链接

编译过程

编译过程如下，这里以c文件来解释：

预处理->编译->汇编->链接。今天我们的主角是链接，不过前面的过程大致提一下下而已。

预处理：

它的作用是把以"#"开头的如#include，#define，#ifdef，#if等，对应的命令如下：

gcc -E a.c -o a.i

-o：这个参数的意思是把转化的文件重命名，如果不指定的话会有默认的文件名。

编译：

它的作用是把预处理后的文件翻译为汇编文件，对应的命令如下：

gcc -S a.i -o a.S

汇编：

它的作用是把汇编后的文件翻译为机器能懂的二进制文件，对应的命令如下：

gcc -c a.S -o a.o

a.o文件尼也是我们今天要介绍的重点，它是一个elf格式的文件，可以用命令：

readelf -a a.o可以查看该文件的格式，这里不讲多了，下面会详细介绍。

链接：

汇编后的文件其实就是各个段的集合，比如数据段，代码段等。那么链接的作用就是

把各个.o的文件的各个相同的段集合起来，构成一个可以执行的二进制文件，它的命令

比较多，后面也会一一的介绍。

后面的编译器就不用gcc了，直接用arm-linux-gcc，arm-linux-ld, arm-linux-objdump来说明。

静态链接

说了今天的主角是链接，而这里主要说明的是静态链接，所以在链接前我们假设已经把各个模块

的.c文件翻译为.o文件了，现在要做的是链接的过程。链接过程主要完成两个任务：

符号解析：这里我们把.o文件称为**目标文件**，目标文件里定义的**变量名**或者**函数名**等称为**符号**。

那么一个目标文件里有自己定义的符号和引用其他文件定义的符号（变量或者函数）。

符号解析的作用就是把这些引用的符号（自己定义的和引用其他文件的）与符号关联起

来不明白了吧(小瓜皮)，什么意思？为了能理解elf文件格式我们先来个开胃菜，意思就是有一**个**内存地址存放的是一个定义好了的符号的内容，我们把这个地址称为**符号定义地址**，还有**种**地址是定义的符号被四处调用，每调用一次都会用一个内存地址把这个符号存储起来，我们把这样的地址称为**符号引用地址**。那么符号解析的作用就是把这些四处的符号引用地址都指向符号定义地址(也就是关联的意思)。就这么一段话搞得我到处找资料解释"关联"的意思！！！请善待我的成果。

重定位：编译器和汇编器一般在没有链接时会生成从地址0开始的代码和数据段，这里我就不说段了，改为节（数据节，代码节）为什么要说成节，是因为elf格式都是节。连接器就把每个模块的节合并起来构成新的节，并为每个节分配新的内存地址，同样会为每一个符号重新分配内存这个过程称为**重定位节和符号定义**；还有个过程就是上面说的把每个引用符号与符号定义关联起来称为**重定位节中的符号引用**。

下面我们来继续深入研究一下符号解析和重定位的过程：

符号解析

符号解析之目标文件

上面说了把.o文件称为目标文件，其实这样的说法不准确。目标文件由如下三种：

可重定位的目标文件：也就是各个.c文件（或者说模块）被编译为的.o文件。

可执行目标文件：就是把各个模块链接起来形成的文件，也就是可执行文件。

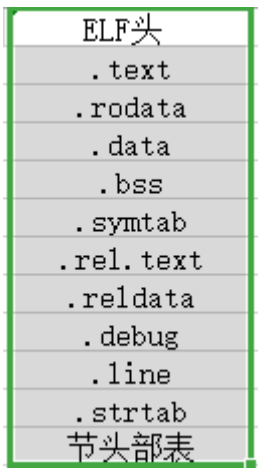
共享目标文件：一种特殊的可重定位的目标文件，可以在加载或者运行时被动态地加载进内存并链接。

比如库。

符号解析之可重定位目标文件

好了重点来了，小老弟们。我们这里的目标文件重点说的是.o文件，就不是共享目标文件和可执行的目标文件。

上面说到.o文件是一种elf文件的格式，可执行的目标文件格式与它类似，但是这里的重点不是可执行的目标文件格式，如下图：



我们把ELF头和节头部表放在后面说，先说中间的节。

.text ：已编译程序的机器代码，也就是所谓的的代码段

.rodata：只读数据，比如printf中的语句格式等

.data：好！！要注意了，这里存放的是**已初始化的全局和静态c变量**，而局部变量则是存在栈里面

不在.data，.bss节里面。

.bss：**未初始化的全局和静态c变量，以及所有被初始化为0的全局或者静态变量。**但是它并不占内存空间，而是在

运行的时候分配内存，并初始化为0。

.symtab：符号表，描述上面提到的（函数和变量名符号）信息，下面会重点介绍

.rel.text：一个.text节中位置的列表，上面说了链接的重定位过程有一个重定位引用符号，那么在链接的时候就是

修改这里的信息，其实就是修改引用地址相对于.text的偏移量而已，后面具体说明。

.rel.data：上面的表中少了个点（"."）和上面的.rel.text一样，只是这里针对的是变量的重定位，而上面是针对代 码的重定位。

.debug：一个调试符号表，其条目是程序中定义的局部变量和类型定义，程序中定义和引用的全局变量，以及

原始的C源文件(这个不是重点)

.line：原始C源程序中的行号和.text节中机器指令之间的映射

.strtab：一个字符串表，其内容包括.symtab和.debug节中的符号表，以及节头部中的节名字。比如.symtab的

字符偏移，就是从.strtab来的

ELF头：其实ELF头，节头部表，.symtab等都是结构体的形式来表述的，比如这里的ELF头开头16字节描述系统的字的大小和字节顺序，剩下的描述了ELF头的大小，目标文件的类型，机器类型(如x86-64)，节头部表的文件的偏移，节头部表条目的数量和大小，而不同节的位置和大小由节头部表描述，具体结构体如下：

```
#define EI_NIDENT 16
typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    ElfN_Addr    e_entry;
    ElfN_Off      e_phoff;
    ElfN_Off      e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;
    uint16_t      e_phnum;
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
    uint16_t      e_shstrndx;
} ElfN_Ehdr;
```

e_ident[EI_NIDENT]：开始的16字节

e_type：目标文件类型

e_machine：机器类型

e_version：ELF版本号

e_entry：入口地址，规定ELF程序的入口虚拟地址，操作系统在加载完该程序后从这个地址开始执行进程的指 令。可重定位目标文件一般没有入口地址，则这个值为0，也就是只有加载后的可执行的目标文件才会有

e_phoff：程序头表在文件中的偏移，可重定位目标文件不存在程序头表（可执行的目标文件有），故该值为0。

e_shoff：节头表在文件中的偏移

e_flags：ELF标志位，用来标识一些ELF文件平台相关的属性，相关常量的格式一般为EF_machine_flag， machine为平台，flag为标志

e_ehsize：ELF文件头本身的大小

e_phentsize：程序头表表项的大小，可重定位目标文件不存在程序头表（可执行的目标文件有）故该值为0

e_phnum：程序头表表项的数目，可重定位目标文件不存在程序头表（可执行的目标文件有）故该值为0

e_shentsize：节头部表表项的大小

e_shnum：节头部表表项的数目

e_shstrndx：字符串表在节头部表中的下标（上面说了不同节的位置和大小由节头部表描述）

来截个图具体的来看一下，输入命令比如a.o文件：readelf -a a.o：

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          216 (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:         11
  Section header string table index: 8
```

总结就是ELF头描述基本信息，重点记住它描述了节头部表的信息，而节头部表也有相关的格式描述其他的节，很重要。既然ELF头描述的是节头部表，而节头部表描述的是ELF头与节头部表之间的节的信息，那么我们按照顺序

再来说。

节头部表：它描述的是ELF头与节头部表之间节的信息，也没什么重要说的，输入readelf -a a.o如下图：

```
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 00001d 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000344 000010 08      9   1  4
  [ 3] .data             PROGBITS        00000000 000054 000000 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 000054 000000 00  WA  0   0  4
  [ 5] .rodata           PROGBITS        00000000 000054 00000c 00   A  0   0  1
  [ 6] .comment          PROGBITS        00000000 000060 000026 01  MS  0   0  1
  [ 7] .note.GNU-stack   PROGBITS        00000000 000086 000000 00      0   0  1
  [ 8] .shstrtab         STRTAB          00000000 000086 000051 00      0   0  1
  [ 9] .symtab           SYMTAB          00000000 000290 0000a0 10     10   8  4
  [10] .strtab           STRTAB          00000000 000330 000014 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)
```

哦，想起来，这个要重点注意地址项和偏移项，这里会和后面要说到的命令参数比如arm-linux-ld -Ttext

有关系以及.lds脚本里面的SECTIONS的定义相关。这里简单说一下，比如有一个a.o文件我可以用命令

arm-linux-ld -Ttext 0x30000000 -o a.elf来修改.text的地址项，而off（偏移）表示的是在一个可执行的文件中这些节应该在放在可执行的文件中的哪个地方。Size表示这个节的大小。

中间的节中我们重点说一下.symtab。

.symtab:它描述的是符号的信息，符号就是上面说的函数和变量名称。具体符号有哪些看如下描述：

1）：由模块m定义并能被其他模块引用的全局符号，全局链接器符号对应于非静态的C函数和全局变量

2）：由其他模块定义并被模块m引用的全局符号。这些符号称为外部符号，对应于在其他模块中定义的

非静态c函数和全局变量

3）：只被模块m定义和引用的局部符号，它们对应于带static属性的c函数和全局变量，这些符号在模块 m中任何位置都可见，但是不能被其他模块引用

上面说了.symtab节也有相关的结构，如下：

typedef struct

{

int name;

char type

char binding;

char reserved;

short section;

long value;

long size;

}Elf64_Symbol;

name：指的是相对于.strtab 节的偏移量，因为.strtab是一个字符表存储字符的，而.symtab是相关字符的信息 的，所以索引相关的字符应该从.strtab查询，也就是要查询的字符相对于.strtab的偏移量是多少，而这个

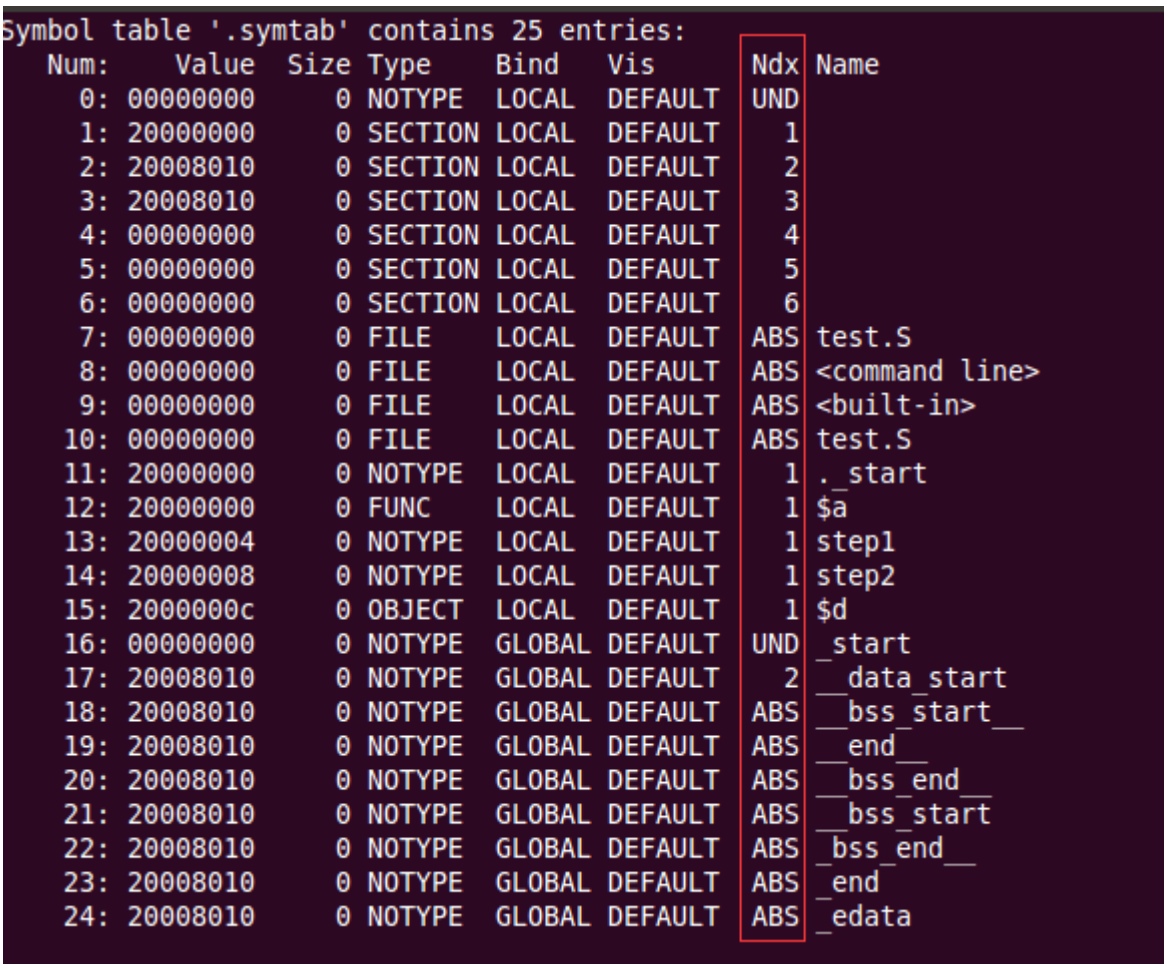
偏移量的值就是这里的name的值。

type：该字符是函数还是变量

binding：是本地的还是全局的

reserved：保留

section：每个符号都被分配到目标文件的某一个节，由该节来指定，它也是节头部表的索引，在下图中由Ndx来 指示：



```
Symbol table '.symtab' contains 25 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 20000000     0 SECTION LOCAL  DEFAULT    1
     2: 20008010     0 SECTION LOCAL  DEFAULT    2
     3: 20008010     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    6
     7: 00000000     0 FILE    LOCAL  DEFAULT  ABS test.S
     8: 00000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
     9: 00000000     0 FILE    LOCAL  DEFAULT  ABS <built-in>
    10: 00000000     0 FILE    LOCAL  DEFAULT  ABS test.S
    11: 20000000     0 NOTYPE  LOCAL  DEFAULT    1 _start
    12: 20000000     0 FUNC    LOCAL  DEFAULT    1 $a
    13: 20000004     0 NOTYPE  LOCAL  DEFAULT    1 step1
    14: 20000008     0 NOTYPE  LOCAL  DEFAULT    1 step2
    15: 2000000c     0 OBJECT  LOCAL  DEFAULT    1 $d
    16: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND _start
    17: 20008010     0 NOTYPE  GLOBAL DEFAULT    2 __data_start
    18: 20008010     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start__
    19: 20008010     0 NOTYPE  GLOBAL DEFAULT  ABS __end__
    20: 20008010     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_end__
    21: 20008010     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start
    22: 20008010     0 NOTYPE  GLOBAL DEFAULT  ABS _bss_end__
    23: 20008010     0 NOTYPE  GLOBAL DEFAULT  ABS _end
    24: 20008010     0 NOTYPE  GLOBAL DEFAULT  ABS _edata
```

1: 表示.text节，3表示.data节等，为什么1是代码节，3是数据节，上面就说了，它也是节头部表的索引，我们来看节头部表的图(Section Headers)的[Nr]列与这里的Ndx是一一对应的。图中还有ABS,UND等字符，有这么一个小知识点，有三个伪节，它们在节头部表中没有条目（也就是说，在节头部表中没有相关ABS,UND等字符的索引值），哪三个伪节尼：ABS代表不该被重定义的符号；UNDEF(也就是上图的UND)代表未定义的符号，也就是在本目标模块中引用，但是却在其他地方定义的符号；COMMON表示还未被分配位置的未初始化的数据目标，对

COMMON符号，value字段给出对齐要求，而size给出最小的大小。这里要注意的是只有可重定位目标文件中才有伪节，可执行目标文件中是没有的。上面的.bss节存储未初始化的全局和静态c变量，以及所有被初始化为0的全局或者静态变量，那么.bss与COMMON有什么区别？

COMMON和.bss的区别。现代的GCC版本根据以下规则来将可重定位目标文件中的符号分配到COMMON和

.bss中。

COMMON ： 未初始化的全局变量

.bss：未初始化的静态变量，以及初始化为0的全局或静态变量

继续上面的结构体说明

volue：表示各个符号在各个节中的偏移量

size：表示符号的大小（函数占的空间大小，变量占的空间大小）

好了关于可重定位的elf格式也算是全部说完了，下面我们来大致总结一下各个节的描述

符号解析之elf文件格式总结

ELF头描述了基本的信息（大可不必关心），但是它描述了节头部表的大小，条目个数等。这个没多大的信息量

节头部表描述了节头部表与ELF头之间节的信息，这些节所处的地址，大小，以及这些节在可执行目标文件中所在的偏移，也就是这些节相对于可执行目标文件中所在的偏移被确定，然后具体的运行时地址也就被分配了。

然后是节头部表与ELF头之间节的.strtab节存储的是符号名称，其他节想要显示某一个符号的名字是什么，的时候，都要从.strtab节中查询，查询的结果是相对于.strtab节的偏移,比如.symtab节想索引某一个字符，.symtab里的name就是一个int型的索引值，该值就是符号相对于.strtab节的偏移量。然后在介绍了.symtab节描述各个符号是什么类型的符号，是函数还是变量，这些符号应该是放在哪个节里面是.text还是.data，是本地的还是全局的这些信息。至于.rel.text .rel.data我们后面会详细的介绍，因为链接有两步骤：符号解析和重定位符号。把他两的解释放在重定位符号的解释中去说明。

符号解析的过程

上面提到过，符号解析就是把四处引用符号怎么和符号定义相互关联起来，那么这样做了之后，当要去执行取某一个变量的内容或者跳转到函数的定义的地址时，就回去变量或函数**定义**（不是引用）的位置去取相关的值。看到这里希望能明白符号引用与符号定义的区别，上面有说到。这里不概述是用什么方法关联的，只说关联的过程中遇到的问题。对于静态的局部变量的话，关联很简单，这个静态的变量不能被其他的模块引用，地址分配时就是唯一的，只要定义该变量的模块调用就从这个地址获取，其他模块调用就不行，好，那么问题来了，若是多个模块定义相同名称的全局的变量该怎么办？比如a.c文件定义一个全局变量int i = 0，在b.c文件中同样定义int i = 0，那在把各个模块链接起来的时候请问b文件在某一处在调用i的值的时候，是采用a文件里的变量的值，还是采用b自己定义的值？？下面我们做一个小节来说说连接器是如何解析多重定义的全局的符号的。

连接器解析多重定义的全局的符号

Linux编译系统采用如下的方法：

在编译时，汇编器输出每个全局符号，或者强或者弱，函数和已初始化的全局变量是强符号，未初始化的全局变量是弱符号，根据强弱的定义，Linux链接器使用下面的规则来处理多重定义的符号名：

规则1：不允许有多个同名的强符号。

规则2：如果有一个强符号和多个弱符号，那么选择强符号。

规则3：有多个弱符号，就任选一个。

这里做个试验我们有两个文件a.c b.c都定义了全局变量i如下图：

```
root@ubuntu:~/TEST2# cat a.c b.c
#include<stdio.h>
int i = 0;
int main()
{
        printf("hello\n");
        return 0;
}
#include<stdio.h>
int i = 1;
int main()
{
        printf("world\n");
        return 0;
}
root@ubuntu:~/TEST2#
```

在我们用arm-linux-gcc -c a.c b.c -o a.o b.o构成两个可重定位文件a.o和b.o文件。然后我们把它量链接在一起试试，如下图：

```
root@ubuntu:~/TEST2# ls
a.c  a.o  b.c  b.o
root@ubuntu:~/TEST2# arm-linux-ld a.o b.o -o ab
b.o(.data+0x0): multiple definition of `i'
a.o(.bss+0x0): first defined here
b.o(.text+0x0): In function `main':
: multiple definition of `main'
a.o(.text+0x0): first defined here
arm-linux-ld: warning: cannot find entry symbol _start; defaulting to 00008074
a.o(.text+0x10): In function `main':
: undefined reference to `printf'
b.o(.text+0x10): In function `main':
: undefined reference to `printf'
root@ubuntu:~/TEST2#
```

那么上图就会出现多重定义的错误i和main，当然里面还有个错误是printf，因为编译器用的是arm，不是gcc所以没有相应的库来定义printf函数。

所以我们在写代码时涉及到多个模块时，确定定义的全局变量不会再其他模块被引用的情况下，一定要加一个static关键字，或者根据模块来定义变量的名称，避免static定义过多导致内存利用率低。

上面是规则1的引用，我们来看规则2的应用存在的代码风险问题，试验就不做了，用arm没打印库，用gcc没启动文件，当然主要是我还有很多知识点不懂，目前做不了，不过可以大致说一下：

a.c文件里面代码如下: b.c文件里面的代码如下：

#include<stdio.h> int x ;

void f(); void f()

int x = 15213; {

void main() x = 15212;

{ }

f();

printf("x= %d\n",x);

}

在运行时，根据规则2，取强，那么x**定义**取的值为15213，结果在运行main函数的时候调用f函数导致x的值被改变，若是f函数误操作写了个x = 15212；那么本来main里面不想改变x的值却因为f函数改变了，那么这就产生耦合关系，但是程序还能跑，导致这个bug很难去发现，所以**写代码对于全局变量一定要赋初值**。不然得到意想不到的结果。

好了上面就是解析全局符号的过程，那么下面我们来说说静态库来解析引用。

连接器如何使用静态库来解析引用

首先说明静态库是怎么来的，现在假设我们并不知道库的定义，我们只知道.o文件（可重定位），只能用.o文件来实现连接的过程，那么若是模块多的话，实现连接起来费时，解决费时的问题是把各个模块集合为一个大的.o文件，那么确实可以解决时间问题，但是却引来了空间大小的问题。可能占用空间非常大，为了解决时间和空间的问题，引来了库的概念，怎么一个过程？就是把多个模块集合成一个库，若是某一个模块比如a.o要与b.a这个库进行链接那么，在a.o中调用b.a的某一个函数，会把b.a中对应的模块与a.o进行合并，而不是把全部合并起来。怎么实现的？来，我们看一下原理：

在符号解析阶段，链接器从左到右按照它们在编译器驱动程序命令上出现的顺序来扫描可重定位目标文件(.o文件)和存档文件(.a文件)，在这次扫描中，链接器维护一个可重定位目标文件的集合E(这个集合中的文件会被合并起来形成可执行文件)，一个未解析的符号(即引用了但是尚未定义的符号)集合U，以及一个在前面输入文件中已定义的符号集合D。初始时，E，U和D均为空。

对于命令行上的每个输入文件f，链接器会判断f是一个目标文件还是一个存档文件。如果f是一个目标文件，那么链接器把f添加到E，修改U和D来反映f中的符号定义和引用，并继续下一个输入文件。

如果f是一个存档文件，那么链接器就尝试匹配U中未解析的符号和由存档文件成员定义的符号，若是存档文件成员m，定义一个符号来解析U中的一个引用，那么就将m加到E中，并且链接器修改U和D来反映m中的符号定义和引用，若是该库没有，那么继续找下一个库，直到链接器处理下一个输入文件

如果当链接器完成对命令行上输入文件的扫描后，U是非空的，那么链接器就会输出一个错误并终止。否则，它会合并和重定位E中的目标文件，构成输出的可执行文件。

那么上面的算法会有一个问题，那就是再写入命令比如arm-linux-ld a.o b.a -o ab的时候**库文件必在可重定位目标文件的后面**，若是库文件在前面集合U就是空的，但是当扫描到.o文件时，有个符号要从某一个库文件中获取，但是库文件在.o文件的前面，链接器就不会再扫描库文件了(比如有个原因U是空的继续下一个输入文件)，那么U就是

到最后是非空的，那么就会出错并终止链接。

文章写到这也就快结束了，链接分两个步骤：符号解析和重定位，上面就是解析符号的大致过程了，下面我们来谈谈重定位的过程。

重定位

重定位由两步组成：

1：重定位节和符号定义

重定位节的意思是就是把各个合并后的节重新分配内存，同样重定位**符号定义**（注意这里是符号定义不是引用）就是给符号定义重新分配地址。

2：重定位节中的符号引用

这一步中，链接器修改代码节和数据节中每个符号的引用，使得他们指向正确的运行地址，要执行这一步，链接器依赖于可重定位目标模块中称为重定位条目的数据结构。

上面的重定位节和符号定义的内存如何分配这里就不详细说明，我们来说说重定位条目的结构。

重定位条目

当汇编器生成一个目标模块时，它并不知道数据和代码最终将放在内存中的什么位置。它也不知道这个模块引用的任何外部定义的函数或者全局变量的位置。所以，无论何时编译器遇到对最终位置未知的目标引用，它就会生成一个重定位条目，告诉链接器在将目标文件合成可执行文件时如何修改这个引用。代码的重定位条目放在.rel.text中已初始化数据的重定位条目放在.rel.data中。重定位条目格式如下：

typedef struct

{

long offset;

long type;

long symbol;

long addend;

}Elf64_Rela;

offset：指**符号引用**地址相对于某一节的偏移量，比如.text中有个引用符号i，i相对于.text的偏移量。

symbol：可以理解为符号名称，但是它是一个索引值，上面就有提到是相对于字符表的索引。

addend：一个有符号常数，一些类型的重定位要使用它对被修改引用的值做偏移调整。

type：告诉链接器如何修改新的引用。

对于type有两种最基本的类型

1)：R_X86_64_PC32。重定位一个使用32位PC相对地址的引用，一个PC相对地址就是距程序计数器(PC)的当前运 行时值的偏移量，当CPU执行一条使用PC相对寻址的指令时，它就将在指令中编码的32位值加上PC的当前运 行时值，PC值通常是下一条指令在内存中的地址。后面的实际应用中我们会看到这是如何计算的。

2)：R_X86_64_32。重定位一个使用32位绝对地址的引用。通过绝对寻址，CPU直接使用在指令中编码的32位值 作为有效地址，不需要进一步修改。

重定位符号引用

算法是这样：假设Elf64_Rela结构体的值如下：

r.offset = 0xf;

r.symbol = sum;

r.type = R_X86_64_PC32

r.addend = -4

假设我们知道ADDR(.text) = 0x4004d0

符号定义的地址为ADDR(r.symbol) = 0x4004e8

根据r.offset知道符号被调用的地址是refaddr = ADDR(.text) + offset = 0x4004df

那么符号引用到符号定义的偏移refptr为（refaddr - ADDR(r.symbol) + r.addend) = 0x5

那么PC = PC + refptr，这里PC的值是调用某一个符号的地址值。为什么PC要加这个偏移，是因为PC指针指向了符号引用的位置要调到符号定义的位置，那么PC指针的偏移量不正好是符号引用到定义的偏移量吗。

实际的例子

我们有一段简单的汇编代码如下：

```
root@ubuntu:~# cat test.S
.text
.global _start
._start:
        b step1
step1:
        ldr pc, =step2
step2:
        b step2
root@ubuntu:~#
```

我们输入arm-linux-gcc -c test.S -o test.o命令，把它翻译为二进制文件后，在输入如下的命令：

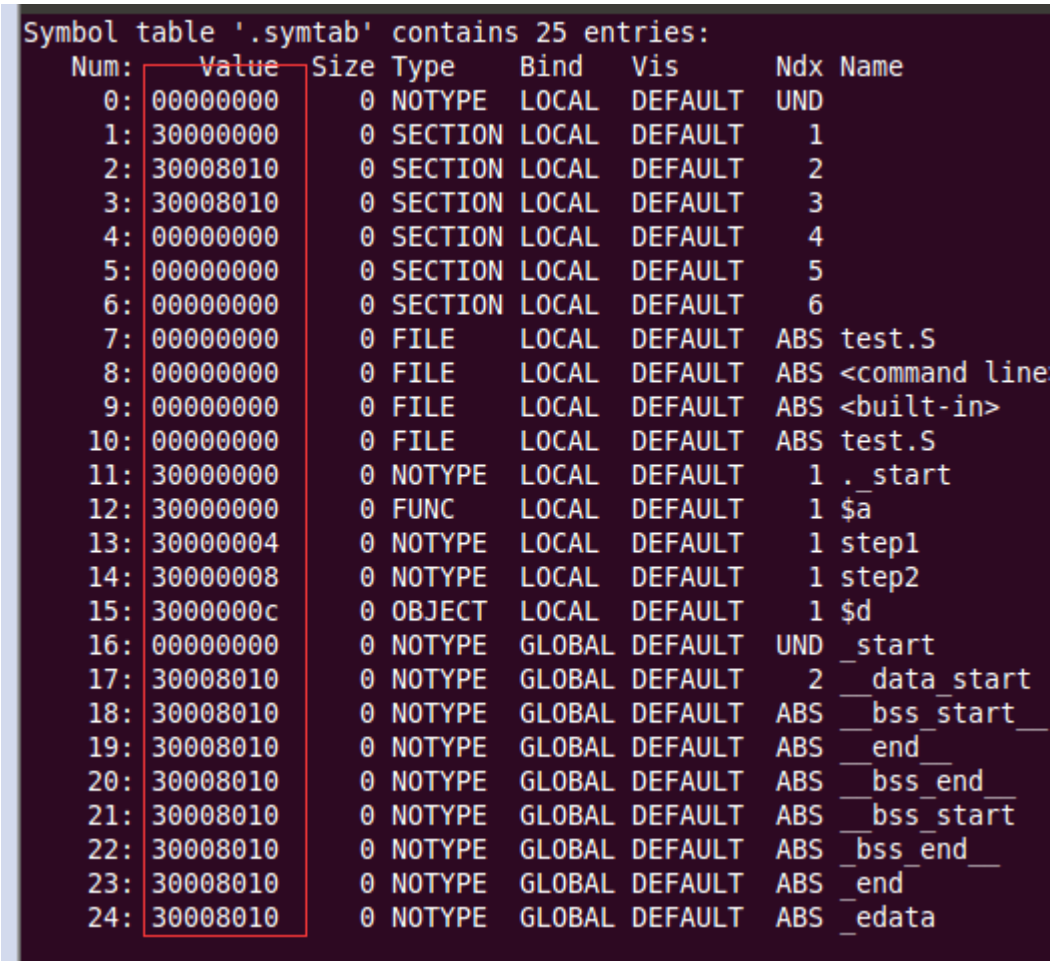
arm-linux-ld -Ttext 0x30000000 test.o -o test.elf，然后用readelf -a test.elf来查看文件如下图：

```
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        30000000 008000 000010 00  AX  0   0  4
  [ 2] .data             PROGBITS        30008010 008010 000000 00  WA  0   0  1
  [ 3] .bss              NOBITS          30008010 008010 000000 00  WA  0   0  1
  [ 4] .shstrtab         STRTAB          00000000 008010 00002c 00      0   0  1
  [ 5] .symtab           SYMTAB          00000000 008154 000190 10      6  16  4
  [ 6] .strtab           STRTAB          00000000 0082e4 000095 00      0   0  1
```

我们可以看到.text的地址就变为30000000，我们还可以查看下图的ELF头：

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            ARM
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x30000000
  Start of program headers:          52 (bytes into file)
  Start of section headers:          32828 (bytes into file)
  Flags:                             0x202, has entry point, GNU EABI, software FP
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         7
  Section header string table index: 4
```

这一行说明了函数的入口地址就是0x30000000也就是-Ttext指定的位置。但是，我会看到如下图：



```
Symbol table '.symtab' contains 25 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 30000000     0 SECTION LOCAL  DEFAULT    1
     2: 30008010     0 SECTION LOCAL  DEFAULT    2
     3: 30008010     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    6
     7: 00000000     0 FILE    LOCAL  DEFAULT  ABS test.S
     8: 00000000     0 FILE    LOCAL  DEFAULT  ABS <command line
     9: 00000000     0 FILE    LOCAL  DEFAULT  ABS <built-in>
    10: 00000000     0 FILE    LOCAL  DEFAULT  ABS test.S
    11: 30000000     0 NOTYPE  LOCAL  DEFAULT    1 .start
    12: 30000000     0 FUNC    LOCAL  DEFAULT    1 $a
    13: 30000004     0 NOTYPE  LOCAL  DEFAULT    1 step1
    14: 30000008     0 NOTYPE  LOCAL  DEFAULT    1 step2
    15: 3000000c     0 OBJECT  LOCAL  DEFAULT    1 $d
    16: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND _start
    17: 30008010     0 NOTYPE  GLOBAL DEFAULT    2 __data_start
    18: 30008010     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start__
    19: 30008010     0 NOTYPE  GLOBAL DEFAULT  ABS __end__
    20: 30008010     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_end__
    21: 30008010     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start
    22: 30008010     0 NOTYPE  GLOBAL DEFAULT  ABS _bss_end__
    23: 30008010     0 NOTYPE  GLOBAL DEFAULT  ABS _end
    24: 30008010     0 NOTYPE  GLOBAL DEFAULT  ABS _edata
```

符号表的value本来是表示符号相对于节的偏移量的，但是这个文件是**可执行的目标文件**会对每个符号重新分配地址，再次验证-Ttext表示的就是代码段的运行地址。

除了用上面的-Ttext的参数来指定代码段的位置，当然我们也可以用脚本来设置，具体请看<嵌入式linux完全开发手册>的第3章 3.1的arm-linux-ld选项这一节。