静态链接之符号解析

当我们组件一个大的项目时，需要将多个源文件编译之后得到的可重定位目标文件结合在一起，这一过程不仅仅是将代码和数据简单聚合在一起这么简单。因为很多文件中的数据和函数是全局可见的（在其他模块中依旧可以使用），所以实际上我们在只需要在其中一处进行定义，在别的模块下通过符号引用真实的内容。所以简单来说链接最核心的工作就是在本模块下通过一张表记录下所有全局数据，之后为每个引用位置定位到实际定义数据的地方。

在我看来，通常写C语言程序有三类最主要的错误：一是语法错误，二是逻辑错误，三是链接错误。链接错误在日常程序错误中占了很大的比重，所以学习链接的意义还是很大的。另外，链接本身处于编译器、计算机体系结构和操作系统的交叉点上，要彻底说清楚需对上述三门课程知识有比较全面的了解。

本文将会借助一个短小的C语言例子贯穿整篇文章，全部的分析将围绕下面的代码展开。

// main.c

extern void swap();

int buf[2] = {1, 2};

int main()

{

    swap();

    return 0;

}

// swap.c

extern int buf[];

int \*bufp0 = &buf[0];

static int \*bufp1;

void swap()

{

    static int temp;

    bufp1 = &buf[1];

    temp = \*bufp0;

    \*bufp0 = \*bufp1;

    \*bufp1 = temp;

}

代码本身很短，原本可以放在一个源文件中，不过为了说清楚关于链接的内容，将它们拆分在两个独立的模块中，一个是main.c一个是swap.c。在main.c模块中定义了buf数组，这个数组是全局变量，除了本main.c模块中的函数可见外，所有与其链接的文件中的函数也均可以对其引用，该模块下有关键字extern用来表示引用其他全局数据，此处是引用全局函数swap()。在swap.c文件中，首先通过extern引用了全局变量buf，并通过全局变量指针bufp0得到了指向buf[0]的指针，通过本地静态变量bufp1得到了指向buf[1]的指针，二者的区别不仅在一个是全局变量一个是本地全局变量，另外bufp0变量在运行前被初始化，bufp1是在程序执行过程中被赋值的，需要思考的是bufp0是什么时候通过什么方式得到buf[0]元素的运行时地址的。另外，swap.c模块中还定义了局部静态全局变量static，我们在接下来也会比较二者的区别。

在讲解链接之前，首先我们需要分解编译的过程，了解编译过程可以分为几个阶段以及各个阶段产生的中间过渡文件。可以通过使用gcc时加-v选项查看编译完整的过程。

1. 预处理阶段：预处理阶段运行预处理器（cpp）将.c结尾的源文件变成.i结尾的中间文件，其中.i文件相比于.c文件最大的区别在于.i文件将头文件中的内容插入到了.c文件中。
2. 编译阶段：运行编译器（ccl）将上面生成的文件编译生成汇编代码，这步是整个编译过程最困难的一步，生成的中间文件是汇编文件，以.s结尾。
3. 汇编阶段：汇编阶段将上面生成的汇编语言逐句转换成机器语言（由as完成）。当这步执行完之后，实际上生成的.o结尾的文件已经是机器语言级别的了，相比于最终的执行文件最大的区别在于每条指令还没有实际的运行时地址，也没有将引用的符号解析成具体数据定义的地址。
4. 链接阶段：输入文件是多个.o结尾的二进制文件，通过链接器（ld）将它们链接成完整的可执行文件，生成运行时地址，完成符号解析工作。

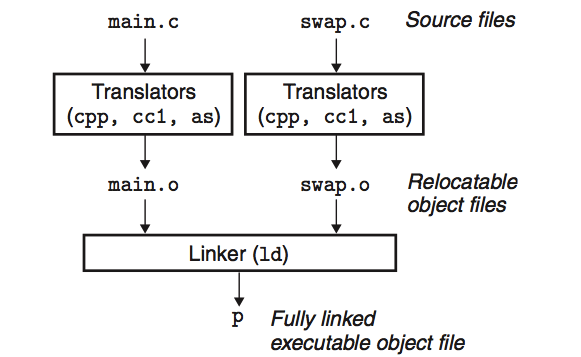


图1：编译流程图

在上面的描述中，我可以避开了二进制文件的区别。简而言之就是完成汇编阶段之后生成的二进制文件和最终链接之后得到的二进制文件具体有哪些区别，以及它们的名字又是什么，因为接下来全文在描述中都会遵照这一术语来，所以还是有必要先规范下生成文件的名称。我们将这些通过了汇编之后的二进制文件统称为目标文件，目标文件有三中类型：一是经过汇编的各个模块，此时它们叫做可重定位目标文件，这个时候的目标文件的起始地址是从0开始的，它并不能被执行。二是共享目标文件，和是上面提到的可重定位目标文件没有什么太大区别，最主要就是共享目标文件可以被共享。三是可执行文件，可执行文件就是完成了符号解析和重定位的二进制文件。

通过上面的简要介绍，大致可以得知链接阶段的两个核心环节：符号解析和重定位。

在说清楚符号解析之前首先得清楚什么叫做符号，当我们定义和引用全局变量和全局函数时，定义某个全局变量（函数）的模块需要记录下这些数据的名字和基本信息，引用别的模块定义的内容同样需要被记录下，当然，这里的记录是需要通过字段类型来区分的。而记录下这些内容的表就称为符号表。要说清楚符号表是什么又的先从目标文件的格式说起，必须先知道目标文件各个部分的意义才能接下去讨论。

在linux下，现代目标文件使用的是ELF格式，虽然可重定位目标文件和可执行文件都是ELF格式的，但二者间还是有一定的差异。

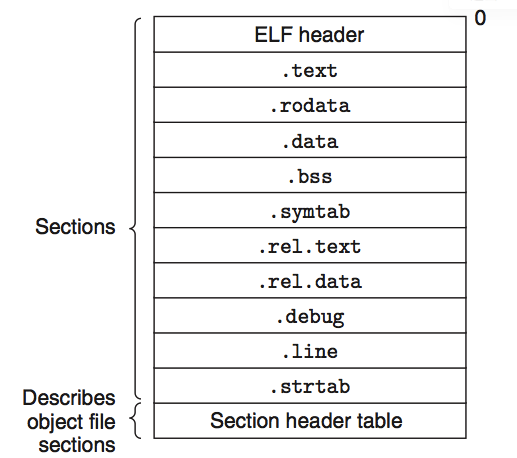
首先附上一张可重定位目标文件的ELF布局图。

图2：可重定位目标文件ELF

该文件对应的是源文件完成编译的前三个步骤后得到的内容，可重定位目标文件最上部分是ELF头，记录了系统架构、数据大端小端存储方式、程序入口、文件类型等，都是一些和底层相关的基本信息。下图展示的就是main.o文件的ELF头部分，可以通过readelf –h [可重定位目标文件]的方式来看到ELF头内容。

图3：可重定位目标文件ELF头

接下来是由多个称之为节的部分构成，下面我来简要介绍下各个节的内容及其作用。

.text节内容是二进制机器指令部分，也就是我们常说的文本段中的内容。

.rodata节中的内容是程序中只读字符串内容，比如printf中控制字符的内容。

.data已初始化的全局变量，如果某个模块定义了全局变量并且对其进行了初始化，则在其生成的可执行目标文件文件中的.data段中就有这个全局变量值。

.bss相比与.data节最大的区别就是在.data节中存储的是未初始化的全局变量。

.symtab节存储的是可重定位目标文件的符号表，符号表记录了全局变量、函数的一些信息，在进行重定位时需要用到。

.rel.text记录了.text节中位置的列表，重定位时需要用到。

.rel.data记录了被模块定义和引用的全局变量的重定位信息。

其他几个节在本篇文章中暂时用不到，所以不再一一说明。

既然上面已经出现了符号表，我们有必要着重介绍一下符号表。最先需要讨论的是到底哪些符号是需要被记录在符号表中的？记录在符号表中的有三类信息：一是在某个模块中定义为全局变量并可被其他模块所引用的变量（数据）的符号，二是在某个模块内引用定义在别的模块下的全局变量（函数）的符号，三是定义在某个模块下并且只能被该模块所引用的静态全局变量（函数）的符号。

上述第三种就是我们常说的带有static关键字的变量，它们虽然只能被本模块下的函数使用或调用，但依旧是定义在.data和.bss区，甚至在函数内定义静态变量，依旧是存储在上述区域内，而不是运行时在栈中分配。

指导了符号表中需要存储的符号有哪些，接下来我们通过真实的符号表来介绍符号表中各个段意义。下图是分别是main.o和swap.o的符号表内容。

图4：main.o符号表

图5：swap.o符号表

我们需要关注的内容有main.o符号表中num号为8、9、10的三个符号：buf、main、swap,以及swap.o符号表中num号为5、6、10、11、12的五个符号：bufp1、temp.1483、bufp0、buf、swap。

其中size列表示该符号所指代数据的大小，在main模块中定义了buf数组，其大小为8。在swap模块中定义了bufp0、temp、bufp1和swap（函数），大小分别是4、4、4、56。这些都是在某个模块内被定义的部分，它们都具有实际的大小。符号表中除了定义部分还有引用部分，引用符号虽然也在符号表中具有相应的条目，但是大小却为0，比如main模块符号表中的swap函数和swap模块中的buf变量。可见，模块表中记录的符号如果是定义在该模块下则具有实际的大小（分配了空间），引用的话不会分配空间，大小为0。

Type列代表了符号的类型，OBJECT表示变量，FUNC表示函数，NOTYPE表示是引用（所有引用无论变量函数都为NOTYPE）。Bind列表示符号的作用域，LOCAL表示只能在本模块内，对应的是静态数据，GLOBAL表示全局变量，对应的是全局变量和函数。Ndx列表示的是该符号实际数据存储在哪个节中，其中Ndx=3表示在.data节中，Ndx=1表示在.text节中，Ndx=5表示在.bss节中。

总结上面提到的内容，我们可以用三个具有代表性的完整例子结束对符号表的讨论。

1. buf数组是定义在main模块中并被swap模块所引用，大小为8字节存储在main.o文件的.data节中的全局变量。
2. swap函数是定义在swap模块下并被main模块所引用，大小为56字节存储在swap.o文件的.text节中的函数。
3. bufp0 是定义在swap模块中且只能被本模块引用大小为4字节的存储在swap.o文件的.bss节中的静态全局变量。

符号解析的主要任务是将每一个符号引用和一个具体的符号定义联系起来，就是说如果在某个模块内引用了某一个定义在其他模块下的变量，需要将二者联系起来。

在编程过程中，我们常因为疏忽给不同的变量或函数起了同样的名字，大部分情况下编译器会报错并终止，但是还是会有部分情况会通过最终的链接，但是最终结果很可能出乎我们预期，所以需要明确一下具体是如何操作的。因为引用并未实际分配存储，在链接后实际存取的依旧是在其他模块定义的数据，所以接下来讨论的重名（重定义）问题只讨论定义时重名的情况。

首先说的是全局变量和全局函数，全局变量我们按照其是否初始化将其分为强符号和弱符号，初始化的全局变量的符号是强符号，未初始化的全局变量的符号是弱符号。当多个模块中出现重名的强符号时，编译器报错。当强符号和弱符号冲突时，选择强符号。多个弱符号冲突时任选一个。

再者是静态全局变量，静态全局变量相比全局变量很特殊，因为static已经将各个模块的冲突域隔离开，所以不同模块下的的静态全局变量是可以重名的且相互不影响，所以我们讨论静态全局变量时考虑重名的范围是在同一模块内而不是各个模块间。同样地，如果同一模块内定义了重名且已初始化的静态全局变量是不能通过编译的。如果重名静态全局变量有一个初始化了，有一个没有初始化，则选择已初始化的。多个未初始化的重名静态全局变量不详。所以说编译器对静态全局变量符号重定义给出的解决方案和全局变量多模块间重定义的解决方案类似。

但是静态变量还有更让人生惑的内容，那就是静态局部变量。做一个简单的区分就是静态全局变量是某个模块内且定义在所有函数之外并被static关键字修饰的变量，静态局部变量就是某个模块内某函数内部定义的且被static修饰的变量。二者都是分配在.data区（初始化）或.bss区（未初始化），也就是说它们和全局变量实际上存储的位置都是相同的。但各个函数内定义的静态局部变量在编译时被解析成变量名+数值形式，如果有重定义现象，编译器会自动选取不同值保证唯一性，所以静态局部变量在重定义方面基本可以理解成不同的栈变量，和静态全局变量不会出现重定义问题，多个函数间重名的静态局部变量也不会出现重定义现象。

当然了，为了防止在大项目中出现上述出现了符号重定义却被编译器通过编译的情况，我们可以在使用gcc编译时候加上-fno-common，这样当出现多重定义时，编译器会输出警告信息。

到这里本文初步介绍了编译过程和中间生成的文件、对全局变量、静态全局变量、静态局部变量的存储位置以及解析方式有了更深的了解。另外还介绍了编译器在部分默认执行方式，这样在我们编程遇到某些错误时就能通过原理找出问题。不过静态链接的重定位部分本文尚未涉及，写一篇文章将会描述链接中重定位以及进程虚拟存储器映像的问题。