linux下的静态库和动态库

收藏

我们通常把一些公用函数制作成函数库，供其它程序使用。函数库分为静态库和动态库两种。
静态库在程序编译时会被连接到目标代码中，程序运行时将不再需要该静态库。
动态库在程序编译时并不会被连接到目标代码中，而是在程序运行是才被载入，
因此在程序运行时还需要动态库存在。
本文主要通过举例来说明在Linux中如何创建静态库和动态库，以及使用它们。
在创建函数库前，我们先来准备举例用的源程序，并将函数库的源程序编译成.o文件。

第1步：编辑得到举例的程序--hello.h、hello.c和main.c；
hello.c(见程序2)是函数库的源程序，其中包含公用函数hello，
该函数将在屏幕上输出"Hello XXX!"。hello.h(见程序1)为该函数库的头文件。
main.c(见程序3)为测试库文件的主程序，在主程序中调用了公用函数hello。

```
/*hello.h*/
#ifndef HELLO_H
#define HELLO_H

void hello(const char *name);

#endif

/*hello.c*/
#include <stdio.h>
void hello(const char *name)
{
 printf("Hello %s!\n", name);
}

/*main.c*/
#include "hello.h"
int main()
```

```
{
 hello("everyone");
 return 0;
}
```

第2步：将hello.c编译成.o文件；
无论静态库，还是动态库，都是由.o文件创建的。因此，我们必须将源程序hello.c通过gcc先编译成.o文件。
在系统提示符下键入以下命令得到hello.o文件。

```
# gcc -c hello.c
```

下面我们先来看看如何创建静态库，以及使用它.

第3步：由.o文件创建静态库；
静态库文件名的命名规范是以lib为前缀，紧接着跟静态库名，扩展名为.a。
例如：我们将创建的静态库名为myhello，则静态库文件名就是libmyhello.a。
在创建和使用静态库时，需要注意这点。创建静态库用ar命令。

```
# ar cr libmyhello.a hello.o
```

第4步：在程序中使用静态库；

静态库制作完了，如何使用它内部的函数呢？只需要在使用到这些公用函数的源程序中
包含这些公用函数的原型声明，然后在用gcc命令生成目标文件时指明静态库名，
gcc将会从静态库中将公用函数连接到目标文件中。
注意，gcc会在静态库名前加上前缀lib，然后追加扩展名.a得到的静态库文件名来查找静态库文件。

```
# gcc -o hello main.c -L. -lmyhello
# ./hello
Hello everyone!
```

我们删除静态库文件试试公用函数hello是否真的连接到目标文件 hello中了。

```
# rm libmyhello.a
# ./hello
Hello everyone!
```

程序照常运行，静态库中的公用函数已经连接到目标文件中了。

我们继续看看如何在Linux中创建动态库。我们还是从.o文件开始。

第5步：由.o文件创建动态库文件；

动态库文件名命名规范和静态库文件名命名规范类似，也是在动态库名增加前缀lib，
但其文件扩展名为.so。例如：我们将创建的动态库名为myhello，
则动态库文件名就是libmyhello.so。用gcc来创建动态库。

在系统提示符下键入以下命令得到动态库文件libmyhello.so。

```
# gcc -shared -fPCI -o libmyhello.so hello.o
```
或者
```
# gcc -shared -fpic -o libmyhello.so hello.o
```

第6步：在程序中使用动态库；

在程序中使用动态库和使用静态库完全一样，也是在使用到这些公用函数的源程序中
包含这些公用函数的原型声明，然后在用gcc命令生成目标文件时指明动态库名进行编译。
我们先运行gcc命令生成目标文件，再运行它看看结果。

```
# gcc -o hello main.c -L. -lmyhello
# ./hello
./hello: error while loading shared libraries: libmyhello.so:
cannot open shared object file: No such file or directory
```

出错了！快看看错误提示，原来是找不到动态库文件libmyhello.so。
程序在运行时，会在/usr/lib和/lib等目录中查找需要的动态库文件。
若找到，则载入动态库，否则将提示类似上述错误而终止程序运行。
我们将文件libmyhello.so复制到目录/usr/lib中，再试试。

```
# mv libmyhello.so /usr/lib
# ./hello
Hello everyone!
```

成功了。这也进一步说明了动态库在程序运行时是需要的。

我们回过头看看，发现使用静态库和使用动态库编译成目标程序使用的gcc命令完全一样，
那当静态库和动态库同名时，gcc命令会使用哪个库文件呢？

```
# ls
hello.c hello.h hello.o libmyhello.a libmyhello.so main.c
# gcc -o hello main.c -L. -lmyhello
# ./hello
./hello: error while loading shared libraries: libmyhello.so:
cannot open shared object file: No such file or directory
```

从程序hello运行的结果中很容易知道，当静态库和动态库同名时， gcc命令将优先使用动态库。

为了在同一系统中使用不同版本的库，可以在库文件名后加上版本号为后缀，
例如： libhello.so.1.0,由于程序连接默认以.so为文件后缀名。所以为了使用这些库，
通常使用建立符号连接的方式。

```
# gcc -shared -Wl,-soname,libhello.so.1 -o libhello.so.1.0 hello.o
# ln -s libhello.so.1.0 libhello.so.1
# ln -s libhello.so.1 libhello.so
```

-Wl 表示后面的参数也就是-soname,libhello.so.1直接传给连接器ld进行处理。
实际上，每一个库都有一个soname，当连接器发现它正在查找的程序库中有这样一个名称，
连接器便会将soname嵌入连接中的二进制文件内，而不是它正在运行的实际文件名，
在程序执行期间，程序会查找拥有 soname名字的文件，而不是库的文件名，
换句话说，soname是库的区分标志。 这样做的目的主要是允许系统中多个版本的库文件共存，
习惯上在命名库文件的时候通常与soname相同 libxxxx.so.major.minor
其中，xxxx是库的名字，major是主版本号，minor 是次版本号

如果要和多个库连接，而每个库的连接方式不一样，执行如下命令

```
# gcc main.c -o test -Wl,-Bstatic -L. -lhello -Wl,-Bdynamic -L. -lbye
```

动态库的路径问题
为了让执行程序顺利找到动态库，有三种方法:
(1) 把库拷贝到/usr/lib和/lib目录下。

(2) 在LD_LIBRARY_PATH环境变量中加上库所在路径。
例如动态库libhello.so在/home/ting/lib目录下，以bash为例，
使用命令： $export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/jk/lib

(3) 修改/etc/ld.so.conf文件，把库所在的路径加到文件末尾，并执行ldconfig刷新。
这样，加入的目录下的所有库文件都可见。

查看库中的符号
有时候可能需要查看一个库中到底有哪些函数，nm命令可以打印出库中的涉及到的所有符号。
库既可以是静态的也可以是动态的。nm列出的符号有很多，常见的有三种，
一种是在库中被调用，但并没有在库中定义(表明需要其他库支持)，用U表示;
一种是库中定义的函数，用T表示，这是最常见的;
一种是“弱态”符号，它们虽然在库中被定义，但是可能被其他库中的同名符号覆盖，用W表示.

$nm libhello.so

使用ldd命令查看hello依赖于哪些库

$ldd hello

发表于 @ 2007年12月20日 11:30:00 | 评论(0) | 举报 | 收藏

旧一篇:linux下的netlink编程 | 新一篇:linux下的内核模块编程

给dipperkun的留言 只有注册用户才能发表评论! 登录 注册