序言  
1. OBJECT文件  
   导言  
   ELF头(ELF Header)  
   Sections  
   String表(String Table)  
   Symbol表(Symbol Table)  
   重定位(Relocation)  
2. 程序装载与动态连接  
   导言  
   Program头(Program Header)  
   Program装载(Program Loading)  
   Dynamic连接(Dynamic Linking)  
3. C LIBRARY  
   C Library  
  
   \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
  
          导言  
  
   \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
  
  ELF: 可执行连接格式  
  
可执行连接格式是UNIX系统实验室(USL)作为应用程序二进制接口  
(Application Binary Interface(ABI)而开发和发布的。工具接口标准委  
员会(TIS)选择了正在发展中的ELF标准作为工作在32位INTEL体系上不同操  
作系统之间可移植的二进制文件格式。  
  
假定开发者定义了一个二进制接口集合，ELF标准用它来支持流线型的软件  
发展。 应该减少不同执行接口的数量。因此可以减少重新编程重新编译的  
代码。  
  
  
关于这片文档  
  
这篇文档是为那些想创建目标文件或者在不同的操作系统上执行文件的开发  
着准备的。它分以下三个部分：  
  
\* 第一部分, “目标文件Object Files”描述了ELF目标文件格式三种主要  
  的类型。  
\* 第二部分, “程序转载和动态连接”描述了目标文件的信息和系统在创建  
  运行时程序的行为。  
\* 第三部分, “C 语言库”列出了所有包含在libsys中的符号,标准的ANSI C  
  和libc的运行程序，还有libc运行程序所需的全局的数据符号。  
  
注意: 参考的X86体系已经被改成了Intel体系。  
  
   \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
  
   1. 目标文件(Object file)  
  
   \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
  
   ========================= 序言 =========================  
  
  
第一部分描述了iABI的object文件的格式, 被称为ELF(Executable  
and Linking Format). 在object文件中有三种主要的类型。  
  
\* 一个可重定位(relocatable)文件保存着代码和适当的数据，用来和其他的  
  object文件一起来创建一个可执行文件或者是一个共享文件。  
\* 一个可执行(executable)文件保存着一个用来执行的程序；该文件指出了  
  exec(BA\_OS)如何来创建程序进程映象。  
\* 一个共享object文件保存着代码和合适的数据，用来被下面的两个链接器  
  链接。第一个是连接编辑器[请参看ld(SD\_CMD)]，可以和其他的可重定位和  
  共享object文件来创建其他的object。第二个是动态链接器，联合一个  
  可执行文件和其他的共享object文件来创建一个进程映象。  
  
一个object文件被汇编器和联接器创建, 想要在处理机上直接运行的object  
文件都是以二进制来存放的。那些需要抽象机制的程序，比如象shell脚本，  
是不被接受的。  
  
在介绍性的材料过后，第一部分主要围绕着文件的格式和关于如何建立程序。  
第二部分也描述了object文件的几个组成部分，集中在执行程序所必须的信息上。  
  
  
     文件格式  
  
Object文件参与程序的联接(创建一个程序)和程序的执行(运行一个程序)。  
object 文件格式提供了一个方便有效的方法并行的视角看待文件的内容，  
在他们的活动中，反映出不同的需要。例 1-1图显示了一个object文件的  
组织图。  
  
+ 图1-1: Object文件格式  
  
  Linking 视角                      Execution 视角  
  ============                      ==============  
  ELF header                        ELF header  
  Program header table (optional)   Program header table  
  Section 1                         Segment 1  
  ...                               Segment 2  
  Section n                         ...  
  Section header table              Section header table (optional)  
  
一个ELF头在文件的开始，保存了路线图(road map)，描述了该文件的组织情况。  
sections保存着object 文件的信息，从连接角度看：包括指令，数据，  
符号表，重定位信息等等。特别sections的描述会出项在以后的第一部分。  
第二部分讨论了段和从程序的执行角度看文件。  
  
假如一个程序头表（program header table）存在，那么它告诉系统如何来创建一  
个进程的内存映象。被用来建立进程映象(执行一个程序)的文件必须要有一个程  
序头表（program header table）；可重定位文件不需要这个头表。一个  
section头表（section header table）包含了描述文件sections的信息。每个  
section在这个表中有一个入口；每个入口给出了该section的名字，大小，  
等等信息。在联接过程中的文件必须有一个section头表；其他object文件可要  
可不要这个section头表。  
  
注意: 虽然图显示出程序头表立刻出现在一个ELF头后，section头表跟着其他  
section部分出现，事实是的文件是可以不同的。此外，sections和段(segments)  
没有特别的顺序。只有ELF头（elf header）是在文件的固定位置。  
  
  
数据表示  
object文件格式支持8位、32位不同的处理器。不过，它试图努力的在更大  
或更小的体系上运行。因此，object文件描绘一些控制数据需要用与机器  
无关的格式，使它尽可能的用一般的方法甄别object文件和描述他们的内容。  
在object文件中剩余的数据使用目标处理器的编码方式，不管文件是在哪台  
机子上创建的。  
  
+ 图 1-2: 32-Bit Data Types  
  
  Name           Size Alignment   Purpose  
  ====           ==== =========   =======  
  Elf32\_Addr      4       4       Unsigned program address  
  Elf32\_Half      2       2       Unsigned medium integer  
  Elf32\_Off       4       4       Unsigned file offset  
  Elf32\_Sword     4       4       Signed large integer  
  Elf32\_Word      4       4       Unsigned large integer  
  unsigned char   1       1       Unsigned small integer  
  
所有的object文件格式定义的数据结构是自然大小(natural size),为相关  
的类型调整指针。如果需要，数据结构中明确的包含了确保4字节对齐的填  
充字段。来使结构大小是4的倍数。数据从文件的开始也有适当的对齐。  
例如，一个包含了Elf32\_Addr成员的结构将会在文件内对齐到4字节的边界上。  
  
因为移植性的原因，ELF不使用位字段。  
  
  
   ========================== ELF Header ==========================  
  
  
一些object文件的控制结构能够增长的，所以ELF头包含了他们目前的大小。  
假如object文件格式改变，程序可能会碰到或大或小他们不希望的控制结构。  
程序也有可能忽略额外（extra）的信息。  
对待来历不明(missing)的信息依靠上下文来解释，假如扩展被定义，它们  
将会被指定。  
  
+ 图 1-3: ELF Header  
  
  #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;  
  
\* e\_ident  
  
  这个最初的字段标示了该文件为一个object文件，提供了一个机器无关  
  的数据，解释文件的内容。在下面的ELF的鉴别（ELF Identification）  
  部分有更详细的信息。  
  
\* e\_type  
  
  该成员确定该object的类型。  
  
               Name        Value  Meaning  
               ====        =====  =======  
               ET\_NONE         0  No file type  
               ET\_REL          1  Relocatable file  
               ET\_EXEC         2  Executable file  
               ET\_DYN          3  Shared object file  
       ET\_CORE         4  Core file  
       ET\_LOPROC  0xff00  Processor-specific  
       ET\_HIPROC  0xffff  Processor-specific  
  
  虽然CORE的文件内容未被指明，类型ET\_CORE是保留的。  
  值从 ET\_LOPROC 到 ET\_HIPROC(包括ET\_HIPROC)是为特殊的处理器保留的。  
  如有需要，其他保留的变量将用在新的object文件类型上。  
  
\* e\_machine  
  
  该成员变量指出了运行该程序需要的体系结构。  
  
                    Name      Value  Meaning  
            ====      =====  =======  
                    EM\_NONE       0  No machine  
    EM\_M32        1  AT&T WE 32100  
                    EM\_SPARC      2  SPARC  
                    EM\_386        3  Intel 80386  
                    EM\_68K        4  Motorola 68000  
                    EM\_88K        5  Motorola 88000  
                    EM\_860        7  Intel 80860  
                    EM\_MIPS       8  MIPS RS3000  
  
  如有需要，其他保留的值将用到新的机器类型上。特殊处理器名使用机器名来  
  区别他们。例如，下面将要被提到的成员flags使用前缀EF\_;在一台EM\_XYZ机器  
  上，flag称为WIDGET，那么就称为EF\_XYZ\_WIDGET。  
  
\* e\_version  
  
  这个成员确定object文件的版本。  
  
                 Name         Value  Meaning  
                 ====         =====  =======  
                 EV\_NONE          0  Invalid version  
EV\_CURRENT       1  Current version  
  
  值1表示原来的文件格式；创建新版本就用>1的数。EV\_CURRENT值(上面给  
  出为1)如果需要将指向当前的版本号。  
  
\* e\_entry  
  
  该成员是系统第一个传输控制的虚拟地址，在那启动进程。假如文件没有  
  如何关联的入口点，该成员就保持为0。  
  
\* e\_phoff  
  
  该成员保持着程序头表（program header table）在文件中的偏移量(以字节计数)。  
  假如该文件没有程序头表的的话，该成员就保持为0。  
  
\* e\_shoff  
  
  该成员保持着section头表（section header table）在文件中的偏移量(以字节  
  计数)。假如该文件没有section头表的的话，该成员就保持为0。  
  
\* e\_flags  
  
  该成员保存着相关文件的特定处理器标志。  
  flag的名字来自于EF\_<machine>\_<flag>。看下机器信息“Machine Information”  
  部分的flag的定义。  
  
\* e\_ehsize  
  
  该成员保存着ELF头大小(以字节计数)。  
  
\* e\_phentsize  
  
  该成员保存着在文件的程序头表（program header table）中一个入口的大小  
  (以字节计数)。所有的入口都是同样的大小。  
  
\* e\_phnum  
  
  该成员保存着在程序头表中入口的个数。因此，e\_phentsize和e\_phnum  
  的乘机就是表的大小(以字节计数).假如没有程序头表（program header table），  
  e\_phnum变量为0。  
  
\* e\_shentsize  
  
  该成员保存着section头的大小(以字节计数)。一个section头是在section  
  头表(section header table)的一个入口；所有的入口都是同样的大小。  
  
\* e\_shnum  
  
  该成员保存着在section header table中的入口数目。因此，e\_shentsize和  
  e\_shnum的乘积就是section头表的大小(以字节计数)。  
  假如文件没有section头表，e\_shnum值为0。  
  
\* e\_shstrndx  
  
  该成员保存着跟section名字字符表相关入口的section头表(section header   
  table)索引。假如文件中没有section名字字符表，该变量值为SHN\_UNDEF。  
  更详细的信息 看下面“Sections”和字符串表(“String Table”) 。  
  
  
  ELF 鉴别(Identification)  
  
在上面提到的，ELF提供了一个object文件的框架结构来支持多种处理机，多  
样的数据编码方式，多种机器类型。为了支持这个object文件家族，最初的几  
个字节指定用来说明如何解释该文件，独立于处理器，与文件剩下的内容无关。  
  
ELF头(也就是object文件)最初的几个字节与成员e\_ident相一致。  
  
+ 图 1-4: e\_ident[] Identification Indexes  
  
  Name           Value  Purpose  
  ====           =====  =======  
  EI\_MAG0      0  File identification  
  EI\_MAG1      1  File identification  
  EI\_MAG2      2  File identification  
  EI\_MAG3      3  File identification  
  EI\_CLASS      4  File class  
  EI\_DATA      5  Data encoding  
  EI\_VERSION      6  File version  
  EI\_PAD      7  Start of padding bytes  
  EI\_NIDENT     16  Size of e\_ident[]  
  
通过索引访问字节，以下的变量被定义。  
  
\* EI\_MAG0 to EI\_MAG3  
  
  文件的前4个字符保存着一个魔术数(magic number)，用来确定该文件是否  
  为ELF的目标文件。  
  
                  Name       Value  Position  
                  ====       =====  ========  
  ELFMAG0    0x7f   e\_ident[EI\_MAG0]  
                  ELFMAG1    'E'    e\_ident[EI\_MAG1]  
                  ELFMAG2    'L'    e\_ident[EI\_MAG2]  
                  ELFMAG3    'F'    e\_ident[EI\_MAG3]  
  
\* EI\_CLASS  
  
  接下来的字节是e\_ident[EI\_CLASS]，用来确定文件的类型或者说是能力。  
  
                 Name           Value  Meaning  
                 ====           =====  =======  
                 ELFCLASSNONE       0  Invalid class  
                 ELFCLASS32         1  32-bit objects  
ELFCLASS64         2  64-bit objects  
  
  文件格式被设计成在不同大小机器中可移植的，在小型机上的不用大型机上  
  的尺寸。类型ELFCLASS32支持虚拟地址空间最大可达4GB的机器；使用上面  
  定义过的基本类型。  
  
  类型ELFCLASS64为64位体系的机器保留。它的出现表明了object文件可能   
  改变，但是64位的格式还没有被定义。如果需要，其他类型将被定义，会  
  有不同的类型和不同大小的数据尺寸。  
  
\* EI\_DATA  
  
  字节e\_ident[EI\_DATA]指定了在object文件中特定处理器数据的编码  
  方式。当前定义了以下编码方式。  
  
             Name           Value  Meaning  
             ====           =====  =======  
     ELFDATANONE        0  Invalid data encoding  
             ELFDATA2LSB        1  See below  
             ELFDATA2MSB        2  See below  
  
  更多的关于编码的信息出现在下面。其他值保留，将被分配一个新的编码  
  方式，当然如果必要的话。  
  
\* EI\_VERSION  
  
  字节e\_ident[EI\_VERSION]表明了ELF头的版本号。  
  现在这个变量的是一定要设为EV\_CURRENT，作为上面e\_version的解释。  
  
\* EI\_PAD  
  
  该变量标识了在e\_ident中开始的未使用的字节。那些字节保留并被设置为  
  0；程序把它们从object 文件中读出但应该忽略。假如当前未被使用的字节  
  有了新的定义，EI\_PAD变量将来会被改变。  
  
一个文件的数据编码指出了如何来解释一个基本的object文件。在上述的  
描述中，类型ELFCLAS32文件使用占用1，2和4字节的目标文件。下面定义的  
编码方式，用下面的图来描绘。数据出现在左上角。  
  
  
ELFDATA2LSB编码指定了2的补数值。  
最小有意义的字节占有最低的地址。  
  
+ 图1-5: Data Encoding ELFDATA2LSB  
  
               0------+  
      0x0102   |  01  |  
               +------+  
               0------1------+  
    0x010204   |  02  |  01  |  
               +------+------+  
               0------1------2------3------+  
  0x01020304   |  04  |  03  |  02  |  01  |  
               +------+------+------+------+  
  
ELFDATA2LSB编码指定了2的补数值。  
最大有意义的字节占有最低的地址。  
  
+ 图1-6: Data Encoding ELFDATA2MSB  
  
               0------+  
      0x0102   |  01  |  
               +------+  
               0------1------+  
    0x010204   |  01  |  02  |  
               +------+------+  
               0------1------2------3------+  
  0x01020304   |  01  |  02  |  03  |  04  |  
               +------+------+------+------+  
  
  
机器信息  
  
为了确定文件，32位Intel体系结构的需要以下的变量。  
  
+ 图1-7: 32-bit Intel Architecture Identification, e\_ident   
  
  Position           Value  
  ========           =====  
  e\_ident[EI\_CLASS]  ELFCLASS32  
  e\_ident[EI\_DATA]   ELFDATA2LSB  
  
处理器确认ELF头里的e\_machine成员，该成员必须为EM\_386。  
  
ELF报头里的e\_flags成员保存了和文件相关的位标记。32位Intel体系上未  
定义该标记；所以这个成员应该为0；  
  
  
   =========================== Sections ===========================  
  
  
一个object文件的section header table可以让我们定位所有的sections。  
section header table是个Elf32\_Shdr结构的数组(下面描述)。一个section  
报头表(section header table)索引是这个数组的下标。ELF header table  
的e\_shoff成员给出了section报头表的偏移量(从文件开始的计数)。e\_shnum  
告诉我们section报头表中包含了多少个入口；e\_shentsize 给出了每个  
入口的大小。  
  
一些section报头表索引是保留的；那些特别的索引在一个object文件中  
将没有与之对应sections。  
  
+ 图1-8: Special Section Indexes  
  
  Name             Value  
  ====             =====  
  SHN\_UNDEF            0  
  SHN\_LORESERVE   0xff00  
  SHN\_LOPROC      0xff00  
  SHN\_HIPROC      0xff1f  
  SHN\_ABS         0xfff1  
  SHN\_COMMON      0xfff2  
  SHN\_HIRESERVE   0xffff  
  
\* SHN\_UNDEF  
  
  该值表明没有定义，缺少，不相关的或者其他涉及到的无意义的section。  
  例如，标号“defined”相对于section索引号SHN\_UNDEF是一个没有被  
  定义的标号。  
  
注意: 虽然索引0保留作为未定义的值，section报头表包含了一个索引0的  
入口。因此，假如ELF报头说一个文件的section报头表中有6个section入口  
的话，e\_shnum的值应该是从0到5。最初的入口的内容以后在这个section中  
被指定。  
  
\* SHN\_LORESERVE  
  
  该值指定保留的索引范围的最小值。  
  
\* SHN\_LOPROC through SHN\_HIPROC  
  
  该值包含了特定处理器语意的保留范围。  
  
  
\* SHN\_ABS  
  
  该变量是相对于相应参考的绝对地址。  
  例如，section号的标号是绝对地址，不被重定位影响。  
  
\* SHN\_COMMON  
  
  该section的标号是一个公共(common)的标号，就象FORTRAN COMMON  
  或者不允许的C扩展变量。  
  
\* SHN\_HIRESERVE  
  
  该值指定保留的索引范围的上限。系统保留的索引值是从SHN\_LORESERVE  
  到SHN\_HIRESERVE；该变量不涉及到section报头表（section header table）。  
  因此，section报头表不为保留的索引值包含入口。  
  
sections包含了在一个object文件中的所有信息，除了ELF报头，程序报头  
表(program header table)，和section报头表(section header table)。  
此外，object文件的sections满足几天条件：  
  
\* 每个在object文件中的section都有自己的一个section的报头来描述它。  
  section头可能存在但section可以不存在。  
\* 每个section在文件中都占有一个连续顺序的空间(但可能是空的)。  
\* 文件中的Sections不可能重复。文件中没有一个字节既在这个section中  
  又在另外的一个section中。  
\* object文件可以有"非活动的"空间。不同的报头和sections可以不覆盖到  
  object文件中的每个字节。"非活动"数据内容是未指定的。  
  
一个section头有如下的结构。  
  
+ 图1-9: Section Header  
  
  typedef struct {  
      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;  
  
\* sh\_name  
  
  该成员指定了这个section的名字。它的值是section报头字符表section的  
  索引。[看以下的“String Table”], 以NULL空字符结束。  
  
\* sh\_type  
  
  该成员把sections按内容和意义分类。section的类型和他们的描述在下面。  
  
\* sh\_flags  
  
  sections支持位的标记，用来描述多个属性。    
  Flag定义出现在下面。  
  
\* sh\_addr  
  
  假如该section将出现在进程的内存映象空间里，该成员给出了一个该section  
  在内存中的位置。否则，该变量为0。  
  
\* sh\_offset  
  
  该成员变量给出了该section的字节偏移量(从文件开始计数)。SHT\_NOBITS  
  类型的section(下面讨论)在文件中不占空间，它的sh\_offset成员定位在  
  文件中的概念上的位置。  
  
\* sh\_size  
  该成员给你了section的字节大小。除非这个section的类型为SHT\_NOBITS，  
  否则该section将在文件中将占有sh\_size个字节。SHT\_NOBITS类型的section  
  可能为非0的大小，但是不占文件空间。  
  
\* sh\_link  
  
  该成员保存了一个section报头表的索引连接，它的解释依靠该section的  
  类型。以下一个表描述了这些值。  
  
\* sh\_info  
  
  该成员保存着额外的信息，它的解释依靠该section的类型。以下一个表描  
  述了这些值。  
  
\* sh\_addralign  
  
  一些sections有地址对齐的约束。例如，假如一个section保存着双字，系统  
  就必须确定整个section是否双字对齐。所以sh\_addr的值以sh\_addralign的值  
  作为模，那么一定为0。当然的，仅仅0和正的2的次方是允许的。值0和1意味  
  着该section没有对齐要求。  
  
\* sh\_entsize  
  
  一些sections保存着一张固定大小入口的表，就象符号表。对于这样一个  
  section来说，该成员给出了每个入口的字节大小。如果该section没有  
  保存着一张固定大小入口的表，该成员就为0。  
  
section头成员sh\_type指出了section的语意。  
  
+ 图1-10: Section Types, sh\_type  
  
  Name               Value  
  ====               =====  
  SHT\_NULL               0  
  SHT\_PROGBITS           1  
  SHT\_SYMTAB             2  
  SHT\_STRTAB          3  
  SHT\_RELA          4  
  SHT\_HASH          5  
  SHT\_DYNAMIC            6  
  SHT\_NOTE          7  
  SHT\_NOBITS          8  
  SHT\_REL          9  
  SHT\_SHLIB             10  
  SHT\_DYNSYM            11  
  SHT\_LOPROC    0x70000000  
  SHT\_HIPROC    0x7fffffff  
  SHT\_LOUSER    0x80000000  
  SHT\_HIUSER    0xffffffff  
  
\* SHT\_NULL  
  
  该值表明该section头是无效的；它没有相关的section。  
  该section的其他成员的值都是未定义的。  
  
\* SHT\_PROGBITS  
  
  该section保存被程序定义了的一些信息，它的格式和意义取决于程序本身。  
  
\* SHT\_SYMTAB and SHT\_DYNSYM  
  
  那些sections保存着一个符号表（symbol table）。一般情况下，一个  
  object文件每个类型section仅有一个，但是，在将来，这个约束可能被放宽。  
  典型的是，SHT\_SYMTAB为连接器提供标号，当然它也有可能被动态连接时使用。  
  作为一个完整的符号表，它可能包含了一些动态连接时不需要的标号。  
  因此，一个object文件可能也包含了一个SHT\_DYNSYM的section，它保存着  
  一个动态连接时所需最小的标号集合来节省空间。  
  看下面符号表“Symbol Table”的细节。  
  
\* SHT\_STRTAB  
  
  该section保存着一个字符串表。一个object文件可以包含多个字符串表的  
  section。看下面字符串表“String Table”的细节。  
  
\* SHT\_RELA  
  
  该section保存着具有明确加数的重定位入口。就象object文件32位的  
  Elf32\_Rela类型。一个object文件可能有多个重定位的sections。具体细节  
  看重定位``Relocation''部分。  
  
\* SHT\_HASH  
  
  该标号保存着一个标号的哈希(hash)表。所有的参与动态连接的object  
  一定包含了一个标号哈希表（hash table）。当前的，一个object文件  
  可能只有一个哈希表。详细细节看第二部分的哈希表"Hash Table"。  
  
\* SHT\_DYNAMIC  
  
  该section保存着动态连接的信息。当前的，一个object可能只有一个动态  
  的section，但是，将来这个限制可能被取消。详细细节看第二部分的动态  
  section(“Dynamic Section”)。  
  
\* SHT\_NOTE  
  
  该section保存着其他的一些标志文件的信息。详细细节看第二部分的“Note   
  Section” 。  
  
\* SHT\_NOBITS  
  
  该类型的section在文件中不占空间，但是类似SHT\_PROGBITS。尽管该section  
  不包含字节，sh\_offset成员包含了概念上的文件偏移量。  
  
\* SHT\_REL  
  
  该section保存着具有明确加数的重定位的入口。  
  就象object文件32位类型Elf32\_Rel类型。一个object文件可能有多个  
  重定位的sections。具体细节看重定位``Relocation''部分。  
  
\* SHT\_SHLIB  
  
  该section类型保留但语意没有指明。包含这个类型的section的程序  
  是不符合ABI的。  
  
\* SHT\_LOPROC through SHT\_HIPROC  
  
  在这范围之间的值为特定处理器语意保留的。  
  
\* SHT\_LOUSER  
  
  该变量为应用程序保留的索引范围的最小边界。  
  
\* SHT\_HIUSER  
  
  该变量为应用程序保留的索引范围的最大边界。在SHT\_LOUSER和HIUSER的  
  section类型可能被应用程序使用，这和当前或者将来系统定义的section  
  类型是不矛盾的。  
  
其他section类型的变量是保留的。前面提到过，索引0(SHN\_UNDEF)的section  
头存在的，甚至索引标记的是未定义的section引用。这个入口保存着以下的  
信息。  
  
+ 图1-11: Section Header Table Entry: Index 0  
  
  Name            Value    Note  
  ====            =====    ====  
  sh\_name           0      No name  
  sh\_type        SHT\_NULL  Inactive  
  sh\_flags          0      No flags  
  sh\_addr           0      No address  
  sh\_offset         0      No file offset  
  sh\_size           0      No size  
  sh\_link SHN\_UNDEF  No link information  
  sh\_info     0      No auxiliary information  
  sh\_addralign     0      No alignment  
  sh\_entsize        0      No entries  
  
一个section报头（section header table）的sh\_flags成员保存着1位标记，  
用来描述section的属性。以下是定义的值；其他的值保留。  
  
+ 图1-12: Section Attribute Flags, sh\_flags  
  
  Name                Value  
  ====                =====  
  SHF\_WRITE             0x1  
  SHF\_ALLOC             0x2  
  SHF\_EXECINSTR         0x4  
  SHF\_MASKPROC   0xf0000000  
  
假如在sh\_flags中的一个标记位被设置，该section相应的属性也被打开。  
否则，该属性没有被应用。未明的属性就设为0。  
  
\* SHF\_WRITE  
  
  该section包含了在进程执行过程中可被写的数据。  
  
\* SHF\_ALLOC  
  
  该section在进程执行过程中占据着内存。一些控制section不存在一个  
  object文件的内存映象中；对于这些sections，这个属性应该关掉。  
  
\* SHF\_EXECINSTR  
  
  该section包含了可执行的机器指令。  
  
\* SHF\_MASKPROC  
  
  所有的包括在这掩码中的位为特定处理语意保留的。  
  
在section报头中，两个成员sh\_link和sh\_info的解释依靠该section的类型。  
  
+ 图1-13: sh\_link and sh\_info Interpretation  
  
  sh\_type      sh\_link                        sh\_info  
  =======      =======                        =======  
  SHT\_DYNAMIC  The section header index of    0  
               the string table used by  
               entries in the section.  
  SHT\_HASH     The section header index of    0  
               the symbol table to which the  
               hash table applies.  
  SHT\_REL,     The section header index of    The section header index of  
  SHT\_RELA     the associated symbol table.   the section to which the  
                                              relocation applies.  
  SHT\_SYMTAB,  The section header index of    One greater than the symbol  
  SHT\_DYNSYM   the associated string table.   table index of the last local  
                                              symbol (binding STB\_LOCAL).  
  other        SHN\_UNDEF                      0  
  
  
Special Sections   特殊的Sections  
  
不同的sections保存着程序和控制信息。下面列表中的section被系统使用，  
指示了类型和属性。  
  
+ 图1-14: Special Sections  
  
  Name         Type           Attributes  
  ====         ====           ==========  
  .bss         SHT\_NOBITS     SHF\_ALLOC+SHF\_WRITE  
  .comment     SHT\_PROGBITS   none  
  .data        SHT\_PROGBITS   SHF\_ALLOC+SHF\_WRITE  
  .data1       SHT\_PROGBITS   SHF\_ALLOC+SHF\_WRITE  
  .debug       SHT\_PROGBITS   none  
  .dynamic     SHT\_DYNAMIC    see below  
  .dynstr      SHT\_STRTAB     SHF\_ALLOC  
  .dynsym      SHT\_DYNSYM     SHF\_ALLOC  
  .fini        SHT\_PROGBITS   SHF\_ALLOC+SHF\_EXECINSTR  
  .got         SHT\_PROGBITS   see below  
  .hash        SHT\_HASH       SHF\_ALLOC  
  .init        SHT\_PROGBITS   SHF\_ALLOC+SHF\_EXECINSTR  
  .interp      SHT\_PROGBITS   see below  
  .line        SHT\_PROGBITS   none  
  .note        SHT\_NOTE       none  
  .plt         SHT\_PROGBITS   see below  
  .rel<name>   SHT\_REL        see below  
  .rela<name>  SHT\_RELA       see below  
  .rodata      SHT\_PROGBITS   SHF\_ALLOC  
  .rodata1     SHT\_PROGBITS   SHF\_ALLOC  
  .shstrtab    SHT\_STRTAB     none  
  .strtab      SHT\_STRTAB     see below  
  .symtab      SHT\_SYMTAB     see below  
  .text        SHT\_PROGBITS   SHF\_ALLOC+SHF\_EXECINSTR  
  
\* .bss  
  
  该sectiopn保存着未初始化的数据，这些数据存在于程序内存映象中。  
  通过定义，当程序开始运行，系统初始化那些数据为0。该section不占  
  文件空间，正如它的section类型SHT\_NOBITS指示的一样。  
    
\* .comment  
  
   该section保存着版本控制信息。  
  
\* .data and .data1  
  这些sections保存着初始化了的数据，那些数据存在于程序内存映象中。  
  
\* .debug  
  
  该section保存着为标号调试的信息。该内容是未指明的。  
  
\* .dynamic  
  
  该section保存着动态连接的信息。该section的属性将包括SHF\_ALLOC位。  
  是否需要SHF\_WRITE是跟处理器有关。第二部分有更详细的信息。  
  
\* .dynstr  
  
  该section保存着动态连接时需要的字符串，一般情况下，名字字符串关联着  
  符号表的入口。第二部分有更详细的信息。  
  
\* .dynsym  
  
  该section保存着动态符号表，如“Symbol Table”的描述。第二部分有更  
  详细的信息。  
    
\* .fini  
  
  该section保存着可执行指令，它构成了进程的终止代码。  
  因此，当一个程序正常退出时，系统安排执行这个section的中的代码。  
  
\* .got  
  
  该section保存着全局的偏移量表。看第一部分的“Special Sections”和  
  第二部分的“Global Offset Table”获得更多的信息。  
  
\* .hash  
  
  该section保存着一个标号的哈希表。看第二部分的“Hash Table”获得更多  
  的信息。  
  
\* .init  
  
  该section保存着可执行指令，它构成了进程的初始化代码。  
  因此，当一个程序开始运行时，在main函数被调用之前(c语言称为main)，  
  系统安排执行这个section的中的代码。  
  
\* .interp  
  
  该section保存了程序的解释程序(interpreter)的路径。假如在这个section  
  中有一个可装载的段，那么该section的属性的SHF\_ALLOC位将被设置；否则，  
  该位不会被设置。看第二部分获得更多的信息。   
  
\* .line  
  
  该section包含编辑字符的行数信息,它描述源程序与机器代码之间的对于  
  关系。该section内容不明确的。  
  
\* .note  
  
  该section保存一些信息，使用“Note Section”(在第二部分)中提到的格式。  
  
\* .plt  
  
  该section保存着过程连接表（Procedure Linkage Table）。看第一部分的  
  ``Special Sections''和第二部分的“Procedure Linkage Table”。  
  
\* .rel<name> and .rela<name>  
  
  这些section保存着重定位的信息，看下面的``Relocation''描述。  
  假如文件包含了一个可装载的段，并且这个段是重定位的，那么该section的  
  属性将设置SHF\_ALLOC位；否则该位被关闭。按照惯例，<name>由重定位适用  
  的section来提供。因此，一个重定位的section适用的是.text，那么该名字  
  就为.rel.text或者是.rela.text。  
  
\* .rodata and .rodata1  
  
  这些section保存着只读数据，在进程映象中构造不可写的段。看第二部分的  
  ``Program Header''获得更多的资料。  
  
\* .shstrtab  
  
  该section保存着section名称。  
  
\* .strtab  
  
  该section保存着字符串，一般地，描述名字的字符串和一个标号的入口相关  
  联。假如文件有一个可装载的段，并且该段包括了符号字符串表，那么section  
  的SHF\_ALLOC属性将被设置；否则不设置。  
  
\* .symtab  
  
  该section保存着一个符号表，正如在这个section里``Symbol Table''的  
  描述。假如文件有一个可装载的段，并且该段包含了符号表，那么section  
  的SHF\_ALLOC属性将被设置；否则不设置。  
  
\* .text  
  
  该section保存着程序的``text''或者说是可执行指令。  
  
  
前缀是点(.)的section名是系统保留的，尽管应用程序可以用那些保留的  
section名。应用程序可以使用不带前缀的名字以避免和系统的sections  
冲突。object文件格式可以让一个定义的section部分不出现在上面的列  
表中。一个object文件可以有多个同样名字的section。  
  
为处理器体系保留的section名的形成是通过置换成一个体系名的缩写。  
该名字应该取自体系名，e\_machine使用的就是。例如，.Foo.psect就是  
在FOO体系上定义的名字。  
  
现存的扩展名是历史遗留下来的。  
  
       Pre-existing Extensions  
       =======================  
.sdata     .tdesc  
.sbss      .lit4  
.lit8      .reginfo  
.gptab     .liblist  
.conflict  
  
  
   =================== String Table 字符串表=========================  
  
  
String table sections 保存着以NULL终止的一系列字符，一般我们称为字  
符串。object文件使用这些字符串来描绘符号和section名。一个字符串的  
参考是一个string table section的索引。第一个字节，即索引0，被定义保  
存着一个NULL字符。同样的，一个string table的最后一个字节保存着一个  
NULL字符，所有的字符串都是以NULL终止。索引0的字符串是没有名字或者说  
是NULL，它的解释依靠上下文。一个空的string table section是允许的；  
它的section header的成员sh\_size将为0。对空的string table来说，非0的  
索引是没有用的。  
  
一个 settion 头的 sh\_name 成员保存了一个对应于该 setion 头字符表部分  
的索引（就象ELF头的 e\_shstrndx 成员所特指的那样。下表列出了一个有 25 字节  
的字符串表（这些字符串和不同的索引相关联）：  
  
  
       Index   +0   +1   +2   +3   +4   +5   +6   +7   +8   +9  
       =====   ==   ==   ==   ==   ==   ==   ==   ==   ==   ==  
          0    /0   n    a    m    e    .    /0   V    a    r       
         10    i    a    b    l    e    /0   a    b    l    e  
         20    /0   /0   x    x    /0  
  
  
+ Figure 1-15: String Table Indexes  
  
  Index   String  
  =====   ======  
      0   none  
      1   "name."  
      7   "Variable"  
     11   "able"  
     16   "able"  
     24   null string  
  
  
如上所示，一个字符串表可能涉及该 section 中的任意字节。一个字符串可能   
引用不止一次；引用子串的情况是可能存在的；一个字符串也可能被引用若干次；而  
不被引用的字符串也是允许存在的。  
  
   ==================== Symbol Table 符号表=========================  
  
  
一个object文件的符号表保存了一个程序在定位和重定位时需要的定义和引用的信息。  
一个符号表索引是相应的下标。0表项特指了该表的第一个入口，就象未定义的符号  
索引一样。初始入口的内容在该 section 的后续部分被指定。  
  
                             Name       Value  
                             ====       =====  
     STN\_UNDEF      0  
  
一个符号表入口有如下的格式：  
  
+ Figure 1-16: Symbol Table Entry  
  
  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;  
  
\* st\_name  
  
  该成员保存了进入该object文件的符号字符串表入口的索引（保留了符号名的表达字符）。    
  如果该值不为 0 ，则它代表了给出符号名的字符串表索引。否则，该符号无名。  
  
注意：External C 符号和object文件的符号表有相同的名称。  
  
\* st\_value  
  
  该成员给出了相应的符号值。它可能是绝对值或地址等等（依赖于上下文）；  
  细节如下所述。  
  
\* st\_size  
  
  许多符号和大小相关。比如，一个数据对象的大小是该对象所包含的字节数目。  
  如果该符号的大小未知或没有大小则这个成员为 0 。  
  
\* st\_info  
  
  成员指出了符号的类型和相应的属性。相应的列表如下所示。下面的代码说明了  
  如何操作该值。   
  
    #define ELF32\_ST\_BIND(i) ((i)>>4)  
    #define ELF32\_ST\_TYPE(i) ((i)&0xf)  
    #define ELF32\_ST\_INFO(b, t) (((b)<<4)+((t)&0xf))  
  
\* st\_other  
  
  该成员目前为 0 ，没有含义。  
  
\* st\_shndx  
  
  每一个符号表的入口都定义为和某些 section 相关；该成员保存了相关的 section  
  头索引。就象 Figure 1-8 {\*}和相关的文字所描述的那样，某些 section 索引  
  指出了特殊的含义。  
  
一个符号的属性决定了可链接性能和行为。  
  
+ Figure 1-17: Symbol Binding, ELF32\_ST\_BIND  
  
  Name        Value  
  ====        =====  
  STB\_LOCAL       0  
  STB\_GLOBAL      1  
  STB\_WEAK        2  
  STB\_LOPROC     13  
  STB\_HIPROC     15  
  
\* STB\_LOCAL  
  
  在包含了其定义的object文件之外的局部符号是不可见的。不同文件中的具有相同  
  名称的局部符号并不相互妨碍。  
  
\* STB\_GLOBAL  
  
  全局符号是对所有的object目标文件可见的。一个文件中的全局符号的定义可以  
  满足另一个文件中对（该文件中）未定义的全局符号的引用。    
  
\* STB\_WEAK  
  
  弱符号相似于全局符号，但是他们定义的优先级比较低一些。  
  
\* STB\_LOPROC through STB\_HIPROC  
  
  其所包含范围中的值由相应的处理器语义所保留。  
  
全局符号和弱符号的区别主要在两个方面。  
  
\* 当链接器链接几个可重定位的目标文件时，它不允许 STB\_GLOBAL 符号的同名  
多重定义。另一方面，如果一个全局符号的定义存在则具有相同名称的弱符号名不会  
引起错误。链接器将认可全局符号的定义而忽略弱符号的定义。与此相似，如果有一个  
普通符号（比如，一个符号的 st\_shndx 域包含 SHN\_COMMON),则一个同名的弱符号  
不会引起错误。链接器同样认可普通符号的定义而忽略弱符号。  
  
\* 当链接器搜索档案库的时候，它选出包含了未定义的全局符号的存档成员。该成员  
的定义或者是全局的或者是一个弱符号。链接器不会为了解决一个未定义的弱符号  
选出存档成员。未定义的弱符号具有 0 值。  
  
在每一个符号表中，所有具有 STB\_LOCAL 约束的符号优先于弱符号和全局符号。  
就象上面 "sections" 中描述的那样，一个符号表部分的 sh\_info 头中的成员  
保留了第一个非局部符号的符号表索引。  
  
符号的类型提供了一个为相关入口的普遍分类。  
  
+ Figure 1-18: Symbol Types, ELF32\_ST\_TYPE  
  
  Name         Value  
  ====         =====  
  STT\_NOTYPE       0  
  STT\_OBJECT       1  
  STT\_FUNC         2  
  STT\_SECTION      3  
  STT\_FILE         4  
  STT\_LOPROC      13  
  STT\_HIPROC      15  
  
\* STT\_NOTYPE  
  
  该符号的类型没有指定。  
  
\* STT\_OBJECT  
  
  该符号和一个数据对象相关，比如一个变量、一个数组等。  
  
\* STT\_FUNC  
  
  该符号和一个函数或其他可执行代码相关。  
  
\* STT\_SECTION  
  
  该符号和一个 section 相关。这种类型的符号表入口主要是为了重定位，一般的  
  具有 STB\_LOCAL 约束。  
  
  
\* STT\_FILE  
  
  按惯例而言，该符号给出了和目标文件相关的源文件名称。一个具有 STB\_LOCAL   
  约束的文件符号，其 section 索引为 SHN\_ABS ，并且它优先于当前对应该文件的  
  其他 STB\_LOCAL 符号。  
  
\* STT\_LOPROC through STT\_HIPROC  
  
  该范围中的值是为处理器语义保留的。  
  
共享文件中的函数符号（具有 STT\_FUNC 类型）有特殊的意义。当其他的目标文件  
从一个共享文件中引用一个函数时，链接器自动的为引用符号创建一个链接表。除了  
STT\_FUNC 之外，共享的目标符号将不会自动的通过链接表引用。  
  
  
如果一个符号涉及到一个 section 的特定定位，则其 section 索引成员 st\_shndx  
将保留一个到该 section 头的索引。当该 section 在重定位过程中不断  
移动一样，符号的值也相应变化，而该符号的引用在程序中指向同样的定位。某些  
特殊的 section 索引有其他的语义。  
  
  
\* SHN\_ABS  
  
  该符号有一个不会随重定位变化的绝对值。  
  
\* SHN\_COMMON  
  
  该符号标识了一个没有被分配的普通块。该符号的值给出了相应的系统参数，就象  
  一个 section 的 sh\_addralign 成员。也就是说，链接器将分配一个地址给  
  该符号，地址的值是 st\_value 的倍数。该符号的大小指出了需要的字节数。  
    
  
\* SHN\_UNDEF  
  
  该 section 表索引表明该符号是未定义的。当链接器将该目标文件和另一个定义  
  该符号的文件相装配的时候，该文件内对该符号的引用将链接到当前实际的定义。  
  
如上所述，符号表的 0 索引（STN\_UNDEF）是保留的，它包含了如下内容：  
  
+ Figure 1-19: Symbol Table Entry: Index 0  
  
  Name        Value    Note  
  ====        =====    ====  
  st\_name       0      No name  
  st\_value      0      Zero value  
  st\_size       0      No size  
  st\_info       0      No type, local binding  
  st\_other      0  
  st\_shndx  SHN\_UNDEF  No section  
  
  
    Symbol Values（符号值）  
  
符号表入口对于不同的目标文件而言对 st\_value 成员有一些不同的解释。  
  
\* 在可重定位文件中， st\_value 保存了 section 索引为 SHN\_COMMON 符号  
的强制对齐值。  
  
\* 在可重定位文件中， st\_value 保存了一个符号的 section 偏移。也就是说，  
st\_value 是从 st\_shndx 定义的 section 开头的偏移量。  
  
\* 在可执行的和可共享的目标文件中， st\_value 保存了一个虚拟地址。为了使  
这些文件符号对于动态链接器更为有效，文件层面上的 section 偏移让位于内存  
层面上的虚拟地址（ section 编号无关的）。  
  
  
尽管符号表值对于不同的目标文件有相似的含义，相应的程序还是可以有效地访问数据。  
  
  
   ====================== Relocation （重定位）==========================  
  
  
重定位是连接符号引用和符号定义的过程。比如，当一个程序调用一个函数的时候，  
相关的调用必须在执行时把控制传送到正确的目标地址。换句话说，重定位文件应当  
包含有如何修改他们的 section 内容的信息，从而允许可执行文件或共享目标文件  
为一个进程的程序映像保存正确的信息。重定位入口就是这样的数据。  
  
+ Figure 1-20: Relocation Entries  
  
  typedef struct {  
      Elf32\_Addr r\_offset;  
      Elf32\_Word r\_info;  
  } Elf32\_Rel;  
  
  typedef struct {  
      Elf32\_Addr r\_offset;  
      Elf32\_Word r\_info;  
      Elf32\_Sword r\_addend;  
  } Elf32\_Rela;  
  
\* r\_offset  
  
  该成员给出了应用重定位行为的地址。对于一个重定位文件而言，该值是从该  
  section 开始处到受到重定位影响的存储单位的字节偏移量。对一个可执行文件  
  或一个共享目标而言，该值是受到重定位影响的存储单位的虚拟地址。  
  
  
\* r\_info  
  
  该成员给出了具有受重定位影响因素的符号表索引和重定位应用的类型。比如，  
  一个调用指令的重定位入口应当包含被调用函数的符号索引。如果该索引是   
  STN\_UNDEF （未定义的符号索引），重定位将使用 0 作为该符号的值。重定位  
  类型是和处理器相关的。当正文(text)引用到一个重定位入口的重定位类型或符  
  号表索引，它表明相应的应用 ELF32\_R\_TYPE或 ELF32\_R\_SYM 于入口的 r\_info   
  成员。  
  
    #define ELF32\_R\_SYM(i) ((i)>>8)  
    #define ELF32\_R\_TYPE(i) ((unsigned char)(i))  
    #define ELF32\_R\_INFO(s, t) ((s)<<8+(unsigned char)(t))  
  
\* r\_addend  
  
  该成员指定一个常量加数（用于计算将要存储于重定位域中的值）。  
  
  如上所述，只有 Elf32\_Rela 入口包含一个明确的加数。Elf32\_Rel 类型  
  的入口在可以修改的地址中存储一个隐含的加数。依赖于处理器结构，一种形式  
  或其他形式也许是必须的或更为方便的。因此，特定机器的应用应当使用一种排他  
  性的形式或依赖于上下文的另一种形式。  
  
  一个重定位 section 关联了两个其他的 section ：一个符号表和一个可修改  
  的 section 。该 section 头的成员 sh\_info 和 sh\_link （在上文中的  
  “ section ”部分中有描述）指示了这种关系。重定位入口中的成员 r\_offset  
  对于不同的目标文件有少许差异。  
  
\* 在可重定位文件中，r\_offset 表示了一个 section 偏移。也就是说，重定位  
  section自己描述了如何修改其他在文件中的其他section； 重定位偏移量指  
  明了一个在第二个section中的存储器单元。  
  
\* 在可执行和共享的目标文件中，r\_offset 表示一个虚拟地址。为了使得这些  
  文件的重定位入口更为有用（对于动态链接器而言），该 section 偏移（文件  
  中）应当让位于一个虚拟地址（内存中的）。  
  
尽管为了允许相关的程序更为有效的访问而让 r\_offset 的解释对于不同的目标  
文件有所不同，重定位类型的含义是相同的。  
  
  
   Relocation Types（重定位类型）  
  
重定位入口描述了怎样变更下面的指令和数据域（位数在表的两边角下）。  
  
+ Figure 1-21: Relocatable Fields   
  
    +---------------------------+  
    |          word32           |  
   31---------------------------0  
  
  
\* word32  
  
  指定一个以任意字节对齐方式占用 4 字节的 32 位域。这些值使用与 32 位 Intel  
  体系相同的字节顺序。  
  
                           3------2------1------0------+  
     0x01020304    |  01  |  02  |  03  |  04  |  
                          31------+------+------+------0  
  
下面的计算假设正在将一个可重定位文件转换为一个可执行或共享的目标文件。  
从概念上来说，链接器合并一个或多个可重定位文件来组成输出。它首先决定  
怎样合并、定位输入文件，然后更新符号值，最后进行重定位。对于可执行文件  
和共享的目标文件而言，重定位过程是相似的并有相同的结果。下面的描述使用  
如下的约定符号。  
  
\* A  
  
  表示用于计算可重定位的域值的加数。  
  
\* B  
  
  表示了在执行过程中一个共享目标被加载到内存时的基地址。一般情况下，一个  
  共享object文件使用的基虚地址为0，但是一个可执行地址就跟共享object文件  
  不同了。  
  
\* G  
  
  表示了在执行过程中重定位入口符号驻留在全局偏移表中的偏移。请参阅  
  第二部分中的“ Global Offset Table （全局偏移表）”获得更多  
  的信息。    
  
\* GOT  
  
  表示了全局偏移表的地址。请参阅第二部分中的“ Global Offset Table   
  （全局偏移表）”获得更多的信息。  
  
\* L  
  
  表示一个符号的过程链接表入口的位置（ section 偏移或地址）。一个过程  
  链接表入口重定位一个函数调用到正确的目的单元。链接器创建初始的链接表，  
  而动态链接器在执行中修改入口。  
  请参阅第二部分中的“ Procedure Linkage Table （过程链接表）”获得更多  
  的信息  
  
\* P  
  
  表示（section 偏移或地址）被重定位的存储单元位置（使用 r\_offset 计算的）。  
  
\* S  
  
  表示索引驻留在重定位入口处的符号值。  
  
一个重定位入口的 r\_offset 值指定了受影响的存储单元的首字节的偏移  
或虚拟地址。重定位类型指定了哪一位(bit)将要改变，以及怎样计算它们的值。  
在 SYSTEM V 体系中仅仅使用 Elf32\_Rel 重定位入口，将要被重定位的域中  
保留了加数。在所有的情况下，加数和计算结果使用相同字节顺序。  
  
+ Figure 1-22（表 1-22）: Relocation Types（重定位类型）  
  
  Name            Value  Field   Calculation  
  ====            =====  =====   ===========  
  R\_386\_NONE        0    none    none  
  R\_386\_32     1    word32  S + A  
  R\_386\_PC32     2    word32  S + A - P  
  R\_386\_GOT32     3    word32  G + A - P  
  R\_386\_PLT32     4    word32  L + A - P  
  R\_386\_COPY     5    none    none  
  R\_386\_GLOB\_DAT    6    word32  S  
  R\_386\_JMP\_SLOT    7    word32  S  
  R\_386\_RELATIVE    8    word32  B + A  
  R\_386\_GOTOFF     9    word32  S + A - GOT  
  R\_386\_GOTPC    10    word32  GOT + A - P  
  
有的重定位类型有不同于简单计算的语义。  
  
\* R\_386\_GOT32  
  
  这种重定位类型计算全局偏移表基地址到符号的全局偏移表  
  入口之间的间隔。这样另外通知了 link editor 建立一个全局偏移表 。  
  
  
\* R\_386\_PLT32  
  
  这种重定位类型计算符号的过程链接表入口地址，并另外通知链接器建立一个  
  过程链接表。  
  
  
\* R\_386\_COPY  
  
  链接器创建该重定位类型用于动态链接。它的偏移成员涉及一个可写段中的一个  
  位置。符号表索引指定一个可能存在于当前 object file 或在一个shared object  
  中的符号。在执行过程中，动态链接器把和 shared object 符号相关的数据  
  拷贝到该偏移所指定的位置。  
  
\* R\_386\_GLOB\_DAT  
  
  这种重定位类型用于设置一个全局偏移表入口为指定符号的地址。该特定的重定位  
  类型允许你决定符号和全局偏移表入口之间的一致性。  
  
  
\* R\_386\_JMP\_SLOT {\*}  
  
  链接器创建该重定位类型用于动态链接。其偏移成员给出了一个过程链接表入口的  
  位置。动态链接器修改该过程链接表入口以便向特定的符号地址传递控制。  
  [参阅第二部分中的 "Procedure Linkage Table(过程链接表)"]  
  
\* R\_386\_RELATIVE  
  
  链接器创建该重定位类型用于动态链接。其偏移成员给出了包含表达相关地址值  
  的一个 shared object 中的位置。动态链接器计算相应的虚拟地址（把该  
  shared object 装载地址和相对地址相加）。该类型的重定位入口必须为  
  符号表索引指定为 0 。  
  
  
\* R\_386\_GOTOFF  
  
  这种重定位类型计算符号值和全局偏移表地址之间的不同。另外还通知链接器  
  建立全局偏移表（GOT）。  
  
  
  
\* R\_386\_GOTPC  
  
  这种重定位类型类似于 R\_386\_PC32 ，不同的是它在计算中使用全局偏移表。  
  这种重定位中引用的符号通常是 \_GLOBAL\_OFFSET\_TABLE\_ ，该符号通知了  
  链接器建立全局偏移表（GOT）。  
  
   \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
  
2. PROGRAM LOADING AND DYNAMIC LINKING  
                   程序装入和动态链接  
   \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
  
   ======================== Introduction（介绍） =========================  
  
  
第二部分描述了 object file 信息和创建运行程序的系统行为。其中部分信息  
适合所有的系统，其他信息是和特定处理器相关的。  
  
  
可执行和共享的 object file 静态的描绘了程序。为了执行这样的程序，系统  
用这些文件创建动态的程序表现，或进程映像。一个进程映像有用于保存其代码、  
数据、堆栈等等的段。这个部分的主要章节讨论如下的内容。  
  
  
\* 程序头（Program header）。该章节补充第一部分，描述和程序运行相关的  
  object file 结构。即文件中主要的数据结构、程序头表、定位段映像，也  
  包含了为该程序创建内存映像所需要的信息。  
  
\* 载入程序（Program loading）。在给定一个 object file 时，系统为了  
  让它运行必须将它载入内存。  
  
\* 动态链接（Dynamic linking）。在载入了程序之后，系统必须通过解决组  
  成该进程的 object file之间的符号引用问题来完成进程映像的过程。  
  
注意：指定了处理器范围的 ELF 常量是有命名约定的。比如，DT\_ , PT\_ ，  
用于特定处理器扩展名，组合了处理器的名称（如 DT\_M32\_SPECIAL ）。  
没有使用这种约定但是预先存在的处理器扩展名是允许的。  
  
  
       Pre-existing Extensions  
                       （预先存在的扩展名）  
       =======================  
      DT\_JMP\_REL  
  
  
   ====================== Program Header（程序头）  ======================  
  
  
一个可执行的或共享的 object file 的程序头表是一个结构数组，每一个  
结构描述一个段或其他系统准备执行该程序所需要的信息。一个 object file  
段包含一个或多个部分（就象下面的“段目录”所描述的那样）。程序头仅仅对于  
可执行或共享的 object file 有意义。一个文件使用 ELF 头的 e\_phentsize  
和 e\_phnum 成员来指定其拥有的程序头大小。[参阅 第一部分中的 "ELF 头"]  
  
  
+ Figure 2-1: Program Header  
  
  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;  
  
\* p\_type  
  
  该成员指出了这个数组的元素描述了什么类型的段，或怎样解释该数组元素的信息。  
  类型值和含义如下所述。  
  
\* p\_offset  
  
  该成员给出了该段的驻留位置相对于文件开始处的偏移。  
  
\* p\_vaddr  
  
  该成员给出了该段在内存中的首字节地址。  
  
\* p\_paddr  
  
  在物理地址定位有关联的系统中，该成员是为该段的物理地址而保留的。由于  
  System V 忽略了应用程序的物理地址定位，该成员对于可执行文件和共享的  
  object 而言是未指定内容的。  
  
  
\* p\_filesz  
  
  该成员给出了文件映像中该段的字节数；它可能是 0 。  
    
\* p\_memsz  
  
  该成员给出了内存映像中该段的字节数；它可能是 0 。  
  
\* p\_flags  
  
  该成员给出了和该段相关的标志。定义的标志值如下所述。  
  
\* p\_align  
  
  就象在后面“载入程序”部分中所说的那样，可载入的进程段必须有合适的  
  p\_vaddr 、 p\_offset 值，取页面大小的模。该成员给出了该段在内存和  
  文件中排列值。 0 和 1 表示不需要排列。否则， p\_align 必须为正的 2 的幂，  
  并且 p\_vaddr 应当等于 p\_offset 模 p\_align 。  
  
  
某些入口描述了进程段；其他的则提供补充信息并且无益于进程映像。已经  
定义的入口可以以任何顺序出现，除非是下面明确声明的。后面是段类型值；  
其他的值保留以便将来用于其他用途。  
  
  
+ Figure 2-2: Segment Types, p\_type  
  
  Name             Value  
  ====             =====  
  PT\_NULL              0  
  PT\_LOAD              1  
  PT\_DYNAMIC           2  
  PT\_INTERP            3  
  PT\_NOTE              4  
  PT\_SHLIB             5  
  PT\_PHDR              6  
  PT\_LOPROC   0x70000000  
  PT\_HIPROC   0x7fffffff  
  
\* PT\_NULL  
  
  该数组元素未使用；其他的成员值是未定义的。这种类型让程序头表忽略入口。  
  
\* PT\_LOAD  
  
  该数组元素指定一个可载入的段，由 p\_filesz 和 p\_memsz 描述。文件中  
  字节被映射到内存段中。如果该段的内存大小（ p\_memsz ）比文件大小（ p\_filesz ）  
  要大，则多出的字节将象段初始化区域那样保持为 0 。文件的大小不会比内存大小值大。  
  在程序头表中，可载入段入口是以 p\_vaddr 的升序排列的。  
  
\* PT\_DYNAMIC  
  
  该数组元素指定动态链接信息。参阅 后面的“动态部分”以获得更多信息。  
  
\* PT\_INTERP  
  
  该数组元素指定一个 null-terminated 路径名的位置和大小（作为解释程序）。  
  这种段类型仅仅对可执行文件有意义（尽管它可能发生在一个共享 object 上）；  
  它在一个文件中只能出现一次。如果它出现，它必须先于任何一个可载入段入口。  
  参阅 后面的“程序解释器”（Program Interpreter）以获得更多的信息。  
  
  
\* PT\_NOTE  
  
  该数组元素指定辅助信息的位置和大小。参阅 后面的“注意部分”以获得细节。  
  
\* PT\_SHLIB  
  
  该段类型保留且具有未指定的语义。具有一个这种类型数组元素的程序并不  
  遵守 ABI 。  
  
\* PT\_PHDR  
  
  该数组元素（如果出现），指定了程序头表本身的位置和大小（包括在文件中  
  和在该程序的内存映像中）。更进一步来说，它仅仅在该程序头表是程序内存映像  
  的一部分时才有效。如果它出现，它必须先于任何可载入段入口。参阅 后面的  
  “程序解释器”（Program Interpreter）以获得更多的信息。  
  
  
\* PT\_LOPROC through PT\_HIPROC  
  
  该范围中的值保留用于特定处理器的语义。  
  
注意：除非在别处的特殊要求，所有的程序头的段类型是可选的。也就是说，  
一个文件的程序头表也许仅仅包含和其内容相关的元素。  
  
  
  
  Base Address（基地址）  
  
可执行和共享的 object file 有一个基地址，该基地址是与程序的 object file  
在内存中映像相关的最低虚拟地址。基地址的用途之一是在动态链接过程中重定位  
该程序的内存映像。  
  
  
一个可执行的 object file 或 一个共享的 object file 的基地址是在  
执行的时候从三个值计算而来的：内存载入地址、页面大小的最大值 和 程序可  
载入段的最低虚拟地址。就象在“程序载入”中所描述的那样，程序头中的虚拟地址  
也许和程序的内存映像中实际的虚拟地址并不相同。为了计算基地址，必须确定与  
PT\_LOAD 段 p\_vaddr 的最小值相关的内存地址。获得基地址的方法是将内存  
地址截去最大页面大小的最接近的整数倍。由于依赖载入内存中的文件类型，  
该内存地址和 p\_vaddr 值可能匹配也可能不匹配。  
  
  
就象在第一部分中 "Section" 中描述的那样， .bss section 具有 SHT\_NOBITS  
的类型。尽管在文件中不占用空间，它在段的内存映像中起作用。通常，没有初始化  
的数据驻留在段尾，因此使得在相关的程序头元素中的 p\_memsz 比 p\_filesz 大。  
  
  
  Note Section（注解部分）  
  
有的时候供应商或系统设计者需要用特定的信息标记一个  
object file 以便其他程序检查其兼容的一致性，等等此类。 SHT\_NOTE   
类型的 section 和 PT\_NOTE 类型的程序头元素能够被用于此目的。 section   
和程序头中的注解信息包含了任意数目的入口，每一个入口的格式都是对应于特定  
处理器格式的 4-字节数组。下面的标签有助于解释注释信息的组织形式，但是这些  
标签不是规格说明的一部分。  
  
  
+ Figure 2-3: Note Information  
  
  namesz  
  descsz  
  type  
  name ...  
  desc ...  
  
\* namesz and name  
  
  名字中 namesz 的第一个字节包含了一个 null-terminated 字符  
  表达了该入口的拥有者或始发者。没有正式的机制来避免名字冲突。从  
  惯例来说，供应商使用他们自己的名称，比如 "XYZ Computer Company" ，  
  作为标志。如果没有提供名字， namesz 值为 0 。 如果有必要，确定  
  描述信息4-字节对齐。 这样的填充信息并不包含在namesz 中。  
  
  
\* descsz and desc  
  
  desc 中 descsz 的首字节包含了注解描述符。ABI 不会在一个描述符内容中  
  放入任何系统参数。如果没有描述符， descsz 将为 0 。  如果有必要，确定  
  描述信息4-字节对齐。 这样的填充信息并不包含在descsz中。  
  
  
\* type  
  
  该 word 给出了描述符的解释。每一个创造着（originator） 控制着自己的类型；  
  对于单单一个类型值的多种解释是可能存在的。因此，一个程序必须辨认出该名字  
  和其类型以便理解一个描述符。这个时候的类型必须是非负的。ABI 没有定义  
  描述符的含义。  
  
为了举例说明，下面的解释段包含两个入口。  
  
+ Figure 2-4: Example Note Segment  
  
           +0   +1   +2   +3  
          -------------------  
  namesz           7  
  descsz           0           No descriptor  
    type           1  
    name   X    Y    Z    spc   
           C    o    /0   pad  
  namesz           7  
  descsz           8  
    type           3  
    name   X    Y    Z    spc  
           C    o    /0   pad  
    desc         word0  
                 word1  
  
注意：系统保留的注解信息没有名字 (namesz==0) ，有一个零长度的名字  
(name[0]=='/0') 现在还没有类型为其定义。所有其他的名字必须至少有  
一个非空的字符。  
  
  
注意：注解信息是可选的。注解信息的出现并不影响一个程序的 ABI 一致性，  
前提是该信息不影响程序的执行行为。否则，该程序将不遵循 ABI 并将出现  
未定义的行为。  
  
  
   ===================== Program Loading（程序载入） =====================  
  
当创建或增加一个进程映像的时候，系统在理论上将拷贝一个文件的段到一个虚拟  
的内存段。系统什么时候实际地读文件依赖于程序的执行行为，系统载入等等。一个  
进程仅仅在执行时需要引用逻辑页面的时候才需要一个物理页面，实际上进程通常会  
留下许多未引用的页面。因此推迟物理上的读取常常可以避免这些情况，改良系统的  
特性。为了在实践中达到这种效果，可执行的和共享的 object file 必须具有  
合适于页面大小取模值的文件偏移和虚拟地址这样条件的段映像。  
  
虚拟地址和文件偏移在 SYSTEM V 结构的段中是模 4KB（0x1000） 或大的 2 的幂。  
由于 4KB 是最大的页面大小，因此无论物理页面大小是多少，文件必须去适合页面。  
  
  
+ Figure 2-5: Executable File  
  
           File Offset   File                  Virtual Address  
           ===========   ====                  ===============  
                     0   ELF header  
  Program header table  
                         Other information  
                 0x100   Text segment          0x8048100  
                         ...  
                         0x2be00 bytes         0x8073eff  
               0x2bf00   Data segment          0x8074f00  
                         ...  
                         0x4e00 bytes          0x8079cff  
               0x30d00   Other information  
                         ...  
  
+ Figure 2-6: Program Header Segments（程序头段）  
  
  Member    Text Data  
  ======    ====         ====  
  p\_type    PT\_LOAD      PT\_LOAD  
  p\_offset  0x100 0x2bf00  
  p\_vaddr   0x8048100 0x8074f00  
  p\_paddr   unspecified unspecified  
  p\_filesz  0x2be00 0x4e00  
  p\_memsz   0x2be00 0x5e24  
  p\_flags   PF\_R+PF\_X    PF\_R+PF\_W+PF\_X  
  p\_align   0x1000 0x1000  
  
尽管示例中的文件偏移和虚拟地址在文本和数据两方面都适合模 4KB ，但是还有  
4 个文件页面混合了代码和数据（依赖于页面大小和文件系统块的大小）。  
  
  
\* 第一个文本页面包含了 ELF 头、程序头以及其他信息。  
\* 最后的文本页包含了一个数据开始的拷贝。  
\* 第一个数据页面有一个文本结束的拷贝。  
\* 最后的数据页面也许会包含与正在运行的进程无关的文件信息。  
  
理论上，系统强制内存中段的区别；段地址被调整为适应每一个逻辑页面在地址空间  
中有一个简单的准许集合。在上面的示例中，包含文本结束和数据开始的文件区域将  
被映射两次：在一个虚拟地址上为文本而另一个虚拟地址上为数据。  
  
  
数据段的结束处需要对未初始化的数据进行特殊处理（系统定义的以 0 值开始）。  
因此如果一个文件包含信息的最后一个数据页面不在逻辑内存页面中，则无关的  
数据应当被置为 0 （这里不是指未知的可执行文件的内容）。在其他三个页面中  
"Impurities" 理论上并不是进程映像的一部分；系统是否擦掉它们是未指定的。  
下面程序的内存映像假设了 4KB 的页面。  
  
  
+ Figure 2-7: Process Image Segments（进程映像段）  
  
  Virtual Address  Contents            Segment  
  ===============  ========            =======  
        0x8048000  Header padding      Text  
                   0x100 bytes  
        0x8048100  Text segment  
                   ...  
                   0x2be00 bytes  
        0x8073f00  Data padding  
                   0x100 bytes  
        0x8074000  Text padding        Data  
                   0xf00 bytes  
        0x8074f00  Data segment  
                   ...  
                   0x4e00 bytes  
        0x8079d00  Uninitialized data  
                   0x1024 zero bytes  
        0x807ad24  Page padding  
                   0x2dc zero bytes  
  
可执行文件和共享文件在段载入方面有所不同。典型地，可执行文件段包含了  
绝对代码。为了让进程正确执行，这些段必须驻留在建立可执行文件的虚拟地址  
处。因此系统使用不变的 p\_vaddr 作为虚拟地址。  
  
  
另一方面，共享文件段包含与位置无关的代码。这让不同进程的相应段虚拟地址  
各不相同，且不影响执行。虽然系统为各个进程选择虚拟地址，它还要维护各个  
段的相对位置。因为位置无关的代码在段间使用相对定址，故而内存中的虚拟地址  
的不同必须符合文件中虚拟地址的不同。下表给出了几个进程可能的共享对象虚拟  
地址的分配，演示了不变的相对定位。该表同时演示了基地址的计算。  
  
  
+ Figure 2-8: Example Shared Object Segment Addresses  
  
  Sourc             Text        Data  Base Address  
  =====             ====        ====  ============  
  File             0x200     0x2a400           0x0  
  Process 1   0x80000200  0x8002a400    0x80000000  
  Process 2   0x80081200  0x800ab400    0x80081000  
  Process 3   0x900c0200  0x900ea400    0x900c0000  
  Process 4   0x900c6200  0x900f0400    0x900c6000  
  
  
   ==================== Dynamic Linking （动态链接） =====================  
  
  
一个可执行文件可能有一个 PT\_INTERP 程序头元素。在 exec(BA\_OS) 的  
过程中，系统从 PT\_INTERP 段中取回一个路径名并由解释器文件的段创建初始的  
进程映像。也就是说，系统为解释器“编写”了一个内存映像，而不是使用原始  
的可执行文件的段映像。此时该解释器就负责接收系统来的控制并且为应用程序  
提供一个环境变量。  
  
  
解释器使用两种方法中的一种来接收系统来的控制。首先，它会接收一个文件描述符  
来读取该可执行文件，定位于开头。它可以使用这个文件描述符来读取 并且（或者）  
映射该可执行文件的段到内存中。其次，依赖于该可执行文件的格式，系统会载入  
这个可执行文件到内存中而不是给该解释器一个文件描述符。伴随着可能的文件描述符  
异常的情况，解释器的初始进程声明应匹配该可执行文件应当收到的内容。解释器本身  
并不需要第二个解释器。一个解释器可能是一个共享对象也可能是一个可执行文件。  
  
  
\* 一个共享对象（通常的情况）在被载入的时候是位置无关的，各个进程可能不同；  
  系统在 mmap(KE\_OS) 使用的动态段域为它创建段和相关的服务。因而，一个  
  共享对象的解释器将不会和原始的可执行文件的原始段地址相冲突。  
  
  
\* 一个可执行文件被载入到固定地址；系统使用程序头表中的虚拟地址为其创建段。  
  因而，一个可执行文件解释器的虚拟地址可能和第一个可执行文件相冲突；这种  
  冲突由解释器来解决。  
  
  
  
Dynamic Linker（动态链接器）  
  
当使用动态链接方式建立一个可执行文件时，链接器把一个 PT\_INTERP 类型  
的元素加到可执行文件中，告诉系统把动态链接器做为该程序的解释器。  
  
注意：由系统提供的动态链接器是和特定处理器相关的。  
  
Exec(BA\_OS) 和动态链接器合作为程序创建进程，必须有如下的动作：  
  
\* 将可执行文件的内存段加入进程映像中；  
\* 将共享对象的内存段加入进程映像中；  
\* 为可执行文件和它的共享对象进行重定位；  
\* 如果有一个用于读取可执行文件的文件描述符传递给了动态链接器，那么关闭它。  
\* 向程序传递控制，就象该程序已经直接从 exec(BA\_OS) 接收控制一样。  
  
链接器同时也为动态链接器构建各种可执行文件和共享对象文件的相关数据。就象  
在上面“程序头”中说的那样，这些数据驻留在可载入段中，使得它们在执行过程  
中有效。（再一次的，要记住精确的段内容是处理器相关的。可以参阅相应处理器  
的补充说明来获得详尽的信息。）  
  
  
\* 一个具有 SHT\_DYNAMIC 类型的 .dynamic section 包含各种数据。驻留在  
  section 开头的结构包含了其他动态链接信息的地址。  
  
\* SHT\_HASH 类型的 .hash section 包含了一个 symbol hash table.  
  
\* SHT\_PROGBITS 类型的 .got 和 .plt section 包含了两个分离的 table:  
  全局偏移表和过程链接表。 下面的 section 演示了动态链接器使用和改变  
  这些表来为 object file 创建内存映像。  
  
  
由于每一个遵循 ABI 的程序从一个共享对象库中输入基本的系统服务，因此动态  
链接器分享于每一个遵循 ABI 的程序的执行过程中。  
  
  
就象在处理器补充说明的“程序载入”所解释的那样，共享对象也许会占用与记录在  
文件的程序头表中的地址不同的虚拟内存地址。动态链接器重定位内存映像，在应用程序  
获得控制之前更新绝对地址。尽管在库被载入到由程序头表指定的地址的情况下绝对地址  
应当是正确的，通常的情况却不是这样。  
  
  
如果进程环境 [see exec(BA\_OS)] 包含了一个非零的 LD\_BIND\_NOW 变量，  
动态链接器将在控制传递到程序之前进行所有的重定位。举例而言，所有下面的  
环境入口将指定这种行为。  
  
  
\* LD\_BIND\_NOW=1  
\* LD\_BIND\_NOW=on  
\* LD\_BIND\_NOW=off  
  
其他情况下， LD\_BIND\_NOW 或者不在环境中或者为空值。动态链接器可以不急于  
处理过程链接表入口，因而避免了对没有调用的函数的符号解析和重定位。参阅  
"Procedure Linkage Table"获取更多的信息。  
  
  
  
   Dynamic Section（动态section）  
  
  
假如一个object文件参与动态的连接，它的程序头表将有一个类型为PT\_DYNAMIC  
的元素。该“段”包含了.dynamic section。一个\_DYNAMIC特别的符号，表明了  
该section包含了以下结构的一个数组。  
  
+ Figure 2-9: Dynamic Structure  
  
  typedef struct {  
      Elf32\_Sword d\_tag;  
      union {  
          Elf32\_Sword d\_val;  
          Elf32\_Addr d\_ptr;  
      } d\_un;  
  } Elf32\_Dyn;  
  
  extern Elf32\_Dyn \_DYNAMIC[];  
  
对每一个有该类型的object，d\_tag控制着d\_un的解释。  
  
\* d\_val  
  
  那些Elf32\_Word object描绘了具有不同解释的整形变量。  
  
\* d\_ptr  
  
  那些Elf32\_Word object描绘了程序的虚拟地址。就象以前提到的，在执行时，  
  文件的虚拟地址可能和内存虚拟地址不匹配。当解释包含在动态结构中的地址  
  时是基于原始文件的值和内存的基地址。为了一致性，文件不包含在  
  重定位入口来纠正在动态结构中的地址。  
  
  
以下的表格总结了对可执行和共享object文件需要的tag。假如tag被标为  
mandatory，ABI-conforming文件的动态连接数组必须有一个那样的入口。  
同样的，“optional”意味着一个可能出现tag的入口，但是不是必须的。  
  
+ Figure 2-10: Dynamic Array Tags, d\_tag  
  
  Name               Value  d\_un         Executable   Shared Object  
  ====               =====  ====         ==========   =============  
  DT\_NULL                0  ignored mandatory    mandatory  
  DT\_NEEDED 1  d\_val optional     optional  
  DT\_PLTRELSZ 2  d\_val optional     optional  
  DT\_PLTGOT 3  d\_ptr optional     optional  
  DT\_HASH 4  d\_ptr mandatory    mandatory  
  DT\_STRTAB 5  d\_ptr mandatory    mandatory  
  DT\_SYMTAB 6  d\_ptr mandatory    mandatory  
  DT\_RELA 7  d\_ptr mandatory    optional  
  DT\_RELASZ 8  d\_val mandatory    optional  
  DT\_RELAENT 9  d\_val mandatory    optional  
  DT\_STRSZ 10  d\_val mandatory    mandatory  
  DT\_SYMENT 11  d\_val mandatory    mandatory  
  DT\_INIT 12  d\_ptr optional     optional  
  DT\_FINI 13  d\_ptr optional     optional  
  DT\_SONAME 14  d\_val ignored      optional  
  DT\_RPATH 15  d\_val optional     ignored  
  DT\_SYMBOLIC 16  ignored ignored      optional  
  DT\_REL 17  d\_ptr mandatory    optional  
  DT\_RELSZ 18  d\_val mandatory    optional  
  DT\_RELENT 19  d\_val mandatory    optional  
  DT\_PLTREL 20  d\_val optional     optional  
  DT\_DEBUG 21  d\_ptr optional     ignored  
  DT\_TEXTREL 22  ignored optional     optional  
  DT\_JMPREL 23  d\_ptr optional     optional  
  DT\_LOPROC     0x70000000  unspecified  unspecified  unspecified  
  DT\_HIPROC     0x7fffffff  unspecified  unspecified  unspecified  
  
\* DT\_NULL  
  
   一个DT\_NULL标记的入口表示了\_DYNAMIC数组的结束。  
  
\* DT\_NEEDED  
  
  这个元素保存着以NULL结尾的字符串表的偏移量，那些字符串是所需库的名字。  
  该偏移量是以DT\_STRTAB 为入口的表的索引。看“Shared Object Dependencies”  
  关于那些名字的更多信息。动态数组可能包含了多个这个类型的入口。那些  
  入口的相关顺序是重要的，虽然它们跟其他入口的关系是不重要的。  
  
\* DT\_PLTRELSZ  
  
  该元素保存着跟PLT关联的重定位入口的总共字节大小。假如一个入口类型  
  DT\_JMPREL存在，那么DT\_PLTRELSZ也必须存在。  
  
\* DT\_PLTGOT  
  
  该元素保存着跟PLT关联的地址和（或者）是GOT。具体细节看处理器补充  
  （processor supplement）部分。  
  
\* DT\_HASH  
  
  该元素保存着符号哈希表的地址，在“哈希表”有描述。该哈希表指向  
  被DT\_SYMTAB元素引用的符号表。  
  
\* DT\_STRTAB  
  
  该元素保存着字符串表地址，在第一部分有描述，包括了符号名，库名，  
  和一些其他的在该表中的字符串。  
  
\* DT\_SYMTAB  
  
  该元素保存着符号表的地址，在第一部分有描述，对32-bit类型的文件来  
  说，关联着一个Elf32\_Sym入口。  
  
\* DT\_RELA  
  
  该元素保存着重定位表的地址，在第一部分有描述。在表中的入口有明确的  
  加数，就象32-bit类型文件的Elf32\_Rela。一个object文件可能好多个重定位  
  section。当为一个可执行和共享文件建立重定位表的时候，连接编辑器连接  
  那些section到一个单一的表。尽管在object文件中那些section是保持独立的。  
  动态连接器只看成是一个简单的表。当动态连接器为一个可执行文件创建一个  
  进程映象或者是加一个共享object到进程映象中，它读重定位表和执行相关的  
  动作。假如该元素存在，动态结构必须也要有DT\_RELASZ和DT\_RELAENT元素。  
  当文件的重定位是mandatory，DT\_RELA 或者 DT\_REL可能出现（同时出现是  
  允许的，但是不必要的）。  
  
  
\* DT\_RELASZ  
  
  该元素保存着DT\_RELA重定位表总的字节大小。  
  
\* DT\_RELAENT  
  
  该元素保存着DT\_RELA重定位入口的字节大小。  
    
  
\* DT\_STRSZ  
  
  该元素保存着字符串表的字节大小。  
  
\* DT\_SYMENT  
  
  该元素保存着符号表入口的字节大小。  
  
\* DT\_INIT  
  
  该元素保存着初始化函数的地址，在下面“初始化和终止函数”中讨论。  
  
\* DT\_FINI  
  
  该元素保存着终止函数的地址，在下面“初始化和终止函数”中讨论。   
  
\* DT\_SONAME  
  
  该元素保存着以NULL结尾的字符串的字符串表偏移量，那些名字是共享  
  object的名字。偏移量是在DT\_STRTAB入口记录的表的索引。关于那些名字看  
  Shared Object Dependencies 部分获得更多的信息。  
  
\* DT\_RPATH  
  
  该元素保存着以NULL结尾的搜索库的搜索目录字符串的字符串表偏移量。  
  在共享object依赖关系（Shared Object Dependencies）中有讨论  
  
\* DT\_SYMBOLIC  
  
  在共享object库中出现的该元素为在库中的引用改变动态连接器符号解析的算法。  
  替代在可执行文件中的符号搜索，动态连接器从它自己的共享object开始。假如  
  一个共享的object提供引用参考失败，那么动态连接器再照常的搜索可执行文件  
  和其他的共享object。  
    
  
\* DT\_REL  
  
  该元素相似于DT\_RELA，除了它的表有潜在的加数，正如32-bit文件类型的  
  Elf32\_Rel一样。假如这个元素存在，它的动态结构必须也同时要有DT\_RELSZ  
  和DT\_RELENT的元素。  
  
\* DT\_RELSZ  
  
  该元素保存着DT\_REL重定位表的总字节大小。  
  
\* DT\_RELENT  
  
  该元素保存着DT\_RELENT重定为入口的字节大小。     
  
\* DT\_PLTREL  
  
  该成员指明了PLT指向的重定位入口的类型。适当地， d\_val成员保存着   
  DT\_REL或DT\_RELA。在一个PLT中的所有重定位必须使用相同的转换。  
  
\* DT\_DEBUG  
  
  该成员被调试使用。它的内容没有被ABI指定；访问该入口的程序不是  
  ABI-conforming的。  
  
\* DT\_TEXTREL  
  
  如在程序头表中段许可所指出的那样，这个成员的缺乏代表没有重置入  
  口会引起非写段的修改。假如该成员存在，一个或多个重定位入口可能  
  请求修改一个非写段，并且动态连接器能因此有准备。  
    
  
\* DT\_JMPREL  
  
  假如存在，它的入口d\_ptr成员保存着重定位入口（该入口单独关联着  
  PLT）的地址。假如lazy方式打开，那么分离它们的重定位入口让动态连接  
  器在进程初始化时忽略它们。假如该入口存在，相关联的类型入口DT\_PLTRELSZ  
  和DT\_PLTREL一定要存在。  
  
\* DT\_LOPROC through DT\_HIPROC  
  
  在该范围内的变量为特殊的处理器语义保留。除了在数组末尾的DT\_NULL元素，  
  和DT\_NEEDED元素相关的次序，入口可能出现在任何次序中。在表中不出  
  现的Tag值是保留的。  
  
  
  Shared Object Dependencies（共享Object的依赖关系）  
  
当连接器处理一个文档库时，它取出库中成员并且把它们拷贝到一个输出的  
object文件中。当运行时没有包括一个动态连接器的时候，那些静态的连接服  
务是可用的。共享object也提供服务，动态连接器必须把正确的共享object  
文件连接到要实行的进程映象中。因此，可执行文件和共享的object文件之间  
存在着明确的依赖性。  
  
当动态连接器为一个object文件创建内存段时，依赖关系（在动态结构的  
DT\_NEEDED入口中记录）表明需要哪些object来为程序提供服务。通过  
重复的连接参考的共享object和他们的依赖关系，动态连接器可以建造一个  
完全的进程映象。当解决一个符号引用的时候，动态连接器以宽度优先搜索  
（breadth-first）来检查符号表，换句话说，它先查看自己的可实行程序  
中的符号表，然后是顶端DT\_NEEDED入口（按顺序）的符号表，再接下来是  
第二级的DT\_NEEDED入口，依次类推。共享object文件必须对进程是可读的；  
其他权限是不需要的。  
  
注意：即使当一个共享object被引用多次（在依赖列关系表中），动态连接器  
只把它连接到进程中一次。  
  
在依赖关系列表中的名字既被DT\_SONAME字符串拷贝，又被建立object文件  
时的路径名拷贝。例如，动态连接器建立一个可执行文件（使用带DT\_SONAME  
入口的lib1共享文件）和一个路径名为/usr/lib/lib2的共享object库，  
那么可执行文件将在它自己的依赖关系列表中包含lib1和/usr/bin/lib2。  
  
假如一个共享object名字有一个或更多的反斜杠字符(/)在这名字的如何地方，  
例如上面的/usr/lib/lib2文件或目录，动态连接器把那个字符串自己做为路径名。  
假如名字没有反斜杠字符(/)，例如上面的lib1，三种方法指定共享文件的  
搜索路径，如下：  
  
\* 第一，动态数组标记DT\_RPATH保存着目录列表的字符串（用冒号(:)分隔）。  
  例如，字符串/home/dir/lib:/home/dir2/lib:告诉动态连接器先搜索  
  /home/dir/lib，再搜索/home/dir2/lib，再是当前目录。  
  
\* 第二，在进程环境中（see exec(BA\_OS)），有一个变量称为LD\_LIBRARY\_PATH  
  可以保存象上面一样的目录列表（随意跟一个分号(;)和其他目录列表）。  
  以下变量等于前面的例子：  
    LD\_LIBRARY\_PATH=/home/dir/lib:/home/dir2/lib:  
    LD\_LIBRARY\_PATH=/home/dir/lib;/home/dir2/lib:  
    LD\_LIBRARY\_PATH=/home/dir/lib:/home/dir2/lib:;  
  所以的LD\_LIBRARY\_PATH目录在DT\_RPATH指向的目录之后被搜索。尽管一些  
  程序（例如连接编辑器）不同的处理分号前和分号后的目录，但是动态连接  
  不会。不过，动态连接器接受分号符号，具体语意在如上面描述。  
  
\* 最后，如果上面的两个目录查找想要得到的库失败，那么动态连接器搜索  
  /usr/lib.  
  
注意：出于安全考虑，动态连接器忽略set-user和set-group的程序的  
LD\_LIBRARY\_PATH所指定的搜索目录。但它会搜索DT\_RPATH指明的目录和  
/usr/lib。  
  
  
Global Offset Table(GOT全局偏移量表)  
  
一般情况下，位置无关的代码不包含绝对的虚拟地址。全局偏移量表在私有数据  
中保存着绝对地址，所以应该使地址可用的，而不是和位置无关性和程序代码段  
共享能力妥协。一个程序引用它的GOT(全局偏移量表)来使用位置无关的地址并且  
提取绝对的变量，所以重定位位置无关的参考到绝对的位置。  
  
初始时，GOT(全局偏移量表)保存着它重定位入口所需要的信息 [看第一部分的  
“Relocation”]。在系统为一个可装载的object文件创建内存段以后，动态  
连接器处理重定位入口，那些类型为R\_386\_GLOB\_DAT的指明了GOT(全局偏移量表)。  
动态连接器决定了相关的标号变量，计算他们的绝对地址，并且设置适当的内存  
表入口到正确的变量。虽然当连接编辑器建造object文件的时候，绝对地址  
是不知道，连接器知道所以内存段的地址并且能够因此计算出包含在那里的  
标号地址。  
  
假如程序需要直接访问符号的绝对地址，那么这个符号将有一个GOT(全局偏移量表)  
入口。因为可执行文件和共享文件有独立的GOT(全局偏移量表)，一个符号地址  
可能出现在不同的几个表中。在交给进程映象的代码控制权以前，动态连接器处  
理所有的重定位的GOT(全局偏移量表)，所以在执行时，确认绝对地址是可用的。  
  
该表的入口0是为保存动态结构地址保留的（参考\_DYNAMIC标号）。这允许  
象动态连接程序那样来找出他们自己的动态结构（还没有处理他们的重  
定向入口）。这些对于动态连接器是重要的，因为它必要初始化自己而不  
能依赖于其他程序来重定位他们的内存映象。在32位Interl系统结构中，在  
GOT中的人口1和2也是保留的，具体看以下的过程连接表（Procedure Linkage  
Table）。  
  
系统可以为在不同的程序中相同的共享object选择不同的内存段；它甚至可以  
为相同的程序不同的进程选择不同的库地址。虽然如此，一旦进程映象被建立  
以后，内存段不改变地址。只要一个进程存在，它的内存段驻留在固定的虚拟  
地址。  
  
GOT表的格式和解释是处理器相关的。在32位Intel体系结构下，标号  
\_GLOBAL\_OFFSET\_TABLE\_可能被用来访问该表。  
  
+ Figure 2-11: Global Offset Table  
  
  extern Elf32\_Addr \_GLOBAL\_OFFSET\_TABLE\_[];  
  
标号\_GLOBAL\_OFFSET\_TABLE\_可能驻留在.got section的中间，允许负的和非负  
的下标索引这个数组。  
  
  
       Procedure Linkage Table（PLT过程连接表）  
  
就象GOT重定位把位置无关的地址计算成绝对地址一样，PLT过程连接表重定位  
位置无关的函数调用到绝对的地址。从一个可执行或者共享的object文件到另外的，  
连接编辑器不解析执行的传输（例如函数的调用）。因此，连接编辑器安排程序  
的传递控制到PLT中的入口。在SYSTEM V体系下，PLT存在共享文本中，但是它们  
使用的地址是在私有的GOT中。符号连接器决定了目标的绝对地址并且修改GOT的  
内存映象。因此，在没有危及到位置无关、程序文本的共享能力的情况下。动态  
连接器能重定位人口。  
  
  
+ Figure 2-12: Absolute Procedure Linkage Table {\*}  
       绝对的过程连接表  
  
  .PLT0:pushl   got\_plus\_4  
        jmp     \*got\_plus\_8  
        nop; nop  
        nop; nop  
  .PLT1:jmp     \*name1\_in\_GOT  
        pushl   $offset  
        jmp     .PLT0@PC  
  .PLT2:jmp     \*name2\_in\_GOT  
        pushl   $offset  
        jmp     .PLT0@PC  
        ...  
  
+ Figure 2-13: Position-Independent Procedure Linkage Table  
       位置无关（或者说位置独立）的过程连接表  
  .PLT0:pushl   4(%ebx)  
        jmp     \*8(%ebx)  
        nop; nop  
        nop; nop  
  .PLT1:jmp     \*name1@GOT(%ebx)  
        pushl   $offset  
        jmp     .PLT0@PC  
  .PLT2:jmp     \*name2@GOT(%ebx)  
        pushl   $offset  
        jmp     .PLT0@PC  
        ...  
  
注意：如图所示，PLT的指令使用了不同的操作数地址方式，对绝对代码和  
对位置无关的代码。但是，他们的界面对于动态连接器是相同的。  
  
以下的步骤，动态连接器和程序协作（cooperate）通过PLT和GOT来解析符号  
引用。  
  
1. 当第一次创建程序的内存映象时，动态连接器为在GOT中特别的变量设置  
   第二次和第三次的入口。下面关于那些变量有更多的解释。  
  
2. 假如PLT是位置无关的，那么GOT的地址一定是保留在%ebx中的。每个在进程  
   映象中共享的object文件有它自己的PLT，并且仅仅在同一个object文件中，  
   控制传输到PLT入口。从而，要调用的函数有责任在调用PLT入口前，设置PLT  
   地址到寄存器中。  
  
3. 举例说明，假如程序调用函数name1,它的传输控制到标号.PLT1.  
  
4. 第一个指令跳到在GOT入口的name1地址。初始话时，GOT保存着紧跟着的push1  
   指令的地址，而不是真实的name1的地址。  
  
5. 因此，程序在堆栈中压入(push)一个重定位的偏移量。重定位的偏移量是  
   一个32位，非负的字节偏移量（从定位表算起）。指派的重定位入口将是  
   一个R\_386\_JMP\_SLOT类型，它的偏移量指明了GOT入口（在前面的jmp指令中  
   被使用）。该重定位入口也包含一个符号表的索引，因此告诉动态连接器  
   哪个符号要被引用，在这里是name1。  
  
6. 在压入(push)一个重定位的偏移量后，程序跳到.PLT0,在PLT中的第一个入口。  
   push1指令在堆栈中放置第二个GOT入口(got\_plus\_4 or 4(%ebx))的值,  
   因此，给动态连接器一个word的鉴别信息。然后程序跳到第三个GOT入口  
   (got\_plus\_8 or 8(%ebx))，它传输控制到动态连接器。  
  
7. 当动态连接器接到控制权，它展开堆栈，查看指派的重定位入口，寻找符号的  
   值，在GOT入口中存储真实的name1地址，然后传输控制想要目的地。  
  
8. PLT入口的并发执行将直接传输控制到name1，而不用第二次调用动态连接器  
   了。所以，在.PLT1中的jmp指令将转到name1,代替“falling through”  
   转到pushl指令。  
  
LD\_BIND\_NOW环境变量能改变动态连接器的行为。假如这个变量为非空，动态  
连接器在传输控制到程序前计算PLT入口。换句话说，动态连接器处理重定位  
类型为R\_386\_JMP\_SLOT的入口在进程初始化时。否则，动态连接器计算PLT入口  
懒惰的，推迟到符号解析和重定位直到一个表入口的第一次执行。  
  
  
注意：一般来说，以懒惰（Lazy）方式绑定是对全应用程序执行的改进。  
因为不使用的符号就不会招致动态连接器做无用功。然而，对一些应用程序，  
两种情况使用懒惰（Lazy）方式是不受欢迎的。  
  
第一 初始的引用一个共享object函数比后来的调用要花的时间长，因为动  
     态连接器截取调用来解析符号。一些应用程序是不能容忍这样的。  
第二 假如这个错误发生并且动态连接器不能解析该符号，动态连接器将终止  
     程序。在懒惰（Lazy）方式下，这可能发生在任意的时候。一再的，一  
     些应用程序是不能容忍这样的。通过关掉懒惰（Lazy）方式，在应用程  
     序接到控制前，当在处理初始话时发生错误，动态连接器强迫程序，使  
     之失败。  
  
      Hash Table(哈希表)  
  
Elf32\_Word object的哈希表支持符号表的访问。  
标号出现在下面帮助解释哈希表的组织，但是它们不是规范的一部分。  
  
+ Figure 2-14: Symbol Hash Table  
  
  nbucket  
  nchain  
  bucket[0]  
  ...  
  bucket[nbucket - 1]  
  chain[0]  
  ...  
  chain[nchain - 1]  
  
  
bucket数组包含了nbucket入口，并且chain数组包含了nchain个入口;索引从0开始。  
bucket和chain保存着符号表的索引。Chain表入口类似于符号表。符号表入口的  
数目应该等于nchain；所以符号表的索引也选择chain表的入口。  
一个哈希函数(如下的)接受一个符号名并且返回一个可以被计算机使用的bucket索引  
的值。因此，假如一个哈希函数返回一些名字的值为X，那么bucket[x%nbucket]  
将给出一个索引y（既是符号表和chain表的索引）。假如符号表入口不是期望的，  
chain[y]给出下一个符号表的入口（使用相同的哈希变量）。可以沿着chain  
链直到选择到了期望名字的符号表入口或者是碰到了STN\_UNDEF的入口。  
  
+ Figure 2-15: Hashing Function  
  
  unsigned long  
  elf\_hash(const unsigned char \*name)  
  {  
      unsigned long       h = 0, g;  
    
      while (\*name) {  
          h = (h << 4) + \*name++;  
          if (g = h & 0xf0000000)  
              h ^= g >> 24;  
          h &= ~g;  
      }  
      return h;  
  }  
  
  
       Initialization and Termination Functions  
          初始化和终止函数  
  
  
在动态连接妻建立进程映象和执行重定位以后，每一个共享object得到适当  
的机会来执行一些初始话代码。初始化函数不按特别的顺序被调用，但是  
所有的共享object初始化发生在执行程序获得控制之前。  
  
类似地，共享的object可能包含终止函数，它们在进程本身开始它的终止之后  
被执行（以atexit(BA\_OS)的机制）。  
  
共享object通过设置在动态结构中的DT\_INIT和DT\_FINI入口来指派它们的初始化  
和终止函数，如上动态section（Dynamic Section）部分描述。典型的，那些函数  
代码存在.init和.fini section中，第一部分的“section”已经提到过。  
  
注意：尽管atexit(BA\_OS)的终止处理一般可可正常完成，但是不保证在死进程上  
被执行。特别的，假如\_exit被调用（看exit(BA\_OS)）或者假如进程死掉，那么  
进程是不执行终止处理的。因为它收到一个信号，该信号可捕获或忽略。  
  
   \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
  
     3. C LIBRARY  
  
   \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
  
   ========================== C Library ===========================  
  
C库，libc,包含了所有的符号（包含在libsys)，另外，包含在在下面两个  
表中列出的运行函数。第一个表中的运行函数是ANSI C标准的。  
  
+ Figure 3-1: libc Contents, Names without Synonyms  
  
  abort        fputc        isprint      putc         strncmp  
  abs        fputs        ispunct      putchar      strncpy  
  asctime      fread        isspace      puts         strpbrk  
  atof        freopen      isupper      qsort        strrchr  
  atoi        frexp        isxdigit     raise        strspn  
  atol        fscanf       labs         rand         strstr  
  bsearch      fseek        ldexp        rewind       strtod  
  clearerr     fsetpos      ldiv         scanf        strtok  
  clock        ftell        localtime    setbuf       strtol  
  ctime        fwrite       longjmp      setjmp       strtoul  
  difftime     getc         mblen        setvbuf      tmpfile  
  div        getchar      mbstowcs     sprintf      tmpnam  
  fclose       getenv       mbtowc       srand        tolower  
  feof        gets         memchr       sscanf       toupper  
  ferror       gmtime       memcmp       strcat       ungetc  
  fflush       isalnum      memcpy       strchr       vfprintf  
  fgetc        isalpha      memmove      strcmp       vprintf  
  fgetpos      iscntrl      memset       strcpy       vsprintf  
  fgets        isdigit      mktime       strcspn      wcstombs  
  fopen        isgraph      perror       strlen       wctomb  
  fprintf      islower      printf       strncat      
  
再加上, libc 保存着以下的服务。  
  
+ Figure 3-2: libc Contents, Names with Synonyms  
  
  \_\_assert     getdate      lockf \*\*     sleep        tell \*\*   
  cfgetispeed  getopt       lsearch      strdup       tempnam  
  cfgetospeed  getpass      memccpy      swab         tfind  
  cfsetispeed  getsubopt    mkfifo       tcdrain      toascii  
  cfsetospeed  getw         mktemp       tcflow       \_tolower  
  ctermid      hcreate      monitor      tcflush      tsearch  
  cuserid      hdestroy     nftw         tcgetattr    \_toupper  
  dup2        hsearch      nl\_langinfo  tcgetpgrp    twalk  
  fdopen       isascii      pclose       tcgetsid     tzset  
  \_\_filbuf     isatty       popen        tcsendbreak  \_xftw  
  fileno       isnan        putenv       tcsetattr      
  \_\_flsbuf     isnand \*\*    putw         tcsetpgrp      
  fmtmsg \*\*    lfind        setlabel     tdelete        
  
  \*\* = Function is at Level 2 in the SVID Issue 3 and therefore at  
       Level 2 in the ABI.  
  
包括上面同义（Synonyms）表列出的标号，对于<name> 入口已经存在的\_<name>   
形式（带一个下划线，上面没有列出来）优先权高于它们的名字。所以，例如，  
libc同时包含了getopt和\_getopt。  
  
在常规的上列中，其他地方以下没有被定义。  
  
int \_\_filbuf(FILE \*f);  
This function returns the next input character for f, filling  
its buffer as appropriate. It returns EOF if an error occurs.  
  
int \_\_flsbuf(int x, FILE \*f);  
This function flushes the output characters for f as if  
putc(x, f) had been called and then appends the value of x to  
the resulting output stream. It returns EOF if an error occurs  
and x otherwise.  
  
int \_xftw(int, char \*, int (\*)(char \*, struct stat \*, int), int);  
Calls to the ftw(BA\_LIB) function are mapped to this function  
when applications are compiled. This function is identical to  
ftw(BA\_LIB), except that \_xftw() takes an interposed first  
argument, which must have the value 2.  
  
  
要了解更多的关于SVID,ANSI C,POSIX的知识，可看该章节其他的库section部分。  
该节“System Data Interfaces”后有更多的描述。  
  
Global Data Symbols  
                    全局数据符号  
  
  
libc库需要一些外部的全局数据符号（为了它自己的常规工作而定义的）。  
所有向libsys库请求的数据符号一定要让libc提供，就象下面表中的数据符号。  
  
正式定义的数据object被他们的符号描述，看System V接口定义，第三版本  
或者第6章节的数据定义（Data Definitions）section（在适当的处理器  
补充到System V ABI）。  
  
在下面表中的入口有<name>-\_<name>的形式。一对符号都代表了一些数据。  
下划线的synonyms假设满足ANSI C标准。  
  
  
+ Figure 3-3: libc Contents, Global External Data Symbols  
  
  getdate\_err optarg  
  \_getdate\_err opterr  
  \_\_iob optind  
optopt