@toc

文件系统与动静态库的基本了解

文件系统

了解Access Modify Change

当文件没有被打开时，他们存放在哪里呢？是**磁盘上面**，我们可以通过命令行上面输入**ls -l**(读取存储在磁盘上的文件信息，然后显示出来) 或者 **stat filename**来查看文件信息，ls命令已经很熟悉了，我们这里来看看stat命令

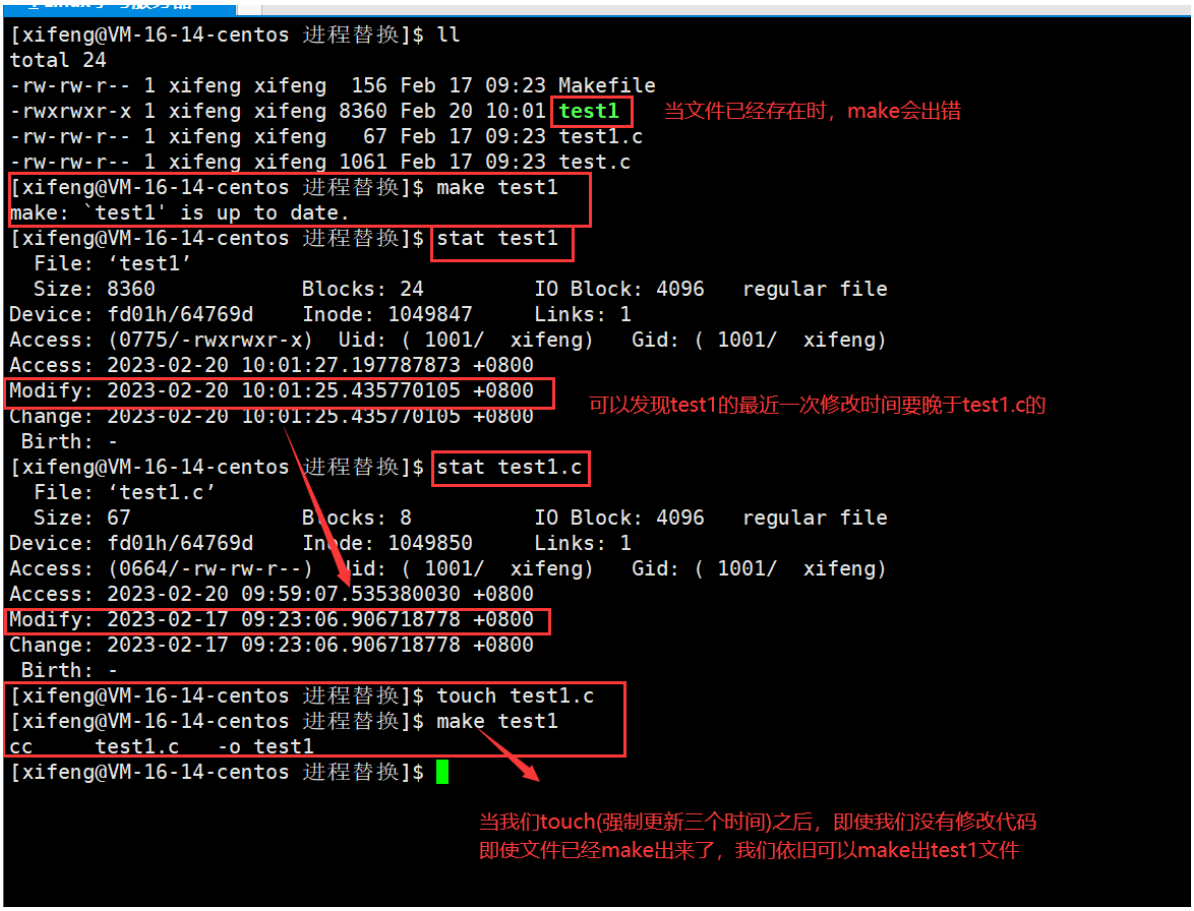
```
[xifeng@VM-16-14-centos 动静态库]$ stat otherd
  File: ‘otherd’
  Size: 4096            Blocks: 8          IO Block: 4096   directory
Device: fd01h/64769d    Inode: 1049868     Links: 3
Access: (0775/drwxrwxr-x)  Uid: ( 1001/  xifeng)   Gid: ( 1001/  xifeng)
Access: 2023-02-18 11:35:53.129161471 +0800
Modify: 2023-02-18 10:53:50.244635625 +0800
Change: 2023-02-18 10:53:50.244635625 +0800
 Birth: -
```

这里我们来着重了解一下这三个时间：

- Access：最近一次访问时间
- Modify：最近一次修改时间-->指文件内容
- Change：最后一修改时间-->指文件属性

我们经常使用的Makefile和make，我们知道当我们不做修改时，如果已经make过一次之后，再次make就会报错说文件已经存在(虚拟依赖除外)，那么操作系统到底为什么能够知道文件已经编译过不需要在编译了呢？**原因就在于Modify也就是修改时间，如果一个文件刚刚编译形成了一个可执行程序，那么形成的可执行程序它的修改时间一定比文件的修改时间要晚，这样Makefile就可以根据两个文件的最**

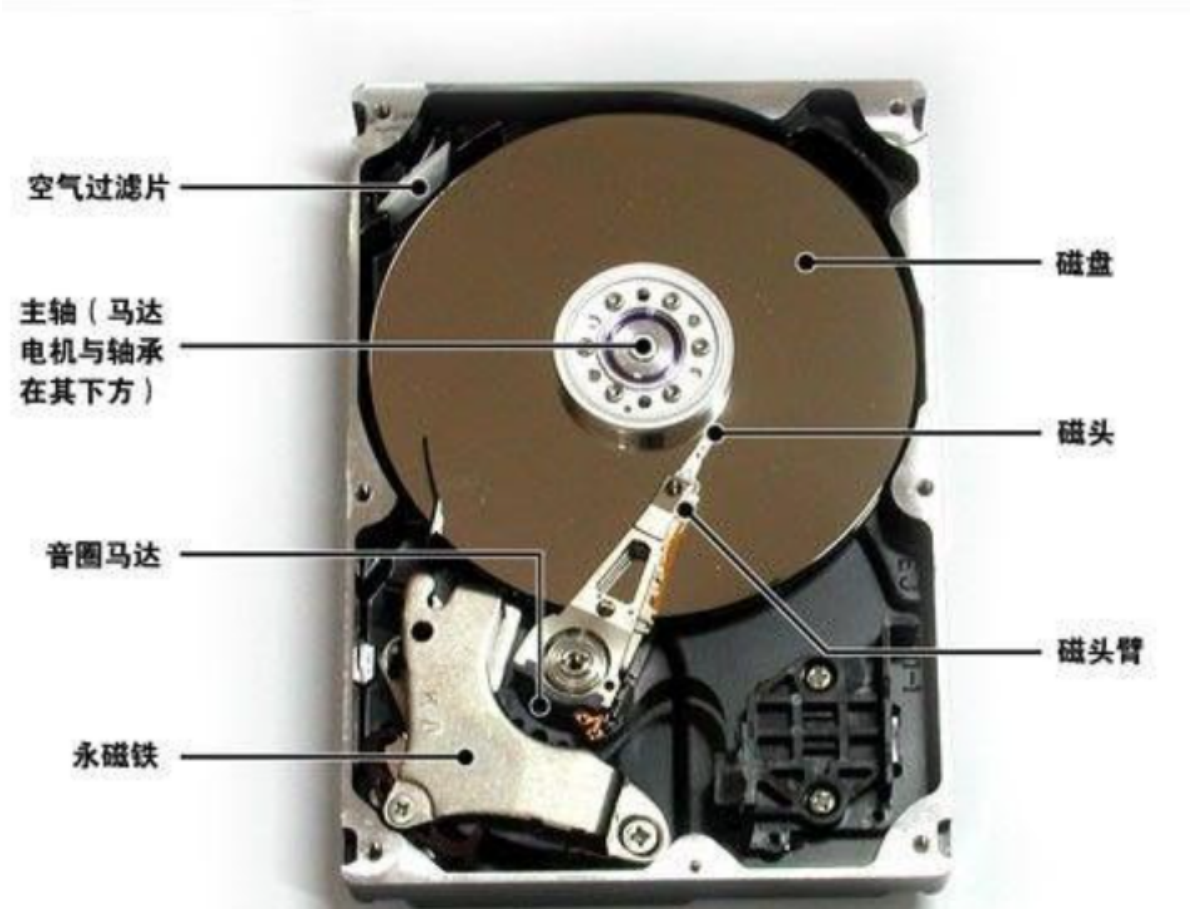
近一次的修改时间来判断是否需要继续编译，以下是例子：



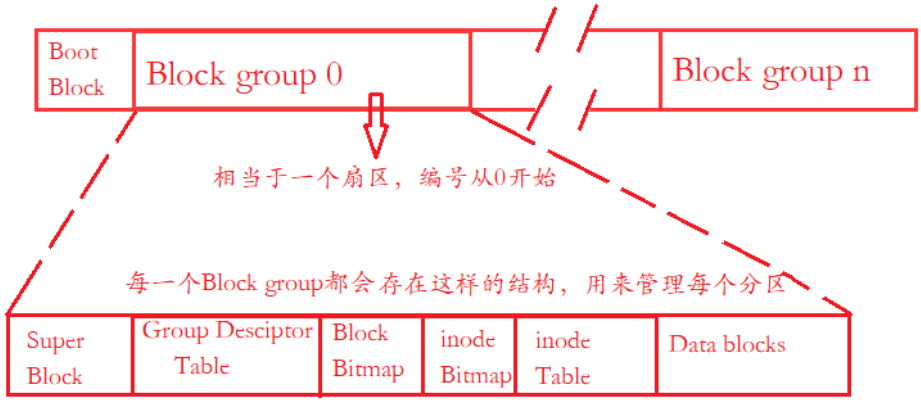
- 要注意虽然说Access是最近一次的访问时间，可是linux内核版本在2.6左右往上，**这个访问时间都不会被立即刷新，要有一定的时间间隔，OS才会自动进行更新时间**，主要是因为访问这一操作比较频繁，如果频繁的刷新会让系统变卡
- **当我们修改文件内容的是时候，大概率是会修改文件的属性的**，比如：可能会更改文件的大小(文件大小也是属性)

inode

Linux上：文件名在系统层面没有意义，文件名是给我们用户用的，Linux中真正标识一个文件，是通过文件的inode编号来标识的，当我们的文件没有被加载到内存中的时候，它们就存储在磁盘上面，**磁盘中最小的存储单元是扇区(1扇区=512Bytes)，文件系统的最小存储单元是block(1Block=4kb=8扇区)**



我们都知道磁盘上面有很多的扇形分区，如果我们把扇形的分区抽象成长方形来看待我们就会得到这样一个图：

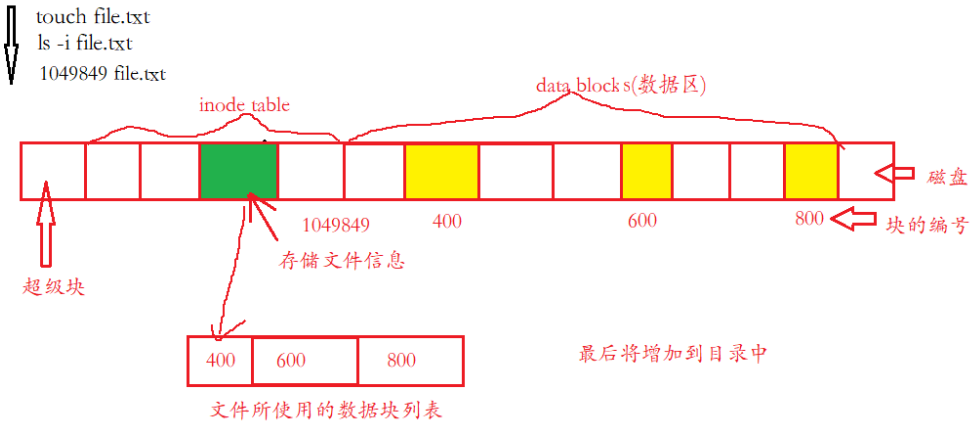


- Linux ext2文件系统，上图为磁盘文件系统图（内核内存映像肯定有所不同），磁盘是典型的块设备，硬盘分区被划分为一个个的block。一个block的大小是由格式化的时候确定的，并且不可以更改。例如mke2fs的-b选项可以设定block大小为1024、2048或4096字节。而上图中启动块（Boot Block）的大小是确定的
- Block Group：ext2文件系统会根据分区的大小划分数个Block Group都有这相同的结构组成
- 超级块(Super Block)：存放文件系统本身的结构信息。记录信息主要有：block和inode的总量，未使用的block和inode的数量，一个Block和一个inode的大小，最近一次的挂载的时间，最近一次写入数据的时间，最近一次检验磁盘的时间等其他文件系统的相关信息。Supper Block的信息被破坏就相当于整个文件系统结构就被破坏了

- **GDT(Group Descriptor Table)**，组描述符表。由很多组描述符组成，整个分区分成多少个组就对应有多少个组描述符。每个组描述符（Group Descriptor）存储一个组的描述信息，例如在这个组中从哪里开始是inode表，从哪里开始是数据块，空闲的inode和数据块还有多少个等等。
- 块位图(Block Bitmap)：这里面记录着Data Block中哪个数据块被占用，哪个数据块没被占用
- inode位图(inode Bitmap)：每个bit表示一个inode空间是否空闲可用
- inode节点表(inode Table)：存放文件属性如文件大小，所有者，最近修改时间等等
- 数据区(Data blocks)：存放文件内容

将属性和文件数据分开存放实际上应该如何实现呢?

我们以touch一个文件为例来看：



创建文件主要有以下四个操作：

1. 存储属性

   内核先找到空闲的inode节点(这里是1049849)，内核把文件信息记录到其中

2. 存储数据

   该文件需要三个磁盘块，内核找到了三个空闲块400，600,800.将内核缓冲区的第一块数据复制到400接着是600依次类推

3. 记录分配情况

   文件内容按顺序400,600,800存放，内核在inode上的磁盘分布记录了上述块列表

4. 添加文件名到目录

   新的文件名file.txt。Linux如何在当前文件目录中记录这个文件内核将入口(149849,file.txt)添加到目录文件，文件名和inode之间的对应关系将文件名和文件内容及其属性连接起来

几个相关问题：(仅是我的理解)

目录的本质是什么呢?

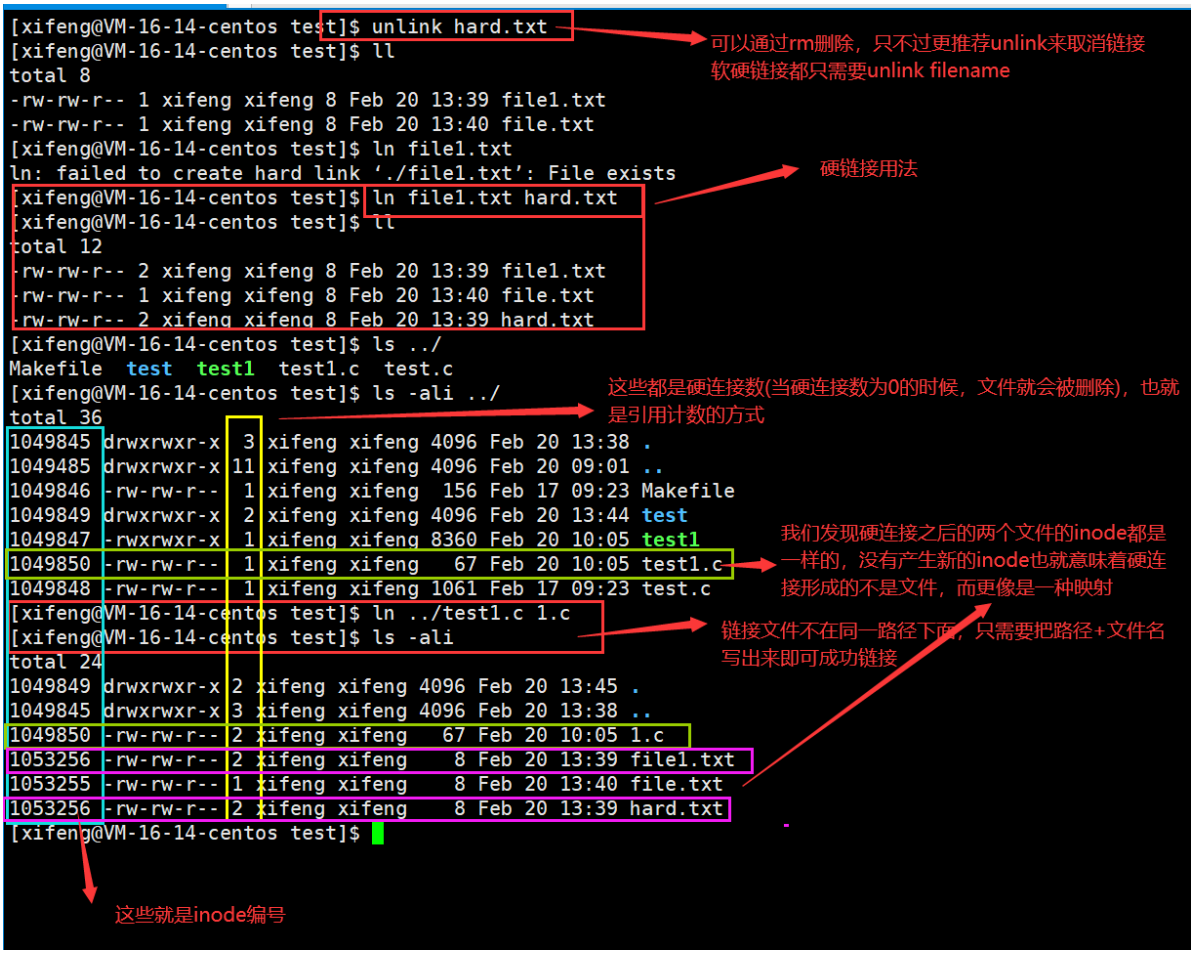
本质可以理解为存放了文件名和对应的inode的链接关系的文件，可以通过文件名去找到对应的inode，从而可以通过inode table来查看文件的属性或者内容数据

删除的本质是什么呢?

删除的本质其实就是把inode bitmap中的对应位置改为未被占用即可，这也是有时候误删之后可以恢复的原理，当如果误删掉一个文件之后如果自己不会恢复就尽量保持原样，不要再进行多余的操作，去找专业的人去解决，因为保持刚删除后的样子能最大程度保证删除之后的数据块没有被覆盖，这样恢复的可能会更大(也就是把inode和数据块之间重新建立链接)

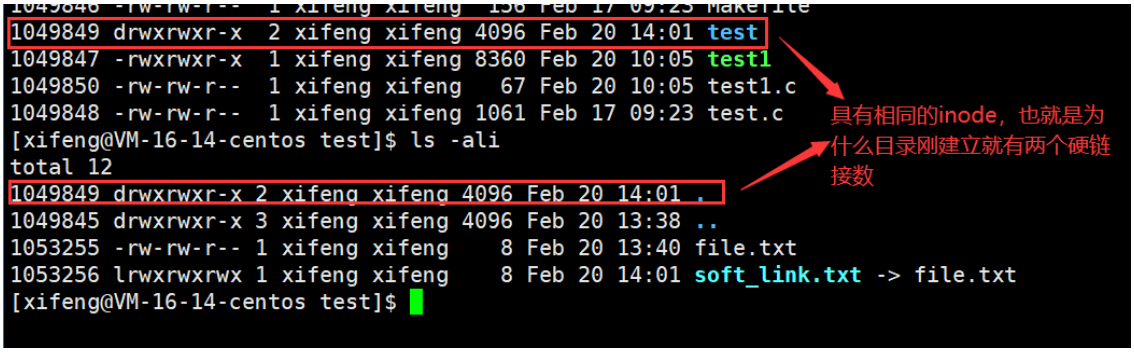
硬链接

首先怎么使用硬链接，在shell中的做法是：**ln 被链接的文件路径及文件名 文件名**



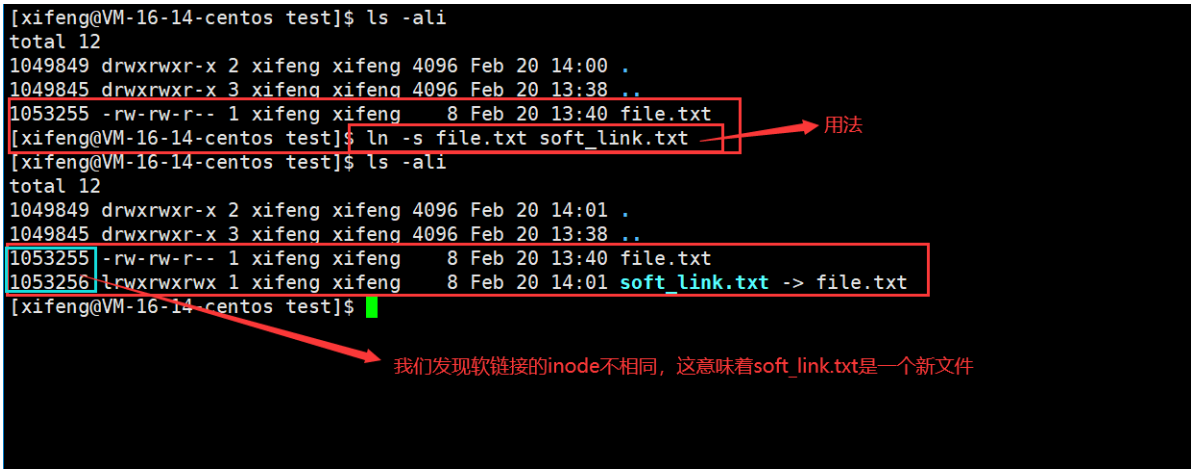
我们看到，真正找到磁盘上文件的并不是文件名，而是inode。 其实在Linux中可以让多个文件名对应于同一个inode，**硬链接本质就不是一个独立的文件，而是一个文件名和inode编号的映射关系，因为它没有自己的inode**

- file1.txt和hard.txt的链接状态完全相同，它们被称为指向文件的硬连接。内核记录了这个连接数inode是1053256硬连接数是2
- 我们在删除文件时干了两件事：
    1. 在目录中将对应的记录删除
    2. 将硬连接数-1，如果为0，则将对应的磁盘释放
- 目录文件在创建时就有两个硬连接



软链接

用法与硬连接相同只是多了个-s：**ln -s 要链接文件的路径以及文件名 文件名**



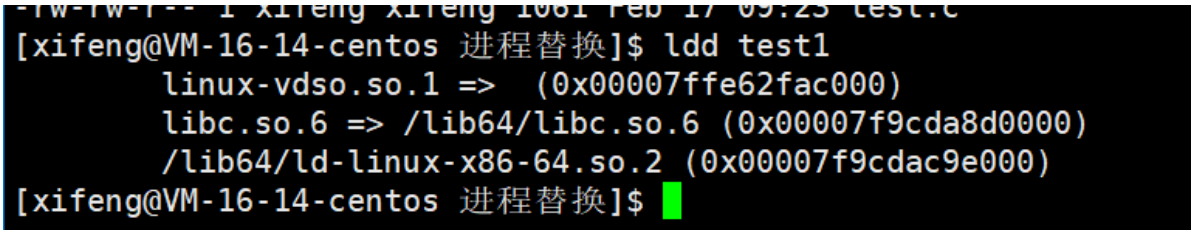
- 软链接形成的是一个新的文件，因为它具有自己的inode属性，也有自己的数据块(保存的是指向文件的所在路径和文件名)
- 如果被链接的文件被删除，那么链接文件也会失效

静态库与动态库

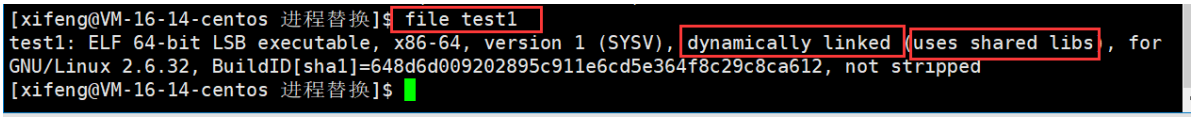
概念

在Linux中如果是动态库，库文件是以.so为后缀的；静态库是以.a为后缀的；在windows中动态库.dll，静态库.lib

- 静态库（.a）：程序在编译链接的时候把库的代码链接到可执行文件中。程序运行的时候将不再需要静态库
- 动态库（.so）：程序在运行的时候才去链接动态库的代码，多个程序共享使用库的代码。
- 一个与动态库链接的可执行文件仅仅包含它用到的函数入口地址的一个表，而不是外部函数所在目标文件的整个机器码
- 在可执行文件开始运行以前，外部函数的机器码由操作系统从磁盘上的该动态库中复制到内存中，这个过程称为动态链接（dynamic linking）
- 动态库可以在多个程序间共享，所以动态链接使得可执行文件更小，节省了磁盘空间。操作系统采用虚拟内存机制允许物理内存中的一份动态库被要用到该库的所有进程共用，节省了内存和磁盘空间



- **ldd filename**：显示可执行程序依赖的库，这里就是libc.so.6库名字就是c库
- 库文件的命名：libXXXX.so or libXXXX.a-...
- 动静态库的名字是去掉lib前缀和.so-或者.a-后缀剩下的部分
- 一般云服务器可能没有内置语言的静态库，而只有动态库可以自己添加一下**sudo yum install glibc -static**



同样file命令也可以查看可执行程序是什么链接，这里就是动态链接，使用的是共享库

静态库的制作

首先我们我们先了解一下原理，我们都知道一个文件想要到可执行，需要以下几个步骤：预处理，编译，汇编，链接。其中汇编之后会形成一个.o下标的二进制文件，并不可以被直接执行,叫可重定向目标文件(-c 是开始进行程序的编译，完成汇编工作就停下)，这个.o文件就是制作动静态库所需的文件，因为其本身已经具有可执行的属性了，只是没有被链接，动静态库其实也就是打包这些.o文件并且附代上.h的头文件，了解到这里我们开始进行一个简单的静态库的制作

- 制作静态库我们可以使用**ar -rc**打包静态库，ar是gnu归档工具，rc表示replace and create
- 查看已经制作完成的静态库:**ar -tv**查看打包内容

  t:列出静态库中的文件
  v:verbose 详细信息

举例:

1. 首先我们先创建四个文件，文件内容如下:

```
//add.h,add.c,sub.h,sub.c
//sub.h
#include<stdio.h>
extern int sub(int x,int y);
//sub.c
#include"sub.h"
int sub(int x,int y)
{
  return x-y;
}
//add.h
#include<stdio.h>
extern int add(int x, int y);
//add.c
#include "add.h"
int add(int x,int y)
{
  return x + y;
}
```

2. 创建出Makefile文件

```
1: Makefile
 1 #打包静态库需要用到ar -rc
 2 libstatic.a:add.o sub.o
 3   ar -rc $@ $^
 4 %o:%c
 5   gcc -c $<
 6 .PHONY:clean
 7 clean:
 8   rm -rf *.o *.a
 9 .PHONY:libs
10 libs:
11   mkdir libs
12   cp *.h libs
13   cp libstatic.a libs
```

libstatic.a文件依赖于add.o sub.o两个文件
使用ar -rc打包， $@ $^可以自动推导文件名

%通配符，也就是所有的.o文件依赖于.c文件， &<意思就是依次展开

现象就是会生成两个.o文件

*通配符，删除所有后缀为.o和.a的文件

这个就是创建一个虚拟依赖libs，当执行make libs时，会先生成一个libs目录，然后把所有的.h文件拷贝到这个目录里，把libstatic.a拷贝进目录中

- 我们之前写的那些代码也都用了库(如c库)，为什么没有指名这些选项呢？--->之前的库，在系统的默认路径下:/lib64/usr/lib, /usr/include等等
- 所以如果我们不想带这些选项，我们可以直接把对应的库和头文件拷贝到默认路径下**首先是可行的，但是非常不推荐这样做，这会污染库**，有时如果库的一些文件与拷贝文件重名还会覆盖掉原先的库
- 一般软件安装的过程其实也就是上面的过程

```
[xifeng@VM-16-14-centos mylibstatic]$ ll
total 20
-rw-rw-r-- 1 xifeng xifeng  59 Feb 17 17:03 add.c
-rw-rw-r-- 1 xifeng xifeng  50 Feb 17 17:03 add.h
-rw-rw-r-- 1 xifeng xifeng 193 Feb 17 18:47 Makefile
-rw-rw-r-- 1 xifeng xifeng  56 Feb 17 18:08 sub.c
-rw-rw-r-- 1 xifeng xifeng  47 Feb 17 18:08 sub.h
[xifeng@VM-16-14-centos mylibstatic]$ make
cc    -c -o add.o add.c
cc    -c -o sub.o sub.c
ar -rc libstatic.a add.o sub.o
[xifeng@VM-16-14-centos mylibstatic]$ ll
total 32
-rw-rw-r-- 1 xifeng xifeng   59 Feb 17 17:03 add.c
-rw-rw-r-- 1 xifeng xifeng   50 Feb 17 17:03 add.h
-rw-rw-r-- 1 xifeng xifeng 1240 Feb 20 16:46 add.o
-rw-rw-r-- 1 xifeng xifeng 2688 Feb 20 16:46 libstatic.a
-rw-rw-r-- 1 xifeng xifeng  193 Feb 17 18:47 Makefile
-rw-rw-r-- 1 xifeng xifeng   56 Feb 17 18:08 sub.c
-rw-rw-r-- 1 xifeng xifeng   47 Feb 17 18:08 sub.h
-rw-rw-r-- 1 xifeng xifeng 1240 Feb 20 16:46 sub.o
[xifeng@VM-16-14-centos mylibstatic]$ make libs
mkdir libs
cp *.h libs
cp libstatic.a libs
[xifeng@VM-16-14-centos mylibstatic]$ ll
total 36
-rw-rw-r-- 1 xifeng xifeng   59 Feb 17 17:03 add.c
-rw-rw-r-- 1 xifeng xifeng   50 Feb 17 17:03 add.h
-rw-rw-r-- 1 xifeng xifeng 1240 Feb 20 16:46 add.o
drwxrwxr-x 2 xifeng xifeng 4096 Feb 20 16:46 libs
-rw-rw-r-- 1 xifeng xifeng 2688 Feb 20 16:46 libstatic.a
-rw-rw-r-- 1 xifeng xifeng  193 Feb 17 18:47 Makefile
-rw-rw-r-- 1 xifeng xifeng   56 Feb 17 18:08 sub.c
-rw-rw-r-- 1 xifeng xifeng   47 Feb 17 18:08 sub.h
-rw-rw-r-- 1 xifeng xifeng 1240 Feb 20 16:46 sub.o
[xifeng@VM-16-14-centos mylibstatic]$ ls libs
add.h  libstatic.a  sub.h
[xifeng@VM-16-14-centos mylibstatic]$
```

原始文件

运行之后产生的.o文件和静态库文件

把头文件和静态库打包到一个目录中去

这样就把静态库自做完成并打包了

使用静态库

我们制作好了静态库应该如何使用呢？其实也就是链接的过程，我们需要先写一段简单的代码：

```
//因为头文件不在同一目录下，所以需要指定一下路径
#include "./libs/add.h"
#include "./libs/sub.h"
int main()
{
  int x=20;
  int y=10;
  printf("add = %d\n",add(x,y));
  printf("sub = %d\n",sub(x,y));
  return 0;
}
```

```
1: Makefile
 1 test:test.c
 2   gcc -o $@ $^ -I./libs -L./libs -l static
 3   # -I是头文件的搜索路径
 4   # -L是库文件的搜索路径
 5   # -l+库文件名 -->除去lib和.a-/.so-之后剩下的部分
 6 # gcc -o $@ $^ 仅仅是这样是不行的，因为编译器找不到静态库的那些头文件，链接不上
 7 # 需要告知详细的路径
 8 .PHONY:clean
 9 clean:
10   rm -rf test
```

另外写法(不在同一级时)

```
1: Makefile
 1 mytest:mytest.o add.o sub.o
 2 #obj= mytest.o add.o sub.o
 3 #mytest:$(obj)
 4   gcc -o $@ $^
 5 %.o:./mylibstatic/%.c
 6   gcc -c $< #这里展开的是add和sub的.c文件，因为不在同一路径下所以需要指定路径
 7 %.o:%.c #这里展开的是mytest.c
 8   gcc -c $< #依次展开
 9 .PHONY:clean
10 clean:
11   rm -rf mytest *.o
```

```
[xifeng@VM-16-14-centos other]$ ./test
add = 30
sub = 10
[xifeng@VM-16-14-centos other]$
```

这样就可以去执行了，因为静态链接的特性，编译的时候会把静态库代码拷贝进我们所写的测试代码中去，所以只要指定好路径，编译形成可执行文件之后就可以直接运行了

动态库的制作

动态库的制作原理和静态库基本一致，总的来说就是打包.o文件具体我们来看操作，还是上面的例子：

1. 创建四个文件并写上相应的内容

```
//sub.h sub.c add.h add.c
```

2. 创建出Makefile文件

```
1: Makefile                                                    buffers
 1 #形成一个共享库
 2 libdynamic.so:add.o sub.o
 3   gcc -shared -o $@ $^ #-shared和-o的位置不能够掉换
 4 #加上-fPIC其他跟之前一样
 5 %o:%c
 6   gcc -fPIC -c $<
 7 .PHONY:clean
 8 clean:
 9   rm -rf *.o libdynamic.so
10 .PHONY:libd
11 libd:
12   mkdir libd
13   cp *.h libd
14   cp libdynamic.so libd
15
```

加上-shared意味着形成共享库

加上-fPIC

- **gcc -fPIC -c $<**产生.o目标文件，程序内部的地址解决方案是：与位位置无关，库文件可以在内存的任何位置加载，而且不影响和其他程序的关联性(-fPIC的作用)
- -shared就是形成一个动态链接的共享库

之后直接make，make libd即可形成动态库

使用动态库

在使用之前我们得先认识到一个东西：**文件编译形成可执行文件的是编译器**，而形成的可执行文件要运行，需要的是**加载器**，两者不是同一个东西

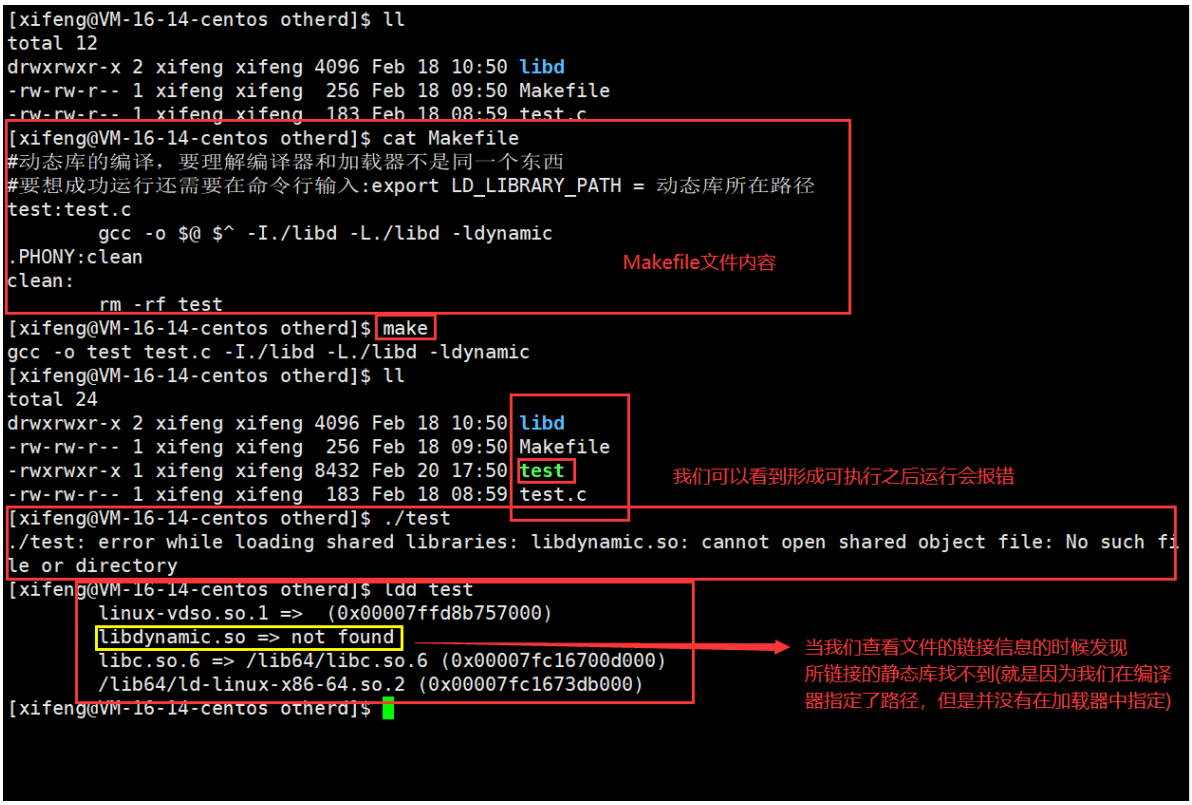
使用动态库同样需要写一个简单的代码

```c
#include "./libd/add.h"
#include "./libd/sub.h"
int main()
{
  int x=20;
  int y=10;
  printf("add = %d\n",add(x,y));
  printf("sub = %d\n",sub(x,y));
  return 0;
}
```

然后开始写Makefile，内容与静态库的使用基本一致

```
1 #动态库的编译，要理解编译器和加载器不是同一个东西
2 #要想成功运行还需要在命令行输入:export LD_LIBRARY_PATH = 动态库所在路径
3 test:test.c
4   gcc -o $@ $^ -I./libd -L./libd -ldynamic
5 .PHONY:clean
6 clean:
7   rm -rf test
```

静态库这样就直接可以运行了，但是动态库不行，这只是代表了编译能通过，但是运行时会报错



这种情况的解决方法有很多种但是这里推荐一种**使用LD_LIBRARY_PATH**具体用法如下：

- export LD_LIBRARY_PATH = 库所在的绝对路径
- 为什么这里不需要指定库名(这里是dynamic)呢？从ldd可以看出可执行文件已经知道所链接的库名了，只是没有路径找不到
- 这种方法需要每一次打开端口的时候都设置一次，export只适用于当前登录，退出之后所做的修改就清除了

还有其他的方法就是:

1. 拷贝.so文件到系统共享库路径下，一般指/usr/lib不过不推荐
2. ldconfig 配置/etc/ld.so.conf.d/，ldconfig更新

总结如何制作

1. 所有的源代码，都需要先被编译成为.o(可重定向目标文件)
2. 制作动静态库的本质就是将所有的.o打包(使用ar或者gcc来进行打包)
3. 交付**头文件** + **-.a 或者 -.so** 文件

gcc和g++优先链接动态