Mach-O 与动态链接

写在前面

Mach-O 简单分析描述了 Mach-O 文件的基本结构；Mach-O 与静态链接 概述了符号，分析了符号表（symbol table），这两篇算是本文的前置知识，本文旨在分析动态链接相关内容。

关于动态链接，《深入理解计算机系统》和《程序员的自我修养》有着非常棒的分析，但这两本书都是站在 Linux

生态 ELF (Executable and Linkable Format) 的视角分析问题；本文借鉴这两本书，站在 Mach-O 的角度梳理我学习到的内容。

原本想在本文中将动态链接的方方面面的内容都装进去，慢慢发现这不是一个好主意。相较于静态链接，动态链接相关内容复杂得多，也更有意思一些，可以从各个视角去研究窥探；换句话说，随便挑一个切入点进行展开，都能完成一篇博客。

本文和Mach-O 与静态链接类似，只是站在 Mach-O 视角，结合笔者自己的理解，将 Mach-O 本身与动态链接相关的结构给串起来，以期望对动态链接有一个基本的理解，重点仍然放在 Mach-O 文件本身上，基本上不涉及 xnu 和 dyld 的源码分析。

> 分析 xnu 和 dyld 能帮助更全面准确地理解动态链接，但这不是本文的任务。

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：

```
0000000000001fcd  jmpq  0x2d(%rip)
0000000000001fd3  nop
```

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

关于RIP-relative 的更加内容可参考：

- 64位下的相对指令地址
- Intel x86-64 Manual Vol2

间接寻址

除了 RIP-relative 寻址，也得提一下间接寻址。间接寻址是相对于直接寻址而言的，即目标地址并不是计算得到的地址值，而是该地址值存储的数据。

简单来说，如果看到如下二进制的反汇编内容：

```
0000000000001fcd  jmpq  *0x2d(%rip)
0000000000001fd3  nop
```

> 对于间接寻址，反汇编代码中，地址值前有一个*

则第一行代码 jmpq 的跳转目标地址是 0x2000 (0x1fd3 + 0x2d) 里存储的内容，并非 0x2000 本身。

几个基本概念

在展开分析之前，先罗列本文高频出现的一些概念：

- 镜像：xnu、dyld 都将 Mach-O 文件看作镜像（image），本文所指的镜像即 Mach-O 文件
- 目标文件：即只编译未链接的可重定位文件
- dylib：动态链接库，在 ELF 生态中，常被称作「共享对象」，或者「共享文件」，本文称作 dylib，或者 dylibs
- dyld：dyld 是 Apple 生态操作系统（macOS、iOS）的动态链接器，本文直接使用 dyld 指代 Mach-O 的动态链接器

引子

静态链接比较简单，原理上也容易理解，实践上却存在很多问题，典型问题有两点：

1. 极大浪费磁盘和内存空间
2. 给程序的更新、部署和发布带来很多麻烦

稍微描述一下第二点。比如程序 Program1 所使用的 Lib.o 是由一个第三方厂商提供的，当该厂商更新了 Lib.o 的时候，那么 Program1 的开发者就要拿到最新版的 Lib.o，然后将其与 Program1.o 链接后，将新的 Program1 整个发布给用户。即一旦程序有任何模块的更新，整个程序就得重新链接、发布给用户。

动态链接是对这两个问题的解决方案。所谓动态链接，简单地讲，就是不对那些组成程序的目标文件进行链接，等到程序要运行时才进行链接。也就是说，把链接这个过程推迟到运行时再进行，这就是动态链接（Dynamic Linking）的基本思想。

动态链接的背景和基本思想理解起来蛮容易的，但实践中需要处理不少问题。本文以一个具体的 case 引出 Mach-O 动态链接中值得我们关心的问题。

首先，有一个文件 say.c：

```
#include <stdio.h>

char *kHelloPrefix = "Hello";

void say(char *prefix, char *name)
{
    printf("%s, %s\n", prefix, name);
}
```

该模块很简单，定义了两个符号：常量字符串`kHelloPrefix`，以及函数`say`。使用 gcc 把`say.c`编译成 dylib：

```
$ gcc -fPIC -shared say.c -o libsay.dylib
```

再定义一个使用 say 模块的 main.c：

```
void say(char *prefix, char *name);
extern char *kHelloPrefix;

int main(void)
{
    say(kHelloPrefix, "Jack");
```

```
    return 0;
}
```

把 main.c 编译成可重定位中间文件（只编译不链接）：

```
$ gcc -c main.c -o main.o
```

此时的 main.o 是不可执行的，需要使用链接器 ld 将 sayHello 链接进来：

```
$ ld main.o -macosx_version_min 10.14 -o main.out -lSystem
```

这样就生成了可执行文件 main.out，执行该文件，打印「Hello, Jack」。此时若使用`xcrun dyldinfo -dylibs`查看 main.out 的依赖库，会发现有两个依赖库：

```
$ xcrun dyldinfo -dylibs main.out
attributes     dependent dylibs
               /usr/lib/libSystem.B.dylib
               libsay.dylib
```

这两个动态库的依赖在 Mach-O 文件中对应两条 type 为`LC_LOAD_DYLIB`的 load commands，使用`otool -l`查看

如下：

```
Load command 12
          cmd LC_LOAD_DYLIB
      cmdsize 56
         name /usr/lib/libSystem.B.dylib (offset 24)
   time stamp 2 Thu Jan  1 08:00:02 1970
      current version 1252.200.5
compatibility version 1.0.0

Load command 13
          cmd LC_LOAD_DYLIB
      cmdsize 40
         name libsay.dylib (offset 24)
   time stamp 2 Thu Jan  1 08:00:02 1970
      current version 0.0.0
```

> LC_LOAD_DYLIB命令的顺序和 ld 的链接顺序一致。

LC_LOAD_DYLIB命令参数描述了 dylib 的基本信息，结构比较简单：

```
struct dylib {
    union lc_str  name;
    uint32_t timestamp;
    uint32_t current_version;
    uint32_t compatibility_version;
};
```

无论是静态链接，还是动态链接，符号都是最重要的分析对象；来看看 main.out 的符号表（symbol table）：

00002078	00000002	String Table Index	__mh_execute_header
0000207C	03	Type	
		02	N_ABS
		01	N_EXT
0000207D	01	Section Index	1 (__TEXT,__text)
0000207E	0010	Description	
		0010	REFERENCED_DYNAMICALLY
00002080	0000000000001000	Value	4096
00002088	00000016	String Table Index	_main
0000208C	0F	Type	
		0E	N_SECT
		01	N_EXT
0000208D	01	Section Index	1 (__TEXT,__text)
0000208E	0000	Description	
00002090	0000000000001F90	Value	8080 ($+0)
00002098	0000001C	String Table Index	_kHelloPrefix
0000209C	01	Type	
		00	N_UNDF
		01	N_EXT
0000209D	00	Section Index	NO_SECT
0000209E	0200	Description	
		0	REFERENCE_FLAG_UNDEFINED_NON_LAZY
		Library Ordinal	2 (libsay.dylib)
000020A0	0000000000000000	Value	0
000020A8	0000002A	String Table Index	_say
000020AC	01	Type	
		00	N_UNDF
		01	N_EXT
000020AD	00	Section Index	NO_SECT
000020AE	0200	Description	
		0	REFERENCE_FLAG_UNDEFINED_NON_LAZY
		Library Ordinal	2 (libsay.dylib)
000020B0	0000000000000000	Value	0
000020B8	0000002F	String Table Index	dyld_stub_binder
000020BC	01	Type	
		00	N_UNDF
		01	N_EXT
000020BD	00	Section Index	NO_SECT
000020BE	0100	Description	
		0	REFERENCE_FLAG_UNDEFINED_NON_LAZY
		Library Ordinal	1 (libSystem.B.dylib)
		0100	N_SYMBOL_RESOLVER
000020C0	0000000000000000	Value	0

可以看到，symbol table 中有三个未绑定的外部符号：`_kHelloPrefix`、`_say`、`dyld_stub_binder`；本文接下来对 Mach-O 文件结构的分析将围绕这 3 个符号进行展开。

结构分析

先将 Mach-O 中与动态链接相关的结构给罗列出来：

- Section

    - __TEXT __stubs
    - __TEXT __stub_helper
    - __DATA __nl_symbol_ptr
    - __DATA __got
    - __DATA __la_symbol_ptr
- Load Command
    - LC_LOAD_DYLIB
    - LC_SYMTAB
    - LC_DYSYMTAB
- Symbol Table
- Indirect Symbol Table
- Dynamic Loader Info
    - Binding Info
    - Lazy Binding Info

涉及若干个 sections、load commands，以及 indirect symbol table、dynamic loader info 等。其中`LC_LOAD_DYLIB`这个命令上文已经提到，它描述了镜像依赖的 dylibs。`LC_SYMTAB`定义的符号表（symbol table）是镜像所用到的符号（包括内部符号和外部符号）的集合，Mach-O 与静态链接对该命令和符号表有详细描述，本文不再赘述。

Indirect Symbol Table

每一个可执行的镜像文件，都有一个 symbol table，由`LC_SYMTAB`命令定义，包含了镜像所用到的所有符号信息。那么 indirect symbol table 是一个什么东西呢？本质上，indirect symbol table 是 index 数组，即每个条目的内

容是一个 index 值，该 index 值（从 0 开始）指向到 symbol table 中的条目。Indirect symbol table 由`LC_DYSYMTAB`定义，后者的参数类型是一个`dysymtab_command`结构体，详见dysymtab_command，该结构体内容非常丰富，目前我们只需要关注`indirectsymoff`和`nindirectsyms`这两个字段：

```
struct dysymtab_command {
    uint32_t cmd;
    uint32_t cmdsize;



    uint32_t indirectsymoff;
    uint32_t nindirectsyms;


};
```

`indirectsymoff`和`nindirectsyms`这两个字段定义了 indirect symbol table 的位置信息，每一个条目是一个 4 bytes 的 index 值。

Indirect symbol table 的结构还是蛮容易理解的，但其存在的意义是啥？先别急，后面会讲到，总之它是为`__stubs`、`__got`等 section 服务的。

上文 main.out 的 indirect symbol table 可使用`otool -I main.out`查看，一共包括 5 个条目：

```
Indirect symbols for (__TEXT,__stubs) 1 entries
  address            index
  0x0000000000001fbe     3

Indirect symbols for (__DATA,__nl_symbol_ptr) 2 entries
  address            index
  0x0000000000002000     4

0x0000000000002008 ABSOLUTE

Indirect symbols for (__DATA,__got) 1 entries
  address            index
  0x0000000000002010     2

Indirect symbols for (__DATA,__la_symbol_ptr) 1 entries
  address            index
  0x0000000000002018     3
```

> 第三个条目的 index 值有些奇怪，还没搞清楚...

__text 里的外部符号

回到上文提到的 main.out，使用`otool -tv main.out`命令查看 main.out 代码段的反汇编内容下：

```
_main:
0000000000001f90  pushq %rbp
0000000000001f91  movq  %rsp, %rbp
0000000000001f94  subq  $0x10, %rsp
0000000000001f98  leaq  0x3f(%rip), %rsi
0000000000001f9f  movq  0x6a(%rip), %rax
0000000000001fa6  movl  $0x0, -0x4(%rbp)
```

```
0000000000001fad  movq  (%rax), %rdi
0000000000001fb0  callq 0x1fbe
0000000000001fb5  xorl  %eax, %eax
0000000000001fb7  addq  $0x10, %rsp
0000000000001fbb  popq  %rbp
0000000000001fbc  retq
```

上述是 main 函数的反汇编代码，注意第 6 行和第 9 行，这两行的指令分别引用了`_kHelloPrefix`和`_say`符号；这两个符号未绑定，如果是静态链接，这俩处的地址值是 0；但此处是动态链接，符号目标地址值分别指向的是偏移 0x6a 和 0x09，本文所在环境，采用的 PC 近址寻址，所以`_kHelloPrefix`和`_say`的目标地址分别是：

```
_kHelloPrefix 的目标虚拟地址 = 0x1fa6（第 7 行指令的虚拟地址） +
_say 的目标虚拟地址 = 0x1fb5（第 10 行指令虚拟地址） + 0x09 = 0x
```

`0x2010`和`0x1fbe`分别对应 main.out 中的哪个结构呢？答案是 section(__DATA __got) 和 section(__TEXT __stubs)，使用`otool -s`命令可以查看这两个 section 的地址和内容：

```
$ otool main.out -s __DATA __got
main.out:
Contents of (__DATA,__got) section
0000000000002010  00 00 00 00 00 00 00 00

$ otool main.out -s __TEXT __stubs
main.out:
Contents of (__TEXT,__stubs) section
0000000000001fbe  ff 25 54 00 00 00
```

Mach-O 的代码段对 dylib 外部符号的引用地址，要么指向到__got，要么指向到__stubs。什么时候指向到前者，什么时候指向到后者呢?

站在逻辑的角度，符号有两种：数据型和函数型；前者的值指向到全局变量/常量，后者的值指向到函数。在动态链接的概念里，对这两种符号的绑定称为：non-lazy binding、lazy binding。对于前者，在程序运行前（加载时）就会被绑定；对于后者，在符号被第一次使用时（运行时）绑定。

section(__DATA __got)

对于程序段__text里的代码，对数据型符号的引用，指向到了__got；可以把__got看作是一个表，每个条目是一个地址值。

在符号绑定（binding）前，__got里所有条目的内容都是0，当镜像被加载时，dyld 会对__got每个条目所对应的符号进行重定位，将其真正的地址填入，作为条目的内容。换句话说，__got各个条目的具体值，在加载期会被 dyld 重写，这也是为啥这个 section 被分配在 __DATA segment 的原因。

问题来了，dyld 是如何知道__got中各个条目对应的符号信息（譬如符号名字、目标库等）呢? Mach-O 简单分析 已经提到过，每个 segment 由LC_SEGMENT命令定义，该命令后的参数描述了 segment 包含的 section 信息，是谓 section header，对应结构体（x86_64架构）

是section_64：

```
struct section_64 {
    char      sectname[16];
    char      segname[16];

    uint32_t  reserved1;
    uint32_t  reserved2;
    uint32_t  reserved3;
};
```

对于__got、__stubs、__nl_symbol_ptr、__la_symbol_ptr这几个 section，其reserved1描述了该 list 中条目在 indirect symbol table 中的偏移量。

举个栗子，本文的 main.out 中的__got的 section header 的reserved1字段值为 3，它有一个条目，那么该条目对应的符号在 symbol table 中的 index，等于 indirect symbol table 中第 3+1 个条目的值；有点绕口，用伪代码表示，main.out 的__got的第一个条目对应的符号是：

```
__got[0]->symbol = symbolTable[indirectSymbolTable[__got.se
```

算是把__got讲清楚了，总之一句话，__got为 dyld 服务，用来存放 non-lazy 符号的最终地址值。

现在该说说__stub。

section(__TEXT __stubs)

对于程序段__text里的代码，对函数型符号的引用，指向到了__stubs。和__got一样，__stubs也是一个表，每个表项是一小段jmp代码，称为「符号桩」。和__got不同的是，__stubs存在于 __TEXT segment 中，所以其中的条目内容是不可更改的。

查看__stubs里的反汇编内容：

```
$ otool -v main.out -s __TEXT __stubs
main.out:
Contents of (__TEXT,__stubs) section
0000000000001fbe  jmpq  *0x54(%rip)
```

来看看jmp指令跳到哪里去，这里使用的间接寻址，真正的地址值存在 0x0000000000002018 中。

> 0x0000000000000x2018 = 0x0000000000001fbe + 0x54

0x2018是哪个部分？答案是 section(__DATA __la_symbol_ptr)...

section(__DATA __la_symbol_ptr)

使用otool -s查看__la_symbol_ptr的内容：

```
$ otool main.out -s __DATA __la_symbol_ptr
main.out:
Contents of (__DATA,__la_symbol_ptr) section
0000000000002018  d4 1f 00 00 00 00 00 00
```

__la_symbol_ptr的内容是 0x1FD4（小端），所以__stubs第一个 stub 的 jump 目标地址是 0x1FD4。该地址坐落于 section(__TEXT __stub_helper)。

section(__TEXT __stub_helper)

看看__stub_helper里的内容：

```
$ otool -v main.out -s __TEXT __stub_helper
main.out:
Contents of (__TEXT,__stub_helper) section
0000000000001fc4  leaq  0x3d(%rip), %r11
0000000000001fcb  pushq %r11
0000000000001fcd  jmpq  *0x2d(%rip)
0000000000001fd3  nop
0000000000001fd4  pushq $0x0
0000000000001fd9  jmp 0x1fc4
```

__stubs第一个 stub 的 jump 目标地址在第 8 行；这几条汇编代码比较简单，可以看出，代码最终会跳到第 6 行；之后该何处何从？

不难计算，第 6 行跳转目标地址是 0x2000 (0x1fd3 + 0x2d)存储的内容，0x2000 在哪里呢？0x2000 坐落于 section(__DATA __nl_symbol_ptr)。

section(__DATA __nl_symbol_ptr)

按惯例，查看__nl_symbol_ptr里的内容：

```
$ otool main.out -s __DATA __nl_symbol_ptr
main.out:
Contents of (__DATA,__nl_symbol_ptr) section
0000000000002000  00 00 00 00 00 00 00 00
0000000000002008  00 00 00 00 00 00 00 00
```

啥是__nl_symbol_ptr？和__got类似，__nl_symbol_ptr也是用来存储 non-lazy symbol 绑定后的地址。只是__got是为__text代码段中的符号服务的，而__nl_symbol_ptr不是。

如上，__nl_symbol_ptr的第一个条目的符号是dyld_stub_binder。

> dyld_stub_binder是一个函数，为啥它被当做一个 non-lazy symbol 处理，这是因为它是所有 lazy binding 的基础，所以有些特殊。

dyld_stub_binder

dyld_stub_binder也是一个函数，定义于 dyld_stub_binder.S，由 dyld 提供。

Lazy binding symbol 的绑定工作正是由 dyld_stub_binder 触发，通过调用dyld::fastBindLazySymbol来完成。

Lazy Binding 分析

上文结合 main.out 实例，对 Mach-O 与动态链接相关的结构做了比较全面的分析。Non-lazy binding 比较容易理解，这里稍微对如上内容进行整合，整体对 lazy binding 基本逻辑进行概述。

对于__text代码段里需要被 lazy binding 的符号引用（如上文 main.out 里的_say），访问它时总会跳转到 stub 中，该 stub 的本质是一个 jmp 指令，该 stub 的跳转目标地址坐落于__la_symbol_ptr。

首次访问_say时：

1. _say对应的__la_symbol_ptr条目内容指向到__stub_helper
2. __stub_helper里的代码逻辑，通过各种辗转最终调用dyld_stub_binder函数
3. dyld_stub_binder函数通过调用 dyld 内部的函数找到_say符号的真实地址
4. dyld_stub_binder将地址写入__la_symbol_ptr条目
5. dyld_stub_binder跳转到_say符号的真实地址

之后再次访问_say时，stub 里的 jmp 指令直接跳转符号的真实地址，因为该地址已经被写到__la_symbol_ptr条目中。

写在后面

分析到这里，有种神清气爽的感觉，是那种费了老半天劲

儿爬到高处欣赏风景的感觉。理解这些与动态链接有关的内容后，再去分析其他比较底层的东西，或许就有眉目了。

此处做一个小结：

1. 本文并未对`Dynamic Loader Info`内容进行展开，其中涉及 Bind 相关的各种 opcode。原因有俩，其一，笔者对它们的了解并不深刻；其二，至少目前来看，了解它们的意义不大。
2. 笔者看了许多其他博客，发现很少有谈到`__got`这个 section 的，本文所提到的`__got`的作用，在他们的博客中，都被冠到`__nl_symbol_ptr`中；猜测的原因是编译器的处理姿势不一样了；不过这也不是很重要了，真正重要的是会自己分析，毕竟无论是`__got`，还是`__nl_symbol_ptr`，只是一个名字而已。

关于 Mach-O 的动态链接，还有很多值得分析的问题，之后的博客或许会有补充，包括但不限于：

- dyld 是如何工作的?
- 如何理解 fishhook?
- Objective-C 与动态链接碰撞出了什么样的火花?

更多阅读

- 《深入理解计算机系统》
- 《程序员的自我修养》
- Inside a Hello World executable on OS X
- 深入剖析Macho(1)