

万万没想到，一个可执行文件原来包含了这么多信息！

C语言与C++编程 2020-07-01

The following article is from 编程珠玑 Author 守望先生



编程珠玑

Linux, C语言, C++, 数据结构与算法, 计算机基础, 数据库, 工具, 资源【...

作者：守望，Linux应用开发者，目前在公众号【编程珠玑】分享Linux/C/C++/数据结构与算法/工具等原创技术文章和学习资源。

拿到一个编译好的可执行文件，你能获取到哪些信息？文件大小，修改时间？文件类型？除此之外呢？实际上它包含了很多信息，这些你都知道吗？

示例程序

```
//main.c
#include<stdio.h>
void testFun()
{
    printf("公众号：编程珠玑\n");
}
int main(void)
{
    testFun();
    return 0;
}
```

编译得到可执行文件main:

```
$ gcc -o main main.c
```

ELF头信息

只需要一条简单的命令，就可以获取很多信息

```
$ readelf -h main
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                               ELF64
  Data:                               2's complement, little endian
  Version:                             1 (current)
  OS/ABI:                               UNIX - System V
  ABI Version:                           0
  Type:                                 EXEC (Executable file)
  Machine:                               Advanced Micro Devices X86-64
  Version:                               0x1
  Entry point address:                   0x400430
  Start of program headers:              64 (bytes into file)
  Start of section headers:              6648 (bytes into file)
  Flags:                                 0x0
  Size of this header:                    64 (bytes)
  Size of program headers:                56 (bytes)
  Number of program headers:              9
  Size of section headers:                64 (bytes)
  Number of section headers:              31
  Section header string table index:     28
```

程序位数

Class: **ELF64**

Class展示了该程序的位数，如这里显示的是**ELF64**，如果你将它放到一个**32**位系统中运行，运行得起来就怪了。换句话说，**64**位系统上能运行**32**位和**64**位的程序，但是**32**位系统上，无法运行**64**位的程序。

大小端

Data: **2's complement, little endian**

还记得那个到处可见的面试题吗？如何判断当前CPU是大端还是小端？除了各种秀代码的方式，你想到这个方式了吗？

找一个该平台上的正运行的可执行文件或系统库，然后使用`readelf -h`看一下，是不是很快就看出来？多么明显的little endian。

关于大小端，更多内容可参考《谈谈字节序的问题》。

运行平台

Machine: Advanced Micro Devices X86-64

做嵌入式相关的可能经常需要做交叉编译，而编译出来的程序到底对不对呢？比如你在86平台编译arm的程序，最终生成的可执行文件到底能不能运行在arm平台呢？通过Machine字段就可以很容易确定，从这里可以看到，它是运行在x86平台的。

同样的，当你在交叉编译的时候，发现总有一个库链接不上，但是库又存在，不妨看看这个库和你要编译的平台是否匹配。

链接了哪些动态库？

编好的程序依赖了哪些动态库呢？可不要放到另外一个平台就起不来啊。瞅瞅：

```
$ ldd main
linux-vdso.so.1 => (0x00007ffe750e7000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f749920a000)
/lib64/ld-linux-x86-64.so.2 (0x00007f74995d4000)
```

原来链接了这些库，所以当你在网上下载一些程序，运行的时候提示你某些so找不到，不妨看看它链接的动态库在什么位置，你的机器上到底有没有吧。

新增的函数和全局变量包含了吗？

新增了一个全局变量或者函数，但是编译完之后，不确定有没有？

```
$ nm main |grep testFun
0000000000400526 T testFun
```

`nm`看下就知道了。当然了，如果你看到某个库的函数前面的标志不是**T**，而是**U**，说明该函数未在该库中定义。

`nm`主要用于查看**elf**文件的符号表信息。

有符号表吗

我们都知道，没有符号表的程序，在**core**之后是没有太多有效信息可看的，也是无法使用**gdb**正常调试的，这个在《**GDB**调试入门，看这篇就够了》中已经有提到了，那么怎么看有没有符号表呢？

```
$ file main
main: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically link
```

咦？你看这里是不是也可以看到程序位数，适用平台等信息？

如果使用**file**命令看到最后是**not stripped**，那么则含有符号表，一般线上的程序可能会选择去掉符号表信息，因为可以大大减少可执行文件的空间占用。

```
$ strip main
```

这个时候再看看：

```
$ nm main
no main symbols
```

程序占用空间太大？

为什么程序的占用空间这么大？不妨看看是不是使用了过多的静态变量或全局变量：

```
$ size main
   text      data      bss      dec      hex filename
   1261       552        8     1821     71d main
```

看到**data**部分的大小了吗？看起来并没有多少，如果这里占用空间过大，那可能是你程序中用到了太多的全局变量和静态变量或常量。当然了，**如果你的全局变量都是初始化为0的，那么data这里是不会有明显的变化的（为什么？）**。

在开头分别加下面这一行，其影响可执行文件的效果不一样奥。

```
char str[1000] = {0};
char str[1000] = {1};
```

包含某个字符串吗

这个程序里面包含什么特殊的字符串吗？可以搜索一下：

```
$ strings main |grep hello
hello,
```

嗯？这样一想，好像还可以把版本号信息写进去呢。

C还是C++？

如果将前面的程序按照C++编译：

```
$ g++ -o main main.c
$ nm main |grep test
0000000000400526 T _Z7testFunv
```

你会发现使用g++编译出来的test函数符号前带头，后带尾，**这也是C++中有重载和C中**

没有重载的原因之一。

函数的汇编代码是？

反汇编所有代码：

```
$ objdump -d main
```

那如果要反汇编特定函数(如main函数)呢？先按照地址顺序输出符号表信息：

```
$ nm -n main |grep main -A 1
000000000400537 T main
000000000400550 T __libc_csu_init
```

我们得到main的开始地址为0x400537，结束地址为0x400550。

反汇编：

```
$ objdump -d main --start-address=0x400537 --stop-address=0x400550
000000000400537 <main>:
400537: 55                      push    %rbp
400538: 48 89 e5                mov     %rsp,%rbp
40053b: b8 00 00 00 00          mov     $0x0,%eax
400540: e8 e1 ff ff ff          callq   400526 <testFun>
400545: b8 00 00 00 00          mov     $0x0,%eax
40054a: 5d                      pop     %rbp
40054b: c3                      retq
40054c: 0f 1f 40 00             nopl    0x0(%rax)
```

看看只读数据区有哪些内容？

当我们尝试修改常量字符串的时候，编译器会提示我们，它们是只读的，真的如此吗？

```
$ readelf main -x .rodata
Hex dump of section '.rodata':
 0x004005d0 01000200 00000000 68656c6c 6f2ce585 .....hello,...
 0x004005e0 ace4bc97 e58fb7ef bc9ae7bc 96e7a88b .....
 0x004005f0 e78fa0e7 8e9100 .....

```

看到了吗？我们的**hello**，字符串放在了这里。

总结

本文仅列出了一些比较常见的可执行文中能读到的信息，欢迎补充。

思考

对于**a**和**b**，它们的内存存储区域是一样的吗？为什么？

```
char *a = "hello,world";
char a[] = "hello,world";
```

sizeof计算**a**和**b**的大小一样吗？又为什么？

●编号760，输入编号直达本文

●输入m获取文章目录

C语言与C++编程

分享C/C++技术文章

喜欢此内容的人还喜欢

来看一道“简单的”C语言面试题

C语言与C++编程

白居易最暖心的一首诗：除了你，我谁都不想要

诗词世界

淑宝，谢谢你！

中央广电总台中国之声