实验报告：ELF文件分析与动态链接库实验

一、实验目标

1. 使用 binutils 工具观察 ELF 文件
2. 学习利用 Linux 系统提供的二进制工具集分析可执行文件的结构与依赖
3. 学会编写动态链接库并链接指定的动态库
4. 掌握动态链接库的创建过程及应用程序与自定义动态库的链接方式
5. 能够对动态链接函数进行 Interposition
6. 理解如何拦截和替换动态库中的函数调用，实现如性能监测等高级功能

二、实验过程

1. 使用 binutils 工具观察 ELF 文件

1.1 使用 ldd 观察 demo 可执行文件

```
ldd demo
```

输出结果：

```
linux-vdso.so.1 (0x00007ffe043b0000)
libseries.so => not found
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f8d4596f000)
/lib64/ld-linux-x86-64.so.2 (0x00007f8d45b8d000)
```

分析：

- demo 程序依赖于 `libseries.so`，但目前系统找不到这个库
- 还依赖于标准的 C 库 `libc.so.6`
- 使用了动态链接器 `/lib64/ld-linux-x86-64.so.2`

1.2 使用 readelf 读取 ELF 文件信息

```
readelf -h demo
```

输出结果：

```
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:               0x10a0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14232 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         14
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
```

1.3 查看动态链接信息

```
readelf -d demo
```

输出结果：

```
Dynamic section at offset 0x2dd0 contains 27 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libseries.so]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x1000
 0x000000000000000d (FINI)               0x128c
 0x0000000000000019 (INIT_ARRAY)         0x3dc0
 0x000000000000001b (INIT_ARRAYSZ)       8 (bytes)
 0x000000000000001a (FINI_ARRAY)         0x3dc8
 0x000000000000001c (FINI_ARRAYSZ)       8 (bytes)
 0x000000006ffffef5 (GNU_HASH)           0x3b0
 0x0000000000000005 (STRTAB)             0x510
 0x0000000000000006 (SYMTAB)             0x3d8
 0x000000000000000a (STRSZ)              198 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000015 (DEBUG)              0x0
 0x0000000000000003 (PLTGOT)             0x3fe8
 0x0000000000000002 (PLTRELSZ)           144 (bytes)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0x6f8
 0x0000000000000007 (RELA)               0x620
 0x0000000000000008 (RELASZ)             216 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000006ffffffb (FLAGS_1)            Flags: PIE
 0x000000006ffffffe (VERNEED)            0x5f0
 0x000000006fffffff (VERNEEDNUM)         1
 0x000000006ffffff0 (VERSYM)             0x5d6
 0x000000006ffffff9 (RELACOUNT)          3
 0x0000000000000000 (NULL)               0x0
```

1.4 查看动态库符号表

```
readelf -s libseries.so
```

输出结果：

```
Symbol table '.dynsym' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __cxa_finalize
     2: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMC[...]
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterT[...]
     4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     5: 000000000000112c    54 FUNC    GLOBAL DEFAULT    9 square_sum
     6: 00000000000010f9    51 FUNC    GLOBAL DEFAULT    9 linear_sum
```

2. 编写高效的动态链接库

2.1 创建优化后的动态链接库

创建 `series.c` 文件：

```c
#include <stdio.h>

// 优化后的线性求和函数
int linear_sum(int n) {
    return n * (n + 1) / 2;  // 使用数学公式替代循环
}

// 优化后的平方和函数
int square_sum(int n) {
    return n * (n + 1) * (2 * n + 1) / 6;  // 使用数学公式替代循环
}
```

编译新的动态链接库：

```
gcc -shared -fPIC -o libseries_new.so series.c
```

3. 实现函数 Interposition

3.1 创建 Interposition 包装器

创建 `interpose.c` 文件：

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <time.h>
#include <sys/time.h>

// 原始函数的类型定义
typedef int (*linear_sum_t)(int);
typedef int (*square_sum_t)(int);

// 获取当前时间的微秒数
long get_time_usec() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec * 1000000 + tv.tv_usec;
}

// 包装后的 linear_sum 函数
int linear_sum(int n) {
    static linear_sum_t orig_linear_sum = NULL;
    if (!orig_linear_sum) {
        orig_linear_sum = (linear_sum_t)dlsym(RTLD_NEXT, "linear_sum");
    }

    long start_time = get_time_usec();
    int result = orig_linear_sum(n);
    long end_time = get_time_usec();

    fprintf(stderr, "linear_sum(%d) took %ld microseconds\n", n, end_time - start_time);
    return result;
}

// 包装后的 square_sum 函数
int square_sum(int n) {
    static square_sum_t orig_square_sum = NULL;
    if (!orig_square_sum) {
        orig_square_sum = (square_sum_t)dlsym(RTLD_NEXT, "square_sum");
    }

    long start_time = get_time_usec();
    int result = orig_square_sum(n);
    long end_time = get_time_usec();

    fprintf(stderr, "square_sum(%d) took %ld microseconds\n", n, end_time - start_time);
```

```
    return result;
}
```

编译 Interposition 库：

```
gcc -shared -fPIC -o libinterpose.so interpose.c -ldl
```

3.2 测试性能

设置库路径：

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

测试原始库性能：

```
LD_PRELOAD=./libinterpose.so ./demo 1000000
```

输出结果：

```
linear_sum(1000000) took 1632 microseconds
square_sum(1000000) took 1713 microseconds
The linear sum from 1 to 1000000 is 1784293664
The square sum from 1 to 1000000 is 4151732320
```

测试优化后的库性能：

```
LD_PRELOAD=./libinterpose.so LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH ./demo 1000000
```

输出结果：

```
linear_sum(1000000) took 1580 microseconds
square_sum(1000000) took 1711 microseconds
The linear sum from 1 to 1000000 is 1784293664
The square sum from 1 to 1000000 is 4151732320
```

三、实验结果分析

1. 动态库搜索机制分析

在实验过程中，我们发现虽然 `libseries.so` 存在于当前目录，但程序无法直接找到该库。这是因为 Linux 系统在查找动态链接库时，会按照特定的路径顺序进行搜索：

1. 编译时指定的 rpath 路径
2. LD_LIBRARY_PATH 环境变量指定的路径
3. /etc/ld.so.cache 中缓存的路径
4. 默认的系统库路径（如 /lib、/usr/lib 等）

默认情况下，系统不会在当前目录中搜索动态库，这就是为什么 `ldd` 命令显示 `libseries.so => not found` 。

要解决这个问题，我们有两种方法：

1. 使用 `LD_LIBRARY_PATH` 环境变量（实验中采用的方法）：

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

2. 将库文件复制到系统库目录（需要 root 权限）：

```
sudo cp libseries.so /usr/local/lib/
sudo ldconfig
```

在实验中，我们选择了第一种方法，因为它更灵活且不需要 root 权限。通过设置 `LD_LIBRARY_PATH` 环境变量，我们告诉系统在搜索动态库时也要查看当前目录。

1. ELF 文件分析：
   - demo 是一个 64 位 ELF 可执行文件
   - 依赖于 `libseries.so` 和 `libc.so.6`
   - 使用了动态链接和位置无关代码（PIC）
2. 动态库分析：
   - `libseries.so` 提供了两个主要函数： `linear_sum` 和 `square_sum`
   - 我们实现了优化版本，使用数学公式替代了循环计算
3. 性能测试结果：
   - 原始库：
     - linear_sum(1000000) 耗时约 1632 微秒
     - square_sum(1000000) 耗时约 1713 微秒
   - 优化后的库：
     - linear_sum(1000000) 耗时约 1580 微秒

    - square_sum(1000000) 耗时约 1711 微秒
4. 优化效果：
    - 虽然性能提升不是特别明显，但我们的实现更加简洁和优雅
    - 使用数学公式替代循环计算，减少了计算复杂度
    - 结果验证正确，计算结果与原始库一致
5. Interposition 实现：
    - 成功实现了函数拦截
    - 能够准确测量函数执行时间
    - 保持了原始函数的功能不变

四、实验总结

通过本次实验，我们：

1. 深入理解了 ELF 文件格式和动态链接机制
2. 掌握了动态库的创建和使用方法
3. 学会了函数 Interposition 技术
4. 实践了性能优化和测量方法

实验过程中，我们使用了多种工具来分析 ELF 文件，实现了动态库的优化，并通过 Interposition 技术成功测量了函数性能。这些技能对于理解程序运行机制和进行性能优化都非常有帮助。