Contents

Linux Kernel Coding Style (Linux 内核代码风格)

This is a short document describing the preferred coding style for the linux kernel. Coding style is very personal, and I won't _force_ my views on anybody, but this is what goes for anything that I have to be able to maintain, and I'd prefer it for most other things too. Please at least consider the points made here.
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。

First off, I'd suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it's a great symbolic gesture.
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。

Anyway, here goes:
不管怎样，现在我们开始：

Chapter 1: Indentation （缩进）

Tabs are 8 characters, and thus indentations are also 8 characters. There are heretic movements that try to make indentations 4 (or even 2!) characters deep, and that is akin to trying to define the value of PI to be 3.
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。

Rationale: The whole idea behind indentation is to clearly define where a block of control starts and ends. Especially when you've been looking at your screen for 20 straight hours, you'll find it a lot easier to see how the indentation works if you have large indentations.
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。

Now, some people will claim that having 8-character indentations makes the code move too far to the right, and makes it hard to read on a 80-character terminal screen. The answer to that is that if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.
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。

In short, 8-char indents make things easier to read, and have the added benefit of warning you when you're nesting your functions too deep. Heed that warning.
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。

The preferred way to ease multiple indentation levels in a switch statement is to align the "switch"; and its subordinate "case"; labels in the same column instead of "double-indenting" the "case" labels. E.g.:
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;
}
```

Don't put multiple statements on a single line unless you have something to hide:
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;
```

Don't put multiple assignments on a single line either. Kernel coding style is super simple. Avoid tricky expressions.
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。

Outside of comments, documentation and except in Kconfig, spaces are never used for indentation, and the above example is deliberately broken.
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

Get a decent editor and don't leave whitespace at the end of lines.
选用一个好的编辑器，不要在行尾留空格。

Chapter 2: Breaking long lines and strings (把长的行和字符串打散)

Coding style is all about readability and maintainability using commonly available tools.
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

The limit on the length of lines is 80 columns and this is a hard limit.
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。

Statements longer than 80 columns will be broken into sensible chunks. Descendants are always substantially shorter than the parent and are placed substantially to the right. The same applies to function headers with a long argument list. Long strings are as well broken into shorter strings.
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;
}
```

Chapter 3: Placing Braces （大括号和空格的放置）

The other issue that always comes up in C styling is the placement of braces. Unlike the indent size, there are few technical reasons to choose one placement strategy over the other, but the preferred way, as shown to us by the prophets Kernighan and Ritchie, is to put the opening brace last on the line, and put the closing brace first, thusly:
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
}
```

This applies to all non-function statement blocks (if, switch, for, while, do). E.g.:
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;
}
```

However, there is one special case, namely functions: they have the opening brace at the beginning of the next line, thus:
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
}
```

Heretic people all over the world have claimed that this inconsistency is ... well ... inconsistent, but all right-thinking people know that (a) K&R are _right_ and (b) K&R are right. Besides, functions are special anyway (you can't nest them in C).
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）。

Note that the closing brace is empty on a line of its own, _except_ in the cases where it is followed by a continuation of the same statement, ie a "while" in a do-statement or an "else" in an if-statement, like this:
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);
```

and
和

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

Rationale: K&R.
理由：K&R。

Also, note that this brace-placement also minimizes the number of empty (or almost empty) lines, without any loss of readability. Thus, as the supply of new-lines on your screen is not a renewable resource (think 25-line terminal screens here), you have more empty lines to put comments on.
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。

Do not unnecessarily use braces where a single statement will do.
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();
```

This does not apply if one branch of a conditional statement is a single statement. Use braces in both branches.
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 Spaces （空格）

Linux kernel style for use of spaces depends (mostly) on function-versus-keyword usage. Use a space after (most) keywords. The notable exceptions are sizeof, typeof, alignof, and __attribute__, which look somewhat like functions (and are usually used with parentheses in Linux, although they are not required in the language, as in: "sizeof info" after "struct fileinfo info;" is declared).
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”）。

So use a space after these keywords:
所以在这些关键字之后放一个空格：
if, switch, case, for, do, while

but not with sizeof, typeof, alignof, or __attribute__. E.g.:
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);

Do not add spaces around (inside) parenthesized expressions. This example is *bad*:
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);

When declaring pointer data or a function that returns a pointer type, the preferred use of '*' is adjacent to the data name or function name and not adjacent to the type name. Examples:
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);

Use one space around (on each side of) most binary and ternary operators, such as any of these:
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：
= + - < > * / % | & ^ <= >= == != ? :

but no space after unary operators:
但是一元操作符后不要加空格：
& * + - ~ ! sizeof typeof alignof __attribute__ defined

no space before the postfix increment & decrement unary operators:
后缀自加和自减一元操作符前不加空格：
++ ---

and no space around the '.' and "->" structure member operators.
“**.**”和“**->**”结构体成员操作符前后不加空格。

Do not leave trailing whitespace at the ends of lines. Some editors with "smart" indentation will insert whitespace at the beginning of new lines as appropriate, so you can start typing the next line of code right away. However, some such editors do not remove the whitespace if you end up not putting a line of code there, such as if you leave a blank line. As a result, you end up with lines containing trailing whitespace.
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。

Git will warn you about patches that introduce trailing whitespace, and can optionally strip the trailing whitespace for you; however, if applying a series of patches, this may make later patches in the series fail by changing their context lines.
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。

Chapter 4: Naming （命名）

C is a Spartan language, and so should your naming be. Unlike Modula-2 and Pascal programmers, C programmers do not use cute names like ThisVariableIsATemporaryCounter. A C programmer would call that variable "tmp", which is much easier to write, and not the least more difficult to understand.
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。

HOWEVER, while mixed-case names are frowned upon, descriptive names for global variables are a must. To call a global function "foo" is a shooting offense.
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。

GLOBAL variables (to be used only if you _really_ need them) need to have descriptive names, as do global functions. If you have a function that counts the number of active users, you should call that "count_active_users()" or similar, you should _not_ call it "cntusr()".
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()”。

Encoding the type of a function into the name (so-called Hungarian notation) is brain damaged - the compiler knows the types anyway and can check those, and it only confuses the programmer. No wonder MicroSoft makes buggy programs.
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。

LOCAL variable names should be short, and to the point. If you have some random integer loop counter, it should probably be called "i". Calling it "loop_counter" is non-productive, if there is no chance of it being mis-understood. Similarly, "tmp" can be just about any type of variable that is used to hold a temporary value.
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。

If you are afraid to mix up your local variable names, you have another problem, which is called the function-growth-hormone-imbalance syndrome. See next chapter.
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）。

Chapter 5: Typedefs

Please don't use things like "vps_t".
不要使用类似“vps_t”之类的东西

It's a _mistake_ to use typedef for structures and pointers. When you see a
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;

in the source, what does it mean?
这代表什么意思呢?

In contrast, if it says
相反，如果是这样
struct virtual_container *a;

you can actually tell what "a" is.
你就知道“a”是什么了。

Lots of people think that typedefs "help readability". Not so. They are useful only for:
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. totally opaque objects (where the typedef is actively used to _hide_ what the object is).
   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）。

   Example: "pte_t" etc. opaque objects that you can only access using the proper accessor functions.
   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。

   NOTE! Opaqueness and "accessor functions" are not good in themselves. The reason we have them for things like pte_t etc. is that there really is absolutely _zero_ portably accessible information there.

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. Clear integer types, where the abstraction _helps_ avoid confusion whether it is "int" or "long".
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 are perfectly fine typedefs, although they fit into category (d) better than here.
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

NOTE! Again - there needs to be a _reason_ for this. If something is "unsigned long", then there's no reason to do
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;

but if there is a clear reason for why it under certain circumstances might be an "unsigned int" and under other configurations might be "unsigned long", then by all means go ahead and use a typedef.
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. when you use sparse to literally create a _new_ type for type-checking.
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. New types which are identical to standard C99 types, in certain exceptional circumstances.
和标准 C99 类型相同的类型，在某些例外的情况下

Although it would only take a short amount of time for the eyes and brain to become accustomed to the standard types like 'uint32_t', some people object to their use anyway.
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。

Therefore, the Linux-specific 'u8/u16/u32/u64' types and their signed equivalents which are identical to standard types are permitted -- although they are not mandatory in new code of your own.
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。

When editing existing code which already uses one or the other set of types, you should conform to the existing choices in that code.
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. Types safe for use in userspace.

可以在用户空间安全使用的类型。

In certain structures which are visible to userspace, we cannot require C99 types and cannot use the 'u32' form above. Thus, we use __u32 and similar types in all structures which are shared with userspace.

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。

Maybe there are other cases too, but the rule should basically be to NEVER EVER use a typedef unless you can clearly match one of those rules.

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。

In general, a pointer, or a struct that has elements that can reasonably be directly accessed should _never_ be a typedef.

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。

Chapter 6: Functions （函数）

Functions should be short and sweet, and do just one thing. They should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24, as we all know), and do one thing and do that well.

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。

The maximum length of a function is inversely proportional to the complexity and indentation level of that function. So, if you have a conceptually simple function that is just one long (but simple) case-statement, where you have to do lots of small things for a lot of different cases, it's OK to have a longer function.

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。

However, if you have a complex function, and you suspect that a less-than-gifted first-year high-school student might not even understand what the function is all about, you should adhere to the maximum limits all the more closely. Use helper functions with descriptive names (you can ask the compiler to in-line them if you think it's performance-critical, and it will probably do a better job of it than you would have done).

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

Page 11

效果往往会比你写一个复杂函数的效果要好。）

Another measure of the function is the number of local variables. They shouldn't exceed 5-10, or you're doing something wrong. Re-think the function, and split it into smaller pieces. A human brain can generally easily keep track of about 7 different things, anything more and it gets confused. You know you're brilliant, but maybe you'd like to understand what you did 2 weeks from now.
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。

In source files, separate functions with one blank line. If the function is exported, the EXPORT* macro for it should follow immediately after the closing function brace line. E.g.:
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);
```

In function prototypes, include parameter names with their data types. Although this is not required by the C language, it is preferred in Linux because it is a simple way to add valuable information for the reader.
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。

Chapter 7: Centralized exiting of functions (集中的函数退出途径)

Albeit deprecated by some people, the equivalent of the goto statement is used frequently by compilers in form of the unconditional jump instruction.
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。

The goto statement comes in handy when a function exits from multiple locations and some common work such as cleanup has to be done.
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。

The rationale is:
理由是：

- unconditional statements are easier to understand and follow
  无条件语句容易理解和跟踪
- nesting is reduced
  嵌套程度减小
- errors by not updating individual exit points when making modifications are prevented
  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
- saves the compiler work to optimize redundant code away ;)
  减轻了编译器的工作，无需删除冗余代码

```
int fun(int )
{
        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;
}
```

Chapter 8: Commenting (注释)

Comments are good, but there is also a danger of over-commenting. NEVER try to explain HOW your code works in a comment: it's much better to write the code so that the _working_ is obvious, and it's a waste of time to explain badly written code.
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。

Generally, you want your comments to tell WHAT your code does, not HOW. Also, try to avoid

Page 13

putting comments inside a function body: if the function is so complex that you need to separately comment parts of it, you should probably go back to chapter 5 for a while. You can make small comments to note or warn about something particularly clever (or ugly), but try to avoid excess. Instead, put the comments at the head of the function, telling people what it does, and possibly WHY it does it.
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。

When commenting the kernel API functions, please use the kernel-doc format. See the files Documentation/kernel-doc-nano-HOWTO.txt and scripts/kernel-doc for details.
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 style for comments is the C89 "/* ... */" style.
Linux 的注释风格是 C89“/* ... */”风格。

Don't use C99-style "// ..." comments.
不要使用 C99 风格“// ...”注释。

The preferred style for long (multi-line) comments is:
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.
 */
```

It's also important to comment data, whether they are basic types or derived types. To this end, use just one data declaration per line (no commas for multiple data declarations). This leaves you room for a small comment on each item, explaining its use.
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。

Chapter 9: You've made a mess of it (你已经把事情弄糟了)

That's OK, we all do. You've probably been told by your long-time Unix user helper that "GNU emacs" automatically formats the C sources for you, and you've noticed that yes, it does do that, but the defaults it uses are less than desirable (in fact, they are worse than random typing - an infinite number of monkeys typing into GNU emacs would never make a good program).
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）

So, you can either get rid of GNU emacs, or change it to use saner values. To do the latter, you can stick the following in your .emacs file:
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))
```

This will define the M-x linux-c-mode command. When hacking on a module, if you put the string -*- linux-c -*- somewhere on the first two lines, this mode will be automatically invoked. Also, you may want to add

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

to your .emacs file if you want to have linux-c-mode switched on automagically when you edit source files under /usr/src/linux.
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，你也可能需要添加到你的.emacs 文件里。

But even if you fail in getting emacs to do sane formatting, not everything is lost: use "indent".
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”。

Now, again, GNU indent has the same brain-dead settings that GNU emacs has, which is why you need to give it a few command line options. However, that's not too bad, because even the

makers of GNU indent recognize the authority of K&R (the GNU people aren't evil, they are just severely misguided in this matter), so you just give indent the options "-kr -i8" (stands for "K&R, 8 character indents"), or use "scripts/Lindent", which indents in the latest style.
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" has a lot of options, and especially when it comes to comment re-formatting you may want to take a look at the man page. But remember: "indent" is not a fix for bad programming.
“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。

Chapter 10: Kconfig configuration files (Kconfig 配置文件)

For all of the Kconfig* configuration files throughout the source tree, the indentation is somewhat different. Lines under a "config" definition are indented with one tab, while help text is indented an additional two spaces. Example:
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.
```

Features that might still be considered unstable should be defined as dependent on "EXPERIMENTAL":
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)"
  ...
```

while seriously dangerous features (such as write support for certain filesystems) should advertise this prominently in their prompt string:

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

...

For full documentation on the configuration files, see the file Documentation/kbuild/kconfig-language.txt.
要查看配置文件的完整文档，请看 Documentation/kbuild/kconfig-language.txt。

Chapter 11: Data structures (数据结构)

Data structures that have visibility outside the single-threaded environment they are created and destroyed in should always have reference counts. In the kernel, garbage collection doesn't exist (and outside the kernel garbage collection is slow and inefficient), which means that you absolutely _have_ to reference count all your uses.
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。

Reference counting means that you can avoid locking, and allows multiple users to have access to the data structure in parallel - and not having to worry about the structure suddenly going away from under them just because they slept or did something else for a while.
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。

Note that locking is _not_ a replacement for reference counting. Locking is used to keep data structures coherent, while reference counting is a memory management technique. Usually both are needed, and they are not to be confused with each other.
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。

Many data structures can indeed have two levels of reference counting, when there are users of different "classes". The subclass count counts the number of subclass users, and decrements the global count just once when the subclass count goes to zero.
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。

Examples of this kind of "multi-level-reference-counting" can be found in memory management ("struct mm_struct": mm_users and mm_count), and in filesystem code ("struct super_block": s_count and s_active).
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。

Remember: if another thread can find your data structure, and you don't have a reference count on it, you almost certainly have a bug.
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。

Chapter 12: Macros, Enums and RTL （宏，枚举和 RTL）

Names of macros defining constants and labels in enums are capitalized.
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

Enums are preferred when defining several related constants.
在定义几个相关的常量时，最好用枚举。

CAPITALIZED macro names are appreciated but macros resembling functions may be named in lower case.
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。

Generally, inline functions are preferable to macros resembling functions.
一般的，如果能写成内联函数就不要写成像函数的宏。

Macros with multiple statements should be enclosed in a do - while block:
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)
```

Things to avoid when using macros:
使用宏的时候应避免的事情：

1) macros that affect control flow:

影响控制流程的宏

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

is a _very_ bad idea. It looks like a function call but exits the "calling"
function; don't break the internal parsers of those who will read the code.
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) macros that depend on having a local variable with a magic name:
依赖于一个固定名字的本地变量的宏：

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

might look like a good thing, but it's confusing as hell when one reads the code and it's prone to breakage from seemingly innocent changes.
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) macros with arguments that are used as l-values: FOO(x) = y; will bite you if somebody e.g. turns FOO into an inline function.
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) forgetting about precedence: macros defining constants using expressions must enclose the expression in parentheses. Beware of similar issues with macros using parameters.
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)
```

The cpp manual deals with macros exhaustively. The gcc internals manual also covers RTL which is used frequently with assembly language in the kernel.
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。

Chapter 13: Printing kernel messages (打印内核消息)

Kernel developers like to be seen as literate. Do mind the spelling of kernel messages to make a good impression. Do not use crippled words like "dont"; use "do not" or "don't" instead. Make the messages concise, clear, and unambiguous.
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。

Kernel messages do not have to be terminated with a period.
内核信息不必以句号（译注：英文句号，即点）结束。

Printing numbers in parentheses (%d) adds no value and should be avoided.
在小括号里打印数字(%d)没有任何价值，应该避免这样做。

There are a number of driver model diagnostic macros in <linux/device.h> which you should use to make sure messages are matched to the right device and driver, and are tagged with the right level: dev_err(), dev_warn(), dev_info(), and so forth. For messages that aren't associated with a particular device, <linux/kernel.h> defines pr_debug() and pr_info().
<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()。

Coming up with good debugging messages can be quite a challenge; and once you have them, they can be a huge help for remote troubleshooting. Such messages should be compiled out when the DEBUG symbol is not defined (that is, by default they are not included). When you use dev_dbg() or pr_debug(), that's automatic. Many subsystems have Kconfig options to turn on -DDEBUG. A related convention uses VERBOSE_DEBUG to add dev_vdbg() messages to the ones already enabled by DEBUG.
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。

Chapter 14: Allocating memory (分配内存)

The kernel provides the following general purpose memory allocators: kmalloc(), kzalloc(), kcalloc(), and vmalloc(). Please refer to the API documentation for further information about them.
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。

The preferred form for passing a size of a struct is the following:
传递结构体大小的首选形式是这样的：
p = kmalloc(sizeof(*p), ...);

The alternative form where struct name is spelled out hurts readability and introduces an opportunity for a bug when the pointer variable type is changed but the corresponding sizeof that is passed to a memory allocator is not.
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。

Casting the return value which is a void pointer is redundant. The conversion from void pointer to any other pointer type is guaranteed by the C programming language.
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。

Chapter 15: The inline disease (内联弊病)

There appears to be a common misperception that gcc has a magic "make me faster" speedup option called "inline". While the use of inlines can be appropriate (for example as a means of replacing macros, see Chapter 12), it very often is not. Abundant use of the inline keyword leads to a much bigger kernel, which in turn slows the system as a whole down, due to a bigger icache footprint for the CPU and simply because there is less memory available for the pagecache. Just think about it; a pagecache miss causes a disk seek, which easily takes 5 miliseconds. There are a LOT of cpu cycles that can go into these 5 miliseconds.
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。

A reasonable rule of thumb is to not put inline at functions that have more than 3 lines of code in them. An exception to this rule are the cases where a parameter is known to be a compiletime constant, and as a result of this constantness you *know* the compiler will be able to optimize most of your function away at compile time. For a good example of this later case, see the kmalloc() inline function.
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。

Often people argue that adding inline to functions that are static and used only once is always a win since there is no space tradeoff. While this is technically correct, gcc is capable of inlining these automatically without help, and the maintenance issue of removing the inline when a second user appears outweighs the potential value of the hint that tells gcc to do something it would have done anyway.
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。

Chapter 16: Function return values and names (函数返回值及命名)

Functions can return values of many different kinds, and one of the most common is a value indicating whether the function succeeded or failed. Such a value can be represented as an error-code integer (-Exxx = failure, 0 = success) or a "succeeded" boolean (0 = failure, non-zero = success).
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）。

Mixing up these two sorts of representations is a fertile source of difficult-to-find bugs. If the C language included a strong distinction between integers and booleans then the compiler would find these mistakes for us... but it doesn't. To help prevent such bugs, always follow this convention:
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：

If the name of a function is an action or an imperative command, the function should return an error-code integer. If the name is a predicate, the function should return a "succeeded" boolean.
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。

For example, "add work" is a command, and the add_work() function returns 0 for success or -EBUSY for failure. In the same way, "PCI device present" is a predicate, and the pci_dev_present() function returns 1 if it succeeds in finding a matching device or 0 if it doesn't.
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。

All EXPORTed functions must respect this convention, and so should all public functions. Private (static) functions need not, but it is recommended that they do.
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。

Functions whose return value is the actual result of a computation, rather than an indication of whether the computation succeeded, are not subject to this rule. Generally they indicate failure by returning some out-of-range result. Typical examples would be functions that return pointers; they use NULL or the ERR_PTR mechanism to report failure.
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。

Chapter 17: Don't re-invent the kernel macros (不要重新发明内核宏)

The header file include/linux/kernel.h contains a number of macros that you should use, rather than explicitly coding some variant of them yourself. For example, if you need to calculate the length of an array, take advantage of the macro
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]))

Similarly, if you need to calculate the size of some structure member, use
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))

There are also min() and max() macros that do strict type checking if you need them. Feel free to peruse that header file to see what else is already defined that you shouldn't reproduce in your code.
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。

Chapter 18: Editor modelines and other cruft (编辑器模式行和其他需要罗嗦的事情)

Some editors can interpret configuration information embedded in source files, indicated with special markers. For example, emacs interprets lines marked like this:
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，emac 能够解释被标记成这样的行：

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

Or like this:
或者这样的：

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

Vim interprets markers that look like this:
Vim 能够解释这样的标记：

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

Do not include any of these in source files. People have their own personal editor configurations, and your source files should not override them. This includes markers for indentation and mode configuration. People may use their own custom mode, or may have some other magic method for making indentation work correctly.
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。

Appendix I: References

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/

--
Last updated on 2007-July-13.