在 Linux 代码中，经常可以看到在 C 代码中，嵌入部分汇编代码，这些代码要么是与硬件体系相关的，要么是对性能有关键影响的。

在很久以前，我特别惧怕内嵌汇编代码，直到后来把汇编部分的短板补上之后，才彻底终结这种心理。

也许你在工作中，几乎不会涉及到内嵌汇编代码的工作，但是一旦进入到系统的底层，或者需要对时间关键场景进行优化，这个时候你的知识储备就发挥重要作用了！

这篇文章，我们就来详细聊一聊在 C 语言中，如何通过 asm 关键字来嵌入汇编语言代码，文中的 8 个示例代码从简单到复杂，逐步深入地介绍内联汇编的关键语法规则。

希望这篇文章能够成为你进阶高手路上的垫脚石！

PS:

> 1. 示例代码中使用的是 Linux 系统中 AT&T 汇编语法；
> 2. 文章中的 8 个示例代码，可以在公众号后台回复【426】，即可收到下载地址；

# 一、基本 asm 格式

gcc 编译器支持 2 种形式的内联 asm 代码：

> 1. 基本 asm 格式：不支持操作数；
> 2. 扩展 asm 格式：支持操作数；

## 1. 语法规则

```
asm [volatile] ("汇编指令")
```

1. 所有指令，必须用双引号包裹起来；
2. 超过一条指令，必须用\n分隔符进行分割，为了排版，一般会加上\t；
3. 多条汇编指令，可以写在一行，也可以写在多行；
4. 关键字 asm 可以使用 **asm** 来替换；
5. volatile 是可选的，编译器有可能对汇编代码进行优化，使用 volatile 关键字之后，告诉编译器不要优化手写的内联汇编代码。

## 2. test1.c 插入空指令

```
#include <stdio.h>
int main()
{
    asm ("nop");
    printf("hello\n");
    asm ("nop\n\tnop\n\t"
   "nop");
    return 0;
}
```

注意：C语言中会自动把两个连续的字符串字面量拼接成一个，所以"nop\n\tnop\n\t" "nop" 这两个字符串会自动拼接成一个字符串。

生成汇编代码指令：

```
gcc -m32 -S -o test1.s test1.c
```

test1.s 中内容如下(只贴出了内联汇编代码相关部分的代码)：

```
#APP
# 5 "test1.c" 1
  nop
# 0 "" 2
#NO_APP
  // 这里是 printf 语句生成的代码。
#APP
# 7 "test1.c" 1
  nop
  nop
  nop
# 0 "" 2
#NO_APP
```

可以看到，内联汇编代码被两个注释(#APP ... #NO_APP)包裹起来。在源码中嵌入了两个汇编代码，因此可以看到 gcc 编译器生成的汇编代码中包含了这两部分代码。

这 2 部分嵌入的汇编代码都是空指令 nop，没有什么意义。

## 3. test2.c 操作全局变量

在 C 代码中嵌入汇编指令，目的是用来计算，或者执行一定的功能，下面我们就来看一下，如何在内联汇编指令中，操作全局变量。

```
#include <stdio.h>

int a = 1;
int b = 2;
int c;

int main()
{
    asm volatile ("movl a, %eax\n\t"
        "addl b, %eax\n\t"
        "movl %eax, c");
    printf("c = %d \n", c);
    return 0;
}
```

关于汇编指令中编译器的基本知识：

eax, ebx 都是 x86 平台中的寄存器(32位)，在基本asm格式中，寄存器的前面必须加上百分号%。

32 位的寄存器 eax 可以当做 16 位来使用(ax)，或者当做 8 位来使用(ah, al)，本文只会按照 32 位来使用。

代码说明：

> movl a, %eax　// 把变量 a 的值复制到 %eax 寄存器中；
>
> addl b, %eax　// 把变量 b 的值 与 %eax 寄存器中的值(a)相加，结果放在 %eax 寄存器中；
>
> movl %eax, c　// 把 %eax 寄存器中的值复制到变量 c 中；


生成汇编代码指令：

```
gcc -m32 -S -o test2.s test2.c
```

test2.s 内容如下(只贴出与内联汇编代码相关部分)：

```
#APP
# 9 "test2.c" 1
  movl a, %eax
  addl b, %eax
  movl %eax, c
# 0 "" 2
#NO_APP
```

可以看到，在内联汇编代码中，可以直接使用全局变量 a, b 的名称来操作。执行 test2，可以得到正确的结果。

思考一个问题：为什么在汇编代码中，可以使用变量a, b, c?

查看 test2.s 中内联汇编代码之前的部分，可以看到：

```
  .file "test2.c"
  .globl  a
  .data
  .align 4
  .type a, @object
  .size a, 4
a:
  .long 1
  .globl  b
  .align 4
  .type b, @object
  .size b, 4
b:
  .long 2
  .comm c,4,4
```

变量 a, b 被 .globl 修饰，c 被 .comm 修饰，相当于是把它们导出为全局的，所以可以在汇编代码中使用。

那么问题来了：如果是一个局部变量，在汇编代代码中就不会用 .globl 导出，此时在内联汇编指令中，还可以直接使用吗?

眼见为实，我们把这 3 个变量放到 main 函数的内部，作为局部变量来试一下。

## 4. test3.c 尝试操作局部变量

```
#include <stdio.h>
int main()
{
    int a = 1;
    int b = 2;
    int c;

    asm("movl a, %eax\n\t"
        "addl b, %eax\n\t"
        "movl %eax, c");
    printf("c = %d \n", c);
    return 0;
}
```

生成汇编代码指令：

```
gcc -m32 -S -o test3.s test3.c
```

在 test3.s 中可以看到没有 a, b, c 的导出符号，a 和 b 没有其他地方使用，因此直接把他们的数值复制到栈空间中了：

```
movl  $1, -20(%ebp)
movl  $2, -16(%ebp)
```

我们来尝试编译成可执行程序：

```
$ gcc -m32 -o test3 test3.c
/tmp/ccuY0TOB.o: In function `main':
test3.c:(.text+0x20): undefined reference to `a'
test3.c:(.text+0x26): undefined reference to `b'
test3.c:(.text+0x2b): undefined reference to `c'
collect2: error: ld returned 1 exit status
```

编译报错：找不到对 a,b,c 的引用！那该怎么办，才能使用局部变量呢？扩展 asm 格式！

# 二、扩展 asm 格式

## 1. 指令格式

asm [volatile] ("汇编指令" : "输出操作数列表" : "输入操作数列表" : "改动的寄存器")

格式说明

1. 汇编指令：与基本asm格式相同；
2. 输出操作数列表：汇编代码如何把处理结果传递到 C 代码中；
3. 输入操作数列表：C 代码如何把数据传递给内联汇编代码；
4. 改动的寄存器：告诉编译器，在内联汇编代码中，我们使用了哪些寄存器；
5. “改动的寄存器”可以省略，此时最后一个冒号可以不要，但是前面的冒号必须保留，即使输出/输入操作数列表为空。

关于“改动的寄存器”再解释一下：gcc 在编译 C 代码的时候，需要使用一系列寄存器；我们手写的内联汇编代码中，也使用了一些寄存器。

为了通知编译器，让它知道: 在内联汇编代码中有哪些寄存器被我们用户使用了，可以在这里列举出来，这样的话，gcc 就会避免使用这些列举出的寄存器

## 2. 输出和输入操作数列表的格式

在系统中，存储变量的地方就2个：寄存器和内存。因此，告诉内联汇编代码输出和输入操作数，其实就是告诉它：

1. 向哪些寄存器或内存地址输出结果;
2. 从哪些寄存器或内存地址读取输入数据;

这个过程也要满足一定的格式：

```
"[输出修饰符]约束"(寄存器或内存地址)
```

### （1）约束

就是通过不同的字符，来告诉编译器使用哪些寄存器，或者内存地址。包括下面这些字符：

a: 使用 eax/ax/al 寄存器；

b: 使用 ebx/bx/bl 寄存器；

> c: 使用 ecx/cx/cl 寄存器；
>
> d: 使用 edx/dx/dl 寄存器；
>
> r: 使用任何可用的通用寄存器；
>
> m: 使用变量的内存位置；

先记住这几个就够用了，其他的约束选项还有：D, S, q, A, f, t, u等等，需要的时候再查看文档。

### （2）输出修饰符

顾名思义，它使用来修饰输出的，对输出寄存器或内存地址提供额外的说明，包括下面4个修饰符：

1. +：被修饰的操作数可以读取，可以写入；
2. =：被修饰的操作数只能写入；
3. %：被修饰的操作数可以和下一个操作数互换；
4. &：在内联函数完成之前，可以删除或者重新使用被修饰的操作数；

语言描述比较抽象，直接看例子！

## 3. test4.c 通过寄存器操作局部变量

```
#include <stdio.h>

int main()
{
    int data1 = 1;
    int data2 = 2;
    int data3;

    asm("movl %%ebx, %%eax\n\t"
        "addl %%ecx, %%eax"
        : "=a"(data3)
        : "b"(data1),"c"(data2));

    printf("data3 = %d \n", data3);
    return 0;
}
```

有 2 个地方需要注意一下啊：

1. 在内联汇编代码中，没有声明“改动的寄存器”列表，也就是说可以省略掉(前面的冒号也不需要)；
2. 扩展asm格式中，寄存器前面必须写 2 个%；

代码解释：

1. "b"(data1),"c"(data2) ==> 把变量 data1 复制到寄存器 %ebx，变量 data2 复制到寄存器 %ecx。这样，内联汇编代码中，就可以通过这两个寄存器来操作这两个数了；
2. "=a"(data3) ==> 把处理结果放在寄存器 %eax 中，然后复制给变量data3。前面的修饰符等号意思是：会写入往 %eax 中写入数据，不会从中读取数据；

通过上面的这种格式，内联汇编代码中，就可以使用指定的寄存器来操作局部变量了，稍后将会看到局部变量是如何从经过栈空间，复制到寄存器中的。

生成汇编代码指令：

```
gcc -m32 -S -o test4.s test4.c
```

汇编代码 test4.s 如下：

```
  movl  $1, -20(%ebp)
  movl  $2, -16(%ebp)
  movl  -20(%ebp), %eax
  movl  -16(%ebp), %edx
  movl  %eax, %ebx
  movl  %edx, %ecx
#APP
# 10 "test4.c" 1
  movl %ebx, %eax
  addl %ecx, %eax
# 0 "" 2
#NO_APP
    movl  %eax, -12(%ebp)
```

可以看到，在进入手写的内联汇编代码之前：

1. 把数字 1 通过栈空间(-20(%ebp))，复制到寄存器 %eax，再复制到寄存器 %ebx;
2. 把数字 2 通过栈空间(-16(%ebp))，复制到寄存器 %edx，再复制到寄存器 %ecx;

这 2 个操作正是对应了内联汇编代码中的“输入操作数列表”部分：`"b"(data1),"c"(data2)`。

在内联汇编代码之后(#NO_APP 之后)，把 %eax 寄存器中的值复制到栈中的 -12(%ebp) 位置，这个位置正是局部变量 data3 所在的位置，这样就完成了输出操作。

## 4. test5.c 声明改动的寄存器

在 test4.c 中，我们没有声明改动的寄存器，所以编译器可以任意选择使用哪些寄存器。从生成的汇编代码 test4.s 中可以看到，gcc 使用了 %edx 寄存器。

那么我们来测试一下：告诉 gcc 不要使用 %edx 寄存器。

```
#include <stdio.h>
int main()
{
    int data1 = 1;
    int data2 = 2;
    int data3;

    asm("movl %%ebx, %%eax\n\t"
        "addl %%ecx, %%eax"
        : "=a"(data3)
        : "b"(data1),"c"(data2)
        : "%edx");

    printf("data3 = %d \n", data3);
    return 0;
}
```

代码中，asm 指令最后部分 "%edx"，就是用来告诉 gcc 编译器：在内联汇编代码中，我们会使用到 %edx 寄存器，你就不要用它了。

生成汇编代码指令：

```
gcc -m32 -S -o test5.s test5.c
```

来看一下生成的汇编代码 test5.s：

```
    movl  $1, -20(%ebp)
  movl  $2, -16(%ebp)
  movl  -20(%ebp), %eax
  movl  -16(%ebp), %ecx
  movl  %eax, %ebx
#APP
# 10 "test5.c" 1
  movl %ebx, %eax
  addl %ecx, %eax
# 0 "" 2
#NO_APP
  movl  %eax, -12(%ebp)
```


可以看到，在内联汇编代码之前，gcc 没有选择使用寄存器 %edx。

## 三、使用占位符来代替寄存器名称

在上面的示例中，只使用了 2 个寄存器来操作 2 个局部变量，如果操作数有很多，那么在内联汇编代码中去写每个寄存器的名称，就显得很不方便。

因此，扩展 asm 格式为我们提供了另一种偷懒的方法，来使用输出和输入操作数列表中的寄存器：占位符！

占位符有点类似于批处理脚本中，利用 $1, $2...来引用输入参数一样，内联汇编代码中的占位符，从输出操作数列表中的寄存器开始从 0 编号，一直编号到输入操作数列表中的所有寄存器。

还是看例子比较直接！

### 1. test6.c 使用占位符代替寄存器

```
#include <stdio.h>
int main()
{
    int data1 = 1;
    int data2 = 2;
    int data3;

    asm("addl %1, %2\n\t"
        "movl %2, %0"
        : "=r"(data3)
```

```
        : "r"(data1),"r"(data2));

    printf("data3 = %d \n", data3);
    return 0;
}
```

代码说明：

1. 输出操作数列表"=r"(data3)：约束使用字符 r, 也就是说不指定寄存器，由编译器来选择使用哪个寄存器来存储结果，最后复制到局部变量 data3中；
2. 输入操作数列表"r"(data1),"r"(data2)：约束字符r, 不指定寄存器，由编译器来选择使用哪 2 个寄存器来接收局部变量 data1 和 data2；
3. 输出操作数列表中只需要一个寄存器，因此在内联汇编代码中的 %0 就代表这个寄存器(即：从 0 开始计数)；
4. 输入操作数列表中有 2 个寄存器，因此在内联汇编代码中的 %1 和 %2　就代表这 2 个寄存器(即：从输出操作数列表的最后一个寄存器开始顺序计数)；

生成汇编代码指令：

```
gcc -m32 -S -o test6.s test6.c
```

汇编代码如下 test6.s：

```
  movl  $1, -20(%ebp)
  movl  $2, -16(%ebp)
  movl  -20(%ebp), %eax
  movl  -16(%ebp), %edx
#APP
# 10 "test6.c" 1
  addl %eax, %edx
  movl %edx, %eax
# 0 "" 2
#NO_APP
  movl  %eax, -12(%ebp)
```


可以看到，gcc 编译器选择了 %eax 来存储局部变量 data1，%edx 来存储局部变量 data2 ，然后操作结果也存储在 %eax 寄存器中。

是不是感觉这样操作就方便多了？不用我们来指定使用哪些寄存器，直接交给编译器来选择。

在内联汇编代码中，使用 %0、%1 、%2 这样的占位符来使用寄存器。

别急，如果您觉得使用编号还是麻烦，容易出错，还有另一个更方便的操作：扩展 asm 格式还允许给这些占位符重命名，也就是给每一个寄存器起一个别名，然后在内联汇编代码中使用别名来操作寄存器。

还是看代码！

## 2. test7.c 给寄存器起别名

```
#include <stdio.h>
int main()
{
    int data1 = 1;
    int data2 = 2;
    int data3;

    asm("addl %[v1], %[v2]\n\t"
        "movl %[v2], %[v3]"
        : [v3]"=r"(data3)
        : [v1]"r"(data1),[v2]"r"(data2));

    printf("data3 = %d \n", data3);
    return 0;
}
```

代码说明：

1. 输出操作数列表：给寄存器(gcc 编译器选择的)取了一个别名 v3；
2. 输入操作数列表：给寄存器(gcc 编译器选择的)取了一个别名 v1 和 v2；

起立别名之后，在内联汇编代码中就可以直接使用这些别名( %[v1], %[v2], %[v3])来操作数据了。

生成汇编代码指令：

```
gcc -m32 -S -o test7.s test7.c
```

再来看一下生成的汇编代码 test7.s：

```
  movl  $1, -20(%ebp)
  movl  $2, -16(%ebp)
  movl  -20(%ebp), %eax
  movl  -16(%ebp), %edx
#APP
# 10 "test7.c" 1
  addl %eax, %edx
  movl %edx, %eax
# 0 "" 2
#NO_APP
  movl  %eax, -12(%ebp)
```

这部分的汇编代码与 test6.s 中完全一样！

# 四、使用内存地址

在以上的示例中，输出操作数列表和输入操作数列表部分，使用的都是寄存器(约束字符：a, b, c, d, r等等)。

我们可以指定使用哪个寄存器，也可以交给编译器来选择使用哪些寄存器，通过寄存器来操作数据，速度会更快一些。

如果我们愿意的话，也可以直接使用变量的内存地址来操作变量，此时就需要使用约束字符 m。

## 1. test8.c 使用内存地址来操作数据

```
#include <stdio.h>
int main()
{
    int data1 = 1;
    int data2 = 2;
    int data3;

    asm("movl %1, %%eax\n\t"
        "addl %2, %%eax\n\t"
        "movl %%eax, %0"
```

```
        : "=m"(data3)
        : "m"(data1),"m"(data2));

    printf("data3 = %d \n", data3);
    return 0;
}
```

代码说明：

1. 输出操作数列表 "=m"(data3)：直接使用变量 data3 的内存地址；
2. 输入操作数列表 "m"(data1),"m"(data2)：直接使用变量 data1, data2 的内存地址；

在内联汇编代码中，因为需要进行相加计算，因此需要使用一个寄存器(%eax)，计算这个环节是肯定需要寄存器的。

在操作那些内存地址中的数据时，使用的仍然是按顺序编号的占位符。

生成汇编代码指令：

```
gcc -m32 -S -o test8.s test8.c
```

生成的汇编代码如下 test8.s：

```
  movl  $1, -24(%ebp)
  movl  $2, -20(%ebp)
#APP
# 10 "test8.c" 1
  movl -24(%ebp), %eax
  addl -20(%ebp), %eax
  movl %eax, -16(%ebp)
# 0 "" 2
#NO_APP
  movl  -16(%ebp), %eax
```


可以看到：在进入内联汇编代码之前，把 data1 和 data2 的值放在了栈中，然后直接把栈中的数据与寄存器 %eax 进行操作，最后再把操作结果(%eax)，复制到栈中 data3 的位置(-16(%ebp))。

## 五、总结

通过以上 8 个示例，我们把内联汇编代码中的关键语法规则进行了讲解，有了这个基础，就可以在内联汇编代码中编写更加复杂的指令了。

希望以上内容对您能有所帮助！谢谢！

文章中的 8 个示例代码，可以在公众号后台回复【426】，即可收到下载地址。

---------- End ----------

让知识流动起来，越分享，越幸运！

星标公众号，能更快找到我！

Hi~你好，我是道哥，一枚嵌入式开发老兵。

推荐阅读

1. C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻
2. 原来gdb的底层调试原理这么简单
3. 一步步分析-如何用C实现面向对象编程
4. 图文分析：如何利用Google的protobuf，来思考、设计、实现自己的RPC框架
5. 都说软件架构要分层、分模块，具体应该怎么做(一)
6. 都说软件架构要分层、分模块，具体应该怎么做(二)