Contents

GNU C 允许在 C 代码中嵌入汇编代码，这种特性被称为内联汇编。使用内联汇编可以同时发挥 C 和汇编的强大能力。

本文介绍 GCC 的内联汇编拓展，Clang 编译器兼容大部分 GCC 语言拓展，因此 GNU C 的内联汇编特性大部分在 Clang 中工作正常。

本文实验环境如下：

```
Linux Friday 5.8.17-300.fc33.x86_64 #1 SMP Thu Oct 29 15:55:40 UTC 2020
    x86_64 x86_64 x86_64 GNU/Linux
gcc (GCC) 10.2.1 20201016 (Red Hat 10.2.1-6)
```

使用 64 位 AT&T 风格 x86 汇编，为了和编译器自动生成的注释区分开，我添加的注释使用##风格。

1 基本内联汇编

基本内联汇编是 GCC 对内联汇编最简陋的支持，它实际上已经没有任何使用价值了，介绍它只是为了说明使用内联汇编的基本原理和问题。

基本内联汇编的语法如下：

```
asm asm_qualifiers ( AssembleInstructions )
```

`asm_qulifiers`包括以下两个修饰符：

- volatile: 指示编译器不要对 asm 代码段进行优化
- inline: 指示编译器尽可能小的假设 asm 指令的大小

这两个修饰符的意义先不用深究，本文会逐步介绍它们的作用。

`asm`不是 ISO C 中的关键字，如果我们开启了 -std=c99 等启用 ISO C 的编译选项，代码将无法成功编译。然而，内联汇编对于许多 ISO C 程序是必须的，GCC 通过 __asm 给程序员开了个后门。使用 **__asm** 替代 asm 可以让程序作为 ISO C 程序成功编译。volatile 和 inline 也有加 __ 的版本。

`AssembleInstructions`是我们手写的汇编指令。基本内联汇编的例子如下：

```
__asm__ __valatile__(
    "movq %rax, %rdi \n\t"
    "movq %rbx, %rsi \n\t"
);
```

编译器不解析 asm 块中的指令，直接把它们插入到生成的汇编代码中，剩下的任务有汇编器完成。这个过程有些类似于宏。为了避免我们手写的汇编代码挤在一起，导致指令解析错误，通常在每一条指令后面都加上`\n\t`获得合适的格式。

编译器不解析 asm 块中的指令的一个推论是：GCC 对我们插入的指令毫不知情。这相当于我们人为地干涉了 GCC 自动的代码生成，如果我们处理不当，很可能导致最终生成的代码是错误的。考虑以下代码段：

```
#include <stdio.h>

int
main()
{
    unsigned long long sum = 0;
    for (size_t i = 1; i <= 10; ++i)
    {
        sum += i;
    }
    printf("sum: %llu\n", sum);
    return 0;
}
---------------------------------------
              output
---------------------------------------
sum: 55
```

这段代码很简单，只是简单的整数求和。反汇编结果如下：

```
    .file   "basic-asm.c"
    .text
    .section    .rodata
.LC0:
    .string "sum: %llu\n"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc                  ## 进入函数
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6         ## 分配局部变量
    subq    $16, %rsp
    movq    $0, -8(%rbp)            ## sum
    movq    $1, -16(%rbp)           ## i
    jmp .L2
.L3:                                ## for body
    movq    -16(%rbp), %rax         ## sum += i
    addq    %rax, -8(%rbp)
    addq    $1, -16(%rbp)           ## ++i
.L2:
    cmpq    $10, -16(%rbp)          ## for 条件判断
    jbe .L3
    movq    -8(%rbp), %rax          ## 传递参数给 printf
    movq    %rax, %rsi              ## x86-64 通常可以使用 6 个寄存器传递
       参数
    movl    $.LC0, %edi             ## 从做往右依次为 %rdi, %rsi, %rdx, %
       rcx, %r8, %r9
    movl    $0, %eax                ## 更多的参数通过堆栈传递
    call    printf
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (GNU) 10.2.1 20201016 (Red Hat 10.2.1-6)"
    .section    .note.GNU-stack,"",@progbits
```

可以看到在 for body 中，变量i被分配到-16(%rbp)中，我们在sum += i前插入这段代码来验证基本内联汇编的处理过程。

```
__asm__ __volatile__(
        "movq $100, -16(%rbp)\n\t"
        );
```

反汇编结果如下：

```
    .file   "basic-asm.c"
    .text
    .section    .rodata
.LC0:
    .string "sum: %llu\n"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movq    $0, -8(%rbp)
    movq    $1, -16(%rbp)
    jmp .L2
.L3:
#APP                        ## 可以看到编译器直接将我们的指令插入到了汇编
    文件中
# 9 "basic-asm.c" 1
    movq $100, -16(%rbp)

# 0 "" 2
#NO_APP
    movq    -16(%rbp), %rax
    addq    %rax, -8(%rbp)
    addq    $1, -16(%rbp)
.L2:
    cmpq    $10, -16(%rbp)
    jbe .L3
    movq    -8(%rbp), %rax
    movq    %rax, %rsi
    movl    $.LC0, %edi
    movl    $0, %eax
    call    printf
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (GNU) 10.2.1 20201016 (Red Hat 10.2.1-6)"
    .section    .note.GNU-stack,"",@progbits
```

我们通过基本内联汇编将变量i的值修改为100,因此程序会直接退出**for**循环，运行结果为：

```
sum = 100
```

基本内联汇编中没有程序员和编译器的交流，程序员不知道编译器将生成怎样的代码，编译器也不知道程序员希望它怎样生成代码为内联汇编的结果几乎不可控制，因此内联汇编没有任何实用价值。

2 拓展内联汇编

从上面基本内联汇编的介绍可以发现，生成正确的代码需要程序员和编译器的通力合作，只有充分的交流才能确保结果的正确。拓展内联汇编很好的实现了程序员和编译器的交流，程序员不再打乱编译器的代码生成，而是提供充分信息来辅助、微调编译器的代码生成。

2.1 基本原理和思路

在编译器生成代码的过程是一个动态的过程，变量可能被分配到寄存器（如 rax）中，也可能被分配到内存中; 一个整型字面值可能是 32 位立即数，也可能是 64 位大立即数; 可能使用 rax 寄存器，也可能使用 rbx 寄存器。程序员任何擅自的篡改都会导致生成错误的代码。

拓展内联汇编从程序员处获取信息，并根据获取的信息调整自己生成代码的行为。比如，程序员要求将某个变量分配到 rax 寄存器中，编译器就会将该变量分配在 rax 中，并调整其他部分的代码，使程序员的要求不影响正确代码的生成。

因此，使用拓展内联汇编的基本思路就是：提供尽可能多的信息给编译器。程序员提供的信息越多，出错的概率就越小。除了提供信息，程序员还应该清楚地明白 GCC 对内联汇编做的假设和限制。

2.2 语法结构

拓展内联汇编的语法结构如下：

```
asm asm-qualifiers ( AssemblerTemplate
                 : OutputOperands
                 [ : InputOperands
                 [ : Clobbers ] ])

asm asm-qualifiers ( AssemblerTemplate
                      :
                      : InputOperands
                      : Clobbers
                      : GotoLabels)
```

`asm`、`asm-qualifiers`和基本内联汇编基本相同。基本内联汇编提供了在汇编中跳转到 C Label 的能力，因此`asm_qualifiers`中增加了 **goto**。goto 修饰符只能用于第二种形式中。

`AssemblerTemplate`是程序员手写的汇编指令，但是增加了几种更方便的表示方法。

可以将拓展内联汇编 asm 块看成一个黑盒，我们给一些变量、表达式作为输入，指定一些变量作为输出，指明我们指令的副作用，运行后这个黑盒会按照我们的要求将结果输出到输出变量中。

`OutputOperands`表示输出变量，`InputOperands`表示输入变量，`Clobbers`表示副作用（asm 块中可能修改的寄存器、内存）等。

拓展内联汇编语法结构比较复杂，没法一下讲清楚，先给出一个例子一览全貌。

```
// 测试 val 的第 bit 位是否为 1
int
bittest(unsigned long long val, unsigned long long bit)
{
    int ret;
    __asm__ (
        "movl $0, %0 \n\t"           // %0 代表 ret（第 0 个输入/输出）
        "btq %2, %1 \n\t"            // %1 代表 val（第 1 个输入/输出），%2
            代表 bit。btq 指令将 val 的第 bit 位存入 CF 中
        "jnc %=f \n\t"                // 若 CF 标记为 1,将 ret 设置为 1
        "movl $1, %0 \n\t"
        "%=: movl $0, %0 \n\t"
        : "=&rm" (ret)               // ret 为输出变量。该变量可以被分配到
            通用寄存器或内存中中。不允许该输出变量与输入重叠。
        : "r" (val), "r" (bit)       // val 和 bit 是输入变量，分配到任意通
            用寄存器中
        : "cc", "memory"             // asm 块可能读取、修改条件寄存器和内
            存
            );
    return ret;
}
```

这个例子使用到了拓展内联汇编的绝大多数功能。

2.3 汇编方言

GCC 支持多种汇编方言，x86 汇编默认使用 AT&T 语法，但也支持 Intel 语法。GCC 生成的汇编指令可以通过编译选项 -masm=dialect 切换。如果使用 Intel 语法，那么 asm 块中的 AT&T 语法就无法正确编译，反之亦然。可以通过`{ dialect 0 | dialect 1 | dialect 2 ... }`来兼容多种方言。这里使用`bt`指令（bit test）来说明使用方法。

```
"bt{l %[Offset],%[Base] | %[Base],%[Offset]}; jc %l2"
```

编译器根据编译选项 -masm 展开后为：

```
"btl %[Offset],%[Base] ; jc %l2"    /* att dialect */
"bt %[Base],%[Offset]; jc %l2"      /* intel dialect */
```

‘%l2 代表 C Label。

2.4 特殊字符串

内联汇编中使用%N表示第N个输入/输出（从 0 开始数），使用{、}和|表示不同方言。AT&T 语法中寄存器要加%前缀，因此%需要被跳脱。拓展内联汇编中，%要写成%%，如%%rax。

GCC 还特别提供了%=生成在所有 asm 块中唯一的数字，这个功能用于生成不重复的 local label 供跳转指令使用。我们最开始的bittest()就使用了这个功能。

介绍到这里，实际上就说完了AssemblerTemplate的全部内容，开始介绍输出列表、输入列表、修改列表的细节。

2.5 输出列表

语法结构如下：

```
[ [asmSymbolicName] ] constraint (cvariablename)
```

- asmSymboicName

  我们可以给cvariablename起一个只能在该 asm 中使用的别名，并通过%[asmSymbolicName]访问它。比如：[value] "=m"(i)，可以在 asm 块中通过%[value]访问它。

- constraint（限制）和 modifier（修饰语）

  constraint 在拓展汇编中至关重要，它和 modifier 是拓展内联汇编和基本内联汇编的根本差异之处。它们的作用都是给编译器提供信息，不同之处在于：constraint 提供输入/输出变量位置的信息（如分配到寄存器还是内存中），modifier 只能用于输出，提供输出变量的行为信息（如只读/读写，是否可以在指令中交换次序）。

- cvariablename

  cviriablename 是一个 C 变量，因为它是 asm 的输出变量，必须要可以被修改，因此必须是左值。

因为 modifier 只能用于输出变量上，因此只先介绍 modifier。

constraint 用来表示输入/输入变量的位置，既有通用的（如任意通用寄存器，不同平台对应不同寄存器），也有特定平台的拓展（如 x86 中的 a，对应寄存器 (r|e)ax），使用时查阅 GCC 手册即可。本文只介绍几个常用的通用、x86、RISC-V constraint。

modifier 有以下四个：

- =: 操作对象是只写的。这意味着操作对象中的值可以被丢弃，并且写入新数据。
- +: 操作对象是读写的。这以为着可以在 asm 中合法的读取操作对象，并且 C 变量在进入 asm 块时就已经加载到对应的位置中。
- &: 指示该操作对象一定不能和输入重叠。

- %: 表示该操作对象可以交换次序。这个 modifer 我不是很理解，似乎没有大的用处。

&比较难以理解，单独解释。GCC 假设汇编代码在产生输出前会消耗掉输入，可能会将不相关的输出/输入分配到同一个寄存器中。实际上输入和输出的次序不一定满足 GCC 的假设，这时就会出错。举两个例子说明这个问题。

细心的读者应该会注意到在`testbit()`函数中，输出`ret`被分配到寄存器或内存中，constraint 中使用了&。假如我们删除掉&会怎么样呢？

```
// 测试程序
// bittest()删除 &
int
main()
{
    if (bittest(1, 0))
        printf("0\n");
    else
        printf("1\n");

    return 0;
}
```

编译后运行结果为 1。这很显然是错的。反汇编`bittest()`，关键部分代码如下：

```
    movq    %rdi, -24(%rbp)     ## 第一个参数（val）
    movq    %rsi, -32(%rbp)     ## 第二个参数（bit）
    movq    -24(%rbp), %rax     ## 变量 val 分配到 rax 中
    movq    -32(%rbp), %rdx
#APP
# 23 "bt.c" 1
    movl $0, %eax               ## 变量 ret 也被分配到 rax 中
    btq %rdx, %rax
    jnc 15f                     ## 15 是 %= 生成的
    movl $1, %eax
    15:

# 0 "" 2
#NO_APP
```

可以发现，错误的根源在于我们指示 GCC 将变量`ret`和`val`分配到通用寄存器中，GCC 假设输入在产生输出前就被消耗（输入/输入分配到同一个寄存器中不会出错），因此将`ret`和`val`都分配到了寄存器 rax 中。在执行 bt 指令前，我们将返回值`ret`设置成`0`，覆盖了`val`，导致错误。

还有一种可能的输入/输出重叠的情况：输出 A 被分配到寄存器中，输出 B 被分配到内存中，访问内存 B 时错误地使用了输出 A 被分配到的寄存器。访问内存中的 B 很可能需要使用到寄存器（如内存寻址），GCC 这时将访问 B 过程中使用到的寄存器视为输入，根据“输入在产生输出之前就被消耗掉了”的假设，GCC 很可能会在访问 B 的过程中使用 A 对应的寄存器（假设在访问完 B 后才写入 A, 这时情况正常）。实际情况可能不符合 GCC 的假设，用户可能在访问 B 之前写入 A, 在访问 B 时使用的寄存器中的值（这个值错误地变成了 A 的值）可能是错误的。

陷阱：

- GCC 不保证在进入 asm 块时，输出变量已被加载到 constraint 指定的位置中。如果需要 GCC 在进入 asm 块时将变量加载到 constraint 指定的位置中，请使用+。
- constraint 是指定变量在 asm 块中的位置，而不是在函数中的位置。变量`val`的 constraint 为 r 说明它在进入/退出 asm 块时被分配到通用寄存器中，但在进入 asm 块前它的位置是不确定的。如果要控制 asm 块外变量被分配的位置，可以使用 GNU C 的寄存器变量拓展。

2.6 输入列表

语法结构如下：

```
[ [asmSymbolicName] ] constraint (cexpression)
```

`asmSymbolicName`和`constraint`和输出列表一样。

输入列表中不可以使用=和+这两个 constraint。

因为输入是只读的，因此不要求输入是左值，任何 C 表达式都可以作为输入。

GCC 假设输入是只读的，在退出 **asm** 块时输入的值不被改变。我们不能通过修改列表来告知 GCC 我们将修改一个输入。如果我们确实需要修改输入，有两种办法：

- 使用可读写的输出替换输入。
- 将输入绑定到一个不使用的输出上。

第一种方法的原理显而易见，加上+限制的输出在进入 asm 块时就被分配到对应的位置中，除了可以写外，跟输入变量没有区别。

第二种方法是变通方法，当我们将输入绑定（放入同一个位置）到一个不使用的输出时，我们修改输入就相当于生成输出，绕开了 GCC 不修改输入的规定。使用这种方法要小心 GCC 发现输出变量未使用，将 asm 优化掉，需要添加 volatile 修饰符。

我个人建议使用第一种方法，虽然第一种方法在语意上不太合适，但能够实现我们的目的，并且比较好理解。

2.7 修改列表

在使用内联汇编时，我们写的汇编代码可能会产生一些副作用，GCC 必须清楚地知道这些副作用才能调整自己的行为，生成正确的代码。

举一个可能导致生成错误代码的例子。我们使用字符串复制指令`movsb`将一段内存复制到另一个地址，`movsb`会读取、修改寄存器 rsi 和 rdi 的值，如果我们不告诉 GCC 我们写的汇编代码有“修改 rsi 和 rdi”的副作用，GCC 会认为 rsi 和 rdi 没有被修改，生成错误的代码。

在使用内联汇编时我们必须提供给 GCC 尽可能多的信息，汇编代码可能有哪些副作用（修改了哪些寄存器，是否访问内存）是使用内联汇编时需要始终考虑的问题。

修改列表 (*Clobeerrs*) 的语法结构如下：

```
: "Clobber" (cexpression)
```

`Clobber`有以下几个：

- `cc`: 条件（标准）寄存器。如 x86 的 EFLAGS 寄存器。
- `memory`: 读/写内存。为了确保读取到正确的值，GCC 可能会在进入 asm 块前将某些寄存器写入内存中，也可能在必要的时候将内存中存储的寄存器值重新加载到寄存器中。
- 寄存器名：如 x86 平台的 rax 等，直接写原名即可。

2.8 constraint

这里介绍几个常用的 constraint：

- `r`: 通用寄存器
- `i`: 在汇编时或运行时可以确定的立即数
- `n`: 可以直接确定的立即数，如整形字面量
- `g`: 任意通用寄存器、内存、立即数

这些是 GCC 提供的通用 constraint，在不同处理器上有不同的实现。比如 x86 上的通用寄存器是 rax、r8 等，在 RISC-V 上是 x0 到 x31。

有些指令，如 x86 常用的`mov`指令，两个操作数既可以都是寄存器、也可以一个是寄存器一个是内存地址。这时就有三种组合，我们可以将 constraint 可以分为多个候选组合传递给 GCC, 如：

```
:   "m,r,r" (output)
:   "r,m,r" (input)
```

constraint 通过，分组，并且一一对应。上面的代码段相当于以下三个输出/输入列表组合在一起：

```
:   "m" (output)
:   "r" (input)
------------------
:   "r" (output)
:   "r" (input)
------------------
:   "r" (output)
:   "m" (input)
```

一个输入/输出可以有多个 constraint，GCC 会自动选择其中最好的一个。如：`"rm"(output)`表示`output`既可以分配到通用寄存器中，也可以分配到内存中，由 GCC 自己选择。

多 constraint 和分组的 constraint 是两码事。还拿 x86 上的`mov`指令举例，`mov`指令不允许两个操作数都是内存地址，因此我们不能写出这样的输出/输入列表：

```
:   "rm" (output)
:   "rm" (input)
```

这个列表表示`output`和`input`都可以分配到内存或通用寄存器中，可能出现两变量同时被分配到内存中的情况，这是 mov 指令就会出错。

2.9 goto 列表

GCC 提供了在内联汇编中使用 C Label 的功能，但这个功能有限制，这能在 asm 块没有输出时使用。C Label 在内联汇编中直接当成汇编的 label 使用即可，唯一要注意的是在内联汇编中 C label 的命名。

在内联汇编中使用`%lN`来访问 C label, 因为`%l`在内联汇编中已经有了特殊的意义（x86 平台的修饰符，表示寄存器的低位子寄存器，如 rax 中的 eax），因此 GCC 将 C label 对应的`N`设置为输入输出总数加 goto 列表中 C label 的位置。

```
asm goto (
    "btl %1, %0\n\t"
    "jc %l2"
    : /* No outputs. */
    : "r" (p1), "r" (p2)
    : "cc"
    : carry);

return 0;

carry:
return 1;
```

标签`carry`之前有两个输入，`carry`在 goto 列表的第 0 位，因此使用`%l2`引用`carry`。

3 杂项

3.1 标记寄存器的使用

在某些平台，比如 x86, 存在标记寄存器。GCC 允许将标记寄存器中的某个标准输出到 C 变量中，这个变量必须是标量（整形、指针等）或者是布尔类型。当 GCC 支持这个特性时，会与定义宏`__GCC_ASM_FLAG_OUTPUTS__`。

标记输出约束为`@cccond`，其中`cond`为指令集定义的标准条件, 在 x86 平台上即条件跳转的后缀。

因为访问的是标记寄存器中的标记（很可能是一个比特），因此不能在 asm 块中通过%0等形式显示访问，也不能给多个约束。

使用标记寄存器，可以简化前面的testbit()：

```
int
bittest(unsigned long long val, unsigned long long bit)
{
    int ret;
    __asm__ __volatile__(
        "btq %2, %1 \n\t"
        : "=@ccc" (ret)
        : "r" (val), "r" (bit)
        : "cc", "memory"
            );
    return ret;
}
```

3.2 asm 的大小

为了生成正确的代码，一些平台需要 GCC 跟踪每一条指令的长度。但是内联汇编由编译器完成，指令的长度却只有汇编器知道。

GCC 使用比较保守的办法，假设每一条指令的长度都是该平台支持的最长指令长度。asm 块中所有语句分割符（如;等）和换行符都作为指令结束的标准。

通常，这种办法都是正确的，但在遇到汇编器的伪指令和宏时可能会产生意想不到的错误。汇编器的伪指令和宏最终会被汇编器转换为多条指令，但是在编译器眼中它只是一条指令，从而产生误判，生成错误的代码。

因此，尽量不要在 **asm** 块中使用伪指令和宏。

3.3 X86 特定

x86 平台有些些专门的 constraint , 如：

- a: ax 寄存器，在 32 位处理器上是 eax, 在 64 位处理器上是 rax
- b(bx),c(cx),d(dx),S(si),D(di): 类似与a
- q: 整数寄存器。32 位上是a、b、c、d，64 位增加了 r8 ~ r15 8 个寄存器

x86 中一个大寄存器可以重叠地分为多个小寄存器，比如 rax 第 32 位可以作为 eax 单独使用，eax 低 16 位又可以作为 ax 单独使用，ax 高 8 位可以做为 ah 单独使用、低 8 位可以作为 al 单独使用。针对这种情况，GCC 在 x86 平台专门提供了一些修饰符来调整生成的汇编代码中寄存器、内存地址等的格式。

```
uint16_t  num;
asm volatile ("xchg %h0, %b0" : "+a" (num) );
```

这段代码将num分配到 ax 寄存器中，在 64 位处理器上是 rax，32 位处理器上是 eax，但程序员只需要访问它的子寄存器 ah 和 al。h表示访问 ah（bh、ch 等），b表示访问 al（bl、cl 等）。因此内联汇编指令插入到汇编代码中时变成了xchg ah, al，而不是原始的xchg rax, rax或xchg eax, eax。

完整的 GCC x86 修饰符可以在手册中找到。

3.4 RISC-V 特定

GCC 对 RISC-V 平台提供了以下额外的 constrait。

- f: 浮点寄存器（如果存在的花）
- I: 12 比特立即数
- J: 整数 0
- K: 用于 CSR 访问指令的 5 比特的无符号立即数
- A: 存储在通用寄存器中的地址

GCC 没有提供对 RISC-V 特定寄存器的 constrait，如果我们需要将变量分配到特定的寄存器，只能通过分配寄存器变量的方式曲线救国。

3.5 寄存器变量

寄存器变量是 ISO C 的特性，语法为:

```
register type cvariable
```

如：

```
register size_t i;
for (i = 0; i < 100; ++i)
    /* do something */
```

ISO C 中的寄存器变量特性只是“建议”将某个变量分配到寄存器中，最终是否分配到寄存器中由编译器决定，并且没有提供指定寄存器的语法，分配到哪个寄存器也由编译器决定。

GCC 拓展了 ISO C 中寄存器变量的特性，提供了指定寄存器的语法，只要分配的寄存器合法就会分配成功。

语法结构如下：

```
register type cvariable asm ("register")
```

如：

```
register unsigned long long i asm ("rax");  // x86
```

该代码段将变量i分配到寄存器 rax 中。

因为变量在寄存器中，因此寄存器变量的使用有以下限制：

- 全局寄存器变量不能有初始值不能初始化。可执行文件无法给寄存器提供初值。
- 不能使用 volatile 等修饰符。
- 不能取地址。

寄存器变量仅仅是指示编译器将变量放置在特定的寄存器中，不意味这在该变量的整个生命周期中该变量都独占该寄存器，该寄存器很可能会被分配为别的变量使用。程序员只可以假设在声明时变量在指定的寄存器中，之后的语句中不能假设该变量仍在该寄存器中，生成的任何指令都可能修改该寄存器的值。

寄存器变量既可以全局变量也可以是局部变量。由于上面提到的限制，将全局变量声明为寄存器变量几乎总是错误的做法，很可能破坏 C ABI, 对性能也未必有大的提升，仅在极有限的场景下使用全局寄存器变量，因此不解释全局寄存器变量。

当 GCC 没有提供将变量分配到特定寄存器中的 constraint 时，我们将该变量声明为局部寄存器变量，并将其分配到特定的寄存器中。然后紧贴着写内联汇编，分配到寄存器中就使用r constrait。

以下代码封装了 RISC-V ecall。

```
void
sbi_console_putchar(int ch)
{
    register int a0 asm ("x10") = ch;           // 变量 a0 分配到寄存器
       x10 中
    register uint64_t a6 asm ("x16") = 0;       // 变量 a6 分配到寄存器
       x16 中
    register uint64_t a7 asm ("x17") = 1;       // 变量 a7 分配到寄存器
       x17 中
    __asm__ __volatile__ (
        "ecall \n\t"
        : /* empty output list */
        : "r" (a0), "r" (a6), "r" (a7)
        : "memory"
        );
}
```

陷阱:

- 小心在定义了寄存器变量后，使用寄存器变量前，某些语句修改的寄存器的值
- 局部寄存器变量只能配合内联汇编使用，或者按照标准 C ABI 在函数之间传递。其他所有用法都是未定义的，工作正常仅仅是运气。

4 总结

准则：

- 尽可能不要使用宏和伪指令
- 使用= constraint 时不要假设在进入 asm 块时，变量已被分配到寄存器中
- 不要修改输入变量，除非它和输出相关联
- 尽可能考虑全面，尽可能提供多的信息
- 小心输出输入重叠，使用&constraint 解决这个问题
- 较宽泛的 constraint 可以给 GCC 更大自由，生成更好的代码，但程序员要考虑的事情也变多了
- 小心打字错误，如将`$1`打成1, 这可能导致段错误
- 小写指令的操作对象类型错误，这可能导致段错误。如 x86 的`cmov`指令要求源是寄存器或内存位置，目的操作对象是寄存器。

5 参考

- Machine Modes: GCC 中的机器模式概念，和 x86 修饰符有关。
- Using the GNU Compiler Collection (GCC): GCC 官方文档。
- New asm flags feature for x86 in GCC 6: 解释了 GCC 6 引入的在内联汇编中以标记寄存器为输出的的新功能