Linux系统中程序库文件简介

简介

库文件一般就是编译好的二进制文件，用于在链接阶段同目标代码一起生成可执行文件，或者运行可执行文件的时候被加载，以便调用库文件中的某段代码。它与可执行文件相同之处是：两者都是编译好的二进制文件（本文中我们讨论的二进制文件，假设都是Linux上面最常见的ELF格式）；与可执行文件不同的是：库文件无法直接执行（直观上来看它的源代码中没有main函数，而只是一些函数模块的定义和实现，没有运行的入口主函数，所以无法直接执行）。我们开发的程序，无论是运行的时候，还是编译、链接的时候，一般都需要借助一些库来实现它们的功能，而很少直接只通过程序源代码生成完全独立的可执行文件。有许多著名的常用的用于开发或者运行程序所需的库，例如Qt库，gtk库，甚至是标准C库等等，通过使用它们可以充分体会到模块化编程和代码重用等的好处。本文对Linux库的编译，生成，使用进行了简单的介绍，并且通过一个简单例子（开发自己的库）进行说明。

主要内容

原理

举例

其它

［原理］

为便于理解，我们可以将库分为三种类型：静态库,共享库，动态加载库,。下面分别介绍。

一、 静态库

静态库实际就是一些目标文件(一般以.o结尾)的集合，静态库一般以.a结尾，只用于链接生成可执行文件阶段。具体来说，以c程序为例，一般我们编译程序源代码的时候，过程大致是这样的：以.c为后缀的源文件经过编译生成.o的目标文件，以.o为后缀的目标文件经过链接生成最终可执行的文件。我们可以在链接的时候直接链接.o的目标文件，也可以将这些.o目标文件打包集中起来，统一链接，而这个打包生成的集中了所有.o文件的文件，就是静态库。静态库只在程序链接的时候使用，链接器会将程序中使用到函数的代码从库文件中拷贝到应用程序中，一旦链接完成生成可执行文件之后，在执行程序的时候就不需要静态库了。由于每个使用静态库的应用程序都需要拷贝所用函数的代码，所以静态链接的生成的可执行文件会比较大，多个程序运行时占用内存空间比较大(每个程序在内存中都有一份重复的静态库代码)，但是由于运行的时候不用从外部动态加载额外的库了，速度会比共享库快一些。

我们将在后面的例子中看到静态库的生成和应用的具体过程。

二、共享库

1、共享库的概念

共享库以.so结尾. (so == share object) 在程序链接的时候并不像静态库那样从库中拷贝使用的函数代码到生成的可执行文件中，而只是作些标记，然后在程序开始启动运行的时候，动态地加载所需库（模块）。所以，应用程序在运行的时候仍然需要共享库的支持。共享库链接出来的可执行文件比静态库链接出来的要小得多，运行多个程序时占用内存空间比也比静态库方式链接少(因为内存中只有一份共享库代码的拷贝)，但是由于有一个动态加载的过程所以速度稍慢。

2、共享库的命名

一般一个共享库有三个名字：soname, real-name, linker-name。

soname是用于区分版本的名字，它可能就是指向real-name（如果有这个文件的话）的链接，名称的形式一般是lib\*.so.X.Y(这里的X，Y就是代表版本号)。

real-name是包含真正代码的实现文件。

linker-name是传递给连接器的名字，用于链接的搜索，一般它可能就是指向soname的连接，名称的形式一般是lib\*.so。

这样做的目的主要是允许系统中多个版本的库文件共存，习惯上在命名库文件的时候通常与soname相同 。

下面看一个实例:

[quietheart@lv-k test]$ ls -l /usr/lib/libncurses\*

lrwxrwxrwx 1 root root     20 2010-07-26 16:51 libncurses.so -> /lib/libncurses.so.5

lrwxrwxrwx 1 root root     13 2010-07-26 17:18 libncurses.so.5 -> libtermcap.so

上面的libncurses.so.5就是soname, 其中ncurses是库名，5分别是主版本号(major)，当然也可以有次版本号(minor)和发行号(release)。(类似于libncurses.so.5.0.0)。这里的".so"表示共享库。通常soname只是real name的一个链接。而libtermcap.so 是ncurse库的real-name, 也就是包含真实代码实现的文件。libncurses.so 则是linker name,用于应用程序链接的时候的一个搜索名使用这个名字传递给链接器进行链接。 它通常是soname的一个链接,形式为libname.so。

实际上，每一个库都有一个soname，当连接器发现它正在查找的程序库中有这样一个名称，连接器便会将soname嵌入连结中的二进制文件内，而不是它正在运行的实际文件名，在程序执行期间，程序会查找拥有 soname名字的文件，而不是库的文件名，换句话说，soname是库的区分标志。

3、共享库的装载

(1) 在所有基于GNU glibc的系统(当然包括Linux)中，在启动一个ELF二进制执行程序时，一个特殊的程序"程序装载器"会被自动装载并运行。在linux中，这个程序装载器就是/lib/ld-linux.so.X(X是版本号)。它会查找并装载应用程序所依赖的所有共享库。被搜索的目录保存在/etc/ls.so.conf文件中，但如果某个所使用的库的路径不在搜索之列，那么就只好自己加上了。当然，如果程序每次启动都要去搜索一遍，势必效率不堪忍受。Linux系统已经考虑这一点，对共享库采用了缓存管理。ldconfig就是实现这一功能的工具，其缺省读取/etc/ld.so.conf文件，对所有共享库按照一定规范建立符号连接，然后将信息写入/etc/ld.so.cache。每次搜索的时候实际通过ld.so.cache这个缓存文件进行搜索，/etc/ld.so.cache的存在大大加快了程序的启动速度。当然，每次修改ld.so.conf文件之后，应当运行一下“ldconfig”命令以便把信息也更新到缓存文件中。

(2) 也可以通过设置环境变量LD\_LIBRARY\_PATH来设置ld的装载路径。这样装载器就会首先搜索该变量的目录，然后才是默认目录。但是记住，LD\_LIBRARY\_PATH是用于开发和测试的，你可以将一些用于测试的替代共享库的目录放到该变量中，类似于/etc /ld.so.preload的作用。但是该变量不应该用于正常用户的正常程序。

(3) 如果不使用LD\_LIBRARY\_PATH环境变量，可以通过如下方式给装载器传入路径:

[quietheart@lv-k test]$ /lib/ld-linux.so.2 --library-path PATH EXECUTABLE

总之，想要你的共享库被装载，那么一般通过一下三个方式：

a）拷贝你的库到默认的库搜索路径/usr/lib中。

b）或设置环境变量LD\_LIBRARY\_PATH,在其中添加你的库所在的路径。

c) 或修改配置文件/etc/ld.so.conf加入你的库所在的路径，并刷新缓存ldconfig。

三、 动态加载库

1. 概念

动态加载库(dynamically loaded (DL) libraries)是指在程序运行过程中可以加载的函数库。而不是像共享库一样在程序启动的时候加载。DL对于实现插件和模块非常有用，因为他们可以让程序在允许时等待插件的加载。在Linux中，动态库的文件格式跟共享库没有区别，主要区别在于共享库是程序启动时加载，而动态加载库是运行的过程中加载。

2.使用

有专门的一组API用于完成打开动态库，查找符号，处理出错，关闭动态库等功能。下面对这些接口函数逐一介绍:

(1) dlopen

函数原型：void \*dlopen(const char \*libname,int flag);

功能描述：dlopen必须在dlerror，dlsym和dlclose之前调用，表示要将库装载到内存，准备使用。

如果要装载的库依赖于其它库，必须首先装载依赖库。如果dlopen操作失败，返回NULL值；如果库已经被装载过，则dlopen会返回同样的句柄。

参数中的libname一般是库的全路径，这样dlopen会直接装载该文件；如果只是指定了库名称，在dlopen会按照下面的机制去搜寻：

a.根据环境变量LD\_LIBRARY\_PATH查找

b.根据/etc/ld.so.cache查找

c.查找依次在/lib和/usr/lib目录查找。

flag参数表示处理未定义函数的方式，可以使用RTLD\_LAZY或RTLD\_NOW。RTLD\_LAZY表示暂时不去处理未定义函数，先把库装载到内存，等用到没定义的函数再说；RTLD\_NOW表示马上检查是否存在未定义的函数，若存在，则dlopen以失败告终。

(2) dlerror

函数原型：char \*dlerror(void);

功能描述：dlerror可以获得最近一次dlopen,dlsym或dlclose操作的错误信息，返回NULL表示无错误。dlerror在返回错误信息的同时，也会清除错误信息。

(3) dlsym

函数原型：void \*dlsym(void \*handle,const char \*symbol);

功能描述：在dlopen之后，库被装载到内存。dlsym可以获得指定函数(symbol)在内存中的位置(指针)。

如果找不到指定函数，则dlsym会返回NULL值。但判断函数是否存在最好的方法是使用dlerror函数，

(4) dlclose

函数原型：int dlclose(void \*);

功能描述：将已经装载的库句柄减一，如果句柄减至零，则该库会被卸载。如果存在析构函数，则在dlclose之后，析构函数会被调用。

[举例]

下面，通过一个具体的实例，对前面介绍的三种库的开发，使用，以及部署进行演示。

首先给出我们所需要的程序的源代码：

[quietheart@lv-k test]$ ls

main.cpp  myfile.cpp  myfile.h

[quietheart@lv-k test]$ cat main.cpp

//main.cpp文件，生成可执行文件的源代码。

#include<iostream>

#include "myfile.h"

using std::cout;

using std::endl;

int main(int argc, char \*argv[])

{

cout<<"begin test"<<endl;

printInfo();

return 0;

}

[quietheart@lv-k test]$ cat myfile.h

//myfile.h，库的头文件

#ifndef \_\_MYFILE\_H

#define \_\_MYFILE\_H

void printInfo();

#endif

[quietheart@lv-k test]$ cat myfile.cpp

//myfile.cpp，库的实现代码

#include "myfile.h"

#include <iostream>

using std::cout;

using std::endl;

void printInfo()

{

cout<<"hello"<<endl;

}

从以上我们可以知道，我们的代码只个文件：main.cpp,myfile.h,和myfile.cpp。

一、不使用库

只要有一点Linux编程知识的读者都会知道，有了这三个文件，我们就可以通过"g++ main.cpp myfile.cpp"命令，生成我们的可执行文件并运行。具体过程如下：

[root@lv-k test]# ls

main.cpp  myfile.cpp  myfile.h

[root@lv-k test]# g++ main.cpp myfile.cpp

[root@lv-k test]# ls

a.out  main.cpp  myfile.cpp  myfile.h

[root@lv-k test]# ./a.out

begin test

hello

在上面生成可执行文件的过程中，我们并没使用任何库的概念，也就是说我们将所文件都做为可执行文件的一个部分，最终生成了一个无论是编译、链接，还是运行的时候，都不依赖于任何库的，独立执行的可执行文件（实质上严格来讲这个可执行文件还是依赖库的，至少它依赖iostream库，这里不考虑这些）。用下载过许多软件的朋友们的“术语”来说，我们现在生成的程序，就是一个免安装的“绿色”软件^\_^。如果程序的源代码结构很复杂的化，那么这样编译的缺点是非常明显的，那就是这个程序采用非模块化的方式编译，至少不满足可重用性的特点。后面我们将介绍使用库来实现模块化编译这个程序。

二、使用库

我们将要用如下的方式组织我们的程序：

1）对应我们应用程序，也就是可执行文件的代码只有一个，那就是main.cpp文件。

2）我们的可执行文件的生成或者运行，依赖外部的一些库，例如iostream，以及myfile.h所对应的库(名称叫my)，这里我们重点关注my这个我们自己定义的库。

3）将myfile.h和myfile.cpp单独进行编译生成库my，库my的实现实际是通过myfile.cpp生成的；而myfile.h的作用就是让使用这个库的程序知道这个库包含了什么功能的函数，也就是说，库的头文件只是一个让其它程序使用该库的接口文件。它存在的作用就是让使用这个库的程序通过#include "myfile.h"来包含，这样就声明了一些库中必要的函数。

这样就实现了模块化的目的，将整个程序划分为可执行文件部分，以及可执行文件所使用的库这两个相对独立的部分。后面将讲述如何真正在编译执行的角度实现这种划分的思想。

1、静态库方式

采用如下方式进行：

1.1生成静态库：

1)只编译myfile.cpp生成myfile.o

[quietheart@lv-k test]$g++ -c myfile.cpp

2)根据myfile.o生成库libmy.a

[quietheart@lv-k test]$ar r libmy.a myfile.o

3)删除myfile.o和myfile.cpp

[quietheart@lv-k test]$rm myfile.cpp myfile.o

这样，我们第2步使用ar命令，将.o文件打包添加到libmy.a中，打包生成的lib\*.a文件，就是静态库（相对的动态库是lib\*.so文件）。关于ar命令，除了r选项还有c，s等选项： r表明将模块加入到静态库中，c表示创建静态库，s表示生产索引。具体参见ar命令的用户手册。另外这里，为简明起见，我们删除了myfile.o和myfile.cpp文件。

1.2使用静态库链接并运行：

1)使用libmy.a进行编译连接:

[quietheart@lv-k test]$g++ main.cpp -L./ -lmy

2)运行程序:

[quietheart@lv-k test]$./a.out

这里，可以修改libmy.a的名字为libmy2.a，这样就相应地用“g++ main.cpp -L./ -lmy2”进行链接。这样只在编译的时候需要对静态库进行链接，生成可执行文件之后，静态库对这个程序而言就没有用了（当然对别的程序的生成可能有用），至此生成的可执行文件，也可以称作是免安装的“绿色软件”^\_^。

进一步的说明：

这里的-L选项，使用-L.表示将当前目录加入到库搜索路径。否则使用默认的库搜索路径搜索库文件，也就是/usr/lib目录。

另外个类似的容易混淆的参数-I, 它表示搜索头文件的路径。使用它这样gcc在查找头文件的时候会首先到-I指定的目录查找头文件，然后才是系统默认目录，也就是/usr/include。

这里的-l选项, -lname表示库搜索目录下的libname.a 或者libname.so文件 ,这也是为什么库文件都以lib开头的原因之一，如果你的库文件不是libmy,而是my. 那就不能用-l参数编译了。 可以这样：

[quietheart@lv-k test]$g++ main.cpp -L. my.a -o test

注意: $g++ -L. -lmy main.o -o test 会出错！。

原因是: -l是链接器选项，必须要放到被编译文件的后面。 所以上面的命令中-lmy一定要放到 main.o的后面。

2、共享库方式

采用如下方式进行：

1.1生成共享库：

1)只编译myfile.cpp生成myfile.o

[quietheart@lv-k test]$g++ -c myfile.cpp

2)根据myfile.o生成动态库libmy.so

[quietheart@lv-k test]$g++ -shared -fPCI -o libmy.so myfile.o

3)删除myfile.o和myfile.cpp

[quietheart@lv-k test]$rm myfile.cpp myfile.o

这里，我们第2步生成的libmy.so就是动态连接库（共享库）。实践发现使用gcc也行，还可用"g++ -shared -o libmy.so myfile.o"。

进一步的说明:  -fpic或者-fPIC表明创建position independent code，这通常是创建共享库必须的。另外，-Wl 表明给链接器传送参数，所以这里-soname, library\_name 为给链接器的参数。-shared 表明是使用共享库。

下面是使用a.c和b.c创建共享库的示例：

gcc -fPIC -g -c -Wall a.c

gcc -fPIC -g -c -Wall b.c

gcc -shared -Wl,-soname, libmyab.so.1 -o libmyab.so.1.0.1 a.o b.o -lc

说明: lc == libc

还几个需要注意的地方：

a.不推荐使用strip处理共享库，最好不要使用-fomit-frame-pointer编译选项，

b.-fPIC和-fpic都可以产生目标独立代码，具体应用取决于平台，-fPIC是always work，

尽管其产生的目标文件可能会大些； -fpic产生的代码小，执行速度快，但可能有平台依赖限制。

c.一般情况下，-Wall,-soname,your\_soname编译选项是需要的。当然，-share选项更不能丢。

1.2使用共享库libmy.so进行链接:

1)编译链接：

[quietheart@lv-k test]$g++ main.cpp -L./ -lmy

这里，不要和libmy.a冲突了,如果同时存在libmy.a和libmy.so会优先选择libmy.so。编译选项类似前面请参照链接静态库，时候的选项。

1.3运行时加载共享库libmy.so：

1)将动态库移动到/usr/lib等标准路径：

[quietheart@lv-k test]$sudo cp libmy.so /usr/lib

2)运行程序:

[quietheart@lv-k test]$./a.out

这里注意和静态库不同，还需要把共享库移动到特定的位置，因为共享库在运行之后还有用。

实际有三种方法来让运行的程序可以加载到你的共享库文件：

a）拷贝共享库文件到/usr/lib

b)或设置环境变量LD\_LIBRARY\_PATH加上你共享库文件的路径

c) 或修改配置文件/etc/ld.so.conf加入你共享库文件的路径，并刷新缓存ldconfig

至此生成的程序，就不是免安装的"绿色"软件了，因为想要你的程序运行在一台机器上，除了可执行文件本身，你还要将这个可执行文件所依赖的库拷贝到系统的某个目录中以便运行的时候加载，这个拷贝的过程“污染”了系统，所以说它不是"绿色"的。

3、动态加载库方式

动态加载库的方式，就是说启动程序的时候不用将事先编译好的库加载，而是在用到的时候动态加载库，这种方式常常用在插件加载的方式上。需要修改我们的main.cpp文件如下：

[quietheart@lv-k test]$vim main.cpp

#include<iostream>

#include<dlfcn.h>

using std::cout;

using std::cerr;

using std::endl;

int main(int argc, char \*argv[])

{

//初始变量

void \*handle;

void (\*pPrint)(void);

char \*error;

cout<<"Begin to call:"<<endl;

//动态加载库，RTLD\_LAZY表示如果遇到标号不存在的情况，不做处理，而是运行时候再说。

handle = dlopen("./libmy.so", RTLD\_LAZY);

if(handle == NULL)

{

cerr<<dlerror()<<endl;

exit(1);

}

//找到将要调用的函数标号,采用的标号不是库文件源代码中的printInfo，

//而是通过"nm libmy.so" 或者"readelf -s libmy.so"查询得知，标号名称为：\_Z9printInfov

pPrint = (void(\*)())dlsym(handle,"\_Z9printInfov");

error = dlerror();

if( error != NULL )

{

cerr<<error<<endl;

exit(1);

}

//调用函数

pPrint();

//关闭

dlclose(handle);

return 0;

}

由上可知，使用动态加载的方式，不用在可执行文件中包含库的头文件就能够使用库中的函数。并且，库文件的搜索路径除了像前面那样通过系统确定，我们还可以在代码中“硬性”指定在哪里加载库。

具体编译运行过程如下。

3.1，生成动态库文件：

1)只编译myfile.cpp生成myfile.o

[quietheart@lv-k test]$g++ -c myfile.cpp

2)根据myfile.o生成动态库libmy.so

[quietheart@lv-k test]$g++ -shared -fPCI -o libmy.so myfile.o

3)删除myfile.o和myfile.cpp

[quietheart@lv-k test]$rm myfile.cpp myfile.o

这里，我们第2步生成的libmy.so就是动态加载的库（生成方式和共享库一样）。

3.2，编译链接执行可执行程序：

1)生成可执行文件：

[quietheart@lv-k test]$g++ main.cpp -ldl

2)运行：

[quietheart@lv-k test]$./a.out

这里，我们链接的时候需要通过指定"-ldl"，这样才能使用动态加载技术提供的那些函数。

另外，注意：在源代码main.cpp中我们可以看到，这里使用dlsym的时候，采用的标号不是库文件源代码myfile.cpp中的printInfo，而实际是通过"nm libmy.so" 或者"readelf -s libmy.so"查询得知的，标号名称为："\_Z9printInfov"的函数。通过这样的方式，加载，可以不用包含库的头文件而使用库的函数。 请猜一猜，这个时候的软件，是绿色的吗？^\_^

[其它]

1, nm命令可以查可能一个库中的符号

nm列出的符号有很多，常见的有三种，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。

\*查看库中所有的符号

$nm libhello.so

\*假设开发者希望知道上文提到的 hello库中是否定义了 printf():

$nm libhello.so |grep printf

U printf

U表示符号printf被引用，但是并没有在函数内定义，由此可以推断，要正常使用hello库，必须有其它库支持，再使用ldd命令查看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.

2，ldd命令可以查询一个程序依赖哪些共享库：

\*查看一个程序依赖那些共享库：

[root@lv-k test]# ldd a.out

linux-gate.so.1 =>  (0x00322000)

libdl.so.2 => /lib/libdl.so.2 (0x002f3000)

libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x025ab000)

libm.so.6 => /lib/libm.so.6 (0x002c8000)

libgcc\_s.so.1 => /lib/libgcc\_s.so.1 (0x0013d000)

libc.so.6 => /lib/libc.so.6 (0x0016d000)

/lib/ld-linux.so.2 (0x0014e000)

3，使用readelf可以查看一个文件是否是可执行文件，是否是库文件。

一般而言，Linux中的可执行文件和库文件的都是elf的二进制文件，所以就以elf文件为例，而这个命令也是针对elf格式文件的。

readelf更多的命令选项应当查看man手册，这里给出几个简单的例子。

\*查看一个文件是否为库文件：

[root@lv-k test]# readelf -h libmy.so

ELF Header:

Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00

Class:                             ELF32

Data:                              2's complement, little endian

Version:                           1 (current)

OS/ABI:                            UNIX - System V

ABI Version:                       0

Type:                              DYN (Shared object file)

Machine:                           Intel 80386

Version:                           0x1

Entry point address:               0x550

Start of program headers:          52 (bytes into file)

Start of section headers:          2768 (bytes into file)

Flags:                             0x0

Size of this header:               52 (bytes)

Size of program headers:           32 (bytes)

Number of program headers:         5

Size of section headers:           40 (bytes)

Number of section headers:         27

Section header string table index: 24

这里，Type字段就指明了DYN类型，就是共享目标文件，实际lib\*.so就属于这个类型。

\*查看一个可执行文件：

[root@lv-k test]# readelf -h a.out

ELF Header:

Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00

Class:                             ELF32

Data:                              2's complement, little endian

Version:                           1 (current)

OS/ABI:                            UNIX - System V

ABI Version:                       0

Type:                              EXEC (Executable file)

Machine:                           Intel 80386

Version:                           0x1

Entry point address:               0x80486e0

Start of program headers:          52 (bytes into file)

Start of section headers:          3728 (bytes into file)

Flags:                             0x0

Size of this header:               52 (bytes)

Size of program headers:           32 (bytes)

Number of program headers:         8

Size of section headers:           40 (bytes)

Number of section headers:         29

Section header string table index: 26

这里，Type字段就指明了EXEC类型，就是可执行文件。

\*查看一个静态库文件：

[root@lv-k test]# readelf -h libmy.a

File: libmy.a(myfile.o)

ELF Header:

Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00

Class:                             ELF32

Data:                              2's complement, little endian

Version:                           1 (current)

OS/ABI:                            UNIX - System V

ABI Version:                       0

Type:                              REL (Relocatable file)

Machine:                           Intel 80386

Version:                           0x1

Entry point address:               0x0

Start of program headers:          0 (bytes into file)

Start of section headers:          516 (bytes into file)

Flags:                             0x0

Size of this header:               52 (bytes)

Size of program headers:           0 (bytes)

Number of program headers:         0

Size of section headers:           40 (bytes)

Number of section headers:         15

Section header string table index: 12

这里，Type字段就指明了REL类型，就是可重定位文件，实际静态库文件就是这个类型。

\*查看一个目标文件：

[root@lv-k test]# readelf -h myfile.o

ELF Header:

Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00

Class:                             ELF32

Data:                              2's complement, little endian

Version:                           1 (current)

OS/ABI:                            UNIX - System V

ABI Version:                       0

Type:                              REL (Relocatable file)

Machine:                           Intel 80386

Version:                           0x1

Entry point address:               0x0

Start of program headers:          0 (bytes into file)

Start of section headers:          516 (bytes into file)

Flags:                             0x0

Size of this header:               52 (bytes)

Size of program headers:           0 (bytes)

Number of program headers:         0

Size of section headers:           40 (bytes)

Number of section headers:         15

Section header string table index: 12

这里，Type字段就指明了REL类型，就是可重定位文件，可见，目标文件类型和静态库一样，实际我们可以将目标文件当做静态库来用。

\*查看一个普通文件：

[root@lv-k test]# readelf -h myfile.cpp

readelf: Error: Unable to read in 0x7473 bytes of section headers

readelf: Error: Not an ELF file - it has the wrong magic bytes at the start

可见，如果不是elf格式的文件，就会输出错误信息。

更多信息需要对elf格式有一定的了解。

至此，已经将Linux上面库文件相关的原理以及实现过程简单进行了介绍，

比较好的参考资料：

http://blog.chinaunix.net/u/19573/showart\_1822303.html

**作者：QuietHeart**

**Email：quiet\_heart000@126.com**

**日期：2011年7月5日**