# 目标文件

## 目标文件格式

目标文件是源代码编译后但未进行链接的中间文件，内容和结构与可执行文件相似，所以与可执行文件的格式几乎一样。包括Windows下的PE-COFF文件格式、Linux下的ELF文件格式、Intel/Microsoft下的OMF（Object Module Format）格式、Unix a.out格式和MS-DOS .COM格式。

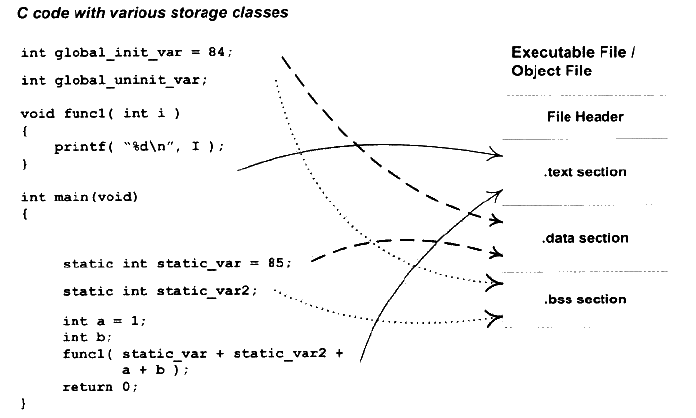
## 目标文件是什么样的

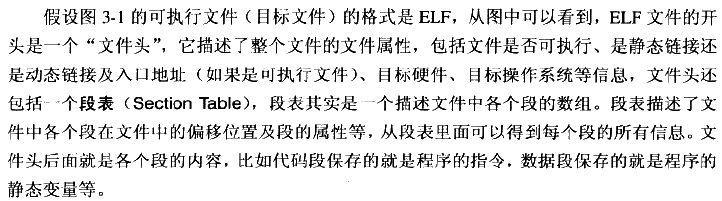
目标文件中包含编译后的机器指令、数据和其他一些连接时需要的信息，如符号表、调试信息、字符串等。这些信息按照不同属性，以Section或Segment的形式存储。

代码段(.code/.text)：存放程序源代码编译后的机器指令。

数据段(.data)：存放已初始化的全局变量和局部静态变量。

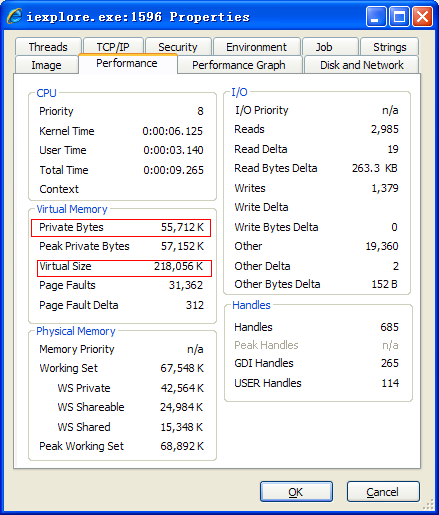
(.bss)：未初始化的全局变量和局部静态变量。.bss段只是为未初始化的全局变量和局部静态变量预留位置，他并没有内容，所以在文件中也不占据空间。





总体来说，程序源代码被编译后主要分为代码段和数据段，代码段存放程序指令，数据段和.bss段存放程序数据。程序的指令和数据分开存放好处有三：

1. 程序被装载后，数据和指令分别被影射到两个虚存区域，这两个虚存区域的权限分别被设置为可读写和只读，可以防止程序指令被有意或无意改写。
2. 现代CPU的缓存一般都被设计成数据缓存和指令缓存分离，所以程序的指令和数据被分开存放对CPU的缓存命中率提高有好处。
3. 多个进程共享同一份代码，大大节省了内存空间的占用。Iexplore占用虚存总数为218056KB，但私有数据部分只有55712KB。



## 挖掘SimpleSection.o

源文件SimpleSection.c：

1. int printf**(** const char **\*** formai**,** **...** **);**
2. int global\_init\_var **=** 84**;**
3. int global\_uninit\_var**;**
4. void func1**(** int i **)**
5. **{**
6. printf**(** "%d\n"**,** i **);**
7. **}**
8. int main**()**
9. **{**
10. static int static\_var **=** 85**;**
11. static int static\_var2**;**
12. int a **=** 1**;**
13. int b**;**
14. func1**(** static\_var **+** static\_var2 **+** a **+** b **);**
15. **return** a**;**
16. **}**

### 预编译

命令：gcc -E SimpleSection.c -o SimpleSection.i或cpp SimpleSection.c > SimpleSection.i。

主要处理规则如下：

1. 删除并展开所有#define宏定义。
2. 处理所有条件预编译指令。
3. 递归处理#include指令，将被包含的文件插入到预编译指令位置。
4. 删除所有注释。
5. 添加行号和文件名，以便于编译时编译器产生调试用的行号信息及用于编译时产生编译错误或警告时能够显示行号。
6. 保留所有#pragma编译器指令。

**经过预编译后的.i文件不包含任何宏定义，因为所有宏已被展开，并且包含的文件也已经被插入到.i文件中。所以当无法判断宏定义是否正确或头文件包含是否正确时，可以查看预编译后的文件来确定问题。**

预编译后SimpleSection.i：

1. # 1 "SimpleSection.c"
2. # 1 "<built-in>"
3. # 1 "<command-line>"
4. # 1 "SimpleSection.c"
5. int printf**(** const char **\*** formai**,** **...** **);**
6. int global\_init\_var **=** 84**;**
7. int global\_uninit\_var**;**
8. void func1**(** int i **)**
9. **{**
10. printf**(** "%d\n"**,** i **);**
11. **}**
12. int main**()**
13. **{**
14. static int static\_var **=** 85**;**
15. static int static\_var2**;**
16. int a **=** 1**;**
17. int b**;**
18. func1**(** static\_var **+** static\_var2 **+** a **+** b **);**
19. **return** a**;**
20. }

### 编译

命令：gcc -S SimpleSection.i -o SimpleSection.s。

编译过程就是把预处理完的文件进行一系列词法分析、语法分析、语义分析及优化后产生的相应汇编代码文件。

编译后的SimpleSection.s：

1. **.**file "SimpleSection.c"
2. **.**globl global\_init\_var
3. **.**data
4. **.**align 4
5. **.**type global\_init\_var**,** @object
6. **.**size global\_init\_var**,** 4
7. global\_init\_var**:**
8. **.**long 84
9. **.**comm global\_uninit\_var**,**4**,**4
10. **.**section **.**rodata
11. **.**LC0**:**
12. **.**string "%d\n"
13. **.**text
14. **.**globl func1
15. **.**type func1**,** @function
16. func1**:**
17. **.**LFB0**:**
18. **.**cfi\_startproc
19. pushl **%**ebp
20. **.**cfi\_def\_cfa\_offset 8
21. **.**cfi\_offset 5**,** **-**8
22. movl **%**esp**,** **%**ebp
23. **.**cfi\_def\_cfa\_register 5
24. subl $24**,** **%**esp
25. movl 8**(%**ebp**),** **%**eax
26. movl **%**eax**,** 4**(%**esp**)**
27. movl $**.**LC0**,** **(%**esp**)**
28. call printf
29. leave
30. **.**cfi\_restore 5
31. **.**cfi\_def\_cfa 4**,** 4
32. ret
33. **.**cfi\_endproc
34. **.**LFE0**:**
35. **.**size func1**,** **.-**func1
36. **.**globl main
37. **.**type main**,** @function
38. main**:**
39. **.**LFB1**:**
40. **.**cfi\_startproc
41. pushl **%**ebp
42. **.**cfi\_def\_cfa\_offset 8
43. **.**cfi\_offset 5**,** **-**8
44. movl **%**esp**,** **%**ebp
45. **.**cfi\_def\_cfa\_offset 8
46. **.**cfi\_offset 5**,** **-**8
47. movl **%**esp**,** **%**ebp
48. **.**cfi\_def\_cfa\_register 5
49. andl $**-**16**,** **%**esp
50. subl $32**,** **%**esp
51. movl $1**,** 24**(%**esp**)**
52. movl static\_var.1235**,** **%**edx
53. movl static\_var2.1236**,** **%**eax
54. addl **%**edx**,** **%**eax
55. addl 24**(%**esp**),** **%**eax
56. addl 28**(%**esp**),** **%**eax
57. movl **%**eax**,** **(%**esp**)**
58. call func1
59. movl 24**(%**esp**),** **%**eax
60. leave
61. **.**cfi\_restore 5
62. **.**cfi\_def\_cfa 4**,** 4
63. ret
64. **.**cfi\_endproc
65. **.**LFE1**:**
66. **.**size main**,** **.-**main
67. **.**data
68. **.**align 4
69. **.**type static\_var.1235**,** @object
70. **.**size static\_var.1235**,** 4
71. static\_var.1235**:**
72. **.**long 85
73. **.**local static\_var2.1236
74. **.**comm static\_var2.1236**,**4**,**4
75. **.**ident "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
76. .section .note.GNU-stack,"",@progbits

### 汇编

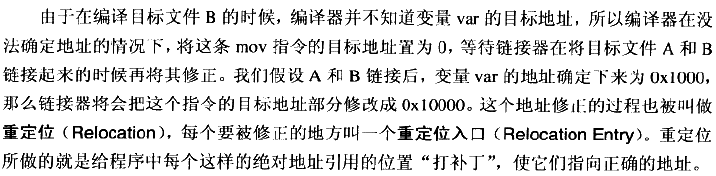
命令：gcc -c SimpleSection.s -o SimpleSection.o或as SimpleSection.s –o SimpleSection.o。

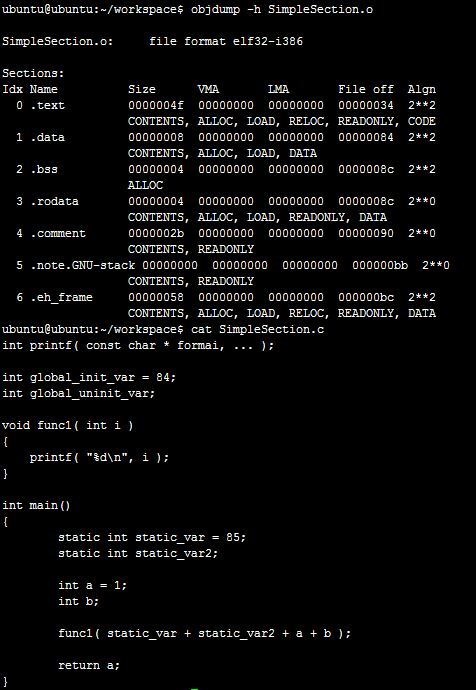
将汇编代码转变成机器可执行指令。

### 链接

命令：ld。

链接过程中包括了地址和空间分配、符号决议和重定位等步骤。





还有一个size命令可以用来查看ELF文件的代码段、数据段和BSS段的长度。使用objdump –x -s -d SimpleSection.o命令可以查看文件头和各个段的详细内容。

ubuntu@ubuntu**:~/**workspace**$** objdump **-**x **-**s **-**d SimpleSection.o

SimpleSection.o**:** file format elf32**-**i386

SimpleSection.o

architecture**:** i386**,** flags 0x00000011**:**

HAS\_RELOC**,** HAS\_SYMS

start address 0x00000000

Sections**:**

Idx Name Size VMA LMA File off Algn

0 **.text** 0000004f 00000000 00000000 00000034 2**\*\***2

CONTENTS**,** ALLOC**,** LOAD**,** RELOC**,** **READONLY,** CODE

1 .data 00000008 00000000 00000000 00000084 2**\*\***2

CONTENTS**,** ALLOC**,** LOAD**,** DATA

2 **.bss** 00000004 00000000 00000000 0000008c 2**\*\***2

ALLOC

3 .rodata 00000004 00000000 00000000 0000008c 2**\*\***0

CONTENTS**,** ALLOC**,** LOAD**,** **READONLY,** DATA

4 .comment 0000002b 00000000 00000000 00000090 2**\*\***0

CONTENTS**,** **READONLY**

5 .note.GNU**-**stack 00000000 00000000 00000000 000000bb 2**\*\***0

CONTENTS**,** **READONLY**

6 .eh\_frame 00000058 00000000 00000000 000000bc 2**\*\***2

CONTENTS**,** ALLOC**,** LOAD**,** RELOC**,** **READONLY,** DATA

SYMBOL TABLE**:**

00000000 l df **\*ABS\*** 00000000 SimpleSection.c

00000000 l d **.text** 00000000 **.text**

00000000 l d .data 00000000 .data

00000000 l d **.bss** 00000000 **.bss**

00000000 l d .rodata 00000000 .rodata

00000004 l O .data 00000004 static\_var.1235

00000000 l O **.bss** 00000004 static\_var2.1236

00000000 l d .note.GNU**-**stack 00000000 .note.GNU**-**stack

00000000 l d .eh\_frame 00000000 .eh\_frame

00000000 l d .comment 00000000 .comment

00000000 g O .data 00000004 global\_init\_var

00000004 O **\***COM**\*** 00000004 global\_uninit\_var

00000000 g F **.text** 0000001b func1

00000000 **\***UND**\*** 00000000 printf

0000001b g F **.text** 00000034 main

Contents of section **.text:**

0000 5589e583 ec188b45 08894424 04c70424 U......E..D$...**$**

0010 00000000 e8fcffff ffc9c355 89e583e4 ...........U....

0020 f083ec20 c7442418 01000000 8b150400 ... .D$.........

0030 0000a100 00000001 d0034424 18034424 ..........D$..D**$**

0040 1c890424 e8fcffff ff8b4424 18c9c3 ...$......D$...

Contents of section .data**:**

0000 54000000 55000000 T...U...

Contents of section .rodata**:**

0000 25640a00 %d..

Contents of section .comment**:**

0000 00474343 3a202855 62756e74 752f4c69 .GCC**:** **(**Ubuntu**/**Li

0010 6e61726f 20342e36 2e332d31 7562756e naro 4.6.3**-**1ubun

0020 74753529 20342e36 2e3300 tu5**)** 4.6.3.

Contents of section .eh\_frame**:**

0000 14000000 00000000 017a5200 017c0801 .........zR..**|**..

0010 1b0c0404 88010000 1c000000 1c000000 ................

0020 00000000 1b000000 00410e08 8502420d .........A....B.

0030 0557c50c 04040000 1c000000 3c000000 .W..........**<**...

0040 1b000000 34000000 00410e08 8502420d ....4....A....B.

0050 0570c50c 04040000 .p......

Disassembly of section **.text:**

00000000 **<**func1**>:**

0**:** 55 **push** %ebp

1**:** 89 e5 **mov** %esp**,%ebp**

3**:** 83 ec 18 **sub** $0x18**,%esp**

6**:** 8b 45 08 **mov** 0x8**(%ebp),%eax**

9**:** 89 44 24 04 **mov** %eax**,**0x4**(%esp)**

d**:** c7 04 24 00 00 00 00 movl $0x0**,(%esp)**

10**:** R\_386\_32 .rodata

14**:** e8 fc ff ff ff **call** 15 **<**func1**+**0x15**>**

15**:** R\_386\_PC32 printf

19**:** c9 **leave**

1a**:** c3 **ret**

0000001b **<**main**>:**

1b**:** 55 **push** %ebp

1c**:** 89 e5 **mov** %esp**,%ebp**

1e**:** 83 e4 f0 **and** $0xfffffff0**,%esp**

21**:** 83 ec 20 **sub** $0x20**,%esp**

24**:** c7 44 24 18 01 00 00 movl $0x1**,**0x18**(%esp)**

2b**:** 00

2c**:** 8b 15 04 00 00 00 **mov** 0x4**,%edx**

2e**:** R\_386\_32 .data

32**:** a1 00 00 00 00 **mov** 0x0**,%eax**

33**:** R\_386\_32 **.bss**

37**:** 01 d0 **add** %edx**,%eax**

39**:** 03 44 24 18 **add** 0x18**(%esp),%eax**

3d**:** 03 44 24 1c **add** 0x1c**(%esp),%eax**

41**:** 89 04 24 **mov** %eax**,(%esp)**

44**:** e8 fc ff ff ff **call** 45 **<**main**+**0x2a**>**

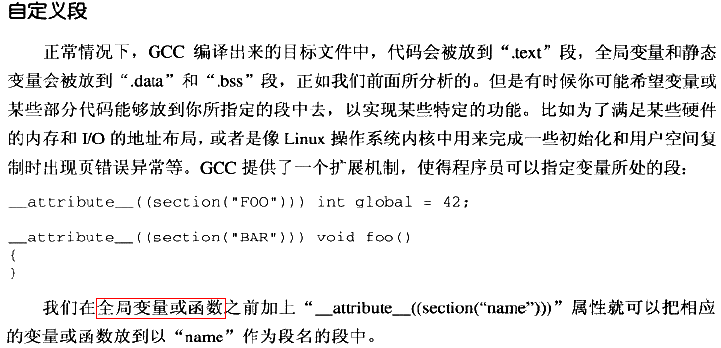
45**:** R\_386\_PC32 func1

49**:** 8b 44 24 18 **mov** 0x18**(%esp),%eax**

4d**:** c9 **leave**

4e**:** c3 ret

从上述命令返回的结果中SYMBOL TABLE一段可以看出，本应预期存在于.bss段的变量global\_uninit\_var没有被存放在任何段，只是一个未定义的“COMMON”符号。有些编译器会将全局的未初始化变量存放在目标文件的.bss段，有些则不存放，只是预留一个未定义的全局变量符号，等到最终连接成可执行文件的时候再在.bss段分配空间。



注：该扩展机制只对全局变量或函数有效。

## ELF文件结构描述

### 文件头

1. ubuntu**@**ubuntu**:~/**workspace**$ readelf** **-h** SimpleSection.o
2. ELF Header**:**
3. Magic**:** 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
4. Class**:** ELF32
5. Data**:** 2's complement, little endian
6. Version: 1 (current)
7. OS/ABI: UNIX - System V
8. ABI Version: 0
9. Type: REL (Relocatable file)
10. Machine: Intel 80386
11. Version: 0x1
12. Entry point address: 0x0
13. Start of program headers: 0 (bytes into file)
14. Start of section headers: 372 (bytes into file)
15. Flags: 0x0
16. Size of this header: 52 (bytes)
17. Size of program headers: 0 (bytes)
18. Number of program headers: 0
19. Size of section headers: 40 (bytes)
20. Number of section headers: 13
21. Section header string table index: 10

ELF文件头结构及相关常数定义在/usr/include/elf.h中，以32位版本的头文件为例：

**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 \*/

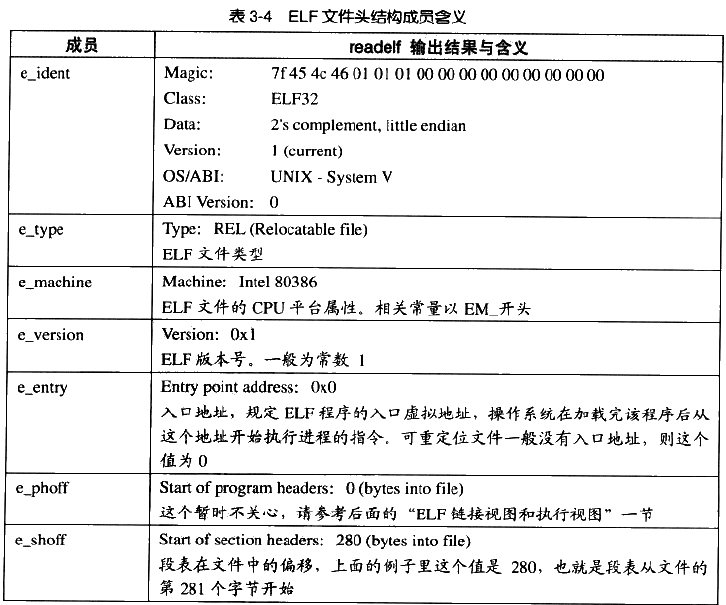
Elf32\_Half e\_shentsize**;** /\* Section header table entry size \*/

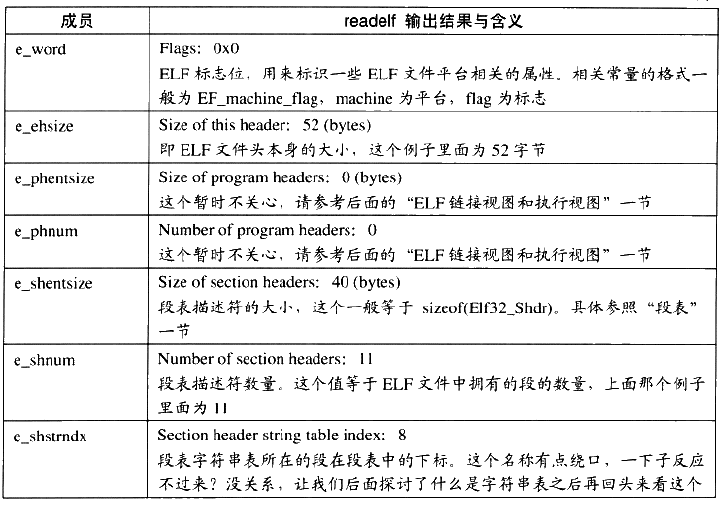
Elf32\_Half e\_shnum**;** /\* Section header table entry count \*/

Elf32\_Half e\_shstrndx**;** /\* Section header string table index \*/

**}** Elf32\_Ehdr**;**

除了e\_ident成员对应了文件头中的Class、Data、Version、OS/ABI和ABI Version外，其他的参数与Elf32\_Ehdr中的成员都一一对应，对照表如下：





### 段表

段表在ELF文件中的位置由ELF文件头的e\_shoff成员决定，他是一个以ELF32\_Shdr结构体为元素的数组，数组元素的个数等于段的个数。ELF32\_Shdr结构体又被称为段描述符。

**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 \*/

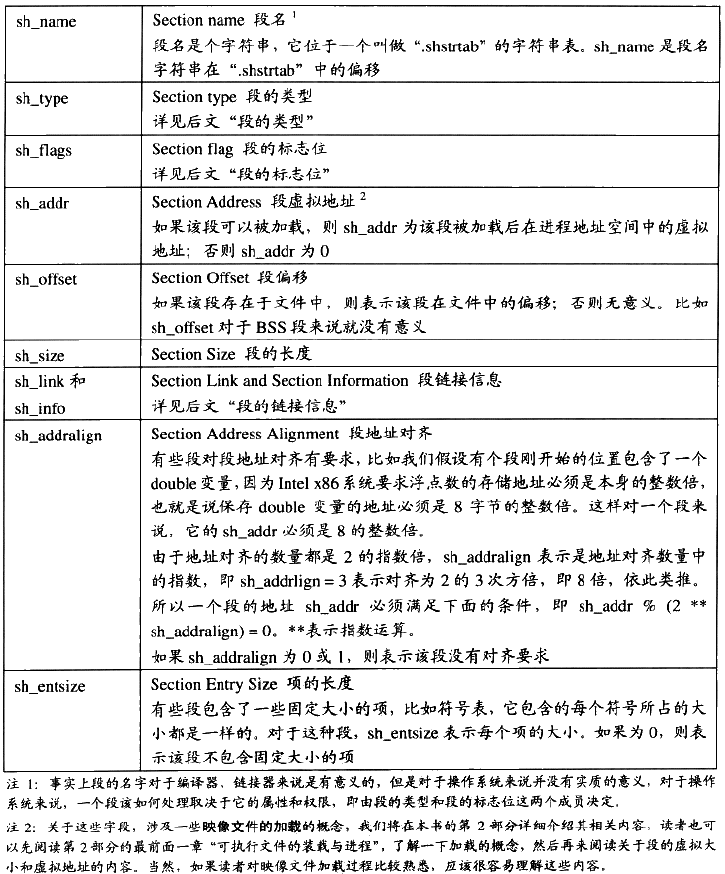
Elf32\_Word sh\_link**;** /\* Link to another section \*/

Elf32\_Word sh\_info**;** /\* Additional section information \*/

Elf32\_Word sh\_addralign**;** /\* Section alignment \*/

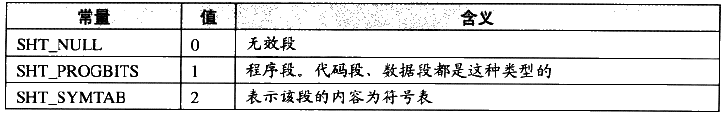
Elf32\_Word sh\_entsize**;** /\* Entry size if section holds table \*/

**}** Elf32\_Shdr**;**



段的名字不能真正的表示段的类型，对于编译器和链接器来说，主要决定段的属性的是Elf32\_Shdr中的sh\_type和sh\_flags。

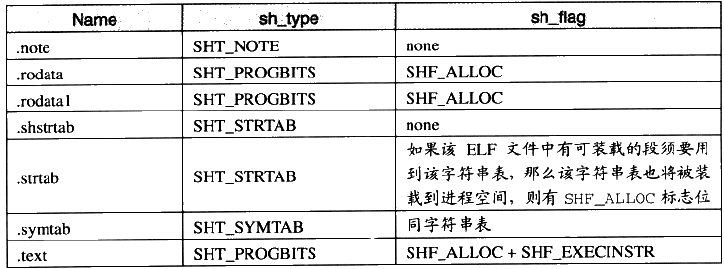
段的类型：

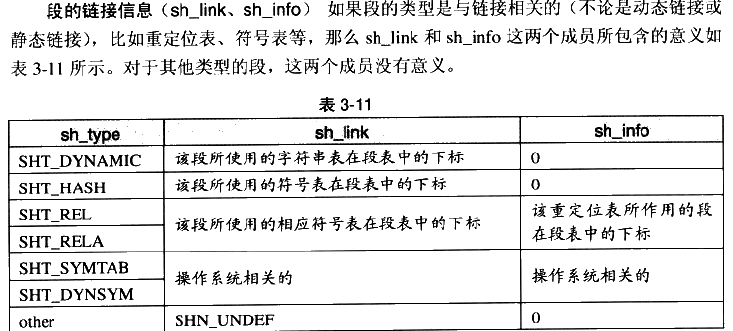




段的标志位(表示该段在进程虚拟地址空间中的属性)：





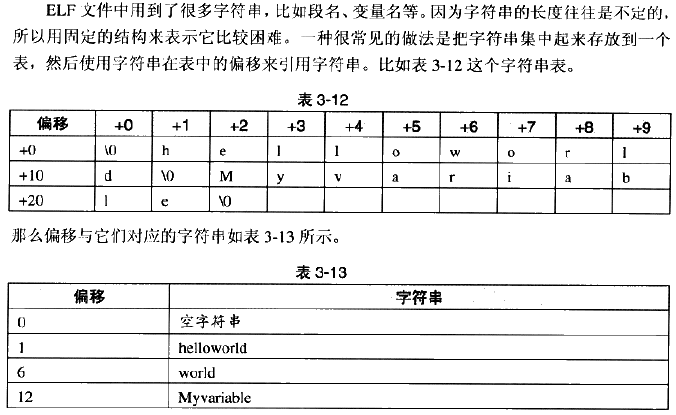


### 重定位表

对于每个需要重定位的代码段或数据段，都会有一个相应的重定位表，如.rel.text就是针对.text段的重定位表

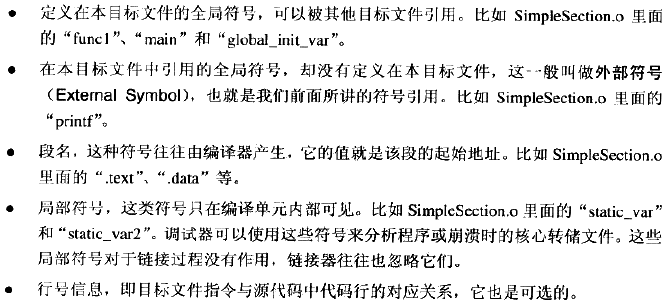
### 字符串表

只需分析ELF文件头，就可以得到段表和段字符串表的位置，从而解析整个ELF文件。



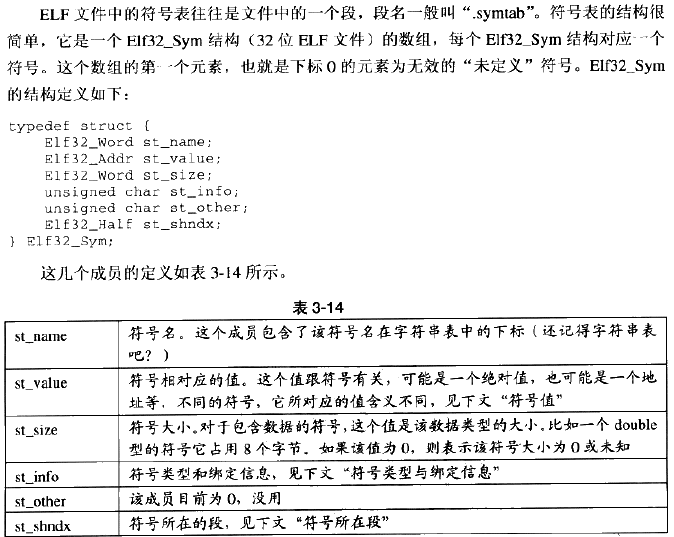
## 链接的接口—符号

每个定义的符号有一个对应的值，叫做符号值，对于变量和函数来说，符号值就是他们的地址。

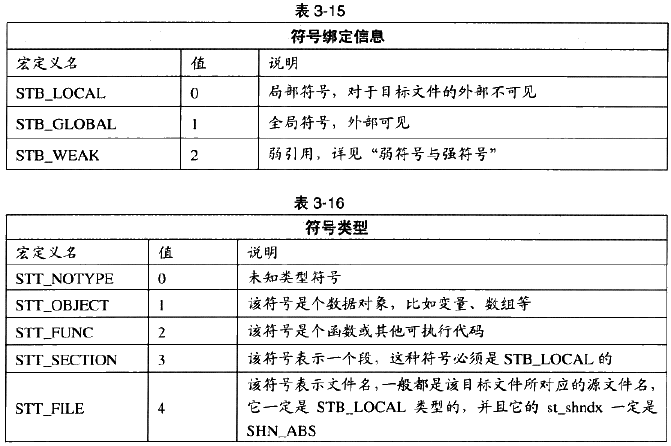


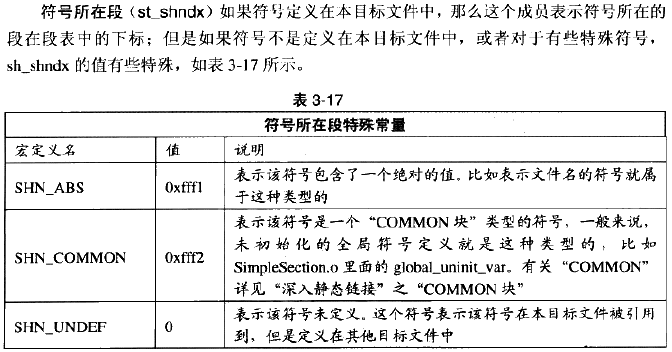
最值得关注的就是全局符号，即上面分类中的第一类和第二类。因为链接过程只关心全局符号的相互粘合，局部符号、段名、行号等都是次要的，他们对于其他目标文件来说是不可见的。

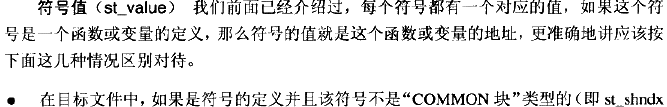
### ELF符号表结构

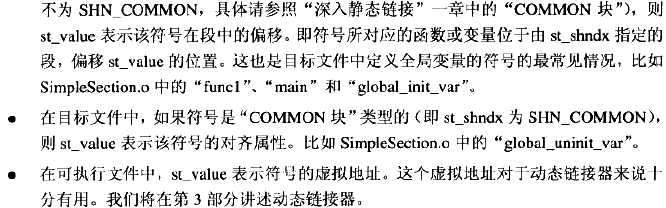


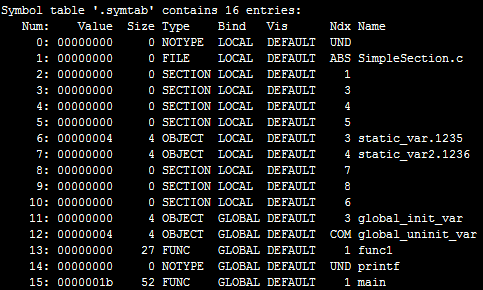
st\_info（符号类型和绑定信息）：低四位表示符号的类型，高28位表示符号绑定信息。





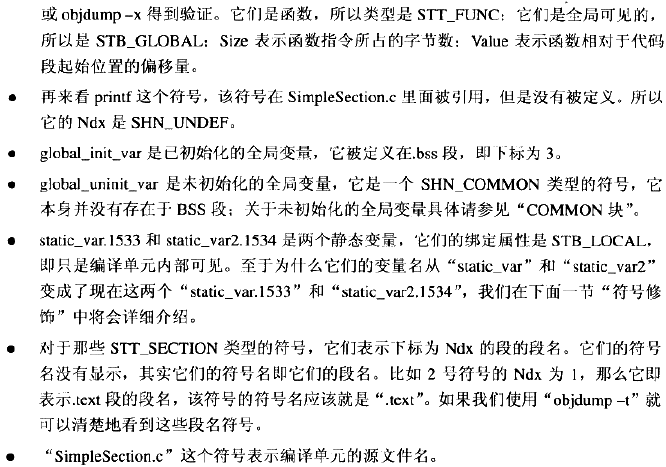






Readelf的输出格式与上面描述的Elf32\_Sym的各个成员几乎一一对应，Num为符号表数组的下标；Vis目前在C/C++中未使用；Ndx即st\_shndx，表示该符号所属的段。第一个符号，即下标为0的符号，永远是一个未定义的符号。

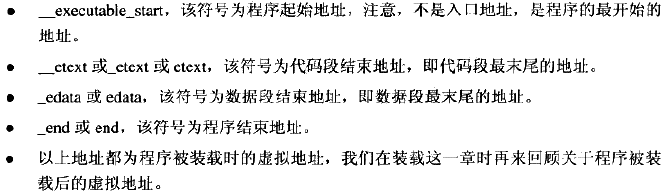


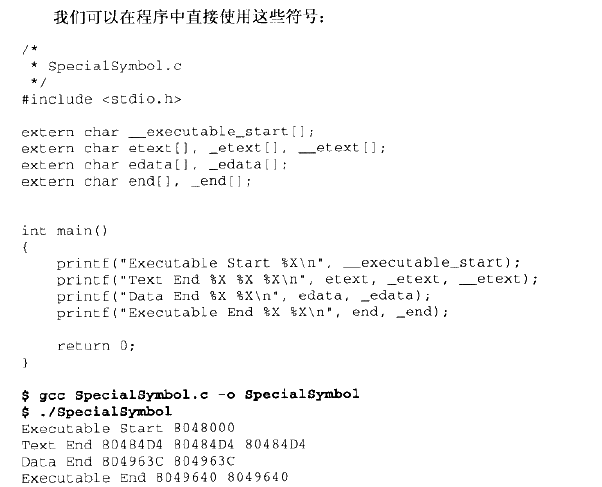


### 特殊符号

当使用ld作为链接器来链接生产可执行文件时，他会为我们定义很多特殊的符号，这些符号并没有在程序中定义，但是可以直接声明并且引用他，我们称之为特殊符号。其实这些符号是被定义在ld链接器的链接脚本中的。注意，只有在使用ld链接生产可执行文件的时候这些符号才会存在。

几个具有代表性的特殊符号如下：





### 符号修饰与函数签名

略。

### extern “C”

略。

### 1.5.5 弱符号与强符号

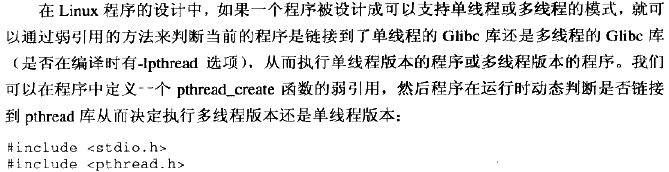
对于C/C++语言来说，编译器默认函数和初始化了的全局变量为强符号，未初始化的全局变量为弱符号。我们也可以通过GCC的\_\_attribute\_\_((weak))来定义任何一个强符号为弱符号。强符号和弱符号都是针对定义来说的，不是针对符号的引用。

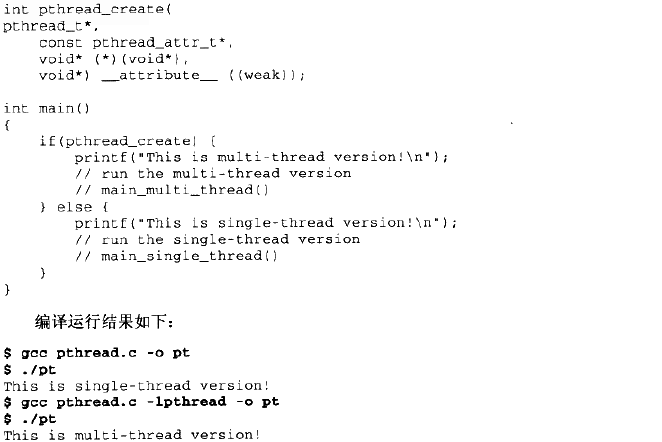


**强引用和弱引用** 目前所看到的对外部目标文件的符号引用在目标文件被最终链接成可执行文件时，他们需要被正确决议，如果没有找到该符号的定义，链接器就会报符号未定义错误，这种被称为强引用。与之对应还有一种弱引用，在处理弱引用时，如果该符号有定义，则链接器将该符号的引用决议；如果该符号未被定义，则链接器对于该引用不报错。链接器处理强引用和弱引用的过程几乎一样，只是对于未定义的弱引用，链接器不认为他是一个错误。一般对于未定义的弱引用，链接器默认其为0，或者一个特殊的值，以便于程序代码能够识别。弱引用和弱符号主要用于库的链接过程，弱符号跟链接器的COMMON块概念联系很紧密。

在GCC中，可以通过使用\_\_attribute\_\_((weakref))扩展关键字来声明对一个外部函数的引用为弱引用。

库中定义的弱符号可以被用户定义的强符号所覆盖，从而使得程序可以使用自定义版本的库函数；或者程序可以对某些扩展功能模块的引用定义为弱引用，当我们将扩展模块与程序连接在一起时，功能模块就可以正常使用，如果去掉这些模块，那么程序也可以正常连接，只是缺少了相应功能，使得程序的功能更加容易裁剪和组合。

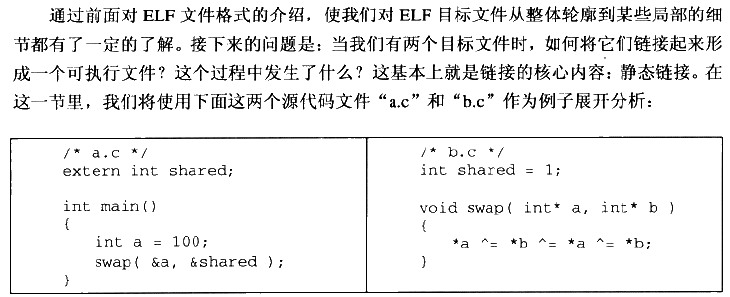




## 调试信息

略。

# 静态链接

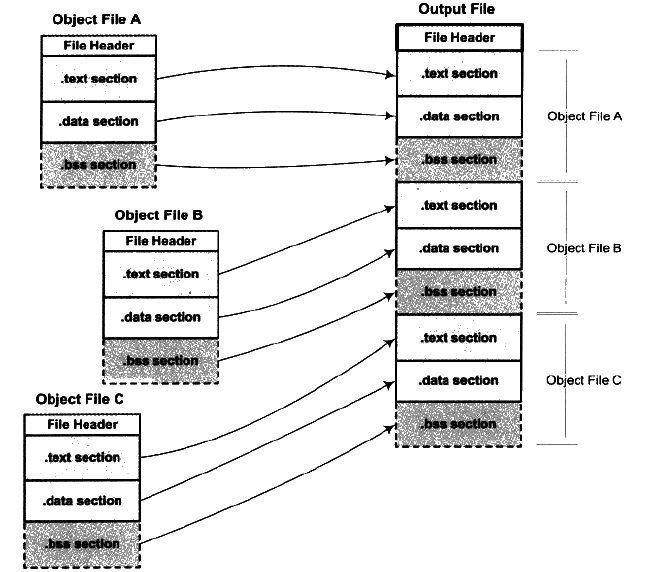


## 空间与地址分配

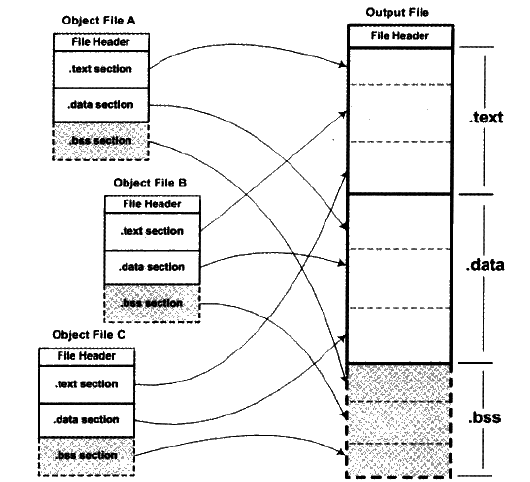
阐述链接过程中输出文件中的空间如何分配给输入文件。

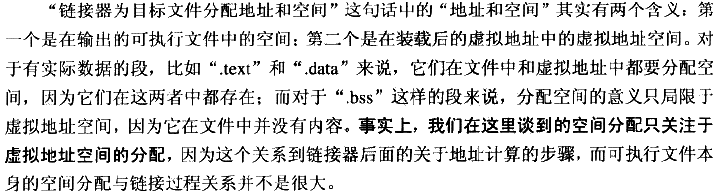
### 按序叠加

将输入的目标文件按照次序叠加起来。

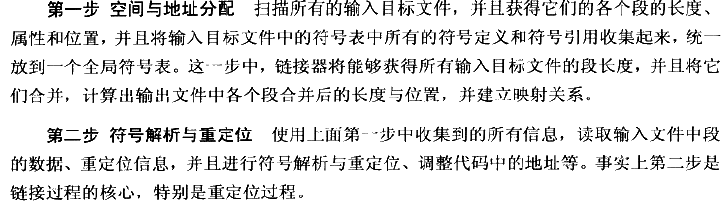


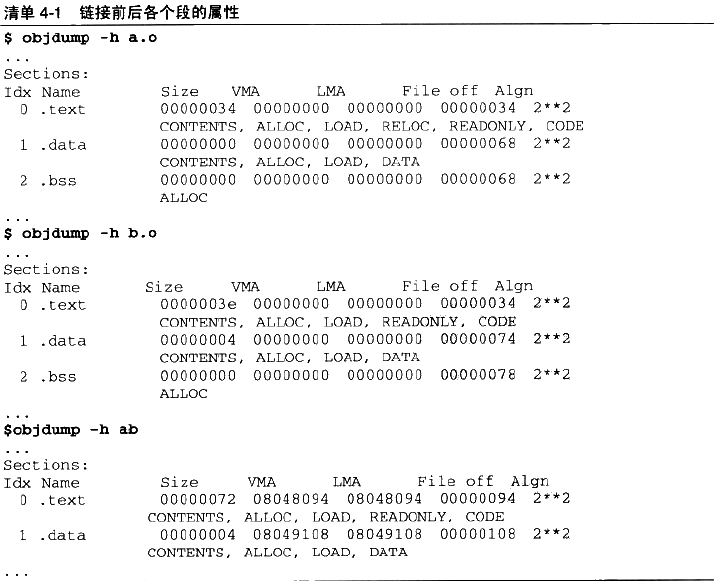
### 相似段合并

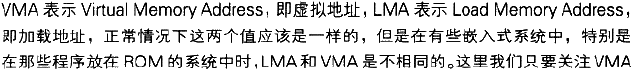




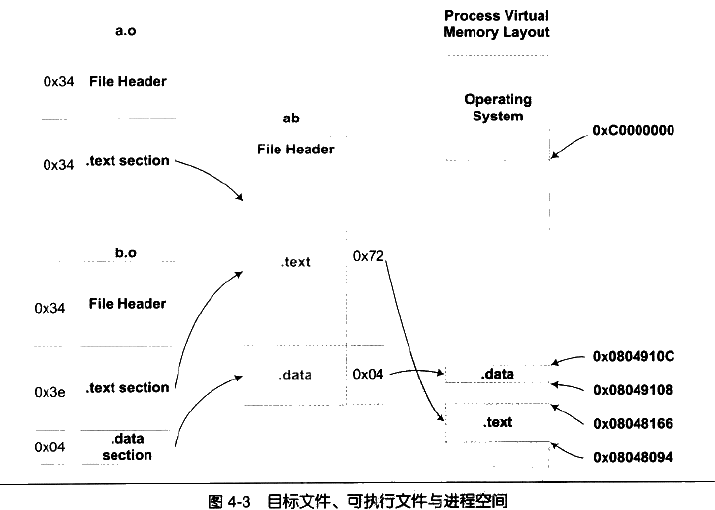
采用这种空间分配策略的链接器一般采用“两步链接”的方法。







链接后程序中使用的地址已经是程序在进程中的虚拟地址即上面各段中的VMA和Size，文件便宜已不需关注。



在Linux下，ELF可执行文件默认从地址0x08048000开始分配。

### 符号地址的确定

即将链接完成后各个段的虚拟起始地址与符号在段中的偏移量相加，获得符号的最终虚拟地址。

## 符号解析与重定位

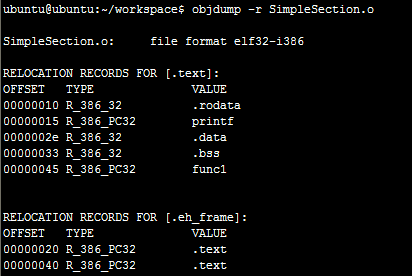
### 重定位

静态链接的核心内容。

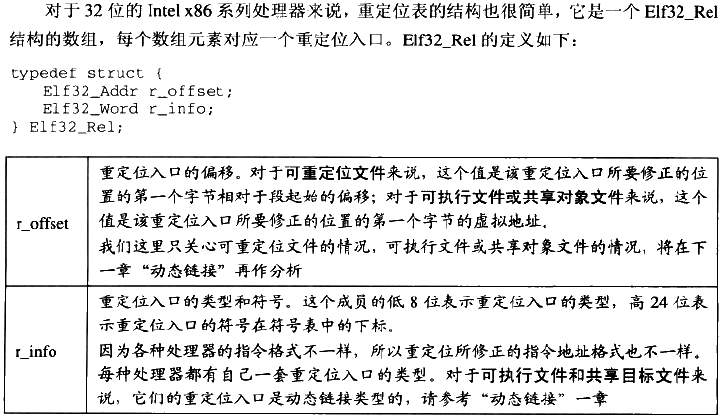
链接器在完成地址和空间分配之后就已经可以确定所有符号的虚拟地址了，那么链接器就可以根据符号的地址对每个需要重定位的指令进行地址修正。

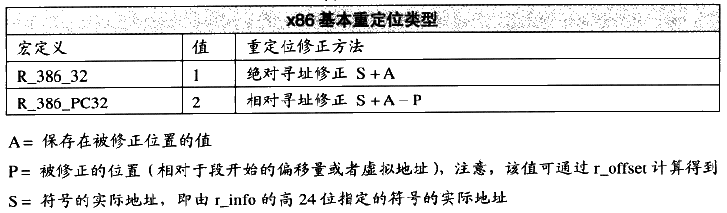
### 重定位表

objdump –r a.o可用来查看目标文件的重定位表，每个需要被重定位的地方叫一个重定位入口。



重定位入口的偏移表示该入口在要被重定位的段中的位置。

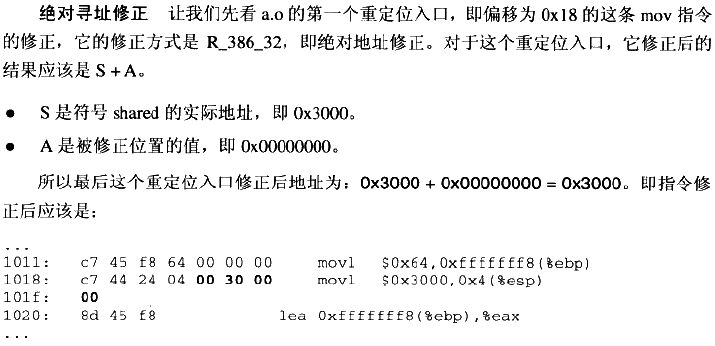


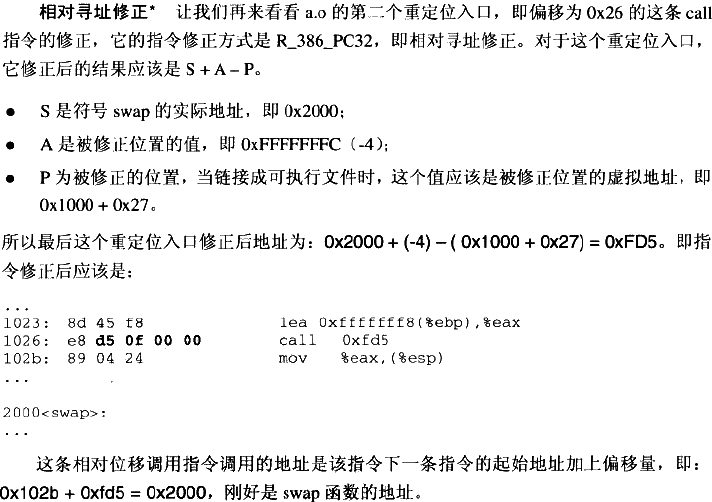


### 符号解析

重定位过程也伴随着符号的解析过程，每个目标文件都可能定义一些符号，也可能引用到定义在其他目标文件的符号。重定位的过程中，每个重定位的入口都是对一个符号的引用，那么当链接器需要对某个符号的引用进行重定位时，他就要确定这个符号的目标地址。这时链接器就会去查找由所有输入目标文件的符号表组成的全局符号表，找到相应的符号进行定位。

### 指令修正方式

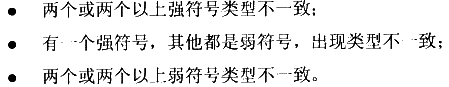




绝对地址修正后的地址为该符号的实际地址，相对寻址修正后的地址为符号距离被修正位置的地址差。

## COMMON块

主要用于解决符号类型不一致的问题。

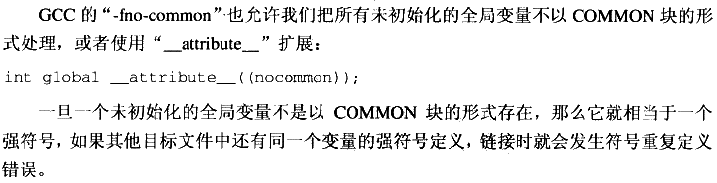


1. 链接器报符号多重定义错误。
2. 最终输出结果中的符号所占空间与强符号相同。
3. 取最大的那块作为实际的空间使用。

COMMON块机制最早来源于Fortran，早起的Fortran没有动态分配空间的机制，程序员必须事先声明他所需要的临时空间的大小，这个信息就保存在目标文件的COMMON块中。当将这些目标文件链接在一起时，取最大的那块作为实际的空间使用。

当编译器将一个编译单元编译成目标文件时，如果该编译单元包含弱符号（未初始化的全局变量就是典型的弱符号），那么该弱符号最终所占用的空间大小未知，编译器此时无法为该符号在BSS段分配空间，此时就需要将其临时存放在COMMON块中。

链接器在链接过程中可以确定弱符号的大小，所以链接器可以在最终输出文件的BSS段为其分配空间，总体看来，未初始化全局变量最终还是被放在BSS段。



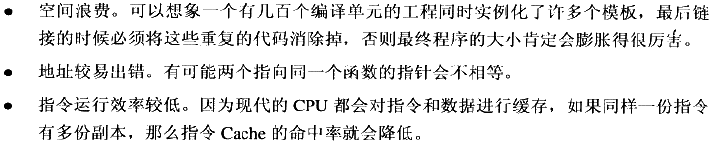
## C++相关问题

C++重复代码消除；全局构造与析构；二进制兼容性。

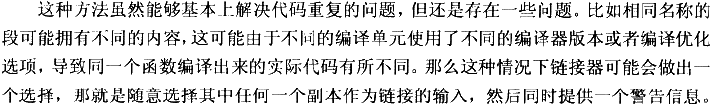
### 重复代码消除

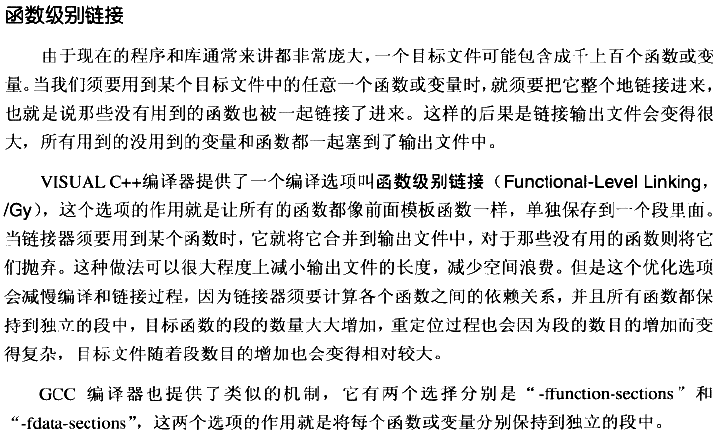
当一个模板在多个编译单元同时实例化成相同的类型的时候，他并不知道自己是否在别的编译单元也被实例化了。所以当一个模板在多个编译单元同时实例化成相同类型的时候，必然会生成重复的代码。

不处理重复代码有如下问题：



解决问题的办法是将模板的实例化代码单独存放在一个段里，每个段只包含一个模板实例，链接器在最终连接的时候区分相同的模板实例段，然后将他们合并入最后的代码段。

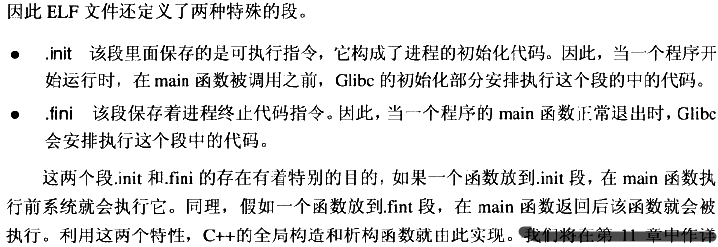




### 全局构造与析构

在main函数被调用之前，为了程序能够顺利执行，要先初始化进程执行环境，如堆分配初始化、线程子系统等。C++的全局对象构造函数在main之前被执行，析构函数在main之后被执行。

Linux系统下一般程序的入口是“\_start”，这个函数是Linux系统库（Glibc）的一部分。当我们的程序与Glibc库链接在一起形成最终可执行文件后，这个函数就是程序的初始化部分的入口，程序初始化部分完成一系列初始化过程之后，会调用main函数来执行程序的主体。在main函数执行完成后，返回到初始化部分，进行一些清理工作，然后结束进程。



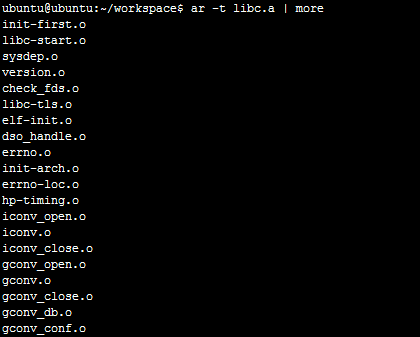
### C++与ABI

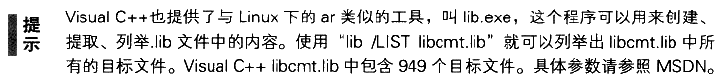
我们把符号修饰标准、变量内存布局、函数调用方式等这些跟可执行代码二进制兼容性相关的内容称为ABI(Application Binary Interface)。

## 静态库链接

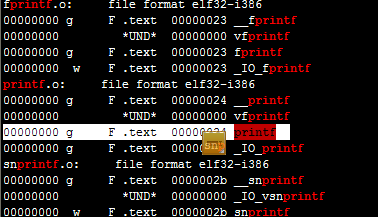
一般情况下，一种语言的开发环境往往会附带有语言库，这些库就是对操作系统的API的包装。

其实一个静态库可以简单地看成一组目标文件的集合，即很多目标文件经过压缩打包后形成的一个文件。例如：glibc本身是用C语言开发的，它由成百上千个C语言源代码文件组成，也就是说，编译完成以后有相同数量的目标文件，把这些零散的目标文件直接提供给库的使用者，很大程度上会造成文件传输、管理和组织方面的不便，于是通常使用ar压缩程序将这些目标文件压缩到一起，并对其进行编号和索引，以便于查找和检索。

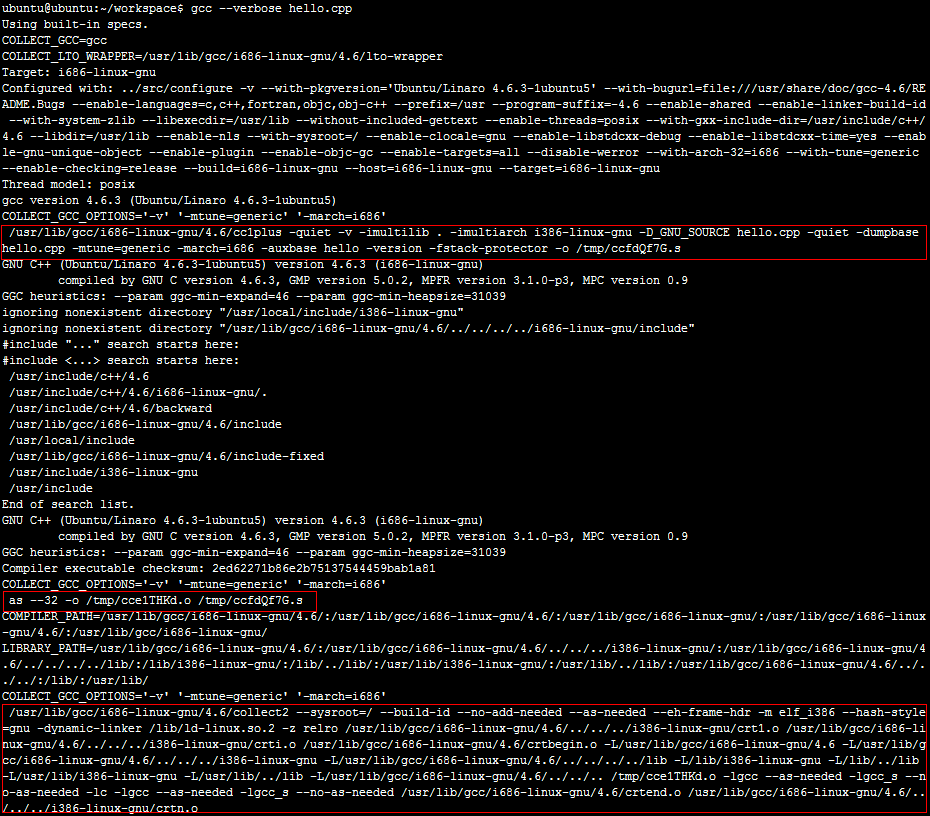


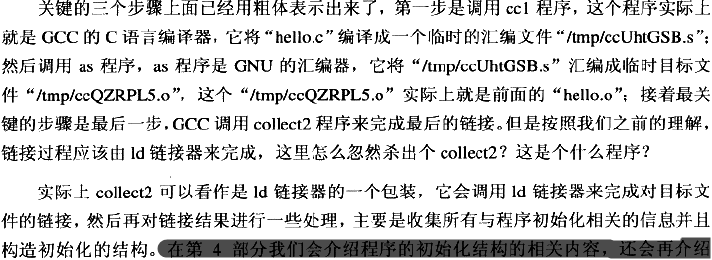


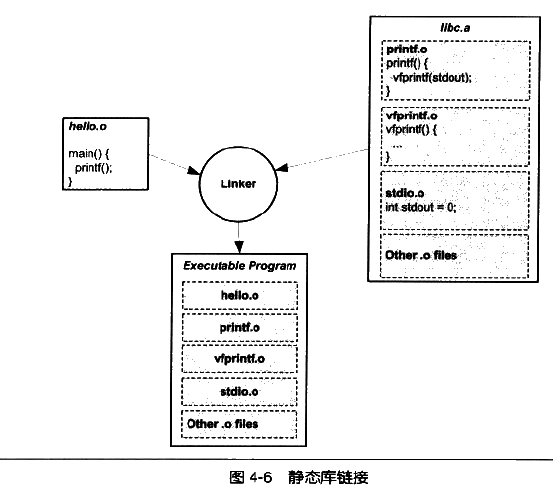
使用命令objdump -t libc.a | grep printf看到printf函数在libc.a库中的位置。



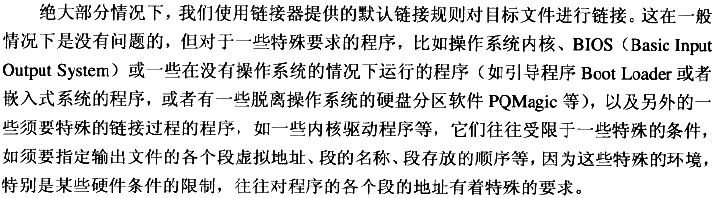
我们可以使用gcc的—verbose编译选项将整个编译链接过程的中间步骤打印出来：

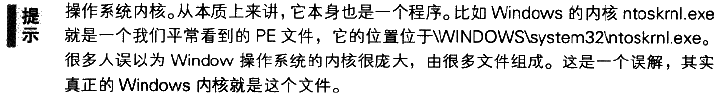






## 链接过程控制





### 链接控制脚本

1. 使用命令行来给链接器指定参数。
2. 将链接指令存放在目标文件里面，编译器经常会通过这种方法向链接器传递指令。如visual C++编译器会把链接参数放在PE目标文件的.drectve段以用来传递参数。
3. 使用链接控制脚本。本节介绍，也是最为灵活、最为强大的链接控制方法。

在没有指定链接脚本时，ld使用默认链接脚本，可以使用ld –verbose命令查看ld默认的链接脚本。默认的ld链接脚本存放在/usr/lib/ldscripts下，不同的平台、输出文件格式都有相应的链接脚本。为了更加精确地控制链接过程，我们可以自己写一个脚本，然后指定该脚本为链接控制脚本，如：ld –T link.script。

### 最“小”的程序

该程序可以脱离C语言运行库；该程序不使用库的\_start作为程序入口，也无需由库负责初始化后调用main函数来执行程序的主体部分；该程序将所有段合并到一个段。

tinyHello.c:

1 char **\*** str **=** "Hello world!\n"**;**

2

3 void print**()**

4 **{**

5 asm**(**

6 "movl $13, %%edx \n\t" // str长度为13

7 "movl %0, %%ecx \n\t" // ecx = str

8 "movl $0, %%ebx \n\t" // 文件句柄，stdout为0

9 "movl $4, %%eax \n\t" // write调用号为4

10 "int $0x80 \n\t"

11 **::**"r"**(**str**):**"edx"**,** "ecx"**,** "ebx"

12 **);**

13 **}**

14

15 void exit**()**

16 **{**

17 asm**(**

18 "movl $42, %ebx \n\t"

19 "movl $1, %eax \n\t" // exit调用号为1

20 "int $0x80 \n\t"

21 **);**

22 **}**

23

24 void nomain**()**

25 **{**

26 print**();**

27 exit**();**

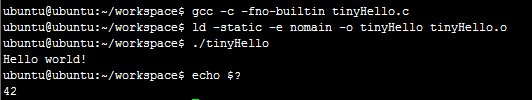
28 **}**

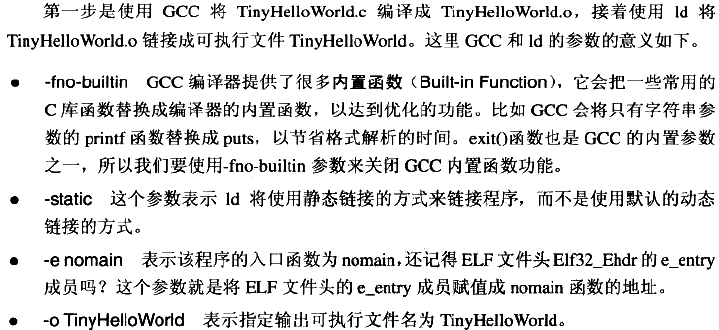
print()函数使用了Linux的write系统调用，exit()函数使用了exit系统调用。系统调用通过0x80中断实现，其中eax为调用号，ebx、ecx、edx等通用寄存器用来传递参数，比如write调用是往一个文件句柄写入数据，如果用C语言来描述它的原型就是：int write(int filedesc, char \* buffer, int size);

同理，exit系统调用中，ebx表示进程退出码。我们平时的main程序中的return数值会返回给系统库，由系统库将该数值传递给exit系统调用，这样父进程就可以接收到子进程的退出码，tinyHello的退出码就是42。

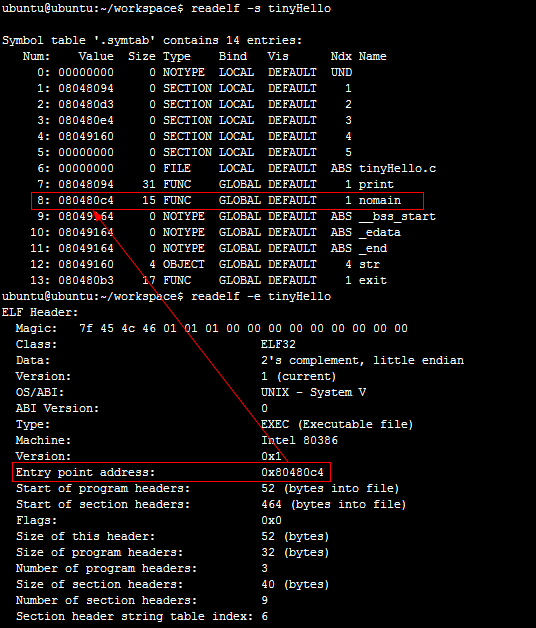
需要调用exit系统调用结束进程是因为如果是普通程序，main()函数结束后控制权返回给系统库，由系统库负责调用exit，退出程序。而nomain结束后系统控制权不会返回，可能会执行到nomain后面不正常的指令，最后导致进程异常退出。

编译并运行程序可见：

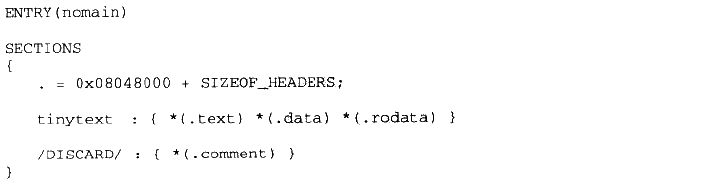


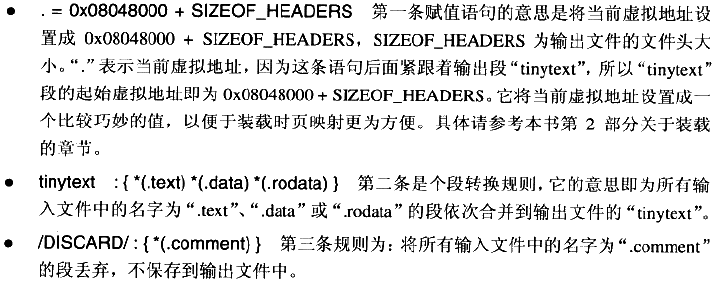


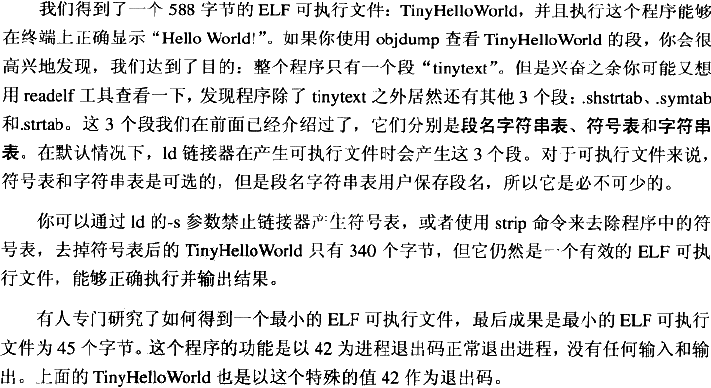
交叉比较tinyHello文件头和符号表即可看出tinyHello的入口函数为nomain：

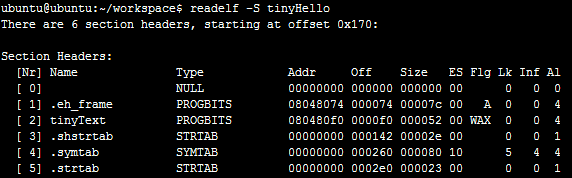


### 使用ld链接脚本

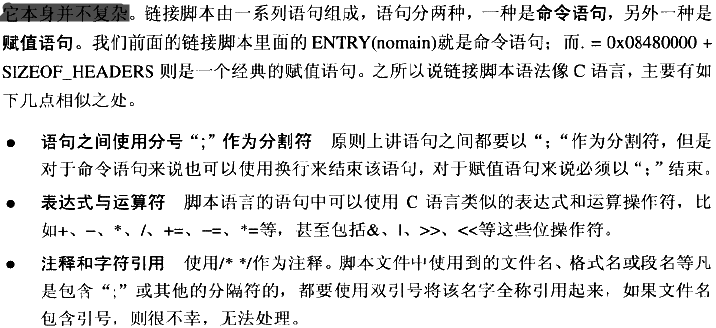




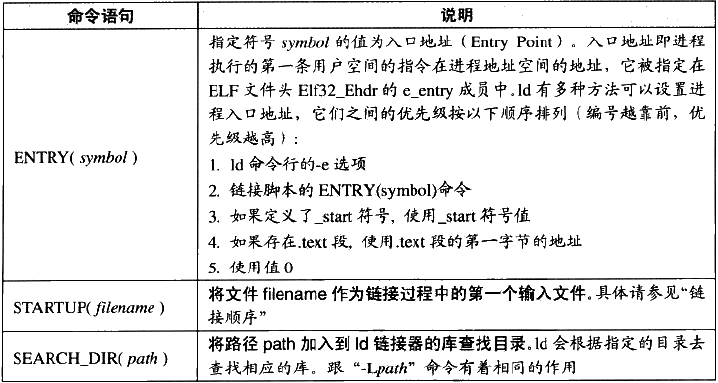


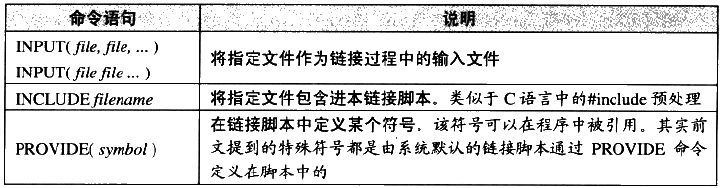


### ld链接脚本语法简介

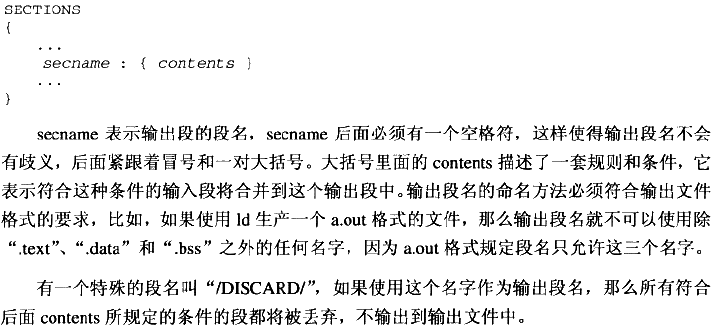


常用命令语句：

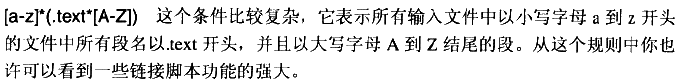




SECTIONS命令：



Contents规则中可以包含若干个条件，每个条件之间以空格隔开，如果输入段符合这些条件中的任意一个即表示这个输入段符合contents规则。如\*(.data)表示所有输入文件中的.data段符合条件，\*是通配符。在contents规则中可以使用正则表达式。

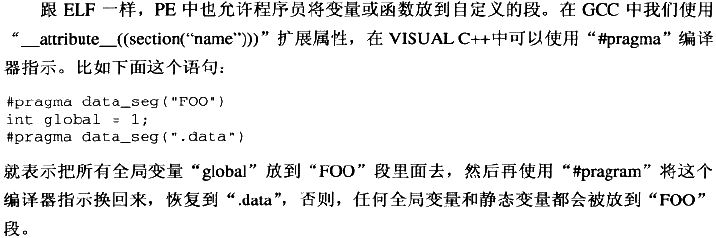


## BFD库

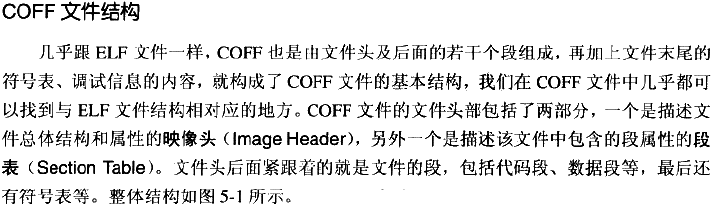
BFD库是一个GNU项目，他的目标是希望通过一种统一的接口来处理不同的目标文件格式。现在GAS（GNU汇编器）、链接器ld、调试器gdb及binutils的其他工具都通过BFD库来处理目标文件，而不是直接操作目标文件。

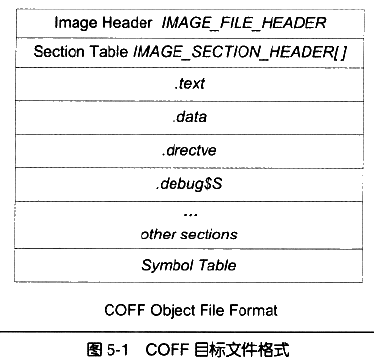
# Windows PE/COFF

## Windows的二进制文件格式PE/COFF



## PE的前身——COFF

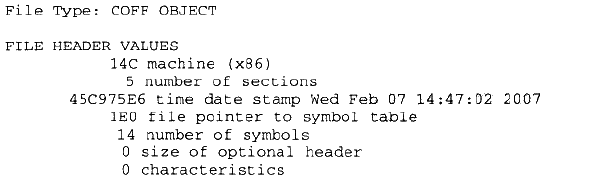


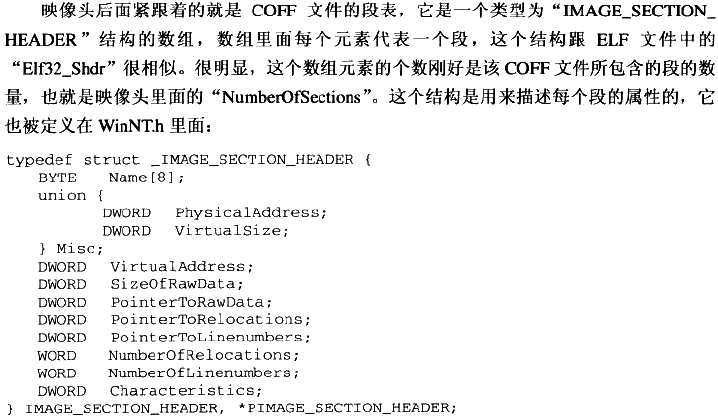


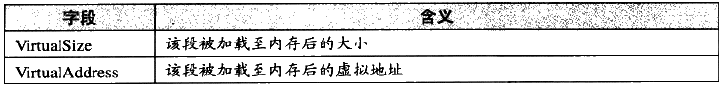
因为PE文件在装载时被直接映射到进程的虚拟空间中运行，他是进程的虚拟空间的映像，所以PE可执行文件很多时候被叫做映像文件。

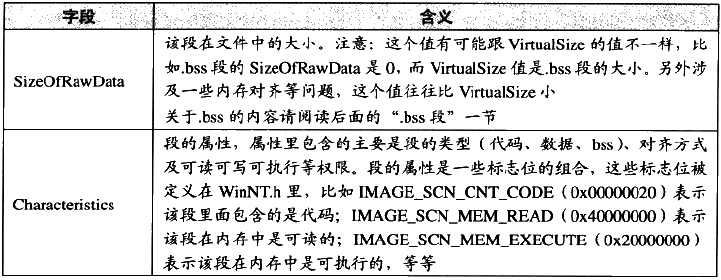


映像头中的成员和dumpbin输出的FILE\_HEADER\_VALUES中的内容是意一一对应的：

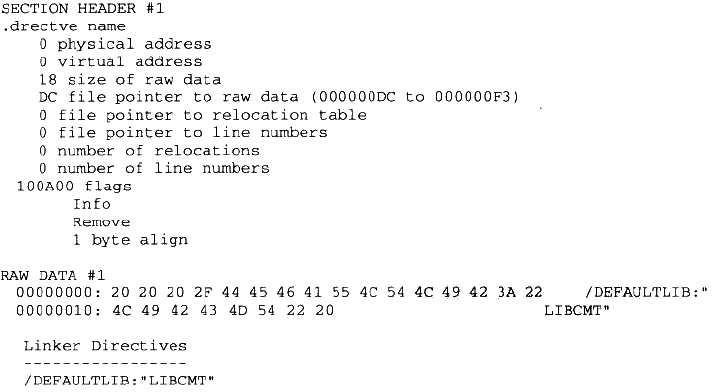




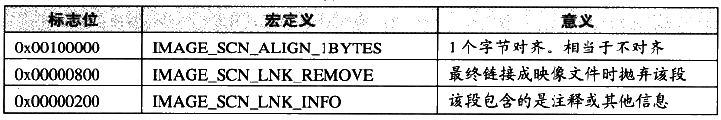


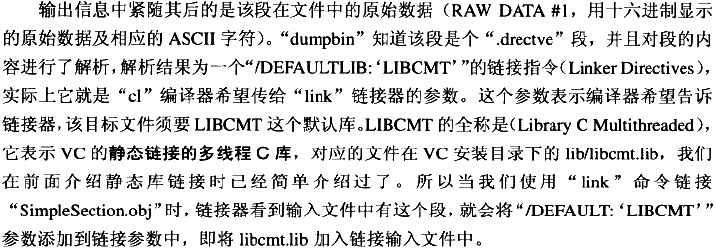


## 链接指信息

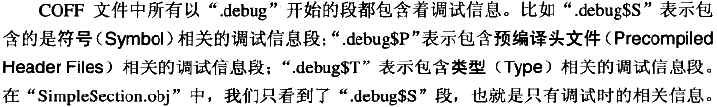


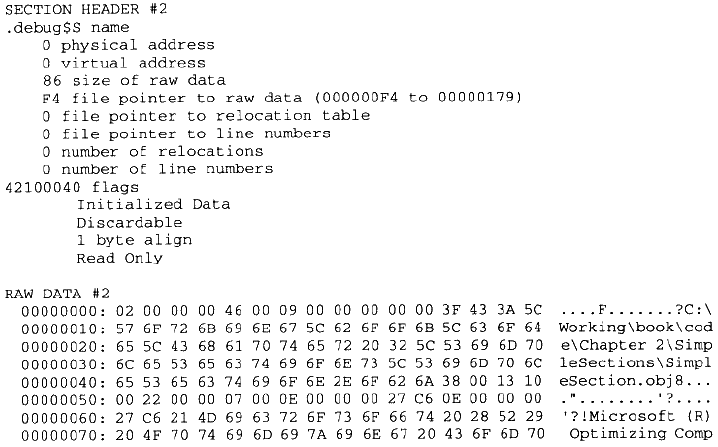
.drectve段的内容是编译器传递给链接器的指令，即编译器希望告诉链接器应该怎样链接这个目标文件。.drectve段的标志位为0x100A00，即IMAGE\_SECTION\_HEADERS里面的characteristics成员。



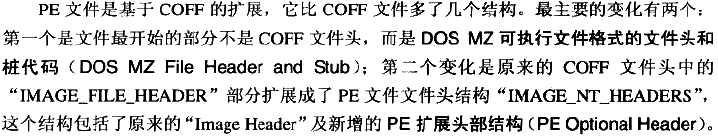


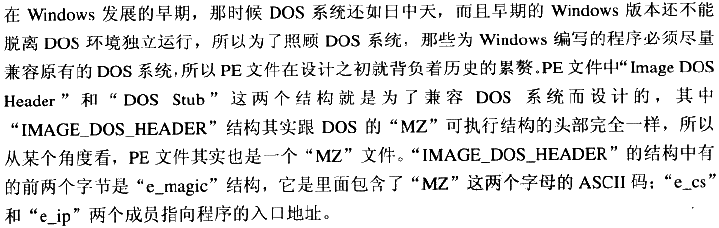
## 调试信息

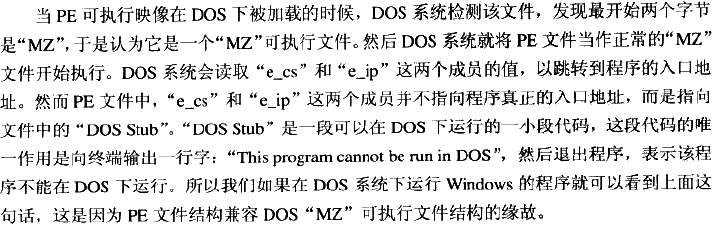


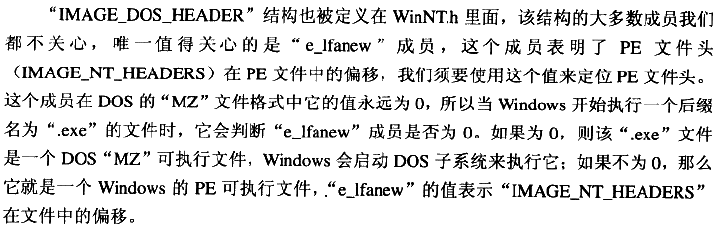


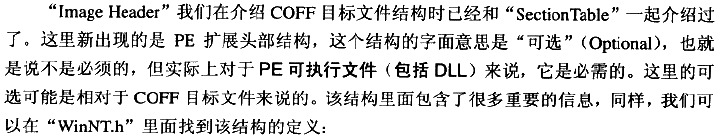
## Windows下的ELF——PE

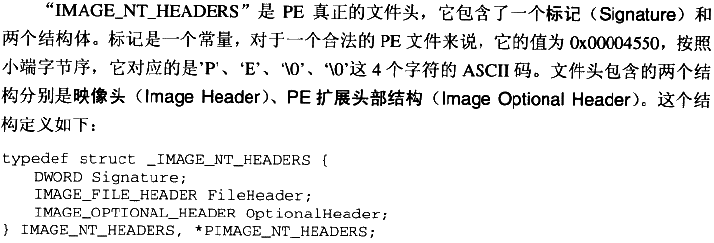


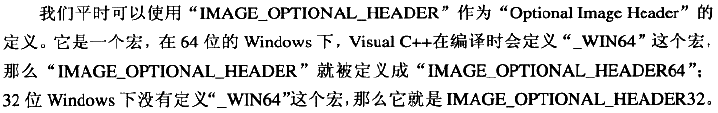




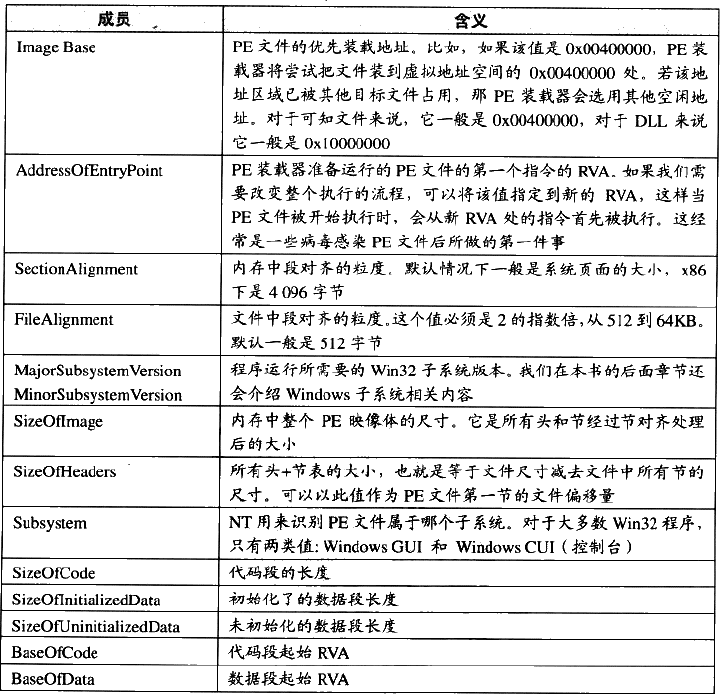






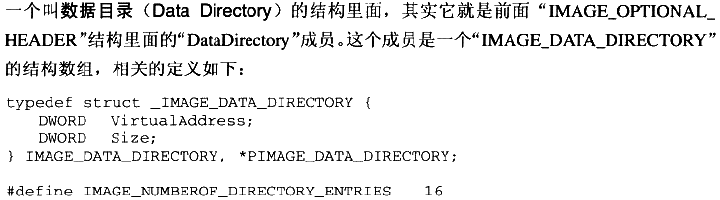


下面是PE扩展头中几个和装载相关的成员的含义：



### PE数据目录









# 可执行文件的装载与进程

## 进程虚拟地址空间

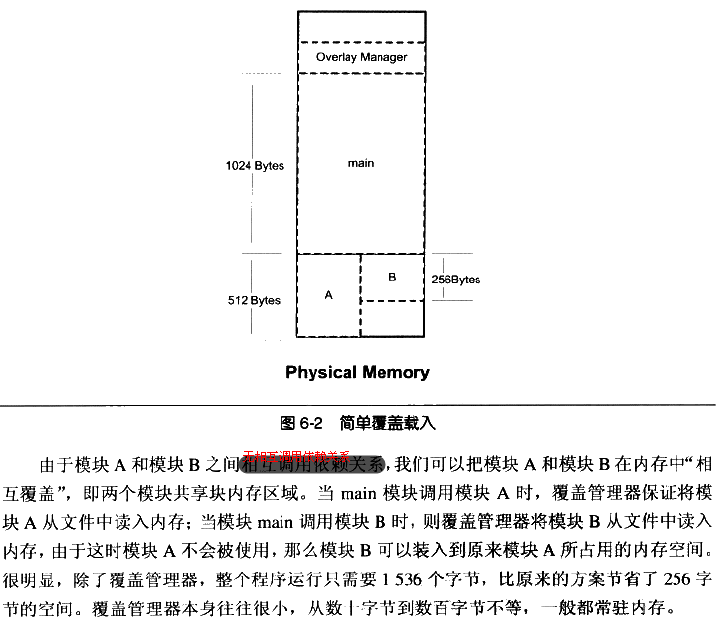
对于Linux操作系统，32位地址空间下，整个4G虚拟空间一般被划分成两部分，其中操作系统本身用去了一部分：从地址0xC0000000到0xFFFFFFFF，共1GB。剩下的从0x00000000到0xBFFFFFFF共3GB的空间都是留给进程使用的。

## 装载的方式

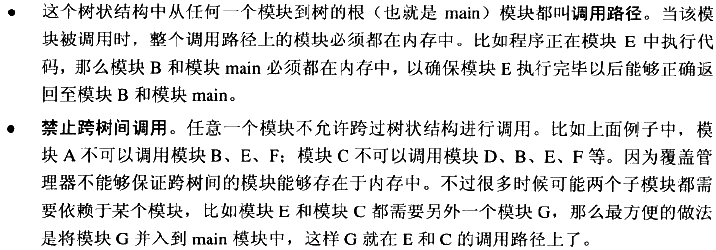
覆盖装入和页映射是两种典型的动态装载方法，他们都利用了程序的局部性原理。

### 覆盖装入

程序员在编写程序的时候必须手工将程序分割成若干块，然后编写一个小的辅助代码来管理这些模块的覆盖管理器。最简单的情况下，一个程序有一个主模块main，main分别会调用到模块A和模块B，但是A和B之间不能相互调用



在更复杂的调用关系场景下，程序员需要手工将模块按照他们之间的调用依赖关系组织成树状结构，而覆盖管理器需要保证两点：



### 页映射

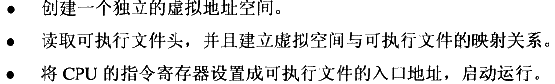
参见操作系统基本原理。

## 从操作系统角度看可执行文件的装载

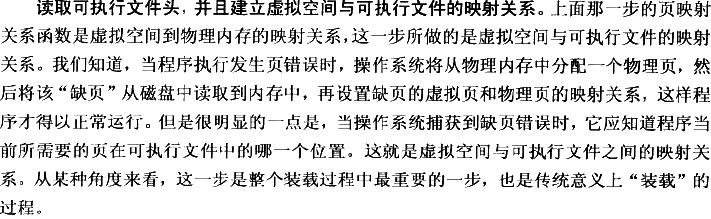
### 进程的建立

从操作系统的角度来看，一个进程最关键的特征是他拥有独立的虚拟地址空间，这使他有别于其他进程。

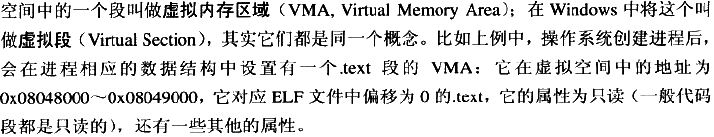
创建一个进程，然后装载相应的可执行文件并执行，在有虚拟内存的情况下，最开始只需要做三件事：



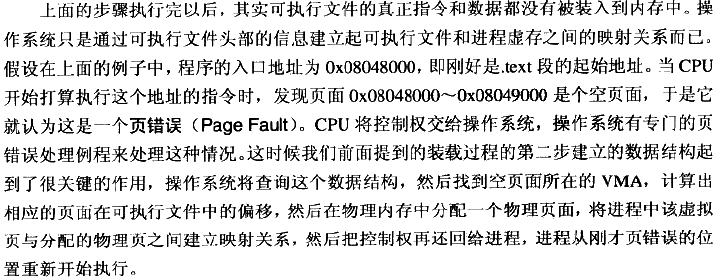
**创建虚拟地址空间**（建立虚拟空间到物理内存的映射关系）。一个虚拟空间由一组页映射函数将虚拟空间的各个页映射至相应的物理空间，那么创建一个虚拟空间实际并不是创建空间，而是创建映射函数所需要的相应的数据结构。在i386的Linux下，创建虚拟地址空间实际上只是分配一个页目录就可以了，映射关系可以等到后面程序发生页错误的时候再进行设置。



这种虚拟空间与可执行文件之间的映射关系保存在操作系统内部的一个数据结构中，操作系统保存这种结构是因为当程序发生段错误时，他可以通过查找这样的一个数据结构来定位错误页在可执行文件中的位置。Linux中将进程虚拟

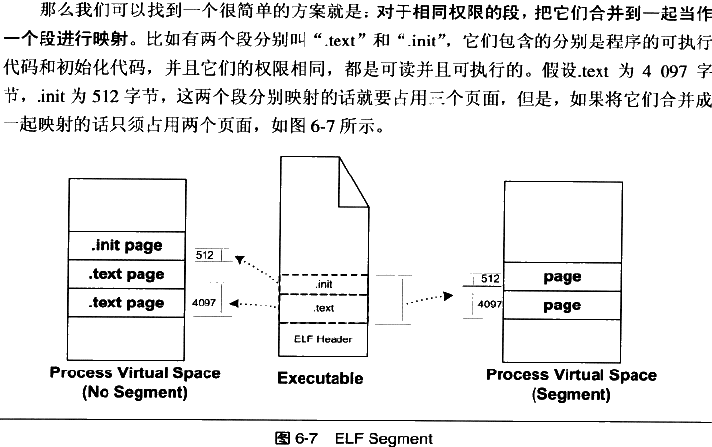


### 页错误

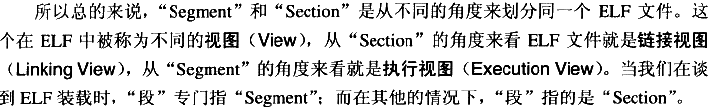


## 进程虚存空间分布

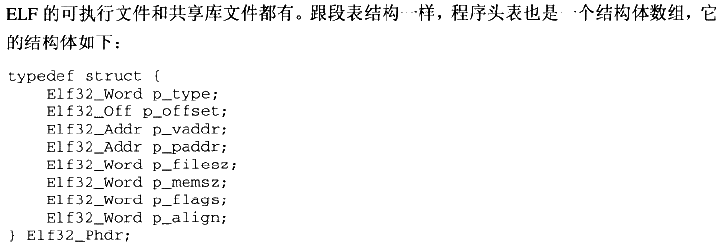
### ELF文件链接视图和执行视图



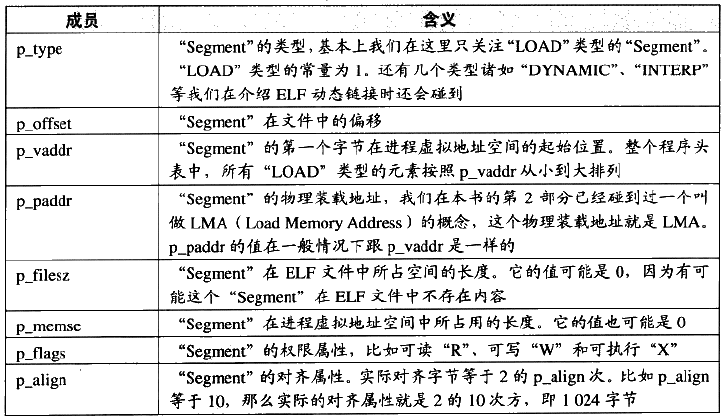
ELF可执行文件引入了“Segment”的概念，一个Segment包含一个或多个属性类似的Section，这样做的好处是可以明显减少页面内部碎片，从而节省内存空间。使用readelf –l命令可以查看ELF文件该如何被操作系统映射到进程的虚拟空间。

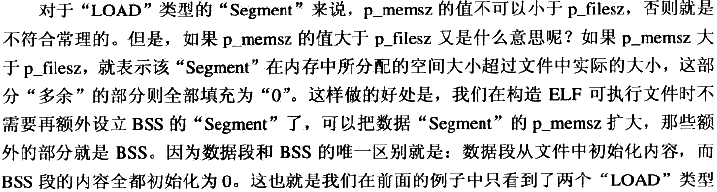






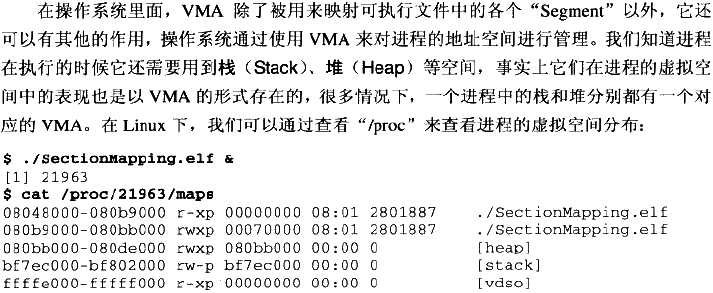
Elf32\_Phdr结构体的成员与readelf –l打印文件头表显示的结果一一对应。

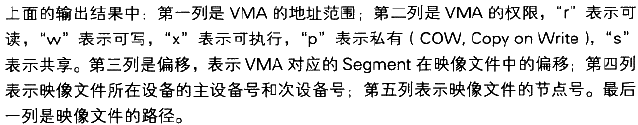


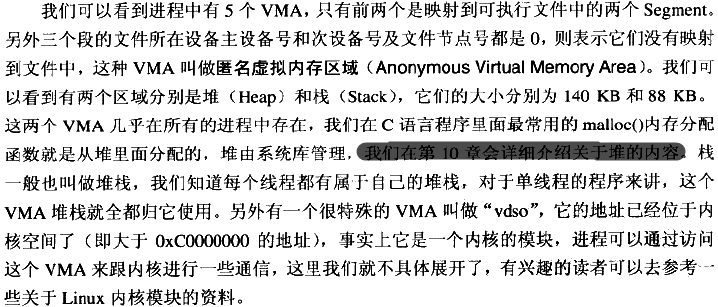




### 堆和栈

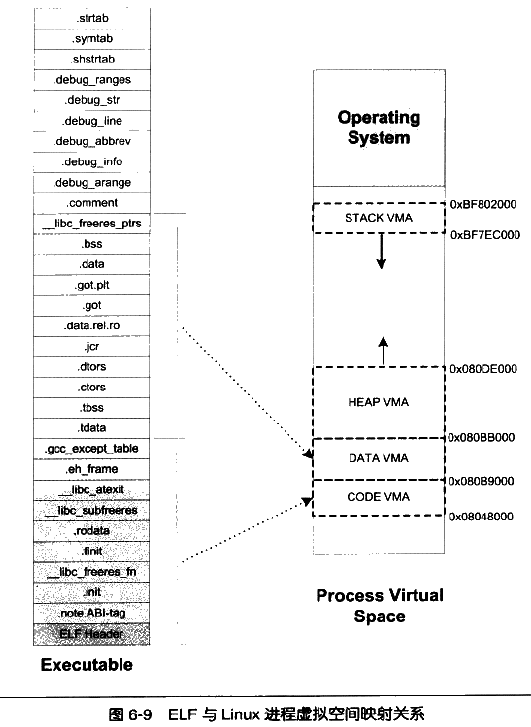






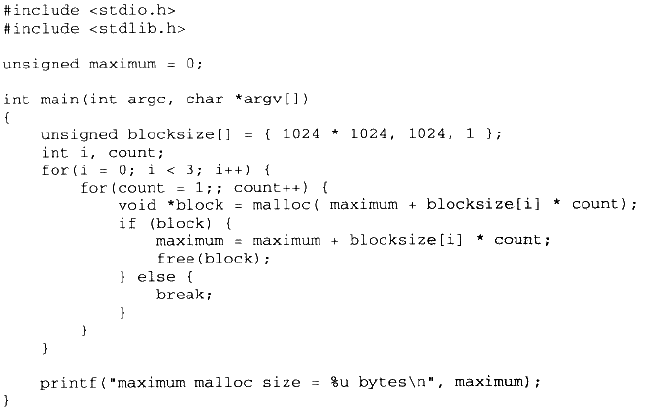
操作系统通过给进程空间划分出一个个VMA来管理进程的虚拟空间；基本原则是将相同权限属性的、有相同映像文件的Section映射成一个VMA；一个进程基本上可以分为如下几种VMA区域：



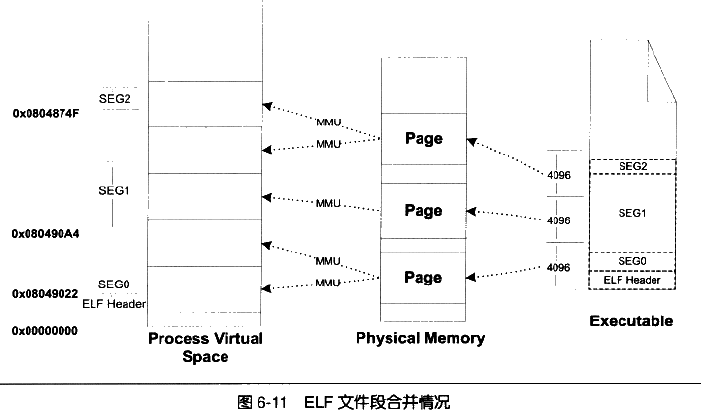


### 堆的最大申请数量

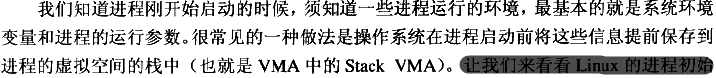
下面这段程序可以测试malloc最大内存申请数量。



### 段地址对齐

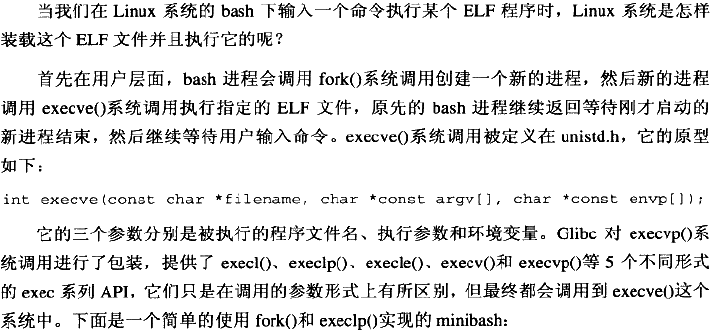


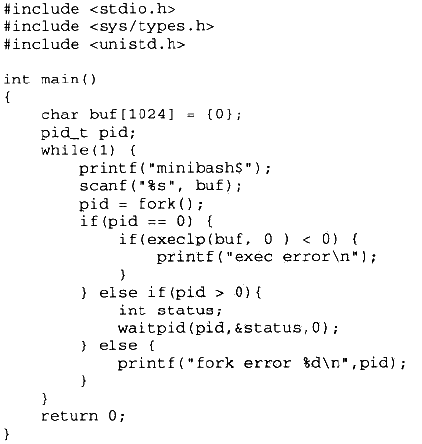
### 进程栈初始化

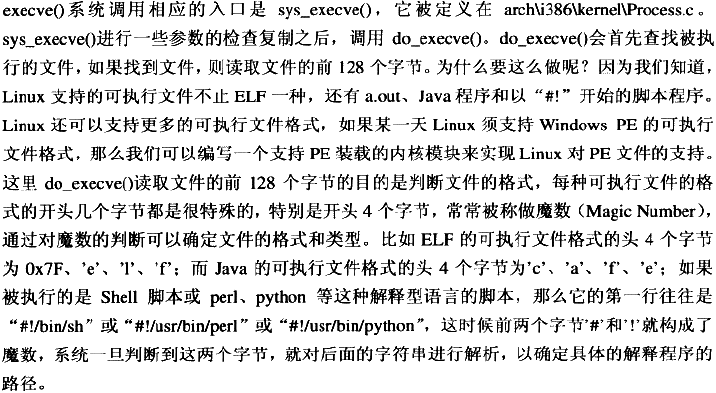


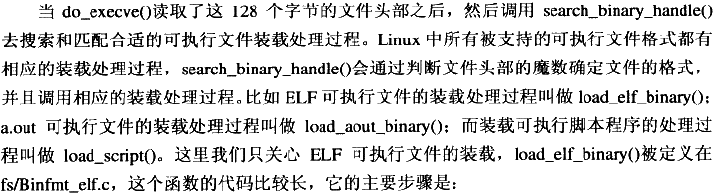
进程启动后，程序的库部分会把堆栈里的初始化信息中的参数信息传递给main()函数，也就是我们熟知的argc和argv两个参数。

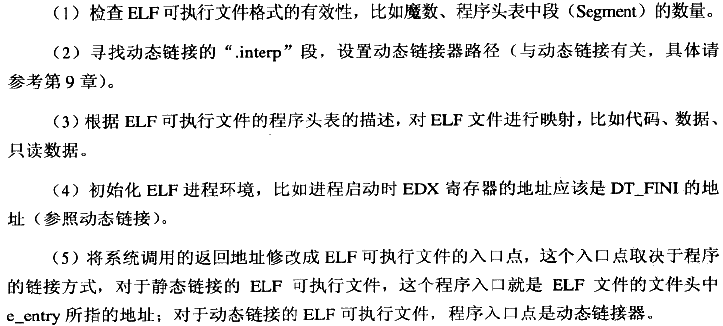
## Linux内核装载ELF过程简介



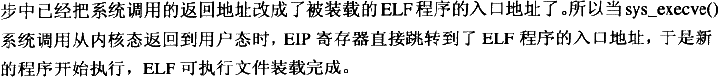




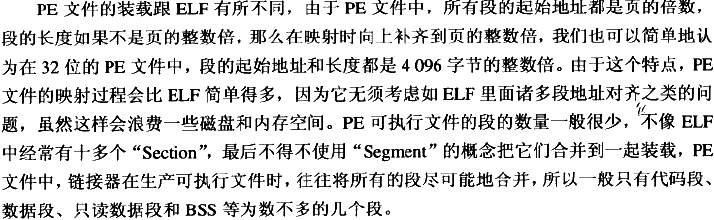


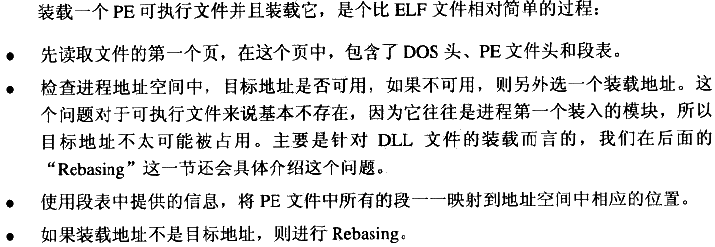


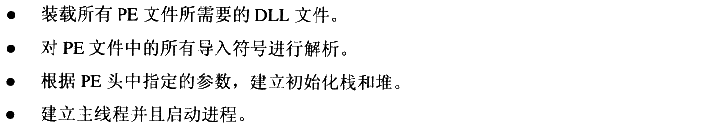




## Windows PE的装载





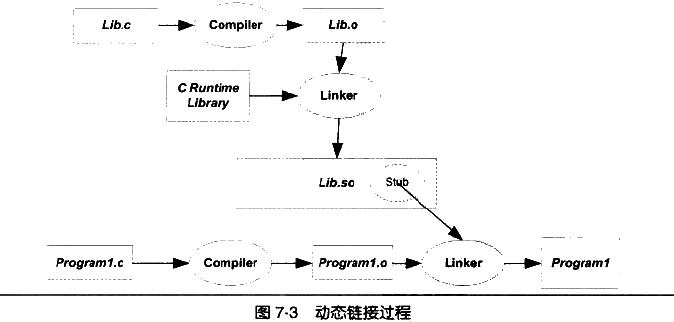


# 动态链接

## 5.1 为什么要动态链接

* 节省内存和磁盘空间
* 便于程序开发和发布
* 提高程序的可扩展性和兼容性

## 5.2 简单的动态链接例子



动态链接器和普通共享对象一样被影射到了进程的地址空间，在系统开始运行Program1之前，首先会把控制权交给动态链接器，由他完成所有的动态链接工作以后再把控制权交给Program1，然后开始执行。

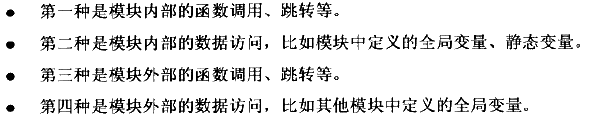
共享对象的最终装载地址在编译时是不确定的，而是在装载时，装载器根据当前地址空间的空闲情况，动态分配一块足够大小的虚拟地址空间给相应的共享对象。

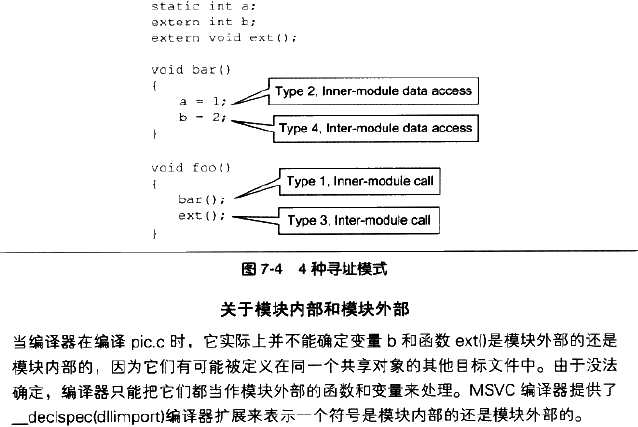
## 5.3 地址无关代码

共享对象在编译时不能假设自己在进程虚拟地址空间中的位置。

装载时重定位是解决动态模块中有绝对地址引用的办法之一，但是他有一个很大的缺点是指令部分无法在多个进程之间共享。而我们就是希望程序模块中共享的指令部分在装载时不需要因为装载地址的改变而改变，所以实现的基本想法就是把指令中那些需要被修改的部分分离出来，跟数据部分放在一起，这样指令部分就可以保持不变，而数据部分可以在每个进程中拥有一个副本。这种方案就是目前被称为地址无关代码的技术。

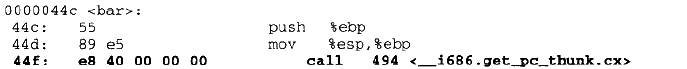
共享模块中四中地址引用分别为：

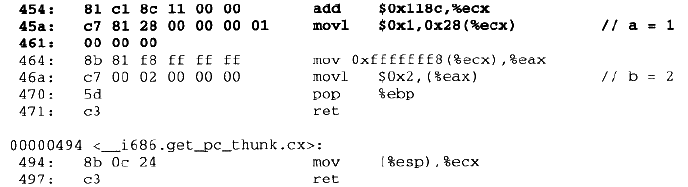




Type 1. 相对位置固定，无需重定位。

Type 2. 任何指令与他需要访问的模块内部数据间的相对位置是固定的，那么只需要相对于当前指令加上固定的偏移量就可以访问模块内部数据了。但是现代体系结构中，数据的相对寻址往往没有相对于当前指令地址(PC)的寻址方式，所以ELF采用了一个十分巧妙的方法得到当前PC值，然后再加上一个偏移量就可以达到访问相应变量的目的了。下面是现在ELF共享对象里使用的方法：

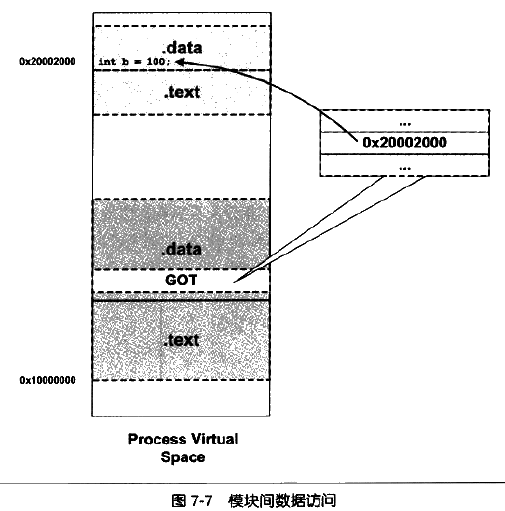


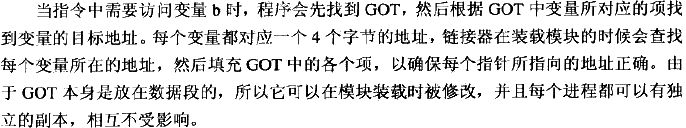


当处理器执行call指令后，下一条指令的地址会被压到栈顶，而esp就是栈顶指针寄存器，\_\_i686.get\_pc\_thunk.cx的作用就是把call的下一条指令地址值存放到ecx寄存器。

Type 3. 要使得代码地址无关，基本的思想就是把跟地址相关的部分放到数据段里。ELF的做法是在数据段里面建立一个指向这些变量的指针数组，也被称为全局偏移表(Global Offset Table, GOT)，当代码需要引用该全局变量时，可以通过GOT中相应的项间接引用。

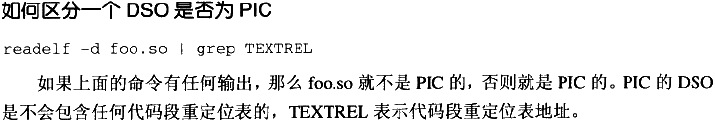
GOT做到指令的地址无关性也是通过类似Type2中的方法实现的



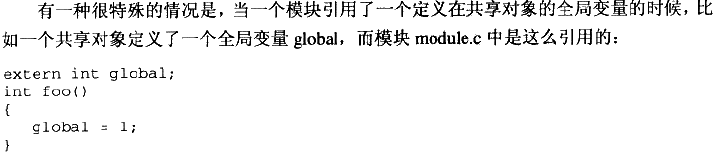


变量b的地址=（当前指令地址+GOT相对于当前指令的偏移+变量b在GOT中的偏移）中存放的4字节地址。“GOT相对于当前指令的偏移”和“变量b在GOT中的偏移”均在编译时确定。

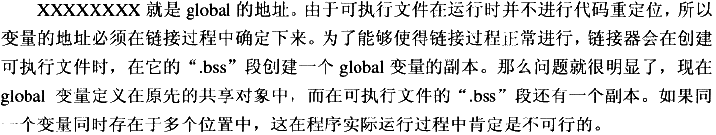
Type 4. 同样采用GOT的方式，但是其中相应的项保存的是目标函数的地址。这种方法很简单，但是存在一些性能问题，实际上ELF采用了一种更加复杂和精巧的方法。

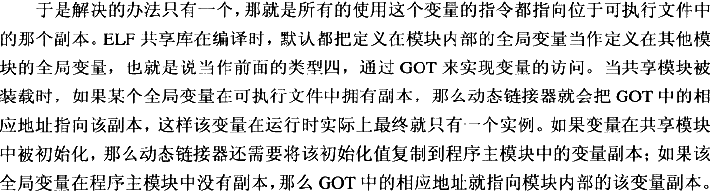


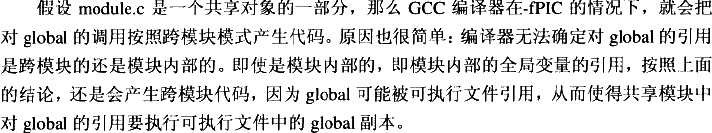
### 5.3.1 共享模块的全局变量问题



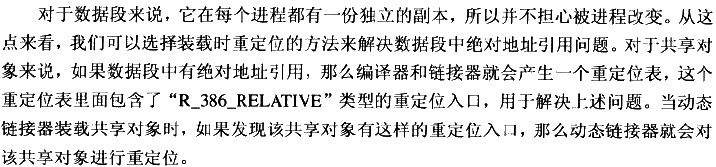
如果这个module.c是程序可执行文件的一部分，那么在这种情况下，由于程序主模块的代码并不是地址无关代码，那么他引用这个全局变量的方式和普通数据访问方式一样：movl $0x1, XXXXXXX。







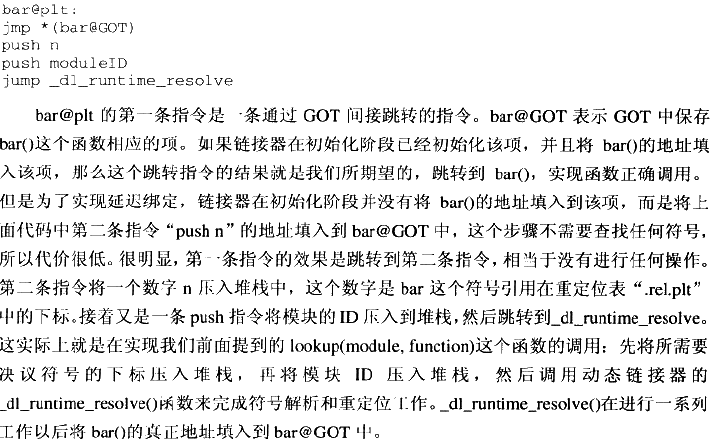
### 5.3.2 数据段地址无关性

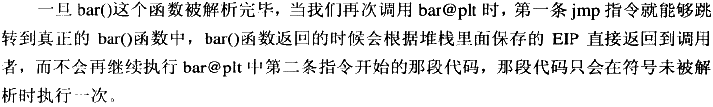


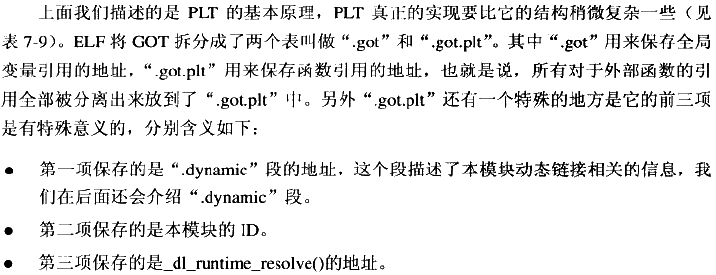
## 5.4 延迟绑定（PLT）

据统计ELF程序在静态链接下要比动态库快大约1%~5%，而动态链接比静态链接慢的主要原因主要在于动态链接下对于全局和静态的数据访问都要进行复杂的GOT定位，然后间接寻址；对于模块间的调用也要先定位GOT，然后再进行间接跳转。而且在程序启动时，动态链接器会进行链接工作，寻找并装载所需要的共享对象，然后进行符号查找地址重定位等工作。

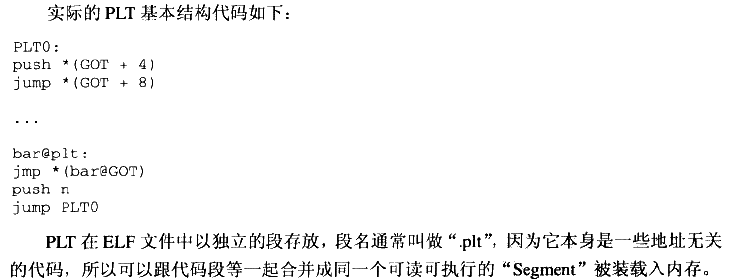
而延迟绑定是程序在启动时模块间的函数调用都没有进行绑定，而是在函数第一次被调用时才进行绑定。为了实现PLT，函数调用并不直接通过GOT跳转，而是通过一个叫做PLT项的结构来进行跳转。每个外部函数在PLT中都有一个对应的项，比如bar()函数在PLT中的项的地址我们称之为bar@plt。







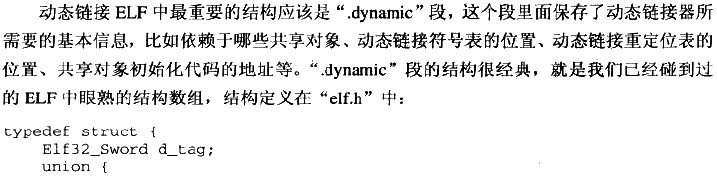
实际PLT的结构与上面例子中的稍有不同，为了减少代码重复，ELF把例子中的最后两条指令放到PLT的第二三项。



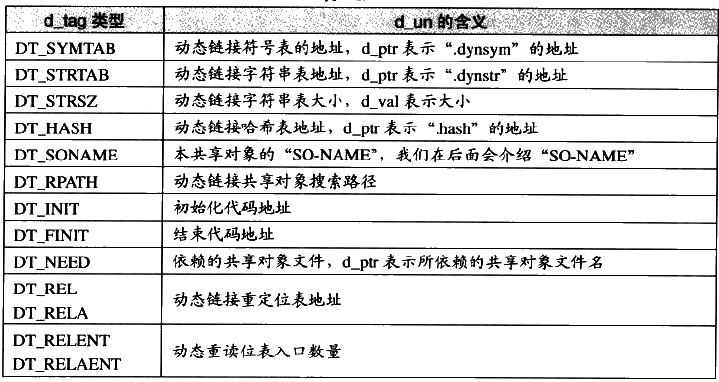
## 5.5 动态链接相关结构

.interp段：指定动态链接器位置。

.dynamic段：

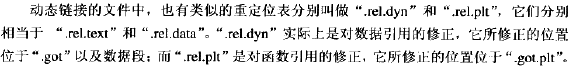






动态符号表

.rel.dyn/.rel.plt：动态链接重定位表



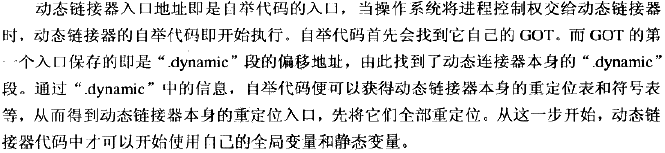
动态链接时进程堆栈初始化信息

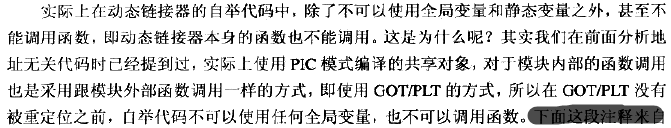
## 5.6 动态链接的步骤和实现

动态链接的步骤基本上分为3步：1.启动动态链接器本身；2.装载所有需要的共享对象；3.重定位和初始化。

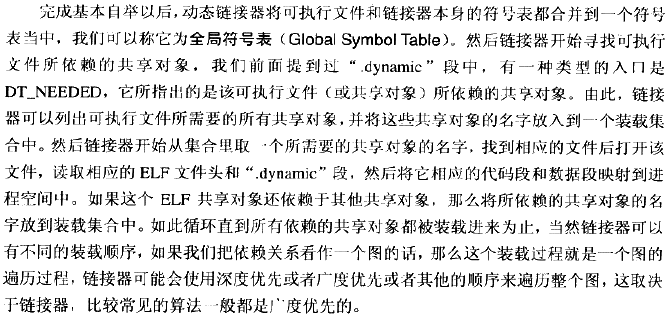
### 5.6.1 动态链接器自举

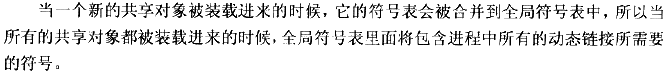
动态链接器本身不可以依赖于其他任何共享对象，且动态链接器本身所需要的全局和静态变量的重定位工作必须由其自身完成。





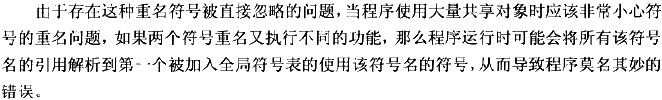
### 5.6.2 装载共享对象

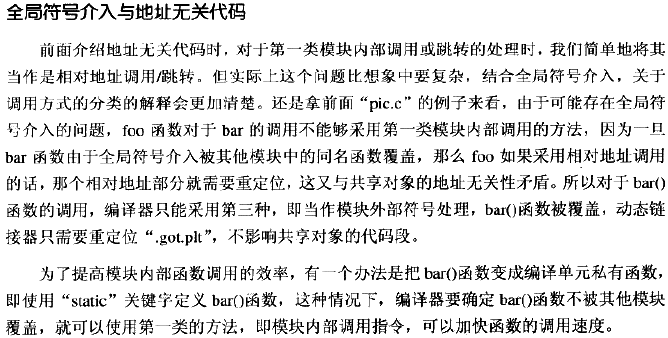




符号优先级（多个不同模块定义同一个符号时）：

实际上在Linux下的动态链接器是这样处理的：当一个符号需要被加入全局符号表时，如果相同的符号名已经存在，则后加入的符号被忽略。且很明显这与动态链接器的装载顺序相关。





### 5.6.3 重定位和初始化

上面的步骤完成后，链接器开始重新遍历可执行文件和每个共享对象的重定位表，将他们的GOT/PLT中的每个需要重定位的位置进行修正。

重定位完成之后，如果某个共享对象有“.init”段，那么动态链接器会执行该段中的代码，用以实现共享对象特有的初始化过程。可执行文件中的.init段不会在这里被执行。

### 5.6.4 Linux动态链接器实现

暂略

# 运行库

## 入口函数和程序初始化

# X 其他

ld –e ：指定程序入口，默认入口为\_start。