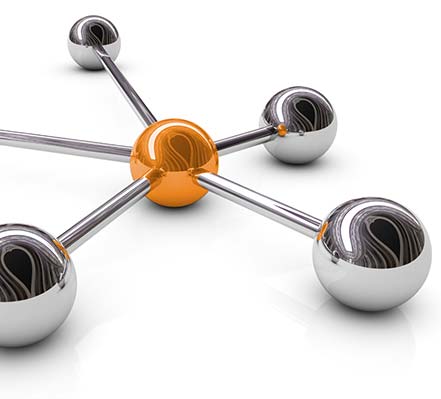
# [C编译: 动态连接库 (.so文件)](http://www.cnblogs.com/vamei/archive/2013/04/04/2998850.html)

作者：Vamei 出处：http://www.cnblogs.com/vamei 欢迎转载，也请保留这段声明。谢谢！

在“[纸上谈兵: 算法与数据结构](http://www.cnblogs.com/vamei/archive/2013/03/22/2974052.html)”中，我在每一篇都会有一个C程序，用于实现算法和数据结构 (比如栈和相关的操作)。在同一个程序中，还有用于测试的main()函数，结构体定义，函数原型，typedef等等。

这样的做法非常不“环保”。算法的实际运用和算法的实现混在一起。如果我想要重复使用之前的源程序，必须进行许多改动，并且重新编译。最好的解决方案是实现模块化: 只保留纯粹的算法实现，分离头文件，并编译一个库(library)。每次需要使用库的时候(比如使用栈数据结构)，就在程序中include头文件，连接库。这样，不需要每次都改动源程序。

我在这里介绍如何在UNIX环境中创建共享库 (shared library)。UNIX下，共享库以so为后缀(shared object)。共享库与Windows下的DLL类似，是在程序运行时动态连接。多个进程可以连接同一个共享库。



共享库

本文使用Ubuntu测试，使用gcc作为编译器。

### 程序清理

下面程序来自[纸上谈兵: 栈 (stack)](http://www.cnblogs.com/vamei/archive/2013/03/14/2960201.html)，是栈数据结构的C实现:

[复制代码](javascript:void(0);)

/\* By Vamei \*/

/\* use single-linked list to implement stack \*/

#include <stdio.h>

#include <stdlib.h>

typedef struct node \*position;

typedef int ElementTP;

// point to the head node of the list

typedef struct node \*STACK;

struct node {

ElementTP element;

position next;

};

STACK init\_stack(void);

void delete\_stack(STACK);

ElementTP top(STACK);

void push(STACK, ElementTP);

ElementTP pop(STACK);

int is\_null(STACK);

void main(void)

{

ElementTP a;

int i;

STACK sk;

sk = init\_stack();

push(sk, 1);

push(sk, 2);

push(sk, 8);

printf("Stack is null? %d\n", is\_null(sk));

for (i=0; i<3; i++) {

a = pop(sk);

printf("pop: %d\n", a);

}

printf("Stack is null? %d\n", is\_null(sk));

delete\_stack(sk);

}

/\*

\* initiate the stack

\* malloc the head node.

\* Head node doesn't store valid data

\* head->next is the top node

\*/

STACK init\_stack(void)

{

position np;

STACK sk;

np = (position) malloc(sizeof(struct node));

np->next = NULL; // sk->next is the top node

sk = np;

return sk;

}

/\* pop out all elements

\* and then delete head node

\*/

void delete\_stack(STACK sk)

{

while(!is\_null(sk)) {

pop(sk);

}

free(sk);

}

/\*

\* View the top frame

\*/

ElementTP top(STACK sk)

{

return (sk->next->element);

}

/\*

\* push a value into the stack

\*/

void push(STACK sk, ElementTP value)

{

position np, oldTop;

oldTop = sk->next;

np = (position) malloc(sizeof(struct node));

np->element = value;

np->next = sk->next;

sk->next = np;

}

/\*

\* pop out the top value

\*/

ElementTP pop(STACK sk)

{

ElementTP element;

position top, newTop;

if (is\_null(sk)) {

printf("pop() on an empty stack");

exit(1);

}

else {

top = sk->next;

element = top->element;

newTop = top->next;

sk->next = newTop;

free(top);

return element;

}

}

/\* check whether a stack is empty\*/

int is\_null(STACK sk)

{

return (sk->next == NULL);

}

[复制代码](javascript:void(0);)

上面的main()部分是用于测试，不属于功能模块，在创建库的时候应该去掉。

程序中的一些声明，会被重复利用。比如:

[复制代码](javascript:void(0);)

typedef struct node \*position;

typedef int ElementTP;

// point to the head node of the list

typedef struct node \*STACK;

struct node {

ElementTP element;

position next;

};

STACK init\_stack(void);

void delete\_stack(STACK);

ElementTP top(STACK);

void push(STACK, ElementTP);

ElementTP pop(STACK);

int is\_null(STACK);

[复制代码](javascript:void(0);)

这一段程序声明了一些结构体和指针，以及栈操作的函数原型。当我们其他程序中调用库时 (比如创建一个栈，或者执行pop操作)，同样需要写这些声明。我们把这些在实际调用中需要的声明保存到一个头文件mystack.h。在实际调用的程序中，可以简单的include该头文件，避免了每次都写这些声明语句的麻烦。

经过清理后的C程序为mystack.c:

[复制代码](javascript:void(0);)

/\* By Vamei \*/

/\* use single-linked list to implement stack \*/

#include <stdio.h>

#include <stdlib.h>  
#include "mystack.h"  
  
  
/\*

\* initiate the stack

\* malloc the head node.

\* Head node doesn't store valid data

\* head->next is the top node

\*/

STACK init\_stack(void)

{

position np;

STACK sk;

np = (position) malloc(sizeof(struct node));

np->next = NULL; // sk->next is the top node

sk = np;

return sk;

}

/\* pop out all elements

\* and then delete head node

\*/

void delete\_stack(STACK sk)

{

while(!is\_null(sk)) {

pop(sk);

}

free(sk);

}

/\*

\* View the top frame

\*/

ElementTP top(STACK sk)

{

return (sk->next->element);

}

/\*

\* push a value into the stack

\*/

void push(STACK sk, ElementTP value)

{

position np, oldTop;

oldTop = sk->next;

np = (position) malloc(sizeof(struct node));

np->element = value;

np->next = sk->next;

sk->next = np;

}

/\*

\* pop out the top value

\*/

ElementTP pop(STACK sk)

{

ElementTP element;

position top, newTop;

if (is\_null(sk)) {

printf("pop() on an empty stack");

exit(1);

}

else {

top = sk->next;

element = top->element;

newTop = top->next;

sk->next = newTop;

free(top);

return element;

}

}

/\* check whether a stack is empty\*/

int is\_null(STACK sk)

{

return (sk->next == NULL);

}

[复制代码](javascript:void(0);)

#include "..."; 语句将首先在工作目录寻找相应文件。如果使用gcc时，增加-I选项，将在-I提供的路径中寻找。

### 制作.so文件

我们的目标是制作共享库，即.so文件。

首先，编译stack.c:

$gcc -c -fPIC -o mystack.o mystack.c

-c表示只编译(compile)，而不连接。-o选项用于说明输出(output)文件名。gcc将生成一个目标(object)文件mystack.o。

注意-fPIC选项。PIC指Position Independent Code。共享库要求有此选项，以便实现动态连接(dynamic linking)。

生成共享库:

$gcc -shared -o libmystack.so mystack.o

库文件以lib开始。共享库文件以.so为后缀。-shared表示生成一个共享库。

这样，共享库就完成了。.so文件和.h文件都位于当前工作路径(.)。

### 使用共享库

我们编写一个test.c，来实际调用共享库:

[复制代码](javascript:void(0);)

#include <stdio.h>

#include "mystack.h"

/\*  
 \* call functions in mystack library  
 \*/

void main(void)

{

ElementTP a;

int i;

STACK sk;

sk = init\_stack();

push(sk, 1);

push(sk, 2);

push(sk, 8);

printf("Stack is null? %d\n", is\_null(sk));

for (i=0; i<3; i++) {

a = pop(sk);

printf("pop: %d\n", a);

}

printf("Stack is null? %d\n", is\_null(sk));

delete\_stack(sk);

}

[复制代码](javascript:void(0);)

注意，我们在程序的一开始include了mystack.h。

编译上述程序。编译器需要知道.h文件位置。

* 对于#include "..."，编译器会在当前路径搜索.h文件。你也可以使用-I选项提供额外的搜索路径，比如-I/home/vamei/test。
* 对于#include <...>，编译器会在默认include搜索路径中寻找。

编译器还需要知道我们用了哪个库文件，在gcc中:

* 使用-l选项说明库文件的名字。这里，我们将使用-lmystack (即libmystack库文件)。
* 使用-L选项说明库文件所在的路径。这里，我们使用-L. (即.路径)。

如果没有提供-L选项，gcc将在默认库文件搜索路径中寻找。

你可以使用下面的命令，来获知自己电脑上的include默认搜索路径:

$`gcc -print-prog-name=cc1` -v

获知库默认搜索路径:

$gcc -print-search-dirs

我们所需的.h和.so文件都在当前路径，因此，使用如下命令编译:

$gcc -o test test.c -lmystack -L.

将生成test可执行文件。

使用

$./test

执行程序

### 运行程序

尽管我们成功编译了test可执行文件，但很有可能不能执行。一个可能是权限问题。我们需要有执行该文件的权限，见[Linux文件管理背景知识](http://www.cnblogs.com/vamei/archive/2012/09/09/2676792.html)

另一个情况是:

./test: error while loading shared libraries: libmystack.so: cannot open shared object file: No such file or directory

这是因为操作系统无法找到库。libmystack.so位于当前路径，位于库文件的默认路径之外。尽管我们在编译时(compile time)提供了.so文件的位置，但这个信息并没有写入test可执行文件(runtime)。可以使用下面命令测试:

$ldd test

ldd用于显示可执行文件所依赖的库。显示:

linux-vdso.so.1 => (0x00007fff31dff000)

libmystack.so => not found

libc.so.6 => /lib/x86\_64-linux-gnu/libc.so.6 (0x00007fca30de7000)

/lib64/ld-linux-x86-64.so.2 (0x00007fca311cb000)

这说明test可执行文件无法找到它所需的libmystack.so库文件。

为了解决上面的问题，我们可以将.so文件放入默认搜索路径中。但有时，特别是多用户环境下，我们不享有在默认搜索路径写入的权限。

一个解决方案是设置LD\_LIBRARY\_PATH环境变量。比如:

$export LD\_LIBRARY\_PATH=.

这样，可执行文件执行时，操作系统将在先在LD\_LIBRARY\_PATH下搜索库文件，再到默认路径中搜索。环境变量的坏处是，它会影响所有的可执行程序。如果我们在编译其他程序时，如果我们不小心，很可能导致其他可执行文件无法运行。因此，LD\_LIBRARY\_PATH环境变量多用于测试。

另一个解决方案，即提供-rpath选项，将搜索路径信息写入test文件(rpath代表runtime path)。这样就不需要设置环境变量。这样做的坏处是，如果库文件移动位置，我们需要重新编译test。使用如下命令编译test.c:

$gcc -g -o test test.c -lmystack -L. -Wl,-rpath=.

-Wl表示，-rpath选项是传递给连接器(linker)。

test顺利执行的结果为:

Stack is null? 0

pop: 8

pop: 2

pop: 1

Stack is null? 1