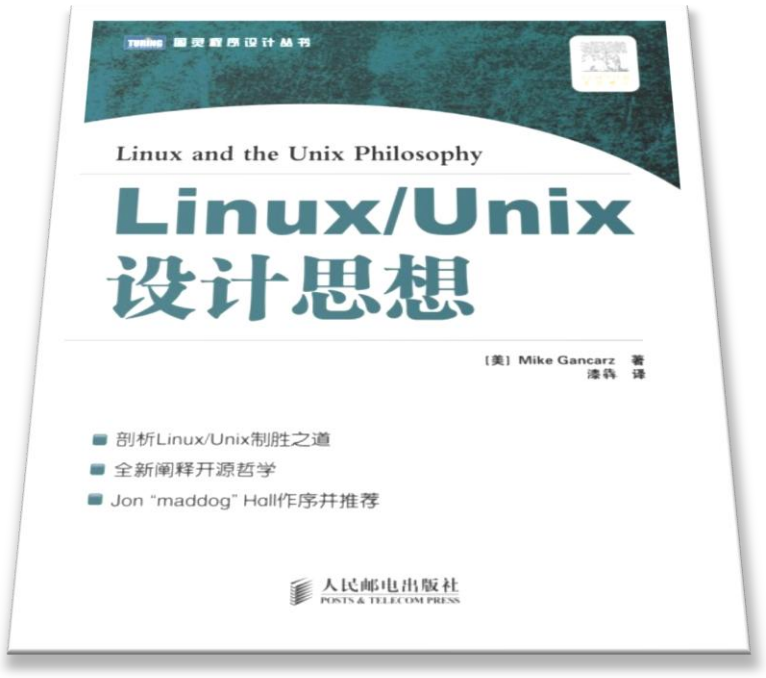
试读版

Linux/Unix 设计思想



每一本书都应该有一句献辞。

本书献给有勇气率先尝试Linux的人们。

继续前进，朋友们！

——MG

译 者 序

“布道者”指的是那些坚定地信仰某一宗教，并且不遗余力地向人们传播此宗教的修道者。本书的作者 Mike Gancarz 就是这样一位“技术布道者”。他是 Linux/Unix 最主要的倡导者之一，也是最早开发出 X Window System 的先驱。他把一些在 Unix/Linux 社区里口口相传的哲学思想总结提炼，最后集结成这样一本完整的 Linux/Unix 哲学理论书呈现给所有的读者。是的，我们每一个人都能够站在巨人的肩膀上。

这本书写于 2003 年，要知道每隔几年计算机世界就会发生沧海桑田般的变化。Google 的热潮才刚刚过去，我们现在又有了 Facebook、Twitter，还有云计算。“吹尽狂沙始到金”，在这些热潮的背后，Linux/Unix 一直都是计算机世界的重要基石，也可以说其哲学教义是这一波又一波网络风潮的动力源泉。而且，正如作者所说的，哲学就是哲学，它不会过时。

这并不是一本讨论技术细节的书，书如其名，它阐明的要点在于“思想”、“道”以及“哲学”。它没有什么高深莫测的知识要点，也没有那些让人觉得晦涩难懂的技术细节，作者通过实例将 Linux/Unix 的哲学思想娓娓道来。翻阅这本书，你可以体会到 Linux/Unix 给计算机世界所有人们带来的自由和乐趣。Linux/Unix 哲学体系像是一个“宗教”，也代表着一种先进的科学技术文化，其中包含了协作、创新、自由等人类孜孜不倦追求的精神。它的成功不只是因为技术优势，更重要的是它所蕴涵和贯彻的开放与共享精神。

如果你是一名 Linux/Unix 的高级用户，尽管你大可安心享用你喜欢的这个工具，而不去关心它的基础理论。不过，多了解一些内在机制没有坏处，这是一个锦上添花的过程。如果你是一位门外汉，那么来吧，它会给你打开一扇窗。原来，除了 Windows 之外，操作系统的世界里也有别样的风景。如果用“武侠”来作一个类比，这本书就好像是一部教你修炼内功的秘笈，无论新手老手，修炼基本内功都是一件必须持之以恒甚至可以毕生研习的事情，而同时我们也要知道，有时候优秀程序员和普通程序员水平差距的关键也正在于此。

本书的组织结构在作者 Mike Gancarz 自己撰写的前言中已经做了归纳总结，此处不再赘言。

此外，我在翻译本书的过程中，得到了很多朋友的帮助，他们是：刘园园、向梓鑫、冯海涛、常亚平和冯乐宇。我还要感谢图灵的编辑杨海玲、何建辉等，谢谢他们的信任与大力支持。“此外，在图灵社区书稿试读活动中，李琳骁、周兵、臧秀涛、张伸等网友也提供了很多宝贵的意见，我也要向他们表示衷心的谢意。

序

对我来说，1969 年发生了三件大事。

那时候我还只是美国东部一所知名大学的学生，无意中我发现了一项引人入胜的新业务——编写计算机软件。当时的大街小巷里根本就没有什么计算机商店，更没地方去购买现成的包装好的现成软件。事实上，那会儿普通计算机的内存还不到 4000 个“字”（word），处理器每分钟只能处理几百万条指令（而现在的处理器每秒就能处理几十亿条指令），并且，普通计算机的售价甚至高达几十万美元。在当时，开发软件的宗旨就是充分利用机器的功能。那时候，编写软件都是从零开始，由客户确定并提供输入内容，同时也要明确他们预期的输出。随后软件团队（无论是公司内部的团队还是外部的顾问）负责编写软件来完成该项转换工作。如果软件不能工作，那编程人员就得不到报酬。显然，就像任何量身定做的项目一样，获取软件的代价不菲，硬件产品也是如此。

幸运的是，DEC 公司用户协会（DECUS）有一个程序库，人们可以将自己编写的软件贡献出来供他人使用。程序作者会将他们的代码提交到 DECUS 程序库里，然后程序库管理人员会制作一份程序目录，复制和邮寄目录的费用都是由那些要求得到程序副本的人们来承担。软件是免费的，但它们的分发（以及印刷目录等的费用）却需要费用。

我通过 DECUS 得到了很多免费软件。这简直就是太棒了，因为当时我只是一个穷学生，没钱去购买定制的软件。我必须在软件和啤酒之间做一个抉择，要知道 1969 年用来购买文本编辑器软件的 5 美元可以买 10 扎啤酒呢。如果要试用并购买“商业”软件的话，那我就该喝西北风去了。

为什么这些人会把自己编写的软件贡献出来呢？他们之所以编写软件，是因为他们在工作或研究中有使用软件的需要。他们（大方地）认为，或许其他人也能从中受益。他们还（合理地）希望软件的使用者可以帮他们改善这些软件，或者至少给他们提一些改进意见。

1969 年发生的第二个重大事件就是，Ken Thompson、Dennis Ritchie 和其他一些在新泽西州 AT＆T 公司（美国电话电报公司）贝尔实验室的研究人员一起开始编写一款操作系统，最终这成为众人皆知的“UNIX”。在编写该操作系统的过程中，他们形成了一套理论，那就是操作系统必须由一些小的可重复使

用的模块组成，而不是一组大型的独立程序。此外，他们还为操作系统开发出具有“可移植性”的架构，完全不必理会计算机的主 CPU 指令集，它就能运行在从最小的嵌入式设备到最大的超级计算机等各种设备上。最终，这个操作系统辗转流传到了大学里，然后又被传播到美国各大公司。在这些地方，它主要是在服务器上运行。

第三个重大事件在未来很大程度上改变了我和其他几百万人的生活。但当时，除了身在芬兰赫尔辛基一对自豪的父母外，没有人注意到这件事情，那就是 Linus Torvalds 的诞生，他就是未来的 Linux 之父。

接下来的十年，计算机科学稳步发展。各大公司都开始投资开发新的软件和技术。虽然每家公司对计算机应用的研究日趋深入，但他们仍遵循着一个原则，那就是购买量身定做的软件。当然，按照今天的标准来说，他们所使用的计算机容量出奇地小、速度出奇地缓慢、代价出奇地昂贵。然而，AT＆T 公司的 Unix 开发者和给予过他们帮助的一些大学一直在创建一种鼓励软件重用的操作系统，同时，为了避免重写代码、重复创造，这些人并没有一味追求代码的高效运行。大学和计算机科学的研究者们乐于获取操作系统的源代码，以便他们能够共同完善其功能。那真是一个美好的年代。

然后在 20 世纪 80 年代初又发生了三件大事。

第一件事是 Unix 的商业化。这时候，小型机的成本已经下降，而编程和用户培训这一块的成本开始超出了基本硬件成本。Sun 公司开创了一类新市场，其中的客户都要求拥有一个可以运行在多种处理器之上的“开放式系统”，而不是当时像 MVS、MPE、VMS 以及其他的商业操作系统那样的“专有”系统。一些公司如 DEC、IBM 和惠普都开始考虑为自己创造一个“商业化”Unix 版本。为了保护他们的投资，并从 AT＆T 公司那里获得更低的专利费用，他们只采用二进制文件的形式来发布自己的系统。因此，Unix 系统源代码的价格变得非常昂贵，几乎没人能负担得起。通过共享方式来改进 Unix 的举措便被暂时搁置。

第二件事便是微处理器的出现。最初，它创造了一种经由 BBS、杂志、计算机俱乐部来共享软件的氛围。但随后新的概念又产生了，那就是“带塑料薄膜包装的”软件。这些软件是针对那些 “商品化”处理器架构而编写的，比如英特尔或摩托罗拉的产品；那时硬件的生产量级也只是上百或数千而已。最早的商业化操作系统是 CP/M，后来变成微软的 MS-DOS 和苹果的操作系统，商业软件产品的数量持续增长。我仍然记得自己第一次走进电脑商店时的情景，我看到了很多来自不同厂商、有着不同硬件架构的计算机摆在货架上，可软件产品却只那么三四个（包括一套文字处理软件、一款电子表格、某种“调制解调器软件”产品）。当然，这些软件产品都不附带其源代码。如果在有了这些软件及其相关文档后，还是无法让它们运行的话，那你就无计可施。渐渐地，以公开协作方式编写软件的方法被一些商业做法所取代，这些做法包括采用二进制文件发布产品、最终用户许可证（告诉你该如何使用你所购买的软件）、软件专利和永久版权。

幸运的是，在那些日子里，每家公司的客户数量都不太大。只要致电各公司的服务热线就可以得到技术支持。不过如果想要以这些公司开发的软件为基础添加新功能，那是一项几乎不可能完成的任务。

第三件事实际上是前两个事件的结果。在麻省理工学院的一个小办公室里，一位名叫 Richard Stallman 的研究员决定解密 Unix 和其他软件的源代码。他讨厌这种愈演愈烈的软件闭源趋势，并决定于 1984 年启动一个项目来编写一个永远开放源代码的完整操作系统。当然，源代码自由发布有其副作用，也就是对那些愿意获取源代码来编译自己操作系统的人们来说，后续开发出的软件也应该永远是免费的。他把这个项目称为“GNU”，意为“GNU 不是 Unix”（GNU is Not Unix），以表达他得不到 Unix 源代码的愤懑之情。

时光依然在流逝。微软成为了操作系统产业的主导。与此同时，其他系统供应商也发布了不同版本的 Unix，打着所谓“创新”的名号，其中大部分版本都互不兼容。整个市场充斥着各种各样僵化死板的商业软件，每个精心包装的程序都是为某个需要独特解决方案的商品市场而编写的。

接下来，在 1990 年前后，再一次发生了三件大事。

第一件事情是一小群 Unix 专业人员围坐在一张桌子旁，比较着不同市场中不同类型的软件，其中一个人发问道（这个人就是我）：

“你为什么喜欢 Unix？”

在座的大多数人一开始都不知道该怎么回答，这个问题久久地萦绕在会议桌上方，然后我们每个人开始给出自己热爱并忠于这个操作系统的理由。“代码的重用”，“高效、精心编写的实用程序”，“简单，但是优雅”，诸如此类理由被人们提出来并得到了详细阐述。我们当中没有一个人注意到，在大家提出这些想法的同时，有一个人正在做详细的记录。后来，这些想法成为了本书第一版的核心思想，而且那也是市面上第一次出现这样一本关于 Unix 设计思想的“哲学书”。

为什么大家需要一本关于“Unix 设计思想”的书呢？这是为了帮助 Unix 初学者了解该系统的真正威力和优雅设计底蕴。向他们展示在编写程序时使用 Unix 下的工具和结构可以节省大量人力物力。Unix 还可以帮助他们扩展计算机系统的功能，而不是帮倒忙。可以这么说，Unix 让人们能够站在前人的肩膀之上。

第二件事情是，1990 年 GNU 项目已完成大半。它们包括命令解释器、系统工具、编译器、各种库文件，等等。这些 GNU 软件与其他一些免费软件如 Sendmail、BIND 和 X Window System 结合在一起，只是唯独少了操作系统的核心，也就是被称为 “内核”的那一部分。

内核是系统的大脑，它控制程序运行的时间、程序能够得到哪些内存、程序可以打开什么文件，等等。它被留在最后实现的原因是因为内核每天都在发生变化，随时都在改进。一个没能提供任何工具或是命令解释器的内核没有任何用处。多年以来，让所有的工具能为其他操作系统和其他内核所用是一件非

常有意义的事。

第三件事情就是，1990 年的 12 月 Linus Torvalds 已经成长为一名就读于芬兰赫尔辛基大学的学生，而且他刚刚得到了一台最新的英特尔 386 电脑。Linus 认识到，当时微软的操作系统并没有充分利用 386 的能力。于是，他决定自行编写一个内核，将它和其他免费的软件结合在一起，创建一个完整的操作系统。待到事后他才确定，这个内核应该遵循 GNU 项目的通用公共许可证（General Public License，GPL）。他在互联网上的一些新闻组发布了关于这个内核项目的消息，并开始了工作。

1994 年 4 月我了解到 Kurt Reisler，也就是 DECUS 的 Unix SIG（Special Interest Group，特别兴趣小组）的主席，试图筹集资金让一名程序员来到美国与 DECUS 的 SIG 小组讨论他正在开发的项目。最后，我请求 DEC 的管理层提供了这笔资金，因为我相信 Kurt 一贯秉持的远见卓识和敏锐的洞察力。1994 年 5 月，我参加了 DECUS 在新奥尔良举办的这次活动，并见到了这位程序员，他就是 Linus Torvalds。从第一眼看到他的操作系统开始，我的生活就此改变。

在那之前的几年里，我一直都在大力主张重新编写这个我热爱了多年的 Unix 操作系统，不过其实现方式必须能够鼓励人们在他人的工作基础上去阅读、修改和完善源代码。事实上 Unix 的成长远远超出了大多数人对它的基本认识，因为自由软件运动已经囊括了数据库、多媒体软件、商业软件以及其他对人们有价值的软件。

而且，软件的生产潮流又一次改变了。硬件工艺越来越精细，价格却越来越便宜，互联网上的协作开发也越来越容易，软件信息传播的速度如此之快，以至于百转千回之后，主流的开发组终于能够走到一起来开发那些可以帮助他们解决自己问题的软件。软件再也不需要由那些自诩为“德鲁伊”（druid）[1]的人来开发，他们坐在高耸入云的“大教堂”里使用昂贵机器创建程序。现在，每个人都可以发出“代码就在这里”的呼声。这些人可能就坐在自己的家中或教室里，他们都能为计算机科学的发展作出自己的杰出贡献，无论是身在美国、巴西、中国还是芬兰的赫尔辛基。成千上万的项目开始启动，不计其数的程序员在为它们卖力工作，这种态势不断扩大，发展速度也日渐迅猛。

未来的计算机编程世界里，单凭程序员小团体不大可能可以完成百分百满足所有人需求的软件。相反，网络上会出现一大批以开源形式存在的项目，它们的开发人员盼望着业内顾问和增值的经销商能够将这些软件加以改进，并定制出针对客户精确需求的解决方案。

Linux 和 GNU 项目的理念看似是 Unix 哲学的“下一个发展阶段”，其实它不过是 Unix 的强势回归。*The UNIX Philosophy*[2] 一书中阐明的准则直到今天依然适用，甚至其重要性变得益发明显。源代码的可用性还让人们清楚地了解到这些编程大师们是如何创建他们的系统，而且也能激励你自己去编写

[1] 德鲁伊属于欧洲“凯尔特”人中的特权阶级，是部落的支配者、王室顾问、神的代言人，有着至尊地位。——译者注

[2] *The Unix Philosophy*，作者Mike Gancarz，Digital Press，1995年出版。

功能更强、运行速度更快的代码。

借助巨人的肩膀，愿你能攀得更高。

祝学习愉快！

Jon“maddog”Hall

Linux 国际协会（Linux Internopational）执行理事

致　　谢

一本书的诞生就像是孩子的出世：每个都是不同的。我第一本书 *The UNIX Philosophy* 的撰写和出版都相当顺利。但如果预先知道在写第二本书的时候会面临重重困难，我都不大可能会签下这份出版合同。我当时真应该三思而后行。

在经历了发生在写作和出版本书过程中的个人波折之后，我承认这本书就像是一种个人和哲学两方面的宣泄。所以，伟大全能的上帝功不可没，他再一次引领我渡过难关。他是我优先并重点感谢的对象，上帝创造了我，使我得以完成这个使命。

我要感谢巴特沃斯–海纳曼（Butterworth-Heinemann）出版集团旗下数字出版社（Digital Press）的策划编辑 Pam Chester，是她不遗余力地劝说我创作这个新版本，并且每当出现问题，她都不断地鼓励我。我还要感谢她的老板 Theron Shreve，是他的耐心让本书最终付梓。

我很幸运，有一群杰出的开发人员帮我审阅这本书。我要谢谢 Jim Lieb、Scott Morris、Rob Lembree 和 Jes Sorensen 提出的精彩见解。他们的意见促使我从不同的视角去看待事物，并且深化了本书中不少有趣的内容。他们还在最后阶段给了我不少鼓励。

我尤其要感谢的是 Jon“maddog”Hall，不仅因为他给本书写了一篇精彩的序言，更因为他倡导 Unix 和 Linux 的巨大努力，这给我们所有人塑造了一个典范。几乎没有人能够像他这样坚持不懈，把 Unix 的理念传达给这么多人。

我还要感谢数字出版社的前任出版人 Phil Sutherland，我们就本书最初的想法进行了一系列的电子邮件交流。尽管最后本书并没有以我们两个之前设想的方式完成，但他劝服我创作本书的努力并没有白费。这件事情就是需要花费更长一些时间。

家里有一个作者会让每一位家庭成员体会到与作家一起生活的滋味。谢谢你，Sarah，你为我打了那么多字，这个活儿对你来说一定很无聊。还有 Adam，写书的时间本是应该用来与你玩彩弹射击的，对你作出的牺牲我表示深表谢意。最重要的是，我要感谢我的妻子 Viv 始终如一地支持整本书的创作，我欠你的“老公该做的事情”的清单现在肯定都有一千多米长了。

引　言

The UNIX Philosophy 出版了几年之后，数字出版社（Digital Press）的前出版人 Phil Sutherland 再次和我联系，约我创作一本关于 Linux 的书。他认为 Linux 很快就会成为计算机世界的重要角色，并相信市场需要一本“Linux 哲学”之类主题的书。我们通了一年多电子邮件，就这个主题展开了深入探讨。市场的需求变得日益明朗，可写的东西也很多，只是我总觉得还没有找到感觉。

Phil 督促我写一本 Linux 版的 *The UNIX Philosophy*。从市场营销的角度来说，很有道理；然而，我却观察到 Linux 社区有一些不同寻常的迹象，Unix 的世界也正勃发出一股新鲜活力。很明显，老牌 Unix 信徒已经成功地将有关“Unix 思维方式”的思想灌输到了新一代人的头脑里，这些人不但包括充满激情的黑客，还有其他热衷于在自己机器上探寻 Unix 奥秘的狂热爱好者。是的，Linux 就是 Unix，只不过它已不再是上一个时代的 Unix。

我也曾考虑过写一本名为“The Linux Manifesto”（Linux 宣言）的书，在里面描述这种创新精神，可最终我清楚地意识到，“开源”才是 Linux 的精髓所在。因此，为了更好地契合主题，我得把“Linux 宣言”更名为“开源宣言”。但是，开源社区领袖 Eric Raymond 早就创作过一本关于“我们为什么要这样做”的经典大作——*The Cathedral and the Bazaar*[3]。我并不想东施效颦，于是，便只能暂时搁置这个想法。

不过，虽然最初的想法没有形成作品，好想法却总会改头换面，以一种不同的形式再度出现。两年前，数字出版社的 Pam Chester 向我指出，*The UNIX Philosophy* 中的一些信息已经过时，问我是否有兴趣做一下修订。可是，哲学怎么能修订呢？如果它就是真理，那么，不管是一个月、一年，还是几百年以后，它始终都是真理。

后来，我终于想明白了。*The UNIX Philosophy* 描述了一种 Unix 的思维方式，它其实就是“第一代系统”；但现在，它正在演变成“第二代系统”。而且就本质来说，相比“第一代系统”，这“第二代系统”是一个更全面、更成熟、更有价值的版本。虽然 *The UNIX Philosophy* 讲述了其最基本的原理，但是本书会将原来的概念提炼升华，而且还探索到新的领域。当然，它也仍然保留了原来的哲学信条——因为，真理永远是真理。

因此，*The UNIX Philosophy* 的读者们会发现，本书有很多他们已经谙熟于心的思想。这是因为，Unix 哲学的基本准则并没有发生改变。不过，我还是重新审阅了每一个章节，仔细思量该如何才能恰如其分地在 Linux 语境下阐明这些准则。从某种意义上来说，这本书是 *The UNIX Philosophy* 的修订稿，但它也探索了新的领域。本书的许多技术审稿人在反馈中也都谈到了这一点。这本书给了他们有别于前版的全新视角去审视 Unix。一方面，它描述了 Unix 哲学对 Unix 程序员编写操作系统所产生的影响；另一方面，它还提出了一个更为

[3] *The Cathedral and the Bazaar:Musings on Linux and Open Source by an Accidental Revolutionary*（大教堂与集市：一个偶然革命引发的关于Linux和开放源码之深思），作者Eric S. Raymond, O'Reilly&Associates，2001年出版。

大胆的命题，那就是，展示这套哲学理念对计算机世界其他领域的影响，其影响甚至还涉及计算机世界之外的天地。

Linux 和开源软件的出现，使得软件世界面临洗牌。Unix 哲学正在成为计算机行业的主导思想。很多人都采用了它的准则。这些思想造成了整个行业的巨大动荡，原来的偶像纷纷被抛弃，人们开始探寻一种不同的方式，以个人角度去利用计算技术。Linux 正好填补了这一空白，为主流电脑用户提供了使用 Unix 的第一手体验。越来越多人开始使用 Linux，他们见证并亲历着这种全新的工作方式。另一方面，Unix 的信徒也发现，个人计算技术又重新回到了他们熟悉的领域。

其实，Linux 哲学就是 Unix 哲学的加强版，这是我要着重强调的。当然这与一些人的观念冲突了，他们宁愿相信“Linux 不是 Unix”。不过我由衷地承认，作为一种现象，Linux 确实与 Unix 表现出来的特性颇为不同。在软件开发过程中，Linux 社区有意克服了专有软件的开发模式，但 Unix 社区一直没有清晰地意识到其实它也应该这么做。

Linux 社区较为精明，也更了解市场。它意识到，Linux 想要成功，就必须在每一轮竞争中都出其不意，想别人所不能想，做别人所不能做。迄今为止，它的表现非常出色。据传，Linux 开发人员已经超过一百万人。他们誓言要去“攻占”其他操作系统开发人员的地盘，在每一个计算机重要领域里都取得杰出成就——不管是安装使用的便捷性、图形用户界面、硬件兼容性、可靠性、安全性、整体性能、网络开发、数据库，还是游戏，等等。

经过整个 Linux 开发社区的不懈努力，Linux 已经成为计算机行业中不可或缺的一员，任何人都无法否认这个事实。

谁会是本书的受益者

最早的时候，只有系统程序员才会对 Unix 的资料感兴趣。今天，Linux 用户和开发人员组成了一个更加多元化的社区，容纳着各种个人爱好和专业兴趣。在 Linux 成为计算机世界主流的同时，那些从未听说过 Unix 的人们就已经见过或用过 Linux，甚至拥有了自己的 Linux 系统。在本书中，我避免谈及那些底层的技术细节，力求给这些人提供 Unix、Linux 和开源软件中极富启发性的观点。这些未曾深入研究过其哲学教义的人们会发现，他们无需陷在命令参数、编程接口等技术细节里，就可以通过本书了解到 Unix 的思想，以及这些思想在 Linux 中的实现。

Linux 开发人员可以从这本书中领会到 Unix 的奠基性思想，同时，也会从介绍 Linux 是如何接受并发扬这些思想的内容中获益。他们会认识到，尽管 Linux 拥有出色的图形用户界面环境，但是其内在价值是源于它背后的哲学理念。想要成为一个真正的 Linux “超级用户”，就必须去了解：为什么 Linux 基于文本的理念和工具会有着压倒性的竞争优势？

“老牌”Unix 程序员可以考虑读一读本书，这会让他们明白，为什么 Unix 没有消亡，而且为什么它永远都不会消亡。在 Linux 的世界中，Unix 如凤凰涅槃，以功能更强的新形式重生。这本书还会告诉他们，在新的千年里，Linux 将如何以全新的、激动人心的方式来满足计算需求，同时依然固守着初始的

Unix 哲学准则。

在当前快节奏、竞争日趋激烈的开发环境下，一些经验丰富的开发人员经常会感到，迫于巨大压力他们往往会背离优秀软件的设计原则。然而，很多情况，公司只是为了追赶最新潮流而使用新的开发流程，而不是为了确保他们所选择的方法有足够的事实根据。那么，当设计中有看起来无法解决的问题，或是解决方案的前提条件令人心存疑虑时，这本书可以帮助你进行理性反思。

今天，商业世界的 IT 主管们对 Linux 操作系统也有所耳闻。出于节约成本的目的，许多人正在考虑将 Linux 引入到他们的环境中。他们通常会采取这样的策略：在一些不那么关键的应用上，使用相对便宜的英特尔或 AMD 处理器的 Linux 服务器来取代老旧的 Solaris 和 HP-UX 平台。随着 IBM 公司和甲骨文公司正逐步加强他们的 Linux 产品，相较前几年而言，会有更多人接受将主要服务器切换到 Linux 平台的方案。

然而，迄今为止，IT 主管们一直不敢将自己的桌面用户迁移到 Linux 平台。这本书会让他们意识到，Linux 不仅是一个可行选项，实际上它还是*更好的替代方案*。他们会发现，相比同类型的 Windows 产品，Linux 和开源软件蕴含的哲学实际上使得台式机更易于维护，成本更低，也更安全。这些哲学理念不仅让 Linux 成为一款理想的服务器，也会在桌面产品上发扬光大。所以，在人们不以为然地回应“是的，但是……”之前，我只想说一句话：来读一读本书吧！

最后，从事抽象思维工作的人也能从 Unix 哲学的学习过程中发现价值。可以这么说，很多软件开发的技巧同样适用于其他行业。作家、平面设计师、教师和演讲家可能会发现，“快速建立原型”和“巧用杠杆效应”在他们各自的领域也能发挥作用。而且，随着大量价廉物美甚至是免费的 Linux 应用程序的出现，人们可以尽情享用那些在过去因太昂贵而无法使用的工具。

章节概述

第 1 章——Unix 哲学：集思广益的智慧。这一章探讨了 Unix 哲学的发展史，以及它是如何成型的。我还概要介绍了一些 Unix 哲学的基本准则，作为对后续章节中长篇论述的序曲。

第 2 章——人类的一小步。这一章说明了为什么以小模块来构建大型系统是最佳的方案。它详细探讨了无论是在软件系统还是现实世界中，小型模块都能较好地通过接口组装在一起。这一章的最后部分阐释了“让程序只做好一件事”的重要性。

第 3 章——快速建立原型的乐趣和好处。这一章强调，想要设计一款成功产品，就要尽早建立原型。我们将讨论“人类构建的三个系统”这一主题，这是所有软件都要历经的开发阶段。如果想要构建三类系统中最完善、最稳健的那“第三个系统”，那“快速建立原型”就是最快也是最佳的途径。

第 4 章——可移植性的优先权。这一章以不同视角来审视软件的可移植性。这里会强调，设计过程中软件开发人员必须在效率和可移植性之间作出取舍。本章以雅达利公司的视频计算机系统（Atari VCS）为范例来探讨高效率和有限

移植性。这一章还强调了一个非常重要却往往被人们忽视的目标，也就是数据的可移植性。此外，这一章还对典型 Unix 用户的工具集合做了个案研究，并借此完美地验证了一个事实，那就是可移植性能够延长软件的使用寿命。

第 5 章——软件的杠杆效应。这一章讨论了“软件杠杆效应”，也就是可重用组件带来的巨大影响。我们可以看到，怎样使用 shell 脚本来获取超强的软件杠杆作用。

第 6 章——交互式程序的高风险。这一章开篇就定义了什么是“强制性的用户界面”（Captive User Interface），并建议开发者们少用这类界面，而是集中精力去让程序之间能够更好地交互。这一章表达了一个思想：所有的程序都是过滤器。这一章末尾将讨论“在 UNIX 环境中的过滤器”。

第 7 章——更多 Unix 哲学：十条小准则。这一章列举了 Unix 开发人员普遍遵循的一些概念，但它们没被当做是 Unix 哲学的主要内容。由于这一章探讨的是更为底层的概念，对技术不那么感兴趣的读者可以跳过这一章。然而，纯粹的 Unix“信徒”很可能会发现这一章的内容相当有趣。

第 8 章——让 Unix 只做好一件事。这一章通过 Unix 邮件处理程序 MH 来说明该如何构建良好的 Unix 应用程序。章节末尾还对 Unix 的哲学准则加以总结，展示怎样才能将各个模块集结在一起以获得强大的功能。

第 9 章——Unix 和其他操作系统的哲学。这一章将 Linux 和 Unix 的理念与其他几个操作系统的哲学进行了对比，从而强调了 Unix 的独特性。

第 10 章——拨开层层迷雾：Linux 与 Windows 的比较。这一章开篇便对 Linux 和微软的 Windows 做了一个深层比较。通过事实来对比“基于文本的系统”与“基于图形用户界面的系统”的情况。这一章还归纳总结了 Windows 系统中哪些方面遵循了 Unix 哲学，而哪些又是完全与之背离的。

第 11 章——大教堂？多怪异。这一章探讨了 Linux 社区推广的开源软件（Open Source Software，OSS）概念是如何契合 Unix 哲学理念的。当开放源码软件在各个层次均采用“Unix 的思维方式”时，它的传播效果最佳。

第 12 章——Unix 的美丽新世界。这一章论述了新技术领域采用 Unix 哲学准则的情况与前景。它着重于几大关键点和一些不那么关键的技术，并借此来说明 Unix 的想法不再是只为 Unix 开发人员所用。对于这一章，本书技术审稿人最常见的评论就是：“我从没想过可以用那种方法。”读一读这一章，它会开阔你的眼界。

经过以上对本书的概述，你可能已经看出来，我在文中尽量避免涉及过多的技术细节，因为各个 Linux 发行版中细节都有所不同。我在书中强调，Unix 哲学其实是一种设计方法论，它偏重于更高层次上的通用方法，而不是某些具体的细节。

我努力保持本书的文字轻松易懂。虽然可能有人喜欢阅读那些冷冰冰、不带感情色彩的大部头技术资料，但我们大多数人还是更喜欢兼具娱乐性和知识性的文字，也就是“信息娱乐”（infotainment）。这种理念也暗合了 Linux 和 Unix 的文化。在 Unix 社区及后起的 Linux 社区中，一直都有保持幽默机智的传

统。此感觉也体现在各个方面，从最早贝尔实验室的 Unix 文件到 Linux 开发人员经常光顾的网站。也许，这就是为什么 Linux 开发人员会如此享受他们工作的一个理由。

当然，你也不要被这些 “浮华文字”的外表所蒙蔽。我们探讨的是颇为严肃的事物。遵守这些准则，就能从软件中获得价值；忽视这些准则，就会失去这些价值。恰当运用 Unix 哲学中蕴含的原理有助于你编写出非常成功的软件产品。如果在开发过程中背离这些原则，则往往会让开发者们错过市场的良机。

听过我讲座的学生们提到，一开始他们对这些思想还只是一知半解，不过，你会时不时回想起它们。强大的思想都有这样的魔力。如果你从来都没有接触过 Unix 哲学，那么就请准备好迎接一次有趣的旅程吧！

本书还大致介绍了 *The UNIX Philosophy* 一书。这样做有两个原因。首先，人们很喜欢在互联网上转发里面的准则，使其成了不朽传奇，所以，就算是没买过那本书的人也能从本书中受益。第二个原因就是，我在那本书的最后一段作出了预测：“Unix 必将成为一个风靡全世界的操作系统，只是时间早晚的问题。”而现在，我们见证到了，源自 Unix 系统的 Linux 操作系统正在通过每天每台机器中的每个应用实现着这个预言。

The Unix Philosophy 简介

操作系统是一个生机勃勃的软件实体。它就像是计算机的灵魂和神经系统，赋予每一个电子器件和硅片元件生动的个性。它给计算机带来了生命。

究其本质而言，操作系统体现着创造者的哲学思想。苹果的 Mac / OS 系统通过非常直观和高度面向对象的用户界面来宣称：“它就在那里，在你的面前。”作为个人电脑革命中无可争议的领军产品，微软的 MS-DOS 系统试图带给桌面用户一种“主流系统的味道”。此外，美国老牌电脑公司 DEC 的 OpenVMS 系统则认为用户畏惧这种有电子思维的机器，所以只提供为数不多的几个强大选项帮助用户完成某个任务。

在起步阶段，Unix 操作系统的创作者就采取了一种颇为激进的理念：他们假定自己的软件用户从一开始就能够熟练操作计算机。于是，整个 Unix 哲学都围绕着“用户知道自己在做什么”的思想而展开。其他操作系统的设计者总是煞费苦心地去迎合各层次用户，从新手到专家；而 Unix 的设计者则采取了比较冷淡的态度：“如果你不能理解，那你就不属于这个世界”。

正是因为这样的态度，Unix 在初期无法得到广泛接受。它只局限于学术研究界的小角落里，因为它给象牙塔里的人们带来了一种令人陶醉的神秘气息。（“是的，Unix 文件系统的层次和万物的自然规律仿佛有共通之处，处处体现着造物者的神奇。”）此外，“技术偏执狂”也喜欢随意摆弄 Unix，比起以往的任何系统，Unix 给出的自由空间更多。

令人扼腕的是，商业世界却未能看到它的价值所在。Unix 渐渐成为黑客的玩具，处处透着神秘。几乎没有盈利性企业敢去冒着投资风险，采用一个出自研究实验室，由大学资助而且只能靠购买方自行作技术支持的操作系统。因此，在其诞生后的 15 年内，Unix 一直如世外高手般默默无闻。

然后在 20 世纪 80 年代初期，惊天逆转开始了。人们纷纷开始流传一个传说：比起人们现在使用的任何操作系统，有一个操作系统更加灵活、更易移植，而且还拥有更强大的性能。此外，它还能以非常低廉的成本运行在任何机器上。

关于 Unix 的这些消息，听起来似乎有点儿好得令人难以置信，但历史总是能够证明一切。不管何时，当某个彻底改变大家世界观的激进想法出现时，我们会本能地去怪罪那些带来新思想的信使。计算机主流世界的任何人都可以看到，这些“Unix 的狂热分子”讨论的并非技术改良，而是技术革命。

正当 Unix 开始渗透到计算机世界时，很显然许多企业官僚都厌恶这种革命性的思想。他们更喜欢由个人电脑和大型机构成的井然有序世界。他们坚持认为，掌握那些他们绞尽脑汁才学得会的简单命令来应付日常工作安稳地保住自己的饭碗就够了。Unix 成为洪水猛兽，并不是因为它具有邪恶本质，而是因为它威胁到了现状。

多年以来，Unix 先驱都过着默默无闻的生活。找不到人来支持他们的激进观点。即使一些有同情心的人士愿意聆听 Unix 倡导者的长篇大论，他们通常的反应也只是“Unix 是不错，但如果想真正做些工作，你应该使用____。”（请在空白处填写你最喜爱的主流操作系统的名字。）但是，操作系统的哲学有点儿类似于信仰。当一个人自以为找到真理时，他是不会轻易放弃的。于是，Unix 信徒们只能继续勇往直前，顽强地坚守着他们的信仰，并相信总有一天世界会改变，软件的理想国终将来临。

商业界忙于建造阻碍 Unix 发展的壁垒，而学术界却张开双臂欢迎它的到来。那些伴随着彩电、微波炉、视频游戏成长起来的年轻一代步入大学校园，而这些大学已经可以几乎不费分文地获得到 Unix 的发行版光盘。这些年轻人纯洁无暇的内心犹如洁白画布，教授们更愿意在这样的白纸上绘制计算机世界非主流的蓝图。

接下来的故事尽人皆知。

如今，就算是在过去不可想象的领域，Unix 也已获得了人们的广泛认可。在学术界，它更无可争议地成为首选系统，其在军事和商业领域的应用也一天天地扩大。

多年以来，我不断告诉人们，Unix 迟早会成为风靡全世界的操作系统。我的预言并没有失手。然而，很讽刺的是，这一款操作系统却不能再叫做“Unix”了。因为有些企业早已意识到这个名字蕴含的商业价值，并委托他们的律师抢先一步将其注册为自己的商标。因此，未来人们设计出的界面、提出的标准，以及诸多的应用只能被命名为“开放式系统”。不过大家可以放心，很显然，Unix 哲学还将会是它们背后的驱动力。

目　　录

第 1 章　Unix 哲学：集思广益的智慧……1
1.1　NIH 综合征……2
1.2　Unix 的开发……2
1.3　Linux：一个人加上一百万人的智慧……4
1.4　Unix 哲学概述……5
第 2 章　人类的一小步……9
2.1　准则 1：小即是美……10
2.2　简化软件工程……12
2.2.1　小程序易于理解……12
2.2.2　小程序易于维护……13
2.2.3　小程序消耗的系统资源较少……14
2.2.4　小程序容易与其他工具相结合……15
2.3　关于“昆虫”的研究……16
2.4　准则 2：让每一个程序只做好一件事……16
第 3 章　快速建立原型的乐趣和好处……19
3.1　知识与学习曲线……19
3.1.1　事实上，每个人有自己的学习曲线……20
3.1.2　大师们也知道，变化不可避免……21
3.1.3　为什么软件会被称为“软件”……21
3.2　准则 3：尽快建立原型……23
3.2.1　原型的建立是学习的过程……24
3.2.2　建立早期原型能够降低风险……24
3.3　人类创造的三个系统……25
3.4　人类的“第一个系统”……26
3.4.1　在背水一战的情况下，人类创建了“第一个系统”……26
3.4.2　没有足够的时间将事情做好……26
3.4.3　“第一个系统”是单枪匹马或是一小群人开发的……26
3.4.4　“第一个系统”是一个“精简、其貌不扬的计算机器”……27
3.4.5　“第一个系统”的概念可以激发他人的创造力……28
3.5　人类的“第二个系统”……29
3.5.1　“专家”使用“第一个系统”验证过的想法来创建“第二个系统”……29
3.5.2　“第二个系统”是由委员会设计的……30

3.5.3 “第二个系统”臃肿而缓慢……31
3.5.4 “第二个系统”被大张旗鼓地誉为伟大的成就……32
3.6 人类的“第三个系统”……32
3.6.1 “第三个系统”由那些为“第二个系统”所累的人们创建……32
3.6.2 “第三个系统”通常会改变“第二个系统”的名称……33
3.6.3 最初的概念保持不变并显而易见……33
3.6.4 “第三个系统”结合了“第一个系统”和“第二个系统”的最佳特性……34
3.6.5 “第三个系统”的设计者有充裕的时间将任务做好……34
3.7 Linux 既是“第三个系统”，又是“第二个系统”……34
3.8 建立“第三个系统”……35
第 4 章 可移植性的优先权……39
4.1 准则 4：舍高效率而取可移植性……40
4.1.1 下一……的硬件将会跑得更快……41
4.1.2 不要花太多时间去优化程序……42
4.1.3 最高效的方法通常不可移植……43
4.1.4 可移植的软件还减少了用户培训的需求……45
4.1.5 好程序永不会消失，而会被移植到新平台……45
4.2 准则 5：采用纯文本文件来存储数据……48
4.2.1 文本是通用的可转换格式……49
4.2.2 文本文件易于阅读和编辑……49
4.2.3 文本数据文件简化了 Unix 文本工具的使用……49
4.2.4 可移植性的提高克服了速度的不足……51
4.2.5 速度欠佳的缺点会被明年的机器克服……52
第 5 章 软件的杠杆效应……55
5.1 准则 6：充分利用软件的杠杆效应……57
5.1.1 良好的程序员编写优秀代码，优秀的程序员借用优秀代码……57
5.1.2 避免 NIH 综合征……58
5.1.3 允许他人使用你的代码来发挥软件杠杆效应……61
5.1.4 将一切自动化……62
5.2 准则 7：使用 shell 脚本来提高杠杆效应和可移植性……64
5.2.1 shell 脚本可以带来无与伦比的杠杆效应……65
5.2.2 shell 脚本还可以充分发挥时间的杠杆效应……67
5.2.3 shell 脚本的可移植性比 C 程序更高……68
5.2.4 抵制采用 C 语言来重写 shell 脚本的愿望……69
第 6 章 交互式程序的高风险……72
6. 1 准则 8：避免强制性的用户界面……74

6.1.1 CUI 假定用户是人类……76
6.1.2 CUI 命令解析器的规模庞大且难以编写……77
6.1.3 CUI 偏好“大即是美”的做法……78
6.1.4 拥有 CUI 的程序难以与其他项目相结合……79
6.1.5 CUI 没有良好的扩展性……80
6.1.6 最重要的是，CUI 无法利用软件的杠杆效应……80
6.1.7 “CUI 有什么关系？人们都不愿意打字了。”……81
6.2 准则 9：让每一个程序都成为过滤器……82
6.2.1 自有计算技术以来，人们编写的每一个程序都是过滤器……82
6.2.2 程序不创建数据，只有人类才会创建数据……83
6.2.3 计算机将数据从一种形式转换成另一种……84
6.3 Linux 环境：将程序用作过滤器……84
第 7 章 更多 Unix 哲学：十条小准则……88
7.1 允许用户定制环境……89
7.2 尽量使操作系统内核小而轻量化……90
7.3 使用小写字母并尽量简短……91
7.4 保护树木……93
7.5 沉默是金……94
7.6 并行思考……95
7.7 各部分之和大于整体……97
7.8 寻求 90%的解决方案……99
7.9 更坏就是更好……100
7.10 层次化思考……102
第 8 章 让 Unix 只做好一件事……105
第 9 章 Unix 和其他操作系统的哲学……113
9.1 雅达利家用电脑：人体工程的艺术……114
9.2 MS-DOS：七千多万用户的选择不会错……117
9.3 VMS 系统：Unix 的对立面……119
第 10 章 拨开层层迷雾：Linux 与 Windows 的比较……123
10.1 内容为王，傻瓜……126
10.1.1 视觉内容：“用自己的眼睛去看。”……128
10.1.2 有声内容：“听得到吗？”……129
10.1.3 文字内容：“视频可以终结广播明星，却消灭不了小报。”……131
第 11 章 大教堂？多怪异……143
第 12 章 Unix 的美丽新世界……153

第 1 章

1 Unix哲学：集思广益的智慧

这个世纪的哲学会成为下一个世纪的常识。

——中国幸运饼干[4]

许多人都将发明 Unix 操作系统的殊荣授予 AT&T 公司的 Ken Thompson，从某种意义上来说，他们是对的。1969 年在新泽西州美利山 AT＆T 公司的贝尔实验室，Thompson 编写出了 Unix 的第一个版本。它作为 Space Travel 程序的平台运行在 Digital PDP-7 小型机上。此前，Space Travel 程序运行在由麻省理工学院开发的 Multics 系统上。

Unix 的开发基于 Multics 系统，后者属于最早的一批分时操作系统。在 Multics 开发之前，大多数计算机操作系统都运行在批处理模式下，这迫使程序员们要去编辑大堆的打孔卡片或纸带。在那些日子里，编程是一个耗时费力的过程。当时有一句流行语是：“上帝帮帮那些打翻了打孔卡片盒的傻瓜吧。”干过卡片机编程的人都懂。

Thompson 借鉴了 Multics 的许多特性，并将它们融入到早期的 Unix 版本，其中最主要的特点就是分时处理。如果没有这种特性，那些在当前 Unix 系统或是其他操作系统上被人们视作理所当然的大部分功能，就会失去它们真正的力量。

Thompson 的开发工作从借鉴 Multics 的想法入手，对于 Unix 开发人员而言，这样的套路可谓是驾轻就熟：良好的程序员写出优秀的软件，优秀的程序员“窃取”优秀的软件。当然，我们并没有暗示 Thompson 是一个小偷。但正是他这种在某些方面避免 NIH（Not Invented Here，非我发明）综合征的意愿和基于别人的成果添加颇具创造性价值的做法，大力推动了这一款或许是历史上最精巧操作系统的出台。我们还将在后面探讨“窃取”软件的意义。现在只需要记住，将一个想法与人共享就如同一个大脑里有了两个想法。

1.1 NIH 综合征

软件开发人员经常会受到 NIH 综合征的影响。在查看别人编写的软件解决方案时，他认为自己完全可以做得更好。也许他真的能更为痛快利落地完成这项工作，但他并不知道别的开发人员当时面临的限制条件。他们可能迫于时间

[4] 在美国的中餐馆吃完饭后，服务生一般会送上一小碟幸运饼干（Fortune Cookie），外表金黄，呈菱角状，脆脆的没有什么特殊的味道。这个饼干可以用手掰开，里面有一张纸条，写着一些关于运势的签语。——编者注

或预算的压力，于是，只能集中精力处理这个解决方案中的某些特定部分。

NIH 综合征的特点就是人们会为了证明自己能够提供更加卓越的解决方案而放弃其他开发人员已经完成的工作。这种狂妄自大的行径说明此人并无兴趣去维护他人竭尽全力提供的最佳工作成果，也不想以此为基础去挑战新的高度。这不仅是个自私自利的做法，还浪费了大量宝贵时间，而这些时间其实完全可以用来提供其他解决方案。更糟的是，新的解决方案有时候只是稍作了一些改进，或根本没有本质区别，从而使得这个问题变得更糟。

偶尔，新的解决方案确实会更好，这只是因为开发人员早已了解前人做过的工作，因此他们可以“取其精华，去其糟粕”，改善此解决方案。这是对前人工作的加强和延伸，并不是 NIH 综合征。这种借鉴其他开发者的概念是 Linux 世界中一种常见做法，当然前提条件是每个人都能得到源代码。事实上，在原有软件的基础上进行加强和扩展也是 Unix 哲学的核心概念之一。

1.2 Unix 的开发

人们为 Unix 的可移植性感到惊叹，但一开始它并没有这个特性。Thompson 最早是使用汇编语言编写 Unix 的。1972 年，他采用了 B 语言这种具有可移植性的编程语言改写了代码。很明显，他可能是想利用不断涌现的新硬件的优势。1973 年另一位 AT&T 公司贝尔实验室的成员 Dennis Ritchie 对 B 语言进行了扩展和调整，将其发展为今天全世界程序员为之爱恨交加的 C 语言。

同样，Thompson 提供了一个后来广为 Unix 开发人员仿效的范例：背水一战的人们常常能够编写出伟大的程序。在必须编写某应用程序的时候，如果满足条件（1）它必须满足实际的需要；（2）周围并不存在任何了解该如何编写此程序的“专家”；（3）没有足够时间 “完美”完成任务，那么，写出优秀软件作品的概率就比较高。就好比 Thompson 的情况，他需要使用一种可移植的语言来编写这个操作系统，因为他必须将他的程序从一个硬件架构迁移到另外的架构；他找不到任何所谓的可移植性操作系统专家；他当然也没有足够的时间去把事情做到“完美”。

不过，Ken Thompson 一人对 Unix 哲学整体发展的推动作用毕竟是有限的。虽然他在文件系统结构、管道、I/O 子系统和可移植性等领域的设计方面做出了杰出贡献，然而 Unix 哲学的成型还是众人努力的结果。初期与 Unix 打交道的每一个人都将各自的专业领域知识应用在 Unix 上，帮助塑造了这一哲学理念。下面列出了一些贡献者和他们的主要贡献（参见表 1-1）。

表 1-1 Unix 哲学的主要贡献者及其重大贡献

贡 献 者	所作的贡献
Alfred Aho	文本扫描、分析、排序
Eric Allman	电子邮件
Kenneth Arnold	屏幕更新
Stephen Bourne	Bourne shell 命令解释器
Lorinda Cherry	交互式计算器
Steven Feldman	计算机辅助软件工程
Stephen Johnson	编译器设计工具
William Joy	文本编辑，类似 C 的命令语言
Brian Kernighan	正则表达式、编程规范、计算机排版、计算机辅助指令

Michael Lesk	高级文本格式、拨号网络
John Mashey	命令解释器
Robert Morris	桌面计算器
D. A. Nowitz	拨号网络
Joseph Ossanna	文本格式化语言
Dennis Ritchie	C 语言
Larry Wall	Patch 工具、Perl 语言、*rn* 网络新闻阅读器
Peter Weinberger	文本检索

上述人士只是 Unix 这一独特系统中最早和最知名的参与者，实际上最终有数以千计的人参与了 Unix 的开发工作。几乎每一篇以 Unix 主要组件为主题的论文，都列举了一大批贡献者。正是这些贡献者共同塑造了 Unix 哲学，让它逐渐被人们理解接受并流传至今。

1.3 Linux：一个人加上一百万人的智慧

如果说 Ken Thompson 是 Unix 的创造者，那么 Linus Torvalds 就是 Linux 操作系统的发明人，当时他还是芬兰赫尔辛基大学的一名学生。1991 年 8 月 25 日他发出了那篇现在广为人知的新闻组主题文章，这篇以“嗨，大家好……我正在编写一个（免费）的操作系统”开头的文章对他的命运产生了深远影响。

Thompson 和 Torvalds 两人至少有一点相似之处，那就是对事物的好奇之心。我们可以找到证据，Thompson 编写 Space Travel 程序只是为了好玩而已。而 Torvalds 在痴迷于类 Unix 操作系统——Minix 的同时，也完全是因为非常感兴趣才会将流行的 Unix 命令解释器 bash 进行改编并运行在他的“玩具”操作系统上。同时，这些在一开始只是“为了好玩”的举动，却最终对软件产业产生了深远影响。

一开始，Linux 也不是一款具备可移植性的操作系统。Torvalds 无意将它移植到英特尔 386 之外的其他架构之上。从某种意义上说，他也只是背水一战，因为他的手头只有少量计算机硬件可供选择。因此，最初他并没有采取任何进一步的举措而只是将自己拥有的资源发挥到极致。但是他发现良好的设计原则和扎实的开发模式还是引领着他去把 Linux 变成一个可移植的系统。从那一刻开始，别的人接过了这个接力棒，很快便将 Linux 移植到了其他架构。

在 Torvalds 的 Linux 出现之前，借鉴他人编写的软件已成为相当普遍的做法。事实上也就是因为这样，Richard M. Stallman 才会在具有里程碑意义的 GNU 公共授权协议（GPL）下正式确立了这一思想。GPL 是一个适用于软件的法律协议，基本保证了软件的源代码可以自由提供给任何想要得到它的人。Torvalds 最终为 Linux 采用了 GPL 协议，这个举动免除了所有人对于相关法律与版权纠纷的后顾之忧，让他们可以自由借用 Linux 的源代码[5]。由于 Torvalds 将 Linux 免费开放出来，因此其他人自然也会将他们的软件免费提供出来以共同发展 Linux。

从一开始，Linux 已经表现出它确实是一个与与 Unix 非常相似的操作系统。

[5] Torvalds借鉴的东西远不止是GPL。Stallman和其他人都曾经指出Linux实际上只是一个操作系统内核而已。绝大部分处于内核外围的程序都来源于自由软件基金会里的GNU项目。追随者理所当然地将这整套系统叫做“GNU/Linux”，在这套系统中，Linux内核是最为关键的一个组成部分。荣誉应当属于有功人士，我们也意识到大多数人已经作出了自己的选择，将这套系统称为Linux。

它的开发人员全盘接受了 Unix 的哲学原理，然后再从头编写了这个新的操作系统。问题是在 Linux 的世界里，几乎再没有其他程序是重新编写的。一切应用都是建立在其他人写好的代码和概念之上。因此很自然地，Linux 成为了 Unix 系统演变的下一步，或许更准确地说，它是 Unix 的一个大飞跃。

类似于 Unix，在 Linux 技术发展的早期，有许多开发者参与其中并提供了帮助。不同的是，Unix 开发者数量最多的时候也就几千人，而今天 Linux 的开发者数量却早已达到了几百万之多。这才是登峰造极的 Unix！正是这种大规模的开发格局，保证了 Unix 的后代 Linux 将在很长时间内都是一款具备强大竞争力的系统。

Linux 为 Unix 世界重新激起波澜，所谓的“开源”要比“专有”软件或是那些没有现成源代码的软件优越。多年以来，Unix 开发人员一直坚信这一点。但计算机行业的其他人却被一些专有软件公司的大量宣传所蒙蔽，他们误认为任何借来的或是免费的软件在性能上都无法比拟那些要付费（有时甚至是耗费巨资）的软件。

在市场营销方面，Linux 社区也更为精明，他们知道只要市场工作做得好，就算是劣质软件也可以成功销售出数百万份。当然，这并不是说 Linux 是伪劣产品。只是，有别于它的前身 Unix 社区，Linux 社区认识到，即使是世界上最好的软件，也只有当人们对它产生了解并认识到它的真正价值时，才会为人所用。

我们将在后续章节再深入探讨这些主题。现在，让我们把 Linux 和 Unix 的历史留在过去，继续前行。事情会更加有趣。

1.4 Unix 哲学概述

Unix 哲学的几条准则看似简单。事实上，它们简单到会容易使人们忽略其重要性。这就是它们颇具欺骗性的地方。其实，简单的外表下掩盖着一个事实：如果人们能够始终如一地贯彻它们，这些准则可是非常行之有效的。

以下这份清单会让你对 Unix 哲学的准则有初步的认识。本书其余部分则会帮助你理解它们的重要性。

(1) 小即是美。相对于同类庞然大物，小巧的事物有着无可比拟的巨大优势。其中一点就是它们能够以独特有效的方式结合其他小事物，而且这种方式往往是最初的设计者没能预见到的。

(2) 让每一个程序只做好一件事情。通过集中精力应对单一任务，程序可以减少很多冗余代码，从而避免过高的资源开销、不必要的复杂性和缺乏灵活性。

(3) 尽快建立原型。大多数人都认同“建立原型”（prototyping）是任何项目的一个重要组成部分。在其他方法论中，建立原型只是设计阶段中一个不太重要的组成部分，然而，在 Unix 环境下它却是达成完美设计的主要工具。

(4) 舍高效率而取可移植性。当 Unix 作为第一个可移植系统而开创先河时，它曾经掀起过轩然大波。今天，可移植性早被视作现代软件设计中一个理所当

然的特性，这更加充分说明这条 Unix 准则早就在 Unix 之外的系统中获得了广泛认可。

(5) 使用纯文本文件来存储数据。舍高效率而取可移植性强调了可移植代码的重要性。其实可移植性数据的重要性绝不亚于可移植代码。在关于可移植性的准则中，人们往往容易忽视可移植性数据。

(6) 充分利用软件的杠杆效应。很多程序员对可重用代码模块的重要性只有一些肤浅的认识。代码重用能帮助人们充分利用软件的杠杆效应。一些 Unix 的开发人员正是遵循这个强大的理念，在相对较短的时间内编写出了大量应用程序。

(7) 使用 shell 脚本来提高杠杆效应和可移植性。shell 脚本在软件设计中可谓是一把双刃剑，它可以加强软件的可重用性和可移植性。无论什么时候，只要有可能，编写 shell 脚本来替代 C 语言程序都不失为一个良好的选择。

(8) 避免强制性的用户界面。Unix 开发人员非常了解，有一些命令用户界面为什么会被称为是“强制性的”（captive）用户界面。这些命令在运行的时候会阻止用户去运行其他命令，这样用户就会成为这些系统的囚徒。在图形用户界面中，这样的界面被称为“模态”（modal）。

(9) 让每一个程序都成为过滤器。所有软件程序共有的最基本特性就是，它们只修改而从不创造数据。因此，基于软件的过滤器本质，人们就应该把它们编写成执行过滤器任务的程序。

以上列出了 Unix 开发人员所奉行的信条。在其他一些 Unix 书籍中你也会找到类似清单，因为这些都是大家公认的 Unix 基本理念。如果你也采用这些准则，那么人们就会认为你是一个“Unix 人”。

下面还列出了 10 条次要准则，这些准则正在渐渐发展成 Unix 世界信仰体系的一个组成部分[6]。并非每个与 Unix 打交道的人都会将它们奉为信条，而且在严格意义上其中一些并不能算作是 Unix 的特性。不过，它们看起来依然是 Unix 文化（当然也包括 Linux 文化）不可或缺的一部分。

(1) 允许用户定制环境。Unix 用户喜欢掌控系统环境，并且是整个环境。很多 Unix 应用程序绝对不会一刀切地使用交互风格，而是将选择的权利留给用户。它的基本思想就是，程序应该只提供解决问题的机制，而不是为解决问题的方法限定标准。让用户去探索属于自己的通往计算机的佳境之路吧。

(2) 尽量使操作系统内核小而轻巧。尽管对新功能的追求永无止境，Unix 开发人员还是喜欢让操作系统最核心部分保持最小的规模。当然，他们并不总是能做到这一点，但这是他们的目标。

(3) 使用小写字母，并尽量保持简短。使用小写字母是 Unix 环境中的传统，尽管这么做的理由已不复存在，但人们还是保留了这个传统。今天，许多 Unix 用户之所以要使用小写的命令和神秘的名字，不再是因为有其限制条件，而是他们就喜欢这么做。

[6] 考虑到我给出的这些词语，比如“信条”、“准则”和“信仰体系”，人们或许会思考，Unix哲学除了介绍技术外，是否还描述了一种文化现象。

(4) 保护树木。Unix 用户普遍不太赞成使用纸质文档。而是在线存储所有文字档案。此外，使用功能强大的在线工具来处理文件是非常环保的做法。

(5) 沉默是金。在需要提供出错信息的时候，Unix 命令是出了名地喜欢保持沉默。虽然很多经验丰富的 Unix 用户认为这是可取的做法，可其他操作系统的用户却并不认同此观点。

(6) 并行思考。大多数任务都能分解成更小的子任务。这些子任务可以并行运行，因而，在完成一项大任务的时间内，可以完成更多子任务。今天已涌现出大量对称多处理（symmetric multiprocessing ，SMP）设计，这说明计算机行业正在朝着并行处理的方向发展。

(7) 各部分之和大于整体。小程序集合而成的大型应用程序比单个的大程序更灵活，也更为实用，本条准则正是源于此想法。两种解决方案可能具备同样的功能，可集合小程序的方法更具有前瞻性。

(8) 寻找 90%的解决方案。百分百地完成任何事情都是很困难的。完成 90%的目标会更有效率，并且更节省成本。 Unix 开发人员总是在寻找能够满足目标用户 90%要求的解决方案，剩下的 10%则任由其自生自灭。

(9) 更坏就是更好。Unix 爱好者认为具有“最小公分母”的系统是最容易存活的系统。比起高品质而昂贵的系统，那些便宜但有效的系统更容易得到普及。于是，PC 兼容机的世界从 Unix 世界借鉴了此想法，并取得巨大成功。这其中的关键字是包容。如果某一事物的包容性强到足以涵盖几乎所有事物，那它就比那些“独家”系统要好很多。

(10) 层次化思考。Unix 用户和开发人员都喜欢分层次来组织事物。例如，Unix 目录结构是最早将树结构应用于文件系统的架构之一。Unix 的层次思考已扩展到其他领域，如网络服务命名器、窗口管理、面向对象开发。

看完这份准则清单，你可能会觉得所有内容都有点儿小题大做。“小即是美”不是什么大不了的道理。“只做好一件事”本身听起来也相当狭隘。“舍高效率而取可移植性”也并不是那种能够改变世界的真知灼见。

这就是 Unix 所蕴含的道理吗？Linux 难道也只是给目光短浅的人准备的小型操作系统？

也许我们应该提及，大众汽车公司曾经围绕着“小即是美”的概念开展了一次成功的汽车营销活动，并借此销售了数以百万计的汽车；或者想一想主流 Unix 供应商 Sun 公司的事例，它的商业战略基于“集中所有资源推出最好的拳头产品”这一思想，或者换句话说，也就是“只做好一件事”的理念。那么，人们对于掌上电脑、无线网络访问和手持视频的兴趣是否与可移植性有关呢？

让我们现在就开始这个精彩的旅程吧。

第 2 章

人类的一小步

2

大约 30 年前，当美国人边开着大型轿车边享受着其他国家民众的艳羡目光时，大众汽车却在美国开展了一项主题为“小即是美”的广告营销活动。那时，这家德国汽车制造厂商的举动似乎有些不合时宜。要知道虽然大众的甲壳虫汽车在欧洲获得了巨大成功，可是让它们奔驰在美国的广袤大地上看起来可有点儿滑稽，就像是巨人土地上的侏儒。尽管如此，大众汽车却一直不遗余力地向美国家庭传递小型车很适合他们的理念。

然后，令人意想不到的事情发生了：中东的石油部长们向世界表明，是的，他们完全某些事务上完全有自主权——也就是，可以大幅度抬高每桶石油的价格。他们通过大量囤积原油让市场供需的天平倾向供应商的一侧。每加仑[7]汽油的价格猛涨到了 1 美元以上。全世界那些被中东扼住能源动脉的地区不得已修建了很多天燃气管道。

长期以来，美国人对大轿车的迷恋世人皆知，然而他们也开始意识到，的确“小即是美”。为了缓解日益羞涩的经济状况，他们开始向汽车制造商订购小型汽车，大众汽车公司成了他们的救星。这款昔日里的“可笑小车”变成了时髦的必需品。

随着时间的推移，人们发现小型车的确拥有一些大型车无法比拟的优势。除了能够降低油耗外，小型车的操纵方式也备受大家喜爱——它们就像是英国生产的跑车，而不是长了轮子的远洋客轮。小型车还能轻而易举地挤进狭小的停车位。它结构简单也因此更易于保养维护。

就在美国人日益热衷小型汽车的同时，AT&T 公司贝尔实验室位于新泽西州的研究小组也发现，小型软件程序也拥有某些优势。他们觉得小程序就如同小型汽车一样，更易于操控，有着更强的适应性，维护起来也比大程序方便得多。这给我们带来了 Unix 哲学的第一条准则。

2.1 准则 1：小即是美

如果你准备开始编写一个程序，请从小规模开始并尽量保持。无论是设计简单的过滤器、图形软件包还是庞大的数据库，你应该尽自己所能将它的规模降至最小，实用即可。请抵制诱惑避免让它成为庞然大物。软件开发应该力求简单。

[7] 1加仑≈3.785升。——编者注

传统程序员的心中经常怀有一个编写出伟大程序的隐秘渴望。在着手准备开发项目时，他们仿佛想要花费数周、数月甚至数年的时间去开发一个能够解决世界上全部问题的程序。这种做法不仅商业代价高昂，而且也脱离现实。其实在现实世界中，只要把一些小巧的解决方案组合起来，几乎不存在解决不了的问题。之所以会选择实施如此大规模的解决方案，根本原因在于我们并没有完全理解该问题。

科幻小说作家 Theodore Sturgeon 曾写道：“90%的科幻小说是垃圾。但其实所有事物中的 90%都一无是处。”这同样适用于最传统的软件。任何程序的大部分代码实际上都没在执行它所宣称的功能。

你对此感到怀疑？让我们来看一个例子。假设你想编写一个将 A 文件复制到 B 文件的程序。以下就是一个典型的文件复制程序可能要执行的步骤。

(1) 要求用户输入源文件的名称。
(2) 检查源文件是否存在。
(3) 如果源文件不存在，提示用户。
(4) 询问用户目标文件的名称。
(5) 检查目标文件是否存在。
(6) 如果目标文件存在的话，询问用户是否需要替换该文件。
(7) 打开源文件。
(8) 如果源文件内容为空，提示用户。必要的话可以退出此程序。
(9) 打开目标文件。
(10) 将源文件的数据复制到目标文件里。
(11) 关闭源文件。
(12) 关闭目标文件。

请注意，文件只在步骤(10)中才真正被复制。其他操作步骤虽然也很有必要，却与复制文件这个目标并无直接关联。如果仔细研究，你会发现这些其他步骤也可以应用于除文件复制之外的众多任务。这里只是恰好也要用，但并非是任务的真正组成部分。

优秀的 Unix 程序应该只提供类似于步骤(10)的功能，别无其他。进一步来说，一个严格遵循 Unix 哲学的程序会在调用时就已获取了有效的源文件名和目标文件名。它应该只负责复制数据。显然，如果程序所做的事情只是复制数据，那的确只会是很小的程序。

我们仍然有疑问，那程序中有效的源文件名和目标文件名来自何方呢？答案很简单：从其他小程序那里获取。这些小程序将执行不同的功能：获取文件名，检查文件是否存在，或确定其中内容是否为空。

“等一下！”你可能会想。我们是不是在说 Unix 里面存在着只检查文件是否存在的程序？没错。标准的 Unix 发行版本包括几百个小命令和小型应用程序，它们自身只能完成一些简单功能。，类似 test 命令这种执行平常小任务的程序很多，比如只确定文件的可读性，等等。这听起来似乎无足轻重，但要知道 `test` 命令可是 Unix 上使用得最频繁命令之一[8]。

[8] 一些Linux/Unix的shell功能（命令解释器）比如bash就嵌入了test命令，人们不再需要调用新（接下页）

就自身而言，小程序做的事情并会太多。它们经常只实现一两个功能。但要是将它们结合在一起，你就能体会到它们真正的力量。它们的整体功能大于各个局部功能的简单相加。大型复杂的任务就此迎刃而解。你只需要在命令行上输入这些小命令，就可以编写出一个新的应用程序。

2.2 简化软件工程

人们常常说，Unix 给程序员提供了世界上最丰富的系统环境。原因之一就是，那些在其他操作系统上很难执行的任务在 Unix 上却能轻松实现。是小程序让这些任务变得简单了吗？是的，这毋庸置疑！

❑ 2.2.1 小程序易于理解

小程序只会集中精力完成一项功能，因此将小程序集合在一起的“全业务”解决方案能将失误减到最少。它们只包含为数不多的几个算法，绝大部分都与要完成的工作直接相关。

另一方面，大程序相对更复杂，也给人们带来了理解障碍。程序的规模越大，就会越发背离作者的初衷。代码的行数也会多得令人难以忍受。比如，程序员可能会忘记某个子程序到底在哪些文件中，或是难以交叉引用变量并记住它们的用法。调试代码也会成为噩梦。

当然，总会有一些让人难以理解的程序，不管它们的规模是大是小，这是因为本身它们执行的功能就晦涩难懂。但这样的程序并不多见。一般来讲，中等水平的程序员都能毫不费力地理解大多数小程序。因此，相比大程序，这可以算作是小程序的一个明显优势。

现在，你可能想知道小程序什么时候会变成一个大程序。答案就是，视情况而定。此环境下的大程序在彼环境中也许只是个中等规模的程序；而且，同样的代码在这个程序员眼里算是大程序，而在别的程序员眼中可能只是小菜一碟。下面列举了一些迹象，如果你编写的软件有出现这些迹象，就表明该软件已经偏离了 Unix 的基本操作思路。

- ❑ 传递给函数调用的参数数量过多，导致代码超出了屏幕的宽度。
- ❑ 子程序代码的长度超过了整个屏幕或是一张A4纸的长度。注意，使用较小的字体且在窗口较大的工作站显示器上，你可以有空间扩展一下代码量。只是请不要得意忘形。
- ❑ 要靠阅读代码注释，你才能记住子程序到底在做些什么。
- ❑ 在获取目录列表的时候，屏幕显示不下这些源文件的名称。
- ❑ 你发现某个文件已经变得很难控制，无法定义程序的全局变量。
- ❑ 在开发某程序的过程中，你已经无法记起一个给定的错误信息是在什么条件下引发的。
- ❑ 你会发现自己不得不在纸上打印出源代码才能更好地组织思路。

（接上页）进程来执行这个命令，从而减少了系统资源的开销。不过这样做的缺点就是，如果你一直往shell中添加命令，它的规模会不断扩大，直到运行非shell命令的代价因Linux/Unix所采取的创建新进程方式而变得非常高昂。所以，最好的做法就是将经常使用的命令驻留在内核的高速缓存中，这样就不需要从磁盘中获取这些命令，从而避免巨大的时间开销。

这些警示可能会激怒一些程序员，也就是身处“大就是好”阵营的那一拨人。他们会争辩不是每个程序都能成为小程序。我们的这个世界如此巨大，总有一些需要使用大型机去解决的超大问题。所以，我们需要编写宏大的程序来解决它们。

这是一个很大的误区。

往往有这样一类软件工程师，为自己能编写出规模宏大的程序而深感骄傲，可除了他自己，没有任何人能读懂这些程序。他会认为只有这样才具有“职业保障”。可以这么说，就只剩他编写的应用程序要大过其自负情结。在传统软件工程环境中，这样的软件工程师可以说是屡见不鲜。

通过这种手段来确保饭碗会带来如下问题：这些人效力的公司一定会意识到，他们最终还是会跳槽到别的地方而将烂摊子留给公司。明智的公司知道采取一些措施来防止这种情况的发生。他们会聘请那些真正懂得易于维护的软件才更具价值的人。

软件设计师再也不能说：“上帝保佑接手的那个可怜虫。”优秀的设计师必须不遗余力地追求软件的可维护性。他们会给自己的代码加上完整而不冗余的注释。他们会尽量编写短小精悍的子程序。他们大幅削减代码，只留下绝对必要的那些部分。经过如此努力产生的作品一定是易于维护的小程序。

❑ 2.2.2 小程序易于维护

小程序通常更容易理解，也就更易于维护，因为理解程序是软件维护的第一步。毫无疑问，你肯定听说过此观点，但是有许多程序员会忽视软件维护问题。他们认为，既然自己费时费力写好了程序，那么别人也同样会愿意花点儿时间去维护它。

大多数软件工程师不会满足于靠维护他人软件为生。他们认为（也许真是）只有编写新软件才能真正挣到钱，而不是靠修复旧有软件的 bug。不幸的是，用户却不这么想。他们希望软件能够稳定运行。如果软件无法做到这点，用户就会对程序供应商产生极大不满。要知道，那些无法好好维护软件的公司可经营不了太久。

软件维护不是一份很吸引人的工作，因此，程序员一直在想办法让这个任务变得更容易，甚至干脆避开它。然而，大部分人都无法推卸这份责任。幸运的话，他们也许能够将这份烦人的工作交给新手来做。但更多时候，软件维护这一职责还是会落在原作者身上，他必须尽其所能让这个任务变得更为轻松愉快。小程序则完美地满足了这一需求。

❑ 2.2.3 小程序消耗的系统资源较少

因为它们的可执行镜像只占用了少量内存，操作系统就更能轻而易举地为它们分配空间。这大大降低了内存交换和分页的需求，往往能够显著提高系统性能。“轻量级”是一个在 Unix 世界颇为流行的术语（比如，我们一般都认为小程序就是一个轻量级的进程）。

大型程序则有着庞大的二进制镜像文件（binary image），操作系统要花费巨大代价来装载它们。频繁发生的分页和交换动作，进而严重影响性能。为了

解决大型程序的资源需求，操作系统设计者试图通过创建动态加载库和共享运行库等功能来对此作出改善。不过，这些都只治标不治本。

在职业生涯的初期，我碰到过一位计算机硬件工程师，他经常会开玩笑地说：“所有的程序员都想要更多的核心资源！我们只能再多加一条内存，才能堵住他们的嘴。”其实，他这个玩笑还是有一定道理。给程序员更多内存，程序就会运行得更快一些，编写程序的时间也就更短，这样会大大地提高生产效率。

让我们来反思一下这个“更多核心资源”的解决方案。在说上述那段话时，我的硬件工程师朋友无意中透露了他的判断依据，即他周围的程序员偏爱编写大型、复杂的程序。这并不奇怪。那会儿我们用来开发应用程序的操作系统平台并不是 Unix。

如果我们一直持续不断地使用 Unix，并全面接受这种小程序理念，就不会提出更多内存需求。那这个笑话就应该是这样的：“所有的程序员都想要更多的 MIPS！”（MIPS 的意思是每分钟执行上百万条指令，这种衡量 CPU 性能的方法很流行，却不一定准确。）

为什么在今天的计算机世界里，MIPS 度量法会成为一个热点呢？因为 Unix 应用得越来越普遍，小程序的使用激增。尽管小程序在执行时只占用系统小部分内存，这些额外的 CPU 马力却给它们带来了最大化的利益。人们只需要将它们加载到内存中，小程序就能迅速地完成其工作，然后释放掉它们占用的内存供其他小程序去使用。显然，如果 CPU 的能力不足，那么每个程序就必须在内存中驻留得久一些，然后才能加载别的小程序来完成其他工作。

内存越大，使用小程序的系统就更能从中受益。大容量内存使得更多小程序可以在内核高速缓存中驻留更长的时间，以减少对二级存储的依赖。小程序也更容易放置在高速缓存中。正如我们将在后面看到的，同时运行的小程序数量越多，整体系统的性能就越高。这也构成了 Unix 哲学的另外一个基本原理，我们将在以后讨论。

❑ 2.2.4 小程序容易与其他工具相结合

任何与复杂大型程序打过交道的人都知道这一点。庞大的单一程序仿佛容纳着整个世界。这些大型程序试图替代一个个单一程序来提供人们需要的每一个功能，而这恰恰有碍自己去配合其他的应用程序一起工作。

人们也许会有疑问，能够提供许多种数据格式转换功能的大应用程序难道不比那种只能转换一种数据格式的小程序更有价值？表面上，这听起来像是一个合理的假设。只要这个应用程序恰恰支持你所需要的转换格式，你就会感觉良好。可是，在需要让该应用程序转换某种它并不支持的数据格式时，那该怎么办？

编写大型程序的软件开发人员通常都有一种错觉，误以为他们能够处理各种意外事件（即他们的程序能处理今天存在的任何数据格式）。这是个问题。虽然开发人员能够处理当前的数据格式，但他们并不了解未来会出现什么使他们的程序变得过时的新格式。那些骄傲自大的、编写大型复杂程序的人们不但喜欢预测未来，并且还认为未来和现在不会有什么大不同。

另一方面，那些编写小程序的人则会保持低调，尽量避免去预测未来的状况。他们对明天作出的唯一假设就是它必将和今天不同。未来，新的接口会出现，数据格式也将发展。程序的互动方式也随着大家口味的变化而不同。新硬件技术出来，旧算法就会显得过时。变化不可避免。

总之，与编写大型程序相比，你会发现自己更乐于编写小程序。其简单性也使得它们的编写、理解和维护更加容易。此外，人与计算机都会发现它们更具有包容性。当然最重要的就是，在编写这些程序的同时，你就可以想出方案以应对那些无法预见的情况。

向前看。未来可能会来得比你想象的快。

2.3 关于“昆虫”的研究

让我们花一点时间来探讨一下昆虫（bug）吧——不是软件 bug，而是现实世界里的昆虫。

在尼罗河源头维多利亚湖的周围，栖息着一种特别的飞虫。已故探险家 Jacques Cousteau 曾经拍摄到令人叹为观止的录像，画面中这种飞虫聚集在湖泊上方和附近的丛林里，像一片黑压压的浓雾。这种昆虫大小和外观很像蚊子，有时候它们密集在湖泊上，人们会误以为那是水上龙卷风或是小型飓风。

鸟类经常成群结队地突袭这些“昆虫群”，以享受这大自然恩赐的饕餮大餐。鸟儿们单次空袭所吞噬的昆虫数量高达几百万只。尽管遭遇到这种大规模的捕食攻击，可还是会有很多昆虫得以幸存，从其在空中形成的巨大阴影来看，它们几乎没有消减的迹象。

Cousteau 对这些特别昆虫的生命周期做了深入密切的观察。他发现，昆虫的成年阶段极其短暂——大约只有 6~12 个小时。即使它们能够幸免于难，没有成为那些鸟儿的午餐，其生命也很短暂，只不过能在阳光下振翅的几个小时而已。

那么，这些成年生活只有短短一天时间的昆虫在做些什么呢？繁衍后代。它将这件我们人类需要花上数年的事情压缩在几个小时内完成。很显然它的做法很成功，因为这个物种并没有灭绝，其数量之多令人肃然起敬。如果说种群的生存延续是它们的目标，那么这些长着翅膀的微不足道的小生命的确很了解自己的优先任务。

这些飞虫的一生只做一件事情，而且它们完成得很好。 Unix 的开发人员认为，软件也应该这样做。

2.4 准则 2：让每一个程序只做好一件事

最好的程序应该像 Cousteau 所拍摄的湖泊飞虫一样，全部能量只用来执行单一任务，并且将它完成得很好。程序被加载到内存中，行使完它的功能，然后退出，让下一个目标单一的程序开始运行。这听起来很简单，可你会发现，太多软件开发人员无法坚持朝着这个简单方向努力。

软件工程师经常会不由自主地成为“功能控”（creeping featurism，这是业

内惯常的说法。一开始，程序员也许只想编写一个简单的应用程序，可随后他的创新精神会慢慢占据上风，促使他在这里添加一个功能，在那里加入一个选项。很快他的程序就成为一个不折不扣的大杂烩，对原有规划而言，大多数功能也无法创造更多价值。也许这些发明中会有一点儿可取之处。（当然我们并不是要扼杀人们的创造力。）但是，开发者必须考虑这些功能是否归属于这片代码。下面一组问题可以为你的决断提供良好依据。

- 这个程序需要与用户互动吗？用户是否要在文件或是命令行中给出必要的参数？
- 这个程序需要输入特殊格式的数据吗？系统上是否有能提供该格式转换功能的程序？
- 这个程序需要输出特殊格式的数据吗？明文的ASCII文本是否就足够？
- 有没有其他程序可以执行你想要完成的类似功能，而不需要重新编写代码？

前 3 个问题的答案通常都是否定的。真正需要直接与用户交互的应用程序比较罕见。大多数程序都能良好运行，完全不需要在它们的常规活动中加入对话解析器。同样，只要使用标准的输入和输出数据格式，大多数程序就可以满足大多数需要。在那些需要特殊文件格式的情况下，我们完全可以使用其他通用程序来作数据转换。否则，我们需要为每一个新程序另起炉灶。

Unix 的 `ls` 命令是绝佳范例，可以充分说明 Unix 应用程序是怎样误入歧途的。截至到目前，它已经拥有了二十多个选项，而且一直在增加。伴随着每个新版 Unix 的出现，它都会有更多选项。我们暂且不理会那些稀奇古怪的功能，先来看看一项更基本的功能，也就是它格式化输出结果的方式。最纯粹的 `ls` 命令应该这样列出一个目录中的文件名（未经排序）：

```
/home/gancarz -> ls
txt
doc
scripts
bin
calendar
X11
database
people
mail
slides
projects
personal
bitmaps
src
memos
```

然而，大多数版本的 `ls` 命令都是按照如下格式输出结果：

```
mail        calendar    people      slides
X11         database    personal    src
bin         doc         projects    txt
bitmaps     memos       scripts
```

一开始，将文件名显示在整齐的列中似乎是种明智做法。但现在 `ls` 的代码里面还包括了列格式化的功能，这个任务与列出文件目录内容已经没有太大关系。列的格式化可以很简单，也可以很复杂，取决于它的使用环境。例如，默认状态下，`ls` 假定用户正在使用 80 个字符宽的旧式字符终端。那如果在一个 132 字符宽的终端屏幕的窗口系统上调用 `ls` 命令，会是什么样子？假设用户喜欢将输出结果分两列显示，而不是四列？假设在终端使用可变宽的字符集呢？假设用户希望在每显示完五行文件名之后有一条实线作为分割呢？这样的情况不胜枚举。

平心而论，`ls` 命令还是保留了逐行显示文件夹下各个文件名的能力。其实这就是它应该做的全部工作，我们完全可以把列格式化的工作交给其他的命令。这样，`ls` 命令就会小得多（也就会更易于理解、易于维护、占用较少的系统资源等）。

编写这种“只做好一件事”的应用最终会产生一个较小的程序，因此这两项原则是相辅相成的。小程序往往只具有单一功能，而单一功能的程序往往也会很小。

这种做法的潜在好处就是，你可以集中精力去解决当前的任务，全心全意做好本职工作。如果无法让程序只做好一件事，那么你很可能并不理解自己正在试图解决的问题。接下来的一章中，我们将讨论如何以 Unix 的方式去理解问题。现在，让我们着眼于小处，只做好一件事。

如果尼罗河上的湖泊飞虫都能做到这一点，那对我们来说又会有多难呢？

第 3 章

3 快速建立原型的乐趣和好处

拉玛人喜欢三五成群地去做一切事情。

——选自阿瑟·克拉克的 *Rendezvous with Rama*[9]

3.1 知识与学习曲线

漫步在华尔街，你很快就会发现那些“菜鸟”投资者根本不知道自己在做些什么。要在股市中挣到钱，大家都知道必须“低买高卖”。然而，年复一年，“豺狼们”还是从可怜的“羊羔们”那里搜刮着数以百万计的美元。小羊羔们（指的就是你和我）经常在市场关键的转折点踏错节拍，这些可都是有据可查的。

其实机构投资者的表现也不见得就好多少。大多数养老基金经理、共同基金投资组合经理和专业理财人士在市场中的表现同样是一年不如一年，尽管其中很多人的年薪都已超过了百万美元。

研究表明，指数基金的长期表现要比 77%的共同基金好，指数基金是指以某一指数的成分股为投资对象的基金，如标准普尔 500 指数（Standard & Poor’s 500）。尽管世界金融市场在大肆炒作投资热点机会，已公布的记录却阐明了一个严峻的现实：哪怕我们在报纸上的股票版面随机购买一些股票，大多数人的表现也会可圈可点，不逊于任何其他投资者，不管他们是业余股民还是专业人士。

虽然在股票市场一直立于不败之地很难，可有些人还是做到了。比如，富达麦哲伦基金（Fidelity Magellan Fund）著名的前任经理彼得·林奇（Peter Lynch），他在 20 世纪 80 年代累积创造了令人惊讶的交易记录，让他的客户都变得非常富有；“股神”沃伦·巴菲特（Warren Buffett）有着“奥马哈先知”的美誉，也为伯克希尔·哈撒韦公司（Berkshire Hathaway）的股东获取了巨额利润；约翰·邓普顿爵士（Sir John Templeton）则在世界各地寻求投资的绝佳机会，同样谋得了巨大财富。

尽管他们都很成功，可这些传奇式的投资者也坦言，他们并不总是无往不

[9] 该书中文版名为《与拉玛相会》，已由四川少年儿童出版社于1998年出版。——编者注

利。他们会讲述自己失败的投资经历，在他们购买了那些自己感到信心十足的公司的股票后，一年内这些股票的市值却狂跌一半。他们会发出感慨：“该买的没买!”而那些不太有希望的股票反倒暴涨了 10 倍或是更高。虽然他们的整体表现仍然远超普通投资者，但他们知道，自己还是有很多东西要学习。

其他专业人士也需要掌握很多新知识。医生必须努力跟上医学研究方面的最新发展。会计人员需要学习税法的新变化。律师得研究新的法院判决案例。精算师、销售人员、卡车司机、装配工、水管工、时装设计师、法官、研究人员、建筑工人和工程师都需要不断地去学习新知识。

❑ 3.1.1 事实上，每个人有自己的学习曲线

想一想吧。你最近一次碰到一个能准确无误了解每次行动结果的天才是在什么时候？我并不是说天赋异禀的人不存在。我只是想表明，天才是非常罕见的。杰出的才能通常需要付出艰苦的努力和学习，再加上一点点好运气。

工程师可能是最佳范例，用来证实大多数人还在学习这一事实。例如，如果航空工程师了解航空领域的所有知识，那为什么他们还是需要试飞员？为什么通用汽车公司的工程师需要对他们的车进行“道路测试”？为什么电脑工程师必须要把他们的产品带到现场作测试，然后才投入大规模的生产？如果工程师们对他们的工作有十分把握，那就不需要质量保证部门，因为开发阶段就应该能保证产品的高质量。

如果这样的话，Unix 工程师们就可以使用 cat[10]命令来编写程序了。

软件工程师尤其需要高强度持续不断地学习，他们有着一条“陡峭”的学习曲线。软件很难一蹴而就。软件工程这项工作包含着不断的修订过程，在这个行当中，试验和错误都屡见不鲜，应用程序的最终成型来之不易，它建立在令人沮丧的无数个小时的返工之上。

请注意，我们并不是说人们永远无法真正地精通某个行当类。只是，它所需要的时间比大多数人想象的要久得多。人类的平均学习曲线进一步延伸下去会比第一次出现的时候更加“陡峭”。当今世界存在着如此多的变数，想真正掌握某些知识有时候需要穷极人的一生，而且人们不可能学会所有的事物。

❑ 3.1.2 大师们也知道，变化不可避免

几乎没有什么项目会一直遵守原有的规格说明书，而不发生任何变化。营销需求会改变，供应商不能提供产品，关键部件的表现可能会与原设想完全不同。因此，建立原型并做测试能够暴露设计上的缺陷。正是这些因素使得复杂技术产品的开发工作成为一项最需要小心处理的任务。

发生变化是不可避免的，这通常要归咎于人们沟通交流的失败。当产品的最终用户试图解释其需求时，他的叙述与实际情况会有出入。他可能忽略了一些事情，或是没能精确表达出他脑海里那些想法的细节。因此，工程师只能用想象力来填补这些空白。尽管他会仔细研读需求文档，但是在设计过程中难免

[10] 所有Unix命令中最简单的一个，cat命令会把用户键入的所有内容都导入到一个文件。有别于文本编辑器或文字处理器，cat命令不能修改已经输入的文本。

还是会带上自己的主观偏见。有时候工程师试图去猜测最终用户的想法或假定“他是想这样做，而不是那样做”。更糟糕的是，工程师与最终用户之间往往还隔着一些人，比如产品经理、销售团队、支持人员，这些人会进一步曲解最终用户的期望。

人的知识总是有限的。除了雾里看花之外，现实经验也决定了我们不会随随便便成功。大家应该坦然面对这个现实。在设计过程中，我们就要考虑到未来必将发生变化。这样，对目标了解得越充分，就越能降低对产品做重大修改的成本。

❑ 3.1.3 为什么软件会被称为“软件”

软件工程比任何其他工程学科都更需要返工，因为软件涉及到抽象概念。如果准确描述硬件都会有困难的话，那可以想象一下人们形容那些只存在于脑海里的想法或芯片中电流的传导模式该是多么困难。我一下就想到了一句格言：“入此门者，莫存希望”（Abandon all hope, all ye who enter here）。

如果最终用户可以详细阐明其想要的功能，如果软件工程师能够完全了解用户现在和未来的需求，那我们可能就不需要软件了。每个编写出来的程序可以第一时间就被烧录到只读存储器（ROM）里。遗憾的是，这种完美的世界并不存在。

在早期做软件设计师的日子里，我常常会追求软件的完美。我反复修改子程序，直到它们能够最快速地运行，代码也足够干净整洁；我还重复审阅程序，希望能找出哪怕一丁点的改进之处；在想到任何新点子的时候，我都会去添加新的功能（是的，我就是传说中的“功能控”）。我尽情发挥，直到老板将我拽回到现实。

“该发布软件了。”他宣布。

“可是我还没有干完呢！再多给我几天时间吧……”

“永远都没有做完的软件，只有发布的软件。”

在软件发布之后，没有人真正知道会发生什么。经验丰富的工程师可能会有一些模糊概念，大概知道软件面对的客户群，它会被应用于什么样的环境之下，等等。但他们很难预料这款软件产品最终的命运到底是成功还是失败。

我一度在一家大型电脑公司担任电话技术支持工程师。在干过多年操作系统的设计工作之后，我转换成了一个与以往截然不同的角色，开始为我们编写的程序作客户支持。这段工作经历确实让我眼界大开。从事软件设计工作时，你其实很容易掉入一个怪圈，认为一切尽在自己的掌握，误以为自己很了解人们会如何使用你的软件。

你错了。也许你确实对此信心十足。但是，安然待在办公室工作间的工程师，身边围绕的同事与自己一样同样拥有高学历，他能意识得到下列诸多情况吗？

- ❑ 某个在交易日导致系统反复崩溃的内核bug可能会给一家华尔街企业造成每

小时100多万美元经济损失。一个星期之后，该公司在心烦意乱的情况下选择安装原系统竞争对手的产品。可再过了一个月，他们还是重新切换成最早的系统，因为尽管它有一些缺陷，但在这两个系统之间，还是原来的系统显得更为可靠。

- 一家石油公司可能会使用某个系统来对墨西哥湾进行地震波扫描以寻找天然的石油储备。他们每天都会获取到千兆字节数据，这需要在系统集群上安装90多块大硬盘。人们必须能够随时获取所有数据。而且出于安全考量，没有任何一名地质学家能够持有访问所有关键数据的密钥，这是因为公司很担心会有人带着在哪开采石油的机密信息跑掉，这将会给公司造成数十亿美元的损失。
- 一家每天在数据库中存储超过1000万条记录的电信公司想设计一个新系统，让它每天可以插入1亿条记录。每秒处理1000次数据库的插入动作本身就已经够困难的了，可现在，他们不得不在高峰期处理每秒高达5000次的数据插入工作。
- 某家电力公司的系统是美国最大一个州的电网中央控制器。一旦该系统出现故障，就可能导致百万人断电。

Unix 的开发人员当然不知道 Unix 上会发生什么事情。MS-DOS、OpenVMS 和一切其他操作系统的开发人员也一样。每一个新的操作系统（Linux 也不例外）都会将它们的设计者带到未知领域。他们唯一的希望就是不断收集系统使用报告，然后纠正过程中的相应偏差。

例如，Ken Thompson 在编写第一个 Unix 内核时，有没有注意到可移植性的重要性呢？显然没有，因为他使用的是汇编语言。后来，他采取了一门高级语言对它进行了改写，也改变了原来的方向。那么， Dennis Ritchie 是否预料到了 C 语言将成为一门通用编程语言，让数百万的程序员为之爱恨交加呢？也没有。他那本经典著作 *The C Programming Language* 的修订版包括了对该语言规范的修订，这更加让我们确信他在第一次创作的时候并没有将它完全折腾好。

因此，我们大多数人都还在学习。即使我们自负到认为自己知晓一切，也总会有人改变对我们的期望。那我们到底该如何开发软件呢？接下来的这条原则是关键。

3.2 准则 3：尽快建立原型

“尽快”就是越快越好，火速进行。你可以先花少量时间规划这个应用程序，然后便可以创建原型。开始编写代码吧，就好像你的生命完全取决于这个原型一样。记住要趁热打铁，我们根本就没有时间来浪费！

这个想法与许多人认为的“适当工程方法”（proper engineering methodology）背道而驰。大部分人都曾被要求，在进入编程阶段之前应该充分考虑好自己的设计。他们说：“你必须撰写一份功能规格书，并定期对此进行复核，以确保自己没有脱离正确的轨道。撰写一份设计规范可以理清自己的思路。在你开启编译器之前，应该确保 90%的设计已经完成。”

大体上，这些原则听起来都不错，但它们都没有给予原型足够的重视。只有建立原型，你的构想才能首先通过可视化、现实可行的方法得到验证。在此

之前，它们只不过是你脑海中一些零零碎碎的想法。那时候，你对这些概念几乎没什么认识，同时，别人也不可能像你一样去理解问题。你需要大家在项目进行之前达成共识。通过原型来提供一个关于目标的具体化表象，会得到客户认同。

❑ 3.2.1 原型的建立是学习的过程

越早开始建立原型，就离可发布产品的状态越近。原型可以显示哪些想法可行，最重要的是，哪些是不可行的。你需要这种对于你所选开发思路的肯定或否定。在早期发现设想有误，只会令你遭受小小的挫折，这要远远好于到后期才发现问题。比如，大家花了好几个月召开产品开发会议，却完全没发现潜伏在其中的巨大设计缺陷，而等到产品发布日期前 3 个星期问题才暴露出来时，会让你措手不及的。

❑ 3.2.2 建立早期原型能够降低风险

有了一个具体原型，你就可以指着它说道：“产品将会是这个样子。”如果能把它演示给值得信赖的用户看看，并获得他们的反馈，你就会了解你的设计是否具有针对性了。很多时候，你会收到大量批评，这没什么关系，因为你已经获得了宝贵的反馈信息。让一小群人批评你总好过在产品发布后听到万千用户要求召回软件而汇聚在一起的愤怒呼吁。

每一个正确设计的背后都有着数百个错误的设计方案。我们可以在早期剔除掉不良设计方案，形成了优胜劣汰的筛选过程，该过程会引导你越来越接近有质量保证的成品。你可以尽快发觉那些无用的算法、总是丢一个节拍的计时器，以及无法与用户进行交互的界面。这些试验可以帮助你“吹尽狂沙始到金”。

大多数人都同意建立原型有很多好处。即使是那些教授传统软件工程方法的学者们也都很快认识到了它的价值。然而，原型只是用来完成目标的一个手段，而非目标本身。建立原型的主旨应该是创建一个我们称之为“第三个系统”的产品。不过，在讨论该“第三个系统”之前，我们需要谈谈在它之前的两个系统。

3.3 人类创造的三个系统

人类只具备创建三个系统的能力。不管如何努力，无论为之奋斗的时间有多久，人类最终都会意识到想要打破这个规律只是徒劳。人类根本无法建立第四个系统。只有自欺欺人的家伙才不相信这个铁律。

为什么只有三个？这是个难以回答的问题。人们或许能从科学、哲学和宗教的观点出发，得到一些猜测性的理论。每个理论都会对这种情况给出一个貌似合理的解释。但是，最简单的解释可能就是，人类的系统设计过程就像人类自身一样，都必须要经历三个生命阶段：未成年、成年和老年。

在未成年阶段，人们普遍充满活力。他们就像是街区里新来的孩子：浑身活力四射，渴望受到关注，而且显示出无穷潜力。在一个人从未成年阶段发展到成熟阶段的同时，他会成长为对这个世界更加有用的人。他们的职业生涯逐

渐成形，与他人的长期合作关系得到发展，在世俗事务中的影响力越来越大。这个人开始给人留下深刻印象——好的坏的或是其他。待到年老的时候，这个人会丧失掉年少时的许多机理能力。而且，随着体格机能的逐渐衰退，其对世俗的影响也会消退，职业生涯都将成为往昔记忆。人们开始抗拒这种改变。然后，沉淀下来的就是那些基于人生经历的宝贵智慧。

人类设计的系统也经历着同样的发展阶段。每个系统都具备与人类生命发展阶段所对应的那些特性。所有的系统都遵循着这条发展路径：从未成年开始，过渡到成熟阶段，直到以老年作为终结。

正如一些人无法活到天年一样，总有一些系统到达不了成熟期。通常，这是由外部环境所决定的。开发计划也许会改变；项目经费可能会被撤销；潜在客户或改变主意，决定购买别家供应商的产品。任何这些因素都有可能导致系统中途夭折。然而，在正常情况下，人们还是可以引领这些系统经历所有的三个发展阶段。为了阐明这个道理，我们将这些阶段称为人类的三个系统。

大多数 Unix 开发人员并没有听说过人类的三个系统这个说法，但他们的做法却表明这三个系统的确是无处不在的。让我们仔细研究一下它们的一些特性吧。

3.4 人类的“第一个系统”

❑ 3.4.1 在背水一战的情况下，人类创建了“第一个系统”

通常情况下开发人员担负着巨大压力，他们必须要赶在截止日期前交付项目，或是满足其他一些时间紧迫的需求。种种压力会激发出他内心灵感创意的火花。最终，在他深思熟虑良久，并在脑海中反复琢磨他的构想之后，这一点火花变成了一小团火焰。工作仍在继续。他的创意本能开始占据上风。灵感的火焰变得越发明亮。

某一刻，他会意识到一部分构想并不只局限在“达成目标”这一范围内。他感觉自己好像在反复思量一些更重要的事物。此时，如果他认为这些构想可以提供更合理的解决方案，那么他就会逐渐淡忘最早制定的目标。

❑ 3.4.2 没有足够的时间将事情做好

如果有充足的时间，那他就不会受迫于赶进步的压力。在有压力的情况下就只能靠临场发挥了。临场发挥虽然是一种妥协之举，可他却只能毫无妥协地勇往直前——哪怕朝着错误的方向。至少，他的旁观者都是这样认为的。当开发人员时间不够，只能背水一战的时候，他就可能会打破常规去行事。当然，在他那些思想保守的同事看来，他已经完全丧失了理智。

通常他会受到人们的指责。“他不能这样侥幸行事！”他们坚称，“他根本不知道自己在做什么。他错得太离谱了。”那他又是如何回应的呢？“是的，它很丑，但它管用啊！”

时间的匮乏迫使他必须集中精力去处理这个任务中的重要事项，并忽视掉其他细枝末节。因此，他计划将一些细节留给后续版本去实现。可是我们要注意到，他可能永远不会有机会去完成所谓的“后续版本”。但是，正是因

为相信自己在未来会“填补空白”，他才不会脱离正确的轨道，而这个理由也变成了人们为第一版所有缺点作辩护的藉口。

❑ 3.4.3 “第一个系统”是单枪匹马或是一小群人开发的

这其中一个原因要归咎于主流世界里许多人对“第一个系统”开发人员所做的事情不屑一顾。这些人无法立足于他所在的高度去看待事物，所以也就理解不了他的兴奋之情。因此，他们得出结论，他做的工作很有趣，但并未有趣到足以让他们自己也投身其中。

为什么人们会避免参与“第一个系统”的开发工作？第二个原因更为实际：建立“第一个系统”有很大的风险。没有人知道“第一个系统”是否具备能够过渡到“第二个系统”的特性。失败的可能性总是超过 80%。用业界行话来说，与失败的“第一个系统”联系在一起可是颇有职业风险的。因此，一些人宁可等到系统构想被证实可行之后才会开始行动。（后面我将进一步讨论，他们通常会成为“第二个系统”的开发人员。）

随之而来的风险就是，不了解真相的经理很有可能会宣称这“第一个系统”可以作为正式产品，并过早地将它交付给市场部门。这通常会导致销售人员过分炒作这个还不是很完善的系统，而且它的缺陷也会暴露无疑。系统会频繁崩溃，往往还发生在最尴尬的时刻。因此，用户会对这个系统产生强烈偏见，认为它是一款质量很差的产品。最终，这还会严重影响到系统的销售状况。

当一个小组建立“第一个系统”的时候，他们热情高昂，很快就能把事情办好。精诚协作的力量注入到这个团队，最终促成了一个强大有凝聚力的整体。团队成员们拥有共同的信念，心怀热切渴望来促使系统成型。他们了解自己的目标并为之孜孜不倦地工作。在这样一个亢奋的环境里，人们感到精神振奋的同时也会精疲力尽。一旦系统成功，它还能会给人们带来巨大的成就感。

有一点是肯定的：“第一个系统”几乎不太可能是由一大群人完成的。一旦团队规模大到影响到成员之间的日常互动交流时，工作效率就会降低，人与人之间也会产生冲突。成员们开始有着各自隐秘的日程安排。而且，在大家开始追求一己私利的同时，就会形成个人的“一亩三分地”。诸如此类事件总是会淡化系统最终目标，使其难以实现。

❑ 3.4.4 “第一个系统”是一个“精简、其貌不扬的计算机器”

它用最小的成本获得差强人意的性能。它的开发人员在大多数时候不得不采取权宜之计，被迫通过硬编码（hard-wire）方式来编写应用程序的很多代码，以牺牲功能和灵活性为代价来换取简洁和速度。华而不实的修饰性功能则都留给下一个版本。任何与系统目标无关的事物都被排除在外。软件确实能完成它的使命——也就仅此而已。

当人们将常见的成熟系统和这“第一个系统”作比较的时候，他们通常会对后者的高性能啧啧称奇。他们很想知道为什么自己的系统比不上这个“新来的孩子”。用流行标准来测试的话，这个不起眼的新产品在性能上超过了他们最喜爱的产品，人们对此有些扼腕。这似乎有点儿不公平，事实上也是如此。将“第一个系统”与那些已经第 *n* 次发布的系统放在一起比较，其实有点儿“关公战秦琼”，因为各自的设计者有着不同的目标。

“第一个系统”的设计者们高度集中精力以解决目前的问题，让某些东西能工作，甚至也不管是什么功能先运行起来再说。后续的设计者通常要花费大量时间来添加新功能，以满足他们察觉到的市场新动向。就整体而言，这些功能往往会对系统性能产生负面影响。是的，后续系统能完成的任务会更多，但是人们需要为这些新功能付出一定的代价。

❑ 3.4.5 “第一个系统”的概念可以激发他人的创造力

“第一个系统”使人们陷入了天马行空的思考，“如果……就会怎么样？”它让人们的胃口大开：想要更多的特性、更多的功能、更多的一切。人们会发表一些冒进言论，“想一想它的可能性！”或“想象一下，在子虚乌有生物科技公司我们能用这个系统来完成的事情！”

下面列出了一些领域和技术，在这些领域中人们充满了创意非凡的想象。眼下，它们已经派生出了许多“第一个系统”。

- ❑ 人工智能
- ❑ 生物技术
- ❑ 数字成像
- ❑ 数字音乐
- ❑ 电子货币系统和无钞社会
- ❑ 基因工程和克隆技术
- ❑ 因特网和万维网
- ❑ 交互式电视
- ❑ 火星登陆计划
- ❑ 微型机械
- ❑ 纳米技术
- ❑ 质量管理（六西格玛、全面质量管理等）
- ❑ 虚拟现实
- ❑ 无线技术

这些能够激发他人想象力的概念成为了“第二个系统”紧随“第一个系统”产生的主要原因。如果“第一个系统”并没有什么令人兴奋的东西，那人们得出的结论都会是：它的确符合某些人的需要，但是这些人（打着哈欠）表示还有更合适的工具。很多“第一个系统”早已夭折，因为它未能激励其围观者采用它的概念来实现其他伟大而美好的事物。

SourceForge（http://www.sourceforge.net）一个流行的网络资源库，展示的都是一些新型“第一个系统”。其中一些软件曾让人们为它们显示出来的新概念或新技术而感到欢欣鼓舞，并围绕其开展了大量相关活动。遗憾的是，不是所有展现出来的事物都符合“第一个系统”的成功标准，其中一些根本无法带给人们任何启发。因此，SourceForge 也成为了各种各样“第一个系统”的网络墓地。

3.5 人类的“第二个系统”

“第二个系统”是一个怪胎。在人类创建的三个系统中，它最受关注，而且往往会取得商业上的成功。对于那些喜欢规避风险的个人来说，它的确提供

了某种程度上的安全感，而且让人们更容易上手使用。取决于市场的规模，它可能获得数千甚至数百万用户的青睐。然而，颇具讽刺意味的是，在许多方面“第二个系统”其实是三个系统中最差的那个。

❑ 3.5.1 “专家”使用“第一个系统”验证过的想法来创建“第二个系统”

“第一个系统”早期的成功深深地吸引了一些人，他们积极参与进来并希望将自己的名字与这个系统紧密相连，从而获得这样那样的回报。每个人都想和成功产品搭上关系。

这个自封的“专家”群体通常包括许多对“第一个系统”颇有微词的批评家。这些人为自己没能参与“第一个系统”的设计工作而深感懊丧，他们发泄着对“第一个系统”创始人的不满，并声称自己完全可以做得更好。有时候他们是对的。他们确实可能在系统设计某些特定方面做得更好。在重新设计“第一个系统”中几个基本算法的时候，他们的专业知识派上了用场。但请记住：“第一个系统”的设计者（们）并没有足够时间去把事情做好，而这些专家中的许多人一来知道什么是正确的做法，二来也有充足时间和资源来做好它。

另一方面，他们这些“吃不到葡萄”的专家会对别人来之不易的成就大倒酸水。这里面大有一些 NIH 的意味，也就是曾经流行的“非我发明”综合征。虽然其中许多人其实也有能力去建立“第一个系统”，他们只是没能抢得先机而已。他们参与“第二个系统”的开发工作不是为了找乐子，而是希望借此机会用自己的设计机制来取代“第一个系统”的机制，从而“改善原有设计师明显的业余尝试”，为自己获得专业口碑。

这种态度往往会引起“第一个系统”设计者的愤怒。偶尔，他们也会奋力反击。流行 X Window Sytem 系统的先驱 Bob Scheifler 就曾不客气地回应过认为他的早期设计风格过于随意的批评：“如果不喜欢，你完全可以随意编写你自认为具备行业标准的窗口系统。”

❑ 3.5.2 “第二个系统”是由委员会设计的

“第一个系统”是小团队的心血结晶，这个团队的规模通常少于 7 人。然而，为“第二个系统”的设计工作做出贡献的人却有几十、数百，具体到 Linux 更是数以千计。“第一个系统”的成功就像一块磁石，它吸引到很多人，哪怕他们只是对这个发人深省的想法有一丁点兴趣。其中确有一些人是真心希望能进一步拓展早期的思想，但大部分人却只是凑凑热闹而已。

“第二个系统”的设计委员会公开开展其业务。它把会议通告发布在大家都看得到的网络信息库和用户新闻组上，他们还会通过其他知名的信息渠道来公布消息。它会发布设计文档，骄傲地展示所有贡献者的名字。该委员还会试图确保所有参与者都能获得应有的荣誉——有时甚至并不是他们应得的。打个比方，如果委员会是在建立一条人行道，那其成员都会希望在水泥地上镌刻上他们的大名。

尽管该团队（而不是因为）举办了这些无关紧要的活动，一些真正的设计工作还是在进行。委员会的一些成员主动参与了主要设计作品的工作，并能够交付高质量的软件；有些人则作为某些软件关键领域的“看门人”，以确保所有的修改不仅能达成预想中的目标意图，而且经过了精心设计；另外一些人则

扮演着“魔鬼代言人”的角色，促使委员会提供真实的解决方案；还有一些人表示，他们愿意提供一些有趣的议题供大家讨论，由此帮大家理清新的设计思路。

不幸的是，这种由委员会来负责设计工作的做法有一些缺陷。对于小组（一种至少包括两个人的组织机构）而言，就所有要点达成一致意见几乎是不可能的。为了体现自己的价值，每个参与者都想确保自己对总体设计贡献出一些想法，而不管自己是否具备该设计领域的专业知识。其实这些人的对错并不重要。他们只是出于一己之私，希望能有机会对自己说：“与这些专家在一起，我还是能坚持自己的立场，因此，这让我也成为了一名专家。”人们要的只是一头“敏捷羚羊”，可当你把所有人贡献出的想法累加在一起时，却得到了一个“笨重的大象”。

❑ 3.5.3 “第二个系统”臃肿而缓慢

它拥有的特性正好与“第一个系统”相反（即，“第一个系统”是精简的，而这“第二个系统”却像一个步履维艰的巨人）。如果“第一个系统”至少需要 1MB 内存的话，那么“第二个系统”完全无法运行在任何少于 4MB 内存的机器。哪怕是运行在 1 兆赫兹主频机器上的“第一个系统”，其高吞吐量也让人们为之交口称赞，而“第二个系统”的用户却在哀叹，就算机器拥有千兆赫兹主频，它的性能却慢如蜗牛。

“这是因为‘第二个系统’具有更多的功能，”委员会如此辩解，“你的付出是有回报的。”

“第二个系统”的确有着更多特性。它拥有一系列令人印象深刻的功能，这也是它赖以成功的原因之一，但普通用户用得上的功能只是其中的一小部分，其他的反而碍事。“第二个系统”运行得非常缓慢，因为它必须耗费资源来处理大量冗余的“优势”。

通常，我们只能通过购买更多硬件来让它运行得更快一些。我一直相信，计算机制造商对“第二个系统”的热爱正是出于此原因。在可移植软件占据主流的今天，大多数“第二个系统”都能运行在几乎所有供应商的平台上，只要这个平台足够大。然而，想要充分利用新技术的优势，客户往往需要耗费巨资购买新设备。“第二个系统”的庞大规模实际上提升了各种硬件产品的销售量：更快的 CPU、更大的硬盘、更高容量的磁带驱动器以及大量的内存芯片。这可给硬件厂商带来了巨大的利润。

因此，这“第二个系统”带给大家的感受可谓是喜忧参半。你得到了很多功能，有一些也许能用得上。而且你也有机会去说服你的老板，是时候购买新机器了。

❑ 3.5.4 “第二个系统”被大张旗鼓地誉为伟大的成就

“第二个系统”在市场上大展宏图。它的灵活性、多种选项和可扩展性为其在商业上获得了广泛好评。经销商们赞美其优秀品质：今天它就实现了未来的技术。在各个方面它都远超现有的系统。用户可谓是别无他求。

不明真相的群体有点儿不堪炒作重负，他们期待着专家的解答，而专家们也愿意为他们解答问题。设计委员会的任何成员（它现在已经发展到数百人）即刻披上了“专家的外衣”，备受人们尊重。其他人则煞有介事地评判设计委员会所做的工作，由此也获得了一呼百应的公信力。

人们对这个系统的兴趣有增无减。它成为了媒体的宠儿，各大杂志都蜂拥而上跟踪它的进度，揭秘其诀窍的书籍开始出现。人们召开各种会议让那些严肃认真的关注者探讨它的未来，研讨会帮助那些对其一知半解的人们探究它的历史。随着越来越多追随者参与进来，它俨然成为一股势不可挡的潮流。

一旦所有人都在吹捧这 “第二个系统”是一项杰出的成就，人们就会对此深信不疑，它便成为大家脑海中根深蒂固的思想。例如，X Window System 吸收了许多超越其基本窗口功能的特性。尽管其中大部分功能几乎用不上，且它们的存在也极大地影响了系统的整体表现。但 X Window System 还是得以生存了下来，因为它毕竟是“第二个系统”。虽然它有无数缺点，但它还是强大到足以超越 Unix 市场上的其他任何窗口系统。它拥有至高无上的地位。而且，“第二个系统”是其他系统无可替代的——除掉“第三个系统”。

第 4 章

可移植性的优先权 4

1991 年的 USENIX 技术大会，我作为与会者漫步在纳什维尔的 Opryland 酒店大堂里，尽情欣赏着这座豪华的乡村音乐城堡。我穿行在拥挤的人群中，希望能够邂逅某个名人，然后炫耀一下自己刚刚签订了一份出版合同。

那时，我可谓是春风得意。出版社毫不犹豫地接受了我的提议，让我出一本关于 Unix 哲学的著作。（“很多人希望阅读这样的一本书。”）合同谈判的进展非常顺利。（“我希望先拿到 *X* 美元的订金。”“没问题，我们答应你的要求。”）交稿截止日期也很宽松。（“我们会额外多给你两个月，不想让你有太大的压力。”）是的，再没有比这更好的开局了。

说老实话，我却战战兢兢，心乱如麻。我得到了这份梦想的合同，这本书的创作题材我也渴望已久，可最初那股欣喜若狂的兴奋劲儿过去之后，我得直面现实，其实我并没有做好精神准备：见鬼，我真的要开始写书了吗？

我过去的写作经验只不过是为地方性娱乐杂志撰写一些专题文章。我学会了如何串词；我知道要注意主题句、实意动词的运用，以及被动语态的用法；我可以吸引并保持住读者的兴趣。但是，通过撰写杂志文章我成为了“一名不错的短跑选手”。而现在，我却要开始跑“马拉松”。

我第一时间飞奔到一个朋友那里去请求帮助。身为完成了几部大著的作家，他早已经历过类似的涅槃过程。我问他，应该怎么做才能处理好这项看似无法完成的任务？

“先去购买一台笔记本电脑。”他回答道。

看到我脸上疑惑不解的神情，他解释说，写书这件事情很特别，你要么无法下笔，要么思如泉涌。你需要精神高度集中才能将连绵不绝的想法倾注于在纸上。你必须时时刻刻考虑该如何创作这本书：早晨刷牙，开车上下班，会议间歇，吃午饭的时候，在健身俱乐部锻炼的时候，与家人一起看电视的时候，或是临睡之前。笔记本电脑是唯一那种适合随时输入文字的便携式设备，强大到足以让你在任何地方都能创作出一本书。

受益于高速发展的现代微系统技术，笔记本电脑的厚度通常还没有三孔活页夹厚，却能容纳下与桌面 PC 相当的计算能力。它的重量低于 5 磅[11]，紧凑的设计使得它像大学教材般便于携带。笔记本电脑拥有硬盘驱动器和内置

[11] 1磅≈0.45千克。——编者注

的调制解调器，因此能够处理诸如电子表格的制作、文字处理、编程等日常计算任务。

几年前，苹果电脑公司曾经策划过一款电视广告，两名高管在讨论不同款个人电脑的优缺点。两个人都谈到了技术规格，其中一个人说的话饱含深意：最强大的计算机并不是那一款有着最快 CPU、最大磁盘驱动器或是最强劲软件的机器。使用最频繁的那台计算机才最为强大。

如果按这个标准来判断，迟早有一天笔记本电脑会被列为有史以来最强大的电脑。虽然它不具备如实验室超级计算机那般快得令人眩目的处理能力，它也没有最新的磁盘存储容量技术，它的图形功能可能永远赶不上最先进的桌面显示屏。在它身上，你很难看到什么高性能机器的影子，但它有一个足以打动人的优势：便携性。

这带给我们下一条 Unix 哲学的准则[12]。你可能需要特别留意此条准则，因为它充分说明了 Unix 为何能成为软件领域的“常青树”。如果拿人的寿命来作类比，它可以说早就万寿无疆了。

4.1 准则 4：舍高效率而取可移植性

软件开发过程涉及无数选择，每个选择都意味着各种妥协。有时，开发人员必须编写简短的子程序，因为他根本就没时间去编写复杂程序。其他时候，有限的内存可能是个制约因素。而在某些情况下，人们必须避免使用过多的小数据包来挤占网络带宽，因为我们采用的网络协议对大数据块的传输更为有利。程序员总是要在一大堆方案中作出取舍，尽量去满足那些往往自相矛盾的目标。

其中，程序员要面对的一个艰难选择就是：高效率与可移植性。这也是一个极其折磨人的抉择，因为偏向高效率往往会导致代码不可移植，而选择可移植性却又会让软件的性能不那么尽如人意。

高效率软件不会浪费 CPU 周期。它充分利用了底层硬件的特性，可往往完全忽视了可移植性的问题。它利用了一些硬件功能，如图形加速器、高速缓冲存储器以及专门的浮点指令，等等。

虽然从纯粹主义者的立场而言，高效率的软件非常有吸引力。然而，可移植性意味着软件能够运行在许多不同的机器上，这使得人们考量的天平向可移植性这一端倾斜。这其中资金层面的因素要大于技术层面：在今天的计算环境中，那些只能运行在一种体系架构上的软件，其潜在市场竞争力会大打折扣。

当然，重视可移植性并不意味着你一定要开发出低效率、技术上晦涩难懂的软件。相反，从可移植性软件中获取最佳性能需要更高超的技术水平。如果水平不够，还有一个替代方案，那就是等到可用的硬件问世再动手。

❑ 4.1.1 下一……的硬件将会跑得更快

曾经我们在省略号处填入“年”：明年的硬件将会跑得更快。但由于当今硬件技术的迅猛发展，有的时候我们完全可以说，下个季度甚至下个月的硬件

[12] “便携性”与“可移植性”的英文单词同为portability。——编者注

可能会跑得更快。如今，电脑制造商的产品开发周期日益缩短，他们只用花比过去短得多的时间就能推出新的型号。厂商们你追我赶，经常个个都号称自己的产品拥有更高性能和更低价格，借此与竞争对手一较高下。因此，不管你现在正在使用什么样的电脑，很快你就会觉得它又老又笨，就像当年大学实验室里的庞然大物一样。

硬件设计周期日益紧缩的趋势还在加剧。今天的半导体设计人员使用复杂的模拟器来构建升级版芯片。由于模拟器（它们自身就是一些强大的计算机）的运算速度持续提高，因此开发人员也就可以更快完成设计任务。然后，这些新的芯片便成为未来模拟器采用的动力引擎。一代代处理器如滚雪球一般发展。半导体设计世界的动向令人目不暇接，芯片的性能不断螺旋式加速上升。

随着高速机器逐步淘汰原来的慢机器，问题的关键已经不再是软件是否能充分利用硬件优势，而是软件能否运行在更新的机器上。你可能要花上几天或是几周的时间为今天平台上的一个应用调试优化出最佳性能，然而不久却发现，下次硬件升级会使得软件运行速度“自然而然地”快了 10 倍。不过，具备可移植性的软件才能利用下一款超级计算机的优势。

在 Unix 环境中，可移植性的含义通常意味着人们要转而采用 shell 脚本来编写软件。shell 脚本由多个可执行的 Unix 命令构成，它们被放置在单独的文件里，可以间接由 Unix 的命令解释器来执行。因为典型 Unix 发布版本都会拥有一大堆小型而具备单一用途的命令，所以 shell 脚本几乎可以轻而易举地构建所有任务，最底层任务除外。

shell 脚本附带的好处就是，比起 C 语言编写的程序，它们更具备可移植性。你应该只在别无他法的时候才考虑用 C 语言来编写程序，虽然这个想法可能听起来很另类。但在 Unix 环境中，C 程序缺乏 shell 脚本的可移植性。C 程序往往要依赖头文件、计算机体系结构，以及具体 Unix 版本中一系列不可移植的特性。在 Unix 从 16 位架构移植到 32 位和 64 位的同时，大量软件曾因为 C 语言相对较低的可移植性而无法使用。在这个方面，C 语言只比 20 世纪 80 年代的汇编语言强那么一点点。

如果你想让自己的程序能在 Unix 之外的系统上运行，那么可以选择其他的语言，每一门编程语言都有自己独特的优缺点。如果你追求的是严格意义上的可移植性，那么 Perl 和 Java 便能较好地满足这一要求，它们在 Unix 和 Windows 平台上都能用。然而，将一门语言成功地移植到 Unix 之外的平台，并不意味着这个平台就能坚持 Unix 的哲学理念和工作方式。比如说，你会发现 Windows 上的 Java 开发人员往往是隐藏底层细节的互动式开发环境（interactive development environment，IDE）的忠实支持者，而 Unix 平台下的 Java 开发人员却喜欢能帮助自己深入研究图形用户界面内部细节的工具和环境。

❑ 4.1.2 不要花太多时间去优化程序

如果程序运行的速度还算可以，那么就接受事实，承认它已经满足了你的需求。在琢磨着优化子程序和消除关键瓶颈的同时，我们还应该思考如何能在未来的硬件平台上提升性能。不要单纯为了求快而优化软件。请记住，明年的机器很快就要问世了。

很多 Unix 程序员常犯的错误就是，为了获得性能上一些微不足道的优势而采用 C 语言去重写 shell 脚本。这纯粹是浪费时间，我们完全可以把这些时间用在获取用户的建设性回应上。也许有时候 shell 脚本的运行速度真的不够快。不过，就算你确实需要较高的性能，而且坚信只有 C 语言才能达到效果，还是得三思而后行。尽管程序确实会有得满足特殊需求的时候，但通常情况下还是没必要用 C 语言来重写脚本。

请关注“微优化”（micro-optimizations）这个概念。如果（而且这个“如果”是不容忽视的）必须要优化 C 程序的性能，Unix 平台提供了 `prof` 和其他一些工具来定位使用得最频繁的子程序。对这些被调用过成百甚至数千次的子程序进行优化，可以产生事半功倍的效果。

另一种提高程序性能的方法就是研究如何处理数据。例如，我曾写过一个简单的 shell 脚本，可以在不到一秒的时间内搜索分散在几千个文件中的二百多万行代码。我的诀窍就是先建立一个数据索引。每个索引行包括一个单独的词以及所有包含这个词的文件列表。大多数程序使用的字符不会超过 20 000 个，这意味着索引数将不会超过 20 000 行。对于 Unix 工具 `grep` 来说，查找 20 000 行的文本是一个相对简单的任务。一旦 `grep` 定位到这个单独的词条，它便会打印出与该词条相关的文件名列表。它的查找速度非常快，因为它采用的方式是提前查阅好数据，这也使得这个程序具有了可移植性。

❑ 4.1.3 最高效的方法通常不可移植

任何时候，但凡一个程序利用了某些特殊硬件功能的优势，它在变得更高效的同时，却也不那么可移植了。特殊功能有时候会极大改善软件的性能，但它们需要采用与硬件设备相关的代码，当目标硬件被更快的版本取代之后，人们就需要更新代码。虽然更新与特定硬件相关的代码为系统程序员提供了许多“就业保障”，可雇主却得不到什么好处。

在我的职业生涯中，我加入过的一个设计团队要为一款新硬件平台创建一版早期的 X Window System。这期间有位工程师编写了几个演示程序，绕过 X Window System 并应用了该硬件的先进图形功能。另一位工程师也编写了一套类似的演示程序，不过它构建在 X Window System 提供的可移植性接口之上。第一个工程师的演示效果令人赞不绝口，因为他使用了最先进的图形加速器，拥有令人难以置信的高效性能并将硬件特性发挥到极致。相反第二个工程师的演示程序的运行速度就较慢一些，但因为运行于 X Window System 之上，它却扎扎实实地保持了可移植性。

最终，曾经最先进的图形硬件变得不那么先进，而且公司开发出了一款更快但完全不向前兼容的新型图形芯片组。在新硬件上重新实现第一个工程师的演示程序需要付出艰苦卓绝的努力，所有人都没那么多时间。因此，第一个演示程序默默无闻地消失了。而另外一个能够运行在 X Window System 上的演示程序具备可移植性，不需要做什么额外修改工作就被移植到了新系统，在我撰写本文的时候，它可能仍然在使用中。

当你出于效率考量利用到一些特定硬件的功能时，你的软件就成为兜售该硬件的工具，而失去了软件自身的立场。这限制了它作为一个软件产品的能力，

而它的销售价格也实现不了其内在价值。

那些与硬件平台紧密结合的软件只在该硬件平台具有竞争力的时候才能维持它的价值。一旦该平台的优势消失殆尽，软件的价值就会大大地下降。为了保值，它必须从一个平台移植到另一个可用的、更新更快的架构。如果不能迅速采取行动转移到下一个可用的硬件平台，那对软件而言就意味着命运的终结。市场的机会之门在关闭之前，只会短暂地开放。如果软件赶不上这个时间窗，它的市场地位就会被竞争对手取代。有人甚至认为，无法将自己的软件升级到最新平台是软件企业倒闭的首要原因，超过了其他所有原因的总和。

衡量应用程序是否成功的一个标准就是它能够在多少个系统上运行。显然，与一款备受市场瞩目且可以运行在多个供应商系统上的应用相比，依赖于单一供应商硬件平台的程序将难以成为前者的主要竞争对手。从本质上讲，可移植的软件比高效率的软件更有价值。

在转移到一个新平台的时候，充分考虑了可移植性因素的软件大大降低了它的平台转移成本。由于开发人员不需要耗费太多时间在移植工作上，因此他便可以把更多时间用来开发新功能，由此吸引到更多用户，同时也赋予产品新的商业优势。因此，从它诞生的那天开始，可移植的软件就更有价值。人们在早期为可移植性付出的所有努力都会在后期得到丰厚回报。

❑ 4.1.4 可移植的软件还减少了用户培训的需求

在用户花时间学会了该如何使用某个应用程序组件之后，如果未来平台上还运行这款软件组件，他之前的辛苦就会得到回报。该软件的未来版本可能会略有变化，但支撑它的核心理念和用户接口很可能始终如一。在产品每次升级的过程中，用户对该产品的体验也得到了增强，随着时间的推移，他就会由新手用户变成高级用户。

从 Unix 的最早版本开始，这种从新手用户转变为高级用户的过程一直在持续进行。人们在早期版本中学到的大部分知识仍然直接适用于今天的 Linux 平台。诚然，Linux 的确添加了一些自己的新特性，但总的来说，由于整体用户环境（从 shell 到常用工具）都具备可移植性，因此那些经验丰富的 Unix 用户在 Linux 环境中还是能游刃有余。

❑ 4.1.5 好程序永不会消失，而会被移植到新平台

你有没有注意到，有些程序以这样那样的形式一直存在着？人们总是觉得它们特别好用。它们拥有真正的内在价值。总会有人出于好玩或是谋取利润等目的，主动承担起编写或将它们移植到当前流行硬件平台上的使命。

就拿 Emacs 风格的文本编辑器来说。虽然在某些方面，它们是 Unix 体系中的反面典型，但它们一直都是广大程序员和 Unix 爱好者的最爱。不单 Unix，你还能在其他系统中找到它的变种。虽然多年来，有些版本的 Emacs 早已发展为繁琐的庞然大物，可在它的最简版本中，Emacs 还是提供了一个不错的“无模式”手段，用于文本的输入和编辑。

另外一个好例子就是微软的 Office 和类似产品中包含的那些典型商业程序。人们发现，为了在现代商业环境中高效工作，大家总是需要像文字处理器、电

子表格和演示工具这样的桌面程序。

然而，没有任何个人、公司、组织甚至于国家能够将一个不错的主意占为己有。最终，其他人都会注意到这个想法，并且开始制作所谓的“合理复制版本”。聪明的群体能够意识到它的价值，他们会努力在尽可能多的平台上实现这个想法，并尽可能多地抢占市场份额。实现这一目标的最有效方式就是编写可移植的软件。

案例研究：雅达利2600 游戏机

让我们来研究一下雅达利2600游戏机（Atari 2600），或者也可以叫它雅达利视频计算机系统（Atari Video Computer System，VCS）。VCS是史上第一款成功的家庭视频游戏系统，可谓是天时地利之作。它成功地迎合了人们的想象力，当时大家刚刚从本地酒吧和电玩城那里浅尝到Space Invaders（太空入侵者）游戏的滋味，并准备好了将这个视频游戏的新世界带入自己家中。它是第一款盒式一可编程的游戏机，改变了整个游戏行业的格局，并给人们带来了莫大的游戏乐趣,这些曾经只藏在校园实验室和软件专业人士的隐藏目录里的游戏，被引入普通家庭的电视中。如果说程序员有着与生俱来对游戏的向往，那这种热情与美国乃至全世界人群喜好玩游戏而产生的巨大利润相比，可以说是微不足道。

雅达利2600在问世之初就有着颇高的性价比，它的表现不错，大家付出的真金白银物有所值。当时，游戏机的售价约为100美元。它配备了两个游戏操纵杆和一对桨状控制手柄。雅达利还在其中捆绑了一个名为“战斗”的游戏盒带，它包括各种结合了坦克、喷气式飞机以及福克飞机的两人战斗游戏。

雅达利并没有从游戏控制台的销售量中挣到什么钱，庞大的利润来自游戏盒带。这些盒带每盒售价为14美元到35美元不等，它们成为雅达利的主要收入来源，同时，大量小型软件公司也希望能从电子游戏热潮中分得一杯羹。从雅达利的运营角度来看，一旦它收回了开发游戏盒带的投资成本，其余的都是纯利润。

我有个朋友就在一家为雅达利2600生产游戏盒带的软件公司谋得一职。他解释说，将一个象棋游戏或是“射击”游戏挤进只有不到8K ROM的机器上可真不容易。这相当于在大众甲壳虫汽车里塞满20个人，不是每个人都有机会能看到窗外的风景。

在这段为游戏盒带编写代码的职业生涯中，他编写了一些最高效但不具备可移植性的软件。他把指令视为数据，数据当成指令。在执行水平回扫动作时，也就是电视机上的光束在扫完屏幕右边的最后一个光点再返回到屏幕左边的时候，软件还采用了某些特殊操作。为了节省内存空间，他绞尽脑汁用上了每一个可能的捷径。他的代码看上去很美，却是软件维护者的噩梦。

在2600产品发售期内，雅达利公司还推出了一款以6502为基础的800系统，它成为家用电脑产品线的旗舰产品。800系统是第一款真正意义上的计算机，因为它有一个打字机式的键盘，以及二级存储和通信设备的接口。它的售价将近1000美元，对2600主导的市场也没有造成很大的威胁，可这份好运只持续到内存芯片降价之前。

由于其他供应商的竞争和800电脑深受欢迎的图形扩展功能,雅达利迫切需

要推出一款面向大众市场，能够运行800电脑软件的视频游戏机。这款新机器被称为5200,它的出现使得大众市场的电脑盲们也可以运行技术人员在800电脑上玩的同款游戏。

一旦大众市场发现了这款魅力无穷的新机器能够提供更为平滑的图形界面，2600系统便被人们毫不犹豫地抛弃。随后雅达利2600游戏盒带价格立即下跌到谷底。预料到2600寿终正寝命运的经销商们开始降价销售他们所拥有的库存产品，市场上充斥着甩卖的2600盒带。这进一步促使它的价格暴跌，同时也波及了很多软件公司。

对盒带生产商而言，苦难并没有结束。虽然大部分在2600上流行的游戏也成为5200的热门，但在此之前，它们必须经过重新编写才能运行在新的硬件平台上。由于2600盒带上的代码过于高效，几乎没有丝毫的可移植性。这意味着人们需要花费巨大代价对这些软件进行改写。

问题是，虽然这些游戏盒带软件可谓说是有史以来最高效的产品，但在新硬件问世的当天，它的价值却一落千丈，这都是因为它不具备可移植性，无法重新编译并在5200上重用。如果这些代码是可移植的，也许可以改写视频游戏产业的历史。雅达利公司很有可能成为世界上最大的软件供应商。

最后请注意，今天你可能只需要花个几美分便能买到一个雅达利2600的游戏盒带，不过就算你会去买，我估计也只是怀旧的成分居多。与此同时，为了购买微软Office功能最强大的版本，你花的钱不会少于几百美金。有部分原因就是，在英特尔公司发布更强大的8086处理器版本时，微软Office都能从旧平台迁移到新平台。这虽然让Office程序一直保持在技术领域的前沿，可却要付出很高的代价。

然而，Office的开发人员必须时刻保持警惕，密切关注未来Open-Office的动向和其他出现在Linux市场中的Office克隆产品。其中一些产品的内在可移植性要远远高于Office，这可能意味着未来它们能够更加适应商业技术的发展演变。微软也许得耗费巨大财力物力才能维护Office的霸主地位。只要英特尔继续生产向后兼容指令集的CPU芯片，微软的Office组件（从这个意义而言，微软Windows本身也一样）的移植工作就不会太难。但如果有人设计出了一款非常先进的机器，每个人都希望拥有，哪怕它有着与英特尔毫不兼容的结构，那微软就将面临庞大的工作量，需要大费周章将Windows和Office移植到新的架构，否则就只能坐以待毙。

这个故事的寓意是什么？可移植性有着巨大回报。其他关于效率方面的考量不过是小问题。

迄今为止，我们一直在对高效软件和可移植性软件作比较。能够轻松移到新平台的代码远比那些需要利用硬件专属特性的代码更具有优势。我们已经看到，完全可以用实际标准（即金钱）来判定这个原理的重要性。为了保住自己的利润根基，软件公司应该努力实现产品的可移植性，并在这个过程中力争提高效率。

然而，对于可移植性这一目标而言，可移植的代码只完成了一半工作。所有应用程序都由指令和数据构成。指令的可移植性可以确保代码还能运行在明年的机器上。那么数据呢？置之不理吗？当然不行。Unix 程序员不仅得编写

可移植代码，数据也一样。

怎样才能让数据具备可移植性呢？ Unix 哲学的下一条准则提供了一个解决方案。

第 7 章

更多Unix哲学：十条小准则

7

至此，我们已经探讨了那些构成 Unix 哲学核心的准则，它们是 Unix 世界的坚实基石。任何对这些准则怀有强烈异议的人都无法理直气壮地宣称自己是“Unix 人”。即使他们觉得自己就是，也会招来 Unix 社区（Linux 社区也包含其中）的普遍质疑，被认为缺乏对 Unix 及其理念的真正信仰。

在经受过 Unix“宗教教条”式的训导之后，我们准备着手阐述其中的一些理论教义。Unix 和 Linux 开发人员不遗余力地维护着我们所讨论过的准则的完整性。而另一方面，人们对这些准则也许还是会有一些认同感。虽然不是每一位“Unix 人”都完全同意本章谈到的观点，但就整体而言，Unix 社区（如今的 Linux 社区）普遍遵守着这些原则。

你会发现本章谈及的某些重点事项更侧重于我们应该怎么做，而不是为什么要那样做。我会尽量做一些解释，但你要知道有些事情其实本就没道理可言，只是一贯的传统做法而已。就像宗教一样，Unix 也有它自己的传统，而 Linux 则以一种“新瓶装旧酒”的方式表现出同样的特征。

很明显，我们并没有过多地谈及开源，这可是大多数 Linux“意外革命者”奉为圭臬的东西。因为开源精神在 Linux 文化中根深蒂固，早已成为如 Richard M. Stallman 和其他先驱者的战斗口号，所以，只将它作为一个章节中的小插曲来讨论是不够的，稍后我们会对开源做更深入的探究。

其他操作系统的支持者也认同其中一些 Unix 哲学的小准则，这并不奇怪。好的想法往往能传播到整个计算机世界。其他系统的软件开发人员会发现，某些 Unix 概念表面上看起来并不适用，但实际上却很有价值。于是，他们会把这些理念融入到自己的设计中，甚至有的时候这些系统和应用程序都会有带有 Unix 的影子。

7.1 允许用户定制环境

多年以前，我曾为 X Window System 编写过一个叫 `uwm` 的窗口管理器，也是一款用户界面。它给用户提供了很多在今天的窗口系统中被人们视为理所当然的功能：移动窗口、调整大小、改变堆叠顺序等。这个程序得到了广泛好评，后来成为了“第 10 版 X Window System 的标准窗口管理器”。它提出的概念在当时鲜为人知，可今天却成为窗口管理器的概念鼻祖，被广泛应用于 X Window System 之上。现今的窗口管理器大量借鉴了 `uwm` 及其衍生产品中原始理念。

Uwm 赖以成功的原因之一要归结于 Bob Scheifler，也就是他曾在麻省理工学院举行的 X Window System 早期设计会议上发出的一句感慨。Bob 是 X Window System 整体设计的一个杰出贡献者，当时他正在审阅我为“Unix 窗口管理器”撰写的几页设计规范，突然他脱口而出：“假如我不想用鼠标左键来做那个呢!”他接着建议，也许用户可能会喜欢自己定制每个鼠标按钮的初始功能。

如果我是一个漫画人物，你就会看到当时我脑子里的那个灯泡一下子亮了起来。

uwm 的开发沿袭了这个思路，并在定制用户界面领域开辟了一片新天地。X Window System 本来就可以让用户选择自己的窗口管理器。而 uwm 程序又在用户定制上更进一步，允许用户选择窗口管理器的外观、感觉和行为，甚至鼠标的移动、按钮的点击、颜色、字体、菜单选项都可以由用户自行决定。这个概念是如此强大，以至于 X Window System 的开发人员后来还设计了一款特别的资源管理器，它几乎能让用户控制屏幕里每一个元素。这种前所未有的灵活性到今天仍未逢敌手，即便微软和苹果的桌面环境也无法企及。其他系统只将如此灵活定制的权力赋予了开发人员，而 X Window System 却将其提供给了普通用户。

早些时候我们曾经说过，人们在某些事情上投入越多，就越希望获得更多回报。在观察人们如何使用 uwm 的时候我发现，如果有机会去调整使用环境，那人们会很乐意那么做。内置的灵活性使得用户可以更加投入地学习如何使用某个应用程序，从而获得最大收益。随着人们在量身定做的环境中变得如鱼得水的同时，他们就会愈发难以适应那种没有定制功能的环境。

今天，大部分 Linux 环境都遵循着这个基本理念。在刚开始使用 Linux 的时候，人们普遍会觉得这真是个烦人的系统，因为它太灵活了。过多选择反而让大家无所适从。从一开始 Linux 就提供了各种各样的选择：众多发行版、多种窗口管理器、多个桌面、多款文件系统等等。你也用不着只盯着一家厂商购买 Linux 操作系统。事实上，你甚至都不需要浪费钱。任何一家供应商网站都提供免费下载的 Linux 系统。

最终，用户还是找到了适合自己的 Linux 操作方式。他们选好了发行版，并用心学习该如何最大限用地利用系统中的诸多功能。一旦他们倾注在这个系统上的时间和精力达到一定程度，切回到其他操作系统就变得困难无比。他们对 Linux 的热情之大以至于他们宁愿改变自己去适应原本不喜欢的事物，而不是完全弃用之。

有些人对 Linux 颇有微词，抱怨它迫使用户先要投入大量时间和精力先学习，然后才能真正有效地利用好 Linux。他们认为，使用 Linux 很容易“搬起石头砸自己的脚”。这也许没错。但是 Linux 的倡导者 Jon“maddog”Hall 曾下过论断，搬起石头砸自己的脚总比裹足不前要好。

7.2 尽量使操作系统内核小而轻量化

这是 Unix 纯粹主义者的一个热门话题，多年以来也成为大家辩论的主题。Unix 内核包括一些例程，它们基本上只管理内存子系统以及与外围设备相连的

接口，不做其他事情。任何时候只要人们想得到更高效的应用性能，他们第一个建议就是将应用程序放在内核中运行。这样可以减少程序运行时在存储空间上下文切换的次数，付出的代价则是内核规模变大，并且不易兼容其他 Unix 内核。

在 X Window System 的早期开发阶段，人们产生过强烈分歧，讨论将 X 服务器部分例程嵌入到 Unix 内核中是否能产生更高性能。（X 服务器是 X Window System 的一部分，用来从鼠标和键盘事件中捕捉用户输入，并在屏幕上呈现对应的图形对象。）X 服务器运行在用户空间（即内核之外的存储空间），这使得它相对更具有可移植性。

“将应用放在内核”阵营提倡，要充分利用这种减少内核和用户空间上下文切换次数的优势来提高性能。他们的理由是，内核规模的迅速增长不会有什么影响，因为相比早期的系统，现代 Unix 拥有更多可用内存。因此，还有充足空间去满足应用程序运行的需要。

另一方面，“将一切放在用户空间”阵营却认为那样将使 X 服务器变得不可移植。而且，修改 X 服务器的程序员不单要成为合格的图形软件开发人员，还需要变成 Unix 内核高手。可是，成为 Unix 内核高手的代价不菲，开发人员必定得放弃自己在 X 服务器图形领域中的一些兴趣，X 服务器的开发工作便会受到影响。

那么，两大阵营最终是如何解决这个分歧的呢？系统自己作出了选择。在貌似成功地将 X 服务器移入内核之后，测试人员发现 X 服务器的 bug 不仅会使 X Window System 崩溃，甚至还波及整个操作系统。操作系统崩溃比 X 服务器崩溃还要严重，因此今天大多数 X 服务器仍然驻留在用户空间里。“将一切放在用户空间”阵营得了一分。

只要能抵制住将一切放在内核空间的诱惑，保持内核小而轻量化就会容易得多。在启动任务时，通过降低数据复制或修改的次数，小巧轻便的内核还是能够加快任务在用户空间的激活速度。这最终使 Linux 能够更为简单有效地将那些“只做好一件事”的小程序集合在一起。将单一功能的小程序快速激活对 Unix 整体性能起到了至关重要的作用。

7.3 使用小写字母并尽量简短

对 Unix 系统而言，人们首先注意到的一个特点就是它采用的几乎都是小写。Unix 用户不需要使用大写锁定键来进行操作，所有输入采用小写字母即可。

这么做有两个原因。首先，小写字母看起来更轻松。如果一个人必须长时间与文本打交道，那就能明显地感觉到小写文本看起来比大写的舒服很多。第二点也是更重要的一点就是，小写字母有向上伸和往下伸两种形状，也就是从字母主体延伸出来的向上或向下的细线，从字母“t”、“g”、“p”就可见一斑。在你阅读的时候，它们能够给眼睛传递一些智能化提示，让阅读变得更为轻松。

Unix 是区分大小写的。例如，“MYFILE.TXT”和“MyFile.txt”并不是同一个文件。Unix 常用命令和文件名采用的都是小写。大写通常用来吸引人们的注意。例如，在目录中将文件命名为“README”就是一种视觉提示，让用户在进行

别的操作之前先来阅读此文件内容。此外，ls 命令在列出目录文件名的时候，通常都是按字母顺序排列的，这样大写的文件名就排在文件列表的前面。人们就会更多地注意到这个文件。

对于长久以来已适应了其他不区分大小写的操作系统的人们来说，在初次接触 Unix 的时候往往会无所适从，可最终他们还是能够适应，许多人甚至喜欢上这个特性。

Unix 的另一个怪癖就是它的文件名通常都很短。常用命令的长度一般不超过两或三个。Unix 奉行简约至上，你会发现诸如 ls、mv、cp 等这些并不直观的命令。拥有多个长单词的命令往往会使用首字母缩写的形式。例如，“parabolic anaerobic statistical table analyzer”（抛物厌氧统计表分析器）会简略缩写成“pasta”。

使用简短名称的传统由来已久。最早，Unix 是在使用电传打字机（teletype）的系统上开发的，并没有 CRT 终端。在使用电传打字机输入信息的时候，每当人们按下一个键都会伴随有哒哒的提示音，最快的打字员每分钟约能输入 15~20 个单词。因此，采用简短名称来描述事情便成为流行做法。事实上在过去，大多数不会打字的计算机程序员拿着计算机根本就做不了什么事情（好吧，也许还是能做一点点）。

为什么 Linux 用户仍然坚持着这个传统呢？毫无疑问，今天的 PC 键盘让人们大大提高了输入速度，所以简洁性就不再是必要条件。但这么做的原因是，采用较短的名称可以在一个命令行中包含更多命令。你应该还记得 Linux 的 shell 有管道机制，它可以把一个命令的输出作为另一个命令的输入。

这个强大的功能在 Linux 用户中非常盛行。他们经常将很多命令串在同一个命令行里。如果名称过长，他们使用的窗口就会显示不下这行命令。解决方案就是尽量采用简短的命令名，因此当你在屏幕上打开几个相邻窗口的时候，较小的窗口就可以包括更多的命令。

当然，你也许会想：“有这个必要么？”为什么不直接点击鼠标来执行呢？如果你正在反复使用某个命令，这没问题。但要记住，Linux 和 Unix 之所以这么强大是因为它们可以将多个命令动态结合起来去构建新的应用。而今天流行的桌面环境却做不到这一点。WYSIWYG（what you see is what you get，所见即所得）这个词其实也意味着 WYSIATI（what you see is all there is，所见即一切）。点击鼠标能完成的工作毕竟很有限，假如人们非要尝试着去提供一个图形化 shell，那充其量也不过就是一款繁琐僵化的用户界面。就算使用“快捷方式”也只会缩短运行单个命令所花费的时间。[13]

[13] Linux上的各种桌面环境都可以让你采用快捷方式来执行多个命令。但是，如果没有这种动态结合多个命令的能力，它也不过是僵化的技能。未来它是否会继续保持这种僵化模式，还有待观察。谁也不知道在创意非凡的Linux人的头脑中，潜藏着什么奇思妙想。