linux中共享库以.so结尾。共享库在链接时候并不像静态库把代码都添加到可执行文件中，而只是作些标记，记录需要的符号。然后在程序加载或运行时，添加所需要的符号。

共享库的使用可分为动态链接(dynamic linking)和动态装载(dynamic loading)。

共享库的动态链接

同静态库一样，共享库的创建有两步：编译.o文件和把.o文件打包。

生成.o目标文件：

$ cat test_old.c

点击(此处)折叠或打开

```
1. #include <stdio.h>
2. void fun(void)
3. {
4.         printf("I'm old\n");
5. }
```

```
1. $ gcc -fPIC -c test_old.c
```

-fPIC参数的意思是表明创建位置无关代码（position independent code），这通常是创建共享库必须的。因为动态库和静态库不同，链接时是没法确定代码运行时的地址的，所以要告诉编译器该代码有可能在任意地址运行。

生成共享库：

```
1. $ gcc -shared -o libtest.so test_old.o
```

这样，共享库libtest.so就创建好了。

链接：

$ cat main.c

点击(此处)折叠或打开

```c
#include <stdio.h>

extern void fun(void);

int main(int argc, const char *argv[])
{
        fun();
        return 0;
}
```

链接时需要用-L指定动态库所在的位置。

```
$ gcc main.c -ltest -L.
```

执行：

```
$ ./a.out
./a.out: error while loading shared libraries: libtest.so: cannot open shared object file: No such file or directory
```

我们发现报错信息说加载共享库时找不到共享库目标文件。
程序运行前，会把这个程序需要的符号都加载到内存（虚拟内存）中，那加载器怎么知道这个程序需要哪些符号（symbols）呢？实际上，这些信息在链接时已经被加入到elf文件中。当然，链接时加进去的只是符号的名字，而没有符号的内容。
对于动态库的符号加载，程序运行前需要先找到这个动态库，然后把程序需要的符号内容找出来导入进去。目前我们的问题是加载器不知道去哪里找这个库。有四种方法可以解决这个问题：

1. 设置LD_LIBRARY_PATH环境变量

```
$ export LD_LIBRARY_PATH=`pwd`
```

2. 把共享库添加到默认的库路径中
可以使用软链接的方式：

```
$ ln -s `pwd`/libtest.so /usr/lib
```

3. 把路径添加到/etc/ld.so.conf中，再执行ldconfig

```
$ sudo echo `pwd` >>/etc/ld.so.conf
$ sudo ldconfig
```

4. 链接时使用-R选项(或 -rpath)指定run-time path.

```
$ gcc main.c -ltest1 -L. -Wl,-R.
```

选择一种方法，设置好后再运行，可以看到程序正常跑起来了。

```
$ ./a.out
I'm old
```

使用动态链接共享库的优点：

1. 减少可执行文件的大小。

2. 方便更新。

例如我们想用test_new.c代替test_old.c，先创建新的共享库：

```
$ gcc -fPIC -c test_new.c
$ gcc -shared -o libtest.so test_new.o
```

只需要替换libtest.so，无需重新链接，就可执行：

```
$ ./a.out
I'm new
```

共享库缺点：

容易产生版本问题。

例如，开发时使用的某个动态库版本是1.0的，开发完发布给用户。而用户机子上该动态库版本是0.9，就有可能引起某些意想不到的问题。

共享库的动态加载

共享库还可以在程序运行过程中动态加载。而不是在程序启动的时候加载。
linux中使用dl库可以实现动态加载。

下面是动态加载相关API的介绍：

1. dlopen

```
1. void* dlopen(const char *libname,int flag);
```

dlopen必须在dlerror，dlsym和dlclose之前调用，表示要将库装载到内存，准备使用。
如果要装载的库依赖于其它库，必须首先装载依赖库。如果dlopen操作失败，返回NULL值；如果库已经被装载过，则dlopen会返回同样的句柄。
参数中的libname是库的路径和名称。flag参数表示处理未定义函数的方式，可以使用RTLD_LAZY或RTLD_NOW。RTLD_LAZY表示暂时不去处理未定义函数，先把库装载到内存，等用到没定义的函数再说；RTLD_NOW表示马上检查是否存在未定义的函数，若存在，则dlopen以失败告终。

2. dlerror

```
1. char* dlerror(void);
```

获得最近一次dlopen,dlsym或dlclose操作的错误信息，返回NULL表示无错误。
dlerror在返回错误信息的同时，也会清除错误信息。

3. dlsym

```
1. void* dlsym(void *handle,const char *symbol);
```

返回指定函数(symbol)的指针。
如果找不到指定函数，则dlsym会返回NULL值。但判断函数是否存在最好的方法是使用dlerror函数，

4. dlclose

```
1. int dlclose(void *);
```

将已经装载的库句柄减一，如果句柄减至零，则该库会被卸载。如果存在析构函数，则析构函数会被调用。

使用举例：

$cat test_dl.c

点击(此处)折叠或打开

```
1. #include  <stdio.h>
2. #include  <dlfcn.h>
3. #include  <stdlib.h>
4. #include  <string.h>
5. int  main(int  argc,  char  **argv)
6. {
7.          void  *handle;
8.          void  (*print)(int);
9.          char  *error;
10.
11.          handle  =  dlopen("./libmyprint.so",  RTLD_LAZY);
12.          if  (!handle)  {
13.                  fputs(dlerror(),  stderr);
14.                  exit(1);
15.          }
16.
17.          print  =  dlsym(handle,  "myprint");
18.          if  ((error  =  dlerror())  !=  NULL)  {
19.                  fputs(error,  stderr);
20.                  exit(1);
21.          }
22.
23.          print(1);
24.          print(2);
25.          print(3);
26.          dlclose(handle);
27.
28.          return 0;
29. }
```

共享库代码：

$cat myprint.c

点击(此处)折叠或打开

```
#include <stdio.h>
void myprint(int num)
{
        if (num == 1)
                printf("I'm Bob.\n");
        else if (num == 2)
                printf("I'm John.\n");
        else
                printf("Who am I?\n");
}
```

编译共享库：

```
gcc -fPIC -c myprint.c
gcc -shared -o libmyprint.so myprint.o
```

链接：

需要加上-rdynamic参数，通知链接器将库所有符号添加到动态符号表中。并链接libdl.so库。

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

执行结果：

$ ./test_dl

```
I'm Bob.
I'm John.
Who am I?
```

小结

本文介绍了共享库的动态链接和动态加载。对于elf文件的链接和装载只是略略带过。

如需深入理解，可以研究Linux中提供的readelf, nm, objdump等命令。也可以参考经典书籍linkers and loaders.