计算机那些事(4)——ELF文件结构

2018-05-21 | 2019-05-14 | 计算机原理

前文结尾说到编译器编译源代码后生成的文件叫做**目标文件**，而目标文件经过编译器链接之后得到的就是**可执行文件**。那么目标文件到底是什么？它和可执行文件又有什么区别？链接到底又做了什么呢？接下来，我们将探索一下目标文件的本质。

目标文件的格式

目前，PC平台流行的 **可执行文件格式（Executable）** 主要包含如下两种，它们都是 **COFF（Common File Format）** 格式的变种。

- Windows下的 **PE（Portable Executable）**
- Linux下的 **ELF（Executable Linkable Format）**

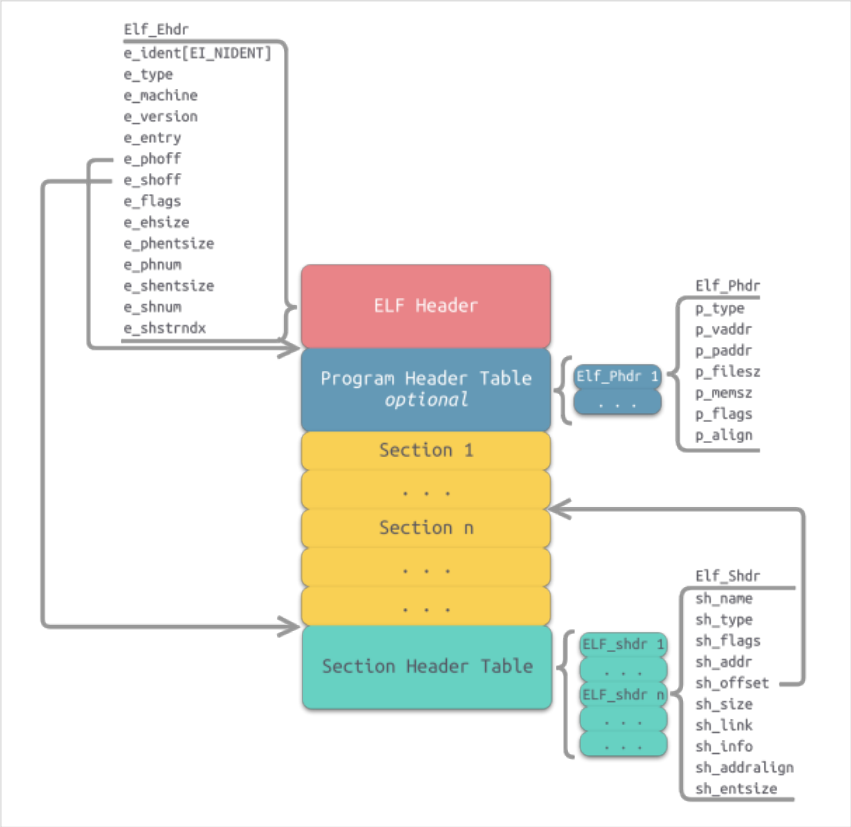
目标文件就是源代码经过编译后但未进行连接的那些中间文件（Windows的`.obj`和Linux的`.o`），它与可执行文件的格式非常相似，所以一般跟可执行文件格式一起采用同一种格式存储。在Windows下采用**PE-COFF**文件格式；Linux下采用**ELF**文件格式。

事实上，除了**可执行文件**外，**动态链接库（DDL，Dynamic Linking Library）**、**静态链接库（Static Linking Library）** 均采用可执行文件格式存储。它们在Window下均按照PE-COFF格式存储；Linux下均按照ELF格式存储。只是文件名后缀不同而已。

- 动态链接库：Windows的`.dll`、Linux的`.so`
- 静态链接库：Windows的`.lib`、Linux的`.a`

下面，我们将以ELF文件为例进行介绍。

ELF文件结构



注意：段（`Segment`）与节（`Section`）的区别。很多地方对两者有所混淆。段是程序执行的必要组成，当多个目标文件链接成一个可执行文件时，会将相同权限的节合并到一个段中。相比而言，节的粒度更小。

如图所示，为ELF文件的基本结构，其主要由四部分组成：

- ELF Header
- ELF Program Header Table (或称Program Headers、程序头)
- ELF Section Header Table (或称Section Headers、节头表)
- ELF Sections

从图中，我们就能看出它们各自的数据结构以及相互之间的索引关系。下面我们依次进行介绍。

ELF Header

我们可以使用readelf工具来查看ELF Header。

```
$ readelf -h hello.o

ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          672 (bytes into file)
```

```
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         13
  Section header string table index: 10
```

ELF文件结构示意图中定义的`Elf_Ehdr`的各个成员的含义与readelf具有对应关系。如下表所示：

成员	含义
e_ident	Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
	Class: ELF32
	Data: 2's complement, little end
	Version: 1(current)
	OS/ABI: UNIX - System V
	ABI Version: 0
e_type	Type: REL (Relocatable file)
	ELF文件类型
e_machine	Machine: Advanced Micro Devices X86-64
	ELF文件的CPI平台属性
e_version	Version: 0x1
	ELF版本号。一般为常数1
e_entry	Entry point address: 0x0
	入口地址，规定ELF程序的入口虚拟地址，操作系统在加载完该程序后从这个地址开始执行进程的指令。可重定位指令一般没有入口地址，则该值为0
e_phoff	Start of program headers: 0(bytes into file)
e_shoff	Start of section headers: 672 (bytes into file)
	Section Header Table 在文件中的偏移
e_word	Flags: 0x0
	ELF标志位，用来标识一些ELF文件平台相关的属性。
e_ehsize	Size of this header: 64 (bytes)
	ELF Header本身的大小
e_phentsize	Size of program headers: 0 (bytes)
e_phnum	Number of program headers: 0

成员	含义
e_shentsize	Size of section headers: 64 (bytes)
	单个Section Header大小
e_shnum	Number of section headers: 13
	Section Header的数量
e_shstrndx	Section header string table index: 10
	Section Header字符串表在Section Header Table中的索引

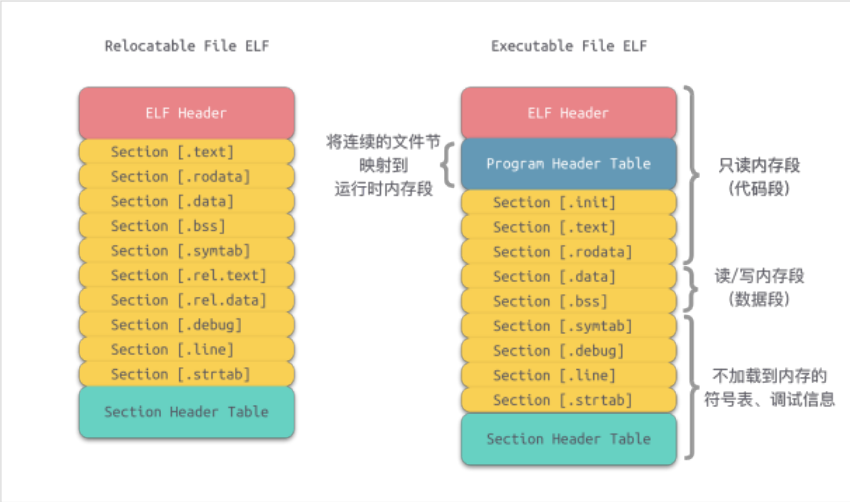
ELF魔数

每种可执行文件的格式的开头几个字节都是很特殊的，特别是开头4个字节，通常被称为**魔数（Magic Number）**。通过对魔数的判断可以确定文件的格式和类型。如：ELF的可执行文件格式的头4个字节为`0x7F`、`e`、`l`、`f`；Java的可执行文件格式的头4个字节为`c`、`a`、`f`、`e`；如果被执行的是Shell脚本或perl、python等解释型语言的脚本，那么它的第一行往往是`#!/bin/sh`或`#!/usr/bin/perl`或`#!/usr/bin/python`，此时前两个字节`#`和`!`就构成了魔数，系统一旦判断到这两个字节，就对后面的字符串进行解析，以确定具体的解释程序路径。

ELF文件类型

ELF文件主要有三种类型，可以通过ELF Header中的`e_type`成员进行区分。

- **可重定位文件（Relocatable File）**：`ETL_REL`。一般为`.o`文件。可以被链接成可执行文件或共享目标文件。静态链接库属于可重定位文件。
- **可执行文件（Executable File）**：`ET_EXEC`。可以直接执行的程序。
- **共享目标文件（Shared Object File）**：`ET_DYN`。一般为`.so`文件。有两种情况可以使用。
  - 链接器将其与其他可重定位文件、共享目标文件链接成新的目标文件；
  - 动态链接器将其与其他共享目标文件、结合一个可执行文件，创建进程映像。



ELF Section Header Table

ELF 节头表是一个节头数组。每一个节头都描述了其所对应的节的信息，如节名、节大小、在文件中的偏移、读写权限等。**编译器、链接器、装载器都是通过节头表来定位和访问各个节的属性的。**

我们可以使用readelf工具来查看节头表。

```
$ readelf -S hello.o

There are 13 section headers, starting at offset 0x2a0:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000000000  00000040
       0000000000000015  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  000001f0
       0000000000000030  0000000000000018   I      11     1     8
  [ 3] .data             PROGBITS         0000000000000000  00000055
       0000000000000000  0000000000000000  WA       0     0     1
  [ 4] .bss              NOBITS           0000000000000000  00000055
       0000000000000000  0000000000000000  WA       0     0     1
  [ 5] .rodata           PROGBITS         0000000000000000  00000055
       000000000000000d  0000000000000000   A       0     0     1
  [ 6] .comment          PROGBITS         0000000000000000  00000062
       0000000000000035  0000000000000001  MS       0     0     1
  [ 7] .note.GNU-stack   PROGBITS         0000000000000000  00000097
       0000000000000000  0000000000000000           0     0     1
  [ 8] .eh_frame         PROGBITS         0000000000000000  00000098
       0000000000000038  0000000000000000   A       0     0     8
  [ 9] .rela.eh_frame    RELA             0000000000000000  00000220
       0000000000000018  0000000000000018   I      11     8     8
  [10] .shstrtab         STRTAB           0000000000000000  00000238
       0000000000000061  0000000000000000           0     0     1
  [11] .symtab           SYMTAB           0000000000000000  000000d0
       0000000000000108  0000000000000018          12     9     8
  [12] .strtab           STRTAB           0000000000000000  000001d8
       0000000000000013  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

ELF文件结构示意图中定义的`Elf_Shdr`的各个成员的含义与readelf具有对应关系。如下表所示：

成员	含义
sh_name	节名
	节名是一个字符串，保存在一个名为`.shstrtab`的字符串表（可通过Section Header索引到）。sh_name的值实际上是其节名字符串在`.shstrtab`中的偏移值
sh_type	节类型
sh_flags	节标志位
sh_addr	节地址：节的虚拟地址
	如果该节可以被加载，则sh_addr为该节被加载后在进程地址空间中的虚拟地址；否则sh_addr为0
sh_offset	节偏移

成员	含义
	如果该节存在于文件中，则表示该节在文件中的偏移；否则无意义，如sh_offset对于BSS 节来说是没有意义的
sh_size	节大小
sh_link、sh_info	节链接信息
sh_addralign	节地址对齐方式
sh_entsize	节项大小
	有些节包含了一些固定大小的项，如符号表，其包含的每个符号所在的大小都一样的，对于这种节，sh_entsize表示每个项的大小。**如果为0，则表示该节不包含固定大小的项。**

节类型（sh_type）

节名是一个字符串，只是在链接和编译过程中有意义，但它并不能真正地表示节的类型。对于编译器和链接器来说，主要决定节的属性是节的类型（`sh_type`）和节的标志位（`sh_flags`）。

节的类型相关常量以`SHT_`开头，上述`readelf -S`命令执行的结果省略了该前缀。常见的节类型如下表所示：

常量	值	含义
SHT_NULL	0	无效节
SHT_PROGBITS	1	**程序节。**代码节、数据节都是这种类型。
SHT_SYMTAB	2	**符号表**
SHT_STRTAB	3	**字符串表**
SHT_RELA	4	**重定位表。**该节包含了重定位信息。
SHT_HASH	5	**符号表的哈希表**
SHT_DYNAMIC	6	动态链接信息
SHT_NOTE	7	提示性信息
SHT_NOBITS	8	表示该节在文件中没有内容。如`.bss`节
SHT_REL	9	该节包含了重定位信息
SHT_SHLIB	10	保留
SHT_DNYSYM	11	**动态链接的符号表**

节标志位（sh_flag）

节标志位表示该节在进程虚拟地址空间中的属性。如是否可写、是否可执行等。相关常量以`SHF_`开头。常见的节标志位如下表所示：

常量	值	含义
SHF_WRITE	1	表示该节在进程空间中可写
SHF_ALLOC	2	表示该节在进程空间中需要分配空间。有些包含指示或控制信息的节不需要在进程空间中分配空间，就不会有这个标志。
SHF_EXECINSTR	4	表示该节在进程空间中可以被执行

节链接信息（sh_link、sh_info）

如果节的类型是与链接相关的（无论是动态链接还是静态链接），如**重定位表、符号表、**等，则`sh_link`、`sh_info`两个成员所包含的意义如下所示。其他类型的节，这两个成员没有意义。

sh_type	sh_link	sh_info
SHT_DYNAMIC	该节所使用的**字符串表**在节头表中的下标	0
SHT_HASH	该节所使用的**符号表**在节头表中的下标	0
SHT_REL	该节所使用的**相应符号表**在节头表中的下标	该重定位表所作用的节在节头表中的下标
SHT_RELA	该节所使用的**相应符号表**在节头表中的下标	该重定位表所作用的节在节头表中的下标
SHT_SYMTAB	操作系统相关	操作系统相关
SHT_DYNSYM	操作系统相关	操作系统相关
other	SHN_UNDEF	0

ELF Sections

节的分类

上述ELF Section Header Table部分已经简单介绍了节类型。接下来我们来介绍详细一些比较重要的节。

.text节

`.text`节是保存了程序代码指令的**代码节。一段可执行程序，如果存在Phdr，则`.text`节就会存在于`text`段中。**由于`.text`节保存了程序代码，所以节类型为`SHT_PROGBITS`。

.rodata节

`rodata`节保存了只读的数据，如一行C语言代码中的字符串。由于`.rodata`节是只读的，所以只能存在于一个可执行文件的**只读段**中。因此，只能在`text`段（不是`data`段）中找到`.rodata`节。由于`.rodata`节是只读的，所以节类型为`SHT_PROGBITS`。

.plt节（过程链接表）

`.plt`节也称为**过程链接表（Procedure Linkage Table），其包含了动态链接器调用从共享库导入的函数所必需的相关代码。**由于`.plt`节保存了代码，所以节类型为`SHT_PROGBITS`。

.data节

`.data`节存在于`data`段中，**其保存了初始化的全局变量等数据。**由于`.data`节保存了程序的变量数据，所以节类型为`SHT_PROGBITS`。

.bss节

`.bss`节存在于`data`段中，占用空间不超过4字节，仅表示这个节本省的空间。**`.bss`节保存了未进行初始化的全局数据。**程序加载时数据被初始化为0，在程序执行期间可以进行赋值。由于`.bss`节未保存实际的数据，所以节类型为`SHT_NOBITS`。

.got.plt节（全局偏移表-过程链接表）

`.got`节保存了**全局偏移表。`.got`节和`.plt`节一起提供了对导入的共享库函数的访问入口，由动态链接器在运行时进行修改。**由于`.got.plt`节与程序执行有关，所以节类型为`SHT_PROGBITS`。

.dynsym节（动态链接符号表）

`.dynsym`节保存在`text`段中。**其保存了从共享库导入的动态符号表。**节类型为`SHT_DYNSYM`。

.dynstr节（动态链接字符串表）

`.dynstr`保存了动态链接字符串表，表中存放了一系列字符串，这些字符串代表了符号名称，以空字符作为终止符。

.rel.*节（重定位表）

重定位表保存了重定位相关的信息，**这些信息描述了如何在链接或运行时，对ELF目标文件的某部分或者进程镜像进行补充或修改。**由于重定位表保存了重定位相关的数据，所以节类型为`SHT_REL`。

.hash节

`.hash`节也称为`.gnu.hash`，其保存了一个用于查找符号的散列表。

.symtab节（符号表）

`.symtab`节是一个`ElfN_Sym`的数组，保存了符号信息。节类型为`SHT_SYMTAB`。

.strtab节（字符串表）

`.strtab`节保存的是符号字符串表，表中的内容会被`.symtab`的`ElfN_Sym`结构中的`st_name`引用。节类型为`SHT_STRTAB`。

.ctors节和.dtors节

`.ctors`（**构造器**）节和`.dtors`（**析构器**）节分别保存了指向构造函数和析构函数的函数指针，**构造函数是在main函数执行之前需要执行的代码；析构函数是在main函数之后需要执行的代码。**

符号表

节的分类中我们介绍了`.dynsym`节和`.symtab`节，两者都是符号表。那么它们到底有什么区别呢？存在什么关系呢？

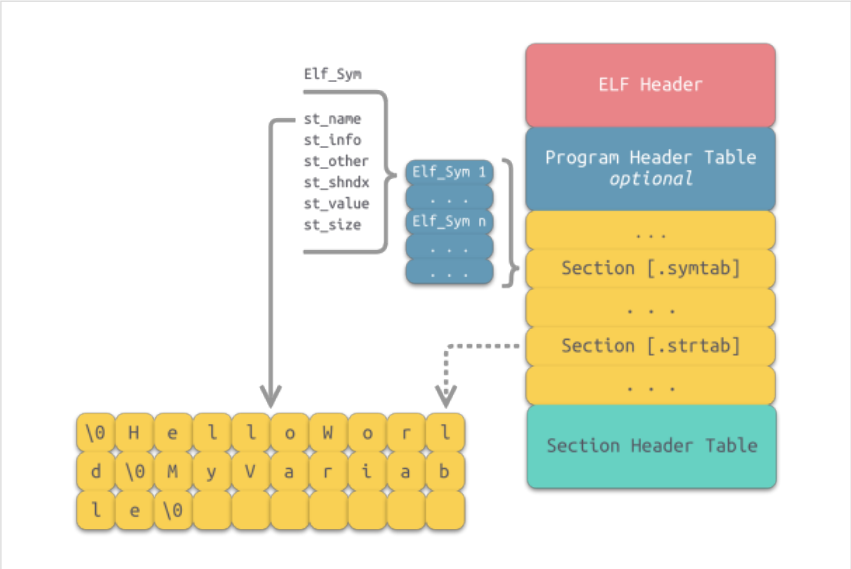
符号是对某些类型的数据或代码（如全局变量或函数）的符号引用，函数名或变量名就是符号名。例如，`printf()`函数会在动态链接符号表`.dynsym`中存有一个指向该函数的符号项（以`Elf_Sym`数据结构表示）。在大多数共享库和动态链接可执行文件中，存在两个符号表。即`.dynsym`和`.symtab`。

`.dynsym`保存了引用来自外部文件符号的全局符号。如`printf`库函数。**`.dynsym`保存的符号是`.symtab`所保存符合的子集，`.symtab`中还保存了可执行文件的本地符号。**如全局变量，代码中定义的本地函数等。

既然`.dynsym`是`.symtab`的子集，那为何要同时存在两个符号表呢?

通过`readelf -S`命令可以查看可执行文件的输出，一部分节标志位（`sh_flags`）被标记为了**A（ALLOC）、WA（WRITE/ALLOC）、AX（ALLOC/EXEC）**。其中，`.dynsym`被标记为ALLOC，而`.symtab`则没有标记。

ALLOC表示有该标记的节会在运行时分配并装载进入内存，而`.symtab`不是在运行时必需的，因此不会被装载到内存中。**`.dynsym`保存的符号只能在运行时被解析，因此是运行时动态链接器所需的唯一符号。**`.dynsym`对于动态链接可执行文件的执行是必需的，而`.symtab`只是用来进行调试和链接的。



上图所示为通过符号表索引字符串表的示意图。符号表中的每一项都是一个`Elf_Sym`结构，对应可以在字符串表中索引得到一个字符串。该数据结构中成员的含义如下表所示：

成员	含义
st_name	符号名。该值为该符号名在字符串表中的偏移地址。
st_value	符号对应的值。存放符号的值（可能是地址或位置偏移量）。
st_size	符号的大小。
st_other	0
st_shndx	符号所在的节
st_info	符号类型及绑定属性

使用readelf工具我们也能够看到符号表的相关信息。

```
$ readelf -s hello.o

Symbol table '.symtab' contains 11 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hello.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
```

```
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     9: 0000000000000000    21 FUNC    GLOBAL DEFAULT    1 main
    10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
```

字符串表

类似于符号表，在大多数共享库和动态链接可执行文件中，也存在两个字符串表。即`.dynstr`和`.strtab`，分别对应于`.dynsym`和`symtab`。此外，还有一个`.shstrtab`的节头字符串表，用于保存节头表中用到的字符串，可通过`sh_name`进行索引。

ELF文件中所有字符表的结构基本一致，如上图所示。

重定位表

重定位就是将符号定义和符号引用进行连接的过程。可重定位文件需要包含描述如何修改节内容的相关信息，从而使可执行文件和共享目标文件能够保存进程的程序镜像所需要的正确信息。

重定位表是进行重定位的重要依据。我们可以使用objdump工具查看目标文件的重定位表：

```
$ objdump -r hello.o


hello.o:     file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000005 R_X86_64_32       .rodata
000000000000000a R_X86_64_PC32     puts-0x0000000000000004


RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text
```

重定位表是一个`Elf_Rel`类型的数组结构，每一项对应一个需要进行重定位的项。
其成员含义如下表所示：

成员	含义
r_offset	重定位入口的偏移。
	对于**可重定位文件**来说，这个值是该重定位入口所要修正的位置的第一个字节相对于节起始的偏移
	对于**可执行文件或共享对象文件**来说，这个值是该重定位入口所要修正的位置的第一个字节的虚拟地址
r_info	重定位入口的类型和符号
	因为不同处理器的指令系统不一样，所以重定位所要修正的指令地址格式也不一样。每种处理器都有自己的一套重定位入口的类型。
	对于**可执行文件和共享目标文件**来说，它们的重定位入口是动态链接类型的。

重定位是目标文件链接成为可执行文件的关键。我们将在后面的进行介绍。

参考

1. Executable and Linkable Format (ELF)
2. 《Linux 二进制分析》
3. 《程序员的自我修养——链接、装载与库》
4. Executable and Linkable Format

(完)

ELF文件 # 目标文件 # 可执行文件

‹ 计算机那些事(3)——程序构建及编译原理

计算机那些事(5)——链接、静态链接、动态链接 ›