UNIX 系统编程

需要说明的话：

1、关于 UNIX：3 方面内容

2、第一次开这门课

2、你为什么选这门课

3、知识的投资

4、参考书：UNIX 系统编程，Linux 程序设计，以前者为主

第一章 预备知识

1、UNIX 发展历史

第一阶段：诞生

Unix 的诞生和 Multics（Multiplexed Information and Computing System）是有一定渊源的。Multics 是由麻省理工学院，AT&T 贝尔实验室和通用电气合作进行的操作系统项目，被设计运行在 GE-645 大型主机上，但是由于整个目标过于庞大，糅合了太多的特性，Multics 虽然发布了一些产品，但是性能都很低，最终以失败而告终。

AT&T 最终撤出了投入 Multics 项目的资源，其中一个开发者 Ken Thompson 则继续为 GE-645 开发软件，并最终编写了一个太空旅行游戏。经过实际运行后，他发现游戏速度很慢而且耗费昂贵——每次运行会花费 75 美元。

在 Dennis Ritchie 的帮助下，Thompson 用 PDP-7 的汇编语言重写了这个游戏，并使其在 DEC PDP-7 上运行起来。这次经历加上 Multics 项目的经验，促使 Thompson 开始了一个 DEC PDP-7 上的新操作系统项目。Thompson 和 Ritchie 领导一组开发者，开发了一个新的分时多用户操作系统。这个系统包括命令解释器和一些实用程序，这个项目被称为 UNICS（Uniplexed Information and Computing System），后来才被改为 UNIX。

UNIX 的第一功，是 1971 年为贝尔实验室的专利部门进行“文字处理”的支持工作。首个 UNIX 应用程序是 nroff（1）文本格式化程序的前身。这个项目也让他们名正言顺的购买了一台功能强大得多 PDP-11 小型机。万幸的是，当时管理层还未意识到 Thompson 和其同事所编写的字处理系统就快孵化出一个操作系统，因为贝尔实验室并没有开发操作系统的计划——AT&T 加入 Multics 联盟正是为了避免自行开发一个操作系统。UNIX 在贝尔实验室计算群落中重要而永久的地位确立。

最初的 Unix 是用汇编语言编写的，一些应用是由叫做 B 语言的解释型语言和汇编语言混合编写的。B 语言在进行系统编程时不够强大，所以 Thompson 和 Ritchie 对其进行了改造，增加了数据类型和结构，并与 1971 年共同发明了 C 语言。1973 年 Thompson 和 Ritchie 用 C 语言重写了 Unix。在当时，为了实现最高效率，系统程序都是由汇编语言编写，所以 Thompson 和 Ritchie 此举是极具大胆创新和革命意义的。用 C 语言编写的 Unix 代码简洁紧凑、易移植、易读、易修改，为此后 Unix 的发展奠定了坚实基础。Thompson 和 Ritchie 也

因此获得 1983 年度的图灵奖。

第二阶段：黑客（Hacker）

1974 年，Thompson 和 Ritchie 合作在 CACM（Communication of the ACM）上第一次公开展示了 UNIX，作者在名为“The Unix Time Sharing System”的文章中描述了 UNIX 前所未有的简洁设计，并报告了 600 多例 UNIX 应用——这些都是安装在即使按照那个年代的标准性能都很低的机器上，但是“性能的局限不仅成就了经济性，而且鼓励了设计的简约”。

CACM 论文发表后，全球各个研究实验室和大学都嚷着要亲身体验 UNIX。根据 1958 年为解决反托拉斯案例达成的和解协议，AT&T（贝尔实验室的母公司）被禁止进入计算机相关的商业领域，所以 UNIX 不能够成为一种商品。并且，根据和解协议的规定，贝尔实验室必须将非电话业务的技术许可给任何提出要求的人。Ken Thompson 开始默默回应那些请求，将磁带和磁盘一包包的寄出去。AT&T 以分发许可证的方法，对 UNIX 仅仅收取很少的费用，大学和研究机构就能获得 UNIX 源代码以进行研究。

那时候个人机还未出现，对能用得起 UNIX 的小型机的使用管制要比大型机少得多，因此，在上世纪 70 年代最早搞 UNIX 的通常都是头发蓬乱的嬉皮士和准嬉皮士们，摆弄操作系统的乐趣对他们来说不仅意味着可以在计算机学科的前沿纵情挥洒，而且在于可以挑战伴随“大计算”的所有技术假定和商业实践：COBOL、商务套装和 IBM 批处理大型机。许多大学都对 UNIX 做出过贡献。多伦多大学计算机系发明了 200dpi 的打印机/绘图仪；耶鲁大学的计算机专家和学生们改进了 shell；普渡大学工程系对 UNIX 性能作了重要改进，推出了支持大量用户的 UNIX 版本；加州大学伯克利分校的学生开发了新 shell 和许多小型实用工具；而澳大利亚的新南威尔士大学的 John Lions 写的“莱昂氏 UNIX 源代码分析”至今被 UNIX 程序员奉为圣典。

在这种从学术界到工业实验室，然后又回到学术界，最后到不断增加的商业用户的循环过程中，UNIX 不断发展，日益完善。在此期间，AT&T 发布了 UNIX 早期的几个版本，从 V1 到 V5。到了 1975 年，当 UNIX 发展到 V6 时，AT&T 认识到了 UNIX 的价值。因此 AT&T 一方面继续发展内部使用的 UNIX 版本 V7（被 UNIX 程序员认为是第一个完全意义上的 UNIX），一方面成立 UNIX 系统实验室（Unix System Lab, USL）开发对外正式发行的 UNIX 版本，同时 AT&T 宣布对 UNIX 产品拥有所有权。USL 对外发布的第一个商业版本是 System III，当时一份 System III 的源码许可证的官方价格为 4 万美元。

在 UNIX 最初的发展过程中，加州大学伯克利分校早在 1974 年就开始了对 UNIX 的研究，而 Ken Thompson 利用 1975~1976 的年休在此教学，更对 UNIX 的研究注入了强劲的活力。1977 年（AT&T 的 UNIX V6 发布后不久），当时还默默无闻的伯克利毕业生 Bill Joy 工作的加州大学伯克利分校计算机系统研究小组(CSRG, Computer Science Research Group)发布了第一版的 BSD（Berkeley Software Distribution），从此伯克利的 BSD 和 AT&T 的 UNIX System 就成 UNIX 发展的两大阵营，前者属于学院派，而后者则是商业中的代表。

第三阶段：内战

BSD UNIX 在 UNIX 的历史发展中具有相当大的影响力，被很多商业厂家采用，成为很多商用 UNIX 的基础,而 AT&T 与其同时存在的 UNIX 版本的影响就小得多。同时很多研究项目也是以 BSD UNIX 为研究系统，例如 1980 年美国国防部高级研究计划局（DAPRA，Defense Advanced Research Projects Agency）需要在 UNIX 环境下的 VAX 机上实现全新的 TCP/IP 协议栈时，就选择了伯克利 UNIX 作为平台，这份合同无疑成为 UNIX 自诞生以来最关键的转折点。

而 AT&T 的 UNIX 系统实验室，同时也在不断改进他们的商用 UNIX 版本，直到他们吸收了 BSD UNIX 中已有的各种先进特性，并结合其本身的特点，推出了 Unix System V 版本之后，情况才有了改变。

虽然 AT&T 的 UNIX System V 也是非常优秀的 UNIX 版本，但是 BSD UNIX 在 UNIX 领域内的影响更大。AT&T 的 UNIX 系统实验室一直关注着 BSD 的发展。1992 年，UNIX 系统实验室指控 BSDI——一家发行商业 BSD UNIX 的公司——违反了 AT&T 的许可权，发布自己的 UNIX 版本，并进一步指控伯克利计算机系统研究组泄漏了 UNIX 的商业机密（此时的 4.3BSD 中来自 AT&T Unix 的代码已经不足 10%）。这个官司影响了很多 UNIX 厂商，使他们不得不从 BSD UNIX 转向 UNIX System V,以避免法律问题。以至于当今大多数商业

UNIX 版本都是基于 UNIX System V 的。

这件有关 UNIX 版权的案子直到 UNIX 系统实验室被 AT&T 卖给了 Novell 公司后才得以解决。Novell 不打算陷入这样的法律纷争中，因此就采用了比较友好的做法，伯克利的 CSRG 被允许自由发布 BSD，但是其中来自于 AT&T 的代码必须完全删除。因此 CSRG 就对他们最新的 4.4BSD 进行了修改，删除了那些来自于 AT&T 的源代码，发布了 4.4 BSD Lite 版本（该系统是不完整的，尤其对于英特尔 386 体系的计算机系统）。由于这个版本不存在法律问题，4.4BSD Lite 成为了现代 BSD 系统的基础版本。

Novell 比较友善的做法还不止对 BSD，它把自己的 UNIX 改名为 Unixware，而将 UNIX 商标赠送给 X/Open——一个由众多 UNIX 厂家组成的联盟，这样这个联盟内的所有成员均可使用 UNIX 商标。从此之后，UNIX 不再是专有产品的名字了。同时，由于 BSD 系统已经十分成熟，作为对操作系统进行研究的目标已经达到，伯克利计算机系统研究组（CSRG）在发布了 4.4BSD-lite2 之后就解散了，小组的科研人员有些进入了 UNIX 商业公司，有些继续进行其他计算机领域的研究。此时，严格意义上的 UNIX System V 和 BSD UNIX 都不复存在了，存在的只是他们的各种后续版本。

在 UNIX System V 之后的一些典型商业 UNIX 版本包括：IBM 公司的 AIX；HP 公司的 HP/UX；SUN 公司的 Solaris；SGI 公司的 IRIX 等；SCO 公司的 OpenServer 等。

Sun 是最早的工作站厂商，但一直在 UNIX 工作站领域不断发展。其操作系统 SunOS 是基于 4.2BSD 开发的，直到 SunOS 4。但是在此之后，Sun 将操作系统的开发工作转向了 System V，这个新版本为 Solaris 2，或者称为 SunOS 5，因此也可以将 SunOS 4 称为 Solaris 1.0，但是 SunOS 4 与 Solaris 2 分属两个流派，其中的差别就比较大了。值得一提的是 Sun 公司的创始人之一的 Bill Joy，他就是 BSD 的主创人员之一，对 UNIX 的贡献主要包括基于分页的虚存系统、csh、vi、NFS、以及最为重量级的 UNIX 应用——TCP/IP。后来在 Sun 公司又参与设计 Sparc 芯片，参与设计 Java，并一手缔造出 JINI 规范。一般人终其一生，能有他的一项成就，就足以傲视业界，遑论软硬兼修了。

IRIX 是 SGI 公司的 Unix，这也是一种基于 Unix System V 的产品。SGI 的 Unix 图形工作站是图形图象处理领域内的顶级产品，这一方面是由于 SGI 的硬件性能相当优秀，另一方面在软件方面，SGI 开发了工作站下的图形图象处理软件，成为这个领域的领先者。SGI 在图形图象领域的领先地位掩盖了他作为一家 UNIX 厂商在操作系统领域内的努力，事实上在他们还生产超级计算机，在多处理器和并行计算等大负荷计算方面都有独到的研究。

SCO UNIX 是在国内比较有名气的 UNIX 操作系统，并且较早进入中国市场。SCO（the Santa Cruz Operation）成立于 1978 年，同年售出第一个商用 C 编译器（Whitesmiths）。1980 年 SCO 和微软联合开发了基于 AT&T UNIX V6 的 XENIX，1987 年 SCO 收购了微软的 XENIX 版权，1995 年收购了 Novell 公司的 UnixWare 业务，1995 年 SCO 推出了 OpenServer Release 5。SCO 之所以能占有市场，并不是其产品特别出色，而是因为在小型机特别昂贵的年代，对一些追求稳定的行业来说，使用能在 x 8 6 上运行的 SCO UNIX，可以节约大量成本。因此早期的银行、金融行业的终端大多使用 SCO UNIX，SCO 产品一度占据 UNIX 中低端市场份额的 40%,但是随着 Linux 的崛起，SCO 经营情况每况愈下。2004 年 3 月，SCO 宣称 IBM 的 Linux 产品使用了 SCO 的 Unix 源代码程序，对 IBM 提出了盗用商业机密,侵权干涉，不公平竞争以及违背契约等多项指控，并开出了要求赔偿 10 亿美元的天价。同年 5 月，SCO 公司对全球 Linux 用户发出了这样的警告：“UNIX 知识产权的问题一天得不到正确解决,Linux 的用户也一天得不到安宁”。为了表示自己的决心，SCO 还戏剧性地宣布暂停发售自己所有的 Linux 产品。SCO 的指控立即引来了业界，尤其是 Linux 社区的熊熊怒火，不但被痛斥为“败类”(因为 SCO 原本是 Linux 阵营的重要厂商)，其网站也因遭遇拒绝服务式攻击而瘫痪。成千上万的 Linux 用户在网上发起请愿运动，直接要求：SCO，请你起诉我！截至 5 月 22 日，已经有 2500 多个单位和个人主动要求 SCO 来起诉，有人写道：“SCO，请你起诉我和我的母亲吧，因为我们都热爱 Linux”。在软件史上，这样令人感动的场景可以说绝无仅有。但 SCO 并没有因此止步，5 月 17 日，它又致函全球 1500 家大企业，警告它们：使用 Linux 将承担相应的法律责任。SCO 上演的这一出闹剧进一步震动了市场，震撼了全球 IT 界。在三年多时间的诉讼过程中，事件扑朔迷离，时不时的闪现微软的影子，2007 年 8 月，美国联邦法官做出裁决称，Novell 是 UNIX 和 UnixWare 的版权拥有者。SCO 必须向 Novell 支付损失费。2007 年 12 月 28 日，SCO 接到了美国纽约纳斯达克上市资格委员会的通知，其股票将从纳斯达克市场摘牌。

第四阶段：开源

在两大阵营不断纷争的过程中，UNIX 仍在艰难的发展，但是微软的崛起使得他们失去了太多的市场，而 IBM 的霸主地位更使他们感到无奈，Sun 公司的出现也使得 UNIX 大树上这根源码来源的树枝逐渐枯萎，甚至 386 芯片的下线也没能引起他们足够的重视。

上世纪 80 年代初个人计算机的出现，给个人拥有一个 UNIX 系统的愿望带来了一线曙光。但是使用 8086 芯片的计算能力还不足以在其上实现真正正常工作的多用户系统。进入

九十年代之后，英特尔公司推出的 80386 芯片使个人计算机的计算能力大大提高，在个人计算机上构建一个可以真正使用的 UNIX 也成为了可能。

事实上此时能运行在 X86 平台上的 UNIX 相当有限，Mircosoft 的 XENIX 是一种（后来发展成为 SCO Unix），但不能指望能自由、免费使用这个商用系统。要移植 UNIX 到 X86 平台上便需要 UNIX 的源代码，而此时受 AT&T 的许可权的限制，UNIX 代码还不能被自由获得（但在 BSD 与 AT&T 的法律纠纷之后的 4.4 BSD Lite 不再受这个限制了，可以在 BSD 许可下自由使用）。很多计算机爱好者为了实现一个可以自由使用的操作系统，不断进行着努力。自由软件基金会的 GNU 计划的目的就是打算创建一个自由的、与 UNIX 类似的操作系统，为了实现这个目的，GNU 开发了很多非常有效的工具，包括编译器和很多系统命令，然而 GNU 计划中的操作系统内核 ——HURD，却进展缓慢，从而无法构建一个完整的操作系统。很快，就有计算机爱好者开始考虑在个人计算机平台上构建一个 UNIX 内核。芬兰一位计算机研究生 Linus Torvalds 开始了这个工作，并取得了相当大的成功，他通过学习 Minix——一个用于教学目的的简单 UNIX 系统，在 x86 平台上构建了 Linux 内核，这个内核和 GNU 的系统工具结合起来，成为当前非常流行的 GNU/Linux 系统。

与这些努力相比，BSD 方面的研究人员的行动则比较迟缓，一个原因是 Unix 技术上已经相当成熟，计算机系统研究组的大部分成员已经把注意力转向了其他研究项目，另外 AT&T 与 BSD 的法律纷争也使得 BSD 发展受到了阻碍。但是还是有两个将 BSD 移植到 80386 平台的开发工作同时进行，一个是 BSD/386 小组，他们的研究成果是商业版本的 BSD/OS，属于商业公司 BSDI。另一个就是 386BSD 计划，后来发展成为 FreeBSD。

CSRG 研究人员的条件比较优越，拥有足够的 UNIX 系统，因此对个人计算机平台上的 UNIX 系统没有太急迫的要求。386BSD 计划由 Bill Jolitz 等研究人员发起，将 4.3BSD Net/2 移植到 80386 平台上，并使用 386BSD 的名称发布。但是移植工作是一个复杂的任务，直到 386BSD 0.5 版本，系统中仍然存在相当多的问题没有解决。于是在 1993 年，另一些研究者决定加入这个计划，打算和 Bill Jolitz 一起修正系统中存在的问题。但是这时计划的组织者 Bill Jolitz 突然决定退出，使得 386BSD 计划面临停止的危险。Bill Jolitz 作为计划的发起者和负责者并没有对这个计划以后该如何发展给出具体意见，因此 386BSD 计划是到此为止还是继续发展，就取决于其他开发者的决定。幸运的是，该项目的另三个参与者 Nate Williams, Rod Grimes 和 Jordan K. Hubbard 决定继续这项非常有意义的工作，他们采用由 David Green man 创造的名字 FreeBSD 作为系统的新名字，从此有了一种任何人都可以自由使用的 UNIX

操作系统 —— FreeBSD。

FreeBSD 1.0 版本通过光盘和 Internet 来发布。任何人都可以通过购买光盘或者通过 Internet 下载的方法，自由获得 FreeBSD 系统，使得 FreeBSD 取得了很大成功。FreeBSD 虽然可以自由获得，然而 BSD 与 AT&T 的法律纠纷仍然威胁着 FreeBSD 系统的合法性。就在 FreeBSD 得到相当多用户欢迎的时候，Unix 系统实验室（此时已经卖给了 Novell）与伯克利计算机系统研究小组的法律纷争有了结论。虽然最后不必进行赔偿，但是 BSD UNIX 系统中必须去除原来来自 AT&T 的源码。伯克利计算机系统研究小组去除了这些不到 10%源码，发布了 4.4BSD Lite，其他基于 BSD 的 UNIX，包括 FreeBSD 在内，都被要求立即转换到 4.4 BSD Lite 上去。

这对 FreeBSD 是一次相当严重的打击，虽然 4.4 BSD Lite 只删除了一小部分代码，但尤其对于英特尔 80386 平台，缺乏这些代码，系统就不能正常运转。FreeBSD 小组必须解决两个任务，首先是将 FreeBS D 从原本的 4.3BSD 迁移到 4.4BSD 上，再将删除的源码完全重写。这些任务相当于将 4.4BSD Lite 重新移植到 80386 上，因此这花费了 FreeBSD 核心小组很大的精力。因此直到 1995 年 1 月他们才发布了 FreeB SD 2.0，这次就是一个完全的 4.4BSD Lite 的系统了。但是在大约一年时间之内，FreeBSD 不能在原有 1.0 基础上进行改进并推出新版本，而这个时期正是 Internet 进一步发展的阶段，FreeBSD 错过了一个发展壮大的好时机。而其竞争对手 Linux，则取得了巨大的成功。

由于 UNIX 商标属于 X/Open 组织，而 FreeBSD 只是一个自由操作系统，从法律角度上看 FreeBS D 不能被叫 UNIX（不能使用 UNIX 做商标）。但是基于 UNIX 本身的历史，FreeBSD 可以算最原汁原味的 UNIX，在有的方面，它更具传统特色——或者说 BSD UNIX 的学院特色。当前，UNIX 商标其实是只具有象征性的含义，没有人介意到底那些系统是 X/Open 的成员，可以被称作 UNIX，那些不是。UNIX 已经成为一个广泛的概念，只要是按照 UNIX 为模板进行开发，所有的应用程序在 C 源程序级与其他 UNIX 相互兼容，也同样被所有使用者承认为 UNIX 系统。因此这里的 UNIX 包括 BSD 和 System V 在内的各种系统，也包括像 Linux 这样的兼容系统。

到目前，Internet 上常见的 UNIX 系统有以下几种：各商业公司的基于 AT&T 的 UNIX 系统（非 Intel 平台据多），主要是 IBM 的 AIX，HP 的 HP/UX，SGI 的 IRIX，Sun 的 Solaris 等，基于 BSD UNIX 的 BSDI 和 FreeBSD，以及 Linux。其中 FreeBSD 和 Linux 是可在 X86 上运行的免费的操作系统，我们能够使用的也基本上只有这两个 UNIX 系统，但在这里，我

们还要分清楚 Linux 和 FreeBSD 的区别，主要有两个：1、FreeBSD 是由最初的 BSDUNIX 一路发展下来的正统的 UNIX 系统，而 Linux 是一个遵循 POSIX 标准所有系统代码全部重新编写了的操作系统。2、FreeBSD 是完整的一个操作系统，而 Linux 只是一个内核，加上各种 GNU 软件构成的操作系统，所以，才会有很多的 Linux 系统，如 Red Hat、SUSE、Debian、ubuntu、Slackware、Turbo Linux、RedFlag、BluePoint 等。

回顾 UNIX 的发展，可以注意到 UNIX 与其他商业操作系统的不同之处主要在于其开放性。在系统开始设计时就考虑了各种不同使用者的需要，因而 UNIX 被设计为具备很大可扩展性的系统。由于它的源码被分发给大学，从而在教育界和学术界影响很大，进而影响到商业领域中。大学生和研究者为了科研目的或个人兴趣在 UNIX 上进行各种开发，并且不计较金钱利益，将这些源码公开，互相共享，这些行为极大丰富了 UNIX 本身。很多计算机领域的科学家和技术人员遵循这些方式，开发了数以千计的自由软件，包括 FreeBSD 在内。正因为如此，当今的 Internet 才如此丰富多采，与其他商业网络不同，才能成为真正的全球网络。开放是 UNIX 的灵魂，也是 Internet 的灵魂。

第五阶段：未来

UNIX 发展将走向何方？——操作系统厂商正在改进 UNIX 特性，使之承担更复杂的应用和服务器任务。 面对 Linux 和开放系统硬件日益严重的威胁，UNIX 厂商推出了功能更强大的 UNIX 操作系统版本，并承诺更好地支持关键数据中心的操作。与以前的版本相比，来自 IBM、Compaq、Caldera、HP 和 Sun 的最新 UNIX 版本具有更好的伸缩性、安全性以及可管理性：这些系统可以在 32 位多处理器系统上运行基于 Linux 和 Web 的应用。不久，多数 UNIX 系统将可以在 64 位 Intel 处理器上运行，并可以用于执行以前需要大型机或超级计算机的应用，UNIX 系统开始占领过去一直由大型机占领的领域。

UNIX 的头号应用是数据库和分布式应用。尽管 UNIX 在低端输给了 Windows 和 Linux，但是许多公司的网络核心仍是 UNIX 的天下。虽然许多网络的前端使用 Windows 或 Linux，但是后端系统一般仍是 UNIX。UNIX 在 2000 年全部操作系统收入中所占份额由 54%增长到 57%。大多数 Unix 厂商正在对 UNIX 的特性进行改进以处理更复杂的应用和服务器任务。例如，厂商正在扩展这种操作系统的群集和多处理功能，使 UNIX 在与 Windows 2000 和 Linux 这类操作系统竞争时更具竞争力。UNIX 市场份额的龙头老大 Sun 公司在其新版 Solaris 里包括对 Jini 的支持。Jini 是一种用于连接网络应用和提供网络服务的基于 Java 的技术。这种流行操作系统的下一版本还将包括对 Itanium 处理器的支持。Sun 8 目前包括在 Solaris 上

运行重新编译的 Linux 应用以及使用 Gnome 桌面界面的功能。同时，IBM 公司全力推进其 AIX 5L Unix 软件包。它是一种设计用于事务处理密集环境以及在 RS/6000 和 Itanium 服务器上进行高速处理的操作系统，AIX 5L 支持像多路径 I/O 和对称多处理等特性，AIX 5 还增加了 Workload Manager，使 IT 专业人员可以定义管理应用的方式以及向应用分配处理器时钟周期、实际内存和磁盘 I/O，实现应用的优化运行。

2、UNIX 文化

UNIX 自诞生之日，便一直应用于生产领域，比 PC 机、工作站等都要早，与第一块半导体存储器是同一时代的古物。也许只有 IBM 的 VM/CMS 敢说它比 UNIX 更老。

UNIX 比其他任何操作系统都更广泛的应用在各种机型上，从超级计算机到手持计算机到嵌入式设备，从工作站到服务器到 PC 机到微型计算机。

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

UNIX 的生命力委实令人称奇，尽管其他技术如浮游般生生灭灭，计算机性能成千倍增长，语言历经嬗变，业界规范多次变革——然而 UNIX 依然巍然屹立，仍在运行，在创造价值。性能-时间的指数曲线对软件开发过程所引发的结果，就是每过 18 个月，就有一半的知识会过时，UNIX 并不承诺让你免遭此劫，只是让你的知识投资更趋稳定，因为不变的东西有很多——语言、系统调用、工具用法——它们积年不变，甚至可用数十载。

UNIX 的稳定和成功在很大程度上归功于它与生俱来的内在优势，归功于 Ken Thompson，Dennis Ritchie，Brian Kernighan，Dong McIlroy，Rob Pike 和其他早期 UNIX 开发者一开始就作出的设计决策。这些设计决策，连同设计哲学、编程艺术、技术文化一起，已经被反复证明是健康可靠的。

也许对 UNIX 最持久的异议在于“行为的最终逻辑尽可能被推后到使用端”，因为“最终用户永远比操作系统设计人员更清楚他们究竟需要什么”。UNIX 致力于“提供机制，而非策略”。UNIX 原本是为技术人员设计的操作系统，因此 UNIX 应用程序通常提供的很多的行为选项和令人眼花缭乱的定制功能让一般用户望而却步。但是 Doug McIlroy 相信“用错误的方式解决正确的问题总比用正确的方法解决错误的问题好”。

UNIX 哲学是这样的：

1、模块原则：使用简洁的接口拼合简单的部件。

2、清晰原则：清晰胜于机巧。

3、组合原则：设计时考虑拼接组合。

4、分离原则：策略同机制分离，接口同引擎分离。

5、简洁原则：设计要简洁，复杂度能低则低。

6、吝啬原则：除非确无它法，不要编写庞大的程序。

7、透明原则：设计要可见，以便审查和调试。

8、健壮原则：健壮源于透明与简洁。

9、表示原则：把知识叠入数据以求逻辑质朴而健壮。

10、通俗原则：接口设计避免标新立异。

11、缄默原则：如果一个程序没什么好说的，就沉默。

12、补救原则：出现异常时，马上退出并给出足够错误信息。

13、经济原则：宁花机器一分，不花程序员一秒。

14、生成原则：避免手工 hack，尽量编写程序去生成程序。

15、优化原则：雕琢前先要有原型，跑之前先学会走。

16、多样原则：决不相信所谓“不二法门”的断言。

17、扩展原则：设计着眼未来，未来总比预想来得快。

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

UNIX 哲学是这样的：KISS（Keep It Simple，Stupid！）。

3、POSIX 标准和 ANSI C 标准

UNIX 派系繁多，各版本之间或多或少存在各种差异，这不利于操作系统本身的发展，更不利于最终用户的使用。1985 年， IEEE 标准委员会开始在 ANSI 的支持下责成 IEEE 操作系统技术委员会标准小组委员会（TCOS-SS）制定有关可移植性操作系统接口标准 POSIX(Portable Operating System Interface for Computing Systems)。到了 1986 年 4 月，IEEE 就制定出了试用标准。该标准基于现有的 UNIX 实践和经验，描述了操作系统的调用服务接口，用于保证编制的应用程序可以在源代码一级上在多种操作系统上移植运行。它是在一个 UNIX 用户组早期工作的基础上取得的，该 UNIX 用户组原来试图将 AT&T 的系统 V 和

Berkeley CSRG 的 BSD 系统的调用接口之间的区别重新调和集成。

第一个正式标准是在 1988 年 9 月份批准的 IEEE 1003.1-1988，即我们现在所说的 POSIX.1 标准。

POSIX.1 仅规定了系统服务应用程序编程接口（API），仅概括了基本的系统服务标准，因此期望对系统的其它功能也制定出标准。1989 年，POSIX 的工作被转移至 ISO/IEC 社团，并由 15 工作组继续将其制定成 ISO 标准。1990 年，就有十多个批准的计划在进行，有近 300 多人参加每季度为期一周的会议。着手的工作有命令与工具标准(POSIX.2)、测试方法标准（POSIX.3）、实时 API（POSIX.4）等。与此同时，还有一些组织也在制定类似的标准，如 X/Open，AT&T，OSF 等。

目前最新的POSIX标准是IEEE Std 1003.1, 2004 Edition，可通过http://www.unix.org浏览。POSIX标准被组织为 4 个部分（共有 3600 多页）：

基本定义（Base Definition）：通用术语与概念、头文件

系统接口（System Interface）：函数的定义

命令解释程序和工具（Shell and Utilities）：命令的定义

基本原理（Retionale）：有关历史信息的讨论以及为什么某些特性包含在标准中，而某些未包含。

有许多 UNIX 和类 UNIX 都宣称支持 POSIX，甚至 Windows NT，尽管如此，相互之间也不能做到完美的可移植，因为有些系统不是完全支持 POSIX，有些内容 POSIX 没有定义，有些方面 POSIX 只是建议。

值得一提的是 POSIX 制定初期，恰逢 Linux 诞生之时，这个 UNIX 标准为 Linux 提供了极为重要的信息，使得 Linux 的能够在该标准的指导下进行开发，能够与绝大多数 UNIX 系统兼容。可以这样说，Linux 能到今天，POSIX 是关键元素之一，其他的当然包括 MINIX，GNU，Internet。

C 语言由 Dennis M. Ritchie 在 1973 年设计和实现。从那以后使用者逐渐增加。到 1978 年 Ritchie 和 Bell 实验室的另一位程序专家 Kernighan 合写了著名的《The C Programming Language》，将 C 语言推向全世界，许多国家都出了译本，国内有一些 C 语言书就是这本书的翻译或者编译。由这本书定义的 C 语言后来被人们称作 K&R C。

随着 C 语言使用得越来越广泛，出现了许多新问题，人们日益强烈地要求对 C 语言进行标准化。这个标准化的工作由美国国家标准局（ANSI）负责，最终在 1988 年 10 月颁布

的 ANSI 标准 X3.159-1989，也就是后来人们所说的 ANSI C 标准。由这个标准定义的 C 语言被称作 ANSI C。 ANSI C 很快被采纳为国际标准，因此也叫 ISO C。参考文献 3 是按照 ANSI C 修改后的 C 语言教程。新版 POSIX 标准包含了 ISO C。

POSIX 是系统标准，ANSI C 是语言标准。

4、系统程序开发的预备知识

虽然我们讲的是“UNIX 系统编程”，但考虑到实际情况，我们选用 Linux 作为测试平台，这并不影响我们对知识理解的偏差，因为它们都服从 POSIX 标准。事实上，从某些方面讲，Linux 比 UNIX 更 UNIX。现在假定你会 Linux 的基本操作，我们的测试平台是 Red Hat 9 及以上版本，建议你在自己的个人机上安装虚拟机或双系统。尽管 UNIX 是用 C 语言写的，但 C 语言并不是 UNIX 和 Linux 环境下的唯一编程语言，下面列出一些常用语言：

Ada	C	C++	Eiffel	Forth
Fortran	Icon	Java	JavaScript	Lisp
Modula 2	Modula 3	Oberon	Objective C	Pascal
Perl	PostScript	Prolog	Python	Scheme
Smalltalk	SQL	Tcl/Tk	Bourne Shell	

从知识的实用角度讲，作为一名 UNIX 程序员，你必须至少会两门编程语言：Shell 和 C。关于 Shell 编程，我们主要在“Linux 基础与应用”中讲述，而 C，即使你已学过了，你也不敢说你是 C 语言高手或熟练者。

让我们还是从史上最著名的 Hello World 开始吧。

4.1、编辑

下面是文件 hello.c 的源代码：

```
#include <stdio.h>
int main()
{
   printf(“Hello World!\n”);
   return 0;
}
```

需要用一个编辑器来输入这个 C 语言源程序。一般我们都用 vi，因为她跟 UNIX 一样

悠久，使用她有一种复古的情结，还因为几乎在所有的 UNIX 下都能见到她的身影，还因为她本身也很好用——如果你用熟了的话（关于 vi 的使用，请 man vi）。也有人喜欢用 emacs 或其他编辑器，这都无关紧要，用熟了都好。这里我们需要注意的是，如果你想成为一个 UNIX 程序员，那么 C 语言程序设计风格就显得很重要，下面列出一些，更多的参考文献 3：

每行只写一条语句；

缩进 Tab=8，当缩进多于 3 个时，考虑修改程序；

运算符两边各加一个空格；

尽量使用符号常数且用大写字母拼写，不使用“幻数”；

在函数定义中左右花括号各占一行，在一般语句中，左花括号跟在语句后边，而右花括号独占一行且和左花括号外语句对齐；

函数应尽量短小精悍，一个函数只做一件事且要做好，函数不超过 2 屏，内部变量不超过 10 个，否则，重新考虑程序；

注释过多不好，说明是做什么的，而不是怎么做的；

浮点常量取整数时，后面加小数点和 0；

变量命名应尽量精简，内部变量尽量一个字符，尤其是循环变量，全局变量需要描述性定义。

4.2、编译、链接

Linux 下的 C 编译器是 gcc，其他 UNIX 平台下编译器的名字可能不一样（如：cc），但用法基本一样。

1984 年，Richard Stallman 发起了自由软件运动，GNU (Gnu's Not Unix)项目应运而生，3 年后，最初版的 gcc 横空出世，成为第一款可移植、可优化、支持 ANSI C 的开源 C 编译器。gcc 最初的全名是 GNU C Compiler，之后随着 gcc 支持的语言越来越多，它的名称变成了 GNU Compiler Collection。

gcc 是 Linux 平台下最常用的编译程序，它是 Linux 平台编译器的事实标准。

gcc 是可以在多种硬件平台上编译出可执行程序的超级编译器，目前，gcc 支持的体系结构有四十余种，常见的有 X86 系列、Arm、PowerPC 等，其执行效率与一般的编译器相比要高 20%~30%。

gcc 除了支持 C 语言外，还支持多种其他语言，例如 C++、Ada、Java、Objective-C、FORTRAN、Pascal 等。

gcc 还能按需求生成各种类型的输出文件，包括中间文件，汇编文件，目标文件和可执行文件。

由于 gcc 能支持各种不同的目标体系结构，因此，gcc 也是在基于 Linux 的嵌入式开发领域用得最普遍的一种编译器，它既支持基于宿主的开发（简单讲就是要为某平台编译程序，就在该平台上编译），也支持交叉编译（即在 A 平台上编译的程序是供平台 B 使用的）。

gcc 的基本使用格式为：gcc [选项] 文件 1 文件 2 …

gcc 有丰富的选项，这里只介绍最基本的选项，若想知道更多，可以 man gcc。

gcc 的编译过程需要经历四个相互关联的步骤：预处理(也称预编译，Preprocessing)、编译(Compilation)、汇编(Assembly)和连接(Linking)。

在预处理阶段（gcc 调用 cpp），输入的是 C 语言的源文件，通常为*.c。这个阶段主要处理源文件中包含的各个头文件。该阶段会生成一个中间文件*.i，可用 E 选项获得*.i 文件，如下所示（-o 选项指定输出文件）：

gcc -E -o hello.i hello.c

在编译阶段（gcc 调用 ccl），输入的是中间文件*.i，编译后生成汇编语言文件*.s。这个阶段对应的 gcc 选项为 S，如下所示（注意区分大小写）：

gcc -S -o hello.s hello.i 或者：

gcc -S -o hello.s hello.c

在汇编阶段，将输入的汇编文件*.s 转换成目标文件*.o。这个阶段对应的 gcc 选项为 c，如下所示：

gcc -c -o hello.o hello.s 或者：

gcc -c -o hello.o hello.i 或者：

gcc -c -o hello.o hello.c

最后，在链接阶段（gcc 调用 ld）将目标文件*.o 与相关库文件链接成一个可执行文件，gcc 对可执行文件的命名没有规定，可以是任意合法的文件名，如下所示：

gcc -o helloexe.c hello.o

gcc -o helloexe.c hello.s

gcc -o helloexe.c hello.i

gcc -o helloexe.c hello.c

若没有指定 o 选项，则 gcc 缺省可执行文件名为 a.out。当然我们一般不会命名可执行

文件为*.c，且一般在调试程序时，只需用命令：gcc -o hello hello.c 即可。

4.3、运行

只需在命令行输入你的可执行文件名（如：hello），然后回车即可。当然问题并没有这么简单，如果 PATH 环境变量中不包含你的可执行文件所在的目录，shell 就找不到 hello 程序，则无法运行你的程序；或者，如果 PATH 包含的某个目录中有另一个名为 hello 的程序，则 shell 就会执行那个程序。如果 PATH 中这样的目录出现在你的 hello 所在目录之前，也会发生这种情况。为了避免这种潜在的问题，我们在程序名前加上一个./（例如./hello）。它特别指示 shell 去执行当前目录下给定名称的程序（因为“.”表示当前目录）。

4.4、头文件和库文件

用 C 语言进行程序设计时，需要把常量、数据类型和函数的声明都放在头文件中（扩展名为.h）。在程序中将系统定义的头文件包含在一对尖括号中（如：#include <stdio.h>），gcc 在预处理阶段会在一个标准的位置来查找这个文件，一般这些头文件总是在/usr/include 目录及其子目录下。你也可以建立自己的头文件，在程序中用双引号包含你的头文件（如 #include “myinclude.h”），双引号告诉 gcc 在标准的位置寻找头文件之前，先在包含源文件的目录中查找头文件。你当然可以把你自己的头文件放在任何位置，只要在双引号中指出其绝对路径即可。

源程序中包含了正确的头文件并不意味着你的麻烦就结束了。因为头文件给出了符号声明和函数原型，但它并没有为函数提供实际的代码。

对！定义函数的实际代码在库文件中，库是一组预先编译好的函数的集合，这些函数都是按照可重用的原则编写的。有很多库文件。显然我们应该告诉 gcc 到什么位置的哪个库文件中去提取在程序中要用到的函数（譬如 printf（）），那么怎么告诉呢？

首先，Linux 操作系统提供的库文件一般存储在/usr/lib 及其子目录中，gcc 默认的查找目录为/usr/lib，其次，ANSI C 语言的标准库函数在库文件 libc.a 中，gcc 默认的库文件为 libc.a。也就是说，如果你的程序调用的都是 C 标准库函数，那就不用告诉 gcc 库文件的路径和名字。遗憾的是，由于种种原因，libc.a 并没有包含全部的 C 标准库函数。如果要引用其他位置的其他库文件，则要把库文件所在的目录加到系统环境变量 LD_LIBRARY_PATH 中，或者在命令行用选项 L 指定，而把库文件在命令行用选项 l 指定，-lx 表示指定库文件 libx.a 或 libx.so。gcc 对库文件的搜索顺序为：命令行上由-L 指定的目录，LD_LIBRARY_PATH 环境变量中的目录，缺省库目录（/usr/lib）。

和头文件一样，你也可以拥有你自己的库文件，下面的例子给出了方法。不建议把你自己的库文件拷贝到缺省库目录下，因为一方面要求库文件名为 lib*.a 格式，另一方面不便于移植。

库文件有两种形式：静态库和共享库。静态库也称作归档文件（archive），它们的文件名都以.a 结尾，如 C 标准静态库为/usr/lib/libc.a。静态库的一个缺点是，当我们同时运行许多应用程序并且它们都使用来自同一个函数库的函数时，就会在内存中有同一函数的多份拷贝，在程序文件自身中也有多份同样的拷贝，这将消耗大量宝贵的内存和磁盘空间。共享库的保存位置与静态库是一样，共享库的文件名都以.so 结尾，如 C 标准数学共享库为/usr/lib/libm.so。程序使用共享库时，它的链接方式是这样的：它本身不再包含函数代码，而是引用运行时可访问的共享代码。当编译好的程序被装载到内存中执行时，函数引用被解析并产生对共享库的调用，如果有必要共享库才被加载到内存中。通过这种方法，系统可只保留共享库的一份拷贝并供许多应用程序同时使用，并且在磁盘上也仅保存一份。另一个好处是共享库的更新可以独立于依赖它的应用程序。

大多数 UNIX 和 Linux 都支持共享库，gcc 在链接库文件时先找共享库，再找静态库。

我们可通过运行工具 ldd 来查看程序链接的共享库。例如：

```
ldd   hello
    libc.so.6 => /lib/libc.so.6 (0x42000000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

一个静态库的例子。

下面我们来创建一个小型函数库，它包含两个函数，然后在一个示例程序中调用其中一个函数。这两个函数分别是 fred 和 bill，它们只打印欢迎信息。

（1） 为两个函数分别创建各自的源文件（将它们分别命名为 fred.c 和 bill.c）。下面是第一个源文件：

```
#include <stdio.h>
void fred(int arg)
{
   printf("fred: you passwd %d\n",arg);
}
```

下面是第二个源文件：

```
#include <stdio.h>
void bill(char *arg)
{
   printf("bill: you passwd %s\n",arg);
}
```

（2） 分别编译这两个函数，产生要包含在库文件中的目标文件。这通过选项-c 来实现，-c 的作用是阻止编译器创建一个完整的程序。如果此时试图创建一个完整的程序将不会成功，因为我们还未定义 main 函数。

$ gcc -c bill.c fred.c

$ ls *.o

bill.o fred.o

（3） 为我们的库文件创建一个头文件 mylib.h。这个头文件将声明我们库文件中的函数，它应该被所有希望使用我们库文件的应用程序所包含。

```
/*
This is mylib.h. It declares the functions fred and bill for users
*/
void bill(char *);
void fred(int);
```

（4） 编写一个调用 bill 函数的程序 program.c，该程序非常简单。它包含库的头文件并且调用库中的一个函数。

```
#include "mylib.h"
int main()
{
   bill("Hello World!");
   return 0;
}
```

（5） 创建并使用一个库文件。我们用 Linux 工具 ar 程序创建一个归档文件并将目标文件添加进去。

ar crv libmy.a bill.o fred.o

a – bill.o

a – fred.o

库文件创建好了，两个目标文件也已添加。我们的函数库现在就可以使用了。一种方法是在编译器命令行的文件列表中添加该库文件，如下所示：

gcc -o program program.c libmy.a

也可以用-l 选项来访问我们的函数库，但是因为其未保存在标准位置，所以必须用-L 选项来指示编译器在何处可以找到它，如下所示：

gcc -o program -L. program.c -lmy

“-L.”指示 gcc 在当前目录下寻找库文件，“-lmy”指示 gcc 寻找名为 libmy.so 或 libmy.a 的库文件。这里注意-l 的位置很重要，它必须出现在被编译文件的后面，因为 gcc 从左向右读取命令行参数，只有那些与未解析的引用相符的条目才会被装载近来。当然，你还可以用以下方法编译该程序：

gcc -o program program.c bill.o

算是领教了“提供机制，而非策略”的厉害了！

5、Makefiles

make 工具允许用户递增地对一组程序模块进行重新编译。要使用 make，就必须在一个描述文件（description file）中说明模块之间的依赖关系。make 工具通过描述文件来查看是否有内容需要更新。

描述文件指定了目标（target）和其他组件之间存在的依赖关系。以#起始的行是注释行。描述文件中的依赖关系的格式如下所示：

```
target:     components
TAB         rule
```

第一行被称为依赖关系（dependency），依赖关系可以只有目标而没有组件，第二行被称为规则（rule）。描述文件中规则行的第一个字符必须是 TAB 字符，依赖关系后面可能会有一个或多个规则，也可以没有规则。

默认的描述文件名为 makefile 或 Makefile，如果用户输入 make 时不带任何额外的参数，make 工具就在当前的目录中查找 makefile 或 Makefile，并将其作为描述文件使用。

例如：为了编译 hello.c 文件，可以在 hello.c 所在的目录下写一个 makefile 文件：

```
# This is my first makefile
hello:hello.c
    gcc  -o  hello  hello.c
```

然后在命令行直接输入 make 命令，就可以完成编译 hello.c 的任务。注意 make 是“**更新**”，也就是说这里只有 hello 不存在或者它的依赖组件（即 hello.c）被更新了，才会执行规则 gcc。

当然可以写得更复杂些：

```
# This is my first makefile
hello:hello.o
    gcc  -o  hello  hello.o
hello.o:hello.s
    gcc -c  -o hello.o hello.s
hello.s: hello.i
    gcc -S  -o  hello.s hello.i
hello.i:hello.c
    gcc -E  -o  hello.i hello.c
```

这个文件只是演示依赖关系的例子，实际工作中当然没必要这样做了。

描述文件中还可以包含以下形式的宏定义（macro definition）：

```
NAME = value
```

每当$(NAME)出现在描述文件中时，make 都会在处理之前用 value 来取代它。如：

```
# This is my first makefile
EXE = hello
CC = gcc
$(EXE):$(EXE).c
    $(CC)  -o  $(EXE)  $(EXE).c
```

当 makefile 比较大时，这种宏定义是很方便的。

make 命令也允许在命令行上指定目标的名字。在这种情况下，make 只对指定的目标进行更新。如果在命令行上没有显式的指定目标，make 就只检查描述文件中的第一个目标。

通常，描述文件都将第一个目标命名为 all，并使它依赖于所有其他的目标。

为了调试方便，有时候需要删除一些中间文件，则在 makefile 中经常增加 clean 目标：

```
# This is my first makefile
EXE = hello
CC = gcc
all：$(EXE)
$(EXE):$(EXE).c
    $(CC)  -o  $(EXE)  $(EXE).c
clean：
    rm  *.o
```

一般我们总是通过 make all（all 可以不写）和 make clean 来调试程序或工程。描述文件也可以不是 makefile 或 Makefile，这是需要-f 参数来指定描述文件。

更多内容，请关注 GNU make 手册。

6、man

有什么问题吗？请 man 吧！

大多数 UNIX 系统都有在线文档，用于描述本系统支持的 POSIX 标准，这些在线文档被称为 man page（联机帮助页面），这里的“man”就表示系统手册中的“手册（manual）”。我们可以用 man 工具来查看这些文档的内容，man 工具用一种可读的形式来显示这些在线文档的页面。例如：man ls 就可以查看 ls 命令的格式、基本用法、参数等内容。

但是 POSIX 并不要求这个手册工具能提供很多的功能，对于每个 POSIX 中的条目（命令、函数、工具等），那么 POSIX 标准只要求 man 显示一条描述了该条目的语法、选项和操作数的消息。

有些条目可能会重复，如 write 既是一条命令，又是一个函数，如何分别用 man 去查看呢？大多数 UNIX 实现都将手册页面划分成小节，只要在 man 命令中加上对应小节号即可分别查看。UNIX 联机帮助页面典型的小节编码为：1、用户命令；2、系统调用；3、C 库函数、4、设备和网络接口；、5、文件格式；6、游戏与演示；7、环境、表和 troff 宏；8、系统维护。例如：用 man 1 write 查看命令，而用 man 2 write 查看函数。用命令 man -k name 查看所有包含 name 条目的摘要。