预备工作 1——了解你的编译器

杨侯哲 李煦阳
孙一丁 李世阳 杨科迪

日期：2020 年 9 月至 2021 年 9 月

目录

1 实验描述

以你熟悉的编译器，如 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：

1. 完整的编译过程都有什么？
2. 预处理器做了什么？
3. 编译器做了什么？
4. 汇编器做了什么？
5. 链接器做了什么？

并尽可能地对其实现方式有所了解。

1.1 方法

以一个简单的 C（C++）源程序为例，调整编译器的程序选项获得各阶段的输出，研究它们与源程序的关系，以此撰写调研报告。二进制文件或许需要利用某些系统工具理解，如 `objdump`、`nm`。

进一步地，可以调整你认为**关键**的编译参数（如优化参数、链接选项参数），比较目标程序的大小、运行性能等。

你的源程序可以包含尽可能丰富的语言特性（如函数、全局变量、常量、各类宏、头文件...），以更全面探索每一个阶段编译器进行的工作。

1.2 实验要求

要求：

撰写调研报告（符合科技论文写作规范，包含完整结构：题目、摘要、关键字、引言、你的工作和结果的具体介绍、结论、参考文献，文字、图、表符合格式规范，建议使用 latex 撰写）[1]

（可基于此模板，该模板所在网站是一个很流行的 latex 文档协同编辑网站，copy 此 project 即可成为自己的项目，在其上编辑即可，更多 latex 参考资料[2]）。

期望： 不要当作“命题作文”，更多地发挥主观能动性，做更多探索。如：

1. 细微修改程序，观察各阶段输出的变化，从而更清楚地了解编译器的工作；
2. 调整编译器的程序选项，例如加入调试选项、优化选项等，观察输出变化、了解编译器；
3. 尝试更深入的内容，例如令编译器做自动并行化，观察输出变化、了解编译器；
4. 与预习作业 1 中的优化问题相结合等等。

[1] 你可以搜索“vscode+latex workshop”以配置 `latex` 环境，再进一步了解“如何用 latex 书写中文”。

[2] LaTeX 入门、LaTeX 命令与符号汇总、LaTeX 数学公式等符号书写

基础样例程序：

```
int main()
{
    int i, n, f;
    cin >> n;
    i = 2;
    f = 1;
    while (i <= n)
    {
        f = f * i;
        i = i + 1;
    }
    cout << f << endl;
}
```

2) 斐波那契数列

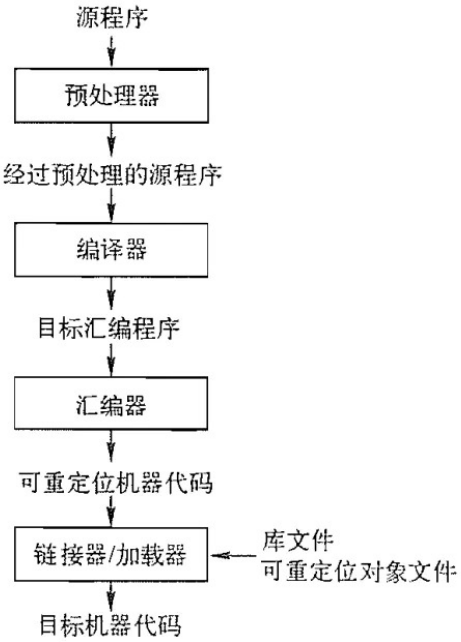
```
int main()
{
    int a, b, i, t, n;

    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;
    }
}
```

2 参考流程

以下内容仅供参考，更多的细节希望同学们亲自动手体验，详细了解各阶段的作用。

以一个 C 程序为例，整体的流程如图所示：



简单来说，不同阶段的作用如下：

预处理器 处理源代码中以 # 开始的预编译指令，例如展开所有宏定义、插入 include 指向的文件等，以获得经过预处理的源程序。

编译器 将预处理器处理过的源程序文件翻译成为标准的汇编语言以供计算机阅读。

汇编器 将汇编语言指令翻译成机器语言指令，并将汇编语言程序打包成可重定位目标程序。

链接器 将可重定位的机器代码和相应的一些目标文件以及库文件连接在一起，形成真正能在机器上运行的目标机器代码。

我们将以一段简单的 C 代码为例：

```c
#include<stdio.h>
int main(){
    int a,b;
    // 输入变量
    scanf("%d%d",&a,&b);
    // 输出结果
    printf("Hello World %d\n",a+b);
    return 0;
}
```

2.1 预处理器

预处理阶段会处理预编译指令，包括绝大多数的 # 开头的指令，如 `include define if` 等等，对 `include` 指令会替换对应的头文件，对 `define` 的宏命令会直接替换相应内容，同时会删除注释，添加行号和文件名标识。

对于 gcc，通过添加参数*-E* 令 gcc 只进行预处理过程，参数*-o* 改变 gcc 输出文件名，因此通过命令 `gcc main.c -E -o main.i`，即可得到预处理后文件。

观察预处理文件，可以发现文件长度远大于源文件，这就是将代码中的头文件进行了替代导致的结果。另外，实际上预处理过程是 gcc 调用了另一个程序（C Pre-Processor 调用时简写作 cpp）完成的过程，有兴趣的同学可以自行尝试。

2.2 编译器

编译过程是我们整门课程着重讲述的过程，具体来说分为六步，详细解释可以查看课程的预习 PPT，简单来说分别为

词法分析 将源程序转换为单词序列。对于 llvm，你可以通过以下命令获得 token 序列：

```
clang -E -Xclang -dump-tokens main.c
```

语法分析 将词法分析生成的词法单元来构建语法树。对于 gcc，你可以通过 `-fdump-tree-original-raw` flag 获得文本格式的 AST 输出。llvm 可以通过如下命令获得相应的 AST：

```
clang -E -Xclang -ast-dump main.c
```

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。

中间代码生成 完成上述步骤后，很多编译器会生成一个明确的低级或类机器语言的中间表示。

你可以通过`-fdump-tree-all-graph`和`-fdump-rtl-all-graph`两个 gcc flag 获得中间代码生成的多阶段的输出。生成的`.dot` 文件可以被 `graphviz` 可视化，vscode 中直接有相应插件。你可以看到控制流图（CFG），以及各阶段处理中（比如优化、向 IR 转换）CFG 的变化。你可以额外使用`-Ox`、`-fno-*` 等 flag 控制编译行为，使输出文件更可读、了解其优化行为。

llvm 可以通过下面的命令生成 llvm IR：

```
clang -S -emit-llvm main.c
```

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。

在第一周的预习作业中，很多同学对编译器如何进行代码优化感到疑问，在这个步骤中你可以通过 llvm 现有的优化 pass 进行代码优化探索。

在 llvm 官网对所有 pass 的分类[3]中，共分为三种：Analysis Passes、Transform Passes 和 Utility Passes。Analysis Passes 用于分析或计算某些信息，以便给其他 pass 使用，如计算支配边界、控制流图的数据流分析等；Transform Passes 都会通过某种方式对中间代码形式的程序做某种变化，如死代码删除，常量传播等。

llvm 可以通过下面的命令生成每个 pass 后生成的 llvm IR，以观察差别：

```
llc -print-before-all -print-after-all a.ll > a.log 2>&1
# 因为输出的内容过长，在命令行中无法完整显示，这时必须要对输出进行重定向
# 0、1、2 是三个文件描述符，分别表示标准输入 (stdin)、标准输出 (stdout)、标准错误 (stderr)
# 因此 2>&1 的具体含义就不难理解，你也可以试试去掉重定向描述，看看实际效果
```

同样，你也可以通过下面的命令指定使用某个 pass 以生成 llvm IR，以特别观察某个 pass 的差别：

```
opt -<module name> <test.bc> /dev/null
```

所有的 module name 对应的命令行参数也可以在[4]查到。

上面的指令需要用到 bc 格式，即 llvm IR 的二进制代码形式，而我们之前生成的是 llvm IR 的文本形式。当然我们也可以通过添加额外命令行参数的方式直接使用 ll 格式的 llvm IR，这里留给同学们自行探究。

我们可以通过下面的命令让 bc 和 ll 这两种 llvm IR 格式互转，以统一文件格式：

```
llvm-dis a.bc -o a.ll # bc 转换为 ll
llvm-as a.ll -o a.bc # ll 转换为 bc
```

如果你认为本部分的探索内容过于“黑盒”，你也可以尝试去阅读各大编译器，如 llvm 各个 pass 的源码[5]。尽管你现在可能并不了解大多数 pass 实际上是怎么工作的，但可能对你大作业的代码优化部分程序的编写有帮助。

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 main.S # 生成 x86 格式目标代码
arm-linux-gnueabihf-gcc main.i -S -o main.S # 生成 arm 格式目标代码
```

[3] llvm 对所有 pass 的简述
[4] llvm 的 pass 参数
[5] llvm 源码所在仓库

2.3 汇编器

汇编过程实际上把汇编语言程序代码翻译成目标机器指令的过程。其最终生成的是可重定位的机器代码。这一步一般被视为编译过程的“后端”，你可以在一些网上资料，比如这里，进行宏观的了解。

希望同学们在报告中详细分析并阐述汇编器处理的结果以及汇编器的具体功能分析。你可能会用到反编译工具（你可以在文后的 Makefile 中找到简单的使用示例）。

```
gcc main.S -c -o main.o
```

2.4 链接器加载器

由汇编程序生成的目标文件不能够直接执行。大型程序经常被分成多个部分进行编译，因此，可重定位的机器代码有必要和其他可重定位的目标文件以及库文件链接到一起，最终形成真正在机器上运行的代码。进而连接器对该机器代码进行执行生成可执行文件。可以尝试对可执行文件反汇编，看一看与上一阶段反汇编结果的不同。

在这一阶段，你可以尝试调整链接相关参数，如`-static`。

```
gcc main.o -o main
```

当你执行可执行文件时，便会使用到加载器，以将二进制文件载入内存。这不是我们要研究的事了。

2.5 样例 Makefile 文件

```
.PHONY: pre, lexer, ast-gcc, ast-llvm, cfg, ir-gcc, ir-llvm, asm, obj, exe, antiobj,
antiexe

pre:
    gcc main.c -E -o main.i

lexer:
    clang -E -Xclang -dump-tokens main.c

# 生成`main.c.003t.original`
ast-gcc:
    gcc -fdump-tree-original-raw main.c

# 生成`main.ll`
ast-llvm:
    clang -E -Xclang -ast-dump main.c

```

```
18  # 会生成多个阶段的文件 (.dot)，可以被 graphviz 可视化，可以直接使用 vscode 插件
19  # (Graphviz (dot) language support for Visual Studio Code)。
20  # 此时的可读性还很强。`main.c.011t.cfg.dot`
21  cfg:
22      gcc -O0 -fdump-tree-all-graph main.c
23
24  # 此时可读性不好，简要了解各阶段更迭过程即可。
25  ir-gcc:
26      gcc -O0 -fdump-rtl-all-graph main.c
27
28  ir-llvm:
29      clang -S -emit-llvm main.c
30
31  asm:
32      gcc -O0 -o main.S -S -masm=att main.i
33
34  obj:
35      gcc -O0 -c -o main.o main.S
36
37  antiobj:
38      objdump -d main.o > main-anti-obj.S
39      nm main.o > main-nm-obj.txt
40
41  exe:
42      gcc -O0 -o main main.o
43
44  antiexe:
45      objdump -d main > main-anti-exe.S
46      nm main > main-nm-exe.txt
47
48  clean:
49      rm -rf *.c.*
50
51  clean-all:
52      rm -rf *.c.* *.o *.S *.dot *.out *.txt *.ll *.i main
```