**插件架构**

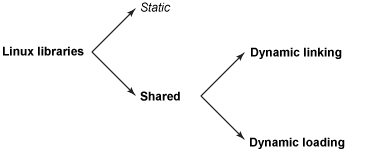
# Linux 动态库剖析

## 进程与 API

动态链接的共享库是 GNU/Linux® 的一个重要方面。该种库允许可执行文件在运行时动态访问外部函数，从而（通过在需要时才会引入函数的方式）减少它们对内存的总体占用。本文研究了创建和使用静态库的过程，详细描述了开发它们的各种工具，并揭秘了这些库的工作方式

库用于将相似函数打包在一个单元中。然后这些单元就可为其他开发人员所共享，并因此有了**模块化编程**这种说法 — 即，从模块中构建程序。Linux 支持两种类型的库，每一种库都有各自的优缺点。**静态库**包含在编译时静态绑定到一个程序的函数。**动态库**则不同，它是在加载应用程序时被加载的，而且它与应用程序是在运行时绑定的。图 1 展示了 Linux 中的库的层次结构。

图 1. Linux 中的库层次结构

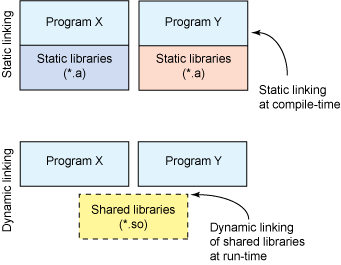


使用共享库的方法有两种：您既可以在运行时动态链接库，也可以动态加载库并在程序控制之下使用它们。本文对这两种方法都做了探讨。

静态库较适宜于较小的应用程序，因为它们只需要最小限度的函数。而对于需要多个库的应用程序来说，则适合使用共享库，因为它们可以减少应用程序对内存（包括运行时中的磁盘占用和内存占用）的占用。这是因为多个应用程序可以同时使用一个共享库；因此，每次只需要在内存上复制一个库。要是静态库的话，每一个运行的程序都要有一份库的副本。

GNU/Linux 提供两种处理共享库的方法（每种方法都源于 Sun Solaris）。您可以动态地将程序和共享库链接并让 Linux 在执行时加载库（如果它已经在内存中了，则无需再加载）。另外一种方法是使用一个称为**动态加载**的过程，这样程序可以有选择地调用库中的函数。使用动态加载过程，程序可以先加载一个特定的库（已加载则不必），然后调用该库中的某一特定函数（图 2 展示了这两种方法）。这是构建支持插件的应用程序的一个普遍的方法。我稍候将在本文探讨并示范该应用程序编程接口（API）。

图 2. 静态链接与动态链接



## 用 Linux 进行动态链接

现在，让我们深入探讨一下使用 Linux 中的动态链接的共享库的过程。当用户启动一个应用程序时，它们正在调用一个可执行和链接格式（Executable and Linking Format，ELF）映像。内核首先将 ELF 映像加载到用户空间虚拟内存中。然后内核会注意到一个称为 .interp 的 ELF 部分，它指明了将要被使用的动态链接器（/lib/ld-linux.so），如清单 1 所示。这与 UNIX® 中的脚本文件的解释器定义（#!/bin/sh）很相似：只是用在了不同的上下文中。

##### 清单 1. 使用 readelf 来显示程序标题

mtj@camus:~/dl$ **readelf -l dl**

Elf file type is EXEC (Executable file)

Entry point 0x8048618

There are 7 program headers, starting at offset 52

Program Headers:

Type Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align

PHDR 0x000034 0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4

INTERP 0x000114 0x08048114 0x08048114 0x00013 0x00013 R 0x1

[Requesting program interpreter: /lib/ld-linux.so.2]

LOAD 0x000000 0x08048000 0x08048000 0x00958 0x00958 R E 0x1000

LOAD 0x000958 0x08049958 0x08049958 0x00120 0x00128 RW 0x1000

DYNAMIC 0x00096c 0x0804996c 0x0804996c 0x000d0 0x000d0 RW 0x4

NOTE 0x000128 0x08048128 0x08048128 0x00020 0x00020 R 0x4

GNU\_STACK 0x000000 0x00000000 0x00000000 0x00000 0x00000 RW 0x4

...

注意，ld-linux.so 本身就是一个 ELF 共享库，但它是静态编译的并且不具备共享库依赖项。当需要动态链接时，内核会引导动态链接（ELF 解释器），该链接首先会初始化自身，然后加载指定的共享对象（已加载则不必）。接着它会执行必要的再定位，包括目标共享对象所使用的共享对象。LD\_LIBRARY\_PATH 环境变量定义查找可用共享对象的位置。定义完成后，控制权会被传回到初始程序以开始执行。

再定位是通过一个称为 Global Offset Table（GOT）和 Procedure Linkage Table（PLT）的间接机制来处理的。这些表格提供了 ld-linux.so 在再定位过程中加载的外部函数和数据的地址。这意味着无需改动需要间接机制（即，使用这些表格）的代码：只需要调整这些表格。一旦进行加载，或者只要需要给定的函数，就可以发生再定位（稍候在 [用 Linux 进行动态加载](https://www.ibm.com/developerworks/cn/linux/l-dynamic-libraries/index.html#dynamicloading) 小节中会看到更多的差别）。

再定位完成后，动态链接器就会允许任何加载的共享程序来执行可选的初始化代码。该函数允许库来初始化内部数据并备之待用。这个代码是在上述 ELF 映像的 .init 部分中定义的。在卸载库时，它还可以调用一个终止函数（定义为映像的 .fini 部分）。当初始化函数被调用时，动态链接器会把控制权转让给加载的原始映像。

## 用 Linux 进行动态加载

Linux 并不会自动为给定程序加载和链接库，而是与应用程序本身共享该控制权。这个过程就称为动态加载。使用动态加载，应用程序能够先指定要加载的库，然后将该库作为一个可执行文件来使用（即调用其中的函数）。但是正如您在前面所了解到的，用于动态加载的共享库与标准共享库（ELF 共享对象）无异。事实上，ld-linux 动态链接器作为 ELF 加载器和解释器，仍然会参与到这个过程中。

动态加载（Dynamic Loading，DL）API 就是为了动态加载而存在的，它允许共享库对用户空间程序可用。尽管非常小，但是这个 API 提供了所有需要的东西，而且很多困难的工作是在后台完成的。表 1 展示了这个完整的 API。

##### 表 1. Dl API

| **函数** | **描述** |
| --- | --- |
| **dlopen** | 使对象文件可被程序访问 |
| **dlsym** | 获取执行了 dlopen 函数的对象文件中的符号的地址 |
| **dlerror** | 返回上一次出现错误的字符串错误 |
| **dlclose** | 关闭目标文件 |

该过程首先是调用 dlopen，提供要访问的文件对象和模式。调用 dlopen 的结果是稍候要使用的对象的句柄。mode 参数通知动态链接器何时执行再定位。有两个可能的值。第一个是 RTLD\_NOW，它表明动态链接器将会在调用 dlopen 时完成所有必要的再定位。第二个可选的模式是RTLD\_LAZY，它只在需要时执行再定位。这是通过在内部使用动态链接器重定向所有尚未再定位的请求来完成的。这样，动态链接器就能够在请求时知晓何时发生了新的引用，而且再定位可以正常进行。后面的调用无需重复再定位过程。

还可以选择另外两种模式，它们可以按位 OR 到 mode 参数中。RTLD\_LOCAL 表明其他任何对象都无法使加载的共享对象的符号用于再定位过程。如果这正是您想要的的话（例如，为了让共享的对象能够调用原始进程映像中的符号），那就使用 RTLD\_GLOBAL 吧。

dlopen 函数还会自动解析共享库中的依赖项。这样，如果您打开了一个依赖于其他共享库的对象，它就会自动加载它们。函数返回一个句柄，该句柄用于后续的 API 调用。dlopen 的原型为：

#include <dlfcn.h>

void \***dlopen**( const char \*file, int mode );

有了 ELF 对象的句柄，就可以通过调用 dlsym 来识别这个对象内的符号的地址了。该函数采用一个符号名称，如对象内的一个函数的名称。返回值为对象符号的解析地址：

void \***dlsym**( void \*restrict handle, const char \*restrict name );

如果调用该 API 时发生了错误，可以使用 dlerror 函数返回一个表示此错误的人类可读的字符串。该函数没有参数，它会在发生前面的错误时返回一个字符串，在没有错误发生时返回 NULL：

char \***dlerror**();

最后，如果无需再调用共享对象的话，应用程序可以调用 dlclose 来通知操作系统不再需要句柄和对象引用了。它完全是按引用来计数的，所以同一个共享对象的多个用户相互间不会发生冲突（只要还有一个用户在使用它，它就会待在内存中）。任何通过已关闭的对象的 dlsym 解析的符号都将不再可用。

char \***dlclose**( void \*handle );

## 动态加载示例

了解了 API 之后，下面让我们来看一看 DL API 的例子。在这个应用程序中，您主要实现了一个 shell，它允许操作员来指定库、函数和参数。换句话说，也就是用户能够指定一个库并调用该库（先前未链接于该应用程序的）内的任意一个函数。首先使用 DL API 来解析该库中的函数，然后使用用户定义的参数（用来发送结果）来调用它。清单 2 展示了完整的应用程序。

##### 清单 2. 使用 DL API 的 Shell

#include <stdio.h>

#include <dlfcn.h>

#include <string.h>

#define MAX\_STRING 80

void invoke\_method( char \*lib, char \*method, float argument )

{

void \*dl\_handle;

float (\*func)(float);

char \*error;

/\* Open the shared object \*/

dl\_handle = **dlopen**( lib, RTLD\_LAZY );

if (!dl\_handle) {

printf( "!!! %s\n", dlerror() );

return;

}

/\* Resolve the symbol (method) from the object \*/

func = **dlsym**( dl\_handle, method );

error = **dlerror**();

if (error != NULL) {

printf( "!!! %s\n", error );

return;

}

/\* Call the resolved method and print the result \*/

printf(" %f\n", (\*func)(argument) );

/\* Close the object \*/

dlclose( **dl\_handle** );

return;

}

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

{

char line[MAX\_STRING+1];

char lib[MAX\_STRING+1];

char method[MAX\_STRING+1];

float argument;

while (1) {

printf("> ");

line[0]=0;

fgets( line, MAX\_STRING, stdin);

if (!strncmp(line, "bye", 3)) break;

sscanf( line, "%s %s %f", lib, method, &argument);

invoke\_method( lib, method, argument );

}

}

构建这个应用程序，需要通过 GNU Compiler Collection（GCC）使用如下的编译行。选项 -rdynamic 用来通知链接器将所有符号添加到动态符号表中（目的是能够通过使用 dlopen 来实现向后跟踪）。-ldl 表明一定要将 dllib 链接于该程序。

gcc -rdynamic -o dl dl.c -ldl

再回到 [清单 2](https://www.ibm.com/developerworks/cn/linux/l-dynamic-libraries/index.html#list2)，main 函数仅充当解释器，解析来自输入行的三个参数（库名、函数名和浮点参数）。如果出现 bye 的话，应用程序就会退出。否则的话，这三个参数就会传递给使用 DL API 的 invoke\_method 函数。

首先调用 dlopen 来访问目标文件。如果返回 NULL 句柄，表示无法找到对象，过程结束。否则的话，将会得到对象的一个句柄，可以进一步询问对象。然后使用 dlsym API 函数，尝试解析新打开的对象文件中的符号。您将会得到一个有效的指向该符号的指针，或者是得到一个 NULL 并返回一个错误。

在 ELF 对象中解析了符号后，下一步就只需要调用函数。要注意一下这个代码和前面讨论的动态链接的差别。在这个例子中，您强行将目标文件中的符号地址用作函数指针，然后调用它。而在前面的例子是将对象名作为函数，由动态链接器来确保符号指向正确的位置。虽然动态链接器能够为您做所有麻烦的工作，但这个方法会让您构建出极其动态的应用程序，它们可以再运行时被扩展。

调用 ELF 对象中的目标函数后，通过调用 dlclose 来关闭对它的访问。

清单 3 展示了一个如何使用这个测试程序的例子。在这个例子中，首先编译程序而后执行它。接着调用了 math 库（libm.so）中的几个函数。完成演示后，程序现在能够用动态加载来调用共享对象（库）中的任意函数了。这是一个很强大的功能，通过它还能够给程序扩充新的功能。

##### 清单 3. 使用简单的程序来调用库函数

mtj@camus:~/dl$ **gcc -rdynamic -o dl dl.c -ldl**

mtj@camus:~/dl$ **./dl**

> **libm.so cosf 0.0**

1.000000

> **libm.so sinf 0.0**

0.000000

> **libm.so tanf 1.0**

1.557408

> **bye**

mtj@camus:~/dl$

## 工具

Linux 提供了很多种查看和解析 ELF 对象（包括共享库）的工具。其中最有用的一个当属 ldd 命令，您可以使用它来发送共享库依赖项。例如，在 dl 应用程序上使用 ldd 命令会显示如下内容：

mtj@camus:~/dl$ **ldd dl**

linux-gate.so.1 => (0xffffe000)

libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0xb7fdb000)

libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7eac000)

/lib/ld-linux.so.2 (0xb7fe7000)

mtj@camus:~/dl$

ldd 所告诉您的是：该 ELF 映像依赖于 linux-gate.so（一个特殊的共享对象，它处理系统调用，它在文件系统中无关联文件）、libdl.so（DL API）、GNU C 库（libc.so）以及 Linux 动态加载器（因为它里面有共享库依赖项）。

readelf 命令是一个有很多特性的实用程序，它让您能够解析和读取 ELF 对象。readelf 有一个有趣的用途，就是用来识别对象内可再定位的项。对于我们这个简单的程序来说（[清单 2](https://www.ibm.com/developerworks/cn/linux/l-dynamic-libraries/index.html#list2) 展示的程序），您可以看到需要再定位的符号为：

mtj@camus:~/dl$ **readelf -r dl**

Relocation section '.rel.dyn' at offset 0x520 contains 2 entries:

Offset Info Type Sym.Value Sym. Name

08049a3c 00001806 R\_386\_GLOB\_DAT 00000000 \_\_gmon\_start\_\_

08049a78 00001405 R\_386\_COPY 08049a78 stdin

Relocation section '.rel.plt' at offset 0x530 contains 8 entries:

Offset Info Type Sym.Value Sym. Name

08049a4c 00000207 R\_386\_JUMP\_SLOT 00000000 dlsym

08049a50 00000607 R\_386\_JUMP\_SLOT 00000000 fgets

08049a54 00000b07 R\_386\_JUMP\_SLOT 00000000 dlerror

08049a58 00000c07 R\_386\_JUMP\_SLOT 00000000 \_\_libc\_start\_main

08049a5c 00000e07 R\_386\_JUMP\_SLOT 00000000 printf

08049a60 00001007 R\_386\_JUMP\_SLOT 00000000 dlclose

08049a64 00001107 R\_386\_JUMP\_SLOT 00000000 sscanf

08049a68 00001907 R\_386\_JUMP\_SLOT 00000000 dlopen

mtj@camus:~/dl$

从这个列表中，您可以看到各种各样的需要再定位（到 libc.so）的 C 库调用，包括对 DL API（libdl.so）的调用。函数 \_\_libc\_start\_main 是一个 C 库函数，它优先于程序的 main 函数（一个提供必要初始化的 shell）而被调用。

其他操作对象文件的实用程序包括：objdump，它展示了关于对象文件的信息；nm，它列出来自对象文件（包括调试信息）的符号。还可以将 EFL 程序作为参数，直接调用 Linux 动态链接器，从而手动启动映像：

mtj@camus:~/dl$ **/lib/ld-linux.so.2 ./dl**

> **libm.so expf 0.0**

1.000000

>

另外，可以使用 ld-linux.so 的 --list 选项来罗列 ELF 映像的依赖项（ldd 命令也如此）。切记，它仅仅是一个用户空间程序，是由内核在需要时引导的。

# 基于事件驱动的应用开发模型

首先，在分布式系统中，事件是异步非耦合的，这是系统实时性的要求。因为在分布式系统中，往往各个关键应用既是服务器应用，也是客户应用，相互之间从功能上看是对等关系，如果在同步情况下，一个应用驱动了另一个应用的事件，这个应用则必须堵塞，以等待事件执行的返回状态，这样这个应用在这段时间内则不能处理实时事件。而各个应用之间相互不完全依赖的情况也决定了分布式系统中事件的两端必须为非耦合。

事件驱动的开发模型包括如下几个部分：

* 事件定义库：该库定义了应用所有认知的事件描述，基本信息包括事件标识，事件特征，事件处理例程，事件所属软件包等。
* 事件检测模块：该模块负责处理事件的收集和分发，当接收事件后，分析事件的类型、特征，然后作出相应的处理，该模块是开发模型的核心。
* 事件执行模块：负责获知事件入口，解码事件携带数据，正确执行相应事件。事件执行模块可以支持事件的持久性，持久性事件的接收方如果中间停止，而在再次启动后仍可以接收事件。
* 事件信道：是分布式应用之间事件流动的通道。事件信道是单向的，适应分布式应用的多样性要求，事件信道存在多种类型，主要有：1）永久可靠的信道。应用之间具有频繁的事件往来或固定的周期性的关键事件要采用这类信道；2）一次性可靠信道。事件的收发具有偶然性，不经常性，但是事件不可丢失；3）广播数据报不可靠信道。信道上的事件是一对多的关系，可以容忍偶然的数据报丢失或失序，如某一些关键应用的指示存在的心跳事件可采用该类信道；4）单播数据报信道；5）数据报可靠信道。这类信道主要适应于一对多的可靠事件，该信道在不可靠数据报基础上增加了数据报序列号和事件重发请求，以便形成可靠的事件通道。事件通道为分布式应用提高运行效率提供了基础设施。
* 事件输入源：是驱动事件的源应用。包括了事件数据的编码和封装，编码是为了使得事件可以在不同的系统平台上流动，封装的事件数据中，头部信息固定的为公共可以识别的信息。
* 最后要提供事件驱动应用API：包括了事件的注册API，以及实现以上各功能的各类API。API要求简单明了，为应用程序员

https://github.com/imiskolee/mempool