C语言三方库的调用和编写

Java开发架构师 关注

2019.08.07 16:39:07 字数 2,117 阅读 772

1. 三方库相关指令

gcc -l

-l 参数就是用来指定程序要链接的库，-l 参数紧接着就是库名，那么库名跟真正的库文件名有什么关系呢？就拿数学库来说，他的库名是 m，他的库文件名是 libm.so，很容易看出，把库文件名的头lib和尾 .so 去掉就是库名了。好了现在我们知道怎么得到库名， 当我们自已要用到一个第三方提供的库名字 libtest.so，那么我们只要把 libtest.so 拷贝到 /usr/lib 里，编译时加上 -ltest 参数，我们就能用上 libtest.so 库了（当然要用 libtest.so 库里的函数，我们还需要与 libtest.so 配套的头文件）

gcc -L

放在/lib 和 /usr/lib 和 /usr/local/lib里的库直接用-l参数就能链接了，但如果库文件没放在这三个目录里，而是放在其他目录里，这时我们只用-l参数的话，链接还是会出错，出错信息大概是："/usr/bin/ld: cannot find -lxxx"，也就是链接程序 ld 在那3个目录里找不到libxxx.so。这时另外一个参数-L就派上用场了，比如常用的X11 的库，它在 /usr/X11R6/lib 目录下，我们编译时就要用 -L/usr/X11R6/lib -lX11 参数，-L 参数跟着的是库文件所在的**目录名**。再比如我们把libtest.so 放在/aaa/bbb/ccc 目录下，那链接参数就是 -L/aaa/bbb/ccc -ltest

gcc -I

-I 参数是用来指定头文件目录，/usr/include 目录一般是不用指定的，gcc 知道去那里找，但是如果头文件不在 /usr/include 里我们就要用 -I 参数指定了，比如头文件放在/myinclude 目录里，那编译命令行就要加上 -I/myinclude 参数了，如果不加你会得到一个 "xxxx.h: No such file or directory" 的错误。-I 参数可以用相对路径，比如头文件在当前目录，可以用 -I. 来指定。

gcc –shared

-shared该选项指定生成动态连接库（让连接器生成T类型的导出符号表，有时候也生成弱连接W类型的导出符号），不用该标志外部程序无法连接。相当于一个可执行文件

gcc -fPIC

-fPIC：表示编译为位置独立的代码，不用此选项的话编译后的代码是位置相关的所以动态载入时是通过代码拷贝的方式来满足不同进程的需要，而不能达到真正代码段共享的目的。

使用ar 命令创建或者操作静态库

ar archivefile objfile

archivefile：archivefile 是静态库的名称

objfile:objfile 是已.o 为扩展名的中间目标文件名，可以多个并列参数 意义

-r **将**objfile **文件插入静态库尾或者替换静态库中同名文件**

-x **从静态库文件中抽取文件**objfile

-t **打印静态库的成员文件列表**

-d **从静态库中删除文件**objfile

-s **重置静态库文件索引**

-v **创建文件冗余信息**

-c **创建静态库文件**

2. 调用三方库

使用数学库libm 举例

```
/* cos example */
Test.c
#include <stdio.h>      /* printf */
#include <math.h>       /* cos */

#define PI 3.14159265

int main ()
{
  double param, result;
  param = 60.0;
  result = cos ( param * PI / 180.0 );
  printf ("The cosine of %f degrees is %f.\n", param, result );
  return 0;
}
```

编译：gcc -g -Wall -o test1 test1.c –lm

通过l 指令，将libm 链接进程序

> [root@izbp1e2kit9gbr3xm1zerrz clib]# ./test1 The cosine of 60.000000 degrees is 0.500000.
>
> [root@izbp1e2kit9gbr3xm1zerrz clib]# find / -name "libm*"|grep libm*.so
>
> /usr/lib/i686/nosegneg/libm.so.6
>
> /usr/lib/libm.so.6
>
> /usr/lib64/libm.so.6
>
> /usr/lib64/libm.so
>
> /usr/share/ltrace/libm.so.conf [root@izbp1e2kit9gbr3xm1zerrz clib]# ldd ldd: missing file arguments
>
> Try `ldd --help' for more information. [root@izbp1e2kit9gbr3xm1zerrz clib]# ldd test1
>
> linux-vdso.so.1 => (0x00007ffd6716e000)
>
> libm.so.6 => /lib64/libm.so.6 (0x00007f8d4da70000) libc.so.6 => /lib64/libc.so.6 (0x00007f8d4d6a3000)
>
> /lib64/ld-linux-x86-64.so.2 (0x00007f8d4dd7d000)
>
> #include <stdio.h> #include <stdlib.h> #include <pthread.h> #include <sys/syscall.h> #include <sys/types.h>
>
> #define MAX_NUM 1000
>
> // thread callback
>
> static void* _run(void* arg) { int piyo = 10;

```
char* name = (char*)arg; int i;



for(i = 0; i < MAX_NUM; ++i) {

printf("[thread %u]: <%d, hi!> from %s\n", syscall(SYS_gettid), i,

name);

sleep(1);

}

return NULL;

}

#define _INT_PTX (3)

char* thread_name[] = {"A", "B", "C", "D", "E", "F", "G", "H"};



int main(void) { int i;

pthread_t tx[_INT_PTX];



puts("main beign");

printf("main thread id :%u\n",/*syscall(SYS_gettid)*/getpid());



for(i=0; i<_INT_PTX; ++i) {

//create threads

rt = pthread_create(tx+i, NULL, _run, thread_name[i]); if(rt < 0) {
```

```
printf("pthread_create create error! rt = %d, i=%d\n", rt, i); break;

}

}




for(i=0; i<_INT_PTX; ++i) { pthread_join(tx[i], NULL);

}

puts("end");



return 0;

}
```

编译命令

```
gcc -Wall -g -o FightingSaintBuddha.out FightingSaintBuddha.c -lpthread
```

3. 编写三方库

linux 下有两种库:动态库和静态库(共享库) 二者的不同点在于代码被载入的时刻不同。

静态库的代码在编译过程中已经被载入可执行程序,因此体积比较大。

动态库(共享库)的代码在可执行程序运行时才载入内存，在编译过程中仅简单的引用，因此代码体积比较小。

不同的应用程序如果调用相同的库,那么在内存中只需要有一份该动态库(共享库)的实例。

静态库和动态库的最大区别,静态情况下,把库直接加载到程序中,而动态库链接的时候,它只是保留接口,将动态库与程序代码独立,这样就可以提高代码的可复用度，和降低程序的耦合度。

静态库在程序编译时会被连接到目标代码中，**程序运行时将不再需要该静态库。**

动态库在程序编译时并不会被连接到目标代码中，而是在**程序运行是才被载入，因此在程序运行时还需要动态库存在**

自己编写动态库

不同的UNIX 系统,链接动态库方法，实现细节不一样

编译PIC 型.o 中间文件的方法一般是采用 C 语言编译器的-KPIC 或者-fpic 选项,有的

UNIX 版本C语言编译器默认带上了PIC标准.创建最终动态库的方法一般采用C语言编译器的-G或者-shared选项，或者直接使用工具ld创建。

最主要的是GCC 命令行的一个选项:

-shared该选项指定生成动态连接库（让连接器生成T类型的导出符号表，有时候也生成弱连接W类型的导出符号），不用该标志外部程序无法连接。相当于一个可执行文件

-fPIC：表示编译为位置独立的代码，不用此选项的话编译后的代码是位置相关的所以动态载入时是通过代码拷贝的方式来满足不同进程的需要，而不能达到真正代码段共享的目的。

-L.：表示要连接的库在当前目录中

-ltest：编译器查找动态连接库时有隐含的命名规则，即在给出的名字前面加上 lib，后面加上.so 来确定库的名称

LD_LIBRARY_PATH：这个环境变量指示动态连接器可以装载动态库的路径。当然如果有 root 权限的话，可以修改/etc/ld.so.conf 文件，然后调用

/sbin/ldconfig 来达到同样的目的，不过如果没有 root 权限，那么只能采用输出

LD_LIBRARY_PATH 的方法了。

```
#ifndef  CACULATE_HEAD   #define  CACULATE_HEAD

//add
```

```
int add(int a, int b); int sub(int a, int b); int div(int a, int b); int mul(int a, int b);

#endif

#include "caculate.h"

int add(int a, int b)

{

return (a+b);

}
```

```
int sub(int a, int b)
{
    return (a-b);
}

int div(int a, int b)
{
    return (int)(a/b);
}

int mul(int a, int b)
{
    return(a*b);
}
```

编译生成 so

```
gcc -g -Wall -shared -fpic caculate.c -o libcac.so
```

```c
#include <stdio.h>
#include "caculate.h"

int main(int argc, char* argv[])
{
    int a = 20;
    int b = 10;
    printf("%d + %d = %d\n", a, b, add(a, b));
```

gcc -g Wall test2.c -o test2 -L./-lcac

```
[root@izbp1e2kit9gbr3xm1zerrz clib]# ./test2
20 + 10 = 30
20 - 10 = 10
20 * 10 = 200
20 / 10 = 2
```

```c
//test3.c
#include <stdio.h>
#include <dlfcn.h>
#define DLL_FILE_NAME "libcac.so"
```

```
int main(int argc, char* argv[])
{
    void *handle = NULL;
```

1

> gcc -g -Wall test3.c -o test3.out -lbl

自己编写静态库

使用ar命令创建或者操作静态库

ar archivefile objfile

archivefile：archivefile 是静态库的名称

objfile:objfile 是已.o 为扩展名的中间目标文件名，可以多个并列

参数	意义
-r	将 objfile 文件插入静态库尾或者替换静态库中同名文件
-x	从静态库文件中抽取文件 objfile
-t	打印静态库的成员文件列表
-d	从静态库中删除文件 objfile
-s	重置静态库文件索引
-v	创建文件冗余信息
-c	创建静态库文件

> gcc -o caculate.o -c caculate.c

> ar -rcs libcac.a caculate.o

> gcc -g -Wall test2.c -o test4.out -L.libcac.a

小编推荐一个学习C/C++、Linux服务器技术交流的学习群（https://jq.qq.com/?_wv=1027&k=5s2103O），无论你是大牛还是小白，是想转行还是想入行多可以来了解一起学习一起进步一起交流技术！群内有很多干货和技术资料分享！喜欢的给个点赞和关注转发！