彻底理解链接器：五，重定位

Original 码农的荒岛求生 码农的荒岛求生 2018-09-25 21:06

我们继续来讲解链接器的重定位。

程序的运行过程就是CPU不断的从内存中取出指令然后执行执行的过程，对于函数调用来说比如我们在C/C++语言中调用简单的加法函数add，其对应的汇编指令可能是这样的:

```
call 0x4004fd
```

其中0x4004fd即为函数add在内存中的地址，当CPU执行这条语句的时候就会跳转到0x4004fd这个位置开始执行函数add对应的机器指令。

再比如我们在C语言中对一个全局变量g_num不断加一来进行计数，其对应的汇编指令可能是这样的：

```
mov 0x400fda %eax
add $0x1 %eax
```

这里的意思是把内存中 0x400fda 这个地址的数据放到寄存器当中，然后将寄存器中的数据加一，在这里g_num这个全局变量的内存地址就是0x400fda。

好奇的同学可能会问，那这些函数以及数据的内存地址是怎么来的呢？

确定程序运行时的内存地址就是接下来我们要讲解的重点内容，这里先给出答案，可执行文件中代码以及数据的运行时内存地址是链接器指定的，也就是上面示例中add的内存地址0x4004fd其是链接器指定的。确定程序运行时地址的过程就是这里重定位(Relocation)。

为什么这个过程叫做重定位呢，之所以叫做重定位是因为确定可执行文件中代码和数据的运行时地址是分为两个阶段的，在第一个阶段中无法确定这些地址，只有在第二个阶段才可以确定，因此就叫做重定位。接下来让我们来看看这两个阶段，合并同类型段以及引用符号的重定位。

编译器的工作

让我们回忆一下前几节的内容，源文件首先被编译器编译生成目标文件，目标文件种有三段内容：数据段、代码段以及符号表，所有的函数定义被放在了代码段，全局变量的定义放在了数据段，对外部变量的引用放到了符号表。

编译器在将源文件编译生成目标文件时可以确定一下两件事：

- 定义在该源文件中函数的内存地址
- 定义在该源文件中全局变量的内存地址

注意这里的内存地址其实只是相对地址，相对于谁的呢，相对于自己的。为什么只是一个相对地址呢？因为在生成一个目标文件时编译器并不知道这个目标文件要和哪些目标文件进行链接生成最后的可执行文件，而链接器是知道要链接哪些目标文件的。因此编译器仅仅生成一个相对地址。

而对于引用类的变量，也就是在当前代码中引用而定义是在其它源文件中的变量，对于这样的变量编译器是无法确定其内存地址的，这不是编译器需要关心的，确定引用类变量的内存地址是链接器的任务，链接器在进行链接时能够确定这类变量的

内存地址。因此当编译器在遇到这样的变量时，比如使用了外部定义的函数时，其在目标文件中对应的机器指令可能是这样的:

> *call 0x000000*

也就是说对于编译器不能确定的地址都这设置为空(0x000000)，同时编译器还会生成一条记录，该记录告诉链接器在进行链接时要修正这条指令中函数的内存地址，这个记录就放在了目标文件的.rel.text段中。相应的如果是对外部定义的全局变量的使用，则该记录放在了目标文件的.rel.data段中。即链接器需要在链接过程中根据.rel.data以及.rel.text来填好编译器留下的空白位置
(0x000000)。因此在这里我们进一步丰富目标文件中的内容，如图所示：

目标文件

生成目标文件后，编译器完成任务，编译器确定了定义在该源文件中函数以及全局变量的相对地址。对于编译器不能确定的引用类变量，编译器在目标文件的.rel.text以及.rel.data段中生成相应的记录告诉链接器要修正这些变量的地址。

接下来就是链接器的工作了。

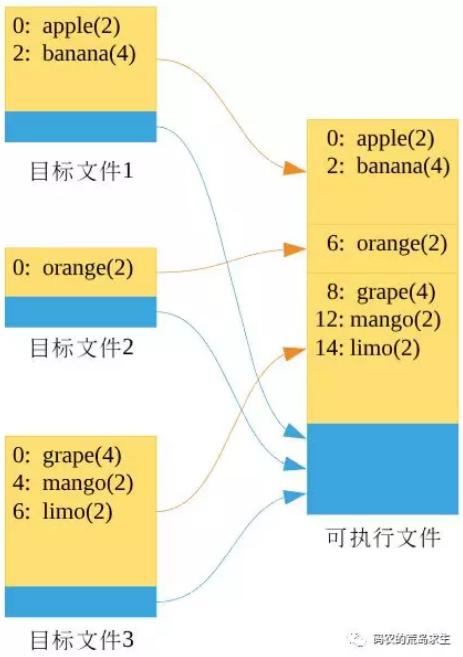
链接器的工作

我们在静态库下可执行文件的生成一节中知道，链接器会将所有的目标文件进行合并，所有目标文件的数据段合并到可执行文件的数据段，所有目标文件的代码段合并到可执行文件的代码段。当所有合并完成后，各个目标文件中的相对地址也就确定了。因此在这个阶段，链接器需要修正目标文件中的相对地址。

在这里我们以合并目标文件中的数据段为例来说明链接器是如何修正目标文件的相对地址的，合并代码段时修正相对位置的原理是一样的。

我们假设链接器需要链接三个目标文件：

- 目标文件一：该文件数据段定义了两个变量apple和banana，apple的长度为2字节，banana的长度4字节，因此目标文件一的数据段长度为6字节。从图中也可以看出apple的内存地址为0，也就是相对地址，即apple这个变量在目标文件一的地址是0，banana的地址为2。
- 目标文件二：该文件的数据段比较简单，只定义了一个变量orange，其长度为2，因此该目标文件的数据段长度为2。
- 目标文件三：该文件的数据段定义了三个变量grape、mango以及limo，其长度分别为4字节、2字节以及2字节，因此该目标文件的数据段长度为8字节。



链接器在链接三个目标文件时其顺序是依次链接的，链接完成后：

- 目标文件一：该数据段的起始地址为0，因此该数据段中的变量的最终地址不变。
- 目标文件二：由于目标文件一的数据段长度为6，因此链接完成后该数据段的起始地址为6(这里的起始地址其实就是偏移offset)，相应的orange的最终内存地址为0+offset即6。
- 目标文件三：由于前两个数据段的长度为8，因此该数据段的起始地址为8(即offset为8)，因此所有该数据段中的变量其地址都要加上该offset，即grape的最终地址为8，即0+offset，mango的最终地址为4+offset即12，limo的最终地址为6+offset即14。

从这个过程中可以看到，数据段中的相对地址是通过这个公式来修正的，即：

> 相对地址 + offset(偏移) = 最终内存地址

而每个段的偏移只有在链接完成后才能确定，因此对相对地址的修正只能由链接器来完成，编译器无法完成这项任务。

当所有目标文件的同类型段合并完毕后，数据段和代码段中的相对地址都被链接器修正为最终的内存位置，这样所有的变量以及函数都确定了其各自位置。

至此，重定位的第一阶段完成。接下来是重定位的第二阶段，即引用符号的重定位。

相对地址是编译器在编译过程中确定了，在链接器完成后被链接器修正为最终地址，而对于编译器没有确定的所引用的外部函数以及变量的地址，编译器将其记录在了.rel.text和.rel.data中。

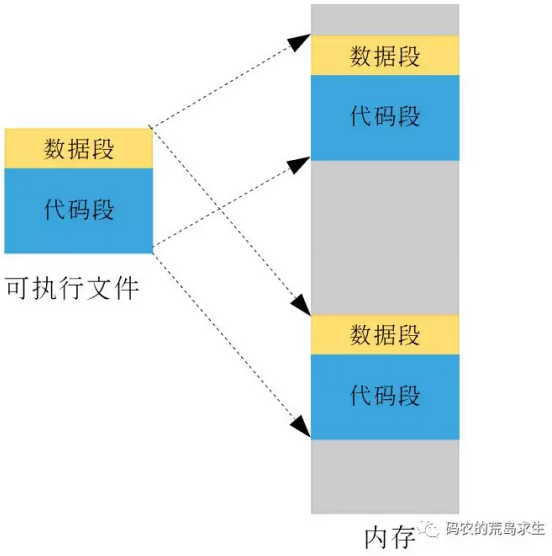
由于在第一阶段中，所有函数以及数据都有了最终地址，因此重定位的第二阶段就相对简单了。我们知道编译器引用外部变量时将机器指令中的引用地址设置为空(比如call 0x000000)，并将该信息记录在了目标文件的.rel.text以及.rel.data段中。因此在这个阶段链接器依次扫描所有的.rel.text以及.rel.data段并找到相应变量的最终地址(这些位置都已在第一阶段确定)，并将机器指令中的0x000000修正为所引用变量的最终地址就可以了。

到这里链接器的重定位就讲解的这里，作为程序员一般很少会有问题出现在重定位阶段，因此这个阶段对程序员相对透明。请同学们注意一点，这里的分析仅限于目标文件的静态链接。我们知道静态链接下，链接器会将需要的代码和数据都合并到可执行文件当中，因此需要确定代码和数据的最终位置。而对于动态链接库来说情况则有所不同，动态链接库可以同时被多个进程使用，如果动态链接库的机器指令中不可以存在引用变量的最终位置，否则在被多个进程使用时会出现一个进程中使用的数据被其它进程修改。因此动态库下的机器指令都是PIC代码，即位置无关代码(Position-Independent Code)。关于PIC的机制原理就不在这里阐述了，对此感兴趣的同学可以关注微信公众号，码农的荒岛求生，我会在那里来讲解。

问题：为什么链接器能确定运行时地址

我们知道只有把可执行文件加载到内存当中程序才可以开始运行。不同的程序会被加载到内存的不同位置。我们从前两节的过程中可以看出，链接器完全没有考虑不同的程序会被加载不同的内存位置被执行。比如对于一个可执行文件我们分别运行

两次，如下图所示，因为两个程序数据段变量的地址是一样的，那么程序一的数据会不会被程序二修改呢？



如果你去试一试的话就会发现显然不会有这种问题的。而当可执行文件加载到内存的时候也不会根据程序加载的起始地址再去修改可执行文件中变量的地址(这样就启动速度就太慢了)，那么操作系统又是如何能做到基于同一个可执行文件的两个程序能在各自的内存空间中运行而不相互干扰呢，链接器在可执行文件中确定的到底是不是程序最终的运行地址呢，我会在后面的文章当中给出答案，欢迎同学们关注微信公共账号码农的荒岛求生获取更多内容。

《彻底理解链接器：六，大型项目是如何被构建(build)出来的》，欢迎关注微信公众号，码农的荒岛求生，获取更多内容。

收录于话题 #链接器 6

‹ 上一篇
彻底理解链接器：四，库与可执行文件的生成

下一篇 ›
彻底理解链接器：六，大型项目是如何被构建出来的