SO 文件抽取及加载器实现

Author: ThomasKing

一、前言

学习 SO 文件格式和 linker 已有很长一段时间，现实现 SO 文件的抽取以及相应的加载器，目的在于学习总结，并对之前帖子未涉及到的知识进行补充。当然，为了让总结更有意思，先从文件格式的定义上入手，自定义一个简单的文件格式。为了保持平台上的兼容性和实现的简单性，在对 SO 抽取的过程中，保持与指令集相关的相对结构，不涉及与平台相关的指令集重定位等。后文在此格式基础上，实现一个类似 linker 的加载器，对抽取后的 SO 文件进行加载。限于水平，难免会有错误和疏漏之处，请各位大大斧正。

二、SO 文件抽取

1、文件格式定义

形形色色的文件格式大量充斥在电子设备中，若不是出于对某方面的学习研究，相信各位读者不会去看如此晦涩的文件格式定义。当然，ELF 文件格式也不例外。为了规避对代码指令的调整，完成对 SO 文件的抽取，首先定下骨架。

以 ARM 平台为例，存在如图所示的寻址方式。若调整一下某一 section 的相对地址，则需调整相应 text 中涉及到的指令 LDR、B 系列指令。

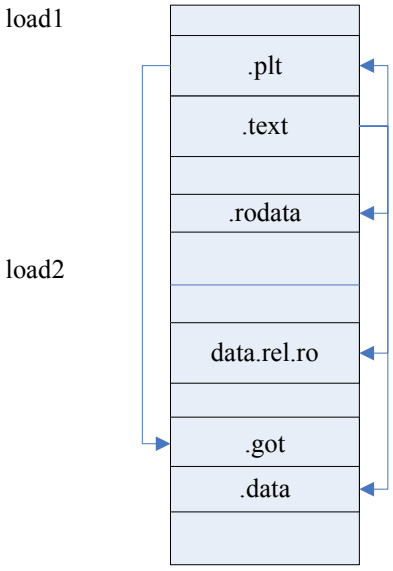

图 1 寻址示意图

类似其他文件格式，首先定义文件头部分：

```
#define   MAG0        'M'
#define   MAG1        'E'
#define   MAG2        'L'
#define   MAG3        'F'
```

```
#define    MAG          "MELF"
#define    MAGSIZE      4
typedef struct _Elf32_MiniEhdr{
    unsigned char      e_magic[MAGSIZE];
    Elf32_Word         e_filesz;
    Elf32_Word         e_memsz;
    Elf32_Word         e_DYNvaddr;
    Elf32_Half         e_DYNnum;
    Elf32_Half         e_EDXnum;
    Elf32_Word         e_EDXvaddr;
}Elf32_MiniEhdr;
```

魔术 magicnum[4] = {‘M’, ‘E’, ‘L’, ‘F’}(这个按个人喜好即可)。

e_filesz 和 e_memsz 分别表示文件大小和 bss 预留后的文件大小，即将骨架中的 LOAD1 和 LOAD2 合并。由于起始 vaddr 为 0，也不需要 vaddr 来保存虚地址。当然，也可以只用一个字段，只保存总的大小即可。

e_DYNvaddr、e_DYNnum 表示 dynamic segment 虚地址和大小;当定义为 ARM 平台时，e_EDXvaddr 和 e_EDXnum 表示 exidx 的虚地址和大小，这部分和原 SO 文件相同。目的是为了利用 linker 现有的一些函数(如 dlsym 等)，抽取后的 SO 文件需要保存一些必要的格式。若不需要，则可随意变化。

Strtab 和 symtab 结构保持不变，故加载后可以使用 dlsym 进行符号查找。Strtab 中只保存依赖 SO 文件名和导出符号的名字，symtab 仅保存自定义的导出符号，过滤掉系统生成的导出符号，后续抽取实现详细介绍。为了降低复杂度，虚地址和大小仍然在 dynamic 存储。

重新定义 rel 和 plt.rel，将 rel 中的导入符号归入 plt.rel 中，plt.rel 的结构定义如下：

```
typedef struct _Elf32_MiniPltRel{
    Elf32_Addr     r_offset;
    Elf32_Word     r_soIndex;
} Elf32_MiniPltRel;
```

r_offset 仍保存需要重定位的虚地址，r_soIndex 保存依赖库的编号。Rel 的重定位方式保持不变，而 plt.rel 采用预重定位，即在抽取时预先根据依赖库，找到虚地址重定位；加载时，仅需在此虚地址上加上依赖库的起始地址，其过程就与 rel 重定位类似了(细心的读者可能已经发现，由于不同版本源码编译生成的 SO 导出符号不尽相同，那么预重定位就依赖目标设备了，不具有通用性。这里这么做的目的，仅仅是为了说明一种修改方式，抛砖引玉，至于怎么修正或者采用更好的其他方式，那就靠读者实现了)。

2、SO 文件抽取

上面大概描述了抽取的内容，抽取后的效果图如下：

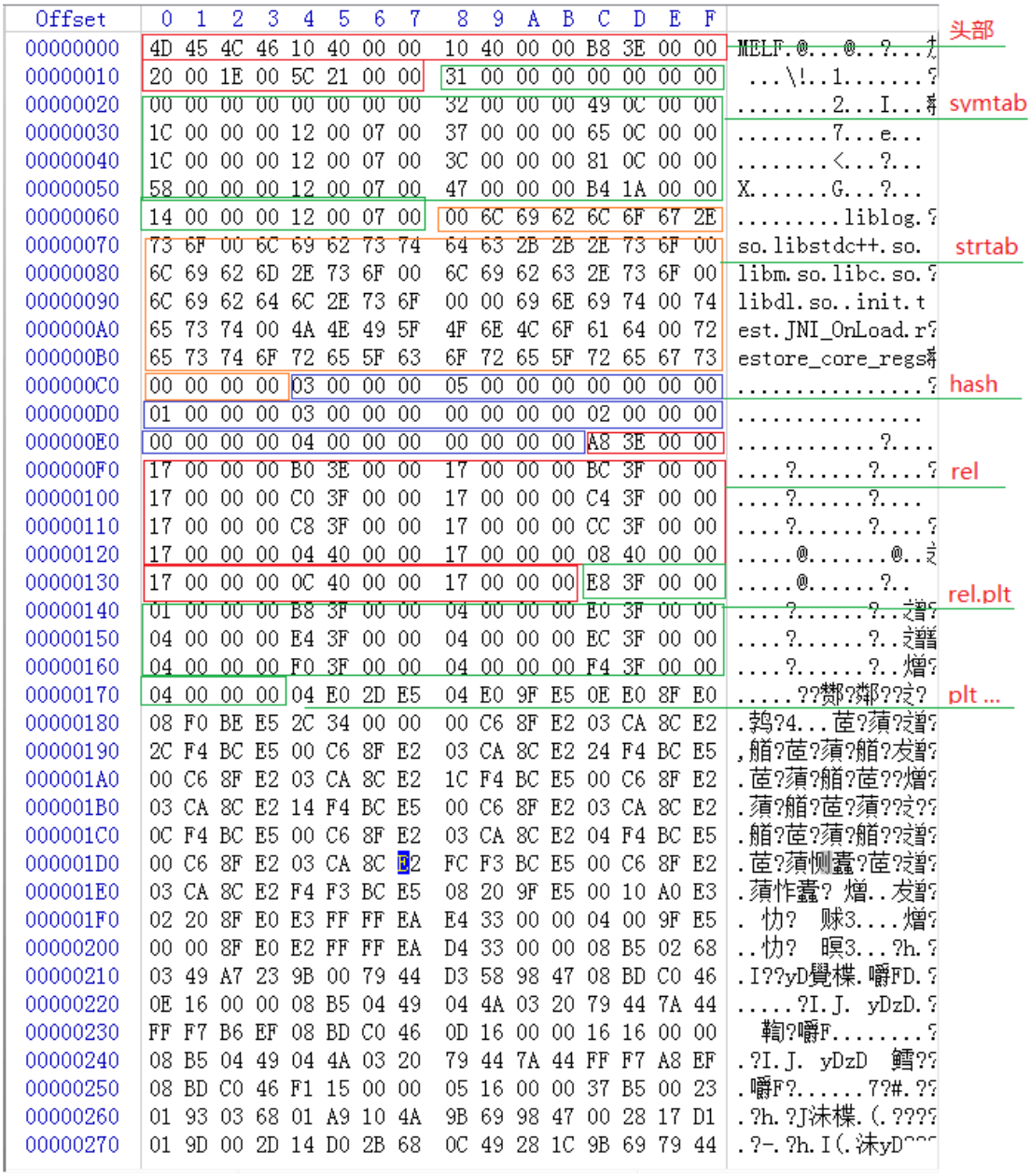

图 2 抽取后的文件格式

一般编译生成的 SO 文件 symtab 中存在许多以'_'和'__'导出符号(图 3)，这些符号并不是用户所导出的，故可以将其过滤掉(当然，strtab 和 hash 都需要相应修改，后面介绍)。

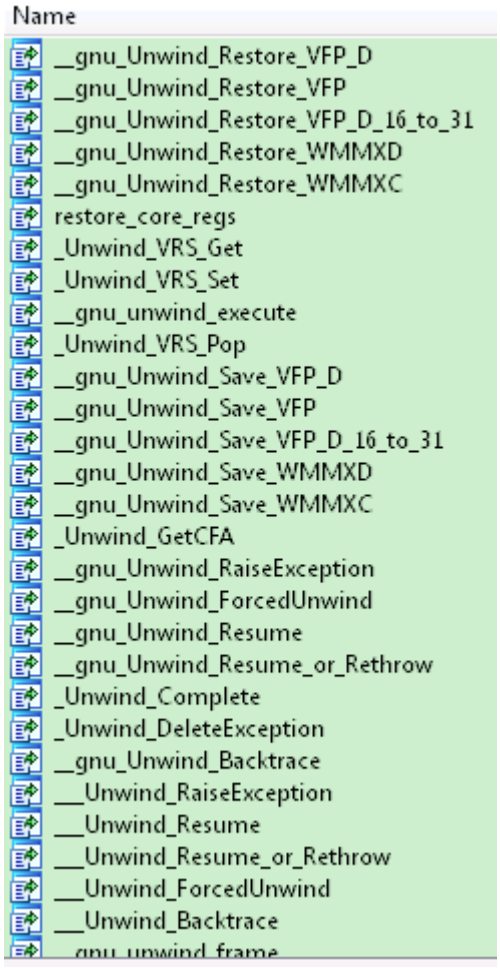

图 3 系统导出符号

过滤方式如下：

```
for(i=1;i<si->nchain;i++){       //nchain = nsym
   Elf32_Sym *sym = (Elf32_Sym*)(si->symtab + i);
   const char *name = si->strtab + sym->st_name;
   if(sym->st_shndx != 0){      //不能过滤导入符号
      if(name[0] != '_'){
         memcpy(new_symtab + new_sym_count, sym, sizeof(Elf32_Sym));
         new_sym_count ++;
      }
   }
}
```

Strtab 定义结构定义为：前面存放依赖函数库名，后面存放导出符号名，如图 2 所示。dynamic 中 DT_NEEDED 保存了依赖库名在 strtab 中的偏移，故后续需要对此修正。

通过之前的学习，hash 表由 nbucket、nchain、bucket 和 chain 四部分组成。从 linker.c 中知道，寻找符号时，通过计算 hashval % nbucket 加快符号查找过程。另外，hash 算法本身 bucket 值的选取是有一定讲究的。通过各种谷歌后，在 BFD 中查到答案：

```
static const unsigned elf_buckets[] ={
    1, 3, 17, 37, 67, 97, 131, 197, 263, 521, 1031, 2053, 4099, 8209,
    16411, 32771, 0
```

```
};
```

根据 elf_buckets 数组选取大于符号数的最小值，算法即：

```
static unsigned findBestBucket(unsigned nsyms){
    unsigned i, best_size;

    for (i = 0; elf_buckets[i] != 0; i++){
        best_size = elf_buckets[i];
        if (nsyms < elf_buckets[i + 1])
            break;
    }
    return best_size;
}
```

根据 linker 中的符号查找即可得到 hash 表的生成算法如下(具体参看源码：Mini_elf_generate.c)

```
    for(i=1;i<new_sym_count;i++){
      unsigned hashval = elfhash(si->strtab + new_symtab[i].st_name) % new_nbuckets;
      if(t_bucket[hashval] == -1){
        new_bucket[hashval] = i;
      }else{
        new_chain[t_bucket[hashval]] = i;
      }
      t_bucket[hashval] = i;
    }
```

对于 rel，其结构不变，仅将其中的导入符号提取出，归入 plt.rel 中。对于 plt.rel 中的导入符号，查找其所对应的依赖库索引，保存在 r_soIndex。最后在按照 r_soIndex，对 plt.rel 进行排序，便于后续 strtab 的生成(过程较为繁琐，详见源码 Mini_elf_generate.c)

后续的骨架部分直接拷贝即可，但需要做好虚地址修正：

1. Rel 和 plt.rel 保存了需要重定位的虚地址，由于文件变小，需要修正此虚地址
2. 由于 strtab 等重新生成及文件结构变化，dynamic 中的所有项需要修正。
3. Symtab 中的 st_value 函数地址，构造、析构函数的虚地址发生变化，故也需要对 st_value、init_array 等中保存的虚地址进行修正。

修正部分比较繁琐，限于代码长度，源码 Mini_elf_generate.c 中生成文件时，保持了 plt 等结构原有的虚地址，故仅只需修正 dynamic，简化修正流程。(上面没提及到的内容请参看源码。代码写得弱，请各位读者轻拍)

三、加载器

根据 linker 的源码可知，SO 加载的过程大致分为如下阶段：将 SO 文件映射到内存并生成相应的 soinfo 结构；对符号进行重定位；最后调用构造函数。借鉴 linker 加载流程，对抽取后的 SO 文件加载也分为上述三大步。加载过程即可放在加载器的 init_array 中，也可放入 JNI_OnLoad。放入 init_array 中需要一些额外处理(比如对抽取后的 SO 中 JNI_OnLoad 的调用时机)，这里为了简单起见，加载过程放入 JNI_OnLoad 函数中。

1) 文件映射

读取文件 Elf32_MiniEhdr，根据其 mem_sz 进行加载。由于将骨架中的 LOAD1 和 LOAD2 进行了合并，故一并加载时需要注意内存权限，即：

```
mmap(NULL, ehdr.e_memsz, PROT_READ | PROT_WRITE | PROT_EXEC,
                MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```

2) 重定位

rel 的重定位与 linker 相同；plt.rel 的重定位则只需将依赖库文件的首地址修正虚地址即可。依赖库文件首地址不通过对/proc/self/maps 的解析获得，而是通过 dlopen 获得 soinfo *handle 对象，直接从 handle->base 获取。

3）构造函数

这部分与 linker 相同，调用 call_constructors 即可。

加载器的其他细节参看源码：Mini_elf_loader.c。加载器中使用的结构体 soinfo 保存加载信息，目的在于将其链接到 SO 链中，供 linker 查找和相关函数使用。限于篇幅，将在后续文章中介绍。

四、参考文献

linker 源码
BFD 源码
http://bbs.pediy.com/showthread.php?t=193279