



南开大学  
Nankai University

南 开 大 学

计算机学院和网络空间安全学院

编译原理实验报告

---

## 预备工作一

---

陈宇轩 1911400

年级：2019 级

专业：物联网工程

指导教师：王刚

2021 年 9 月 25 日

## 摘要

以 gcc、llvm 等为研究对象，利用各种方法，通过样例代码，简单的探究了代码处理过程中的各个步骤环节，包括预处理器、编译器、汇编器和链接器。

**关键字：**Compiler , gcc , llvm , AST , RTL , Linker

## 目录

一、 总体流程	1
(一) 预处理器	1
(二) 编译器	1
1. 词法分析	1
2. 语法分析	2
3. 语义分析	3
4. 中间代码生成和优化	3
5. 代码生成	6
6. gcc 的优化级别	6
(三) 汇编器	7
1. 指令选择	7
2. 寄存器分配	7
3. 指令调度	7
4. 指令编码	7
5. 一个例子	7
(四) 链接器、加载器	9
二、 总结	9

## 一、 总体流程

### (一) 预处理器

C/C++ 代码的预处理器会在输入文件被编译之前根据预处理指令处理程序，主要包括文件包含、宏替换和条件编译等，取决于预处理指令怎样书写。

实验使用的代码如下：

```
1 #include<stdio.h>
2 int main(){
3     int i,n,f;
4     scanf("%d",&n);
5     i=2;
6     f=1;
7     while(i<=n){
8         f=f*i;
9         i=i+1;
10    }
11    printf("%d\n",f);
12 }
```

使用 gcc 进行预编译处理，利用下列指令

```
1 gcc test.c -E -o test.i
```

将文件预编译结果输出到 `test.i` 文件中，此时它仍然是一个 C 语言文件。gcc 调用了 C Pre-Processor 完成了例如替换 include 指令对应的头文件、添加行号和文件名标识等工作。因为读入了 `stdio.h`，预处理完成的文件变得十分庞大，观察一些内容就能知道，C Pre-Processor 在预处理程序时会先隐式的读取 `stdc-predef.h` 文件，接着再根据 `stdio.h` 读入标准输入输出所需要的类型、宏和函数。

具体的实现细节在官方给出的[说明文档](#)中有记录。而 `test.i` 的内容就是编译器要处理的全部输入了。

### (二) 编译器

#### 1. 词法分析

这一步要将源程序转换为有意义的单词 (token) 序列，编译器通过调用扫描器，让扫描器来完成这项工作。命令

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

能够获得 token 序列，储存在 `token.txt` 文件中：

```
1 typedef 'typedef' [StartOfLine] Loc=</usr/lib/llvm-6.0/lib/clang
  /6.0.0/include/stddef.h:62:1>
2 long 'long' [LeadingSpace] Loc=</usr/lib/llvm-6.0/lib/clang/6.0.0/
  include/stddef.h:62:9 <Spelling=<built-in>:79:23>>
3 unsigned 'unsigned' [LeadingSpace] Loc=</usr/lib/llvm-6.0/lib/clang
  /6.0.0/include/stddef.h:62:9 <Spelling=<built-in>:79:28>>
4 int 'int' [LeadingSpace] Loc=</usr/lib/llvm-6.0/lib/clang/6.0.0/
  include/stddef.h:62:9 <Spelling=<built-in>:79:37>>
```

```

5 identifier 'size_t' [LeadingSpace] Loc=</usr/lib/llvm-6.0/lib/clang
  /6.0.0/include/stddef.h:62:23>
6 semi ';' Loc=</usr/lib/llvm-6.0/lib/clang/6.0.0/include/stddef
  .h:62:29>
7 ...

```

例如 llvm 处理的 token 序列结构形如：

```

1 属性 '内容' ([位置]) 定义位置

```

记录了每个单词的类型、“相貌”——它的值——和位置类型以及具体位置，以便之后的流程使用。

这一阶段，源代码程序被输入到扫描器中，扫描器对源代码进行简单的词法分析，将源代码字符序列分割。

lex 是一种生成扫描器的工具，它工作的基本步骤是

1. 程序员编写文件，指定需要被扫描的词汇模式，lex 生成的扫描器将根据这种模式进行工作

2. 对文件运行 lex，生成扫描器的 C 代码

3. 编译、链接 C 代码，生成可执行文件

lex 的工作依赖于

## 2. 语法分析

接下来，编译器要将单词序列处理为语法树。以 gcc 为例，使用命令

```

1 gcc fdump-tree-original-raw test.c

```

得到两个输出文件，[a.out](#)和[test.c.003t.original](#)，a.out 是 test.c 编译的可执行文件，test.c.003t.original 就是获得的 AST 文件：

```

1 ;; Function main (null)
2 ;; enabled by -tree-original
3
4 @1      statement_list  0   : @2      1   : @3
5 @2      bind_expr      type: @4      vars: @5      body: @6
6 @3      return_expr    type: @4      expr: @7
7 @4      void_type      name: @8      algn: 8
8 @5      var_decl       name: @9      type: @10     scpe: @11
9                               srcp: test.c:3      size: @12
10                               algn: 32      used: 1
11 @6      statement_list  0   : @13     1   : @14     2   : @15
12                               3   : @16     4   : @17     5   : @18
13                               6   : @19     7   : @20     8   : @21
14                               9   : @22     10  : @23     11  : @24
15                               12  : @25     13  : @26
16 ...

```

它的形式很容易让人联想到形如“如果你的答案是 x，请跳至题目 x”的一种测试选择题，但是这里选项变成了分支，最终组成树的形式。每一行是树的一个节点，行头是唯一的 ID，接着是节点的属性，之后是子节点的 ID 和父节点中它们各自的属性。

### 3. 语义分析

标准委员会制订了语义标准，编译器将会根据这些标准、生成的符号表和语法树来检测源程序是否符合语义，进行类型检查等。

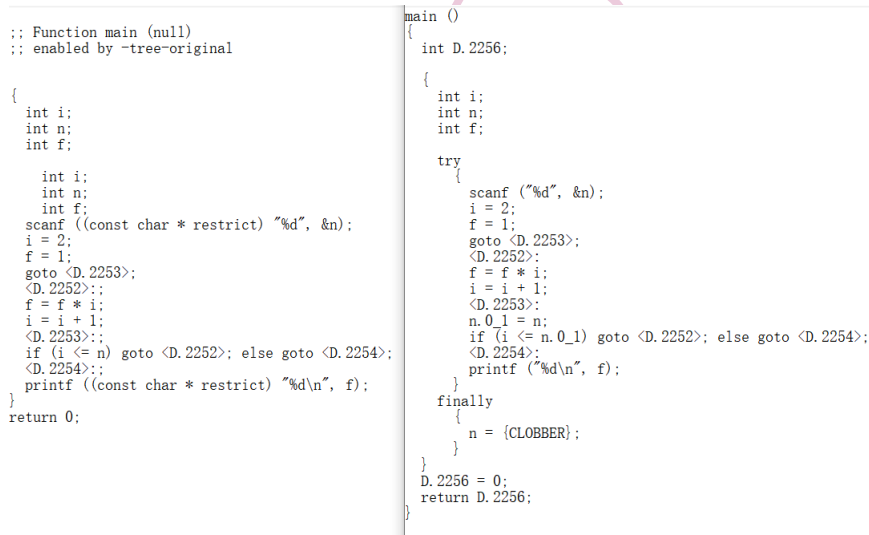
### 4. 中间代码生成和优化

这时大部分编译器已经生成了一组较源程序语言更加贴近底层，但是比二进制代码或汇编语言更加清晰，更加便于分析逻辑结构的中间代码。它从源程序中提取出语义和语法结构，但并不依赖源程序语言或底层机器，是沟通前端和后端的桥梁。

gcc 有三种不同的中间表示语言：AST/GENERIC(C 前端) 比较完善的表示了前端语言的信息；GIMPLE 从 AST 转换而来，是一种前端无关的中间表示；以及 RTL(Register Transfer Language)，它就是那个硬件无关以满足可复用性、高度抽象以满足可优化性的中间语言。

gcc 在将源代码转化成中间代码的过程中会依次遍历这三种中间语言，AST 已经在之前的语法分析阶段生成了，并且由于不同的前端高级语言，gcc 生成的 AST 是不同的，因此就需要转换为统一的 GIMPLE 形式。

可以使用 `-fdump-tree-gimple`、`-fdump-rtl-all` 等 flag 来获得不同阶段的表达式，也可以获得 CFG 和对应的 dot 图像。例如，我们对比 AST 和 GIMPLE 状态的代码：如图1所示



```

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

{
  int i;
  int n;
  int f;

  int i;
  int n;
  int f;
  scanf ((const char * restrict) "%d", &n);
  i = 2;
  f = 1;
  goto <D.2253>;
<D.2252>:
  f = f * i;
  i = i + 1;
<D.2253>:
  if (i <= n) goto <D.2252>; else goto <D.2254>;
<D.2254>:
  printf ((const char * restrict) "%d\n", f);
}
return 0;

```

```

main ()
{
  int D.2256;

  {
    int i;
    int n;
    int f;

    try
    {
      scanf ("%d", &n);
      i = 2;
      f = 1;
      goto <D.2253>;
<D.2252>:
      f = f * i;
      i = i + 1;
<D.2253>:
      n.0_1 = n;
      if (i <= n.0_1) goto <D.2252>; else goto <D.2254>;
<D.2254>:
      printf ("%d\n", f);
    }
    finally
    {
      n = {CLOBBER};
    }
  }
  D.2256 = 0;
  return D.2256;
}

```

图 1: GENERIC 和 GIMPLE

左边的文件来自[test.c.003t.original](#)，右边的则是[test.c.004t.gimple](#)，这两个文件可以通过使用 `-fdump-tree-all-graph` 这个 flag 得到。对比之下就会发现，GIMPLE 表达式使用了临时变量来储存计算中的中间值乃至返回值。除此之外，GIMPLE 相比 GENERIC 还有一些其他优势，诸如 GENERIC 是以树结构表示和储存的、GIMPLE 前端无关而 GENERIC 相关、GIMPLE 本质是线性代码序列，能够更方便有效的进行后续的编译优化等。

gcc 在生成 GIMPLE 形式后，会先在这个层面上进行优化——低级化、构建 CFG 等一系列处理——再转换为 RTL 形式。这些操作被称为 GCC Pass，每一种 pass 完成一种处理，结果再作为下一个 pass 的输入。

gcc 描述 pass 的核心数据结构在 `passes.c` 文件中。优化 pass 分成 4 类，分别是 GIMPLE\_PASS、RTL\_PASS、SIMPLE\_IPA\_PASS、IPA\_PASS，其中除了 RTL\_PASS 以 RTL 为处理对象，其他都以 GIMPLE 为对象。通过 `-fdump-tree-all-graph` 这个 flag 可以生成具体的

层次变化和对应的 CFG 流程图, 通过名字我们就能大致看出来 gcc 在这个阶段采取了怎样的优化。

获得 GIMPLE 形式表达后, gcc 后续还进行了许多处理步骤, 保存在特定后缀的文件中, 每一遍的操作都属于 IPA Pass(过程间优化), 就像做菜时的手法, 它们在各个过程间被适当的调用, 以完成一定的工作, 也有可能被重复调用。

上面谈到的有关 pass 的过程是借助 Pass Manager 完成的, 它位于 gcc 源码中的 passes.c、tree-optimize.c、tree-pass.h 文件中, 根据 passes.def 中的定义来处理 pass。pass 工作的原理就是让每一个 pass 定义一个结构, 记录这个 pass 何时运行、怎样运行、针对怎样的中间表达形式以及是否需要其他数据结构支持。在设置好 pass 以某种特定的规则运行以后, Pass Manager 就会保证它们按照规定执行。

阅读这些中间文件——使用 `-fdump-tree-all-graph` 获得的一系列文档——会发现这些处理中代码形式并没有太大变化, 有一些关于返回值和计算中间量的存储和表示的变化, 但没有更多信息显得非常费解, 这可能是由于源程序逻辑过于简单, 以至于体现不出意图。

更加简单的方式就是阅读说明文档和源码。[GCC Internals](#)中给出了获得 GIMPLE 表示后, 获得 RTL 之前, gcc 做了哪些工作。此时调用的 pass 都属于 Tree SSA Pass, SSA 是静态单赋值, 是一种中间表示形式, 每个名字在 SSA 中只被赋值一次。

现在就能够知道一些文件处理的意义了, 例如 `test.c.019t.fixup_cfg1` 和 `test.c.020t.ssa` 之间调用了 Enter static single assignment form 处理文件, 这个 pass 重写了程序使其成为 SSA 形式。如图2

```

;; Function main (main, funcdef_no=0, decl_uid=2247, cgraph_uid=0, symbol_order=0)
main ()
{
  int f;
  int n;
  int i;
  int D.2256;

  <bb 2> [0.00%]:
  scanf ("%d", &n);
  i = 2;
  f = 1;
  goto <bb 4> [0.00%]

  <bb 3> [0.00%]:
  f = f * i;
  i = i + 1;

  <bb 4> [0.00%]:
  n_0_1 = n;
  if (i <= n_0_1)
    goto <bb 3> [0.00%]
  else
    goto <bb 5> [0.00%]

  <bb 5> [0.00%]:
  printf ("%d\n", f);
  n = {CLOBBER};
  D.2256 = 0;

  <L3> [0.00%]:
  return D.2256;
}

;; Function main (main, funcdef_no=0, decl_uid=2247, cgraph_uid=0, symbol_order=0)
main ()
{
  int f;
  int n;
  int i;
  int D.2256;
  int n_0_1;
  int _10;

  <bb 2> [0.00%]:
  scanf ("%d", &n);
  i_0 = 2;
  f_7 = 1;
  goto <bb 4> [0.00%]

  <bb 3> [0.00%]:
  f_11 = f_3 * i_2;
  i_12 = i_2 + 1;

  <bb 4> [0.00%]:
  # i_2 = PHI <i_6(2), i_12(3)>
  # f_3 = PHI <f_7(2), f_11(3)>
  n_0_1 = n;
  if (i_2 <= n_0_1)
    goto <bb 3> [0.00%]
  else
    goto <bb 5> [0.00%]

  <bb 5> [0.00%]:
  printf ("%d\n", f_3);
  n = {v} {CLOBBER};
  _10 = 0;

  <L3> [0.00%]:
  return _10;
}

```

图 2: CFG 和 SSA

处理后所有 `is_gimple_reg` 变量<sup>1</sup> 都会以 `SSA_NAME` 引用 (后面加上了序号), 为每个基本块按需加载 PHI 节点<sup>2</sup> (在基本块 4 中添加了 PHI 指令) 等。这个 pass 定义在 `tree-ssa.c` 中, 由 `pass_build_ssa` 定义描述。<sup>3</sup> 这样, 我们就能一步步了解 gcc 在中间代码生成中做了什么。

接下来是向 RTL 的转化, 使用 `-fdump-rtl-all-graph`, 可以得到 RTL 形式下 CFG 的变化和对应 dot 图。

使用 `-fdum-rtl-all` 可以得到 RTL 形式和在在其上做的一系列优化, 获得的第一个文件 `test.c.229r.expand`

<sup>1</sup>无法寻址的堆栈变量

<sup>2</sup>PHI 节点由 PHI 指令实现。PHI 指令是和基本块相关联的, 它会根据上文 BLOCK 的内容决定取值

<sup>3</sup>令人失望的是, 可能是由于这份 gcc internal 和发布在 github 上的 gcc 源码版本不同, 我并没有找到源代码

给了关于 GIMPLE 形式是怎样转化为 RTL 形式的一些提示。

```

1  ;; Function main (main, funcdef_no=0, decl_uid=2247, cgraph_uid=0,
    symbol_order=0)
2
3  Partition 1: size 4 align 4
4      f_3
5  Partition 0: size 4 align 4
6      i_2
7  Partition 2: size 4 align 4
8      n
9
10 ;; Generating RTL for gimple basic block 2
11
12 ;; Generating RTL for gimple basic block 3
13
14 ;; Generating RTL for gimple basic block 4
15
16 ;; Generating RTL for gimple basic block 5
17
18 ;; Generating RTL for gimple basic block 6
19
20 try_optimize_cfg iteration 1
21
22 Merging block 3 into block 2...
23 Merged blocks 2 and 3.
24 Merged 2 and 3 without moving.
25 Merging block 7 into block 6...
26 Merged blocks 6 and 7.
27 Merged 6 and 7 without moving.
28 Removing jump 36.
29 Merging block 8 into block 6...
30 Merged blocks 6 and 8.
31 Merged 6 and 8 without moving.
32
33 try_optimize_cfg iteration 2
34 ...

```

代码的格式和最初的 c 代码也没有太大区别，声明了函数和三个变量，接下来是一段声明，“为 gimple 基本块 x 建立 RTL”，虽然没有具体细节，但是从中可以知道从 GIMPLE 向 RTL 转换是以基本块为单位的。接着是对已经转化为 RTL 形式，但仍保持着原来基本块结构的代码的优化，文件中记录了对基本块、跳转语句的一些操作。[GCC Internals](#)中记录了 RTL 的 pass 的简短介绍和如何在源码中找到它们。

代码的 RTL 表示被称为 INSN，它和之前的形式看起来完全不同了。有些 INSN 表示实际的指令，有些代表 switch 语句的跳转表，有些表示程序跳转所对应的标号，还有一些可以表示各种不同的声明信息。所有的 INSN 被一个双向链表所链接。

INSN 的具体格式如下：

(insn 第 1 个操作数 第 2 个操作数 第 3 个操作数 第 4 个操作数... 第 7 操作数)

在一个函数中，每一个 INSN 都具有唯一的标识 ID(整数)，也就是第一个操作数，同时，每个 INSN 还包含了它的前驱 (第二个操作数) 和后继 (第三个操作数)。第四个操作数表示该条指令序列所在的基本块，这可以从列举的第二条指令看出。第五个操作数就是这个 INSN 的主体了，描述了该 INSN 的 RTL 指令模版。第六个操作数是 INSN 的代码，即该 INSN 描述动作的对应指令的索引值。第七操作数未使用。

```

1  ...
2  (note 1 0 4 NOTE_INSN_DELETED)
3  (note 4 1 2 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
4  (note 2 4 3 2 NOTE_INSN_FUNCTION_BEG)
5  (insn 3 2 6 2 (parallel [
6      (set (mem/v/f/c:DI (plus:DI (reg/f:DI 82 virtual-stack-vars)
7          (const_int -8 [0xffffffffffffffff8])) [1 D.2259+0 S8
8              A64])
9          (unspec:DI [
10             (const_int 40 [0x28])
11             ] UNSPEC_SP_TLS_SET))
12      (set (scratch:DI)
13          (const_int 0 [0]))
14      (clobber (reg:CC 17 flags))
15      ]) "test.c":2 -1
16  (nil))
17  ...

```

这是[test.c.229r.expand](#)开头的几个 INSN，即使不知道 RTL 语言的细节，通过 INSN 中的描述和 DI 等关键字，也能够猜出大概是在完成完善基本块、分配堆栈等任务。

经过 gcc 对中间代码的一遍遍优化，最终我们能得到面向后端的汇编语言。

## 5. 代码生成

上面关于中间语言的过程隐藏的很隐蔽，直接以预处理后的[test.i](#) 文件作为输入

```

1  gcc test.i -S -o test_x86.S #生成 x86 格式目标代码
2  arm-linux-gnueabi-gcc test.i -S -o test_arm.S #生成 arm 格式目标代码

```

我们就能获得 arm 和 x86 格式的目标语言格式代码[test\\_x86.S](#)和 [test\\_arm.S](#)，上面的工作都完成了。

## 6. gcc 的优化级别

gcc 提供了编译时的优化选项供我们选择，分为从 O0 到 O3 的级别，和 Os、Og、Ofast 等特殊级别。在编译时加入这些选项可以粗略的查看 gcc 提供了怎样的优化。

```

1  gcc -O0 -o o0.S -S -masm=att test.i #获得O0级别输出
2  gcc -O1 -o o1.S -S -masm=att test.i #获得O1级别输出

```

其中 O0 不做任何优化，也是默认的编译选项，[o0.S](#)的内容就是 gcc 直接获得的汇编代码。它和执行 O1 级优化获得的[o1.S](#)的区别在于，O1 优化试图消耗较少的编译时间，以获得一定的优化，它主要对代码的分支、常量以及表达式等进行优化。相较于 O0 不做任何优化，它会延迟栈的弹出时间，简化连续的、条件相关的比较语句的跳转，执行循环优化等。O2 优化则会尝试更多的寄存



器级和指令级的优化，占用更多的编译时间，O3 自然更加复杂。随着优化等级的一步步上升，它们消耗的时间和资源，以及最终获得的代码复杂程度都会上升。通过阅读 [GCC MANUAL](#)<sup>p151</sup> 开始的关于编译优化的内容，我们能获得更多信息。

### (三) 汇编器

汇编器属于编译器的后端，这里将要输出对应的机器代码。它的工作是生成可重定位的机器代码，为此，汇编器需要翻译程序中那些能够完成自己任务的代码块，通常还需要进行优化，使得最终链接器把所有文件整合生成可执行文件的过程和结果更优。汇编器同样有一定的工作流程，以 Beignet 的后端部分为例，它分为四个步骤：

#### 1. 指令选择

这一阶段中，汇编器会非常简单、快速、草率的生成指令序列。这看起来很像之前中间代码生成时做的事：先翻译，再优化。此时我们会在尽可能控制成本的基础上，把每个基本块翻成一串 ISA 指令组，而做法就是简单的把每一句中间语言指令不计数量地翻译成 ISA 指令去完成功能。在这个过程中，事实上我们输出的是一些 `SelectionInstruction` 对象，它能够通过编码函数一对一的翻译为 ISA，并且它仍然使用未定位的虚拟寄存器及其元组。

#### 2. 寄存器分配

它分为两个步骤：

1. 为需要的指令处理向量。步骤一需要扫描所有需要被发射的向量。为了避免不同向量使用同一个寄存器可能引发的冲突和干扰，Beignet 把向量从小到大排列并按顺序分配，还识别了包含在较大向量中的子向量保留。

2. 执行寄存器分配。步骤二就是将每个虚拟寄存器和物理寄存器相关联。寄存器显然不能任意分配，我们需要考虑各种策略和机制保证它们在之后的过程中不会出错。

#### 3. 指令调度

引入寄存器以后，指令的执行顺序很有可能需要被做出调整，这部分优化不能提前完成。我们可以想象汇编器以调用 `pass` 的形式对指令进行遍历和优化。

#### 4. 指令编码

这一步骤就是将代码最终在机器上运行可能遇到的，包括基于硬件基础的问题解决，成为真正的机器码。

#### 5. 一个例子

还是举开头的 C 代码为例，获得汇编器处理后的文件，并用反汇编查看其代码

```
1 gcc -c -o test.o test.c      #获得汇编后的可重定位文件test.o
2 objdump -d test.o > test-anti-obj.S    #反汇编查看获得的文件
3 nm test.o > test-nm-obj.txt    #列出文件中的符号信息(定义出的函数、全局变量等)
```

我们获得了 `test-anti-obj.S` 和 `test-nm-obj.txt`，分别是反汇编指令和 `name` 指令的结果。

```

1  0000000000000000 <main>:
2  0:  55                push   %rbp
3  1:  48 89 e5           mov     %rsp,%rbp
4  4:  48 83 ec 20        sub     $0x20,%rsp
5  8:  64 48 8b 04 25 28 00 mov     %fs:0x28,%rax
6  f:  00 00
7  11: 48 89 45 f8        mov     %rax,-0x8(%rbp)
8  15: 31 c0              xor     %eax,%eax
9  17: 48 8d 45 ec        lea     -0x14(%rbp),%rax
10 1b: 48 89 c6           mov     %rax,%rsi
11 1e: 48 8d 3d 00 00 00 00 lea     0x0(%rip),%rdi    # 25 <main+0x25>
12 25: b8 00 00 00 00     mov     $0x0,%eax
13 2a: e8 00 00 00 00     callq   2f <main+0x2f>
14 2f: c7 45 f0 02 00 00 00 movl    $0x2,-0x10(%rbp)
15 36: c7 45 f4 01 00 00 00 movl    $0x1,-0xc(%rbp)
16 3d: eb 0e             jmp     4d <main+0x4d>
17 3f: 8b 45 f4          mov     -0xc(%rbp),%eax
18 42: 0f af 45 f0       imul    -0x10(%rbp),%eax
19 46: 89 45 f4          mov     %eax,-0xc(%rbp)
20 49: 83 45 f0 01       addl    $0x1,-0x10(%rbp)
21 4d: 8b 45 ec          mov     -0x14(%rbp),%eax
22 50: 39 45 f0          cmp     %eax,-0x10(%rbp)
23 53: 7e ea            jle     3f <main+0x3f>
24 55: 8b 45 f4          mov     -0xc(%rbp),%eax
25 58: 89 c6            mov     %eax,%esi
26 5a: 48 8d 3d 00 00 00 00 lea     0x0(%rip),%rdi    # 61 <main+0x61>
27 61: b8 00 00 00 00     mov     $0x0,%eax
28 66: e8 00 00 00 00     callq   6b <main+0x6b>
29 6b: b8 00 00 00 00     mov     $0x0,%eax
30 70: 48 8b 55 f8       mov     -0x8(%rbp),%rdx
31 74: 64 48 33 14 25 28 00 xor     %fs:0x28,%rdx
32 7b: 00 00
33 7d: 74 05            je      84 <main+0x84>
34 7f: e8 00 00 00 00     callq   84 <main+0x84>
35 84: c9              leaveq  %rsp,%rbp
36 85: c3              retq

```

即使不懂得汇编语言，第一眼我们就能发现它较之前利用-S 参数只编译生成的 `test.S` 文件要紧凑，同时至少能注意到的是，机器指令中所有的跳转地址都定义为 `<main+0xXX>` 的形式。

再看 `nm` 指令获得的输出

```

1          U _GLOBAL_OFFSET_TABLE_
2          U __isoc99_scanf
3 0000000000000000 T main
4          U printf
5          U __stack_chk_fail

```

这里 U 和 T 都是符号类型，U 表示未定义，需从其他对象文件链接进来，可以看出都是外部的

全局偏移表和库函数等；T 表示符号在代码段中，也就是 main 函数。这些在链接完成后想必会有所变化。

#### (四) 链接器、加载器

链接器的工作，是确保此前文件中的依赖关系是正确的，将给定的目标文件集合拼接打包，最终对其进行重定位。

还是看例子。链接汇编后的.o 文件并对其执行反汇编等操作

```
1 gcc -o TEST test.o
2 objdump -d TEST > TEST-anti-obj.S
3 nm TEST > TEST-nm-obj.txt
```

最终我们获得了可执行文件 `TEST`，和对它执行反汇编和 `nm` 指令获得的输出 `TEST1-anti-obj.S`、`TEST1-nm-obj.txt`。现在反汇编文件已经大到不适合放进文中了，这当然是因为程序运行所需要的所有代码现在都被链接进了 `TEST` 中，它们各自成块，彼此调用。`nm` 指令现在获得的部分输出如下：

```
1 ...
2 0000000000200fa8 d _GLOBAL_OFFSET_TABLE_
3          w __gmon_start__
4 000000000000082c r __GNU_EH_FRAME_HDR
5 00000000000005a8 T __init
6 0000000000200db0 t __init_array_end
7 0000000000200da8 t __init_array_start
8 0000000000000820 R _IO_stdin_used
9          U __isoc99_scanf@@GLIBC_2.7
10         w _ITM_deregisterTMCloneTable
11         w _ITM_registerTMCloneTable
12 0000000000000810 T __libc_csu_fini
13 00000000000007a0 T __libc_csu_init
14         U __libc_start_main@@GLIBC_2.2.5
15 000000000000071a T main
16         U printf@@GLIBC_2.2.5
17 0000000000000680 t register_tm_clones
18         U __stack_chk_fail@@GLIBC_2.4
19 0000000000000610 T _start
20 0000000000201010 D __TMC_END__
```

果然，现在文件中有大量的符号，它们原本存放在库文件或其他地方，现在被集中到了可执行文件中。

但是，如果在链接时使用 `-static` 参数，再执行同样操作，我们获得的文件<sup>4</sup>的大小会比现在恐怖的多，这就是静态链接和动态链接的区别，是它们在对库的取舍——我全都要还是一会儿来拿——和链接时机的选择——现在就执行和开始执行时再说——的不同造成的。

## 二、 总结

<sup>4</sup>`staticTEST-anti-obj.S`、`staticTEST-nm-obj.txt`