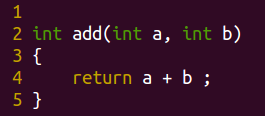
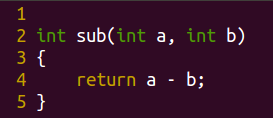
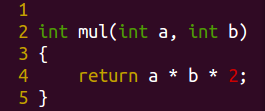
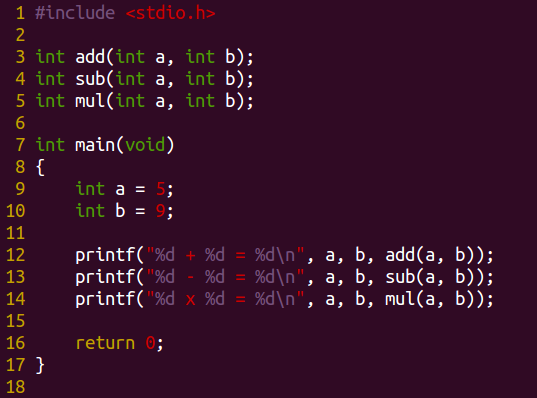
# Linux系统动态库加载过程分析

在Linux系统开发中，我们频繁的使用动态库（又称共享库），它相较于静态库而言有节省空间、便于更新等优点。但同时，动态库也有其缺点，加载速度相较于静态库而言较慢。那么，为什么调用动态库内的函数要比调用静态库内函数速度慢呢？它的加载过程具体又是怎样的呢？我们可借助gdb调试工具和反汇编工具objdump来找寻原因。

首先准备简单的动态库测试函数：

准备测试程序：



借助gcc工具生成动态库，链接动态库，编译生成可执行文件，并帮助动态链接器指定动态库加载位置。

1. gcc -c -fPIC add.c sub.c mul.c
2. gcc -shared -o libmymath.so add.o sub.o mul.o
3. gcc main.c -o app -L ./ -l mymath -I ./
4. export LD\_LIBRARY\_PATH=./

接下来，我们来研究下，在 main.c 中调用共享库的函数 add是如何实现的。首先反汇编看一下动态库libmymath.so，方便后期数据比对。(由于数据较多，这里只保留了与后期分析相关联的部分，同时为了方便观察地址，我们以32位系统为例。)

$ objdump libmymath.so -dS

...

00000538 <add>:

538: 55 push %ebp

539: 89 e5 mov %esp,%ebp

53b: 8b 45 0c mov 0xc(%ebp),%eax

53e: 8b 55 08 mov 0x8(%ebp),%edx

541: 01 d0 add %edx,%eax

543: 5d pop %ebp

544: c3 ret

Disassembly of section .fini:

...

然后我们反汇编一下可执行文件app的指令：

$ objdump -dS app

...

Disassembly of section .plt:

08048460 <add@plt-0x10>:

8048460: ff 35 04 a0 04 08 pushl 0x804a004

8048466: ff 25 08 a0 04 08 jmp \*0x804a008

804846c: 00 00 add %al,(%eax)

...

08048470 <add@plt>:

8048470: ff 25 0c a0 04 08 jmp \*0x804a00c

8048476: 68 00 00 00 00 push $0x0

804847b: e9 e0 ff ff ff jmp 8048460 <\_init+0x2c>

...

080485cd <main>:

int main(void)

{

80485cd: 55 push %ebp

80485ce: 89 e5 mov %esp,%ebp

80485d0: 83 e4 f0 and $0xfffffff0,%esp

80485d3: 83 ec 20 sub $0x20,%esp

int a = 5;

80485d6: c7 44 24 18 05 00 00 movl $0x5,0x18(%esp)

80485dd: 00

int b = 9;

80485de: c7 44 24 1c 09 00 00 movl $0x9,0x1c(%esp)

80485e5: 00

printf("%d + %d = %d\n", a, b, add(a, b));

80485e6: 8b 44 24 1c mov 0x1c(%esp),%eax

80485ea: 89 44 24 04 mov %eax,0x4(%esp)

80485ee: 8b 44 24 18 mov 0x18(%esp),%eax

80485f2: 89 04 24 mov %eax,(%esp)

80485f5: e8 76 fe ff ff call 8048470 <add@plt>

...

从上述反汇编结果来看add 函数并没有直接链接到可执行文件中。而且 call 8048470 <add@plt>这条指令调用的也不是 add 函数的地址。共享库是位置无关代码，在运行时可以加载到任意地址，其加载地址只有在动态链接时才能确定，所以在 main 函数中不可能直接通过绝对地址调用add函数，而是通过间接寻址来找 add 函数的。

对照上面的指令，我们使用 gdb 跟踪一下：

$ gdb app

...

(gdb) start

Temporary breakpoint 1 at 0x80485d6: file main.c, line 6.

Starting program: /home/itcast/lib/app

Temporary breakpoint 1, main () at main.c:6

6 int a = 5;

(gdb) si

7 int b = 9;

(gdb) si

9 printf("%d + %d = %d\n", a, b, add(a, b));

(gdb) si

0x080485ea 9 printf("%d + %d = %d\n", a, b, add(a, b));

(gdb) si

0x080485ee 9 printf("%d + %d = %d\n", a, b, add(a, b));

(gdb) si

0x080485f2 9 printf("%d + %d = %d\n", a, b, add(a, b));

(gdb) si

0x080485f5 9 printf("%d + %d = %d\n", a, b, add(a, b));

(gdb) si

0x08048470 in add@plt ()

跳转到 .plt 段中，现在将要执行一条 jmp \*0x804a00c指令，我们看看0x804a00c这个地址里存的是什么：

(gdb) x 0x804a00c

0x804a00c <add@got.plt>: 0x08048476

对应app反汇编结果，我们发现原来0x08048476就是其下一条指令push $0x0的地址。好，继续跟踪下去：

(gdb) si

0x08048470 in add@plt ()

(gdb) si

0x08048476 in add@plt ()

(gdb) si

0x0804847b in add@plt ()

(gdb) si

0x08048460 in ?? ()

(gdb) si

0x08048466 in ?? ()

(gdb) si

0xf7ff04f0 in ?? () from /lib/ld-linux.so.2

最终进入了动态链接器 /lib/ld-linux.so.2 ，在其中完成动态链接的过程并调用 add 函数，我们不深入这些细节了，直接用 finish 命令返回到 main 函数：

(gdb) si

0xf7ff04f2 in ?? () from /lib/ld-linux.so.2

(gdb) finish

Run till exit from #0 0xf7ff04f2 in ?? () from /lib/ld-linux.so.2

0x080485fa in main () at main.c:9

9 printf("%d + %d = %d\n", a, b, add(a, b));

这时，再来看看0x804a00c这个地址里保存的是什么：

(gdb) x 0x804a00c

0x804a00c <add@got.plt>: 0xf7fd4538

(gdb) x 0xf7fd4538

0xf7fd4538 <add>: 0x8be58955

我们发现0x804a00c中不再保存其下一条指令push $0x0的地址，而存入了一个新的地址，继续跟踪这个地址找到了add函数真正被加载到内存的位置。其中的0x8be58955正对应文档开头反汇编动态库所得到的add函数前三条指令。由于我们所使用的计算机采用小端法存储，所以低位保存在低字节上。

动态链接器已经把 add 函数的地址存在这里了，所以下次再调用 add 函数就可以直接从 jmp \*0x804a00c 指令直接跳到它首条指令的地址，而不必再进入 /lib/ld-linux.so.2 做动态链接了。

我们首次查看0x804a00c的时候，其内部并没有保存add函数实际的地址。而当函数被调用，动态链接器加载完成，会将add真正加载至内存的地址填写到与plt对应的got中。有一种描述这种绑定动态库函数的方式，称之为“延迟绑定”。正是由于首次调用的这一延迟，导致调用动态库函数不像调用静态库函数那样快捷。