更好的阅读体验：https://blog.dwj601.cn/Operation/Shell/shell-basic/

实验1 Shell 基本命令

在开始之前，我们可以使用 `echo $SHELL` 命令查看当前的命令解释器是什么：

```
root@dwj2:~# echo $SHELL
/bin/bash
```

1.1 常用命令

改变目录 cd

```
cd ../
```

`../` 表示上一级，`./` 表示当前一级（也可以不写），`/` 表示从根目录开始。

打印目录内容 ls

```
ls
```

- `-l` 参数即 long listing format，表示打印详细信息。
- `-h` 参数即 human-readable，会使得结果更加可读，例如占用存储空间加上单位等等。

打印当前路径 pwd

```
pwd
```

打印当前用户名

```
whoami
```

创建文件夹 mkdir

```
mkdir <FolderName>
```

创建文件 touch

```
touch <FileName>
```

复制 cp

```
cp [option] <source> <target>
```

- `-r` 表示递归复制，`-i` 用来当出现重名文件时进行提示。
- source 表示被拷贝的资源。
- target 表示拷贝后的资源名称或者路径。

移动 mv

```
mv [option] <source> <target>
```

- `-i` 用来当出现重名文件时进行提示。
- source 表示被移动的资源。
- target 表示移动后的资源名称或者路径（可以以此进行重命名）。

删除 rm

```
rm [option] <source>
```

- `-i` 需要一个一个确认，`-f` 表示强制删除，`-r` 表示递归删除。

打印 echo

```
echo "hello"
```

- 将 echo 后面紧跟着的内容打印到命令行解释器中。可以用来查看环境变量的具体值。
- 也可以配合输出重定向符号 `>` 将信息打印到文件中实现创建新文件的作用。例如 `echo "hello" > file.txt` 用于创建 file.txt 文件并且在其中写下 `hello` 信息。

查看 cat

```
cat [option] <source>
```

- `-n` 或 `--number` 表示将 source 中的内容打印出来的同时带上行号。

- 也可以配合输出重定位符号 `>` 将信息输出到文件中。例如 `cat -n a.txt > b.txt` 表示将 a.txt 文件中的内容带上行号输出到 b.txt 文件中。

分页查看 more

```
more <source>
```

与 `cat` 类似，只不过可以分页展示。按空格键下一页，`b` 键上一页，`q` 键退出。

可以配合管道符 `|` (将左边的输出作为右边的输入) 与别的命令组合从而实现分页展示，例如 `tree | more` 表示分页打印文件目录。

输出重定向

标准输出（stdout）默认是显示器。

`>` 表示创建或覆盖，`>>` 表示追加。

1.2 重要命令

查找 find *

```
find <path> <expression>
```

- `-maxdepth <num>`, `-mindepth <num>`: 最大、最小搜索深度。

匹配 grep *

```
grep [option] <pattern> <source>
```

使用正则表达式在指定文件中进行模式匹配。

- `-n` 显示行号，`-i` 忽略大小写，`-r` 递归搜索，`-c` 打印匹配数量。

实验2 用户与权限管理

在 Windows 中，我们对用户和权限的概念接触的并不多，因为很多东西都默认设置好了。但是在 GNU/Linux 中，很多文件的权限都需要自己配置和定义，因此「用户与权限管理」的操作方法十分重要。我们从现象入手逐个进行讲解。

首先以 root 用户身份登录并进入 `/opt/OS/task2/` 目录，然后创建一个测试文件 `root_file.txt` 和一个测试文件夹 `root_folder`。使用 `ls -l` 命令列出当前目录下所有文件的详细信息：

```
root@dwj2:/opt/OS/task2#
root@dwj2:/opt/OS/task2#
root@dwj2:/opt/OS/task2#
root@dwj2:/opt/OS/task2# ls -l
总计 8
-rw-r--r-- 1 root root   18  9月 27 09:58 root_file.txt
drwxr-xr-x 2 root root 4096  9月 27 09:58 root_folder
root@dwj2:/opt/OS/task2#
root@dwj2:/opt/OS/task2#
root@dwj2:/opt/OS/task2#
```

可以看到一共有 6 列信息，从左到右依次为：用户访问权限、与文件链接的个数、文件属主、文件属组、文件大小、最后修改日期、文件/目录名。$2-5$ 列的信息都很显然，第 1 列信息一共有 10 个字符，其中第 1 个字符表示当前文件的类型，共有如下几种：

文件类型	符号
普通文件	`-`
目录	`d`
字符设备文件	`c`
块设备文件	`b`
符号链接	`l`
本地域套接口	`s`
有名管道	`p`

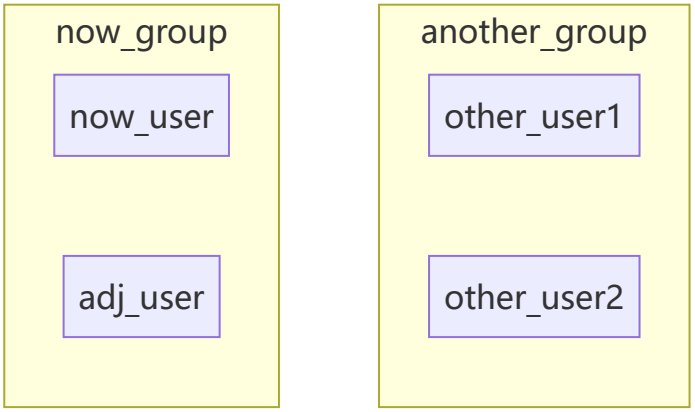
第 $2-10$ 共 9 个字符分 3 个一组，分别表示「属主 user 权限、属组 group 权限、其他用户 other 权限」，下面分别介绍「用户和组」以及「权限管理」两个概念。

2.1 关于用户和组

首先我们需要明确用户和组这两个概念的定义：

- **文件用户（user/u）**：文件的拥有者。
- **所属组（group/g）**：与该文件关联的用户组，组内成员享有特定的权限。
- **其他用户（others/o）**：系统中不属于拥有者或组的其他用户。

举个例子。对于当前用户 `now_user` 以及当前用户所在的组 `now_group`，同组用户 `adj_user` 和其他用户 `other_user` 可以形象的理解为以下的集合关系：



2.2 关于权限管理

一共有 3 种权限，如下表所示：

	文件	目录
`r`	可查看文件	能列出目录下内容
`w`	可修改文件	能在目录中创建、删除和重命名文件
`x`	可执行文件	能进入目录

那么我们平时看到的关于权限还有数字的配置，是怎么回事呢？其实是对上述字符配置的八进制数字化。读 r 对应 4，写 w 对应 2，可执行 x 对应 1，例如如果一个文件对于所有用户都拥有可读、可写、可执行权限，那么就是 `rwxrwxrwx`，对应到数字就是 777。

2.3 相关命令

下面罗列一些用户与权限管理相关的命令，打 $*$ 表示重要命令。

提升权限 sudo

```
sudo ...
```

使用 sudo 前缀可以使得当前用户的权限提升到 root 权限，如果是 root 用户则无需添加。

查看当前用户 whoami

```
whoami
```

打印当前用户名。

创建用户 useradd

```
useradd <username>
```

添加用户。

删除用户 userdel

```
userdel <username>
```

删除用户。

- `-r` 表示同时删除数据信息。

修改用户信息 usermod

```
usermod
```

- 使用 `-h` 参数查看所有用法。

修改用户密码 passwd

```
passwd <username>
```

切换用户 su

```
su <username>
```

添加 `-` 参数则直接进入 `/home/<username>/` 目录（如果有的话）。

查看当前用户所属组 groups

```
groups
```

创建用户组 groupadd

```
groupadd
```

删除用户组 groupdel

```
groupdel
```

改变属主 chown *

```
chown <user> <filename>
```

将指定文件 filename 更改属主为 user。

改变属组 chgrp *

```
chgrp <group> <filename>
```

将指定文件 filename 更改属组为 group。

改变权限 chmod *

```
chmod <option> <filename>
```

将文件 filename 更改所有用户对应的权限。举个例子就知道了：让 `demo.py` 文件只能让所有者拥有可读、可写和可执行权限，其余任何用户都只有可读和可写权限。

```
# 写法一
chmod u=rwx,go=rw demo.py

# 写法二
chmod 766 demo.py
```

至于为什么数字表示法会用 $4, 2, 1$，是因为 $4, 2, 1$ 刚好对应了二进制的 $001, 010, 100$，三者的组合可以完美的表示出 $[0, 7]$ 范围内的任何一个数。

默认权限 umask

```
umask
```

- `-S` 显示字符型默认权限

直接使用 `umask` 会显示 4 位八进制数，第一位是当前用户的 uid，后三位分别表示当前用户创建文件时的默认权限的补，例如 0022 表示当前用户 uid 为 0，创建的文件/目录默认权限为 $777-022=755$。

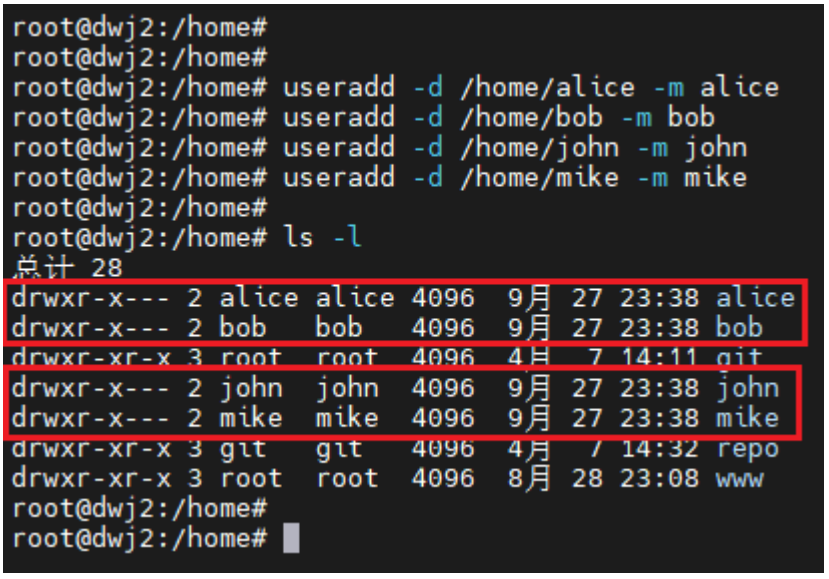
可能是出于安全考虑，文件默认不允许拥有可执行权限，因此如果 `umask` 显示为 0022，则创建的文件默认权限为 644，即每一位都 -1 以确保是偶数。

练习

一、添加 4 个用户：alice、bob、john、mike

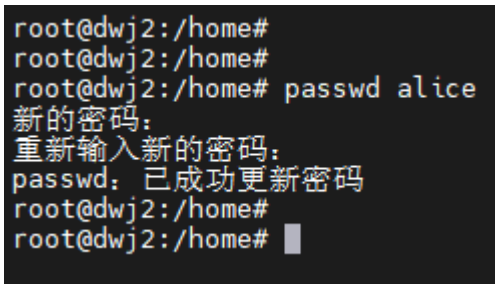
首先需要确保当前是 root 用户，使用 `su root` 切换到 root 用户。然后在创建用户时同时创建该用户对应的目录：

```
useradd -d /home/alice -m alice
useradd -d /home/bob -m bob
useradd -d /home/john -m john
useradd -d /home/mike -m mike
```



二、为 alice 设置密码

```
passwd alice
```



三、创建用户组 workgroup 并将 alice、bob、john 加入

- 创建用户组：

```
groupadd workgroup
```

- 添加到新组：

```
usermod -a -G workgroup alice
usermod -a -G workgroup bob
usermod -a -G workgroup john
```

  - `-a`：是 `--append` 的缩写，表示将用户添加到一个组，而不会移除她已有的其他组。这个选项必须与 `-G` 一起使用
  - `-G`：指定要添加用户的附加组（即用户可以属于多个组），这里是 workgroup

- 将 workgroup 作为各自的主组：

```
usermod -g workgroup alice
usermod -g workgroup bob
usermod -g workgroup john
```

  - `-g`：用于指定用户的主组（primary group）。主组是当用户创建文件或目录时默认分配的组

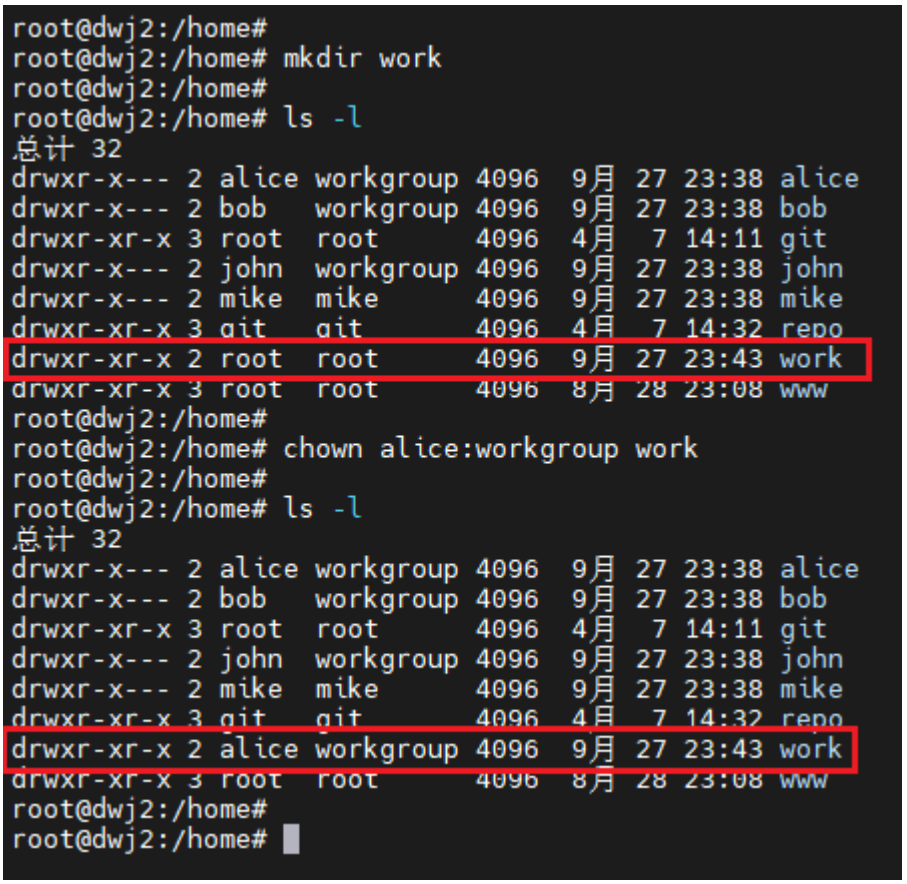
```
root@dwj2:/home#
root@dwj2:/home#
root@dwj2:/home# groupadd workgroup
root@dwj2:/home#
root@dwj2:/home# usermod -a -G workgroup alice
root@dwj2:/home# usermod -a -G workgroup bob
root@dwj2:/home# usermod -a -G workgroup john
root@dwj2:/home#
root@dwj2:/home# usermod -g workgroup alice
root@dwj2:/home# usermod -g workgroup bob
root@dwj2:/home# usermod -g workgroup john
root@dwj2:/home#
root@dwj2:/home# ls -l
总计 28
drwxr-x--- 2 alice workgroup 4096  9月 27 23:38 alice
drwxr-x--- 2 bob   workgroup 4096  9月 27 23:38 bob
drwxr-xr-x 3 root  root      4096  4月  7 14:11 git
drwxr-x--- 2 john  workgroup 4096  9月 27 23:38 john
drwxr-x--- 2 mike  mike      4096  9月 27 23:38 mike
drwxr-xr-x 3 git   git       4096  4月  7 14:32 repo
drwxr-xr-x 3 root  root      4096  8月 28 23:08 www
root@dwj2:/home#
root@dwj2:/home#
```

四、创建 `/home/work` 目录并将其属主改为 alice，属组改为 workgroup

```shell
# 创建目录
mkdir work

# 修改属主和属组
chown alice:workgroup work

# 或者
chown alice.workgroup work
```

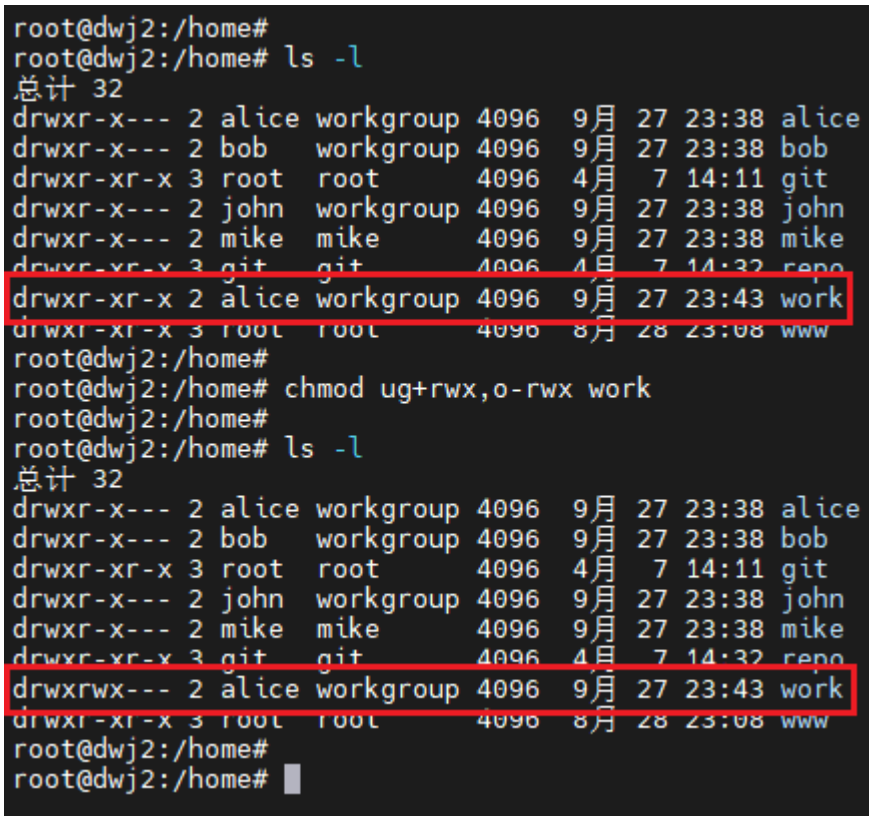


五、修改 work 目录的权限

使得属组内的用户对该目录具有所有权限，属组外的用户对该目录没有任何权限。

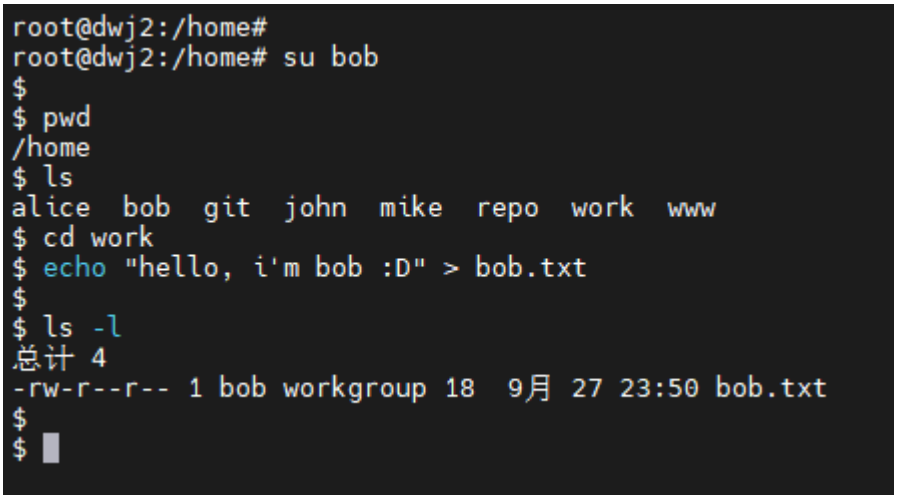
```
# 写法一
chmod ug+rwx,o-rwx work

# 写法二
chmod 770 work
```



六、权限功能测试

以 bob 用户身份在 work 目录下创建 `bob.txt` 文件。可以看到符合默认创建文件的权限格式 644 ：



同组用户与不同组用户关于「目录/文件」的 `rw` 权限测试。

- 关于 770 目录。由于 work 目录被 bob 创建时权限设置为了 770，bob 用户与 john 用户属于同一个组 workgroup，因此 john 因为 $g = 7$ 可以进入 work 目录进行操作，而 bob 用户与 mike 用户不属于同一个组，因此 mike 因为 $o = 0$ 无法进入 work 目录，更不用说查看或者修改 work 目录中的文件了。
- 关于 644 文件。现在 john 由于 770 中的第二个 7 进入了 work 目录。由文件默认的 644 权限可以知道：john 因为第一个 4 可以读文件，但是不可以写文件，因此如下图所示，可以执行 `cat` 查看文件内容，但是不可以执行 `echo` 编辑文件内容。至于 mike，可以看到无论起始是否在 work 目录，都没有权限 `cd` 到 work 目录或者 `ls` 查看 work 目录中的内容。

```
root@dwj2:/home# su john
$ pwd
/home
$ cd work
$ ls -l
总计 4
-rw-r--r-- 1 bob workgroup 18  9月 27 23:50 bob.txt
$
$ cat bob.txt
hello, i'm bob :D
$
$ echo "hi i'm john" >> bob.txt
sh: 7: cannot create bob.txt: Permission denied
$
root@dwj2:/home# su mike
$ pwd
/home
$ cd work
sh: 2: cd: can't cd to work
$
$ cat work/bob.txt
cat: work/bob.txt: 权限不够
$
$
root@dwj2:/home# cd work
root@dwj2:/home/work#
root@dwj2:/home/work# su mike
$ pwd
/home/work
$
$ ls -l
ls: 无法打开目录 '.': 权限不够
$ cat bob.txt
cat: bob.txt: 权限不够
$
$
```

实验3 进程管理与调试

在熟悉了 bash shell 的基本命令以及 GNU/Linux 中用户与权限管理的基本概念后，我们就可以开始尝试管理 GNU/Linux 中的程序了。当然每一个程序在开始运行后都会成为一个或多个进程，因此接下来简单介绍一下 GNU/Linux 的进程管理。最后再通过调试一个 C 程序来熟悉 GNU 调试工具 gdb (GNU Debugger) 的使用。

3.1 进程管理相关命令

查看进程状态 ps

```
1  ps
```

动态查看进程状态 top

```
1  top
```

杀死某个进程 kill

```
kill -9 <PID>
```

练习

一、编写一个 shell 程序 `badproc.sh` 使其不断循环

```
#! /bin/bash
while echo "I'm making files!"
do
    mkdir adir
    cd adir
    touch afile
    sleep 10s
done
```

```
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3# ll
总计 8
drwxr-xr-x 2 root root 4096 10月  8 17:30 ./
drwxr-xr-x 5 root root 4096 10月  8 17:30 ../
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3# touch badproc.sh
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3# ll
总计 8
drwxr-xr-x 2 root root 4096 10月  8 17:34 ./
drwxr-xr-x 5 root root 4096 10月  8 17:30 ../
-rw-r--r-- 1 root root    0 10月  8 17:34 badproc.sh
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3# vim badproc.sh
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3# cat badproc.sh
#! /bin/bash
while echo "I'm making files!"
do
    mkdir adir
    cd adir
    touch afile
    sleep 10s
done
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3#
```

二、为 `badproc.sh` 增加可执行权限

```
1  chmod u+x badproc.sh
```

```
root@dwj2:/opt/OS/task3# ll
总计 12
drwxr-xr-x 2 root root 4096 10月  8 17:34 ./
drwxr-xr-x 5 root root 4096 10月  8 17:30 ../
-rw-r--r-- 1 root root  109 10月  8 17:34 badproc.sh
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3# chmod u+x badproc.sh
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3# ll
总计 12
drwxr-xr-x 2 root root 4096 10月  8 17:34 ./
drwxr-xr-x 5 root root 4096 10月  8 17:30 ../
-rwxr--r-- 1 root root  109 10月  8 17:34 badproc.sh*
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3#
```

三、在后台执行 `badproc.sh`

```
./badproc.sh &
```

- `&` 表示后台执行

```
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3# ll
总计 12
drwxr-xr-x 2 root root 4096 10月  8 17:34 ./
drwxr-xr-x 5 root root 4096 10月  8 17:30 ../
-rwxr--r-- 1 root root  109 10月  8 17:34 badproc.sh*
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3# ./badproc.sh &
[1] 1488084
root@dwj2:/opt/OS/task3# I'm making files!
I'm making files!
I'm making files!
I'm making files!
I'm making files!
```

四、利用 `ps` 命令查看其进程号

```
ps aux | grep badproc
```

```
root@dwj2:/opt/OS/task3# tree
.
├── adir
│   ├── adir
│   │   ├── adir
│   │   │   ├── adir
│   │   │   │   ├── adir
│   │   │   │   │   ├── adir
│   │   │   │   │   │   ├── adir
│   │   │   │   │   │   │   ├── adir
│   │   │   │   │   │   │   │   ├── adir
│   │   │   │   │   │   │   │   │   ├── adir
│   │   │   │   │   │   │   │   │   │   ├── adir
│   │   │   │   │   │   │   │   │   │   │   └── afile
│   │   │   │   │   │   │   │   │   │   └── afile
│   │   │   │   │   │   │   │   │   └── afile
│   │   │   │   │   │   │   │   └── afile
│   │   │   │   │   │   │   └── afile
│   │   │   │   │   │   └── afile
│   │   │   │   │   └── afile
│   │   │   │   └── afile
│   │   │   └── afile
│   │   └── afile
│   └── afile
└── badproc.sh

14 directories, 15 files
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3# ps aux | grep badproc
root     1488084  0.0  0.2  10592  4008 pts/0    S    17:36   0:00 /bin/bash ./badproc.sh
root     1488966  0.0  0.1   9588  2240 pts/0    S+   17:39   0:00 grep --color=auto badproc
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3# I'm making files!
```

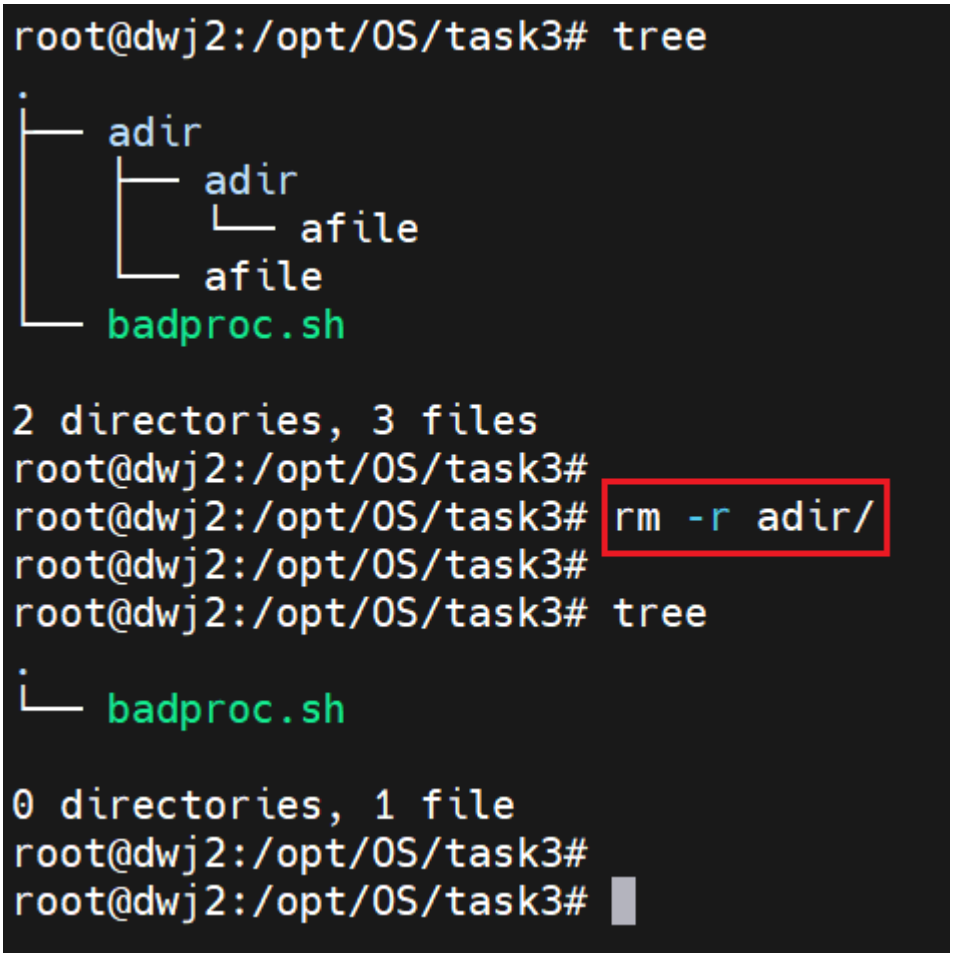
五、利用 `kill` 命令杀死该进程

```
1  kill -9 <PID>
```

```
root@dwj2:/opt/OS/task3# ll
总计 12
drwxr-xr-x 2 root root 4096 10月  8 17:46 ./
drwxr-xr-x 5 root root 4096 10月  8 17:30 ../
-rwxr--r-- 1 root root  109 10月  8 17:44 badproc.sh*
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3# ./badproc.sh &
[1] 1491819
root@dwj2:/opt/OS/task3# I'm making files!
I'm making files!

root@dwj2:/opt/OS/task3# ps aux | grep badproc
root     1491819  0.0  0.2  10592  3912 pts/0    S    17:46   0:00 /bin/bash ./badproc.sh
root     1491977  0.0  0.1   9588  2264 pts/0    S+   17:47   0:00 grep --color=auto badproc
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3# kill -9 1491819
root@dwj2:/opt/OS/task3#
[1]+  已杀死               ./badproc.sh
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3#
```

六、删除 `badproc.sh` 程序运行时创建的目录和文件



3.2 gdb 调试相关命令

基本命令参考 GNU OF GDB 官网：https://www.gnu.org/software/gdb/。常用的如下：

开始运行

```
1  r
```

- r 即 run

设置断点

```
1  break <line>
```

- line 即行号

运行到下一个断点

```
1  c
```

- c 即 continue

练习

一、创建 `fork.c` 文件

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    /* fork another process */
    pid_t  pid;
    pid = fork();

    if (pid < 0) {
        /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    } else if (pid == 0) {
        /* child process */
        printf("This is child process, pid=%d\n", getpid());
        execlp("/bin/ls", "ls", NULL);
        printf("Child process finished\n"); /*这句话不会被打印，除非execlp调用未成功*/
    } else {
        /* parent process */
        /* parent will wait for the child to complete */
        printf("This is parent process, pid=%d\n", getpid());
        wait (NULL);
        printf ("Child Complete\n");
        exit(0);
    }
}
```

```
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3# cat fork.c -n
     1  #include <stdio.h>
     2  #include <stdlib.h>
     3  #include <unistd.h>
     4  #include <sys/types.h>
     5  #include <sys/wait.h>
     6
     7  int main() {
     8          /* fork another process */
     9          pid_t  pid;
    10          pid = fork();
    11
    12          if (pid < 0) {
    13                  /* error occurred */
    14                  fprintf(stderr, "Fork Failed");
    15                  exit(-1);
    16          } else if (pid == 0) {
    17                  /* child process */
    18                  printf("This is child process, pid=%d\n", getpid());
    19                  execlp("/bin/ls", "ls", NULL);
    20                  printf("Child process finished\n"); /*这句话不会被打印，除非execlp调用未成功*/
    21          } else {
    22                  /* parent process */
    23                  /* parent will wait for the child to complete */
    24                  printf("This is parent process, pid=%d\n", getpid());
    25                  wait (NULL);
    26                  printf ("Child Complete\n");
    27                  exit(0);
    28          }
    29  }
    30
root@dwj2:/opt/OS/task3#
```

这段程序首先通过调用 `fork()` 函数创建一个子进程，并通过 `pid` 信息来判断当前进程是父进程还是子进程。在并发的逻辑下，执行哪一个进程的逻辑是未知的。

二、编译运行 `fork.c` 文件

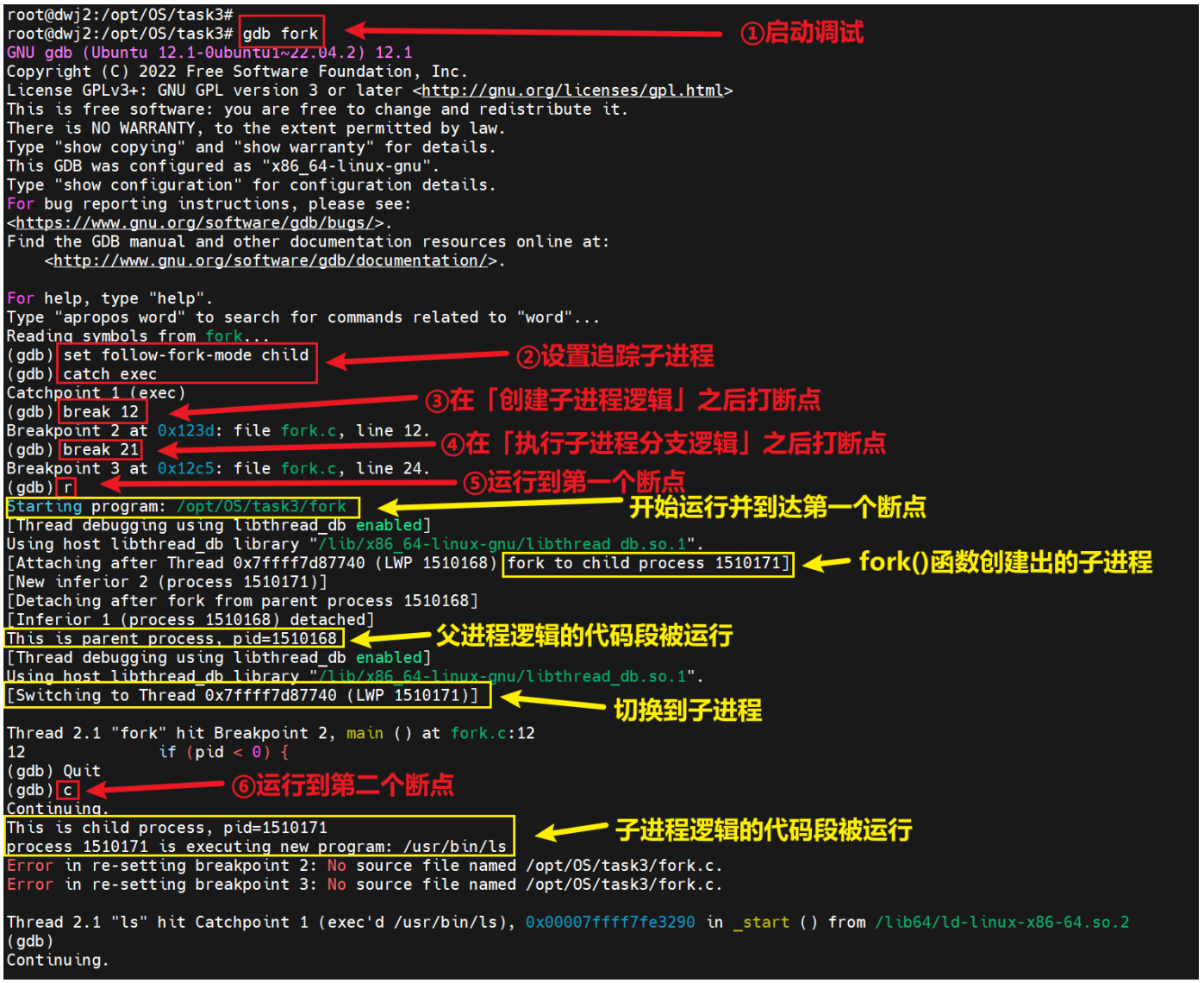
```
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3# gcc -g -o fork fork.c
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3# ll
总计 36
drwxr-xr-x 2 root root  4096 10月  8 18:06 ./
drwxr-xr-x 5 root root  4096 10月  8 17:30 ../
-rwxr--r-- 1 root root   109 10月  8 17:44 badproc.sh*
-rwxr-xr-x 1 root root 19040 10月  8 18:06 fork*
-rw-r--r-- 1 root root   693 10月  8 17:55 fork.c
root@dwj2:/opt/OS/task3#
root@dwj2:/opt/OS/task3# ./fork
This is parent process, pid=1498960
This is child process, pid=1498961
badproc.sh  fork  fork.c
Child Completeroot@dwj2:/opt/OS/task3#
```

从上述运行结果可以看出：并发时，首先执行父进程的逻辑，然后才执行子进程的逻辑。

三、gdb 调试

在 fork 创建子进程后追踪子进程：

```
gdb fork
set follow-fork-mode child
catch exec
```



运行到第一个断点时分别观察父进程 1510168 和子进程 1510171：

```
root@dwj2:~# pmap 1510168
1510168:   /opt/OS/task3/fork
000055555554000      4K r---- fork
000055555555000      4K r-x-- fork
000055555556000      4K r---- fork
000055555557000      4K r---- fork
000055555558000      4K rw--- fork
000055555559000    132K rw---   [ anon ]
00007ffff7d87000     12K rw---   [ anon ]
00007ffff7d8a000    160K r---- libc.so.6
00007ffff7db2000   1620K r-x-- libc.so.6
00007ffff7f47000    352K r---- libc.so.6
00007ffff7f9f000      4K ----- libc.so.6
00007ffff7fa0000     16K r---- libc.so.6
00007ffff7fa4000      8K rw--- libc.so.6
00007ffff7fa6000     52K rw---   [ anon ]
00007ffff7fbb000      8K rw---   [ anon ]
00007ffff7fbd000     16K r----   [ anon ]
00007ffff7fc1000      8K r-x--   [ anon ]
00007ffff7fc3000      8K r---- ld-linux-x86-64.so.2
00007ffff7fc5000    168K r-x-- ld-linux-x86-64.so.2
00007ffff7fef000     44K r---- ld-linux-x86-64.so.2
00007ffff7ffb000      8K r---- ld-linux-x86-64.so.2
00007ffff7ffd000      8K rw--- ld-linux-x86-64.so.2
00007ffffffde000    132K rw---   [ stack ]
ffffffffff600000      4K --x--   [ anon ]
 total             2780K
root@dwj2:~#
```

```
root@dwj2:~# pmap 1510171
1510171:   /opt/OS/task3/fork
0000555555554000      4K r---- fork
0000555555555000      4K r-x-- fork
0000555555556000      4K r---- fork
0000555555557000      4K r---- fork
0000555555558000      4K rw--- fork
00007ffff7d87000     12K rw---   [ anon ]
00007ffff7d8a000    160K r---- libc.so.6
00007ffff7db2000   1620K r-x-- libc.so.6
00007ffff7f47000    352K r---- libc.so.6
00007ffff7f9f000      4K ----- libc.so.6
00007ffff7fa0000     16K r---- libc.so.6
00007ffff7fa4000      8K rw--- libc.so.6
00007ffff7fa6000     52K rw---   [ anon ]
00007ffff7fbb000      8K rw---   [ anon ]
00007ffff7fbd000     16K r----   [ anon ]
00007ffff7fc1000      8K r-x--   [ anon ]
00007ffff7fc3000      8K r---- ld-linux-x86-64.so.2
00007ffff7fc5000    168K r-x-- ld-linux-x86-64.so.2
00007ffff7fef000     44K r---- ld-linux-x86-64.so.2
00007ffff7ffb000      8K r---- ld-linux-x86-64.so.2
00007ffff7ffd000      8K rw--- ld-linux-x86-64.so.2
00007ffffffde000    132K rw---   [ stack ]
ffffffffff600000      4K --x--   [ anon ]
 total             2648K
```

运行到第二个断点时观察子进程 1510171：

```
root@dwj2:~# pmap 1510171
1510171:   ls
0000555555554000     16K r---- ls
0000555555558000     80K r-x-- ls
000055555556c000     32K r---- ls
0000555555574000     12K rw--- ls
0000555555577000      4K rw---   [ anon ]
00007ffff7fbd000     16K r----   [ anon ]
00007ffff7fc1000      8K r-x--   [ anon ]
00007ffff7fc3000      8K r---- ld-linux-x86-64.so.2
00007ffff7fc5000    168K r-x-- ld-linux-x86-64.so.2
00007ffff7fef000     44K r---- ld-linux-x86-64.so.2
00007ffff7ffb000     16K rw--- ld-linux-x86-64.so.2
00007ffffffde000    132K rw---   [ stack ]
ffffffffff600000      4K --x--   [ anon ]
 total              540K
root@dwj2:~#
```

从上述子进程的追踪结果可以看出，在父进程结束之后，子进程成功执行了 `pid == 0` 的逻辑并开始调用 `ls` 工具。

实验4 GNU/Linux 编程

这里的 GNU/Linux 编程对应的高级语言是 C/C++ 编程语言。本部分主要是为了熟悉 C/C++ 编程中的 **静态链接** 与 **动态链接** 逻辑。

4.1 GCC 基础

相比于在 Windows 进行 C/C++ 编程时需要自己额外安装编译器集合 MSVC (Microsoft Visual C++) 或 MinGW (Minimalist GNU for Windows)，GNU/Linux 发行版 Ubuntu22.04 已经默认配置好了编译器集合 GCC (GNU Compiler Collection)，我们可以利用 GCC 提供的前端工具 gcc 等快捷地使用编译器集合中的所有工具。具体命令可以参考 GCC 官方在线文档：https://gcc.gnu.org/onlinedocs/。

我们可以使用 `gcc --version` 命令查看当前的 GCC 版本：

```
root@dwj2:/opt/OS/task4# gcc --version
gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

因此我们选择版本最相近的手册 gcc-11.5.0 进行阅读。对于最基本的编译操作和理论，已经在 计算机系统基础 课程中有所学习，就不重复造轮子了，常见的指令直接跳转阅读即可。

环境变量

对于当前路径下链接出来的可执行文件 demo，为什么 `demo` 无法正常执行，`./demo` 就可以正常执行？如下图所示：

```
root@dwj2:/opt/OS/task4# ll
总计 28
drwxr-xr-x 2 root root  4096 10月  9 17:28 ./
drwxr-xr-x 6 root root  4096 10月  9 10:47 ../
-rwxr-xr-x 1 root root 15960 10月  9 17:28 demo*
-rw-r--r-- 1 root root    65 10月  9 17:28 demo.c
root@dwj2:/opt/OS/task4#
root@dwj2:/opt/OS/task4# cat demo.c
#include <stdio.h>
int main() {
        printf("hello\n");
        return 0;
}
root@dwj2:/opt/OS/task4#
root@dwj2:/opt/OS/task4# demo
找不到命令 “demo”，您的意思是：
  “deno” 命令来自 Snap 软件包 deno (v1.44.1)
  “nemo” 命令来自 Debian 软件包 nemo (5.2.4-1)
  “memo” 命令来自 Debian 软件包 memo (1.7.1-4)
  “idemo” 命令来自 Debian 软件包 ivtools-bin (2.0.11d.a1-1build1)
输入 “snap info <snapname>” 以查看更多版本。
root@dwj2:/opt/OS/task4#
root@dwj2:/opt/OS/task4# ./demo
hello
root@dwj2:/opt/OS/task4#
root@dwj2:/opt/OS/task4#
```

根本原因是 bash 默认执行 PATH 环境变量下的可执行文件，显然上述的 demo 可执行文件并不在 PATH 对应的路径下，那么 PATH 路径都有哪些呢？我们使用 `echo $PATH | tr ':' '\n'` 打印出来：

```
root@dwj2:/opt/OS/task4# echo $PATH | tr ':' '\n'
/usr/local/sbin
/usr/local/bin
/usr/sbin
/usr/bin
/usr/games
/usr/local/games
/snap/bin
```

能不能使用 demo 运行呢？有很多解决办法，但根本逻辑都是将当前路径加入到 PATH 环境变量。下面补充几个 gcc 和 g++ (编译 C++ 用) 相关的环境变量：

- 头文件搜索路径
  - `C_INCLUDE_PATH`: gcc 找头文件的路径。
  - `CPLUS_INCLUDE_PATH`: g++ 找头文件的路径。

- 库文件搜索路径
  - `LD_LIBRARY_PATH`：找动态链接库的路径。
  - `LIBRARY_PATH`：找静态链接库的路径。

gcc 选项

在计算机系统基础中已经学习到了，C/C++ 最基本的编译链就是 `-E`、`-S`、`-c`、`-o`，每一个参数都包含前面所有的参数。下面主要讲讲 `-I<dir>`，`-L<dir>` 和 `-l<name>` 三个参数：

- `-I<dir>` 顾名思义就是「头文件导入」的搜索目录。例如下面的编译语句：

  ```
  1  gcc -I/opt/OS/task4/include demo.c
  ```

  注意：当我们不使用 -o 参数指定 outfile 的名称时，默认是 `a.out`，如下图所示，其中 demo 可执行文件是之前用来输出 `'hello'` 的：

```
root@dwj2:/opt/OS/task4# vim ./include/add.h
root@dwj2:/opt/OS/task4#
root@dwj2:/opt/OS/task4# cat demo.c -n
     1  #include <stdio.h>
     2  #include <add.h>
     3  int main() {
     4          printf("1 + 2 = %d\n", add(1, 2));
     5          return 0;
     6  }
root@dwj2:/opt/OS/task4#
root@dwj2:/opt/OS/task4# cat ./include/add.h -n
     1  int add(int x, int y) {
     2          return x + y;
     3  }
root@dwj2:/opt/OS/task4#
root@dwj2:/opt/OS/task4# gcc demo.c
demo.c:2:10: fatal error: add.h: 没有那个文件或目录
    2 | #include <add.h>
      |          ^~~~~~~
compilation terminated.
root@dwj2:/opt/OS/task4#
root@dwj2:/opt/OS/task4# gcc -I/opt/OS/task4/include demo.c
root@dwj2:/opt/OS/task4#
root@dwj2:/opt/OS/task4# ./demo
hello
root@dwj2:/opt/OS/task4#
root@dwj2:/opt/OS/task4# ./a.out
1 + 2 = 3
root@dwj2:/opt/OS/task4#
root@dwj2:/opt/OS/task4#
```

- `-L<dir>` 顾名思义就是「库文件连接」搜索目录。例如下面的编译语句：

```
gcc -o x11fred -L/usr/openwin/lib x11fred.c
```

- `-l<name>` 比较有意思，就是直接制定了库文件是哪一个。正因为有了这样的用法，我们在给库文件（`.a` 表示静态库文件，`.so` 表示动态库文件）起名时，就只能起 `lib<name>.a` 或 `lib<name>.so`。例如下面的编译语句：

```
gcc -o fred -lm fred.c
# 等价于
gcc -o fred /usr/lib/libm.a fred.c
```

4.2 静态链接 | 动态链接

对于下面的函数库与调用示例：

```
// addvec.c
void addvec(int* x, int* y, int* z, int n) {
    for(int i = 0; i < n ; i++) {
        z[i] = x[i] + y[i];
    }
}

// multvec.c
void multvec(int* x, int* y, int* z, int n) {
    for(int i = 0; i < n ; i++) {
        z[i] = x[i] * y[i];
    }
}

// vector.h
void addvec(int* x, int* y, int* z, int n);
void multvec(int* x, int* y, int* z, int n);

// main.c
#include <stdio.h>
#include "vector.h"
int x[2] = {1, 2}, y[2] = {3, 4}, z[2];
int main() {
    addvec(x, y, z, 2);
    printf("z = [%d, %d]\n", z[0], z[1]);
    return 0;
}
```

生成静态库文件 `libvector.a` 并链接至可执行文件 p1 中：

```
# 将两个自定义库函数编译为可重定位目标文件 addvec.o 和 multvec.o
gcc -c addvec.c multvec.c

# 将两个可重定位目标文件打包成静态库文件 libvector.a
ar crv libvector.a addvec.o multvec.o

# 生成静态链接的可执行文件 p1
gcc -static -o p1 main.c -L. -lvector
```

生成动态库文件 `libvector.so` 并链接至可执行文件 p2 中：

```
# 将两个自定义库函数编译为动态库文件 libvector.so
gcc -shared -o libvector.so addvec.c multvec.c

# 生成动态链接的可执行文件 p2
gcc -o p2 main.c -L. -lvector

# 使用 ./p2 执行之前需要明确一下动态库文件的链接搜索路径，否则会找不到动态库文件
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
```

最后我们查看一下 p1 和 p2 详细信息，如下图所示。显然静态链接的可执行文件 p1 占用的存储空间远大于动态连接的可执行文件 p2。

```
root@dwj2:/opt/OS/task4/TestLink# ./p1
z = [4, 6]
root@dwj2:/opt/OS/task4/TestLink# ./p2
z = [4, 6]
root@dwj2:/opt/OS/task4/TestLink# ls -lh --block-size=K
总计 940K
-rw-r--r-- 1 root root   1K 10月  9 19:46 addvec.c
-rw-r--r-- 1 root root   2K 10月  9 19:48 addvec.o
-rw-r--r-- 1 root root   3K 10月  9 19:48 libvector.a
-rwxr-xr-x 1 root root  15K 10月  9 19:48 libvector.so
-rw-r--r-- 1 root root   1K 10月  9 19:47 main.c
-rw-r--r-- 1 root root   1K 10月  9 19:46 multvec.c
-rw-r--r-- 1 root root   2K 10月  9 19:48 multvec.o
-rwxr-xr-x 1 root root 880K 10月  9 19:48 p1
-rwxr-xr-x 1 root root  16K 10月  9 19:48 p2
-rw-r--r-- 1 root root   1K 10月  9 19:47 vector.h
root@dwj2:/opt/OS/task4/TestLink#
```