elf文件类型六 Dynamic Section（动态section）

来源: ChinaUnix博客 日期： 2008.01.28 15:16 (共有条评论) 我要评论

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

搜索

最新资讯 更多>>
· 谷歌劝说诺基亚采用Android操作..
· Apache 基金会确认退出 JCP 执..
· Chrome 10 新功能探秘：新增GP..
· 金山宣布开源其安全软件
· 女黑客在开源会议上抱受骚扰
· 21款值得关注的Linux游戏
· 马化腾：腾讯半年后彻底转型，..
· [多图] Chrome OS 预发布版本多..
· Lubuntu 11.04 默认应用抢先一览
· Red Hat宣布收购云计算软件提供..

论坛热点 更多>>
· do_execve时候用户栈中参数的..
· swapinfo -atm 问题
· Linux 的优点简述
· VM虚拟机上得Red Hat Linux上..
· 我看成了上海男人喜欢女人毛..
· 校车展览，看了你就知道
· 在遇到他之前，唯一需要做的..
· GRUB的疑问
· 从来没有人真正付足书价——..
· 云存储 vs 网盘

文档更新 更多>>
· orcale queue
· 谁可以推荐几本经典的操作系统的..
· 【北京】某物联网公司招云计算应..
· 【北京】某物联网公司招云计算应..
· 谁能推荐几本关于操作系统的书
· 如何添加网络接口eth1
· 葡萄牙语入门教材的选取与经验分享
· 葡萄牙语就业前景分析
· 葡萄牙语学习经验交流
· 山

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 > 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。

本文来自**ChinaUnix**博客，如果查看原文请
点：http://blog.chinaunix.net/u/22754/showart_472563.html