**高级语言的编译：链接及装载过程介绍**

大龙，志超2015-01-22 15:00

**引言**

随着越来越多功能强大的高级语言的出现，在服务器计算能力不是瓶颈的条件下，很多同学会选择开发效率高，功能强大的虚拟机支持的高级语言（Java），或者脚本语言（Python，Php）作为实现功能的首选，而不会选择开发效率低，而运行效率高的 C/C++ 作为开发语言。而这些语言一般情况下是运行在虚拟机或者解释器中，而不需要直接跟操作系统直接打交道。

虚拟机和解释器相当于为高级语言或者脚本语言提供了一个中间层，隔离了与操作系统之间进行交互的细节，这为工程师们减少了很多与系统底层打交道的麻烦，大大提高了工程师的开发效率。但是这样也造成了工程师们长期工作在高级语言之上，在有时候需要与链接库，可执行文件，CPU 体系结构这些概念有交互的时候会比较陌生。

因此，本文为了让读者可以对源代码如何编译到二进制可执行程序有一个整体的了解，将会从一下几个方面介绍一下程序编译，链接和装载的基本原理。

1. 首先我们介绍一下不同的 CPU 体系结构和不同的操作系统对应的可执行文件的格式。
2. 然后以几个简单的 C 程序为例子，介绍一下编译器和链接器对程序源代码所做的处理。
3. 最后我们看一下程序执行的时候，装载器对程序的处理以及操作系统对其的支持。

**CPU体系结构**

我们现在大部分同学接触到的 PC 机或者服务器使用的 CPU 都是 X86\_64 指令集体系结构，这是一种基于 CISC（复杂指令集体系结构）。我们在计算机组成原理的课程里面都学到，其实CPU指令集类型中除了 CISC，还有另外一种 RISC 类型的 CPU 体系结构，也就是简单指令集体系结构，比如 SUN 的 SPARC 指令集，IBM 的 PowerPC 指令集都是基于 RISC 指令集的 CPU 体系结构。我们这里不去深究各种体系结构的细节，我们关心的是在其中一种 CPU 体系结构中编译的代码能够在另一种体系结构下面运行么？

答案是否定的，因为所谓的二进制程序，其实都是有一条一条的 CPU 指令组成，二进制程序执行的过程中，也是由 CPU 把这些指令 load 到指令流中一条一条执行。不同的 CPU 体系结构的指令集是不一样的，指令的长度和组成都有区别。所以让SPARC的CPU去执行一个编译成 X86 的 CPU 指令集的二进制程序是不可行的。

**操作系统**

记得当年 java 语言刚兴起的时候，有一个很大的卖点就是跨平台执行。为什么能够跨平台执行呢，这是因为 java 程序经过 javac 编译之后得到了一种 java 虚拟机可以执行的字节码文件。只要在不同的操作系统上（Windows，Linux，MacOS）上装上自己所属版本的 java 虚拟机之后，就可以执行在另外一种操作系统下面编译的 java 字节码程序。那么，为什么经过 gcc/g++ 编译过的二进制程序不能跨平台执行呢？

我们刚才说了，java 程序能够跨平台执行是因为不同系统平台上面安装的 java 虚拟机能够识别同一种 java 字节码。那么我们是不是可以推断，不同的操作系统二进制程序不能跨平台执行，是因为不同操作系统下面二进制文件的格式不同呢？

事实确实是这样的。我们都知道的是一个程序编译成二进制之后，运行的时候是从 main 函数开始执行的。但是这个程序是怎么样 load 到内存中的，执行流又是如何准确的定位到 main 函数的地址的。其实这些工作都是操作系统替我们做的。我们可以大胆想一下，main 函数之前操作系统都需要做哪些工作。首先，操作系统肯定要分配一块虚拟地址空间；然后系统需要把二进制程序中的代码和数据 load 到这个地址空间中，随后系统会根据某种特定的文件格式，找到其中某一个特定的位置（初始化段），做一些程序运行前的初始化工作，比如环境变量初始化和全局变量的处理，然后开始执行我们的 main 函数。这里的“某种特定的文件格式”就是为什么二进制程序不能跨平台运行的原因。

这里我真正想说的是，每一种操作系统有自己的二进制文件格式，操作系统把二进制可执行程序load到内存中之后，会根据默认的这种格式寻找各种数据，比如代码段，数据段和初始化段。所以说 Windows 下面的 exe 可执行文件，lib 静态库，dll 动态库是不可以直接运行在 Linux 系统下面的；MacOS 下面的 Mach-O 可执行文件，静态链接库（a库），动态链接库（so库）也是不能够直接放在 Linux 系统下面运行的。反之亦然，Linux 下面的 ELF 可执行文件，静态链接库（a库），动态链接库（so库）同样不能够在 Window 系统下面运行。

**源代码的编译**

说完了 CPU 体系结构和操作系统对二进制文件格式的影响，下面我们从几个例子看一下从源代码文件如何经过处理最终变成一个可执行文件。不同的系统下有不同的编译器，比如 Windows 下有 vs 自带的 C++ 编译器，Linux 和 Unix 下面有 gcc/g++ 编译器。也有很多不同的编程语言，各自有自己的编译器把相应的源代码编译成二进制可执行程序。尽管有些实现细节不同，这些编译器的工作原理和过程是一致的。由于 Linux 和 c语言使用相对广泛，同时笔者对 Linux 和C/C++相对熟悉，本文剩下的部分都是基于在 Linux 平台下使用 gcc/g++ 编译器编译 c/c++ 源代码进行说明和解释。

本文的初衷是让工程师对程序源代码如何通过编译器，链接器和装载器最终成为一个进程运行在系统中的整个过程有一个基本的理解，所以并不会涉及到编译器如何通过进行词法分析，语法分析和语义分析最终得到目标二进制文件。因此本文剩下的部分主要集中在 gcc/g++ 如何形成一个 Linux 认识的 elf 可执行文件的。

**C源码文件**

首先我们简单回顾下 C 源码程序中变量和函数的基本概念。

我们先来区分一下声明和定义的概念。在 C 语言程序中，我们可以声明一个变量和或者一个函数，也可以定义一个变量或者函数。这两个的区别如下：

* 声明一个全局变量或者函数是告诉编译器，在当前的源文件中可能会用到这个变量或者调用这个函数，但是这个变量或者函数不在当前文件中定义，而会在其他的某个文件中定义，请编译器编译本文件的时候不要报错。
* 定义一个变量是告诉编译器在生成的目标文件中预留一个空间，如果变量有初始值，请编译器在目标文件中保存这个初始值。
* 定义一个函数是请编译器在这个文件的目标文件中生成这个函数的二进制代码。

然后我们需要来看一下 C 源码程序中的变量类型和函数类型，最基本的 C 程序（不使用 C++ 功能）是比较简单的，我们可以声明定义变量和局部变量，全局变量和局部变量可以声明为 static 变量和非 static 变量。除此之外，我们可以通过 malloc 动态申请变量。他们的区别如下：

* 非 static 全局变量表示这个变量存在于程序执行的整个生命周期，同时可以被本源码文件之外的其他文件访问到。
* static 全局变量表示这个变量存在于程序执行的整个生命周期，但是只能被本源码文件的函数访问到。
* 非 static 局部变量表示，这个变量只在本变量所在的函数的执行上下文中存在（实际上这种变量是在函数执行栈的函数栈帧中）
* static 局部变量其实属于全部变量的范畴，它存在于程序执行的整个生命周期，但是作用域被局限在定义这个变量的代码块中（大括号包含的范围）
* 动态申请的变量表明这个变量是在运行过程中，由函数动态的从进程的地址空间中申请一块空间，并使用这个空间存储数据。

对于函数而言，我们同样可以定义 static 和非 static 的函数，区别如下：

* 非 static 函数定义表明这是一个全局的函数，可以被本源码文件的其他文件访问到。
* static 函数限制了本函数只能被本源码文件的函数调用。

我们用下面这个小程序来看看编译器对上述这些变量都做了什么样的处理。

int g\_a = 1; //定义有初始值全局变量

int g\_b; //定义无初始值全局变量

static int g\_c; //定义全局static变量

extern int g\_x; //声明全局变量

extern int sub(); //函数声明

int sum(int m, int n) { //函数定义

return m+n;

}

int main(int argc, char\* argv[]) {

static int s\_a = 0; //局部static变量

int l\_a = 0; //局部非static变量

sum(g\_a,g\_b);

return 0;

}

**目标文件**

在我们用 gcc 编译这个程序来查看编译器做了哪些事情之前，其实我们可以简单梳理一下，为了让这个程序能够运行起来，编译器至少都需要做哪些事情。首先，我们都会默认 CPU 会根据程序的代码一条一条执行；其实，当遇到了条件判断或者函数调用，程序就会发生指令流的跳转；还有，程序代码执行的过程中需要操作各种变量指向的数据。从这三个“理所当然”的行为里面，我们可以推断出编译器至少需要做哪些事情。第一，CPU 肯定不能理解这些高级语言代码，编译器需要把代码编译成二进制指令。 第二，指令流跳转的时候，CPU 怎么能找到要跳转的位置，编译器需要为每个定义的函数所在的位置定义一个标签，每个标签有一个地址，调用每个函数的时候就相当于跳转到那个标签指向的地址。 第三，CPU 如何能找到那些变量指向的数据，编译器需要为每一个变量定义一个标签，每个标签同样有一个地址，这个地址指向内存中的数据空间。

我们来实际看一下编译器的行为，我们先把这个这个编译成目标文件看一下：  
gcc -c test.c -o test.o && nm test.o

0000000000000000 D g\_a

0000000000000004 C g\_b

0000000000000000 b g\_c

U g\_x

0000000000000014 T main

0000000000000004 b s\_a.1597

0000000000000000 T sum

首先我们用 gcc -c 命令把 test.c 源码文件编译成 test.o 目标文件，需要注意的是虽然目标文件也是二进制文件，但是和可执行文件是有区别的，目标文件仅仅把当前的源码文件编译成二进制文件，并没有经过链接过程，是不能够执行的。然后我们用 nm 命令可以查看一下目标文件的 symbol 信息。这里我们看到了nm命令的输出默认有三列，其中最左边的一列是变量的相对地址，中间的一列表示变量所在的段的类型，右面表示变量的名字。我们来分别看一下这三列的意思。

* 首先最左边这一列是变量在所在段的相对地址，我们看到 g\_a 和 g\_c 的相对地址是相同的，这并不冲突，因为他们处于不同的段中（ D 和 b 表示它们在目标文件中处于不同的段中)。
* 第二列表示变量所处的段的类型，比如我们这里看到了有 D,C,b,T 这些类型的段，实际上编译器支持的段类型比这个还多。我们同样不去深究各个段类型的意思，只要明白不同的段存放的是不同的数据即可，比如 D 段就是数据段，专门存放有初始值的全局变量，T 段表示代码段，所有的代码编译后的指令都放到这个段中。在这里我们可以注意到同一个段中的变量相对地址是不能重复的。
* 第三列表示变量的名字，这里我们看到局部的静态变量名字被编译器修改为 s\_a.1597，我们应该能猜得到编译器这么做的原因。s\_a 是一个局部静态变量，作用域限制在定义它的代码块中，所以我们可以在不同的作用域中声明相同名字的局部静态变量，比如我们可以在sum函数中声明另外一个 s\_a。但是我们上面提过，局部静态变量属于全局变量的范畴，它是存在于程序运行的整个生命周期的，所以为了支持这个功能，编译器对这种局部的静态变量名字加了一个后缀以便标识不同的局部静态变量。

细心的读者应该能看到，为什么这里的变量声明 g\_x 没有地址呢？我们在C源码文件部分曾经提到过，变量和函数的声明本质上是给编译器一个承诺，告诉编译器虽然在本文件中没有这个变量或者函数定义，但是在其他文件中一定有，所以当编译器发现程序需要读取这个变量对应的数据，但是在源文件中找不到的时候，就会把这个变量放在一个特殊的段（段类型为 U）里面，表示后续链接的时候需要在后面的目标文件或者链接库中找到这个变量，然后链接成为可执行二进制文件。

从上面这些信息的解释其实我们可以看出来，编译器没有那么复杂，它做的任何事情都是为了支持语言级别的功能。

**目标文件的链接**

通过上一个部分的一个小程序，我们讨论了 C 源码文件的基本组成部分，编译器对这些组成部分的处理以及编译器这么做背后的原理。同时我们也留下了一个需要在其他目标文件中寻找的变量名。这一小节，我们讨论一下，在编译器把各个 C 源代码文件编译成目标文件之后，链接器需要对这些目标文件做什么样的处理。

首先我们尝试一下对上一小节得到的目标文件链接一下看看有什么结果：gcc test.o -o test

test.o: In function `main':

test.c:(.text+0x2c): undefined reference to `g\_x'

collect2: ld returned 1 exit status

当我们尝试把这个目标文件进行链接成为可执行文件时，链接器报错了。因为我们之前通过变量声明承诺过的变量并没有在其他的目标文件或者库文件中找到，所以链接器无法得到一个完整可执行程序。我们尝试用另外一个 C 程序修复这个问题：

int g\_x = 100;

int sub() {}

把这个文件编译成目标文件gcc -c test2.c -o test2.o; nm test2.o

0000000000000000 D g\_x

0000000000000000 T sub

现在我们尝试把这两个目标文件链接成为可执行文件：gcc test.o test2.o -o test; nm test, 这时我们发现输出了比目标文件多很多的信息，其中定义了很多为了实现不同语言级别的功能而需要的段，在这里我们关心的是源文件中定义的那些变量对应的 symbol 及其地址，如下图所示：

00000000004005e8 T \_fini

0000000000400390 T \_init

00000000004003d0 T \_start

...

0000000000601018 D g\_a

0000000000601038 B g\_b

0000000000601030 b g\_c

000000000060101c D g\_x

00000000004004c8 T main

0000000000601034 b s\_a.1597

0000000000400504 T sub

00000000004004b4 T sum

在最终的可以执行文件里面，我们可以看到，首先，之前在第一个源文件中声明的变量 g\_x 和声明的函数 sub 最终在第二个目标文件中找到了定义；其次，在不同目标文件中定义的变量，比如 g\_a, g\_x 都会放在了数据段中(段类型为 D)；还有，之前在目标文件中变量的相对地址全部变成了绝对地址。

所以我们再一次进行总结一下链接器需要对源代码进行的处理：

1. 对各个目标文件中没有定义的变量，在其他目标文件中寻找到相关的定义。
2. 把不同目标文件中生成的同类型的段进行合并。
3. 对不同目标文件中的变量进行地址重定位。

这也是链接器所需要实现的最基本的功能。

**装载运行**

上面的几个小节中我们讨论了编译器把一个 C 源码文件编译成一个目标文件需要做的最基本的处理，也讨论了链接器把多个目标文件链接成可执行文件时需要具备的最基本的功能。在这一个小节我们来讨论一下可执行文件如何被系统装载运行的。

**动态链接库**

我们都知道，在我们写程序的过程中，不会自己实现所有的功能，一般情况下会调用我们所需要的系统库和第三方库来实现我们的功能。在上面两个小节的示例代码中，为了说明问题的简单起见，我们仅仅声明，定义了几个变量和函数，并没有使用任何的库函数。那么现在假设我们需要调用一个库函数提供的功能，这个时候可执行文件又是什么样的呢，我们再看一个小例子：

#include <stdio.h>

#include <string.h>

int main(int argc, char\* argv[]) {

char buf[32];

strncpy(buf, "Hello, World\n", 32);

printf("%s",buf);

}

我们把这个文件编译成可执行文件并且查看一下它的symbolsgcc test3.c -o test3; nm test3:

00000000004005b4 T main

U printf@@GLIBC\_2.2.5

U strncpy@@GLIBC\_2.2.5

我们应该能看到类似上述的输出，我们在“目标文件”这一小节曾经看到过这种类型的 symbol。当时是在目标文件中，同样也是没有地址，我们说这是编译器留给链接器到后面的目标文件中寻找变量定义的。但是现在我们检查的是可执行文件，为什么可执行文件里面仍然有这种没有地址的 symbols 呢？

我们前面提到过，编译器没有什么特别的，它做的所有事情都是为了支持编程语言级别的功能，这里同样不例外。这里可执行文件中的“未定义”的 symbols 其实是为了支持动态链接库的功能。

我们先来回顾一下动态链接库应该有一个什么样的功能。所谓动态链接库是指，程序在运行的时候才去定位这个库，并且把这个库链接到进程的虚拟地址空间。对于某一个动态链接库来说，所有使用这个库的可执行文件都共享同一块物理地址空间，这个物理地址空间在当前动态链接库第一次被链接时 load 到内存中。

现在我们看一下二进制文件中对动态链接库中的函数怎么处理的，objdump -D test3 | less，搜索printf我们应该能看到以下内容：

0000000000400490 <strncpy@plt>:

400490: ff 25 6a 0b 20 00 jmpq \*0x200b6a(%rip) # 601000 <\_GLOBAL\_OFFSET\_TABLE\_+0x18>

400496: 68 00 00 00 00 pushq $0x0

40049b: e9 e0 ff ff ff jmpq 400480 <\_init+0x20>

...

00000000004004b0 <printf@plt>:

4004b0: ff 25 5a 0b 20 00 jmpq \*0x200b5a(%rip) # 601010 <\_GLOBAL\_OFFSET\_TABLE\_+0x28>

4004b6: 68 02 00 00 00 pushq $0x2

4004bb: e9 c0 ff ff ff jmpq 400480 <\_init+0x20>

我们看到可执行文件中为 strncpy 和 printf 分别生成了三个代理 symbol，然后代理 symbol 指向的第一条指令就是跳转到\_GLOBAL\_OFFSET\_TABLE\_这个 symbol 对应的代码段中的一个偏移位置，而在 linux 中，这个\_GLOBAL\_OFFSET\_TABLE\_对应的代码段是为了给“地址无关代码”做动态地址重定位用的。我们提过，动态链接库可以映射到不同进程的不同的虚拟地址空间，所以属于“地址无关代码”，链接器把对这个函数的调用代码跳转到程序运行时动态装载地址。

Linux 提供了一个很方便的命令查看一个可执行文件依赖的动态链接库，我们查看一下当前可执行文件的动态库依赖情况：ldd test3:

linux-vdso.so.1 => (0x00007fff413ff000)

libc.so.6 => /lib/x86\_64-linux-gnu/libc.so.6 (0x00007fe202ae7000)

/lib64/ld-linux-x86-64.so.2 (0x00007fe202eb2000)

ldd 命令模拟加载可执行程序需要的动态链接库，但并不执行程序，后面的地址部分表示模拟装载过程中动态链接库的地址。如果尝试多次运行 ldd 命令，我们会发现每次动态链接库的地址都是不一样的，因为这个地址是动态定位的。我们平常工作中，如果某一个二进制可执行文件报错找不到某个函数定义，可以用这个命令检查是否系统丢失或者没有安装某一个动态链接库。

我们在上面的小程序最后加一个sleep(1000);，然后查看一下运行时的内存映射分配，cd /proc/21509 && cat maps，应该可以看到下面这一段：

7feeef61f000-7feeef7d4000 r-xp 00000000 fd:01 135891 /lib/x86\_64-linux-gnu/libc-2.15.so

7feeef7d4000-7feeef9d3000 ---p 001b5000 fd:01 135891 /lib/x86\_64-linux-gnu/libc-2.15.so

7feeef9d3000-7feeef9d7000 r--p 001b4000 fd:01 135891 /lib/x86\_64-linux-gnu/libc-2.15.so

7feeef9d7000-7feeef9d9000 rw-p 001b8000 fd:01 135891 /lib/x86\_64-linux-gnu/libc-2.15.so

我们可以看到进程运行时，系统为 libc 库在进程地址空间中映射了四个段，因为每个段权限不同，所以不能合并为一个段。对这些动态链接库的调用最终会跳转到这里显示的地址中。

根据以上这些信息，我们在这里继续总结一下链接器需要对动态链接库需要做的最基本的事情：

1. 链接库在将目标文件链接成可执行文件的时候如果发现某一个变量或者函数在目标文件中找不到，会按照 gcc 预定义的动态库寻找路径寻找动态库中定义的变量或者函数。
2. 如果链接库在某一个动态链接库中找到了该变量或者函数定义，链接库首先会把这个动态链接库写到可执行文件的依赖库中，然后生成这个当前变量或者函数的代理 symbol.
3. 在\_GLOBAL\_OFFSET\_TABLE\_代码中生成真正的动态跳转指令，并且在库函数（比如strncpy,printf）代理symbol中跳转到\_GLOBAL\_OFFSET\_TABLE\_中相应的偏移位置。

前面我们一直在讨论动态链接库（so库），其实在各个平台下面都有静态链接库，静态链接库的链接行为跟目标文件非常类似，但是由于静态库有一些问题，比如因为每个可执行文件都有静态库的一个版本，这导致库升级的时候很麻烦等问题，现在静态库用的非常少，所以这里我们不去深究。

**main函数之前**

在“操作系统”这一小节中，我们曾简单提过，在程序的 main 函数执行之前，进程需要做一些初始化工作，然后才会调用 main 函数执行程序逻辑。在“动态链接库”在这一小节中，我们提到了对于动态链接库，我们需要在系统启动的时候把需要的库动态链接到进程的地址空间。在本节中，我们综合这些步骤，从可执行文件的目标代码中简单跟踪一下，Linux 是如何把 elf 文件 load 到内存中并且最终调用到 main 函数的。

在“目标文件的链接”这一小节中，我们展示了部分nm test的结果，其中\_start这个 symbol 是故意被留下来的，因为对于 elf 文件格式来说，linux 系统在为进程分配完虚拟地址空间并且把代码 load 到内存之后，是从这\_start对应的地址开始执行的。这个地址记录在 elf 文件的头中，系统读取 elf 文件时可以得到这个地址。下面我们就从\_start这个 symbol 对应的指令开始并追踪一下我们感兴趣的关键点。

0000000000400510 <\_start>:

...

400526: 48 c7 c1 70 06 40 00 mov $0x400670,%rcx

40052d: 48 c7 c7 f4 05 40 00 mov $0x4005f4,%rdi

400534: e8 b7 ff ff ff callq 4004f0 <\_\_libc\_start\_main@plt>

/\*

.start这个段会去执行libc库中的\_\_libc\_start\_main的指令，

这里需要注意一下传给这个函数的两个参数值“0x400670”和“0x4005f4”，

其中一个是\_\_libc\_csu\_init的地址，一个是main函数的地址

\*/

...

00000000004004f0 <\_\_libc\_start\_main@plt>:

4004f0: ff 25 22 0b 20 00 jmpq \*0x200b22(%rip) # 601018 <\_GLOBAL\_OFFSET\_TABLE\_+0x30>

0000000000400670 <\_\_libc\_csu\_init>:

...

4006b0: e8 e3 fd ff ff callq 400498 <\_init>

...

0000000000400498 <\_init>:

...

4004a6: e8 65 02 00 00 callq 400710 <\_\_do\_global\_ctors\_aux>

...

00000000004005f4 <main>:

...

400626: e8 95 fe ff ff callq 4004c0 <strncpy@plt>

...

40063f: e8 9c fe ff ff callq 4004e0 <printf@plt>

...

400649: e8 b2 fe ff ff callq 400500 <sleep@plt>

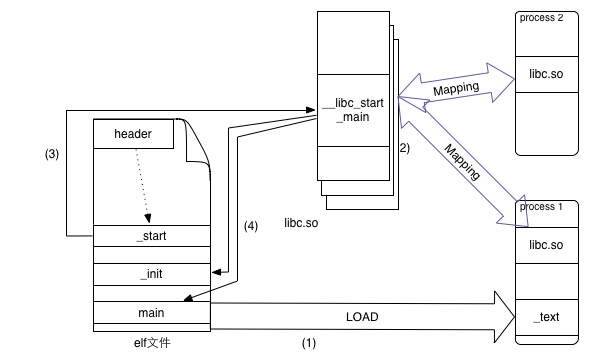
...

我们先来简单解释一下上述贴的几段指令的意思，首先在 \_start 对应的指令中，经过一些处理之后，会用\_\_libc\_csu\_init的地址和main的地址作为参数调用\_\_libc\_start\_main，这个函数是在libc库中实现的，也就是linux中所有的可执行程序都共享同一段初始化代码，篇幅原因我们不去查看\_\_libc\_start\_main的实现了。我们需要知道的是，在\_\_libc\_start\_main作为一些处理之后，会先调用\_\_libc\_csu\_init对应的指令，然后调用main对应的指令。

main对应的指令就是我们自己的main函数了，\_\_libc\_csu\_init接着会调用\_init的指令，然后会调用\_\_do\_global\_ctors\_aux这个 C++ 程序员都应该熟悉的 symbol 对应的指令，\_\_do\_global\_ctors\_aux对应的指令会进行所有的全局变量初始化，或者 C++ 中的全局对象构造等操作。

根据上述信息，我们总结一下当我们通过bash运行一个程序的时候，Linux 做了哪些事情：

1. 首先 bash 进行 fork 系统调用，生成一个子进程，接着在子进程中运行 execve 函数指定的 elf 二进制程序（ Linux中执行二进制程序最终都是通过 execve 这个库函数进行的），execve 会调用系统调用把 elf 文件 load 到内存中的代码段(\_text)中。
2. 如果有依赖的动态链接库，会调用动态链接器进行库文件的地址映射，动态链接库的内存空间是被多个进程共享的。
3. 内核从 elf 文件头得到\_start的地址，调度执行流从\_start指向的地址开始执行，执行流在\_start执行的代码段中跳转到libc中的公共初始化代码段\_\_libc\_start\_main，进行程序运行前的初始化工作。
4. \_\_libc\_start\_main的执行过程中，会跳转到\_init中全局变量的初始化工作，随后调用我们的main函数，进入到主函数的指令流程。



至此，我们讨论了从一个 C 语言程序的源代码，到运行中的进程的全过程。

**一个小例子**

在明白了编译器如何把我们的源代码“转变”成二进制可执行程序之后，我们就能够知道怎么样去看某一段代码被编译成二进制之后是一个什么样子，然后就可以按照编译器的“习惯”写出高效的代码。 这一小节我们分析一个网上的小例子，下面是一个网友列出的两段程序，在面试中被问到各有什么优缺点。

程序1：

if（k > 8）{

for (int h=0;h<100;h++) { //doSomething }

} else {

for (int h=0;h<100;h++) { //doSomething }

}

程序2：

for (int h=0;h<100;h++) {

if (k>8) { //doSomething }

else { //doSomething }

}

从编程规范上看，很明显程序2 是好于程序1 的，因为如果“doSomething”的部分比较复杂，程序2 紧凑而不冗余，而且可以把 if 和 else 分支“doSomething”公共的部分提取出来放在 for 循环下面。但是有经验的工程师马上也能看出来，虽然程序1稍显冗余，但是其执行速度比程序2 是要快的，为什么快我们从编译器生成的目标文件分析一下，我们的测试程序如下：

程序1：

if(type == 0) {

for(i=0; i<cnt; i++) {

sum += data[i];

}

}else if(type == 1) {

for(i=0; i<cnt; i++) {

sum += (i&0x01)?(-data[i]):data[i];

}

}

程序2：

for(i=0; i<cnt; i++) {

if(type == 0) {

sum += data[i];

}else {

sum += (i&0x01)?(-data[i]):data[i];

}

}

编译成可执行文件后的片段为：

程序1：

4005d7: 83 7d ec 00 cmpl $0x0,-0x14(%rbp) /\* type==0判断 \*/

4005db: 75 29 jne 400606 <calc\_1+0x44> /\* 条件判断失败则跳到else分支 \*/

4005dd: c7 45 fc 00 00 00 00 movl $0x0,-0x4(%rbp)

------------------------------循环体开始---------------------------

4005e4: eb 16 jmp 4005fc <calc\_1+0x3a> /\* 跳到循环条件比较指令 \*/

4005e6: 8b 45 fc mov -0x4(%rbp),%eax /\* 循环内第一条指令 \*/

...

4005f8: 83 45 fc 01 addl $0x1,-0x4(%rbp)

4005fc: 8b 45 fc mov -0x4(%rbp),%eax

4005ff: 3b 45 e8 cmp -0x18(%rbp),%eax /\* 循环条件比较 \*/

400602: 7c e2 jl 4005e6 <calc\_1+0x24> /\* 跳到循环开始阶段 \*/

------------------------------循环体结束----------------------------

400604: eb 4a jmp 400650 <calc\_1+0x8e>

400606: 83 7d ec 01 cmpl $0x1,-0x14(%rbp) /\* else分支，type==1判断 \*/

...

/\* type==1 分支基本与type==0的分支是一致的 \*/

程序2：

400671: eb 4d jmp 4006c0 <calc\_2+0x6b>

------------------------------循环体开始---------------------------

400673: 83 7d ec 00 cmpl $0x0,-0x14(%rbp) /\* type==0 \*/

400677: 75 14 jne 40068d <calc\_2+0x38> /\* 条件判断失败则跳到else分支 \*/

...

400686: 8b 00 mov (%rax),%eax

400688: 01 45 f8 add %eax,-0x8(%rbp)

40068b: eb 2f jmp 4006bc <calc\_2+0x67>

40068d: 8b 45 fc mov -0x4(%rbp),%eax /\* else分支 \*/

400690: 83 e0 01 and $0x1,%eax

400693: 84 c0 test %al,%al

400695: 74 13 je 4006aa <calc\_2+0x55>

...

4006c0: 8b 45 fc mov -0x4(%rbp),%eax

4006c3: 3b 45 e8 cmp -0x18(%rbp),%eax /\* 循环条件比较 \*/

4006c6: 7c ab jl 400673 <calc\_2+0x1e> /\* 跳到循环开始阶段 \*/

------------------------------循环体结束----------------------------

通过对比源程序和汇编指令，我们程序1和程序2 编译完之后的汇编指令分别进行了对比标注。我们可以对比一下，在程序1 的汇编指令中，经过一次条件判断之后，执行流会跳到相应的循环指令段中（if/else），然后循环执行整段的指令。而在程序2 的汇编指令中，在每一次执行循环指令段的过程中，都有条件的判断和跳转（if/else）。 所以这里我们可以总结一下程序2 比程序1速度快的原因：

1. 程序2 中每次循环体的执行都需要执行比较指令和跳转指令，如果循环次数非常多（比如大于百万次），就相当于多执行百万条指令。
2. 现代的 CPU 都是流水线模式模式，也有指令预取模块，也就是说同一时间段内，有多条指令在 CPU 内运行，同时也有预测的指令预取。如果发生了指令跳转，就很有可能造成后续的指令全部被刷出 CPU，重新跳转到新地址执行，浪费多个 CPU 周期。

通过我们的分析，我们可以说，如果此程序段处于整个项目中非瓶颈的位置，程序2作为优先选择的是可以接受的。但是如果此程序段处于速度瓶颈位置，程序1是占有优势的。

**结束语**

本文中，我们使用 C 语言为例串讲了源代码程序如何经过编译，链接和装载最终成功在 Linux 系统运行起来。从代码细节上看，这是一个漫长复杂的过程，但是只要抓住其中的主线，就会发现其实编译器和链接器所做的时候都是为了满足我们的功能和需求，正所谓万变不离其宗。

另外虽然我们使用 C 语言进行说明的，在同一种系统中，其他的语言编译得到的二进制文件是一样的格式的。比如在 Linux 下面使用 go 语言编译的源代码，最终编译出来的二进制文件仍然是elf格式的，我们可以使用同一套工具（比如 nm,objdump,readelf）查看和调试这样的代码。由此我们可以得知，虽然go的编译器和gcc编译器细节实现上有所不同，但所做的工作基本是一样的。

显然这样一篇短文不可能很详尽的把编译，链接和装载这么复杂的过程描述的很细致。本文的初衷是为了让同学们对这个过程有一个直观的了解，有兴趣的同学其实还有大把的细节可以去探索，最后的“参考”小节中有几个不错的资源，可以为有兴趣的同学提供参考。

**参考**

1. 俞甲子，石凡，潘爱民: 程序员的自我修养
2. <http://www.lurklurk.org/linkers/linkers.html>
3. John Levine: Linkers and Loaders