# **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：

#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

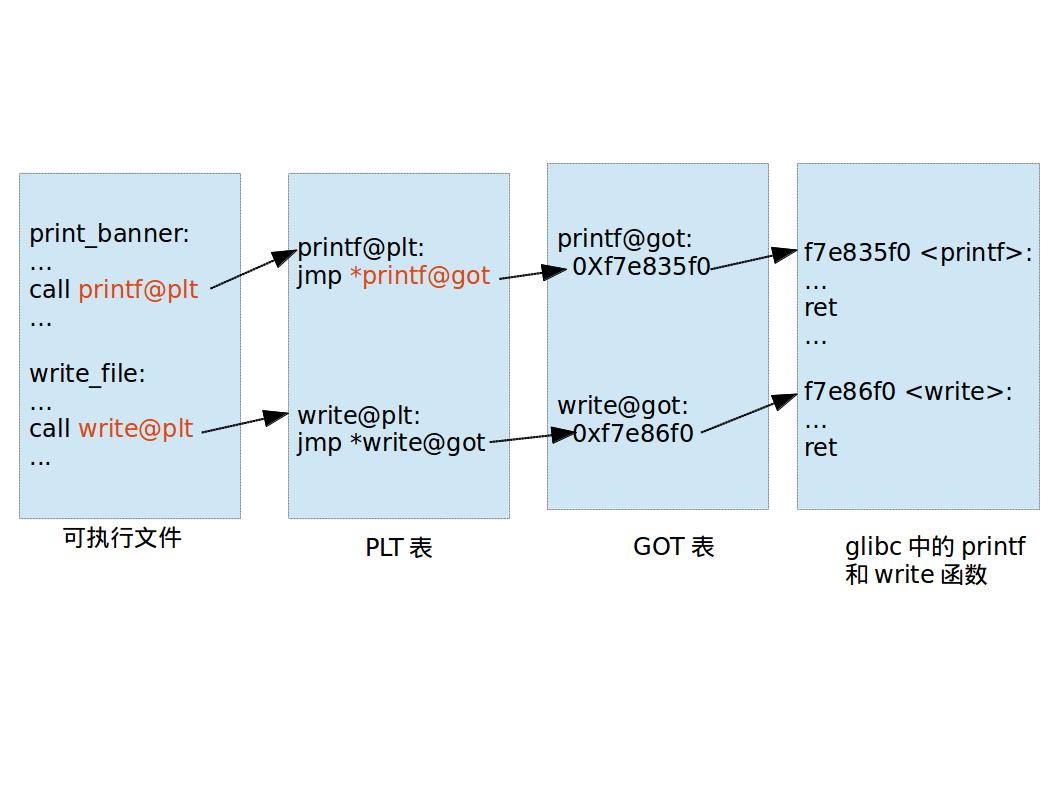
如果可执行文件中调用多个动态库函数，那每个函数都需要这两样东西，这样每样东西就形成一个表，每个函数使用中的一项。

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。

# **聊聊Linux动态链接中的PLT和GOT（２）——延迟重定位**

在上文中（[聊聊Linux动态链接中的PLT和GOT（１）——　何谓PLT与GOT](http://blog.csdn.net/linyt/article/details/51635768" \t "http://blog.csdn.net/linyt/article/details/_blank)）介绍解决动态库函数调用使用GOT表技术，然后PLT从GOT中获取地址并完成调用。这个前提是GOT必须在PLT执行之前，所有函数都已完成运行时重定位。

然而在Linux的世界里面，几乎所有可能的事情，都尽可能地延迟推后，直至无法退避时，才做最后的修正工作。典型的案例有：

* fork之后父子进程内存的写时拷贝机制
* Linux用户态内存空间分配与物理内存分配机制
* C++库的string类写时拷贝机制

当然，也少不了动态链中的延迟重定位机制。

# 延迟重定位

如果可执行文件调用的动态库函数很多时，那在进程初始化时都对这些函数做地址解析和重定位工作，大大增加进程的启动时间。所以Linux提出延迟重定位机制，只有动态库函数在被调用时，才会地址解析和重定位工作。

进程启动时，先不对GOT表项做重定位，等到要调用该函数时才做重定位工作。要实现这个机制必须要有一个状态描述该GOT表项是否已完重定位。

一个显而易见的方案是在GOT中增加一个状态位，描述一个GOT表项是否已完成重定位，那么每个函数就有两个GOT表项了。相应的PLT伪代码如何：

void printf@plt()

{

if (printf@got[0] ！= RELOCATED) { // 如果没完成重定位

调用重定位函数

printf@got[1] = 地址解析发现的printf地址;

printf@got[0] = RELOCATED;

}

jmp \*printf@got[1];

}

这个方案每个函数使用两个GOT表项，占用内存明显增长了一倍。但仔细观察GOT表项中的状态位和真实地址项，这两项在任何时候都不会同时使用，那么这两个变量能复用一个GOT项来实现呢？答案是可以的，Linux动态链接器就使用类似的巧妙方案，将这两个GOT表项合二为一。

具体怎么做呢？很简单，先将上面的代码倒过来写：

void printf@plt()

{

address\_good:

jmp \*printf@got // 链接器将printf@got填成下一语句lookup\_printf的地址

lookup\_printf:

调用重定位函数查找printf地址，并写到printf@got

goto address\_good;

}

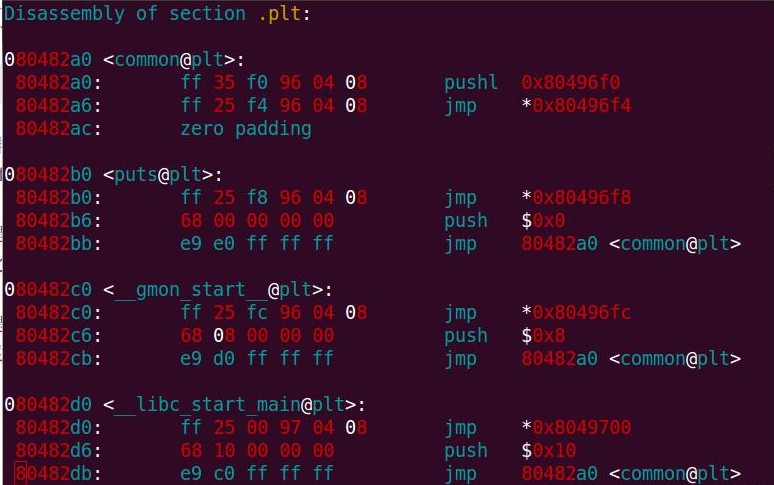
****在链接成可执行文件test时，链接器将printf@got表项的内容填写lookup\_printf标签的地址****。

也即是程序第一次调用printf是时，通过printf@got表项引导到查找printf的plt指令的后半部分。在后半部分中跳到动态链接器中将printf址解析出来，并重定位回printf@got项内。

那么神奇的作用来，第二次调用printf时，通过printf@got直接跳到printf函数执行了。

下面是test可执行文件，通过objdump -d test > test.asm命令反编译之后生成汇编代码，可以看到整个跳转过程。

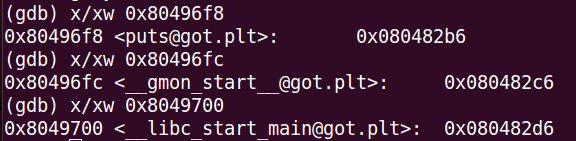
下面是test.asm文件中与PLT/GOT相关的部分，并对一些容易引起误解的地方做了修改。



我将第一项plt表修改成**<common@plt>**项了，objdump -d输出结果会使用错误的符号名。那是因为该项是没有符号的，而objdump输出时，给它找了一个地址接近符号，所以会显示错误的符号名，为了避免引起误解，直接删掉。

每个plt指令中的jmp \*0xf80496xx 都是访问相应的got项。在函数第一次调用之前，这些got项的内容都是链接器生成的，它的值指向对应plt中jmp的下一条指令。

下面是使用gdb命令，查看test可执行文中函数的got表内容，如下：



将两张图对照一下，就可以看到前面说到的规律。

最后所有plt都跳转到common@plt中执行，这是动态链接做符号解析和重定位的公共入口，而不是每个plt表都有重复的一份指令。为了减少PLT指令条数，Linux提炼成了公共函数。从这一点来看，Linux也是拼了。

# **聊聊Linux动态链接中的PLT和GOT（３）——公共GOT表项**

前文（[聊聊Linux动态链接中的PLT和GOT（２）——延迟重定位](http://blog.csdn.net/linyt/article/details/51636753" \t "http://blog.csdn.net/linyt/article/details/_blank)）提到所有动态库函数的plt指令最终都跳进公共plt执行，那么公共plt指令里面的地址是什么鬼？

把test可执行文的共公plt贴出来：

080482a0 <puts@plt-0x10>:

80482a0: pushl 0x80496f0

80482a6: jmp \*0x80496f4

...

第一句，pushl 0x80496f0，是将地址压到栈上，也即向最终调用的函数传递参数。   
第二句，jmp \*0x80496f4，这是跳到最终的函数去执行，不过猜猜就能想到，这是跳到能解析动态库函数地址的代码里面执行。

****0x80496f4****里面住着是何方圣呢？下面使用gdb调试器将它请出来：

$ gdb -q ./test...

(gdb)x/xw 0x80496f40x80496f4 <\_GLOBAL\_OFFSET\_TABLE\_+8>: 0x00000000

(gdb) b main

Breakpoint 1 at 0x80483f3

(gdb) r

Starting program: /home/ivan/test/test/test

Breakpoint 1, 0x80483f3 in main ()

(gdb) x/xw 0x80496f40x80496f4 <\_GLOBAL\_OFFSET\_TABLE\_+8>: 0xf7ff06a0

从调试过程可以发现，0x80496f4属于GOT表中的一项，进程还没有运行时它的值是0x00000000，当进程运行起来后，它的值变成了0xf7ff06a0。如果做更进一步的调试会发现这个地址位于动态链接器内，对应的函数是****\_dl\_runtime\_resolve****。

嗯，是不是想到了什么呢。所有动态库函数在第一次调用时，都是通过XXX@plt -> 公共@plt -> \_dl\_runtime\_resolve调用关系做地址解析和重定位的。

谈到这里，其实还有谜底是没有解开的，以printf函数为例：

* \_dl\_runtime\_resolve是怎么知要查找printf函数的
* \_dl\_runtime\_resolve找到printf函数地址之后，它怎么知道回填到哪个GOT表项
* 到底\_dl\_runtime\_resolve是什么时候被写到GOT表的

前２个问题，只需要一个信息就可以了知道，这个信息就在藏在在函数对应的xxx@plt表中，以printf@plt为例：

printf@plt>:

jmp \*0x80496f8

push $0x00

jmp common@plt

第二条指令就是秘密所在，每个xxx@plt的第二条指令push的操作数都是不一样的，它就相当于函数的id，动态链接器通过它就可以知道是要解析哪个函数了。

真有这么神吗？这不是神，是编译链接器和动态链接器故意安排的巧合罢了。

使用readelf -r test命令可以查看test可执行文件中的重定位信息，其中****.rel.plt****这一段就大有秘密：

$ readelf -r test

....

Relocation section '.rel.plt' at offset 0x25c contains 3 entries:

Offset Info Type Sym.Value Sym. Name

080496f8 00000107 R\_386\_JUMP\_SLOT 00000000 puts

080496fc 00000207 R\_386\_JUMP\_SLOT 00000000 \_\_gmon\_start\_\_

08049700 00000407 R\_386\_JUMP\_SLOT 000000000 \_\_libc\_start\_main

再看看各函数plt指令中的push操作数：   
printf对应push 0x0   
****gmon\_start****对应push 0x8   
\_\_libc\_start\_main对应push 0x10

这３个push操作数刚好对应３个函数在****.rel.plt****段的偏移量。在\_dl\_runtime\_resolve函数内，根据这个offset和.rel.plt段的信息，就知道要解析的函数。再看看.rel.plt最左边的offset字段，它就是GOT表项的地址，也即\_dl\_runtime\_resolve做完符号解析之后，重定位回写的空间。

****第三个问题：到底\_dl\_runtime\_resolve是什么时候被写到GOT表的****。   
答案很简单，可执行文件在Linux内核通过exeve装载完成之后，不直接执行，而是先跳到动态链接器（ld-linux-XXX）执行。在ld-linux-XXX里将\_dl\_runtime\_resolve地址写到GOT表项内。

事实上，不单单是预先写\_dl\_runtime\_resolve地址到GOT表项中，在i386架构下，除了每个函数占用一个GOT表项外，GOT表项还保留了３个公共表项，也即got的前３项，分别保存：

****got[0]: 本ELF动态段(.dynamic段）的装载地址****   
****got[1]：本ELF的link\_map数据结构描述符地址****   
****got[2]：\_dl\_runtime\_resolve函数的地址****

动态链接器在加载完ELF之后，都会将这３地址写到GOT表的前３项。   
其实上述公共的plt指令里面，还有一个操作数是没有分析的，其实它就是got[1]（本ELF的link\_map)地址，因为只有link\_map结构，结合.rel.plt段的偏移量，才能真正找到该elf的.rel.plt表项。

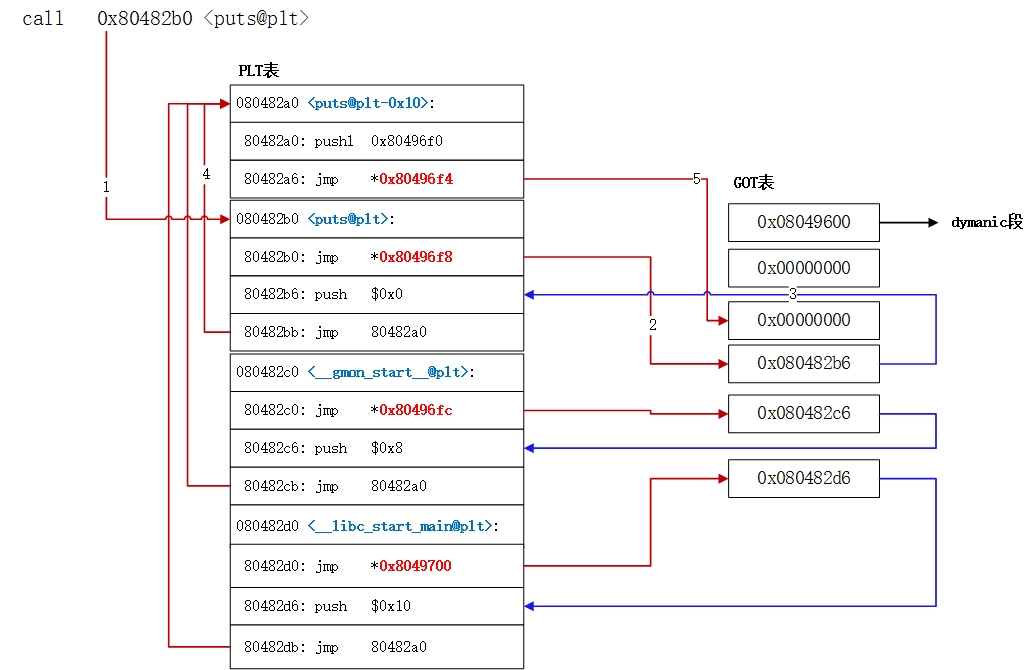
有兴趣的读者可以使用gdb，在执行到main函数时，将GOT表的这３项数据看一下，验证一下。

好了，谈到这里是否对PLT和GOT机制有个更清晰认识了呢？最后一篇会使用图文结构将整个PLT/GOT机制串起来。

# **聊聊Linux动态链接中的PLT和GOT（4）—— 穿针引线**

# 编译时的PLT和GOT关系图

前几篇文章一直在讨论PLT和GOT的结构细节，编译完成之后，PLT和GOT的对应关系是怎么样的呢，下面是编译完成之后，PLT和GOT关系图。



图中重点标注了从调用printf函数语句的汇编指令call puts@plt跳转过程，图中使用编号来表标跳转顺序。

****PLT****表结构有以下特点：

1. PLT表中的第一项为公共表项，剩下的是每个动态库函数为一项（当然每项是由多条指令组成的，jmp \*0xXXXXXXXX这条指令是所有plt的开始指令）
2. 每项PLT都从对应的GOT表项中读取目标函数地址

****GOT****表结构有以下特点：

1. GOT表中前3个为特殊项，分别用于保存 .dynamic段地址、本镜像的link\_map数据结构地址和\_dl\_runtime\_resolve函数地址；但在编译时，无法获取知道link\_map地址和\_dl\_runtime\_resolve函数地址，所以编译时填零地址，进程启动时由动态链接器进行填充
2. 3个特殊项后面依次是每个动态库函数的GOT表项

如果将PLT和GOT抽象起来描述，可以写成以下的伪代码：

plt[0]:

pushl got[1]

jmp \*got[2]

plt[n]: // n >= 1

jmp \*got[n+2] // GOT前3项为公共项，第3项开始才是函数项，plt[1]对应的GOT[3]，依次类推

push (n-1)\*8

jmp plt[0]

got[0] = address of .dynamic section

got[1] = address of link\_map object( 编译时填充0）

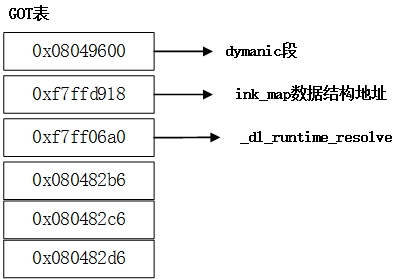
got[2] = address of \_dl\_runtime\_resolve function (编译时填充为0)

got[n+2] = plt[n] + 6 (即plt[n]代码片段的第二条指令）

# 进程起动后的GOT表

PLT属于代码段，在进程加载和运行过程都不会发生改变，PLT指向GOT表的关系在编译时已完全确定，唯一能发生变化的是GOT表。

Linux加载进程时，通过execve系统调用进入内核态，将镜像加载到内存，然后返回用户态执行。返回用户态时，它的控制权并不是交给可执行文件，而是给动态链接器去完成一些基础的功能，比如上述的GOT[1]，GOT[2]的填写就是这个阶段完成的。下图是动态链接器填完GOT[1]，GOT[2]后的GOT图：



估计大家比较好奇的是，动态链接器怎么知道GOT的首地址？这个秘密就藏在ELF的.dynamic段里面，详见下面readelf -d test输出结果中的****PLTGOT****项：

ivan@ivan:~/test/test$ readelf -d test

Dynamic section at offset 0x600 contains 24 entries:

Tag Type Name/Value

0x00000001 (NEEDED) Shared library: [libc.so.6]

0x0000000c (INIT) 0x8048274

0x0000000d (FINI) 0x8048488

0x00000019 (INIT\_ARRAY) 0x80495f4

0x0000001b (INIT\_ARRAYSZ) 4 (bytes)

0x0000001a (FINI\_ARRAY) 0x80495f8

0x0000001c (FINI\_ARRAYSZ) 4 (bytes)

0x00000004 (HASH) 0x8048168

0x00000005 (STRTAB) 0x80481e0

0x00000006 (SYMTAB) 0x8048190

0x0000000a (STRSZ) 74 (bytes)

0x0000000b (SYMENT) 16 (bytes)

0x00000015 (DEBUG) 0x0

0x00000003 (PLTGOT) 0x80496ec

0x00000002 (PLTRELSZ) 24 (bytes)

0x00000014 (PLTREL) REL

0x00000017 (JMPREL) 0x804825c

0x00000011 (REL) 0x8048254

0x00000012 (RELSZ) 8 (bytes)

0x00000013 (RELENT) 8 (bytes)

0x6ffffffe (VERNEED) 0x8048234

0x6fffffff (VERNEEDNUM) 1

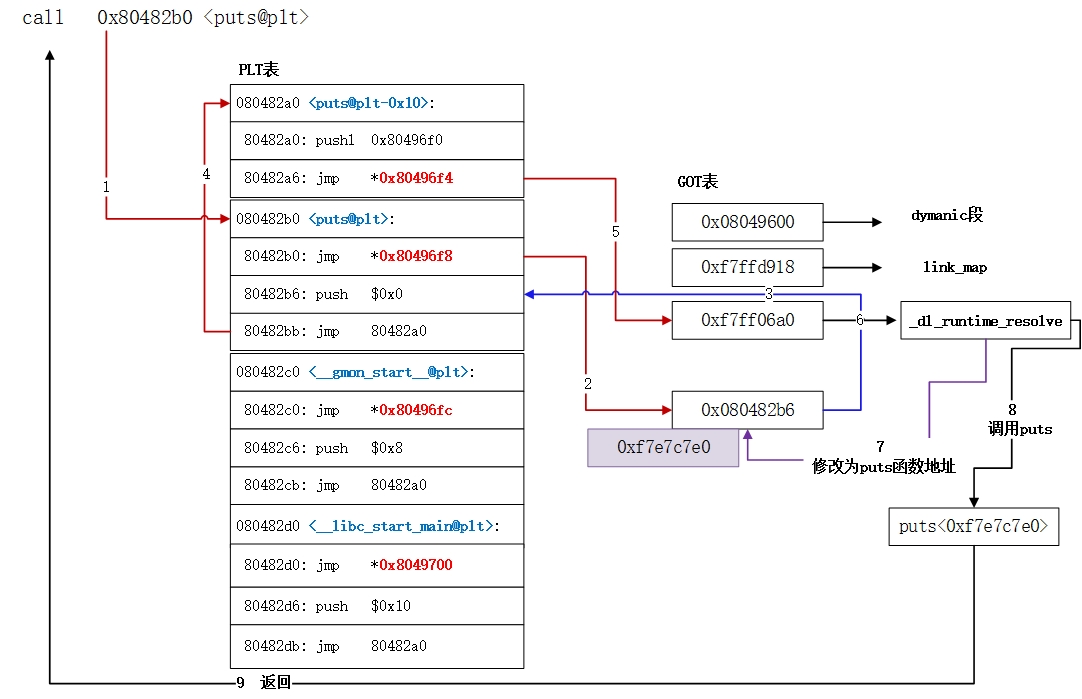
0x6ffffff0 (VERSYM) 0x804822a

0x00000000 (NULL) 0x0

其实.dynamic段还藏着很多其它信息，都是跟动态运行相关的信息，有兴趣的读者可以自行分析，这里不详细介绍。

# 动态重定位执行过程

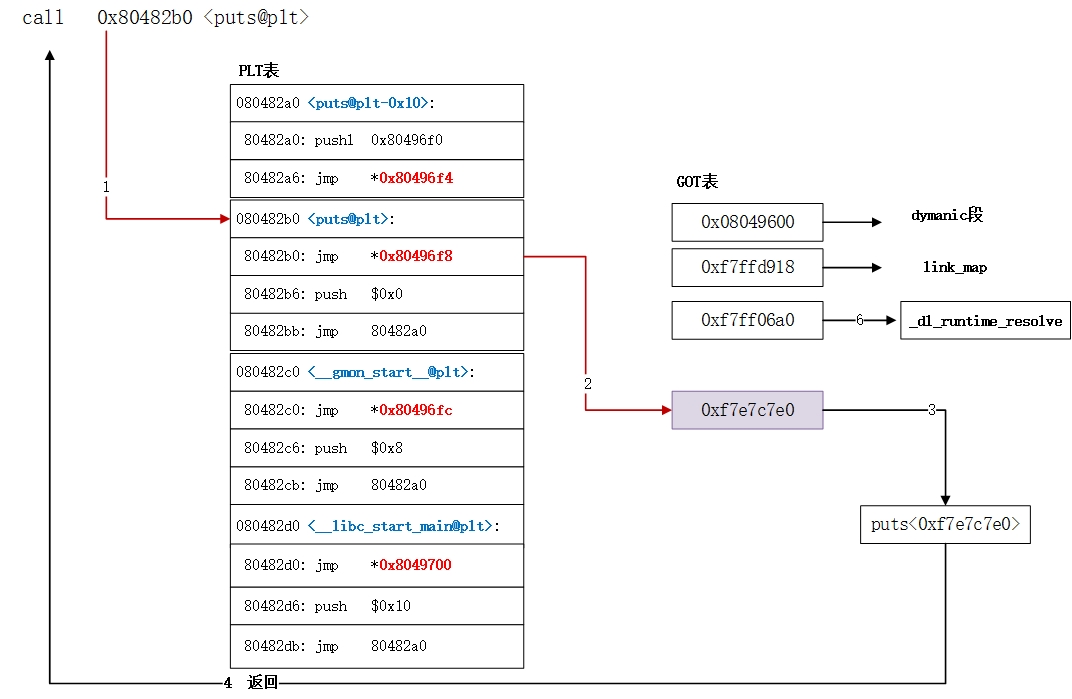
Linux 动态链接器提供动态重位功能，所有外部函数只有调用时才做重定位，实现延迟绑定功能。下面是以调用puts函数为例画出了整个动态重定位的执行过程：



在 \_dl\_runtime\_resolve函数内完成puts符号查找后，将该函数地址地址重定位到对应的GOT表项，并调用。

# 重定位之后的调用

GOT表项已完成重定位的情况下，PLT利用GOT表直接调用到真实的动态库函数，下面puts函数的调用过程：



# 总结

对于PLT和GOT的原理，一共分享了以下知识点：   
1. 为什么会有PLT和GOT表，它完成什么功能   
2. Linux如何通过 PLT和GOT表配合，完成延迟重定位功能   
3. PLT和GOT的结构是怎么样的，并且介绍每种场景下PLT的执行过程

关于PLT/GOT的基本知识写到这样就有清晰的认识了，但是Linux还有其它场景也会使用PLT/GOT，以后遇到时再展开讨论。

最后，本系列文章所有二进制分析，都是基于以下代码编译出来的可执行文件（32位）进行分析。

#include <stdio.h>

void print\_banner()

{

printf("Welcome to World of PLT and GOT\n");

}

int main(void)

{

print\_banner();

return 0;

}