

第 00 章作业 - Linux 知识补充 - 动态链接库的编译与使用

1652817 钟钰琛 计算机科学与技术

2018 年 10 月 10 日

1 linux 下的动态链接库

动态链接库在程序编译时不会被链接到目标代码中, 而是程序运行时才被载入. 不同的应用程序如果调用相同的库, 那么内存里秩序有一份该共享库的实例, 避免了空间浪费. 动态库在程序运行时才被载入, 这样如果更新动态库, 不需要重新编译调用它的源程序.

好处:

- 可以实现进程间资源共享
- 升级动态库更容易
- 节约内存和减少交换操作
- 节省磁盘空间

缺点:

- 运行程序时其依赖的动态库必须存在, 否则无法运行
- 速度比静态链接慢

编译方法: 举个例子, hello.c 有一个函数 hello, printf 一个 hello world, 然后另一个测试文件 main.c 调用函数 hello

```
// hello.c
#include <stdio.h>
void hello()
```

```
{  
    printf("hello, world\n");  
}
```

```
// main.c  
#include <stdio.h>  
void hello();  
int main()  
{  
    printf("call hello()\n");  
    hello();  
    return 0;  
}
```

先给 hello.c 生成动态库 libhello.so

```
gcc -shared -fpic hello.c -o libhello.so
```

然后需要把当前路径放到动态库的默认路径中，有两种办法 1. 直接输入 `ldconfig /root/hello` (当前路径是 /root/hello)

2. 修改 /etc/profile, 把路径添加到 `LD_LIBRARY_PATH`, 然后 `source /etc/profile` 即可.

如果没有这一步，虽然可以顺利编译，但是运行会 runtime error 哈哈. `./hello: symbol lookup error: ./hello: undefined symbol: hello`

最后生成目标文件

```
gcc -o hello main.c -L. -lhello
```

运行结果图见下一页.

2 按要求写出下列几种常用情况的动态链接库的测试样例

2.1 子目录 01

```
192.168.159.22 x
Last login: wed Oct 10 10:40:28 2018 from 192.168.159.101
[root@vm-linux ~]# cd hello/
[root@vm-linux hello]# ls
hello.c libhello.so main.c
[root@vm-linux hello]# gcc -o hello main.c -L. -lhello
[root@vm-linux hello]# ./hello
call hello()
./hello: symbol lookup error: ./hello: undefined symbol: hello
[root@vm-linux hello]# ldconfig /root/hello
[root@vm-linux hello]# gcc -o hello main.c -L. -lhello
[root@vm-linux hello]# ./hello
call hello()
hello,world[root@vm-linux hello]# ^C
[root@vm-linux hello]# ^C
[root@vm-linux hello]#
```

图 1: hello.c 例子

makefile 如下:

```
CC=gcc

test2: test2.c libtest1.so
    $(CC) -o $@ $< -L. -ltest1

libtest1.so: test1.c
    $(CC) -shared -fpic $< -o $@

clean:
    rm test2 *.so
```

make 后, 仍需输入 ldconfig \$(pwd) 后才能运行

```
[root@vm-linux 01]# ls
makefile test1.c test2.c
[root@vm-linux 01]# make
gcc -shared -fpic test1.c -o libtest1.so
gcc -o test2 test2.c -L. -ltest1
[root@vm-linux 01]# ./test2
./test2: error while loading shared libraries: libtest1.so: cannot open shared object file: No such file or directory
[root@vm-linux 01]# ldconfig $(pwd)
[root@vm-linux 01]# ./test2
1652817钟钰琛[root@vm-linux 01]#
```

图 2: 01

如果替换 libtest1.so

```
[root@vm-linux 01]# ll
总用量 32
-rwxr-xr-x 1 root root 8104 10月 10 11:24 libtest1.so
-rw-r--r-- 1 root root 138 10月 10 11:23 makefile
-rw-r--r-- 1 root root 67 10月 10 11:29 test1.c
-rwxr-xr-x 1 root root 8480 10月 10 11:24 test2
-rw-r--r-- 1 root root 83 10月 10 04:59 test2.c
[root@vm-linux 01]# gcc -shared -fpic test1.c -o libtest1.so
[root@vm-linux 01]# ./test2
1659999钟钰琛[root@vm-linux 01]#
```

图 3: 替换 libtest1.so

```
[root@vm-linux 01]# make clean
rm test2 *.so
[root@vm-linux 01]# ll
总用量 12
-rw-r--r-- 1 root root 138 10月 10 11:23 makefile
-rw-r--r-- 1 root root 67 10月 10 11:29 test1.c
-rw-r--r-- 1 root root 83 10月 10 04:59 test2.c
[root@vm-linux 01]#
```

图 4: make clean

2.2 子目录 02

makefile 如下:

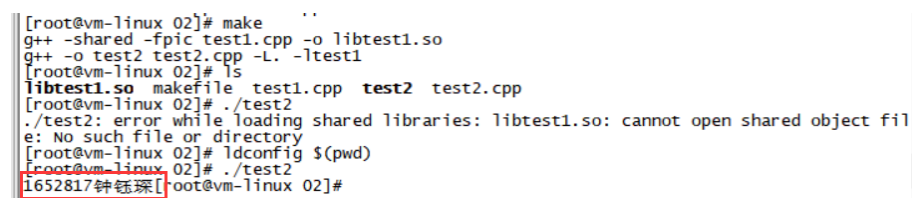
```
CC=g++

test2: test2.cpp libtest1.so
    $(CC) -o $@ $< -L. -ltest1

libtest1.so: test1.cpp
    $(CC) -shared -fpic $< -o $@

clean:
    rm test2 *.so
```

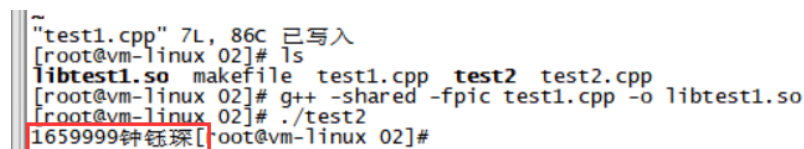
make 后, 仍需输入 `ldconfig $(pwd)` 后才能运行



```
[root@vm-linux 02]# make
g++ -shared -fpic test1.cpp -o libtest1.so
g++ -o test2 test2.cpp -L. -ltest1
[root@vm-linux 02]# ls
libtest1.so  makefile  test1.cpp  test2  test2.cpp
[root@vm-linux 02]# ./test2
./test2: error while loading shared libraries: libtest1.so: cannot open shared object file: No such file or directory
[root@vm-linux 02]# ldconfig $(pwd)
[root@vm-linux 02]# ./test2
1652817钟钰琛[root@vm-linux 02]#
```

图 5: 02

如果替换 libtest1.so



```
~
"test1.cpp" 7L, 86C 已写入
[root@vm-linux 02]# ls
libtest1.so  makefile  test1.cpp  test2  test2.cpp
[root@vm-linux 02]# g++ -shared -fpic test1.cpp -o libtest1.so
[root@vm-linux 02]# ./test2
1659999钟钰琛[root@vm-linux 02]#
```

图 6: 替换 libtest1.so

2.3 总目录下

和上面两个作业的 makefile 一样, 不赘述

```
[root@vm-linux 02]# make clean
rm test2 *.so
[root@vm-linux 02]# ll
总用量 12
-rw-r--r-- 1 root root 142 10月 10 11:34 makefile
-rw-r--r-- 1 root root 86 10月 10 11:36 test1.cpp
-rw-r--r-- 1 root root 105 10月 10 11:32 test2.cpp
[root@vm-linux 02]#
```

图 7: make clean

关于“同名坑”:

看了群里的讨论, 我尝试了几种方法:

1. LD_LIBRARY_PATH=../01 临时加入这种最方便, 也最可行. 我在测试的时候没有遇到问题.

```
[root@vm-linux 01]# LD_LIBRARY_PATH=../01 ./test2
1652817钟钰琛[root@vm-linux 01]# cd ../02
[root@vm-linux 02]# LD_LIBRARY_PATH=../02 ./test2
1652817钟钰琛[root@vm-linux 02]#
```

图 8: LD_LIBRARY_PATH 方法

2. 在/etc/ld.so.conf.d/路径下添加 test.conf, ”/root/1652817-000105/01”、”/root/1652817-000105/02” 在输入 ldconfig 更新列表. 然后再回到/root/1652817-000105 目录下, 执行 make.

然后在 01 目录执行./test2, 没问题. 却在 02 目录执行./test2 出现报错, 这不是没找到 libtest1.so, 而是找到了 01 目录的 libtest1.so. 如果删除

```
[root@vm-linux 02]# ./test2
./test2: symbol lookup error: ./test2: undefined symbol: _Z3funv
[root@vm-linux 02]#
```

图 9: 报错

01 目录的 libtest1.so, 将会报没找到.so 的错误.

仔细想想, 还是因为同名了, 动态链接的时候, 会使用第一个匹配到的动态库. 由于 01 目录的 libtest1.so 在前, 所以在 02 目录执行./test2 的时

候仍匹配到了 01 目录的 libtest1.so.

没办法, 只能换名. 要么就用第一种方法.

```
libdynDwarf.so.9.3 -> libdynDwarf.so.9.3.1
libdynC_API.so.9.3 -> libdynC_API.so.9.3.1
libcommon.so.9.3 -> libcommon.so.9.3.1
/usr/lib64/mysql:
libmysqlclient.so.18 -> libmysqlclient_r.so
/root/1652817-000105/01:
libtest1.so -> libtest1.so
/root/1652817-000105/02:
libtest1.so -> libtest1.so
/lib:
libhello.so -> libhello.so
/lib64:
libzip.so.2 -> libzip.so.2.1.0
libwbclient.so.0 -> libwbclient.so.0.14
libn2c.so.0 -> libn2c.so.0.0.0
```

图 10: ldconfig -v 的结果; 动态链接时, 会使用第一个匹配到的库