HZ BOOKS
华章科技

Linux/Unix
技术丛书

Linux 环境编程
从应用到内核

高峰 李彬 著

Programming in Linux Environment
from Userspace to Kernel

- Linux领域第一本将应用编程与内核实现相结合的图书
- Linux环境编程的进阶指导，解析Linux接口的工作原理，帮助应用开发人员快速深入内核，掌握Linux系统运行机制

机械工业出版社
China Machine Press

Linux/Unix
技术丛书

Linux
环境编程
从应用到内核

Programming in Linux Environment
from Userspace to Kernel

高峰 李彬 著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Linux 环境编程：从应用到内核 / 高峰，李彬著 . —北京：机械工业出版社，2016.5
（Linux/Unix 技术丛书）

ISBN 978-7-111-53610-9

I. L… II. ① 高… ② 李… III. Linux 操作系统 IV. TP316.89

中国版本图书馆 CIP 数据核字（2016）第 086961 号

Linux 环境编程：从应用到内核

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：佘 洁 责任校对：殷 虹
印 刷：三河市宏图印务有限公司 版 次：2016 年 6 月第 1 版第 1 次印刷
开 本：186mm × 240mm 1/16 印 张：38
书 号：ISBN 978-7-111-53610-9 定 价：99.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzit@hzbook.com

Preface 前　言

为什么要写这本书

我从事 Linux 环境的开发工作已有近十年的时间，但我一直认为工作时间并不等于经验，更不等于能力。如何才能把工作时间转换为自己的经验和能力呢？我认为无非是多阅读、多思考、多实践、多分享。这也是我在 ChinaUnix 上的博客座右铭，目前我的博客一共有 247 篇博文，记录的大都是 Linux 内核网络部分的源码分析，以及相关的应用编程。机械工业出版社华章公司的 Lisa 正是通过我的博客找到我的，而这也促成了本书的出版。

其实在 Lisa 之前，就有另外一位编辑与我聊过，但当时我没有下好决心，认为自己无论是在技术水平，还是时间安排上，都不足以完成一本技术图书的创作。等到与 Lisa 洽谈的时候，我感觉自己的技术已经有了一些沉淀，同时时间也相对比较充裕，因此决定开始撰写自己技术生涯的第一本书。

对于 Linux 环境的开发人员，《Unix 环境高级编程》（后文均简称为 APUE）无疑是最为经典的入门书籍。其作者 Stevens 是我从业以来最崇拜的技术专家。他的 Advanced Programming in the Unix Environment、Unix Network Programming 系列及 TCP/IP Illustrated 系列著作，字字珠玑，本本经典。在我从业的最初几年，这几本书每本都阅读了好几遍，而这也为我进行 Linux 用户空间的开发奠定了坚实的基础。在掌握了这些知识以后，如何继续提高自己的技能呢？经过一番思考，我选择了阅读 Linux 内核源码，并尝试将内核与应用融会贯通。在阅读了一定量的内核源码之后，我才真正理解了 Linux 专家的这句话“Read the fucking codes”。只有阅读了内核源码，才能真正理解 Linux 内核的原理和运行机制，而此时，我也发现了 Stevens 著作的一个局限——APUE 和 UNP 毕竟是针对 Unix 环境而写的，Linux 虽然大部分与 Unix 兼容，但是在很多行为上与 Unix 还是完全不同的。这就导致了书中的一些内容与 Linux 环境中的实际效果是相互矛盾的。

现在有机会来写一本技术图书，我就想在向 Stevens 致敬的同时，写一本类似于 APUE

风格的技术图书，同时还要在 Linux 环境下，对 APUE 进行突破。大言不惭地说，我期待这本书可以作为 APUE 的补充，还可以作为 Linux 开发人员的进阶读物。事实上，本书的写作布局正是以 APUE 的章节作为参考，针对 Linux 环境，不仅对用户空间的接口进行阐述，同时还引导读者分析该接口在内核的源码实现，使得读者不仅可以知道接口怎么用，同时还可以理解接口是怎么工作的。对于 Linux 的系统调用，做到知其然，知其所以然。

读者对象

根据本书的内容，我觉得适合以下几类读者：

- 在 Linux 应用层方面有一定开发经验的程序员。
- 对 Linux 内核有兴趣的程序员。
- 热爱 Linux 内核和开源项目的技术人员。

如何阅读本书

本书定位为 APUE 的补充或进阶读物，所以假设读者已具备了一定的编程基础，对 Linux 环境也有所了解，因此在涉及一些基本概念和知识时，只是蜻蜓点水，简单略过。因为笔者希望把更多的笔墨放在更为重要的部分，而不是各种相关图书均有讲解的基本概念上。所以如果你是初学者，建议还是先学习 APUE、C 语言编程，并且在具有一定的操作系统知识后再来阅读本书。

Linux 环境编程涉及的领域太多，很难有某个人可以在 Linux 的各个领域均有比较深刻的认识，尤其是已有 APUE 这本经典图书在前，所以本书是由高峰、李彬两个人共同完成的。

高峰负责第 0、1、2、3、4、12、13、14、15 章，李彬负责第 5～11 章。两位不同的作者，在写作风格上很难保证一致，如果给各位读者带来了不便，在此给各位先道个歉。尽管是由两个人共同写作，并且负责的还是我们各自相对擅长的领域，可是在写作的过程中我们仍然感觉到很吃力，用了将近三年的时间才算完成本书。对比 APUE，本书一方面在深度上还是有所不及，另一方面在广度上还是没有涵盖 APUE 涉及的所有领域，这也让我们对 Stevens 大师更加敬佩。

本书使用的 Linux 内核源代码版本为 3.2.44，glibc 的源码版本为 2.17。

勘误和支持

由于作者的水平有限，主题又过于宏大，书中难免会出现一些错误或不准确的地方，

如有不妥之处，恳请读者批评指正。如果你发现有什么问题，或者有什么疑问，都可以发邮件至我的邮箱 gfree.wind@gmail.com，期待您的指导！

致谢

首先要感谢伟大的 Linux 内核创始人 Linus，他开创了一个影响世界的操作系统。

其次要感谢机械工业出版社华章公司的编辑杨绣国老师（Lisa），感谢你的魄力，敢于找新人来写作，并敢于信任新人，让其完成这么大的一个项目。感谢你的耐心，正常的一年半的写作时间，被我们生生地延长到了将近三年的时间，感谢你在写作过程中对我们的鼓励和帮助。

然后要感谢我的搭档李彬，在我加入当前的创业公司后，只有很少的空闲时间和精力来投入写作。这时，是李彬在更紧张的时间内，承担了本书的一半内容。并且其写作态度极其认真，对质量精益求精。没有李彬的加入，本书很可能就半途而废了。再次感谢李彬，我的好搭档。

最后我要感谢我的亲人。感谢我的父母，没有你们的培养，绝没有我的今天；感谢我的妻子，没有你的支持，就没有我事业上的进步；感谢我的岳父岳母对我女儿的照顾，使我没有后顾之忧；最后要感谢的是我可爱的女儿高一涵小天使，你的诞生为我带来了无尽的欢乐和动力！

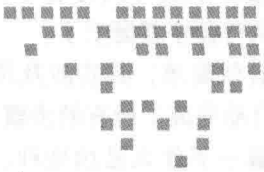
谨以此书，献给我最亲爱的家人，以及众多热爱 Linux 的朋友们。

高　峰

中国北京

2016 年 3 月

目　录 *Contents*



第 0 章 Chapter 0

基础知识

基础知识是构建技术大厦不可或缺的稳定基石，因此，本书首先来介绍一下书中所涉及的一些基础知识。这里以第 0 章命名，表明我们要注重基础，从 0 开始，同时也是向伟大的 C 语言致敬。

基础知识看似简单，但是想要真正理解它们，是需要花一番功夫的。除了需要积累经验以外，更需要对它们进行不断的思考和理解，这样，才能写出高可靠性的程序。这些基础知识很多都可以独立成文，限于篇幅，这里只能是简单的介绍，都是笔者根据自己的经验和理解进行的总结和概括，相信对读者会有所帮助。感兴趣的朋友可以自己查找更多的资料，以得到更准确、更细致的介绍。

注意 本书中的示例代码为了简洁明了，没有考虑代码的健壮性，例如不检查函数的返回值、使用全局变量等。

0.1 一个 Linux 程序的诞生记

一本编程书籍如果开篇不写一个“hello world”，就违背了“自古以来”的传统了。因此本节也将以 hello world 为例来说明一个 Linux 程序的诞生过程，示例代码如下：

```
#include <stdio.h>

int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

下面使用 gcc 生成可执行程序：gcc -g -Wall 0_1_hello_world.c -o hello_world。这样，一个 Linux 可执行程序就诞生了。

整个过程看似简单，其实涉及预处理、编译、汇编和链接等多个步骤。只不过 gcc 作为一个工具集自动完成了所有的步骤。下面就分别来看看其中所涉及的各个步骤。

首先来了解一下什么是预处理。预处理用于处理预处理命令。对于上面的代码来说，唯一的预处理命令就是 #include。它的作用是将头文件的内容包含到本文件中。注意，这里的“包含”指的是该头文件中的所有代码都会在 #include 处展开。可以通过“gcc -E 0_1_hello_world.c”在预处理后自动停止后面的操作，并把预处理的结果输出到标准输出。因此使用“gcc -E 0_1_hello_world.c > 0_1_hello_world.i”，可得到预处理后的文件。

理解了预处理，在出现一些常见的错误时，才能明白其中的原因。比如，为什么不能在头文件中定义全局变量？这是因为定义全局变量的代码会存在于所有以 #include 包含该头文件的文件中，也就是说所有的这些文件，都会定义一个同样的全局变量，这样就不可避免地造成了冲突。

编译环节是指对源代码进行语法分析，并优化产生对应的汇编代码的过程。同样，可以使用 gcc 得到汇编代码，而非最终的二进制文件，即“gcc -S 0_1_hello_world.c -o 0_1_hello_world.s”。gcc 的 -S 选项会让 gcc 在编译完成后停止后面的工作，这样只会产生对应的汇编文件。

汇编的过程比较简单，就是将源代码翻译成可执行的指令，并生成目标文件。对应的 gcc 命令为“gcc -c 0_1_hello_world.c -o 0_1_hello_world.o”。

链接是生成最终可执行程序的最后一个步骤，也是比较复杂的一步。它的工作就是将各个目标文件——包括库文件（库文件也是一种目标文件）链接成一个可执行程序。在这个过程中，涉及的概念比较多，如地址和空间的分配、符号解析、重定位等。在 Linux 环节下，该工作是由 GNU 的链接器 ld 完成的。

实际上我们可以使用 -v 选项来查看完整和详细的 gcc 编译过程，命令如下。

```
gcc -g -Wall -v 0_1_hello_word.c -o hello_world。
```

由于输出过多，此处就不粘贴结果了。感兴趣的朋友可以自行执行命令，查看输出。通过 -v 选项，可以看到 gcc 在背后做了哪些具体的工作。

0.2 程序的构成

Linux 下二进制可执行程序的格式一般为 ELF 格式。以 0.1 节的 hello world 为例，使用 readelf 查看其 ELF 格式，内容如下：

```
ELF Header:
Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class: ELF32
Data: 2's complement, little endian
Version: 1 (current)
```

```
OS/ABI: UNIX - System V
ABI Version: 0
Type: EXEC (Executable file)
Machine: Intel 80386
Version: 0x1
Entry point address: 0x8048320
Start of program headers: 52 (bytes into file)
Start of section headers: 5148 (bytes into file)
Flags: 0x0
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes)
Number of program headers: 9
Size of section headers: 40 (bytes)
Number of section headers: 36
Section header string table index: 33
Section Headers:
[Nr] Name Type Addr Off Size ES Flg Lk Inf Al
[ 0] NULL 00000000 000000 000000 00 0 0 0
[ 1] .interp PROGBITS 08048154 000154 000013 00 A 0 0 1
[ 2] .note.ABI-tag NOTE 08048168 000168 000020 00 A 0 0 4
[ 3] .note.gnu.build-i NOTE 08048188 000188 000024 00 A 0 0 4
[ 4] .gnu.hash GNU_HASH 080481ac 0001ac 000020 04 A 5 0 4
[ 5] .dynsym DYNSYM 080481cc 0001cc 000050 10 A 6 1 4
[ 6] .dynstr STRTAB 0804821c 00021c 00004a 00 A 0 0 1
[ 7] .gnu.version VERSYM 08048266 000266 00000a 02 A 5 0 2
[ 8] .gnu.version_r VERNEED 08048270 000270 000020 00 A 6 1 4
[ 9] .rel.dyn REL 08048290 000290 000008 08 A 5 0 4
[10] .rel.plt REL 08048298 000298 000018 08 A 5 12 4
[11] .init PROGBITS 080482b0 0002b0 000024 00 AX 0 0 4
[12] .plt PROGBITS 080482e0 0002e0 000040 04 AX 0 0 16
[13] .text PROGBITS 08048320 000320 000188 00 AX 0 0 16
[14] .fini PROGBITS 080484a8 0004a8 000015 00 AX 0 0 4
[15] .rodata PROGBITS 080484c0 0004c0 000015 00 A 0 0 4
[16] .eh_frame_hdr PROGBITS 080484d8 0004d8 000034 00 A 0 0 4
[17] .eh_frame PROGBITS 0804850c 00050c 0000c4 00 A 0 0 4
[18] .init_array INIT_ARRAY 08049f08 000f08 000004 00 WA 0 0 4
[19] .fini_array FINI_ARRAY 08049f0c 000f0c 000004 00 WA 0 0 4
[20] .jcr PROGBITS 08049f10 000f10 000004 00 WA 0 0 4
[21] .dynamic DYNAMIC 08049f14 000f14 0000e8 08 WA 6 0 4
[22] .got PROGBITS 08049ffc 000ffc 000004 04 WA 0 0 4
[23] .got.plt PROGBITS 0804a000 001000 000018 04 WA 0 0 4
[24] .data PROGBITS 0804a018 001018 000008 00 WA 0 0 4
[25] .bss NOBITS 0804a020 001020 000004 00 WA 0 0 4
[26] .comment PROGBITS 00000000 001020 00006b 01 MS 0 0 1
[27] .debug_aranges PROGBITS 00000000 00108b 000020 00 0 0 1
[28] .debug_info PROGBITS 00000000 0010ab 000094 00 0 0 1
[29] .debug_abbrev PROGBITS 00000000 00113f 000044 00 0 0 1
[30] .debug_line PROGBITS 00000000 001183 000043 00 0 0 1
[31] .debug_str PROGBITS 00000000 0011c6 0000cb 01 MS 0 0 1
[32] .debug_loc PROGBITS 00000000 001291 000038 00 0 0 1
[33] .shstrtab STRTAB 00000000 0012c9 000151 00 0 0 1
[34] .symtab SYMTAB 00000000 0019bc 000490 10 35 51 4
[35] .strtab STRTAB 00000000 001e4c 00025a 00 0 0 1
```

由于输出过多，后面的结果并没有完全展示出来。ELF 文件的主要内容就是由各个

section 及 symbol 表组成的。在上面的 section 列表中，大家最熟悉的应该是 text 段、data 段和 bss 段。text 段为代码段，用于保存可执行指令。data 段为数据段，用于保存有非 0 初始值的全局变量和静态变量。bss 段用于保存没有初始值或初值为 0 的全局变量和静态变量，当程序加载时，bss 段中的变量会被初始化为 0。这个段并不占用物理空间——因为完全没有必要，这些变量的值固定初始化为 0，因此何必占用宝贵的物理空间？

其他段没有这三个段有名，下面来介绍一下其中一些比较常见的段：

- debug 段：顾名思义，用于保存调试信息。
- dynamic 段：用于保存动态链接信息。
- fini 段：用于保存进程退出时的执行程序。当进程结束时，系统会自动执行这部分代码。
- init 段：用于保存进程启动时的执行程序。当进程启动时，系统会自动执行这部分代码。
- rodata 段：用于保存只读数据，如 const 修饰的全局变量、字符串常量。
- symtab 段：用于保存符号表。

其中，对于与调试相关的段，如果不使用 -g 选项，则不会生成，但是与符号相关的段仍然会存在，这时可以使用 strip 去掉符号信息，感兴趣的朋友可以自己参考 strip 的说明进行实验。一般在嵌入式的产品中，为了减少程序占用的空间，都会使用 strip 去掉非必要的段。

0.3 程序是如何“跑”的

在日常工作中，我们经常会说“程序“跑”起来了”，那么它到底是怎么“跑”的呢？在 Linux 环境下，可以使用 strace 跟踪系统调用，从而帮助自己研究系统程序加载、运行和退出的过程。此处仍然以 hello_world 为例。

```
strace ./hello_world
execve("./hello_world", ["./hello_world"], [/* 59 vars */]) = 0
brk(0) = 0x872a000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
    0xb7778000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=80063, ...}) = 0
mmap2(NULL, 80063, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7764000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0000\226\1\0004\0\0\0"...,
    512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1730024, ...}) = 0
mmap2(NULL, 1743580, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0)= 0xb75ba000
mprotect(0xb775d000, 4096, PROT_NONE) = 0
mmap2(0xb775e000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_
```

```
    DENYWRITE, 3, 0x1a3) = 0xb775e000
mmap2(0xb7761000, 10972, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_
    ANONYMOUS, -1, 0) = 0xb7761000
close(3) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
    = 0xb75b9000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb75b9900, limit:1048575,
    seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0,
    useable:1}) = 0
mprotect(0xb775e000, 8192, PROT_READ) = 0
mprotect(0x8049000, 4096, PROT_READ) = 0
mprotect(0xb779b000, 4096, PROT_READ) = 0
munmap(0xb7764000, 80063) = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
    0xb7777000
write(1, "Hello world!\n", 13Hello world!
) = 13
exit_group(0) = ?
```

下面就针对 strace 输出说明其含义。在 Linux 环境中，执行一个命令时，首先是由 shell 调用 fork，然后在子进程中来真正执行这个命令（这一过程在 strace 输出中无法体现）。strace 是 hello_world 开始执行后的输出。首先是调用 execve 来加载 hello_world，然后 ld 会分别检查 ld.so.nohwcap 和 ld.so.preload。其中，如果 ld.so.nohwcap 存在，则 ld 会加载其中未优化版本的库。如果 ld.so.preload 存在，则 ld 会加载其中的库——在一些项目中，我们需要拦截或替换系统调用或 C 库，此时就会利用这个机制，使用 LD_PRELOAD 来实现。之后利用 mmap 将 ld.so.cache 映射到内存中，ld.so.cache 中保存了库的路径，这样就完成了所有的准备工作。接着 ld 加载 c 库——libc.so.6，利用 mmap 及 mprotect 设置程序的各个内存区域，到这里，程序运行的环境已经完成。后面的 write 会向文件描述符 1（即标准输出）输出 "Hello world!\n"，返回值为 13，它表示 write 成功的字符个数。最后调用 exit_group 退出程序，此时参数为 0，表示程序退出的状态——此例中 hello-world 程序返回 0。

0.4 背景概念介绍

0.4.1 系统调用

系统调用是操作系统提供的服务，是应用程序与内核通信的接口。在 x86 平台上，有多种陷入内核的途径，最早是通过 int 0x80 指令来实现的，后来 Intel 增加了一个新的指令 sysenter 来代替 int 0x80——其他 CPU 厂商也增加了类似的指令。新指令 sysenter 的性能消耗大约是 int 0x80 的一半左右。即使是这样，相对于普通的函数调用来说，系统调用的性能消耗也是巨大的。所以在追求极致性能的程序中，都在尽力避免系统调用，譬如 C 库的 gettimeofday 就避免了系统调用。

用户空间的程序默认是通过栈来传递参数的。对于系统调用来说，内核态和用户态使用的是不同的栈，这使得系统调用的参数只能通过寄存器的方式进行传递。