可执行文件生成概述

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

一个典型程序的转换处理过程

经典的“ hello.c ”C-源程序

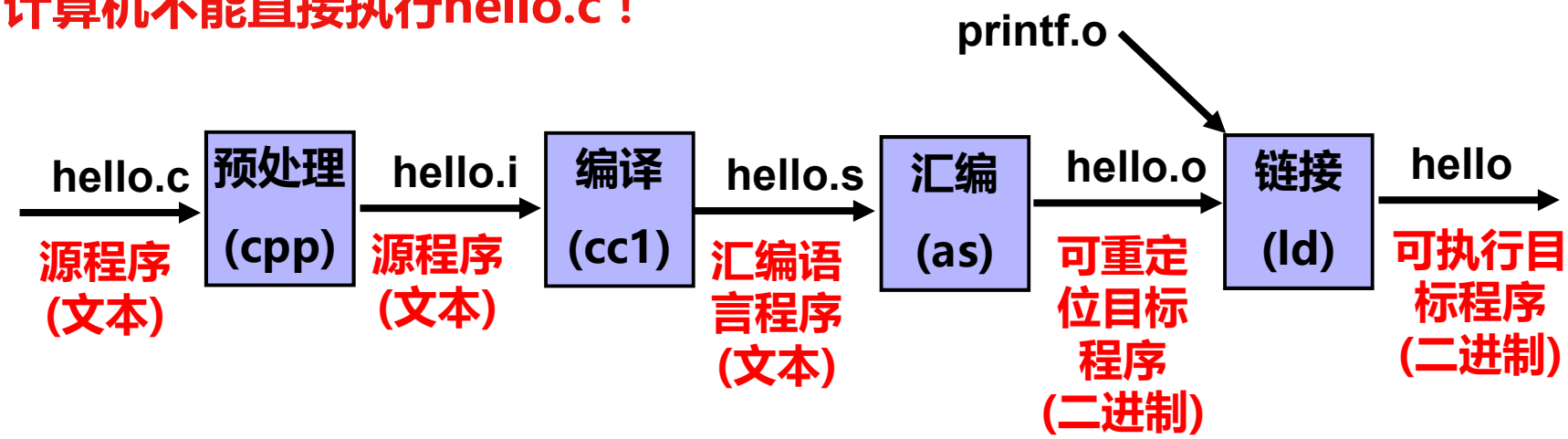
```
#include <stdio.h>

int main()
{
printf("hello, world\n");
}
```

hello.c的ASCII文本表示

```
# i n c l u d e <sp> < s t d i o .
35 105 110 99 108 117 100 101 32 60 115 116 100 105 111 46
h > \n \n i n t <sp> m a i n ( ) \n {
104 62 10 10 105 110 116 32 109 97 105 110 40 41 10 123
\n <sp> <sp> <sp> <sp> p r i n t f ( " h e l
10 32 32 32 32 112 114 105 110 116 102 40 34 104 101 108
l o , <sp> w o r l d \ n " ) ; \n }
108 111 44 32 119 111 114 108 100 92 110 34 41 59 10 125
```

功能：输出“hello,world”

计算机不能直接执行hello.c！



预处理

- 预处理命令
  - $gcc –E hello.c –o hello.i
  - $cpp hello.c > hello.i
- 处理源文件中以“#”开头的预编译指令，包括：
  - 删除“#define”并展开所定义的宏
  - 处理所有条件预编译指令，如“#if”，“#ifdef”，“#endif”等
  - 插入头文件到“#include”处，可以递归方式进行处理
  - 删除所有的注释“//”和“/* */”
  - 添加行号和文件名标识，以便编译时编译器产生调试用的行号信息
  - 保留所有#pragma编译指令（编译器需要用）
- 经过预编译处理后，得到的是预处理文件（如，hello.i），它还是一个可读的文本文件 ，但不包含任何宏定义

头文件（.h文件）的作用

c1.c

```
#include "global.h"

int f() {
  return g+1;
}
```

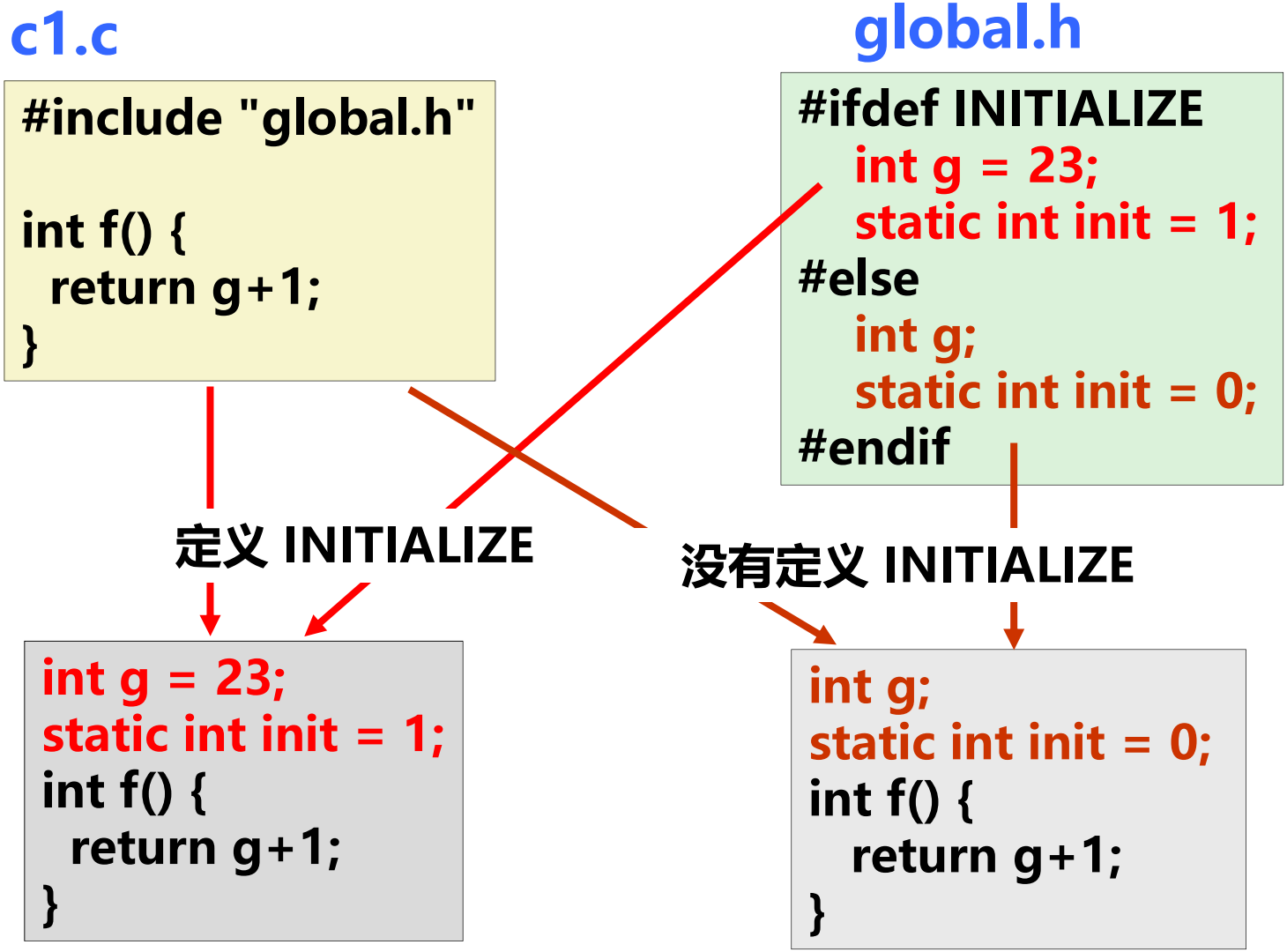
c2.c

```
#include <stdio.h>
#include "global.h"

int main() {
  if (!init)
      g = 37;
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

global.h

```
#ifdef INITIALIZE
    int g = 23;
    static int init = 1;
#else
    int g;
    static int init = 0;
#endif
```

预处理



#include指示被执行，插入.h文件的内容到源文件中

编译

- 编译过程就是将预处理后得到的预处理文件（如 hello.i）进行词法分析、语法分析、语义分析、优化后，生成汇编代码文件
- 用来进行编译处理的程序称为编译程序（编译器，Compiler）
- 编译命令
  - $gcc –S hello.i –o hello.s
  - $gcc –S hello.c –o hello.s
  - $/user/lib/gcc/i486-linux-gnu/4.1/cc1 hello.c
- 经过编译后，得到的汇编代码文件（如 hello.s）还是可读的文本文件，CPU无法理解和执行它

gcc命令实际上是具体程序（如ccp、cc1、as等）的包装命令，用户通过gcc命令来使用具体的预处理程序ccp、编译程序cc1和汇编程序as等

汇编

- 汇编代码文件（由汇编指令构成）称为汇编语言源程序
- 汇编程序（汇编器）用来将汇编语言源程序转换为机器指令序列（机器语言程序）
- 汇编指令和机器指令一一对应，前者是后者的符号表示，它们都属于机器级指令，所构成的程序称为机器级代码
- 汇编命令
  - $gcc –c hello.s –o hello.o
  - $gcc –c hello.c –o hello.o
  - $as hello.s -o hello.o　　（as是一个汇编程序）
- 汇编结果是一个可重定位目标文件（如，hello.o），其中包含的是不可读的二进制代码，必须用相应的工具软件来查看其内容

链接

- 预处理、编译和汇编三个阶段针对一个模块（一个*.c文件）进行处理，得到对应的一个可重定位目标文件（一个*.o文件）
- 链接过程将多个可重定位目标文件合并以生成可执行目标文件
- 链接命令
  - $gcc –static –o myproc main.o test.o
  - $ld –static –o myproc main.o test.o

–static 表示静态链接，如果不指定-o选项，则可执行文件名为“a.out”

从本周开始主要介绍如何进行程序模块的链接

链接器的由来

- 原始的链接概念早在高级编程语言出现之前就已存在
- 最早程序员用机器语言编写程序，并记录在纸带或卡片上

穿孔表示0，未穿孔为1

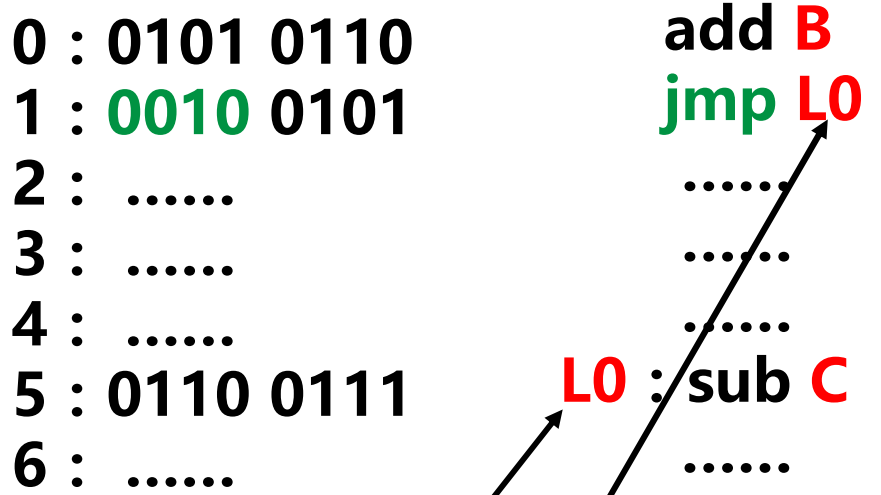
假设：0010-jmp

0：0101 0110
1：0010 0101
2：……
3：……
4：……
5：0110 0111
6：……

若在第5条指令前加入指令，则程序员需重新计算jmp指令的目标地址（重定位），然后重新打孔。

链接器的由来

- 用符号表示跳转位置和变量位置，是否简化了问题？
- 汇编语言出现
  - 用助记符表示操作码
  - 用符号表示位置
  - 用助记符表示寄存器
  - .....
- 更高级编程语言出现
  - 程序越来越复杂，需多人开发不同的程序模块
  - 子程序（函数）起始地址和变量起始地址是符号定义（definition）
  - 调用子程序（函数或过程）和使用变量即是符号的引用（reference）
  - 一个模块定义的符号可以被另一个模块引用
  - 最终须链接（即合并），合并时须在符号引用处填入定义处的地址



如上例，先确定L0的地址，再在jmp指令中填入L0的地址

使用链接的好处

链接带来的好处1：模块化

（1）一个程序可以分成很多源程序文件

（2）可构建公共函数库，如数学库，标准C库等

（代码重用，开发效率高）

链接带来的好处2：效率高

（1）时间上，可分开编译

只需重新编译被修改的源程序文件，然后重新链接

（2）空间上，无需包含共享库所有代码

源文件中无需包含共享库函数的源码，只要直接调用即可

（如，只要直接调用printf()函数，无需包含其源码）

可执行文件和运行时的内存中只需包含所调用函数的代码

而不需要包含整个共享库

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

可执行文件的生成

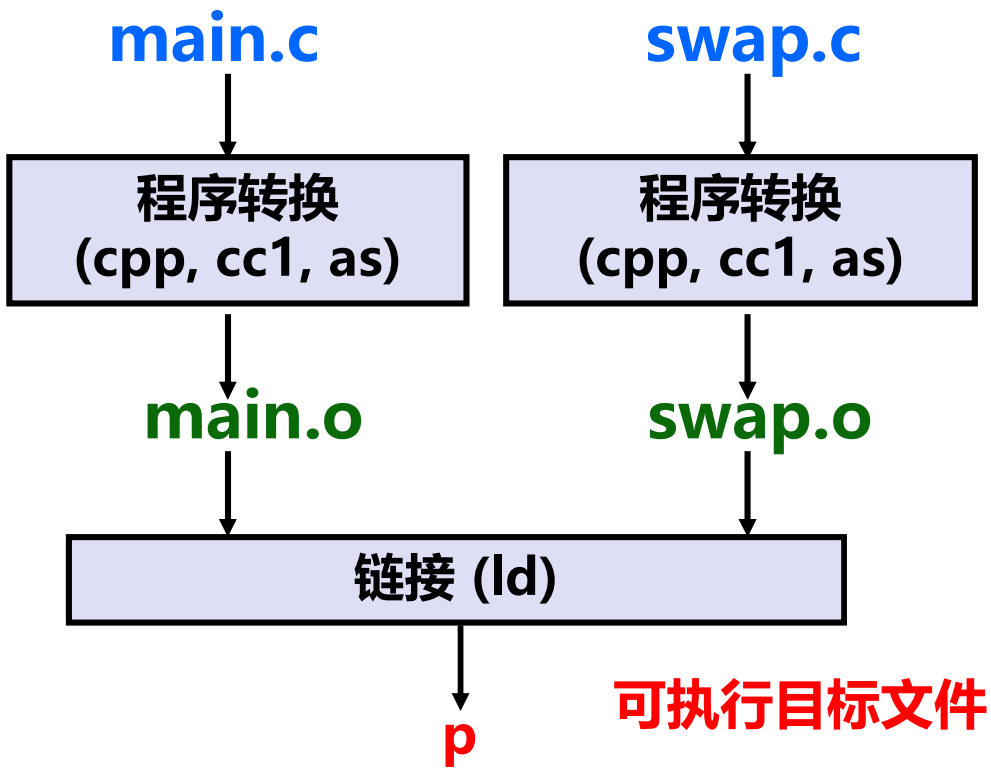
- 使用GCC编译器编译并链接生成可执行程序P:
  - $ gcc -O2 -g -o p main.c swap.c
  - $ ./p

-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



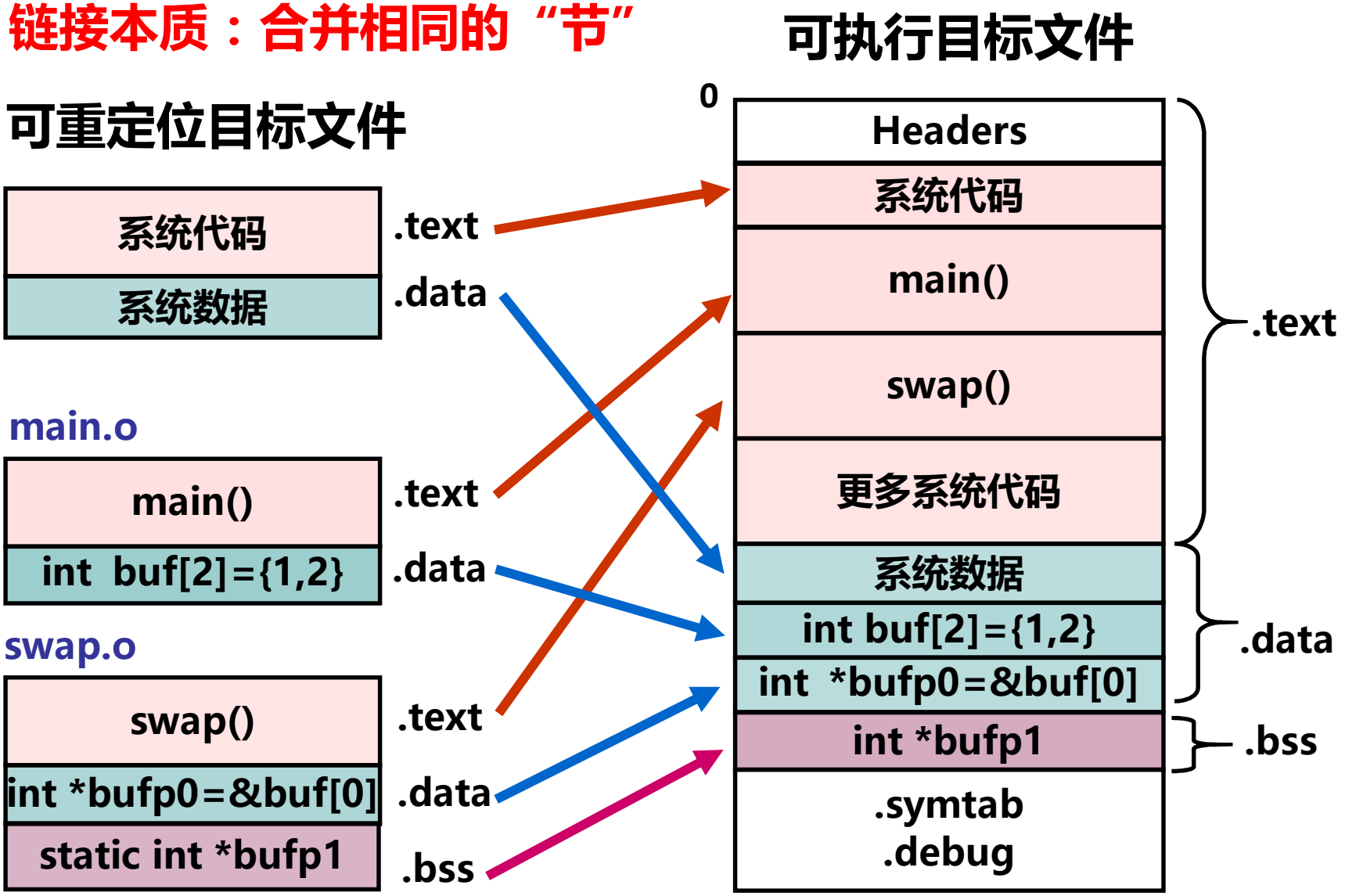
源程序文件

分别转换（预处理、编译、汇编）为可重定位目标文件

可执行目标文件

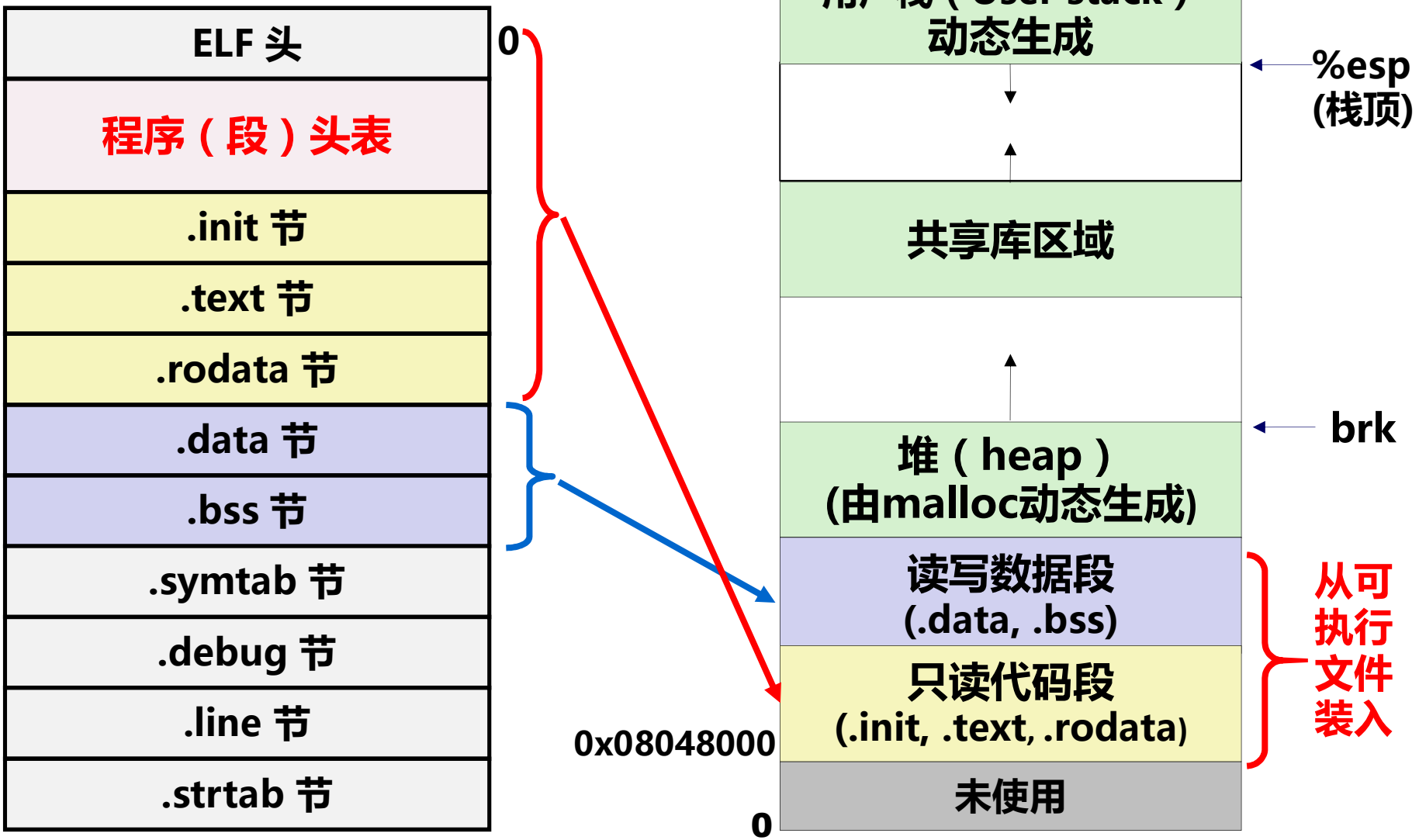
链接过程的本质

链接本质：合并相同的“节”



可执行文件的存储器映像

程序(段)头表描述如何映射



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
```