Linux下动态库和静态库的制作及使用

在实际的开发过程中，编写程序往往都需要依赖很多基础的底层库，比方说平时用的较多的标准C库，数学库等等；我们会频繁的使用这些库里的函数，这些函数大多数都是前人为我们写好的，所以值得庆幸的是我们的工作不必从零开始，我们要做的只是在恰当的位置调用合适的库函数去实现相应的功能，充分利用前人的劳动成果，就是“站在巨人的肩膀上”。本文主要简述Linux下库的制作以及使用方法。

1. **什么是库**

库从本质上来说是一种可执行代码的二进制格式，可以被载入内存中执行。根据链接时期的不同，库又有：静态库和共享库（动态库）二者的不同点在**于代码被载入的时刻不同。**

静态库的代码在**编译过程**中**已经被载入可执行程序**，因此体积较大。

共享库的代码是在可执行程序运行时才载入内存的，在编译过程中仅简单的引用，因此**代码体积较小**。

1. **初识静态库与动态库**

1.静态函数库

这类库的名字一般是libxxx.a，xxx为库的名字。利用静态函数库编译成的文件比较大，因为整个函数库的所有数据都会被整合进目标代码中，他的优点就显而易见了，**即编译后的执行程序不需要外部的函数库支持**，因为所有使用的函数都已经被编译进去了。当然这也会成为他的**缺点**，因为如果静态函数库改变了，那么你的程序必须重新编译。

2.动态函数库

这类库的名字一般是libxxx.M.N.so，同样的xxx为库的名字，M是库的主版本号，N是库的副版本号。当然也可以不要版本号，但名字必须有。相对于静态函数库，动态函数库在编译的时候并没有被编译进目标代码中，你的程序执行到相关函数时才调用该函数库里的相应函数，因此动态函数库所产生的可执行文件比较小。由于函数库没有被整合进你的程序，而是程序运行时动态的申请并调用，所以程序的运行环境中必须提供相应的库。动态函数库的改变并不影响你的程序，所以动态函数库的升级比较方便。linux系统有几个重要的目录存放相应的函数库，如/lib /usr/lib。

1. **静态库与动态库的比较**

静态库其实从某种意义上来说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。

（1）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。

（2）再者，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。

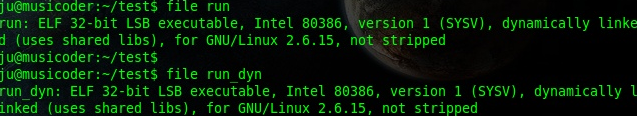
但是静态库也有自己的优点：编译后的执行程序不需要外部的函数库支持，因为所有使用的函数都已经被编译进去了。

|  |  |  |
| --- | --- | --- |
| **对比点** | **静态库** | **动态库** |
| 库命名 | libxxx.a | libxxx.so |
| 库文件格式 | .a文件只是众多.o文件的打包 | .so文件包含支持动态链接和加载的信息。 |
| 制作方法 | 使用ar将.o文件打包为.a | 使用gcc，在汇编阶段制作特殊的.o文件(-fPIC)和在链接阶段基于前期的.o制作特殊的.so文件(-shared)。 |
| 库内容和最终可执行文件的整合关系 | 整个函数库的所有数据都会被整合进最终的可执行文件中，所以最终的a.out较大。 | 函数库没有被整合进你的可执行文件中，最终的a.out较小。函数库的内容以.so的形式独立于a.out存在。而且一个系统中的同一个so文件可能同时被多个a.out所使用，体现出so的共享特性。 |
| 库内的内容何时被**加载**到内存 | 函数库的内容因为已经被整合进最终的可执行文件，所以在可执行文件被加载到内存中执行时，静态链接的函数库的内容也同时被加载。 | 函数库内容什么时候被加载到内存有两种形式：  静态加载：链接时通过-l的方式在a.out文件中记录了其依赖的so的信息列表，并在a.out被启动加载到内存执行时由操作系统负责帮助它找到所有依赖的so并在此时加载到内存中。 |
| 部署的优缺点 | * 优点：因为整个函数库的所有数据都会被整合进目标代码中，他的优点就显而易见了，即编译后的执行程序不需要外部的函数库支持。 * 缺点：因为所有使用的函数都已经被编译进去了。当然这也会成为他的缺点，因为如果静态函数库改变了，那么你的程序必须重新编译和链接。 | * 缺点：由于函数库没有被整合进你的程序，所以如果机器里没有这些库文件就不能运行。 * 优点：动态函数库的改变并不影响你的程序，所以动态函数库的升级/更新比较方便。 |

1. **如何判断一个程序有没有链接动态库**

（1）file命令

file程序是用来判断文件类型的，啥文件一看都清楚明了。



（2）ldd命令

看动态库，如果目标程序没有链接动态库，则打印“not a dynamic executable” （不是动态可执行文件）

1. **静态库的制作**
2. **为pr1和pr2生成object文件**

gcc -O -c pr1.c pr2.c

1. **ls**

**(3) 链接静态库**

为了在编译程序中正确找到库文件,静态库必须按照 lib[name].a 的规则命名,如下例中[name]=pr.

ar参数意义：

*c: create的意思*

*r：在库中插入模块(替换)。当插入的模块名已经在库中存在，则替换同名的模块。*

*s：写入一个目标文件索引到库中，或者更新一个存在的目标文件索引。*

*v：该选项用来显示执行操作选项的附加信息。*

*t：显示库的模块表清单。一般只显示模块名。*

**ar -crsv libpr.a pr1.o pr2.o**

**ar -t libpr.a //显示静态库所依赖的文件**

**(4) 编译链接选项**

-L 及-l 参数放在后面.其中,-L 加载库文件路径,-l 指明库文件名字.

gcc -o main main.c -L./ -lpr //生成main

-I后面接头文件 (大写的i)

-L后面接库文件路径路径

-l后面接库文件名,除了“lib”和“.a”部分,全名为libpr.a

**(5)执行目标程序**

./main

1. **动态库的制作**

**(1)生成动态库 xxx.so**

gcc -fPIC -Wall -c pr1.c pr2.c

PIC告诉编译器产生与位置无关代码(Position-Independent Code)， 则产生的代码中，没有绝对地址，全部使用相对地址，故而代码可以被加载器加载到内存的任意位置，都可以正确的执行。这正是共享库所要求的，共享库被加载时，在内存的位置不是固定的。

gcc -shared -o libpr.so pr1.o pr2.o

or use one line:

gcc -O -fPIC -shared -o libpr.so pr1.c pr2.c

**(2)编译时调用动态库**

gcc -o test main.c –L. -lpr

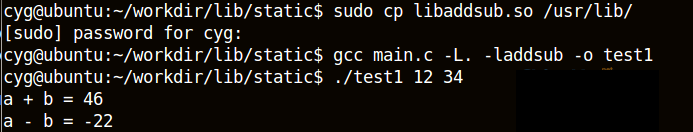
采用该方法执行会报告./test: error while loading shared libraries: libpr.so: cannot open shared object file: No such file or directory

**原因**：因为在动态函数库使用时，会查找/usr/lib、/lib目录下的动态函数库，而此时我们生成的库不在里边。

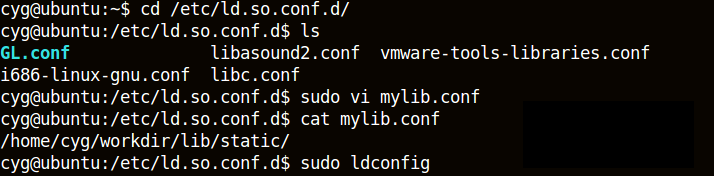
当系统加载可执行代码时候，能够知道其所依赖的库的名字，但是还需要知道绝对路径。它先后搜索elf文件的DT\_RPATH段---环境变量LD\_LIBRARY\_PATH、/etc/ld.so.cache文件列表、/usr/lib、/lib目录找到库文件后将其载入内存。

这个时候有好几种方法可以让他成功运行：

1. 最直接最简单的方法就是把so拉到/usr/lib或/lib中去，但这好像有点污染环境吧。需要root权限，在别人的电脑上会很麻烦；会把系统目录弄得混乱。



1. 新建并编辑/etc/ld.so.conf.d/my.conf文件，加入库所在目录的路径，执行ldconfig命令更新ld.so.cache文件但是需要root权限。

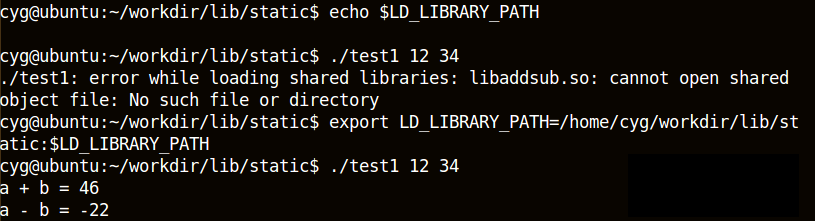


（3）export LD\_LIBRARY\_PATH=/tmp

不过这样export 只对当前shell有效，当另开一个shell时候，又要重新设置。可以把export LD\_LIBRARY\_PATH=/tmp 语句写到 ~/.bashrc中，这样就对当前用户有效了，写到/etc/bash.bashrc中就对所有用户有效了。

echo $LD\_LIBRARY\_PATH

不过LD\_LIBRARY\_PATH的设定作用是全局的，过多的使用可能会影响到其他应用程序的运行，所以多用在调试。



小结：

总而言之，静态库是以空间换时间，动态库是以时间换空间。无论你是在Linux平台还是Windows平台下做开发，库的使用都大同小异。熟练的使各种库，会给我们带来许多便利，减少工作的负担加快工程的进度，从此升职，加薪不是梦，希望对你有所帮助。

案例:

add.c

#include

int add(int a,int b)

{

return a + b;

}

sub.c

#include

int sub(int a,int b)

{

return a - b;

}

head.h

#ifndef \_HEAD\_H\_

#define \_HEAD\_H\_

extern int add(int a,int b);

extern int sub(int a,int b);

#endif

main.c

#include

int main(int argc,char \*argv[])

{

int a,b;

if(argc < 3)

{

fprintf(stderr,"Usage : %s argv[1] argv[2].\n",argv[0]);

return -1;

}

a = atoi(argv[1]);

b = atoi(argv[2]);

printf("a + b = %d\n",add(a,b));

printf("a - b = %d\n",sub(a,b));

return 0;

}