编译原理与设计实验报告

姓名：卜梦煜　　学号：1120192419　　班级：07111905

1. 实验名称

编译器认知实验

2. 实验目的

了解工业界常用的编译器 GCC 和 LLVM，熟悉编译器的安装和使用过程，观察编译器工作过程中生成的中间文件的格式和内容，了解编译器的优化效果，为编译器的学习和构造奠定基础。

3. 实验内容

本实验主要的内容为在 Linux 平台上安装和运行工业界常用的编译器 GCC 和 LLVM，安装完成后编写简单的测试程序，使用编译器编译，观察中间输出结果，并使用不同的优化编译命令进行优化编译，对比运行效率。

对于 GCC 编译器，完成编译器安装和测试程序编写后，按如下步骤完成：

- 查看编译器的版本：gcc --version
- 使用编译器编译单个文件：gcc hello.c -o hello
- 使用编译器编译链接多个文件：gcc hello1.c hello2.c -o hello
- 查看预处理结果：gcc -E hello.c -o hello.i
- 查看语法分析树：gcc -fdump-tree-all hello.c
- 查看中间代码生成结果：gcc -fdump-rtl-all hello.c
- 查看生成的目标代码（汇编代码）：gcc –S hello.c –o hello.s
- 优化编译：gcc hello.c -o hello -On，其中 n 分别为 0、1、2、3

对于 LLVM 编译器，完成编译器安装和测试程序编写后，按如下步骤完成：

- 查看编译器的版本：llvm-as --version
- 使用编译器编译单个文件：clang hello.c -o hello
- 使用编译器编译链接多个文件：clang hello1.c hello2.c -o hello
- 查看编译流程和阶段：clang -ccc-print-phases test.c -c
- 查看词法分析结果：clang test.c -Xclang -dump-tokens -c
- 查看词法分析结果 2：clang test.c -Xclang -dump-raw-tokens -c
- 查看语法分析结果：clang test.c -Xclang -ast-dump -c
- 查看语法分析结果 2：clang test.c -Xclang -ast-view -c

- 查看编译优化的结果：clang test.c -S -mllvm -print-after-all
- 查看生成的目标代码结果：clang test.c –S
- 优化编译：clang hello.c -o hello -On，其中 n 分别为 0、1、2、3

4. 实验环境

Ubuntu 18.04.6, GCC 7.5.0, LLVM 12.0.1(Debug)

5. 实验过程与步骤

5.1 编译器安装

（1）GCC 可在终端运行命令 sudo apt-get install gcc 安装。安装完成后运行命令 gcc –version 检查是否安装成功。

（2）LLVM 和 Clang 需在官网下载源码，手动安装。安装前需要预安装依赖项 GNU Make、CMake、python、GCC。安装 LLVM 时需留有足够的内存、硬盘空间和交换区，在 LLVM 目录下创建 build 文件夹，在该文件夹中运行命令 cmake -G "Unix Makefiles" -DLLVM_ENABLE_ASSERTIONS=On -DCMAKE_BUILD_TYPE=Debug ../ 编译，然后运行命令 sudo make install 安装，可使用 -jn 加速，其中 n 表示 CPU 核数。安装完成后运行命令 llvm-as –version 检查是否安装成功。

5.2 编写测试程序

测试程序包括两部分，均为语言认知实验中编写的 C++矩阵乘法代码。一部分有一个文件，用于编译单个文件；另一部分有两个文件，用于编译、链接多个文件，其中文件一使用的函数定义写在文件二中，从而实现编译链接多个文件的要求。

5.3 运行编译器进行观测

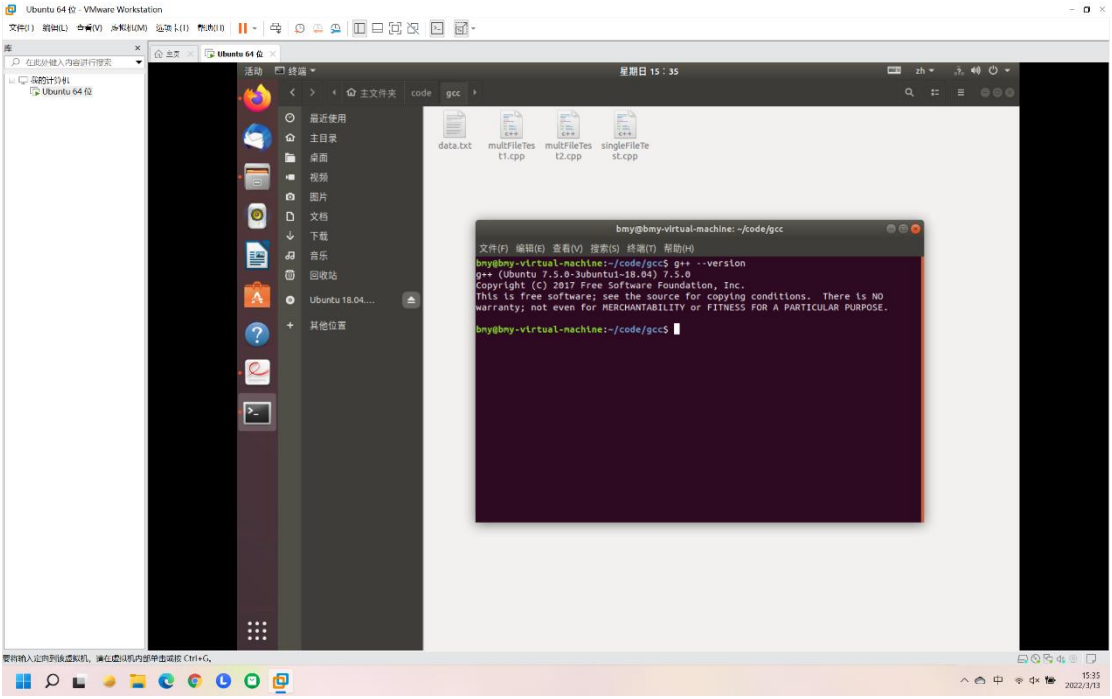
分别在终端用 GCC 和 LLVM 编译器按照“3. 实验内容”中命令依次执行，学习编译器基本使用方法，查看分析编译器的中间结果及其与源码的对应关系，掌握用编译器优化编译的方法，并对比 GCC 和 LLVM 优化的效率。

6. 结果分析

6.1 GCC 运行结果分析

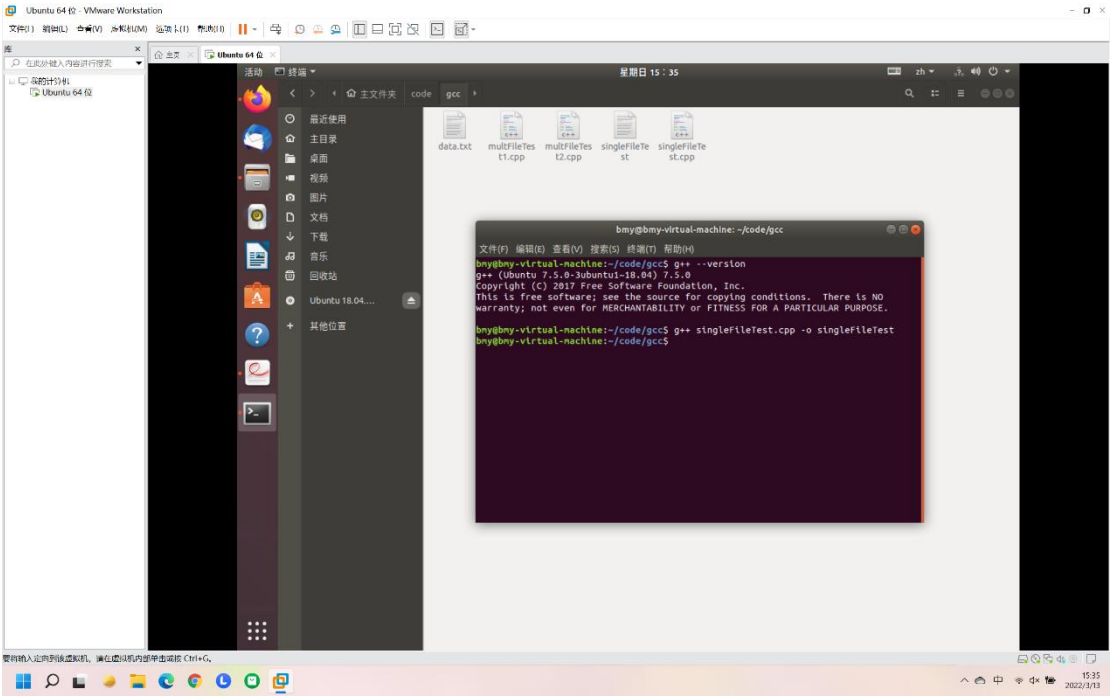
GCC 编译器各步骤结果如下：

（1）查看编译器的版本：g++ --version



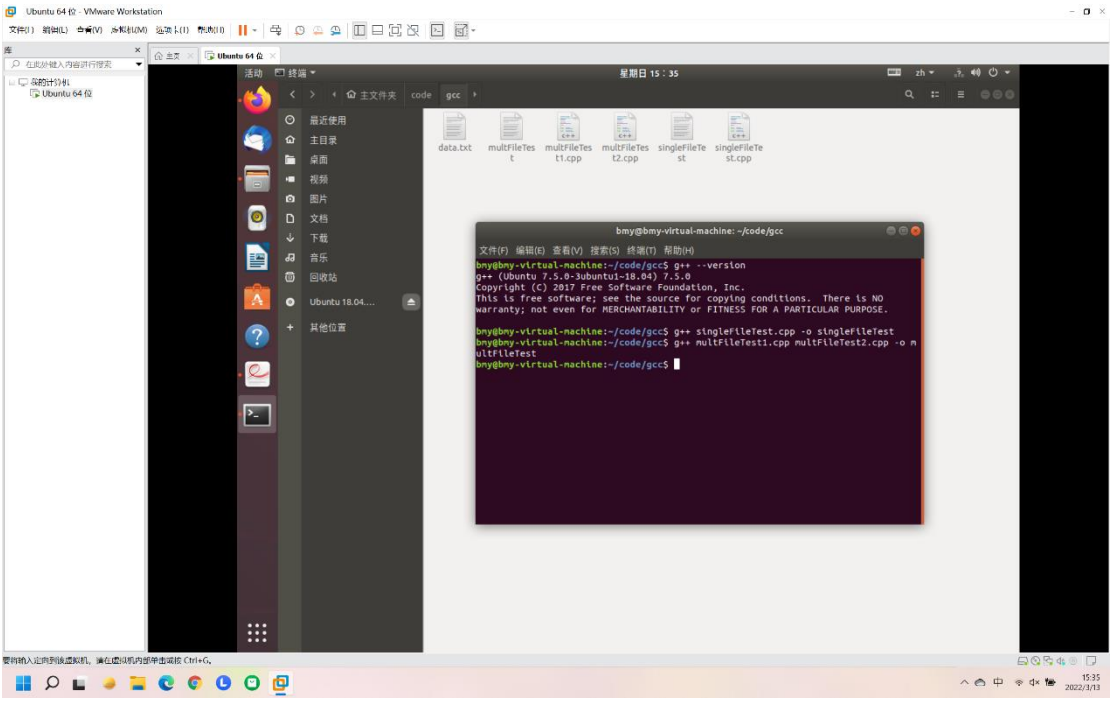
由图可知，GCC 版本为 7.5.0。

（2）使用编译器编译单个文件：g++ singleFileTest.cpp -o singleFileTest



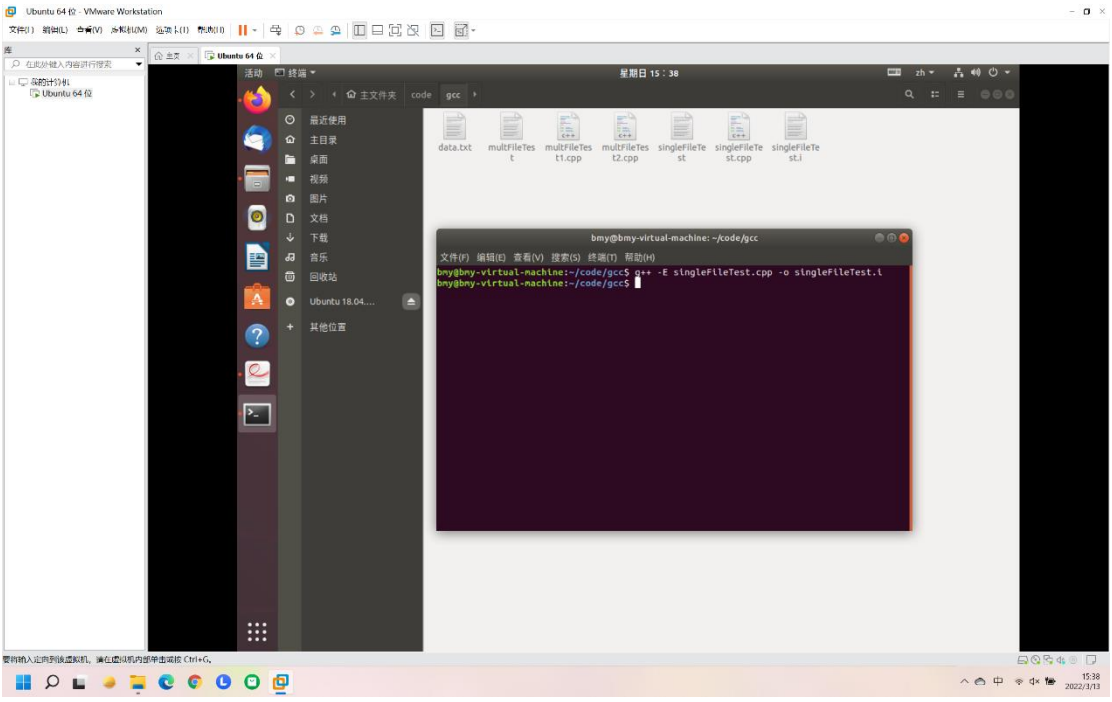
由图可知，该命令编译生成可执行文件 singleFileTest。

（3）使用编译器编译链接多个文件：g++ multFileTest1.cpp multFileTest2.cpp -o multFileTest

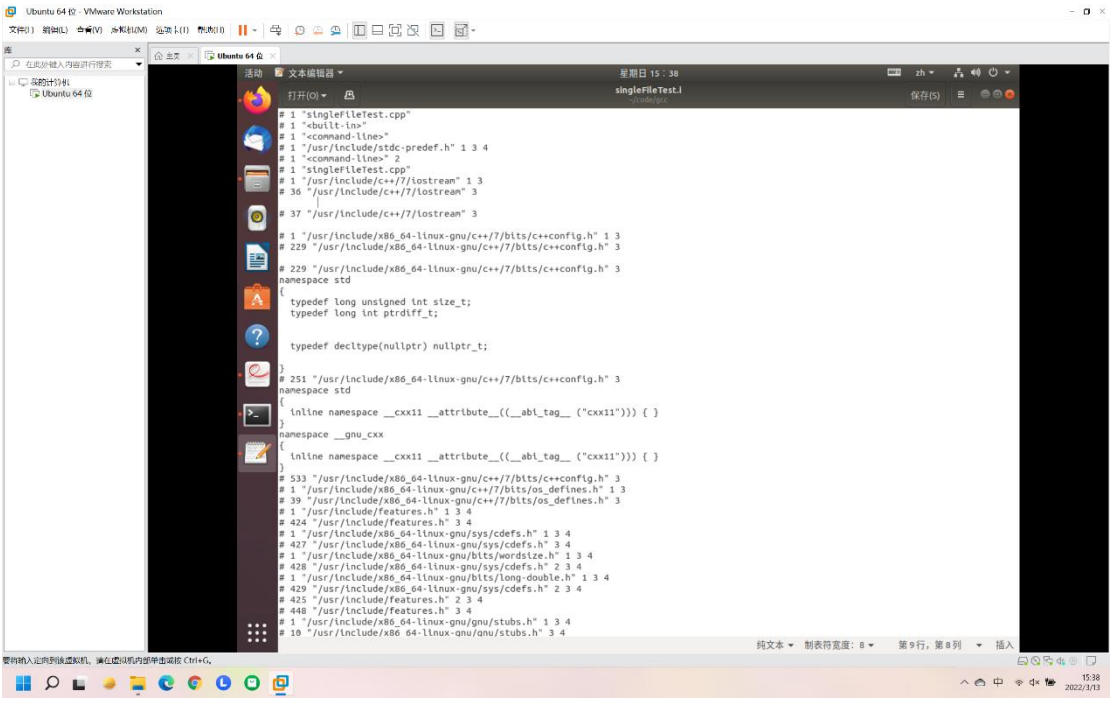


由图可知，该命令编译、链接两个.cpp 文件，生成一个可执行文件。查阅相关资料知，编译器首先编译多个.cpp 源文件，生成多个.o(.obj)目标文件。当所有文件都编译完成后，GCC 链接这些文件，生成一个可执行文件。

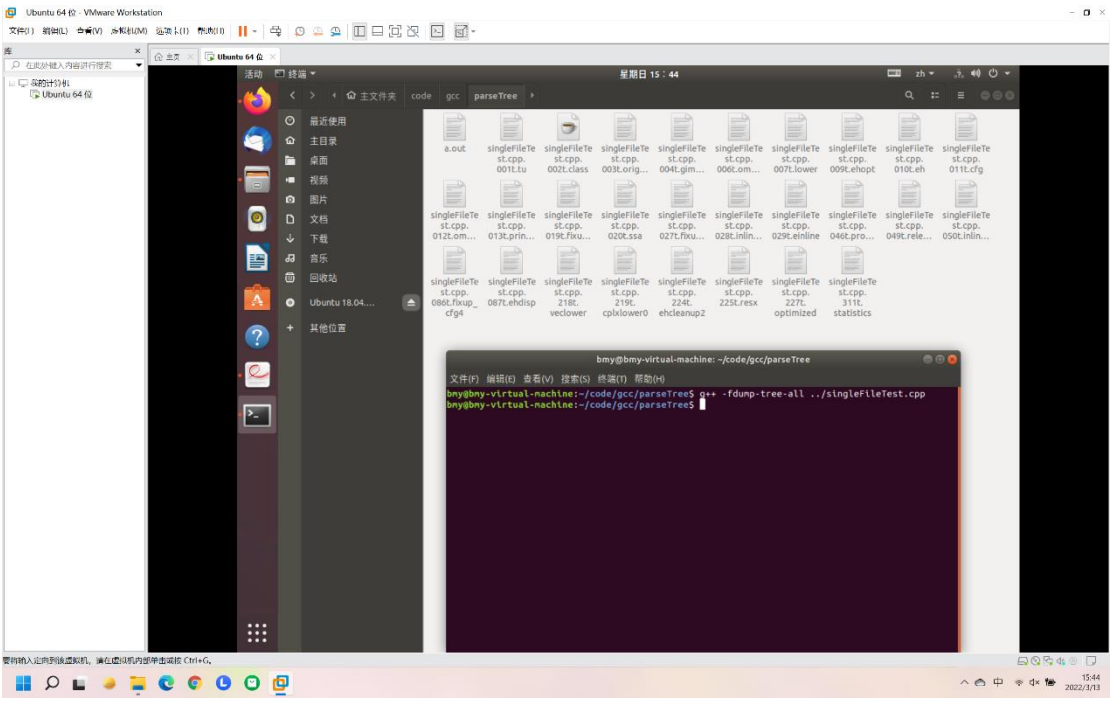
（4）查看预处理结果：g++ -E singleFileTest.cpp -o singleFileTest.i



预处理后.i 文件如下图：

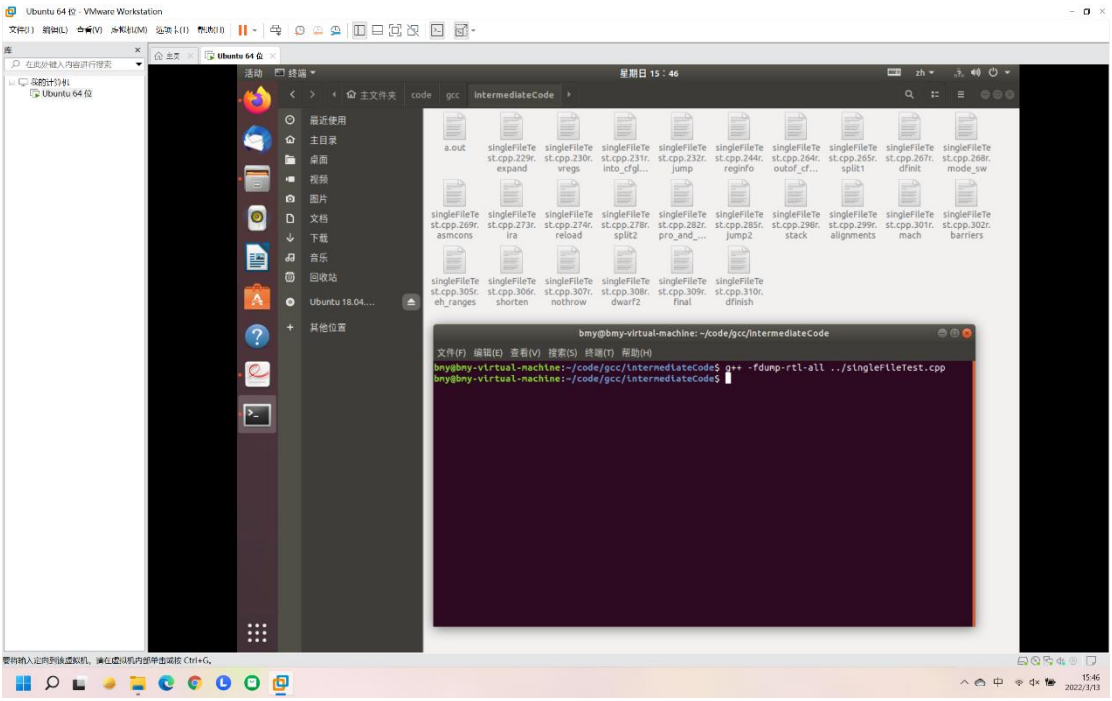


（5）查看语法分析树：g++ -fdump-tree-all singleFileTest.cpp



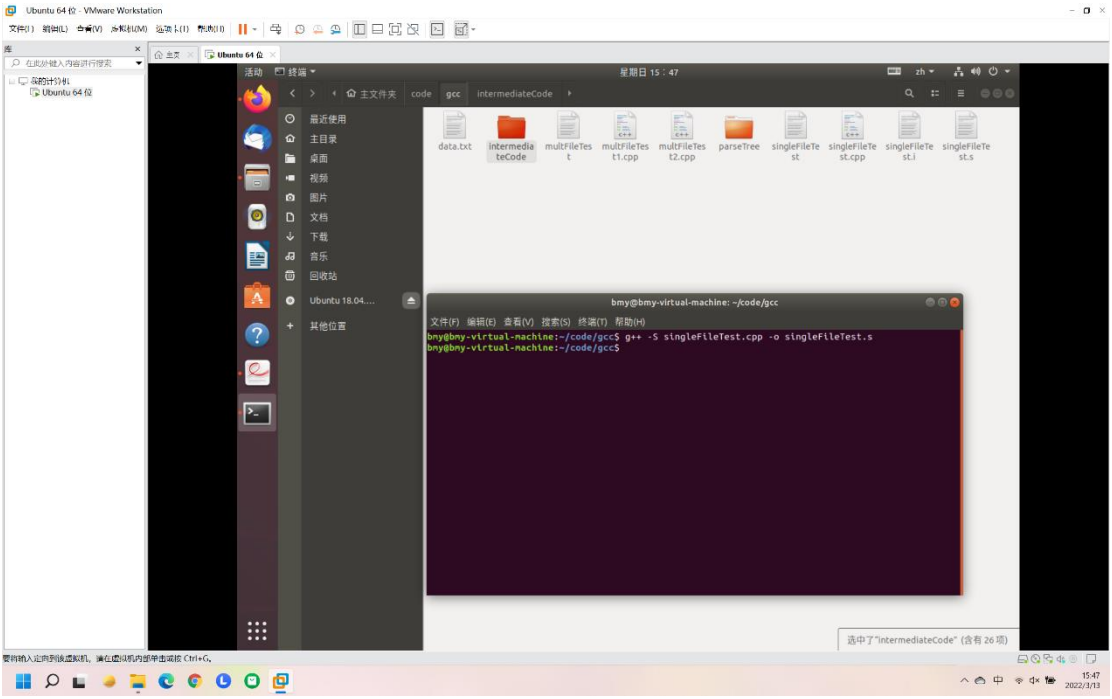
由图可知，该命令生成较多中间文件。

（6）查看中间代码生成结果：g++ -fdump-rtl-all singleFileTest.cpp

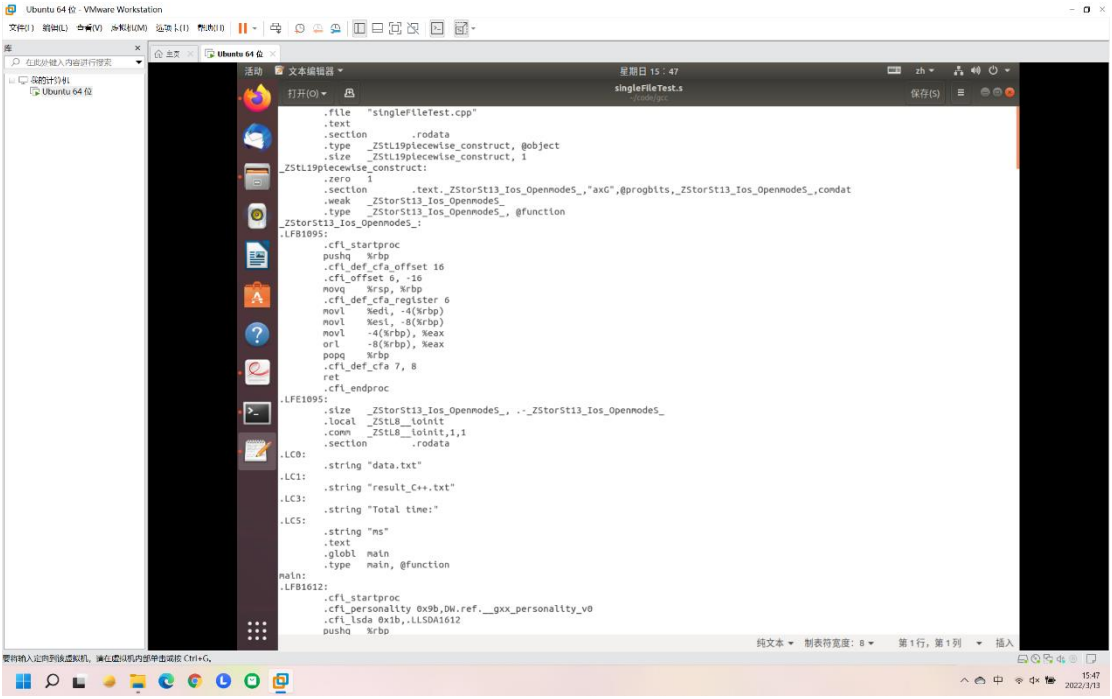


由图可知，该命令生成较多中间文件。

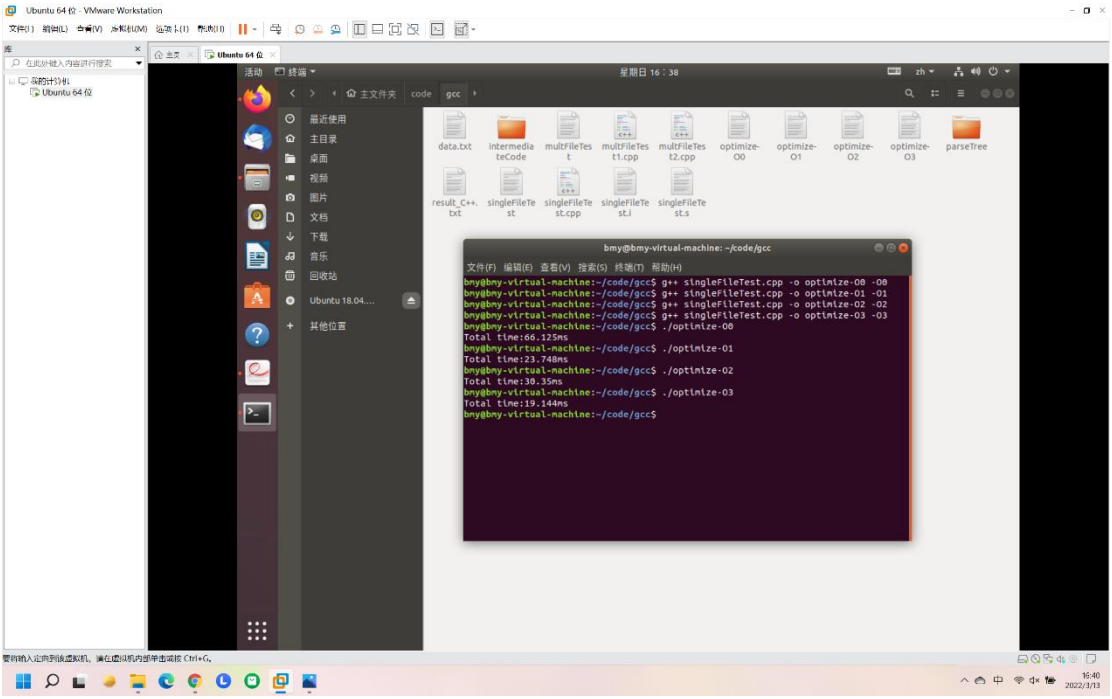
（7）查看生成的目标代码（汇编代码）：g++ –S singleFileTest.cpp –o singleFileTest.s



-S 选项含义为仅编译为汇编语言的代码文件，不进行汇编和链接操作。查看汇编语言文件如下：



（8）优化编译：g++ singleFileTest.cpp -o singleFileTest -On，其中 n 分别为 0、1、2、3



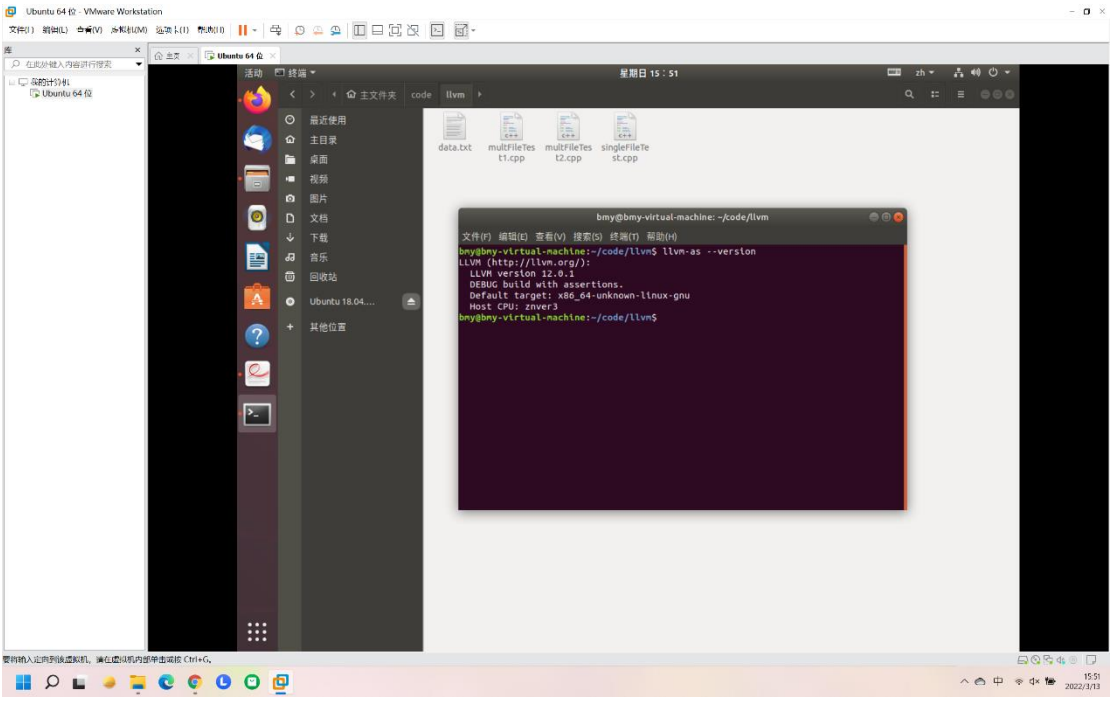
分别使用-O0、-O1、-O2、-O3 优化命令优化编译，分别运行四种优化后的可执行文件 10 次，运行速度统计如下：

gcc	O0/ms	O1/ms	O2/ms	O3/ms
0	65.675	23.94	30.142	18.593
1	64.5	23.814	30.359	18.961
2	66.16	23.814	30.89	19.249
3	65.031	24.224	30.422	19.749
4	65.208	24.56	30.356	18.989
5	64.869	23.988	30.695	18.92
6	64.822	24.176	30.777	19.273
7	65.238	24.149	30.651	18.869
8	64.633	24.631	30.705	18.93
9	65.671	24.097	30.46	19.393
平均用时	65.1807	24.1393	30.5457	19.0926

由图可知，源程序优化后，性能往往能得到提升，且优化等级越高性能提升越大，但这不是绝对的，存在高级优化后性能降低的情况。

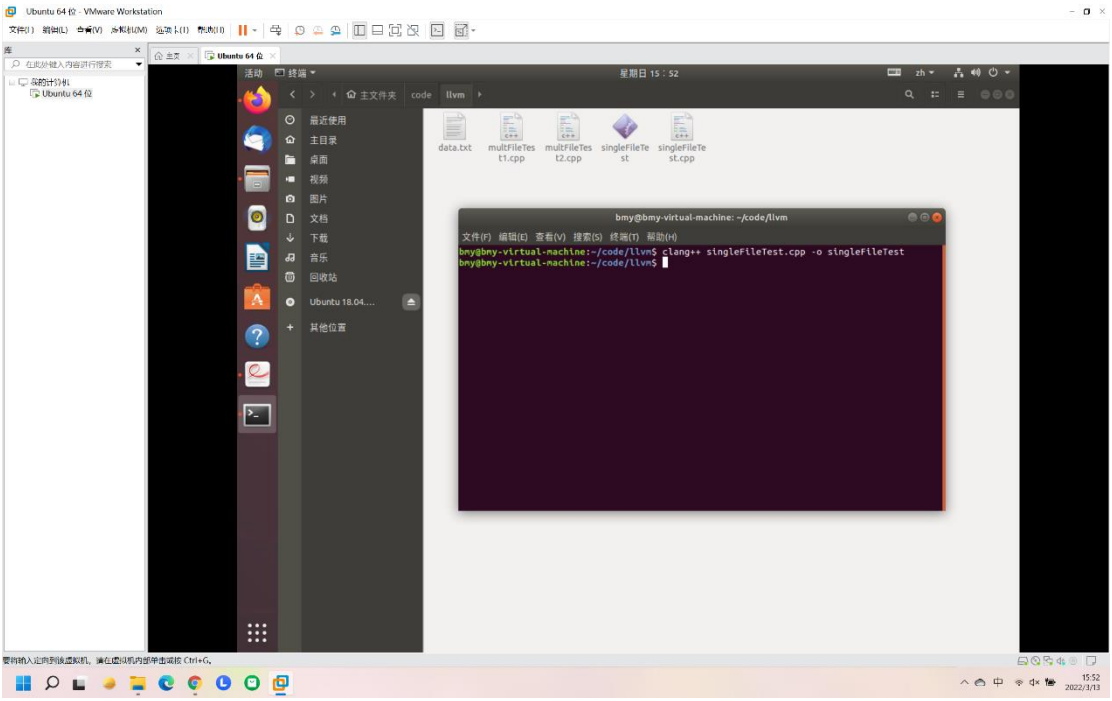
6.2 LLVM 运行结果分析

（1）查看编译器的版本：llvm-as –version



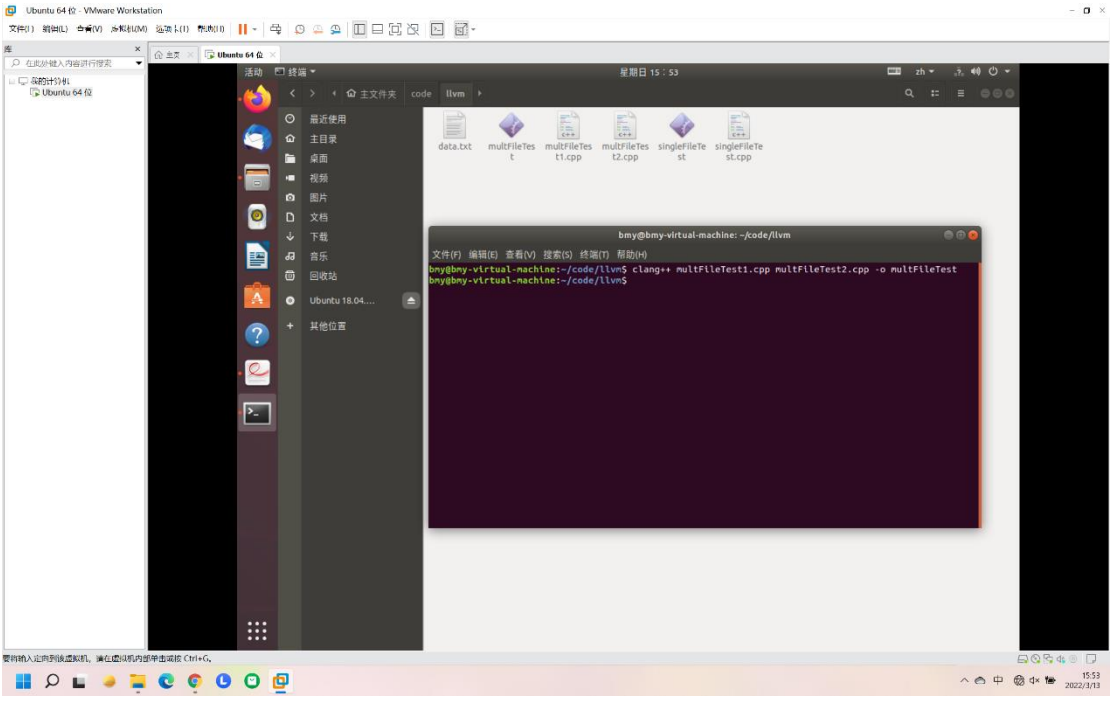
由图可知，LLVM 编译器版本为 12.0.1(Debug)版。

（2）使用编译器编译单个文件：clang++ singleFileTest.cpp -o singleFileTest



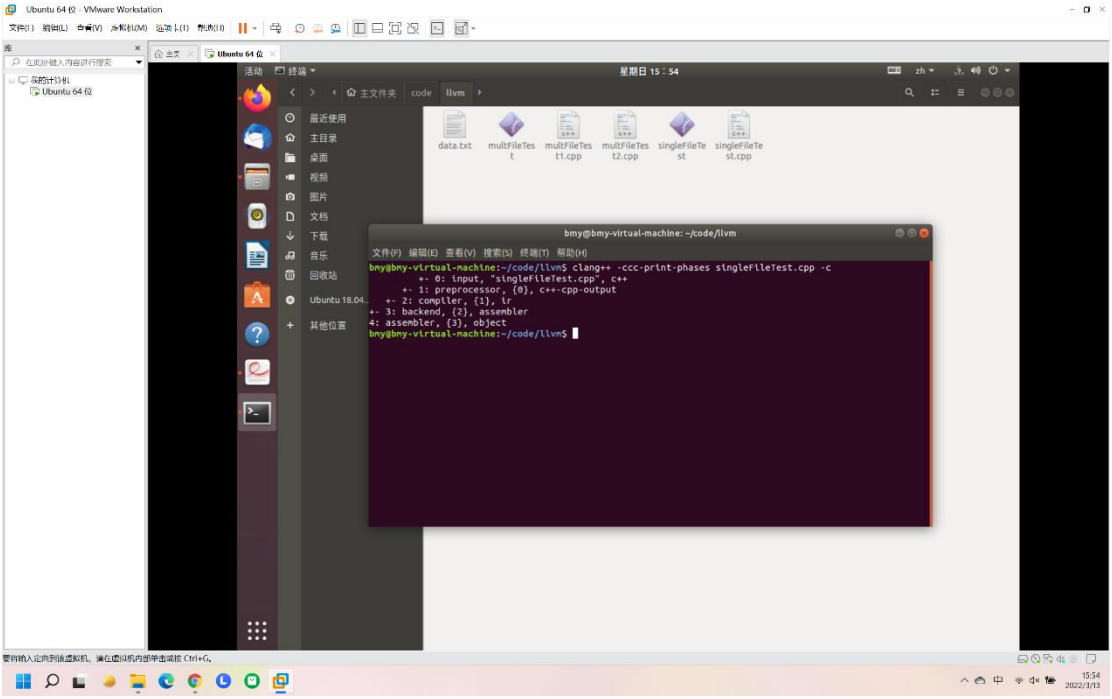
由图可知，该命令编译生成可执行文件 singleFileTest。

（3）使用编译器编译链接多个文件：clang++ multFileTest1.c multFileTest2.c -o multFileTest



由图可知，该命令编译生成可执行文件 multFileTest。LLVM 编译链接生成可执行文件的过程与 GCC 相同。

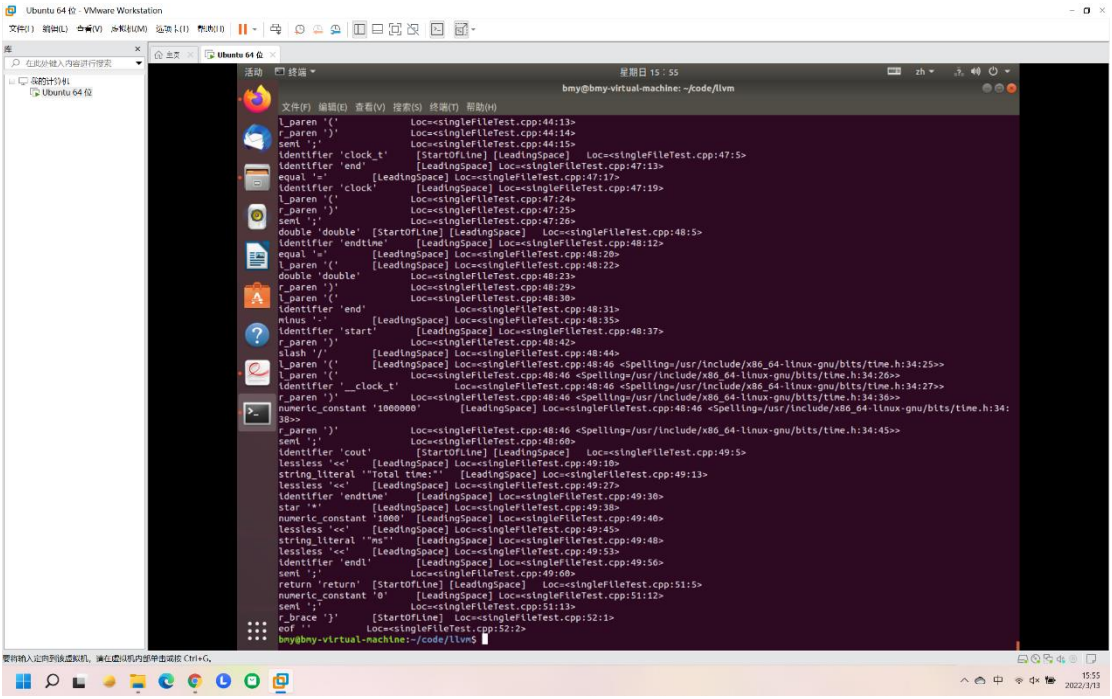
（4）查看编译流程和阶段：clang++ -ccc-print-phases singleFileTest.cpp -c



由图可知，该编译过程分为 5 步：

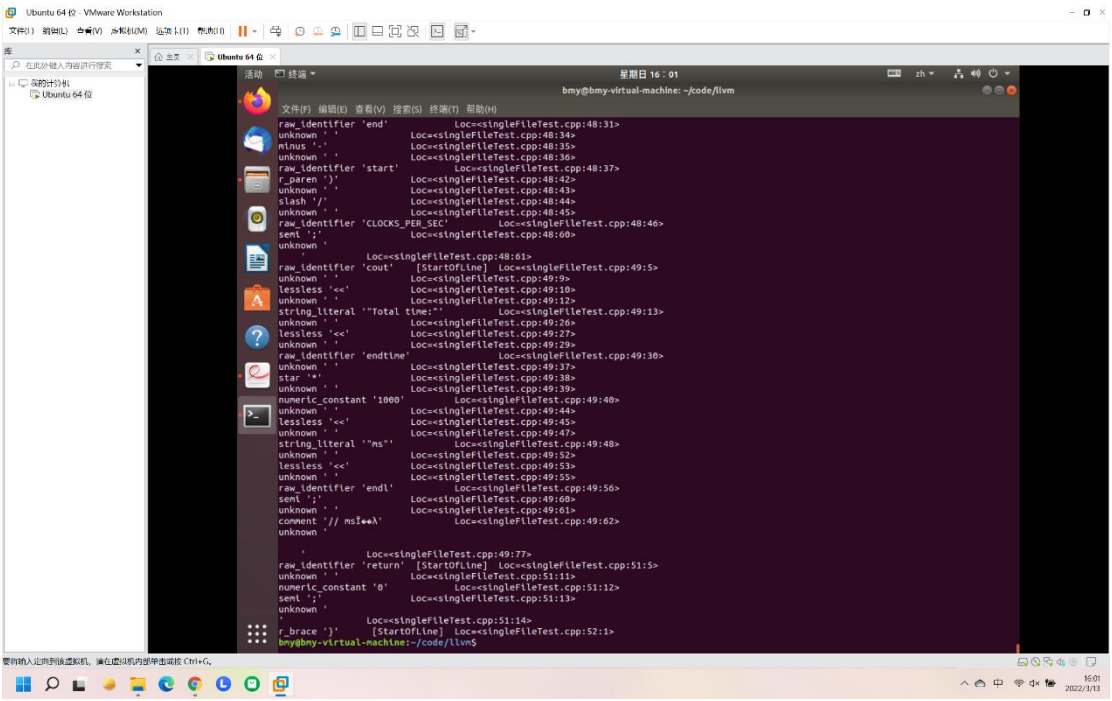
- 输入源代码文件；
- 预处理；
- 编译程序，进行词法分析、语法分析、语义分析、检查源代码是否错误，生成 IR 中间表示代码；
- 代码生成，生成汇编代码；
- 汇编，生成.obj 目标文件。

（5）查看词法分析结果：clang++ singleFileTest.cpp -Xclang -dump-tokens -c



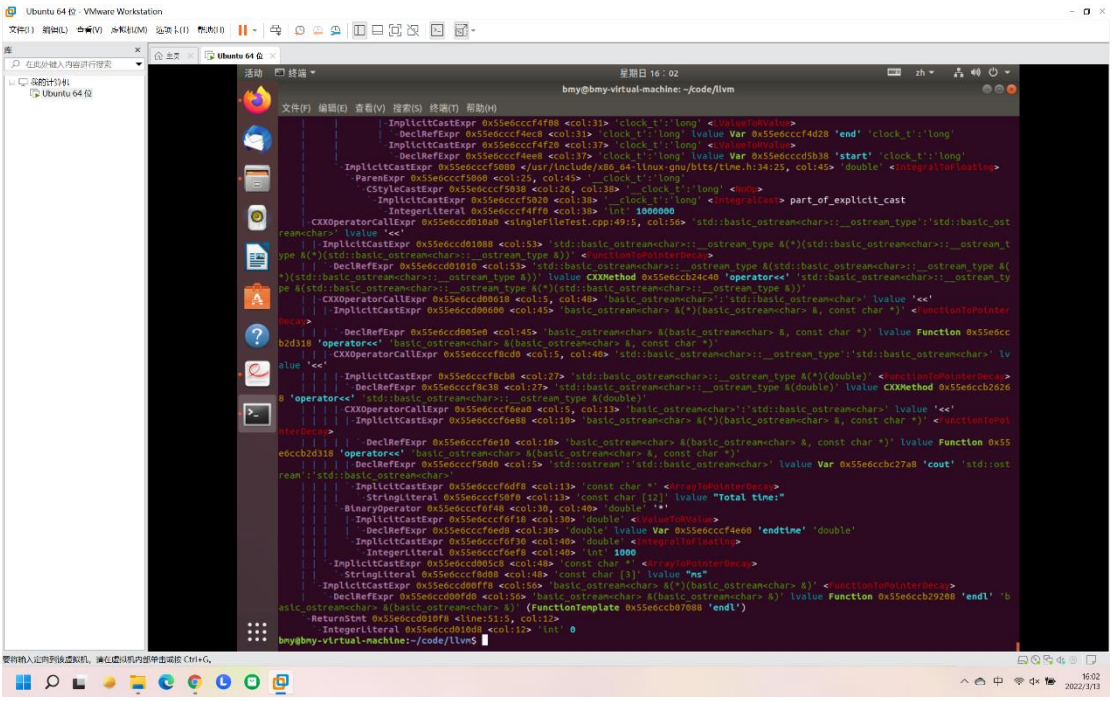
由图可知，该命令进行源程序词法分析，将代码切分成 Token，并输出源程序词法分析结果。

（6）查看词法分析结果 2：clang++ singleFileTest.cpp -Xclang -dump-raw-tokens -c



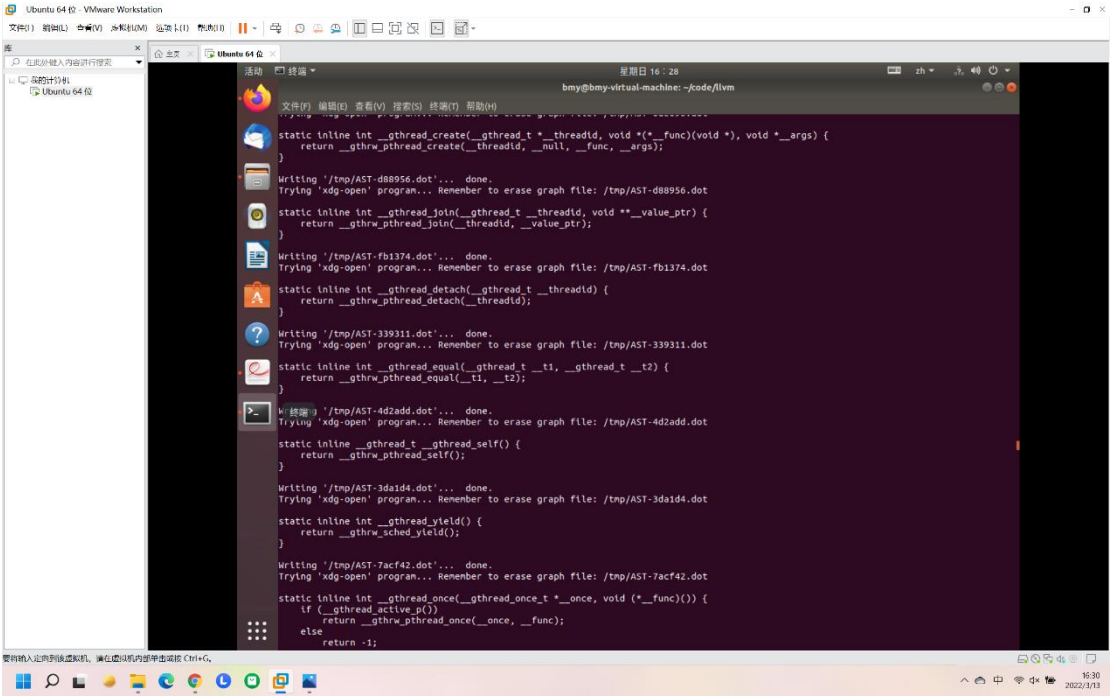
该命令进行词法分析，与（5）中命令的区别是，词法分析前并未对源程序进行预处理，源程序中的空格、换行符、注释等内容也进行了词法分析。

（7）查看语法分析结果：clang++ singleFileTest.cpp -Xclang -ast-dump -c

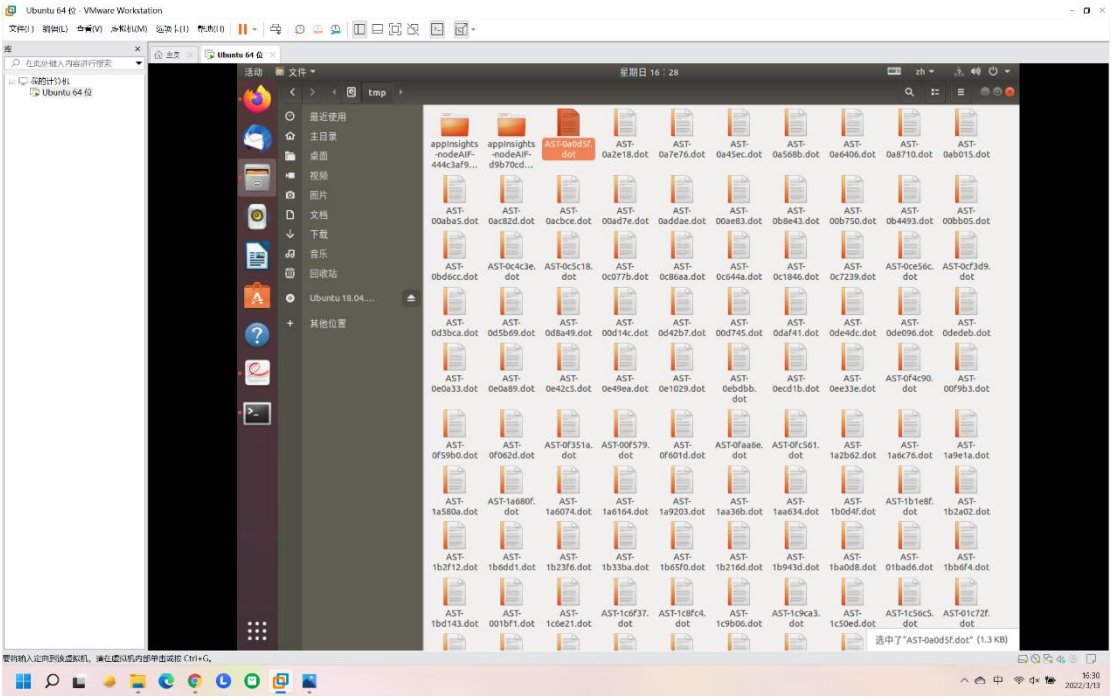


由图可知，该命令进行语法分析，检查源程序语法是否正确，以文本缩进的形式生成语法树。

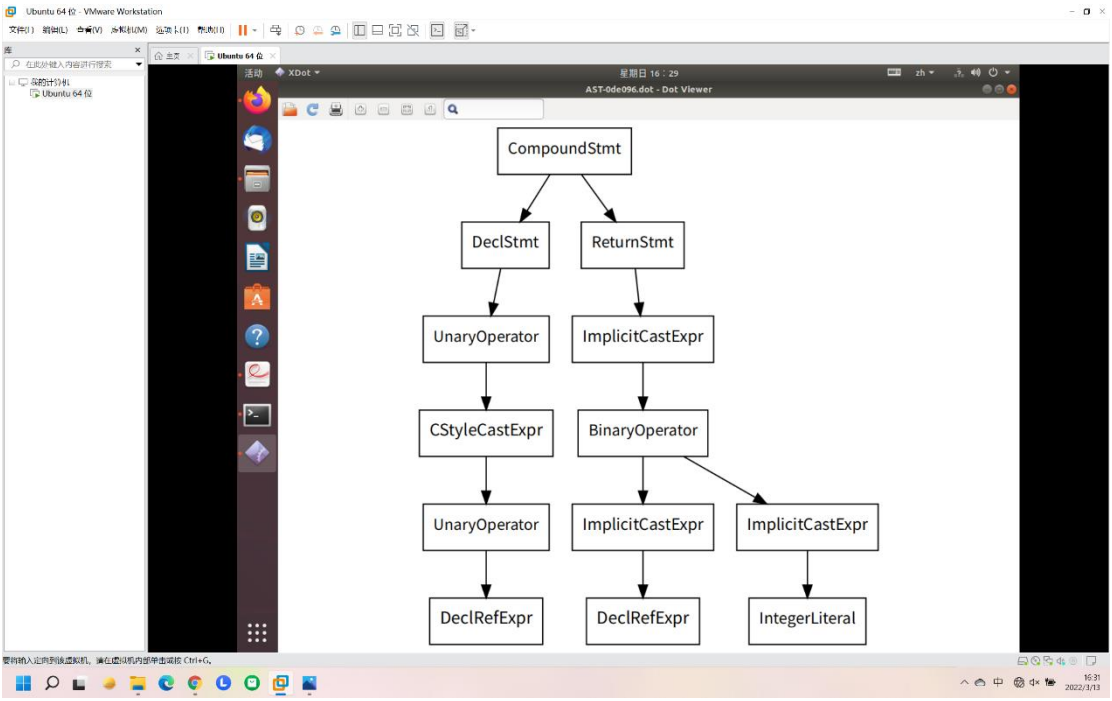
（8）查看语法分析结果 2：clang++ singleFileTest.cpp -Xclang -ast-view -c



由图可知，该命令进行语法分析，生成拓扑图结构的语法树.dot 文件，生成文件如下：

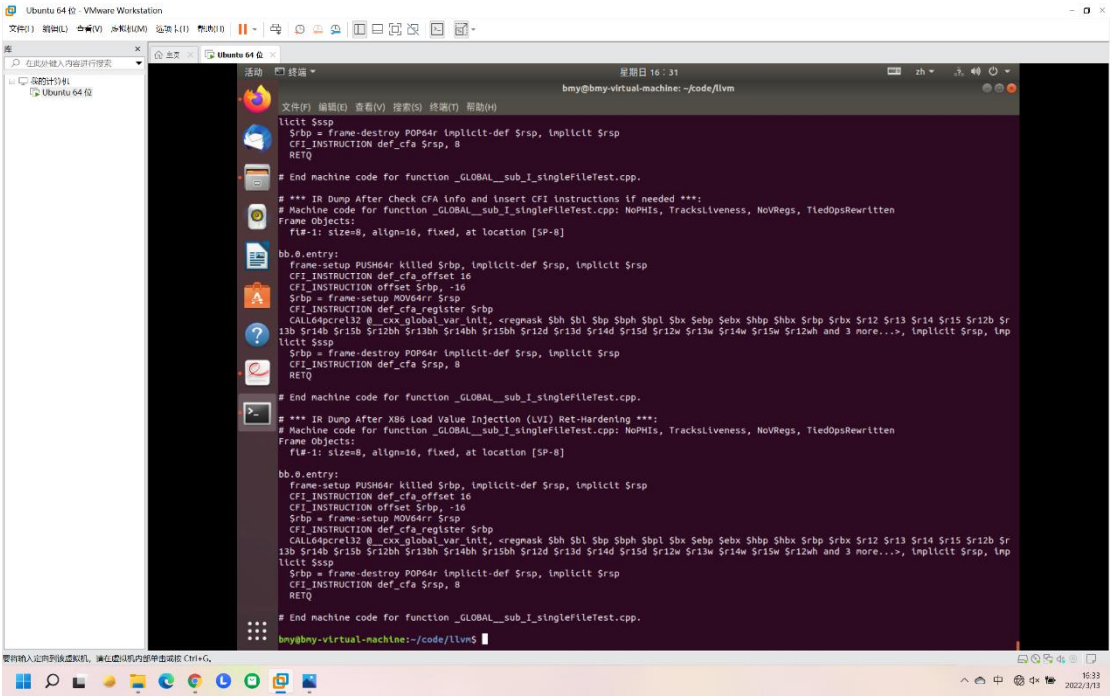


拓扑图结构的语法树示例如下：

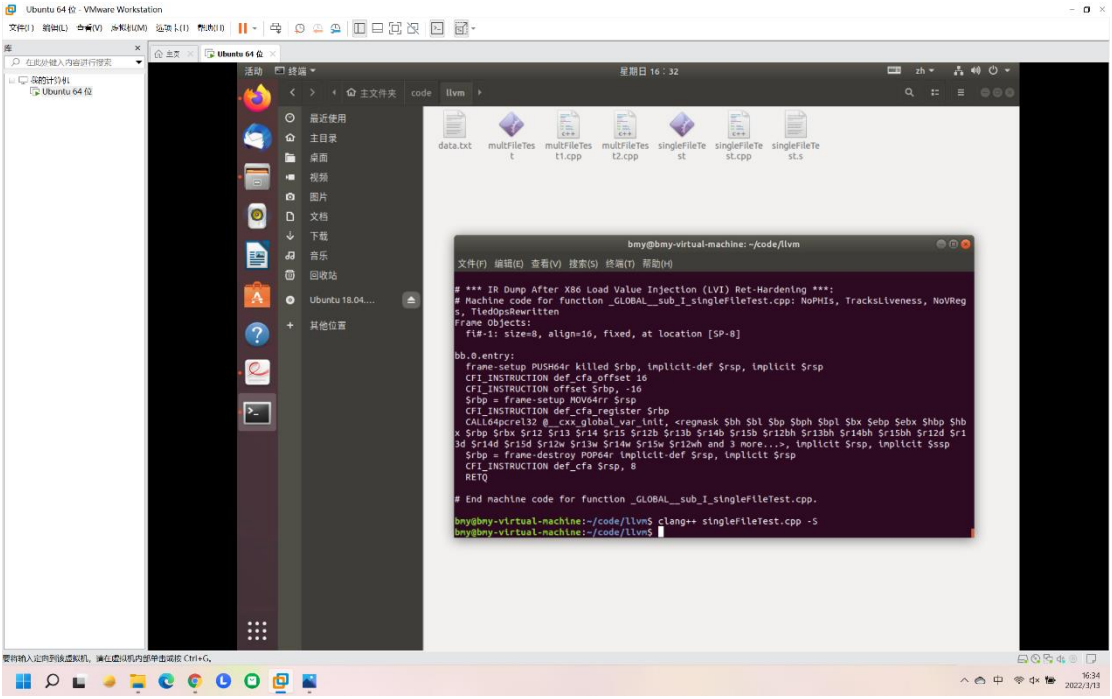


由图可知，语法树通过拓扑图表示代码间调用关系。

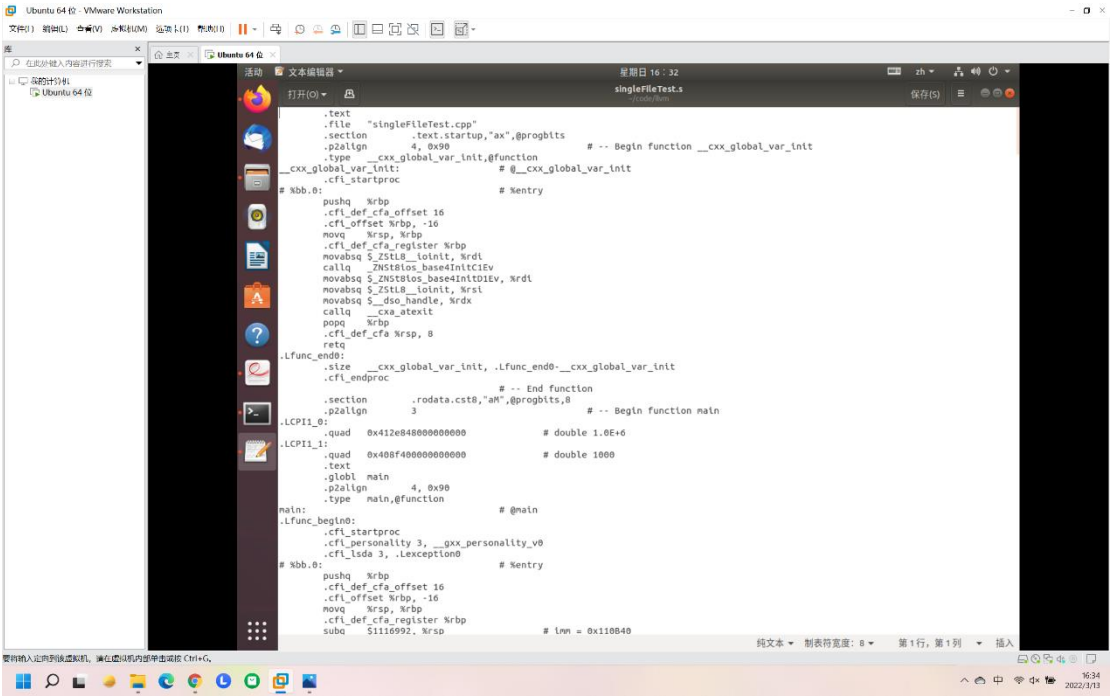
（9）查看编译优化的结果：clang++ singleFileTest.cpp -S -mllvm -print-after-all



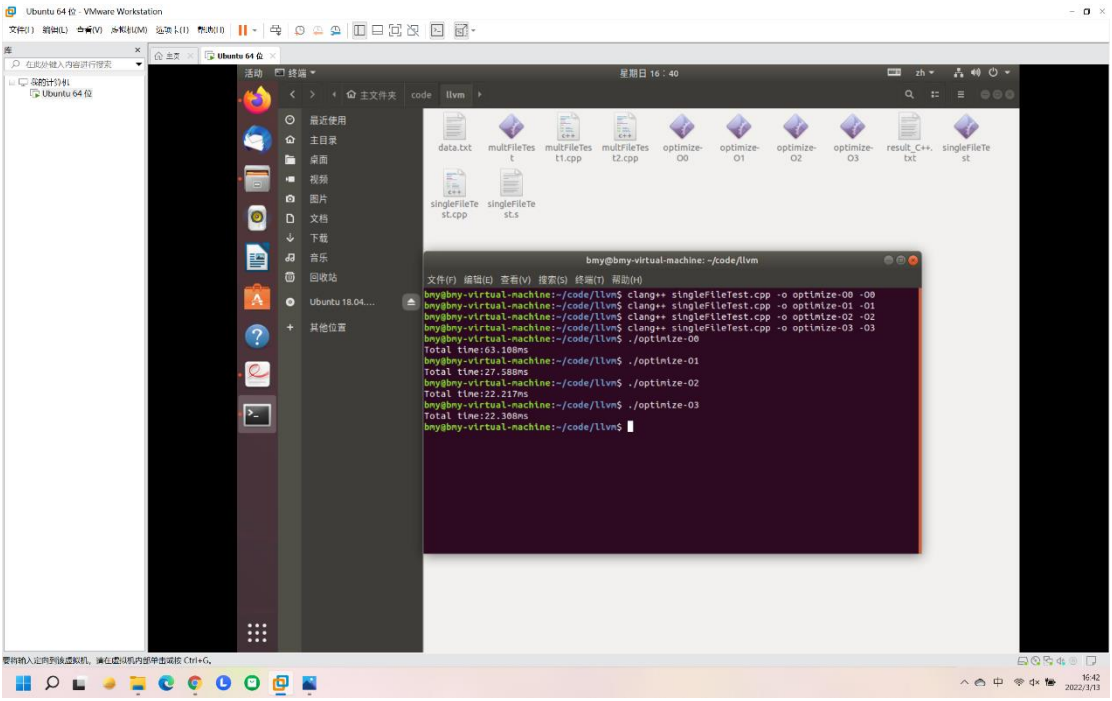
（10）查看生成的目标代码结果：clang++ singleFileTest.cpp –S



由图可知，该命令生成汇编语言文件，查看汇编代码如下：



（11）优化编译：clang++ singleFileTest.cpp -o hello -On，其中 n 分别为 0、1、2、3



分别使用-O0、-O1、-O2、-O3 优化命令优化编译，分别运行四种优化后的可执行文件 10 次，运行速度统计如下：

llvm	O0/ms	O1/ms	O2/ms	O3/ms
0	63.221	28.523	23.013	22.611
1	63.392	27.942	22.57	22.324
2	62.157	28.077	22.385	22.36
3	62.927	27.688	22.237	22.427
4	62.31	27.944	22.286	22.469
5	62.988	28.079	22.626	22.826
6	62.599	28.162	22.572	22.342
7	63.315	27.658	22.407	22.502
8	61.842	28.546	22.04	22.227
9	61.953	27.587	22.527	22.38
平均用时	62.6704	28.0206	22.4663	22.4468

由图可知，源程序经过 LLVM 编译器优化后，性能得到提升，且对测试程序，高级优化能够获得更好的性能。

6.3 GCC 与 LLVM 对比分析

GCC 和 LLVM 编译器各级优化后的程序平均运行时间对比如下：

	O0/ms	O1/ms	O2/ms	O3/ms
gcc	65.1807	24.1393	30.5457	19.0926
llvm	62.6704	28.0206	22.4663	22.4468

对比可知，从优化后的平均执行时间来看，LLVM 优化的程序运行效率高于 GCC。但这不是绝对的，存在 GCC 优化的程序运行效率高于 LLVM 的情况。

7. 实验心得体会

通过本次实验，我有如下收获：

（1）熟悉了 GCC 编译器和 LLVM 编译器，掌握了 GCC、LLVM 安装与使用的基本方法。实验中安装 GCC 编译器比较简单，可直接用 apt-get 安装；用手动编译的方式安装 LLVM 比较麻烦，对虚拟机配置要求较高，且比较费时。

（2）了解了 GCC、LLVM 编译的过程，观察、了解了编译器工作过程中生成的中间文件。GCC 和 LLVM 运行的过程基本包括：预处理、编译、后端生成汇编代码、汇编生成目标文件、文件链接生成可执行文件。

（3）了解了编译器的优化效果。编译器优化包括 O0、O1、O2、O3 四个等级，等级越高优化效果越好，但往往并不绝对，存在高级优化效率低于低级优化的情况。同时，LLVM 编译的文件执行效率往往高于 GCC 编译的文件。