分析 dyld 的启动过程

之前的博文Mach-O 与动态链接基于 Mach-O 文件结构，分析了 Mach-O 的动态链接逻辑。当一个可执行 Mach-O 文件被执行时，内核在完成进程创建等一系列初始化后，会把后续的链接以及镜像加载工作交给 dyld 来完成。

本文对 dyld 的启动过程稍作分析，在分析的过程中会回答如下两个问题：

- 程序是如何加载的，或者说，程序的入口代码（譬如`main`函数）是如何被调用的?
- dyld 逻辑的启动过程是怎样的?

本文所分析对象包括：

- dyld Mach-O 文件（在`/usr/lib/dyld`中，本文所分析的版本是 640.2，对应源码还没来得及开源）
- dyld-635.2 源码
- xnu-4903.221.2 源码

dyld 本身也是一个 dylib，但它有一些特殊性。对于普通 dylib 来说，它的重定位工作由 dyld 来完成；它也可以依赖与其他 dylib，其中被依赖的 dylib 由 dyld 负责链接和装载。对于 dyld 来说，它是如何被加载的呢？它是否可以依赖于其他的 dylib?

首先，dyld 本身不可以依赖于其他任何 dylib，使用`xcrun dyldinfo -dylibs`查看可以证实这一点：

```
$ xcrun dyldinfo -dylibs dyld
for arch x86_64:
attributes     dependent dylibs
for arch i386:
attributes     dependent dylibs
```

如上结果（也可以使用 MachOView 工具查看）显示，dyld 没有依赖任何其他 dylib；不过这一点显然是可以人为控制的，在编写动态链接器时保证不使用任何其他库资源（包括系统库）就可以了。

那么对于另一个问题：dyld 是如何被加载的呢？或者说，它的代码是如何被执行的呢?

dyld 的代码逻辑开始于dyldStartup.s，它用汇编实现了名为`__dyld_start`的函数，主要做两件事情：

- 调用`dyldbootstrap::start`方法，后者返回可执行文件的入口，暂称为`_main`
- 填入参数，调用`_main`

> `dyldbootstrap::start`也定义于 dyld 中，该函数对应的符号是`__ZN13dyldbootstrap5startEPK12macho_headeriPPKclS2_Pm`，定义于dyld.order。

所以说，可执行文件的入口，是由 dyld 负责找到并调用的，如何找到？这个问题比较简单，对于依赖于 dyld 的可执行文件，其逻辑代码的起点（entry point）可以从`LC_MAIN`这个 load command 里解析得到，该指令的参数

记录了程序入口指令相对于镜像文件的 file offset；dyld 完成 rebase、binding 等工作后，会去调用该地址的函数。

> 貌似从 10.7 开始，Mac OS X (macOS) 里的可执行文件都得依赖于 dyld。

另一个关键问题是：内核是如何找到`__dyld_start`函数的呢?

关于这个问题，参考StackOverflow: What is required for a Mach-O executable to load?，尝试从`LC_UNIXTHREAD`这个 load command 里挖掘答案。

dyld 本身逻辑的加载，依赖于内核调用，内核在对 dyld 镜像进行解析时，从`LC_UNIXTHREAD`这个 load command 中提取 dyld 的 entry point。

使用`otool -l`查看`LC_UNIXTHREAD`的结构：

```
Load command 9
        cmd LC_UNIXTHREAD
    cmdsize 184
     flavor x86_THREAD_STATE64
      count x86_THREAD_STATE64_COUNT
   rax    0x0000000000000000 rbx    0x0000000000000000 rcx
   rdx    0x0000000000000000 rdi    0x0000000000000000 rsi
   rbp    0x0000000000000000 rsp    0x0000000000000000 r8
    r9    0x0000000000000000 r10    0x0000000000000000 r11
   r12    0x0000000000000000 r13    0x0000000000000000 r14
   r15    0x0000000000000000 rip    0x0000000000001000 rfla
   cs     0x0000000000000000 fs     0x0000000000000000 gs
```

它包含了三个有效信息：flavor、count、thread_state。flavor 描述的是进程类型（x86_64 架构对应的是`x86_THREAD_STAT64`，i386 架构对应的是`x86_THREAD_STAT32`）；count 描述 thread_state 的长度；至于`thread_state`，描述的是进程状态，「进程」这个概念的完成需要硬件和软件通力协作，不同不同架构的`thread_state`不同，对于`x86_THREAD_STAT64`，`thread_state`结构定义于/osfmk/mach/i386/_structs.h，如下：：

```
#define _STRUCT_X86_THREAD_STATE64 struct x86_thread_state6
_STRUCT_X86_THREAD_STATE64
{
  __uint64_t  rax;
  __uint64_t  rbx;
  __uint64_t  rcx;
  __uint64_t  rdx;
  __uint64_t  rdi;
  __uint64_t  rsi;
  __uint64_t  rbp;
  __uint64_t  rsp;
  __uint64_t  r8;
  __uint64_t  r9;
  __uint64_t  r10;
  __uint64_t  r11;
  __uint64_t  r12;
  __uint64_t  r13;
  __uint64_t  r14;
  __uint64_t  r15;
  __uint64_t  rip;
```

```
  __uint64_t  rflags;
  __uint64_t cs;
  __uint64_t  fs;
  __uint64_t  gs;
};
```

对于`x86_THREAD_STAT64`类型的 thread_state，rip 存储了进程的 entry point（相对于镜像的 file offset）；接下来的 xnu 源码分析是为了证实这一点。

从/bsd/kern/mach_loader.c里的`parse_machfile`函数开始看，顾名思义，`parse_machfile`用于解析 Mach-O 文件，摘录其解析 load commands 的`LC_UNIXTHREAD`分支：

```
static load_return_t parse_machfile(
  struct vnode        *vp,
  vm_map_t            map,
  thread_t            thread,
  struct mach_header  *header,
  off_t               file_offset,
  off_t               macho_size,
  int                 depth,
  int64_t             aslr_offset,
  int64_t             dyld_aslr_offset,
  load_result_t       *result,
  load_result_t       *binresult,
  struct image_params *imgp
)
{

  case LC_UNIXTHREAD:
    if (pass != 1)
```

```
      break;
    ret = load_unixthread(
                          (struct thread_command *) lcp,
                          thread,
                          slide,
                          result);

}
```

跟着load_unixthread往下看，关键代码如下：

```
static load_return_t load_unixthread(
  struct thread_command *tcp,
  thread_t              thread,
  int64_t               slide,
  load_result_t         *result
)
{

  ret = load_threadentry(thread,
                          (uint32_t *)(((vm_offset_t)tcp) +
                          sizeof(struct thread_command)),
                          tcp->cmdsize - sizeof(struct thread
                          &addr);
  if (ret != LOAD_SUCCESS)
    return(ret);

  if (result->using_lcmain || result->entry_point != MACH_V

    return (LOAD_FAILURE);
  }

  result->entry_point = addr;
```

```
  result->entry_point += slide;

}
```

提取 entry point 的重任被辗转到`load_threadentry`，后者逻辑也挺简单，所做的事情不过是把锅甩到`thread_entrypoint`，不同平台的`thread_entrypoint`定义不同，对于 x86 架构，该函数定义于 [xnu/osfmk/i386/bsd_i386.c](), 它的逻辑蛮简单：

```
kern_return_t thread_entrypoint(
    __unused thread_t       thread,
    int                     flavor,
    thread_state_t          tstate,
    __unused unsigned int   count,
    mach_vm_offset_t        *entry_point
)
{
  if (*entry_point == 0)
    *entry_point = VM_MIN_ADDRESS;

  switch (flavor) {
  case x86_THREAD_STATE32:
    {
      x86_thread_state32_t *state25;

      state25 = (i386_thread_state_t *) tstate;
      *entry_point = state25->eip ? state25->eip: VM_MIN_AD
      break;
    }

  case x86_THREAD_STATE64:
```

```
    {
      x86_thread_state64_t *state25;

      state25 = (x86_thread_state64_t *) tstate;
      *entry_point = state25->rip ? state25->rip: VM_MIN_AD
      break;
    }
  }
  return (KERN_SUCCESS);
}
```

分析到这里基本实锤了，对于 x86_64 架构，`LC_UNIXTHREAD` 命令里的`thread_state->rip`指定了 dyld 的 entry point。

OK，是时候对 dyld 以及程序的启动做个总结了（针对 x86_64 架构的 Mach-O 可执行文件）：

- 对于依赖于 dyld 的可执行文件，进程的 entry point 是 dyld 的`__dyld_start`函数
- 内核通过解析`LC_UNIXTHREAD`，从`thread_state->rip`里获得 dyld 的入口（即`__dyld_start`）
- 镜像的入口由`dyldbootstrap::start`函数发现，返回到`__dyld_start`，被后者调用
- 镜像的入口可通过解析`LC_MAIN`得到

写在后面

本文对 xnu 和 dyld 源码的分析完全属于盲人摸象，分析思路是先提出问题，然后分析源码，尝试给出自己的回

答。除了理解上可能存在偏差外，还有如下问题没有搞清楚：

- 个人感觉，`LC_UNIXTHREAD`是 dyld 的专有命令，但没有找到权威的说明
- 现在还能折腾出不依赖 dyld 的程序可执行文件吗？尝试弄了一把，但没有成功，貌似依赖 dyld 是 macOS 对应用程序的强制行为

更多阅读：

- StackOverflow: What is required for a Mach-O executable to load?
- iOS 程序 main 函数之前发生了什么
- dyld: Dynamic Linking On OS X
- Dyld系列之一： _dyld_start之前