Chinese translated version of Documentation/CodingStyle

If you have any comment or update to the content, please post to LKML directly.
However, if you have problem communicating in English you can also ask the
Chinese maintainer for help. Contact the Chinese maintainer, if this
translation is outdated or there is problem with translation.

Chinese maintainer: Zhang Le <r0bertz@gentoo.org>

Documentation/CodingStyle的中文翻译

如果想评论或更新本文的内容，请直接发信到LKML。如果你使用英文交流有困难的话，也可以向中文版维护者求助。如果本翻译更新不及时或者翻译存在问题，请联系中文版维护者。

中文版维护者： 张乐 Zhang Le <r0bertz@gentoo.org>
中文版翻译者： 张乐 Zhang Le <r0bertz@gentoo.org>
中文版校译者： 王聪 Wang Cong <xiyou.wangcong@gmail.com>
                wheelz <kernel.zeng@gmail.com>
                管旭东 Xudong Guan <xudong.guan@gmail.com>
                Li Zefan <lizf@cn.fujitsu.com>
                Wang Chen <wangchen@cn.fujitsu.com>
以下为正文

Linux内核代码风格

这是一个简短的文档，描述了linux内核的首选代码风格。代码风格是因人而异的，而且我不愿意把我的观点强加给任何人，不过这里所讲述的是我必须要维护的代码所遵守的风格，并且我也希望绝大多数其他代码也能遵守这个风格。请在写代码时至少考虑一下本文所述的风格。

首先，我建议你打印一份GNU代码规范，然后不要读它。烧了它，这是一个具有重大象征性意义的动作。

不管怎样，现在我们开始：

第一章：缩进

制表符是8个字符，所以缩进也是8个字符。有些异端运动试图将缩进变为4（乃至2）个字符深，这几乎相当于尝试将圆周率的值定义为3。

理由：缩进的全部意义就在于清楚的定义一个控制块起止于何处。尤其是当你盯着你的屏幕连续看了20小时之后，你将会发现大一点的缩进会使你更容易分辨缩进。

现在，有些人会抱怨8个字符的缩进会使代码向右边移动的太远，在80个字符的终端屏幕上就很难读这样的代码。这个问题的答案是，如果你需要3级以上的缩进，不管用何种方式你的代码已经有问题了，应该修正你的程序。

简而言之，8个字符的缩进可以让代码更容易阅读，还有一个好处是当你的函数嵌套太深的时候可以给你警告。留心这个警告。

在switch语句中消除多级缩进的首选的方式是让“switch”和从属于它的“case”标签对齐于同
一列，而不要“两次缩进”“case”标签。比如：

```
        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;
        }
```

不要把多个语句放在一行里，除非你有什么东西要隐藏：

```
        if (condition) do_this;
          do_something_everytime;
```

也不要在一行里放多个赋值语句。内核代码风格超级简单。就是避免可能导致别人误读的表达式。

除了注释、文档和Kconfig之外，不要使用空格来缩进，前面的例子是例外，是有意为之。

选用一个好的编辑器，不要在行尾留空格。

第二章：把长的行和字符串打散

代码风格的意义就在于使用平常使用的工具来维持代码的可读性和可维护性。

每一行的长度的限制是80列，我们强烈建议您遵守这个惯例。

长于80列的语句要打散成有意义的片段。每个片段要明显短于原来的语句，而且放置的位置也明显的靠右。同样的规则也适用于有很长参数列表的函数头。长字符串也要打散成较短的字符串。唯一的例外是超过80列可以大幅度提高可读性并且不会隐藏信息的情况。

```
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;
}
```

第三章：大括号和空格的放置

C语言风格中另外一个常见问题是大括号的放置。和缩进大小不同，选择或弃用某种放置策略并没有多少技术上的原因，不过首选的方式，就像Kernighan和Ritchie展示给我们的，是把起始大括号放在行尾，而把结束大括号放在行首，所以：

```
        if (x is true) {
                we do y
        }
```

这适用于所有的非函数语句块（if、switch、for、while、do）。比如：

```
        switch (action) {
        case KOBJ_ADD:
                return "add";
        case KOBJ_REMOVE:
                return "remove";
        case KOBJ_CHANGE:
                return "change";
        default:
                return NULL;
        }
```

不过，有一个例外，那就是函数：函数的起始大括号放置于下一行的开头，所以：

```
        int function(int x)
        {
                body of function
        }
```

全世界的异端可能会抱怨这个不一致性是……呃……不一致的，不过所有思维健全的人都知道（
a）K&R是_正确的_，并且（b）K&R是正确的。此外，不管怎样函数都是特殊的（在C语言中，函数是不能嵌套的）。

注意结束大括号独自占据一行，除非它后面跟着同一个语句的剩余部分，也就是do语句中的“while”或者if语句中的“else”，像这样：

```
        do {
                body of do-loop
        } while (condition);
```

和

```
        if (x == y) {
                ..
        } else if (x > y) {
                ...
        } else {
                ....
        }
```

理由：K&R。

也请注意这种大括号的放置方式也能使空（或者差不多空的）行的数量最小化，同时不失可读性。因此，由于你的屏幕上的新行是不可再生资源（想想25行的终端屏幕），你将会有更多的空行来放置注释。

当只有一个单独的语句的时候，不用加不必要的大括号。

```
if (condition)
```

```
        action();
```

这点不适用于本身为某个条件语句的一个分支的单独语句。这时需要在两个分支里都使用大括号。

```
if (condition) {
        do_this();
        do_that();
} else {
        otherwise();
}
```

3.1：空格

Linux内核的空格使用方式（主要）取决于它是用于函数还是关键字。（大多数）关键字后要加一个空格。值得注意的例外是sizeof、typeof、alignof和__attribute__，这些关键字某些程度上看起来更像函数（它们在Linux里也常常伴随小括号而使用，尽管在C语言里这样的小括号不是必需的，就像“struct fileinfo info”声明过后的“sizeof info”）。

所以在这些关键字之后放一个空格：

```
        if, switch, case, for, do, while
```

但是不要在sizeof、typeof、alignof或者__attribute__这些关键字之后放空格。例如，

```
        s = sizeof(struct file);
```

不要在小括号里的表达式两侧加空格。这是一个反例：

```
        s = sizeof( struct file );
```

当声明指针类型或者返回指针类型的函数时，“*”的首选使用方式是使之靠近变量名或者函
数名，而不是靠近类型名。例子：

```
        char *linux_banner;
        unsigned long long memparse(char *ptr, char **retptr);
        char *match_strdup(substring_t *s);
```

在大多数二元和三元操作符两侧使用一个空格，例如下面所有这些操作符：

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

但是一元操作符后不要加空格：

```
        &  *  +  -  ~  !  sizeof  typeof  alignof  __attribute__  defined
```

后缀自加和自减一元操作符前不加空格：

```
        ++  --
```

前缀自加和自减一元操作符后不加空格：

```
        ++  --
```

“.”和“->”结构体成员操作符前后不加空格。

不要在行尾留空白。有些可以自动缩进的编辑器会在新行的行首加入适量的空白，然后你就可以直接在那一行输入代码。不过假如你最后没有在那一行输入代码，有些编辑器就不会移除已经加入的空白，就像你故意留下一个只有空白的行。包含行尾空白的行就这样产生了。

当git发现补丁包含了行尾空白的时候会警告你，并且可以应你的要求去掉行尾空白；不过如果你是正在打一系列补丁，这样做会导致后面的补丁失败，因为你改变了补丁的上下文。

第四章：命名

C是一个简朴的语言，你的命名也应该这样。和Modula-2和Pascal程序员不同，C程序员不使用类似ThisVariableIsATemporaryCounter这样华丽的名字。C程序员会称那个变量为“tmp”，这样写起来会更容易，而且至少不会令其难于理解。

不过，虽然混用大小写的名字是不提倡使用的，但是全局变量还是需要一个具描述性的名字。称一个全局函数为“foo”是一个难以饶恕的错误。

全局变量（只有当你真正需要它们的时候再用它）需要有一个具描述性的名字，就像全局函数。如果你有一个可以计算活动用户数量的函数，你应该叫它“count_active_users()”或者类似的名字，你不应该叫它“cntuser()”。

在函数名中包含函数类型（所谓的匈牙利命名法）是脑子出了问题——编译器知道那些类型而且能够检查那些类型，这样做只能把程序员弄糊涂了。难怪微软总是制造出有问题的程序。

本地变量名应该简短，而且能够表达相关的含义。如果你有一些随机的整数型的循环计数器，它应该被称为“i”。叫它“loop_counter”并无益处，如果它没有被误解的可能的话。类似的，“tmp”可以用来称呼任意类型的临时变量。

如果你怕混淆了你的本地变量名，你就遇到另一个问题了，叫做函数增长荷尔蒙失衡综合症。请看第六章（函数）。

第五章：Typedef

不要使用类似“vps_t”之类的东西。

对结构体和指针使用typedef是一个错误。当你在代码里看到：

```
vps_t a;
```

这代表什么意思呢？

相反，如果是这样

```
struct virtual_container *a;
```

你就知道“a”是什么了。

很多人认为typedef“能提高可读性”。实际不是这样的。它们只在下列情况下有用：

(a) 完全不透明的对象（这种情况下要主动使用typedef来隐藏这个对象实际上是什么）。

例如：“pte_t”等不透明对象，你只能用合适的访问函数来访问它们。

注意！不透明性和“访问函数”本身是不好的。我们使用pte_t等类型的原因在于真的是

完全没有任何共用的可访问信息。

(b) 清楚的整数类型，如此，这层抽象就可以帮助消除到底是“int”还是“long”的混淆。

u8/u16/u32是完全没有问题的typedef，不过它们更符合类别(d)而不是这里。

再次注意！要这样做，必须事出有因。如果某个变量是“unsigned long“，那么没有必要

```
typedef unsigned long myflags_t;
```

不过如果有一个明确的原因，比如它在某种情况下可能会是一个“unsigned int”而在其他情况下可能为“unsigned long”，那么就不要犹豫，请务必使用typedef。

(c) 当你使用sparse按字面的创建一个新类型来做类型检查的时候。

(d) 和标准C99类型相同的类型，在某些例外的情况下。

虽然让眼睛和脑筋来适应新的标准类型比如“uint32_t”不需要花很多时间，可是有些人仍然拒绝使用它们。

因此，Linux特有的等同于标准类型的“u8/u16/u32/u64”类型和它们的有符号类型是被允许的——尽管在你自己的新代码中，它们不是强制要求要使用的。

当编辑已经使用了某个类型集的已有代码时，你应该遵循那些代码中已经做出的选择。

(e) 可以在用户空间安全使用的类型。

在某些用户空间可见的结构体里，我们不能要求C99类型而且不能用上面提到的“u32”类型。因此，我们在与用户空间共享的所有结构体中使用__u32和类似的类型。

可能还有其他的情况，不过基本的规则是永远不要使用typedef，除非你可以明确的应用上述某个规则中的一个。

总的来说，如果一个指针或者一个结构体里的元素可以合理的被直接访问到，那么它们就不应该是一个typedef。

第六章：函数

函数应该简短而漂亮，并且只完成一件事情。函数应该可以一屏或者两屏显示完（我们都知道ISO/ANSI屏幕大小是80x24），只做一件事情，而且把它做好。

一个函数的最大长度是和该函数的复杂度和缩进级数成反比的。所以，如果你有一个理论上很简单的只有一个很长（但是简单）的case语句的函数，而且你需要在每个case里做很多很小的事情，这样的函数尽管很长，但也是可以的。

不过，如果你有一个复杂的函数，而且你怀疑一个天分不是很高的高中一年级学生可能甚至搞不清楚这个函数的目的，你应该严格的遵守前面提到的长度限制。使用辅助函数，并为之取个具描述性的名字（如果你觉得它们的性能很重要的话，可以让编译器内联它们，这样的

效果往往会比你写一个复杂函数的效果要好。）

函数的另外一个衡量标准是本地变量的数量。此数量不应超过5－10个，否则你的函数就有问题了。重新考虑一下你的函数，把它分拆成更小的函数。人的大脑一般可以轻松的同时跟踪7个不同的事物，如果再增多的话，就会糊涂了。即便你聪颖过人，你也可能会记不清你2个星期前做过的事情。

在源文件里，使用空行隔开不同的函数。如果该函数需要被导出，它的EXPORT*宏应该紧贴在它的结束大括号之下。比如：

```
int system_is_up(void)
{
        return system_state == SYSTEM_RUNNING;
}
EXPORT_SYMBOL(system_is_up);
```

在函数原型中，包含函数名和它们的数据类型。虽然C语言里没有这样的要求，在Linux里这是提倡的做法，因为这样可以很简单的给读者提供更多的有价值的信息。

第七章：集中的函数退出途径

虽然被某些人声称已经过时，但是goto语句的等价物还是经常被编译器所使用，具体形式是无条件跳转指令。

当一个函数从多个位置退出并且需要做一些通用的清理工作的时候，goto的好处就显现出来了。

理由是：

- 无条件语句容易理解和跟踪
- 嵌套程度减小
- 可以避免由于修改时忘记更新某个单独的退出点而导致的错误
- 减轻了编译器的工作，无需删除冗余代码;)

```
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;
}
```

第八章：注释

注释是好的，不过有过度注释的危险。永远不要在注释里解释你的代码是如何运作的：更好的做法是让别人一看你的代码就可以明白，解释写的很差的代码是浪费时间。

一般的，你想要你的注释告诉别人你的代码做了什么，而不是怎么做的。也请你不要把注释放在一个函数体内部：如果函数复杂到你需要独立的注释其中的一部分，你很可能需要回到第六章看一看。你可以做一些小注释来注明或警告某些很聪明（或者槽糕）的做法，但不要加太多。你应该做的，是把注释放在函数的头部，告诉人们它做了什么，也可以加上它做这些事情的原因。

当注释内核API函数时，请使用kernel-doc格式。请看
Documentation/kernel-doc-nano-HOWTO.txt和scripts/kernel-doc以获得详细信息。

Linux的注释风格是C89“/* ... */”风格。不要使用C99风格“// ...”注释。

长（多行）的首选注释风格是：

```
/*
 * 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.
 */
```

注释数据也是很重要的，不管是基本类型还是衍生类型。为了方便实现这一点，每一行应只声明一个数据（不要使用逗号来一次声明多个数据）。这样你就有空间来为每个数据写一段小注释来解释它们的用途了。

第九章：你已经把事情弄糟了

这没什么，我们都是这样。可能你的使用了很长时间Unix的朋友已经告诉你“GNU emacs”能
自动帮你格式化C源代码，而且你也注意到了，确实是这样，不过它所使用的默认值和我们想要的相去甚远（实际上，甚至比随机打的还要差——无数个猴子在GNU emacs里打字永远不
会创造出一个好程序）（译注：请参考Infinite Monkey Theorem）

所以你要么放弃GNU emacs，要么改变它让它使用更合理的设定。要采用后一个方案，你可以把下面这段粘贴到你的.emacs文件里。

```
(defun linux-c-mode ()
  "C mode with adjusted defaults for use with the Linux kernel."
  (interactive)
  (c-mode)
  (c-set-style "K&R")
  (setq tab-width 8)
  (setq indent-tabs-mode t)
  (setq c-basic-offset 8))
```

这样就定义了M-x linux-c-mode命令。当你hack一个模块的时候，如果你把字符串
-*- linux-c -*-放在头两行的某个位置，这个模式将会被自动调用。如果你希望在你修改/usr/src/linux里的文件时魔术般自动打开linux-c-mode的话，你也可能需要添加

```
(setq auto-mode-alist (cons '("/usr/src/linux.*/.*\\.[ch]$" . linux-c-mode)
                            auto-mode-alist))
```

到你的.emacs文件里。

不过就算你尝试让emacs正确的格式化代码失败了，也并不意味着你失去了一切：还可以用“
indent”。

不过，GNU indent也有和GNU emacs一样有问题的设定，所以你需要给它一些命令选项。不过，这还不算太糟糕，因为就算是GNU indent的作者也认同K&R的权威性（GNU的人并不是坏人，他们只是在这个问题上被严重的误导了），所以你只要给indent指定选项“-kr -i8”（代表“K&R，8个字符缩进”），或者使用“scripts/Lindent”，这样就可以以最时髦的方式
缩进源代码。

“indent”有很多选项，特别是重新格式化注释的时候，你可能需要看一下它的手册页。不过
记住：“indent”不能修正坏的编程习惯。

第十章：Kconfig配置文件

对于遍布源码树的所有Kconfig*配置文件来说，它们缩进方式与C代码相比有所不同。紧挨在“config”定义下面的行缩进一个制表符，帮助信息则再多缩进2个空格。比如：

```
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.
```

仍然被认为不够稳定的功能应该被定义为依赖于“EXPERIMENTAL”：

```
config SLUB
        depends on EXPERIMENTAL && !ARCH_USES_SLAB_PAGE_STRUCT
        bool "SLUB (Unqueued Allocator)"
        ...
```

而那些危险的功能（比如某些文件系统的写支持）应该在它们的提示字符串里显著的声明这一点：

```
config ADFS_FS_RW
        bool "ADFS write support (DANGEROUS)"
        depends on ADFS_FS
        ...
```

要查看配置文件的完整文档，请看Documentation/kbuild/kconfig-language.txt。

第十一章：数据结构

如果一个数据结构，在创建和销毁它的单线执行环境之外可见，那么它必须要有一个引用计数器。内核里没有垃圾收集（并且内核之外的垃圾收集慢且效率低下），这意味着你绝对需要记录你对这种数据结构的使用情况。

引用计数意味着你能够避免上锁，并且允许多个用户并行访问这个数据结构——而不需要担心
这个数据结构仅仅因为暂时不被使用就消失了，那些用户可能不过是沉睡了一阵或者做了一些其他事情而已。

注意上锁不能取代引用计数。上锁是为了保持数据结构的一致性，而引用计数是一个内存管理技巧。通常二者都需要，不要把两个搞混了。

很多数据结构实际上有2级引用计数，它们通常有不同“类”的用户。子类计数器统计子类用
户的数量，每当子类计数器减至零时，全局计数器减一。

这种“多级引用计数”的例子可以在内存管理（“struct mm_struct”：mm_users和mm_count）
和文件系统（“struct super_block”：s_count和s_active）中找到。

记住：如果另一个执行线索可以找到你的数据结构，但是这个数据结构没有引用计数器，这里几乎肯定是一个bug。

第十二章：宏，枚举和RTL

用于定义常量的宏的名字及枚举里的标签需要大写。

```
#define CONSTANT 0x12345
```

在定义几个相关的常量时，最好用枚举。

宏的名字请用大写字母，不过形如函数的宏的名字可以用小写字母。

一般的，如果能写成内联函数就不要写成像函数的宏。

含有多个语句的宏应该被包含在一个do-while代码块里：

```
#define macrofun(a, b, c)                       \
        do {                                    \
                if (a == 5)                     \
                        do_this(b, c);          \
        } while (0)
```

使用宏的时候应避免的事情：

1) 影响控制流程的宏：

```
#define FOO(x)                                  \
        do {                                    \
                if (blah(x) < 0)                \
                        return -EBUGGERED;      \
        } while(0)
```

非常不好。它看起来像一个函数，不过却能导致“调用”它的函数退出；不要打乱读者大脑里

的语法分析器。

2) 依赖于一个固定名字的本地变量的宏：

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

可能看起来像是个不错的东西，不过它非常容易把读代码的人搞糊涂，而且容易导致看起来不相关的改动带来错误。

3) 作为左值的带参数的宏： FOO(x) = y；如果有人把FOO变成一个内联函数的话，这种用法就会出错了。

4) 忘记了优先级：使用表达式定义常量的宏必须将表达式置于一对小括号之内。带参数的宏也要注意此类问题。

```
#define CONSTANT 0x4000
#define CONSTEXP (CONSTANT | 3)
```

cpp手册对宏的讲解很详细。Gcc internals手册也详细讲解了RTL（译注：register transfer language），内核里的汇编语言经常用到它。

第十三章：打印内核消息

内核开发者应该是受过良好教育的。请一定注意内核信息的拼写，以给人以好的印象。不要用不规范的单词比如“dont”，而要用“do not”或者“don't”。保证这些信息简单、明了、无
歧义。

内核信息不必以句号（译注：英文句号，即点）结束。

在小括号里打印数字(%d)没有任何价值，应该避免这样做。

<linux/device.h>里有一些驱动模型诊断宏，你应该使用它们，以确保信息对应于正确的设备和驱动，并且被标记了正确的消息级别。这些宏有：dev_err(), dev_warn(), dev_info()等等。对于那些不和某个特定设备相关连的信息，<linux/kernel.h>定义了pr_debug()和pr_info()。

写出好的调试信息可以是一个很大的挑战；当你写出来之后，这些信息在远程除错的时候就会成为极大的帮助。当DEBUG符号没有被定义的时候，这些信息不应该被编译进内核里（也就是说，默认地，它们不应该被包含在内）。如果你使用dev_dbg()或者pr_debug()，就能自动达到这个效果。很多子系统拥有Kconfig选项来启用-DDEBUG。还有一个相关的惯例是使用VERBOSE_DEBUG来添加dev_vdbg()消息到那些已经由DEBUG启用的消息之上。

第十四章：分配内存

内核提供了下面的一般用途的内存分配函数：kmalloc()，kzalloc()，kcalloc()和vmalloc()。请参考API文档以获取有关它们的详细信息。

传递结构体大小的首选形式是这样的：

```
        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，这样会降低可读性，并且可能会引入bug。有可能指针变量类型被改变时，而对应的传递给内存分配函数的sizeof的结果不

变。

强制转换一个void指针返回值是多余的。C语言本身保证了从void指针到其他任何指针类型的转换是没有问题的。

第十五章：内联弊病

有一个常见的误解是内联函数是gcc提供的可以让代码运行更快的一个选项。虽然使用内联函数有时候是恰当的（比如作为一种替代宏的方式，请看第十二章），不过很多情况下不是这样。inline关键字的过度使用会使内核变大，从而使整个系统运行速度变慢。因为大内核会占用更多的指令高速缓存（译注：一级缓存通常是指令缓存和数据缓存分开的）而且会导致pagecache的可用内存减少。想象一下，一次pagecache未命中就会导致一次磁盘寻址，将耗时5毫秒。5毫秒的时间内CPU能执行很多很多指令。

一个基本的原则是如果一个函数有3行以上，就不要把它变成内联函数。这个原则的一个例外是，如果你知道某个参数是一个编译时常量，而且因为这个常量你确定编译器在编译时能优化掉你的函数的大部分代码，那仍然可以给它加上inline关键字。kmalloc()内联函数就是一个很好的例子。

人们经常主张给static的而且只用了一次的函数加上inline，如此不会有任何损失，因为没有什么好权衡的。虽然从技术上说这是正确的，但是实际上这种情况下即使不加inline gcc也可以自动使其内联。而且其他用户可能会要求移除inline，由此而来的争论会抵消inline自身的潜在价值，得不偿失。

第十六章：函数返回值及命名

函数可以返回很多种不同类型的值，最常见的一种是表明函数执行成功或者失败的值。这样的一个值可以表示为一个错误代码整数（-Exxx＝失败，0＝成功）或者一个“成功”布尔值（
0＝失败，非0＝成功）。

混合使用这两种表达方式是难于发现的bug的来源。如果C语言本身严格区分整形和布尔型变量，那么编译器就能够帮我们发现这些错误……不过C语言不区分。为了避免产生这种bug，请
遵循下面的惯例：

> 如果函数的名字是一个动作或者强制性的命令，那么这个函数应该返回错误代码整数。如果是一个判断，那么函数应该返回一个“成功”布尔值。

比如，“add work”是一个命令，所以add_work()函数在成功时返回0，在失败时返回-EBUSY。
类似的，因为“PCI device present”是一个判断，所以pci_dev_present()函数在成功找到
一个匹配的设备时应该返回1，如果找不到时应该返回0。

所有导出（译注：EXPORT）的函数都必须遵守这个惯例，所有的公共函数也都应该如此。私有（static）函数不需要如此，但是我们也推荐这样做。

返回值是实际计算结果而不是计算是否成功的标志的函数不受此惯例的限制。一般的，他们通过返回一些正常值范围之外的结果来表示出错。典型的例子是返回指针的函数，他们使用NULL或者ERR_PTR机制来报告错误。

第十七章：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，而不要自己写一些它们的变种。比如，如果你需要计算一个数组的长度，使用这个宏

```
  #define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
```

类似的，如果你要计算某结构体成员的大小，使用

```
  #define FIELD_SIZEOF(t, f) (sizeof(((t*)0)->f))
```

还有可以做严格的类型检查的min()和max()宏，如果你需要可以使用它们。你可以自己看看那个头文件里还定义了什么你可以拿来用的东西，如果有定义的话，你就不应在你的代码里自己重新定义。

第十八章：编辑器模式行和其他需要罗嗦的事情

有一些编辑器可以解释嵌入在源文件里的由一些特殊标记标明的配置信息。比如，emacs能够解释被标记成这样的行：

```
-*- mode: c -*-
```

或者这样的：

```
/*
Local Variables:
compile-command: "gcc -DMAGIC_DEBUG_FLAG foo.c"
End:
*/
```

Vim能够解释这样的标记：

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

不要在源代码中包含任何这样的内容。每个人都有他自己的编辑器配置，你的源文件不应该覆盖别人的配置。这包括有关缩进和模式配置的标记。人们可以使用他们自己定制的模式，或者使用其他可以产生正确的缩进的巧妙方法。

附录 I：参考

The C Programming Language, 第二版, 作者Brian W. Kernighan和Denni
M. Ritchie. Prentice Hall, Inc., 1988. ISBN 0-13-110362-8 (软皮),
0-13-110370-9 (硬皮). URL: http://cm.bell-labs.com/cm/cs/cbook/

The Practice of Programming 作者Brian W. Kernighan和Rob Pike. Addison-Wesley,
Inc., 1999. ISBN 0-201-61586-X. URL: http://cm.bell-labs.com/cm/cs/tpop/

cpp，gcc，gcc internals和indent的GNU手册——和K&R及本文相符合的部分，全部可以在
http://www.gnu.org/manual/找到

WG14是C语言的国际标准化工作组，URL: http://www.open-std.org/JTC1/SC22/WG14/

Kernel CodingStyle，作者greg@kroah.com发表于OLS 2002：
http://www.kroah.com/linux/talks/ols_2002_kernel_codingstyle_talk/html/

--
最后更新于2007年7月13日。