# 动态库和静态库

由 enlai.feng创建于二月 05, 2024

在linux编程中，动态库和静态库是两种不同类型的库文件，用于将代码模块化，以便可以在多个程序之间共享代码。

## 目标文件

**在解释静态库和动态库之前，需要简单了解一下什么是目标文件。目标文件常常按照特定格式来组织，在linux下，它是ELF格式（Executable Linkable Format，可执行可链接格式），而在windows下是PE（Portable Executable，可移植可执行）。**

而通常目标文件有三种形式：

- **可执行目标文件。**即我们通常所认识的，可直接运行的二进制文件。
- **可重定位目标文件。**包含了二进制的代码和数据，可以与其他可重定位目标文件合并，并创建一个可执行目标文件。
- **共享目标文件。**它是一种在加载或者运行时进行链接的特殊可重定位目标文件。

一个简单实例：

```
//main.c
#include<stdio.h>
#include<math.h>
int main(int argc,char *argv[])
{
    printf("hello 编程珠玑\n");
    int b = 2;
    double a = exp(b);
    printf("%lf\n",a);
    return 0;
}
```

- 上述代码计算e的2次方并打印结果。由于代码中用到了exp函数，它位于数学库libm.so或者libm.a中，因此编译时需要加上-lm。
- 生成可重定位目标文件main.o

```
$ gcc -c main.c   #生成可重定位目标文件
$ readelf -h main.o  #查看elf文件头部信息
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
(省略其他内容)
```

通过上面的命令将main.c生成为可重定位目标文件。通过readelf命令也可以看出来：REL (Relocatable file)。

观察共享目标文件libm.so：

```
$ readelf -h /lib/x86_64-linux-gnu/libm.so.6
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
  ABI Version:                       0
  Type:                              DYN (Shared object file)
（省略其他内容）
```

不同系统中libm.so的位置可能不一样，你可以通过locate命令来查找。locate命令的用法可参考《Linux中的文件查找技巧》。从结果可以看到，libm.so是**共享目标文件**（Shared object file）。

查看可执行目标文件main：

```
$ gcc -o main main.o -lm  #编译成最终的可执行文件
$ readelf -h main         #查看ELF文件头
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
(省略其他内容)
```

**如果使用到的函数没有在libc库中，那么你就需要指定要链接的库**，本文中需要链接libm.so或libm.a。可以看到，最终生成的main类型是Executable file，即**可执行目标文件。**

## 静态库（Static Libraries）

静态库通常以**.a（archive）**扩展名结尾。它们是一组预编译的目标文件（.o文件），这些文件在程序编译时被整合（链接）到最终的可执行文件中。当你编译一个程序并链接到一个静态库时，库中的代码会被复制到最终的可执行文件中。

```
$ gcc -c main.c
$ gcc -static -o main main.o -lm
```

在这个过程中，就会用到系统中的静态库libm.a。这个过程做了什么呢？首先第一条命令会将main.c编译成**可重定位目标文件main.o**，第二条命令的static参数，告诉链接器应该使用静态链接，-lm参数表明链接libm.a这个库（类似的，如果要链接libxxx.a,使用-lxxx即可）。由于main.c中使用了libm.a中的exp函数，因此链接时，会将libm.a中需要的代码“拷贝”到最终的可执行文件main中。

特别注意，必须把-lm放在后面。放在最后时它是这样的一个解析过程：

- 链接器从左往右**扫描可重定位目标文件**和**静态库**
- 扫描main.o时，发现一个未解析的符号exp，记住这个未解析的符号
- 扫描libm.a，找到了前面未解析的符号，因此提取相关代码
- 最终没有任何未解析的符号，**编译链接完成**

那如果将-lm放在前面，又是怎样的情况呢？

- 链接器从左往右扫描可重定位目标文件和静态库
- 扫描libm.a，由于前面**没有任何未解析的符号**，因此**不会提取任何代码**
- 扫描main.o，发现未解析的符号exp
- 扫描结束，还有一个未解析的符号，因此**编译链接报错**

**如果把-lm放在前面，编译结果如下：**

```
$ gcc -static -lm -o main main.o
main.o: In function `main':
main.c:(.text+0x2f): undefined reference to `exp'
collect2: error: ld returned 1 exit status
```

**我们看看最终生成的文件大小：**

```
$ ls -lh main
-rwxrwxr-x 1 hyb hyb 988K 6月  27 20:22 main
```

**生成的可执行文件大小为988k。**

**由于最终生成的可执行文件中已经包含了exp相关的二进制代码，因此这个可执行文件在一个没有libm.a的linux系统中也能正常运行。**

**优点：**

- 由于所有必要的代码都包含在可执行文件中，因此在运行时不需要额外的依赖。
- 程序启动时可能更快，因为所有的代码都已经在本地可用。

**缺点：**

- 可执行文件的大小会更大，因为它包含了整个库的代码。

- 更新库需要重新编译和链接所有使用该库的程序。

## 动态库（Dynamic Libraries）

动态库在Linux中通常以**.so（shared object）扩展名结尾**。与静态库不同，动态库在程序运行时被加载。程序启动时，它不会包含库的代码；相反，它会有一个引用，指明在需要时从哪里加载库。动态库和静态库类似，但是它并不在链接时将需要的二进制代码都“拷贝”到可执行文件中，而是仅仅“拷贝”一些重定位和符号表信息，这些信息可以在程序运行时完成真正的链接过程。linux中通常以.so（shared object）作为后缀。

通常我们编译的程序**默认就是使用动态链接**：

```
$ gcc -o main main.c -lm  #默认使用的是动态链接
```

我们来看最终生成的文件大小：

```
$ ls -lh main
-rwxrwxr-x 1 hyb hyb 8.5K 6月  27 20:25 main
```

可以看到，通过动态链接的程序**只有8.5k!**

另外我们还可以通过ldd命令来观察可执行文件链接了哪些动态库：

```
$ ldd main
    linux-vdso.so.1 =>  (0x00007ffc7b5a2000)
    libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fe9642bf000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe963ef5000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fe9645c8000)
```

正因为我们并没有把libm.so中的二进制代码“拷贝”可执行文件中，**我们的程序在其他没有上面的动态库时，将无法正常运行。**

**优点：**

- 可执行文件更小，因为它们不包含整个库的代码，只包含需要在运行时加载库的引用。
- 库可以在不同的程序之间共享，节省内存。
- 更新库不需要重新编译链接所有程序，只需替换库文件即可。

**缺点：**

- 如果动态库在运行时不可用，程序将无法启动或运行。
- 程序启动时可能稍慢，因为需要加载外部库。

## 创建和使用静态库和动态库的基本命令：

**创建静态库：**

```
gcc -c foo.c bar.c      # 编译源文件到目标文件
ar rcs libfoobar.a foo.o bar.o  # 创建静态库
```

**创建动态库：**

```
gcc -fPIC -c foo.c bar.c      # 编译源文件到目标文件，生成位置无关代码
gcc -shared -o libfoobar.so foo.o bar.o  # 创建动态库
```

**链接静态库：**

```
gcc -o myprogram myprogram.c -L. -lfoobar  # 链接静态库到你的程序
```

**链接动态库：**

```
gcc -o myprogram myprogram.c -L. -lfoobar  # 链接动态库到你的程序
```

# 有什么区别

静态库被使用目标代码最终和可执行文件在一起（它只会有自己用到的），而动态库与它相反，它的目标代码在运行时或者加载时链接。正是由于这个区别，会导致下面所介绍的这些区别。

## 可执行文件大小不一样

从前面也可以观察到，静态链接的可执行文件要比动态链接的可执行文件要大得多，因为它将需要用到的代码从二进制文件中“拷贝”了一份，而动态库仅仅是复制了一些重定位和符号表信息。

## 占用磁盘大小不一样

如果有多个可执行文件，那么静态库中的同一个函数的代码就会被复制多份，而动态库只有一份，因此**使用静态库占用的磁盘空间相对比动态库要大。**

## 扩展性与兼容性不一样

如果静态库中某个函数的实现变了，那么可执行文件必须重新编译，而对于动态链接生成的可执行文件，只需要更新动态库本身即可，不需要重新编译可执行文件。正因如此，**使用动态库的程序方便升级和部署。**

## 依赖不一样

**静态链接的可执行文件不需要依赖其他的内容即可运行，而动态链接的可执行文件必须依赖动态库的存在。**所以如果你在安装一些软件的时候，提示某个动态库不存在的时候也就不奇怪了。

即便如此，**系统中一般存在一些大量公用的库**，所以使用动态库并不会有什么问题。

## 复杂性不一样

相对来讲，动态库的处理要比静态库要复杂，例如，如何在运行时确定地址？多个进程如何共享一个动态库？当然，作为调用者我们不需要关注。另外动态库版本的管理也是一项技术活。这也不在本文的讨论范围。

## 加载速度不一样

由于静态库在链接时就和可执行文件在一块了，而动态库在加载或者运行时才链接，因此，对于同样的程序，静态链接的要比动态链接加载更快。所以选择静态库还是动态库是空间和时间的考量。但是通常来说，牺牲这点性能来换取程序在空间上的节省和部署的灵活性时值得的。再加上**局部性原理**，牺牲的性能并不多。