工作情况总结汇报:

内容:

1.rt-thread的ELF动态加载流程

2.PC模拟对ELF动态加载的静态定制

文件内容:

dl\_elf.c,dl\_elf.h:对rt-thread解耦并进行源码解析后生成的文件内容

elf32.c,elf32.h,main.c:对ELF文件进行内存字节流部分解析生成的文件内容

dl\_elf.c,dl\_elf.h:

文件头定义了ELF相对主要的一些结构以及某些字段的定义,其中部分字段适用于使用ELF文件,是属于共有部分,被标注为ELF32的部分仅仅适用于32位设备所使用.

DL\_CHAR4 dlmod\_load\_shared\_object(dlmod\_t \*module, void \*module\_ptr);

DL\_CHAR4 dlmod\_load\_relocated\_object(dlmod\_t \*module, void \*module\_ptr);

上述为被dlmodule调用的俩个接口,从参数看:module为模块数据结构,module\_ptr指向以字节流载入到内存中的ELF文件,整个功能的目的是,以载入的文件内容去初始化module的数据结构,将ELF中与程序执行有关的部分实现一个动态的迁移,这个在执行时临时进行的动作被称为动态加载,执行完毕后module结构内拥有了该模块执行所有相关的内容.

以dlmod\_load\_shared\_object为例,解析ELF文件从最开始的部分进行,那么ELF文件最开始就会有一个sizeof(Elf32\_Ehdr)大小的空间用于存放对应类型的一个数据集合,内部包括了以下几个重要的字段(相当的部分字段源码注释中包含进去了):

e\_phoff; //程序头表偏移量

e\_shoff; //段头表偏移量

这俩个偏移量可以求得程序头表和节头表所在位置,它们都是相对文件起始位置布局的,当文件被载入到内存时(起始位置为:elf),对应的内存存储地址计算方式即为:

Elf32\_Ehdr \*ehdr = (Elf32\_Ehdr \*)(elf);

Elf32\_Shdr \*shdr = (Elf32\_Shdr \*)(elf + ehdr->e\_shoff); //内存布局是一个数组

Elf32\_Phdr \*phdr = (Elf32\_Phdr \*)(elf, ehdr->e\_phoff); //内存布局是一个数组

ELF32\_1BYTE2 sidx = ehdr->e\_shnum; //段(节)头表项

ELF32\_1BYTE2 pidx = ehdr->e\_phnum; //程序头表项

原rt-thread的dlelf模块会先遍历程序头表项:

for (index = 0; index < pidx; index++)-->phdr[index]

将所有phdr[index].p\_type == PT\_LOAD的段找到并载入到module中,而且载入时也会大致按原内存布局载入

之后便开始进行重定位的工作了,遍历节头表:

for (index = 0; index < sidx; index++)-->shdr[index]

找到所有shdr[index].sh\_type == SHT\_REL的段,这里存放了一个重定位项,和上述求相对文件起始位置布局的动作一致:

Elf32\_Rel \*rel = (Elf32\_Rel \*)(elf + shdr[index].sh\_offset); //内存布局是一个数组

其中有俩个关联连接相对有用:节头表有一个字段sh\_link会将所需的其余节关联起来,形成引用链,通过link1 = shdr[index].sh\_link即可获得符号表的索引,那么shdr[link1]就是符号表,那么再进行一次动作link2 = shdr[link1].sh\_link即可获得字符串表的索引,那么shdr[link2]就是字符串表,当找到重定位项数组后,同样的遍历该数组,而这里的项数目是通过:shdr[index].sh\_size / sizeof(Elf32\_Rel)求得的,遍历数组时使用rel->r\_info中附带的索引信息获得存储在符号表数组中的哪一个下标中,取出对应符号,然后进行重定位.这里如果重加载失败后还会进入主程序导出的符号表中寻找符号,用于做地址替换,但这个动作是rt-thread早期使用到的方式,未知原因的被遗弃掉了,那么使用PC段在外部完成这个动作就是下一个要去完成的事情.完成重定位后,module就会获得一个包括主程序绝对内存地址的各个符号的更新后的符号表.之后处理.dynsym段,处理这个段时没有指定的索引,无法直接寻找到对应的节所在的位置,从ehdr->e\_shstrndx获得段头自己使用的一个字符串表段,通过shdr[ehdr->e\_shstrndx]获得该字符串表段,之后通过:

DL\_BYTE1 \*shstrab = elf + shdr[elf\_module->e\_shstrndx].sh\_offset;

获得节自用字符串表的存储地址,那么每一个节的.name信息就保存在此处,而节中保存的就是对当前字符串表段起始位置产生的相对偏移量,所以使用(shstrab + shdr[index].sh\_name)即可获得各个节保存的名字的起始地点,然后一一使用strcmp功能进行比对,寻找到唯一的.dynsym段,寻找到该段后同样使用:link1 = shdr[index].sh\_link,link2 = shdr[link1].sh\_link获得符号表和字符串表的索引并获得对应的符号表和字符串表,遍历这个.dynsym的符号表中的每一个符号项,将符号项信息中带有标记为:STB\_GLOBAL或STT\_FUNC的符号纪录下来,以相似的方式从字符串表获得符号名称以及符号相对地址,而符号的相对地址会通过从载入内存布局中的程序段起始位置开始,添加symtab[index].st\_value的相对偏移转化为绝对地址,那么这里就是该符号使用的位置

整个加载流程以载入程序段,更新重定位项,载入重定位符号表为主,完成后程序即可从对应的内存地址处开始执行.这是加载程序代码段的部分.而另一个函数dlmod\_load\_relocated\_object则将程序中的.text(文本段),.rodata(只读数据段),.data(数据段),.bss段加载进入内存中,和上述的解析流程相似的方式,遍历整个节头表,寻找满足:shdr[index].sh\_type和shdr[index].sh\_flags满足对应值的段,被标记为上述的各个数据段,这些段是程序执行时所需要操作的目标,而加载则就是将各个段拷贝到指定生成的内存空间即可,设置程序执行的入口函数,然后再一次复刻上述的流程解析可重定向段,遍历可重定向段的符号表中各个符号,满足标记!= STN\_UNDEF且STT\_SECTION或STT\_OBJECT时,然后确定时指定的四个段中的哪一项,然后获得对应的地址,然后对该地址重定向即可,这里的重定向是使用到了dlmod\_relocate.程序执行需要的俩大数据节集合载入后,整个加载流程告一段落,而其他的数据节,则不参与程序的执行.属于辅助使用的部分.

早期在载入程序段中,处理可重定向段时,做了一个额外的检查,用于跳过内核符号表关联这一个动作,实际的动作也就是在外部完成这一个加载的动作,那么因为缺失各种资源,本次尝试二次加载时,采用的是切块化的做法,先导出内核符号表,然后利用导出到内存中的内核符号表去重定向ELF文件的各个符号,之后修改该ELF文件的起始四个字节的标记,即可完成该动作,而目前研究到了第一部分将近末尾的部分并为第二部分完成的大致的流程,具体见elf32.c中的流程:

void elf32\_export\_ksymtab(ELF32\_1BYTE1 \*elf, Elf32\_KSymtab \*ksymtab);

可完成对主内核ELF文件中的指定段解析并提取对应的内容,载入到内存中,目前整个流程在PC模拟时进行到了:(//搜索指定段名的段)因为这里缺少一个可以满足提取内核符号表的对应Demo,那么下面的流程暂时未验证,但有把握基本的方向没有错误,简要调试做部分细微调整基本上就可以了,这是第一部分的工作进行到的位置

void elf32\_load\_ksymtab(ELF32\_1BYTE1 \*elf, Elf32\_KSymtab \*ksymtab);

可将第一部分解析提取出的内核符号表以传参的方式载入,对指定的模块elf文件进行修饰,整个流程也是按rt-thread的源码复刻了一遍,基本的方向没有大问题,目前唯一比较担心的地方是,对可重定位项的相对地址使用内核符号表的绝对地址修饰这一部分,是摘出的rt-thread的dlmod\_relocate的代码,其中有部分逻辑并不是很清楚,各个场景中也未可知实际上使用到了哪一种,只有经过联合调试确定后才知道具体情况,这是第二部分的工作进行到的位置