# 关于动态链接库的分析

在linux系统中动态链接库文件用.so后缀标记，一般命名规则为libxxx.so。

## 1 链接产生动态库.so与编译源码产生的二进制文件.o的关系

现在有工程，源文件包括：

|  |
| --- |
| main1.cpp  myAPI.cpp  myAPI.h |

其中myAPI.cpp，myAPI.h定义了两个函数ADD(), MINUS()；

main1.cpp中则调用ADD(), MINUS()

### <实验1>

1)编译产生myHandler.o

|  |
| --- |
| g++ -fPIC -c -o myAPI.o myAPI.cpp |

2)用myAPI.o文件编译链接产生main1

|  |
| --- |
| g++ main1.cpp myAPI.o -o main1 |

3)编译、链接、运行成功，然后用ldd main1查看可执行文件的依赖：

|  |
| --- |
| linux-vdso.so.1 => (0x00007fffed77c000)  libstdc++.so.6 => /usr/lib/x86\_64-linux-gnu/libstdc++.so.6 (0x00007f21fc26b000)  libc.so.6 => /lib/x86\_64-linux-gnu/libc.so.6 (0x00007f21fbea1000)  libm.so.6 => /lib/x86\_64-linux-gnu/libm.so.6 (0x00007f21fbb98000)  /lib64/ld-linux-x86-64.so.2 (0x00007f21fc5ed000)  libgcc\_s.so.1 => /lib/x86\_64-linux-gnu/libgcc\_s.so.1 (0x00007f21fb982000) |

4)删除myAPI.o，再运行./main1。

结果程序正常运行。

分析：这说明，main文件中包含有myAPI.o myHandler.o中的所有可执行代码，除了因为main.cpp中包含了<iostream>库所以需要在运行时动态加载一些.so文件外，没有依赖任何其他库文件。

### <实验2>

1)编译产生myHandler.o

|  |
| --- |
| g++ -fPIC -c -o myAPI.o myAPI.cpp |

2)用myAPI.o文件打包产生libmyAPI.so

|  |
| --- |
| g++ -shared myAPI.o -o libmyAPI.so |

1. 用libmyAPI.so和main1.cpp编译链接产生main2

|  |
| --- |
| g++ -o main2 main1.cpp -L. -lmyAPI |

3)编译、链接、运行成功，然后用ldd main2查看可执行文件的依赖：

|  |
| --- |
| linux-vdso.so.1 => (0x00007fffed77c000)  libmyAPI.so (0x00007fe029706000)  libstdc++.so.6 => /usr/lib/x86\_64-linux-gnu/libstdc++.so.6 (0x00007f21fc26b000)  libc.so.6 => /lib/x86\_64-linux-gnu/libc.so.6 (0x00007f21fbea1000)  libm.so.6 => /lib/x86\_64-linux-gnu/libm.so.6 (0x00007f21fbb98000)  /lib64/ld-linux-x86-64.so.2 (0x00007f21fc5ed000)  libgcc\_s.so.1 => /lib/x86\_64-linux-gnu/libgcc\_s.so.1 (0x00007f21fb982000) |

4)删除myAPI.o，再运行./main2，程序正常运行

5)删除myAPI.so，再运行./main2，程序运行失败

|  |
| --- |
| ./main2: error while loading shared libraries: libmyAPI.so: cannot open shared object file: No such file or directory |

分析：在4)删除myAPI.o后程序运行成功，说明./main2的运行与myAPI.o无关。在4)删除myAPI.so后执行程序提示加载动态库失败，恢复myAPI.so后随机成功运行，直接说明main2依赖myAPI.so文件。原因是main2中缺少某部分执行代码，这部分代码包含在libmyAPI.so中，而main2记录这部分代码的来源，所以当使用ldd main2查看其依赖列表时有libmyAPI.so。

根据<实验1>及<实验2>，可以得出结论，.so文件中完全包含了.o中多有可执行代码。另外.o文件不是可执行文件，而.so是可执行文件。.so文件允许程序一开始不加载他所包含的代码，当需要运行它包含的代码时，才根据文件中的路径动态家去加载它的代码。

注：

在编译链接main2时，遇到一个有意思的小麻烦：

能够成功编译链接的指令为：g++ -o main2 main1.cpp -L. -lmyAPI

但如果交换源文件和动态链接库的顺序：g++ -o main2 -L. -lmyAPI main1.cpp

g++会报错：

|  |
| --- |
| /tmp/ccvwLNKz.o：在函数‘main’中：  main1.cpp:(.text+0x14)：对‘ADD(int, int)’未定义的引用  main1.cpp:(.text+0x4b)：对‘MINUS(int, int)’未定义的引用  collect2: error: ld returned 1 exit status |

原因不明，这可能与g++组织组织文件依赖关，总之记住以后让源文件在前，依赖在后。

## 2 .so文件的路径问题

现在有工程，源文件包括：

|  |
| --- |
| main1.cpp  myAPI.cpp  myAPI.h |

其中myAPI.cpp，myAPI.h定义了两个函数ADD(), MINUS()；

main1.cpp中则调用ADD(), MINUS()

### <实验3>

1. 首先产生libmyAPI.so，这一次将动态库放入./libs目录下

2)编译链接产生main3

|  |
| --- |
| g++ -o main3 main1.cpp -L. -L./libs -lmyAPI |

1. 成功产生目标，然而运行时，系统报错：

|  |
| --- |
| ./main3: error while loading shared libraries: libmyAPI.so: cannot open shared object file: No such file or directory |

1. ldd main3查看依赖列表

|  |
| --- |
| ...  libmyAPI.so => not found  ...#其他都正常，省略 |

发现main3的依赖列表中缺省libmyAPI.so的路径，

恩？？？步骤2)中明明给定了libmyAPI.so的找寻路径-L./libs，为什么依赖列表中会没有？更奇怪的是当我们将libmyAPI.so拷贝到main3所在的目录下时，又运行成功，这又是怎么回事？

分析：

首先排除g++在链接时不会去验证libmyAPI.so是否真实存在这种猜想，因为如果libmyAPI.so不存在，2)中的编译指令不会通过，直接提示缺少库文件。

实际上，原因在于连接器ld的工作模式，注意到2)中指定动态库的参数-lmyAPI而不是直接给其名称。在执行这条编译指令时，链接器ld（此处有疑问：在执行g++指令时，到时会不会是用ld指令？）会自动将-lmyAPI扩展为libmyAPI.so然后根据参数中的库文件查找路径-L.、-L./libs去寻找libmyAPI.so，可能编译器会验证下libmyAPI.so是否包含目标的依赖（包括直接依赖和间接依赖），如果包含则将参数指定的路径写入依赖表，不包含的直接忽略（此处有疑问，也可能连基本的验证一下libmyAPI.so是否包含源文件中依赖的代码都不会做，但从<实验5>结果看更有可能会验证），当遇到依赖文件不能完全满足编译目标的需求时，编译器不会报错，只有在需要链接时才会报错。又因为编译时依赖的对象是动态库，不会将libmyAPI.so中的代码全部打包到main3中，简单记录libmyAPI.so文件的路径，以便以后运行时动态去加载。然而，执行编译指令时，用户指定的查找路径-L.、-L./libs，应该只是用来验证，不会将这个查询路径连同文件名一起填写到main3的依赖列表中，此时只会填写为libmyAPI.so，要想加上路径，用户必须在编译参数中写上动态库的全称（当然也可以是绝对路径，但是这样做不利于程序移植）如：

|  |
| --- |
| g++ -o main3 main1.cpp ./libs/libmyAPI.so |

此时再去查看main3的依赖列表：

|  |
| --- |
| ...  ./libs/libmyAPI.so (0x00007f07463ce000)  ... |

在真正运行main3时，ld会首先查找其配置目录/etc/ld.so.conf.d/下的若\*.conf文件（可以是到某个.conf文件的链接），文件中配置了ld的默认查询路径，若默认路径下目录下有无libmyAPI.so文件，再直接查询依赖列表中的路径是否有效。

## 3 .so文件间依赖的问题

现在有工程，源文件包括：

|  |
| --- |
| main2.cpp  myAPI.cpp  myAPI.h  myHandler.cpp  myHandler.h |

其中myAPI.cpp，myAPI.h定义了两个函数ADD(), MINUS()；

myHandler.cpp,，myHandler.h中也定义了两个函数mutil\_ADD(), mutil\_MINUS()，而两个函数分别调用了ADD(), MINUS()函数；

main.cpp中则调用mutil\_ADD(), mutil\_MINUS()

### <实验4>

1)链接产生.so文件

|  |
| --- |
| g++ myAPI.o -shared -o libs/libmyAPI.so  g++ myHandler.o -shared -o libs/libmyHandler.so |

2)用.so文件编译链接产生main4

|  |
| --- |
| g++ main2.cpp libs/libmyAPI.so libs/libmyHandler.so -o main4 |

结果链接失败：

|  |
| --- |
| ./libs/libmyHandler.so：对‘ADD(int, int)’未定义的引用  ./libs/libmyHandler.so：对‘MINUS(int, int)’未定义的引用  collect2: error: ld returned 1 exit status |

1. 产看libmyHandler.so的依赖列表ldd ./libs/libmyHandler.so

|  |
| --- |
| statically linked |

表明，libmyHandler.so中没有依赖任何动态库。

然而，myHandler.cpp中不是调用了myAPI.cpp定义的函数么，为什么编译指令

g++ myHandler.o -shared -o libs/libmyHandler.so可以通过编译呢？

原因是这种情况下，编译器最多验证下指定的依赖文件（包括.so或.a）是否存在，不会进一步验证依赖文件是否覆盖源文件的需求，只有当需要连接时才会验证依赖是否有效。

分析：由于myHandler.cpp文件使用了myAPI.cpp中定义的函数，然而在编译链接libmyHandler.so时既没有声明需要链接myAPI.o文件也没有链接libmyAPI.so，所以现在的libmyHandler.so中的ADD()，MINUS()属于未定义的引用。

### <实验5>

1)链接产生.so文件

|  |
| --- |
| g++ -fPIC -c -o libs/myAPI.o myAPI.cpp  g++ -fPIC -c -o libs/myHandler.o myHandler.cpp  g++ libs/myAPI.o -shared -o libs/libmyAPI.so  g++ libs/myHandler.o libs/libmyAPI.so -shared -o libs/libmyHandler.so |

2)用.so文件编译链接产生main5

|  |
| --- |
| g++ main2.cpp libs/libmyAPI.so libs/libmyHandler.so -o main5  # g++ main2.cpp libs/libmyHandler.so -o main5 |

3)编译、运行成功，产看main5依赖表

|  |
| --- |
| ...  libs/libmyHandler.so (0x00007f1c5030e000)  libs/libmyAPI.so (0x00007f1aea515000)  ... |

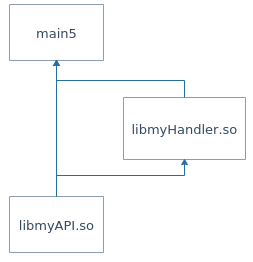
？？？为啥2)中明明声明了libs/libmyAPI.so，libs/libmyHandler.so两个依赖，为啥依赖列表中只有libs/libmyHandler.so

可能编译器会验证下libmyAPI.so是否包含目标的依赖（包括直接依赖和间接依赖），如果包含则将参数指定的路径写入依赖表，不包含的直接忽略。

1. 产看ldd libs/libmyHandler.so 依赖表

|  |
| --- |
| ...  libs/libmyAPI.so (0x00007f69ac934000)  ... |

此时的依赖层次图：



## 4同库新旧版同引用问题

考虑一种特殊情况，要在同一个程序中链接同一个库的新旧两个版本。

### <实验6>

目录结构如下：

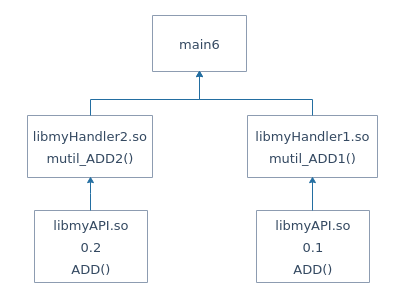
|  |
| --- |
| libs2/  myAPI.cpp  myAPI.h  libs1/  myAPI.cpp  myAPI.h  myHandler1.cpp  myHandler1.h  myHandler2.cpp  myHandler2.h  main4.cpp |

其中libs2/myAPI.cpp与libs1/myAPI.cpp中都定义有命名为的ADD()和MINUS()函数（参数列表，返回值，函数名均相同，区别是函数打印的信息不同），而myHandler2.cpp和myHandler1.cpp则分别调用llibs2/myAPI.cpp与libs1/myAPI.cpp中的同名函数。main4.cpp中又调用myHandler.cpp和myHandler1.cpp中定义的函数。

Makefile如下：

|  |
| --- |
| step4:  g++ -fPIC -c -o libs2/myAPI.o libs2/myAPI.cpp  g++ -fPIC -c -o libs1/myAPI.o libs1/myAPI.cpp  g++ -fPIC -c -o libs2/myHandler2.o myHandler2.cpp  g++ -fPIC -c -o libs1/myHandler1.o myHandler1.cpp  g++ libs2/myAPI.o -shared -o libs2/libmyAPI.so  g++ libs1/myAPI.o -shared -o libs1/libmyAPI.so  g++ libs2/myHandler2.o libs2/libmyAPI.so -shared -o libs2/libmyHandler2.so  g++ libs1/myHandler1.o libs1/libmyAPI.so -shared -o libs1/libmyHandler1.so  g++ main4.cpp libs1/libmyHandler1.so libs/libmyHandler.so -o main6 |

理论上依赖关系应该如下图：



ldd main6查看main6关系列表：

|  |
| --- |
| ...  libs1/libmyHandler1.so (0x00007f61c7837000)  libs2/libmyHandler2.so (0x00007f61c7635000)  libs1/libmyAPI.so (0x00007f61c6ce7000)  libs2/libmyAPI.so (0x00007f61c6ae5000)  ... |

哇塞，貌似很科学的样子。然后运行发现，完全不正确：

|  |
| --- |
| libs1: ADD #正确的话应该打印libs2: ADD  libs1: ADD #正确的话应该打印libs2: ADD  mutil\_ADD2(1,2,3) = 6  libs1: MINUS #正确的话应该打印libs2: MINUS  libs1: MINUS #正确的话应该打印libs2: MINUS  mutil\_MINUS2(1,2,3) = -4  libs1: ADD  libs1: ADD  mutil\_ADD1(1,2,3) = 6  libs1: MINUS  libs1: MINUS  mutil\_MINUS1(1,2,3) = -4 |

当注释掉main4.cpp调用myHandler1.cpp的代码后，编译运行：

|  |
| --- |
| libs2: ADD  libs2: ADD  mutil\_ADD2(1,2,3) = 6  libs2: MINUS  libs2: MINUS  mutil\_MINUS2(1,2,3) = -4 |

结果完全正确！！！

接着，注释掉调用myHandler2.cpp的代码后，编译运行，

|  |
| --- |
| libs1: ADD  libs1: ADD  mutil\_ADD1(1,2,3) = 6  libs1: MINUS  libs1: MINUS  mutil\_MINUS1(1,2,3) = -4 |

结果还是完全正确！！！

为啥，单独编译就能得到正确结果，一起调用就不行呢？？？

猜测可能和libs1/libmyAPI.so，libs2/libmyAPI.so文件名重复有关，接下来再进行第2种情况的实验。

### <实验7>

不修改任何代码，仅修改Makefile

|  |
| --- |
| step4:  g++ -fPIC -c -o libs2/myAPI.o libs2/myAPI.cpp  g++ -fPIC -c -o libs1/myAPI.o libs1/myAPI.cpp  g++ -fPIC -c -o libs2/myHandler2.o myHandler2.cpp  g++ -fPIC -c -o libs1/myHandler1.o myHandler1.cpp  g++ libs2/myAPI.o -shared -o libs2/libmyAPI2.so  g++ libs1/myAPI.o -shared -o libs1/libmyAPI1.so  g++ libs2/myHandler2.o libs2/libmyAPI2.so -shared -o libs2/libmyHandler2.so  g++ libs1/myHandler1.o libs1/libmyAPI1.so -shared -o libs1/libmyHandler1.so  g++ main4.cpp libs1/libmyHandler1.so libs2/libmyHandler2.so -o main6 |

编译时让.so文件名不同，ldd main6查看main6关系列表：

|  |
| --- |
| ...  libs1/libmyHandler1.so (0x00007f61c7837000)  libs2/libmyHandler2.so (0x00007f61c7635000)  libs1/libmyAPI1.so (0x00007f61c6ce7000)  libs2/libmyAPI2.so (0x00007f61c6ae5000)  ... |

依然很有道理的样子，然后编译，运行

|  |
| --- |
| libs1: ADD  libs1: ADD  mutil\_ADD2(1,2,3) = 6  libs1: MINUS  libs1: MINUS  mutil\_MINUS2(1,2,3) = -4  libs1: ADD  libs1: ADD  mutil\_ADD1(1,2,3) = 6  libs1: MINUS  libs1: MINUS  mutil\_MINUS1(1,2,3) = -4 |

结果还是不对！！！

修改.so文件名行不通，那么修改.cpp和.h文件名呢？于是有情况3

### <实验8>

目录结构如下：

|  |
| --- |
| libs2/  myAPI.cpp  myAPI.h  libs1/  myAPI1.cpp  myAPI1.h  myHandler1\_1.cpp  myHandler1\_1.h  myHandler2.cpp  myHandler2.h  main5.cpp |

其中libs2/myAPI.cpp与libs1/myAPI1.cpp中都定义有命名为的ADD()和MINUS()函数（参数列表，返回值，函数名均相同，区别是函数打印的信息不同），而myHandler2.cpp和myHandler1\_1.cpp则分别调用llibs2/myAPI.cpp与libs1/myAPI1.cpp中的同名函数。main4.cpp中又调用myHandler.cpp和myHandler1\_1.cpp中定义的函数。

Makefile：

|  |
| --- |
| step4:  g++ -fPIC -c -o libs2/myAPI.o libs2/myAPI.cpp  g++ -fPIC -c -o libs1/myAPI1.o libs1/myAPI1.cpp  g++ -fPIC -c -o libs2/myHandler2.o myHandler2.cpp  g++ -fPIC -c -o libs1/myHandler1\_1.o myHandler1\_1.cpp  g++ libs2/myAPI.o -shared -o libs2/libmyAPI2.so  g++ libs1/myAPI1.o -shared -o libs1/libmyAPI1.so  g++ libs2/myHandler2.o libs2/libmyAPI2.so -shared -o libs2/libmyHandler2.so  g++ libs1/myHandler1\_1.o libs1/libmyAPI1.so -shared -o libs1/libmyHandler1\_1.so  g++ main5.cpp libs1/libmyHandler1\_1.so libs2/libmyHandler2.so -o main7.out |

编译，运行，结果还是不对，好吧，

看来问题就是，函数重名！！！

### <实验9>

最后，修改Makefile，企图将所有代码均打包给main6

|  |
| --- |
| step4:  g++ -fPIC -c -o libs2/myAPI.o libs2/myAPI.cpp  g++ -fPIC -c -o libs1/myAPI1.o libs1/myAPI1.cpp  g++ -fPIC -c -o libs2/myHandler2.o myHandler2.cpp  g++ -fPIC -c -o libs1/myHandler1\_1.o myHandler1\_1.cpp  g++ main4.cpp libs2/myAPI.o libs1/myAPI1.o libs2/myHandler2.o libs1/myHandler1\_1.o -o main7.out |

结果，编译报错：

|  |
| --- |
| g++ -fPIC -c -o libs2/myAPI.o libs2/myAPI.cpp  g++ -fPIC -c -o libs1/myAPI1.o libs1/myAPI1.cpp  g++ -fPIC -c -o libs2/myHandler2.o myHandler2.cpp  g++ -fPIC -c -o libs1/myHandler1\_1.o myHandler1\_1.cpp  g++ main4.cpp libs2/myAPI.o libs1/myAPI1.o libs2/myHandler2.o libs1/myHandler1\_1.o -o main7  libs1/myAPI1.o：在函数‘ADD(int, int)’中：  myAPI1.cpp:(.text+0x0): `ADD(int, int)'被多次定义  libs2/myAPI.o:myAPI.cpp:(.text+0x0)：第一次在此定义  libs1/myAPI1.o：在函数‘MINUS(int, int)’中：  myAPI1.cpp:(.text+0x43): `MINUS(int, int)'被多次定义  libs2/myAPI.o:myAPI.cpp:(.text+0x43)：第一次在此定义  collect2: error: ld returned 1 exit status  Makefile:2: recipe for target 'step4' failed  make: \*\*\* [step4] Error 1 |

分析：果然，函数多次被定义，是不行的！！！即使c++支持重载，但是也要求参数列表不能相同，事实证明在同一个程序中不能同时引用相同库的不同版本，主要就是因为函数重复定义