unix 编程

艺术

OCR 识别 + 精确校对版

Eric S. Raymond[1] | 德山书生[2]

版本：0.01

---

[1] 作者：埃里克·斯蒂芬·雷蒙（Eric Steven Raymond，著名黑客。）

[2] 编者：万泽，湖南常德人氏。邮箱：a358003542@gmail.com。

目 录

哲学

> 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

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，会使你写出更好地软件，即使你是在非 Unix 平台开发。

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质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形只是，他们用类似禅宗“教外别传”[译注[1]]的方式，通过言传身教传授给后辈。

软件工程师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。

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元前的事了——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个时代的古物。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他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他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

---

[1] 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。

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，或者被其他操作系统挤出市场。可是在今天，化身为 Linux、BSD、Solaris、MacOS X 以及好几种其它变种的 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[2]。Unix 也多次经历了类似的转变。
>
> —Ken Thompson

至少，Unix 的一个核心技术——C 语言——已经在其他系统中植根。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已经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惊奇。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，业界规范几次变革——然而 Unix 依然巍然屹立，仍在运行，仍在创造价值，仍然能够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。而在其他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

---

[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。

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）要学的知识属于那一类。这些便是 Unix 有众多忠实用拥趸的原因。

Unix 的稳定和成功在很多程度上归功于它与生俱来的内在优势，归功于 Ken Thompson, Dennis Ritchie, Brain Kernighan, Doug McIroy, Rob Pike 和其他早期 Unix 开发者一开始作出的设计决策。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。

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，只是等不了大雅之堂：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 和其他开源 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[3]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）。

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。其他的选择不是没有，但是每一个这样的选择同样面临争议，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，从而把工具箱和界面的“观感”（策略）推后到应用层。Unix 其他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。

---

[3] 同抑制，而钳制有胁迫之意。

> 贝尔实验室的 Dick Hamming 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性用户，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的操作系统，也许早就从人们的视线中消失了。

Unix 之得

最近 Linux 爆炸式的发展和 Intemet 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。

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。

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。

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”（Portabje 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。

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 (Intemet 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 Intemet 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。

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。

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。

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。

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”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足处，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。

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）。

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 Mcllroy 在 [Mcllroy78] 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[4]。

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] 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。”

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。

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。

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

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。）

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。

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。

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。）

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。

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。

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。）

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

缄默原则：如果一个程序没什么好说的，就保持沉默

Unix 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 1 1 章末尾进一步讨论缄默原则及其理由。）

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[5]：“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，

---

[5] 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。

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

Mcllroy 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。

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……

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。）

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”[6]。他是对的。

---

[6] 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 C. A. R.

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。

> 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

Hoare。

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。）

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。

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。

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桌的“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。kiss

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。[7]

---

[7] 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

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。

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。

通过显然是独立发现的形式表达了一系列的 Unix 格言，甚至还使用了同样的结语来进行阐述。

历史——双流记

History: A Tale of Two Cultures

> 忘记过去的人，注定要重蹈覆辙。
>
> 《理性生活》（1905 年）
>
> —George Santayana

前事不忘，后事之师。Unix 的历史悠久且丰富多彩，许多内容仍然以坊间传说、猜想，以及（更常见的是）Unix 程序员集体记忆中的战争创伤等形式鲜活地留存着。本章我们将通过回顾 Unix 的历史来阐明如今的 Unix 文化为什么会呈现当前这种状态。

Unix 的起源及历史，1969—1995

小型实验原型系统的后继产品往往备受令人讨厌的“第二版效应”折磨。由于迫切希望把所有首次开发时遗漏的功能都添加进去，往往导致设计十分庞大、过于复杂。其实，还有一个因不常遇到而鲜为人知的“第三版效应”：有时候，在第二系统不堪自身重负而崩溃之后，有可能返璞归真，走上正道。

最初的 Unix 就是一个第三系统。Unix 的祖辈是小而简单的兼容分时系统（CTSS，Compatible Time-Sharing System），也算曾经实施过的分时系统的第一代或者第二代了（取决于不同的定义，具体我们在此不作讨论）。Unix 的父辈是颇具开拓性的 Multics 项目，该项目试图建立一个具备众多功能的“信息功用体／应用工具（information utilitv）”，能够很漂亮地支持大群用户对大型计算机的交互式分时使用。唉，Multics 最后因不堪自身重负而崩溃了。但 Unix 却正是从它的废墟中破壳而出的。

创世纪：1969-1971

Unix 于 1969 年诞生于贝尔实验室的计算机科学家 Ken Thompson 的头脑中。Thompson 曾经是 Multics 项目的研究人员，饱受当时几乎作为铁律而到处应用的原始批量计算的困扰。然而在六十年代晚期，分时系统还是个新鲜玩意儿。计算机科学家 John McCarthy（Lisp 语言的发明者[1]）几乎是在十年前才首次发表了分时系统的构想，而直到 Unix 诞生前七年的 1962 年才第一次真正部署使用，因此当时的分时系统尚处实验阶段，像喜怒无常的野兽，性能极不稳定。

那个时代计算机硬件的原始程度，恐怕亲历者现在也很难以记清。那时最强大的机器所拥有的计算能力和内存还不如现在一个普通的手机。[2]视频显示终端才刚刚起步，六年以后才得到广泛应

---

[1] McCarthy：1971 年图灵奖获得者，主要贡献在人工智能方面；The concept was first described publicly in early 1957 by Bob Bemer as part of an article in *Automatic Control Magazine*. The first project to implement a timesharing system was initiated by John McCarthy.

[2] Ken Thompson 让我知道，如今手机的随机存储器（RAM）容量比 PDP-7 的随机存储器和磁盘存储量的总和还要多；那个年代所谓“大磁盘”的容量也不过 1 兆字节。

用。最早分时系统的标准交互设备就是 ASR-33 电传打字机——一个又慢又响的设备，只能在大卷的黄色纸张上打印大写字母。Unix 命令简洁、少说多作的传统正是从 ASR-33 开始的。

当贝尔实验室（Bell Labs）从 Multics 研究联盟中退出时，Ken Thompson 带着从 Multics 激发的灵感——如何创建一个文件系统——留了下来。他甚至没能留下一台机器来玩自己编写的“星际旅行”，这是个科幻游戏——模拟驾驶一艘火箭在太阳系中遨游。Unix 就在一台废弃的 PDP-7 小型机[3]（图 2-1）上问世了。这台 PDP-7 成为了“星际旅行”的游戏平台和 Thompson 关于操作系统设计思路的试验场。

Unix 的完整起源故事可参见 [Ritchie79]，这是从 Thompson 第一个合作者 Dennis Ritchie 的角度讲述的。Dennis Ritchie 后来以 Unix 的合作发明者和 C 语言的发明者而闻名于世。Dennis Ritchie、Doug Mcllroy 和其他一些同事，已经习惯了 Multics 环境下的交互计算方式，不愿意放弃这一能力。Thompson 的 PDP-7 操作系统给了他们一条救生绳。

---

[3] 网页http://www.fags.org/fags/dec-fag/pdp8上有关于 PDP 计算机的常见问题解答（FAQ），对在历史上除此（对 Unix 诞生所作贡献）之外默默无闻的 PDP-7 做了一些说明。

Ritchie 评述道：“我们希望保留的不仅仅是一个良好的编程环境，还包括一种能够形成伙伴关系的系统。经验告诉我们，远程访问（remote-access）和分时系统支持的公用计算，其本质不是用终端机代替打孔机来输入程序，而是鼓励频繁的交流。”计算机不应仅被视为一种逻辑设备而更应视为社群的立足点，这种观念深入人心。ARPANET（现今 Internet 的直系祖先）也发明于 1969 年。“伙伴关系”这一旋律将一直鸣奏在 Unix 的后继历史中。

Thompson 和 Ritchie“星际旅行”的实现引起了关注。起先，PDP-7 的软件不得不在通用电气公司（GE）的大型机上交叉编译。Thompson 和 Ritchie 为支持游戏开发而在 PDP-7 上编制的实用程序成了 Unix 的核心——虽然直到 1970 年才产生 Unix 这个名字。最初的缩写是“UNICS”（单路信息与计算服务，Uniplexed

Information and Computing Service），Ritchie 后来称之为“一个有点反叛 Multics 味道的双关语”，因为 Multics 是多路信息与计算服务（MULTIplexed Information and Computing Service）的英文缩写。

即使在最早期，PDP-7 Unix 已经拥有现今 Unix 的诸多共性，提供的编程环境也比当时读卡式批处理大型机的环境要舒服得多。Unix 几乎可以称得上第一个能让程序员直接坐在机器旁，飞快捕获稍纵即逝的灵感，并能一边编写一边测试的系统。Unix 的整个发展进程中都能吸引那些不堪忍受其它操作系统局限性的程序员自愿为它进行开发，这也一直是 Unix 不断拓展其能力的模式。这种模式早在贝尔实验室时就已确立了。

Unix 的轻装开发和方法上不拘一格的传统与生俱来。Multics 是项庞大的工程，硬件开发出来前必须编写几千页的技术说明书，而第一份跑起来的 Unix 代码只是在三个人头脑风暴了一把，然后由 Ken Thompson 花了两天时间来实现罢了——还是在一台破烂机器上完成的，而那个机器本来只作为一台“真正”计算机的图形终端！

Unix 的第一功，是 1971 年为贝尔实验室的专利部门进行“文字处理”的支持工作。首个 Unix 应用程序是 nroff(l) 文本格式化程序的前身。这个项目也让他们名正言顺地购买了一台功能强大得多的 PDP-11 小型机。万幸的是，当时管理层还未意识到 Thompson 和其同事所编写的字处理系统就快孵化出一个操作系统。贝尔实验室并没有开发操作系统的计划——AT&T 加入 Multics 联盟正是为了避免自行开发一个操作系统。不管怎样，整个系统还是取得了令人振奋的成功。Unix 在贝尔实验室计算群落中的重要而永久地位由此确立，并且开创了 Unix 历史的下一个主旋律——与文档格式化、排版和通讯工具的紧密结合。1972 年版的手册宣称装机量达 10 台。

Doug Mcllroy[Mcllroy91] 后来这样描述这个时代：“外界的压

力和纯粹出于对技艺的荣誉感，促使人们在有了更好更多的初步思路后，去重写或抛开已有的大量代码。从来没听说什么职业竞争和势力范围保护：好东西太多了，没有人需要把这些创新占为已有。”但是直到四分之一世纪后，人们才真正体会到他的话的含义。

出埃及记：1971-1980

最初的 Unix 用汇编语言写成，应用程序用汇编语言和解释型语言 B 混和编写。B 语言的优点在于小巧，能在 PDP-7 上运行，但是作为系统编程语言还不够强大，所以 Dennis Ritchie 给它增加了数据类型和结构。C 语言从 1977 年起自 B 语言进化而来；1973 年，Thompson 和 Ritchie 成功地用新语言重写了整个 Unix 系统。这是一个大胆的举动——那时为了最大程度地利用硬件性能，系统编程都通过汇编器来完成。与此同时，可移植操作系统的概念几乎鲜为人知。l979 年，Ritchie 终于可以这么写了：“很肯定，Unix 的成功很大程度上源自其以高级语言作为表述方式所带来的可读性、可改性和可移植性”，虽然理想与现实此时尚有一线距离。

1974 年在《美国计算机通信》（Communications of the ACM）上发表的一篇论文中 [Ritchie—Thompson] 第一次公开展示了 Unix。文中作者描述了 Unix 前所未有的简洁设计，并报告了 600 多例 Unix 应用——这些都是安装在即便按照那个年代的标准，性能都算很低的机器上，但是（正如 Ritchie 和 Thompson 所写）“性能的局限不仅成就了经济性，而且鼓励了设计的简约”。

CACM 论文发表后，全球各个研究实验室和大学都嚷着要亲身体验 Unix。根据 1958 年为解决反托拉斯案例达成的和解协议，AT&T（贝尔实验室的母公司）被禁止进入计算机相关的商业领域。所以，Unix 不能够成为一种商品。实际上，根据和解协议的规定，贝尔实验室必须将非电话业务的技术许可给任何提出要求

的人。Ken Thompson 开始默默回应那些请求，将磁带和磁盘一包包地寄送出去——据传说，每包里都有一张字条，写着“love，ken”（爱你的，ken）。

这离个人机出现还有些年。那时候，不仅运行 Unix 所必须的硬件设备价格超出个人的承受范围，而且也没人敢奢望这种情况会在可预见的未来改变。因此，只有预算充足的大机构才用得起 Unix 机器：公司、高校、政府机构等。但是，对这些小型机的使用管制要比那些大型机少得多，因此，Unix 的发展迅速笼罩了一层反传统文化的氛围。在上世纪 70 年代早期，最早搞 Unix 编程的通常都是头发蓬乱的嬉皮士和准嬉皮士们。摆弄操作系统的乐趣对他们来说不仅意味着可以在计算机科学的前沿上纵情挥洒，而且在于可以去推翻伴随“大计算”的所有技术假定和商业实践：卡式打孔机、COBOL、商务套装、IBM 批处理大型机都成了看不上眼的过时事物；Unix 黑客们沉浸在同时编织未来和编写系统的狂欢中。

那些日子的兴奋从 Douglas Comer 的话语中可见一斑：“许多大学都对 Unix 作出过贡献。多伦多大学计算机系发明了 200dpi 的打印机，绘图仪，并且开发了用打印机模拟照相排版机的软件；耶鲁大学的计算机专家和学生们改进了 Unix 的 shell；普渡大学的电子工程系对 Unix 的性能作了重要改进，推出了支持大量用户的 Unix 版本：普渡大学还开发出了最早的 Unix 计算机网络之一；加州大学伯克利分校的学生开发了新 shell 和许多小型实用工具。1970 年代后期贝尔实验室发布 Unix V7 版本时，很显然，该系统解决了许多部门的运算问题，也综合了许多高校的创意。最终诞生了一个更强大的系统。思想潮流开始了新一轮循环，从学术界流向工业实验室，然后又回到学术界，最后流向了不断增加的商业用户。”[Comer]

现代 Unix 程序员公认的第一个完全意义上的 Unix 是 1979 年发布的 V7 版本[4]。第一代 Unix 用户群一年前就已形成。此时，Unix 用于支撑贝尔系统（Bell System）所有操作 [Hauben]，并且传播到高校中，甚至远至澳大利亚——在那里，John Lions 对 V6 版源码的注释 [Lions] 成了 Unix 内核的第一个正式文档。许多资深的 Unix 黑客仍然珍藏着一份拷贝。

> Lions 的书是地下出版界轰动一时的大事。由于侵犯版权等诸如此类的问题，该书不能在美国出版，所以大家就你拷给我、我拷给你。我也有一份拷贝，至少是第六手了。在那个时代，若没有 Lions 的书，你就当不成内核黑客。
>
> —Ken Arnold

Unix 产业也初露端倪。1978 年，第一个 Unix 公司（the Santa Cruz Operation，SCO）成立，同年售出第一个商用 C 编译器（Whitesmiths）。1980 年，西雅图一家还不起眼的软件公司——微

4

软也加入到 Unix 游戏中，他们把 AT&T 版本移植到微机上，取名为 XENIX 来销售。但是微软把 Unix 作为一个产品的热情并没有持续多久（尽管直到 1990 年左右，微软的大部分内部开发工作都用的是 Unix）。

TCP/IP 和 Unix 内战：1980-1990

在 Unix 的发展过程中，加州大学伯克利分校很早就成为唯一最重要的学术热点。伯克利分校早在 1974 年就开始了对 Unix 的研究，而 Ken Thompson 利用 1975—1976 的年休在此教学，更对 Unix 的研究注入了强劲活力。1977 年，当时还默默无闻的伯克利毕业生 Bill Joy 管理的实验室发布了第一版 BSD。到 1980 年，伯克利分校成了为这个 Unix 变种积极作贡献的高校子网的核心。有关伯克利 Unix（包括 *vi*(1) 编辑器）的创意和代码不断从伯克利反馈到贝尔实验室。

1980 年，国防部高级研究计划局（DARPA，Defense Advanced Research Projects Agency）需要请人在 Unix 环境下的 VAX 机上实现全新的 TCP/IP 协议栈。那时，运行 ARPANET 的 PDP-10 已处耄耋之年，而数据设备公司（DEC）可能被迫放弃 PDP-10 以支持 VAX 的种种迹象也空穴来风。DARPA 曾考虑和 DEC 公司签订实现 TCP/IP 的合同，但是因为担心 DEC 可能不太乐意改动他们的专有 VAX/VMS 操作系统 [Libes—Ressler] 而打消了这个念头。最后，DARPA 选择了伯克利 Unix 作为平台——显然因为可以毫无阻碍地拿到它的源 [Leonard]。

伯克利计算机科学研究组当时拥有天时地利，还有最强大的开发工具；而 DARPA 的合同无疑成为 Unix 历史上自诞生以来最关键的转折点。

在 1983 年 TCP/IP 实现随 Berkeley4.2 版发布之前，Unix 对网络的支持一直是最薄弱的。早期的以太网实验不尽人意。贝

尔实验室开发了一个难看但还能用的工具 UUCP（Unix to Unix Copy Program），可在普通电话线上通过调制解调器来传送软件。[5]UUCP 可以在分布很广的机器之间转发邮件，并且（在 1981 年 Usenet 发明后）支持 Usenet——一个分布式的电子公告牌系统，允许用户把文本信息传播到任何拥有电话线和 Unix 系统的机器上。

尽管如此，已经意识到 ARPANET 光明前景的少数 Unix 用户感觉自己似乎陷在一潭死水中。没有 FTP，没有 telnet，只有限制重重的远程作业执行和慢得要死的连接。在 TCP/IP 诞生之前，Unix 和 Intemet 文化尚未融合。Dennis Ritchie 将计算机视为“鼓励密切交流”的工具这一设想还只是围绕单机分时系统或同一计算中心的学术社群，并没有扩展到自 1970 年代中期开始 ARPA 用户群逐渐形成的一个分布全美的“网络国家”。早期 ARPANET 的用户对着自己蹩脚的硬件时，也只能想：凑合着用 Unix 吧。

有了 TCP/IP，一切都变了。ARPANET 和 Unix 文化自边缘开始融合，这种发展最终使两者都免遭灭亡。不过，首先还得经过炼狱，起因是两个毫不相干的灾难：微软的兴起和 AT&T 的拆分。

1981 年，微软同 IBM 就新型 IBM PC 达成了历史性交易。比尔 • 盖茨从西雅图计算机产品公司（SCP，Seattle Computer Products）买下了 QDOS（Quick and Dirty Operating System）。QDOS 是 SCP 公司的 Tim Paterson 花六个星期凑出来的 CP/M 翻版。盖茨对 Paterson 和 SCP 公司隐瞒了同 IBM 的交易，以五万美元的价格买下了所有版权。后来，盖茨又说服了 IBM 公司允许微软将 MS-DOS 从硬件中剥离出来单独出售。接下来的十年中，盖茨利用这个非他所写的程序变成了超级亿万富翁，而比首笔交易更加精明的商业策略更是让微软垄断了桌面计算机市场。作为产品的

---

5 当时，如果调制解调器的速度能达到 300 波特时，UUCP 跑得还是不错的。

XENIX 很快就弃而不用了，最终卖给了 SCO 公司。

那时，没什么人能看出微软会多么成功（或有多大破坏性）。因为 IBM PC-1 硬件条件不足以来运行 Unix，所以 Unix 人群几乎没注意这个产品（尽管，具有讽刺意味的是，DOS 2.0 光芒能盖过 CP/M，主要因为微软的合创者 Paul Allen 在 DOS 2.0 中融入了一些 Unix 的特征，包括子目录和管道等）。还有更有趣的事呢——比如说 1982 年 SUN 微系统公司的出世。

SUN 微系统公司的创立者 Bill Joy、Andreas Bechtolsheim 和 Vinod Khosla 打算制造出一种内置网络功能的 Unix 梦幻机器。他们综合了斯坦福大学设计的硬件和伯克利分校开发的 Unix，取得了辉煌的成功，开创了工作站产业。随着 Sun 公司越来越像传统商家而不再像一个无拘无束的新公司时，Unix 大树上的这根分支源码来源的树枝逐渐枯萎，然而当时并没有人在意这一点。伯克利分校仍然随同源码一起销售 BSD。一份 System III 源码许可证的官方价格为 4 万美元：贝尔实验室对非法流传贝尔 Unix 源码磁带的行为睁只眼闭只眼，各个高校也依然同贝尔实验室交换代码，看起来 Sun 公司对 Unix 的商业化似乎对它再好不过了。

C 语言也在 1982 年有望被选为 Unix 世界外的系统编程语言。仅仅只用了五年左右的时间，C 语言就几乎让机器码汇编语言完全失去了作用。到了九十年代早期，C 和 C++ 不仅统治了系统编程领域，而且成为应用编程的主流。到九十年代晚期，其他所有传统编译语言实际上都已经过时了。

1983 年，在 DEC 公司取消 PDP-10 的后继机型的“木星”（Jupiter）开发计划后，运行 Unix 的 VAX 机器开始代之成为主流的互联网机器，直到被 Sun 工作站取代。到 1985 年，尽管 DEC 极力抵抗，还是有 25% 左右的 VAX 用上了 Unix。但是取消木星计划的长期效应并明显。更主要的是，MIT 人工智能实验室以 PDP-10 为中心的黑客文化的消亡激发了 Richard StallMan 开始编制 GNU ——一个完全自由的 Unix 克隆版本。

到 1983 年，IBM PC 可使用不下六种的 Unix 通用操作系统：uNETix、Venix、Coherent、QNX、Idris 和运行在 Snitek PC 子板上的移植版本。但是 System V 和 BSD 版本仍然没有 Unix 移植——两个群体都悲观地认为 8086 微处理器不够强大，根本就没打算这么做。IBM PC 上的这些 Unix 通用操作系统无一取得显著的商业成功，但表明了市场迫切需求运行 Unix 的低价硬件，而主要厂商并不供应。个人用户谁也买不起，更何况源码许可证上还挂着 4 万美元的价签呢。

1983 年，美国司法部在针对 AT&T 的第二起反托拉斯诉讼中获胜，并拆分了贝尔系统。这时 Sun 公司（及其效仿者！）已经取得了成功。这次判决将 AT&T 从 1958 年的禁止将 Unix 产品化的和解协议中解脱了出来。AT&T 马上忙不迭地将 Unix System V 商业化——这一举措差点扼杀了 Unix。

> 确实如此。但他们的营销策略却将 Unix 推向了全球。
>
> —Ken Thompson

大多数 Unix 支持者都认为 AT&T 的拆分是个好消息。我们原以为，在拆分后的 AT&T、Sun 公司及效仿 Sun 的小公司中，我们看到了一个健康的 Unix 产业核心——利用基于低廉的 68000 芯片的工作站——能够挑战并最终打破压迫在计算机行业上的垄断者——IBM。

那时，没有人意识到，Unix 的产业化会破坏 Unix 源码的自由交流，而恰是后者滋养了 Unix 系统早期的活力。AT&T 只知道用保密从软件中获利，只会用集中控制模式开发商业产品，对源码散发严加防护。因为唯恐官司上身，非法交易的 Unix 源码也越来越乏人问津。来自高校的贡献随之开始枯竭。

更糟的是：刚刚进入 Unix 市场的几家大公司立马犯下了重大的战略性错误，其中之一就是试图通过产品差异化来寻求有利地位

——这个策略导致了各种 Unix 接口的分歧，它抛弃了 Unix 的跨平台兼容性，造成了 Unix 市场分割。

另一个更微妙的错误就是以为个人计算机和微软不关 Unix 前景的事。Sun 微系统公司未能意识到，日用品化的个人机最终会无可避免地动摇其工作站市场的根基。AT&T 公司为了成为计算机行业执牛耳者[6]，针对小型机和大型机采取了不同的策略，结果两个摊子都砸了。几家小公司试图在 PC 机上支持 Unix，但都资金不足，仅专注于将产品出售给开发者和工程师，从未关注微软所瞄准的商用和家庭市场。

事实上，AT&T 拆分后的数年内，Unix 社区却在忙着 Unix 大战的第一阶段——System V Unix 和 BSD Unix 之间的内部争吵。争吵分成不同的层面，有些属于技术层面（socket 对 stream，BSD tty 对 System V termio），有些则属于文化层面。分歧可以大致划分为长发派和短发派。程序员和技术人员往往与伯克利和 BSD 站在一边，而以商业为目标的人则倾向 AT&T 和 System V。长发派，重唱着十年前 Unix 早期的主题，喜欢自我标榜为企业帝国的叛逆者，比如一家小公司贴的海报那样，上面画着一个标着“BSD”的 X 翼星际战机快速飞离巨大的 AT&T 死星，后者在熊熊烈火中粉身碎骨。就这样，罗马在燃烧，而我们还在拉小提琴。

但是，AT&T 拆分当年发生的另一件事对 Unix 产生了更深远的影响。程序员兼语言学家 Larry Wall 发明了 patch (1) 实用程序。Patch 程序是一个将 diff(1) 生成的修改记录（changebar）写入基础文件的简单工具，这意味着 Unix 开发人员之间可通过传送补丁——代码的渐增变化——进行协作，而不必传送整个代码文件。这一点非常重要，不仅因为补丁要比整个文件小，更因为即使基础文件和补丁制作者拿到的版本之间变化很大，仍然可以很干净地应用补丁。运用这个工具，基于共有源码库的开发流可以分开、并行、

---

6 古代歃血为盟，盟主执牛耳。

最后合拢。patch 程序比其它任何单一工具都更能促进 Internet 上的协作开发——这种方式在 1990 年后让 Unix 获得新生。

1985 年，Intel 的第一枚 386 芯片下线。它具有用平面地址空间寻址 4G 内存的能力。笨拙的 8086 和 286 的段寻址旋即废弃。这是条大新闻，因为这意味着占据主导地位的 Intel 家族终于有了一款无需作出痛苦妥协就能运行 Unix 的微处理器。对 Sun 公司和其它工作站厂商来说，这真是不祥之兆，可惜它们并未觉察到。

同样在 1985 年，Richard Stallman 发表了 GNU 宣言（the GNU manifesto）[Stallman]，并发起了自由软件基金会（Free Software Foundation）。没有谁把他和他的 GNU 当回事，结果证明这是个大错误。同年，在一项与此不相干的开发行动中，X window 系统的创始人发布了 X window 的源码，而无需版税、约束和授权。这项决策的直接结果就是 X window 成为不同 Unix 厂商之间合作的安全中立区，并挫败了专属的竞争对手，成为了 Unix 的图形引擎。

以调解 System V 和 Berkeley API 为目标的严肃的标准化工作始于 1983 年，产生了/usr/group 标准。随之为 1985 年 IEEE 支持的 POSIX 标准。这些标准描述了 BSD 和 SVR3（System V Release 3）调用的交集，综合了伯克利出色的信号处理和作业控制，以及 SVR3 的终端控制。所有后续的 Unix 标准其核心都加入了 POSIX，后续开发的各种 Unix 版本也严格遵循这个标准。后来的现代 Unix 核心 API 唯一主要的补充就是 BSD 套接字。

1986 年，前面提到的发明 patch(1) 的 Larry Wall 开始开发 Perl 语言，后者是最先也最广泛使用的开源脚本语言。1987 年年初，GNU C 编译器的第一版问世，到 1987 年年底，GNU 工具包的核心部分——编辑器、编译器、调试器以及其它基本的开发工具——都已就位。同时，X window 系统也开始在相对低廉的工作站上露面了。这些因素都为 20 世纪 90 年代的 Unix 开源发展提供了利器。

同样是在 1986 年，PC 技术挣脱了 IBM 的掌控。IBM 仍然试

图在产品系列上维持高价格性能比，更青睐高利润的大型机市场，所以在新的 PS/2 系列产品上拒用 386 而选择了较弱的 286。PS/2 系列为了杜绝仿冒而围绕一个专有总线结构进行设计，结果成了代价高昂的大败笔[7]。最积极进取的效仿者康柏（Compaq），发布了第一款 386 机器，靠这张牌打败了 IBM。虽然主频只有 16MHz，但是 386 也算能跑起来 Unix 了。这是第一款可以叫 Unix 机器的 PC。

这会儿已经能够想象 Stallman 的 GNU 项目可以和 386 机器配合而制造出 Unix 工作站，它比当时任何方案都要便宜一个数量级。奇怪的是，没人想到这步棋。来自小型机和工作站世界的大多数 Unix 裎序员，依然鄙视廉价的 80x86 芯片，而钟情基于 68000 的高雅设计。尽管许多程序员都为 GNU 工程做出了贡献，但在 Unix 人群中，这个 GNU 项目仍然被视为一个唐吉诃德式的狂想，短期内还无法实用。

Unix 社区从未丢弃叛逆气质。但是回头看来，我们几乎和 IBM 或者 AT&T 一样，对迫近我们的未来毫无所知。即使是数年前就开始对专有软件开展精神讨伐的 Richard Stallman 也未能真正理解 Unix 的产品化会对其所在社区有多大破坏力；他关心的是更抽象的长期论题。其余的人还一直企盼企业规则能有些精明的变化，从此市场分割、营销不利和战略漂忽不定等问题将不复存在，从而救赎回 Unix 拆分之前的世界。但是祸不单行。

很多人都知道 Ken Olsen（DEC 的 CEO）在 1988 年将 Unix 描绘成“蛇油”（骗人的万灵油）。从 1982 年起，DEC 就一直在销售其开发的用于 PDP-11 的 Unix 变种，但真正希望的却是将业务回到自己专有的 VMS 操作系统上来。DEC 和其它小型机厂商碰到了大麻烦，陷入 Sun 微系统公司和其它工作站厂商功能强劲、价格低

[7] PS/2 毕竟还是在后来的 PC 机上留了一记——使鼠标成为标准外设，这也是为什么你机箱后面的鼠标接口会叫做“PS/2 端口”

廉的机器重重包围中。这些工作站大多运行的是 Unix。

但是 Unix 产业自身的问题却更为严峻。1988 年，AT&T 持有了 Sun 公司 20% 的股份。作为 Unix 市场领军的这两家公司，终于开始清醒地认识到 PC，IBM 和微软构成的威胁，也终于认识到过去五年的争斗令他们几无所获。AT&T 和 Sun 的联盟以及以 POSIX 为核心的技术标准的发展，最终弥合了 System V 和 BSD Unix 之间的裂痕。但是，当二线商家（IBM、DEC、HP 等）创建开放软件基金会（Open Software Foundation）并结成盟友和以“Unix 国际”为代表的“AT&T/Sun 轴心”对抗时，Unix 内战的第二阶段开始了。更多回合的 Unix 与 Unix 三家的战斗随而爆发。

这段时间中，微软从家庭和小型商用市场赚了数十亿美元的钱，而争战不休的 Unix 各方却从未决意涉足这些市场。1990 年，Windows 3.0——来自微软总部 Redmond 发布的第一个成功的图形操作系统——巩固了微软的统治地位，为微软在九十年代荡平并最终垄断桌面应用市场创造了条件。

1989 年到 1993 年是 Unix 的中世纪。当时，似乎 Unix 社群所有的梦想都破灭了。相互争斗的战事已使专有 Unix 产业衰落得像个吵闹的肉店，无力振起挑战微软的雄心。大多数 Unix 编程者青睐的优雅的 Motorola 芯片也已经输给了 Intel 丑陋但廉价的处理器。GNU 项目没能开发出自由的 Unix 内核，尽管从 1985 年 GNU 就不断作出此承诺，其信用令人质疑。PC 技术被无情地商业化了。1970 年代的 Unix 黑客先锋们人近中年，步履开始蹒跚。硬件便宜了，但 Unix 还是太贵。我们幡然醒悟：过去的 IBM 垄断让位于现在的微软垄断，而微软设计糟糕的软件像浊流一样，围着我们越涨越高。

反击帝国：1991-1995

1990 年，William Jolitz 把 BSD 移植到了 386 机器上，这是黑暗中的第一缕曙光。1991 年起一系列杂志文章对此进行了报道。向 386 移植 BSD 的移植之所以可能，是由于伯克利黑客 Keith Bostic 一定程度上受 Stallman 影响，早在 1988 年他就开始努力从 BSD 码中清除 AT&T 专有代码。但是，Jolitz 在 1991 年年底退出 386-BSD 项目，并毁掉了自己的成果，使该项目受到严重打击。对于此事的起因众说纷纭，不过公认的一点是 Jolitz 希望将其代码以源码形式无限制地发布，因此当项目的企业赞助商选择了更专有的授权模式时，他火了。

1991 年 8 月，当时默默无闻的芬兰大学生 Linus Torvalds 宣布了 Linux 项目。据称 Torvalds 最主要的激励是学校里用的 Sun Unix 太贵了。Torvalds 还说，要是早知道有 BSD 项目，他就会加入 BSD 组而不是自己做一个。但是 386BSD 直到 1992 年早些时候才下线，而此时 Linux 第一版已经发布好几个月了。

不回头看，人们无法发现这两个项目的重要性。那时，即使在 Intemet 黑客文化内部也没有多少人关注它们，遑论更广大的 Unix 社区。当时 Unix 社区还在盯着比 PC 机性能更强大的机器，仍试图把 Unix 的特有品质与软件业的常规专有模式扯到一起。

又过了两年，经历了 1993—1994 年的互联网大爆炸，Linux 和开源 BSD 的真正重要性才为整个 Unix 世界所了解。但不幸的是对 BSD 支持者来说，AT&T 对 BSDI（赞助 Jolitz 移植的创业公司）的诉讼消耗了大量时间，使一些关键的 Berkeley 开发者转向了 Linux。

> 代码抄袭和窃取商业秘密的行为从未被证实。他们花了两年的时间也没找到确凿的侵权代码。要不是 Novell 从 AT&T 买下了 USL、并达成协议，这场官司还会拖得更久。结果是从发布包中 18000 个组成文件中删掉了三个，对其它文件作了一

> 些小修改。另外，伯克利大学也同意为约 70 个文件增加 USL 版权，但同时约定这些文件仍然可以自由重新分发。
>
> —Marshall Kirk McKusick

这项和解为开创了从专有控制下获取一个自由而完整可用的 Unix 的先河，但对 BSD 自身的影响却是灾难性的。当伯克利的计算机科学研究组于 1992—1994 年间被关闭时，情况更糟了；随后，BSD 社区内的派系斗争又将 BSD 开发分割成三个方向间的竞争。结果，BSD 这一脉在关键时刻落后于 Linux，Unix 社区的领先地位拱手让人。

与此前各种版本的 Unix 开发相比，Linux 和 BSD 的开发相当不同。它们植根于互联网，依赖分布式开发和 Larry Wall 的 patch(1) 工具，通过 email 和 Usenet 新闻组招募开发者。因此，当互联网服务提供商（ISP）的业务于 1993 年因通信技术的变革和 Internet 骨干网的私有化（超出 Unix 历史范围，不述）而扩展时，Linux 和 BSD 也得到了巨大的推动力。但对廉价互联网的需求却是由另一件事创造的：1991 年万维网（WWW）的发明。万维网是互联网中的“杀手级应用”，图形用户界面技术对大量的非技术型最终用户有着不可抗拒的魅力。

互联网的大规模市场推广，既增加了潜在开发者的数量，又降低了分布式开发的处理成本，这些影响可从 XFree86 之类的项目上看出。XFree86 利用 Internet 为中心的模式建立了一个比官方 X 联盟更有效的开发组织。1992 年诞生的第一版 XFree86 赋予了 Linux 和 BSD 一直缺乏的图形用户界面引擎。下个十年里，XFree86 将领导 X 的开发，X 联盟越来越多的行为都是把源自 XFree86 社区的创新汇聚回 X 联盟产业赞助者中。

到 1993 年年末，Linux 已经具备了 Internet 能力和 X 系统。整套 GNU 工具包从一开始就内置其中，以提供高质量的开发工具。除了 GNU 工具，Linux 好像一个魅力聚宝盆，囊括了二十年来分散在十几种专有 Unix 平台上的开源软件之精华。尽管正式说来 Linux

内核还是测试版（0.99 的水平），但稳定性已经让人刮目相看。Linux 上软件之多、质量之高，已经达到一个产品级操作系统的水准。

在旧学派的 Unix 开发者中，一部分脑筋活络的人开始注意到，做了多年的平价 Unix 之梦从一个意想不到的方向悄然成真。它既不是来自 AT&T，也不是来自 Sun，或者任何一个传统厂商，也不是出于学术界有组织的工作成果。它就这样从 Internet 的石头缝中跳了出来，浑然天成，以令人惊奇的方式重新规划拼装了 Unix 的传统元素。

另一方面，商业运作继续进行。1992 年 AT&T 抛售了其手中 Sun 公司的股份，然后在 1993 年把 Unix 系统实验室（Unix Systems Laboratories）卖给了 Novell；Novell 又于 1994 年将 Unix 商标转手给 X/Open 标准组（X/open standards group）；同年 AT&T 和 Novell 加入了 OSF（开放软件基金会），Unix 之战尘埃落定。1995 年，SCO 从 Novell 手中买下了 UnixWare（以及最初 Unix 源码的权利）。1996 年，X/Open 和 OSF 合并，创立了一个大型 Unix 标准组。

但是，传统 Unix 厂商和他们战后的烂摊子看来确是越来越无关紧要了。Unix 社区的动作和精力都在转向 Linux、BSD 及开源开发者。1998 年，IBM、Intel 和 SCO 宣布启动蒙特里项目（the Moterey project），最后一次努力试图将所有现存的专有 Unix 整合成一个大系统，开发者和业内媒体坐看笑话。原地兜了三年的圈之后，此项目在 2001 年戛然而止。

2000 年 SCO 把 UnixWare 和原创的 Unix 源码包出售给了 Caldera——一家 Linux 发行商，整个产业变迁终告结束。但 1995 年后，Unix 的故事就成了开源运动的故事。故事还有一半没讲呢，我们要回到 1961 年，从互联网黑客文化的起源开始讲起。

黑客的起源和历史：1961-1995

Unix 传统是一种隐性的文化，不只是一书袋的技术窍门。这种传统传达着一个有关美和优秀设计的价值体系；里面有它的江湖和侠客。与 Unix 传统的历史交织在一起的则是另一种隐性文化，一种更难归别的文化。它也有自己的价值体系、江湖和侠客，部分与 Unix 文化交迭，部分源于它处。人们老是把这种文化称为“黑客文化”，从 1998 年起，这种文化已经很大程度上和计算机行业出版界所称的“开源运动”重合了。

Unix 传统、黑客文化以及开源运动间的关系微妙而复杂。三种隐性文化背后往往是同一群人，然而其间的关系并未因此而简化。但是，从 1990 年以来，Unix 的故事很大程度上成了开源世界的黑客们改变规则、从保守的专有 Unix 厂商手中夺取主动权的故事。因此，今天 Unix 身后的历史，有一半就是黑客的历史。

游戏在校园的林间：1961-1980

黑客文化的根源可以追溯到 1961 年，这一年 MIT 购买了第一台 PDP-1 小型机。PDP-1 是最早的一种交互式计算机，并且（不象其它机器）在那时并非天价，所以没有对它的使用做太多限时规定。因此 PDP-1 吸引了一帮好奇的学生。他们来自技术模型铁路俱乐部（TMRC，Tech Model Railroad Club），带着一种好玩的心态摆弄这台设备。《黑客：计算机革命中的英雄》（Hackers: Heroes of the Computer Revolution）[Levy] 一书对这个俱乐部的早期情况作了有趣的描写。他们最著名的成就是“太空大战（SPACEWAR）”——一款宇宙飞船决斗游戏，灵感大概来自 Lensman 的星际故事《E. E. ‘Doc’ Smith》。[8]

[8] “SPACEWAR”和 Ken Thompson 的“Space Travel”毫不相干，除了都吸引科幻迷的共同点。

TMRC 来实验的几个人后来是成了 MIT 人工智能实验室的核心成员，而这个实验室在六七十年代成为前沿计算机科学的世界级中心之一。这些人也把 TMRC 的行话和内部笑话带了进来，包括一种精巧（但无害）的恶作剧传统“hacks”。人工智能实验室的程序员应该是第一群自称“hacker” 的人。

1969 年后，MIT AI 实验室和斯坦福、Bolt Beranek & Newman 公司（BBN）、卡内基 -梅隆大学（CMU：Carnegie-Mellon University）以及其它顶级计算机科学研究实验室通过早期的 ARPANET 联上了网。研究人员和学生第一次尝到了快速网络联接消除了地域限制的甜头，通过网络，远方的人通常比与身边少有来往的同事更容易合作和建立友谊。

实验性的 ARPANET 网上到处都是软件、点子、行话和大量幽默。一种类似共享文化的东西开始成形，其中最早、最持久的典型产物之一就是“术语文件（Jargon File）”，列举了 1973 年发源于斯坦福、1976 年后在 MIT 经过多次修订的共享行内名词，并一路收集了 CMU、耶鲁和其它 ARPANET 站点的行话。

从技术性而言，早期的黑客文化大都基于 PDP-10 小型机。下列已经成为历史的操作系统他们都用过：TOPS-10、TOPS-20、Multics、ITS 和 SAIL。他们利用汇编器和各种 Lisp 方言编程。PDP-10 的黑客们后来接手运行 ARIPANET，因为别人不愿意干这件事。后来，他们成了互联网工程工作组（IETF，Intemet Engineering Task Force）的创建骨干，并作为创始人，开创了通过 RFC（Requests For Comment）进行标准化的传统。

从社会性而言，他们年轻，天资过人，几乎全是男性，献身编程达到痴迷的地步，决不墨守成规——后来被人们唤做“极客（geek）”。他们往往也是头发蓬松的嬉皮士和准嬉皮士。他们有远见，把计算机看作构建社区的工具。他们读 Robert Heinlein 和 J. R. R. Tolkien 的书，参加复古协会（Society for Creative Anachronism），双关语说起来没完。抛开这些怪癖（也许正由于这些原

因），他们中的许多人都跻身世界上最聪明的程序员之列。

他们并不是 Unix 程序员。早期的 Unix 社群成员大部分来自院校、政府和商业研究实验室的同一帮“极客”，但是两种文化有明显的分野。其中之一就是我们前面已经谈到的早期 Unix 孱弱的网络能力。直到 1980 年后，才真正出现了基于 Unix 的 ARPANET 网络连接，之前一个人同时涉足两个阵营的情况并不多见。

协作式开发和源码共享是 Unix 程序员的法宝。然而，对于早期的 ARRPNET 黑客，这还不只是一种策略，它更像一种公众信仰，部分起源于“要么发表要么掉”的学术规则，并且（更极端地）几乎发展成为关于网络思想社区的夏尔丹式理想主义（Chardinist idealism）。这些黑客中最著名的 Richard M. Stallman 后来成了严守教义的苦行僧。

互联网大触合与自由软件运动：1981-1991

1983 年后，随着 BSD 植入了 TCP/IP，Unix 文化和 ARPANET 文化开始融合。既然两种文化都由同一类人（实际上，就有少数几位很有影响的人同属两种文化阵营）构成，一旦沟通环节到位，两种文化的融合就水到渠成。ARPANET 黑客学到了 C 语言，用起了管道、过滤器和 shell 之类的行话。Unix 程序员学到了 TCP/IP，也开始互称“黑客”。1983 年，木星项目的取消虽然葬送了 PDP-10 的前途，却加速了两种文化融合的进程。到 1987 年，这两种文化已经完全融合在一起，绝大多数黑客都用 C 编程，自如地使用源于 25 年前技术模型铁路俱乐部（TMRC）创造的行话。

在 1979 年，我和 Unix 文化、ARPANET 文化都有密切联系，当时这种情况还很少见。到 1985 年，这就已经不稀奇了。1991 年我将以前的 ARPANET“术语文件”（Jargon File）扩展成《新黑客词典》（New Hacker’s Dictionary）[Raymond96]，此时两种文化实际上已经融为一体。把生于 ARPANET、长于 Usenet 的“术语文

件”作为这次融合的标志真是再恰当不过了。）

但是 TCP/IP 联网和行话并不是后 1980 黑客文化从其 ARPANET 根源继承的全部东西，还有 Richard M. StallMan 和他的精神革命。

Richard M. Stallman（他的登陆名 RMS 更为人熟知）早在 1970 年代晚期就已经证明他是当时最有能力的程序员之一。Emacs 编辑器就是他众多发明中的一项。对 RMS 来说，1983 年木星（Jupiter）项目的取消仅仅只是宣告了麻省理工学院人工智能实验室（MIT AI Lab）文化的最终解体。其实早在几年前随着实验室众多最优秀的成员纷纷离去，帮忙管理与之竞争的 Lisp 机器时，这种解体就已经开始了。RMS 觉得自己被逐出了黑客的伊甸园，他把这一切都归咎于专有软件。

1983 年，Stallman 创建了 GNU 项目，致力于编一个完全自由的操作系统。尽管 Stallman 既不是、也从来没有成为一个 Unix 程序员，但在后 1980 的大环境下，实现一个仿 Unix 操作系统成了他追求的明确战略目标。RMS 早期的捐助者大都是新踏入 Unix 土地的老牌 ARPANET 黑客，他们对代码共享的使命感甚至比那些有更多 Unix 背景的人强烈。

1985 年，RMS 发表了 GNU 宣言（the GNU Manifesto）。在宣言中，他有意从 1980 年之前的 ARPANET 黑客文化价值中创造出一种意识形态——包括前所未见的政治伦理主张、自成体系而极具特色的论述以及激进的改革计划。RMS 的目标是将后 1980 的松散黑客社群变成一台有组织的社会化机器以达到一个单纯的革命目标。也许他未意识到，他的言行与当年卡尔 • 马克思号召产业无产阶级反抗工作的努力如出一辙。[9]

---

[9] 请注意作者的立场是偏向 Torvalds 的，所以这里类比马克思多少有点暗黑的意思。读者请自己判断。

RMS 宣言引发的争论至今仍存于黑客文化中。他的纲要远不止于维护一个代码库，已经暗含了废除软件知识产权主张的精髓。为了追求这个目标，RMS 将“自由软件（free software）”这一术语大众化，这是将整个黑客文化的产品进行标识的首次尝试。他撰写了“通用公共许可证（General Public License，GPL）”，后者成了一个既充满号召力又颇具争议的焦点，具体原因我们将在 16 章研讨。读者可以去 GNU 站点http://www.gnu.org了解 RMS 立场及自由软件基金会（Free Software Foundation）的更多情况。

“自由软件（free software）”这个术语既是一种描述，也是为黑客进行文化标识的一个尝试。从某个层次上说，这是相当成功的。在 RMS 之前，黑客文化中的人们彼此当作“同路人”，说着同样的行话，但没人费神去争辩“黑客”是什么或者应该是什么。在他之后，黑客文化更加有自我意识。价值冲突（即使反对 RMS 的人也经常以他的方式说话）成为辩论中的常见特点。RMS，这个魅力超凡又颇具争议的人物本身已经成为了一个文化英雄，因此到 2000 年时，人们已经很难将他本人和他的传奇区分开来。《自由中的自由》（*Free as in Freedom*）[Williams] 对他的刻画非常精彩。

RMS 的论点甚至影响了那些对其理论持怀疑态度的黑客的行为。1987 年，他说服了 BSD Unix 的管理者，让他们相信，将 AT&T 的专有代码清除出去、发布一个无限制的版本是个好主意。然而，尽管他花了不下十五年的苦功夫，后 1980 黑客文化却从未统一在他的理想之下。

其他黑客，更多出于实用角度而非思想观念的原因，重新认识到了开放式协作开发的价值。在八十年代后期离 Richard Stallman 位于 MIT 九楼办公室不远的几座楼里，X 开发组搞得红红火火。这个项目由一些 Unix 厂商资助，这些厂商此前一直为 X window 系统的控制权和知识产权争论不休，结果发现还不如向所有人自由开放。1987 至 1988 年间，X 的开发预示了一个极为庞大的分布式社

群，后者将在五年后重新定义 Unix 的前沿方向。

> X 是首批由服务于全球各地不同组织的许多个人以团队形式开发的大规模开源项目之一。电子邮件使创意得以在这个群体中快速传播，问题由此得以快速解决，而开发者可以人尽其才。软件更新可以在数小时之内发送到位，使得每个节点在整个开发过程中步调一致。网络改变了软件的开发模式。
>
> —Keith Packard

X 开发者们不替 GNU 总计划帮腔，但也不唱反调。1995 年以前，GNU 计划最强烈的反对者是 BSD 开发者。BSD 开发者觉得自己编写自由发布和修改软件的年头比 RMS 宣言长得多，坚决抵制 GNU 自称的在历史性和思想性上的首创。他们尤其反对 GPL 的传染性或“病毒般”的特性，坚持 BSD 许可证比 GPL“更自由”，因为 BSD 对代码重用的限制要比 GPL 少。

尽管 RMS 的自由软件基金会已开发了整套软件工具包的绝大部分，但是未能开发出核心部件，因此形势对 RMS 仍然不利。GNU 项目创立十年了，GNU 内核仍是空中楼阁。尽管 Emacs 和 GCC 之类的单个工具被证明非常有用，但是没有内核的 GNU 既不能对专有 Unix 的霸权构成威胁，又不能有效抵抗日渐严重的微软垄断。

1995 年后，关于 RMS 思想体系的争论稍稍发生了变化。反对者的观点跟 Linus Torvalds 和本书作者越来越近。

Linux 和实用主义者的应对：1991-1998

即使在 HURD（GNU 内核）计划停转之时，新的希望还是出现了。1990 年代早期，价廉性优的 PC 机加上方便快捷的互联网，对寻找机会挑战自我的新生代年轻程序员是极大的诱惑。自由软件基金会编写的用户软件工具包铺平了一条摆脱高成本专有软件

开发工具的前进道路。意识服从经济，而不是领导：一些新手加入了 RMS 的革命运动，高举 GPL 大旗，另一些人则更认同整体意义上的 Unix 传统，加入了反对 GPL 的阵营，但其他大部分人置身事外，一心编码。

Linus Torvalds 巧妙地跨越了 GPL 和反 GPL 的派别之争。他利用 GNU 工具包搭起了自创的 Linux 内核，用 GPL 的传染性质保护它，但拒绝认同 RMS 许可协议反映的思想体系计划。Torvalds 明确表示他认为自由软件通常更好，但他偶尔也用专有软件。即使在他自己的事业中，他也拒绝成为狂热分子。这一点极大地吸引了大多数黑客，他们虽然早就反感 RMS 的言辞，但他们的怀疑论一直缺个有影响力或者令人信服的代言人。

Torvalds 令人愉快的实用主义及灵活而低调的行事风格，促使黑客文化在 1993 至 1997 年间取得了一连串令人惊奇的胜利，不仅仅在技术上的成功，还让围绕 Linux 操作系统的发行、服务和支持产业有了坚实的开端。结果，他的名望和影响也一飞冲天。Torvalds 成为了互联网时代的英雄：到 1995 年为止，他只用了四年时间就在整个黑客文化界声名显赫，而 RMS 为此花了十五年，而且他还远远超过了 Stallman 向外界贩卖“自由软件”的记录。与 Torvalds 相比，RMS 的言辞渐渐显得既刺耳又无力。

1991 至 1995 年间，Linux 从概念型的 0.1 版本内核原型，发展成为能够在性能和特性上均堪媲美专有 Unix 的操作系统，并且在连续正常工作时间等重要统计数据上打败了这些 Unix 中的绝大部分。1995 年，Linux 找到了自己的杀手级应用——开源的 web 服务器 Apache。就像 Linux，Apache 出众地稳定和高效。很快，运行 Apache 的 Linux 机器成了全球 ISP 平台的首选。约 60% 的网站选用 Apache，[10]轻松击败了另两个主要的专有型竞争对手。

---

[10] 当月和以往的 web 服务器占有量数据可从 Netcraft 的 web 服务器月度调查（monthly Netcraft Web Server Survey）中获得。

Torvalds 未作的一件事就是提供新的思想体系——一套关于黑客行为的新理论基础或繁衍神话，以及一套吸引黑客文化圈内圈外人士的正面论述，以消弭 RMS 对知识产权的不友善。1997 年，当我试图探寻为什么 Linux 开发没有在几年前崩溃时，我偶然地填补了这个空白。我所发表论文 [Raymond01] 的技术结论归纳在本书第 19 章。对于这段历史梗概，只要看看第一条结论核心规则的冲击就够了：“如果有足够多眼睛的关注，所有的 bug 都无处藏身”。

这段观察暗含了过去四分之一世纪在黑客文化中从未有人敢相信的东西：用这种方法做出的软件，不仅比我们专有竞争者的东西更优雅，而且更可靠、更好用。这个结果出乎意料地向“自由软件”的论述发起了直接挑战，而 Torvalds 本人从未有意于此。对于大多数黑客和几乎所有的非黑客而言，“用自由软件是因为它运行得更好”轻而易举地盖过了“用自由软件是因为所有软件都该是自由的”。

在我的论文中关于“大教堂”（集权、封闭、受控、保密）和“集市”（分权、公开、精细的同僚复审）两种开发模式的对比成为了新思潮的中心思想。从某种重要意义上来说，这仅仅是对 Unix 在拆分前根源的回归——Mcllroy 在 1991 年阐述了同侪压力如何对 1970 年代早期 Unix 的发展产生了积极影响、Dennis Rithchie 在 1979 年对伙伴关系的反思，这是此两者的延续，并与早期 ARPANET 同侪评审的学术传统及其分布式精神社区的理想主义相得益彰。

1998 年初，这种新思潮促使网景公司（Netscape Communications）公布了其 Mozilla 浏览器的源码。媒体对此事件的关注促成了 Linux 在华尔街的上市，推动了 1999-2001 年间科技股的繁荣。事实证明，此事无论对黑客文化的历史还是对 Unix 的历史都是一个转折点。

开源运动：1998 年及之后

到 1998 年 Mozilla 源码公布的时候，黑客社区其实算是一个众多派系或部落的松散集合，包括了 Richard Stallman 的自由软件运动（Free Software Movement）、Linux 社区、Perl 社区、Apache 社区、BSD 社区、X 开发者、互联网工程工作组（IETF），还有至少一打以上的其它组织。这些派系相互交叠，一个开发者很可能同时隶属两个或更多组织。

一个部落的凝聚力可能来自他们维护的代码库，或是一个或多个有着超凡影响力的领导者，或是一门语言、一个开发工具，或是一个特定的软件许可，或是一种技术标准，或是基础结构某个部分的管理组织。各派系既论资排辈，也追逐当前的市场份额及认知度。因此，资格最老的大概要算 IETF，其历史可以连续追根溯源到 1969 年 ARPANET 的发源期；BSD 社区尽管市场安装数量要比 Linux 少得多，但是因为其传统可连续追溯到 1970 年代末，所以还是拥有相当高的声望；可追溯到 1980 年代初的 Stallman 的自由软件运动，无论从历史贡献，还是从作为几个最常用的软件工具维护者的角度，都足以令其跻身高级部落行列。

1995 年后，Linux 扮演了一个特殊的角色：既是社区内多数软件的统一平台，又是黑客中最被认可的品牌。Linux 社区随之显现了兼并其它亚部落的倾向——甚至包括争取并吸纳一些专有 Unix 相关的黑客派系。整个黑客文化开始凝聚在一个共同目标周围：尽力推动 Linux 和集市开发模式向前发展。

因为后 1980 黑客文化已经深深植根于 Unix，新目标成了 Unix 传统争取胜利的不成文纲要。黑客社区的许多高级领导人也都是 Unix 的老前辈，八十年代分拆后内战的伤痕犹在，他们将 Linux 作为实现 Unix 早期叛逆梦想的最后和最大的希望，而汇聚在 Linux 旗下。

Mozilla 源码的公布使各方意见更为集中。1998 年 3 月，为了深入研究共同目标和策略，召开了一次空前的社团重要领导人峰会，与会者几乎代表了所有的主要部落。这次会议为所有派系的共同开发方式确立了一个新标记——开源。

六个月之内，黑客社区中几乎所有部落都接受了用“开源”的新旗帜。IETF 和 BSD 开发组这种老团体更是把他们从过去到现在所作的东西都追加上了这一标记。实际上，到 2000 年，黑客文化不仅让“开源”这个辞令统一了当前实践和未来计划，而且也对自己的历史重新有了鲜活的认识。

Netscape 开放源码的宣告和 Linux 的新近崛起产生的激励效应远远超越了 Unix 社区和黑客文化。从 1995 年开始，所有阻拦在微软 Windows 滚滚巨轮前的各种平台（MacOS；Amiga；OS/2；DOS；CP/M；较弱小的专有 Unix；各类大型机小型机和过时的微型机操作系统）的开发者团结到了 Sun 微系统公司的 Java 语言周围。许多不满微软的 Windows 开发者也加入了 Java 阵营，希望至少能够和微软保持名义上的独立。但是 Sun 公司运作 Java 的几个层面都（我们将在第 14 章予以讨论）既拙劣又排斥他人。许多 Java 开发者喜欢上了开源运动中的新生事物，于是就像此前跟随 Netscape 加入 Java 一样，又跟随它加入了 Linux 和开源运动。

开源行动的积极分子热烈欢迎来自各个领域的移民潮。老一辈 Unix 人也开始认同新移民的梦想：不能只是被动忍受微软的垄断，而是要从微软手中夺回关键市场。开源社区成员们合力争取主流世界的认同，开始乐于同大公司结盟——这些公司，随着微软的绳索勒得越来越紧，也越来越害怕对自己的业务失去控制。

唯一的例外是 Richard Stallman 和自由软件运动。“开源”明显要用一个意识形态中性的公众标签来取代 Stallman 钟爱的“自由软件”。新标签无论对于历史上一贯反对“自由软件”的 BSD 黑客之类的团体，还是对于不愿在 GPL 是非之争中表态的人均能接受。Stallman 尝试着接受这个术语，但随后又以其未能代表其思想

的核心为由而排斥它。从此，自由软件运动坚持同“开源”划清界限，这也许成了 2003 年黑客文化中最重大的政治分歧。

“开源”背后另一个（也是更重要的）意图是希望将黑客社区的方法以一种更亲和市场、更少对抗性的方式介绍给外部世界（尤其是主流商用市场）。幸运的是，在这方面，它取得了绝对成功——这也重新激起了人们对其根源——Unix 传统——的兴趣。

Unix 的历史教训

在 Unix 历史中，最大的规律就是：距开源越近就越繁荣。任何将 Unix 专有化的企图，只能陷入停滞和衰败。

回顾过去，我们早该认识到这一点。1984 年至今，我们浪费了十年时间才学到这个教训。如果我们日后不思悔改，可能还得大吃苦头。

虽然我们在软件设计这个重要但狭窄的领域比其他人聪明，但这不能使我们摆脱对技术与经济相互作用影响的茫然，而这些就发生在我们的眼皮底下。即使 Unix 社区中最具洞察力、最具远见卓识的思想家，他们的眼光终究有限。对今后的教训就是：过度依赖任何一种技术或者商业模式都是错误的——相反，保持软件及其设计传统的的灵活性才是长存之道。

另一个教训是：别和低价而灵活的方案较劲。或者，换句话说，低档的硬件只要数量足够，就能爬上性能曲线而最终获胜。经济学家 Clayton Christensen 称之为“破坏性技术”，他在《创新者窘境》（The Innovator’s Dilemma）[Christensen] 一书中以磁盘驱动器、蒸汽挖土机和摩托车为例阐明了这种现象的发生。当小型机取代大型机、工作站和服务器取代小型机以及日用 Intel 机器又取代工作站和服务器时，我们也看到了这种现象。开源运动获得成功正是由于软件的大众化。Unix 要繁荣，就必须继续采用吸纳低价而

灵活的方案的诀窍，而不是去反对它们。

最后，旧学派的 Unix 社区因采用了传统的公司组织、财务和市场等命令机制而最终未能实现“职业化”。只有痴迷的“极客”和具有创造力的怪人结成的反叛联盟才能把我们从愚蠢中拯救出来——他们接着教导我们，真正的专业和奉献精神，正是我们在屈服于世俗观念的“合理商业做法”之前的所作所为。

如何在 Unix 之外的软件技术领域借鉴这些经验教训，就作为一个简单的练习留给读者吧。

对比：Unix 哲学同其他哲学的比较

> 如果你不知道怎样表现得高人一等，找个 Unix 用户，让他做给你看。
>
> 呆伯通讯 3.0，1994 年
>
> —Scott Adams

操作系统的设计，在明显和微妙两方面，造就了该系统下软件开发的风格。本书大部分内容描绘了此两者之间的联系：Unix 操作系统设计，以及由此发展出的编程设计哲学。为了便于对照，我们不妨把经典的 Unix 方式和其它主要操作系统的设计和编程习俗作一番比较。

操作系统的风格元素

开始讨论特定的操作系统之前，我们需要一个组织框架，来了解操作系统的设计是如何对编程风格产生或健康或病态的影响。

总的来说，与不同操作系统相关的设计和编程风格可以追溯出三个源头：(a) 操作系统设计者的意图；(b) 成本和编程环境的限制对设计的均衡影响；(c) 文化随机漂移，传统无非就是先入为主。

即使我们承认每个操作系统社区中都存在文化随机漂移现象，那么去探究一下设计者的意图和成本及环境造成的局限也能揭示一些有趣的规律，帮助我们通过比对来更好地理解 Unix 风格。我们可以通过分析操作系统最重要的不同之处把这些规律明确化。

什么是操作系统的统一性理念

Unix 有几个统一性的理念或象征，并塑造了它的 API 及由此形成的开发风格。其中最重要的一点应当是“一切皆文件”模型及在此基础上建立的管道概念[1]。总的来说，任何特定操作系统的开发风格均受到系统设计者灌注其中的统一性理念的强烈影响——由系统工具和 API 塑造的模型将反渗到应用编程中。

相应地，将 Unix 和其他操作系统作比较时，最基本的问题是：这个操作系统存在对其开发有具有决定作用的统一性理念吗？如果有，它和 Unix 的统一性理念有何不同？

彻头彻尾的反 Unix 系统，就是没有任何统一性理念，胡乱堆砌起的一些唬人特性而已。

多任务能力

各种操作系统最基本的不同之处之一就是操作系统支持多进程并发的能力。最低端的操作系统（如 DOS 或 CP/M），基本上就是一个顺序的程序加载器，根本不具备多任务能力。这种操作系统在通用计算机上已经毫无竞争力。

再往上一个层次，操作系统可具有**协作式多任务**（cooperative multitasking）能力。这种系统能够支持多个进程，但是一个进程

[1] 对没有 Unix 经验的读者来说，管道就是连接一个程序输出和另一个程序输入的通路。我们将在第 7 章讨论如何应用这个理念来帮助程序间的协作。

运行前必须等待前一个进程主动放弃占用处理器（这样一来，简单的编程错误就很容易将机器挂起）。这种操作系统风格是对一种硬件的暂时性适应，这种硬件虽然功能强大到支持并行操作，但要么缺乏周期性时钟中断[2]，要么缺乏内存管理单元，或者两者都缺。这种系统也过时了，不再具有竞争力了。

Unix 系统拥有**抢先式多任务**（preemptive multitasking）能力。在 Unix 中，时间片由调度程序来分配，这个调度程序定期中断或抢断正在运行的进程而把控制权交给下一个进程。几乎所有的现代操作系统都支持抢占式调度。

注意，“多任务”跟“多用户”不是一回事。一个操作系统可以进行多任务处理而只支持单用户，在这种情况下，计算机支持的是单个控制台和多个后台进程。真正的多用户支持需要多个用户权限域，我们将在随后讨论内部边界时进一步讨论这个特性。

彻头彻尾的反 Unix 系统一，就是绝无多任务处理能力——或者通过对进程管理增设诸多的规定、限制和特殊情况来削弱多任务能力——的一个废物。

协作进程

在 Unix 中，低价的进程生成和简便的进程间通讯（IPC Inter-Process Communication）使众多小工具、管道和过滤器组成一个均衡系统成为可能。我们将在第 7 章探讨这个均衡体系。在这里，我们需要指出代价高昂的进程生成和 IPC 会带来什么后果。

管道虽然在技术上容易发现，但影响却很大。进程是自主

[2] 硬件的周期性时钟中断对分时系统来说就像心跳一样重要。时钟中断定义了单位时间片的大小，每发生一次中断，就是告诉系统可以转换到另一个任务了。在 2003 年，各种 Unix 通常将这个“心跳”设置为每秒 60 次或每秒 100 次。

> 运算单元的统一性记号、而进程控制是可编程的——如果没有这些概念，那么管道技术就不可能这么简单。和 Multics 一样，Unix 的 shell（外壳）只是另外一个进程；进程控制并非受 JCL（作业控制语言）之赐。
>
> —Doug McIlroy

如果操作系统的进程生成代价昂贵，且/或进程控制非常困难、不灵活，后果通常是：

- 编写怪物般巨大的单个程序成为更自然的编程方式。
- 很多策略必须在这些庞大程序中表述。这会助长使用 C++ 和诡谲的内部代码层级，而不是 C 和相对平坦的内部层级。
- 当进程间不得不进行通讯时，要么只能采用笨拙、低效、不安全的机制（比如临时文件），要么就得依赖太多彼此的实现细节，要么彼此需了解对方的太多实现细节。
- 广泛使用多线程来完成某些任务，而这些任务 Unix 只需用互通的多进程就能处理。
- 必须学习和使用异步 I/O。

这些就是操作系统环境的局限性所导致的常见风格缺陷（甚至应用程序编程中也一样）的实例。

管道和所有其他经典 Unix IPC 方法有一个精微的性质，就是要求把程序间的通讯简化到某一程度而促使功能分离。相反地，如果没有与管道等效的机制，则程序必须在完全相互了解对方内部细节的基础上设计程序，才能实现彼此间的合作。

一个操作系统，如果没有灵活的 IPC 和使用 IPC 的强大传统，程序间就得通过共享结构复杂的数据实现通讯。由于一旦有新的程序加入通讯圈，圈子里所有程序的通讯问题都必须重新解决，所以解决方案的复杂度与协作程序数量的平方成正比。更糟糕的是，其

中任何一个程序的数据结构发生变化，都说不定会给其它程序带来什么隐蔽的 bug。

> Word、Excel、PowerPoint 和其他微软程序对彼此的内部具有“密切”——有些人可能称之为“杂乱”——的了解。在 Unix 中，一组程序设计时不仅要尽量考虑相互协作，而且要考虑和未知程序的协作。
>
> —Doug McIlroy

我们将在第 7 章再谈这个主题。

彻头彻尾的反 Unix 系统，就是让进程的生成代价高昂，让进程的控制困难而死板，让 IPC 可有可无，对它不予支持或支持很少。

内部边界

Unix 的准绳是：程序员最清楚一切。当你对自己的数据进行危险操作（例如执行 rm -rf *.）时，Unix 并不阻止你，也不会让你确认。另一方面，Unix 却小心避免你踩在别人的数据上。事实上，Unix 提倡设立多个帐户，每一个帐户具有专属、可能不同的权限，以保护用户不受行为不端程序的侵害[3]。系统程序通常都有自己的“伪用户（pseudo-user）帐号”，以访问专门的系统文件，而不需要无限制的（或者说超级用户的）访问权限。

Unix 至少设立了三层内部边界来防范恶意用户或有缺陷的程序。一层是内存管理：Unix 用硬件自身的内存管理单元（MMU）来保证各自的进程不会侵入到其它进程的内存地址空间。第二层是为多用户设置的真正权限组——普通用户（非 root 用户）的进程未经允许，就不能更改或者读取其他用户的文件。第三层是把涉及关

[3] 现在时髦的术语是基于角色的安全策略（Role-Based Security）。

键安全性的功能限制在尽可能小的可信代码块上。在 Unix 中，即使是 shell（系统命令解释器）也不是什么特权程序。

操作系统内部边界的稳定不仅是一个设计的抽象问题，它对系统安全性有着重要的实际影响。

彻头彻尾的反 Unix 系统，就是抛弃或回避内存管理，这样失控的进程就可以任意摧毁、搅乱或破坏掉其它正在运行的程序：弱化甚至不设置权限组，这样用户就可以轻而易举地修改他人的文件和系统的关键数据（例如，掌控了 Word 程序的宏病毒可以格式化硬盘）；依赖大量的代码，如整个 shell 和 GUI，这样任何代码的 bug 或对代码的成功攻击都可以威胁到整个系统。

文件属性和记录结构

Unix 文件既没有记录结构（record structure）也没有文件属性。在一些操作系统中，文件具有相关的记录结构：操作系统（或其服务程序库）通过固定长度的纪录，了解文件，或文本行终止符以及 CR/LF（回车／换行）是不是该作为单个逻辑字符读取。

在另一些操作系统中，文件和目录可以具备相关的名字／属性对——（例如）采用编外数据（out-of-band data）将文档文件同能够解读它的应用程序关联起来。（Unix 处理这种联系的典型方法是让应用程序识别“特征数”或是文件内的其它类型数据。）

操作系统级的记录结构通常只是一个优化手段，几乎只会使 API 和程序员的生活复杂化之外没别的用，还会助长不透明的面向记录的文件格式，使得文本编辑器之类的通用工具无法正确读取。

文件属性会很有用，但是（我们在第 20 章将发现）在面向字节流工具和管道的世界中，它可能引发一些棘手的语义问题。对文件属性的操作系统级支持会诱导程序员使用不透明的文件格式，让他们依靠文件属性将文件格式同对应的解读程序绑在一起。

彻头彻尾的反 Unix 系统，应用一套拙劣的记录结构，任何特定的工具能否像文件编写者希望的那样读懂文件，完全是靠运气。加入文件属性，并让系统依赖于这些文件属性，就无法通过查看文件内的数据来确定文件的语义。

二进制文件格式

如果你的操作系统使用二进制文件格式存放关键数据（如用户帐号记录），应用程序采用可读文本格式的传统就很可能无法形成。我们将在第 5 章详细解释为什么这是一个问题。现在只要注意，这种做法可能会带来以下后果就够了。

- 即使支持命令行接口、脚本和管道，也几乎无法形成过滤器。
- 数据文件足有通过专用工具才能访问。开发者的思维会以工具而非数据为中心。这样，不同版本的文件格式很难兼容。

彻头彻尾的反 Unix 系统，让所有文件格式都采用不透明的二进制格式，后者要用重量级的工具才能读取和编辑。

首选用户界面风格

我们将在第 11 章详细讨论**命令行界面**（CLI）和**图形用户界面**（GUI）的差异所产生的影响。操作系统的设计者把哪一种选作一般表现模式，将影响设计的许多方面——从进程调度、内存管理直到应用程序使用的**应用程序编程接口**（API）。

第一款 Macintosh 已经发布很多年了，不用说人们也会觉得操作系统的 GUI 没做好是个问题。Unix 的教训则相反：CLI 没做好是一个不太明显但同样严重的缺陷。

如果操作系统的 CLI 功能很弱或根本不存在，其后果会是：

- 程序设计不会考虑以未预料到的方式相互协作——因为无法这样设计。输出不能用作输入。
- 远程系统管理更难于实现，更难以使用，更强调网络。[4]
- 即便简单的非交互程序也将招致 GUI 开销或复杂的脚本接口。
- 服务器、守护程序和后台进程几乎无法写出，至少很难以优雅的方式写出。

彻头彻尾的反 Unix 系统，就是没有 CLI，没有脚本编程能力——或者，存在 CLI 不能驱动的重要功能。

目标受众

不同的操作系统设计是为了适应不同的目标受众。有的为后台工作设计，有的则设计成桌面系统。有的为技术用户而设计，有的则为最终用户设计。有的能在实时控制应用中单机工作，有的则为分时系统和普遍联网的环境设计。

一个重要的差异是客户端与服务器之分。“客户端”可以理解为：轻量，只支持单个用户，能够在小型机器上运行，随需开关机器，没有抢先式多任务处理，为低延迟作了优化，大量资源都用在花哨的用户界面上。“服务器”可以解释为：重量，能够连续运行，为吞吐量优化，完全抢占式多任务处理以处理多重会话。所有的操作系统最初都是服务器操作系统。客户端操作系统的概念仅在二十世纪七十年代后期随着价格不高、性能一般的 PC 硬件的出现才产生。客户端操作系统更关注用户的视觉体验，而不是 7*24 小时的连续正常运行。

所有这些变数都对开发风格产生影响。其中最明显的就是目标用户能够容忍的界面复杂的级别，以及如何在可感知复杂度和成

---

[4] 微软重建 Hotmail 时认真考虑了这个问题。参见 [BrooksD]。

本、性能等其它变数之间权衡轻重。人们常说，Unix 是程序员写给程序员的——这个目标用户群在界面复杂度的承受力方面是出了名的。

> 这与其说是一个目标不如说是一个结果。如果“用户”这个词带有“单纯得傻乎乎”的蔑视含义，我憎恨一个为“用户”设计的系统。
>
> —Ken Thompson

彻头彻尾的反 Unix 系统，就是一个自认为比你自己更懂你在干什么的操作系统，然后雪上加霜的是，它还做错了。

开发的门坎

区分操作系统的另一个重要尺度是纯用户转变为开发者的门坎高度。这里有两个重要的成本动因。一个是开发工具的金钱成本，另一个是成为一个熟练开发者的时间成本。有些开发文化还形成了一个社会性门坎，但这通常是背后的技术成本带来的结果，而不是根本原因。

昂贵的开发工具和复杂晦涩的 API 造就了小群的精英编程文化。在这种文化中，编程项目是大型而严肃的活动——为了证明所投资的软（人力）硬资本物有所值，这些工程必须如此。大型而严肃的工程常常产生大型、严肃的程序（而且，更常见的是，大型而昂贵的失败）。

廉价工具和简单接口支持的是轻松编程、玩家文化和开拓探索。编程项目可以很小（通常，正式的项目结构显然毫无必要），失败了也不是什么大灾难。这改变了人们开发代码的风格；尤其是，他们往往不会过分依赖已经失败的方法。

轻松编程往往会产生许多小程序和一个自我增强、不断扩展的知识社区。在廉价硬件的世界里，是否存在这样一个社区日益成为

一个操作系统能否长寿的重要因素。

Unix 开创了轻松编程的先河。Unix 的众多首创之一就是将编译器和脚本工具放在默认安装中，可供所有用户使用，支持了一种跨越众多机器的玩家开发文化。在很多 Unix 下写代码的人并不认为自己在写代码——他们认为是在为普通任务的自动化编写脚本，或在定制环境。

彻头彻尾的反 Unix 系统，不可能进行轻松编程。

操作系统的比较

当我们将 Unix 和其他操作系统对比时，Unix 设计决策的逻辑就更清楚了。这里，我们只是纵览各种设计，对各操作系统技术特性的具体讨论请参考 OSData 网站。[5]

图 3-1 表明我们要纵览的各种分时系统之间的渊源关系。其它一些操作系统（灰色标记，不一定是分时系统）因脉络的关系也包括进来了。实线框里的系统仍旧存在。“出生”日期是指第一次发布的时间；[6]“死亡”日期通常指厂商终结系统的时间。

实线箭头表示存在起源关系或很强的设计影响（例如，后开发的系统 API 有意通过逆向工程以匹配先前的系统）；短画线表示重大的设计影响；点线表示微弱的设计影响。不是所有的起源关系被开发者认可；实际上，有些出于法律或企业策略的原因被官方否认，但在业界其实是公开的秘密。

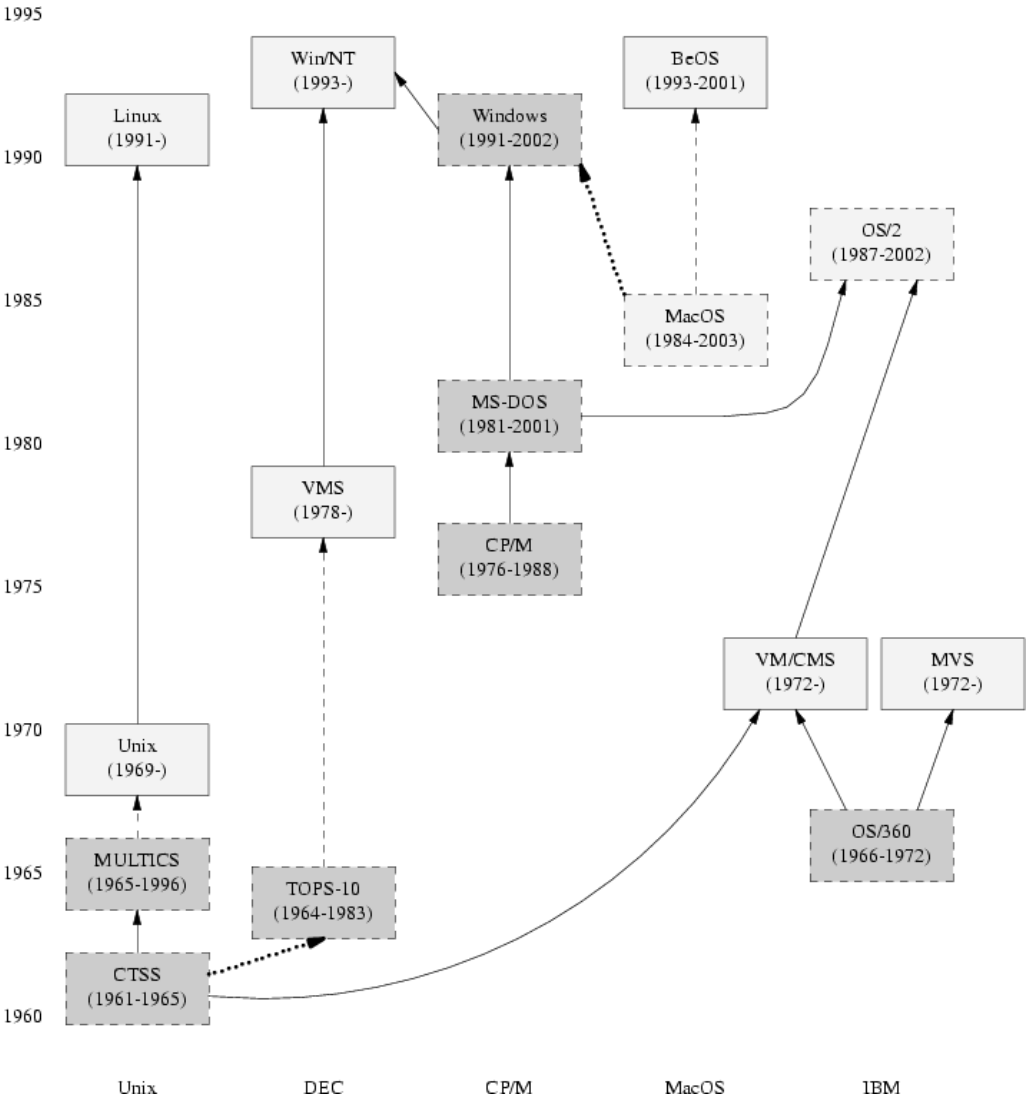
“Unix”框包括所有的专有 Unix，包括 AT&T 版本和早期的伯克利版本。“Linux”框包括所有的开源 Unix（均在 1991 年开

---

[5] 参考 OSData 网站http://www.osdata.com/。

[6] Multics 除外，它影响最大的时期是自技术规格公布的 1965 年到实际发布的 1969 年间。

始）。这些开源 Unix 与早期的 Unix 有渊源关系，它建立在 1993 年诉讼协议后从 AT&T 专有控制下解放出来的代码的基础上。[7]



VMS

VMS 是一个专有操作系统，最初由数字设备公司（Digital Equipment Corporation）为 VAX 小型机开发。VMS 于 1978 年面

[7] 这次诉讼的详细情况可以参考 Marshall Kirk McKusick 在 [open-Sources] 中的论文。

世，是二十世纪八十年代和九十年代早期一个非常重要的产业化操作系统产品，无论在 Compaq 并购 DEC，还是 Hewlett-Packard 并购 Compaq 之后，这个系统一直得到了维护。直到 2003 年中期，这个产品仍在销售和支持，尽管今天已经没有多少人用它搞新的开发了[8]。在这里提出 VMS，是为了对比 Unix 和来自小型机时代的其它 CLI 操作系统。

VMS 具有完全抢占式多任务处理能力，但是进程生成的开销极为昂贵。VMS 文件系统有复杂的记录类型（但还不是记录属性）概念。这些特性造成了我们此前描述的后果，尤其（在 VMS 中）是程序庞大、个体臃肿的倾向。

VMS 的特点是具有长长的、可读的、类 COBOL 的系统命令和命令选项。它具有非常全面的在线帮助（针对的不是 API，而是可执行程序和命令行语法）。事实上，VMS 命令行界面及其帮助系统就代表了 VMS 的组织结构。尽管在该系统上已经具有翻新版的 X window，但冗长的 CLI 仍然对编程设计的风格产生了重要影响。主要可以理解为：

- 命令行功能的使用频率——要打的字越多，愿意用的人就越少。
- 程序的大小——人们希望少打字，因此想少用几个软件，于是将更多功能捆绑到大型程序中。
- 程序可接受选项的数量和类型——必须遵守帮助系统规定的语法限制。
- 帮助系统的易用性——很完备，但缺少辅助的搜索和查找工具，并且索引做得很差。这样不容易获取大量的知识，鼓励了专业化而阻碍了轻松编程。

[8] 更多信息可以从 OpenVMS.org 网站http://www.openvsm.org获得。

VMS 的内部边界系统有口皆碑。它为真正的多用户操作而设计，完全利用硬件 MMU 来保护进程互不干扰。系统命令解释器具有优先权，但在另一方面，关键功能封装则做得相当不错。VMS 的安全漏洞一直都很罕见。

VMS 工具最初很贵，界面也很复杂。大卷大卷的 VMS 程序员文档只有纸张形式，因此要查找任何东西都既费时又费钱。这往往会阻碍探索性编程，降低人们对大型工具包的学习兴趣。直到几乎被厂商抛弃之后，VMS 才形成一种轻松编程和玩家文化，但这种文化并非很强。

和 Unix 一样，VMS 早就有了客户端／服务器的划分。作为一个通用分时系统，VMS 在它的时代是成功的。VMS 的目标受众基本是技术用户和大量应用软件的商业领域，这也意味着用户对其复杂度尚能容忍。

MacOS

Macintosh 操作系统是 1980 年代初由 Apple 公司设计的，灵感来自此前 Xerox 公司 Palo Alto 研究中心在 GUI 方面的开拓性工作。1984 年与 Macintosh 计算机一起面世。MacOS 经历过两场重大的设计变革，第三场正在酝酿中。第一次是从一次只支持一个应用程序转变到能够多任务协作处理多个应用程序 (MultiFinder)；第二次是从 68000 处理器到 PowerPC 处理器的转变，既保留了 68K 应用程序的二进制向后兼容，又为 PowerPC 应用程序引入了高级共享库管理系统，代替了原来的 68K 基于陷阱指令的代码共享系统（trap instruction-based code sharing system）；第三次是在 MacOS X 中把 MacOS 设计理念和来自 Unix 的架构融合起来。如果没有特别指出，此处讨论仅限 OS-X 之前的版本。

MacOS 有一个不同于 Unix 的坚定统一性理念：Mac 界面方针（the Mac Interface Guidelines）。这些方针非常详细地说明了应

用程序 GUI 的表现形式和行为模式。这些原则的一致性在很多重要方面影响了 Mac 用户的文化。不遵循这些原则、简单移植 DOS 或 Unix 程序的产品立即遭到了 Mac 用户的拒绝，在市场上一败涂地，而这并不罕见。

这些方针的主旨是：东西永远呆在你摆的地方。文档、目录和其它东西在桌面上都有固定的、系统不会弄乱的位置，重启后桌面依然保持原样。

Macintosh 的统一性理念非常强大，我们上面讨论过的其它设计方案要么受其影响，要么就无人问津。所有的程序都得有 GUI，根本没有 CLI。脚本的功能有倒是有，但绝对不像 Unix 中那样常用，很多 Mac 程序员根本就不去学习。MacOS 界面至上的 GUI 做法（被组织到单个的主事件循环中）导致了其薄弱的非抢占式的程序调度能力。这个弱程序调度器以及所有的 MultiFinder 应用程序都在单个大地址空间运行，这意味着使用分离的进程甚或线程来代替轮询[9]是不现实的。

然而，MacOS 的应用程序并非总是庞然大物。系统的 GUI 支持代码，部分在硬件自带的 ROM 实现，部分在共享库中实现，通过事件接口同 MacOS 中的程序进行通信，这个接口从诞生起就一直相当稳定。这样，这种操作系统的设计提倡的是把应用引擎和 GUI 接口相对清晰地分离开来。

MacOS 也强烈支持把应用程序的元数据（如菜单结构）从引擎中隔离。MacOS 文件分“数据分支”（data fork）（Unix 风格的字节包，包含文档或程序代码）和“资源分支”（resource fork）（一套用户定义的文件属性）。Mac 应用程序通常是这样设计的，（比如）程序中使用的图像和声音存储在资源分支中，可以

---

[9] 轮询法的概念是，由 CPU 定时发出询问，依序询问每一个周边设备是否需要其服务，有即给予服务，服务结束後再问下一个周边，接着不断周而复始。

独立于应用程序码进行修改。

MacOS 系统的内部边界系统很弱。因为基于只有单个用户这样的固定设想，所以没有用户权限组。多任务处理是协作式的，不是抢占式的。所有的 MultiFinder 应用都在同一个地址空间运行，所以任何应用程序的不良代码都能破坏操作系统低层内核以外的任何部分。针对 MacOS 机器的安全攻击程序很容易编写，这个系统一直没遭到大规模攻击只是因为没人有兴趣罢了。

Mac 程序员和 Unix 程序员在设计上往往走截然相反的路，即，他们的设计是从界面向内进行，而不是从引擎向外进行（我们将在 20 章讨论这种方式的影响）。MacOS 的一切设计共同促成了这种做法。

Macintosh 的目标是作为是服务非技术终端用户的客户端操作系统，这就意味着用户对界面复杂度的容忍度很低；Macintosh 文化下的开发者于是非常非常擅长设计简洁的界面。

假设你已经有 Macintosh 机器，那么晋级为开发者的代价一向不高。因此，尽管界面相当复杂，Mac 很早也形成了一种浓厚的玩家文化。开发小工具、共享软件和用户支持软件的传统一直非常盛行。

经典的 MacOS 已经寿终正寝。MacOS 大多数功能已被引入 Macos X，并同源自 Berkeley 传统的 Unix 架构结合在一起[10]。同时，像 Linux 这样的前沿 Unix 也开始从 MacOS 中借鉴一些理念，如文件属性（资源分支的泛化）。

---

[10] MacOS x 实际是两层专有代码（OpenStep 移植码和经典 Mac GUI）和开源 Unix 核心上（Darwin）的组合。

OS/2

OS/2 是作为 IBM 命名为“ADOS”（Advanced DOS）的开发项目诞生的，也是想成为 DOS 4 的三个竞争者之一。那时，IBM 和微软在正式合作，为 PC 机开发下一代操作系统。OS/2 1.0 版本首发于 1987 年，为 286 机开发，并不成功。针对 386 的 2.0 版本发布于 1992 年，但那时 IBM 和微软联盟已经破裂。微软走向一个不同的（而且更赚钱的）方向——Windows 3.0。OS/2 虽然吸引了一小部分忠诚的拥趸，但从来没有吸引到足够多的开发者和用户。直到 1996 年后 IBM 把它纳入 Java 计划前，OS/2 在桌面市场一直排在 Macintosh 之后，位居第三。最新版本是 1996 年发布的 4.0 版本。那些早期的版本在嵌入式系统中找到了出路，时至 2003 年年中，还在全球众多银行自动柜员机上运行。

和 Unix 一样，OS/2 使用抢先式多任务处理，不能在没有 MMU 的机器上运行（早期版本使用 286 的内存分段来模拟 MMU）。跟 Unix 不同的是，OS/2 从来都不是一个多用户系统。虽然它的进程生成开销相对较低，但是 IPC 困难而脆弱。网络能力最初仅限于 LAN 协议，但后续版本也增加了 TCP/IP 协议栈。因为没有类似于 Unix 的服务守护程序，所以，OS/2 处理多功能网络的能力一直欠佳。

OS/2 既有 CLI 又有 GUI。OS/2 流传下来的亮点大多围绕它的桌面 Workplace Shell（WPS）。有一些技术从 AmigaOS Workbench[11]的开发者处授权得到。AmigaOS Workbench 是 GUI 桌面的先驱，直到 2003 年，在欧洲还拥有众多忠实的爱好者。这也是 OS/2 的能力超过 Unix（这一点尚有争论）的唯一设计领域。WPS（WorkplaceShell）是一个干净、强大、面向对象的设计，具

[11] 作为对某些 Armga 技术的回报，IBM 给予了 Commodore 公司 REXX 脚本语言的授权。此项交易详情请查询：http://www.os2bbs.com/os2news/OS2Warp.html。

有易懂的行为特性和良好的可扩展性。几年后，OS/2 成为 Linux GNOME 工程的模型。

WPS 的类层次设计是 OS/2 的统一性理念之一。另一个统一性理念是多线程处理。OS/2 程序员大量使用线程，部分代替了对等进程间的 IPC，协作程序工具包传统也因此没能形成。

OS/2 的内部边界达到了单用户操作系统的预期。运行的进程互不干扰，内核空间也和用户空间互不干扰，但是没有了基于每用户的特权组。这意味着文件系统无法防范恶意代码。另一个结果是没有类似于起始目录的东西，应用程序的数据往往散布在整个系统中。

缺乏多用户能力所产生的进一步后果就是在用户空间不存在权限区别。这样，开发者往往只信任内核代码。Unix 中许多由用户态守护进程处理的系统任务在 OS/2 中只好塞进内核或 WPS，结果是两者都臃肿。

OS/2 有一种和二进制模式相对的文本模式（文本模式下 CR/LF 被读作单个的行结束符，在二进制模式下无此含义），但是没有其它的文件记录结构。OS/2 支持文件属性，效仿 Macintosh 风格，用文件属性来支持桌面持久性。系统数据库大都是二进制格式。

首选 UI 风格贯穿于 WPS。从人体工程学角度来说，WPS 的用户界面要强于 Windows，虽然还没有达到 Macintosh 的标准（OS/2 最活跃的时段在经典 MacOS 的历史上处于早期）。与 Linux 和 Windows 一样，OS/2 的用户界面围绕多个独立的窗口任务组，而不是让运行的应用程序占据整个桌面。

OS/2 的目标对象是商业和非技术的最终用户，意味着对界面复杂度的容忍度较低。OS/2 既可用作客户端操作系统，也可用作文件和打印服务器。

在二十世纪九十年代初期，OS/2 社区的开发者开始转向受

Unix 启发、模仿 POSIX 接口的 EMX 环境。到二十世纪九十年代后期，已经有很多 Unix 软件被移植到 OS/2 上。

任何人都可以下载 EMX，包括 GNU 编译器集合以及其它开源开发工具。IBM 不时在 OS/2 开发包中发布系统文档，并被转载到了许多 BBS 和 FTP 站点上。正因为如此，到 1995 年，用户开发的 OS/2 软件的“Hobbes”FTP 档案已经超过了 1GB。一个崇尚小巧工具、探索编程和共享软件的强大传统形成了，即使 OS/2 自身已经被丢进了历史的垃圾箱，这个传统仍然还会长期拥有一批忠诚的追随者。

在 Windows 95 发布以后，OS/2 社区在微软的围剿和 IBM 的支援下，对 Java 的兴趣与曰俱增。自 Netscape 在 1998 年初公开源码后，他们的方向又（陡然）转到了 Linux 上。

一个多任务处理、但单用户的操作系统到底能走多远？OS/2 是一个相当有趣的案例。从其得出的大部分结论都可以很好地运用到其它同类型操作系统中，尤其是 AmigaOS[12]和 GEM[13]。直到 2003 年，大量的 OS/2 材料还可从网上获得，包括一些闪光的历史。[14]

Windows NT

Windows NT（New Technology）是微软为高端个人用户和服务器设计的操作系统：发行的版本实际上有好几个，我们为了讨论方便把它们视为一个系统。自从 2000 年公布的 Windows ME

---

[12] AmigaOS 的主页http://os.amiga.com/

[13] GEM 操作系统 http://www.geocities.com/SiliconValley/Vista/6148/gem.html

[14] 例如，参考 OS Voicehttp://www.os2voice.org/ 和 OS/2 BBS.COMhttp://www.os2bbs.com/

终结后，目前所有的 Window 操作系统都以 Window NT 为基础；Windows2000 是 NT 5，Windows XP（本书写作时是 2003 年）是 NT 5.1。NT 起源自 VMS，很多重要特性与 VMS 相同。

NT 是逐步堆积而成的，缺乏对应于 Unix“一切皆文件”或 MacOS 桌面的统一性理念。由于它的核心技术没有扎根于一小群稳固的中枢观念中，[15]因此每过几年就会过时。每一代技术——DOS（1981），Windows 3.1（1992），Windows 95（1995），Windows NT 4（1996），Windows 2000（2000），Windows XP（2002）和 Windows Server 2003（2003）——随着旧方式被宣告过时而不再有良好支持，开发者必须以不同的方式从头学起。

下面是其它一些后果：

- GUI 功能与继承自 DOS 和 VMS 的戏留命令行界面不能稳定共存。
- 套接字编程没有类似 Unix 那种“一切皆是文件句柄”的统一数据对象，因此在 Unix 中很简单的多道程序设计和网络应用到 NT 下则要牵涉更多基础性概念。

NT 的一些文件系统类型也有文件属性，但仅限用于为实现某些文件系统的访问控制列表，因此对开发风格不会产生太大影响。NT 也有文本和二进制这两种记录类型区别，时不时地讨人嫌（NT 和 OS/2 都从 DOS 那里继承了这个不良特性）。

尽管支持抢先式多任务处理，但进程生成却很昂贵——虽然比不上 VMS，但是（平均生成一个进程需要 0.1 秒左右）要比现在的 Unix 高出一个数量级。脚本功能薄弱，操作系统广泛使用二进制文件格式。除了此前我们总结过的，还有这些后果：

15　也许，会有人争辩说，所有微软操作系统的统一性理念是：“套牢客户”。

- 大多数程序都不能用脚本调用。程序间依赖复杂脆弱的远程过程调用（RPC）来通信，这是滋生 bug 的温床。
- 根本就不存在通用工具。没有专用软件就不可能读取或编辑文档和数据库。
- 随着时间的推移，CLI 越来越被忽略了，原因是环境稀缺。薄弱 CLI 引起的问题不仅没有得到改善，反而越来越糟糕。（Windows Server 2003 试图稍稍扭转这种趋势。）

Unix 的系统配置和用户配置数据分散存放在众多的 dotfiles（名字以“.”开头的文件）和系统数据文件中，而 NT 则集中存放在注册表中。以下后果贯穿于设计中：

- 注册表使得整个系统完全不具备正交性。应用程序的单点故障就会损毁注册表，经常使得整个操作系统无法使用、必须重装。
- **注册表蠕变**（registry creep）现象：随着注册表的膨胀，越来越大的存取开销拖慢了所有程序的运行。

互联网上的 NT 系统因易受各种蠕虫、病毒、损毁程序以及破解（crack）的攻击而臭名昭著。原因很多，但有一些是根本性的，最根本的就是：NT 的内部边界漏洞太多。

NT 有访问控制列表，可用于实现用户权限组管理，但许多遗留代码对此视而不见，而操作系统为了不破坏向后兼容性又允许这种现象的存在。在各个 GUI 客户端之间的消息通讯机制也没有安全控制，如果加上的话，也会破坏向后兼容性。

虽然 NT 将要使用 MMU，出于性能方面的考虑，NT 3.5 后的版本将系统 GUI 和优先内核一起塞进了同一个地址空间。为了获得和 Unix 相近的速度，最新版本的 NT 甚至将 Web 服务器也塞进了内核空间。

由于这些内部边界漏洞产生的协合效应，要在 NT 上达到真正的安全实际上是不可能的[16]。如果入侵者随便作为什么用户把一段代码运行起来（例如，通过 Outlook email 宏功能），这段代码就可以通过窗口系统向其它任何运行的应用程序发送虚假信息。只要利用缓存溢出或 GUI 及 Web 服务器的缺口就可以控制整个系统。

因为 Windows 没有处理好程序库的版本控制问题，所以长期备受被称为“DLL 地狱（DLL hell）”配置问题的折磨，在这个问题中，安装新程序可以任意升级（或降级）现有程序运行依赖的库文件。专用的应用程序库和厂商提供的系统库都存在这个问题：应用程序和特定版本的系统库一起发布非常普遍，一旦没有特定的系统库，应用程序就会无声无息地垮掉。”[17]

从好的一面来看，NT 提供了足够的特性来支持 Cygwin。Cygwin 是一个在实用工具和 API 两个层次上实现 Unix 的兼容层，而且只有极少的特性损失[18]。Cygwin 允许 C 程序既可以使用 Unix API 又可以使用原生 API，许多为形势所迫不得不使用 Windows 的 Unix 黑客在 Windows 系统上安装的第一个程序就是 Cygwin。

NT 操作系统的目标用户主要是非技术型最终用户，意味着对界面复杂度的容忍度非常低。NT 既可作客户端又可作服务器。

在其历史早期，微软依靠第三方开发商提供应用软件。起初，微软还公布 Windows API 的完整文档，并保持其开发工具的低价格。但是，随着时间的推移、竞争者的相继倒下，微软转而青睐内部开发的战略，开始向外界隐藏 API，开发工具也越来越昂贵。早

16　实际上，微软已经于 2003 年 3 月公开承认 NT 系统的安全是不可靠的。

17　在有处理库版本问题能力的.NET 开发框架发布后，DLL hell 问题有所缓解一一但是直到 2003 年.NET 只随 NT 最高端的服务器版本提供。

18　Cygwin 很大程度上符合“单一 Unix 规范”，但是要求直接硬件存取的程序会被上层的 Windows 内核限制。以太网卡就是出了名的问题多。

在 Windows 95 时期，微软就要求将保密协议作为购买专业级开发工具的一个条件。

围绕 DOS 和 Windows 早期版本形成的玩家文化和轻松开发文化已经足够壮大，即使在微软日益加强的排挤（包括为了把业余开发者非法化而设立的各种认证计划）下也足以自我维系。共享软件从未消亡，而在 2000 年后，迫于开源操作系统和 Java 的市场压力，微软的策略也略有转变。但是，随着时间的推移，供“专业”编程使用的 Windows 接口越来越复杂，将轻松（或严肃！）编程的门槛越抬越高。

这段历史的后果就是业余 NT 和职业 NT 开发者的设计风格存在尖锐的分歧——两个群体之间几乎不通气。尽管小型工具和共享软件的玩家文化非常活跃，但职业 NT 项目却往往产出庞然大物，甚至比那些 VMS 一样的“精英”操作系统还要臃肿。

Windows 下的 Unix 风格的 shell 功能、命令集和 API 函数库来自第三方，包括 UWIN、Interix 和开源 Cygwin。

BeOS

Be 公司作为一家硬件厂商成立于 1989 年，基于 PowerPC 芯片开发了颇具开拓精神的多处理机器。BeOS 操作系统是 Be 公司为给硬件增值而发明的一种新型、内置网络功能的操作系统模型，吸收了 Unix 和 MacOS 两个家族的经验教训，但又不和任何一个雷同。他们的努力造就了一个雅致、简洁、令人激动的设计，在其定位的多媒体平台这个角色上表现卓越。

BeOS 的统一性理念是“深入地线程化”、多媒体流和数据库形式的文件系统。BeOS 的设计目标是尽可能减少内核延迟，从而能非常适合实时处理大量数据，如音频和视频流。既然支持线程本地存储从而不需共享所有地址空间，BeOS 的“线程”实际上就是 Unix 术语中的轻量级进程。IPC 通过共享内存实现，快速而高效。

BeOS 采用的是 Unix 模型，在字节级以上没有文件结构。BeOS 和 MacOS 一样支持和使用文件属性。事实上，BeOS 文件系统就足一个数据库，可以按任意属性索引。

BeOS 借鉴 Unix 的设计是巧妙的内部边界设计。BeOS 充分应用了 MMU，而且有效地使各个运行进程互不干涉。虽然 BeOS 是个单用户操作系统（不用登录），但在文件系统和操作系统内部的其它地方都支持类似 Unix 的权限组。这些措施用于保护系统的关键文件免受不信任代码的侵袭：用 Unix 的术语来讲，就是用户在启动时作为匿名用户登录，另一个“用户”是 root。如果需要完整的多用户操作，其实对系统上层产生的变化也会很小，实际上确实存在一个 BeLogin 实用程序。

BeOS 倾向使用二进制文件格式和文件系统自带的数据库，而不使用类 Unix 的文本格式。

BeOS 的首选 UI 风格是 GUI，在界面设计上大量借鉴了 MacOS 的经验，但是完全支持 CLI 和脚本功能。BeOS 的命令行 shell 是移植自 Unix 最主要的开源 shell—bash(1)，通过 POSIX 兼容库运行。移植 Unix CLI 软件在设计上相当容易。Unix 模式的整套脚本、过滤器和守护进程的基础设施都到位了。

BeOS 的目标定位是作为一个专门针对近实时（near-real-time）多媒体处理（尤其是音频和视频操控）的客户端操作泵统。BeOS 的目标受众包括技术和商业用户，这也意味着用户对界面复杂度的容忍度属中等。

BeOS 的开发门槛很低：尽管操作系统是专有的，但是开发工具并不贵，而且很容易获得整套文档。BeOS 项目起初部分为了通过 RISC 技术把 Intel 硬件拉下马，在互联网大爆炸后，继续往一个纯软件方向努力。在 1990 年代初 Linux 形成时期，BeOS 的战略家就已经一直关注着、而且也充分意识到一个庞大的轻松开发者团体的价值。事实上，他们成功地吸引了一批非常忠诚的追随者；

到 2003 年，至少有五个以上的不同工程正在努力试图用开源复兴 BeOS。

不幸的是，BeOS 的经营战略却不像其技术设计那样精明。起初，BeOS 软件捆绑在专用硬件上，市场推广时对目标应用的说明也含混不清。后来（1998 年），BeOS 被移植到通用 PC 机上，更紧密关注多媒体应用，但是从未吸引到足够数量的应用和用户群。最后，到 2001 年，BeOS 死于微软的反竞争运动（2003 年仍在进行诉讼）和各种已具备多媒体功能的 Linux 的联合打击之下。

MVS

MVS（多重虚拟存储）是 IBM 大型计算机的旗舰操作系统，起源可以追溯到 OS/360。OS/360 诞生于 1960 年代中期，是 IBM 当时很新型的 System/360 计算机系统上向客户推荐的操作系统。今天 IBM 大型机操作系统的核心还保留着 OS/360 的后裔代码。虽然整个代码几乎都已经重写了，但是基本设计大多原样未动；向后兼容性被虔诚地保留了下来。这种兼容性甚至达到这种地步：即使历经三代结构升级，为 OS/360 编制的应用程序还能不加修改就在装有 MVS 的 64 位 z/系列大型机上运行。

在上述讨论过的所有操作系统中，MVS 是唯一可视为比 Unix 还要悠久的操作系统（不确定性在于随着时间的推移，MVS 究竟发展到了什么地步）。这个操作系统也是受 Unix 概念和技术影响最小的操作系统，因而代表了跟 Unix 反差最强烈的一种设计。MVS 的统一性理念是：一切皆批处理。系统的设计目标是尽可能最有效利用机器批处理巨大规模的数据，尽量减少与人类用户的交互。

原生的 MVS 终端（3270 系列）只能以块模式运行。用户通过屏幕修改终端的本地存储。用户按下发送键前主机不会产生任何中断。不可能实现 Unix 原始模式（raw mode）下那种字符层面上的

交互。

TSO 是和 Unix 交互环境最近似的等价物，自身能力非常有限。对于系统其它部分来说，每个 TSO 用户都是模拟批作业。这个设施非常昂贵——太贵了，主要限于开发者和系统维护者使用。仅仅需要通过终端运行应用程序的普通用户几乎从不使用 TSO。相反，他们通过事务监视器工作。这是一种多用户应用服务器，可以进行协作式多任务处理并支持异步输入/输出。从效果上来说，每种事务监视器都是一个专用的分时插件（和运行 CGI 的 Web 服务器很像，但不完全一样）。

面向批处理体系所带来的另一个后果就是生成进程非常缓慢。I/O 系统有意用较高的准备成本（及其带来的延迟）来换取更好的吞吐能力。这些选择对于批处理操作来说非常适宜，但是对于交互响应来说却是致命的。可以预见，如今 TSO 用户将把几乎所有的时间都花在 ISPF（一个对话驱动的交互环境）上。除了启动一个 ISPF 实例外，程序员几乎不在原生的 TSO 上做任何事情。这避免了生成进程的开销，代价是引进了一个非常庞大的程序。这个程序，除了不会启动机房的咖啡壶，什么事都能做。

MVS 使用机器 MMU，进程有独立的地址空间，只能通过共享内存支持进程间通信，也有线程功能（MVS 称之为“子任务”），但用得很少，主要因为只有用汇编语言编写的程序才能方便地使用这个功能。与此相反，典型的批处理应用是由 JCI。（Job Control Language，作业控制语言）粘合在一起的由重量级程序调用组成的短序列，也提供脚本功能，但却是出了名的困难和死板。每个作业里的程序通过临时文件通信：过滤器之类的东西几乎毫无用武之地。

每个文件都有记录格式，有时是隐式的（例如，JCL 的内联输入文件继承了穿孔卡做法，默认为 80 字节固定长度的记录格式），但更通常的情况是明确指定。许多系统配置文件都采用文本格式，但应用程序文件通常采用特定的二进制文件。一些检查文件

的通用工具出于迫切需求才被开发出来，但这依然还是一个难以解决的问题。

文件系统的安全性在最初设计中根本未予考虑。然而，当人们发现安全性十分必要时，IBM 以一种颇具灵感的方式加了进去：他们规定了一套通用安全性 API，然后在处理每个文件存取请求前调用这个接口。结果是，产生了三种相互竞争的安全性程序包，各代表不同的设计理念——三种都相当好，在 1980 年到 2003 年中期始终没被攻破。这种多样性就允许用户安装时选择最适合实际安全策略的安全包。

网络功能也是后来才加进去的。网络连接和本地文件操作使用同一套接口的概念不存在；两者的编程接口相互独立而且区别很大。这的确帮助 TCP/IP 成为了首选网络协议，不着痕迹地挤掉了 IBM 原生的 SNA（System Network Architecture，系统网络体系）。在 2003 年，同一机器上两者都使用的情况虽然常见，但是 SNA 正在逐渐消亡。

除了在运行 MVS 的大企业内部，MVS 上的轻松编程几乎不存在。这主要不在于工具自身的成本，而在于环境的成本——在往计算机系统上扔进几百万美元后，每个月为编译器花费几百美元就是小钱了。然而，在这个社区内也存在一个繁荣的自由软件文化，主要是编程和系统管理工具。第一个计算机用户组，SHARE，就是 IBM 用户在 1955 年成立的，到今天也依然很兴旺。

考虑到架构上的巨大差别，MVS 是第一款符合单一 Unix 规范（Single Unix Specification）的非 System-V 操作系统，这件事非同寻常（但还是得看到，从 Unix 软件移植过来的软件往往碰到 ASCII 对 EBCDIC 字符集的麻烦）。从 TSO 启动 Unix shell 是可能的——Unix 文件系统专门设置成 MVS 数据集格式。MVS Unix 字符集是特殊 EBCDIC 代码页，交换了“新行”和“换行”（Unix 中的“换行”对 MVS 就是“新行”），但是系统调用却是在 MVS 内核上实现的实时系统调用。

随着开发环境的费用下降到爱好者能够承受的范围，公共领域的 MVS 版本（版本 3.8，始于 1979 年）拥有了一小群用户，人数虽少却在不断增长。这个系统及其开发工具和运行所用的仿真器，花一张 CD 的价钱就可以全部获得[19]。

MVS 的目标始终定位在后勤部门。和 VMS 和 Unix 一样，MVS 提前区分了服务器和客户端。后勤用户对界面复杂度不仅可以忍受，而且非常期待，因为他们愿意把昂贵的计算机资源尽可能花在需要处理的工作上而不是界面上。

VM/CMS

VM/CMS 是 IBM 另一个大型机操作系统。从历史来说，这是 Unix 的伯父；它们共同的祖先是 CTSS——由 MIT 于 1963 年间开发出来并在 IBM 7094 大型机上运行的一个系统。CTSS 开发组后来又去开发了 Multics，也就是 Unix 的直系祖先。IBM 在剑桥大学组建了一个开发团队，为 IBM 360/40——开发分时系统拥有分页 MMU[20]（在 IBM 系统上第一次）的改进型 360 系列机器。此后很多年，MIT 和 IBM 程序员一直保持交流。新系统拥有一个与 CTSS 非常类似的用户界面，备有名为 EXEC 的 shell 和大量的实用程序，与 Multics 及后来 Unix 使用的实用程序非常类似。

从另一层意义看来，VM/CMS 和 Unix 之间就像是游乐宫里的镜像。VM/CMS 系统的统一性理念是虚拟机，由 VM 组件提供，每台虚拟机看起来就和运行其上的物理机是一样的。它们都是抢先式多任务处理，要么运行单用户的操作系统 CMS，要么运行一个完整的多任务处理操作系统（如 MVS，Linux 或者 VM 自己）。虚拟

19　http://www.cbttape.org/cdrom.htm

20　要开发的机器和最初目标是开发定制微码的 40 系列，但是 40 机器不够强劲；生产部署转向了 360/67 系列

机对应 Unix 的进程、后台程序和仿真器，它们之间的通信通过连接一个虚拟机的虚拟穿孔机和另一个虚拟机的虚拟读卡机来完成。另外，CMS 内提供了一个叫作“CMS 管道”的分层工具环境，直接取自 Unix 的管道模型，但在结构上已经扩展到可以支持多道输入和输出。

当虚拟机之间的通讯还没明确建立时，它们是完全隔绝的。操作系统具有和 MVS 一样的高可靠性、伸缩性和安全性，而且灵活性和易用性比 MVS 要好得多。另外，CMS 中类似内核的部分不需要得到 VM 组件的信任，对它的操控是完全隔离的。

尽管 CMS 是面向记录的，但这些记录实际上等价于 Unix 文本工具所用的行。CMS 的数据库更好地集成到 CMS 管道中，而 Unix 中的大多数数据库都独立于操作系统。近年来，CMS 已经扩展到完全支持单一 Unix 规范。

CMS 采用交互式和会话式 UI 风格，和 MVS 相差很远、但和 VMS、Unix 近似，大量使用一个叫 XEDIT 的全屏幕编辑器。

VM/CMS 出现在客户端/服务器的区分之前，现今和 IBM 模拟终端一起几乎完全作为服务器操作系统使用。在 Windows 主宰桌面市场之前，VM/CMS 不仅在 IBM 内部、而且也为大型机客户站点提供字处理服务和电子邮件服务——实际上，由于 VM 早就有提供成千上万用户的伸缩性，许多机器专门安装 VM 系统，只用它运行这些应用程序。

Rexx 脚本语言支持编程的风格和 shell、awk、perl 或 python 有几分相似。因此，轻松编程（特别是系统管理员的轻松编程）在 VM/CMS 上非常重要。由于允许自由流通，管理员通常更愿意在虚拟机上而不是直接在裸机上运行产品级 MVS，因此，人们很容易获得 CMS 并充分利用其灵活性（有一些 CMS 工具可允许访问 MVS 文件系统）。

VM/CMS 在 IBM 中的历史同 Unix 在数字设备公司（DEC，他

们生产了首次运行 Unix 的硬件）中的历史惊人地相似。IBM 花了数年时间才明白自己的非正式分时系统的战略意义，与早期 Unix 社区行为非常类似的 VM/CMS 编程者社区就在那时兴起了。这些编程者分享想法和对系统的发现，最重要的是他们分享实用工具的源码。尽管 IBM 多次试图宣布 VM/CMS 结束，但这个社区——包括 IBM 自己的 MVS 系统开发者——坚持维持这个系统的存活。VM/CMS 甚至也经过和 Unix 同样的循环，从事实上的开源到闭源，再回到开源——只不过没有 Unix 开源那样彻底罢了。

然而，VM/CMS 所缺乏的是一个像 C 语言那样的东西。VM 和 CMS 都用汇编语言编写，而且一直如此。和 C 最像的是 PL/I 的各种删节版，IBM 用其进行系统编程，但从来没提供给客户。因此，尽管 360 系列已经升级到 370 系列、XA 系列，最后到现在的 z 系列，这个操作系统却仍然截止在最初架构的框框中。

自 2000 年以来，IBM 以前所未有的力度在大型机上推广 VM/CMS 系统——作为能同时容纳成千上万虚拟 Linux 机的手段。

Linux

Linux 由 Linus Torvalds 于 1991 年发明，是 1990 年后出现的新学派开源 Unix 阵营（也包括 FreeBSD、NetBSD、OpenBSD 和 Darwin）的领头羊，代表了整个阵营的设计方向。Linux 的技术趋势可视为整个阵营的典型。

Linux 并不含任何来自原始 Unix 源码树的代码，但却是一个依照 Unix 标准设计、行为像 Unix 的操作系统。在本书的其余部分，我们重点强调的是 Unix 和 Linux 的延续性。无论从技术还是从关键开发者两个方面看，这种延续性都极其紧密——但此处，我们的重点是介绍 Linux 正在前进的几个方向，这些也正是 Linux 开始与“经典”Unix 传统分道扬镳的标志。

Linux 社区的许多开发者和积极分子都有夺取足量桌面用户市

场份额的雄心壮志。这就使 Linux 的目标受众比“旧学派”Unix 广泛得多，后者主要瞄准服务器和技术型工作站市场。这一点影响了 Linux 黑客设计软件的方式。

最明显的变化就是首选界面风格的转变。最初，设计 Unix 是为了在电传打字机和低速打印终端上使用。Unix 生涯的大多数时间被用在字符型视频显示终端上，没有图形和色彩能力。大多数 Unix 程序员仍然固执地坚持使用命令行，即使大型终端用户应用程序很早就已经移植到基于 X 的 GUI 中了。这种状况也一直体现在 Unix 操作系统及应用程序的设计中。

另一方面，Linux 的用户和开发者不断自我调整来消弭非技术用户对 CLI 的恐惧。他们比旧学派 Unix、甚至同时代专有 Unix 更看重 GUI 及其工具的开发。其它开源 Unix 也在发生同样变化，变化虽小，但意义深远。

贴近终端用户的愿望使得 Linux 开发者比专有 Unix 更注重系统安装的平稳性和软件发布问题。由此产生的结果就是 Linux 的二进制包系统远比专有 Unix 的类似系统复杂，所设计的界面（2003 年只取得部分成功）更合乎非技术型用户的口味。

Linux 社区比旧学派 Unix 社区更希望将他们的软件变成能够联接其它环境的通用渠道。因此，Linux 的特色就是能支持其它操作系统特有文件系统格式的读（更常见的是）写以及联网方式。Linux 也支持同一硬件上的多重启动，并在 Linux 自身的软件中进行模拟。Linux 的长期目标是包容；Linux 模拟的目的就是为了吸收[21]。

包容竞争者的目标加上贴近终端用户的动力，促使 Linux 开发

---

[21] Linux 模拟并包容的策略与一些竞争者实施的收买并扩展的策略所产生的结果显著不同。对于初学者，Linix 并不会为了把用户锁定到增强版上而丧失与被模拟物的兼容性。

者广泛吸收非 Unix 操作系统的设计理念，甚至到了使传统 Unix 显得十分孤立的地步。Linux 应用程序采用 Windows 的.INI 格式文件进行配置是一个小例子，我们将在第 10 章予以讨论。Linux 2.5 采纳了扩展文件属性，加上其它一些特性，就可以模仿 Macintosh 的“资源分叉”语义。这也是写作本书时最近的一个重要例子。

> 但是，Linux 绘出“因为没安装对应的软件，所以打不开文件”这种 Mac 式诊断之时，就是 Linux 不再是 Unix 之日。
>
> —Doug McIlroy

其余的专有 Unix（如 Solaris、HP-UX、AIX 等）都是为庞大 IT 预算设计的庞大产品。人们愿意掏钱努力优化，追求在高端的先进硬件上达到最大效能。因为很多 Linux 部件源自 PC 爱好者，所以强调用尽量少的资源做尽可能多的事。当专有 Unix 牺牲在低端硬件上的性能而专门为多处理器和服务器集群调优时，Linux 核心开发者的选择很明确：不能为在高端硬件上获得最大性能收益，而在低端机器上增加复杂度和开销。

事实上，不难理解 Linux 用户社区中相当一部分人要从过时了的硬件中榨取有用东西的做法，就像 1969 年 Ken Thompson 对 PDP-7 一样。因此，Linux 应用程序不得不始终保持瘦小精干的体态，而这是无法在专有 Unix 下的应用软件中体验到的。

这些趋向对 Unix 整体的发展产生了影响，我们将在第 20 章回顾这个话题。

种什么籽，得什么果

我们做过尝试，选择一个现在或者过去同 Unix 一争高下的分时系统进行比较，但似乎能入围者并不多。大多数（Multics、ITS、DTSS、TOPS-IO、TOPS-20、MTS、GCOS、MPE 还有其它不

下十几种）操作系统早已消亡，已渐渐从计算机领域的集体记忆中淡出。在我们已经讨论的操作系统中，VMS 和 OS/2 也已濒临死亡，而 MacOS 已经被 Unix 的派生系统所收纳。MVS 和 VM/CMS 仅仅局限于单一的专有大型机领域。只有独立于 Unix 传统外的 Microsoft Windows 系统还算是一个真正活着的竞争对手。

我们在第一章说明了 Unix 的优势，那当然是问题的部分答案：然而，把问题反换一下：究竟 Unix 竞争者的什么劣势让它们失败，其实更有说服力。

这些竞争对手最明显的通病是不可移植性。大部分 1980 年前的 Unix 竞争者都被拴到单个硬件平台上，随着这个硬件的消亡而消亡。为什么 VMS 可以坚持这么久？值得我们作为案例研究一个原因是：VMS 成功地从最初的 VAX 硬件移植到了 Alpha 处理器（2003 年正从 Alpha 移植到 Itanium 上）。MacOS 也在 1980 年代后期成功完成了从摩托罗拉 68000 到 PowerPC 芯片的迁跃。微软的 Windows 处在计算机商品化将通用计算机市场扁平化到单一 PC 文化的时期，真是生逢其时。

自 1980 年起，对于那些要么被 Unix 压倒要么已经先 Unix 而去的其它系统，不断重现的另一个特有弱点是：不具备良好的网络支持能力。

在一个网络无处不在的世界，即使为单个用户设计的系统也需要多用户能力（多种权限组）——因为如果不具备这一点，任何可能欺骗用户运行恶意代码的网络事务都将颠覆整个系统（Windows 宏病毒只是冰山一角）。如果不具备强大的多任务处理能力，操作系统同时处理网络传输和运行用户程序的能力将被削弱。操作系统还需要高效的 IPC，这样网络程序彼此能够通信，并且能够与用户的前台应用程序通信。

Windows 在这些领域具有严重缺陷却逃脱了惩罚，这仅仅因为它们在连网变得真正重要以前就形成了垄断地位，并拥有一群已经

对机器经常崩溃和无数安全漏洞习以为常的用户。微软的这种地位并不稳定，Linux 阵营正是利用这一点成功地（于 2003 年）在服务器操作系统市场取得了重大突破。

在个人机刚刚进入全盛时期的 1980 年左右，操作系统设计者认为 Unix 和其它传统的分时系统笨重、麻烦、不适合单用户个人机这个美丽新世界，而弃之不理——根本不顾 GUI 接口往往要求改造多任务处理能力，来适应不同窗口及其部件的绑定执行线程的事实。青睐客户端操作系统的趋势非常强烈，服务器操作系统就像已经逝去的蒸汽机时代的遗物一样遭到冷落。

但是，正如 BeOS 设计者们所注意到的那样，如果不实现某些近似通用分时系统的东西，就无法满足普遍联网的要求。单用户客户端操作系统在互联网世界里不可能繁荣。

这个问题促使客户端操作系统和服务器操作系统重新汇到了一起。首先，互联网时代之前的 1980 年代晚期，人们首次尝试通过局域网进行点对点联网，这种尝试暴露了客户端操作系统设计模式的不足：网络中的数据必须放到集合点上才能实现其享，因此如果没有服务器就做不到这一点。同时，人们对 Macintosh 和 Windows 客户端操作系统的体验也抬高了客户所能容忍的最低用户体验质量的门坎。

到了 1990 年，随着非 Unix 分时系统模型的实际消亡，客户端操作系统设计者还是拿不出来多少可能解决这一挑战的方案。他们可以吸收 Unix（如 MacOS X 所做的），或通过一次一个补丁重复发明一些大致等价的功能（如 Windows），或试图重新发明整个世界（如 BeOS，但失败了）。但与此同时，各种开源 Unix 的类客户端能力不断增强，开始能够使用 GUI 并能在廉价的个人机上运行。

然而，这些压力在两类操作系统上并未达到上面描述所意味的那种对称。将服务器操作系统特性，如多用户优先权组和完全多任务处理，改装到客户端操作系统上非常困难，很可能打破对旧版本

客户端的兼容性，而且通常做出的系统既脆弱又令人不满意，不稳定也不安全。另一方面，将 GUI 应用于服务器操作系统，所出现的问题却大部分可通过灵活处理和投入更廉价硬件资源得到解决。就像造房子一样，在坚实的地基上修理上层建筑当然要比更换地基而不破坏上层建筑来得容易。

除了拥有与生俱来的服务器操作系统体系优势外，Unix 一直不明确界定自己的目标受众。Unix 的设计者和实现者从不自认为已经完全清楚 Unix 的所有潜在用途。

因此，与之竞争的客户端操作系统把自己改造成服务器操作系统，Unix 比起来更有能力把自己改造成客户端操作系统。尽管 1990 年代 Unix 的复苏有多方面的技术和经济因素，但正是这一点，使 Unix 在前述所有操作系统设计风格的讨论中最为抢眼。

模块性：保持清晰，保持简洁

> 软件设计有两种方式：一种是设计得极为简洁，没有看得到的缺陷；另一种是设计得极为复杂，有缺陷也看不出来。第一种方式的难度要大得多。
>
> 《皇帝的旧衣》，CACM 1981 年 2 月
>
> —C. A. R. Hoare

代码划分的方法有一个自然的层次体系，随着程序员必须面对的复杂度日益增加，这个体系也在演变中。一开始，一切都是一大块机器码。最早的过程语言首先引入了用子程序划分代码的概念。后来，我们发明了服务程序库，在多个程序间其享公用函数。再后来，我们发明了独立地址空间和可以相互通信的进程。今天，我们习以为常地把程序系统分布在通过成千上万英里的网络电缆连接的多台主机上。

Unix 的早期开发者也是软件模块化的先锋。在他们之前，模块化原则只是计算机科学的理论，还不是工程实践。在研究工程设计中模块经济性的《设计原理》（Design Rules）[Baldwin-Clark]

这本探路性质的著作中，作者以计算机行业的发展为研究案例，并认为，相对硬件而言，Unix 社区实际上第一个将模块分解法系统地应用到了生产软件中。毫无疑问，自从 19 世纪晚期人们采用标准螺纹以来，硬件的模块性就一直是工程技术的基石之一。

模块化原则在这里展开来说就是：要编写复杂软件又不至于一败涂地的唯一方法，就是用定义清晰的接口把若干简单模块组合起来，如此一来，多数问题只会出现在局部，那么还有希望对局部进行改进或优化，而不至于牵动全身。

相对其他程序员而言，Unix 程序员骨子里的传统是：更加笃信重视模块化、更注重正交性和紧凑性等问题。

> 早期的 Unix 程序员擅长模块化是因为他们被迫如此。操作系统就是一堆最复杂的代码。如果没有良好的架构，操作系统就会崩溃。在人们早期开发 Unix 时就犯过几次这种错，代码不得不全数报废。虽然大家可以把这些怪罪于早期的（非结构化）C 语言，但主要还是因为操作系统太复杂，太难编写。所以，我们既需要改进工具（如 C 语言的结构化），也需要养成使用工具的好习惯（如 Rob Pike 提出的编程原理），这样才能应对这种复杂性。
>
> —Ken Thompson

早期的 Unix 黑客为此在很多方面进行了艰苦的努力。1970 年的时候，函数调用开销昂贵，不是因为调用语句太复杂（PL/1.Algo），就是因为编译器牺牲了调用时间来优化其它因素，如快速内层循环（fast inner loops）。这样，代码往往就写成一大块。Ken 和其他早期 Unix 开发者知道模块化是个好东西，但是他们记得 PL/1 的经验，不愿意编写小函数，怕影响性能。

> Dennis Ritchie 告诉所有人 C 中的函数调用开销真的很小很小，极力倡导模块化。于是人人都开始编写小函数，搞模块化。然而几年后，我们发现在 PDP-11 中函数调用开销仍然昂贵，而 VAX 代码往往在“CALLS”指令上花费掉 50% 的运行

> 时间。Dennis 对我们撒了谎！但为时已晚，我们已经欲罢不能……
>
> —Steve Johnson

今天所有的编程者，无论是不是 Unix 下的程序员，都被教导要在程序的子程序层上进行模块化。有些人学会了在模块或抽象数据类型层上玩这一手，并称之为“良好的设计”。设计模式运动正在进行一项宏伟的努力，希望更进一步，找到成功的设计抽象原则，以组织大规模程序的结构。

将这些问题作一个更好的划分是一个有价值的目标，而且到处都可以找到有关模块划分的优秀方法。我们不期望太深入地涵盖与程序模块化相关的所有问题：首先，因为该论题本身就足够写整整一本（或好几本）书：其次，因为这是一本关于 Unix 编程艺术的书。

我们在此会更详细地分析 Unix 传统是如何教导我们遵循模块化原则的。本章中的例子仅限于进程单元内。我们将在第 7 章分析其它一些情形，那里，程序划分为几个协作进程是个不错的想法，我们还将讨论实现这种划分所采用的具体技术。

封装和最佳模块大小

模块化代码的首要特质就是封装。封装良好的模块不会过多向外部披露自身的细节，不会直接调用其它模块的实现码，也不会胡乱共享全局数据。模块之间通过应用程序编程接口（API）——一组严密、定义良好的程序调用和数据结构来通信。这就是模块化原则的内容。

API 在模块间扮演双重角色。在实现层面，作为模块之间的滞塞点（choke point），阻止各自的内部细节被相邻模块知晓；在设计层面，正是 API（而不是模块间的实现代码）真正定义了整个体

系。

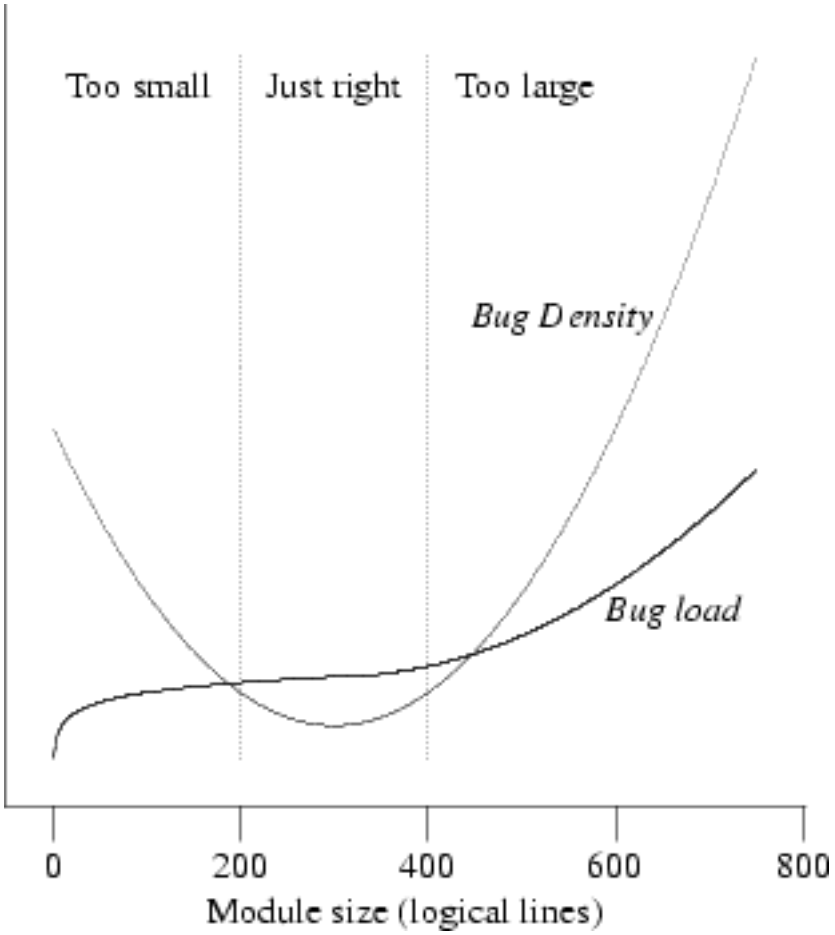
有一种很好的方式来验证 API 是否设计良好：如果试着用纯人类语言描述设计（不许摘录任何源代码），能否把事情说清楚？养成在编码前为 API 编写一段非正式书面描述的习惯，是一个非常好的办法。实际上，一些最有能力的开发者，一开始总是定义接口，然后编写简要注释，对其进行描述，最后才编写代码——因为编写注释的过程就阐明了代码必须达到的目的。这种描述能够帮助你组织思路，本身就是十分有用的模块说明，而且，最终你可能还想把这些说明做成路标文档（roadmap document），方便以后的人阅读代码。

模块分解得越彻底，每一块就越小，API 的定义也就越重要。全局复杂度和受 bug 影响的程度也会相应降低。软件系统应设计成由层次分明的嵌套模块组成，而且每个层面上的模块粒度应降至最低，计算机科学领域从二十世纪七十年代起就已经渐渐明白了这个道理（有 [Parnas] 之类文章为证）。

然而，也可能因过度划分造成模块太小。证据 [hatton97] 如下：绘制一张缺陷密度和模块大小关系图，发现曲线呈 U 形，凹面向上（见图 4-1）。跟中间大小的模块相比，模块过大或者过小都和更多的 bug 相关联。另一个观察这些同样数据的方法是，绘制每个模块的代码行数和 bug 的关系曲线图。曲线看上去大致成对数上升至平坦的“最佳点”（对应缺陷密度曲线中的最小值），然后按代码行数的平方上升（这正是人们根据 Brook 定律对整个曲线的直观预期）。[1]

---

[1] Brook 定律预言道：对一个已经延期的项目，增加程序员只会使该项目更加延期。更一般地，这个定律预言：项目成本和错误率按程序员人数的平方增长。



在模块很小时，bug 发生率也出乎意料地增多，这在大量以不同语言实现的各种系统中均是如此。Hatton 曾经提出过一个模型，将这种非线性同人类短期记忆的记忆块大小相比较[2]。这种非线性的另一种解释是，模块小时，几乎所有复杂度都在于接口；想要理解任何一部分代码前必须理解全部代码，因此阅读代码非常困难。我们将在第 7 章讨论程序划分的更高级形式；在那里，当组件进程规模更小以后，接口协议的复杂度也就决定了系统的整体复杂度。

用非数学术语来说，Hatton 的经验数据表明，假设其它所有因素（如程序员能力）都相同，200 到 400 之间逻辑行的代码是

[2] 在 Hatton 的模型中，程序员可以短期记忆的最大模块大小的微小差别对其他的效率具有倍增效应。这可能是 Fred Brooks 等人对效率的数量级（甚至更大）变化规律研究所作的最重要贡献。

“最佳点”，可能的缺陷密度达到最小。[3]这个大小与所使用的语言无关——这个结论有力支持了本书中其它地方提出的建议，即尽可能用最强大的语言和工具编程。当然，不能完全照搬这些具体数字。根据分析人员对逻辑行的理解以及其它偏好（比如注释是否剔除）的不同，代码行的统计方法会有较大差别。根据经验，Hatton 建议逻辑行与物理行之间为两倍的折算率，即最佳物理行数建议应在 400 至 800 行之间。

紧凑性和正交性

具有最佳尺寸的模块并不意味着代码有高质量。由于受到同样的人类认知限制，语言和 API（如程序库集和系统调用）也会产生 Hatton U 形曲线。

因此，在设计 API、命令集、协议以及其它让计算机工作的方法时，Unix 程序员已经学会了认真考虑另外两个特性：紧凑性和正交性。

紧凑性

紧凑性就是一个设计是否能装进人脑中的特性。测试软件紧凑性的一个很实用的好方法是：有经验的用户通常需要操作手册吗？如果不需要，那么这个设计（或者至少这个设计的涵盖正常用途的子集）就是紧凑的。

紧凑的软件工具和顺手的自然工具一样具有同样的优点：让人乐于使用，不会在你的想法和工作之间格格不入，使你工作起来更

[3] 也就是不考虑注释，一个代码模块（文件）最好小于 500 行。我以后会按照这个原则来 python 编程，看看合不合适。

有成效——完全不像那些蹩脚的工具，用着别扭，甚至还会把你弄伤。

紧凑不等于“薄弱”。如果一个设计构建在易于理解且利于组合的抽象概念上，则这个系统能在具有非常强大、灵活的功能的同时保持紧凑。紧凑也不等同于“容易学习”：对于某些紧凑设计而言，在掌握其精妙的内在基础概念模型之前，要理解这个设计相当困难；但一旦理解了这个概念模型，整个视角就会改变，紧凑的奥妙也就十分简单了。对很多人来说，Lisp 语言就是这样一个经典的例子。

> 紧凑也不意味着“小巧”。即使一个设计良好的系统，对有经验的用户来说没什么特异之处、“一眼”就能看懂，但仍然可能包含很多部分。
>
> —Ken Arnold

极少有绝对意义上紧凑的软件设计，不过从宽松一些的意义上，许多软件设计还是相对紧凑的。他们有一个紧凑的工作集：一个功能子集，能够满足专家用户 80% 以上的一般需求。实际上，这类设计通常只需要一个参考卡（reference card）或备忘单（cheat sheet），而不是一本手册。相对严格紧凑性而言，我们将此类设计称为**“半紧凑型”**。

也许最好还是用例子来阐明这个概念。Unix 系统调用 API 是半紧凑的，而 C 标准程序库无论如何都算不上是紧凑的。Unix 程序员很容易记住满足大多数应用编程（文件系统操作、信号和进程控制）的系统调用子集，但现代 Unix 上的 C 标准库却包括成百上千个条目，如数学函数等，一个程序员不可能把所有这些都记在脑中。

《魔数七，加二或减二：人类信息处理能力的局限性》（The Magical Number Seven，Plus or Minus Two: Some Limits on Our Capacity for Processing Information[Miller]）是认知心理学的基础性

文章之一（顺带一句，这也正是美国本地电话号码只有七位的原因）。这篇文章表明，人类短期记忆能够容纳的不连续信息数就是七，加二或减二。这给了我们一个评测 API 紧凑性的很好的经验法则：编程者需要记忆的条目数大于七吗？如果大于七，则这个 API 不太可能算是严格紧凑的。

在 Unix 工具软件中，make（1）是紧凑的；autoconf（1）和 automake（1）则不是。在标记语言中，HTML 是半紧凑的，DocBook（我们将在第 18 章讨论这个文件标记语言）则不是。man（7）宏是紧凑的，troff（1）标记则不是。

在通用编程语言中，C 和 Python 是半紧凑的；Perl，java，Emacs Lisp，和 shell 则不是（尤其是严格的 shell 编程，要求你必须知道其他六个工具，如 sed（1）和 awk（1）等）。C++ 是反紧凑性的——该语言的设计者已经承认，他根本不指望有哪个程序员能够完全理解 C++。有些不具备紧凑性的设计具有足够的内部功能冗余，结果程序员通过选择某个工作的语言子集就能够搞出能满足 80% 普通任务的紧凑方言。比如，Perl 就有这种伪紧凑性。此类设计存在一个固有的陷阱：当两个程序员试图就一个项目进行交流时，他们可能会发现，对工作子集的不同选择成了他们理解和修改代码的巨大障碍。

然而，不紧凑的设计也未必注定会灭亡或很糟糕。有些问题域简直是太复杂了，一个紧凑的设计不可能有如此跨度。有时，为了其它优势，如纯性能和适应范围等，也有必要牺牲紧凑性。troff 标记就是一个很好的例子，BSD 套接字 API 也是如此。把紧凑性作为优点来强调，并不是要求大家把紧凑性看作一个绝对要求，而是要像 Unix 程序员那样：合理对待紧凑性，设计中尽量考虑，决不随意抛弃。

正交性

正交性是有助于使复杂设计也能紧凑的最重要特性之一。在纯粹的正交设计中，任何操作均无副作用；每一个动作（无论是 API 调用、宏调用还是语言运算）只改变一件事，不会影响其它。无论你控制的是什么系统，改变每个属性的方法有且只有一个。

显示器就是正交控制的。你可以独立改变亮度而不影响对比度，而色彩平衡控制（如果有的话）也独立于前两个属性。想象一下，如果按亮度按钮会影响色彩平衡，这样的显示器调节起来会有多么困难：每次调节亮度之后还得调节色平衡进行补偿。更糟糕的是，如果对比度控制也影响色平衡，那么要改变对比度或色平衡同时保持另一个不变，你必须严格按照正确的方法同时调节两个旋钮。

非正交的软件设计不胜枚举。例如，代码中常见的一类设计错误出现在从某一（源）格式到另一（目标）格式进行数据读取和解析过程中。如果设计者想当然地认为源格式总是存储在某个磁盘文件中，那么他可能会编写一个打开和读取指定文件名的转换函数。但是，通常情况下，输入也完全有可能就是一个文件句柄。如果转换函数是正交设计的，例如，无需额外打开一个文件，那么以后当转换函数要处理来自标准输入、网络套接字或其它来源的数据流时，可能会省事一些。

人们通常认为 Doug Mcllroy“只做好一件事”的忠告是针对简单性的建议。但是，这句话也暗含了对正交性至少同等程度的强调。

如果一个程序做好一件事之外，顺带还做其它事情的时候既不增加系统的复杂度也不会使系统更易产生 bug，就没什么问题。我们将在第 9 章检视一个名为 ascii 的程序，这个程序能打印 ASCII 字符的同名符，包括十六进制值、八进制值和二进制值；其副作用是可以对 0-255 范围内的数字进行快速进制转换。这第二个作用并

不算违反正交性，因为所有支持该用途的特性全部是主功能所必需的，而且这样也没有增加程序文档化或维护的难度。

如果副作用扰乱了程序员或用户的思维模式，带来种种不便甚至可怕的结果（最好还是忘掉吧），这就是出现了非正交性问题。尤其在没有忘记这些副作用时，你总要被迫做额外工作来抑制或修正它们。

《程序员修炼之道》（The Pragmatic Programmer）[Hunt-Thomas] 一书中对正交性以及如何达到正交性有精彩的讨论。正如该书所指出的，正交性缩短了测试和开发的时间，因为那种既不产生副作用也不依赖其它代码副作用的代码校验起来要容易得多——需要测试的情况组合要少得多。如果正交性代码出现问题，把它替换掉而不影响系统其余部分也很容易做到。最后，正交性代码更容易文档化和复用。

重构（refactoring）概念是作为“极限编程（Extreme Programming）”学派的一个明确思想首次出现的，跟正交性紧密相关。重构代码就是改变代码的结构和组织，而不改变其外在行为。当然，自从软件领域诞生之日起，软件工程师就一直在从事这项工作，给这种做法命名并把重构的一套技术方法确定下来，则非常有效地帮助了人们集中思路。因为重构概念与 Unix 设计传统关注的核心问题非常契合，所以 Unix 开发者很快就吸收了这一术语和它的思想[4]。

Unix 的基本 API 设计在正交性方面虽不完美，但也颇为成功。比如，我们理所当然地认为能够打开文件进行写入操作，而无需为此进行排他锁定。并不是所有的操作系统都如此优雅。老式

[4] 在这一概念的奠基性著作“重构»（Refactoring）[Fowler] 一书中，作者差一点就道出了“重构的原则性目标就是提高正交性”的天机。但是由于缺少这个概念，他只能从几个不同的方向接近这个思想：比如消除重复代码和各种“坏味道”，大部分就是指一些违背正交性的做法。

（System III）的信号就不是正交的，因为信号接收的副作用是把信号处理器（signal handler）重置成缺省的“接收即崩溃”（die-on-receipt）。许多大幅修正也不是正交的，如 BSD 套接字 API，还有一些更大的修正也不是正交的，如 X window 系统的绘图库。

但是，就整体而言，Unix API 是一个很好的例子：否则，将不仅不会、也不可能这么广泛地被其它操作系统上的 C 库效仿。所以，即便不是 Unix 程序员，Unix API 也值得学习，因为从中可以学到一些关于正交性的东西。

SPOT 原则

《程序员修炼之道》（The Pragmatic Programmer）针对一类特别重要的正交性明确提出了一条原则——“不要重复自身（Don’t Repeat Yourself）”，意思是说：任何一个知识点在系统内都应当有一个**唯一**、明确、权威的表述。在本书中，我们更愿意根据 Brian Kemighan 的建议，把这个原则称为“真理的单点性（Single Point of Truth）”或者 SPOT 原则。

重复会导致前后矛盾、产生隐微问题的代码，原因是当你修改重复点时，往往只改变了一部分而并非全部。通常，这也意味着你对代码的组织没有想清楚。

常量、表和元数据只应该声明和初始化一次，并导入其它地方。无论何时，重复代码都是危险信号。复杂度是要花代价的，不要为此重复付出。

通常，可以通过重构去除重复代码；也就是说，更改代码的组织而不更改核心算法。有时重复数据好像无法避免，但碰到这种情况时，下面问题值得你思考：

- 如果代码中含有重复数据是因为在两个不同的地方必须使用两个不同的表现形式，能否写个函数、工具或代码生成程序，让

其中一个由另一个生成，或两者都来自同一个来源？

- 如果文档重复了代码中的知识点，能否从部分代码中生成部分文档，或者反之，或者两者都来自同一个更高级的表现形式？
- 如果头文件和接口声明重复了实现代码中的知识点，是否可以找到一种方法，从代码中生成头文件和接口声明？

数据结构也存在类似的 SPOT 原则：“无垃圾，无混淆”（No junk，no confusion）。“无垃圾”是说数据结构（模型）应该最小化，比如，不要让数据结构太通用，居然还能表示不可能存在的情况。“无混淆”是指在真实世界中绝对明确清晰的状态在模型中也应该同样明确清晰。简言之，SPOT 原则就是提倡寻找一种数据结构，使得模型中的状态跟真实世界系统的状态能够一一对应。

更深入 Unix 传统一步，我们可以从 SPOT 原则得出以下推论：

- 是不是因为缓存了某个计算或查找的中间结果而复制了数据？仔细考虑一下，这是不是一种过早优化；陈旧的缓存（以及保持缓存同步所必需的代码层）是滋生 bug 的温床，而且如果（实际经常是）缓存管理的开销比预想的要高，甚至可能降低整体性能[5]。
- 如果有大量重复的样板代码，是不是可以用单一的更高层表现形式生成这些代码、然后通过提供不同的细调选项生成不同个例呢？

到此，读者应该能看出一个轮廓逐渐清晰的模式。

---

[5] 不良缓存的一个典型例子是 csh（1）rehash 指令。欲了解详情可键入 man 1 csh。另一个例子参见 12.4.3。

在 Unix 世界中，SPOT 原则作为一个统一性理念很少被明确提出过——但是 Unix 传统中 SPOT 原则在各种形式的代码生成器中充分体现。我们将在第 9 章讨论这些技法。

紧凑性和强单一中心

要提高设计的紧凑性，有一个精妙但强大的方法，就是围绕“解决一个定义明确的问题”的强核心算法组织设计，避免臆断和捏造。

> 形式化往往能极其明晰地阐述一项任务。如果一个程序员只认识到自己的部分任务属于计算机科学一些标准领域的问题——这儿来点深度优先搜索，那儿来点快速排序——是不够的。只有当任务的核心能够被形式化，能够建立起关于这项工作的明确模型时，才能产生最好的结果。当然，最终用户没有必要理解这个模型。统一核心的存在本身就给人很舒服的感觉，不会出现像在使用看似无所不能的瑞士军刀式程序中非常普遍的“他们到底为什么这样做”的情形。
>
> —Doug McIlroy

这是 Unix 传统中常常被忽视的一个优点。其实，Unix 许多非常有效的工具都是围绕某个单一强大算法直接转换的一个瘦包装器（thin wrapper）。

最清楚的例子也许就是 diff（1）——一个 Unix 用于报告相关文件不同之处的工具。这个工具及其搭裆 patch（1）已经成为当代 Unix 网络分布式开发风格的核心。diff 的可贵性之一在于它很少标新立异。它既没有特殊情况，也没有令人痛苦的边界条件，因为它使用一个简单、从数学上看很可靠的序列比较方法。这导致了以下结果：

> 由于采用了数学模型和可靠的算法，Unix diff 和其仿效者形成鲜明的对比。首先，diff 的核心引擎小巧可靠，没有

> 一行代码需要维护。其次，结果清晰一致，不会出现试探法可能带来的意外。
>
> —Doug McIlroy

这样，使用 diff 的人无需完全理解核心算法，就能对 diff 在任何给定条件下的行为形成一种直觉。在 Unix 中，其它通过强大核心算法达到这种特定清晰性的著名例子非常多：

- 通过模式匹配从文件中挑选文本行的 grep（1）实用程序是一个简单包装器，围绕正则表达式（regular-expression）模式的形式代数问题（参见 8.2.2 部分的讨论）。如果它没有这个一致的数学模型，它可能就会很像最古老的 Unix 中原始的 glob（1）设计，只是一堆无法组合在一起的专门通配符。
- 用于生成语法解析器的 yacc（1）实用程序是围绕 LR（1）语法形式理论的瘦包装器。它的搭档——词法分析生成器 lex（1），则是围绕不确定有限态自动机的瘦包装器。

以上这三个程序都极少出 bug，大家认为它们绝对理所当然地应该正确运行，而且它们也非常紧凑，程序员用起来得心应手。这些良好性能只有一部分归功于长期服务和频繁使用所产生的改进，绝大部分还是因为建立在强大且被证明为正确的算法核心上，它们从一开始就无需多少改进。

与形式法相对的是**试探法**——凭经验法则得出的解决方案，在概率上可能正确，但不一定总是正确。有时我们使用试探法是因为不可能找到绝对正确的解决方案。例如，想一想垃圾邮件过滤：一个算法上完美的垃圾邮件过滤器需要完全解决自然语言的理解问题。其它一些时候，我们使用试探法是因为所有已知的形式上正确的方法开销都贵得难以想象。虚拟内存管理就是这样一个例子：虽然确实存在接近完美的解决方案，但是它们需要的运行时间太长，以至其相比试探法所能获得的任何理论上的收益优势完全被抵消掉了。

试探法的问题在于这种方案会增生出大量特例和边界情况。通常情况下，当试探法失效，如果没什么其它方法的话，你必须采用某种恢复机制作为后备。复杂度一增加，所有常见的问题都会随之而来。为了折衷，一开始就要小心使用试探法。始终要记着问一问，如果试探法以增加代码复杂性为代价，根据会获得的性能来判断一下是否值得这么做——不要猜想可能产生的性能差异，在做出决定前应该实际衡量一下。

分离的价值

本书开头，我们引用了禅的“教外别传，不立文字”。这不仅是为了追求风格上的异国情调，而是因为 Unix 的核心概念一向都有清瘦如禅般的简洁性，在围绕这些核心概念发生的历史事件中如影随形，熠熠生辉。这种特性也反映在 Unix 的基础性著作中，如《C 程序设计语言》（C Programming Language）[Kemighan-Ritchiel] 和向世人介绍 Unix 的 1974 年 CACM 论文。文中最常被人引用的一句话是这样的：“..... 限制不仅提倡了经济性，而且某种程度上提倡了设计的优雅”。要达到这种简洁性，尽量不要去想一种语言或操作系统最多能做多少事情，而是尽量去想这种语言或操作系统最少能做的事情——不是带着假想行动，而是从零开始（禅称为“初心”(beginner’s mind）或者叫“虚一心”（empty mind））。

要达到紧凑、正交的的设计，就从零开始。禅教导我们：依附导致痛苦；软件设计的经验教导我们：依附于被人忽略的假定将导致非正交、不紧凑的设计，项目不是失败就是成为维护的梦魇。

禅授超然，可以得教化，去苦痛。Unix 传统也从产生设计问题的特定、偶然的情形讲授分离的价值。抽象、简化、归纳。因为我们编制软件是为了解决问题，所以我们不可能完全超然于问题之外——但是值得费点心思，看看可以抛弃多少先入之见，看看这样

做能不能使设计变得更紧凑、更正交。这样做下来，代码复用经常由此变为可能。

关于 Unix 和禅的关系的笑话同样也是 Unix 传统中一个仍然鲜活的部分[6]。这绝非偶然。

软件是多层的

一般来说，设计函数或对象的层次结构可以选择两个方向。选择何种方向、何时选择，对代码的分层有着深远的影响。

自顶向下和自底向上

一个方向是自底向上，从具体到抽象——从问题域中你确定要进行的具体操作开始，向上进行。例如，如果为一个磁盘驱动器设计固件，一些底层的原语可能包括“磁头移至物理块”、“读物理块”、“写物理块”、“开关驱动器 LED”等。

另一个方向是自顶向下，从抽象到具体——从最高层面描述整个项目的规格说明或应用逻辑开始，向下进行，直到各个具体操作。这样，如果要为一个能处理不同介质的大容量存储控制器设计软件，可以从抽象的操作开始，如“移到逻辑块”、“读逻辑块”、“写逻辑块”、“开关状态指示”等。这和以上命名方式类似的硬件层操作的不同之处在于，这些操作在设计时就考虑到要能在不同的物理设备间通用。

以上这两个例子可视为同一类硬件的两种设计方式。在这种情况下，你的选择无非是两者取其一：要么抽象化硬件（这样，对象封装了实际事物，程序只不过是针对这些事物的操控动作列表），

---

[6] 要了解 Unix 和禅交融的最近例子，可参阅附录 D。

要么围绕某个行为模型组织代码（然后在行为逻辑流中嵌入实际执行的硬件操控动作）。

许多不同的情形中都会出现类似的选择。设想你在编写 MIDI 音序器软件，可以围绕最项层（音轨定序）或围绕最底层（切换音色或采样以及驱动波形发生器）组织代码。

有一个非常具体的方法可以考量二者的差异，那就是问问设计是围绕主事件循环（常常具备与其非常接近的高级应用逻辑）组织，还是围绕主循环可能调用的所有操作的服务库组织代码。自顶向下的设计者通常先考虑程序的主事件循环，以后才插入具体的事件。自底向上的设计者通常先考虑封装具体的任务，以后再按某种相关次序把这些东西粘合在一起。

如果要举一个更大的例子，可以考虑网页浏览器的设计。网页浏览器的顶层设计是对浏览器预期行为的规格说明：可以解析什么类型的 URL（http:，ftp: 还是 file:），可以渲染哪些类型的图像，是否可以或者带哪些限制来支持 Java 或 Javascript 等等。与这个顶层意图相对应的实现层是浏览器的主事件循环；在每个周期内，这个循环等待、收集、分派用户的动作（例如点击网页链接或在某个域内键入字符）。

但是，网页浏览器要正常工作还必须调用大量域原语操作。其中一组跟建立连接、通过连接发送数据和接收响应有关。另一组则是浏览器将使用的 GUI 工具包操作。然而，可能还有第三组集合，即“将接收的 HTML 从文本转换为文档对象树”的解析机制。

从哪端开始设计相当重要，因为对端的层次很可能受到最初选择的限制。尤其是，如果程序完全自顶向下设计，你很可能发现自己陷入非常不舒服的境地，应用逻辑所需要的域原语和真正能实现的域原语无法匹配。另一方面，如果程序完全自底向上设计，很可能发现自己做了许多与应用逻辑无关的工作——或者，就像你想要造房子，却仅仅只设计了一堆砖头。

自从二十世纪六十年代有关结构化程序设计的论战后，编程新手往往被教导以“正确的方法是自顶向下”：逐步求精，在拥有具体的工作码前，先在抽象层面上规定程序要做些什么，然后用实现代码逐步填充。当以下三个条件都成立时，自顶向下不失为好方法：（a）能够精确预知程序的任务，（b）在实现过程中，程序规格不会发生重大变化，（c）在底层，有充分自由来选择程序完成任务的方式。

这些条件容易在相对接近最终用户和软件设计的较上层——应用软件编程——中得到满足。但即便如此，这些前提也常常满足不了。在用户界面经过最终用户测试前，别指望能提前知道什么算是字处理软件或绘图程序的“正确”行为方式。如果纯粹地自顶向下编程，常常产生在某些代码上的过度投资效应，这些代码因为接口没有通过实际检验而必须废弃或重做。

为了应对这种情况，出于自我保护，程序员尽量双管齐下——一方面以自顶向下的应用逻辑表达抽象规范，另一方面以函数或库来收集底层的域原语，这样，当高层设计变化时，这些域原语仍然可以重用。

Unix 程序员继承了一个居于系统程序设计核心的传统，在这一传统中，底层的原语是硬件层操作，后者特性固定且极其重要。因此，出于后天学得的本能，Unix 程序员更倾向于自底向上的编程方式。

无论是否是系统程序员，当你用一种探索的方式编程，想尽量领会你还没有完全理解的软件、硬件抑或真实世界的现象时，自底向上法看起来也会更有吸引力。它给你时间和空间去细化含糊的规范，同时也迎合了程序员身上人类通有的懒惰天性——当必须丢弃和重建代码时，与之相比，如果用自顶向下的设计，需要抛弃的代码往往更多。

因此实际代码往往是自顶向下和自底向上的综合产物。同一个

项目中经常同时兼有自顶向下的代码和自底向上的代码。这就导致了“胶合层”的出现。

胶合层

当自顶向下和自底向上发生冲突时，其结果往往是一团糟。顶层的应用逻辑和底层的域原语集必须用胶合逻辑层来进行阻抗匹配(impedance match)。

Unix 程序员几十年的教训之一就是：胶合层是个挺讨厌的东西，必须尽可能薄，这一点极为重要。胶合层用来将东西粘在一起，但不应该用来隐藏各层的裂痕和不平整。

在网页浏览器这个例子中，胶合层包括渲染代码 (rendering code)，它使用 GUI 域原语将从发过来的 HTML 中解析出的文档对象绘制成平面的可视化表达——即显示缓冲区中的位图。渲染代码作为浏览器中最易产生 bug 的地方而臭名昭著。它的存在，是为了解决 HTML 解析（因为形式不良的标记太多了）和 GUI 工具包（可能未必具有真正需要的原语）中存在的问题。

网页浏览器的胶合层不仅要协调内部规范和域原语集，而且还要协调不同的外部规范：HTTP 标准化的网络行为、HTML 文档结构、各种图形和多媒体格式以及用户对 GUI 的行为预期。

一个容易产生 bug 的胶合层还不是设计所能遇到的最坏命运。如果设计者意识到胶合层的存在，并试图围绕自身的一套数据结构或对象把胶合层组织成一个中间层，结果却导致出现两个胶合层——一个在中间层之上，另一个在中间层之下。那些天资聪慧但经验不足的程序员特别容易掉进这种陷阱：他们将每种类别（应用逻辑、中间层和域原语集）的基本集都做得很好，就像教科书上的例子一样漂亮，结果却因为整合这些漂亮代码所需的多个胶合层越来越厚，而最终在其中苦苦挣扎。

薄胶合层原则可以看作是分离原则的升华。策略（应用逻辑）应该与机制（域原语集）清晰地分离。如果有许多代码既不属于策略又不属于机制，就很有可能除了增加系统的整体复杂度之外，没有任何其它用处。

实例分析：被视为薄胶合层的 C 语言

C 语言本身就是一个体现薄粘合层有效性的良好例子。

上个世纪九十年代后期，Gerrit Blaauw 和 Fred Brooks 在《计算机体系：概念和演化》(Computer Architecture: Concepts and Evolution) [BlaauwBrooks] 一书中提出，每一代计算机的体系结构，从早期的大型机到小型机、工作站再到 PC，都在趋近同一种形式。技术年代越靠后，设计越接近 Blaauw 和 Brooks 所称的“经典体系”：二进制表示、平面地址空间、内存和运行期存储（寄存器）的区分、通用寄存器、定长字节的地址解析、双地址指令、高位字节优先[7]以及大小一致为 4 位或 6 位整数倍（6 位系列现在已经不存在了）的数据类型。

Thompson 和 Ritchie 将 C 语言设计成一种结构汇编程序，可为理想化的处理器和存储器体系服务，他们期望这种体系能有效建立在大多数普通计算机上。幸运的是，他们的理想化处理器模型机是 PDP-11——一款设计非常成熟、优雅的小型机，非常接近 Blaauw & Brook 的经典体系。凭借敏锐的判断力，Thompson 和 Rithcie 拒绝在其语言中加入 PDP-11 不匹配的少数特性（比如低位优先字节序）中的绝大多数[8]。

---

[7] 高位字节优先 (big-endian) 和低位字节优先 (little-endian) 术语指比特在机器字内解析顺序的架构选择。虽然没有规范的位置，但你在网上搜索“On Holy Wars and a Plea for Peace”，会找到有关这个论题的一篇经典而有趣的文章。

[8] 人们普遍以为自增自减符特性被 C 语言采用是因为它们代表了

PDP-11 成为接下来几代微处理器架构的重要模型。结果证明，C 语言的基本抽象相当优美地反映出了经典体系。这样，C 语言一开始就非常适合微处理器，而且随着硬件更紧密地向经典架构靠拢，C 语言不仅没有随其假设的过时而失去价值，反而更加适合微处理器了。这种硬件向经典体系会聚的非常著名的例子就是：1985 年后 Intel 的 386 机器用平面存储地址空间代替了 286 糟糕的分段内存寻址。跟 286 相比，纯 C 语言实际上更适合 386。

计算机架构的实验性时代在二十世纪八十年代中期结束，同期，C 语言（和近亲后代 C++）作为通用程序设计语言所向无敌，两者在时间上并非巧合。C 语言，作为经典体系之上一个薄而灵活的胶合层，在经过了 20 年后，现在看来似乎可以算是其定位的结构汇编程序中的最佳设计。除了紧凑、正交和分离（与最初设计时的机器架构分离），C 语言还拥有我们将在第 6 章讨论的透明性这一重要特性。C 语言之后的少数语言设计（是否比 C 语言更好还有待证明），为了不被 C 语言所吞并，不得不进行大的改动（比如引进垃圾收集功能等），以和 C 语言保持功能上的足够距离。

这段历史很值得回味和了解，因为 C 语言向我们展示了一个清晰、简洁的最简化设计能够多么强大。如果 Thompson 和 Ritchie 当初没有这么明智，他们设计的语言也许能完成更多任务，但要依赖更强的前提，永远都无法满意地从原始的硬件平台移植出去，也必将随着外部世界的改变而消亡。但相反的是，C 语言一直生机勃勃——而 Thompson 和 Ritchie 所树立的榜样从此影响了 Unix 的开发风格。正如法国作家、冒险家、艺术家和航空工程师安东尼 • 德 • 圣埃克苏佩里（Antoine de Saint-Exupéry）在论飞机设计时所说的：“La perfection est atteinte non quand il ne reste rien à ajouter, mais quand il ne reste rien à enlever”（完美之道，不在无可增加，而在无可删减）。

PDP-11 的机器指令，这其实没有根据。按照 Dennis Ritchie 的说法，在 PDP-11 出现之前，这些操作符就在前辈 B 语言中出现了。

Ritchie 和 Thompson 坚信该格言。即便当早期 Unix 软件所受的种种资源限制得到缓解之后很久，他们仍努力使 C 语言成为尽可能薄的“硬件之上的胶合层”。

> 以前每当我要求在 C 语言中加一些特别奢侈的功能时，Dennis 就对我说，“如果你需要 PL/1，你知道到哪里去找”。他不必和那些说着：“但我们需要在销售材料中加一个卖点”的销售人员打交道。
>
> —Mike Lesk

在标准化之前最好先有个有效的参考实现，C 语言的历史在这方面教了我们一课。我们将在第 17 章讨论 C 语言和 Unix 标准的发展时再谈这个话题。

程序库

Unix 编程风格强调模块性和定义良好的 API，它所产生的影响之一就是：强烈倾向于把程序分解成由胶合层连接的库集合，特别是共享库（在 Windows 和其它操作系统下叫做“动态连接库”(DLL)。

如果谨慎而聪明地处理设计，那么常常可以将程序划分开来，一个是用户界面处理的主要部分（策略），另一个是服务例程的集合（机制），中间不带任何胶合层。当程序要进行图形图像、网络协议包、硬件接口控制块等多种数据结构的具体操作处理时，这种方法特别合适。《可复用库架构的守则和方法》（The Discipline and Method Architecture for Reusable Libraries）[Vo] 一书中收集了 Unix 传统中关于体系的一些不错的通用性建议，尤其适合这种程序库的资源管理。

在 Unix 下，通常是清晰地划分出这种层次，并把服务程序集中在一个库中并单独文档化。在这样的程序中，前端专门解决用户

界面和高层协议的问题。如果设计更仔细一些，可以将原始的前端分离出来，用适于不同用途的其它部件代替。通过实例研究，你还会发现其它一些优势。

这捎带引起了一个小问题。在 Unix 世界里，作为“程序库”发布的库必须携带练习程序 (exerciser program)。

> API 应该随程序一起提供，反之亦然。如果一个 API 必须要编写 C 语言代码来使用，考虑到 C 代码不能方便地从命令行调用，刚这个 API 学习和使用起来就更困难。反之，如果接口唯一开放、文档化的形式是程序，而无法方便地从 C 程序中调用这些接口，也会非常痛苦——例如，老版本 Linux 中的 route(1)。
>
> —Henry Spencer

除了学习起来更容易外，库的练习程序常常可以作为优秀的测试框架。因此，有经验的 Unix 程序员并不仅仅把这些练习程序看作是为库使用者提供便利，也会认为代码应已经过很好的测试。

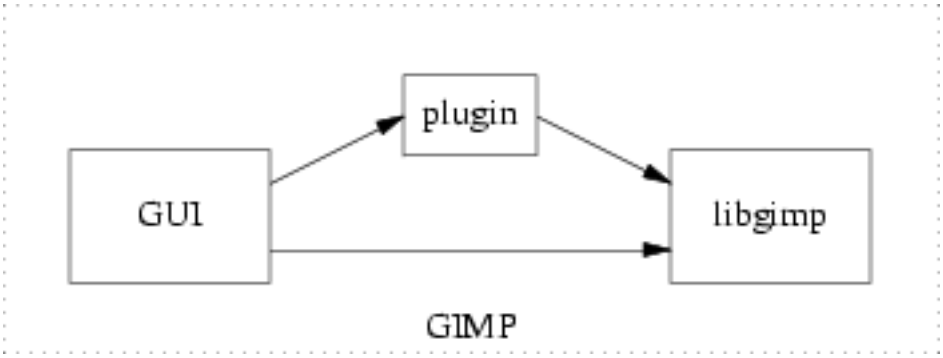
库分层的一个重要形式是**插件**，即拥有一套已知入口、可在启动以后动态从入口处载入来执行特定任务的库。这种模式必须将调用程序作为文档详备的服务库组织起来，以使得插件可以回调。

实例分析：GIMP 插件

GIMP（GNU 图像处理程序，GNU Image Manipulation program）是一个由交互方式 GUI 驱动的图形图像编辑器。但是 GIMP 被做成了一个图像处理和辅助程序的库，由一个相对较薄的控制层代码调用。驱动码知道 GUI，但不直接知道图像格式；反过来，程序库程序知道图像格式和图像操作，但不知道 GUI。

这个库层次已经文档化了（而且，实际上已作为“libgimp”发布，供其它程序使用）。这意味着 C 程序写成的所谓“插件”可以

由 GIMP 动态载入，然后调用该库进行图像处理，实际上掌握了和 GUI 同一级别的控制权（参见图 4.2）。



插件可用来完成多种专用转换，如色图调整（colormap hacking）、模糊和去斑；可用于读写非 GIMP 自带的文件格式；也可用于扩展功能，如编辑动画和窗口管理器主题：通过在 GIMP 内核中编写图像调整逻辑脚本，还可实现其他多种图像调整处理的自动化。万维网中有各种 GIMP 插件的注册中心。

虽然大多数 GIMP 插件都是小巧简单的 C 程序，但是也有可能编制一个插件让库 API 能被脚本语言调用。我们将在第 11 章分析“多价程序”模式时讨论这种可能性。

Unix 和面向对象语言

1980 年代中期起，大多数新的语言设计都已自带了对“面向对象”(OO) 编程的支持。回想一下，在面向对象的编程中，作用于具体数据结构的函数和数据一起被封装在可视为单元的一个对象中。相反，非 OO 语言中的模块使数据和作用于该数据的函数的联系变得相当无规律，而且模块间还经常互相泄漏数据或内部细节。

OO 设计理念的价值最初在图形系统、图形用户界面和某些仿真程序中被认可。使大家惊讶并逐渐失望的是，很难发现 OO 设计在这些领域以外还有多少显著优点。其中原因值得我们去探究一番。

在 Unix 的模块化传统和围绕 OO 语言发展起来的使用模式之间，存在着某些紧张对立的关系。Unix 程序员一直比其他程序员对 OO 更持怀疑态度，原因之一就源于**多样性原则**。OO 经常被过分推崇为解决软件复杂性问题的唯一正确办法。但是，还有其它一些原因，这些原因值得我们在第 14 章讨论具体 OO（面向对象）语言之前作为背景问题加以探讨，这也将有助于我们对 Unix 的一些非 OO 编程风格特征有更深刻的认识。

前面我们提到，Unix 的模块化传统就是薄胶合层原则，也就是说，硬件和程序顶层对象之间的抽象层越少越好。这部分是因为 C 语言的影响。在 C 语言中模仿真正的对象很费力。正因为这样，堆砌抽象层是一件非常累人的事。这样，C 语言中的对象层次倾向于比较平坦和透明。即使 Unix 程序员使用其它语言，他们也愿意继续沿用 Unix 模型教给他们的薄胶合/浅分层风格。

OO 语言使抽象变得很容易——也许是太容易了。OO 语言鼓励“具有厚重的胶合和复杂层次”的体系。当问题域真的很复杂、确实需要大量抽象时，这可能是好事，但如果编码员到头来用复杂的办法来做简单的事情——仅仅是因为他们能够这样做，结果便适得其反。

所有的 OO 语言都显示出某种使程序员陷入过度分层陷阱的倾向。对象框架和对象浏览器并不能代替良好的设计和文档，但却常常被混为一谈。过多的层次破坏了透明性：我们很难看清这些层次，无法在头脑中理清代码到底是怎样运行的。简洁、清晰和透明原则统统被破坏了，结果代码中充满了晦涩的 bug，始终存在维护问题。

可能正是因为许多编程课程都把厚重的软件分层作为实现表达原则的方法来教授，这种趋势还在恶化。根据这种观点，拥有很多类就等于在数据中嵌入了很多知识。问题在于，胶合层中的“智能数据”却经常不代表任何程序处理的自然实体——仅仅只是胶合物而已。（这种现象的一个确定标志就是抽象子类或混入 (mix-in's)

类的不断扩散。）

OO 抽象的另一个副作用就是程序往往丧失了优化的机会。例如，a+a+a+a 可以用 a*4 来表示，如果 a 是整数，也可以表示成 a«2。但是如果构建了一个类并重新定义了操作符，就根本没什么东西可表明运算操作的交换律、分配律和结合律。既然不能查看对象内部，就不可能知道两个等价表达式中哪一个更有效。这本身并不是在新项目中避免使用 OO 技法的正当理由，那样只会导致过早优化。但这却是在把非 OO 代码转换为类层次之前需要三思而后行的原因。

Unix 程序员往往对这些问题有本能的直觉。在 Unix 下，OO 语言没能代替非 OO 的主力语言，如 C、Perl（其实有 OO 功能，但用得不多）和 shell 等，这种直觉似乎也是原因之一。跟其它正统领域相比，Unix 世界对 OO 语言的批判更直接了当；Unix 程序员知道什么时候不该用 OO；就算用 OO，他们也尽可能保持对象设计的整洁清晰。正如《网络风格的元素》（The Elements of Networking Style）一书的作者在另一个略有不同的背景下所说的 [Padlipshy]：“如果你知道自己在做什么，三层就足够了；但如果你不知道自己在做什么，十七层也没用。”

OO 在其取得成功的领域（GUI、仿真和图形）之所以能成功，主要原因之一可能是因为在这些领域里很难弄错类型的本体问题。例如，在 GUI 和图形系统中，类和可操作的可见对象之间有相当自然的映射关系。如果你发现增加的类和所显示的对象没有明显对应关系，那么很容易就会注意到胶合层太厚了。

Unix 风格程序设计所面临的主要挑战就是如何将分离法的优点（将问题从原始的场景中简化、归纳）同代码和设计的薄胶合、浅平透层次结构的优点相结合。

我们将在第 14 章探讨面向对象的语言时继续讨论并应用以上一些观点。

模块式编码

模块性体现在良好的代码中，但首先来自良好的设计。在编写代码时，问问自己以下这些问题，可能会有助于提高代码的模块性：

- 有多少全局变量？全局变量对模块化是毒药，很容易使各模块轻率、混乱地互相泄漏信息[9]。
- 单个模块的大小是否在 Hatton 的“最佳范围”内？如果回答是“不，很多都超过”的话，就可能产生长期的维护问题。知道自己的“最佳范围”是多少吗？知道与你合作的其他程序员的最佳范围是多少吗？如果不知道，最好保守点儿，坚持 Hatton 最佳范围的下限。
- 模块内的单个函数是不是太大了？与其说这是一个行数计算问题，还不如说是一个内部复杂性问题。如果不能用一句话来简单描述一个函数与其调用程序之间的约定，这个函数可能太大了[10]。

> 就我个人而言，如果局部变量太多，我倾向于拆分子程序。另一个办法是看代码行是否存在（太多）缩进。我几乎从来不看代码长度。
>
> —Ken Thompson

- 代码是不是有内部 API——即可作为单元向其他人描述的函数调用集和数据结构集，并且每一个单元都封装了某一层次的函

9　全局变量同时也意味着代码不能重入；也就是说，同一进程的多个实例可能彼此干涉。

10　很多年前，我从 Kernighan 和 Plauger 的《编程风格的元素》(The Elements of Programming Style) 一书中学到一个非常有用的原则，就是在函数原型之后立即写一行注释。每个函数都这样，决无例外。

数，不受其它代码的影响？好的 API 应是意义清楚，不用看具体如何实现就能够理解的。对此有一个经典的测试方法：通过电话向另一个程序员描述。如果说不清楚，API 很可能就是太复杂，设计太糟糕了。

- API 的入口点是不是超过七个？有没有哪个类有七个以上的方法？数据结构的成员是不是超过七个？
- 整个项目中每个模块的入口点数量如何分布[11]？是不是不均匀？有很多入口点的模块真的需要这么多入口点吗？模块复杂性往往和入口点数量的平方成正比——这也是简单 API 优于复杂 API 的另一个原因。

你可能会发现，如果把以上这些问题和第 6 章关于透明性和可见性问题的清单加以此较，将颇有启发性。

[11] 收集这种信息有一个简便的方法，就是分析 etags(1) 或 ctags(1) 等工具程序生成的标记文件。