

Mach-O 与静态链接

本文以《深入理解计算机系统》第七章「链接」为纲，以《程序员的自我修养》为主要参考，以 Mach-O 文件为研究对象，旨在整理静态链接相关的知识。

- 概述
 - 符号 & 模块
 - 静态链接
 - RIP-relative 寻址
- 中间文件的符号分析
 - Relocation Symbol Table
 - Symbol Table
- 可执行文件的符号分析

下一篇[Mach-O 与动态链接](#)分析 Mach-O 动态链接相关内容。

概述

本文的核心主题是静态链接，这一部分会对静态链接的内容做一个概述；在介绍链接之前，有必要先理清两个基本概念：符号、模块。

符号 & 模块

先说符号（symbol）。计算机世界里的符号概念起源于汇编，在汇编之前，远古大神的代码都是机器代码，机器代码充满了各种各样的数值，这些数值描述着指令、操作数、地址值；可以想到，全是数值的机器代码可读性非常

糟糕；对于地址值而言，当程序变更时，地址值就得重新计算，这种操作实在过于黑暗，于是先驱者发明了汇编语言，使用符号来帮助记忆，如下：

```
jmp foo
foo:
    addl $4, %ecx
```

`jmp`符号表示跳转指令，`foo`符号描述子程序的地址；无论`foo`前插入还是减少了代码，程序员无需关心它的具体地址值，汇编器在汇编时会自动计算，极大解放生产力。

对于表示地址值的符号而言，可以把它理解为键值对，符号是 `key`，地址值是 `value`。在之后的计算机发展中，「符号」表达的意思基本上没有变化。

再说模块。稍有规模的现代项目源码，常被按照功能或性质进行划分，分别形成不同的功能模块，不同的模块之间按照层级结构或其他结构来组织。譬如，在 C 语言，若干个变量和函数组成一个模块，存放在一个 `.c` 的源码文件中，然后这些源代码文件按照目录结构来组织。

对于 Objective-C 项目，每个 `.m` 或者 `.mm` 文件构成一个模块。

大规模软件往往拥有成千上万个模块，这些模块相互依赖又相互独立。软件模块化有很多好处，譬如更易理解、重用等；对于编译器而言，其好处是每个模块可以单独编译，改变部分代码无需重新编译整个程序。

当一个程序被切割成多个模块后，现代编译器会对每个模块进行单独编译，为每一个模块生成中间文件。这些中间文件终究会被组合形成一个单一的可执行文件。

将程序各个模块的中间文件组合形成一个单一的可执行文件并不是一件容易的事情，因为模块间往往会互相依赖，或者说它们之间存在通信。因此，从编译器的角度看，模块之间如何组合的问题可以归结为处理模块间通信的问题。对于静态语言 C/C++ 而言，模块间的通信方式有两种：一种是模块间的函数调用，另外一种则是模块间的变量访问；函数访问需知道目标函数的地址，变量访问也须知道目标变量的地址，所以这两种方式都可以归结为一种方式：模块间符号的引用。

静态链接

上文铺垫了符号和模块的概念，现在来说说静态链接。所谓静态链接，其本质是把程序各个模块的中间文件粘在一起，拼装成一个整体；换句话说，以模块的中间文件为输入，产生一个新的 Mach-O 文件（往往是可执行文件）。链接的主要内容就是把各个模块之间相互引用的部分都处理好，使得各个模块之间能够正确地衔接，从原理上讲，即把一些指令对其他符号地址的引用加以修正。按照《程序员的自我修养》的说法，静态链接主要过程包括：

- 地址和空间分配（Address and Storage Allocation）
- 符号决议（Symbol Resolution）
- 重定位（Relocation）

其中最核心也最值得拧出来分析的是「重定位」过程。静

态链接的重定位是围绕符号进行的，私以为，搞清楚了 Mach-O 文件中的符号，也就搞清楚了静态链接。

接下来的重心是对中间文件以及可执行文件的符号进行分析。

RIP-relative 寻址

本文所在环境的系统架构是 x86-64，很多指令的寻址方式是 RIP-relative 寻址。虽然笔者对汇编不甚熟悉，但是为了后续分析和阅读方便，还是得花些笔墨整理一下 RIP-relative 寻址相关内容。

RIP 的全拼是：Relative Instruction Pointer

按照笔者的粗浅理解，基于 RIP 计算目标地址时，目标地址等于当前指令的下一条指令所在地址加上偏移量。简单来说，若看到如下二进制的反汇编内容：

```
00000000000001fcd  jmpq  0x2d(%rip)
00000000000001fd3  nop
```

则第一行代码 jmpq 的跳转目标地址是： $0x1fd3 + 0x2d = 0x2000$ 。

关于RIP-relative 的更多内容可参考：

- [64位下的相对指令地址](#)
- [Intel x86-64 Manual Vol2](#)

中间文件的符号分析

Mach-O 的文件类型有很多种（详见[mach-o/loader.h](#) 里以MH_为前缀的 type 宏），常见的有：

- MH_OBJECT: 中间文件
- MH_EXECUTE: 可执行文件

曾在[Mach-O 简单分析](#)中分析了 Mach-O 文件的结构，相关内容不再赘述。

上文多处提到「中间文件」，它其实就是一种MH_OBJECT类型的 Mach-O 文件，还常被称作「中间目标文件」「可重定位文件」，通常以.o为后缀，

这一部分重点分析中间文件与符号相关的结构。将使用下面两个源代码文件 a.c 和 b.c 作为例子展开分析：

```
extern int shared;
void swap(int *a, int *b);
int main() {
    int a = 100;
    swap(&a, &shared);
    return 0;
}
```

```
int shared = 42;
void swap(int *a, int *b) {
    int temp = &a;
    &a = &b;
    &b = temp;
}
```

从代码中可以看到，b.c总共定义了两个全局符号，一个是变量shared，另外一个函数swap；a.c里面定义了一个全局符号main；后者引用了前者的俩符号。

使用 gcc 将这俩文件分别编译成目标文件 a.o 和 b.o：

先看看a.o的 __TEXT __text 的反汇编内容：

0x0 (_main):			
000001D8	55	pushq	%rbp
000001D9	4889E5	movq	%rsp, %rbp
000001DC	4883EC10	subq	\$0x10, %rsp
000001E0	488D7DF8	leaq	-0x8(%rbp), %rdi
000001E4	488B3500000000	movq	_shared(%rip), %rsi
000001EB	C745FC00000000	movl	\$_0x0 (_main)", -0x4(%rbp)
000001F2	C745F864000000	movl	\$0x64, -0x8(%rbp)
000001F9	B000	movb	\$_0x0 (_main)", %al
000001FB	E800000000	callq	_swap
00000200	31C9	xorl	%ecx, %ecx
00000202	8945F4	movl	%eax, -0xc(%rbp)
00000205	89C8	movl	%ecx, %eax
00000207	4883C410	addq	\$0x10, %rsp
0000020B	5D	popq	%rbp
0000020C	C3	ret	

本文对 Mach-O 文件的查看主要使用 MachOView 工具。

图中使用红框标记了a.o中的两处符号引用，分别对应 movq 操作和 callq 操作：

- 48 8B35 00000000: 其中48是 movq 的操作码，00000000描述_shared的符号值（地址值）
- E8 00000000: 其中E8是 callq 的的操作码，00000000描述_swap的符号值（地址值）

gcc编译器中的符号名，通常都含有_前缀。

可以看到，在a.o的代码段，该符号对应的地址值都被置为0。

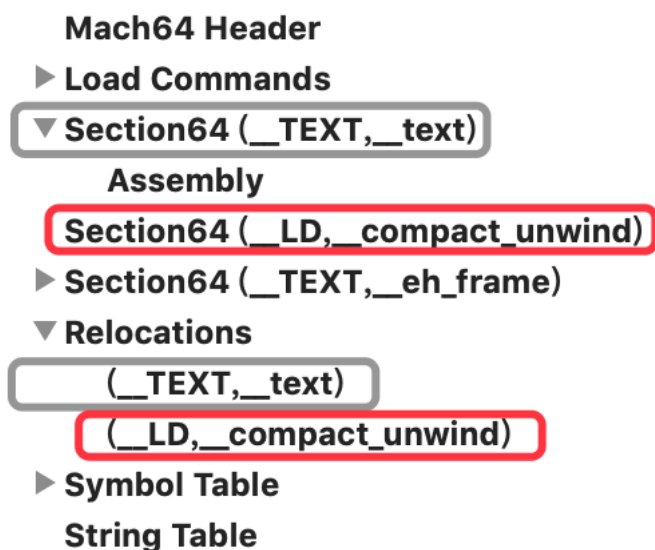
问题来了，编译器对中间文件进行链接时，如何知道该对哪些指令进行地址调整呢？这些指令的哪些部分要被调整呢？又该如何调整呢？

Relocation Symbol Table 正是解决这个问题的。

Relocation Symbol Table

在每个可重定位的 Mach-O 文件中，有一个叫重定位（Relocation）的区域，专门用来保存这些和重定位相关的信息。

某个 section 如果内含需要被重定位的字节，就会有一个 relocation table 与此对应：



在[Mach-O 简单分析](#)里介绍过 section 的结构（相关结构体定义于[mach-o/loader.h](#)中的 section_64），其中有两个字段描述了其对应的relocation table：

- `reloff`: relocation table 的 file offset
- `nreloc`: relocation table 的 entry 数量

Relocation table 可以看作是一个 relocation entry 的数组，每个 relocation entry 占 8 个字节：

Offset	Data	Description	Value
00000270	00000024	Address	0x24
00000274	2D000002	Symbol	_swap
		Type	X86_64_RELOC_BRANCH
		External	True
		PCRelative	True
		Length	4
00000278	0000000F	Address	0xF
0000027C	3D000001	Symbol	_shared
		Type	X86_64_RELOC_GOT_LOAD
		External	True
		PCRelative	True
		Length	4

对应结构体是[relocation_info](#):

```
struct relocation_info {
    int32_t    r_address;
    uint32_t   r_symbolnum:24,
               r_pcrel:1,
               r_length:2,
               r_extern:1,
               r_type:4;
};
```

对这几个字段稍作说明：

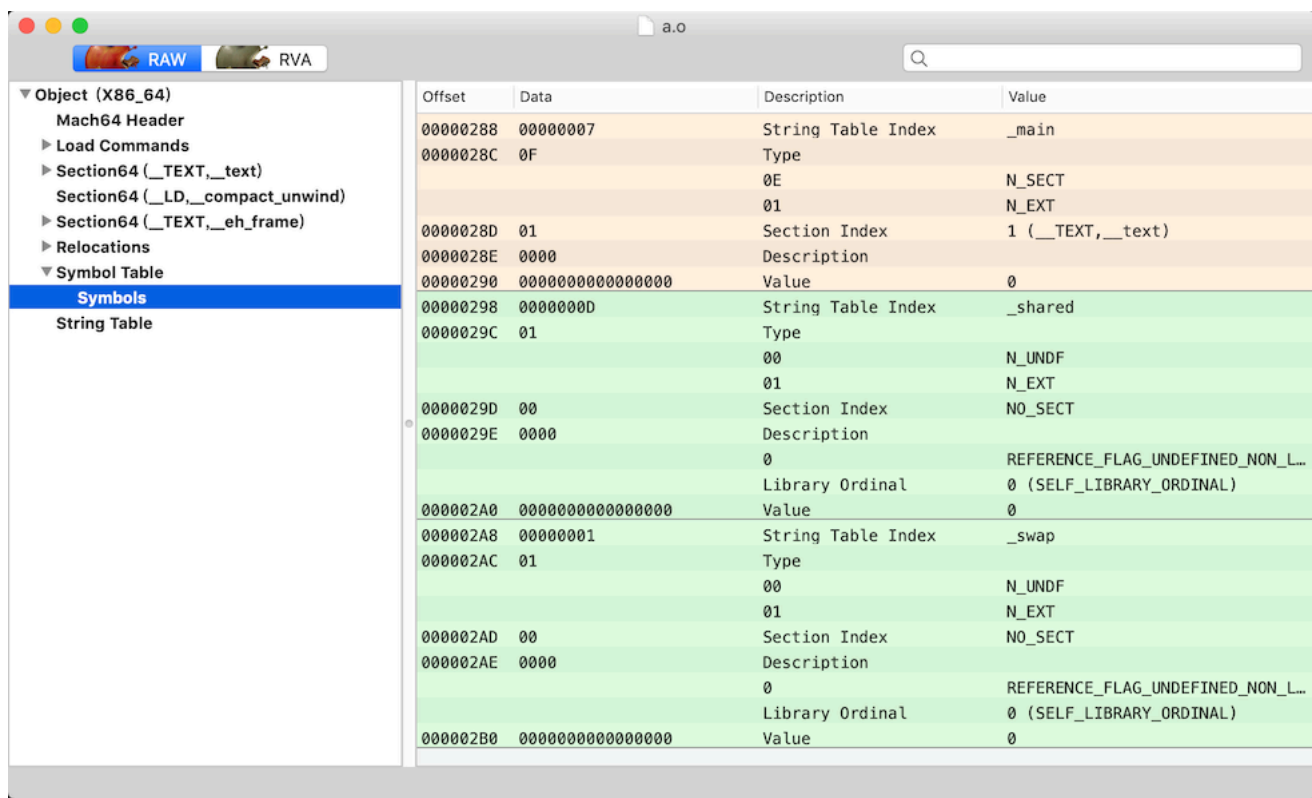
- `r_address`和`r_length`字段描述了需要被 relocation 的字节范围，其中`r_address`是相对于 section 的偏移量
- `r_pcrel`表示地址值是 PC 相对地址值
- `r_extern`标记该符号是否是外部符号
- `r_symbolnum`, index 值，对于外部符号，它描述了符

号在 symbol table 中的位置；如果是内部符号，它描述了符号所在的 section 的index

- r_type, 符号类型

Symbol Table

从上文的r_symbolnum可以看出， relocation_info并未完整描述符号信息，它只是告诉链接器哪些指令需要调整地址。符号的具体信息（包括符号名等）在 symbol table 中：



Offset	Data	Description	Value
00000288	00000007	String Table Index	_main
0000028C	0F	Type	
	0E		N_SECT
	01		N_EXT
0000028D	01	Section Index	1 (__TEXT,__text)
0000028E	0000	Description	
00000290	0000000000000000	Value	0
00000298	0000000D	String Table Index	_shared
0000029C	01	Type	
	00		N_UNDF
	01		N_EXT
0000029D	00	Section Index	NO_SECT
0000029E	0000	Description	
	0		REFERENCE_FLAG_UNDEFINED_NON_L...
		Library Ordinal	0 (SELF_LIBRARY_ORDINAL)
000002A0	0000000000000000	Value	0
000002A8	00000001	String Table Index	_swap
000002AC	01	Type	
	00		N_UNDF
	01		N_EXT
000002AD	00	Section Index	NO_SECT
000002AE	0000	Description	
	0		REFERENCE_FLAG_UNDEFINED_NON_L...
		Library Ordinal	0 (SELF_LIBRARY_ORDINAL)
000002B0	0000000000000000	Value	0

Symbol table 由谁定义呢？或者说，链接器是如何找到 symbol table 的呢？链接器是通过 LC_SYMTAB 这个 load command 找到 symbol table 的，关于 load command，在[Mach-O 简单分析](#)里有过介绍，此处不再赘述；

LC_SYMTAB 对应的 command 结构体如下：

```
struct symtab_command {
```

```
uint32_t cmd;
uint32_t cmdsize;
uint32_t symoff;
uint32_t nsyms;
uint32_t stroff;
uint32_t strsize;
};
```

这个命令告诉了链接器（无论是本文所述的静态链接器，还是后面博客要提到的动态链接器）symbol table 和 string table 的位置信息；symtab_command这个结构体比较简单，symoff和nsyms指示了符号表的位置和条目，stroff和strsize指示了字符串表的位置和长度。

每个 symbol entry 长度是固定的，其结构由内核定义，详见[nlist.h](#)：

```
struct nlist_64 {
    union {
        uint32_t n_strx;
    } n_un;
    uint8_t n_type;
    uint8_t n_sect;
    uint16_t n_desc;
    uint64_t n_value;
};
```

结构体nlist_64（或nlist）描述了符号的基本信息，xnu用5个字段描述了symbol信息，其中n_un、n_sect、n_value比较容易理解：

- `n_un`, 符号的名字 (在一个 Mach-O 文件里, 具有唯一性)
- `n_sect`, 符号所在的 section index (内部符号有效值从 1 开始, 最大为 255)
- `n_value`, 符号的地址值 (在链接过程中, 会随着其 section 发生变化)

`n_type`和`n_desc`表达的意思稍微复杂点; 都是多功能组合字段, 其中, 对于中间文件而言, `n_desc`没啥意义, 此乃个人理解。如下关于`n_type`的信息也是我的个人梳理, 主要参考[kernel/nlist_64](#)和[nlist.h](#)。`n_type`是一个 8 bit 的复合字段:

- `bit[5:8]`: 如果不为 0, 表示这是一个与调试有关的符号, 值意义类型详见[mach-o/stab.h](#)
- `bit[4:5]`: 若为 1, 则表示该符号是私有的 (外部符号)
- `bit[1:4]`: 符号类型
 - `N_UNDF (0x0)`: 未定义
 - `N_ABS (0x2)`: 符号地址指向到绝对地址, 链接器后期不会再修改
 - `N_SECT (0xe)`: 本地符号, 即符号定义于当前 Mach-O
 - `N_PBUD (0xc)`: 预绑定符号
 - `N_INDR (0xa)`: 表示该符号和另一个符号是同一个, `n_value`指向到 string table, 即该同名符号的名字
- `bit[0:1]`: 表示这是外部符号, 即该符号要么定义在

外部，要么定义在本地但是可以被外部使用

有了 relocation table 和 symbol table，链接器就与足够的信息进行链接处理了。

可执行文件的符号分析

先使用ld工具（静态链接器）对如上 a.o、b.o 进行链接，生成可执行文件：

```
ld a.o b.o -macosx_version_min 10.14 -o ab.out -lSystem
```

使用 MachOView 工具查看可执行文件 ab.out 的代码段 __TEXT __text 的反汇编内容，如下：

Offset	Data	Description
0x1F70 (_main):		
00000F70	55	pushq %rbp
00000F71	4889E5	movq %rsp, %rbp
00000F74	4883EC10	subq \$0x10, %rsp
00000F78	488D7DF8	leaq -0x8(%rbp), %rdi
00000F7C	488D357D0000000	leaq "0x2000 (_shared)"(%rip), %rsi
00000F83	C745FC000000000	movl \$0x0, -0x4(%rbp)
00000F8A	C745F8640000000	movl \$0x64, -0x8(%rbp)
00000F91	E80A0000000	callq "0x1FA0 (_swap)"
00000F96	31C0	xorl %eax, %eax
00000F98	4883C410	addq \$0x10, %rsp
00000F9C	5D	popq %rbp
00000F9D	C3	ret
00000F9E	90	nop
00000F9F	90	nop
0x1FA0 (_swap):		
00000FA0	55	pushq %rbp
00000FA1	4889E5	movq %rsp, %rbp
00000FA4	48897DF8	movq %rdi, -0x8(%rbp)
00000FA8	488975F0	movq %rsi, -0x10(%rbp)
00000FAC	488B75F8	movq -0x8(%rbp), %rsi
00000FB0	8B06	movl (%rsi), %eax
00000FB2	8945EC	movl %eax, -0x14(%rbp)
00000FB5	488B75F0	movq -0x10(%rbp), %rsi
00000FB9	8B06	movl (%rsi), %eax
00000FBB	488B75F8	movq -0x8(%rbp), %rsi
00000FBF	8906	movl %eax, (%rsi)
00000FC1	8B45EC	movl -0x14(%rbp), %eax
00000FC4	488B75F0	movq -0x10(%rbp), %rsi
00000FC8	8906	movl %eax, (%rsi)
00000FCA	5D	popq %rbp
00000FCB	C3	ret

图中红线部分分别是符号 `_shared` 和 `_swap` 对应的地址，链接前，a.o 中此两处的地址值均为 0；在 ab.out 中，链接器根据 a.o 的 relocation table 的信息，对此两处地址进行了调整，将它们修改为有效地址。

花些时间分析修正后 `_shared` 和 `_swap` 所对应的文件偏移地址。

先看 `_shared`，其所在指令的下一条指令相对于文件的偏

移地址是 0x00000F83，相对偏移是 0x0000007D（小端），计算得到的_shared符号的地址值等于 0x00001000，该地址值对应的是 ab.out 中的 __DATA __data：

▼ Executable (X86_64)	pFile	Data LO
Mach64 Header	00001000	2A 00 00 00
▶ Load Commands		
▼ Section64 (__TEXT,__text)		
Assembly		
▶ Section64 (__TEXT,__eh_frame)		
Section64 (__DATA,__data)		

该地址所存储的值 0x0000002A（小端）恰好等于 42。

对于_swap符号也是类似，其所在指令的下一条指令相对于文件的偏移地址是 0x00000F96，相对偏移是 0x00000000A，计算得到的目标地址值等于 0x00000FA0，恰好是_swap子程序的起始地址。

另外一个需要注意到的事实是：ab.out 中再也没有 relocation table 了，这不难理解，ab.out 中的符号都得到了重定位，relocation table 已经没有存在的必要了。

Relocation table 没有了，symbol table 呢？令人震惊的是，ab.out 的 symbol table 中条目更多了：

00002050	00000002	String Table Index	__mh_execute_header
00002054	03	Type	
	02	N_ABS	
	01	N_EXT	
00002055	01	Section Index	1 (__TEXT,__text)
00002056	0010	Description	
	0010	REFERENCED_DYNAMICALY	
00002058	0000000000001000	Value	4096
00002060	00000016	String Table Index	_main
00002064	0F	Type	
	0E	N_SECT	
	01	N_EXT	
00002065	01	Section Index	1 (__TEXT,__text)
00002066	0000	Description	
00002068	0000000000001F70	Value	8048 (\$+0)
00002070	0000001C	String Table Index	_shared
00002074	0F	Type	
	0E	N_SECT	
	01	N_EXT	
00002075	03	Section Index	3 (__DATA,__data)
00002076	0000	Description	
00002078	0000000000002000	Value	8192 (\$+0)
00002080	00000024	String Table Index	_swap
00002084	0F	Type	
	0E	N_SECT	
	01	N_EXT	
00002085	01	Section Index	1 (__TEXT,__text)
00002086	0000	Description	
00002088	0000000000001FA0	Value	8096 (\$+48)
00002090	0000002A	String Table Index	dyld_stub_binder
00002094	01	Type	
	00	N_UNDF	
	01	N_EXT	
00002095	00	Section Index	NO_SECT
00002096	0100	Description	
	0	REFERENCE_FLAG_UNDEFINED_NON_LAZY	
		Library Ordinal	1 (libSystem.B.dylib)
	0100	N_SYMBOL_RESOLVER	
00002098	0000000000000000	Value	0

其中的_main、_shared、_swap我们是熟悉的，另外两个是啥？没有查到关于__mh_execute_header特别权威的资料，目测与虚拟地址有关，简单来说，Mach-O 某个字节的虚拟地址等于__mh_execute_header的地址值加上该字节在 Mach-O 文件中的 file offset；对于 dyld_stub_binder，它和动态链接有关，后面博客讲动态链接时再说。

现在的问题是：ab.out 中_main、_shared、_swap这几个 symbol entry 存在的意义是啥？

我的理解是：如果从程序正常运行的角度来看，这几个符号没啥用。事实上，使用strip工具可以将这几个 symbol entry 从 symbol table 中抹掉。

最后，我的总体感受是：理解静态链接的关键在于理解符

号，还是蛮容易的。