



日志

◀ 后缀树简介

Makefile中函数wildcard ▶

Linux下用gcc生成静态库和动态库

2011-06-15 17:29:38 | 分类: 默认分类

订阅 | 字号

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。
共享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。

1.3 库存在的意义

库是别人写好的现有的，成熟的，可以复用的代码，你可以使用但要记得遵守许可协议。
现实中每个程序都要依赖很多基础的底层库，不可能每个人的代码都从零开始，因此库的存在意义非同寻常。
共享库的好处是，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。

1.4 库文件是如何产生的在 linux 下

静态库的后缀是 .a ，它的产生分两步
Step 1. 由源文件编译生成一堆 .o ，每个 .o 里都包含这个编译单元的符号表
Step 2.ar 命令将很多 .o 转换成 .a ，成文静态库
动态库的后缀是 .so ，它由 gcc 加特定参数编译产生。
具体方法参见后文实例。

1.5 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

1.6 如何知道一个可执行程序依赖哪些库

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 (0x40021000)/lib/ld-linux.so.2

=> /lib/ld- linux.so.2 (0x40000000)

可以看到 ln 命令依赖于 libc 库和 ld-linux 库

1.7 可执行程序在执行的时候如何定位共享库文件

当系统加载可执行代码时候，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

1.8 在新安装一个库之后如何让系统能够找到他

如果安装在 /lib 或者 /usr/lib 下，那么 ld 默认能够找到，无需其他操作。

如果安装在其他目录，需要将其添加到 /etc/ld.so.cache 文件中，步骤如下

- 1. 编辑 /etc/ld.conf 文件，加入库文件所在目录的路径
- 2. 运行 ldconfig ，该命令会重建 /etc/ld.so.cache 文件

二、用gcc生成静态和动态链接库的示例

我们通常把一些公用函数制作成函数库，供其它程序使用。

函数库分为 静态库 和 动态库 两种。

静态库在程序编译时会被连接到目标代码 中，程序运行时将不再需要该静态库。

动态库在程序编译时并不会被连接到目标 代码中，而是在程序运行是才被载入，因此在程序运行时还需要动态库存在。

本文主要通过举例来说明在 Linux 中如何创建静态库和动态库，以及使用它们。

为了便于阐述，我们先做一部分准备工 作。

2.1 准备好测试代码 hello.h 、 hello.c 和 main.c ；

hello.h(见程序 1) 为该函数库的头文件。

hello.cpp(见程序 2) 是函数库的源程序，其中包含公用函数 hello ，该函数将在屏幕上输出 "Hello XXX!"

main.cpp(见程序 3) 为测试库文件的主程序，在主程序中调用了公用函数 **hello**。

程序 1: hello.h

```
#ifndef HELLO_H
#define HELLO_H

void hello( const char * name ) ;

#endif
```

程序2: hello.cpp

```
#include <stdio.h>
void hello( const char * name ) {
    printf ( "Hello %s!\n" , name ) ;
}
```

程序3: main.cpp

```
#include "hello.h"
int main( )
{
    hello( "everyone" ) ;
    return 0;
}
```

2.2 问题的提出

注意：这个时候，我们编译好的 **hello.o** 是无法通过 **gcc -o** 编译的，这个道理非常简单，**hello.c** 是一个没有 **main** 函数的 **.c** 程序，因此不够成一个完整的程序，如果使用 **gcc -o** 编译并连接它，GCC 将报错。

无论静态库，还是动态库，都是由 **.o** 文件创建的。因此，我们必须将源程序 **hello.c** 通过 **gcc** 先编译成 **.o** 文件。

这个时候我们有 三种思路：

- 1) 通过编译多个源文件，直接将目标代码合成一个 **.o** 文件。
- 2) 通过创建静态链接库 **libmyhello.a**，使得 **main** 函数调用 **hello** 函数时可调用静态链接库。
- 3) 通过创建动态链接库 **libmyhello.so**，使得 **main** 函数调用 **hello** 函数时可调用静态链接库。

2.3 思路一：编译多个源文件

在系统提示符下键入以下命令得到 **hello.o** 文件。

g++ -c hello.cpp

为什么不适用 **g++-o hello hello.cpp** 这个道理 我们之前已经说了，使用 **-c** 是什么意思呢？这涉及到 **g++** 编译选项 的常识。

我们通常使用的 **g++ -o** 是将 **.cpp** 源文件编译成为一个可执行的二进制代码，这包括调用作为 **GCC** 内的一部分真正的 **C** 编译器（**ccl**），以及调用 **GNU C** 编译器的输出中实际可执行代码的外部 **GNU** 汇编器

和 连接器工具。

而 `g++ -c` 是使用 GNU 汇编器将 源文件转化为目标代码之后就结束，在这种情况下连接器并没有被执行，所以输出的目标文件不会包含作为 Linux 程序在被 装载和执行时所必须的包含信息，但它可以在以后被连接到一个程序。

我们运行 `ls` 命令看看 是否生存了 `hello.o` 文件。

```
# ls
```

```
hello.cpp hello.h hello.o main.cpp
```

在 `ls` 命令结果 中，我们看到了 `hello.o` 文件，本 步操作完成。

同理编译 `main`

```
#g++ -c main.cpp
```

将两个文件链接成一个 `.o` 文件。

```
#g++ -o hello hello.o main.o
```

运行

```
# ./hello
```

```
Hello everyone!
```

完成 ^_^!

2.4 思路二：静态链接库

下面我们来看看如何创建静态库，以及使用它。

静态库文件名的命名规范是以 `lib` 为前缀，紧接着跟静态库名，扩展名为 `.a`。例如：我们将创建的静态库名为 `myhello`，则静态 库文件名就是 `libmyhello.a`。在创建 和使用静态库时，需要注意这点。创建静态库用 `ar` 命令。

在系统提示符下键入以下命令将创建静态库文件 `libmyhello.a`。

```
# ar cr libmyhello.a hello.o
```

我们同样运行 `ls` 命令查看 结果：

```
# ls
```

```
hello.cpp hello.h hello.o libmyhello.a main.cpp
```

`ls` 命令结果 中有 `libmyhello.a`。

静态库制作完了，如何使用它内部的函数呢？只需要在使用到这些公用函数的源程序 中包含这些公用函数

的原型声明，然后在用 `gcc` 命令生成 目标文件时指明静态库名，`gcc` 将会从静 态库中将公用函数连接到目标文件中。注意，`gcc` 会在静态 库名前加上前缀 `lib`，然后追 加扩展名 `.a` 得到的静 态库文件名来查找静态库文件。

在程序 3:main.cpp 中，我们 包含了静态库的头文件 `hello.h`，然后在 主程序 `main` 中直接调 用公用函数 `hello`。下面先 生成目标程序 `hello`，然后运 行 `hello` 程序看看 结果如何。

```
# g++ -o hello main.cpp -L. -lmyhello
```

```
# ./hello
```

```
Hello everyone!
```

我们删除静态库文件试试公用函数 `hello` 是否真的 连接到目标文件 `hello` 中了。

```
# rm libmyhello.a
```

```
rm: remove regular file `libmyhello.a'? y
```

```
# ./hello
```

```
Hello everyone!
```

程序照常运行，静态库中的公用函数已经连接到目标文件中了。

静态链接库的一个缺点是，如果我们同时运行了许多程序，并且它们使用了同一个库 函数，这样，在内存中会大量拷贝同一库函数。这样，就会浪费很多珍贵的内存和存储空间。使用了共享链接库的Linux就可以避免这个问题。

共享函数库和静态函数在同一个地方，只是后缀有所不同。比如，在一个典型的 Linux系统，标准的共享数序函数库是`/usr/lib/libm.so`。

当一个程 序使用共享函数库时，在连接阶段并不把函数代码连接进来，而只是链接函数的一个引用。当最终的函数导入内存开始真正执行时，函数引用被解析，共享函数库的 代码才真正导入到内存中。这样，共享链接库的函数就可以被许多程序同时共享，并且只需存储一次就可以了。共享函数库的另一个优点是，它可以独立更新，与调 用它的函数毫不影响。

2.5 思路三、动态链接库（ 共享函数库 ）

我们继续看看如何在 Linux 中创建动 态库。我们还是从 `.o` 文件开 始。

动态库文件名命名规范和静态库文件名命名规范类似，也是在动态库名增加前缀 `lib`，但其文 件扩展名为 `.so`。例如： 我们将创建的动态库名为 `myhello`，则动态 库文件名就是 `libmyhello.so`。用 `g++` 来创建动态库。

在系统提示符下键入以下命令得到动态库文件 `libmyhello.so` 。

```
# g++ -shared -fPIC -o libmyhello.so hello.o
```

“`PIC`” 命令行标记告诉 `GCC` 产生的代码不要包含对函数和变量具体内存位置的引用，这是因为现在还不知道使用该消息代码的应用程序会将它连接到哪一段内存地址空间。这样编译出的 `hello.o` 可以被用于建立共享链接库。建立共享链接库只需要用 `GCC` 的 “`-shared`” 标记即可。

我们照样使用 `ls` 命令看看 动态库文件是否生成。

```
# ls
```

```
hello.cpp hello.h hello.o libmyhello.so main.cpp
```

调用动态链接库编译目标文件。

在程序中使用动态库和使用静态库完全一样，也是在使用到这些公用函数的源程序中 包含这些公用函数的原型声明，然后在用 `gcc` 命令生成 目标文件时指明动态库名进行编译。我们先运行 `gcc` 命令生成 目标文件，再运行它看看结果。

```
# g++ -o hello main.cpp -L. -lmyhello
```

{也可以这样：`g++ -o hello main.cpp /home/zhdr/test/libmyhello.so` 就不用 `mv libmyhello.so /usr/lib`。但也会碰到下面的问题，也需要：`chcon -t texrel_shlib_t /home/zhdr/test/libmyhello.so` }

使用 “`-lmyhello`” 标记来告诉 `GCC` 驱动程序 在连接阶段引用共享函数库 `libmyhello.so` 。“`-L.`” 标记告诉 `GCC` 函数库可能位于当前目录。否则 `GNU` 连接器会 查找标准系统函数目录。

```
# ./hello
```

```
./hello: error while loading shared libraries: libmyhello.so: cannot open shared object file: No such file or directory
```


错误提示，找不到动态库文件 `libmyhello.so` 。程序在 运行时，会在 `/usr/lib` 和 `/lib` 等目录中 查找需要的动态库文件。若找到，则载入动态库，否则将提示类似上述错误而终止程序运行。我们将文件 `libmyhello.so` 复制到目录 `/usr/lib` 中，再试试。

```
# mv libmyhello.so /usr/lib
```

```
# ./hello
```

Hello everyone!

■这步后我没有成功,报错内容如下: `/hello: error while loading shared libraries: /usr/lib/libmyhello.so: cannot restore segment prot after reloc: Permission denied`

google了一下，发现是 SELinux搞的鬼，解决办法有两个：

1.

```
chcon -t texrel_shlib_t /usr/lib/libmyhello.so
(chcon -t texrel_shlib_t "你不能share的库的绝对路径")
```

2.

```
#vi /etc/sysconfig/selinux file
或者用
#gedit /etc/sysconfig/selinux file
修改SELINUX=disabled
重启
```

#

这也进一步说明了动态库在程序运行时是需要的。

(注：以下来自转载，没亲自验证)

ldd hello

执行 test, 可以看到它是如何调用动态库中的函数的。

```
[pin@localhost 20090505]$ ldd hello
linux-gate.so.1 => (0x00110000)
libmyhello.so => /usr/lib/libmyhello.so (0x00111000)
libc.so.6 => /lib/libc.so.6 (0x00859000)
/lib/ld-linux.so.2 (0x0083a000)
```

我们回过头看看，发现使用静态库和使用动态库编译成目标程序使用的gcc命令完全一样，那当静态库和动态库同名时，gcc命令会使用哪个库文件呢？抱着对问题必究到底的心情，来试试看。

先删除除.c和.h外的所有文件，恢复成我们刚刚编辑完举例程序状态。

```
# rm -f hello hello.o /usr/lib/libmyhello.so
```

ls

```
hello.c hello.h main.c
```

#

再来创建静态库文件libmyhello.a和动态库文件libmyhello.so。

```
# gcc -c hello.c
```

```
# ar cr libmyhello.a hello.o
```

```
# gcc -shared -fPIC -o libmyhello.so hello.o
```

ls

```
hello.c hello.h hello.o libmyhello.a libmyhello.so main.c
```

#

通过上述最后一条ls命令，可以发现静态库文件libmyhello.a和动态库文件libmyhello.so都已经生成，并都在当前目录中。然后，我们运行 gcc命令来使用函数库myhello生成目标文件hello，并运行程序 hello。

```
# gcc -o hello main.c -L. -lmyhello
```

```
# ./hello
```

```
./hello: error while loading shared libraries: libmyhello.so: cannot open shared object file: No such file or directory
```

#

从程序hello运行的结果中很容易知道，当静态库和动态库同名时，gcc命令将优先使用动态库。

Note:

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

I -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。

I -L.：表示要连接的库在当前目录中

I -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

I 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。

I 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。

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。

静态库链接时搜索路径顺序：

1. ld会去找GCC命令中的参数-L
2. 再找gcc的环境变量LIBRARY_PATH
3. 再找内定目录 /lib /usr/lib /usr/local/lib 这是当初compile gcc时写在程序内的

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。

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

附注：

作为Linux程序开发员，最好对开发工具和资源的位置有个初步了解。下面简要 介绍一下主要的文件夹和应用程序。

1.应用程序(Applications)

应用程序通常都有固定的文件夹，系统通用程序放在/usr/bin，日后系统管 理员在本地计算机安装的程序通常放在/usr/local/bin或者/opt文件夹下。除了系统程序外，大部分个人用到的程序都放在/usr /local下，所以保持/usr的整洁十分重要。当升级或者重装系统的时候，只要把/usr/local的程序备份一下就可以了。

一些其他的程序有自己特定的文件夹，比如X Window系统，通常安装在/usr/X11中，或 者/usr/X11R6。GNU的编译器GCC，通常放置在/usr/bin或者/usr /local/bin中，不同的Linux版本可能位置稍有不同。

2.头文 件(Head Files)

在C语言和其他语言中，头文件声明了系统函数和库函数，并且定义了一些常量。对 于C语言，头文件基本上散落于/usr/include和它的子文件夹下。其他的编程语言的库函数分布在编译器定义的地方，比如在一些Linux版本 中，X Window系统库函数分布在/usr/include/X11，GNU C++的库函数分布 在/usr/include/g++。这些系统库函数的位置对于编译器来说都是“标准位置”，即编译器能够自动搜寻这些位置。

如果想引用位于标准位置之外的头文件，我们需要在调用编译器的时候加上-I标 志，来显式的说明头文件所在文件夹。比如，

```
$ gcc -I/usr/openwin/include hello.c
```

会告诉编译器除了标 准位置外，还要去/usr/openwin/include看看有没有所需的头文件。详细情况见编译器的使用手册(man gcc)。

库函数 (Library Files)

库函数就是函数的仓库，它们都经过编译，重用性不错。通常，库函数相互合作，来 完成特定的任务。比如操控屏幕的库函数(cursers和ncursers库函数)，数据库读取库函数(dbm库函数)等。

系统调用的标准库函数一般位于/lib以及/usr/lib。C编译器(精确点 说，连接器)需要知道库函数的位置。默 认情况下，它只搜索标准C库函数。

库函数文件通常开头字母是lib。后面的部分标示库函数的用途(比如C库函数用c标识， 数学库函数用m标 示)，小数点后的后缀表明库函数的类型：

- .a 指静态链接库
- .so 指动态链接库

去/usr/lib看一下，你会发现，库函数都有动态和静态两个版本。

与头文件一样，库函数通常放在标准位置，但我们也可以通过-L标识符，来添加新 的搜索文件夹，-l指定特定的库函数文件。比如

```
$ gcc -o x11fred -L/usr/openwin/lib x11fred.c -lX11
```



上述命令就会在编译期间，链接位于/usr/openwin/lib文件夹下的 libX11函数库，编译生成x11fred。

静态链接库(Static Libraries)

最简单的函数库就是一些函数的简单集合。调用库函数中的函数时，需要在调用函数 中include定义库函数的头文件。我们用-l选项添加标准函数库之外的函数库。

静态函数库，也称为 *archives* ，通常以后缀.a结尾。

 推荐

分享到：

阅读(552) | 评论(0) | 引用 (0) | 举报

◀ 后缀树简介

Makefile中函数wildcard ▶

最近读者

评论