南　　开　　大　　学

网络空间安全学院

预备工作 1 实验报告

了解编译器及 LLVM IR 编程

2011696 付寅聪

年级：2020 级

专业：物联网工程

指导教师：王刚

2022 年 9 月 27 日

摘要

以 GCC、LLVM 等为研究对象，深入地探究语言处理系统的完整工作过程

关键字：gcc,llvml

目录

一、 实验流程

(一) 完整编译过程

如图1所示

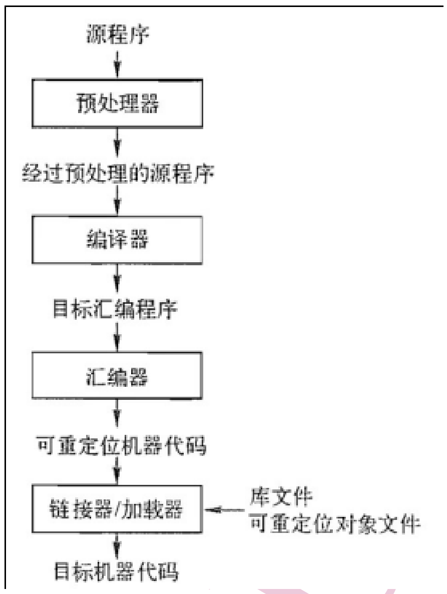


图 1: Caption

1. 源程序：即源语言所写程序，也就是我们平时用高级语言写的代码，如 C、C++、Python 等。

2. 预处理器：预处理也叫预编译，主要用于执行预编译命令，以 C++ 为例，预编译的操作包括：将所有的 define 删除，并且展开所有的宏定义；处理所有的条件预编译指令，如 #if、#ifdef；处理 #include 预编译指令，将被包含的文件插入到该预编译指令的位置；过滤所有的注；添加行号和文件名标识。

3. 编译器：编译器的核心功能是把源代码翻译成目标代码，也是本文介绍的主要内容，主要包括词法分析、语法分析、语义分析、代码优化、目标代码生成、目标代码优化等。

4. 汇编器：汇编器会对由编译器产生的汇编语言处理，生成可重定位的机器码。

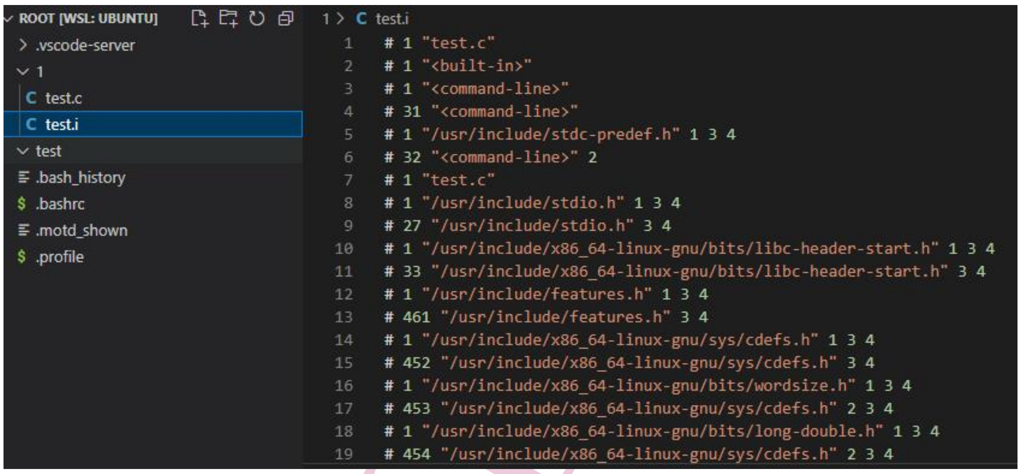
5. 链接器/加载器：将不同的源文件产生的目标文件进行链接，从而形成一个可以执行的程序。链接分为静态链接和动态链接。

(二) 预处理器

代码

斐波那契数列

```
#include<stdio.h>
#define NUM 10
// fibonacci function
int main()
{
    int a, b, i, t, n = NUM;
    a = 0;
    b = 1;
    i = 1;
    printf("a:%d\nb:%d\n", a, b);
    while (i < n)
    {
        t = b;
        b = a + b;
        printf("b:%d\n", b);
        a = t;
        i = i + 1;
    }
}
```



```
# 1 "test.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "test.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
# 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 461 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 452 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 453 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
# 454 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
```

通过 gcc test.c -E -o test.i 预处理, 生成预编译文件, 预编译就是将要包含 (include) 的文件插入源文件中, 将宏定义展开, 根据条件编译命令选择要使用的代码, 最后将这些代码输出到一个".i" 文件中等待进一步出来.

预编译过程主要处理那些源代码文件中以"#" 开始的预编译指令。比如"#include"、"#define" 等. 经过预编译后的.i 文件不包含任何宏定义, 因为所有的宏已经被展开, 并且包含的文件也已经被插入到.i 文件中。所以当我们无法判断宏定义是否正确或头文件包含是否正确的时候, 可以查看预编译后的文件来确定问题。

1.include<stdio.h> 表示在寻找头文件时, 直接去头文件目录中寻找–/usr/include/

2. 去掉注释

3. 宏替换用 10 替换 NUM

(三) 编译器

编译过程就是把预处理完的文件进行一系列词法分析、语法分析、语义分析及优化后生成相应的汇编代码文件, 这个过程是整个程序构建的核心部分, 也是最复杂的部分之一。编译过程一般分为 6 个步骤: 扫描、语法分析、语义分析、源代码优化、代码生成和目标代码优化.

1. 词法分析: 源代码程序被输入到扫描器 (Scanner), 扫描器对源代码进行简单的词法分析, 运用类似于有限状态机 (Finite State Machine) 的算法可以很轻松的将源代码字符序列分割成一系列的记号 (Token)

clang -E -Xclang -dump-tokens test.c

token 序列

```
s/cdefs.h:55:54>>
__attribute '__attribute__'        [LeadingSpace]  Loc=</usr/include/stdio.h:446:14>
l_paren '('        [LeadingSpace]  Loc=</usr/include/stdio.h:446:28>
l_paren '('                Loc=</usr/include/stdio.h:446:29>
identifier '__format__'            Loc=</usr/include/stdio.h:446:30>
l_paren '('        [LeadingSpace]  Loc=</usr/include/stdio.h:446:41>
identifier '__scanf__'             Loc=</usr/include/stdio.h:446:42>
comma ','                  Loc=</usr/include/stdio.h:446:51>
numeric_constant '2'       [LeadingSpace]  Loc=</usr/include/stdio.h:446:53>
comma ','                  Loc=</usr/include/stdio.h:446:54>
numeric_constant '0'       [LeadingSpace]  Loc=</usr/include/stdio.h:446:56>
r_paren ')'                Loc=</usr/include/stdio.h:446:57>
r_paren ')'                Loc=</usr/include/stdio.h:446:58>
r_paren ')'                Loc=</usr/include/stdio.h:446:59>
semi ';'                   Loc=</usr/include/stdio.h:446:60>
extern 'extern'    [StartOfLine]   Loc=</usr/include/stdio.h:451:1>
int 'int'          [LeadingSpace]  Loc=</usr/include/stdio.h:451:8>
identifier 'vfscanf'       [LeadingSpace]  Loc=</usr/include/stdio.h:451:12 <Spelling=/usr/include/stdio.h:451:24>>
l_paren '('        [LeadingSpace]  Loc=</usr/include/stdio.h:451:12 <Spelling=/usr/include/stdio.h:452:10>>
identifier 'FILE'                  Loc=</usr/include/stdio.h:451:12 <Spelling=/usr/include/stdio.h:452:11>>
star '*'           [LeadingSpace]  Loc=</usr/include/stdio.h:451:12 <Spelling=/usr/include/stdio.h:452:16>>
restrict '__restrict'              Loc=</usr/include/stdio.h:451:12 <Spelling=/usr/include/stdio.h:452:17>>
identifier '__s'           [LeadingSpace]  Loc=</usr/include/stdio.h:451:12 <Spelling=/usr/include/stdio.h:452:28>>
comma ','                  Loc=</usr/include/stdio.h:451:12 <Spelling=/usr/include/stdio.h:452:31>>
const 'const'      [StartOfLine]  [LeadingSpace]     Loc=</usr/include/stdio.h:451:12 <Spelling=/usr/include/stdio.h:453:4>>
char 'char'        [LeadingSpace]  Loc=</usr/include/stdio.h:451:12 <Spelling=/usr/include/stdio.h:453:10>>
star '*'           [LeadingSpace]  Loc=</usr/include/stdio.h:451:12 <Spelling=/usr/include/stdio.h:453:15>>
restrict '__restrict'              Loc=</usr/include/stdio.h:451:12 <Spelling=/usr/include/stdio.h:453:16>>
identifier '__format'      [LeadingSpace]  Loc=</usr/include/stdio.h:451:12 <Spelling=/usr/include/stdio.h:453:27>>
comma ','                  Loc=</usr/include/stdio.h:451:12 <Spelling=/usr/include/stdio.h:453:35>>
identifier '__gnuc_va_list'        [LeadingSpace]  Loc=</usr/include/stdio.h:451:12 <Spelling=/usr/include/stdio.h:453:37>>
identifier '__arg'         [LeadingSpace]  Loc=</usr/include/stdio.h:451:12 <
```

```
    Spelling=/usr/include/stdio.h:453:52>>
r_paren ')'            Loc=</usr/include/stdio.h:451:12 <Spelling=/usr/
    include/stdio.h:453:57>>
asm '__asm__'     [LeadingSpace] Loc=</usr/include/stdio.h:451:12 <Spelling=/
    usr/include/x86_64-linux-gnu/sys/cdefs.h:174:52>>
l_paren '('       [LeadingSpace] Loc=</usr/include/stdio.h:451:12 <Spelling=/
    usr/include/x86_64-linux-gnu/sys/cdefs.h:174:60>>
string_literal '""'              Loc=</usr/include/stdio.h:451:12 <Spelling=<
    scratch space>:21:1>>
......
```

2. 语法分析: 将词法分析生成的词法单元来构建抽象语法树

clang -E -Xclang -ast-dump test.c

AST 输出

```
s/cdefs.h:55:54>>
__attribute '__attribute__'      [LeadingSpace] Loc=</usr/include/stdio.h
    :446:14>
l_paren '('       [LeadingSpace] Loc=</usr/include/stdio.h:446:28>
l_paren '('            Loc=</usr/include/stdio.h:446:29>
identifier '__format__'          Loc=</usr/include/stdio.h:446:30>
l_paren '('       [LeadingSpace] Loc=</usr/include/stdio.h:446:41>
identifier '__scanf__'           Loc=</usr/include/stdio.h:446:42>
comma ','              Loc=</usr/include/stdio.h:446:51>
numeric_constant '2'      [LeadingSpace] Loc=</usr/include/stdio.h:446:53>
comma ','              Loc=</usr/include/stdio.h:446:54>
numeric_constant '0'      [LeadingSpace] Loc=</usr/include/stdio.h:446:56>
r_paren ')'            Loc=</usr/include/stdio.h:446:57>
r_paren ')'            Loc=</usr/include/stdio.h:446:58>
r_paren ')'            Loc=</usr/include/stdio.h:446:59>
semi ';'               Loc=</usr/include/stdio.h:446:60>
extern 'extern'   [StartOfLine]  Loc=</usr/include/stdio.h:451:1>
int 'int'         [LeadingSpace] Loc=</usr/include/stdio.h:451:8>
identifier 'vfscanf'      [LeadingSpace] Loc=</usr/include/stdio.h:451:12 <
    Spelling=/usr/include/stdio.h:451:24>>
l_paren '('       [LeadingSpace] Loc=</usr/include/stdio.h:451:12 <Spelling=/
    usr/include/stdio.h:452:10>>
identifier 'FILE'                Loc=</usr/include/stdio.h:451:12 <Spelling=/
    usr/include/stdio.h:452:11>>
star '*'          [LeadingSpace] Loc=</usr/include/stdio.h:451:12 <Spelling=/
    usr/include/stdio.h:452:16>>
restrict '__restrict'            Loc=</usr/include/stdio.h:451:12 <Spelling=/
    usr/include/stdio.h:452:17>>
identifier '__s'          [LeadingSpace] Loc=</usr/include/stdio.h:451:12 <
    Spelling=/usr/include/stdio.h:452:28>>
comma ','              Loc=</usr/include/stdio.h:451:12 <Spelling=/usr/
    include/stdio.h:452:31>>
const 'const'     [StartOfLine] [LeadingSpace]    Loc=</usr/include/stdio.h
    :451:12 <Spelling=/usr/include/stdio.h:453:4>>
```

```
char 'char'        [LeadingSpace] Loc=</usr/include/stdio.h:451:12 <Spelling=/
    usr/include/stdio.h:453:10>>
star '*'           [LeadingSpace] Loc=</usr/include/stdio.h:451:12 <Spelling=/
    usr/include/stdio.h:453:15>>
restrict '__restrict'            Loc=</usr/include/stdio.h:451:12 <Spelling=/
    usr/include/stdio.h:453:16>>
identifier '__format'     [LeadingSpace] Loc=</usr/include/stdio.h:451:12 <
    Spelling=/usr/include/stdio.h:453:27>>
comma ','                  Loc=</usr/include/stdio.h:451:12 <Spelling=/usr/
    include/stdio.h:453:35>>
identifier '__gnuc_va_list'      [LeadingSpace] Loc=</usr/include/stdio.h
    :451:12 <Spelling=/usr/include/stdio.h:453:37>>
identifier '__arg'        [LeadingSpace] Loc=</usr/include/stdio.h:451:12 <
    Spelling=/usr/include/stdio.h:453:52>>
r_paren ')'                Loc=</usr/include/stdio.h:451:12 <Spelling=/usr/
    include/stdio.h:453:57>>
asm '__asm__'      [LeadingSpace] Loc=</usr/include/stdio.h:451:12 <Spelling=/
    usr/include/x86_64-linux-gnu/sys/cdefs.h:174:52>>
l_paren '('        [LeadingSpace] Loc=</usr/include/stdio.h:451:12 <Spelling=/
    usr/include/x86_64-linux-gnu/sys/cdefs.h:174:60>>
string_literal '""'              Loc=</usr/include/stdio.h:451:12 <Spelling=<
    scratch space>:21:1>>
......
```

3. 语义分析使用语法树和符号表中信息来检查源程序是否与语言定义语义一致，进行类型检查等

4. 中间代码生成: 生成一个明确的低级或类机器语言的中间表示

clang -S -emit-llvm test.c

LLVM IR

```
; ModuleID = 'test.c'
source_filename = "test.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
    :16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [11 x i8] c"a:%d\0Ab:%d\0A\00", align 1
@.str.1 = private unnamed_addr constant [6 x i8] c"b:%d\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 10, i32* %6, align 4
  store i32 0, i32* %2, align 4
  store i32 1, i32* %3, align 4
  store i32 1, i32* %4, align 4
  %7 = load i32, i32* %2, align 4
  %8 = load i32, i32* %3, align 4
  %9 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([11 x i8], [11
      x i8]* @.str, i64 0, i64 0), i32 %7, i32 %8)
  br label %10

10:                                               ; preds = %14, %0
  %11 = load i32, i32* %4, align 4
  %12 = load i32, i32* %6, align 4
  %13 = icmp slt i32 %11, %12
  br i1 %13, label %14, label %24

14:                                               ; preds = %10
  %15 = load i32, i32* %3, align 4
  store i32 %15, i32* %5, align 4
  %16 = load i32, i32* %2, align 4
  %17 = load i32, i32* %3, align 4
  %18 = add nsw i32 %16, %17
  store i32 %18, i32* %3, align 4
  %19 = load i32, i32* %3, align 4
  %20 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([6 x i8], [6
      x i8]* @.str.1, i64 0, i64 0), i32 %19)
  %21 = load i32, i32* %5, align 4
  store i32 %21, i32* %2, align 4
  %22 = load i32, i32* %4, align 4
  %23 = add nsw i32 %22, 1
  store i32 %23, i32* %4, align 4
  br label %10

24:                                               ; preds = %10
  %25 = load i32, i32* %1, align 4
  ret i32 %25
}

declare dso_local i32 @printf(i8*, ...) #1

attributes #0 = { noinline nounwind optnone uwtable "correctly-rounded-divide
    -sqrt-fp-math"="false" "disable-tail-calls"="false" "frame-pointer"="all"
     "less-precise-fpmad"="false" "min-legal-vector-width"="0" "no-infs-fp-
    math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-
    signed-zeros-fp-math"="false" "no-trapping-math"="false" "stack-protector
    -buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+
    mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #1 = { "correctly-rounded-divide-sqrt-fp-math"="false" "disable-
```

```
    tail-calls"="false" "frame-pointer"="all" "less-precise-fpmad"="false" "
    no-infs-fp-math"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-
    math"="false" "no-trapping-math"="false" "stack-protector-buffer-size"="8
    " "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+
    x87" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 10.0.0-4ubuntu1 "}
```

5. 代码优化: 进行与机器无关的代码优化步骤改进中间代码, 生成更好的目标代码.
llc -print-before-all -print-after-all a.ll > a.log 2>1

LLVM IR

```
llc: error: llc: a.ll: error: Could not open input file: No such file or
    directory
```

6. 代码生成: 以中间表示形式作为输入, 将其映射到目标语言 gcc main.i -S -o test.S

x86 格式目标代码

```
        .file   "test.c"
        .text
        .section        .rodata
.LC0:
        .string "a:%d\nb:%d\n"
.LC1:
        .string "b:%d\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $32, %rsp
        movl    $10, -8(%rbp)
        movl    $0, -20(%rbp)
        movl    $1, -16(%rbp)
        movl    $1, -12(%rbp)
        movl    -16(%rbp), %edx
        movl    -20(%rbp), %eax
        movl    %eax, %esi
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        jmp     .L2
.L3:
        movl    -16(%rbp), %eax
        movl    %eax, -4(%rbp)
        movl    -20(%rbp), %eax
        addl    %eax, -16(%rbp)
        movl    -16(%rbp), %eax
        movl    %eax, %esi
        leaq    .LC1(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        movl    -4(%rbp), %eax
        movl    %eax, -20(%rbp)
        addl    $1, -12(%rbp)
.L2:
        movl    -12(%rbp), %eax
        cmpl    -8(%rbp), %eax
        jl      .L3
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
        .section        .note.GNU-stack,"",@progbits
        .section        .note.gnu.property,"a"
        .align 8
        .long    1f - 0f
        .long    4f - 1f
        .long    5
0:
        .string  "GNU"
1:
        .align 8
        .long    0xc0000002
        .long    3f - 2f
2:
        .long    0x3
3:
        .align 8
4:
```

（四） 汇编器

汇编器会对由编译器产生的汇编语言处理，生成可重定位的机器码

gcc test.s -c -o test.o 生成目标文件

（五） 链接器

链接器，将不同的源文件产生的目标文件进行链接，从而形成一个可以执行的程序。链接分为静态链接和动态链接。

gcc test.o -o test

（六） LLVM IR 编程

我负责编写代码体现的 SysY 特性有表达式, 注释, 输入输出

源程序

```
#include<stdio.h>
int main()
{
    int a=1;
    int b=1;
    int c=a+b;
    int d=a==b;
    int e=0&&1||1;
    printf("%d\n",c);
    printf("%d\n",d);
    printf("%d\n",e);
    }
}

#include<stido.h>
int main(){
    int a=1;
    if(a==1){
        printf("%d",a};
    }
    else{
        printf("%d".0);
    }
    while(a<5)
    {
        a+=1;
        printf("%d",1);
    }
}
```

LLVM IR

```
define dso_local i32 @main() #0 {          ;define开头 ,i32为函数返回类型,main是
    函数名
   %1 = alloca i32, align 4                    ;为5个变量分配内存
   %2 = alloca i32, align 4
   %3 = alloca i32, align 4
   %4 = alloca i32, align 4
   %5 = alloca i32, align 4
   store i32 1, i32*%1, align 4                ;为%1,%2两个变量赋值1
   store i32 1, i32*%2, align 4
   %6 = load i32, i32* %1,  align 4         ;获取%1,%2两个变量的值
   %7 = load i32, i32*%2,align 4
   %8 = add nsw i32 %6, %7                  ;为第三个变量赋值前两个变量之和
   store i32 %8,i32*%3,align 4
   %9 = load i32, i32*%1, align 4           ;获取%1,%2的值
   %10 = load i32, i32* %2,align 4
   %11 = icmp eq i32 %9, %10                ;比较两指
   %12 = zext i1 %11 to i32                     ;将%11扩展为32位
   store i32 %12, i32*%4, align 4
   store i32 1, i32*%5, align 4
   %13 = load i32, i32*%3, align 4
   %14 = call i32(i8*, ...) @printf(i8*getelementptr inbounds ([4 x i8], [4 x
        i8]*@.str, i64 0, i64 O), i32 %13)          ;输出
   %16 = call i32(i8*, ...)@printf(i8*getelementptr inbounds([4 x i8], [4 x
       i8]*@.str,  164 0, i64 0)。132 815)
   %17 = load i32, i32*%5,align 4
   %18 = call i32(i8*, ...)@printf(i8*getelementptr inbounds ([4 x i8],[4 x
       i8]*@.str, i64
   ret i32 0
}



define dso_local i32 @main() #0{
   %1 = alloca i32,  align 4
   %2 = alloca i32,  align 4
   store i32 0,  i32* %1,align 4
   store i32 1, i32*%2, align 4
   %3 = load i32, i32*%2, align 4
   %4 = icmp eq i32 %3, 1
5:              ; preds = %0
   %6 = load i32, i32*%2,align 4
   %7 = call i32(i8*, ...)Cprintf(i8*getelementptr inbounds ([3 x i8],[3 x i8
       ]*@.str, i64 0, i64 0), i32 %6)
   br label %10

8:                        ; preds = %0
  %9= call i32(i8*, ...)@printf(i8*getelementptr inbounds([3 x i8],[3 x i8
       ]*0.str, i64 0, i64 0),i32 0)
   br label %10

10:                 ; preds = %8, %5
   br label %11                                ;br 是无条件分支, label 可
       以理解为一个代码标签, 指代下面那个代码块

11:               ; preds = %14, %10
   %12 = load i32, i32*%2, align 4
   %13 = icmp slt i32 %12, 5
   br i1 %13, label %14, label %18         ;br 是有条件分支, 它根据 i1 和两个
       label 的值, 用于将控制流传输到当前函数中
的不同基本块。


14:                     ; preds = %11
   %15 = load i32, i32*%2, align 4
   %16 = add nsw i32 %15, 1
   store i32 %16,i32*%2,align 4
   %17 = call i32(i8*, ...)@printf(i8* getelementptr inbounds ([3 x i8],[3 x
       i8]*@.str, i64 e, i64 0),i32 1)
   br label %11

18:                 ; preds = %11
   %19 = load i32,i32*%1,align 4
   ret i32 %19
}
```

使用 clang 命令可将其编译位可执行程序

```
clang test.ll -o test
```

二、 总结

编译器的核心功能是把源代码翻译成目标代码，这里的目标机器代码并不一定是机器码：如果要将源语言编译成汇编语言，这里的目标语言就是汇编语言；如果打算直接编译成机器码，也就是跳过汇编器，那这里的目标语言就是机器码。编译器一个很重要的任务就是报告他在编译的过程中发现的源程序中的错误