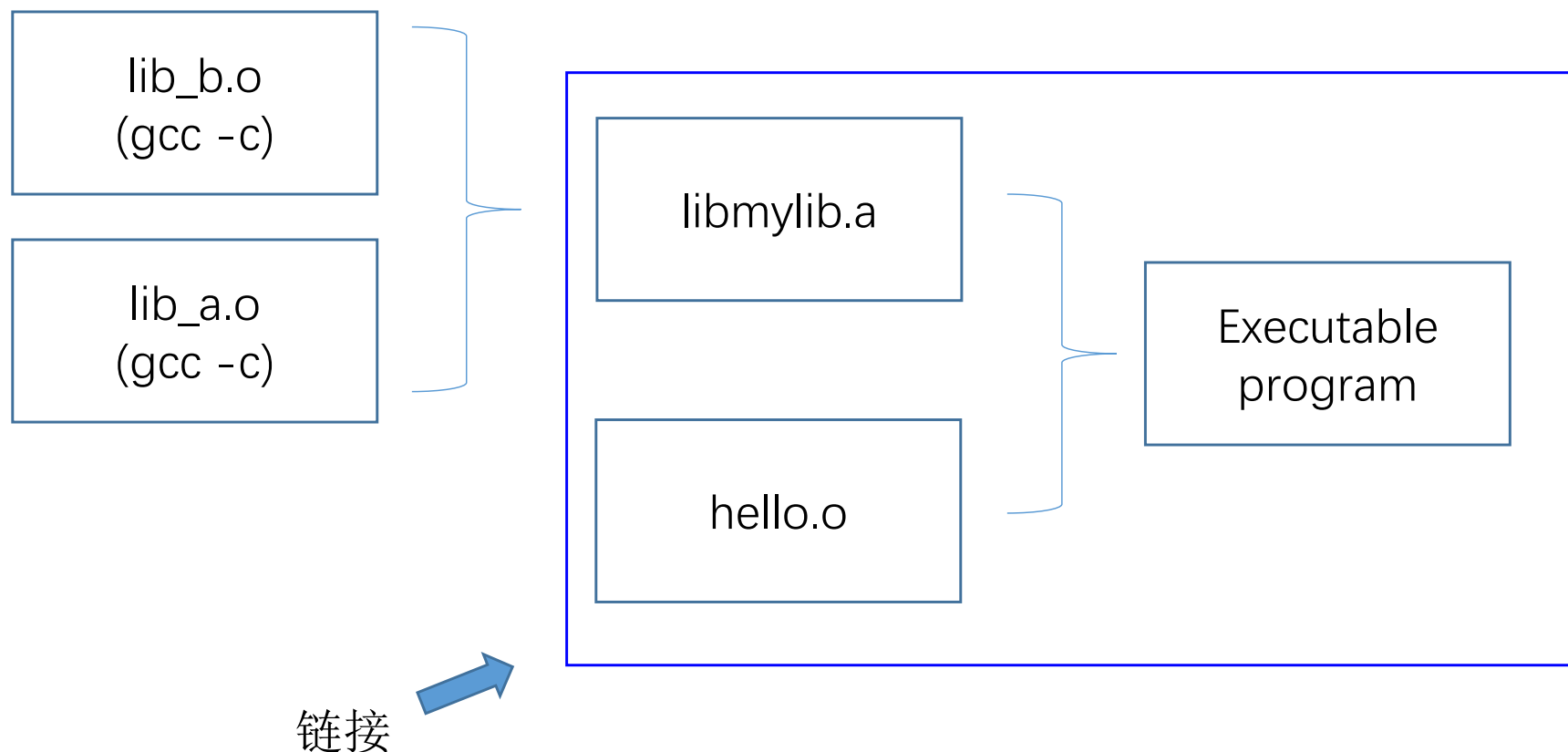




# 静态链接、动态链接

[hanzhuo@smail.nju.edu.cn](mailto:hanzhuo@smail.nju.edu.cn) 韩茁

# 静态链接



**静态链接**是指在编译阶段直接把静态库加入到可执行文件中去，这样可执行文件会比较大。而**动态链接**则是指链接阶段仅仅只加入一些描述信息，而程序执行时再从系统中把相应动态库加载到内存中去。

# 静态链接



```
C test.c  x  C my_lib.c  M Makefile
1
2 extern void hello_other_lib(int*, int*);
3 extern int share;
4 int main(){
5     int a = 1;
6     hello_other_lib(&a, &share);
7     return 0;
8 }
```

```
C test.c  C my_lib.c  x  M Makefile
1 int share = 100;
2 void hello_other_lib(int* a, int* b){
3     int c = *a;
4     *a = *b;
5     *b = c;
6 }
```

1. 使用gcc -c命令分别得到test.o以及my\_lib.o
3. 使用ar命令生成libmylib.a或直接使用gcc生成最终可执行文件。
4. 使用ld(或gcc) -L./ -lmylib通过静态链接库mylib以及test.o生成最后的可执行文件

```
gcc -c my_lib.c -o my_lib.o
gcc -c test.c -o test.o
```



```
ar crv libmylib.a my_lib.o
ld(gcc) -o test test.o -L./ -lmylib
```

```
gcc -o test test.o my_lib.o
```

生成静态库  
(库文件名为liba.a则在链接时为-la)  
libXXX.a, XXX为库名

直接链接

c: 建立库文件; r: 将文件插入库文件中; v: 程序执行时现时详细的信息



# 静态链接

使用objdump -h查看test.o、my\_lib.o(或libmylib.a)以及最终的可执行文件的所有节信息。

关注text字段和data  
字段的大小



```
my_lib.o:      file format elf64-x86-64

Sections:
Idx Name          Size      VMA           LMA           File off  Algn
---
0 .text           0000002c  0000000000000000  0000000000000000  00000040  2**0
CONTENTS, ALLOC, LOAD, READONLY, CODE
1 .data           00000004  0000000000000000  0000000000000000  0000006c  2**2
CONTENTS, ALLOC, LOAD, DATA
2 .bss            00000000  0000000000000000  0000000000000000  00000070  2**0
ALLOC
3 .comment         0000002e  0000000000000000  0000000000000000  00000070  2**0
CONTENTS, READONLY
4 .note.GNU-stack  00000000  0000000000000000  0000000000000000  0000009e  2**0
CONTENTS, READONLY
5 .eh_frame        00000038  0000000000000000  0000000000000000  000000a0  2**3
CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

```
objdump -h ${file_name}|grep -A1 -E "\.text|\.data|\.rodata"
```

```
=====test.o=====
0 .text           00000027  0000000000000000  0000000000000000  00000040  2**0
CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
1 .data           00000000  0000000000000000  0000000000000000  00000067  2**0
CONTENTS, ALLOC, LOAD, DATA
=====my_lib.o=====
0 .text           0000002c  0000000000000000  0000000000000000  00000040  2**0
CONTENTS, ALLOC, LOAD, READONLY, CODE
1 .data           00000004  0000000000000000  0000000000000000  0000006c  2**2
CONTENTS, ALLOC, LOAD, DATA
=====test=====
0 .text           00000053  000000000004000e8  000000000004000e8  000000e8  2**0
CONTENTS, ALLOC, LOAD, READONLY, CODE
--
2 .data           00000004  00000000000601000  00000000000601000  00001000  2**2
CONTENTS, ALLOC, LOAD, DATA
```



# 静态链接

链接时发生了什么？

1. 空间和地址分配
2. 符号解析和重定位



# 静态链接

链接时发生了什么？

```
linkers > static_linker > C a.c > main()
1  extern int shared;
2
3  int main(){
4      int a = 100;
5      swap(&a, &shared);
6  }

linkers > static_linker > C b.c > swap(int *, int *)
1  int shared = 1;
2  void swap(int *a, int *b)
3      *a^=*b^=*a^=*b;
4  }
```

两个C源代码a.c和b.c

执行指令

- gcc -c a.c -o a.o
- gcc -c b.c -o b.o
- gcc -o ab a.o b.o



# 静态链接

链接时发生了什么？

重定位信息

链接操作前

```
$ objdump -r a.o
```

```
a.o:      file format elf32-i386
```

```
RELOCATION RECORDS FOR [.text]:
OFFSET    TYPE           VALUE
0000001c  R_386_32              shared
00000027  R_386_PC32            swap
```



# 静态链接

链接时发生了什么？

objdump -d a.o

a.o: file format elf32-i386

Disassembly of section .text:

00000000 <main>:

0:	8d 4c 24 04	lea	0x4(%esp),%ecx
4:	83 e4 f0	and	\$0xfffffffff0,%esp
7:	ff 71 fc	pushl	0xfffffffffc(%ecx)
a:	55	push	%ebp
b:	89 e5	mov	%esp,%ebp
d:	51	push	%ecx
e:	83 ec 24	sub	\$0x24,%esp
11:	c7 45 f8 64 00 00 00	movl	\$0x64,0xfffffffff8(%ebp)
18:	c7 44 24 04 00 00 00	movl	\$0x0,0x4(%esp)
1f:	00		
20:	8d 45 f8	lea	0xfffffffff8(%ebp),%eax
23:	89 04 24	mov	%eax,(%esp)
26:	e8 fc ff ff ff	call	27 <main+0x27>
2b:	83 c4 24	add	\$0x24,%esp
2e:	59	pop	%ecx
2f:	5d	pop	%ebp
30:	8d 61 fc	lea	0xfffffffffc(%ecx),%esp
33:	c3	ret	

shared

swap

链接操作前，此时并没有分配存储器运行时地址，因为目前基址部分显示为

00000000





# 静态链接

链接时发生了什么？

链接操作后

```
$objdump -d ab
```

```
ab:      file format elf32-i386
```

```
Disassembly of section .text:
```

```
08048094 <main>:
```

```
8048094: 8d 4c 24 04
```

```
8048098: 83 e4 f0
```

```
804809b: ff 71 fc
```

```
804809e: 55
```

```
804809f: 89 e5
```

```
80480a1: 51
```

```
80480a2: 83 ec 24
```

```
80480a5: c7 45 f8 64 00 00 00
```

```
80480ac: c7 44 24 04 08 91 04
```

```
80480b3: 08
```

```
80480b4: 8d 45 f8
```

```
80480b7: 89 04 24
```

```
80480ba: e8 09 00 00 00
```

```
80480bf: 83 c4 24
```

```
80480c2: 59
```

```
80480c3: 5d
```

```
80480c4: 8d 61 fc
```

```
80480c7: c3
```

```
lea    0x4(%esp),%ecx
```

```
and     $0xffffffff0,%esp
```

```
pushl   0xffffffffc(%ecx)
```

```
push    %ebp
```

```
mov     %esp,%ebp
```

```
push    %ecx
```

```
sub     $0x24,%esp
```

```
movl    $0x64,0xffffffff8(%ebp)
```

```
movl    $0x8049108,0x4(%esp)
```

```
lea     0xffffffff8(%ebp),%eax
```

```
mov     %eax,(%esp)
```

```
call    80480c8 <swap>
```

```
add     $0x24,%esp
```

```
pop     %ecx
```

```
pop     %ebp
```

```
lea     0xffffffffc(%ecx),%esp
```

```
ret
```

# 动态链接



```
C my_lib.c x C my_lib.h Makefile.dynamic C dynamic_a.c C dynamic_b.c
1 #include<stdio.h>
2 #include"my_lib.h"
3
4 char* dynamic_lib_name = "MY_DYNAMIC_LIB";
5 int global_num = 0;
6 void hello_other_lib(const char* name, const char* yourname){
7     printf("Hello dynamic lib, I'm %s, and your name is %s, global num is %d\n", name, yourname, global_num);
8 }
```

```
1 #ifndef MY_LIB_H
2 #define MY_LIB_H
3 void hello_other_lib(const char* name, const char* yourname);
4 extern char* dynamic_lib_name;
5 extern int global_num;
6 #endif
```

```
gcc -fPIC -c -g my_lib.c -o my_lib.o
gcc -shared my_lib.o -o libd.so
gcc -c -g dynamic_a.c -o da.o
gcc -c -g dynamic_b.c -o db.o
gcc -o da da.o -L./ -ld
gcc -o db db.o -L./ -ld
```

```
C my_lib.c C my_lib.h Makefile.dynamic C dynamic_a.c x
1 #include"my_lib.h"
2
3 int main(){
4     int a = 5;
5     hello_other_lib("A", dynamic_lib_name);
6     global_num = 2;
7     hello_other_lib("A", dynamic_lib_name);
8     getchar();
9     return 0;
10 }
```

-fPIC 作用于编译阶段，告诉编译器产生与位置无关代码(Position-Independent Code)，则产生的代码中，没有绝对地址，全部使用相对地址，故而代码可以被加载器加载到内存的任意位置，都可以正确的执行。

-shared: 产生共享对象文件



# 动态链接

## 1.动态链接器自举

动态链接器本身也是一个不依赖其他共享对象的共享对象，需要完成自举。

## 2.装载共享对象

将可执行文件和链接器自身的符号合并成为全局符号表，开始寻找依赖对象。加载对象的过程可以看做图的遍历过程；新的共享对象加载进来后，其符号将合并入全局符号表；加载完毕后，全局符号表将包含进程动态链接所需全部符号。

## 3.重定位和初始化

链接器遍历可执行文件和共享对象的重定位表，将它们GOT/PLT中每个需要重定位的位置进行修正。完成重定位后，链接器执行.init段的代码，进行共享对象特有的初始化过程（例如C++里全局对象的构造函数）。

## 4.转交控制权

完成所有工作，将控制权转交给程序的入口开始执行。

ref: <https://www.cnblogs.com/linhaostudy/p/10544917.html>

《程序员的自我修养》——链接、装载与库



# 动态链接

## 可能会用到的命令

**ldd**: 查看引用的动态库的链接和名字

**objdump**和**readelf**: 查看目标代码，查看各节地址和符号表等信息

**gdb**:调试，查看运行时地址等信息

**cat /proc/pid/maps**: 查看内存映像，其中**pid**为进程**id**。可以看到是否正确加载到所需要的动态库以及程序的内存分布。



---

# Thanks !