本文主要想描述一个程序从源代码到加载到内存中各个阶段程序的大概的样子，便于大家在脑海里形成可视化的图像，且能够对程序有个整体把握，从微观看程序。本文讲解主要针对linux x86\_64平台。限于篇幅，具体细节请查阅文章最后的参考资料。

文章主要分5部分：

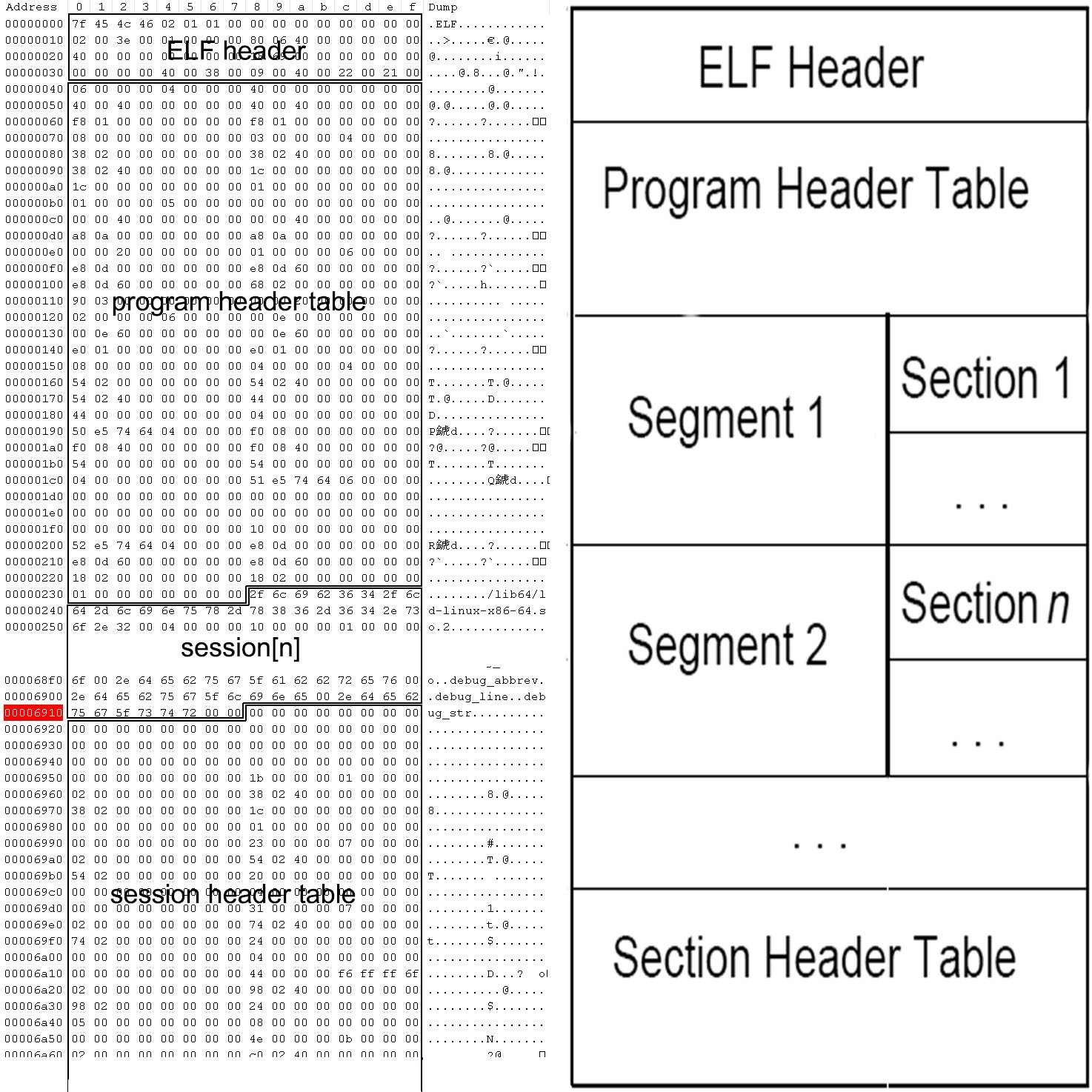
* 程序源码
* 程序存放在硬盘中的样子
* 程序加载到内存中的样子
* 程序访问动态库的函数的过程
* 参考资料

# 程序源码

1. #include<iostream>
2. #include<cstdio>
4. **int** add\_print(**int** a, **int** b)
5. {
6. **int** c = a + b;
7. std::cout << c << std::endl;
8. **return** c;
10. }
11. **int** main(**int** argc, **char**\* argv[], **char**\* env[])
12. {
13. **int** aa = 1;
14. **int** bb = 2;
15. **int** cc = add\_print(aa, bb);
16. std::printf("%d,%d,%d\n",aa,bb,cc);
17. **return** 0;
18. }

|  |
| --- |
| gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1~18.04.1) ，cpu架构x86\_64  编译程序  g++ -no-pie -fno-stack-protector -g test.cpp -o test |
|  |

# 程序存放在硬盘中的样子



Linux应用程序二进制接口(ABI, Application binary Interface)规定的目标文件格式为ELF(Executable and Linking Format)；Windows下使用PE(Portable Executable)；MacOS使用Mach-O(Mach Object)。本文主要讲解Linux下的ELF格式。ABI规范包括两部分gABI（Generic ABI即通用ABI）和psABI(Processor Suppliment ABI即处理器补充ABI)。

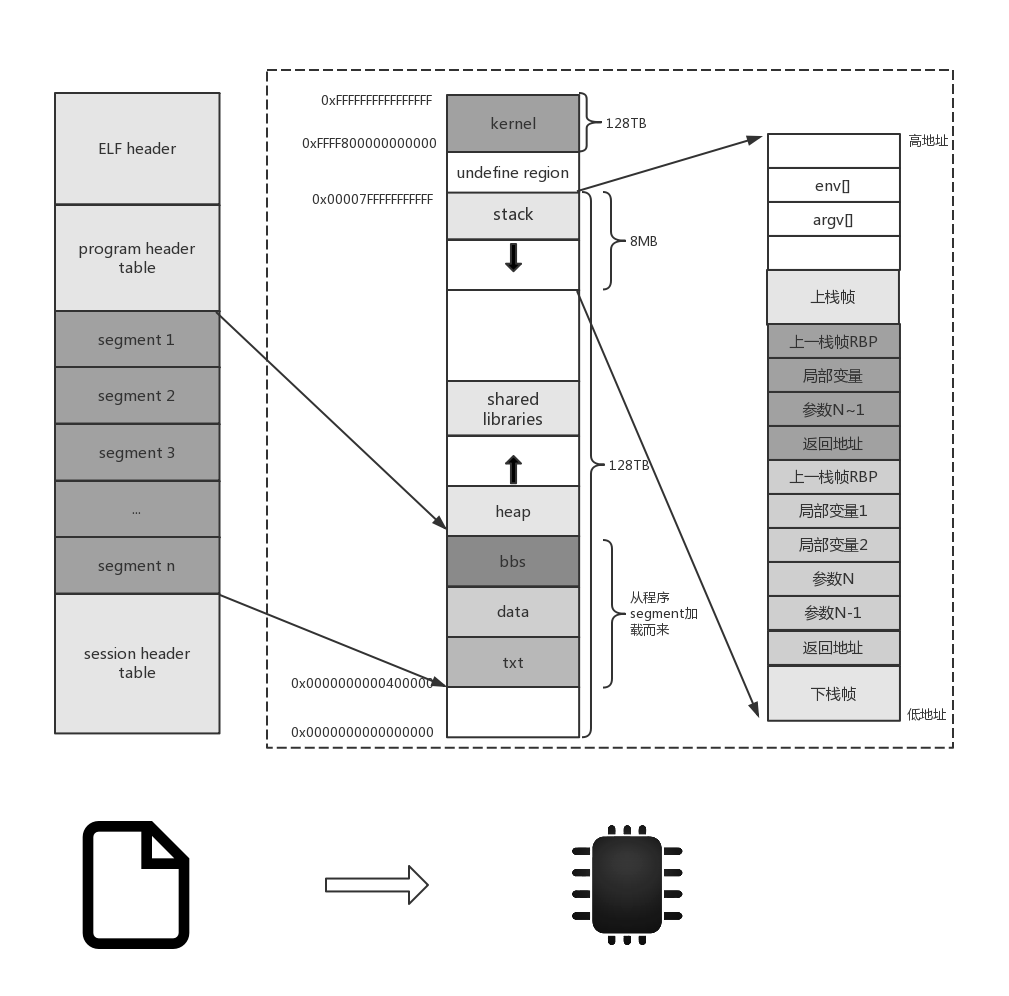
ELF文件格式对程序文件的内容提供了双重视图。Section链接时使用，保存着大量用于链接的目标文件信息：指令、数据、符号表、重定位信息等。Segment运行时使用，保存文本、数据、堆栈等信息。

|  |  |
| --- | --- |
| ELF头部在文件的开始部分，保存着关于此程序文件组织的路线图（road map） | |
| Start of program headers | 程序头部表的起始位置 |
| Size of program headers \* Number of program headers | 程序头部表的大小 |
| Start of section headers | 节头部表的起始位置 |
| Size of section headers \* Number of section headers | 节头部表的大小 |
| Size of this header | ELF头部大小 |

|  |
| --- |
| 查看程序ELF信息  readelf -a test  **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: 0x400680  Start of program headers: 64 (bytes into file)  Start of section headers: 26904 (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: 34  Section header string table index: 33  **Section Headers: (部分省略)**  [Nr] Name Type Address Offset  Size EntSize Flags Link Info Align  [ 0] NULL 0000000000000000 00000000  0000000000000000 0000000000000000 0 0 0  [ 1] .interp PROGBITS 0000000000400238 00000238  000000000000001c 0000000000000000 A 0 0 1  [13] .text PROGBITS 0000000000400680 00000680  0000000000000252 0000000000000000 AX 0 0 16  [23] .data PROGBITS 0000000000601040 00001040  0000000000000010 0000000000000000 WA 0 0 8  [24] .bss NOBITS 0000000000601060 00001050  0000000000000118 0000000000000000 WA 0 0 32  [33] .shstrtab STRTAB 0000000000000000 000067d4  0000000000000143 0000000000000000 0 0 1  **Program Headers:**  Type Offset VirtAddr PhysAddr  FileSiz MemSiz Flags Align  PHDR 0x0000000000000040 0x0000000000400040 0x0000000000400040  0x00000000000001f8 0x00000000000001f8 R 0x8  INTERP 0x0000000000000238 0x0000000000400238 0x0000000000400238  0x000000000000001c 0x000000000000001c R 0x1  [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]  LOAD 0x0000000000000000 0x0000000000400000 0x0000000000400000  0x0000000000000aa8 0x0000000000000aa8 R E 0x200000  LOAD 0x0000000000000de8 0x0000000000600de8 0x0000000000600de8  0x0000000000000268 0x0000000000000390 RW 0x200000  DYNAMIC 0x0000000000000e00 0x0000000000600e00 0x0000000000600e00  0x00000000000001e0 0x00000000000001e0 RW 0x8  NOTE 0x0000000000000254 0x0000000000400254 0x0000000000400254  0x0000000000000044 0x0000000000000044 R 0x4  GNU\_EH\_FRAME 0x00000000000008f0 0x00000000004008f0 0x00000000004008f0  0x0000000000000054 0x0000000000000054 R 0x4  GNU\_STACK 0x0000000000000000 0x0000000000000000 0x0000000000000000  0x0000000000000000 0x0000000000000000 RW 0x10  GNU\_RELRO 0x0000000000000de8 0x0000000000600de8 0x0000000000600de8  0x0000000000000218 0x0000000000000218 R 0x1  **Section to Segment mapping:**  Segment Sections...  00  01 .interp  02 .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version\_r .rela.dyn .rela.plt .init .plt .text .fini .rodata .eh\_frame\_hdr .eh\_frame  03 .init\_array .fini\_array .dynamic .got .got.plt .data .bss  04 .dynamic  05 .note.ABI-tag .note.gnu.build-id  06 .eh\_frame\_hdr  07  08 .init\_array .fini\_array .dynamic .got |

注：段和节类型的作用请查看参考文档

# 程序加载到内存中的样子



x86架构下的Linux虚拟内存布局， 0x00000000-0xbfffffff(3G)为用户空间, 0xc0000000-0xffffffff(1G)为内核空间。 x86-64架构下的虚拟内存布局，0x0000000000000000- 0x00007fffffffffff(128TB)为用户空间, 0xffff800000000000-0xffffffffffffffff(128TB)为内核空间。

x86-64架构CPU都遵循AMD的Canonical form, 即只有虚拟地址的最低48位才会在地址转换时被使用, 且任何虚拟地址的48位至63位必须与47位一致(sign extension)。 也就是说, 总的虚拟地址空间为256TB( 2^48 )。

* stack

栈；可读可写；函数栈帧，存放参数和局部变量等数据

* heap

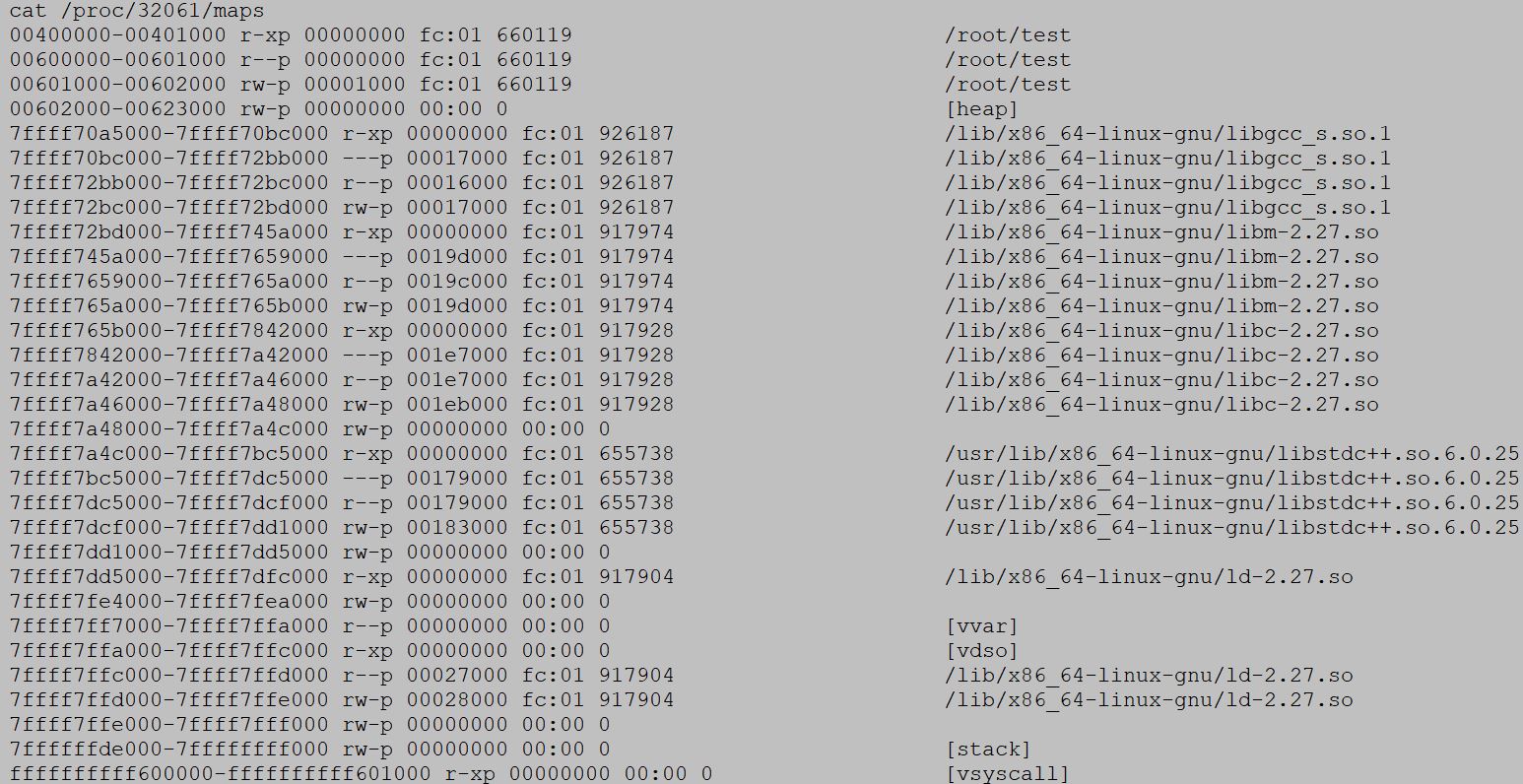
堆；可读可写；动态分配，如new、malloc等

* data、bbs

数据段；可读可写；全局变量、静态变量、常量字符串等数据

* text

代码段；可读可执行；可执行的机器指令



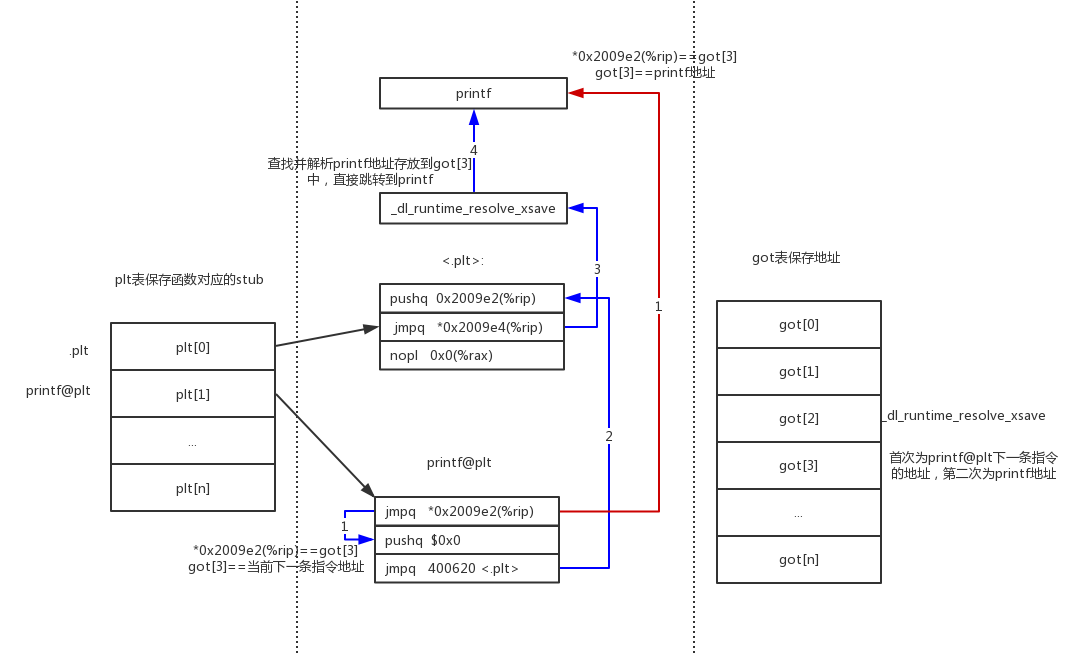
图上显示进程内存映射，包括了程序本身的text、bss、dada段，还有程序依赖的动态库的映射。



Linux x64和x86相比，主要区别在于参数的传递上：

* x86使用栈传递全部参数。GCC默认将函数参数从右至左push到栈中。由函数调用方负责平衡栈。
* x64优先使用寄存器传递参数。对于前6个参数，分别使用rdi, rsi, rdx, rcx, r8, r9传递参数。参数超过6个时使用栈传递额外的参数。同样由调用方平衡栈。
* 二者都使用eax/rax存储函数返回值。

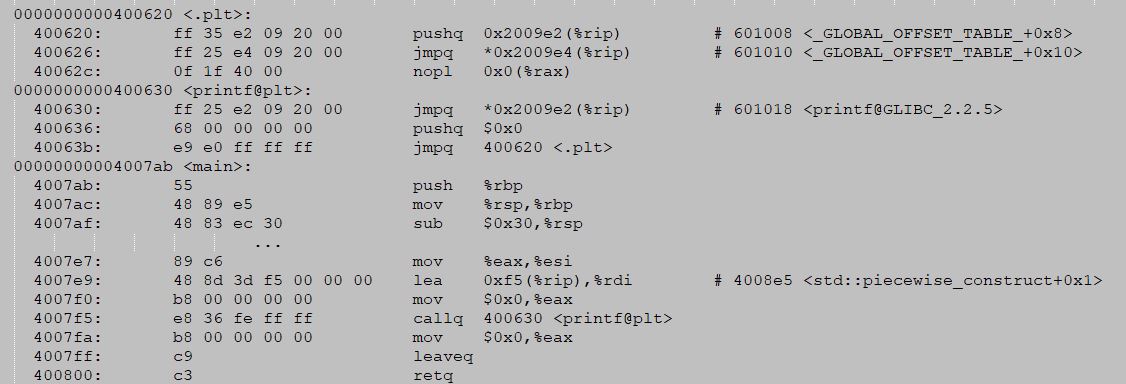
# 程序访问动态库的函数的过程



程序首次调用printf时走上图蓝色线，之后调用走红色线。

* GOT（Global Offset Table，全局偏移表）是Linux ELF文件中用于定位全局变量和函数的一个表。GOT表前三项是特殊的：GOT[0]包含.dynamic段的地址，.dynamic段包含了动态链接器用来绑定函数地址的信息，比如符号的位置和重定位信息;GOT[1]包含动态链接器的标识;GOT[2]包含动态链接器的延迟绑定代码的入口点(\_dl\_runtime\_resolve\_xsave)。GOT的其他表项为本模块要引用的一个全局变量或函数的地址。
* PLT（Procedure Linkage Table，过程链接表）是Linux ELF文件中用于延迟绑定的表，即函数第一次被调用的时候才进行绑定。PLT是一个以16字节表项的数组形式出现的代码序列。其中PLT[0]是一个特殊的表项，它跳转到动态链接器中执行;每个定义在共享库中并被本模块调用的函数在PLT中都有一个表项，从PLT[1]开始．模块对函数的调用会转到相应PLT表项中执行，这些表项由三条指令构成。第一条指令是跳转到相应的GOT存储的地址值中．第二条指令把函数相应的ID压入栈中，第三条指令跳转到PLT[0]中调用动态链接器解析函数地址，并把函数真正地址存入相应的GOT表项中。被调用函数GOT对应表项中存储的最初地址为相应PLT表项中第二条指令的地址值，函数第一次被调用后．GOT表项中的值就为函数的真正地址。因此，第一次调用函数时开销比较大．但是其后的每次调用都只会花费一条指令和一个间接的存储器引用。

|  |
| --- |
| 反汇编程序  objdump -d test | c++filt |



# 参考资料

《Tool Interface Standard (TIS) Portable Formats Specification, version 1.2》

《System V ABI Edition 4.1》

《[System V ABI - DRAFT 24 April 2001](https://refspecs.linuxfoundation.org/elf/gabi4+/contents.html)》

《System V Application Binary Interface x86-64 Architecture Processor Supplement Draft Version v0.99》

《Linux Extensions to gABI》

《DWARF Debugging Information Format Version 5》

《C++ ABI for Itanium, v1.75》

《Linkers & Loaders by John R. Levine》

《程序员的自我修养—链接、装载与库》

《Professional Assembly Language Richard Blum》

《x86\_x64体系探索及编程》

《Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 2 (2A, 2B, 2C & 2D):Instruction Set Reference, A-Z》

《Using the GNU Compiler Collection》

《The GNU Binary Utilities》

《Debugging with gdb》