

**综合实训报告**

**ELF文件解析器的设计与实现**

|  |  |
| --- | --- |
| **姓名：** | **金海炳** |
| **学号：** | **2018317210116** |
| **班级：** | **计科1801** |
| **指导老师：** | **任继平 李小霞 王颖** |
|  |  |

**中国 武汉**

**二○二一年九月**

**2021.09**

目录

[1实训目的及内容 2](#_Toc82349584)

[1.1实训目的 2](#_Toc82349585)

[1.2实训内容 2](#_Toc82349586)

[2理论基础 3](#_Toc82349587)

[2.1 ELF格式简介 3](#_Toc82349588)

[2.2 ELF文件的两种视图 4](#_Toc82349589)

[2.3 ELF头 5](#_Toc82349590)

[2.4节区头 6](#_Toc82349591)

[2.4 程序头 9](#_Toc82349592)

[2.5 字符串表 10](#_Toc82349593)

[2.6 符号表 11](#_Toc82349594)

[2.7 重定位表 11](#_Toc82349595)

[2.8 动态链接 13](#_Toc82349596)

[3系统设计 15](#_Toc82349597)

[3.1体系结构设计 15](#_Toc82349598)

[3.2数据设计 15](#_Toc82349599)

[4系统实现与测试 17](#_Toc82349600)

[4.1系统实现 17](#_Toc82349601)

[4.2系统测试 26](#_Toc82349602)

[5结论与展望 35](#_Toc82349603)

[6参考资料 36](#_Toc82349604)

# 1实训目的及内容

## 1.1实训目的

ELF格式作为Linux操作系统平台的可执行文件格式，与开源的Linux系统一样，具有较多开放的学习和参考资源，是解析可执行程序格式很好的案例。通过解析可执行文件格式，有助于学生理解计算机语言与操作系统之间的二进制接口，其已构成系统性理解计算机体系的必要环节。

## 1.2实训内容

（1）分析ELF格式文档

本实训提供ELF格式文档的参考内容。结合计算机语言的编译，链接知识及操作系统对ELF格式的加载，运行过程，分析ELF格式文档，理解其文档内容，在分析过程中，可以充分利用网络资源，最终达到能正确理解ELF格式。

（2）分析readelf工具的使用手册

本实训提供需实现的readelf工具的命令选项及简单说明，结合ELF格式，正确理解readelf工具的命令选项。

（3）模拟实现readelf工具的源代码

根据对readelf工具的命令选项和ELF格式的理解，实现解析源代码。

（4）对实现过程和效果进行分析总结，撰写实训报告

对解析过程中的技术难点和采用的解决办法进行分析说明，并对每个命令选项达到的效果进行自我评价。撰写实训报告。

（5）参与实训答辩

每个学生结合源代码、演示运行和实训报告，参与答辩组组织的实训答辩。

# 2理论基础

## 2.1 ELF格式简介

可执行链接格式（Executable and Linking Format）最初是由 UNIX 系统实验室（UNIX System Laboratories，USL）开发并发布的，作为应用程序二进制接口（Application Binary Interface，ABI）的一部分。工具接口标准（Tool Interface Standards，TIS）委员会将还在发展的 ELF 标准选作为一种可移植的目标文件格式，可以在 32 位 Intel 体系结构上的很多操作系统中使用[1,2]。

ELF 标准的目的是为软件开发人员提供一组二进制接口定义，这些接口可以延伸到多种操作环境，从而减少重新编码、重新编译程序的需要。接口的内容包括目标模块格式、可执行文件格式以及调试记录信息与格式等[3]。

目标文件有三种类型：

（1）可重定位文件（Relocatable File）。包含适合于与其他目标文件链接来创建可执行文件或者共享目标文件的代码和数据。

（2）可执行文件（Executable File）。包含适合于执行的一个程序，此文件规定了exec()如何创建一个程序的进程映像。

（3）共享目标文件（Shared Object File）。包含可在两种上下文中链接的代码和数据。首先链接编辑器可以将它和其它可重定位文件和共享目标文件一起处理，生成另外一个目标文件。其次，动态链接器（Dynamic Linker）可能将它与某个可执行文件以及其它共享目标一起组合，创建进程映像。

目标文件全部是程序的二进制表示，目的是直接在某种处理器上直接执行。

表2-1 ELF 中常用数据格式

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 大小 | 对齐 | 目的 |
| Elf32\_Addr | 4 | 4 | 无符号程序地址 |
| Elf32\_Half | 2 | 2 | 无符号中等整数 |
| Elf32\_Off | 4 | 4 | 无符号文件偏移 |
| Elf32\_SWord | 4 | 4 | 有符号大整数 |
| Elf32\_Word | 4 | 4 | 无符号大整数 |

表2-1 续表

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 大小 | 对齐 | 目的 |
| unsigned char | 1 | 1 | 无符号小整数 |

目标文件中的所有数据结构都遵从“自然”大小和对齐规则。如果必要，数据结构可以包含显式的补齐，例如为了确保 4 字节对象按 4 字节边界对齐。数据对齐同样适用于文件内部。

ELF文件内部具有复杂的的数据结构，这些数据结构的定义都在<elf.h>文件中。其中大量的flag、type等采用宏定义，这些宏定义也都在<elf.h>文件中。因此，在解析ELF文件的时候，需要大量地写映射函数，将这些宏定义的数据信息以人类可读的方式显示出来。

## 2.2 ELF文件的两种视图

目标文件既要参与程序链接又要参与程序执行。出于方便性和效率考虑，目标文件格式提供了两种并行视图，分别反映了这些活动的不同需求。

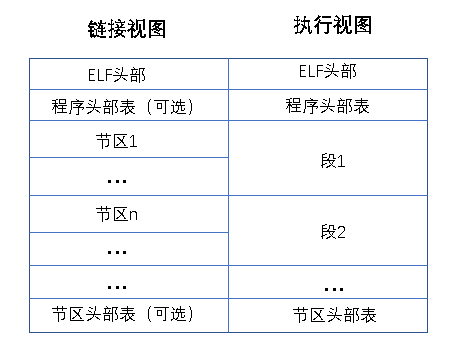


图 2-1 目标文件格式

文件开始处是一个 ELF 头部（ELF Header），用来描述整个文件的组织。节区部分包含链接视图的大量信息：指令、数据、符号表、重定位信息等等。

程序头部表（Program Header Table），如果存在的话，告诉系统如何创建进程映像。用来构造进程映像的目标文件必须具有程序头部表，可重定位文件不需要这个表。

节区头部表（Section Heade Table）包含了描述文件节区的信息，每个节区在表中都有一项，每一项给出诸如节区名称、节区大小这类信息。用于链接的目标文件必须包含节区头部表，其他目标文件可以有，也可以没有这个表。

## 2.3 ELF头

文件的最开始几个字节给出如何解释文件的提示信息。这些信息独立于处理器，也独立于文件中的其余内容。ELF Header 部分可以用以下数据结构表示：

typedef struct

{

unsigned char e\_ident[EI\_NIDENT]; /\* Magic number and other info \*/

Elf32\_Half e\_type; /\* Object file type \*/

Elf32\_Half e\_machine; /\* Architecture \*/

Elf32\_Word e\_version; /\* Object file version \*/

Elf32\_Addr e\_entry; /\* Entry point virtual address \*/

Elf32\_Off e\_phoff; /\* Program header table file offset \*/

Elf32\_Off e\_shoff; /\* Section header table file offset \*/

Elf32\_Word e\_flags; /\* Processor-specific flags \*/

Elf32\_Half e\_ehsize; /\* ELF header size in bytes \*/

Elf32\_Half e\_phentsize; /\* Program header table entry size \*/

Elf32\_Half e\_phnum; /\* Program header table entry count \*/

Elf32\_Half e\_shentsize; /\* Section header table entry size \*/

Elf32\_Half e\_shnum; /\* Section header table entry count \*/

Elf32\_Half e\_shstrndx; /\* Section header string table index \*/

} Elf32\_Ehdr; /\* ELF头结构体(32位) \*/

typedef struct

{

unsigned char e\_ident[EI\_NIDENT]; /\* Magic number and other info \*/

Elf64\_Half e\_type; /\* Object file type \*/

Elf64\_Half e\_machine; /\* Architecture \*/

Elf64\_Word e\_version; /\* Object file version \*/

Elf64\_Addr e\_entry; /\* Entry point virtual address \*/

Elf64\_Off e\_phoff; /\* Program header table file offset \*/

Elf64\_Off e\_shoff; /\* Section header table file offset \*/

Elf64\_Word e\_flags; /\* Processor-specific flags \*/

Elf64\_Half e\_ehsize; /\* ELF header size in bytes \*/

Elf64\_Half e\_phentsize; /\* Program header table entry size \*/

Elf64\_Half e\_phnum; /\* Program header table entry count \*/

Elf64\_Half e\_shentsize; /\* Section header table entry size \*/

Elf64\_Half e\_shnum; /\* Section header table entry count \*/

Elf64\_Half e\_shstrndx; /\* Section header string table index \*/

} Elf64\_Ehdr; /\* ELF头结构体(64位) \*/

其中，e\_ident 数组给出ELF的一些标识信息，e\_type是文件类型，e\_machine给出文件的目标体系结构类型，e\_version是目标文件版本，e\_entry是程序入口的虚拟地址，e\_phoff是程序头部表格（Program Header Table）的偏移量（按字节计算），e\_shoff是节区头部表格（Section Header Table）的偏移量（按字节计算），e\_flags保存与文件相关的，特定于处理器的标志，e\_ehsize是ELF 头部的大小，e\_phentsize是程序头部表格的表项大小，e\_phnum是程序头部表格的表项数目，e\_shentsize是节区头部表格的表项大小，e\_shnum是节区头部表格的表项数目，e\_shstrndx是节区头部表格中与节区名称字符串表相关的表项的索引。

通过e\_phoff可以访问程序头，通过e\_shoff可以访问节区头。通过e\_shoff和e\_shnum可以遍历节区头，通过e\_phoff和e\_phnum可以遍历程序头。通过e\_shstrndx可以访问shstr表。

## 2.4节区头

每个节区头部可以用以下数据结构描述：

typedef struct

{

Elf32\_Word sh\_name; /\* Section name (string tbl index) \*/

Elf32\_Word sh\_type; /\* Section type \*/

Elf32\_Word sh\_flags; /\* Section flags \*/

Elf32\_Addr sh\_addr; /\* Section virtual addr at execution \*/

Elf32\_Off sh\_offset; /\* Section file offset \*/

Elf32\_Word sh\_size; /\* Section size in bytes \*/

Elf32\_Word sh\_link; /\* Link to another section \*/

Elf32\_Word sh\_info; /\* Additional section information \*/

Elf32\_Word sh\_addralign; /\* Section alignment \*/

Elf32\_Word sh\_entsize; /\* Entry size if section holds table \*/

} Elf32\_Shdr; /\* 节区头结构体(32位) \*/

typedef struct

{

Elf64\_Word sh\_name; /\* Section name (string tbl index) \*/

Elf64\_Word sh\_type; /\* Section type \*/

Elf64\_Xword sh\_flags; /\* Section flags \*/

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

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

Elf64\_Xword sh\_size; /\* Section size in bytes \*/

Elf64\_Word sh\_link; /\* Link to 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; /\* 节区头结构体(64位) \*/

其中，sh\_name给出节区名称，是节区头部字符串表节区（Section Header String Table Section）的索引；sh\_type为节区的内容和语义进行分类；sh\_flags是1 位形式的标志；sh\_addr是节区的第一个字节应处的位置；sh\_offset值给出节区的第一个字节与文件头之间的偏移；sh\_size给出节区的长度（字节数）；sh\_link给出节区头部表索引链接；sh\_info给出附加信息；sh\_addralign是地址对齐约束；sh\_entsize包含固定大小的项目。

通过sh\_size和sh\_entsize可以遍历节区中的项目。

有一些特殊节区包含了包含了程序和控制信息，是elf文件中比较重要的部分，如表2-2所示：

表 2-2 常见特殊节区

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 属性 | 含义 |
| .bss | SHT\_NOBITS | SHF\_ALLOC +  SHF\_WRITE | 包含将出现在程序的内存映像中的为初始化数据。根据定义，当程序开始执行，系统将把这些数据初始化为 0。此节区不占用文件空间。 |
| .comment | SHT\_PROGBITS | (无) | 包含版本控制信息。 |
| .data | SHT\_PROGBITS | SHF\_ALLOC +  SHF\_WRITE | 这些节区包含初始化了的数据，将出现在程序的内存映像中。 |
| .data1 | SHT\_PROGBITS | SHF\_ALLOC +  SHF\_WRITE |
| .debug | SHT\_PROGBITS | (无) | 此节区包含用于符号调试的信息。 |

表 2-2 续表

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 属性 | 含义 |
| .dynamic | SHT\_DYNAMIC |  | 此节区包含动态链接信息。节区的属性将包含 SHF\_ALLOC 位。是否 SHF\_WRITE 位被设置取决于处理器。 |
| .dynstr | SHT\_STRTAB | SHF\_ALLOC | 此节区包含用于动态链接的字符串，大多数情况下这些字符串代表了与符号表项相关的名称。 |
| .dynsym | SHT\_DYNSYM | SHF\_ALLOC | 此节区包含了动态链接符号表。 |
| .fini | SHT\_PROGBITS | SHF\_ALLOC +  SHF\_EXECINSTR | 此节区包含了可执行的指令，是进程终止代码的一部分。程序正常退出时，系统将安排执行这里的代码。 |
| .got | SHT\_PROGBITS |  | 此节区包含全局偏移表。 |
| .hash | SHT\_PROGBITS | SHF\_ALLOC +  SHF\_EXECINSTR | 此节区包含了可执行指令，是进程初始化代码的一部分。当程序开始执行时，系统要在开始调用主程序入口之前（通常指 C 语言的 main 函数）执行这些代码。 |
| .interp | SHT\_PROGBITS |  | 此节区包含程序解释器的路径名。如果程序包含一个可加载的段，段中包含此节区，那么节区的属性将包含 SHF\_ALLOC 位，否则该位为 0。 |
| .line | SHT\_PROGBITS | (无) | 此节区包含符号调试的行号信息，其中描述了源程序与机器指令之间的对应关系。其内容是未定义的。 |
| .note | SHT\_NOTE | (无) | 此节区中包含注释信息，有独立的格式。 |
| .plt | SHT\_PROGBITS |  | 此节区包含过程链接表（procedure linkage table）。 |

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 属性 | 含义 |
| .relname | SHT\_REL |  | 这些节区中包含了重定位信息。如果文件中包含可加载的段，段中有重定位内容，节区的属性将包含 SHF\_ALLOC 位，否则该位置 0。传统上 name 根据重定位所适用的节区给定。例如 .text 节区的重定位节区名字将是：.rel.text 或者 .rela.text。 |
| .relaname | SHT\_RELA |  |
| .rodata | SHT\_PROGBITS | SHF\_ALLOC | 这些节区包含只读数据，这些数据通常参与进程映像的不可写段。 |
| .rodata1 | SHT\_PROGBITS | SHF\_ALLOC |
| .shstrtab | SHT\_STRTAB |  | 此节区包含节区名称。 |
| .strtab | SHT\_STRTAB |  | 此节区包含字符串，通常是代表与符号表项相关的名称。如果文件拥有一个可加载的段，段中包含符号串表，节区的属性将包含SHF\_ALLOC 位，否则该位为 0。 |
| .symtab | SHT\_SYMTAB |  | 此节区包含一个符号表。如果文件中包含一个可加载的段，并且该段中包含符号表，那么节区的属性中包含SHF\_ALLOC 位，否则该位置为 0。 |
| .text | SHT\_PROGBITS | SHF\_ALLOC +  SHF\_EXECINSTR | 此节区包含程序的可执行指令。 |

## 2.4 程序头

可执行文件或者共享目标文件的程序头部是一个结构数组，每个结构描述了一个段或者系统准备程序执行所必需的其它信息。目标文件的“段”包含一个或者多个“节区”，也就是“段内容（Segment Contents）”。程序头部仅对于可执行文件和共享目标文件有意义。

程序头部的数据结构如下：

typedef struct

{

Elf32\_Word p\_type; /\* Segment type \*/

Elf32\_Off p\_offset; /\* Segment file offset \*/

Elf32\_Addr p\_vaddr; /\* Segment virtual address \*/

Elf32\_Addr p\_paddr; /\* Segment physical address \*/

Elf32\_Word p\_filesz; /\* Segment size in file \*/

Elf32\_Word p\_memsz; /\* Segment size in memory \*/

Elf32\_Word p\_flags; /\* Segment flags \*/

Elf32\_Word p\_align; /\* Segment alignment \*/

} Elf32\_Phdr; /\* 程序头结构体(32位) \*/

typedef struct

{

Elf64\_Word p\_type; /\* Segment type \*/

Elf64\_Word p\_flags; /\* Segment 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 \*/

} Elf64\_Phdr; /\* 程序头结构体(64位) \*/

其中，p\_type描述的段的类型；p\_offset给出从文件头到该段第一个字节的偏移；p\_vaddr给出段的第一个字节将被放到内存中的虚拟地址；p\_paddr仅用于与物理地址相关的系统中；p\_filesz给出段在文件映像中所占的字节数；p\_memsz给出段在内存映像中占用的字节数；p\_flags给出与段相关的标志；p\_align给出段在文件中和内存中如何对齐。

## 2.5 字符串表

字符串表节区包含以 NULL（ASCII 码 0）结尾的字符序列，通常称为字符串。ELF目标文件通常使用字符串来表示符号和节区名称。对字符串的引用通常以字符串在字符串表中的下标给出。

一般，第一个字节（索引为 0）定义为一个空字符串。类似的，字符串表的最后一个字节也定义为 NULL，以确保所有的字符串都以 NULL 结尾。索引为 0 的字符串在不同的上下文中可以表示无名或者名字为 NULL 的字符串。

允许存在空的字符串表节区，其节区头部的 sh\_size 成员应该为 0。对空的字符串表而言，非 0 的索引值是非法的。

在ELF文件中一般存在两个字符串表.strtab和.dynstr。后者记录了动态段和动态符号表的字符串，前者记录了一般符号（函数名、变量名）的字符串。

## 2.6 符号表

目标文件的符号表中包含用来定位、重定位程序中符号定义和引用的信息。符号表索引是对此数组的索引。索引 0 表示表中的第一表项，同时也作为未定义符号的索引。

符号表的定义如下：

typedef struct

{

Elf32\_Word st\_name; /\* Symbol name (string tbl index) \*/

Elf32\_Addr st\_value; /\* Symbol value \*/

Elf32\_Word st\_size; /\* Symbol size \*/

unsigned char st\_info; /\* Symbol type and binding \*/

unsigned char st\_other; /\* Symbol visibility \*/

Elf32\_Section st\_shndx; /\* Section index \*/

} Elf32\_Sym; /\* 符号表项结构体(32位) \*/

typedef struct

{

Elf64\_Word st\_name; /\* Symbol name (string tbl index) \*/

unsigned char st\_info; /\* Symbol type and binding \*/

unsigned char st\_other; /\* Symbol visibility \*/

Elf64\_Section st\_shndx; /\* Section index \*/

Elf64\_Addr st\_value; /\* Symbol value \*/

Elf64\_Xword st\_size; /\* Symbol size \*/

} Elf64\_Sym; /\* 符号表项结构体(64位) \*/

其中，st\_name包含目标文件符号字符串表的索引，其中包含符号名的字符串表示；st\_value给出相关联的符号的取值；st\_size是符号具有相关的尺寸大小；st\_info给出符号的类型和绑定属性；st\_shndx给出相关的节区头部表索引。

## 2.7 重定位表

可重定位文件必须包含如何修改其节区内容的信息，从而允许可执行文件和共享目标文件保存进程的程序映像的正确信息。重定位表项就是这样一些数据。

重定位表项的格式如下：

/\* Relocation table entry without addend (in section of type SHT\_REL). \*/

typedef struct

{

Elf32\_Addr r\_offset; /\* Address \*/

Elf32\_Word r\_info; /\* Relocation type and symbol index \*/

} Elf32\_Rel; /\* 重定位表项Rel结构体(32位) \*/

/\* I have seen two different definitions of the Elf64\_Rel and

Elf64\_Rela structures, so we'll leave them out until Novell (or

whoever) gets their act together. \*/

/\* The following, at least, is used on Sparc v9, MIPS, and Alpha. \*/

typedef struct

{

Elf64\_Addr r\_offset; /\* Address \*/

Elf64\_Xword r\_info; /\* Relocation type and symbol index \*/

} Elf64\_Rel; /\* 重定位表项Rel结构体(64位) \*/

/\* Relocation table entry with addend (in section of type SHT\_RELA). \*/

typedef struct

{

Elf32\_Addr r\_offset; /\* Address \*/

Elf32\_Word r\_info; /\* Relocation type and symbol index \*/

Elf32\_Sword r\_addend; /\* Addend \*/

} Elf32\_Rela; /\* 重定位表项Rela结构体(32位) \*/

typedef struct

{

Elf64\_Addr r\_offset; /\* Address \*/

Elf64\_Xword r\_info; /\* Relocation type and symbol index \*/

Elf64\_Sxword r\_addend; /\* Addend \*/

} Elf64\_Rela; /\* 重定位表项Rela结构体(64位) \*/

其中，r\_offset给出了重定位动作所适用的位置；r\_info给出要进行重定位的符号表索引，以及将实施的重定位类型；r\_addend员给出一个常量补齐，用来计算将被填充到可重定位字段的数值。

如上所述，只有 Elf32\_Rela 项目可以明确包含补齐信息。类型为 Elf32\_Rel 的表项在将被修改的位置保存隐式的补齐信息。 依赖于处理器体系结构，各种形式都可能存在，甚至是必需的。因此，对特定机器的实现可以仅使用一种形式，也可以根据上下文使用不同的形式。

重定位节区会引用两个其它节区：符号表、要修改的节区。节区头部的 sh\_info 和sh\_link 成员给出这些关系。不同目标文件的重定位表项对 r\_offset 成员具有略微不同的解释。

1. 在可重定位文件中，r\_offset 中包含节区偏移。就是说重定位节区自身描述了如何修改文件中的其他节区；重定位偏移 指定了被修改节区中的一个存储单元。
2. 在可执行文件和共享的目标文件中，r\_offset 中包含一个虚拟地址。为了使得这些文件的重定位表项对动态链接器更为有用，节区偏移（针对文件的解释）让位于虚地址（针对内存的解释）。

尽管对 r\_offset 的解释会有少许不同，重定位类型的含义始终不变。

## 2.8 动态链接

实现程序加载和动态链接的主要技术有：

*  程序头部（Program Header）：描述与程序执行直接相关的目标文件结构信息。用来在文件中定位各个段的映像。同时包含其他一些用来为程序创建进程映像所必需的信息。
*  程序加载：给定一个目标文件，系统加载该文件到内存中，启动程序执行。
*  动态链接：系统加载了程序以后，必须通过解析构成进程的目标文件之间的符号引用，以便完整地构造进程映像。

如果一个目标文件参与动态链接，它的程序头部表将包含类型为 PT\_DYNAMIC 的元素。此“段”包含.dynamic 节区。该节区采用一个特殊符号\_DYNAMIC 来标记，其中包含如下结构的数组：

typedef struct

{

Elf32\_Sword d\_tag; /\* Dynamic entry type \*/

union

{

Elf32\_Word d\_val; /\* Integer value \*/

Elf32\_Addr d\_ptr; /\* Address value \*/

} d\_un;

} Elf32\_Dyn; /\* 动态链接表项结构体(32位) \*/

typedef struct

{

Elf64\_Sxword d\_tag; /\* Dynamic entry type \*/

union

{

Elf64\_Xword d\_val; /\* Integer value \*/

Elf64\_Addr d\_ptr; /\* Address value \*/

} d\_un;

} Elf64\_Dyn; /\* 动态链接表项结构体(64位) \*/

其中，d\_val 是此 Elf32\_Word 对象表示一个整数值，不同的值有不同的解释；d\_ptr 此 Elf32\_Addr 对象代表程序的虚拟地址。

在 exec() 期间，系统从PT\_INTERP 段中检索路径名，并从解释器文件的段创建初始的进程映像。也就是说，系统并不使用原来可执行文件的段映像，而是为解释器构造一个内存映像。接下来是解释器从系统接收控制，为应用程序提供执行环境。

在构造使用动态链接技术的可执行文件时，连接编辑器向可执行文件中添加一个类型为 PT\_INTERP 的程序头部元素，告诉系统要把动态链接器激活，作为程序解释器。系统所提供的动态链接器的位置是和处理器相关的。

Exec() 和动态链接器合作，为程序创建进程映像，其中包括以下动作：

(1). 将可执行文件的内存段添加到进程映像中；

(2). 把共享目标内存段添加到进程映像中；

(3). 为可执行文件和它的共享目标执行重定位操作；

(4). 关闭用来读入可执行文件的文件描述符，如果动态链接程序收到过这样的文件描述符的话；

(5). 将控制转交给程序，使得程序好像从 exec 直接得到控制。

# 3系统设计

## 3.1体系结构设计

本系统由4大模块组成：

（1） 参数解析模块。负责解析输入命令中的参数，并标识相应的掩码信息。

（2）错误处理模块。当命令输入不合法时，输出错误提示。

（3） ELF文件结构解析与定位模块。解析ELF文件头，定位ELF节区头、程序头、主要程序段、主要节区的地址信息，赋值相应的结构体指针。

（4）结果输出模块。根据参数解析模块解析出来的掩码信息，对相应的节区头、程序头、段、节区进行解析，进行宏定义的映射，输出结果。

如图3-1所示：

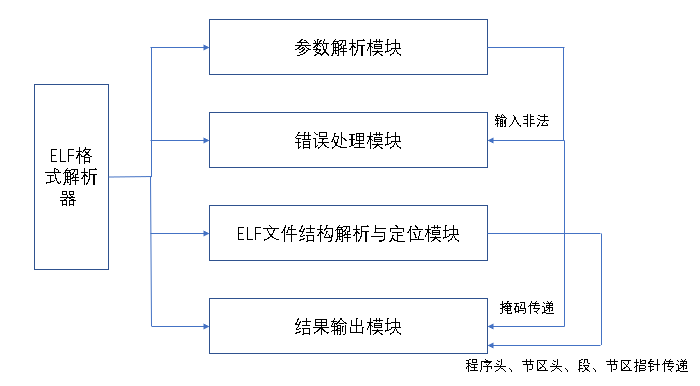


图3-1 ELF解析器体系结构

## 3.2数据设计

在include的头文件<elf.h>中，定义了ELF文件的数据结构。为了系统实现的高效和便捷，我在 “memory.h”头文件中定义了内存区封装的数据结构，如下：

typedef struct \_memory{

char \* ptr;//起始地址

int len;//内存大小(字节数量)

} mem\_t;

其中，ptr指向该段内存的起始地址，len描述了该段内存的大小。

在 ”myelf.h” 头文件中定义了一些数据结构和宏，如下：

1. 对64位的elf文件的封装。

typedef struct \_elf64{

mem\_t bytes; //elf文件的字节缓存区

struct stat \*stat; //elf文件的stat结构,包含文件信息

Elf64\_Ehdr \* ehdr;//elf文件头

Elf64\_Shdr \* shdr\_vec; //段表

Elf64\_Shdr \* shstrtab\_shdr; //shstrtab段描述符指针

Elf64\_Shdr \* strtab\_shdr; //strtab段描述符指针

Elf64\_Shdr \* dynstr\_shdr; //dynstr段描述符指针

Elf64\_Shdr \* sym\_shdr; //symtab段描述符指针

Elf64\_Shdr \* dyn\_sym\_shdr; //.dynsym段描述符指针,动态库文件有这种段

Elf64\_Shdr \* text\_shdr; //.text段描述符指针

Elf64\_Shdr \* dynamic\_shdr; //.dynamic段描述符指针

Elf64\_Shdr \*\* rel\_shdr\_vec;//重定位段表

int rel\_shdr\_vec\_len; //重定位段表的数量

mem\_t text; //.text代码段

mem\_t data; //.data数据段

mem\_t rodata; //.rodata段

Elf64\_Sym \* sym\_vec; //symtab段指针

Elf64\_Sym \* dyn\_sym\_vec; //.dynsym段指针

mem\_t shstrtab; //shstrtab段

mem\_t strtab; //strtab段

mem\_t dynstr; //dynstr段

Elf64\_Phdr \* phdr; //程序头

} elf64\_t;

1. 根据偏移获取地址、获取段名、获取变量名、获取段数、获取节区数的宏定义。

//获取在elf字节缓存区的指定偏移的字节值

#define value\_in\_elf\_offset(elf,offset) ((elf)->bytes.ptr[(offset)])

// 获取elf位数: 1: 32位, 2: 64位

#define elf\_bits(elf) value\_in\_elf\_offset(elf,4)

// 获取elf的大小端: 1 little endian

#define elf\_endian(elf) value\_in\_elf\_offset(elf,5)

// 获取elf文件的大小,也是字节缓存区的长度

#define elf\_size(elf) ((elf)->stat->st\_size)

// 获取指定偏移在elf字节缓存区的地址

#define elf\_bytes\_offset\_addr(elf,offset) ((elf)->bytes.ptr+(offset))

//通过偏移查找shstrtab字符表,段名在这个表中

#define get\_shstr\_in\_offset(elf,offset) ((elf)->shstrtab.ptr+(offset))

//通过偏移查找strtab字符表,函数名，变量名在这个表中

#define get\_str\_in\_offset(elf,offset) ((elf)->strtab.ptr+(offset))

//通过偏移查找dynstr字符表,函数名，变量名在这个表中

#define get\_dynstr\_in\_offset(elf,offset) ((elf)->dynstr.ptr+(offset))

//获取elf字节缓存区地址

#define elf\_bytes\_addr(elf) elf\_bytes\_offset\_addr((elf),0)

//获取节区表中的节区数量

#define elf\_shdr\_vec\_num(elf) ((elf)->ehdr->e\_shnum)

//获取程序头表中的段数量

#define elf\_phdr\_num(elf) ((elf)->ehdr->e\_phnum)

1. 打印相关信息的位掩码定义。

#define PRT\_SHDR 1<<0 //段表掩码

#define PRT\_SYM 1<<1 //符号表掩码

#define PRT\_XOUT 1<<2 //文件二进制掩码

#define PRT\_EHDR 1<<3 //ELF文件头掩码

#define PRT\_DYN 1<<4 //动态节区掩码

#define PRT\_REL 1<<8 //重定位段掩码

#define PRT\_PHDR 1<<9 //程序头掩码

#define PRT\_SDT 1<<10 //节区详细信息掩码

#define PRT\_SGRP 1<<11 //节组掩码

#define PRT\_NOTE 1<<12 //注释节区掩码

#define PRT\_UNWD 1<<13 //UNWIND段掩码

# 4系统实现与测试

## 4.1系统实现

系统整理流程图如图4-1所示。输入的命令首先传递给参数解析模块，参数解析模块将命令中的参数解析出来传给文件结构解析与指针定位模块，将掩码信息传递给输出模块，若参数不合法则将进入错误处理模块。输出模块根据指针定位和掩码信息解析具体的节、段，输出结果。

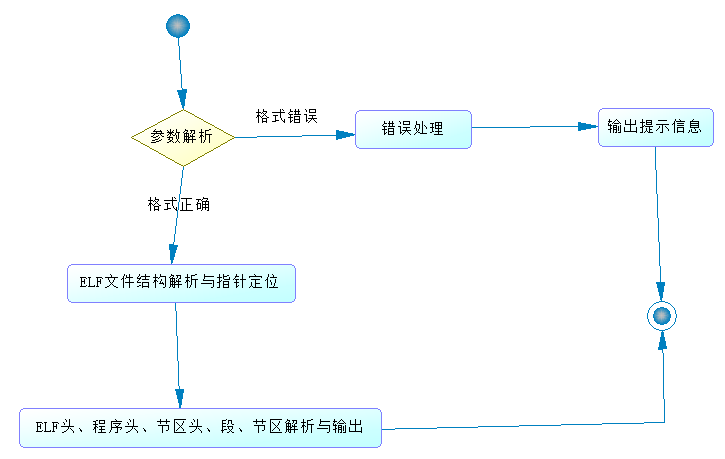


图4-1 系统流程图

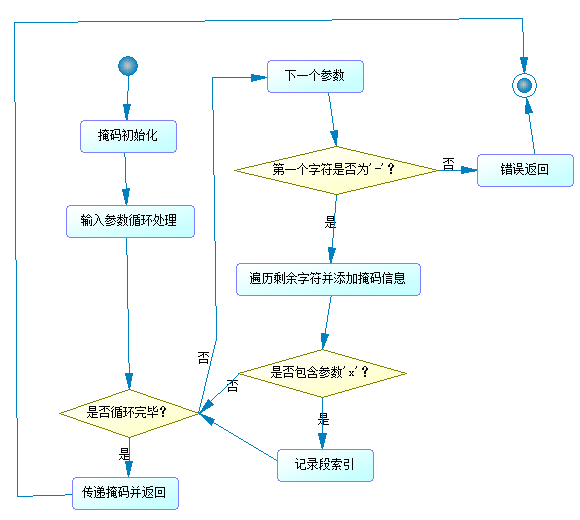


图4-2 参数解析流程图

参数解析是一个循环参数并进行判断的过程，根据不同的参数产生不同的掩码信息，如图4-2所示。

ELF文件结构解析与指针定位模块主要完成ELF头的解析和各种指针的定位，如图4-3所示。

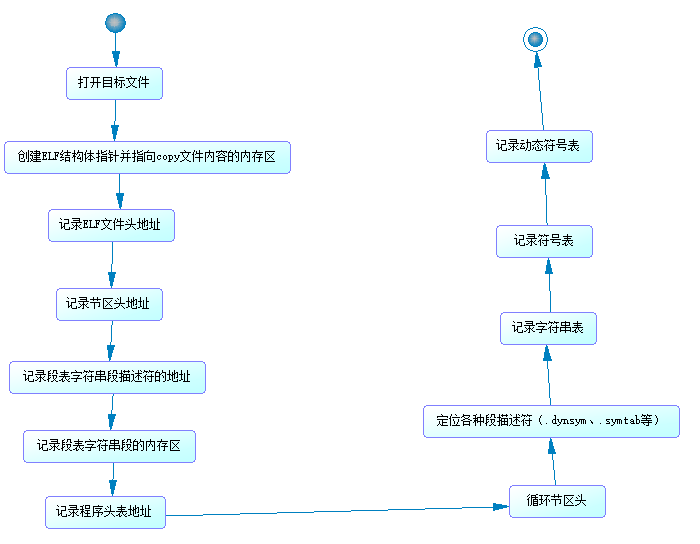


图4-3 ELF文件结构解析与指针定位流程图

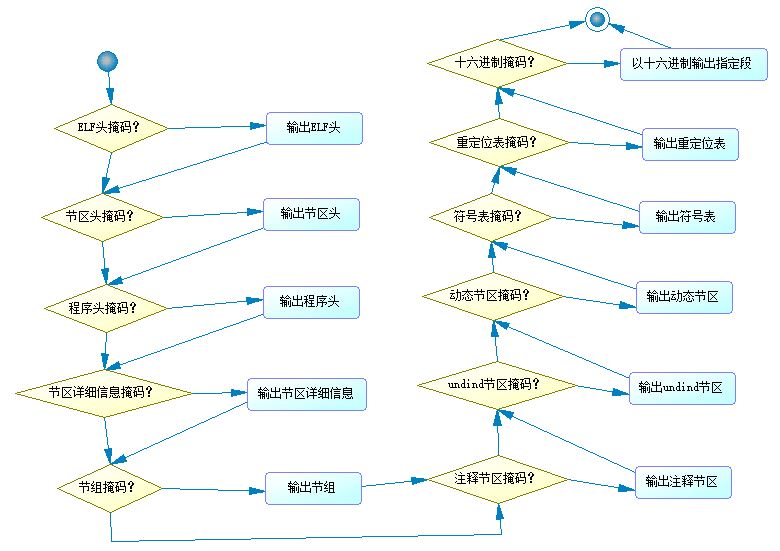


图4-4 结果输出流程图

结果输出模块是一个大模块，根据掩码信息，调用各个子模块进行具体的解析与输出，如图4-4所示。

ELF头的打印主要是对ELF结构体中的成员进行解析，其中有一些可以直接打印，如魔数、偏移地址、大小等，有一些需要根据<elf.h>头文件的宏定义进行映射，如类别、版本、数据格式、标志位等。如图4-5所示。

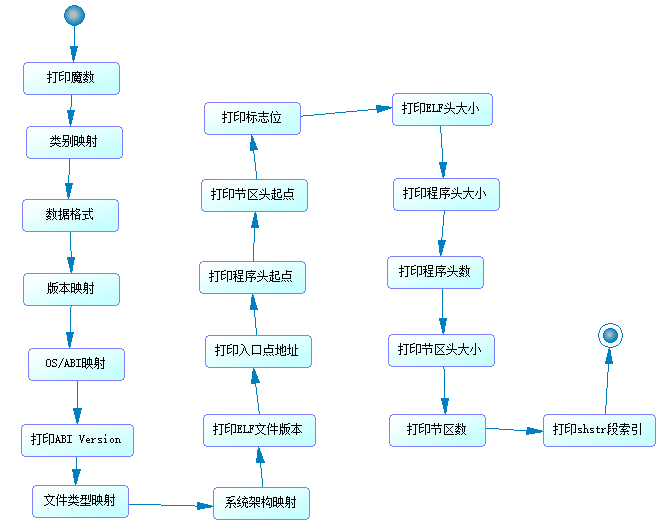


图4-5 ELF头打印流程图

节区头的解析方式与ELF头类似，如图4-6所示。

节区详细信息的打印，在节区头的基础上，需要对段名、flags等成员信息作具体解析，输出详细信息，如图4-7所示。

程序头的解析方式与节区头类似，如图4-8所示。

节组信息在打印的时候，首先需要遍历节区头，寻找目标文件中是否包含节组信息。若不包含，则输出提示信息，否则按照节区详细信息的格式输出节组信息，如图4-9所示。

注释段在打印的时候，也需要遍历节区头，确定注释段位置。指针定位后，需要解析注释类型，根据不同的注释类型解析成员值，进行映射，打印真实的注释信息，如图4-10所示。

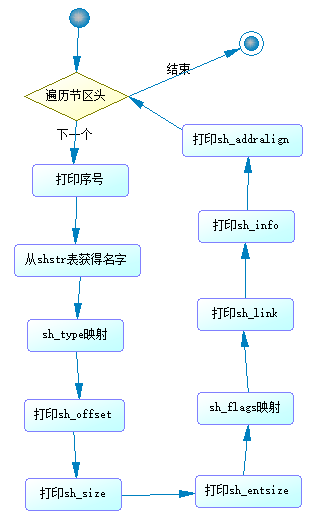


图4-6 节区头打印流程图

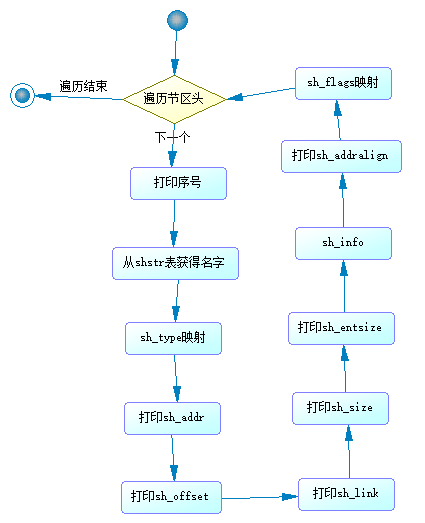


图4-7 节区详细信息打印流程图

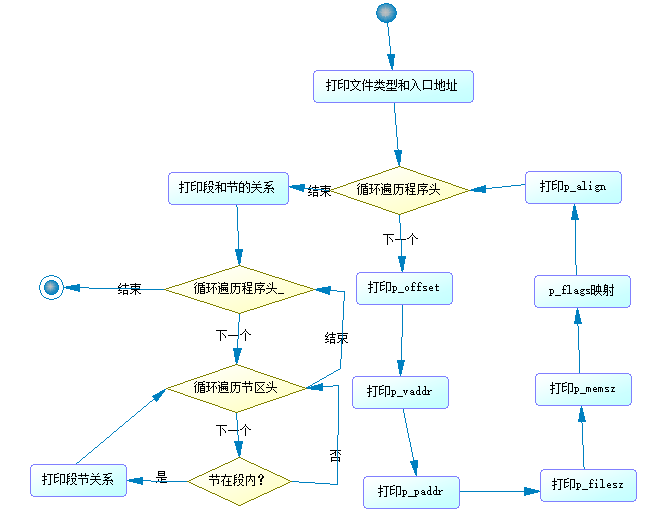


图4-8 程序头打印流程图

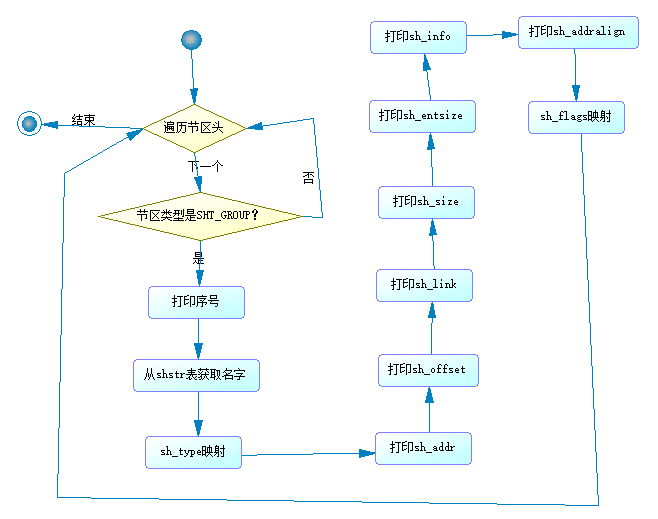


图4-9 节组打印流程图

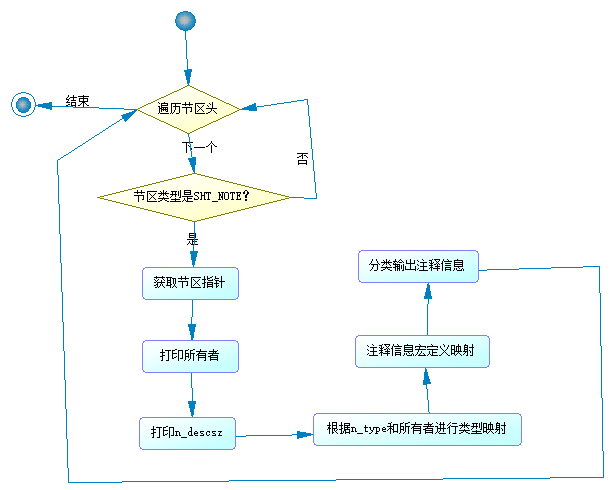


图4-10 注释段打印流程图

符号表在打印的时候，需要遍历符号表表项。从st\_info中提取类型信息进行映射，提取bind信息进行映射，映射的宏定义在标准头文件<elf.h>中。还要区分该符号表是否动态符号表，若是动态符号表则需要到dynstr表中去查找名字，否则到strtab表中去查找名字，如图4-11所示。

在本系统自定义的elf结构体中记录了重定位表的二级指针，根据这个二级指针可以定位重定位表的位置。从而遍历节区表项，打印相关信息，如图4-12所示。

unwind段的打印比较简单。目前的ELF文件unwind段只支持IA-64的机器。因此，首先要识别机器类型，若不是IA-64机器，则输出提示信息，否则打印unwind段信息，如图4-13所示。

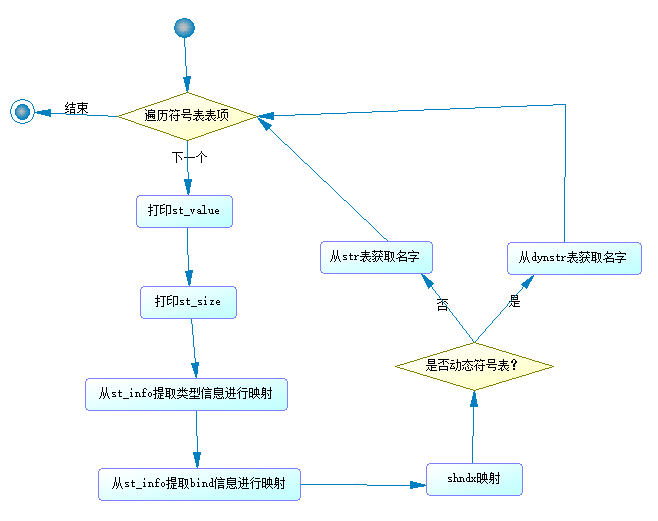


图4-11 符号表打印流程图

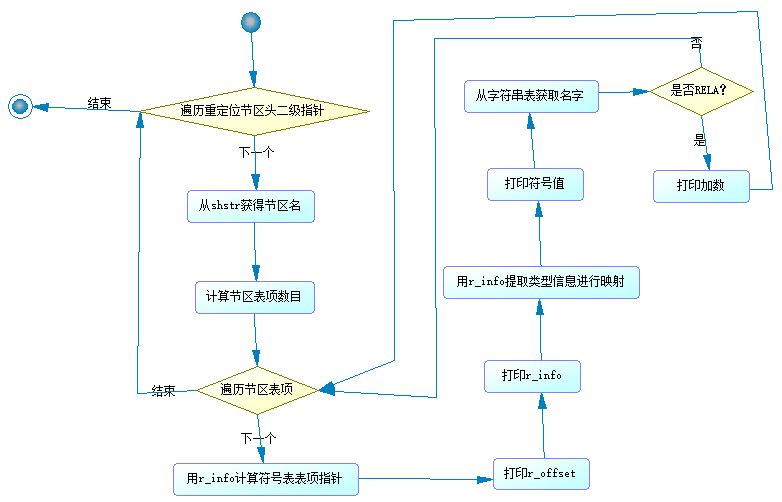


图4-12 重定位表打印流程图

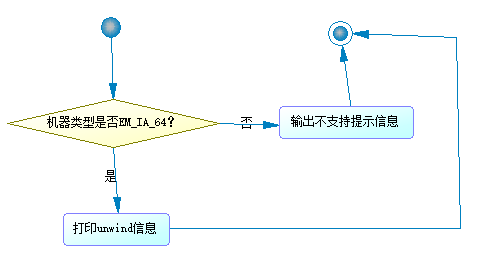


图4-13 unwind段打印流程图

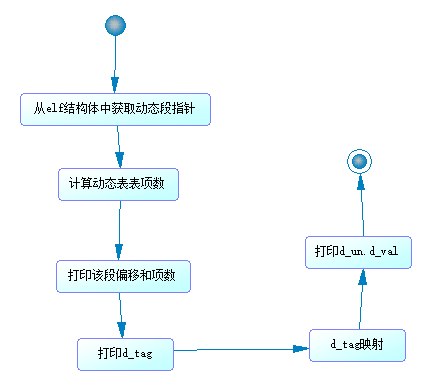


图4-14 动态表打印流程图

动态表在打印的时候需要对d\_tag作类型映射，给根据d\_tag的类型，对于d\_un有不同的解释，如图4-14所示。

以十六进制输出指定段。从elf结构体中获取指定段的指针，以16进制方式4字节对齐输出，同时在行末输出该行数据的char字符串，如图4-15所示。

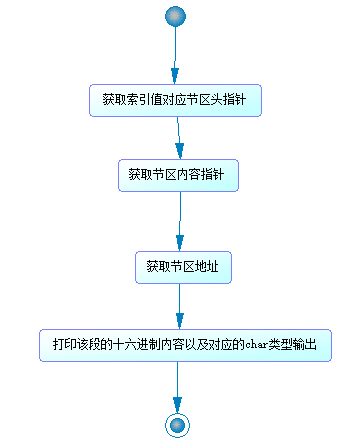


图4-15 以十六进制打印指定段流程图

## 4.2系统测试

（1）-h参数。

测试用例如表4-1所示：

表4-1 -h参数测试用例

|  |  |
| --- | --- |
| 测试用例编号 | ELF\_TEST\_001 |
| 测试项目 | ELF格式解析器 |
| 测试标题 | -h参数测试 |
| 预置条件 | 无 |
| 输入 | -h参数及目标文件 |
| 操作步骤 | 在shell中执行myelf <目标文件> -h |
| 预期输出 | elf文件头信息 |

测试结果如图4-16所示：



图4-16 -h参数测试结果

测试评价：结果与readelf一致。

（2）-l参数。

测试用例如表4-2所示：

表4-2 -l参数测试用例

|  |  |
| --- | --- |
| 测试用例编号 | ELF\_TEST\_002 |
| 测试项目 | ELF格式解析器 |
| 测试标题 | -l参数测试 |
| 预置条件 | 无 |
| 输入 | -l参数及目标文件 |
| 操作步骤 | 在shell中执行myelf <目标文件> -l |
| 预期输出 | Elf程序头信息(如果有数据的话) |

测试结果如图4-17所示：

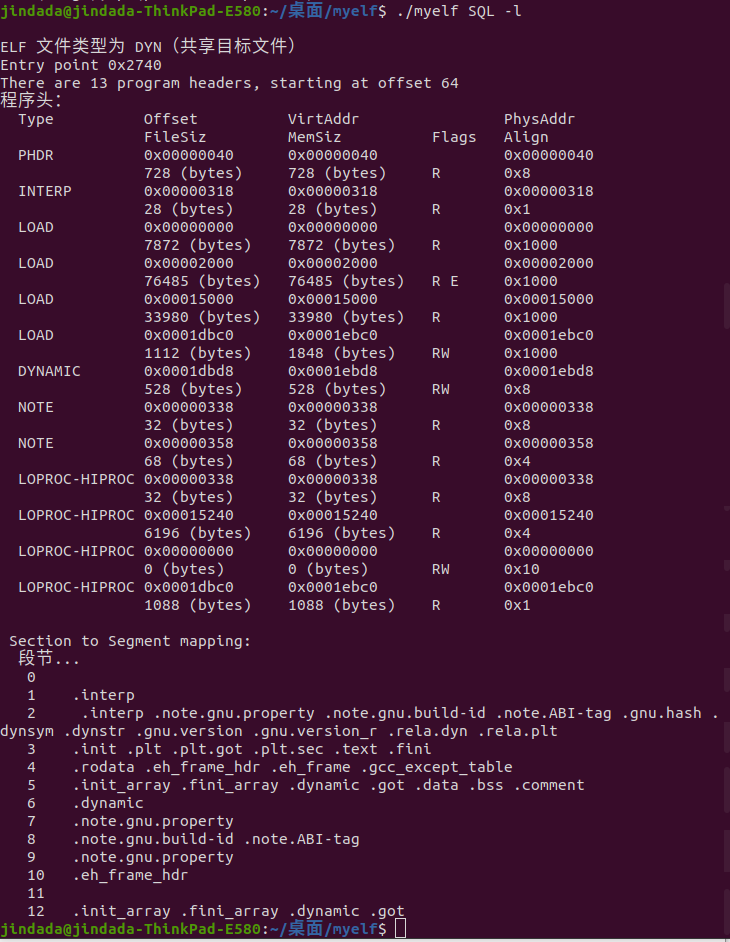


图4-17 -l参数测试结果

测试评价：结果与readelf一致。

（3）-S参数。

测试用例如表4-3所示：

表4-3 -S参数测试用例

|  |  |
| --- | --- |
| 测试用例编号 | ELF\_TEST\_003 |
| 测试项目 | ELF格式解析器 |
| 测试标题 | -S参数测试 |
| 预置条件 | 无 |
| 输入 | -S参数及目标文件 |
| 操作步骤 | 在shell中执行myelf <目标文件> -S |
| 预期输出 | Elf节头信息(如果有数据的话) |

测试结果如图4-18所示：

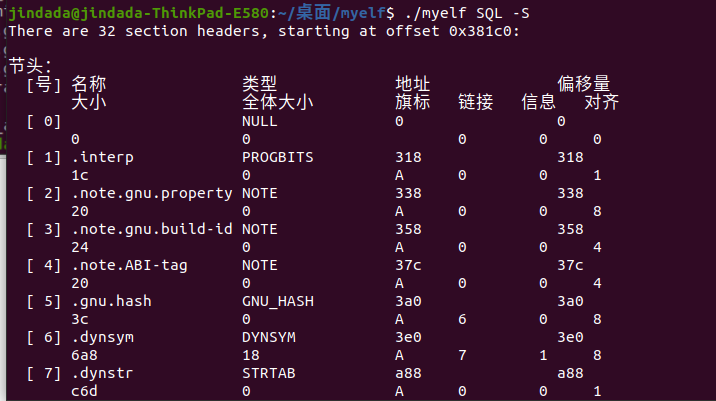


图4-18 -S参数测试结果

测试评价：结果与readelf一致。

（4）-g参数。

测试用例如表4-4所示：

表4-4 -g参数测试用例

|  |  |
| --- | --- |
| 测试用例编号 | ELF\_TEST\_004 |
| 测试项目 | ELF格式解析器 |
| 测试标题 | -g参数测试 |
| 预置条件 | 无 |
| 输入 | -g参数及目标文件 |
| 操作步骤 | 在shell中执行myelf <目标文件> -g |
| 预期输出 | 显示节组信息(如果有数据的话) |

测试结果如图4-19所示：

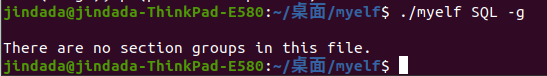


图4-19 -g参数测试结果

测试评价：结果与readelf一致。

（5）-t参数。

测试用例如表4-5所示：

表4-5 -t参数测试用例

|  |  |
| --- | --- |
| 测试用例编号 | ELF\_TEST\_005 |
| 测试项目 | ELF格式解析器 |
| 测试标题 | -t参数测试 |
| 预置条件 | 无 |
| 输入 | -t参数及目标文件 |
| 操作步骤 | 在shell中执行myelf <目标文件> -t |
| 预期输出 | 显示节的详细信息(-S的) |

测试结果如图4-20所示：



图4-20 -t参数测试结果

测试评价：结果与readelf一致。

（6）-s参数。

测试用例如表4-6所示：

表4-6 -s参数测试用例

|  |  |
| --- | --- |
| 测试用例编号 | ELF\_TEST\_006 |
| 测试项目 | ELF格式解析器 |
| 测试标题 | -s参数测试 |
| 预置条件 | 无 |
| 输入 | -s参数及目标文件 |
| 操作步骤 | 在shell中执行myelf <目标文件> -s |
| 预期输出 | 显示符号表段中的项（如果有数据的话） |

测试结果如图4-21所示：



图4-21 -s参数测试结果

测试评价：除Name属性外与readelf一致。readelf命令显示的Name中包含了以@开头的动态链接信息，这一部分信息不在符号表内，暂时不知道如何获取。

（7）-e参数。

测试用例如表4-7所示：

表4-7 -e参数测试用例

|  |  |
| --- | --- |
| 测试用例编号 | ELF\_TEST\_007 |
| 测试项目 | ELF格式解析器 |
| 测试标题 | -e参数测试 |
| 预置条件 | 无 |
| 输入 | -e参数及目标文件 |
| 操作步骤 | 在shell中执行myelf <目标文件> -e |
| 预期输出 | 显示全部头信息，等价于: -h -l -S |

测试结果如图4-22所示：



图4-22 -e参数测试结果

测试评价：结果与readelf一致。

（8）-n参数。

测试用例如表4-8所示：

表4-8 -n参数测试用例

|  |  |
| --- | --- |
| 测试用例编号 | ELF\_TEST\_008 |
| 测试项目 | ELF格式解析器 |
| 测试标题 | -n参数测试 |
| 预置条件 | 无 |
| 输入 | -n参数及目标文件 |
| 操作步骤 | 在shell中执行myelf <目标文件> -n |
| 预期输出 | 显示note段（内核注释）的信息 |

测试结果如图4-23所示：

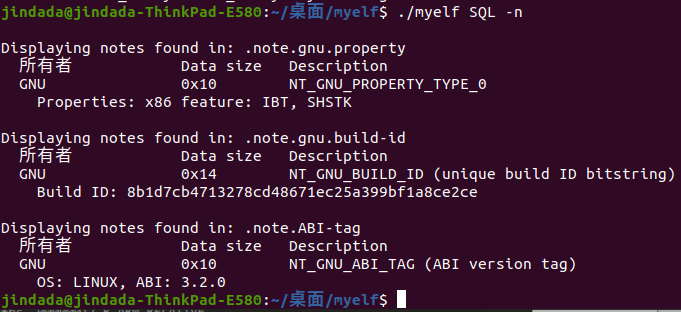


图4-23 –n参数测试结果

测试评价：结果与readelf一致。

（9）-r参数。

测试用例如表4-9所示：

表4-9 -r参数测试用例

|  |  |
| --- | --- |
| 测试用例编号 | ELF\_TEST\_009 |
| 测试项目 | ELF格式解析器 |
| 测试标题 | -r参数测试 |
| 预置条件 | 无 |
| 输入 | -r参数及目标文件 |
| 操作步骤 | 在shell中执行myelf <目标文件> -r |
| 预期输出 | 显示可重定位段的信息 |

测试结果如图4-24所示：



图4-24 -r参数测试结果

测试评价：除符号名称属性外与readelf一致。readelf命令显示的符号名称中包含了以@开头的动态链接信息，这一部分信息不在重定位表内，暂时不知道如何获取。

（10）-u参数。

测试用例如表4-10所示：

表4-10 -u参数测试用例

|  |  |
| --- | --- |
| 测试用例编号 | ELF\_TEST\_0010 |
| 测试项目 | ELF格式解析器 |
| 测试标题 | -u参数测试 |
| 预置条件 | 无 |
| 输入 | -u参数及目标文件 |
| 操作步骤 | 在shell中执行myelf <目标文件> -u |
| 预期输出 | 显示unwind段信息，当前只支持IA64 ELF的unwind段信息 |

测试结果如图4-25所示：

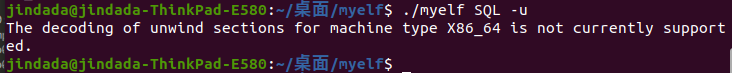


图4-25 -u参数测试结果

测试评价：结果与readelf一致。

（11）-d参数。

测试用例如表4-11所示：

表4-11 -d参数测试用例

|  |  |
| --- | --- |
| 测试用例编号 | ELF\_TEST\_0011 |
| 测试项目 | ELF格式解析器 |
| 测试标题 | -d参数测试 |
| 预置条件 | 无 |
| 输入 | -d参数及目标文件 |
| 操作步骤 | 在shell中执行myelf <目标文件> -d |
| 预期输出 | 显示动态段的信息 |

测试结果如图4-26所示：

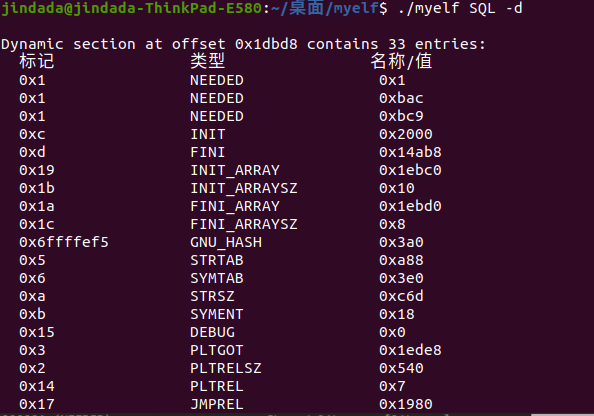


图4-26 -d参数测试结果

测试评价：标记和类型属性与readelf命令一直，但是结构体d\_un对应不同的类型，其值有不同的含义。由于类型过多，没有足够时间分类显示d\_un结构体的真正含义。而其中大多数类型的值代表的就是一个整数值，因此这里用十六进制整数值显示。

（12）-x参数。

测试用例如表4-12所示：

表4-12 -x参数测试用例

|  |  |
| --- | --- |
| 测试用例编号 | ELF\_TEST\_0012 |
| 测试项目 | ELF格式解析器 |
| 测试标题 | -x参数测试 |
| 预置条件 | 无 |
| 输入 | -x参数及目标文件 |
| 操作步骤 | 在shell中执行myelf <目标文件> -x <number> |
| 预期输出 | 以16进制方式显示指定段内内容 |

测试结果如图4-27所示：



图4-27 -x参数测试结果

测试评价：结果与readelf一致。

# 5结论与展望

本次实训我详细阅读了ELF格式文档，充分查阅网络资料，理解了理解ELF格式，掌握了readelf命令及主要参数，模拟readelf实现了一个ELF格式解析器，并且对各个参数进行了测试。

本次实训的重点在于理解ELF文件的格式，用C语言实现ELF文件解析器。而理解过程中的难点在于特殊节区及其作用的理解、动态链接的理解，实现过程中的难点在于段、节区的指针定位以及多种类结构体的理解与应用。其中，<elf.h>头文件是关键，<elf.h>头文件定义了ELF文件的所有数据结构以及宏定义。没有<elf.h>文件做参考，是无法写出宏定义映射的，从而也就无法解析出二进制背后真正的含义。

我实现的解析器模拟readelf工具，实现了其中的12个主要参数，分别是：

（1）-h(elf header)，显示elf文件开始的文件头信息。

（2）-l(program headers),segments 显示程序头（段头）信息(如果有数据的话)。

（3）-S(section headers),sections 显示节头信息(如果有数据的话)。

（4）-g(section groups),显示节组信息(如果有数据的话)。

（5）-t,section-details 显示节的详细信息(-S的)。

（6）-s,symbols 显示符号表段中的项（如果有数据的话）。

（7）-e,headers 显示全部头信息，等价于: -h -l -S 。

（8）-n,notes 显示note段（内核注释）的信息。

（9）-r,relocs 显示可重定位段的信息。

（10）-u,unwind 显示unwind段信息。

（11）-d,dynamic 显示动态段的信息。

（12）-x 以16进制方式显示指定段内内容。

其中，9个参数的显示做到了与readelf一致，3个参数有微小差别。

在实现本系统的过程中，我也发现了一些需要注意的地方：

1. 滕启明的《ELF 文件格式分析》有些过时了，该文中的数据结构定义是32位的，而现在的机器基本都是64位的。而且03年ELF文件格式中的许多保留位，如今已经有了新的含义。
2. dynsym符号表的符号名不在strtab表中，而在dynstr表中。
3. 解析NOTE段NT\_GNU\_BUILD\_ID时，需要按照单字节进行解析显示。
4. 重定位表中的符号在dynsym中，需要到dynstr中查找名字。
5. 动态段不同类型的dyn.d\_un代表的含义不同，数量巨大，实现起来有一些困难。

对于符号表的名字中关于动态链接的信息，以及对于动态段中不同类型d\_un代表的不同含义的解析，是本系统待完善的地方。另外，很多的宏定义在不同类型的机器上需要进行不同的映射，本系统在某些types、flags映射的时候只考虑了X86架构。因此，在本系统的基础上，可以完善现有参数在实现时的映射内容，以及拓展更多的参数。

# 6参考资料

[1] Tool Interface Standards(TIS) Committee, Portable Formats Specifi cation, Version 1.1.

[2] Tool Interface Standards(TIS) Committee, Executable and Linking Format(ELF) Specification, Version 1.2, May,1995.

[3] 滕启明. ELF 文件格式分析. PKU/SSDB-03-TN-005,2003.