更多嵌入式 Linux 学习资料，请关注：一口 Linux 回复关键字:1024


个人微信

公众号

一、什么是库?

在 windows 平台和 linux 平台下都大量存在着库。一般是软件作者为了发布方便、替换方便或二次开发目的，而发布的一组可以单独与应用程序进行 compile time 或 runtime 链接的二进制可重定位目标码文件。
本质上来说库是一种可执行代码的二进制形式，这个文件可以在编译时由编译器直接链接到可执行程序中，也可以在运行时由操作系统的 runtime enviroment 根据需要动态加载到内存中。
一组库，就形成了一个发布包，当然，具体发布多少个库，完全由库提供商自己决定。
由于 windows 和 linux 的本质不同，因此二者库的二进制是不兼容的。
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。
本文仅讨论 linux 下的库。

二、库的分类

库有两种：静态库和共享库（动态库）。

```
win32 平台下，静态库通常后缀为.lib，动态库为.dll ；
linux 平台下，静态库通常后缀为.a，动态库为.so 。
```

从本质上来说，由同一段程序编译出来的静态库和动态库，在功能上是没有区别的。不同之处仅仅在于其名字上，也就是“静态”和“动态”。
二者均以文件的形式存在，其本质上是一种可执行代码的二进制格式，可以被载入内存中执行。 无论是动态链接库还是静态链接库，它们无非是向其调用者提供变量、函数和类。

1. 静态库

所谓静态库，就是在静态编译时由编译器到指定目录寻找并且进行链接，一旦链接完成，最终的可执行程序中就包含了该库文件中的所有有用信息，包括代码段、数据段等。

2. 动态库

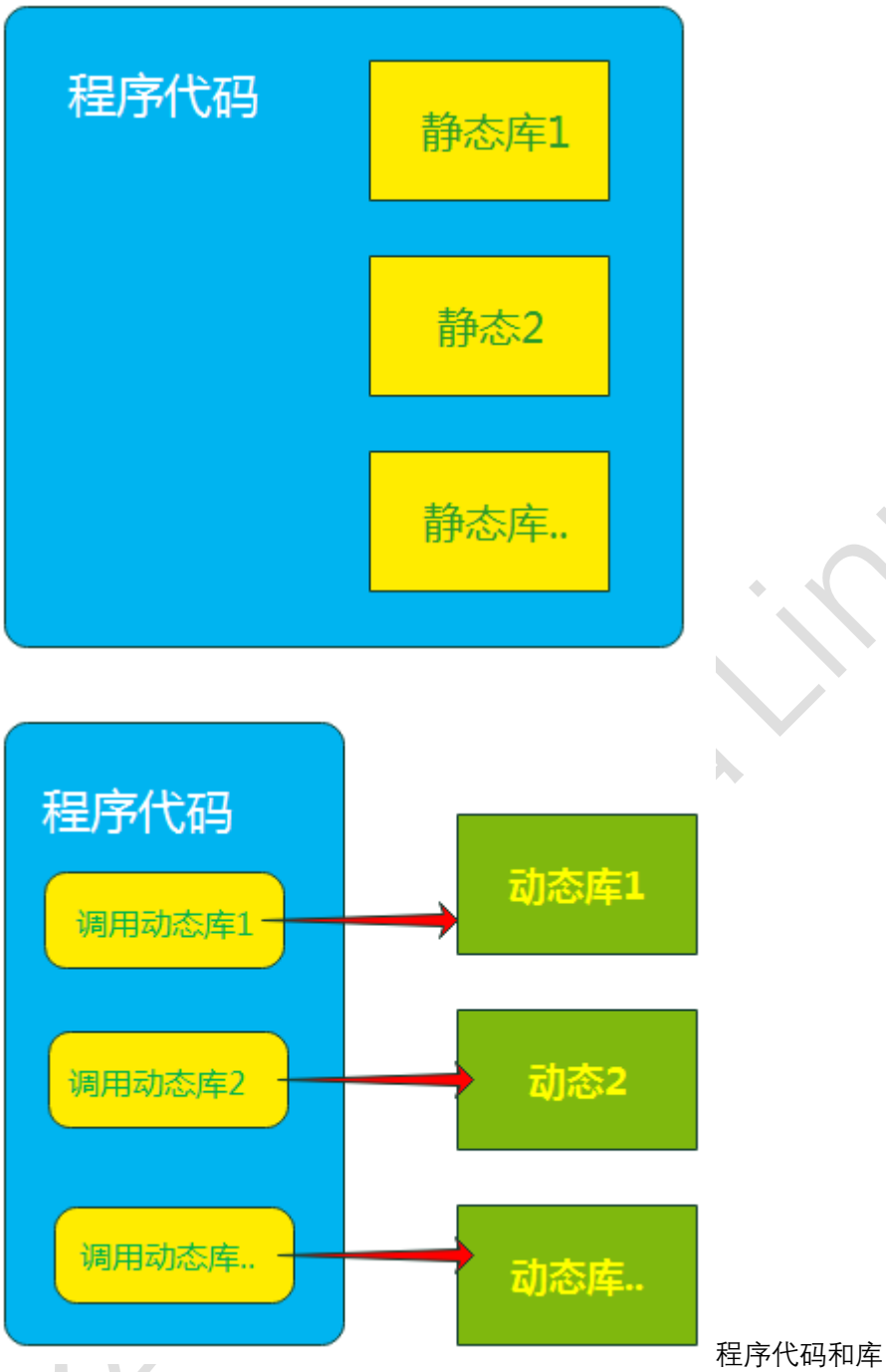
所谓动态库，就是在应用程序运行时，由操作系统根据应用程序的请求，动态到指定目录下寻找并装载入内存中，同时需要进行地址重定向。

3. 区别

我们以**编译链接、载入时刻**两点来讨论静态库和动态库的区别。

编译链接

静态链接库在程序编译时会被链接到目标代码中，目标程序运行时将不再需要改动态库，移植方便，体积较大，浪费控件和资源，因为所有相关的对象文件与牵涉到库都被链接合成一个可执行文件，这样导致可执行文件的体积较大。
动态库在程序编译时并不会被链接到目标代码中，而是在程序运行时才被载入，因为可执行文件体积较小。有了动态库，程序的升级会相对比较简单，比如某个动态库升级了，只需要更换这个动态库的文件，而不需要去更换可执行文件。但要注意的是，可执行程序在运行时需要能找到动态库文件。可执行文件时动态库的调用者。



程序代码和库

载入时刻

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。

4. 优缺点

相对于动态库，静态库的优点在于直接被链接进可执行程序中，之后，该可执行程序就不再依赖于运行环境的设置了（当然仍然会依赖于 CPU 指令集和操作系统支持的可执行文件格式等硬性限制）。
而动态库的优点在于，用户甚至可以在程序运行时随时替换该动态库，这就构成了动态插件系统的基础。具体使用静态库和动态库，由程序员根据需要自己决定。
另外，需要说明的一点是，从底层实现上，动态库的效率可能会比静态库稍差一点点，注意，这里用了“可能”二字，具体差不差，还得看写程序的人。之所以可能会差，主要原因在于，程序总无法直接调用动态库中的函数符号，而只能通过调用操作系统的 runtime enviroment 接口来动态载入某个函数符号，同时获得该函数符号在内存中的地址，将其保存为函数指针进行调用，这就在函数调用时增加了一次间接寻址的过程。

三、库文件的制作

1. 库文件命名

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.x.y.z，含义如下图所示：

2. 制作库文件常用参数

首先需要了解 gcc 编译库要用到一些参数，很重要。

参数	含义
-shared	指定生成动态链接库。

参数	含义
-static	指定生成静态链接库。
-fPIC	表示编译为位置独立的代码，用于编译共享库。目标文件需要创建成位置无关码，概念上就是在可执行程序装载它们的时候，它们可以放在可执行程序的内存里的任何地方。
-L	表示要连接的库在当前目录中。
-l	指定链接时需要的动态库。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。
-Wall	生成所有警告信息。
-ggdb	此选项将尽可能的生成 gdb 的可以使用的调试信息。
-g	编译器在编译的时候产生调试信息。
-c	只激活预处理、编译和汇编,也就是把程序做成目标文件(.o 文件)。
-Wl,options	把参数(options)传递给链接器 ld。如果 options 中间有逗号,就将 options 分成多个选项,然后传递给链接程序。

3. 库源文件

假定我们要将以下两个文件制作成库文件 add.c

```
int add(int x,int y)
{
    return x+y;
}
int sub(int x,int y)
{
    return x-y;
}
```

add.h

```
int add(int x,int y);
int sub(int x,int y);
```

4. 制作静态库并使用

1. 需要把 add.c 编译成.o 文件

```
gcc -c add.c
```

2. 使用 ar 命令生成静态库 libadd.a

```
ar -rc libadd.a add.o        遵循静态库命名的规则 lib + 名字 + .a
```

3. 使用静态库 要是用静态库 libadd.a，只需要包含 add.h,就可以使用函数 add()、sub()。

```
#include <stdio.h>
#include "add.h"
void main()
{
    printf("add(5,4) is %d\n",add(5,4));
    printf("sub(5,4) is %d\n",sub(5,4));
}
```

静态库的文件可以放在任意的位置，编译时只需要找到该库文件即可。

```
gcc test.c -o run libadd.a
```

4. 库和头文件如果在其他目录下

使用一下命令编译：

```
gcc -c -I /home/xxxx/include test.c //假设test.c要使用对应的静态库
gcc -o test -L /home/xxxxx/lib test.o libadd.a
```

或者

```
gcc -c -I /home/xxxx/include -L /home/xxxxx/lib  libadd.a test.c
```

1）. 通过-I(是大 i)指定对应的头文件 2）. 通过-L 制定库文件的路径,libadd.a 就是要用的静态库。 3）. 在 test.c 中要包含静态库的头文件。

5. 制作动态库并使用

1. 把 add.c 编译成动态链接库 libadd.so

```
gcc -fPIC -o libadd.o -c add.c
gcc -shared -o libadd.so libadd.o
```

也可以直接使用一条命令

```
gcc -fPIC -shared -o libadd.so add.c
```

2. 动态库的安装 通常动态库拷贝到/lib 下即可:

```
sudo cp libadd.so /lib
```

3. 使用动态库

```
#include <stdio.h>
#include "add.h"
void main()
{
    printf("add(5,4) is %d\n",add(5,4));
    printf("sub(5,4) is %d\n",sub(5,4));
}
```

编译动态库：

```
gcc static -o run  -ladd
```

注意观察编译时动态库的名字与库文件对应关系

```
libadd.so<--------->-ladd
```

去掉 .so，lib 简化成 l，其他字母保留。

6. 动态加载的函数库 Dynamically Loaded (DL) Libraries

动态加载的函数库 Dynamically loaded (DL) libraries 是一类函数库，它可以在程序运行过程中的任何时间加载。它们特别适合在函数中加载一些模块和 plugin 扩展模块的场合，因为它可以在当程序需要某个 plugin 模块时才动态的加载。

Linux 系统下，DL 函数库与其他函数库在格式上没有特殊的区别，它们创建的时候是标准的 object 格式。主要的区别就是这些函数库不是在程序链接的时候或者启动的时候加载，而是通过一个 API 来打开一个函数库，寻找符号表，处理错误和关闭函数库。通常 C 语言环境下，需要包含这个头文件。

dlopen()

dlopen 函数打开一个函数库然后为后面的使用做准备。C 语言原形是：

```
 void * dlopen(const char *filename, int flag);
 参数

filename
如果文件名 filename 是以“/”开头，也就是使用绝对路径，那么 dlopne 就直接使用它，而不去查找某些环境变量或者系统设置的函数库所在的目录了。否则 dlopen（）就会按照下面的次序查找函数库文件：
1. 环境变量 LD_LIBRARY 指明的路径。
2. /etc/ld.so.cache 中的函数库列表。
3. /lib 目录，然后/usr/lib。
```

一些很老的 a.out 的 loader 则是采用相反的次序，也就是先查 /usr/lib，然后是/lib。

flag
的值必须是 RTLD_LAZY 或者 RTLD_NOW，RTLD_LAZY 的意思是
resolve undefined symbols as code from the dynamic library is executed，而 RTLD_NOW 的含义是
resolve all undefined symbols before dlopen() returns and fail if this cannot be done'。
返回值
dlopen（）函数的返回值是一个句柄，然后后面的函数就通过使用这个句柄来做进一步的操作。如果打开失败 dlopen()就返回一个 NULL。如果一个函数库被多次打开，它会返回同样的句柄。

如果有好几个函数库，它们之间有一些依赖关系的话，例如 X 依赖 Y，那么你就要先加载那些被依赖的函数。 例如先加载 Y，然后加载 X。

dlerror()

通过调用 dlerror()函数，我们可以获得最后一次调用 dlopen()，dlsym()，或者 dlclose（）的错误信息。

dlsym()

如果你加载了一个 DL 函数库而不去使用当然是不可能的了，使用一个 DL 函数库的最主要的一个函数就是 dlsym()，这个函数在一个已经打开的函数库里面查找给定的符号。这个函数如下定义：

```
 void * dlsym(void *handle, char *symbol);
```

参数
handle
就是由 dlopen 打开后返回的句柄，
symbol
是一个以 NIL 结尾的字符串。
功能：
如果 dlsym()函数没有找到需要查找的 symbol，则返回 NULL。如果你知道某个 symbol 的值不可能是 NULL 或者 0，那么就很好，你就可以根据这个返回结果判断查找的 symbol 是否存在了；不过，如果某个 symbol 的值就是 NULL，那么这个判断就有问题了。标准的判断方法是先调用 dlerror()，清除以前可能存在的错误，然后调用 dlsym（）来访问一个 symbol，然后再调用 dlerror（）来判断是否出现了错误。

dlclose()

dlopen()函数的反过程就是 dlclose（）函数，dlclose（）函数用力关闭一个 DL 函数库。 Dl 函数库维持一个资源利用的计数器，当调用 dlclose 的时候，就把这个计数器的计数减一，如果计数器为 0，则真正的释放掉。真正释放的时候，如果函数库里面有_fini()这个函数，则自动调用_fini（）这个函数，做一些必要的处理。 Dlclose（）返回 0 表示成功，其他非 0 值表示错误。

举例

```
#include <stdio.h>
#include <dlfcn.h>
void main()
{
    int (*add)(int x,int y);
    int (*sub)(int x,int y);
    void *libptr;
    libptr=dlopen("./libadd.so",RTLD_LAZY); //加载动态库
    add=dlsym(libptr,"add"); //获取函数地址
    sub=dlsym(libptr,"sub");
    printf("add(5,4) is %d\n",add(5,4));
    printf("sub(5,4) is %d\n",sub(5,4));
    dlclose(libptr);
}
```

四、库的两个查看命令

1. 查看依赖库命令 ldd

使用 ldd 命令可以查看一个可执行程序依赖哪些库。
这个命令非常有用，实际工作中经常会一直各种库，而有些程序的执行需要依赖好几种库，各种库的版本又很多历史版本，经常会出现库不兼容的情况，我们需要根据实际情况，适当的降低版本或者升级版本。
例如：

```
zhh@ubuntu:~/test$ ldd /lib/x86_64-linux-gnu/libpthread-2.23.
        linux-vdso.so.1 =>  (0x00007ffc09565000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007
        /lib64/ld-linux-x86-64.so.2 (0x00007fb3a0745000)
```

可以看到线程库 libpthread-2.23.so 依赖于 libc 库和 ld-linux 库。

2. nm

nm 工具可以打印出库中的涉及到的所有符号，下面是我们查看我们创建的动态库 libadd.a：

```
zhh@ubuntu:~/test$ nm libadd.a

add.o:
0000000000000000 T add
0000000000000014 T sub
```

nm

五、库的安装

在新安装一个库之后如何让系统能够找到他，有以下几种方法：

1. 拷贝到/lib 或者/usr/lib 下

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

2.通过配置文件/etc/profile

永久生效的环境变量设置，编辑/etc/profile 即可。

```
vi /etc/profile
```

在文件里末尾加上对应的环境变量信息。

动态库环境变量设置：

```
export LD_LIBRARY_PATH=/home/peng/mylib/
```

/home/peng/mylib/指的是动态库文件夹所在位置。即，.so 等文件在/home/peng/mylib/下。

编辑完成，保存编辑并退出；使配置即时生效：

```
source /etc/profile
```

3./etc/ld.so.conf

编辑/etc/ld.so.conf 文件，加入库文件所在目录的路径

```
vim /etc/ld.so.conf
```

在里面添加动态库所在路径即可，例如

```
/usr/local/lib/
```

运行 ldconfig，该命令会重建/etc/ld.so.cache 文件

七、常见库的移植

1. jpeg 库,用于 jpeg 图像处理

下载地址：

http://www.ijg.org/files/

解压

```
tar xvzf jpegsrc.v6b.tar.gz
cd jpeg-6b
```

生成 Makefile

```
./configure --host=arm-linux-gnueabihf --prefix=$PWD/temp_install
```

编译， 安装

```
make
make install
```

注意这个库的安装程序有 BUG,不会自动创建发布的 lib,include,man 等,因此要手工创建,要不先把其它库做好,再安装这个库

```
mkdir -p /home/peng/jpeg-6b/temp_install/include
mkdir -p /home/peng/jpeg-6b/temp_install/lib
mkdir -p /home/peng/jpeg-6b/temp_install/man/man1
```