

南开大学

编译系统原理

作业题目：编译原理课程作业报告之预习作业一

学 号：1811463

姓 名：赵梓杰

年 级：2018 级

专 业：信息安全专业

学 院：网络空间安全学院

完成日期：2020 年 9 月

摘 要

我相信，很多的同学和我一样，大学的第一段代码都是利用 VS，编写 C++ 的“Hello,world!”开始的，不过当接触到编译原理之后，才知道我们之前编译运行一气呵成的 Hello,wrold 代码也隐藏了很多的进程和任务，而在这一个完整的编译过程中，又大致可以分为预处理、编译、汇编和链接四个过程。本文以 Linux 环境下 GCC 编译器编译 C++ 为例，结合预习 PPT 中所学知识、网络上查阅的相关资料以及相关课外读物，对编译器的主要工作流程进行深度(简单)分析，以希望能够对编译原理课程有更深的认识(期末考更高的分数)。(本文中所用的代码见附录)

关键词：编译器，预处理器，编译器，汇编器，链接

目录

摘要	I
目录	II
第一章 编译环境简介	1
第一节 GCC 编译器简介	1
第二节 GCC 基本用法	1
第二章 语言处理系统处理过程	2
第一节 预处理——预编译	2
2.1.1 预处理过程的概述	2
2.1.2 预处理语句	3
2.1.3 预处理语句的文法	3
2.1.4 预处理器调用命令	3
2.1.5 输出结果分析	5
第二节 重中之重——编译	8
2.2.1 预处理过程的概述	8
2.2.2 GCC 所采用的 AT&T 格式汇编语言	8
2.2.3 编译器调用指令	9
2.2.4 编译器过程详解	10
2.2.5 编译器优化处理	13
第三节 汇编	16
2.3.1 汇编过程概述	16
2.3.2 汇编样例输出	16
2.3.3 目标文件的介绍	17
2.3.4 深度理解 fib.o 文件	19
第四节 链接——一个很容易被“我们”忽视的环节	21
2.4.1 链接环节的介绍	21
2.4.2 链接的概念	22
2.4.3 链接样例输出	22
第五节 整体优化	23

2.5.1	pipe 优化原理概述	23
2.5.2	pipe 优化样例	24
第三章	总结	26
参考文献	27
附录 A	28

第一章 编译环境简介

在开发一些程序的时候，大部分情况下我们都不会采用汇编等语言，而是采用一些高级语言进行开发。利用高级语言进行程序开发定然离不开高级语言编译器，因此就引出了非常强大的编译器组件——GCC。

第一节 GCC 编译器简介

GCC 原为 GNU C 语言的编译器，顾名思义，只能编译 C 语言的程序代码，后续做了优化和拓展，支持了 C++ 等语言。

而 GCC 更为强大的一点即为 GCC 允许程序员将编译过程中得到的语法中间表达导出成为数据文件，可以让程序员通过中间文件进一步了解到编译代码过程中所经历的过程。

第二节 GCC 基本用法

由于本文主要讨论的并非为 GCC，GCC 仅仅只是协助我们进行了解语言处理系统的流程，所以在这对 GCC 的用法仅做简单介绍。

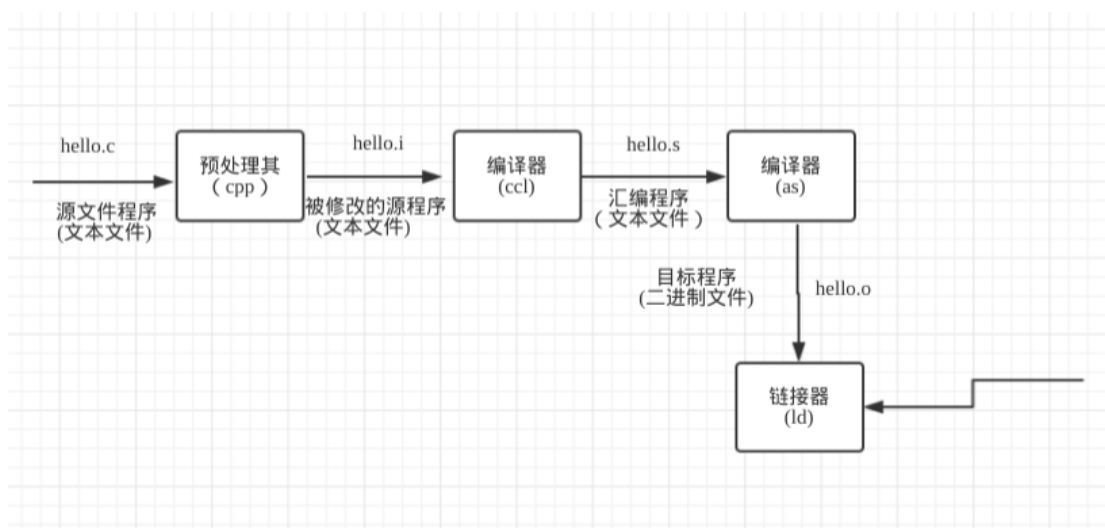
GCC 的基本指令格式为 `gcc [options] filenames`

其中 options 代表编译器所需要的编译选项，为可选参数，即可为空；filenames 为编译的文件名，支持相对路径

第二章 语言处理系统处理过程

之前在 Python 课程和 C++ 课程中老师都反复强调过，Python 属于解释型语言，而 C++ 属于编译型语言，而编译型语言最大的特点就是在代码编写完成之后需要经过编译器（比如我们之前一直使用的 VS2019）编译过程之后，才能才内存中进行加载运行操作，而本文所需要深入了解的就是这个从写完代码到生成可执行的二进制机器码的过程，也就是“编译”。

其实上面所描述的“编译”并非真正意义上的编译，在 VS2019 等集成开发环境中，“编译”其实也就是构建，即将编译与链接合并到一起进行执行。而我们对“编译”进行细化，那就是分为预处理、编译、汇编、链接，分别对应预处理器、编译器、汇编器和链接器。



linemarkers 行标代码

第一节 预处理——预编译

2.1.1 预处理过程的概述

预处理即为预编译，是在整个编译过程开始之前，也就是在编译器对源程序进行编译之前，源代码文本中所有的预处理语句会先经过预处理器的预处理，预处理过程中并不会对代码进行解析。本文中所讨论的 C++ 中的预处理器其实也就是一个宏处理器，之前 C++ 课上老师交给我们 `#include<iostream>` 等 `#` 开头的其实就是宏，也包括我们后续在数据结构课程中学习的 `#define`, `#if`, `#endif` 等，预处理器会对其 `#` 的宏命令进行替换，也就是将文本进行直接的替换，使得预处理后的文件成为一个不包含预处理指令的 C++ 文件。

之所以预处理过程存在，很大程度上是为了便捷我这种很懒又很菜的程序员，比较切合实际，比如我们不需要自己从头到尾写一个输入输出的类，而是可以直接 `#include<iostream>` 即可，同样的是，我们也可以借助 `#define` 将 `long long int` 缩写为 `ll`，对后面代码的撰写和美观程度也都很有帮助。

2.1.2 预处理语句

预处理的命令通常以 `#` 开头，独占一行，`#` 前不能有非空白符之外的符号，常用的预处理命令如下：

`#define` 定义一个预处理宏

`#undef` 取消宏的定义

`#include` 包含一个文件

`#if` 预处理语法中的条件命令

`#ifdef` 判断某个宏是否被定义，若定义，则执行后面的语句

`#ifndef` 判断某个宏是否被定义，若不被定义，则执行后面的语句

`#else` `#elif` `#endif` 与 `#ifdef` 和 `#ifndef` 定义相关

`#pragma` 说明编译器的信息

`#warning` 显示编译警告信息

`#error` 显示编译错误信息

2.1.3 预处理语句的文法

预处理并不是对整个源代码文件的全部代码进行分析，而是将源代码切分成一些小的 token，识别语句中哪些是 C++ 语句，哪些是预处理语句，预处理语句的一般格式为：

`#command name(xxx) token(s)`

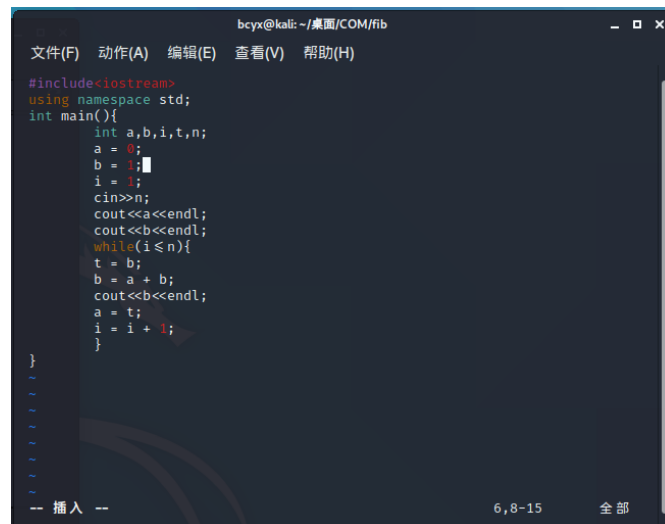
其中 `commad` 为预处理命令的名称；`name` 为宏的名称，可以带参数。

2.1.4 预处理器调用命令

Linux 中主要使用 `g++` 和 `gcc` 调用预处理器，处理文件 `fib.cpp` & `fib.c` 文件得到 `fib_cpp.i` 和 `fib_c.i` 的命令为：

`g++ -E fib.cpp -o fib.i`

`cpp fib.cpp > fib_test.i`



```
bcyx@kali: ~/桌面/COM/fib
文件(F) 动作(A) 编辑(E) 查看(V) 帮助(H)

#include<iostream>
using namespace std;
int main(){
    int a,b,i,t,n;
    a = 0;
    b = 1;
    i = 1;
    cin>>n;
    cout<<a<<endl;
    cout<<b<<endl;
    while(i<=n){
        t = b;
        b = a + b;
        cout<<b<<endl;
        a = t;
        i = i + 1;
    }
}

-- 插入 --
6,8-15 全部
```

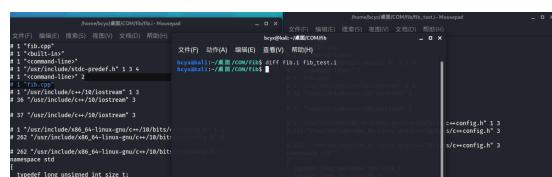
源程序 fib.cpp 代码

fib.cpp 经过两种预处理方法之后得到 fib.i 和 fib_test.i。



预处理源程序 fib.cpp

因为我们生成了 fib.i 和 fib_test.i 文件，这两个文件是一样的，也就侧面印证了上面的两种得到预处理结果指令的。



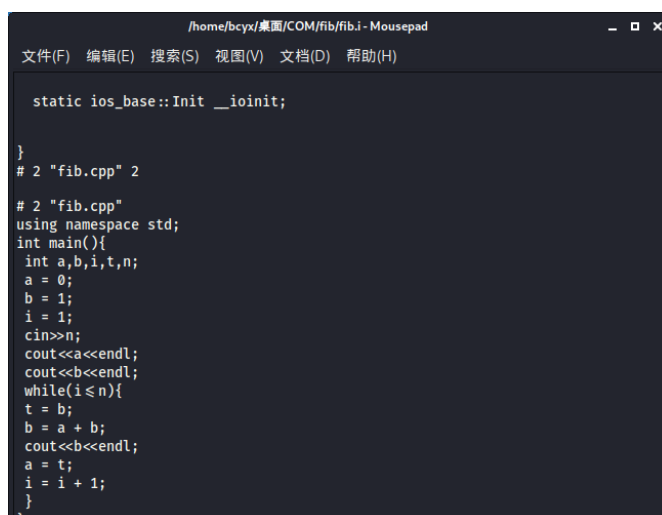
diff 指令判断

通过对输出结果的分析，我们可以看到，原来仅仅十几行的代码变成了上百万行，主要原因是预处理将 iostream 库文件的内容复制过来，因此使得代码行数看上去变得非常多。



```
/home/bcyx/桌面/COM/fib/fib.i - Mousepad
文件(F) 编辑(E) 搜索(S) 视图(V) 文档(D) 帮助(H)
# 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++/10/iostream" 1 3
# 36 "/usr/include/c++/10/iostream" 3
# 37 "/usr/include/c++/10/iostream" 3
# 1 "/usr/include/x86_64-linux-gnu/c++/10/bits/c++config.h" 1 3
# 262 "/usr/include/x86_64-linux-gnu/c++/10/bits/c++config.h" 3
# 262 "/usr/include/x86_64-linux-gnu/c++/10/bits/c++config.h" 3
namespace std
{
    typedef long unsigned int size_t;
    typedef long int ptrdiff_t;
    typedef decltype(nullptr) nullptr_t;
}
# 284 "/usr/include/x86_64-linux-gnu/c++/10/bits/c++config.h" 3
```

预处理后部分代码



```
/home/bcyx/桌面/COM/fib/fib.i - Mousepad
文件(F) 编辑(E) 搜索(S) 视图(V) 文档(D) 帮助(H)
static ios_base::Init __ioinit;
}
# 2 "fib.cpp" 2
# 2 "fib.cpp"
using namespace std;
int main(){
    int a,b,i,t,n;
    a = 0;
    b = 1;
    i = 1;
    cin>>n;
    cout<<a<<endl;
    cout<<b<<endl;
    while(i<=n){
        t = b;
        b = a + b;
        cout<<b<<endl;
        a = t;
        i = i + 1;
    }
}
```

预处理后部分代码

2.1.5 输出结果分析

通过上面的实验以及一些其它简单代码的测试，我们可以发现，预处理器在预编译过程中删除掉了一些东西，添加了一些东西，替换了一些东西，保留了一些内容。

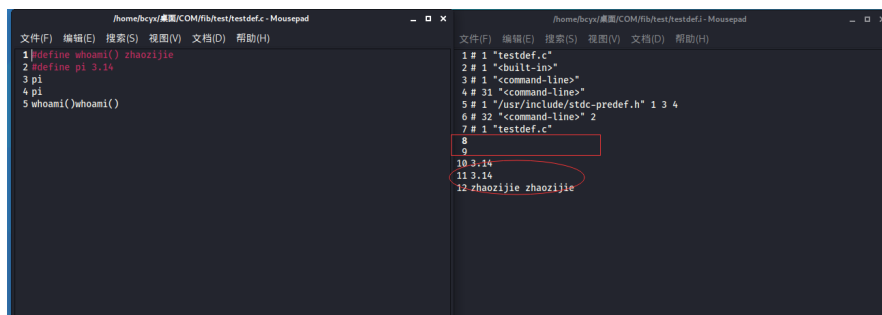
①我们可以明显发现预处理结果后的代码 fib.i 长度为 30000 多行，而生成前的 fib.cpp 长度仅仅只有 20 多行，这是因为 `#include<iostream>` 源码长度为 30000 行，在预处理过程中，对 `#include<iostream>` 预编译指令，将被包含的文件内容插入到该指令的位置，特别需要强调的是，这个过程是递归进行的，即 `#include<xxx>` 的被包含文件中可能还会存在包含其他文件，直至不存在 `#include` 为止。（特别需要注意，在处理后的 fib.i 中，`#include` 已经被取代

所以不存在了)

```
1 # 1 "fib.cpp"
2 # 1 "<built-in>"
3 # 1 "<command-line>"
4 # 1 "/usr/include/stdc-predef.h" 1 3 4
5 # 1 "<command-line>" 2
6 # 1 "fib.cpp"
7 # 1 "/usr/include/c++/10/iostream" 1 3
8 # 36 "/usr/include/c++/10/iostream" 3
9
10 # 37 "/usr/include/c++/10/iostream" 3
```

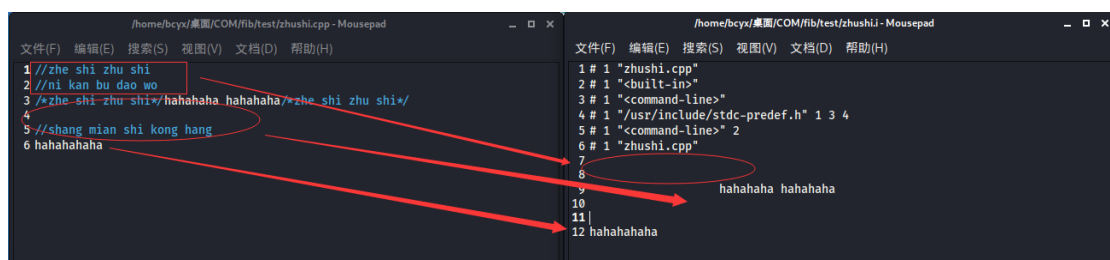
#include 内容被替代

②宏定义的替换，所有的 #define 宏定义都会被空行替换，所有使用宏定义的位置的内容都会被其实际所对应的内容所题画，例如下面的 pi 和 whoami() 都在使用时候被展开，而宏定义的位置则用空行替代，同时也可以印证在预处理器进行预处理的阶段时，虽然我们的代码明显存在没有分号等问题，但是预处理器不会关心程序代码，仅仅只是进行文本的替换。



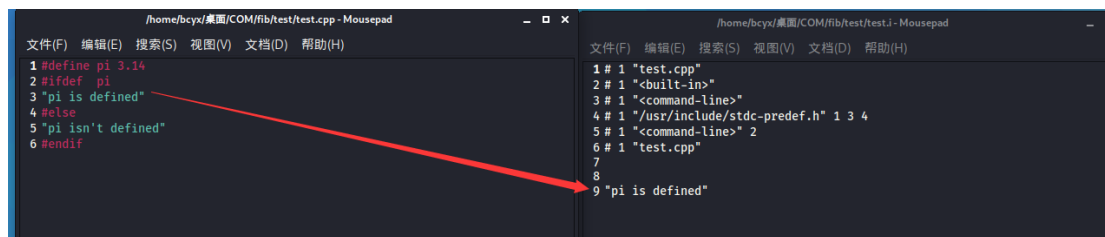
#define 宏定义

③注释和空格空行的替换，在预处理器进行预处理的过程中所有的注释语句都会被替换成空格或者空行，如果当前行全为注释语句，会替换成空行，而如果当前行存在非注释的语句，则会将当前的注释替换为空格，同样在预处理器中存在着尾部空行舍弃问题，即如果代码的尾部存在空行或空格的情况，则末尾的空行和空格会被舍弃。



注释和空格空行的替换

④预处理器会对所有的条件编译指令, 比如 `#if`, `#ifdef`, `#elif`, `#else`, `#endif` 来确定是否需要各个部分进行预处理, 以便于将源程序的代码尽可能的简化, 比如下图中的 `test.cpp` 经过预处理后, 因为我们已经对 `pi` 进行了宏定义, 所以会执行 `#ifdef` 这条路。



条件预编译指令

⑤预处理器会保留所有的 `#pragma` 指令, 因为编译器要使用这个指令。在 `#pragma` 中常用指令分别是 `#pragma once`(头文件只加载一次, 不多次加载出现冗余情况)`#pragma warning`(对一些特殊情况的报错信息进行单独处理)。

保留 `#pragma` 指令

⑥添加行号和文件名标识, 比如 `#2 "fib.cpp" 2`, 以便于编译时编译器产生他调试用的行号信息及用于编译时产生编译错误和警告的时候能够显示行号。

我们可以在 `fib.i` 中发现生成了很多如下图所示的代码, 这种类型的代码被称为行标即 `linemarkers`, 其格式遵循: `# linenum filename flags`

其中, 语句中的 `linenum` 对应的 `filename` 中的某一行的代码。

通过查阅官方文档我们可以发现 `flag` 对应的值为 1, 2, 3, 4:

1 This indicates the start of a new file.

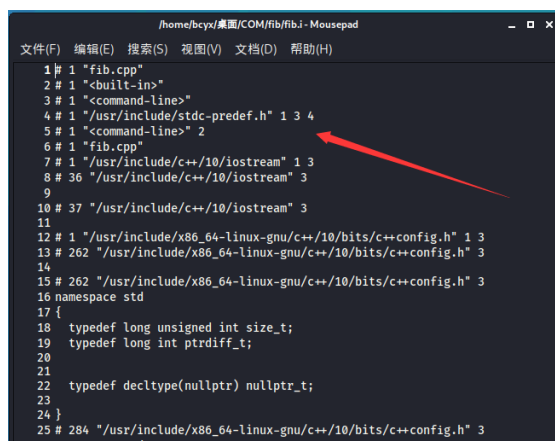
2 This indicates returning to a file (after having included another file).

3 This indicates that the following text comes from a system header file, so certain warnings should be suppressed.

4 This indicates that the following text should be treated as being wrapped in an implicit extern "C" block.

用我们蹩脚的中文翻译即为:

- 1 表示引入了一个新的文件
- 2 表示返回了一个新的文件 (该文件是包含了其他文件之后的文件)
- 3 表示以下文本来自系统头文件, 因此应该阻止某些警告
- 4 表示以下文本应被视为包裹在隐式 extern "C" 块中

A screenshot of a code editor window titled "/home/bcys/桌面/COM/fib/fib.i - Mousepad". The editor displays C++ code with line markers (line numbers and file names) at the beginning of each line. A red arrow points to line 4, which is "# 1 \"/usr/include/stdc-predef.h" 1 3 4". The code includes various system headers and defines typedefs for size_t, ptrdiff_t, and nullptr_t.

```
1 # 1 "fib.cpp"
2 # 1 "<built-in>"
3 # 1 "<command-line>"
4 # 1 "/usr/include/stdc-predef.h" 1 3 4
5 # 1 "<command-line>" 2
6 # 1 "fib.cpp"
7 # 1 "/usr/include/c++/10/iostream" 1 3
8 # 36 "/usr/include/c++/10/iostream" 3
9
10 # 37 "/usr/include/c++/10/iostream" 3
11
12 # 1 "/usr/include/x86_64-linux-gnu/c++/10/bits/c++config.h" 1 3
13 # 262 "/usr/include/x86_64-linux-gnu/c++/10/bits/c++config.h" 3
14
15 # 262 "/usr/include/x86_64-linux-gnu/c++/10/bits/c++config.h" 3
16 namespace std
17 {
18     typedef long unsigned int size_t;
19     typedef long int ptrdiff_t;
20
21
22     typedef decltype(nullptr) nullptr_t;
23
24 }
25 # 284 "/usr/include/x86_64-linux-gnu/c++/10/bits/c++config.h" 3
```

linemarkers 行标代码

第二节 重中之重——编译

2.2.1 预处理过程的概述

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对应的汇编代码文件, 这个过程也就是整个语言处理系统处理过程中最为重要的过程 (从我们的课程名称《编译系统原理》就可以看出编译占据了三分之一的地位), 同样执行编译过程的编译器也就是 GCC 的最为核心的部件, 从简而言, 编译器就是首先检查代码是否有语法错误, 确认代码无误之后, GCC 才会继续使用预处理器生成的与处理文件生成汇编文件。

2.2.2 GCC 所采用的 AT&T 格式汇编语言

在 GCC 中采用的是 AT&T 格式的汇编语言, 即我们如果要对编译器的编译过程进行分析, 我们首先要了解 GCC 编译器输出的汇编语言的语法。

因为我们无论是之前借助 VS2019 对 C++ 的学习, 亦或是汇编语言的学习, 都是主要对 Intel 和 MIPS 格式的汇编语言进行学习, 尽管汇编语言之间的宏观理念是大体相似的, 但是对于一些细节问题的关注上, AT&T 和其它还是有很大哦的区别。

①Intel 的指令使用字母的大写格式，而 AT&T 使用小写字母。

②在常规的 Intel 语法中，第一个参数常常表示目的操作数，而第二个参数表示源操作数；但是对于 AT&T 而言，第一个参数为源操作数，第二个参数则为目的操作数，即 Intel 的赋值采用从右向左的方向，而 AT&T 赋值则是从左向右的方向，更贴合我们正常语言的语法。

③AT&T 格式中寄存器名前要加上% 作为前缀，在立即操作数前需要添加\$ 作为前缀，而 Intel 不需要

	AT&T	Inter
寄存器	pushl %eax	PUSH EAX
立即数	pushl \$1	PUSH 1

④在 AT&T 汇编格式中，操作数的字长由操作符的最后一个字母决定，后缀'b'、'w'、'l' 分别表示操作数为字节（byte，8 比特）、字（word，16 比特）和长字（long，32 比特）；而在 Intel 汇编格式中，操作数的字长是用"byte ptr"和"word ptr" 等前缀来表示的。

	AT&T	Intel
操作数位置	addl \$1,%eax	add eax,1
操作数字长	movb val,%al	mov al,byte ptr val

⑤在 AT&T 汇编格式中，绝对专一和调用指令的操作数前都要加上" 作为前缀

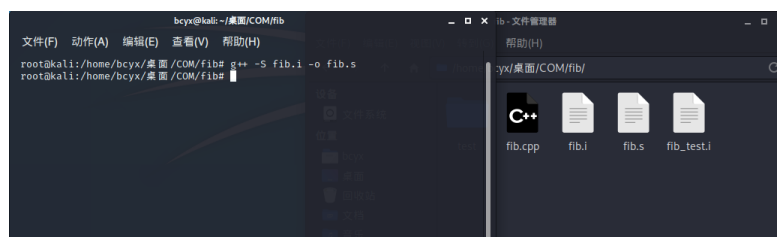
⑥在 AT&T 汇编格式中内存操作数的寻址方式是：section:disp(base, index, scale),而 Intel 汇编格式中,内存操作数的寻址方式为:section:[base + index*scale + disp]。

2.2.3 编译器调用指令

Linux 可以使用下面这条指令将预处理后生成的 fib.i 文件编译后得到 fib.s:

```
g++ -S fib.i -o fib.s
```

其中参数-S 代表 gcc 只生成汇编源代码而不进行汇编。



生成 fib.s 编译结果文件

fib.s 文件即为汇编器的输入文件

因为现在版本的 GCC 将预编译和编译这两个步骤合并成了一个步骤，使用了一个叫做 cc1 的程序来实现这两个步骤，这个 cc1plus 程序位于“/usr/lib/gcc/x86_64-linux-gnu/10”里面（因 Linux 系统版本而异），因此我们可以直接调用 cc1plus 来实现对 fib.cpp 的编译和预编译。

```
/usr/lib/gcc/x86_64-linux-gnu/10/cc1plus fib.c
```

上下文无关语法

2.2.4 编译器过程详解

编译器作为最重要过程——编译的执行器，自然我要用很大的篇幅去较为详细的去解释其编译的过程（主要是可以顺便温习一下前两章的预习知识）

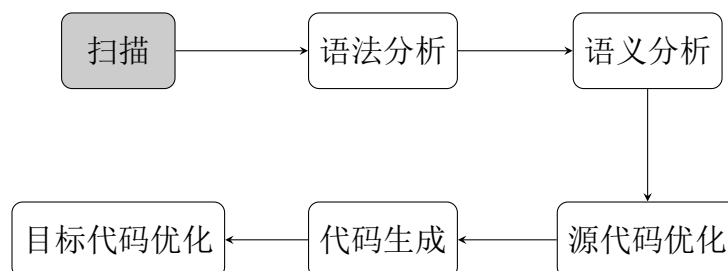


图 1: 编译过程分析

①词法分析

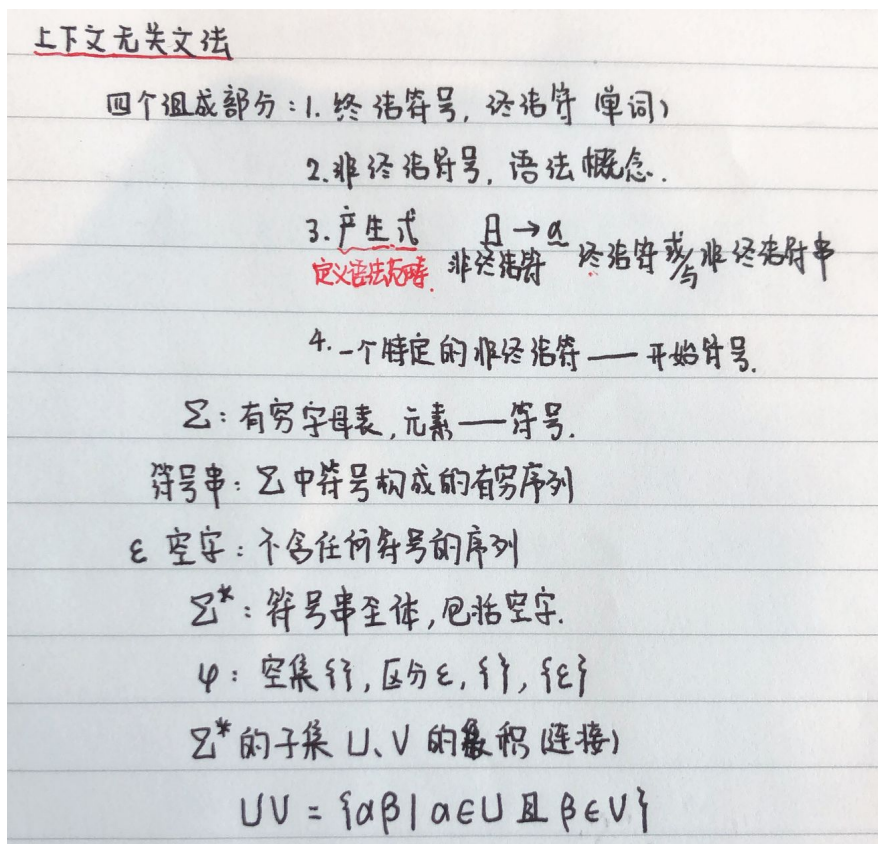
首先源代码程序会被输入到扫描器,扫描器的工作也十分的简单,就是运用类似有限状态机的算法将源代码的字符序列分割成一系列的记号,也就是 token。

词法分析产生的记号一般分为以下几类:关键字、标识符、字面常量(包括数字、字符串等)和特殊符号(加减乘除的符号等),在识别记号的同时,扫描器其实也完成了一些其它的工作,比如将标识符放到符号表,将数字和字符串常量存放到文字表等,以备后续编译的继续使用。

课上(预习 PPT 音频)中王老师提到的 lex 程序——一个可以实现此法扫描的程序,它会按照用户之前描述好的词法规则将输入的字符串分割成一个个记号,因为类似 lex 的程序存在,编译器的开发者就没必要为每一个编译器开发一个独立的词法扫描器,而是根据不同的需要对词法规则进行修改即可

②语法分析

扫描器后的语法分析器就会对其产生的记号进行语法分析,从而生成语法树,整个过程采用了上下文无关语法,关于上下文无关语法的介绍见下图个人笔记:



上下文无关语法

简单来讲,由语法生成器生成的语法树就是以表达式为节点的树,主要是从

层次关系的层面去对单词流生成的单词之间的关系进行揭示。

③语义分析

语义分析主要有语义分析器来完成，语法分析仅仅是完成了对表达式的语法层面的分析，但是语法分析并不能确定这个语句是否真正的有意义，比如我们在 cpp 中对两个指针做乘法运算是肯定没有意义的，但是语法分析过程中，在语法层面它是合法的。

编译器所能分析的语义是静态语义，所谓静态语义是指在编译器就可以确定的语义，与之对应的动态语义就是只能在运行期才能确定的语义。

静态语义通常包括声明和类型的匹配，类型的转换。比如当一个浮点型的表达式赋值给一个整型的表达式时，其中隐含了一个浮点型强制转换为整型的过程，而语义分析就是要完成这个步骤，而如果发现将一个浮点型赋值给一个指针变量的时候，语义分析程序就会发现这个类型不匹配，编译器就会报错。而动态语义一般是在运行过程中出现的语义相关的问题，比如将 0 作为除数就是一个运行期的语义错误。

经过语义分析阶段过程后，整个语法树的表达式都被标识了类型，如果有些类型需要做隐式转换，语义分析程序会在语法树中插入相应的转换节点。语义分析器也会对符号表中的符号类型做了更新。

④中间语言生成

因为优化工作在语法树上直接进行优化比较困难，比如将 $(2+6)$ 直接改成 8，因此源代码优化器往往将整个语法树转换成中间代码，它其实就是语法树的顺序表示，其实已经非常接近目标代码了，但是它一般是跟目标机器和运行时候的环境无关，比如不包含数据的尺寸、变量地址等，中间代码有很多类型，我们使用的是三地址码，最基本的三地址码是这样的： $x = y \text{ op } z$

这个三地址码表示将变量 y 和 z 进行 op 操作以后，赋值给 x 。中间代码使得编译器可以被分成前端和后端，编译器前端负责产生机器无关的中间代码，编译器后端将中间代码转换成目标机器代码。

⑤目标代码生成与优化

源代码级优化器产生中间代码标志着后面的过程都属于编译器后端，编译器后端主要包括了代码生成器和目标代码优化器。

代码生成器将中间代码转换成目标机器代码，这个过程依赖于目标机器，因为不同机器对应着不同的寄存器、不同的字长甚至不同的数据类型。

而后目标代码优化器会对上述代码生成器产生的目标代码进行优化，比如

寻找合适的寻址方式、使用位移来代替惩罚运算、删除一些多余的指令等。

在看预习 PPT 的音频时候，我开始一直不理解，不管我是扫描、语法分析、语义分析、源代码优化、代码生成还是目标代码的优化，一个编译器干了这么多活之后，编译生成了目标代码，但是一直有一个问题，之前我用 IDA 打开各种程序，都会发现不管是临时变量 `i`，还是常量 `a`，都拥有自己的地址，那么我如果想要执行一个程序，必须要有地址，那么这个地址在哪里设置的呢？我开始是认为如果源代码都在同一个编译单元里，那么我们顺序赋址即可，但是如果我是调用的别的编译单元、程序模块的变量呢？

(开始没看到链接器的时候，这个地址问题确实让我很疑惑，看到链接器之后就大致了解了整个过程，后面在链接器链接过程会详述)

2.2.5 编译器优化处理

①编译优化概述

编译阶段会对代码进行优化处理，它涉及到的问题不仅仅和编译技术本身有关，同时和机器的硬件环境也有很大的关系，他一般分为两部分，一部分是对中间代码的优化，不依赖于机器，也就是上文中提到的源代码优化器，而另一部分则是主要针对目标代码的生成而进行的。

对于不依赖于计算机及其的优化，主要是删除公共表达式、循环优化(代码外提，强度削弱，变换循环控制和已知量的合并等)、无用赋值的删除等。

对于依赖机器的优化，主要是考虑如何充分利用机器的硬件寄存器存放相关的变量以减少内存的访问次数，根据一些机器的特点调整使得目标代码更简洁，效率更高。

②-O 优化选项简介

-O 选项可以使编译器对代码进行自动优化编译，输出效率更高的可执行文件。-O 一般后面跟上数字来指定优化的级别，比如-O0、-O1 等，没有参数默认为-1，最高可以为 3。优化级别越高，产生的代码的执行效率越高，但是编译的过程花费的时间也就越长。

③-O0 关闭优化

设置-O0 将关闭所有的优化选项，对应的指令为：

```
g++ -O0 -S fib.i -o fib_0.s
```

④-O1 一级优化

编译器会尝试减少代码的体积和代码运行时间，但不会执行会花费大量时

```

1 | .file "fib.cpp"
2 | .text
3 | .section .rodata
4 | .type __ZStL19piecewise_construct, @object
5 | .size __ZStL19piecewise_construct, 1
6 | __ZStL19piecewise_construct:
7 | .zero 1
8 | .local __ZStL8__init
9 | .comm __ZStL8__init,1,1
10 | .text
11 | .globl main
12 | .type main, @function
13 | main:
14 | .LFB1572:
15 | .cfi_startproc
16 | pushq %rbp
17 | .cfi_def_cfa_offset 16
18 | .cfi_offset 6, -16
19 | movq %rsp, %rbp
20 | .cfi_def_cfa_register 6
21 | subq $12, %rsp
22 | movl $0, -(%rbp)
23 | movl $1, -4(%rbp)
24 | movl $1, -12(%rbp)
25 | leaq -20(%rbp), %rax

```

O0 优化后 fib_0.s 的部分内容

间优化代码的操作，对应的指令为：

g++ -O1 -S fib.i -o fib_1.s

```

1 | .file "fib.cpp"
2 | .text
3 | .globl main
4 | .type main, @function
5 | main:
6 | .LFB1573:
7 | .cfi_startproc
8 | pushq %r15
9 | .cfi_def_cfa_offset 16
10 | .cfi_offset 15, -16
11 | pushq %r14
12 | .cfi_def_cfa_offset 24
13 | .cfi_offset 14, -24
14 | pushq %r13
15 | .cfi_def_cfa_offset 32
16 | .cfi_offset 13, -32
17 | pushq %r12
18 | .cfi_def_cfa_offset 40
19 | .cfi_offset 12, -40
20 | pushq %rbp
21 | .cfi_def_cfa_offset 48
22 | .cfi_offset 8, -48
23 | pushq %rax
24 | .cfi_def_cfa_offset 56

```

O1 优化后 fib_1.s 的部分内容

我们可以借助 diff 查看二者的区别

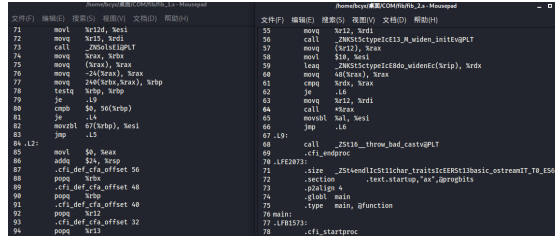
```

1 | .cfa_offset 8, -48
2 | movq %rax, %r15
3 | .cfa_offset 16, -32
4 | movq %r15, %r14
5 | .cfa_offset 24, -24
6 | movl $0, -(%rbp)
7 | movl $1, -4(%rbp)
8 | movl $1, -12(%rbp)
9 | leaq -20(%rbp), %rax
10 | .LFB1572:
11 | .cfa_offset 16, -16
12 | .cfa_offset 24, -24
13 | .cfa_offset 32, -32
14 | .cfa_offset 40, -40
15 | .cfa_offset 48, -48
16 | .cfa_offset 56, -56
17 | .cfa_offset 64, -64
18 | .cfa_offset 72, -72
19 | .cfa_offset 80, -80
20 | .cfa_offset 88, -88
21 | .cfa_offset 96, -96
22 | .cfa_offset 104, -104
23 | .cfa_offset 112, -112
24 | .cfa_offset 120, -120
25 | .cfa_offset 128, -128
26 | .cfa_offset 136, -136
27 | .cfa_offset 144, -144
28 | .cfa_offset 152, -152
29 | .cfa_offset 160, -160
30 | .cfa_offset 168, -168
31 | .cfa_offset 176, -176
32 | .cfa_offset 184, -184
33 | .cfa_offset 192, -192
34 | .cfa_offset 200, -200
35 | .cfa_offset 208, -208
36 | .cfa_offset 216, -216
37 | .cfa_offset 224, -224
38 | .cfa_offset 232, -232
39 | .cfa_offset 240, -240
40 | .cfa_offset 248, -248
41 | .cfa_offset 256, -256
42 | .cfa_offset 264, -264
43 | .cfa_offset 272, -272
44 | .cfa_offset 280, -280
45 | .cfa_offset 288, -288
46 | .cfa_offset 296, -296
47 | .cfa_offset 304, -304
48 | .cfa_offset 312, -312
49 | .cfa_offset 320, -320
50 | .cfa_offset 328, -328
51 | .cfa_offset 336, -336
52 | .cfa_offset 344, -344
53 | .cfa_offset 352, -352
54 | .cfa_offset 360, -360
55 | .cfa_offset 368, -368
56 | .cfa_offset 376, -376
57 | .cfa_offset 384, -384
58 | .cfa_offset 392, -392
59 | .cfa_offset 400, -400
60 | .cfa_offset 408, -408
61 | .cfa_offset 416, -416
62 | .cfa_offset 424, -424
63 | .cfa_offset 432, -432
64 | .cfa_offset 440, -440
65 | .cfa_offset 448, -448
66 | .cfa_offset 456, -456
67 | .cfa_offset 464, -464
68 | .cfa_offset 472, -472
69 | .cfa_offset 480, -480
70 | .cfa_offset 488, -488
71 | .cfa_offset 496, -496
72 | .cfa_offset 504, -504
73 | .cfa_offset 512, -512
74 | .cfa_offset 520, -520
75 | .cfa_offset 528, -528
76 | .cfa_offset 536, -536
77 | .cfa_offset 544, -544
78 | .cfa_offset 552, -552
79 | .cfa_offset 560, -560
80 | .cfa_offset 568, -568
81 | .cfa_offset 576, -576
82 | .cfa_offset 584, -584
83 | .cfa_offset 592, -592
84 | .cfa_offset 600, -600
85 | .cfa_offset 608, -608
86 | .cfa_offset 616, -616
87 | .cfa_offset 624, -624
88 | .cfa_offset 632, -632
89 | .cfa_offset 640, -640
90 | .cfa_offset 648, -648
91 | .cfa_offset 656, -656
92 | .cfa_offset 664, -664
93 | .cfa_offset 672, -672
94 | .cfa_offset 680, -680
95 | .cfa_offset 688, -688
96 | .cfa_offset 696, -696
97 | .cfa_offset 704, -704
98 | .cfa_offset 712, -712
99 | .cfa_offset 720, -720
100 | .cfa_offset 728, -728
101 | .cfa_offset 736, -736
102 | .cfa_offset 744, -744
103 | .cfa_offset 752, -752
104 | .cfa_offset 760, -760
105 | .cfa_offset 768, -768
106 | .cfa_offset 776, -776
107 | .cfa_offset 784, -784
108 | .cfa_offset 792, -792
109 | .cfa_offset 800, -800
110 | .cfa_offset 808, -808
111 | .cfa_offset 816, -816
112 | .cfa_offset 824, -824
113 | .cfa_offset 832, -832
114 | .cfa_offset 840, -840
115 | .cfa_offset 848, -848
116 | .cfa_offset 856, -856
117 | .cfa_offset 864, -864
118 | .cfa_offset 872, -872
119 | .cfa_offset 880, -880
120 | .cfa_offset 888, -888
121 | .cfa_offset 896, -896
122 | .cfa_offset 904, -904
123 | .cfa_offset 912, -912
124 | .cfa_offset 920, -920
125 | .cfa_offset 928, -928
126 | .cfa_offset 936, -936
127 | .cfa_offset 944, -944
128 | .cfa_offset 952, -952
129 | .cfa_offset 960, -960
130 | .cfa_offset 968, -968
131 | .cfa_offset 976, -976
132 | .cfa_offset 984, -984
133 | .cfa_offset 992, -992
134 | .cfa_offset 1000, -1000
135 | .cfa_offset 1008, -1008
136 | .cfa_offset 1016, -1016
137 | .cfa_offset 1024, -1024
138 | .cfa_offset 1032, -1032
139 | .cfa_offset 1040, -1040
140 | .cfa_offset 1048, -1048
141 | .cfa_offset 1056, -1056
142 | .cfa_offset 1064, -1064
143 | .cfa_offset 1072, -1072
144 | .cfa_offset 1080, -1080
145 | .cfa_offset 1088, -1088
146 | .cfa_offset 1096, -1096
147 | .cfa_offset 1104, -1104
148 | .cfa_offset 1112, -1112
149 | .cfa_offset 1120, -1120
150 | .cfa_offset 1128, -1128
151 | .cfa_offset 1136, -1136
152 | .cfa_offset 1144, -1144
153 | .cfa_offset 1152, -1152
154 | .cfa_offset 1160, -1160
155 | .cfa_offset 1168, -1168
156 | .cfa_offset 1176, -1176
157 | .cfa_offset 1184, -1184
158 | .cfa_offset 1192, -1192
159 | .cfa_offset 1200, -1200
160 | .cfa_offset 1208, -1208
161 | .cfa_offset 1216, -1216
162 | .cfa_offset 1224, -1224
163 | .cfa_offset 1232, -1232
164 | .cfa_offset 1240, -1240
165 | .cfa_offset 1248, -1248
166 | .cfa_offset 1256, -1256
167 | .cfa_offset 1264, -1264
168 | .cfa_offset 1272, -1272
169 | .cfa_offset 1280, -1280
170 | .cfa_offset 1288, -1288
171 | .cfa_offset 1296, -1296
172 | .cfa_offset 1304, -1304
173 | .cfa_offset 1312, -1312
174 | .cfa_offset 1320, -1320
175 | .cfa_offset 1328, -1328
176 | .cfa_offset 1336, -1336
177 | .cfa_offset 1344, -1344
178 | .cfa_offset 1352, -1352
179 | .cfa_offset 1360, -1360
180 | .cfa_offset 1368, -1368
181 | .cfa_offset 1376, -1376
182 | .cfa_offset 1384, -1384
183 | .cfa_offset 1392, -1392
184 | .cfa_offset 1400, -1400
185 | .cfa_offset 1408, -1408
186 | .cfa_offset 1416, -1416
187 | .cfa_offset 1424, -1424
188 | .cfa_offset 1432, -1432
189 | .cfa_offset 1440, -1440
190 | .cfa_offset 1448, -1448
191 | .cfa_offset 1456, -1456
192 | .cfa_offset 1464, -1464
193 | .cfa_offset 1472, -1472
194 | .cfa_offset 1480, -1480
195 | .cfa_offset 1488, -1488
196 | .cfa_offset 1496, -1496
197 | .cfa_offset 1504, -1504
198 | .cfa_offset 1512, -1512
199 | .cfa_offset 1520, -1520
200 | .cfa_offset 1528, -1528
201 | .cfa_offset 1536, -1536
202 | .cfa_offset 1544, -1544
203 | .cfa_offset 1552, -1552
204 | .cfa_offset 1560, -1560
205 | .cfa_offset 1568, -1568
206 | .cfa_offset 1576, -1576
207 | .cfa_offset 1584, -1584
208 | .cfa_offset 1592, -1592
209 | .cfa_offset 1600, -1600
210 | .cfa_offset 1608, -1608
211 | .cfa_offset 1616, -1616
212 | .cfa_offset 1624, -1624
213 | .cfa_offset 1632, -1632
214 | .cfa_offset 1640, -1640
215 | .cfa_offset 1648, -1648
216 | .cfa_offset 1656, -1656
217 | .cfa_offset 1664, -1664
218 | .cfa_offset 1672, -1672
219 | .cfa_offset 1680, -1680
220 | .cfa_offset 1688, -1688
221 | .cfa_offset 1696, -1696
222 | .cfa_offset 1704, -1704
223 | .cfa_offset 1712, -1712
224 | .cfa_offset 1720, -1720
225 | .cfa_offset 1728, -1728
226 | .cfa_offset 1736, -1736
227 | .cfa_offset 1744, -1744
228 | .cfa_offset 1752, -1752
229 | .cfa_offset 1760, -1760
230 | .cfa_offset 1768, -1768
231 | .cfa_offset 1776, -1776
232 | .cfa_offset 1784, -1784
233 | .cfa_offset 1792, -1792
234 | .cfa_offset 1800, -1800
235 | .cfa_offset 1808, -1808
236 | .cfa_offset 1816, -1816
237 | .cfa_offset 1824, -1824
238 | .cfa_offset 1832, -1832
239 | .cfa_offset 1840, -1840
240 | .cfa_offset 1848, -1848
241 | .cfa_offset 1856, -1856
242 | .cfa_offset 1864, -1864
243 | .cfa_offset 1872, -1872
244 | .cfa_offset 1880, -1880
245 | .cfa_offset 1888, -1888
246 | .cfa_offset 1896, -1896
247 | .cfa_offset 1904, -1904
248 | .cfa_offset 1912, -1912
249 | .cfa_offset 1920, -1920
250 | .cfa_offset 1928, -1928
251 | .cfa_offset 1936, -1936
252 | .cfa_offset 1944, -1944
253 | .cfa_offset 1952, -1952
254 | .cfa_offset 1960, -1960
255 | .cfa_offset 1968, -1968
256 | .cfa_offset 1976, -1976
257 | .cfa_offset 1984, -1984
258 | .cfa_offset 1992, -1992
259 | .cfa_offset 2000, -2000
260 | .cfa_offset 2008, -2008
261 | .cfa_offset 2016, -2016
262 | .cfa_offset 2024, -2024
263 | .cfa_offset 2032, -2032
264 | .cfa_offset 2040, -2040
265 | .cfa_offset 2048, -2048
266 | .cfa_offset 2056, -2056
267 | .cfa_offset 2064, -2064
268 | .cfa_offset 2072, -2072
269 | .cfa_offset 2080, -2080
270 | .cfa_offset 2088, -2088
271 | .cfa_offset 2096, -2096
272 | .cfa_offset 2104, -2104
273 | .cfa_offset 2112, -2112
274 | .cfa_offset 2120, -2120
275 | .cfa_offset 2128, -2128
276 | .cfa_offset 2136, -2136
277 | .cfa_offset 2144, -2144
278 | .cfa_offset 2152, -2152
279 | .cfa_offset 2160, -2160
280 | .cfa_offset 2168, -2168
281 | .cfa_offset 2176, -2176
282 | .cfa_offset 2184, -2184
283 | .cfa_offset 2192, -2192
284 | .cfa_offset 2200, -2200
285 | .cfa_offset 2208, -2208
286 | .cfa_offset 2216, -2216
287 | .cfa_offset 2224, -2224
288 | .cfa_offset 2232, -2232
289 | .cfa_offset 2240, -2240
290 | .cfa_offset 2248, -2248
291 | .cfa_offset 2256, -2256
292 | .cfa_offset 2264, -2264
293 | .cfa_offset 2272, -2272
294 | .cfa_offset 2280, -2280
295 | .cfa_offset 2288, -2288
296 | .cfa_offset 2296, -2296
297 | .cfa_offset 2304, -2304
298 | .cfa_offset 2312, -2312
299 | .cfa_offset 2320, -2320
300 | .cfa_offset 2328, -2328
301 | .cfa_offset 2336, -2336
302 | .cfa_offset 2344, -2344
303 | .cfa_offset 2352, -2352
304 | .cfa_offset 2360, -2360
305 | .cfa_offset 2368, -2368
306 | .cfa_offset 2376, -2376
307 | .cfa_offset 2384, -2384
308 | .cfa_offset 2392, -2392
309 | .cfa_offset 2400, -2400
310 | .cfa_offset 2408, -2408
311 | .cfa_offset 2416, -2416
312 | .cfa_offset 2424, -2424
313 | .cfa_offset 2432, -2432
314 | .cfa_offset 2440, -2440
315 | .cfa_offset 2448, -2448
316 | .cfa_offset 2456, -2456
317 | .cfa_offset 2464, -2464
318 | .cfa_offset 2472, -2472
319 | .cfa_offset 2480, -2480
320 | .cfa_offset 2488, -2488
321 | .cfa_offset 2496, -2496
322 | .cfa_offset 2504, -2504
323 | .cfa_offset 2512, -2512
324 | .cfa_offset 2520, -2520
325 | .cfa_offset 2528, -2528
326 | .cfa_offset 2536, -2536
327 | .cfa_offset 2544, -2544
328 | .cfa_offset 2552, -2552
329 | .cfa_offset 2560, -2560
330 | .cfa_offset 2568, -2568
331 | .cfa_offset 2576, -2576
332 | .cfa_offset 2584, -2584
333 | .cfa_offset 2592, -2592
334 | .cfa_offset 2600, -2600
335 | .cfa_offset 2608, -2608
336 | .cfa_offset 2616, -2616
337 | .cfa_offset 2624, -2624
338 | .cfa_offset 2632, -2632
339 | .cfa_offset 2640, -2640
340 | .cfa_offset 2648, -2648
341 | .cfa_offset 2656, -2656
342 | .cfa_offset 2664, -2664
343 | .cfa_offset 2672, -2672
344 | .cfa_offset 2680, -2680
345 | .cfa_offset 2688, -2688
346 | .cfa_offset 2696, -2696
347 | .cfa_offset 2704, -2704
348 | .cfa_offset 2712, -2712
349 | .cfa_offset 2720, -2720
350 | .cfa_offset 2728, -2728
351 | .cfa_offset 2736, -2736
352 | .cfa_offset 2744, -2744
353 | .cfa_offset 2752, -2752
354 | .cfa_offset 2760, -2760
355 | .cfa_offset 2768, -2768
356 | .cfa_offset 2776, -2776
357 | .cfa_offset 2784, -2784
358 | .cfa_offset 2792, -2792
359 | .cfa_offset 2800, -2800
360 | .cfa_offset 2808, -2808
361 | .cfa_offset 2816, -2816
362 | .cfa_offset 2824, -2824
363 | .cfa_offset 2832, -2832
364 | .cfa_offset 2840, -2840
365 | .cfa_offset 2848, -2848
366 | .cfa_offset 2856, -2856
367 | .cfa_offset 2864, -2864
368 | .cfa_offset 2872, -2872
369 | .cfa_offset 2880, -2880
370 | .cfa_offset 2888, -2888
371 | .cfa_offset 2896, -2896
372 | .cfa_offset 2904, -2904
373 | .cfa_offset 2912, -2912
374 | .cfa_offset 2920, -2920
375 | .cfa_offset 2928, -2928
376 | .cfa_offset 2936, -2936
377 | .cfa_offset 2944, -2944
378 | .cfa_offset 2952, -2952
379 | .cfa_offset 2960, -2960
380 | .cfa_offset 2968, -2968
381 | .cfa_offset 2976, -2976
382 | .cfa_offset 2984, -2984
383 | .cfa_offset 2992, -2992
384 | .cfa_offset 3000, -3000
385 | .cfa_offset 3008, -3008
386 | .cfa_offset 3016, -3016
387 | .cfa_offset 3024, -3024
388 | .cfa_offset 3032, -3032
389 | .cfa_offset 3040, -3040
390 | .cfa_offset 3048, -3048
391 | .cfa_offset 3056, -3056
392 | .cfa_offset 3064, -3064
393 | .cfa_offset 3072, -3072
394 | .cfa_offset 3080, -3080
395 | .cfa_offset 3088, -3088
396 | .cfa_offset 3096, -3096
397 | .cfa_offset 3104, -3104
398 | .cfa_offset 3112, -3112
399 | .cfa_offset 3120, -3120
400 | .cfa_offset 3128, -3128
401 | .cfa_offset 3136, -3136
402 | .cfa_offset 3144, -3144
403 | .cfa_offset 3152, -3152
404 | .cfa_offset 3160, -3160
405 | .cfa_offset 3168, -3168
406 | .cfa_offset 3176, -3176
407 | .cfa_offset 3184, -3184
408 | .cfa_offset 3192, -3192
409 | .cfa_offset 3200, -3200
410 | .cfa_offset 3208, -3208
411 | .cfa_offset 3216, -3216
412 | .cfa_offset 3224, -3224
413 | .cfa_offset 3232, -3232
414 | .cfa_offset 3240, -3240
415 | .cfa_offset 3248, -3248
416 | .cfa_offset 3256, -3256
417 | .cfa_offset 3264, -3264
418 | .cfa_offset 3272, -3272
419 | .cfa_offset 3280, -3280
420 | .cfa_offset 3288, -3288
421 | .cfa_offset 3296, -3296
422 | .cfa_offset 3304, -3304
423 | .cfa_offset 3312, -3312
424 | .cfa_offset 3320, -3320
425 | .cfa_offset 3328, -3328
426 | .cfa_offset 3336, -3336
427 | .cfa_offset 3344, -3344
428 | .cfa_offset 3352, -3352
429 | .cfa_offset 3360, -3360
430 | .cfa_offset 3368, -3368
431 | .cfa_offset 3376, -3376
432 | .cfa_offset 3384, -3384
433 | .cfa_offset 3392, -3392
434 | .cfa_offset 3400, -3400
435 | .cfa_offset 3408, -3408
436 | .cfa_offset 3416, -3416
437 | .cfa_offset 3424, -3424
438 | .cfa_offset 3432, -3432
439 | .cfa_offset 3440, -3440
440 | .cfa_offset 3448, -3448
441 | .cfa_offset 3456, -3456
442 | .cfa_offset 3464, -3464
443 | .cfa_offset 3472, -3472
444 | .cfa_offset 3480, -3480
445 | .cfa_offset 3488, -3488
446 | .cfa_offset 3496, -3496
447 | .cfa_offset 3504, -3504
448 | .cfa_offset 3512, -3512
449 | .cfa_offset 3520, -3520
450 | .cfa_offset 3528, -3528
451 | .cfa_offset 3536, -3536
452 | .cfa_offset 3544, -3544
453 | .cfa_offset 3552, -3552
454 | .cfa_offset 3560, -3560
455 | .cfa_offset 3568, -3568
456 | .cfa_offset 3576, -3576
457 | .cfa_offset 3584, -3584
458 | .cfa_offset 3592, -3592
459 | .cfa_offset 3600, -3600
460 | .cfa_offset 3608, -3608
461 | .cfa_offset 3616, -3616
462 | .cfa_offset 3624, -3624
463 | .cfa_offset 3632, -3632
464 | .cfa_offset 3640, -3640
465 | .cfa_offset 3648, -3648
466 | .cfa_offset 3656, -3656
467 | .cfa_offset 3664, -3664
468 | .cfa_offset 3672, -3672
469 | .cfa_offset 3680, -3680
470 | .cfa_offset 3688, -3688
471 | .cfa_offset 3696, -3696
472 | .cfa_offset 3704, -3704
473 | .cfa_offset 3712, -3712
474 | .cfa_offset 3720, -3720
475 | .cfa_offset 3728, -3728
476 | .cfa_offset 3736, -3736
477 | .cfa_offset 3744, -3744
478 | .cfa_offset 3752, -3752
479 | .cfa_offset 3760, -3760
480 | .cfa_offset 3768, -3768
481 | .cfa_offset 3776, -3776
482 | .cfa_offset 3784, -3784
483 | .cfa_offset 3792, -3792
484 | .cfa_offset 3800, -3800
485 | .cfa_offset 3808, -3808
486 | .cfa_offset 3816, -3816
487 | .cfa_offset 3824, -3824
488 | .cfa_offset 3832, -3832
489 | .cfa_offset 3840, -3840
490 | .cfa_offset 3848, -3848
491 | .cfa_offset 3856, -3856
492 | .cfa_offset 3864, -3864
493 | .cfa_offset 3872, -3872
494 | .cfa_offset 3880, -3880
495 | .cfa_offset 3888, -3888
496 | .cfa_offset 3896, -3896
497 | .cfa_offset 3904, -3904
498 | .cfa_offset 3912, -3912
499 | .cfa_offset 3920, -3920
500 | .cfa_offset 3928, -3928
501 | .cfa_offset 3936, -3936
502 | .cfa_offset 3944, -3944
503 | .cfa_offset 3952, -3952
504 | .cfa_offset 3960, -3960
505 | .cfa_offset 3968, -3968
506 | .cfa_offset 3976, -3976
507 | .cfa_offset 3984, -3984
508 | .cfa_offset 3992, -3992
509 | .cfa_offset 4000, -4000
510 | .cfa_offset 4008, -4008
511 | .cfa_offset 4016, -4016
512 | .cfa_offset 4024, -4024
513 | .cfa_offset 4032, -4032
514 | .cfa_offset 4040, -4040
515 | .cfa_offset 4048, -4048
516 | .cfa_offset 4056, -4056
517 | .cfa_offset 4064, -4064
518 | .cfa_offset 4072, -4072
519 | .cfa_offset 4080, -4080
520 | .cfa_offset 4088, -4088
521 | .cfa_offset 4096, -4096
522 | .cfa_offset 4104, -4104
523 | .cfa_offset 4112, -4112
524 | .cfa_offset 4120, -4120
525 | .cfa_offset 4128, -4128
526 | .cfa_offset 4136, -4136
527 | .cfa_offset 4144, -4144
528 | .cfa_offset 4152, -4152
529 | .cfa_offset 4160, -4160
530 | .cfa_offset 4168, -4168
531 | .cfa_offset 4176, -4176
532 | .cfa_offset 4184, -4184
533 | .cfa_offset 4192, -4192
534 | .cfa_offset 4200, -4200
535 | .cfa_offset 4208, -4208
536 | .cfa_offset 4216, -4216
537 | .cfa_offset 4224, -4224
538 | .cfa_offset 4232, -4232
539 | .cfa_offset 4240, -4240
540 | .cfa_offset 4248, -4248
541 | .cfa_offset 4256, -4256
542 | .cfa_offset 4264, -4264
543 | .cfa_offset 4272, -4272
544 | .cfa_offset 4280, -4280
545 | .cfa_offset 4288, -4288
546 | .cfa_offset 4296, -4296
547 | .cfa_offset 4304, -4304
548 | .cfa_offset 4312, -4312
549 | .cfa_offset 4320, -4320
550 | .cfa_offset 4328, -4328
551 | .cfa_offset 4336, -4336
552 | .cfa_offset 4344, -4344
553 | .cfa_offset 4352, -4352
554 | .cfa_offset 4360, -4360
555 | .cfa_offset 4368, -4368
556 | .cfa_offset 4376, -4376
557 | .cfa_offset 4384, -4384
558 | .cfa_offset 4392, -4392
559 | .cfa_offset 4400, -4400
560 | .cfa_offset 4408, -4408
561 | .cfa_offset 4416, -4416
562 | .cfa_offset 4424, -4424
563 | .cfa_offset 4432, -4432
564 | .cfa_offset 4440, -4440
565 | .cfa_offset 4448, -4448
566 | .cfa_offset 4456, -4456
567 | .cfa_offset 4464, -4464
568 | .cfa_offset 4472, -4472
569 | .cfa_offset 4480, -4480
570 | .cfa_offset 4488, -4488
571 | .cfa_offset 4496, -4496
572 | .cfa_offset 4504, -4504
573 | .cfa_offset 4512, -4512
574 | .cfa_offset 4520, -4520
575 | .cfa_offset 4528, -4528
576 | .cfa_offset 4536, -4536
577 | .cfa_offset 4544, -4544
578 | .cfa_offset 4552, -4552
579 | .cfa_offset 4560, -4560
580 | .cfa_offset 4568, -4568
581 | .cfa_offset 4576, -4576
582 | .cfa_offset 4584, -4584
583 | .cfa_offset 4592, -4592
584 | .cfa_offset 4600, -4600
585 | .cfa_offset 4608, -4608
586 | .cfa_offset 4616, -4616
587 | .cfa_offset 4624, -4624
588 | .cfa_offset
```

⑤-O2 二级优化

设置了 O2 后，编译器会试图提高代码性能而不会增大体积和大量占用的编译时间，对应指令为：

```
g++ -O2 -S fib.i -o fib_2.s
```



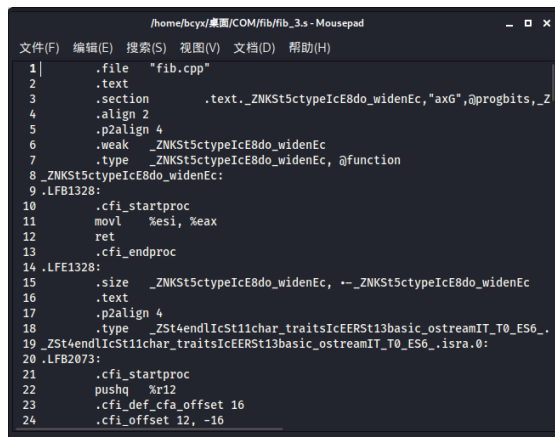
O2 优化

通过对循环的计数，在 O1 优化的基础上，O2 级优化会减少循环的展开次数。

⑥-O3 三级优化

三级优化主要针对程序空间大小进行优化，但也会消耗更多的代码编译时间，对应指令为：

```
g++ -O3 -S fib.i -o fib_3.s
```



O3 优化

⑦总结

可以看到，针对 O0、O1、O2、O3 四个优化的等级而言，优化的级别越高，虽然最后生成的代码的执行效率越高，但是代码量也会越大，同时编译过程所花费的时间也会越长，因此需要在其中进行一个权衡。

```
bcyx@kali:~/桌面/COM/fib$ ls -al
总用量 1436
drwxr-xr-x 3 bcyx bcyx 4096 9月 27 15:56 .
drwxr-xr-x 4 bcyx bcyx 4096 9月 25 18:00 ..
-rw-r--r-- 1 root root 2896 9月 27 15:27 fib_0.s
-rw-r--r-- 1 root root 2554 9月 27 15:30 fib_1.s
-rw-r--r-- 1 bcyx bcyx 4380 9月 27 15:45 fib_2.s
-rw-r--r-- 1 bcyx bcyx 4380 9月 27 15:56 fib_3.s
-rw-r--r-- 1 root root 208 9月 25 18:09 fib.cpp
-rw-r--r-- 1 bcyx bcyx 711808 9月 27 13:14 fib.i
-rw-r--r-- 1 root root 2896 9月 27 13:16 fib.s
-rw-r--r-- 1 bcyx bcyx 711808 9月 25 18:15 fib_test.i
drwxr-xr-x 2 bcyx bcyx 4096 9月 26 01:02 test
bcyx@kali:~/桌面/COM/fib$
```

ls -al 查看优化文件大小

第三节 汇编

2.3.1 汇编过程概述

汇编器所执行的汇编过程就是将汇编代码转换成机器可以执行的指令，每一个汇编语句几乎都对应了一条机器指令，所以汇编器的汇编过程相对于之前的编译过程而言，没有那么复杂的语法，也没有什么语义分析，更不需要指令优化，只是单纯的需要根据汇编指令和机器指令的对照表进行一一翻译即可。

2.3.2 汇编样例输出

Linux 中借助 g++ 让编译器使用编译器的输出结果 fib.s 生成目标机器指令文件 fib.o 的命令为：

```
g++ -c fib.s -o fib.o
```

或者直接从源代码 CPP 文件开始，经过预编译、编译和汇编直接输出目标文件。

```
g++ -c fib.cpp -o fib.o
```

其中 -c 选项表示只进行编译而不链接。

```
文件(F)  动作(A)  编辑(E)  查看(V)  帮助(H)

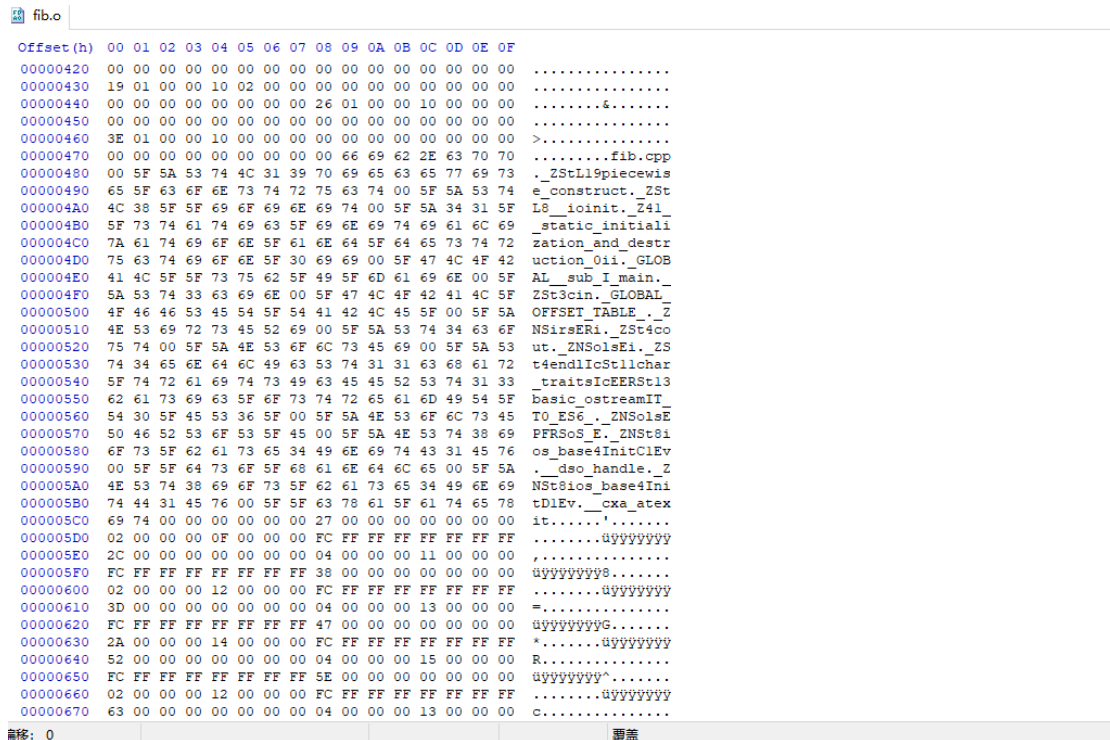
bcyx@kali:~/桌面/COM/fib$ gcc -c fib.s -o fib.o
bcyx@kali:~/桌面/COM/fib$
```

汇编指令

同时我们还可以利用 file 查看 fib.o 的类型。

```
bcyx@kali:~/桌面/COM/fib$ file fib.o
fib.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

借助 file 查看 fib.o 的类型



The image shows a hex editor window titled 'fib.o'. It displays a memory dump with addresses from 00000420 to 00000670. The hex values are shown in columns, and the corresponding ASCII values are shown on the right. The file is identified as an ELF 64-bit LSB relocatable file, x86-64, version 1 (SYSV), not stripped.

借助 HxD 工具查看.o 文件

2.3.3 目标文件的介绍

目标文件就是源代码编译后但是没有进行链接的那些中间文件,也就是 fib.o 文件以及在 win 下的.obj 文件,通过上面借助 file 查看 fib.o 的类型可以看出,.o 文件为 Relocatable File 即可重定位文件,这类文件包含了代码和数据,可以被用来链接成可执行文件或共享目标文件。

目标文件中的内容包含有编译后的机器指令代码、数据以及一些链接时所需要的信息,比如符号表、调试信息、字符串等。一般目标文件会将这些信息按不同的属性,以节的形式进行存储,有时候也可以称之为段。他们在一般情况下都表示一个一定长度的区域,基本上不加以区别,唯一的区别就是在 ELF 的链接视图和装载视图的时候。

程序的源代码编译后的机器指令经常放在代码段里,代码段常见的名字有.code 和.text,全局变量和局部静态变量会放在数据段.data 里。(在上学期的漏洞利用课程中还学到了.idata 即可执行文件所使用的动态链接库等外来函数与文件的信息,.rsrc 中存放程序的资源包括图标和菜单等)

readelf -h fib.o

```
bcyx@kali:~/桌面/COM/fib$ readelf -h fib.o
ELF 头:
Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
类别:    ELF64
数据:    2 补码, 小端序 (little endian)
Version: 1 (current)
OS/ABI:   UNIX - System V
ABI 版本: 0
类型:     REL (可重定位文件)
系统架构: Advanced Micro Devices X86-64
版本:     0x1
入口点地址: 0x0
程序头起点: 0 (bytes into file)
Start of section headers: 2176 (bytes into file)
标志:     0x0
Size of this header: 64 (bytes)
Size of program headers: 0 (bytes)
Number of program headers: 0
Size of section headers: 64 (bytes)
Number of section headers: 15
Section header string table index: 14
```

readelf -h fib.o

magic 参数中最开始的第一个字节对应 ASCII 字符中的 DEL 控制符号, 后面三个刚好是 ELF 的三个字母的 ASCII, 后面的字节 02 代表 ELF 为 64 位的, 对应的 01 为 32 位, 第六个字是字节序, 00 为无序, 01 为小端序, 02 为大端序 (小端序和大端序主要体现在 MSB 和 LSB 的位置), 第七个字节为 ELF 的版本号。

readelf -S fib.o

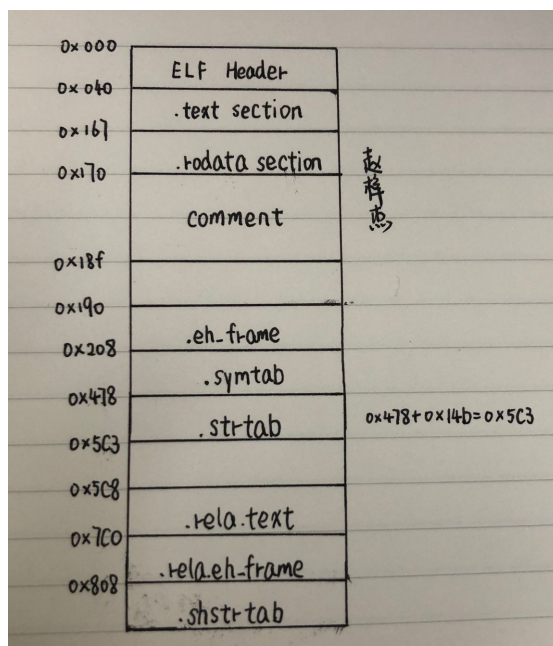
```
bcyx@kali:~/桌面/COM/fib$ readelf -S fib.o
There are 15 section headers, starting at offset 0x880:

节头:
[节] 名称      类型      地址      大小      标志      链接      信息      偏移量
[ 0]          NULL      0000000000000000 00000000 00000000 00000000 00000000
[ 1] .text      PROGBITS 0000000000000000 00000040 00000000 00000000 00000000
[ 2] .rela.text RELA      0000000000000000 000000c8 00000000 00000000 00000000
[ 3] .data      PROGBITS 0000000000000000 00000167 00000000 00000000 00000000
[ 4] .bss       NOBITS    0000000000000000 00000167 00000000 00000000 00000000
[ 5] .rodata    PROGBITS 0000000000000000 00000167 00000000 00000000 00000000
[ 6] .init_array INIT_ARRAY 0000000000000000 00000168 00000000 00000000 00000000
[ 7] .rela.init_array RELA      0000000000000000 000007a8 00000000 00000000 00000000
[ 8] .comment   PROGBITS 0000000000000000 00000170 00000000 00000000 00000000
[ 9] .note.GNU-stack PROGBITS 0000000000000000 0000018f 00000000 00000000 00000000
[10] .eh_frame   PROGBITS 0000000000000000 00000190 00000000 00000000 00000000
[11] .rela.eh_frame RELA      0000000000000000 000007c0 00000000 00000000 00000000
[12] .symtab     SYMTAB    0000000000000000 00000208 00000000 00000000 00000000
[13] .strtab     STRTAB    0000000000000000 00000478 00000000 00000000 00000000
[14] .shstrtab   STRTAB    0000000000000000 00000008 00000000 00000000 00000000
Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
l (large), p (processor specific)
bcyx@kali:~/桌面/COM/fib$
```

借助 readelf 查看文件的 elf 结构

我们根据上面的 elf 结构可以简单画出 elf 结构图的简化版本:

程序源代码在被编译之后主要分成了两种段, 分别是程序指令和程序数据, 代码段属于程序指令, 而数据段和.bss 段属于程序数据。之所以这样, 第一个原因是数据区域和指令区域的读写问题, 数据区域对于进程而言应该是可读写的, 而指令区域仅仅是可读的, 为了安全考虑需要进行区分; 第二个原因则是缓存和



手画简单的 elf 结构图

内存共享问题。

简单提一句.bss段，一般而言，未初始化的全局变量和局部静态变量一般都放在.bss段中，但是同样存在特例，即如果 `static int x1 = 0;` 按照常理它应该放在.data段，但是因为计算机对0默认是未初始化的，所以x1仍然放在.bss段中。

2.3.4 深度理解 fib.o 文件

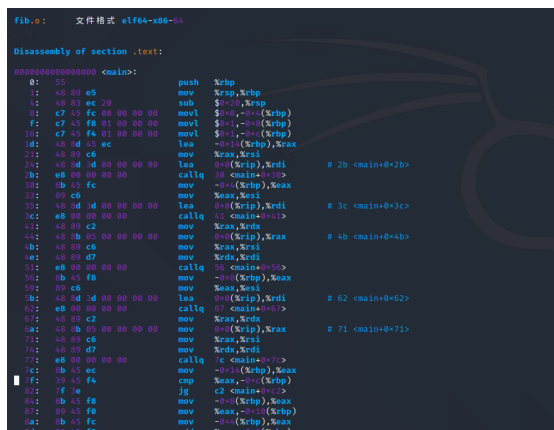
①借助 size 指令查看 ELF 文件的代码段、数据段和 BSS 段的长度。

```
bcyx@kali:~/桌面/COM/fib$ size fib.o
text    data    bss     dec     hex filename
416      8        1     425     1a9 fib.o
bcyx@kali:~/桌面/COM/fib$
```

size fib.o 查看相关段的长度

②代码段

借助 objdump 和 -s 参数查看代码段，上面的 contents of section .text 就是.text 以十六进制打印出来的内容，对照下面的反汇编结果，可以发现，.text 中所包含的就是 fib.c 里面的函数和变量，结尾处的 c3 也对应了 main 函数的最后一条指令 ret。



fib.S 文件

第四节 链接——一个很容易被“我们”忽视的环节

2.4.1 链接环节的介绍

在上面编译介绍的时候我写到之前在刚接触的时候我有过一点小的问题,就是编译汇编这些过程无论如何都只是停留在对指令的变换和优化,而内存呢?那些变量、参数的地址又是什么时候得到的呢?

之前看到过一个小故事,我觉得还挺有意思,想和老师您分享一下,就是传说在距离地球几十万光年的某个星球上,住满了程序员,那里的人们写代码的时候将所有的源代码都写在一个文件中,发展至今,一个程序源代码的文件长达数百万行,以至于运维的程序员每天要从早到晚不停的维护,甚至没有能力维护某些程序了,因此那里的人们开始寻找一项新的解决办法.....

可能故事是一个冷笑话(不过我真的觉得很好笑),但是这也展示出了链接的重要性。在介绍链接之前需要引入几个小的概念。

①重定位

因为我们的程序不是一成不变的,肯定会有后期的修改和添加,我们将修改程序后重新计算各个目标的地址过程叫做重定位。

②符号

我其实一直有一个错觉,推进计算机发展的其实是懒惰(狗头保命),比如符号这个概念,它本身就是为了减少在跳转的时候出现的错误,也是为了减少跳转时候所写的代码量,它用来表示一个地址,这个地址可以代表一个子程序的起始地址,也可以是一个变量的起始地址,符号的出现让上个世纪的程序员不禁感叹“解放了生产力就提高了劳动力”。

一个程序会被分割成很多模块,很多的包,但是这些模块最后如何组成一

个完整的程序呢？模块的拼接过程就是这个让很多人（包括我）都很忽视的环节——链接。

2.4.2 链接的概念

程序员们将每个源代码模块独立的编译，然后按照需要将他们组装起来，这个组装的过程就称之为——链接。链接的主要内容就是把各个模块之间相互引用的部分衔接处理好，使得模块能够正常的拼接在一起。从本质上考虑，链接器无非就是把一些指令对其它符号地址的引用加以修正，链接过程主要包括了地址和空间分配、符号决议和重定位等步骤。

链接处理可以分成两种，一种是静态链接而另一种则是动态链接。

① 静态链接

静态链接过程主要是把可重定位文件依次读入，分析各个文件的文件头，进而依次读入各个文件的节区，并计算各个节区的虚拟内存位置。之后，利用计算出来的存储位置，对一些需要进行重定位的符号进行处理，设定他们的虚拟内存地址等，最后生成一个可执行文件或者是动态链接库。

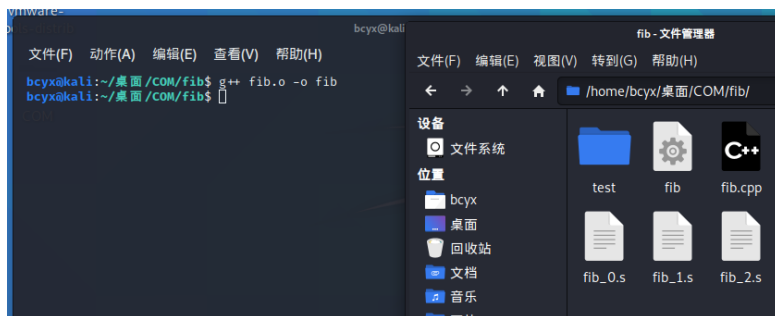
② 动态链接

动态链接所做的只是在最终的可执行程序中，记录下少量的信息，在这个可执行文件被执行的时候，动态链接库的全部内容被映射到虚地址空间中，动态链接程序将根据可执行程序中之前记录下来的信息找到相应的函数指针的地址，进而能够调用这些函数，完成执行过程。

2.4.3 链接样例输出

使用连接器生成 fib 可执行文件的指令是：

```
g++ fib.o -o fib
```



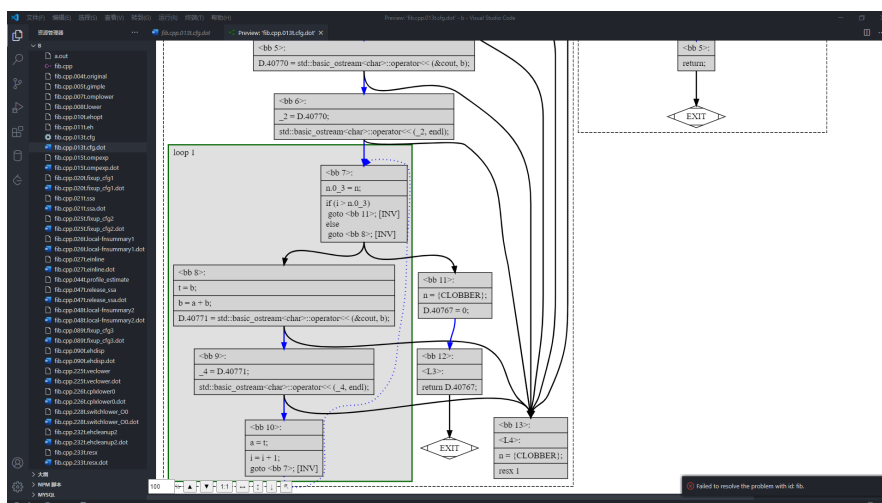
生成 fib 文件

输入样例验证 fib 文件

```
bcyx@kali:~/桌面/COM/fib$ ./fib
5
0
1
1
2
3
5
8
```

验证 fib 文件

当然我们可以通过生成.dot 文件进一步了解编译优化过程 `g++ -O0 -fdump-tree-all-graph fib.cpp`



.dot 可视化文件

第五节 整体优化

2.5.1 pipe 优化原理概述

将源代码编译成可执行文件需要四个步骤，并且还会有中间文件的产生，读写文件都是 I/O 操作，而 I/O 操作会大大减慢 GCC 编译器完成编译的速度。

pipe 优化方式，会将上一步编译的结果通过管道传递给下一步，这使得中间文件的读写全部在内存中完成，而不需要 I/O 操作，这将大大提高编译器的编译效率。

由此可知，和-O 参数的编译不一样的，pipe 优化并不是针对编译过程中的某一个阶段进行优化，而是对整个编译过程的整体性进行优化，因此在文末最后将简单介绍 pipe 优化及优化结果。

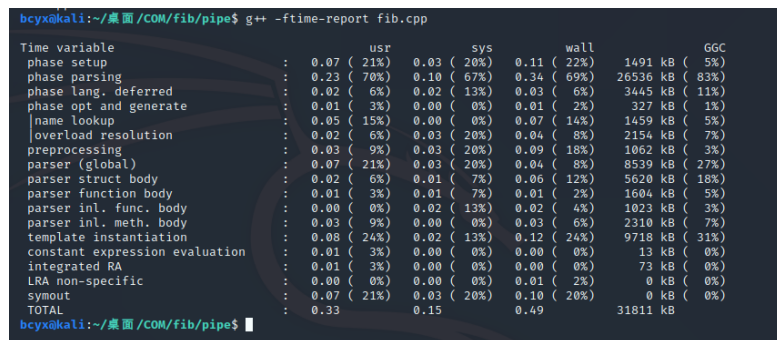
2.5.2 pipe 优化样例

g++ 中调用 pipe 优化选项的命令是：g++ -pipe fib.cpp
生成可执行文件 a.out

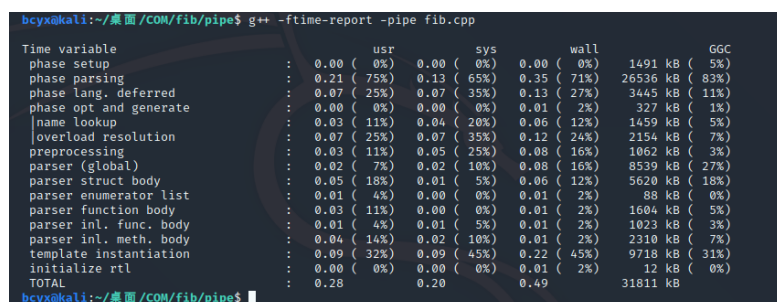


pipe 优化生成可执行文件 a.out

我们可以借助-ftime-report 来查看 gcc 编译器编译过程的耗时：



未使用 pipe 优化



使用 pipe 优化

```
bcyx@kali:~/桌面/COM/fib/pipe$ ls
a.out b.out fib.cpp
bcyx@kali:~/桌面/COM/fib/pipe$ diff a.out b.out
bcyx@kali:~/桌面/COM/fib/pipe$
```

diff 比较优化产生文件是否有差异

同时我们也可以发现，是否使用 pipe 优化，只是影响了编译过程中的耗时，并不会对最终的编译结果有任何影响，因为 pipe 优化只是改变了编译过程中中间文件的读写和存储方式，并不会给编译结果带来实质性的影响。

第三章 总结

简单的对自己这次实验做一个总结，首先我觉得锻炼的是自己的实践能力，因为疫情期间没有什么上机课同时也不太重视实操，所以可能很多 Linux 指令都不习惯了，这次报告算是对自己之前一些知识的温习；其次就是对 Latex 的学习，报告采用 Latex 的形式，在一定程度上提前于毕业论文让我接触到了 Latex，感觉到了 Latex 在撰写论文和大型报告方面的功能之强大，也让我摆脱了只能用 markdown 写一些论文和报告的问题；最后就是对编译器知识的进一步理解，当然我在上面所提到的可能也只是编译器很浅显的表面的知识，很多地方我没有特别深入去追究其原理和组成，但是也在一定程度上完善了我对编译器的理解，因为我个人参加的国创项目组正在研究 AFL 的二进制代码编译相关，这次学习和实操也在一定程度上给我对 AFL 的学习提供了较好的理论基础。

参考文献

- [1] 俞甲子. 程序员的自我修养. 北京: 电子工业出版社, 2009.
- [2] 王春红. 浅谈编译程序的工作过程 [J]. 河东学刊, 1999.
- [3] GCC 官方文档——<https://gcc.gnu.org/onlinedocs/>
- [4] 预处理指令集锦——https://blog.csdn.net/maopig/article/details/6740695?utm_medium=dist-task-blog-OPENSEARCH-2.channel_param&depth_1-utm_source=distribute.pc_relevant.none-task-blog-OPENSEARCH-2.channel_param
- [5] 预编译输出的行标志——<https://www.it610.com/article/683525.htm>
- [6] AT&T 汇编格式简介——https://blog.csdn.net/jtli_embeddedcv/article/details/9320813?utm_task-blog-title-1&spm=1001.2101.3001.4242

文中所涉及到的源码:

```
#include<iostream>
using namespace std;
int main(){
    int a,b,i,t,n;
    a = 0;
    b = 1;
    i = 1;
    cin>>n;
    cout<<a<<endl;
    cout<<b<<endl;
    while(i<=n){
        t = b;
        b = a + b;
        cout<<b<<endl;
        a = t;
        i = i + 1;
    }
}
```

文中所涉及到的汇编源码:

```
.file "fib.cpp"
.text
.section .rodata
.type _ZStL19piecewise_construct, @object
.size _ZStL19piecewise_construct, 1
_ZStL19piecewise_construct:
.zero 1
.local _ZStL8__ioint
.comm _ZStL8__ioint,1,1
.text
.globl main
.type main, @function
main:
.LFB1572:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
```



```

movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $32, %rsp
movl $0, -4(%rbp)
movl $1, -8(%rbp)
movl $1, -12(%rbp)
leaq -20(%rbp), %rax
movq %rax, %rsi
leaq _ZSt3cin(%rip), %rdi
call _ZNSirsERi@PLT
movl -4(%rbp), %eax
movl %eax, %esi
leaq _ZSt4cout(%rip), %rdi
call _ZNSolsEi@PLT
movq %rax, %rdx
movq
    _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_@GOTPCREL(%rip),
    %rax
movq %rax, %rsi
movq %rdx, %rdi
call _ZNSolsEPFRSoS_E@PLT
movl -8(%rbp), %eax
movl %eax, %esi
leaq _ZSt4cout(%rip), %rdi
call _ZNSolsEi@PLT
movq %rax, %rdx
movq
    _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_@GOTPCREL(%rip),
    %rax
movq %rax, %rsi
movq %rdx, %rdi
call _ZNSolsEPFRSoS_E@PLT
.L3:
movl -20(%rbp), %eax
cmpl %eax, -12(%rbp)
jg .L2
movl -8(%rbp), %eax
movl %eax, -16(%rbp)
movl -4(%rbp), %eax

```

```

    addl %eax, -8(%rbp)
    movl -8(%rbp), %eax
    movl %eax, %esi
    leaq _ZSt4cout(%rip), %rdi
    call _ZNSolsEi@PLT
    movq %rax, %rdx
    movq
        _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_@GOTPCREL(%rip),
        %rax
    movq %rax, %rsi
    movq %rdx, %rdi
    call _ZNSolsEPFRSoS_E@PLT
    movl -16(%rbp), %eax
    movl %eax, -4(%rbp)
    addl $1, -12(%rbp)
    jmp .L3
.L2:
    movl $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE1572:
    .size main, .-main
    .type _Z41__static_initialization_and_destruction_0ii, @function
_Z41__static_initialization_and_destruction_0ii:
.LFB2070:
    .cfi_startproc
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq %rsp, %rbp
    .cfi_def_cfa_register 6
    subq $16, %rsp
    movl %edi, -4(%rbp)
    movl %esi, -8(%rbp)
    cmpl $1, -4(%rbp)
    jne .L7
    cmpl $65535, -8(%rbp)

```

```
jne .L7
leaq _ZStL8__ioinit(%rip), %rdi
call _ZNSt8ios_base4InitC1Ev@PLT
leaq __dso_handle(%rip), %rdx
leaq _ZStL8__ioinit(%rip), %rsi
movq _ZNSt8ios_base4InitD1Ev@GOTPCREL(%rip), %rax
movq %rax, %rdi
call __cxa_atexit@PLT
.L7:
nop
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE2070:
.size _Z41__static_initialization_and_destruction_0ii,
    .-_Z41__static_initialization_and_destruction_0ii
.type _GLOBAL__sub_I_main, @function
_GLOBAL__sub_I_main:
.LFB2071:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl $65535, %esi
movl $1, %edi
call _Z41__static_initialization_and_destruction_0ii
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE2071:
.size _GLOBAL__sub_I_main, .-_GLOBAL__sub_I_main
.section .init_array,"aw"
.align 8
.quad _GLOBAL__sub_I_main
.hidden __dso_handle
```

```
.ident "GCC: (Debian 10.2.0-7) 10.2.0"  
.section .note.GNU-stack,"",@progbits
```
