Linking

1 基本概念

.c 格式的源码通过预处理器、编译器和汇编器，得到了.o 格式的可重定位目标文件（relocatable object file）。此类文件中，变量和函数可以只有声明没有定义。链接器的角色即是将多个目标文件组合成为可执行文件。

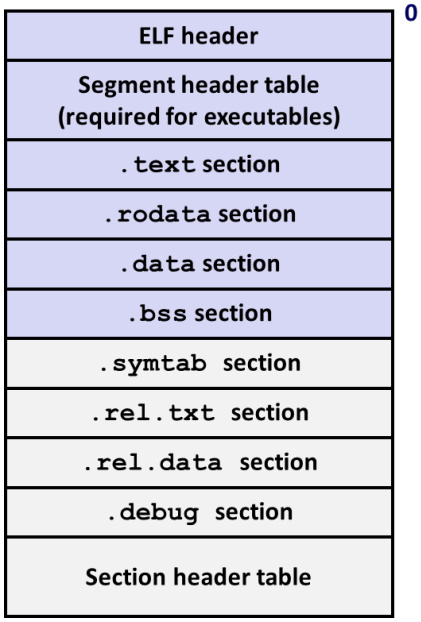
链接器使得程序具备了以下特性：

- 模块化：对复杂的程序，分不同文件放置，便于管理；
- 效率：重编译时，仅需要对一部分重编译及重链接，节省时间；通用函数归入同一文件，节省空间。

2 链接器的原理

2.1 目标文件的类型

链接器可处理的文件格式实际上不仅限于.o 文件，还包括可执行文件和.so 格式的动态链接库。它们都参照 ELF（可执行、可连接格式）的标准进行编码。



图：ELF 分段格式

©Randy Bryant, CMU

2.2 链接器的工作

程序中的符号（symbol）意为函数名与变量名。函数的定义、原型及变量定义被视为符号的定义，函数调用及变量引用被视为符号的引用。

链接器的工作分为两步：解析目标文件中的全局符号、查找每个符号的定义位置；以及将符号地址重定位。

2.2.1 解析符号

链接器不处理函数内部的局部变量。它只处理三类符号：本文件定义的全局符号（public）、本文件声明但在外部定义的全局符号（extern）、仅在本文件使用的静态符号（static）。这些符号被保存在 ELF 的.symtab（symbol table）段落中。该段落保存了符号的名称、占用大小及内存位置参数。解析过程中，链接器试图使每一处符号引用均指向一处（且仅一处）符号定义。

链接器处理歧义符号（不同文件中同名的非静态符号）的规则如下：

- 如果仅 1 个进行初始化，则进行初始化的符号作为 public，其余作为 extern。
- 如果 0 个进行了初始化，则链接器随机挑选一个作为 public，其余作为 extern。即便被定义为不同的类型：若一个文件定义为 int、另一定义为 double，则一处修改，另一处将读到意外值。
- 如果超过 1 个进行了初始化，则报错 link time error。

为了避免上述的意外情况，使用全局变量应注重以下规则：

- 尽量避免使用；
- 如必须使用，尽量使用 static；
- 如必须跨文件使用，尽量初始化；
- 如果使用外部变量，则用 extern 声明。

2.2.2 重定位

在待链接的目标文件中，符号的地址均是指向以 PC 为基准的相对地址。在重定位操作中，链接器将各目标文件的数据段和代码段合并，然后将所有符号的相对地址都改为绝对地址，最后基于此更新程序中所有符号引用的地址。

3 静态和动态链接库

链接库提供了一种将常用的函数打包存放的解决方案。包括静态和动态两种。

3.1 静态链接库

静态链接库文件格式为.a，它将多个目标文件编制索引，并集结存放于一个档案（archive）中。链接时，可指示链接器在档案中寻找未解析的符号定义。当链接器发现档案中某个目标文件中含有定义，则将其链接至可执行文件中。

在命令行链接指令中使用静态链接库时，需要注意将库放在命令行末尾。因为链接器只按从左到右的顺序解析符号。如果右边的目标文件完成了符号解析，发现了未定义符号，这时便不会再回到左边的库里再进行解析，于是报错。

3.2 动态链接库

静态库在使用上有如下不便之处：库每次进行版本更新时，程序员都要进行显式的重链接；且若有多个进程均使用到同一静态库，则库中的代码段会在每个进程实例中产生副本，造成内存空间的资源浪费。动态链接库解决了这些问题。

动态链接库文件格式为.so。它在装载时（load-time）或运行时（run-time）才会被加载到内存中，而每个进程实例的共享库段（位于堆和栈之间）仅保存指向这些内存地址的映射。如此即实现了资源共享；且更新时仅需要更新库，不再需要重链接。

由于动态链接库的优势，静态链接库实际上已经过时。

3.3 使用链接库

gcc 编译如需包含库，则使用-Lpath -lxxx 标记。加入该标记后，编译器将在 path 目录下搜寻名为 libxxx 的库，优先搜寻 libxxx.so，然后搜寻 libxxx.a。若为静态库，则将库链接入可执行文件。若为动态库，则运行时将在该目标位置搜寻动态库。

运行时如需加载动态库，则调用 dlopen 方法：

```
void * dlopen(const char * pathname, int mode);
```

第一个参数指定动态库的文件路径，需要在编译时用-l 标记指明。第二个参数指明是否立刻计算库的依赖性。如果设置为 RTLD_NOW 的话，则立刻计算；如果设置的是 RTLD_LAZY，则在需要的时候才计算。另外，可以指定 RTLD_GLOBAL，它使得那些在以后才加载的库可以获得其中的符号。

该方法返回一个句柄。调用该方法后，可以用 dlsym 方法通过该句柄获得库中特定符号（函数或变量）的指针：

```
void * dlsym(void * handle, const char * symbol);
```

第一个参数是此前获得的句柄，第二个参数为符号的名称字符串。

动态库使用完毕后，调用 dlclose 方法，传入句柄，即可将其卸载。

使用 gcc 将一些代码编译为动态链接库，需要加入-shared 标记。一般还会加入-fPIC 标记，表示生成与地址无关的代码，以便于加载时的重定位操作。

4 示例：函数捕获

函数捕获，是指使程序在运行时，调用库函数时改为执行程序员自定义的函数。这种技术对于程序安全及调试均有所作用。

以下提供了实现函数捕获的三种方式：编译时、链接时和装载/运行时。以对 malloc 函数的追踪为例：将原函数中对 malloc 的调用改为对一个自定义函数的调用，而后者调用实际的 malloc 库函数的同时，也打印该函数所分配的内存地址。

4.1 编译时捕获

malloc 函数对库的调用在 malloc.h 头文件中。因而可以编写一个自定义的 malloc.h 头文件，在其中定义如下宏：

```
#define malloc(size) mymalloc(size)
```

然后编写一个含有 mymalloc 函数的自定义程序 mymalloc.c。由于其中调用了库中实际版本的 malloc，因而要将 mymalloc.c 预先编译为目标文件 mymalloc.o。

最后，在对原程序进行 gcc 编译时加入-I.标记，表示**在程序目录下，而非默认的标准目录下，搜索头文件**（关键步骤）；并且将 mymalloc.o 一并链接入内。由此，原程序中的 malloc.h 头文件将使用自定义的版本；从而原程序中对 malloc 函数的调用，通过上述定义的宏，均转为对 mymalloc 函数的调用。

4.2 链接时捕获

gcc 编译时，可以通过-Wl 标志传入链接选项。**通过--wrap,malloc 链接选项，可以将原函数中对 malloc 的调用，改为对名为__wrap_malloc 函数的调用；而对实际 malloc 函数的调用则应通过调用__real_malloc 完成**。

因此，在 mymalloc.c 中，加入下述代码：

```
void *__real_malloc(size_t size);


/* malloc wrapper function */
void *__wrap_malloc(size_t size)
{
    void *ptr = __real_malloc(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```

然后在链接时，加入-Wl,--wrap,malloc 标志，即实现了对 malloc 函数的替换。

4.3 运行时捕获

unix 提供了 LD_PRELOAD 这一环境变量，它指示**在执行某一程序时，预先加载一个指定的动态链接库**。

因而，在运行程序前，首先编写自定义版本的 malloc，并编译为动态链接库 mymalloc.so；在编译原程序（假设编译为 intr）之后，设置环境变量如下：

```
setenv LD_PRELOAD "./mymalloc.so"; ./intr
```

由此原程序中对 malloc 的调用，便将指向预加载的 mymalloc.so 中的 malloc。