第 4 章 “笨”出来的文化和哲学

1997 年对于我来说，是一个比较重要的年份。因为这一年我拥有了一台属于自己的多媒体电脑，而且还预装了我认为是当时最先进的Win95 操作系统，甚至还带有一个不知道干什么用的 33.6K的Modem。从此便渐渐远离了DOS、远离了UCDOS、CCED[①]、WPS，也远离了我最喜爱的《仙剑奇侠传》[②]。伴随而来的是VB5、金山盘古和Louts123，以及后来的《剑侠情缘II》。这一年我听说了Unix这个操作系统。但是当我知道它是 1969 年的产品时，就很不屑地将它抛在了脑后。这一年开始订阅《电脑爱好者》杂志（**个人觉得这句话说得有点过于庄重了**）。

转眼就到了 1998 年，《泰坦尼克号》上映。作为当时的潮男靓女们自然也不会放过。于是请来班里几乎所有的同学齐聚我家，围在我的多媒体电脑前面，一同欣赏花 2 块钱租来的 3 张 VCD。好多女生被杰克和露丝的真爱所打动，流下了几滴鳄鱼的眼泪。我也被电影中气势磅礴的场景所震动，真想知道他们是怎么拍摄的。

在小女生们还在悄悄谈论自己什么时候也能遇到像杰克这样的白马王子的时候，我的新一期《电脑爱好者》到了。其中一篇名为《Linux——自由之花含苞欲放》的文章让我眼前一亮，终于知道了《泰坦尼克号》那气势磅礴的场面不是拍出来的，而是用运行 Linux 的电脑制作出来的。也知道了这个世界上还有比 Win95 更强大的系统，而且还有源代码可看。Linux 成了我当时唯一的追求。但是一直苦于不见庐山真面目，只能零星地找一些资料意淫一下。

幸运很快就降临到了我的头上。一次去同学家借电脑游戏，在他的一堆盗版游戏盘中找到了一张封面印有 Linux 字样的“游戏”盘，兴奋得我都没跟同学说再见就抱着它飞回了家。之后就遭到了 Linux 一系列的折磨，这个“游戏”真是太不好玩了。

要说就此放下，那实在不是我的性格。作为那个时候必然是单身的我，一定要唱响“爱要越挫越勇，爱要肯定执着”的单身情歌，四处去寻找“翻先生”。

功夫总是不负有心人的。一日，跟几个要好的同学到一位女同学家去玩，在她爸爸的书柜里找到了一本有关 Unix 使用方法的书（书名想不起来了）。一边嘴里念叨“你爸也太过时了吧，还在看……”，一边不屑地翻看几页。我“看”字还没说完，我的眼睛几乎都要掉出

---

① 一个当年最为流行的电子表格制作软件。

② 因为要在 Win95 的 DOS7 下空出 520K 的内存是挺费事的事情。

来了。里面所讲述的内容，与我正在研究的 Linux 几乎一模一样，这似乎就是我梦寐以求的有关 Linux 的书。再看看封面，确实是关于 Unix 的，没有 Linux 的半点影子。不管了，先借回家再说吧。

在抱着这本书回家的时候，还在想：这本书里讲述的内容怎么跟我的 Linux 那么像，它能帮我搞定 Linux 吗？心中充满了各种疑问。不知不觉已经回到了家中。

迅速打开电脑，照着书上的内容，逐一的试验。试验的结果让我惊叹不已。在 Linux 上得到的结果跟书上说的一模一样。一度认为是书印错了，毕竟“Unix”和“Linux”有点像的。可是细想一下，出版社不至于犯这种连低级都说不上的错误。虽然通过这本有关 Unix 使用方法的书让我可以自如地使用 Linux 了，但是我依然很茫然：一个 90 年代的产品怎么能跟一个 60 年代的产品那么像？带着这种茫然，我继续去探究。这让我感觉穿越到了 60 年代，那些“史前黑客”们就围绕在我左右，甚至想像自己就是其中的一员。这进一步让我对 Linux 着迷，着迷到差一点与 Win98 失之交臂。由于一直沉浸在自己是“史前黑客”这个梦里，一直想着如何入侵五角大楼，也让我给家里带来了一个不小的“祸端”。

在一个风雨交加的夜里，写完作业，马上开启我的宝贝电脑，像往常一样开始仔细研读那本有关 Unix 使用方法的书，并在 Linux 上尝试验证。不知过了多久，我已经开始有点睡意了。但是当看到书上说黑客们可以通过电话线，利用一个叫 Modem 的东西能进入别人的电脑时，立马就精神起来。依稀记着我的电脑也应该有 Modem。在 Linux 下通过命令查看一下，确认了我的记忆。这可乐坏了我。于是接上电话线，照着书上的例子一步一步地操作起来。很快，电脑里传来了“吱……吱……滴答滴答……”的声音，这让我更加兴奋，根本就不想睡觉了，立马就开始畅想自己就是一名黑客了。当我回过神之后，声音也消失了，提示我输入用户名和密码。虽然反复试了几次都没登录成功，但依然在暗自庆幸：“嘿嘿，只差最后一步了。”（**被你打败了**）

在接下来的一个星期里，几乎每天晚上都要试上一试，研究一下该如何才能绕过密码验证这一关。有时甚至在上一些不敢兴趣的课时，也在寻思这件事情。

直到有一天，我在课堂上，具体上什么课已经不记得了，正在为这件事情发呆的时，老师突然叫到了我。正当我在想该如何混过老师的提问时，老师告诉我是我妈来找我，才稍微定了定神。但马上觉得事情有点不对，我妈找我来干什么？虽然不知道来者何意，但总比被老师骂好。于是就兴冲冲地冲出了教室。

在我左脚刚刚迈过教室的门，右脚还在教室里的时候，也没来得及看清我妈站在什么地方，一个巴掌就重重地打在了我的脸上。

“小兔崽子，你给谁打电话，花了我 3000 多块钱。”

“我没打电话！”我很冤枉地辩驳道。

“你还嘴硬，你看咱们家这个月的话费单。”我妈把一叠纸推在了我的面前。

我揉了揉刚刚被打得满天星光的眼，接过我妈手里的话费单一看，我也懵了。还以为眼睛被打坏了，又揉了揉，确认眼睛没坏。话费单上清晰地记载着 100 多个越洋电话记录。马上就想到了这几天自己做过的事情。原来每次远程连线，我都拨到了美国，也难怪这么贵了。于是就向妈妈如实交代了我这几天的所作所为。

我妈的确是天底下最好的人。当我把这些说完，她不但不再生气，还主动带我去开通了一个互联网账号。从此我便认识了“163”，虽然那个时候要 9 块钱一小时。

借助于互联网，让我更加深入地认识了 Linux，知道了它实际上是 Unix 的一个克隆，但是不含一行 Unix 的源代码。也知道了起源于 60 年代的 Unix 并没有消亡，而且还更加繁荣。不但有 Linux 这样的克隆，还有 FreeBSD、Solaris 这样的传统分支。整个互联网世界也是 Unix 的天下，Win9x，乃至 WinNT，在互联网世界都不值得一提。感谢我的妈妈，有了您的支持，让我开足了眼界。

Unix，这个源于 1969 年的设计，缘何能够让人们如此地心驰神往，以至于诞生自 1991 年的 Linux 也要克隆自它，甚至不含有它的一行代码，外在的表现也要与它极其相似呢？不能不说这是一种魅力，一种能够让人魂牵梦绕为之神伤的魅力。这种魅力源自于 Unix 那种不可多得的、经久不衰的传统文化和设计哲学！

4.1 Unix 的文化和哲学

文化与哲学是一对没有分裂完好的连体孪生兄弟。哲学是文化的结果，文化是哲学的一种沉淀。世界上不存在一种脱离了文化的哲学，也没有一种脱离了哲学的文化。换句话说，有什么样的文化就会产生什么样的哲学，有什么样的哲学就会促使什么样的文化形成。

传统科学文化由是如此。这方面比较好的佐证就是杨振宁[①]大师归国后潜心研究《周易》十几年得出来的一个结论：“中国文化近几百年来没有重大科学发现的根本原因是：《周易》的哲学思想只有‘归纳’。不具备西方哲学所侧重的‘推演’。”虽然炮轰者甚众，但如果从杨振宁是个物理学家这个角度来看，还是情有可原的。因为证明现代物理理论正确的普适方法是：先从数学推导再进行哲学推导。只要数学上正确且不与哲学相悖，就会认为这种理论是正确的。可见哲学在人类传统科学文化发展中所占据的是怎样的一个地位了。

然而，传统科学领域与计算机科学领域又有很大不同。计算机科学领域技术变革如此之快，以至于软硬件环境的变化可以用日新月异来形容。计算机科学技术暂如朝露，很难演变成为一种文化。可是，例外就像面包和牛奶一样，总是会有的。的确有极少数的计算机技术被证明经久耐用，足以演进成为一种强势的技术文化、有鲜明的技术特色和世代相传的设计哲学。

Unix 文化便是其一。互联网文化又是其一。由于 Linux 诞生于互联网，发展于互联网。

① 著名美籍华裔科学家、诺贝尔物理学奖获得者。广为国人所知的大事记是 82 岁高龄时迎娶 28 岁的翁帆为妻。

因此，Linux 能够非常荣幸地将这两者无可争议地合二为一。

Unix 的诞生年代对于计算机这门科学来说，是远古时期的 1969 年，基本上就是一个古生物，活化石。在分时系统领域，除了 IBM 的 VM/CMS，没有谁敢说它比 Unix 资格更老。尽管计算机领域的其他技术犹如蜉蝣般生生灭灭，计算机性能成千上万倍地增长，编程语言也历经嬗变，甚至这个行业的传统规范都有过数次变革，可 Unix 却依旧屹立不倒。究其原因则是 Unix 一直能够让人们对它的知识投资长期稳定不变。不变的东西有很多，如：编程语言、系统调用、工具用法等。它们有些已经存在了数十年。而反观其他操作系统，除了商标没变之外，有哪些东西还能让你燃起儿时的记忆呢？就像我之前提到的 UCDOS、CCED 等，甚至连商标都消失了。

这一切都归功于 Unix 那与生俱来的内在优势，归功于它的设计者们一开始就作出的正确的设计决策。这些设计决策，连同设计哲学、编程方法、技术文化一起，从 Unix 的婴儿期到今天的成长路程中，已经被反复证明是健康可靠的，从而才使得 Unix 取得了今天的成功。Linux 将这些融入自身，借助于互联网的自由、公平、开放、透明，迅速地成长起来，成为备受人们喜爱和追捧的新一代操作系统。

Unix 的原本用途——作为大中型计算机的通用分时系统，已经开始遭到各种个人电脑的围剿，正在迅速地退出历史舞台。Linux 因继承了 Unix 文化所展现出来的特点，自然也锁定了它的一些用途。Linux 究竟能否在目前由微软主宰的主流商务桌面市场中占有一席之地，人们依然心存疑问。况且，AT&T、Sun、Novell，以及其他一些大型商业销售商和标准联盟在 Unix 定位和市场推广方面不断铸下的大错甚至都成了经典的笑柄。可是发展到今天，Ubuntu 等面向桌面的 Linux 发行版的繁荣，显然已经证明 Unix 文化拥有着如此顽强的生命力。即使十几年的管理不善都丝毫无法钳制它的勃勃生机。以 Linux 为代表的各种类 Unix 系统，正在迅速而有效地解决着 Unix 文化的问题，让这种文化渗透到了计算机行业的各个领域。即便是 Windows 这种与 Linux 格格不入的系统，也要抄上一抄。

虽然 Unix 这种传统文化和哲学有如此之魅力，但却是“笨”出来的。为此我给大家整理了“四大笨”以证明我的观点。同时期望藉此让大家去细细体会 Unix 的传统文化和设计哲学到底是什么，反映在哪些层面上，Linux 又是如何表达出来的！

4.2 “四大笨”之一：万般皆文本

初学 Linux 使用的人们，坐在 Linux 老手旁边，看他们帮自己解决一些工作上的问题时，往往会惊叹于他们似乎不怎么使用 GUI。而更让你惊讶的是他们频繁使用“管道”和“I/O 重定向”的频率。这让你总是有一种莫名的敬佩感不知觉地从心中幽幽然升起。

其实，Linux 老手们并不是刻意要在你们面前显摆他们有多么的牛 x 而去使用让你眼花的文本界面；更不是为了在你们面前装 13 而大量使用“管道”和“I/O 重定向”。只因这样

更简单、更灵活、更强大、更具效率，更符合 Linux 的使用方式和设计特点。

也许你还没有察觉，或是并没有太在意。不仅仅老手们喜欢使用文本界面，甚至在Linux中除了那些能够执行的程序文件之外，其他的一切几乎都是文本的。如存储用户名和密码的passwd文件，存储图片的xmp[①]文件等，甚至一些程序也是文本的。Linux中提供了好多与文本文件相关的命令，如cat、grep、more、diff、head、tail等，举不胜举。而且它们从来不生成什么文件，只将处理结果输出到屏幕上。即便要生成文件，也只能利用“IO重定向”来达成目的。

Linux 给人的感觉是它非常讨厌即轻巧又快捷的二进制，偏好即笨重又低效的文本，是不是很傻啊？可是这些并不是 Linux 的专利，Unix 也是这样！

4.2.1 二进制的烦恼

对于计算机程序而言，二进制数据是最容易处理的数据，因为计算机就是二进制的。但是二进制数据并不适合于人类的阅读和传播。

首先，由于人的祖先给人类遗传下来了十根手指，导致人能非条件反射地搞清楚 10 进制而搞不清楚二进制；其次，即便都是认识二进制的计算机，由于内部数据格式约定的不同（如高字节优先对低字节优先，或 32 位对 64 位）还会导致数据无法互用的问题。

为了解决人看不懂的问题，人们发明了大量二进制文件查看器。但似乎总是跟不上历史的脚步，过不了多久就会有那么几种被淘汰。少数能够保留下来的，也无法让人一目了然。人们只能继续去做大量的重复而又没有创造性的劳动。

为了解决二进制数据的互用问题，人们发明了好多种序列化与反序列化的方法。但经常遇到的麻烦就是，即使是一个很小的程序，序列化与反序列化的代码也会导致程序变得臃肿肥大。虽然 Python 和 Java 这样的现代语言内置了这种机制从而大大减少了工作量，但也因为种种原因，很多时候不能让人如意。最常见的情况就是数据传输的两端必须保持严格的一致，否则就会产生各种潜在的问题，而且这些问题很难被发现。尤其是在大型应用系统中，这种潜在问题的爆发，往往会引起故障雪崩。

二进制数据所存在的最根本的问题是它的不透明性。不只是对人的不透明，还包括对异构机器的不透明，以及对其他工具的不透明。这种不透明，导致了系统通用性的降低，从而也带来了扩展障碍和维护成本的显著提升。这种似乎躺着都会中枪的系统，最终的宿命只能是被历史所淘汰。

① Linux 下最常见的图片格式之一，最初是为 X Windows 而设计的。它是由 C 语言的语法构成的纯文本文件，可以直接包含在 C 语言程序中。

4.2.2 文本的快乐

文本相对于二进制的快乐首先是带给人的，因为人能够直接看懂。这就不需要为文本编写特定的解析工具，人不用去为它付出额外的劳动。毕竟“懒惰”也是人的本性之一。要不然我们发明计算机干嘛，直接掰指头算微积分那该有多“勤快”？

文本的另外一个快乐是带给异构计算机的，因为文本在任何计算机中的存储方式都是相同的。这里所指的文本是以 ASCII 编码的英文文本。如果你一定要强调中文，那么我就告诉你还有 UTF-8 这个玩意儿。不管怎样，这些编码在任何计算机中都支持，任何常见的操作系统也都支持，效果是完全一样的。如果一定要找到一个例外，那就是采用 MVS 系统的 IBM 大型机，它使用 EBCDIC 对文本进行编码。不过这东西很多人这辈子都是无缘与它相见的，所以也就别为这点事儿烦心了。

文本的这种对人的直观性和对机器的通用性，非常适合于做配置文件，更适合于传递消息。配置文件因为需要人来编写，使用文本很少会给人带来可配置内容上的理解障碍。使用文本通信，最大的好处就是在调试阶段。人能直观地看到输入了什么内容，输出了什么内容，这期间出了什么问题自然是一目了然了。

使用文本通信还可以为系统的未来省不少力气。最为著名的反例就是采用二进制通信的 TCP/IP 协议。IPv4 的地址是 32 位的，马上就要用光了。于是就提出了 IPv6，将地址扩展到 128 位。这个变动可不是仅仅修改一下地址长度的问题，几乎整个 TCP/IP 的架构都进行了重新设计，不但费时费力，还存在推广难题。如果当初 TCP/IP 采用的是文本通信呢？在地址需要更大的值时，直接写不就行了吗？

当然，文本也不是十全十美没有毛病的。计算机本身不知道文本是个什么东西，要想让它理解一段文本代表什么含义，就需要写程序去解析。这或许是大多数人不容易接受文本（主要是用于通信）的最根本原因。但是，在你还不能证明解析文本所带来的性能损耗就是系统性能瓶颈的时候，或已经证明那就是性能瓶颈但证明不了超过了性能需求预期的时候，就不要改主意。否则就是过早优化。如果你很想设计一个复杂的二进制文件格式，或一个复杂的二进制通信协议时，最好的办法就是去睡上一觉，或许在你醒来的时候这种感觉就过去了。

也正是由于文本的这个无法让计算机直接理解的特性，让你根本不敢去设计内容过于丰富，或结构异常复杂的文本内容，这等于是给自己找麻烦。因此而获得的好处就是程序之间不会互相干涉内部状态，强化了封装。文本的这一点点劣势，因其强化了封装，也成为了一种强劲的优势。

另外一个容易被人诟病的问题就是文本的位密度。但是细想想也不见得比二进制差多少。毕竟它们还是用了八位字节中的 7 位。而且还有一个不能忽视的事实就是当今世界上的无损压缩算法越来越先进高效。同等信息量的文本和二进制数据经过压缩后，在空间尺度上没有任何差别。如果再能忍受一下压缩算法所导致的效率损失，那么这个问题也就荡然无存

了。何况计算机发展的趋势是越来越快的，这点效率上的损失也会因此而在极短的时间内得到弥补。套用一句广告词：用文本，它好我也好*^_^*！

接下来我们再看看 Linux 是怎样使用文本的。

4.2.3 文本之于配置文件

用 Linux 的人，应该对/etc/passwd、/etc/group、/etc/inittab 等这些配置文件不陌生。它们都是使用文本的良好用例。但是这些配置文件并不是随意编写的，都有一个统一的风格。这个统一的风格在 Unix 世界称之为 DSV 风格。

DSV 是“Delimiter-Separated Values”的缩写，翻译过来是“分隔符分割值”。翻译的比较绕口，如果你有更好的翻译，请联系我。其实最主要的意思就是使用“分隔符”将一个一个的“值”分割开来，便于取值处理。“值”也可以理解为“字段”。

DSV 只是一种风格，并没有规定什么。所以一个 DSV 风格的文本文件，可以含有多种“分隔符”，也可以用多种方式理解“值”或“字段”。比如这些“值”可以有内置的特定含义，也可以是“键-值”对儿的形式，由“键”来修饰“值”的含义。甚至“值”或“字段”本身也可以是 DSV 风格的，用来描述更复杂的子项。

比如/etc/passwd 就是一个很好的例子。它包含有换行符（\n）和冒号“:”这两种分隔符。换行符将文件分割为多个独立的行，每行描述一个用户的账户信息。冒号用于分割账户信息的各种字段。至于“键-值”对儿的例子，恐怕最为著名的就是 Windows 的 ini 文件。用等号“=”来分割“键”和“值”，左为“键”右为“值”。虽然 Windows 与 Linux 是格格不入的两个东西，但是在这方面还是有些共同认识的。

DSV 风格的文件，如果遇到“值”或“字段”中含有“分隔符”的情况，一般建议使用反斜杠“\”进行转义。让使用转义方法显得更为强大的是读取这种文件的代码可以通过 C 风格的转义符嵌入非打印字符数据，进一步扩大了文本的应用范围。

DSV 风格的文件能够被绝大多数的传统 Unix 工具程序所支持，因此它具有极高的通用性。在我们自己编写程序的时候，为了能够利用这种 Linux 先天就具备的优秀资源，应尽量采用 DSV 风格来设计我们的文件，无论配置什么。

4.2.4 文本之于程序组合

一个人你让他同时做 100 件事，可能一件都做不好，或者还没等做到第 100 件事的时候就累死了。但是，如有 100 个人，每个人只让他做一件事，估计每个人都会“感谢你八辈祖宗”，兢兢业业地把自己的事情做好。

当你有一种机制，能让只做一件不同于其他人的事情的 100 个人，进行两两、三三或更

多个体的组合，你会发现这 100 个人可以完成更多的工作。人与人之间最大的障碍就是沟通。如果你选择的机制可以突破沟通问题，那么这 100 个人，甚至更少的人，就可以高效地完成任何的工作，包括那些你没有预想到的。当然，你可以谈什么成本效益问题。但是如果在你这里累死了人，不用说成本问题，以后是否还能有效益都是一个未知数。

程序也是如此。一个功能繁多的程序不可能比只有一个功能的程序更高效、更稳定；多个不同功能的程序有效地组合在一起，可以展现出比原有功能多得多的功能，以及一些你预想不到的功能；虽然多个程序组合在一起会有一些成本上的担忧，但是获得的效益远比付出的成本要高得多。

在输入输出方面，Unix 世界的传统是极力提倡采用简单、文本化、面向流、设备无关的格式，自然也包括 Linux。在这些系统下，多数程序都会尽可能的采用简单过滤的形式，即将一个简单的文本输入流过滤成一个简单的文本输出流。尤其是那些最基本的命令程序，有些甚至输入和输出的内容都是相同的。因为不这样做，它们就无法做到任意组合衔接。

文本流之于程序，就如同朴实无华的语言之于人一样，不需用机巧的智慧去理解它深层的含义，不会在沟通上形成障碍。文本流就像前面说的那样，能够加强程序本身的封装性。而那些所谓的精致的、易用的、强大的进程间通信技术，比如远程过程调用所使用的对象序列化，都会导致各程序间内部状态的频繁变化，从而给程序本身带来巨大的复杂性，难于开发和维护。

要想让程序具备良好的组合特性，就要使程序之间彼此独立。作为输入端的程序应该尽可能地不去考虑作为输出端的程序该是什么。让一端的程序能够替换成另一个截然不同的程序，而完全不会惊扰另一端的程序。文本流有其先天的优势，毕竟文本流在字节层面，是没有任何结构可言的。处理的方法只有一种，就是读取它。不管你是从文件、从标准输入，乃至于网络，都是这样。不管想怎么处理，你需要的数据就明显地放在那儿了，怎么解释它们都有道理。即便在最糟糕的情况下，你自己也能看懂它。

4.2.5 文本之于通信协议

既然文本流在程序组合过程中能够发挥巨大优势，那么在基于网络通信的这种另类的程序组合上文本流仍然会是救命良方。

在这方面，要单独从 Linux 中寻找例子是有些困难的。毕竟到了通信协议这个层面就已经超出操作系统的范围。但那些经典的、历史悠久的、至今仍被广泛应用的通信协议案例，却是无处不在的。这里首推的就是 HTTP 协议。

HTTP 协议就是一种基于文本的通信协议。它采用类似 RFC-822/MIME 格式的消息来发起或应答客户端的请求。在这种格式中，一个独立的文本行被视作一个消息字段。以冒号“:”后接空格做为分隔符，左侧为字段名，右侧为字段值。字段名不得包含空格，通常用横线“-”

代替空格。空行可以解释为消息的结束，也可以解释为接下来是非结构化文本，具体如何解释，由前面的消息属性决定。

HTTP 协议已经不仅仅被用于处理 Web 请求，基本上就是互联网的通用协议。自从万维网在 1993 年左右吸引到足够量的用户以来，应用协议的设计者们就越来越倾向于在 HTTP 之上构建自己的专用协议，并使用 Web 服务器作为通用服务器平台。或许总是有人对这一事实不愤，但是这就是潮流这就是趋势，不信咱们就走着瞧！

4.2.6 硬件也文本

从前面的介绍来看，说 Linux 已经用文本武装到了牙齿一点都不为过。但是这还没完，它连五脏六腑都没有放过——硬件也文本。

Linux 很崇尚的一种设计习惯就是：不管是保存在磁盘中的真实文件，还是用于保存文件的磁盘本身，乃至鼠标、键盘、显示器、声卡、显卡等一切硬件，它都认为是文件，更有甚的是连需要通过网络访问的其他电脑也没放过。会让你有一种：手拿文件句柄，走遍天下都不怕的感觉。虽然不至于所有硬件都能被容易地使用文本来操作，但绝大多数已经是这样了。不信你就打开/sys 和/proc 目录下面那些代表某个具体硬件设备的文件看看，是不是都是文本的？

这带来几大好处。第一，什么都是文件，程序员就开心死了，不用搞清硬件怎么工作的，直接当普通文件读写就能控制；第二，代表硬件的文件是文本，不用专门的程序就能打开看看它是个什么状态；第三，要修改硬件的特性，可能都不需要写程序了，直接用文本编辑器就能搞定；第四，文本相对于二进制的优点，即便硬件上有很大的变化，反映在文本中可能只是增加了几个字段或几行。

硬件也文本，并不是对文本的滥用。毕竟它给我们带来了实实在在的好处。要做评价，也只能说明文本的强大已经超出了我们的想象。

4.3 “四大笨”之二：四处用脚本

若说到在 Linux 下的编程，稍微对编程有所了解的人都会想到 C 语言。Linux 的内核、shell、基础命令程序，也的确是用 C 语言编写的。这首先证明了一点：C 语言很强很通用。到目前为止，C 语言依然垄断着计算机工业中几乎所有的系统编程。而且也正因为是 C 语言，才使得 Unix，以及后来的 Linux 能够这么广泛地被人们去研究、去改进、去制作自己的分支，以至于我们能在各种硬件平台上使用它们。

但是细心的人会发现，Linux 启动过程中所涉及的各种程序，很少有 C 语言的痕迹。它们大多是“文本”，能够被计算机执行的“文本”。不单单在启动过程中是这样，那些用于

安装软件的工具 yum、apt-get，甚至是 configure 和 Makefile 也都是文本。而且你可能还没注意到，那些用于系统管理的工具，如配置 ADSL 拨号上网的工具、配置守护进程的工具等，很多也都是“文本”的。

估计你现在一定会骂我在混淆视听，因为那些根本就不是什么“文本”，那是程序，是“脚本程序”。是的，那些就是“脚本程序”。我的确有混淆视听的嫌疑，不过我只是想强调一下“脚本程序”是以文本的形式存在的这个特点。可以很容易地让人看清它们做了什么，也很容易修改。

四处用脚本是所有类 Unix 系统不同于其他系统的一个显著特征。

4.3.1 富饶的脚本语言

催生人们在 Linux 中大量使用“脚本”来编写程序，并不仅仅是因为“脚本”对人直观、容易修改这种显著特性所决定的。另外一个主要的原因就是 Linux 所支持的脚本语言种类十分丰富。

所有类 Unix 系统所必备的 shell，其本身就是一个强大的脚本解释器。所以从 shell 诞生的那一天起，shell 就是那些不懂 C 语言，又必须在 Unix 上编写程序的用户们的首选工具。

这就给了人们一种新的选择。使用 shell 编程不用去理会让人头晕的“指针”；shell 程序可以直接利用系统命令来完成一些需要用大量 C 代码的功能；shell 编程不用去理会数据类型，不用考虑烦人的数值和字符数据的转换问题；shell 程序同样提供顺序、选择分支和循环这三种能够构建任意算法的基础设施。因此，很快就能够被非专业用户所接受、掌握，并编写出非常实用的程序。

随着时间的推移，这些非专业用户想往更高的方向发展，遇到了一些 shell 处理起来会很蹩脚的问题，比如分析文本和修改文本（别忘了“万般皆文本”）。这个时候他们会发现有 awk 和 sed。也只需要写几行脚本就能将这些问题处理得很好。而且它们也跟 shell 配合得天衣无缝。或许这个时候会觉得加入了 awk 和 sed 的 shell 脚本有些难看。没关系，还有 Perl 和 TCL。Perl 天生就是为处理文本而存在的，TCL 也不含糊。

如果觉得这些语言都太老气了，有些过时了。不要紧，还有 Python、Ruby 等这些现代脚本语言。它们除了不能写操作系统内核之外，几乎什么都能干，而且还是面向对象的。

不管怎样，在 Linux 下能够选择的脚本语言都是极其丰富的。它们最大的特点就是简单、好学且资料丰富。简单就意味着容易维护，好学就容易吸引用户，资料丰富就不会在解决 bug 上出现障碍。即便是专业的程序员，也会因为这些特点而特别偏好脚本语言。导致的一个结果就是脚本程序在 Linux 中的大爆发。

4.3.2 为什么不是 C

C 语言是 Unix 的母语，这是毋庸置疑的。前面也说过，正是因为有了 C 语言，才使得 Unix 有了今天的成就。但为什么在 Linux 中有这么多程序，甚至是关键程序，不是用 C 语言编写的呢？

脚本程序由于是解释执行的，在执行效率上自然是会有很大损失的。并且大家都知道，C 语言所编写的程序又是以效率著称的。但是 C 语言是一种编译型语言。要想让 C 语言的程序能够运行，必须经过编译和链接这两个步骤。能够将由几十个源代码文件构成的 C 语言程序，有条不紊地编译完成并能最终链接成一个可执行程序，本身就是一件费时又费力的事情。如果一旦程序有问题，还必须使用专门的调试工具一点点地去跟踪判断，修正之后再重复那些复杂的编译和链接步骤，这又是一个极需技巧的事情。积累并掌握技巧又是一件费时又费力的事情。在早些年，计算机性能不佳的时候，这些付出或许是值得的。但是放到现在，处理器的速度至少快了几千倍，内存大了几千倍，硬盘甚至大了几万倍，而价格却更低了。从经济角度分析，机器的时间成本早已远远低于人的时间成本了。那么 C 语言在机器效率上的优势根本没有任何意义。脚本程序能够给人节省下来的时间成本，则更具经济效益。其实要论机器效率，汇编语言比 C 语言要好上几十倍，但是目前还有谁在用汇编语言编程呢？

C 语言在设计的时候，最主要的一个目标就是能够让程序员自己处理内存管理的问题。这使得 C 语言很强大但又太过于灵活，导致了很多陷阱的出现。稍微一不注意，程序中就会存在难以发觉的 Bug，甚至是严重的安全漏洞。程序员们大多是要以时间或失败为代价去积累经验，才能尽量避免这些问题的发生。而且效率在大多数应用中根本就不是问题，首要的是正确。脚本程序的简单和直观正是正确的起点，C 语言的灵活却是错误的根源。

但是 C 语言并不是一无是处，也是 Unix 的精华。C 语言作为通用程序设计语言是所向无敌的。C 语言本身也非常简洁和紧凑，资料丰富且容易学习。C 语言之后的少数语言设计，为了不被 C 语言所吞并，不得不进行大的改动，比如引进垃圾回收机制等，以和 C 语言能够在功能上保持足够距离。也正是因为这样，C 语言始终没有消失，只是它的光辉在 Linux 中稍稍地被脚本程序所遮挡了一下。

4.3.3 脚本的不足和混合编程

虽然效率并不是脚本程序的缺点，但是种类过于丰富却是一个极大不足。编写一个复杂的应用，往往很难使用一种脚本语言包杆到底。因为脚本语言都有自己适用的场景。为了能够快速有效地完成某个应用，就需扬长避短，利用多种脚本语言混合编程。

多脚本语言的混合编程是一种知识密集型的编程方法，但不是编码密集型的（这是能够

被普遍接受的原因）。为了能够良好地使用这种方法，就要求程序员不仅仅要具备相当数量的多种语言知识，还必须具备能够判断这些语言的适用场景、以及如何将它们有效地组合在一起的经验。

实际上混合编程并不是脚本语言的专利，任何编程语言都行，只要你能找准那些语言的特点。比如我就曾经使用过 Basic 和 C 进行混合编程，去完成一个 DOS 版万年历程序。为了支持鼠标点击操作，用 C 完成了鼠标中断的处理。余下的部分都用 Basic 来完成。

在 Linux 中大量应用脚本程序的场景，好多都是这种混合编程的典范。比如 Linux 的启动过程。主程序 init 是用 C 语言写的，具体到启动流程的各个环节则是 shell 脚本程序。这也就引出了脚本程序的一个主要用途……

4.3.4 强力胶水

一般来说，程序的设计是有两个方向可以选择的。一个是自底向上的设计，一个是自顶向下的设计。

自底向上的设计，就是从程序要完成的功能出发，从确定要进行的具体操作开始，向上进行。比如，设计一个网页浏览器，一些基本操作原型需要包括 HTML 解释器，JavaScript 解释器、网络协议处理等。然后将这些功能串起来，构成一个浏览器。

自顶向下的设计，就是从程序的规格说明或应用逻辑开始，向下进行。比如，同样的网页浏览器，这个时候要先确定功能：支持什么类型的 URL（http:、ftp:还是 file:），支持何种规范的 HTML，是否提供 JavaScript 的支持等。然后一一去实现这些功能。

具体采用何种设计方法，是一个非常重要的问题。因为对应层次的功能实现很可能会受到最初选择的限制。尤其是采用自顶向下设计时，应用逻辑所需要的操作可能根本无法实现。然而，如果采用自底向上的设计，很可能做了很多与应用逻辑无关的操作——或者，你想造房子，最后你造出了一堆砖头。

在目前的大多数实践中，很多人采用的是被认为“正确的自顶向下”方法。如果能够精确的定义程序的需求，且在实现中不会有需求上的变化，而且保证能够实现，那么这的确是一个好方法。但别忘了我说的是如果，事实的情况是没有如果。因为经常遇到的情况是需求很难精确定义，从而导致需求还没有来得及实现就发生了变化。而且很多需求都只是用户的臆想，受制于其他需求而根本无法实现。

为了应对这种两难的问题，程序员的选择就是双管齐下——一边以自顶向下的方式规划具体程序的应用逻辑，一边采用自底向上的方法去封装常用操作原型而形成程序库。这样，即便当需求发生变化时，程序库仍然可以重用。所以大多数情况下，我们经常使用的程序就是这两种方法所结合的产物，这就导致了“胶合层”的出现。因为自顶向下和自底向上是两种容易产生冲突的方法。顶层的应用逻辑和底层的常用操作原型必须用胶合逻辑进行适配。

其实胶合层是一个非常讨厌的东西，必须尽可能的“薄”。这一点非常重要。胶合层用来将东西粘在一起，但不应该用来隐藏各层的裂痕和不平整。脚本语言因其本身的简单和功能限制，在胶合层的实现时，充当了“强力胶水”的角色。

在网页浏览器的例子中，最为著名的应属 Mozilla 的设计。图 4.1 展示了 Mozilla 的基本架构。

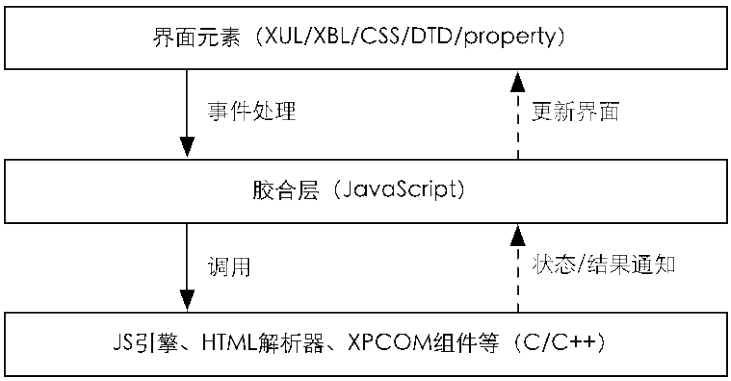

图4.1 Mozilla基本架构

Mozilla 作为网页浏览器，复杂程度是不亚于操作系统的，它由数百万行的 C++代码构成。这样复杂的软件能够被很好的维护起来，最根本的原因就是架构的优良所带来的结果。主要体现在：1.分离界面和实现；2.针对接口编程；3.分层设计；4.容易扩展。能够将这 4 个方面真切地付诸于实际，胶合层功不可没。从一个包含 HTML 标签、CSS、JavaScript 等内容的 html 文件到将这些内容显示为用户所见的网页效果，这个过程被称之为页面渲染。页面渲染代码因是作为浏览器最容易导致 bug 丛生的地方而臭名昭著。这种架构也使得页面渲染代码能够藉由用于胶合层的脚本语言来实现。脚本语言简单、清晰易读的特性，使得 bug 缺少了一个滋生的温床。

4.3.5 极端的例子

有一个使用胶合层更为极端的例子就是 Emacs。

或许到目前，大家已经能够略有所察觉，在 Unix 的文化里，是极其强调简约主义的。换句话说就是崇尚少吃饭多干活。然而 Emacs 却恰恰与这相反。它异常庞大，维护起来着实不易；大量使用 Lisp 脚本来胶合少得可怜的 C 程序，非常消耗系统资源。但 Emacs 能做的事儿也是异常丰富的。只有你想不到，没有它做不到。

在这个地方，忠于 Unix 文化的人们可以用两种方式来看待 Emacs 的这种设计风格。一种是完全否定它，另一种就是发展出一种考虑到复杂度的非教条的方法。我个人更推崇后者。

因为抱怨 Emacs 庞大，和把 Linux 系统中的全部 shell 脚本程序算在一起而抱怨 shell 庞大，是一样地不公平。

Emacs 完全可以视为一个围绕在小巧锐利的工具集合上的虚拟机或框架，只是这个工具集恰好是 Lisp 编写的罢了。从这个角度出发，shell 和 Emacs 的主要区别就在于 Linux 发行版并没有把所有 shell 脚本程序连同 shell 一起发布。因为 Emacs 内置了感觉臃肿的通用语言而反对 Emacs，就像因为 shell 具有条件和 for 循环而拒绝使用 shell 脚本一样无聊和可笑。毕竟使用 shell 跟会不会写 shell 脚本程序没有半毛钱关系，使用 Emacs 也没人强迫你必须会 Lisp。如果一定要对 Emacs 鸡蛋里挑骨头，那只能说由于历史过久，导致其内置工具积累过多而产生了一些问题，但是这个跟用户有什么关系呢？讨厌就不用，Emacs 也不会罢工。

基于这种观点，GNOME 和 KDE 的开发者们从来不会因自己所维护的系统过于庞大而惶惶不可终日。而且还应该籍此而总结出另外一个直观重要的概念，那就是……

4.3.6 软件的适度规模

崇尚简约是无可厚非的优良传统，但小巧未必就能万事大吉。小巧锐利的工具很难跨越的一个鸿沟就是数据共享问题，除非它们能生存在彼此之间通信便利的框架之中。非常幸运的是，Linux 系统本身就是这样一个框架，GNOME 是，KDE 是，Emacs 更是。只是为了获取这种能够对共享上下文环境进行统一管理的框架，是要以高复杂度作为代价来换取的。能够对共享上下文环境进行统一管理的最直接的益处就是，用户不再需要担负底层部件的命名和资源管理问题。

Linux 中所提供的这种框架就是前面说到的管道、IO 重定向以及 shell。整合工作全部由 shell 脚本来完成。但是对于某些大型应用软件，由于 Linux 自身的这种框架在功能上、效率上和可移植性上并不能满足它们的需求，因此它们不得不寻求自己的解决之道。比如：Emacs 很重要的一个任务是要将非常多的文本缓冲区及其辅助进程与文件系统统一在一起，这大大超出了 shell 的能力。像 Gnome 和 KDE 这种桌面环境，甚至都将 shell 本身并入了自身，而且它们最核心的 GUI 通信机制，也不是简单的管道或 IO 重定向能够满足的。

一味地追求简约、小巧锐利，那是教条主义。从这些活生生的例子所展示出的始终能够吸引着无数人对其追捧中可以看到：简约不是太简单。

但是这种共享上下文的统一管理究竟能给你带来多大益处，也是需要三思而后行的。由于需求的贪婪，赋予程序太多的设想和任务而最终失败的例子在我们日常工作中比比皆是。如果你很幸运，你的产品需要运行在 Linux 上。那么解决这个问题的最好办法就是尝试使用 Linux 自身的框架，只有实证了它不行才去考虑编写庞大的程序。绝大多数的结果就是你的程序运行良好。Linux 中那些无处不在的脚本不就是最好的证据吗？

4.4 “四大笨”之三：规律无处寻

Linux 初学者直接寻求经验丰富的老手来帮忙，是最为快捷的学习方法。但是困惑也随之而来。初学者们会发现，找不同的人在解决相同问题的时候，所采用的方法都是大相径庭的，根本找不到规律。老手们在解决同一个问题上所持的观点也非常迥异，有些甚至是截然相反。更让初学者恼火的是，不同的 Linux 发行版，在对待同一个问题的时候，也会有完全不同的实现方法。这些越发地让初学者感到恐惧，不知道 Linux 该如何学起。

需要面对这种困惑的不单单是初学者们，那些所谓“经验丰富”的老手们也大多是在为了这样那样的差别而疲于奔命。难道 Linux 是一个十分“不靠谱”的系统，解决一个问题就没有一个统一的方法吗？

答案是：这个真没有！

因为 Linux 本身的设计是一种基于机制与策略相分离的设计，又因为 Linux 极其自由的特性，任何人都可以修改、包装、发行，还不用对任何人负责，这就导致了不同的发行版在提供同一个功能的时候会有很大的实现上的差别。而且即便在相同的发行版中，不同的用户由于对知识覆盖面的不同，依然可以选择不同的方式去完成同一个功能。结果就是，从表面看，Linux 在使用上几乎没有规律可寻。那么要学好 Linux，能够轻松驾驭 Linux，还能编写出符合 Linux 传统的程序，有技巧吗？

答案是：这个可以有！

4.4.1 机制与策略

透过现象看本质，到底什么是机制，什么是策略，为什么要将机制与策略分离呢？

要搞清楚机制和策略，可以从我们生活中最实在的例子出发。比如：吃饭和睡觉。吃饭可以在家里吃、去饭店吃、去亲戚朋友家吃，可以站着吃、坐着吃、躺在沙发上边看电视边吃，怎么吃都是吃；睡觉可以在床上睡、地上睡、沙发上睡，可以躺着睡、趴着睡、蜷着睡、搂着老婆一起睡，怎么睡都是睡。那么吃饭和睡觉就是机制。至于怎么吃，怎么睡则是策略。至于怎么吃着香，怎么睡着沉，那是你自个的事儿，别人说什么都是白说。

机制和策略反映在系统设计上，就是目标功能和实现方法。目标功能是必须要做的事情，为了实现一个目标功能可以有很多种方法。具体采用那种方法往往有很多因素的考量，如易用性、扩展性、简洁性、效率等方面，至于是好是坏，用的人舒服就行。目标功能却不是这样，它很多时候必不可少。就像饭得吃，觉得睡一样，少一样你都活不了。

从机制和策略的这些差别上，我们就很容易得出一个结论——它们最好能分开，别掺合在一起。因为如果生把它们揉在一起的话，那至少会有两个负面效应：一是策略变得很死板，很难适应未来的需求变化；二是任何策略上的变化很可能会改变机制本身。这就像有人规定

你必须怎么吃饭和睡觉一样，基本上都会让你吃不香睡不好。如果能将两者剥离，就有可能在探索新策略的时候不会对机制产生什么实质性的影响。毕竟总是能吃得香睡得好，身体才能倍儿棒不是？

实现这种剥离的方法也很简单。最方便的就是将一个应用当作一个库来写，这个库包含许多能够由内嵌脚本语言驱动的服务程序，而整个应用的控制流程则用脚本来撰写。前面所讲的“两大笨”不就是朝着这个目标而努力的吗？

4.4.2 接口与引擎

要说设计一个简单应用也需要抽象出机制和策略来，那显然太过高瞻远瞩了，有点够不到边儿的感觉。但是这种设计原则依然值得学习和参考。为了能够得着边儿，就放低一些姿态，让接口与引擎分离。

这里所说的接口与面向对象中的 interface 可不是一回事。这里的接口是指程序与人的交互界面，可以是文本的，也可以图形的。至于引擎也跟汽车、摩托车之类的没什么关系，它是指一个程序的核心算法。

在 Linux 中这样的例子很常见，最为有名的就是 vi 和 Emacs 了。在文本的控制台下，vi 和 Emacs 提供了基于文本的操作界面；在图形环境下，vi 和 Emacs 又能提供基于图形的操作界面。在一些脚本语言的解释器中，这种设计也很常见，比如 Python。当没有指定具体要执行的 py 文件时，“python”命令会进入一个交互式界面；而指定具体要执行的 py 文件时，“python”就会执行对应的脚本，并且在脚本结束之后也随之退出。

这样做的最大好处就是可以减少工作量，同时又具备良好的适应能力。因为有关文本编辑或脚本解释的代码只需要写一份就行，程序如何跟人交互可以视当前环境而定。带给用户的最大感受就是这个程序很贴心，永远能满足用户的喜好。

其实这就是现如今特别流行的 MVC（模型—视图—控制器）模式。M 代表“模型”，在 Linux 世界里通常被称为“引擎”，模型包含了应用程序专属的数据结构和逻辑；V 代表“视图”，是应用程序专属数据结构和逻辑的可视化形式，视图组件一般由模型负责通知更新，并且作出相应变化；C 代表“控制器”，处理用户的请求并将它们反馈给模型。在具体实践中，视图和控制器部分的结合，往往比它们与模型的结合更为紧密，有些时候就是一体的。

在Linux中，MVC模式的应用比其他任何领域都要普遍，这主要得益于Unix中“只做一件事并做好”的优秀传统，和自身强大易用的IPC（进程间通信）机制。估计好多人在看到这个事实的时候，会非常憎恨自己没有早一点学习Linux或Unix。否则那些什么倒霉的Struts[①]

① Struts 通过采用 Java Servlet/JSP 技术，实现了基于 Java EE Web 应用的 MVC 设计模式应用框架。

或许就会出自于自己的手下，让Craig McClanahan[①]见鬼去！

4.4.3 不用重新造轮子

这年头程序员们最喜闻乐见的话题，应该就是“重用”了，也就是“不用重新造轮子”。重用，大多是指源代码的重用，于是搞出了各种面向对象的编程方法，甚至总结出了一套“设计模式”。好多教科书也都是这么教导我们的。但是好多人一想到“重用”就头晕，尤其是设计模式。这并不是说设计模式不好，只是学习并能掌握设计模式的确不是一件容易的事儿，必须花费大量的时间和精力去想象和理解，还得有足够的实践机会。要是能够将设计模式运用得炉火纯青、收放自如的话，这年头基本上就会成为被众多粉丝顶礼膜拜的技术大神。

如果真是这样，那么 Linux 下的程序员就人人都是大神。得益于 Linux 的策略与机制分离设计，“重用”这事儿就太小儿科了。而且在 Linux 下的重用，你连代码长什么样都不用管，编好的程序直接重用。如果你还没有忘记前面的两个小节（“万般皆文本”和“四处用脚本”）的话，那么在那个地方所介绍的一些技巧就是干这种事儿的。

尤其是在编写具有 GUI 界面的工具程序时更是如此。因为很多程序在 CLI 的文本界面中就已经提供了，比如查看系统进程信息的 ps 命令。如果你想编写一个 GUI 界面的类似于 Windows 的“进程管理器”这样的程序，完全不用了解 Linux 内部的运行机制和有关的系统 API，直接利用 ps 命令做一个 GUI 的外壳就成了。如果不只是查看进程，还想关闭一些占用你内存的进程，还有 kill 命令可以包装。查看磁盘使用率还有 du 命令，等等……

4.4.4 内在的支持

如果你说用 shell 做不了图形界面，那没关系。不是还有各种强大的脚本语言吗？比如 Python，做个 GUI 完全不成问题。如果你觉得用 Python 做 GUI 有点怪，那不是还有 C，甚至 C++呢吗？

永远不要忘记 Linux 是一个支持多进程的操作系统。在 Linux 中创建进程比创建线程还容易，Linux 进程的资源开销也一点不比线程多。进程与线程更为不同的特性是，进程并不总是一个程序的并行执行分支，还可以是另外一个完全独立的程序，甚至是脚本。那么制作一个 GUI 外壳就非常简单了，直接利用多进程机制启动另外一个具备相关功能的程序或脚本就行了。

我想，现在在你的心中一定是有一个大大的问号的。众所周知的是：进程具有完全独立的内存和资源空间，互相不会干涉。那么两个进程如何才能像线程那样自如的交换数据，能够实现我们想要的 GUI 包装呢？

---

① Struts 之父，前 Sun 公司的杰出科学家。

别忘了，前面就提到过，Linux 自身就提供了一套强大易用的 IPC 机制。这使得在 Linux 中，两个进程进行数据交换是极其容易的事情，甚至比线程那种“资源共享同步”还要方便。

Linux 提供的 IPC 机制主要有：信号、管道、IO 重定向、共享内存、套接字等多种类型。

这其中信号相对较弱，只能提供一个字长的信息，这就相当于两个人平时各干各的，有事儿时吼一声。这种方式虽然即简陋又简单，但是效果有时非常好。

比较常用的是管道和 IO 重定向，前面几个小节就已经多次提到过了。管道只提供单向通信能力，即一侧输入一侧输出。在具体编程中，管道经常与 IO 重定向结合使用，最常用的技巧就是将一个进程的标准输出重定向给管道的输入端，另外一个进程就能从管道的输出端获取前者的输出结果。

共享内存很适合大块数据的共享，但是否必须要用，应做多方面的考量，因为经常会引起竞争。解决这种竞争不但麻烦，还到处都是陷阱，一不小心就容易犯错误。不到万不得已不建议使用。

套接字可以说是最为强大的灵活的 IPC 机制了。原本套接字是用于网络通信的，但是那本身就是一种不同进程间的通信，只是这两个进程不在一个机器上罢了。既然不在一个机器上都能通信，那么在一个机器上的自然也不在话下了。

有这么多 IPC 机制可以选择，总是会有一种适合于你的。而且这也提醒你，如果你的程序规模连做“接口和引擎分离”都觉得有些小题大做，那么就可以考虑支持一种 IPC，或许在不久的将来就有人能够用得上。其实，只要将结果输出到屏幕上（代表标准输出），那就已经支持了。

4.4.5 沉默是金，吝啬是银

如果你从来没想过要去了解什么 IPC 机制，或者对这些完全不敢兴趣，只是想写程序。那么也不要着急上来就是 GUI，而且也不要搞什么交互式操作，更不要有多么复杂 NB 的屏幕输出提示。换句话说，就是先让程序的核心功能运行起来，多简陋都行。在 Linux 下一直流行一句名言：“沉默是金，吝啬是银。”

不让程序有多么复杂 NB 的输出提示，说起来还是有一些历史原因的。遥想 Unix 诞生之初，显示器可是金贵的奢侈品。那个时候程序要输出点什么可见的东西都是通过打印机打印出来的。首先就是不环保，其次就是慢得要死，输出多了肯定遭人骂。虽然现在 CRT 显示器的价格可能比同等重量的切糕便宜不少，但是这一传统还是有很强的实用意义的。最典型的就是当有人将你的程序的屏幕输出重定向给另外一个程序的输入时，那些看似复杂 NB 的输出，好多时候基本上是没用的。好一点的情况是付出一点时间来找出有用的，最坏的情况会导致根本没法用，不幸的是最坏的情况总是会发生。一旦这样，你的程序恐怕只能是自娱自乐的了。少说话，多办事，永远是有好处的。沉默是金。

吝啬是银就是指不要轻易写大程序。估计大多数程序员在写 GUI 程序时，都会有类似经验，那就是 GUI 代码占据了程序 95%的代码，而且跟程序本身的功能还没多大关系，bug 还经常出现在这个地方。在任何时候，能够尽快将没有问题的程序交给用户，都是最明智的决定。即便他们会抱怨使用起来不方便，但毕竟他们已经开始用了。至于好用不好用、方便不方便的问题，可以接下来协商解决。前面说不要搞什么交互式操作，也是基于这个因素。因为你最初设计的交互式操作，用户也未必买账，而且还延误了你的开发时间，怎么算都是不划算的。

一旦你的程序做到了即沉默又吝啬，那么接下来的问题就好办了，除非核心功能上有毛病。因为这个时候你已经做到了“机制与策略”相分离了。用户喜欢什么样的操作方式，给他们提供就是了。或者用户自己来增加新的操作方式，也是没什么问题的。

4.5 “四大笨”之四：配置乱生根

在学习 Linux 的时候，让人受不了的不单单是“规律无处寻”，还得去学习和掌握各式各样的配置文件。即便配置文件可以不用关心，环境变量也总是出来捣乱，让某些程序的行为“异常诡异”。即使这些你都想视而不见，只是想敲敲命令，鼓捣点你认为比较浅显的东西，各种命令的命令行选项也会绕得你头晕目眩。从此大叫一声：Linux 咋就这么难学？

这与前面刚说过的“规律无处寻”差不多，是导致初学者十分困惑的另外一个“笨”现象——配置乱生根。不过与“规律无处寻”所不同的是，后者还能有一个抓住本质的“机制”。要严格说起来，“配置”也算是一种机制。但是你要说“配置”是一种策略，也不见得不恰当。所以，“配置”跟其他某些具体的“机制”相比，就要显得飘渺了一些。

4.5.1 什么是不可配的

如果你要问 Unix：“你有哪些东西能够配置？”Unix 会非常无谓地答道：“一切！”

是的，就是一切。这源于“策略与机制相分离”的这种十分优良的传统：只要有可能，就建立机制并把策略的决定权交给用户。它所带来的最直接的收益就是，按照这种方式所开发出来的程序往往功能十分强大且非常灵活。但负面影响也不可谓不巨。虽然 Unix 真正的专家用户们用起来非常顺手，但是由于程序接口选项众多，配置文件像杂草一样随处生根疯长，也彻底打击了初学者和普通用户。

那么如果这个问题是问 Linux 的，Linux 的回答就要比 Unix 谨慎很多：“需要的时候。”很多时候，这不能不说是 Linux 的一个进步。毕竟一味的追求传统就是教条主意的固步自封了。

其实 Linux 上的大多数程序都是源自于 Unix 的，所以很多时候你并不会发现这种改变的设计倾向。不过到目前为止，经过被称为“新学派 Unix 程序”的那些 Linux 开发者们的

不断努力，已经有了很显著的变化。人们已经习惯，在设计一个程序时总是会考虑：哪些不应该可以配置！

首先，对于能够可靠地进行自动检测的东西，就是不应该可以配置的。这是最容易犯的错误。比如在做并行运算开发的时候，经验告诉我们，并行数量至少应该与电脑的计算核心数量相匹配才好。好多程序员为了让这种程序具备普适性，提供了一个并行数量的配置，让使用者根据自己电脑的实际计算核心数量进行配置。但实际情况是很多用户从不去理会这个配置项，总是导致无法完全发挥程序的并行计算能力。而当你在文档中或口头培训中提及这个配置时，用户们的眼中浮现出来的永远是困惑。显然在程序中自动监测电脑有多少个计算核心要比提供一个配置项，对于用户来说要省事很多。而且检测电脑有多少个计算核心又不是什么困难的事情。

其次，用户不应该看到优化开关。让程序高效准确的运行永远是程序员的事情，跟用户没有半毛钱关系。最常见的一个反例就经常出现在一些视频格式的转化软件中。它们经常要求用户提供采用何种编码方案、多少码率等配置信息。其实用户要的就是转化成 iPad，或者 PS3 能播放的格式。iPad 只需要 1024×768 的分辨率就够了，至于 PS3，很多人是想体验 1080p 全高清的震撼的。有多少用户能够知道 iPad 或 PS3 应该采用何种编码方案，多少码率才能体验全高清的震撼呢？

最后，能用脚本胶合或简单管道实现的任务，就不需要配置。因为能够简单利用其他程序来完成的任务，就没有必要增加本程序的复杂度。一个比较正面的例子就是“ls”命令，它永远不会提供分页输出的命令选项。因为只需要结合“more”或“less”命令就能够提供这样的功能。

其实在 Linux 的这些“笨”中，只有“配置乱生根”是最为让人迷惑和诟病的特性。好处有、害处也不少。但并不能因为害处多就全盘予以否定，毕竟因它所能够带来的好处而获得的实际收益要比因害处所带来的实际损失要多很多。十全十美的东西是没有的，学习 Linux 的同学们，在这个地方就忍一忍吧-*_^-!

4.5.2 配置三元素

本节的最开始就说过的配置文件、环境变量、命令行选项，这些就是起配置作用的三元素。

由于 Linux 是一个多用户的操作系统，那么就必然会涉及针对不同用户的配置问题，必须要引出系统级别和用户级别配置的差别。所以如果要详细划分，可以划分为系统配置文件、系统环境变量、用户配置文件、用户环境变量和命令行选项。注意顺序问题，凡是跟系统有关的都是最高优先级、跟用户有关的都是低优先级，文件优先于变量，命令行选项优先级别最低。程序在查询配置信息时，都是要按照这种优先级排序来确认的。这样做的一个好处是，最后被确认的配置信息永远能够覆盖到最先确认的配置信息，最先确认的配置信息也可以帮

助后面要确认的信息确定位置。由此可见，配置的响应优先级就完全是逆序的了。这与我们常规的理解是不矛盾的，毕竟通过命令行给定的配置信息，是用户最想获得的特性。

在考虑使用何种机制向程序提供配置信息时，要牢记这样的原则：调用时可能发生变化的配置信息，使用命令行选项；改动很少但确实应该由各个用户控制的配置信息，使用用户配置文件或用户环境变量；需要由系统管理员设置而不需要用户改变的整体系统级选项数据，应该使用系统配置文件或系统环境变量。

4.5.3 配置文件

配置文件比较文绉绉的称呼是“运行控制文件”，存放与具体程序相关的声明信息，有些时候甚至是可执行的命令，在程序启动时解析。

对于系统级配置文件，就像在第三章中描述的那样，应该放在/etc 目录下。对于用户配置文件，应该放置在用户的“home”目录下，并且一般是隐藏文件。由于 Linux 下隐藏文件是以“.”开头的，所以这类配置文件也被称为“点文件”。

如果一个程序的配置信息很多，那么它也可以拥有一个配置目录或点目录。每个目录应该包含数个与同一个程序相关的配置文件。多个程序共用一个配置目录或点目录，很难保证不会出问题。

配置文件或配置目录的命名方式没有严格规定，基本上是保持与其所负责的程序的名称一致即可。一些较为古老的程序会使用一些较为古老的约定：使用可执行文件名后加“rc”后缀的方式（rc 代表运行控制）。比如/etc/bashrc 和.bashrc，前者是 Bash 的系统配置文件，后者是 Bash 的用户配置文件。更有的一些甚至就是以 rc 为名的，比如/etc/rc.d 目录，大多数 Linux 发行版本都将它作为 init 程序的配置目录。

配置文件的内容同样没有严格规定，不过也有一些约定俗成的规则可供参考。比如在“万般皆文本”一节中提到的 DSV 风格的文本内容。如果配置的程序是某种语言的解释器，那么它的内容一般会使用具体语言本身。最为著名的例子就是 shell 本身和 Emacs。前者的 bashrc 文件实际上就是 shell 脚本程序，后者的.emacs 也是用 Lisp 编写的。当然，为了避免用户为了编写配置文件而不得不去过多地学习新的知识，最好的办法就是能够提供一个全方位的例子配置文件，并有相关文档说明，让用户能够容易地按照例子删减。优秀的云计算平台 Hadoop 就是一个非常好的例子。

4.5.4 环境变量

当Linux程序启动时，它的运行环境会包含一组名字和值的关联（名字和值都是字符串）。

有些是由用户手工设置的，有些是由系统在登录时设置的，有些是由shell或虚拟终端[①]设置的。这就是环境变量。在Linux下，环境变量一般会携带文件搜索路径、系统默认值、当前用户ID和进程ID等信息，以及其他有关程序运行时环境的关键信息。

环境变量的读取是非常容易的。在 C 和 C++中，环境变量的值可以通过库函数 getenv 获得。Perl 和 Python 在启动时，会初始化环境变量字典对象。其他语言通常也都采用以上两种方式之一。常见的系统环境变量见表 4-1 的描述，注意大小写。

表 4-1　常见系统环境变量

变量名	用　　途
USER	当前会话登录的账户名（BSD 约定）
LOGNAME	当前会话登录的账户名（System V 约定）
HOME	当前会话的用户 home 目录，这个变量非常重要，很多程序要通过这个环境变量来了解自己的用户配置文件的位置
COLUMNS	文本控制台或虚拟终端窗口以字符为单位的列数，通过这个环境变量，CLI 的程序可以了解一行内最多能输出多少字符
LINES	文本控制台或虚拟终端窗口以字符为单位的行数，通过这个环境变量，CLI 的程序可以了解屏幕上最多可以显示多少行文本
SHELL	当前使用的 shell 的名字
PATH	shell 搜索可执行程序的目录列表
TERM	当前会话控制台或虚拟终端的名称。最大的用途就是程序能够知道可以使用哪些特性，比如是否可以输出中文等

需要注意的是，当程序是用 shell 以外的方式启动时，有些系统环境变量甚至是全部系统环境变量都还未设置。特别是那些守护进程，这些系统环境变量都还未设置——即使设置了，那些值也未必有意义。这类程序，尽量不要使用环境变量。另外，当环境变量有多个值的时候，需要使用冒号“:”作为分割，PATH 变量就是最典型的例子。

其实环境变量有时候是很难区分什么是系统的，什么是用户的。如果一定要严格划分，那么任何用户都能访问到的就算是系统环境变量，但是值对于每个用户都可能不同，比如HOME。相反，如果只有某个具体的用户才能访问到的，就应该算是用户环境变量，毕竟那是因具体的某个用户而存在的。

尽管应用程序可以在系统定义的范围之外自由解释环境变量，但是这样的做法在 Linux 中是很少见的。而且环境变量也不适合把结构化信息作为值传递到程序中（虽然原则上可行）。所以，大多数情况都是直接使用现有的环境变量，标新立异的设计一个新的环境变量

① 第 10 章 生死与共的“兄弟”有较为详细的介绍。

大多不是明智之举。应该优先考虑配置文件。

4.5.5 命令行选项

在 Linux 中，几乎所有程序都会提供几个命令行选项。这样做的一个好处是程序的配置信息可以由脚本指定，这对于作为管道或过滤器的程序尤其重要。

有三种约定可以区分命令行选项和普通的参数：原始的 Unix 风格、GNU 风格和 X tookit 风格。

原始的 Unix 风格命令行选项，是以连字符“-”开头的单个字符。如果选项后面不带参数，则被称之为模式选项，模式选项是可以组合在一起使用的。例如，如果-a 和-b 是模式选项，那么-ab 或-ba 就都正确，而且会启用这两个选项。如果选项有参数，这些参数要紧接在选项后面（是否以空格分隔可选）。

GNU 风格则使用两个连续的连字符“--”后接选项关键字（注意，不是单个字符）。这种风格是因为有好多程序过于复杂，导致单个字符不够用了而发展起来的一种治标不治本的方法。较原始的 Unix 风格更容易让人理解，但作为我们这种非英语为母语的同胞们也经常输入错误或记不住。GNU 风格的选项不用空格分隔就不能组合使用。选项参数既可以用空格分隔也可以使用单个等号“=”来分隔。

最让人痛恨的恐怕应该是 X toolkit 风格了。它使用单连字符和选项关键字，并且由 X toolkit 进行解析。最要命的是，X toolkit 先要过滤并处理某些特别的选项，比如-geometry 和-display，然后再把过滤好的命令行传递给应用程序去解析。如果你不清楚它会过滤哪些选项，就会死活都找不出你的程序为什么接收不到某些选项。所以，这种东西最好别碰它。

在表 4-2 中我列举了最常用的 Unix 风格命令行选项的含义，从 a 到 z 都覆盖到了。如果你的程序需要提供某些方面的配置选项，那么可以直接使用。如果遇到冲突了，那么最好采用 GNU 风格。因为这样，可以使你的程序非常通用，也容易理解。当然，这对初学者也是有帮助的，因为 Linux 下大多数程序也是按照表中的约定去做的。

表 4-2 常用的 Unix 风格命令行选项的含义

变量名	用 途
-a	所有选项（all），不带参数；或添加（append），这个时候与-d 相对应
-b	缓冲区（buffer）或数据块（block）大小，带参数； 批处理（batch），不带参数
-c	命令（command），带参数；检查（check），不带参数

（续）

变量名	用 途

-d	调试（debug），带或不带参数；带参数指定调试信息级别，这个非常普遍； 偶尔具有删除（delete）或目录（directory）的含义
-D	定义（define），带参数。比如 C 编译器的宏定义；
-e	执行（execute），带参数；编辑（edit），不带参数； 偶尔具有排除（exclude）或扩展（extend）的含义
-f	文件（file），带参数，表示文件输入；强制（force），多数不带参数
-g	GDB 调试信息，带或不带参数。主要用于 GCC 来生成 GDB 的特有调试信息
-h	表头（header），通常不带参数。启用、禁用或修改程序生成报表的表头； 帮助（help），或许这是最普遍的理解
-i	初始化（initiallize）或交互（interactive），通常不带参数
-I	包含（include），带参数，比如 C 编译器的头文件搜索路径
-k	保留（keep），不带参数，禁止某个文件、信息或资源的常规删除操作； 偶尔有杀死（kill）的含义
-l	列表（list）、长模式（long）或登录（login），不带参数； 加载（load），带参数；偶尔会有代表长度（length）或锁定（lock）的含义
-m	消息（message），带参数； 偶尔具有邮件（mail）、模式（mode）和修改（modify）的含义
-n	数字（number），带参数；否（not），不带参数
-o	输出（output），带参数
-p	端口（port），带参数；协议（protocol），带参数
-q	安静（quite），不带参数。禁止正常的结果输出或诊断输出
-r 或-R	递归（recurse）或反向（reverse），不带参数
-s	缄默（silent）,不带参数。与-q 类似，如果两者都支持，-q 表示“安静”，-s 表示“绝对缄默”；主题（subject），带参数；偶尔具有大小（size）的含义
-t	标记（tag），带参数
-u	用户（user），带参数
-v	冗长（verbose），带或不带参数；版本（version），不带参数
-V	版本（version），不带参数，比-v 常见
-w	宽度（width），带参数；警告（warning），不带参数
-x	启用调试，同-d 类似，带或不带参数；提取（extract），带参数
-y	是（yes），不带参数
-z	启用压缩，不带参数

4.6 什么样的文化

这是一个什么样的传统文化和设计哲学呢？可以用“简单”和“傻 x”来概括，也就是 Keep It Simple and Stupid，简称 KISS。这是所有 Unix 以及 Linux 这样的类 Unix 操作系统的传统文化和设计哲学。这里最重要的就是傻 x，因为往往傻 x 和精明就是一念之间的事儿。

4.6.1 “傻 x”的精明

能够说明这种 Unix 传统文化和设计哲学“傻 x”的地方有很多，比如：文件就是简单的纯文本文件，在字节层面再无结构可言，处理这种文件有时费时又费力；文件一旦删除就再也无法恢复，对于用户的一些误操作没有任何补救措施；安全模型过于原始，使用起来复杂难耐，只有一个权限超大的 root 用户，一处被攻击则意味着全盘沦陷；等等这类。其他的选择不是没有，可是基本上都没什么好下场。比如：试图想改变文件无结构这个问题的方法有很多，但现在却没有几个人能记起它们都是什么，即便是如今最先进的 Hadoop 分布式文件系统中，推荐的文件类型依然是纯文本文件；文件删了就没法恢复，很多人觉得这个设计有些欠妥，但是当使用能够恢复文件的文件系统而导致自己的艳照漫天飞时，又憎恨为什么自己删不掉它们；虽说安全模型过于原始，可是那些改变策略的家伙却一直被病毒和流氓软件所困扰，而坚持采用这种安全模型的 Linux，纵使已经非常广泛地被人们所应用，却很少有人担心自己的系统被病毒或流氓软件入侵。

其实，这些都不是最主要的争论点。最主要的，而且是旷日持久的争论，恰恰是这种设计哲学的最重要的一个特性——提供一套“机制，而不是策略”。基于这种设计哲学而产生的一个设计倾向就是：行为的最终逻辑被尽可能推后到使用端。所以你会发现：在 Linux 中可以有很多种 shell 供用户选择；图形界面也有很多种窗口管理器供用户选择；甚至重要的系统初始化过程，在不同的发行版都有不同的策略和配置文件集。每一个 Linux 用户都可以有一个与众不同的 Linux。这一切都让初学者那么眼花缭乱，最终不知所措。因此，这种设计哲学也导致了 Linux 等这些类 Unix 系统丧失了大量初级用户，经常被人嘲笑它们的“傻”。

然而，如果只看眼前的话，这种自由放纵还有点“傻 x”的传统文化和设计哲学，的确会让所有采用这一哲学而设计的系统丧失大量初级用户。但是从长远考虑，最终你会发觉这个“傻 x”换来了至关重要的优势：策略相对短寿，而机制才是长命百岁。那些标榜着提供简洁性操作的设计常常会走进明日进化的死胡同。

Linux 爆炸式的发展也正是受益于将这种“傻 x”式的设计哲学发挥到极致的结果所致。加之互联网技术重要性的渐增，开源运动的兴起，都给了我们充足的理由来否定对这种设计哲学的怀疑。虽然“开源”这个术语和开源的定义一直被人们误认为是随性懒散的，甚至有一些“傻 x”的嫌疑。但是自由共享源码的同僚严格复审的开发方式也是打从 Unix 文化的

诞生之日起就拥有的最具特色的部分。这也就是 Linux 内核一直采用基于互联网这种松散式开发方式，而却没有导致自身分裂或者凌乱不堪的根本所在，更没人敢嘲笑 Linux 内核的开发者们都是“傻 x”。开源的一个重要好处就是将弱点暴露给世人，而不是成为那些所谓的核心机密。这吸引了大量 hacker 的涌入。能够修复一个软件的漏洞被这群人认为是好玩儿的。好玩儿则代表了趣味性，而趣味性又是峰值效率的标志。充满痛苦的工作只会浪费劳动力和耗尽创造性。这导致的一个结果就是：高质量的开源开发工具在这个星球上已经极其丰富，开源的应用程序也已经达到甚至超过了它们专属同侪的高度。这些开发工具和应用程序可以自由修改、重用和再造，又节省了大多数软件开发者 90%的工作量。显然只要花 10%的努力就能够完成 100%的工作又是一件好玩儿的事情。这就像野火一样，无法控制地蔓延开来，使得开源软件的力量越来越大，大到人类已经无法阻止了。况且一直有一个地方是被人们所忽视的，那就是没有任何人说过开源的软件一定是免费的。到了这份田地，还有人说开源者们“傻 x”吗？

4.6.2 “简单”不简单

虽然“傻 x”是这种文化和设计哲学的重要部分，但是也不能忽略“简单”这一本质特性。你可以说是因为过于简单而使得它“傻 x”，也可以说是因为“傻 x”导致了简单。反正既然说的是哲学，你就得用哲学的思想去思考。

简单使得更加专注。最显著的特征就是我们熟知的 Linux 系统上的每一个工具软件大都只会做一件事情，其他类 Unix 系统也是如此。开发者可以足够专注地将这一件事情做到极致。这样所产生的一个现象就是：很少有人去尝试替换这个工具软件，实际上也没有人能够做到。除非这个工具软件要做的事情没人去做了，但是这种事情发生的概率又很少。对知识的投资极具性价比！

简单使得系统更具普适性。比如 Linux 系统中大多数文件都是纯文本文件，各种程序之间的通信方式也是文本流。因为所谓文本就是没有特定格式的简单字符序列。好处就是人们不需要专门的工具就可以很容易地读写和编辑。在编写和调试程序的时候，可以非常容易地察觉输入和输出的变化。文件可以长期保存，永远不会因为格式的过期而导致没有合适的查看工具。每个需要与其他程序通信的程序不用假设对方的通信协议，也不用限制自己的通信协议。这样，即便面对未来的程序也不会发生问题。

简单容易让人掌握。Linux 系统和其他类 Unix 系统所提供的 API（应用编程接口），大多数就是 C 的标准库函数。有过 C 语言开发经验的程序员，就可以直接在这些系统上进行开发，不需要特别学习。这样带来的好处是显而易见的，即所有采用标准 C 所编写的程序可以不经修改地移植到这些系统上；在这些系统上所开发的软件也可以很容易地移植到其他支持标准 C 的系统上。这主要的原因也是因为 C 起源于 Unix，Unix API 的很大一部分成为了

C 的标准库函数。由于 C 语言的简单才使得大多数开发人员都掌握了它，从而奠定了这个坚实的基础。

简单才能灵活。按照这种设计哲学而设计的应用程序接口会尽量做到小巧，并通过提供众多的程序粘合手段，使得所有类 Unix 系统的基础工具箱的各种组件连纵开合后，将获得单个工具设计者无法想象的效果。这是彻头彻尾的灵活性。“一根筷子轻轻被折断，十根筷子牢牢把成团”这样的科学真谛，也在这里体现的淋漓尽致。

综上所述，这些都是简单所带来的好处，但并不仅限于这些，只是无法一一穷举。看似简单的东西往往是不简单的。这与我们传统的唐诗宋词是有一拼的。谁都知道五言绝句好记又好读，毕竟只有 20 个字嘛，还合辙押韵。可是现今能够流传下来的却不多。因为能写五言绝句的都是大师级的人物，能够写好的都是大师中的大师。虽然只有 20 个字，却能表达出用 2 万字都无法描摹到位的意境，难度是绝对的不同凡响的。可是大师就那么几个，你不能指望他们什么都不干天天写这个不是？

正如五言绝句是中华民族传统文化的精粹那样，“简单”是 Unix 文化的精粹。不同的是，五言绝句需要大师写，而 Unix 的“简单”加上它的“傻 x”却塑造了一批大师。更有甚者，Linux 继承了 Unix 的传统文化和设计哲学之后，又与互联网文化相融合，使得我们人人都能成为大师！

4.7 这一切的基础大师的阐释

大道理讲过千百遍它依然是大道理，一点都不实际。比较实际的东西就是：Unix 文化并没有什么高深完美的科学理论作为依托。它跟其他工程领域一样，植根于丰富的实践经验，是经过不断的总结和融合而逐步形成的。它是自下而上的，不是自上而下的，更注重于实用性。你很难在所谓正规的方法学或标准中找到它们。它属于那种出自于人们本能的隐性知识，换句话说就是所谓的专业经验。它鼓励那种分轻重缓急和怀疑一切的态度，并促使你以幽默达观的心态对待这些。

Unix 管道的发明人、Unix 传统的奠基人之一 Doug McIlroy 从某些侧面对 Unix 文化做了一些阐释：

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

（2）假定每个程序的输出都会成为另一个程序的输入，即便那个程序目前可能还不存在。输出中不要有不相关的信息出现，这会形成干扰。避免使用严格的分栏格式或二进制格式输入。尽量避免使用交互式输入。

（3）尽可能早地将设计和编译的软件投入试用，即使是操作系统也不例外，理想情况下，应该在几个星期内。拙劣的代码不值得珍惜，扔掉重写。

（4）优先使用工具而不是晦涩的帮助文档来减轻编程任务的负担。工欲善其事，必先利其器。

Rob Pike，最伟大的 C 语言大师之一，在 *Notes on Programming in C* 中则从编程的角度对 Unix 文化进行了阐释，这也反映了一些侧面：

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

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

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

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

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

原则 6： 没有原则 6。

这些或许还不够全面。于是，Eric S. Raymond 大师，在他的著作《Unix 编程艺术》一书中说道：“Unix 哲学中更多的内容不是这些先哲们口头表述出来的，而是由他们所作的一切和 Unix 本身所做出的榜样体现出来的。”并且做了这样的总结：

- 模块原则：使用简单的接口拼合简单的部件。
- 清晰原则：清晰胜于机巧。
- 组合原则：设计时考虑拼接组合。
- 分离原则：策略同机制分离，接口同引擎分离。
- 简洁原则：设计要简洁，复杂度能低则低。
- 吝啬原则：除非确无它法，不要编写庞大的程序。
- 透明性原则：设计要可见，以便审查和调试。
- 健壮原则：健壮源于透明与简洁。
- 表示原则：把知识叠入数据以求逻辑质朴而健壮。
- 通俗原则：接口设计避免标新立异。
- 缄默原则：如果一个程序没什么好说的，就沉默。
- 补救原则：出现异常时，马上退出并给出足够的错误信息。
- 经济原则：宁化机器一分，不花程序员一秒。
- 生成原则：避免手工 hack，尽量编写程序去生成程序。
- 优化原则：雕琢前先要有原型，跑之前先学会走。
- 多样原则：决不相信所谓“不二法门”的断言。
- 扩展原则：设计着眼未来，未来总比预想来得快。

一共 17 条原则，我喜欢称它们为“ESR的 17 条”，它涉及Unix文化的方方面面，可以说是事无巨细了。很多讲述软件工程的书籍或文章都会或多或少的提及若干，甚至大多数操作系统在设计和实现上也会或多或少的有所体现。但它们始终没能将这些原则自始至终的贯彻到底，从而也导致了大部分像我这样的苦逼程序员一直被蹩脚的工具、糟糕的设计、过度的劳作和臃肿的代码所困扰，而且还奇迹般地形成了习惯，总给人展现出那种“沃德田·诺·布雷斯·博苏富斯基”[①]的酷。

① 就是“我的天哪，不累死不舒服”的谐音。