**寻址方式：**

在存储器中，操作数或指令字写入或读出的方式，有地址指定方式、相联存储方式和堆栈存取方式。几乎所有的计算机，在内存中都采用地址指定方式。当采用地址指定方式时，形成操作数或指令地址的方式称为寻址方式。寻址方式分为两类，即指令寻址方式和数据寻址方式，前者比较简单，后者比较复杂。值得注意的是，在传统方式设计的计算机中，内存中指令的寻址与数据的寻址是交替进行的。

**指令寻址：**

1. **顺序寻址方式**

由于指令地址在内存中按顺序安排，当执行一段程序时，通常是一条指令接一条指令地顺序进行。也就是说，从存储器取出第1条指令，然后执行这条指令；接着从存储器取出第2条指令，再执行第二条指令；接着再取出第3条指令。

这种程序顺序执行的过程，称为指令的顺序寻址方式。为此，必须使用程序计数器（又称指令计数器）PC来计数指令的顺序号，该顺序号就是指令在内存中的地址。

1. **跳跃寻址方式**

当程序转移执行的顺序时，指令的寻址就采取跳跃寻址方式。所谓跳跃，是指下条指令的地址码不是由程序计数器给出，而是由本条指令给出。注意，程序跳跃后，按新的指令地址开始顺序执行。因此，程序计数器的内容也必须相应改变，以便及时跟踪新的指令地址。

采用指令跳跃寻址方式，可以实现程序转移或构成循环程序，从而能缩短程序长度，或将某些程序作为公共程序引用。指令系统中的各种条件转移或无条件转移指令，就是为了实现指令的跳跃寻址而设置的。

**操作数寻址**

形成操作数的有效地址的方法称为操作数的寻址方式。由于大型机、小型机、微型机和单片机结构不同，从而形成了各种不同的操作数寻址方式。

1. **隐含寻址**

这种类型的指令，不是明显地给出操作数的地址。而是在指令中隐含着操作数的地址。例如，单地址的指令格式，就不明显地在地址字段中指出第2操作数的地址，而是规定累加寄存器AC作为第2操作数地址。指令格式明显指出的仅是第1操作数的地址D。因此，累加寄存器AC对单地址指令格式来说是隐含地址。如：DAA。

1. **立即寻址**

指令的地址字段指出的不是操作数的地址，而是操作数本身，这种寻址方式称为立即寻址。立即寻址方式的特点是指令执行时间很短，因为它不需要访问内存取数，从而节省了访问内存的时间。如：MOV AX,5678H 注意：立即数只能作为源操作数，不能作为目的操作数。

1. **直接寻址**

直接寻址是一种基本的寻址方法，其特点是：在指令格式的地址的字段中直接指出操作数在内存的地址。由于操作数的地址直接给出而不需要经过某种变换，所以称这种寻址方式为直接寻址方式。在指令中直接给出参与运算的操作数及运算结果所存放的主存地址，即在指令中直接给出有效地址。

1. **间接寻址**

间接寻址是相对直接寻址而言的，在间接寻址的情况下，指令地址字段中的形式地址不是操作数的真正地址，而是操作数地址的指示器，或者说此形式地址单元的内容才是操作数的有效地址。

1. **寄存器寻址方式和寄存器间接寻址方式**

当操作数不放在内存中，而是放在CPU的通用寄存器中时，可采用寄存器寻址方式。显然，此时指令中给出的操作数地址不是内存的地址单元号，而是通用寄存器的编号（可以是8位也可以是16位（AX，BX，CX，DX））。指令结构中的RR型指令，就是采用寄存器寻址方式的例子。如：MOV DS，AX

实例：

movl $values, %edi //用于把values标签引用的内存位置的地址传送给EDI寄存器

movl %ebx, (%edi) //把EBX寄存器中的值传送给EDI寄存器中包含的内存位置。

寄存器间接寻址方式与寄存器寻址方式的区别在于：指令格式中的寄存器内容不是操作数，而是操作数的地址，该地址指明的操作数在内存中。

1. **相对寻址方式**

相对寻址是把程序计数器PC的内容加上指令格式中的形式地址D而形成操作数的有效地址。程序计数器的内容就是当前指令的地址。“相对”寻址，就是相对于当前的指令地址而言。采用相对寻址方式的好处是程序员无须用指令的绝对地址编程，因而所编程序可以放在内存的任何地方。指令格式：MOV　AX，[BX+1200H]操作数物理地址PA=(DS/SS)\*10H+EA EA=(BX/BP/SI/DI)+(6/8)位偏移量Disp 对于BX，SI，DI寄存器来说段寄存器默认为DS，对于SP来说，段寄存器默认为SS

1. **基址寻址方式**

在基址寻址方式中将CPU中的基址寄存器的内容，加上变址寄存器的内容而形成操作数的有效地址。基址寻址的优点是可以扩大寻址能力，因为与形式地址相比，基址寄存器的位数可以设置得很长，从而可以在较大的存储空间中寻址。

1. **变址寻址方式**

变址寻址方式与基址寻址方式计算有效地址的方法很相似，它把CPU中某个变址寄存器的内容与偏移量D相加来形成操作数有效地址。

但使用变址寻址方式的目的不在于扩大寻址空间，而在于实现程序块的规律变化。为此，必须使变址寄存器的内容实现有规律的变化（如自增1、自减1、乘比例系数）而不改变指令本身，从而使有效地址按变址寄存器的内容实现有规律的变化。

1. **块寻址方式**

块寻址方式经常用在输入输出指令中，以实现外存储器或外围设备同内存之间的数据块传送。块寻址方式在内存中还可用于数据块移动

**Linux内存对齐：**

1. 内存对齐原因：不是所有的硬件平台都能访问任意地址上的任意数据；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。

2. 性能原因：数据结构（尤其是栈）应该尽可能得在自然边界上对齐，原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；而对齐的内存访问仅需要一次访问。

**对齐规则：**

每个特定平台的编译器都有自己默认的“对齐系数”（也叫对齐模数），程序员可以通过预编译命令#pragma pack(n)，n=1,2,4,8,16来改变这一系数，其中的n就是你要指定的“对齐系数”。n不可大于16且必须是这几个数，即为2的倍数，不然会报警告。

规则：

1. 数据成员对齐规则：结构（struct）或者联合（union）的数据成员，第一个数据成员放在offset为0的地方，以后每个数据成员的对齐按照#pragma pack指定的数值和这个数据成员自身长度中，比较小的那个进行。

2. 结构（或联合）的整体对齐规则：在数据成员完成各自对齐之后，结构（联合）本身也要进行对齐，对齐将按照#pragma pack指定的数值和结构（或联合）最大数据成员长度中，比较小的那个进行。

3. 结合1、2可推断：当#pragma pack的n值等于或者超过所有数据成员长度的时候，这个n值得大小将不产生任何效果。

实例：

#pragma pack(n) /\*n = 1,2,4,8,16\*/

struct test\_t{

int a;

char b;

short c;

char d;

};

其中：

sizeof(char) = 1

sizeof(short) = 2

sizeof(int) = 4

通过更改#pragma pack(n)的对齐系数，查看sizeof(struct test\_t)的值。

当#pragma pack(1)，即n = 1 时：

成员对齐：

#pragma pack(1)

struct test\_t {

int a; //长度4 > 1 按1对齐；起始offset=0；存放位置空间[0,3]

char b; //长度1 = 1 按1对齐，起始offset=4；存放位置空间[4]

short c; //长度2 > 1 按1对齐；起始offset=5；存放位置区间[5,6]

char d; //长度1 = 1 按1对齐，起始offset=7；存放位置区间[7]

}

成员总大小 = 8

sizeof(struct test\_t) = 8

整体对齐：

整体对齐系数：min(max(int, short, char), 1) = 1

当#pragma pack(2)，即n = 2 时：

成员对齐：

#pragma pack(2)

struct test\_t {

int a; //长度4 > 2 按2对齐；起始offset=0；存放位置空间[0,3]

char b; //长度1 < 2 按1对齐，起始offset=4；存放位置空间[4]

short c; //长度2 = 1 按1对齐；起始offset=6；存放位置区间[6,7]

char d; //长度1 < 2 按1对齐，起始offset=8；存放位置区间[8]

//最后补上1个空位。以满足整体是2的倍数对齐。

}

成员总大小 = 9

sizeof(struct test\_t) = 10

整体对齐：

整体对齐系数：min(max(int, short, char), 2) = 2

8字节对齐(#pragma pack(8))  
 输出结果：sizeof(struct test\_t) = 12  
 分析过程：  
 1) 成员数据对齐  
 #pragma pack(8)  
 struct test\_t {  
 int a;  /\* 长度4 < 8 按4对齐；起始offset=0；存放位置区间[0,3] \*/  
 char b;  /\* 长度1 < 8 按1对齐；起始offset=4；存放位置区间[4] \*/  
  short c; /\* 长度2 < 8 按2对齐；起始offset=6；存放位置区间[6,7] \*/  
  char d;  /\* 长度1 < 8 按1对齐；起始offset=8；存放位置区间[8]

//最后还要补上3个空位，以满足总长度是4(int是4个字节，4<8)的整数倍

};

成员总大小=9

整体对齐  
 整体对齐系数 = min((max(int,short,char), 8) = 4

另外例子：默认4字节对齐

struct test{

char a; //1 < 4 按1对齐，偏移offset=1，存放位置区间[0]

int b; //4 = 4 按4对齐，偏移offset=4，存放位置区间[4,7]

short c; //2 < 4 按2对齐，偏移offset=8，存放位置区间[8-9]

//整体对齐时，要保持是4的倍数，因此要有2个空位

};

sizeof(struct test) = 12。

**获取对象对齐的方法：\_\_alignof\_\_**

\_\_alignof\_\_运算符可以查询对象的对齐方法，或类型通常需要的最小对齐。其语法和sizeof类似。

如果\_\_alignof\_\_的操作数是一个对象而不是类型，则其值是其类型所需对齐，并考虑使用对齐相关的\_\_attribute\_\_扩展指定的任何最小对齐。

实例：

#include <stdio.h>

#pragma pack(2)

int main()

{

struct test{

char i;

int a;

long c;

int d;

};

int x = \_\_alignof\_\_(struct test);

int y = sizeof(struct test);

printf("alignof x = %d\n", x);

printf("sizefo y = %d\n", y);

printf("\_\_alignof\_\_(long) = %d\n", \_\_alignof\_\_(long));

printf("sizeof(long) = %d\n", sizeof(long));

}

执行结果：

alignof x = 2

sizefo y = 18

\_\_alignof\_\_(long) = 8

sizeof(long) = 8

**GCC内嵌汇编：**

**一、基本的内联汇编：**

格式：

\_\_asm\_\_ [\_\_volatile\_\_](

"instruction list"

); 或者：

asm [\_\_volatile\_\_](

"instruction list"

)

其中：

1、\_\_asm\_\_：是GCC定义的关键字asm的宏定义（#define \_\_asm\_\_ asm），它用来声明一个内联汇编表达式，任何一个内联汇编都以他开头，他是必不可少的，如果要编写符合ANSI C标准的代码（即与ANSI C兼容），那就要使用\_\_asm\_\_；

2、\_\_volatile\_\_：他是GCC关键字volatile的宏定义，这个选项是可选的，他向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”；如果不是用\_\_volatile\_\_，则当你使用了优化选项-O进行优化编译时，GCC将根据自己的判断来决定是否将这个内联汇编表达式中的指令优化掉；如果编写符合ANSI C标准的代码（即与ANSI C兼容），那就要使用\_\_volatile\_\_。

3、instruction list：它是汇编指令列表，可以是空列表，比如：

\_\_asm\_\_ \_\_volatile\_\_("");或者\_\_asm\_\_("");都是合法的内联汇编表达式，只不过这两条语句什么都不做，没有什么意义；但是并非所有的"instruction list"为空的内联汇编表达式都是没有意义的，比如：\_\_asm\_\_("":::"memory")，就是有意义的，他向GCC声明：“我对内存做了改动”，这样，GCC在编译的时候，就会将此因素考虑进去。

注意：在基本的内联汇编中，无法使用局部变量。必须使用全局的变量。

实例：

//无法编译通过。

#include <stdio.h>

int main()

{

int n = 1;

asm volatile(

"mov n, %eax"

);

return 0;

}

//编译通过

#include <stdio.h>

int n = 1;

int main()

{

asm volatile(

"mov n, %eax"

);

return 0;

}

instruction list编写规则：

当指令列表中有多条指令时，可以在一对双引号中全部写出，也可将一条或多条指令放在一对引号中，所有指令放在多对引号中；如果是将所有指令写在一对双引号中，那么相邻两条指令之间必须用分号“;”或换行符（\n）隔开，如果使用换行符，通常在\n后面还要跟一个\t；或者是相邻两条指令分别卸载单独的两行中。

规则1：任意两条指令之间要么被分号（;）或换行符（\n）或（\n\t）分隔开，要么单独放在两行。

规则2：单独放在两行的方法既可以通过\n或者\n\t的方法来实现，也可以真正的放在两行。

规则3：可以使用1对或多对引号，每1对引号中可以放1条或者多条指令，所有指令都要放在双引号中。如果将指令放在多对引号中，则除了最后一对双引号之外，前面的所有双引号里的最后一条指令都要以一个分号（;）或（\n）或（\n\t）。

如：

\_\_asm\_\_(

"movl %eax, %ebx

sti\n"

"popl %edi;"

"sub %ecx, %bx"

)

**二、带有C/C++表达式的内联汇编：**

GCC允许你通过C/C++表达式指定内联汇编中“instruction list”中的指令来输入和输出，你甚至可以不用关心到底使用了哪些寄存器，完全依靠GCC来安排和指定，这一点可以让程序员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。

1、带有C/C++表达式的内联汇编语句的格式：

\_\_asm [\_\_volatile\_\_]("instruction list":Output:Input:Clobber/Modify);

圆括号中的内容被冒号“:”分为四部分：

这几部分书写规则：

A、 第四部分的"Clobber/Modify"可以为空；如果"Clobber/Modify"为空，则其前面的冒号（：）必须省略，比如：

\_\_asm\_\_("movl %%eax,%%ebx":"=b"(foo):"a"(inp):);

是非法的，而语句

\_\_asm\_\_("movl %%eax,%%ebx":"=b"(foo):"a"(inp));

则是合法的。

B、如果第一部分的"instruction list"为空，则input、output、Clobber/Modify可以为空，也可以不为空；比如：

\_\_asm\_\_("":::"memory");和\_\_asm\_\_(""::);

都是合法的写法。

C、如果Output、Input和Clobber/Modify都为空,那么，Output、Input之前的冒号（:）可以省略，也可以不省略；如果都省略，则此汇编就退化为一个基本汇编，否则，仍然是一个带有C/C++表达式的内联汇编，此时"instruction list"中的寄存器的写法要遵循相关规定，比如：寄存器名称前面必须使用两个百分号(%%)，基本内联汇编中的寄存器名称前面只有一个百分号(%)，比如：

\_\_asm\_\_("movl %%eax,%%ebx"::);

\_\_asm\_\_("movl %%eax,%%ebx":);

\_\_asm\_\_("movl %%eax,%%ebx");

都是正确的写法，而

\_\_asm\_\_("movl %eax,%ebx"::);

\_\_asm\_\_("movl %eax,%ebx":);

\_\_asm\_\_("movl %%eax,%%ebx");

都是错误的写法。

D、如果Input、Clobber/Modify为空，但Output不为空，则Input前面的冒号(:)可以省略，也可以不省略。比如：

\_\_asm\_\_("movl %%eax,%%ebx":"=b"(foo):);

\_\_asm\_\_("movl %%eax,%%ebx":"=b"(foo));

都是正确的。

E、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前面的冒号必须省略，而Output前面的冒号必须保留。如果Clobber/Modify不为空，而Input和Output都为空，则Input和Output前面的冒号都必须保留。比如

\_\_asm\_\_("movl %%eax,%%ebx"::"a"(foo));

\_\_asm\_\_("movl %%eax,%%ebx":::"ebx");

注意：基本内联汇编中的寄存器名称前面只能有一个百分号（%），而带有C/C++表达式的内联汇编中的寄存器名称前面必须有两个百分号（%%）。

实例：

#include <stdio.h>

int main()

{

int n, a, x;

\_\_asm\_\_ \_\_volatile\_\_(

"mov $4, %%rax":"=a" (n)

)

\_\_asm\_\_ \_\_volatile\_\_(

"mov %1, %0":"=a" (a):"m" (n)

)

\_\_asm\_\_ \_\_volatile(

"mov %%rax, %0":"=m" (x)

)

printf("n = %d\n", n);

printf("a = %d\n", a);

printf("x = %d\n", x);

return 0;

}

Output输出部分：

Output部分用来指定当前内联汇编语句的输出，成为输出表达式。

格式为："操作约束" (输出表达式)

在上面的例子"mov $4, %%rax":"=a" (n)中，这个语句的Output部分就是（"=a" (n)），他是一个操作表达式，指定了一个内联汇编语句的输出部分。

Output部分由两个部分组成：由双引号括起来的部分和又圆括号括起来的部分，这两个部分是一个Output部分不可缺少的。

用双括号括起来的部分就是C/C++表达式，它用于保存当前内联汇编语句的一个输出值，其操作就是C/C++赋值语句"="的左值部分，因此，圆括号中指定的表达式只能是C/C++中赋值语句的左值表达式，即放在等号=左边的表达式，也就是说，Output部分只能作为C/C++赋值操作左边的表达式。

用算引号括起来的部分就指定了C/C++中赋值表达式的右值来源，这个部分被称作是“操作约束”（Operation Constraint），也可以称为“输出约束”。在这个例子中的操作约束就是"=a"，这个约束包括两个部分：等号和字母a。等号说明圆括号中的表达式n是一个只写的表达式，只能被用作当前内联汇编语句的输出，而不能作为输入；字母a是寄存器ras/eax/ax/al的缩写，说明n的值要从寄存器rax中获取，也就是说n=%rax，最终这句话被转换为汇编指令就是：mov %rax, address\_of\_n

注意：很多文档中都声明,所有输出操作的的操作约束都必须包含一个等号(=),但是GCC的文档中却明确地声明,并非如此;因为等号(=)约束说明当前的表达式是一个只写的,但是还有另外一个符号:加号(+),也可以用来说明当前表达式是可读可写的;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只读的;因此,对于输出操作来说,肯定必须是可写的,而等号(=)和加号(+)都可表示可写,只不过加号(+)同时也可以表示可读;所以,对于一个输出操作来说,其操作约束中只要包含等号(=)或加号(+)中的任意一个就可以了;  
等号(=)与加号(+)的区别:等号(=)表示当前表达式是一个纯粹的输出操作,而加号(+)则表示当前表达式不仅仅是一个输出操作,还是一个输入操作;但无论是等号(=)还是加号(+),所表示的都是可写,只能用于输出,只能出现在Output部分,而不能出现在Input部分;在Output部分可以出现多个输出操作表达式,多个输出操作表达式之间必须用逗号(,)隔开;

Input部分：

Input部分用来指定当前内联汇编语句的输入，称为输入表达式。

格式："操作约束"(输出表达式)

在上面例子"mov %1, %0":"=a" (a):"m" (n)，其中表达式"m"(n)就是输入表达式，用来表示一个对当前内联汇编的输入。

Input同样也由两部分组成：由双引号括起来的部分和由圆括号括起来的部分；这两个部分对于当前内联汇编语句的输入来说也是必不可少的。

在这个例子中，由双引号括起来的部分是"a"，用圆括号括起来的部分是(n)。

用双引号括起来的部分就是C/C++表达式，它为当前内联汇编语句提供一个输入值，在这里，圆括号中的表达式n是一个C/C++语言的表达式，它不必是左值表达式，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，Input可以是一个变量、一个数字，还可以是一个复杂的表达式(如:a+b/c\*d);  
比如,上例还可以这样写:

\_\_asm\_\_("movl %0,%%rax"::"a"(foo));

\_\_asm\_\_("movl %0,%%rax"::"a"(0x12345));

\_\_asm\_\_("movl %0,%%rax"::"a"(va:vb/vc));

用双引号括起来的部分就是C/C++中赋值表达式的右值表达式，用于约束当前内联汇编语句中的当前输入。这个部分称为"操作约束"，也可以称为是"输入约束"。与输出表达式中的操作约束不同的是，输入表达式中的操作约束不允许指定等号(=)约束或加号(+)约束。也就是说，它只能是只读的，约束中必须指定一个寄存器约束，例子中的字母a表示当前输入变量n要通过寄存器EAX输入到当前内联汇编语句中。

**三、操作约束：Operation Constraint**

操作约束只会出现在带有C/C++表达式的内联汇编语句中，每个Input和Output表达式都必须指定自己的操作约束Operation Constraint。

约束类型有：寄存器约束，内存约束，立即数约束，通用约束。

操作表达式的格式：

"约束"(C/C++表达式)

1、寄存器约束：

当你的输入或者输出需要借助一个寄存器时，你需要为其指定一个寄存器约束。可以直接指定一个寄存器的名字。比如：

\_\_asm\_\_ \_\_volatile\_\_("mov %0, %%rbx"::"rax" (rcx))

也可以指定寄存器的所写名称，比如：字母a，那么GCC将会根据当前操作表达式中C/C++表达式的宽度来使用%rax、%eax、%ax还是%al。如：

unsigned short \_\_shrt;

\_\_asm\_\_ \_\_volatile\_\_("mov $0, %%bx"::"a" (\_\_shrt));

由于变量\_\_shrt是16位无符号类型大小是两个字节，所以，编译器编译出来的汇编代码中，则会让变来那个使用%ax。

无论是Input还是Output操作约束，都可以使用寄存器约束。

常用的寄存器约束的缩写：

r：I/O，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认为是合适的

q：I/O，表示使用一个通用寄存器,与r的意义相同

g：I/O，表示使用寄存器或内存地址

m：I/O，表示使用内存地址;  
 a：I/O，表示使用%eax/%ax/%al;  
 b：I/O，表示使用%ebx/%bx/%bl;  
 c：I/O，表示使用%ecx/%cx/%cl;  
 d：I/O，表示使用%edx/%dx/%dl;  
 D：I/O，表示使用%edi/%di;  
 S：I/O，表示使用%esi/%si;  
 f：I/O，表示使用浮点寄存器;  
 t：I/O，表示使用第一个浮点寄存器;  
 u：I/O，表示使用第二个浮点寄存器;  
 A：I/O，表示把%eax与%edx组合成一个64位的整数值;  
 o：I/O，表示使用一个内存位置的偏移量;  
 V：I/O，表示仅仅使用一个直接内存位置;  
 i：I/O，表示使用一个整数类型的立即数;  
 n：I/O，表示使用一个带有已知整数值的立即数;  
 F：I/O,表示使用一个浮点类型的立即数;

2、内存约束：

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（指针变量），不想借助任何寄存器，则可以使用内存约束。比如：

\_\_asm\_\_ \_\_volatile\_\_(

"mov %%rax, %0":"m" (x)

)

内存约束使用约束名“m”，表示的是使用系统支持的任何一种内存方式，不需要借助寄存器。

内存约束使用约束名"m"，表示的是使用系统支持的任何一种内存方式，不需要借助于寄存器；使用内存约束方式进行输入输出时，由于不借助于寄存器，所以，GCC不会按照你的声明对其做任何的输入输出处理；GCC只会直接拿来使用，对这个C/C++表达式而言，究竟是输入还是输出，完全依赖于你写在"instruction list"中的指令对其操作的方式；所以，不管你把操作约束和操作表达式放在Input部分还是放在Output部分，GCC编译生成的汇编代码都是一样的，程序的执行结果也都是正确的；本来我们将一个操作表达式放在Input或Output部分是希望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不会为它做任何事情，那么放在哪里就无所谓了；但是从程序员的角度来看，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；

3、立即数约束：

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或内存，则可以使用立即数约束。

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

\_\_asm\_\_ \_\_volatile\_\_("movl %0, %%eax"::"i"(100));

立即数约束使用约束名"i"表示输入表达式是一个整数类型的立即数，不需要借助于任何寄存器，只能用于Input部分。使用约束名"F"表示输入表达式是一个浮点数类型的立即数。

4、通用约束：

约束名“g”可以用于输入和输出，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。

约束名“0,1,2,3,4,5,6,7,8,9”只能用户输入，表示与第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++表达式在实际操作中,无论使用寄存器方式、内存方式还是立即数方式都无所谓时,或者程序员想实现一个灵活的模板,以让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。

例如:

#define JUST\_MOV(foo) \_\_asm\_\_("movl %0,%%eax"::"g"(foo))

则，JUST\_MOV(100)和JUST\_MOV(var)就会让编译器产生不同的汇编代码

对于JUST\_MOV(100)的汇编代码为:

#APP

  movl $100,%eax      #立即数方式

#NO\_APP

对于JUST\_MOV(var)的汇编代码为:

#APP  
 movl 8(%ebp),%eax   #内存方式;  
 #NO\_APP

像这样的效果,就是通用约束g的作用

5、修饰符：

等号（=）和加号（+）作为修饰符，只能用于Output部分，等号表示当前输出表达式为只写，加号表示当前输出表达式的属性为可读可写。这两个修饰符用于约束对输出表达式的操作，他们被写在输出表达式的约束部分中，并且只能写在第一个字符的位置。

字符&也写在输出表达式的约束部分，用户约束寄寄存器的分配，但是只能写在约束部分的第二个字符位置上。

用符号&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操作表达式被输入之前输出。即:GCC会先使用输出值对被修饰符&修饰的Output操作表达式进行填充,然后,才对Input操作表达式进行输入。

这样的话,如果不使用修饰符&对Output操作表达式进行修饰,一旦后面的Input操作表达式使用了与Output操作表达式相同的寄存器,就会产生输入输出数据混乱的情况。相反,如果用修饰符&修饰输出操作表达式,那么,就意味着GCC会先把Input操作表达式的值输入到选定的寄存器中,然后经过处理,最后才用输出值填充对应的Output操作表达式。

所以,修饰符&的作用就是要求GCC编译器为所有的Input操作表达式分配别的寄存器,而不会分配与被修饰符&修饰的Output操作表达式相同的寄存器;修饰符&也写在操作约束中,即:&约束;由于GCC已经规定加号(+)或等号(=)占据约束的第一个字符,那么&约束只能占用第二个字符。

例如：

int \_\_out, \_\_in1, \_\_in2;  
 \_\_asm\_\_(

"popl %0\n\t"  
        "movl %1,%%esi\n\t"  
        "movl %2,%%edi\n\t"  
        :"=&a"(\_\_out)  
        :"r"(\_\_in1),"r"(\_\_in2)

);

**注意**: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(意思是GCC可以从多个寄存器中选取一个,或使用非寄存器方式)时,比如"r"或"g"时,此Output操作表达式使用符号&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/立即数约束时,则此Output操作表达式使用符号&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符号&来修饰,因为\_\_in1和\_\_in2都已经被指定了固定的寄存器,或使用了内存方式,GCC无从选择;如果你已经为某个Output操作表达式指定了修饰符&,并指定了固定的寄存器,那么,就不能再为任何Input操作表达式指定这个寄存器了,否则会出现编译报错;  
 比如:  
 \_\_asm\_\_("popl %0; movl %1,%%esi; movl %2,%%edi;":"=&a"(\_\_out):"a"(\_\_in1),"c"(\_\_in2));  
 对这条语句的编译就会报错;  
 相反,你也可以为Output指定可选约束,比如"r"或"g"等,让GCC为此Output操作表达式选择合适的寄存器,或使用内存方式,GCC在选择的时候,会排除掉已经被Input操作表达式所使用过的所有寄存器,然后在剩下的寄存器中选择,或者干脆使用内存方式;  
 比如:  
 \_\_asm\_\_("popl %0; movl %1,%%esi; movl %2,%%edi;":"=&r"(\_\_out):"a"(\_\_in1),"c"(\_\_in2));  
 这三个修饰符（=、+、&）只能用在Output操作表达式中,而修饰符%则恰恰相反,它只能用在Input操作表达式中。

修饰符%用于向GCC声明:"当前Input操作表达式中的C/C++表达式可以与下一个Input操作表达式中的C/C++表达式互换"。这个修饰符一般用于符合交换律运算的地方，比如：加、乘、按位与&、按位或|等等;

例如:

\_\_asm\_\_("addl %1,%0\n\t":"=r"(\_\_out):"%r"(\_\_in1),"0"(\_\_in2));  
 其中,"0"(\_\_in2)表示使用与第一个Input操作表达式("r"(\_\_in1))相同的寄存器或内存，由于使用符号%修饰\_\_in1的寄存器方式r，那么就表示，\_\_in1与\_\_in2可以互换位置。加法的两个操作数交换位置之后和不变。

**四、占位符：**

每一个占位符对应一个Input/Output操作表达式。

带C/C++表达式的内联汇编有两种展位符：序号占位符和名称占位符。

1、序号占位符：

GCC规定：一个内联汇编语句中最多只能有10个Input/Output操作表达式，这些操作表达式按照它们被列出来的顺序依次赋予编号0到9，对于占位符中的数字而言，与这些编号是对应的。比如：占位符%0对应编号为0的操作表达式，占位符为%1对应编号为1的操作表达式，以此类推。

由于占位符前面要有一个百分号%，为了区别占位符与寄存器，GCC规定：在带有C/C++表达式的内联汇编的语句指令列表里列出的寄存器名称前面必须使用两个百分号%%，以用来区别占位符语法。

GCC对占位符进行编译的时候，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/立即数。

例如：

\_\_asm\_\_(

"addl %1, %0":"=a"(\_\_out):"m"(\_\_in1),"a"(\_\_in2)

)

这个语句中，%0对应Output操作表达式"=a"(\_\_out)，而"=a"(\_\_out)指定的寄存器是%eax，所以，占位符%0被替换为%eax。占位符%1对应Input操作表达式"m"(\_\_in1)，而"m"(\_\_in1)被指定为内存，所以占位符%1被替换位\_\_in1的内存地址。

一句话描述：序号占位符就是前面描述的%0、%1、%2、%3、%4、%5、%6、%7、%8、%9，其中，每一个占位符对应一个Input/Output的C/C++表达式。

2、名称占位符：

由于GCC中限制这种占位符的个数最多只能由这10个,这也就限制了Input/Output操作表达式中C/C++表达式的数量做多只能有10个。如果需要的C/C++表达式的数量超过10个，那么这些需要占位符就不够用了。

GCC内联汇编提供了名称占位符来解决这个问题。即：使用一个名字字符串与一个C/C++表达式对应，这个名字字符串就称为名称占位符，而这个名字通常使用与C/C++表达式中的变量完全相同的名字;

使用名字占位符时,内联汇编的Input/Output操作表达式中的C/C++表达式的格式如下:

[name] "constraint"(变量)

此时,指令列表中的占位符的书写格式如下:

%[name]  
 这个格式等价于序号占位符中的%0,%1,$2等等。

使用名称占位符时,一个name对应一个变量。

例如:  
 \_\_asm\_\_(

"imull %[value1],%[value2]":[value2] "=r"(data2):[value1] "r"(data1),"0"(data2)

);

此例中，名称占位符value1就对应变量data1，名称占位符value2对应变量data2。GCC编译的时候同样会把这两个占位符分别替换成对应的变量所使用的寄存器/内存地址/立即数，而且也增强了代码的可读性。

这个例子,使用序号占位符的写法如下：

\_\_asm\_\_(

"imull %1,%0":"=r"(data2):"r"(data1),"0"(data2)

);

**五、寄存器/内存修改标示(Clobber/Modify)**

有时候，当你想通知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部分声明这些寄存器或内存。

1、寄存器修改通知:

这种情况一般发生在一个寄存器出现在指令列表中，但又不是Input/Output操作表达式所指定的,也不是在一些Input/Output操作表达式中使用"r"或"g"约束时由GCC选择的,同时,此寄存器被指令列表中的指令所修改,而这个寄存器只供当前内联汇编语句使用的情况。比如：

\_\_asm\_\_("movl %0,%%ebx"::"a"(\_\_foo):"bx");

这个内联汇编语句中，%ebx出现在指令列表中，并且被指令修改了，但是却未被任何Input/Output操作表达式是所指定，所以,你需要在Clobber/Modify部分指定"bx",以让GCC知道这一点。

**因为你在Input/Output操作表达式中指定的寄存器,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对这些寄存器的状态是非常清楚的,它知道这些寄存器是被修改的,你根本不需要在Clobber/Modify部分声明它们;但除此之外,GCC对剩下的寄存器中哪些会被当前内联汇编语句所修改则一无所知;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部分声明它们,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不一致,从而造成程序执行错误。**

在Clobber/Modify部分声明这些寄存器的方法很简单，只需要将寄存器的名字用双引号括起来就可以，如果要声明多个寄存器,则相邻两个寄存器名字之间用逗号隔开。

例如:

\_\_asm\_\_("movl %0,%%ebx; popl %%ecx"::"a"(\_\_foo):"bx","cx");  
 这个语句中,声明了bx和cx，告诉GCC：寄存器%ebx和%ecx可能会被修改，要求GCC考虑这个因素。

**寄存器名称串**：

"al"/"ax"/"eax":代表寄存器%eax  
"bl"/"bx"/"ebx":代表寄存器%ebx  
"cl"/"cx"/"ecx":代表寄存器%ecx  
"dl"/"dx"/"edx":代表寄存器%edx  
"si"/"esi":代表寄存器%esi  
"di"/"edi":代表寄存器%edi  
 所以,只需要使用"ax","bx","cx","dx","si","di"就可以了,因为他们都代表对应的寄存器;  
 如果你在一个内敛汇编语句的Clobber/Modify部分向GCC声明了某个寄存器内存发生了改变,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代码生成之后,再将其内容回复;  
 另外需要注意的是,如果你在Clobber/Modify部分声明了一个寄存器,那么这个寄存器将不能再被用作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"/"g",GCC也不会选择已经被声明在Clobber/Modify部分中的寄存器;  
例如:  
 \_\_asm\_\_("movl %0,%%ebx"::"a"(\_\_foo):"ax","bx");  
 这条语句中的Input操作表达式"a"(\_\_foo)中已经指定了寄存器%eax,那么在Clobber/Modify部分中个列出的"ax"就是非法的;编译时,GCC会报错;  
2.内存修改通知:  
 除了寄存器的内容会被修改之外,内存的内容也会被修改;如果一个内联汇编语句的指令列表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操作表达式中使用"m"约束,这种情况下,你需要使用在Clobber/Modify部分使用字符串"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\d"  
          "rep\n\t"  
          "stosb"  
          :/\*no output\*/  
          :"a"(c),"D"(s),"c"(count)  
          :"cx","di","memory");  
  return s;  
}  
 如果一个内联汇编语句的Clobber/Modify部分存在"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很可能已经和内存处的内容不一致了;  
3.标志寄存器修改通知:

当一个内联汇编中包含影响标志寄存器eflags的条件,那么也需要在Clobber/Modify部分中使用"cc"来向GCC声明这一点;

## 文件系统扩展属性（Extended attributes）：

在一个文件系统上，这是附加在一个文件或者文件夹上的键值对关联的属性。他们也被称作EA或者xattrs。可以在一个文件上添加任何键值对的属性。键值对字符串的长度受限于所实现EA的文件系统。

一个使用EA的注明例子就是ACL（访问控制列表）。

名称空间：

EA通常使用句点分割的像“user.foo”这样的格式。第一部分指定一个名称空间，这用来解决相同EA名称的冲突。

“user”名称空间：此命名空间被正常的Linux权限系统所保护（也就是对文件有写权限的用户可以去设置EA），

“root”或者“system”名称空间：只能被root用户设置，ACL使用这个名称空间。

“security”名称空间：SELinux使用这个空间，如'security.selinux'属性。

命令：

getfattr setfattr attr setfacl getfacl等。

## 访问控制机制：

**自主访问控制（Discretionary Access Control，DAC）：**

由客体的属主对字节的客体进行管理，由属主自己决定是否将自己的客体访问权或者部分访问权授予其他主体，这种控制方式是自主的。也就是说，在自主访问控制下，用户可以按照自己的意愿，有选择的与其他用户共享他的文件。

自主访问控制是保护系统资源不被非法访问的一种有效手段。但这种控制是自主的，即它是以保护用户的个人资料的安全为目标并以个人的意志为转移的。

自主访问控制是一种比较宽松的访问控制,一个主题的访问权限具有传递性。[1]

计算机信息系统可信计算基定义和控制系统中命名用户对命名客体的访问。

实施机制（例如：访问控制表）允许命名用户以用户和（或）用户组的身份规定并控制客体的共享；阻止非授权用户读取敏感信息。并控制访问权限扩散。

自主访问控制机制根据用户指定方式或默认方式，阻止非授权用户访问客体。访问控制的粒度是单个用户。没有存取权的用户只允许由授权用户指定对客体的访问权。阻止非授权用户读取敏感信息。

自主访问控制有两个至关重要的标准：

（1）文件和数据的所有权：系统中的每个物体都有所有者。在大多数DAC系统中，物体的所有者是产生这个物体的人（或事件，或另一个物体）。那么此物体的自主访问控制权限由它的产生者决定；

（2）访问权限及获批：物体的所有者拥有访问权限，并且可以批准他人试图访问的请求。

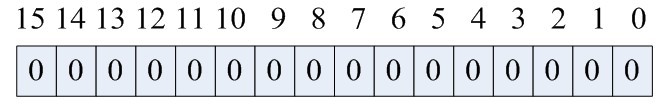
**可信计算基：**

## 数据结构：位图法

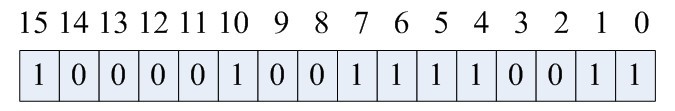
位图法就是bitmap的缩写。所谓bitmap，就是用每一位来存放某种状态，适用于大规模数据，但数据状态又不是很多的情况。通常是用来判断某个数据存不存在的。在STL中有一个bitmap容易，其实就是位图法。

定义：unsigned int bit[N]

在这个数组里面，可存N\*sizeof(int)\*8个数据，但最大的数只能是N \* sizeof(int) \* 8 - 1。假如，我们要存储的数据范围为0-15，则我们只需要使得N=1，这样就可以把数据存进去。其实也就是要存入的数是几就在数组对应的位上写1。比如15，则在第15位上写1。



假如，存储数据为【5,1,7,15,0,4,6,10】，则存入这个结构中的情况为：



实例: unsigned char bit[8\*1024]，可存数据：8\*1024\*sizeof(char)-1

注意：此处定义的是char，而不是int，所以bit数组每个元素占一个字节，8位。如果换成int的话，则bit数组每个元素占4个字节，32位，下面的计算公式将8改为32。

写入1234：

计算字节位置（即数据应该放在数组的第几个元素中，元素是字节）：

int nBytePos = 1234/8=154（每个字节有8个位，所以每个字节可以存入8个数据）

位位置（上面计算出的那个字节的第几位）：

int nBitPos = 1234%8=2

即1234应该存放在数组下标38字节处，把该字节的2号位（0~7）置为1。

unsigned short val = 1 << nBitPos

bit[nBytePos] = bit[nBytePos] | val

程序状态字：

用一个专门的寄存器来指示处理器状态，称为程序状态字（PSW Program Status Word）

程序状态寄存器PSW是计算机系统的核心部件--运算器的一部分。PSW用来存放两类 信息：一类是提现当前指令执行结果的各种状态信息，如有无进位（CY位），有无溢出 （OV位），结果正负（SF位），结果是否为零（ZF位），奇偶标志位（P位）等；另一类 是存放控制信息，如允许中断（IF位），跟踪标志（TF位）等。有些机器中将PSW称为 标志寄存器FR（Flag Register）。

程序状态字通常包括以下状态代码：

CPU的工作状态码--指明管态还是目态，用来说明当前在CPU上执行的是操作系统 还是一般用户，从而决定是否可以使用特权指令或拥有其他的特殊权力。

条件码--反映指令执行后的结果特征。

中断屏蔽码--指出是否允许中断。

目态🡪管态

其转换的唯一途径是通过中断。

管态🡪目态

可用设置PSW（修改程序状态字）实现。