demo简述

实现了一个简单的demo，输入被调用函数的虚拟地址，然后解析对应的elf文件，输出该虚拟地址对应的函数名称

- 基于32位elf进行实现，如果想改为64位也很容易
- 目前的测试程序中，地址是写死为t1的main函数对应的虚拟地址

运行效果：

```
orange0o0@LAPTOP-8TA12ITF:~/test$ ./elf_parser t1
预期是输出：main，实际输出：main
```

- 实际实现时，我们完全可以使用c标准库的elf.h，不用像源码中那样手搓对应的结构体（实际上是gpt帮我搓的哈哈哈）
- elf.h中还会提供很多相关的很有用的宏
- 后面的图片在md进行复制之后无法正常显示，详见同级目录下的README.pdf

elf学习

前言&总结

主要参考一份中文文档进行学习，中文pdf在同级目录下

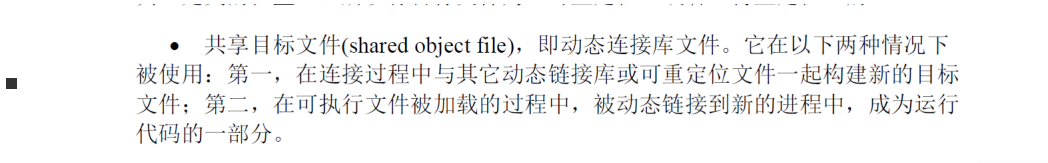
做这个demo的过程中深刻地体会到了对问题进行解耦的重要性，厘清思路对问题进行合理地拆分，对于整个项目的完成太重要了。

事实上，我认为最重要的步骤就在于厘清思路，并且拆分问题的这个过程。这项任务完成之后，后面无非就是不停的debug了。更何况还有gpt，当我能够把问题拆解成一个个小问题之后（以本demo为例，拆解成一个个小的函数，我只需要告诉gpt这个函数的预期输入和预期输出，gpt就可以很准确地给出c语言的实现了！）

管中窥豹，一个小demo如此，相信一个大的工程项目也是如此。我们首先要对整个问题进行解耦，并且尽量细化，这样，我们在整个项目的实现过程中才能始终做到心中有数。

ELF静态结构

- 目标文件/ELF文件可以分为三种类型（unix系统）
  - relocatable file
  - shared object file
    - 它会在两种情况下被使用，目前我对这里提到的两种的情况都不是完全掌握



- executable file

文件格式概述

有点难以理解：对于同一份二进制的elf文件，我们可以用不同的视角来看待。这里应该只是涉及到一个逻辑视角的问题

这个文档质量一般，主要是实例不多，概念太多，所以去找了其他资料进行学习。

> https://www.openeuler.org/zh/blog/lijiajie128/2020-11-03-ELF%E6%96%87%E4%BB%B6%E6%A0%BC%E5%BC%8F%E8%A7%A3%E6%9E%90.html

该链接的学习方式是直接上demo,动手跟着做一遍，理解确实更深。

看完之后，我们需要回答PA中的问题：

我们已经有了函数的起始地址，（通过解析jarl/jal指令得到），如何通过解析elf文件来获取地址和函数名称之间的映射关系呢?

demo

总体思路

首先，我们知道在所有section中，type为STRTAB的有4种，我们需要关注有两种，分别是strtbl和symtbl，前者存储索引与函数名称之间的映射关系，后者存储函数的虚拟地址与索引之间的关系。

下面，我们如何分别读取这两个section呢?

- 对于strtbl：

思路不难：以二进制方式读入elf文件，由于elf header固定在文件的起始位置，e_shstrndx字段的位置也是固定的，那么我们可以比较轻松地获得其section index，拿到了这个index之后，由于Section Header Table（我们可以使用header种的**e_shoff**来轻松访问它）本质上就是一个数组（数组元素为一个结构体），我们可以直接使用index找到strtbl的各种需要的信息（目前感觉一个offset字段就够了）

> 后记，你记错了，elf header的e_shstrndx字段记录的是shstrtbl的索引，而不是strtbl！这就意味着，和下面的symtbl一样，我们需要进行遍历整个Session header Table才可以！

> 在对于Section Header Table进行读取时，注意利用**e_shentsize**字段和**e_shnum**字段哦
>
> 这时，我也就体会到了我们为什么需要专门设置这两个字段了hhh

- 对于symtbl

由于elf header中并没有直接给出symtbl的在Section Header Table中的ndx，那么我们似乎必须要通过遍历的方式来找到它的具体位置。

具体来说，遍历整个Section Header Table数组，对于每个元素，我们获取它的name字段的值，根据该index获取实际的str，然后判断该str是否等于我们预期的.symtbl。

获取到了symtbl的offset之后，我们就可以对这个数组进行遍历了。

思路大概理清楚了，下面先用一个demo，并且就以该教程中提供的例子来进行实践。

我们希望实现一个函数，其输入是一个jarl/jal的虚拟地址，输出是对应的函数名的字符串。

在完成这个demo的过程中充分体会到了一点软件工程的思想：逐步解耦（厘清思路）、一个模块一个模块实现。螺旋上升

首先，我们需要调用elfInfoInit函数，把elf文件的内容读取到我们的全局变量中：

已经实现：

```
orange0o0@LAPTOP-8TA12ITF:~/test$ ./elf_parser t1
7f 45 4c 46 02 01 01 00
00 00 00 00 00 00 00 00
03 00 3e 00 01 00 00 00
60 10 00 00 00 00 00 00
40 00 00 00 00 00 00 00
e0 36 00 00 00 00 00 00
00 00 00 00 40 00 38 00
0d 00 40 00 1f 00 1e 00
orange0o0@LAPTOP-8TA12ITF:~/test$ hexdump -C t1
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  03 00 3e 00 01 00 00 00  60 10 00 00 00 00 00 00  |..>.....`.......|
00000020  40 00 00 00 00 00 00 00  e0 36 00 00 00 00 00 00  |@........6......|
00000030  00 00 00 00 40 00 38 00  0d 00 40 00 1f 00 1e 00  |....@.8...@.....|
00000040  06 00 00 00 04 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000050  40 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |@.......@.......|
00000060  d8 02 00 00 00 00 00 00  d8 02 00 00 00 00 00 00  |................|
00000070  08 00 00 00 00 00 00 00  03 00 00 00 04 00 00 00  |................|
00000080  18 03 00 00 00 00 00 00  18 03 00 00 00 00 00 00  |................|
```

下一步，使用我们获取session header table的相关信息：相关代码如下：

```c
//Elf32_Ehdr的定义是和数据的实际排布方式一致的，所以，按照手册，给出
    //完整的定义方式是一种更加高效的方式！
typedef struct {
    unsigned char e_ident[16]; /* ELF identification */
    __uint16_t e_type;         /* Object file type */
    __uint16_t e_machine;      /* Machine type */
    __uint32_t e_version;      /* Object file version */
    __uint32_t e_entry;        /* Entry point address */
    __uint32_t e_phoff;        /* Program header offset */
    __uint32_t e_shoff;        /* Section header offset */
    __uint32_t e_flags;        /* Processor-specific flags */
```

```
    __uint16_t e_ehsize;        /* ELF header size */
    __uint16_t e_phentsize;     /* Size of program header entry */
    __uint16_t e_phnum;         /* Number of program header
entries */
    __uint16_t e_shentsize;     /* Size of section header entry */
    __uint16_t e_shnum;         /* Number of section header
entries */
    __uint16_t e_shstrndx;      /* Section name string table index
*/
} Elf32_Ehdr;

typedef struct {
    __uint32_t sh_name;         /* Section name */
    __uint32_t sh_type;         /* Section type */
    __uint32_t sh_flags;        /* Section attributes */
    __uint32_t sh_addr;         /* Virtual address in memory */
    __uint32_t sh_offset;       /* Offset in file */
    __uint32_t sh_size;         /* Size of section */
    __uint32_t sh_link;         /* Link to related section */
    __uint32_t sh_info;         /* Miscellaneous information */
    __uint32_t sh_addralign;    /* Address alignment boundary */
    __uint32_t sh_entsize;      /* Size of entries, if section has
table */
} Elf32_Shdr;

typedef struct SHTbl{
    int e_shentsize;
    int e_shnum;
    __uint64_t e_shoff;
}SHTbl;
SHTbl sh_tbl;

    //获取section header table的相关信息，具体包括：e_shentsize、
e_shnum、e_shoff
bool get_sh_info(){
    Elf32_Ehdr* header = (Elf32_Ehdr*)elfInfo;

    if (header->e_shoff == 0 || header->e_shentsize == 0 ||
header->e_shnum == 0) {
        fprintf(stderr, "Invalid section header.\n");
        return false;
    }

    sh_tbl.e_shentsize = header->e_shentsize;
    sh_tbl.e_shnum = header->e_shnum;
    sh_tbl.e_shoff = header->e_shoff;

    return true;

```

```
55  }
```

但是debug发现，出现问题：

```
orange0o0@LAPTOP-8TA12ITF:~/test$ ./elf_parser t1
部分elf-header信息:
7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
03 00 3e 00 01 00 00 00 60 10 00 00 00 00 00 00
40 00 00 00 00 00 00 00 e0 36 00 00 00 00 00 00
00 00 00 00 40 00 38 00 0d 00 40 00 1f 00 1e 00
Invalid section header.
```

猜测是在执行下面这行程序时：

```
1  Elf32_Ehdr* header = (Elf32_Ehdr*)elfInfo;
```

ELFInfo和header在字节层面没有正确对应起来

事实证明这是多虑了，处理机会自动按照小端的规则帮我们处理：对于elfInfo的低地址的数据，自然也会赋值到header对应的属性的低地址字段，所以不是它的问题。

好吧，其实问题稍微有些隐晦：

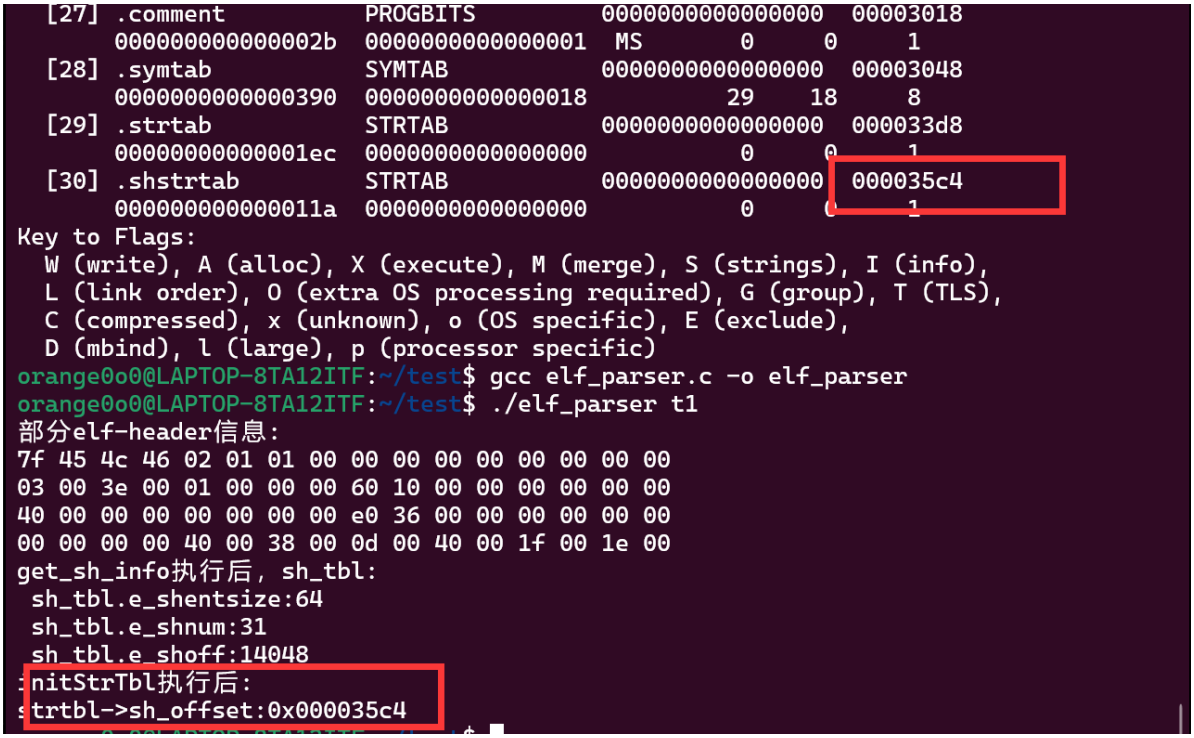
```
orange0o0@LAPTOP-8TA12ITF:~/test$ readelf -h t1
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
```

我当前的elf文件是64位的，但是我的结构体是按照32位进行设计的！

果然，修改结构体设计之后，发现是正常进行了解析！好好好

```
orange0o0@LAPTOP-8TA12ITF:~/test$ ./elf_parser t1
部分elf-header信息:
7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
03 00 3e 00 01 00 00 00 60 10 00 00 00 00 00 00
40 00 00 00 00 00 00 00 e0 36 00 00 00 00 00 00
00 00 00 00 40 00 38 00 0d 00 40 00 1f 00 1e 00
get_sh_info执行后，sh_tbl:
 sh_tbl.e_shentsize:64
 sh_tbl.e_shnum:31
 sh_tbl.e_shoff:14048
orange0o0@LAPTOP-8TA12ITF:~/test$ readelf -h t1
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1060
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14048 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
orange0o0@LAPTOP-8TA12ITF:~/test$
```

下一步，我们初始化shstrtbl的相关信息



实践之后才想起来，我们这里的顺序应该是先获得.shstrtbl这个section的相关信息（你问我为什么？因为手册是这样写的）

> 你会发现一个问题，由于一开始对任务的解耦不够彻底，具体来说，到了demo的中期还是会觉得有点迷茫，就，不知道当前在做的事情在整个demo的实现中有何作

下面首先实现一个函数：findByName

函数源码如下：

```
Elf64_Shdr* findByName(char* name) {
    // 获取字符串表的起始地址
    char* strTab = (char*)&elfInfo[shstrtbl->sh_offset];
    // 遍历所有 section headers
    for(int i = 0; i < sh_tbl.e_shnum; i++) {
        Elf64_Shdr* shdr = (Elf64_Shdr*)&elfInfo[sh_tbl.e_shoff + i * sh_tbl.e_shentsize];
        char* secName = &strTab[shdr->sh_name];

        // 比较 section 的名字和输入的名字
        if(strcmp(secName, name) == 0) {
            return shdr;
        }
    }
    // 如果没有找到匹配的 section，返回 NULL
    return NULL;
}
```

> 输入：已知的session name，输出该session的相关信息(返回类型为Elf64_Shdr*)

```
  [27] .comment          PROGBITS         0000000000000000  00003018
       000000000000002b  0000000000000001  MS       0     0     1
  [28] .symtab           SYMTAB           0000000000000000  00003048
       0000000000000390  0000000000000018          29    18     8
  [29] .strtab           STRTAB           0000000000000000  000033d8
       00000000000001ec  0000000000000000           0     0     1
  [30] .shstrtab         STRTAB           0000000000000000  000035c4
       000000000000011a  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), l (large), p (processor specific)
orange0o0@LAPTOP-8TA12ITF:~/test$ ./elf_parser t1
部分elf-header信息:
7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
03 00 3e 00 01 00 00 00 60 10 00 00 00 00 00 00
40 00 00 00 00 00 00 00 e0 36 00 00 00 00 00 00
00 00 00 00 40 00 38 00 0d 00 40 00 1f 00 1e 00
get_sh_info执行后，sh_tbl:
 sh_tbl.e_shentsize:64
 sh_tbl.e_shnum:31
 sh_tbl.e_shoff:14048
initShStrTbl执行后:
shstrtbl->sh_offset:0x000035c4
strtbl->sh_offset:0x000033d8
orange0o0@LAPTOP-8TA12ITF:~/test$
```

在有了findByName这个强大的工具之后，我们就可以轻松地获得strtab、symtab的相关信息了！

下面，我们似乎已经不知不觉间进入收尾工作了！我们整个demo的输入，是一个函数的逻辑地址，我们只需要遍历symtab这个section，获取到它的name字段，然后再利用strtab这个section，获取到实际的字符串，就可以完成整个demo了！

最后，我们给出getFunName的实现：

```
char* getFunName(__uint64_t vaddr) {
    // 获取字符串表的起始地址
    char* strTab = (char*)&elfInfo[strtbl->sh_offset];

    // 获取符号表的起始地址
    Elf64_Sym* symTab = (Elf64_Sym*)&elfInfo[symtbl->sh_offset];

    // 计算符号表中的符号数量
    int numSymbols = symtbl->sh_size / symtbl->sh_entsize;

    // 遍历符号表
    for(int i = 0; i < numSymbols; i++) {
        Elf64_Sym* symbol = &symTab[i];

        // 比较符号的虚拟地址和输入的虚拟地址
        if(symbol->st_value == vaddr) {
            // 如果找到匹配的符号，返回符号的名字
            return &strTab[symbol->st_name];
        }
    }

    // 如果没有找到匹配的符号，返回 NULL
    return NULL;
}
```

由此，进一步加深了我们对于section的认知，读取不同的section，我们所需要的读取方式都不一样，就以上面的两个框框之处的section为例：

strTab，我们只需要用一个char*来记录一下即可，实际使用时，就是像遍历中的那样，strTab[symbol->st_name]即可访问这个section中的字符串，但是，symTab就不一样了，我们甚至需要为这个section单独写一个结构体来存储其对应的信息。

> 写到这里，我有底气说，给我足够的时间、完整的手册，我也可以实现一个完整的`readelf`工具！👀👀👀👀🤓🤓🤓🤓

继续使用t1进行测试，结果很棒，圆满完成！