目 录

1 实现引导扇区

一个操作系统从开机到开始运行，需要经历“引导，加载内核进入内存，跳入保护模式，开始执行内核”。操作系统使用 Loader 模块来加载内核进入内存，并跳入保护模式。引导扇区负责将 Loader 加载入内存。

1.1 引导扇区格式

引导扇区是软盘的第 0 个扇区。我们把 Loader 模块复制到软盘上，然后引导扇区将找到并加载它。

引导扇区开头有一个很重要的数据结构，叫做 BPB。这个 BPB 数据结构使得软盘被操作系统识别。加上 BPB 数据结构之后，引导扇区的格式如下面的代码所示：

```
    jmp short LABEL_START
    nop

    BS_OEMName DB 'ForrestY' ; 生厂商名字
    BPB_BytsPerSec DW 512 ; 每扇区字节数
    BPB_SecPerClus DB 1 ; 每簇多少扇区
    BPB_RsvdSecCnt DB 1 ; Boot记录占用多少扇区
    BPB_NumFATS DB 2 ; 共有多少FAT表
    BPB_RootEntCnt DW 224 ; 根目录文件数最大值
    BPB_TotSec16 DW 2880 ; 逻辑扇区总数
    BPB_Media DB 0xF0 ; 媒体描述符
    BPB_FATSz16 DW 9 ; 每FAT扇区数
    BPB_SecPerTrk DW 18 ; 每磁道扇区数
    BPB_NumHeads DW 2 ; 磁头数
    BPB_HiddSec DD 0 ; 隐藏扇区数
    BPB_TotSec32 DD 0 ; 记录扇区数
    BS_DrvNum DB 0 ; 中断13的驱动器号
    BS_Reservedl DB 0 ; 未使用
    BS_BootSig DB 29h ; 扩展引导标记
    BS_VolID DD 0 ; 卷序列号
    BS_VolLab DB 'pengsida001' ; 卷标，必须11个字节
    BS_FileSysType DB 'FAT12' ; 文件系统类型

LABEL_START:
    ; ...
```

1.2 加载 Loader 进入内存

为了加载 Loader 文件，我们首先需要知道 Loader 模块所在的位置。我们假设 Loader 模块存放在根目录中，而根目录信息存放在根目录区中。根目录区从第 19 个扇区开始，由 BPB_RootEntCnt 个目录条目组成。

目录条目占用 32 字节，它的格式如下：

名称	偏移	长度	描述
DIR_Name	0	0xB	文件名8字节，扩展名3字节
DIR_Attr	0xB	1	文件属性
保留位	0xC	10	保留位
DIR_WrtTime	0x16	2	最后一次写入时间
DIR_WrtDate	0x18	2	最后一次写入日期
DIR_FstClus	0x1A	2	此条目对应的开始簇号
DIR_FileSize	0x1C	4	文件大小

所以，我们只要 Loader 模块的目录条目，就可以根据 DIR_FstClus 的值找到 Loader 模块。

1.2.1 寻找 Loader 的目录条目

我们通过遍历根目录区来寻找 Loader 模块目录条目，代码如下：

```
    ; 根目录的第一个扇区号是19
    SectorNoOfRootDirectory equ 19
    ; 数据缓冲区的基地址
    BaseOfLoader equ 09000h
    ; 数据缓冲区的偏移地址
    OffsetOfLoader equ 0100h
    ; Loader模块的名字
    LoaderFileName db "LOADER BIN", 0

    ; wSectorNo地址单元存放着要读取的扇区号
    mov word [wSectorNo], SectorNoOfRootDirectory
LABEL_SEARCH_IN_ROOT_DIR_BEGIN:
    ; wRootDirSizeForLoop地址单元存放着扇区数
    cmp word [wRootDirSizeForLoop], 0
    jz LABEL_NO_LOADERBIN
    ; 每遍历一次，扇区数减一
    dec word [wRootDirSizeForLoop]
    ; 设置es:bx，指定数据缓冲区的地址
    mov ax, BaseOfLoader
    mov es, ax
    mov bx, OffsetOfLoader
    ; 设置要读的扇区号
    mov ax, [wSectorNo]
    ; 设置要读的扇区数
    mov cl, 1
    ; 读取一个扇区的内容
    call ReadSector

    mov si, LoaderFileName
    mov di, OffsetOfLoader
    cld
    ; dx代表着接下来的循环次数
    ; 一个扇区512字节，一个根目录条目32字节，所以需要循环16次
    mov dx, 10h
LABEL_SEARCH_FOR_LOADERBIN:
    cmp dx, 0
    jz LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR
```

```
    dec dx
    ; 比较文件名称，文件名称为11个字节，所以比较11次
    mov cx, 11
LABEL_CMP_FILENAME:
    cmp cx, 0
    jz LABEL_FILENAME_FOUND
    dec cx
    ; 将LoaderFileName处的字节读入al
    lodsb
    ; 比较数据缓冲区中的字节
    cmp al, byte [es:di]
    jz LABEL_GO_ON
    jmp LABEL_DIFFERENT
LABEL_GO_ON:
    inc di
    jmp LABEL_CMP_FILENAME
LABEL_DIFFERENT:
    ; 让di指向下一个条目
    ; 每个目录条目为32字节
    and di, 0FFE0h
    add di, 20h
    mov si, LoaderFileName
    jmp LABEL_SEARCH_FOR_LOADERBIN
LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR:
    ; 读取下一个扇区号
    add word [wSectorNo], 1
    jmp LABEL_SEARCH_IN_ROOT_DIR_BEGIN
LABEL_NO_LOADERBIN:
    mov dh, 2
    call DispStr
    jmp $
LABEL_FILENAME_FOUND:
    jmp $
```

上述代码中，用到了读取一个扇区内容的函数 ReadSector。需要读软盘的时候，要用到 BIOS 中断 int 13h。

当 ah=00h 时，int 13h 用于复位软驱，此时使用 dl 指定驱动器号。

当 ah=02h 时，int 13h 用于从磁盘将数据读入 es:bx 指向的缓冲区中。当读取错误时，CF 会被置一。此时需要设置如下的寄存器值：

```
    al = 要读扇区数
    ch = 磁道号
    cl = 起始扇区号
    dh = 磁头号
    dl = 驱动器号
    es:bx指定数据缓冲区
```

在代码中添加一个 ReadSector 函数，用于读取软盘。

ReadSector 函数将 al 和 cl 作为参数，al 是相对扇区号，cl 是要读取的扇区数。在函数中，程序根据相对扇区号获得磁道号、起始扇区号和磁头号。软盘中一个磁道有 18 个扇区，于是将相对扇区号除以 18，得到的商和余数分别是总磁道号和起始扇区号。软盘

中，因为有两面，所以分别要磁头号 0 和磁头号 1 标记。现在还需要确定的是在哪个磁头号的第几个磁道号。软盘结构中，不是先排完 0 磁头号再排 1 磁头号的，而是交错排列。总磁头号为偶数的位于磁头号 0，总磁头号为奇数的位于磁头号 1。所以只要判断总磁头号的奇偶就能得到磁头号。将总磁头号与 1 相与，为 0 的话磁头号就是 0，为 1 的话磁头号就是 1。然后将总磁头号除以 2，就能得到相对于磁头的起始磁道号。读取软盘扇区的代码如下；

```
ReadSector:
    push bp
    mov bp, sp
    sub esp, 2
    ; 处理int 13h所需要的参数
    ; cl存放着要读取的扇区数
    mov byte [bp-2], cl
    push bx
    ; bl存放着每个磁道上的扇区数
    mov bl, [BPB_SecPerTrk]
    ; ax/bl，商放在al中，余数放在ah中
    div bl
    ; 得到当前磁道中的起始扇区号
    inc ah
    ; 设置cl的值为起始扇区号
    mov cl, ah
    mov dh, al
    shr al, 1
    ; 设置ch的值为磁道号
    mov ch, al
    ; 设置dh的值为磁头号
    and dh, 1
    pop bx
    ; 设置dl的值为驱动器号
    mov dl, [BS_DrvNum]
.GoOnReading:
    mov ah, 2
    mov al, byte [bp-2]
    int 13h
    jc .GoOnReading

    add esp, 2
    pop bp
    ret
```

1.2.2 寻找 FAT 项

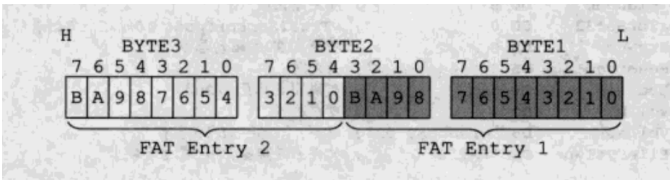
现在我们得到 Loader 模块的目录条目了，也就能得到 Loader 模块对应的开始簇号，也就能够得到 Loader.bin 的起始扇区号。通过这个扇区号，我们可以做到两件事：

- 把起始扇区装入内存。
- 通过这个扇区号找到 FAT 中的项，从而找到 Loader 占用的其他所有扇区。

在此先介绍一下软盘上的文件系统 FAT12，它由引导扇区、两个 FAT 表、根目录区和数据区组成，格式如下图:

我们之前获得的根目录条目就存放在根目录区中。现在我们拥有 Loader 模块的簇号，需要根据 FAT 表来找到 Loader 的所有簇。FAT 表由 FAT 项组成，每个 FAT 项长为 12 字节，FAT 项的值代表文件下一个簇号。如果 FAT 项的值大等于 0xFF8，那么代表当前簇是本文件的最后一个簇。如果 FAT 项的值为 0xFF7，表示它是一个坏簇。所以我们只要拥有起始簇号 n，就对应着 FAT 表中第 n 个 FAT 项，从而就能找到找到文件的所有簇号，也就找到了文件所占用的所有扇区。

FAT 项的格式如下:



实现代码如下:

```
1    bOdd db 0
2    SectorNoOfFAT1 equ 1
```

```
3   GetFATEntry:
4       push es
5       push bx
6       ; ax存放着起始扇区号，对应着第[ax]个FAT项
7       push ax
8       ; Loader模块在内存中的起始地址
9       mov ax, BaseOfLoader
10      ; 在Loader模块后面留出4k空间用于存放FAT项，作为数据缓冲区
11      sub ax, 0100h
12      mov es, ax
13      ; 恢复ax的值
14      pop ax
15      ; bOdd用于判断FAT项从第0位开始还是从第4位开始
16      mov byte [bOdd], 0
17      ; FAT项占用1.5个字节，所以ax先乘以3，再除以2
18      mov bx, 3
19      mul bx
20      mov bx, 2
21      ; 商放在ax中，余数放在dx中
22      div bx
23      ; 判断FAT项从第0位开始还是从第4位开始
24      ; ax为奇数时，FAT项从第4位开始。ax为偶数时，FAT项从第0位开始
25      cmp dx, 0
26      jz LABEL_EVEN
27      mov byte [bOdd], 1
28  LABEL_EVEN:
29      xor dx, dx
30      ; BPB_BytsPerSec是每个扇区占用的字节数
31      mov bx, [BPB_BytsPerSec]
32      ; ax存放着FAT项相对于FAT的扇区号，bx存放着FAT项在扇区中的偏移
33      div bx
34      push dx
35      mov bx, 0
36      ; ax加上FAT的扇区号，当前值为FAT项所在的扇区号
37      add ax, SectorNoOfFAT1
38      ; cl存放着要读取的扇区数
39      mov cl, 2
40      ; 读取FAT项所在的扇区，为了防止FAT项跨越两个扇区，所以一次读取两个扇区
41      call ReadSector
42      pop dx
43      add bx, dx
44      ; es是数据缓冲区基地址，bx是FAT项在扇区中的偏移
45      ; 现在ax中存放着FAT项的值
46      mov ax, [es:bx]
47      cmp byte [bOdd], 1
48      jnz LABEL_EVEN_2
49      ; 如果FAT从第4位开始，就将ax右移4位
50      shr ax, 4
51  LABEL_EVEN_2:
52      ; 如果FAT从第0位开始，就只保留ax的低12位
53      and ax, 0FFFh
54  LABEL_GET_FAT_ENTRY_OK:
55      pop bx
56      pop es
57      ret
```

1.2.3 加载 Loader

现在我们可以通过遍历根目录区来找到 Loader 模块对应的根目录条目。从根目录条目中找到相应的簇号，然后根据 FAT 表中的 FAT 项找到文件的下一个簇号。这里的簇号是相对于数据区的簇号。为了获得整个软盘中的簇号，需要加上根目录区的起始簇号，在加上根目录区簇的数量。因为数据区的簇号是从 2 开始的，所以还要减去 2。最后根据实际的簇号得到 Loader 模块所在的扇区号，然后将扇区号作为 ReadSector 函数的参数，读取相应扇区的数据。实现代码如下：

```
LABEL_FILENAME_FOUND:
    mov ax, RootDirSectors
    and di, 0FFE0h
    add di, 01Ah
    mov cx, word [es:di]
    push cx
    add cx, ax
    add cx, DeltaSectorNo
    mov ax, BaseOfLoader
    mov es, ax
    mov bx, OffsetOfLoader
    mov ax, cx
LABEL_GOON_LOADING_FILE:
    mov cl, 1
    call ReadSector
    pop ax
    call GetFATEntry
    ; 检查FAT项的值是否是0FFFh
    cmp ax, 0FFFh
    jz LABEL_FILE_LOADED
    push ax
    ; RootDirSectors是根目录扇区数
    mov dx, RootDirSectors
    add ax, dx
    ; 根目录开始扇区号为19，数据区第一个簇的簇号是2
    ; 为了正确求得FAT项对应的簇号，定义了DeltaSectorNo equ 17
    ; FAT项对应的簇号 = RootDirSectors + DeltaSectorNo + 起始簇号
    add ax, DeltaSectorNo
    ; es:bx指向数据缓冲区
    ; 读取一个扇区结束后，bx的值加512字节
    add bx, [BPB_BytsPerSec]
    jmp LABEL_GOON_LOADING_FILE
```

1.2.4 执行 Loader 模块的代码

前几个小结将 Loader 模块加载进了内存，放在 BaseOfLoader:OffsetOfLoader 处。现在只要将程序跳转到数据缓冲区的地址，就可以开始执行 Loader 模块的代码。实现代码如下：

```
jmp BaseOfLoader:OffsetOfLoader
```

回想本节一开始说的，一个操作系统从开机到开始运行，需要经历“引导，加载内核进入内存，跳入保护模式，开始执行内核”。现在我们只做到了引导，也就是将 Loader 模块加载进内存。Loader 要做的事情还有两件：

- 加载内核入内存。
- 跳入保护模式。

1.3 引导扇区完整的实现代码

前面小节虽然有贴代码，但是都是一些细节上的代码。现在感受一下完整的实现代码，以此对引导扇区整个的实现框架有清楚的认识。在阅读代码之前，先了解一下软盘上 FAT12 文件系统引导扇区的格式，从而对代码框架有更全面的认识。FAT12 引导扇区格式如下所示：

名称	偏移	长度	内容	Orange'S的值
BS_jmpBoot	0	3	一个短跳转指令	jmp LABEL_START nop
BS_OEMName	3	8	厂商名	'ForrestY'
BPB_BytsPerSec	11	2	每扇区字节数	0x200
BPB_SecPerClus	13	1	每簇扇区数	0x1
BPB_RsvdSecCnt	14	2	Boot 记录占用多少扇区	0x1
BPB_NumFATs	16	1	共有多少 FAT 表	0x2
BPB_RootEntCnt	17	2	根目录文件数最大值	0xE0
BPB_TotSec16	19	2	扇区总数	0xB40
BPB_Media	21	1	介质描述符	0xF0
BPB_FATSz16	22	2	每 FAT 扇区数	0x9
BPB_SecPerTrk	24	2	每磁道扇区数	0x12
BPB_NumHeads	26	2	磁头数（面数）	0x2
BPB_HiddSec	28	4	隐藏扇区数	0
BPB_TotSec32	32	4	如果BPB_TotSec16是0，由这个值记录扇区数	0
BS_DrvNum	36	1	中断 13 的驱动器号	0
BS_Reserved1	37	1	未使用	0
BS_BootSig	38	1	扩展引导标记（29h）	0x29
BS_VolID	39	4	卷序列号	0
BS_VolLab	43	11	卷标	'OrangeS0.02'
BS_FileSysType	54	8	文件系统类型	'FAT12'
引导代码及其他	62	448	引导代码、数据及其他填充字符等	引导代码（剩余空间被0填充）
结束标志	510	2	0xAA55	0xAA55

引导扇区实现代码如下：

```
org 07c00h

BaseOfStack equ 07c00h
BaseOfLoader equ 09000h
OffsetOfLoader equ 0100h

RootDirSectors equ 14 ; 根目录区占用了14扇区
SectorNoOfRootDirectory equ 19
SectorNoOfFAT1 equ 1
DeltaSectorNo equ 17

; FAT12引导扇区固有的头信息
; FAT12引导扇区格式已经在本节开头说明

; BS_jmpBoot，长度要求为3字节
; 因为jmp short LABEL_START指令是2字节，所以需要加个nop指令，使得长度为3字节
jmp short LABEL_START
nop

BS_OEM DB 'ForrestY'
BPB_BytsPerSec DW 512
BPB_SecPerClus DB 1
BPB_RsvdSecCnt DW 1
BPB_NumFATs DB 2
BPB_RootEntCnt DW 224
BPB_TotSec16 DW 2880
BPB_Media DB 0xF0
BPB_FATSz16 DW 9
BPB_SecPerTrk DW 18
BPB_NumHeads DW 2
BPB_HiddSec DD 0
BPB_TotSec32 DD 0
BS_DrvNum DB 0
BS_Reservd1 DB 0
BS_BootSig DB 29h
BS_VolID DD 0
BS_VolLab DB 'pengsida001'
BS_FileSysType DB 'FAT12   '

LABEL_START:
    ; 给每个段寄存器赋初值为代码段基址
    mov ax, cs
    mov ds, ax
    mov es, ax
    mov ss, ax
    ; 给堆栈指针赋初值，指向栈顶
    mov sp, BaseOfStack

    ;mov ax, 0600h
    ;mov bx, 0700h
    ;mov cx, 0
    ;mov dx, 0184fh
    ;int 10h

```

```
55      ; mov dh, 0
56      ; call DispStr
57
58      ; ah=00h时，int 13h的功能是复位软驱。使用dl来指定驱动器号
59      xor ah, ah
60      xor dl, dl
61      int 13h
62
63      ; 根目录区起始扇区号SectorNoOfRootDirector为19
64      ; 因为是要搜索根目录区中的根目录条目，所以要从头遍历
65      ; wSectorNo地址单元中存放着要读取的扇区号
66      mov word [wSectorNo], SectorNoOfRootDirectory
67  LABEL_SEARCH_IN_ROOT_DIR_BEGIN:
68      ; 检查是否已经将根目录区中的所有扇区都遍历完
69      cmp word [wRootDirSizeForLoop], 0
70      jz LABEL_NO_LOADERBIN
71      ; 将循环数减一
72      dec word [wRootDirSizeForLoop]
73      ; int 13h将扇区内容读入es:bx指定的数据缓冲区
74      mov ax, BaseOfLoader
75      mov es, ax
76      mov bx, OffsetOfLoader
77      ; ReadSector将ax和cl中的值作为参数
78      ; ReadSector将从第ax个扇区开始的cl个扇区读入ex:bx指定的数据缓冲区中
79      mov ax, [wSectorNo]
80      mov cl, 1
81      call ReadSector
82
83      ; ds存放着代码段段基址，si为LoaderFileName的偏移地址，ds:si就指向了代码段中定义
            的"LOADER  BIN"字符串
84      mov si, LoaderFileName
85      ; es存放着数据缓冲区的基地址，di为存放扇区数据处的偏移地址，es:di就指向了扇区数
            据
86      mov di, OffsetOfLoader
87      cld ; 从低位到高位
88      mov dx, 10h ; 一个扇区共有512字节，一个根目录条目为32字节，所以遍历一个扇区需要
            16次
89  LABEL_SEARCH_FOR_LOADERBIN:
90      cmp dx, 0 ; 检查循环是否结束
91      jz LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR
92      dec dx
93      ; 文件名为11个字节。只要检查根目录条目前11个字节与文件名字符串是否相等就行了
94      mov cx, 11
95  LABEL_CMP_FILENAME:
96      cmp cx, 0 ; 检查循环是否结束
97      jz LABEL_FILENAME_FOUND
98      dec cx
99      lodsb ; 将ds:si指向的字节读入al，且将si的值自动加一
100     cmp al, byte [es:di] ; 比较两个地址处的字节是否相同
101     jz LABEL_GO_ON
102     jmp LABEL_DIFFERENT
103 LABEL_GO_ON:
104     inc di ; 将di的值加一，指向下一个字节
105     jmp LABEL_CMP_FILENAME
106 LABEL_DIFFERENT:
107     ; 如果文件名中11个字节有一个不相等，就跳过这个根目录条目
```

```
108     ; 这时候需要处理di和si
109     ; di最多加11，这时候di也只是改变低4位。所以让di的低4位与0000b相与
110     ; 因为根目录条目是32字节，所以di的值肯定是32的倍数，di的第4位肯定是0
111     ; 让di的第4位到第7位与1110b相与，让di的第4位变为0，让di指向本条目开头
112     ; 其实，因为di最多改变低4位，所以and di, 0FFF0h也能达到相同的效果
113     and di, 0FFE0h
114     ; di加32，指向下一个根目录条目
115     add di, 20h
116     ; 让si重新指向文件名字符串的开头
117     mov si, LoaderFileName
118     jmp LABEL_SEARCH_FOR_LOADERBIN
119 LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR:
120     ; 当前扇区没有Loader模块，需要搜索下一个扇区的内容
121     ; 将要读取的扇区号加一
122     add word [wSectorNo], 1
123     jmp LABEL_SEARCH_IN_ROOT_DIR_BEGIN
124 LABEL_NO_LOADERBIN:
125     jmp $ ; 找不到Loader模块，程序陷入死循环
126 LABEL_FILENAME_FOUND:
127     mov ax, RootDirSectors
128     ; 让di指向根目录条目开头
129     and di, 0FFE0h
130     ; 让di指向此条目对应的开始簇号
131     add di, 01Ah
132     ; 将开始簇号存储在cx中
133     mov cx, word [es:di]
134     push cx ; 存储扇区在FAT表中的序号
135     ; 一个簇有一个扇区，所以簇号就是扇区号。cx目前存放着扇区号
136     ; 让cx与根目录区扇区数相加，并将结果存储在cx中
137     ; 根目录区起始扇区号为19，数据区的第一个簇的簇号是2
138     ; 所以cx加上19再减去2，就得到了相对扇区号
139     add cx, ax
140     add cx, DeltaSectorNo ; DeltaSectorNo equ 19
141     mov ax, BaseOfLoader
142     mov es, ax
143     mov bx, OffsetOfLoader
144     ; ax作为ReadSector函数的参数，存放着相对扇区号
145     mov ax, cx
146 LABEL_GOON_LOADING_FILE:
147     ; cl作为ReadSector函数的参数，存放着要读取的扇区数
148     mov cl, 1
149     call ReadSector
150     ; 将扇区在FAT表中的项号存储在ax中
151     pop ax
152     ; GetFATEntry返回后，ax存放着Loader模块下一个簇号
153     call GetFATEntry
154     ; 判断ax是否为0FFFh，如果是，说明当前扇区是Loader模块最后一个扇区
155     cmp ax, 0FFFh
156     jz LABEL_FILE_LOADED
157     push ax
158     mov dx, RootDirSectors
159     add ax, dx
160     add ax, DeltaSectorNo
161     ; 让bx指向数据缓冲区的下一个512个字节的开头，用于存放Loader模块的下一个扇区
162     add bx, [BPB_BytsPerSec]
163     jmp LABEL_GOON_LOADING_FILE
```

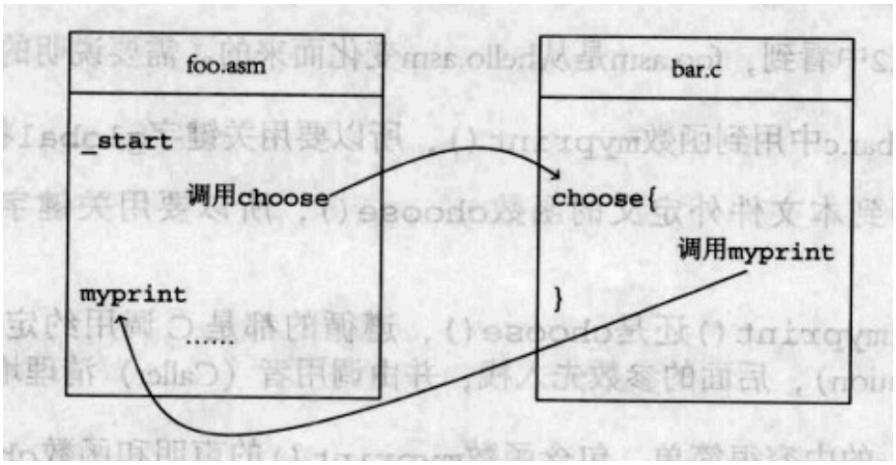
```
164 LABEL_FILE_LOADED:
165     ; 开始执行Loader模块的代码
166     jmp BaseOfLoader:OffsetOfLoader
167 
168 wRootDirSizeForLoop dw RootDirSectors ; 根目录区中的扇区数
169 wSectorNo dw 0 ; 用于存放要读取的扇区号
170 bOdd db 0 ; 判断FAT项是从字节中的第0位开始还是从第4位开始的
171 
172 LoaderFileName db "LOADER  BIN", 0 ; 文件名一定是11字节
173 
174 ReadSector:
175     push bp
176     mov bp, sp
177     sub esp, 2
178     ; bp-2地址单元中存放着要读取的扇区数
179     mov byte [bp-2], cl
180     push bx
181     ; BPB_SecPerTrk是每磁道的扇区数
182     mov bl, [BPB_SecPerTrk]
183     ; ax中存放着要读取的扇区号
184     ; ax/bl的商扇区所在的磁道号，存放在al中
185     ; ax/bl是要读取的扇区相对于当前磁道的起始扇区号，存放在ah中
186     div bl
187     ; 磁道的扇区号从1开始，所以要将ah的值加一
188     int ah
189     ; 根据int 13h的要求，cl要存放相对于磁道的起始扇区号
190     ; dh存放磁头号。软盘中的磁头号不是0就是1。
191     ; 软盘的排列并不是按照我们所想象的"把0面先排完了再开始排1面“，而是交替排列的
192     ; 偶数的磁道号的磁头号为0，奇数的磁道号的磁头号为1
193     ; 所以将al中存放着总磁道号与1相与就可以得到磁头号
194     ; ch存放磁道号。原先al存放着软盘总的磁道号，而软盘有两面，所以需要除以2，得到相
            对于当前磁头的磁道号
195     mov cl, ah
196     mov dh, al
197     and dh, 1
198     shr al, 1
199     mov ch, al
200     pop bx
201     mov dl, [BS_DrvNum]
202 .GoOnReading:
203     ; ah指定int 13h的工作模式
204     mov ah, 2
205     ; al存放着要读取的扇区数
206     mov al, byte [bp-2]
207     int 13h
208     jc .GoOnReading ; 如果读取错误CF会被置为1，这里程序会重读，直到正确为止
209 
210     add esp, 2
211     pop bp
212     ret
213 
214 ; 将ax作为输入参数，指定FAT表中FAT项的序号
215 ; 将ax作为返回参数，将文件下一个簇号放在ax中
216 GetFATEntry:
217     push es
218     push bx
```

```
219     push ax
220     mov BaseOfLoader
221     sub ax, 0100h ; 在BaseOfLoader之前空出4k的空间用于存放FAT项所在的扇区
222     mov es, ax
223     pop ax
224     mov byte [bOdd], 0
225     mov bx, 3
226     mul bx
227     mov bx, 2
228     div bx
229     cmp dx, 0
230     jz LABEL_EVEN
231     mov byte [bOdd], 1
232 LABEL_EVEN:
233     xor dx, dx
234     mov bx, [BPB_BytsPerSec]
235     div bx
236     push dx
237     mov bx, 0
238     add ax, SectorNoOfFAT1
239     mov cl, 2
240     call ReadSector
241
242     pop dx
243     add dx, dx
244     mov ax, [es:bx]
245     cmp byte [bOdd], 1
246     jnz LABEL_EVEN_2
247     shr ax, 4
248 LABEL_EVEN_2:
249     and ax, 0FFFh
250 LABEL_GET_FAT_ENTRY_OK:
251     pop bx
252     pop es
253     ret
```

2 汇编与 C 混合编程

因为编写内核的过程中，需要用到汇编与 C 的混合编程，所以在此通过一个例子去熟悉它。

程序入口 _start 在 foo.asm 中，一开始程序会调用 bar.c 中的函数 choose()，在 choose() 函数中又将调用 foo.asm 中的函数 myprint() 来打印字符串。整个过程如下图：



例子如下，先看 foo.asm 的代码：

```
1   ; foo.asm
2   extern choose
3
4   [section .data]
5
6   num1st dd 3
7   num2nd dd 4
8
9   [section .text]
10
11  global _start
12  global myprint
13
14  _start:
15      push dword [num2nd]
16      push dword [num1st]
17      call choose
18      add esp, 8
19      mov ebx, 0
20      mov eax, 1
21      int 0x80
22
23  myprint:
24      mov edx, [esp + 8]
25      mov ecx, [esp + 4]
26      mov ebx, 1
```

```
27          mov eax, 4
28          int 0x80
29          ret
```

在看 bar.c 的代码：

```
1   void myprint(char* msg, int len);
2
3   int choose(int a, int b)
4   {
5       if(a >= b)
6           myprint("the first one\n", 15);
7       else
8           myprint("the second one\n", 16);
9       return 0;
10  }
```

以上的例子有几点需要说明的：

- foo.asm 中一定要定义"_start"，而且还要用过 global 关键字将它导出，这样链接程序才能找到它，作为程序的入口点。
- 因为 bar.c 要用到 foo.asm 中的 myprint() 函数，所以需要用 global 关键字将其导出。
- 因为 foo.asm 要用到 bar.c 中的函数 choose()，所以要用 extern 关键字声明。

3 ELF 文件

内核在 linux 编译后是 ELF 格式。为了让 Loader 将内核装载进内存，我们还需要研究 ELF 格式。

iABI object 文件格式是 UNIX 系统实验室作为应用程序二进制借口 (ABI) 而开发和发布的，又被称为 ELF 文件格式。在 ELF 目标文件中有三种主要的类型，如下：

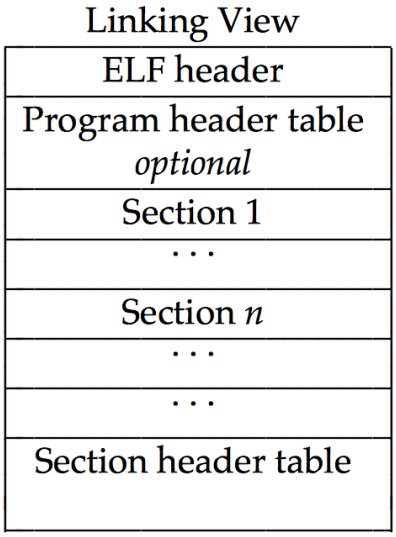
- 可重定位文件，保存着代码和数据，用来和其他的 object 文件一起来创建一个可执行文件或共享文件。
- 可执行文件，保存着用来执行的程序，该文件指出了 exec 如来创建程序进程映像。
- 共享文件，保存着代码和数据，用来被链接编辑器和动态链接器连接。链接编辑器可以将当前共享文件与其他的可重定位和共享文件链接，从而创建其他的目标文件。动态链接器可以将当前共享文件与一个可执行文件和其他的共享文件链接，从而创建一个进程映像。

ELF 目标文件由汇编器和链接编辑器创建，是程序的二进制表现形式，用于在处理器上直接运行。

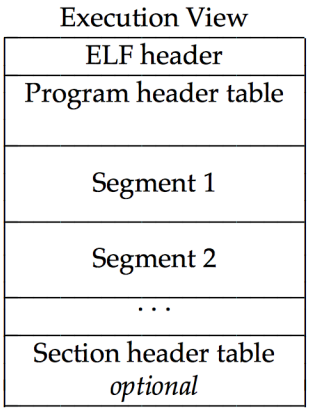
3.1 文件格式

object 文件参与了程序的链接和程序的运行。object 文件为这两个过程提供了不同的视角，因此链接过程中和执行过程中的 object 文件格式是不一样的。

以下是链接过程中 object 文件的格式：



以下是执行过程中 object 文件的格式:



可以看出，object 文件由 4 部分组成，如下:

- ELF header，它位于文件的开头，描述了该文件的结构。
- Program header table，用于告诉系统如何来创建一个进程的内存映像。执行过程中必须有这个程序头表。
- Sections，包含了指令、数据、符号表和重定位信息等。
- Section header table，包含了 section 的信息。每个 section 在这个表中有一个 entry，每个 entry 给出了相应 section 的名字、大小等信息。链接过程中必须有这个节头表。

3.1.1 数据表示

ELF 文件为了能够支持 8 位到 32 位不同架构的处理器，定义了一些与机器无关的数据类型:

```
1    // Elf32_Addr 无符号程序地址，4字节大小
2    // Elf32_Half 无符号中等大小整数，2字节大小
3    // Elf32_Off 无符号文件偏移，4字节大小
4    // Elf32_SWord 有符号大整数，4字节大小
5    // Elf32_Word 无符号大整数，4字节大小
6    // unsigned char 无符号小整数，1字节大小
```

3.2 ELF header

ELF 头的定义代码如下:

```
#define EI_NIDENT 16
typedef struct
{
    unsigned char e_ident[EI_NIDENT];
    Elf32_Half e_type;
    Elf32_Half e_machine;
    Elf32_Word e_version;
    Elf32_Addr e_entry;
    Elf32_Off e_phoff;
    Elf32_Off e_shoff;
    Elf32_Word e_flags;
    Elf32_Half e_ehsize;
    Elf32_Half e_phentsize;
    Elf32_Half e_phnum;
    Elf32_Half e_shentsize;
    Elf32_Half e_shnum;
    Elf32_Half e_shstrndx;
}Elf32_Ehdr;
```

ELF header 各个成员解释如下：

- e_ident, 为 16 字节的字符数组，头 4 个字节为“.ELF”，用于表明该文件是一个 ELF 文件。接下来的 12 个字节是一些与机器无关的信息。
- e_type, 用于确定该 object 文件的类型。下图是 e_type 的值与对应的文件类型。

Name	Value	Meaning
ET_NONE	0	No file type
ET_REL	1	Relocatable file
ET_EXEC	2	Executable file
ET_DYN	3	Shared object file
ET_CORE	4	Core file
ET_LOPROC	0xff00	Processor-specific
ET_HIPROC	0xffff	Processor-specific

- e_machine 用于表明该程序需要的体系结构。下图是 e_machine 的值与对应的文件类型。

Name	Value	Meaning
EM_NONE	0	No machine
EM_M32	1	AT&T WE 32100
EM_SPARC	2	SPARC
EM_386	3	Intel 80386
EM_68K	4	Motorola 68000
EM_88K	5	Motorola 88000
EM_860	7	Intel 80860
EM_MIPS	8	MIPS RS3000

- e_version 用于表明该 object 文件的版本。下图是 e_version 的值与对应的文件类型。

Name	Value	Meaning
EV_NONE	0	Invalid version
EV_CURRENT	1	Current version

- e_entry 为程序的入口地址。如果程序没有这个入口点，该成员的值将保持为 0。
- e_phoff 代表了程序头表在文件中的偏移量。如果 object 文件中没有程序头表，该成员的值将保持为 0。
- e_shoff 代表了节头表在文件中的偏移量。如果 object 文件中没有节头表，该成员的值将保持为 0。
- e_flags 存放着与 object 文件有关的处理器标志。
- e_ehsize 代表 ELF 头的大小。
- e_phentsize 代表程序头表中 entry 的大小。程序头表中每个 entry 的大小相同。
- e_phnum 代表程序头表中 entry 的数目。
- e_shentsize 代表节头表中每个条目的大小。
- e_shnum 代表节头表中条目的数目。
- e_shstrndx 代表着节头表中一个条目的序号，这个条目与 section 名字字符表有关。

3.2.1 ELF 鉴别

在前面的小节有提到，ELF 文件可以支持多种机器类型。为了支持这个特性，object 文件的头 16 个字节被用来说明如何解释该文件。这 16 个字节与处理器无关，和 object 文件剩下的内容也无关。ELF header 的 e_ident 成员是一个数组，用于存放这 16 个字节。接下来解释该数组中的各个序号以及相应的值：

- EI_MAG0 EI_MAG3 是 4 个宏，值分别为 0 3。数组中该序号相应的值如下所示：

Name	Value	Position
ELFMAG0	0x7f	e_ident[EI_MAG0]
ELFMAG1	'E'	e_ident[EI_MAG1]
ELFMAG2	'L'	e_ident[EI_MAG2]
ELFMAG3	'F'	e_ident[EI_MAG3]

这 4 个字节用于表示该文件格式为 ELF。

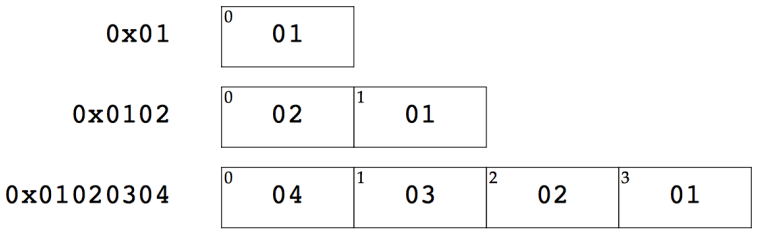
- 宏 EI_CLASS，值为 4，e_ident[EI_CLASS] 用于确定 object 文件的位数。值和相应的意义如下所示：

Name	Value	Meaning
ELFCLASSNONE	0	Invalid class
ELFCLASS32	1	32-bit objects
ELFCLASS64	2	64-bit objects

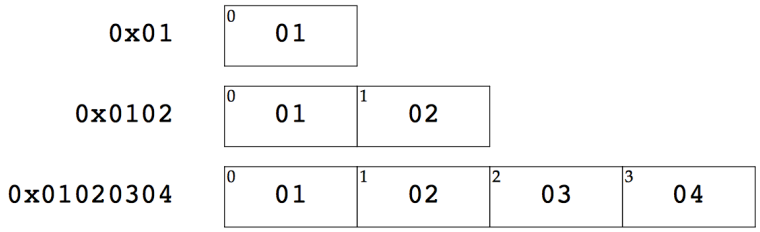
- 宏 EI_DATA，值为 5，e_ident[EI_DATA] 指定了 object 中特定处理器数据的编码方式。

  当 e_ident[EI_DATA] 值为 0 时，表明了无效的编码方式。

  当 e_ident[EI_DATA] 值为 1 时，表明了 ELFDATA2LSB 编码方式，值最小的字节占着最低的地址，如下所示：



  当 e_ident[EI_DATA] 值为 2 时，表明了 ELFDATA2MSB 编码方式，值最大的字节占着最低的地址，如下所示：



- 宏 EI_VERSION，值为 6，e_ident[EI_VERSION] 指定了 ELF header 的版本。
- 宏 EI_PAD，值为 7，用于标识 e_ident 未使用字节的开头。接下来 e_ident[7] e_ident[15] 都不会被用到，这些值都被置为 0。

3.3 Sections

object 文件中的 section header table 用于定位所有的 section。一个 section table header 的索引就是这个数组的下标。节头表的一些索引是比较特殊的，如下所示：

- SHN_UNDEF，值为 0，指向了没有定义的或无意义的 section。
- SHN_LORESERVE 和 SHN_LOPROC，值为 0xff00，是被保留的索引的最小值。
- SHN_HIPROC,值为 0xff1f,被保留的索引的最大值。从 SHN_LOPROC 到 SHN_HIPROC 的索引用于处理器特定的意义。
- SHN_ABS，值为 0xfff1，指向的 section 的符号是绝对地址。
- SHN_COMMON，值为 0xfff2，指向的 section 的符号是一个公共的符号。
- SHN_HIRESERVE，值为 0xffff，是被保留的索引的最大值。从 SHN_LORESERVE 到 SHN_HIRESERVE 的索引都是被保留的。

Seciton 的格式需要满足以下几点条件：

- object 文件中的每个 section 都有一个相应的 section header，用于描述 section。
- 每个 section 占据的空间都是连续的。
- sections 之间不可以相互重叠。

3.3.1 Section header

Section header 的定义代码如下：

```
typedef struct
{
    Elf32_Word sh_name;
    Elf32_Word sh_type;
    Elf32_Word sh_flags;
    Elf32_Addr sh_addr;
    Elf32_Off sh_offset;
    Elf32_Word sh_size;
    Elf32_Word sh_link;
    Elf32_Word sh_info;
    Elf32_Word sh_addralign;
    Elf32_Word sh_entsize;
}Elf32_Shdr;
```

Section header 的成员解释如下：

- sh_name，是 section 名字字符表的索引，用于查找当前 section 的名字。

- sh_type，将 section 按内容和意义分类。
- sh_flags，用于描述 section 的属性。
- sh_addr，用于指定 section 在内存的地址。如果 section 不会出现在进程的内存映像空间中，那么这个值为 0。
- sh_offset，用于表明该 section 相对于 object 开头的偏移。
- sh_size，表示了该 section 的大小。
- sh_link，该成员包含了该 section 在 section header table 中的索引链接。该索引链接的解释依靠于该 section 的类型。
- sh_info，它包含了多余的信息。这些信息的解释依赖于节的类型。
- sh_addralign，一些 sections 有地址对齐的约束。
- sh_entsize，一些 sections 保存着一张存放着 entry 的表，该成员值代表着 entry 的大小。如果 section 没有这张表，那么成员值为 0。

3.3.2 sh_type

sh_type 定义了下列宏：

Name	Value
SHT_NULL	0
SHT_PROGBITS	1
SHT_SYMTAB	2
SHT_STRTAB	3
SHT_RELA	4
SHT_HASH	5
SHT_DYNAMIC	6
SHT_NOTE	7
SHT_NOBITS	8
SHT_REL	9
SHT_SHLIB	10
SHT_DYNSYM	11
SHT_LOPROC	0x70000000
SHT_HIPROC	0x7fffffff
SHT_LOUSER	0x80000000
SHT_HIUSER	0xffffffff

以下是对 sh_type 各个宏的解释：

- SHT_NULL，表明该节头无效。

- SHT_PROGBITS，代表着该 section 保存了一些由程序定义的信息。
- SHT_SYMTAB，为链接器提供符号。
- SHT_DYNSYM，代表着该 section 保存着一个动态链接时所需最小的符号集合来节省空间。
- SHT_STRTAB，代表着该 section 保存着一个字符串表。
- SHT_RELA，代表着该 section 保存着具有明确加数的重定位入口。
- SHT_HASH，代表着该 section 保存着一个符号哈希表。
- SHT_DYNAMIC，代表该 section 保存着动态链接的信息。
- SHT_NOTE，代表该 section 保存着其他一些标志文件的信息。
- SHT_NOBITS，代表着该 section 在文件中不占空间。
- SHT_REL，代表该 section 保存着一个重定位入口，但是这个重定位入口没有明确加数。
- SHT_SHLIB，代表该 section 类型是保留的。如果程序包含该类型的 section，那么这个程序不符合 ABI。
- SHT_LOPROC 到 SHT_HIPROC，这个范围内的值被保留，用于处理器特定的语义。
- SHT_LOUSER，该变量是为应用程序保留的索引的最小值。
- SHT_HIUSER，该变量是为应用程序保留的索引的最大值。在 SHT_LOUSER 和 SHT_HIUSER 之间的 section 类型可能会被应用程序使用。

在之前说过，section header 结构体中 sh_link 成员和 sh_info 成员包含的信息与 section 的类型有关。既然已经列出了 section 的各个类型，那么就来看看这两个成员与 section 类型的关系，如下表所示。

sh_type	sh_link	sh_info
SHT_DYNAMIC	该 section header 在字符表中的索引值被该 section 中的 entry 所使用	0
SHT_HASH	该 section header 在符号表中的索引值应用于符号哈希表中	0

SHT_REL SHT_RELA	该索引值是相关的符号表的索引值	该 section header 的索引值应用于重定位
SHT_SYMTAB SHT_DYNSYM	该索引值是相关的字符表的索引值	比字符表中最大的索引更大的一个值
other	SHU_UNDEF	0

3.3.3 sh_flags

sh_flags 定义了如下的宏：

Name	Value
SHF_WRITE	0x1
SHF_ALLOC	0x2
SHF_EXECINSTR	0x4
SHF_MASKPROC	0xf0000000

以下是对 sh_flags 各个宏的解释：

- SHF_WRITE，代表该 section 包含了一些数据，这些数据在程序执行时可以被改写。
- SHF_ALLOC，代表该 section 在程序执行时占用着内存。
- SHF_EXECINSTR，代表该 section 包含可执行的程序指令。
- SHF_MASKPROC，这个宏中的各个位被保留，用于处理器特定的语义。

在 section header table 的索引值 0 处的 section header 的值如下所示：

```
1  Elf32_Shdr header = table[0];
2  header.sh_name = 0; // 代表该section没有名字
3  header.sh_type = SHT_NULL;
4  header.sh_flags = 0;
5  header.sh_addr = 0;
6  header.sh_offset = 0;
7  header.sh_size = 0;
8  header.sh_link = SHN_UNDEF;
9  header.sh_info = 0;
10 header.sh_addralign = 0;
11 header.entsize = 0;
```

3.3.4 特殊的节

下面是一些特定的节，它们被系统使用，有着特定的类型和属性。

Name	Type	Attributes
.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	see below
.dynstr	SNT_STRTAB	SHF_ALLOC
.dynsym	SHT_DYNSYM	SHF_ALLOC
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.got	SHT_PROGBITS	see below
.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.interp	SHT_PROGBITS	see below
.line	SHT_PROGBITS	none
.note	SNT_NOTE	none
.plt	SHT_PROGBITS	see below
.relname	SHT_REL	see below
.relaname	SHT_RELA	see below
.rodate	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	none
.strtab	SHT_STRTAB	see below
.symtab	SHT_SYMTAB	see below
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR

下面是对这些节的介绍:

.bss	这个 section 保存着未初始化的数据，这些数据存放在程序内存映像中。当程序开始运行时，这些数据被初始化为 0。该 section 不占文件空间。
.comment	该 section 保存着版本控制的信息。
.data .data1	这两个 sections 保存着初始化了的数据，这些数据存放在程序内存映像中。
.debug	该 section 保存着为符号调试的信息，这些信息内容是未指明的。

.dynamic	该 section 保存着动态链接的信息。
.dynstr	该 section 保存着动态链接是需要的字符串。一般情况下，名字字符串与符号表的 entry 相关联。
.dynsym	该 section 保存着动态符号表。
.fini	该 section 保存着可执行指令，这些指令构成了进程的终止代码。当一个程序正常退出时，系统将执行这个 section 中的指令。
.got	该 section 保存着全局的偏移量表。
.hash	该 section 保存着一个符号的哈希表。
.init	该 section 保存着可执行指令，这些指令构成了进程的初始化代码。当一个程序的 main 函数被调用之前，系统将执行这个 section 中的指令。
.interp	该 section 保存着程序的 interpreter 的路径。
.line	该 section 包含了用于符号 debug 的行数信息，它描述了源程序与机器代码之间的对应关系。该 section 的内容是未指明的。
.plt	该 section 保存着过程链接表。
.rel .rela	该 section 保存着重定位的信息。
.rodata	该 section 保存着只读数据，这些数据将在进程映像中构造不可写的段。
.rodata1	
.shstrtab	该 section 保存着 section 的名称。
.strtab	该 section 保存着字符串。一般情况下，名字字符串与符号表的 entry 相关联。
.symtab	该 section 保存着一个符号表。
.text	该 section 保存着程序的可执行指令。

3.4 字符串表

字符串表保存着以 NULL 终止的一系列字符。object 文件使用这些字符串来描述符号和 section 名。通过索引字符串表我们就能获得相应的字符串。字符串表的索引 0 是一个 NULL 字符，随后每个字符串都是以 NULL 终止的。例子如下：

Index	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	n	a	m	e	.	\0	V	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

section header 的 sh_name 成员保存着字符串表的索引，用于指向 section 的名字。

3.5 符号表

object 文件的符号表存放着一个个条目，这些条目保存了一个程序在定位和重定位时需要的定义和引用的信息。符号表相当于一个数组，符号表索引是相应的下标。这些 entry 的定义代码如下：

```
typedef struct
{
    Elf32_Word st_name;
    Elf32_Addr st_value;
    Elf32_Word st_size;
    unsigned char st_info;
    unsigned char st_other;
    Elf32_Half st_shndx;
}Elf32_Sym;
```

下面介绍该结构体的各个成员：

st_name	这个成员的值是符号字符串表的索引。符号字符串表存放着符号的名字。如果该成员值为 0，说明该符号无名。
st_value	该成员的值是相应的符号值。
st_size	代表着符号的大小。
st_info	成员指出了符号的类型和相应的 binding 属性。
st_other	该成员值为 0，没有相应的含义。
st_shndx	符号表条目有相对应的 section。该成员的值是 section header table 的索引，用于定位 section 的 section header。

3.5.1 st_info

st_info 包含了符号表的类型和 binding 属性。一个符号的 binding 决定了链接的可视性和行为。binding 属性值定义了下列宏：

Name	Value
STB_LOCAL	0
STB_GLOBAL	1
STB_WEAK	2
STB_LOPROC	13
STB_HIPROC	15

下面介绍 binding 属性各个宏的意义：

STB_LOCAL	局部符号表，表示该符号只在当前 object 文件可见。
STB_GLOBAL	全局符号表，表示该符号对所有的 object 文件可见。
STB_WEAK	弱符号表，相当于全局符号，只是优先级较低。
STB_LOPROC-STB_HIPROC	这个范围的值被保留，用于处理器特定的语义。

全局符号与弱符号的区别如下：

- 当链接器链接几个可重定位的 object 文件时，它不允许 STB_GLOBAL 符号的同名多重定义。如果全局符号与弱符号重名，将不会引起错误，而是选择忽略弱符号的定义。如果普通符号与弱符号重名，也不会引起错误，链接器同样是忽略弱符号的定义。
- 当链接器搜索档案库时，它将选出包含了未定义的全局符号的存档成员。这个成员的定义有可能是全局符号，也可能是弱符号。链接器不会为了解决一个未定义的弱符号而选出存档成员。未定义的弱符号值为 0。

在每个符号表中，局部符号的优先级都高于弱符号和全局符号。

st_info 还包含了符号的类型，它为相关的 entry 提供了分类。与类型相关的宏定义如下所示：

Name	Value
STT_NOTYPE	0
STT_OBJECT	1
STT_FUNC	2
STT_SECTION	3
STT_FILE	4
STT_LOPROC	13
STT_HIPROC	15

下面介绍各个宏的具体含义：

STT_NOTYPE	代表了该符号的类型没有指定。
STT_OBJECT	说明该符号和一个数据对象相关。
STT_FUNC	说明该符号和一个函数或其他可执行代码相关。这个类型的符号又称为函数符号。
STT_SECTION	说明该符号和一个 section 相关。这种类型的 entry 主要是为了重定位，一般情况下具有 STB_LOCAL 约束。

STT_FILE	代表该符号给出了和目标文件相关的源文件名称。一个具有 STB_LOCAL 约束的 STT_FILE 类型的符号，它的 section 索引为 SHN_ABS。
STT_LOPROC-STT_HIPROC	这个范围的值被保留，用于处理器特定的语义。

共享文件中函数符号具有特殊的意义。当其他的 object 文件从一个共享文件中引用一个函数时，链接器将自动地为引用符号创建一个链接表。除了函数符号，共享的目标符号将不会自动地通过链接表引用。

3.5.2 st_shndx

该成员的值是 section header table 的索引，用于定位 section 的 section header。当重定位时，section 将不断移动，该成员的值也会相应的变化。某些特殊的 section 有着相应特殊的 st_shndx，于是有相应的宏。这些宏的定义在介绍 Sections 的小节中有提到，如下所示：

SHN_ABS	代表该符号的值不会随着重定位而变化。
SHN_COMMON	代表该符号标识了一个没有被分配的普通块，该符号的大小指出了需要的字节数。
SHN_UNDEF	代表该符号是未定义的。当链接器将该 object 文件和另一个定义了该符号的文件相链接时，该 object 文件中此符号将会被定义另一个文件中同名符号的定义。

3.5.3 st_value

不同类型的 object 文件中 st_value 成员的解释不同，相应的解释如下：

- 在可重定位文件中，st_value 的值代表一个符号相对于当前 section 的偏移量。如果当前 section 的索引是 SHN_COMMON，那么 st_value 保存着符号的强制对齐值。
- 在可执行文件和可共享文件中，st_value 保存着一个虚拟地址。为了使得符号对于动态链接器更为有效，文件中的 section 偏移量将让步于内存层面上的虚拟地址。

3.5.4 符号表的 0 索引

符号表的 0 索引是保留的，它的各个值如下所示：

```
Elf32_Sym zero_index = table[0];
zero_index.st_name = 0;
zero_index.st_valu = 0;
zero_index.st_size = 0;
zero_index.st_info = 0;
zero_index.st_other = 0;
zero_index.st_shndx = SHN_UNDEF;
```

3.6 重定位

重定位是将符号引用与符号定义链接起来的过程。比如，当一个程序调用一个函数的时候，相关的调用必须在执行时把控制权转移到正确的目标地址。为了实现这个过程，重定位文件应当包含如何修改 section 内容的信息，从而保证可执行文件和可共享文件中存放着用于程序映像的信息。重定位条目定义如下：

```
typedef struct
{
    Elf32_Addr r_offset;
    Elf32_Word r_info;
}Elf32_Rel;

typedef struct
{
    Elf32_Addr r_offset;
    Elf32_Word r_info;
    Elf32_SWord r_addend;
}Elf32_Rela;
```

对于结构体各个成员的解释如下：

r_offset	该成员给出了重定位行为执行的位置。对于一个重定位文件而言，成员值是受重定位影响的存储单元相对于当前 section 的偏移量。对于一个可执行文件或共享文件而言，成员值是受到重定位影响的存储单元的虚拟地址。
r_info	该成员给出了受重定位影响的符号表的索引值，同时还给出了所执行的重定位的类型。比如，call 指令的重定位 entry 的 r_info 存放着被调用函数的符号表索引值。如果符号表索引值是 STN_UNDEF，那么成员值为 0。重定位的类型与处理器有关。
r_addend	该成员给出了一个常量加数，用于计算重定位域的值。

从这两个结构体的定义可以看出，只有 Elf32_Rela 的条目包含了显式的加数，Elf32_Rel 的条目存放着隐式的加数。对于不同的处理器架构而言，这两种 entry 中

的一个可能是必要的或更为方便的。因此，对于特定的机器，使用哪一种 entr 将取决于上下文。

一个重定位的 section 与一个符号表和一个可修改的 section 有关。section header 中的 sh_info 和 sh_link 成员指明了这种关系。

3.6.1 重定位类型

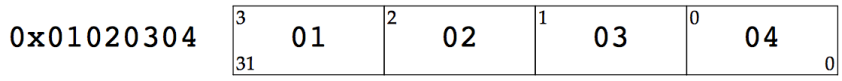
在介绍重定位类型的宏之前，为了简化描述，做了如下的标记。

A	表示用于计算重定位域的加数
B	表示在执行过程中一个共享目标被加载到内存时的基地址。一般情况下，一个共享文件使用的虚拟基地址为 0。
G	表示了相对于全局偏移表的偏移。重定位 entry 的符号在执行时将会驻留在这个表中。
GOT	表示了全局偏移表的地址。
L	表示了一个符号的过程链接表的位置。一个过程链接表 entry 将重定位一个函数调用到正确的目的单元。链接器将创建初始的过程链接表，在执行过程中动态链接器将会修改这些 entry。
P	表示了存储单元被重定位以后的位置。
S	表示了某些特定符号的值，这些符号的索引驻留在重定位 entry 中。

重定位类型如下图所示：

Name	Value	Field	Calculation
R_386_NONE	0	none	none
R_386_32	1	*word32*	S + A
R_386_PC32	2	*word32*	S + A - P
R_386_GOT32	3	*word32*	G + A - P
R_386_PLT32	4	*word32*	L + A - P
R_386_COPY	5	none	none
R_386_GLOB_DAT	6	*word32*	S
R_386_JMP_SLOT	7	*word32*	S
R_386_RELATIVE	8	*word32*	B + A
R_386_GOTOFF	9	*word32*	S + A - GOT
R_386_GOTPC	10	*word32*	GOT + A - P

在介绍各个重定位类型之前，首先介绍一下 word32。word32 是一个 32 位的域，4 字节长，与 32 位 intel 体系有相同的字节排序。如下所示：



接下来介绍各个重定位类型：

R_386_GOT32	这个类型的重定位计算了全局偏移表基地址到符号的全局偏移表 entry 之间的距离。同时还通知了链接器建立一个全局偏移表。
R_386_PLT32	这个类型的重定位计算了符号的过程链接表 entry 的地址，同时还通知链接器建立一个过程链接表。
R_386_COPY	这个类型的重定位用于动态链接。它的 r_offset 成员指向一个可写段的位置。在执行过程中，动态链接器将与共享文件的符号有关的数据拷贝到 r_offset 指定的位置。
R_386_GLOB_DAT	这个类型的重定位用于将一个特定符号的地址设定为全局偏移表的 entry。该重定位类型允许你决定符号和全局偏移表 entry 之间的一致性。
R_386_JMP_SLOT	这个类型的重定位用于动态链接。它的 r_offset 成员给出了过程链接表 entry 的地址。动态链接器可以修改该过程链接表的 entry，以便向特定的符号地址传递控制。
R_386_ReLATIVE	这个类型的重定位用于动态链接。它的偏移成员给出了共享文件中的一个位置，这个位置存放着相对地址的值。动态链接器通过将共享文件的装载地址与该相对地址相加得到相应的虚拟地址。该重定位 entry 在符号表的索引值必须为 0。
R_386_GOTOFF	这个类型的重定位用于计算符号值与全局偏移表地址之间的不同。另外还通知链接器建立全局偏移表。
R_386_GOTPC	这个类型的重定位在计算中使用了全局偏移表。同时它还会通知链接器建立全局偏移表。

3.7 Program header

program header 对于可执行文件和共享文件而言是必须的，它的定义代码如下：

```
typedef struct
{
    Elf32_Word p_type;
    Elf32_Off p_offset;
    Elf32_Addr p_vaddr;
    Elf32_Addr p_paddr;
    Elf32_Word p_filesz;
    Elf32_Word p_memsz;
    Elf32_Word p_flags;
    Elf32_Word p_align;
}Elf32_Phdr;
```

p_type	该成员指出了该元素描述了什么类型的段，或者如何解释该元素的信息。
p_offset	该成员给出了该段的基地址相对于文件开始处的偏移。
p_vaddr	该成员给出了该段在内存中的基地址。
p_paddr	该成员为该段的物理地址而保留。
p_filesz	该成员给出了文件映像中该段的字节数。
p_memsz	该成员给出了内存映像中该段的字节数。
p_flags	该成员给出了与该段相关的标志。
p_align	该成员给出了该段在内存和文件中的对齐值。

3.7.1 p_type

p_type 定义了下列宏：

Name	Value
PT_NULL	0
PT_LOAD	1
PT_DYNAMIC	2
PT_INTERP	3
PT_NOTE	4
PT_SHLIB	5
PT_PHDR	6
PT_LOPROC	0x70000000
PT_HIPROC	0x7fffffff

对各个宏的解释如下：

PT_NULL	表示该元素未使用，其他的成员值也是未定义的。
PT_LOAD	表示该元素指定了可载入的段。
PT_DYNAMIC	表示该元素指定了动态链接信息。
PT_INTERP	表示该元素制定了一个 null-terminated 路径名的位置和大小。这个段类型只对可执行文件有意义。
PT_NOTE	该数组元素指定了辅助信息的位置和大小。
PT_SHLIB	该段类型是保留的，而且具有未定义的语义。具有该类型元素的程序不遵守 ABI。
PT_PHDR	该元素指定了 program header table 的位置和大小。
PT_LOPROC-PT_HIPROC	该范围中的值被保留用于处理器特定的语义。

3.7.2 基地址

可执行文件和共享文件中有一个基地址，这个基地址是 object 文件在内存映像中最低的虚拟地址。基地址用于在动态链接过程中重定位该程序的内存映像。

一个可执行文件或一个共享文件的基地址由内存载入地址、页面大小的最大值和程序载入段的最低虚拟地址计算得到。获得基地址的方法是将内存地址减去最大页面大小的最接近的倍数。该内存地址是否与 p_vaddr 匹配取决于内存中的文件格式。

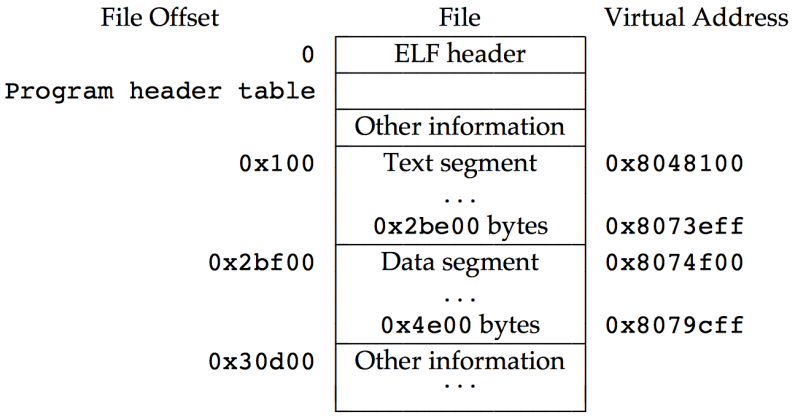
3.7.3 Note Section

供应商或系统设计者需要用特定的信息标记一个 object 文件，以便其他程序检查其是否兼容。SHT_NOTE 类型的 section 和 PT_NOTE 类型的 program header 元素就是用于此目的。这些 section 和 program header 中包含了注解信息。如果注解信息不影响程序的执行行为，那么注解信息的出现不会影响一个程序 ABI 的一致性，否则程序将不遵守 ABI 的规范并出现未定义的行为。

3.8 程序装载

当创建或增加一个进程映像时，系统将拷贝一个文件的段到一个虚拟的内存段中。系统什么时候实际地读文件依赖于程序的执行行为。一个进程只有在执行时需要引用逻辑页面时才需要一个物理页面。进程实际执行过程中，通常会留下许多为引用的特面。为了改良系统性能，可能推迟物理上频繁的读取。为了在实际操作中达到这种高效果，可执行文件和共享文件必须具有符合特定条件的段映像，这些段映像的文件偏移量和虚拟地址应该是页面大小的整数倍。

下面是一个可执行文件的例子：



例子中文件偏移量和虚拟地址在文本段和数据段都是 4KB 的整数倍，不过还是会有 4 个文件页面混合了代码和数据：

- 第一个文本页面包含了 ELF 头、程序头以及其他信息。
- 最后的文本页包含了一个数据开始的拷贝。
- 第一个数据页面有一个文本结束的拷贝。
- 最后的额数据页面可能会包含与正在运行的进程无关的文件信息。

例子中包含文本结束和数据开始的文件区域将会被影射两次，在一个虚拟地址上是文本，而另一个虚拟地址上是数据。

可执行文件和共享文件在段载入有两方面的不同：

- 一方面，可执行文件段包含了绝对代码。为了让进程正确执行，这些段必须位于建立可执行文件的虚拟地址处。因此系统使用不变的 p_vaddr 作为虚拟地址。
- 另一方面，共享目标段包含与位置无关的代码。这可以让不同进程的相应段虚拟地址各不相同，而且不影响执行。因为位置无关的代码在段间使用相对定址，所以内存中的虚拟地址的不同必须符合文件中虚拟地址的不同。

3.9 动态链接

一个可执行文件可能有一个 PT_INTERP 类型的 program header 元素。在执行过程中，系统从 PT_INTERP 段中取出一个路径名并由解释器文件的段创建出事的进程映像。也就是说，系统为解释器编写了一个内存映像，而不是使用原始的可执行文件的段映像。此时，解释器将负责接收来自于系统的控制，并且为应用程序提供一个环境变量。

解释器有两种方法接收来自系统的控制：

- 它会接收一个文件描述符来读取可执行文件，定位于开头。使用这个文件描述符来读取并映射该可执行文件的段到内存中。
- 依赖于该可执行文件的格式，系统会将这个可执行文件载入到内存中，而不是给该解释器一个文件描述符。

3.9.1 动态链接器

当使用动态链接方式建立一个可执行文件时，链接器会把一个 PT_INTERP 类型的元素加载到可执行文件中，让系统将动态链接器作为该程序的解释器。

Exec(BA_OS) 和动态链接器将一起为程序创建进程，过程如下：

- 将可执行文件的内存段加入到进程映像中。
- 将共享对象的内存段加入到进程映像中。
- 为可执行文件和它的共享对象进行重定位。
- 如果有一个用于读取可执行文件的文件描述符传递给动态链接器，那么就需要关闭它。
- 向程序传送控制，就像该程序直接从 Exec(BA_OS) 接收控制一样。

链接器同时也为动态链接器构建各种可执行文件和共享文件的相关数据，如下所示：

- 一个具有 SHT_DYNAMIC 类型的.dynamic section。这个 section 开头的结构体包含了其他动态链接信息的地址。
- 一个 SHT_HASH 类型的.hash section，包含了一个 symbol hash table。
- 一个 SHT_PROGBITS 类型的.got section，包含了全局偏移表。
- 一个 SHT_PROGBITS 类型的.plt section，包含了过程链接表。

3.9.2 dynamic section

如果一个 object 文件参与动态链接，那么它的 program header table 将会有一个 PT_DYNAMIC 类型的元素。这个段包含了.dynamic section。一个特殊的符号 _DYNAMIC 标识了这个 section。.dynamic section 包含了以下的数组：

```
1  typedef struct
2  {
3      Elf32_SWord d_tag;
4      union
5      {
6          Elf32_SWord d_val;
```

```
7             Elf32_Addr d_ptr;
8         }d_un;
9     }Elf32_Dyn;
10
11    extern Elf32_Dyn _DYNAMIC[];
```

各成员解释如下：

d_tag	控制着 d_un 的解释。
d_val	描述了具有不同解释的整型变量。
d_ptr	描述了程序的虚拟地址。

3.9.3 Shared Object Dependencies

当链接器处理一个文档库时，它取出库中的成员，并且把它们拷贝到一个输出的 object 文件中。当运行时没有包括一个动态链接器时，那些静态链接是可用的。共享文件也提供相应的服务，动态链接器必须把正确的共享文件链接到要实现的进程映像中。因此，可执行文件和共享文件之间存在着明确的依赖性。

当动态链接器为一个 object 文件创建内存段时，依赖关系表明了需要哪些 object 文件来为程序提供服务。通过重复的链接这些共享 object 和它们的依赖关系，动态链接器可以创造出一个完整的进程映像。当解决一个符号引用时，动态链接器将使用广度优先搜索来检查符号表。

在依赖关系列表中的名字既被 DT_SONAME 字符串拷贝，又被建立 object 文件时的路径名拷贝。有三种指定共享文件搜索路径的方法，如下所示：

- 动态数组标记 DT_RPATH 保存着目录列表的字符串。
- 在进程环境中 LD_LIBRARY_PATH 变量保存着目录列表的字符串。
- 动态链接器通过搜索/usr/lib 来获得共享文件。

4 从 Loader 到内核

4.1 用 loader 加载内核到内存

将内核加载到内存的过程和加载 Loader 的过程类似，根据根目录区的条目找到内存所在的扇区，然后将其读进内存。在此再看一遍根目录条目的格式:

名称	偏移	长度	描述
DIR_Name	0	0xB	文件名 8 字节，扩展名 3 字节
DIR_Attr	0xB	1	文件属性
保留位	0xC	10	保留位
DIR_WrtTime	0x16	2	最后一次写入时间
DIR_WrtDate	0x18	2	最后一次写入日期
DIR_FstClus	0x1A	2	此条目对应的开始簇号
DIR_FileSize	0x1C	4	文件大小

因为过程与之前的类似，所以在下面的代码中，我就不详细地注释了。

```
1   org 0100h
2   BaseOfStack equ 0100h
3   BaseOfKernelFile equ 08000h
4   OffsetOfKernelFile equ 0h
5
6   jmp LABEL_START
7
8   LABEL_START:
9       mov ax, cs
10      mov ds, ax
11      mov es, ax
12      mov ss, cs
13      mov sp, BaseOfStack
14
15      mov word [wSectorNo], SectorNoOfRootDirectory
16      xor ah, ah
17      xor dl, dl
18      int 13h
19  LABEL_SEARCH_IN_ROOT_DIR_BEGIN:
20      cmp word [wRootDirSizeForLoop], 0
21      jz LABEL_NO_KERNELBIN
22      dec word [wRootDirSizeForLoop]
23      mov ax, BaseOfKernelFile
24      mov es, ax
25      mov ax, OffsetOfKernelFile
26      mov bx, ax
27      mov ax, [wSectorNo]
28      mov cl, 1
```

```
29          call ReadSector
30
31          mov si, KernelFileName
32          mov di, OffsetOfKernelFile
33          cld
34          mov dx, 10h
35      LABEL_SEARCH_FOR_KERNELBIN:
36          cmp dx, 0
37          jz LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR
38          dec dx
39          mov cx, 11
40      LABEL_CMP_FILENAME:
41          cmp cx, 0
42          jz LABEL_FILENAME_FOUND
43          dec cx
44          lodsb
45          cmp al, byte [es:di]
46          jz LABEL_GO_ON
47          jmp LABEL_DIFFERENT
48      LABEL_GO_ON:
49          inc di
50          jmp LABEL_CMP_FILENAME
51      LABEL_DIFFERENT:
52          and di, 0FFE0h
53          add di, 20h
54          mov si, KernelFileName
55          jmp LABEL_SEARCH_FOR_KERNELBIN
56      LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR:
57          add word [wSectorNo], 1
58          jmp LABEL_SEARCH_IN_ROOT_DIR_BEGIN
59      LABEL_NO_KERNELBIN:
60          jmp $
61      LABEL_FILENAME_FOUND:
62          mov ax, RootDirSectors
63          and di, 0FFF0h
64          push eax
65          ; 根目录条目偏移量为01Ch的地方存放着内核的大小
66          mov eax, [es:di + 01Ch]
67          mov dword [dwKernelSize], eax
68          pop eax
69
70          ; 根目录条目偏移量为01Ch的地方存放着内核的开始簇号
71          add di, 01Ah
72          mov cx, word [es:di]
73          push cx
74          add cx, ax
75          add cx, DeltaSectorNo
76          mov ax, BaseOfKernelFile
77          mov es, ax
78          mov ax, OffsetOfKernelFile
79          mov bx, ax
80          mov ax, cx
81      LABEL_GOON_LOADING_FILE:
82          mov cl, 1
83          call ReadSector
84          pop ax
```

```
85          call GetFATEntry
86          cmp ax, 0FFFh
87          jz LABEL_FILE_LOADED
88          push ax
89          mov dx, RootDirSectors
90          add ax, dx
91          add ax, DeltaSectorNo
92          add bx, [BPB_BytsPerSec]
93          jmp LABEL_GOON_LOADING_FILE
94      LABEL_FILE_LOADED:
95          call KillMotor ; 关闭驱动马达
96          jmp $
97      KillMotor:
98          push dx
99          mov dx, 03F2h
100         mov al, 0
101         out dx, al
102         pop dx
103         ret
```

这样一来，内核就被我们加载进了内存中。