Linux下gcc编译生成动态链接库*.so文件并调用它

原创 flyztek 2017-06-23 00:29:35 32692 版权

Linux下gcc编译生成动态链接库*.so文件并调用它

动态库*.so在linux下用c和c++编程时经常会碰到，最近在网站找了几篇文章介绍动态库的编译和链接，总算搞懂了这个之前一直不太了解得东东，这里做个笔记，也为其它正为动态库链接库而苦恼的兄弟们提供一点帮助。

1、动态库的编译

下面通过一个例子来介绍如何生成一个动态库。这里有一个头文件：so_test.h，三个.c文件：test_a.c、test_b.c、test_c.c，我们将这几个文件编译成一个动态库：libtest.so。

//so_test.h：

```
#include "stdio.h"
void test_a();
void test_b();
void test_c();
```

//test_a.c：

```
#include "so_test.h"
void test_a()
{
 printf("this is in test_a...\n");
}
```

//test_b.c：

```
#include "so_test.h"
void test_b()
{
 printf("this is in test_b...\n");
}
```

//test_c.c：

```
#include "so_test.h"
void test_c()
{
 printf("this is in test_c...\n");
}
```

将这几个文件编译成一个动态库：libtest.so

```
$ gcc test_a.c test_b.c test_c.c -fPIC -shared -o libtest.so
```

2、动态库的链接

在1、中，我们已经成功生

个程序来调用这个库里的函

test.c：

```
#include "so_test.h"
int main()
{
test_a();
test_b();
test_c();
return 0;
}
```

将test.c与动态库libtest.so链接生成执行文件test：

```
$ gcc test.c -L. -ltest -o test
```

测试是否动态连接，如果列出libtest.so，那么应该是连接正常了

```
$ ldd test
```

执行test，可以看到它是如何调用动态库中的函数的。

3、编译参数解析

最主要的是GCC命令行的一个选项:

-shared该选项指定生成动态连接库（让连接器生成T类型的导出符号表，有时候也生成弱连接W类型的导出符号），不用该标志外部程序无法连接。相当于一个可执行文件

-fPIC：表示编译为位置独立的代码，不用此选项的话编译后的代码是位置相关的所以动态载入时是通过代码拷贝的方式来满足不同进程的需要，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

-ltest：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

LD_LIBRARY_PATH：这个环境变量指示动态连接器可以装载动态库的路径。

当然如果有**root**权限的话，可以修改**/etc/ld.so.conf**文件，然后调用**/sbin/ldconfig**来达到同样的目的，不过如果没有**root**权限，那么只能采用输出**LD_LIBRARY_PATH**的方法了。

4、注意

调用动态库的时候有几个问题会经常碰到，有时，明明已经将库的头文件所在目录 通过 **"-I" include**进来了，库所在文件通过 **"-L"**参数引导，并指定了**"-l"**的库名，但通过**ldd**命令察看时，就是死活找不到你指定链接的**so**文件，这时你要作的就是通过修改 **LD_LIBRARY_PATH**或者**/etc/ld.so.conf**文件来指定动态库的目录。通常这样做就可以解决库无法链接的问题了。

在linux下可以用export命令来设置这个值，在linux终端下输入:

```
export LD_LIBRARY_PATH=/opt/au1200_rm/build_tools/bin:
$LD_LIBRARY_PATH:
```

然后再输入:export

即会显示是否设置正确

export方式在重启后失效，所以也可以用 vim /etc/bashrc ，修改其中的LD_LIBRARY_PATH变量。

例如：

LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/au1200_rm/build_tools/bin。

Linux下静态库，动态库，以及arm平台下库的基本概念

一、基本概念

1.1、什么是库

在 windows 平台和 Linux 平台下都大量存在着库。

本质上来说库是 一种可执行代码的二进制形式，可以被操作系统载入内存执行。

由于 windows 和 linux 的平台不同（主要是编译器、汇编器和连接器 的不同），因此二者库的二进制是不兼容的。

本文仅限于介绍 linux 下的库。

1.2、 库的种类

linux 下的库有两种：静态库和共享库（动态库）。

二者的不同点在于代码被载入的时刻不同。

静态库的代码在编译过程中已经被载入可执行程序，因此体积较大。

静态用.a为后缀， 例如： libhello.a

共享库(动态库)的代码是在可执行程序运行时才载入内存的，在编译过程中仅简单的引用，因此代码体积较小。

动态通常用.so为后缀， 例如：libhello.so

共享库(动态库)的好处是，不同的应用程序如果调用相同的库，那么在内存里只需要有一份该共享库的实例。

为了在同一系统中使用不同版本的库，可以在库文件名后加上版本号为后缀,例如： libhello.so.1.0,由于程序连接默认以.so为文件后缀名。所以为了使用这些库，通常使用建立符号连接的方式。

ln -s libhello.so.1.0 libhello.so.1 ln -s libhello.so.1 libhello.so

1.3、静态库，动态库文件在linux下是如何生成的：

以下面的代码为例，生成上面用到的hello库：

```
/* hello.c */

#include "hello.h"

void sayhello()

{

printf("hello,world ");

}
```

首先用gcc编绎该文件，在编绎时可以使用任何合法的编绎参数，例如-g加入调试代码等：

```
$gcc -c hello.c -o hello.o
```

1、生成静态库 生成静态库使用ar工具，其实ar是archive的意思

```
$ar cqs libhello.a hello.o
```

2、生成动态库 用gcc来完成，由于可能存在多个版本，因此通常指定版本号：

$gcc -shared -o libhello.so.1.0 hello.o

1.4、库文件是如何命名的，有没有什么规范：

在 linux 下，库文件一般放在/usr/lib和/lib下，

静态库的名字一般为libxxxx.a，其中 xxxx 是该lib的名称；

动态库的名字一般为libxxxx.so.major.minor，xxxx 是该lib的名称，major是主版本号，minor是副版本号

1.5、可执行程序在执行的时候如何定位共享库(动态库)文件 ：

当系统加载可执行代码(即库文件)的时候，能够知道其所依赖的库的名字，但是还需要知道绝对路径，此时就需要系统动态载入器(dynamic linker/loader)

对于 elf 格式的可执行程序，是由 ld-linux.so* 来完成的，它先后搜索 elf 文件的 DT_RPATH 段—环境变量LD_LIBRARY_PATH—/etc/ld.so.cache 文件列表— /lib/,/usr/lib 目录找到库文件后将其载入内存

如： export LD_LIBRARY_PATH='pwd'

将当前文件目录添加为共享目录

1.6、使用ldd工具，查看可执行程序依赖那些动态库或着动态库依赖于那些动态库：

ldd 命令可以查看一个可执行程序依赖的共享库，

例如 # ldd /bin/lnlibc.so.6

=> /lib/libc.so.6 (0×40021000)/lib/ld-linux.so.2

=> /lib/ld- linux.so.2 (0×40000000)

可以看到 ln 命令依赖于 libc 库和 ld-linux 库

使用以下的命令查看arm平台的依赖关系

注意：arm-linux-readelf -d busybox | grep Shared grep后面可以不要,注意大小Shared第一个字母大写

1.7、使用nm工具，查看静态库和动态库中有那些函数名（T类表示函数是当前库中定义的，U类表示函数是被调用的，在其它库中定义的，W类是当前库中定义，被其它库中的函数覆盖）。：

有时候可能需要查看一个库中到底有哪些函数，nm工具可以打印出库中的涉及到的所有符号，这里的库既可以是静态的也可以是动态的。

nm列出的符号有很多，

一种是在库中被调用，但并没有在库中定义(表明需要其他库支持)，用U表示；

一种是在库中定义的函数，用T表示，这是最常见的；

另外一种是所 谓的"弱态"符号，它们虽然在库中被定义，但是可能被其他库中的同名符号覆盖，用W表示。

例如，假设开发者希望知道上文提到的hello库中是否引用了printf()：

```
$nm libhello.so | grep printf
```

发现printf是U类符号，说明printf被引用，但是并没有在库中定义。

由此可以推断，要正常使用hello库，必须有其它库支持，使用ldd工具查看hello依赖于哪些库：

```
$ldd hello libc.so.6=>/lib/libc.so.6(0x400la000) /lib/ld-linux.so.2=>/lib/ld-linux.so.2 (0x40000000)
```

从上面的结果可以继续查看printf最终在哪里被定义，有兴趣可以Go on

1.8、使用ar工具，可以生成静态库，同时可以查看静态库中包含那些.o文件，即有那些源文件构成。

可以使用 ar -t libname.a 来查看一个静态库由那些.o文件构成。

可以使用 ar q libname.a xxx1.o xxx2.o xxx3.o ... xxxn.o 生成静态库

Linux下进行程序设计时，关于库的使用：

一、gcc/g++命令中关于库的参数：

-shared： 该选项指定生成动态连接库（让连接器生成T类型的导出符号表，有时候也生成弱连接W类型的导出符号），不用该标志外部程序无法连接。相当于一个可执行文件

-fPIC：表示编译为位置独立(地址无关)的代码，不用此选项的话，编译后的代码是位置相关的，所以动态载入时，是通过代码拷贝的方式来满足不同进程的需要，而不能达到真正代码段共享的目的。

-L：指定链接库的路径，-L. 表示要连接的库在当前目录中

-ltest：指定链接库的名称为test，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

LD_LIBRARY_PATH：这个环境变量指示动态连接器可以装载动态库的路径。

当然如果有root权限的话，可以修改/etc/ld.so.conf文件，然后调用 /sbin/ldconfig

不过如果没有root权限，那么只能采用修改LD_LIBRARY_PATH环境变量的方法了。

调用动态库的时候，有几个问题会经常碰到：

1、有时，明明已经将库的头文件所在目录 通过 "-I" include进来了，库所在文件通过 "-L"参数引导，并指定了"-l"的库名，但通过ldd命令察看时，就是死活找不到你指定链接的so文件，这时你要作的就是通过修改 LD_LIBRARY_PATH或者/etc/ld.so.conf文件来指定动态库的目录。通常这样做就可以解决库无法链接的问题了。

二、静态库链接时搜索路径的顺序：

1. ld会去找gcc/g++命令中的参数-L；

2. 再找gcc的环境变量LIBRARY_PATH,它指定程序静态链接库文件搜索路径；

```
export LIBRARY_PATH=$LIBRARY_PATH:data/home/billchen/lib
```

3. 再找默认库目录 /lib /usr/lib /usr/local/lib，这是当初compile gcc时写在程序内的。

三、动态链接时、执行时搜索路径顺序：

1. 编译目标代码时指定的动态库搜索路径；

2. 环境变量LD_LIBRARY_PATH指定动态库搜索路径，它指定程序动态链接库文件搜索路径；

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:data/home/billchen/lib
```

3. 配置文件/etc/ld.so.conf中指定的动态库搜索路径；

4. 默认的动态库搜索路径/lib；

5. 默认的动态库搜索路径/usr/lib。

四、静态库和动态链接库同时存在的问题：

当一个库同时存在静态库和动态库时，比如libmysqlclient.a和libmysqlclient.so同时存在时：

在Linux下，动态库和静态库同事存在时，gcc/g++的链接程序，默认链接的动态库。

可以使用下面的方法，给连接器传递参数，看是否链接动态库还是静态库。

```
-WI,-Bstatic -llibname //指定让gcc/g++链接静态库
```

使用：

```
gcc/g++ test.c -o test -WI,-Bstatic -llibname
```

```
-WI,-Bdynamic -llibname //指定让gcc/g++链接动态库
```

使用：

```
gcc/g++ test.c -o test -WI,-Bdynamic -llibname
```

如果要完全静态加在，使用-static参数，即将所有的库以静态的方式链入可执行程序，这样生成的可执行程序，不再依赖任何库，同事出现的问题是，这样编译出来的程序非常大，占用空间。

五、有关环境变量：

LIBRARY_PATH环境变量：指定程序静态链接库文件搜索路径

LD_LIBRARY_PATH环境变量：指定程序动态链接库文件搜索路径

六、动态库升级问题：

在动态链接库升级时，

不能使用cp newlib.so oldlib.so，这样有可能会使程序core掉；

而应该使用:

rm oldlib.so 然后 cp newlib.so oldlib.so

或者

mv oldlib.so oldlib.so_bak 然后 cp newlib.so oldlib.so

为什么不能用cp newlib.so oldlib.so ？

在替换so文件时，如果在不停程序的情况下，直接用 cp new.so old.so 的方式替换程序使用的动态库文件会导致正在运行中的程序崩溃。

解决方法:

解决的办法是采用"rm＋cp" 或"mv＋cp" 来替代直接"cp" 的操作方法。

linux系统的动态库有两种使用方法：运行时动态链接库，动态加载库并在程序控制之下使用。

1、为什么在不停程序的情况下，直接用 cp 命令替换程序使用的 so 文件，会使程序崩溃？

很多同学在工作中遇到过这样一个问题，在替换 so 文件时，如果在不停程序的情况下，直接用cp new.so old.so的方式替换程序使用的动态库文件会导致正在运行中的程序崩溃，退出。

这与 cp 命令的实现有关，cp 并不改变目标文件的 inode，cp 的目标文件会继承被覆盖文件的属性而非源文件。实际上它是这样实现的：

strace cp libnew.so libold.so 2>&1 |grep open.*lib.*.so

open("libnew.so", O_RDONLY|O_LARGEFILE) = 3

open("libold.so", O_WRONLY|O_TRUNC|O_LARGEFILE) = 4

在 cp 使用"O_WRONLY|O_TRUNC" 打开目标文件时，原 so 文件的镜像被意外的破坏了。这样动态链接器 ld.so 不能访问到 so 文件中的函数入口。从而导致 Segmentation fault，程序崩溃。

ld.so 加载 so 文件[illegible]

2、怎样在不停止程序的情况下替换so文件，并且保证程序不会崩溃？

答案是采用"rm＋cp" 或"mv＋cp" 来替代直接"cp" 的操作方法。

在用新的so文件 libnew.so 替换旧的so文件 libold.so 时，如果采用如下方法：

rm libold.so //如果内核正在使用libold.so，那么inode节点不会立刻别删除掉。

cp libnew.so libold.so

采用这种方法，目标文件 libold.so 的 inode 其实已经改变了，原来的 libold.so 文件虽然不能用 "ls"查看到，但其 inode 并没有被真正删除，直到内核释放对它的引用。

（即：rm libold.so，此时，如果ld.so正在加在libold.so，内核就在引用libold.so的inode节点，rm libold.so的inode并没有被真正删除，当ld.so对libold.so的引用结束，inode才会真正删除。这样程序就不会崩溃，因为它还在使用旧的libold.so，当下次再使用libold.so时，已经被替换，就会使用新的libold.so）

同理，mv只是改变了文件名，其 inode 不变，新文件使用了新的 inode。这样动态链接器 ld.so 仍然使用原来文件的 inode 访问旧的 so 文件。因而程序依然能正常运行。

(即：mv libold.so ***后，如果程序使用动态库，还是使用旧的inode节点，当下次再使用libold.so时，就会使用新的libold.so)

到这里，为什么直接使用"cp new_exec_file old_exec_file"这样的命令时，系统会禁止这样的操作，并且给出这样的提示"cp: cannot create regular file `old': Text file busy"。这时，我们采用的办法仍然是用"rm+cp"或者"mv+cp"来替代直接"cp"，这跟以上提到的so文件的替换有同样的道理。

但是，为什么系统会阻止 cp 覆盖可执行程序，而不阻止覆盖 so 文件呢？

这是因为 Linux 有个 Demand Paging 机制，所谓"Demand Paging"，简单的说，就是系统为了节约物理内存开销，并不会程序运行时就将所有页（page）都加载到内存中，而只有在系统有访问需求时才将其加载。"Demand Paging"要求正在运行中的程序镜像（注意，并非文件本身）不被意外修改，因此内核在启动程序后会锁定这个程序镜像的 inode。

对于 so 文件，它是靠 ld.so 加载的，而ld.so毕竟也是用户态程序，没有权利去锁定inode，也不应与内核的文件系统底层实现耦合。

gcc指定头文件路径及动态链接库路径

本文详细介绍了linux 下gcc头文件指定方法，以及搜索路径顺序的问题。另外，还总结了 gcc动态链接的方法以及路径指定。同样也讨论了搜索路径的顺

的操作性，希望读者自己去走一遍。
一.#include <>与#include ""

#include <>直接到系统指定的某些目录中去找某些头文件。
#include ""先到源文件所在文件夹去找，然后再到系统指定的某些目录中去找某些头文件。

二.gcc指定头文件的三种情况：

1.会在默认情况下指定到/usr/include文件夹(更深层次的是一个相对路径，gcc可执行程序的路径是/usr/bin/gcc，那么它在实际工作时指定头文件头径是一种相对路径方法，换算成绝对路径就是加上/usr/include，如#include 就是包含/usr/include/stdio.h)

2.GCC还使用了-I指定路径的方式，即
gcc -I 头文件所在文件夹(绝对路径或相对路径均可) 源文件
举一个例子：
设当前路径为/root/test,其结构如下：
include_test.c
include/include_test.h
有两种方法访问到include_test.h。
1. include_test.c中#include "include/include_test.h"然后gcc include_test.c即可
2. include_test.c中#include 或者#include 然后gcc –I include include_test.c也可

3. 参数：-nostdinc使编译器不再系统缺省的头文件目录里面找头文件,一般和-I联合使用,明确限定头文件的位置。

在编译驱动模块时，由于非凡的需求必须强制GCC不搜索系统默认路径，也就是不搜索/usr/include要用参数-nostdinc，还要自己用-I参数来指定内核头文件路径，这个时候必须在Makefile中指定。

头文件搜索顺序：
1.由参数-I指定的路径(指定路径有多个路径时，按指定路径的顺序搜索)

2.然后找gcc的环境变量 C_INCLUDE_PATH,
CPLUS_INCLUDE_PATH, OBJC_INCLUDE_PATH

3.再找内定目录
/usr/include
/usr/local/include
/usr/lib/gcc-lib/i386-linux/2.95.2/include
/usr/lib/gcc-lib/i386-linux/2.95.2/../../../../include/g++-3
/usr/lib/gcc-lib/i386-linux/2.95.2/../../../../i386-linux/include

库文件，但是如果装gcc的时候，是有给定的prefix的话，那么就是
/usr/include
prefix/include
prefix/xxx-xxx-xxx-gnulibc/include
prefix/lib/gcc-lib/xxxx-xxx-xxx-gnulibc/2.8.1/include

三.Linux指定动态库路径

众所周知，Linux动态库的默认搜索路径是/lib和/usr/lib。动态库被创建后，一般都复制到这两个目录中。当程序执行时需要某动态库， 并且该动态库还未加载到内存中，则系统会自动到这两个默认搜索路径中去查找相应的动态库文件，然后加载该文件到内存中，这样程序就可以使用该动态库中的函 数，以及该动态库的其它资源了。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法来

1.在配置文件/etc/ld.so.conf中指定动态库搜索路径。
可以通过编辑配置文件/etc/ld.so.conf来指定动态库的搜索路径，该文件中每行为一个动态库搜索路径。每次编辑完该文件后，都必须运行命令ldconfig使修改后的配置生效。

举一个例子：
所有源文件：
源文件1: lib_test.c

```
#include
void prt()
{
printf("You found me!!!/n");
}
```

源文件2: main.c

```
void prt();
int main()
{
prt();
return 0;
}
```

操作过程：
我们通过以下命令用源程序lib_test.c来创建动态库 lib_test.so。

```
# gcc –o lib_test.o -c lib_test.c
# gcc -shared -fPIC -o lib_test.so lib_test.o
#
```

或者直接一条指令：

```
#gcc –shared –fPIC –o lib_test.so lib_test.c
#
```

注意：
-fPIC参数声明链接库的代码段是可以共享的，
-shared参数声明编译为共享库。请注意这次我们编译的共享库的名字叫做
lib_test.so，这也是Linux共享库的一个命名的惯例了：后缀使用so，而名称使用libxxxx格式。

接着通过以下命令编译main.c,生成目标程序main.out。

```
# gcc -o main.out -L. –l_test main.c
#
```

请注意为什么是-l_test?

然后把库文件移动到目录/root/lib中。

```
# mkdir /root/lib
# mv lib_test.so /root/lib/ lib_test.so
#
```

最后编辑配置文件/etc/ld.so.conf，在该文件中追加一行/root/lib。

运行程序main.out:

```
# ./main.out
./main.out: error while loading shared libraries: lib_test.so: cannot
open shared object file: No such file or directory
#
```

出错了，系统未找到动态库lib_test.so。找找原因，原来在编辑完配置文件/etc/ld.so.conf后，没有运行命令ldconfig，所以刚才的修改还未生效。我们运行ldconfig后再试试。

```
# ldconfig
# ./main.out
You found me!!!
```

#
程序main.out运行成功，并且打印出正确结果。

2.通过环境变量LD_LIBRARY_PATH指定动态库搜索路径。
通过设定环境变量LD_LIBRARY_PATH也可以指定动态库搜索路径。当通过该环境变量指定多个动态库搜索路径时，路径之间用冒号":"分隔。下面通过例2来说明本方法。

举一个例子：
这次我们把上面得到的文件lib_test.so移动到另一个地方去，如/root下面，然后设置环境变量LD_LIBRARY_PATH找到lib_test.so。设置环境变量方法如下：
export LD_LIBRARY_PATH=/root
#
然后运行：
#./main.out
You found me!!!
#
注意：设置环境变量LD_LIBRARY_PATH=/root是不行的，非得export才行。

3.在编译目标代码时指定该程序的动态库搜索路径。
还可以在编译目标代码时指定程序的动态库搜索路径。-Wl,表示后面的参数将传给link程序ld（因为gcc可能会自动调用ld）。这里通过gcc 的参数"-Wl,-rpath,"指定
举一个例子：
这次我们还把上面得到的文件lib_test.so移动到另一个地方去，如/root/test/lib下面，
因为我们需要在编译目标代码时指定可执行文件的动态库搜索路径，所以需要用gcc命令重新编译源程序main.c(见程序2)来生成可执行文件main.out。
gcc -o main.out -L. –l_test -Wl,-rpath,/root/test/lib main.c
#

运行结果：
./main.out
You found me!!!
#

程序./main.out运行成功，输出的结果正是main.c中的函数prt的运行结果。因此程序main.out搜索到的动态库是/root/test/lib/lib_test.so。

关于-Wl,rpath的使用方法我再举一个例子，应该不难从中看出指定多个路径的方法：
gcc -Wl,-rpath,/home/arc/test,-rpath,/lib/,-rpath,/usr/lib/,-rpath,/usr/local/lib test.c

以上介绍了三种指定动态库搜索路径的方法，加上默认的动态库搜索路径/lib和/usr/lib，共五种动态库的搜索路径，那么它们搜索的先后顺序是什么呢？读者可以用下面的方法来试验一下：
（1） 用前面介绍的方法生成5个lib_test.so放在5个不同的文件夹下面，要求每一个lib_test.so都唯一对应一个搜索路径，并注意main.out程序输出的不同。
（2） 运行main.out，即可看出他是那个搜索路径下的，然后删除这个路径下的lib_test.so，然后再运行。依此类推操作，即可推出搜索顺序。

可以得出动态库的搜索路径搜索的先后顺序是：

1.编译目标代码时指定

2.环境变量LD_LIBRARY_PATH指定的动态库搜索路径；

3.配置文件/etc/ld.so.conf中指定的动态库搜索路径；

4.默认的动态库搜索路径/lib；

5.默认的动态库搜索路径/usr/lib。

在上述1、2、3指定动态库搜索路径时，都可指定多个动态库搜索路径，其搜索的先后顺序是按指定路径的先后顺序搜索的。有兴趣的读者自己验证。

PS：此文网上搜得，原始出处已经无法访问，故无法给出原文链接。