同济大学计算机网络
实验报告

姓名：　涂远鹏-1652262　
题目：　程序的静态及动态编译　

1.Linux 下的动态编译

(1)什么叫动态编译?

答：动态编译是某些程式语言在执行时用来增进效能的方法。尽管这技术源于Self[来源请求]，但使用此技术最为人所知的是 Java。此技术可以做到一些只在执行时才能完成的最佳化。使用动态编译的执行环境一开始执行速度较慢，之后，完成大部分的编译和再编译后，会执行得比非动态编译程式快很多。因为初始化时的效能延迟，动态编译不适用于一些情况。在许多实作中，一些可以在编译时期做的最佳化被延到执行时期才编译，导致不必要的效能降低。即时编译是一种动态编译的形式。

动态编译的可执行文件需要附带一个的动态链接库，在执行时，需要调用其对应动态链接库中的命令。所以其优点一方面是缩小了执行文件本身的体积，另一方面是加快了编译速度，节省了系统资源。缺点一是哪怕是很简单的程序，只用到了链接库中的一两条命令，也需要附带一个相对庞大的链接库；二是如果其他计算机上没有安装对应的运行库，则用动态编译的可执行文件就不能运行。

(2)给出 printf("hello，world");程序的 gcc 动态编译命令，可执行文件字节数是多少?

答：字节数为8368

```
[root@RHEL74-SVR test]# gcc -c -fpic hello.c
[root@RHEL74-SVR test]# gcc -shared -fpic -o libhello.so hello.o
[root@RHEL74-SVR test]# mv libhello.so /usr/lib
[root@RHEL74-SVR test]# gcc -o hello -L. -lhello
[root@RHEL74-SVR test]# ll
总用量 20
-rwxr-xr-x. 1 root root 8368 9月  17 20:13 hello
-rw-r--r--. 1 root root  105 9月  17 20:12 hello.c
-rw-r--r--. 1 root root 1568 9月  17 20:12 hello.o
[root@RHEL74-SVR test]#
```

(3)给出 cout << "hello，world";程序的 c++/g++动态编译命令，可执行文件字节数是多少?

答：字节数为8888

```
[root@RHEL74-SVR test]# g++ -cpp -fpic hello2.cpp
[root@RHEL74-SVR test]# c++ hello2.cpp -o hello2
[root@RHEL74-SVR test]# ll
```

```
[root@RHEL74-SVR test]# ll
总用量 72
-rwxr-xr-x. 1 root root 8880 9月  18 20:56 a.out
-rwxr-xr-x. 1 root root 8368 9月  18 20:38 hello
-rwxr-xr-x. 1 root root 8888 9月  18 20:56 hello2
-rw-r--r--. 1 root root   88 9月  18 20:49 hello2.cpp
-rw-r--r--. 1 root root 2544 9月  18 20:52 hello2.o
-rw-r--r--. 1 root root  140 9月  18 20:38 hello.c
-rw-r--r--. 1 root root 1608 9月  18 20:38 hello.o
-rwxr-xr-x. 1 root root 8544 9月  18 20:50 libhello2.so
-rwxr-xr-x. 1 root root 8112 9月  18 20:38 libhello.so
[root@RHEL74-SVR test]#
```

(4)给出第一周作业中 mysql_demo.cpp 的动态编译命令，可执行文件字节数是多少?

```
总用量 32
-rw-------. 1 root root  1412 9月  15 10:55 anaconda-ks.cfg
-rwxr-xr-x. 1 root root 14200 9月  16 10:49 a.out
```

(5)如何查找某个可执行文件所依赖的动态链接库?

答：可以通过 ldd 命令查看一个可执行程序依赖的共享库，如下图所示，通过 ldd 指令查看 libhello.so 所依赖的共享库：

```
[root@RHEL74-SVR test]# ldd libhello.so
        linux-vdso.so.1 =>  (0x00007ffd4d528000)
        libstdc++.so.6 => /lib64/libstdc++.so.6 (0x00007f689bb7e000)
        libm.so.6 => /lib64/libm.so.6 (0x00007f689b87c000)
        libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007f689b666000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f689b299000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f689c088000)
[root@RHEL74-SVR test]#
```

CSDN 上的详细解释:

```
(1) $ ldd /bin/grep
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/libc.so.6 (0xb7eca000)
        /lib/ld-linux.so.2 (0xb801e000)

(2) $ LD_TRACE_LOADED_OBJECTS=1 /bin/grep
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/libc.so.6 (0xb7e30000)
        /lib/ld-linux.so.2 (0xb7f84000)

(3) $ LD_TRACE_LOADED_OBJECTS=1 /lib/ld-linux.so.2 /bin/grep
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/libc.so.6 (0xb7f7c000)
        /lib/ld-linux.so.2 (0xb80d0000)
```

第(1)个命令，我们运行了`ldd` 于 `/bin/grep`。我们可以看到命令的输出是我们想要的，那就是 `/bin/grep` 所依赖的动态链接库。

第(2)个命令设置了一个叫LD_TRACE_LOADED_OBJECTS 的环境变量，然后就好像在运行命令 `/bin/grep` (但其实并不是)。 其运行结果和ldd的输出是一样的！

第(3)个命令也是设置了环境变量LD_TRACE_LOADED_OBJECTS，然后调用了动态链接库 `ld-linux.so` 并把`/bin/grep` 作为参数传给它。我们发现，其输出结果还是和前面两个一样的。

2.Linux 下的 gcc 静态编译

(1)什么叫静态编译?

答：通常情况下，对函数库的链接是放在编译时期（compile time）完成的。所有相关的对象文件（object file）与牵涉到的函数库（library）被链接合成一个可执行文件 （executable file）。程序 在运行 时，与函数库再无瓜葛，因为所有需要的函数已拷贝到自己门下。所以这些函数库被成为静态库（static libaray），通常文件 名为“libxxx.a”的形式。

(2)给出 printf("hello，world");程序的 gcc 静态编译命令，可执行文件字节数是多少?

答:

首先安装静态编译所需的包 glibc-static-2.17-222.el7.x86_64.rpm(从官网上下即可)

```
[root@RHEL74-SVR home]# rpm -i glibc-static-2.17-222.el7.x86_64.rpm
[root@RHEL74-SVR home]#
```

Gcc 编译源程序生成的可执行文件的字节数为 861072 字节:

```
[root@RHEL74-SVR test]# gcc hello.c -static -o hello
[root@RHEL74-SVR test]# ll
总用量 864
-rwxr-xr-x. 1 root root   8888 9月  18 21:05 a.out
-rwxr-xr-x. 1 root root 861072 9月  30 17:08 hello
-rw-r--r--. 1 root root     88 9月  18 20:49 hello2.cpp
-rw-r--r--. 1 root root     99 9月  18 21:03 hello.c
[root@RHEL74-SVR test]#
```

3. Linux 下的 c++/g++静态编译

答:

首先安装 stdc++对应的包:

```
Last login: Sun Sep 30 17:14:14 2018
[root@RHEL74-SVR ~]# yum install libstdc++-static
已加载插件：fastestmirror, langpacks
Repository 'c7-media': Error parsing config: Error parsing "baseurl = 'file:///medi
a/CentOS/\nfile:///media/cdrom/\nfile:///media/cdrecorder/\n# file:///home/CentOS-7
-x86_64-DVD-1804/'": URL must be http, ftp, file or https not ""
Repository base is listed more than once in the configuration
Loading mirror speeds from cached hostfile
```

然后就是编译过程生成的可执行文件字节数为 1608232，比之前的大多了:

```
[root@RHEL74-SVR ~]# cd /home/test
[root@RHEL74-SVR test]# g++ hello2.cpp -static -o hello2-s
[root@RHEL74-SVR test]# ll
总用量 2436
-rwxr-xr-x. 1 root root    8888 9月  18 21:05 a.out
-rwxr-xr-x. 1 root root  861072 9月  30 17:08 hello
-rw-r--r--. 1 root root      88 9月  18 20:49 hello2.cpp
-rwxr-xr-x. 1 root root 1608232 9月  30 17:16 hello2-s
-rw-r--r--. 1 root root      99 9月  18 21:03 hello.c
[root@RHEL74-SVR test]#
```

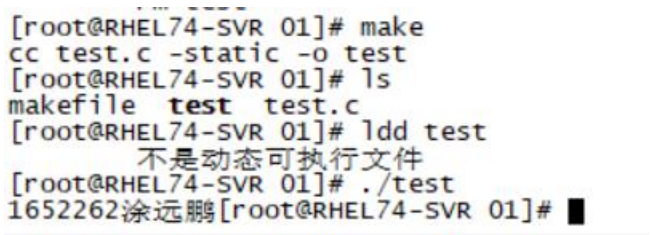
4. 按要求写出下列几种常用情况的静态编译测试样例

4.1 新建 1652262-000104 文件夹并新建对应的源程序：

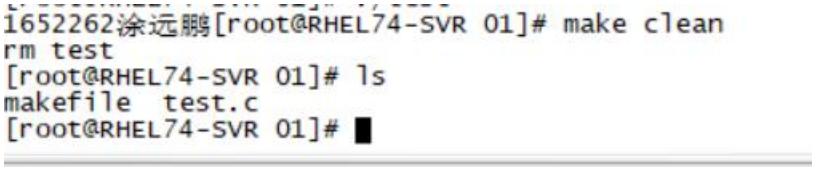
4.2 在子目录 01 下建立 test.c，并写出满足要求的 makefile 文件：

(1)编写对应的 makefile 内容显示如下：

(2)测试 make 指令效果，显示结果正确：



(3)测试 make clean 指令效果，显示结果正确：



4.3 **在子目录 02 下建立 test.cpp，并写出满足要求的 makefile 文件：**

(1)创建文件夹并编写对应的 makefile 内容显示如下：

(2)测试 make 指令效果，显示结果正确：

```
[root@RHEL74-SVR 02]# make
g++ test.cpp -static -o test
[root@RHEL74-SVR 02]# ls
makefile  test  test.cpp
[root@RHEL74-SVR 02]# ./test
1652262涂远鹏[root@RHEL74-SVR 02]# ldd test
        不是动态可执行文件
[root@RHEL74-SVR 02]#
```

(3)测试 make clean 指令效果，显示结果正确：

```
[root@RHEL74-SVR 02]# make clean
rm test
[root@RHEL74-SVR 02]# ls
makefile  test.cpp
[root@RHEL74-SVR 02]#
```

4.4 **在 1652262-000104 目录下写一个满足要求的 makefile 文件：**

(1)makefile 内容显示如下：

```
[root@RHEL74-SVR test]# cd ..
[root@RHEL74-SVR home]# vi makefile
SUB = `ls -d */`

subdirs:
        for dir in $(SUB); do \
$(MAKE) -C $$dir; \
done

clean:
        for dir in $(SUB); do \
$(MAKE) clean -C $$dir; \
done
```

(2)测试 make 指令效果，显示结果正确：

```
[root@RHEL74-SVR 1652262-000104]# make
for dir in `ls -d */`; do \
make -C $dir; \
done
make[1]: 进入目录“/home/16-000104/1652262-000104/01”
cc test.c -static -o test
make[1]: 离开目录“/home/16-000104/1652262-000104/01”
make[1]: 进入目录“/home/16-000104/1652262-000104/02”
g++ test.cpp -static -o test
make[1]: 离开目录“/home/16-000104/1652262-000104/02”
[root@RHEL74-SVR 1652262-000104]# cd 01
[root@RHEL74-SVR 01]# ll
总用量 852
-rw-r--r--. 1 root root     51 10月  4 10:36 makefile
-rwxr-xr-x. 1 root root 861072 10月  4 10:45 test
-rw-r--r--. 1 root root     98 10月  4 09:58 test.c
[root@RHEL74-SVR 01]# cd ..
[root@RHEL74-SVR 1652262-000104]# cd 02
[root@RHEL74-SVR 02]# ll
总用量 1580
-rw-r--r--. 1 root root      52 10月  4 10:38 makefile
-rwxr-xr-x. 1 root root 1608232 10月  4 10:45 test
-rw-r--r--. 1 root root     122 10月  4 09:59 test.cpp
[root@RHEL74-SVR 02]# ./test
1652262涂远鹏[root@RHEL74-SVR 02]# cd ..
[root@RHEL74-SVR 1652262-000104]# cd 01
[root@RHEL74-SVR 01]# ./test
1652262涂远鹏[root@RHEL74-SVR 01]#
```

(3)测试 make clean 指令效果，显示结果正确:

```
[root@RHEL74-SVR 1652262-000104]# make clean
for dir in `ls -d */`; do \
make clean -C $dir; \
done
make[1]: 进入目录“/home/16-000104/1652262-000104/01”
rm test
make[1]: 离开目录“/home/16-000104/1652262-000104/01”
make[1]: 进入目录“/home/16-000104/1652262-000104/02”
rm test
make[1]: 离开目录“/home/16-000104/1652262-000104/02”
[root@RHEL74-SVR 1652262-000104]# cd 01
[root@RHEL74-SVR 01]# ll
总用量 8
-rw-r--r--. 1 root root 51 10月  4 10:36 makefile
-rw-r--r--. 1 root root 98 10月  4 09:58 test.c
[root@RHEL74-SVR 01]# cd ..
[root@RHEL74-SVR 1652262-000104]# cd 02
[root@RHEL74-SVR 02]# ll
总用量 8
-rw-r--r--. 1 root root  52 10月  4 10:38 makefile
-rw-r--r--. 1 root root 122 10月  4 09:59 test.cpp
[root@RHEL74-SVR 02]#
```