彻底理解链接器：二，符号决议

Original 码农的荒岛求生 码农的荒岛求生 2018-09-20 21:32

收录于话题
#链接器

6个 >

在链接器可操作的元素一节中我们提到，所有的应用程序都是链接器将所需要的一个个简单的目标文件汇集起来形成的。你可以将这个过程想象成拼图游戏，每个拼块就是一个简单的目标文件：

1，拼图游戏当中的每个拼块都依赖于其它拼块提供的拼接口，这就好比我们写的程序模块依赖于其它模块提供的编程接口，比如我们在list.c中实现了一种特定的链表数据结构，其它模块需要使用这种链表，这就是模块间的依赖。而链接器其中一项任务就是要确保提供给链接器进行链接的目标文件集合之间依赖是成立的（也就是说，不会出现在被依赖的模块中链接器找不到需要的接口），这就是后面我们要讲到的符号决议(Symbol Resolution)，开篇提到的第一个问题就来自这个过程。

2，我们在拼图游戏当中通常都是将一整幅图按组成部位一部分一部分拼接好，然后将这些比较完整的大的组成部分拼接成最后一整副图。这就好比链接器会首先将程序每个模块当中目标文件集合链接成库，然后再将各个库进行链接最终形成可执行程序。这就是后面我们要讲到的可执行程序的生成(这也是我们在上一篇文章当中留在本章讨论的)。

3，链接器还有一项任务是无法用这个拼图游戏来类比的，但是这项重要的任务对程序员不可见，作为程序员几乎不会在这个过程遇到问题，这项任务就是重定位。

通过拼图这个游戏的类比，我们给出链接器的工作过程：

首先，链接器对给定的目标文件或库的集合进行符号决议以确保模块间的依赖是正确的。

其次，链接器将给定的目标文件集合进行拼接打包成需要的库或最终可执行文件。

最后，链接器对链接好的库或可执行文件进行重定位。

接下来我们详细的讲解下每一个过程。

首先讲解链接器的符号决议过程。

在这个过程当中，链接器需要做的工作就是确保所有目标文件中的符号引用都有唯一的定义。要想理解这句话我们首先来看看一个典型的c文件里都有些什么。

c源文件中都有什么

如图所示是一个典型的c源文件，该文件中的变量可以划分为两类：

- 全局变量：比如x_global_uninit，x_global_init，fn_c。只要程序没有结束运行，全局变量都可以随时使用。注意，用static修饰的全局变量比如y_global_uninit，其生命周期也等同于程序的运行周期，只是这种全局变量只能在所被定义的文件当中使用，对其它文件不可见。

- 局部变量：比如y_local_uninit，y_local_init，局部局部变量的生命周期和全局变量不同，局部变量变量只能在相应的函数内部使用，当函数调用完成后该函数中的局部变量也就无法使用了。因为局部变量只存在于函数运行时的栈帧当中，函数调用完成后相应的栈帧被自动回收(如果你还不能理解这句话是什么意思没有关系，我会在后面的文章当中详细讲解程序运行时的内存模型)。

```
1  /* 1，定义未初始化的全局变量 */
2  int x_global_uninit;
3
4  /* 2，定义初始化的全局变量 */
5  int x_global_init = 1;
6
7  /* 3，定义未初始化的全局私有变量，该变量只能在当前文件中使用 */
8  static int y_global_uninit;
9
10 /* 4，定义未初始化的全局私有变量，该变量只能在当前文件中使用 */
11 static int y_global_init = 2;
12
13 /* 5，声明全局变量，该变量的定义在其它文件 */
14 extern int z_global;
15
16 /* 6，函数声明，该函数的定义在其它文件 */
```

```c
int fn_a(int x, int y);

/* 7，函数定义，因使用static修饰，该函数只能在当前文件中使用 */
static int fn_b(int x)
{
  return x+1;
}

/* 8，函数定义，该函数可以被其它文件使用 */
int fn_c(int x_local)
{
  /* 9，未初始化的局部变量 */
  int y_local_uninit;
  /* 10，已初始化的局部变量 */
  int y_local_init = 3;

  /* 对全局变量，局部变量以及函数的使用*/
  x_global_uninit = fn_a(x_local, x_global_init);
  y_local_uninit = fn_a(x_local, y_local_init);
  y_local_uninit += fn_b(z_global);
  return (y_global_uninit + y_local_uninit);
}
```


目标文件里有什么

编译器的任务就是把人类可以理解的代码转换成机器可以执行的机器指令，源文件编译后形成对应的目标文件，这个我们在之前的章节中已经多次提到过了。源文件被编译后生成的目标文件中本质上只有两部分：

- 代码部分：你可能会想，一个源文件中不都是代码吗，这里的代码指的是计算机可以执行的机器指令，也就是源文件中定义的所有函数。比如上图中定义的函数fn_b以及fn_c。

- 数据部分：源文件中定义的全局变量。如果是已经初始化后的全局变量，该全局变量的值也存在于数据部分。

到目前为止，你可以把一个目标文件简单的理解为由两部分组成，代码部分中保存的是CPU可以执行的机器指令，这些机器指令来自程序员所定义的函数，编译器将这些定义的函数翻译成机器指令并存放在目标文件的代码部分。数据部分存放的是机器指令所操作的数据。因此目前，你可以简单的将目标文件理解为一个只有两部分的文件，如图所示：

你可能会好奇函数中定义的局部变量为什么没有放到目标文件的数据段当中，这是因为局部变量是函数私有的，局部变量只能在该函数内部使用而全局变量时没有这个限制的，所以函数私有的局部变量被放在了代码段中，作为机器指令的操作数。

编译器在编译过程中遇到外部定义的全局变量或函数时，只要编译器能找到相应的变量声明就会在心里默念“all is well，all is well(一切顺利)”，从这里可以看出编译器的要求还是很低的，至于所使用变量的定义编译器是不会费力去四处搜索，而是愉快的继续接下来的编译。注意，这里再次强调一下，编译器在遇到外部定义的全局变量或者函数时只要能在当前文件找到其声明，编译器就认为编译正确。而寻找使用变量定义的这项任务就被留给了链接器。链接器的其中一项任务就是要确定所使用的变量要有其唯一的定义。虽然编译器给链接器留了一项任务，但为了让链接器工作的轻松一点编译器还是多做了一点工作的，这部分工作就是符号表(Symbol table)。

符号表(Symbol table)

我们在上一节中提到，虽然编译器很不厚道的给链接器留了一项任务，但是编译器为了链接器工作的轻松一点还是做了一点事情，这就是符号表。那符号表中保存的是什么呢，符号表中保存的信息有两部分：

- 该目标文件中引用的全局变量以及函数
- 该目标文件中定义的全局变量以及函数

以上图中的代码为例，编译器在编译过程中每次遇到一个全局变量或者函数名都会在符号表中添加一项，最终编译器会统计出如下所示的一张符号表：

名字	类型	是否可被外部引用	区域
z_global	引用，未定义		
fn_a	引用，未定义		
fn_b	定义	否	代码段
fn_c	定义	是	代码段
x_global_init	定义	是	数据段
y_global_uninit	定义	否	数据段
x_global_uninit	定义	是	数据段
y_global_init	定义	否	数据段

z_global以及fn_a是未定义的，因为在当前文件中，这两个变量仅仅是声明，编译器并没有找到其定义。剩余的变量编译器都可以在当前文件中找到其定义。

fn_b以及fn_c为当前文件定义的函数，因为在代码段。

剩余的符号都是全局变量，因此放在了数据段。

有同学可能会问，为什么全局变量y_global_uninit ， y_global_init以及函数fn_b不可被其它目标文件引用，这是因为这些变量用static修饰过了，在C语言中经static修饰过的函数的函数以及变量都是当前文件私有的，对外部不可见，这里一定要注意。所以static这个关键字的用法就是，如果你认为一个变量只应该被当前文件使用而不暴露给外部，那么你就可以使用static关键字修饰一下。

本质上整个符号表只是想表达两件事：

- 我能提供给其它文件使用的符号
- 我需要其它文件提供给我使用的符号

这里还有一个问题就是，编译器将统计的这张符号表放在哪里了呢？

符号表存放在哪里

在目标文件里有什么这一小节中，我们将一个目标文件简单的划分了两段，数据段和代码段，现在我们要向目标文件中再添加一段，而符号表也被编译器很贴心的放在目标文件中，因此一个目标文件可以理解为如图所示的三段，而符号表中的内容就是上一节当中编译器统计的表格。

目标文件

有了符号表，链接器就可以进行符号决议了。

符号决议的过程

在上一节符号表中，我们知道符号表给链接器提供了两种信息，一个是当前目标文件可以提供给其它目标文件使用的符号，另一个其它目标文件需要提供给当前目标文件使用的符号。有了这些信息链接器就可以进行符号决议了。如图所示，假设链接器需要链接三个目标文件：

链接器会依次扫描每一个给定的目标文件，同时链接器还维护了两个集合，一个是已定义符号集合D，另一个是未定义符合集合U，下面是链接器进行符合决议的过程：

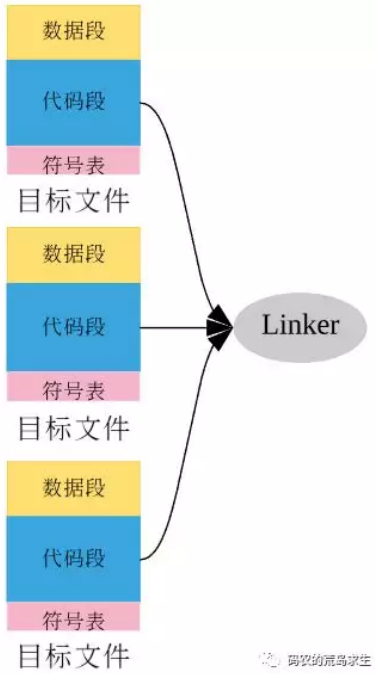
1，对于当前目标文件，查找其符号表，并将已定义的符号并添加到已定义符号集合D中。

2，对于当前目标文件，查找其符号表，将每一个当前目标文件引用的符号与已定义符号集合D进行对比，如果该符号不在集合D中则将其添加到未定义符合集合U中。

3，当所有文件都扫描完成后，如果为定义符号集合U不为空，则说明当前输入的目标文件集合中有未定义错误，链接器报错，整个编译过程终止。

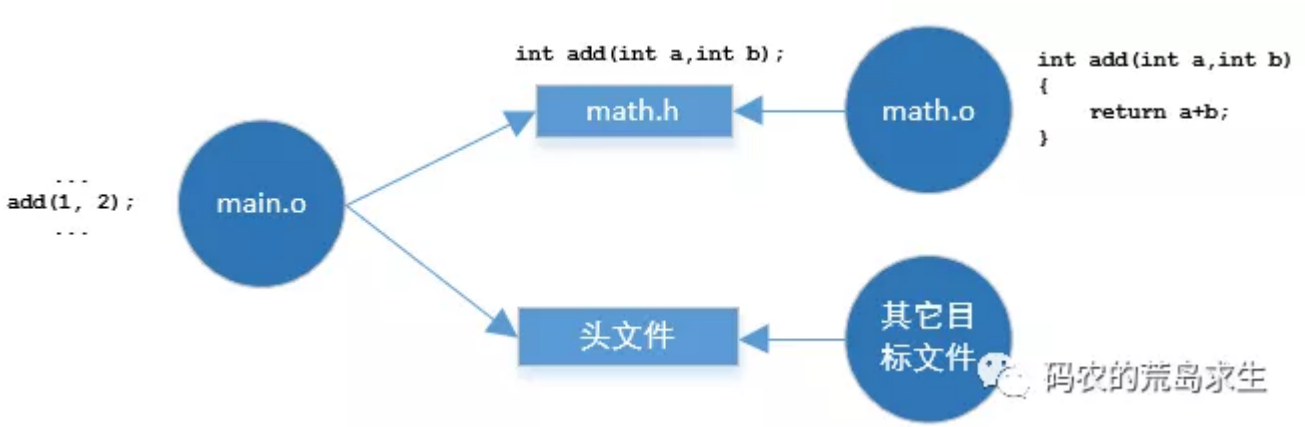
上面的过程看似复杂，其实用一句话概括就是只要每个目标文件所引用变量都能在其它目标文件中找到唯一的定义，整个链接过程就是正确的。

如果你觉得上面的解释比较晦涩的话，你也可以将链接符号决议这个过程想象成如下的游戏：
新学期开学后，幼儿园的小朋友们都带了礼物要和其它的小朋友们分享，同时每个小朋友也有自己的心愿单，每个小朋友都可以依照自己的心愿单去其它的小朋友那里拿礼物，整个过程结束后，每个小朋友都能拿到自己想要的礼物。
在这个游戏当中，小朋友就好比目标文件，每个小朋友自己带的礼物就好比每个目标文件的已定义符号集合，心愿单就好比每个目标文件中未定义符号的集合。

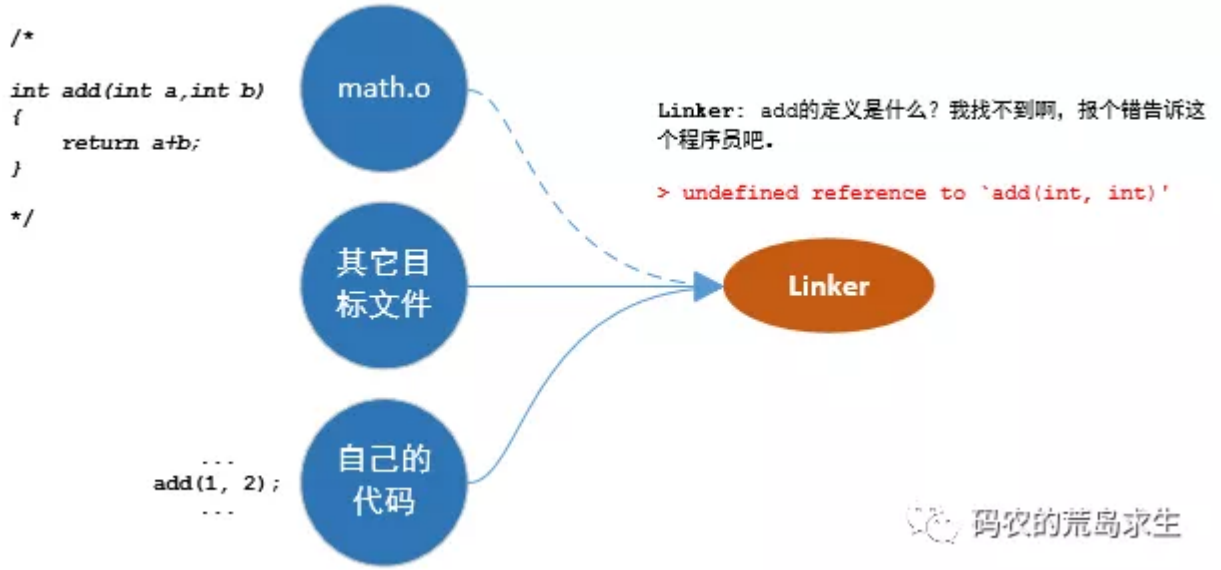


实例说明undefined reference

假设我们写了一个math.c的数字计算程序，其中定义了一个add函数，该函数在main.c中被引用到，那么很简单，我们只需要在main.c中include写好的math.h头文件就可以使用add函数了，如图所示：



但是由于粗心大意，一不小心把math.c中的add函数给注释掉了，当你在写完main.c、打算很潇洒的编译一下时，出现了很经典的undefined reference to `add(int, int)`错误，如图所示：



这个错误其实是这样产生的：
1， 链接器发现了你写的代码math.o中引用了外部定义的add函数(不要忘了，这是通过检查目标文件math.o中的符号表得到的信息)，所以链接器开始查找add函数到底是在哪里定义的。

2，链接器转而去目标文件math.o的目标文件符号表中查找，没有找到add函数的定义。

3，链接器转而去其它目标文件符号表中查找，同样没有找到add函数的定义。

4，链接器在查找了所有目标文件的符号表后都没有找到add函数，因此链接器停止工作并报出错误undefined reference to `add(int, int)'，如上图所示。

因此如果你很清楚链接器符号决议这个过程的话就会进行如下排查：
1：main.c中对add函数的函数名有没有写正确。
2：链接命令中有没有包含math.o，如果没有添加上该目标文件。
3：如果链接命令没有问题，查看math.c中定义的add函数定义是否有问题。
4：如果是C和C++混合编程时，确保相应的位置添加了extern "C"。

一般情况下经过这几个步骤的排查基本能够解决问题。
所以当你再次看到undefined reference这样的错误的是时候，你就应该可以很从容的去解决这类问题了。

接下来我们讲解一下链接器的第二个工作过程，库与可执行文件的生成。

《彻底理解链接器：三，库与可执行文件的生成》，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：一，前言

下一篇 ›
彻底理解链接器：三，库与可执行文件的生成