Linux 内核编码风格

这是一份简短的描述 Linux 内核编码风格的首选文档。编码风格是很个人的行为，我不想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。

所以请至少考虑一下本文所述的观点。

第一节 缩进

TAB 键是 8 个字符，因此缩进也是 8 个字符。但有捣蛋份子确试图将缩进变为 4 个（甚至 2）字符，这和尝试把圆周率定义为 3 没有什么两样。

理由：缩进背后的全部意图在于清晰的界定一个控制块的开始和结束。尤其是当已经连续工作 20 小时以上的时候，你会发现，如果缩进大一些，可以更容易区分不同的控制块。

现在，有些人可能会抱怨 8 个字符的缩进会使代码向右边缩进得太多，导致在 80 个字符的终端屏幕上很难阅读。但问题是如果你需要 3 级以上的缩进，不管缩进的深度如何，都应该改进你的代码。

总之，8 个字符的缩进可以让代码可读性更高，同时还可以在嵌套过多时发出警告，请对其保持警觉。

减少 switch 语句中缩进级数的首选方式是让“switch”和从属于它的“case”分支左对齐于同一列，而不要“两次缩进”“case”分支。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;
}
```

不要在同一行里放多个语句，除非你想掩盖什么东西^^

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

最好也不要把多个赋值语句放在同一行。

内核代码应该是非常非常简单的！请避免使用过于巧妙的表达式^^

除了注释、文档和 Kconfig 之外，最好不要使用空格来缩进，前面的例子就是一个反面典型^^

选择一个合适的编辑器，并且不要在行尾留空格！

第二节 适当的换行

代码风格的意义在于，即使采用不同的编辑器，仍然可以保证代码的可读性和可维护性。

每一行长度的限制是 80 列，并且强烈建议应该是首选的。

长度超过 80 列的语句可以分成几个有意义的片段，每个片段要明显短于原来的语句，并且放的位置也要明显的靠右。这个规则对于有很长参数列表的函数同也同样适用。长字符串也要打散成较短的字符串。只有一种情况，即超过 80 列可以大幅度提高可读性并且不会隐藏信息的情况，可以例外^^

```
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;
}
```

第三节 放置大括号和空格

C 语言风格中另外一个问题是大括号的放置。这和缩进的大小不一样，选择或者是放弃某种放置策略几乎没有技术上的原因，但首选的方式，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;
```

```
}
```

不过呢，有一种特殊情况，函数定义：它们的起始大括号放置于下一行的开头，就象这样：

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

全世界的异端可能会抱怨这个自相矛盾的规则，呃。。。确实是自相矛盾，不过所有思想健全的人都知道，K&R 是正确的，K&R 是正确的。。。况且，不管怎样说，函数都是特殊的（在 C 语言中，函数是不能嵌套定义的）^^

一般来说结束大括号应独自占据一行，除非它后面跟着同一个语句的剩余部分，比如说 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。

当然这样做也是有好处的，可以使空（或者差不多空的）行的数量最小化，同时不损失可读性。因此，由于你的屏幕上的新行的供应不是可回收的资源（想想这是 25 行的终端屏幕），你将会有更多的空行来放置注释。

仅有一个单独的语句时，不用加不必要的大括号。

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

有一种情况需要注意，当某个条件语句的一个分支为单独语句，而另一个分支为多条语句，这时应该两个分支里都使用大括号。

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

第 3.1 节 空格

Linux 内核中空格的使用主要取决于它是用于函数还是关键字。除了 sizeof、typeof、alignof 和__attribute__外，其余的关键字后都要加一个空格，主要是因为它们有点像函数（它们在 Linux 里也常常伴随小括号使用，尽管这类小括号不是必需的，就像“struct fileinfo info”声明过后的“sizeof info”）。

所以下面这几个关键字之后要放一个空格：

if, switch, case, for, do, while

但是 sizeof、typeof、alignof 或者__attribute__这些关键字后面，就不要放了。例如，

s = sizeof(struct file);

不要在小括号内的表达式两侧加空格。这是一个反例：

s = sizeof(struct file);

当声明指针类型变量或者返回值为指针类型的函数时，“*”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。例如，

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

大多数二元和三元操作符两侧各有一个空格，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

后缀自增和自减一元操作符前不加空格：

++ --

前缀自增和自减一元操作符后不加空格：

++ --

结构体成员操作符“.”和“->”前后不用加空格

不要在行尾留空格。有些编辑器具备智能缩进功能：可以在回车换行时会自动在新行插入一个空格，这样你就可以直接在这一行输入代码了。但问题是如果你没有输入代码，它们也不会移除已经插入的空格，这样就像你刻意留下一个空行。那么，当统计行数的时候，结果可能会偏多。

当 Git 发现补丁中包含行尾空白时会有警告提示，并且可以应你的要求去掉行尾空格；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。

第四节：命名

C 语言是一门朴素的语言，那么命名也应该如此。和 Modula-2 和 Pascal 程序员不一样，C 程序员不会使用类似 ThisVariableIsATemporaryCounter 这样酷毕的名字，而仅用“tmp”就可以了，这样写起来更容易，也不会另其难以理解。

尽管如此，给全局变量定义一个一目了然的名字还是必须的。如果谁把全局变量定义成“foo”，应该拉出去毙了^^

全局变量需要一个一目了然的名字，全局函数也是这样。如果你有一个用来统计活动用户数量的函数，你应该命名为“count_active_users()”或者是类似的名字，而不应该叫"cntusr()"。

在函数名中包含函数类型的做法（所谓的匈牙利命名法）应该是脑子坏掉了-无论如何编译器是知道那些数据类型，并且能够进行类型检查，这样做只会困扰程序员。难怪微软的程序满是 BUG^^

局部变量应该言简意赅。如果需要定义一个变量来表示循环记数器，它应该被命名为“i”。命名为“loop_counter”是没什么好处的，如果它没有被误解的话。同理，只要是临时的变量，不管什么类型都可以定义为“tmp”。

可如果你怕混淆了本地变量名，你就遇到了另一个问题，叫做函数-增长-荷尔蒙-失衡-综合症。请看第六节（函数）。

第五节：Typedef

请不要定义类似“vps_t”之类的类型。

使用 typedef 来定义结构和指针的别名是错误的，当你看见代码里面有这样一句，

```
vps_t a;
```

“a”是什么呢？

反过来，如果代码这样写，

```
struct virtual_container *a;
```

你就能知道"a"是什么了。

许多人认为 typedef 能帮助提高程序可读性，但事实确不是这样的。它们只在下列情况下有用：

(a) 完全不透明的对象（这种情况要主动隐藏这个对象的实际意义）。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 等类型的原因在于确实是完全没有任何共用的可访问信息。

(b) 定义清晰的整数类型，这样就可以帮助我们避免把"int"和"long"混淆。u8/u16/u32 是完美的 typedef，不过它们更符合(d)中所言，而不是这里。再次注意！必须要出师有名！如果某个变量是“unsigned long“，那么没有必要这样做，

```
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”，那么请不要犹豫，务必使用 typedef。

(c) when you use sparse to literally create a _new_ type for type-checking.

(d) 在某种特殊情况下，和标准 C99 类型相同的类型。

虽然让眼睛和头脑来适应类似“uint32_t”一样的标准类型几乎不需要花时间，可还是有些人拒绝接受。

因此，“u8/u16/u32/u64”类型和它们的有符号类型，可以说是 Linux 的事实标准，是可以使用的，但并不做强制要求。

在修改代码时，如果该代码中已经选择了某个类型集，那么应该遵循现有规则。

(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，并且不能用上面提到的“u32”类型。因此，我们在和用户空间共享的所有结构体中使用__u32 或者是类似的类型。

可能还有其他情况也会用到 typedef，不过基本的规则是永远不要使用 typedef，除非可以明确的应用上述规则中的一个。

第六节：函数

函数应该短小精悍，并且只实现一个功能。它们应该一屏或者两屏显示完（我们都知道 ISO/ANSI 屏幕大小是 80x24），只实现一个功能，并把它做好。

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，如果你有这样一个函数：只有一个 switch 语句，但有很多个不同的 case 分支，每个分支里面只需要做很小的事情。这样的函数尽管很长，但也是可以接受的。

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甚至可能搞不清楚这个函数的功能，你应该更严格的遵守最大限制。

进行函数拆分，并为之取个具有描述性的名字（如果你觉得其对性能要求严格的话，你可以要求编译器将它们内联展开，它往往会比你更好的完成任务。）

函数另外一个衡量标准是局部变量的数量。通常来说，不应超过 5－10 个，否则这个函数就有问题了。需要重新考虑一下，把它分拆成更小的函数。这是为什么呢？人的大脑一般可以轻松的同时跟踪 7 个不同的事物，如果再多的话，就会糊涂了。即便你聪颖过人，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，但在 Linux 里这是首选的做法，因为这样可以很简单的给读者提供更多的有价值的信息。

第七节 集中函数出口

虽然有些人反对使用 goto 语句，但编译器仍在频繁使用无条件跳转指令，而 goto 仅仅是该指令的化身。

当一个函数可能从多个位置退出，并且退出时需要做一些共性的动作，比如清理内存。这时，goto 的好处就显而易见了。

理由是：

- 无条件跳转语句容易理解和跟踪
- 缩减嵌套层数。
- 在修改退出代码时，可以避免因遗漏某个分支造成的错误。
- 减轻编译器的负担，提高编译速度。

```
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;
}
```

第八节：注释

注释是好的，但也存在过度注释的危险。永远不要尝试在注释里面解释代码是如何工作的：代码看起来一目了然是最理想的，况且给糟糕的代码添加注释无疑是在浪费时间。

一般来说，你希望你的注释能够表述代码实现的功能是什么，而不是怎么实现的。同时，还要注意避免把注释放到函数体里面：如果函数复杂到需要分段注释才可以的话，你或许应该回顾一下第 6 节。对于代码中一些比较巧妙的（或者糟糕的）做法，可以添加适当的注释，但不宜太多。推荐的做法是把注释放在函数前面，解释一下这个函数是什么功能，为什么这么做。

如果是内核的 API 函数，请使用 kernel-doc 风格，Documentation/kernel-doc-nano-HOWTO.txt 和 scripts/kernel-doc 中有详细描述。

Linux 注释的风格是 C89 的风格，形如“/* ... */”，不要使用 C99 风格的注释，形如“// ...”

多行注释的首选风格是：

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

变量的注释也很重要，不管是基本类型还是衍生类型的。为了实现这一点，需要每行只声明一个变量。这样就可以有空间来添加一些小巧精致的注释来解释它的用途了。

第九节 You've made a mess of it

关于 emacs，省略

第十节 Kconfig 配置文件

关于 Kconfig，省略。可以参考文档 Documentation/kbuild/kconfig-language.txt。

第十一节 数据结构

一个数据结构，若是多个线程都可以进行读写访问，那么它就需要一个引用记数器。内核里是没有垃圾回收机制的，这就意味着你必须自行监控。

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。

值得注意的是，上锁并不能取代引用记数。上锁是为了保证数据一致性，而引用记数则是一个内存管理技巧。一般来说二者都是需要的，千万不要搞混喽！

记得：如果另外一个线程也能够访问这个数据结构，可它又没有引用记数器，那你几乎已经收获一个 BUG 了！

第十二节 宏、枚举和 RTL

定义是常量的宏和枚举类型的成员需要全大写。

```
#define CONSTANT 0x12345
```

在定义几个有关联的常量的时候，枚举类型是首选。

宏的名字用大写，但宏函数可以用小写。

一般来说，如果能设计为内联函数，就不要用宏函数。

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：

1) 不要影响程序流程:

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

这是一个非常糟糕的定义。

2) 宏里面包含来源不明的变量

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

可能看起来还不错，可一旦谁修改了相关代码，并且确认修改代码完全没有问题，但程序却崩溃了，这会让人感觉匪夷所思。

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)
```

第十三节 内核调试信息

内核开发者显然应该是受过良好教育的。请务必注意内核调试信息的拼写，给人留下好印象。不要用不规范的单词比如“dont”，而要用“do not”或者“don't”。保证这些调试信息尽量简单、

明了、无歧义。

内核调试信息的结尾不需要句号。

打印数字时，把数字放在小括号里，(%d)，是没有意义的，应该尽量避免。

在<linux/device.h>中定义了设备分级调试宏，你应该使用它们，确保设备名称是正确的，并且标记了正确的级别：dev_err(), dev_warn(), dev_info()。对于那些不和某个设备相关连的信息，<linux/kernel.h>定义了 pr_debug()和 pr_info()。

版本发布时包含言简意赅的调试信息是非常有挑战的！一旦能做到这一点，将为远程故障诊断提供巨大帮助。当然，这些信息在 DEBUG 没有定义的情况下是不会被加入内核的。

第十四节 分配内存

内核提供了几个通用的内存分配函数：kmalloc(), kzalloc(), kcalloc(), 和 vmalloc()。可以参考 API 文档来获取它们的详细信息。

传递一个结构体大小的首选方式应该是这样的，

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

还有一种方式是选择结构体的名字，这样可能会降低可读性，还可能让 BUG 有机可乘：当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 忘记修改了，BUG 来了，程序崩溃了，系统挂掉了。

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。

第十五节 内联弊病

内联，充满魔幻色彩的 gcc 加速选项之一，很多人对此趋之若鹜。虽然使用内联函数有时是恰当的（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，不会有任何损失，因为这种情况下没有什么好权衡的。虽然从技术上说，这是正确的，不过 gcc 可以在没有提示的情况下自动使其内联，而且其他用户可能会要求移除 inline，此种维护上的争论会抵消可以告诉 gcc 来做某些事情的提示带来的潜在价值。因为不管有没有 inline，这种函数都会被内联。

第十六节 函数返回值和命名

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（-E**＝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。为了避免产生这类 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 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。

第十七节 不要重复发明内核宏

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。

第十八节：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。