编译原理预备工作：了解编译器

1811398 鲁含章

摘要

一个现代编译器的主要工作流程有：源代码 (source code) → 预处理器 (preprocessor) → 编译器 (compiler) → 目标代码 (object code) → 链接器 (Linker) → 可执行程序 (executable)。本文中，通过几个 c 和 c++ 源程序的编译过程，展示了编译器工作的各个阶段的功能以及代码优化的一些特性，帮助笔者更好地认识了编译器，为后序工作做好准备。

关键字： G++,GCC, 预处理器, 编译器, 汇编器, 链接器

1 引言

1.1 实验环境

- 平台: `ubuntu18.04.1`
- 编译器: `g++`

1.2 源程序选择与设计

斐波那契数列 fib.cpp. 为了解编译器的基本功能，首先使用由作业要求里提供的斐波那契数列程序作为测个试源程序，作一个小修改，将计算数列结果的程序段封装为函数，观察测试结果。

斐波那契公式:

$$fib(n+2) = fib(n+1) + fib(n)$$

```cpp
#include <iostream>
using namespace std;

void fib(int n){
    int  a=0;
    int  b=1;
    int  i=1;
    cout<<a<<endl;
    cout<<b<<endl;
    while(i<n){
        int  t=b;
        b=a+b;
        cout<<b<<endl;
        a=t;
        i=i+1;
```

```
    }
}

int main(){
    int n;
    cin>>n;
    fib(n);
    return 0;
}
```

优化功能测试 test.c. 撰写另外一个程序，此程序主要观察编译器的优化功能，见代码优化章节。

2 预处理器

预处理器是在真正的编译开始之前由编译器调用的独立程序（在 gcc 中是 CPreProcessing，简称 CPP）。预处理器的功能有：

- 有条件编译源文件的某些部分（由 #if、#ifdef、#ifndef、#else、#elif 和 #endif 指令控制）。
- 替换文本宏，同时可能对标识符进行拼接或加引号（由 #define 和 #undef 指令与 # 和 ## 运算符控制）。
- 包含其他文件（由 #include 指令控制）

  命令：g++ -E *.cpp -o *.i

2.1 头文件包含指令

使用命令 g++ -E fib.cpp -o fib.i 得到 fib 的预处理文件，可以发现 #include 的内容被全部替换，代码增加至 28185 行, 另外命名空间的内容也被引入。

```
# 1 "fib.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "fib.cpp"
# 1 "/usr/include/c++/7/iostream" 1 3
# 36 "/usr/include/c++/7/iostream" 3

# 37 "/usr/include/c++/7/iostream" 3

...

namespace std __attribute__ ((__visibility__ ("default")))
{
```

```
16
17  # 60 "/usr/include/c++/7/iostream" 3
18    extern istream cin;
19    extern ostream cout;
20    extern ostream cerr;
21    extern ostream clog;
22
23
24    extern wistream wcin;
25    extern wostream wcout;
26    extern wostream wcerr;
27    extern wostream wclog;
28
29
30
31
32    static ios_base::Init __ioinit;
33
34
35  }
36  # 2 "fib.cpp" 2
37
38  # 2 "fib.cpp"
39  using namespace std;
```

2.2 预处理条件指令与宏定义函数

编写一个新的程序 preTest.cpp，用于观察预处理部分对于含条件的预处理语句，以及宏定义函数的处理。代码如下：

```
1  //"prepTest.cpp"
2  #define MAX(a,b) (a>b?a:b)  //宏定义函数最大值
3
4  int main() {
5      int x=0;
6      int y=1;
7
8      #if (1<2)
9      x=x+y;
10     #else
11     x=x-y;
12     #endif
13
14     int c=MAX(x,y);
15     return 0;
16 }
```

得到的 prepTest.i 文件里可以看到：

- 预处理条件指令的假值部分被直接置为空行；
- MAX(x,y) 被替换为 (x>y?x:y)，注意到变量名与宏定义中的变量不同也可以替换；
- 另外注释的部分也被去除。

```
# 1 "prepTest.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "prepTest.cpp"


int main() {
    int x=0;
    int y=1;


    x=x+y;


    int c=(x>y?x:y);
    return 0;
}
```

3 编译阶段

编译阶段，在这个阶段中，Gcc 首先要检查代码的规范性、是否有语法错误等，以确定代码的实际要做的工作，在检查无误后，Gcc 把代码翻译成汇编语言。用户可以使用”-S”选项来进行查看，该选项只进行编译而不进行汇编，生成汇编代码。

编译器的主要工作流程分成 6 个阶段

- 词法分析
- 语法分析
- 语义分析
- 中间代码生成
- 代码优化

- 目标代码生成

  命令：gcc/g++ -S *.i -o *.s

词法分析. [1] 词法分析阶段源代码程序被输入到扫描器，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的算法可以将源代码字符序列分割成一系列的记号（Token）。

词法分析产生的记号一般可以分为如下几类：关键字、标识符、字面量（包含数字、字符串等）和特殊符号（如加号、等号）。在识别记号的同时，扫描器也完成了其他工作，比如将标识符存放到符号表，将数字、字符串常量存放到文字表等，以备后面的步骤使用。

深入了解一点，可以查到在 gcc 中的 libcpp 目录下，便是 gcc 对 c 系编程语言的词法分析模块，其中的 cpplib.h 文件便是外部程模块问词法分析模块的接口文件。

语法分析. 语法分析阶段，对由扫描器产生的记号进行语法分析，从而产生语法树（Syntax Tree）。整个分析过程采用了上下文无关语法（Context-free Grammar）的分析手段。

使用命令：g++ -fdump-tree-original-raw fib.cpp 得到一个 AST 文件'fib.i.003t.original'，文件中描述了语法树的情况，例如 fib.cpp 的 main() 函数被解析为一个 183 个节点的语法树（如下）：

在其中我们可以找到 main 函数里第一行"int n;" 分析得到的子树，以 @5 号节点为根节点 (srcp 指向 fib.cpp 文件的 20 行)，该节点属性为 var_decl（变量声明），子节点中：name 指向 @9 号节点，@9 号节点则为一个 strg 值为 n 的 identifier_node（标识符）；type 指向 @10 号节点，类型为 integer_type（整形），对应语句中'int'，而 @10 号节点的子节点里则是对该类型更加详细的定义。

```
;; Function int main() (null)
;; enabled by -tree-original

@1      statement_list   0   : @2       1   : @3
@2      bind_expr        type: @4       vars: @5       body: @6
@3      return_expr      type: @4       expr: @7
@4      void_type        name: @8       algn: 8
@5      var_decl         name: @9       type: @10      scpe: @11
                         srcp: fib.cpp:20              size: @12
                         algn: 32       used: 1
@6      statement_list   0   : @13      1   : @14      2   : @15
                         3   : @16
@7      init_expr        type: @10      op 0: @17      op 1: @18
@8      type_decl        name: @19      type: @4       srcp: <built-in>:0
                         note: artificial
@9      identifier_node  strg: n        lngt: 1
@10     integer_type     name: @20      size: @12      algn: 32
                         prec: 32       sign: signed   min : @21
                         max : @22
@11     function_decl    name: @23      type: @24      scpe: @25
                         srcp: fib.cpp:19              lang: C
                         link: extern
@12     integer_cst      type: @26     int: 32
```

```
@13     decl_expr        type: @4
@14     cleanup_point_expr type: @4       op 0: @27
@15     cleanup_point_expr type: @4       op 0: @28
@16     return_expr      type: @4       expr: @29
@17     result_decl      type: @10      scpe: @11      srcp: fib.cpp:19
                         note: artificial              size: @12
                         algn: 32
@18     integer_cst      type: @10     int: 0
@19     identifier_node  strg: void     lngt: 4
@20     type_decl        name: @30      type: @10      srcp: <built-in>:0
                         note: artificial
@21     integer_cst      type: @10     int: -2147483648
@22     integer_cst      type: @10     int: 2147483647

...

@182    const_decl       name: @181     type: @156     scpe: @156
                         srcp: ios_base.h:491          cnst: @183
@183    integer_cst      type: @156    int: 2
```

语义分析. 语义分析 [2] 是编译过程的一个逻辑阶段，语义分析的任务是对结构上正确的源程序进行上下文有关性质的审查，进行类型审查。语义分析是审查源程序有无语义错误，为代码生成阶段收集类型信息。

中间代码生成. 中间代码 [3] 生成是产生中间代码的过程。所谓“中间代码”是一种结构简单、含义明确的记号系统，这种记号系统复杂性介于源程序语言和机器语言之间，容易将它翻译成目标代码。另外，还可以在中间代码一级进行与机器无关的优化。

代码优化. GCC 提供了为了满足用户不同程度的的优化需要，提供了近百种优化选项，用来对编译时间，目标文件长度，执行效率这个三维模型进行不同的取舍和平衡 [4]。

优化的方法总体上将有以下几类：

- 精简操作指令
- 尽量满足 cpu 的流水操作
- 通过对程序行为地猜测，重新调整代码的执行顺序
- 充分使用寄存器
- 对简单的调用进行展开

另外 GCC 还提供了-O0，-O1，-O2，-O3 四级优化选项，在后面的代码优化章节再展开尝试。

目标代码生成. 目标代码生成是编译的最后一个阶段。目标代码生成器把语法分析后或优化后的中间代码变换成目标代码。

命令 g++ -S fib.i -o fib.s

fib.s. 得到的.s 文件即为汇编语言的代码文件，以下片段对应的是 main() 函数

```
.LFE1493:
    .size   _Z3fibi, .-_Z3fibi
    .globl  main
    .type   main, @function
main:
.LFB1494:
    .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    %fs:40, %rax
    movq    %rax, -8(%rbp)
    xorl    %eax, %eax
    leaq    -12(%rbp), %rax
    movq    %rax, %rsi
    leaq    _ZSt3cin(%rip), %rdi
    call    _ZNSirsERi@PLT
    movl    -12(%rbp), %eax
    movl    %eax, %edi
    call    _Z3fibi
    movl    $0, %eax
    movq    -8(%rbp), %rdx
    xorq    %fs:40, %rdx
    je  .L7
    call    __stack_chk_fail@PLT
.L7:
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

4 汇编阶段

汇编指把汇编语言代码翻译成目标机器指令的过程。对于被翻译系统处理的每一个 C 语言源程序，都将最终经过这一处理而得到相应的目标文件。

汇编生成的是一个可重定位文件，其中包含有适合于其它目标文件链接来创建一个可执行的或者共享的目标文件的代码和数据。

命令：g++ -c fib.cpp -o fib.o

生成的.o 文件若直接查看会是乱码，在终端中使用反汇编命令 objdump -j .text -S fib.o 可以查看 fib.o 文件的具体内容，以及每条机器码对应的汇编代码，对 main 对应的部分来说，可以看到 main 前面的 00000000000000b5 应当是函数的入口，这在后面中是可以重定位的，到指令'c3'，也就是 retq 为止是 main 函数的代码段内容。

```
00000000000000b5 <main>:
  b5:   55                      push   %rbp
  b6:   48 89 e5                mov    %rsp,%rbp
  b9:   48 83 ec 10             sub    $0x10,%rsp
  bd:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  c4:   00 00
  c6:   48 89 45 f8             mov    %rax,-0x8(%rbp)
  ca:   31 c0                   xor    %eax,%eax
  cc:   48 8d 45 f4             lea    -0xc(%rbp),%rax
  d0:   48 89 c6                mov    %rax,%rsi
  d3:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # da <main+0x25>
  da:   e8 00 00 00 00          callq  df <main+0x2a>
  df:   8b 45 f4                mov    -0xc(%rbp),%eax
  e2:   89 c7                   mov    %eax,%edi
  e4:   e8 00 00 00 00          callq  e9 <main+0x34>
  e9:   b8 00 00 00 00          mov    $0x0,%eax
  ee:   48 8b 55 f8             mov    -0x8(%rbp),%rdx
  f2:   64 48 33 14 25 28 00    xor    %fs:0x28,%rdx
  f9:   00 00
  fb:   74 05                   je     102 <main+0x4d>
  fd:   e8 00 00 00 00          callq  102 <main+0x4d>
 102:   c9                      leaveq
 103:   c3                      retq
```

另外，用 nm 命令也可以得到 fib.o 文件中的符号信息（函数和全局变量）

命令：nm fib.o

```
                 U __cxa_atexit
                 U __dso_handle
                 U _GLOBAL_OFFSET_TABLE_
000000000000014d t _GLOBAL__sub_I__Z3fibi
00000000000000b5 T main
                 U __stack_chk_fail
0000000000000000 T _Z3fibi
0000000000000104 t _Z41__static_initialization_and_destruction_0ii
                 U _ZNSirsERi
                 U _ZNSolsEi
                 U _ZNSolsEPFRSoS_E
```

```
                 U _ZNSt8ios_base4InitC1Ev
                 U _ZNSt8ios_base4InitD1Ev
                 U _ZSt3cin
                 U _ZSt4cout
                 U _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
0000000000000000 r _ZStL19piecewise_construct
0000000000000000 b _ZStL8__ioinit
```

在输出里面，可以发现源程序里的函数名被加上了各种前缀，另外，nm 为每个名称注明了一个属性，查询后可知其有如下含义：

- T：该符号位于代码区 text section
- U：该符号在当前文件中是未定义的，即该符号的定义在别的文件中。

5 链接阶段

链接器（Linker）是一个程序，将一个或多个由编译器或汇编器生成的目标文件外加库链接为一个可执行文件。目标文件是包括机器码和链接器可用信息的程序模块。gcc 调用的链接器为 ld。

- 解析未定义的符号引用
- 将目标文件中的占位符替换为符号的地址
- 完成程序中各目标文件的地址空间的组织，这涉及重定位工作

命令：g++ -o fib fib.o

得到的可执行程序可以通过 objdump -d fib 进行反汇编，同样以 main 函数段为例，可以看到函数入口变为了 0000000000000a3f，与之前的'b5' 不同：

```
0000000000000a3f <main>:
 a3f:   55                      push   %rbp
 a40:   48 89 e5                mov    %rsp,%rbp
 a43:   48 83 ec 10             sub    $0x10,%rsp
 a47:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
 a4e:   00 00
 a50:   48 89 45 f8             mov    %rax,-0x8(%rbp)
 a54:   31 c0                   xor    %eax,%eax
 a56:   48 8d 45 f4             lea    -0xc(%rbp),%rax
 a5a:   48 89 c6                mov    %rax,%rsi
 a5d:   48 8d 3d dc 06 20 00    lea    0x2006dc(%rip),%rdi        # 201140 <
    _ZSt3cin@@GLIBCXX_3.4>
 a64:   e8 a7 fd ff ff          callq  810 <_ZNSirsERi@plt>
 a69:   8b 45 f4                mov    -0xc(%rbp),%eax
 a6c:   89 c7                   mov    %eax,%edi
 a6e:   e8 17 ff ff ff          callq  98a <_Z3fibi>
 a73:   b8 00 00 00 00          mov    $0x0,%eax
```

```
a78:   48 8b 55 f8             mov    -0x8(%rbp),%rdx
a7c:   64 48 33 14 25 28 00    xor    %fs:0x28,%rdx
a83:   00 00
a85:   74 05                   je     a8c <main+0x4d>
a87:   e8 b4 fd ff ff          callq  840 <__stack_chk_fail@plt>
a8c:   c9                      leaveq
a8d:   c3                      retq
```

在.o 文件中，诸如 cout()、printf() 一类的的的完整函数定义并没有出现，而后续却可以使用，查询了解到：这是因为系统把这些函数实现都被做到名为 libc.so.6 的库文件中去了，默认情况下，gcc 会到系统默认的搜索路径”/usr/lib”下进行查找，从而链接到 libc.so.6 库函数中，得以实现这些函数

6 补充：代码优化

代码优化的基本内容在编译阶段章节里已经阐述，这里编写一个程序，用来测试编译器的优化功能。代码如下：

```c
#include<stdio.h>

static void wait(long long int d)
{
    for(; d > 0; d--);
    printf("1");
}
void main()
{
    unsigned long i = 0;
    while(1){
        wait(30000);
        if(++i == 16)
            i = 0;
        printf("i");
    }
}
```

使用以下命令生成编译的结果

```
gcc -S -O0 test.c -o t0.s
gcc -S -O1 test.c -o t1.s
gcc -S -O2 test.c -o t2.s
gcc -S -O3 test.c -o t3.s
```

为方便比较，将 4 个文件并列显示，从左到右分别是使用了 O0、O1、O2、O3 的结果，首先可以明显感受到代码量的逐次减少（O2 与 O3 相近可能是因为程序本身复杂性较低的原因），阅读可以发现一些更细节的工作：

- O1 将 main 中调用的函数 wait 作了一次展开，减少了调用跳转造成的性能消耗，另外两级同样有此操作。
- 在 O0 里出现了很多的内存调用，而在 O1 里内存调用几乎没有，计算基本在寄存器之间进行。例如变量 i 在 O0 是存在-8(%rbp) 中，而 O1 则选择存在%rbx 里。
- 在 O1 里的 24 行'cmove %rbp,%rbx' 有一个小操作，首先因为 if 语句里只有 1 个赋值语句，故用 cmove（根据比较结果执行）指令从而减少了 if 带来的跳转。另外，在 O0 里是通过立即数 0 赋值给 i 所在的位置，而 O1 通过在循环外提前提前将 0 存在寄存器%rbp 里，进一步减少了循环里对内存的访问。
- 如上一条，还有一些循环相关的操作被提出到了循环外。
- O2 里，优化操作对一些对输出没有意义的操作进行了删除，代码量以及执行时间大大减少。例如 wait 函数里的空循环被完全删除。

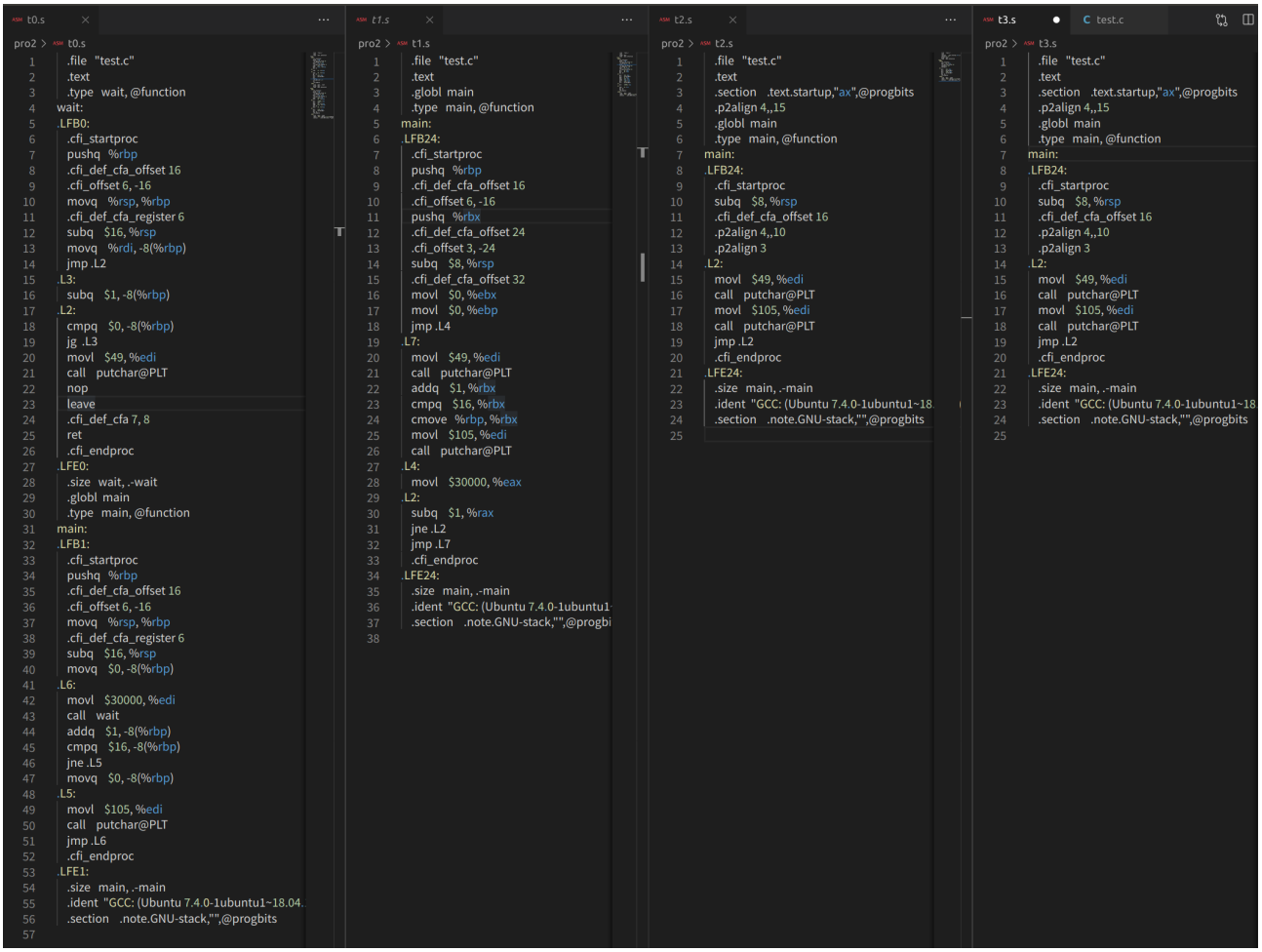

图 1: 优化结果

通过 GCC 文档 [5] 可知 gcc 几个不同的-o 优化选项实际上是开启了不同数量的优化选项，而使用-f 命令行可以引用每个单独的优化技术，这里不再赘述。

7 结论

通过观察几个源程序在编译流程中每一步的输出结果，并且查阅相关资料，结合课堂上老师讲授过的部分内容，笔者熟悉了 gcc 编译器的使用，深入了解了各个阶段的工作与功能，还对代码优化过程有了初步的认识。基于此为后续的实验打好了基础，也通过对底层加深的认识为以后在其他课程与研究做好了铺垫。

参考文献

[1] NoneName *https://blog.csdn.net/m0*$_4$6338137/*article*/*details*/104558432, 2020

[2] BaiduBaike *https://baike.baidu.com/item/%E8%AF%AD%E4%B9%89%E5%88%86%E6%9E%90/8853372?fr=aladdin*,2018

[3] BaiduBaike *https://baike.baidu.com/item/%E4%B8%AD%E9%97%B4%E4%BB%A3%E7%A0%81%E7%94%9F%E6%88%90/8853318?fr=aladdin*,2018

[4] 沈二月 *https://blog.csdn.net/m0*$_4$6338137/*article*/*details*/104558432, 2016

[5] GNU *https://gcc.gnu.org/wiki*,2020