1 库的分类

根据链接时期的不同，库又有静态库和动态库之分。

静态库是在链接阶段被链接的（好像是废话，但事实就是这样），所以生成的可执行文件就不受库的影响了，即使库被删除了，程序依然可以成功运行。

有别于静态库，动态库的链接是在程序执行的时候被链接的。所以，即使程序编译完，库仍须保留在系统上，以供程序运行时调用。（TODO：链接动态库时链接阶段到底做了什么）

2 静态库和动态库的比较

链接静态库其实从某种意义上来说也是一种粘贴复制，只不过它操作的对象是目标代码而不是源码而已。因为静态库被链接后库就直接嵌入可执行文件中了，这样就带来了两个问题。

首先就是系统空间被浪费了。这是显而易见的，想象一下，如果多个程序链接了同一个库，则每一个生成的可执行文件就都会有一个库的副本，必然会浪费系统空间。

再者，人非圣贤，即使是精心调试的库，也难免会有错。一旦发现了库中有bug，挽救起来就比较麻烦了。必须一一把链接该库的程序找出来，然后重新编译。

而动态库的出现正弥补了静态库的以上弊端。因为动态库是在程序运行时被链接的，所以磁盘上只须保留一份副本，因此节约了磁盘空间。如果发现了bug或要升级也很简单，只要用新的库把原来的替换掉就行了。

那么，是不是静态库就一无是处了呢？

答曰：非也非也。不是有句话么：存在即是合理。静态库既然没有湮没在滔滔的历史长河中，就必然有它的用武之地。想象一下这样的情况：如果你用libpcap库编了一个程序，要给被人运行，而他的系统上没有装pcap库，该怎么解决呢？最简单的办法就是编译该程序时把所有要链接的库都链接它们的静态库，这样，就可以在别人的系统上直接运行该程序了。

所谓有得必有失，正因为动态库在程序运行时被链接，故程序的运行速度和链接静态库的版本相比必然会打折扣。然而瑕不掩瑜，动态库的不足相对于它带来的好处在现今硬件下简直是微不足道的，所以链接程序在链接时一般是优先链接动态库的，除非用-static参数指定链接静态库。

**动态链接库**

1. 创建动态链接库

**[cpp]** [view plaincopy](http://blog.csdn.net/orzlzro/article/details/6460058)

1. #include<stdio.h>
2. **void** hello()
3. {
4. printf("hello world/n");
5. }

用命令gcc -shared hello.c -o libhello.so编译为动态库。可以看到，当前目录下多了一个文件libhello.so。

2. 再编辑一个测试文件test.c，内容如下

**[cpp]** [view plaincopy](http://blog.csdn.net/orzlzro/article/details/6460058)

1. #include<stdio.h>
2. **int** main()
3. {
4. printf("call hello()");
5. hello();
6. }

编译 gcc test.c -lhello

-l 选项告诉编译器要使用hello这个库。奇怪的地方是动态库的名字是libhello.so,这里却使用hello.

但这样还不行，编译会出错。

In function `main':

test.c:(.text+0x1d): undefined reference to `hello'

collect2: ld returned 1 exit status

这是因为hello这个库在我们自己的路径中，编译器找不到。

需要使用-L选项，告诉hello库的位置

gcc test.c -lhello -L. -o test

-L .告诉编译器在当前目录中查找库文件

3. 编译成功后执行./test, 仍然出错

说找不到库

有两种方法：

一、可以把当前路径加入 /etc/ld.so.conf中然后运行ldconfig，或者以当前路径为参数运行ldconfig（要有root权限才行）。

二、把当前路径加入环境变量LD\_LIBRARY\_PATH中

当然，如果你觉得不会引起混乱的话，可以直接把该库拷入/lib,/usr/lib/等位置（无可避免，这样做也要有权限），这样链接器和加载器就都可以准确的找到该库了。

我们采用第二种方法：

export LD\_LIBRARY\_PATH=.:$LD\_LIBRARY\_PATH

这样，再执行就成功了。

**下面再讲讲静态链接库**

仍使用刚才的hello.c和test.c。

1. gcc -c hello.c 注意这里没有使用-shared选项

2. 把目标文件归档    ar -r libhello.a hello.o

    程序 ar 配合参数 -r 创建一个新库 libhello.a 并将命令行中列出的对象文件插入。采用这种方法，如果库不存在的话，参数 -r 将创建一个新的库，而如果库存在的话，将用新的模块替换原来的模块。

3. 在程序中链接静态库

           gcc test.c -lhello -L. -static -o hello.static

或者   gcc test.c libhello.a -L. -o hello.static

生成的hello.static就不再依赖libhello.a了

**两个有用的命令**

file程序是用来判断文件类型的，在file命令下，所有文件都会原形毕露的。

顺便说一个技巧。有时在 windows下用浏览器下载tar.gz或tar.bz2文件，后缀名会变成奇怪的tar.tar，到Linux有些新手就不知怎么解压了。但 Linux下的文件类型并不受文件后缀名的影响，所以我们可以先用命令file xxx.tar.tar看一下文件类型，然后用tar加适当的参数解压。

|  |
| --- |
|  |

另外，还可以借助程序ldd实用程序来判断。

ldd是用来打印目标程序（由命令行参数指定）所链接的所有动态库的信息的，如果目标程序没有链接动态库，则打印“not a dynamic executable”，ldd的用法请参考manpage。

转载：<http://blog.csdn.net/orzlzro/article/details/6460058>

gcc -E filename.c -o file\_name.i   进行预编译的结果输出到了终端上，因而需要重定向，或者指明输出的文件名(-o选项)。--> \*.i的文件。  
  
gcc -S filename.i 编译后即停止,不进行汇编.对于每个输入的非汇编语言文件,输出文件是汇编语言文件.这里会自动生成filename.s。--> \*.s的文件。  
  
gcc -c 编译或汇编源文件,但是不作连接.编译器输出对应于源文件的目标文件.(-C告诉编译器不要丢掉注释)。--> \*.o的文件。  
  
gcc -o output\_name  指定输出文件为file.该选项不在乎GCC产生什么输出,无论是可执行文件,目标文件,汇编文件还是预处理后的C代码.--> 可执行的二进制文件。  
  
-l(library)   连接名为library的库文件.连接器在标准搜索目录中寻找这个库文件,库文件的真正名字是`lib(library).a'.连接器会 当做文件名得到准确说明一样引用这个文件.  
  
-L(dir)  在`-L选项的搜索路径列表中添加dir目录.  
  
-static  在支持动态连接(dynamic linking)的系统上,阻止连接共享库.该选项在其他系统上无效.  
  
-shared  生成一个共享目标文件,他可以和其他目标文件连接产生可执行文件.只有部分系统支持该选项.

#include <stdio.h>  
void print1(int i)

{

int j; for(j=0;j<i;j++)

{

printf("%d \* %d = %d\n",j,j,j\*j);

}

 }  
hello2.c \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#include <stdio.h>  
void print2(char \*arr)

{

char c;

int i=0;

while((c=arr[i++])!='\0')

{

printf("%d\*\*\*\*%c\n",i,c);

}

}  
hello.c \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

void print1(int);

void print2(char \*);  
int main(int argc,char \*\*argv)

{

int i=100;

char \*arr="THIS IS LAYMU'S HOME!";

print1(i);

print2(arr);  
return 0;

}

可以看到hello.c要用到hello1.c中的print1函数和hello2.c中的print2函数。所以可以把这两个函数组合为库，以供更多的程序作为组件来调用。

方法一：将hello1.c和hello2.c编译成静态链接库.a

[root@localhost main5]#gcc -c hello1.c hello2.c

  //将hello1.c和hello2.c分别编译为hello1.o和hello2.o，其中-c选项意为只编译不链接。

[root@localhost main5]#ar -r libhello.a hello1.o hello2.o

//将hello1.o和hello2.o组合为libhello.a这个静态链接库

[root@localhost main5]#cp libhello.a /usr/lib

//将libhello.a拷贝到/usr/lib目录下，作为一个系统共享的静态链接库

[root@localhost main5]#gcc -o hello hello.c -lhello

//将hello.c编译为可执行程序hello，这个过程用到了-lhello选项，这个选项告诉gcc编译器到/usr/lib目录下去找libhello.a的静态链接库

以上的过程类似于windows下的lib静态链接库的编译及调用过程。  
方法二：将hello1.o和hello2.o组合成动态链接库.so

[root@localhost main5]#gcc -c -fpic hello1.c hello2.c

 //将hello1.c和hello2.c编译成hello1.o和hello2.o，-c意为只编译不链接，-fpic意为位置独立代码，指示编译程序生成的代码要适合共享库的内容这样的代码能够根据载入内存的位置计算内部地址。

[root@localhost main5]#gcc -shared hello1.o hello2.o -o hello.so

  //将hello1.o和hello2.o组合为shared object，即动态链接库

[root@localhost main5]#cp hello.so /usr/lib

  //将hello.so拷贝到/usr/lib目录下

[root@localhost main5]#gcc -o hello hello.c hello.so

//将hello.c编译链接为hello的可执行程序，这个过程用到了动态链接库hello.so