作者：代兴　邮箱：503268771@qq.com　Vx 公众号：东方架构师

动态库的本质

动态库的本质是代理。

动态库的核心为 GOT、PLT。代理通过 GOT、PLT 实现。
GOT 代理符号，GOT 为 64 位整数变量(即指针)，指向一个符号，符号可以是变量、函数。
PLT 代理函数，实现延迟绑定，使用时才查找对应的函数，把函数地址更新到对应.got.plt 位置。
源码看不到 GOT、PLT。使用汇编、ELF、dump 查看 GOT、PLT。
访问符号时，如果符号有 GOT 代理，优先用 GOT 代理访问符号。

动态库，最终生成 so 文件，文件名称格式为 libxxx.so。
动态库使用 ELF 文件格式，包含 symbol、section、segment 等。

```
section .got         存放符号的代理
section .plt         存放延迟绑定的函数的代理
section .got.plt     存放延迟绑定的函数的地址
```

动态库被加载到内存映射区，实现多个进程共享。动态库在内存中，分为数据区、代码区，可以在进程的内存布局中查看。
数据区，每个进程私有。代码区，只读，多个进程共享。
程序加载时，查找依赖的动态库，重置 GOT 的值。程序运行时，延迟绑定 PLT，查找依赖的动态库函数，重置.got.plt 的值。

用 C 程序简单模拟动态库

说明：模拟的代码，做了简化，仅为示例。动态库的真实情况更加复杂。

编写代码： mock_so.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

// so 中的变量
int32_t so_param = 3333;

// 变量的代理。在 GOT
uint64_t so_param_GOT = (uint64_t)&so_param;

// 函数的代理。在 GOT
void so_func();
uint64_t so_func_GOT = (uint64_t)&so_func;
```

```
// so 中的函数
void so_func()
{
    // 用代理，取变量的地址
    uint64_t param_addr = so_param_GOT;
    // 用代理，取变量的值
    int32_t param_value = *((int32_t *)param_addr);
    // 用代理，取函数的地址
    uint64_t func_addr = so_func_GOT;

    printf("\n == so_func() == \n");
    printf("so_param   addr = %#llx  value = %d \n", param_addr, param_value);
    printf("so_func()  addr = %#llx  \n", func_addr);
    printf("\n");
}

//--------------------------------

// 重新定义一个变量。对应 so_param
int32_t main_param = 0;

// 函数的代理。对应 so_func 。这里简单示意。
uint64_t so_func_PLT = 0;

int main()
{
    // 动态重定位。指向 main 的变量
    so_param_GOT = (uint64_t)&main_param;
    // 动态重定位。指向 main 的变量
    so_func_GOT = (uint64_t)&so_func_PLT;

    //---------------------------------------
    // 初始化，把 so 的变量的值拷贝过来
    main_param = so_param;

    //---------------------------------------
    // 修改 main 的变量
    main_param = 6666;
    // 动态重定位。这里省略动态查找函数的过程
    so_func_PLT = (uint64_t)&so_func;

    //---------------------------------------
    // 用代理，调用 so 的函数。这里简单示意。
    void (*tmp_func)();
    tmp_func = (void (*)())so_func_PLT;
    tmp_func();

    //---------------------------------------
    // 查看 so 的变量
```

```
    uint64_t so_param_addr = (uint64_t)&so_param;
    int32_t so_param_value = so_param;
    // 查看 main 的变量
    uint64_t main_param_addr = (uint64_t)&main_param;
    int32_t main_param_value = main_param;

    printf("\n == main() == \n");
    printf("so_param     addr = %#llx  value = %d \n", so_param_addr, so_param_value);
    printf("main_param   addr = %#llx  value = %d \n", main_param_addr, main_param_value);
    printf("\n");

    //--------------------------------------
    // 函数的地址
    uint64_t so_func_addr = (uint64_t)&so_func;
    uint64_t so_func_PLT_addr = (uint64_t)&so_func_PLT;
    printf("so_func         addr = %#llx \n", so_func_addr);
    printf("so_func_PLT     addr = %#llx value = %#llx \n", so_func_PLT_addr, so_func_PLT);
    printf("\n");

    // GOT 的地址和值
    printf("so_param_GOT   addr = %#llx value = %#llx \n", &so_param_GOT, so_param_GOT);
    printf("so_func_GOT    addr = %#llx value = %#llx \n", &so_func_GOT, so_func_GOT);
    printf("\n");

    return 0;
}
```

编译代码：

```
gcc mock_so.c -o mock_so
```

运行代码：

```
[root@192 so]# ./mock_so

 == so_func() ==
so_param   addr = 0x601068  value = 6666
so_func()  addr = 0x601070



 == main() ==
so_param     addr = 0x601048  value = 3333
main_param   addr = 0x601068  value = 6666

so_func         addr = 0x4005bd
so_func_PLT     addr = 0x601070 value = 0x4005bd

so_param_GOT   addr = 0x601050 value = 0x601068
so_func_GOT    addr = 0x601058 value = 0x601070
```

分析结果：

so_param、main_param 是 2 个独立的变量，地址分别为 0x601048、0x601068，值分别为 3333、6666。
main_param 对应 so_param，main_param 用 so_param 初始化。main 程序可以修改 main_param ，不能修改 so_param 。

变量的代理关系为 so_param_GOT > main_param 。
函数的代理关系为 so_func_GOT > so_func_PLT > so_func 。
可以使用代理访问被代理的符号。

so_param_GOT 的值为 0x601068，等于 main_param 的地址。
so_func_GOT 的值为 0x601070，等于 so_func_PLT 的地址。
so_func_PLT 的值为 0x4005bd，等于 so_func 的地址。

分析代码。

分类	代码部分	说明

动态库的定义部分	// so 中的变量
	int32_t so_param = 3333;
	
	// 变量的代理。在 GOT
	uint64_t so_param_GOT =
	(uint64_t)&so_param;
	
	// 函数的代理。在 GOT
	void so_func();
	uint64_t so_func_GOT = (uint64_t)&so_func;
	
	// so 中的函数
	void so_func()
		动态库包含一个变量 so_param。
		动态库包含一个函数 so_func()。
		动态库生成一个代理 so_param_GOT，指向变量 so_param。
		动态库生成一个代理 so_func_GOT，指向函数 so_func()。

main 的定义部分	// 重新定义一个变量。对应 so_param
	int32_t main_param = 0;
	
	// 函数的代理。对应 so_func 。这里简单示意。
	uint64_t so_func_PLT = 0;
	
	int main()
		main 包含一个变量 main_param，对应变量 so_param。
		main 包含一个代理 so_func_PLT，对应函数 so_func()。

重置动态库的 GOT	// 动态重定位。指向 main 的变量
	so_param_GOT = (uint64_t)&main_param;
	// 动态重定位。指向 main 的变量
	so_func_GOT = (uint64_t)&so_func_PLT;
		代理 so_param_GOT，指向变量 so_param。
		代理 so_func_GOT，指向代理 so_func_PLT。

初始化变量	// 初始化，把 so 的变量的值拷贝过来
	main_param = so_param;
		main_param 用 so_param 初始化。

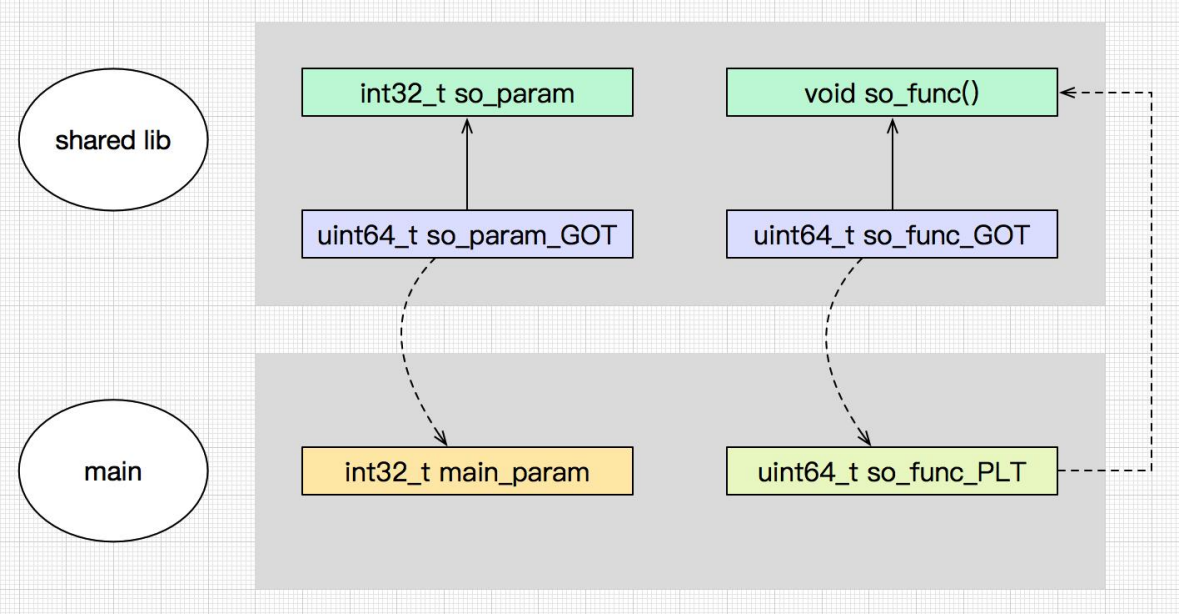
修改变量	// 修改 main 的变量
	main_param = 6666;
		main_param 修改值。

初始化 PLT	// 动态重定位。这里省略动态查找函数的过程
	so_func_PLT = (uint64_t)&so_func;
		PLT 有延迟绑定功能，这里没有实现，这里简单示例。
		代理 so_func_PLT，指向函数 so_func()。

main 使用 PLT	// 用代理，调用 so 的函数。这里简单示意。
	void (*tmp_func)();
	tmp_func = (void (*))so_func_PLT;
	tmp_func();
		使用代理 so_func_PLT 找到动态库的函数，然后调用函数。

动态库使用 GOT	void so_func()
	{
	// 用代理，取变量的地址
	uint64_t param_addr = so_param_GOT;
	// 用代理，取变量的值
	int32_t param_value = *((int32_t *)param_addr);
	// 用代理，取函数的地址
	uint64_t func_addr = so_func_GOT;
		使用代理 so_param_GOT 找到动态库的变量，然后读值。
		使用代理 so_func_GOT 找到动态库的函数。

代理关系
动态库包含变量 so_param、函数 so_func。
动态库包含代理 so_param_GOT、代理 so_func_GOT。
main 包含变量 main_param、代理 so_func_PLT。
代理 so_param_GOT，初始指向变量 so_param。后续重置代理，指向变量 main_param。
代理 so_func_GOT，初始指向函数 so_func。后续重置代理，指向代理 so_func_PLT。
代理 so_func_PLT，延迟绑定，指向函数 so_func。



分析复杂的动态库

编写代码： shared_bird.h

```
#ifndef _SHARED_BIRD_H_
#define _SHARED_BIRD_H_

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
```

```
// 占位。忽略。
extern int32_t shared_unused;

// main 使用此变量
extern int32_t shared_color;

// main 不使用此变量
extern int32_t shared_speed;

// 占位。忽略。
extern int32_t shared_unused2;

// main 使用此函数
extern void shared_dance();

// main 不使用此函数
extern void shared_smile();

// 小工具。输出符号的地址和值
void print_symbol(char *desc, uint64_t addr, int32_t value)
{
    printf("%-20s  addr = %#14llX", desc, addr);
    if (value != 0)
    {
        printf("    value = %#X ", value);
    }
    printf("\n");
}

#endif
```

编写代码： shared_bird.c

```
#include "shared_bird.h"

// 占位。忽略。
int32_t shared_unused = 0xA1A2A3A4;

// main 使用此变量
int32_t shared_color = 0xB1B2B3B4;

// 本地变量，外部不可见
static int32_t local_height = 0xC1C2C3C4;

// main 不使用此变量
int32_t shared_speed = 0x91929394;

// 占位。忽略。
int32_t shared_unused2 = 0x31323334;
```

```
// so 在内存的首地址。查看 elf 和 dump，找到 local_func 的偏移。
#define so_base_addr ((uint64_t) & local_func - 0x00097c)

// 本地函数。外部不可见。
static void local_func()
{
    printf("== shared  section  .GOT == \n");
    // 查看 elf 和 dump。用偏移找到 GOT 的地址。
    uint64_t got_addr = so_base_addr + 0x201fb0;
    // 代理存放 64 位地址。依次遍历。
    uint64_t *got_ptr = (uint64_t *)got_addr;
    int k;
    for (k = 0; k < 8; k++)
    {
        uint64_t *tmp_ptr = got_ptr + k;
        uint64_t tmp_value = *tmp_ptr;
        printf("GOT   addr = %p   value = %#1411X \n", tmp_ptr, tmp_value);
    }
    printf("\n");
}

// main 不使用此函数
void shared_smile()
{
    printf("== shared_smile() == \n");
}

// main 使用此函数
void shared_dance()
{
    // 取地址
    uint64_t color_addr = (uint64_t)&shared_color;
    uint64_t height_addr = (uint64_t)&local_height;
    uint64_t speed_addr = (uint64_t)&shared_speed;
    int tmp1 = 111111;

    // 取值
    int32_t color_value = shared_color;
    int32_t height_value = local_height;
    int32_t speed_value = shared_speed;
    int tmp2 = 222222;

    // 查看初始的变量
    uint64_t color_addr_init = height_addr - 4;
    int32_t color_value_init = *((int32_t *)color_addr_init);

    // 共享函数的地址
    uint64_t shared_dance_addr = (uint64_t)&shared_dance;
```

```
    uint64_t shared_smile_addr = (uint64_t)&shared_smile;

    // 本地函数的地址
    uint64_t local_func_addr = (uint64_t)&local_func;

    // 查看 elf 和 dump，找到 shared_dance 的偏移。
    uint64_t shared_dance_addr_init = so_base_addr + 0x000a19;

    // 外部 so 的函数
    // main 使用了函数 printf
    uint64_t func_printf_addr = (uint64_t)&printf;
    // main 没有使用函数 execvp
    uint64_t func_execvp_addr = (uint64_t)&execvp;
    int tmp3 = 333333;

    printf("== shared_dance() == \n");
    print_symbol("shared_color", color_addr, color_value);
    print_symbol("shared_color_init", color_addr_init, color_value_init);
    print_symbol("local_height", height_addr, height_value);
    print_symbol("shared_speed", speed_addr, speed_value);

    print_symbol("local_func()", local_func_addr, 0);
    print_symbol("shared_smile()", shared_smile_addr, 0);
    print_symbol("shared_dance()", shared_dance_addr, 0);
    print_symbol("shared_dance()_real", shared_dance_addr_init, 0);

    print_symbol("printf()", func_printf_addr, 0);
    print_symbol("execvp()", func_execvp_addr, 0);
    printf("\n");

    // 调用本地函数
    local_func();
}
```

编写代码： main.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include "shared_bird.h"

// 变量
int32_t main_month = 0xF1F2F3F4;

void main_print()
{
    // 占位。忽略。
    uint64_t unused_addr = (uint64_t)&shared_unused;
    uint64_t unused2_addr = (uint64_t)&shared_unused2;
```

```
    int tmp1 = 0x111111;

    // 取地址
    uint64_t color_addr = (uint64_t)&shared_color;
    uint64_t month_addr = (uint64_t)&main_month;
    uint64_t shared_func_addr = (uint64_t)&shared_dance;
    // 外部 so 的函数 printf
    uint64_t func_printf_addr = (uint64_t)&printf;
    int tmp2 = 0x222222;

    // 取值
    int32_t color_value = shared_color;
    int32_t month_value = main_month;
    int tmp3 = 0x333333;

    printf("== main_print() == \n");
    print_symbol("shared_color", color_addr, color_value);
    print_symbol("main_month", month_addr, month_value);
    print_symbol("shared_dance()", shared_func_addr, 0);
    print_symbol("printf()", func_printf_addr, 0);
    printf("\n");
}

int main()
{
    // 修改动态库的变量
    shared_color = 0x71727374;

    // 查看 ELF 文件，找到 shared_dance 的.got.plt
    uint64_t shared_dance_GOT_PLT_addr = 0x000000601030;
    uint64_t *shared_dance_GOT_PLT_ptr = (uint64_t *)shared_dance_GOT_PLT_addr;
    // 调用 shared_dance() 之前，查看.got.plt 的值
    uint64_t shared_dance_GOT_PLT_value = *shared_dance_GOT_PLT_ptr;

    // 调用动态库的函数
    shared_dance();

    // 调用 shared_dance() 之后，查看.got.plt 的值
    printf("== main  section  .GOT.PLT == \n");
    uint64_t shared_dance_GOT_PLT_value2 = *shared_dance_GOT_PLT_ptr;
    printf("shared_dance_GOT_PLT  before call = %#1411X  \n", shared_dance_GOT_PLT_value);
    printf("shared_dance_GOT_PLT  after  call = %#1411X  \n", shared_dance_GOT_PLT_value2);
    printf("\n");

    // 调用 main 的函数
    main_print();

    // 休眠进程。方便查看内存布局。
    sleep(900000);
```

```
    return 0;
}
```

编译代码：

```
# 生成动态库
gcc shared_bird.c -fPIC -shared -o libshared_bird.so
gcc shared_bird.c -fPIC -shared -S -o libshared_bird.so.s
# 动态库，放入默认库目录，刷新
cp libshared_bird.so  /usr/lib  &&  ldconfig

# 生成 main 程序
gcc main.c -L.  -lshared_bird -o main
gcc main.c -L.  -lshared_bird -S -o main.s

# 生成 ELF 文件、dump 文件
objdump -D main  > main.dump.txt
readelf -a main  > main.elf.txt
objdump -D libshared_bird.so  > libshared_bird.so.dump.txt
readelf -a libshared_bird.so  > libshared_bird.so.elf.txt
```

运行代码：

```
[root@192 so]# ./main
== shared_dance() ==
shared_color          addr =       0X60105C    value = 0X71727374
shared_color_init     addr = 0X7F10BB5DA04C    value = 0XB1B2B3B4
local_height          addr = 0X7F10BB5DA050    value = 0XC1C2C3C4
shared_speed          addr = 0X7F10BB5DA054    value = 0X91929394
local_func()          addr = 0X7F10BB3D897C
shared_smile()        addr = 0X7F10BB3D8A07
shared_dance()        addr =       0X400700
shared_dance()_real   addr = 0X7F10BB3D8A19
printf()              addr =       0X4006F0
execvp()              addr = 0X7F10BB0D0260

== shared  section  .GOT ==
GOT   addr = 0x7f10bb5d9fb0   value =       0X4006F0
GOT   addr = 0x7f10bb5d9fb8   value =       0X400700
GOT   addr = 0x7f10bb5d9fc0   value =              0
GOT   addr = 0x7f10bb5d9fc8   value = 0X7F10BB3D8A07
GOT   addr = 0x7f10bb5d9fd0   value =       0X60105C
GOT   addr = 0x7f10bb5d9fd8   value = 0X7F10BB5DA054
GOT   addr = 0x7f10bb5d9fe0   value =              0
GOT   addr = 0x7f10bb5d9fe8   value = 0X7F10BB0D0260

== main  section  .GOT.PLT ==
shared_dance_GOT_PLT  before call =       0X400706
shared_dance_GOT_PLT  after  call = 0X7F10BB3D8A19

== main_print() ==
```

```
shared_color          addr =       0X60105C    value = 0X71727374
main_month            addr =       0X601054    value = 0XF1F2F3F4
shared_dance()        addr =       0X400700
printf()              addr =       0X4006F0
```

进程的内存布局：

```
[root@192 so]# ps aux | grep /main
root      57757  0.0  0.0   6276   364 pts/3    S+   15:18   0:00 ./main
root      58116  0.0  0.0 112812   980 pts/4    S+   15:24   0:00 grep --color=auto /main
[root@192 so]# cat /proc/57757/maps
00400000-00401000               r-xp              00000000              08:03              741917
/root/code/x86-asm/common2/so/main
00600000-00601000               r--p              00000000              08:03              741917
/root/code/x86-asm/common2/so/main
00601000-00602000               rw-p              00001000              08:03              741917
/root/code/x86-asm/common2/so/main
7f10bb00a000-7f10bb1ce000 r-xp 00000000 08:03 15928                    /usr/lib64/libc-2.17.so
7f10bb1ce000-7f10bb3cd000 ---p 001c4000 08:03 15928                    /usr/lib64/libc-2.17.so
7f10bb3cd000-7f10bb3d1000 r--p 001c3000 08:03 15928                    /usr/lib64/libc-2.17.so
7f10bb3d1000-7f10bb3d3000 rw-p 001c7000 08:03 15928                    /usr/lib64/libc-2.17.so
7f10bb3d3000-7f10bb3d8000 rw-p 00000000 00:00 0
7f10bb3d8000-7f10bb3d9000 r-xp 00000000 08:03 17693552                 /usr/lib/libshared_bird.so
7f10bb3d9000-7f10bb5d9000 ---p 00001000 08:03 17693552                 /usr/lib/libshared_bird.so
7f10bb5d9000-7f10bb5da000 r--p 00001000 08:03 17693552                 /usr/lib/libshared_bird.so
7f10bb5da000-7f10bb5db000 rw-p 00002000 08:03 17693552                 /usr/lib/libshared_bird.so
7f10bb5db000-7f10bb5fd000 r-xp 00000000 08:03 611075                   /usr/lib64/ld-2.17.so
7f10bb7f1000-7f10bb7f4000 rw-p 00000000 00:00 0
7f10bb7fa000-7f10bb7fc000 rw-p 00000000 00:00 0
7f10bb7fc000-7f10bb7fd000 r--p 00021000 08:03 611075                   /usr/lib64/ld-2.17.so
7f10bb7fd000-7f10bb7fe000 rw-p 00022000 08:03 611075                   /usr/lib64/ld-2.17.so
7f10bb7fe000-7f10bb7ff000 rw-p 00000000 00:00 0
7ffe383fc000-7ffe3841d000 rw-p 00000000 00:00 0                        [stack]
7ffe38443000-7ffe38445000 r-xp 00000000 00:00 0                        [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                [vsyscall]
```

分析结果：

查看内存布局，重点关注数据区、代码区。

main程序的代码区	00400000-00401000 r-xp 00000000 08:03 741917 /root/code/x86-asm/common2/so/main
main程序的只读数据区	00600000-00601000 r--p 00000000 08:03 741917 /root/code/x86-asm/common2/so/main
main程序的可读可写数据区	00601000-00602000 rw-p 00001000 08:03 741917 /root/code/x86-asm/common2/so/main
动态库的代码区	7f10bb3d8000-7f10bb3d9000 r-xp 00000000 08:03 17693552 /usr/lib/libshared_bird.so

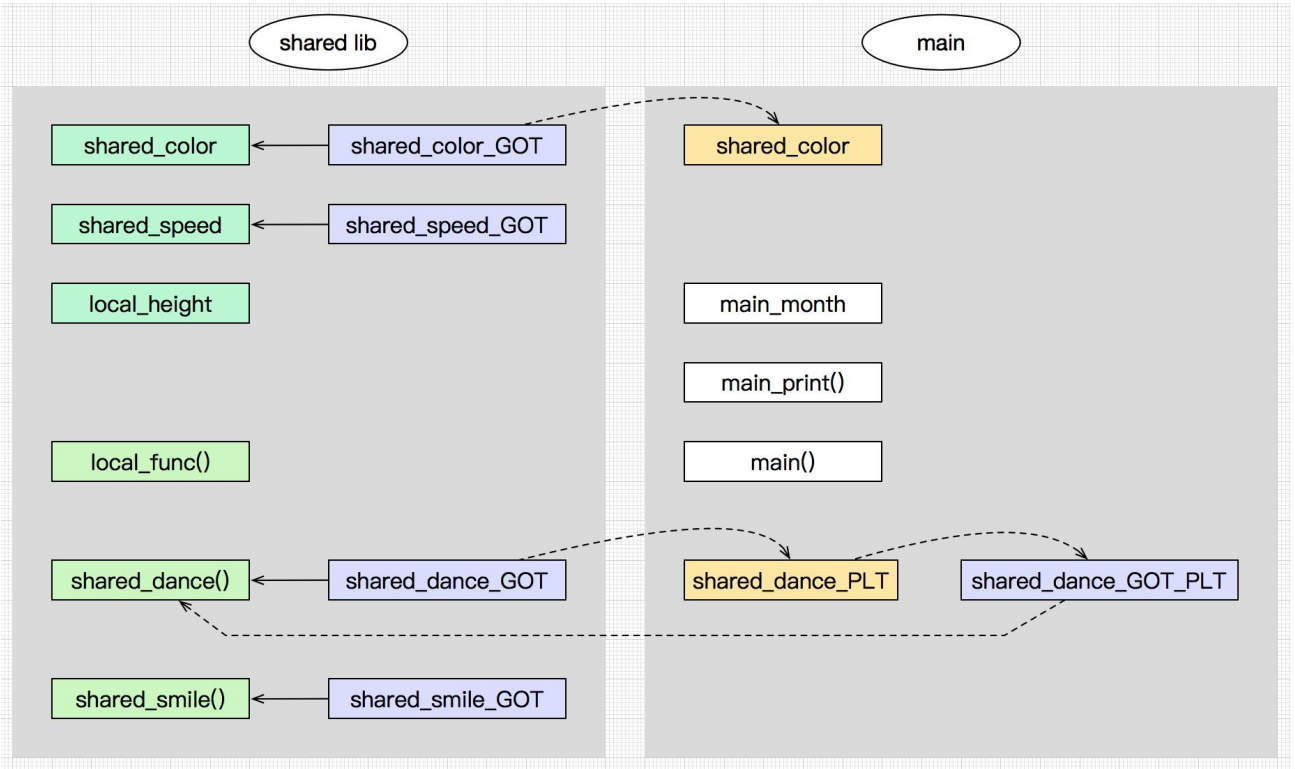
动态库的只读数据区	7f10bb5d9000-7f10bb5da000 r--p 00001000 08:03 17693552 /usr/lib/libshared_bird.so
动态库的可读可写数据区	7f10bb5da000-7f10bb5db000 rw-p 00002000 08:03 17693552 /usr/lib/libshared_bird.so

符号的代理关系图

为了简化说明，省略函数的入参、返回值等，重点查看动态库的符号。

动态库有共享的变量 shared_color，生成对应的代理 shared_color_GOT。
动态库有共享的变量 shared_speed，生成对应的代理 shared_speed_GOT。
动态库有本地的变量 local_height，不生成对应的代理。
动态库有本地的函数 local_func()，不生成对应的代理。
动态库有共享的函数 shared_dance()，生成对应的代理 shared_dance_GOT。
动态库有共享的函数 shared_smile()，生成对应的代理 shared_smile_GOT。

main 依赖动态库的变量 shared_color，生成一个隐式变量 shared_color，并且把代理 shared_color_GOT 指向隐式变量 shared_color。
main 不依赖动态库的变量 shared_speed，不修改代理 shared_speed_GOT。
main 依赖动态库的函数 shared_dance()，生成 2 个隐式符号 shared_dance_PLT、shared_dance_GOT_PLT，并且用隐式符号 shared_dance_PLT 代理函数 shared_dance(),把代理 shared_dance_GOT 指向隐式符号 shared_dance_PLT。
main 不依赖动态库的函数 shared_smile()，不生成对应的代理 PLT。
隐式符号 shared_dance_PLT，延迟绑定函数，最后指向动态库的函数 shared_dance() 。



动态库的符号和代理

查看文件 libshared_bird.so.elf.txt、libshared_bird.so.dump.txt，这里关注需要的符号。

查看符号表

```
Symbol table '.symtab' contains 68 entries:
    36: 0000000000202050     4 OBJECT  LOCAL  DEFAULT   23 local_height
    37: 000000000000097c   139 FUNC    LOCAL  DEFAULT   11 local_func
    53: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@@GLIBC_2.2.5
    54: 0000000000000a19   454 FUNC    GLOBAL DEFAULT   11 shared_dance
    58: 0000000000000a07    18 FUNC    GLOBAL DEFAULT   11 shared_smile
    60: 000000000020204c     4 OBJECT  GLOBAL DEFAULT   23 shared_color
    61: 0000000000202054     4 OBJECT  GLOBAL DEFAULT   23 shared_speed
    64: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND execvp@@GLIBC_2.2.5
```

查看 .got

```
Relocation section '.rela.dyn' at offset 0x5c8 contains 14 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000201fb0  000400000006 R_X86_64_GLOB_DAT 0000000000000000 printf@GLIBC_2.2.5 + 0
000000201fb8  000d00000006 R_X86_64_GLOB_DAT 0000000000000a19 shared_dance + 0
000000201fc8  000e00000006 R_X86_64_GLOB_DAT 0000000000000a07 shared_smile + 0
000000201fd0  001500000006 R_X86_64_GLOB_DAT 000000000020204c shared_color + 0
000000201fd8  000c00000006 R_X86_64_GLOB_DAT 0000000000202054 shared_speed + 0
000000201fe8  000700000006 R_X86_64_GLOB_DAT 0000000000000000 execvp@GLIBC_2.2.5 + 0
```

查看 .plt

```
00000000000007d0 <.plt>:
 7d0:   ff 35 32 18 20 00       pushq  0x201832(%rip)        # 202008 <_GLOBAL_OFFSET_TABLE_+0x8>
 7d6:   ff 25 34 18 20 00       jmpq   *0x201834(%rip)        # 202010 <_GLOBAL_OFFSET_TABLE_+0x10>
 7dc:   0f 1f 40 00             nopl   0x0(%rax)
00000000000007e0 <print_symbol@plt>:
 7e0:   ff 25 32 18 20 00       jmpq   *0x201832(%rip)        # 202018 <print_symbol@@Base+0x2016f3>
 7e6:   68 00 00 00 00          pushq  $0x0
 7eb:   e9 e0 ff ff ff          jmpq   7d0 <.plt>
0000000000000800 <puts@plt>:
 800:   ff 25 22 18 20 00       jmpq   *0x201822(%rip)        # 202028 <puts@GLIBC_2.2.5>
 806:   68 02 00 00 00          pushq  $0x2
 80b:   e9 c0 ff ff ff          jmpq   7d0 <.plt>
0000000000000810 <printf@plt>:
 810:   ff 25 1a 18 20 00       jmpq   *0x20181a(%rip)        # 202030 <printf@GLIBC_2.2.5>
 816:   68 03 00 00 00          pushq  $0x3
 81b:   e9 b0 ff ff ff          jmpq   7d0 <.plt>
```

查看 .got.plt

```
Relocation section '.rela.plt' at offset 0x718 contains 6 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000202018  001200000007 R_X86_64_JUMP_SLO 0000000000000925 print_symbol + 0
```

```
000000202028  000300000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
000000202030  000400000007 R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0
```

符号名称	符号类型	说明	原始符号的地址	GOT 代理的地址
shared_color	OBJECT	共享变量	000000000020204c	000000201fd0
shared_speed	OBJECT	共享变量	0000000000202054	000000201fd8
local_height	OBJECT	本地变量	0000000000202050	本地变量没有 GOT 代理
shared_dance	FUNC	共享函数	0000000000000a19	000000201fb8
shared_smile	FUNC	共享函数	0000000000000a07	000000201fc8
local_func	FUNC	本地函数	000000000000097c	本地函数没有 GOT 代理
printf	FUNC	依赖其他动态库	延迟绑定。	000000201fb0
execvp	FUNC	依赖其他动态库	延迟绑定。	000000201fe8

动态库的函数 shared_dance()

查看文件 shared_bird.c、libshared_bird.so.s、libshared_bird.so.dump.txt。

源码 shared_bird.c	汇编文件 libshared_bird.so.s	dump 文件 libshared_bird.so.dump.txt
void shared_dance()	shared_dance:	0000000000000a19 <shared_dance>:
uint64_t color_addr = (uint64_t)&shared_color;	movq shared_color@GOTPCREL(%rip), %rax movq %rax, -8(%rbp)	mov 0x2015a8(%rip),%rax # 201fd0 <shared_color@@Base-0x7c> mov %rax,-0x8(%rbp)
uint64_t height_addr = (uint64_t)&local_height;	leaq local_height(%rip), %rax movq %rax, -16(%rbp)	lea 0x20161d(%rip),%rax # 202050 <local_height> mov %rax,-0x10(%rbp)
uint64_t speed_addr = (uint64_t)&shared_speed;	movq shared_speed@GOTPCREL(%rip), %rax movq %rax, -24(%rbp)	mov 0x20159a(%rip),%rax # 201fd8 <shared_speed@@Base-0x7c> mov %rax,-0x18(%rbp)

int32_t color_value = shared_color;	movq shared_color@GOTPCREL(%rip), %rax
	movl (%rax), %eax
	movl %eax, -32(%rbp)
		mov 0x201580(%rip),%rax # 201fd0 <shared_color@@Base-0x7c>
		mov (%rax),%eax
		mov %eax,-0x20(%rbp)

int32_t height_value = local_height;	movl local_height(%rip), %eax movl %eax, -36(%rbp)	mov 0x2015f5(%rip),%eax # 202050 <local_height> mov %eax,-0x24(%rbp)

int32_t speed_value = shared_speed;	movq shared_speed@GOTPCREL(%rip), %rax
	movl (%rax), %eax
	movl %eax, -40(%rbp)
		mov 0x201573(%rip),%rax # 201fd8 <shared_speed@@Base-0x7c>
		mov (%rax),%eax
		mov %eax,-0x28(%rbp)

uint64_t shared_dance_addr =	movq shared_dance@GOTPCREL(%rip), %ra	mov 0x20152b(%rip),%rax # 201fb8

(uint64_t)&shared_dance;	x movq %rax, -72(%rbp)	<shared_dance@@Base+0x20159f> mov %rax,-0x48(%rbp)
uint64_t shared_smile_addr = (uint64_t)&shared_smile;	movq shared_smile@GOTPCREL(%rip), %rax movq %rax, -80(%rbp)	mov 0x201530(%rip),%rax # 201fc8 <shared_smile@@Base+0x2015c1> mov %rax,-0x50(%rbp)
uint64_t local_func_addr = (uint64_t)&local_func;	leaq local_func(%rip), %rax movq %rax, -88(%rbp)	lea -0x127(%rip),%rax # 97c <local_func> mov %rax,-0x58(%rbp)
printf("== shared_dance() == \n");	call puts@PLT	callq 800 <puts@plt>
print_symbol("execvp()", func_execvp_addr, 0);	call print_symbol@PLT	callq 7e0 <print_symbol@plt>
local_func();	call local_func	callq 97c <local_func>

结合 ELF 文件、汇编代码，总结动态库的符号的访问方式：
1、共享变量，使用 GOT 代理取地址、取值。
shared_color 是动态库的共享变量。
取地址，使用 `movq shared_color@GOTPCREL(%rip), %rax` ，shared_color@GOTPCREL 是 GOT 代理，其值为 shared_color 的地址。
取值，先使用 `movq shared_color@GOTPCREL(%rip), %rax` 取地址写入 rax，再使用 `movl (%rax), %eax` 取值写入 eax 。
2、本地变量，使用普通模式取地址、取值。
local_height 是动态库的本地变量。
取地址，使用 `leaq local_height(%rip), %rax` ，直接取 local_height 的 rip 相对地址。
取值，使用 `movl local_height(%rip), %eax` ，直接取 local_height 的值。
3、共享函数，使用 GOT 代理取地址。
shared_dance()是共享函数。取地址，使用 `movq shared_dance@GOTPCREL(%rip), %rax` ，其中 shared_dance@GOTPCREL 是 GOT 代理。
shared_smile()是共享函数。取地址，使用 `movq shared_smile@GOTPCREL(%rip), %rax` ，其中 shared_smile@GOTPCREL 是 GOT 代理。
4、本地函数，使用普通模式取地址。
local_func()是本地函数。取地址，使用 `leaq local_func(%rip), %rax` ，直接取 local_func 的 rip 相对地址。
5、共享函数，使用 PLT 代理调用函数。
printf()是共享函数。调用函数，使用 `call puts@PLT` ，其中 puts@PLT 是 puts 的 PLT 代理。
6、本地函数，使用普通模式调用函数。
local_func()是本地函数。调用函数，使用 `call local_func` 。

动态库的 GOT 代理

进程启动多次，动态库在进程的内存布局可能发生变化。查看动态库的内存地址，需要使用相对偏移。

```
// so 在内存的首地址。查看 elf 和 dump，找到 local_func 的偏移。
#define so_base_addr ((uint64_t) & local_func - 0x00097c)
```

local_func()是本地函数，其地址在动态库的内存范围内。

查看 libshared_bird.so.elf.txt，找到符号 local_func 的描述，地址为 000000000000097c，含义是在动态库的

内存偏移。

```
    37: 000000000000097c   139 FUNC    LOCAL  DEFAULT   11 local_func
```

符号 local_func 的当前地址，减去 local_func 的偏移，等于动态库的首地址。

查看 libshared_bird.so.elf.txt，printf 的 GOT 的偏移为 000000201fb0。动态库的首地址加上 GOT 的偏移，等于 GOT 的地址。

```
    // 查看 elf 和 dump。用偏移找到 GOT 的地址。
    uint64_t got_addr = so_base_addr + 0x201fb0;
```

查看 libshared_bird.so.elf.txt，找到 GOT 列表。

```
Relocation section '.rela.dyn' at offset 0x5c8 contains 14 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000201fb0  000400000006 R_X86_64_GLOB_DAT 0000000000000000 printf@GLIBC_2.2.5 + 0
000000201fb8  000d00000006 R_X86_64_GLOB_DAT 0000000000000a19 shared_dance + 0
000000201fc0  000500000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
000000201fc8  000e00000006 R_X86_64_GLOB_DAT 0000000000000a07 shared_smile + 0
000000201fd0  001500000006 R_X86_64_GLOB_DAT 000000000020204c shared_color + 0
000000201fd8  000c00000006 R_X86_64_GLOB_DAT 0000000000202054 shared_speed + 0
000000201fe0  000600000006 R_X86_64_GLOB_DAT 0000000000000000 _Jv_RegisterClasses + 0
000000201fe8  000700000006 R_X86_64_GLOB_DAT 0000000000000000 execvp@GLIBC_2.2.5 + 0
```

许多 GOT 在一起，使用指针依次遍历，输出 GOT 的地址和值。

```
    // 代理存放 64 位地址。依次遍历。
    uint64_t *got_ptr = (uint64_t *)got_addr;
```

查看输出的结果。GOT 的地址，都在动态库的数据区。GOT 的值，有些在低地址因为被 main 加载时重置了。

```
== shared  section  .GOT ==
GOT   addr = 0x7f10bb5d9fb0   value =       0X4006F0
GOT   addr = 0x7f10bb5d9fb8   value =       0X400700
GOT   addr = 0x7f10bb5d9fc0   value =              0
GOT   addr = 0x7f10bb5d9fc8   value = 0X7F10BB3D8A07
GOT   addr = 0x7f10bb5d9fd0   value =       0X60105C
GOT   addr = 0x7f10bb5d9fd8   value = 0X7F10BB5DA054
GOT   addr = 0x7f10bb5d9fe0   value =              0
GOT   addr = 0x7f10bb5d9fe8   value = 0X7F10BB0D0260
```

问题：shared_color、shared_speed 都是共享变量，为什么打印出来的地址差别很大？

```
shared_color          addr =       0X60105C    value = 0X71727374
shared_speed          addr = 0X7F10BB5DA054    value = 0X91929394
```

共享变量，使用 GOT 代理取地址。

main 使用了 shared_color，所以在 main 的数据区新建一个隐式符号对应 shared_color。然后把动态库的 shared_color 的 GOT 代理指向隐式符号。

main 没有使用 shared_speed，所以没有修改动态库的 shared_speed 的 GOT 代理。

查看打印的动态库的 GOT 信息，shared_color 的 GOT 的值为 0X60105C，shared_speed 的 GOT 的值为 0X7F10BB5DA054。正好等于符号 shared_color、shared_speed 的地址。

```
GOT   addr = 0x7f10bb5d9fd0   value =       0X60105C
GOT   addr = 0x7f10bb5d9fd8   value = 0X7F10BB5DA054
```

问题：shared_smile()、shared_dance()都是共享函数，为什么打印出来的地址差别很大？

```
shared_smile()        addr = 0X7F10BB3D8A07
shared_dance()        addr =       0X400700
```

共享函数，使用 GOT 代理取地址。
main 使用了 shared_dance()，所以在 main 的代码区新建一个 PLT 代理。然后把动态库的 shared_dance()的 GOT 代理指向 PLT 代理。
main 没有使用 shared_smile()，所以没有修改动态库的 shared_smile()的 GOT 代理。
查看打印的动态库的 GOT 信息，shared_dance()的 GOT 的值为 0X400700，shared_smile()的 GOT 的值为 0X7F10BB3D8A07。正好等于符号 shared_dance()、shared_smile()的地址。

```
GOT   addr = 0x7f10bb5d9fb8   value =       0X400700
GOT   addr = 0x7f10bb5d9fc8   value = 0X7F10BB3D8A07
```

main 文件的符号和代理

查看文件 main.elf.txt、main.dump.txt，这里关注需要的符号。

查看符号表

```
Symbol table '.symtab' contains 73 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
    47: 000000000040082d    83 FUNC    GLOBAL DEFAULT   13 print_symbol
    50: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@@GLIBC_2.2.5
    52: 0000000000601054     4 OBJECT  GLOBAL DEFAULT   24 main_month
    54: 00000000004006f0     0 FUNC    GLOBAL DEFAULT  UND printf@@GLIBC_2.2.5
    55: 0000000000400700     0 FUNC    GLOBAL DEFAULT  UND shared_dance
    56: 0000000000601058     4 OBJECT  GLOBAL DEFAULT   25 shared_unused2
    66: 0000000000400949   157 FUNC    GLOBAL DEFAULT   13 main
    67: 000000000060105c     4 OBJECT  GLOBAL DEFAULT   25 shared_color
    68: 0000000000601060     4 OBJECT  GLOBAL DEFAULT   25 shared_unused
    70: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND sleep@@GLIBC_2.2.5
    72: 0000000000400880   201 FUNC    GLOBAL DEFAULT   13 main_print
```

查看符号复制

```
Relocation section '.rela.dyn' at offset 0x590 contains 4 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000601058  000f00000005 R_X86_64_COPY     0000000000601058 shared_unused2 + 0
00000060105c  001000000005 R_X86_64_COPY     000000000060105c shared_color + 0
000000601060  000b00000005 R_X86_64_COPY     0000000000601060 shared_unused + 0
```

查看 .plt

```
00000000004006c0 <.plt>:
  4006c0:   ff 35 42 09 20 00       pushq       0x200942(%rip)                          #    601008
<_GLOBAL_OFFSET_TABLE_+0x8>
  4006c6:   ff 25 44 09 20 00       jmpq          *0x200944(%rip)                       #    601010
<_GLOBAL_OFFSET_TABLE_+0x10>
  4006cc:   0f 1f 40 00             nopl   0x0(%rax)
00000000004006e0 <puts@plt>:
  4006e0:   ff 25 3a 09 20 00       jmpq   *0x20093a(%rip)        # 601020 <puts@GLIBC_2.2.5>
  4006e6:   68 01 00 00 00          pushq  $0x1
  4006eb:   e9 d0 ff ff ff          jmpq   4006c0 <.plt>
```

```
00000000004006f0 <printf@plt>:
  4006f0:   ff 25 32 09 20 00       jmpq   *0x200932(%rip)        # 601028 <printf@GLIBC_2.2.5>
  4006f6:   68 02 00 00 00          pushq  $0x2
  4006fb:   e9 c0 ff ff ff          jmpq   4006c0 <.plt>
0000000000400700 <shared_dance@plt>:
  400700:   ff 25 2a 09 20 00       jmpq   *0x20092a(%rip)        # 601030 <shared_dance>
  400706:   68 03 00 00 00          pushq  $0x3
  40070b:   e9 b0 ff ff ff          jmpq   4006c0 <.plt>
```

查看 .got.plt

```
Relocation section '.rela.plt' at offset 0x5f0 contains 7 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000601020  000200000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
000000601028  000800000007 R_X86_64_JUMP_SLO 00000000004006f0 printf@GLIBC_2.2.5 + 0
000000601030  000900000007 R_X86_64_JUMP_SLO 0000000000400700 shared_dance + 0
```

符号名称	符号类型	说明	符号的地址
shared_color	OBJECT	复制动态库的符号	000000000060105c
shared_dance@plt	FUNC	shared_dance 的 PLT 代理	0000000000400700
printf@plt	FUNC	printf 的 PLT 代理	00000000004006f0
main_month	OBJECT	本地变量	0000000000601054
main_print	FUNC	本地函数	0000000000400880
main	FUNC	本地函数	0000000000400949

main 文件的函数 main_print()

查看文件 main.c、main.s、main.dump.txt。

源码 main.c	汇编文件 main.s	dump 文件 main.dump.txt
void main_print()	main_print:	0000000000400880 <main_print>:
uint64_t color_addr = (uint64_t)&shared_color;	movq $shared_color, -32(%rbp)	movq $0x60105c,-0x20(%rbp)
uint64_t month_addr = (uint64_t)&main_month;	movq $main_month, -40(%rbp)	movq $0x601054,-0x28(%rbp)
uint64_t shared_func_addr = (uint64_t)&shared_dance;	movq $shared_dance, -48(%rbp)	movq $0x400700,-0x30(%rbp)
int32_t color_value = shared_color;	movl shared_color(%rip), %eax movl %eax, -64(%rbp)	mov 0x200790(%rip),%eax # 60105c <shared_color> mov %eax,-0x40(%rbp)
int32_t month_value = main_month;	movl main_month(%rip), %eax movl %eax, -68(%rbp)	mov 0x20077f(%rip),%eax # 601054 <main_month> mov %eax,-0x44(%rbp)

结合 ELF 文件、汇编代码，总结主程序中符号的访问方式：

1、主程序依赖动态库的共享变量，在主程序的数据区创建一个变量副本，复制共享变量的值。

2、主程序依赖动态库的共享函数，在主程序的代码区创建一个 PLT 代理，在主程序的数据区创建一个.got.plt，

实现延迟绑定。
3、共享变量、本地变量，取地址的方式相同。
shared_color 是动态库的共享变量。取地址，使用 `movq $shared_color, -32(%rbp)` 。直接使用数据区的副本。
main_month 是本地变量。取地址，使用 `movq $main_month, -40(%rbp)` 。
4、共享变量、本地变量，取值的方式相同。
shared_color 是动态库的共享变量。取值，使用 `movl shared_color(%rip), %eax` 。直接使用数据区的副本。
main_month 是本地变量。取值，使用 `movl main_month(%rip), %eax` 。
5、共享函数，取地址使用 PLT 代理。
shared_dance()是动态库的共享函数。
取地址，使用 `movq $shared_dance, -48(%rbp)` ，等于 `movq $0x400700,-0x30(%rbp)` 。
地址 0x400700 表示 PLT 代理 `0000000000400700 <shared_dance@plt>` 。

动态库的 PLT 代理

main 使用动态库的函数 shared_dance()。
main 创建一个 PLT 代理 shared_dance@plt，在代码区。plt 的第一行跳转到.got.plt 601030。

```
0000000000400700 <shared_dance@plt>:
  400700:   ff 25 2a 09 20 00       jmpq   *0x20092a(%rip)        # 601030 <shared_dance>
  400706:   68 03 00 00 00          pushq  $0x3
  40070b:   e9 b0 ff ff ff          jmpq   4006c0 <.plt>
```

main 创建一个.got.plt，在数据区，存放函数跳转地址。

```
000000601030  000900000007 R_X86_64_JUMP_SLO 0000000000400700 shared_dance + 0
```

在调用函数 shared_dance()的前后，打印.got.plt 的值，值发生变化。

```
shared_dance_GOT_PLT  before call =       0X400706
shared_dance_GOT_PLT  after  call = 0X7F10BB3D8A19
```

调用函数之前，.got.plt 的值为 0X400706，等于 shared_dance@plt 的第二行的地址。
调用函数的过程中，实现延迟绑定。把函数的编号 0x3 入栈，然后执行查找函数，把函数的真实地址写到.got.plt。
调用函数之后，.got.plt 的值为 0X7F10BB3D8A19 ，等于动态库的函数的真实地址。

```
shared_dance()_real   addr = 0X7F10BB3D8A19
```

后续调用函数 shared_dance()，先跳转到 400700，后跳转到 0X7F10BB3D8A19。

问题：如何找到动态库的函数 shared_dance()的真实地址？
找到动态库的起始地址。使用本地函数的地址减去偏移。

```
// so 在内存的首地址。查看 elf 和 dump，找到 local_func 的偏移。
#define so_base_addr ((uint64_t) & local_func - 0x00097c)
```

在 ELF 中找到函数 shared_dance()的偏移。

```
Symbol table '.symtab' contains 68 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
    54: 0000000000000a19   454 FUNC    GLOBAL DEFAULT   11 shared_dance
```

计算：动态库的起始地址 + 函数的偏移 = 函数的真实地址

```
    // 查看 elf 和 dump，找到 shared_dance 的偏移。
    uint64_t shared_dance_addr_init = so_base_addr + 0x000a19;
```