

同济大学计算机系

计算机网络课程实验报告



学 号 1552239

姓 名 岳昊玮

专 业 计算机科学与技术

授课老师 沈坚

一、Linux 下的动态链接库

动态链接库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 平台上动态链接库的后缀名是“.dll”，Linux 平台上的后缀名是“.so”。Linux 上动态库一般是 libxxx.so；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。

静态库的原则是“以空间换时间”，增加程序体积，减少运行时间；动态库则是“以时间换空间”，增加了运行时间，但减少了程序本身的体积，带来的好处是节省内存空间。还具有以下优点：

(1)运行时占用较少的硬件资源

当多个程序使用同一个函数库时，动态链接库的使用可以减少在磁盘和物理内存中加载的代码的重复量。在运行时，只有当有 EXE 程序确实要调用该 DLL 模块的情况下，系统才会将它们装载到内存空间中。

(2)有助于模块化体系结构开发

动态链接库有助于促进模块式程序的开发，支持多语言程序。这样可以帮助我们开发要求提供多个语言版本的大型程序或要求具有模块式体系结构的程序。对于一个大型的系统，如果用一个执行文件完成，程序将会很庞大，而且还可能有许多重复的功能。如果将程序分成一系列的主程序和 DLL，可以减少开发的工作量和提高开发的速度。

(3)修改升级软件方便

当动态链接库中的函数需要更新或修复时，部署和安装动态链接库不要求重新建立程序与该 DLL 的连接。此外，如果多个程序使用同一个 DLL，那么多个程序都将从该更新或修复中获益。当您使用定期更新或修复的第三方 DLL 时，此问题可能会更频繁地出现，升级到 DLL 更为容易解决这些问题。

(4)隐藏实现细节

在某些情况下，我们在做程序时可能想隐藏例程实现的一些细节，动态链接库就是一项非常不错的实现方法。动态链接库的例程可以被应用程序访问，而不显示其中的代码细节。还有很重要的一点就是 .so 与语言的无关性，可以用许多种编程语言来编写。有助于解决平台差异和应用程序的本地化

在 LINUX 系统下，创建动态链接库是件非常简单的事情。只要在编译函数库源程序时加上 -shared 选项即可，这样所生成的执行程序即为动态链接库。从某种意义上来说，动态链接库也是一种执行程序。按一般规则，动态库的命名应为 lib*.so.*。其中第一个*为动态库的名字，第二个*为版本号，比如 libc.so.6。

动态链接库是目标文件的集合，目标文件在动态链接库中的组织方式是按照特殊方式形成的，库中函数和变量的地址是相对的，不是绝对地址，其真实地址在调用动态库的程序加载时形成的。

安装库文件就是把库文件放在系统目录下，能让程序找见就可以。系统默认的目录是 /lib 和 /usr/lib 或者还有其他的。也可以不放在系统默认路径下，自己在链接库配置文件中配置一个路径。

链接库的配置文件，即配置程序运行时查找动态库的路径，配置文件是 /etc/ld.so.conf。系统文件 /lib 和 /usr/lib 是默认的查找的目录，不用配置。其余的自己添加的路径可以配置进去。

用命令 `gcc -fPIC -shared hello.c -o libhello.so` 编译为动态库。

gcc 一些参数解析

-shared: 指定生成动态链接库。

-static: 指定生成静态链接库。

-fPIC: 表示编译为位置独立的代码,用于编译共享库。目标文件需要创建成位置无关码,概念上就是在可执行程序装载它们的时候,它们可以放在可执行程序的内存里的任何地方。不用此选项的话编译后的代码是位置相关的所以动态加载时是通过代码拷贝的方式来满足不同进程的须要,而不能达到真正代码段共享的目的。

-L: 表示要连接的库在当前目录中。

-l: 指定链接时需要的动态库。编译器查找动态连接库时有隐含的命名规则,即在给出的名字前面加上 `lib`,后面加上 `.so` 来确定库的名称。

-Wall: 生成所有警告信息。

-ggdb: 此选项将尽可能的生成 `gdb` 的可以使用的调试信息。

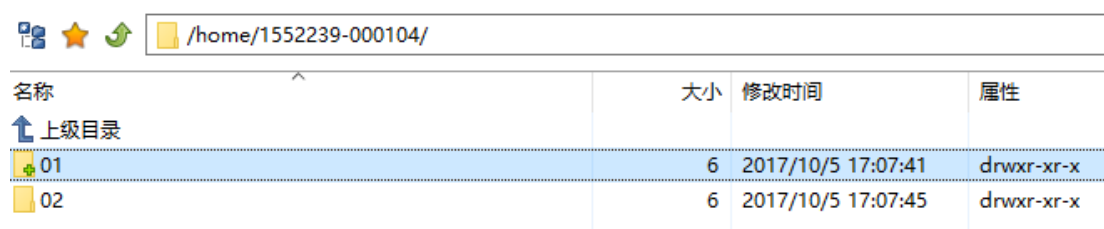
-g: 编译器在编译的时候产生调试信息。

-c: 只激活预处理、编译和汇编,也就是把程序做成目标文件(.o 文件)。

-Wl,options: 把参数(options)传递给链接器 `ld`。如果 options 中间有逗号,就将 options 分成多个选项,然后传递给链接程序。

二、 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 建立空目录



名称	大小	修改时间	属性
上级目录			
01	6	2017/10/5 17:07:41	drwxr-xr-x
02	6	2017/10/5 17:07:45	drwxr-xr-x

2.2 在子目录 01 下建立两个源程序文件 test1.c/test2.c(分别打印自己的学号及姓名), 并写出满足要求的 makefile 文件

因为刚开始写 `makefile` 的时候是把生成动态链接库*.so 和生成可执行文件分成了两条命令在写, 所以这部分内容就出现在了报告中, 下面我会再贴上一条 `make` 命令执行两个操作的。

唯一需要 `makefile` 改动的就是 `build: libtest1.so test2`

下面是 `makefile` 内容

```
.PHONY: build test2 clean
CC := gcc
target := test2

build: libtest1.so test2

# link parameter
LIB := libtest1.so

$(LIB): test1.o
    $(CC) -o $@ -shared $<
# -fPIC : 表示编译为位置独立的代码，用于编译共享库。
test1.o: test1.c
    $(CC) -c -fPIC $<

$(target): test2.out
# -L : 表示要连接的库在当前目录中 -l : 指定链接时需要的动态库
test2.out: test2.c ./$(LIB)
    $(CC) test2.c -L. -ltest1 -o test2
# LD_LIBRARY_PATH 告诉test2先在当前路径下寻找lib
    LD_LIBRARY_PATH=. ./test2
clean:
    rm -f *.o *.so $(target)
```

两条命令的

在 01 目录下编写好 test1.c/test2.c/makefile 之后执行

```
[root@Anokoro 01]# make
gcc -c -fPIC test1.c
gcc -o libtest1.so -shared test1.o
[root@Anokoro 01]# make test2
gcc test2.c -L. -ltest1 -o test2
LD_LIBRARY_PATH=. ./test2
1552239-岳昊玮
[root@Anokoro 01]# ./test2
./test2: error while loading shared libraries: libtest1.so: cannot open shared object file: No such file or directory
[root@Anokoro 01]# ^C
[root@Anokoro 01]# LD_LIBRARY_PATH=. ./test2
1552239-岳昊玮
[root@Anokoro 01]#
```

然后修改 test1.c 的内容，重新 make，得到一个新的 libtest1.so，但是不重新 make test2，相当于替换了仅仅 libtest1.so:

```
[root@Anokoro 01]# vim test1.c
//test1.c
#include <stdio.h>
int fun(){
    printf("1234567-"); //打印你的学号
    return 0;
}
~
```

LD_LIBRARY_PATH:该环境变量主要用于指定查找共享库（动态链接库）时除了默认路径之外的其它路径。这里指定为当前路径。

```
[root@Anokoro 01]# make
gcc -c -fPIC test1.c 仅执行make，重新生成libtest1.so
gcc -o libtest1.so -shared test1.o
[root@Anokoro 01]# LD_LIBRARY_PATH=. ./test2
1234567-岳昊玮
[root@Anokoro 01]#
```

执行 make clean 后清除掉除了.c 和 makefile 以外的所有文件

```
[root@Anokoro 01]# make clean
rm -f *.o *.so test2
```

一条 make 执行所有操作：

```
[root@Anokoro 01]# make clean
rm -f *.o *.so test2
[root@Anokoro 01]# make
gcc -c -fPIC test1.c
gcc -o libtest1.so -shared test1.o
gcc test2.c -L. -ltest1 -o test2
LD_LIBRARY_PATH=. ./test2
1552239-岳昊玮 _
```

2.3 在子目录 02 下建立两个源程序文件 test1.cpp/test2.cpp(分别打印自己的学号及姓名)，并写出满足要求的 makefile 文件

```
.PHONY: build test2 clean
CC := g++
target := test2

build: libtest1.so test2

# link parameter
LIB := libtest1.so

$(LIB): test1.o
    $(CC) -o $@ -shared $<

test1.o: test1.cpp
    $(CC) -c -fPIC $<

$(target): test2.out
# -L : 表示要连接的库在当前目录中 -l : 指定链接时需要的动态库
test2.out: test2.cpp ./$(LIB)
    $(CC) test2.cpp -L. -ltest1 -o test2
# LD_LIBRARY_PATH 告诉test2先在当前路径下寻找lib
LD_LIBRARY_PATH=. ./test2
clean:
    rm -f *.o *.so $(target)
```

两条命令的

```
[root@Anokoro 02]# mv test1.c test1.cpp
[root@Anokoro 02]# mv test2.c test2.cpp
[root@Anokoro 02]# make
g++ -c -fPIC test1.cpp
g++ -o libtest1.so -shared test1.o
[root@Anokoro 02]# make test2
g++ test2.cpp -L. -ltest1 -o test2
LD_LIBRARY_PATH=. ./test2
1552239-岳昊玮
[root@Anokoro 02]# LD_LIBRARY_PATH=. ./test2
1552239-岳昊玮
[root@Anokoro 02]#
```

这里其实为了偷懒，我是从01里面cp的，修改了文件里面必要的内容和文件名

如上图，编译器用的是 g++，而且没有报错，说明文件里面的内容也符合 c++ 的语法规则。

接下来和上题一样，修改 test1.cpp 里面的学号，重新 make，模拟替换掉.so 文件

```

[root@Anokoro 02]# vim test1.cpp
//test1.cpp
#include <iostream>
using namespace std;
int fun(){
    cout<<"1234567-"; //打印你的学号
    return 0;
}
~

"test1.cpp" 7L, 110C 已写入
[root@Anokoro 02]# make
g++ -c -fPIC test1.cpp
g++ -o libtest1.so -shared test1.o
[root@Anokoro 02]# LD_LIBRARY_PATH=. ./test2
1234567-岳昊玮
[root@Anokoro 02]#

```

只重新生成了.so，
没有重新生成test2

libtest1.so	8 KB	2017/10/5 19:19:26	-rwxr-xr-x
makefile	512	2017/10/5 11:11:55	-rw-r--r--
*+ test1.cpp	110	2017/10/5 19:19:04	-rw-r--r--
test1.o	2 KB	2017/10/5 19:19:26	-rw-r--r--
test2	8 KB	2017/10/5 19:13:40	-rwxr-xr-x
*+ test2.cpp	130	2017/10/5 19:04:13	-rw-r--r--

执行 make clean，清除掉 cpp 和 makefile 以外的所有文件

```

[root@Anokoro 02]# make clean
rm -f *.o *.so test2

```

一条 make 执行所有操作

```

[root@Anokoro 02]# make clean
rm -f *.o *.so test2
[root@Anokoro 02]# make
g++ -c -fPIC test1.cpp
g++ -o libtest1.so -shared test1.o
g++ test2.cpp -L. -ltest1 -o test2
LD_LIBRARY_PATH=. ./test2
1552239-岳昊玮
[root@Anokoro 02]#

```

2.4 在 1551234-000104 目录下写一个满足下列要求的 makefile 文件

```

# 需要排除的目录
exclude_dirs := include bin
# 取得当年子目录深度为1的所有目录名称
dirs := $(shell find . -maxdepth 1 -type d)
echo:$(dirs)
    echo $(dirs)
# basename 命令用于去掉路径信息，返回纯粹的文件名，如果指定的文件有扩展名，则将扩展名也一并去掉。
dirs := $(basename $(patsubst ./%,%,$(dirs)))
# filter-out 反过滤函数 和“filter”函数实现的功能相反。过滤掉字符串“TEXT”中所有符合模式
dirs := $(filter-out $(exclude_dirs),$(dirs))

SUBDIRS := $(dirs)
# addprefix 添加前缀_clean_，避免clean子目录操作同名，
clean_dirs := $(addprefix _clean_,$(SUBDIRS) )
prom := test2

.PHONY: subdirs $(SUBDIRS) clean

# 执行默认make target
$(SUBDIRS):
    $(MAKE) -C $@
subdirs: $(SUBDIRS)
# 执行clean
$(clean_dirs):
    $(MAKE) -C $(patsubst _clean_%,%,$@) clean
clean: $(clean_dirs)

```

```

[root@Anokoro 02]# cd ..
[root@Anokoro 1552239-000104]# make clean
make -C 01 clean
make[1]: 进入目录"/home/1552239-000104/01"
rm -f *.o *.so test2
make[1]: 离开目录"/home/1552239-000104/01"
make -C 02 clean
make[1]: 进入目录"/home/1552239-000104/02"
rm -f *.o *.so test2
make[1]: 离开目录"/home/1552239-000104/02"
[root@Anokoro 1552239-000104]# make
make -C 01
make[1]: 进入目录"/home/1552239-000104/01"
gcc -c -fPIC test1.c
gcc -o libtest1.so -shared test1.o
gcc test2.c -L. -ltest1 -o test2
LD_LIBRARY_PATH=. ./test2
1552239-岳昊玮
make[1]: 离开目录"/home/1552239-000104/01"
make -C 02
make[1]: 进入目录"/home/1552239-000104/02"
g++ -c -fPIC test1.cpp
g++ -o libtest1.so -shared test1.o
g++ test2.cpp -L. -ltest1 -o test2
LD_LIBRARY_PATH=. ./test2
1552239-岳昊玮
make[1]: 离开目录"/home/1552239-000104/02"
echo 01 02
01 02
[root@Anokoro 1552239-000104]#

```

这里有输出是因为我在子makefile里面加了执行生成的可执行文件的语句。