聊聊Linux动态链接中的PLT和GOT（1）——何谓PLT与GOT

在介绍PLT和GOT出场之前，先以一个简单的例子引入两个主角，各位请看以下代码：

```c
#include <stdio.h>

void print_banner()
{
    printf("Welcome to World of PLT and GOT\n");
}

int main(void)
{
    print_banner();

    return 0;
}
```

编译:

```
gcc -Wall -g -o test.o -c test.c -m32
```

链接:

```
gcc -o test test.o -m32
```

注意：现代Linux系统都是x86_64系统了，后面需要对中间文件test.o以及可执行文件test反编译，分析汇编指令，因此在这里使用-m32选项生成i386架构指令而非x86_64架构指令。

经编译和链接阶段之后，test可执行文件中print_banner函数的汇编指令会是怎样的呢？我猜应该与下面的汇编类似：

```
080483cc <print_banner>:
 80483cc:    push %ebp
 80483cd:    mov  %esp, %ebp
 80483cf:    sub  $0x8, %esp
 80483d2:    sub  $0xc, %esp
 80483d5:    push $0x80484a8
 80483da:    call **<printf函数的地址>**
 80483df:    add $0x10, %esp
 80483e2:    nop
 80483e3:    leave
 80483e4:    ret
```

print_banner函数内调用了printf函数，而printf函数位于glibc动态库内，所以在编译和链接阶段，链接器无法知知道进程运行起来之后printf函数的加载地址。故上述的**<printf函数地址>** 一项是无法填充的，只有进程运运行后，printf函数的地址才能确定。

那么问题来了：进程运行起来之后，glibc动态库也装载了，printf函数地址亦已确定，上述call指令如何修改（重定位）呢?

一个简单的方法就是将指令中的**<printf函数地址>**修改printf函数的真正地址即可。

但这个方案面临两个问题：

```
现代操作系统不允许修改代码段，只能修改数据段
如果print_banner函数是在一个动态库（.so对象）内，修改了代码段，那么它就无法做到系统内所有进程共享同一个动态库。
```

因此，printf函数地址只能回写到数据段内，而不能回写到代码段上。

注意：刚才谈到的回写，是指运行时修改，更专业的称谓应该是运行时重定位，与之相对应的还有链接时重定位。

说到这里，需要把编译链接过程再展开一下。我们知道，每个编译单元（通常是一个.c文件，比如前面例子中的test.c）都会经历编译和链接两个阶段。

编译阶段是将.c源代码翻译成汇编指令的中间文件，比如上述的test.c文件，经过编译之后，生成test.o中间文件。print_banner函数的汇编指令如下（使用强调内容**objdump -d test.o**命令即可输出）：

```
00000000 <print_banner>:
       0:  55                   push %ebp
       1:  89 e5                mov %esp, %ebp
       3:  83 ec 08             sub   $0x8, %esp
       6:  c7 04 24 00 00 00 00 movl  $0x0, (%esp)
       d:  e8 fc ff ff ff       call  e <print_banner+0xe>
      12:  c9                   leave
      13:  c3                   ret
```

是否注意到call指令的操作数是fc ff ff ff，翻译成16进制数是0xfffffffc（x86架构是小端的字节序），看成有符号是-4。这里应该存放printf函数的地址，但由于编译阶段无法知道printf函数的地址，所以预先放一个-4在这里，然后用重定位项来描述：**这个地址在链接时要修正，它的修正值是根据printf地址（更确切的叫法应该是符号，链接器眼中只有符号，没有所谓的函数和变量）来修正，它的修正方式按相对引用方式。**

这个过程称为链接时重定位，与刚才提到的运行时重定位工作原理完全一样，只是修正时机不同。

链接阶段是将一个或者多个中间文件（.o文件）通过链接器将它们链接成一个可执行文件，链接阶段主要完成以下事情：

```
各个中间文之间的同名section合并
对代码段，数据段以及各符号进行地址分配
链接时重定位修正
```

除了重定位过程，其它动作是无法修改中间文件中函数体内指令的，而重定位过程也只能是修改指令中的操作数，换句话说，链接过程无法修改编译过程生成的汇编指令。

那么问题来了：**编译阶段怎么知道printf函数是在glibc运行库的，而不是定义在其它.o中**

答案往往令人失望：**编译器是无法知道的**

那么编译器只能老老实实地生成调用printf的汇编指令，printf是在glibc动态库定位，或者是在其它.o定义这两种情况下，它都能工作。如

果是在其它.o中定义了printf函数，那在链接阶段，printf地址已经确定，可以直接重定位。如果printf定义在动态库内（链接阶段是可以知道printf在哪定义的，只是如果定义在动态库内不知道它的地址而已），链接阶段无法做重定位。

根据前面讨论，运行时重定位是无法修改代码段的，只能将printf重定位到数据段。那在编译阶段就已生成好的call指令，怎么感知这个已重定位好的数据段内容呢？

答案是：**链接器生成一段额外的小代码片段，通过这段代码支获取printf函数地址，并完成对它的调用。**

链接器生成额外的伪代码如下：

```
.text
...

// 调用printf的call指令
call printf_stub
...

printf_stub:
    mov rax, [printf函数的储存地址] // 获取printf重定位之后的地址
    jmp rax // 跳过去执行printf函数

.data
...
printf函数的储存地址：
    这里储存printf函数重定位后的地址
```

链接阶段发现printf定义在动态库时，链接器生成一段小代码print_stub，然后printf_stub地址取代原来的printf。因此转化为链接阶段对printf_stub做链接重定位，而运行时才对printf做运行时重定位。

动态链接姐妹花PLT与GOT

前面由一个简单的例子说明动态链接需要考虑的各种因素，但实际总结起来说两点：

```
需要存放外部函数的数据段
获取数据段存放函数地址的一小段额外代码
```

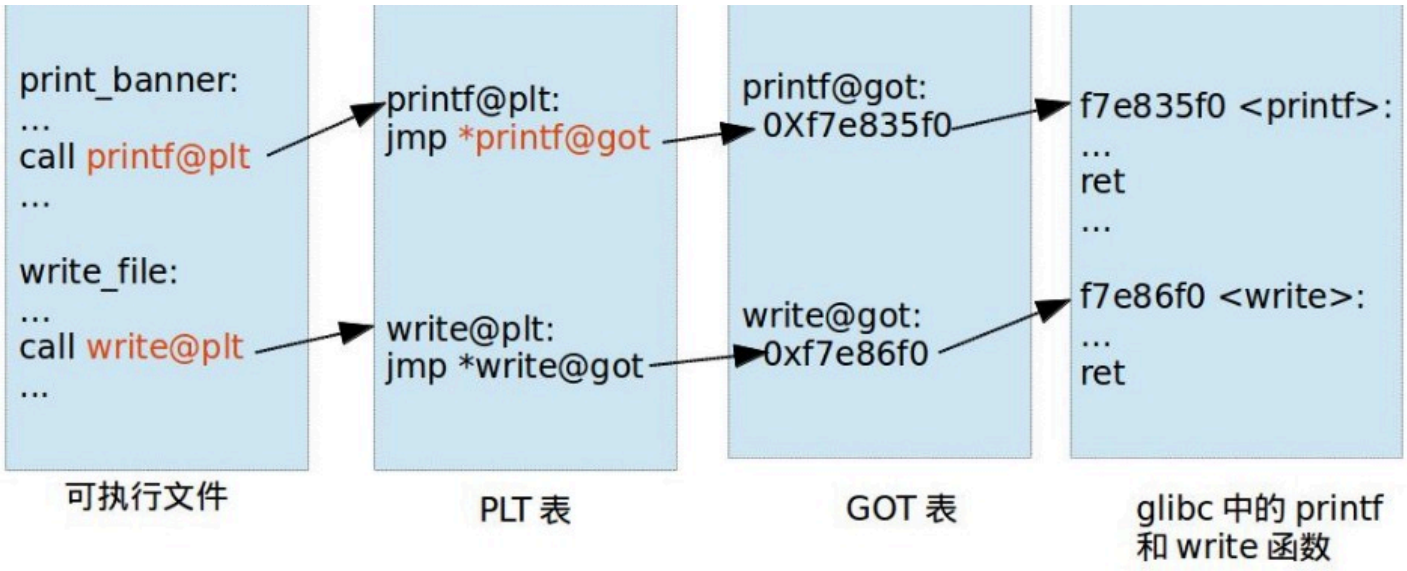
如果可执行文件中调用多个动态库函数，那每个函数都需要这两样东西，这样每样东西就形成一个表，每个函数使用中的一项。

总不能每次都叫这个表那个表，于是得正名。存放函数地址的数据表，称为重局偏移表（GOT, Global Offset Table），而那个额外代码段表，称为程序链接表（PLT，Procedure Link Table）。它们两姐妹各司其职，联合出手上演这一出运行时重定位好戏。

那么PLT和GOT长得什么样子呢？前面已有一些说明，下面以一个例子和简单的示意图来说明PLT/GOT是如何运行的。

假设最开始的示例代码test.c增加一个write_file函数，在该函数里面调用glibc的write实现写文件操作。根据前面讨论的PLT和GOT原理，test在运行过程中，调用方（如print_banner和write_file)是如何通过PLT和GOT穿针引线之后，最终调用到glibc的printf和write函数的？

我简单画了PLT和GOT雏形图，供各位参考。



当然这个原理图并不是Linux下的PLT/GOT真实过程，Linux下的PLT/GOT还有更多细节要考虑了。这个图只是将这些躁声全部消除，让大家明确看到PLT/GOT是如何穿针引线的。