

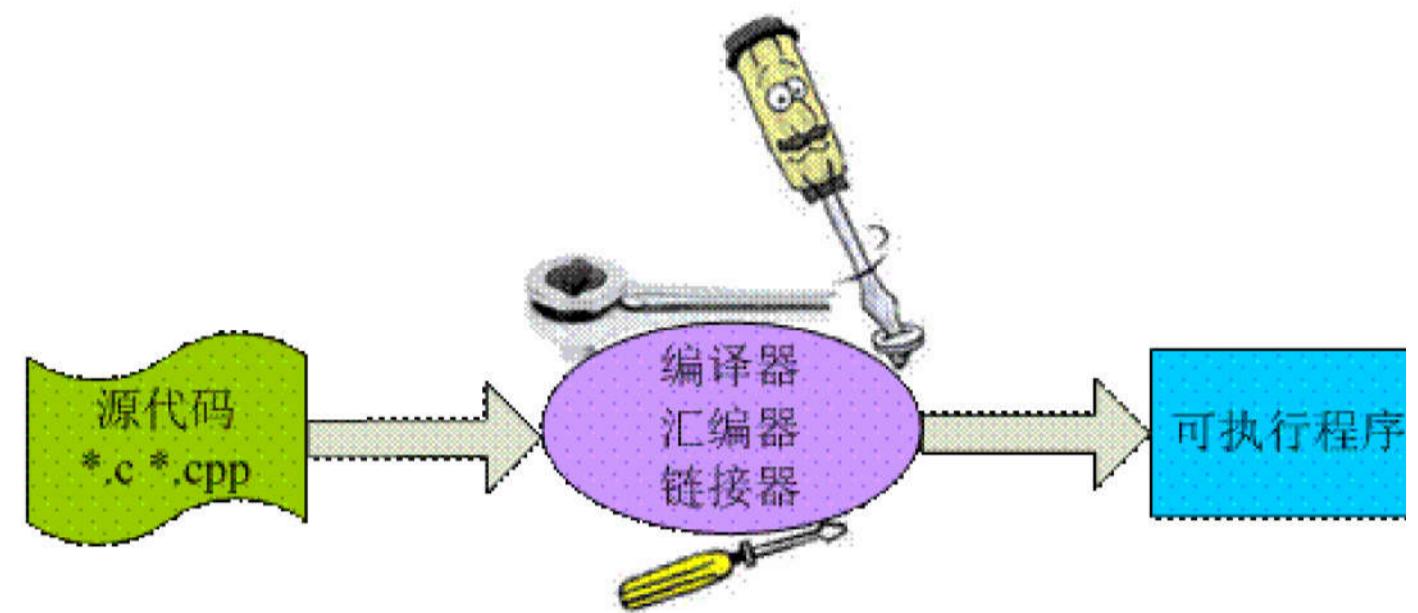


Linux系统中“动态库”和“静态库”那点事儿【转】

转自: <http://blog.chinaunix.net/uid-23069658-id-3142046.html>

今天我们主要来说说Linux系统下基于动态库(.so)和静态(.a)的程序那些猫腻。在这之前，我们需要了解一下源代码到可执行程序之间到底发生了什么神奇而美妙的事情。

在Linux操作系统中，普遍使用**ELF**格式作为可执行程序或者程序生成过程中的中间格式。**ELF**（**Executable and Linking Format**, 可执行连接格式）是**UNIX**系统实验室（**USL**）作为应用程序二进制接口（**Application Binary Interface, ABI**）而开发和发布的。工具接口标准委员会（**TIS**）选择了正在发展中的**ELF**标准作为工作在32位Intel体系上不同操作系统之间可移植的二进制文件格式。本文不对**ELF**文件格式及其组成做太多解释，以免冲淡本文的主题，大家只要知道这么个概念就行。以后再详解**Linux**中的**ELF**格式。源代码到可执行程序的转换时需要经历如下图所示的过程：



昵称：张昊华-sky
园龄：3年
粉丝：65
关注：9
+加关注

2017年12月						
日	一	二	三	四	五	六
26	27	28	29	30	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

搜索

随笔分类(2428)

- 【a1本人原创】(44)
- 【Android底层】(15)
- 【Android基础】(16)
- 【Android应用】(4)
- 【BLE】(9)
- 【C++】(1)
- 【C语言】(17)
- 【java】(17)
- 【lcd驱动】(5)
- 【linux usb驱动】(39)
- 【linux触摸屏驱动】(16)
- 【linux内存管理】(101)
- 【linux内核】(586)
- 【Linux内核定时器】(27)
- 【Linux文件系统】(4)
- 【linux无线驱动】
- 【Nand flash】(4)
- 【Object-c】
- 【Python】(11)
- 【RTOS】(2)
- 【UBI-flash文件系统】(4)
- 【uboot】(5)
- 【Windows】(16)
- 【代码管理】(67)
- 【各种框架】(12)
- 【各种使用技巧】(156)
- 【公司、市场、创业】(24)
- 【黑客】(10)
- 【科技讯息】(29)
- 【裸机开发】(16)
- 【嵌入式基础】(1047)

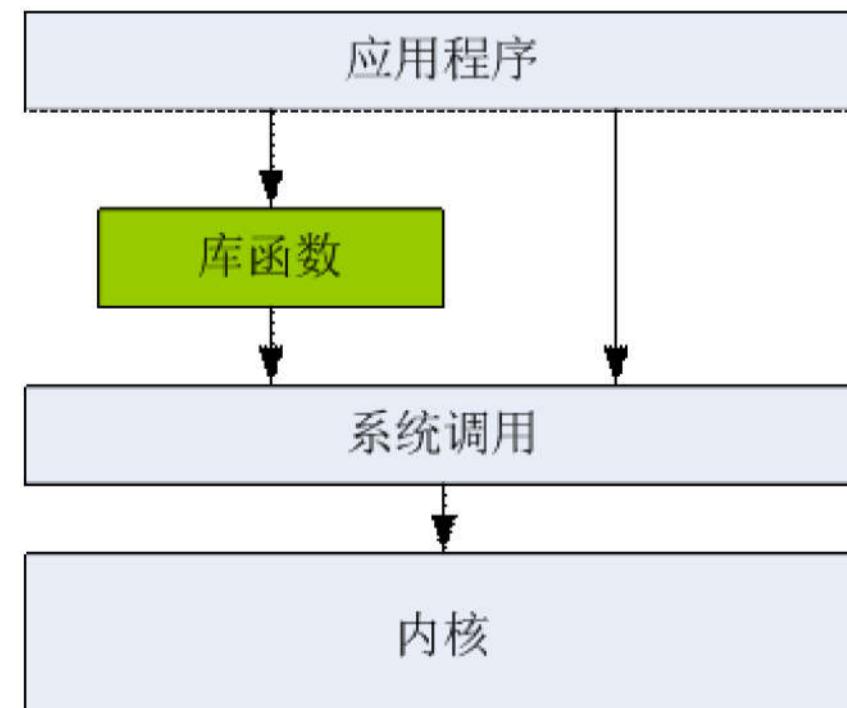
| 编译是指把用高级语言编写的程序转换成相应处理器的汇编语言程序的过程。从本质上讲，编译是一个文本转换的过程。对嵌入式系统而言，一般要把用C语言编写的程序转换成处理器的汇编代码。编译过程包含了C语言的语法解析和汇编码的生成两个步骤。编译一般是逐个文件进行的，对于每一个C语言编写的文件，可能还需要进行预处理。

| 汇编是从汇编语言程序生成目标系统的二进制代码（机器代码）的过程。机器代码的生成和处理器有密切的联系。相对于编译过程的语法解析，汇编的过程相对简单。这是因为对于一款特定的处理器，其汇编语言和二进制的机器代码是一一对应的。汇编过程的输入是汇编代码，这个汇编代码可能来源于编译过程的输出，也可以是直接用汇编语言书写的程序。

| 连接是指将汇编生成的多段机器代码组合成一个可执行程序。一般来说，通过编译和汇编过程，每一个源文件将生成一个目标文件。连接器的作用就是将这些目标文件组合起来，组合的过程包括了代码段、数据段等部分的合并，以及添加相应的文件头。

GCC是Linux下主要的程序生成工具，它除了编译器、汇编器、连接器外，还包括一些辅助工具。在下面的分析过程中我会教大家这些工具的基本使用方法，Linux的强大之处在于，对于不太懂的命令或函数，有一个很强大的“男人”时刻stand by your side，有什么不会的就去命令行终端输入：`man [命令名或函数名]`，然后阿拉神灯就会显灵了。

对于最后编译出来的可执行程序，当我们执行它的時候，操作系统又是如何反应的呢？我们先从宏观上来个总体把握，如图2所示：



作为**UNIX**操作系统的一种，Linux的操作系统提供了一系列的接口，这些接口被称为**系统调用**（**System Call**）。在**UNIX**的理念中，系统调用“提供的是机制，而不是策略”。C语言的库函数通过调用系统调用实现，库函数对上层提供了C语言库文件的接口。在应用程序层，通过调用C语言库函数和系统调用实现功能。一般来说，应用程序大多使用C语言库函数实现其功能，较少使用系统调用。

那么最后的可执行文件到底是什么样子呢？前面已经说过，这里我们不深入分析**ELF**文件的格式，只是给出它的一个结构图和一些简单的说明，以方便大家理解。

【人工智能】(22)
 【摄像头】(48)
 【思维】(24)
 【四轴飞行器】(3)
 【算法】(17)
 【网络】(10)

相册(1)

【背景图】(1)

积分与排名

积分 - 274895

排名 - 661

最新评论

1. Re:转: 嵌入式linux下usb驱动开发方法-看完少走弯路【转】

之前在华清远见的官网，找客服要了很多嵌入式linux驱动开发教程

-嵌入式资深爱好者

2. Re:本地搭建SVN局域网服务器【转】

访问No repository found in svn://... 找不到地址

-野味@有点甜

3. Re:Understand:高效代码静态分析神器详解（一）【转】

这个有windows版本的.我用过~

-角角头小坏蛋

4. Re:AI创投的冰与火之歌：泡沫、跟

风、短板和有钱花不出去的沮丧【转】

-呆水哥的快乐人生

ELF文件格式包括三种主要的类型：可执行文件、可重定向文件、共享库。

1. 可执行文件（应用程序）

可执行文件包含了代码和数据，是可以直接运行的程序。

2. 可重定向文件 (*.o)

可重定向文件又称为目标文件，它包含了代码和数据（这些数据是和其他重定位文件和共享的object文件一起连接时使用的）。

.o文件参与程序的连接（创建一个程序）和程序的执行（运行一个程序），它提供了一个方便有效的方法来用并行的视角看待文件的内容，这些.o文件的活动可以反映出不同的需要。

Linux下，我们可以用gcc -c编译源文件时可将其编译成*.o格式。

3. 共享文件 (*.so)

也称为动态库文件，它包含了代码和数据（这些数据是在连接时候被连接器ld和运行时动态连接器使用的）。动态连接器可能称为ld.so.1, libc.so.1或者 ld-linux.so.1。我的CentOS6.0系统中该文件为：/lib/ld-2.12.so

链接器的角度



加载器的角度



偶忍不住又要罗嗦一句了，相信俺，我的唠叨对大家是有好处。我为什么用这种方法呢？因为我在给大家演示动态库的用法，完了之后我就把libtest.so给删了，然后再重构ld.so.cache，对我的系统不会任何影响。倘若我是开发一款软件，或者给自己的系统DIY一个非常有用的功能模块，那么我更倾向于将libtest.so拷贝到/lib、/usr/lib目录下，或者我还有可能在/usr/local/lib/目录下新建一文件夹xxx，将so库拷贝到那儿去，并在/etc/ld.so.conf.d/目录下新建一文件mytest.conf，内容只有一行 “/usr/local/lib/xxx/libtest.so” ，再执行ldconfig。如果你之前还是不明白怎么解决那个“not found”的问题，那么现在总该明白了吧。

方法二：动态加载。

动态加载是非常灵活的，它依赖于一套Linux提供的标准API来完成。在源程序里，你可以很自如的运用API来加载、使用、释放so库资源。以下函数在代码中使用需要包含头文件：dlfcn.h

函数原型	说明
const char *dlerror(void)	当动态链接库操作函数执行失败时，dlerror可以返回出错信息，返回值为NULL时表示操作函数执行成功。
void *dlopen(const char *filename, int flag)	用于打开指定名字 (filename) 的动态链接库，并返回操作句柄。调用失败时，将返回NULL值，否则返回的是操作句柄。
void *dlsym(void *handle, char *symbol)	根据动态链接库操作句柄 (handle) 与符号 (symbol) ，返回符号对应的函数的执行代码地址。由此地址，可以带参数执行相应的函数。
int dlclose (void *handle)	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。2.2在程序中使用动态链接库函数。

函数原型

说明

```
const char *dlerror(void)
```

当动态链接库操作函数执行失败时，`dlerror`可以返回出错信息，返回值为NULL时表示操作函数执行成功。

```
void *dlopen(const char *filename, int flag)
```

用于打开指定名字（`filename`）的动态链接库，并返回操作句柄。调用失败时，将返回NULL值，否则返回的是操作句柄。

```
void *dlsym(void *handle, char *symbol)
```

根据动态链接库操作句柄（`handle`）与符号（`symbol`），返回符号对应的函数的执行代码地址。由此地址，可以带参数执行相应的函数。

```
int dlclose (void *handle)
```

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。2.2在程序中使用动态链接库

偶忍不住又要罗嗦一句了，相信俺，我的唠叨对大家是有好处。我为什么用这种方法呢？因为我在给大家演示动态库的用法，完了之后我就把libtest.so给删了，然后再重构ld.so.cache，对我的系统不会任何影响。倘若我是开发一款软件，或者给自己的系统DIY一个非常有用的功能模块，那么我更倾向于将libtest.so拷贝到/lib、/usr/lib目录下，或者我还有可能在/usr/local/lib/目录下新建一文件夹xxx，将so库拷贝到那儿去，并在/etc/ld.so.conf.d/目录下新建一文件mytest.conf，内容只有一行“/usr/local/lib/xxx/libtest.so”，再执行ldconfig。如果你之前还是不明白怎么解决那个“not found”的问题，那么现在总该明白了吧。

方法二：动态加载。

动态加载是非常灵活的，它依赖于一套Linux提供的标准API来完成。在源程序里，你可以很自如的运用API来加载、使用、释放so库资源。以下函数在代码中使用需要包含头文件：`dllfcn.h`

函数原型	说明
<code>const char *dlerror(void)</code>	当动态链接库操作函数执行失败时， <code>dlerror</code> 可以返回出错信息，返回值为NULL时表示操作函数执行成功。
<code>void *dlopen(const char *filename, int flag)</code>	用于打开指定名字（ <code>filename</code> ）的动态链接库，并返回操作句柄。调用失败时，将返回NULL值，否则返回的是操作句柄。
<code>void *dlsym(void *handle, char *symbol)</code>	根据动态链接库操作句柄（ <code>handle</code> ）与符号（ <code>symbol</code> ），返回符号对应的函数的执行代码地址。由此地址，可以带参数执行相应的函数。
<code>int dlclose (void *handle)</code>	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。2.2在程序中使用动态链接库函数。

函数原型

说明

```
const char *dlerror(void)
```

当动态链接库操作函数执行失败时，`dlerror`可以返回出错信息，返回值为NULL时表示操作函数执行成功。

```
void *dlopen(const char *filename, int flag)
```

用于打开指定名字（`filename`）的动态链接库，并返回操作句柄。调用失败时，将返回NULL值，否则返回的是操作句柄。

```
void *dlsym(void *handle, char *symbol)
```

根据动态链接库操作句柄（`handle`）与符号（`symbol`），返回符号对应的函数的执行代码地址。由此地址，可以带参数执行相应的函数。

```
int dlclose (void *handle)
```

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。2.2在程序中使用动态链接库

偶忍不住又要罗嗦一句了，相信俺，我的唠叨对大家是有好处。我为什么用这种方法呢？因为我在给大家演示动态库的用法，完了之后我就把libtest.so给删了，然后再重构ld.so.cache，对我的系统不会任何影响。倘若我是开发一款软件，或者给自己的系统DIY一个非常有用的功能模块，那么我更倾向于将libtest.so拷贝到/lib、/usr/lib目录下，或者我还有可能在/usr/local/lib/目录下新建一文件夹xxx，将so库拷贝到那儿去，并在/etc/ld.so.conf.d/目录下新建一文件mytest.conf，内容只有一行“/usr/local/lib/xxx/libtest.so”，再执行ldconfig。如果你之前还是不明白怎么解决那个“not found”的问题，那么现在总该明白了吧。

方法二：动态加载。

动态加载是非常灵活的，它依赖于一套Linux提供的标准API来完成。在源程序里，你可以很自如的运用API来加载、使用、释放so库资源。以下函数在代码中使用需要包含头文件：`dllfcn.h`

函数原型	说明
<code>const char *dlerror(void)</code>	当动态链接库操作函数执行失败时， <code>dlerror</code> 可以返回出错信息，返回值为NULL时表示操作函数执行成功。
<code>void *dlopen(const char *filename, int flag)</code>	用于打开指定名字（ <code>filename</code> ）的动态链接库，并返回操作句柄。调用失败时，将返回NULL值，否则返回的是操作句柄。
<code>void *dlsym(void *handle, char *symbol)</code>	根据动态链接库操作句柄（ <code>handle</code> ）与符号（ <code>symbol</code> ），返回符号对应的函数的执行代码地址。由此地址，可以带参数执行相应的函数。
<code>int dlclose (void *handle)</code>	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。2.2在程序中使用动态链接库函数。

函数原型

说明

```
const char *dlerror(void)
```

当动态链接库操作函数执行失败时，`dlerror`可以返回出错信息，返回值为NULL时表示操作函数执行成功。

```
void *dlopen(const char *filename, int flag)
```

用于打开指定名字（`filename`）的动态链接库，并返回操作句柄。调用失败时，将返回NULL值，否则返回的是操作句柄。

```
void *dlsym(void *handle, char *symbol)
```

根据动态链接库操作句柄（`handle`）与符号（`symbol`），返回符号对应的函数的执行代码地址。由此地址，可以带参数执行相应的函数。

```
int dlclose (void *handle)
```

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。2.2在程序中使用动态链接库

偶忍不住又要罗嗦一句了，相信俺，我的唠叨对大家是有好处。我为什么用这种方法呢？因为我在给大家演示动态库的用法，完了之后我就把libtest.so给删了，然后再重构ld.so.cache，对我的系统不会任何影响。倘若我是开发一款软件，或者给自己的系统DIY一个非常有用的功能模块，那么我更倾向于将libtest.so拷贝到/lib、/usr/lib目录下，或者我还有可能在/usr/local/lib/目录下新建一文件夹xxx，将so库拷贝到那儿去，并在/etc/ld.so.conf.d/目录下新建一文件mytest.conf，内容只有一行“/usr/local/lib/xxx/libtest.so”，再执行ldconfig。如果你之前还是不明白怎么解决那个“not found”的问题，那么现在总该明白了吧。

方法二：动态加载。

动态加载是非常灵活的，它依赖于一套Linux提供的标准API来完成。在源程序里，你可以很自如的运用API来加载、使用、释放so库资源。以下函数在代码中使用需要包含头文件：`dllfcn.h`

函数原型	说明
<code>const char *dlerror(void)</code>	当动态链接库操作函数执行失败时， <code>dlerror</code> 可以返回出错信息，返回值为NULL时表示操作函数执行成功。
<code>void *dlopen(const char *filename, int flag)</code>	用于打开指定名字（ <code>filename</code> ）的动态链接库，并返回操作句柄。调用失败时，将返回NULL值，否则返回的是操作句柄。
<code>void *dlsym(void *handle, char *symbol)</code>	根据动态链接库操作句柄（ <code>handle</code> ）与符号（ <code>symbol</code> ），返回符号对应的函数的执行代码地址。由此地址，可以带参数执行相应的函数。
<code>int dlclose (void *handle)</code>	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。2.2在程序中使用动态链接库函数。

函数原型

说明

```
const char *dlerror(void)
```

当动态链接库操作函数执行失败时，`dlerror`可以返回出错信息，返回值为NULL时表示操作函数执行成功。

```
void *dlopen(const char *filename, int flag)
```

用于打开指定名字（`filename`）的动态链接库，并返回操作句柄。调用失败时，将返回NULL值，否则返回的是操作句柄。

```
void *dlsym(void *handle, char *symbol)
```

根据动态链接库操作句柄（`handle`）与符号（`symbol`），返回符号对应的函数的执行代码地址。由此地址，可以带参数执行相应的函数。

```
int dlclose (void *handle)
```

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。2.2在程序中使用动态链接库

偶忍不住又要罗嗦一句了，相信俺，我的唠叨对大家是有好处。我为什么用这种方法呢？因为我在给大家演示动态库的用法，完了之后我就把libtest.so给删了，然后再重构ld.so.cache，对我的系统不会任何影响。倘若我是开发一款软件，或者给自己的系统DIY一个非常有用的功能模块，那么我更倾向于将libtest.so拷贝到/lib、/usr/lib目录下，或者我还有可能在/usr/local/lib/目录下新建一文件夹xxx，将so库拷贝到那儿去，并在/etc/ld.so.conf.d/目录下新建一文件mytest.conf，内容只有一行“/usr/local/lib/xxx/libtest.so”，再执行ldconfig。如果你之前还是不明白怎么解决那个“not found”的问题，那么现在总该明白了吧。

方法二：动态加载。

动态加载是非常灵活的，它依赖于一套Linux提供的标准API来完成。在源程序里，你可以很自如的运用API来加载、使用、释放so库资源。以下函数在代码中使用需要包含头文件：`dllfcn.h`

函数原型	说明
<code>const char *dlerror(void)</code>	当动态链接库操作函数执行失败时， <code>dlerror</code> 可以返回出错信息，返回值为NULL时表示操作函数执行成功。
<code>void *dlopen(const char *filename, int flag)</code>	用于打开指定名字（ <code>filename</code> ）的动态链接库，并返回操作句柄。调用失败时，将返回NULL值，否则返回的是操作句柄。
<code>void *dlsym(void *handle, char *symbol)</code>	根据动态链接库操作句柄（ <code>handle</code> ）与符号（ <code>symbol</code> ），返回符号对应的函数的执行代码地址。由此地址，可以带参数执行相应的函数。
<code>int dlclose (void *handle)</code>	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。2.2在程序中使用动态链接库函数。

函数原型

说明

```
const char *dlerror(void)
```

当动态链接库操作函数执行失败时，`dlerror`可以返回出错信息，返回值为NULL时表示操作函数执行成功。

```
void *dlopen(const char *filename, int flag)
```

用于打开指定名字（`filename`）的动态链接库，并返回操作句柄。调用失败时，将返回NULL值，否则返回的是操作句柄。

```
void *dlsym(void *handle, char *symbol)
```

根据动态链接库操作句柄（`handle`）与符号（`symbol`），返回符号对应的函数的执行代码地址。由此地址，可以带参数执行相应的函数。

```
int dlclose (void *handle)
```

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。2.2在程序中使用动态链接库

偶忍不住又要罗嗦一句了，相信俺，我的唠叨对大家是有好处。我为什么用这种方法呢？因为我在给大家演示动态库的用法，完了之后我就把libtest.so给删了，然后再重构ld.so.cache，对我的系统不会任何影响。倘若我是开发一款软件，或者给自己的系统DIY一个非常有用的功能模块，那么我更倾向于将libtest.so拷贝到/lib、/usr/lib目录下，或者我还有可能在/usr/local/lib/目录下新建一文件夹xxx，将so库拷贝到那儿去，并在/etc/ld.so.conf.d/目录下新建一文件mytest.conf，内容只有一行“/usr/local/lib/xxx/libtest.so”，再执行ldconfig。如果你之前还是不明白怎么解决那个“not found”的问题，那么现在总该明白了吧。

方法二：动态加载。

动态加载是非常灵活的，它依赖于一套Linux提供的标准API来完成。在源程序里，你可以很自如的运用API来加载、使用、释放so库资源。以下函数在代码中使用需要包含头文件：`dllfcn.h`

函数原型	说明
<code>const char *dlerror(void)</code>	当动态链接库操作函数执行失败时， <code>dlerror</code> 可以返回出错信息，返回值为NULL时表示操作函数执行成功。
<code>void *dlopen(const char *filename, int flag)</code>	用于打开指定名字（ <code>filename</code> ）的动态链接库，并返回操作句柄。调用失败时，将返回NULL值，否则返回的是操作句柄。
<code>void *dlsym(void *handle, char *symbol)</code>	根据动态链接库操作句柄（ <code>handle</code> ）与符号（ <code>symbol</code> ），返回符号对应的函数的执行代码地址。由此地址，可以带参数执行相应的函数。
<code>int dlclose (void *handle)</code>	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。2.2在程序中使用动态链接库函数。

函数原型

说明

```
const char *dlerror(void)
```

当动态链接库操作函数执行失败时，`dlerror`可以返回出错信息，返回值为NULL时表示操作函数执行成功。

```
void *dlopen(const char *filename, int flag)
```

用于打开指定名字（`filename`）的动态链接库，并返回操作句柄。调用失败时，将返回NULL值，否则返回的是操作句柄。

```
void *dlsym(void *handle, char *symbol)
```

根据动态链接库操作句柄（`handle`）与符号（`symbol`），返回符号对应的函数的执行代码地址。由此地址，可以带参数执行相应的函数。

```
int dlclose (void *handle)
```

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。2.2在程序中使用动态链接库

偶忍不住又要罗嗦一句了，相信俺，我的唠叨对大家是有好处。我为什么用这种方法呢？因为我在给大家演示动态库的用法，完了之后我就把libtest.so给删了，然后再重构ld.so.cache，对我的系统不会任何影响。倘若我是开发一款软件，或者给自己的系统DIY一个非常有用的功能模块，那么我更倾向于将libtest.so拷贝到/lib、/usr/lib目录下，或者我还有可能在/usr/local/lib/目录下新建一文件夹xxx，将so库拷贝到那儿去，并在/etc/ld.so.conf.d/目录下新建一文件mytest.conf，内容只有一行“/usr/local/lib/xxx/libtest.so”，再执行ldconfig。如果你之前还是不明白怎么解决那个“not found”的问题，那么现在总该明白了吧。

方法二：动态加载。

动态加载是非常灵活的，它依赖于一套Linux提供的标准API来完成。在源程序里，你可以很自如的运用API来加载、使用、释放so库资源。以下函数在代码中使用需要包含头文件：`dllfcn.h`

函数原型	说明
<code>const char *dlerror(void)</code>	当动态链接库操作函数执行失败时， <code>dlerror</code> 可以返回出错信息，返回值为NULL时表示操作函数执行成功。
<code>void *dlopen(const char *filename, int flag)</code>	用于打开指定名字（ <code>filename</code> ）的动态链接库，并返回操作句柄。调用失败时，将返回NULL值，否则返回的是操作句柄。
<code>void *dlsym(void *handle, char *symbol)</code>	根据动态链接库操作句柄（ <code>handle</code> ）与符号（ <code>symbol</code> ），返回符号对应的函数的执行代码地址。由此地址，可以带参数执行相应的函数。
<code>int dlclose (void *handle)</code>	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。2.2在程序中使用动态链接库函数。

函数原型

说明