1

哲学

Philosophy: Philosophy Matters

Those who do not understand Unix are condemned to reinvent it, poorly.

不懂 Unix 的人注定最终还要重复发明一个蹩脚的 Unix。

Usenet 签名, *1987* 年 *11* 月

—Henry Spencer

1.1 文化？什么文化

这是一本讲 Unix 编程的书，然而在这本书里，我们将反复提到“文化”、“艺术”以及“哲学”这些字眼。如果你不是程序员，或者对 Unix 涉水未深，这可能让你感觉很奇怪。但是 Unix 确实有它自己的文化；有独特的编程艺术；有一套影响深远的设计哲学。理解这些传统，会使你写出更好的软件，即使你是在非 Unix 平台上开发。

工程和设计的每个分支都有自己的技术文化。在大多数工程领域中，就一个专业人员的素养组成来说，有些不成文的行业素养具有与标准手册及教科书同等重要的地位（并且随着专业人员经验的日积月累，这些经验常常会比书本更重要）。资深工程师们在工作中会积累大量的隐性知识，他们用类似禅宗“教外别传”[译注[1]]的方式，通过言传身

[1] 教外别传。禅宗用语。不依文字、语言，直悟佛陀所悟之境界，即称为教外别传。又称单传。故禅宗又作别传宗，系教外别传宗之略称。据《祖庭事苑卷五怀禅师前》录载，禅宗传法诸祖亦以三藏教乘接引弟子，至达摩祖师时，始单传心印，破执显宗，即所谓教外别传，不立文字，直指人心，见性成佛。后四句英文为“A special transmission independent of the scriptures. Not founded on words or letters. By directly pointing to the mind. One’s nature is seen, enlightenment is attained”. —译者注

教传授给后辈。

软件工程算是此规则的一个例外：技术变革如此之快，软件环境日新月异，软件技术文化暂如朝露。然而，例外之中也有例外。确有极少数软件技术被证明经久耐用，足以演进为强势的技术文化、有鲜明特色的艺术和世代相传的设计哲学。

Unix 文化便是其一。互联网文化又是其一——或者，这两者在 21 世纪无可争议地合二为一。其实，从 1980 年代早期开始，Unix 和互联网便越来越难以分割，本书也无意强求区分。

1.2 Unix 的生命力

Unix 诞生于 1969 年，此后便一直应用于生产领域。按照计算机工业的标准，那已经是好几个地质纪年前的事了——比 PC 机、工作站、微处理器甚至视频显示终端都要早，与第一块半导体存储器是同一时代的古物。在现今所有分时系统中，也只有 IBM 的 VM/CMS 敢说它比 Unix 资格更老，但是 Unix 机器的服务时间却是 VM/CMS 的几十万倍；事实上，在 Unix 平台上完成的计算量可能比所有其它分时系统加起来的总和还要多。

Unix 比其它任何操作系统都更广泛地应用在各种机型上。从超级计算机到手持计算机到嵌入式网络设备，从工作站到服务器到 PC 机到微型计算机。Unix 所能支持的机器架构和奇特硬件可能比你随便抓取任何其它三种操作系统所能支持的总和还要多。

Unix 应用范围之广简直令人难以置信。没有哪一种操作系统能像 Unix 那样，能同时在作为研究工具、定制技术应用的友好宿主机、商用成品软件平台和互联网技术的重要部分等各个领域都大放异彩。

从 Unix 诞生之日起，各种信誓旦旦的预言就伴随着它，说 Unix 必将衰败，或者被其它操作系统挤出市场。可是在今天，化身为 Linux、BSD、Solaris、MacOS X 以及好几个其它变种的 Unix，却显得前所未有的强大。

> Robert Metcalf[以太网络的发明者]曾说过：如果将来有什么技术来取代以太网，那么这个取代物的名字还会叫“以太网”。因此以太网是永远不会消亡的（注[2]）。Unix 也多次经历了类似的转变。
>
> —Ken Thompson

[2] 事实上，以太网已经两次被不同的技术所取代，只是名字没有变。第一次是双绞线取代了同轴电缆，第二次是千兆以太网的出现。

至少，Unix 的一个核心技术——C 语言——已经在其它系统中植根。事实上，如果没有无处不在的 C 语言这个通用语言，还如何奢谈系统级软件工程。Unix 还引入了如今广泛采用的带目录节点的树形文件名字空间以及用于程序间通信的管道机制。

Unix 的生命力和适应力委实令人称奇。尽管其它技术如蜉蝣般生生灭灭，计算机性能成千倍增长，语言历经嬗变，业界规范多次变革——然而 Unix 依然巍然屹立，仍在运行，仍在创造价值，仍然能赢得这个星球上无数最优秀、最聪明的软件技术人员的忠诚。

性能—时间的指数曲线对软件开发过程所引发的结果，就是每过 18 个月，就有一半的知识会过时。Unix 并不承诺让你免遭此劫，只是让你的知识投资更趋稳定。因为不变的东西有很多：语言、系统调用、工具用法——它们积年不变，甚至可以用上数十载。而在其它操作系统中则无法预判什么东西会持久不变，有时候甚至整个操作系统都会被淘汰。在 Unix 中，持久性知识和短期性知识有着明显的区别，人们在一开始学习的时候，就能提前判断（命中率约有九成）要学的知识属于哪一类。这些便是 Unix 有众多忠实拥趸的原因。

Unix 的稳定和成功在很大程度上归功于它与生俱来的内在优势，归功于 Ken Thompson, Dennis Ritchie, Brian Kernighan, Doug McIroy, Rob Pike 和其他早期 Unix 开发者一开始就作出的设计决策。这些决策，连同设计哲学、编程艺术、技术文化一起，从 Unix 的婴儿期到今天的成长路程中，已经被反复证明是健康可靠的，而 Unix 才得以有今天的成功。

1.3 反对学习 Unix 文化的理由

Unix 的耐用性及其技术文化对于喜爱 Unix 的人们、以及技术史家来说肯定颇为有趣。但是，Unix 的本源用途——作为大中型计算机的通用分时系统，由于受到个人工作站的围剿，正迅速地退出舞台，隐入历史的迷雾之中。因而 Unix 究竟能否在目前被 Microsoft 主宰的主流商务桌面市场上取得成功，人们自然也存在着一定的疑问。

外行常常把 Unix 当作是教学用的玩具或者是黑客的沙盒而不屑一顾。有一本著名的抨击 Unix 的书——《Unix 反对者手册》（Unix Hater's Handbook）[Garfinkel]，几乎从 Unix 诞生时就一直奉行反对路线，将 Unix 的追随者描写成一群信奉邪教的怪人

和失败者。AT&T、Sun、Novell，以及其他一些大型商业销售商和标准联盟在 Unix 定位和市场推广方面不断铸下的大错也已经成为经典笑柄。

即使在 Unix 世界里，Unix 的通用性也一直受到怀疑，摇摆在危崖边。在持怀疑态度的外行人眼中，Unix 很有用，不会消亡，只是登不了大雅之堂；注定只能是个小众的操作系统。

挫败这些怀疑者的不是别的，正是 Linux 和其它开源 Unix（如现代 BSD 各个变种）的崛起。Unix 文化是如此的有生命力，即使十几年的管理不善也丝毫未箝制它的勃勃生机。现在 Unix 社区自身已经重新控制了技术和市场，正快速而有效地解决着 Unix 的问题（第 20 章将有详述）。

1.4 Unix 之失

对于一个始于 1969 年的设计来说，在 Unix 设计中居然很难找到硬伤，这着实令人称奇。其它的选择不是没有，但是每一个这样的选择同样面临争论，无论是 Unix 爱好者，还是操作系统设计社群的人们。

Unix 文件在字节层次以上再无结构可言。文件删除了就没法恢复。Unix 的安全模型公认地太过原始。作业控制有欠精致。命名方式非常混乱。或许拥有文件系统本身就是一个错误。我们将在第 20 章讨论这些技术问题。

但是，也许 Unix 最持久的异议恰恰来自 Unix 哲学的一个特性，这一条特性是 X window 设计者首先明确提出的。X 致力于提供 一套“机制，而不是策略”，以支持一套极端通用的图形操作，从而把使用工具箱和界面的“观感”（策略）推后到应用层。Unix 其它系统级的服务也有类似的倾向：行为的最终逻辑被尽可能推后到使用端。Unix 用户可以在多种 shell 中进行选择。而 Unix 应用程序通常会提供很多的行为选项和令人眼花缭乱的定制功能。

这种倾向也反映出 Unix 的遗风：原本是为技术人员设计的操作系统；同时也表明设计的信念：最终用户永远比操作系统设计人员更清楚他们究竟需要什么。

> 贝尔实验室的 Dick Hamming[3]在 1950 年代便树立了此信条：尽管计算机稀缺昂贵，但是开放式的计算模式，即客户可以为系统写出自己的应用程序，这一点势在必行，因为“用错误的方式解决正确的问题总比用正确的方法解决错误的问题好”。
>
> —Doug McIlroy

然而这种选择机制而不是策略的代价是：当用户“**可以**”自己设置策略时，他们其实是“必须”自己设置策略。非技术型的终端用户常常会被 Unix 丰富的选项和接口风格搞得晕头转向，于是转而选择那些伪称能够给他们提供简洁性的操作系统。

只看眼前的话，Unix 的这种自由放纵主义风格会让它失去很多非技术型用户。但从长远考虑，最终你会发觉这个“错误”换来至关重要的优势：策略相对短寿，而机制才会长存。现今流行的界面观感常常会变成明日进化的死胡同（去问问那些使用已经过时的 X 工具包的用户，他们会有一肚子苦水倒给你！）。说来说去，只提供机制不提供方针的哲学能使 Unix 长久保鲜；而那些被束缚在一套方针或界面风格内的操作系统，也许早就从人们的视线中消失了。[4]

1.5 Unix 之得

最近 Linux 爆炸式的发展和 Internet 技术重要性的渐增，都给我们充足的理由来否定怀疑者的论断。其实，退一步说，就算怀疑者的断言正确，Unix 文化也同样值得研习，因为在有些方面，Unix 及其外围文化明显比任何竞争对手都出色。

1.5.1 开源软件

尽管“开源”这个术语和开源定义（the Open Source Definition）直到 1998 年才出现，但是自由共享源码的同僚严格复审的开发方式打从 Unix 诞生起就是其文化最具特色的部分。

[3] 对，就是创立“汉明距离”和“汉明码”的那位汉明（Hamming）。

[4] 注：X 的架构者之一 Jim Gettys（也为本书撰写了部分内容）鞭辟入里地揭示了 X 的自由放纵主义才使得它有如此成就。无论就其专门建议，还是对 Unix 理念的表述，这篇小文都极其值得一读。

最初十年中的 AT&T 原始 Unix，及其后来的主要变种 Berkeley Unix，通常都随源代码一起发布。下文要提到的 Unix 的优势，大多数也由此而来。

1.5.2 跨平台可移植性和开放标准

Unix 仍是唯一一个在不同种类的计算机、众多厂商、各种专用硬件上提供了一个一致的、文档齐全的应用程序接口（API）的操作系统。Unix 也是唯一一个从嵌入式芯片、手持设备到桌面机，从服务器到专门用于数值计算的怪兽级计算机以及数据库后端都腾挪有余的操作系统。

Unix API 几乎就可以作为编写真正可移植软件的硬件无关标准。难怪最初 IEEE 称之为 “可移植操作系统标准”（Portable Operating System Standard）的 POS 很快就被大家加了后缀变成了“POSIX”[译注：缩写为 POSIX 是为了读音更像 Unix]。 确实，只有称之为 Unix API 的等价物才能算是这种标准比较可信的模型。

其它操作系统只提供二进制代码的应用程序，并随其诞生环境的消亡而消亡，而 Unix 源码却是永生的。至少，永生在数十年不断维护翻修它们的 Unix 技术文化之中。

1.5.3 Internet 和万维网

美国国防部将第一版 TCP/IP 协议栈的开发合同交给一个 Unix 研发组就是因为考虑到 Unix 大部分是开放源码。除了 TCP/IP 之外, Unix 也已成为互联网服务提供商 (Internet Service Provider)行业不可或缺的核心技术之一。甚至在 1980 年代中期 TOPS 系列操作系统消亡之际, 大部分互联网服务器(实际上 PC 以上所有级别的机器)都依赖于 Unix。

在 Internet 市场上，Unix 甚至面对 Microsoft 可怕的行销大锤也毫发无伤。 虽然成型于 TOPS-10 的 TCP/IP 标准（互联网的基础） 在理论上可以与 Unix 分开, 但当应用在其它操作系统上时， 一直都饱受兼容性差、不稳定、bug 太多等问题的困扰。实际上，理论和规格说明人人都可以获取, 但是只有 Unix 世界中你才见得到这些稳固可靠的现实成果。[5]

[5] 别的操作系统通常已经复制或者沿袭了 Unix TCP/IP 的实现， 但是很遗憾，他们没学到 Unix 实现代码幕后的同僚复审这个强力传统，看看 RFC1025(*TCP and IP Bake Off*——“TCP/IP 不同实现大比拼”)就知道我是什么意思了。

互联网技术文化和 Unix 文化在 1980 年代早期开始汇合，现在已经共生共存，难以分割。万维网的设计——也就是互联网的现代面孔，从其祖先 ARPANET 所得到的，不比从 Unix 得到的更多。实际上,统一资源定位符 URL(Uniform Resource Locator)作为 Web 的核心概念，也是 Unix 中无处不在的统一文件名字空间概念的泛化。要作为一个有效的网络专家，对 Unix 及其文化的理解绝对是必不可少的。

1.5.4 开源社区

伴随早期 Unix 源码发布而形成的社群从未消亡——在 1990 年代早期互联网技术的爆炸式发展之后，这个社群新造就了整整一代的使用家用机的狂热黑客。

今天，Unix 社区是各种软件开发的强大支持组。 高质量的开源开发工具在 Unix 世界极为丰富(在本书中我们会讲到很多)。开源的 Unix 应用程序已经达到、或者超越它们专属同侪的高度[Fuzz]。整个 Unix 操作系统连同完整的工具包、基本的应用套件，都可以在互联网上免费获取。既然能够改编、重用、再造，节省自己 90%的工作量，为什么还要从零开始编码呢?

通过协作开发与代码复用路上艰辛的探索，才耕耘出代码共享的传统。不是在理论上，而是通过大量工程实践，才有了这些并非显而易见的设计规则：程序得以形成严丝合缝的工具套装，而不是应景的解决对策。本书的一个主要目的就是阐明这些原则。

今天, 方兴未艾的开源运动给 Unix 传统注入了新的血液、新的技术方法，同时也带来了新一代年轻而有才华的程序员。包括 Linux 操作系统以及共生的应用程序如 Apache、Mozilla 等开源项目已经使 Unix 传统在主流世界空前亮眼与成功。如今，在争相对未来计算基础设施进行定义的这场竞争中，开源运动似乎已经站在了胜利的边缘——新架构的核心正是运行在互联网上的 Unix 机器。

1.5.5 从头到脚的灵活性

许多操作系统自诩比起 Unix 来有多么的“现代”，用户界面又是多么的“友好”。它们漂亮外表的背后，却是以貌似精巧实则脆弱狭隘难用的编程接口，把用户和开发者禁锢在单一的界面方针下。在这样的操作系统中，完成设计者（指操作系统）预见的任务

很容易，但如果要完成设计者没有预料到的任务，用户不是无计可施就是痛苦不堪。

相反，Unix 具有非常彻底的灵活性。Unix 提供众多的程序粘合手段，这意味着 Unix 基本工具箱的各种组件连纵开合后，将收到单个工具设计者无法想象的功效。

Unix 支持多种风格的程序界面（通常也因为给终端用户增加了明显的系统复杂度而被视为 Unix 的一个缺点），从而增加了它的灵活性；只管简单数据处理的程序而无需背上精巧图形界面的担子。

Unix 传统将重点放在尽力使各个程序接口相对小巧、简洁和正交——这也是另一个提高灵活性的方面。整个 Unix 系统，容易的事还是那么容易，困难的事呢，至少是有可能做到的。

1.5.6 Unix Hack 之趣

那些夸夸其谈 Unix 技术优越性的家伙一般不会提到 Unix 的终极法宝、它赖以成功的原因：Unix Hack 的趣味。

一些 Unix 的玩家有时羞于认同这一点，似乎这会破坏他们的正统形象。但是，确实如此，同 Unix 打交道，搞开发就是好玩；现在是，且一向如是。

并没有多少操作系统会被人们用“好玩”来描述。实际上，在其它操作系统下搞开发的摩擦和艰辛，就像是有人比喻的“把一头搁浅的死鲸推下海”[6]一样费力不讨好；或者，最客气的也就是“尚可容忍”、“不是太痛苦”之类形容词。与之成鲜明对比的是，在 Unix 世界里，操作系统以成就感而不是挫折感来回报人们的努力。Unix 下的程序员通常会把 Unix 当作一个积极有效的帮手，而不是把操作系统当作一个对手还非得用蛮力逼迫它干活。

这一点有着实实在在的重要经济意义。趣味性在 Unix 早期的历史中开启了一个良性循环。正是因为人们喜爱 Unix，所以编制了更多的程序让它用起来更好， 而如今，连编制一个完整商用产品级的开源 Unix 操作系统都成了一项爱好。如果想知道这是多么惊人的伟绩，想想看你什么时候听说过谁为了好玩来临摹 OS/360 或者 VAX VMS 或者 Microsoft Windows 就行了。

从设计角度来说，趣味性也绝非无足轻重。对于程序员和开发人员来说，如果完成某项任务所需要付出的努力对他们是个挑战却又恰好还在力所能及的范围内，他们就会

[6] 语出 Stephen C. Johnson 对 IBM MVS TSO 机制发的牢骚，此人是 yacc 的作者。

觉得很有乐趣。因此，趣味性是一个峰值效率的标志。充满痛苦的开发环境只会浪费劳动力和创造力；这样的环境会在无形之中耗费大量时间、资金，还有机会。

就算 Unix 在其它各个方面都一无是处，Unix 的工程文化仍然值得学习，它使得开发过程充满乐趣。乐趣是一个符号，意味着效能、效率和高产。

1.5.7 Unix 的经验别处也可适用

在探索开发那些我们如今已经觉得理所当然的操作系统特性的过程中，Unix 程序员已经积累了几十年的经验。哪怕是非 Unix 的程序员也能够从这些经验中获益。好的设计原则和开发方法在 Unix 上实施相对容易，所以 Unix 是一个学习这些原则和方法的良好平台。

在其它操作系统下，要做到良好实践通常要相对困难一些，但是尽管如此，Unix 文化中的有益经验仍然可以借鉴。多数 Unix 代码(包括所有的过滤器、主要脚本语言和大多数代码生成器)都可以直接移植到任何只要支持 ANSI C 的操作系统中(原因在于 C 语言本身就是 Unix 的一项发明，而 ANSI C 程序库表述了相当大一部分的 Unix 服务)。

1.6 Unix 哲学基础

Unix 哲学起源于 Ken Thompson 早期关于如何设计一个服务接口简洁、小巧精干的操作系统的思考，随着 Unix 文化在学习如何尽可能发掘 Thompson 设计思想的过程中不断成长，同时一路上还从其它许多地方博采众长。

Unix 哲学说来不算是一种正规设计方法。它并不打算从计算机科学的理论高度来产生理论上完美的软件。那些毫无动力、松松垮垮而且薪水微薄的程序员们，能在短短期限内，如同神灵附体般造出稳定而新颖的软件——这只不过是经理人永远的梦呓罢了。

Unix 哲学（同其它工程领域的民间传统一样）是自下而上的，而不是自上而下的。Unix 哲学注重实效，立足于丰富的经验。你不会在正规方法学和标准中找到它，它更接近于隐性的半本能的知识，即 Unix 文化所传播的**专业经验**。它鼓励那种分清轻重缓急的感觉，以及怀疑一切的态度，并鼓励你以幽默达观的态度对待这些。

Unix 管道的发明人、Unix 传统的奠基人之一 Doug McIlroy 在[McIlroy78]中曾经说过：

（i）让每个程序就做好一件事。如果有新任务，就重新开始，不要往原程序中加入新功能而搞得复杂。

（ii）假定每个程序的输出都会成为另一个程序的输入，哪怕那个程序还是未知的。输出中不要有无关的信息干扰。避免使用严格的分栏格式和二进制格式输入。不要坚持使用交互式输入。

（iii）尽可能早地将设计和编译的软件投入试用，哪怕是操作系统也不例外，理想情况下，应该是在几星期内。对拙劣的代码别犹豫，扔掉重写。

（iv）优先使用工具而不是拙劣的帮助来减轻编程任务的负担。工欲善其事，必先利其器。

后来他这样总结道（引自《Unix 的四分之一世纪》（A Quarter Century of Unix [Salus]））：

Unix 哲学是这样的：一个程序只做一件事，并做好。程序要能协作。程序要能处理文本流，因为这是最通用的接口。

Rob Pike，最伟大的 C 语言大师之一，在《Notes on C Programming》中从另一个稍微不同的角度表述了 Unix 的哲学[Pike]：

原则 1：你无法断定程序会在什么地方耗费运行时间。瓶颈经常出现在想不到的地方，所以别急于胡乱找个地方改代码，除非你已经证实那儿就是瓶颈所在。

原则 2：估量。在你没对代码进行估量，特别是没找到最耗时的那部分之前，别去优化速度。

原则 3：花哨的算法在 n 很小时通常很慢，而 n 通常很小。花哨算法的常数复杂度很大。除非你确定 n 总是很大，否则不要用花哨算法（即使 n 很大，也优先考虑原则 2）。

原则 4：花哨的算法比简单算法更容易出 bug、更难实现。尽量使用简单的算法配合简单的数据结构。

原则 5：数据压倒一切。如果已经选择了正确的数据结构并且把一切都组织得井井有条，正确的算法也就不言自明。编程的核心是数据结构，而不是算法[7]。

[7] Pike 的原稿在这里补充了“（参考 Brooks p. 102.）”。引用是来自于 The Mythical Man－Month【Brooks】早期的版本；引语为“给我看流程图而不让我看（数据）表，我仍会茫然不解；如果给我看（数据）表，通常就不需要流程图了；数据表是够说明问题了。”

原则 6：没有原则 6。

Ken Thompson——Unix 最初版本的设计者和实现者，禅宗偈语般地对 Pike 的原则 4 作了强调：

拿不准就穷举。

Unix 哲学中更多的内容不是这些先哲们口头表述出来的，而是由他们所作的一切和 Unix 本身所作出的榜样体现出来的。从整体上来说，可以概括为以下几点：

1. 模块原则：使用简洁的接口拼合简单的部件。
2. 清晰原则：清晰胜于机巧。
3. 组合原则：设计时考虑拼接组合。
4. 分离原则：策略同机制分离，接口同引擎分离。
5. 简洁原则：设计要简洁，复杂度能低则低。
6. 吝啬原则：除非确无它法，不要编写庞大的程序。
7. 透明性原则：设计要可见，以便审查和调试。
8. 健壮原则：健壮源于透明与简洁。
9. 表示原则：把知识叠入数据以求逻辑质朴而健壮。
10. 通俗原则：接口设计避免标新立异。
11. 缄默原则：如果一个程序没什么好说的，就沉默。
12. 补救原则：出现异常时，马上退出并给出足够错误信息。
13. 经济原则：宁花机器一分，不花程序员一秒。
14. 生成原则：避免手工 hack，尽量编写程序去生成程序。

15. 优化原则：雕琢前先要有原型，跑之前先学会走。

16. 多样原则：决不相信所谓“不二法门”的断言。

17. 扩展原则：设计着眼未来，未来总比预想来得快。

如果刚开始接触 Unix，这些原则值得好好体味一番。谈软件工程的文章常常会推荐大部分的这些原则，但是大多数其它操作系统缺乏恰当的工具和传统将这些准则付诸实践，所以，多数的程序员还不能自始至终地贯彻这些原则。蹩脚的工具、糟糕的设计、过度的劳作和臃肿的代码对他们已经是家常便饭了；他们奇怪，Unix 的玩家有什么好烦的呢。

1.6.1 模块原则：使用简洁的接口拼合简单的部件

正如 Brian Kernighan 曾经说过的：“计算机编程的本质就是控制复杂度”[Kernighan-Plauger]。排错占用了大部分的开发时间，弄出一个拿得出手的可用系统，通常与其说出自才华横溢的设计成果，还不如说是跌跌撞撞的结果。

汇编语言、编译语言、流程图、过程化编程、结构化编程、所谓的人工智能、第四代编程语言、面向对象、以及软件开发的方法论，不计其数的解决之道被抛售者吹得神乎其神。但实际上这些都用处不大，原因恰恰在于它们“成功”地将程序的复杂度提升到了人脑几乎不能处理的地步。就像 Fred Brooks 的一句名言[Brooks]：没有万能药。

要编制复杂软件而又不至于一败涂地的唯一方法就是降低其整体复杂度——用清晰的接口把若干简单的模块组合成一个复杂软件。如此一来，多数问题只会局限于某个局部，那么就还有希望对局部进行改进而不至牵动全身。

1.6.2 清晰原则：清晰胜于机巧

维护如此重要而成本如此高昂；在写程序时，要想到你不是写给执行代码的计算机看的，而是给人——将来阅读维护源码的人，包括你自己——看的。

在 Unix 传统中，这个建议不仅意味着代码注释。良好的 Unix 实践同样信奉在选择

算法和实现时就应该考虑到将来的可扩展性。而为了取得程序一丁点的性能提升就大幅度增加技术的复杂性和晦涩性，这个买卖做不得——这不仅仅是因为复杂的代码容易滋生 bug，也因为它会使日后的阅读和维护工作更加艰难。

相反，优雅而清晰的代码不仅不容易崩溃——而且更易于让后来的修改者立刻理解。这点非常重要，尤其是说不定若干年后回过头来修改这些代码的人可能恰恰就是你自己。

> 永远不要去吃力地解读一段晦涩的代码三次。第一次也许侥幸成功，但如果发现必须重新解读一遍——离第一次太久了，具体细节无从回想——那么你该注释代码了，这样第三次就相对不会那么痛苦了。
>
> —Henry Spencer

1.6.3 组合原则：设计时考虑拼接组合

如果程序彼此之间不能有效通信，那么软件就难免会陷入复杂度的泥淖。

在输入输出方面，Unix 传统极力提倡采用简单、文本化、面向流、设备无关的格式。在经典的 Unix 下，多数程序都尽可能采用简单过滤器的形式，即将一个输入的简单文本流处理为一个简单的文本流输出。

抛开世俗眼光，Unix 程序员偏爱这种做法并不是因为他们仇视图形用户界面，而是因为如果程序不采用简单的文本输入输出流，它们就极难衔接。

Unix 中，文本流之于工具，就如同在面向对象环境中的消息之于对象。文本流界面的简洁性加强了工具的封装性。而许多精致的进程间通讯方法，比如远程过程调用，都存在牵扯过多各程序间内部状态的倾向。

要想让程序具有组合性，就要使程序彼此独立。在文本流这一端的程序应该尽可能不要考虑文本流另一端的程序。将一端的程序替换为另一个截然不同的程序，而完全不惊扰另一端应该很容易做到。

GUI 可以是个好东西。有时竭尽所能也不可避免复杂的二进制数据格式。但是，在做一个 GUI 前，最好还是应该想想可不可以把复杂的交互程序跟干粗活的算法程序分离开，每个部分单独成为一块，然后用一个简单的命令流或者是应用协议将其组合在一起。

在构思精巧的数据传输格式前，有必要实地考察一下，是否能利用简单的文本数据格式；以一点点格式解析的代价，换得可以使用通用工具来构造或解读数据流的好处是值得的。

当程序无法自然地使用序列化、协议形式的接口时，正确的 Unix 设计至少是，把尽可能多的编程元素组织为一套定义良好的 API。这样，至少你可以通过链接调用应用程序，或者可以根据不同任务的需求粘合使用不同的接口。

（我们将在第 7 章详细讨论这些问题。）

1.6.4 分离原则：策略同机制分离，接口同引擎分离

在 Unix 之失的讨论中，我们谈到过 X 系统的设计者在设计中的基本抉择是实行“机制，而不是策略”这种做法——使 X 成为一个通用图形引擎，而将用户界面风格留给工具包或者系统的其它层次来决定。这一点得以证明是正确的，因为策略和机制是按照不同的时间尺度变化的，策略的变化要远远快于机制。GUI 工具包的观感时尚来去匆匆，而光栅操作和组合却是永恒的。

所以，把策略同机制揉成一团有两个负面影响：一来会使策略变得死板，难以适应用户需求的改变，二来也意味着任何策略的改变都极有可能动摇机制。

相反，将两者剥离，就有可能在探索新策略的时候不足以打破机制。另外，我们也可以更容易为机制写出较好的测试（因为策略太短命，不值得花太多精力在这上面）。

这条设计准则在 GUI 环境之外也被广泛应用。总而言之，这条准则告诉我们应该设法将接口和引擎剥离开来。

实现这种剥离的一个方法是，比如，将应用按照一个库来编写，这个库包含许多由内嵌脚本语言驱动的 C 服务程序，而至于整个应用的控制流程则用脚本来撰写而不是用 C 语言。这种模式的经典例子就是 Emacs 编辑器，它使用内嵌的脚本语言 Lisp 解释器来控制用 C 编写的编辑原语操作。我们会在第 11 章讨论这种设计风格。

另一个方法是将应用程序分成可以协作的前端和后端进程，通过套接字上层的专用应用协议进行通讯；我们会在第 5 章和第 7 章讨论这种设计。前端实现策略，后端实现

机制。比起仅用单个进程的整体实现方式来说，这种双端设计方式大大降低了整体复杂度，bug 有望减少，从而降低程序的寿命周期成本。

1.6.5 简洁原则：设计要简洁，复杂度能低则低

来自多方面的压力常常会让程序变得复杂（由此代价更高，bug 更多），其中一种压力就是来自技术上的虚荣心理。程序员们都很聪明，常常以能玩转复杂东西和耍弄抽象概念的能力为傲，这一点也无可厚非。但正因如此，他们常常会与同行们比试，看看谁能够鼓捣出最错综复杂的美妙事物。正如我们经常所见，他们的设计能力大大超出他们的实现和排错能力，结果便是代价高昂的废品。

> “错综复杂的美妙事物”听来自相矛盾。Unix 程序员相互比的是谁能够做到“简洁而漂亮”并以此为荣，这一点虽然只是隐含在这些规则之中，但还是很值得公开提出来强调一下。
>
> —Doug McIlroy

更为常见的是(至少在商业软件领域里)，过度的复杂性往往来自于项目的要求，而这些要求常常基于当月的推销热点，而不是基于顾客的需求和软件实际能够提供的功能。许多优秀的设计被市场推销所需要的大堆大堆“特性清单”扼杀——实际上，这些特性功能几乎从未用过。然后，恶性循环开始了：比别人花哨的方法就是把自己变得更花哨。很快，庞大臃肿变成了业界标准，每个人都在使用臃肿不堪、bug 极多的软件，连软件开发人员也不敢敝帚自珍。

无论以上哪种方式，最后每个人都是失败者。

要避免这些陷阱，唯一的方法就是鼓励另一种软件文化，以简洁为美，人人对庞大复杂的东西群起而攻之——这是一个非常看重简单解决方案的工程传统，总是设法将程序系统分解为几个能够协作的小部分，并本能地抵制任何用过多噱头来粉饰程序的企图。

这就有点 Unix 文化的意味了。

1.6.6 吝啬原则：除非确无它法，不要编写庞大的程序

“大”有两重含义：体积大，复杂程度高。程序大了，维护起来就困难。由于人们对花费了大量精力才做出来的东西难以割舍，结果导致在庞大的程序中把投资浪费在注定要失败或者并非最佳的方案上。

（我们会在第 13 章就软件的最佳大小进行更多的详细讨论。）

1.6.7 透明性原则：设计要可见，以便审查和调试

因为调试通常会占用四分之三甚至更多的开发时间，所以一开始就多做点工作以减少日后调试的工作量会很划算。一个特别有效的减少调试工作量的方法就是设计时充分考虑透明性和显见性。

软件系统的透明性是指你一眼就能够看出软件是在做什么以及怎样做的。显见性指程序带有监视和显示内部状态的功能，这样程序不仅能够运行良好，而且还可以看得出它以何种方式运行。

设计时如果充分考虑到这些要求会给整个项目全过程都带来好处。至少，调试选项的设置应该尽量不要在事后，而应该在设计之初便考虑进去。这是考虑到程序不但应该能够展示其正确性，也应该能够把原开发者解决问题的思维模型告诉后来者。

程序如果要展示其正确性，应该使用足够简单的输入输出格式，这样才能保证很容易地检验有效输入和正确输出之间的关系是否正确。

出于充分考虑透明性和显见性的目的，还应该提倡接口简洁，以方便其它程序对其进行操作——尤其是测试监视工具和调试脚本。

1.6.8 健壮原则：健壮源于透明与简洁

软件的健壮性指软件不仅能在正常情况下运行良好，而且在超出设计者设想的意外条件下也能够运行良好。

大多数软件禁不起磕碰，毛病很多，就是因为过于复杂，很难通盘考虑。如果不能够正确理解一个程序的逻辑，就不能确信其是否正确，也就不能在出错的时候修复它。

这也就带来了让程序健壮的方法，就是让程序的内部逻辑更易于理解。要做到这一点主要有两种方法：透明化和简洁化。

> 就健壮性而言，设计时要考虑到能承受极端大量的输入，这一点也很重要。这时牢记组合原则会很有益处；经不起其它一些程序产生的输入（例如，原始的 Unix C 编译器据说需要一些小小的升级才能处理好 Yacc 的输出）。当然，这其中涉及的一些形式对人类来说往往看起来没什么实际用处。比如，接受空的列表/字符串等等，即使在人们很少或者根本就不提供空字符串的地方也得如此，这可以避免在用机器生成输入时需要对这种情况进行特殊处理。
>
> —Henry Spencer

在有异常输入的情况下，保证软件健壮性的一个相当重要的策略就是避免在代码中出现特例。bug 通常隐藏在处理特例的代码以及处理不同特殊情况的交互操作部分的代码中。

上面我们曾说过，软件的透明性就是指一眼就能够看出来是怎么回事。如果“怎么回事”不算复杂，即人们不需要绞尽脑汁就能够推断出所有可能的情况，那么这个程序就是简洁的。程序越简洁，越透明，也就越健壮.

模块性（代码简朴，接口简洁）是组织程序以达到更简洁目的的一个方法。另外也有其它的方法可以得到简洁。接下来就是另一个。

1.6.9 表示原则：把知识叠入数据以求逻辑质朴而健壮

即使最简单的程序逻辑让人类来验证也很困难，但是就算是很复杂的数据，对人类来说，还是相对容易地就能够推导和建模的。不信可以试试比较一下，是五十个节点的指针树，还是五十行代码的流程图更清楚明了；或者，比较一下究竟用一个数组初始化器来表示转换表，还是用 switch 语句更清楚明了呢？可以看出，不同的方式在透明性和清晰性方面具有非常显著的差别。参见 Rob Pike 的原则 5。

数据要比编程逻辑更容易驾驭。所以接下来，如果要在复杂数据和复杂代码中选择一个，宁愿选择前者。更进一步：在设计中，你应该主动将代码的复杂度转移到数据之中去。

此种考量并非 Unix 社区的原创，但是许多 Unix 代码都显示受其影响。特别是 C 语言对指针使用控制的功能，促进了在内核以上各个编码层面上对动态修改引用结构。在

结构中用非常简单的指针操作就能够完成的任务，在其它语言中，往往不得不用更复杂的过程才能完成。

（我们将在第 9 章再讨论这些技术。）

1.6.10 通俗原则：接口设计避免标新立异

（也就是众所周知的“最少惊奇原则”。）

最易用的程序就是用户需要学习新东西最少的程序——或者，换句话说，最易用的程序就是最切合用户已有知识的程序。

因此，接口设计应该避免毫无来由的标新立异和自作聪明。如果你编制一个计算器程序，‘＋’应该永远表示加法。而设计接口的时候，尽量按照用户最可能熟悉的同样功能接口和相似应用程序来进行建模。

关注目标受众。他们也许是最终用户，也许是其他程序员，也许是系统管理员。对于这些不同的人群，最少惊奇的意义也不同。

关注传统惯例。Unix 世界形成了一套系统的惯例，比如配置和运行控制文件的格式，命令行开关等等。这些惯例的存在有个极好的理由：缓和学习曲线。应该学会并使用这些惯例。

（我们将在第 5 章和第 10 章讨论这些传统惯例。）

> 最小立异原则的另一面是避免表象相似而实际却略有不同。这会极端危险，因为表象相似往往导致人们产生错误的假定。所以最好让不同事物有明显区别，而不要看起来几乎一模一样。
>
> —Henry Spencer

1.6.11 缄默原则：如果一个程序没什么好说的，就保持沉默

Unix 中最古老最持久的设计原则之一就是：若程序没有什么特别之处可讲，就保持沉默。行为良好的程序应该默默工作，决不唠唠叨叨，碍手碍脚。沉默是金。

“沉默是金”这个原则的起始是源于 Unix 诞生时还没有视频显示器。在 1969 年的缓慢的打印终端，每一行多余的输出都会严重消耗用户的宝贵时间。现在，这种情况已不复存在，一切从简的这个优良传统流传至今。

> 我认为简洁是 Unix 程序的核心风格。一旦程序的输出成为另一个程序的输入，就很容易把需要的数据挑出来。站在人的角度上来说——重要信息不应该混杂在冗长的程序内部行为信息中。如果显示的信息都是重要的，那就不用找了。
>
> —Ken Arnold

设计良好的程序将用户的注意力视为有限的宝贵资源，只有在必要时才要求使用。

（我们将在第 11 章末尾进一步讨论缄默原则及其理由。）

1.6.12 补救原则：出现异常时，马上退出并给出足量错误信息

软件在发生错误的时候也应该与在正常操作的情况下一样，有透明的逻辑。最理想的情况当然是软件能够适应和应付非正常操作；而如果补救措施明明没有成功，却悄无声息地埋下崩溃的隐患，直到很久以后才显现出来，这就是最坏的一种情况。

因此，软件要尽可能从容地应付各种错误输入和自身的运行错误。但是，如果做不到这一点，就让程序尽可能以一种容易诊断错误的方式终止。

同时也请注意 Postel 的规定[8]：“宽容地收，谨慎地发”。Postel 谈的是网络服务程序，但是其含义可以广为适用。就算输入的数据很不规范，一个设计良好的程序也会尽量领会其中的意义，以尽量与别的程序协作；然后，要么响亮地倒塌，要么为工作链下一环的程序输出一个严谨干净正确的数据。

然而，也请注意这条警告：

> 最初 HTML 文档推荐“宽容地接受数据”，结果因为每一种浏览器都只接受规范中一个不同的超集，使我们一直倍感无奈。要宽容的应该是规范而不是它们的解释工具。
>
> —Doug McIlroy

[8] Jonathan Postel 是第一个互联网 RFC 系列标准的编纂者，也是互联网的主要架构者之一。网上有一个由 Postel 实验网络中心（Postel Center for Experimental Networking）维护的纪念网页 <http://www.postel.org/postel.html>。

McIlroy 要求我们在设计时要考虑宽容性，而不是用过分纵容的实现来补救标准的不足。否则，正如他所指出的一样，一不留神你会死得很难看。

1.6.13 经济原则：宁花机器一分，不花程序员一秒

在 Unix 早期的小型机时代，这一条观点还是相当激进的（那时机器要比现在慢得多也贵得多）。如今，随着技术的发展，开发公司和大多数用户（那些需要对核爆炸进行建模或处理三维电影动画的除外）都能够得到廉价的机器，所以这一准则的合理性就显然不用多说啦！

但不知何故，实践似乎还没完全跟上现实的步伐。如果我们在整个软件开发中很严格的遵循这条原则的话，大多数的应用场合都应该使用高一级的语言，如 Perl、Tcl、Python、Java、Lisp，甚至 shell——这些语言可以将程序员从自行管理内存的负担中解放出来（参见[Ravenbrook]）。

这种做法在 Unix 世界中已经开始施行，尽管 Unix 之外的大多数软件商仍坚持采用旧 Unix 学派的 C(或 C++)编码方法。本书会在后面详细讨论这个策略及其利弊权衡。

另一个可以显著节约程序员时间的方法是：教会机器如何做更多低层次的编程工作，这就引出了……

1.6.14 生成原则：避免手工 hack，尽量编写程序去生成程序

众所周知，人类很不善于干辛苦的细节工作。因此，程序中的任何手工 hacking 都是滋生错误和延误的温床。程序规格越简单越抽象，设计者就越容易做对。由程序生成代码几乎(在各个层次)总是比手写代码廉价并且更值得信赖。

我们都知道确实如此（毕竟这就是为什么会有编译器、解释器的原因），但我们却常常不去考虑其潜在的含义。对于代码生成器来说，需要手写的重复而麻木的高级语言代码，与机器码一样是可以批量生产的。当代码生成器能够提升抽象度时——即当生成器的说明性语句要比生成码简单时，使用代码生成器会很合算，而生成代码后就根本无需再费力地去手工处理了。

在 Unix 传统中，人们大量使用代码生成器使易于出错的细节工作自动化。Parser/Lexer 生成器就是其中的经典例子，而 makefile 生成器和 GUI 界面式的构建器（interface builder）则是新一代的例子。

（我们会在第 9 章讨论这些技术。）

1.6.15 优化原则：雕琢前先得有原型，跑之前先学会走

原型设计最基本的原则最初来自于 Kernighan 和 Plauger 所说的“90%的功能现在能实现，比 100%的功能永远实现不了强”。做好原型设计可以帮助你避免为蝇头小利而投入过多的时间。

由于略微不同的一些原因，Donald Knuth（程序设计领域中屈指可数的经典著作之一《计算机程序设计艺术》的作者）广为传播普及了这样的观点：“过早优化是万恶之源”[9]。他是对的。

还不知道瓶颈所在就匆忙进行优化，这可能是唯一一个比乱加功能更损害设计的错误。从畸形的代码到杂乱无章的数据布局，牺牲透明性和简洁性而片面追求速度、内存或者磁盘使用的后果随处可见。滋生无数 bug，耗费以百万计的人时——这点芝麻大的好处，远不能抵消后续排错所付出的代价。

经常令人不安的是，过早的局部优化实际上会妨碍全局优化（从而降低整体性能）。在整体设计中可以带来更多效益的修改常常会受到一个过早局部优化的干扰，结果，出来的产品既性能低劣又代码过于复杂。

在 Unix 世界里，有一个非常明确的悠久传统（例证之一是 Rob Pike 以上的评论，另一个是 Ken Thompson 关于穷举法的格言）：**先制作原型，再精雕细琢。优化之前先确保能用**。或者：先能走，再学跑。“极限编程”宗师 Kent Beck 从另一种不同的文化将这一点有效地扩展为：先求运行，再求正确，最后求快。

所有这些话的实质其实是一个意思：先给你的设计做个未优化的、运行缓慢、很耗内存但是正确的实现，然后进行系统地调整，寻找那些可以通过牺牲最小的局部简洁性而获得较大性能提升的地方。

[9] 完整的句子是这样的：“97%的时间里，我们不应考虑蝇头小利的效率提升：过早优化是万恶之源”。Knuth 自称这一观点来自 C. A. R. Hoare。

制作原型对于系统设计和优化同样重要——比起阅读一个冗长的规格说明，判断一个原型究竟是不是符合设想要容易得多。我记得 Bellcore 有一位开发经理，他在人们还没有谈论“快速原型化”和“敏捷开发”前好几年就反对所谓的“需求”文化。他从不提交冗长的规格说明，而是把一些 shell 脚本和 awk 代码结合在一起，使其基本能够完成所需要的任务，然后告诉客户派几个职员来使用这些原型，问他们是否喜欢。如果喜欢，他就会说“在多少多少个月之后，花多少多少的钱就可以获得一个商业版本”。他的估计往往很精确，但由于当时的文化，他还是输给了那些相信需求分析应该主导一切的同行。

—Mike Lesk

借助原型化找出哪些功能不必实现，有助于对性能进行优化；那些不用写的代码显然无需优化。目前，最强大的优化工具恐怕就是 delete 键了。

我最有成效的一天就是扔掉了 1000 行代码。

—Ken Thompson

（我们将在第 12 章对相关内容进行深一步讨论。）

1.6.16 多样原则：决不相信所谓“不二法门”的断言

即使最出色的软件也常常会受限于设计者的想象力。没有人能聪明到把所有东西都最优化，也不可能预想到软件所有可能的用途。设计一个僵化、封闭、不愿与外界沟通的软件，简直就是一种病态的傲慢。

因此，对于软件设计和实现来说，Unix 传统有一点很好，即从不相信任何所谓的“不二法门”。Unix 奉行的是广泛采用多种语言、开放的可扩展系统和用户定制机制。

1.6.17 扩展原则：设计着眼未来，未来总比预想快

如果说相信别人所宣称的“不二法门”是不明智的话，那么坚信自己的设计是“不二法门”简直就是愚蠢了。决不要认为自己找到了最终答案。因此，要为数据格式和代

码留下扩展的空间，否则，就会发现自己常常被原先的不明智选择捆住了手脚，因为你无法既要改变它们又要维持对原来的兼容性。

设计协议或是文件格式时，应使其具有充分的自描述性以便可以扩展。一直，总是，要么包含进一个版本号，要么采用独立、自描述的语句，按照可以随时插入新的、换掉旧的而不会搞乱格式读取代码的方法组织格式。Unix 经验告诉我们：稍微增加一点让数据部署具有自描述性的开销，就可以在无需破坏整体的情况下进行扩展，你的付出也就得到了成千倍的回报。

设计代码时，要有很好的组织，让将来的开发者增加新功能时无需拆毁或重建整个架构。当然这个原则并不是说你能随意增加根本用不上的功能，而是建议在编写代码时要考虑到将来的需要，使以后增加功能比较容易。程序接合部要灵活，在代码中加入“如果你需要……”的注释。有义务给之后使用和维护自己编写的代码的人做点好事。

也许将来就是你自己来维护代码，而在最近项目的压力之下你很可能把这些代码都遗忘了一半。所以，设计为将来着眼，节省的有可能就是自己的精力。

1.7 Unix 哲学之一言以蔽之

所有的 Unix 哲学浓缩为一条铁律，那就是各地编程大师们奉为圭臬的“KISS”原则：

Unix 提供了一个应用 KISS 原则的良好环境。本书的剩余部分将帮助你学习如何应用这个原则。

1.8 应用 Unix 哲学

这些富有哲理的原则决不是模糊笼统的泛泛之谈。在 Unix 世界中，这些原则都直接来自于实践，并形成了具体的规定，我们已经在上文中阐述了一些。以下列举的只是部分内容：

- 只要可行，一切都应该做成与来源和目标无关的过滤器。
- 数据流应尽可能文本化（这样可以使用标准工具来查看和过滤）。
- 数据库部署和应用协议应尽可能文本化（让人可以阅读和编辑）。
- 复杂的前端（用户界面）和后端应该泾渭分明。
- 如果可能，用 C 编写前，先用解释性语言搭建原型。
- 当且仅当只用一门语言编程会提高程序复杂度时，混用语言编程才比单一语言编程来得好。
- 宽收严发（对接收的东西要包容，对输出的东西要严格）。
- 过滤时，不需要丢弃的信息决不丢。
- 小就是美。在确保完成任务的基础上，程序功能尽可能少。

在本书的余下部分，我们会看到这些 Unix 的设计原则及其衍生的设计规则被反复运用于实践。毫不奇怪，这些往往与其它传统中最优秀的软件工程实践思想不谋而合。[10]

1.9 态度也要紧

看到该做的就去做——短期来看似乎是多做了，但从长期来看，这才是最佳捷径。如果不能确定什么是对的，那么就只做最少量的工作，确保任务完成就行，至少直到明白什么是对的。

[10] 我在本书准备工作的后期发现一个值得注意的例子就是 Butler Lampson 的《Hints for Computer System Design》[lampson]。这本书不仅通过显然是独立发现的形式表达了一系列的 Unix 格言，甚至还使用了许多同样的结语来进行阐述。

要良好的运用 Unix 哲学，你就应该不断追求卓越。你必须相信，软件设计是一门技艺，值得你付出所有的智慧、创造力和激情。否则，你的视线就不会超越那些简单、老套的设计和实现；你就会在应该思考的时候急急忙忙跑去编程。你就会在该无情删繁就简的时候反而把问题复杂化——然后你还会反过来奇怪你的代码怎么会那么臃肿、那么难以调试。

要良好地运用 Unix 哲学，你应该珍惜你的时间决不浪费。一旦某人已经解决了某个问题，就直接拿来利用，不要让骄傲或偏见拽住你又去重做一遍。永远不要蛮干；要多用巧劲，省下力气到需要的时候再用，好钢用在刀刃上。善用工具，尽可能将一切都自动化。

软件设计和实现应该是一门充满快乐的艺术，一种高水平的游戏。如果这种态度对你来说听起来有些荒谬，或者令你隐约感到有些困窘，那么请停下来，想一想，问问自己是不是已经把什么给遗忘了。如果只是为了赚钱或是打发时间，你为什么要搞软件设计而不是别的什么呢？你肯定曾经也认为软件设计值得你付出激情……

要良好地运用 Unix 哲学，你需要具备（或者找回）这种态度。你需要用心。你需要去游戏。你需要乐于探索。

我们希望你能带着这种态度来阅读本书的其它部分。或者，至少，我们希望本书能帮助你重拾这种态度。