

Mach-O文件的全称是Mach Object File Format,类似于Windows上的exe文件, Linux上的ELF文件,macOS/iOS系统中可执行文件、动态库、共享库格式都是Mach-O。

先说一个容易产生困惑的点, Mach-O和Mac没有什么关系。Mac是苹果电脑Macintosh的简称, 而Mach是一种操作系统微内核, 苹果公司的设备上操作系统内核使用的是Mach。在Mach内核中, 一种可执行文件格式是Mach-O。所以不要被Mach-O和Mac相似的名字迷惑了, 实际上两者的关系不大。

在介绍Mach-O文件格式之前, 首先了解一下通用二进制格式。

通用二进制格式

无论是PC还是手机, 发展至今都经过了不同的CPU架构。日常项目中也经常会用到和架构相关的问题, 如

```
invalid armv7.....
```

先看一下不同平台的CPU架构。

移动平台CPU架构

手机上的CPU架构主要有armv7,armv7s,arm64。具体如下:

1. 从iPhone5s及以后的所有机型, 其CPU架构都是arm64的。
2. iPhone5、iPhone5C, 其CPU架构都是armv7s。
3. iPhone4s以及之前的机型, 其CPU架构都是armv7的。

PC平台CPU架构

i386针对的是32位的CPU处理器。x86_64是针对x86架构的64位处理器。

通用二进制格式

通用二进制格式（Universal Binary），又称为胖二进制(Fat Binary)。通用二进制文件实际上就是将支持不同CPU架构的二进制文件打包成一个文件，系统在加载运行时，会根据通用二进制文件中提供的架构，选择和当前系统匹配的二进制文件。因此，很多人认为，将通用二进制文件称为胖二进制文件更为合适。

mac OS中自带了很多的通用二进制文件，使用file命令可以查看这些通用二进制文件的信息，比如使用file命令查看python的信息：

```
file /Users/.../Desktop/python
/Desktop/python: Mach-O universal binary with 2 architectures:
[x86_64:Mach-O 64-bit executable x86_64] [i386:Mach-O executable
i386]
/Desktop/python (for architecture x86_64): Mach-O 64-bit
executable x86_64
/Desktop/python (for architecture i386): Mach-O executable i386
```

可以看到，python通用二进制文件包含两种架构的Mach-O文件，分别是x86_64架构和i386架构。

看一下代码中是如何定义通用二进制文件的，在/usr/include/mach-o目录下有通用二进制相关的文件。在fat.h中可以看到通用二进制文件头部结构fat_header的定义（从文件名的角度来看，通用二进制文件称为胖二进制文件也更为合适）：

```
#define FAT_MAGIC    0xcafebabe
#define FAT_CIGAM    0xbebafeca /* NXSwapLong(FAT_MAGIC) */

struct fat_header {
    uint32_t    magic;        /* FAT_MAGIC or FAT_MAGIC_64 */
    uint32_t    nfat_arch;    /* number of structs that follow */
};
```

magic是一个固定的值，值是0xcafebabe或0xbebafeca，表示这是一个通用二进制文件；

nfat_arch表示的是该通用二进制文件包含多少个架构文件（也就是Mach-O文件）。

在fat_header之后紧跟着的是多个fat_arch结构体，fat_arch的定义如下：

```

struct fat_arch_64 {
    cpu_type_t  cputype;    /* cpu specifier (int) */
    cpu_subtype_t  cpusubtype; /* machine specifier (int) */
    uint64_t  offset;      /* file offset to this object file */
    uint64_t  size;        /* size of this object file */
    uint32_t  align;       /* alignment as a power of 2 */
    uint32_t  reserved;    /* reserved */
};

```

其中：cputype指定了cpu的类型

cpusubtype指定了cpu的子类型

offset指定了该架构数据相对于文件开头的偏移量

size指定了该架构数据的大小

align指定了数据的内存对齐边界，值必须是2的次方

reserved是保留字段，只有64位架构的有，32位架构的无此字段。

cputype的部分取值有：

```

#define CPU_TYPE_X86          ((cpu_type_t) 7)
#define CPU_TYPE_I386        CPU_TYPE_X86
#define CPU_TYPE_X86_64      (CPU_TYPE_X86 | CPU_ARCH_ABI64)
#define CPU_TYPE_ARM         ((cpu_type_t) 12)
#define CPU_TYPE_ARM64       (CPU_TYPE_ARM | CPU_ARCH_ABI64)
#define CPU_TYPE_POWERPC     ((cpu_type_t) 18)
#define CPU_TYPE_POWERPC64   (CPU_TYPE_POWERPC | CPU_ARCH_ABI64)

```

cpusubtype的部分取值有：

```

#define CPU_SUBTYPE_X86_ALL    ((cpu_subtype_t)3)
#define CPU_SUBTYPE_X86_64_ALL ((cpu_subtype_t)3)
#define CPU_SUBTYPE_X86_ARCH1 ((cpu_subtype_t)4)
#define CPU_SUBTYPE_X86_64_H   ((cpu_subtype_t)8)

```

使用MachOview软件看一下python的信息：

▼ Fat Binary	Offset	Data	Description	Value
Fat Header	00000000	BEBAFECA	Magic Number	FAT_CIGAM
▶ Executable (X86_64)	00000004	02000000	Number of Architecture	2
▶ Executable (X86)	00000008	07000001	CPU Type	CPU_TYPE_X86_64
	0000000C	03000080	CPU SubType	CPU_SUBTYPE_X86_64_ALL
	00000010	00100000	Offset	4096
	00000014	705A0000	Size	23152
	00000018	0C000000	Align	4096
	0000001C	07000000	CPU Type	CPU_TYPE_I386
	00000020	03000000	CPU SubType	CPU_SUBTYPE_I386_ALL
	00000024	00700000	Offset	28672
	00000028	205A0000	Size	23072
	0000002C	0C000000	Align	4096

可以清楚的看到，Fat Header里面的内容：

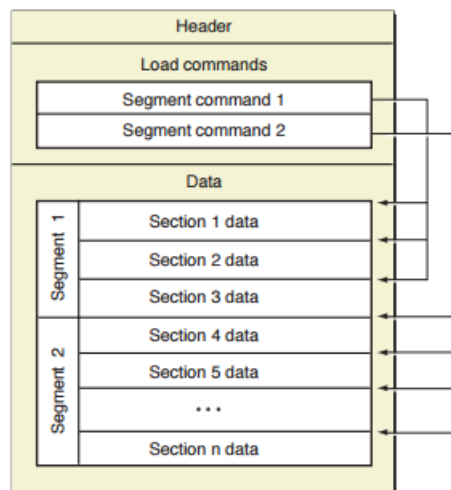
首先是fat_header，包含两种架构，后面跟着两个fat_arch结构。从图中可以看到，Fat Header之后就是两个Executable文件，也就是可执行文件，也就是Mach-O文件。

Mach-O文件

Mach-O是mac OS系统可执行文件的格式，平时用到的可执行文件，动态库，静态库，Dsym文件，都是Mach-O格式的文件。

看一下苹果官方文档中对Mach-O文件的介绍：

Figure 1 Mach-O file format basic structure

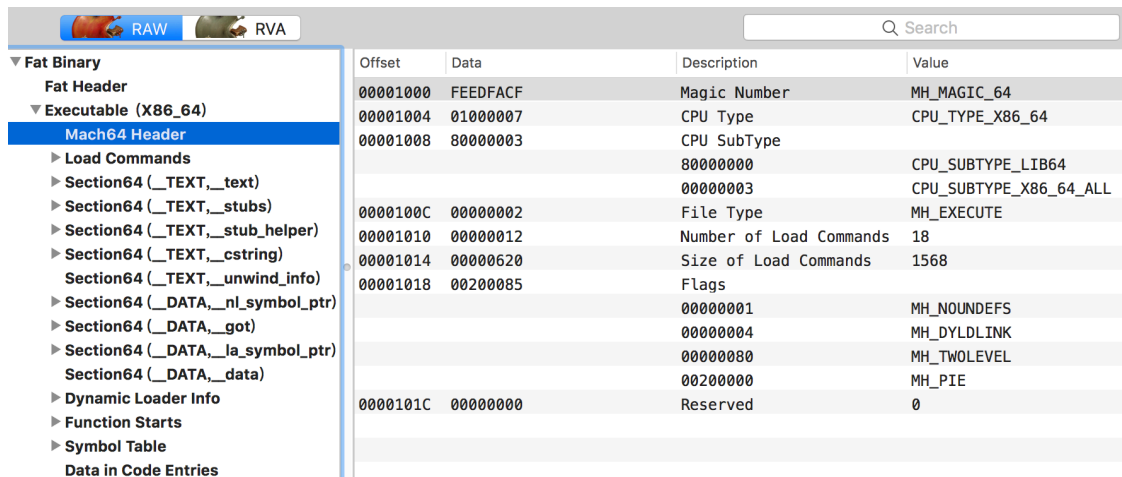


可以看到，一个Mach-O文件包含三部分：Header、Load commands、Data。实际上，除了上述三部分意外，还有Loader info链接信息及一些其他的数据。

从图中可以看到，Load commands中的数据>Data中的数据是有对应关系的。具体是如何对应的，下面会介绍。

Header

Mach-O头部，描述了Mach-O的cpu类型，文件类型以及加载命令大小、条数等信息。还是使用MachOview看一下python可执行文件中的Mach-O文件：



Offset	Data	Description	Value
00001000	FEEDFACF	Magic Number	MH_MAGIC_64
00001004	01000007	CPU Type	CPU_TYPE_X86_64
00001008	80000003	CPU SubType	CPU_SUBTYPE_LIB64
			CPU_SUBTYPE_X86_64_ALL
0000100C	00000002	File Type	MH_EXECUTE
00001010	00000012	Number of Load Commands	18
00001014	00000620	Size of Load Commands	1568
00001018	00200085	Flags	MH_NOUNDEFS
			MH_DYLDLINK
			MH_TWOLEVEL
			MH_PIE
0000101C	00000000	Reserved	0

通过MachOview可以得到Mach-O header中包含的信息，包含了Magic Number、Cpu type、Cpu subtype、file type等。看一下代码中对于Mach-O header的定义，相关的代码在mach-o/loader.h中：

```
struct mach_header_64 {
    uint32_t    magic;        /* mach magic number identifier */
    cpu_type_t  cputype;      /* cpu specifier */
    cpu_subtype_t  cpusubtype; /* machine specifier */
    uint32_t    filetype;     /* type of file */
    uint32_t    ncmds;        /* number of load commands */
    uint32_t    sizeofcmds;   /* the size of all the load commands */
    uint32_t    flags;        /* flags */
    uint32_t    reserved;     /* reserved */
};

/* Constant for the magic field of the mach_header_64 (64-bit architectures) */
#define MH_MAGIC_64 0xfeedfacf /* the 64-bit mach magic number */
#define MH_CIGAM_64 0xcffaedfe /* NXSwapInt(MH_MAGIC_64) */
```

magic字段是一个固定的值，为0xfeedfacf或者0xcffaedfe，表示的是这是一个Mach-O格式的文件。

cpu_type_t 和 cpu_subtype_t和上文中提到的一样，这里不再介绍。

filetype表示Mach-O文件的具体类型，它的部分取值如下：

```
#define MH_OBJECT    0x1    /* relocatable object file */
#define MH_EXECUTE   0x2    /* demand paged executable file */
#define MH_PRELOAD    0x5    /* preloaded executable file */
#define MH_DYLIB      0x6    /* dynamically bound shared library */
#define MH_DYLINKER   0x7    /* dynamic link editor */
#define MH_BUNDLE     0x8    /* dynamically bound bundle file */
#define MH_DSYM       0xa    /* companion file with only debug
sections */
```

这里python的Mach-O文件类型是MH_EXECUTE，如果我们查看的是Dsym文件，则文件类型是MH_DSYM。

ncmds 表示的是Mach-O文件中加载命令的数量。

sizeofcmd 表示的是Mach-O文件加载命令的大小。

flags 表示文件标志。

reserved 是保留字段，64位cpu架构特有。

Mach-O文件头除了提供一些格式信息外，还为加载命令提供信息。ncmds 和 sizeofcmd 在加载过程中会比较有用。

Load commands

Mach-O Header之后就是Load commands,也就是加载命令。加载命令的作用是，在Mach-O文件被加载到内存时，加载命令告诉内核加载器或者动态链接器如何调用。

可执行文件运行之后会生成一个进程，进程实际上就是特殊文件在内存中加载得到的结果，这种文件必须使用操作系统可以认知的格式，这样才对该文件引入依赖库，初始化运行环境以及顺利的执行创造了条件。在该过程中，就会用到Load command。

还是先使用MachOview看一下Mach-O文件中的加载命令：

Executable (X86_64)	Offset	Data	Description	Value
Mach64 Header				
▼ Load Commands				
LC_SEGMENT_64 (__PAGEZERO)	00001020	00000019	Command	LC_SEGMENT_64
	00001024	00000048	Command Size	72
► LC_SEGMENT_64 (__TEXT)	00001028	5F5F504147455A45524F00...	Segment Name	__PAGEZERO
► LC_SEGMENT_64 (__DATA)	00001038	0000000000000000	VM Address	0
LC_SEGMENT_64 (__LINKEDIT)	00001040	0000000100000000	VM Size	4294967296
LC_DYLD_INFO_ONLY	00001048	0000000000000000	File Offset	0
LC_SYMTAB	00001050	0000000000000000	File Size	0
LC_DYSYMTAB	00001058	00000000	Maximum VM Protection	00000000
LC_LOAD_DYLINKER				VM_PROT_NONE
LC_UUID	0000105C	00000000	Initial VM Protection	00000000
LC_VERSION_MIN_MACOSX				VM_PROT_NONE
LC_SOURCE_VERSION	00001060	00000000	Number of Sections	0
LC_MAIN	00001064	00000000	Flags	
LC_LOAD_DYLIB (.Python)				
LC_LOAD_DYLIB (libSystem.B.dylib)				
LC_LOAD_DYLIB (CoreFoundation)				

可以看到，Mach-O文件中有多条加载命令，加载命令的前两个字段分别是Command和Command Size。load command的数据结构定义如下：

```
struct load_command {
    uint32_t cmd;          /* type of load command */
    uint32_t cmdsize;      /* total size of command in bytes */
};
```

cmdsize字段表示当前加载命令的大小。

cmd字段代表当前加载命令的类型，加载命令的类型不同，结构体就不同。对于不同类型的加载命令，他们都会在load_command结构体后面加上一个或者多个字段来表示自己特定的结构体信息。加载命令的类型比较多，其部分取值如下：

```
#define LC_SEGMENT 0x1 /* segment of this file to be mapped */
#define LC_THREAD 0x4 /* thread */
#define LC_UNIXTHREAD 0x5 /* unix thread (includes a stack) */
#define LC_PREPAGE 0xa /* prepage command (internal use) */
#define LC_DYSYMTAB 0xb /* dynamic link-edit symbol table info */
#define LC_LOAD_DYLIB 0xc /* load a dynamically linked shared library */
#define LC_CODE_SIGNATURE 0x1d /* local of code signature */
.....
```

下面介绍一些常见的commands。

LC_MAIN

表示main函数在Mach-O文件中的偏移

LC_UUID

确定文件的唯一标识，crash解析中也会有这个，去检测dsym文件和crash文件是否匹配

LC_LOAD_DYLINKER

指定动态链接器linker的路径，通常为

```
/usr/lib/dyld
```

LC_LOAD_DYLIB

描述当前Mach-O文件需要哪些动态库

LC_DYLD_INFO_ONLY && LC_DYLD_INFO

这两个命令主要是给动态链接器 dyld 做 link 使用的。动态 linker 的主要作用是二进制加载到内存后，将动态链接库的信息链接起来。

该命令对应的结构体是：


```

struct dyld_info_command {
    uint32_t cmd;      /* LC_DYLD_INFO or LC_DYLD_INFO_ONLY */
    uint32_t cmdsize;   /* sizeof(struct dyld_info_command) */

    uint32_t rebase_off; /* file offset to rebase info */
    uint32_t rebase_size; /* size of rebase info */

    uint32_t bind_off;   /* file offset to binding info */
    uint32_t bind_size;  /* size of binding info */

    uint32_t weak_bind_off; /* file offset to weak binding info */
    /*
    uint32_t weak_bind_size; /* size of weak binding info */

    uint32_t lazy_bind_off; /* file offset to lazy binding info */
    /*
    uint32_t lazy_bind_size; /* size of lazy binding infs */

    uint32_t export_off; /* file offset to lazy binding info */
    uint32_t export_size; /* size of lazy binding infs */
};

```

1. rebase 地址重定位
2. bind 地址绑定，找到使用的外部符号信息
3. weak bind 弱引用绑定，C++ 程序需要的
4. lazy bind 地址懒绑定，延迟绑定，有的外部函数，不需要在加载时就完成地址绑定，可以在运行到的时候再去查找外部函数地址
5. export 要导出的函数信息

LC_SYMTAB

符号表。命令对应的结构体是：

```

struct symtab_command {
    uint32_t cmd;      /* LC_SYMTAB */
    uint32_t cmdsize;   /* sizeof(struct symtab_command) */
    uint32_t symoff;    /* symbol table offset */
    uint32_t nsyms;     /* number of symbol table entries */
    uint32_t stroff;    /* string table offset */
    uint32_t strsize;   /* string table size in bytes */
};

```

1. symoff 符号表在 MachO 文件中的偏移
2. nsyms 符号数量
3. stroff 字符串表在 MachO 文件中的偏移
4. strsize 字符串表大小

LC_CODE_SIGNATURE

应用签名相关信息。

LC_SEGMENT_64

表示的是将64位的段映射到进程的地址空间。可以看一下段加载命令的数据结构：

```
struct segment_command_64 { /* for 64-bit architectures */
    uint32_t    cmd;        /* LC_SEGMENT_64 */
    uint32_t    cmdsize;    /* includes sizeof section_64 structs
*/
    char        segname[16]; /* segment name */
    uint64_t    vmaddr;     /* memory address of this segment */
    uint64_t    vmsize;     /* memory size of this segment */
    uint64_t    fileoff;    /* file offset of this segment */
    uint64_t    filesize;   /* amount to map from the file */
    vm_prot_t   maxprot;    /* maximum VM protection */
    vm_prot_t   initprot;   /* initial VM protection */
    uint32_t    nsects;     /* number of sections in segment */
    uint32_t    flags;      /* flags */
};
```

cmd 和 cmdsize上面已经介绍过了，不再重复。

segname字段表示的是该segment的名称。

vmaddr字段表示段的虚拟内存地址。

vmsize字段表示段所占的虚拟内存的大小。

fileoff字段表示段数据在文件中的偏移。

filesize字段表示段数据的实际大小。

maxprot字段表示页面所需要的最高内存保护。

initprot字段表示页面初始的内存保护。

nsects字段表示该segment包含了多少个section(节区)，一个段可以包含0个或者多个section。

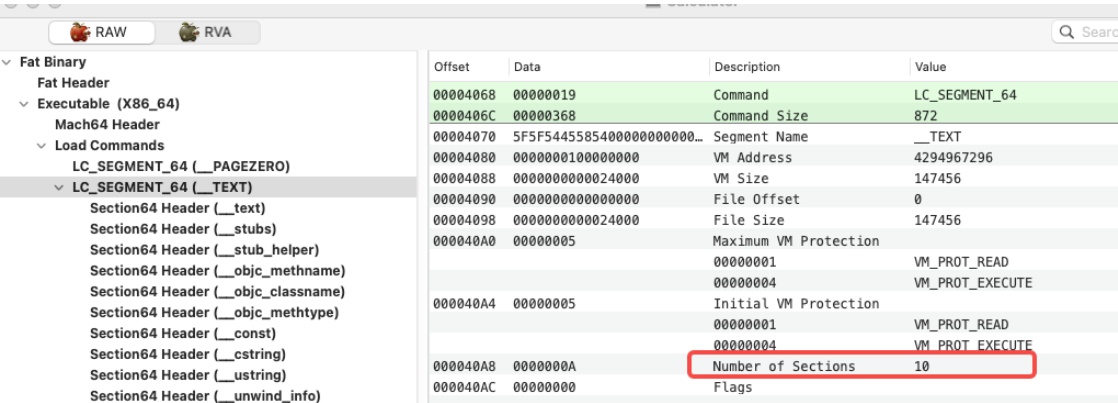
flags字段是段的标志信息。

加载命令中有fileoff、filesize、VM Adress、VM Size几个字段。表示的是将offset处加载filesize大小到虚拟内存VM Adress开始的地方，占用的虚拟内存大小是VM Size。

Segment

上面多次提到了Segment,Segment可以翻译为段。Segment可以理解成是编译器将权限相同的多个Section合并后形成的Section集合，就是段。每一个段的权限相同，或者说，编译时候，编译器把权限相同的数据放在一起，成为段（segment）。一个段可以有0个或者多个section组成。

如下图，__TEXT Segment包含10个section。



Offset	Data	Description	Value
00004068	00000019	Command	LC_SEGMENT_64
0000406C	00000368	Command Size	872
00004070	5F5F544558540000000000...	Segment Name	__TEXT
00004080	0000000100000000	VM Address	4294967296
00004088	0000000000024000	VM Size	147456
00004090	0000000000000000	File Offset	0
00004098	0000000000024000	File Size	147456
000040A0	00000005	Maximum VM Protection	
	00000001		VM_PROT_READ
	00000004		VM_PROT_EXECUTE
000040A4	00000005	Initial VM Protection	
	00000001		VM_PROT_READ
	00000004		VM_PROT_EXECUTE
000040A8	0000000A	Number of Sections	10
000040AC	00000000	Flags	

介绍一下常见的Segment。

_PAGEZERO

_PAGEZERO段用于存储空指针，不具有访问权限。因此上述几个字段都是0。
_PAGEZERO一般为Mach-O可执行文件的第一个Segment

__TEXT

__TEXT segment 包含可执行代码块和只读数据，代码和只读数据在映射后都不具备内存写属性。__TEXT 中可以包含多 section，比如 __text, __cstring, __picsymbol_stub, __symbol_stub, __const, __literal4, __literal8

__DATA

__DATA segment 包含写属性的数据，也常包含多个 section，比如 __data, __la_symbol_ptr, __nl_symbol_ptr, __dyld, __const, __mod_init_func, __mod_term_func, __bss, __common

__LINKEDIT

链接段包含了一些符号表、间接符号表、rebase操作码、绑定操作码、导出符号、函数启动信息、数据表、代码签名、字符串表等数据。该加载命令下没有Section，需要配合LC_SYMTAB来解析symbol table和string table。

section

来简单看一下section的数据结构：

```
struct section_64 { /* for 64-bit architectures */
    char        sectname[16]; /* name of this section */
    char        segname[16]; /* segment this section goes in */
    uint64_t    addr; /* memory address of this section */
    uint64_t    size; /* size in bytes of this section */
    uint32_t    offset; /* file offset of this section */
    uint32_t    align; /* section alignment (power of 2) */
    uint32_t    reloff; /* file offset of relocation entries */
    uint32_t    nreloc; /* number of relocation entries */
    uint32_t    flags; /* flags (section type and
attributes)*/
    uint32_t    reserved1; /* reserved (for offset or index) */
    uint32_t    reserved2; /* reserved (for count or sizeof) */
    uint32_t    reserved3; /* reserved */
};
```

sectname字段表示该section的name。section名都是小写字母，如__text, __data, __cstring等。

segname字段表示该section所属的segment的segmentName。

addr字段表示该section的内存起始地址。

size字段表示该section的大小。

offset字段表示该section相对文件的偏移量。

align字段表示字节区的内存对齐边界。

reloff表示重定位信息的文件偏移。

nreloc表示重定位条目的数目。

flags是section的一些标志属性。

使用MachOview看一下Section的信息：

▼ Fat Binary	Offset	Data	Description	Value
Fat Header	00001288	5F5F6E6C5F73796D626F6C5...	Section Name	__nl_symbol_ptr
▼ Executable (X86_64)	00001298	5F5F444154410000000000...	Segment Name	__DATA
Mach64 Header	000012A8	0000000100002000	Address	4294975488
▼ Load Commands	000012B0	0000000000000010	Size	16
LC_SEGMENT_64 (__PAGEZERO)	000012B8	00002000	Offset	8192
▶ LC_SEGMENT_64 (__TEXT)	000012BC	00000003	Alignment	8
▼ LC_SEGMENT_64 (__DATA)	000012C0	00000000	Relocations Offset	0
Section64 Header (__nl_symbol_ptr)	000012C4	00000000	Number of Relocations	0
Section64 Header (__got)	000012C8	00000006	Flags	00000006
Section64 Header (__la_symbol_ptr)				S_NON_LAZY_SYMBOL_POINTERS
Section64 Header (__data)	000012CC	00000014	Indirect Sym Index	20
LC_SEGMENT_64 (__LINKEDIT)	000012D0	00000000	Reserved2	0
LC_DYLD_INFO_ONLY	000012D4	00000000	Reserved3	0
LC_SYMTAB				
LC_DYSYMTAB				
LC_LOAD_DYLINKER				
LC_UUID				

这是Mach-O文件中某个Section的信息，和section的数据结构是一一对应的。

介绍一下常见的section，前面是Segment名称，后面是section名称。注意：Segment名称都是大写，section名称都是小写。

__TEXT.__text

主程序代码

__TEXT.__cstring

C 语言字符串

__TEXT.__const

const 关键字修饰的常量

__TEXT.__stubs

用于 Stub 的占位代码，很多地方称之为桩代码。

__TEXT.__stubs_helper

当 Stub 无法找到真正的符号地址后的最终指向

__TEXT.__objc_methname

Objective-C 方法名称

__TEXT.__objc_methtype

Objective-C 方法类型

__TEXT.__objc_classname

Objective-C 类名称

__DATA.__data

初始化过的可变数据

__DATA.__la_symbol_ptr

lazy binding 的指针表，表中的指针一开始都指向 __stub_helper

__DATA.nl_symbol_ptr

非 lazy binding 的指针表，每个表项中的指针都指向一个在装载过程中，被动态链机器搜索完成的符号

__DATA.__const

没有初始化过的常量

__DATA.__cfstring

程序中使用的 Core Foundation 字符串（CFStringRef）

__DATA.__bss

BSS，存放为初始化的全局变量，即常说的静态内存分配

__DATA.__common

没有初始化过的符号声明

__DATA.__objc_classlist

Objective-C 类列表

__DATA.__objc_protolist

Objective-C 原型

__DATA.__objc_imginfo

Objective-C 镜像信息

__DATA.__objc_selfrefs

Objective-C self 引用

__DATA.__objc_protorefs

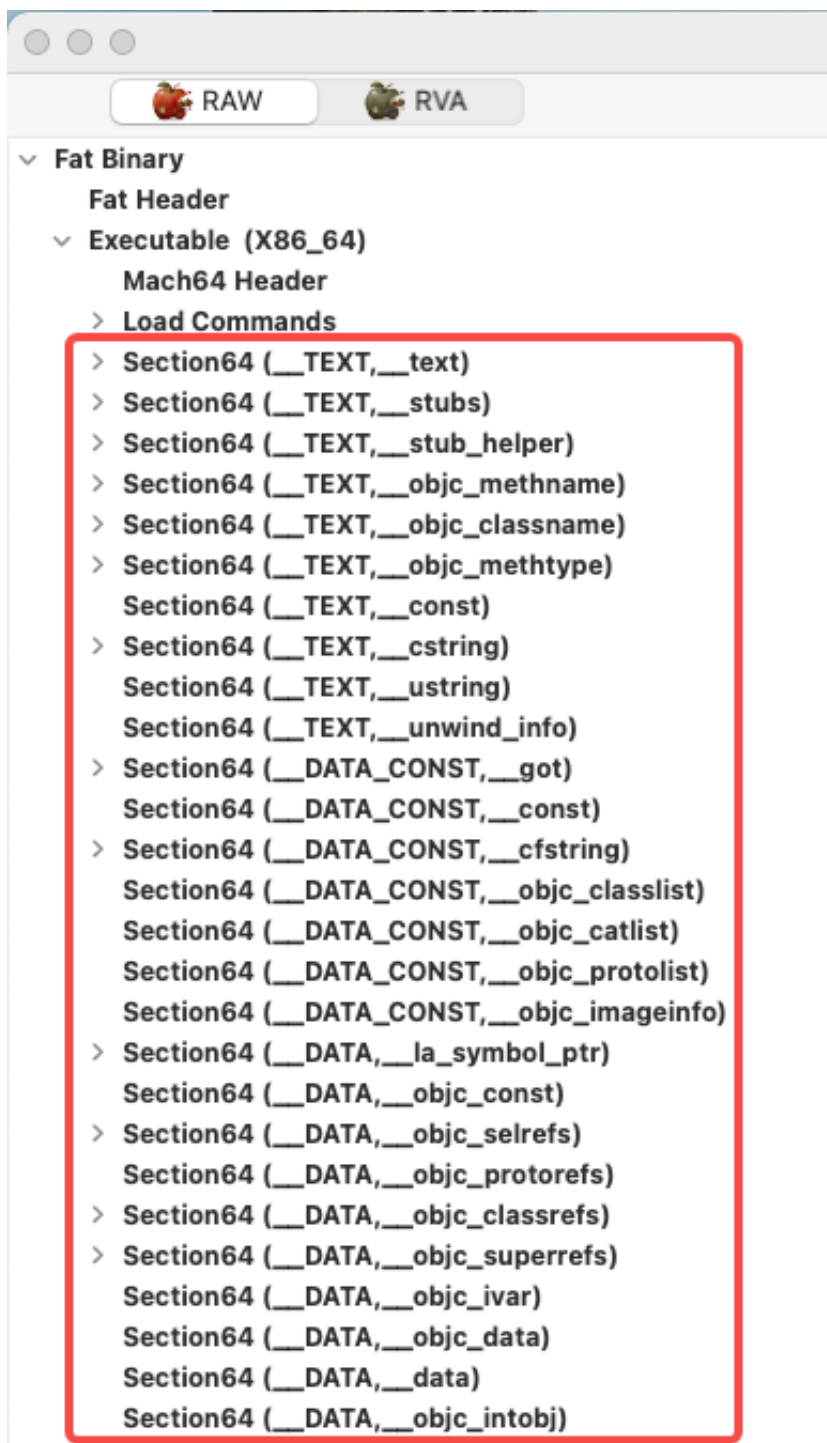
Objective-C 原型引用

__DATA.__objc_superrefs

Objective-C 超类引用

Data

Mach-O中Load Commands之后的就是Data数据。每个段的数据都保存在这里，这里存放了具体的数据与代码。使用MachOView看一下Data：



上图红框中的就是Data部分。

开头说了，Load Commands和数据有对应关系。具体来说，一个commands对应一块数据。如下：

__TEXT Segment包含10个Section，


```

v Load Commands
  LC_SEGMENT_64 (__PAGEZERO)
v LC_SEGMENT_64 (__TEXT)
  Section64 Header (__text)
  Section64 Header (__stubs)
  Section64 Header (__stub_helper)
  Section64 Header (__objc_methname)
  Section64 Header (__objc_classname)
  Section64 Header (__objc_methtype)
  Section64 Header (__const)
  Section64 Header (__cstring)
  Section64 Header (__ustring)
  Section64 Header (__unwind_info)

```

这10个Section对应的数据是：

```

-----
> Section64 (__TEXT,__text)
> Section64 (__TEXT,__stubs)
> Section64 (__TEXT,__stub_helper)
> Section64 (__TEXT,__objc_methname)
> Section64 (__TEXT,__objc_classname)
> Section64 (__TEXT,__objc_methtype)
  Section64 (__TEXT,__const)
> Section64 (__TEXT,__cstring)
  Section64 (__TEXT,__ustring)
  Section64 (__TEXT,__unwind_info)

```

__DATA_CONST Segment包含7个Section,

```

-----
v Load Commands
  LC_SEGMENT_64 (__PAGEZERO)
> LC_SEGMENT_64 (__TEXT)
v LC_SEGMENT_64 (__DATA_CONST)
  Section64 Header (__got)
  Section64 Header (__const)
  Section64 Header (__cfstring)
  Section64 Header (__objc_classlist)
  Section64 Header (__objc_catlist)
  Section64 Header (__objc_protolist)
  Section64 Header (__objc_imageinfo)

```

这7个Section对应的数据是：

```
> Section64 (__DATA_CONST,__got)
  Section64 (__DATA_CONST,__const)
> Section64 (__DATA_CONST,__cfstring)
  Section64 (__DATA_CONST,__objc_classlist)
  Section64 (__DATA_CONST,__objc_catlist)
  Section64 (__DATA_CONST,__objc_protolist)
  Section64 (__DATA_CONST,__objc_imageinfo)
```

Loader info链接信息

链接信息包含了动态加载信息 Dynamic Loader Info、函数起始地址表、符号表、动态符号表、字符串表等信息，如下所示：

```
> Section64 (__DYNAMIC,__dynamic_info)
  > Dynamic Loader Info
  > Function Starts
  > Data in Code Entries
  > Symbol Table
  > Dynamic Symbol Table
  String Table
```

一些问题

指令和数据分开

Mach-O文件中，命令和数据是分开的，为何要这样设计？这样设计有三个好处：

1. 数据和指令可以被映射到两个不同的虚拟内存区域。数据区域是可读写的，指令区域是只读的。分开存可以很方便的设置两个区域的权限。
2. 两个区域分离，提高了程序的局部性，有助于提高缓存的命中率。解释一下，就是段越小，功能越单一，越有助于提供缓存命中率。相反如果是很大的段，那么有一模一样段的可能性非常小，缓存命中率也就更低。
3. 系统运行该程序的多个副本时，指令是一样的。那么内存只需要保存一份指令，只生成几份数据就可以，能够节省内存。

Segment和section

Segment中可以包含多个section,为何要设计这样的结构?

部分的Segment, 主要是__TEXT和__DATA, 可以进一步分解为Section。之所以按照Segment->Section的结构组织方式, 是因为在同一个Segment下的section, 可以控制相同权限, 也可以不严格按照Page的大小进行内存对齐, 节省内存空间。而Segment对外整体暴露, 在程序载入阶段映射成一个完整的虚拟内存, 更好的做到内存对齐。