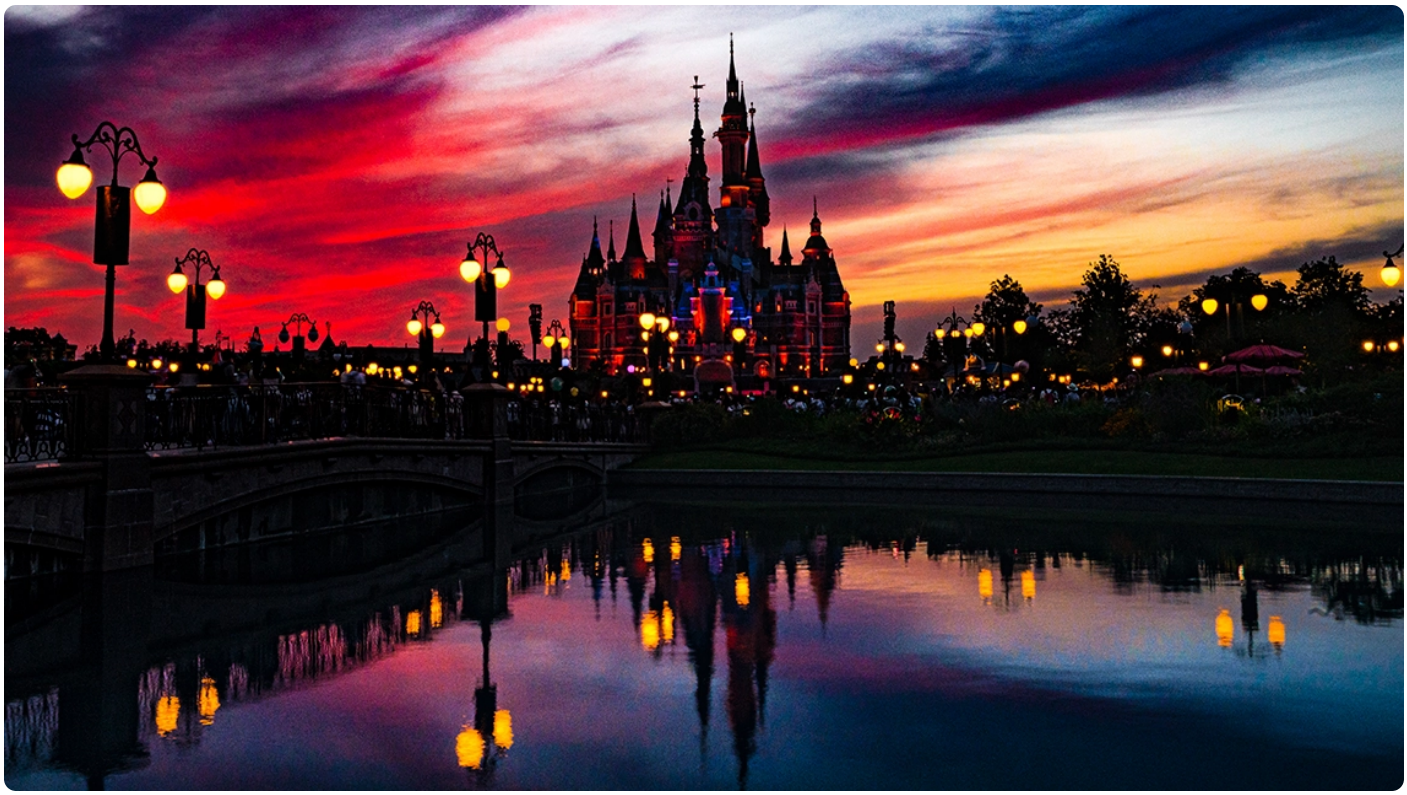
27 | 编译器在链接程序时发生了什么？

2022-02-25 于航



讲述：于航

时长 13:51　大小 12.70M

你好，我是于航。

我曾在 01 讲 的最后提到，C 代码的完整编译流程可以被分为四个阶段：代码预处理、编译优化、汇编，以及链接。在前三个阶段中，编译器会对输入的源代码文件依次进行分析、优化和转换，并生成可以在当前平台上使用的对象文件。紧接着，链接过程可以将程序依赖的所有对象文件进行整合，并生成最终的二进制可执行文件。

今天，我就来带你深入看看，这个“链接”的过程究竟是怎样执行的。按照发生时刻的不同，链接可以被分为编译时链接、加载时链接，以及运行时链接三种类型。其中，编译时链接又被称为“静态链接”，它是链接的一种最基本形态，今天我们便从它开始入手。

这一讲中，我会以 Linux 系统下的静态链接为例来进行介绍。虽然在其他操作系统中，这个过程的发生细节可能有所不同，但总的来看，静态链接在处理对象文件时采用的基本方法和目的都是一致的，你可以依此类推，举一反三。

静态链接 vs 动态链接

一个程序在被编译时，我们可以选择性地为其使用静态链接或动态链接。那么，二者的概念和使用场景有什么区别呢？这是经常被大家讨论的一个问题，接下来我们一起看看。

“静态链接”中的“静态”，实际上是指**在用户准备执行这个程序前，它正常运行所依赖的全部代码实现便已经“静静地躺在那里”，成为了整个可执行文件的一部分**。相对地，使用动态链接编译的程序，编译器只会为这些依赖代码在可执行程序文件中留下用于临时占位的“槽”。而只有当用户开始调用程序时，相关代码才会被真正加载到内存。而这就是“动态”一词的重要体现之一。

静态链接与动态链接两者的区别，也同样体现在了与程序相关的一些性质上，其中最关键的三点是**程序可执行文件的体积、程序的执行效率，以及程序的可移植性**。

对于静态链接来说，由于所有依赖代码都被打包在一起，因此它对应的可执行文件体积会相对较大。但也正是因为不需要外部依赖，所以可移植性较好，程序的执行效率也不会受到影响。相对地，动态链接由于不会将依赖代码打包，所以它对应的可执行文件可以被维持在一个较小的体积。但由于需要程序运行所在环境中包含这些依赖，因此可移植性相对较差。并且，由于这些依赖代码仅会在用户实际调用程序那一刻，才被加载进内存，程序的启动效率会受到一定影响。

但实际来看，在具体使用上，静态链接与动态链接两者并不是完全互斥的。通常来说，在编译程序时，我们会对那些基础且常见的公有代码库（如 libc、libm、libthread 等）采用动态链接。这些系统库已经成为支持现代操作系统正常运作的一部分，因此在大多数情况下它们都不可或缺。而对于应用程序独有的那部分实现，我们一般采用静态链接，让它们能够直接成为二进制可执行文件的一部分。

静态链接的处理过程

接下来，我们将使用如下所示的两段代码，来进一步探究静态链接的详细发生过程（这里为了方便你观察，我将它们放在了同一个代码块中）。你可以通过每段代码上方的注释信息，来区分它们所属的不同文件。

复制代码

```
// main.c
#define LEN 2
extern int sharedArr[LEN];
```

```
extern int sum(int *arr, int n);
int* array = sharedArr;
int main(void) {
  int val = sum(array, LEN);
  return val;
}

// sum.c
#define LEN 2
int sharedArr[LEN] = { 1, 2 };
int sum(int *arr, int n) {
  int i, s = 0;
  for (i = 0; i < n; i++) {
    s += arr[i];
  }
  return s;
}
```

这个程序的逻辑十分简单。在文件 sum.c 中，我们首先定义了名为 sharedArr 的全局数组。接着，又定义了名为 sum 的函数，该函数会计算并返回给定数组 arr 的前 n 个元素之和。而在文件 main.c 中，我们定义了指针变量 array，该变量将会引用 sum.c 文件内的全局数组 sharedArr。而在 main 函数中，通过调用 sum 函数，我们返回了指针 array 所指向数组的前两项之和。

接着，我们使用 GCC 来将上述代码分别编译为一个二进制可执行文件 main，以及对应的两个目标文件 main.o 与 sum.o。与上述编译流程相关的命令如下所示：

复制代码

```
gcc main.c sum.c -o main  # 生成可执行文件 main;
gcc -c main.c -o main.o  # 生成目标文件 main.o;
gcc -c sum.c -o sum.o  # 生成目标文件 sum.o;
```

在接下来的内容中，我将以这三个文件为例，来带你深入观察上面两个目标文件是如何在静态链接的一系列处理后被整合在一起，并体现在最终的可执行文件里的。首先，让我们从目标文件入手，来看看 Linux 平台上的 .o 文件究竟有何不同。

可重定位目标文件的基本结构

回顾我在 25 讲 中“ELF 文件类型”这一小节里介绍的内容，你能够知道，Linux 平台上的 .o 目标文件实际上是一种被称为“可重定位文件”的 ELF 文件类型。因此，它的内部也同样遵循着

与 ELF 二进制可执行文件类似的布局方式。只是相较于后者，它不包含有与动态视图相关的多种 Segment 与 Program 头部。因此，各种不同类型的 Section 便成为了用于描述它所有特征的基本组成结构。

这里，我将目标文件中的几个与静态链接过程密切相关的 Section 整理在了表格中，你可以先对它们的基本功能有一个大致印象。我会在接下来的内容中，带你仔细观察它们是如何在链接过程中发挥重要作用的。

Section 名称	基本功能
.symtab	符号表。其内部存放有源代码中定义和引用的所有函数与全局变量的信息
.rela.text	.text Section 的重定位信息表。其内部存放有当链接器将当前文件与其他文件通过静态链接组合时，.text 中需要被修改的代码
.rela.data	.data Section 的重定位信息表。其内部存放有当链接器将当前文件与其他文件通过静态链接组合时，.data 中需要被修改的值

每一个可重定位目标文件内都存在有一个符号表，它包含了该文件对应源码内使用到的所有全局变量和函数信息。而通过使用名为 “nm” 的命令，我们可以查看这些文件中的符号表。这里，通过对 main.o 与 sum.o 这两个目标文件使用该命令，你会得到如下图所示的输出结果：

```
➜  workspace nm main.o
0000000000000000 D array
0000000000000000 T main
                 U sharedArr
                 U sum
```

```
➜  workspace nm sum.o
0000000000000000 D sharedArr
0000000000000000 T sum
```


可以看到，不带有任何参数的 nm 命令会打印出有关符号的三部分信息（对应于图片中的三列）：

- 第一列的数字值为符号在对应 Section 中的偏移位置；
- 第二列中的大写字母表明了符号的具体类型。这里，D 表明符号为已初始化的全局数据，即位于 .data Section；T 表示符号为函数，即位于 .text Section；U 则表示符号是未定义的；
- 第三列为符号的具体名称。

链接器在整合 main.o 与 sum.o 这两个目标文件时，它的最重要工作之一就是为每一个程序使用到的符号找到与它匹配的符号定义，这个过程通常被称为“符号解析”。而如果链接器在搜索完所有输入的目标文件后，仍存在无法被解析的符号，它便会终止程序处理，并抛出类似 “undefined reference to `symbol`” 的错误信息。

但在此之前，链接器会首先对这些目标文件进行扫描，来获取它们各自的 Section 相关信息，同时计算出待输出文件中的 Section 布局信息。而为了便于后续处理，链接器还会将所有目标文件内的符号信息收集起来，统一放到一起，作为一个“全局符号表”来使用。接下来，我们就详细看看静态链接的第一个步骤，符号解析的具体流程。

步骤一：符号解析

编译器在编译源代码时，会为无法在当前编译单元内找到定义的符号生成一个特定的符号表条目，同时把“为该符号寻找定义”这个重任交给链接器。而链接器在随后进行符号解析时，便会在包含有全部符号信息的全局符号表中进行搜索。

如果链接器在这个过程中找到了符号的多个定义，它便会按照一定的规则来进行解析。编译器在编译源代码时，会为每一个全局符号指定对应的“强弱”信息，并同时将其隐含地编码在符号对应的符号表条目中。通常来说，函数和已初始化的全局变量为强符号，而未初始化的全局变量则是弱符号。而链接器对符号定义的选择会根据如下规则进行：

- 如果有一个强符号和多个弱符号同名，则选择强符号；
- 如果有多个弱符号同名，则从这些弱符号中任意选一个（通常会选择其中类型占用空间最大的那个）；
- 如果存在多个强符号同名，则抛出链接错误。

符号之所以会有强弱之分，主要是为了做到这一点：**当不确定某个符号是否被用户显式定义的情况下，链接器仍然可以选择使用对应的弱类型符号版本来编译程序**。这种能力通常被用在各类框架中，以便为某类程序编译所依赖的代码部分提供默认实现。除此之外，在模块化的代码

调试场景中（比如单元测试中的桩代码），当某个待测试模块的依赖模块还没有被实现时，链接器可以选用标记为弱类型的默认版本来编译程序。

比如，在 GCC 中，我们可以通过为函数显式添加 `__attribute__((weak))` 标记的形式，来将它标记为弱符号。但需要注意的是，这种方式会对代码的可移植性产生一定影响。

步骤二：重定位

通过上面的步骤，链接器已经可以为每一个外部符号引用，找到一个与之关联的符号定义了。比如，对于这一讲开头的实例来说，位于 main.o 文件内的未定义符号 sharedArr 与 sum 便会与 sum.o 文件内的同名符号进行匹配。

到这里，链接器便可以根据之前收集到的所有信息，开始将多个目标文件内相同类型的 Section 进行合并。同时，为这些 Section，以及所有输出文件内使用到的符号指定运行时的 VAS 地址。在这一步中，链接器会通过名为“重定位”的步骤来修改 .data 与 .text 两个 Section 中，对每个符号的引用信息，使得它们可以指向正确的运行时地址。

重定位的一个主要目的在于，将之前各个独立编译单元（目标文件）内，所有对外部符号的引用地址进行修正。比如在我们之前的例子中，main.o 文件内便存在有两个外部符号引用 sharedArr 与 sum。编译器在编译该文件时，由于尚不清楚这些符号定义的真实所在位置，因此会使用默认值（比如 0）来作为它们在机器代码中的地址占位符。

当然，除了对引用地址的修正外，可以看到，同样是在 main.o 文件内，array 变量的具体值实际上也依赖于外部符号 sharedArr 的确切所在地址。而且，已初始化的全局变量，其初始值被存放在了 .data 中，因此，这个位于该 Section 中的 array 变量的初始值，也需要被一同修改。

到这里，我们能够得知，链接器的另一个重要作用便是在组合各个目标文件的同时，对我们上面提到的这些值进行修正。而这一过程的正确执行，便依赖于我在前面介绍的两个特殊 Section，即 .rela.data 与 .rela.text。通过为 readelf 命令添加 “-r” 参数，我们可以查看 main.o 文件内，这两个 Section 的内容，如下图所示：

```
➜  workspace readelf -r ./main.o

Relocation section '.rela.text' at offset 0x220 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000000b  000800000002 R_X86_64_PC32     0000000000000000 array - 4
000000000018  000b00000004 R_X86_64_PLT32    0000000000000000 sum - 4

Relocation section '.rela.data' at offset 0x250 contains 1 entry:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000000  000900000001 R_X86_64_64       0000000000000000 sharedArr + 0
```

这两个特殊的 Section 通常也被称为“重定位表”，在它们的内部，以一行行表项的形式分别保存着链接器在重定位时需要在 .text 与 .data 中修改的具体位置和方式。这里每个表项中第一列的 Offset 属性，表明该重定向目标在对应 Section 中的偏移；Type 属性表明了重定位类型，即链接器在处理该重定位表项时，需要使用的特定方式；Sym.Value 属性为当前重定位符号的值；最后的 “Sym. Name + Addend” 属性为符号的名称，外加在计算该符号地址时需要被修正的量。

通过使用 objdump 命令，我们可以在 main.o 文件内找到上述重定向目标在各个 Section 中的位置，具体如下图所示（这里，我用红色的框将这些位置标注了出来）：

```
➜  workspace objdump -M intel -d main.o

main.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   rbp
   1:   48 89 e5                mov    rbp,rsp
   4:   48 83 ec 10             sub    rsp,0x10
   8:   48 8b 05 00 00 00 00    mov    rax,QWORD PTR [rip+0x0]        # f <main+0xf>
   f:   be 02 00 00 00          mov    esi,0x2
  14:   48 89 c7                mov    rdi,rax
  17:   e8 00 00 00 00          call   1c <main+0x1c>
  1c:   89 45 fc                mov    DWORD PTR [rbp-0x4],eax
  1f:   8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
  22:   c9                      leave
  23:   c3                      ret
➜  workspace objdump -s -j .data ./main.o

./main.o:     file format elf64-x86-64

Contents of section .data:
 0000 00000000 00000000                    ........
```


总的来看，上面我提到的重定位类型，在 X86-64 体系下可以被分为 6 种，如下表所示：

重定位类型	计算方式
R_X86_64_NONE	none
R_X86_64_64	S + A
R_X86_64_PC32	S + A - P
R_X86_64_GOT32	G + A
R_X86_64_PLT32	L + A - P
R_X86_64_COPY	none

在本例中，我们需要特别关注的是其中的 R_X86_64_PC32、R_X86_64_PLT32 与 R_X86_64_64。表格中的最后一列指明了链接器在更正地址时需要遵循的计算方式。这里，S 表示符号的实际地址；A 表示重定位条目中符号对应的 Addend（在 i386 中，由于 ELF32 的特殊要求，这个值被直接存放在被修改的内存位置上）；P 表示被修改的具体位置；L 表示在 PLT 中该符号的入口地址（PLT 通常用于实现函数的间接调用，我会在第 29 讲中介绍与它相关的内容）。

接下来，我们以 .rela.text 中的第一个重定位条目为例，来看看链接器在处理重定位时的具体流程。根据标注的类型 R_X86_64_PC32，我们能够知道更正值的具体计算方式为 “S + A - P”。其中，S 表示符号 array 在输出文件中的实际地址，这里我们可以通过 nm 命令来查看：

```
➜  workspace nm ./main | grep array
0000000000601020 D array
```

可以看到，这个地址为十六进制值 0x601020。紧接着，A 表示符号 array 在重定位条目中的 Addend，即 -4。最后，P 表示当前重定向条目在输入文件中的修改位置。同样地，使用 objdump 命令，我们可以得到这个值。

```
0000000000400536 <main>:
  400536:       55                      push   rbp
  400537:       48 89 e5                mov    rbp,rsp
  40053a:       48 83 ec 10             sub    rsp,0x10
  40053e:       48 8b 05 db 0a 20 00    mov    rax,QWORD PTR [rip+0x200adb]        # 601020 <array>
  400545:       be 02 00 00 00          mov    esi,0x2
  40054a:       48 89 c7                mov    rdi,rax
  40054d:       e8 08 00 00 00          call   40055a <sum>
  400552:       89 45 fc                mov    DWORD PTR [rbp-0x4],eax
  400555:       8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
  400558:       c9                      leave
  400559:       c3                      ret
```

上图中的红框标注了该行机器指令的起始位置，在此基础之上，再向“右侧”移动 3 个字节，我们便可得到重定向的修改位置，即 0x400541。最后，按照公式 “S + A - P” 进行计算，即 数学公式: $0x601020 - 0x4 - 0x400541$，链接器便可得到最终的修改值 0x200adb。通过与上图中使用绿色框标注的内容进行比对，我们可以验证这个结果。这里需要注意的是，图中的“左侧”为低地址位（LSB），因此其字节顺序与我们的结果相反。

通过类似的方式，链接器可以完成对所有重定位条目的处理过程。而在此之后，输出的可执行文件中，所有符号也都有了正确的初始值和引用地址。随后，程序便可以被操作系统加载进内存，正常运行。

总结

今天我主要为你介绍了 Linux 中的静态链接，以此为例，带你深入了解了编译器在链接程序时发生了什么。

静态链接与动态链接相对应，主要是指“**在链接过程中，来自于不同目标文件的代码会被整合为二进制可执行文件的一部分**”这个过程。总的来看，静态链接被分为两个步骤：符号解析与重定位。

其中，符号解析是指为应用程序使用的所有符号正确匹配对应符号定义的过程。当有重名的多个符号定义存在时，链接器会按照一定规则来选择适用的版本。

而在重定位过程中，链接器会将输入的多个目标文件的同类型 Section 进行合并，并为它们和所有程序使用到的符号分配运行时的 VAS 地址。紧接着，借助重定位表中的信息，链接器可以对上一步中得到的外部符号，进行地址及值上的修正。

思考题

查阅相关资料，尝试了解下：在 Linux 系统中，以后缀 “.a” 结尾的静态库文件与 .o 目标文件，二者之间有什么关系呢？欢迎在评论区告诉我你的发现。

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

上一篇　26 | 进程是如何使用操作系统内存的？

下一篇　28 | 程序可以在运行时进行链接吗？

精选留言 (2)

写留言

liu_liu
2022-02-25

老师有个疑问，S + A - P 中的 A 不是代表 Addend 吗？为什么是待修改位置上的值呢？

作者回复: 观察的很细致！这里确实是我把 ELF32 与 ELF64 下的重定位方式搞混了，因为比如对于 R_386_32 和 R_X86_64_64 来说，两个的计算方式都是 S + A，但前者中的 A 实际指的是被修改内存所在位置上的值，本质上就是符号的 addend。而对于后者，也就是 ELF64 来说，addend 被单独拎出来与重定向条目放到一起了。这里我来纠正一下，稍后我也会修改下文章内容！感谢！

这里我把 ELF 标准中的相关段落也放出来，大家在后面的学习中也注意类似的问题。文档（http://www.skyfree.org/linux/references/ELF_Format.pdf）第 29 页：“The SYSTEM V architecture uses only Elf32_Rel relocation entries, the field to be relocated holds the addend.”

3

LDxy
2022-02-25

一个.a静态库文件可以包含多个.o目标文件