目标文件

- 特性
  - 后缀 — linux — .o
  - 包含代码和数据
  - 可经过链接器处理生成
    - 可执行文件
    - 共享目标文件
- 结构
  - 特性 — 开头是文件头，后面是一个个段，再后面是一个个 table
  - 组成
    - 文件头 (ELF Header)
      - 内涵 — 描述了整个文件的属性
      - 内容
        - 文件类型
          - 可执行
          - 静态、动态链接
        - 入口地址 — 如果是可执行文件
        - 目标硬件
        - 目标操作系统
        - ......
      - 结构
        - 位置 — /usr/include/elf.h
        - 32bit — ELF描述符

```
typedef struct
{
  unsigned char  e_ident[EI_NIDENT];   /* Magic number and other info */
  Elf32_Half     e_type;               /* Object file type */
  Elf32_Half     e_machine;            /* Architecture */
  Elf32_Word     e_version;            /* Object file version */
  Elf32_Addr     e_entry;              /* Entry point virtual address */
  Elf32_Off      e_phoff;              /* Program header table file offset */
  Elf32_Off      e_shoff;              /* Section header table file offset */
  Elf32_Word     e_flags;              /* Processor-specific flags */
  Elf32_Half     e_ehsize;             /* ELF header size in bytes */
  Elf32_Half     e_phentsize;          /* Program header table entry size */
  Elf32_Half     e_phnum;              /* Program header table entry count */
  Elf32_Half     e_shentsize;          /* Section header table entry size */
  Elf32_Half     e_shnum;              /* Section header table entry count */
  Elf32_Half     e_shstrndx;           /* Section header string table index */
} Elf32_Ehdr;
```

      - 查看
        - 命令 — readelf -h <file>

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          1300 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         19
  Section header string table index: 18
```

        - 打印内容与 结构体对应

表 3-4 ELF 文件头结构成员含义

成员	readelf 输出结果与含义
e_ident	Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00 Class: ELF32 Data: 2's complement, little endian 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: Intel 80386 ELF 文件的 CPU 平台属性，相关常量以 EM_ 开头
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。可重定位文件一般没有入口地址，则这个值为 0
e_phoff	Start of program headers: 0 (bytes into file) 这个暂时不关心，请参考后面的 "ELF 链接视图和执行视图" 一节
e_shoff	Start of section headers: 280 (bytes into file) 段表在文件中的偏移，上面的例子里这个值是 280，也就是段表从文件的第 281 个字节开始
e_word	Flags: 0x0 ELF 标志位，用来标识一些 ELF 文件平台相关的属性。相关常量的格式一般为 EF_machine_flag，machine 为平台，flag 为标志
e_ehsize	Size of this header: 52 (bytes) 即 ELF 文件头本身的大小，这个例子里面为 52 字节
e_phentsize	Size of program headers: 0 (bytes) 这个暂时不关心，请参考后面的 "ELF 链接视图和执行视图" 一节
e_phnum	Number of program headers: 0 这个暂时不关心，请参考后面的 "ELF 链接视图和执行视图" 一节
e_shentsize	Size of section headers: 40 (bytes) 段表描述符的大小，这个一般等于 sizeof(Elf32_Shdr)，具体参照 "段表" 一节
e_shnum	Number of section headers: 11 段表描述符数量，这个值等于 ELF 文件中拥有的段的数量，上面那个例子里面为 11
e_shstrndx	Section header string table index: 8 段表字符串表所在的段在段表中的下标，这个名称有点绕口，一下子反应不过来？没关系，让我们在后面探讨了什么是字符串表之后再回头来看这个

      - 结果
        - 前16字节
          - 1-4 字节 — ELF 魔数
            - 内涵 — 7f 45 4c 46 — ''.encode().fromhex('7f454c46') b'\x7fELF'
            - 特性 — 魔数错误，系统会拒绝加载
          - 5 字节 — 表示 32位还是 64位
            - 32位为1
            - 64位为2
          - 6 字节 — 表示字节序
            - 小端为1
            - 大端为2
          - 7 字节 — 表示版本号 — 1
          - 8-16 字节 — 0
        - Type — 文件类型
          - 内涵
            - 可执行文件为2
            - 课重定位文件为 1
            - 共享目标文件为 3
        - Machine — 机器类型
        - Section header string table index — .shstrtab 这个段，即段表字符串表，在段表中的下标，段表是个数组
    - 段 (Section)
      - 本质 — 文件中的一块数据
      - 特性 — 不同属性的信息存储在特定的段中
      - 属性
        - 特性 — 真正决定段的属性的是段类型 (sh_type) 和 段的标志位 (sh_flags)

Name	sh_type	sh_flag
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.comment	SHT_PROGBITS	none
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.debug	SHT_PROGBITS	none
.dynamic	SHT_DYNAMIC	SHF_ALLOC + SHF_WRITE 在有些系统下 .dynamic 段可能是只读的，所以没有 SHF_WRITE 标志位
.hash	SHT_HASH	SHF_ALLOC
.line	SHT_PROGBITS	none
.note	SHT_NOTE	none
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	none
.strtab	SHT_STRTAB	如果该 ELF 文件中有可装载的段须要用到该字符串表，那么该字符串表也将被装载到进程空间，则有 SHF_ALLOC 标志位
.symtab	SHT_SYMTAB	同字符串表
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR

        - 段类型 (sh_type) — 内涵

常量	值	含义
SHT_NULL	0	无效段
SHT_PROGBITS	1	程序段、代码段、数据段都是这种类型的
SHT_SYMTAB	2	表示该段的内容为符号表
SHT_STRTAB	3	表示该段的内容为字符串表
SHT_RELA	4	重定位表。该段包含了重定位信息，具体参考 "静态地址决议和重定位" 这一节
SHT_HASH	5	符号表的哈希表。见 "符号表" 这一节
SHT_DYNAMIC	6	动态链接信息。具体见 "动态链接" 一章
SHT_NOTE	7	提示性信息
SHT_NOBITS	8	表示该段在文件中没内容，比如.bss 段
SHT_REL	9	该段包含了重定位信息，具体参考 "静态地址决议和重定位" 这一节
SHT_SHLIB	10	保留
SHT_DNYSYM	11	动态链接的符号表。具体见 "动态链接" 一章

        - 段的标志位 (sh_flag) — 内涵

常量	值	含义
SHF_WRITE	1	表示该段在进程空间中可写
SHF_ALLOC	2	表示该段在进程空间中须要分配空间。有些包含指示或控制信息的段不须要在进程空间中被分配空间，它们一般不会有这个标志。像代码段、数据段和.bss 段都会有这个标志位
SHF_EXECINSTR	4	表示该段在进程空间中可以被执行，一般指代码段

      - 系统保留段
        - 代码段 (Code Section)
          - 内涵 — 存放机器指令
          - 名称
            - .text
            - .code
          - 特性 — 目标文件中
            - 起始地址为 0x00000000
            - 引用未定义在本文件中的符号时 — 引用的值或地址是没有意义的，要通过链接重定位
        - 数据段 (Data Section)
          - 内涵 — 存放
            - 已初始化的全局变量
            - 已初始化的局部静态变量
            - (非静态的变量在运行时体现在寄存器或栈上)
          - 名称 — .data
        - 只读数据段 (Read Only Data Section)
          - 内涵 — 存放常量
            - const 的变量
            - 字符串常量 — 有时候会放到 .data 中
          - 特性 — 字符串常量有时候会放到 .data 中
          - 名称 — .rodata
        - BSS 段 (Block Started by Symbol)
          - 内涵 — 存放
            - 可执行文件中 — 未初始化的全局变量
            - 未初始化的局部静态变量
          - 名称 — .bss
          - 特性
            - 有些编译器 不会将 未初始化的全局变量 放在 .bss
            - 初始化但是 为0 的局部静态变量，放在 .bss 中
            - 在 文件内的大小是0，但是装载的时候会占用空间
        - .plt
        - .got
        - 其他段
          - 注释信息段 (Comment Section) — 名称 — .comment
          - 堆栈提示段
            - 名称 — .note.GNU-stack
            - 特性 — objdum -h ，有CONTENTS 但是 size 为0
          - .rodata1、.debug、.dynamic、.hash、.line、.note、.strtab、.symtab、shstrtab、.init、.fini
    - 表 (Table)
      - 本质 — 辅助性的段
      - 类型
        - 段表 (Section Header Table)
          - 内涵 — 描述每一个段的属性、段在文件中的偏移位置
          - 特性 — 编译器、链接器、装载器都是依靠段表来定位和访问各个段的属性
          - 结构 — 一个以 Elf32_Shdr 为元素，即段描述符，的数组

```
typedef struct
{
  Elf32_Word   sh_name;        /* Section name (string tbl index) */
  Elf32_Word   sh_type;        /* Section type */
  Elf32_Word   sh_flags;       /* Section flags */
  Elf32_Addr   sh_addr;        /* Section virtual addr at execution */
  Elf32_Off    sh_offset;      /* Section file offset */
  Elf32_Word   sh_size;        /* Section size in bytes */
  Elf32_Word   sh_link;        /* Link to another section */
  Elf32_Word   sh_info;        /* Additional section information */
  Elf32_Word   sh_addralign;   /* Section alignment */
  Elf32_Word   sh_entsize;     /* Entry size if section holds table */
} Elf32_Shdr;
```

          - 查看
            - 命令 — readelf -S <file>
            - 打印内容与 结构体对应

成员	含义
sh_name	Section name 段名 段名是个字符串，它位于一个叫做 ".shstrtab" 的字符串表。sh_name 是段名字符串在 ".shstrtab" 中的偏移
sh_type	Section type 段的类型 详见后文 "段的类型"
sh_flags	Section flag 段的标志位 详见后文 "段的标志位"
sh_addr	Section Address 段虚拟地址 如果该段可以被加载，则 sh_addr 为该段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	Section Offset 段偏移 如果该段存在于文件中，则表示该段在文件中的偏移；否则无意义。比如 sh_offset 对于 BSS 段来说就没有意义
sh_size	Section Size 段的长度
sh_link 和 sh_info	Section Link and Section Information 段链接信息 详见后文 "段的链接信息"
sh_addralign	Section Address Alignment 段地址对齐 有些段对段地址对齐有要求，比如我们假设有个段刚开始的位置包含了一个 double 变量，因为 Intel x86 系统要求浮点数的存储地址必须是本身的整数倍，也就是说保存 double 变量的地址必须是 8 字节的整数倍。这样对一个段来说，它的 sh_addr 必须是 8 的整数倍。由于地址对齐的数量都是 2 的指数倍，sh_addralign 表示是地址对齐数量中的指数，即 sh_addralign = 3 表示对齐为 2 的 3 次方倍，即 8 倍，依此类推。所以一个段的地址 sh_addr 必须满足下面的条件，即 sh_addr % (2 ** sh_addralign) = 0。** 表示指数运算。如果 sh_addralign 为 0 或 1，则表示该段没有对齐要求
sh_entsize	Section Entry Size 项的长度 有些段包含了一些固定大小的项，比如符号表，它包含的每个符号所占的大小都是一样的。对于这种段，sh_entsize 表示每个项的大小。如果为 0，则表示该段不包含固定大小的项

            - 结果
              - Name — 段的名称
              - Type — 段的类型
              - Addr — 段虚拟地址
                - 如果该段可以被加载 — 在进程虚拟空间中的地址
                - 否则 0x00
              - off — 段偏移
              - Size — 段的长度
              - Flg — 段的标志位
              - LK 和 Inf — 内涵
        - 字符串表(String Table)
          - 内涵
            - 集中存放变量名、等的表
            - 字符串被集中放在一个表中

偏移	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
+0	\0	h	e	l	l	o	w	o	r	l
+10	d	\0	M	y	v	a	r	i	a	b
+20	l	e	\0							

            - 要引用的话通过数字下标

偏移	字符串
0	空字符串
1	helloworld
6	world
12	Myvariable

          - 名称 — .strtab
        - 段表字符串表 (Section Header String Table)
          - 内涵
            - 集中存放段的名称的表
            - 字符串被集中放在一个表中
            - 要引用的话通过数字下标
          - 名称 — .shstrtab
        - 符号表 (Symbol Table)
          - 符号 (Symbol)
            - 内涵
              - 函数名
              - 变量名
              - ......
            - 实例
              - 全局符号
                - Internal Symbol
                  - 内涵 — 定义在本文件中的 全局函数或全局变量
                  - 特性 — 可以被其他 文件引用
                  - 符号值
                    - 函数对应的第一行机器码的addr — 一般在 .text 中
                    - 变量的值 存储的位置 addr — 一般在 .data、.rodata、.bss 中
                - External Symbol
                  - 内涵 — 在本文件中引用的，在别的文件中定义的全局函数，或全局变量
                  - 符号值
                    - 函数对应的第一行机器码的addr — 一般在 .text 中
                    - 变量的值 存储的位置 addr — 一般在 .data、.rodata、.bss 中
              - 局部符号 — 特性
                - 只在 编译单元内部可见 — 对于链接过程没有作用
                - 可以用来分析程序、分析核心转储文件
              - 段名 — 符号值 — 该段的addr
              - 行号信息
                - 内涵 — 机器代码与源代码的对应
                - 特性 — 可选
              - 文件名
              - COMMON 块类型的符号 — 特性 — st_shndx 为 SHN_COMMON
              - 未定义的符号 — 特性 — st_shndx 为 SHN_UNDEF
              - 特殊符号
                - 内涵 — 链接器在链接的过程中定义的符号
                - 特性 — 虽然不是在程序中定义的，但是可以声明并引用

```
$ cat e.c
#include<stdio.h>
extern char __executable_start[];
int main(int argc, char const *argv[])
{
    printf("Execuable start >>> %x\n",__executable_start);
    return 0;
}
$ ./e
Execuable start >>> 6c964000
```

                  - 数据类型为 void*
                - 实例
                  - __executable_start — 内涵 — 程序的起始地址
                  - __etext
                    - 内涵 — 代码段的介绍地址
                    - 又称
                      - _etext
                      - etext
                  - _edata
                    - 又称 — edata
                    - 内涵 — 数据段的结束地址
                  - _end
                    - 又称 — end
                    - 内涵 — 程序结束地址
            - 符号修饰(Name Decoration)
              - 又称 — 符号改编 (Name Mangling)
              - 发生条件 — 编译 cpp
              - 内涵
                - 函数名 — 根据 函数的函数签名生成 特定的符号，即修饰后符号名(Decorated Name)，以表示函数
                  - gcc
                    - 规则
                      - ① 以 _Z 开头
                      - 是否是 某个命名空间或 类中的函数
                        - 是
                          - ② 跟个N
                          - ③ <func name length><func name>E<param type list>
                        - 否 — ② <func name length><func name><param type list>
                      - param type
                        - int — i
                        - float — f
                        - char — c
                        - double — d
                        - long — l
                        - short — s
                        - 空 — v
                  - Visual C++
                - 变量名 — gcc — 规则 — 没有命名空间没有类 — _ZZ<func name length><func name><param type list>E<variable name length><variable name>
            - 强符号 和 弱符号
              - 问题 — 符号重复定义 — 一般是全局变量名 或 函数名 才会造成冲突
              - 强符号 (Strong Symbol)
                - 内涵
                  - 函数名
                  - 初始化的全局变量
                - 特性
                  - 链接器不允许强符号被多次定义 — 不同目标文件中不能有同名的强符号 — 否则报错
                  - 引用的时候，优先引用强符号
              - 弱符号 (Weak Symbol)
                - 内涵 — 未初始化的全局变量
                - 特性 — 引用的时候，如果引用的符号是弱符号，个数又有多个 — 优先引用占用空间最大的一个 (Common Block 机制)
            - 强引用 額 弱引用
          - 内涵 — 存放 全局符号 和 局部符号的信息
          - 名称 — .symtab
          - 结构 — Elf32_Sym 的数组

```
typedef struct
{
  Elf32_Word     st_name;    /* Symbol name (string tbl index) */
  Elf32_Addr     st_value;   /* Symbol value */
  Elf32_Word     st_size;    /* Symbol size */
  unsigned char  st_info;    /* Symbol type and binding */
  unsigned char  st_other;   /* Symbol visibility */
  Elf32_Section  st_shndx;   /* Section index */
} Elf32_Sym;
```

成员	含义
st_name	符号名。这个成员包含了该符号名在字符串表中的下标（还记得字符串表吧？）
st_value	符号相对应的值。这个值跟符号有关，可能是一个绝对值，也可能是一个地址等，不同的符号，它所对应的值含义不同，见下文 "符号值"
st_size	符号大小。对于包含数据的符号，这个值是该数据类型的大小。比如一个 double 型的符号它占用 8 个字节。如果该值为 0，则表示该符号大小为 0 或未知
st_info	符号类型和绑定信息，见下文 "符号类型与绑定信息"
st_other	该成员目前为 0，没用
st_shndx	符号所在的段，见下文 "符号所在段"

            - 符号类型和绑定信息 (st_info) — 共32位，低4位表示符号类型，高28位表示符号绑定的信息
            - 符号所在段 (st_shndx)
              - 如果该符号是 Internal Symbol — 此处表示该符号所在段在 段表中的下标
              - 特殊

宏定义名	值	说明
SHN_ABS	0xfff1	表示该符号包含了一个绝对的值。比如表示文件名的符号就属于这种类型的
SHN_COMMON	0xfff2	表示该符号是一个 "COMMON 块" 类型的符号，一般来说，未初始化的全局符号定义就是这种类型的，比如 SimpleSection.o 里面的 global_uninit_var。有关 "COMMON" 详见 "深入静态链接" 之 "COMMON 块"
SHN_UNDEF	0	表示该符号未定义。这个符号表示该符号在本目标文件被引用到，但是定义在其他目标文件中

                - SHN_COMMON
                - SHN_ABS
                - SHN_UNDEF
            - 符号值 (st_value)
              - 如果 不是 COMMON 块类型 — 该符号在该符号所处段中的偏移
              - 如果是 COMMON 类型 — 表示该符号的对齐属性
              - 可执行文件中 — 表示该符号的虚拟地址
        - 重定位表 (Relocation Table)
          - 名称 — .rel.<section name>
            - .rel.text
            - .rel.data
          - 段类型 — SHT_REL
          - 特性
            - sh_link 表示符号表的下标
            - sh_info 表示它作用于哪个段
            - 所有 可重定位文件 都包含重定位表
            - 每个有需要重定位内容的段，都有一个对应的重定位表 — .text的重定位表，名为 .rel.text，.data的 名为 .rel.data
          - 结构 — Elf32_Rel 的数组

```
typedef struct
{
  Elf32_Addr   r_offset;   /* Address */
  Elf32_Word   r_info;     /* Relocation type and symbol index */
} Elf32_Rel;
```

成员	含义
r_offset	重定位入口的偏移。对于可重定位文件来说，这个值是该要修正的位置的第一个字节相对于段起始的偏移；对于可执行文件或共享对象文件来说，这个值是该要修正的位置的第一个字节的虚拟地址。我们这里只关心可重定位文件的情况，可执行文件或共享对象文件的情况，将在下一章 "动态链接" 再作分析
r_info	重定位入口的类型和符号。这个成员的低 8 位表示重定位入口的类型，高 24 位表示重定位入口的符号在符号表中的下标。因为各种处理器的指令格式不一样，所以重定位所修正的指令地址格式也不一样，每种处理器都有自己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，请参考 "动态链接" 一章

          - 查看
            - 命令 — objdump -r <file>
            - 结果
              - OFFSET — 重定位入口 (Relocation Entry) — 每个要被重定位的地方
                - 重定位入口的偏移 — 即表示要修改的地方在所处段的位置
              - TYPE — 重定位入口的类型 — x86 基本类型

x86 基本重定位类型

宏定义	值	重定位修正方法
R_386_32	1	绝对寻址修正 S + A
R_386_PC32	2	相对寻址修正 S + A − P

A = 保存在被修正位置的值
P = 被修正的位置（相对于段开始的偏移量或者虚拟地址），注意，该值可通过 r_offset 计算得到
S = 符号的实际地址，即由 r_info 的高 24 位指定的符号的实际地址

                - R_386_PC32 — 内涵
                  - 相对位移调用指令
                  - 相对寻址修正
                  - 真正填入的值 是 S+A-P — 即表示下一条指令与符号实际地址的偏差
                - R_386_32 — 内涵
                  - 真正填入的就是符号值本身
                  - 绝对寻址修正
              - VALUE — 重定位入口要填入符号值的符号