚至连 Microsoft 也无可奈何。）

所以，安装一套 UNIX —— 我个人喜爱 LINUX 但还有其他种类的（是的，你可以同时安装 Linux 及 DOS/Windows 在同一电脑上）。学习它，使用它，配置它。用它在 Internet 上冲浪。阅读它的源代码。修改它的源代码。你会得到比在 Microsoft 操作系统上更好的编程工具（包括 C，LISP，Python 及 Perl）。你会觉得乐趣无穷，学到在你成为大师之前意识不到的更多的知识。

想知道更多关于学习 Unix 的信息，访问 The Loginataka。

想知道如何得到一份 Linux，访问我在哪里可以获得 Linux。（译者：对于中文读者来讲，最简单的方式未过于前往附近的 D 版/正版光盘店。）

你可以在 `www.bsd.org` 找到 BSD Unix 的求助及其他资源。

我有写一篇关于 Unix 和 Internet 基础的入门文章。

（注：如果你是一个新手，我不推荐自己独立安装 Linux 或者 BSD。安装 Linux 的话，寻求本地 Linux 用户组的帮助；或联系 Open Projects Network。LISC 维护着一些 IRC 频道，在那里你可以获得帮助。）

3.4.3 学会如何使用 WWW 和写 HTML

黑客文化建造的大多东西都在你看不见的地方发挥着作用，帮助工厂、办公室和大学正常运转，表面上很难看到它对非黑客的普通人的生活的影响。Web 是一个大大的例外。即便政客也同意，这个巨大耀眼的黑客玩具正在改变整个世界。单是这个原因（还有许多其它的），你就需要学习掌握 Web。

这并不是仅仅意味着如何使用浏览器（谁都会），而是要学会如何写 HTML，Web 的标记语言。如果你不会编程，写 HTML 会教你一些有助于学习的思考习惯。因此，先完成一个主页。（网上有很多好的教程，这是一个。）

但仅仅拥有一个主页不能使你成为一名黑客。Web 里充满了各种网页。大多数是毫无意义的，零信息量垃圾 —— 界面时髦的垃圾，注意，垃圾的水准都类似（更多信息访问 The HTML Hell Page）。

要想有价值，你的网页必须有内容 —— 它必须有趣或对其它黑客有帮助。这是下一个话题所涉及的……

3.4.4 如果你不懂实用性的英语，学习吧

作为一个美国人和一个以英语为母语的人，我以前很不情愿提到这点，免得成为一种文化上的帝国主义。但相当多以其他语言为母语的人一直劝我指出这一点，那就是英语是黑客文化和 Internet 的工作语言，你需要懂得以便在黑客社区顺利工作。

这一点千真万确。大概 1991 年的时候我就了解到许多黑客在技术讨论中使用英语，甚至当他们的母语都相同，英语对他们而言只是第二语言的时候；据我知道的报导，当前英语有着比其他语言丰富得多的技术词汇，因此是一个对于工作来说相当好的工具。基于类似的原因，英文技术书籍的翻译通常不令人满意（如果有翻译的话）。

Linus Torvalds，一个芬兰人，用英语注释他的代码（很明显这对他来说不是凑巧）。他流利的英语成为他能够管理全球范围的 Linux 开发人员社区的重要因素。这是一个值得学习的例子。

3.5 黑客文化中的地位

像大部分不涉及金钱的文化一样，黑客王国靠声誉运转。你设法解决有趣的问题，但它们到底多有趣，你的解法有多好，是要由那些和你具有同样技术水平的人或比你更厉害的人去评判的。

相应地，当你在玩黑客游戏时，你得认识到你的分数主要靠其他黑客对你的技术的评价给出（这就是为什么只有在其它黑客称你为黑客时，你才算得上是一名黑客）。这个事实常会被黑客是一项孤独的工作这一印象所减弱；也会被另一个黑客文化的禁忌所减弱（现在逐渐减弱但仍强大）：拒绝承认自我或外部评估与一个人的动力有关系。

特别地，黑客王国被人类学家们称为一种奉献文化。在这里你不是凭借你对别人的统治来建立地位和名望，也不是靠美貌，或拥有其他人想要的东西，而是靠你的奉献。尤其是奉献你的时间，你的创造和你的技术成果。

要获得其他黑客的尊敬，基本上有五种事情你可以干：

3.5.1 写开放源代码软件

第一个（也是最集中的和传统的）是写些被其他黑客认为有趣或有用的程序，并把程序源代码提供给整个黑客文化使用。

（过去我们称之为“free software（自由软件）”，但这却使很多不知 free 的精确含义的人感到困惑。现在我们很多人，根据搜索引擎网页内容分析至少有 2:1 的比率，使用“open-source”software（开放源代码软件）这个词）。

黑客王国里最受尊敬的偶像是那些写了大型的、好用的、具有广泛用途的软件，并把它们公布出去，使得每人都在使用他软件的人。

3.5.2 帮助测试并调试开放源代码软件

黑客也尊敬那些使用、测试开放源代码软件的人。在这个并非完美的世界上，我们不可避免地要花大多数的开发时间在调试阶段。这就是为什么任何有头脑的开放源代码的作者都会告诉你好的 beta 测试员（知道如何清楚描述出错症状，很好地定位错误，能忍受快速发布中的 bug，并且愿意使用一些简单的诊断工具）象红宝石一样珍贵。甚至他们中的一个能判断出哪个测试阶段是延长的，哪个是令人精疲力尽的噩梦，哪个只是一个有益的小麻烦。

如果你是个新手，试着找一个你感兴趣的正在开发的程序，做一个好的 beta 测试员。你会自然地从帮着测试，进步到帮着抓臭虫，到最后帮着改程序。你会从中学到很多，并且与未来会帮你的人结下友谊。

3.5.3 公布有用的信息

另一个好事是收集整理有用有趣的信息做成网页或文档如 FAQ 列表，且让他们容易获得。

主要技术 FAQ 的维护者受到几乎同其他开放源代码的作者一样多的尊敬。

3.5.4 帮助维护基础设施的运转

黑客文化（还有 Internet 的工程方面的发展，就此而言）是靠自愿者运转的。要使 Internet 能正常工作，就要有大量枯燥的工作不得不去完成 —— 管理 mail list，新闻组，维护大型软件库，开发 RFC 和其它技术标准等等。

做这类事情的人会得到很多尊敬，因为每人都知道这些事情是十分花时间又不象编程那样好玩。做这些事情需要奉献精神。

3.5.5 为黑客文化本身服务

最后，你可以为这个文化本身做宣传（例如，象我这样，写一个“如何成为黑客”的正面的教程 :–)）（译者：不知道 Barret 把它翻成中文算不算？）。这并非一定要在你已经在这个圈子呆了很久，因以上四点中的某点而出名，有一定声誉后才能去做。

黑客文化没有领袖。精确地说，它确实有些文化英雄、部落长者、历史学家和发言人。若你在这圈内呆的够长，你或许成为其中之一。记住：黑客们不相信他们的部落长者的自夸的炫耀，因此大举追求这种名誉是危险的。与其奋力追求，不如先摆正自己的位置等它自己到你的手中 —— 那时则要做到谦虚和优雅。

3.6 黑客和书呆子（Nerd）的联系

同流行的迷思相反，做一名黑客并不一定要你是个书呆子。但它确实有帮助，而且许多黑客事实上是书呆子。做一个深居简出的人有助于你集中精力进行十分重要的事情，如思考和编程。

因此，很多黑客都愿意接受“书呆子”这个外号，更有甚者使用更尖刻的“geek（怪人）”一词并引以为豪 —— 这是一种宣布他们独立于主流社会的声明方式。访问 The Geek Page 参加更多的讨论。

如果你能集中足够的精力做好黑客工作同时还能有正常的生活，这很好。现在作到这一点比我在 1970 年代是新手的时候要容易的多；如今主流文化对技术怪人要友善的多。甚至有越来越多的人意识到黑客通常是很好的恋人和配偶的材料。

如果你因为生活上不如意而迷上做黑客，那也没什么 —— 至少你不会分神了。或许以后你会找到自己的另一半。

3.7 风格的意义

重申一下，作为一名黑客，你必须进入黑客精神之中。当你不在计算机边上时，你仍然有很多对黑客工作有帮助的事情可做。它们并不能替代真正的编程（没有什么能），但很多黑客都那么做，并感到它们与黑客的本质存在某些基本的联系。

学会流畅地用母语写作。尽管程序员不能写好文章的错误看法相当普遍，但是有令人惊讶数目的黑客（包括所有我知道的最棒的）都是不错的作家。

阅读科幻小说。参加科幻小说讨论会。（一个碰到黑客和未来会成为黑客的人的好方法）

学禅，并且/或者练功习武。（精神修炼看来是惊人相似。）

练就能分析音乐的听觉，学会鉴赏特别的音乐。学会玩某种乐器，或唱歌。

提高对双关语、文字游戏的鉴赏能力。

这些事情，你已经做的越多，你就越是天生做黑客的材料。至于为什么偏偏是这些事情，原因并不完全清楚，但它们都涉及用到左 – 右脑能力的综合，这似乎是关键所在（黑客们既需要清晰的逻辑思维，有时又需要偏离逻辑跳出问题的表象）。

最后，还有一些不要去做的事情。

不要使用愚蠢的，哗众取宠的 ID 或昵称。

不要卷入 Usenet（或其他地方的论坛）的骂战。

不要自称为“cyberpunk（网络叛客）”，也不要浪费时间和那些人打交道。

不要让你寄出的 Email 或张贴的帖子充满错误的拼写和乱七八糟的语法。

做以上的事情，只会招来嘲笑。黑客们个个记忆超群 —— 你将需要数年的时间让他们忘记你犯下的错误。

网名的问题值得深思。将身份隐藏在虚假的名字后是骇客、解密者、d00dz 及其他低等生物幼稚愚蠢的行为特点。黑客不会做这些事；他们对他们所作的感到骄傲，而且乐于人们将作品与他们的真名相联系。因此，若你现在用假名，放弃它。在黑客文化里它会令你失败的。

3.8 其它资源

Peter Seebach 维护着一个非常好的 Hacker FAQ，专给那些不懂如何与黑客打交道的经理看的。如果 Peter 的站点不能访问，下面这个 Excite 搜索应该有一份拷贝。

我也著有黑客文化简史。

我写了一份《大教堂与市集》，对于 Linux 及开放源代码文化现象有详细的解释。我也在这个话题上进一步阐述导致的结局 —— 开拓智域。

Rick Moen 写了一份很好的关于如何运转一个 Linux 用户组的文档。

我和 Rick Moen 合作完成了另一份关于《提问的智慧》的文章，可以让你事半功倍地获得帮助。

如果你想知道 PC、UNIX 及 Internet 基本概念和工作原理，参考 The Unix and Internet Fundamentals HOWTO。

当你释放出一个软件或为其打补丁，试着按软件发行惯例 HOWTO 去做。（以上的提到的文章的中文版大多都可以在 www.aka.org.cn 和 www.linuxforum.net 找到。）

3.9 FAQ（常问问题解答）

问：你能教我做黑客吗？
问：那么，我要如何开始？
问：我得什么时候开始学？现在会不会太迟了？
问：要学多久才能学会黑客道？
问：Visual Basic 及 Delphi 是好的入门语言吗？
问：你能帮我“黑”掉一个站点吗？或者教我怎么黑它？
问：我怎么样才能得到别人帐号的密码？
问：我如何入侵/查看/监视别人的 Email？
问：我如何才能在 IRC 聊天室里偷到频道 op 的特权？
问：我被黑了。你能帮我避免以后再被攻击吗？
问：我的 Windows 软件出现问题了。你能帮我吗？
问：我在哪里能找到可以与之交流的真正的黑客？
问：你能推荐一些有关黑客的好书吗？
问：成为一名黑客我需要擅长数学吗？
问：我该从那种语言学起？
问：我需要什么样的机器配置？
问：我得因此憎恨和反对 Microsoft 吗？
问：但开放源代码软件不会使程序员丢饭碗吗？
问：我要如何开始？哪里有免费的 Unix？

问：你能教我做黑客吗？

答：自从第一次发布这份文档，我每周都会收到一些请求，（频繁的话一天几封）要我“教会他们做黑客”。遗憾的是，我没有时间和精力来做这个；我自己的黑客项目，及我作为一个开放源代码倡导者的四处奔波已经占用了我 110% 的时间。

即便我想教你，黑客也依然基本上是一项自行修炼的的态度和技术。当真正的黑客想帮助你的时候，如果你乞求他们一汤匙一汤匙“喂”你的话，你会发现他们不会尊重你。

先去学一些东西。显示你在尝试，你能靠自己去学习。然后再去向你遇到的黑客请教特殊的问题。

如果你发 E-mail 给一位黑客寻求他的帮助，这是两件首要记住的事情。第一，写出来的文字显得懒且粗心的人通常非常懒于思考且非常马大哈，不能成为好黑客 —— 因此注意拼写正确，使用正确的语法及发音，否则你可能会无人理睬。第二，不要试图要求回复到一个 ISP 帐号，而那个帐号与你的发信地址不同。这样做的人一般是使用盗用帐号，不会有人有兴趣为虎作伥帮助窃贼的。

问：那么，我要如何开始？

答：对你而言最佳的入门方式也许是去参加 LUG（Linux 用户组）的聚会。你可以找到在 LDP 的综合 Linux 信息页面上找到类似的组织；也许有一个在你家附近的，而且非常有可能与一所大学或学校挂钩。如果你提出要求，LUG 成员兴许会给你一套 Linux，当然此后会帮你安装并带你入门。

问：我得什么时候开始学？现在会不会太迟了？

答：你有动力学习的时候就是好时候。大多数人看来都是在 15 – 20 岁之间开始感兴趣的，但据我所知，在此年龄段之外的例外也是有的。

问：要学多久才能学会黑客道？

答：这取决于你的聪明程度和努力程度。大多数人只要他们专注，就能在 18 个月到 2 年之间学会一套令人尊敬的技能。但是，不要以为就此结束了；如果你是一个真正的黑客，你要用你的余生来学习和完善你的技术。

问：Visual Basic 及 Delphi 是好的入门语言吗？

答：不，因为他们不是可移植的。他们不是那些语言的开放源代码实现，所以你被限制在厂商选择支持的那些平台里。接受这样一种垄断局面不是黑客的态度。

Visual Basic 特别糟糕。它是 Microsoft 的私有语言这个事实就足够让它脸面全无，不像其他的 Basic，它是一种设计糟糕的语言会教给你坏的编程习惯。

其中一个坏习惯是会依赖于单一厂商的函数库、控件及开发工具。一般而言，任何不能够支持至少 Linux 或者一种 BSD，或其他第三方操作系统的语言，都是一种不适合应付黑客工作的语言。

问：你能帮我“黑”掉一个站点吗？或者教我怎么黑它？

答：No。任何读完这份 FAQ 后还问这个问题的人，都是无可救药的蠢材，即使有时间指教我也不会理睬。任何发给我的此类 E-mail 都会被忽略或被痛骂一顿。

问：我怎么样才能得到别人帐号的密码？

答：这是骇客行为。滚得远远的，白痴。

问：我如何入侵/查看/监视别人的 Email？

答：这是骇客行为。在我面前消失，混蛋。

问：我如何才能在 IRC 聊天室里偷到频道 op 的特权？

答：这是骇客行为。去 S 吧，冥顽不灵的家伙。

问：我被黑了。你能帮我避免以后再被攻击吗？

答：不行。目前为止，每次问我这个问题的，都是一些运行 Microsoft Windows 的菜鸟。不可能有效的保护 Windows 系统免受骇客攻击；太多缺陷的代码和架构使保护 Windows 的努力有如隔靴搔痒。唯一可靠的预防来自转移到 Linux 或其他设计得至少足够安全的系统。

问：我的 Windows 软件出现问题了。你能帮我吗？

答：当然。进入 DOS 方式，然后键入“format c:”。你遇到的任何问题将会在几分钟之内消失。

问：我在哪里能找到可以与之交流的真正的黑客？

答：最佳办法是在你附近找一个 Unix 或 Linux 的用户组，参加他们的聚会。（你可以在 Metalab 的 LDP 站点找到一些指向用户组的链接。）

我过去曾说过不能在 IRC 上找到真正的黑客，但我发觉现在情况有所改变。显然一些真正的黑客的社区像 GIMP 及 Perl，也有 IRC 频道了。）

问：你能推荐一些有关黑客的好书吗？

答：我维护着一份 Linux Reading List HOWTO，也许你会觉得有用。Loginataka 也很有意思。

关于 Python 的介绍，请访问在 Python 站点上的入门资料。

问：成为一名黑客我需要擅长数学吗？

答：不用。黑客道很少使用常规的数学或算术，不过你绝对需要能逻辑性地思考和进行精密的推理。

尤其是你不会用到微积分或电路分析（我们把这些留给电子工程师们 :-)）。一些有限数学（包括布尔代数，集合论，组合数学，图论）的背景知识会有帮助。

问：我该从那种语言学起？

答：HTML —— 如果你还不懂的话。市面上有一大堆的封面精美，宣传得天花乱坠的糟糕的 HTML 书籍，不幸的是很少有好的。我最喜欢的是 HTML: The Definitive Guide。

但 HTML 不完全是一种编程语言。当你准备开始编程时，我推荐从 Python 起步。你会听到一大群人推荐 Perl，并且 Perl 依然比 Python 流行得多，但是难学得多且（以我之见）设计得不是很好。

C 确实重要，但它要比 Python 或 Perl 难多了。不要尝试先学 C。

Windows 用户不要满足于 Visual Basic。它会教给你坏习惯，而且它不可以移植，只能在 Windows 下运行。避免它。

问：我需要什么样的机器配置？

答：过去个人电脑能力相当不够并且内存小，结果给黑客的学习过程设置了人为的障碍。不过一段时间以前开始就不是这样了；任何配置比一台 Intel 486DX50 好的机器都有足够的能力进行开发工作，X，及 Internet 通讯，同时你现在买的最小的磁盘都大得富足了。（依 Barret 之见，现在要至少 Pentium 166MMX 才够。）

选择用来学习的机器时重要的一点是注意配件是否是 Linux 兼容的（或 BSD 兼容，如果你选择学 BSD）。同刚才提到的一样，大多数现在的机器都是符合的；唯一的值得注意的区域在于 modem 和打印机；有些具备为 Windows 设计的配件的机器不会在 Linux 下工作。

关于硬件兼容性有一个 FAQ；最新版本在这里。

问：我得因此憎恨和反对 Microsoft 吗？

答：不，你不必如此。不是因为 Microsoft 不令人讨厌，而是因为黑客文化早在 Microsoft 出现之前就存在了，且将在 Microsoft 成为历史后依然存在。你耗费在憎恨 Microsoft 的任何力气不如花在爱你的技术上。写好的代码 —— 那会相当有效地打击 Microsoft 又不会让你得到恶报应。

问：但开放源代码软件不会使程序员丢饭碗吗？

答：看起来不太可能 —— 目前为止，开放源代码软件产业似乎创造了更多的就业机会而不是减少就业机会。如果写一个程序比起不写来是纯经济收益的话，那么在写完后，程序员应该得到报酬不管程序是否是开放源代码。并且，无论写出多么“免费自由”的软件，都存在更多对新的，定制的软件的需求。我有这方面更多的论述，放在开放源代码网站资料中。

问：我要如何开始？哪里有免费的 Unix？

答：在本份文档的某个地方我已经提到过何处可以得到最常用的免费 Unix。要做一名黑客，你需要自立自强，以及自学能力。现在开始吧……

4 开拓智域

在观察由开放源代码版权所定义的“官方”意识形态与真正玩家的行为后，发现到一些矛盾之处。重新检视真正掌控开放源代码拥有权的文化。我们发现暗藏着一个源自于 Locke 的土地产权理论的现象。我们将此与玩家文化跟“礼物文化”关联起来，意即参与者透过投入时间，精力，及创造力所生产的礼物来竞争并获取名望。接下来检视在文化中，迁涉到对冲突解决的分析，并发展一些常规。

4.1 矛盾的现象

任何人观察忙碌、有巨大生产力的互联网开放源代码软件世界一阵子，一定都会注意到一个很有趣的矛盾，开放源代码玩家们，所说及所做之间的矛盾 —— 即开放源代码官方的意识形态及其实际的实践。

文化是善于应变的机器。开放源代码文化是对应到一系统可指认出的驱力及压力。一如往常，文化对环境的适应力显示出清析的意识形态，及隐含地，潜意识或半意识的意识形态。而，并非不寻常地，半意识的适应力扮演与清析的意识形态同样重要的地位。

在本文中，我们将深掘此一矛盾的根源，并用来发掘这些驱力及压力。我们将演绎一些关于玩家文化习俗的有趣的事。并透过建议一些方式，以升华这一玩家文化的知识层次。

4.2 玩家意识形态的多样性

互联网开放源代码文化的意识形态（玩家们说他们所相信的）本身是相当复杂的话题。所有成员都同意开放源代码（也就是，软件可以免费散播及能够毫无困难地演化及修改成适合自己所需）是件好事，并值得投入大量群体的力量。这样的一致很有效地定义了文化的成员资格。不过，有许多理由值得我们考量，尤其是许多个体及次文化信念的多样化。

一种是狂热；一种不论开放源代码开发仅仅是个到达终点的便利工具（好的工具，有趣的玩具，及有意思的游戏）或者本身就是终点的热情。

狂热之辈会说“自由软件是我的生命！我生命的意义在于生产有用、优美的程序及资讯资源，然后送给人家。”中等热诚者会说“开放源代码是好东西，我愿意花下大量时间协助”。低度热诚之流则会说“对，开放源码有时还可以。我玩弄它，并且尊重建造它的人们”。

另一种是对商业软件或企图统治商业软件市场的企业的敌意。

非常反对商业的人会说“商业软件是偷窃及聚财。我写自由软件来结束这个恶魔。”中等反商业的人会说“商业软件一般还好啦。程序设计师需要有收入，但是那些推行劣质产品并且胡搞一通的则是恶魔”不反商业的人会说“商业软件还好啦。我用或写开放源代码软件是因为我比较喜欢它”。

这样的态度组合表现出九种开放源代码次文化。值得将这分别指出的原因是它们暗示着不同的议题，及不同的适应，及合作的行为。

从历史上来看，最引人注目及最有组织的玩家文化是非常狂热及非常反商业的。由 Richard M. Stallman（RMS）所设立的自由软件基金会，自 1980 年代早期，大力支持着开放源代码的发展，包括 Emacs 及 GCC。到目前为止，在互联网开放源代码世界依然是最基本的，而且看来依然会在未来继续下去。

许多年来，FSF 是唯一最重要的开放源代码的焦点，产生大量至今依然对该文化非常重要的工具。FSF 同时也是对外界来说，唯一对玩家文化的赞助者。他们有效地定义该词“free software”，慎重地定义其重要性（新一点的“open source”则慎重地避免一些误解）。

因此，内内外外对玩家文化的认知都倾向指出该文化是 FSF 的热情态度及反商业目标（RMS 自己否认他是反商业的，但他的程序是被许多人这样的解读，包含他许多的口头游击战）。FSF 的强力而明显的口号“赶走软件聚财员外！”变成玩家意识形态的中心，而 RMS 是最接近玩家文化的领导者的地位。

FSF 的版权条文，“一般大众版权”（GPL），表达了 FSF 的态度。它被开放源代码界广泛地使用。北卡的 Sunsite 是 Linux 界最大及最知名的软件库。在 1997 七月，大约一半的 Sunsite 软件套件都是用 GPL 做版权声明。

但 FSF 并不是游戏中的唯一成员。还有一些比较安静的，比较不那么激烈，而且在玩家文化中，对市场比较友善。实用主义者比较不像早期 FSF 那样的传统上的意识形态。这些传统包含了，更重要的，纠缠着 UNIX 技术文化及前商业的互联网。

典型的实用主义者态度是适度地反商业，而其主要对商业的抱怨并非因为“聚宝库”，而是该世界对较好的技术 UNIX，开放标准及开放源代码软件的顽强拒用。如果实用主义者痛恨任何东西，那大概不是“聚宝”，而是在软件市场的龙头；过去是 IBM，现在 Microsoft。

对实用主义者来说，GPL 的重要性是个工具而非终点目标。它主要的价值不是用来对付“聚宝”，而是用来鼓励分享软件及成长市集模式发展团体。实用主义者以拥有好工具及玩具来衡量价值，而非不喜欢商业，并且会使用高品质商业软件而不会有意识形态上的不舒适。同时，其开放源代码经验已经教导他技术品质的标准，很少会有封闭软件能够达到。

许多年来，实用主义者的观点在玩家文化中，常顽固地透过拒绝完全使用 GPL 版权及 FSF 章程的方式来表达。从 1980 年代到 1990 年代早期，这样的态度倾向与 Berkeley Unix 的风靡有关，BSD 版权的使用者，及早期以 BSD 源代码为基础来建立开放源代码 UNIXes 的付出。这些付出，无法建立起大规模的市集团体，因而变得支离破碎并没有效率。

一直到 Linux 在 1993–1994 年间开始爆发，实用主义者找到了很有力的基础。虽然 Linus Torvalds 从未反对过 RMS，他设下了商业 Linux 工业成长的良好典范，透过保证商业用途软件上的高品质，及轻微地嘲弄清教徒及狂热文化中的元素。

Linux 快速成长的副作用引介了许多新的玩家进入，而 Linux 是他们主要的兴趣，FSF 的信念则是历史的兴趣。虽然新一波的 Linux 玩家会描述 Linux 为“GNU 一代的选择”，大部份会仿效 Torvalds 而不是 Stallman。

许多的反商业清教徒都发现他们自己处于少数地位。在 Netscape 于 1998 年二月宣布要公开其 Navigator 5.0 源代码之前，许许多多的事情都已经改变了。这激起商业世界对“自由软件”的更大兴趣。其后果是唤起玩家文化探索空前的机会及重贴其产品标签，由“自由软件”成为“开放源代码”，以符合大众参与的认证许可。

在一个增援开发中，文化中的实用主义者在 1990 年代有许多个中心。其它半独立的团体及其半意识及有魔力的领导者开始由 UNIX/Internet 的根发芽。在这些当中，在 Linux 之后，最重要的是 Larry Wall 带领的 Perl 文化。比较小，但依然很重要，的是由 John Osterhout 创建的 Tcl 及 Guido Van Rossum 的 Python 语言。所有这三个团体都透过发布其非 GPL 版权方案来表达其意识形态。

4.3 杂乱的理论，清教徒实践

然而，在历经了所有的这些改变，依然存在广泛的舆论在什么是“自由软件”或“开放源码”。对共同理论的最清楚的表述可在各式开放源代码版权中找到，都有一些重要的共同元素。在 1997 年，这些共同元素被结合到 Debian Free Software Guidelines，后来变成开放源代码定义。在开放源代码定义的指导下，一份开放源代码版权必须要保护无条件的由任何人或团体来修改开放源代码软件的权利。

因此，与 OSD 相关的理论（及 OSD–conformant 版权，诸如 GPL，BSD，及 Perl's Artistic License）就是任何人可以玩弄任何东西。没有人可以防止一大票人来玩弄任何开放源代码产品（诸如，以 Free Software Foundations 的 gcc C compiler），复制源代码，将它们朝各种不同的方向演化，但都可以宣称依然是该产品。

但在实际上，这样的“分歧”几乎没有发生过。大计划分歧的状况很少，而且几乎都是由重新贴个标签及大规模的自我正当化来完成。很清楚的，在 GNUEmacs/XEmacs 分支，或 gcc/egcs 分支，或各式各样的 BSD 分支群，这些分歧者都感觉到他们是在对抗一个相当巨大力量的社会规范。

事实上（与 anyone–can–hack–anything 的大众理论相矛盾之处）开放源代码文化有个很复杂但大体自我正当化的拥有权习俗。这些习俗规范了谁能修改软件，这些状况决定了谁能够修改，谁有权力再发行修改过的版本到团体中。

这些文化的禁条明确地规范基准。因此，我们在此摘要一些重要的部份会对未来有点用处。

对计划分歧来说，社会的压力很大。除非实在是非这样做不可，透过大众的自我审判。发行一个未经原创者同意对计划的改变是受到阻止的，除非在特殊状况下，诸如一些一般的修正码。将一个人的名字从计划历史，或维护者中抹去是绝对不可行的，除非当事者同意。在本文，我们将细细检验这些禁条及拥有权习俗。我们将探索它们是如何运作的，并且揭露在其内中的社会动力及开放源代码团体的诱因结构。

4.4 开放源码及拥有权

当产权可无限复制时，“拥有权”的意义为何，或者延申来说，整个文化不具有强制高压力量关系或稀有物质经济? 事实上，在开放源代码文化的例子中这是个很容易回答的问题。软件计划的拥有者是那些拥有独有权力，被大规模团体认定，可再发行修改过的版本。

（在讨论“拥有权”这一节中，我使用单数，因为所有计划都是由某个个体所拥有。有时候，应该要了解有些计划是由一群人所拥有的。我们会在本文后段检验这样族群的内部的动力。）

根据标准的开放源代码版权，所有参与者在演化游戏中都平等。但在实际上，大众对“正式”版，即由大众认可的维护者所整合并认可的版本，及支援厂商的“游离”修补版，有很明显的差别待遇。游离修补版是不寻常的，而且一般都不受信任。

大众发行版是基础的是很容易了解的。习俗鼓励个人需要时可以进行对软件的修补。习俗对那些再将修改版发行的人或发展团队的待遇是不同的。当修改版在开放源代码团体中被发行时，用以与原有版本来竞争，拥有权就变成是个问题。

一般来说有三种方法来看一个开放源代码计划的拥有权。第一种，最明显的，是计划的创建者。当计划只有一个维护者/创建者，而创建者还在活动时，习俗不允许质疑谁是计划的拥有者。

第二个方式是计划的拥有权是由上一个拥有者所转移过来的（有时可称为“转交指挥棒”）。在这个团体中，一般认可，计划拥有者不再感兴趣或无能再进行维护的时候，会将责任交给下一个继任者。

在大计划中这就很重要，控制权的转移往往伴随着华丽的吟咏。对大多数开放源代码团体中，大家所不知道的，它实际影响着拥有者对继承者的选择，习俗明白说明正统嫡系的重要性。

对小一点的计划，在历史记录上记录一下计划拥有者的改变即可。如果前一位拥有者并非志愿的转移控制权，他可以在一段时间内在团体透过向大众公开来取回控制权。

第三种获取计划拥有权的方式是观察该计划需要工作，而拥有者却失去兴趣或消失了。如果您希望做这件事，您需要去找到拥有者。如果您找不到，那么可以在相关的地方宣告（像在该领域的新闻讨论网）该计划似乎变成孤儿，而您正在考虑负起责任来认养。

习俗要求您在您宣告您是新的拥有者之前，要等待一段时间。在这段时间内，如果有人宣告他们已经在这方面开始工作，那么他们的宣告胜过您的。多让大众知道您对这方面的兴趣是很好的方式。最好是你在许多相关的讨论区中宣告（相关新闻讨论网，mailing lists）；而如果您有耐心等待回应。一般，您制造越多的注意，让上一个拥有者或其它的宣告者来反应，当没有人反应时，您的宣告会更有效。

如果您已经通过在计划使用者中这样的过程，而没有人异议，那么您可以宣告该孤儿计划的拥有权，并且在历史档中记下一笔。不过，这比正式交棒来的不安全，您并不被视为嫡系，除非您在使用者族群中做了许多的改善以后。

我已经观察了这样的习俗二十年了，可以回朔到前 FSF 的古早开放源代码软件历史中。他们有许多非常有趣的特色。最有趣的事，大多数的玩家都不需要人家告知便知道要这样做。的确，以上所述是第一次完整写下的摘要。

值得一提的事，潜意识的习俗，实在是令人惊人的协调。我已经不夸张地观察到数百个开放源代码计划的演化，而我还可以用手指来算出这些违反传统的例子。

第三个有趣的特色是这些传统在时间下的演化，他们在协调的方向下如此运作。这样的方向鼓舞更多的大众责任，大众注意，及更关心于现有拥有者保留原有拥有人的成就及历史记录。

这些特色建议了这些习俗并非意外发生的，而是某种暗示性的条文，或是开放源代码文化中的生产形式，在运作中这些是完全基本的条件。

早先一位回响者指出，相对于互联网玩家文化的骇客/海盗文化（“warez d00dz”主要集中于破解游戏及海盗 BBS）也有类同的两种生产形式。我们将会在文中稍后回到 d00dz 的比较。

4.5 Locke 及土地头衔

要将整个一般的型态了解，历史上，在玩家之外，有些类同的传统，与玩家的习俗是相同的。一些历史或政治哲学的学生或者会认出，这个产权的理论暗示着与英美一般惯例法理论中的土地产权相同的观念!

在这个理论中，有三种方法来确认土地的拥有权。

在边疆，领土可以存在而没有拥有者，一个人可以透过开垦来获取拥有权，在未有人拥有土地时，混以血汗，架起栏干，并且抵御个人的名衔。

一般转移土地的方法是头衔转移，也就是从上一位拥有者的手中收到契据。在这个理论中，“头衔传承”观念是很重要的。拥有权理想证据是整个契据传承及转移可用于追朔土地在最早期被开垦时的规模。

最后，一般法理论了解到土地头衔可能会遗失或被抛弃（例如，如果拥有者死去而无继承者，或者要建立头衔传承的文件遗失了）。无主的土地可透过 adverse possession 的方式所认领 —— 透过迁入，改善它，并为其抵御，就由如是开垦它一般。

这个理论，正如玩家习俗一般，会在中央权威薄弱或不存在的背景下有机地演化。它在挪威及日耳曼部族演化了数千年。因为它被早期英国政治哲学家 John Locke 理性而系统化，有时它被称为“Locke”产权理论。

类似逻辑的理论倾向在高度经济或生存价值，而没有任何权威有足够的力量来强制分配稀有物品。在以打猎聚货的文化中，大家认为没有产权观念的想法下，这一点甚至也有。例如，在 Kalahari 沙漠的 !Kung San 布西曼族，并没有猎地的拥有权。但水洞或泉却有类同于 Locke 的产权存在。

!Kung San 的范例是具有教育性的，因为它显示了 Locke 产权习俗适用的时机 —— 当从该资源所能获取的回报大过于需要抵御的代价时。猎地并非产权，因为打猎所能获得的回报很难预测而可变的，而且并非每日生存的必须品。水洞，另一方面来说，对生存来说非常重要，并且小到足够抵御。

在本文标题中，“智域”（“noosphere”）是所有思想的领土，所有可能想法的空间。我们在玩家拥有权习俗中看见了 Locke 产权理论在智域次集合 —— 所有程序的空间 —— 上的暗示。即此“开拓智域”，正如所有新的开放源代码计划的建立者所做的事一般。

Fare Rideau <rideau@ens.fr> 正确地指出玩家们并不是在单纯的思想领土中运作。他断定玩家们所拥有的是程序计划 —— 有目标地专注在物质劳力上（发展，服务，等等），而与名望，信赖度等等连上关系。他因此断定由玩家计划所延展的空间，并非智域，而是一种双重的智域性质，由智域计划拓展的空间。（有位天体物理学家在此同意，在词语学上，正确地可称这个重复的空间为“ergosphere”或“工作的领域”。）

在实际上，智域及工域的分别在本文目地中并不重要。在纯净的思想中，“智域”要存在实在也很难；大概要有柏拉图式的哲学家才会相信了。而将智域及工域分开，也只有在某人希望断定想法（智域的元素）不能

被拥有，但如计划之类的可以被拥有的时候，才会有实际作用。这个问题导致一个智慧财产权理论的问题，远远超过本文探讨的范围。

为了避免困惑，要注意到智域或工域跟整个虚拟的电子媒体被称为“cyberspace”（大部份玩家伪装的地方）是绝不相同的。产权规范法则在此是完全不同于物质阶层的 —— 基本上，拥有媒体或机器属于“cyberspace”便拥有“cyberspace”的一部份。

Locke 的结构强烈建议开放源代码玩家观察这些所为的习俗以持有对付出的代价所得的回报。这些回报必然要比开拓计划所付出的更重要 —— 花在维护“头衔传承”的版本历史上的代价，花在引起大众注意的时间代价，及等待领养孤儿计划的时间上。

再者，由开放源代码所获得的“成果”必然是远超过简单的使用软件，是一些会被分歧所连累或稀释的其它东西。如果就是使用软件这么简单，应该不会有对分歧的禁条存在，而开放源代码拥有权不会像土地产权一样的类同。事实上，这样的世界确实（即使用为唯一成果）在现有开放源代码版权中存在。

我们可以现在就将一些可能的成果候选者扫除。因为您不能够强制网路连线，找寻在那里的存在的力量。同样地，开放源代码文化不能构有任何像钱或内部稀有经济的类同品，因此玩家不能够追求与物质富裕相关的任何事物。

在开放源代码活动中，有个方法可以协助人们变得更富裕。偶而，某人在玩家文化中所获得的名望可在现实生活中获得经济上的重大好处。它可以使您获的更好的职业收入，或者顾问合约，或者书约。

这样的副作用对大多数玩家来说是很少见的；这可独立做为一个解释，既使我们经常见到玩家们抗议说，他们所为是出发于理想或爱，而不是为了钱。

不论如何，调停这样的经济副作用很值得加以检验。以下我们会看到了解在开放源代码文化自身的名望动力，很可以自我解释。

4.6 玩家文化即礼物经济

要了解名望在开放源代码文化中的角色，我们需要从历史移到进一步的人类学及经济，并检验交换文化及礼物文化之间的不同。人类对社会地位的竞争有天生的驱动力；它与我们的演化史息息相关。在农业发展之前，90%的历史，我们的祖先生活在游牧打猎的生活形态中。地位高个体获取较建康的伴侣并取用最好的食物。这个透过地位来表达自我的驱力表现在多方面，大致上是由于生存货物的缺乏所致。

大部份方式中，人类采行组织的方式来获取稀有货品及所需。每种方式都有其获取社会地位的方式。

最简单的是命令阶层。在命令阶层中，稀有品的分配是由中心权威来完成，并以武力做为后盾。命令阶层所达程度很有限；他们在组织成长时变得越来越兽性而无效率。基于这个理由，在大家族中的命令阶层往往在不同大型经济形态中变成寄生虫。在命令阶层中，社会地位主要是透过取得高压力量来达到。

占我们社会主导地位的是交换经济。这是对稀有品的复杂采用形式，不像命令阶层模式，它成就很高。稀有品的分配是透过分散的交易及志愿合作（事实上，竞争野心是产生合作行为的主要效应）。在交换经济中，社会地位主要是透过控制用以交易的东西来决定（不一定需要是物质的）。

大多数人在精神上都有受到以上两种模式的影响，并决定如何与他人互动。政府，军队，及组织罪犯（举例而言）皆为在我们称为"自由市场"的广泛交换经济下的命令阶层寄生虫。不过，其实还有第三种模式，在根本上完全与两者不同，而且除了人类学家以外，一般人并不知晓；即礼物文化。

礼物文化不是因为稀有而采用而是丰富才采用。它们是在没有物质稀有问题，而生存必须品丰富的族群中掘起。我们可以观察到礼物文化在气候温和及食物丰足的原始生态系中发展。我们可以看见他们是在我们社会阶层中的确定地位，特别是那些商业中富裕的族群。

丰富使得命令关系结构变得很难维系，而交换关系则变程序无意义的游戏。在礼物文化中，社会地位不是由您能控制多少而决定，而是由您给出多少来决定。

因此这是瓜基乌图酋长的冬季赠礼舞会。这是百万富翁精心的大众慈善行为。这是玩家长时间付出的生产高品质开放源代码。

透过这样的检验，开放源代码玩家社会很明显地是个礼物文化。在其中，没有严重的“生存必须品”的短缺 —— 硬碟空间，网路频宽，电脑速度。软件是免费地分享的。这样的丰富产生一种状况，即唯一的竞争是同跻间的名望。

不过，这样的观察并不足以完全解释整个玩家文化的特性。cracker d00dz 也有相同于礼物文化的特质，但他们的行为是大不相同的。其族群文化的智力在玩家中算是很强及独有的。他们聚集秘密，而非分享；一个人需要投靠某个 cracker 组织来获取破解的软件，而非取得如何破解的技巧。

（译注：很多人认为 Hacker 及 Cracker 之间没有明显的界线。但实际上，这是错误的观点。Hacker 及 Cracker 不但可以很容易的分开，而且可以分出第三群 —— "海盗"Internet Pirate 出来，一般大众认定的"破坏份子"，事实上是这第三种。Hacker 及 Cracker 都有明确的定义，要发表有关 Hacker 及 Cracker 之间的评议之前，最好要详细调查一番，否则招惹这两群技术高明的族群都不是好受的事。比较容易判断的方式，"Hacker 从来不自称 Hacker；Cracker 会自称 Cracker；自称 Hacker 的不是 Hacker；自称 Cracker 的不见得是 Cracker；被确认为

Hacker 称为 Hacker 的，是 Hacker；而 Richard M. Stallman 是 Hacker 圣者；"相信大家应该可以看出来，为何大众对 Hacker 及 Cracker 会有错误观点的由来，Pirate 利用这样的漏洞来污染整个玩家文化在大众的观点。）

在此所展示的，看来不是很明显，是礼物文化运作方式不只是一种。历史及价值是很重要的。我已经将整个玩家文化做了大体的摘要；现有行为状态并非神秘的。玩家透过选择他们竞争的形式来定义他们的文化。而本文的接下来的部份将会检验这些形式。

4.7 驾御之乐无穷

在做这个“名望游戏”分析的同时，顺便说一下，我并非有意诋毁或忽略单纯的设计美妙软件并使其工作的艺术上的满足。我们都经历过这样的满足并爱上它。这些重大动机，对那些并非是那么执着的人们来说，从一开始就不会成为玩家，正如不爱音乐的人不会成为作曲家一样。因此，我们应该要考虑到另一种玩家行为模式，即以技艺为单纯的主要动机。这个“技艺”模式应可解释玩家传统在技艺的机会上及结果的品质上的最大效果。这是否建议与“名望游戏”模式相冲突或不同的结果呢?

（译注：在电脑科学中，有个很大的争议，即“软件设计是艺术”及“软件设计是工业”的争执。最著名的“软件艺术家”可说是 Knuth 这位大师。我们在此所见到的，便是“软件艺术”。“软件工业”在电脑科学上，可由软件工程来代表。）

不见得。在检视“技艺”模式中，我们回到同样的问题上，玩家圈子运作像礼物文化。如果品质没有尺度来衡量，要如何将品质最佳化呢?如果稀有品经济不运作，除了同跻评估外有什么度量呢?这显示出任何技艺文化最终要架构在名望游戏中 —— 而事实上，我们观察到正是这个动力，在中世纪同业公会中，推动许多历史上的技艺文化。

从一个重要的观点来看，“技艺”模式比起“礼物文化”来得弱；其自身，并没有帮助我们解释在本文开始所说的矛盾。

最后，“技艺”动机本身可能如我们所假设的，在心理上不会离名望游戏太远。想想看您美妙的程序被锁在抽屉中，而不再被使用。现在想像它被许多人很满意的有效使用。那一个梦想比较满足您呢?

不过，我们将会注意这个技匠模式。它直觉地揭露了许多玩家，并相当完美地解释了许多的个体的行为。

在我发表了本文的第一个版本后，许多匿名的回响者建议：“您没办法用获取名望的动机来工作，不过名望是当您把工作做好，而获取的真实回报。”这是个微妙而重要的观点。名望诱因可继续运作，不论工匠是否关心到它们；因此，最终，不管玩家是否了解其自身的行为是名望游戏的一部份，他的行为都会被这个游戏所修饰。

4.8 名望的多面性

在每个礼物文化中，有许多理由表明同跻名望是值得付出的：

第一，最明显的，同跻之间的好名望是基本的回报。我们是为此而活，这是个我们在前面已触及的革命性动力。（许多人试着将驱策他们的动力，由名望换成其它升华的形式，而与同跻之间没有明显的关连，诸如“荣耀”，“建全伦理”，“信仰”等等。；这些并没有改变其中的机制。）

其次，名望是（在单纯的礼物经济中，是唯一的方法）吸引注意力及与他人合作的好办法。如果一个人有雅量，智慧，公平处置，领导能力，及其它的好品德，这将会非常有力地说服其他人来一起共同合作。

第三，如果您的礼物经济与交换经济或命令结构纠缠关联，您的名望将会溢满并使您获得更高的地位。

在这些一般理由之外，玩家文化独特的状况造成名望比"真实世界"礼物文化更有价值。

主要的“特例”是某人送出的手工艺品（或者，换另一种方式来解译，是明显地看得出是一个人下足精力及时间所制的礼物）是非常复杂的。其价值很明显地跟物质礼物或交换经济金钱完全不同的。要客观地区别出好礼物及坏礼物是更加困难的。所以，赠与者企图的成功是微妙地由同跻间的风评来决定。

另一种特质是相对纯正的开放源代码文化。大部份礼物文化是由这些元素所组成的 —— 由交换经济关系，诸如交换奢侈品，或是由命令经济关系，诸如家族或部落族群。在开放源代码文化中，并没有类同的方式存在；因而，除了透过争取同跻名望之外别无它法。

4.9 拥有权及名望诱因

我们现在准备将之前的分析放在一起，成为一个连续而完整的玩家拥有权传统。我们了解到这是由进驻智域而来的；它是在玩家礼物文化中的同辈名望，及其所附带的副作用。由此这样的了解，我们可以分析 Locke 在玩家圈子的产权习俗，是一种名望诱因的最大化；这可确保同跻名望可由该获得的人取得，而不会漏到其它地方去。

我们以上所观察到的三个禁条，在这个分析下，就很有点道理了。在某些状况下，有些人胡搞，有些人的名望会遭到不公平的处理；这些禁条（及相关习俗）试图避免以下这些发生。

分歧计划是不好的，因为它将过去的贡献者置于牺牲名望的危险中，而他们只好在分歧后，继续在两个案子中都活动。（一般来说这会变得很混乱而且难以实践。）发行一个游离修补版会使拥有者受到不公平的名望危机。就算正式版是完美的，拥有者也会被这些游离版中的臭虫发出的高射炮射中（但请见[RP]）。偷偷摸摸地将某人的名字将计划中移除，在文化条文中，是大恶不赦的事。他偷取受害者的礼物并且变成小偷的礼物。这三项禁条的违反者伤害整个开放源代码团体及受害者本身。这还迁连他们会伤害到整个团体，导致降低潜在贡献者对礼物/产品会收到回报感受的可能性。很重要的对其他两个禁条有其它可能的解释。

第一，玩家会对分歧计划表示厌恶，并且悲叹重复的工作，在未来，需要在所有几个子计划中付出代价。他们也会观察到分歧会将共同开发者团体分开，导致两个子计划有较少于原有计划的智群在工作。

有位回响者指出，在分歧的工作中，很少有见到能够有多于一个分歧能够存活，并且在长期享有“市场占有率”。这加强了所有参与者参与的诱因，互相合作并避免分歧，因为很难事先知道，谁将会站在输的一边，而看到他们辛苦的工作不是整个消失就是逐渐漠落。

不喜欢游离修补通常可被解释为造成错误追踪的复杂度增高，而且会造成维护者增加他们的工作量，而他们自己通常就已经有很多自己需要做的事了。

这些解释都可以考虑为正确的，而他们确实在 Locke 逻辑下的拥有权理论下行得通。但同时理性吸引力，他们无法解释为何当这些偶而发生的违反禁条事件，会导致如此在情绪及领土上的反感 —— 不只是在受伤的一方，而且旁观者及观察者也会反应严厉。冷血地关心工作量及维护量的重复并不足以解释这些观察者的行为。

对于第三个禁条。除了名望游戏的分析以外很难解释。事实上这项禁条很少被分析为“它不公平”的结论，则揭露的其本身的问题，而我们会在下一节看到。

4.10 自我的问题

在本文的一开始，我提到了文化的潜意识适应知识通常是起源于其意识形态。一个很大的事实范例即 Locke 的拥有权传统被广泛地使用，虽然事实上，它们违反标准版权所陈述的条文。

我在与玩家们讨论到名望游戏的分析时，观察到另一个有趣现象的范例。即许多玩家拒绝这个分析，并且强烈反对承认他们的行为是起源于渴望同跻名望的动机，或者我在此不太精确地标为，“自我满足”。

这展现了玩家文化有趣的一面。很清悉的大家都不信任并鄙视自我中心及自我出发的动机；自我奖励被残酷的批评，既使该团体事实上对此可获取许多有利之处。很大一部份，事实上，该文化的“大大”及部族长者被要求要谦逊并幽默地自我贬低，以便维护自己的地位。这样的态度几乎使自尊呼之欲出用以编织解释整个诱因结构。

在大体下，可以确认的，这一般起源于欧裔美国人对“自我”的负面态度。整个玩家圈子都告诉自己，渴望自我满足是很坏的动机（至少是不成熟的）；自我只是对追求女性时的可容忍的怪僻，而且通常被认为是个精神病状。只有在升华的形式或伪装的形式如“同跻名望”，“自尊”，“内行气派”或“成就的自豪”才是可被接受的。

我可以写一整篇论文来讨论这个文化传承中的不良部份的根，我们可以发见，相信我们有真正的“无私”动机（违背心理学及行为的证据），将会造成自我迷惘的巨大伤害。或者我可以，如果 Friedrich Wilhelm Nietzsche 及 Ayn Rand 没有已经将整个解构“利他主义”成为许多不同的个别兴趣的工作完成。

我并非在此做道德哲学或是心理学，因此我在此仅观察一个以认为自我是邪恶的观点所造成的一些伤害，就是：它在情绪上造成许多玩家很难以清析地了解自己文化的社会动力！

但我们并没有在此线调查中完成。在玩家文化中对于自我出发的行为的禁忌是如此的深，玩家要怀疑是否该采用其它的可行方式。当然了这样的禁条在其它礼物文化中并没有这么强，像剧场或巨富之间的同跻名望！

4.11 人性的价值

在建立了名望是玩家文化回馈机制的中心之后，我们现在需要了解为何它看来如此的重要，然而确依然是半隐密地并且不受承认。对比的是海盗文化的指令式。在那个文化，追求地位的行为是露骨的甚或炫耀的。这些 crackers 追求宣称释放"zero-day warez"（在原版软件释放的当天破解该软件并再发行）但却对如何做到的闭口不提。这些魔术师并不喜欢把密技公开。因此，这导致了 cracker 文化的知识库在整体上进步缓慢。

（译注：社会大众及 Eric S. Raymond 对骇客文化有所误解，在整体知识库的发展上，骇客文化透过结帮来进化，因此骇客文化有点像帮派文化。其整体知识库的发展上，步调并不慢。）

在玩家团体中，对比来说，一个人的工作代表一个人的表达方式。这是非常严格的英才制度（最好的工匠胜利）而且有很强的意识，即品质会说话。最大的吹牛就是程序“能跑”，而任何有能力的程序设计师会看到的是好东西。因此，玩家文化的知识库增加快速。

对抗自大驱使的装模作样的禁条导致增加生产力。但那是个次级效应；在此受到保护的是该团体同跻评估系统的资讯品质。即，自吹自擂或自我放大是被镇压的，因为在创造及合作的行为者中，它就像杂音一样搞乱实验室中的美妙。

玩家文化传播礼物的媒介是无形的，其通讯频道很难表现情绪的细微差别，而面对面的成员接触是唯一的例外。这使得该文化比其它礼物文化对杂音有更低的容忍力，而需要花不少时间来解释大众谦逊需要部族长者。

谈吐谦逊对一个有志成为成功计划维护者的人也是有用； 他必须要使该团体相信他有很好的判断能力，因为一个维护者的大部份工作是判断其他人的工作。谁愿意贡献给那些无法判断其工作品质的人呢，或者谁愿意贡献给那些会吞食计划成果的人呢? 潜在的贡献者希望计划领导者有足够的谦逊及品味，当客观时机到来时，有能力说，“对，这比我的版本工作来得好，我会用它” —— 并且将成就让该收到的人收取。

另一个在开放源代码世界需要谦逊行为的理由，您很少希望给大众认为计划已经“完成”的映象。这可能会导致潜在的贡献者觉得不须要贡献。要将您的工作成效最大化，需要对您程序的状态谦逊。如果您吹牛自己的程序，然后说“没什么价值，它对 x，y，及 z 无效，因此它并不好”，然后很快的自己偷偷补上 x，y，及 z。

最后，我自己亲眼见到有些玩家领导者自鄙的行为反映了害怕成为崇拜的教首。Linus Torvalds 及 Larry Wall 两者的行为都很明显地想要避免成为这样的对象。有一次跟 Larry Wall 一起吃晚餐，我开玩笑“您是在场的头号玩家 —— 您选餐厅”。他畏缩了。这样做是对的； 无法分辨他们领导的分享价值会搞砸一个很好的团体，这是个他及 Linus 都无法忽略的一点。另一方面来说，大部份玩家喜欢有 Larry 的问题，如果他们有办法自我承认这一点。

4.12 名望游戏模型对整体的密切关系

名望游戏的分析有些不是很明显的其它关连。许多都是从建立一个成功的计划可获得比与现有计划合作获得更大的名望而来。一个计划如果有许多创新，也会获得许多的名望，相反于“me，too”的对现有计划的持续改善。另一方面来说，只有作者了解的软件或是需要非入门者的，在名望游戏中，通常贡献一个现有计划比自己建立一个来得受人注目。最后，要与现有成功的计划相竞争比填补一个空缺的壁灶来的困难的多。

因此，有个与邻居的最佳距离（类同计划间的竞争）。太接近就会有其中一个产品会变成“me，too！”的有线价值，一个贫乏的礼物（其中一个可能最好放弃）。离得太远，没有人有能力使用，了解，或察觉到另一位的付出的关系（同样，贫乏的礼物）。这产生一个在进驻智域中的型态，如拓荒者散播在实体边疆上 —— 并非散乱的，但很像是散乱的碎形波。计划倾向开始于在边疆的范围填补空缺。

有些非常成功的计划变成“目录杀手”； 没有人愿意再去与那些已经建立起来的计划竞争，因为对玩家来说实在太难。大家最多是发现自己所须，而在这些成功的计划中，新增附加功能。典型的“目录杀手”范例是 GNU Emacs； 由 1980 年代开始，它的多样化功能就已经填补整个程序设计编辑器的目录生态系，没有人再去试图写一个新的出来。因而，人们只写 Emacs modes。

整体来说，这两种倾向（缝隙填补及目录杀手）在时序上的发展可用来预测未来的倾向。在 1970 年代，大部份开放源代码都是玩具或范例。在 1980 年代，则是在开发工具及互联网工具。在 1990 年代，这项行动移向作业系统。在每个案例中，当前一个问题接近被处理掉时，一个新而更加复杂的问题层次被开始攻下。

这种倾向在不远的未来有点有趣的关连。在 1998 早期，Linux 看来很像是“开放源代码作业系统”目录的杀手 —— 所有为其它竞争的作业系统，现在都已经开始为 Linux device drivers 及 extensions 写作。而大部份这个文化所想要的开放源代码的低阶的工具都已经存在了。还有什么吗?

应用软件。当 2000 年接近，看来预测开放源代码发展的能力会逐步迈向最后的处女地带 —— 为非技术人员所设计的程序 – 是不过份的。稍早的指标是 GIMP 的发展，Photoshop–like 的影像处理软件，这是个开放源代码第一个主要对 end–user–friendly GUI 的界面软件，并可被考虑为在过去十年中足勘与商业软件相比的应用软件。另一种则为流言四传的应用软件工具 KDE 及 GNOME。

最后，名望游戏分析解释了屡次引用的格言，即您无法用自称玩家来变成玩家 —— 当其它玩家称您为玩家时，您才是玩家。“玩家”，以这种观点来考量，是某人显示出了（透过贡献礼物）他或她有技术能力及了解名望游戏的运作。这样的评断大半是一种知觉及传承，并且只有身在文化中之辈才会意会。

4.13 智域特质及领土于动物行为学的影响

了解产权习俗的结果会帮助我们从另一个角度来看它； 即动物行为学，特别是领土动物行为学。 产权是一种动物领土观念的抽象化，是演化来用以降低同物种之间暴力发生。透过划出界线，并尊重其他的界线，一苹狼如果与其它发生战斗，可能会导致它受伤或致死，并降低生存繁演的机会。

类同的，在人类社会中的产权功能是避免人际间的冲突，透过设立疆域清楚地分别合平的行为及侵略的行为。有时大家喜欢将人类产权看成是个抽象的社会传统，但这完全是错的。任何人有苹狗就会知道，当狗对陌生人接近时的吠叫，是一种介于动物领土及人类产权之间的连续性。我们本地的狼兄地就本能地比许多人类政治理论家知道的更清楚。

宣告领土（就像制造领土）是个表示实现愿望的行为，一种表示在什么样的界线下将会防御。社会支持产权宣告是一种减少阻力及促进合作的行为。这些比栏杆或狗吠更抽象的“宣告产权”依然是有效的，甚至既使它只是在 README 档案中简单的描述计划维护者的名字，也是有效的。它依然是个抽象的领土，而且（就像其它的产权形式）我们本能式的产权模式是由领土演化而用来解决冲突解决。

这个动物行为学分析，第一眼看来很抽象，很难以跟真正玩家的行为关连起来。但它有一些重要的结果。其中之一解释了全球资讯网的大众化，特别是解释了为何开放源代码计划有个公开网站看来会比没有网站的来得重要。

客观地想，这看来很难解释。与原来在原创及维护上所付出的力量来说，一个网页很简单，因此很难想像一篇网页是个重要或非凡的付出。

网站的本身功能并不足以解释一切。网页的通讯功能可以混合 FTP 站，mailing list，及 Usenet。事实上一个计划的平日通讯很少是完全透过网站来通讯的，反而 mailing list 或 newsgroup 更重要。那么，为什么公开网站会变成是计划的中心呢?

“home page”这个隐喻提供了一个重要的线索。当在建立开放源代码计划时，它是个在智域中宣告领土的行为（而传统也认知为此），它在心理层次上并非十足强制的。毕竟，软件并没有位置的本质，而且可以立即复制的。它在我们本能的“领土”及“产权”记号上是可同化的，不过，要在付出一点代价后。

一个计划的网页表现了抽象的进驻可能的计划领域空间，透过全球资讯网表达其王国的“本土”领域。从智域降到“cyberspace”并没有让我们通过真实世界的栏干及狗吠，但它确实保障我们宣告产权的力量，并使我们对领土更加感到安全。而这也是有网页的计划看起来更加地“真实”。

这个动物行为学的分析，也鼓励我们更加详细的检验，在开放源代码文化中，这个处理冲突的机制。他带领我们预期，除了将名望诱因最大化以外，拥有权传统将会在避免冲突及解决冲突上扮演一个角色。

4.14 冲突的起因

在开放源代码软件的冲突大致有以下主要议题：

由谁来做下计划的决策?

由谁来接受荣耀或谴责，承受什么样的?

要如何减低负担，特别是在复杂的错误追踪中避免劣质版本?

技术上来说，什么是正确的事?

如果我们看看“什么是正确的事”议题第二眼，它将应该要消失掉。对任何这样的问题，要看是否有个客观的方式来让大家都接受。如果有，那游戏结束大夥都赢。如果没有，那么就变成“谁来决定?”。

所以，这三个问题的冲突解决理论将可解决一个计划的三大问题：（A）决定设计时的闲扯要何时而止，（B）要如何决定那些贡献者接受荣耀及如何授与，及（C）如何保持一个团队不会变成多重分歧。

拥有权的角色在解决（A）及（C）的问题上是很清楚的。惯例确保计划拥有者可做下决策。我们在以前曾见过，惯例会对分歧者施加压力并稀释其拥有权价值。

注意这些惯例是合理的，甚至对某些不关心名望游戏的人来说都很有帮助。我们可从检验纯正的“工匠”模型的玩家文化来看出。在此观点下，这些惯例很少与稀释名望诱因有关，而比较保护工匠在选择眼光上的权益。

技匠模型并不足以解释关于（B）的玩家惯例，谁做了什么而接受什么荣耀（因为对一个纯正的工匠来说，并不关心名望游戏，因此没有其它动机可循）。要分析这一点，我们需要将 Locke 理论带到另一个新高点，并检验冲突及运作的权力在计画中及计划之间。

4.15 计划结构及拥有权

一般的个案来说是一个计划有一个单一个拥有者/维护者。在这种状况下没有可能会有冲突。拥有者可做下所有的决定，拥有所有的益处，及受到所有的谴责。唯一会冲突的问题可能是继承者 —— 当旧的拥有者失去兴趣或不见了，由谁来当新的拥有者。该团体亦对避免分歧很感兴趣。这些兴趣都由文化规范所表达，即当一个拥有者/维护者对该计划不再有兴趣，应该公开地将名衔交给下一位。

非一般的最简单的例子，是许多位共同维护者在一位“仁慈的独裁者”下工作。在共同计划中习惯于这个模式；我们在大计划像 Linux kernel 或 Emacs 中看到，及解决“由谁来决定”的问题，这看起来不会比其它的方式来得糟糕。

典型来说，一个仁慈的独裁者组织由一个拥有者–维护者组织，建立者吸引贡献者演化而来。即使拥有者依然在位，它也有计划部份的功劳由谁来获取的争执存在。

在这种状况下，习俗有义务由拥有者/独裁者来公平地处理贡献者的贡献（例如透过在 README 或历史档中提及）。在 Locke 的产权模型术语来说，这意味透过贡献一个计划来获取部份的名望（正面或负面的）。

追朔这个逻辑，我们见到“仁慈的领导者”并非真正拥有整个计划。虽然他有权力做下决策，他事实上是透过交易名望来交换其他人的工作。这类比就像佃农耕种，是很难抗拒的，除了有时候当某个贡献者不再活动，他的名字依然继续获得好处。

一个仁慈的独裁者计划加上许多参与者，他们倾向于发展出两族贡献者；一般贡献者及共同发展者。一个典型的途径，变成共同开发者，可获得计划较大部份的责任。另一种是以“lord high fixer”的角色，专长在修

正许多臭虫。以这种方式或其它种类的，共同开发者在整个计划中，是那些以透过投资时间并有重大贡献的贡献者。

次系统拥有者角色在我们的分析中特别重要并且要求更进一步的检验。玩家喜欢说“权威要负责”。接受维护责任的共同作者可获得一个次系统的掌控权及所有相关部份的计划，只受到计划领导者的修正（可说是建筑师）。我们观察到这项法则有效地圈起 Locke 模型的计划产权，并且同样地有其它产权界线避免冲突的效用。

传统上，“独裁者”或计划中的领导者及共同作者，是预期要与共同作者在关键决定上做协商。特别是该决定与共同作者所拥有的次系统有关（也就是说，有投资时间并负责任）。一个有智慧的领导者，认识出计划内部产权界线的作用，是不会轻率地影响或反转由次系统拥有者的决定。

有些很大的计划完全不吃“仁慈的独裁者”这一套。一个方法是转共同作者成为投票委员（像 Apache）。另一种为轮换独裁者，即控制权由一位成员轮换到另一位资深成员（为 Perl 组织所采用）。

这样复杂的安排通常被考虑为不太稳定及复杂。很明显地这些困难的认知，为这些委员及设计者本身所了解；这些问题是玩家文化中，大家清悉地了解的。不论如何，我认为有些玩家对委员会或轮换独裁者组织心中不舒服，因为它与过去 Locke 模型的思考大不相同。在这些复杂的组织中，不论是做拥有权或是名望回报的计算是有问题的。很难看出其内部界线在那里，除非高度的协调及信任，否则很难避免冲突。

4.16 冲突与冲突解决

我们曾经见过在计划中，角色复杂度的增加是由设计权威及部份产权的分布来表现。这是个有效的方式来分散诱因，同时也稀释了计划领导者的权威性 —— 更重要地，它稀释了领导对潜在冲突镇压的权威性。当设计上的技术争执看来会导致互相残杀的冲突时，它们很少是起因于吵架的。这些通常都是由个别的领域的权威来负责解决。

另一种解决冲突的方式是“资深” —— 如果两个贡献者或贡献群有所争执，而争执无法客观地解决，并且都不拥有该争执的领域，曾经为这个计划付出较多的一方胜利（即在计划中，拥有较多产权的一方胜利）。

这些法则通常都足以解决大部份计划的争议。当这些不生效时，计划领导者的命令通常可解决。争议在通过这两关之后还存在的很罕见。

冲突除非在两个关键上都指向不同的方向的时候（"当权者要负责"及"长者胜利"）是不会变得严重的，而计划领导者的权威是很微弱或不发生作用的。这种状况最明显地是在继承权争执时，而领导者不在时。我曾见过这样的战斗一次。非常地丑陋，痛苦，折磨，只有当所有参与者都变得精疲力竭时，将它交给外人来处理，我锺心地期望不要再看见任何这一类的争执。

最终，当所有冲突解决机制，在所有玩家社团中都失效时。唯一剩下的机制只剩吵架及躲避 —— 对那些拒绝合作，破坏传统的人们的公众审判。

4.17 薪传机制及与学界的关联

在这篇文章的先前版本曾提出一个研究问题：这个团体究竟是如何告知并指导成员符合习俗?这些习俗是否是在半知觉状态下，不言自明的或自我组织起来的?是否是由范例所导引的?是否由明白的教条所授?

由教条所授明显的很罕见，因为至今为止只有少数几条该文化的规范曾经有人提出过...

大部份的规范都是由范例所导引的。用个很简单的例子，在所有软件发行中的规范中，应该都会有个 README 或 READ.ME 档案，用以说明这个软件的简略说明。这个传统至少从 1980 年早期就已经建立起来了，但至今尚未有人将本条写下。这是由观察许多软件发行而来的。

另一方面，有些玩家习俗是自我组织起来的，尤其是一旦有其中的份子了解到这个基本的（或者不自觉地）名望游戏。大部份玩家不需要被传授以我在第三节所列的三个禁忌，或者可说这些是不言自明的。这个现象导出更进一步的分析 —— 而我们可能可以在探讨玩家在探索该文化智库的过程中找到解释。

大部份文化透过暗示（更精确一点“神秘”，透过宗教式或神秘性的方法）做为薪传机制。这些秘密对外来者是不揭露的，但可预期到可被热诚的新手所发现或理解。要被内部所接受，他需要展示他对该文化的神秘的了解及学习程度，以接受认可。

玩家文化对此是不凡地警觉，并且大量使用这些暗示及测试。我们可在至少三种层次的过程运作中找到：

犹如密码式的神秘。拿个例子，在 USENET 新闻讨论群中有个 alt.sysadmin.recovery 有个很明显的秘密；您不可能在不知道之前就提出问题，而知道该秘密您才会被准许进入。其中的老手对揭露这个秘密有很严格的禁忌。

对特定技术秘密的基本要求。一个人必须要在送出礼物之前先吸收大量的技术知识（例如，必须要至少了解一种主要的电脑语言）。这个隐藏线索由大处至小节都有作用，犹如做为品质过滤（诸如抽象思考力，坚持，及精神力量）的功能用以发挥该文化的力量。

神秘社会内容。要与该文化迁连上关系必须要参加特定计划。每个计划都是活生生的玩家社会文化范例，即贡献者必须要调查并了解该团体的社会及技术以便能有效参与。（具体来说，一般的方式是透过阅读该计划的网页或邮件）透过这些计划团体，新手体验到有经验的老玩家的社会行为。

在透过探索这些秘密的过程中，这位未来的玩家学到丰富的知识，而使得这些禁条及其它习俗不言自明。

有些人可能会偶然议论玩家礼物文化结构是其中心秘密。一个人在掏心吐胆的展现其对名望游戏及其暗藏的习俗，禁忌及使用的了解之前，是不被考虑为受传承的。但这不重要；所有文化都对其未来的参与者有这样的要求。更进一步地说，玩家文化表明对其参与者没有野心 —— 或者，至少，没有人因为我揭露这些而来跟我吵架!

有大量的人对本文回应指出玩家拥有权习俗看来十分接近（而且很可能直接源于）学术界的业务，特别是科学研究团体。研究团体有很类似的问题，特别是在开采潜在可能的思考领域，及展现非常类似的对问题可行的解决途径上，使用同跻检视及名望。

既然许多玩家曾经在学界打滚过（通常都是在大学时学会玩电脑），在了解过玩家文化后，将玩家文化与学界做会类比自然是不稀奇的。

玩家“礼物文化”有明显地与学界平行类同的特质。一旦研究员取得终身职，他不再需要担心生存问题（的确，终身职的观念可回溯至早期的礼物文化，即“自然哲学家”基本上是富裕的绅士，时间满满可奉献于研究上。）在生存的危机解除后，名望成了驱动的目标，也就是在期刊或媒体上，鼓励分享新点子及研究成果。这造成客观而正面的功能，因为科学研究，就如玩家文化一般，非常依赖“站在巨人的肩膀上”，而不需要一再地重新发现一些非常基本的原理。

有些人则进一步推论玩家文化只是研究团体风气的一种反射，而且已经几乎到达相当程度。这可能讲得过头了，因为玩家文化似乎不过是以高中学历的聪明人们所架构起来的!

这里还有一些有趣的可能性存在。我怀疑学界与玩家文化的类同型式，并不止是因为其起源类同，还因为它们所做的事，在自然法则及人类本能的连系之下，而演化出最理想的社会组织。历史的裁决似乎断定自由市场资本主义是整体来说对经济效益最佳的方式；或者，以类同的方式来说，名望游戏礼物文化是在生产（及检验）高品质创造力的工作上最佳的合作方式。

这一点如果是真的话，那就比学术兴趣更加有意义了。因为这提出了一点与教堂观与市集观中稍微不同的观点；即，最终，软件产品工业资本家模式的末日到来，而将被排出竞争，从资本主义开始产生大量的剩馀财富，而导致大量程序设计师生活下饥荒后的礼物文化下的那一刻开始。（译注：换言之，也就是大软件公司自掘坟场。任意垄断操纵软件市场的后果，造成软件设计师无法生存，引发大规模投向以名望为基础的礼物文化中的活动，增强了玩家文化的后盾。我们可以在科技史的发展上发现许多类同现象。）

4.18 结论：由文化到文化规范

我们已经检视了用以控制及规范开放源代码软件产权的习俗。我们已经见到这是如何揭露出在其下的权益特质及与Locke的土地产权理论的关系。我们已经将玩家文化与“礼物文化”关连起来，即参与者透过投入时间，精力，及创造力来竞争名望。我们已经检验过在文化中的有相关的冲突解决分析。下一个合理的问题应该是"为何这些这么重要?" 玩家们并无意识分析而发展出这些习俗，而且直到今日，下意识地遵守这些习俗。这些有意识地分析并没有立即明显地有任何的实用性 —— 除非，或许，我们可以进一步推展这些描述成为处方，并演绎出一些改善这些习俗功能的方法。

在英式美国一般法传统下，在玩家文化及土地产权理论之间，我们已经发现一个相当合理的类比。历史上来说 [Miller]，欧洲种族文化发明了这个传统来解决他们的争执解决系统，即由未书写的，下意识的习俗系统到明白地由种族中智者所记下的惯例法规 —— 然后最后白纸黑字写下。

或者，既然我们的人口逐步上扬，而对所有成员的薪传越来越困难，是为玩家文化做点类同的事的时候了 —— 发展一套对解决各种争议的实用"程序码"（written code），可增加在开放源代码计划的实力，及一套仲裁传统，即团体中的资深成员可做一些争执调解。

本文中的分析已经将这样的"程序码"大纲划出，将过去暗示性的变成明白书写的。不会出现在以上没有出现过的; 他们需要由各计划的建立者或拥有者志愿采用。也不会完全地苛刻，因为在该文化的压力会随时间而改变。最后，要将这样的"程序"付诸实现，他们必须要反射出该玩家部落的广泛接受。

我已经开始进行这样的"程序码"，可能会叫做"Malvern Protocol"，以我所住的小镇名字来命名。如果在本文中的这些分析逐渐受到广泛接受，我会让大众都可取得Malvern Protocol，用以解决争执范例"程序"。有兴趣批评及发展这套"程序"的团体，或者希望提供一些他们的想法的，都很欢迎与我连络。

4.19 对进一步研究的一些问题

该文化（也是我自己的文化）了解到，不跟着一位仁慈的独裁者的模式是脆弱的。这样的计划大多都失败了。有些则惊人地成功而重要（Perl，Apache，KDE）。没有人真正了解其中的差别在何处。（每个计划的生命力，与其参与者的组织动力生息相关，是个相当暧 的观点，是否真得存在一个组织可重复实行无碍的策略呢?）就观察到的事实而论，我们确实发现到，成功的计划，获取比需要相同工作量的除错及协助成功计划的工作来得更高的名望。这是否是个对相同付出的理性价值评断呢? 或者它是个我们在此所演绎出来潜意识的领土模型所造成的次级效应?

4.20 参考文件，附注，及感谢

20. Bibliography，Notes，and Acknowledgements

[Miller] Miller，William Ian; Bloodtaking and Peacemaking： Feud，Law，and Society in Saga Iceland; University of Chicago Press 1990，ISBN 0-226-52680-1. A fascinating study of Icelandic folkmoot law，which both illuminates the ancestry of the Lockean theory of property and describes the later stages of a historical process by which custom passed into customary law and thence to written law.

[Mal] Malaclypse the Younger; Principia Discordia，or How I Found Goddess and What I Did To Her When I Found Her; Loompanics，ISBN 1-55950-040-9. Amidst much enlightening silliness，the `SNAFU principle' provides a rather trenchant analysis of why command hierarchies don't scale well. There's a browseable HTML version.

[BCT] J. Barkow，L. Cosmides，and J. Tooby (Eds.); The adapted mind： Evolutionary psychology and the generation of culture. New York： Oxford University Press 1992. An excellent introduction to evolutionary psychology. Some of the papers bear directly on the three cultural types I discuss （command/exchange/gift)，suggesting that these patterns are wired into the human psyche fairly deep.

[MHG] Goldhaber，Michael K.; The Attention Economy and the Net. I discovered this paper after my version 1.7. It has obvious flaws (Goldhaber's argument for the inapplicability of economic reasoning to attention does not bear close examination)，but Goldhaber nevertheless has funny and perceptive things to say about the role of attention-seeking in organizing behavior. The prestige or peer repute I have discussed can fruitfully be viewed as a particular case of attention in his sense.

[HH] I have summarized the history of hackerdom at http//www.catb.org/~esr/faqs/hacker-hist.html. The book that will explain it really well remains to be written，probably not by me.

[N] The term `noosphere' is an obscure term of art in philosophy derived from the Greek `nous' meaning `mind'，`spirit'，or `breath'. It is pronounced KNOW-uh-sfeer (two o-sounds，one long and stressed，one short and unstressed tending towards schwa). If one is being excruciatingly correct about one's orthography，it is properly spelled with a diaresis over one `o' -- just don't ask me which one.

[RP] There are some subtleties about rogue patches. One can divide them into `friendly' and `unfriendly' types. A `friendly' patch is designed to be merged back into the project's main-line sources under the maintainer's control (whether or not that merge actually happens); an `unfriendly' one is intended to yank the project in a direction the maintainer doesn't approve. Some projects (notably the Linux kernel itself) are pretty relaxed about friendly patches and even encourage independent distribution of them as part of their beta-test phase. An unfriendly patch，on the other hand，represents a decision to compete with the original and is a serious matter. Maintaining a whole raft of unfriendly patches tends to lead to forking.

I am indebted to Michael Funk <mwfunk@uncc.campus.mci.net> for pointing out how instructive a contrast with hackers the pirate culture are. Robert Lanphier <robla@real.com> contributed much to the discussion of egoless behavior. Eric Kidd <eric.kidd@pobox.com> highlighted the role of valuing humility in preventing cults of personality. The section on global effects was inspired by comments from Daniel Burn <daniel@tsathoggua.lab.usyd.edu.au>. Mike Whitaker <mrw@entropic.co.uk> inspired the main thread in the section on acculturation.

5 魔法大锅炉

本文分析了正在不断发展的开放源代码现象的经济基础。我们首先推翻了一些流行的关于软件开发中投资和软件价格结构的神话，给出了一个关于开放源代码协作稳定性的游戏规则分析。我们给出了九种开放源代码开发的可发展模型，其中两种是不盈利的，七种是盈利的。接着我们发展了一种定性的理论，说明什么时候封闭代码在经济上是合理的。然后我们考察了当前市场上发明的几种新颖的开放源代码开发的盈利方法学，包括赞助系统和任务市场的引入。我们最后做出了结论，试着对将来做了一些预测。

5.1 近乎魔法

在威尔士的神话中，Ceridwen 女神有一口巨大的锅，当女神念动只有她自己知道的咒语时，那口锅就变出奇妙的食物。在现代科学中，Buckminster Fuller 提出了一种“短暂化”的概念，认为在早期的物理资源投资越来越多的被信息内容所代替的情况下，技术会变得越来越有效和廉价。Arthur C. Clarke 指出“任何足够高级的技术都与魔法别无二致”，从而把二者联系起来。对很多人来说，开放源代码社区的成功看来就像难以置信的魔法。高质量的软件变得免费，在充满竞争而且资源稀缺的现实世界，这似乎不能继续下去，但是它进行的还不错。要点在哪？Ceridwen 的大锅只是一个小诡计吗？如果不是，在这种情况下，“短暂化”是怎么工作的——女神究竟念动了什么咒语？

5.2 超越高手的才能

开放源代码文化的经验肯定使许多学习过软件开发的人们感到困惑。“大教堂和市集”一文描述了分散协作软件开发是怎样有效的推翻了 Brooks 的定律，产生了使一个独立的工程具有空前可靠性和质量的开发方式。“开拓智域”一文揭示了市集模式开发风格中的社会动力学，这应该用人类学家所谓的“赠与文化”的术语而不是常规的交换经济术语来理解，在这种文化中，成员在做出贡献大小方面竞争。本文中我们将开始推翻一些流行的关于软件生产经济学的神话；然后对“大教堂和市集”和“开拓智域”两篇文章进行经济学、搏弈论和商业模型领域的分析，发展一种新的概念工具，来理解开放源代码开发者的赠与文化在交换经济里也可以继续下去的理由。

先不岔开话题，沿着上面的线索分析，我们需要抛弃（至少要暂时忽略）在"赠于文化"层次上的分析。“开拓智域”中赠于文化的存在是基于生存所需要的物质资料极大的丰富，以至于社会交换已经不很重要的环境里；这种分析虽然在纯粹的精神世界中非常有说服力，但针对现实生活中大多数开放源码开发者实际所处的综合经济环境来说，这种解释则显得有些无力。对许多人来说，社会交换仍然是他们努力工作的驱动力，但是已经渐渐失去了吸引力。必须在资源匮乏的经济学中为他们的行为找到足够的理由，才能使这些行为在物质资料丰富的赠于文化中得以立足。

因此，我们现在将（从整个资源匮乏经济学领域）思考维持开放源码开发的协作和交换模式。在分析的过程中，通过深入剖析和列举实例，我们同时也就回答了那个非常实际的问题：“我如何通过开放源码来赚钱？”。不过，这个问题是根据与软件开发本质相悖的普遍软件开发经济模型而提出的，首先我们需要展示一下隐藏在这个问题之后的许多思维误区。

（在展开分析之前还有最后一个需要说明的是：本文中对开放源码开发模式的讨论和提倡，不能被理解为对封闭源码模式的彻底否定，也没有反对现有的软件知识产权体系，更不是对"共享"的无私呼吁。虽然开发源码开发团体中的一些人仍然热衷于这些讨论，但从“大教堂和市集”发表以来，经验已经清楚的表明这些争论没有必要的。重要的是开放源码的开发模式和经济效益能够制造出质量更好、可靠性更高、成本更低、可以选择的方案更多的好产品来。）

5.3 制造业的错觉

我们需要注意的是计算机程序和其他类型的工具和资本货物一样，都有两种经济价值：使用价值和销售价值。

程序的使用价值就是它作为工具的经济价值；销售价值是它作为作为商品的价值。（用经济学的专业说法，销售价值是产品最终价值，使用价值是产品中间价值）

当大多数人说到软件产业时，总是按照拥有下列特性的“工厂模式”经济来分析：

1. 大多数开发者的劳动由销售价值的收入来支付
2. 软件的销售价值与开发成本（例如，功能复制所需的资源花费）和使用价值成一定比例

换句话说，人们有很强的思维惯性去假定软件具有标准工业品的特性。但是这两个假设都错了。

首先，编写用于出售的代码只是编程行业的冰山一角。在微机世界前期，大家普遍认为世界上 90% 的代码在银行和保险公司内部编写。这虽然已经不再是事实 —— 现在其他行业也越来越加大了软件开发的力度，金融行业所占的比例从而下降 —— 但是短期内我们仍将会看到大约 95% 的代码是公司内部编写。

这些代码包括大多数为中等或大规模公司所定制的 MIS，金融和数据库软件。包括象设备驱动这样的专业技术代码（几乎没有人靠卖设备驱动赚钱，这一点我们将会在后面讨论）；包括日益增长的数控机器的各种嵌入式代码 —— 从机械工具和喷气客机、汽车、微波炉甚至烤面包炉。

大多数这种内部代码与其环境集成在一起，复制和再利用十分困难（不论环境是商业办公室的程序套件还是联合收割机的加油系统）。因而一旦环境变化，需要做许多工作使软件与之同步。

这种工作称为“维护”。任何软件工程师或系统分析员都会告诉你这就是程序员的大部分工资的来源（超过 75%）。因此，大多数程序员工时花费在编写和维护更本不能卖的内部代码上（当然大多数程序员以此为生）—— 读者们也许乐意去查查报纸上的“诚聘信息”部分的编程工作列表检验一下。

我强烈的希望读者试试浏览本地报纸的招聘信息，看看编程。数据处理，和包含软件开发工作的软件工程项目等等。将这些工作按照其目的是使用还是销售进行分类，你将深受启发。

很明显，即使为"销售"定义了最大范围，20 人中还是至少有 19 个由使用价值资助（作为产品中间价值）。这就是为什么我们认为软件工业中以销售价值驱动的部分只占 5% 原因。注意，本文中其他部分的分析并非完全倚赖于这个数；即使这个数字达到 15% 甚至 20%，在经济上的推论结果仍然八九不离十。

（当我在技术讨论会上演讲时，我经常由讨论两个问题开始：听众为写软件付多少钱，和有多少薪水是依赖于软件的销售价值的。第一个问题应者甚众，而第二个问题则寥寥无几，大而且量的听众对这个问题十分诧异）

其次，经过对实际客户行为的调查，软件销售价值与其开发和升级成本相关的理论很容易被推翻。开发和升级成本相关的商品（对打折之前来说）占很大比例 —— 食品，汽车，机械工具，甚至有许多无形的产品 —— 例如，音乐、地图或数据库资料的复制权。这产品在生产者倒闭后仍然能保持甚至增加其销售价值。

与上述形成鲜明对比的是。当一个软件产品生产者歇业时（或者如果产品开发被终止），几乎没有客户愿意为其花钱，而不管它理论上的使用价值或同样功能产品的开发费用有多高。（要检验这个说法，去你附近的软件商店打折柜台看看吧:–)）

在生产者失败时，零售商的行为很有启示。他们知道一些生产者不知道的东东。他们深知：客户愿意花费的价格在很大程度上由卖主未来可以提供的服务决定。（这里的“服务”被广义的理解为完善，升级和后续产品）。

换句话说，软件主要是一个稳定的服务性行业，认为它是制造性行业是没有理由的错觉。

另外，检查一下我们为什么会有这些惯性思维也很有益处。它们也许来自于软件生产者大力宣传的销售类产品，这些是的软件业一小部分，也是宣传的唯一的一部分，大多数明显和重头的广告宣传的产品是昙花一现的短期产品，就像游戏，他们几乎不需要提供后续服务（合同规定的除外）

另外，值得注意的是，制造业错觉所倡导的价格体系事实上会越过保持开发预算不崩溃的底线。既然（像一般认为地）超过典型软件产品周期花费的 75% 在维护，调试和扩展上，那么通常的那种只采用高额售价，极低相关服务费用的定价策略，只会导致各方面都差的服务。

用户的损失在于，即使软件是服务性行业，工厂模式促使生产者减低服务质量。如果生产者靠出卖比特挣钱，大量的努力是制造比特并将它们推销出门；帮助服务部分，因为不是利润的中心，将会成为只付出的一点点努力和资源，为了避免激怒用户所设的垃圾站。

另一方面是大多数生产者使用这种工厂模式会导致长远的失败。为满足无限的售后服务和技术支持需要的固定价格产品提供资金，只有在那些膨胀足够迅速的市场里 —— 其过去的销售和未来的收入能够满足支持和生存周期的花费 —— 才能存活。一旦市场成熟和销量下降，维持生计，大多数生产者除了消减单独产品的开支之外没有别的选择。

不管是直接（废止产品）还是间接（支持很差），都会把客户推给竞争对手（因为这些行为损害了依附于服务产品的期望值）。短期来看，可以通过将修订过 bug 的版本发布为新产品避免这个陷阱。而长远来看，避免陷阱的唯一可能是对行业进行有效的市场垄断。最终，只有唯一的幸存。

事实上，我们一再的看到这种缺乏支持的模式害死一些市场环境中很强大的竞争者，（这种模式对那些那些经历过计算机发展史幸存下来的人尤其深刻，包括个人操作系统，字处理，通用财务程序或商业软件）。这种不正确的动机来自于工厂模式导致的赢家通吃的态势，而且最后即使你是赢家的客户也会遭殃。

如果不是工厂模式，那又是什么？为了有效的控制软件生存周期真实的花费体系（同时在经济学和非正式场合的意义上的“有效”），我们需要一个建立于服务合同，合约，和买卖双方持续交易基础上的价格体系。所以，在以效益为目的的自由市场条件下，我们能管窥大多数成熟的软件工业最终遵循的价格体系。

为什么说开放源码软件的地位日益增长，不仅仅是技术，也是经济上对主流秩序的挑战？上述内容给我们一些启示。软件开发的“free”，会将我们推向以服务支配的世界，同时暴露出一直依赖销售封闭源码产品的方式有多脆弱。

“free”的概念很容易被误解为其他含义。降低产品花费会导致支撑软件业的整个基础投入增长，而不是降低。只有汽车的价格降低时，汽车的需求才会上升 —— 这也是为什么在开放源码世界中，另外那 5% 的根据销售价值付酬的程序员不好受的原因。在 free 的变革中，有损失的不是程序员而是那些没看清形势而将赌注押在封闭源码策略上的投资者。

5.4 信息应该免费的神话

与工厂模式错觉相呼应的是，思考开放源码经济的人们还常常被另外一个神话搞糊涂。那就是“信息应该免费”。这常常以数字信息产品的复制边际成本几乎为零来解释，这个解释暗示了其价格似乎就应该为零。

其实你只要考虑一下诸如藏宝图，瑞士银行的账号口令，或计算机服务的确认口令，等等信息的价值，就很容易看破这个神话。即使这些确认信息可以不用任何花费的复制，但是被其确认的对象无法复制。也就是说，非零的边缘成本由被那些确认信息继承下来。

提到这个神话的主要目的是声明它与开放源码的经济价值的讨论无关；就象我们在后面将会看到的，即使假设软件是符合制造业产品（非零）价值结构，仍是如此。所以我们没必要钻软件是否应该免费的牛角尖。

5.5 驳斥公用悲剧说

质疑主流模式，看看我们是否能建立另一种模式 —— 对是支撑起开放源码协作的原 因作出有力的经济学解释。

这个问题需要从两个不同的方面考查。一个方面是我们要解释那些为开放源码作出 贡献的人士的个体行为；另一方面，我们需要理解那种支撑象 Linux 和 Apache 这样的 开放源码项目的经济力量。

Hardin 的著名寓言告诉我们：设想一个乡村农夫们拥有一片公用绿地。他们在那里放牧牲畜。但是放牧使公用性退化，撕裂草皮，留下泥泞，很难恢复。如果没有对分配放牧的权利达成协议（或约定）以防止过度放牧；所有牧主都还会赞成尽可能快的增加牲畜数量，以便在公共绿地变成泥潭之前榨取最大的利润。

大多数人使用象这样的直觉的合作模式。这事实上并不是对开放源码 —— 他们是（供不应求的）自由骑士，而不是（被过度使用的）过剩的公共货物 —— 经济问题的正确判断，不过，我在大多数未充分考虑的反对声后面都听到过类似的看法。

公共拥有的悲剧预言只会出现三种结果。一种是泥潭；一种是为了村民的利益，强制性的使用某种分配协定（共产主义的解决方案）；第三种是公用被打破，村民各筑藩篱，保护自己的一小块草地（私有制的解决方案）。

当人们本能的的将这种模式应用于开放源码合作时，因此预计它只有很不稳定的短暂的半衰期。因为没有明显的方法去强制在互联网上工作的程序员执行工作时间分配策略，这种模式就断言公用将会打破，结果是出现各种各样的封闭代码软件和反馈给公用的工作量迅速减少。

事实上，经验清楚的显示出了与之相反的趋势。开放源码开发的广度和深度（由 Matalab 和 freshmeat.net 的每日宣布的数据统计）在稳定增加。很明显，这些都得出“公用悲剧”模式无法描述事态的发展。

答案的一部分正是建立在软件使用并不降低其价值的事实基础之上。实际上，对开放源码软件来说，当用户被其修正和特性（代码补丁）把握之后，软件的广泛使用还会增加其价值。公用悲剧被颠覆了，越放牧，草长得越高。

答案的另一部分是基于很难收取那些为公用源码基础所作的小补丁的市场价值。假设我为一个恼人的 bug 写了个修正，而且有人认为这个修正值钱；我如何才能从那些人手里拿到钱？对于这种小额的，通常也是适当的付款，常规的付费体系如此昂贵竟成为真正的问题。

比起价钱不仅仅很难收取，也许如何定价还要难得多。让我们想一想，假设互联网上已经拥有理论上完美的小额付费系统 —— 即安全，方便，又不需要更多手续费。而你写了个补丁叫做“Linux 内核的某某修正”。你该要价多少？在潜在购买者还没看到补丁时，他们又该如何判断值不值得为它付费呢？

我们的问题就像 F.A.Hayek 的“计算问题”在哈哈镜中的变形 —— 它就像个超市，既要估价补丁的功能值多少，又要相信定价是合理的以促进交易。

不幸的是，超市方式有一系列的不足，所以补丁的作者 —— 打补丁的黑客有两种选择：躺在补丁上收钱，或免费扔出去。第一种选择将一无所获。第二种也可能如此，不过或者它会促使其他人提供互惠的给予，以解决上面那位黑客所头疼的问题。第二种明显无私的选择，在这种游戏情况中，竟然事实上是自私的。

在分析这种合作时，自由软件的开发所面临的问题会很重要（他们可能会工作在清贫，或没有足够的回报的情况下），这并不是由最终用户的数量决定的。开放源码项目的复杂性和沟通所带来的成本几乎完全和参与的开发者的数量成函数关系；拥有更多的几乎从不看源码的最终用户对此似乎没有任何益处。这只会增加在项目邮件列表中无聊问题出现频率，但是建立一个相关的常用问题列表，不理睬那些显然不读 FAQ 的人（事实上这已经是通用做法），可以很容易解决这个问题。

开放源码软件的真正最重要的自由软件开发问题是提交补丁功能时的磨合成本。可能的贡献者在声望上小有收获（见《开拓智域》一文），而没有金钱上的补偿，想着“根本不值得提交这个修订，因为我不得不打补

丁，写修改记录，在 FSF 任务文件上署名……”。因为这个原因，拥有大量贡献者（其次才是成功）的项目很强壮。与之相反的是，每个有许多相互有制约关系的项目都需要有从始到终的贡献者。这种磨合成本就像政治一样呆板。总之，自由软件项目本身可以向你解释为何松散，无组织的 Linux 文化，比紧密组织且集中管理的 BSD 项目的努力，更能吸引合作能量的意向；以及为何自由软件基金会，也在 Linux 崛起时重要性相对的减弱。

这条路不管走多远都是好的。但是，这只是在黑客写了补丁并公布了这个补丁后的事后诸葛亮式解释。我们需要的另一半答案是对为何 JRH 最初会写这个补丁，而不是为拥有销售回报的封闭源码程序工作。作出经济解释。到底什么商业模式创造了开放源码开发繁荣发展的环境呢？

5.6 封闭源码的原因

在给开放源码经营模式分类之前，我们应该先大致地考虑一下封闭的代价。当我们封闭源码时，我们究竟在保护什么？

比方说你雇了某人来编写和组织一个（不妨说）为你的生意专用的结算软件，那么和开放源码比起来，封闭源码一点也不会有助于解决问题。如果你想封闭源码，唯一合理的理由就是你想把这个软件卖给别人，或者不让你的竞争者使用它。

比较明显的原因是你在保护销售价值，但是对 95% 的供内部使用的软件来说这没意义。那么封闭还有别的什么好处吗？

第二个原因（保持竞争优势）还有待检验。假如说你把那个结算软件开放源码了，它流行起来，并且从社会上得到了改进。现在，你的竞争者也开始使用它了，他没有花开发费用就得到了好处，而且影响了你的生意。这是不是一种反对开放源码的理由呢？

可能是 —— 也可能不是。真正的问题在于你从分散开发负担中得到的好处是否多于由那些不劳而获的人带来的竞争损失。许多人倾向于为这类交易作苍白的辩解，方法是：（a）避而不谈从额外的开发帮助中得到的功能上的改进。（b）不认为开发费用是降低了，而是假定你无论如何也是要承担这些开发费用的，所以把它们作为开放源码（如果你这么选择的话）的代价是错误的。

还有别的许多封闭源码的根本就是荒谬的理由。举例说，你可能误以为封闭源码可以使你的商用系统更加安全，不容易被破解或闯入。如果是这样，我建议你立刻找一个密码专家来诊断一下你的系统。真正的猜疑心很重的人都知道不能相信封闭源码程序的安全性，因为这是他们是从惨痛的教训中学到的。安全性是可靠性的一个方面； 只有那些被彻底检查过的算法和代码实现才可能被相信是安全的。

5.7 使用价值集资模型

使用价值与销售价值之间的差别让我们注意到的一个基本事实是只有销售价值本身受到了来自从封闭源码到开放源码这个转变的威胁；使用价值并没有。

如果使用价值，而不是交换价值，的确是软件发展的根本驱动力；而且开放源代码的发展的确是比源代码封闭要更加有影响力和更加有效率，那么我们应该期待着去寻找一种环境，在这种环境中光是使用价值已能够完全地促使开放源代码向前发展。

实际上，这样的几个环境模型并不难以找到。在这样的模型中，开放源代码的全职开发者的生存完全可以由（开放源代码的）使用价值来实现。

5.7.1 Aapache 的个案：（价值分享）

假如你在为一个拥有高效性高可靠性网络服务器的商业公司服务。也许这个服务器是用来为电子商务服务的，也许是作为一个出售广告的高可视性的媒体输出装置，也许只是用来构建一个门户站点。你需要一天 7 小时的在线时间，你需要速度，还有规范性。

那么你该如何做呢？这里有些基本的策略可以供你参考：

购买一个私有的网络服务器，这样，你是在冒险相信卖方的宣传与你的需求是一致的，你在冒险相信卖方的技术竞争力能给提供完善的保障。即使假设这两个方面是有保障的，网络服务器本身也会由于缺乏规范的服务而出现问题。你只能通过卖方的经过挑选提供的几种工具来维护的你的服务器。这种购买私有的服务器的路子并非一个很大众化的方法！

自己做一个！做一个自己的网络服务器在目前还是不可忽略的一种调剂办法；网络服务器并不太复杂，当然比浏览器要简单。一个专门用途的网络服务器可以做得功能专一但很好用。走这条路的话，你能得到你所需要的各种特性和自己的规范，尽管在其升级的过程中你要付出很多。或许你的公司在你离开或退休后，还会发现这个服务器有了这样或那样的问题。

参加 Apache 小组！Apache 服务器是有一个通过 Internet 交流的小组写出来的 —— 小组成员都是系统管理员，他们相信比较明智的做法是将他们的能力集合起来去写，并提高一个单一方向的代码集而不是去花费时间

各自同时写完全不相关的代码。这样做的结果是他们能够同时发挥“自己做一个”和大范围大规模测试代码的优势。

选择 Apache 小组的优势很明显。到底有多明显，可以根据 Netcraft 的每周回顾来判断一下。Netcraft 上说 Apache 服务器从其诞生起一直在稳定地夺取其他私有服务器的市场份额。1999 年 6 月，Apache 的各种版本占有了 61% 的市场份额 <http://www.netcraft.com/survey> —— 没有合法的拥有者，没有组织机构，也根本没有合同制约的组织形式在背后操纵。

总的说起来，Apache 的故事提供了一个模式：软件使用者通过支持开放源代码计划而发现了这个模式，他们发现这样做能以最小的代价给他们带来越来越好的软件，比其他任何方法都要有效。

5.7.2 Cisco 的备案：风险均摊

一些年以前，两个 Cisco（网络产品制造厂家）的程序员被分配来写一个分布式的打印系统的程式代码用做 Cisco 的合作网络的应用。这个项目的挑战性很大。这个系统要使任意一个用户能在这个网络上的任意一台打印机上打印东西（而用户和打印机可能只是隔壁或者相隔几千公里），当打印机没有纸了或其他紧急情况系统要能够将任务导向另一台附近的打印机。系统还要能够将这一个突发时间报告给打印机管理员。

他们两个对 Unix 上的打印软件做了一些很不错的修改，加上一些包的原语言，就做成了那项工作，但接着问题就来了。

问题是两个程序员都不愿意在 Cisco 永远呆下去。结果两名程序员都将离开，而软件也会无人维护而“腐烂”（就是无法满足实际应用中不断变化的要求而失去其应用）。没有任何一个人愿意看到这样的情况在他自己或工作上发生，那两个程序员也认为他们已经做了 Cisco 公司要求他们做的事情，其他的问题已经不是他们的工作范围了。

于是他们跑到他们的经理那里要求将这个打印软件的源代码开放。他们认为这样的话 Cisco 不仅不会失去什么反而会得到更多。通过协作鼓励用户和软件开发合作者的组织的发展，Cisco 能够弥补因为软件原创人员的离开所带来的损失。

Cisco 的故事引出另一个模式：源代码开放使开发一个软件的风险被众多协作者分摊了而且投资分花费很小。所有的团体都发现源代码的开放，以及一个成员各自独立却互相协作的社区的存在将提供一个无风险的开发环境，而且这个环境是有商业价值的 —— 它能够自己赚钱养活自己！

5.8 为何销售价值存在问题

开放源码使得直接获取软件销售利润非常困难。困难并不是来自技术方面的，因为源代码和可执行代码一样易于拷贝，并且版权法和许可证法的约束不同使得通过开放源码软件来获取销售利润比封闭源码软件难。

真正的困难来自维护开放源码发展的许可证本身。因为三个相互推动的原因，大多数的开放源码许可证禁止对用户使用、分发、修改软件的权利进行限制，以此避免有人利用开放源码软件牟取直接利润。为了更好的理解这些原因，我们有必要对这些许可证所涉及的社会背景 —— 黑客文化（可以访问下面网址：http://www.catb.org/~esr/faqs/hacker-howto.html 做一番探讨。

原因与对市场的敌视无关，虽然这样的误解在黑客圈外至今广为流传。不排除有小部分的黑客确实一直对商业动机抱有敌意，但大部分的黑客还是愿意与一些以盈利为目的 Linux 集成商（如 REDHAT、SUSE、Caldera）合作的。这也表明只要符合他们的意愿，大多数的黑客会乐意和商家合作的。如此看来，黑客们敌视以获取直接利润为目的的许可证的真正原因非常微妙也非常有趣。

原因之一，对等原则。大多数开放源码的开发者允许别人利用他们的成果来获取利益，还有许多开放源码的开发者同时还规定不允许某一方（有时源码的开发者除外）出于特权地位来牟取利润。只要黑客们自己潜意识里打算从他们开发的软件或补丁中赢利，他们一般也愿意别人来与他合作，共同赢利。

原因之二，意想不到的后果。黑客们发现在许可证中对软件的商业应用与销售进行限制和收费（为获得销售利润而通常采用的做法）会使得人际关系变得淡漠。其中一个特例就是所谓的“盗版光盘”，这本来应该鼓励的，但现在却被认为是违法和不道德的。总的来说，对用户使用、销售、修改、分发软件的权力（以及版权协议中其他复杂权利）进行限制会导致人们循规蹈矩，时时刻刻担心自己会犯法（这种担心会随着人们使用的软件包的增加而愈演愈烈）。这无疑是非常不妙的，因此简化许可证，解除许可证中的各项限制已成为大势所趋。

原因之三，也是最关键的一个原因，就是代码共享。这种赠与文化在《开拓智域》一文中有生动的描述。某些许可证体系中用来保护知识产权或者限制直接获取销售利润的各项规定使得人们不能合法的实现代码共享，（如 Sun 公司的 JiniJava “社区资源”许可证）。然而代码共享却被认为是最后一根救命“稻草”（《开拓智域》一文中大段大段的解释了这个问题），当软件维护者无力承担或者放弃对代码的维护时（比方说是一个非常封闭的许可证），代码共享就非常关键了。

黑客群体对于对等原则还是有所妥协的，所以他们能够容忍一些象 Netscape 的 NPL（NPL 明确规定不允许非公开源码的产品使用开放源码的 Mozilla 代码）一样给予源码创作者一些特权的许可证。对于第二条原因，妥

协的就少一些。而对第三条原因极少会作出让步（这也是 Sun 公司的 JAVAand Jini Community License 计划遭到黑客们广泛反对的原因）。

上述原因解释了开放源码定义中的各项条款。这些条款从一些典型的自由软件版权协议（如 GPL 协议，BSD 协议，MIT 协议以及 Artistic 协议）的细微特征中表达了黑客群体的思想，它们（虽然不是有意的，但客观上）使得获取直接利润极为困难。

5.9 间接销售价值模式

然而，还是有办法来开拓与软件服务相关的市场，从而获得间接销售价值。有五种已知的和两种正在探索的模式（未来可能会发展出更多的新发展模式）。

5.9.1 失败的领导者/市场定位者

在这种模式中，利用开放源代码软件为直接产生收入的专有软件来创造或维持一种市场位置。在大多数普遍的情形中，开放源代码的客户端软件带动了服务器软件的销售，或者可增加了门户网站的访问量/广告收入。

网景公司（Netscape）在 1998 年开放了 Mozilla 浏览器的源代码时，就是使用了这种策略。他们浏览器端的商业收入只占总收入的 13%，而且在 Microsoft 开始发布 Internet Explorer 后市场份额还在下降。IE 强大的市场营销（及其捆绑策略后来成为反托拉斯案的核心问题）迅速的吞噬了 Netscape 浏览器的市场份额，造成了 Microsoft 试图垄断浏览器市场，并利用微软强加给用户的 HTML 的“标准”，形成逐步把 Netscape 赶出服务器市场的态势。

通过开放仍然流行的 Netscape 浏览器的源代码，Netscape 有效的阻止了 Microsoft 垄断浏览器的可能性。他们期望开放源代码协作会加速浏览器的开发和测试，并希望能降低 Microsoft 的 IE 的发展速度，阻止它独自定义 HTML 标准。

这个策略生效了。在 1998 年 11 月，Netscape 实际上开始从 IE 那里夺回市场份额。在 1999 年初 Netscape 被 AOL 收购时，保持 Mozilla 所取得的竞争优势是很明显的，这一点可以从 AOL 的行动中显而易见，AOL 首先对外的承诺的就是继续支持 Mozilla 计划，虽然她还处在 alpha 测试阶段。

5.9.2 糖霜策略

这种模式是针对硬件制造商的（这里的硬件包括从以太网或其他外部设备直到计算机系统的所有东西）。市场压力迫使硬件公司书写和维护软件（从设备驱动程序、配置工具直到整个操作系统的级别），但是软件本身并不是利润中心。它是一项开支 —— 通常是一项重要开支。

在这种情况下，开放源代码是一种很好的策略。由于没有赢利上的损失，所以没有负面影响。销售商获得的是奇迹般膨胀的开发人员队伍，对用户需求获得更加快速、灵活的反应能力，并且通过同行检查而获得的更好的可靠性。而且可以免费得到了其他系统的移植。这种做法还可在很大程度上提高客户对公司的信任度，因为客户的技术人员可以花费了更多的时间根据自己的需要定制代码。

有一些经常被销售商提出的反对开放硬件驱动程序源代码的理由。为了不把它们和这里的更加一般的问题搅在一起，我在附录里专门讨论了这个问题。

开放源代码的“将来获益”的效果在糖霜策略中体现的尤其强烈。硬件产品有一个有限的制造和支持的生命周期，在那以后，用户就自己照顾自己了。但如果他们可以获得驱动程序的源代码，并可根据需要加以修改的话，他们就更可能高高兴兴的成为同一公司的回头客。

糖霜模式的一个非常戏剧性的例子是苹果公司在 1999 年三月中旬决定开放它们的 MacOSX 服务器的操作系统“Darwin”的代码。

5.9.3 奉送食谱，开办饭店

在这种模式中，开放源代码软件建立了一种市场定位，并不是为了象在失败的领导者/市场定位者模式中一样针对封闭源代码软件，而是针对服务。

（我曾经把这种模式称为“奉送剃刀，销售刀片”，但是软件和服务二者的关联并不如剃刀/刀片所类比的那么紧密。）

这是红帽和其他 Linux 发行商所采用的模式。他们卖的其实并不是软件代码本身，而是通过组合和测试一个能转的操作系统产生的附加价值，这个操作系统被担保有销路并与同一品牌的操作系统兼容。构成他们的价值的其他元素包括免费安装和提供可选的持续技术支持合同。

开放源代码的创造市场的能力极为强大，尤其是对那些天生就作服务的公司来说更是如此。进来一个非常有教育意义的例子是 DigitalCreations 公司，它是一个创建于 1998 年的 web 站点设计机构，专长于复杂的数据库和事务站点的开发。他们的主要工具，公司的知识财产——皇冠上的明珠，是一个对象发布系统，它曾经有过几个名字，现在被称为 Zope。

当 DigitalCreations 的人寻找风险投资时，风险投资商仔细的估计了他们的预期市场份额，他们的人力资源和那套工具后，就建议 DigitalCreations 开放 Zope 的源代码。

从传统的软件工业标准来看，这看起来绝对是一个疯狂的举动。常规的商业学校认为象 Zope 这种核心知识财富是一个公司的掌上明珠，是在任何情况下也不能放弃的。但是那位风险投资商从两个相互关联的角度来考虑问题，一个是 Zope 的真实核心资产实际上是它的人员的大脑和技术；第二个是 Zope 作为一个创造新市场的标准而不仅仅是一个秘密武器会产生更多的价值。

为了看清这一点，请比较两种情况。在通常情况下，Zope 保留为 DigitalCreations 的秘密武器。让我们假定它是一个很有效的武器。结果，公司可以在很短的时间内交付高质量的软件 —— 但是没人知道这个秘密武器。满足客户是容易的，但是建造一个客户群体是困难的。

然而那个风险投资商看到了对 Zope 系统开放源码可以为 DigitalCreations 的真正财富 —— 它的技术员工产生巨大的广告效应。他期望使用 Zope 的客户会认为雇用象 DigitalCreations 这样的专家会比自己开发自己的 Zope 技术更加高效。

Zope 的一个负责人曾经非常公开的确认了他们的开放源代码策略“开启了许多其它方式无法开启的门”。潜在的客户确实反应了这种情况 —— 所以 DigitalCreations 公司迅速发展起来。

另一个很近的例子是 e-smith 公司 <http://www.e-smith.net>。这个公司出售定制的开放源代码的 Linux 的 Internet 安全服务器。他们的一个负责人描述了 e-smith 迅速扩展的免费下载服务，他说“大多数公司都要考虑软件盗版问题，而我们把它看作一个自由市场。” <http://www.globetechnology.com/gam/News/19990625/BAND.html>

5.9.4 附加产品

在这种模式中，我们出售开放源代码的附加产品。在低端市场，出售杯子和 T 恤衫；在高端市场上，出售专门编辑并出版的文档和书籍。

O'Reilly 集团是一个附加产品公司的很好的例子，他出版了很多优秀的开放源代码软件的参考资料。O'Reilly 实际上雇用和支持了一些著名的开放源代码黑客（例如 Larry Wall 和 Brain Behlendorf），并以次提高它在市场上的声望。

5.9.5 未来免费，出售现在

在这种模式下，我们以封闭的许可证发布软件的可执行文件和源代码，但是包含一个有关封闭条款的期限。比如，我们可以写一个许可证，允许免费的散发软件，禁止不付报酬的商业应用，并保证发布一年以后或开发商终止开发后软件将在 GPL 保护之下。

在这种模式下，客户可以保证产品能够根据他们的需要定制，因为他们可以得到源代码。产品的将来也是得到保证的——许可证保证了如果始创公司失败后，开放源代码社区仍能够接管该产品。

因为销售价格和数量是依赖于客户对产品的期望值，始创公司可以享受到比以封闭源代码许可证发行的软件更优厚的收入。而且，因为老的代码是在 GPL 保护下的，所以它可以得到同行认真的检查、排错和添加其他小功能，这样可以为原创者减轻 75% 的维护负担。

这种模式被 Aladdin 公司成功的采用了，它创造了流行的 Ghostscript 程序（一个 PostScript 解释器，它可以把 PostScript 翻译成许多打印机的内部语言）。

这种模式的主要缺点是那些封闭的条款倾向于抑制产品开发早期的同行检查和参与，而那时是最需要的大家的参与的时候。

5.9.6 软件免费，销售品牌

这还是一个试探性的商业模式。我们开放一项软件技术，保留测试包或一套兼容性标准，然后卖给用户一个品牌认证，保证他们对这种技术的实现和其他具有这种品牌的产品相互兼容。

（这是 Sun 公司应该对待 Java 和 Jini 的方式。）

5.9.7 软件免费，销售内容

这时另一种试探性的商业模式。想象一些象股票信息订阅的服务。价值既不在客户端软件也不再服务器商，而在于提供客观的的可靠的信息。因此我们开放所有的软件，出售内容订阅。当黑客们把客户端移植到新的平台上或者以不同方式扩展它时，我们的市场自动扩展了。

（这是为什么 AOL 应该开放它的客户端软件。）

5.10 何时开放，何时封闭

在考察了支持开放源代码软件开发的几种商业模型之后，我们可以来讨论一下何时开放源代码、何时封闭源代码才有经济意义这样的一般性问题了。首先，我们必须弄清楚每种策略如何盈利。

5.10.1 靠什么盈利?

封闭源代码的方式让你可以从秘密的比特中收取利润；另一方面，它阻止了其他同行对代码进行检验的可能性。开放源代码方式为其他同行检验创造了条件，而且你也不能从秘密的比特中获得利润。

从秘密的比特中盈利很好理解；传统的软件商业模型就是围绕着它建立的。但是直到近来，其他同行检验代码的价值还未被很好的理解。然而，Linux 操作系统使得我们对问题的认识更加清晰，这些认识我们本应在几年前从 Internet 核心软件和其他软件工程分支的发展历史中就应该学到——开放源代码的同行检验是得到高可靠性和高质量的软件的唯一可伸缩的方法。

因此，在一个竞争的市场上，寻找高可靠性和高质量软件的客户会给那些开放源代码软件开发人员以回报，是他们探索出怎样在服务、附加值和与软件相关的辅助市场中维持一个稳定的收支循环。这种现象正是 Linux 令人惊讶的成功背后的原因，Linux 在 1996 年的一片空白发展到 1998 年末的商业服务器市场的 17%，而且似乎会在两年之内占领这个市场（1999 年初，IDC 预测 Linux 将在 2003 年成长的比所有其它操作系统的总和还要快）。

开放源代码的一个几乎同样重要的作用是作为一种传播开放标准，围绕它建立市场的手段作用。Internet 的戏剧性增长得益于没人拥有 TCP/IP；没人有权封锁 Internet 的核心协议。

TCP/IP 和 Linux 成功的所造就的互连网络对世界的影响是显而易见的，开放的系统最终减少了信任和平等的问题 —— 如果大家都能够看到底层结构是怎样工作的话，他们就会理所当然的更加信任它；人们更加喜欢一个所有人都是平等的底层结构，而不是一个某一方具有获利的特权并可以施加控制的底层结构。

然而，其实为了向软件用户说明平等的重要性时，我们不必非要强调网络的影响力。没有哪个软件用户在质量和功能类似的开放源代码软件存在的条件下放弃开放源码软件，而去选择封闭源代码软件，非要让自己被某个供应商垄断控制才高兴。软件对消费者的事务越重要，这个问题就越突出——它越重要，消费者就越不能容忍自己被另外一方控制。

最后，和信任问题相关的开放源代码的重要优势就是它的光明前景。如果源代码是开放的，即使发行者垮掉了，客户还是能掌握一些资源。这对于糖霜策略尤其重要，因为硬件趋向于较短的生命周期，但是作用更加普遍，并转换成开放源代码的增长价值。

5.10.2 它们怎样相互作用?

当从秘密比特得到的回报比从开放源码高的时候，从经济意义上说应该封闭源代码。当从开放源代码得到的收益比从秘密比特高的时候，那么无疑开放源代码更有意义。

从表面上看，这是一个很普通的想法。但是当我们注意到开放源代码的回报比秘密比特更加难以度量和预计时，就是说回报常常被低估而不是被高估，这一点就不那么平淡无奇了。实际上，直到 1998 年初业界主流开始重新考虑遵从 Mozilla 发行源代码的前提时，开放源代码的回报一直被普遍错误的认为是零。

那么我们怎样评价开放源代码的回报呢? 一般的说这是一个困难的问题，但是我们可以象处理其他任何一个预言性问题一样来处理它。我们可以从观察开放源代码成功和失败的案例开始。试着抽象出一个模型，至少给出一个定性的感觉，在什么情况下开放源代码对投资者或追求最大回报的商业操能产生净收益。然后我们再用数据来细化这个模型。

从《大教堂和市集》一文的分析中，我们可以得到开放源代码在（a）可靠性/稳定性/可扩展性至关重要时，和（b）设计和实现的正确性除了采用其他同行检验的办法外难以验证时具有高的投资回报。（在实践中多数重要程序都符合第二个标准。

当软件对一个消费者至关重要时，消费者为避免被一个垄断的供应商所控制的愿望提升了他对开放源代码的兴趣（也因此提升了开放源代码厂商的市场竞争力）。因此，另一个标准（c）当软件是一项非常重要的资产时（例如，很多企业中的 MIS 部门），封闭源码会把用户推向开放源代码一方。

在应用程序领域，我们看到开放源代码底层软件创造了信任和平等的结果，随着时间的推移，一定会吸引到更多的客户，从而胜过封闭源代码底层软件；在这个迅速扩张的市场上占有较小的份额常常比在封闭的和迟缓的市场上占有较大份额还要好。因此，对于基础结构软件，开放源代码的方式比利用知识产权得到收益的封闭源代码方式会得到更高的长期回报。

实际上，潜在用户根据发行商的策略推知它的将来发展能力，同时他们有不愿接受一个垄断供货商的本能，因为这将意味着要处处受到约束；除非已经有了一个压倒性的市场力量，否则你可以选择一个开放源代码的方式也可以选择一个从封闭代码直接受益的方式 —— 但是不可能同时选择二者。（在别的地方可以看到类似的情况，举例来说，在电子市场上用户常常拒绝购买单独货源的设计。）这种情况的消极性可以消除一些：在网络占支配地位的地方，开放源代码似乎是正确的选择。

我们可以总结一下这种逻辑：在（d）创建一个公共计算和通讯的底层结构时，开放源代码软件似乎可以比封闭源代码软件成功的获得更大的回报。

最后，我们注意到，相对于核心算法和基础知识已被很好理解的服务提供商，提供唯一或独特服务的商家更加担心竞争对手会模仿他们的方法。因此，在（e）核心方法（或功能）是公有知识一部分时，开放源代码更加可能取胜。

实现了 Internet 核心软件，Apache，和 ANSI 标准的 UnixAPI 的 Linux 系统是上面分析的五个标准的典型样板。在十五年建造自己的封闭协议（如 DECNET，XNS，IPX 等等）帝国的尝试失败之后，90 年代中期数据网络重又向 TCP/IP 集中，这生动的印证了这种市场向开放源代码演化的道路。

另一方面，开放源代码对拥有自己独特的创造价值的软件资产的公司没有太多意义（强烈满足条件（e）），下面这些情况也不太适用与开放源码，比如软件（a）对失效相对不敏感，（b）可以用同行检验以外的方式来验证，不是（c）关键事务的，并且不是主要从（d）网络作用或普遍使用上获得价值的。

作为一个极端的例子，1999 年初由一家公司问我“我们是否应该开放源代码？”，这家公司为锯木机编写计算切割模式的软件，可以从原木中获得最大的板材。我的结论是“不”。他们唯一接近满足的条件是（c）；但是在紧要关头一个熟练的操作员可以手工的决定切割模式。

值得指出的是，满足这些条件的特定产品或技术会随时间发生变化，从下文的案例中我们会看到这一点。

总而言之，下面的条件宜于采用开放源代码模式：

(a) 可靠性/稳定性/可扩充性非常关键时

(b) 设计和实现的正确性不能很容易的用其他同行检验以外的方法验证时

(c) 软件对用户控制他/她的事务非常关键时

(d) 软件用来创建一个公共计算和通讯基础结构时

(e) 关键方法（或等价功能）是公共工程知识的一部分时

5.10.3 Doom：一个案例

id 软件公司卖得最火的游戏 Doom 的历史展示了市场压力和产品演化怎样改变了封闭源代码软件相对于开放源代码的收益数量。

当 Doom 在 1993 年末第一次发布时，它的主观视角，实时动画是极为独特的（条件（e）的对立面）。不仅因为它那令人叫绝的视觉效果，而且在很长一段时间内没人知道他们是怎样在低级的处理器上实现这些效果。这些秘密的比特可以获得非常重要的收益。而且，开放源代码的潜在收益很低。作为一个单独的游戏，这个软件（a）它的故障的代价很小，（b）不是非常难于验证，（c）对任何一个用户来说都不是至关重要的，（d）并不得益于网络。所以 Doom 成为封闭源代码在经济上是很合理的。

然而，Doom 周围的市场不是静止的。竞争对手发明了它的动画技术的等价功能，其他的“主观射击”游戏比如毁灭公爵（DukeNukem）等开始出现。当这些游戏侵蚀Doom 的市场份额时，秘密比特的收益开始下降。

另一方面，扩展市场份额的努力带来了新的技术挑战——更好的可靠性，更多的游戏特色，更大的用户群，和跨平台。随着"deathmatch"的多人游戏模式和Doom游戏服务的出现，市场开始显示出对网络的依赖。所有这些需求都要求id公司在下面版本的游戏中花费更多的精力。

所有这些趋势都提升了开放源代码的回报。在某一点回报曲线交叉，开放源代码成为 id 公司在经济上合理的选择，他们可以从诸如游戏扩展选集等第二市场上获益。在这一点之后的某个时间，事情确实发生了。1997 年末 Doom 的完整源代码被公开发行。

5.10.4 知晓何时放手

Doom 是一个有趣的案例，因为它既不是一个操作系统也不是一个通讯/网络软件；因此这远离了开放源代码的通常的明显的例子。确实，Doom 的生命周期，包括交叉点，可以作为今天的代码生态中应用软件的典型 —— 在这个生态环境中，通讯和分布计算软件要求较高健壮性/可靠性/可扩充性、只能通过同行检验来验证，并且常常超越技术环境和竞争者之间的界限（包含信任和平等）。

Doom 从一个单机游戏演化到 deathmatch 模式。网络计算越来越重要。同样的趋势可以从最重要的商业应用程序，如 ERP 系统看到。商务网络把供应商和客户更加紧密的联系在一起 —— 当然，它们包含在整个万维网的体系结构之中。这种情况到处可见，开放源代码的回报稳步增加。

如果当前的趋势继续下去的话，下个世纪软件技术和产品管理的核心挑战将是知晓应该何时放手 —— 何时把封闭源代码转变为开放源代码体系结构，从而得到同行检验的好处，并从服务和其他第二市场上得到更高的回报。

大家很明显都不想在任何一个方向上离交叉点太远。除了这个，等待太长时间面临着严重的风险 —— 你可能会被一个走向开放源代码的同一市场上的竞争对手铲平。

这个问题之所以严重的原因是，可以被吸引到某类产品的开放源代码合作者的用户群和专家群是有限的，而且这些人很难于转移。如果两个功能基本相同的竞争代码一先一后开放源代码，那么先开放的更加可能吸引更多数的用户和更多数的最激情的合作开发人员；后开放的则不得不吃剩饭。吸引来的人员难以转移，因为用户对软件已经熟悉，而开发人员已经在代码上投资了很多的时间。

5.11 开放源代码的商业运作

在开放式源代码的社区中，通常是以一种倾向于增强开放式源代码生产效益的方式来组织其自身的商业活动的。尤其在 LINUX 的世界里，存在着一个具有重要经济意义的事实，那就是存在有许多相互竞争的发行商，而他们形成了一个与开发团体相分离的、独立的层次。

开发人员写源代码，并且使得这些源代码在互连网上是可以被下载的。每个发行商都从这些可下载的源代码中选取一些，并将它们进行综合，包装，并且注册商标，最后将其买给顾客。用户可以选择发行商的产品，也可以通过直接从开发商的网站下载源代码而增补其自己已安装的发行版。

这一分化出来由发行商形成的层的作用是为创造了一个非常易于改变、可对产品不断完善的内在市场。开发人员为了吸引更多的发行商和顾客的注意力，在他们软件的质量上彼此竞争。而发行商则为了从用户那里赚得更多的钱，互相在他们选择源代码的策略以及他们给软件带来的附加价值上竞争。

内在市场结构中的第一特征就是网络中没有什么源代码是不可缺少的。开发商可能倒闭，即使他们的那部分底层代码没有直接被其他开发者所用，为吸引更多注意力而导致的竞争将倾向于尽快产生一个在功能上可替代的产品。发行商可能在没有破坏或修改开放源代码的情况下就破产了。整个开放式源代码的商业系统作为一个整体，与任何一个独立的封闭源代码的操作系统的发行商相比较而言，对市场需求有着更快的反应，并且在抑制巨大的波动及自我创新方面有着更强的能力。

开放源码另一个重要的特征就是通过分工降低成本，提高了效率。开发商不愿经受传统的封闭源代码项目中那种例行公事般的压力，而是象这样来工作：没有来自市场方面的那些不得要领、分散注意力的表单；没有要求他们使用不适合的而且已过时的语言或开发环境的强制命令；没有打着突出产品的特性和保护知识产权的幌子要求用一种新的，不兼容的方式重新设计“轮胎”的命令；而且最重要的是没有项目完成最后期限的约束。这样，公司就不会在产品还没有做好以前，就匆匆忙忙地推出一个 1.0 版本，正如 DEMARCO 和 LISTER 在从对"做完了再喊我"管理模式的讨论中所作出的评论（见《开发队伍与产品》一文）那样，这种模式通常不仅会有益于质量的提高，而且实际上有助于一项真正的研究成果以最快的速度进行传播。

另一方面，发行商们可以专门从事他们能高效完成的事情。这样，他们就可以集中精力在系统的综和一体化，包装，质量保证及服务方面，而不用去考虑所需要的大量的资金问题以及使正在进行的软件开发保持其竞争力的问题。

通过作为开放式源代码商业模式中不可缺少的一部分，即来自于用户的不断的信息反馈和监督，无论是发行商还是开发商都会比较诚实一些。

5.12 成功的复制

公用的悲剧也许并不在于他们对现如今存在的开放式源代码商业模式发展的适应性，但这并不意味着不存在任何理由去怀疑开放式源代码社区内目前的状况是否能持续下去。主要的参与者是否会随着风险的进一步增大而背叛共同的合作?

这一问题可以从几种不同的层次来提出。我们的那个与"成功的公用"相反的故事是基于这样一种论断的，那就是个人对开放式源代码的贡献价值很难以量化的方式来衡量。但是这一论断对于像 LINUX 的发行商那些已经拥有一部分与开放式源代码相连系的收入的公司来说，就没有太大的影响力了。而且，他们每天的贡献价值已经量化了。但是，现在这种合作角色稳固吗?

对这一问题的研究将导致我们对一些问题有趣的思考，譬如现如今真实世界中开放式源代码软件的经济状况，以及什么才是未来软件业中真正的软件服务行业中的典范。

从实际的角度来讲，适用于现存的开放式源代码社区的这一问题通常可以用两种不同的方式来提出。一种 LINUX 将分裂吗? 另一种是与第一个相反的，LINUX 将发展成为一个处于支配地位，类似于垄断性的产品?

当暗示 LINUX 将分裂时，我们不能不联想到 20 世纪 80 年代 UNIX 版本分裂的历史，许多人又重新开始思考历史是否回重演。尽管无休止的有关开放标准的讨论，尽管有许许多多的联盟，协作和合同，UNIX 的所有权归属还是分裂了。事实证明卖方通过增加或改变操作系统设备从而使他们的产品与众不同的愿望比他们通过维持其兼容性，不断的减少独立软件开发商的进入障碍，以及降低维持与顾客的固定业务关系的总成本，来增大UNIX的整个市场份额的兴趣要更强烈。

但是上述情况不大可能发生在 LINUX 身上，这是基于一个很简单的原因，那就是 LINUX 的所有开发商都被限制基于开放源码这样的根基来进行开发和其他运作。而且事实上，对于他们其中的任何一个发行商来说都不

太可能保持他们产品的与众不同，因为使得 LINUX 的源代码得以高效发展的许可证条款要求他们与所有的发行商一起分享源代码。任何一个发行商只要一开发出新的特性，他们有的竞争对手都可以免费克隆它。

因为所有的发行商都深知这一点，所以甚至没有人想过要实施一个阴谋，一个和导致 UNIX 标准分裂的策略类似的计划。相反，LINUX 的发行商被迫以一种实际上对顾客和整个市场有利的方式进行竞争。那就是，他们必须在服务、技术支持、以及实际上能使得安装和使用都比较方便的设计方面进行竞争。

共同的开放的源代码还去除了垄断的可能性。当 LINUX 社区内的人们担心这一问题时，通常会抱怨一个叫"RED HAT（红帽子）"的名字，而"REDHAT"是 LINUX 最大的也是最成功的发行商，它几乎拥有美国市场上 90% 的份额。但是还有一个值得引人注目的事情，那就是在被大家期盼已久的 REDHAT 的 6.0 版本在 1999 年 5 月份宣布发行后的一段时间里，通过从 REDHAT 自己的 FTP 站点下载光盘镜像，一个图书发行商和许多其他光盘软件发行商就已经开始以比 RED HAT 更底的价格进行销售了，而且事实上在这段时间里 REDHAT 的 CD-ROMS 还没有真正的成批装船销售。

但是，REDHAT 自己并未对此事怒不可遏，因为他们非常清楚的知道他们没有也不可能拥有他们他们产品中二进制数据中的任一个比特。因为 LINUX 社区里的社会准则不允许他们这样做。在后来的日子里出现了 JOHNGILMORE 的著名的论断，那就是互连网上的人将对互连网的检查制度解释为对它的破坏和一些例行公事的程序。基于此，对 LINUX 负责的黑客们则巧妙地将企图控制源代码也解释为是对它们的破坏和一些例行公事的手续。对于 REDHAT 来说，他们如果反对对他们的新产品在发行之前进行克隆，这一行为将严重地使他们未来吸引开发商们进行共同合作的能力大打折扣。

也许就目前来说，以一种与法律相结合的形式来表达 LINUX 社区准则的软件许可证制度正积极主动的阻止了 REDHAT 对他们的基于开放源码产品的垄断。他们唯一能卖的就是一个品牌、服务以及与那些自愿付给他们钱的用户之间的技术支持关系。这不会让压倒性的垄断局面出现有太大的的可能性。

5.13 开放研发和再开发

投资者向开放源码世界投资的另一个原因就是要改变他。开发者逐渐感觉到他们可以从他们想干的事情中获得报酬，而不是用自己的正式工作的收入来维持他们对开放源码运动的爱好。象 RedHat，O'Reilly Associates 和 VA LinuxSystem 这样的公司正在探索通过雇佣并维持稳定且能干的开放源码程序员来建立半独立的研发机构需要多大的投入。

这种方式只有在公司通过迅速扩大市场所带来的收入能够足够用于支付那种研究实验室时才是经济上可行的。O'Reilly 之所以能够负担 Perl 和 Apache 的主要作者来完成他们的工作是因为经过努力公司能够出售和 Perl 以及 Apache 相关的书；VALinux System 能够让实验室有足够的经费来源的原因是随着 Linux 的繁荣，他们可以卖掉更多的工作站和服务器；RedHat 可以负担他的高级研发实验室也是由于实验室可以不断提升公司的 Linux 产品的价值并吸引更多的用户。

在将专利、商业秘密等知识产权看成是企业的掌上明珠的文化的熏陶中，这种思想（开放源码）对于传统软件产业的战略家来说简直是无法解释（尽管自由软件市场事实上在不断地扩大）。为什么你花钱来做的研究得到的成果却可以让你的每个竞争对手都可以无偿享用呢?

看来可以有两个合理的解释。一个是随着这些公司继续在他们的市场中保持领先位置，他们就可以从开放研发中获得巨大的市场占有率所带来的回报。通过开放研发来换取“明天”的利润，这似乎有些天方夜谈，不过有意思的是要不是真的如此，为什么那些公司都毫不迟疑的容忍了自由的存在呢?

在这个资本家都拼命盯着投资风险评估的世界上，虽然风险投资分析是必要的，但是这并不能很好的解释明星效应，因为实际上投资人自己也对投资风险不是很清楚。如果被问及，他们就会告诉你他们做了他们所从属的团体所认为是对的的事情。拙笔和前面所提及的三个公司的总裁非常熟悉，因此可以说明我所说的结论绝对不是骗人。实际上我还在 1998 年末亲自在 VALinux Systems 公司干过一段，因此我可以对他们提出一些“正确的”建议，我发现公司对我所做的基本上没有任何反对。

经济学家会问，那么如何为这些工作计算报酬呢？如果我们已经接受了前面提到的“做正确的事”的说法不是空洞的做作的话，我们接下来就会想到，“做正确的事”会给公司带来什么好处呢？对这个问题的回答既不令人惊讶，也不困难。实际上在其他产业，表面上的大公无私，实际上都是为了给企业赢得好的名气。

为名气努力，并将此看成是一种可以在未来的市场中得到回报的无形资产，这已经不是一件新鲜事了。这些公司的行为显示他们正在建立信誉，这是一个很高价值的多么大的利益啊。他们很明确的希望能够不惜高价请到真正的高人来做项目，并非为了直接从中赢利，即使是在股票准备上市前资本非常匮乏的阶段也是如此。而且至少到现在为止，这种做法已经开始从市场中获得回报了。

这些公司的头头们心里都十分清楚信誉对公司来说是多么重要。客户群中的志愿者们不仅帮助他们做研发，也是一种非正式的市场伙伴，这些都是他们的靠山。公司和用户之间的关系是非常亲密的，通常是建立在公司内部或外部相互信任的私人关系之上。

这些现象增进了我以前从另一个角度所作出的推断的理解。象 RedHat，VA 和 O'Reilly 这些公司和他们的客户以及开发人员之间的关系和传统的制造业完全不同。这是一种非常有意思的特别模式，是一种知识密集型的

服务产业。除了技术工业以外，我们还可以从法律界、临床医学界和学院中找到这种模式的影子。

实际上，我们可以看出开放源码公司雇佣优秀的黑客和大学聘请知名教授之间有异曲同工之妙。在实现方式上，二者都有些象工业革命前贵族们对高雅艺术的投资方式，一些方面的相似性是显而易见的。

5.14 由此及彼

资金支持（当然也要从中获利）源码开放开发的市场机制仍然在迅速的发展之中。本文中所述及的商业模型并不是最终的定论。投资商还在不断从软件产业变革的结果中不断总结经验，这种新模式面向服务的而不是强调保护知识产权，他们将会在一个适当

软件业在思想上的革命将给原来人们仅通过向 5% 的市场价值投资来赢利的方式带来好处；传统意义上服务业不如制造业有利可图（可是医生或律师会告诉你，服务业的创业者所获得的回报更高）。然而，当软件用户可以从自由软件产品中获得许多好处并可以节省开支的时候，从投资中可以获得更多的利润。一个类似的例子是从传统的语音电话网络向现在的互连网发展所带来的巨大影响。

对于节约开支和更好用的承诺正在创造一个巨大的市场机会，许多企业和风险投资商们开始来开拓这个市场了。在本文的第一份草稿完成的时候，硅谷一家非常著名的风险投资机构开始下了头注，他们投资了一家提供 24×7 的 Linux 技术支持的服务公司，一般预计在 1999 年年底之前，会有几家 Linux 厂商和一些与自由软件相关的股票上市，他们的融资应该会非常成功。

另一个很有意思的发展方向是系统性的创造一个自由软件开发上的外包市场。SourceXchange 公司 <http://www.sourcexchange.com/process.html>和 CoSource 公司 <http://www.cosource.com/> 分别代表了两种稍有区别的将减价拍卖模式应用于开放源码软件开发的新尝试。

整体的趋势已经很明显了。在前面提到的IDC预测中可以看出 Linux 会在 2003 年之前以比其他操作系统都快得多的速度增长。Apache 现在占有 60% 的市场份额，而且还在不断增长。互连网的传播是爆炸式的，象 InternetOperating System Counter 给出的调查报告显示 Linux 和其他开放源码系统已经是互连网上主机所采用的主流系统，而且在以比封闭系统更快的速度扩大市场占有率。不断开拓互连网领域自由软件的需要并不只是由编制更多的软件来决定，更重要的是各个公司的商业行为和软件的使用/购买模式使然。这个趋势现在看来正在不断加快。

5.15 结论：自由软件变革之后

在向自由软件形式过渡完成之后，整个软件产业将会是什么样子呢?

为了回答这个问题，有必要根据软件所需要为用户提供的服务程度将软件分分类，服务体现了软件的开放性，这种划分又是与软件所业服务的市场化程度紧密相关的。这个提法的精髓恰好与我们日常所说的三个名词相似：应用程序（基本没有商品化的服务，没有或缺少开放的技术标准）、构件（服务商品化、标准性很强）、中间件（需要一些商品化的服务、有技术标准但是不完善）。当前（1999 年）对于上面三种软件的典型例子就是字处理软件（应用程序）、TCP/IP 协议包（构件）和数据库引擎（中间件）。

前面关于分配方式的分析向我们展示了构件、应用程序和中间件三种软件形式将会以不同的方式向自由软件体系过渡，以及他们各自体现出的自由软件与封闭软件相结合的形式。还需要指出的是，在软件业的某一领域自由软件普及程度还要受到那里的网络影响力是否很强，软件企业倒闭所带来的负面影响程度以及软件产品在多大程度上还是一种商业上敏感的资本资源等因素的影响。

如果不局限于某个特定的领域，从软件业的整体角度考虑我们可以大胆的作出如下预言：

象因特网、互连网、操作系统以及其他需要在竞争的软件各方互相交叉的底层通讯软件等构件产品会逐渐全部开放，这些软件将由今天象 RedHat 这样赢利的软件发行商或其他服务机构将会与用户团体来共同维护。

另一方面，应用程序类型的软件会继续保持封闭的状态。这种软件通常是他们未公开的算法使用价值非常高或使用的技术非常先进，促使用户仍然愿意花钱去购买这些封闭源码的软件，同时这也意味着这种软件可靠性要求非常底，并且可能导致行业垄断的风险还在可以容忍的范围内。这种现象最有可能出现在网络影响比较小的垂直性市场领域中。我们以前提到的一个 lumber–mill 就是这种产品，1999 年最亮丽的软件产品 —— 生物分子结构识别软件也属于这一类。

中间件，象数据库工具、开发工具或其他用于特定领域的高端应用程序协议软件包将是一种自由与封闭的融合。这些中间件软件产品是会逐渐走向封闭还是开放或许将取决于软件的破产风险，为打开市场而所需的成本越高的软件将更需要开放。

不管怎样，为描绘一个完整的蓝图，我们仍然应该看到无论是应用程序还是中间件，这都是一个静态的划分。在前文“何时会开放”一节里面我们已经分析了对于任何一个软件产品都将要走过一个从理智的封闭到理智的开放这样一个生命周期，对整个软件产业来说同样是这个道理。

随着关键技术的普及和标准化，随着商品化的服务在软件产业中所占的比重越来越大，应用程序会逐渐转化为中间件，比如在将数据库前端接口和数据库引擎分开以后，数据库接口就成为了一种中间件。当中间件产

品所需服务越来越要商品化时，就轮到他们逐渐转化为开放源码的构件了，我们今天看到的操作系统的变革就是这种例子。

我们可以预料到在未来，随着自由软件所带来的强大竞争力，某个软件的最终命运将不是走向灭亡就是成为开放构件系统的一部分。虽然这对于那些打算永远从封闭软件中赚取利润的软件企业来说的确是个坏消息，但是软件产业作为一个整体仍然是一种产业，那时新的高层应用软件将不断开放，私有化的智力资源垄断某个软件将只有一个有限的生命周期，最终将纷纷转化为自由软件。

最后，我们要看到这种从封闭到开放的变革还是主要要由软件产品的用户来推动才能不断发展。越来越多的高质量软件将被创造出来并得到长期使用，而不是被某些人藏在密室里得不到发展。这种奇迹用 Ceridwen 的魔锅来比喻还不够恰当，因为魔锅变出来的食品如果不吃就会逐渐腐烂掉，而自由软件世界中的软件将是取之不尽的宝藏。在自由软件中你拥有最自由的自由，无论你是打算提供商业服务还是打算为他作出贡献，自由软件世界将向所有人提供一个不断积累、取之不竭的宝贵财富。

5.16 参考文献和致谢

[CatB]大教堂和市集 <http://www.catb.org/~esr/writings/cathedral-bazaar/>

[HtN] 开拓智域 <http://www.catb.org/~esr/writings/homesteading/>

[DL] De Marco and Lister，Peopleware 合著的 Productive Projects and Teams(New York; Dorset House，1987; ISBN 0–932633–05–6)

[SH] Shawn Hargreaves 写过的一篇关于如何将开放源码和游戏制作相结合的佳作 Playing the Open SourceGame <http://www.talula.demon.co.uk/games.html>.

在完成本文的过程中，通过与 David D. Friendman 的几次激烈讨论帮我进一步提炼了介绍如何加强开放源码团体合作的“翻身的平民”一章。感谢 Marshall van Alstyne 为我指出了“热门信息产品”的确切含义，我欠了他一个人情。Indiana 组织的 RayOntko 给了我许多有益的批评。还有许许多多在我今年 6 月发表演讲时的热心听众也给了我很多帮助，如果你是听众中的一员，你就会明白我指的是谁。

在我公布这篇文章以后，我还通过电子邮件收到了许多关于自由软件发展模式的材料，这些材料不断充实了这篇文章的内容。LloydWood 指出了“将来获益”自由软件发展模式的重要性；Doug Dante 提醒我注意“未来免费”这种商业模式；Lionel OlivieraGresse 帮我给一个商业运作模式起了一个更好听的名字；Stephen Turnbull 对于无视自由骑士现象给了我当头一棒。

5.17 为何封闭驱动程序源码的硬件厂商会浪费投资商的金钱

外围设备开发商，象网卡、硬盘驱动器或显卡的制造商，他们的传统作法就是将驱动程序的源代码封闭起来。但是这种现象现在已经有所改变，比如 Adaptec 公司和 Cyclades 公司已经习惯于将他们的各种板卡的驱动程序源代码和相应文档公开化。不过要想让开放源代码成为一种普遍的作法还是有不少困难的。在本附录中我们就是打算澄清在商业领域中仍然维持封闭源代码体系的一些错误观念。

假定你是一个硬件制造商，你也许会担心将驱动程序代码的开放会泄露你硬件如何工作的许多重要秘密，从而让你的竞争对手可以通过分析你的源代码来给你造成一种不公平的竞争环境。这种想法在三、五年才会将产品更新换代的时代里也许还站得住脚；但是今天即使将源代码开放，你的竞争对手也将不得不花费占整个产品更新周期的一大部分来琢磨你已经公开了的代码，因为现在产品更新的周期大大的缩短了，你的竞争对手将没有足够的时间来好好思考和革新他们自己的产品。所以说他们去研究你开放的源代码的时刻实际上已经钻进了你的圈套。

不管怎样，在今天代码中的秘密不会被隐藏很久了。硬件驱动程序并不象操作系统或应用程序那么复杂，他们一般都很小，很容易被反编译和模仿，这种活连一个十几岁的电脑初学者也可以搞定，而且实际上常常也被这些人搞定。可以毫不夸张的说，世界上现有数以千计的为 Linux 或 FreeBSD 工作的有激情的优秀程序员，他们愿意为任何一种新的板卡编写驱动程序。由于许多种类的硬件设备有着相对简单和标准化的接口规范，比如常见的磁盘控制器或网卡，热情澎湃的黑客们即使在没有文档也不需要反编译已有的驱动程序的条件下就可以迅速的写出正确的驱动程序来，而且常常比原生产厂家还要来得快。

即使遇到象显卡这样的复杂设备，也难不倒用反编译工具武装起来的牛人。这种工作即不需要花费很大的精力，也很难说是否违法，而且在全球程序员的共同努力下，已经可以对 Linux 做任何在法律上合法反向工程了。从 Metalab 网站查一查 Linux 核心和设备驱动程序库所能支持的硬件类型列表，你就会立刻明白前面所言非虚，Metalab 的网址是：<http://metalab.unc.edu/pub/Linux/hardware/!INDEX.html>。访问该网站时你还可以留意一下新的驱动程序正在以何等迅速的速度不断涌现。

保守你驱动程序中的秘密从短期效应上来说还是有诱惑力的，但是从长期战略的角度来看则不可取，特别是当你的竞争对手都已经将源码开放的时候。如果你非要固执的封闭你的源代码，那就只能将那些代码烧到电路板上的 ROM 中，而只对外公开访问接口了。所以赶紧开放你的源代码吧，迅速扩大市场，你要相信自己有能力通过自身的不断思考和创新来吸引更多的本来属于你的竞争对手的潜在用户群。

坚持走封闭的路线是一条死胡同，你的秘密将不可避免的被逐步暴露，你将无法得到自由程序员的帮助，也没有什么愚蠢的竞争对手会去花时间模仿你的设计。更重要的是你如果及早采纳开放的思想本来可以获得更广阔的发展空间，但是你却遗憾的错过了。由于你的设备太保守、缺少资料和固步自封，并且不能认识到你自己的错误，因此互连网上大部分的网络管理员和超过 17% 的商业数据中心所形成的巨大市场将把你的硬件设备从他们的采购清单中删除，而把目光转向其他开放的硬件厂商中去。

5.18 本文档修订记录

你现在看到的是本文档的 1.14 版

在下面的列表中，一些微小的修订和印刷版就不再列出了。

1999 年 5 月 20 日，1.1 版 —— 草稿

1999 年 6 月 18 日，1.2 版 —— 第一用于私下交流的版本

1999 年 6 月 24 日，1.5 版 —— 对外公布的第一个版本

1999 年 6 月 24 日，1.6 版 —— 作了一些小改动，给出了“hacker”的定义。

1999 年 6 月 24 日，1.7 版 —— 澄清了一些标准

1999 年 6 月 24 日，1.9 版 —— 增加了关于“将来获益”、“未来免费”发展模式的讨论和关于封闭的代价的章节

1999 年 6 月 24 日，1.10 版 —— 给“刀片”模式取了一个更好的标题

1999 年 6 月 25 日，1.13 版 —— 更正了关于 Netscape 公司 13% 收入的问题，增加了关于自由骑士的分析，更正了封闭网络协议的列表。

1999 年 6 月 25 日，1.14 版 —— 增加了 e-smith 公司的例子

1999 年 7 月 9 日，1.15 版 —— 更新了关于硬件驱动附录的内容，并在 Rich Morin 的帮助下给了“热门货”一个更好的解释。