

- Linux开发环境搭建
- GCC
  - GCC安装
  - GCC工作流程
  - 工作流程演示
- 静态库
  - 静态库的制作
    - 命名规则
    - 制作过程
    - 制作过程演示
  - 静态库的使用
- 动态库
  - 动态库的制作和使用
    - 命名规则
    - 动态库的制作
  - 动态库加载失败的原因
  - 解决动态库加载失败的问题
    - 方法一：添加环境变量LD\_LIBRARY\_PATH
      - 添加临时环境变量
      - 用户级别下配置永久环境变量
      - 系统级别下配置永久环境变量
    - 方法二：修改/etc/ld.so.cache文件列表
    - 方法三：把文件放入/lib/, /usr/lib目录
  - 静态库和动态库的对比
- Makefile
  - 为什么需要Makefile
  - Makefile文件命名和规则
    - 文件命名
    - Makefile规则
  - Makefile工作原理
  - Makefile变量
    - 规则
    - 用法举例：
    - 实际操作演示
  - Makefile模式匹配
  - Makefile函数
- GDB调试
  - 准备工作
  - GDB命令
- 文件I/O
  - 标准C库I/O函数和Linux系统I/O函数对比
  - 虚拟地址空间
  - 文件描述符
  - Linux系统I/O函数
  - 模拟实现ls-l指令
  - 文件属性操作函数

- 目录操作函数
- 目录遍历函数
- dup和dup2函数
- fcntl函数

# Linux开发环境搭建

这节课主要讲了如何安装vmware、vmware下如何安装ubuntu、使用Xshell和vscode连接ubuntu，详见视频。  
遇到问题先看评论区，找不到再看博客。

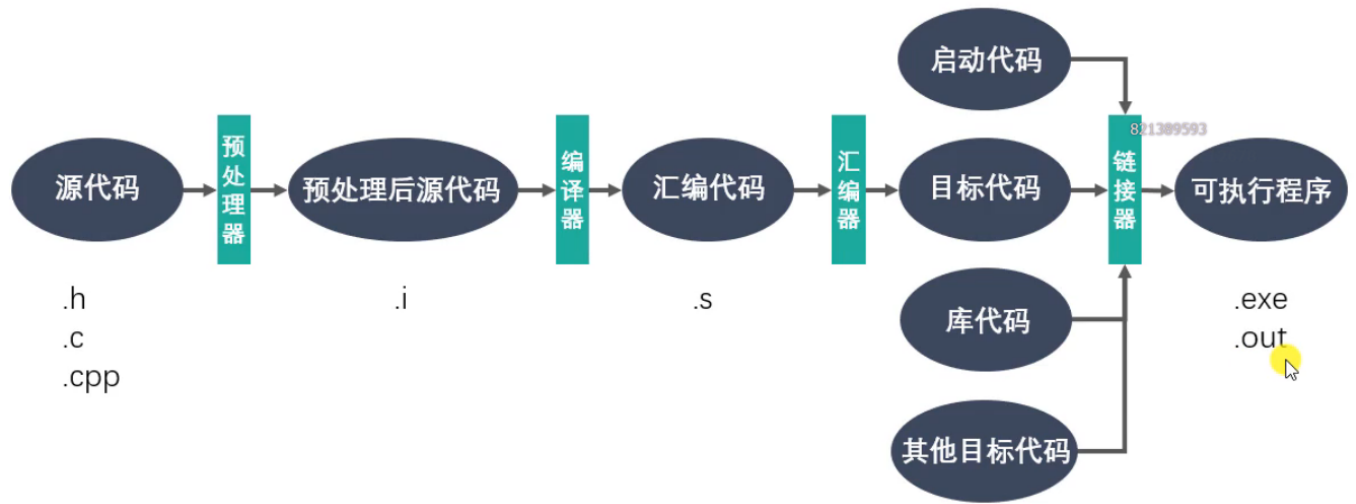
## GCC

### GCC安装

我们在编辑器写好代码后，要运行，这个时候就需要GCC工具的帮助了。  
在终端输入以下指令安装GCC以及检查版本：

```
sudo apt install gcc g++ # 安装
gcc/g++ -v/--version # 查询版本
```

### GCC工作流程




由图可以看到，我们首先需要在代码编辑器上书写源代码文件 **.h/.c/.cpp** 文件，源代码文件经预处理器处理后得到 **.i** 文件，之后再经过编译器得到汇编代码 **.s** 文件，接着通过汇编器处理得到目标代码文件 **.o** 文件；该文件和启动代码、库代码、其他目标代码经过链接器链接得到可执行程序 **.exe/.out** 文件。  
所以，整个工作流程先后产生如下文件：**.h/.c/.cpp -> .i -> .s -> .o -> .exe/.out**。

### 工作流程演示

首先需要了解GCC编译常用参数选项，如图：

15 / GCC常用参数选项

gcc编译选项	说明
-E	预处理指定的源文件，不进行编译
-S 	编译指定的源文件，但是不进行汇编
-c	编译、汇编指定的源文件，但是不进行链接
-o [file1] [file2] / [file2] -o [file1]	将文件 file2 编译成可执行文件 file1
-I directory	指定 include 包含文件的搜索目录
-g	在编译的时候，生成调试信息，该程序可以被调试器调试
-D	在程序编译的时候，指定一个宏
-w	不生成任何警告信息

首先在vscode上书写test.c文件，如下图：

test.c

test.c > main()

1

#include<stdio.h>

2

#include<stdlib.h>

3

4

#define PI 3.14

5

6

int main(){

7

8

// this is test code

9

int sum = PI + 10;

10

11

printf("Hello World!\n");

12

13

system("pause");

14

return 0;

15

}

接着在终端输入以下指令对文件进行预处理得到test.i文件，

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

打开test.i文件，发现前面有一长串对于#include库的展开；拉到最后面，可以看到预处理后的代码如下：

```

1865
1866
1867
1868  # 6 "test.c"
1869  int main(){
1870
1871
1872      int sum = 3.14 + 10;
1873
1874      printf("Hello World!\n");
1875
1876      system("pause");
1877      return 0;
1878  }
1879

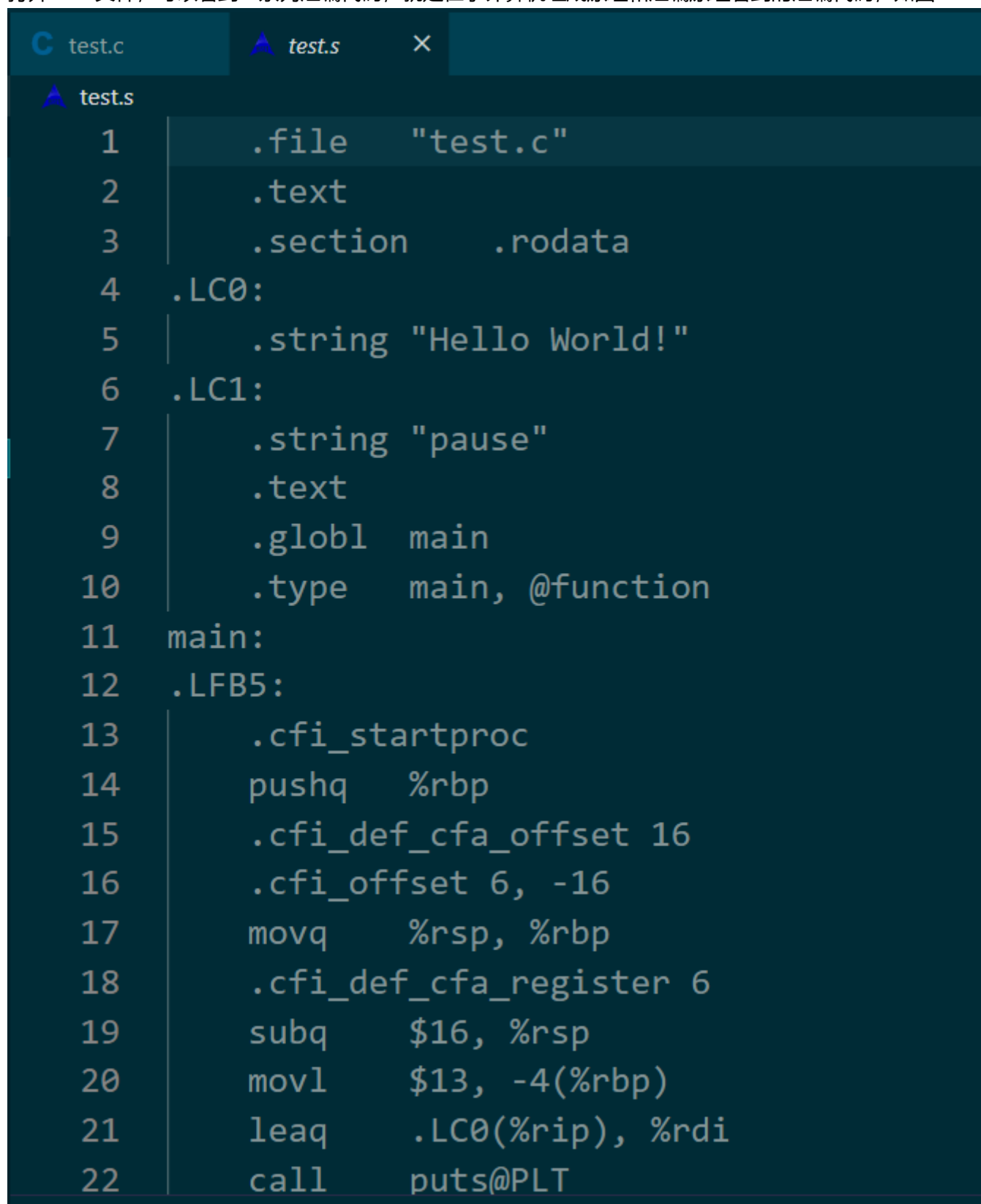
```

- 可以发现，#include引入的库已经在前面展开了，所以这里没有显示
- 同时#define定义的常量也被替换到代码里面了
- 注释this is test code也消失了

之后对test.i文件进行编译操作，得到test.s文件，

```
gcc test.i -S -o test.s
```

打开test.s文件，可以看到一系列汇编代码，就是在学计算机组成原理和汇编原理看到的汇编代码，如图：

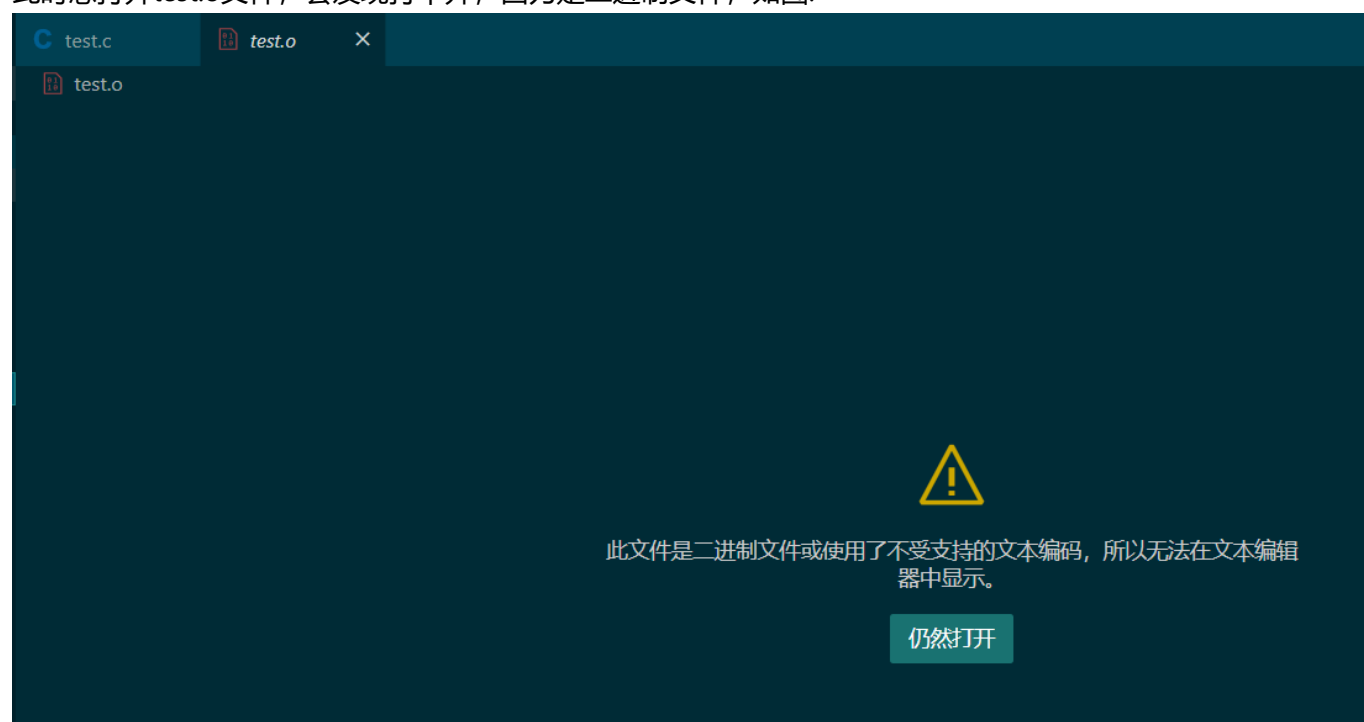


```
test.c  test.s  X
test.s
1      .file    "test.c"
2      .text
3      .section    .rodata
4      .LC0:
5      .string "Hello World!"
6      .LC1:
7      .string "pause"
8      .text
9      .globl  main
10     .type   main, @function
11     main:
12     .LFB5:
13     .cfi_startproc
14     pushq   %rbp
15     .cfi_def_cfa_offset 16
16     .cfi_offset 6, -16
17     movq    %rsp, %rbp
18     .cfi_def_cfa_register 6
19     subq    $16, %rsp
20     movl    $13, -4(%rbp)
21     leaq    .LC0(%rip), %rdi
22     call    puts@PLT
```

之后对test.s文件进行汇编操作，得到test.o文件，

```
gcc test.s -s -o test.o
```

此时想打开test.o文件，会发现打不开，因为是二进制文件，如图：



链接生成可执行文件，直接输入以下代码一步到位：

```
./test.o
```

如图：

```
eason@eason:~/Linux$ ./test.o
Hello World!
sh: 1: pause: not found
```

## 静态库

### 静态库的制作

#### 命名规则

- Linux: libxxx.a  
lib: 前缀，固定  
xxx: 库的名字，自己定  
.a: 后缀，固定
- Windows: libxxx.lib

#### 制作过程

- gcc获得 .o 文件

- 将 .o 文件打包, 使用ar工具(archive)

```
ar rcs libxxx.a xxx.o xxx.o
r-将文件插入备存文件中
c-建立备存文件
s-索引
```

## 制作过程演示

往ubuntu里面拖入两个文件: calc和library(在课程文件里下载), 打开后发现calc有如下文件:

```
nowcoder@nowcoder:~/Linux/lesson04/calc$ ls
add.c  div.c  head.h  main.c  mult.c  sub.c
nowcoder@nowcoder:~/Linux/lesson04/calc$ tree
.
├── add.c
├── div.c
├── head.h
├── main.c
├── mult.c
└── sub.c
```

821389593

因为制作需要 .o 文件, 所以需要先根据calc里面的 .c 文件编译汇编得到, 输入如下指令:

```
gcc -c add.c div.c mult.c sub.c
```

可以得到如图所示效果:

```
nowcoder@nowcoder:~/Linux/lesson04/calc$ gcc -c add.c div.c mult.c sub.c
nowcoder@nowcoder:~/Linux/lesson04/calc$ ls
add.c  add.o  div.c  div.o  head.h  main.c  mult.c  mult.o  sub.c  sub.o
nowcoder@nowcoder:~/Linux/lesson04/calc$ tree
.
├── add.c
├── add.o
├── div.c
├── div.o
├── head.h
├── main.c
├── mult.c
├── mult.o
├── sub.c
└── sub.o
```

接着将 .o 文件打包得到静态库文件，首先输入以下指令：

```
ar rcs libcalc.a add.o div.o mult.o sub.o
```

可以得到所示文件：

```
nowcoder@nowcoder:~/Linux/lesson04/calc$ ar rcs libcalc.a add.o sub.o mult.o div.o
nowcoder@nowcoder:~/Linux/lesson04/calc$ ls
add.c add.o div.c div.o head.h libcalc.a main.c mult.c mult.o sub.c sub.o
nowcoder@nowcoder:~/Linux/lesson04/calc$ tree
.
├── add.c
├── add.o
├── div.c
├── div.o
├── head.h
├── libcalc.a
├── main.c
├── mult.c
├── mult.o
├── sub.c
└── sub.o
```

## 静态库的使用

打开引入的library文件，看里面有什么内容：

```
nowcoder@nowcoder:~/Linux/lesson05$ cd library/
nowcoder@nowcoder:~/Linux/lesson05/library$ tree
.
├── include
│   └── head.h
├── lib
├── main.c
└── src
    ├── add.c
    ├── div.c
    ├── mult.c
    └── sub.c
```

可以发现里面有几个文件：include、lib、main.c、src。其实我们平常在github上下载的源码的结构也和这个类似，include里面存储的是我们需要的头文件，lib里面存储的是操作需要用到的库，main.c是运行文件，src文件夹里面存储的是头文件里面所声明的函数的定义。

同时可以看到此时lib文件夹里面是没有我们生成的静态库的，可以使用src里面的源码再次生成，也可以使用 **cp** 命令把我们在calc文件夹下生成的libcalc.a的库复制到lib文件夹里面。如下：



```
cp libcalc.a ../library/lib/
```

操作后可以发现文件结构发生如下变化：

```
nowcoder@nowcoder:~/Linux/lesson05/library$ tree
.
├── include
│   └── head.h
├── lib
│   └── libcalc.a
├── main.c
└── src
    ├── add.c
    ├── div.c
    ├── mult.c
    └── sub.c
```

之后往终端输入以下指令编译main.c：

```
gcc main.c -o app
```

可以发现出现以下错误：

```
nowcoder@nowcoder:~/Linux/lesson05/library$ tree
.
├── include
│   └── head.h
├── lib
│   └── libcalc.a
├── main.c
└── src
    ├── add.c
    ├── div.c
    ├── mult.c
    └── sub.c

3 directories, 7 files
nowcoder@nowcoder:~/Linux/lesson05/library$ gcc main.c -o app
main.c:2:10: fatal error: head.h: 没有那个文件或目录
  #include "head.h"
          ^~~~~~
compilation terminated.
```

错误的意思是没有在main.c的同级目录下发现它所需要的head.h头文件。根据文件目录结构可以发现，该头文件在include目录下，所以需要以下指令进行编译：

```
gcc main.c -o app -I ./include/  
-I 表示指定include包含文件的搜索目录
```

但是又发现报错如下：

```
nowcoder@nowcoder:~/Linux/lesson05/library$ gcc main.c -o app -I ./include/  
/tmp/ccP5mHri.o: 在函数‘main’中：  
main.c:(.text+0x3a): 对‘add’未定义的引用  
main.c:(.text+0x5c): 对‘subtract’未定义的引用  
main.c:(.text+0x7e): 对‘multiply’未定义的引用  
main.c:(.text+0xa0): 对‘divide’未定义的引用  
collect2: error: ld returned 1 exit status
```

该错误的意思是，在head.h头文件里面只发现了这些函数的声明，但是没有定义，这个时候就要在命令里面加上我们生成的静态库。因为静态库是根据这几个函数的源码编译汇编产生的 .o 文件打包得到的，所以可以输入以下指令：

```
gcc main.c -o app -I ./include/ -l calc -L ./lib  
-l 指定所需要库的名称，注意是库的名称calc不是静态库文件libcalc.a  
-L 指定所需要库的查找目录，即该库是存放在哪个目录下的
```

之后可以发现命令没有报错，而且可以得到图示的文件和执行效果：

```
nowcoder@nowcoder:~/Linux/lesson05/library$ ls  
app include lib main.c src  
nowcoder@nowcoder:~/Linux/lesson05/library$ ./app  
a = 20, b = 12  
a + b = 32  
a - b = 8  
a * b = 240  
a / b = 1.666667  
nowcoder@nowcoder:~/Linux/lesson05/library$
```

## 动态库

### 动态库的制作和使用

#### 命名规则

- Linux: libxxx.so  
lib: 前缀，固定  
xxx: 库的名字，自己定  
.so: 后缀，固定  
在linux下是一个可执行文件

- Windows: libxxx.dll

## 动态库的制作

- gcc得到 .o 文件，得到和位置无关的代码

```
gcc -c -fpic/fPIC a.c b.c
```

- gcc得到动态库

```
gcc -shared a.o b.o -o libXXX.so
```

接下去的过程和静态库的制作使用过程一样，但是当输入指令生成可执行文件时候，会报如下错误，意思是动态库加载失败，这个时候就要去了解动态库实现原理从而去解决这个问题

```
eason@eason:~/Linux/library$ gcc main.c -o main.a -I ./include/
/tmp/ccz3tGXz.o: 在函数'main'中:
main.c:(.text+0x3a): 对'add'未定义的引用
main.c:(.text+0x5c): 对'subtract'未定义的引用
main.c:(.text+0x7e): 对'multiply'未定义的引用
main.c:(.text+0xa0): 对'divide'未定义的引用
collect2: error: ld returned 1 exit status
eason@eason:~/Linux/library$ gcc main.c -o main.a -I ./include/ -l calc -L ./lib/
eason@eason:~/Linux/library$ tree
.
├── include
│   └── head.h
├── lib
│   └── libcalc.so
├── main.a
├── main.c
└── src
    ├── add.c
    ├── div.c
    ├── mult.c
    └── sub.c

3 directories, 8 files
eason@eason:~/Linux/library$ ./main.a
./main.a: error while loading shared libraries: libcalc.so: cannot open shared object file: No such file or directory
```

## 动态库加载失败的原因

首先需要了解静态库和动态库工作的原理：

- 静态库：GCC进行链接时，会把静态库中的代码 **打包** 到可执行程序中
- 动态库：GCC进行链接时，动态库的代码 **不会被打包** 到可执行程序中；而是在启动后动态库才会被动态加载到内存中

可以通过ldd (list dynamic dependencies) 命令查询动态库依赖关系证明这一定，如下图：

```
eason@eason:~/Linux/library$ ll
总用量 36
drwxrwxr-x 5 eason eason 4096 7月 27 13:51 ./
drwxrwxr-x 6 eason eason 4096 7月 27 09:58 ../
drwxrwxr-x 2 eason eason 4096 7月 27 09:51 include/
drwxrwxr-x 2 eason eason 4096 7月 27 13:49 lib/
-rwxrwxr-x 1 eason eason 8424 7月 27 13:51 main.a*
-rw-rw-r-- 1 eason eason 306 7月 27 09:51 main.c
drwxrwxr-x 2 eason eason 4096 7月 27 09:51 src/
eason@eason:~/Linux/library$ ldd main.a
        linux-vdso.so.1 (0x00007ffe771b2000)
        libcalc.so => not found
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe3c9f1e000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fe3ca511000)
eason@eason:~/Linux/library$
```

可以发现libcalc.so后面是not found，表示只是把动态库信息加载到代码中了，但是动态库代码却没有放进去，这个时候就需要ld动态载入器来获取依赖库的绝对路径去解决问题，如下图：

### ■ 如何定位共享库文件呢？

当系统加载可执行代码时候，能够知道其所依赖的库的名字，但是还需要知道绝对路径。此时就需要系统的动态载入器来获取该绝对路径。对于elf格式的可执行程序，是由ld-linux.so来完成的，它先后搜索elf文件的 **DT\_RPATH**段 → 环境变量 **LD\_LIBRARY\_PATH** → **/etc/ld.so.cache**文件列表 → **/lib/**，**/usr/lib**目录找到库文件后将其载入内存。

## 解决动态库加载失败的问题

因为DT\_RPATH段是不可以修改的，所以我们只能从后面三个入手解决。

方法一：添加环境变量LD\_LIBRARY\_PATH

### 添加临时环境变量

输入以下指令给动态库添加环境变量：

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/eason/Linux/library/lib
其中：后面的内容是libcalc.so所在的绝对路径
```

添加完成后，可以通过以下指令查看是否添加成功：

```
echo $LD_LIBRARY_PATH
终端的输出为：:/home/eason/Linux/library/lib
```

这个时候输入ldd命令查看main.a的依赖关系，可以发现动态库有依赖了，依赖正好是我们添加的环境变量，而且可执行文件也可以运行了，如图：

```
eason@eason:~/Linux/library$ tree
.
├── include
│   └── head.h
├── lib
│   └── libcalc.so
├── main.a
├── main.c
└── src
    ├── add.c
    ├── div.c
    ├── mult.c
    └── sub.c

3 directories, 8 files
eason@eason:~/Linux/library$ ldd main.a
        linux-vdso.so.1 (0x00007ffc3058e000)
        libcalc.so => /home/eason/Linux/library/lib/libcalc.so (0x00007fef674ee000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fef670fd000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fef678f2000)
eason@eason:~/Linux/library$ ./main
-bash: ./main: 没有那个文件或目录
eason@eason:~/Linux/library$ ./main.a
a = 20, b = 12
a + b = 32
a - b = 8
a * b = 240
a / b = 1.666667
```

但是这个方法有个问题，当我们退出当前终端，重新打开另一个终端再次执行main.a的时候，会再次报动态库加载失败的错误，见图最下面：

```
Your Hardware Enablement Stack (HWE) is supported until April 2023.
Last login: Thu Jul 27 14:41:38 2023 from 192.168.108.1
eason@eason:~$ ls
examples.desktop  Linux  snap  公共的  模板  视频  图片  文档  下载  音乐  桌面
eason@eason:~$ cd Linux/
eason@eason:~/Linux$ ld
ld: 没有输入文件
eason@eason:~/Linux$ ls
calc  gcc-test  library
eason@eason:~/Linux$ cd lib
-bash: cd: lib: 没有那个文件或目录
eason@eason:~/Linux$ cd library/
eason@eason:~/Linux/library$ tree
.
├── include
│   └── head.h
├── lib
│   └── libcalc.so
├── main.a
├── main.c
└── src
    ├── add.c
    ├── div.c
    ├── mult.c
    └── sub.c

3 directories, 8 files
eason@eason:~/Linux/library$ ./main.a
./main.a: error while loading shared libraries: libcalc.so: cannot open shared object file: No such file or directory
```

原因是之前配置的环境变量是在那个终端中的环境中配置的，是临时的，所以重新打开一个终端后会再次出现同样错误。

有两种方式解决临时配置环境变量的问题，一是用户级别下配置永久的环境变量，二是系统级别下配置。

### 用户级别下配置永久环境变量

首先返回根目录，输入指令 `ll`，会发现有如图所示的 `.bashrc` 文件，我们通过修改该文件实现用户级别下配置永久环境变量。

```
eason@eason:~/Linux/library$ cd ~
eason@eason:~$ ls
examples.desktop  Linux  snap  公共的  模板  视频  图片  文档  下载  音乐  桌面
eason@eason:~$ ll
总用量 120
drwxr-xr-x 19 eason eason 4096 7月 27 14:55 ./
drwxr-xr-x  3 root  root  4096 7月 26 04:16 ../
-rw-r--r--  1 eason eason 1512 7月 27 14:54 .bash_history
-rw-r--r--  1 eason eason  220 7月 26 04:16 .bash_logout
-rw-r--r--  1 eason eason 3771 7月 26 04:16 .bashrc
drwx----- 13 eason eason 4096 7月 26 05:11 .cache/
drwx----- 13 eason eason 4096 7月 26 04:40 .config/
drwxrwxr-x  3 eason eason 4096 7月 26 05:06 .dotnet/
-rw-r--r--  1 eason eason 8980 7月 26 04:16 examples.desktop
drwx-----  3 eason eason 4096 7月 26 04:36 .gnupg/
-rw-r--r--  1 eason eason  628 7月 27 09:11 .ICEauthority
drwxrwxr-x  6 eason eason 4096 7月 27 09:58 Linux/
drwx-----  3 eason eason 4096 7月 26 04:25 .local/
-rw-r--r--  1 eason eason  807 7月 26 04:16 .profile
drwx-----  3 eason eason 4096 7月 26 04:38 snap/
drwx-----  2 eason eason 4096 7月 26 05:29 .ssh/
-rw-r--r--  1 eason eason    0 7月 26 04:43 .sudo_as_admin_successful
-rw-r--r--  1 eason eason  790 7月 26 05:29 .viminfo
drwxrwxr-x  5 eason eason 4096 7月 27 09:11 .vscode-server/
-rw-rw-r--  1 eason eason  183 7月 26 05:06 .wget-hsts
```

- 在命令行输入 `vim .bashrc`，对该文件进行修改
- 在文件最后一行插入环境变量 `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/eason/Linux/library/lib`
- 保存修改退出，输入 `source .bashrc` 使得修改生效

此时执行 `ldd main.a` 发现动态库有依赖，再次执行 `main.a` 文件发现执行成功！

### 系统级别下配置永久环境变量

和用户级别下一样是修改文件来添加依赖，只不过要修改的文件在 `/etc/profile` 下，步骤如下：

- 命令行输入 `sudo vim /etc/profile`
- 在文件最后一行插入环境变量 `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/eason/Linux/library/lib`
- 保存修改退出，输入 `source /etc/profile` 使得修改生效

此时执行 `ldd main.a` 发现动态库有依赖，再次执行 `main.a` 文件发现执行成功！

### 方法二：修改 `/etc/ld.so.cache` 文件列表

- 在终端输入 `vim /etc/ld.so.cache` 命令修改该文件，但是发现该文件是一个二进制文件，所以无法修改
- 所以通过间接修改 `/etc/ld.so.conf` 文件，终端输入指令 `sudo vim /etc/ld.so.conf` 进入修改，在文件最后一行插入环境变量 `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/eason/Linux/library/lib`
- 保存修改退出，输入 `sudo ldconfig` 使得修改生效

此时执行 `ldd main.a` 发现动态库有依赖，再次执行 `main.a` 文件发现执行成功！

### 方法三：把文件放入 `/lib/`，`/usr/lib` 目录



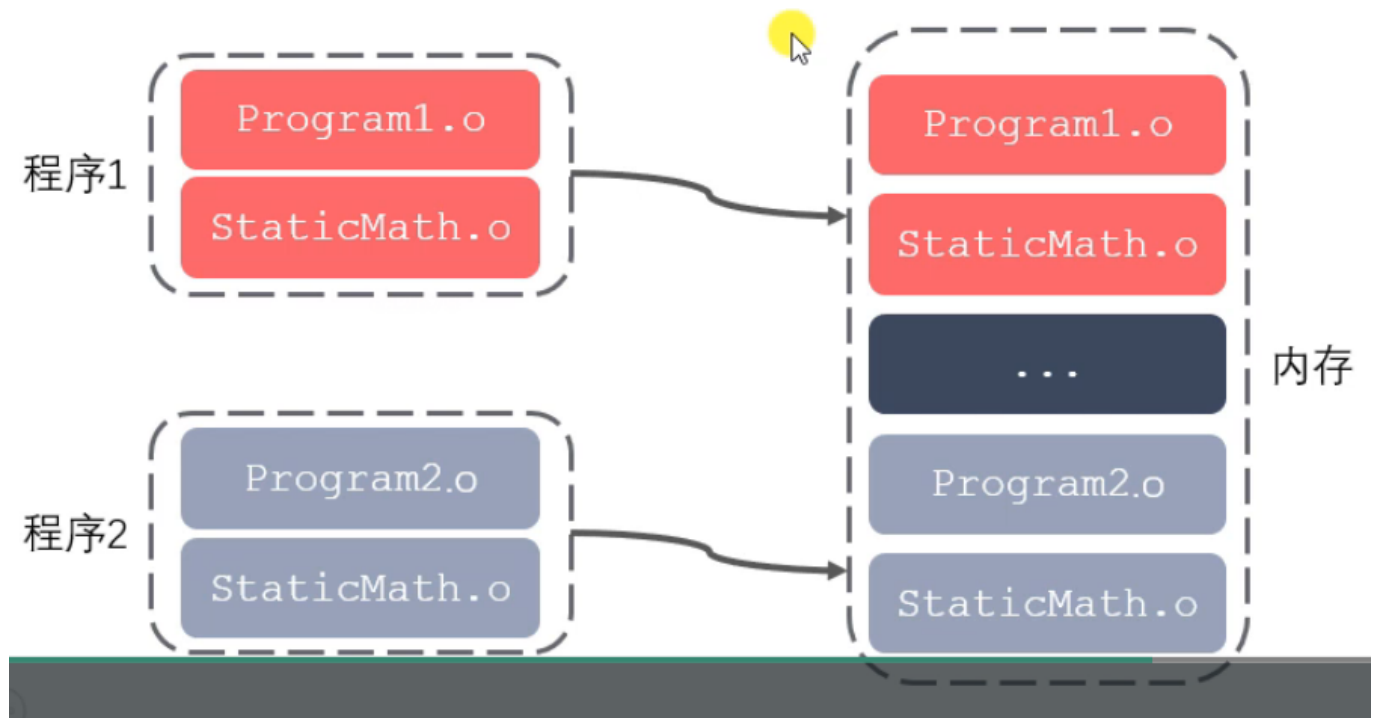
不推荐，因为这两个目录里面就有很多系统文件，可能自己添加的文件和系统文件一样导致冲突。

## 静态库和动态库的对比

静态库优缺点：

- 优点
  - 静态库被打包到应用程序中，加载速度快
  - 发布程序无需提供静态库，移植方便
- 缺点
  - 消耗系统资源，浪费内存
  - 更新、部署、发布麻烦

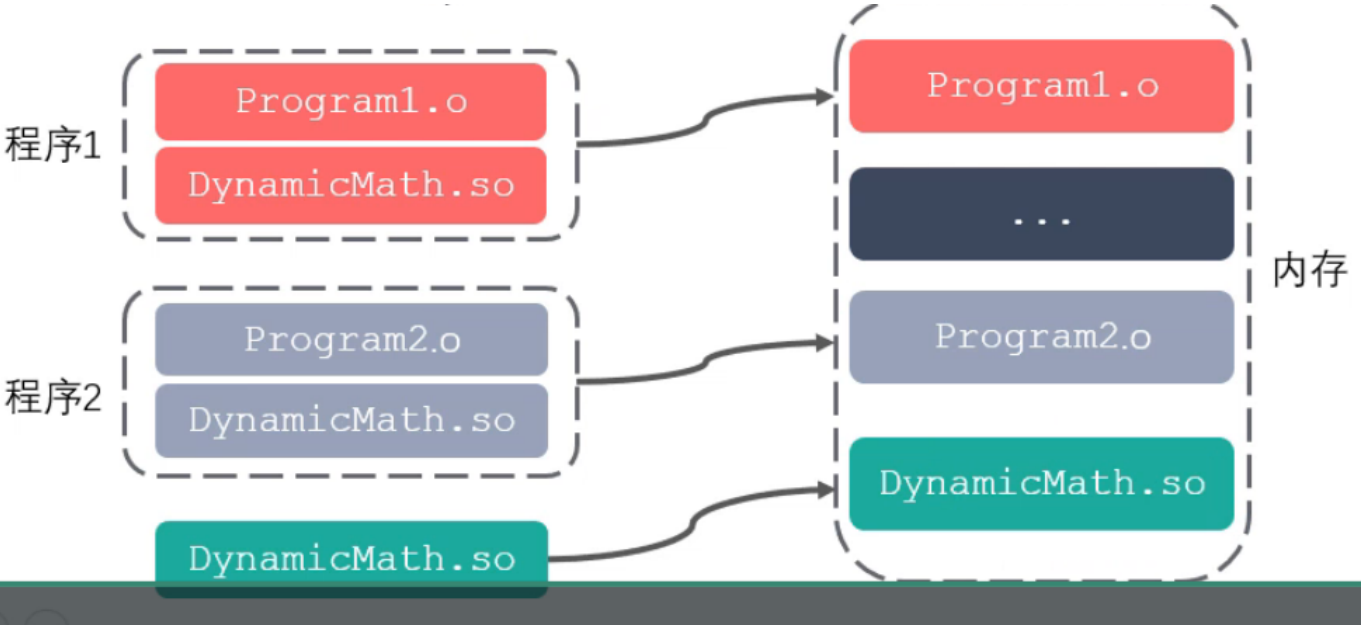
所以一般在库较小的时候会使用静态库。



动态库优缺点：

- 优点
  - 可以实现进程间资源共享（共享库）
  - 更新、部署、发布简单
  - 可以控制何时加载动态库
- 缺点
  - 加载速度比静态库慢
  - 发布程序时需要提供依赖的动态库

所以一般在库较大时使用动态库。



# Makefile

## 为什么需要Makefile

在实际的工程中，源文件是非常多的，通常我们可以按照它的类型、功能等的放到多个目录下。所以对应的，对这些工程文件进行编译运行，肯定不是只输入一条简单的gcc命令就可以的，需要输入多条gcc命令以执行。但是我们不可能每次执行的时候都输入多条gcc命令，这样的话工程量大、繁杂，而且容易出错。这个时候Makefile的重要性就体现出来了，他通过一系列的规则指定那些文件先编译、后编译、重新编译，所以它就像一个脚本一样。总的来说，它的好处在于“自动化编译”，只需要一个make命令，这个工程就可以根据Makefile文件实现自动编译。

## Makefile文件命名和规则

### 文件命名

makefile或Makefile

### Makefile规则

一个Makefile文件中可以有一个或多个规则

目标...: 依赖...

命令 (shell命令)

...

目标: 最终要生成的文件

依赖: 生成目标所需要的文件或目标

命令: 通过执行命令对依赖操作生成目标, 命令前必须Tab缩进

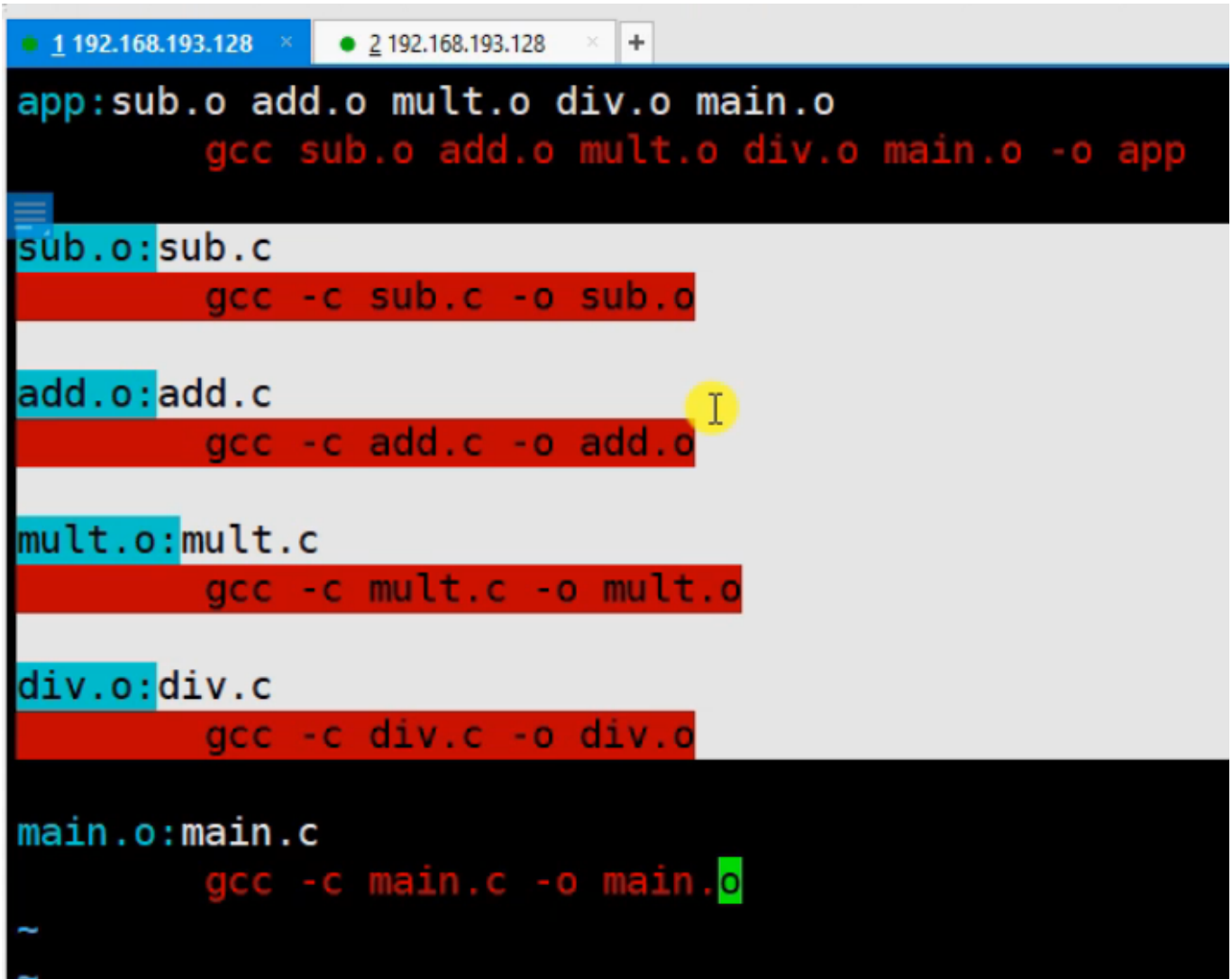
Makefile中其他规则一般为第一条规则服务

## Makefile工作原理



- 命令在执行前，需要先检查规则中的依赖是否存在
  - 如果存在，执行命令
  - 如果不存在，向下检查其它规则，检查是否有一个规则是用来生成这个依赖的，若有则执行
- 检查更新，在执行规则中的命令时，会比较目标和依赖的时间
  - 如果依赖时间比目标时间晚，需要重新生成目标
  - 如果依赖时间比目标时间早，目标不需要更新，对应规则中的命令不需要执行

以上原理可以结合两张图理解，同时做一个对比：



```
1 192.168.193.128 x 2 192.168.193.128 x +
app:sub.o add.o mult.o div.o main.o
    gcc sub.o add.o mult.o div.o main.o -o app

sub.o:sub.c
    gcc -c sub.c -o sub.o

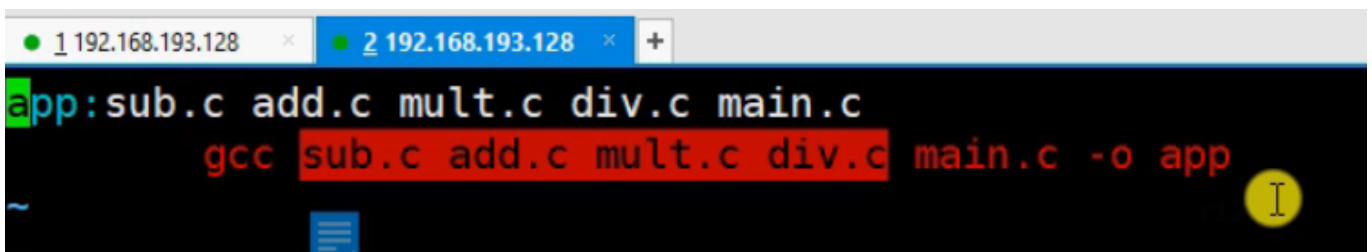
add.o:add.c
    gcc -c add.c -o add.o

mult.o:mult.c
    gcc -c mult.c -o mult.o

div.o:div.c
    gcc -c div.c -o div.o

main.o:main.c
    gcc -c main.c -o main.o

~
```



```
1 192.168.193.128 x 2 192.168.193.128 x +
app:sub.c add.c mult.c div.c main.c
    gcc sub.c add.c mult.c div.c main.c -o app

~
```

如上两图，如果main.c发生了改变，则第一种写法比第二种写法要好，效率更高。

以图一举例，main.o: main.c这一行中，依赖比目标的时间晚，所以要重新生成目标main.o。所以在第一张图中，第一行和最后一行都要重新编译；而在图二中，则所有文件都要重新编译。所以图一的写法比图二好。

感觉这里也体现了耦合与解耦合的思想。

## Makefile变量

## 规则

- 自定义变量
  - 变量名 = 变量值, 例如var=hello
- 预定义变量
  - AR: 归档维护程序的名称, 默认值为ar
  - CC: C编译器的名称, 默认值为cc
  - CXX: C++编译器的名称, 默认值为g++
  - \$@: 获取目标的完整名称
  - \$<: 获取第一个依赖文件的名称
  - \$^: 获取所有依赖文件
- 获取变量的值
  - \$(变量名)

## 用法举例:

- 上面定义了变量var=hello, 所以可以使用\$(var)获取var的变量值hello
- 例如现在定义了如下规则和命令

```
app: main.c a.c b.c
    gcc -c main.c a.c b.c -o app
```

可以改写为: **注意: 自动变量只能在规则的命令中使用**

```
app: main.c a.c b.c
    $(CC) -c $^ -o $@
```

## 实际操作演示

如图，我们有一个这样的Makefile文件：

```
main.a: add.o sub.o div.o mult.o main.o
        gcc add.o sub.o div.o mult.o main.o -o main.a

sub.o:sub.c
        gcc -c sub.c -o sub.o

add.o:add.c
        gcc -c add.c -o add.o

div.o:div.c
        gcc -c div.c -o div.o

mult.o:mult.c
        gcc -c mult.c -o mult.o

main.o:main.c
        gcc -c main.c -o main.o
```

使用上面的规则，可以改写成如下的：

```
#定义变量
src=add.o sub.o div.o mult.o main.o
target=main.a

$(target):$(src)
    $(CC) $(src) -o $target

sub.o:sub.c
    gcc -c sub.c -o sub.o

add.o:add.c
    gcc -c add.c -o add.o

div.o:div.c
    gcc -c div.c -o div.o

mult.o:mult.c
    gcc -c mult.c -o mult.o

main.o:main.c
    gcc -c main.c -o main.o
```

但是可以看到，这个文件还是有点麻烦的，因为下面重复写了很多由 `.c` 编译为 `.o` 的语句，如果这个工程包含很多这种操作，将会是十分繁杂且容易出错的，所以可以学习 **模式匹配** 做出改进！

## Makefile模式匹配

- `%.o:%.c`
  - `%`：通配符，匹配的是一个字符串
  - 两个`%`匹配的是同一个字符串

例子：

```
%.o:%.c
    gcc -c $< -o $@
```

所以使用模式匹配可以优化为以下形式，十分抽象了：

```
#定义变量
src=add.o sub.o div.o mult.o main.o
target=main.a

$(target):$(src)
    $(CC) $(src) -o $(target)

%.o:%.c
    $(CC) -c $< -o $@
```

但是这个文件还有值得优化的地方：定义变量那里，如果src源文件里面有很多文件，我们需要一个个手动打上去就太麻烦了，而且容易出错，所以可以学习 **函数** 进行改进。

## Makefile函数

- \$(wildcard PATTREN...)
  - 参数：wildcard是文件名，PATTREN指的是某个目录或多个目录下对应的某种类型的文件，如果有多个目录，一般使用空空间隔
  - 功能：获取指定目录下指定类型的文件列表
  - 返回：得到的若干个文件的文件列表，文件名之间使用空空间隔
  - 示例：

```
$(wildcard *.c ./sub/*.c)
返回值格式：a.c b.c c.c d.c e.c
```

- \$(patsubst <pattern>, <replacement>, <text>)
  - 功能：查找<text>中的单词（单词以空格 Tab 回车 换行分隔）是否符合模式<pattern>，如果匹配，就用<replacement>替换
  - 返回：替换后的字符串

所以使用这个函数，可以把文件改进为以下形式：

```
#定义变量
#add.o sub.o div.o mult.o main.o
src=$(wildcard ./*.c)
objs=$(subst %.c, %.o, $(src))
target=main.a

$(target):$(objs)
    $(CC) $(objs) -o $(target)

%.o:%.c
    $(CC) -c $< -o $@
```

但是可以发现生成的文件中有很多.o文件，这些都是我们不需要的，如图：

```
eason@eason:~/Linux/Makefile$ make
cc -c mult.c -o mult.o
cc -c main.c -o main.o
cc -c add.c -o add.o
cc -c div.c -o div.o
cc -c sub.c -o sub.o
cc ./mult.o ./main.o ./add.o ./div.o ./sub.o -o main.a
eason@eason:~/Linux/Makefile$ ls
add.c  div.c  head.h  main.c  Makefile  mult.o      redis-5.0.10.tar.gz  sub.o
add.o  div.o  main.a  main.o  mult.c    redis-5.0.10  sub.c
eason@eason:~/Linux/Makefile$ ./main.a
a = 20, b = 12
a + b = 32
a - b = 8
a * b = 240
a / b = 1.666667
eason@eason:~/Linux/Makefile$
```

所以我们可以再次修改Makefile文件删除多余文件，如下图：

```
#定义变量
#add.o sub.o div.o mult.o main.o
src=$(wildcard ./*.c)
objs=$(subst %.c, %.o, $(src))
target=main.a

$(target):$(objs)
    $(CC) $(objs) -o $(target)

%.o:%.c
    $(CC) -c $< -o $@

clean:
    rm $(objs) -f
```

但是这个时候执行make是不会重新编译文件的，所以我们可以指定执行clean，通过输入make clean，如下图：

```
eason@eason:~/Linux/Makefile$ ls
add.c  div.c  head.h  main.c  Makefile  mult.o      redis-5.0.10.tar.gz  sub.o
add.o  div.o  main.a  main.o  mult.c    redis-5.0.10  sub.c
eason@eason:~/Linux/Makefile$ ./main.a
a = 20, b = 12
a + b = 32
a - b = 8
a * b = 240
a / b = 1.666667
eason@eason:~/Linux/Makefile$ vim Makefile
eason@eason:~/Linux/Makefile$ make
make: "main.a"已是最新。
eason@eason:~/Linux/Makefile$ make clean
make: *** 没有规则可制作目标“clean”。 停止。
eason@eason:~/Linux/Makefile$ vim Makefile
eason@eason:~/Linux/Makefile$ make clean
rm ./add.o ./mult.o ./main.o ./div.o ./sub.o -f
eason@eason:~/Linux/Makefile$ ls
add.c  div.c  head.h  main.a  main.c  Makefile  mult.c    redis-5.0.10  redis-5.0.10.tar.gz  sub.c
```

## GDB调试

### 准备工作

在调试之前，要执行以下指令，把调试信息加入到文件中

```
gcc -g -Wall program.c -o program
```

### GDB命令

- 启动和退出

- gdb 可执行程序
  - quit
- 给程序设置参数及获取参数
  - set args ...
  - show args
- GDB使用帮助
  - help
- 查看当前文件代码
  - list/l 从默认位置显示
  - list/l 行号 从指定的行显示
  - list/l 函数名 从指定的函数显示
- 查看非当前文件代码
  - list/l 文件名: 行号
  - list/l 文件名: 函数名
- 设置显示的行数
  - show list/listsiz
  - set list/listsiz 行数
- 断点操作
  - 设置断点
    - b/break 行号
    - b/break 函数名
    - b/break 文件名: 行号
    - b/break 文件名: 函数
  - 查看断点
    - i/info b/break
  - 删除断点
    - d/del/delete 断点编号
  - 设置断点无效
    - dis/disable 断点编号
  - 设置断点生效
    - ena/enable 断点编号
  - 设置断点条件, 一般用在循环的位置
    - b/break 10 if i=5
- 调试命令
  - 运行GDB程序
    - start, 程序停在第一行
    - run, 遇到断点才停



- 继续运行，遇到下一个断点才停
  - c/continue
- 向下执行一行代码，不会进入函数体
  - n/next
- 变量操作
  - p/print 变量名，打印变量值
  - ptype 变量名，打印变量类型
- 向下单步调试，遇到函数会进入函数体
  - s/step
  - finish，跳出函数体
- 自动变量操作
  - display num，自动打印指定变量的值
  - i/info display
  - undisplay 编号
- 其它操作
  - set var 变量名=变量值
  - until，跳出循环

## 文件I/O

感觉这一章节讲的东西和底层的联系很多，I/O、虚拟地址空间等，结合计组、OS、Linux理解。

### 标准C库I/O函数和Linux系统I/O函数对比

主要讲了C库I/O的过程。

C库I/O函数有缓冲区，Linux系统I/O函数没有缓冲区，所以可以根据这个特性选择合适的I/O方式。

所以Linux的输出想要显示到屏幕上，需要使用\n等进行刷新，它不会像C一样代码运行结束自动刷新，如下：

```
#include<stdio.h>

int main(){

    printf("Linux has no flush, so the next line without endl will not be
output!\n");
    printf("nextlint!");

    return 0;
}
```

```
eason@eason:~/Linux/chapter1/IO$ gcc flush.c -o flush
eason@eason:~/Linux/chapter1/IO$ ./flush
Linux has no flush, so the next line without endl will not be output!
```

## 虚拟地址空间

## 文件描述符

## Linux系统I/O函数

这些东西可以查看官方文档：输入`man \space 2 \space XXX`（要查看的东西）；如果是C库则是`man \space 3 \space XXX`。

介绍了`open`、`errno`、`mode`、`read`、`write`、`lseek`、`stat`、`lstat`，可以边看官方文档边结合老师给的例子学习。

## 模拟实现ls-l指令

目的：教我们使用`st-mode`字段

同时也理解了Linux中`ls-l`指令的底层操作时怎么样的，可以结合`inode`和`stat`官方文档一起学习。

## 文件属性操作函数

- `access`：判断文件权限或文件是否存在
- `chmod`：修改文件权限
- `chown`：修改文件所有者
- `truncate`：缩减或拓展文件大小

## 目录操作函数

- `mkdir`
- `rmdir`
- `rename`
- `chdir`
- `getcwd`

## 目录遍历函数

- `opendir`
- `readdir`
- `closedir`

## dup和dup2函数

- `dup`：复制文件描述符，两个文件描述符指向的是同一个文件，相当于浅拷贝
- `dup2`：重定向文件描述符

## fcntl函数

`fcntl`在Linux中可以做五件事情，但是课程中只需掌握其中两件，应该还是比较重要和常见的：

- 复制文件描述符
- 设置/获取文件的状态标志