- 重定位节中的符号引用。在这一步中，链接器修改代码节和数据节中对每个符号的引用，使得它们指向正确的运行时地址。要执行这一步，链接器依赖于可重定位目标模块中称为重定位条目(relocation entry)的数据结构，我们接下来将会描述这种数据结构。

7.7.1 重定位条目

当汇编器生成一个目标模块时，它并不知道数据和代码最终将放在内存中的什么位置。它也不知道这个模块引用的任何外部定义的函数或者全局变量的位置。所以，无论何时汇编器遇到对最终位置未知的目标引用，它就会生成一个重定位条目，告诉链接器在将目标文件合并成可执行文件时如何修改这个引用。代码的重定位条目放在.rel.text 中。已初始化数据的重定位条目放在.rel.data 中。

图 7-9 展示了 ELF 重定位条目的格式。offset 是需要被修改的引用的节偏移。symbol 标识被修改引用应该指向的符号。type 告知链接器如何修改新的引用。addend 是一个有符号常数，一些类型的重定位要使用它对被修改引用的值做偏移调整。

code/link/elfstructs.c

```
1  typedef struct {
2      long offset;    /* Offset of the reference to relocate */
3      long type:32,   /* Relocation type */
4           symbol:32; /* Symbol table index */
5      long addend;    /* Constant part of relocation expression */
6  } Elf64_Rela;
```

code/link/elfstructs.c

图 7-9 ELF 重定位条目。每个条目表示一个必须被重定位的引用，并指明如何计算被修改的引用

ELF 定义了 32 种不同的重定位类型，有些相当隐秘。我们只关心其中两种最基本的重定位类型：

- R_X86_64_PC32。重定位一个使用 32 位 PC 相对地址的引用。回想一下 3.6.3 节，一个 PC 相对地址就是距程序计数器(PC)的当前运行时值的偏移量。当 CPU 执行一条使用 PC 相对寻址的指令时，它就将在指令中编码的 32 位值加上 PC 的当前运行时值，得到有效地址(如 call 指令的目标)，PC 值通常是下一条指令在内存中的地址。
- R_X86_64_32。重定位一个使用 32 位绝对地址的引用。通过绝对寻址，CPU 直接使用在指令中编码的 32 位值作为有效地址，不需要进一步修改。

这两种重定位类型支持 x86-64 小型代码模型(small code model)，该模型假设可执行目标文件中的代码和数据的总体大小小于 2GB，因此在运行时可以用 32 位 PC 相对地址来访问。GCC 默认使用小型代码模型。大于 2GB 的程序可以用-mcmodel=medium(中型代码模型)和-mcmodel=large(大型代码模型)标志来编译，不过在此我们不讨论这些模型。

7.7.2 重定位符号引用

图 7-10 展示了链接器的重定位算法的伪代码。第 1 行和第 2 行在每个节 s 以及与每个节相关联的重定位条目 r 上迭代执行。为了使描述具体化，假设每个节 s 是一个字节数组，每个重定位条目 r 是一个类型为 Elf64_Rela 的结构，如图 7-9 中的定义。另外，还