2022.11.9 第十次小班课

张荟萱

linking很重要

我们接下来使用一个真实的场景尝试阐述这个问题：假设我们有一个理想的多层网络协议，每一层只需要用到下一层的方法，而最终用户看到的一般是最高层的方法。

不同层的方法想必应该实现在不同的文件里，这样我们才能比较容易地切换每一层的实现却不影响别的层的效果。

在最底层我们实现的方法在packetio.c里，在中间层我们实现的方法在ip.c里，在最高层我们实现的方法在socket.c里。

可是...我们怎么在ip.c里调用packetio.c里的东西呢?

最直接的方法:

```
//ip.c
//...
void func_from_ip()
{
    //...
    func_from_packetio();
    //...
}
//...
```

```
//packetio.c
//...
void func_from_packetio()
{
    //contents
}
//...
```

```
gcc ip.c packetio.c -g -o ip
```

这work吗?

这是可以工作的!

但在规范中是严格反对这样做的（编译会出现Warning）。原因是在ip.c中调用这个函数的时候，会根据函数参数的传递情况给出一个默认的隐式的声明，而这种默认的声明的返回值都是int，也就是说如果你定义的函数返回了某些长度超过int的东西那么会出现一些隐式的错误。

同时如果参数列表中出现了float，那么参数传递也可能会出现一定的问题。

解决这个问题的方法是在调用函数的地方先声明一下这个函数，也就是这样:

```
//ip.c
//...
void func_from_packetio();
void func_from_ip()
{
    //...
    func_from_packetio();
    //...
}
//...
```

然后就可以正常工作了。

但这样做会带来另一个问题——如果我需要在很多文件里调用很多外部函数，那么在每个文件里都要把所有这些声明重写一次，这实在是太麻烦了。

所以一个很自然的想法就是，我们把所有的声明写在一个地方然后粘贴到每个文件的开头就好了！

这就出现了.h文件，我们可以把所有的声明写在一个.h文件里，在所有用到这个外部函数的地方include这个文件，然后C预处理器就会把.h文件里的东西插入到文件里

也就是这样：

```
//packetio.h
void func_from_packetio();
```

```
//ip.c
#include "packetio.h"
//...
void func_from_ip()
{
    //...
    func_from_packetio();
    //...
}
//...
```

这样工作得很好。

但是我们忽略了另一个问题——如果我们想访问在另一个模块中定义的全局变量怎么办？

假设我们在ip层定义了一个路由表（这里简单理解为一个数组），而我希望自己路由器的程序能够访问这个路由表（这里简化为打印对应数组元素），这怎么办？

```
//ip.c
#include "packetio.h"
int route[10000];//路由表

void func_from_ip()
{
    //contents
}
```

```c
// router.c
#include <stdio.h>
int main()
{
    int t;
    scanf("%d",&t);
    if(t<10000&&t>0)
        printf("%d\n",route[t]);
    return 0;
}
```

```
gcc router.c ip.c -g -o router
```

这没法正常工作！因为对于全局变量是没办法做一个默认的隐式声明的，这样在编译router.c的时候就会报错！

那么如果我们类似地先声明呢？

```c
// router.c
#include <stdio.h>
int route[10000];
int main()
{
    int t;
    scanf("%d",&t);
    if(t<10000&&t>0)
        printf("%d\n",route[t]);
    return 0;
}
```

链接器会遇到一个很棘手的问题：用户访问的route...到底是router.c里的，还是ip.c里的？还是用户希望这俩是同一个？

接下来我们开始讨论知识性的内容：在Linking这个层面，要解决的问题其实简单说来就是你是谁（符号解析），你在哪（重定位）。

而我们主要讨论的linking要处理的文件是ELF文件，其格式如下：

ELF头
.text（已编译程序的机器代码)
.rodata(只读数据，全局的const常量，printf中的格式字符串，switch语句的跳转表)
.data（已初始化的全局和静态变量）
.bss（未初始化的（或被初始化为0的）全局和静态C变量）（它不占据实际的空间，运行时直接给这些变量在内存中分配空间即可）
.symtab（一个符号表，不需要强制使用-g生成）
.rel.text（一个.text节中的位置的列表，表示链接器需要修改.text中的这些位置，一般是调用外部函数或引用全局变量的指令，可执行文件中一般将其省略）

ELF头
.rel.data（全局变量的重定位信息，主要是初始化为全局变量地址或外部函数地址的全局变量）
.debug（调试符号表，必须以-g编译）
.line（原始C程序的行号与.text机器指令的映射，-g）
.strtab（字符串表，包括.symtab和.debug的符号表，节头部表中的节名字）
节头部表（不同界的位置和大小）

那么如何解决你是谁这个问题呢？汇编器要构造一个**符号表**，放在上面的.symtab节中，我们举例说明如何阅读这张符号表：

示例程序：

```
//2.c
int a;
int buf[10];
static int t=5;
static int m;
int c[2]={1,2};
void call_2()
{
    static int t=7;
    int d=8;
    printf("calling 2\n");
}
void Call_2()
{
    printf("Calling 2\n");
}
```

阅读符号表的方法：

```
gcc -c 2.c
readelf -s 2.o
```

我们会看到这样的输出：

```
Symbol table '.symtab' contains 19 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS 2.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     4 OBJECT  LOCAL  DEFAULT    4 m
     6: 0000000000000000     4 OBJECT  LOCAL  DEFAULT    3 t
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     8: 0000000000000010     4 OBJECT  LOCAL  DEFAULT    3 t.2254
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
    10: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
    11: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
    12: 0000000000000004     4 OBJECT  GLOBAL DEFAULT  COM a
    13: 0000000000000020    40 OBJECT  GLOBAL DEFAULT  COM buf
    14: 0000000000000008     8 OBJECT  GLOBAL DEFAULT    3 c
    15: 0000000000000000    30 FUNC    GLOBAL DEFAULT    1 call_2
    16: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSET_TABLE_
    17: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
    18: 000000000000001e    19 FUNC    GLOBAL DEFAULT    1 Call_2
```

一个打印出来的条目包含如下几个内容：Value、Size，Type，Bind，Vis，Ndx，Name

Name不用说，给出了这个符号的名字，Vis全是DEFAULT我们不管，关键来看看别的部分。

首先，Bind负责表示符号是局部的还是全局的。

一个符号可以被分为两种：**全局符号**和**局部符号**，其中局部符号就是由static修饰的**全局变量或函数**，你不能在别的文件里访问这个它们，就像C++的class中用private声明的成员一样。而全局符号则对应于其余的函数和全局变量。

在全局符号之中，有些符号并不在这个模块中被定义，而是在别的模块中定义而被这个模块引用，这样的符号称为**外部符号**。比如我们试图在ip.c中访问在packetio.c中定义的函数，这个函数就是一个外部符号，但**外部符号也是一种全局符号**。

值得说明的是，在一个函数内部定义的所谓"局部变量"是不出现在符号表里的（比如示例中的d），因为他们被放在栈上，函数调用时创建，函数结束时销毁，不涉及到与别的文件的交互，编译器已经处理好了他们，我们讨论的符号都是全局变量或者函数。

static的另一种行为就是，如果你在一个函数内部定义了一个static的变量，那么在你多次调用这个函数的时候，访问的是同一个变量，对这个变量的修改会累计下来，即它的行为很类似于一个全局变量。原理是被static修饰的变量不会如我们先前所言的局部变量放在栈上，而是会被放在全局的位置上，比如上面示例中的t。

但是一个问题是，我完全可以在不同的函数之中定义同名的静态变量，这怎么办？

编译器会解决这个问题，他会给在函数中定义的静态变量加上不同的后缀，比如上面我们看到的t.2254，来区分不同的局部变量（值得说明的是，结合上面的例子可以看到，全局的static变量是没有后缀的。

而可以看到我们关系的TYPE基本要么是数据（object），要么是函数（func）

Size给出了这个目标以字节为单位的大小，比如可以看到c的size=8，buf的size=40

Ndx给出了这个符号应该被放在ELF文件中的哪个节，比如call_2是一个函数，那么它显然应该被放在代码段，也就是.text节里，对应于Ndx=1；而c是一个已初始化的全局变量，应该被放在.data节中，对应于Ndx=3，而a是一个未初始化的全局变量，应该放在.bss节中...？

需要说明的是，这里还有3个特殊的伪节，在节头部表中是没有条目的，包括ABS（不该被重定位的符号），UNDEF（未定义的符号）和COMMON（未被分配位置的未初始化数据的目标）。这三个伪节只有可重定位目标文件中有，可执行目标文件中是没有的。比较confusing的是.bss和COMMON的区分，其区别如下：未初始化的全局变量放在COMMON伪节中，而未初始化的静态变量和初始化为0的全局或静态变量放在.bss中，稍后我们会看到这样做的原因。

先应用下这个规则，我们就可以看到buf和a都被放在COMMON伪节中，而m则被放在.bss中（对应于Ndx=4）

最后一个问题是value，在可重定位的模块来说，value是距离定义目标的节的起始位置的偏移，比如call_2的value=0，表示其距离.text节的开头的距离是0，而Call_2的value=0x1e=30，表示其距离.text节的开头是30，这是合理的，因为call_2的size=30。而对于COMMON来说，value给出了对齐要求，而size给出了最小的大小。

值得说明的是，单纯声明却没有定义或引用的符号是不会出现在符号表中的，比如我声明了一个函数`void f();`但我却既没有定义它的函数体，也没有在别的地方引用这个函数，那么f是不会出现在符号表里的。

请不要忘记，我们做的这一切都是为了解决“你是谁”的这个问题。对于有static修饰的符号很好处理，因为这些符号不能被别的模块引用。但是全局符号则并不好处理，链接器很可能面对的问题如我们最开始所言——他要如何得知程序想引用的是哪个route呢?

我们把全局符号分成强符号与弱符号两种，强符号是定义的函数或初始化了的全局变量，而弱符号是未初始化的全局变量或没有定义的函数。

这就是COMMON伪节的意义——放在COMMON伪节里的东西全都是弱符号，但是放在.bss节里的是强符号。

而链接器会这样解析：不允许有多个同名的强符号，如果有同名的一个强符号和多个弱符号则解析为强符号，如果有同名的多个弱符号则任意选择一个。

也就是说，我们上面的例子中的两个route属于同名的多个弱符号，也就是说它会任意地选择一个，但....更糟糕的是，链接器不检查类型，也就是说即使同名的两个符号类型不同，链接器也不会作出区分，这就导致了一些糟糕的错误，相关的例子不再讨论。

为了避免这样的错误，可以用GCC-fno-common来调用链接器来避免这样的错误，这个选项会让链接器遇到多重定义的全局符号时报错。（一些助教在新的lab machine上试验表明新的gcc需要加入-fcommon指令才会使用COMMON伪节，但我在原来的ubuntu18环境下仍然能得到和书上类似的结果。）

但我们的问题并没有解决，我们不希望自己的程序里出现多重定义的符号，那么一个类似的想法仍然是——我们把定义扔到一个地方，然后要求所有文件引用这个定义就好了!

这样我们可以写一个ip.h：

```
//ip.h
int route[10000];
```

```
//ip.c
#include "ip.h"
//...
```

```
//route.c
#include "ip.h"
//...
```

这样写有用吗?

没用！因为include的行为只是单纯把.h文件里的声明插入到文件开头，这样和我们分别在两个文件里声明是没区别的，我们仍然得到了两个弱符号！

甚至如果你在.h文件里写了一个初始化，那就相当于我定义了重名的强符号，这样会导致链接错误！

那怎么办?

我们要做的实际上是这样一件事：我们希望所有东西只在一个地方被定义，但是在所有用到它的地方它都已经被声明了，而且我们希望保证我们操作的符号是确定的同一个。

这时就要引入extern关键字。我们可以在ip.h里这样声明这个变量：

```
//ip.h
extern int route[10000];
```

这个声明的意义在于，它保证了如果我在本模块里没有定义route，那么所有对route的引用都会直接去外部查找，如果外部找不到对route的定义则会直接报错。

用一个例子来解释：

```
//示例程序
int d;
void f()
{
    d=1;
}
```

这个程序本身是完全正确的，模块就会操作自己声明的这个d，但是如果有另一个程序也声明了全局的d，那我就不知道我操作的d是哪一个了。

但如果我这样写：

```
//示例程序
extern int d;
void f()
{
    d=1;
}
```

那么链接器会直接去外部找d的定义，如果没有找到会直接报错。

这样我只需要保证我只在某个文件中定义了唯一的route，而其它文件只是include了ip.h，那么我可以保证所有模块操作的是同一个route了。

值得说明的是，没有函数体的函数声明默认是extern的

总结一下，一个可能的合理的多文件组织结构是这样的：

我们把所有的声明（全局变量声明，函数声明，乃至于结构体或者类的声明）写在一个.h文件里（比如packetio.h,ip.h），然后把所有的定义（全局变量定义，函数体）写在一个对应的.c文件里(比如packetio.c,ip.c)，所有想要引用这些全局变量或者函数的程序只需要include这些.h文件而不需要做任何额外的事情，在链接的时候将程序自己与对应.c文件汇编出的.o文件链接起来就可以实现我们想要的功能了。

此外，为了防止重复声明，我们需要确保如果这个.h文件里的东西只被复制给一个文件一次，这可以通过两种方式实现：#ifndef和#pragma once

一个例子：

```
//ip.h
#ifndef __IP_H_
#define __IP_H_
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <map>
#include <cstdlib>
#include <mutex>
#include "packetio.h"
#include "device.h"
using namespace std;
#pragma once
extern map <pair<string,uint>,int> router;//map dst_add to device number
extern map <pair<string,uint>,string> routing_tb;//map dst_add to mac_add to be
sent
extern map <pair<string,uint>,uint> routing_tab;

typedef unsigned int uint;
extern mutex g_lock;

int sendIPPacket (const struct in_addr src , const struct in_addr dest ,
int proto , const void * buf , int len, const int dev_id ,const uint identi) ;

typedef int (*IPPacketReceiveCallback) ( const void * buf , int len ,int id) ;

unsigned getipbystr(const char* s);

string getstrbyip(unsigned ip);

int receiveippacket(const int id);

int setIPPacketReceiveCallback(IPPacketReceiveCallback callback ) ;

int setRoutingTable ( const struct in_addr dest ,
const struct in_addr mask ,
const void * nextHopMAC , const char * device ) ;

int find_maskip(uint ip);

#endif
```

```
//ip.cpp
#include "ip.h"
#include "packetio.h"
#include "device.h"
#include <string>
#include <algorithm>
#include <cstring>
#include <unistd.h>
#include <map>
```

```
using namespace std;
mutex g_lock;
map <pair<string,uint>,int> router;//map dst_add to device number
map <pair<string,uint>,string> routing_tb;//map dst_add to mac_add to be sent
map <pair<string,uint>,uint> routing_tab;//map dstinfo to ip_add to be sent


int sendIPPacket (const struct in_addr src , const struct in_addr dest ,int
proto ,
const void * buf , int len, const int dev_id,const uint identi)
{


}

IPPacketReceiveCallback ipcallback=NULL;

int setIPPacketReceiveCallback(IPPacketReceiveCallback callback )
{

}

int receiveippacket(int dev_id)
{

}

uint getipbystr(const char* s)
{

}

string getstrbyip(unsigned ip)
{

}
```

（上面并不是什么标准或者规范的要求写法，只是我自己学到的一种可行的写法）

这样写的优点是，我只需修改链接的指令，就可以改变我调用的函数具体的行为，而不需要对调用这个函数的源文件做任何修改。比如我实现了两种发包的方式，分别名叫ip1.cpp和ip2.cpp，那么我只需要修改与我的文件链接的是ip1.o还是ip2.o就可以了。

当然，多个文件的编译链接同样需要大家学会写makefile，这个之后有空再讲。

接下来讲一个故事，假设你没有学好linking，你写了这样的一个头文件：

```
//ip.h(probably buggy!)
int route[10000]={1,2,3};
```

那么如果有很多文件都include这个头，linker会报错

这个报错怎么办呢？你一知半解地看了看ics，决定这样修改：

```
//ip.h(probably weird)
static int route[10000]={1,2,3}
```

请问这样做会带来什么问题?

问题就是，每个模块都开了一个自己的静态route，在每个模块中使用的route都不是同一个，因此你可能会遇到"明明我在这里编辑了路由表，为什么那里没体现呢？？？"这样的问题。

在上面的讨论中我们一直讨论的是将一组可重定位目标文件链接起来，但是在现实中我们还有很多标准的库函数要使用，那么我们如何使用这些库函数呢?

一种方法是要求编译器辨认程序使用的所有的库函数并且直接生成对应的代码，但问题是C语言中有很多标准的库函数，如果要求编译器做到这一点对编译器而言是个相当大的挑战，另一方面每一次对标准库函数的修改都会要求编译器版本的更新，这有点过于繁琐。

另一个方法就是把这些库函数都放在同一个可重定位目标模块中，然后只需要和这个模块链接即可，但问题仍然是库函数实在太多，而一个程序又很可能不会使用所有的库函数，这就导致生成的可执行文件占据了不必要的巨大的空间，这很显然也是不妙的。

另一个方法就是干脆像上面那样，我们直接把每个函数创建一个独立的可重定位目标文件，然后大家自己链接需要的东西，这同样不太妙，因为如果你调用了100个标准库函数，那么想要准确地写出链接的指令是一件相当困难的事情。

所以，我们提出了静态库的概念，所谓静态库其实就是把先前的可重定位目标模块（比如一堆.o文件）打包成一个文件，而链接器在链接时可以从这个文件中选择程序调用的模块连接起来，而没有调用的模块则不被引用，这样程序员只需要包含一个静态库，链接器就可以完成剩下的事情。在linux下，一个静态库往往是一个.a文件

但这会带来别的问题——假设某程序p.o中的某个符号在静态库x.a中定义，而x.a中某符号在y.a中定义，那么...

由于链接器只会选择程序调用的模块链接起来，也就是说，当我们引用一个静态库的时候，我希望链接器已经知道了我要从这个静态库中得到什么。

因此，当我们引用x.a时，我们希望链接器已经读取了p.o，从而得知了需要链接x.a中的哪些文件，以此类推。

于是对静态库的链接是这样进行的：链接器从前向后扫描输入的文件，同时维护三个集合E,U,D，E表示最终将被链接到一起的可重定位目标文件，U表示前面的可重定位目标文件中没有定义的符号，D表示前面的可重定位目标文件中已经定义的符号。

对于一个输入文件，如果它是一个目标文件（比如.o），那么链接器会将其加入E，同时用其中定义的符号尝试解析U中的符号并将其中定义的所有符号加入D

对于一个存档文件.a，我们会根据U中所有符号检查其中是否有一个目标模块能够解析一个未定义符号，如果有这样的模块m.o，那么我们就对m.o重复上述输入文件的操作，重复这一过程直到U和D不再变化为止。

如果扫描完所有文件U仍然是非空的（即仍然存在未被解析的符号），那么链接器报错，否则将E中所有文件合并和重定位输出最终的目标文件。

这样一个很直观的要求是，如果p中的符号是在q.a中定义的，那么p必须出现在q.a之前，不然在扫描到q.a时，它无法预测到后面某个p会用到自己的某个符号，这样它不会把对应的目标模块加入E，从而p的符号无法正确解析（但如果都是.o文件则没有这个问题，因为.o文件会把自己定义的符号都加入D中，这样无论先后都可以正确解析。）

可是这又会带来新的问题——循环引用呢?

假设p.o中引用了一个x.a中模块m.o，而m.o引用了y.a中的一个符号n.o，n.o又引用了x.a中的符号q.o，这样无论x.a写在前面还是y.a写在前面，都没办法正确解析所有的符号！

但这没关系，因为我们可以在命令行上重复库，我们可以按这个顺序写：p.o x.a y.a x.a，这样所有符号就能够正确解析了。

我们用了这么大的篇幅，尝试解决”你是谁“这个问题，接下来我们要想办法解决”你在哪“这个问题。

回顾一下，对于局部变量我们很清楚地知道它在哪——它就在栈上，由编译器决定把它放在栈上的什么位置，直接输出相对栈指针的偏移量就可以了。

但是对于全局变量，我们...我们不知道它在哪——我们有一堆文件，每个文件定义了一堆全局变量，它们都在自己的.data或者.bss之类的地方，但是我们最后只需要一个文件，这个文件只有一个.data段，所以链接器要先把这一堆节合并成一个节（这个合并过程是有序的，改变顺序会改变这些全局变量在内存中出现的位置），接下来链接器将运行时的内存地址赋给新的聚合节，赋给输入模块定义的每个节，赋给输入模块定义的每个符号，这样程序中的每条指令和全局变量都和唯一的运行时内存地址对应起来了。

但是还有一些问题——在代码段中我们会访问全局变量，在编译时我们不知道这个全局变量在哪，我们留了一个重定位信息交给链接器解决，链接器必须解决这个问题。同样，数据段中有些全局变量被初始化为了别的全局变量的地址，我们在编译时也不知道这个地址是什么，链接器必须正确填写数据段的这些部分。

链接器依赖于汇编器生成的重定位条目来做到这件事，这里我们只讨论两种重定位方式：X86_64_PC32（重定位32位PC相对地址的引用）和X86_64_32（重定位32位绝对地址的引用）

一个重定位条目一般包含如下几个部分，首先是offset，表明被重定位的对象离节的开头有多远。然后是type表明重定位方式（我们这里只讨论PC32和32），然后是symbol表明重定位的符号是谁，最后是一个addend，其中最让人困惑的可能就是这个addend

我们直观地来讨论这个问题：考虑一个重定位条目r，其所在的节是s，对于PC相对引用，我们需要计算出引用的目标与下一条指令的地址之差，而引用的目标的地址已经由链接器确定了，我们记为$Addr(r.symbol)$。

而下一条指令的地址是什么呢？当前被重定位对象的位置是由其所在节的开头加上一些偏移量决定的，即$Addr(s)+r.offset$，但这只是指出了当前被重定位对象的位置，我们假设要重定位对象的位置与下一条指令的PC之差为$|\delta|$，那么这里计算出的PC相对值应该是$Addr(r.symbol)-(Addr(s)+r.offset+|\delta|)$，而这里的$r.addend$实际上就是这个$-|\delta|$，也就是要重定位的位置的地址-下一条PC的地址，于是就有了教材上伪代码里的计算公式。

至于重定位绝对引用，我们还是知道引用的目标地址就是$Addr(r.symbol)$，那么道理上我们填入这个值就可以了？

有一个小问题——如果我是在访问一个数组元素呢？如果我在访问一个数组元素，只填入这个值我们永远只能找到数组开头。

这是我们不想看到的，所以我们在重定位绝对引用时，我们实际填入的重定位值是$Addr(r.symbol)+r.addend$，这样我们就能找到数组中的元素了。

在解决了所有这些问题之后，我们就完成了链接，得到了一个可执行目标文件，一个ELF格式的可执行目标文件格式如下：

ELF头
段头部表

ELF头
.init（定义了一个小函数_init，程序初始化代码会调用）
.text（已编译程序的机器代码）
.rodata(只读数据，全局的const常量，printf中的格式字符串，switch语句的跳转表)
.data（已初始化的全局和静态变量）
.bss（未初始化的（或被初始化为0的）全局和静态C变量）（它不占据实际的空间，运行时直接给这些变量在内存中分配空间即可）
.symtab（一个符号表，不需要强制使用-g生成）
.debug（调试符号表，必须以-g编译）
.line（原始C程序的行号与.text机器指令的映射，-g）
.strtab（字符串表，包括.symtab和.debug的符号表，节头部表中的节名字）
节头部表（不同界的位置和大小）

ELF可执行文件很容易加载到内存，可执行文件连续的片被映射到连续的内存段，程序头部表描述了这种映射关系（教材上的程序头部表可以用objdump -x得到）

```
    LOAD off    0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**21
         filesz 0x0000000000000878 memsz 0x0000000000000878 flags r-x
    LOAD off    0x0000000000000db8 vaddr 0x0000000000200db8 paddr 0x0000000000200db8 align 2**21
         filesz 0x0000000000000258 memsz 0x0000000000000260 flags rw-
```

如何解读这个程序头部表？这两个LOAD分别对应于从可执行文件中加载的只读内存段（代码段，从ELF头到**.rodata**）和读写内存段（数据段，.data和.bss），第一个off表示其在**目标文件中的偏移量**，而vaddr和paddr表示其开始的内存位置（教材上提到代码段总是从0x400000处开始，但我这里反映的可能由于版本关系有一定差异），而align是对齐要求，filesz表示目标文件中的段的大小，而memsz表示加载到内存中需要的段大小，多出来的部分是用于给初始化为0的.bss数据分配的。最后的flag是标志位，其中r表示可读，w表示可写，x表示可以执行，-则表示没有这些权限，即代码段的东西可读、不可写、可执行，但数据段的东西可读可写不可执行，而align表示对齐要求，即要求
$vaddr\%align = off\%align$

而将可执行目标文件加载到内存时，会把代码段加载到**0x400000开始的地址**处，接下来是数据段，由于数据段的对齐要求代码段和数据段之间会有间隙。地址空间随机化（ASLR）会随机运行时堆、共享库和用户栈的位置，只保持它们的相对位置不变。

一个C程序的入口点是_start函数的地址，在系统文件ctrl.o中定义，_start函数调用系统启动函数__libc_start_main，该函数定义在libc.o中，它初始化执行环境，调用用户的main函数。

我们讨论了很多静态库相关内容，但是这并不足以满足我们的要求——静态库需要定期维护和更新，而如果静态库被更新了，所有用户就必须重新下载新的静态库并且重新将调用了这些静态库的程序重新链接，不然就会带来错误，但这样做成本是非常高昂的。

另一个问题是很多C程序使用相同的标准库函数，我们没必要把这些东西在每个可执行文件中都存储一份副本，我们希望能够在执行时再将我们需要的部分加载到内存。

这就产生了共享库。共享库是一个目标模块，在**运行和加载时可以加载到任意的内存地址，并且和一个在内存中的程序链接起来**，这个过程称为动态链接。在Linux系统中用.so后缀标识共享目标文件。

共享库有两种共享方式，首先是对于一个库只有一个.so文件，所有引用这个库的可执行目标文件共享这个.so文件中的代码和数据，而不是像静态库那样把.so文件里的内容复制到可执行文件本身之中。此外，一个共享库的.text节的一个副本可以被不同的进程共享。

那么，动态链接是如何进行的呢?

在链接时，链接器并不将.so文件中真正的代码和数据拷贝到可执行文件中，而是复制了一些重定位和符号表信息，而在加载器加载和运行可执行文件时，加载器加载的是部分链接的可执行文件，这样的文件中会包含一个.interp节，这一节包含动态链接器的路径名（动态链接器本身就是一个共享目标，比如linux系统上的ld-linux.so)，加载器首先加载和运行这个动态链接器，这个动态链接器重定位共享库.so文件的文本和数据到某些内存段，然后重定位可执行目标文件中的引用，最后开始执行。

而共享库的另一个功能就是对于同时运行的多个进程，我们可以共享同一个副本。但这是如何做到的?一部分机制涉及到虚拟内存的问题，这里不做讨论。另一部分机制是我们要讨论的：一个共享库在不同的进程中可能被映射到不同的虚拟地址上，那么我们要如何动态地访问共享库中的全局变量和函数呢?

我们使用一种位置无关代码（PIC）的机制来解决这个问题，共享库的编译必须总是使用PIC机制。

一个事实是，无论我们在内存中的何处加载目标模块，代码段和数据段的距离是一定的，这样的话我们可以为每个目标木块生成一个全局偏移量表（GOT），这个表中的每个表项对应于一个被这个目标模块引用的全局数据目标，编译器为GOT中的每个条目生成一个重定位记录，加载时动态链接器重定位每个条目使之获得目标的正确的**绝对地址**，每个引用全局目标的目标模块都有自己的GOT，在访问数据的时候只需要将自己GOT表对应自己要访问的表项读出（这一点在链接时就能完成），读出的就是正确的全局变量的地址了。

而对于函数调用，我们没必要把一个共享库里面定义的所有函数都重定位一次，那样做太浪费了。而就算我们只重定位我们会调用的函数，也是一个很大的工作量，会导致加载时间变得很长。所以我们使用一种延迟绑定（lazy bind）机制。具体来说，在GOT表之外，我们引入PLT表，每个PLT表的表项是一个16byte的短代码，PLT[0]跳转到动态链接器，PLT[1]调用系统启动函数，而其余的PLT表项对应于程序调用的不同的共享库的函数。

其余每个PLT表项的行为是一致的——它们首先尝试跳转到GOT表对应的表项。而对于函数调用，GOT[0]和GOT[1]对应于动态链接器在解析函数地址时要用的信息，而GOT[2]是动态链接器在ld-linux.so模块中的入口点，其余表项对应于PLT的表项。

那么具体的行为是什么呢？假设我想调用共享库的函数f，那么我会先进入f对应的PLT表项，然后跳转到f对应的GOT表项，在第一次调用时，f对应的GOT表项就指向PLT表项的下一项，那么实际上就是继续执行了PLT中的下一行，这个下一行是把调用函数（f）的ID压入栈中，然后跳转到PLT[0]，而PLT[0]通过GOT[1]把动态链接器的参数压入栈中，然后通过GOT[2]跳转到动态链接器，动态链接器可以确定想执行的f的地址，然后把对应的GOT表项重写成这个地址，然后正常执行f

而当之后再执行f的时候，我们仍然进入对应的PLT表项，然后跳转到对应的GOT表项，但是这时GOT表项已经是正确的函数地址了，这样我们就可以正常执行这个函数。