1. PIC 数据引用

编译器通过运用以下这个有趣的事实来生成对全局变量的 PIC 引用：无论我们在内存中的何处加载一个目标模块(包括共享目标模块)，数据段与代码段的距离总是保持不变。因此，代码段中任何指令和数据段中任何变量之间的距离都是一个运行时常量，与代码段和数据段的绝对内存位置是无关的。

想要生成对全局变量 PIC 引用的编译器利用了这个事实，它在数据段开始的地方创建了一个表，叫做*全局偏移量表*(Global Offset Table，GOT)。在 GOT 中，每个被这个目标模块引用的全局数据目标(过程或全局变量)都有一个 8 字节条目。编译器还为 GOT 中每个条目生成一个重定位记录。在加载时，动态链接器会重定位 GOT 中的每个条目，使得它包含目标的正确的绝对地址。每个引用全局目标的目标模块都有自己的 GOT。

图 7-18 展示了示例 libvector.so 共享模块的 GOT。addvec 例程通过 GOT[3]间接地加载全局变量 addcnt 的地址，然后把 addcnt 在内存中加 1。这里的关键思想是对 GOT[3]的 PC 相对引用中的偏移量是一个运行时常量。

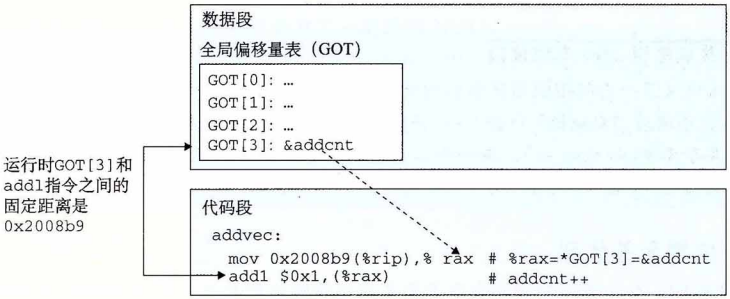

图 7-18　用 GOT 引用全局变量。libvector.so 中的 addvec 例程通过 libvector.so 的 GOT 间接引用了 addcnt

因为 addcnt 是由 libvector.so 模块定义的，编译器可以利用代码段和数据段之间不变的距离，产生对 addcnt 的直接 PC 相对引用，并增加一个重定位，让链接器在构造这个共享模块时解析它。不过，如果 addcnt 是由另一个共享模块定义的，那么就需要通过 GOT 进行间接访问。在这里，编译器选择采用最通用的解决方案，为所有的引用使用 GOT。

2. PIC 函数调用

假设程序调用一个由共享库定义的函数。编译器没有办法预测这个函数的运行时地址，因为定义它的共享模块在运行时可以加载到任意位置。正常的方法是为该引用生成一条重定位记录，然后动态链接器在程序加载的时候再解析它。不过，这种方法并不是 PIC，因为它需要链接器修改调用模块的代码段，GNU 编译系统使用了一种很有趣的技术来解决这个问题，称为*延迟绑定*(lazy binding)，将过程地址的绑定推迟到第一次调用该过程时。

使用延迟绑定的动机是对于一个像 libc.so 这样的共享库输出的成百上千个函数中，一个典型的应用程序只会使用其中很少的一部分。把函数地址的解析推迟到它实际被调用的地方，能避免动态链接器在加载时进行成百上千个其实并不需要的重定位。第一次调用过程的运行时开销很大，但是其后的每次调用都只会花费一条指令和一个间接的内存引用。

延迟绑定是通过两个数据结构之间简洁但又有些复杂的交互来实现的，这两个数据结