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] 官方英文网站，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。

---

[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。

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 McIlroy, 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。

---

[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。

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。

---

[3] 同抑制，而钳制有胁迫之意。

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。

> 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 McIlroy 在 [Mcll-

roy78] 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. 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。

---

[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。”

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 11 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，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

[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。

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。

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

[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. Hoare。

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。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]

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

[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。

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。

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

---

[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。

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 McIlroy 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。

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

[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。

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 McIlroy[McIlroy91] 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

4

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

[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。

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 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

6　古代歃血为盟，盟主执牛耳。

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、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

[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”

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]

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

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

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]

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

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。

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。

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

[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。

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

[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。

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

[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。

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。

---

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

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。

---

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

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而上。“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

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

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。

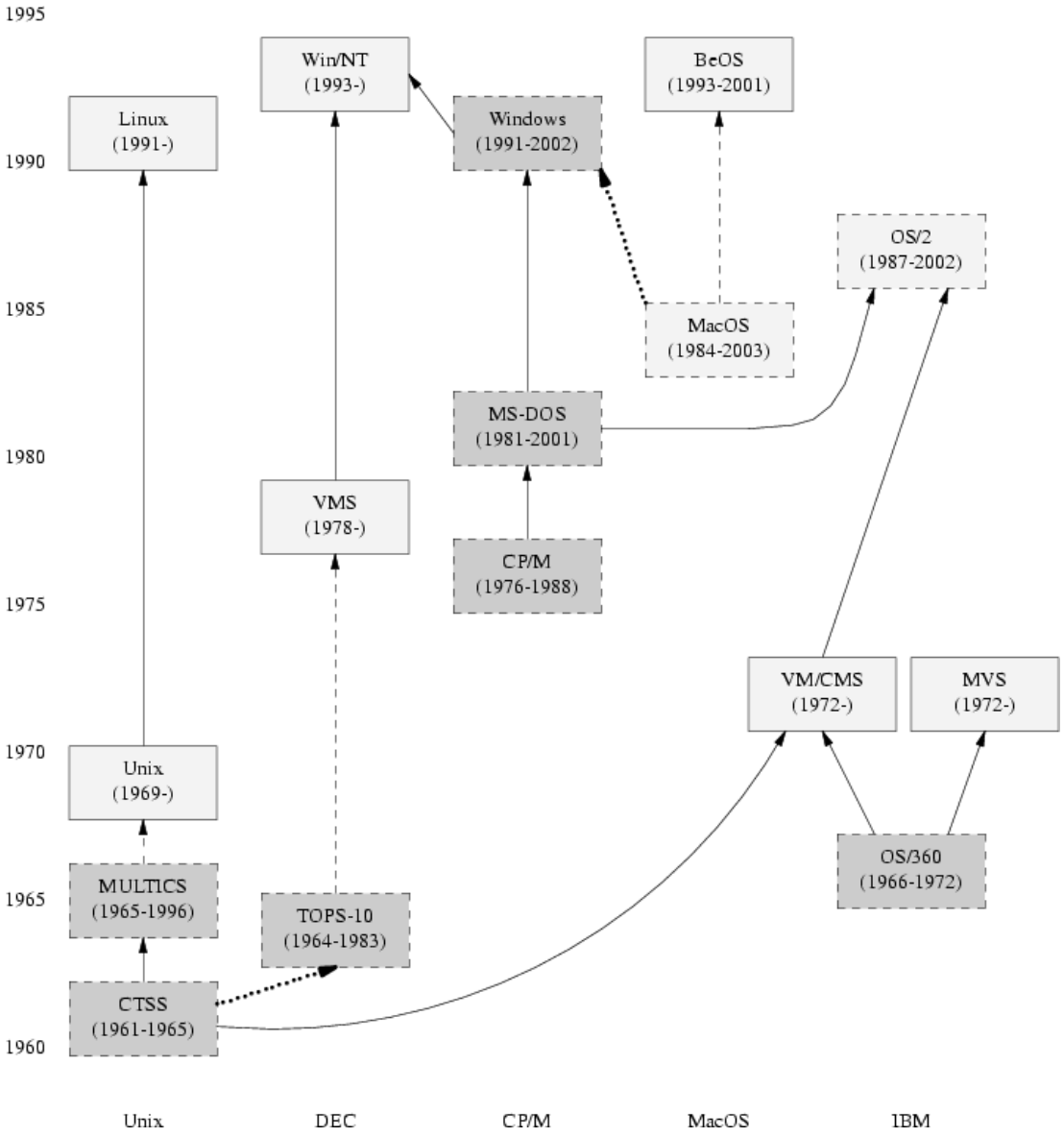
实线箭头表示存在起源关系或很强的设计影响（例如，后开发的系统 API 有意通过逆向工程以匹配先前的系统）；短画线表示重大的设计影响；点线表示微弱的设计影响。不是所有的起源关系被开发者认可；实际上，有些出于法律或企业策略的原因被官方否认，但在业界其实是公开的秘密。

“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]

---

[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。

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



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。

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。

---

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

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

[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。

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）。

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

---

[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。

[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。

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 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。

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/

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：

- 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。

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”。

- 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

[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。

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

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——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。

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

[19] http://www.cbttape.org/cdrom.htm

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。

---

[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

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。

---

[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。

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。

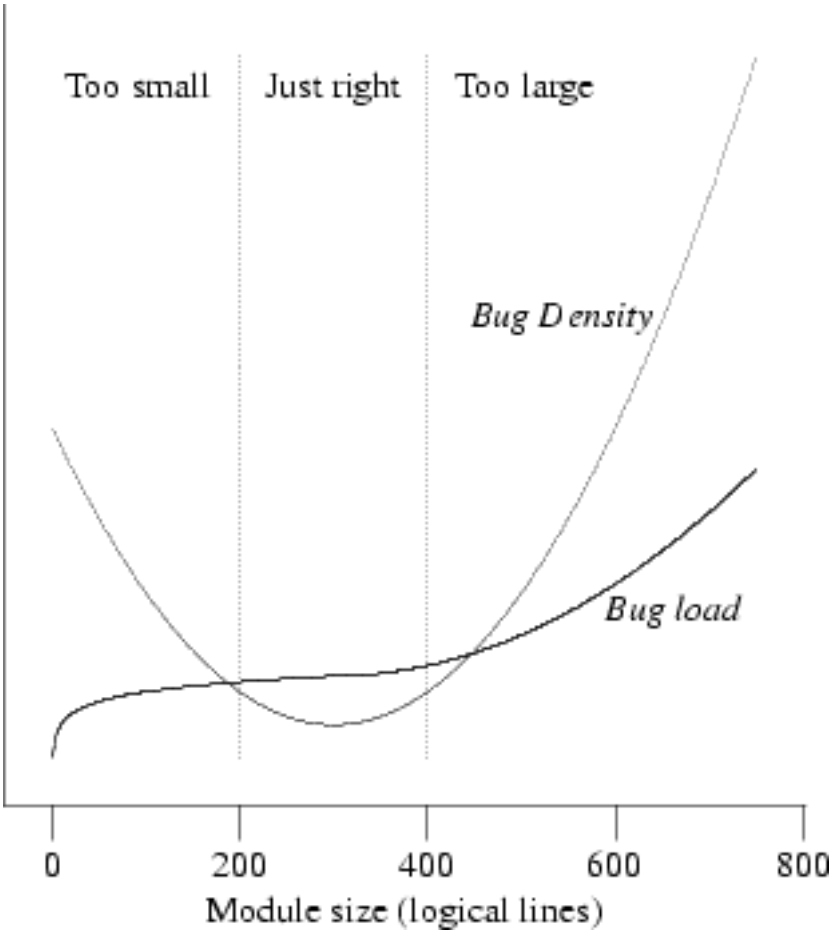
API 在模块间扮演双重角色。在实现层面，作为模块之间的滞塞点（choke point），阻止各自的内部细节被相邻模块知晓；在设计层面，正是 API（而不是模块间的实现代码）真正定义了整个体系。

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]



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

[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。

[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

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“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。

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。

[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。

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 McIlroy“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（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

[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。

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。

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

[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。

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），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）。

6　要了解 Unix 和禅交融的最近例子，可参阅附录 D。

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]。

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

---

[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 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。

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）。

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

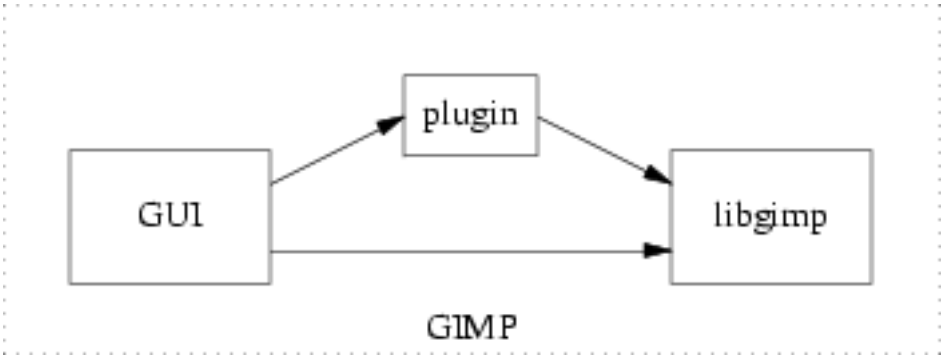
除了学习起来更容易外，库的练习程序常常可以作为优秀的测试框架。因此，有经验的 Unix 程序员并不仅仅把这些练习程序看作是为库使用者提供便利，也会认为代码应已经过很好的测试。

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。

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。

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。

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。

文本化：好协议产生好实践

> 众所周知，人类几千年前就已经发明了算盘之类的计算设备。但很少有人知道，人类第一次使用通用计算机协议是在《旧约》中——当时摩西用控制海中止了埃及人的进程。
>
> —Tom Galloway

我们将在本章分析 Unix 传统所教导的两种不同而又紧密相关的设计：设计将应用数据存储在永久存储器中的文件格式，和在协作程序中（可能会通过网络）传递数据和命令的应用协议。

这两种设计的共通之处在于：两者都与内存数据结构的序列化有关。对于计算机程序的内部操作而言，一个复杂数据结构最简便的表达就是所有字段都用机器自带的数据格式（如整数的二进制补码）来表示，而所有指针都是实际的存储地址（相对于名称引用而言）。但是这些表示法并不适合数据的存储和传输；数据结构的存储地址一旦离开存储器就毫无意义，发送未经处理的原生数据格式又会因为不同机器采用不同约定（如高位字节序对低位字节序，或 32 位对 64 位）而产生传输数据的互用问题 (interoperability)。

为了便于数据的传输和存储，像链表这样的数据结构，其可遍历的准空间部署需要平整化或序列化成字节流表达，以便日后能从这个表达中恢复数据结构。序列化（保存）操作有时也称为**列集** (marshaling)，其反向操作（载入）称为散集 (unmarshaling)。这些术语通常使用在面向对象的语言中，如 C++，Python 或者 Java 中对对象的操作，但同样可用于其它一些操作，如图形编辑器将图形文件载入内存并在修改后存盘。

C 和 C++ 程序员要维护的代码中，进行列集和散集操作的特别代码占了很大比例——即便所选择的序列化表达很简单，如二进制结构的转储（非 Unix 环境下的一种通用技巧）也是如此。Python 和 Java 等现代语言往往内置了散集和列集函数，可应用于任何对象或代表对象的字节流，大大减少了工作量。

但是由于种种原因，这些天真的方法常常不尽人意，原因既包括我们上面提到的机器间的互用问题，也包括对其它工具不透明这一负面特征。如果应用程序是网络协议，出于经济性考虑，可能会要求内部数据结构（比如携带源地址和目标地址的信息）不是序列化成单个的大型数据包 (blob)，而是序列化成接收设备可拒绝的一系列尝试性处理事务或信息（这样一来，如果目的地址无效，则较大的整块信息就会被拒收）。

互用性、透明性、可扩展性和存储/事务处理的经济性——这些都是设计文件格式和应用协议时需要考虑的重要方面。互用性和透明性要求我们在此类设计中要重点考虑数据表达的清晰问题，而不是首先考虑实现的方便性和可能达到的最高性能。既然二进制协议很难扩展和干净地抽取子集，可扩展性当然也青睐文本化协议。事务处理的经济性有时则会提出相反的要求——但我们应看到，首先考虑这个标准就是一种过早优化，不这么做往往是明智选择。

最后，我们必须注意数据文件格式与常用于设置 Unix 程序启动选项的运行控制文件之间的区别。最根本的区别是（偶尔也有例外，如 GNU Emacs 的配置接口）程序通常不修改自己的运行控制文件——信息流是单向的，从启动时的文件读取流向应用程序的设置。相反，数据文件格式的属性同命名资源联系在一起，应用程序既可能读也可能写。配置文件通常都可以手工编辑，体积很小，而数据文件通常由程序生成，多大都有可能。

历史上，Unix 对这两种表达采用过相关但又不同的约定。第 10 章将论述控制文件的各种约定：本章仅论述数据文件的约定。

文本化的重要性

管道和套接字既可以传输文本也可以传输二进制数据。但是，我们将在第 7 章看到的例子却都是文本化的，理由非常充分：那就是我们在第 1 章引用的 Doug McIlroy 的建议。文本流是非常有用的通用格式，因为人无需专门工具就可以很容易地读写和编辑文本流，这些格式是透明的（或可以设计成透明的）。

同时，正是文本流的限制帮助了强化封装：因为文本流不鼓励内容丰富、编码结构密集的复杂表达，也不提倡程序互相干涉内部状态。我们在第 7 章结尾部分讨论 RPC 时继续这个话题。

当你很想设计一个复杂的二进制文件格式，或一个复杂的二进制应用协议时，通常，明智的做法是躺下来等待这种感觉过去。如果担心性能问题，就在应用协议之上或之下的某个层面上压缩文本协议流，最终产生的设计会比二进制协议更干净，性能可能也更好（文本压缩起来更好、更快）。

> Unix 历史上有一个二进制格式的反面教材，那就是设备无关的 *troff* 程序读取设备信息二进制文件的方式，当时可能出于速度方面的考虑。最初的实现以一种不太可移植的方法从文本描述中生成该二进制文件。为了能在新机器上快速移植础 *ditroff* 而避免重新编写二进制文件的麻烦，我把它剥离了出来，只让 *ditroff* 读取文本文件。读取文件的代码经过精心编制，速度的损失可以忽略不计。
>
> —Henry Spencer

设计一个文本协议往往可以为系统的未来省不少力气。一个具体原因就是格式本身不能表示数字域的范围。二进制格式通常指定了给定值的分配位数，要扩展位数非常困难。例如，IPv4 的地址是 32 位的，要将地址位数扩展到 128 位（如 IPv6）就需要进行大修补[1]。相反，如果在文本格式中需要更大的

[1] 传说一些早期的飞机订票系统对乘客人数只分配一个字节。随着第一架载客超过 255 名的波音 747 投入使用，可以想象这些系统碰到了多少麻烦。

值，直接写就是了。也许某个特定程序不能接受那个范围内的值，但是跟修改存储在格式中的所有数据相比，修改这个程序通常要容易得多。

使用二进制协议的唯一正当理由是：如果要处理大批量的数据集，因而确实关注能否在介质上获得最大位密度，或是非常关心将数据转化为芯片核心结构所必须的时间或指令开销。大图像和多媒体数据的格式有时可以算是前者的例子，对延时有严格要求的网络协议有时则可以算是后者的例子。

> SMTP 或类 HTTP 的文本协议存在的问题则相反。这些协议往往占据昂贵的带宽资源，解析速度很慢。最小的 X 请求是 4 个字节：最小的 HTTP 请求大约是 100 个字节。X 请求，包括已摊销的传输成本，100 条指令的数量级就可执行了：一度，一位 Apache[Web 服务器] 开发者自豪地声称他们已经精简到了 7000 条指令。对图形而言，输出时带宽就是一切；硬件设计已使得图形卡母线成为如今限制小操作的唯一瓶颈，因此，如果协议不想成为更糟糕的瓶颈，最好设计得非常紧凑。这是极端情况。
>
> —Jim Gettys

这些问题在 X 以及其它极端情况下也存在——比如，为支持特大图形而设计图形文件格式，但却往往只是另一种过早优化热。文本格式的位密度未必一定比二进制格式低多少；毕竟，它们还是用了八位字节中的七位。一旦你需要生成测试加载、或需要检查程序生成的格式实例并想弄明白个中究竟，本来无需解析文本所带来的收益往往马上就全部损失殆尽。

另外，设计紧凑二进制格式的思路往往不能够兼顾干净扩展的要求。X 的设计者就有这方面的教训：

> 从目前的 X 框架来看，我们确实没有设计出足够好的结构，使得对协议微小的扩展仍会对它造成影响；当然，有时我们可以做到这一点，但如果有一个更好的框架会更好。
>
> —Jim Gettys

当认为找到一种极端情况，有足够理由使用二进制文件格式或协议时，需仔细考虑扩展性，并在设计中为以后发展留出余地。

实例分析：Unix 口令文件格式

在许多操作系统中，验证用户登录并开始用户会话所必需的用户数据都是不透明的二进制数据库。相反，在 Unix 中，这种数据是文本文件，采用一行一条、字段用冒号分隔的记录格式。

例 5.1 是几行随机选择的文件行：

```
games:*:12:100:games:/usr/games:
gopher:*:13:30:gopher:/usr/lib/gopher-data:
ftp:*:14:50:FTP User:/home/ftp:
esr:0SmFuPnH5JlNs:23:23:Eric S. Raymond:/home/esr:
nobody:*:99:99:Nobody:/:
```

即使不知道字段的语义，我们也能发现这些数据在二进制格式中很难压缩得更紧。要达到冒号分隔符的功能至少需要相同的空间（通常是字节数或 NUL）。每个用户的记录要么需要终止符（不太可能比一个新行符更短），要么很浪费地补齐到定长。

如果知道数据的实际语义，则通道二进制编码节省空间的实际可能性几乎不存在。数字形式的用户 ID（第三）和组 ID（第四）字段都是整数，这样，大多数机器上的一个二进制表达至少需要 4 个字节，比文本格式表达 999 以下数字所需要的长度更长。不过，让我们暂且忽略这些，假设是最佳情形，即数字域在 0 到 255 范围内。

我们可以压缩数字域（第三字段和第四字段），把这些数字域的位长缩小到用单字节表示，口令字符串（第二字段）采用 8 位编码。在本例中，这样可以节省 8% 左右的空间。

这 8% 的假定低效率却带给我们很多好处：可以避免武断地限制数字域范围，可以使我们能够使用自己选择的任何老式文本编辑器修改口令文件，而无

需编制专用工具来编辑二进制文件（虽然在口令文件这个例子本身，我们需要对并发编辑特别小心），而且还让我们能够用 grep(1) 这样的文本流工具对用户帐号信息进行特别的搜索、过滤和报告。

我们的确要十分小心，不要在任何文本字段中嵌入冒号。良好的做法应该是这样：告诉文件先用换码符嵌入冒号再写代码，然后告诉文件读取代码对其解释。对此，Unix 传统偏爱使用反斜杠。

通过字段位置而不是明确的标记来传达结构信息使得这种格式的读写都很轻松，但是有些死板。如果一个键所关联的属性集要发生改动，那么以下描述的标记格式可能是更好的选择。

既然一般情况下很少读取[2]，也不经常修改，所以经济性不是口令文件一开始就要考虑的主要因素。既然文件中的不同数据（特别是用户 ID 和组 ID）不会从原始机器上搬移出去，互用性也不是问题。因此很明显，对口令文件而言，遵循透明性原则才是正确的选择。

实例分析：.newsrc 格式

Usenet 新闻是一个全球性分布式公告牌系统 (BBS)，比今天的 P2P 网络要早 20 年。它使用的信息格式与 RFC 822 电子邮件信息格式非常相似，只不过不是直接发送给个人接受者，而是发给主题组。所有在入网站点上张贴的文章先转发到登记为友邻的站点上，最终发送到整个网内的所有站点。

几乎所有的 Usenet 读者都理解.newsrc 文件，该文件记录了使用者已经阅读过哪些信息。尽管该文件名字很像一个运行控制文件，但不仅启动时要读取该文件，而且通常在新闻阅读器运行结束时还要更新该文件。自 1980 年左右出现第一个新闻阅读器以后，.newsrc 格式就固定了下来。例 5.2 是.newsrc 文件中具有代表性的一段代码。

.newsrc 实例

[2] 口令文件通常在每个用户对话的登录阶段才读一次，之后 ls(1) 之类的实用程序为了将数字用户 ID 和组 ID 映射成名称才偶尔读取这些口令文件。

```
rec.arts.sf.misc! 1-14774,14786,14789
rec.arts.sf.reviews! 1-2534
rec.arts.sf.written: 1-876513
news.answers! 1-199359,213516,215735
news.announce.newusers! 1-4399
news.newusers.questions! 1-645661
news.groups.questions! 1-32676
news.software.readers! 1-95504,137265,137274,140059,140091,140117
alt.test! 1-1441498
```

每行都为以第一个字段为名的新闻组设置属性。新闻组名称之后紧跟一个字符，表明文件对应的用户目前是否订阅了该组；冒号表示订阅，惊叹号表示没有订阅。其余部分是一系列逗号分隔的文章编号或文章编号范围，表明用户已经阅读过哪些文章。

非 Unix 程序员也许会自然而然地去试图设计一个快速二进制格式，其中每个新闻组的状态采用固定的长二进制记录或由一系列内含长度字段的自描述二进制信息包来表示。这种二进制表示的要点在于：在成对字长字段内用二进制数据来表示范围，目的是避免启动时解析所有范围表达式的开销。

这种布局也许读写都能比文本格式快，但是会产生其它问题。固定记录长度这种简单的实现会造成人为限制新闻组名称的长度，（更严重的是）限制已读文章数量范围的最大值。用一种更复杂的二进制包格式可避免这种长度限制，但用户无法自行查看或编辑——而当需要重新设置某个新闻组中的某个已读状态字段时，这种能力非常有用。而且，这种格式还不一定能移植到其它类型的机器上。

最初的新闻阅读器设计者舍经济性而取透明性和可操作性。当然，反过来从经济性角度来考虑，这种取舍也不无道理。由于.newsrc 文件有可能变得很大，某些新型阅读器（GNOME 的 Pan 阅读器）就使用对速度优化的专用格式来避免启动等待。但对其他实现者而言，文本表达在 1980 年看起来就是很好的折衷方案，而且随着机器速度的提高和存储器价格的下降，这种选择现在

愈发显得明智。

实例分析：PNG 图形文件格式

PNG（可移植网络图形）是位图图形的一种文件格式。PNG 更像 GIF，而不像 JPEG，其不同之处在于采用了无损压缩，并为艺术线条 (line art) 和图标而不是照片图像的应用程序进行了优化。高质量的文档和开源参考库可从其站点http://www.libpng.org/pub/png上获得。

PNG 格式是二进制文件格式中一个经过周密设计的优秀例子。既然图形文件包含了大量的数据，如果像素数据用文本格式来存储的话，尺寸和网络下载时间都会显著提高，因此二进制格式非常合适。传输经济性是要考虑的主要问题，透明性则牺牲了[3]。但是，设计者对互用性非常谨慎：PNG 格式指定了字节顺序、整数的字长、优先顺序，和（但缺少）字段间的填充。

PNG 文件由一系列字节块 (chunk) 构成，每个都是自描述格式，以块类型名和块长度开头。由于这种组织形式，PNG 不需要版本号。随时都可以增加新的块类型：块类型名称中的第一个字母告知使用 PNG 的软件当前块是否可被安全忽略。

PNG 文件头同样值得研究。它设计得非常聪明，能使各种常见的文件损坏情况（如 7 位传输连接，或 CR 字符和 LF 字符的损坏）很容易被发现。

PNG 标准精确全面，编写得非常好，可以作为如何撰写文件格式标准的范例来使用。

数据文件元格式

数据文件元格式是一套句法和词法约定，这套约定或者已经正式标准化，或者已经通过实践得到了充分的确定，已有标准服务库来处理列集和散集操

[3] 别搞错，PNG 格式支持另一种透明性——透明像素。

作。

Unix 已经形成或采纳了适合多种应用程序的不同元格式。尽可能使用这些元格式（而不是标新立异自己的格式）是个好习惯。第一个好处是使用服务库可以避免编写大量的用户解析代码和生成代码。但最重要的好处还是开发者甚至很多用户都能立即认出这些格式，有亲切感，这就减少了学习新程序的磨合成本（friction cost）。

在以下讨论中，当我们说到“传统 Unix 工具”时，我们指 grep(1)、sed(1)、awk(1) 和 cut(1) 这些文本搜索和变换工具的组合。对以上工具所提倡的面向行格式的解析，Perl 和其他脚本语言通常自带支持功能。

以下就是一些可以作为典范使用的标准格式。

DSV 风格

DSV 代表“Delimiter-Separated Values（**分隔符分隔值**）”。我们在文本元格式引用的第一个例子/etc/passwd 文件就是一个使用冒号作为值分隔符的 DSV 格式。在 Unix 中，对字段值可能包含空格的 DSV 格式，冒号是默认的分隔符。

/etc/passwd 格式（每个记录一行，字段用冒号分隔）是 Unix 中非常传统的格式，经常用于处理表列数据。其它经典的例子包括描述安全组的/etc/group 文件和在操作系统不同运行级别中控制 Unix 服务程序启动和关闭的/etc/inittab 文件。

这种风格的数据文件一般应通过反斜杠（\）转义符支持在数据域中包含冒号。更为普遍的是，读取这种文件的代码可通过忽略反斜线转义的换行符支持连续记录，并且允许通过 C 风格的反斜杠转义符嵌入非打印字符数据。

当数据为列表、名称（在首字段）为关键字、而且记录通常很短（少于 80 个字符）时，这种格式最适用。这种格式和传统的 Unix 工具配合得很好。

有时候也可以看到冒号以外的字段分隔符，如管道字符“|”，甚至用 ASCII NUL。Unix 的旧学派做法偏爱 TAB，这可在 cut(1) 和 paste(1) 的默认设置中反映出来。但随着格式设计者逐渐意识到，TAB 和 SPACE 在视觉上无法区别而引起了很多令人恼火的小麻烦，这种做法也逐渐改变了。

这种格式之于 Unix，就像 CSV（逗号分隔值）格式之于 Unix 世界外的 Microsoft Windows 和其它操作系统。Unix 中很少用到以逗号分隔字段、双引号用来转义逗号、没有连续行的 CSV 格式。

事实上，Microsoft 版 CSV 是一个如何设计文本文件格式的典型反面例子。问题首先出现在字段正好含有分隔字符（在这种情况下是逗号）的情况中。Unix 的方法是简单的用反斜杠转义分隔符，用双反斜杠表示反斜杠字面值。在解析文件时，这种设计只要检查一种特殊情况（转义符），发现转义符时只要一个操作（解析跟在转义符后的字符）。后者不仅方便了分隔符的处理，而且还能自由处理转义符和新行符。CSV 则相反，如果字段值中存在分隔符，就将整个字段值包括在双引号内。如果字段值包含双引号，整个字段值也得包括在双引号内，字段中的单个双引号需要重复两遍才能表明自己并不结束整个字段。

到处使用特殊情况所带来的不良结果是双重的。首先，分析程序的复杂度（以及 bug 的易发性）提高了。其次，由子格式规定既复杂又不明确，不同实现对边缘情况的处理也不同。有时，通过在一行的最后一个字段前使用双引号来支持连续行——但只有部分产品这么做！在微软自己的应用软件，有时甚至是同一个应用程序的不同版本之间（Excel 就是最明显的例子），CSV 文件都存在不兼容的情况。

RFC 822 格式

RFC 822 格式源自互联网电子邮件信息采用的文本格式；（在被 RFC 2822 取代前）RFC 822 一直是描述这种格式的主要互联网 RFC。MIME（多用途网际邮件扩充协议，Multipurpose Internet Mail Extension）提供了在 RFC 822 格式信息中嵌入类型化二进制数据的方法（在网上搜索以上

这些名称的任何一个都可以找到相关标准）。

在这种元格式中，记录属性每行存放一个，以类似邮件头字段名的标记命名，用冒号后接空白作为结束。字段名不得包含空格；通常用横线代替空格。该行的其余部分都是属性值，除了结尾的空格和换行。以 tab（制表符）或 whitespace（空格符）开始的物理行被解释为当前逻辑行的延续。空行可能被解释为记录的结束，也可能表明接下来是非结构化的文本。

在 Unix 中，对那些带属性的或任何与电子邮件类似的倍息，这都是传统而且首选的文本元格式。一般来说，这种格式非常适合具有不同字段集合而字段中数据层次又扁平（没有递归或树形结构）的记录。

Usenet 使用的就是这种格式，万维网使用的 HTTP1.1（以及后续版本）也使用这种格式。这种格式非常便于人工编辑。在属性搜索上，传统的 Unix 搜索工具仍能使用，只不过寻找记录边界要比“每行一个记录”的格式要多费些周折。

RFC 822 格式的一个弱点是，当一个文件中有不止一个 RFC 822 信息或记录时，记录边界可能不太明显——可怜的死脑筋的计算机如何知道一条信息的非结构化正文结束，而下一个记录头开始的地方在哪里呢？历史上已经存在过好几个不同的分隔邮箱中信息的约定。每条信息的第一行以字符串“From”和发送者信息开头的这种最古老、受到最广泛支持的方法并不适合其它类型的记录；这种方法也要求转义信息文本行以“From”开头（通常用“>”）——这种做法经常引起混淆。

有些邮件系统使用那些不太可能出现在信息中的控制符作为分隔行，如连续使用几个 ASCII 01 （control-A）字符。MIME 标准通过在邮件头中包含一个确定的信息长度避开了这个问题，但这是一个不太稳妥的解决方案，一旦对信息进行了手工编辑，这种解决方案很容易出问题。更好的解决方案参见本章下面描述的 record-jar 风格。

看看自己的电子邮箱就可以找到 RFC 822 格式的例子。

Cookie-Jar 格式

Cookie-jar 格式是 fortune(1) 程序为随机引用数据库而使用的一种格式。这种格式很适用记录只是一堆非结构化文本的情况。这种格式简单使用跟随%% 的新行符（或者有时只有一个%）作为记录分隔符。例 5.3 是来自电子邮件签名引用文件的部分例行。

例 5.3 fortune 文件实例

```
”Among the many misdeeds of British rule in India, history will look
upon the Act depriving a whole nation of arms as the blackest.”
        -- Mohandas Gandhi, ”An Autobiography”, pg 446
%
The people of the various provinces are strictly forbidden to have
in their possession any swords, short swords, bows, spears, firearms,
or other types of arms. The possession of unnecessary implements
makes difficult the collection of taxes and dues and tends to foment
uprisings.
        -- Toyotomi Hideyoshi, dictator of Japan, August 1588
%
”One of the ordinary modes, by which tyrants accomplish their
purposes without resistance, is, by disarming the people, and making
it an offense to keep arms.”
        -- Supreme Court Justice Joseph Story, 1840
```

寻找记录分隔符时接受% 后的空格是个好做法，有助于解决人为编辑的错误。更好的做法就是使用%%，并忽略从%% 到行结束处的所有文本。

> cookie-jar 分隔符最初是“%%\n”。我当时希望找个能比“%”更显眼的东西。事实上，任何“%%”之后的内容都作注释处理（至少我是这样写的）。
>
> —Ken Arnold

简单的 cookie-jar 格式适用于词以上结构没有自然顺序，而且结构不易区别的文本段，或适用于搜索关键字而不是文本上下文的文本段。

Record-Jar 格式

cookie-jar 记录分隔符和 RFC 822 记录元格式结合得非常好，产生一种我们称之为“record-jar”的格式。如果文本格式要支持显式字段数目可变的多重记录，众望所归的方法就是采用例 5.4 类似的格式。

例 5.4 以 record-jar 格式表达的三颗行星基本数据

```
Planet: Mercury
Orbital-Radius: 57,910,000 km
Diameter: 4,880 km
Mass: 3.30e23 kg
%%
Planet: Venus
Orbital-Radius: 108,200,000 km
Diameter: 12,103.6 km
Mass: 4.869e24 kg
%%
Planet: Earth
Orbital-Radius: 149,600,000 km
Diameter: 12,756.3 km
Mass: 5.972e24 kg
Moons: Luna
```

当然，记录分隔符也可以是一个空行，但是由“%%\n”构成的一行更为清晰，也不大可能在编辑时无意产生（两个可打印字符比一个好，因为这样不可能由单个笔误引起）。在类似这样的格式中，直接忽略空行是个不错的办法。

如果记录中含有部分非结构化文本，record-jar 格式就非常接近邮箱格式。在这种情况下，关键有一个定义良好的转义分隔符的方法，这样分隔符才能出现在文本中；否则，读取记录的程序总有一天会卡在形式不良文本部分上。本书指出了和字节填充（本章后面部分将有描述）类似的一些技巧。

Record-jar 格式适合于那些类似 DSV 文件、但又有可变字段数目而且可能伴随无结构文本的字段属性关系集合。

XML

XML 语法类似于 HTML，非常简单——尖括号括起的（<>）标签和“&”记号引导字面值序列。它几乎和纯文本标记一样简单，但又能表达递归嵌套的数据结构。XML 只是一种低级的语法，需要文档类型定义（例如 XHTML）和相关的应用逻辑赋予其语义。

XML 非常适合复杂的数据格式（旧学派 Unix 传统会为此使用类似 RFC 822 的节格式），尽管对简单的数据来说未免有些大材小用。它尤其适合那些 RFC 822 元格式不太好处理、有复杂递归或嵌套数据结构的格式。对这种格式的详细介绍，可参见《XML in a Nutshell》一书 [Harold-Means].

> 设计文本格式最难处理的问题是引句（quoting）、空格符和其它低级语法细节。用户文件格式常常因为语法结构上的轻微错误而不能跟类似格式匹配。使用 XML 之类的标准格式，可以由标准程序库来校验并解析，解决了这些问题中的绝大部分。
>
> —Keith Packard

例 5.5 是一个基于 XML 的配置文件简单实例。它是 Linux 下随开源 KDE office suite（套装办公软件）发布的 *kdeprint* 工具的一部分，描述了从图像转换到 PostScript 的过滤操作选项，以及如何将它们映射成过滤器命令行参数。另一个有益示例请参见第 8 章对 *Glade* 的讨论。

XML 的一个优势在于经常无需知道数据语义，仅通过语法检查就能发现形式不良、损坏或错误生成的数据。

XML 最严重的问题是无法很好和传统的 Unix 工具协作。读取 XML 格式的软件需要 XML 解析器，这就意味着需要庞大复杂的程序。同样，XML 本身也相当庞大，要在所有的标记中找到数据很困难。

XML 占据明显优势的应用领域是文档文件的标记格式（我们将在第 18 章予以更详细的讨论）。在大块纯文本中，此类文件的标记往往比较稀疏，因此 Unix 传统工具仍能出色完成简单的文本搜索和转换。

例 5.5 XML 实例

```
<?xml version="1.0"?>
<kprintfilter name="imagetops">
    <filtercommand
           data="imagetops %filterargs %filterinput %filteroutput" />
    <filterargs>
        <filterarg name="center"
                   description="Image centering"
                   format="-nocenter" type="bool" default="true">
            <value name="true" description="Yes" />
            <value name="false" description="No" />
        </filterarg>
        <filterarg name="turn"
                   description="Image rotation"
                   format="-%value" type="list" default="auto">
            <value name="auto" description="Automatic" />
            <value name="noturn" description="None" />
            <value name="turn" description="90 deg" />
        </filterarg>
        <filterarg name="scale"
                   description="Image scale"
                   format="-scale %value"
                   type="float"
```

```
                        min="0.0" max="1.0" default="1.000" />
        <filterarg name="dpi"
                   description="Image resolution"
                   format="-dpi %value"
                   type="int" min="72" max="1200" default="300" />
    </filterargs>
    <filterinput>
        <filterarg name="file" format="%in" />
        <filterarg name="pipe" format="" />
    </filterinput>
    <filteroutput>
        <filterarg name="file" format="> %out" />
        <filterarg name="pipe" format="" />
    </filteroutput>
</kprintfilter>
```

这些领域间有一个很有趣的沟通桥梁，就是 PYX 格式——面向行的 XML 转换，可由传统的 Unix 面向行文本工具进行修改，然后再无损转换成 XML。在网上搜索“Pyxie”即可找到相关资源。xmltk 工具包则采取相反办法，提供类似 *grep*(1) 和 *sort*(1) 的面向流工具来过滤 XML 文档；在网上搜索“xmltk”即可找到。

选择 XML 可以简化问题，也可能使问题复杂化。对它的大肆吹捧很多，但不要不加批判地采用或拒绝，否则就会成为时尚的牺牲品。请谨慎选择，牢记 KISS 原则。

Windows INI 格式

许多微软的 Windows 程序都使用类似例 5.6 的文本数据格式。这个例子将名为 account、directory、numeric_id 和 developer 的可选资源和名

为 python、sng、fetchmail 和 py-howto 的项目关联在一起。如果某个指定的输入项没有提供值，则 DEFAULT 输入项将提供相应值。

例 5.6 .INI 文件实例

```
[DEFAULT]
account = esr

[python]
directory = /home/esr/cvs/python/
developer = 1

[sng]
directory = /home/esr/WWW/sng/
numeric_id = 1012
developer = 1

[fetchmail]
numeric_id = 18364

[py-howto]
account = eric
directory = /home/esr/cvs/py-howto/
developer = 1
```

这种风格的数据文件格式并不是 Unix 自带的，但在 Windows 影响下，一些 Linux 程序（特别是 Samba，一种在 Linux 系统上获取 Windows 文件共享的工具套件）也开始支持这种格式。这种格式可读性好，设计得不错，但和 XML 一样，不能与 *grep*(1) 或常规 Unix 脚本工具很好地配合使用。

如果数据围绕指定的记录或部分能够自然分成“名称－属性对”两层组织结构，.INI 格式非常适用。但这种格式并不适合数据存在完全递归树形结构的

情况（XML 更适合）。而对于简单的名称－值关系列表，这种格式又是大材小用（这时应使用 DSV 格式）。

Unix 文本文件格式的约定

Unix 关于文本数据格式应该怎样的传统由来已久。这些约定大多来源于我们刚刚讨论过的 Unix 标准元格式中的一个或多个格式。若非确有特殊原因，最好还是遵循这些约定。

我们将在第 10 章讨论程序运行控制文件使用的一套不同约定。但大家应该注意，以下一些原则（尤其在词法级别上，字符如何组合成标记的规则）也同样适用这套约定。

- **如果可能，以新行符结束的每一行只存一个记录**。这样用文本流工具提取记录就非常容易。为了和其它操作系统交换数据，最好让文件格式的解析器不受行结束符是 LF 还是 CR-LF 的影响。在这种格式中，习惯上忽略结尾的空白，以防范常见的编辑错误。

- **如果可能，每行不超过 80 个字符**。这样使格式可以在普通尺寸的终端视窗上浏览。如果很多记录一定要超过 80 个字符，考虑使用分节格式（stanza format）（见下文）。

- **使用“#”引入注释**。能在数据文件中嵌入注解和说明会非常好。最好是把它们作为文件结构的一部分，便可被知道这种格式的工具保存下来。对于解析时不保存的说明，惯例上采用“#”作为起始字符。

- **支持反斜杠约定**。支持嵌入不可打印控制字符的最自然方法，就是解析 C 语言风格的反斜杠转义——\n 表示新行，\r 表示回车，\t 表示制表符，\b 表示退格，\f 表示走纸，\e 表示 ASCII escape (27)，\nnn 或 \onnn 或 \0nnn 表示八进制值为 nnn 的字符，\xnn 表示十六进制值为 nn 的字符，\dnnn 表示十进制值为 nnn 的字符，\\ 表示实际意义上的反斜杠。还有一个较新但也应当遵守的约定是使用 \unnn 表示十六进制的 Unicode 字面值。

- **在每行一条记录的格式中，使用冒号或任何连续的空白作为字段分隔符**。冒号约定似乎起源于 Unix 的口令文件。如果某个字段必须包含分隔符，使用反斜杠前缀进行转义。

- **不要过分区别 tab 和 whitespace**。否则，当用户编辑器的 tab 设置不同时，会产生很多令人头痛的麻烦。这条原则是治愈头痛的良方。况且，一般来说，眼睛很难区别 tab 和 whitespace。仅使用 tab 作为分隔符尤其容易产生问题；相反，允许使用连续的 tab 和空格作为分隔符却非常有效。

- **优先选用十六进制而不是八进制**。和三位的八进制数字相比，两位或四位的十六进制数字更容易直观地与字节以及今天的 32 位和 64 位字对应起来；而且，效率也或多或少高一点。强调该准则是因为 *od*(1) 等一些较老的 Unix 工具违反了这条准则，这是较老的 PDP 小型机的机器语言指令字段大小所产生的历史遗留问题。

- **对于复杂的记录，使用“节（stanza）”格式：一个记录若有多行，就使用%%\n 或%\n 作为记录分隔符**。在人们肉眼检查文件时，这种分隔符是非常有用而且直观的边界标志。

- **在节格式中，要么每行一个记录字段，要么让记录格式和 RFC 822 电子邮件头类似，用冒号终止的字段名关键字作为引导字段**。当字段经常空缺或者超过 80 个字符，或者当记录很稀疏时（如经常有空字段），适用第一二种方案。

- **在节格式中，支持连续行**。解释文件时，或者抛弃空格符之后的反斜杠，或者将空格符之后的新行符解释为单个空格；这样，一个很长的逻辑行就能够折叠成多个很短（容易编辑！）的物理行。在这些格式中，习惯上忽略结尾的空格，可防范常见的编辑错误。

- **要么包含一个版本号，要么将格式设计成相互独立的自描述字节块**。哪怕只存在一丁点格式发生改变或扩展的可能性，也要包含一个版本号，这样代码才能够有条件地在所有版本上正确运行。换句话说，将格式设计成自描述字节块，无需立即破坏旧代码就可以增加新的块类型。

- **注意浮点数取整问题**。由于所用转换库质量的不同，浮点数从二进制格式转换成文本格式再转换回二进制格式时可能会有精度损失。如果列集/散集的结构中包含浮点数，应该从两个方向都测试一下转换。如果看上去任何一个方向的转换都可能存在取整误差，做好将浮点字段作为未处理的二进制格式或字符串编码形式转储的准备。如果在 C 语言或调用了 C printf/scanf 的语言中编程，C99 的%a 指示符可以解决这个问题。
- **不要仅对文件的一部分进行压缩或二进制编码**。见下文……

文件压缩的利弊

许多现代的 Unix 项目，如 OpenOffice.org 和 AbiWord，现在都使用 *zip*(1) 或 *gzip*(1) 压缩的 XML 作为数据文件格式。压缩的 XML 综合了空间经济性和文本格式的一些优势——尤其是避免了二进制格式常常必须要为那些特定情况下（如特殊选项或超大范围）可能用不到的信息分配空间的问题。但目前对此还存在争论，也正是这种争论引发了本章讨论的一些主要折衷方案。

一方面，实验表明，经过压缩的 XML 文件通常比 Microsoft Word 自带的二进制文件格式明显要小，虽然大家可能认为二进制文件占用的空间更小。原因同 Unix“就做好一件事”的基本哲学原理有关。创建一个简单的工具来做好压缩，要比仅对文件某些部分进行特别压缩更有效，原因在于，压缩工具可以扫描所有数据，然后找到信息中的所有重复部分进行压缩。

同时，将表现形式的设计和具体压缩的方法分离，将来就可能只要对实际文件解析做最少量修改便可以使用不同的压缩方法——或许根本就不需要任何修改。

从另一方面来看，压缩确实在某种程度上损害了透明性。尽管人可以根据上下文推测解开压缩文件是否会看到有用的东西，但是直到 2003 年中期，file(1) 之类的工具仍然还无法看穿这个“包装层”。

可能有人会提倡那些没有这么结构化的压缩格式——比方说，不要 *zip*(1)

提供的内部结构和自识别头部块，直接用 *gzip*(1) 压缩 XML 数据。尽管使用类似 *zip*(1) 的格式能解决识别问题，但对于用比较简单脚本语言编写的程序来说，解码这些文件会相当棘手。

这些解决方案中的任何一种（纯文本，纯二进制或压缩文本）都可能是最佳方案，具体取决于对存储经济性、可显性或让浏览工具编写起来尽可能简单等问题的权衡考虑。上述讨论并非要鼓吹哪一种方法比其它方法更好，而是就如何考虑清楚各种选择方案和设计出折中方案提出一些建议。

人们一直说，真正的 Unix 式解决方案也许是用 *file*(1) 透过压缩看文件前缀——如果不行，就围绕 *file*(1) 写一个 shell 脚本，对压缩内容执行 gunzip(1) 再看。

应用协议设计

我们将在第 7 章讨论“将复杂应用程序划分成几个协作进程、通过应用程序专用命令集或协议通信”的优点。所有将数据文件格式设计成文本格式的好理由同样适用于应用程序专用协议的设计。

如果应用协议是文本式的，而且仅凭肉眼就能很容易地分析，那么很多好事情就更容易实现了。事务转存更容易解释。测试负载也更容易编写。

服务器进程通常由 *inetd*(8) 之类的统一控制程序（harness programs）调用，其方式是服务器程序从标准输入中接收命令，然后将响应发送到标准输出。我们将会在第 11 章更详细地描述这种“CLI 服务器”模式。

CLI 服务器的命令集是为达到简洁性而设计的，这种服务器程序有一个可贵的特性，就是测试人员能够直接向服务器进程键入命令来探知软件的工作情况。

另一个需要牢记在心的问题是端对端（end-to-end）设计守则。每一个协议设计者都应该读一读经典的《系统设计中的端对端论》（End-to-End Arguments in System Design）[Saltzer]。人们经常会严肃地对究竟协议

栈哪一层应该处理安全和认证之类的功能提出问题。这篇文章为如何考虑这个问题提供了一些很好的概念性手段。除此以外，还有第三个问题，就是为获得良好的性能而设计应用协议。我们将在第 12 章更详细讨论这个问题。

1980 年以前，互联网应用协议的设计传统一直独立于 Unix 发展[4]。但自那以后，这些传统已经完全融入了 Unix 实践。

下面我们将研究三个使用最广泛，也是被广大互联网 hacker 看作典范的应用协议实例：SMTP、POP3 和 IMAP，来说明互联网的风格。这三个协议分别致力于邮件传输（和万维网一起构成网络最重要的两个应用）的不同方面，而所涉及的问题（传输消息、设置远程状态、报告错误状态）对非电子邮件应用协议也颇具普遍意义，并且通常也可采用类似的技巧来处理。

实例分析：SMTP，一个简单的套接字协议

例 5.7 是 RFC 2821 描述的 SMTP 协议（简单邮件传送协议）中的一个处理实例。在这个例子中，C: 行由发送邮件的邮件传输代理（MTA）发送，S: 行由接收邮件的 MTA 返回。用斜体字强调的是注释，并非事务处理的实际部分。

例 5.7 SMTP 会话实例

```
C: <client connects to service port 25>
C: HELO snark.thyrsus.com                sending host identifies self
S: 250 OK Hello snark, glad to meet you   receiver acknowledges
C: MAIL FROM: <esr@thyrsus.com>          identify sending user
S: 250 <esr@thyrsus.com>... Sender ok    receiver acknowledges
C: RCPT TO: cor@cpmy.com                 identify target user
S: 250 root... Recipient ok              receiver acknowledges
C: DATA
```

[4] 互联网协议的行结束符通常是 CR-LF，而不是 Unix 的单 LF，这就是这段前 Unix 历史的遗留痕迹。

```
S: 354 Enter mail, end with ”.” on a line by itself
C: Scratch called.  He wants to share
C: a room with us at Balticon.
C: .                                    end of multiline send
S: 250 WAA01865 Message accepted for delivery
C: QUIT                                 sender signs off
S: 221 cpmy.com closing connection      receiver disconnects
C: <client hangs up>
```

邮件就是这样在互联网机器上传输的。注意以下特征：请求的命令参数格式，应答由状态码和紧接其后的指示信息构成，以及“DATA”命令的有效数据部分以一个只有单个“．”的行结束。

SMTP 是互联网上仍在使用的最古老的两三个应用协议之一。这个协议简单有效经受住了时间的考验。我们在这里重点指出的几个特征，也经常在互联网其它协议中出现。如果说设计良好的互联网应用协议有个原型的话，那么这个原型一定是 SMTP。

实例分析：POP3，邮局协议

另一个经典的互联网协议是 POP3，即邮局协议（Post Office Protocol）。这个协议也用于邮件传输，但是 SMTP 是邮件发送者启动事务处理的“推（push）”协议，而 POP3 是邮件接收者启动事务处理的“拉（pull）”协议。不连续访问互联网的用户（如拨号连接）可以让他们的邮件存在一个邮箱机器上，然后使用 POP3 连接将邮件通过网线接收到自己的电脑上。

例 5.8 是 POP3 会话的一个例子。其中，C: 行由客户端发送，S: 行由邮件服务器端发送。可以看到它跟 SMTP 有很多相似之处。这个协议也是文本协议，也是面向行的，发送的有效消息部分由单点行加上行终止符结束，甚至

使用同一个退出命令——QUIT。如同 SMTP，每次客户端操作都经过回复行确认，回复行以状态码开头，其中包括了可供人眼识别的提示信息。

例 5.8 POP3 会话实例

```
C: <client connects to service port 110>
S: +OK POP3 server ready <1896.6971@mailgate.dobbs.org>
C: USER bob
S: +OK bob
C: PASS redqueen
S: +OK bob's maildrop has 2 messages (320 octets)
C: STAT
S: +OK 2 320
C: LIST
S: +OK 2 messages (320 octets)
S: 1 120
S: 2 200
S: .
C: RETR 1
S: +OK 120 octets
S: <the POP3 server sends the text of message 1>
S: .
C: DELE 1
S: +OK message 1 deleted
C: RETR 2
S: +OK 200 octets
S: <the POP3 server sends the text of message 2>
S: .
C: DELE 2
S: +OK message 2 deleted
C: QUIT
S: +OK dewey POP3 server signing off (maildrop empty)
```

```
C: <client hangs up>
```

与 SMTP 有一些不同之处，最明显的区别是 POP3 使用状态标记，而不是像 SMTP 那样使用 3 位数字的状态码。当然，请求的语义也不同。但是两者的族谱相似性（本章稍后讨论通用互联网元协议时会对此详细说明）很明显。

实例分析：IMAP，互联网消息访问协议

为了完整展示互联网应用协议的三剑客，我们最后再看看 IMAP——另一个设计风格略有不同的邮局协议。请看例 5.9：跟前面一样，C: 行由客户端发送，S: 行由邮件服务器发送。用斜体字强调的是注释，并非事务处理的实际部分。

例 5.9 IMAP 会话实例

```
C: <client connects to service port 143>
S: * OK example.com IMAP4rev1 v12.264 server ready
C: A0001 USER ”frobozz” ”xyzzy”
S: * OK User frobozz authenticated
C: A0002 SELECT INBOX
S: * 1 EXISTS
S: * 1 RECENT
S: * FLAGS (\Answered \Flagged \Deleted \Draft \Seen)
S: * OK [UNSEEN 1] first unseen message in /var/spool/mail/esr
S: A0002 OK [READ-WRITE] SELECT completed
C: A0003 FETCH 1 RFC822.SIZE                    \textit{Get message sizes}
S: * 1 FETCH (RFC822.SIZE 2545)
S: A0003 OK FETCH completed
C: A0004 FETCH 1 BODY[HEADER]                   \textit{Get first message header}
S: * 1 FETCH (RFC822.HEADER {1425}
```

```
<server sends 1425 octets of message payload>
S: )
S: A0004 OK FETCH completed
C: A0005 FETCH 1 BODY[TEXT]                    \textit{Get first message body}
S: * 1 FETCH (BODY[TEXT] {1120}
<server sends 1120 octets of message payload>
S: )
S: * 1 FETCH (FLAGS (\Recent \Seen))
S: A0005 OK FETCH completed
C: A0006 LOGOUT
S: * BYE example.com IMAP4rev1 server terminating connection
S: A0006 OK LOGOUT completed
C: <client hangs up>
```

IMAP 对有效载荷部分的分隔方法略有不同，它不是用一个点号来结束，而是将有效载荷的长度直接放在有效载荷之前发送。这稍稍增加了服务器的负担（消息必须提前完成组合，无法在初始化后流转），但使客户端工作更容易了——客户端可以提前知道需要分配多少存储空间作为整个处理消息的缓冲区。

同时，应注意，每个响应都标上了由请求提供的序列标签，本例中这个标签的形式为 A000n，但客户端也可以在这个位置上生成任何其它标记。这个特性使 IMAP 命令无需等待响应就可以流向服务器端；然后客户端的状态机就能够在数据回来时直接解析响应和有效数据载荷。这样可以减少等待时间。

IMAP（为取代 POP3 协议而设计）是一个成熟而强大的互联网应用协议的优秀设计典范，值得学习和效仿。

应用协议元格式

就像数据文件元格式是为了简化存储的序列化操作而发展出来一样，应用协议元格式是为了简化网络间事务处理的序列化操作而发展出来的。但在这种情况中采取的折衷略有不同：因为网络带宽要比存储昂贵得多，所以需更加重视事务处理的经济性。尽管如此，文本格式的透明性和互用性优势仍然十分显著，所以大多数设计者还是抵制住了牺牲可读性来优化性能的诱惑。

经典的互联网应用元协议

Marshall Rose 的 RFC 3117《论应用协议的设计》(On the Design of Application Protocols) [5]很好地概括了互联网应用协议设计中的种种问题。它明确了我们在分析 SMTP、POP 和 IMAP 时所注意到的一些经典互联网协议的描述手段 (trope)，并对其进行了具启发意义的分类。推荐大家一读。

经典的互联网元协议是文本格式，使用单行请求和响应，但有效数据载荷可以多行。

有效数据载荷要么是 8 位组数据作为前导，要么以“\r\n”行作为结束符。在后一种情况下，有效数据载荷在字节上已被补齐；所有以句点“.”开始的行前面需要另加一个句点，接收方既负责识别结束符又负责去除补齐字节。应答行由状态码和后接人可识别的消息构成。

这种经典风格的关键优势是可以随时扩展。解析器和状态机框架无需太多修改就能够适应新的请求，而且代码也容易编写，使其可以解析未知请求并返回错误信息或直接忽略这些未知请求。SMTP、POP3 和 IMAP 在使用过程中都经历了相当频繁的小扩展，互用性问题极少。相比之下，那些设计比较简单的二进制协议，却以不耐用而臭名昭著。

---

[5] 从 <ftp://ftp.rfc-editor.org/in-notes/rfc3117.txt> 处参阅 RFC 3117。

作为通用应用协议的 HTTP

自从万维网在 1993 年左右吸引到足量用户以来，应用协议的设计者已经越来越倾向于在 HTTP 上构建专用协议，并使用网页服务器作为通用服务平台。

这是一种可行的方案，因为在事务层上，HTTP 非常简单和通用。HTTP 请求采用类似 RFC-822/MIME 格式的消息：通常，消息头包含识别和认证信息，第一行是对通用资源指示符（URI）指定的某个资源的方法调用。最重要的方法是 GET（获取资源）、PUT（修改资源）和 POST（将数据发送给某个表单或后端进程）。URI 最重要的形式是 URL，即统一资源定位符（Uniform Resource Locator）；URL 通过服务类型、主机名称和在主机上的位置对资源进行识别。HTTP 响应只是一种 RFC-822/MIME 消息，可以包含由客户端解释的任意内容。

网页服务器处理 HTTP 的传输和请求多路复合层，同时也处理标准的服务类型，如 http 和 ftp。要编写可以处理自定义服务类型的网页服务器插件相对比较容易，也很容易分派 URI 格式的其它元素。

除了避免很多底层细节之外，这种方法也意味着应用协议可以通过标准的 HTTP 服务端口，不需要自身的 TCP/IP 服务端口。这成为一个非常显著的优势：大多数防火墙都开放 80 端口，而试图穿透其它端口则可能遇到技术上和政治上的困难。

风险也伴随这种优势而来：这意味着网页服务器和插件会越变越复杂，任何这些代码的破解都可能带来巨大的安全问题。要隔离并关闭出问题的服务也可能会更加困难。通常此时就要在安全和便利间做出折衷。

RFC 3205《论使用 HTTP 作为底层》（On the Use of HTTP As a Substrate）[6]向正在考虑将 HTTP 作为应用协议底层使用的人提出了很好的设计建议，文中还总结了各种权衡方案和所涉及的问题。

---

[6] 参见 RFC 3205<http://www.fags.org/rfcs/rfc3205.html>

实例分析：CDDB/freedb.org 数据库

音频 CD 由一序列称为 CDDA-WAV 的数字格式音轨组成。在一般计算机拥有足够速度和声音处理能力来实时解码之前几年，音频 CD 是为让简单消费型电子产品能够播放而设计的。正因为如此，它没有提供相应的格式来记录甚至非常简单的一些元信息，如唱片专辑和歌曲音轨标题等。但是现代计算机中的 CD 播放器需要这些信息，这样用户可以整理和编辑播放列表。

连上互联网，网上有（至少两个）资料库可提供根据 CD 上音轨长度表计算出的散列码与艺术家/专辑名/音轨名之间的对应记录。最初的一个资料库是 cddb.org，但另一个名为 freedb.org 的站点也许现在资料更完整，用的人也更多。随着新 CD 的不断推出，这两个站点都依靠用户来完成更新数据库的巨大任务。当 CDDB 决定将用户提供的所有信息都收为专有后，作为开发者的反抗，freedb.org 发展了起来。

对这些服务的查询本可以作为基于 TCP/IP 的自定义应用协议来实现，但是那样就要求采取以下步骤，如给它分配一个新的 TCP/IP 端口号，还要为它费力地从成千上万的防火墙上争取到通路。为了避免这些麻烦，该服务基于 HTTP 作为简单的 CGI 查询来实现（就好像 CD 的散列码是用户通过在网页上填表提供的）。

由于作了这种选择，所有现行各种编程语言的 HTTP 和 Web 访问程序库的基础代码都可以支持数据库的查询和更新。结果，在 CD 播放器中增加这样的支持功能几乎不费吹灰之力，而实际上现在每款 CD 播放软件都知道如何使用这些功能。

实例分析：互联网打印协议

互联网打印协议（IPP，Internet Printing Protocol）是一个非常成功、被广泛采用的网络访问打印机控制标准。指向 RFC 的链接、各种实现和大量的其它相关材料都可以在 IETF 的打印机工作组站点 <http://www.pwg.org/ipp/> 获得。

IPP 使用 HTTP1.1 作为传输层。所有的 IPP 请求都通过 HTTP POST 方法调用发送；响应是普通的 HTTP 响应。（《互联网打印协议模型和结构基本原理》(Rationale for the Structure of the Model and Protocol for the Internet Printing Protocot)，RFC 2568 的 4.2 节很好解释了做出这种选择的理由，值得正在考虑编写新应用协议的所有人员研究）。

在软件方面，HTTP1.1 得到了广泛应用。HTPP1.1 已经解决了许多传输层问题，不然，这些问题将使协议的设计者和实现者无法集中精力解决打印的域语义问题。HTTP1.1 能够很干净地进行扩展，因此 IPP 还有足够的发展空间。人们非常熟悉处理 POST 请求的 CGI 编程模型，开发工具也很丰富。

大多数具有网络能力的打印机已经内置了网页服务器，因为这是能人工远程查询打印机状态的一个自然的方法。这样，向打印机固件增加 IPP 服务的额外成本并不是很高。（这一点也适用于许许多多具有网络能力的硬件，包括自动售货机、自动咖啡机[7]和自动澡盆！）

基于 HTTP 的 IPP 协议的唯一严重缺点就是协议完全由客户端的请求来驱动。这样，模型中就不存在可供打印机向客户返回异步报警信息的余地（然而，聪明的客户可以运行一个很小的 HTTP 服务程序，来接收打印机发送的 HTTP 请求格式报警信息）。

BEEP：块可扩展交换协议

BEEP（原先叫 BXXP）是一种通用协议，对于通用底层这一角色来说和 HTTP 一样具有竞争力。之所以说竞争，是因为目前为止还没有一个制定较完善的元协议非常适合真正的对等网（P2P）应用，不像客户端 -服务器应用——HTTP 在这一领域游刃有余。访问 <http://www.beepcore.org/beepcore/docs/sl-beep.jsp> 项目网站可以找到有关标准以及数种语言的开源实现。

BEEP 的特点是既支持客户端/服务器模式，又支持对等网模式（peer-

[7] 参见 RFC 2324<http://www.ietf.org/rfc/rfc2324.txt> 和 RFC 2325<http://www.ietf.org/rfc/rfc2325.txt>

to-peer）。协议作者们对 BEEP 协议和支持库进行了良好设计。这样，只要选取正确组件就能省去很多杂务，如数据编码、流控、堵塞（congestion）处理、端对端加密支持和用多路传输组成长应答等。

从内部而言，BEEP 用户端（peer）之间相互交换自描述二进制包序列，后者与 PNG 中字节块类型非常相像。跟经典的互联网协议或 HTTP 相比，BEEP 设计对经济性的强调高于透明性，当数据量非常大时可能是更好的选择。BEEP 也避免了 HTTP 所有请求都必须由客户端发起的问题：在服务器端需要向客户端异步返回状态信息的情况下，BEEP 协议会更好。

到 2003 年年中，BEEP 仍然是一项新兴技术，只有几个演示项目。但是 BEEP 论文是非常不错的关于设计最佳协议的分析材料：即使 BEEP 本身无法得到广泛采用，这些论文仍然具有相当重要的指导价值。

XML-RPC，SOAP 和 Jabber

应用协议设计的一种发展趋势是在 MIME 中使用 XML 来架构请求和有效数据载荷。BEEP 用户端使用这种格式进行信道协商。有三个重要协议始终采用 XML 路线，包括 XML-RPC，用于远程过程调用的 SOAP（Simple Object Access Protocol，简单对象访问协议）和用于即时消息和在线状态报告（instant messaging and presence）的 Jabber。这三个协议都基于 XML 文档。

XML-RPC 非常具有 Unix 精神（其作者宣称在二十世纪七十年代通过阅读原始的 Unix 源码学会了编程）。它着意追求最简化，但是仍然非常强大，对于绝大多数专为传送布尔/整数/浮点/字符串标量数据类型的各种 RPC 应用提供一种方法，使它们能够以一种轻量、易于理解和监控的方式来完成任务。XML-RPC 类型实体比文本流丰富，但仍然简单而具有移植性，可被有效地用于检查接口复杂性。它有很多开源实现。XML-RPC 主页 <http://www/xmlrpc.com/> 做得不错，提供了规格书和多个开源实现。

SOAP 是一个较重量级的 RPC 协议，数据类型更为丰富，包括数组和类似 C 的结构。这个协议受到了 XML-RPC 的启发，但一直被批评为过分追求

“第二系统效应”设计的受害者，这么说不无道理。直到 2003 年中，SOAP 标准还在制定当中，但是在 Apache 上的试验版实现一直紧跟其设计草案。大家可以很容易地在网上搜索到 Perl、Python、Tcl 和 Java 中的开源客户端模块。W3C 规范的草案可从 <http://www.w3.org/TR/SOAP/> 获得。

作为远程过程调用方法的 XML-RPC 和 SOAP 都会带来某种风险，我们将在第 7 章结尾部分予以讨论。

Jabber 是一个为支持即时消息和在线状态报告而设计的对等协议。Jabber 作为应用协议的有趣之处在于：它支持 XML 表单和随时更新文档（live document）的输送。规格说明、文档和开源实现都可以通过 Jabber 款件基金会的站点 <http://www.jabber.org/about/overview.html> 获得。

透明性：来点儿光

> 美在计算科学中的地位，要比在其它任何技术中的地位都重要，因为软件是太复杂了。美是抵御复杂的终极武器。
>
> 《机器美学：优雅和技术本质》（1998 年）
>
> —David Gelernter

我们在前一章讨论了把数据格式和应用协议进行文本化的重要性，这种方式的表达容易让人分析和参与，使一些设计品质得以提升，虽然 Unix 传统非常重视这些品质，但很少（如果有的话）明确谈论过，那就是：透明性和可显性。

如果没有阴暗的角落和隐藏的深度，软件系统就是透明的。透明性是一种被动品质。如果实际上能预测到程序行为的全部或大部分情况，并能建立简单的心理模型，这个程序就是透明的，因为可以看透机器究竟在干什么。

如果软件系统所包含的功能是为了帮助人们对软件建立正确的“做什么、怎样做”的心理模型而设计，这个软件系统就是可显的。因此，举例来说，对用户而言，良好的文档有助于提高可显性；对程序员而言，良好的变量和函数名有助于提高可显性。可显性是一种主动品质。在软件中要达到这一点，仅仅做到不晦涩是不够的，还必须尽力做到有帮助[1]。

---

[1] 一位有经济头脑的朋友评论：“可显性降低进入门槛；透明性则减少代码中的存在成本。”

透明性和可显性对用户和软件开发人员都很重要。但是重要性体现在不同的方面。用户喜欢 UI 中的这些特性，是因为这意味着学习曲线比较平缓。当人们说 UI“直观”时，很大程度上是指 UI 的透明性和可显性；剩下一部分则来源于最小立异原则。我们将在第 11 章更深入分析这些使用户界面舒适而有效的特性。

软件开发者喜欢代码本身（用户不可见部分）的这些品质，因为他们经常需要对代码有很好理解后才能进行修改和调试。同时，如果程序的设计使内部数据流程非常容易理解，则这个程序更不可能因设计者没有注意到的不良交互而崩溃，更可能优雅地向前发展（包括适应新维护者接手的变化）。

透明性是本章引言 David Gelemter 所说的“美”的主要构成部分。Unix 程序员借鉴了数学家的说法，经常使用更明确的术语“优雅”来表达 Gelemter 所说的品质。优雅是力量与简洁的结合。优雅的代码事半功倍；优雅的代码不仅正确，而且显然正确；优雅的代码不仅将算法传达给计算机，同时也把见解和信心传递给阅读代码的人。通过追求代码的优雅，我们能够编写更好的代码。学习编写透明的代码是学习如何编写优雅代码的第一关，很难的一关——而关注代码的可显性则帮助我们学习如何编写透明的代码。优雅的代码既透明又可显。

通过两个极端例子来辨别透明性和可显性之间的差别可能会更容易些。Linux 的内核源码相当透明（相对其行为的内在复杂性而言），但根本不具备可显性——要获得必要的知识以便融入代码中并理解开发者的惯用语相当困难，而一旦做到，一切便豁然开朗[2]。另一方面，Emacs Lisp 库是可显的，但却不透明。要获得足够的知识来领会一件事很容易，但要理解整个系统却相当困难。

我们将在本章分析 Unix 设计的几个特性，这些特性不仅提高了用户界面的透明性和可显性，而且还提高了用户通常无法看见部分的透明性和可显性。

---

[2] Linux 内核在可显性上做了很多努力，包括 Linux 内核源码 tarball 中的 Documentation 子目录和相当多的教程网站和指导书籍。这些努力比起内核变化的速度来说远远不够，文档还将长期落后。

我们将逐步提出可以应用到编码和开发实践中的几个有用原则。以后，我们将在第 19 章看看好的发布工程实践（如编写一个内容恰当的 README 文档）如何能够让源码和设计一样具备可显性。

如果需要足够的理由，来回答这些品质为什么很重要，请记住：编写透明、可显的系统而节省的精力，将来完全可能就是自己的财富。

研究实例

本书的通常做法是将实例分析贯穿在基本原理中。但我们在本章首先将分析几个展示透明性和可显性的 Unix 设计，在介绍全部实例后才尝试从中总结经验。本章后半部分的每个分析点都要用到这些实例，这种编排方式会避免在论述中引用读者还没有见到的实例。

实例分析：audacity

首先，我们看一看 UI 设计中展示透明性的一个例子。这就是 audacity，运行在 Unix 系统、Mac OS 和 Windows 上开源的声音文件编辑器。源码、可下载的二进制文件、文档和屏幕截图可以在该项目的站点 <http://audacity.sourceforge.net/> 处获得。

这个程序支持对音频采样的剪切、粘贴和编辑操作，支持多声道编辑和混音。UI 部分相当简洁：audacity 窗口显示声音波形图，可以剪切和粘贴波形图。对波形图的操作在实施的同时直接反映在音频采样中。

多音轨编辑通过尽可能简单的方式支持：屏幕按空间关系显示每条音轨，既表达了各条音轨的同期性，又很容易检查匹配特性。可以用鼠标左右拖动音轨来改变各条音轨的相对时间。

这个 UI 有几个特性非常优秀并且值得效仿：以不同颜色区分了大而直观且可点击的操作按钮，提供了撤销命令，免除了许多尝试风险，音量控制滑杆使得柔度/响度在形状上一眼便可认出。

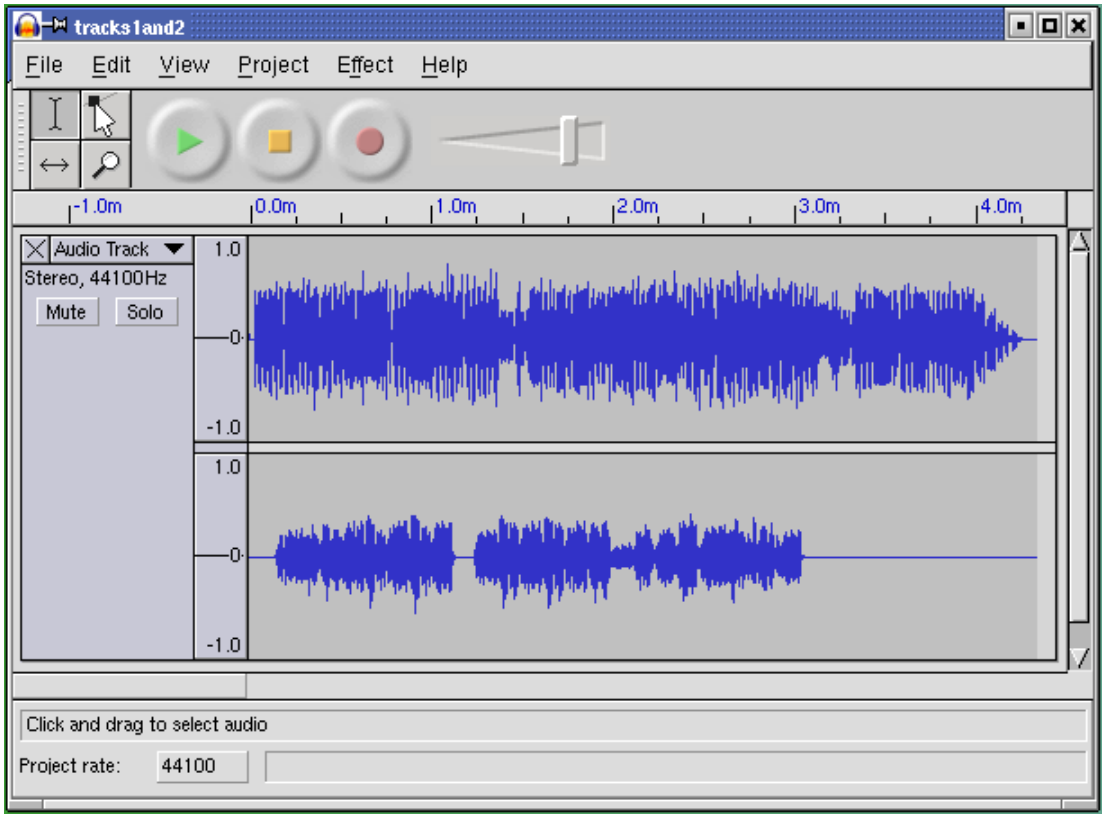


图 6-1：audacity 软件的屏幕截图

但这些都是细节。这个程序的最主要优点在于它具有非常透明、自然的用户界面，在用户和声音文件之间尽可能少地设置障碍。

实例分析：fetchmail 的 -v 选项

fetchmail 是一个网络网关程序，主要目的是在 POP3 或 IMAP 的远程邮件协议和互联网自带的 SMTP 协议之间进行转换，从而进行电子邮件交换。在采用 SLIP（串行线路接口协议）或 PPP 偶尔（端到端协议）连接到 ISP（互联网服务商）的 Unix 机器中，*fetchmail* 使用非常广泛，占据了互联网邮件业务量相当可观的一个部分。

fetchmail 的命令行选项不少于 60 个（正如本书后面所提出，这可能太多了），还有不是在命令行而是在运行控制文件中进行设置的其他选项。在所有这些选项中，最重要的一个选项——目前为止——是 -v，即详细

（verbose）选项。

当使用 -v 选项时，*fetchmail* 将发生的每一单 POP、IMAP 和 SMTP 处理都转储到标准输出设备中。开发者真正能够实时看到远程邮件服务器及邮件传输程序的协议处理代码。用户可以发送附有错误报告的会话记录。例 6.1 是一段典型的记录。

例 6.1 fetchmail 的 -v 记录实例

```
fetchmail: 6.1.0 querying hurkle.thyrsus.com (protocol IMAP)
              at Mon, 09 Dec 2002 08:41:37 -0500 (EST): poll started
fetchmail: running ssh %h /usr/sbin/imapd
              (host hurkle.thyrsus.com service imap)
fetchmail: IMAP< * PREAUTH [42.42.1.0] IMAP4rev1 v12.264 server ready
fetchmail: IMAP> A0001 CAPABILITY
fetchmail: IMAP< * CAPABILITY IMAP4 IMAP4REV1 NAMESPACE IDLE SCAN
              SORT MAILBOX-REFERRALS LOGIN-REFERRALS AUTH=LOGIN
              THREAD=ORDEREDSUBJECT
fetchmail: IMAP< A0001 OK CAPABILITY completed
fetchmail: IMAP> A0002 SELECT ”INBOX”
fetchmail: IMAP< * 2 EXISTS
fetchmail: IMAP< * 1 RECENT
fetchmail: IMAP< * OK [UIDVALIDITY 1039260713] UID validity status
fetchmail: IMAP< * OK [UIDNEXT 23982] Predicted next UID
fetchmail: IMAP< * FLAGS (\Answered \Flagged \Deleted \Draft \Seen)
fetchmail: IMAP< * OK [PERMANENTFLAGS
               (\* \Answered \Flagged \Deleted \Draft \Seen)]
               Permanent flags
fetchmail: IMAP< * OK [UNSEEN 2] first unseen in /var/spool/mail/esr
fetchmail: IMAP< A0002 OK [READ-WRITE] SELECT completed
fetchmail: IMAP> A0003 EXPUNGE
fetchmail: IMAP< A0003 OK Mailbox checkpointed, no messages expunged
```

```
fetchmail: IMAP> A0004 SEARCH UNSEEN
fetchmail: IMAP< * SEARCH 2
fetchmail: IMAP< A0004 OK SEARCH completed
2 messages (1 seen) for esr at hurkle.thyrsus.com.
fetchmail: IMAP> A0005 FETCH 1:2 RFC822.SIZE
fetchmail: IMAP< * 1 FETCH (RFC822.SIZE 2545)
fetchmail: IMAP< * 2 FETCH (RFC822.SIZE 8328)
fetchmail: IMAP< A0005 OK FETCH completed
skipping message esr@hurkle.thyrsus.com:1 (2545 octets) not flushed
fetchmail: IMAP> A0006 FETCH 2 RFC822.HEADER
fetchmail: IMAP< * 2 FETCH (RFC822.HEADER {1586}
reading message esr@hurkle.thyrsus.com:2 of 2 (1586 header octets)
fetchmail: SMTP< 220 snark.thyrsus.com ESMTP Sendmail 8.12.5/8.12.5;
               Mon, 9 Dec
2002 08:41:41 -0500

fetchmail: SMTP> EHLO localhost
fetchmail: SMTP< 250-snark.thyrsus.com
               Hello localhost [127.0.0.1], pleased to meet you
fetchmail: SMTP< 250-ENHANCEDSTATUSCODES
fetchmail: SMTP< 250-8BITMIME
fetchmail: SMTP< 250-SIZE
fetchmail: SMTP> MAIL FROM:<mutt-dev-owner@mutt.org> SIZE=8328
fetchmail: SMTP< 250 2.1.0 <mutt-dev-owner@mutt.org>... Sender ok
fetchmail: SMTP> RCPT TO:<esr@localhost>
fetchmail: SMTP< 250 2.1.5 <esr@localhost>... Recipient ok
fetchmail: SMTP> DATA
fetchmail: SMTP< 354 Enter mail, end with ”.” on a line by itself
#
fetchmail: IMAP< )
fetchmail: IMAP< A0006 OK FETCH completed
```

```
fetchmail: IMAP> A0007 FETCH 2 BODY.PEEK[TEXT]
fetchmail: IMAP< * 2 FETCH (BODY[TEXT] {6742}
 (6742 body octets) ********************.************************.
*******************************.***********************.***********
**********.**********************.***************
fetchmail: IMAP< )
fetchmail: IMAP< A0007 OK FETCH completed
fetchmail: SMTP>. (EOM)
fetchmail: SMTP< 250 2.0.0 gB9ffWo08245 Message accepted for delivery
 flushed
fetchmail: IMAP> A0008 STORE 2 +FLAGS (\Seen \Deleted)
fetchmail: IMAP< * 2 FETCH (FLAGS (\Recent \Seen \Deleted))
fetchmail: IMAP< A0008 OK STORE completed
fetchmail: IMAP> A0009 EXPUNGE
fetchmail: IMAP< * 2 EXPUNGE
fetchmail: IMAP< * 1 EXISTS
fetchmail: IMAP< * 0 RECENT
fetchmail: IMAP< A0009 OK Expunged 1 messages
fetchmail: IMAP> A0010 LOGOUT
fetchmail: IMAP< * BYE hurkle IMAP4rev1 server terminating connection
fetchmail: IMAP< A0010 OK LOGOUT completed
fetchmail: 6.1.0 querying hurkle.thyrsus.com (protocol IMAP)
               at Mon, 09 Dec 2002 08:41:42 -0500: poll completed
fetchmail: SMTP> QUIT
fetchmail: SMTP< 221 2.0.0 snark.thyrsus.com closing connection
fetchmail: normal termination, status 0
```

-v 选项使得 *fethcmail* 的行为具有可显性（可以让人看见协议交换过程），这非常有用。我认为这一点相当重要，所以编写了专门代码，在 -v 处理转储中遮住用户口令，这样，人们无需记住编辑其中的敏感信息就可以传阅

或寄出。

这被证明是一个不错的决定。在八成以上的错误报告中，一个懂行的人只要看一下对话记录，几秒内就可以进行诊断。在 fetchmail 邮件列表中有不少懂行的人——事实上，由于大多数 bug 都非常容易诊断，很少需要我亲自处理。

多年来，fetchmail 已经赢得了“防弹程序”的美名。fetchmail 也可能发生配置错误，但很少彻底无法工作。如果怀疑这归功于十之八九的 bug 都能迅速被发现这一事实，那我打赌你错了。

我们可以从这个例子中学到些东西。教训是：不要让调试工具仅仅成为一种事后追加或者用过就束之高阁的东西。它们是通往代码的窗口：不要只在墙上凿出粗糙的洞，要修整这些洞并装上窗。如果打算让代码一直可被维护，就始终必须让光照进去。

实例分析：GCC

GCC，即现代大多数 Unix 都使用的 GNU C 编译器，也许是关于透明性更好的工程实例。GCC 由一系列处理阶段组成，并由一个驱动程序将其紧密结合在一起。它们是：预处理器、解析器、代码生成器、汇编器和链接器。

前三个阶段都接受可读文本格式的输入，然后输出可读的文本格式（汇编器必须输出而链接器必须输入二进制格式，这一点从定义便可理解）。运用 *gcc*(1) 驱动程序的不同命令行选项不仅可以看到 C 预处理之后、汇编生成之后和目标码生成之后的各个结果，而且可以监控解析进程和代码生成进程中的许多中间步骤的结果。

> 这完全就是第一款 (PDP-11)C 编译器 cc 的结构。
>
> —Ken Thompson

这种组织有很多好处，其中对 GCC 特别重要的一个好处体现在回归测试

中[3]。因为各个中间格式的大部分都是文本格式，在回归测试中，对中间结果使用简单的文本 diff 操作就可以发现并分析结果是否偏离预期结果；没必要使用专门的转储 -分析工具，这些工具完全可能隐藏自身的 bug，而且无论怎样都意味着额外增加维护负担。

这个例子揭示的设计模式是驱动程序应该具备监控开关，这些监控开关仅仅（但足够了）揭示组件间的文本数据流。如同 *fetchmail* 的 -v 选项一样，这些选项也不是事后追加，而是为了可显性才设计进来的。

实例分析：kmail

kmail 是随 KDE 环境一起发布的 GUI 邮件阅读程序。*kmail* 的用户界面非常优雅，设计得很好，包括很多优良特性，如自动显示 MIME multipart 内嵌图像和支持 PGP 密钥加密/解密等。对最终用户相当友好——我心爱但不懂技术的妻子就喜欢使用它。

许多邮件用户代理只是向可显性做了个姿态，用一个命令来切换显示邮件头的所有信息，而不是显示 From（发件人）和 Subject（主题）等选择性信息。但 *kmail* 的用户界面在这个方向上走得更远。

kmail 运行时在窗口底部的单行子窗口显示状态通知，青灰色的背景上显示小字体文字，很阴显是模仿了 Netscape/Mozilla 的状态栏。例如，打开一个邮箱，状态栏显示邮件总数和未读邮件数。这个视觉显示并不突兀：可以不理会这些通知信息，但如果希望的话又很容易注意到它们。

kmail 的 GUI 是很好的用户界面设计。这个 GUI 内容翔实但并不令人分心；它宣扬了我们在第 11 章引证的理由，即让 Unix 工具正常运行的最好策略是在大部分时间里沉默。*kmail* 设计者在借鉴浏览器状态栏的外观和感觉方面显示了良好的品味。

---

[3] 回归测试是一种检测软件修改后是否引入 bug 的方法。方法是对于处在修改变动期的软件，根据某些固定的测试输入，定期检查软件的输出是不是跟先前获得并已知（或假定）正确的输出快照一致。

但是，在你检查为什么信件无法发出时才会领会 *kmail* 开发者的品味。如果在发送过程中仔细观察，可以发现 SMTP 和远端邮件传输处理每一行都在进行的同时，也回显在 *kmail* 状态栏上。

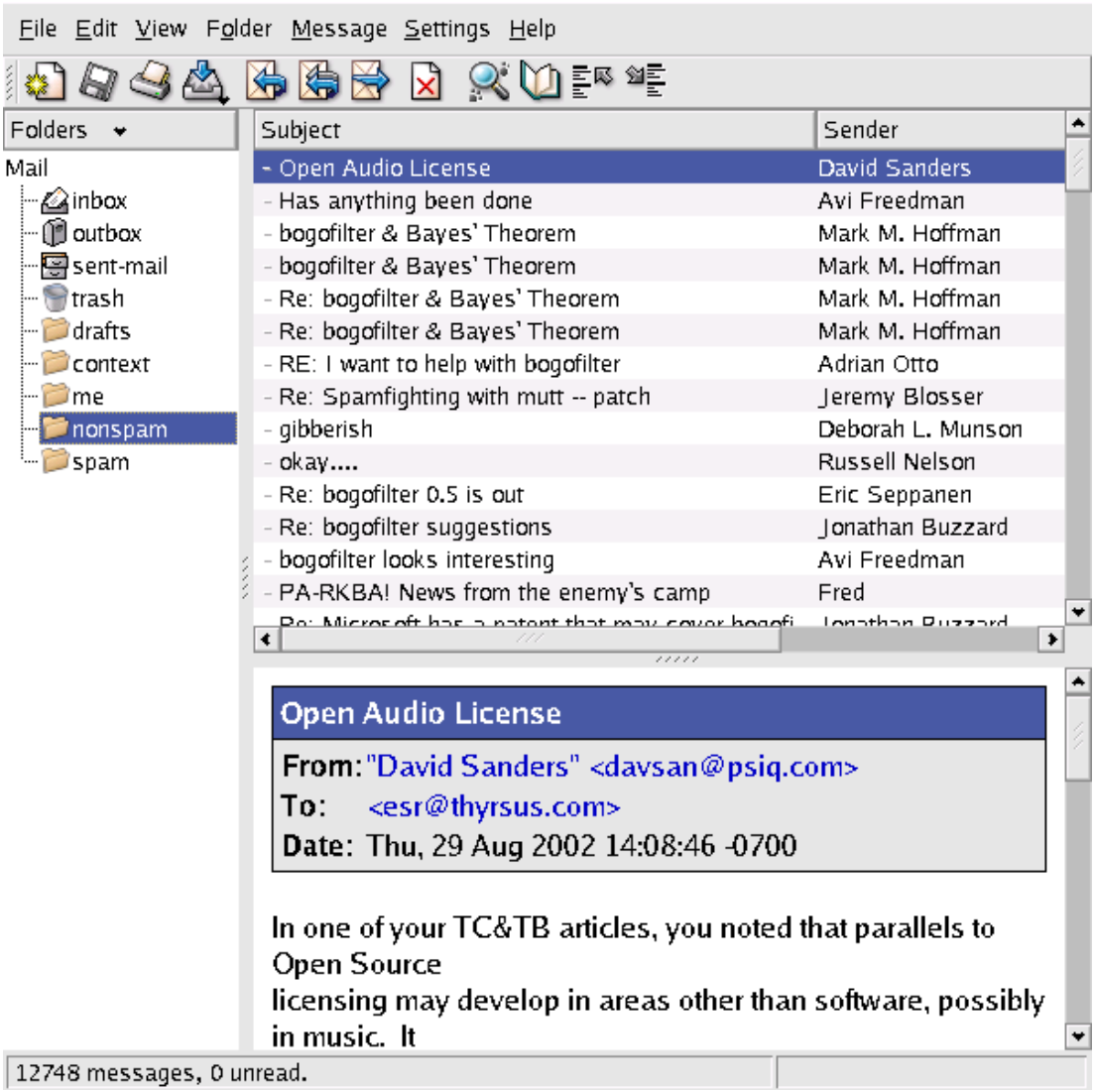


图 6-2: kmail 屏幕截图

kmail 开发者巧妙地避开了一个陷阱，这个陷阱经常让类似 *kmail* 的 GUI 程序故障检测员极端痛苦。和 *kmail* 目标相同的大多数设计者常常完全压制这些信息，生怕这些信息会吓着假想中的邻居阿姨 Tillie，逼她回头去用华丽庸俗又伪装简洁的 windows 机器。

相反，他们为透明性而设计——既显示了事务处理信息，又很容易让人视

而不见。通过恰当的表现形式，他们成功满足了 Tillie 阿姨和解决她计算机问题的“极客”(geeky) 侄子 Melvin。这招很聪明，其它 GUI 界面能够也应该效仿之。

当然，最后，这些可视化信息对 Tillie 阿姨也非常有用，因为这意味着 Melvin 在解决她的电子邮件问题时，不大可能在挫折面前举手服输。

这个例子的教训很明显：让 UI 沉默只做对了一半。真正的聪明是找到一个方法，可以访问具体细节，但又不让它们太显眼。

实例分析：SNG

sng 程序在 PNG 图形格式及其纯文本形式（SNG 即 Scriptable Network Graphics，可脚本化网络图形）之间进行转换，这个文本表达可用普通的文本编辑器检查和修改。对 PNG 文件运行程序，可产生 SNG 文件：对 SNG 文件运行程序，又能恢复等价的 PNG 文件。两个方向都是 l00% 可靠的无损转换。

在语法风格上，SNG 很像 CSS（Cascading Style Sheets，层叠样式表），另一种控制图形表达的语言，这至少朝最小立异原则的方向摆了一个姿态。以下是一个测试实例：

例 6.2 SNG 实例

```
#SNG: This is a synthetic SNG test file

# Our first test is a paletted (type 3) image.
IHDR: {
        width: 16;
        height: 19;
        bitdepth: 8;
        using color: palette;
        with interlace;
```

```
}

# Sample bit depth chunk
sBIT: {
  red: 8;
  green: 8;
  blue: 8;
}

# An example palette: three colors, one of which
# we will render transparent
PLTE: {
   (0,     0, 255)
   (255,   0,   0)
   ”dark slate gray”,
}

# Suggested palette
sPLT {
   name: ”A random suggested palette”;
   depth: 8;
   (0,     0, 255), 255, 7;
   (255,   0,   0), 255, 5;
   ( 70,  70,  70), 255, 3;
}

# The viewer will actually use this...
IMAGE: {
   pixels base64
2222222222222222
2222222222222222
```

```
0000001111100000
0000011111110000
0000111001111000
0001110000111100
0001110000111100
0000110001111000
0000000011110000
0000000111100000
0000001111000000
0000001111000000
0000000000000000
0000000110000000
0000001111000000
0000001111000000
0000000110000000
2222222222222222
2222222222222222
}

tEXt: {                                 # Ordinary text chunk
  keyword: ”Title”;
  text: ”Sample SNG script”;
}

# Test file ends here
```

关键在于，通用图形编辑程序不一定支持各种难懂的 PNG 块类型，这种手段使用户可以对其编辑。无需编写专用代码来迎合 PNG 二进制格式，用户只要将图像转换成纯文本表达，对之进行编辑，最后再转换回图像。另一个可

能的应用是使图像可以被版本控制：在大多数版本控制系统中，文本文件要比二进制文件容易处理得多；对 SNG 表现形式进行 diff 操作，产生的信息也将是有价值的。

然而，此处的收获不仅仅是节省了编写专用代码来处理二进制 PNG 文件的时间。*sng* 程序的代码本身并不特别透明，但是通过让 PNG 的全部内容可显，提高了程序中较大系统的透明性。

实例分析：Terminfo 数据库

Terminfo 数据库是视频显示终端的描述集。每一条记录描述了在终端屏幕上执行的不同操作的转义序列，例如插入行或删除行、删除光标位置到行尾或屏幕末端的全部内容、开始或结束反相、下划线和闪烁等屏幕高亮显示。

terminfo 数据库主要由 *curses*(3) 库使用。这些构成了我们将在第 11 章讨论的“roguelike”式接口风格的基础，也构成了 *mutt*(1)、*lynx*(1) 和 *slrn*(1) 等广泛使用程序的基础。尽管在当今位图显示器上运行的 *xterm*(1) 之类的终端仿真器，所具备的能力都是以 ANSI X3.64 标准终端以及由来已久的 VT100 终端能力为基础，而作了一些小变化，但其中一些变化足以使将 ANSI 能力硬塞进应用程序成为一个糟糕的想法。terminfo 也值得研究，因为在管理其它没有标准方式报告自己能力的外设硬件对经常产生问题，这些问题和 terminfo 所解决的问题在逻辑上非常相似。

terminfo 的设计得益于使用更早期名为 termcap 的能力格式的经验。termcap 描述的数据库以文本格式存放在`/etc/termcap` 大文件中；尽管这个格式已经过时，但是每个 Unix 系统几乎肯定包含了一份拷贝。

通常，用来查找终端类型记录的关键字是环境变量 TERM，就本实例分析而言，这个环境变量似乎是魔术般设置好的[4]。使用 terminfo（或

[4] 事实上，TERM 是系统在登录时设置的。对串行线上的实际终端而言，从 tty（电传打字机终端）行到 TERM 值的映射在启动时由系统配置文件进行设置。视 Unix 版本不同，具体情况也不同。xterm(1) 之类的终端仿真器自己设置这个变量。

termcap）的应用程序在启动时会有小小的延时：当 *curses*(3) 库自身初始化时，它必须查询 TERM 的对应记录并将其载入内存。

使用 termcap 的经验表明，启动延迟主要由解析终端能力的表达文本所要求的时间决定。因此，terminfo 的记录是二进制结构转储，列集和散集的速度能够更快一些。整个数据库有一个主文本格式，即 terminfo 能力文件。该文件（或单个记录）可以使用 terminfo 编译器 *tic*(1) 编译成二进制形式；二进制记录可以由 *infocmp*(1) 反编译成可编辑的文本格式。

从表面上看，这个设计与我们在第 5 章给出的反对二进制缓存的建议矛盾，但实际上它是个极端例子，这里，采用二进制是个好策略。对主文本的编辑很少——实际上，Unix 通常带着预编译过的 terminfo 数据库一起发布，主文本主要作为文档使用。这样，通常可能影响这种方法的同步性和不一致性问题几乎从未发生过。

terminfo 的设计者本可以用另一种方法来优化速度，包含二进制条目的整个数据库本可以放入某种大而晦涩的数据库文件中。事实上，他们实际采取的方法更聪明而且更符合 Unix 精神。Terminfo 记录存放在一个目录层次结构中，在现代 Unix 上通常放在`/ur/share/terminfo` 下。请查阅一下 *terminfo*(5) 的手册页来找到在你系统中的位置。

如果访问 terminfo 目录，可以看到由单个可打印字符命名的子目录。每个子目录下存放以该字符开始的终端类型的记录。这种结构旨在避免在很大的目录中进行线性查找。在更现代的 Unix 文件系统中，为了优化快速查找而以 B-tree 或其它结构来表达目录，子目录就不需要了。

> 我发现，即使在相当现代的 Unix 上，将一个大目录分拆成子目录也能显著提高性能。在一台运行 DEC Unix 的新型 DEC Alpha 机器上，为一所大型教育机构制作的授权用户数据库有好几万个文件（子目录名由名字的首字母和尾字母构成——比如，“Johnson”可能在目录“j_n”中——在我们的测试中最管用。使用头两个字母就没这么好，因为许多系统生成的名字要到结尾才不同）。这也许就是说复杂的目录索引仍然没有它应该表现的那样达标……但即使如此，也使得无需它就能运行良好的结构要比需要它才能运行的结构更具备可移植性。
>
> —Herry Spencer

这样，打开一个 terminfo 记录的开销就是两个文件系统查找操作和一个文件打开操作。但既然从大数据库找到同一个记录本来就需要对数据库进行一个查找操作和一个打开操作，terminfo 结构的增量成本最多也就是一个文件系统的查找操作。事实上，比这个还要低——只是一个文件系统查找操作和大数据库所使用的任何检索方法之间的成本差。这可能微不足道，并且在每个应用程序启动时只进行一次也可以容忍。

terminfo 本身使用文件系统作为一个简单的层级数据库。这种偷懒相当具有建设性，符合经济性原则和透明性原则。这意味着对文件系统进行浏览、检查和修改的所有普通工具都可以用于对 terminfo 数据库进行浏览、检查和修改；无需编写和调试专用工具（用于打包和解包单个记录的 *tic*(1) 和 *infocmp*(1) 工具除外）。这也意味着要加速数据库的访问就得要加速文件系统本身，知道这一点可以使更多应用程序受益，而不仅仅是 *curses*(3) 的用户。

这种结构还有另外一种优点，但在 terminfo 例子中没有展示出来：你开始使用 Unix 的授权机制而不用自己编写带来额外 bug 的访问控制层。这也是采纳而不是对抗 Unix“一切皆文件”基本原则的结果。

terminfo 目录的布局在大多数 Unix 文件系统上都很浪费空间。每条目长度通常在 400 ～ 1400 字节之间，但是文件系统通常为每一个非空磁盘文件至少分配 4k 的空间。出于选择压缩二进制格式的同一个原因，即为了把 terminfo 使用的程序的启动延时降到最小，设计者接受了这个代价。同一价格所能买到的磁盘容量已经猛增了一千倍，更能证明这个决定的正确。

比较这种格式和 Microcsoft Windows 的注册表文件所用的格式很有启发意义。注册表是 Windows 本身及应用程序都使用的属性数据库。所有注册记录都存放在一个大文件中。注册记录既包含文本也包含二进制数据，需要专用的编辑工具。别的不说，这种“一个大文件”的方法还导致了臭名昭著的“注册表蠕变”现象：平均访问时间随着新记录的加入而无限上升。因为系统没有提供标准 API 来编辑注册表，应用程序本身使用专用代码编辑注册表，使得注册表极易受损，甚至能够锁定整个系统。

使用 Unix 文件系统作为数据库是一种策略，对数据库要求简单的其它应用程序可以效仿并从中受益。不这样做的充分理由通常与性能问题无关，更可能的情形是数据库关键字不太适合做文件名。无论如何，这是在原型设计时非常有用的一种很好的快速编程方法。

实例分析：Freeciv 数据文件

Freeciv 是一款受到 Sid Meier 经典的 *Civilization Ⅱ* 启发而制作的开源策略游戏。在该游戏中，每个玩家从一群到处流浪的新石器游牧民开始缔造一个文明。玩家的文明可以探索并拓殖世界，参与战争，从事贸易和研究先进技术。有些玩家实际上可能是人工智能；和这些电脑玩家玩单机游戏很有挑战性。如果谁统治了整个世界，或者第一个研制出先进技术从而获得宇宙飞船飞往半人马座阿尔法星 (Alpha Centauri)，谁就是游戏的胜利者。源码和文档可以在 <http://www.freeciv.org/> 处获得。

我们还会在第 7 章把 Freeciv 这个策略游戏作为客户端/服务器划分的一个例子提出，在这个游戏里，服务器维持共享状态而客户端专注 GUI 表现。但是这个游戏还有另外一个值得注意的体系特征：游戏的大部分固定数据，没有编入服务器的代码中，而是在属性登记表中说明，由游戏服务器在启动时读取。

这个游戏的登记表文件以文本数据文件的格式编写，把文本串（相关的文字和数字属性）放到游戏服务器各个重要数据（比如民族和装备类型）的内部列表中。微语言（minilanguage）具有包含（include）指令，所以游戏数据可以划分为许多语意单元（不同的文件），每一个都可以独立编辑。这个设计方案得到了充分贯彻，根本不需要改动服务器代码，只要简单地在数据文件中进行声明就可以定义新的民族和装备。

Freeciv 服务器的启动解析过程有一个非常有趣的特性，它与 Unix 的两个设计原则有些冲突，因此值得进一步关注。服务器会忽略它不知道如何使用的属性名。这使得可能在不中断启动解析的前提下对服务器还没有使用的属性进行声明。这意味着游戏数据（策略）的开发和服务器引擎（机制）的开发可

以干干净净地分离出来。另一方面，这也意味着启动解析不会理会属性名简单的拼写错误。这种不声不响的“不作为”似乎违反了补救原则。

要解决这个冲突，应注意：使用登记表数据是服务器的工作，但是仔细检查数据的任务可以移交给另一个程序，每次修改登记表时，由编辑人员运行这个程序。Unix 的解决方案可以是一个独立的审核程序，后者分析可以机读的规则集格式规范说明，或服务器源码来确定程序使用的属性集，并解析 Freeciv 的登记表来确定程序提供的属性集，然后准备一份差异报告[5]。

所有 Freeciv 数据文件的聚合在功能上类似于 Windows 注册表，甚至使用和注册表文本部分类似的语法。但是，我们所注意到的 Windows 注册表蠕变和损坏问题在这里并没有发生，原因是没有程序（在 Freeciv 套件内部和外部）写入这些文件。这是一个只能由游戏维护者编辑的只读登记表。

---

[5] 在 Unix 下，此类验证程序的老祖宗是 *lint* ——一个从 c 编译器独立出来的 c 代码校验器。尽管 GCC 已经吸收了其功能，但是老一辈的 Unix 人仍然倾向于把运行验证器的进程称之为“linting”，这个名字也在一些公用程序中保留了下来，如 *xmllint*。

图 6-3: freeciv 游戏的主窗口

对数据文件解析的性能影响已经降到了最低，因为对每个文件而言，解析操作只有在客户端或服务器启动时执行一次。

为透明性和可显性而设计

要为透明性和可显性而设计，就必须运用各种计策来保持代码的简洁，也必须专注代码同其他人交流的方式。在“这个设计能行吗？”之后要提出的头几个问题就是“别人能读懂这个设计吗？这个设计优雅吗？”我们希望，此时大家已经很清楚，这些问题不是废话，优雅不是一种奢侈。在人类对软件的反映中，这些品质对于减少软件 bug 和提高软件长期维护性是最基本的。

透明性之禅

从本章目前为止，已经研究过的例子展现出来的一种模式是：要追求代码的透明，最有效的方法很简单，就是不要在具体操作的代码上叠放太多的抽象层。

在第 4 章讲述分离价值的部分，我们建议对引起设计问题的特殊、意外的情况进行抽象、简化和概括，并尽量从中分离出来。如何抽象的建议同我们在这里提出的不要过度抽象的建议实际上并不矛盾，因为摆脱假设和忘记正在解决的问题并不一样。这也是我们提出要保持薄胶合层建议时的部分用意所在。

禅的一个主要教导是，通常我们都透过源于欲望的偏见和成见的迷雾观察世界。要开悟，我们必须遵循禅的教导，不仅要“去欲望，少依恋”，还要“如实见”——不要让偏见和成见蒙住了眼。

这是给软件设计者的一个非常好的实用建议，也是 Unix 做极简主义者的经典建议的部分隐含意义。软件设计者都是聪明人，可对其处理的应用域形成概念（抽象）。他们把围绕这些概念编写的软件组织起来。然后，调试时，他们经常发现很难透过这些概念弄明白到底发生了什么。

任何禅宗大师都能马上认出这个问题，笑曰“麻三斤”，或许还会好好地敲一下学生[6]。有意识地为透明性而设计，即是在暗处解决透明性。

我们在第 4 章批评了面向对象的编程，对于 1990 年代在 OO 福音下成长起来的编程者，可能是危言耸听。面向对象的设计并非必然是过度复杂的设计，但我们却发现事实往往如此。太多的 OO 设计就像是意大利空心粉一样，把“is-a”和“have-a”的关系弄得一团糟，或者以厚胶合层为特征，在这个胶合层中，许多对象的存在似乎只不过是为在陡峭的抽象金字塔上占个位置罢了。这些设计都不透明，它们（臭名昭著地）晦涩难懂并且难以调试。

正如我们在前面所注意到的那样，Unix 程序员是最早提倡模块化的狂热分子，但又往往以比较平和的方式处理这个问题。保持薄胶合层也是其中的一

6　参考“无门关洞山三斤”的公案 [Mumon]。

个部分：更普遍的是，Unix 传统教导我们不要垒高台，要用设计简单而透明的算法和数据结构紧贴基面。

和禅宗一样，优秀 Unix 代码的简洁依赖于严格自律和高水平技艺，这两者乍看未必会看得出来。透明性是项辛苦的工作，但值得我们努力追求，而且并不为附庸风雅。和禅宗不一样的是，软件需要调试——而且通常在整个使用期都需要不断的维护、向前移植和改写。因此，透明性不仅是一种美学意义上的成功；更是一种胜利，反映在软件整个生命周期上，意味着更低的成本。

为透明性和可显性而编码

透明性和可显性同模块性一样，主要是设计的特性而不是代码的特性。仅仅做对一些底层风格要素，如清晰且统一的代码缩进，或具有良好的变量命名约定，是不够的。这些特性更多与代码中不易硬性规定的特性有关。以下这些问题需要好好思考：

- 程序调用层次中最大的静态深度是多少？也就是说，不考虑递归，为了建立心理模型来理解代码的操作，人们将要调用多少层？提示：如果大于四，就要当心。
- 代码是否具有强大、明显的不变性质[7]？不变性质帮助人们推演代码和发现有问题的情况。
- 每个 API 中的各个函数调用是否正交？或者是否存在太多的特征标志（magic flags）和模式位，使得一个调用要完成多个任务？完全避免模式标志会导致混乱的 API，里面包括太多几乎一模一样的函数，但是频繁使用模式标志更容易产生错误（很多易忘并且易混的模式标记）。

[7] 不变性质是指一个软件设计中各个操作都保持不变的特性。例如，在大多数数据库中，两个记录的关键字不能相同，这就是不变性。在正确处理字符串的 C 程序中，从各个字符串函数退出时，每一个串缓冲区都必须包含终止符 NUL 字节。在一个库存系统中，没有哪部分的总数量可以小于零。

- 是否存在一些顺手可用的关键数据结构或全局唯一的记录器（scoreboard），捕获了系统的高层级状态？这个状态是否容易被形象化和检验，还是分布在数目众多的各个全局变量或对象中，而难以找到？
- 程序的数据结构或分类和它们所代表的外部实体之间，是否存在清晰的一对一映射？
- 是否容易找到给定函数的代码部分？不仅单个函数、模块，还有整个代码，需要花多少精力才能读懂？
- 代码增加了特殊情况还是避免了特殊情况？每一个特殊情况可能对任何其它特殊情况产生影响：所有隐含的冲突都是 bug 滋生的温床。然而更重要的是，特殊情况使得代码更难理解。
- 代码中有多少个 magic number（意义含糊的常量）？通过审查是否很容易查出实现代码中的限制（比如关键缓冲区的大小）？

代码能简单最好。但是如果代码很好地解决了上述问题，则代码也可以复杂，而且不会对维护人员造成认知负担。

读者会发现，把以上这些问题和第 4 章模块性列出的问题作比较将很有启发性。

透明性和避免过度保护

程序员经常建造过分精细的抽象城堡，这一倾向的近亲是过度保护底层细节。尽管在程序的正常操作模式中隐藏这些细节并不是不良作法（fetchmail 的 -v 选项的缺省被关闭），但这些细节必须能够找到。隐藏细节和无法访问细节有着重要区别。

不能展示其行为的程序使故障检测困难得多。所以，经验丰富的 Unix 用户实际上把调试和探测开关的存在视为良好程序的标志，不存在则认为程序可能有问题。不存在表明开发者不是经验不足就是粗心大意：存在则表明开发者很聪明，遵循了透明性原则。

在针对最终用户编写的 GUJ 应用中，如邮件阅读器，过度保护的诱惑特别强烈。Unix 开发者对 GUI 界面比较冷淡的原因之一是，在设计者仓促完成的每个看起来“用户友好”的 GUI 界面中，都令必须解决用户问题的人感到无从下手而非常沮丧——或者，确切地说，在设计者预想的狭窄范围以外，与界面的交互令人感到相当不透明。

更糟的是，对自身在做什么的不透明程序常常夹有很多假定，在设计者没有预计到的使用场合，程序不是无法工作，就是非常不稳定，或者两者皆有。表面光鲜但一压即垮的工具是没有长远价值的。

Unix 传统大力倡导具有灵活性的程序，以适应更广的使用范围和排错情形，包括当用户表明愿意处理时，有能力给用户提供尽可能多的状态和活动信息。这对排错非常有用；对于培养更聪明、更独立自主的用户也很有用。

透明性和可编辑的表现形式

以上这些例子所展现的另一个主题是一些程序的价值，这些程序可以把不易实现透明性的定义域问题转换为容易实现透明性的定义域问题。Audacity、*sng*(1) 和 *tic*(1)/infocmp(1) 程序对都有这种特性。它们所处理的对象无法手工处理或容易阅读：音频文件不是可视的对象，尽管 PNG 格式表达的图像可视，复杂的 PNG 标注块并不可视。以上这三个程序都将二进制文件格式的处理转换成人们可以更容易运用日常生活中获得的直觉和能力来解决的问题。

所有这些例子都遵循的一个原则，即把表现形式的损失降到最小——实际上，这些转换都是可逆、无损的。这个特性非常重要，即使没有明确提出 100% 忠实的应用需求，这个特性也值得实现。这给潜在用户以不断尝试而不会损坏数据的信心。

第 5 章讨论的数据文件格式文本化的所有优点同样适用于 *sng*(1)、*infocmp*(1) 和其同类程序所生成的文本格式。*sng*(1) 的一个重要应用是通过脚本自动生成 PNG 标注——因为 *sng*(1) 的存在，这类脚本更容易编写。

无论何时碰到涉及编辑某类复杂二进制对象的设计问题，Unix 传统都提倡首先考虑，是否能够编写一个类似于 *sng*(1) 或 *tic*(1)/*infocmp*(1) 组的工具，以便能够在可编辑的文本格式和二进制格式之间来回进行无损转换。对于这类程序还没有确定的术语，但我们姑且称之为**文本化器**（*textualizer*）。

如果二进制对象是动态生成的，或者非常大，那么用文本化器转化所有状态可能不实际或根本不可能。在这种情况下，对应的任务是编写一个浏览器。这方面的范例是向 *fsdb*(1)，即不同版本的 Unix 都支持的文件系统调试器；Linux 上的对应程序是 *debugfs*(1)。另外两个例子是浏览 PostgreSQL 数据库的 *psql*(1)，用于在配置 SAMBA 的 Linux 机器上查询 Windows 文件共享的 *smbclient*(1)。这五个程序都是简单的 CLI 程序，可由脚本和测试工具来驱动。

至少以下四个理由表明编写文本化器或浏览器非常值得一试：

- 可以获得良好的学习经验。也许还有其它好方法来认识所处理对象的结构，但是没有哪一个方法明显比这个好。
- 有能力将结构内容转储，以供审查和调试。因为这样的工具方便了转储操作，所以可以经常进行。得到的信息越多，就可能获得更多认识。
- 有能力轻松生成测试负载和特例。这意味着更有可能探测到对象状态空间的死角——并且更容易把相关软件用垮，所以可在用户用垮它之前将其修补好。
- 可以获得可复用的代码。如果编写浏览器/文本化器时很小心，并且保持 CLI 解释器和列集/散列库的分离，就会发现代码可以在实际的应用程序中得到复用。

完成这些后，很可能发现，把文本化器/浏览器作为引擎使用就可以应用“接口和引擎分离”模式（参考第 11 章）。可以受益于这种模式的各种好处。

> 让文本化器甚至能够对损坏的二进制对象进行读写操作，尽管很难，但值得去做。第一，可以对压力测试软件生成受损测试用例；第二，可以让紧急修复

> 更容易。也许很难处理对象结构已经混乱的情况，但至少应该处理结构内容无意义的情况，例如，可以用十六进制表达无意义的值，然后把十六进制转换回无意义的值。
>
> —Henry Spencer

透明性、故障诊断和故障恢复

透明性还有一个与简化调试有关的好处，就是 bug 发作时，透明的系统更容易实施恢复措施——而且，经常是，首先更能抵抗 bug 的破坏。

对比 terminfo 数据库和 Windows 注册表，我们发现注册表出名地容易受到错误代码的破坏。这可能会使整个系统都无法使用。即使系统没有瘫痪，但如果破坏本身干扰了专用的注册表编辑工具，恢复工作就会很困难。

我们的 Unix 实例表明，为透明性设计可以防止这类问题的发生。因为 terminfo 数据库不是单一的大文件，修补一条 terminfo 记录不会使整个 terminfo 数据集都不能用。像 termcap 这样纯文本化的单一大文件格式的解析通常都使用（与二进制结构转储的块读取不同）可以从单点错误恢复的方法。SNG 文件中的语法错误人工就可以解决，不需要专用编辑器，那些专用编辑器可能拒绝载入受损的 PNG 图像。

回到 *kmail* 案例，这个程序使得故障诊断更加容易是因为遵循了补救原则：SMTP 错误信息太扰人，这就非常有用。根本不必解析 kmail 自身产生的混乱信息层来了解同 SMTP 服务器的交互情况。所需做的一切就是到该去的地方查看，因为 kmail 就是透明的，而且不抛弃错误状态的信息（SMTP 本身也是文本化的，并且在其事务处理中包括人可读的状态信息，这也很有用）。

像文本化器和浏览器这样的可显性工具使得故障诊断更加容易。我们已经接触过其中一个原因：它们简化了系统状态的检查。但还有另外一个同样起作用的效应；数据的文本化版本往往具备有用的冗余（比如使用空白进行可视化的分隔或成为清楚的解析分隔符）。这些的出现不仅便于人类阅读，而且还让程序更能抵御微小失误造成的不可挽回的破坏作用。PNG 文件的受损块很

少能够恢复，但是人有识别和从上下文推导的能力，能够修复等价的 SNG 格式。

再一次，健壮原则就非常清楚了。简洁加上透明，降低了费用，缓解了每个人的压力，让人们从中解放出来，更多地投入到新问题中，而不是总为旧错误擦屁股。

为可维护性而设计

如果作者以外的其他人能够顺利地理解和修改软件，则这个软件就是可维护的。可维护性不仅要求代码能够运行；还要求代码能够遵循清晰原则，并且和人以及计算机成功沟通。

对于什么有助于生成可维护的代码，Unix 程序员有许多隐性知识，原因是 Unix 有许多可以追溯到几十年前的源码。基于我们在第 17 章将要讨论的原因，Unix 程序员学到了一种品性，就是宁愿抛弃、重建代码也不愿修补那些蹩脚的代码（参见第 1 章 Rob Pike 对此的考量）。这样，那些顶住了数十年发展压力而留存下来的任何源码都经历了可维护性的考验。这些古老、成功、构建良好的项目和可维护代码一起成为 Unix 社区实践的好榜样。

在评价工具以供使用时，Unix 程序员——特别是开源世界的 Unix 程序员——学会提出的一个问题是：“代码是活代码、睡代码还是死代码？”活代码周围存在一个非常活跃的开发社团。睡代码之所以“睡着”，经常是因为对作者而言，维护代码的痛苦超过了代码本身的效用。死代码则是睡得太久，重新实现一段等价代码更容易。如果希望让代码成为活代码，则最有效的时间花费方法之一就是投入精力使代码具备可维护性（并以此吸引未来的维护者）。

为了透明性和可显性而设计的代码已经朝着可维护性的目标前进了很多。但是我们从本章所举的实例项目中还能观察到其它一些值得效仿的实践。

一个非常重要的实践就是应用清晰原则：选择简单的算法。在第 1 章，我们引用了 Ken Thompson 的话：“拿不准，用穷举”。Thompson 完全明

白复杂算法的代价——不仅在于开始实现时更容易出 bug，而且更在于维护者要完全理解这些算法更困难。

另一个重要的实践是要包含开发者手册（hacker's guide）。在发布源码的同时包含指导文档，简略地描述代码的关键数据结构和算法，这种做法永远得到高度认可。实际上，跟编写最终用户文档相比，Unix 程序员常常更善于编写开发者手册。

开源社区已经抓住并且细述了这个习惯。除了能给未来的维护者提出建议之外，开源项目的开发者指南也是为了便于临时贡献者增加功能和修改 bug 而设计的。随 *fetchmail* 一起发布的 Design Notes（设计笔记）文件就是其中的代表。Linux 内核源码实际上也包括很多这样的文档。

我们将在第 19 章描述 Unix 开发者为了让源码发布便于校验和编译执行码所形成的一些约定。这些实践也促进了代码的可维护性。

多道程序设计：分离进程为独立的功能

> 如果我们相信数据结构，我们就必须相信独立的（因而是并发的）处理方式。我们还会为了其它原因在结构内收集数据吗？为什么我们要容忍那些只给其一不给其二的语言？
>
> —《编程隽言》, ACM SIGPLAN(1982 17 卷 9#)
>
> —Alan Perlis

Unix 最具特点的程序模块化技法就是将大型程序分解成多个协作进程。这在 Unix 世界中通常叫做“多处理”，但在本书中我们将恢复使用老的术语“多道程序设计”，以避免和多处理器的硬件实现相混淆。

多道程序设计是设计中的荒蛮之地，几乎没有好的实践方针。许多程序员尽管精于判断如何将代码分解成子过程 (subroutine)，然而最终还是编写出单个庞然大物般的单进程程序，而这些程序往往失败在自身的内部复杂度之上。

无论在协作进程还是在同一进程的协作子过程层面上，Unix 设计风格都运用“做单件事并做好”的方法，强调用定义良好的进程间通信或共享文件来连通小型进程。因此，Unix 操作系统提倡把程序分解成更简单的子进程，并专注考虑这些子进程间的接口。这至少可通过以下三种方法来实现：

- 降低进程生成的开销。
- 提供方法（shellout[shell 执行模块]、I/O 重定向、管道、消息传递和套接字）简化进程间通信。

- 提倡使用能由管道和套接字传递的简单、透明的文本数据格式。

平价的进程生成和简单的进程控制对能否以 Unix 风格编程起着关键作用。在 VAX VMS 之类的操作系统中，启动进程开销极大、速度缓慢并且需要特别的权限，因此人们别无选择，必须编制单个庞然大物般的程序。幸运的是，Unix 家族操作系统的倾向一直是减轻而不是加重 *fork*(2) 的开销。特别是 Linux，尤擅此道，使得进程生成要比许多其它操作系统的线程生成还要快[1]。

历史上，shell 编程的经历鼓励了许多 Unix 程序员从多个协作进程的角度思考问题。创建由管道连接的多进程组，在后台或前台运行，或者在后台和前台都运行，相对来讲，在 shell 中较容易实现。

在本章剩余部分，我们将分析廉价进程生成的含义，并讨论如何及何时应用管道、套接字和其他进程间通信（IPC）方法将设计划分成协作进程（再下面一章，我们将对接口设计应用同样的功能分离原理）。

尽管将程序划分成协作进程带来了全局复杂度降低的好处，但代价是我们必须更多地关注在进程间传递信息和命令的协议设计（在所有种类的软件系统中，接口都是 bug 聚集之地）。

我们在第 5 章分析了这个设计问题的底层——如何统筹透明、灵活、可扩展的应用协议。但是还存在另一个而且是更高一层的设计问题，我们当时冒失地忽略了。这就是为通信各方设计状态机的问题。

给定模型，如 SMTP、BEEP 或 XML-RPC，则在应用协议语法中运用良好的风格并不难。真正的挑战不是协议语法而是**协议逻辑**——设计一个协议，既有充分的表达能力又有防范死锁的能力。几乎同样重要的是，协议必须**看得出**很有表现力并可防范死锁：也就是说人们必须能够在头脑中尝试对通信程序的行为建模并验证其正确性。

---

[1] 例如，参考《Linux 下提高空闲任务的上下文切换性能》（Improving Context Switching Performance of Idle Tasks under Linux）[Appleton] 引用的结果。

因此，我们的讨论将着眼于那种可以很自然地运用于各种进程间通信的协议逻辑。

从性能调整中分离复杂度控制

然而，首先，我们必须排除一些分神之事。我们的讨论**并非**关于利用并发性提升性能。在开发出可以把全局复杂度降至最低程度的干净体系之前，关注性能问题便是过早优化——万恶之源（详细讨论参见第 12 章）。

一个密切相关的干扰话题是线程（也就是，共用同一存储地址空间的多个并发进程）。使用线程是为了调整性能。为了避免长时间分散注意力，我们将在本章结束时更详细讨论线程。总的来说，线程不是降低而是提高了全局复杂度，因此，除非万不得已，尽量避免使用线程。

而说说模块化原则，则不算跑题：它能使你的程序——和你的生活——更轻松。进程分解的所有理由，都是我们在第 4 章提出的模块划分缘由的延续。

另一个把程序划分成多个协作进程的重要原因是为了更强的安全性。在 Unix 下，必须由普通用户运行的程序，如果又必须拥有对安全性至关重要的系统资源的写访问权限，可以通过一个叫 *setuid bit* 的特性获得[2]。可执行文件是可以拥有 setuid bit 的最小代码单元；因此，必须信任 setuid 可执行程序中的每一行代码（然而，写得好的 setuid 程序首先完成所有需要特权的行为，剩下的时间则把权限交还给用户级）。

通常，一个 setuid 程序只在一个或很少几个操作中需要特权。常常可以把这样的程序划分成两个协作进程，小进程需要 setuid，大的则不需要。当我们能够这样做时，只须信任较小程序的代码。Unix 比竞争对手拥有更好的

[2] setuid 程序并不以调用该程序的用户权限运行，而是以可执行文件拥有者的权限来运行。这个特性可以用来对口令文件等不允许非系统管理员直接修改的文件实施受限的、程序控制的访问权管理。

安全记录，很大一部分就是因为这种划分和委托在 Unix 中是可行的[3]。

Unix IPC 方法的分类

和单进程程序体系一样，最简的就是最好的。本章剩余部分将大致按照编程技术复杂度由低到高的顺序介绍各种 IPC 技法。在使用更晚出现、更复杂的技法前，应该通过实证——用原型和基准检测结果——所有更早出现、更简单的技法都不管用。经常，你会把自己吓一跳。

把任务转给专门程序

廉价的进程生成使程序间的协作变为可能，其中最简单的形式就是一个程序调用另一个程序来完成专门任务。被调用的程序经常通过 *system*(3) 的调用被指定为一个 Unix Shell 命令，因此这通常称做对被调用序“shell out”（外壳执行）。被调用的程序在运行完毕之前接管用户的键盘和显示，退出后，调用程序重新控制键盘和显示并继续运行[4]。因为调用程序并不和被调用程序通信，在被调用程序执行起见，协议设计在这种协作类型中并不成问题，除了（有点琐碎了）调用程序可能传递命令行参数到被调用程序中去改变它的行为。

经典的 Unix shellout 实倒是在邮件或新闻组程序中调用文本编辑器。Unix 的传统中，不要求使用常规文本编辑输入的程序捆绑专门用途的编辑器。相反，允许用户在需要编辑时指定自选的文本编辑器以供调用。

专门程序通常借由文件系统与父进程进行通信，方法是在指定位置读取或修改文件；编辑器或邮件器的 shellout 就是这样工作的。

---

[3] 也就是，按在互联网上每台机器多长时间内会被攻陷来衡量而得出的更好记录。

[4] shellout 编程的一个常见错误是子进程运行时忘记在父进程中阻塞信号。如果不对此防范，进入子进程的中断可能会对父进程造成不期望的副作用。

这种模式的常见变形是专门程序可以接受标准输入，用 C 库的`popen(… ,"w")`或作为 shell 脚本的一部分进行调用。或者专门程序也可以有标准输出，用`popen(… , "r")`或作为 shell 脚本的部分加以调用（如果它既从标准输入读入又向标准输出写出，则可以通过批量方式完成，即完成全部读操作之后才进行写操作）。通常不把这种子进程称为 shellout；目前还没有一个标准术语称呼这个子进程，但完全可以称其为“bolt-on”（栓合）。

这些情态的要点在于专门程序在运行时并不需要跟父进程交流。只有在任意一方（主进程或从进程）接受了另外一个程序的输入，还必须能够对此解析这个意义上来说，它们之间才存在关联协议。

实例分析：mutt 邮件用户代理

mutt 邮件用户代理是现代 Unix 邮件程序中最重要设计传统的典型。它有一个简单的面向屏幕的界面，使用单键命令来浏览和读取邮件。

使用 *mutt* 撰写邮件（以地址为命令行参数进行调用或使用回复命令）时，它检查进程环境变量 EDITOR，然后生成一个临时文件名。环境变量“EDITOR”（编辑器）的值作为命令调用，并以临时文件名为参数[5]。当命令终止时，mutt 恢复控制运行，并认定临时文件包含的文本就是所要的邮件正文。

几乎所有的 Unix 邮件和网络新闻撰写程序都遵循同样的约定。正是因此，创作程序的实现者可以不必编写上百个截然不同的编辑器，用户也不必学习上百个不同的界面。相反，用户可以使用最喜欢的编辑器。

这种策略的重要变形通过 shell 调用一个小型代理程序，让其向正在运行的大程序实例，如编辑器或网页浏览器等传递专门任务。这样，通常已经在 X 显示上运行 *emacs* 实例的开发者可以设置 **EDITOR=emacsclient**，当需要在 *mutt* 中编辑时，在 *emacs* 编辑一块缓存。这样做的目的并不是真正要

[5] 实际上，上述过程有点过分简化了。完整部分请参考第 10 章关于 EDITOR 和 VISUAL 的讨论。

节省内存或其它资源，而是为了让用户能够把所有的编辑动作统一到单独的 *emacs* 进程中（这样一来，举例来说，在不同缓存间进行剪切和粘贴操作时可以捎带诸如字体高亮等 emacs 的内部状态信息）。

管道、重定向和过滤器

继 Ken Thompson 和 Dennis Ritchie 之后，对早期 Unix 影响最重要的人大概非 Doug McIlroy 莫属。他发明的**管道**结构始终闪现在 Unix 设计中，促进了“做单件事并做好”哲学的萌发，并且引发了 Unix 设计中绝大多数后续 IPC 方法的诞生（尤其是用于网络的套接字抽象概念）。

管道依赖这样的约定，即每个程序一开始（至少）有两个 I/O 数据流可用：标准输入和标准输出（文件描述符数字分别为 0 和 1）。许多程序都可写作过滤器，从标准输入顺序读数据，并且只向标准输出写数据。

通常，这些数据流分别和用户的键盘和显示器相连接。但是 Unix shell 普遍支持重定向操作，可以把这些标准输入输出流连接到文件。举例来说，键入：

```
ls >foo
```

把列目录命令 *ls*(1) 的输出写入到名为“foo”的文件中。另一方面，键入：

```
wc <foo
```

将令字数统计程序 *wc*(1) 以文件“foo”为输入，然后把字符数/字数/行数发送到标准输出。

管道操作把一个程序的标准输出连接到另一个程序的标准输入。用这种方式连接起来的一系列程序被称为管线。如果我们键入：

```
ls | wc
```

我们可以看到当前目录列表的字符数/字数/行数[6]（在这种情况下，只有行数有真正意义）。

> 管道一个讨人喜欢的管道线是**"bc | speak"**——一个能说话的桌面计算器。这个计算器知道1×10^{63}以下数字的叫法。
>
> —Doug McIlroy

管道线中所有阶段的程序是并发运行的，注意到这一点很重要。每一段等待前一段的输出作为输入，但在下一段能够运行前没有哪个段必须退出。这个特性在我们接下来分析管道系统的交互作用时非常重要，例如把某个命令的超长输出发送给 *more*(1)。

人们很容易低估管道和重定向的组合能力。《作为第四代语言的 Unix Shell》(Unix Shell As a 4GL) [Schaffer-Wolf] 是一个很有启发意义的例子，该书指出，以这些功能为框架，组合一些简单实用程序，就可以创建和操控简单文本表格形式的关系数据库。

管道的主要缺点是单向性。管道线的成员除了终止外（在这种情况下，前一阶段的程序会在下一个写操作时得到 **SIGPIPE** 信号）不可能回传控制信息。因此，传输数据的协议简化为接受端的输入格式。

目前为止，我们已经讨论了 shell 创建的匿名管道。还有一种变种，叫做“命名管道”，是一种特殊的文件。如果两个程序打开这个文件，一个读取，另一个写入，则命名管道扮演两者间的配接器。命名管道已经成了老古董：在使用中，大部分已经被我们下面将要讨论的命名套接字取代了（有关这个老古董的更多细节，参考以下 System V IPC 的讨论）。

---

[6] 现在的顺序是行数/字数/字符数（newline, word, and byte counts）。

实例分析：为分页程序建立管道

管线有很多用处。举一个例子来说，Unix 的进程列举程序 *ps*(1) 在标准输出上列举进程时，并不关心长列表在用户的滚屏速度太快会造成用户无法看清。Unix 还有另外一个程序 *more*(1)，它按屏幕大小显示标准输入，每次满屏显示后等待用户按键显示下一满屏。

这样，如果用户键入**"ps | more"**，把 *ps*(1) 的输出管道连接至 *more*(1) 的输入，则每次按键后可以继续显示一整屏的进程列表。

如此组合程序的能力极为有用。但此处真正的成果并不是这个漂亮的组合，而是由于管道和 *more*(1) 两者的存在，其它程序可以变得更简单一些。管道意味着 *ls*(1)（以及其它写标准输出的程序）之类的程序无需开发自己的分页程序——我们得以从一个到处都是内置分页程序（自然，每个分页程序都有不同的观感）的世界中解救出来。这就避免了代码的臃肿，降低了全局复杂度。

一个额外好处是，如果需要定制分页程序行为，可以只在一个地方、改变一个程序就行了。确实，可以存在多个分页程序，并且这些分页程序对每一个写标准输出的应用程序都非常有用。

实际上，这确实已经发生了。现代 Unix 中，*more*(1) 基本上已被 *less*(1) 取代。*less*(1) 对显示的文件增加了向后滚屏的能力，不再只能向前滚屏[7]。因为 *less*(1) 独立于使用它的程序，所以只需在 shell 中简单地将“less”指定“more“的 alias，把环境变量 PAGER（分页器）设置成“less”（参考第 10 章），然后就可以在所有正确编写的 Unix 程序中享受到更好分页程序所带来的全部好处了。

---

[7] *less*(1) 的手册页解释说，这个名字遵循了“Less is more”（少即是多）。

实例分析：制作单词表

一个更加有趣的例子是通过管道相连的程序来协作完成某种数据变换。在没有这么灵活的环境中，要实现这点就必须定制代码。

考虑以下管线:

```
tr -c '[:alnum:]' '[\n*]' | sort -iu | grep -v '^[0-9]*$'
```

第一个命令把标准输入中非字母和数字的字符在标准输出上转换为新行。第二个命令对标准输入的行进行排序，对于所有重复的相邻行只保留一个，然后把排好序的数据写到标准输出。第三个命令去掉所有只含数字的行。合起来，这些操作把标准输入的文本生成了经过排序的单词表送到标准输出。

实例分析：pic2grap

程序 *pic2graph* 的 shell 源码和自由软件基金会的 *groff* 文本格式化工具套件一起发布。它把用 PIC 语言编制的图表转换为位图图像。例 7.1 展示了这个代码核心部分的管线。

例 7.1 pic2grah 管线

```
(echo ".EQ"; echo $eqndelim; echo ".EN"; echo ".PS";cat;echo ".PE")|\
     groff -e -p $groffpic_opts -Tps >${tmp}.ps \
     && convert -crop 0x0 $convert_opts ${tmp}.ps ${tmp}.${format} \
     && cat ${tmp}.${format}
```

pic2graph(1) 实现展示了仅仅依靠调用现有工具的管线能够完成多少任务。管线首先把其输入揉制成适当的形式，然后将其传入 *groff*(1) 以生成 PostScript，最后从 PostScript 转换成位图。它对用户隐藏了所有细节，用户只看到 PIC 源从一端进入，另一端产生了可包含在网页中的位图。

这是一个有趣的例子，因为它说明了管道和过滤器怎样使程序适应非预期的用法。解释 PIC 的程序 *pic*(1)，最初只是为了在排版文档中嵌入图表设计的。在包含 *pic*(1) 程序的一套工具中，绝大多数程序都行将过时。但是 PIC 在新用法中仍然十分便利，比如描述内嵌在 HTML 中的图表等。把 *pic*(1) 的输出转换为更现代格式所需要的全部机制，可以由 *pic2graph*(1) 这类工具捆绑在一起，所以它也获得了新生。

作为微型语言的例子，我们将在第 8 章进一步分析 *pic*(1)。

实例分析：bc(1) 和 dc(1)

开始于版本 7 的经典 Unix 工具包包括一对计算器程序：*dc*(1) 程序是一个简单的计算器程序，从标准输入端接受逆波兰标记法（RPN）的文本行，并向标准输出发送计算结果；*bc*(1) 程序接受类似传统代数记数法的更加复杂的中缀表示法，它同样具备设置和读取变量并为复杂公式定义函数的能力。

尽管 *bc*(1) 目前的 GNU 实现版本不依赖其它程序，其经典版本却通过管道把命令传递给 *dc*(1)。在这种分工中，*bc*(1) 完成变量代入和函数展开，并将中缀表示法转换为逆波兰表示法——但实际上本身并不完成计算，相反，把输入表达式转换成 RPN 形式传递给 *dc*(1)，由 *dc*(1) 来完成计算。

这种功能分离具有明显的优势。它意味着用户可以选择自己喜欢的表示法，却无需重复实现任意精度的数字计算逻辑（这有点需要技巧）。组合中的每一个程序都没有任何需要选择表示法的程序复杂。这两个程序既可以独立调试，也可以独立建模。

我们将在第 8 章从专门领域的微型语言这个略有不同的角度再次分析这些程序。

反例分析：为什么 fetchmail 不是管线

按照 Unix 的界定，fetchmail 是个令人不舒服的庞大程序，选项繁多。想想邮件传输的工作方式，有人可能想把它分解成一个管线。暂且假设它已被

分解成这几个程序：一对从 POP3 和 IMAP 站点获取邮件的读取程序，一个本地的 SMTP 队列放置器 (injector)。管线应该能够传递 Unix 信箱格式。当前复杂的 *fetchmail* 配置完全可以由包含命令行的 shell 脚本来取代，甚至还可以在管道线中插入过滤器来拦截垃圾邮件。

```
#!/bin/sh
imap jrandom@imap.ccil.org | spamblocker | smtp jrandom
imap jrandom@imap.netaxs.com | smtp jrandom
# pop ed@pop.tems.com | smtp jrandom
```

这可能会非常优雅，非常 Unix 化。不幸的是，这行不通。我们在早些时候已经简略谈到了其中的原因：管线是单向的。

取信程序（imap 或 pop）必须要完成的一件事是决定应否为已取回的每一条消息发送一个删除请求。在 *fechmail* 目前的结构中，它可以延迟向 POP 或 IMAP 服务器发送删除请求，直到得知本地 SMTP 监听程序已经接管该消息。由管道连接的组件版本将丢失这个特性。

举例来说，请考虑，如果因为 SMTP 监听程序报告磁盘已满导致 SMTP 队列放置程序失败，会发生什么呢？如果取信程序已经删除了这个邮件，我们就完了。这意味着，smtp 队列放置器通知取信程序删除邮件之前，取信程序不能删除邮件。这反过来产生了一大堆问题。他们之间如何通信？确切地说，队列放置器应该返回什么样的消息呢？相应系统的全局整体复杂度以及易出错性，几乎肯定要比单块程序高。

管线是一个了不起的工具，但不是万能的。

包装器

和 shellout 程序相对的是**包装器** (wrapper)。包装器或者将被调用程序专用化，或者为它创建新的接口。包装器经常用于隐藏 shell 管线的复杂细

节。我们将在第 11 章讨论接口包装器。大多数专用化包装器都相当简单，但非常有用。

和 shellout 一样，由于被调用程序执行过程中程序之间并不通信，因此也不存在相关协议；但是，包装器之所以存在，常常源于要指定参数来修改被调用程序的行为。

实例分析：备份脚本

专用化包装器是 Unix shell 和其它脚本语言的经典用途。既常用又典型的一类特化包装器是备份脚本，它可能简单到只有这样的一行：

```
tar -czvf /dev/st0 "$@"
```

这是一个为 *tar*(1) 磁带归档程序编写的包装器，这个包装器只简单地提供一个固定参数（磁带设备/dev/s t0），并把用户提供的其它参数（ "$@")[8]传递给 tar。

安全性包装器和 Bernstein 链

包装器脚本的常见用法是安全性包装器。安全性包装器可调用守门程序检查某类凭证，然后根据返回的状态值有条件地执行另一个程序。

Bernstein 链（ Bernstein chaining ）是一个专用化的安全性包装器技法，由 Daniel J.Bernstein 首先发明。Bernstein 在他的许多程序包中都使用了安全包装器（在 *nohup*(1) 和 *su*(1) 之类命令中使用类似的手法，但是无法检查条件值）。从概念上来说，Bernstein 链和管线类似，只不过每个继发阶段的程序取代了前一阶段的程序，而不是与之并行。

[8] 一个常见的错误是使用$* 而不是"$@"。这在传递含有空格的文件名时会出问题。

通常的应用是把较高安全级别的应用程序限制在某类门卫程序中，由这个程序把状态传递给一个权限较低的程序。这种技法使用一组 exec，或者综合 fork 和 exec 把几个程序粘在一起。所有程序名都在一个命令行中得到指定。每个程序执行某个功能，（如果成功的话）用命令行剩余部分调用 *exec*(2)。

Bernstein 的 *rblsmtpd* 包就是一个原型例子。它用于在邮件滥用防御系统（Mail Abuse Prevention System）的垃圾邮件 DNS 城中查询主机。方法是通过载入环境变量“TCPREMOTEIP”的值取得 IP 地址，然后对其进行 DNS 查询。如果查询成功，*rblsmtpd* 就运行自己的 SMTP 以丢弃这个邮件。如果不成功，它就假定剩余的命令行参数构成一个知道 SMTP 协议的邮件传输代理，交给 *exec*(2) 来运行。

Bernstein 的 qmail 包是另外一个例子。qmail 包含有 condredirect 程序。第一个参数是个邮件地址，剩余部分是门卫程序及其参数。condredirect 首先 fork，然后对门卫程序带参数进行 exec。如果门卫程序成功退出，condredirect 就把在标准输入端（stdin）的待处理邮件发送到指定的邮件地址。在本例子中，与 rblsmtpd 相反，安全策略由子进程决定。这种情况更像经典的 shellout。

更精巧的例子是 qmail 的 POP3 服务器。它由三个程序构成，即 qmail-popup、checkpassword 和 qmail-pop3d。checkpassword 程序是一个独立包，很聪明地命名为 checkpassword，而且让人一点不意外的是，它就是用来检查口令的。POP3 协议包括认证阶段和邮箱操作价段：一旦进入邮箱操作阶段就不能回到认证阶段。这是 Bernstein 链的一个完美应用。

qmail-popup 的第一个参数是 POP3 提示中使用的主机名。取回 POP3 的用户名和口令后，进行 fork 操作，并将其余参数传递给 *exec*(2)。如果程序返回失败，口令肯定错误，qmail-popup 就报告这个错误并等待一个不同的口令。反之，程序就假设已经完成 POP3 对话，qmail-popup 从而退出。

qmail-popp 命令行中指定的程序要从文件描述符 3 处读取三个以 null 结束的字符串[9]。它们是用户名、口令，以及如果有的话，对密码的攻击回

[9] qmail-popup 的标准输入和标准输出是套接字，标准错误（文件描述符为 2）被转向为

应。这一次应该是 checkpassword 老 qmail-3d 的名称及其参数作为参数接受。如果口令不符合，checkpassword 程序就失败退出；反之，它就切换到用户的 uid、gid 和主目录下，并以用户身份执行剩下的命令。

如果应用程序需要 setuid 或 setgid 优先权来初始化连接或获得某些认证，然后放弃这些权限让后续代码无须被信任，这时 Bernstein 链非常有用。在 exec 后，子程序无法把用户真实 ID 设置回 root 用户。同时，这样做也比单进程灵活，因为可以在程序链中插入另一个程序来修改系统的行为。

例如，rblsmtpd（前面已提到）可以插进 Bernstein 链，放在 tcpserver（来自 ucspi-tcp 包）和真正的 SMTP 服务器（通常是 qmail-smtpd）之间。当然，它也跟 *inetd*(8) 和 **sendmail -bs** 合作得很好。

从进程

有时候，子程序通过连接到标准输入和标准输出的管道，变互地和调用程序收发数据。同简单的 shellout 以及我们前面称为“bolt-on”的机制不同之处在于，主进程和从进程都需要内部状态机处理它们之间的协议以避免发生死锁和竞争。比起简单的 shellout，这种结构远远更复杂、更难以调试。

Unix 的 *popen*(3) 调用可以为 shellout 搭建输入或输出管道，但是不能为从进程搭建这两种管道——这似乎有意鼓励更简单的编程。而且，事实上，交互的主/从通信极为复杂，通常只在这两种条件下使用：（a）两者间涉及的协议完全无足轻重，或（b）从进程是为以我们在第 5 章讨论的应用协议进行通讯而设计的。我们将在第 8 章再次分析这个问题并讨论解决方法。

在编写主/从进程对时，一个好方法是，让主进程支持命令行开关或环境变量来允许调用者设置自己的从进程命令。抛开其它好处，这尤其有利于调试：经常可以发现，在开发过程中，从监视和记录主从进程之间事务处理的辅助程序中调用真正的从进程，会带给你很多方便。

日志文件。必须保证下一个分配得到的文件描述是 3。正如一段臭名昭著的的内核注释所写的：“不指望你能理解这个”。

如果发现程序中主/从进程的交互不再微不足道，剩下的，也许是考虑使用套接字或共享内存等技法，走更趋对等结构的路了。

实例分析：scp 和 ssh

两者间通信协议的确无足轻重的一个常见情况是进度显示程序。*scp*(1) 安全拷贝命令把 *ssh*(1) 作为从进程调用，从 ssh 的标准输出中截取足够的信息，然后把报告重新组织成为 ASCII 动画形式的进度条。[10]

对等进程间通信

迄今我们已经讨论的各种通讯方法都存在隐含的层次关系，即一个程序实际上控制或驱动另一个程序，而在反方向却没有或仅有有限的反馈。在通信和网络中，我们常常需要**对等的**通道，通常（但不一定）需要数据能自由地双向流动。接下来，我们将分析 Unix 中的对等通信方法，并在以后几章中逐步展开一些实例分析。

临时文件

把临时文件作为协作程序之间的通信中转站使用，是最古老的 IPC 技法。虽然存在一些缺点，但临时文件在 shell 脚本及一些一次性程序中仍然非常有用，在这些程序中使用其它更复杂、更需协调的通信方式未免有点小题大做。

把临时文件作为 IPC 技法使用最明显的问题是，如果进程在临时文件可被删除前中断，则往往会遗留垃圾数据。另一个更隐蔽的风险是，如果程序的多个实例都使用同一个名字作为临时文件名，则会产生冲突。这就是为什么 shell 脚本的惯例是在临时文件名中包含“`$ $`”符号的原因；这个 shell 变量

10　一个推荐这个实例分析的朋友评论：“是的，是可以不用这种技法……如果从进程返回的信息中恰有几块容易识别的天然贵金块，而你正好又有夹子和防辐射服。”

将被展开为载入 shell 的进程 ID，从而有效地保证了文件名的唯一性（Perl 也支持同样的技巧）。

最后，如果攻击程序知道临时文件将要写入的位置，就可以覆盖掉那个文件，很可能读取生产者进程的数据，或者通过在文件中插入修改或假造的数据哄骗消费者进程。[11]这可是一种安全性风险。如果涉及的进程拥有 root 用户权限，风险就更为严重。当然可以通过仔细设置临时文件目录的权限来缓解这个问题，但很多人都知道，这种部署很难万无一失。

撇开所有这些缺点，临时文件仍然还有一席之地；因为它们很容易创建，很灵活，与那些复杂的方法相比，没那么容易产生死锁和竞争。而且，有时候，别的方法都派不上用场。子进程的调用约定可能要求子进程必须得到一个文件来操作。我们在本章所举的第一个例子，即编辑器 shellout 出的子进程就是个完美的实证。

信号

要让同一台机器上的两个进程相互通信，最简单最原始的方法是一个进程向另一个发送信号。Unix 的信号是一种软中断形式：每个信号都对接收进程产生默认作用（通常是杀掉它）。进程可以声明**信号处理程序**，让信号处理程序覆盖信号的默认行为；处理程序是一个与接受信号异步执行的函数。

信号被设计进 Unix，最初是作为操作系统就某些错误或关键事件通如程序的一种机制，并不是作为 IPC 功能设计进来的。举例来说，**SIGHUP** 信号在会话结束时被发送给每一个从该指定终端会话启动的程序。**SIGINT** 信号，是在用户键入当前定义的中断字符（通常是 control-C）时发送给当前每一个连接键盘的程序。然而，信号对一些 IPC 情形也很有用（而 POSIX 标准信号集就是为了这个使用目的才包含 **SIGUSR1** 和 **SIGUSR2** 两个信号的）。它们经常用作**守护程序**（在后台不间断运行而且不可见的程序）的控制

[11] 这种攻击特别危险的变形就是在生产者进程和消费者进程都期望是临时文件的地方，放一个同名的命名 Unix 域套接字。

通道，操作者或另一个程序用来通知守护程序重新初始化自身，或醒来执行工作，或向已知位置写入内部状态/调试信息的方法。

我坚持认为 **SIGUSR1** 和 **SIGUSR2** 是为 BSD 发明的。人们抓走了一些系统信号，使之成为他们所希望的 IPC 含义。这样一来，（例如），由于 **SIGSEGV** 已被挪用，所以出现分段错误的程序就不会进行核心转储 (coredump) 了。

> 这是一个普遍原则——人们总想挪用你写的任何工具，所以必须把它们设计成要么根本无法挪用要么总是可以干净地挪用。这是你仅有的选择。当然，除非被忽略——这是一个非常可靠的保持清白的方法，但并非最初看起来的那样满意。
>
> —Ken Arnold

随信号 IPC 经常使用的一种技法是所谓 pidfile。需要信号的程序会向已知位置写入一个包含进程 ID(PID) 的小文件（通常放在/var/run 或调用用户的主目录下）。其它程序可以读取这个文件来获得 PID。如果守护程序只允许一个实例运行，则 pidfile 也可作为隐含的文件锁使用。

实际上存在两种不同口味的信号。在老一点的实现（特别是如 V7、System Ⅲ 和早期的 System V）中，给定信号的处理程序无论何时启动，该处理程序就会被复位为信号的默认值。因此，无论处理程序如何设置，快速连续发送两个同样的信号通常就会杀死进程。

BSD 4.x 版本的 Unix 变成了“可靠”的信号，除非用户明确要求，它是不会复位的。它们同时也引入了原语操作，阻塞或临时挂起指定信号集的处理。现代 Unix 支持这两种方式。应该对新代码使用 BSD 风格的不复位入口方法，但是必须进行防预性编程，以防代码移植到不支持这种方法的实现中。

收到 N 个信号并不一定 N 次调用信号处理程序。在老一点的 System V 信号模型中，如果两个或两个以上的信号间隔很小（更确切地说，是在目标进程的同一个时间片内），可能产生各种竞争[12]或异状。根据系统支持的信号语

12 “竞争”是这样的一类问题，在这类问题中，系统的正确行为取决于两个独立事件按照

义的不同，第二个或更后面的信号可能会被忽略掉，可能会误杀一个进程，也可能延迟到前面的实例已经处理完后才发送（在现代 Unix 中，这种情况最可能发生）。

现代的信号 API 可以移植到最近所有的 Unix 版本中，但是不能移植到 Windows 或传统（OS X 以前）的 MacOS 中。

系统守护程序和常规信号

许多著名的系统守护程序都接受 **SIGHUP**（最初，这个信号在有串行线事件时发送给程序，比如挂断调制解调器连接的操作就产生这个信号）作为重新初始化的信号（也就是说，重新载入配置文件）：例子包括 Apache 和 Linux 的 *bootpd*(8)、*gated*(8)、*inetd*(8)、*mountd*(8)、*named*(8)、*nfsd*(8) 和 *ypbind*(8) 实现。在一些例子中，**SIGHUP** 作为其对话关闭信号的最初意义被程序接收（特别是 Linux 的 *pppd*(8)），但是这种作用现在通常都交给 **SIGTERM** 了。

SIGTERM（”终止“）常常扮演温和的关闭信号（区别于 **SIGKILL**，立即杀死进程，而且本身不能被阻塞或另外处理）。**SIGTERM** 的行为通常包括清除临时文件和强制把最新更新刷回数据库以及其它一些类似行为。

在编写守护程序时，请遵循最小立异原则：使用这些约定，阅读手册来寻找现有模型。

实例分析：fetchmail 的信号使用

fetchmail 实用程序通常被配置成在后台运行的守护程序，无需用户干预就能定期从运行控制文件定义的所有远程站点收集邮件，并传送给端口 25 的本地 SMTP 监听程序。为了避免一直占用网络，fetchmail 在收集邮件期间根据用户定义的时间间隔休眠（默认值是 15 分钟）。

正确顺序发生，但是又没有机制来保证这两个独立事件实际上能够这样发生。竞争产生间歇性的、与时间有关的问题，非常难以调试。

当不带参数调用 **fetchmail** 时，它首先检查是否已经有 fetchmail 守护程序在运行（通过查找 pidfile 来完成）。如果没有，fetchmail 采用运行控制文件指定的控制信息正常启动；相反，如果有，fetchmail 新进程仅仅发信号通知老进程，使其立即苏醒并收集邮件，然后新进程就会终止。此外，**fetchmail -q** 向任何正在运行的 fetchmail 后台程序发送终止信号。

这样，键入 **fetchmail** 实际上意味着“现在选出并留一个运行的后台程序等待以后查询；不要拿后台程序是否已经运行这样的细节来烦我”。注意，至于用哪个具体信号来唤醒和终止的细节就无需用户知道了。

套接字

套接字作为一种封装网络数据访问的方法从 Unix 的 BSD 一脉中发展而来。通过套接字通信的两个程序通常都存在双向字节流（存在其它套接字模式和传输方法，但是重要性不大）。字节流既是按序的（也就是说，即使按单个字节发送也按照发送顺序来接收）又是可靠的（套接字用户得到保证，底层的网络将进行错误检测和重发以确保交付）。套接字描述符一旦获得，行为基本上和文件描述符一样。

> 套接字和读/写操作有一个重要区别。如果发送的字节已经到达，但是接收的机器却没有 ACK 确认，发送机器的 TCP/IP 栈将超时。因此得到一个错误不一定意味着字节没有到达；接收端可能已经用上了这些字节。这个问题对可靠协议的设计具有深远的影响，原因是即使不知道哪些数据已经被接收到，也仍然必须正确工作。本地输入/输出（I/O）就是“是/否”的判断，而套接字输入/输出（I/O）则是“是/否/也许”的判断。而且没有任何东西能够确保交付——远端的机器说不定已经被彗星撞毁了。
>
> —Ken Arnold

在创建套接字的时候，可以指定**协议族**来告诉网络层如何解释套接字的名称。人们通常认为套接字和互联网有关，是一种在不同主机上运行的程序之间传递数据的方法，这是 AF_INET 套接字族。在这个套接字族中，地址被解释为主机地址和服务编号对。然而，AF_UNIX（也称为 AF_LOCAL）协议族支持同样的套接字抽象，作为在同一台机器上（名字被解析为特殊文件

的位置，与双向命名管道类似）两个进程之间的通信方式。举个例子，使用 X window 系统的客户端程序和服务器程序通常就使用 AF_LOCAL 套接字进行通信。

所有现代 Unix 都支持 BSD 风格的套接字，一个设计事实是，无论协作进程在何处安置，这些套接字通常都是用于双向 IPC 的正确方法。性能压力可能会促使你使用共享内存、临时文件或其它要求更多局部性条件的技法，但是在现代的情况下，最好设想代码需要增加分布式操作。更重要的是，这些局部性设想可能意味着系统的某个部分与其它部分的内在关系过于密切，超过了良好设计能够容忍的程度。经过套接字强化的分离地址空间是一个特性，不是 bug。

要优雅地使用套接字，在 Unix 传统中，首先得设计这些套接字之间使用的应用协议——即一套请求和响应，能够简洁地表达程序通讯的语义。我们在第 5 章已经讨论了设计应用协议的一些主要问题。

所有新近的 Unix 版本、Windows 和经典的 MacOS 都支持套接字。

实例分析：PostgreSQL　PostgreSQL 是一个开源的数据库程序。如果它的实现是单个庞然大物式程序，就会成为单进程程序，有交互式界面，直接在磁盘上处理数据库文件。界面和实现就会结合在一起，一个程序的两个实例，如果试图同时处理同一数据库，就会产生严重的竞争和死锁问题。

相反，PostgreSQL 套件包括一个叫做 postmaster 的服务器程序和至少三个客户应用程序。每台机器的 postmaster 服务器进程在后台运行并拥有对数据库文件的独占访问权。它通过 TCP/IP 套接字接受 SQL 查询语言的请求，并且返回文本格式的结果。当用户运行 PostgreSQL 客户端时，该客户端启动一个和 postmaster 的会话并与之进行 SQL 事务处理。服务器程序可以一次处理好几个客户端会话，并且对这些请求排队，所以它们不会互相干扰。

因为前端和后端是分开的，所以服务器程序除了知道如何解释客户端的 SQL 请求并返回报表外，不需要知道其它任何事情；另一方面，客户端也不

必知道数据库是如何存储的。客户端可以依不同的需求定制并且拥有不同的用户界面。

这种组织形式在 Unix 数据库中相当典型——甚至经常可以把不同 SQL 客户端和服务器程序结合并匹配起来使用。所产生的互操作性问题只是 SQL 服务器端的 TCP/IP 端口号可能不同，以及客户端和服务器是否支持同一种 SQL 语言。

实例分析：Freeciv 在第 6 章，我们把 Freeciv 作为透明数据格式的例子介绍给大家。但是与其支持多人游戏方式比起来，更关键的是代码的客户端/服务器划分。这是一个其应用必须在广域网上分布，并通过 TCP/IP 套接字进行通信的典型例子。

Freeciv 游戏的运行状态由服务器进程，即游戏引擎来维护。玩家运行 GUI 客户端，通过包协议和服务器交换信息和命令。所有的游戏逻辑都在服务器端处理；GUI 的细节在客户端处理，不同的客户端支持不同的界面风格。

这是多人在线游戏非常典型的一种结构。包协议使用 TCP/IP 进行传输，因此服务器可以处理运行在不同互联网主机上的客户端。其它那些更像实时模拟（特别是仿真视角射击游戏）的游戏使用原始的互联网数据报协议（UDP），并且牺牲了包发送的不确定性来降低延迟。在这种游戏中，用户通常连续地发送控制动作，所以零星的信号丢失是可以忍受的，但是延迟却是致命的。

共享内存

尽管使用套接字通信的两个进程可能在不同机器上（实际上，可能跨越半个地球而通过互联网连接起来），而共享内存要求生产者和消费者程序必须在同一硬件上。但是，如果通信进程能够访问同一个物理内存，则共享内存将是它们之间最快的信息传递方法。

共享内存可能以各种 API 的面貌呈现，但是在现代 Unix 中，共享内存的实现通常依靠使用 *mmap*(2)，把文件映射成可以被多个进程共享的内存。

POSIX 定义了具有 API 的 *shm_open*(3) 功能，支持把文件作为共享内存使用，这通常是对操作系统的提示，告诉它无需把伪文件数据刷到磁盘上。

因为对共享内存的访问不能通过类似于读写调用的规范自动序列化，所以处理共享的程序必须自己处理竞争和死锁问题。典型方法是在共享段中使用信号量变量。此处产生的问题和多线程（参考本章末的讨论）产生的问题类似，但是更容易管理，因为默认值是不共享内存。这样，这些问题能够得到更好的遏制。

在共享内存可用并且可靠的系统中，Apache 网页服务器的记录牌功能使用共享内存，作为 Apache 主进程和它管理的 Apache 映像负载分配池之间通信的方式。现代的 X 实现也使用共享内存在位于同一机器上的客户端和服务器之间传递大图像，以避免套接字通信的开销。这些使用都是得到经验和测试证实的性能优化而不仅仅是体系选择。

所有现代的 Unix 都支持 *mmap*(2) 调用，包括 Linux 和开源的 BSD 版本，这在《单一 Unix 规范》(Single Unix Specification) 中有描述。一般来说，Windows、传统的 MacOS 和其它操作系统都还不支持这个调用。

在有特制的 *mmap*(2) 用之前，两个进程通信的常用方法是打开同一个文件然后将其删除。在所有已打开状态的文件句柄都关闭之前，这个文件不会被删除，但是一些老的 Unix 版本把连接计数降到零，作为可以停止更新文件在磁盘数据的提示。不利的方面是，存储回填用的是文件系统而不是交换设备，被删除文件所在的文件系统在使用该文件的程序关闭之前不能卸载，而且把新进程附到 (attach) 已存在的、按这种方式模拟的共享内存段中极为复杂。

在版本 7 以及 BSD 和 System V 系列分开后，Unix 的进程间通信朝两个不同的方向发展。BSD 方向产生了套接字；另一方面，AT&T 系列发展了命名管道（见前面讨论）和一种基于共享内存双向信息队列、为传输二进制数据而专门设计的 IPC 功能。这被称为“System V IPC”——或者，在那些老前辈当中，被称为“Indian Hill”IPC，以 AT&T 的实验室命名，这是其最初编写的地方。

System V IPC 传递消息的上层已经逐渐被废弃了。而由共享内存和信号

量构成的底层，在同一台机器上运行的进程间需要完成互斥锁定和全局数据共享的情况下，仍旧具有非常重要的应用。这些 System V 的共享内存功能发展成了 POSIX 的共享内存 API，Linux、BSD、MacOS X 和 Windows 都支持，但是经典的 MacOS 不支持。

使用这些共享内存和信号量功能（*shmget* (2)、*semget*(2) 及其类似机能）可以避免通过网络栈复制数据的开销。大型商业数据库（包括 Oracle、DB2、Sybase 和 Informix）大量使用这种技术。

要避免的问题和方法

尽管基于 TCP/IP 的 BSD 风格套接字已经成为主流的 Unix IPC 方法，但是人们对于如何通过多道程序达到正确的划分仍然争论不休。一些过时的方法还没有完全消亡，而一些从其它操作系统中移植过来的（通常与图形或 GUI 编程有关）技术值得怀疑。下面我们将在危险的沼泽中游历：小心鳄鱼。

废弃的 Unix IPC 方法

Unix（诞生于 1969 年）比 TCP/IP（诞生于 1980 年）以及二十世纪九十年代和后期无处不在的网络要早多了。匿名管道、重定向和 shellout 很早就在 Unix 中存在了，但是 Unix 的历史充满了与过时的 IPC 及网络模型联系在一起的 API 遗骸，其中出现在版本 6（1976 年）而在版本 7（1979 年）之前废弃的 *mx*() 机能是其始作俑者。

最后，BSD 套接字胜出，成为与网络统一的 IPC。但是这段摸索用了 15 年，期间留下了很多遗迹。因为 Unix 文档可能还会提到这些，可能会让人误解它们仍在使用，所以知道这些非常有用。这些废弃的方法在《Unix 网络编程》(Unix Network Programming)[Stevens90] 中有更详细的描述。

> 对于旧 AT&T Unix 中所有消亡的 IPC 功能的真正解释就是管理策略。Unix 支持小组由一个底层经理领导，而一些使用 Unix 的项目却由副总裁领

> 导。这些副总裁总有办法制造不可抗拒的请求，并且不能容忍大多数 IPC 机制都可互换这样的异议。
>
> —Doug McIlroy

System V IPC

System V IPC 功能是基于我们此前讨论的 System V 共享内存的消息传递机能。

使用 System V IPC 协作的程序通常定义基于短（不超过 8K）二进制信息交换的共享协议。相关的手册页有 *msgctl*(2) 及其相关页。由于这种方法多半已经被套接字间传递的文本协议所取代，在这里我们就不举例了。

System V IPC 功能存在于 Linux 和其它现代 Unix 中。然而，由于它们是一种历史遗留功能，所以不很经常使用。Linux 版本直到 2003 年中期还存在 bug。但似乎没人愿意修复这些 bug。

Steams

Streams networking（网络流）由 Dennis Ritchie 为 Unix 版本 8（1985 年）创造。一个重新的实现叫做 STREAMS（是的，在文档里全是大写字母），最早发布在 System V Unix（1986 年）3.0 版中。STREAMS 机能在用户进程和指定的内核驱动程序之间提供全双工的接口（功能上类似 BSD 套接字，在初始设置之后通过普通的 *read*(2) 和 *write*(2) 操作进行访问）。设备驱动程序可能是串口或网卡之类的硬件，也可能是为了在用户进程之间传递数据而设置的纯软件模拟设备。

Streams 和 STREAMS[13]都具有一个有趣特性，就是可以把协议转换模块推到内核处理线中，这样用户进程通过全双工通道“看到”的设备实际上被

[13] STREAMS 要复杂得多。据传闻 Dennis Ritchie 曾说过“大声说 streams 时，含义就有所不同了”。

过滤掉了。举例来说，这个能力可用于实现终端设备的行编辑协议。或者，可以实现 IP 或 TCP 之类的协议而不用把它们直接整合在内核当中。

Streams 最早是整理所谓“行守则”——另一种处理串行终端和早期局域网字符流的模式——内核凌乱特性的一种尝试。但随着串行终端从人们视野中淡出，以太局域网到处存在，TCP/IP 逐出其他协议栈而归并到 Unix 内核，STREAMS 所提供的特别灵活性越来越无用武之地。在 2003 年，System V Unix 仍然支持 STREAMS，一些 System V/BSD 的混和 Unix 操作系统，如 Digital Unix 和 Sun Microsystem 的 Solaris 也同样支持 STREAMS。

Linux 和其它开源 Unix 实际上已经抛弃了 STREAMS。LiS（Linux Stream）项目的 Linux 内核模块和程序库可从 <http://www.gcom.com/home/linux/lis/> 获得，但是（直到 2003 年中期）它们还没有整合到 Linux 内核主干中。其他非 Unix 操作系统并不支持它们。

远程过程调用

尽管偶有例外，如 NFS（Network File System，网络文件系统）和 GNOME （GNU Network Object Model Environment，GNU 网络对象模型环境）工程，但是引进 CORBA、ASN.1 和其它远程过程调用接口形式的尝试大多失败了——这些技术至今还没有为 Unix 文化所吸纳。

造成这一点似乎有几个根本原因。其中之一是 RPC 接口不是那么容易做到可显的：也就是说，难以按功能查询接口，而如果不编写和被监控程序同样复杂的专用工具，也难以监控程序的行为（我们已经为此在第 6 章讨论了一些理由）。RPC 接口和库一样具有版本不兼容（version skew）问题，但是更难追查，因为它们是分布的，而且不会体现在链接过程中。

与之相关的问题是，类型标记越丰富的接口往往越复杂，因而越脆弱。随着时间的推移，由于在接口之间传递的类型总量逐渐变大，单个类型越来越复杂，这些接口往往产生类型本体蠕变问题。这是因为结构比字符串更容易失配：如果两端程序的本体不能正确匹配，要让它们通信肯定很难，纠错难上加

难。最成功的 RPC 应用，如网络文件系统，都是那些在应用定义域上本来就只涉及很少量简单数据类型的应用。

支持 RPC 的常见理由是它比文本流方法允许“更丰富”的接口——也就是说，接口可以具有更复杂、更专用的数据类型本体。但是想想简洁原则吧！我们在第 4 章有过考察，接口的功能之一是充当阻隔点，防止模块的实现细节彼此泄漏。因此，支持 RPC 的主要理由同时恰恰证明了 RPC 增加了，而不是降低了程序的全局复杂度。

使用经典的 RPC 太容易用复杂晦涩的方式完成任务，而不是保持其简单。RPC 似乎鼓励生产规模庞大、结构复杂、过度行工的系统，加上令人糊涂的接口、居高不下的全局复杂度、严重的版本不兼容和可靠性问题——简直就是厚胶合层为非作歹的完美实例。

Window COM 和 DCOM 可能是这究竟有多糟的样板，但是这样的例子还有很多。苹果放弃了 OpenDoc，CORBA 和曾经大肆宣传的 Java RMI，随着人们使用经验的增长，都已经从 Unix 世界的视野中消失了。这完全可能是因为这些方法能够解决的问题还不如它们引发的问题多。

Andrew S. Tanenbaum 和 Robbert van Renesse 在《对远程过程调用范式的批评》(A Critique of the Remote Procedure Call Paradigm [Tanenbaum-VanRenesse]) 中对 RPC 的一般问题作了详细分析，这篇文章应该是对考虑将其作为基础架构的人们的当头棒喝。

所有这些问题都预示着使用 RPC 的为数不多的 Unix 项目会存在长期困难。在这些项目中，最著名的恐怕就是 GNOME 桌面项目了[14]。这些问题也使对外公开的 NFS 服务器特别容易受到安全性方面的攻击。

另一方面，Unix 传统强烈赞成使用透明、可显的接口。这就是 Unix 文化不断坚持文本协议 IPC 的动力。经常有人固执认为，相对二进制 RPC 而言，文本协议的解析开销是个性能问题——但是 RPC 接口往往产生更糟糕的

[14] GNOME 的主要竞争对手 KDE 开始时使用 CORBA，但在其 2.0 版本中将之放弃。从那以后，他们一直在寻求一种更轻量级的 IPC 方法。

延迟问题，原因是：（a）无法准确预估出一个指定调用会涉及多少数据的列集和散集，（b）RPC 模型往往鼓励程序员把网络交易视为无成本行为。即使在某个事务处理接口上只额外增加一个来回，往往也会增加足够多的网络延迟，完全抵消了解析或列集的开销。

即使文本流没有 RPC 效率高，但性能损失也是微小和线性的，最好通过硬件升级，而不是耗费开发时间或增加结构复杂度来解决这个问题。使用文本流可能造成的任何性能损失，都可以得到补偿，即有能力设计更简单的系统，更易于监控、建模和理解。

今天，通过 XML-RPC 和 SOAP 等协议，RPC 和 Unix 对文本流的坚持开始以一种有趣的方式融合到一起。这些协议，由于是文本化而且透明的，比它们所取代的丑陋、重量级的二进制序列格式更合乎 Unix 程序员的心意。尽管它们不能解决 Andrew S. Tanenbaum 和 Robbert van Renesse 所提出的更普遍的问题，但它们在某些方面综合了文本流和 RPC 方法的优点。

线程——恐吓或威胁

尽管 Unix 开发者早就已经习惯于通过多个协作进程进行计算，他们仍然没有使用线程（共享整个地址空间的进程）的自发传统。线程最近才从其它地方移植过来，而 Unix 程序员通常不喜欢线程这件事，决不仅仅只是意外或历史的偶然。

从复杂度控制的角度来看，相对拥有独立地址空间的轻量级进程，线程是个糟糕的替代；线程是那些进程生成昂贵、IPC 功能薄弱的操作系统所特有的概念。

从定义上看，尽管进程的子线程通常具有独立的局部变量栈，它们却共享同一全局内存。在这个共享地址空间管理竞争和临界区的任务相当困难，而且成为增加整体复杂度和滋生 bug 的温床。可以这样去做，但是随着锁定机制复杂度的增加，意外交互作用所造成的竞争和死锁机会也相应增加。

线程成为滋生 bug 温床源于它们太容易知道过多彼此的内部状态。与有

着独立地址空间、必须通过明确 IPC 进行通信的进程不同，线程没有自动封装。这样，基于线程的程序不仅产生普通的竞争问题，而且产生了新一类 bug：时序依赖，要重现这些问题都极其困难，遑论修复。

线程开发者已经觉察到这个问题。最近的线程实现和标准则体现了对提供线程本地存储的更多关注，其目的是限制共享全局地址空间所产生的问题。随着线程 API 朝这个方向发展，线程编程开始越来越像是对共享内存的有约束应用了。

> 线程常常阻碍了抽象。为了防止死锁，经常必须了解所使用的库是否使用和如何使用线程，以避免死锁问题。类似地，在程序库中使用线程还可能受到应用层使用线程的影响。
>
> —David Korn

雪上加霜的是，线程的性能成本削弱了其对常规进程分解的优势。尽管线程没有快速转换进程上下文的开销，但是锁定共享数据结构以防互相干涉的开销同样昂贵。

> X server 的执行速度能够达到数百万次/秒 (ops/second)，但不是基于线程实现的；它使用 poll/select 循环。创造多线程实现的种种努力都没有产生好结果。对于图形服务器这种对性能敏感的程序来说，锁定和解锁操作的成本太高了。
>
> —Jim Gettys

这个问题是根本性的，也一直是 Unix 内核的对称多处理设计的长期问题。由于资源锁定变得越来越琐细，锁定导致的延迟也迅速增加，足以超过只锁定更少的核心内存所获得的收益。

线程的最终难题在于，直到 2003 年年中，线程的标准仍然很薄弱而且规范不够明确。Unix 标准在理论上已经一致的程序库，如 POSIX 线程 (1003.1c)，仍然在不同的平台展示了惊人的行为差异，尤其是在信号、同其它 IPC 方法的交互和资源清理时间方面。Windows 和传统 MacOS 自带的线程模型和中断功能与 Unix 差别非常大，即使很简单的线程程序也常常需要相当大的精力才能移植。结果是根本不要指望线程程序可移植。

更多的讨论以及和事件驱动编程的鲜明对照请参考《为什么线程是个馊主意》(Why Threads Are a Bad Idea)[Ousterhout96]。

在设计层次上的进程划分

现在万事俱备，我们应该怎么办呢？

第一个要注意的是，临时文件、交互性更强的主/从进程关系、套接字、RPC 和其它一些双向 IPC 方法在某种程度上是等价的——它们都只不过是程序在生命期内交换数据的方法。我们通过使用套接字或共享内存这种复杂的方法所完成的任务，大多数都可以通过使用临时文件作为信箱和通知信号这种简单的方法来完成。差别很小，主要体现在程序如何建立通信、何时何地完成信息的列集和散集、可能产生何种缓冲问题，以及如何保证获取信息的原子性（也就是说，在何种程度上可以知道来自一边的单个发送行为会在另一边成为单个的接收事件）。

我们已经从 PostgreSQL 实例中看到，降低复杂度的有效方法是把程序划分成客户端/服务器对。PostgreSQL 的客户端和服务器的通信通过基于套接字的应用协议来实现，但是既便使用其它双向 IPC 方法，设计模式也并不会有多大改变。

在应用程序的多个实例必须管理共享资源访问的情况下，这种划分特别有效。一个简单的服务器进程可以管理所有的资源争用，或者协作的每个对等程序端都可以掌管某个关键资源。

客户端/服务器划分也有助于在多个主机上分布高时效要求的 (cycle-hungry) 应用程序。或者可以使它们适应互联网的分布计算（如 Freeciv）。我们将在第 11 章讨论相关的 CLI 服务器模式。

由于所有这些对等进程间的 IPC 技法在某种程度上都很像，所以我们应该主要评估它们引起的程序复杂度开销，以及给设计造成的不透明度。归根结底，这就是为什么 BSD 套接字会在 Unix IPC 中胜出，以及为什么 RPC 根本无法获得更多支持的原因。

线程有着根本性不同。线程支持的并非不同程序之间的通讯，而是单个程序的一个实例内的某种分时形式。线程并不是把大程序分解成行为简单的小程序的方法，实际上是一种性能调整（performance hack）问题。但线程不仅具有此类问题的通病，而且还有自身的特殊症结。

因此，当我们寻找方法，把大程序分解成更简单的协作进程时，在进程内使用线程应该是最后一招而不是第一招。通常，你可以发现避免使用线程是可能的。如果能够使用有限的共享内存和信号量、使用 **SIGIO** 的异步 I/O，或 *poll*(2)/*select*(2)，而不是使用线程，就这样做吧，保持简洁，在本章所列的技法中优先使用位置更前、复杂度更低的方法。

把线程、远程过程调用接口和重量级的面向对象设计结合使用特别危险。如果使用的非常谨慎和优雅，这些技术中的任何一个技术可能都非常有价值——但是如果你被邀请加入要使用这三者的项目，逃之夭夭并不丢面子。

前面我们已经讲过，真实世界里的编程其实就是管理复杂度的问题。能够管理复杂度的工具都是好东西。但当这些工具的作用不是控制而是增加复杂度的时候，最好扔掉，从零开始。永远不要忘记这一点，它是 Unix 智慧的重要组成部分。

微型语言：寻找歌唱的乐符

> 从好的符号体味出的巧妙和启发，就算身边的老师也不过如此。
>
> The World of Mathematics《1956》
>
> —Bertrand Russell

对软件错误模式进行的大量研究得出的一个最一致的结论是，程序员每百行代码出错率和所使用的编程语言在很大程度上无关。[1]更高级的语言可以用更少的行数完成更多的任务，也意味着更少的 bug。

Unix 有个长期传统，即包容小型的、为专门应用领域特制、大量减少程序行数的语言。专门领域语言的实例包括无数 Unix 排版语言（troff，eqn，tbl，pic，grap）、shell 实用程序（awk，sea，dc，bc）和软件开发工具（make，yacc，lex）。专门领域语言和更灵活应用程序的运行控制文件（sendmail，BIND，X）之间、数据文件格式之间、脚本语言之间的界限都很模糊（我们将在第 14 章予以分析）。

从历史上讲，这种类型的专门领域语言在 Unix 世界中称为“小语言”或“微型语言”，原因是相对通用语言而言的，早期这类语言的实例都较小，复杂度较低（这三个术语均指通常情况）。但如果应用领域很复杂（体现在具有

---

[1] Les Hatton 在其写作的 Software Failure（软件故障）一书中用 email 形式报告其分析：“如果用可执行的行数进行密度测量，则不同语言产生的缺陷密度的差异大概只有不同程序员所产生的缺陷密度差异的十分之一。”

很多不同原语操作或涉及繁杂数据结构的处理），其采用的应用语言将不得不比一些通用语言更复杂。我们将保留传统术语“微型语言”以强调：明智的方法通常是保持这些设计尽可能小、尽可能简单。

专门领域的小语言是一种非常强大的设计理念，可以使我们为手头的任务定义自己更高级的语言，以详细规定恰当的方法、规则和算法；比起采用更低级硬编码的设计而言，这降低了全局复杂度。至少有三种方式可以做好微型语言的设计，两种好的，一种危险的。

第一个正确方法是，预先认识到可以使用微型语言设计把编程问题的规格说明提升一个层次；跟通用语言所能支持的表示法相比，这种表示法更紧凑、更具表达力。与代码生成和数据驱动编程一样，微型语言允许我们切实利用“软件的缺陷率和使用的语言在很大程度上无关”这一事实：更具表达力的语言意味着程序更短，bug 更少。

第二个正确方法是，注意到规格说明文件格式越来越类似微型话言——也就是说，结构趋向复杂，控制的应用程序中包含行为。规格说明是否试图描述控制流和数据部署？如果是，也许是把规格说明语言中的控制流从隐式提升到显式的时候了。

错误方法是通过扩展变成微型语言：每次增加一个补丁或者一个仓促而就的特性。在这条道路上，规格说明文件不断滋生出更加隐蔽的控制流向和更加杂乱的专用结构，在不知不觉间变成了一种特别语言。一些传说中的噩梦就是这样产生的：每一位 Unix 大师都会（战战兢兢地）想起的例子是 sendmaill 邮件传输程序用到 sendmail.cf 配置文件。

悲哀的是，大多数人面对第一个微型语言时就采取了错误的方法，并且事后才意识到这个微型语言已经变得多么混乱。接下来的问题是：如何清理？有时候，答案可能意味着要重新构思整个应用程序的设计。语言特性蠕变（Language-by-feature creep）的另外一个臭名昭著的例子是 TECO 编辑器，这个编辑器首先发展了宏，然后随着程序员想要用其包装越来越复杂的编辑程序，才增加了循环和条件。由此产生的丑陋最终只能通过基于意向性语言重新设计整个编辑器才得以解决；这就是 Emacs Lisp（以下将分析）的形成

过程。

所有十分复杂的规格说明文件都希望成为微型语言。因此，事实经常是，防御设计不良微型语言的唯一方法是知道如何设计一个好的微型语言。这并不需要巨大的跨步或涉猎许多形式语言理论；有了现代工具，只要学习一些相对简单的技巧，并将几个优秀范例牢记在心就足够了。

本章我们将分析 Unix 通常支持的各种微型语言，并尽量确定每种微型语言对何种问题可提供有效解决方案。本章并非要穷举所有的 Unix 语言类别，而是展示围绕微型语言建构应用程序所涉及的设计原则。我们将在第 14 章更详细讨论通用编程语言。

首先，我们必须进行一个小小的分类，以便以后区分彼此。

理解语言分类法

图 8.1 所示的所有语言都在本章或本书其余章节的案例分析中有描述。右侧的通用型解释器的说明，请参见第 14 章。

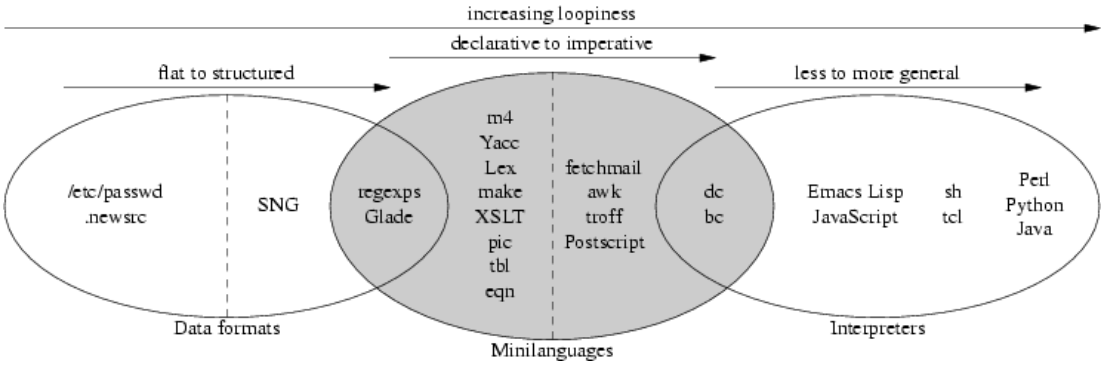


图 8-1: 语言分类

我们在第 5 章分析了 Unix 对数据文件的约定。这些文件的复杂度按谱系排列。其中，底端是简单关联名称和属性的文件；/etc/passwd 和.newsrc 是好例子。往上去，我们看到对数据结构进行列集或序列化的格式；（对等地，）PNG 和 SNG 格式是这种类型的好例子。

如果结构化的数据文件格式不仅表达了结构，而且在某些解释性上下文（也就是数据文件以外的存储）下表达执行行为时，这种格式开始接近于一种微型语言。XML 标记语言往往跨入界限；我们在此要举的例子是 Glade，一个用于构建 GUI 界面的代码生成器。那种既可供人类读写、又可用于生成代码的格式绝对属于微型语言。yacc 和 lex 是其中的经典例子。我们将在第 9 章讨论 glade、yacc 和 lex。

Unix 的宏处理器 m4 是另外一种非常简单的声明性微型语言（也就是说，在这种语言中，程序被表示成一套预期的关系或约束集，而不是明确的行为）。它经常用在其它微型语言的预处理阶段。

Unix 的 makefile 是为了自动化编译过程而设计的，表达了源文件和派生文件[2]之间的依赖关系以及从各个源文件生成派生文件所需的命令。运行 make 时，它用说明来遍历隐式依赖关系树，完成最少的必要工作以更新编译。与 yacc 和 lex 说明一样，makefile 是一种声明性微型语言：它们以约束条件来说明在解释性上下文（在这种情况下指源文件和派生文件共属的文件系统部分）中要完成的行为。我们将在第 15 章分析 makefile。

用于描述 XML 转换的语言 XSLT 处于声明性语言复杂度的顶端。这种语言复杂到通常没有人认为它是一种微型语言，但它包含我们以下将更详细分析的微型语言所具有的一些重要特性。

微型语言的范畴从声明性（具有隐式操作）发展到命令性（具有显式操作）。*fetchmail*(1) 的运行控制语法可视为一种很弱的命令性语言或一种暗含控制流的声明性语言。troff 和 PostScript 排版语言是包含很多专门领域知识的命令性语言。

有些针对专门任务的命令性微型语言几乎快成为通用解释器。当它们明确成为完备图灵机时，它们就是解释器——也就是说，可以进行条件和循环（或

[2] 对技术背景较弱的读者来说：C 程序的编译形式通过编译和链接从 C 源码派生而来，troff 文件的 PostScript 版本从 troff 源程序派生；从后者生成前者的命令是 troff 调用。还有其他类型的派生；makefile 几乎可以表示全部派生关系。

递归）[3]操作，并且具有用于控制结构的特性。相反，有些语言只是在某些情况下是完备图灵机——虽然它们具有能够用于执行控制结构的特性，但只是实际设计目的的副产物。

我们在第 7 章分析的 *bc*(1) 和 *dc*(1) 解释器是明显的完备图灵机语言的好例子，它们都是专用的命令性微型语言。

当我们进入到 Emacs Lisp 和 JavaScript 这类用于特定上下文下运行的完全编程语言时，便来到通用解释器的领域。我们在以后讨论嵌入式脚本语言时将更详细讨论它。

解释器的范畴也是通用性不断增加的范畴；反过来讲，更加通用的解释器对其运行上下文的假设很少。随着通用性增加，通常存在更加丰富的数据类型分类。Shell 和 Tcl 都具有相对简单的数据类型分类；Perl，Python 和 Java 则更加复杂。我们将在第 14 章分析这些通用语言。

应用微型语言

设计微型语言面临两个截然不同的挑战。一个挑战是在工具包中已经存在方便好用的微型语言时，认识到何时可以直接应用它们。另一个挑战是知道应该在什么时候为应用程序设计自定的微型语言。为了帮助读者培养这两个方面的设计意识，本章的一半篇章都是案例分析。

案例分析：sng

我们在第 6 章分析了在 PNG 和逐位等价的可编辑纯文本表达之间进行转换的 *sng*(1)。SNG 数据文件格式在此处值得再次对照研究，这是因为它不完

[3] 任何完备图灵机语言在理论上都可用于通用编程，并且理论上和任何其它类型的完备图灵机语言一样有效。实际上，有些此类语言在其狭窄的特定问题域以外使用时可能令人非常痛苦。

全是一种领域专用语言。它描述了数据格式，但数据和任何默认的动作序列都不相关。

然而，SNG 确实拥有专门领域微型语言所拥有，但 PNG 之类的二进制结构数据格式却没有的一个重要特性——透明性。结构化的数据文件使得那些编辑、转换和生成工具无需知道任何其他方的设计假定、只需通过微型语言介质本身，就可以彼此协作。和专门领域微型语言一样，SNG 如此设计的额外好处是便于人眼分析，便于通用工具编辑。

案例分析：正则表达式

有一类规格说明频繁出现在 Unix 工具和脚本语言中：正则表达式（简写为“regexp”）。我们在此将其视为描述文本模型的一种声明性微型语言；它经常内嵌在其它微型语言中。Regexp 无处不在，几乎没有人认为它是微型语言，但它们取代了那种执行不同（而且不可互用）搜索功能的大堆代码。

以上介绍跳过了 POSIX 扩展、反向引用和国际化特性等细节。更详细的介绍请参见 Mastering Regular Expressions（精通正则表达式）[Friedl]。

正则表达式描述了匹配或不匹配字符串的模式。最简单的正则表达式工具是 *grep*(1)，它用在每行输入和输出之间匹配指定 regexp。正则表达式符号归纳在表 8.1。

表 8.1: 正则表达式实例

Regexp	匹配
”x.y”	x 后面接任何一个字符再接 y。
”x\.y”	x 后面接. 再接 y。
”xz?y”	x 后面最多接一个 z 再接 y: 这样，可以是”xy” 或”xzy”，但不是”xz” 或”xdy”。
”xz*y”	x 后面接任意数量的 z 再接 y: 这样，可以是”xy” 或”xzy” 或”xzzy” 等，但不是”xz” 或”xdy”。
”xz+y”	x 后面接一个或多个 z 再接 y: 这样，可以是”xzy” 或”xzzy” 等，但不是”xy”、”xz” 或”xdy”。
”s[xyz]t”	s 后面接 x、y 或 z 中的任何一个字符再接 t: 这样，可以是”sxt”、”syt” 或”szt”，但不是”st” 或”sat”。
”a[x0-9]b”	a 后面接 x 或 0-9 之间的任何一个字符再接 b: 这样，可以是”axb” 或”a0b” 或”a4b”，但不是”ab” 或”aab”。
”s[^xyz]t”	s 后面接非 x、y、z 的任何一个字符再接 t: 这样，可以是”sdt” 或”set”，但不是”sxt”、”syt” 或”szt”。
”s[^x0-9]t”	s 后面接非 x 或非 0-9 之间的任何一个字符再接 t: 这样，可以是”slt” 或”smt”，但不是”sxt”、”s0t” 或”s4t”。
”^x”	x 在字符串的开头[a]: 这样，可以是”xzy” 或”xzzy”，但不是”yzy” 或”yxy”。
”x$”	x 在字符串的末尾: 这样，可以是”yzx” 或”yx”，但不是”yxz” 或”zxy”。

[a] 不是应该是行首和行尾?

regexp 符号还存在一些小变种:

1. **Glob 表达式**。这是早期 Unix shell 用于文件名匹配的有限通配符约定集。只有三个通配符: ”*” 匹配任何序列的字符（与其它变种中的.* 类似）; ”? ” 匹配任何一个字符（与其它变种中的. 类似）: [...] 和其它

变种中一样，匹配一类字符。有些 shell(csh，bash，zsh) 后来也允许用 {} 表示。这样，`x{a ，b}c` 与 xac 或 xbc 匹配，但不匹配 xc。有些 shell 进一步朝扩展正则表达式的方向扩展了 glob。

2. **基本正则表达式**。最初的 *grep*(1) 实用程序使用这些符号来从文件中提取与指定 regexp 匹配的文本行。行编辑器 *ed*(1) 和字符流编辑器 *sed*(1) 也使用这些符号。Unix 老手认为这些符号是基本的 regexp，或“最普通”(vanilla) 口味的 regexp；首先接触更现代工具的人往往采用以下描述的扩展形式。

3. **扩展的正则表达式**。这是从 grep 公用程序扩展出的 *egrep*(1)，实用程序使用这些符号来从文件中提取与指定 regexp 匹配的文本行。Lex 和 Emacs 编辑器中的正则表达式非常接近 egrep 风格。

4. **Perl 正则表达式**。这是 Perl 和 Python 的 regexp 功能所接受的符号。它们比 egrep 风格的符号更强大。

看了这些启发性例子之后，表 8.2 总结了正则表达式的标准通配符。注意：该表不包括 glob 变种，因此“全部”代表的只是基本、扩展/Emacs 和 Perl/Python 这三种变种[4]。

---

[4] 正则表达式的 POSIX 标准引进了一些符号域，如 [[:lower;;]] 和 [[:digit:]] 等，一些专用工具还有此处未包括的通配符，但这些已经足够解析大多数 regexp。

表 8.2: 对正则表达式的介绍

通配符	支持语言	匹配
\	全部	转义后面的字符。切换后续符号是否作为通配符。根据程序的不同，后续的字母或数字可用各种不同的方式解释。
.	全部	任何字符
^	全部	行首
$	全部	行尾
[...]	全部	在括号内的任何字符。
[^ ...]	全部	除开括号内字符之外的任何字符
*	全部	次数不定的重复前一元素
?	egrep/Emacs Perl/Python	0 次或 1 次重复前一元素
+	egrep/Emacs Perl/Python	1 次或多次重复前一元素
{n}	egrep Perl/Python Emacs 中为 \{ n \}	只 n 次重复前一元素，一些较老的 regexp 引擎不支持。
{n, }	egrep Perl/Python Emacs 中为 \{ n , \}	n 次或 n 次以上重复前一元素，一些较老的 regexp 引擎不支持。
{m,n}	egrep Perl/Python Emacs 中为 \{ m , n \}	最少重复 m 次、最多重复 n 次。一些较老的 regexp 引擎不支持。
|	egrep Perl/Python Emacs 中为 |	接受左边或者右边的元素。通常用于一些模式分类分隔符形式。
(...)	Perl/Python 老一点的版本为 \(... \)	在较新的 regexp 引擎中，如 Perl 和 Python 把这种模式作为组。较老的 regexp 引擎，如 Emacs 和 grep 中，要求 \(... \) 。

支持 regexp 的新语言所采用的设计实践已经固定使用 Perl/Python 的变种。它比其它变种更透明，主要是因为非字母数字字符之前的反斜杠 \ 始终表示该字符的字面值，因此对如何引用 regexp 的元素产生的混淆较少一些。

正则表达式是一个微型语言能够多么简练的极端例子。简单的正则表达式表达了识别行为，不这样做的话，这些行为必须用成百行繁琐易错的代码来实现。

案例分析：Glade

Glade 是 X 的开源 GTK 工具包库所用的界面创建程序[5]。Glade 通过在界面面板上交互地选择、放置和修改窗体部件来生成 GUI 界面。GUI 编辑器生成描述该界面的 XML 文件；反过来，这又可成为几种代码生成器的输入，最终成为界面的 C、C++、Python 或 Perl 代码。生成的代码将调用编写的函数向界面提供行为。

Glade 用于描述 GUI 的 XML 格式是简单的专门领域语言的好例子。参见例 8.1 所举的 Glade 格式的“Hello，world!”GUI。

例 8.1 Glade ”Hello，World”

```
<?xml version="1.0"?>
<GTK-Interface>

<widget>
  <class>GtkWindow</class>
  <name>HelloWindow</name>
  <border_width>5</border_width>
```

[5] 对非 Unix 程序员来说，X 工具包是一种图形库，可向使用它的程序提供 GUI 构件（如标签、按钮、下拉菜单等）。其它绝大多数图形操作系统只提供一个工具包供大家使用。Unix 和 X 都提供多个工具包；这也是我们在第 1 章作为设计目标提出来的机制和策略分开的一个组成部分。GTK 和 Qt 是最流行的两个开源 X 工具包。

```
  <Signal>
    <name>destroy</name>
    <handler>gtk_main_quit</handler>
  </Signal>
  <title>Hello</title>
  <type>GTK_WINDOW_TOPLEVEL</type>
  <position>GTK_WIN_POS_NONE</position>
  <allow_shrink>True</allow_shrink>
  <allow_grow>True</allow_grow>
  <auto_shrink>False</auto_shrink>

  <widget>
    <class>GtkButton</class>
    <name>Hello World</name>
    <can_focus>True</can_focus>
    <Signal>
      <name>clicked</name>
      <handler>gtk_widget_destroy</handler>
      <object>HelloWindow</object>
    </Signal>
    <label>Hello World</label>
  </widget>
</widget>

</GTK-Interface>
```

Glade 格式的有效规格说明包含了响应用户行为的 GUI 指令表。一方面，Glade 的 GUI 把这些规格说明作为结构化的数据文件处理：另一方面，Glade 的代码生成器则用这些规格说明来编写 GUI 的实现码。有些语言（包括 Python）有运行时程序库，可跳过代码生成步骤，简单地在运行时直接从

XML 规格说明（解释 Glade 标记，而不是把这些标记编译到宿主语言中）实例化 GUI。这样，我们可以选择牺牲空间效率来提高启动速度，或反之。

跳过 XML 的冗长，Glade 标记实际是一种相当简单的语言。它只做两件事：说明 GUI 窗体构件层次、关联窗体构件和属性。实际上，我们并不需要懂很多 Glade 工作原理才能读取以上规格说明。事实上，如果有过 GUI 工具包的编程经验，读一读以上例子立刻会使我们很好地想象出 Glade 如何处理规格说明。（认为这段具体的规格说明能在窗口框架上生成一个按钮的读者请举手。）

这种透明性和简单性是一个微型语言设计良好的标志。符号和域对象之间的映射非常清晰。对象之间的关系表达得很直接，舍弃了必须经过思考才能理解的间接表达方式如名字引用。

最终对微型语言的功能测试就是这么简单：不读手册可以编写吗？在相当多的情形中，Glade 的回答是肯定的。例如，如果知道用于描述 GTK 窗体位置信息的 C 常量，就可以立即用`GTK_WIN_POS_NONE` 值改变与这个 GUI 有关的位置信息。

使用 Glade 的优势应该很明显。它专门负责代码生成，所以我们就不必关心代码生成了。这就减少了一个我们必须手工编程的例行任务，从而减少了因此产生的 bug。

更多的信息，包括源码、文档和实例应用的一些链接，都可从Glade 的项目主页获得。Glade 已经移植到了 Windows 上。

案例分析：m4

m4(1) 宏处理程序解释描述文本转换的声明性微型语言。一个 m4 程序就是一套宏命令集，规定了把文本串扩展成其它字符串的方式。在输入文本中应用 m4 程序中的声明就实现了宏扩展并生成输出文本。（C 预处理器也为 C 编译器执行类似的服务，但风格截然不同。）

例 8.2 演示了一个 m4 宏命令，把输入中的每一个”OS” 字符串都转换

成”Operating System” 输出。这是一个很小的例子：m4 支持带参数的宏命令，可胜任更多的任务，而不仅仅是把一个固定的字符串转换成另一个固定的字符串。在 shell 提示中键入`info m4` 可显示这个语言的在线文档。

例 8.2 m4 宏命令实例

```
define(‘OS’, ‘operating system’)
```

m4 宏语言支持条件和递归。两者结合可实现循环，设计便是如此：m4 是一种有意的完备图灵机。但实践中把 m4 当作通用语言使用则非常困难。

通常，对于缺乏内置命名过程标记法和文件包含功能的微型语言，可以使用 m4 宏处理程序作为预处理器。这是扩展基础语言语法的简单方法，于是，与 m4 相结合就可以支持上述功能。

m4 非常出名的一个应用是避免（或至少有效隐藏）本章一开始作为反例提出的第三种设计微型语言的方法。大多数系统管理员现在都可以应用 sendmail 配置提供的 m4 宏语句包生成自己的 sendmail.cf 配置文件。宏使用特性名（或名称/值对）来生成 sendmail 配置语言所用的（更加丑陋的）对应字符串。

然而，请慎重使用 m4。Unix 经验教导微型语言的设计者要谨防宏扩展，[6]具体原因将在本章以下部分讨论。

案例分析：XSLT

和 m4 宏一样，XSLT 也是描述文本流变换的一种语言。但它的任务并不仅仅是简单的宏替换；它还可以描述 XML 数据的变换，包括查询和报表生成。它是用于编写 XML 样式表的语言。关于 XSLT 的实际应用，参见第

6　宏扩展的英文应该拼写成”macro expansion” 还是”macroexpansion” 尚有争议。后者主要在 Lisp 程序员中使用。

18 章 XML 文档处理部分的说明。XSLT 有万维网联盟（World Wide Web Consortium，W3C）标准的描述，并且拥有数个开源实现。

XSLT 和 m4 都是纯声明性和完备图灵的语言，但 XSLT 只支持递归，不支持循环。XSLT 相当复杂，无疑是本章实例分析中最难掌握的语言——而且可能是本书所涉及的最难的语言。[7]

尽管非常复杂，但 XSLT 的确是一种微型语言。它具备微型语言最重要的（尽管不是普遍的）特征：

- 有限的类型分类，（特别是）没有记录结构或数组等类似结构。
- 对外部的有限接口。XSLT 处理器的设计目的是在标准输入和标准输出之间进行过滤，具备有限的读/写文件能力，不能打开套接字或运行子命令。

例 8.3 的程序将 XML 文档中每个元素的每个属性都变换成被该元素直接括入的新标记对，原属性值作为其内容。

在这里简单了解 XSLT，部分原因是举例说明“声明性”并不一定意味着“简单”或“薄弱”，更主要是因为如果必须处理 XML 文档，则总有一天必须面对 XSLT 的挑战。

XSLT: Mastering XML Transformations（XSLT：精通 XML 转换）[Tidwell] 很好地介绍了这种语言。也可在网上找到带实例的简明教程。[8]

例 8.3 XSLT 程序实例

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

[7] 然而，还不清楚 XSLT 是否能变得更简单一些、同时仍能完成任务，所以我们无法把它定性为不好的设计。

[8] 如 XSL Concepts and Practical Use（XSL 的概念和实际应用），参见 <http://nwalsh.com/docs/tutorials/xsl/xsl/slides.html>。

```
                version="1.0">
  <xsl:output method="xml"/>
  <xsl:template match="*">
    <xsl:element name="{name()}">
      <xsl:for-each select="@*">
        <xsl:element name="{name()}">
          <xsl:value-of select="."/>
        </xsl:element>
      </xsl:for-each>
      <xsl:apply-templates select="*|text()"/>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

案例分析：The Documenterls Workbench Tools

正如我们在第 2 章所说，*troff*(1) 排版格式器是 Unix 最早的王牌应用程序。troff 是格式工具套件（总称为 Documenter's Workbench 或 DWB，文档编制工作台）的核心，其中各个组成部分都是不同类型的专门领域微型语言。大多数是 troff 标记语言的预处理或后处理器。开源 Unix 上有个称为 *groff*(1) 的 DWB 增强实现版，它来自于自由软件基金会。

我们将在第 18 章更详细介绍 troff；现在，我们只要注意到 troff 是命令性微型语言的好例子就足够了。这个命令性微型语言接近于一种正式的解释器（具有条件和递归，但没有循环；偶尔是图灵完备的）。

后处理器（用 DWB 的术语来讲就是“驱动器，driver”）通常对 troff 用户是不可见的。1970 年 Unix 开发组就使用最早的 troff 向特殊排字机输送代码；1970 年代后期，这些都被整理成设备无关的微型语言，用于在页面上插入文本和简单图形。后处理器把这种语言（称为”ditroff”，代表”device-independent troff”（设备无关的 troff））转换为现代图像打印机能实际接

受的格式——其中最重要的（在当代也是默认的）是 PostScript。

预处理器更有意思，原因是它们实际上增强了 troff 语言的能力。常用的三个语言是：用于制表的 *tbl*(1)、用于排版数学公式的 *eqn*(1) 和用于绘制图表的 *pic*(1)。还有不太常用但仍然存在的语言，如制图用的 *grn*(1) 及排版参考书目的 *refer*(1) 和 *bib*(1)。所有这些语言等价的开源实现都随 groff 一同发布。*grap*(1) 预处理器提供了一个全面的测绘图功能；还有一个独立于 groff 的开源实现。

其它一些预处理器没有开源实现，也不再普遍使用。其中最著名的是制图用的 *ideal*(1)。较新的 *chem*(1) 可绘制化学结构式；它还是贝尔实验室 netlib 代码的组成部分[9]。

每个预处理器都是一个小程序，接受微型语言并将其编译成 troff 请求。通过查找独特的起始和结束请求，每个小程序识别应该解释的标记（tbl 查找`.TS/.TE`，pic 查找`.PS/.PE` 等），其余的标记则不加修改地传递，这样，大多数预处理器通常都能互不干涉地以任何一种次序运行。也有例外：特别是 chem 和 grap 都产生 pic 命令，因此必须先于该命令出现在管道线中。

```
cat thesis.ms | chem | tbl | refer | grap | pic | eqn \
                                          | groff -Tps >thesis.ps
```

上述只是为演示 DWB 处理管道线而捏造的实例。假想一篇论文，它包含化学公式、数学公式、表格、参考书目、标绘图和图表。（*cat*(1) 命令只是把输入或文件参数拷贝到输出；在此使用这个命令是为了强调操作的顺序。）在实践中，现代 troff 实现往往至少支持调用 *tbl*(1)、*eqn*(1) 和 *pic*(1) 的命令行选项，因此就无须编写以上这样复杂的管道系统。即使必要，这些编译命令通常也只需编写一次，然后隐藏在 makefile 或 shell 脚本包装器中以备重复使用。

[9] http://www.netlib.org/

DWB 的文档标记在某些方面已经过时，但是这些预处理器所处理的问题域仍然表明了微型语言模型的能力——所见即所得的字处理器想具备同样的能力可能非常困难。直到 2003 年，在某些方面，基于 XML 的现代文档标记语言和工具链仍在试图赶上 DWB 早在 1979 年就已经具备的能力。我们将在第 18 章更详细讨论这些问题。

至此，大家应该很熟悉使 DWB 如此强大的设计原因了；所有的工具都共享一个通用文档的文本流表达，格式化系统划分成多个独立的组成部分，可分别调试和改进。管道架构支持插入新的、实验性的预处理器和后处理器，而且不会干扰原有处理器。这是个模块化和可扩展的结构。

DWB 体系作为一个整体教导我们如何把多个专用微型语言组装成一个协作系统。一个预处理器可搭建在另一个预处理器上。事实上，DWB 的工具是管道、过滤器和微型语言结合工具的早期模型范例，影响了 Unix 后面的很多设计。有效的微型语言应该如何设计，可以从各个预处理器的设计中得到很多经验。

其中有个教训是负面的。有时候，在微型语言中用户编写的说明可能无法干净地处理手工插入的低级 troff 标记。这可能会产生交互作用和难以诊断的 bug，原因是管道生成的 troff 是看不见的——即使能看见也不可读。这和把 C 与内嵌汇编器片断混合起来的代码中产生 bug 的情形相似。如果可能的话，把语言层更加完整地分离出来可能会更好一些。微型语言的设计者应该注意到这点。

所有的预处理器语言（尽管 troff 标记本身不是）都具有相对干净、类似 shell 的语法，遵循了我们在第 5 章针对数据文件格式的设计描述过的很多约定。也有一些令人尴尬的例外：特别是，*tbl*(1) 默认使用 tab 键作为表格栏之间的字段分隔符，复制了 *make*(1) 的设计中一个声名狼藉的补丁，而且，当编辑器或其它工具悄悄改变空白的组成时便会引发恼人的 bug，

尽管 troff 本身是一种专用的命令性语言，贯穿至少三个 DWB 微型语言的一个主题却是声明性语义：根据约束条件安排布局。这也是现代 GUI 工具包展现的一个理念——也就是说，对图形对象不用像素坐标表示，真正要做的

是说明它们之间的空间关系（A 在 B 上方，B 在 C 的左方），然后让软件根据这些约束条件计算出 A、B 和 C 的最佳布局。

pic(1) 程序就用这种方法来安排图形的各个部分。例 8.1 所示的语言分类图就是用例 8.4[10]所示的 pic 源码，调用我们在第 7 章作为实例分析的 pic2graph 得出的。

例 8.4 语言分类图—pic 源码

```
# Minilanguage taxonomy
#
# Base ellipses
define smallellipse {ellipse width 3.0 height 1.5}
M: ellipse width 3.0 height 1.8 fill 0.2
line from M.n to M.s dashed
D: smallellipse() with .e at M.w + (0.8, 0)
line from D.n to D.s dashed
I: smallellipse() with .w at M.e - (0.8, 0)
#
# Captions
"" "Data formats" at D.s
"" "Minilanguages" at M.s
"" "Interpreters" at I.s
#
# Heads
arrow from D.w + (0.4, 0.8) to D.e + (-0.4, 0.8)
"flat to structured" "" at last arrow.c
arrow from M.w + (0.4, 1.0) to M.e + (-0.4, 1.0)
"declarative to imperative" "" at last arrow.c
arrow from I.w + (0.4, 0.8) to I.e + (-0.4, 0.8)
```

[10] 描述 *pic*(1) 的 Unix 书把例图作为编码实例涵盖进来是相当传统的。

```
”less to more general” ”” at last arrow.c
#
# The arrow of loopiness
arrow from D.w + (0, 1.2) to I.e + (0, 1.2)
”increasing loopiness” ”” at last arrow.c
#
# Flat data files
”/etc/passwd” ”.newsrc” at 0.5 between D.c and D.w
# Structured data files
”SNG” at 0.5 between D.c and M.w
# Datafile/minilanguage borderline cases
”regexps” ”Glade” at 0.5 between M.w and D.e
# Declarative minilanguages
”m4” ”Yacc” ”Lex” ”make” ”XSLT” ”pic” ”tbl” ”eqn” \
            at 0.5 between M.c and D.e
# Imperative minilanguages
”fetchmail” ”awk” ”troff” ”Postscript” at 0.5 between M.c and I.w
# Minilanguage/interpreter borderline cases
”dc” ”bc” at 0.5 between I.w and M.e
# Interpreters
”Emacs Lisp” ”JavaScript” at 0.25 between M.e and I.e
”sh” ”tcl” at 0.55 between M.e and I.e
”Perl” ”Python” ”Java” at 0.8 between M.e and I.e
```

这是一段非常典型的 Unix 微型语言设计，而且即使从纯语法角度看，这段设计也有一些地方值得关注。注意这段程序看起来多么像 shell 程序：**#** 引导注释，语法很明显地面向标记 (token-oriented)，并采用尽可能简单的字符串约定。*pic*(1) 的设计者知道，除非有强烈、特殊的理由，否则 Unix 程序员所期望的微型语言的语法看起来就是这个样子。最小立异原则在这儿得到了充分运用。

不用费什么力就可以看出，这段代码的第一行是一个宏定义；对 smallellipse() 的后续引用封装了图表的重复设计元素。也无需多仔细研究就可以推导出，box invis 说明了一个没有边框的方框，实际上只是一个供文本输入的框架。arrow 命令同样很明显。

把这些作为线索看一看实际的图表，其余语法片断的意义（如 M.s 的位置引用 last arrow 或 at 0.25 between M.e and I.e 之类的限制条件或某个位置增加向量偏移等）都会立刻清楚了。和 Glade 标记及 m4 一样，无需参考手册，这样的例子就可以教导我们很多语言方面的东西（不幸的是，紧凑的 *troff*(1) 标记却没有这样的特性）。

pic(1) 的例子反映了微型语言一个常见的设计主题，我们发现这个设计主题也反映在 Glade 中——使用微型语言解释器来封装某些形式的约束条件推理，然后将其转化为行为。实际上，我们可以选择把 *pic*(1) 作为命令性语言而不是声明性语言来看待；这个语言具有两者的元素，争论无济于事。

宏与基于约束条件安排布局的结合使 *pic*(1) 具备了 SVG 之类基于向量的现代标记语言无法实现的表达图表的能力。因此，幸运的是，DWB 的一个影响就是 *pic*(1) 很容易脱离 DWB 环境使用。我们在第 7 章作为实例分析的 pic2graph 脚本就是这么一种特别的方法，使用 *groff*(1) 这个改进的 PostScript 能力作为现代位图格式的中间步骤来完成这一任务。

更干净的方法是使用 *pic2plot*(1) 实用程序，它随 GNU plotutils 包一起发布，并利用了 GNU *pic*(1) 代码内部的模块化特性。整个代码分成分析前端和生成 troff 标记的后端，前端和后端通过绘图原语层进行通讯。由于这个设计遵循了模块原则，*pic2plot*(1) 的执行者可把 GNU pic 分析进程分离出来，并用现代绘图库重新实现绘图原语。然而，这种方法的缺点在于输出文本是由 pic2plot 内置字体生成的，与 troff 字体不匹配。

案例分析：fetchmail 的运行控制语法

参见例 8.5。

例 8.5 fetchmailrc 的假想例子

```
# Poll this site first each cycle.
poll pop.provider.net proto pop3
    user ”jsmith” with pass ”secret1” is ”smith” here
    user jones with pass ”secret2” is ”jjones” here with options keep

# Poll this site second, unless Lord Voldemort zaps us first.
poll billywig.hogwarts.com with proto imap:
    user harry_potter with pass ”floo” is harry_potter here

# Poll this site third in the cycle.
# Password will be fetched from ~/.netrc
poll mailhost.net with proto imap:
    user esr is esr here
```

这个运行控制文件可看作命令性微型语言。存在一个隐含的执行流：循环执行查询命令列表（在每个循环结束时休息一会儿），对每个站点依次从其相关用户收集邮件。这个文件不通用；它所能完成的任务就是排队查询行为。

和 *pic*(1) 一样，我们可以把这个微型语言视为一种声明性语言，也可以把它视为一个很弱的命令性语言，并不断讨论其中的区别。一方面，它既没有条件，也没有递归或循环，事实上，它根本没有明确的控制结构。另一方面，它确实描述了行为，而不仅仅描述关系，这又与 Glade GUI 说明这样的纯声明性语法有所区别。

复杂程序的运行控制微型语言经常横跨这个界限。我们强调这个事实是因为，如果问题域允许，命令性微型语言中没有明确的控制结构将是一种巨大的简化。

.fetchmailrc 语法的一个显著特性是使用可选的纯修饰关键字，支持这些关键字仅仅是为了使说明语言读起来更像英语。例中的”with” 关键字和出现一次的”option” 实际上并不必要，但它们有助于使声明语言更易懂。

这类东西的传统术语是**语法糖**（syntactic sugar）；与之联系在一起的一段格言是非常著名的俏皮话："syntactic sugar causes cancer of the semicolon"（语法糖导致分号癌）[11]。事实上，必须谨慎使用语法糖，以免造成的晦涩多于帮助。

我们将在第 9 章看一看，数据驱动编程是怎样帮助通过 GUI 编辑 fetchmail 运行控制文件优雅地解决问题的。

案例分析：awk

awk 微型语言是老式的 Unix 工具，原来大量用于 shell 脚本。和 m4 一样，它的设计目的是为了编写小巧但有表达力的程序，从而把文本输入变换为文本输出。所有的 Unix 发布均包含它的某一版本，其中有几个是开源版本；Unix shell 提示下键入 **info gawk** 命令可以查看在线文档。

awk 程序包括模式/行为对。每个模式都是正则表达式，我们将在第 9 章具体讨论这个概念。awk 程序运行时一行一行过滤输入文件。每一行都顺序经过模式/行为对检查。如果模式和行匹配，则执行相关的行为。

每个行为都用类似 C 子集的语言写成，具有变量、条件、循环和包括整数、字符串和（与 C 不同）字典在内的类型分类。[12]

awk 语言是图灵完备的，而且可以读写文件。在一些版本中，它可以打开并使用网络套接字。但 awk 主要用作报表生成器，尤其用于解释和减少表列数据。它很少单独使用，而是内嵌在脚本中使用。第 9 章有关 HTML 生成的实例分析中就包括了一个 awk 程序实例。

包括 awk 案例分析的目的是，指出 awk 不是一个值得效仿的模型：实际上，从 1990 年以来，它很大部分已经被逐渐废弃了，为新派脚本语言所取代

---

[11] 这句话出自 Alan Perlis，他在 1970 年左右完成了软件模块化方面的一些先驱性工作。所质疑的分号是 Algol 系列的各种语言，包括 Pascal 和 C，把它用作语句分隔符或结束符。

[12] 对于那些从来没有用现代脚本语言编程的读者来说，字典就是一种关键字和值对应的查找表，经常通过散列表实现。C 程序员花费了大量时间编写各种复杂的方式来实现字典。

——特别是 Perl，就是明确成为 awk 的替代程序而设计的。原因值得研究，因为 awk 构成了对微型语言设计者的部分教训。

awk 语言最早的设计目的是针对报表生成的一种小巧、有表达力的专用语言。不幸的是，结果它在复杂度和能力的取舍上做得并不好。作用语言并不紧凑，但它依靠的模式驱动框架阻止了它的通用性——这是两个世界的最糟部分。新派脚本语言可完成 awk 所能完成的任何任务；它们的等价程序，至少和它一样易读。

> awk 逐渐被废弃，还因为更现代的 shell 具有浮点运算、关联数组、正则表达式模式匹配和子串能力，因此无须产生进程生成的开销就能完成小型 awk 脚本的等价程序。
>
> —David Korn

Perl 于 1987 年发布后的数年内 awk 还有竞争力，因为 awk 的实现更小更快。但随着处理器和内存的成本下降，支持相对节约的专用语言在经济方面的优点就不起作用了。程序员不断选择用 Perl 或（后来的）Python 来完成 awk 能完成的任务，而不是在头脑中记住两种不同的脚本语言。[13]到 2000 年，awk 已经成为 Unix 老牌黑客的一种记忆，而且并不留恋。

成本的下降已经改变了微型语言设计中的权衡。通过限制设计能力来提高设计紧凑性可能仍然是个好主意，但通过限制设计能力来节约设备资源则不然。随着时间的推移，设备资源变得越来越廉价，但程序员头脑中的空间只会越来越昂贵。现代微型语言，要么就非常通用而不紧凑，要么就非常不通用而紧凑；不通用也不紧凑的语言则完全没有竞争力。

案例分析：PostScript

PostScript 是一个专门向成像设备描述排版文本和图形的微型语言。它以著名的 Xerox Palo Alto 研究中心的设计工作和最早的激光打印机为基础，

[13] 我曾经是 awk 奇才，但别人提醒我，这个语言只适用于 HTML 的生成，这是本书唯一出现 awk 实例的地方。

并移植到 Unix 中。自 1984 年首次商业发布以来，它只能作为 Adobe，Inc.，的专属产品使用，而且用在苹果机上。1988 年根据授权规定被克隆成非常接近开源的版本，从那以后就成为 Unix 中打印机控制的事实标准。完全开源的版本和大多数最新的 Unix 版本一起发布。[14]PostScript 有很好的技术介绍。[15]

PostScript 和 troff 标记在一些功能上很像：两者的目的都是为了控制打印机和其它成像设备；两者通常都由程序或宏语句包生成，而不是手工编写。但是，考虑了 troff 请求是一套无止境增长的格式控制代码集，并已经具备了一些语言特性之后，PostScript 从头开始作为语言设计，因此更加具有表达力、更加有效。PostScript 之所以有用，就是对用 PostScript 编写的图形的算法说明比它们渲染出的位图更加小，而且占据的内存和通信带宽也更少。

PostScipt 显然是图灵完备的，支持条件、循环、递归和具名过程。类型分类包括整数、实数、字符串和数组（数组的每个元素可以是任何一种类型），但没有结构或等价物。从技术上而言，PostScript 是一种堆栈式语言；PostScript 的原语过程 (primitive procedure)（操作符）的参数通常是先进后出式参数堆栈，结果值重新入栈。

在总计 400 个左右的操作符中大约有 40 个是基本操作符。其中完成任务最多的是 show，即在页面上画一条线。其它操作符包括设置当前字体，改变灰度或色度，画直线、曲线或贝塞尔曲线，填充闭区域，设置剪切区域等等。PostScript 解释器负责把这些命令解释成位图以便显示或打印。

另外一些 PostScript 操作符实现算术、控制结构和过程。这些操作符允许把重复或定型的图形（如由重复字体构成的文本等）表达成带图形的程序。PostScript 的部分效用基于这样一个事实，即打印文本或简单向量图的 PostScript 程序比文本或向量渲染出的位图容量小，而且与设备无关，在网络电缆或串行线上传输速度更快。

---

[14] GhostScript 的项目网站。

[15] A First Guide To PostScript （PostScript 入门指南）。

从历史角度而言，PostScript 的堆栈式解释与一个叫 FORTH 的语言很像，该语言是为了实时控制 1980 年代非常流行的 telescope motor 而设计的。堆栈式语言以支持紧凑、经济的编码而著名，但也以难读而臭名昭著。PostScript 也有这两个特点。

PostScript 经常被实现为打印机的固件。开源实现的 Ghostscript 可以把 PostScript 转换成各种图形格式和（比较弱的）打印机控制语言。其它大多数软件都把 PostScript 看成最终的输出格式，意味着可以输送给具有 PostScript 能力的图形设备但不能用手工编辑或肉眼检查。

PostScript（无论是最初形式还是具有边界框以便嵌入到其它图形中的小变种 EPSF）都是专用控制语言的好设计实例，值得作为模型仔细研究。它也是其它标准的组成部分，如可移植文档格式 PDF 的标准。

案例分析：bc 和 dc

我们首先在第 7 章作为 shellout 的实例分析研究了 *bc*(1) 和 *dc*(1)。它们是命令性专门领域微型语言的实例。

> dc 是 Unix 上最古老的语言：它在 PDP-7 上编写而成，在 Unix 自己移植过去之前就已经移植到了 PDP-11 上。
>
> —Ken Thompson

这两种语言的领域都是无限精度算术。其它程序可以使用这两个语言进行计算，而无需考虑完成这些计算所要求的特殊技法。

> 事实上，设计 dc 的初衷不是为了提供通用的交互式计算器，那可用简单的浮点程序来实现。真正的动机是贝尔实验室对数值分析的长期兴趣：针对数值算法的常量计算，确切地说，得到了能够计算更高精度而不是运算本身所能使用的精度的大力帮助。因此 dc 是任意精度算法。
>
> —Henry Spencer

和 SNG 及 Glade 标记一样，这两种语言的一个优势在于它们的简单性。一旦知道 *dc*(1) 是一个逆波兰标记法计算器，*bc*(1) 是一个代数标记法计算

器，则这两个语言的交互使用便毫不困难。我们将在第 11 章重新讨论最小立异原则的重要性。

这两个微型语言都有条件和循环：它们都是图灵完备的，但类型的分类很有限，只包括无限精度的整数和字符串。这使得它们处在解释性微型语言和完全脚本语言之间。这些编程特性的设计目的是，让它们不要作为计算器涉足普通计算；实际上大多数 dc/bc 用户可能都没有意识到这些。

通常，dc/bc 可对话式使用，但它们支持用户定义程序库的能力使得它们具备额外功用——可编程性。这实际上是命令性微型语言最重要的优势，我们在 DWB 工具的实例分析中发现，无论程序的正常模式是否是对话式，这个优势都非常有利；可以用它们来编写高级程序，并加入针对任务的专门智能。

由于 dc/bc 的界面非常简单（送入一行表达式，得到一行计算值），只要把这些程序作为从进程调用，其它程序和脚本都可以轻而易举地获得所有这些能力。例 8.6 就是一个非常著名的例子，这是一个用 Perl 实现的 Rivest-Shamir-Adelman （RSA）公共密钥算法，广泛发布在签名档和 T 恤上，以示对美国 1995 年的限制密码学出口的抗议；它交给 dc 来完成要求的无限精度算法。

例 8.6 使用 dc 的 RSA 实现

```
print pack”C*”,split/\D+/,‘echo ”16iII*o\U@{$/=$z;[(pop,pop,unpack
”H*”,<>)]}\EsMsKsN0[lN*1lK[d2%Sa2/d0<X+d*lMLa^*lN%0]dsXx++\
lMlN/dsM0<J]dsJxp”|dc‘
```

案例分析：Emacs Lisp

一个专用的解释语言不仅可作为从进程运行完成专门任务，也可成为整个体系的核心；我们将在第 13 章分析这种方法的优点和缺点。troff 请求是早期范例；如今，Emacs 编辑器是最著名、最有效的现代实例。Emacs 是围绕 Lisp 的语言构建而成的，原语既可说明编辑缓存的动作，也可控制从进程。

Emacs 语言对于描述编辑动作和作为其它程序的前端来说足够强大，这意味着它除了常规编辑外还可用于许多其它任务。我们将在第 15 章分析 Emacs 用于日常程序开发（编译、调试、版本控制）的专门能力。Emacs 的“模式”是用户定义的程序库——用 Emacs Lips 编写的程序，针对具体任务定制编辑器——通常是有关编辑的任务，但也有例外。

这样就有了特制的模式，知道大量编程语言的语法，也知道 SGML、XML 和 HTML 等标记语言的语法。但很多人也用 Emacs 模式来发送和接收邮件（这些把 Unix 的系统邮件实用程序当作从进程使用）或 Usenet 新闻。Emacs 可以浏览网页，或作为各种聊天程序的前端。还有一个日历包和 Emacs 自带的计算程序，甚至还有相对广泛的可以选择的 Emacs Lisp 模式游戏（包括模仿 Rogersian 精神病学家的著名 ELIZA 程序的各种后续版本）。[16]

案例分析：JavaScript

JavaScript 是为了嵌入 C 程序而设计的一种开源语言。尽管它也嵌入在网络服务器中，但最早也最出名的表现形式还是客户端 JavaScript，可以让我们在网页中嵌入可执行代码，以供任何具有 JavaScript 能力的浏览器运行。这就是我们要在此处讨论的版本。

JavaScript 是充分的图灵完备的解释性语言，和 Python 一样，具有整数、实数、布尔逻辑、字符串和轻量级的字典对象。数值是带类型的，但变量可容纳任意类型值；类型之间的转换在很多上下文中自动进行。语法上，JavaScript 和 Java 很像，都受到了 Perl 的一些影响，具有类似 Perl 的正则表达式。

尽管有这些特点，客户端的 JavaScript 还不是一个相当通用的语言。它

[16] 利用现代 Unix 机器完成的最蠢的一件事是运行 Emacs 的 Eliza 模式来限制从 Zippy the pinhead 的随机引用。键入 **M-x psychoanalyze-pinhead**; 不耐烦了就键入 control-G。

的能力被严格约束，以防止通过包含 JavaScript 的网页攻击浏览器。它可以接收来自用户的输入、生成或修改网页，但不能直接改变磁盘文件的内容，也不能打开自己的网络连接。

随着时间的推移，JavaScript 语言变得更加通用了，而且对其客户端环境的约束也放松了。这是任何一个成功的专用语言随着其能力在开发者和用户的头脑中逐渐展现时可预期的结果。客户端 JavaScript 现在也可以通过在一个叫做浏览器 DOM（文件对象模型）的单个对象中读写值而与环境交互。虽然这个语言仍然给浏览器遗留了一些 DOM 无法支持的 API，但这些 API 都已过时，没有列入 JavaScript 的 ECMA-262 标准中，而且在将来的版本中可能也不会支持。

JavaScript 的标准参考书是《JavaScript: The Definitive Guide》（JavaScript：权威指南）[FlanaganJavaScript]，源码可下载。[17]JavaScript 成为一个有趣的实例有两个理由。首先，它是我们无需真正进入但可尽量接近的通用语言。其次，客户端 JavaScript 及其通过单个 DOM 对象的浏览器环境之间的结合是个非常棒的设计，能够作为其它嵌入情形的借鉴模型。

设计微型语言

什么时候适合设计一个微型语言？我们已经发现，微型语言提供了把问题说明规格提升一个层次的方法，并已经提供了数个案例分析中的施行方式。这个结论另一面是，只要应用领域的域原语简单而固定不变，微型语言就可能成为一种好方法，但用户可能希望应用方式要灵活多变。

相关观点，请参考对Alternate Hard And Soft Layers的说明和对脚本部件的设计模式的说明。

对微型语言的设计风格和设计方法的有趣总结参见《Notable Design Patterns for Domain-Specific Languages》（领域专用语言的著名设计模

[17] JavaScript 的开源 c 和 Java 实现可在这里获取。

式）[Spinellis]。

选择正确的复杂度

照例，设计微型语言时要记住的第一要素是尽可能保持微型语言的简单。我们用来组织实例分析的分类图表明了复杂度的层次；应该尽可能保持我们的设计靠近左手边。如果只设计一个结构化的数据文件，而无需设计在解释时要修改外部数据的微型语言便可以达到目的，那就尽力去做。

坚持结构化数据而不是微型语言的一个非常实际的理由是，在网络世界，嵌入微型语言功能容易导致滥用，从而引起不便甚至是危险。JavaScript 是“不便”类型中的主要实例：它的设计者并没有料到它会用于弹出式广告，而且是这样令人反感，以至于有人要求浏览器禁止解释 JavaScript。

Microsoft Word 的宏病毒则展示了这种事情会变得多么危险，这个安全漏洞每年造成的停机时间要耗费数十亿美元，而且严重损失生产力。尽管全世界至少有 2000 万[18]Unix 用户，但从来都没有任何和 Windows 频繁爆发的宏病毒类似的 Unix 病毒爆发，注意到这一点非常有益。造成这个事实的原因有好几个，包括 Unix 是本质上更好的安全设计：但至少有一个原因应归功于一事实——Unix 邮件代理不会默认执行用户查看文档中的任何即时内容。[19]

如果有任何可能使应用程序的用户最后要运行来自不受信任来源的程序，应用微型语言的风险特性可能必须被禁止。Java 和 JavaScript 这样的语言已被明确地沙盒化（sandboxed）——也就是说，它们对环境的访问是受限的，目的不仅是为了简化设计，而且是为了防止有 bug 或恶意代码执行潜在的破坏性操作。

另一方面，很多不良设计都被设计者修补过了，但这些设计者并没有勇敢地正视，他们真正需要的其实不是数据文件格式而是微型语言这个事实。把语

---

[18] 2000 万只是 2003 年中期根据 Linux Counter 和其它地方统计数字的保守估计。

[19] 出于这个原因我们在第 6 章分析的 kmail 甚至不会追循 HTML 中的其它站点链接。

言一样的特性加进去作为事后补救措施，这样的事情发生得太频繁了。这个问题最普遍的两个表现是，薄弱而专用的控制结构，和孱弱或根本不存在的过程声明机能。

设计一个偶尔图灵完备的微型语言也非常危险。如果这样做了，将来的某一天，某个聪明的家伙可能认为他需要让语言来替他执行循坏和条件。但由于这些只能用一种晦涩的方式完成，所以他只能写出晦涩的代码。产生的结果在短期内可能有用，但对于他之后的人来说却可能是一个噩梦。

微型语言的设计不仅强大而且回报丰厚，但它同样充满了类似的陷阱。有这样一些设计，采取自底向上的方法把几个底层服务粘合在一起，在研究了问题域之后又会担心其组织是否得当。微型语言的优势之一是可以把一些自顶向下的决策放到微型语言的程序控制流中，从而帮助我们从自底向上的编程中得到一个良好的设计。但如果我们对微型语言设计的本身采取自底向上的方法，则最终我们很可能得到一种丑陋的语法，反映出一种薄弱的语言和未经仔细考虑的实现。

微型语言的设计中存在很多这样的地方，选择上的小小不同就会在工具的可用性和易用性方面造成巨大差异：

> 作为语言设计者，一个很好的原则是考虑给出错误信息的备选方式。当程序员的目的实在不明确时，给出报错信息非常合适，但在目的明确的多数情况下，让语言默默完成正确的事则极有帮助。一个很好的例子就是，C 语言在数组初始化列表的末尾允许额外增加一个逗号，使得数组初始化的编辑和机器生成都更加容易。反例则是各种 HTML 阅读器的过分挑剔，即使遇到极小的嵌套错误也会导致大段文档被悄悄略过。
>
> —Steve Johnson

在这个问题上，和其它地方一样，没有什么可以替代良好的鉴赏力和工程判断。如果要设计一个微型语言，不要半途而废。声明性微型语言应该具有一个明确、一致、类似自然语言的语法能够为人类所阅读。命令性微型语言则应该从用户应该熟悉的语言模型中套用一系列控制结构。把微型语言确实当作语言来考虑；问问自己一些美学方面的问题，如“把它编码进去舒服吗？”，甚至“看上去愉快吗？”在这里，和软件设计的其它地方一样，

David Gelernter 的格言也同样适用：美是抵御复杂的最后武器。

扩展和嵌入语言

一个非常重要的基础性问题是，能否通过扩展或嵌入现有脚本语言来实现自己的微型语言。这往往是实现命令性语言的正确办法，但对于声明性语言就不是那么合适了。

有时候，简单地在解释性语言中编写服务函数就可能实现命令性语言，为了这个目的，我们把这种解释性语言称作“主”语言。自己的微型语言只不过是加载服务库的脚本并把主语言的控制结构和其它功能当作框架使用。主语言提供的一切机能都不必亲自动手。

这是编写微型语言最简单的方法。老派的 Lisp 程序员（包括我在内）热爱这种技术并大量使用。它是 Emacs 编辑器的设计基础，而且在新派的脚本语言，如 Tcl、Python 和 Perl 中，也可以找到。当然，这种技术也存在缺点。

主语言可能无法作为所需要代码库的接口。或者，从内部而言，它的数据类型可能并不足以完成所需要的计算类型。又或者，测量过原型性能后，发现速度太慢。出现以上任何一件情况时，解决方案通常是用 C（或 C++）编码，然后把结果整合到自己的微型语言中。

是用 C 代码扩展脚本语言，还是在 C 程序中嵌入脚本语言，这个选择取决于为此设计的脚本语言本身。扩展脚本语言的方法是动态载入 C 库或模块，C 入口点成为扩展语言中的可见函数。也可以在 C 程序中嵌入脚本语言，方法是向解释器实例发送命令然后接收结果值在 C 中使用。

这两种方法都取决于在 C 的类型分类和脚本语言的类型分类中进行数据转换的能力。有些脚本语言为完全支持这种转换而设计。其中之一就是我们将在第 14 章分析的 Tcl。另外一个是 Guile，即 Lisp 的变种 Scheme 的开源语言。Guile 作为库发布，设计目的便是嵌入到 C 中使用。

可以扩展或嵌入 Perl（尽管在 2003 年还是相当痛苦和困难）。扩展

Python 很容易，但要嵌入 Python 则要困难一点；C 扩展特别频繁地用于 Python 中。Java 有一个接口可调用 C 的“原生方法”，只不过这个实践被明确阻止，因为它往往破坏可移植性。shell 的大多数版本都不是为嵌入性和扩展性设计的，但 Korn shell（ksh93 和后续版本）是个显著的例外。

支持在现有脚本语言上放置命令性微型语言的理由大多并不恰当。为数不多的几个好理由之一是确实需要实现自己的自定义语言进行错误检查。如果是这样，请参考以下关于 yacc 和 lex 的建议。

编写自定义语法

至于声明性微型语言，一个主要问题是，是否应该用 XML 作为语法基础并把语法规定为 XML 文档类型。对于层次复杂的说明性微型语言，这完全可能是正确做法，但我们在第 5 章有关设计数据文件格式时提到的告诫同样适用——XML 可能太大材小用了。如果不用 XML，就遵循最小立异原则，并支持我们描述数据文件时谈到的 Unix 约定（简单的面向标记的语法、支持 C 反斜杠约定等）。

如果确实需要自定义语法，yacc 和 lex（或所用语言中的本地等价实现）也许是我们最好的朋友，除非所用语言的语法很简单，而且手工编写递归下降分析器的工作量非常小。即便这样，yacc 也可能给我们更好的错误恢复。随着语法的发展，yacc 的说明更容易修改。参见第 9 章有关不同实现语言中从 yacc 和 lex 派生的工具。

即使我们决定实现自己的语法，考虑一下重用现有工具所能得到的好处。如果确实需要宏机能，考虑一下用 *m4*(1) 进行预处理是否正确——但首先注意下一节提出的警告。

宏——慎用

宏扩展机能是早期 Unix 语言设计者青睐的策略：当然，C 语言有宏扩展，而且我们已经在一些更加复杂的专用微型语言，如 *pic*(1) 中也看到了

它。m4 预处理器是实现扩展预处理器的通用工具。

很容易说明和实现宏扩展，而且可以用宏扩展实现很多聪明的技巧。早期的设计者似乎都受到了使用编译程序的影响。在汇编程序中，宏经常是架构程序唯一的可用方法。

宏扩展的优势在于它无需知道基础语言的任何基础语法就可以用来扩展该语言。不幸的是，这种能力很容易被滥用，生成奇怪、不透明的代码，成为滋生难于辨识 bug 的温床。

在 C 中，这种问题的典型例子是这样一个宏：

```
#define max(x, y)    x > y ? x : y
```

这个宏至少存在两个问题。一个问题是，如果任何一个参数是个表达式，包括比 > 或?: 优先级更低的运算符，则结果可能令人惊讶。考虑一下表达式 max(a = b, ++c)。

如果程序员忘记 max 是个宏，则他可能会期望先完成赋值 a=b 和对 c 的预增量计算，然后把计算值作为参数输送给 max。

但事实却不是这样的。相反，预处理器会把这个表达式扩展成 a = b > ++c ? a = b : ++c，其中 c 编译器的优先规则使它解释为 a = (b > ++c ? a = b : ++c)。其比较结果被赋值给 a!

这种不良作用可以通过宏定义的防御性编码得到避免。

```
#define max(x, y)    ((x) > (y) ? (x) : (y))
```

采用这个定义，宏会被扩展为 ((a = b) > (++c) ? (a = b) : (++c))。这解决了一个问题——但请注意，c 可能被自增两次！这个陷阱还有更加狡猾的一个版本，比如，向宏传递一个带副作用的函数调用。

一般来说，宏与带副作用的表达式之间的交互作用可能导致不幸的结果，而且难以诊断。C 的宏处理器是一个有意设计得非常轻巧和简单的处理器；更强大的处理器可能会带来更糟糕的麻烦。

TEX 格式语言（参见第 18 章）很好地说明了这个普遍存在的问题。TEX 是有意的完备图灵机（具有条件、循环和递归）；尽管它能完成惊人的任务，但 TEX 代码往往极难阅读和调试。LATEX 是使用最广泛的 TEX 宏语句包，其源文件就是一个很有教育意义的例子：很好地符合了 TEX 的风格，但即便这样，也太难读懂。

与此相比，一个小问题是，宏扩展往往扰乱了错误诊断。基础语言处理器生成的错误报告针对的是宏扩展文本，而不是程序员正在查看的原始文本。如果两者的关系被宏扩展弄得很混乱，则生成的错误报告可能非常难于和错误的实际位置联系起来。

当宏具有多行扩展、有条件地包括/排除文本、将改变扩展文本的行数时，预处理器和宏之间就特别容易产生问题。

内嵌在语言中的宏扩展可以完成一些自我补救、重定行数以备查阅扩展前的文本。例如，*pic*(1) 的宏功能就能这样做。如果宏扩展由预处理器完成，则这个问题更加难以解决。

C 预处理器的解决方法是，无论何时只要包括扩展或进行多行扩展就发送`#line` 指令。C 编译器可对此做出解释并相应地在错误报告中调整行数。不幸的是，m4 却没有这样的功能。

有理由非常谨慎地使用宏扩展。Unix 经验的长期教训之一是宏产生的问题比宏解决的问题多。现代语言和微型语言的设计已经越来越远离宏了。

语言还是应用协议

我们必须要问的另外一个重要问题是，微型语言引擎是否可被其它程序作为从进程交互调用。如果可以，我们的设计可能看上去不太像可供人类交互的会话式语言，而更像我们在第 5 章分析的应用协议。

主要区别在于事务边界的标定程度。人类擅长发现 CLI 的会话式输出在哪里结束，下一个输入的提示在哪里。他们可根据上下文来判断什么重要，什么应该被忽略。计算机程序要完成这个就困难多了。如果输出没有明确的结束标记或没有提前告诉输出长度，则机器就无法判断何时停止读取。

> 更糟糕的事发生在程序的输入被缓冲起来（经常是无意的，如 `stdio` 所为）的时候。在正确的地方公然停止读取的程序仍然可以超界读取。
>
> —Doug McIlroy

如果从属进程没有为这个问题专门设计便和主进程进行通信，如果它们之间同步失败（我们在第 7 章首次提到这个问题），死锁就更容易发生。

对于那些未经仔细设计的微型语言，存在一些斡旋方案。其中绝大多数的原型都是 Tcl 的 expect 包。这个包协助 CLI 间的对话。它围绕以下操作建造：自从进程不断读入，直到匹配某个指定的正则表达式或超时。有了这个操作（当然，以及发送到从进程的操作），即使从进程并非为与其它程序通讯这个角色定制，也往往可能使主从进程间的对话更加可靠。

其它语言中有工作方式类似的 expect 包：在网上以自己最喜爱语言的名称并加上“Tclexpect”关键字进行搜索，很可能找到一些有用的内容。然而，作为微型语言设计者，最好不要假设所有的用户都精通 expect。即使用户是 expect 专家，这也是一个特别厚的胶合层，容易在这儿犯错。

设计微型语言时要小心这个问题。增加一个选项，改变会话行为，使其相应地更像应用协议，同时给出明确的输出结束分隔符和类似的字节补齐，也许是个不错的主意。

生成：提升规格说明的层次

> 程序员束手无策……只有跳脱代码，直起腰，仔细思考数据才是最好的行动。表达是编程的精髓。
>
> 《人月神话，二十周年纪念版》(1975-1995)，103 页
>
> —Fred Brooks

我们在第 1 章中说过，人类其实更善于肉眼观察数据而不是推导控制流程。概括如下：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来得清楚明了？或者（更明显地），比较一下究竟用数组初始化器来表示转换表，还是 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。[1]

数据比程序逻辑更易驾驭。无论数据是普通表格、说明性标记语言、模版系统还是一组可以扩展成程序逻辑的宏，这一点都成立。尽可能把设计的复杂度从程序代码转移到数据中是个好实践，选择便于人类维护和操作的数据表示法也是好实践。把这些表示法转换为便于机器处理的形式是机器该干的事，不是人类的任务。

> 更高级、更富说明力的标记法应具有另一个重要优点，就是更便于利用编译期检查。过程式标记法天生具有复杂的运行时行为，很难在编译期分析。而说明性标记法使人们可以更彻底地理解预期行为，令实现中的错误更容易发现。
>
> —Henry Spencer

---

1 关于这一点的进一步展开请参考 [Bentley]。

这些领悟在理论上都以一系列实践为基础，而这些实践始终是 Unix 程序员工具包的重要组成部分，这些工具包包括高级语言、数据驱动编程、代码生成器、特定领域的微型语言等。这些工具的共同点是，它们都是提升代码生成层次从而使规格说明更简炼的方法。此前我们已经注意到，缺陷密度与编程语言无关；所有这些实践都意味着，无论邪恶力量产生什么作用，我们的 bug 都只会以更少的行数来实施破坏。

我们第 8 章讨论了特定领域微型语言的用法。我们将在第 14 章详细说明非常高级的语言。本章我们要分析数据驱动编程的设计实例和一些专用代码生成的例子：我们将在第 15 章分析一些代码生成工具。和微型语言一样，这些方法都能大幅减少程序的行数，并相应减少调试时间和维护成本。

附录

缩写术语表如 API 之类的请自行搜索之，这里就省略了。

Event timelines of the Unix Industry and of GNU/Linux and Unix are available on the Web. A timeline tree of Unix releases is also available.

参 考 文 献

[Appleton] Randy Appleton. Improving Context Switching Performance of Idle Tasks under Linux. 2001.Available on the Web.

[Baldwin-Clark] Carliss Baldwin and Kim Clark. Design Rules, Vol 1: The Power of Modularity. 2000. MIT Press. ISBN 0-262-024667.

[Bentley] Jon Bentley. Programming Pearls. 2nd Edition. 2000. Addison-Wesley. ISBN 0-201-65788-0.

The third essay in this book, “Data Structures Programs”, argues a case similar to that of Chapter9 with Bentley’s characteristic eloquence. Some of the book is available on the Web.

[FlanaganJavaScript] David Flanagan. JavaScript: The Definitive Guide. 4th Edition. O’Reilly & Associates. 2002. ISBN 1-596-00048-0.

[Spinellis] Journal of Systems and Software. Diomidis Spinellis. “Notable Design Patterns for Domain-Specific Languages”. 56. (1). February 2001. p91-99. Available on the Web.

[Tidwell] Doug Tidwell. XSLT: Mastering XML Transformations. O’Reilly & Associates. 2001. ISBN 1-596-00053-7.