

**综合实训报告**

**ELF文件解析器的设计与实现**

|  |  |
| --- | --- |
| **姓名：** | **詹少雄** |
| **学号：** | **2019308210202** |
| **班级：** | **计科2003** |
| **指导老师：** | **任继平 李小霞** |
|  |  |

**中国 武汉**

**二○二三年七月**

**2023.07**

目录

[1实训目的及内容 2](#_Toc138581242)

[1.1实训目的 2](#_Toc138581243)

[1.2实训内容 2](#_Toc138581244)

[1.3实训安排 4](#_Toc138581245)

[2理论基础 6](#_Toc138581246)

[2.1 ELF文件概述 6](#_Toc138581247)

[2.2 ELF文件作用 6](#_Toc138581248)

[2.3 ELF文件解析选项 9](#_Toc138581249)

[2.4 ELF选项-h的基础理论 9](#_Toc138581250)

[2.5 ELF选项-S的基础理论 11](#_Toc138581251)

[2.6 ELF选项-l的基础理论 17](#_Toc138581252)

[2.6 ELF选项-d的基础理论 19](#_Toc138581253)

[3系统设计 19](#_Toc138581254)

[3.1 -h选项文件头表的数据结构设计 19](#_Toc138581255)

[3.2 -S选项节区头表的数据结构设计 20](#_Toc138581256)

[3.3 -l选项程序头表的数据结构设计 21](#_Toc138581257)

[3.4 -d选项动态链接表的数据结构设计 22](#_Toc138581258)

[4系统实现 23](#_Toc138581259)

[4.1 -h选项文件头表的程序设计 23](#_Toc138581260)

[4.2 -S选项节区头表的程序设计 26](#_Toc138581261)

[4.3 -l选项程序头表的程序设计 29](#_Toc138581262)

[4.4 -d选项动态链接的程序设计 31](#_Toc138581263)

[5系统测试 31](#_Toc138581264)

[5.1测试用例设计 31](#_Toc138581265)

[5.2测试结果及分析 32](#_Toc138581266)

[6结论与体会 39](#_Toc138581267)

[6.1结论 39](#_Toc138581268)

[6.2个人体会 40](#_Toc138581269)

[7参考文献 40](#_Toc138581270)

# 1实训目的及内容

## 1.1实训目的

本次综合实训围绕构建ELF文件解析器进行展开，通过分析ELF文件结构来以及相关的基础知识来构建ELF文件解析器。

通过本实践课程的学习，要求学生能够针对可执行文件解析较为复杂的计算机相关工程问题，对可执行文件格式涉及的问题进行抽象、对可执行文件格式系统建模，并能根据实验方案搭建系统开发环境，以具体可执行文件为实验数据，能正确解析其格式，优化解析方案。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。为学生今后从事计算机工程设计和软硬件开发及科研活动、生产实践打下扎实的良好的实践基础。增强学生的工程实践能力、系统能力。拟达成实训目标有：

实训目标1：（知识目标）对ELF文件格式进行分析，对源语言和硬件平台的知识进行融会贯通，对支持软件和硬件方案进行比较分析，针对具体应用选择实现方案；

实训目标2：（能力目标）熟练基于C/C++语言编程工具及其他辅助工具的能力，合理搭建开发环境、系统研发与部署、能对真实的可执行文件格式数据进行正确分析；

实训目标3：（素质目标）采用三人一组，让每个学生在团队中承担个体、团队成员的角色，独立或合作开展工作，体现团队合作精神；

实训目标4：（素质目标）针对其相关基础知识，设计方案、研究方法、技术路线等问题与他人进行有效沟通和交流，对实施过程和结果，以口头答辩和实训报告的方式准确表达观点；

## 1.2实训内容

|  |  |  |  |
| --- | --- | --- | --- |
| **任务名称** | **参考学时** | **实训内容** | **实训要求** |
| 第一章 问题定位与分析 | 4 | （1）阅读本实训的指导书，了解实训目的，要求和内容  （2）阅读ELF文件格式分析和程序员的自我修养，结合源语言和硬件平台理解ELF主要成分的含义 | 1.理解ELF文件格式内容；  2.结合指导书和ELF文件格式进行项目需求分析；  3.小组项目分工； |
| 第二章 实验方案设计和系统建模 | 12 | （1）系统概要设计  （2）系统详细设计  （3）关键算法选择和设计 | 1.进行系统概要设计建模过程；  2.进行系统详细设计建模过程；  3.对问题解决所需要的核心算法进行分析和选择  4.小组成员通过论证，迭代修改确定最终的实验方案； |
| 第三章 系统开发 | 24 | （1）项目系统开发  （2）项目集成 | 1.熟练掌握和应用C/C++编程语言进行系统的研发；  2.使用核心算法实现问题的解决；  3.熟练运行集成工作进行项目集成； |
| 第四章 实验数据获取与分析 | 12 | （1）对ELF格式文件进行系统测试  （2）对任意的ELF格式文件作为实验数据进行分析  （3）项目报告撰写 | 1.选择合适的系统测试方法；  2.针对不同的ELF格式形式，如库文件，目标格式文件，可执行文件作为测试数据进行测试；  3.项目完善并提出优化方案；  4.小组合作进行项目报告撰写。 |
| 第五章 项目演示与答辩 | 4 | （1）提交设计报告和源程序  （2）小组答辩 | 1.公开答辩和成果展示；  2.根据答辩意见进行项目修改完善。 |

该项目包括五个主要任务。第一章是问题定位与分析，我们需要阅读指导书和相关资料，理解ELF文件格式并分析项目需求，进行小组分工。第二章是实验方案设计和系统建模，我们需要进行系统概要设计和详细设计，并选择和设计关键算法。第三章是系统开发，我们需要使用C/C++编程语言进行系统研发，并进行项目集成。第四章是实验数据获取与分析，我们需要对ELF格式文件进行系统测试和分析，撰写项目报告。第五章是项目演示与答辩，我们需要提交设计报告和源程序，并进行公开答辩和成果展示。本项目涵盖了从问题分析到系统开发、测试、报告撰写和最终演示的全过程，我们需要在小组合作中完成各项任务。

## 1.3实训安排

##### 1.3.1 任务分工

本次实训我们三个人一组共同实现了ELF文件解析器的功能，我们小组的分工如下：

* 詹少雄（计科2003）

负责统筹团队任务，安排实训方案与讲解，并成功完成了-h、-S、-l和-d的ELF解析选项。在项目中展示了出色的组织能力和解析技术，为团队提供了方向和指导。

* 李绅（计科2004）

负责实训方案的实行与讲解，并顺利完成了-n、-e、-r和-d的ELF解析选项。展现了良好的实施能力和解析技术，为团队提供了有关ELF文件的重要信息。

* 杜鹏（计科2003）

负责实训方案的实行与讲解，并圆满完成了-s、-t、-g和-d的ELF解析选项。在项目中展现了优秀的编程能力和解析技术，为团队提供了强大的功能。

通过我们小组成员的紧密合作和各自的分工，我们成功地开发出了一个功能完善的ELF文件解析器。每个人在项目中发挥了自己的优势，共同推动了项目的进展。我们共同面对挑战，克服了遇到的困难，并取得了显著的成果。

##### 1.3.2 任务合作Git流

在本次项目中，我们采用了Github工作流作为我们的团队合作平台。Github是一个广泛使用的代码托管和协作平台，它提供了一系列功能和工具，使得团队成员可以方便地协同工作和管理项目。我们将项目代码托管到Github上。通过创建一个项目仓库，我们可以将我们的代码统一存放在一个地方，并与团队成员共享。这使得团队成员可以轻松地访问、查看和编辑代码。

Github提供了强大的版本控制功能。我们可以使用Git作为代码版本控制工具，记录和管理我们的代码修改历史。这使得团队成员可以轻松地查看和比较不同版本的代码，以及回滚到之前的版本。通过版本控制，我们可以更好地跟踪项目的进展，并确保代码的稳定性和一致性。Github还提供了问题跟踪和任务管理功能。我们可以使用Github的Issue功能来记录和分配项目中的问题和任务。团队成员可以通过创建Issue、分配责任人、添加标签和评论等方式，对问题和任务进行跟踪和讨论。这使得团队成员可以更加高效地协同工作，明确各自的工作职责，并及时解决项目中的问题。Github还支持代码审核和合并功能。团队成员可以通过创建Pull Request来将自己的代码提交给项目的主要负责人进行审核。主要负责人可以对代码进行评论、提出修改建议，并最终决定是否合并到主分支中。这种代码审核和合并机制有助于保证代码的质量和一致性，以及团队成员之间的知识分享和技术交流。

通过将项目打包开源并发布到Github网站，我们不仅可以方便地进行团队合作，还可以向其他开发者和用户展示我们的工作成果。其他开发者可以查看我们的代码、提出建议和改进意见，从而促进项目的进一步发展和改善。

我们的项目已开源至：<https://github.com/Jasaxion/LinuxELF_FileParse>

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图 1 Github开源社区图

# 2理论基础

## 2.1 ELF文件概述

ELF文件格式是一种常见的可执行文件和目标文件格式。

它可以细分为三类：

1. 可重定位文件（Relocatable Files）：包含了链接器所需的代码和数据信息。链接器可以根据这些信息将多个可重定位文件通过链接过程生成可执行文件或共享库。
2. 可执行文件（Executable Files）：包含了最终需要执行的程序。可执行文件中的信息用于指导操作系统生成进程映像，从而执行程序。
3. 共享库文件（Shared Object Files）：共享库中的代码和数据具有两个主要用途。首先，静态链接器（如GNU ld）可以利用共享库和可重定位文件生成最终的目标文件，无论是可执行文件还是新的共享库。其次，动态链接器（如GNU ld-linux.so）可以利用共享库和可执行文件生成最终的进程映像。

经过编译器和链接器的处理，目标文件（可执行文件或共享库）被转化为处理器可识别的二进制指令序列，这些指令描述了程序的行为。需要注意的是，有些程序的执行需要依靠解释器的帮助。

## 2.2 ELF文件作用

ELF文件在程序的连接和执行过程中扮演着重要的角色，因此可以从不同的角度来理解ELF格式的文件：

* 如果用于编译和链接（可重定位文件），编译器和链接器将把ELF文件视为节头表描述的节的集合，其中每个节都包含了程序中的特定信息，如代码、数据和符号表等。可重定位文件的主要目的是提供链接器所需的代码和数据信息，以便生成最终的可执行文件或共享库。此时，ELF文件中的程序头表是可选的，因为它主要用于加载执行阶段。
* 如果用于加载执行（可执行文件），加载器将把ELF文件看作是程序头表描述的段的集合。一个段可能包含多个节，这些节包含了在程序执行过程中需要的代码、数据和其他资源。程序头表描述了这些段在内存中的加载位置、大小和访问权限等信息，以便操作系统正确地将程序映射到内存并执行。与可重定位文件不同，此时ELF文件中的节头表是可选的，因为加载执行阶段更关注段的组织和映射。
* 如果是共享文件（共享库），ELF文件既包含了用于编译和链接的节头表，也包含了用于加载执行的程序头表。共享库既可以作为编译和链接的目标，也可以在程序加载和执行时被动态链接到内存中。

图2和图3展现了两种视图下的模块形式：

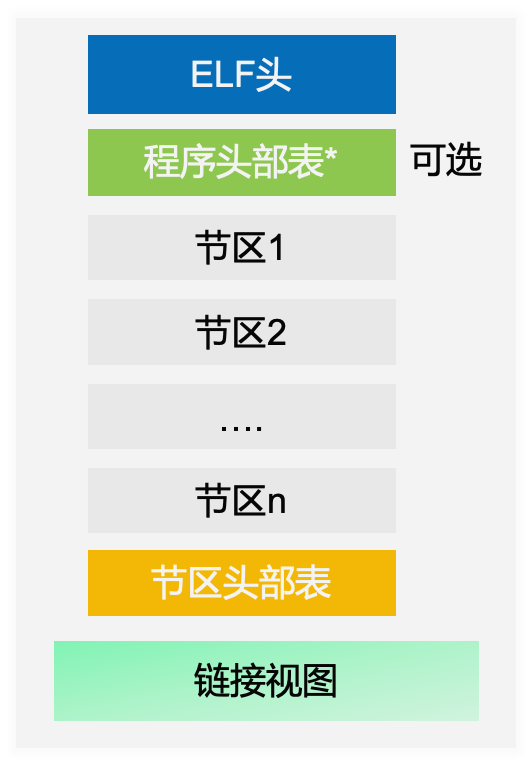


图 2 链接视图

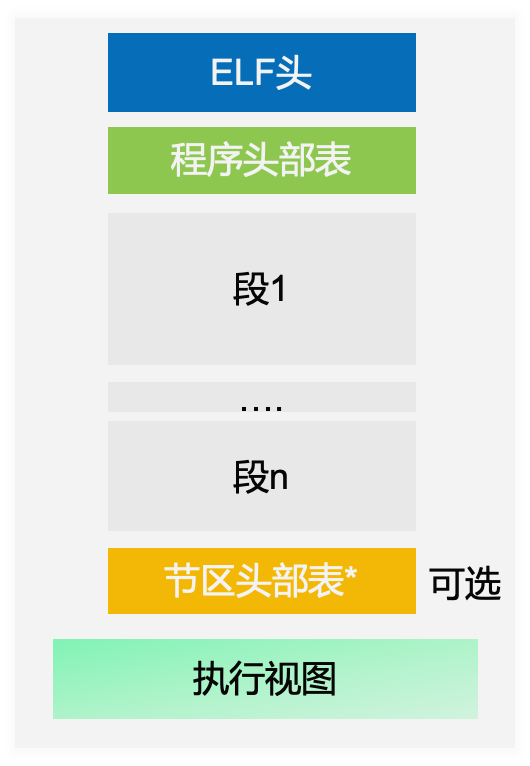


图 3 执行视图

从图3中可以看出，包含了一个ELF头部，它描绘了整个文件的组织结构。它还包括很多节区（section）。这些节有的是系统定义好的，有些是用户在文件在通过.section命令自定义的，链接器会将多个输入目标文件中的相同的节合并。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。除此之外，还包含程序头部表（可选）和节区 头部表，程序头部表，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。而节区头部表（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。

需要注意地是：尽管图中显示的各个组成部分是有顺序的，实际上除了 ELF 头部表以外，其他节区和段都没有规定的顺序。

图4是以程序执行视图来看待的，与左边对应，多了一个段（segment）的概念，编译器在生成目标文件时，通常使用从零开始的相对地址，而在链接过程中，链接器从一个指定的地址开始，根据输入目标文件的顺序，以段（segment）为单位将它们拼装起来。其中每个段可以包括很多个节（section）。

## 2.3 ELF文件解析选项

-a , --all 显示全部信息,等价于 -h -l -S -s -r -d -V -A -I 。

-h , --file-header 显示 elf 文件开始的文件头信息.

-l , --program-headers , --segments 显示程序头（段头）信息(如果有的话)。

-S , --section-headers , --sections 显示节头信息(如果有的话)。

-g , --section-groups 显示节组信息(如果有的话)。

-t , --section-details 显示节的详细信息( -S 的)。

-s , --syms , --symbols 显示符号表段中的项（如果有的话）。

-e , --headers 显示全部头信息，等价于: -h -l -S

-n , --notes 显示 note 段（内核注释）的信息。

-r , --relocs 显示可重定位段的信息。

-u , --unwind 显示 unwind 段信息。当前只支持 IA64 ELF 的 unwind 段信息。

-d , --dynamic 显示动态段的信息。

-V , --version-info 显示版本段的信息。

-A , --arch-specific 显示 CPU 构架信息。

-D , --use-dynamic 使用动态段中的符号表显示符号，而不是使用符号段。

-x , --hex-dump= 以16进制方式显示指定段内内容。 number 指定段表中段的索引,或字符串指定文件中的段名。

-I , --histogram 显示符号的时候，显示 bucket list 长度的柱状图。

-v , --version 显示 readelf 的版本信息。

-H , --help 显示 readelf 所支持的命令行选项。

-W , --wide 宽行输出。

## 2.4 ELF选项-h的基础理论

##### 2.4.1 -h选项的成员参数

选项-h ： 显示ELF的文件头信息

-h 命令可以显示 文件的 ELF 文件头信息，包括文件类型、机器架构、入口地址、程序头表和节头表的偏移和数量等信息。

成员函数及其含义如下：

表 1 ELF-h选项的成员值表

|  |  |
| --- | --- |
| **成员** | **含义** |
| e\_ident | 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 |
|  | ABI Version: 0 |
| e\_type | Type: None (None) |
|  | ELF文件类型 |
| e\_machine | Machine: None |
|  | ELF文件的CPI平台属性 |
| e\_version | Version: 0x0 |
|  | ELF版本号。一般为常数1 |
| e\_entry | Entry point address: 0x0 |
|  | 入口地址，规定ELF程序的入口虚拟地址，操作系统在加载完该程序后从这个地址开始执行进程的指令。可重定位指令一般没有入口地址，则该值为0 |
| e\_phoff | Start of program headers: 0(bytes into file) |
| e\_shoff | Start of section headers: 0 (bytes into file) |
|  | Section Header Table 在文件中的偏移 |
| e\_word | Flags: 0x0 |
|  | ELF标志位，用来标识一些ELF文件平台相关的属性。 |
| e\_ehsize | Size of this header: 0 (bytes) |
|  | ELF Header本身的大小 |
| e\_phentsize | Size of program headers: 0 (bytes) |
| e\_phnum | Number of program headers: 0 |
| e\_shentsize | Size of section headers: 0 (bytes) |
|  | 单个Section Header大小 |
| e\_shnum | Number of section headers: 0 |
|  | Section Header的数量 |
| e\_shstrndx | Section header string table index: 0 |
|  | Section Header字符串表在Section Header Table中的索引 |

##### 2.4.2 Magic魔数

每种可执行文件的格式的开头几个字节都是很特殊的，特别是开头4个字节，通常被称为魔数（Magic Number）。通过对魔数的判断可以确定文件的格式和类型。如：ELF的可执行文件格式的头4个字节为0x7F、e、l、f；Java的可执行文件格式的头4个字节为c、a、f、e；如果被执行的是Shell脚本或perl、python等解释型语言的脚本，那么它的第一行往往是#!/bin/sh或#!/usr/bin/perl或#!/usr/bin/python，此时前两个字节#和!就构成了魔数，系统一旦判断到这两个字节，就对后面的字符串进行解析，以确定具体的解释程序路径。

##### 2.4.3 ELF文件类型

ELF文件主要有三种类型，可以通过ELF Header中的e\_type成员进行区分。

* 可重定位文件（Relocatable File）：ETL\_REL。一般为.o文件。可以被链接成可执行文件或共享目标文件。静态链接库属于可重定位文件。
* 可执行文件（Executable File）：ET\_EXEC。可以直接执行的程序。
* 共享目标文件（Shared Object File）：ET\_DYN，一般为.so文件，有两种情况可以使用。
  + 链接器将其与其他可重定位文件、共享目标文件链接成新的目标文件；
  + 动态链接器将其与其他共享目标文件、结合一个可执行文件，创建进程映像。

##### 2.4.4 机器类型

ELF文件格式被设计成可以在多个平台下使用，但这并不是表示同一个ELF文件可以在不同的平台下使用，而是表示不同平台下的ELF文件都遵循同一套ELF标准。e\_machine成员就表示该ELF文件的平台属性，

* EM\_M32：1 「AT&T WE 32100」
* EM\_SPARC：2 「SPARC」
* EM\_386: 3 「Intel x86」
* EM\_68K：4 「Motorola 680000」
* EM\_88K：5 「Motorola 88000」
* EM\_860：6 「Intel 80860」

## 2.5 ELF选项-S的基础理论

##### 2.5.1 选项- S的介绍

选项-S用于列出节的头信息

段（segment) 和 节（section)是有区别的。 节不是段。 段是程序执行的必要组成部分。 在每个段中会有代码或者数据被划分为不同的节。 而 节头表 则是对这些节的位置和大小的描述，主要是用于链接和调试。

因为节头表没有对程序的内存布局进行描述，对程序内存布局的描述的是 程序头表的任务。 所以节头表就是对程序头的补充。所以我们将“节头表”都填充为无用的值的话程序还是可以正常启动的。 虽然可以修改，但是随着版本的升级，保不准以后会使用。 就跟window平台上的最小PE一样，在xp下可以是100kb以下。win7下则对各个表的限制更严格了。所以需要100kb以上。 win10同理。 所以节头表以后可能会使用。

* **节头和节**

​ 如果二进制文件中缺少了 节头 ，并不意味着节是不存在的。只是没有办法通过节头来引用节了，对于调试器或者反编译（IDA)程序来说，参考的信息就变少了。

每一个节都保存了某种类型的代码和数据。数据可以是程序中的全局变量，也可以是连接器所需要的动态链接的信息。这点跟windows平台的 windowsPE文件格式设计很像。节头可以被抹除，默认的ELF文件是有节头的。如果节头被抹除，那么 objcopy objdump gdb IDA 等工具就可能无法使用。而节头表中的 .dynsym这样的节（记录了函数名和偏移地址，导入导出等符号）就无法分析。进而就增大了逆向的难度。

##### 2.5.2 选项- S的成员参数

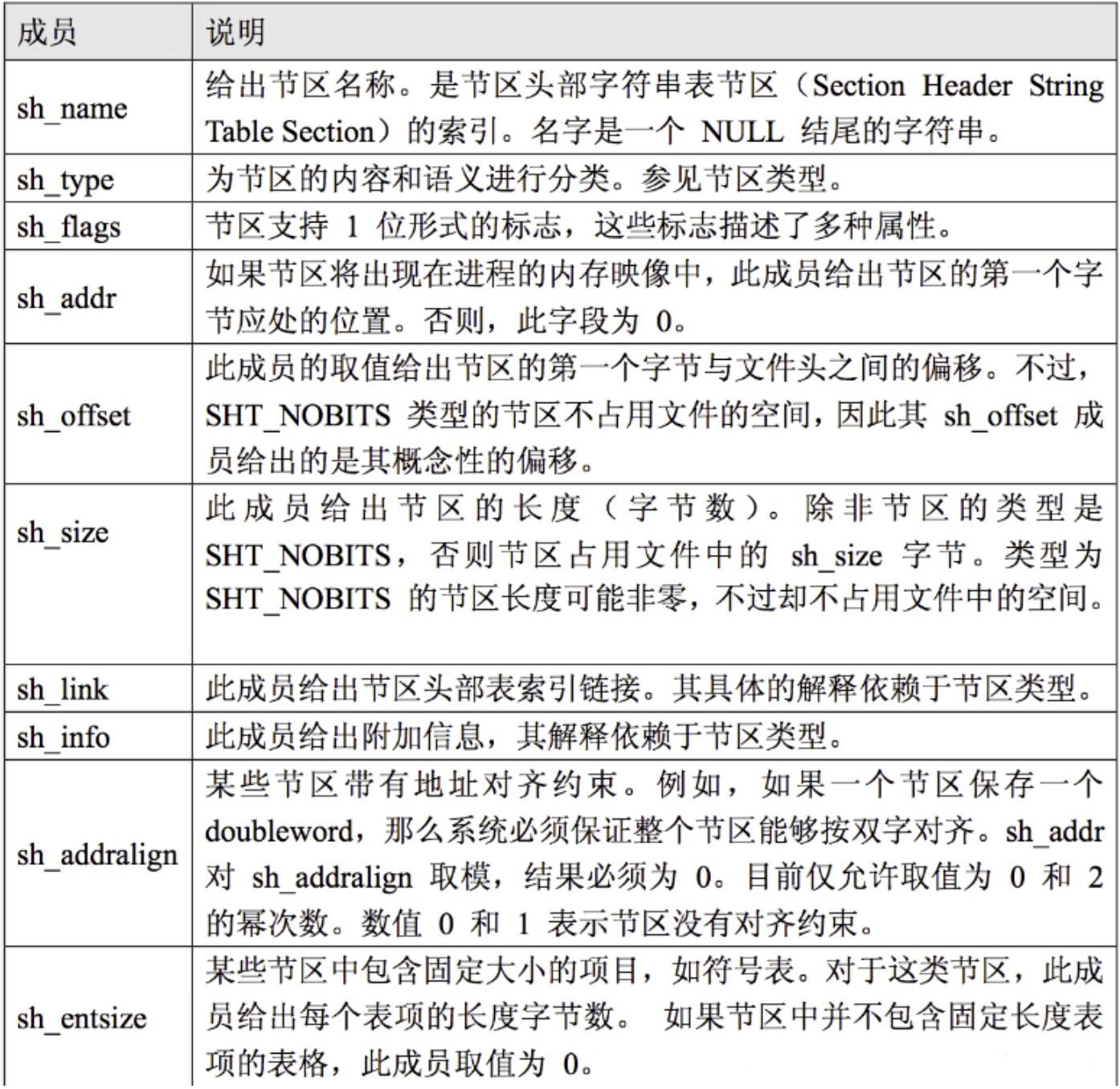


图 4 选项-S的成员参数列表图

sh\_name：0——无名称

sh\_type：SHT\_NULL——非活动

sh\_flags：0——无标志

此字段定义了一个节区中包含的内容是否可以修改，是否可以执行的信息。 如果一个标志位被设置，则该位取值为1. 定义的各位都设置为0

SHF\_WRITE：0x1

SHF\_ALLOC：0x2

SHF\_EXECINSTR：0x4 ——节区包含可执行的机器指令

SHF\_MASKPROC：0xF0000000

sh\_addr：0——无地址

sh\_offset：0——无文件偏移

sh\_size：0——无尺寸大小

sh\_link：SHN\_UNDEF——无链接信息

sh\_inf：0——无辅助信息

sh\_addralign：0——无对齐要求

sh\_entsize：0——无表项

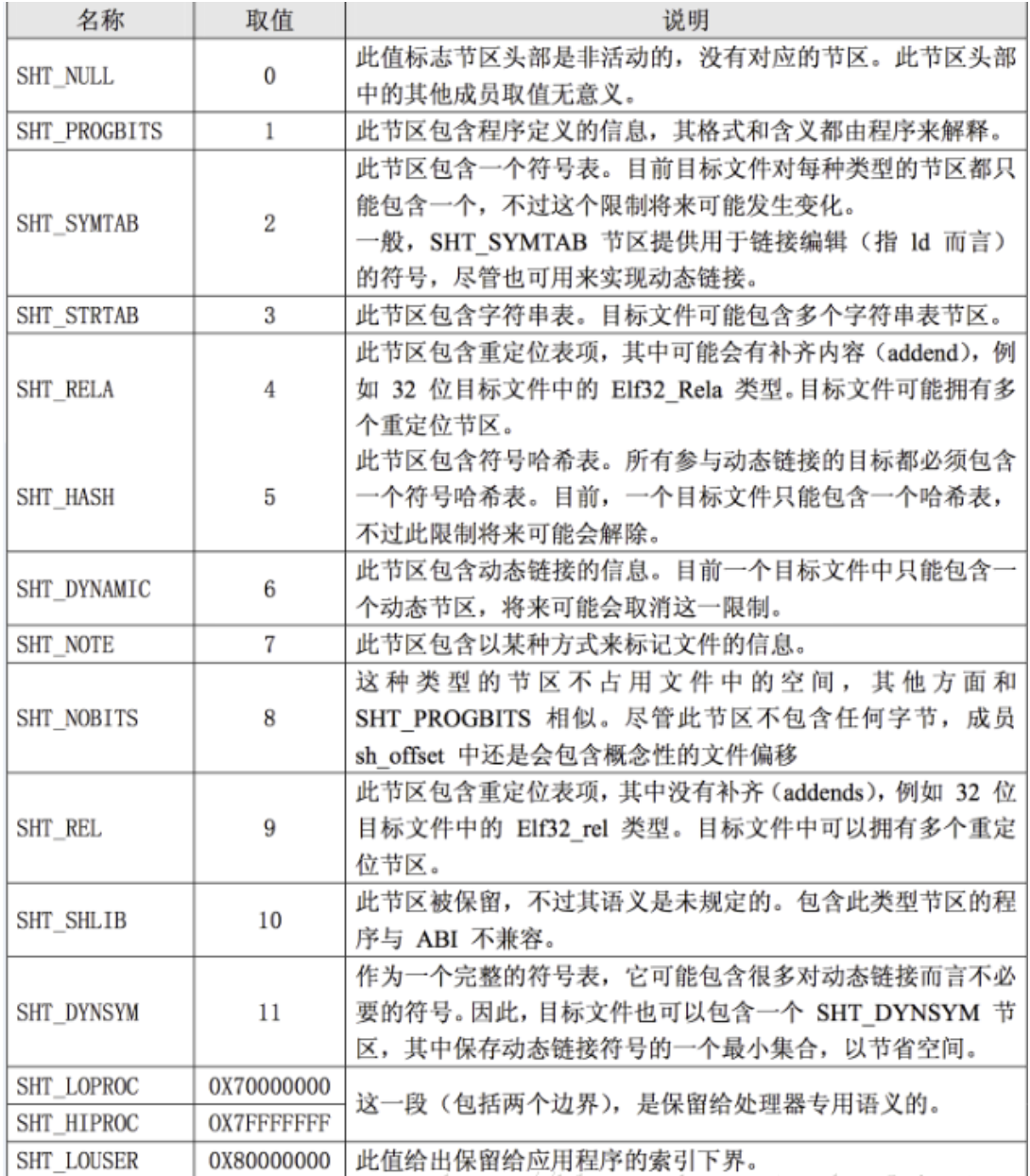


图 5 选项-S的取值列表图

##### 2.5.3 特殊节区

表 2特殊节区列表

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **类型** | **属性** | **含义** |
| .bss | SHT NO BITS | 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\_PROG BITS | (无) | 包含版本控制信息。 |
| .data | SHT PROG BITS | 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。 |
| .data l | SHT\_PROG BITS | SHF\_ALLOC |  |
| SHF WRITE |  |  |  |
| .debug | SHT\_PROG BITS | (无) | 此节区包含用于符号调试的信息。 |
| .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。 |
| .dyn str | 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\_PROG BITS | SHF\_ALLOC SHF\_EXEC INSTR | 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\_PROG BITS |  | 此节区包含全局偏移表。 |
| .hash | SHT\_HASH | SHF\_ALLOC | 此节区包含了一个符号哈希表。 |
| .in it | SHT\_PROG BITS | SHF\_ALLOC SHF\_EXEC INSTR | 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 PROG BITS |  | 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\_PROG BITS | (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\_PROG BITS |  | 此节区包含过程链接表(procedure linkage |
| table) 。 |  |  |  |
| .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。 |
| .rela name | SHT RELA |  |  |
| .rodata | SHT PROG BITS | SHF ALLOC | 这些节区包含只读数据，这些数据通常参与 |
| .rodata l | SHT\_PROG BITS | SHF\_ALLOC | 进程映像的不可写段。 |
| .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 PROG BITS | SHF\_ALLOC SHF EXEC INSTR | 此节区包含程序的可执行指令 |

## 2.6 ELF选项-l的基础理论

##### 2.6.1 选项- l的介绍

ELF 程序头是对二进制文件中段的描述，是程序装载必须的一部分。

* 段（segment) 是在内核装载时被解析的。
  + 主要作用就是描述磁盘上可执行文件的内存布局以及如何映射到内存中。
  + 可以通过引用原始的ELF头中名为： e\_phoff(程序头表的偏移量)的偏移量来得到程序头表。

基本的几个程序段内容如下：

* **PT\_LOAD段**

p\_type描述了段的类型。 一个可执行文件至少要有一个PT\_LOAD类型的段。 这类程序头描述的是可装载的段，

也就是说，这种类型的段会被装载或者映射到内存中。

一般来说，一个动态链接的ELF可执行文件通常包含两个可装载的段。 段类型都为PT\_LOAD

一个是存放程序代码的text段

另一个是存放全局变量和动态链接信息的data段。

上面两个段则会根据p\_align的对齐值在内存中对齐。并且映射到内存中。一般来说，TEXT段也称为代码段，权限一般都是可读可执行的。 对应取值就是PF\_READ\_EXEC，data段一般就是读写权限。 可以修改p\_flags来让我们的程序权限增大。

* **PT\_PHDR段**

此段一般位于elf文件的第一个段。PT\_PHDR段保存了程序头表本身的位置和大小。 phdr表保存了所有的phdr对文件（以及内存镜像）中段的描述信息。

* **PT\_INTERP段**

PT\_INTERP：存放一个以null结尾的字符串位置和大小信息，是对解释器位置的描述；

* **PT\_DYNAMIC段**

动态段是动态链接可执行文件所特有的，包含了动态链接器所需要的一些信息，它包括以下内容

运行时需要链接的共享库列表

* 全局偏移表(GOT)
* 重定位条目相关信息
* **PT\_NOTE**

可以保存于特定供应商或者系统相关附加信息，这个段在程序运行时是不需要的，因为系统会假设可执行文件是本地的，这个段很容易被感染

##### 2.6.2 选项- l的信息解释

-l 选项用于显示ELF文件中的程序头表和节区头表信息，其主要作用有：

1. 显示程序头表信息。程序头表描述ELF文件各个段(代码段、数据段等)在文件中的布局信息。
2. 显示节区头表信息。节区头表描述ELF文件各个节区(代码节、数据节等)在文件中的布局信息。
3. 帮助分析ELF文件 segments和sections的组织结构。

程序头表显示如下信息：

1. p\_type：表示程序段的类型信息，标识了段的类型
   1. 主要类型包括PT\_LOAD，PT\_DYNAMIC，PT\_INFERP
      1. PT\_LOAD类型的段会在创建进程时加载到内存中
      2. PT\_INFERP类型的段包含了.interp节，该节提供了加载二进制文件的解释器的名称
      3. PT\_DYNAMIC包含了.dynamic节，该节告诉解释器如何解析二进制文件用于执行
2. p\_offset：表示分段文件的偏移信息
3. p\_vaddr：表示段段虚拟地址信息
4. p\_paddr：表示段物理地址信息
5. p\_filesz：文件中段的大小
6. p\_memsz：内存中段段大小
7. p\_flags：段的标志信息
   1. 指示了段在运行时的访问权限，有三种重要的类型：
      1. PF\_X（可执行），PF\_W（可写），PF\_R（可读）
8. p\_align：段、文件、内存对齐信息：指定了段所需的内存对齐方式（字节为单位）
   1. 提供了二进制文件的段视图，ELF包括零或多个节，实际上就是把这些节捆绑成单个块
   2. 段提供的可执行视图，只有二进制文件会用到

## 2.6 ELF选项-d的基础理论

# 3系统设计

## 3.1 -h选项文件头表的数据结构设计

**我们主要以32位的数据结构作为主要实现方案。**

32位数据结构设计

typedef struct {

unsigned char e\_ident[16]; /\* ELF "magic数" \*/

unsigned char e\_type[2]; /\* 标识对象文件类型 \*/

unsigned char e\_machine[2]; /\* 指定所需的体系结构 \*/

unsigned char e\_version[4]; /\* 标识目标文件版本 \*/

unsigned char e\_entry[4]; /\* 入口点虚拟地址\*/

unsigned char e\_phoff[4]; /\* 程序头表文件偏移量 \*/

unsigned char e\_shoff[4]; /\* 节头表文件偏移量 \*/

unsigned char e\_flags[4]; /\* 特定于处理器的标志 \*/

unsigned char e\_ehsize[2]; /\* ELF头大小（以字节为单位） \*/

unsigned char e\_phentsize[2]; /\* 程序头表条目大小 \*/

unsigned char e\_phnum[2]; /\* 程序头表条目计数 \*/

unsigned char e\_shentsize[2]; /\* 节头表条目大小 \*/

unsigned char e\_shnum[2]; /\* 节头表项计数 \*/

unsigned char e\_shstrndx[2]; /\* 节头字符串表索引 \*/

} Elf32\_External\_Ehdr;

64位数据结构设计

/\*与Elf32\_External\_Ehdr 一致，不过适用于64位\*/

typedef struct {

unsigned char e\_ident[16]; /\* ELF "magic number" \*/

unsigned char e\_type[2]; /\* Identifies object file type \*/

unsigned char e\_machine[2]; /\* Specifies required architecture \*/

unsigned char e\_version[4]; /\* Identifies object file version \*/

unsigned char e\_entry[8]; /\* Entry point virtual address \*/

unsigned char e\_phoff[8]; /\* Program header table file offset \*/

unsigned char e\_shoff[8]; /\* Section header table file offset \*/

unsigned char e\_flags[4]; /\* Processor-specific flags \*/

unsigned char e\_ehsize[2]; /\* ELF header size in bytes \*/

unsigned char e\_phentsize[2]; /\* Program header table entry size \*/

unsigned char e\_phnum[2]; /\* Program header table entry count \*/

unsigned char e\_shentsize[2]; /\* Section header table entry size \*/

unsigned char e\_shnum[2]; /\* Section header table entry count \*/

unsigned char e\_shstrndx[2]; /\* Section header string table index \*/

} Elf64\_External\_Ehdr;

## -S选项节区头表的数据结构设计

**我们主要以32位的数据结构作为主要实现方案。**

32位数据结构设计

/\* 节区头表32位 \*/

typedef struct {

unsigned char sh\_name[4]; /\* 节名，字符串 tbl 中的索引 \*/

unsigned char sh\_type[4]; /\* 节头表的类型 \*/

unsigned char sh\_flags[4]; /\* 杂项节头表的属性 \*/

unsigned char sh\_addr[4]; /\* 执行时的节头虚拟地址\*/

unsigned char sh\_offset[4]; /\* 节头文件的偏移 \*/

unsigned char sh\_size[4]; /\* 节的大小（以字节为单位） \*/

unsigned char sh\_link[4]; /\* 另一节的索引\*/

unsigned char sh\_info[4]; /\* 附加节头信息 \*/

unsigned char sh\_addralign[4]; /\* 节头对齐 \*/

unsigned char sh\_entsize[4]; /\* 如果节头包含表的话条目大小 \*/

} Elf32\_External\_Shdr;

64位数据结构设计

/\* 节区头表64位 \*/

typedef struct {

unsigned char sh\_name[4]; /\* Section name, index in string tbl \*/

unsigned char sh\_type[4]; /\* Type of section \*/

unsigned char sh\_flags[8]; /\* Miscellaneous section attributes \*/

unsigned char sh\_addr[8]; /\* Section virtual addr at execution \*/

unsigned char sh\_offset[8]; /\* Section file offset \*/

unsigned char sh\_size[8]; /\* Size of section in bytes \*/

unsigned char sh\_link[4]; /\* Index of another section \*/

unsigned char sh\_info[4]; /\* Additional section information \*/

unsigned char sh\_addralign[8]; /\* Section alignment \*/

unsigned char sh\_entsize[8]; /\* Entry size if section holds table \*/

} Elf64\_External\_Shdr;

## 3.3 -l选项程序头表的数据结构设计

**我们主要以32位的数据结构作为主要实现方案。**

32位数据结构设计

/\* 程序头表 \*/

typedef struct {

unsigned char p\_type[4]; /\* 标识程序段类型 \*/

unsigned char p\_offset[4]; /\* 分段文件偏移 \*/

unsigned char p\_vaddr[4]; /\* 段虚拟地址 \*/

unsigned char p\_paddr[4]; /\* 段物理地址 \*/

unsigned char p\_filesz[4]; /\* 文件中的段大小 \*/

unsigned char p\_memsz[4]; /\* 内存中的段大小 \*/

unsigned char p\_flags[4]; /\* 段标志 \*/

unsigned char p\_align[4]; /\* 段对齐、文件和内存 \*/

} Elf32\_External\_Phdr;

64位数据结构设计

typedef struct {

unsigned char p\_type[4]; /\* 标识程序段类型 \*/

unsigned char p\_flags[4]; /\* 段标志 \*/

unsigned char p\_offset[8]; /\* 段文件偏移量 \*/

unsigned char p\_vaddr[8]; /\* 段虚拟地址 \*/

unsigned char p\_paddr[8]; /\* 段物理地址 \*/

unsigned char p\_filesz[8]; /\* 文件中的段大小 \*/

unsigned char p\_memsz[8]; /\* 内存中的段大小 \*/

unsigned char p\_align[8]; /\* 段对齐、文件和内存 \*/

} Elf64\_External\_Phdr;

## 3.4 -d选项动态链接表的数据结构设计

/\* 动态节区结构 \*/

/\*32位\*/

typedef struct {

unsigned char d\_tag[4]; /\* 条目标签值 \*/

union {

unsigned char d\_val[4];

unsigned char d\_ptr[4];

} d\_un;

} Elf32\_External\_Dyn;

/\*64位\*/

typedef struct {

unsigned char d\_tag[8]; /\* 条目标签值 \*/

union {

unsigned char d\_val[8];

unsigned char d\_ptr[8];

} d\_un;

} Elf64\_External\_Dyn;

# 4系统实现

## 4.1 -h选项文件头表的程序设计

图示

描述已自动生成

图 6 选项-h实现程序流程图

以上功能大致代码实现如下：

如果需要完整代码请查看附录或前往我们的Github开源地址进行下载

1. /\*读取文件头\*/

2. /\*头文件信息在Magic 数中\*/

3. int ELF\_process::process\_file\_header(void)

4. {

5. /\*判断是否Magic是否存在问题\*/

6. if ( elf\_header.e\_ident[EI\_MAG0] != ELFMAG0

7. || elf\_header.e\_ident[EI\_MAG1] != ELFMAG1

8. || elf\_header.e\_ident[EI\_MAG2] != ELFMAG2

9. || elf\_header.e\_ident[EI\_MAG3] != ELFMAG3)

10. {

11. printf("Not an ELF file - it has the wrong magic bytes at the start\n");

12. return 0;

13. }

14. /\*输出M相关参数\*/

15. printf("ELF Header:\n");

16. printf(" Magic: ");

17. for (int i = 0; i <EI\_NIDENT ; ++i)

18. printf ("%2.2x ", elf\_header.e\_ident[i]); /\*读取Magic 数\*/

19. printf("\n");

20. printf(" Class: %s\n",

21. get\_elf\_class(elf\_header.e\_ident[EI\_CLASS]));

22.

23. printf (" Data: %s\n",

24. get\_data\_encoding (elf\_header.e\_ident[EI\_DATA]));

25. const char \*str=elf\_header.e\_ident[EI\_VERSION] == EV\_CURRENT ?

26. "(current)" : (elf\_header.e\_ident[EI\_VERSION] != EV\_NONE ? "<unknown>" : "");

27. printf (" Version: %d %s\n",

28. elf\_header.e\_ident[EI\_VERSION],str);

29. printf (" OS/ABI: %s\n",

30. get\_osabi\_name (elf\_header.e\_ident[EI\_OSABI]));

31.

32. printf (" ABI Version: %d\n",

33. elf\_header.e\_ident[EI\_ABIVERSION]);

34.

35. printf (" Type: %s\n",

36. get\_file\_type (elf\_header.e\_type));

37.

38. printf (" Machine: %s\n",

39. get\_machine\_name (elf\_header.e\_machine));

40.

41. printf (" Version: 0x%lx\n",

42. (unsigned long) elf\_header.e\_version);

43.

44. printf (" Entry point address: 0x%x",elf\_header.e\_entry);

45.

46. printf ("\n Start of program headers: %d",elf\_header.e\_phoff);

47.

48. printf (" (bytes into file)\n Start of section headers: %d",elf\_header.e\_shoff);

49. printf (" (bytes into file)\n");

50.

51. printf (" Flags: 0x%lx\n",(unsigned long)elf\_header.e\_flags);

52.

53. printf (" Size of this header: %ld (bytes)\n",(long)elf\_header.e\_ehsize);

54.

55. printf (" Size of program headers: %ld (bytes)\n",(long)elf\_header.e\_phentsize);

56.

57. printf (" Number of program headers: %ld\n",(long)elf\_header.e\_phnum);

58.

59. printf (" Size of section headers: %ld (bytes)\n",

60. (long) elf\_header.e\_shentsize);

61.

62. printf (" Number of section headers: %ld\n",

63. (long) elf\_header.e\_shnum);

64.

65. if (section\_headers != NULL && elf\_header.e\_shnum == SHN\_UNDEF)

66. printf (" (%ld)", (long) section\_headers[0].sh\_size);

67.

68. printf (" Section header string table index: %ld\n",

69. (long) elf\_header.e\_shstrndx);

70.

71. return 1;

73. }

74.

## 4.2 -S选项节区头表的程序设计

图示

描述已自动生成

图 7选项-S实现程序流程图

以上功能大致代码实现如下：

如果需要完整代码请查看附录或前往我们的Github开源地址进行下载

1. //如果是-S或-e指令，显示节头信息和全部头信息

2. else if((option & (1<<2) ) || (option & (1<<6))) //-S || -e

3. {

4.

5. //打印列名

6. printf(" [Nr] Name Type Addr Off Size ES Flg Lk Inf Al\n");

7. //遍历节区头表的所有项，每一个项都对应一个节区

8. for (int i = 0; i < elf\_header.e\_shnum; i++, section++)

9. {

10. printf (" [%2u] ", i);

11.

12. //计算该节区名在shstrtab中的偏移地址

13. countC = flag\_shoff + section->sh\_name;

14.

15. //将文件指针移动到这个地方

16. fseek(file,countC,SEEK\_SET);

17. //名字字符串，长度为20，名字以'\0'结尾，所以多读一些没有关系

18. char string\_name[20];

19. //从文件中读取名字字符串

20. fread(string\_name,20,1,file);

21.

22. //判断节区名是不是"IA\_64",记录unwind节区的索引

23. if(!strcmp(string\_name,"IA\_64")) unwind\_idx=i;

24. //判断节区名和目标的节区名是否一致，相同的话记录目标节区名对应的索引，这个是给-x用的

25. if(!strcmp(string\_name,target\_section\_name)) target\_section\_idx=i;

26.

27. //打印输出节区名

28. printf("%-16s ",string\_name);

29.

30.

31. //打印输出节区类型

32. printf ( " %-15.15s ",

33. get\_section\_type\_name (section->sh\_type));

34.

35. //打印输出节区虚拟地址、文件中偏移地址、节区大小、节区条目大小

36. printf("%6.8lx",(unsigned long) section->sh\_addr);

37. printf ( " %6.6lx %6.6lx %2.2lx",

38. (unsigned long) section->sh\_offset,

39. (unsigned long) section->sh\_size,

40. (unsigned long) section->sh\_entsize);

41.

42. //如果节区有标志位那么就打印输出标志

43. if (section->sh\_flags)

44. printf (" %2.2x ", section->sh\_flags);

45. //否则用空格填充

46. else

47. printf("%4c",32);

48.

49. //打印输出sh\_link和sh\_info，这个因不同节区类型而异，最后输出对齐信息

50. printf ("%2u ", section->sh\_link);

51. printf ("%3u %3lu", section->sh\_info,

52. (unsigned long) section->sh\_addralign);

53.

54. //如果节区名是.dynamic动态链接信息表，那么需要记录节区的偏移地址和大小

55. if (strcmp(string\_name,".dynamic")==0)

56. {

57. dynamic\_addr = section->sh\_offset;

58. dynamic\_size = section->sh\_size;

59. }

60.

61. //如果节区名是.rel.dyn动态链接重定位表，那么需要记录节区的偏移地址和大小

62. if (strcmp(string\_name,".rel.dyn")==0)

63. {

64. rel\_dyn\_offset = section->sh\_offset;

65. rel\_dyn\_size = section->sh\_size;

66. }

67.

68. //如果节区名是.dynsym动态链接符号表，那么需要记录节区的偏移地址和大小

69. if(strcmp(string\_name,".dynsym")==0)

70. {

71. sym\_dyn\_offset = section->sh\_offset;

72. sym\_dyn\_size = section->sh\_size;

73. }

74.

75. //如果节区名是.dynstr动态链接字符表，那么需要记录节区的偏移地址和大小

76. if(strcmp(string\_name,".dynstr")==0)

77. {

78. str\_dyn\_offset = section->sh\_offset;

79. str\_dyn\_size = section->sh\_size;

80. }

81.

82. printf("\n");

83.

84. }

85. }

86.

## 4.3 -l选项程序头表的程序设计

图示

描述已自动生成

图 8选项-l实现程序流程图

以上功能大致代码实现如下：

如果需要完整代码请查看附录或前往我们的Github开源地址进行下载

1. int ELF\_process::process\_program\_headers(FILE \*file)

2. {

3.

4. Elf32\_Phdr\* segment;

5. unsigned long dynamic\_addr;

6. if(elf\_header.e\_phnum == 0)

7. {

8.

9. if(elf\_header.e\_phoff!=0)

10. {

11. printf ("possibly corrupt ELF header - it has a non-zero program"

12. " header offset, but no program headers");

13. }

14. else

15. {

16. printf ("\nThere are no program headers in this file.\n");

17. return 0;

18. }

19.

20. }

21. else

22. {

23. /\*打印相关的信息\*/

24. if(elf\_header.e\_phnum>1)

25. printf ("\nProgram Headers:\n");

26. else

27. printf ("\nProgram Header:\n");

28.

29. if(is\_32bit\_elf)

30. printf(" Type Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align\n");

31. else

32. printf(" Type Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align\n");

33.

34. }

35.

36. /\*获取程序段头表信息\*/

37. if (! get\_program\_headers (file))

38. return 0;

39.

40.

41. unsigned int i;

42. for (i = 0, segment = program\_headers;

43. i < elf\_header.e\_phnum;

44. i++, segment++)

45. {

46. printf (" %-14.14s ", get\_segment\_type (segment->p\_type));

47.

48. if(is\_32bit\_elf) /\*如果是32位的话，那么输出程序段的相关信息\*/

49. {

50. printf ("0x%6.6x ", (unsigned int) segment->p\_offset);

51. printf ("0x%8.8x ", (unsigned int) segment->p\_vaddr);

52. printf ("0x%8.8x ", (unsigned int) segment->p\_paddr);

53. printf ("0x%5.5x ", (unsigned int) segment->p\_filesz);

54. printf ("0x%5.5x ", (unsigned int) segment->p\_memsz);

55. printf ("%c%c%c ",

56. (segment->p\_flags & PF\_R ? 'R' : ' '),

57. (segment->p\_flags & PF\_W ? 'W' : ' '),

58. (segment->p\_flags & PF\_X ? 'E' : ' '));

59. printf ("%#x", (unsigned int) segment->p\_align);

60. }

61.

62. printf("\n");

63.

64. }

65.

66. return 0;

67. }

68.  

## 4.4 -d选项动态链接的程序设计

# 5系统测试

## 5.1测试用例设计

对于-h，-S和-l指令设计的测试代码如下：

1. #include <iostream>

2.

3. using namespace std;

4.

5.

6. int main(){

7. int a = 1;

8. cout << "Hello Linux!!HZAU\n";

9. return 0;

10. }

11.

## 5.2测试结果及分析

##### 5.2.1 选项- h的测试与样例解释

编译产生.so文件

g++ -o test-h.so test-h.cpp

使用./main test-h.so -h 查看头文件信息

文本

描述已自动生成

图 9 运行查看-h选项后的信息显示图

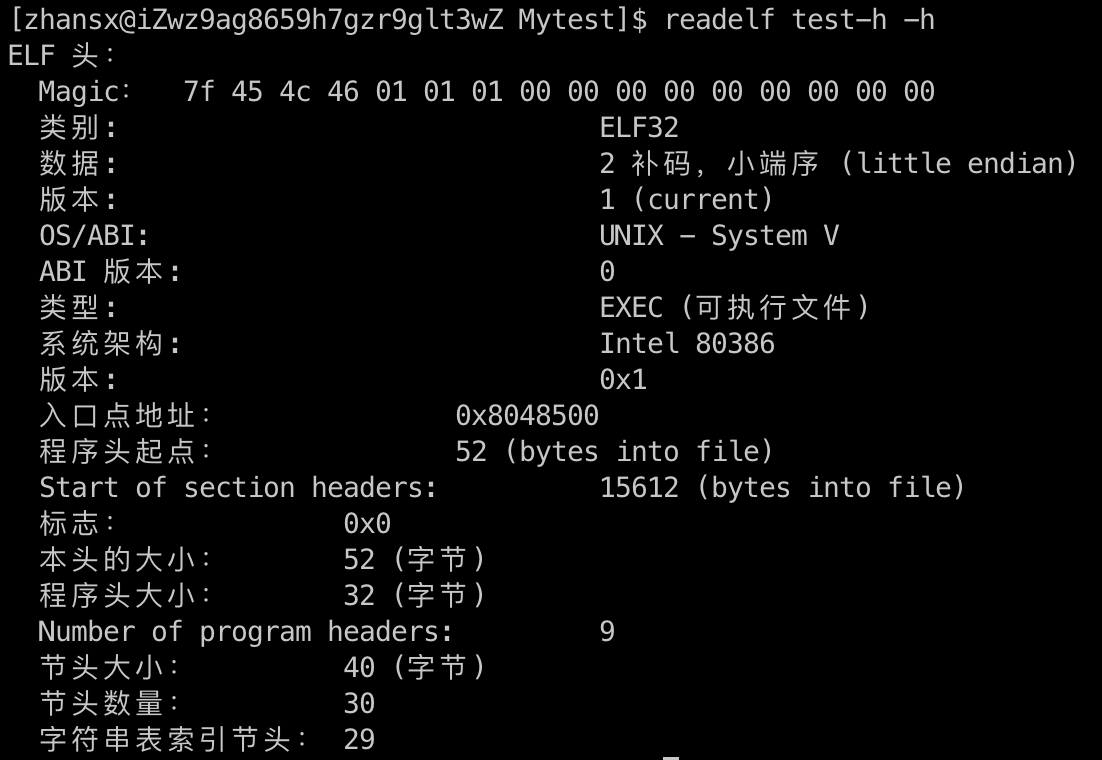


图 10运行readelf查看-h选项后的信息显示图

可以发现，我们的程序与ELF进行对比后，可以正确实现选项-h的内容。

对如上Magic魔数信息的解释：

* 第1个字节：0x7f 表示ELF标记
* 第2个字节：‘E’
* 第3个字节：‘L’
* 第4个字节：‘F’
* 第5个字节：ELF文件类型，也就是class
* 第6个字节：数据编码信息，字节序信息，大端序、小端序
* 第7个字节：ELF版本号：标识ELF Version, 该值等于EV\_CURRENT
* 第8个字节：OS信息
* 第9个字节：ABI 版本信息
* 第10~16个字节：填充位

##### 5.2.2 选项- S的测试与样例解释

首先我们查看文件头信息

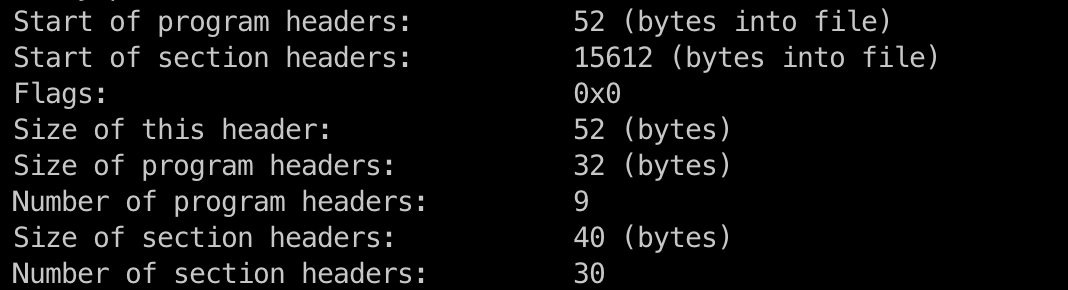


图 11 文件头信息(部分)图

从中可以发现节头表的开始偏移位置位15612字节，每个节头表的大小为40字节，一共有30个节点头表。

* 使用./main test-h.so -S 查看节信息，如下所示：

图片包含 文本

描述已自动生成

图 12 节区头表信息展示图

idex为0的位置：



图 13 索引为0位置的节区图

可以看到 index=0 的 Section Header，内容全部为0。（在某些情况下，它的某部分字段不为0，这些特殊情况当前先跳过） 这是一个非常特别的 Section Header，它的作用是表示 undefined 。它的 index（=0） 也有一个特别的名字，叫 SHN\_UNDEF。

使用hexdump查看索引为0的位置的物理地址信息

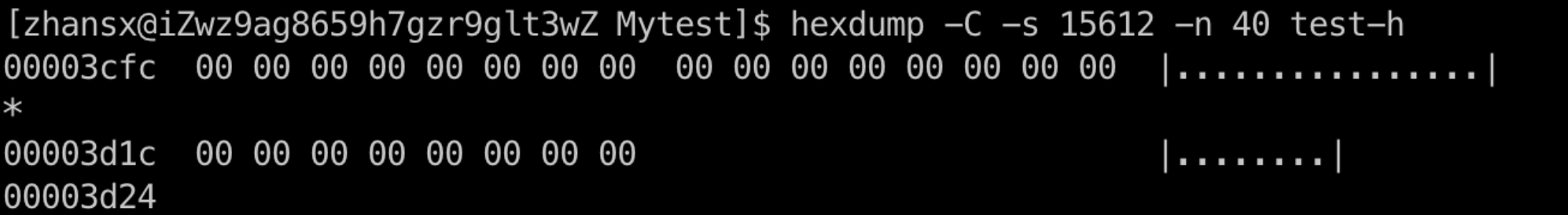


图 14 hexdump查看索引为0位置的物理信息图

那么程序的可执行段在哪个位置呢？答案就在.text节区

现在我们查看.text节区



其偏移地址为16132，长度为40

图形用户界面, 文本, 应用程序

描述已自动生成

图 15hexdump查看索引为.text节区的物理信息图

* 首先前4个字节表示的是sh\_name：

92 00 00 00

值为 90 00 00 00 + little endian = 0x92 = 146

* 接下来的4个字节表示sh\_type

01 00 00 00

值为：“01 00 00 00” + little endian = 0x1 对照表，1对应的类型是 SHT\_PROGBITS，表示这部分的信息，格式和意义完全由程序来定义。内容可以是已初始化的数据，未初始化的数据，comment 或者是程序代码等。

* 接下来的4个字节表示sh\_flag

06 00 00 00

值为：“06 00 00 00” + little endian = 0x6 = SHF\_ALLOC | SHF\_EXECINSTR => 这部分是可执行代码，进程运行时需要放置在内存中。 再对照回 Section Header 列表中 “.text” 那一行

[13] .text PROGBITS 08048500 000500 000245 00 AX 0 0 16

所以可以发现 “.text” 对应的就是 A (alloc), X (execute)。

* 接下来的4个字节表示sh\_addr

00 85 04 08

Section 存放于进程内存映像中的虚存地址，如果 Section 不需要出现在内存中，则为0。Relocatable file 的虚存地址都为0。Executable file 和 Shared object file 才会为有需要的 Section 计算虚存地址。 地址计算为 00 85 04 08 + little endian = 08 04 85 00

* 接下来的4个字节为section在文件中的字节偏移量

00 05 00 00

00 00 05 00 + little endian = 0x500 = 5 \* 16^3 = 20480

* 接下来4个字节表示sh\_size

表示 Section 在文件中占据的字节数。 类型为 NOBITS 的 Section，即使这个值不为0，它在文件中也不占据空间。".bss" Section 就是这个类型。bss 全称为 “block started by symbol”（更简单的记法是：Better Save Space），存放的是未初始化的数据，因为数据无初始值，所以只需要记录它在内存中占据的空间即可，在文件中不需要额外的存储。相反，".data" Section 存储的则是已经初始化的数据，所以需要在文件中记录下初始值，才能在进程内存映像中把这些值带进去。

45 02 00 00

45 02 00 00 + little endian = 00 00 02 45 = 2 \* 16^2 + 4 \* 15 + 5 = 577

* 紧接着的4个字节表示sh\_link

包含另外一个 Section Header 的 index，具体的含义取决于 Section 的类型

00 00 00 00

* 后面4个字节就表示sh\_info

包含了额外的信息，具体的含义取决于 Section 的类型。 需要把 sh\_type，sh\_link 和 sh\_info 联合在一起进行解析。 另外，当 sh\_flags=SHF\_INFO\_LINK 时，sh\_info 则表示另外一个 Section Header 的 index。

00 00 00 00

* 最后8个字节，分别表示sh\_addralign和sh\_entsize
* 前4个字节表示sh\_addralign 表示对齐约束。0或1表示无约束。sh\_addr % sh\_addralign 必须等于0

10 00 00 00

* 后4个字节表示sh\_entsize

有些 Section 会包含一组大小固定的记录，这时 sh\_entsize 就表示记录的大小，通过 sh\_size / sh\_entsize 就能得到记录的个数。如果 Section 没有包含这种类型的记录，则值为0。比如后面会看到的 “.shstrtab” Section，包含的是一组 string，但 string 的长度不一样，于是 sh\_entsize 的值就为0。

00 00 00 00

* 我们再来反复看一下.text节

[13] .text PROGBITS 08048500 000500 000245 00 AX 0 0 16

由此可以得出

1. offset = 0x500 = 20480
2. size = 0x0245 = 577
3. flag = AX => 进程运行期间占据内存 + 这部分是可执行的指令 => 可执行

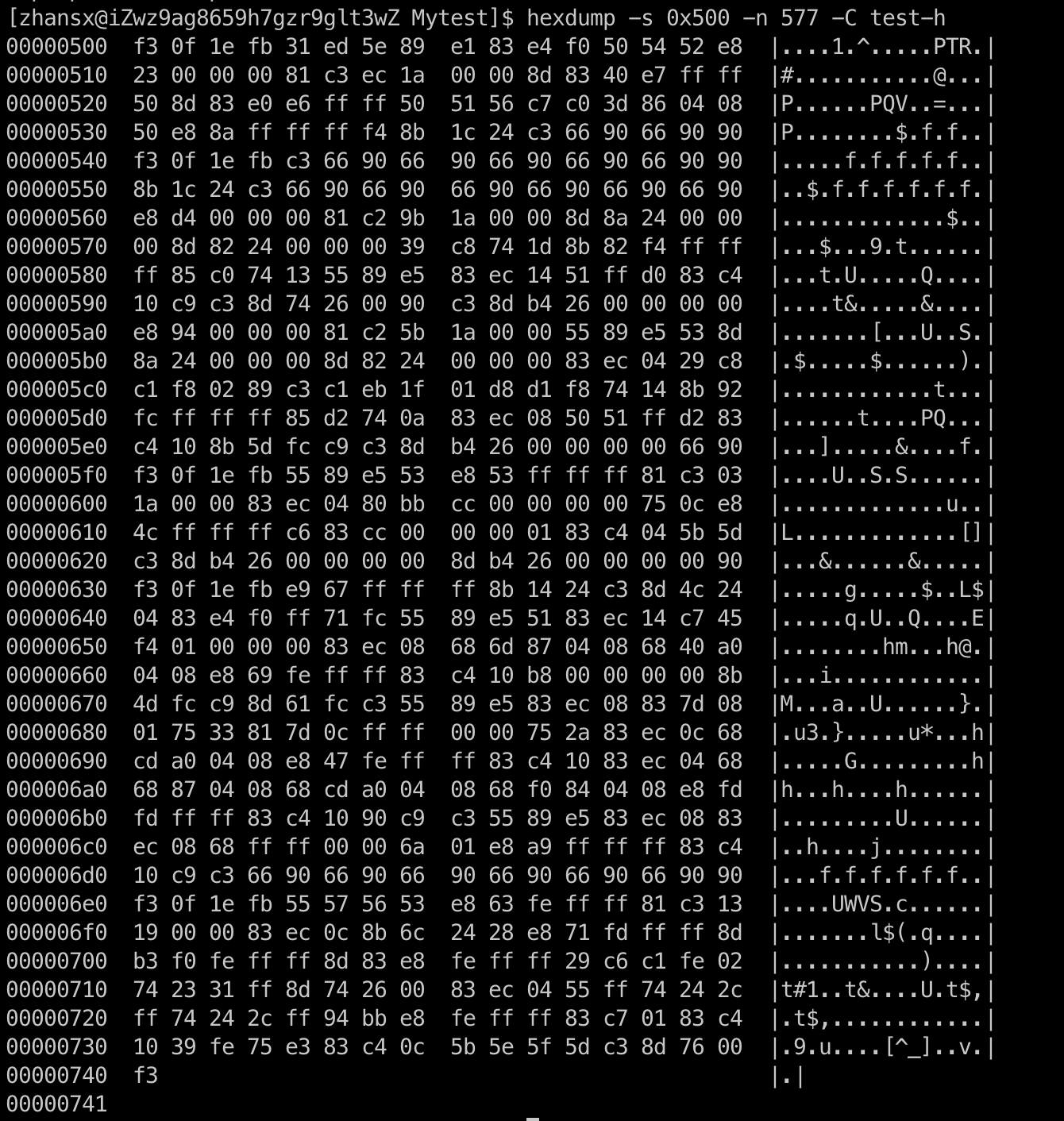


图 16 .text节区的可执行区在物理地址中的信息

可以使用objdump查看其中的内容，红框部分便是mian函数反编译后的结果

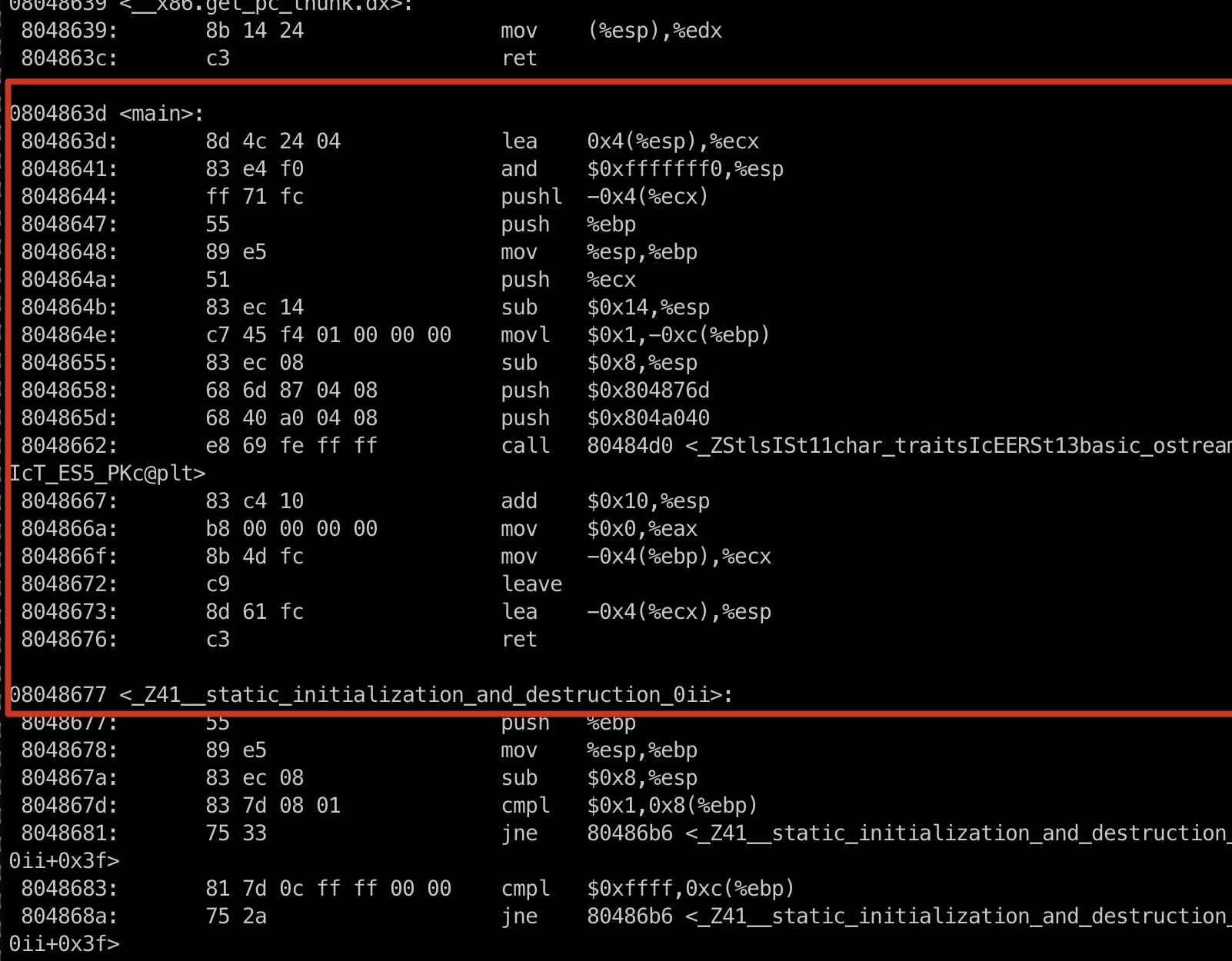


图 17 objdump反编译后的main函数代码图

* Section .rodata：这些节区包含只读数据，这些数据通常参与

[15] .rodata PROGBITS 08048760 000760 000020 00 A 0 0 4

由此可以得出

1. offset = 0x760
2. size = 0x20 = 32
3. flag = A => 进程运行期间占据内存 => 只读

这里的只读数据为输出内容

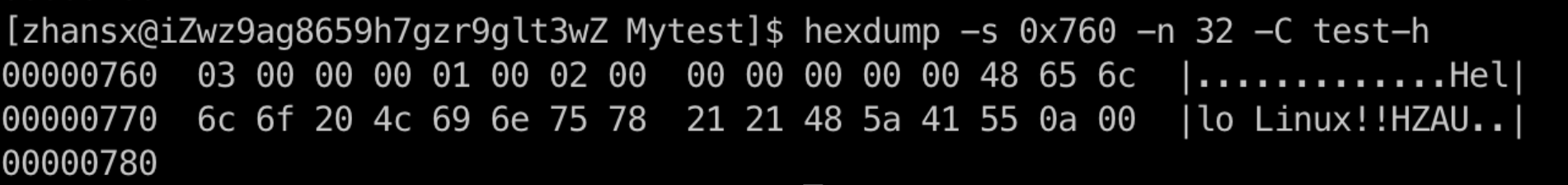


图 18.rodata节区物理地址信息图

结合代码可以发现，该部分存储的为

Cout << “Hello Linux!! HZAU”

的内容

* Section .comment：这个 Section 包含了版本控制信息。

[25] .comment PROGBITS 00000000 001024 00002d 01 MS 0 0 1

1. offset = 0x1024

2. size = 0x2d = 2\*16 + 14 = 46

3. flag = WA => 进程运行期间可写 + 进程运行期间占据内存 => 可读写

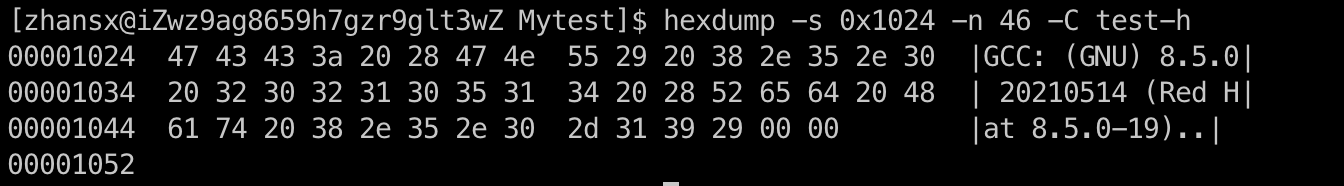


图 19.comment节区物理地址信息图

##### 5.2.3 选项- l的测试与样例解释

使用./main test-h -l 查看节区信息

文本

描述已自动生成

图 20程序头表信息展示图

* Flg标志的含义：
  + R:可读段
  + W:可写段
  + X:可执行段
* Align对齐方式的含义：
  + 段对齐,表示段在文件和内存中的对齐方式。常见的对齐方式有:
  + 0x1:1字节对齐
  + 0x2:2字节对齐
  + 0x4:4字节对齐
  + 0x8:8字节对齐
  + 0x10:16字节对齐
  + 更大的对齐数表示更严格的对齐方式。
  + 对齐方式会影响段在文件和内存中的起始地址,起始地址会是对齐数的整数倍。

如上可知，程序段的名字的内容如下：

1. PHDR:程序头表段，描述ELF文件中的程序头表段。
2. INTERP:程序解释器名,描述动态链接器的路径名。
3. LOAD:可加载段,描述可映射到内存中的段信息。

这里有两个LOAD段,第一个是只读段,第二个是读写段。

1. DYNAMIC:动态链接信息,描述动态链接信息。
2. NOTE:注释段,包含一些注释信息。
3. GNU\_EH\_FRAME:异常帧信息,包含堆栈中的异常处理信息。
4. GNU\_STACK:堆栈段,标记堆栈的可执行性。
5. GNU\_RELRO:只读数据段,标记的数据段中的只读部分。

##### 5.2.4 选项- d的测试与样例解释

# 6结论与体会

## 6.1结论

在本次项目中，我们设计并实现了一个ELF文件解析器，该解析器能够解析ELF格式的文件，我作为小组长，在进行小组实训项目策划的同时，在我的部分我实现了-h、-S、-l和-d指令的功能。

在项目设计中，我将ELF文件解析器划分为几个功能模块，每个模块负责处理特定的指令。首先是-h指令，它允许用户查看ELF文件的文件头信息。我实现了解析文件头的功能，提取出了文件的基本信息，如文件类型、入口地址、段表偏移等，并将这些信息以易读的方式展示出来。再来是-S指令，它用于显示ELF文件的节表。我通过解析节头表，将文件中的各个节的信息提取出来，并以表格的形式呈现给用户。用户可以查看节的名称、偏移地址、大小等重要信息，从而更好地了解ELF文件的结构。至于-l指令，它用于显示ELF文件的重定位信息。在实现这个功能时，我解析了重定位节和符号表，将重定位的目标符号和重定位类型提取出来，并展示给用户。这样用户可以查看ELF文件中的重定位信息，了解符号和地址之间的关系。最后是-d指令，它用于显示ELF文件的动态链接信息。我实现了解析动态节和动态符号表的功能，提取出动态链接库的名称、版本等重要信息，并展示给用户。通过这个功能，用户可以了解ELF文件的动态链接情况，以及与其他动态链接库的依赖关系。

## 6.2个人体会

“纸上谈来终觉浅，绝知此事要躬行”从理论到具体实现。首先我对ELF文件格式进行了深入的学习和理解。ELF文件是一种常见的可执行文件和目标文件格式，它包含了程序在连接和执行过程中所需的重要信息。通过学习ELF文件格式，我对文件结构、节、段、符号表等概念有了更清晰的认识。

通过本次实训实现这些功能，我的ELF文件解析器能够帮助用户更好地理解和分析ELF文件。用户可以通过指定相应的指令，查看文件头信息、节表、重定位信息和动态链接信息，从而深入了解ELF文件的结构和属性。通过实现不同指令的功能，我提高了自己的编程和算法设计能力。解析ELF文件涉及到复杂的数据结构和算法，例如解析节表、符号表和重定位表等。通过编写高效的解析代码，我不仅加深了对这些概念的理解，还提升了自己的编程技巧。

在项目的过程中，我遇到了一些挑战和困难。首先是理解和解析ELF文件格式的复杂性，需要仔细阅读规范和文档，并理解不同部分之间的关系。其次是设计和实现解析器的算法和数据结构，以便高效地解析和提取ELF文件的信息。我在这些方面花费了大量的时间和精力，进行了反复的学习和实践，最终成功地完成了解析器的设计和实现。

在团队协作方面，通过这个项目，我锻炼了团队合作和沟通能力。在项目中，我与团队成员进行了讨论和分工，确保每个人都能贡献自己的技能和专长。我们共同解决了一些技术难题，共享经验和知识，使项目得以顺利完成。与此同时，我们使用Git工作流，将工程项目进行开源共享，并且更加方便我们团队成员进行修改和溯源，采用Github工作流对于我们的项目来说具有重要意义。它提供了一个集中、协作和可追踪的平台，使得团队成员可以更加高效地合作，管理和追踪项目的进展，并与其他开发者进行交流和分享。通过这种方式，我们能够更好地组织和管理我们的项目，提高工作效率，以及促进项目的成功完成。

最后，我通过编写项目总结和报告的过程，通过进行PPT答辩以及撰写报告，进一步强化了自己的沟通和表达能力。通过将项目的设计和实现过程以及取得的成果进行详细的描述和总结，我不仅可以向他人清晰地展示我的工作，还可以反思和总结自己的经验和教训，以便在今后的项目中改进和提升。

总而言之，通过完成ELF文件解析器的设计和实现，我不仅实现了-h、-S、-l和-d指令的功能，还获得了深入理解ELF文件格式和程序链接的能力。同时，我也提高了编程和算法设计能力，锻炼了团队合作和沟通技巧，以及加强了自己的报告撰写以及表达能力。这个项目为我今后的学习和职业发展打下了坚实的基础，让我更加自信和有能力应对更复杂的编程任务和挑战。

**致谢：**

感谢任老师和李老师的陪伴！老师辛苦了！

# 7参考文献

1. 陈逸鹤. 程序员的自我修养. 北京：清华大学出版社，2017. 40-168
2. ELF文件详解. 爱笑的林羽. <https://blog.csdn.net/qq_42875402/article/details/106963323>
3. （美）布赖恩特（Bryant,R.E.）. 深入理解计算机系统(原书第3版). 北京：机械工业出版社，2016. 70-280
4. ELF文件格式. By7e\_f@lc0n. <https://blog.csdn.net/m0_73644864/article/details/129899412>
5. 计算机那些事. 楚权的世界. http://chuquan.me/
6. elf-parser. marler8997,TheCodeArtistmarler8997. https://github.com/TheCodeArtist/elf-parser