Linux 動的ライブラリーの徹底調査

プロセスと API

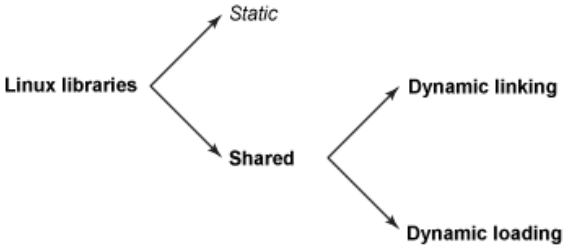
M. Tim Jones
Consultant Engineer
Emulex Corp.

2008年 8月 20日

動的にリンクされる共有ライブラリーは、GNU/Linux® の重要な側面です。なぜなら、実行可能プログラムが実行時にこれらのライブラリーを使って動的に外部機能にアクセスすることによって、メモリーの全体的なフットプリントを縮小できるからです (つまり、必要に応じて機能を取り込むという手段です)。この記事では、動的ライブラリーの作成および使用プロセスを調査し、動的ライブラリーを利用する各種ツールの詳細を説明するとともに、これらのライブラリーが内部ではどのように機能しているのかを探ります。

ライブラリーは、同様の機能を 1 つのユニットにまとめてパッケージ化するために設計されました。複数の機能がパッケージされたユニットは他の開発者と共有できるようになるため、いわゆる**モジュール・プログラミング**が可能になります。つまり、複数のモジュールからプログラムを構築するということです。Linux がサポートする 2 つのタイプのライブラリーは、それぞれに固有の利点と欠点があります。一方のタイプは**静的ライブラリー**で、静的ライブラリーに含まれる機能はコンパイル時に静的にプログラムにバインドされます。それとは異なり、もう一方のタイプである**動的ライブラリー**はアプリケーションがロードされるときにロードされ、バインディングは実行時に行われます。図 1 に、Linux でのライブラリー階層を示します。

図 1. Linux でのライブラリー階層



徹底調査シリーズ他、developerWorks に掲載されている Tim の記事

- Linux ローダブル・カーネル・モジュールの徹底調査
- Linux フラッシュ・ファイルシステムの徹底調査
- Anatomy of Security-Enhanced Linux (SELinux)
- リアルタイム Linux アーキテクチャーの徹底調査

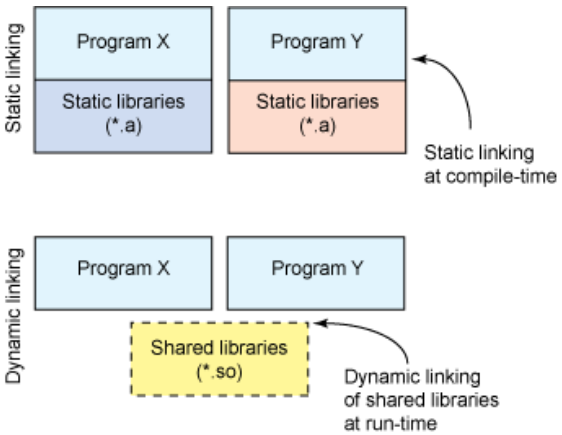
- Linux SCSI サブシステムの徹底調査
- Linux ファイルシステムの徹底調査
- Linux ネットワーク・スタックの徹底調査
- Linux カーネルの徹底調査
- Linux スラブ・アロケーターの徹底調査
- Linux での同期方式の徹底調査
- Tim の一連の「徹底調査」シリーズの記事
- Tim が書いたすべての developerWorks 記事

共有ライブラリーを使用するには、実行時に動的にリンクするか、あるいはプログラムの制御によって動的にロードして使用するかのいずれかの方法を採れます。この記事では、その両方の方式を検討します。

最小限の機能だけが必要な小規模なプログラムには、静的ライブラリーを使うとメリットが得られます。一方、複数のライブラリーが必要となるプログラムには、共有ライブラリーを使用することによって、プログラムのメモリー・フットプリント (プログラム実行時にディスク上に作成されるメモリー領域とメモリー内のメモリー領域の両方) を減らすことができます。これは、複数のプログラムが共有ライブラリーを同時に使用するため、メモリー内には 1 つのライブラリーのコピーがあればよいだけだからです。静的ライブラリーを使用した場合には、実行中のすべてのプログラムがそれぞれに固有のライブラリーのコピーを持つことになります。

GNU/Linux では、共有ライブラリーを 2 通りの方法で扱えるようになっています (それぞれ Sun Solaris を起源とする方法)。まず 1 つは、プログラムと共有ライブリーを動的にリンクし、実行時にライブラリーを (メモリー内にない場合に限り) Linux にロードさせるという方法です。もう 1 つの方法では、**動的ロード**と呼ばれるプロセスのなかで、プログラムが選択的にライブラリーの関数を呼び出します。動的ロードでは、プログラムが特定のライブラリーを (すでにロードされていなければ) ロードしてから、そのライブラリーに含まれる特定の関数を呼び出すことができます (図 2 では、この 2 つの方式を示しています)。これは、プラグインをサポートするアプリケーションを構築する際に一般的に用いられるパターンです。この記事では、このアプリケーション・プログラム・インターフェース (API) について探り、その後に具体的な例を紹介します。

図 2. 静的リンクと動的リンク



Linux での動的リンク

それでは早速、Linux で動的にリンクした共有ライブラリーを使用する場合のプロセスについて詳しく探っていきます。ユーザーがアプリケーションを起動すると、ELF (Executable and Linking Format) イメージが呼び出されます。するとカーネルが ELF イメージをユーザー空間の仮想メモ

リーにロードするプロセスを開始します。カーネルはこのプロセスで、動的リンカー (/lib/ld-linux.so) が使用されることを示す .interp という ELF セクションを検知します (リスト 1 を参照)。このセクションは、UNIX® でのスクリプト・ファイルのインタープリター定義 (#!/bin/sh) のようなもので、単に異なるコンテキストで使用されるだけです。

リスト 1. プログラム・ヘッダーを示す readelf

```
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

  ...

mtj@camus:~dl$
```

ld-linux.so はそれ自体が ELF 共有ライブラリーですが、静的にコンパイルされ、共有ライブラリーの依存関係を持たないことに注目してください。動的リンクが必要になると、カーネルは動的リンカー (ELF インタープリター) をブートストラップし、動的リンカーが自己初期化してから、指定された共有オブジェクト(まだロードされていない場合のみ) をロードします。続いて必要な再配置を行います。これにはターゲットとする共有オブジェクトが使用する共有オブジェクトの再配置も含まれます。そして、使用可能な共有オブジェクトの検索先が、LD_LIBRARY_PATH 環境変数によって定義されます。このプロセスが完了すると制御が元のプログラムに戻り、プログラムの実行が開始されます。

再配置の処理は、GOT (Global Offset Table) と呼ばれる間接参照メカニズム、そして PLT (Procedure Linkage Table) によって行われます。この 2 つのテーブルが指定する外部関数およびデータのアドレスは、ld-linux.so が再配置プロセス中にロードします。これはつまり、間接参照を必要とする (これらのテーブルを使用する) コードを変更する必要はなく、テーブルを調整しさえすればよいということです。再配置はロードと同時に行うことも、特定の関数が必要になったときに行うこともできます (この違いについては、この後の「Linux での動的ロード」で説明します)。

再配置が完了すると、ロードされたあらゆる共有オブジェクトが動的リンカーを使ってオプションの初期化コードを実行できるようになります。この機能を使って、ライブラリーは内部データを初期化し、使用可能な状態に準備します。そのためのコードが定義されているのは、ELF イメージの .init セクションです。ライブラリーがアンロードされるときには、終了関数 (イメージ内に .fini セクションとして定義されています) を呼び出すこともできます。初期化関数が呼び出されると、動的リンカーはロード中の元のイメージに対する制御を手放します。

Linux での動的ロード

Linux は特定のプログラムに対してライブラリーを自動的にロードしてリンクする代わりに、この制御をアプリケーション自体と共有することができます。この場合のプロセスは、動的ロードと呼ばれます。動的ロードでは、アプリケーションがロード対象とする特定のライブラリーを指定してから、このライブラリーを実行可能プログラムとして使用することができます (つまり、ライブラリー内の関数を呼び出すということです)。ただし前にも説明したように、動的ロードに使用される共有ライブラリーは、標準共有ライブラリー (ELF 共有オブジェクト) と何の違いもありません。実際、このプロセスでも動的リンカーが ELF ローダーおよびインタープリターとして関わってきます。

動的ロードのためにある DL (Dynamic Loading) API は、ユーザー空間のプログラムから共有ライブラリーを利用できるようにします。この API は小規模ながらも、必要なすべての機能を提供し、厄介な作業のほとんどを舞台裏で行います。表 1 に、この API が持つすべての関数を記載します。

表 1. Dl API

関数	説明
dlopen	オブジェクト・ファイルをプログラムからアクセスできるようにする
dlsym	dlopen されたオブジェクト・ファイルに含まれるシンボルのアドレスを取得する
dlerror	最後に発生したエラーをストリングで返す
dlclose	オブジェクト・ファイルをクローズする

動的ロードのプロセスは、アクセス対象のファイル・オブジェクトとモードを指定して dlopen を呼び出すところから始まります。dlopen 呼び出しの結果が、後で使用されるオブジェクト・ハンドルになります。動的リンカーに再配置の実行タイミングを指示する mode 引数には、2 つの値が考えられます。その 1 つは RTLD_NOW で、この場合、動的リンカーは dlopen 呼び出し時に必要なすべての再配置を完了することになります。もう 1 つの値、RTLD_LAZY は、必要なときにだけ再配置を実行するためのモードです。このモードは内部で行われ、動的リンカーによってまだ再配置されていないすべての要求がリダイレクトされます。そのため新しい参照が発生していれば、動的リンカーは要求時にそれを認識し、再配置が通常通りに行われるというわけです。その後の呼び出しでは、再配置を繰り返す必要はありません。

使用できるオプションは他にも 2 つあります。これらのオプションはビット単位の OR 演算が行われて mode 引数に組み込まれます。RTLD_LOCAL を使用すると、ロード中の共有オブジェクトのシンボルは、それ以外のオブジェクトによる再配置プロセスでは使用できなくなります。他のオブジェクトの再配置プロセスにもシンボルを有効にしたい場合 (共有オブジェクトが元のプロセス・イメージのシンボルを呼び出せるようにする場合など) は、RTLD_GLOBAL を使用してください。

dlopen 関数は共有ライブラリーの依存関係も自動的に解決します。そのため、他の共有ライブラリーに依存するオブジェクトを開いた場合には、これらの共有ライブラリーも自動的にロードされます。この関数は、以降の API に対する呼び出しで使用されるハンドルを返します。dlopen のプロトタイプは以下のとおりです。

```
#include <dlfcn.h>

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

ELF オブジェクトのハンドルがあれば、dlsym 呼び出しを使用して ELF オブジェクト内のシンボルへのアドレスを識別することができます。この dlsym 関数はシンボル名を引数に取ります (例えば、オブジェクト内に含まれる関数の名前など)。戻り値は、オブジェクト内のシンボルへの解決済みアドレスです。

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

この API での呼び出し中にエラーが発生した場合には、dlerror 関数を使用して人間が理解可能なストリングでエラーの内容を取得してください。この関数には引数はなく、エラーが発生している場合にはエラーを表現するストリングを返し、エラーが発生していない場合には NULL を返します。

```
char *dlerror();
```

最終的に共有オブジェクトを呼び出す必要がなくなったときには、アプリケーションは dlclose を呼び出して、ハンドルおよびオブジェクト参照が不要になったことをオペレーティング・システムに通知することができます。この関数では適切に参照が考慮されるため、共有オブジェクトの複数のユーザーが互いに競合することはありません (共有オブジェクトは、それを使用するユーザーがいる限りメモリー内に残ります)。dlsym によって解決されたクローズ・オブジェクトのシンボルは使用できなくなります。

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

動的ロードの例

DL API について説明し終えたところで、今度は DL API の具体的な例を見ていきましょう。このアプリケーションには基本的に、オペレーターがライブラリー、関数、引数を指定できるシェルを実装します。つまり、ユーザーがライブラリーを指定し、そのライブラリー (このアプリケーションにリンクされていなかったライブラリー) のなかにある任意の関数を呼び出せるということです。DL API を使ってライブラリー内で関数を解決した上で、ユーザー定義の引数 (結果を出力) を設定して関数を呼び出します。このアプリケーション全体をリスト 2 に記載します。

リスト 2. DL API を使用するためのシェル

```
#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 );

 }

}
```

このアプリケーションをビルドするには、GCC (GNU Compiler Collection) で以下のコンパイル行を実行します。`-rdynamic` オプションを使用して、リンカーに、すべてのシンボルを動的シンボル・テーブルに追加するように指示しています (`dlopen` を使用してバックトレースできるようにするためです)。`-ldl` によって、`dllib` がこのプログラムにリンクするように指定されます。

```
gcc -rdynamic -o dl dl.c -ldl
```

リスト 2 をもう一度見てみると、`main` 関数は単なるインタープリターとして機能し、入力行の 3 つの引数 (ライブラリー名、関数名、浮動小数点引数) を構文解析します。入力行に `bye` と入力された場合には、アプリケーションは終了します。そうでなければ 3 つの引数が `invoke_method` 関数に渡され、この関数が DL API を使用します。

最初のステップは、dlopen を呼び出してオブジェクト・ファイルにアクセスできるようにすることです。NULL ハンドルが返された場合、これはオブジェクトが見つからなかったことを示すため、プロセスが終了します。それ以外の場合はオブジェクト・ハンドルが返されるので、問い合わせを続行できます。dlsym API 関数を使用して、新たにオープンされたオブジェクト・ファイル内のシンボルを解決してください。この関数の実行結果は、シンボルへの有効なポインターを取得するか、あるいは NULL を取得してエラーが返されるかのいずれかとなります。

ELF オブジェクトでシンボルが解決された後のステップは、関数を呼び出すだけです。このコードと前に説明した動的リンクとの違いに注意してください。この例では、オブジェクト・ファイル内のシンボルのアドレスを関数ポインターにして関数を呼び出していますが、前の例ではオブジェクトの名前を関数として使っていたため、動的リンカーはシンボルが正しい場所を指すことを確実にすることができました。動的リンカーに面倒な作業をすべて引き受けさせることもできますが、このコードの方法を使えば、実行時に拡張可能な極めて動的なアプリケーションをビルドすることができます。

ELF オブジェクト内のターゲット関数を呼び出したら、dlclose を呼び出して、このオブジェクトへのアクセスを終了します。

リスト 3 に、このテスト・プログラムの使用例を記載します。この例では、プログラムをコンパイルしてから実行します。続いて、数学ライブラリー (libm.so) 内のいくつかの関数を呼び出します。この例からわかるように、プログラムは動的ロードを使用して共有オブジェクト (ライブラリー) 内の任意の関数を呼び出すことができます。この威力が、新しい機能でプログラムを拡張することを可能にします。

リスト 3. 単純なプログラムを使用したライブラリー関数の呼び出し

```
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$
```

ツール

Linux には、ELF オブジェクト (共有ライブラリーを含む) の表示および構文解析に使用できる多種多様なツールが揃っています。なかでもとりわけ便利なのは、共有ライブラリーの依存関係を出力する ldd コマンドです。例えばサンプルの dl アプリケーションで ldd コマンドを使用すると、以下の出力が表示されます。

```
mtj@camus:~/dl$ldd dl
        linux-gate.so.1 =>  (0xffffe000)
        libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0xb7fdb000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7eac000)
        /lib/ld-linux.so.2 (0xb7fe7000)
mtj@camus:~/dl$
```

上記で ldd が伝えている内容は、この ELF イメージは、linux-gate.so (システム・コールを処理する特殊な共有オブジェクトで、ファイルシステム内に関連ファイルを持ちません)、libdl.so (DL API)、GNU C ライブラリー (libc.so)、さらには Linux ダイナミック・ローダー (共有ライブラリー依存関係があるため) に依存しているということです。

readelf コマンドは充実した機能を備えたユーティリティーで、ELF オブジェクトを構文解析して読み込むために使用することができます。readelf の興味深い使用例は、オブジェクト内の再配置可能な項目を識別することです。この記事の単純なサンプル・プログラム (リスト 2 を参照) の場合、再配置が必要なシンボルは以下のように表示されます。

```
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$
```

このリストを見ると、DL API (libdl.so) への呼び出しを含め、さまざまな C ライブラリー呼び出しを libc.so に再配置しなければならないことがわかります。__libc_start_main 関数は、プログラムの main 関数の前に呼び出される C ライブラリー関数です (必要な初期化を行うシェル)。

オブジェクト・ファイルで動作するその他のユーティリティーとしては、オブジェクト・ファイルに関する情報を表示する objdump、オブジェクト・ファイルからシンボル (デバッグ情報を含めて) の一覧を作成する nm があります。また、手動でイメージを開始するように引数を設定して、Linux 動的リンカーを ELF プログラムで直接呼び出すことも可能です。

```
mtj@camus:~/dl$/lib/ld-linux.so.2 ./dl
>libm.so expf 0.0
  1.000000
>
```

さらに、ld-linux.so で (ldd コマンドと同様に) --list オプションを使用して、ELF イメージの依存関係を一覧にすることもできます。このように、動的リンカーはカーネルが必要に応じてブートストラップする、ユーザー空間の単なるプログラムだということを覚えておいてください。

さらに詳しく調べてください

この記事では、動的リンカーのいくつかの機能を簡単に概説したに過ぎません。ELM イメージ・フォーマットについての詳しい紹介、そしてプロセスやシンボル再配置については、「参考文献」セクションを参照してください。さらに Linux でのいつもの例に漏れず、動的リンカーの

ソースをダウンロードして、その内部要素を調べることもできます。動的リンカー・ソースのダウンロード方法についての詳細は、「参考文献」に記載されています。

著者について

M. Tim Jones

M. Tim Jones は組み込みソフトウェアのエンジニアであり、『Artificial Intelligence: A Systems Approach』、『GNU/Linux Application Programming』(現在、第 2 版です) や『AI Application Programming』(こちらも現在、第 2 版です)、それに『BSD Sockets Programming from a Multilanguage Perspective』などの著者でもあります。技術的な経歴は静止軌道衛星用のカーネル開発から、組み込みシステム・アーキテクチャーやネットワーク・プロトコル開発まで、広範にわたっています。また、コロラド州ロングモン所在のEmulex Corp. の顧問エンジニアでもあります。