下载APP

# 06 | 静态链接：变量与内存地址是如何映射的？

2021-11-05 海纳

《编程高手必学的内存知识》 课程介绍 >

**讲述：海纳**

时长 24:23 大小 22.34M

你好，我是海纳。

在第 3 节课里，我们看到进程的内存空间包含代码段、数据段、bss 段、堆和栈等区域。在第 4 节和第 5 节课里，我们对栈的相关知识进行了深入学习。今天我们来看看内存的另一个重要部分：代码段和数据段的组织方式。

我们知道，编程的代码无非是由函数和各种变量，以及对这些变量的读、写所组成，而不管是变量还是函数，它们最终都要存储在内存里。为每个变量和函数正确地分配内存空

领资料

间，记录它们的地址，并把这个地址复写回调用或引用它们的地方，这是一个十分重要且困难的任务。

在我们使用 gcc 时，往往执行一个命令后，就能得到可执行程序，所以你可能会误以为是编译器负责为变量分配内存地址，但是实际上，这个工作是由链接器来完成的。每个变量和函数都有自己的名称，通常我们把这些名称叫做符号。简单来讲，链接器的作用就是为符号转换成地址，一般来说可以分为三种情况：

1. 生成二进制可执行文件的过程中。这种情况称为静态链接；
2. 在二进制文件被加载进内存时。这种情况是在二进制文件保留符号，在加载时再把符号解析成真实的内存地址，这种被称为动态链接；
3. 在运行期间解析符号。这种情况会把符号的解析延迟到最后不得不做时才去做符号的解析，这也是动态链接的一种。

相信你在工作中，尤其是在编译各种开源项目时，肯定遇到过“找不到符号”，或者“undefined reference to X”这样的报错信息，其实这些错误都和编译链接的过程有关系。所以，接下来的 3 节课，我们就重点来分析一下链接器是怎么完成内存地址的映射工作的，了解了这个原理后，再遇到类似的问题，你就知道如何着手去分析了。

今天这节课我们先来探讨静态链接的过程。

## 关于链接的小例子

我们先用一个具体的例子展示一遍编译和链接的全部过程，然后再分析每一步的原理。这个例子包含两个文件，第一个文件是 example.c：

复制代码

```
// example.c
extern int extern_var;
int global_var = 1;
static int static_var = 2;

extern int extern_func();
int global_func() {
    return 10;
}

static int static_func() {
```

```
12      return 20;
13  }
14
15  int main() {
16      int var0 = extern_var;
17      int var1 = global_var;
18      int var2 = static_var;
19      int var3 = extern_func();
20      int var4 = global_func();
21      int var5 = static_func();
22      return var0 + var1 + var2 + var3 + var4 + var5;
23  }
```

第二个文件是 external.c：

复制代码

```
1  // external.c
2  int extern_var = 3;
3  int extern_func() {
4      return 30;
5  }
```

我们先使用 gcc 将两个 c 文件分别编译成.o 目标文件，这个过程称为编译，命令如下：

复制代码

```
1  # gcc example.c -c -o example.o -fno-PIC -g
2  # gcc external.c -c -o external.o -fno-PIC -g
```

这里我给你解释一下命令中的几个选项：

-c 意思是告诉 gcc 不要进行链接，只要编译到.o 就可以了；

-o 指定了输出文件名；

-fno-PIC 是告诉编译器不要生成 PIC 的代码。因为我使用的是 gcc4.8 版本，在编译的过程中默认的模式是 PIC 模式，由于我们今天讨论的内容主要是静态链接的部分，所以需要打开 -fno-PIC 选项。这个选项对动态链接的意义比较大，在下节课讲动态链接时，我会对这个选项给你做详细解释。

-g 选项是打开调试信息，让我们在分析过程中能够对源码有更完整的对应关系。

然后，我们将两个.o 文件链接生成可执行文件，由目标文件生成可执行文件的过程就是链接。命令如下：

```
1 # gcc external.o example.o -o a.out -no-pie
```

复制代码

在这个命令中，-no-pie 表示关闭 pie 的模式。gcc 会默认打开 pie 模式，也就意味着系统 loader 对加载可执行文件时的起始地址，会随机加载。关闭 pie 之后，在 Linux 64 位的系统下，默认的加载起始地址是 0x400000。关于这个选项，我们将在下节课详细讲解。

这样，我们就得到了可执行二进制文件 a.out，以上内容就是编译和链接的全过程了。接下来我们详细看一看链接器在这个过程中发挥的作用。

## 链接器的作用

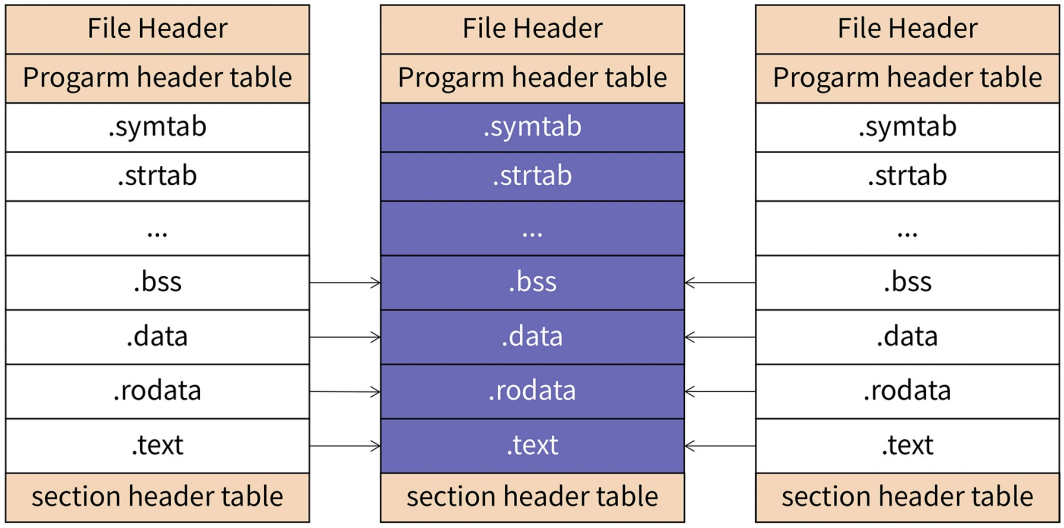
我们继续结合上面的例子来说明，这个例子其实涵盖了程序员在开发过程中，最常用的几种变量类型以及函数类型，分别是：

1. 全局变量：global_var。
2. 静态变量：static_var。
3. 外部变量：extern_var，在 example.c 中使用 extern 关键字进行声明，定义在 external.c 里。
4. 局部变量：var0 … var5。
5. 全局函数：global_func。
6. 静态函数：static_func。
7. 外部函数：extern_func，在 example.c 中使用 extern 关键字进行声明，定义在 external.c 里。

程序员在开发代码的过程中，也是直接跟这些符号打交道的。如果想获取某个变量的值，就直接从变量符号里读取内容；如果想调用某个函数，也是直接写一个函数符号的调用语句。

但是，我们知道，CPU 在执行程序代码的时候，并不理解符号的概念，它所能理解的只有内存地址的概念。不管是读数据，调用函数还是读指令，对于 CPU 而言都是一个个的内存地址。因此，**这里就需要一个连接 CPU 与程序员之间的桥梁，把程序中的符号转换成 CPU 执行时的内存地址。这个桥梁就是链接器，它负责将符号转换为地址。**

**链接器的第一个作用就是把多个中间文件合并成一个可执行文件。**我们在第 3 节课分析过，每个中间文件都有自己的代码段和数据段等多个 section，在合并成一个可执行程序时，多个中间文件的代码段会被合并到可执行文件的代码段，它们数据段也会被合并为可执行文件的数据段。具体的过程可以参考下面这个图：




但是链接器在合并多个目标文件的时候并不是简单地将各个 section 合并就可以了，它还需要考虑每个目标中的符号的地址。这就引出了**链接器的第二个任务：重定位。**所谓重定位，就是当被调用者的地址变化了，要让调用者知道新的地址是什么。

## 两步链接

根据上边的分析，链接器的工作流程也主要分为两步：

**第一步是，链接器需要对编译器生成的多个目标（.o）文件进行合并，一般采取的策略是相似段的合并，最终生成共享文件 (.so) 或者可执行文件。**这个阶段中，链接器对输入的各个目标文件进行扫描，获取各个段的大小，并且同时会收集所有的符号定义以及引用信息，

构建一个全局的符号表。当链接器构造好了最终的文件布局以及虚拟内存布局后，我们根据符号表，也就能确定了每个符号的虚拟地址了。

**第二步是，链接器会对整个文件再进行第二遍扫描，这一阶段，会利用第一遍扫描得到的符号表信息，依次对文件中每个符号引用的地方进行地址替换。也就是对符号的解析以及重定位过程。**

这就是链接器常用的两步链接 (Two-pass linking) 的步骤。简单来讲就是进行两遍扫描：第一遍扫描完成文件合并、虚拟内存布局的分配以及符号信息收集；第二遍扫描则是完成了符号的重定位过程。

重定位是符号解析的重要步骤，是我们理解静态链接和动态链接的基础原理。在 JVM 或者 V8 虚拟机中，对符号的解析的原理与链接器的重定位过程是十分相似的，可见重定位应用得非常广泛，所以接下来我们要重点了解一下重定位的原理。

## 深入分析重定位过程

工欲善其事，必先利其器，在 GNU/linux 下，GNU 的 binutils 提供了一系列编程语言的工具程序，用来查看不同格式下的目标文件。今天我要给你重点介绍两个工具：readelf 和 objdump，这两个工具可以用来解析和读取上一节编译阶段生成的目标文件信息。

一般情况下，我在对二进制文件进行反汇编时会使用 objdump 工具，因为 readelf 工具没有提供反汇编的能力，它更多是用来解析二进制文件信息。

在前面的例子中，我们已经编译出两个.o 目标文件，以及最终链接后的 a.out 可执行文件，接下来我们通过对比.o 文件以及 a.out 文件中符号的差异来分析重定位的过程。

首先，我们通过 objdump 看一下此时目标文件里的反汇编是什么样子的。

复制代码

```
# objdump -S example.o
0000000000000000 <global_func>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   b8 0a 00 00 00          mov    $0xa,%eax
   9:   5d                      pop    %rbp
   a:   c3                      retq
```

```
 9
10 000000000000000b <static_func>:
11    b:   55                      push   %rbp
12    c:   48 89 e5                mov    %rsp,%rbp
13    f:   b8 14 00 00 00          mov    $0x14,%eax
14   14:   5d                      pop    %rbp
15   15:   c3                      retq
16
17 0000000000000016 <main>:
18 int main() {
19   16:   55                      push   %rbp
20   17:   48 89 e5                mov    %rsp,%rbp
21   1a:   48 83 ec 20             sub    $0x20,%rsp
22     int var0 = extern_var;
23   1e:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 24 <main+0xe>
24   24:   89 45 e8                mov    %eax,-0x18(%rbp)
25     int var1 = global_var;
26   27:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 2d <main+0x17>
27   2d:   89 45 ec                mov    %eax,-0x14(%rbp)
28     int var2 = static_var;
29   30:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 36 <main+0x20>
30   36:   89 45 f0                mov    %eax,-0x10(%rbp)
31     int var3 = extern_func();
32   39:   b8 00 00 00 00          mov    $0x0,%eax
33   3e:   e8 00 00 00 00          callq  43 <main+0x2d>
34   43:   89 45 f4                mov    %eax,-0xc(%rbp)
35     int var4 = global_func();
36   46:   b8 00 00 00 00          mov    $0x0,%eax
37   4b:   e8 00 00 00 00          callq  50 <main+0x3a>
38   50:   89 45 f8                mov    %eax,-0x8(%rbp)
39     int var5 = static_func();
40   53:   b8 00 00 00 00          mov    $0x0,%eax
41   58:   e8 ae ff ff ff          callq  b <static_func>
42   5d:   89 45 fc                mov    %eax,-0x4(%rbp)
43   …
```

由于空间的限制，我只保留了 main 函数中源码与汇编码对应的部分内容。你需要注意的是，上边源码与汇编的对应，需要在编译.o 文件时打开 -g 选项，否则就只有汇编代码。

下面我们来分类讨论各种符号的处理方式。

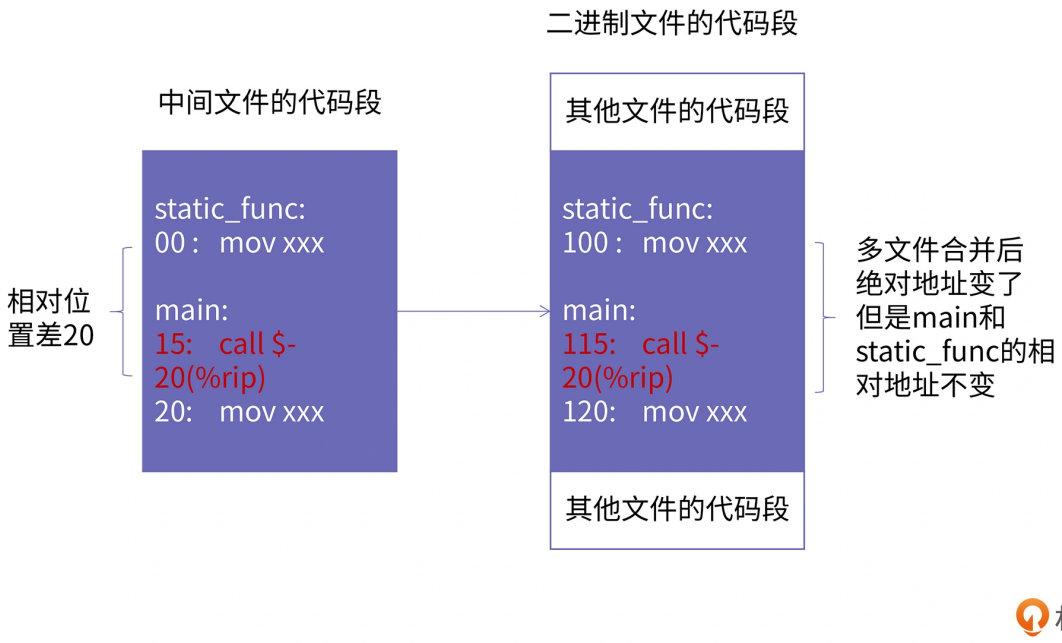
## 各种符号的处理方式

**首先，我来看看局部变量的处理过程。**从反汇编的结果里，我们可以看到，局部变量在程序中的地址，都是基于 %rbp 的偏移这种形式，rbp 寄存器存放的是当前函数栈帧的基地

址。这些局部变量的内存分配与释放，都是在运行时通过 %rbp 的改变来进行的，因此，局部变量的内存地址不需要链接器来操心。

**然后，再来看看比较简单的 static_func，它是唯一不需要重定位的类型**。对 static_func 的调用，所生成的指令的二进制是 e8 ae ff ff ff。其中，e8 是 callq 指令的编码，后边 4 个字节就对应被调函数的地址。注意，这里生成的 ae ff ff ff，如果采用小端的字节序数值来表示，应该是 0xffffffae，也就是对应十进制的 -82。

此时，当 CPU 执行到 callq 这条指令时，rip 寄存器的值指向的是下一条指令的内存地址，也就是 5d 这条指令的内存地址，通过计算 0x5d – 82 可以得到 0xb。从反汇编中可以得到，0xb 刚好是 static_func 的地址。static_func 的链接原理，你可以参考下面这幅图：




从上图中可以看出，同一个编译单元内部，static_func 与 main 函数的相对位置是固定不变的，即便链接的过程中会对不同.o 文件中的代码段进行合并，但是同一个.o 文件内部不同函数之间的位置也会保持不变，因此，我们在编译的时候，就能确定对静态函数调用的偏移。也就是说，静态函数的调用地址在编译阶段就可以确定下来。

我们可以在最终生成的可执行文件的 main 函数中，查看对应位置代码的反汇编。可以验证的是，这里确实没有进行重定位的修正：

复制代码

```
0000000004004ad <main>:
  ...
  4004ef:       e8 ae ff ff ff          callq  4004a2 <static_func>
  ...
```

**接下来，我们再看第三类符号，也就是外部变量、全局变量以及静态变量的处理过程。**你可以从反汇编结果中看到，前三条语句对 extern_var、global_var 和 static_var 的访问，都生成了一条 mov 0x0(%rip)，%eax 的指令。这是因为在这个时候，编译器还无法确定这三个变量的地址，因此，这里先通过 0 来进行占位，以后链接器会将真正的地址回填在这里。

**最后，我们来看对于 extern_func 和 global_func 的调用**，call 指令同样是通过 0 来进行占位，这和全局变量的处理方式一样。

### 处理占位符

我们前面说到，在无法确定变量的真实地址时，先通过 0 来进行占位。所以，我们这里继续观察链接器对 extern_var，static_var，global_var，global_func 以及 extern_func 的重定位过程，看看它们的占位符是如何处理的。

这里我们需要通过 readelf 工具来查看一下目标文件里有哪些信息：

复制代码

```
# readelf -S example.o
There are 12 section headers, starting at offset 0x478:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000000000  00000040
       000000000000007e  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  00000358
       0000000000000078  0000000000000018   I       9     1     8
  [ 3] .data             PROGBITS         0000000000000000  000000c0
       0000000000000004  0000000000000000  WA       0     0     4
  [ 4] .bss              NOBITS           0000000000000000  000000c4
       0000000000000004  0000000000000000  WA       0     0     4
  [ 5] .comment          PROGBITS         0000000000000000  000000c4
```

```
       000000000000002a  0000000000000001  MS       0     0     1
  [ 6] .note.GNU-stack   PROGBITS         0000000000000000  000000ee
       0000000000000000  0000000000000000           0     0     1
  [ 7] .eh_frame         PROGBITS         0000000000000000  000000f0
       0000000000000078  0000000000000000   A       0     0     8
  [ 8] .rela.eh_frame    RELA             0000000000000000  000003d0
       0000000000000048  0000000000000018   I       9     7     8
  [ 9] .symtab           SYMTAB           0000000000000000  00000168
       0000000000000180  0000000000000018          10    10     8
  [10] .strtab           STRTAB           0000000000000000  000002e8
       000000000000006b  0000000000000000           0     0     1
  [11] .shstrtab         STRTAB           0000000000000000  00000418
       0000000000000059  0000000000000000           0     0     1
```

其中的 readelf -S 选项是打印出二进制文件中所有 section-header 的信息。我们可以看到 example.o 里总共包含了 12 个 section，其中，.text 段、.data 段和.bss 段我在前面的课程里都提到过，这里我们重点看下.rela.text 段。

从 section-header 的信息里可以看到，.rela.text 段的类型是 RELA 类型，也就是重定位表。我们在前面讲到，链接器在处理目标文件的时候，需要对目标文件里代码段和数据段引用到的符号进行重定位，而这些重定位的信息都记录在对应的重定位表里。

一般来说，重定位表的名字都是以.rela 开头，比如.rela.text 就是对.text 段的重定位表，.rela.data 是对.data 段的重定位表。因为我们的例子中并没有涉及.data 段的重定位，所以，在上面打印的信息中没有出现.rela.data 段。

好了，接下来我们具体看一下.rela.text 重定位表里的内容。你可以通过 readelf -r 选项来打印二进制文件中的重定位表信息，输出如下：

复制代码

```
Relocation section '.rela.text' at offset 0x330 contains 5 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Adde
000000000020  000d00000002 R_X86_64_PC32     0000000000000000 extern_var - 4
000000000029  000a00000002 R_X86_64_PC32     0000000000000000 global_var - 4
000000000032  000300000002 R_X86_64_PC32     0000000000000000 .data + 0
00000000003f  000e00000002 R_X86_64_PC32     0000000000000000 extern_func - 4
00000000004c  000b00000002 R_X86_64_PC32     0000000000000000 global_func - 4
```

.rela.text 的重定位表里存放了 text 段中需要进行重定位的每一处信息。所以，每个重定位项都会包含需要重定位的偏移、重定位类型和重定位符号。重定位表的数据结构是这样

的：

```
typedef struct {
  Elf64_Addr›    r_offset; /* 重定位表项的偏移地址 */
  Elf64_Xword›   r_info;   /* 重定位的类型以及重定位符号的索引 */
  Elf64_Sxword› r_addend; /* 重定位过程中需要的辅助信息 */
} Elf64_Rela;
```

其中，r_info 的高 32bit 存放的是重定位符号在符号表的索引，r_info 的低 32bit 存放的是重定位的类型的索引。符号表就是.symtab 段，可以把它看成是一个字典，这个字典以整数为 key，以符号名为 value。

在我们的例子中，根据上文的汇编代码来看，.rela.text 段中的重定位表总共有 5 项，分别对应到.text 的 0x20, 0x29, 0x32, 0x3f, 0x4c 偏移处。我们以 0x20 为例，它对应的汇编指令是 0x1e 位置的 8b 05 00 00 00 00。0x20 指向的是这条指令的操作数，在没有重定位之前，它是一个四字节填充的 0，对应的是对变量 extern_var 的访问。

同样地，其余的几处偏移位置分别是访问 global_var、static_var、global_func 和 extern_func 这四个符号（函数和变量都可统一看成是符号）的地方。

接下来，我们着重分析这四个符号的重定位过程。我们可以看到重定位表中的这四项，**它们的类型都是 R_X86_64_PC32。这种类型的重定位计算方式为：S + A – P。**

> 这里的 S 表示完成链接后该符号的实际地址。在链接器将多个中间文件的段合并以后，每个符号就按先后顺序依次都会分配到一个地址，这就是它的最终地址 S。
>
> A 表示 Addend 的值，它代表了占位符的长度。它的具体用法我们下文还会详细分析。
>
> P 表示要进行重定位位置的地址或偏移，可以通过 r_offset 的值获取到，这是引用符号的地方，也就是我们要回填地址的地方，简单说，它就是我们上文提到的用 0 填充的占位符的地址。

这里 S 与 P 所表示的地址都是文件合并之后最终的虚拟地址，由于我们无法获取链接器中间过程的文件，所以，我们需要通过查看链接完成后的可执行文件，来寻找这两个地址。

我们以 extern_var 的变量为例，具体跟踪一遍重定位的过程。

复制代码

```
1 00000000004004ad <main>:
2   4004ad:       55                      push   %rbp
3   4004ae:       48 89 e5                mov    %rsp,%rbp
4   4004b1:       48 83 ec 20             sub    $0x20,%rsp
5   4004b5:       8b 05 75 0b 20 00       mov    0x200b75(%rip),%eax        # 60
6   4004bb:       89 45 e8                mov    %eax,-0x18(%rbp)
```

上边输出部分是对生成可执行文件的反汇编。根据 S、A、P 的定义，我们知道对于 extern_var 来讲：

S 是其最终符号的真实地址，如上汇编里边的注释所示  也就是上面注释的 0x601030 这个地址；

A 是 Addend 的值，可以从重定位表里查到是 -4，对于 A 的具体含义我还会进一步解释；

P 是重定位 offset 的地址，这里是 0x4004b7。

根据公式，我们算出重定位处需要填写的值应该是 0x601030 + (-4) – 0x4004b7 = 0x200b75，也就是最终可执行文件中这条 mov 指令里的值。

到目前为止，我们从链接器的视角推出了最终重定位位置的值，你可能会比较迷糊：系统为什么搞这么一套复杂的公式来计算出这么一个值呢？这个值的真正含义是什么？

针对这个问题，我们再从 CPU 的角度来看下这里的取值关系。从上面 main 函数的反编译的结果可以看到，我们最终对 extern_var 的访问生成的汇编是：

复制代码

```
1 mov    0x200b75(%rip), %eax
```

这是一条 PC 相对偏移的寻址方式。当 CPU 执行到这条指令的时候，%rip 的值存放的是下一条指令的地址，也就是 0x4004bb。这时候可以算出 0x4004bb + 0x200b75 = 0x601030，刚好是 extern_var 的实际地址。

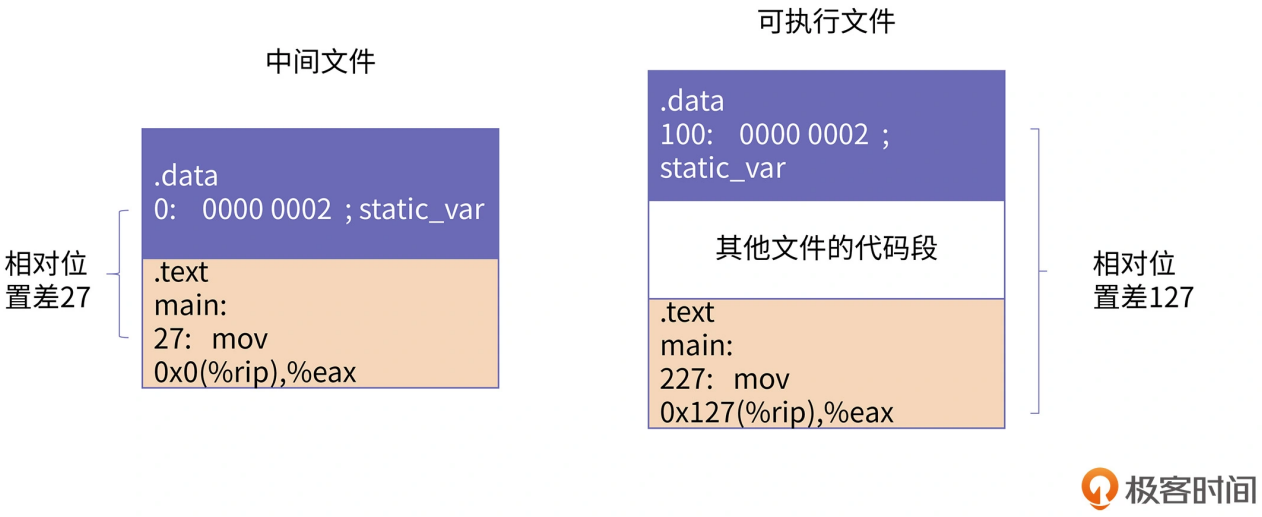
经过正面分析这个重定位的值的作用后，这里我们再来理解一下 S+A-P 这个公式的作用。链接器有了整体的虚拟内存布局后，知道的信息是：需要重定位符号的地址 S 的值是 (0x601030)，以及需要重定位的位置地址 P 的值是 (0x4004b7)。

这时候，链接器需要在指令中占位符的位置填一个值，让程序运行的时候能够找到 S。但程序运行到这条指令的时候，能够拿到的地址就只有 PC 的值，也就是下一条指令的地址 (0x4004bb)。你会发现，重定位地址的值跟下一条 pc 的值，相差的就是这个 Addend(-4)，这个 Addend 实际上就是用来调整 P 的值和执行时 PC 的值之间的差异的，所以它刚好就是占位符的宽度。

## 静态变量

除了上述所讲的四个符号之外，还有一个比较特殊的是 static_var 变量。我们可以从 Sym. Name 里找到其余变量的符号，但 static_var 的符号没有出现，只有一个.data 的符号。

这是因为 static_var 变量本身是一个静态变量，只在本编译单元内可见，不会对外进行暴露，所以它是根据本编译单元的.data 段的地址来进行重定位。也就是说，**static_var 的最终地址就是本编译单元的.data 段的最终地址**。所以，它的重定义方法与 extern_var 等符号的重定位方法是一样的，区别仅仅在于它的符号被隐藏了。如下图所示：




你可能会有疑问，既然静态函数可以在编译的时候确定相对偏移，那为什么静态变量做不到这一点呢？

这是因为静态变量的位置是在 data 段，而对静态函数的访问是在 text 段。对应 text 段内部的偏移可以保证在链接的过程中不发生改变，但由于 text 段和 data 段分属不同的段，在链接的时候大概率会进行重新排布，所以它和引用它的地方之间的相对位置就发生变化了。所以静态变量的地址就需要链接器来进行重定位了。

好，到这里我们就对整个重定位的过程有了清晰的了解。

## 总结

我们今天讲解了在静态链接过程中，变量与内存地址是如何对应起来的。其中，链接器的重定位操作是这个过程中的核心步骤。

我们说，从源文件生成二进制可执行文件，这一过程主要包含了编译和链接两个步骤。其中，编译的作用是生成性能优越的机器码。对于编译单元内部的静态函数，可以在编译时通过相对地址的办法，生成 call 指令，因为无论将来调用者和被调用者被安置到什么地方，它们之间的相对距离不会发生变化。

而其他类型的变量和函数在编译时，编译器并不知道它们的最终地址，所以只能使用占位符（比如 0）来临时代替目标地址。

而链接器的任务是为所有变量和函数分配地址，并把被分配到的地址回写到调用者处。链接的过程主要分为两步，第一步是多文件合并，同时为符号分配地址，第二步则是将符号的地址回写到引用它的地方。其中，地址回写有一个专门的名字叫做重定位。重定位的过程依赖目标文件中的重定位表。

到这里，我们已经对例子中的几种不同类型符号的静态链接有了一个清晰的认识。下面的课程我会继续讲解 loader 和动态链接的过程。

## 思考题

经过了这节课的学习，我们深刻地理解了全局变量，静态变量的用法。请你思考一下，全局变量和静态变量的初值是在哪个阶段决定的？更具体地说，是编译、链接、加载和运行这四个阶段中的哪一个阶段决定的？或者你也可以进行这样的思考，我们在目标文件、可执行文件和运行时内存里，能不能观察到全局变量和静态变量的初始值？

# 吊打面试官

- 谈谈编译器和链接器的作用？

这个问题之所以会被经常问到，是因为很多人对编译器有一个误解，以为它可以直接将源代码翻译成二进制文件。这个问题其实是在考察你对工具链的熟悉程度。所以我们在回答的时候，重点要落在编译和链接的区别上。

编译器的作用主要是把源代码文件翻译成中间结构（例如LLVM IR），然后对它进行优化，以优化程序的性能。编译器的输出是汇编文件，汇编文件中仍然保留了符号。然后，汇编编译器再将汇编文件翻译成中间文件。最后，链接器将中间文件合并起来，组成二进制可执行程序。这种合并不是简单地拼接，而是要为符号分配地址，然后再把地址回填到引用符号的地方，这个过程就是重定位。而重定位才是链接器最重要的任务。

高频面试真题

极客时间

好啦，这节课到这就结束啦。欢迎你把这节课分享给更多对计算机内存感兴趣的朋友。我是海纳，我们下节课再见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 05 | 栈的魔法：从栈切换的角度理解进程和协程


## 精选留言 (3)

写留言

**keepgoing**
2021-11-06

自己的话总结：

1. 静态函数不需要重定位因为和执行单元代码都在.text段，相对位置在编译的时候就能确定了，因为链接器合并中间文件时相对位置不会变。
2. 静态变量需要重定位，因为和编译单元代码段.text分属不同的section，在.data，链接...

展开

作者回复: 总结得真好！来，来，来，笔给你，你来写。哈哈哈。

1

**慢动作**
2021-11-06

为什么要S + A - P，直接用S有什么不好的？

展开

作者回复: S是绝对地址，而S+A-P算出来的是相对地址。使用相对地址的好处是只要引用者和被引用者的相对位置不变，那么它们就可以被安排到任意的位置上。这就可以支持加载地址随机化等安全增强技术啦。


**kylin**

2021-11-06

大端和小端都是从内存的低地址向高地址顺序写入大端 先写字面值高位小端 先写字面值低位例如：0x ff ff ff ae大端 从低地址到高地址写入顺序为：ff ff ff ae小端 从低地址到高地址写入顺序为：ae ff ff ff