**Linux kernel coding style**

# 缩进

制表符是8个字符，因此缩进也是8个字符。 有些异端运动试图使缩进4（甚至2！）个字符深，这类似于试图将PI的值定义为3。

理由：缩进背后的整个想法是明确定义一个控制块开始和结束的位置。 特别是当你连续观看你的屏幕20小时时，如果你有大的缩进，你会发现看到缩进是如何工作的。

现在，有些人会声称具有8个字符的缩进会使代码向右移动得太远，并且使得难以在80个字符的终端屏幕上阅读。 答案是，如果你需要3个以上的缩进级别，无论如何你都会被搞砸，并且应该修复你的程序。

简而言之，8字符缩进使事情变得更容易阅读，并且当您将功能嵌套得太深时，还可以提醒您注意。 听取了警告。

在switch语句中放宽多个缩进级别的首选方法是将switch与其从属case标签对齐在同一列中，而不是双缩进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;

}

除非你有隐藏的东西，否则不要在一行中放置多个语句：

if (condition) do\_this;

 do\_something\_everytime;

不要将多个作业放在一行上。 内核编码风格非常简单。 避免棘手的表达。

除了Kconfig之外的注释和文档之外，空格不会用于缩进，上面的示例被故意破坏。

获得一个体面的编辑器，不要在行尾留下空白。

# Breaking long lines and strings

编码风格都是在使用常用工具的时候具有可读性和可维护性。

行的长度限制是80列，这是一个强烈的首选限制。

超过80列的语句将被分解为合理的块，除非超过80列会显着增加可读性并且不会隐藏信息。 后代总是比父母短得多，而且大大地放在右边。 具有长参数列表的函数头也是如此。 但是，不要破坏用户可见的字符串，例如printk消息，因为这会破坏grep的能力。

# 放置大括号和空格

总是出现在C样式中的另一个问题是大括号的放置。 与缩进大小不同，选择一种放置策略的技术原因很少，但是正如先知Kernighan和Ritchie所示，我们首选的方法是将开放大括号（左大括号）放在最后，并将关闭大括号（右大括号） 放在行首，因此：

if (x is true) {

 we do y

}

这适用于所有非函数语句块（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;

}

但是，有一种特殊情况，即功能：它们在下一行的开始处具有左括号，因此：

int function(int x)

{

 body of function

}

全世界的异教徒都声称这种不一致性是......不一致的，但所有正确思考的人都知道（a）K＆R是对的，（b）K＆R是对的。 此外，函数无论如何都是特殊的（你不能将它们嵌套在C中）。

请注意，右括号在它自己的一行上是空的，除非在后面继续使用相同的语句，即在do语句或if语句中的其他语句，如下所示：

do {

 body of do-loop

} while (condition);

和

if (x == y) {

 ..

} else if (x > y) {

 ...

} else {

 ....

}

理由：K＆R。

另外，请注意，此大括号布局还可以最大限度地减少空行（或几乎空行）的数量，而不会损失可读性。 因此，由于屏幕上的新行的供应不是可再生资源（请考虑这里的25行终端屏幕），因此您有更多的空行来发表评论。

不要不必要地使用大括号，只要一个陈述就可以做到。

if (condition)

 action();

和

if (condition)

 do\_this();

else

 do\_that();

如果条件语句只有一个分支是单个语句，则这不适用; 在后一种情况下，在两个分支中使用大括号：

if (condition) {

 do\_this();

 do\_that();

} else {

 otherwise();

}

## 空格

使用空格的Linux内核风格主要取决于函数与关键字的使用情况。 （大多数）关键字后使用空格。 值得注意的例外是sizeof，typeof，alignof和\_\_attribute\_\_，它们看起来有些像函数（并且通常在Linux中用于括号中，尽管它们在语言中不是必需的，例如：声明了struct fileinfo info之后的sizeof info）。

因此，在这些关键字之后使用空格：

if, switch, case, for, do, while

但不包含sizeof，typeof，alignof或\_\_attribute\_\_。 例如。，

s = sizeof(struct file);

不要在括号内的表达式周围添加空格。 这个例子很糟糕：

s = sizeof( struct file );

当声明指针数据或返回指针类型的函数时，\*的首选用法与数据名称或函数名称相邻，而不与类型名称相邻。 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运算符周围（每边）使用一个空格，例如以下任何一个空格：

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

但一元运算符之后没有空格：

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

后缀增加和减少这两个一元运算符前没有空格：

++ --

前缀增加和减少一元运算符后没有空格：

++ --

在.和- >结构成员操作符周围没有空格。

不要在行尾留下尾随的空白。 一些具有智能缩进的编辑器会根据需要在新行的开头插入空格，以便您可以立即开始输入下一行代码。 但是，如果最终没有在其中放置一行代码，例如，如果留下空白行，则某些编辑器不会删除空白。 因此，最后会出现包含尾随空格的行。

Git会警告你有关引入尾随空白的补丁，并可以选择去除尾随的空白; 但是，如果应用一系列修补程序，则可能会通过更改其上下文行来使系列中的后续修补程序失败。

# 命名

C是斯巴达语言，你的命名应该如此。 与Modula-2和Pascal程序员不同，C程序员不使用像ThisVariableIsATemporaryCounter这样的可爱名字。 一个C程序员会调用这个变量tmp，这样写起来更容易，而且不难理解。

然而，虽然混合大小写的名字被忽略，但全局变量的描述性名称是必须的。 调用全局函数foo是一种射击攻击。

GLOBAL变量（仅在真正需要时才使用）需要具有描述性名称，全局函数也是如此。 如果你有一个函数来计算活动用户的数量，你应该调用count\_active\_users（）或类似的函数，你不应该把它叫做cntusr（）。

# 类型定义

请不要使用vps\_t之类的东西。 对结构和指针使用typedef是一个错误。 当你看到一个

vps\_t a;

在代码中，这是什么意思？ 相反，如果它说

struct virtual\_container \*a;

你能够实际上说出a是什么。

很多人认为typedefs有助于可读性。 并非如此。 它们仅适用于：

1. 完全不透明的对象（其中typedef主动用于隐藏对象）。

例如：pte\_t等不透明的对象，只能使用正确的存取函数访问。

注意

不透明和访问函数本身并不好。 我们拥有像pte\_t这样的东西的原因是，那里确实存在绝对零可移植的信息。

1. 清除整数类型，其中抽象有助于避免混淆，无论它是int还是long。

u8 / u16 / u32是非常好的类型定义，虽然它们比这里更适合类（d）。

注意

再次 - 这需要有一个原因。 如果一些东西没有签名，那么就没有理由去做

typedef unsigned long myflags\_t;

但是如果有明确的理由说明为什么它在某些情况下可能是unsigned int，并且在其他配置下可能是unsigned long，那么通过一切手段继续并使用typedef。

1. 当你使用稀疏字面创建一个新的类型检查类型。
2. 在某些特殊情况下，与标准C99类型相同的新类型。

尽管眼睛和大脑只需要很短的时间来适应像uint32\_t这样的标准类型，但有些人反对使用它们。

因此，特定于Linux的u8 / u16 / u32 / u64类型及其与标准类型相同的已签名等价物是允许的 - 尽管它们在您自己的新代码中不是强制性的。

编辑已使用一种或另一种类型的现有代码时，应该符合该代码中的现有选择。

1. 用户空间中使用的类型安全。

在用户空间可见的某些结构中，我们不能使用C99类型，也不能使用上面的u32表单。 因此，我们在与用户空间共享的所有结构中使用\_\_u32和类似的类型。

也许还有其他一些情况，但是规则基本上应该是永远不要使用typedef，除非你能明确地匹配其中的一个规则。

一般来说，一个指针或者具有可合理直接访问的元素的结构不应该是typedef。

# 函数

函数应该简短而甜蜜，并且只做一件事。它们应该放在一两个屏幕上（正如我们都知道的那样，ISO / ANSI屏幕尺寸是80x24），并且做一件事并且做得很好。

函数的最大长度与该函数的复杂度和缩进级别成反比。所以，如果你有一个概念上简单的函数，它只是一个很长（但很简单）的case-statement，你必须为很多不同的情况做很多小事情，可以有一个更长的函数。

但是，如果你的功能复杂，而且你怀疑一个没有天赋的一年级高中学生可能甚至不明白这个功能是什么，你应该更加严格地遵守最大限度的限制。使用具有描述性名称的助手函数（如果您认为性能至关重要，可以让编译器将它们串联起来，并且它可能比您做得更好）。

函数的另一个度量是局部变量的数量。他们不应该超过5-10，否则你做错了什么。重新思考该功能，并将其分成更小的部分。人类的大脑通常可以轻松地追踪大约7种不同的事物，而且更容易混淆。你知道你很聪明，但也许你想了解你从现在起2周后所做的一切。

在源文件中，用一个空白行分隔函数。 如果函数被导出，那么它的EXPORT宏应该紧跟在闭合函数的大括号之后。 例如。：

int system\_is\_up(void)

{

 return system\_state == SYSTEM\_RUNNING;

}

EXPORT\_SYMBOL(system\_is\_up);

在函数原型中，包含参数名称及其数据类型。 尽管这不是C语言所要求的，但它在Linux中是首选，因为它是向读者添加有价值信息的简单方法。

# 集中退出功能

尽管有些人不推荐使用goto语句，但编译器经常会以无条件跳转指令的形式使用goto语句。

当一个函数从多个位置退出时，goto语句会派上用场，并且必须完成一些常见工作（如清理）。 如果没有清理需要，那么直接返回。

选择标签名称，说明goto的作用或goto存在的原因。 一个好名字的例子可以是out\_free\_buffer：如果goto释放缓冲区。 避免使用像err1：和err2：这样的GW-BASIC名称，因为如果您添加或删除退出路径，您将不得不重新编号它们，并且无论如何都难以验证正确性。

使用gotos的基本原理是：

* 无条件的陈述更容易理解和遵循
* 嵌套减少
* 防止在进行修改时不更新个别出口点的错误
* 保存编译器工作以优化冗余代码;）

int fun(int a)

{

 int result = 0;

 char \*buffer;

 buffer = kmalloc(SIZE, GFP\_KERNEL);

 if (!buffer)

 return -ENOMEM;

 if (condition1) {

 while (loop1) {

 ...

 }

 result = 1;

 goto out\_free\_buffer;

 }

 ...

out\_free\_buffer:

 kfree(buffer);

 return result;

}

需要注意的常见类型错误是一个看起来像这样的错误：

err:

 kfree(foo->bar);

 kfree(foo);

 return ret;

此代码中的错误是在某些退出路径上，foo为NULL。 通常情况下，解决方法是将其分成两个错误标签err\_free\_bar：和err\_free\_foo ::

err\_free\_bar:

 kfree(foo->bar);

err\_free\_foo:

 kfree(foo);

 return ret;

理想情况下，您应该模拟错误来测试所有退出路径。

# 注释

注释很好，但也有过度注释的危险。 永远不要试图在评论中解释你的代码如何起作用：编写代码好得多，这样工作才是明显的，而且解释编写不好的代码是浪费时间。

一般来说，你希望你的注释告诉你的代码是做什么的，而不是如何。另外，尽量避免将注释放入函数体中：如果函数太复杂以至于需要单独评论它的某些部分，那么您应该回到第6章一段时间。 你可以做一些小的注释来提醒或警告某些特别聪明（或丑陋）的东西，但是尽量避免过多。 相反，将注释放在功能的头部，告诉人们它做了什么，可能是为什么它做到了。

在注释内核API函数时，请使用kernel-doc格式。 有关详细信息，请参见：ref：`Documentation / doc-guide / <doc\_guide>`和scripts / kernel-doc。

 长（多行）注释的首选样式是：

/\*

 \* 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.

 \*/

对于net /和drivers / net /中的文件，较长（多行）注释的首选样式稍有不同。

/\* The preferred comment style for files in net/ and drivers/net

 \* looks like this.

 \*

 \* It is nearly the same as the generally preferred comment style,

 \* but there is no initial almost-blank line.

 \*/

注释数据也很重要，无论它们是基本类型还是派生类型。 为此，每行仅使用一个数据声明（多个数据声明不使用逗号）。 这给你留下了每个项目的小评论，解释它的用途。

# You've made a mess of it

没关系，我们都这么做。 您可能已经被您的长期Unix用户助手告知，GNU emacs可以为您自动格式化C源代码，并且您已经注意到，是的，它确实这样做了，但它使用的默认值并不理想（实际上 ，它们比随机输入更糟糕 - 无数的猴子在GNU emacs中打字都不会成为一个好的程序）。

因此，您可以摆脱GNU emacs，或将其更改为使用saner值。 要做后者，你可以在你的.emacs文件中加入以下内容：

(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)

 (setq show-trailing-whitespace t)

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

这将使emacs在〜/ src / linux-trees下的C文件的内核编码风格上更好。

但即使你没有能够让emacs做出理智的格式化，也并非一切都会丢失：使用缩进。

现在，再次，GNU indent与GNU emacs具有相同的脑死亡设置，这就是为什么您需要给它几个命令行选项。 然而，这还不算太坏，因为即使是GNU indent的制造者也承认K＆R的权威性（GNU人不是邪恶的，他们在这件事上只是被严重误导了），所以你只要给出缩进选项-kr -i8（ 代表K＆R，8个字符的缩进），或使用脚本/ Lindent，以最新风格缩进。

缩进有很多选项，特别是当涉及到重新格式化评论时，您可能需要查看手册页。 但请记住：缩进不适用于糟糕的编程。

#  Kconfig configuration files

对于整个源代码树中的所有Kconfig \*配置文件，缩进都有所不同。 配置定义下的行用一个tab缩进，而帮助文本缩进另外两个空格。 例：

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.

严重危险的功能（例如对某些文件系统的写入支持）应在其提示字符串中突出显示：

config ADFS\_FS\_RW

 bool "ADFS write support (DANGEROUS)"

 depends on ADFS\_FS

 ...

有关配置文件的完整文档，请参阅Documentation / kbuild / kconfig-language.txt文件。

# 数据结构

在其创建和销毁的单线程环境之外具有可见性的数据结构应始终具有引用计数。在内核中，垃圾收集不存在（并且在内核垃圾收集之外速度慢并且效率低下），这意味着您绝对必须对所有的使用进行计数

引用计数意味着您可以避免锁定，并允许多个用户并行访问数据结构 - 而不必担心结构突然离开他们，因为他们只是睡了一会儿或做了一些其他事情。

请注意，锁定不是引用计数的替代方法。锁定用于保持数据结构的一致性，而引用计数是一种内存管理技术。通常两者都是需要的，他们不应该相互混淆。

当存在不同类别的用户时，许多数据结构确实可以具有两个级别的引用计数。子类计数计数子类用户的数量，并在子类计数为零时将全局计数递减一次。

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）和文件系统代码（struct super\_block：s\_count和s\_active）中找到。

请记住：如果另一个线程可以找到您的数据结构，并且您没有引用计数，那么您几乎肯定会有一个错误。

# Macros, Enums and RTL

在枚举中定义常量和标签的宏名称大写。

#define CONSTANT 0x12345

 定义几个相关的常量时首选枚举。

大写的宏名称被赞赏，但类似于函数的宏可以用小写字母命名。

通常，内联函数比类似函数的宏更可取。

具有多个语句的宏应放在do-while块中：

#define macrofun(a, b, c) \

 do { \

 if (a == 5) \

 do\_this(b, c); \

 } while (0)

 使用宏时应该避免的事情

1. 影响控制流程的宏：

#define FOO(x) \

 do { \

 if (blah(x) < 0) \

 return -EBUGGERED; \

 } while (0)

是一个非常糟糕的主意。 它看起来像一个函数调用，但退出了调用函数; 不要打破那些将阅读代码的人的内部解析器。

1. 取决于具有魔术名称的局部变量的宏：

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

可能看起来像一件好事，但当人们阅读代码时，它就像地狱一样令人困惑，而且很容易破坏看似无辜的变化。

1. 具有用作l-values的参数的宏：FOO（x）= y; 会有人咬你， 所以将FOO变成内联函数。
2. 忘记优先级：使用表达式定义常量的宏必须将表达式括在括号内。 请注意与使用参数的宏类似的问题。

#define CONSTANT 0x4000

#define CONSTEXP (CONSTANT | 3)

在类似函数的宏中定义局部变量时命名空间冲突：

#define FOO(x) \

({ \

 typeof(x) ret; \

 ret = calc\_ret(x); \

 (ret); \

})

ret是局部变量的通用名称 - \_\_foo\_ret不太可能与现有变量相冲突。

cpp手册详尽地讨论了宏。 gcc内部手册还涵盖了在内核中经常使用汇编语言的RTL。

# 打印内核信息

内核开发人员喜欢被认为是有学问的。不要介意内核消息的拼写，以留下好印象。不要使用像dont这样的残缺字眼;改用或不用。使信息简洁明了，毫不含糊。

内核消息不必以句点结束。

在括号（％d）中打印数字不会增加任何值，应该避免。

在<linux / device.h>中有许多驱动程序模型诊断宏，您应该使用它们来确保消息与正确的设备和驱动程序相匹配，并使用正确的级别进行标记：dev\_err（），dev\_warn（）， dev\_info（）等等。对于与特定设备无关的消息，<linux / printk.h>定义了pr\_notice（），pr\_info（），pr\_warn（），pr\_err（）等。

提供良好的调试信息可能是一个相当大的挑战;一旦你拥有了它们，它们可以为远程故障排除提供巨大的帮助。但是，调试消息打印的处理方式与打印其他非调试消息的方式不同。当其他pr\_XXX（）函数无条件打印时，pr\_debug（）不会;它会默认编译出来，除非定义了DEBUG或CONFIG\_DYNAMIC\_DEBUG。对于dev\_dbg（）也是如此，相关约定使用VERBOSE\_DEBUG将dev\_vdbg（）消息添加到已由DEBUG启用的消息中。

许多子系统都有Kconfig调试选项来打开相应的Makefile中的-DDEBUG;在其他情况下，特定文件#define DEBUG。当调试消息应该被无条件地打印时，例如它已经在调试相关的#ifdef节中，printk（KERN\_DEBUG ...）可以被使用。

# 分配内存

内核提供了以下通用内存分配器：kmalloc（），kzalloc（），kmalloc\_array（），kcalloc（），vmalloc（）和vzalloc（）。 有关它们的更多信息，请参阅API文档。

传递结构大小的首选形式如下：

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

struct name拼写的替代形式会伤害可读性，并且当指针变量类型发生更改但是传递给内存分配器的相应sizeof不是时，会引发bug的机会。

投射作为空指针的返回值是多余的。 C编程语言保证从void指针到任何其他指针类型的转换。

分配数组的首选形式如下：

p = kmalloc\_array(n, sizeof(...), ...);

分配归零数组的首选形式如下：

p = kcalloc(n, sizeof(...), ...);

两种形式都检查分配大小为n \* sizeof（...）的溢出，如果发生则返回NULL。

# The inline disease

似乎有一种常见的错误认识，即gcc有一种神奇的“让我更快”的内联加速选项。虽然使用内联可能是合适的（例如，作为替换宏的手段，请参阅第12章），但它通常不适用。对inline关键字的大量使用会导致更大的内核，从而导致整个系统的速度变慢，这是由于CPU的icache占用空间更大，并且仅仅因为页面缓存可用的内存更少。考虑一下;一个页面缓存缺失导致磁盘寻道，这很容易花费5毫秒。有很多cpu周期可以进入这5毫秒。

一个合理的经验法则是不要内联在其中有3行以上代码的函数中。这个规则的一个例外是一个参数被称为编译时常量的情况，由于这种常数，你知道编译器在编译时能够优化大部分函数。有关后面这种情况的一个很好的例子，请参阅kmalloc（）内联函数。

通常人们认为，将内联添加到静态函数并仅使用一次总是赢，因为没有空间折衷。虽然这在技术上是正确的，但是gcc能够在没有帮助的情况下自动内联这些内容，而当第二个用户出现时删除内联的维护问题超过了提示的潜在价值，该提示告诉gcc执行它本来会做的事情。

# Function return values and names

函数可以返回许多不同类型的值，其中最常见的值是指示函数是成功还是失败的值。 这样的值可以表示为错误代码整数（-Exxx =失败，0 =成功）或成功的布尔值（0 =失败，非零=成功）。

混合这两种表示方式是难以发现的错误的肥沃来源。 如果C语言包含整数和布尔值之间的强烈区别，那么编译器会为我们找到这些错误......但它不会。 为了帮助防止此类错误，请始终遵循以下约定：

如果一个函数的名字是一个动作或一个命令式命令，

该函数应该返回一个错误代码整数。 如果名字

是一个谓词，函数应该返回一个“成功”的布尔值。

例如，add work是一个命令，add\_work（）函数返回0表示成功，或者返回-EBUSY表示失败。 以同样的方式，当前PCI设备是一个谓词，如果pci\_dev\_present（）函数成功找到匹配设备，则返回1，否则返回0。

所有导出的函数都必须遵守这个约定，所有公共函数也应该如此。 私有（静态）功能不需要，但建议他们这样做。

函数的返回值是计算的实际结果，而不是计算是否成功的指示，不受此规则约束。 通常他们通过返回一些超出范围的结果来指示失败。 典型的例子是返回指针的函数; 他们使用NULL或ERR\_PTR机制来报告失败。

# 不要重新发明内核宏

头文件include / linux / kernel.h包含许多您应该使用的宏，而不是自己对它们的某些变体进行明确编码。 例如，如果您需要计算数组的长度，请利用宏

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

同样，如果您需要计算某个结构成员的大小，请使用

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

如果你需要它们，min（）和max（）宏也会进行严格的类型检查。 请随意仔细阅读该头文件，看看还有哪些已经定义的内容，您不应该在代码中进行复制。

# 编辑modelines和其他cruft

有些编辑可以解释源文件中嵌入的配置信息，用特殊标记表示。 例如，emacs解释标记为这样的行：

-\*- mode: c -\*-

或者像这样

/\*

Local Variables:

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

End:

\*/

Vim解释如下的标记：

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

不要在源文件中包含任何这些内容。 人们有他们自己的个人编辑器配置，你的源文件不应该覆盖它们。 这包括缩进和模式配置的标记。 人们可以使用自己的自定义模式，或者可以使用其他一些魔术方法来使缩进工作正确。

# 内联汇编

在特定于体系结构的代码中，您可能需要使用内联汇编来与CPU或平台功能进行交互。如有必要，请不要犹豫。但是，当C可以完成这项工作时，不要无偿地使用内联汇编。你可以并且应该尽可能地从C硬件中取出硬件。

考虑编写简单的帮助函数来包装内联程序集的常见位，而不是轻微地改变它们。请记住，内联汇编可以使用C参数。

大的，非平凡的汇编函数应该放入.S文件中，并在C头文件中定义相应的C原型。装配函数的C原型应该使用asmlinkage。

您可能需要将您的asm语句标记为volatile，以防止GCC在未发现任何副作用的情况下将其删除。但是，您并不总是需要这样做，而这样做会不必要地限制优化。

在编写包含多条指令的单个内联汇编语句时，将每条指令放在单独的引用字符串的单独行中，并使用\ n \ t结束每个字符串，除了最后一行以正确缩进汇编输出中的下一条指令：

asm ("magic %reg1, #42\n\t"

 "more\_magic %reg2, %reg3"

 : /\* outputs \*/ : /\* inputs \*/ : /\* clobbers \*/);

# 条件汇编

尽可能不要在.c文件中使用预处理器条件（#if，#ifdef）;这样做会使代码更难以阅读，逻辑更难以遵循。相反，在定义用于这些.c文件的函数的头文件中使用这些条件，在#else情况下提供无操作存根版本，然后从.c文件无条件地调用这些函数。编译器将避免为存根调用生成任何代码，产生相同的结果，但逻辑仍然易于遵循。

倾向于编译整个函数，而不是部分函数或表达式部分。不要在表达式中放入ifdef，而应将部分或全部表达式分解为单独的辅助函数，并将条件应用于该函数。

如果您的函数或变量可能在特定配置中未使用，并且编译器会警告其定义未使用，请将该定义标记为\_\_maybe\_unused，而不是将其包装在预处理器条件中。 （但是，如果某个函数或变量始终未被使用，请将其删除。）

在代码中，如果可能的话，使用IS\_ENABLED宏将Kconfig符号转换为C布尔表达式，并在正常的C条件中使用它：

if (IS\_ENABLED(CONFIG\_SOMETHING)) {

 ...

}

编译器将不断地将条件折叠起来，并像#ifdef那样包含或排除代码块，所以这不会增加任何运行时开销。 然而，这种方法仍然允许C编译器查看块内的代码，并检查它的正确性（语法，类型，符号引用等）。 因此，如果块内的代码引用不符合条件的符号，则仍然必须使用#ifdef。

在任何不重要的#if或#ifdef块（多于几行）结尾处，在同一行上的#endif后面放置注释，注意使用的条件表达式。 例如：

#ifdef CONFIG\_SOMETHING

...

#endif /\* CONFIG\_SOMETHING \*/

# Appendix 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).

The Practice of Programming by Brian W. Kernighan and Rob Pike. Addison-Wesley, Inc., 1999. ISBN 0-201-61586-X.

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 process/coding-style.rst, by greg@kroah.com at OLS 2002:<http://www.kroah.com/linux/talks/ols_2002_kernel_codingstyle_talk/html/>