静态库和动态库

概述

- 什么是库
- 静态库
- 动态库
- 加载库
- 创建和安装共享库

什么是库

- 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的本质不同，因此二者库的二进制是不兼容的。
- 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，因此体积较大。
  - 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，因此代码体积较小。

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。

静态库

- 静态库对函数库的链接是放在编译时期（compile time）完成的。
  - 程序在运行时与函数库再无瓜葛，移植方便
  - 浪费空间和资源，因为所有相关的对象文件（object file）与牵涉到的函数库（library）被链接合成一个可执行文件（executable file）。

如何创建静态库

- 创建静态库用ar命令，它将很多.o转换成.a
  - # ar crs libmyhello.a hello.o
  - 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。
- 使用静态链接库
  - # gcc -o hello main.c -L. -lmyhello

创建静态库示例

- 在Linux下创建和使用静态库的步骤如下：
  - 1、编写源代码：
    - 假设有两个源文件lib.c和lib.h。

```
//lib.c
#include <stdio.h>
#include "lib.h"

void my_function() {
    // 实现功能
    printf(“hello world\n”);
}
```

```
//lib.h
#ifndef LIB_H
#define LIB_H

void my_function();

#endif // LIB_H
```

- 2、编译源代码：
  - 使用gcc将源代码编译成目标文件。
    - gcc -c lib.c -o lib.o
- 3、创建静态库：
  - 通过ar命令将编译好的目标文件打包成静态库。
    - ar rcs libmylib.a lib.o
    - 参数r：在库中插入模块(替换)。当插入的模块名已经在库中存在，则替换同名的模块。如果若干模块中有一个模块在库中不存在，ar显示一个错误消息，并不替换其他同名模块。默认的情况下，新的成员增加在库的结尾处，可以使用其他任选项来改变增加的位置。
    - 参数c：创建一个库。不管库是否存在，都将创建。
    - 参数s：创建目标文件索引，这在创建较大的库时能加快时间。（补充：如果不需要创建索引，可改成大写S参数；如果.a文件缺少索引，可以使用ranlib命令添加）

- 4、使用静态库：
  - 创建一个使用静态库中函数的测试程序main.c。

```
#include "lib.h"
 int main() {
    my_function();//调用静态库里的函数
    return 0;
}
```

- 5、编译并链接静态库：
  - gcc main.c -lmylib -L. -o myprogram
  - 说明：
    - -lmylib 指定链接库名称为mylib（去掉前缀lib和后缀.a），
    - -L.指定查找库的路径为当前目录。
- 6、运行程序：
  - ./myprogram

共享库（动态库）

- 动态库把对一些库函数的链接载入推迟到程序运行的时期（runtime）。
  - 可以实现进程之间的资源共享。
  - 将一些程序升级变得简单。
  - 甚至可以真正做到链接载入完全由程序员在程序代码中控制。

动态库的命名规则

- 动态链接库的名字形式为 “libxxx.so” 后缀名为 “.so”
- 针对于实际库文件，每个共享库都有个特殊的名字 “soname”。在程序启动后，程序通过这个名字来告诉动态加载器该载入哪个共享库。
- 在文件系统中，soname仅是一个链接到实际动态库的链接。
- 对于动态库而言，每个库实际上都有另一个名字给编译器来用。它是一个指向实际库镜像文件的链接文件。这个时候soname是没有版本号的。

如何创建共享库

- 我们用gcc来创建共享库
  - #gcc -fPIC -Wall -c hello.c
  - # gcc -shared -o libmyhello.so hello.o
    - -fPIC 创建与地址无关的编译程序

创建共享库（动态库）示例

- 在Linux下创建和使用动态库的步骤如下：
  - 1、编写源代码：
    - 假设有两个源文件lib.c和lib.h。

```
//lib.c
#include <stdio.h>
#include "lib.h"

void my_function() {
    // 实现功能
    printf(“hello world\n”);
}
```

```
//lib.h
#ifndef LIB_H
#define LIB_H

void my_function();

#endif // LIB_H
```

- 2、编译源代码：
  - 使用gcc将源代码编译成目标文件。
    - gcc -c lib.c -o lib.o
- 3、创建动态库：
  - 通过gcc -shared 命令将编译好的目标文件打包成动态库。
    - gcc -shared -o libmyhello.so lib.o
    - gcc -shared：编译生成动态库
    - -fPIC 创建与地址无关（地址无关代码（Position Independent Code, PIC））的编译程序
    - -fPIC大概的原理就是：编译时构造全局偏移表（Global Offset Table, GOT），运行时通过GOT中存储的偏移值访问指令和数据。

- 4、使用动态库：
  - 创建一个使用动态库中函数的测试程序main.c。

```
#include "lib.h"
 int main() {
    my_function();//调用静态库里的函数
    return 0;
}
```

- 5、编译main.c：
  - gcc -o main main.c -lmyhello -L.
  - 说明：
    - -lmyhello 指定动态库名称为myhello（去掉前缀lib和后缀.so），
    - -L.指定查找库的路径为当前目录。

- 6、运行程序
  - 增加动态库环境变量的路径：
    - export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/a/mywork/c-program/001/day003/lib_dynamic
    - ./main

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 。

如何让系统能够找到它

- 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.conf.d/*.conf文件中，步骤如下：
  - 1.比如：新建并编辑 /etc/ld.so.conf.d/my.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.conf.d/my.conf文件

动态库的路径问题

- 为了让执行程序顺利找到动态库，有三种方法：
  - (1) 把库拷贝到/usr/lib和/lib目录下。
  - (2) 在LD_LIBRARY_PATH环境变量中加上库所在路径。
  - (3) 添加/etc/ld.so.conf.d/*.conf文件，把库所在的路径加到文件末尾，并执行ldconfig刷新。
    这样，加入的目录下的所有库文件都可见。

linux c之动态打开链接库(dlopen dlsym dlclose)

- 1、 linux提供了加载和处理动态链接库的系统调用
- 2、主要函数
  - 1) dlopen
    - 函数格式：void *dlopen(const char *filename,int flag)
      - filename： 是动态链接库的名称;
      - flag：打开模式如下：
        - RTLD_LAZY 暂缓决定，等有需要时再解出符号
        - RTLD_NOW 立即决定，在dlopen函数返回前解除所有未决定的符号。
      - 返回值：为动态库的指针
    - 函数功能：dlopen以指定模式打开指定的动态库文件，并返回一个句柄给调用进程
    - 示例：
      - void *handle;
      - handle = dlopen("/home/a/mywork/c-program/001/day003/so/libadd.so", RTLD_LAZY);

- 2）dlsym
  - 函数格式：void*dlsym(void *handle,const*symbol）
    - handle：调用dlopen()后获得的句柄指针
    - symbol：是要调用库的函数名称。
    - 返回值;为函数指针
  - 函数功能：用于获得动态库的函数地址
  - 示例：
    - int (*myadd)(int , int);
    - void *handle;
    - handle = dlopen("/home/a/mywork/c-program/001/day003/so/libadd.so", RTLD_LAZY);
    - myadd=dlsym(handle, "myadd");//拿到libadd.so中函数myadd()的入口地址

- 3）dlclose
  - 函数格式：int dlclose(void *handle)
    - handle：调用dlopen()后获得的句柄指针
    - 返回值为0表示成功，否则失败。
  - 函数功能：用于关闭指定句柄的动态链接库，只有当此动态链接库的使用计数为0时,才会真正被系统卸载。
  - 示例：
    - dlclose(handle);

- 4）dlerror
  - 函数格式：void*dlerror(void)
    - 返回值：返回值表示上一次调用是否成功，如果成功，返回值是NULL，如果失败，返回的是char*类型错误信息。
  - 函数功能：用于返回出错信息。
  - 示例：
    - handle = dlopen("/home/a/mywork/c-program/001/day003/so/libadd.so", RTLD_LAZY);
    - if(!handle)
    - {
    - 	fputs(dlerror(), stderr);
    - 	exit(1);
    - }

查看库中的符号

- 1、 nm命令
  - 命令格式： nm [option(s)] [file(s)]
    - 有用的options：
      - -A 在每个符号信息的前面打印所在对象文件名称；
      - -C 输出demangle过了的符号名称；
      - -D 打印动态符号；
      - -l 使用对象文件中的调试信息打印出所在源文件及行号；
      - -n 按照地址/符号值来排序；
      - -u 打印出那些未定义的符号。
  - 命令描述：列出 .o, .a, .so 中的符号信息，包括诸如符号的值、符号类型以及符号名称等。所谓符号，通常指定义出的函数、全局变量等等。

- nm列出的符号有很多，常见的有：
  - A 该符号的值在今后的链接中将不再改变；
  - B 该符号放在BSS段中，通常是那些未初始化的全局变量；
  - D 该符号放在普通的数据段中，通常是那些已经初始化的全局变量；
  - 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；
  - 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。

- 示例1）：

```
a@ubuntu:~/mywork/c-program/001/day003/so$ nm ./libadd.so
0000000000201020 B __bss_start
0000000000201020 b completed.7698
                 w __cxa_finalize
00000000000004a0 t deregister_tm_clones
0000000000000530 t __do_global_dtors_aux
0000000000200e88 t __do_global_dtors_aux_fini_array_entry
0000000000201018 d __dso_handle
0000000000200e90 d _DYNAMIC
0000000000201020 D _edata
0000000000201028 B _end
0000000000000590 T _fini
0000000000000570 t frame_dummy
0000000000200e80 t __frame_dummy_init_array_entry
0000000000000638 r __FRAME_END__
0000000000201000 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
000000000000059c r __GNU_EH_FRAME_HDR
0000000000000460 T _init
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
000000000000057a T myadd
00000000000004e0 t register_tm_clones
0000000000201020 d __TMC_END__
a@ubuntu:~/mywork/c-program/001/day003/so$
```

- 示例2）：
  - （1）nm -u hello.o
    - 显示 hello.o 中的未定义符号，需要和其他对象文件进行链接。
  - （2）nm -A /usr/lib/* 2>/dev/null | grep “T memset”
    - 在 /usr/lib/ 目录下找出哪个库文件定义了 memset 函数。

- 2、ldd命令
  - 命令格式：ldd [选项] <可执行文件或共享库>
  - 功能描述：ldd命令全称为list dynamic dependencies（列出动态依赖）。ldd命令显示一个可执行文件或共享库所依赖的动态链接库列表。它会递归地检查文件所依赖的所有库，并显示它们的路径。通过ldd命令可以了解一个程序运行所需的库文件，以及这些库文件是否存在、版本是否匹配等信息。
  - ldd参数说明
    - --help 获取指令帮助信息；
    - --version 打印指令版本号；
    - -d,--data-relocs 执行重定位和报告任何丢失的对象；
    - -r, --function-relocs 执行数据对象和函数的重定位，并且报告任何丢失的对象和函数；
    - -u, --unused 打印未使用的直接依赖；
    - -v, --verbose 详细信息模式，打印所有相关信息；

- 示例1）：

```
a@ubuntu:~/mywork/c-program/001/day003/so$ ls
libadd.c  libadd.o  libadd.so  main  main.c
a@ubuntu:~/mywork/c-program/001/day003/so$ ldd main
        linux-vdso.so.1 (0x00007ffce144e000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fd73c2f1000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fd73bf00000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fd73c6f7000)
a@ubuntu:~/mywork/c-program/001/day003/so$
```

  - 输出解读：
    - 对于每个所依赖的库，ldd会显示它的路径，并用以下格式标记其状态：
    - => 文件路径：正常找到并链接。
    - => not found：未找到该库文件。
    - => version mismatch：版本不匹配。
    - => incompatible：与可执行文件或其他库不兼容。
    - => symbol not found：找不到某个符号。

- 示例2）：
  - (1) 查看可执行文件所依赖的库：
    - ldd /path/to/executable
  - (2) 查看共享库的依赖关系：
    - ldd /path/to/shared_library.so
  - (3) 显示详细的依赖库信息：
    - ldd -v /path/to/executable
  - (4) 只显示未使用的直接依赖库：
    - ldd -u /path/to/executable
  - (5) 显示函数和数据的重定位信息：
    - ldd -r /path/to/executable

gcc一些参数解析

- -shared：指定生成动态链接库。
- -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。