Linux 系统编程入门

linux文件权限与目录配置

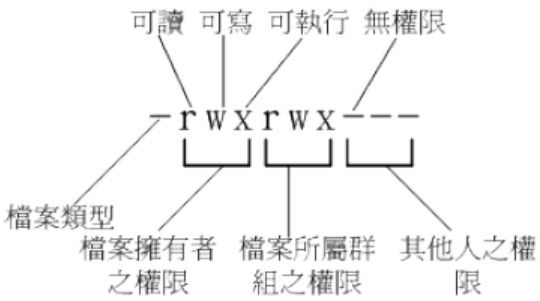


图 5.2.2、文件的类型与权限之内容

- 第一个字符代表这个文件是『目录、文件或链接文件等等』：
  - 当为[d]则是目录，例如上表档名为『.config』的那一行；
  - 当为[-]则是文件，例如上表档名为『initial-setup-ks.cfg』那一行；
  - 若是[l]则表示为连结档(link file)；
  - 若是[b]则表示为装置文件里面的可供储存的接口设备(可随机存取装置)；
  - 若是[c]则表示为装置文件里面的串行端口设备，例如键盘、鼠标(一次性读取装置)。

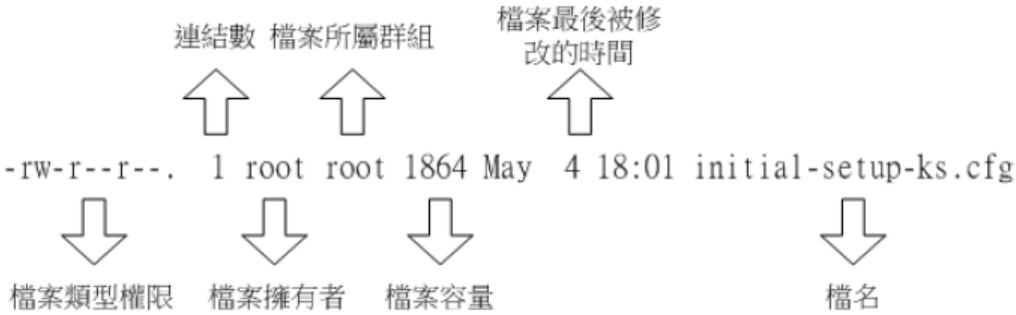


图 5.2.1、文件属性的示意图

> ls -all 可以查看文件的所有信息
>
> ls --help 或man ls 或info ls可以看到他的基础用法

群组、文件拥有者的理解

群组里面有很多文件的拥有者

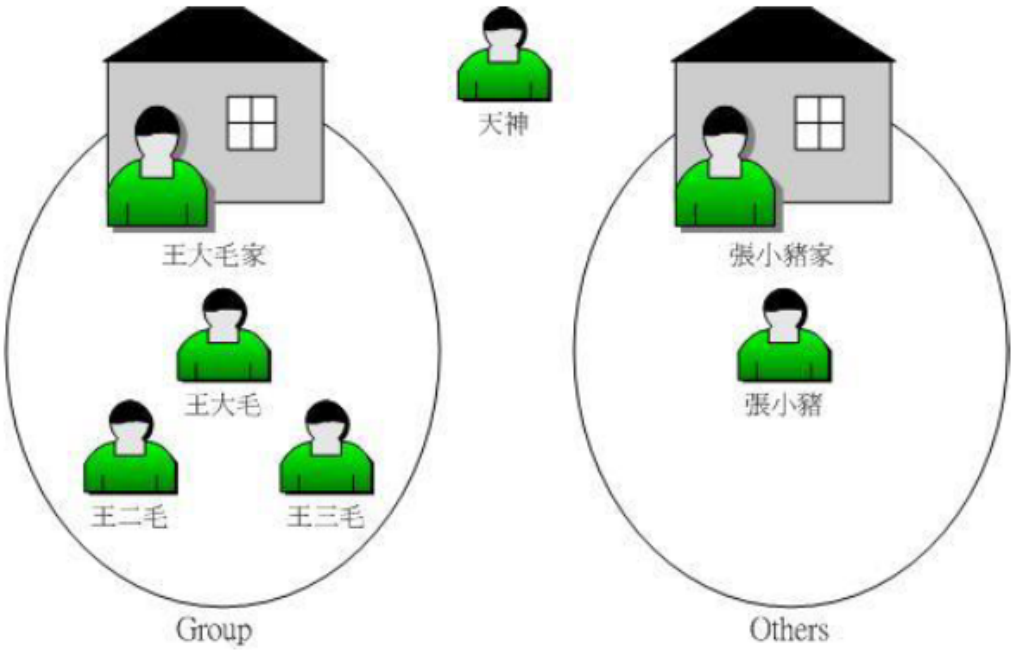


图 5.1.1、每个文件的拥有者、群组与 others 的示意图

改变文件权限与属性

这些修改是在根目录下进行的

在终端输入`su`,进入根目录，可是我现在这个Linux进不去根目录，或许是密码错误了

ls -al 显示文件的所有信息

- chgrp ：改变文件所属群组
- chown ：改变文件拥有者
- chmod ：改变文件的权限，SUID，SGID，SBIT 等等的特性

chgrp：改变群组

user是要改成的群组，必须是已经存在的群组

```
[root@study ~]# chgrp [-R] dirname/filename ...
选项与参数：
-R ：进行递归(recursive)的持续变更，亦即连同次目录下的所有文件、目录
     都更新成为这个群组之意。常常用在变更某一目录内所有的文件之情况。
范例：
[root@study ~]# chgrp users initial-setup-ks.cfg
[root@study ~]# ls -l
-rw-r--r--. 1 root users 1864 May  4 18:01 initial-setup-ks.cfg
[root@study ~]# chgrp testing initial-setup-ks.cfg
chgrp: invalid group:  `testing' <== 发生错误讯息啰～找不到这个群组名～
```

chown:改变所属者

```
[root@study ~]# chown [-R] 账号名称 文件或目录
[root@study ~]# chown [-R] 账号名称:组名 文件或目录
选项与参数：
-R ： 进行递归(recursive)的持续变更，亦即连同次目录下的所有文件都变更

范例：将 initial-setup-ks.cfg 的拥有者改为 bin 这个账号：
[root@study ~]# chown bin initial-setup-ks.cfg
[root@study ~]# ls -l
-rw-r--r--. 1 bin  users 1864 May  4 18:01 initial-setup-ks.cfg

范例：将 initial-setup-ks.cfg 的拥有者与群组改回为 root：
[root@study ~]# chown root:root initial-setup-ks.cfg
[root@study ~]# ls -l
-rw-r--r--. 1 root root 1864 May  4 18:01 initial-setup-ks.cfg
```

chmod：改变文件权限

- 数字类型改变文件权限

  Linux 文件的基本权限就有九个，分别是 owner/group/others 三种身份各有自己的 read/write/execute 权限，先复习一下刚刚上面提到的数据：文件的权限字符为：『-rwxrwxrwx』， 这九个权限是三个三个一组的！其中，我们可以使用数字来代表各个权限，各权限的分数对照表如下：

  r:4
  w:2
  x:1

  每种身份(owner/group/others)各自的三个权限(r/w/x)分数是需要累加的，例如当权限为： [-rwxrwx---] 分数则是：

  owner = rwx = 4+2+1 = 7
  group = rwx = 4+2+1 = 7
  others= --- = 0+0+0 = 0

```
[root@study ~]# ls -al .bashrc
-rw-r--r--. 1 root root 176 Dec 29  2013 .bashrc
[root@study ~]# chmod 777 .bashrc
[root@study ~]# ls -al .bashrc
-rwxrwxrwx. 1 root root 176 Dec 29  2013 .bashrc
```

权限对目录的重要性，上面说的是文件的权限

上面的东西这么说，也太条列式～太教条了～有没有清晰一点的说明啊？好～让我们来思考一下人类社会使用的东西好了！ 现在假设『文件是一堆文件文件夹』，所以你可能可以在上面写/改一些资料。而『目录是一堆抽屉』，因此你可以将文件夹分类放置到不同的抽屉去。 因此抽屉最大的目的是拿出/放入文件夹喔！现在让我们汇整一下数据：

组件	内容	迭代物件	r	w	x
文件	详细资料 data	文件文件夹	读到文件内容	修改文件内容	执行文件内容
目录	檔名	可分类抽屉	读到档名	修改檔名	进入该目录的权限(key)

对于目录来说，w是相当重要的权限，他可以让使用者删除、更新、新建文件或目录。x在目录中是与【能否进入该目录】有关，r是只能读到文件名，不能获取详细的目录信息

刚刚讲这样如果你还是搞不懂～没关系，我们来处理个特殊的案例！假设两个档名，分别是底下这样：

- /dir1/file1
- /dir2

假设你现在在系统使用 dmtsai 这个账号，那么这个账号针对 /dir1, /dir1/file1, /dir2 这三个档名来说，分别需要『哪些最小的权限』才能达成各项任务？ 鸟哥汇整如下，如果你看得懂，恭喜你，如果妳看不懂～没关系～未来再来继续学！

操作动作	/dir1	/dir1/file1	/dir2	重点
读取 file1 内容	x	r	-	要能够进入 /dir1 才能读到里面的文件数据！
修改 file1 内容	x	rw	-	能够进入 /dir1 且修改 file1 才行！
执行 file1 内容	x	rx	-	能够进入 /dir1 且 file1 能运作才行！
删除 file1 文件	wx	-	-	能够进入 /dir1 具有目录修改的权限即可！
将 file1 复制到 /dir2	x	r	wx	要能够读 file1 且能够修改 /dir2 内的数据

目录的相关操作

比较特殊的目录，得要用力的记下来才行：

```
.          代表此层目录
..         代表上一层目录
-          代表前一个工作目录
~          代表『目前用户身份』所在的家目录
~account   代表 account 这个用户的家目录(account 是个账号名称)
```

复制、删除与移动：cp,rm,mv

文件内容查阅

- cat 由第一行开始显示文件内容
- tac 从最后一行开始显示，可以看出 tac 是 cat 的倒着写！
- nl 显示的时候，顺道输出行号！
- more 一页一页的显示文件内容
- less 与 more 类似，但是比 more 更好的是，他可以往前翻页！
- head 只看头几行

- tail 只看尾巴几行
- od 以二进制的方式读取文件内容！

学习bash

- 命令与文件补全功能： ([tab] 按键的好处)
- 命令别名设定功能： (alias)，ls -al显示文件的具体信息，但是此命令有点麻烦，可以给他起个别名，如下

```
alias lm='ls -al'
```

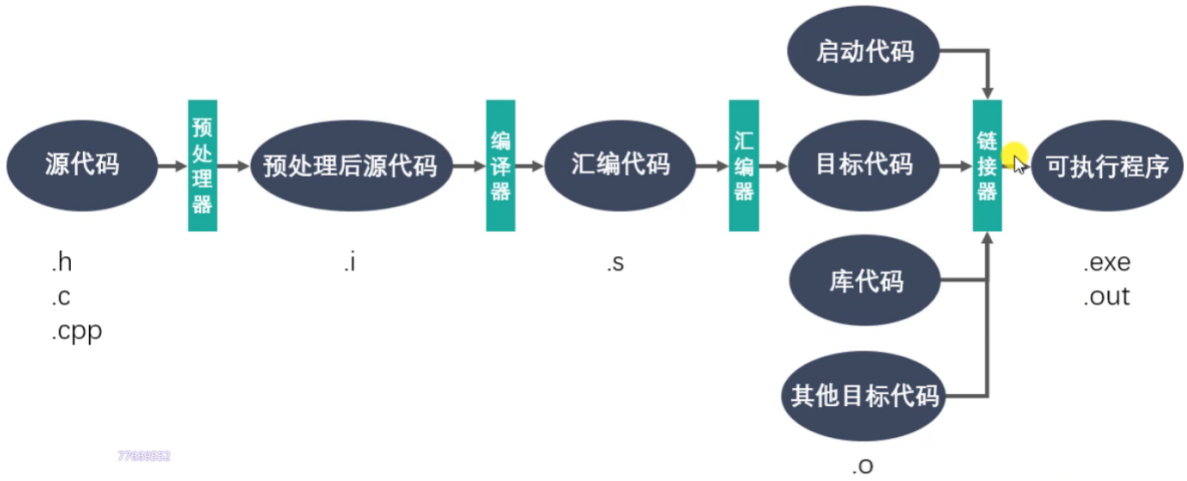
- 想要知道/usr/bin 底下有多少以X 为开头的文件吗？使用：『ls -l /usr/bin/X* 』就能够知道啰

gcc编译

通过gcc编译器进行编译，语法规则`gcc test.c -o app`,test.c是文件名称，-o后面输出的文件名

运行：`./app`

gcc工作流程：



先将头文件展开

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

gcc工作流程

1. -E test.i： 进行预处理，预处理后的源码是 .i类型
2. -S：编译指定的源文件，生成.s结尾的文件
3. -o: -o是指定生成的文件名
4. gcc -c :生成.o文件
5. 直接进行gcc test.c ，编译器会自动把之前的几步都做了，先进行预处理，汇编，生成可执行程序

-o 参数就是执行test.o ，可以直接执行，然后就生成了代码结果

gcc是编译c程序的

g++是编译c++程序的

静态库命名规则

- 命名规则：
  - Linux ： **libxxx.a**
    - lib ： 前缀（固定）
    - xxx ： 库的名字，自己起
    - .a ： 后缀（固定）
  - Windows ： **libxxx.lib**
- 静态库的制作：
  - 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

静态库制作流程：

第一步：

```
gcc -c xxx.c(源码) //生成源码的.o文件
```

```
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lessiono4/calc$ gcc -c add.c div.c main.c mult.c sub.c
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lessiono4/calc$ ls
add.c  add.o  div.c  div.o  head.h  main.c  main.o  mult.c  mult.o  sub.c  sub.o
```

第二步：

```
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lessiono4/calc$ ar rcs libcalc.a add.o sub.o mult.o div.o
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lessiono4/calc$ ls
add.c  add.o  div.c  div.o  head.h  libcalc.a  main.c  mult.c  mult.o  sub.c  sub.o
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lessiono4/calc$ tree
.
├── add.c
├── add.o
├── div.c
├── div.o
├── head.h
├── libcalc.a
├── main.c
├── mult.c
├── mult.o
├── sub.c
└── sub.o
```

```
命令：ar rcs libcalc.a add.o sub.o mult.o div.o //得到静态库
```

静态库使用

学一些linux常用指令：cp -r 递归拷贝 `../`代表上一级目录，下面代码是将lession04的文件夹中的calc和library拷贝到了lession05文件夹中

```
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lessiono4$ cp -r calc/ library ../lession05
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lessiono4$ ls
calc  library
```

```
rm *.o 删除后缀名为.o的文件
```

静态库的使用

```
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lession05/library$ gcc main.c -o app -I ./include/ -l calc -L./lib
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lession05/library$ ls
app  include  lib  main.c  src
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lession05/library$ ./app
a = 20, b = 12
a + b = 32
a - b = 8
a * b = 240
a / b = 1.666667
```

函数的定义在库里，我们在使用静态库时，需要将这些库中的定义与main函数放在同一个目录下

```
-I:找到指定的目录include，这里面定义的是头文件
-l:库的名称
-L：存放库的目录名称
-o:生成的文件名称
```

目录结构：

```
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lession05/library$ tree
.
├── include
│   └── head.h
├── lib
│   └── libsuanshu.a
├── main.c
└── src
    ├── add.c
    ├── add.o
    ├── div.c
    ├── div.o
    ├── mult.c
    ├── mult.o
    ├── sub.c
    └── sub.o
```

使用库的代码

```
gcc main.c -I./include -L ./lib/ -l suanshu -o app  //顺序可以变化，使用静态库
```

```
静态库的制作与使用过程
1. gcc -c xxx.c(源码) //生成源码的.o文件
2. 命令：ar rcs libcalc.a(静态库的名称) add.o sub.o mult.o div.o //得到静态库
3. gcc main.c -I./include -L ./lib/ -l suanshu -o app  //顺序可以变化，使用静态库
```

```
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lession05/library$ gcc main.c -I./include/ -o app -l suanshu -L./lib
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lession05/library$ ls
app  include  lib  main.c  src
```

运行结果：

```
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lession05/library$ ./app
a = 20, b = 12
a + b = 32
a - b = 8
a * b = 240
a / b = 1.666667
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lession05/library$
```

动态库的制作与使用

- 命名规则：
  - Linux ： **libxxx.so**
    - lib ： 前缀（固定）
    - xxx ： 库的名字，自己起
    - .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 libcalc.so

动态库的使用与制作
1.先生成.o文件 gcc -c -fpic add.c mult.c sub.c div.c
2.得到动态库 gcc -shared add.c sub.o mult.o div.o -o libcalc.so
//接下来生成可执行文件
//gcc main.c -o main -I ./include -L lib -l calc //但是有问题，需要解决问题，失败的原因来自于动态库的工作原理，动态库需要配置环境变量才可以
用pwd命令显示当前目录路径
将其配置给环境变量,如下语句
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/media/kimber/Data/Linux/lession06/library/lib
用户级别的配置：
输入vim .bashrc按回车，再按shift+: ，到达最后一行,按o往下插入一行
退出vim命令：按esc退出编辑，按shift+: ，到达最后一行，输入wq退出
再输入source .bashrc 配置一下 ldd命令查看
vim ~/.bashrc可以回到家目录下的vim中（不知道为什么这个配置失败了）
cd ~/ 回到主目录

系统配置输入：sudo vim /etc/profile 这种方式可以配置成功
配置好之后输入 source /etc/profile
再到library路径下输入ldd main检查一下，如果出现如下图中的路径，就说明成功了

```
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lession06/library$ ldd main
        linux-vdso.so.1 (0x00007ffc9a3fa000)
        libcalc.so => /media/kimber/Data/Linux/lession06/library/lib/libcalc.so (0x00007f13a4099000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f13a3e96000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f13a40a5000)
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lession06/library$
```

静态库和动态库工作原理

- 静态库：GCC 进行链接时，会把静态库中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
- 程序启动之后，动态库会被动态加载到内存中，通过 ldd （list dynamic dependencies）命令检查动态库依赖关系
- 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。

Xshell常用快捷键

ctrl + d	删除光标所在位置上的字符相当于VIM里x或者dl
ctrl + h	删除光标所在位置前的字符相当于VIM里hx或者dh
ctrl + k	删除光标后面所有字符相当于VIM里d shift+$
ctrl + u	删除光标前面所有字符相当于VIM里d shift+^
ctrl + w	删除光标前一个单词相当于VIM里db
ctrl + y	恢复ctrl+u上次执行时删除的字符
ctrl + ?	撤消前一次输入
alt + r	撤消前一次动作
alt + d	删除光标所在位置的后单词

移动

ctrl + a	将光标移动到命令行开头相当于VIM里shift+^
ctrl + e	将光标移动到命令行结尾处相当于VIM里shift+$
ctrl + f	光标向后移动一个字符相当于VIM里1
ctrl + b	光标向前移动一个字符相当于VIM里h
ctrl + 方向键左键	光标移动到前一个单词开头
ctrl + 方向键右键	光标移动到后一个单词结尾
ctrl + x	在上次光标所在字符和当前光标所在字符之间跳转
alt + f	跳到光标所在位置单词尾部

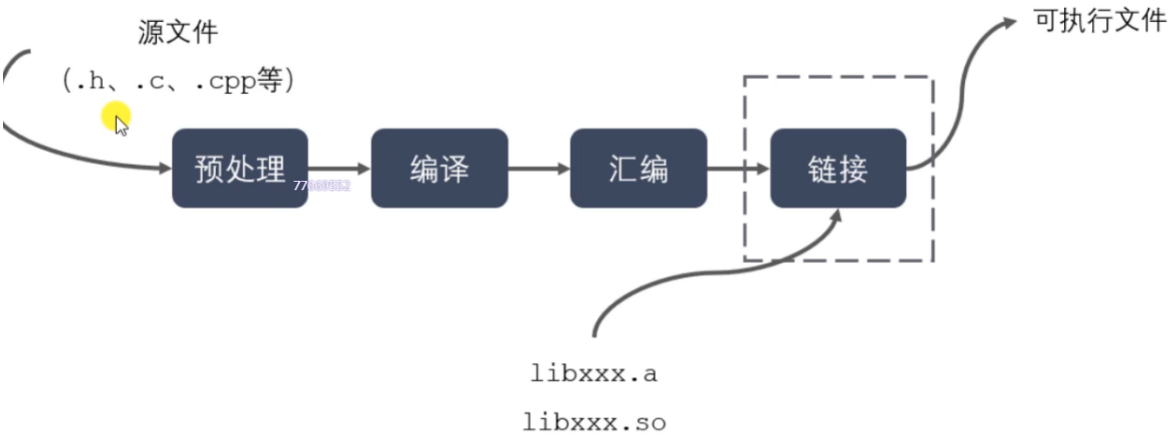
历史命令编辑

```
ctrl + p      返回上一次输入命令字符
ctrl + r          输入单词搜索历史命令
alt    + p        输入字符查找与字符相接近的历史命令
alt    + >        返回上一次执行命令
```

其它

```
ctrl + s          锁住终端
ctrl + q          解锁终端
ctrl + l              清屏相当于命令clear
ctrl + c          另起一行
ctrl + i          类似TAB健补全功能
ctrl + o          重复执行命令
alt    + 数字键     操作的次数
```

程序编译成可执行程序的过程



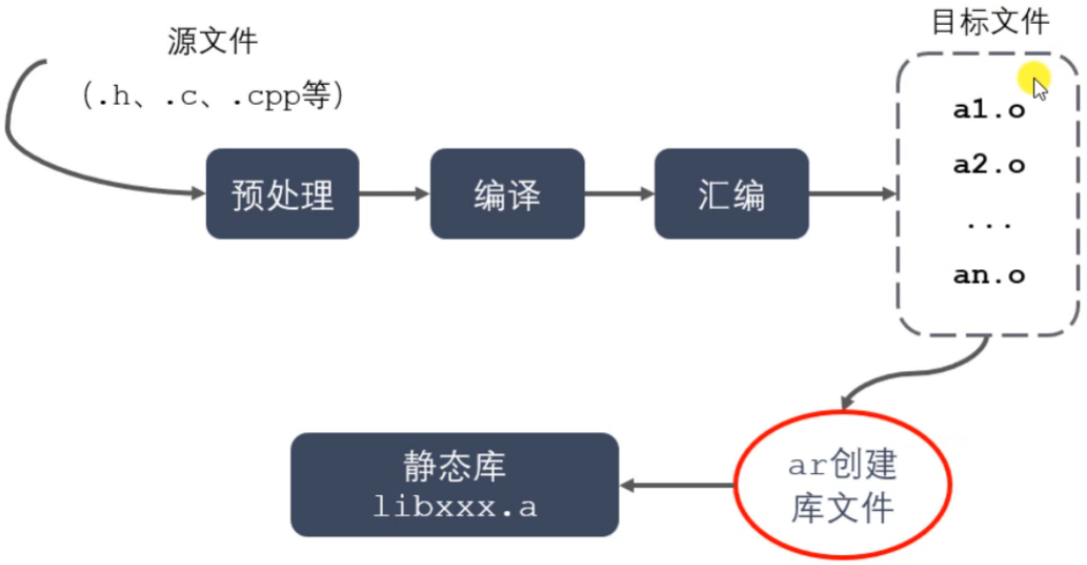
静态库、动态库区别来自链接阶段如何处理，链接成可执行程序。分别称为静态链接方式和动态链接方式。

先对源文件进行预处理，再进行编译成汇编文件，再链接成可执行文件

在链接阶段，静态库是将其中的代码链接到可执行程序中，可以直接加载静态库中的代码

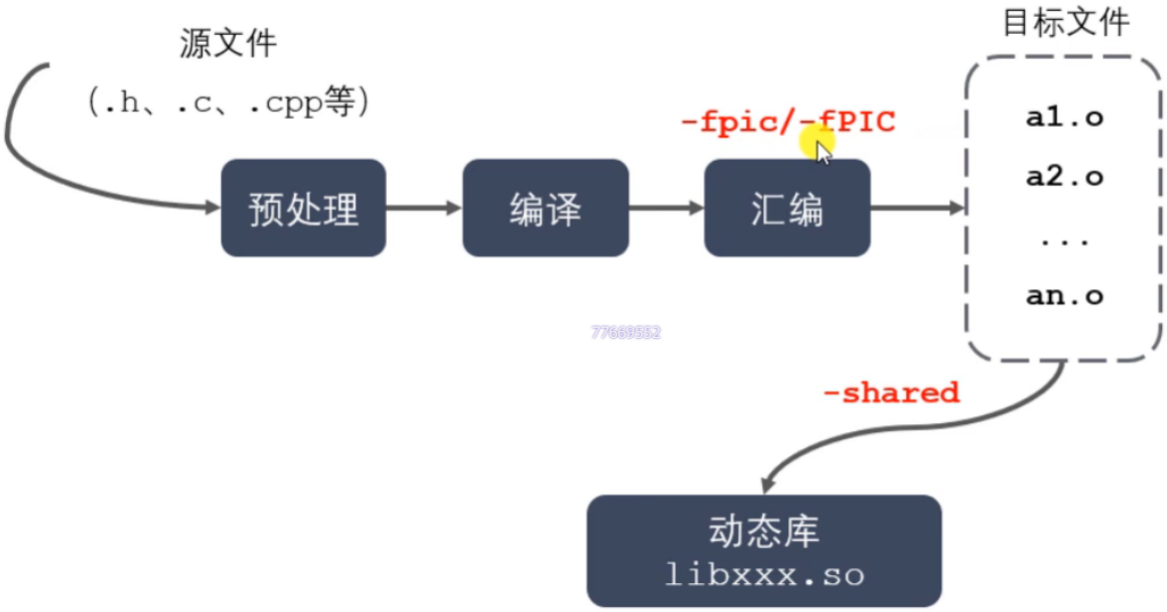
动态库有所不同，动态库链接的是动态库的名称，然后再根据名字去找文件

静态库的制作过程



使用的时候将静态库以及头文件分发给别人

动态库的制作过程



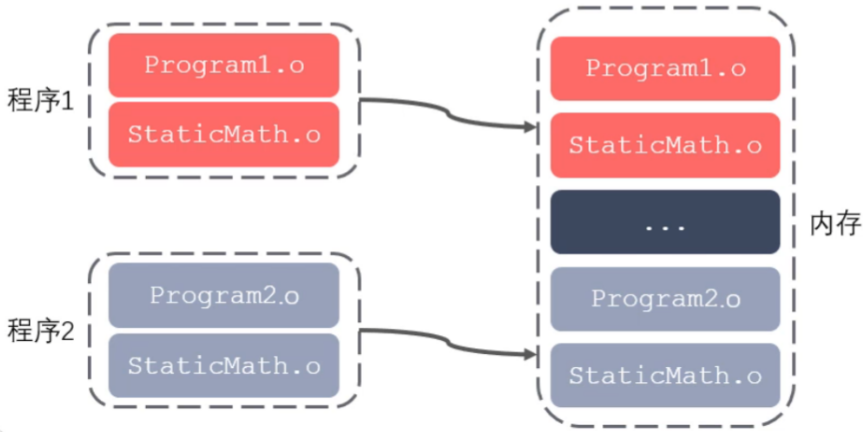
需要加上-fpic，这样可以生成与位置无关的目标代码，静态库的位置是不变的，动态库的加载位置不确定，是变化的，动态库的文件以及头文件也需要发送给别人，动态库使用的时候，是动态的加载其中的api，先去查找动态库的路径，找到之后再加载到内存空间

静态库的优缺点

库比较小，建议使用静态库，库比较大，使用动态库

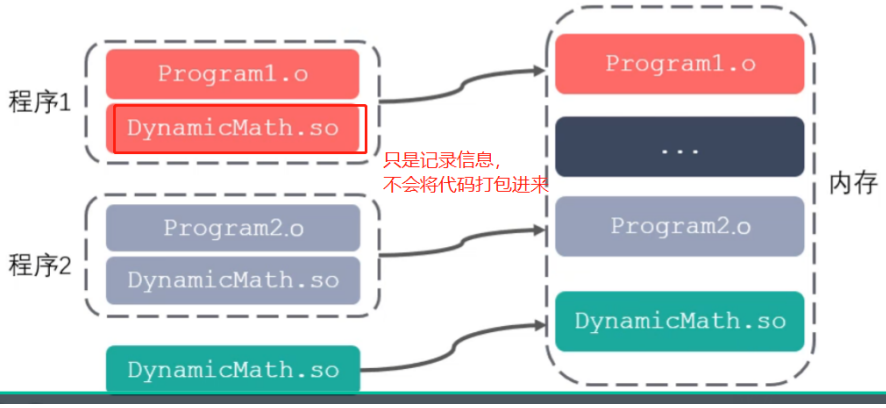
04 / 静态库的优缺点

- 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



动态库的优缺点

- 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



makefile

- 一个工程中的源文件不计其数，其按类型、功能、模块分别放在若干个目录中，Makefile 文件定义了一系列的规则来指定哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作，因为 Makefile 文件就像一个 Shell 脚本一样，也可以执行操作系统的命令。
- Makefile 带来的好处就是“自动化编译”，一旦写好，只需要一个 make 命令，整个工程完全自动编译，极大的提高了软件开发的效率。make 是一个命令工具，是一个解释 Makefile 文件中指令的命令工具，一般来说，大多数的 IDE 都有这个命令，比如 Delphi 的 make，Visual C++ 的 nmake，Linux 下 GNU 的 make。

Makefile文件命名和规则

- 文件命名

  makefile 或者 Makefile

- Makefile 规则
  - 一个 Makefile 文件中可以有一个或者多个规则

    目标 ...: 依赖 ...

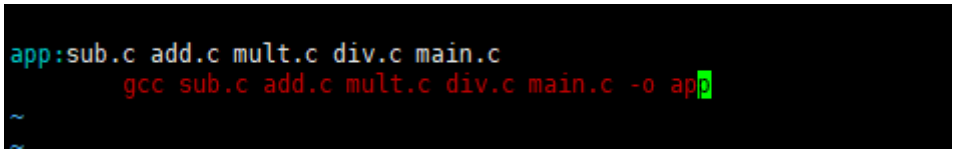
    命令（Shell 命令）

    ...

    - 目标：最终要生成的文件（伪目标除外）
    - 依赖：生成目标所需要的文件或是目标
    - 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）

```
执行：vim Makefile，进入vim中进行Makefile规则指定
app:sub.c add.c mult.c div.c    //这是依赖
    gcc sub.c add.c mult.c div.c -o app //这是命令，前面必须有Tab键
然后回到终端，输入：make，会生成app（可执行程序）
```

这是编写的Makefile文件：



后续操作：

```
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lession07$ make
gcc sub.c add.c mult.c div.c main.c -o app
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lession07$ ls
add.c  app  div.c  head.h  main.c  Makefile  mult.c  redis-5.0.10.tar.gz  sub.c
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lession07$ ./app
a = 20, b = 12
a + b = 32
a - b = 8
a * b = 240
a / b = 1.666667
kimber@i9-12900k-4-ubuntu:/media/kimber/Data/Linux/lession07$
```

简化后的版本，可以参考ppt

```
#定义变量
src=sub.o add.o mult.o div.o main.o
target=app
#获取变量的名称
$(target):$(src)
	$(CC) $(src) -o $(target)

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

再进行简化：

1 Ubuntu-kimber

```
#定义变量
#src=sub.o add.o mult.o div.o main.o
#使用函数获取指定类型的文件
src=$(wildcard ./*.c)
#获取之后，将*.c替换成 *.o,这个函数有三个部分，将第三部分中的符合第一部分的内容替换成第二部分的内容
objs=$(patsubst %.c,%.o,$(src))
target=app
#获取变量的名称
$(target):$(objs)
	$(CC) $(objs) -o $(target)

%.o:%.c
	$(CC) -c $< -o $@
.PHONY:clean
clean:
	rm $(objs) -f
```

最后的.PHONY是伪目标，这样就不会生成clean文件 这样执行 make clean就可以一直删除了

GDB 调试

1. 输入命令,后面要加一个-g

例如：gcc hello.c -o hello -g

2. 然后打开可执行程序：gdb hello
3. 输入l指令从默认位置显示10行代码
4. 打上断点：b 10(在第10行处打上断点) i b 查看断点信息
5. 运行程序：r 单步调试：n，执行完毕：c

IO函数

open打开文件

```
/*
    #include <sys/types.h>
    #include <sys/stat.h>
    #include <fcntl.h>

    // 打开一个已经存在的文件
    int open(const char *pathname, int flags);
        参数：
            - pathname：要打开的文件路径
            - flags：对文件的操作权限设置还有其他的设置
              O_RDONLY,  O_WRONLY,  O_RDWR  这三个设置是互斥的
        返回值：返回一个新的文件描述符，如果调用失败，返回-1

    errno：属于Linux系统函数库，库里面的一个全局变量，记录的是最近的错误号。

    #include <stdio.h>
    void perror(const char *s);作用：打印errno对应的错误描述
        s参数：用户描述，比如hello,最终输出的内容是   hello:xxx(实际的错误描述)


    // 创建一个新的文件
    int open(const char *pathname, int flags, mode_t mode);
*/
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main() {

    // 打开一个文件
    int fd = open("a.txt", O_RDONLY);

    if(fd == -1) {
        perror("open");
    }
    // 读写操作

    // 关闭
    close(fd);

    return 0;
}
```

open创建新文件

```
/*
可以输入man 2 open查看函数描述
    #include <sys/types.h>
    #include <sys/stat.h>
    #include <fcntl.h>
```

```
//创建一个新的文件
    int open(const char *pathname, int flags, mode_t mode);
        参数：
            - pathname：要创建的文件的路径
            - flags：对文件的操作权限和其他的设置
                - 必选项：O_RDONLY,  O_WRONLY, O_RDWR  这三个之间是互斥的
                - 可选项：O_CREAT 文件不存在，创建新文件
            - mode：八进制的数，表示创建出的新的文件的操作权限，比如：0775
            最终的权限是：mode & ~umask
            0777   ->   111111111
        &   0775   ->   111111101
        -----------------------------
                       111111101
        按位与：0和任何数都为0
        umask的作用就是抹去某些权限。

        flags参数是一个int类型的数据，占4个字节，32位。
        flags 32个位，每一位就是一个标志位。

*/
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main() {

    // 创建一个新的文件
    int fd = open("create.txt", O_RDWR | O_CREAT, 0777);

    if(fd == -1) {
        perror("open");
    }

    // 关闭
    close(fd);

    return 0;
}
```

read() write()函数

```
/*
    #include <unistd.h>
    ssize_t read(int fd, void *buf, size_t count);
        参数：
            - fd：文件描述符，open得到的，通过这个文件描述符操作某个文件
            - buf：需要读取数据存放的地方，数组的地址（传出参数）
            - count：指定的数组的大小
        返回值：
            - 成功：
                >0：返回实际的读取到的字节数
                =0：文件已经读取完了
            - 失败：-1 ，并且设置errno
```

```
    #include <unistd.h>
    ssize_t write(int fd, const void *buf, size_t count);
        参数:
            - fd: 文件描述符，open得到的，通过这个文件描述符操作某个文件
            - buf: 要往磁盘写入的数据，数据内容
            - count: 要写的数据的实际的大小
        返回值:
            成功：实际写入的字节数
            失败：返回-1，并设置errno
*/
```

拷贝文件(读写操作)

```
//复制English.txt文件
/*
思路如下:
1.首先通过open函数打开文件（english.txt）
2.创建一个新的文件（拷贝文件）
3.频繁的读写操作
4.关闭文件

*/
#include<stdio.h>
#include<unistd.h>
#include<sys/stat.h>
#include<fcntl.h>

int main(){
    //1.通过open函数打开english.txt文件
    int srcfd = open("english.txt",O_RDONLY); //权限是只读
    if(srcfd == -1){
        perror("open");
        return -1;
    }
    //2.创建一个新的文件(拷贝文件)
    int destfd = open("cpy.txt",O_WRONLY | O_CREAT,0664);  //权限是只写,如果没有就创
建
    if(destfd == -1){
        perror("open");
        return -1;
    }

    //3.频繁的读写操作
    char buf[1024] = {0}; //将数据读到buf这个地址空间

    //如果设置的buf空间不够存放读取的数，就返回读到的自节数
    //如果返回0，就说明读完了，如果返回-1说明读取失败了
    //int len = read(srcfd,buf,sizeof(buf));
    int len = 0;
    while((len = read(srcfd,buf,sizeof(buf))) > 0){
        write(destfd,buf,len);  //将读的len写进目标文件

    }
```

```
    //4.关闭文件
    close(destfd);
    close(srcfd);

    return 0;
}
```

lseek函数

```
/*
    标准C库的函数
    #include <stdio.h>
    int fseek(FILE *stream, long offset, int whence);

    Linux系统函数
    #include <sys/types.h>
    #include <unistd.h>
    off_t lseek(int fd, off_t offset, int whence);
        参数：
            - fd：文件描述符，通过open得到的，通过这个fd操作某个文件
            - offset：偏移量
            - whence:
                SEEK_SET
                    设置文件指针的偏移量
                SEEK_CUR
                    设置偏移量：当前位置 + 第二个参数offset的值
                SEEK_END
                    设置偏移量：文件大小 + 第二个参数offset的值
        返回值：返回文件指针的位置


    作用：
        1.移动文件指针到文件头
        lseek(fd, 0, SEEK_SET);

        2.获取当前文件指针的位置
        lseek(fd, 0, SEEK_CUR);

        3.获取文件长度
        lseek(fd, 0, SEEK_END);

        4.拓展文件的长度，当前文件10b，110b，增加了100个字节
        lseek(fd, 100, SEEK_END)
        注意：需要写一次数据

*/

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main() {
```

```
    int fd = open("hello.txt", O_RDWR);

    if(fd == -1) {
        perror("open");
        return -1;
    }

    // 扩展文件的长度
    int ret = lseek(fd, 100, SEEK_END);
    if(ret == -1) {
        perror("lseek");
        return -1;
    }

    // 写入一个空数据
    write(fd, " ", 1);

    // 关闭文件
    close(fd);

    return 0;
}
```

stat、lstat函数

```
/*
    #include <sys/types.h>
    #include <sys/stat.h>
    #include <unistd.h>

    int stat(const char *pathname, struct stat *statbuf);
        作用：获取一个文件相关的一些信息
        参数:
            - pathname：操作的文件的路径
            - statbuf：结构体变量，传出参数，用于保存获取到的文件的信息
        返回值：
            成功：返回0
            失败：返回-1 设置errno

    int lstat(const char *pathname, struct stat *statbuf);
        作用：获取软链接文件的信息
        参数:
            - pathname：操作的文件的路径
            - statbuf：结构体变量，传出参数，用于保存获取到的文件的信息
        返回值：
            成功：返回0
            失败：返回-1 设置errno

*/

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
```

```c
#include <stdio.h>

int main() {

    struct stat statbuf;

    int ret = stat("a.txt", &statbuf);

    if(ret == -1) {
        perror("stat");
        return -1;
    }

    printf("size: %ld\n", statbuf.st_size);


    return 0;
}
```

模拟ls -l的功能

```c
#include<stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include<pwd.h>
#include<grp.h>
#include<time.h>
#include<string.h>
//模拟实现ls -l指令,实现类似下面的文件信息
//-rwxrwxrwx 1 kimber kimber    12 11月 17 13:40 a.txt
int main(int argc,char* argv[]){
    //判断输入的参数是否正确
    if(argc < 2){
        printf("%s filename\n",argv[0]);
        return -1;
    }
    //通过stat函数获取用户传入的文件的信息
    struct stat st;
    int ret = stat(argv[1], &st); //文件的信息保存在st中
    if(ret == -1){
        perror("stat");
        return -1;
    }

    //获取文件类型和文件权限，保存在字符串数组中
    char perms[11] = {0};
    //判断文件类型，一共有7种权限
    switch (st.st_mode & S_IFMT)
    {
    case S_IFLNK:
        perms[0] = 'l';
        break;
    case S_IFDIR:
        perms[0] = 'd';
```

```
        break;
    case S_IFREG:
        perms[0] = '-';
        break;
    case S_IFBLK:
        perms[0] = 'b';
        break;
    case S_IFCHR:
        perms[0] = 'c';
        break;
    case S_IFSOCK:
        perms[0] = 's';
        break;
    case S_IFIFO:
        perms[0] = 'p';
        break;
    default:
        perms[0] = '?';
        break;
    }
    //文件所有者
    perms[1] = (st.st_mode & S_IRUSR) ? 'r' : '-';
    perms[2] = (st.st_mode & S_IWUSR) ? 'w' : '-';
    perms[3] = (st.st_mode & S_IXUSR) ? 'x' : '-';

    //文件所在组权限
    perms[4] = (st.st_mode & S_IRGRP) ? 'r' : '-';
    perms[5] = (st.st_mode & S_IWGRP) ? 'w' : '-';
    perms[6] = (st.st_mode & S_IXGRP) ? 'x' : '-';

    perms[7] = (st.st_mode & S_IROTH) ? 'r' : '-';
    perms[8] = (st.st_mode & S_IWOTH) ? 'w' : '-';
    perms[9] = (st.st_mode & S_IXOTH) ? 'x' : '-';

    //硬链接数
    int linkNum = st.st_nlink;
    //文件所有者
    char* fileuser = getpwuid(st.st_uid) ->pw_name;
    //文件所在组
    char* fileGrp = getgrgid(st.st_gid)->gr_name;
    //文件大小
    long int fileSize = st.st_size;
    //获取修改的时间,ctime将秒数转换成本地的时间，传递的指针
    char* time = ctime(&st.st_mtime);
    char mtime[512] = {0};
    strncpy(mtime, time, strlen(time) - 1);
    char buf[1024];
    sprintf(buf,"%s %d %s %s %ld %s
%s",perms,linkNum,fileuser,fileGrp,fileSize,mtime,argv[1]);
    printf("%s\n",buf);

    return 0;
}
```

文件属性操作函数

access函数

```
/*
输入man 2 access 查看
    #include <unistd.h>
    int access(const char *pathname, int mode);
    作用：判断某个文件是否有某个权限，或者判断文件是否存在
    参数：
        -pathname：判断文件路径
        -mode:
            R_OK:判断是否有读权限
            W_OK:判断是否有写权限
            X_OK:判断是否有可执行权限
            F_OK:判断文件是否存在
        返回值：成功返回0，失败返回-1
*/
#include <unistd.h>
#include<stdio.h>

int main(){
    int ret = access("a.txt",F_OK);
    if(ret == -1){
        perror("access");
        return -1;
    }
    printf("文件存在！！！\n");
    return 0;
}
```

chmod函数

```
#include <sys/stat.h>
#include<stdio.h>
/*
       #include <sys/stat.h>

       int chmod(const char *pathname, mode_t mode);
       作用：修改文件的权限
       参数：
        -pathname：需要修改的文件的路径
        -mode:需要修改的权限值，八进制的数

       int fchmod(int fd, mode_t mode);


*/
int main(){
    int ret = chmod("a.txt",0775);
    if(ret == -1){
        perror("chmod");
        return -1;
    }
```

```
    return 0;
}
```

truncate函数

```
/*
       #include <unistd.h>
       #include <sys/types.h>
       int truncate(const char *path, off_t length);
       作用：缩减或扩展文件的尺寸至指定的大小
       参数：
       -path:需要修改的文件的路径
       -length：需要最终文件的大小
        返回值：成功返回0，失败返回-1

*/
#include <unistd.h>
#include <sys/types.h>
#include<stdio.h>
int main(){
    int ret = truncate("b.txt",20);
    if(ret == -1){
        perror("truncate");
        return -1;
    }
    return 0;
}
```

目录遍历函数

```
/*
       //打开一个目录
       #include <sys/types.h>
       #include <dirent.h>
       DIR *opendir(const char *name);
       参数：
            -name:需要打开的目录的名称
       返回值：
            DIR *类型，理解为目录流
            错误返回NULL

       //读取目录中的数据
       #include <dirent.h>
       struct dirent *readdir(DIR *dirp);
       参数：
           dirp是opendir返回的值
        返回值：
           struct dirent，代表读取到的文件信息

       #include <sys/types.h>
       #include <dirent.h>
       int closedir(DIR *dirp);
```

```c
*/
//读取某个目录下所有的普通文件
#include <sys/types.h>
#include <dirent.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

int getFileNum(const char* path);
int main(int argc,char* argv[]){
    //没有传入数据就会输出下面的话
    if(argc < 2){
        printf("%s path\n" , argv[0]);
        return -1;
    }
    int num = getFileNum(argv[1]);
    printf("普通文件的个数为： %d\n",num);
    return 0;
}

//用于获取目录下所有普通文件的个数
//1.首先打开文件（opendir） 2.读取目录中的数据，用struct dirent*，这里面包括许多目录的信息

int getFileNum(const char* path){
    //1.打开目录
    DIR* dir = opendir(path);
    if(dir == NULL){
        perror("opendir");
        exit(0);
    }
    struct dirent* ptr;

    int total = 0;

    while((ptr = readdir(dir)) != NULL){
        //获取名称
        char* dname = ptr->d_name;
        //忽略掉. 和..
        if(strcmp(dname,".") == 0 || strcmp(dname,"..") == 0){
            continue;
        }

        //判断是普通文件还是目录
        if(ptr->d_type == DT_DIR){
            //目录，需要继续读取这个目录
            char newpath[256];
            sprintf(newpath,"%s/%s",path,dname);
            total += getFileNum(newpath);

        }
        if(ptr->d_type == DT_REG){
            //普通文件
            total++;
        }
```

```
    }
    closedir(dir);
    return total;
}
```

```
//结构体和 d_type
struct dirent
{
//此目录进入点的 inode
ino_t d_ino;
//目录文件开头至此目录进入点的位移
off_t d_off;
// d_name的长度 , 不包含 NULL 字符
unsigned short int d_reclen;
// d_name所指的文件类型
unsigned char d_type;
//文件名
char d_name[256];
};
```

```
/*
d_type
DT_BLK 块设备
DT_CHR 字符设备
DT_DIR 目录
DT_LNK 软连接
DT_FIFO 管道
DT_REG 普通文件
DT_SOCK 套接字
DT_UNKNOWN 未知
*/
```

chdir函数

```
/*

    #include <unistd.h>
    int chdir(const char *path);
        作用：修改进程的工作目录
            比如在/home/nowcoder 启动了一个可执行程序a.out, 进程的工作目录
/home/nowcoder
        参数：
            path : 需要修改的工作目录

    #include <unistd.h>
    char *getcwd(char *buf, size_t size);
        作用：获取当前工作目录
        参数：
            - buf : 存储的路径，指向的是一个数组（传出参数）
            - size: 数组的大小
        返回值：
            返回的指向的一块内存，这个数据就是第一个参数
```

```
*/
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>

int main() {

    // 获取当前的工作目录，将获取的工作目录保存在buf数组中
    char buf[128];
    getcwd(buf, sizeof(buf));
    printf("当前的工作目录是：%s\n", buf);

    // 修改工作目录,修改到lession13这个目录下
    int ret = chdir("/media/kimber/Data/Linux/lession13/");
    if(ret == -1) {
        perror("chdir");
        return -1;
    }

    // 创建一个新的文件
    int fd = open("chdir.txt", O_CREAT | O_RDWR, 0664);
    if(fd == -1) {
        perror("open");
        return -1;
    }

    close(fd);

    // 获取当前的工作目录
    char buf1[128];
    getcwd(buf1, sizeof(buf1));
    printf("当前的工作目录是：%s\n", buf1);

    return 0;
}
```

mkdir函数

```
/*
    #include <sys/stat.h>
    #include <sys/types.h>
    int mkdir(const char *pathname, mode_t mode);
        作用：创建一个目录
        参数：
            pathname：创建的目录的路径
            mode：权限，八进制的数
        返回值：
            成功返回0，  失败返回-1
*/

#include <sys/stat.h>
#include <sys/types.h>
```

```c
#include <stdio.h>

int main() {

    int ret = mkdir("aaa", 0777);

    if(ret == -1) {
        perror("mkdir");
        return -1;
    }

    return 0;
}
```

dup\dup2函数

```c
/*
 #include <unistd.h>
 int dup(int oldfd);
 作用：复制新的文件描述符，新的文件描述符与旧的文件描述符指向同一个文件
 fd=3,int fd1 = dup(fd);
 fd指向的是a.txt，fd1也是指向a.txt
 从空闲的文件描述符表中找一个最小的，作为新的拷贝文件描述符
*/
#include <unistd.h>
#include<stdio.h>
#include<fcntl.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<string.h>
int main(){
    int fd = open("a.txt",O_CREAT | O_RDWR,0664);

    int fd1 = dup(fd);
     if(fd1 == -1){
        perror("open");
        return -1;
    }
    printf("fd: %d , fd1 : %d\n" ,fd,fd1); //输出fd: 3 , fd1 : 4
    close(fd);

    char* str = "hello,world";
    int ret = write(fd1,str,strlen(str));
    if(ret == -1){
        perror("write");
        return -1;
    }
    close(fd1);

    return 0;
}
```

```c
/*
    #include <unistd.h>
    int dup2(int oldfd, int newfd);
    作用：重定向文件描述符
    oldfd指向a.txt, newfd指向b.txt
    调用函数成功后：newfd和b.txt做close，newfd指向了a.txt
    oldfd必须是一个有效的文件描述符
    oldfd和newfd值相同，相当于什么都没做

*/
#include <unistd.h>
#include<stdio.h>
#include<fcntl.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<string.h>

int main(){
    //创建一个新的文件，返回一个文件描述符
    int fd = open("1.txt",O_RDWR | O_CREAT,0664);
    //判断是否创建成功
    if(fd == -1){
        perror("open");
        return -1;
    }
    int fd1 = open("2.txt",O_RDWR | O_CREAT,0664);
    if(fd1 == -1){
        perror("open");
        return -1;
    }
    printf("fd: %d , fd1 : %d\n" ,fd,fd1);
    int fd2 = dup2(fd,fd1); //现在fd1指向了fd所代表的文件
    if(fd2 == -1){
        perror("dup2");
        return -1;
    }
    //通过fd1去写数据，实际操作的是1.txt ,而不是 2.txt
    char* str = "hello world";
    int len = write(fd1,str,strlen(str));
    if(len == -1){
        perror("write");
        return -1;
    }
    printf("fd: %d , fd1 : %d, fd2 : %d\n" ,fd, fd1, fd2);
    close(fd);
    close(fd1);

    return 0;
}
```

fcntl函数

```
/*

    #include <unistd.h>
    #include <fcntl.h>

    int fcntl(int fd, int cmd, ...);
    参数:
        fd : 表示需要操作的文件描述符
        cmd: 表示对文件描述符进行如何操作
            - F_DUPFD : 复制文件描述符,复制的是第一个参数fd，得到一个新的文件描述符（返回值）
                int ret = fcntl(fd, F_DUPFD);

            - F_GETFL : 获取指定的文件描述符文件状态flag
              获取的flag和我们通过open函数传递的flag是一个东西。

            - F_SETFL : 设置文件描述符文件状态flag
              必选项：O_RDONLY, O_WRONLY, O_RDWR 不可以被修改
              可选性：O_APPEND, O)NONBLOCK
                O_APPEND 表示追加数据
                NONBLOK 设置成非阻塞

        阻塞和非阻塞：描述的是函数调用的行为。
*/

#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>

int main() {

    // 1.复制文件描述符
    // int fd = open("1.txt", O_RDONLY);
    // int ret = fcntl(fd, F_DUPFD);

    // 2.修改或者获取文件状态flag
    int fd = open("1.txt", O_RDWR);
    if(fd == -1) {
        perror("open");
        return -1;
    }

    // 获取文件描述符状态flag
    int flag = fcntl(fd, F_GETFL);
    if(flag == -1) {
        perror("fcntl");
        return -1;
    }
    flag |= O_APPEND;   // flag = flag | O_APPEND

    // 修改文件描述符状态的flag，给flag加入O_APPEND这个标记
    int ret = fcntl(fd, F_SETFL, flag);
```

```
    if(ret == -1) {
        perror("fcntl");
        return -1;
    }

    char * str = "nihao";
    write(fd, str, strlen(str));

    close(fd);

    return 0;
}
```

Linux多进程开发

fork函数

```
/*
    #include <sys/types.h>
    #include <unistd.h>

    pid_t fork(void);
        函数的作用：创建子进程
        返回值：
            fork()返回值会返回两次，一次在父进程中，一次在子进程中
            在父进程中返回创建的子进程的ID
            在子进程中返回0
            如何区分父进程和子进程：通过fork的返回值
            在父进程中返回-1，表示创建子进程失败，并设置errno


*/
#include <sys/types.h>
#include <unistd.h>
#include<stdio.h>


int main(){
    //创建子进程

    pid_t pid = fork();

    //判断是父进程还是子进程
    if(pid > 0){

        //如果大于0，返回的是父进程中创建的子进程的进程号
        printf("pid : %d\n" ,pid);  //打印的是子进程的ID

        printf("i am parent process,pid: %d,ppid: %d\n",getpid(),getppid());
    }else if(pid == 0){

        //当前是子进程,pid是当前进程的id,ppid是父进程的id
        printf("i am child process,%d,ppid: %d\n",getpid(),getppid());
    }
```

```
    //父子进程共享的代码，二者交替执行
    for(int i = 0; i < 5; i++){
        printf("i : %d , pid: %d\n",i ,getpid());
        sleep(1);
    }
    return 0;
}
```

父子进程具体执行细节，子进程会clone父进程，clone出来一个新的虚拟地址空间，用户区数据都一样，但是pid不同

> 注意1：在栈空间存放的是fork函数的返回值，父进程的栈空间存的是10089，子进程存放的是返回值0.
>
> 此后各自进行操作的互不干扰。

> 注意2: 资源的复制是在写入时才会进行的，在此之前，只有以只读方式共享。fork之后父子进程共享文件。如果要写入数据，就互不干扰，父进程要修改会开辟一块新的地址空间存入修改的数据，子进程同样如此

父子进程总结

```
父子进程之间的关系：
    区别：
        1.fork()函数的返回值不同
            父进程中：>0 返回子进程的pid
            子进程中：=0
        2.pcb中的一些数据不同
            当前进程的id pid
            当前进程的父进程的id ppid
            信号集
    共同点：
    某些状态下，子进程刚创建出来，还没有执行任何的写数据的操作
        -用户区的数据
        -文件描述符表
    父子进程对于变量是否共享？
        -刚开始的时候是一样的，共享的。如果修改了数据，就不共享了
```

-刚开始共享（子进程被创建，两个进程没有做任何写的操作），
读时共享，写时拷贝

GDB多进程调试

使用GDB调试的时候，GDB默认只能追踪一个进程，可以在fork函数调用之前，通过指令设置GDB调试工具跟踪父进程或者跟踪子进程，默认跟踪父进程

设置调试父进程或者子进程：set follow fork mode [parent （默认）| child]

例如：`set follow-fork-mode child` 这样设置完成后，程序会停在子进程的断点处，将父进程执行完毕，父进程如果也设置了断点，则不会起作用。

设置调试模式：set detach-on-fork [on | off]。默认为 on ，表示调试当前进程的时候，其它的进程继续运行，如果为 off ，调试当前进程的时候，其它进程被 GDB 挂起。

查看调试的进程：info inferiors
切换当前调试的进程：inferior id
使进程脱离GDB 调试： detach inferiors id

使用查看的命令，可以看到当前的进程信息，带*指的是当前调试的进程，可以通过上面提到的切换命令进行调试进程的切换

```
(gdb) info inferiors
  Num  Description       Executable
* 1    process 90153     /media/kimber/Data/Linux/lesson18/hello
  2    process 90157     /media/kimber/Data/Linux/lesson18/hello
(gdb)
```

如下所示，进程调试已经成功切换

```
(gdb) inferior 2
[Switching to inferior 2 [process 90157] (/media/kimber/Data/Linux/lesson18/hello)]
[Switching to thread 2.1 (process 90157)]
#0  arch_fork (ctid=0x7ffff7fb7810) at ../sysdeps/unix/sysv/linux/arch-fork.h:49
49      ../sysdeps/unix/sysv/linux/arch-fork.h: 没有那个文件或目录.
(gdb) info inferiors
  Num  Description       Executable
  1    process 90153     /media/kimber/Data/Linux/lesson18/hello
* 2    process 90157     /media/kimber/Data/Linux/lesson18/hello
(gdb)
```