ELF(Executable and Linking Format)是一种对象文件的格式，用于定义不同类型的对象文件(Object files)中都放了什么东西、以及都以什么样的格式去放这些东西。它自最早在 System V 系统上出现后，被 xNIX 世界所广泛接受，作为缺省的二进制文件格式来使用。可以说，ELF是构成众多xNIX系统的基础之一，所以作为嵌入式Linux系统乃至内核驱动程序开发人员，你最好熟悉并掌握它。

其实，关于ELF这个主题，网络上已经有相当多的文章存在，但是其介绍的内容比较分散，使得初学者不太容易从中得到一个系统性的认识。为了帮助大家学习，我这里打算写一系列连贯的文章来介绍ELF以及相关的应用。这是这个系列中的第一篇文章，主要是通过不同工具的使用来熟悉ELF文件的内部结构以及相关的基本概念。后面的文章，我们会介绍很多高级的概念和应用，比方动态链接和加载，动态库的开发，C语言Main函数是被谁以及如何被调用的，ELF格式在内核中的支持，Linux内核中对ELF section的扩展使用等等。

好的，开始我们的第一篇文章。在详细进入正题之前，先给大家介绍一点ELF文件格式的参考资料。在ELF格式出来之后，TISC(Tool Interface Standard Committee)委员会定义了一套ELF标准。你可以从这里(http://refspecs.freestandards.org/elf/)找到详细的标准文档。TISC委员会前后出了两个版本，v1.1和v1.2。两个版本内容上差不多，但就可读性上来讲，我还是推荐你读 v1.2的。因为在v1.2版本中，TISC重新组织原本在v1.1版本中的内容，将它们分成为三个部分(books)：

a) Book I

介绍了通用的适用于所有32位架构处理器的ELF相关内容

b) Book II

介绍了处理器特定的ELF相关内容，这里是以Intel x86 架构处理器作为例子介绍

c) Book III

介绍了操作系统特定的ELF相关内容，这里是以运行在x86上面的 UNIX System V.4 作为例子介绍

值得一说的是，虽然TISC是以x86为例子介绍ELF规范的，但是如果你是想知道非x86下面的ELF实现情况，那也可以在http://refspecs.freestandards.org/elf/中找到特定处理器相关的Supplment文档。比方ARM相关的，或者MIPS相关的等等。另外，相比较UNIX系统的另外一个分支BSD Unix，Linux系统更靠近 System V 系统。所以关于操作系统特定的ELF内容，你可以直接参考v1.2标准中的内容。

这里多说些废话：别忘了 Linus 在实现Linux的第一个版本的时候，就是看了介绍Unix内部细节的书：《The of the Unix Operating System》，得到很多启发。这本书对应的操作系统是System V 的第二个Release。这本书介绍了操作系统的很多设计观念，并且行文简单易懂。所以虽然现在的Linux也吸取了其他很多Unix变种的设计理念，但是如果你想研究学习Linux内核，那还是以看这本书作为开始为好。这本书也是我在接触Linux内核之前所看的第一本介绍操作系统的书，所以我极力向大家推荐。(在学校虽然学过操作系统原理，但学的也是很糟糕最后导致期末考试才四十来分，记忆仿佛还在昨天:))

好了，还是回来开始我们第一篇ELF主题相关的文章吧。这篇文章主要是通过使用不同的工具来分析对象文件，来使你掌握ELF文件的基本格式，以及了解相关的基本概念。你在读这篇文章的时候，希望你在电脑上已经打开了那个 v1.2 版本的ELF规范，并对照着文章内容看规范里的文字。

首先，你需要知道的是所谓对象文件(Object files)有三个种类：

1) 可重定位的对象文件(Relocatable file)

这是由汇编器汇编生成的 .o 文件。后面的链接器(link editor)拿一个或一些 Relocatable object files 作为输入，经链接处理后，生成一个可执行的对象文件 (Executable file) 或者一个可被共享的对象文件(Shared object file)。我们可以使用 ar 工具将众多的 .o Relocatable object files 归档(archive)成 .a 静态库文件。如何产生 Relocatable file，你应该很熟悉了，请参见我们相关的基本概念文章和JulWiki。另外，可以预先告诉大家的是我们的内核可加载模块 .ko 文件也是 Relocatable object file。

2) 可执行的对象文件(Executable file)

这我们见的多了。文本编辑器vi、调式用的工具gdb、播放mp3歌曲的软件mplayer等等都是Executable object file。你应该已经知道，在我们的 Linux 系统里面，存在两种可执行的东西。除了这里说的 Executable object file，另外一种就是可执行的脚本(如shell脚本)。注意这些脚本不是 Executable object file，它们只是文本文件，但是执行这些脚本所用的解释器就是 Executable object file，比如 bash shell 程序。

3) 可被共享的对象文件(Shared object file)

这些就是所谓的动态库文件，也即 .so 文件。如果拿前面的静态库来生成可执行程序，那每个生成的可执行程序中都会有一份库代码的拷贝。如果在磁盘中存储这些可执行程序，那就会占用额外的磁盘空间；另外如果拿它们放到Linux系统上一起运行，也会浪费掉宝贵的物理内存。如果将静态库换成动态库，那么这些问题都不会出现。动态库在发挥作用的过程中，必须经过两个步骤：

a) 链接编辑器(link editor)拿它和其他Relocatable object file以及其他shared object file作为输入，经链接处理后，生存另外的 shared object file 或者 executable file。

b) 在运行时，动态链接器(dynamic linker)拿它和一个Executable file以及另外一些 Shared object file 来一起处理，在Linux系统里面创建一个进程映像。

以上所提到的 link editor 以及 dynamic linker 是什么东西，你可以参考我们基本概念中的相关文章。对于什么是编译器，汇编器等你应该也已经知道，在这里只是使用他们而不再对他们进行详细介绍。为了下面的叙述方便，你可以下载test.tar.gz包，解压缩后使用"make"进行编译。编译完成后，会在目录中生成一系列的ELF对象文件，更多描述见里面的 README 文件。我们下面的论述都基于这些产生的对象文件。

make所产生的文件，包括 sub.o/sum.o/test.o/libsub.so/test 等等都是ELF对象文件。至于要知道它们都属于上面三类中的哪一种，我们可以使用 file 命令来查看：

[yihect@juliantec test]$ file sum.o sub.o test.o libsub.so test   
sum.o:     ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped   
sub.o:     ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped   
test.o:    ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped   
libsub.so: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), not stripped   
test:      ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.5, dynamically linked (uses shared libs), not stripped

结果很清楚的告诉我们他们都属于哪一个类别。比方 sum.o 是应用在x86架构上的可重定位文件。这个结果也间接的告诉我们，x86是小端模式(LSB)的32位结构。那对于 file 命令来说，它又能如何知道这些信息？答案是在ELF对象文件的最前面有一个ELF文件头，里面记载了所适用的处理器、对象文件类型等各种信息。在TISCv1.2的规范中，用下面的图描述了ELF对象文件的基本组成，其中ELF文件头赫然在目。

等等，为什么会有左右两个很类似的图来说明ELF的组成格式？这是因为ELF格式需要使用在两种场合：

a) 组成不同的可重定位文件，以参与可执行文件或者可被共享的对象文件的链接构建；

b) 组成可执行文件或者可被共享的对象文件，以在运行时内存中进程映像的构建。

所以，基本上，图中左边的部分表示的是可重定位对象文件的格式；而右边部分表示的则是可执行文件以及可被共享的对象文件的格式。正如TISCv1.2规范中所阐述的那样，ELF文件头被固定地放在不同类对象文件的最前面。至于它里面的内容，除了file命令所显示出来的那些之外，更重要的是包含另外一些数据，用于描述ELF文件中ELF文件头之外的内容。如果你的系统中安装有 GNU binutils 包，那我们可以使用其中的 readelf 工具来读出整个ELF文件头的内容，比如：

[yihect@juliantec test]$ readelf -h ./sum.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: 184 (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: 9

Section header string table index: 6

这个输出结果能反映出很多东西。那如何来看这个结果中的内容，我们还是就着TISCv1.2规范来。在实际写代码支持ELF格式对象文件格式的时候，我们都会定义许多C语言的结构来表示ELF格式的各个相关内容，比方这里的ELF文件头，你就可以在TISCv1.2规范中找到这样的结构定义(注意我们研究的是针对x86架构的ELF，所以我们只考虑32位版本，而不考虑其他如64位之类的)：

这个结构里面出现了多种数据类型，同样可以在规范中找到相关说明：

在我们以后一系列文章中，我们会着重拿实际的程序代码来分析，介时你会在头文件中找到同样的定义。但是这里，我们只讨论规范中的定义，暂不考虑任何程序代码。在ELF头中，字段e\_machine和e\_type指明了这是针对x86架构的可重定位文件，最前面有个长度为16字节的字段中有一个字节表示了它适用于32bits机器，而不是64位的。除了这些之外，另外ELF头还告诉了我们其他一些特别重要的信息，分别是：

a) 这个sum.o的进入点是0x0(e\_entry)，这表面Relocatable objects不会有程序进入点。所谓程序进入点是指当程序真正执行起来的时候，其第一条要运行的指令的运行时地址。因为Relocatable objects file只是供再链接而已，所以它不存在进入点。而可执行文件test和动态库.so都存在所谓的进入点，你可以用 readelf -h 看看。后面我们的文章中会介绍可执行文件的e\_entry指向C库中的\_start，而动态库.so中的进入点指向 call\_gmon\_start。这些后面再说，这里先不深入讨论。

b) 这个sum.o文件包含有9个sections，但却没有segments(Number of program headers为0)。

那什么是所谓 sections 呢？可以说，sections 是在ELF文件里头，用以装载内容数据的最小容器。在ELF文件里面，每一个 sections 内都装载了性质属性都一样的内容，比方：

1) .text section 里装载了可执行代码；

2) .data section 里面装载了被初始化的数据；

3) .bss section 里面装载了未被初始化的数据；

4) 以 .rec 打头的 sections 里面装载了重定位条目；

5) .symtab 或者 .dynsym section 里面装载了符号信息；

6) .strtab 或者 .dynstr section 里面装载了字符串信息；

7) 其他还有为满足不同目的所设置的section，比方满足调试的目的、满足动态链接与加载的目的等等。

一个ELF文件中到底有哪些具体的 sections，由包含在这个ELF文件中的 section head table(SHT)决定。在SHT中，针对每一个section，都设置有一个条目，用来描述对应的这个section，其内容主要包括该 section 的名称、类型、大小以及在整个ELF文件中的字节偏移位置等等。我们也可以在TISCv1.2规范中找到SHT表中条目的C结构定义：

我们可以像下面那样来使用 readelf 工具来查看可重定位对象文件 sum.o 的SHT表内容：[yihect@juliantec test]$ readelf -S ./sum.o   
There are 9 section headers, starting at offset 0xb8:   
    
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 00000b 00  AX  0   0  4   
  [ 2] .data             PROGBITS        00000000 000040 000004 00  WA  0   0  4   
  [ 3] .bss              NOBITS          00000000 000044 000000 00  WA  0   0  4   
  [ 4] .note.GNU-stack   PROGBITS        00000000 000044 000000 00      0   0  1   
  [ 5] .comment          PROGBITS        00000000 000044 00002d 00      0   0  1   
  [ 6] .shstrtab         STRTAB          00000000 000071 000045 00      0   0  1   
  [ 7] .symtab           SYMTAB          00000000 000220 0000a0 10      8   7  4   
  [ 8] .strtab           STRTAB          00000000 0002c0 00001d 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)

这个结果显示了 sum.o 中包含的所有9个sections。因为sum.o仅仅是参与link editor链接的可重定位文件，而不参与最后进程映像的构建，所以Addr(sh\_addr)为0。后面你会看到可执行文件以及动态库文件中大部分sections的这一字段都是有某些取值的(重定位是在将动态链接库和可执行程序装载到内存中时才会发生的)。Off(sh\_offset)表示了该section离开文件头部位置的距离。Size(sh\_size)表示section的字节大小。ES(sh\_entsize)只对某些形式的sections 有意义。比方符号表 .symtab section，其内部包含了一个表格，表格的每一个条目都是特定长度的，那这里的这个字段就表示条目的长度10。Al(sh\_addralign)是地址对齐要求。另外剩下的两列Lk和Inf，对应着条目结构中的字段sh\_link和字段sh\_info。它们中记录的是section head table 中的条目索引，这就意味着，从这两个字段出发，可以找到对应的另外两个 section，其具体的含义解释依据不同种类的 section 而不同，后面会介绍。

注意上面结果中的 Flg ，表示的是对应section的相关标志。比方.text section 里面存储的是代码，所以就是只读的(X)；.data和.bss里面存放的都是可写的(W)数据(非在堆栈中定义的数据)，只不过前者存的是初始化过的数据，比方程序中定义的赋过初值的全局变量等；而后者里面存储的是未经过初始化的数据。因为未经过初始化就意味着不确定这些数据刚开始的时候会有些什么样的值，所以针对对象文件来说，它就没必要为了存储这些数据而在文件内多留出一块空间，因此.bss section的大小总是为0。后面会看到，当可执行程序被执行的时候，动态连接器会在内存中开辟一定大小的空间来存放这些未初始化的数据，里面的内存单元都被初始化成0。可执行程序文件中虽然没有长度非0的 .bss section，但却记录有在程序运行时，需要开辟多大的空间来容纳这些未初始化的数据。

另外一个标志A说明对应的 section 是Allocable的。所谓 Allocable 的section，是指在运行时，进程(process)需要使用它们，所以它们被加载器加载到内存中去。

而与此相反，存在一些non-Allocable 的sections，它们只是被链接器、调试器或者其他类似工具所使用的，而并非参与进程的运行中去的那些 section。比方后面要介绍的字符串表section .strtab，符号表 .symtab section等等。当运行最后的可执行程序时，加载器会加载那些 Allocable 的部分，而 non-Allocable 的部分则会被继续留在可执行文件内。所以，实际上，这些 non-Allocable 的section 都可以被我们用 stip 工具从最后的可执行文件中删除掉，删除掉这些sections的可执行文件照样能够运行，只不过你没办法来进行调试之类的事情罢了。

我们仍然可以使用 readelf -x SecNum 来倾印出不同 section 中的内容。但是，无奈其输出结果都是机器码，对我们人来说不具备可读性。所以我们换用 binutils 包中的另外一个工具 objdump 来看看这些 sections 中到底具有哪些内容，先来看看 .text section 的：[yihect@juliantec test]$ objdump -d -j .text ./sum.o   
    
./sum.o:     file format elf32-i386   
    
Disassembly of section .text:   
    
00000000 :   
   0:   55                      push   %ebp   
   1:   89 e5                   mov    %esp,%ebp   
   3:   8b 45 0c                mov    0xc(%ebp),%eax   
   6:   03 45 08                add    0x8(%ebp),%eax   
   9:   c9                      leave    
   a:   c3                      ret

objdump 的选项 -d 表示要对由 -j 选择项指定的 section 内容进行反汇编，也就是由机器码出发，推导出相应的汇编指令。上面结果显示在 sum.o 对象文件的 .text 中只是包含了函数 sum\_func 的定义。用同样的方法，我们来看看 sum.o 中 .data section 有什么内容：[yihect@juliantec test]$ objdump -d -j .data  ./sum.o   
    
./sum.o:     file format elf32-i386   
    
Disassembly of section .data:   
    
00000000 :   
   0:   17 00 00 00                                         ....

这个结果显示在 sum.o 的 .data section 中定义了一个四字节的变量 gv\_inited，其值被初始化成 0x00000017，也就是十进制值 23。别忘了，x86架构是使用小端模式的。

我们接下来来看看字符串表section .strtab。你可以选择使用 readelf -x :

[yihect@juliantec test]$ readelf -x 8 ./sum.o   
    
Hex dump of section '.strtab':   
  0x00000000 64657469 6e695f76 6700632e 6d757300 .sum.c.gv\_inited   
  0x00000010       00 68630063 6e75665f 6d757300 .sum\_func.ch.

上面命令中的 8 是 .strtab section 在SHT表格中的索引值，从上面所查看的SHT内容中可以找到。尽管这个命令的输出结果不是那么具有可读性，但我们还是得来说一说如何看这个结果，因为后续文章中将会使用大量的这种命令。上面结果中的十六进制数据部分从右到左看是地址递增的方向，而字符内容部分从左到右看是地址递增的方向。所以，在 .strtab section 中，按照地址递增的方向来看，各字节的内容依次是 0x00、0x73、0x75、0x6d、0x2e ....，也就是字符 、's'、'u'、'm'、'.' ... 等。如果还是看不太明白，你可以使用 hexdump 直接dumping出 .strtab section 开头(其偏移在文件内0x2c0字节处)的 32 字节数据：

[yihect@juliantec test]$ hexdump -s 0x2c0 -n 32 -c ./sum.o   
00002c0     s   u   m   .   c     g   v   \_   i   n   i   t   e   d   
00002d0     s   u   m   \_   f   u   n   c     c   h                
00002dd

.strtab section 中存储着的都是以字符 为分割符的字符串，这些字符串所表示的内容，通常是程序中定义的函数名称、所定义过的变量名称等等。。。当对象文件中其他地方需要和一个这样的字符串相关联的时候，往往会在对应的地方存储 .strtab section 中的索引值。比方下面将要介绍的符号表 .symtab section 中，有一个条目是用来描述符号 gv\_inited 的，那么在该条目中就会有一个字段(st\_name)记录着字符串 gv\_inited 在 .strtab section 中的索引 7 。 .shstrtab 也是字符串表，只不过其中存储的是 section 的名字，而非所函数或者变量的名称。

字符串表在真正链接和生成进程映像过程中是不需要使用的，但是其对我们调试程序来说就特别有帮助，因为我们人看起来最舒服的还是自然形式的字符串，而非像天书一样的数字符号。前面使用objdump来反汇编 .text section 的时候，之所以能看到定义了函数 sum\_func ，那也是因为存在这个字符串表的原因。当然起关键作用的，还是符号表 .symtab section 在其中作为中介，下面我们就来看看符号表。

虽然我们同样可以使用 readelf -x 来查看符号表(.symtab)section的内容，但是其结果可读性太差，我们换用 readelf -s 或者 objdump -t 来查看(前者输出结果更容易看懂)：

[yihect@juliantec test]$ readelf -s ./sum.o   
    
Symbol table '.symtab' contains 10 entries:   
   Num:    Value  Size Type    Bind   Vis      Ndx Name   
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND   
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS sum.c   
     2: 00000000     0 SECTION LOCAL  DEFAULT    1   
     3: 00000000     0 SECTION LOCAL  DEFAULT    2   
     4: 00000000     0 SECTION LOCAL  DEFAULT    3   
     5: 00000000     0 SECTION LOCAL  DEFAULT    4   
     6: 00000000     0 SECTION LOCAL  DEFAULT    5   
     7: 00000000     4 OBJECT  GLOBAL DEFAULT    2 gv\_inited   
     8: 00000000    11 FUNC    GLOBAL DEFAULT    1 sum\_func   
     9: 00000001     1 OBJECT  GLOBAL DEFAULT  COM ch

在符号表内针对每一个符号，都会相应的设置一个条目。在继续介绍上面的结果之前，我们还是从规范中找出符号表内条目的C结构定义：

上面结果中 Type 列显示出符号的种类。Bind 列定义了符号的绑定类型。种类和绑定类型合并在一起，由结构中 st\_info 字段来定义。在ELF格式中，符号类型总共可以有这么几种：

类型 STT\_OBJECT 表示和该符号对应的是一个数据对象，比方程序中定义过的变量、数组等，比方上面的 gv\_inited 和 ch；类型 STT\_FUNC 表示该符号对应的是函数，比方上面的 sum\_func函数。类型 STT\_SECTION 表示该符号和一个 section 相关，这种符号用于重定位。关于重定位，我们下文会介绍。

符号的绑定类型表示了这个符号的可见性，是仅本对象文件可见呢，还是全局可见。它的取值主要有三种：STB\_LOCA、STB\_GLOBAL和STB\_WEAK，具体的内容还请参见规范。关于符号，最重要的就是符号的值(st\_value)了。依据对象文件的不同类型，符号的值所表示的含义也略有差异：

a) 在可重定位文件中，如果该符号对应的section index(上面的Ndx)为SHN\_COMMON，那么符号的值表示的是该数据的对齐要求，比方上面的变量 ch 。

b) 在可重定位文件中，除去上面那条a中定义的符号，对于其他的符号来说，其值表示的是对应 section 内的偏移值。比方 gv\_inited 变量定义在 .data section 的最前面，所以其值为0。

c) 在可执行文件或者动态库中，符号的值表示的是运行时的内存地址。

好，咱们再来介绍重定位。在所产生的对象文件 test.o 中有对函数 sum\_func 的引用，这对我们的x386结构来说，其实就是一条call指令。既然 sum\_func 是定义在 sum.o 中的，那对 test.o 来说，它就是一个外部引用。所以，汇编器在产生 test.o 的时候，它会产生一个重定位条目。重定位条目中会包含以下几类东西：

1) 它会包含一个符号表中一个条目的索引，因为这样我们才知道它具体是哪个符号需要被重定位的；

2) 它会包含一个 .text section 中的地址单元的偏移值。原本这个偏移值处的地址单元里面应该存放着 call 指令的操作数。对上面来说，也就是函数 sum\_func 的地址，但是目前这个地址汇编器还不知道。

3) 它还会包含一个tag，以指明该重定位属于何种类型。

当我们用链接器去链接这个对象文件的时候，链接器会遍历所有的重定位条目，碰到像 sum\_func 这样的外部引用，它会找到 sum\_func 的确切地址，并且把它写回到上面 call 指令操作数所占用的那个地址单元。像这样的操作，称之为重定位操作。link editor 和 dynamic linker 都要完成一些重定位操作，只不过后者的动作更加复杂，因为它是在运行时动态完成的，我们以后的文章会介绍相关的内容。概括一下，所谓重定位操作就是：“汇编的时候产生一个空坐位，上面用红纸写着要坐在这个座位上的人的名字，然后连接器在开会前安排那个人坐上去”。

如前面我们说过的，对象文件中的重定位条目，会构成一个个单独的 section。这些 section 的名字，常会是这样的形式：".rel.XXX"。其中XXX表示的是这些重定位条目所作用到的section，如 .text section。重定位条目所构成的section需要和另外两个section产生关联：符号表section(表示要重定位的是哪一个符号)以及受影响地址单元所在的section。在使用工具来查看重定位section之前，我们先从规范中找出来表示重定位条目的结构定义(有两种，依处理器架构来定)：

结构中 r\_offset 对于可重定位文件.o来说，就是地址单元的偏移值(前面的b条)；另外对可执行文件或者动态库来说，就是该地址单元的运行时地址。上面 a条中的符号表内索引和c条中的类型，一起构成了结构中的字段 r\_info。

重定位过程在计算最终要放到受影响地址单元中的时候，需要加上一个附加的数 addend。当某一种处理器选用 Elf32\_Rela 结构的时候，该 addend 就是结构中的 r\_addend 字段；否则该 addend 就是原本存储在受影响地址单元中的原有值。x86架构选用 Elf32\_Rel 结构来表示重定位条目。ARM架构也是用这个。

重定位类型意味着如何去修改受影响的地址单元，也就是按照何种方式去计算需要最后放在受影响单元里面的值。具体的重定位类型有哪些，取决与特定的处理器架构，你可以参考相关规范。这种计算方式可以非常的简单，比如在x386上的 R\_386\_32 类型，它规定只是将附加数加上符号的值作为所需要的值；该计算方式也可以是非常的复杂，比如老版本ARM平台上的 R\_ARM\_PC26。在这篇文章的末尾，我会详细介绍一种重定位类型：R\_386\_PC32。至于另外一些重要的重定位类型，如R\_386\_GOTPC，R\_386\_PLT32，R\_386\_GOT32，R\_386\_GLOB\_DAT 以及 R\_386\_JUMP\_SLOT 等。读者可以先自己研究，也许我们会在后面后面的文章中讨论到相关主题时再行介绍。

我们可以使用命令 readelf -r 来查看重定位信息：

[yihect@juliantec test\_2]$ readelf -r test.o   
    
Relocation section '.rel.text' at offset 0x464 contains 8 entries:   
Offset     Info    Type            Sym.Value  Sym. Name   
00000042  00000902 R\_386\_PC32        00000000   sub\_func   
00000054  00000a02 R\_386\_PC32        00000000   sum\_func   
0000005d  00000a02 R\_386\_PC32        00000000   sum\_func   
0000007a  00000501 R\_386\_32          00000000   .rodata   
0000007f  00000b02 R\_386\_PC32        00000000   [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)   
0000008d  00000c02 R\_386\_PC32        00000000   double\_gv\_inited   
00000096  00000501 R\_386\_32          00000000   .rodata   
0000009b  00000b02 R\_386\_PC32        00000000   [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)

至此，ELF对象文件格式中的 linking view ，也就是上面组成图的左边部分，我们已经介绍完毕。在这里最重要的概念是 section。在可重定位文件里面，section承载了大多数被包含的东西，代码、数据、符号信息、重定位信息等等。可重定位对象文件里面的这些sections是作为输入，给链接器那去做链接用的，所以这些 sections 也经常被称做输入 section。

链接器在链接可执行文件或动态库的过程中，它会把来自不同可重定位对象文件中的相同名称的 section 合并起来构成同名的 section。接着，它又会把带有相同属性(比方都是只读并可加载的)的 section 都合并成所谓 segments(段)。segments 作为链接器的输出，常被称为输出section。我们开发者可以控制哪些不同.o文件的sections来最后合并构成不同名称的 segments。如何控制呢，就是通过 linker script 来指定。关于链接器脚本，我们这里不予讨论。

一个单独的 segment 通常会包含几个不同的 sections，比方一个可被加载的、只读的segment 通常就会包括可执行代码section .text、只读的数据section .rodata以及给动态链接器使用的符号section .dymsym等等。section 是被链接器使用的，但是 segments 是被加载器所使用的。加载器会将所需要的 segment 加载到内存空间中运行。和用 sections header table 来指定一个可重定位文件中到底有哪些 sections 一样。在一个可执行文件或者动态库中，也需要有一种信息结构来指出包含有哪些 segments。这种信息结构就是 program header table，如ELF对象文件格式中右边的 execute view 所示的那样。

我们可以用 readelf -l 来查看可执行文件的程序头表，如下所示：

[yihect@juliantec test\_2]$ readelf -l ./test   
    
Elf file type is EXEC (Executable file)   
Entry point 0x8048464   
There are 7 program headers, starting at offset 52   
    
Program Headers:   
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align   
  PHDR           0x000034 0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4   
  INTERP         0x000114 0x08048114 0x08048114 0x00013 0x00013 R   0x1   
      [Requesting program interpreter: /lib/ld-linux.so.2]   
  LOAD           0x000000 0x08048000 0x08048000 0x0073c 0x0073c R E 0x1000   
  LOAD           0x00073c 0x0804973c 0x0804973c 0x00110 0x00118 RW  0x1000   
  DYNAMIC        0x000750 0x08049750 0x08049750 0x000d0 0x000d0 RW  0x4   
  NOTE           0x000128 0x08048128 0x08048128 0x00020 0x00020 R   0x4   
  GNU\_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4   
    
Section to Segment mapping:   
  Segment Sections...   
   00       
   01     .interp   
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version\_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh\_frame   
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss   
   04     .dynamic   
   05     .note.ABI-tag   
   06

结果显示，在可执行文件 ./test 中，总共有7个 segments。同时，该结果也很明白显示出了哪些 section 映射到哪一个 segment 当中去。比方在索引为2的那个segment 中，总共有15个 sections 映射进来，其中包括我们前面提到过的 .text section。注意这个segment 有两个标志: R 和 E。这个表示该segment是可读的，也可执行的。如果你看到标志中有W，那表示该segment是可写的。

我们还是来解释一下上面的结果，希望你能对照着TISCv1.2规范里面的文本来看，我这里也列出程序头表条目的C结构：

上面类型为PHDR的segment，用来包含程序头表本身。类型为INTERP的segment只包含一个 section，那就是 .interp。在这个section中，包含了动态链接过程中所使用的解释器路径和名称。在Linux里面，这个解释器实际上就是 /lib/ ，这可以通过下面的 hexdump 看出来：[yihect@juliantec test\_2]$ hexdump -s 0x114 -n 32 -C  ./test    
00000114  2f 6c 69 62 2f 6c 64 2d  6c 69 6e 75 78 2e 73 6f  |/lib/ld-linux.so|   
00000124  2e 32 00 00 04 00 00 00  10 00 00 00 01 00 00 00  |.2..............|   
00000134

为什么会有这样的一个 segment？这是因为我们写的应用程序通常都需要使用动态链接库.so，就像 test 程序中所使用的 libsub.so 一样。我们还是先大致说说程序在linux里面是怎么样运行起来的吧。当你在 shell 中敲入一个命令要执行时，内核会帮我们创建一个新的进程，它在往这个新进程的进程空间里面加载进可执行程序的代码段和数据段后，也会加载进动态连接器(在Linux里面通常就是 /lib/ld-linux.so 符号链接所指向的那个程序，它本省就是一个动态库)的代码段和数据。在这之后，内核将控制传递给动态链接库里面的代码。动态连接器接下来负责加载该命令应用程序所需要使用的各种动态库。加载完毕，动态连接器才将控制传递给应用程序的main函数。如此，你的应用程序才得以运行。

这里说的只是大致的应用程序启动运行过程，更详细的，我们会在后续的文章中继续讨论。我们说link editor链接的应用程序只是部分链接过的应用程序。经常的，在应用程序中，会使用很多定义在动态库中的函数。最最基础的比方C函数库(其本身就是一个动态库)中定义的函数，每个应用程序总要使用到，就像我们test程序中使用到的 printf 函数。为了使得应用程序能够正确使用动态库，动态连接器在加载动态库后，它还会做更进一步的链接，这就是所谓的动态链接。为了让动态连接器能成功的完成动态链接过程，在前面运行的link editor需要在应用程序可执行文件中生成数个特殊的 sections，比方 .dynamic、.dynsym、.got和.plt等等。这些内容我们会在后面的文章中进行讨论。

我们先回到上面所输出的文件头表中。在接下来的数个 segments 中，最重要的是三个 segment：代码段，数据段和堆栈段。代码段和堆栈段的 VirtAddr 列的值分别为 0x08048000 和 0x0804973c。这是什么意思呢？这是说对应的段要加载在进程虚拟地址空间中的起始地址。虽然在可执行文件中规定了 text segment和 data segment 的起始地址，但是最终，在内存中的这些段的真正起始地址，却可能不是这样的，因为在动态链接器加载这些段的时候，需要考虑到页面对齐的因素。为什么？因为像x86这样的架构，它给内存单元分配读写权限的最小单位是页(page)而不是字节。也就是说，它能规定从某个页开始、连续多少页是只读的。却不能规定从某个页内的哪一个字节开始，连续多少个字节是只读的。因为x86架构中，一个page大小是4k，所以，动态链接器在加载 segment 到虚拟内存中的时候，其真实的起始地址的低12位都是零，也即以 0x1000 对齐。

我们先来看看一个真实的进程中的内存空间信息，拿我们的 test 程序作为例子。在 Linux 系统中，有一个特殊的由内核实现的虚拟文件系统 /proc。内核实现这个文件系统，并将它作为整个Linux系统面向外部世界的一个接口。我们可以通过 /proc 观察到一个正在运行着的Linux系统的内核数据信息以及各进程相关的信息。所以我们如果要查看某一个进程的内存空间情况，也可以通过它来进行。使用/proc唯一需要注意的是，由于我们的 test 程序很小，所以当我们运行起来之后，它很快就会结束掉，使得我们没有时间去查看test的进程信息。我们需要想办法让它继续运行，或者最起码运行直到让我们能从 /proc 中获取得到想要的信息后再结束。

我们有多种选择。最简单的是，在 test main 程序中插入一个循环，然后在循环中放入 sleep() 的调用，这样当程序运行到这个循环的时候，就会进入“运行-睡眠-运行-睡眠”循环中。这样我们就有机会去看它的虚拟内存空间信息。另外一个方法，是使用调试器，如GDB。我们设置一个断点，然后在调试过程中让test进程在这个断点处暂停，这样我们也有机会获得地址空间的信息。我们这里就使用这种方法。当然，为了能让GDB调试我们的 test，我们得在编译的时候加上"-g"选项。最后我们用下面的命令得到 test 程序对应进程的地址空间信息。

[yihect@juliantec ~]$ cat /proc/`pgrep test`/maps   
00103000-00118000 r-xp 00000000 08:02 544337     /lib/ld-2.3.4.so   
00118000-00119000 r--p 00015000 08:02 544337     /lib/ld-2.3.4.so   
00119000-0011a000 rw-p 00016000 08:02 544337     /lib/ld-2.3.4.so   
0011c000-00240000 r-xp 00000000 08:02 544338     /lib/tls/libc-2.3.4.so   
00240000-00241000 r--p 00124000 08:02 544338     /lib/tls/libc-2.3.4.so   
00241000-00244000 rw-p 00125000 08:02 544338     /lib/tls/libc-2.3.4.so   
00244000-00246000 rw-p 00244000 00:00 0   
00b50000-00b51000 r-xp 00000000 08:02 341824     /usr/lib/libsub.so   
00b51000-00b52000 rw-p 00000000 08:02 341824     /usr/lib/libsub.so   
08048000-08049000 r-xp 00000000 08:05 225162     /home/yihect/test\_2/test   
08049000-0804a000 rw-p 00000000 08:05 225162     /home/yihect/test\_2/test   
b7feb000-b7fed000 rw-p b7feb000 00:00 0   
b7fff000-b8000000 rw-p b7fff000 00:00 0   
bff4c000-c0000000 rw-p bff4c000 00:00 0   
ffffe000-fffff000 ---p 00000000 00:00 0

注意，上面命令中的pgre test 是用`括起来的，它不是单引号，而是键盘上 Esc 字符下面的那个字符。从这个结果上可以看出，所有的段，其起始地址和结束地址(前面两列)都是0x1000对齐的。结果中也列出了对应的段是从哪里引过来的，比方动态链接器/lib/ld-2.3.4.so、C函数库和test程序本身。注意看test程序引入的代码段起始地址是 0x08048000，这和我们 ELF 文件中指定的相同，但是结束地址却是0x08049000，和文件中指定的不一致(0x08048000+0x0073c=0x0804873c)。这里，其实加载器也把数据segment中开头一部分也映射进了 text segment 中去；同样的，进程虚拟内存空间中的 data segment 从 08049000 开始，而可执行文件中指定的是从 0x0804973c 开始。所以加载器也把代码segment中末尾一部分也映射进了 data segment 中去了。

从程序头表中我们可以看到一个类型为 GNU\_STACK 的segment，这是 stack segment。程序头表中的这一项，除了 Flg/Align 两列不为空外， 其他列都为0。这是因为堆栈段在虚拟内存空间中，从哪里开始、占多少字节是由内核说了算的，而不决定于可执行程序。实际上，内核决定把堆栈段放在整个进程地址空间的用户空间的最上面，所以堆栈段的末尾地址就是 0xc0000000。别忘记在 x86 中，堆栈是从高向低生长的。

好，为了方便你对后续文章的理解，我们在这里讨论一种比较简单的重定位类型 R\_386\_PC32。前面我们说过重定义的含义，也即在连接阶段，根据某种计算方式计算出一个新的值(通常是地址)，然后将这个值重新改写到对象文件或者内存映像中某个section中的某个地址单元中去的这样一个过程。那所谓重定位类型，就规定了使用何种方式，去计算这个值。既然是计算，那就肯定需要涉及到所要纳入计算的变量。实际上，具体有哪些变量参与计算如同如何进行计算一样也是不固定的，各种重定位类型有自己的规定。

根据规范里面的规定，重定位类型 R\_386\_PC32 的计算需要有三个变量参与：S，A和P。其计算方式是 S+A-P。根据规范，当R\_386\_PC32类型的重定位发生在 link editor 链接若干个 .o 对象文件从而形成可执行文件的过程中的时候，变量S指代的是被重定位的符号的实际运行时地址，而变量P是重定位所影响到的地址单元的实际运行时地址。在运行于x86架构上的Linux系统中，这两个地址都是虚拟地址。变量A最简单，就是重定位所需要的附加数，它是一个常数。别忘了x86架构所使用的重定位条目结构体类型是 Elf32\_Rela，所以附加数就存在于受重定位影响的地址单元中。重定位最后将计算得到的值patch到这个地址单元中。

或许，咱们举一个实际例子来阐述可能对你更有用。在我们的 test 程序中，test.c 的 main 函数中需要调用定义在 sum.o 中的 sum\_func 函数，所以link editor 在将 test.o/sum.o 联结成可执行文件 test 的时候，必须处理一个重定位，这个重定位就是 R\_386\_PC32 类型的。我们先用 objdump 来查看 test.o 中的 .text section 内容(我只选取了前面一部分)：[yihect@juliantec test\_2]$ objdump -d -j .text ./test.o   
    
./test.o:     file format elf32-i386   
    
Disassembly of section .text:   
    
00000000 <main />:   
   0:   55                      push   %ebp   
   1:   89 e5                   mov    %esp,%ebp   
   3:   83 ec 18                sub    $0x18,%esp   
   6:   83 e4 f0                and    $0xfffffff0,%esp   
   9:   b8 00 00 00 00          mov    $0x0,%eax   
   e:   83 c0 0f                add    $0xf,%eax   
  11:   83 c0 0f                add    $0xf,%eax   
  14:   c1 e8 04                shr    $0x4,%eax   
  17:   c1 e0 04                shl    $0x4,%eax   
  1a:   29 c4                   sub    %eax,%esp   
  1c:   c7 45 fc 0a 00 00 00    movl   $0xa,0xfffffffc(%ebp)   
  23:   c7 45 f8 2d 00 00 00    movl   $0x2d,0xfffffff8(%ebp)   
  2a:   c7 45 f4 03 00 00 00    movl   $0x3,0xfffffff4(%ebp)   
  31:   c7 45 f0 48 00 00 00    movl   $0x48,0xfffffff0(%ebp)   
  38:   83 ec 08                sub    $0x8,%esp   
  3b:   ff 75 f0                pushl  0xfffffff0(%ebp)   
  3e:   ff 75 f4                pushl  0xfffffff4(%ebp)   
  41:   e8 fc ff ff ff          call   42   
  46:   83 c4 08                add    $0x8,%esp   
  49:   50                      push   %eax   
  4a:   83 ec 0c                sub    $0xc,%esp   
  4d:   ff 75 f8                pushl  0xfffffff8(%ebp)   
  50:   ff 75 fc                pushl  0xfffffffc(%ebp)   
  53:   e8 fc ff ff ff          call   54   
  58:   83 c4 14                add    $0x14,%esp   
  ......

如结果所示，在离开 .text section 开始 0x53 字节的地方，有一条call指令。这条指令是对 sum\_func 函数的调用，objdump 将其反汇编成 call 54，这是因为偏移 0x54 字节的地方原本应该放着 sum\_func 函数的地址，但现在因为 sum\_func 定义在 sum.o 中，所以这个地方就是重定位需要做 patch 的地址单元所在处。我们注意到，这个地址单元的值为 0xfffffffc，也就是十进制的 -4(计算机中数是用补码表示的)。所以，参与重定位运算的变量A就确定了，即是 -4。

我们在 test.o 中找出影响该地址单元的重定位记录如下：

[yihect@juliantec test\_2]$ readelf -r ./test.o |  grep 54   
00000054  00000a02 R\_386\_PC32        00000000   sum\_func

果然，如你所见，该条重定位记录是 R\_386\_PC32 类型的。前面变量A确定了，那么另外两个变量S和变量P呢？从正向去计算这两个变量的值比较麻烦。尽管我们知道，在Linux里面，链接可执行程序时所使用的默认的链接器脚本将最后可执行程序的 .text segment 起始地址设置在 0x08048000的位置。但是，从这个地址出发，去寻找符号(函数)sub\_func 和 上面受重定位影响的地址单元的运行时地址的话，需要经过很多人工计算，所以比较麻烦。

相反的，我们使用objdump工具像下面这样分析最终链接生成的可执行程序 ./test 的 .text segment 段，看看函数 sum\_func 和 那个受影响单元的运行时地址到底是多少，这是反向的查看链接器的链接结果。链接器在链接的过程中是正向的将正确的地址分配给它们的。

[yihect@juliantec test\_2]$ objdump -d -j .text ./test   
    
./test:     file format elf32-i386   
    
Disassembly of section .text:   
    
08048498 :   
8048498:       31 ed                   xor    %ebp,%ebp   
......   
08048540 <main />:   
......   
804858a:       83 ec 0c                sub    $0xc,%esp   
804858d:       ff 75 f8                pushl  0xfffffff8(%ebp)   
8048590:       ff 75 fc                pushl  0xfffffffc(%ebp)   
8048593:       e8 74 00 00 00          call   804860c   
8048598:       83 c4 14                add    $0x14,%esp   
804859b:       50                      push   %eax   
......   
  
0804860c :   
804860c:       55                      push   %ebp   
804860d:       89 e5                   mov    %esp,%ebp   
804860f:       8b 45 0c                mov    0xc(%ebp),%eax   
8048612:       03 45 08                add    0x8(%ebp),%   
8048615:       c9                      leave    
8048616:       c3                      ret      
8048617:       90                      nop   
    
......

从中很容易的就可以看出，链接器给函数 sum\_func 分配的运行时地址是 0x0804860c，所以变量S的值就是 0x0804860c。那么变量P呢？它表示的是重定位所影响地址单元的运行地址。如果要计算这个地址，我们可以先看看 main 函数的运行时地址，再加上0x54字节的偏移来得到。从上面看出 main 函数的运行时地址为 0x08048540，所以重定位所影响地址单元的运行时地址为 0x08048540+0x54 = 0x08048594。所以重定位计算的最终结果为：

S+A-P = 0x0804860c+(-4)-0x08048594 = 0x00000074

从上面可以看出，链接器在链接过程中，确实也把这个计算得到的结果存储到了上面 call 指令操作数所在的地址单元中去了。那么，程序在运行时，是如何凭借这样一条带有如此操作数的 call 指令来调用到(或者跳转到)函数 sum\_func 中去的呢？

你看，调用者 main 和被调用者 sum\_func 处在同一个text segment中。根据x86架构或者IBM兼容机的汇编习惯，段内转移或者段内跳转时使用的寻址方式是PC相对寻址。也就是若要让程序从一个段内的A处，跳转到同一段内的B处，那么PC相对寻址会取程序在A处执行时的PC值，再加上某一个偏移值(offset)，得到要跳转的目标地址(B处地址)。那么，对于x86架构来说，由于有规定，PC总是指向下一条要执行的指令，那么当程序执行在call指令的时候，PC指向的是下一条add指令，其值也就是 0x8048598。最后，寻址的时候再加上call指令的操作数0x74作为偏移，计算最终的 sum\_func 函数目标地址为 0x8048598+0x74 = 0x804860c。

有点意思吧:)，如果能绕出来，那说明我们是真的明白了，其实，绕的过程本身就充满着趣味性，就看你自己的心态了。说到这里，本文行将结束。本文所介绍的很多内容，可能在某些同学眼中会过于简单，但是为了体现知识的完整性、同时也为了让大家先有个基础以便更容易的看后续的文章，我们还是在这里介绍一下ELF格式的基础知识。下面一篇关于ELF主题的文章，将详细介绍动态连接的内在实现。届时，你将看到大量的实际代码挖掘。