Mach-O 简单分析

最近尝试以《深入理解计算机系统》相关章节为纲，梳理链接相关知识点；对于 Apple 生态，一切都得从 Mach-O 谈起。

时至今日，介绍 Mach-O 的资料已经很多了，即便如此，还是决定用一篇水文整理一下 Mach-O 结构相关信息，出于如下几个理由：

- 相较于 ELF，Mach-O 的官方资料（权威资料）少得多，这给学习带来了些许困扰
- 讲 Mach-O 文件格式的博客网文虽多，但似乎没有发现在文件结构上讲得非常清楚的，至少大多没有突出重点
- 在记录整理过程中脑袋往往能蹦出许多问题，「提问 -> 回答」或许能帮助理解得更加深刻

如上所述，Mach-O 的官方资料比较少，但是相关源码是开放的：xnu/mach-o。本文主要目的是梳理 Mach-O 文件格式，其他细节内容，譬如典型的 Load Commands、section等，本文不打算展开。

建议

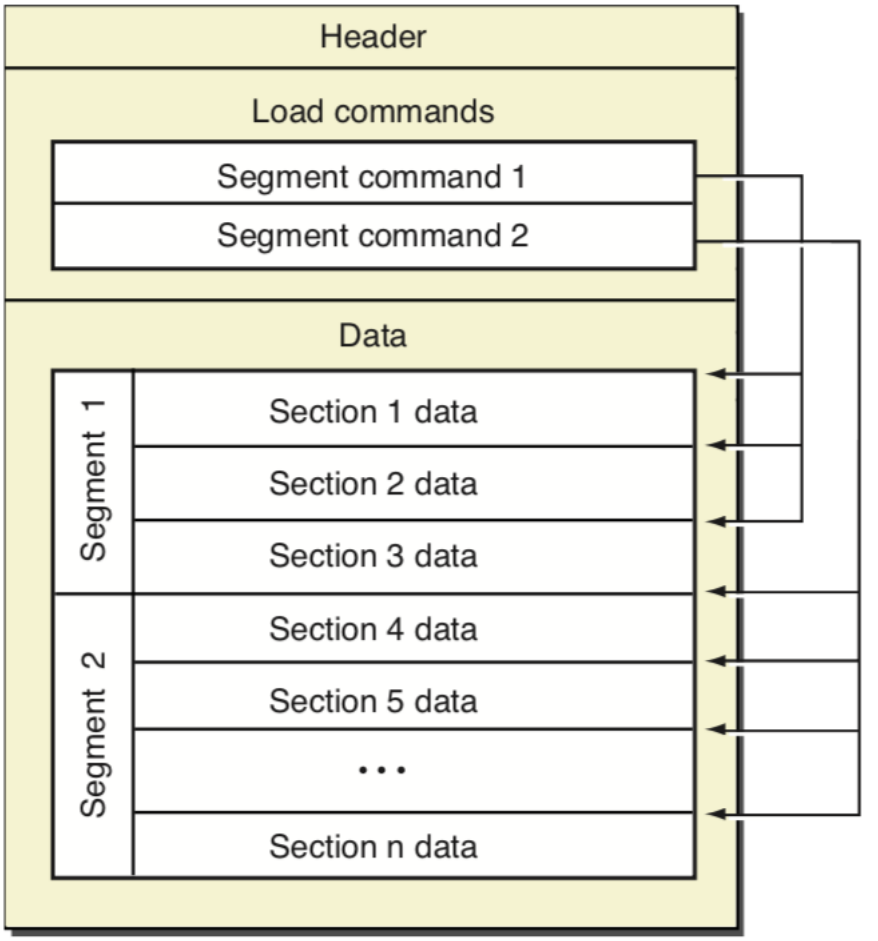
要想彻底搞清楚 Mach-O，了解虚拟内存是必须的，但虚拟内存是一个较大的概念，涉及相关知识时，本文不展开。

- 结构分析

    - Header 的结构
    - Load Commands 的结构
    - Data 的结构
- Mach-O 的结构
- 写在后面

结构分析

关于 Mach-O 的文件格式，在网上常常看到如下这张图，出自官方文档《OS X ABI Mach-O File Format Reference》：



说明

《OS X ABI Mach-O File Format Reference》已无法在 Apple 官网里找到了，但这里有一份 Copy。

通过这张图，可以看到，从布局上，Mach-O 文件分为三个部分：Header、Load Commands、Data。但这张图过于简略，信息不完善，可能会让人困惑。刚看到这张图的时候，笔者的理解是：

- Data 由 segment 组成，segment 由 section 组成
- Header 里有描述 segment 的数据（包括 segment 的数量、各个 segment 的 offset、size，等等）

但事实完全不是这样的，看了一些资料，得到的基本结论是：理解 segment 和 section 的结构并不是那么直观的事情；不过本文的目标就是把它们给理清楚。

在此之前，还是先简单分析 Header 和 Load Commands 的结构吧！

Header 的结构

mach-o/loader.h的`struct mach_header`清晰地定义了 Header 的结构，如下：

```
struct mach_header_64 {
    uint32_t magic;
    cpu_type_t cputype;
    cpu_subtype_t cpusubtype;
    uint32_t filetype;
    uint32_t ncmds;
    uint32_t sizeofcmds;
    uint32_t flags;
    uint32_t reserved;
};
```

需要说明的是，32 bit 和 64 bit 分别对应了不同的结构，但大同小异，本文所述内容均以 64 bit 为 base。可见，在 Mach-O 中，Header 的结构是固定的：

- file offset 固定为 0
- 长度为 32 bytes（对于 64 bit 架构）

借用《深入解析 Mac OS X & iOS 操作系统》的一张图简单描述上述 7 个有效字段：

值得拧出来讲的两个字段是 filetype 和 flags。

先说 filetype，描述了二进制文件的类型，包括了十来个有效值，常打交道的包括：

```
#define MH_OBJECT       0x1
#define MH_EXECUTE      0x2
#define MH_DYLIB        0x6
#define MH_DYLINKER     0x7
```

flags 是杂项，通常它包含的信息用于为动态链接器服务，告诉后者如何工作。

Load Commands 的结构

Load Commands 可以被看作是一个 command 列表，紧贴着 Header，所以它的 file offset 是固定的：0x20。一共有哪些 load commands 呢？Load commands 由内核定义，不同版本的 command 数量不同，本文所参考的内核，一共定义了 50+ load commands，它们的 type 是以 LC_为前缀常量，譬如 LC_SEGMENT、LC_SYMTAB 等。

每个 command 都有独立的结构，但所有 command 结构的前两个字段是固定的：

```
struct load_command {
    uint32_t cmd;
    uint32_t cmdsize;
};
```

第一个字段指定了类型，第二个字段确保它能被正确解析。

本文本不想展开描述 load commands，但是若想理解 segment 和 section，不得不先了解 LC_SEGMENT_64，因为它和 segment、section 有关；该命令由内核解析，内核根据该命令对 Mach-O 文件进行最初的结构化，命令格式如下：

```
struct segment_command_64 {
```

```
    uint32_t   cmd;
    uint32_t   cmdsize;
    char       segname[16];
    uint64_t   vmaddr;
    uint64_t   vmsize;
    uint64_t   fileoff;
    uint64_t   filesize;
    vm_prot_t  maxprot;
    vm_prot_t  initprot;
    uint32_t   nsects;
    uint32_t   flags;
};
```

从这个结构体我们能看出什么？它描述了文件映射的两大问题：从哪里来（`fileoff`、`filesize`）、到哪里去（`vmaddr`、`vmsize`）；它还告诉了内核该区域的名字（`segname`，即 segment name），以及该区域包含了几个 section（`nsects`），以及该区域的保护级别（`initprot`、`maxprot`）。

补充说明

- 每一个 segment 的 VP (Virtual Page) 都根据 `initprot` 进行初始化，`initprot` 指定了如何通过读/写/执行位初始化页面的保护级别；segment 的保护设置可以动态改变，但是不能超过 maxprot 中指定的值（在 iOS 中，+x 和+w 是互斥的）；`initprot`、`maxprot` 的值均用八进制表示（`4=r, 2=w, 1=x`）
- flags 是杂项标志位

- `vmsize` 并不等于 `filesize`，对于 4KB 大小的 VP，vmsize 是 4K 的倍数；换句话说，`vmsize` 一般大于 segment 的实际大小

对于 segment 而言，有了这些信息，其结构其实就足够清晰了，似乎不再需要别的信息来描述；事实也是这样，xnu 内核确实找不到描述 segment 的结构体；对它的描述正是在 LC_SEGMENT_64 里完成的。

不过，似乎还不够啊，虽然知道了 segment 对应的区域和 section 的数量，但是如何知道其中各个 sections 的具体位置和 size 呢?

别急，对于 LC_SEGMENT_64 而言，如果其`nsects`字段大于 0，其命令后面还会紧接着挂载`nsects`个描述 section 的信息，这些信息是`section_64`的列表，`section_64`结构体定义如下：

```
struct section_64 {
    char      sectname[16];
    char      segname[16];
    uint64_t  addr;
    uint64_t  size;
    uint32_t  offset;
    uint32_t  align;
    uint32_t  reloff;
    uint32_t  nreloc;
    uint32_t  flags;
    uint32_t  reserved1;
    uint32_t  reserved2;
    uint32_t  reserved3;
```

```
};
```

结构体`section_64`可以看做 section header，它描述了对应 section 的具体位置，以及要被映射的目标虚拟地址。

回头再看`segment_command_64`的`cmdsize`字段，它的数值并非是`segment_command_64`的 size 大小，还包括了紧接在 command 后面的所有`section_64`结构体的大小。

举个例子，如果 segment 含有 5 个 section，那么对应的 segment_command_64 的 cmdsize 值为：

```
72（segment_command_64本身大小） + 5 * 80（section_64的大小） =
```

对 LC_SEGMENT_64 的分析到此结束，此时应该基本搞清楚了 segment 和 section，得到的事实是：Mach-O 本没有 segment，有了 LC_SEGMENT_64，于是有了 segment。

Data 的结构

有了上文对 LC_SEGMENT_64 的分析，基本上搞清楚了 segment 和 section；再理解 Data 的结构就不难了。

和 Header、Load Commands 不同，Mach-O 对 Data 区域没有任何结构上的定义。它里面盛装的字节本来没有意义，有了 LC_SEGMENT_64 以及其他的 load commands，一切才开始有了意义。

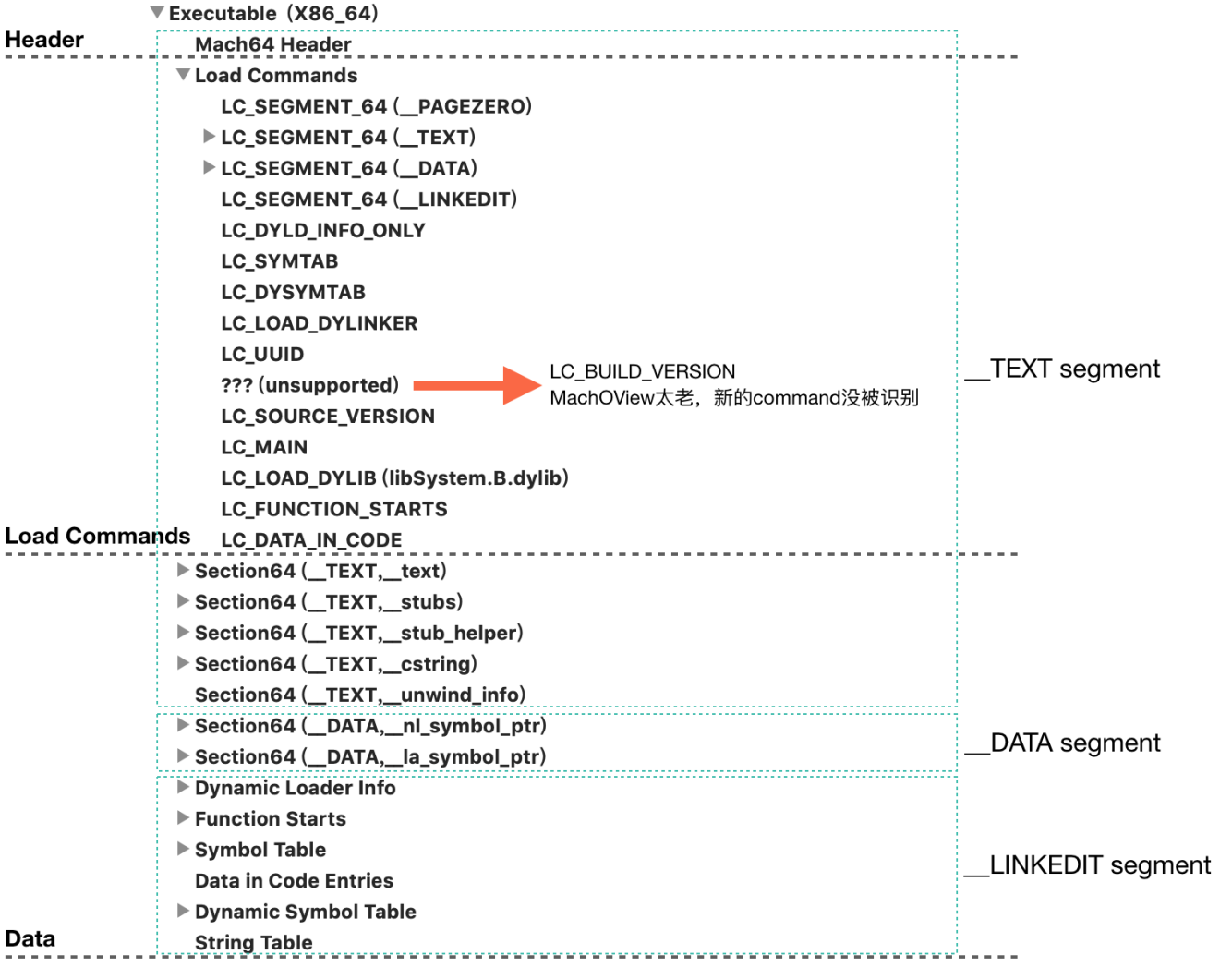
Mach-O 的结构

这一部分结合上面的内容，通过一个具体的 case，综述一下 Mach-O 的结构。写一个简单的 C 文件如下：

```
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

执行`gcc main.c`，得到可执行文件 a.out，使用 MachOView 工具查看，得到如下结构：



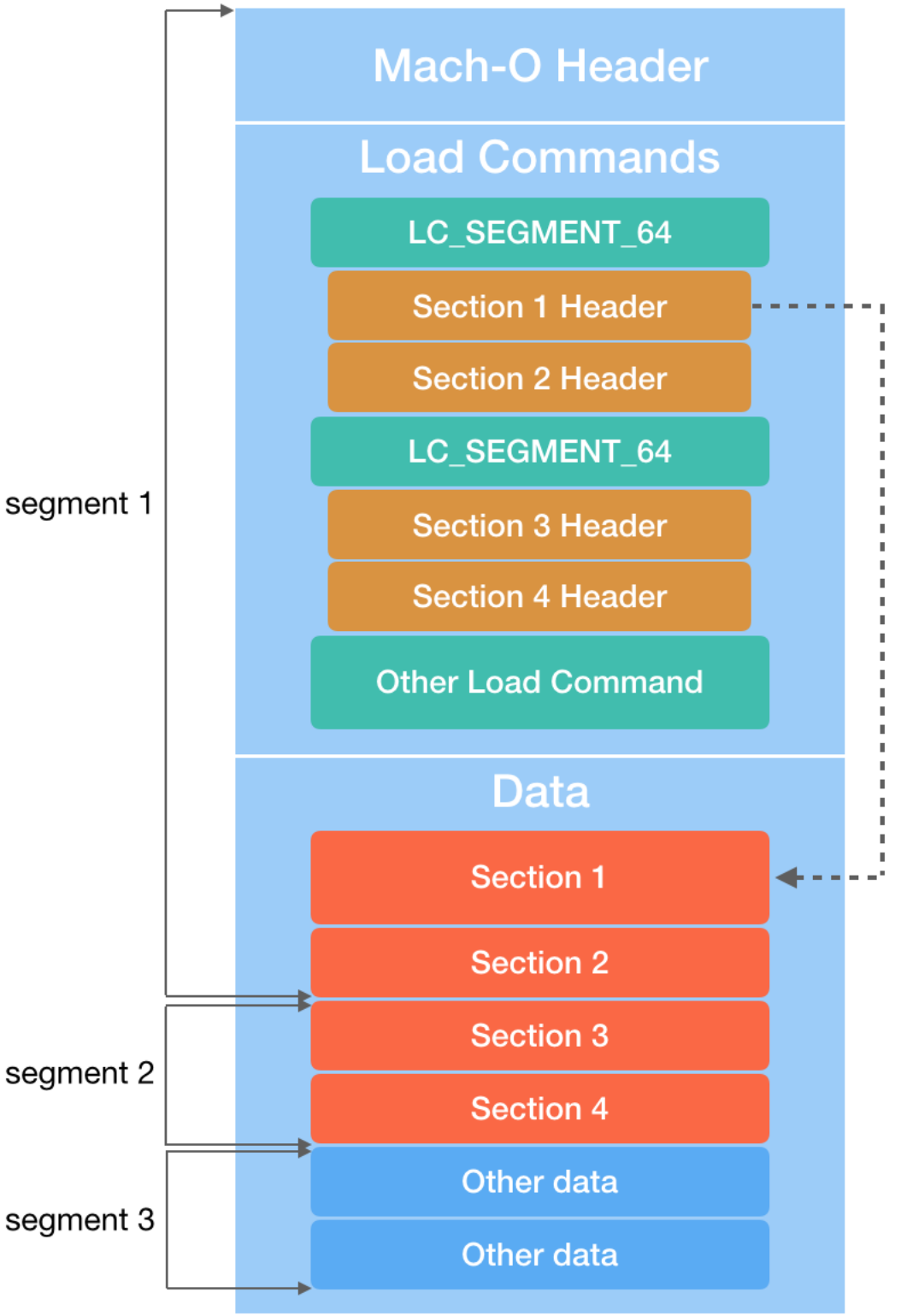
可以得到的信息：

- 一共包括三个 segment：__TEXT、__DATA、__LINKEDIT
- segment 的块范围并非一定在 Data 区内（譬如 __TEXT segment）
- 并非每一个 segment 都由 section 组成（譬如 __LINKEDIT segment）

为啥 __TEXT 的地址范围从 0 开始而非从 _text 这个 section 开始呢？《OS X ABI Mach-O File Format Reference》是这么说的：

> The header and load commands are considered part of the first segment of the file for paging purposes. In an executable file, this generally means that the headers and load commands live at the start of the __TEXT segment because that is the first segment that contains data.

综上，笔者认为一个典型的 Mach-O 结构图的更清晰描述应该是这个样子：



写在后面

刚开始接触 Mach-O 时，我在学习方向上有这么些误区：

- 情不自禁地想办法搜罗各种 load commands 信息，以为了解 Mach-O 的关键在于搞清楚这些 load commands
- 以为 load command 是用来被执行的，很纠结它们是

如何被执行的，它们的执行顺序是如何的，以及是谁在执行

待到对 Mach-O 的研究开始有些眉目时，回过头来看，我认为的正确的认识应该是：

- 各种 load commands 存在的意义是让整个 Mach-O 变得结构化起来
- 对 Mach-O 的分析，主要围绕 header 和 load commands 进行，它们的典型服务对象是内核/链接器；换句话说，有的 commands 是为内核服务的，有些是为链接器（无论是静态链接器，还是动态链接器）服务的
- 在进行下一步的研究分析之前，笼统搞清楚各个 load commands 的内涵，其意义并不大，在分析静态链接、动态链接等主题的具体问题时，再去分析相关的 load commands，才能理解得更深刻