**ELF符号表结构（1）**

ELF文件中的符号表往往是文件中的一个段，段名一般叫".symtab"。符号表的结构很简单，它是一个Elf32\_Sym结构（32位ELF文件）的数组，每个Elf32\_Sym结构对应一个符号。这个数组的第一个元素，也就是下标0的元素为无效的"未定义"符号。Elf32\_Sym的结构定义如下：

|  |
| --- |
| typedef struct { Elf32\_Word st\_name; Elf32\_Addr st\_value; Elf32\_Word st\_size; unsigned char st\_info; unsigned char st\_other; Elf32\_Half st\_shndx; } Elf32\_Sym; |

这几个成员的定义如表3-14所示。

表3-14

|  |  |
| --- | --- |
| st\_name | 符号名。这个成员包含了该符号名在字符串  表中的下标（还记得字符串表吧？） |
| st\_value | 符号相对应的值。这个值跟符号有关，可能  是一个绝对值，也可能是一个地址等，不同的符号，  它所对应的值含义不同，见下文“符号值” |
| st\_size | 符号大小。对于包含数据的符号，这个值是该数据  类型的大小。比如一个double型的符号它占用8个字节。  如果该值为0，则表示该符号大小为0或未知 |
| st\_info | 符号类型和绑定信息，见下文“符号类型与绑定信息” |
| st\_other | 该成员目前为0，没用 |
| st\_shndx | 符号所在的段，见下文“符号所在段” |

符号类型和绑定信息（st\_info） 该成员低4位表示符号的类型（Symbol Type），高28位表示符号绑定信息（Symbol Binding），如表3-15、表3-16所示。  
  
表3-15

|  |  |  |
| --- | --- | --- |
| 符号绑定信息 | | |
| 宏定义名 | 值 | 说明 |
| STB\_LOCAL | 0 | 局部符号，对于目标文件的外部不可见 |
| STB\_GLOBAL | 1 | 全局符号，外部可见 |
| STB\_WEAK | 2 | 弱引用，详见“弱符号与强符号” |

表3-16

|  |  |  |
| --- | --- | --- |
| 符号类型 | | |
| 宏定义名 | 值 | 说明 |
| STT\_NOTYPE | 0 | 未知类型符号 |
| STT\_OBJECT | 1 | 该符号是个数据对象，比如变量、数组等 |
| STT\_FUNC | 2 | 该符号是个函数或其他可执行代码 |
| STT\_SECTION | 3 | 该符号表示一个段，这种符号必须是  STB\_LOCAL的 |
| STT\_FILE | 4 | 该符号表示文件名，一般都是该目标文  件所对应的源文件名，它一定是  STB\_LOCAL类型的，并且它的  st\_shndx一定是SHN\_ABS |

符号所在段（st\_shndx）如果符号定义在本目标文件中，那么这个成员表示符号所在的段在段表中的下标；但是如果符号不是定义在本目标文件中，或者对于有些特殊符号，sh\_shndx的值有些特殊，如表3-17所示。

表3-17

|  |  |  |
| --- | --- | --- |
| 符号所在段特殊常量 | | |
| 宏定义名 | 值 | 说明 |
| SHN\_ABS | 0xfff1 | 表示该符号包含了一个绝对的值。  比如表示文件名的符号就属于这种类型的 |
| SHN\_COMMON | 0xfff2 | 表示该符号是一个“COMMON块”  类型的符号，一般来说，未初始化的全局  符号定义就是这种类型的，比如  SimpleSection.o里面的global\_uninit\_var。  有关“COMMON”详见“深入静态链接”  之“COMMON块” |
| SHN\_UNDEF | 0 | 表示该符号未定义。这个符号表示该符号  在本目标文件被引用到，但是定义在其他目标文件中 |

符号值（st\_value） 我们前面已经介绍过，每个符号都有一个对应的值，如果这个符号是一个函数或变量的定义，那么符号的值就是这个函数或变量的地址，更准确地讲应该按下面这几种情况区别对待。

在目标文件中，如果是符号的定义并且该符号不是"COMMON块"类型的（即st\_shndx不为SHN\_COMMON，具体请参照"深入静态链接"一章中的"COMMON块"），则st\_value表示该符号在段中的偏移。即符号所对应的函数或变量位于由st\_shndx指定的段，偏移st\_value的位置。这也是目标文件中定义全局变量的符号的最常见情况，比如SimpleSection.o中的"func1"、"main"和"global\_init\_var"。

在目标文件中，如果符号是"COMMON块"类型的（即st\_shndx为SHN\_COMMON），则st\_value表示该符号的对齐属性。比如SimpleSection.o中的"global\_uninit\_var"。

在可执行文件中，st\_value表示符号的虚拟地址。这个虚拟地址对于动态链接器来说十分有用。我们将在第3部分讲述动态链接器。

根据上面的介绍，我们对ELF文件的符号表有了大致的了解，接着将以SimpleSection.o里面的符号为例子，分析各个符号在符号表中的状态。这里使用readelf工具来查看ELF文件的符号，虽然objdump工具也可以达到同样的目的，但是总体来看readelf的输出格式更为清晰：

|  |
| --- |
| $ readelf -s SimpleSection.o  Symbol table '.symtab' contains 15 entries: Num: Value Size Type Bind Vis Ndx Name 0: 00000000 0 NOTYPE LOCAL DEFAULT UND  1: 00000000 0 FILE LOCAL DEFAULT ABS SimpleSection.c 2: 00000000 0 SECTION LOCAL DEFAULT 1  3: 00000000 0 SECTION LOCAL DEFAULT 3  4: 00000000 0 SECTION LOCAL DEFAULT 4  5: 00000000 0 SECTION LOCAL DEFAULT 5  6: 00000000  4 OBJECT LOCAL DEFAULT 4 static\_var2.1534 7: 00000004  4 OBJECT LOCAL DEFAULT 3 static\_var.1533 8: 00000000 0 SECTION LOCAL DEFAULT 7  9: 00000000 0 SECTION LOCAL DEFAULT 6  10: 00000000  4 OBJECT GLOBAL DEFAULT 3 global\_init\_var 11: 00000000 27 FUNC GLOBAL DEFAULT 1 func1 12: 00000000 0 NOTYPE GLOBAL DEFAULT UND printf 13: 0000001b 64 FUNC GLOBAL DEFAULT 1 main 14: 00000004  4 OBJECT GLOBAL DEFAULT COM global\_uninit\_var |

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，第一列Num表示符号表数组的下标，从0开始，共15个符号；第二列Value就是符号值，即st\_value；第三列Size为符号大小，即st\_size；第四列和第五列分别为符号类型和绑定信息，即对应st\_info的低4位和高28位；第六列Vis目前在C/C++语言中未使用，我们可以暂时忽略它；第七列Ndx即st\_shndx，表示该符号所属的段；当然最后一列也最明显，即符号名称。从上面的输出可以看到，第一个符号，即下标为0的符号，永远是一个未定义的符号。对于另外几个符号解释如下。

func1和main函数都是定义在SimpleSection.c里面的，它们所在的位置都为代码段，所以Ndx为1，即SimpleSection.o里面，.text段的下标为1。这一点可以通过readelf -a或objdump -x得到验证。它们是函数，所以类型是STT\_FUNC；它们是全局可见的，所以是STB\_GLOBAL；Size表示函数指令所占的字节数；Value表示函数相对于代码段起始位置的偏移量。

再来看printf这个符号，该符号在SimpleSection.c里面被引用，但是没有被定义。所以它的Ndx是SHN\_UNDEF。

global\_init\_var是已初始化的全局变量，它被定义在.bss段，即下标为3。

global\_uninit\_var是未初始化的全局变量，它是一个SHN\_COMMON类型的符号，它本身并没有存在于BSS段；关于未初始化的全局变量具体请参见"COMMON块"。

static\_var.1533和static\_var2.1534是两个静态变量，它们的绑定属性是STB\_LOCAL，即只是编译单元内部可见。至于为什么它们的变量名从"static\_var"和"static\_var2"变成了现在这两个"static\_var.1533"和"static\_var2.1534"，我们在下面一节"符号修饰"中将会详细介绍。

对于那些STT\_SECTION类型的符号，它们表示下标为Ndx的段的段名。它们的符号名没有显示，其实它们的符号名即它们的段名。比如2号符号的Ndx为1，那么它即表示.text段的段名，该符号的符号名应该就是".text"。如果我们使用"objdump -t"就可以清楚地看到这些段名符号。

"SimpleSection.c"这个符号表示编译单元的源文件名。

第一部分： ELF格式概述

ELF(Executable and Linkable Format)是一种对可执行文件、目标文件以及库文件使用的文件格式，它在Linux下成为标准文件已经有很长的一段时间，代替了早期的a.out格式。ELF格式的一个优点是同一个文件格式可以用在Linux Kernel支持的所有体系结构之上。这不仅简化了用户空间工具程序的创建，也简化了内核自身的程序设计，比如必须为可执行程序生成装载例程时。但是文件格式相同并不意味着不同系统上的程序之间存在二进制兼容性，例如FreeBSD和Linux都使用ELF作为二进制格式，但是FreeBSD上的程序不能运行于Linux上，因为两者在系统调用机制和系统调用语义方面仍然有所不同，反之亦然；如果想让二者之间的程序能够运行，必须要有一个中间仿真层。同样PowerPC平台编译的ELF程序，也不能在X86平台的机器上运行，反之依然，因为两者的体系结构是完全不同的。但是由于ELF格式的存在，相同体系结构上的ELF程序本身的相关信息，以及程序的各个部分在二进制文件中的编码方式都是相同的。Linux不仅将ELF用于用户空间应用程序和库，还用于工具模块，另外Linux内核本身也是ELF格式。

备注：ELF文件是一种开放的格式，其规范可以自由获得。

ELF文件有三种类型：

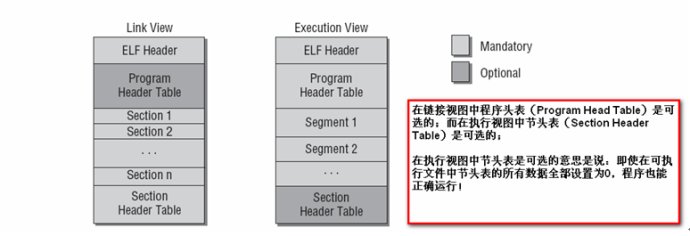
可重定位文件：也就是通常称的目标文件，后缀为.o；

共享文件：也就是通常称的库文件，后缀为.so；

可执行文件：本文主要讨论的文件格式；

总的来说，可执行文件的格式与上述两种文件的格式之间的区别主要在于观察的角度不同：一种称为连接视图（Linking View），一种称为执行视图（Execution View）。

示意图如下：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=70dd16910100zczk&url=http://s4.sinaimg.cn/orignal/70dd1691gbdcb576d3bd3)

第二部分：布局和结构

ELF Header：除了用于标识ELF文件的几个字节之外，ELF文件头还包含了有关文件类型和大小的信息，以及文件加载后程序执行的入口信息等等，总之ELF头部是一个关于本ELF文件的路线图（road map），从总体上描述文件的结构。

程序表头(Program Head Table)：向系统提供了可执行文件的数据在进程虚拟地址空间中组织方式的相关信息，它还表示了文件可能包含的段的数目，段的位置以及用途。

各个段SegmentX(X=1,2,…)和节SectionX(X=1,2,…)：保存了与文件相关的各种形式的数据，例如符号表、实际的二进制代码、固定值或者程序使用的数值常数。

节头表(Section Header Table)：包含了与各段相关的附加信息。

下面我们一个例子，用readelf工具来分析ELF文件：

例如如下：

//test.c

#include <stdio.h>

#include <stdlib.h>

int add(int a, int b)

{

printf("Numbers are added together\n");

return a + b;

}

int main()

{

int a, b;

a = 3;

b = 4;

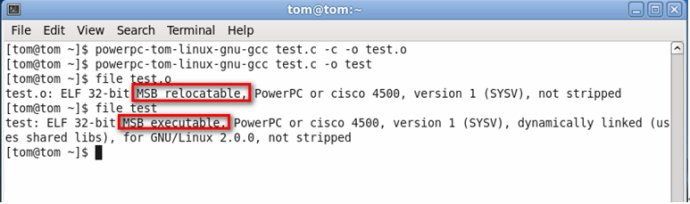
int ret = add(a,b);

printf("Result : %u\n",ret);

exit(0);

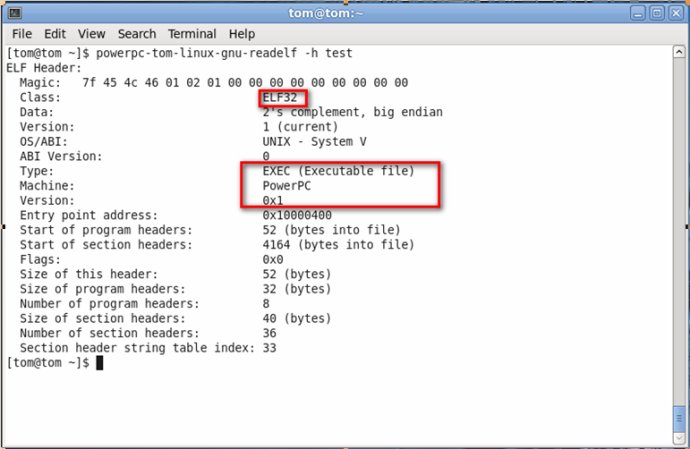
}

我们用file命令来显示编译器生成的两个文件的信息：一个是可执行文件，另一个是可重定位文件，示意图如下：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=70dd16910100zczk&url=http://s9.sinaimg.cn/orignal/70dd1691gbdcb5c206d28)

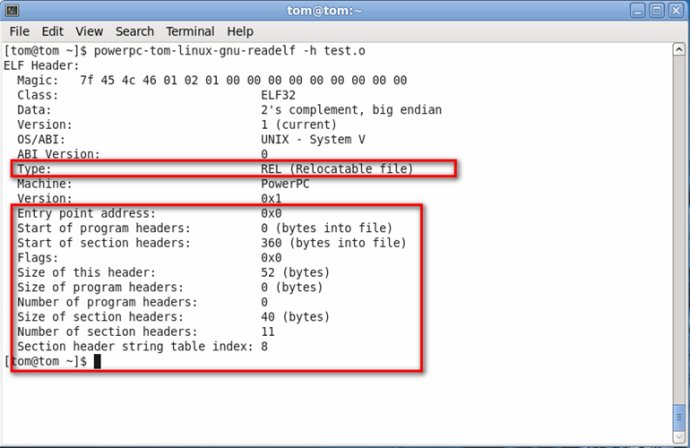
2.1 ELF header文件结构

我们用readelf工具来分析上例中生成ELF文件的ELF header，视图如下：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=70dd16910100zczk&url=http://s3.sinaimg.cn/orignal/70dd1691gbdcb5f1f54f2)

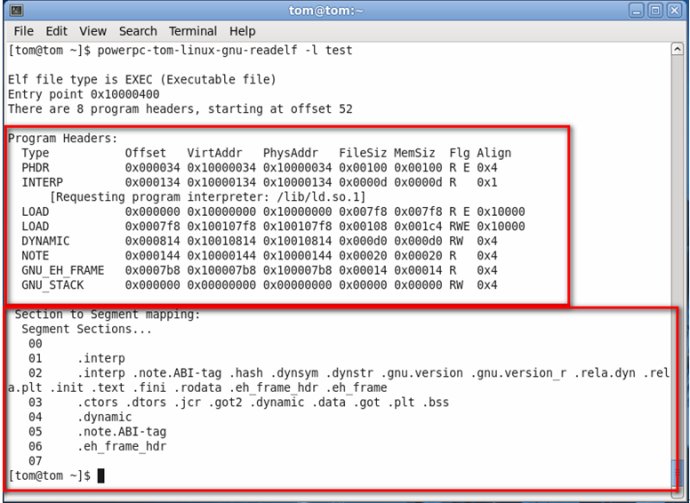
在test文件的开始处是四个标志字节，0x 7f、0x45、0x4c、0x46。其中的0x45、0x4c、0x46分别代码“E”“L”“F”的ASCII码，这使得所有处理ELF文件的工具都可以识别ELF类型的文件。还有一些与机器体系结构相关的信息。比如上图中Machine类型为PowerPC表明该ELF文件运行于PowerPC平台；ELF32表明这是一个运行在32位平台的机器；文件类型为EXEC表明这是一个可执行程序；Version用于区分当前ELF文件的各个修订版本，当前ELF文件是基于版本1；另外还包含ELF文件各个部分的长度和索引的位置，后面会相信讨论。

如果ELF文件是可重定位文件，不同的字段如下图所示：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=70dd16910100zczk&url=http://s2.sinaimg.cn/orignal/70dd1691gf69f1d40c251)  
  
如图所示：test.o的文件类型为REL，即它是一个可重定位的文件，其代码可以移动到任意位置，该文件没有程序头Program Headers,对于需要链接的对象而言该表是不需要的，所以其所以长度均为0。

2.2 程序头表(Program Head Table)分析

下面我们来分析一下ELF可执行文件test中Program Head Table，示意图如下：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=70dd16910100zczk&url=http://s7.sinaimg.cn/orignal/70dd1691gbdcb651ae926)

在Program Headers中列出了8个段，这些段组成了最终在内存中执行的程序，并且还提供了各个段在虚拟地址空间和物理地址空间中的位置、大小、访问授权和其它方面的信息。从上图中我们可以看出，示例程序中包含的各个段的语义如下：

PHDR：保存Program Headers Table

INTERP：制定程序已经从可执行文件映射到内存之后，必须调用的解释器。在这里，解释器并不意味着二进制文件的内容必须由另一个程序解释(比如Java字节代码必须由Java虚拟机来解释)，它指的是这样的程序：通过链接其它库来满足未解决的引用。

LOAD：表示一个需要从二进制文件映射到虚拟地址空间的段，其中保存常量数据（如字符串），程序的目标代码等等。

DYNAMIC：保存了由动态链接区(即INTERP中指定的解释器)使用的信息。

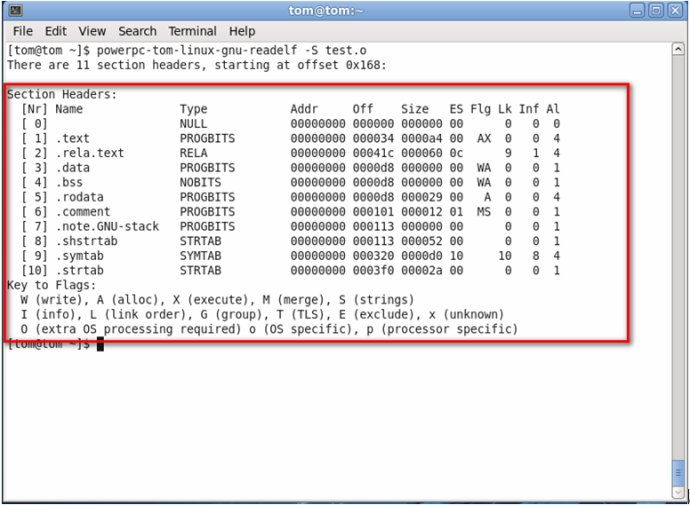
NOTE：保存了专用信息，与当前主题无关。

GNU\_EH\_FRAM和GNU\_STACK:用来分析栈帧，与体系结构相关，在PowerPC体系结构中需要分析栈帧实现回溯。

备注：我们需要特别指出段是可以重叠的，比如在上图中LOAD段从物理地址0x1000 0000到物理地址0x1000 0000+0x007f8的范围，该范围包含了PHDR、INTERP、NOTE、GNU\_EH\_FRAM和GNU\_STACK段，在ELF标准中是允许这种行为的。

2.3 节头标(Section Header Table)分析

在ELF文件中描述各段的内容时，是指定了哪些节的数据映射到段中。在ELF中是有一张节头表来管理文件中的各个节，readelf同样可以用于显示可重定位文件test.o中的各个节，示意图如下：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=70dd16910100zczk&url=http://s12.sinaimg.cn/orignal/70dd1691gbdcb689988eb)

这里指定的偏移量0x168是相对于二进制文件的。节信息不需要复制到在虚拟地址空间中做为可执行文件创建的最终进程映象，尽管如此在ELF二进制文件中节信息总是存在的。

每个节都指定了一个类型，定义了节数据的语义。包含PROGBITS、SYSTAB、RELA和STRTAB。

PROGBITS：程序必须解释的信息，比如二进制代码，程序的二进制代码被称之为text，指的是用作机器代码的二进制信息

SYSTAB：符号表

RELA：重定位信息

STRTAB：用于存储与ELF相关的字符串，但与程序没有直接的关系。

各个节Section都指定了其大小和在二进制文件内部的偏移量。

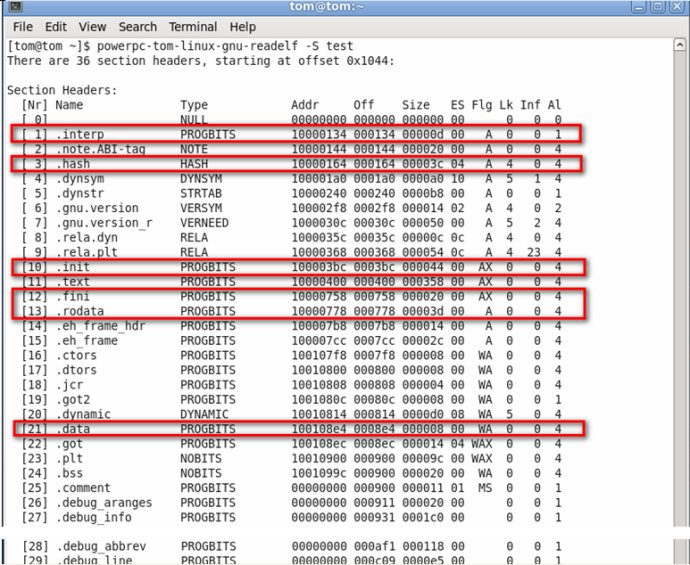
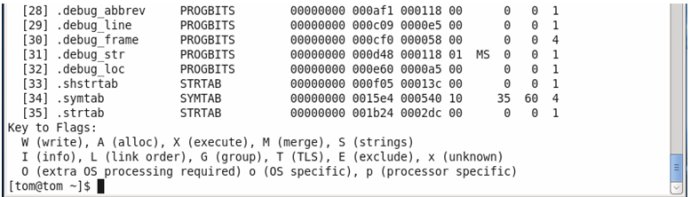
地址Addr：指定加载到虚拟地址空间的位置，因为我们的例子中处理的是一个可链接的对象，目标地址是没有定义的，因而表示为0。

标志Flg：指出各个节Section如何被访问或者处理。我们对标志A比较感兴趣，因为它控制着装载文件时是否将节的数据复制到虚拟地址空间中。

尽管节的名称是可以自由选择的，但是Linux(和其它所有使用ELF的类UNIX系统)都提供了一些标准节，其中一些是强制性的，例如总是有一个名为.text的节来保存二进制代码（即与该ELF文件相关联的程序信息），.rel.text保存了节的重定位信息。

备注：节名以点开始是由系统自身使用的，如果应用程序要想定义自身的节，就不应该以点开头，以避免与系统节名相冲突。

可执行程序包含了一些重定位的信息，示意图如下：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=70dd16910100zczk&url=http://s15.sinaimg.cn/orignal/70dd1691gf69f241b869e)  
[](http://photo.blog.sina.com.cn/showpic.html#blogid=70dd16910100zczk&url=http://s2.sinaimg.cn/orignal/70dd1691gbdcb6f8b1451)

与可重定位文件的11个节相比，可执行文件有36个节，并非所有的节都与我们的讨论有关。上图中标出的节是有具体意义的：

.interp:保存了解释器的文件名，例如在我们的例子中用的是是ld.so.1；

.data:保存了初始化的数据，这是普通程序数据的一部分，可以在程序运行时修改；

.rodata:保存了只读数据，可以读但不可以修改，例如编译器将所有出现在printf中的静态字符串封装到该节；

.init和.fini：保存了进程初始化和结束时所用的代码，这两个节通常是编译器自动添加的，无需应用程序员关注；

.hash：是一个散列表，允许在不对全表元素进行线性搜索的情况下，快速访问所有的符号表项。

备注：可执行文件ELF各个Section中的Addr字段保存了有效地址的值，因为相应的代码必须映射到虚拟地址空间中某些已经定义好的位置，在基于PowerPC的Linux中应用程序通常使用0x1000 0000以上的内存区间。

2.4 符号表(Symbol Table)

符号表是每个ELF文件的一个重要的组成部分，因为它保存了程序实现或者使用过程中所有的（全局）变量和函数，如果程序使用了一个自身代码没有定义的符号，则称之为未定义符号(例如例子中的printf函数是定义在C标准库中，自身没有定义)，此类应用必须在静态链接期间用其它的目标模块或者库来解决，或者在加载期间使用lib/ ld.so.1来解决。我们可以使用nm工具生成程序定义或者使用的所有符号列表，如果下图所示：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=70dd16910100zczk&url=http://s9.sinaimg.cn/orignal/70dd1691g7961250a7598)

左侧一列给出了符号的值，即符号定义在目标文件中的位置，例子中包含了两个不同的符号类型：如果函数定义在text段，缩写为T；而未定义的引用用U标明。逻辑上没有定义的函数没有符号值。

可在可执行ELF文件test中，还会有更多的符号。当时大多数都是由编译器自动生成的，供运行时系统内部使用，以下例子中我们仅标出了同时出现在test.o中的符号：

U exit@@GLIBC\_2.0

1000058c T add

100005dc T main

U puts@@GLIBC\_2.0

解释：exit和puts虽然是没有定义的，但是同时增加了一些版本信息，标明能够提供函数的GNU标准库的最低版本，在我们的例子中要求库的最低版本不能低于2.0，这意味着该程序无法使用Libc5和Lib4工作，因为Libc4和Libc5是Linux专用的C标准库，Glibc\_2.0是该库的第一个跨平台版本，它替换了就版本，它替换了旧版本。

由函数本身定义的add和main已经移到虚拟地址空间中的固定位置(在文件加载时，对应的代码将会映射到这些位置)

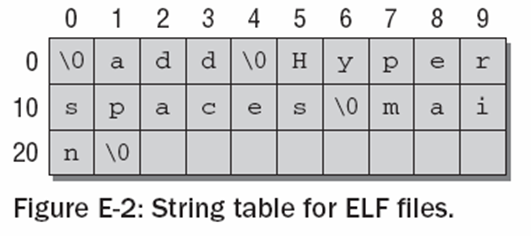
ELF使用以下的三个节（Section）；来实现字符串的管理：

.systalb：确定字符串的名称与其值的索引，但符号的名称不是以字符串的形式出现的，而是表示为某个字符串数组的索引；

.strtab：保存了字符串数组；

.hash：保存了一个hash表用于快速查找符号。

简而言之，符号名在字符串表(string table )中的位置和符号的值，为了说明字符串表示如何管理ELF中的字符串，我们看下面的例子：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=70dd16910100zczk&url=http://s15.sinaimg.cn/orignal/70dd1691gbdcb7551432e)

表的第一个字节是NULL(即ASCII码值为0)，后续的各个字符串通过NULL字节分割，为了引用字符串必须指定一个位置，即字符串的索引。这将选择下一个NULL字节之前的所有字符（如果用NULL字节的位置作为索引，那么将会对应空串）。如果允许索引不仅仅选择字符串的起始地址，也可以选择字符串中间的任何位置，就能够支持字串的用法(但是非常受限)

上述的.strtab节并不是默认情况下的唯一的字符串表，.shstrtab用于存放文件中各个节的文本名称(比如.text)

第三部分 Linux 内核中的数据结构

内核在两处使用了ELF文件格式：ELF用于处理可执行文件和库，用于实现模块。

这些地方使用了不同的代码来读取和操作数据，但这两种情况下都利用了我们本文所介绍的数据结构，其基础是include/elf.h文件，它实现了ELF标准，基本没有做改动！

3.1 数据类型

ELF是一个与CPU和体系结构无关的格式，它不能依赖与特定的字长和字节序（大端还是小端），至少对文件中的那些需要在所有系统上读取和理解的数据元素来说是这样。出现在.text段中的机器代码存储为宿主系统的表示格式，以避免转换工作。为此Linux内核定义了一些数据类型，在所有体系结构上具有相同的位宽，如下所示：

//32-bit ELF 基本类型

typedef \_\_u32 Elf32\_Addr;

typedef \_\_u16 Elf32\_Half;

typedef \_\_u32 Elf32\_Off;

typedef \_\_s32 Elf32\_Sword;

typedef \_\_u32 Elf32\_Word;

// 64-bit ELF 基本类型

typedef \_\_u64 Elf64\_Addr;

typedef \_\_u16 Elf64\_Half;

typedef \_\_s16 Elf64\_SHalf;

typedef \_\_u64 Elf64\_Off;

typedef \_\_s32 Elf64\_Sword;

typedef \_\_u32 Elf64\_Word;

typedef \_\_u64 Elf64\_Xword;

typedef \_\_s64 Elf64\_Sxword;

因为体系结构相关的代码必须总是明确定义整数类型的符号和位宽，ELF标准的数据类型可以毫不费力的通过typedef实现

3.2 ELF头部格式实现

对ELF格式的各种头部文件，32位和64位需要分别定义其数据结构：

32位的ELF头

在include/elf.h定义如下：

#define EI\_NIDENT 16

typedef struct elf32\_hdr{

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;

解释：

e\_ident：可以容纳16个字节，这些字节在所有的体系结构上都是由char数据类型表示的，前4个字节分别为0x7f(即ASCII码中的DEL)和字母E、L、F，我们在2.1节曾介绍过。其它的字节位置有其特定的含义：

e\_ident[4]：EI\_CLASS标识文件的类型，将文件分为32位和64位两类，即ELFCLASS32和ELFCLASS64

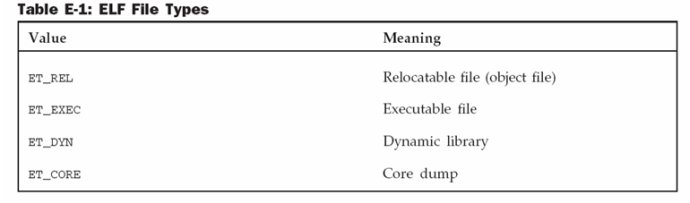
e\_ident[5]:EI\_DATA指定了格式所使用的字节序，即ELFDATA2LSB(小端)和ELFDATA2MSB()大端

ei\_ident[6]:EI\_VERSION指定ELF头文件的版本（该版本可能独立于数据段的版本），当前，值允许使用EV\_CURRENT，这是第一个版本

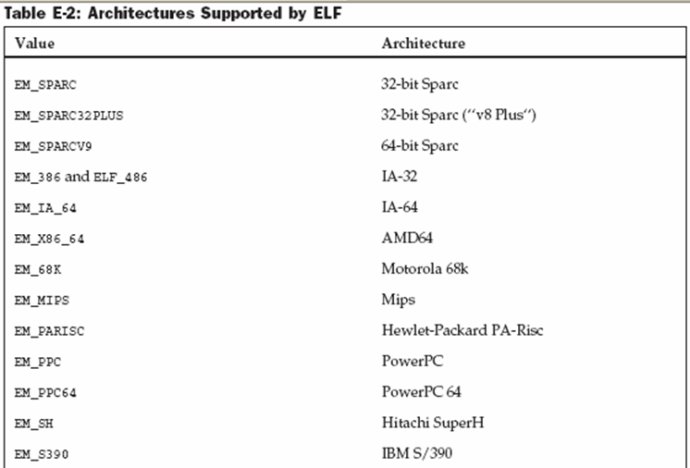
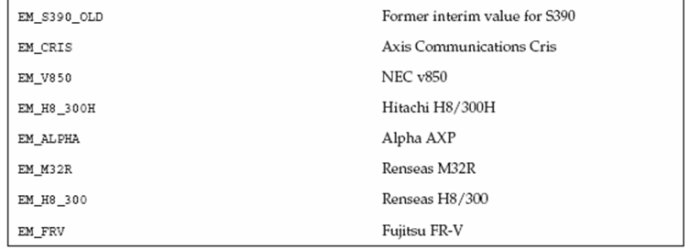
ei\_ident[7]: EI\_OSABI指定采用的OS的ABI版本，当前是UNIX System V ABI

从ei\_ident[8]EI\_PAD起的剩余字节：用NULL来填充，因为这些位置上ELF不需要。

e\_type用去区分各种ELF文件的类型，如下所示：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=70dd16910100zczk&url=http://s2.sinaimg.cn/orignal/70dd1691g796125a62ca1)

e\_machine指定了文件所需的体系结构，下表列出了Linux支持的各种选项：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=70dd16910100zczk&url=http://s10.sinaimg.cn/orignal/70dd1691gbdcb7cc7e279)  
[](http://photo.blog.sina.com.cn/showpic.html#blogid=70dd16910100zczk&url=http://s15.sinaimg.cn/orignal/70dd1691gbdcb7f1f8c6e)

注意：每种体系结构都需要定义elf\_check\_arch，并由内核的通用代码使用，来确保加载的ELF文件可以在相应的体系结构上运行。

e\_version：保存了版本信息，用于区分不同的ELF变体，目前该规范仅支持版本1，由EV\_CURRENT指出来

e\_entry：给出了文件在虚拟内存中的入口点，在程序已经加载并映射到内存之后，执行开始的位置

e\_phoff：保存了程序头表（Program Header Table）在二进制ELF文件中的偏移

e\_shoff：保存了节表头(Section Header Table) 在二进制ELF文件中的偏移

e\_flags：保存了特定于处理器的标志，当前Linux内核不适用该标志e\_ehsize：指定了ELF Header的长度，单位是字节

e\_phentsize：指定了Program Header中一个表项的长度，单位是字节（所有表项的长度均相同）

e\_phnum：指定了Program Header Table中表项的数目

e\_shentsize：指定了Section Header Table中一个表项的长度，单位是字节（所有表项的长度均相同）

e\_shnum：指定了节头表中项的数目

e\_shstrndx：保存了包含各节(Section)名称的字符串表在Section Header中的索引位置

备注：64位下ELF头的数据结构可以同样定义，唯一的差别在于其中使用了对应的64位数据类型，这使得头文件稍大。但两者数据结构的前16字节都是相同的，两种体系结构的类型能够根据这些字节来识别不同字长机器的ELF文件

3.3 Program Header实现

程序头表由几个项实现，其处理方式类似于数组项(项的数据由ELF头中的e\_phnum指定)，表项的数据类型定义为独立的结构，在32的机器上其内容如下：

typedef struct elf32\_phdr{

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;

解释：

p\_type：表示当前项描述的段的种类，为其定义了下列常数：

PT\_NULL：表示没有使用的段

PT\_LOAD：表示可装载段，在程序执行前从二进制文件映射到内存

PT\_DYNAMIC：表示段包含了用于动态链接器的信息

PT\_INTERP：表示当前段指定了用于动态链接的解释程序，比如ld.so.1

PT\_NOTE：指定一个段，其中可能含有专有的编译器信息

还有两个常数PT\_LOPROC和PT\_HIGHPROC用于定义与处理器相关的用途，内核并不使用

p\_offset：指出所描述段从ELF文件起始处的偏移量，以字节为单位

p\_vaddr：给出段的数据映射到虚拟地址空间中的位置(对应PT\_LOAD类型的段)，只支持物理寻找，不支持虚拟寻址的系统将使用p\_paddr保存的信息

p\_filesz：指定了段在ELF文件中的长度

p\_memsz：指定了段在虚拟地址空间中的长度，与文件中物理段的差值可以通过阶段数据或者填充0字节来填充

p\_flags：保存了标志信息，定义了段的访问权限：PF\_R读权限，PF\_W写权限，PF\_X执行权限

p\_align：指定了段在内存和二进制文件中的对齐方式（即p\_vaddr和p\_offset地址必须是模p\_align，即为p\_align的倍数）

比如p\_align的值为0x1000=4KB，这意味着段必须4KB对齐。

对64为体系结构定义的类似的数据结构，与32位相比，唯一的差别在于所使用的数据结构，各数据项的语义都是相同的。其具体内容如下：

typedef struct elf64\_phdr {

Elf64\_Word p\_type;

Elf64\_Word p\_flags;

Elf64\_Off p\_offset; //Segment file offset

Elf64\_Addr p\_vaddr; //Segment virtual address

Elf64\_Addr p\_paddr; //Segment physical address

Elf64\_Xword p\_filesz; //Segment size in file

Elf64\_Xword p\_memsz; //Segment size in memory

Elf64\_Xword p\_align; //Segment alignment, file & memory

} Elf64\_Phdr;

3.4 节头表Section Header Table代码实现

节头表通过数组实现，每个数组项包含一节的信息，各个节构成了程序头表Program Header Table定义的各个段的内容。下来数据结构表示一个节：

typedef struct elf32\_shdr {

Elf32\_Word sh\_name;

Elf32\_Word sh\_type;

Elf32\_Word sh\_flags;

Elf32\_Addr sh\_addr;

Elf32\_Off sh\_offset;

Elf32\_Word sh\_size;

Elf32\_Word sh\_link;

Elf32\_Word sh\_info;

Elf32\_Word sh\_addralign;

Elf32\_Word sh\_entsize;

} Elf32\_Shdr;

解释：

sh\_name：指定了节的名称，其值不是字符串本身，而是字符串表的一个索引。

sh\_type：指定了节的类型，有下列的类型可用：

SH\_NULL表示该节不使用，其数据将会被忽略

SH\_PROGBITS保存程序的相关信息，其格式是不定义的，与这里的讨论无关

SH\_SYMTAB保存一个符合表，其结构我们在2.4节已经讨论，SH\_DYNSYM也保存一个符号表，二者的差别我们在稍后讨论

SH\_STRTAB表示一个包含字符串表的节

SH\_RELA和SHT\_RELA保存重定位信息，我们也将在后面讨论

SH\_HASH定义了一个节保存HASH表，用于快速的查找符号表中的项

SH\_DYNAMIC保存了动态链接表的信息，我们也将在后面讨论

还有类型值SHT\_HIPROC、SHT\_LOPROC、SHT\_HIUSER、SHT\_LOUSER,这些专项特定于CPU和应用程序的用途，与这里讨论的内容无关

sh\_flags：表示节是否可重写（SHF\_WRITE）,是否将其分配虚拟内存(SHR\_ALLOC)，是否包含可执行的机器代码(SHF\_EXECINSTR)

sh\_addr：指定节映射到虚拟地址空间中的位置

sh\_offset：指定了节在内存中的位置

sh\_size：指定了节的长度

sh\_link：引用另一个节头表项，可以根据节类型进行不同的解释，其性能我们在后面单独讨论

sh\_info和sh\_link联用，其确切语义我们也会在下文中再讨论。

sh\_addralign：指定了节数据在内存中的对齐方式

sh\_entsize：指定了节中各个数据项的长度，前提是这些数据项的长度均相同。例如字符串表根据节类型不同，sh\_link和sh\_info的用法也不尽相同，具体情况如下：

第一：SHT\_DYNAMIC类型的节使用sh\_link指向节数据指向的字符串表，这种情况下不使用sh\_info,sh\_info设置为0；

第二：散列表(SHT\_HASH类型的节)使用sh\_link指向所散列的符合表，sh\_info不使用

第三：类型为SHT\_RELA和SHT\_REL的重定位节，使用sh\_link指向相关的符号表，sh\_info中保存的是节头表中的索引，表示对哪个节进行重定向；

第四：sh\_link指定了用作符号表的字符串表（SHT\_SYMTAB和SHT\_DYNSYM），而sh\_info表示符号表中紧随最后一个局部符号之后的索引位置(STB\_LOCAL类型)

而64位系统有一个单独的数据结构，其内容和32系统相同，除了使用64位的数据类型。内容如下：

typedef struct elf64\_shdr {

Elf64\_Word sh\_name; // Section name, index in string tbl

Elf64\_Word sh\_type; // Type of section

Elf64\_Xword sh\_flags; // Miscellaneous section attributes

Elf64\_Addr sh\_addr; // Section virtual addr at execution

Elf64\_Off sh\_offset; // Section file offset

Elf64\_Xword sh\_size; // Size of section in bytes

Elf64\_Word sh\_link; // Index of another section

Elf64\_Word sh\_info; // Additional section information

Elf64\_Xword sh\_addralign; // Section alignment

Elf64\_Xword sh\_entsize; // Entry size if section holds table

} Elf64\_Shdr;

ELF标准定义了若干固定名称的节，这些节用于执行大多数目标文件所需要的标准任务，所有名称都从点开始，以便与用户定义的节或者非标准的节相区分。最重要的标准节如下所示：

.bss：保存程序未初始化的数据，在程序开始时填充0字节

.data：保存已经初始化的数据，例如在编译期间用静态数据初始化的结构，这些数据可以在程序运行期间更改

.rodata：保存了程序使用的只读数据，不能更改

.dynamic和.dynstr：保存了动态信息，后面讨论

.interp：保存了程序解释器的名称，形式为字符串

.shstrtab：包含了一个字符串表，定义了节名称

.strtab：保存了一个字符串表，主要包含了符合表所需要的各个字符串

.symtab：保存了二进制文件的符合表

.init和.fini：保存了程序初始化和结束时执行的机器指令，这两个节的内容有编译器或者辅助工具自动创建，主要是为程序建立一个适当的运行环境

.text：保存了主要的机器指令

3.5 字符串表String Tables

字符串表的格式我们在2.4的结尾处讨论过，因为其格式非常动态，内核不同提供一个固定的数据结构，必需手工分析现存数据

3.6 符号表Symbol Tables

符号表保存了查找程序符号、为符号赋值、重定位符号所需要的全部信息，如上所述，有一个专门类型的节来保存符号表，其格式有下列数据结构定义：

typedef struct elf32\_sym{

Elf32\_Word st\_name;

Elf32\_Addr st\_value;

Elf32\_Word st\_size;

unsigned char st\_info;

unsigned char st\_other;

Elf32\_Half st\_shndx;

} Elf32\_Sym;

解释：

符号表的任务就是将一个字符串和一个值关联起来，例如printf符号表表示printf函数在虚拟空间中的地址，该函数的机器码就存在于该地址处。符号也可能有绝对值，有程序解释，例如数据常数。

一个符号的确切用途有st\_info定义，它分为两个部分(比特位如何划分与我们的讨论不相关)，其中定义了下列信息：

第一：符号的绑定(binding),这确定了符号的可见性，允许有下列三种不同的设置：

1：局部符号(STB\_LOCAL)：只有在目标文件内部可见，在于程序的其它部分连接时是不可见的。如果一个程序的几个目标文件都定义同名的此类符号是没有问题的，这些局部符号间彼此不会干扰。

2：全局符号(STB\_GLOBAL)：在定义的目标文件内部可见，也可以由构成程序的其它目标文件引用，每个全局符号在一个程序的内部只引用一次，否则链接器就会报错。

执行全局符号的位定义引用，将在重定位期间确定相关符号的位置，如果对全局符号的未定义引用无法解决，则拒绝程序执行或者静态绑定。

3：弱符号(STB\_WEAK)：在整个程序中可见，当可以由多个定义。如果程序中一个全局符号和一个局部符号名称相同，则全局符号就优先处理；即使一个弱符号没有定义，程序也是可以静态链接或者动态链接，这种情况下若符号的值为0

第二：符号类型也有若干备选项，只有以下三个与目标的主题相关（对其它值的描述由ELF标准提供）

STT\_OBJCT：符号关联到一二数据对象，比如变量，数组和指针；

STT\_FUNC：符号关联到一个函数或者过程；

STT\_NOTTYPE：符号的类型为制定，用于未定义引用；

Elf32\_Sym结构体除了包含st\_name,st\_value,st\_info，还包含以下成员，其语义如下：

st\_size:制定对象的长度，例如一个指针的长度或者struct对象中包含的字节数，如果长度有位置，其值可以设置为0；

标准的ELF标准不支持st\_other

st\_shndx：保存一个节(在节头表)中的索引，符号将绑定到该节，该符号通常定义在此节的代码中，但下列两个值具有特殊的语义：

SHN\_ABS制定符号的绝对值，不因重定位而改变

SHN\_UNDER标准未定义符号，必须通过外部来源(比如链接目标文件或者库)来解决。

同样，符号表也有一个64位的变体，除了使用的数据类型不同，其内容与32位的对应结构是相同的，如下所示；

typedef struct elf64\_sym {

Elf64\_Word st\_name; // Symbol name, index in string tbl

unsigned char st\_info; // Type and binding attributes

unsigned char st\_other; // No defined meaning, 0

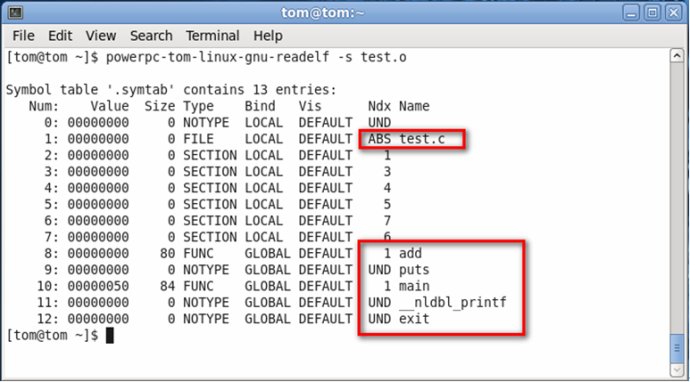
Elf64\_Half st\_shndx; // Associated section index

Elf64\_Addr st\_value; // Value of the symbol

Elf64\_Xword st\_size; // Associated symbol size

} Elf64\_Sym;

我们可以使用readelf来查找程序的符号表中所有的符号，图中标出的六项在test.o目标文件中特别重要，其它的数据项由编译器自动生成的，与我们的讨论无关O(∩\_∩)O~

[](http://photo.blog.sina.com.cn/showpic.html#blogid=70dd16910100zczk&url=http://s4.sinaimg.cn/orignal/70dd1691gbdcb854ffb83)

分析：源文件中名称”test.c”存储为一个绝对值，它是常数不随着重定位而改变。该局部符号使用SET\_FILE类型，将一个目标文件关联到对应的源文件、

文件中定义了两个函数main和add,存储为SET\_FUNC类型的全家符号，两个符号都执行节1，即ELF文件中的.text节，保存了两个函数的机器代码。

\_\_nldbl\_printf、puts、exit符号属于未定义引用，节索引值为UND，因而在程序链接时它们必须关联到标准库中的函数，或者其它库中以该名称定义的函数。因为编译器不指定所涉及的符号的类型，因而这两个符号的类型都是STT\_NOTYPE。

**第四部分重定位项Relocation Entries**

重定位是将ELF文件中为定义符号关联到有效值的处理过程，在我们的例子test.o中，这意味着对printf、exit、puts的未定义引用必须替换为该进程的虚拟地址空间中相应的机器代码所在的空间，在目标文件中用到的符号之处都将被替换。

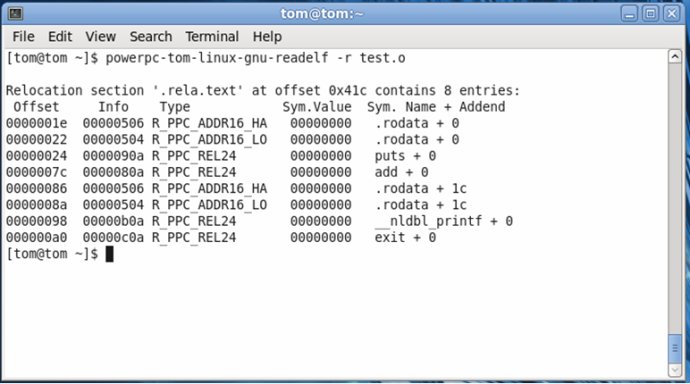
对用户空间程序的替换，内核并不会牵涉其中。因为所有的替换操作都是由外面工具完成的。当对于Linux内核模块来说，情况有所不同，因为内核所接受到得模块裸数据与存储在二进制文件中的数据完全相同，内核负责重定位操作。

在每个目标文件中都有一个专门的表，包含了重定位表项，标识了需要进行重定位的地方。每个表项都包含了下列信息：

第一：一个偏移量，用来指定所要修改的项的位置

第二：对符号的引用（符号表的索引），提供了所要插入的重定位位置的数据

为了说明如何使用重定位信息，我们来看一下此前test.c测试程序，我们用readelf显示的所有重定位项如下图所示：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=70dd16910100zd1i&url=http://s5.sinaimg.cn/orignal/70dd1691gbdcc3bc379b4)

在程序运行时或者test.o产生可执行文件时，如果某些机器代码使用了虚拟地址空间中位置尚不明确的符号或者函数，则会使用offset列的信息。main函数的汇编语言代码调用了若干函数，分别位于偏移量0x 7c,0x98,0xa0如下所示：

00000050 <main>:

50: 94 21 ff e0 stwu r1,-32(r1)

54: 7c 08 02 a6 mflr r0

58: 90 01 00 24 stw r0,36(r1)

5c: 93 e1 00 1c stw r31,28(r1)

60: 7c 3f 0b 78 mr r31,r1

64: 38 00 00 03 li r0,3

68: 90 1f 00 08 stw r0,8(r31)

6c: 38 00 00 04 li r0,4

70: 90 1f 00 0c stw r0,12(r31)

74: 80 7f 00 08 lwz r3,8(r31)

78: 80 9f 00 0c lwz r4,12(r31)

7c: 48 00 00 01 bl 7c <main+0x2c>

80: 90 7f 00 10 stw r3,16(r31)

84: 3c 00 00 00 lis r0,0

88: 30 00 00 1c addic r0,r0,28

8c: 7c 03 03 78 mr r3,r0

90: 80 9f 00 10 lwz r4,16(r31)

94: 4c c6 31 82 crclr 4\*cr1+eq

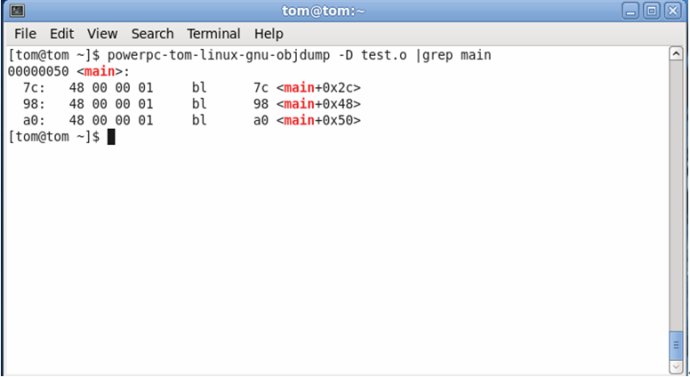
98: 48 00 00 01 bl 98 <main+0x48>

9c: 38 60 00 00 li r3,0

a0: 48 00 00 01 bl a0 <main+0x50>

备注：

我们可以用objdump工具查看，示意图如下：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=70dd16910100zd1i&url=http://s8.sinaimg.cn/orignal/70dd1691gbdcc478c1877)

在printf和add函数的地址已经确定后，必须将它们插入指定的偏移量处，以便可以生成正确运行的可执行代码。

**4.1 Linux内核中相关的数据结构**

由于技术原因，有两种类型的重定位信息，有两种稍有不同的数据结构表示：

第一种：普通重定位，SHT\_REL类型的节中的重定位表项由以下的数据结构表示：

typedef struct elf32\_rel {

Elf32\_Addr r\_offset;

Elf32\_Word r\_info;

} Elf32\_Rel;

解释：

r\_offset:指定需要重定位的项的位置

r\_info:不仅提供了符合表中的一个位置，还包括重定位类型的有关信息。这是通过把值划分为两个部分来达到的。

第二种：需要添加常数的重定位项只出现在SHT\_RELA类型的节中，这类结构由下列数据结构定义：

typedef struct elf32\_rela{

Elf32\_Addr r\_offset;

Elf32\_Word r\_info;

Elf32\_Sword r\_addend;

} Elf32\_Rela;

解释：

这里除了有第一种重定位类型提供的r\_offset和r\_info字段之外，还补充了r\_addend字段，其中存放一个称之为加数(addend)的值。在计算重定位值时，将根据重定位类型，对该值进行不同的处理。

请注意：

在使用elf32\_rel时也会出现加数这个值，尽管在数据结构中没有明确的保存，但链接器根据该值应该在内存中出现的位置，将计算出的重定位长度作为加数填入，该值的用途将在下面的例子中说明。

对两种重定位类型，都有功能等效的64位数据结构：

typedef struct elf64\_rel {

Elf64\_Addr r\_offset; //Location at which to apply the action

Elf64\_Xword r\_info; //index and type of relocation

} Elf64\_Rel;

typedef struct elf64\_rela {

Elf64\_Addr r\_offset; //Location at which to apply the action

Elf64\_Xword r\_info; //index and type of relocation

Elf64\_Sxword r\_addend; //Constant addend used to compute value

} Elf64\_Rela;

解释：这两个数据结构和32位的对应类型非常相似，这里就不在讨论

**4.2** **重定位类型**

ELF标准定义了很多重定位类型，对于每种支持的体系结构，都有一个独立的集合，这些类型大部分用于生成动态库或者与装载位置无关的代码。Linux内核只对模块的重定位感兴趣，因此使用下面的两种重定位类型：相对重定位和绝对重定位。

我们通过一个例子介绍一下相对从定位：

1000058c <add>:

1000058c: 94 21 ff e0 stwu r1,-32(r1)

100005d8: 4e 80 00 20 blr

100005dc <main>:

100005dc: 94 21 ff e0 stwu r1,-32(r1)

...........................

10000608: 4b ff ff 85 bl 1000058c <add>

..........................

10000624: 48 01 03 35 bl 10010958 <\_\_nldbl\_printf@plt>

10000628: 38 60 00 00 li r3,0

1000062c: 48 01 03 45 bl 10010970 <exit@plt>

Disassembly of section .plt:

10010900 <\_\_register\_frame\_info@plt-0x48>:

...

10010948 <\_\_register\_frame\_info@plt>:

...

10010950 <\_\_deregister\_frame\_info@plt>:

...

10010958 <\_\_nldbl\_printf@plt>:

...

10010960 <puts@plt>:

...

10010968 <\_\_gmon\_start\_\_@plt>:

...

10010970 <exit@plt>:

...

解释：

10000608: 4b ff ff 85 bl 1000058c <add>

我们首先介绍一下bl指令的格式：

0-5 6-29 30 31

OPCD LI AA LK

AA=0，表示LI中存放的是相对地址LI\*4，基址是当前指令的地址

AA=1，表示LI中存放的是绝对地址LI\*4

LK=1，表示转移到目的地址的同时，将当前指令的下一条指令存入LR寄存器

LK=0，仅仅表示跳转到目的地址，而不用修改LR寄存器

当前bl指令的机器码是0x4b ff ff 85

所以4\*LI的值为：0b11 1111 1111 1111 1111 1000 0100符号扩展为0xffff ff84对应的真值为：-124(十进制)=-0x7c

因为当前指令的地址为0x1000 0608

所以跳转的目标地址为0x1000 0608+（-0x7c）= 1000058c，即为add函数的入口地址1000058c

定义指令10000624: 48 01 03 35 bl 10010958 <\_\_nldbl\_printf@plt>

同理可以得到4\*LI=0x10334

所以跳转到得目的地址就是0x1000 0624+0x10334=0x1001 0958，即为.plt表中\_\_nldbl\_printf的入口地址。

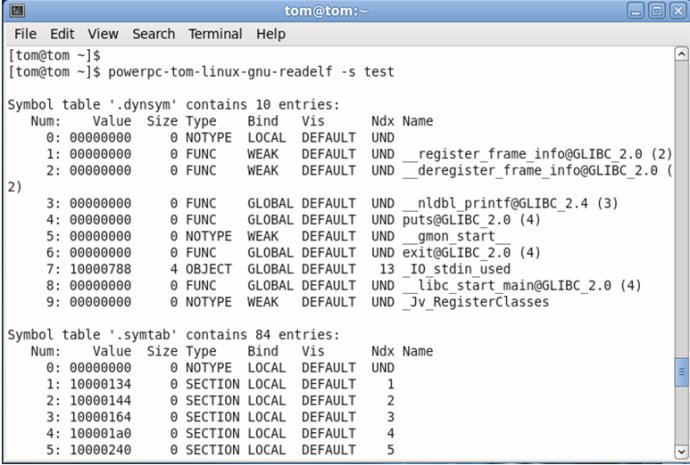
**4.3** **动态链接**

内核对必须与库动态链接产生的ELF文件不感兴趣，模块中的所有引用都可以通过重定位解决，而用户空间程序的动态链接则完全由用户空间中的ld.so进行。以下两个节用来保存动态链接器所需要的数据。

.dynsym保存了有关符号表，包含了所有需要外部引用解决的符号

.dynamic保存了一个数组，数组项为Elf32\_Dyn类型，这些选项提供了以下几个段落所描述的数据。

dynamic的内容可以使用readelf查询，示意图如下：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=70dd16910100zd1i&url=http://s1.sinaimg.cn/orignal/70dd1691g79613b613d50)

输出的内容不仅包含若干生成可执行文件时自动添加的符号，还包括机器代码使用的\_\_nldbl\_printf和exit函数，@GLIBC\_2.0指出至少使用GNU标准库中的2.0版本才能解决这些引用。

dynamic节中数组项的数据类型在内核中定义如下：

typedef struct dynamic{

Elf32\_Sword d\_tag;

union{

Elf32\_Sword d\_val;

Elf32\_Addr d\_ptr;

} d\_un;

} Elf32\_Dyn;

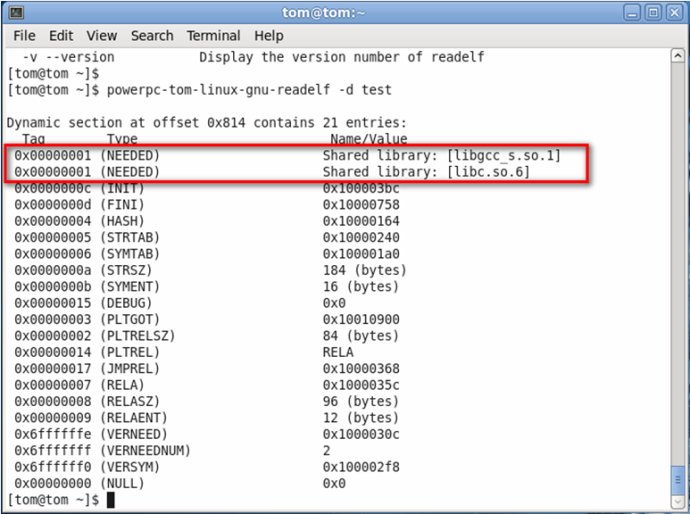
解释：

d\_tag用于区分各种指定信息类型的标记，该结构中的共用体根据该标志进行解释：

d\_un或者保存一个虚拟地址，或者保存一个整数，可以根据特定的标志进行解释。

最重要的标记如下所示：

DT\_NEEDED指定该执行程序所需要的一个动态库，d\_un指向一个字符串表项，给出了库的名称，对于test.c测试程序来说只需要C标准库，如下图所示：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=70dd16910100zd1i&url=http://s16.sinaimg.cn/orignal/70dd1691gbdcc55a5c9cf)

DT\_STRTAB保存了字符串表的位置，其中包括了dynamic节所需要的所有的动态库和符号的名称。

DT\_SYMTAB保存了符号表的位置，其中包含了dynamic节所需要的所有的信息

DT\_INIT和DT\_FINI保存了用于初始化和结束程序的地址。