假设当算法运行时，链接器已经为每个节(用 ADDR(s)表示)和每个符号都选择了运行时地址(用 ADDR(r.symbol)表示)。第 3 行计算的是需要被重定位的 4 字节引用的数组 s 中的地址。如果这个引用使用的是 PC 相对寻址，那么它就用第 5～9 行来重定位。如果该引用使用的是绝对寻址，它就通过第 11～13 行来重定位。

```
1   foreach section s {
2       foreach relocation entry r {
3           refptr = s + r.offset;  /* ptr to reference to be relocated */
4
5           /* Relocate a PC-relative reference */
6           if (r.type == R_X86_64_PC32) {
7               refaddr = ADDR(s) + r.offset; /* ref's run-time address */
8               *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
9           }
10
11          /* Relocate an absolute reference */
12          if (r.type == R_X86_64_32)
13              *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
14      }
15  }
```

图 7-10　重定位算法

让我们来看看链接器如何用这个算法来重定位图 7-1 示例程序中的引用。图 7-11 给出了(用 objdump-dx main.o 产生的)GNU OBJDUMP 工具产生的 main.o 的反汇编代码。

code/link/main-relo.d

```
1   0000000000000000 <main>:
2      0:   48 83 ec 08              sub    $0x8,%rsp
3      4:   be 02 00 00 00           mov    $0x2,%esi
4      9:   bf 00 00 00 00           mov    $0x0,%edi          %edi = &array
5                              a: R_X86_64_32 array            Relocation entry

6      e:   e8 00 00 00 00           callq  13 <main+0x13>     sum()
7                              f: R_X86_64_PC32 sum-0x4        Relocation entry
8     13:   48 83 c4 08              add    $0x8,%rsp
9     17:   c3                       retq
```

code/link/main-relo.d

图 7-11　main.o 的代码和重定位条目。原始 C 代码在图 7-1 中

main 函数引用了两个全局符号：array 和 sum。为每个引用，汇编器产生一个重定位条目，显示在引用的后面一行上。[⊖]这些重定位条目告诉链接器对 sum 的引用要使用 32 位 PC 相对地址进行重定位，而对 array 的引用要使用 32 位绝对地址进行重定位。接下来两节会详细介绍链接器是如何重定位这些引用的。

1. 重定位 PC 相对引用

图 7-11 的第 6 行中，函数 main 调用 sum 函数，sum 函数是在模块 sum.o 中定义的。

⊖ 回想一下，重定位条目和指令实际上存放在目标文件的不同节中。为了方便，OBJDUMP 工具把它们显示在一起。