Linux 动态链接库

关于动态链接库的使用技巧

屈庆磊

quqinglei@icloud.com

2013 年 *8* 月 *14* 日

目录

1 动态链接库百科

动态链接库文件，是一种不可执行的二进制程序文件，它允许程序共享执行特殊任务所必需的代码和其他资源。Windows 提供的 DLL 文件中包含了允许基于 Windows 的程序在 Windows 环境下操作的许多函数和资源。一般被存放在 C:/Windows/System 目录下。Windows 中，DLL 多数情况下是带有 DLL 扩展名的文件，但也可能是 EXE 或其他扩展名；Debian 系统中常常是.so 的文件。它们向运行于 Windows 操作系统下的程序提供代码、数据或函数。程序可根据 DLL 文件中的指令打开、启用、查询、禁用和关闭驱动程序。

2 动态链接库搜索路径

在 *Linux* 中，动态库的搜索路径除了默认的搜索路径外，还可以通过以下三种方式来指定，这个在实现上并没有什么难度，只是一种规定而已。下面讲述三种方式。

2.1 示例代码

```
// dym.c
#include <stdio.h>
void test()
{
        printf("Hello world!\n");
}

// main.c
#include <stdio.h>
void test();
int main()
{
        test();
        return 0;
}

// Makefile
all: a.out

a.out: main.o libdym.so
        gcc -o a.out main.o -L. -ldym

# 一定要命名为 libxxx.so 否则在使用 -L. -ldym 时会找不到
# 因为默认情况下动态链接库的开头是 lib
libdym.so: dym.o
        gcc -shared -fPIC -o libdym.so dym.o

main.o: main.c
        gcc -c main.c

dym.o: dym.c
        gcc -c dym.c

clean:
        rm -f *.o *.so a.out
```

按照上面的代码和 makefile 编译后生成 a.out 和 libdym.so 文件此时运行 a.out 必然会有如下提示:

./a.out: error while loading shared libraries: libdym.so: cannot open shared object file: No such file or directory

2.2 全局法

在上一小节中我们编译了一个程序叫：*a.out*，但无法执行，原因很简单，就是找不到动态链接库，使用 *ldd*[1] *a.out* 可以看到如下提示：

[1] ldd 是用来打印程序对动态链接库依赖的小工具，你可以使用 man ldd 查看手册。

```
linux-gate.so.1 =>  (0xb77c5000)
libdym.so => not found # 找不到此文件
libc.so.6 => /lib/i386-linux-gnu/i686/cmov/libc.so.6 (0xb7655000)
/lib/ld-linux.so.2 (0xb77c6000)
```

我们把动态链接库 libdym.so 的路径导出来写到：/etc/ld.so.conf 里，比如我们现在的路径是：/home/test/hello/ 那么可以在/etc/ld.so.conf 文件里写上如下所示：

```
include /etc/ld.so.conf.d/*.conf # 默认的，我们也可以写到/etc/ld.so.conf.d/hello.conf
# hello.conf 是我随便起的名字，看上一行有一个 include，它会把/etc/ld.so.conf.d/路径里的
# 所有.conf 文件包含，所以写到此路径命名为 xxx.conf 的文件也可以达到同样的目的
/home/test/hello/
```

写完以后在 root 用户下执行 ldconfig，此时我们可以执行: ldd a.out，我们可以得到正常的显示结果:

```
linux-gate.so.1 =>  (0xb7703000)
libdym.so => /home/git/papers/dym/aaaa/libdym.so (0xb76f4000)
libc.so.6 => /lib/i386-linux-gnu/i686/cmov/libc.so.6 (0xb7591000)
/lib/ld-linux.so.2 (0xb7704000)
```

运行 a.out 会得到正常的显示结果，如下：

```
root@debian:/home/test# ./a.out
Hello world!
```

2.3 导出环境变量法

有些时候你作为普通用户没有 root 的权限，在全局法使用上收到限制，此时你可以考虑使用导出全局变量法，如下所示：

```
root@debian:/home/test# export LD_LIBRARY_PATH="/lib:/user/lib:/home/test/hello"
root@debian:/home/test# ./a.out
Hello world!
```

你完全可以把环境变量写到一个脚本中，把可执行程序也写到脚本里执行，在导出环境变量后执行即可！如下：

```
#!/bin/bash
export LD_LIBRARY_PATH="/usr/lib:/lib:/home/test/hello"
./a.out
```

2.4 编译时确定程序的动态库搜索路径

这种方法是可行的，但我不建议这样玩，环境一变就乱套了！不过还是讲一下吧。

```
// 更改后的 Makefile
all: a.out

a.out: main.o libdym.so
        gcc -o a.out main.o -L. -ldym -Wl,-rpath,./

libdym.so: dym.o
        gcc -shared -fPIC -o libdym.so dym.o

main.o: main.c
        gcc -c main.c

dym.o: dym.c
        gcc -c dym.c

clean:
        rm -f *.o *.so a.out
```

以上例子可以直接执行，不需要再添加环境变量或者更改系统配置文件了

3 程序对动态链接库搜索的顺序

我们可以完全自己写代码测出来程序的寻找先后顺序，比如你在每一个路径都写一个不同输出内容的同名函数，然后编译成动态链接库，执行代码测试。关于此例我就不多言了。

(1) 最先搜索的是程序编译时指定的动态链接库路径

(2) 然后是环境变量 LD_LIBRARY_PATH 指定的搜索路径

(3) 然后是 ld.so.conf 文件包含的路径

(4) 如果前三个路径都没有，程序会去`/lib` 路径查找

(5) 最后会去`/usr/lib` 路径去查找