GNU C 9条扩展语法

GNU C 9条扩展语法

GNC CC是一个功能非常强大的跨平台C编译器，它对标准C语言进行了一系列扩展，以增强标准C的功能，这些扩展对优化、目标代码布局、更安全的检查等方面提供了很强的支持。本文把支持GNU扩展的C语言称为GNU C。

Linux　内核代码使用了大量的GNU C扩展，以至于能够编译Linux内核的唯一编译器是GNU CC，以前甚至出现过编译Linux内核要使用特殊的GNU CC版本的情况。本文是对Linux内核使用的GNU C扩展的一个汇总，希望当你读内核源码遇到不理解的语法和语义时，能从本文找到一个初步的解答，更详细的信息可以查看gcc.info。文中的例子取自　Linux 2.4.18。

1、 零长度和变量长度数组

GNU C允许使用零长度数组，在定义变长对象的头结构时，这个特性非常有用。例如：

//include/linux/minix_fs.h

struct minix_dir_entry

{

_u16 inode;

char name[0];

};

结构的最后一个元素定义为零长度数组，它不占结构的空间。在标准C中则需要定义数组长度为1，分配时计算对象大小比较复杂。

GNU C 允许使用一个变量定义数组的长度，比如：

int n=0;

scanf("%d",&n);

int array[n];

2、 case范围

GNU C允许在一个case标号中指定一个连续范围的值，例如：

//arch/i386/kernel/irq.c

case '0' ... '9': c -= '0'; break;

　　case 'a' ... 'f': c -= 'a'-10; break;

　　case 'A' ... 'F': c -= 'A'-10; break;

3、语句表达式

GNU C把包含在括号中的复合语句看做是一个表达式，称为语句表达式，它可以出现在任何允许表达式的地方，你可以在语句表达式中使用循环、局部变量等，原本只能在复合语句中使用。例如：

//include/linux/kernel.h

#define min_t(type,x,y) ({ type __x = (x); type __y = (y); __x < __y ? __x: __y; })

//net/ipv4/tcp_output.c

int full_space = min_t(int, tp->window_clamp,tcp_full_space(sk));

复合语句的最后一个语句应该是一个表达式，它的值将成为这个语句表达式的值。

这里定义了一个安全的求最小值的宏，在标准C中，通常定义为：

#define min(x,y) ((x) < (y) ? (x) : (y))

这个定义计算x和y分别两次，当参数有副作用时，将产生不正确的结果，使用语句表达式只计算参数一次，避免了可能的错误。语句表达式通常用于宏定义。

4、typeof 关键字

使用前一节定义的宏需要知道参数的类型，利用typeof可以定义更通用的宏，不必事先知道参数的类型，例如：

//include/linux/kernel.h

#define min(x,y) ({ /

const typeof(x) _x = (x);　　/

const typeof(y) _y = (y);　　/

(void) (&_x == &_y);　　/

_x < _y ? _x : _y; })

这里typeof(x)表示x的值类型，第3行定义了一个与 x 类型相同的局部变量　_x 并初使化为　x，注意第5行的作用是检查参数x和y的类型是否相同。typeof 可以用在任何类型可以使用的地方，通常用于宏定义。

5、可变参数宏

在GNU C中，宏可以接受可变数目的参数，就象函数一样，例如

在GNU C中，宏可以接受可变数目的参数，就象函数一样，例如：

//include/linux/kernel.h
#define pr_debug(fmt,arg...) printk(KERN_DEBUG fmt,##arg)

这里arg表示其余的参数，可以是零个或多个，这些参数以及参数之间的逗号构成arg的值，在宏扩展时替换arg，例如：

pr_debug("%s:%d",filename,line)

扩展为

printk("<7>" "%s:%d", filename, line)

使用##的原因是处理arg不匹配任何参数的情况，这时arg的值为空，GNU C预处理器在这种特殊情况下，丢弃##之前的逗号，这样

pr_debug("success!/n")

扩展为

printk("<7>" "success!/n")

注意最后没有逗号。

6、标号元素

标准C要求数组或结构变量的初使化值必须以固定的顺序出现，在GNU C中，通过指定索引或结构域名，允许初始化值以任意顺序出现。指定数组索引的方法是在初始化值前写 '[INDEX] =，要指定一个范围使用'[FIRST ... LAST] ='的形式，例如：

//arch/i386/kernel/irq.c

static unsigned long irq_affinity [NR_IRQS] = { [0 ... NR_IRQS-1]= ~0UL };

将数组的所有元素初使化为~0UL，这可以看做是一种简写形式。

要指定结构元素，在元素值前写'FIELDNAME:'，例如：

//fs/ext2/file.c

struct file_operations ext2_file_operations = {

llseek: generic_file_llseek,

read: generic_file_read,

write: generic_file_write,

ioctl: ext2_ioctl,

mmap: generic_file_mmap,

open: generic_file_open,

release: ext2_release_file,

fsync: ext2_sync_file,

};

将结构ext2_file_operations的元素llseek初始化为generic_file_llseek，元素read初始化为genenric_file_read，依次类推。我觉得这是GNU C扩展中最好的特性之一，当结构的定义变化以至元素的偏移改变时，这种初始化方法仍然保证已知元素的正确性。对于未出现在初始化中的元素，其初值为 0。

标准C要求数组或结构体的初始化值必须以固定的顺序出现，在GNU C中，通过指定索引或结构体成员名，允许初始化以任意顺序出现。

unsigned char data[MAX] ={

[0]=10,

[10]=100,

};

struct file_operations ext2_file_operations={

open:ext2_open,

close:ext2_close,

};

在linux 2.6中推荐如下方式：

struct file_operations ext2_file_operations={

.read=ext2_read,

.write=ext2_write,

};

7、当前函数名

GNU C中预定义两个标志符保存当前函数的名字，__FUNCTION__保存函数在源码中的名字，__PRETTY__FUNCTION__保存带语言特色的名字。在C函数中这两个名字是相同的。在C++函数中，__PRETTY_FUNCTION__包括函数返回类型等额外信息，Linux内核只使用了__FUNCTION__。

//fs/ext2/super.c

void ext2_update_dynamic_rev(struct super_block *sb)

void ext2_update_dynamic_rev(struct super_block *sb)

{

struct ext2_super_block *es = EXT2_SB(sb)->s_es;

if (le32_to_cpu(es->s_rev_level) > EXT2_GOOD_OLD_REV)

return;

ext2_warning(sb, __FUNCTION__, "updating to rev %d because of new feature flag, " "running e2fsck is recommended", EXT2_DYNAMIC_REV);

这里__FUNCTION__将被替换为函数名ext2_update_dynamic_rev。虽然__FUNCTION__ 看起来类似于标准C中的__FILE__，但实际上__FUNCTION__是被编译器替换的，不象 __FILE__被预处理器替换。

在C99中支持__func__宏，因此建议使用__func__替代__FUNCTION__

8、特殊属性声明

GNU C允许声明函数、变量和类型的特殊属性，以便进行手工的代码优化和定制代码检查的方法。

要指定一个声明属性，只需要在声明后添加__attribute__((ATTRIBUTE))。其中ATTRIBUTE为属性说明，如果存在多个属性，则以逗号分隔。GNU C 支持noreturn、format、section、aligned、packed等十个属性。这里介绍最常用的：

noreturn属性用于函数，表示该函数从不返回。这可以让编译器生成稍微优化的代码，最重要的是可以消除不必要的警告信息比如未初使化的变量。例如：

//include/linux/kernel.h

#define ATTRIB_NORET __attribute__((noreturn)) ....

asmlinkage NORET_TYPE void do_exit(long error_code) ATTRIB_NORET;

format (ARCHETYPE, STRING-INDEX, FIRST-TO-CHECK)属性用于函数，表示该函数使用printf, scanf或strftime风格的参数，使用这类函数最容易犯的错误是格式串与参数不匹配，指定 format 属性可以让编译器根据格式串检查参数类型。例如：

//include/linux/kernel.h?

asmlinkage int printk(const char * fmt, ...) __attribute__ ((format (printf, 1, 2)));

表示第一个参数是格式串，从第二个参数起根据格式串检查参数。

unused属性用于函数和变量，表示该函数或变量可能不使用，这个属性可以避免编译器产生警告信息。

section ("section-name")属性用于函数和变量，通常编译器将函数放在.text区，变量放在.data区或.bss区，使用section属性，可以让编译器将函数或变量放在指定的节中。例如：

//include/linux/init.h

#define __init __attribute__ ((__section__ (".text.init")))

#define __exit __attribute__ ((unused,__section__(".text.exit")))

#define __initdata __attribute__ ((__section__ (".data.init")))

#define __exitdata __attribute__ ((unused, __section__(".data.exit")))

#define __initsetup __attribute__ ((unused,__section__(".setup.init")))

#define __init_call __attribute__ ((unused,__section__(".initcall.init")))

#define __exit_call __attribute__ ((unused,__section__(".exitcall.exit")))

连接器可以把相同节的代码或数据安排在一起，Linux内核很喜欢使用这种技术，例如系统的初始化代码被安排在单独的一个节，在初始化结束后就可以释放这部分内存。

aligned (ALIGNMENT)属性用于变量、结构或联合类型，指定变量、结构域、结构或联合的对齐量，以字节为单位，例如：

//include/asm-i386/processor.h

struct i387_fxsave_struct {

unsigned short cwd;

.....

} __attribute__ ((aligned (16)));

表示该结构类型的变量以16字节对齐。通常编译器会选择合适的对齐量，显示指定对齐通常是由于体系限制、优化等原因。

packed属性用于变量和类型，用于变量或结构域时表示使用最小可能的对齐，用于枚举、结构或联合类型时表示该类型使用最小的内存。例如：

//include/asm-i386/desc.h

struct Xgt_desc_struct {

unsigned short size;

unsigned long address __attribute__((packed));

};

域address将紧接着size分配。属性packed的用途大多是定义硬件相关的结构，使元素之间没有因对齐而造成的空洞。

9、内建函数

GNU C提供了大量的内建函数，其中很多是标准C库函数的内建版本，例如memcpy，它们与对应的C库函数功能相同，本文不讨论这类函数，其他内建函数的名字通常以__builtin开始。

内建函数__builtin_return_address(LEVEL)返回当前函数或其调用者的返回地址，参数LEVEL指定调用栈的级数，如0表示当前函数的返回地址，1表示当前函数调用者的返回地址，依此类推。例如：

//kernel/sched.c

printk(KERN_ERR "schedule_timeout: wrong timeout " "value %lx from %p/n", timeout,

__builtin_return_address(0));

内建函数__builtin_constant_p(EXP)用于判断一个值是否为编译时常数，如果参数EXP的值是常数，函数返回　1，否则返回　0。

// include/asm-i386/bitops.h

#define test_bit(nr,addr) /

(__builtin_constant_p(nr) ? /

constant_test_bit((nr),(addr)) : /

variable_test_bit((nr),(addr)))

很多计算或操作在参数为常数时有更优化的实现，在GNU C中用上面的方法可以根据参数是否为常数，只编译常数版本或非常数版本，这样既不失通用性，又能在参数是常数时编译出最优化的代码。

内建函数__builtin_expect(EXP, C)用于为编译器提供分支预测信息，其返回值是整数表达式EXP的值，C的值必须是编译时常数。例如：

//include/linux/compiler.h

#define likely(x)　　__builtin_expect((x),1)

#define unlikely(x)　　__builtin_expect((x),0)

// kernel/sched.c

if (unlikely(in_interrupt())) {

printk("Scheduling in interrupt/n");

BUG();

}

这个内建函数的语义是EXP的预期值是C，编译器可以根据这个信息适当地重排语句块的顺序，使程序在预期的情况下有更高的执行效率。上面的例子表示处于中断上下文是很少发生的，第6-7行的目标码可能会放在较远的位置，以保证经常执行的目标码更紧凑。

若不想使用GNU C扩展，那么只需要在gcc参数后面加上-ansi -pedantic即可，使用上述参数后，所有GNC C扩展语法部分将会有编译警报。

linux gcc的属性解析

GNU C的一大特色（却不被初学者所知）就是__attribute__机制。__attribute__可以设置函数属性（Function Attribute）、变量属性（Variable Attribute）和类型属性（Type Attribute）。

__attribute__书写特征是：__attribute__前后都有两个下划线，并切后面会紧跟一对原括弧，括弧里面是相应的__attribute__参数。

__attribute__语法格式为：

__attribute__ ((attribute-list))

其位置约束为：放于声明的尾部“；”之前。

函数属性（Function Attribute）

函数属性可以帮助开发者把一些特性添加到函数声明中，从而可以使编译器在错误检查方面的功能更强大。__attribute__机制也很容易同非GNU应用程序做到兼容之功效。

GNU CC需要使用–Wall编译器来击活该功能，这是控制警告信息的一个很好的方式。下面介绍几个常见的属性参数。

__attribute__ format

该__attribute__属性可以给被声明的函数加上类似printf或者scanf的特征，它可以使编译器检查函数声明和函数实际调用参数之间的格式化字符串是否匹配。该功能十分有用，尤其是处理一些很难发现的bug。

format的语法格式为：format (archetype, string-index, first-to-check)

format属性告诉编译器，按照printf, scanf, strftime或strfmon的参数表格式规则对该函数的参数进行检查。“archetype”指定是哪种风格；“string-index”指定传入函数的第几个参数是格式化字符串；“first-to-check”指定从函数的第几个参数开始按上述规则进行检查。

具体使用格式如下：

__attribute__((format(printf,m,n)))

__attribute__((format(scanf,m,n)))

其中参数m与n的含义为：

m：第几个参数为格式化字符串（format string）；

n：参数集合中的第一个，即参数“…”里的第一个参数在函数参数总数排在第几，注意，有时函数参数里还有“隐身”的呢，后面会提到；

在使用上，__attribute__((format(printf,m,n)))是常用的，而另一种却很少见到。下面举例说明，其中myprint为自己定义的一个带有可变参数的函数，其功能类似于printf：

```
//m=1；n=2
extern void myprint(const char *format,...) __attribute__((format(printf,1,2)));
//m=2；n=3
extern void myprint(int l，const char *format,...) __attribute__((format(printf,2,3)));
```

需要特别注意的是，如果myprint是一个函数的成员函数，那么m和n的值可有点“悬乎”了，例如：

```
//m=3；n=4
extern void myprint(int l，const char *format,...) __attribute__((format(printf,3,4)));
```

其原因是，类成员函数的第一个参数实际上一个“隐身”的“this”指针。（有点C++基础的都知道点this指针，不知道你在这里还知道吗？）

这里给出测试用例：attribute.c，代码如下：

```
extern void myprint(const char *format,...) __attribute__((format(printf,1,2)));
void test()
{
myprint("i=%d\",6);
 myprint("i=%s\",6);
 myprint("i=%s\","abc");
myprint("%s,%d,%d\",1,2);
}
```

运行$gcc –Wall –c attribute.c attribute后，输出结果为：

```
attribute.c: In function `test':
attribute.c:7: warning: format argument is not a pointer (arg 2)
attribute.c:9: warning: format argument is not a pointer (arg 2)
attribute.c:9: warning: too few arguments for format
```

如果在attribute.c中的函数声明去掉__attribute__((format(printf,1,2)))，再重新编译，即运行$gcc –Wall –c attribute.c attribute后，则并不会输出任何警告信息。

注意，默认情况下，编译器是能识别类似printf的“标准”库函数。

__attribute__ noreturn

该属性通知编译器函数从不返回值，当遇到类似函数需要返回值而却不可能运行到返回值处就已经退出来的情况，该属性可以避免出现错误信息。C库函数中的abort()和exit()的声明格式就采用了这种格式，如下所示：

```
extern void exit(int)   __attribute__((noreturn));
extern void abort(void) __attribute__((noreturn));
```

为了方便理解，大家可以参考如下的例子：

```
extern void myexit();
int test(int n)
{
if ( n > 0 )
   {
   myexit();
      /* 程序不可能到达这里*/
   }
   else
{
}
   return 0;
}
```

编译显示的输出信息为：

```
$ gcc –Wall –c noreturn.c
noreturn.c: In function `test':
noreturn.c:12: warning: control reaches end of non-void function
```

警告信息也很好理解，因为你定义了一个有返回值的函数test却有可能没有返回值，程序当然不知道怎么办了！

加上 attribute ((noreturn))则可以很好的处理类似这种问题。把extern void myexit()修改为：extern void myexit() attribute ((noreturn))之后，编译不会再出现警告

信息。

__attribute__ const

该属性只能用于带有数值类型参数的函数上。当重复调用带有数值参数的函数时，由于返回值是相同的，所以此时编译器可以进行优化处理，除第一次需要运算外，其它只需要返回第一次的结果就可以了，进而可以提高效率。该属性主要适用于没有静态状态（static state）和副作用的一些函数，并且返回值仅仅依赖输入的参数。

为了说明问题，下面举个非常“糟糕”的例子，该例子将重复调用一个带有相同参数值的函数，具体如下：

```
extern int square(int n) __attribute__((const));

...

   for (i = 0; i < 100; i++ )

   {

       total += square(5) + i;

   }
```

通过添加__attribute__((const))声明，编译器只调用了函数一次，以后只是直接得到了相同的一个返回值。

事实上，const参数不能用在带有指针类型参数的函数中，因为该属性不但影响函数的参数值，同样也影响到了参数指向的数据，它可能会对代码本身产生严重甚至是不可恢复的严重后果。

并且，带有该属性的函数不能有任何副作用或者是访问全局或静态变量，所以，类似getchar()或time()的函数是不适合使用该属性的。

同时使用多个属性

可以在同一个函数声明里使用多个__attribute__，并且实际应用中这种情况是十分常见的。使用方式上，你可以选择两个单独的__attribute__，或者把它们写在一起，可以参考下面的例子：

```
/* 把类似printf的消息传递给stderr 并退出   */

extern void die(const char *format, ...) __attribute__((noreturn)) __attribute__((format(printf, 1, 2)));
```

或者写成

```
extern void die(const char *format, ...) __attribute__((noreturn, format(printf, 1, 2)));
```

如果带有该属性的自定义函数追加到库的头文件里，那么所以调用该函数的程序都要做相应的检查。

和非GNU编译器的兼容性

庆幸的是，__attribute__设计的非常巧妙，很容易作到和其它编译器保持兼容，也就是说，如果工作在其它的非GNU编译器上，可以很容易的忽略该属性。即使__attribute__使用了多个参数，也可以很容易的使用一对圆括弧进行处理，例如：

```
/* 如果使用的是非GNU C, 那么就忽略__attribute__ */

#ifndef __GNUC__

#define  __attribute__ (x)  /*NOTHING*/

#endif
```

需要说明的是，__attribute__适用于函数的声明而不是函数的定义。所以，当需要使用该属性的函数时，必须在同一个文件里进行声明，例如：

```
/* 函数声明   */

void die(const char *format, ...) __attribute__((noreturn)) __attribute__((format(printf,1,2)));

 void die(const char *format, ...)

{

        /* 函数定义   */

}
```

变量属性（Variable Attributes）

关键字__attribute__也可以对变量（variable）或结构体成员（structure field）进行属性设置。这里给出几个常用的参数的解释，更多的参数可参考本文给出的连接。

在使用__attribute__参数时，你也可以在参数的前后都加上“__”（两个下划线），例如，使用__aligned__而不是aligned，这样，你就可以在相应的头文件里使用它而不用关心头文件里是否有重名的宏定义。

aligned (alignment)

该属性规定变量或结构体成员的最小的对齐格式，以字节为单位。例如：

```
int x __attribute__ ((aligned (16))) = 0;
```

编译器将以16字节（注意是字节byte不是位bit）对齐的方式分配一个变量。也可以对结构体成员变量设置该属性，例如，创建一个双字对齐的int对，可以这么写：

```
struct foo { int x[2] __attribute__ ((aligned (8))); };
```

如上所述，你可以手动指定对齐的格式，同样，你也可以使用默认的对齐方式。如果aligned后面不紧跟一个指定的数字值，那么编译器将依据你的目标机器情况使用最大最有益的对齐方式。例如：

```
short array[3] __attribute__ ((aligned));
```

选择针对目标机器最大的对齐方式，可以提高拷贝操作的效率。

aligned属性使被设置的对象占用更多的空间，相反的，使用packed可以减小对象占用的空间。

需要注意的是，attribute属性的效力与你的连接器也有关，如果你的连接器最大只支持16字节对齐，那么你此时定义32字节对齐也是无济于事的。

packed

使用该属性可以使得变量或者结构体成员使用最小的对齐方式，即对变量是一字节对齐，对域（field）是位对齐。

下面的例子中，x成员变量使用了该属性，则其值将紧放置在a的后面：

```
struct test
{
char a;
int x[2] __attribute__ ((packed));
};
```

其它可选的属性值还可以是：cleanup，common，nocommon，deprecated，mode，section，shared，tls_model，transparent_union，unused，vector_size，weak，dllimport等，详细信息可参考： http://gcc.gnu.org/onlinedocs/gcc-4.0.0/gcc/Variable-Attributes.html#Variable-Attributes

类型属性（Type Attribute）

关键字__attribute__也可以对结构体（struct）或共用体（union）进行属性设置。大致有六个参数值可以被设定，即：
aligned, packed, transparent_union, unused, deprecated 和 may_alias。

在使用__attribute__参数时，你也可以在参数的前后都加上“__”（两个下划线），例如，使用__aligned__而不是aligned，这样，你就可以在相应的头文件里使用它而不用关心头文件里是否有重名的宏定义。

aligned (alignment)

该属性设定一个指定大小的对齐格式（以字节为单位），例如：

```
struct S { short f[3]; } __attribute__ ((aligned (8)));
typedef int more_aligned_int __attribute__ ((aligned (8)));
```

该声明将强制编译器确保（尽它所能）变量类型为struct S或者more-aligned-int的变量在分配空间时采用8字节对齐方式。

如上所述，你可以手动指定对齐的格式，同样，你也可以使用默认的对齐方式。如果aligned后面不紧跟一个指定的数字值，那么编译器将依据你的目标机器情况使用最大最有益的对齐方式。例如：

```
struct S { short f[3]; } __attribute__ ((aligned));
```

这里，如果sizeof（short）的大小为2（byte），那么，S的大小就为6。取一个2的次方值，使得该值大于等于6，则该值为8，所以编译器将设置S类型的对齐方式为8字节。

aligned属性使被设置的对象占用更多的空间，相反的，使用packed可以减小对象占用的空间。

需要注意的是，attribute属性的效力与你的连接器也有关，如果你的连接器最大只支持16字节对齐，那么你此时定义32字节对齐也是无济于事的。

packed

使用该属性对struct或者union类型进行定义，设定其类型的每一个变量的内存约束。当用在enum类型定义时，暗示了应该使用最小完整的类型（it indicates that the smallest integral type should be used）。

下面的例子中，my-packed-struct类型的变量数组中的值将会紧紧的靠在一起，但内部的成员变量s不会被“pack”，如果希望内部的成员变量也被packed的话，my-unpacked-struct也需要使用packed进行相应的约束。

```
struct my_unpacked_struct
{
     char c;
     int i;
};
struct my_packed_struct
{
    char c;
    int  i;
    struct my_unpacked_struct s;
}__attribute__ ((__packed__));
```

变量属性与类型属性举例

下面的例子中使用__attribute__属性定义了一些结构体及其变量，并给出了输出结果和对结果的分析。程序代码为：

```
struct p
{
   int a;
```

```
    char b;
    char c;
}__attribute__((aligned(4))) pp;
struct q
{
    int a;
    char b;
    struct p qn;
    char c;
}__attribute__((aligned(8))) qq;
int main()
{
    printf("sizeof(int)=%d,sizeof(short)=%d.sizeof(char)=%d\n", sizeof(int),sizeof(short),sizeof(char));
    printf("pp=%d,qq=%d \n", sizeof(pp),sizeof(qq));
    return 0;
}
```

输出结果：

sizeof(int)=4,sizeof(short)=2.sizeof(char)=1

pp=8,qq=24

分析：

sizeof(pp):sizeof(a)+ sizeof(b)+ sizeof(c)=4+1+1=6<2 3 =8= sizeof(pp)

sizeof(qq):sizeof(a)+ sizeof(b)=4+1=5

sizeof(qn)=8 ;即qn是采用8字节对齐的，所以要在a，b后面添3个空余字节，然后才能存储qn，4+1+（3）+8+1=17因为qq采用的对齐是8字节对齐，所以qq的大小必定是8的整数倍，即qq的大小是一个比17大又是8的倍数的一个最小值，由此得到17<24+8=24= sizeof(qq)

Linux内核使用的GNU C扩展

和Unix一样，Linux内核也是用C语言实现的。谈到C，几乎所有的人都会立即想到ANSI C标准。但是Linux内核的实现，其实并不完全符合ANSI C标准。实际上，内核开发者总会使用许多gcc提供的C语言的扩展部分。

内核开发者使用的C语言涵盖了ISO C99标准和GNU C的扩展特性，我想，其中让人感兴趣的，应该不在于C99标准上，而是在于它的GNU C扩展特性上。下面，我们就一起来学学内核使用的GNU C扩展特性吧。

1. 内联函数

GNU的C编译器支持内联函数，这一点和ANSI标准完全不一样。在ANSI C中是没有inline这个关键字的。内联函数会在函数调用的地方直接把函数体展开，这样就可以减少函数调用的开销了（寄存器的保存和恢复）。而且，由于编译器会把调用函数的代码和函数本身的代码放在一起优化，所以也会有进一步优化代码的可能。当然，天底下没有白吃的午餐，这样做也是有代价的，那就是生成的代码会变长，这就意味着你必须使用更多的内存空间或更多的指令缓存来执行代码。内核开发者通常把那些对时间要求较高，而本身长度又较短的函数定义成内联函数。当然了，对于大块头的程序，你想把它定义成内联函数也没人反对。只不过，有这必要么？

定义一个内联函数，需要使用static作为关键字，并且用inline限定它。如：

static inline void foo() {...}

内联函数必须在使用之前就定义好，否则编译器没法儿将之展开。由于使用static关键字进行限制，编译时不会为内联函数单独建立一个函数体。内联函数一般定义在头文件中，当然了，如果你仅仅在某个源文件中使用内联函数，也可以把它定义在源文件的开头部分。

在内核中，为了类型安全的原因，优先使用内联函数而不是复杂的宏。

2. 内联汇编

gcc支持在C函数中嵌入汇编指令。当然，在内核编程的时候，只有知道对应的体系结构，才能使用这个功能。因为，不同的体系结构，其汇编指令往往是有很大差异的。

Linux的内核混合使用了C和汇编语言。在偏近体系结构的底层或对执行时间要求严格的地方，一般使用的是汇编语言。而内核其他部分的大部分代码则都是C语言写的。

内嵌汇编的语法如下：

__asm__(汇编语句模板: 输出部分: 输入部分: 破坏描述部分)

共四个部分：汇编语句模板，输出部分，输入部分，破坏描述部分，各部分使用“:”格开，汇编语句模板必不可少，其他三部分可选，如果使用了后面的部分，而前面部分为空，也需要用“:”格开，相应部分内容为空。例如：

__asm__ __volatile__("cli": : :"memory")

1、汇编语句模板

汇编语句模板由汇编语句序列组成，语句之间使用“;”、“\n”或“\n\t”分开。指令中的操作数可以使用占位符引用C语言变量，操作数占位符最多10个，名称如下：%0，%1，…，%9。指令中使用占位符表示的操作数，总被视为long型（4个字节），但对其施加的操作根据指令可以是字或者字节，当把操作数当作字或者字节

使用时，默认为低字或者低字节。对字节操作可以显式的指明是低字节还是次字节。方法是在%和序号之间插入一个字母，“b”代表低字节，“h”代表高字节，例如：%h1。

2、输出部分

输出部分描述输出操作数，不同的操作数描述符之间用逗号格开，每个操作数描述符由限定字符串和C 语言变量组成。每个输出操作数的限定字符串必须包含“=”表示他是一个输出操作数。 例：

__asm__ __volatile__("pushfl ; popl %0 ; cli":"=g" (x))

描述符字符串表示对该变量的限制条件，这样GCC 就可以根据这些条件决定如何分配寄存器，如何产生必要的代码处理指令操作数与C表达式或C变量之间的联系。

3、输入部分

输入部分描述输入操作数，不同的操作数描述符之间使用逗号格开，每个操作数描述符由限定字符串和C语言表达式或者C语言变量组成。 例如：

__asm__ __volatile__ ("lidt %0" : : "m" (real_mode_idt));

4、限制字符

4.1、限制字符列表

限制字符有很多种，有些是与特定体系结构相关，此处仅列出常用的限定字符和i386中可能用到的一些常用的限定符。它们的作用是指示编译器如何处理其后的C语言变量与指令操作数之间的关系。

分类	限定符	描述
通用寄存器	a	将输入变量放入eax，这里有一个问题：假设eax已经被使用，那怎么办？其实很简单：因为GCC 知道eax 已经被使用，它在这段汇编代码 的起始处插入一条语句pushl %eax，将eax 内容保存到堆栈，然 后在这段代码结束处再增加一条语句popl %eax，恢复eax的内容
	b	将输入变量放入ebx
	c	将输入变量放入ecx
	d	将输入变量放入edx
	s	将输入变量放入esi
	d	将输入变量放入edi
	q	将输入变量放入eax，ebx，ecx，edx中的一个
	r	将输入变量放入通用寄存器，也就是eax，ebx，ecx，edx，esi，edi中的一个
	A	把eax和edx合成一个64 位的寄存器(use long longs)
内存	m	内存变量
	o	操作数为内存变量，但是其寻址方式是偏移量类型，也即是基址寻址，或者是基址加变址寻址
	V	操作数为内存变量，但寻址方式不是偏移量类型
	缺	操作数为内存变量，但寻址方式为自动增量
	p	操作数是一个合法的内存地址（指针）
寄存器或内存	g	将输入变量放入eax，ebx，ecx，edx中的一个或者作为内存变量
	X	操作数可以是任何类型
立即数	I	0-31之间的立即数（用于32位移位指令）
	J	0-63之间的立即数（用于64位移位指令）
	N	0-255之间的立即数（用于out指令）
	i	立即数
	n	立即数，有些系统不支持除字以外的立即数，这些系统应该使用“n”而不是“i”
匹配	0,1... 9	表示用它限制的操作数与某个指定的操作数匹配，也即该操作数就是指定的那个操作数，例如“0” 去描述“％1”操作数，那么“%1”引用的其实就是“%0”操作数，注意作为限定符字母的0－9 与指令中的“％0”－“％9”的区别，前者描述操作数，后者代表操作数。
	&	该输出操作数不能使用过和输入操作数相同的寄存器
操作数类型	=	操作数在指令中是只写的（输出操作数）
	+	操作数在指令中是读写类型的（输入输出操作数）
浮点数	f	浮点寄存器
	t	第一个浮点寄存器
	u	第二个浮点寄存器
	G	标准的80387浮点常数
	%	该操作数可以和下一个操作数交换位置例如addl的两个操作数可以交换顺序 （当然两个操作数都不能是立即数）
	#	部分注释，从该字符到其后的逗号之间所有字母被忽略
	*	表示如果选用寄存器，则其后的字母被忽略

5、破坏描述部分

3、破坏描述部分

破坏描述符用于通知编译器我们使用了哪些寄存器或内存，由逗号格开的字符串组成，每个字符串描述一种情况，一般是寄存器名；除寄存器外还有“memory”。例如：“%eax”，“%ebx”，“memory”等。

3. 分支声明

对于条件选择语句，gcc内建了一条指令用于优化，在一个条件经常出现的时候，或者该条件很少出现的时候，编译器可以根据这条指令对分支进行优化。内核把这条指令封装成了宏，比如likely()和unlikely()，这样使用起来非常方便。

例如，下面是一个条件选择语句：

```
if (foo) {
    /* ... */
}
```

如果我们要把这个选择标记成绝少发生的分支，我们可以这样做：

```
/* 我们认为foo绝大部分时间都会为0 */
if (unlikely(foo)) {
    /* ... */
}
```

相反，如果我们要把这个选择标记成绝大部分时间都会发生的分支，可以这样做：

```
/* 我们认为foo通常都不会为0 */
if (likely(foo)) {
    /* ... */
}
```

在想要对某个条件选择语句进行优化之前，一定要搞清楚其中是不是存在这么一个条件，在绝大多数情况下都会成立。这点十分重要：如果你的判断正确，这个条件确实占压倒性的地位，那么性能就会提升，相反，如果你搞错了，性能反而会下降。在对一些错误条件进行判断的时候，常常会用到likely()和unlikely()宏。我们看到，unlikely()在内核中得到了广泛的应用，因为if语句往往用于判断一种特殊情况，而这种情况是绝少发生的。