Linux Kernel Coding Style

https://www.kernel.org/doc/Documentation/CodingStyle

Linus Torvalds

翻译：billwangwl (bill.liangwlw@gmail.com, https://github.com/billwangwl)

这是一篇简短的描述Linux内核推荐使用的代码风格。代码风格是一个非常个人的事情，我不想强迫给任何人我的观点，但是，我维护的任何代码都是这个风格，我也希望其他人的代码也是同样的风格。所以，请考虑这里所说的风格。

首先，我建议你打印一份GNU Coding Standards，不要读。烧掉他们，这是一个很棒的姿态。

接下来我们继续：

第一章 缩进

Tab是8个字符，这样缩进也应该是8个字符。有些异教徒把缩进搞成4个字符，或者2个字符，这就和把PI定义为3一样。

原因：用8个字符的缩进就是为了能够清楚的看出一个代码块的开始和结束。特别是你盯着你的屏幕已经20小时了，大的缩进更容易分辨出代码块。

有人说8个字符的缩进会让代码距离行首太远，在80个字符的屏幕上比较难以阅读。答案是，如果你需要超过3层的缩进，你的代码写烂了，需要修改你的代码了。

简短来说，8个字符的缩进可以让阅读更加容易，并且会提醒你，你的代码是不是嵌套太深了。

switch语句推荐的通过让swith和case对齐来减少缩进，比如：

```
switch (suffix) {
case 'G':
case 'g':
        mem <= 20;
        break;
case 'e:
        me....
```

```
        /* fall through */
    default:
        break;
    }
```

不要把多个语句放在一行来处理，除非你有意要隐藏一些事情：

```
    if (condition) do_thing;
        do_someting_everytime;
```

不要把多个赋值语句放在一行。内核的编码风格很简单，避免有混淆的表达式。

空格只能用于注释。文档的缩进和Kconfig也应该用tab缩进。以上例子是故意用空格的。

使用一个好的编译器，不用在行末留白。

第二章，划分长的行和字符串

好的编码风格就是用通用的工具来让代码更具可读性和可维护性。

强烈推荐每行80个字符的限制。

超过80字符限制的语句需要被划分为短行。划分的后续的行大体上要比第一行短，挨着右侧对齐。同样的，这个办法也适用于有比较长的参数列表的函数头。长的字符串也应该按照这个办法来划分。唯一的例外是划分80个字符会降低可读性，而且会隐藏信息。

```
    void fun(int a, int b, int c)
    {
        if (condition)
            printk(KERN_WARNING "Wang this is a long printk with "
                                "3 parameters a :%u b: %u"
                                "c %u\n", a, b, c);
         else
            next_statements;
```

```
}
```

第三章：括号和空格的位置

另一个和C紧密相关的问题是括号的位置。和缩进不同的是，不管选择如何放置括号，基本上没有基于技术上的考虑，但是从倾向来说，K&R给我们展示了放置括号的方式，把开括号放在行末，闭括号放在行首，比如：

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

这个规格适用于所有非函数的语句（if, switch, for, while, do ），例如：

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

特别的是，函数定义，开括号在行首，比如

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

异教徒会说大喊大叫，这里不一致，不过，好吧，就算不一致，但是所有思维正常的人都会认为第一，K&R是对的，第二，K&R是对的。另外，函数在C里本来就是特殊的，你不能在C里嵌套函数。

注意，闭括号应该独占一行，除非紧跟着闭括号的语句和括号属于同一个语句，比如do....while，或者else, else if，比如：

```
do {
```

```
        body of do-loop
    } while (condition);
    if (x == y) {
        ...
    } else if (x > y)   {
        ....
    } else {
        ...
    }
```

原因：K&R

这样放置括号，可以使空行的数量最小，而且没有损失可读性。如果你只有有限的屏幕大小，通过这样方式括号，可以让你有更多的行来写更多的内容。

在可不使用括号的地方不要使用括号。

```
    if (condition)
        action();
```

和

```
    if (condition)
        do_this();
    else
        do_that();
```

以上规则不适用于如果只有一个branch只有一句话。这时候需要都加上括号：

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

3.1 空格

Linux内核对于空格的使用，大多数是依据函数以及关键字。在关键字后大多使用空格，如之前代码表现的样子。需要注意的是sizeof, typeof, alignog和__attribute__这些后边不需要空格。

在以下的关键字后使用空格：

if, swith, case, for, do, while

在以下关键字后不使用空格，sizeof, typeof, alignof, __attribute，比如：

s = sizeof(struct file);

不要在括号里边加空格，下边的例子表现了这种错误：

s = sizeof(struct file);

当声明了一个指针数据或者返回指针的函数，*紧跟着变量名或者函数名，不要和他们的类型连在一起。比如：

char *linux_banner;

unsigned long long temparse(char *ptr, char **retptr);

char *match_strdump(substring_t *s);

在大多数的二元或者三元操作符的两侧使用空格：

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

但是在一元操作符不要使用空格：

& * + - ~ ! sizeof typeof alignof __attribute__ defined

在自增或者自减运算符前后不要用空格，结构体变量用到的'.'和'->'也不要用空格。

不要在行末有空白（空格，tab等）。有的编辑器会自动缩进，在你输入到下一行时，会自动缩进，如果你需要一个空行的话，这一行就会因为自动缩进的关系，而引入空白。

Git会对行末的空白发出警告，并且有选择的去掉行末的空白（需配置），不过，如果要合入一系列的patch，Git自动去掉行末空白的行为会让后续的patch合入失败。（billwangwl注：前一个patch合入时，Git如果修改了行末的空白，下一个patch合入时，可能会因为diff的结果不同，而无法合入）

第四章 命名

C是斯巴达人的语言，因此你要给他简洁明了的命名【billwangwl注：斯巴达人说话简明，直接了当，这里说C语言和斯巴达语言一样】。和Modula-2, Pascal程序员不同， C程序员不会使用诸如ThisVariableIsATemporaryCounter这类傻乎乎的名字。一个C程序员应该起一个叫tmp的变量，这意味比较容易书写，而且也不难理解。

尽管大小写混写不好看，但是对于一些全局的变量是必不可少的。给一个全局变量命名为"foo"也是不合适的。

全局变量（除非你特别需要全局变量，否则不要用全局变量）的名字应该是描述性的，全局函数的名字也应该是描述性的。如果你有一个函数，用来对登录用户计数，你应该起一个比如“count_active_users()"或者类似的名字，不应该叫”cntusr()"。

把函数类型写进函数名这种匈牙利命名法简直就是脑子进水的行为，编译器知道函数的类型并且会自己检查，这种命名方法只能愚弄程序员。不要希望微软会写出牛逼的程序。

本地变量应该短，而且准确。如果你有一些随机的整数循环技术器，就应该简单的起名‘i'，起个“loop_counter"不太有效率，这种命名不会有混淆。同样的"tmp"也仅仅用于任意保存临时变量的地方。

如果你会混淆本地变量，那么你会有另外一个问题，function-growth-hormone-imbalance syndrome（函数增长荷尔蒙不均衡综合症，第六章会有介绍）

第五章 typedefs

不要使用诸如”vps_t"这类名字。

这个看起来就是个错误，给一个结构体和指针来typedef。当你看到

```
vps_t a;
```

你压根不知道这是什么玩意？

与之对照，如果是

```
struct virtual_container *a;
```

你就可以准确的知道a是什么。

许多人认为typedef能提高阅读性，事实上不是这样子的，他们仅仅用于：

(a) 完全不透明的对象（这里typedef就是用来隐藏对象的）

比如：“pte_t"等等。这类对象你就只能通过他们自己的访问函数来进行。

需要注意的是，不透明的对象和访问函数也不是很好。我们使用诸如pte_t这类的名字是，确实是完全没有可以移植的信息。

(b) 清楚的整数类型，这样避免了带来什么时候用int，什么时候用long的困扰。

u8/u16/u32 是typedef完美的应用，尽管这里的typedef更适用于接下来所说的(d)的原因。

注意，如果你确实需要用typedef干一些事，一定要有确切的原因。比如”unsigned long"，就没有道理写成”typedef unsigned long myflags_t;"。但是如果在某种情况下，确实需要“unsinged int"或者"unsigned long"，那就用typedef来做。

(c) 当你想要用较少的字符来创造一个新的类型来进行类型检查的时候。

(d) 新的类型和C99一样，但是有些情况下却有例外。

尽管可能需要一点点的时间，就会习惯使用诸如uint32_t，不过有人任然拒绝这些。

因此上，Linux特有的'u8/u16/u32/u64"和他们对应的有符号类型是标准类型，当然他们不是强制的，你可以在你新代码中不用他们。

当编辑已有代码时候，你应该用和已有代码一样的方式。

(e) 为了在用户空间使用时的类型安全。

有些类型在用户空间是可见的，我们不能使用C99的类型，并且不能使用上边提到的”u32“这些类型。我们可以使用诸如__u32或者类似的书写， 来让用户空间使用这些类型。【billwangwl注：这里比较绕，这里是说有些kernel类型在用户空间是可见的，比如C99定义的uint32_t，如果我们要在用户空间使用kernel空间定义的类型，我们可以使用__u32这类写法】

可能还会有其他的原因来使用typedef，但是基本的规则是当你使用typedef的时候，应该匹配上边提到的规则。

通常来说，一个指针，或者一个结构体是可以被直接访问的，不要使用typedef。

第六章 函数

函数应该简短清楚，只做一件事。一般只能有一屏幕到两屏幕（ISO/ANSI的屏幕尺寸

是80×24），只做一件事并把他做好。

一个函数的长度与他的复杂度和缩进有关。因此，如果你有一个概念上简单的函数，比如就一个简单的case语句，但是很长，这种情况没有关系，就让他长吧。

如果你有一个复杂的函数，你怀疑一个天赋一般的高一的孩子不能理解他，那么你就应该严格来遵守限制（billwangwl注：前边提到的保持在一到两屏）。使用辅助函数，用描述的名字（如果有性能的要求，你可以用inline，编译器可能会比你直接写一个长的函数做的更好）。

另外一个考量函数的方法是本地变量的个数。不应该超过5-10个，不然你可能写错了。重新考虑函数，然后把他们分成更小的函数。一个人类的大脑一般来说能记忆7个不同事物，再多就乱了。你知道你很聪明，但是你记着你两周前开发的代码么？

在源代码文件中，应该用一个空行分割函数。如果这个函数需要export，那么EXPORT_***宏应该紧接着闭括号。比如

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

在函数原型中，参数要包括类型和名字。尽管这个C语言不强求，但在Linux中更希望如此，因为可以容易的增加一些可读性的内容。

第七章 集中函数的退出语句

虽然有人反对，不过goto或者与之等价的一些跳转语句会在编译器遇到一些无条件跳转指令时经常用到。

当一个函数从多个位置退出或者一些通用的清除语句时，经常会用到goto.

这样做的原因是：

--无条件退出语句更容易理解和阅读

--减少嵌套

--可以避免由于修改而没有更新某个单独的退出状态而引起的错误

--减轻了编译器的工作，而无需删除冗余代码

```
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

注释是不错的做法，但是要避免过度注释。不要试制注释你的代码是如何工作的，更好的做法是用代码来说明是如何工作，给写的烂的代码写注释就是浪费时间。

通常你需要注释你的代码是干嘛的，而不是如何做。同样，需要避免在一个函数体内加上注释，如果一个函数太复杂，以至于你需要解释其中的一些部分，那么你应该回头再看看第六章。你可以用小的注释提醒或者警告一些聪明或者不好的代码，但是要避免不要过头。你可以把注释放在函数头，说明函数的作用，或者为什么他要这么做。

当注释kernel API，请使用kernel-doc格式。请查阅Documentation/kernel-doc-nano-HOWTO.txt和scripts/kernel-doc获取更多细节。

Linux的注释风格是C89 /*...*/，　不要使用C99 //.

推荐的长注释如下：

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

同时也要注意注释数据，不管他们是基本类型还是继承而来的。最后，记着每一行只声明一个变量，这样会给你留下一点地方用来写简短的注释。

第九章事情搞砸了

【billwangwl注】这一章是emacs的一些指导，略。

第十章 Kconfig 配置

对于所有的Kconfig文件，缩进有所不同。在"config"下边的行，用一个tab缩进，help在额外用两个空格缩进。

```
config AUDIT
        bool "Auditing support"
        depends on NET
        help
          Enable auditing infrastructure that can be used with another
          kernel subsystem, .....
```

对于还没有稳定的功能，应该对EXPERIMENTAL有依赖。

config SLUB

depends on EXPERIMENTAL && !ARCH_USES_SLAB_PAGE_STRUCT

bool "SLUB (Unqueued Allocator)"

对于有危险的功能，应该显著的提示

config ADFS_FS_RW

bool "ADFS write support (DANGEROUS)"

depends on ADFS_FS

对于整个Kconfig文件，请参考Documentation/kbuild/kconfig-language.txt

第十一章：数据结构

【billwangwl注：这里的数据结构有两种意思，一种是数据结构的声明，一种是数据结构的对象，为保持和原文的一致性，依照原文保留将data structure的直译】

在单线程环境意外可以创建，销毁的对象，总是应该有一个计数器。kernel里没有垃圾收集（kernel以外的垃圾收集都很慢而且没有什么效率），这意味着你一定要有自己管理的计数器。

计数器意味着你可以避免锁，允许多个用户同时访问数据结构，而且不用担心数据结构因为睡眠或者其他原因突然消失。

需要注意的是锁和引用计数并不是可互相替代的。锁意味着coherent持有数据结构，而计数器是一种内存管理手段。通常两种方法都需要，而且两种方法没有耦合，不会干扰对方。

许多数据结构有两级引用计数，可以在不同级别使用。子一级的计数器会记录子类的使用情况，当子一级的计数器到0时，父一级的计数器再进行减一。

多级计数器的例子可以查看内存管理部分（"struct mm_struct":mm_user 和mm_count)和文件系统的代码（"sturct super_block": s_count和s_active)。

请记住：如果另外一个线程可以访问到你的数据结构，而你没有定义一个计数器，那么一定会有一个bug。

第十二章：宏，枚举和 RTL(register transfer language)

通过宏定义的常量和枚举的标签都需要大写。

```
#define CONSTANT 0x12345
```

当有一些有关系的常量时，推荐使用枚举。

推荐使用大写的宏，但是如果是函数，就应该小写。

通常，inline函数推荐使用嵌入的函数宏。

多行的宏，应该被do-while包含：

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

使用宏的时候需要避免：

1)影响控制流程

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

这种看上去很糟糕，因为return会从调用这个宏的函数里返回。

2) 宏依赖于一个本地的魔鬼数字

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

看上去不错，但是会把看代码的人搞糊涂，不知道index是干嘛的，而且后续改动时出错，比如改掉了调用的地方把index改没有了。

3) 带有参数，并且作为左值的宏：FOO(x) = y;如果有人把这个宏修改成一个inline函数里，就会出错。

4) 忘记了优先级。宏定义的常量，如果是表达式计算得到的，那么就要写到（）里。带参数的宏也要注意同样的问题。

cpp手册(【billwangwl注】这里的cpp是预处理器，不是C++)对宏讲解的很细致。GCC的内部手册对RTL也有详尽的讲解，在kernel中会用到很多汇编语言。

(【billwangwl注】单独的文件可以通过-E参数来查看预处理后的文件，比如gcc -Wall -E test.c，会输出test.c经过预处理后的文件)

第十三章 如何打印 kernel log

kernel开发者就像文学家。注意良好的拼写，给人留下好印象。不要使用残缺不全的单词，比如'dont'，使用'do not'或者'don't'。信息要准确，清楚，不含糊。

kernel信息不用以句号结束。

不要在增加额外的括号来打印数字(%d)，避免这种写法。

在<linux\devcie.h>中定义了一些驱动的诊断宏，你应该使用和需要的等级相匹配的宏，dev_err(),dev_warn(),dev_info()这些。和特定无关设备无关的消息，使用pr_debug, pr_info()，这些定义在<linux\printk.h>中。

要写出良好的debug信息个挑战，一旦你拥有这种本领，那么就会给远程调试带来巨大的帮助（【billwangwl注】这里的远程调试指的是你不在出错现场的调试）当DEBUG信息没有打开时,这些信息不应该包含在内(就是说,默认的这信息是不包括在编译好的镜像之中)。当你使用dev_dbg()或者pr_debug()时，这是默认的行为。许多子系统通过在Kconfig来打开-DDEBUG。还有一个相关的惯例是使用VERBOSE_DEBUG来添加dev_vdbg()消息来给已经打开DEBUG的系统添加信息。

第十四章：申请内存

kernel提供了以下的方法用于通用的申请内存的方法：kmalloc, kzalloc, kcalloc, vmalloc, zalloc。参考API文档了解这些接口。

推荐申请一个结构的方式是：

p = kmalloc(sizeof(*p), ...);

另外一种写法就是siezof(sturct xxx)，这样写会降低可读性，同时，可能会引入bug，比如后来你改了p的类型，但是在kmalloc的地方没有同时修改。

转换返回的void指针到其他类型是多余的。C语言保证void转换到其他类型不会错误。

第十五章：inline 的问题

有一个普遍错误的感觉就是gcc拥有通过inline来加个程序的魔力。正确使用inline(第十二章描述的情况）能带来收益，但是大多数时候往往会出错。大量的使用Inline会产生一个庞大的kernel，然后把整个系统拉慢，因为一个大的内核会占用icache较多的空间，这样就没有给memory留下足够的page cache。想象一下，一个page cache未命中，很容易引起5ms的硬盘查找，而5ms能让cpu干很多事。

一个好的办法是，不要对超过3行的代码使用inline。这个规则的一个例外是，一个参数是编译期常量，并且你清楚编译器能对你的函数进行优化，那么你可以用inline。作为一个好的例子是取看看kmalloc的inline函数。

有时候人们常说，给一个static并且只使用一次的函数，可以添加inline，因为不会增加额外的空间消耗。这仅仅是技术上的正确，gcc有能力会自动将这些函数变成inline，而且会给维护带来麻烦，如果再次使用这个函数时，就会变得糟糕。

第十六章：函数返回值和名字

函数有很多中返回值，其中之一就是通过返回值来知会函数是成功的还是失败的。这样的值可以用erro-code来标示（-Exxx = failure，0 = success) 或者一个标示成功的布尔值（0 = failure, 非零是成功）。

如果混淆以上两种方式，那就很难发现Bug了。如果C语言能够按照类型来识别布尔值和整数，就能发现这种错误，但是C语言不能。为了预防这种错误，请按照以下的惯例：

- 如果一个函数的名字是一个动作或者命令这个函数就应该返回一个error-code。
- 如果函数的名字是个判定，那就返回一个标示成功与否的bool.

比如，“add work"作为一个命令，那么add_work()就该返回0标示成功，返回-EBUSY表示失败。"PCI device present"是一个判定，pci_dev_present()就该返回1标示true，0标示false.

所有EXPORT函数都应该遵守这个规则，所有公共函数也应该遵守这个规则。私有的或者静态的函数可以不遵从这个规则，但是推荐使用。

函数返回值是真实计算出来的结果，而不是表示成功与否，则可以不遵从这个规则。通常他们返回一些范围之外的值标示失败。典型的例子是指针，通过返回NULL或者ERR_PTR来表示错误。

第十七章 不要重新发明内核宏

你应该使用<linux/kernel.h>包含的一些宏，而不是自己写一些类似的。比如你要计算一个数组的长度，你可以使用

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

同样的，如果你需要计算结构某个成员的大小，使用：

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

此外还有min()，max()宏可以进行严格的类型检查。你可以看看这个头文件还有哪些宏你可以使用，不要自己再写一些类似的宏。

第十八章 编辑器模式和其他一些注意事项

一些编辑器可以解释嵌入到源代码中的一些特殊标记的配置信息。比如emacs或者vim都有。

不要把这些内容写到源代码里。每个人都有自己的习惯，你不能把自己的配置信息也写到代码里。

附录：

The C Programming Language, Second Edition

by Brian W. Kernighan and Dennis M. Ritchie.

Prentice Hall, Inc., 1988.

ISBN 0-13-110362-8 (paperback), 0-13-110370-9 (hardback).

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

The Practice of Programming

by Brian W. Kernighan and Rob Pike.

Addison-Wesley, Inc., 1999.

ISBN 0-201-61586-X.

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

GNU manuals - where in compliance with K&R and this text - for cpp, gcc, gcc internals and indent, all available from http://www.gnu.org/manual/

WG14 is the international standardization working group for the programming language C, URL: http://www.open-std.org/JTC1/SC22/WG14/

Kernel CodingStyle, by greg@kroah.com at OLS 2002: http://www.kroah.com/linux/talks/ols_2002_kernel_codingstyle_talk/html/