Linux内核编码风格《中文翻译版》

这是一篇用来描述 linux 内核的首选编码样式的短文档。每个人都有自己的编码风格，我不会 将我的观点强加到任何人的身上，但这正是我所要保持的东西，就像其他许多事情一样。至少请考虑在这里所列出的观点。

首先，我建议打印出GNU编码标准的副本，不要去阅读，直接将它烧毁。这是一个伟大的象征性的姿态。

好，现在正式开始：

# 第1章 缩进

一个Tab键有8个字符位因此一个缩进也是8个字符位 . 有人试图将一个缩进定义场4个字符位甚至2个 , 这无异于试图将Pi的值定义为3.

说明: 缩进的意义在于定义语句块的开始和结尾 .尤其是当你紧盯屏幕20小时后你就更加能体会到缩进的作用了 .

　　现在有些人说8字符长的缩进使得代码太靠右边，且很难在80字符的终端窗口阅读。事实是，如果你的代码需要三层以上的缩进，这是你犯糊涂了，你得修改你的程序。

　　总之，8字符缩进会让程序更好阅读，而而外的好处则是能够提醒你你的程序嵌套的太深了。这个需要警惕。

　　精简 switch 语句中缩进界级别的最好方式就是将"switch"和其下级的"case"放在同一列，而不是双重缩进"case"标签。例如：

    switch (suffix) {

    case 'G':

    case 'g':

        mem <<= 30;

        break;

    case 'M':

    case 'm':

        mem <<= 20;

        break;

    case 'K':

    case 'k':

        mem <<= 10;

        /\* fall through \*/

    default:

        break;

    }

不要将多个语句放在同一行，除非你要掩饰什么：

if (condition) do\_this;

do\_something\_everytime;

　　同样也不要将多个赋值放在同一行。内核编程范式超简单。避免复杂的表达式。

　　还有一个潜规则，是 Kconfig 推荐的，永远不要用空格缩进，上面的例子是故意的。

　　找一个合适的编辑器，记得不要在行末留有空格。

# 第2章 换行

编程范式的意义在于使用常见工具时的可阅读和可维护性。

行的长度限制为80列，这是强烈推荐的设置。

多于80列的语句将被分为合适的数块。子句应该永远比主句短，且比主句更靠右侧。这对有较长参数表的函数声明同样有效。长字符串同样也被打断为短字符串。这样做能在超过80列时提高可阅读性，且不会隐藏信息。

void fun(int a, int b, int c)

{

    if (condition)

        printk(KERN\_WARNING "Warning this is a long printk with "

                        "3 parameters a: %u b: %u "

                        "c: %u \n", a, b, c);

    else

        next\_statement;

}

# 第3章 大括号和空格

关于 C 风格编程的另一个议点就是大括号。跟缩进值不同，大括号的放置并没有技术因素在内，但“先贤“ Kernighan 和 Ritchie 展示为我们的最好方式，是将左大括号放置在行末尾。而右大括号放置在行首，如下：

if (x is true) {

we do y

}

对非函数的语句块也是如此（if、switch、for、while、do）。例如:

switch (action) {

case KOBJ\_ADD:

return "add";

case KOBJ\_REMOVE:

return "remove";

case KOBJ\_CHANGE:

return "change";

default:

return NULL;

}

当然那，有一个特殊的例外， 即函数：它的左大括号在新行的行首，如下：

int function(int x)

{

body of function

}

全世界有不同意见的人都认为此不一致的地方——好吧——很是缺乏一致性，不过只要是能正常思考的都知道《C程序设计语言（第一版）》中是 *right* 而《C程序设计语言（第二版）》中则是 right。无论如何，函数都是特殊的（在 C 语言中你不能嵌套定义它们）。

注意，右大括号单独一行， 除非后面跟着的是同一个语句。例如 do 语句中的 "while"或 if 语句中的 "else" 如下：

do {

body of do-loop

} while (condition);

和

if (x == y) {

..

} else if (x > y) {

...

} else {

....

}

阐述：K&R, 《C程序设计语言》一书的缩写。

同时要注意，这种放置大括号的方式能够在不影响可读性的同时，有效减少空行（或近乎空行）的数量。因此，你不需要刷新资源就可以看倒更多行（想想一下只能显示25行的终端窗口），且你能够有空多的空行来安置注释。

只有一行语句时不用添加多余的大括号。

if (condition)

action();

和

if (condition)

do\_this();

else

do\_that();

判断语句中只有一个分支为单行语句是，不支持这样用，你需要在所有分支中都加入大括号。

if (condition) {

do\_this();

do\_that();

} else {

otherwise();

}

## 3.1 空格

linux内核风格下的空格，其实际应用主要是取决于函数标识符的使用。大多数函数在其标识符后面会加上空格。当然，sizeof, typeof, alignof, 和 \_\_attribute\_\_，像这样的长得像函数（但是因为不需要，所以经常屁股后面不跟着圆括号。例如："sizeof info" 如果在 "struct fileinfo info;"这个声明之后……木有括号……）的，就不需要加空格了。

因此你就知道，在如下关键字后面加个空格：

   if, switch, case, for, do, while

而 sizeof, typeof, alignof, 或者 \_\_attribute\_\_后面就免了吧，例如

s = sizeof(struct file);

当声明指针或者声明一个返回指针的函数时，最好是在把个小小的“\*”离着标识符（不论是变量常量还是函数）近一点，而不是和数据类型近一点，如：

char \*linux\_banner;

unsigned long long memparse(char \*ptr, char \*\*retptr);

char \*match\_strdup(substring\_t \*s);

在双目或者三目运算符周围（左边和右边）用上空格——他们包括（译者注：可能不止，并未对此进行仔细检查）：

=  +  -  <  >  \*  /  %  |  &  ^  <=  >=  ==  !=  ?  :

但是单目运算符就算了，比如这么几个家伙（译者注：同上，未作检查）：

&  \*  +  -  ~  !  sizeof  typeof  alignof  \_\_attribute\_\_  defined

还有几个特殊的，比如自增自减运算符，和他们进行运算的变量标识符中间（译者注：原文这里是说，前后都不用加空格）不用加空格。就是这两个家伙：

  ++  --

然后，“.”和“->”这两个运算符，加括号是作死的行为，你不这么想么？

别在一行的末尾留几个空格。一些具有智能缩进功能的编辑器会在新行的开头适当的插上空格，然后你就可以立即继续写你的程序。但是部分编辑器不会自动删除你程序每一行末尾的空格（虽然你的程序在那几个空格之前已经结束了），甚至这会产生一个完全空白的行，期间充斥着空格这种可恶的东西。

Git在这种情况下会对你进行警告，并提醒你是否由它来为你消灭这些恼人的小东西；但是这种修补总是比不上你自己进行的修补，如果你让Git干了太多这样的活，可能导致你程序行的错乱（所谓进退失据）。

# 第4章 命名

C 语言是粗犷的，你的命名同样如此。与 Modula-2 和 Pascal 不同，C 程序员不使用类似 ThisVariableIsATemporaryCounter 这样有趣的命名。C 程序员会将变量命名为类似 "tmp"的名字，这种名字比较好写，且不难理解。

当然，虽然混合大小写的难以让人接受，但对于全局变量，一个描述性的名字却是必须的。调用一个名为 "foo" 全局函数显然是在找不自在。

全局变量（只有你 真正需要的时候，再用它）需要一个描述性的名字，全局函数也是。如果你有一个计算活跃用户数量的函数，那么命名其为 "count\_active\_users()" 或者类似的名字，而 非 "cntusr()".

将函数类型添加到其名字中（即匈牙利命名法）是有害大脑的——编译器知道、也会检查它的，这只能混淆程序员。所以微软才会生产那么多充满BUG的程序。

局部变量的名字应当简洁了当。如果在循环中需要一些随机数字，你大可以命名其为 "i" 。只要不会产生歧义，命名为 "loop\_counter" 毫无意义。同样的，"tmp" 可以是任何类型的临时变量。

如果你担心弄混局部变量的名字，那你有另一个问题：函数增长荷尔蒙失调综合症。

参见第6章（函数）

# 第5章 Typedefs

不要傻呼呼地使用像"vps\_t"这样的变量类型.

使用typedef来重定义已有结构体和指针本身就是个错误. 当你在源代码中见到这样的定义:

    vps\_t a;

天知道a到底是个什么东西!

如果你看到这样的定义:

   struct virtual\_container \*a;

你完全可以一目然: 哦, a是一个指向...的指针.

多数人都觉得typedefs可以提高可阅读性, 但真理往往掌握在少数人手中. Typedefs只有在如下情况下有用:

**(a) 需要被封装起来的对象(你本来就打算隐藏起类型信息)**

      如: "pte\_t"这种类型.  封装出这样的类型本来就只打算让特定的"访问函数"才能访问.

      请注意: 封装以及"访问函数"本来就不是什么好东西.  The reason we have them for things like pte\_t etc. is that there really is absolutely \_zero\_ portably accessible information there.

**(b) 定长的整数类型.**

这样可以在避免在某些情况下, 搞不清楚到底用的是int还是long, 把你自己搞晕!

      u8/u16/u32都是完美的typdefs, 尽管更应该它们归结至规则(d)下.

      再次重申: 这样定义必须要有合理的理由. 如果某个变量本来就是unsigned long类型, 你硬要它定义成这样,你就是SB了:

     typedef unsigned long myflags\_t;

     如果你有明确的理由, 在某种情形下变量是unsigned int类型, 而在另外的情形下又要变身成为unsigned long类型, 那就尽管去typedef.

**(c) 当你需要使用kernel的sparse工具做变量类型检查时,  你也可以typedef一个类型.**

(d)对于特殊情况下的某些c99标准的新类型

你的大脑和眼睛只需要很短的时间就可以习惯像'uint32\_t'这样的新类型，虽然有的人反对这宗用法。

因此，虽然对于你的新代码来说，linux独有的'u8/u16/u32/u64'类型并不是强制的，但是他们也是与标准类型等价的。

当编辑现有代码时，如果其中已经使用了某一种类型名规范，你应该遵循原样，使用与之相同的类型名。

**(e)用户空间中的类型安全**

对于某些结构，显然我们不能使用c99标准的类型，不能使用上述的‘u32’，因此咱干脆在结构中使用'\_u32'或者类似的类型好了。

也许还有其它情况，但基本规则是，除非你很清除符合某一条规则，否则 永远不要使用 typedef。

通常，一个指针或是一个含有元素的结构体，若能直接访问，永远不是 typedef。

# 第6章 函数

函数这种东西，应该小而精，换句话说，只是专心的做一件事情一直到底。如果你把它赤身裸体的展示于屏幕之上，你应该在两个屏幕（这里一个屏幕的大小根据 ISO/ANSI标准应该是80X24的）之内就可以看完它。

函数的长度和缩进程度和它的复杂程度是成反比的。所以，如果你的函数确实单纯（译者注：是萌妹子那样的单纯不？）而简单，比如用一个长长的switch-case语句来做点简简单单的事情来处理一些单单纯纯的情况，来吧，咱把函数写再长一点。

相反的，如果你的函数相当的复杂，复杂到你严重怀疑一个普普通通的不像你一样天才的高中学生看不懂……返工吧，把函数缩短缩短再缩短，不要犹豫，多造几个辅助函数并给他们几个一看即知的名字，以减少函数复杂性。（当然，如果你觉得这个函数太关键了，如果拆开了写肯定会影响整个程序的性能，那你干脆内联，用那个内联函数或者使用编译器的内联功能（译者注：内联函数在c99标准里面出现，现在gcc不知道支持内联函数否））

规范对于函数的另外一个要求是关于局部变量的个数的：最多5-10个。就算你是天才，你的脑子一般也就能轻松关注7个变量，再多了就绝对会出现一些你意想不到的错误（谁叫你一心多用来着）。好吧，就算你是天才，如果还想明白你在两周之前所写的那些程序的话，还是遵守这些规范吧。

就源代码来说，函数与函数的原型之间应该有一个空白行作为分隔。如果该函数在其他文件中被引用，那么他的 EXPORT\*宏应当和函数最后一行的那个大括号中间没有任何空格。例如：

int system\_is\_up(void)

{

return system\_state == SYSTEM\_RUNNING;

}

EXPORT\_SYMBOL(system\_is\_up);

另外，在函数原型中，当声明参量时应该将参量的类型和标识符放到一起，虽然c语言本身并不要求这种形式，但是在linux kernal里面是要求的——目的是增加每一行代码的信息量。

# 第7章 集中于一处退出函数

虽然很多人不提倡，但是我们这里要经常使用goto进行无条件的跳转。

当函数有很多个出口，使用goto把这些出口集中到一处是很方便的，特别是函数中有许多重复的清理工作的时候。

理由是：

-无条件跳转易于理解

-可以减少嵌套

-可以避免那种忘记更新某一个出口点的问题

-算是帮助编译器做了代码优化

int fun(int a)

{

int result = 0;

char \*buffer = kmalloc(SIZE);

if (buffer == NULL)

return -ENOMEM;

if (condition1) {

while (loop1) {

...

}

result = 1;

goto out;

}

...

out:

kfree(buffer);

return result;

}

# 第8章 注释

有注释当然是好的，但是注释太多就很恶心了。千万不要在注释里面解释你的程序怎么运行的。相对于尝试用注释解释清楚你那恶心的代码，你还不如就写个清晰易懂（译者注：就是小而精，萌妹子一样单纯的~）的函数。

一般的，你的注释是用来说明这段代码是“干啥的”，而不是“怎么干的”。另外，别把注释放到你的函数体里面（译者注：把我放到妹子的怀抱里吧！）：如过你的函数确实复杂到需要用注释来分隔成几段，回第六章擦亮眼睛再看两遍（译者注：那一段正好也是我翻译的）。也可以用几个小注释来提醒大家一些东西（“写的好~”或者“写的真tm糟糕”），但是就不要自取其辱评论自己的东西了……最后，仍然提醒你，一定是把注释安放在函数的前面，然后简单写下这段函数式干啥的，如果可能，你倒是不妨提及为什么你要这样做。

当给kernal api函数注释的时候，请使用kernal-doc格式。该格式的细节你可以参考这两个文件：Documentation/kernel-doc-nano-HOWTO.txt和scripts/kernel-doc（译者注：这个文件路径应当是指内核源码中的路径）

linux当中的注释是c89格式（"/\* ... \*/"）的，而不是c99中新近添加的"// ..."

多于一行（多行）的注释应当准从以下格式：

/\*

\* This is the preferred style for multi-line

\* comments in the Linux kernel source code.

\* Please use it consistently.

\*

\* Description: A column of asterisks on the left side,

\* with beginning and ending almost-blank lines.

\*/

该格式对于注释标识符（常量，变量，函数等）同样适用。换句话说，你最好不要再一行里面同时声明很多个标识符（无论是用逗号还是分号隔开都是不推荐的），一行一个就可以了。这样你就可以在每一行对每一个标识符进行解释。

# 第9章 俺犯错了……

没关系，咱都这么干过。你的那些unix老鸟朋友们可能告诉过你“GNU emacs”这个编辑器能自动给你的c源代码进行排版，然后你貌似可以放心大胆的享受这一切。(⊙o⊙)………没错，这东西确实可以，但是他默认的排版方式极其糟糕（事实上，就算你不管格式一通乱敲也比他好看）。

所以，干脆砸掉你的emacs，要么就取消他的自动排版功能。如果你不想砸了而是想取消那个自动排版功能的话，请在你的.emcas文件里面敲入这么一串东西（译者注：这是emcas lisp,lisp的一个方言，我没有在代码插入功能里面找到lisp的相关选项，请红薯修改）：

(defun c-lineup-arglist-tabs-only (ignored)

   "Line up argument lists by tabs, not spaces"

   (let\* ((anchor (c-langelem-pos c-syntactic-element))

      (column (c-langelem-2nd-pos c-syntactic-element))

      (offset (- (1+ column) anchor))

      (steps (floor offset c-basic-offset)))

     (\* (max steps 1)

        c-basic-offset)))

(add-hook 'c-mode-common-hook

           (lambda ()

             ;; Add kernel style

             (c-add-style

              "linux-tabs-only"

              '("linux" (c-offsets-alist

                         (arglist-cont-nonempty

                          c-lineup-gcc-asm-reg

                          c-lineup-arglist-tabs-only))))))

(add-hook 'c-mode-hook

           (lambda ()

             (let ((filename (buffer-file-name)))

               ;; Enable kernel mode for the appropriate files

               (when (and filename

                          (string-match (expand-file-name "~/src/linux-trees")

                                        filename))

                 (setq indent-tabs-mode t)

                 (c-set-style "linux-tabs-only")))))

这将使你在emcas上面写的c程序符合我们的规范

好吧……如果你仍然无法使你的emacs按照咱的意思来，世界崩溃不了，你可以用下“indent”。

好吧，世界继续崩溃，GNU indent有时候也和GNU emacs一样脑残，这倒是也可以解释为啥这么个小玩意儿居然带上了几个命令行选项。当然，这不算太糟糕，因为GNU indent的开发者只是想强调K&R格式的权威性（GNU表示自己是天然呆，只是无意中误导了大家），所以你可以用选项 "-kr -i8"（意思是"K&R, 8 character indents"，K&R格式，八空格缩进），或者干脆去改indent的配置文件。

当然，indent的选项太多了，所以你在配置注释内容的时候或许需要看下indent的手册，但是你要记住，indent不是你放弃自己人工控制格式的免死金牌。

# 第10章 Kconfig配置文件

在linux kernal整个的源代码里面，Kconfig文件有几处明显的不一致。config定义线以下的地方缩进一个tab，但是帮助字段在此基础上再缩进两个空格。例如：

config AUDIT

     bool "Auditing support"

     depends on NET

     help

       Enable auditing infrastructure that can be used with another

       kernel subsystem, such as SELinux (which requires this for

       logging of avc messages output).  Does not do system-call

       auditing without CONFIG\_AUDITSYSCALL.

对于这段文字里让你认为不可靠的地方，应当参照实践情况：

 config SLUB

     depends on EXPERIMENTAL && !ARCH\_USES\_SLAB\_PAGE\_STRUCT

     bool "SLUB (Unqueued Allocator)"

     ...

当有一些危险而严重的问题（比如一些文件系统不支持你的“写”操作）应当像下面那样进行提示：

config ADFS\_FS\_RW

     bool "ADFS write support (DANGEROUS)"

     depends on ADFS\_FS

     ...

要查看更多细节请看Documentation/kbuild/kconfig-language.txt.

# 第11 章 数据结构

在单线程运行环境中，数据结构的创建与销毁是可见的，可以对其引用进行计数的。同时，在kernal中，并不存在一个内存回收机制（kernal外的机制？你想让系统变的多么慢？），这意味着你必须小心翼翼的对指针的使用进行控制。

对指针进行计数意味着你可以避免死锁，同时允许多线程对你的数据结构进行存取-因此也可以不用担心结构因为等待和中断突然没了踪迹。

请注意"锁"并不是"引用计数"的替代品."锁"的使用是为了保证并发访问时数据的一致性, 而"引用计数"则一种内存管理技术. 通常这两者都是需要的, 不要搞混了.

很多数据结构在有多个不同类别的使用者时会引入二级引用计数机制. 第二级的引用计数用于统计各个不同类别的使用者的数量, 当某个二级引用计算到零时, 第一级的引用计数才会减一.

这种“多级引用计数”的例子能在内存管理中找到  ("struct mm\_struct": mm\_users 和 mm\_count)，在文件系统代码中也有 ("struct super\_block": s\_count 和 s\_active).

记住：如果另有线程能够访问你的数据结构，而你却没有一个计数的参数，基本来说这就是一个BUG。

# 第12章 宏，枚举和RTL

定义常量的宏名字和枚举变量是大写的。

#define CONSTANT 0x12345

当定义一些相关联常量的时候选择用枚举是一个比较好的选择。

全部为大写的宏名称是很好的，但有些和函数相似的宏名称也可以定义为小写的。

通常，把内联函数被定义为宏是比较好的。

多个语句的宏应该被定义在一个do-while循环体里：

#define macrofun(a, b, c)             \

     do {                             \

         if (a == 5)                  \

             do\_this(b, c);           \

     } while (0)

**使用宏时应该避免的情况：**

**1）控制流可变的宏**

#define FOO(x)                    \

     do {                         \

         if (blah(x) < 0)         \

             return -EBUGGERED;   \

     } while(0)

这是一个非常坏的想法。虽然这个宏看起来像一个函数调用，但在这个宏里存在调用别的函数；这些额外的调用会打断那些读代码的人的思路。

**2）宏依赖奇怪名称的本地变量**

#define FOO(val) bar(index, val)

这看起来可能是很好的东西，但是在另一个人读这段代码的时候会引起困惑，而且这种代码也留下了隐患。例如在更改的时候这可能引起错误，即使这个更改看起来似乎没有错。

3）带有作为左值使用的参数的宏：

FOO(x) x = y;

如果有人把FOO当作内联函数使用，就会引起错误。

4）忘记优先级：

用表达式定义宏常量的时候，一定要用换括号括上这个表达式。但也要注意这个和使用带有参数的宏的相似之处。

#define CONSTANT 0x4000

#define CONSTEXP (CONSTANT|3)

# 第13章 打印内核信息

内核开发者通常被视为学者。他们编写内核信息时，好的拼写能够留下良好的印象。不要使用简写，使用 "do not" 或 "don't" 而非 "dont"。保持信息简洁明了。

内核信息不需要使用句点结尾。

在原括号(%d)中打印数字没有任何意义，应该避免。

在  <linux device.h> 中有许多驱动模型诊断宏，你应该使用它们来确保消息匹配正确的设备和驱动 ，并正确的标记它们的级别： dev\_err()、dev\_warn()、dev\_info()等等。对于没有关联特定设备的消息，<linux/printk.h> 中定义了 pr\_debug() 和 pr\_info()。

编写良好的调试信息是一个极大的挑战;一旦你完成了它们，它们将对你远程排错提供机器巨大的帮助。这些信息应该在DEBUG标志（这个并非默认包含的）定义之前就编撰完成。当你使用 dev\_dbg() 或 pr\_debug()，它们将自动运行。很多子系统都有 Kconfig 选项来开启 -DDEBUG。

还有一个惯例是使用 VERBOSE\_DEBUG 为那些已经开启 DEBUG 的用户添加 dev\_vdbg() 消息。

# 第14章 内存分配

内核提供了下列通用内存分配器：kmalloc()、kzalloc()、kcalloc()、vmalloc()、和 vzalloc()。 更多信息，请参阅的 API 文档。

　　传递一个结构体大小的最好方式如下：

p = kmalloc(sizeof(\*p)， ...);

　　另外一种传递方式中，sizeof的操作数是结构体的名字，这样有损可读性，并会在指针类型改变，但传递给内存分配器的大小没有变化时导致BUG。

　　强制转换返回的void指针是冗余的。C语言本身保证了从void指针到其他任何指针类型的转换。

# 第15章 内联弊病

一个很常见的误解就是认为 gcc 提供了一个可以让代码更快运行的选项——内联函数。 虽然有时使用内联函数是恰当的（例如作为一种宏的替代方式，参见章十二），但多数情况下并非如此。"inline" 关键字的泛滥会会使内核变大，从而使整个系统运行速度变慢，因为大内核会占用更多的CPU高速缓存，同时会导致可用内存页缓存减少。想象一下，一次页缓存未命中就会导致一次磁盘寻址，这至少耗费5毫秒。5毫秒足够CPU运行 很多很多的指令。

有一个合理的基本原则，如果一个函数有3行以上的代码，就不要把它变成内联函数。这个原则的一个例外是，若某个参数是一个编译时常数，且你 确定因为这个常量编译器在编译时能 优化掉你的函数的大部分代码，那么加上 inline 关键字。kmalloc()内联函数就是个很好的例子。

人们经常主张可以给只用一次的静态函数加上 inline 关键字，这样不会有任何损失。虽然从技术上来说这样没错，但是实际上 gcc 会自动内联这些函数，而其他用户则可能认为加入 gcc 能够自动完成的功能的代码毫无意义，这将导致维护时的争论。

# 第16章 函数返回值和名称

函数在不同情况下返回不同意义的值，但最常使用的是返回一个值来指示函数是否成功。例如返回整型的错误代码（-Exxx = 失败， 0 = 成功）或者一个表示是否成功的布尔值（0表示失败，非零表示成功）。

混合上面两种返回方法会使代码变得复杂，并且很难找到bug。如果C语言能明确区分整型和布尔型，那么编译器会替我们发现这个问题……但是它不会那么做。为了避免这种问题，一定要谨记如下条例：

如果函数的名称是一个动作，或者一个势在必行的命令，那么函数应该返回一个整型的错误代码。如果函数名类似断言，那么它应该返回一个表示是否成功的布尔值。

例如，“add work”是一个命令， 因此add\_work()函数返回0表示成功，或者返回-EBUSY表示失败。通过相似的方法，“PCI device present”是一个断言，那么pci\_dev\_present()函数返回1表示找到匹配的设备，或者返回0表示没找到。

所有导出的函数一定要遵守这个准则，并且所有的公用函数都应该这样。私有函数（static）则不需要。但我们依然建议你这样做。

而对于另一些函数，他们的返回值是真实的计算结果，而不是指示计算是否成功。通体上看，它们通过返回范围外的值来表示失败。典型的例子就是返回一个指针：使用NULL或者ERR\_PTR来表示错误。

# 第17章 不要重新发明内核宏

头文件 include/linux/kernel.h 包含了一些宏，你应该使用它们，而不是自己写一些它们的 变种。例如，如果你需要计算一个数组的长度，使用下面的宏：

#define ARRAY\_SIZE(x) (sizeof(x) / sizeof((x)[0]))

同样的，如果你要计算某个结构体成员的大小，使用：

#define FIELD\_SIZEOF(t, f) (sizeof(((t\*)0)->f))

如果需要，这里还有可以做严格类型检查的 min() 和max() 宏。仔细看看头文件中还定义了那些东西，只要有定一，你就不要在自己的代码中重新定义它们了。

# 第18章 编辑器模式行和其它琐碎

有些编辑器可以解释嵌在源文件里的，由一些特殊标记标明的配置信息。例如，emacs　能够解释被标记成这样的行：

-\*- mode: c -\*-

或是这样：

/\*

Local Variables:

compile-command: "gcc -DMAGIC\_DEBUG\_FLAG foo.c"

End:

\*/

Vim则能解释如下的标记：

/\* vim:set sw=8 noet \*/

不要在源代码中包含任何类似的内容。每个人都有自己的编辑器配置，你的源文件不应 该覆盖它们。这包括有关缩进和模式配置的标记。人们要能使用他们自定的模 式，或者使用其它可以产生正确的缩进的方法。

# 附录一：参考文献

C语言程序设计（第二版）

Brian W. Kernighan 和 丹尼斯·里奇

Prentice Hall出版社，1988年

ISBN 0-13-110362-8（平装），0-13-110370-9（精装）

网址：http://cm.bell-labs.com/cm/cs/cbook/

编程的实践

Brian W. Kernighan和Rob Pike

Addison-Wesley出版公司，1999

ISBN 0-201-61586-X

网址：http://cm.bell-labs.com/cm/cs/tpop/

GNU手册 - 在遵守与K＆R和本文 - CPP，GCC

海湾合作委员会内部和缩进，都可以从http://www.gnu.org/manual/

WG14是国际标准化工作组的编程

C语言，网址：http://www.open-std.org/JTC1/SC22/WG14/

内核编程规范，2002年 在OLS，由greg@kroah.com：

http://www.kroah.com/linux/talks/ols\_2002\_kernel\_codingstyle\_talk/html/