AT&T ASM

开发一个 OS，尽管绝大部分代码只需要用 C/C++等高级语言就可以了，但至少和硬件相关部分的代码需要使用汇编语言，另外，由于启动部分的代码有大小限制，使用精练的汇编可以缩小目标代码的尺寸。另外，对于某些需要被经常调用的代码，使用汇编可以提高性能。所以我们必须了解汇编语言，即使你有可能并不喜欢它。

如果你是计算机专业的话，在大学里你应该学习过 Intel 格式的 8086/80386 汇编，这里就不再讨论。如果我们选择的 OS 开发工具是 GCC 以及 GAS 的话，就必须了解 AT&T 汇编语言语法，因为 GCC/GAS 只支持这种汇编语法。

本书不会去讨论 8086/80386 的汇编编程，这类的书籍很多，你可以参考它们。这里只会讨论 AT&T 的汇编语法，以及 GCC 的内嵌汇编语法。

1. Syntax

Register Reference

- 引用寄存器要在寄存器号前加百分号%,如“ movl %eax, %ebx”。
- 80386 有如下寄存器：
- 8 个 32-bit 寄存器 %eax，%ebx，%ecx，%edx，%edi，%esi，%ebp，%esp；
- 8 个 16-bit 寄存器，它们事实上是上面 8 个 32-bit 寄存器的低 16 位：%ax，%bx，%cx，%dx，%di，%si，%bp，%sp；
- 8 个 8-bit 寄存器：%ah，%al，%bh，%bl，%ch，%cl，%dh，%dl。它们事实上是寄存器%ax，%bx，%cx，%dx 的高 8 位和低 8 位；
- 6 个段寄存器：%cs(code)，%ds(data)，%ss(stack), %es，%fs，%gs；
- 3 个控制寄存器：%cr0，%cr2，%cr3；
- 6 个 debug 寄存器：%db0，%db1，%db2，%db3，%db6，%db7；
- 2 个测试寄存器：%tr6，%tr7；
- 8 个浮点寄存器栈：%st(0)，%st(1)，%st(2)，%st(3)，%st(4)，%st(5)，%st(6)，%st(7)。

Operator Sequence

操作数排列是从源（左）到目的（右），如“movl %eax(**源**）, %ebx(**目的**）”

Immediately Operator

使用立即数，要在数前面加符号$，**如**“movl $0x04, %ebx”

或者：

para = 0x04

movl $para, %ebx

指令执行的结果是将立即数 04h **装入寄存器** ebx**。**

Symbol Constant

符号常数直接引用 如

value: .long 0x12a3f2de

movl value , %ebx

指令执行的结果是将常数 0x12a3f2de **装入寄存器** ebx**。**

引用符号地址在符号前加符号$，**如**“movl $value, % ebx”**则是将符号** value **的地址装入寄存器** ebx**。**

Length of Operator

操作数的长度用加在指令后的符号表示 b(byte, 8-bit), w(word, 16-bits), l(long, 32-bits)，**如**“movb %al, %bl”，“movw %ax, %bx”，“movl %eax, %ebx ”。

如果没有指定操作数长度的话，编译器将按照目标操作数的长度来设置。比如指令“mov %ax, %bx”**，由于目标操作数** bx **的长度为** word**，那么编译器将把此指令等同于**“movw %ax, %bx”**。同样道理，指令**“mov $4, %ebx”**等同于指令**“movl $4, %ebx”，“push %al”**等同于**“pushb %al”**。对于没有指定操作数长度，但编译器又无法猜测的指令，编译器将会报错，比如指令**“push $4”**。

Sign and Zero Extension

绝大多数面向 80386 的 AT&T 汇编指令与 Intel 格式的汇编指令都是相同的，符号扩展指令和零扩展指令则是仅有的不同格式指令。

符号扩展指令和零扩展指令需要指定源操作数长度和目的操作数长度，即使在某些指令中这些操作数是隐含的。

在 AT&T 语法中，符号扩展和零扩展指令的格式为，基本部分"movs"和"movz"（对应 Intel 语法的 movsx 和 movzx），后面跟上源操作数长度和目的操作数长度。movsbl 意味着 movs （from）byte （to）long；movbw 意味着 movs （from）byte （to）word；movswl 意味着 movs （from）word （to）long。对于 movz 指令也一样。比如指令“movsbl %al, %edx”意味着将 al 寄存器的内容进行符号扩展后放置到 edx 寄存器中。

其它的 Intel 格式的符号扩展指令还有：

- cbw -- sign-extend byte in %al to word in %ax；
- cwde -- sign-extend word in %ax to long in %eax；
- cwd -- sign-extend word in %ax to long in %dx:%ax；
- cdq -- sign-extend dword in %eax to quad in %edx:%eax；

对应的 AT&T 语法的指令为 cbtw，cwtl，cwtd，cltd。

Call and Jump

段内调用和跳转指令为"call"，"ret"和"jmp"，段间调用和跳转指令为"lcall"，"lret"和"ljmp"。

段间调用和跳转指令的格式为“lcall/ljmp $SECTION, $OFFSET”，而段间返回指令则为“lret $STACK-ADJUST”。

Prefix

操作码前缀被用在下列的情况：

- 字符串重复操作指令(rep,repne)；
- 指定被操作的段(cs,ds,ss,es,fs,gs)；
- 进行总线加锁(lock)；
- 指定地址和操作的大小(data16,addr16)；

在 AT&T 汇编语法中，操作码前缀通常被单独放在一行，后面不跟任何操作数。例如，对于重复 scas 指令，其写法为：

```
repne
scas
```

上述操作码前缀的意义和用法如下：

- 指定被操作的段前缀为 cs,ds,ss,es,fs,和 gs。在 AT&T 语法中，只需要按照 section:memory-operand 的格式就指定了相应的段前缀。比如：lcall %cs:realmode_swtch
- 操作数／地址大小前缀是“data16”和"addr16"，它们被用来在 32-bit 操作数／地址代码中指定 16-bit 的操作数／地址。
- 总线加锁前缀“lock”，它是为了在多处理器环境中，保证在当前指令执行期间禁止一切中断。这个前缀仅仅对 ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG,DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD,XCHG 指令有效，如果将 Lock 前缀用在其它指令之前，将会引起异常。
- 字符串重复操作前缀"rep","repe","repne"用来让字符串操作重复“%ecx”次。

Memory Reference

Intel 语法的间接内存引用的格式为：

section:[base+index*scale+displacement]

而在 AT&T 语法中对应的形式为：

section:displacement(base,index,scale)

其中，base 和 index 是任意的 32-bit base 和 index 寄存器。scale 可以取值 1，2，4，8。如果不指定 scale 值，则默认值为 1。section 可以指定任意的段寄存器作为段前缀，默认的段寄存器在不同的情况下不一样。如果你在指令中指定了默认的段前缀，则编译器在目标代码中不会产生此段前缀代码。

下面是一些例子：

-4(%ebp)：base=%ebp，displacement=-4，section 没有指定，由于 base＝%ebp，所以默认的 section=%ss，index,scale 没有指定，则 index 为 0。

foo(,%eax,4)：index=%eax，scale=4，displacement=foo。其它域没有指定。这里默认的 section=%ds。

foo(,1)：这个表达式引用的是指针 foo 指向的地址所存放的值。注意这个表达式中没有

base 和 index，并且只有一个逗号，这是一种异常语法，但却合法。

%gs:foo：这个表达式引用的是放置于%gs 段里变量 foo 的值。

如果 call 和 jump 操作在操作数前指定前缀“*”，则表示是一个绝对地址调用/跳转，也就是说 jmp/call 指令指定的是一个绝对地址。如果没有指定"*"，则操作数是一个相对地址。

任何指令如果其操作数是一个内存操作，则指令必须指定它的操作尺寸(byte,word,long)，也就是说必须带有指令后缀(b,w,l)。

2. GCC Inline ASM

GCC 支持在 C/C++代码中嵌入汇编代码，这些汇编代码被称作 GCC Inline ASM——GCC 内联汇编。这是一个非常有用的功能，有利于我们将一些 C/C++语法无法表达的指令直接潜入 C/C++代码中，另外也允许我们直接写 C/C++代码中使用汇编编写简洁高效的代码。

2.1 Essential Inline ASM

GCC 中基本的内联汇编非常易懂，我们先来看两个简单的例子：

```
__asm__("movl %esp,%eax");  // 看起来很熟悉吧！
```

或者是

```
__asm__ ("
    movl $1,%eax  // SYS_exit
    xor %ebx,%ebx
    int $0x80
");
```

或

```
__asm__ (
    "movl $1,%eax\r\t"
    "xor %ebx,%ebx\r\t"
    "int $0x80"
);
```

基本内联汇编的格式是

```
__asm__ __volatile__("Instruction List");
```

1. __asm__

__asm__是 GCC 关键字 asm 的宏定义：

```
#define __asm__ asm
```

__asm__或 asm 用来声明一个内联汇编表达式，所以任何一个内联汇编表达式都是以它开头的，是必不可少的。

2. Instruction List

Instruction List 是汇编指令序列。它可以是空的，比如：__asm__ __volatile__(""); 或__asm__ ("");都是完全合法的内联汇编表达式，只不过这两条语句没有什么意义。但并非所有 Instruction List 为空的内联汇编表达式都是没有意义的，比如：__asm__ (""":::"memory"); 就非常有意义，它向 GCC 声明："我对内存作了改动"，GCC 在编译的时候，会将此因素考虑进去。

我们看一看下面这个例子：

```
$ cat example1.c
int main(int __argc, char* __argv[])
{
   int* __p = (int*)__argc;
   (*__p) = 9999;

   //__asm__(""":::"memory");

   if((*__p) == 9999)
      return 5;

   return (*__p);
}
```

在这段代码中，那条内联汇编是被注释掉的。在这条内联汇编之前，内存指针__p 所指向的内存被赋值为 9999，随即在内联汇编之后，一条 if 语句判断__p 所指向的内存与 9999 是否相等。很明显，它们是相等的。GCC 在优化编译的时候能够很聪明的发现这一点。我们使用下面的命令行对其进行编译：

```
$ gcc -O -S example1.c
```

选项-O 表示优化编译，我们还可以指定优化等级，比如-O2 表示优化等级为 2；选项-S 表示将 C/C++源文件编译为汇编文件，文件名和 C/C++文件一样，只不过扩展名由.c 变

为.s。

我们来查看一下被放在 example1.s 中的编译结果，我们这里仅仅列出了使用 gcc 2.96 在 redhat 7.3 上编译后的相关函数部分汇编代码。为了保持清晰性，无关的其它代码未被列出。

```
$ cat example1.s
main:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %eax   # int* __p = (int*)__argc
    movl    $9999, (%eax)   # (*__p) = 9999
    movl    $5, %eax        # return 5
    popl    %ebp
    ret
```

参照一下 C 源码和编译出的汇编代码，我们会发现汇编代码中，没有 if 语句相关的代码，而是在赋值语句(*__p)=9999 后直接 return 5；这是因为 GCC 认为在(*__p)被赋值之后，在 if 语句之前没有任何改变(*__p)内容的操作，所以那条 if 语句的判断条件(*__p) == 9999 肯定是为 true 的，所以 GCC 就不再生成相关代码，而是直接根据为 true 的条件生成 return 5 的汇编代码（GCC 使用 eax 作为保存返回值的寄存器）。

我们现在将 example1.c 中内联汇编的注释去掉，重新编译，然后看一下相关的编译结果。

```
$ gcc -O -S example1.c
$ cat example1.s
main:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %eax   # int* __p =(int*)__argc
    movl    $9999, (%eax)   # (*__p) = 9999
#APP
    #__asm__("":::"memory")
#NO_APP
    cmpl    $9999, (%eax)   # (*__p) == 9999
    jne     .L3             # false
    movl    $5, %eax        # true, return 5
    jmp     .L2
.L3:
    movl    (%eax), %eax
.L2:
    popl    %ebp
    ret
```

由于内联汇编语句__asm__(""::"memory")向 GCC 声明，在此内联汇编语句出现的位置内存内容可能了改变，所以 GCC 在编译时就不能像刚才那样处理。这次，GCC 老老实实的将 if 语句生成了汇编代码。

可能有人会质疑：为什么要使用__asm__(""::"memory")向 GCC 声明内存发生了变化？明明“Instruction List”是空的，没有任何对内存的操作，这样做只会增加 GCC 生成汇编代码的数量。

确实，那条内联汇编语句没有对内存作任何操作，事实上它确实什么都没有做。但影响内存内容的不仅仅是你当前正在运行的程序。比如，如果你现在正在操作的内存是一块内存映射，映射的内容是外围 I/O 设备寄存器。那么操作这块内存的就不仅仅是当前的程序，I/O 设备也会去操作这块内存。既然两者都会去操作同一块内存，那么任何一方在任何时候都不能对这块内存的内容想当然。所以当你使用高级语言 C/C++写这类程序的时候，你必须让编译器也能够明白这一点，毕竟高级语言最终要被编译为汇编代码。

你可能已经注意到了，这次输出的汇编结果中，有两个符号：#APP 和#NO_APP，GCC 将内联汇编语句中"Instruction List"所列出的指令放在#APP 和#NO_APP 之间，由于__asm__(""::"memory")中“Instruction List”为空，所以#APP 和#NO_APP 中间也没有任何内容。但我们以后的例子会更加清楚的表现这一点。

关于为什么内联汇编__asm__(""::"memory")是一条声明内存改变的语句，我们后面会详细讨论。

刚才我们花了大量的内容来讨论"Instruction List"为空是的情况，但在实际的编程中，"Instruction List"绝大多数情况下都不是空的。它可以有 1 条或任意多条汇编指令。

当在"Instruction List"中有多条指令的时候，你可以在一对引号中列出全部指令，也可以将一条或几条指令放在一对引号中，所有指令放在多对引号中。如果是前者，你可以将每一条指令放在一行，如果要将多条指令放在一行，则必须用分号（;）或换行符（\n，大多数情况下\n 后还要跟一个\t，其中\n 是为了换行，\t 是为了空出一个 tab 宽度的空格）将它们分开。下面的例子都是合法的写法。

```
__asm__("movl %eax, %ebx
    sti
    popl %edi
    subl %ecx, %ebx");

__asm__("movl %eax, %ebx; sti
    popl %edi; subl %ecx, %ebx");

__asm__("movl %eax, %ebx; sti\n\t popl %edi
    subl %ecx, %ebx");
```

如果你将指令放在多对引号中，则除了最后一对引号之外，前面的所有引号里的最后一条指令之后都要有一个分号(；)或(\n)或(\n\t)。比如：

```
__asm__("movl %eax, %ebx
        sti\n"
        "popl %edi;"
        "subl %ecx, %ebx");

__asm__("movl %eax, %ebx; sti\n\t"
        "popl %edi; subl %ecx, %ebx");

__asm__("movl %eax, %ebx; sti\n\t popl %edi\n"
        "subl %ecx, %ebx");

__asm__("movl %eax, %ebx; sti\n\t popl %edi;"
        "subl %ecx, %ebx");
```

上述原则可以归结为：

- 任意两个指令间要么被分号(；)分开，要么被放在两行；
- 放在两行的方法既可以从通过\n 的方法来实现，也可以真正的放在两行；
- 可以使用 1 对或多对引号，每 1 对引号里可以放任一多条指令，所有的指令都要被放到引号中。

在基本内联汇编中，“Instruction List”的书写的格式和你直接在汇编文件中写非内联汇编没有什么不同，你可以在其中定义 Label，定义对齐(.align n)，定义段(.section name)。例如：

```
__asm__(".align 2\n\t"
        "movl %eax, %ebx\n\t"
        "test %ebx, %ecx\n\t"
        "jne error\n\t"
        "sti\n\t"
        "error: popl %edi\n\t"
        "subl %ecx, %ebx");
```

上面例子的格式是 Linux 内联代码常用的格式，非常整齐。也建议大家都使用这种格式来写内联汇编代码。

3. __volatile__

__volatile__是 GCC 关键字 volatile 的宏定义：

```
#define __volatile__ volatile
```

__volatile__或 volatile 是可选的，你可以用它也可以不用它。如果你用了它，则是向 GCC 声明"不要动我所写的 Instruction List，我需要原封不动的保留每一条指令"，否则当你使用了优化选项(-O)进行编译时，GCC 将会根据自己的判断决定是否将这个内联汇编表达式中的指令优化掉。

那么 GCC 判断的原则是什么？我不知道（如果有哪位朋友清楚的话，请告诉我）。我试验了一下，发现一条内联汇编语句如果是基本内联汇编的话（即只有"Instruction List"，没有 Input/Output/Clobber 的内联汇编，我们后面将会讨论这一点），无论你是否使用__volatile__来修饰，GCC 2.96 在优化编译时，都会原封不动的保留内联汇编中的"Instruction List"。但或许我的试验的例子并不充分，所以这一点并不能够得到保证。

为了保险起见，如果你不想让 GCC 的优化影响你的内联汇编代码，你最好在前面都加上__volatile__，而不要依赖于编译器的原则，因为即使你非常了解当前编译器的优化原则，你也无法保证这种原则将来不会发生变化。而__volatile__的含义却是恒定的。

2.2 Inline ASM with C/C++ Expression

GCC 允许你通过 C/C++表达式指定内联汇编中"Instrcuction List"中指令的输入和输出，你甚至可以不关心到底使用哪个寄存器被使用，完全靠 GCC 来安排和指定。这一点可以让程序员避免去考虑有限的寄存器的使用，也可以提高目标代码的效率。先来看几个例子：

```
__asm__ (" " : : : "memory" );  // 前面提到的

__asm__ ("mov %%eax, %%ebx"
       : "=b"(rv)
       : "a"(foo)
       : "eax", "ebx");

__asm__ __volatile__("lidt %0"
       : "=m" (idt_descr));

__asm__("subl %2,%0\n\t"
       "sbbl %3,%1"
       : "=a" (endlow), "=d" (endhigh)
       : "g" (startlow), "g" (starthigh),
         "0" (endlow), "1" (endhigh));
```

怎么样，有点印象了吧，是不是也有点晕？没关系，下面讨论完之后你就不会再晕了。（当然，也有可能更晕☺）。讨论开始——

带有 C/C++表达式的内联汇编格式为：

```
__asm__ __volatile__("Instruction List"
                : Output
                : Input
                : Clobber/Modify);
```

从中我们可以看出它和基本内联汇编的不同之处在于：它多了 3 个部分(Input，Output，Clobber/Modify)。在括号中的 4 个部分通过冒号(:)分开。

这 4 个部分都不是必须的，任何一个部分都可以为空，其规则为：

- 如果 Clobber/Modify 为空，则其前面的冒号(:)必须省略。比如__asm__("mov %%eax, %%ebx" : "=b"(foo) : "a"(inp) :)就是非法的写法；而__asm__("mov %%eax, %%ebx" : "=b"(foo) : "a"(inp))则是正确的。
- 如果 Instruction List 为空，则 Input，Output，Clobber/Modify 可以不为空，也可以为空。比如__asm__ (" " : : : "memory");和__asm__(" " : :);都是合法的写法。
- 如果 Output，Input，Clobber/Modify 都为空，Output，Input 之前的冒号(:)既可以省略，也可以不省略。如果都省略，则此汇编退化为一个基本内联汇编，否则，仍然是一个带有 C/C++表达式的内联汇编，此时"Instruction List"中的寄存器写法要遵守相关规定，比如寄存器前必须使用两个百分号(%%)，而不是像基本汇编格式一样在寄存器前只使用一个百分号(%)。比如__asm__(" mov %%eax, %%ebx" : :)；__asm__(" mov %%eax, %%ebx" :)和__asm__(" mov %eax, %ebx")都是正确的写法，而__asm__(" mov %eax, %ebx" : :)；__asm__(" mov %eax, %ebx" :)和__asm__(" mov %%eax, %%ebx")都是错误的写法。
- 如果 Input，Clobber/Modify 为空，但 Output 不为空，Input 前的冒号(:)既可以省略，也可以不省略。比如__asm__(" mov %%eax, %%ebx" : "=b"(foo) :)；__asm__(" mov %%eax, %%ebx" : "=b"(foo))都是正确的。
- 如果后面的部分不为空，而前面的部分为空，则前面的冒号(:)都必须保留，否则无法说明不为空的部分究竟是第几部分。比如，Clobber/Modify，Output 为空，而 Input 不为空，则 Clobber/Modify 前的冒号必须省略（前面的规则），而 Output 前的冒号必须为保留。如果 Clobber/Modify 不为空，而 Input 和 Output 都为空，则 Input 和 Output 前的冒号都必须保留。比如__asm__(" mov %%eax, %%ebx" : : "a"(foo))和__asm__(" mov %%eax, %%ebx" : : : "ebx")。

从上面的规则可以看到另外一个事实，区分一个内联汇编是基本格式的还是带有 C/C++表达式格式的，其规则在于在"Instruction List"后是否有冒号(:)的存在，如果没有则是基本格式的，否则，则是带有 C/C++表达式格式的。

两种格式对寄存器语法的要求不同：基本格式要求寄存器前只能使用一个百分号(%)，这一点和非内联汇编相同；而带有 C/C++表达式格式则要求寄存器前必须使用两个百分号(%%)，其原因我们会在后面讨论。

1. Output

Output 用来指定当前内联汇编语句的输出。我们看一看这个例子：

```
__asm__ ("movl %%cr0, %0": "=a" (cr0));
```

这个内联汇编语句的输出部分为"=r"(cr0)，它是一个“操作表达式”，指定了一个输出操作。我们可以很清楚得看到这个输出操作由两部分组成：括号里的部分(cr0)和引号引住的部分"=a"。这两部分都是每一个输出操作必不可少的。括号里的部分是一个 C/C++表达式，用来保存内联汇编的一个输出值，其操作就等于 C/C++的相等赋值 cr0 = output_value，因此，括号中的输出表达式只能是 C/C++的左值表达式，也就是说它只能是一个可以合法的放在 C/C++赋值操作中等号(=)左边的表达式。那么右值 output_value 从何而来呢？

答案是引号中的内容，被称作“操作约束”（Operation Constraint），在这个例子中操作约束为"=a"，它包含两个约束：等号(=)和字母 a，其中等号(=)说明括号中左值表达式 cr0 是一个 Write-Only 的，只能够被作为当前内联汇编的输入，而不能作为输入。而字母 a 是寄存器 EAX / AX / AL 的简写，说明 cr0 的值要从 eax 寄存器中获取，也就是说 cr0 = eax，最终这一点被转化成汇编指令就是 movl %eax, address_of_cr0。现在你应该清楚了吧，操作约束中会给出：到底从哪个寄存器传递值给 cr0。

另外，需要特别说明的是，很多文档都声明，所有输出操作的操作约束必须包含一个等号(=)，但 GCC 的文档中却很清楚的声明，并非如此。因为等号(=)约束说明当前的表达式是一个 Write-Only 的，但另外还有一个符号——加号(+)用来说明当前表达式是一个 Read-Write 的，如果一个操作约束中没有给出这两个符号中的任何一个，则说明当前表达式是 Read-Only 的。因为对于输出操作来说，肯定是必须是可写的，而等号(=)和加号(+)都表示可写，只不过加号(+)同时也表示是可读的。所以对于一个输出操作来说，其操作约束只需要有等号(=)或加号(+)中的任意一个就可以了。

二者的区别是：等号(=)表示当前操作表达式指定了一个纯粹的输出操作，而加号(+)则表示当前操作表达式不仅仅只是一个输出操作还是一个输入操作。但无论是等号(=)约束还是加号(+)约束所约束的操作表达式都只能放在 Output 域中，而不能被用在 Input 域中。

另外，有些文档声明：尽管 GCC 文档中提供了加号(+)约束，但在实际的编译中通不过；我不知道老版本会怎么样，我在 GCC 2.96 中对加号(+)约束的使用非常正常。

我们通过一个例子看一下，在一个输出操作中使用等号(=)约束和加号(+)约束的不同。

```
$ cat example2.c

int main(int __argc, char* __argv[])
{
    int cr0 = 5;
    __asm__ __volatile__("movl %%cr0, %0"
                 :"=a" (cr0));

    return 0;
}

$ gcc -S example2.c

$ cat example2.s
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $4, %esp
    movl    $5, -4(%ebp)     # cr0 = 5
#APP
    movl %cr0, %eax
#NO_APP
    movl    %eax, %eax
    movl    %eax, -4(%ebp)   # cr0 = %eax
    movl    $0, %eax
    leave
    ret
```

这个例子是使用等号(=)约束的情况，变量 cr0 **被放在内存**-4(%ebp)**的位置，所以指令** mov %eax, -4(%ebp)**即表示将**%eax **的内容输出到变量** cr0 **中。**

下面是使用加号(+)约束的情况：

```
    $ cat example3.c
    int main(int __argc, char* __argv[])
    {
        int cr0 = 5;
        __asm__ __volatile__("movl %%cr0, %0"
                : "+a" (cr0));
        return 0;
    }
```

```
$ gcc -S example3.c
$ cat example3.s
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $4, %esp
    movl    $5, -4(%ebp)        # cr0 = 5
    movl    -4(%ebp), %eax      # input ( %eax = cr0 )
#APP
    movl    %cr0, %eax
#NO_APP
    movl    %eax, -4(%ebp)      # output (cr0 = %eax )
    movl    $0, %eax
    leave
    ret
```

从编译的结果可以看出，当使用加号(+)约束的时候，cr0 **不仅作为输出，还作为输入，所使用寄存器都是寄存器约束(字母** a**，表示使用** eax **寄存器)指定的。关于寄存器约束我们后面讨论。**

在 Output **域中可以有多个输出操作表达式，多个操作表达式中间必须用逗号(,)分开。例如：**

```
$ cat example3.c
 __asm__(
    "movl   %%eax, %0 \n\t"
    "pushl  %%ebx \n\t"
    "popl    %1 \n\t"
    "movl   %1, %2"
    : "+a"(cr0), "=b"(cr1), "=c"(cr2));
```

2. Input

Input **域的内容用来指定当前内联汇编语句的输入。我们看一看这个例子：**

```
__asm__("movl %0, %%db7" : : "a" (cpu->db7));
```

例中 Input **域的内容为一个表达式**"a"[cpu->db7)**，被称作“输入表达式”，用来表示一个对当前内联汇编的输入。**

像输出表达式一样，一个输入表达式也分为两部分：带括号的部分(cpu->db7)**和带引号的部分**"a"**。这两部分对于一个内联汇编输入表达式来说也是必不可少的。**

括号中的表达式 cpu->db7 是一个 C/C++语言的表达式，它不必是一个左值表达式，也就是说它不仅可以是放在 C/C++赋值操作左边的表达式，还可以是放在 C/C++赋值操作右边的表达式。所以它可以是一个变量，一个数字，还可以是一个复杂的表达式（比如 a+b/c*d）。比如上例可以改为：

```
__asm__("movl %0, %%db7" : : "a" (foo));
__asm__("movl %0, %%db7" : : "a" (0x1000));
__asm__("movl %0, %%db7" : : "a" (va*vb/vc));
```

引号号中的部分是约束部分，和输出表达式约束不同的是，它不允许指定加号(+)约束和等号(=)约束，也就是说它只能是默认的 Read-Only 的。约束中必须指定一个寄存器约束，例中的字母 a 表示当前输入变量 cpu->db7 要通过寄存器 eax 输入到当前内联汇编中。

我们看一个例子：

```
$ cat example4.c
int main(int __argc, char* __argv[])
{
   int cr0  = 5;
   __asm__ __volatile__("movl %0, %%cr0"::"a" (cr0));
   return 0;
}
$ gcc -S example4.c
$ cat example4.s
main:
   pushl   %ebp
   movl    %esp, %ebp
   subl    $4, %esp
   movl    $5, -4(%ebp)          # cr0 = 5
   movl    -4(%ebp), %eax     # %eax = cr0
#APP
   movl    %eax, %cr0
#NO_APP
   movl    $0, %eax
   leave
   ret
```

我们从编译出的汇编代码可以看到，在"Instruction List"之前，GCC 按照我们的输入约束"a"，将变量 cr0 的内容装入了 eax 寄存器。

3. Operation Constraint

每一个 Input 和 Output 表达式都必须指定自己的操作约束 Operation Constraint，我们这里来讨论在 80386 平台上所可能使用的操作约束。

3.1 Register Constraint

当你当前的输入或输入需要借助一个寄存器时，你需要为其指定一个寄存器约束。你可以直接指定一个寄存器的名字，比如：

```
__asm__ __volatile__("movl %0, %%cr0"::"eax" (cr0));
```

也可以指定一个缩写，比如：

```
__asm__ __volatile__("movl %0, %%cr0"::"a" (cr0));
```

如果你指定一个缩写，比如字母 a**，则** GCC **将会根据当前操作表达式中** C/C++**表达式的宽度决定使用**%eax**，还是**%ax **或**%al**。比如：**

```
unsigned short __shrt;
__asm__ ("mov %0，%%bx" : : "a"(__shrt));
```

由于变量__shrt **是** 16-bit short **类型，则编译出来的汇编代码中，会让变量**__shrt **使用** %ex **寄存器。编译结果为：**

```
    Movw   -2(%ebp), %ax  # %ax = __shrt
#APP
    movl   %ax, %bx
#NO_APP
```

无论是 Input**，还是** Output **操作表达式约束，都可以使用寄存器约束。**

下表中列出了常用的寄存器约束的缩写。

约束	**意义**
r	**表示使用一个通用寄存器，由** GCC **在**%eax/%ax/%al, %ebx/%bx/%bl, %ecx/%cx/%cl, %edx/%dx/%dl **中选取一个** GCC **认为合适的。**
g	**表示使用任意一个寄存器，由** GCC **在所有的可以使用的寄存器中选取一个** GCC **认为合适的。**
q	**表示使用一个通用寄存器，和约束** r **的意义相同。**
a	**表示使用**%eax/%ax/%al
b	**表示使用**%ebx/%bx /%bl
c	**表示使用**%ecx/%cx/%cl
d	**表示使用**%edx/%dx /%dl
D	**表示使用**%edi/%di
S	**表示使用**%esi/%si
f	**表示使用浮点寄存器**
t	**表示使用第一个浮点寄存器**
u	**表示使用第二个浮点寄存器**

3.2 Memory Constraint

如果一个 Input/Output 操作表达式的 C/C++表达式表现为一个内存地址，不想借助于任何寄存器，则可以使用内存约束。比如：

```
__asm__ ("lidt %0" : "=m"(__idt_addr));
__asm__ ("lidt %0" : :"m"(__idt_addr));
```

我们看一下它们分别被放在一个 C 源文件中，然后被 GCC 编译后的结果：

```
$ cat example5.c
/* 本例中，变量__sh 被作为一个内存输入 */
int main(int __argc, char* __argv[])
{
   char* __sh = (char*)&__argc;

   __asm__ __volatile__(
      "lidt %0"
      : /* no output */
      : "m" (__sh)
      );

   return 0;
}

$ gcc -S example5.c

$ cat example5.s
main:
   pushl   %ebp
   movl    %esp, %ebp
   subl    $4, %esp
   leal     8(%ebp), %eax
   movl   %eax, -4(%ebp)  # sh = (char*) &__argc
#APP
   lidt      -4(%ebp)
#NO_APP
   movl    $0, %eax
   leave
   ret
```

```
$ cat example6.c
/* 本例中，变量__sh 被作为一个内存输出 */
int main(int __argc, char* __argv[])
{
    char* __sh = (char*)&__argc;

    __asm__ __volatile__(
        "lidt %0"
        : "=m" (__sh)
        );

    return 0;
}

$ gcc -S example6.c

$ cat example6.s
main:
    pushl  %ebp
    movl   %esp, %ebp
    subl   $4, %esp
    leal   8(%ebp), %eax
    movl   %eax, -4(%ebp) # sh = (char*) &__argc
#APP
    lidt   -4(%ebp)
#NO_APP
    movl   $0, %eax
    leave
    ret
```

首先，你会注意到，在这两个例子中，变量 sh **没有借助任何寄存器，而是直接参与了指令** lidt **的操作。**

其次，通过仔细观察，你会发现一个惊人的事实，两个例子编译出来的汇编代码是一样的!虽然，一个例子中变量 sh **作为输入，而另一个例子中变量** sh **作为输出。这是怎么回事？**

原来，使用内存方式进行输入输出时，由于不借助寄存器，所以 GCC **不会按照你的声明对其作任何的输入输出处理。**GCC **只会直接拿来用，究竟对这个** C/C++**表达式而言是输入还是输出，完全依赖与你写在**"Instruction List"**中的指令对其操作的指令。**

由于上例中，对其操作的指令为 lidt**，**lidt **指令的操作数是一个输入型的操作数，所以事实上对变量** sh **的操作是一个输入操作，即使你把它放在** Output **域也不会改变这一点。所以，对此例而言，完全符合语意的写法应该是将** sh **放在** Input **域，尽管放在** Output **域也会有正确的执行结果。**

所以，对于内存约束类型的操作表达式而言，放在 Input 域还是放在 Output 域，对编译结果是没有任何影响的，因为本来我们将一个操作表达式放在 Input 域或放在 Output 域是希望 GCC 能为我们自动通过寄存器将表达式的值输入或输出。既然对于内存约束类型的操作表达式来说，GCC 不会自动为它做任何事情，那么放在哪儿也就无所谓了。但从程序员的角度而言，为了增强代码的可读性，最好能够把它放在符合实际情况的地方。

约束	意义
M	表示使用系统所支持的任何一种内存方式，不需要借助寄存器。

3.3 Immediately Number Constraint

如果一个 Input/Output 操作表达式的 C/C++表达式是一个数字常数，不想借助于任何寄存器，则可以使用立即数约束。

由于立即数在 C/C++中只能作为右值，所以对于使用立即数约束的表达式而言，只能放在 Input 域。比如：

```
__asm__ __volatile__("movl %0, %%eax" : : "i" (100) );
```

立即数约束很简单，也很容易理解，我们在这里就不再赘述。

约束	意义
i	表示输入表达式是一个立即数(整数)，不需要借助任何寄存器。
F	表示输入表达式是一个立即数(浮点数)，不需要借助任何寄存器。

3.4 Generic Constraint

约束	输入/输出	意义
g	I,O	表示可以使用通用寄存器，内存，立即数等任何一种处理方式。
0-9	I	表示和第 n 个操作表达式使用相同的寄存器/内存。

通用约束 g 是一个非常灵活的约束，当程序员认为一个 C/C++表达式在实际的操作中，究竟使用寄存器方式，还是使用内存方式或立即数方式并无所谓时，或者程序员想实现一个灵活的模板，让 GCC 可以根据不同的 C/C++表达式生成不同的访问方式时，就可以使用通用约束 g。比如：

```
#define JUST_MOV(foo)  \
__asm__ ("movl %0, %%eax" : : "g"(foo))
```

JUST_MOV(100)和 JUST_MOV(var)则会让编译器产生不同的代码。

```
int main(int __argc, char* __argv[])
{
    JUST_MOV(100);
    return 0;
}
```

编译后生成的代码为：

```
main:
    pushl   %ebp
    movl    %esp, %ebp
#APP
    movl $100, %eax
#NO_APP
    movl    $0, %eax
    popl    %ebp
    ret
```

很明显这是立即数方式。而下一个例子：

```
int main(int __argc, char* __argv[])
{
    JUST_MOV(__argc);
    return 0;
}
```

经编译后生成的代码为：

```
main:
    pushl  %ebp
    movl   %esp, %ebp
#APP
    movl   8(%ebp), %eax
#NO_APP
    movl   $0, %eax
    popl   %ebp
    ret
```

这个例子是使用内存方式。

一个带有 C/C++**表达式的内联汇编，其操作表达式被按照被列出的顺序编号，第一个是** 0**，第** 2 **个是** 1**，依次类推，**GCC **最多允许有** 10 **个操作表达式。比如：**

```
__asm__ ("popl %0 \n\t"
    "movl %1, %%esi \n\t"
    "movl %2, %%edi \n\t"
    : "=a"(__out)
    : "r" (__in1), "r" (__in2));
```

此例中，__out 所在的 Output 操作表达式被编号为 0，"r"(__in1)被编号为 1，"r"(__in2)被编号为 2。

再如：

```
__asm__ ("movl %%eax, %%ebx" : : "a"(__in1), "b"(__in2));
```

此例中，"a"(__in1)被编号为 0，"b"(__in2)被编号为 1。

如果某个 Input 操作表达式使用数字 0 到 9 中的一个数字（假设为 1）作为它的操作约束，则等于向 GCC 声明："我要使用和编号为 1 的 Output 操作表达式相同的寄存器（如果 Output 操作表达式 1 使用的是寄存器），或相同的内存地址（如果 Output 操作表达式 1 使用的是内存）"。上面的描述包含两个限定：数字 0 到数字 9 作为操作约束只能用在 Input 操作表达式中，被指定的操作表达式（比如某个 Input 操作表达式使用数字 1 作为约束，那么被指定的就是编号为 1 的操作表达式）只能是 Output 操作表达式。

由于 GCC 规定最多只能有 10 个 Input/Output 操作表达式，所以事实上数字 9 作为操作约束永远也用不到，因为 Output 操作表达式排在 Input 操作表达式的前面，那么如果有一个 Input 操作表达式指定了数字 9 作为操作约束的话，那么说明 Output 操作表达式的数量已经至少为 10 个了，那么再加上这个 Input 操作表达式，则至少为 11 个了，以及超出 GCC 的限制。

5、Modifier Characters（修饰符）

等号(=)和加号(+)用于对 Output 操作表达式的修饰，一个 Output 操作表达式要么被等号(=)修饰，要么被加号(+)修饰，二者必居其一。使用等号(=)说明此 Output 操作表达式是 Write-Only 的，使用加号(+)说明此 Output 操作表达式是 Read-Write 的。它们必须被放在约束字符串的第一个字母。比如"a="(foo)是非法的，而"+g"(foo)则是合法的。

当使用加号(+)的时候，此 Output 表达式等价于使用等号(=)约束加上一个 Input 表达式。比如

```
__asm__ ("movl %0, %%eax; addl %%eax, %0" : "+b"(foo))
```

等价于

```
__asm__ ("movl %0, %%eax; addl %%eax, %0" : "+b"(foo))
```

但如果使用后一种写法，"Instruction List"中的别名也要相应的改动。关于别名，我们

后面会讨论。

像等号(=)和加号(+)修饰符一样，符号(&)也只能用于对 Output 操作表达式的修饰。当使用它进行修饰时,等于向 GCC 声明："GCC 不得为任何 Input 操作表达式分配与此 Output 操作表达式相同的寄存器"。其原因是&修饰符意味着被其修饰的 Output 操作表达式要在所有的 Input 操作表达式被输入前输出。我们看下面这个例子：

```
int main(int __argc, char* __argv[])
{
   int __in1 = 8, __in2 = 4, __out = 3;

   __asm__ ("popl %0 \n\t"
      "movl %1, %%esi \n\t"
      "movl %2, %%edi \n\t"
      : "=a"(__out)
      : "r" (__in1), "r" (__in2));

   return 0;
}
```

此例中，%0 对应的就是 Output 操作表达式，它被指定的寄存器是%eax，整个 Instruction List 的第一条指令 popl %0，编译后就成为 popl %eax，这时%eax 的内容已经被修改，随后在 Instruction List 后，GCC 会通过 movl %eax, address_of_out 这条指令将 %eax 的内容放置到 Output 变量__out 中。对于本例中的两个 Input 操作表达式而言，它们的寄存器约束为"r"，即要求 GCC 为其指定合适的寄存器，然后在 Instruction List 之前将 __in1 和__in2 的内容放入被选出的寄存器中，如果它们中的一个选择了已经被__out 指定的寄存器%eax，假如是__in1，那么 GCC 在 Instruction List 之前会插入指令 movl address_of_in1, %eax，那么随后 popl %eax 指令就修改了%eax 的值，此时%eax 中存放的已经不是 Input 变量__in1 的值了，那么随后的 movl %1, %%esi 指令，将不会按照我们的本意——即将__in1 的值放入%esi 中——而是将__out 的值放入%esi 中了。

下面就是本例的编译结果，很明显，GCC 为__in2 选择了和__out 相同的寄存器%eax，这与我们的初衷不符。

```
main:
   pushl   %ebp
   movl    %esp, %ebp
   subl    $12, %esp
   movl    $8, -4(%ebp)
   movl    $4, -8(%ebp)
   movl    $3, -12(%ebp)
   movl    -4(%ebp), %edx      # __in1 使用寄存器%edx
   movl    -8(%ebp), %eax      # __in2 使用寄存器%eax
```

```
#APP
    popl   %eax
    movl   %edx, %esi
    movl   %eax, %edi
#NO_APP
    movl   %eax, %eax
    movl   %eax, -12(%ebp)  # __out 使用寄存器%eax
    movl   $0, %eax
    leave
```

为了避免这种情况，我们必须向 GCC 声明这一点，要求 GCC 为所有的 Input 操作表达式指定别的寄存器，方法就是在 Output 操作表达式"=a"(__out)的操作约束中加入&约束，由于 GCC 规定等号(=)约束必须放在第一个，所以我们写作"=&a"(__out)。

下面是我们将&约束加入之后编译的结果：

```
main:
    pushl  %ebp
    movl   %esp, %ebp
    subl   $12, %esp
    movl   $8, -4(%ebp)
    movl   $4, -8(%ebp)
    movl   $3, -12(%ebp)
    movl   -4(%ebp), %edx   #__in1 使用寄存器%edx
    movl   -8(%ebp), %eax
    movl   %eax, %ecx       # __in2 使用寄存器%ecx
#APP
    popl   %eax
    movl   %edx, %esi
    movl   %ecx, %edi
#NO_APP
    movl   %eax, %eax
    movl   %eax, -12(%ebp)  #__out 使用寄存器%eax
    movl   $0, %eax
    leave
    ret
```

OK！这下好了，完全与我们的意图吻合。

如果一个 Output 操作表达式的寄存器约束被指定为某个寄存器，只有当至少存在一个 Input 操作表达式的寄存器约束为可选约束时，（可选约束的意思是可以从多个寄存器中选取一个，或使用非寄存器方式），比如"r"或"g"时，此 Output 操作表达式使用&修饰才有意义。如果你为所有的 Input 操作表达式指定了固定的寄存器，或使用内存/立即数约束，则此 Output 操作表达式使用&修饰没有任何意义。比如：

```
__asm__ ("popl %0 \n\t"
    "movl %1, %%esi \n\t"
    "movl %2, %%edi \n\t"
    : "=&a"(__out)
    : "m" (__in1), "c" (__in2));
```

此例中的 Output **操作表达式完全没有必要使用**&**来修饰，因为**__in1 **和**__in2 **都被指定了固定的寄存器，或使用了内存方式，**GCC **无从选择。**

但如果你已经为某个 Output **操作表达式指定了**&**修饰，并指定了某个固定的寄存器，你就不能再为任何** Input **操作表达式指定这个寄存器，否则会出现编译错误。比如：**

```
__asm__ ("popl %0 \n\t"
    "movl %1, %%esi \n\t"
    "movl %2, %%edi \n\t"
    : "=&a"(__out)
    : "a" (__in1), "c" (__in2));
```

本例中，由于__out **已经指定了寄存器**%eax**，同时使用了符号**&**修饰，则再为**__in1 **指定寄存器**%eax **就是非法的。**

反过来，你也可以为 Output **指定可选约束，比如**"r","g"**等，让** GCC **为其选择到底使用哪个寄存器，还是使用内存方式，**GCC **在选择的时候，会首先排除掉已经被** Input **操作表达式使用的所有寄存器，然后在剩下的寄存器中选择，或干脆使用内存方式。比如：**

```
__asm__ ("popl %0 \n\t"
    "movl %1, %%esi \n\t"
    "movl %2, %%edi \n\t"
    : "=&r"(__out)
    : "a" (__in1), "c" (__in2));
```

本例中，由于__out **指定了约束**"r"**，即让** GCC **为其决定使用哪一格寄存器，而寄存器**%eax **和**%ecx **已经被**__in1 **和**__in2 **使用，那么** GCC **在为**__out **选择的时候，只会在**%ebx **和**%edx **中选择。**

前 3 **个修饰符只能用在** Output **操作表达式中，而百分号**[%]**修饰符恰恰相反，只能用在** Input **操作表达式中，用于向** GCC **声明：“当前** Input **操作表达式中的** C/C++**表达式可以和下一个** Input **操作表达式中的** C/C++**表达式互换”。这个修饰符号一般用于符合交换律运算，比如加**(+)**，乘**(*)**，与**(&)**，或**(|)**等等。我们看一个例子：**

```
int main(int __argc, char* __argv[])
{
    int __in1 = 8, __in2 = 4, __out = 3;

    __asm__ ("addl %1, %0\n\t"
    : "=r"(__out)
    : "%r" (__in1), "0" (__in2));

    return 0;
}
```

在此例中，由于指令是一个加法运算，相当于等式__out = __in1 + __in2**，而它与等式**__out = __in2 + __in1 **没有什么不同。所以使用百分号修饰，让** GCC **知道**__in1 **和**__in2 **可以互换，也就是说** GCC **可以自动将本例的内联汇编改变为：**

```
__asm__ ("addl %1, %0\n\t"
    : "=r"(__out)
    : "%r" (__in2), "0" (__in1));
```

下表总结了各种修饰符的意义：

修饰符	**输入/输出**	**意义**
=	O	**表示此** Output **操作表达式是** Write-Only **的。**
+	O	**表示此** Output **操作表达式是** Read-Write **的。**
&	O	**表示此** Output **操作表达式独占为其指定的寄存器。**
%	I	**表示此** Input **操作表达式中的** C/C++**表达式可以和下一个** Input **操作表达式中的** C/C++**表达式互换。**

4. 占位符

什么叫占位符？我们看一看下面这个例子：

```
__asm__ ("addl %1, %0\n\t"
    : "=a"(__out)
    : "m" (__in1), "a" (__in2));
```

这个例子中的%0 **和**%1 **就是占位符。每一个占位符对应一个** Input/Output **操作表达式。我们在之前已经提到，**GCC **规定一个内联汇编语句最多可以有** 10 **个** Input/Output **操作表达式，然后按照它们被列出的顺序依次赋予编号** 0 **到** 9**。对于占位符中的数字而言，和这些编号是对应的。**

由于占位符前面使用一个百分号(%)**，为了区别占位符和寄存器，**GCC **规定在带有**

C/C++表达式的内联汇编中，"Instruction List"中直接写出的寄存器前必须使用两个百分号(%%)。

GCC 对其进行编译的时候，会将每一个占位符替换为对应的 Input/Output 操作表达式所指定的寄存器/内存地址/立即数。比如在上例中，占位符%0 对应 Output 操作表达式"=a"(__out)，而"=a"(__out)指定的寄存器为%eax，所以把占位符%0 替换为%eax，占位符%1 对应 Input 操作表达式"m"(__in1)，而"m"(__in1)被指定为内存操作，所以把占位符%1 替换为变量__in1 的内存地址。

也许有人认为，在上面这个例子中，完全可以不使用%0，而是直接写%%eax，就像这样：

```
__asm__ ("addl %1, %%eax\n\t"
    : "=a"(__out)
    : "m" (__in1), "a" (__in2));
```

和上面使用占位符%0 没有什么不同，那么使用占位符%0 就没有什么意义。确实，两者生成的代码完全相同，但这并不意味着这种情况下占位符没有意义。因为如果不使用占位符，那么当有一天你想把变量__out 的寄存器约束由 a 改为 b 时，那么你也必须将 addl 指令中的%%eax 改为%%ebx，也就是说你需要同时修改两个地方，而如果你使用占位符，你只需要修改一次就够了。另外，如果你不使用占位符，将不利于代码的清晰性。在上例中，如果你使用占位符，那么你一眼就可以得知，addl 指令的第二个操作数内容最终会输出到变量__out 中；否则，如果你不用占位符，而是直接将 addl 指令的第 2 个操作数写为%%eax，那么你需要考虑一下才知道它最终需要输出到变量__out 中。这是占位符最粗浅的意义。毕竟在这种情况下，你完全可以不用。

但对于这些情况来说，不用占位符就完全不行了：

首先，我们看一看上例中的第 1 个 Input 操作表达式"m"(__in1)，它被 GCC 替换之后，表现为 addl address_of_in1, %%eax，__in1 的地址是什么？编译时才知道。所以我们完全无法直接在指令中去写出__in1 的地址，这时使用占位符，交给 GCC 在编译时进行替代，就可以解决这个问题。所以这种情况下，我们必须使用占位符。

其次，如果上例中的 Output 操作表达式"=a"(__out)改为"=r"(__out)，那么__out 在究竟使用那么寄存器只有到编译时才能通过 GCC 来决定，既然在我们写代码的时候，我们不知道究竟哪个寄存器被选择，我们也就不能直接在指令中写出寄存器的名称，而只能通过占位符替代来解决。

5. Clobber/Modify

有时候，你想通知 GCC 当前内联汇编语句可能会对某些寄存器或内存进行修改，希望 GCC 在编译时能够将这一点考虑进去。那么你就可以在 Clobber/Modify 域声明这些寄存器或内存。

这种情况一般发生在一个寄存器出现在"Instruction List"，但却不是由 Input/Output 操作表达式所指定的，也不是在一些 Input/Output 操作表达式使用"r","g"约束时由 GCC 为其选择的，同时此寄存器被"Instruction List"中的指令修改，而这个寄存器只是供当前内联汇编临时使用的情况。比如：

```
__asm__ ("movl %0, %%ebx" : : "a"(__foo) : "bx");
```

寄存器%ebx 出现在"Instruction List 中"，并且被 movl 指令修改，但却未被任何 Input/Output 操作表达式指定，所以你需要在 Clobber/Modify 域指定"bx"，以让 GCC 知道这一点。

因为你在 Input/Output 操作表达式所指定的寄存器，或当你为一些 Input/Output 操作表达式使用"r","g"约束，让 GCC 为你选择一个寄存器时，GCC 对这些寄存器是非常清楚的——它知道这些寄存器是被修改的，你根本不需要在 Clobber/Modify 域再声明它们。但除此之外，GCC 对剩下的寄存器中哪些会被当前的内联汇编修改一无所知。所以如果你真的在当前内联汇编指令中修改了它们，那么就最好在 Clobber/Modify 中声明它们，让 GCC 针对这些寄存器做相应的处理。否则有可能会造成寄存器的不一致，从而造成程序执行错误。

在 Clobber/Modify 域中指定这些寄存器的方法很简单，你只需要将寄存器的名字使用双引号(" ")引起来。如果有多个寄存器需要声明，你需要在任意两个声明之间用逗号隔开。比如：

```
__asm__ ("movl %0, %%ebx; popl %%ecx"
    : /* no output */
    : "a"(__foo) : "bx", "cx" );
```

这些串包括：

声明的串	代表的寄存器
"al","ax","eax"	%eax
"bl","bx","ebx"	%ebx
"cl","cx","ecx"	%ecx
"dl","dx","edx"	%edx
"si","esi"	%esi
"di", "edi"	%edi

由上表可以看出，你只需要使用"ax","bx","cx","dx","si","di"就可以了，因为其它的都和它们中的一个是等价的。

如果你在一个内联汇编语句的 Clobber/Modify 域向 GCC 声明某个寄存器内容发生了改变，GCC 在编译时，如果发现这个被声明的寄存器的内容在此内联汇编语句之后还要继续使用，那么 GCC 会首先将此寄存器的内容保存起来，然后在此内联汇编语句的相关生成代码之后，再将其内容恢复。我们来看两个例子，然后对比一下它们之间的区别。

这个例子中声明了寄存器%ebx **内容发生了改变：**

```
$ cat example7.c
int main(int __argc, char* __argv[])
{
   int in = 8;
   __asm__ ("addl %0, %%ebx"
      : /* no output */
      : "a" (in) : "bx");

   return 0;
}
$ gcc -O -S example7.c
$ cat example7.s
main:
   pushl  %ebp
   movl   %esp, %ebp
   pushl  %ebx      # %ebx 内容被保存
   movl   $8, %eax
#APP
   addl   %eax, %ebx
#NO_APP
   movl   $0, %eax
   movl   (%esp), %ebx     # %ebx 内容被恢复
   leave
   ret
```

下面这个例子的 C **源码与上一个例子除了没有声明**%ebx **寄存器发生了改变之外，其它都相同。**

```
$ cat example8.c
int main(int __argc, char* __argv[])
{
   int in = 8;

   __asm__ ("addl %0, %%ebx"
      : /* no output */
      : "a" (in));

   return 0;
}
```

```
$ gcc -O -S example8.c
$ cat example8.s
main:
    pushl   %ebp
    movl    %esp, %ebp
    movl    $8, %eax
#APP
    addl %eax, %ebx
#NO_APP
    movl    $0, %eax
    popl    %ebp
    ret
```

仔细对比一下 example7.s **和** example8.s**，你就会明白在** Clobber/Modify **域声明一个寄存器的意义。**

另外需要注意的是，如果你在 Clobber/Modify **域声明了一个寄存器，那么这个寄存器将不能再被用做当前内联汇编语句的** Input/Output **操作表达式的寄存器约束，如果** Input/Output **操作表达式的寄存器约束被指定为**"r"**或**"g"**，**GCC **也不会选择已经被声明在** Clobber/Modify **中的寄存器。比如：**

```
__asm__ ("movl %0, %%ebx"
    : : "a"(__foo): "ax", "bx");
```

此例中，由于 Output **操作表达式**"a"(__foo)**的寄存器约束已经指定了**%eax **寄存器，那么再在** Clobber/Modify **域中指定**"ax"**就是非法的。编译时，**GCC **会给出编译错误。**

除了寄存器的内容会被改变，内存的内容也可以被修改。如果一个内联汇编语句"Instruction List"**中的指令对内存进行了修改，或者在此内联汇编出现的地方内存内容可能发生改变，而被改变的内存地址你没有在其** Output **操作表达式使用**"m"**约束，这种情况下你需要使用在** Clobber/Modify **域使用字符串**"memory"**向** GCC **声明：“在这里，内存发生了，或可能发生了改变”。例如：**

```
void* memset(void* __s, char __c, size_t __count)
{
    __asm__("cld\n\t"
        "rep\n\t"
        "stosb"
        : /* no output */
        : "a" (__c),"D" (__s),"c" (__count)
        : "cx","di","memory");
        return __s;
}
```

此例实现了标准函数库 memset，其内联汇编中的 stosb 对内存进行了改动，而其被修改的内存地址 s 被指定装入%edi，没有任何 Output 操作表达式使用了"m"约束，以指定内存地址 s 处的内容发生了改变。所以在其 Clobber/Modify 域使用"memory"向 GCC 声明：内存内容发生了变动。

如果一个内联汇编语句的 Clobber/Modify 域存在"memory"，那么 GCC 会保证在此内联汇编之前，如果某个内存的内容被装入了寄存器，那么在这个内联汇编之后，如果需要使用这个内存处的内容，就会直接到这个内存处重新读取，而不是使用被存放在寄存器中的拷贝。因为这个时候寄存器中的拷贝已经很可能和内存处的内容不一致了。

这只是使用"memory"时，GCC 会保证做到的一点，但这并不是全部。因为使用"memory"是向 GCC 声明内存发生了变化，而内存发生变化带来的影响并不止这一点。比如我们在前面讲到的例子：

```
int main(int __argc, char* __argv[])
{
   int* __p = (int*)__argc;
   (*__p) = 9999;
   __asm__("":::"memory");
   if((*__p) == 9999)
      return 5;

   return (*__p);
}
```

本例中，如果没有那条内联汇编语句，那个 if 语句的判断条件就完全是一句废话。GCC 在优化时会意识到这一点，而直接只生成 return 5 的汇编代码，而不会再生成 if 语句的相关代码，而不会生成 return (*__p)的相关代码。但你加上了这条内联汇编语句，它除了声明内存变化之外，什么都没有做。但 GCC 此时就不能简单的认为它不需要判断都知道(*__p)一定与 9999 相等，它只有老老实实生成这条 if 语句的汇编代码，一起相关的两个 return 语句相关代码。

当一个内联汇编指令中包含影响 eflags 寄存器中的条件标志（也就是那些 Jxx 等跳转指令要参考的标志位，比如，进位标志，0 标志等），那么需要在 Clobber/Modify 域中使用"cc"来声明这一点。这些指令包括 adc, div，popfl，btr，bts 等等，另外，当包含 call 指令时，由于你不知道你所 call 的函数是否会修改条件标志，为了稳妥起见，最好也使用"cc"。

我很少在相关资料中看到有关"cc"的确切用法，只有一份文档提到了它，但还不是 i386 平台的，只是说"cc"是处理器平台相关的，并非所有的平台都支持它，但即使在不支持它的平台上，使用它也不会造成编译错误。我做了一些实验，但发现使用"cc"和不使用"cc"所生成的代码没有任何不同。但 Linux 2.4 的相关代码中用到了它。如果谁知道在 i386 平台上"cc"的细节，请和我联系。

另外，还可以在 Clobber/Modify 域指定数字 0 到 9，以声明第 n 个 Input/Output 操作表达式所使用的寄存器发生了变化，但正如我们在前面所提到的，如果你为某个 Input/Output 操作表达式指定了寄存器，或使用"g","r"等约束让 GCC 为其选择寄存器，GCC 已经知道哪个寄存器内容发生了变化，所以这么做没有什么意义；我也作了相关的试验，没有发现使用它会对 GCC 生成的汇编代码有任何影响，至少在 i386 平台上是这样。Linux 2.4 的所有 i386 平台相关内联汇编代码中都没有使用这一点，但 S390 平台相关代码中有用到，但由于我对 S390 汇编没有任何概念，所以，也不知道这么做的意义何在。