内嵌汇编语法如下：

__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 1：

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

例二（bitops.h）：

```
Static __inline__ void __set_bit(int nr, volatile void * addr)
{
        __asm__(
                        "btsl %1,%0"
                        :"=m" (ADDR)
                        :"Ir" (nr));
}
```

后例功能是将 (*addr)的第 nr位设为 1。第一个占位符%0与 C 语言变量 ADDR对应，第二个占位符%1与 C 语言变量 nr对应。因此上面的汇编语句代码与下面的伪代码等价： btsl nr, ADDR, 该指令的两个操作数不能全是内存变量，因此将 nr的限定字符串指定为“ Ir”，将 nr 与立即数或者寄存器相关联，这样两个操作数中只有 ADDR为内存变量。

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
	“ ”	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 ”，	表示用它限制的操作数与某个指定的操作数匹配，

	“ 1” ...	也即该操作数就是指定的那个操作数，例如“ 0”
	“ 9”	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

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。