目标文件格式概述

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

回顾：一个C语言程序举例

main.c

```
int buf[2] = {1, 2};
void swap();

int main()
{
  swap();
  return 0;
}
```

swap.c

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
   int temp;
   bufp1 = &buf[1];
   temp = *bufp0;
   *bufp0 = *bufp1;
   *bufp1 = temp;
}
```

每个模块有自己的代码、数据（初始化全局变量、未初始化全局变量，静态变量、局部变量）

局部变量temp分配在栈中，不会在过程外被引用，因此不是符号定义

回顾：可执行文件的生成

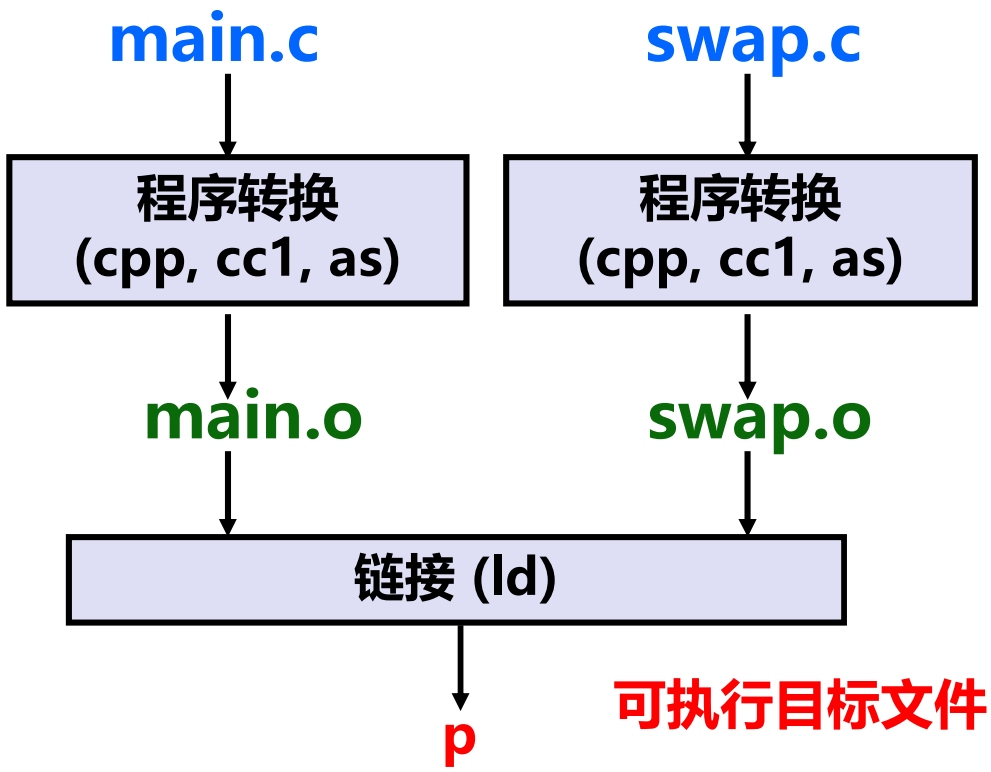
- 使用GCC编译器编译并链接生成可执行程序P:
  - $ gcc -O2 -g -o p main.c swap.c
  - $ ./p

-O2：2级优化

-g：生成调试信息

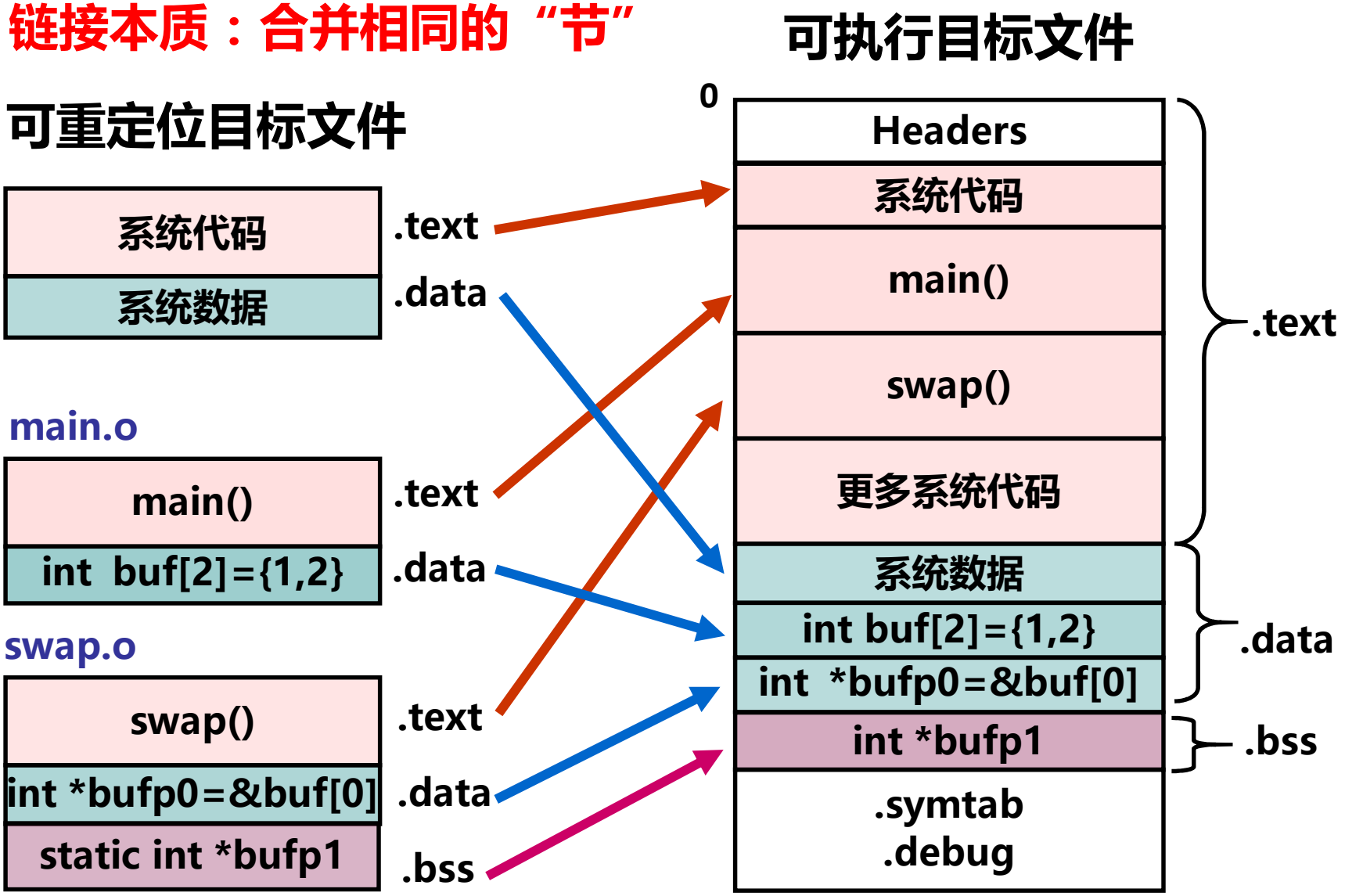
-o：目标文件名

GCC编译器的静态链接过程



回顾：链接过程的本质

链接本质：合并相同的“节”



链接操作的步骤

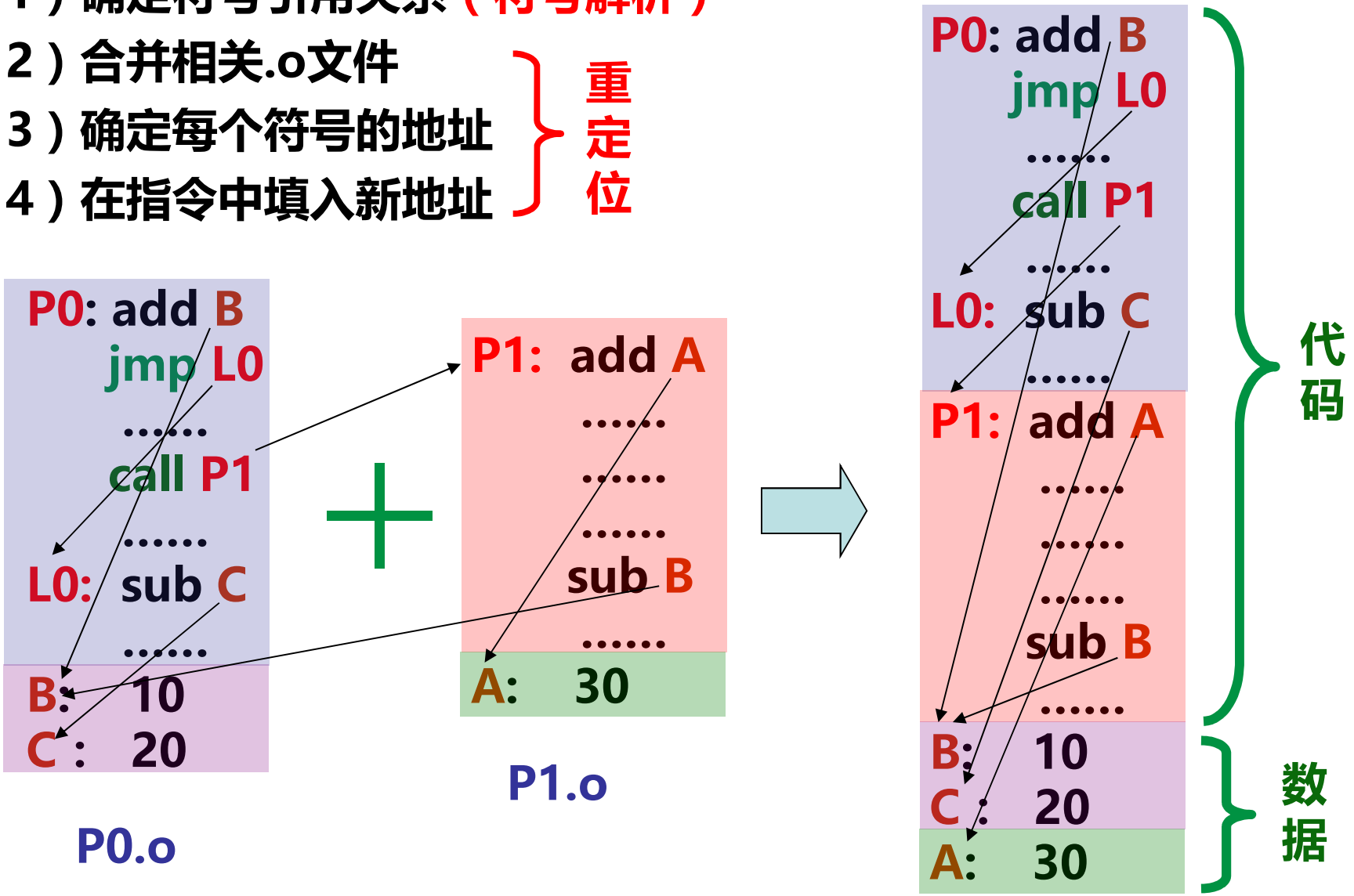
1）确定符号引用关系（符号解析）

2）合并相关.o文件

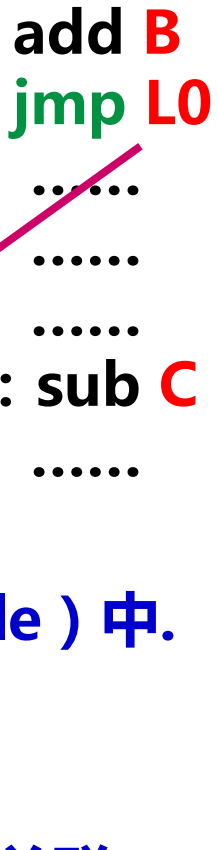
3）确定每个符号的地址

4）在指令中填入新地址

重定位



链接操作的步骤



- Step 1. 符号解析（Symbol resolution）
  - 程序中有定义和引用的符号 (包括变量和函数等)
    - void swap() {…} /* 定义符号swap */
    - swap(); /* 引用符号swap */
    - int *xp = &x; /* 定义符号 xp, 引用符号 x */
  - 编译器将定义的符号存放在一个符号表（ symbol table ）中.
    - 符号表是一个结构数组
    - 每个表项包含符号名、长度和位置等信息
  - 链接器将每个符号的引用都与一个确定的符号定义建立关联
- Step 2. 重定位
  - 将多个代码段与数据段分别合并为一个单独的代码段和数据段
  - 计算每个定义的符号在虚拟地址空间中的绝对地址
  - 将可执行文件中符号引用处的地址修改为重定位后的地址信息

三类目标文件

- 可重定位目标文件 (.o)
  - 其代码和数据可和其他可重定位文件合并为可执行文件
    - 每个.o 文件由对应的.c文件生成
    - 每个.o文件代码和数据地址都从0开始
- 可执行目标文件 (默认为a.out)
  - 包含的代码和数据可以被直接复制到内存并被执行
  - 代码和数据地址为虚拟地址空间中的地址
- 共享的目标文件 (.so)
  - 特殊的可重定位目标文件，能在装入或运行时被装入到内存并自动被链接，称为共享库文件
  - Windows 中称其为 *Dynamic Link Libraries* (DLLs)

目标文件

```
/* main.c */
int add(int, int);
int main( )
{
   return add(20, 13);
}
```

```
/* test.c */
int add(int i, int j)
{
    int x = i + j;
    return x;
}
```

objdump -d test.o

```
00000000 <add>:
       0:   55              push  %ebp
       1:   89 e5           mov   %esp, %ebp
       3:   83 ec 10        sub   $0x10, %esp
       6:   8b 45 0c        mov    0xc(%ebp), %eax
       9:   8b 55 08        mov    0x8(%ebp), %edx
       c:   8d 04 02        lea    (%edx,%eax,1), %eax
       f:   89 45 fc        mov    %eax, -0x4(%ebp)
      12:   8b 45 fc        mov    -0x4(%ebp), %eax
      15:   c9              leave
      16:   c3              ret
```

objdump -d test

```
080483d4 <add>:
 80483d4:   55              push  %ebp
 80483d5:   89 e5           mov   %esp, %ebp
 80483d7:   83 ec 10        sub   $0x10, %esp
 80483da:   8b 45 0c        mov    0xc(%ebp), %eax
 80483dd:   8b 55 08        mov    0x8(%ebp), %edx
 80483e0:   8d 04 02        lea    (%edx,%eax,1), %eax
 80483e3:   89 45 fc        mov    %eax, -0x4(%ebp)
 80483e6:   8b 45 fc        mov    -0x4(%ebp), %eax
 80483e9:   c9              leave
 80483ea:   c3              ret
```

目标文件的格式

- **目标代码（Object Code）指编译器和汇编器处理源代码后所生成的机器语言目标代码**
- **目标文件（Object File）指包含目标代码的文件**
- **最早的目标文件格式是自有格式，非标准的**
- **标准的几种目标文件格式**
  - **DOS操作系统（最简单）：COM格式，文件中仅包含代码和数据，且被加载到固定位置**
  - **System V UNIX早期版本：COFF格式，文件中不仅包含代码和数据，还包含重定位信息、调试信息、符号表等其他信息，由一组严格定义的数据结构序列组成**
  - **Windows： PE格式（COFF的变种），称为可移植可执行（Portable Executable，简称PE）**
  - **Linux等类UNIX：ELF格式（COFF的变种），称为可执行可链接（Executable and Linkable Format，简称ELF）**

Executable and Linkable Format (ELF)

- **两种视图**
  - **链接视图（被链接）：可重定位目标文件 (Relocatable object files)**
  - **执行视图（被执行）：可执行目标文件（Executable object files）**

链接视图

节（section）是ELF 文件中具有相同特征的最小可处理单位

.text节: 代码

.data节: 数据

.rodata: 只读数据

.bss: 未初始化数据

执行视图

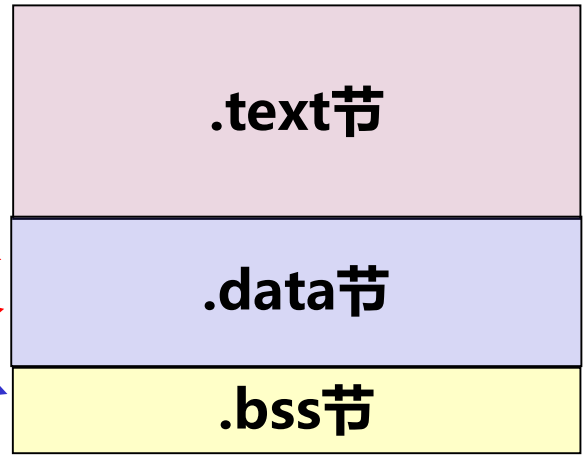
由不同的段（segment）组成，描述节如何映射到存储段中，可多个节映射到同一段，如：可合并.data节和.bss节,并映射到一个可读可写数据段中

链接视图—可重定位目标文件

- 可被链接（合并）生成可执行文件或**共享目标文件**
- **静态链接库文件**由若干个可重定位目标文件组成
- 包含代码、数据（已初始化.data和未初始化.bss）
- 包含**重定位信息**（指出哪些符号引用处需要重定位）
- 文件扩展名为.o（相当于Windows中的 .obj文件）

```
int x=100;
int y;
void prn(int n)
{
   printf( "%d\n" ,n);
}

void main( )
{
    static int a=1;
    static int b;
    int i=200,j;
    prn(x+a+i);
}
```



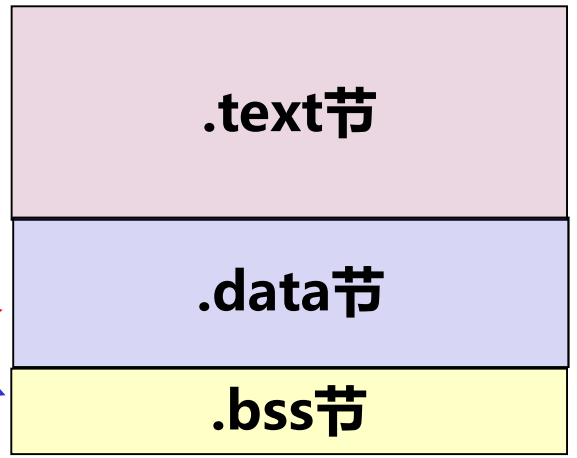
为了**进行链接**，还需要其他许多信息，如符号表、重定位信息等许多其他的节（Section）

执行视图—可执行目标文件

- 包含代码、数据（已初始化.data和未初始化.bss）
- 定义的所有变量和函数已有确定地址（虚拟地址空间中的地址）
- 符号引用处已被重定位，以指向所引用的定义符号
- 没有文件扩展名或默认为a.out（相当于Windows中的 .exe文件）
- 可被CPU直接执行，指令地址和指令给出的操作数地址都是虚拟地址

```
int x=100;
int y;
void prn(int n)
{
   printf( "%d\n" ,n);
}

void main( )
{
    static int a=1;
    static int b;
    int i=200,j;
    prn(x+a+i);
}
```



为了能执行，还需将具相同访问属性的节合并成段（Segment），并说明每个段的属性，如：在可执行文件中的位移、大小、在虚拟空间中的位置、对齐方式、访问属性等

ELF可重定位目标文件

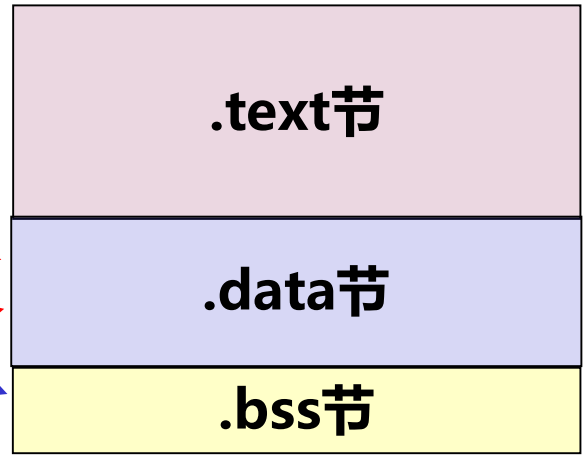
南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

回顾：链接视图—可重定位目标文件

- 可被链接（合并）生成可执行文件或共享目标文件
- 静态链接库文件由若干个可重定位目标文件组成
- 包含代码、数据（已初始化全局变量和局部静态变量.data和未初始化的全局变量和局部静态变量.bss）
- 包含重定位信息（指出哪些符号引用处需要重定位）
- 文件扩展名为.o（相当于Windows中的 .obj文件）

```
int x=100;
int y;
void prn(int n)
{
   printf( "%d\n" ,n);
}

void main( )
{
    static int a=1;
    static int b;
    int i=200,j;
    prn(x+a+i);
}
```



为了进行链接，还需要其他许多信息，如符号表、重定位信息等许多其他的节（Section）

未初始化变量（.bss节）

- C语言规定：
  - 未初始化的全局变量和局部静态变量的默认初始值为0
- 将未初始化变量（.bss节）与已初始化变量（.data节）分开的好处
  - .data节中存放具体的初始值，需要占磁盘空间
  - .bss节中无需存放初始值，只要说明.bss中的每个变量将来在执行时占用几个字节即可，因此，.bss节实际上不占用磁盘空间，提高了磁盘空间利用率
- BSS（Block Started by Symbol）最初是UA-SAP汇编程序中所用的一个伪指令，用于为符号预留一块内存空间
- 所有未初始化的全局变量和局部静态变量都被汇总到.bss节中，通过专门的“节头表（Section header table）”来说明应该为.bss节预留多大的空间

可重定位目标文件格式

ELF 头

- ✓ 包括16字节标识信息、文件类型 (.o, exec, .so)、机器类型（如 IA-32）、节头表的偏移、节头表的表项大小以及表项个数

.text 节

- ✓ 编译后的代码部分

.rodata 节

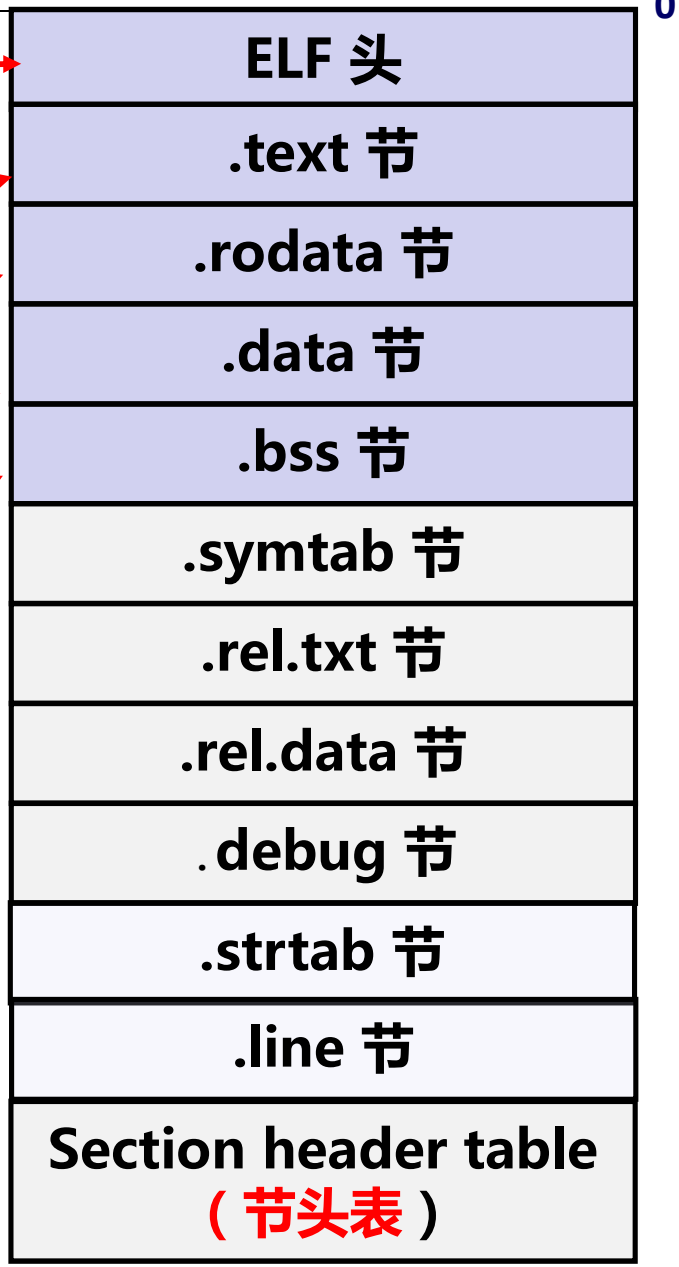
- ✓ 只读数据，如 printf 格式串、switch 跳转表等

.data 节

- ✓ 已初始化的全局变量

.bss 节

- ✓ 未初始化全局变量，仅是占位符，不占据任何实际磁盘空间。区分初始化和非初始化是为了空间效率



switch-case语句举例

```
int sw_test(int a, int b, int c)
{
   int result;
   switch(a) {
   case 15:
      c=b&0x0f;
   case 10:
      result=c+50;
      break;
   case 12:
   case 17:
      result=b+50;
      break;
   case 14:
      result=b
      break;
   default:
      result=a;
   }
   return result;
}
```

```
  movl  8(%ebp), %eax
  subl  $10, %eax
  cmpl  $7, %eax
  ja    .L5
  jmp     *.L8( , %eax, 4)
.L1:
  movl  12(%ebp), %eax
  andl  $15, %eax
  movl  %eax, 16(%ebp)
.L2:
  movl  16(%ebp), %eax
  addl    $50, %eax
  jmp  .L7
.L3:
  movl  12(%ebp), %eax
  addl    $50, %eax
   jmp    .L7
.L4:
   movl  12(%ebp), %eax
   jmp  .L7
.L5:
   addl  $10, %eax
.L7:
```

R[eax]=a-10=i

if (a-10)>7 转 L5

转.L8+4*i 处的地址

跳转表在目标文件的只读数据节中，按4字节边界对齐

```
   .section   .rodata
   .align 4
.L8                    a=
   .long    .L2    10
   .long    .L5    11
   .long    .L3    12
   .long    .L5    13
   .long    .L4    14
   .long    .L1    15
   .long    .L5    16
   .long    .L3    17
```

可重定位目标文件格式

.symtab 节

- ✓ 存放函数和全局变量（符号表）信息，它不包括局部变量

.rel.text 节

- ✓ .text节的重定位信息，用于重新修改代码段的指令中的地址信息

.rel.data 节

- ✓ .data节的重定位信息，用于对被模块使用或定义的全局变量进行重定位的信息

.debug 节

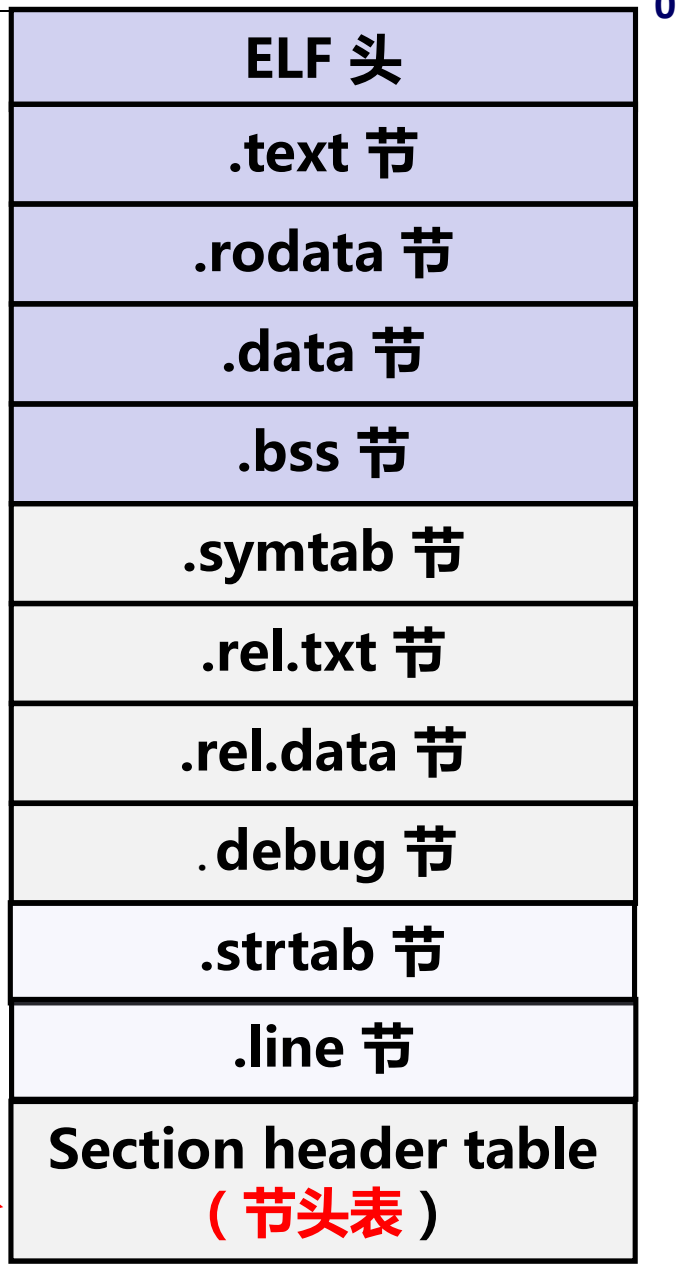
- ✓ 调试用符号表 (gcc -g)

strtab 节

- ✓ 包含symtab和debug节中符号及节名

Section header table（节头表）

- ✓ 每个节的节名、偏移和大小



ELF头（ELF Header）

- ELF头位于ELF文件开始，包含文件结构说明信息。分32位系统对应结构和64位系统对应结构（32位版本、64位版本）
- 以下是32位系统对应的数据结构

```
#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;
```

定义了ELF魔数、版本、小端/大端、操作系统平台、目标文件的类型、机器结构类型、程序执行的入口地址、程序头表（段头表）的起始位置和长度、节头表的起始位置和长度等

魔数：文件开头几个字节通常用来确定文件的类型或格式

a.out的魔数：01H 07H

PE格式魔数：4DH 5AH

加载或读取文件时，可用魔数确认文件类型是否正确

ELF头信息举例

```
$ readelf -h main.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 0
  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:          516 (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:         15
  Section header string table index: 12
```

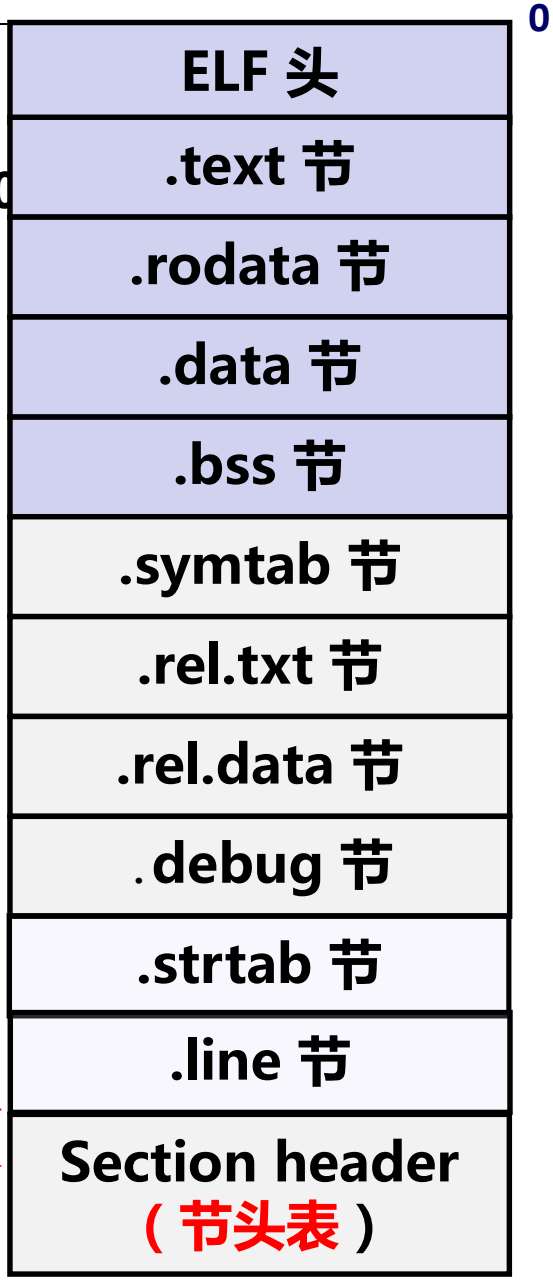
可重定位目标文件的ELF头

ELF文件的魔数

没有程序头表

15x40B

.strtab在节头表中的索引



节头表（Section Header Table）

- 除ELF头之外，节头表是ELF可重定位目标文件中最重要的部分内容
- 描述每个节的节名、在文件中的偏移、大小、访问属性、对齐方式等
- 以下是32位系统对应的数据结构（每个表项占40B）

```
typedef struct {
    Elf32_Word      sh_name;    节名字符串在.strtab中的偏移
    Elf32_Word      sh_type;    节类型：无效/代码或数据/符号/字符串/...
    Elf32_Word      sh_flags;   节标志：该节在虚拟空间中的访问属性
    Elf32_Addr      sh_addr;    虚拟地址：若可被加载，则对应虚拟地址
    Elf32_Off       sh_offset;  在文件中的偏移地址，对.bss节而言则无意义
    Elf32_Word      sh_size;    节在文件中所占的长度
    Elf32_Word      sh_link;    sh_link和sh_info用于与链接相关的节（如
    Elf32_Word      sh_info;    .rel.text节、.rel.data节、.symtab节等）
    Elf32_Word      sh_addralign;  节的对齐要求
    Elf32_Word      sh_entsize; 节中每个表项的长度，0表示无固定长度表项
} Elf32_Shdr;
```

节头表信息举例

$ readelf -S test.o

There are 11 section headers, starting at offset 0x120:

Section Headers:

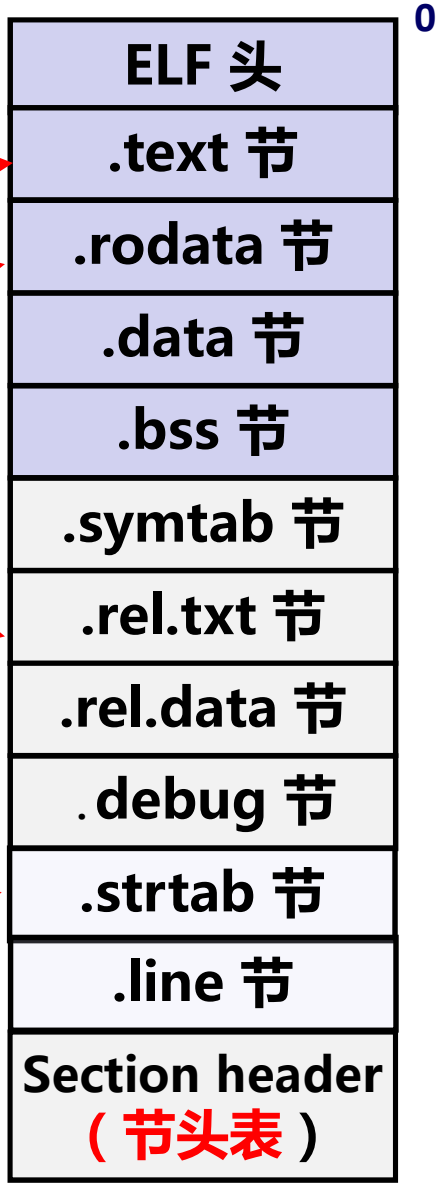
[Nr] Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]	NULL	00000000	000000	000000	00		0	0	0
[1] .text	PROGBITS	00000000	000034	00005b	00	AX	0	0	4
[2] .rel.text	REL	00000000	000498	000028	08		9	1	4
[3] .data	PROGBITS	00000000	000090	00000c	00	WA	0	0	4
[4] .bss	NOBITS	00000000	00009c	00000c	00	WA	0	0	4
[5] .rodata	PROGBITS	00000000	00009c	000004	00	A	0	0	1
[6] .comment	PROGBITS	00000000	0000a0	00002e	00		0	0	1
[7] .note.GNU-stack	PROGBITS	00000000	0000ce	000000	00		0	0	1
[8] .shstrtab	STRTAB	00000000	0000ce	000051	00		0	0	1
[9] .symtab	SYMTAB	00000000	0002d8	000120	10		10	13	4
[10] .strtab	STRTAB	00000000	0003f8	00009e	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)

I (info), L (link order), G (group), x (unknown)

O (extra OS processing required) o (OS specific), p (processor specific)



可重定位目标文件中，每个可装入节的起始地址总是0

节头表信息举例

$ readelf -S test.o

There are 11 section headers, starting at offset 0x120:

Section Headers:

[Nr] Name	Off	Size	ES	Flg	Lk	Inf	Al
[0]	000000	000000	00		0	0	0
[1] .text	000034	00005b	00	AX	0	0	4
[2] .rel.text	000498	000028	08		9	1	4
[3] .data	000090	00000c	00	WA	0	0	4
[4] .bss	00009c	00000c	00	WA	0	0	4
[5] .rodata	00009c	000004	00	A	0	0	1
[6] .comment	0000a0	00002e	00		0	0	1
[7] .note.GNU-stack	0000ce	000000	00		0	0	1
[8] .shstrtab	0000ce	000051	00		0	0	1
[9] .symtab	0002d8	000120	10		10	13	4
[10] .strtab	0003f8	00009e	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)

I (info), L (link order), G (group), x (unknown)

……

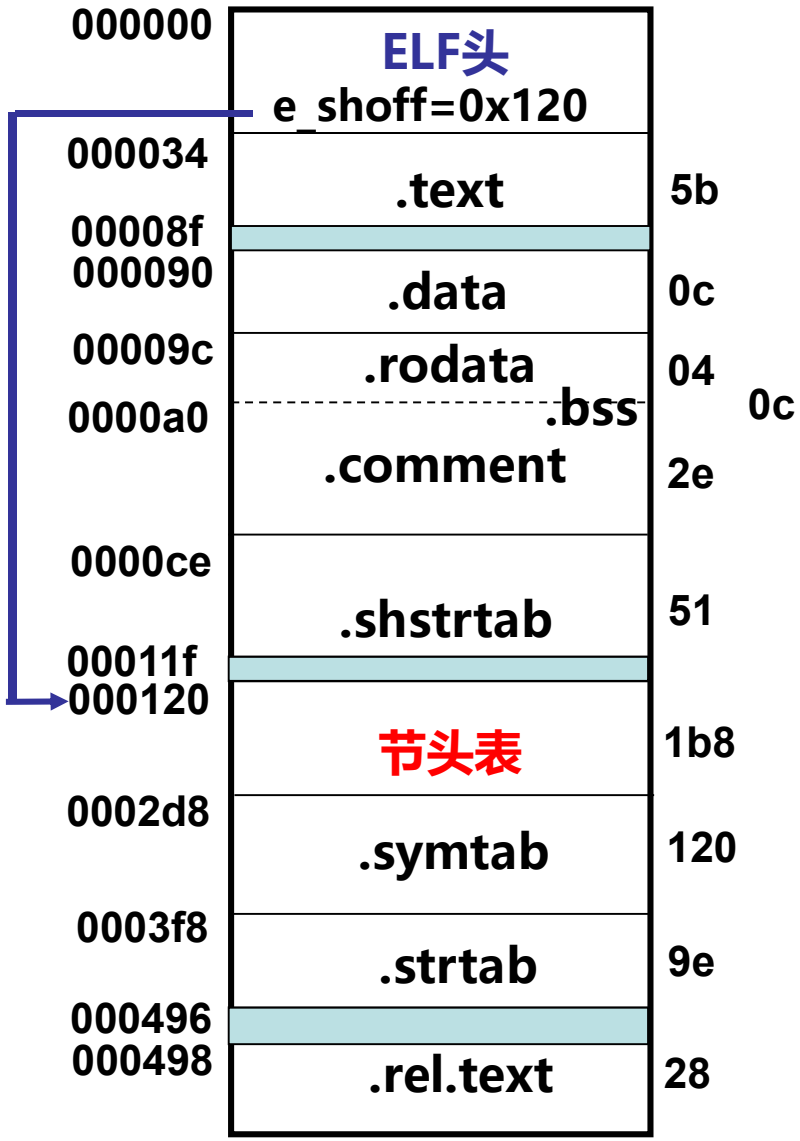
有4个节将会分配存储空间

.text：可执行

.data和.bss：可读可写

.rodata：可读

可重定位目标文件test.o的结构



ELF可执行目标文件

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

回顾：可重定位目标文件格式

ELF 头

- ✓ 包括16字节标识信息、文件类型 (.o, exec, .so)、机器类型（如 IA-32）、节头表的偏移、节头表的表项大小以及表项个数

.text 节

- ✓ 编译后的代码部分

.rodata 节

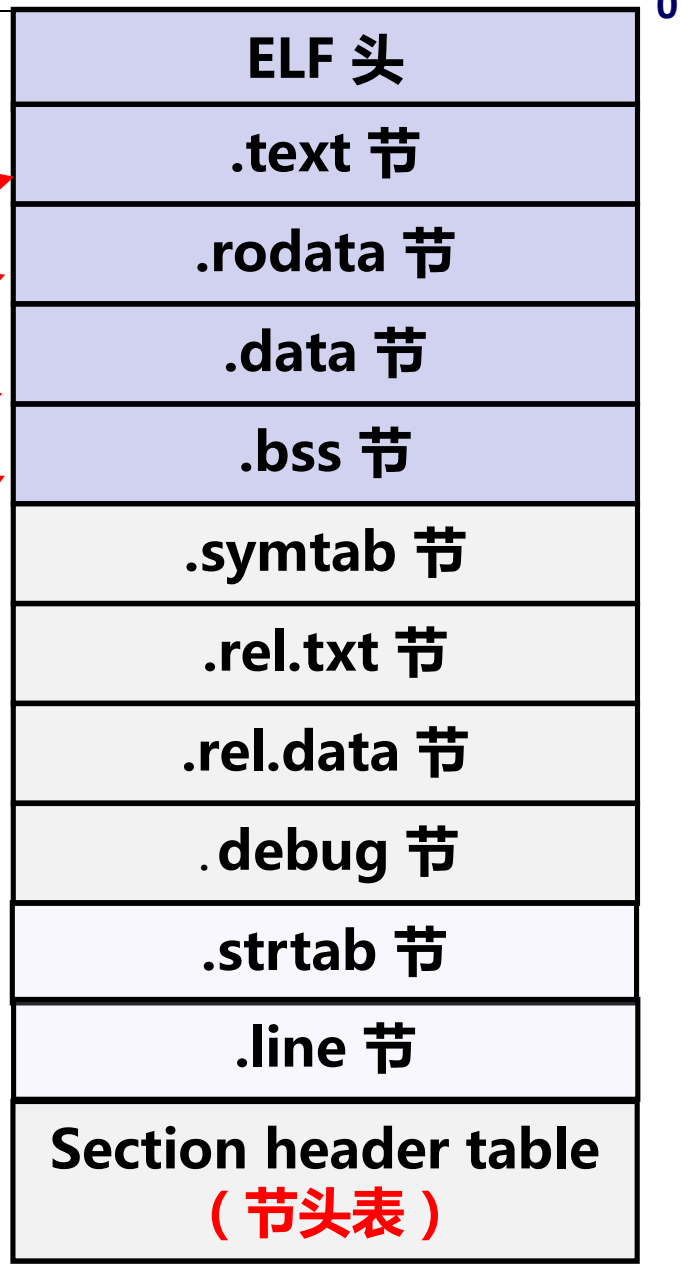
- ✓ 只读数据，如 printf 格式串、switch 跳转表等

.data 节

- ✓ 已初始化的全局变量

.bss 节

- ✓ 未初始化全局变量，仅是占位符，不占据任何实际磁盘空间。区分初始化和非初始化是为了空间效率



回顾：可重定位目标文件格式

.symtab 节
- ✓ 存放函数和全局变量（符号表）信息，它不包括局部变量

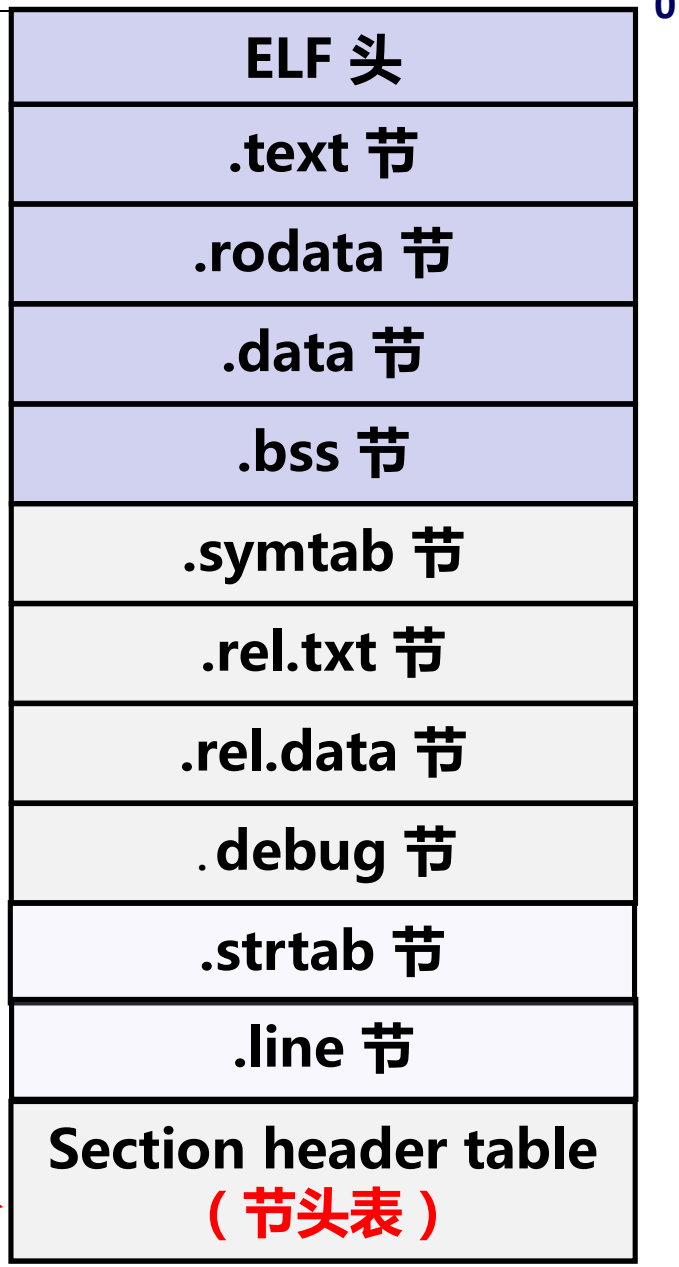
.rel.text 节
- ✓ .text节的重定位信息，用于重新修改代码段的指令中的地址信息

.rel.data 节
- ✓ .data节的重定位信息，用于对被模块使用或定义的全局变量进行重定位的信息

.debug 节
- ✓ 调试用符号表 (gcc -g)

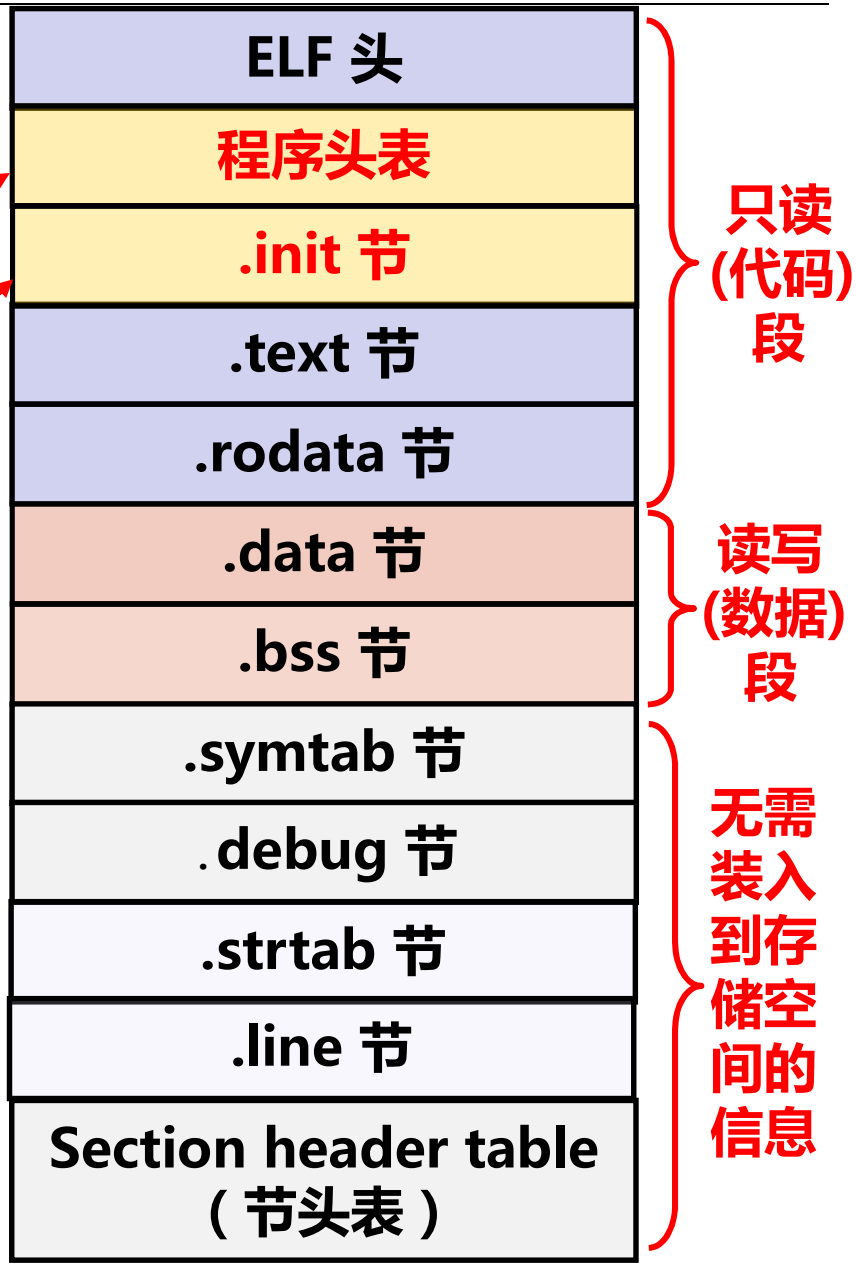
strtab 节
- ✓ 包含symtab和debug节中符号及节名

Section header table（节头表）
- ✓ 每个节的节名、偏移和大小



可执行目标文件格式

- 与可重定位文件稍有不同：
  - ELF头中字段e_entry给出执行程序时第一条指令的地址，而在可重定位文件中，此字段为0
  - 多一个程序头表，也称段头表（segment header table），是一个结构数组
  - 多一个.init节，用于定义_init函数，该函数用来进行可执行目标文件开始执行时的初始化工作
  - 少两个.rel节（无需重定位）



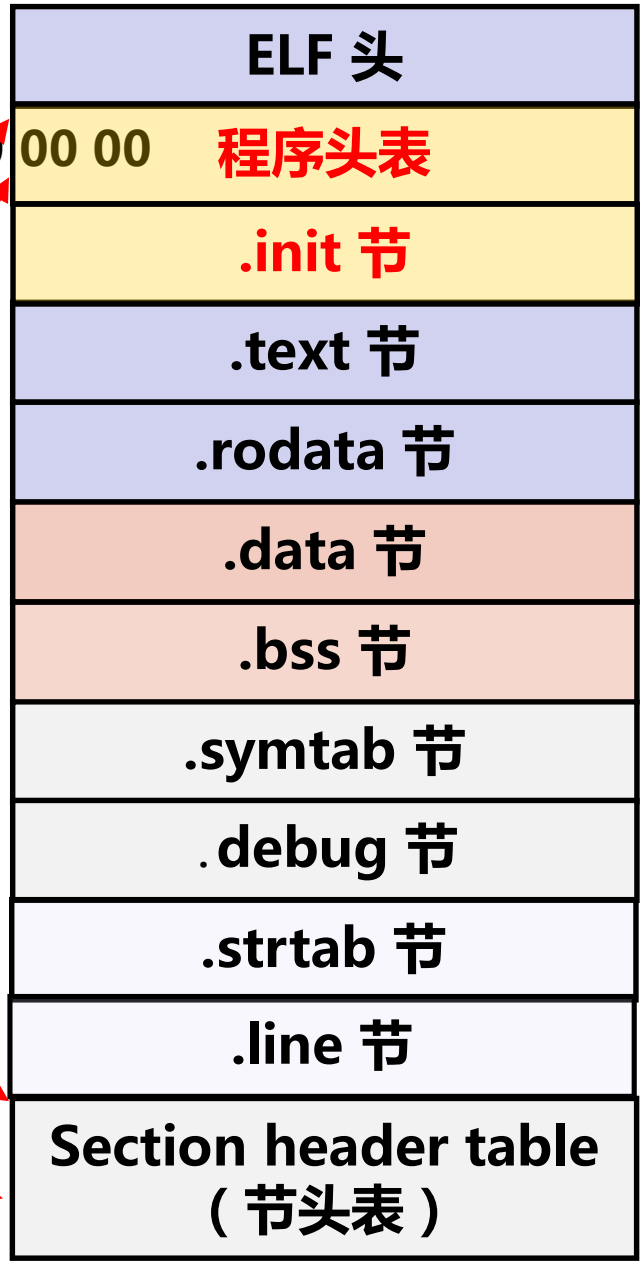
ELF头信息举例

$ readelf -h main 可执行目标文件的ELF头

```
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:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               x8048580
  Start of program headers:          52 (bytes into file)
  Start of section headers:          3232 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         8
  Size of section headers:           40 (bytes)
  Number of section headers:         29
  Section header string table index: 26
```

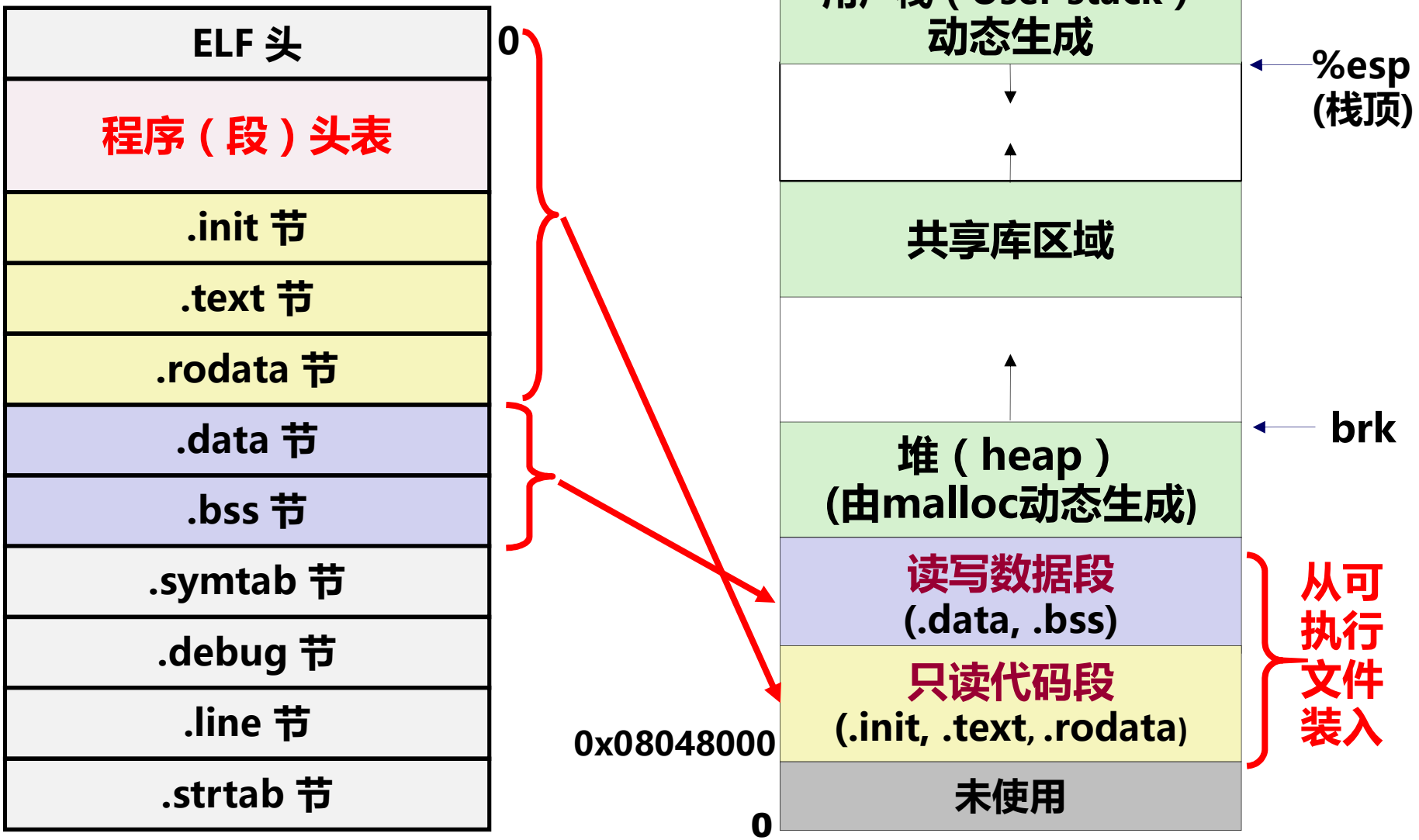
8x32B

29x40B



可执行文件的存储器映像

程序(段)头表描述如何映射



可执行文件中的程序头表

```
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;
```

程序头表描述可执行文件中的节与虚拟空间中的存储段之间的映射关系

一个表项（32B）说明虚拟地址空间中一个连续的段或一个特殊的节

以下是某可执行目标文件程序头表信息

有8个表项，其中两个为可装入段（即Type=LOAD）

$ readelf –l main

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004d4 0x004d4 R E 0x1000
  LOAD           0x000f0c 0x08049f0c 0x08049f0c 0x00108 0x00110 RW  0x1000
  DYNAMIC        0x000f20 0x08049f20 0x08049f20 0x000d0 0x000d0 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x000f0c 0x08049f0c 0x08049f0c 0x000f4 0x000f4 R   0x1
```

可执行文件中的程序头表

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004d4 0x004d4 R E 0x1000
  LOAD           0x000f0c 0x08049f0c 0x08049f0c 0x00108 0x00110 RW  0x1000
  DYNAMIC        0x000f20 0x08049f20 0x08049f20 0x000d0 0x000d0 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x000f0c 0x08049f0c 0x08049f0c 0x000f4 0x000f4 R   0x1
```

SKIP

第一可装入段：第0x00000~0x004d3字节（包括ELF头、程序头表、.init、.text和.rodata节），映射到虚拟地址0x8048000开始长度为0x4d4字节的区域，按0x1000=2^{12}=4KB对齐，具有只读/执行权限（Flg=RE），是只读代码段。

第二可装入段：第0x000f0c开始长度为0x108字节的.data节，映射到虚拟地址0x8049f0c开始长度为0x110字节的存储区域，在0x110=272B存储区中，前0x108=264B用.data节内容初始化，后面272-264=8B对应.bss节，初始化为0，按0x1000=4KB对齐，具有可读可写权限（Flg=RW），是可读写数据段。

可执行文件的存储器映像