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：

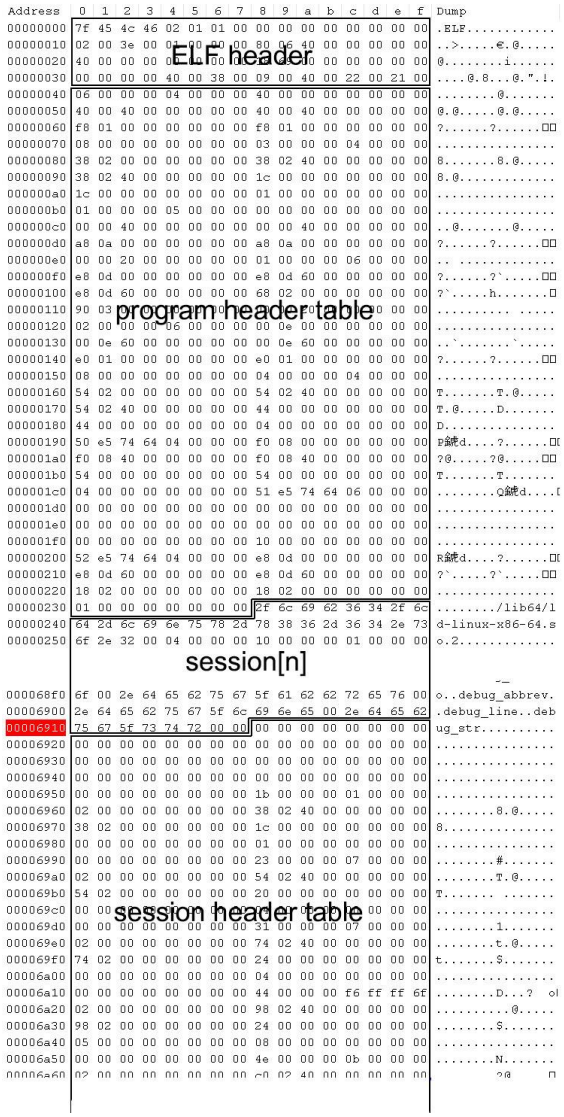
- 程序源码
- 程序存放在硬盘中的样子
- 程序加载到内存中的样子
- 程序访问动态库的函数的过程
- 参考资料

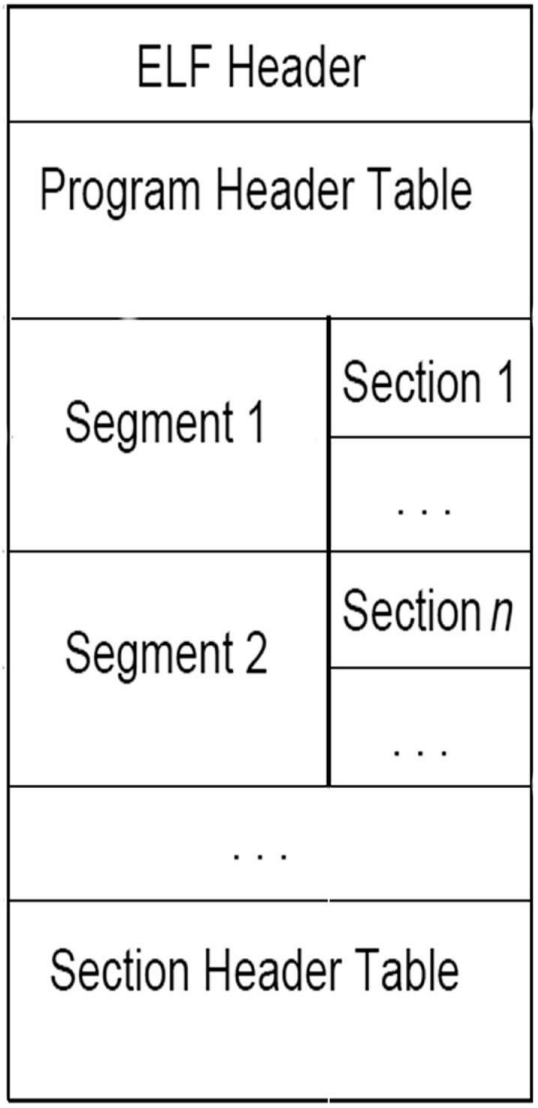
程序源码

```
1.  #include<iostream>
2.  #include<cstdio>
3.
4.  int add_print(int a, int b)
5.  {
6.      int c = a + b;
7.      std::cout << c << std::endl;
8.      return c;
9.
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	程序头部表的大小

headers	
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

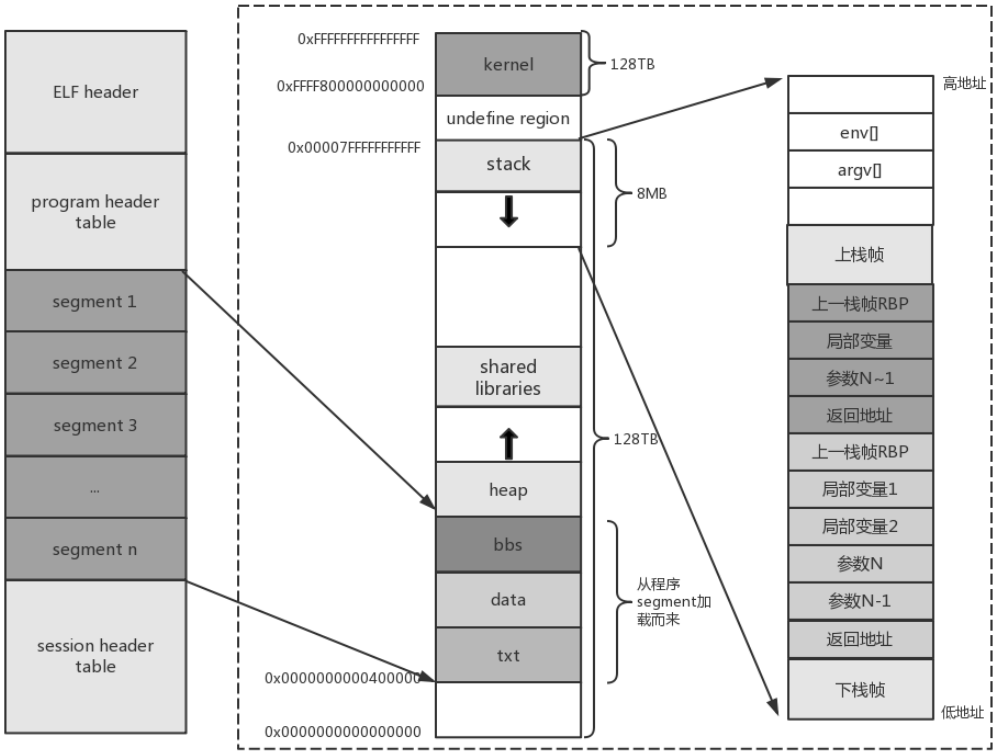
查看程序 ELF 信息

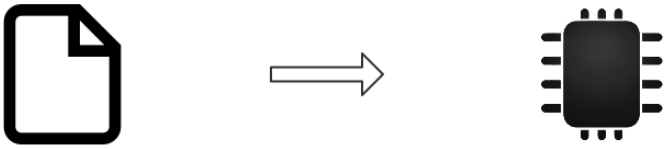
```
readelf -a test

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

```
cat /proc/32061/maps
00400000-00401000 r-xp 00000000 fc:01 660119                             /root/test
00600000-00601000 r--p 00000000 fc:01 660119                             /root/test
00601000-00602000 rw-p 00001000 fc:01 660119                             /root/test
00602000-00623000 rw-p 00000000 00:00 0                                  [heap]
7ffff70a5000-7ffff70bc000 r-xp 00000000 fc:01 926187                     /lib/x86_64-linux-gnu/libgcc_s.so.1
7ffff70bc000-7ffff72bb000 ---p 00017000 fc:01 926187                     /lib/x86_64-linux-gnu/libgcc_s.so.1
7ffff72bb000-7ffff72bc000 r--p 00016000 fc:01 926187                     /lib/x86_64-linux-gnu/libgcc_s.so.1
7ffff72bc000-7ffff72bd000 rw-p 00017000 fc:01 926187                     /lib/x86_64-linux-gnu/libgcc_s.so.1
7ffff72bd000-7ffff745a000 r-xp 00000000 fc:01 917974                     /lib/x86_64-linux-gnu/libm-2.27.so
7ffff745a000-7ffff7659000 ---p 0019d000 fc:01 917974                     /lib/x86_64-linux-gnu/libm-2.27.so
7ffff7659000-7ffff765a000 r--p 0019c000 fc:01 917974                     /lib/x86_64-linux-gnu/libm-2.27.so
7ffff765a000-7ffff765b000 rw-p 0019d000 fc:01 917974                     /lib/x86_64-linux-gnu/libm-2.27.so
7ffff765b000-7ffff7842000 r-xp 00000000 fc:01 917928                     /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7842000-7ffff7a42000 ---p 001e7000 fc:01 917928                     /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7a42000-7ffff7a46000 r--p 001e7000 fc:01 917928                     /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7a46000-7ffff7a48000 rw-p 001eb000 fc:01 917928                     /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7a48000-7ffff7a4c000 rw-p 00000000 00:00 0
7ffff7a4c000-7ffff7bc5000 r-xp 00000000 fc:01 655738                     /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25
7ffff7bc5000-7ffff7dc5000 ---p 00179000 fc:01 655738                     /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25
7ffff7dc5000-7ffff7dcf000 r--p 00179000 fc:01 655738                     /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25
7ffff7dcf000-7ffff7dd1000 rw-p 00183000 fc:01 655738                     /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25
7ffff7dd1000-7ffff7dd5000 rw-p 00000000 00:00 0
7ffff7dd5000-7ffff7dfc000 r-xp 00000000 fc:01 917904                     /lib/x86_64-linux-gnu/ld-2.27.so
7ffff7fe4000-7ffff7fea000 rw-p 00000000 00:00 0
7ffff7ff7000-7ffff7ffa000 r--p 00000000 00:00 0                          [vvar]
7ffff7ffa000-7ffff7ffc000 r-xp 00000000 00:00 0                          [vdso]
7ffff7ffc000-7ffff7ffd000 r--p 00027000 fc:01 917904                     /lib/x86_64-linux-gnu/ld-2.27.so
7ffff7ffd000-7ffff7ffe000 rw-p 00028000 fc:01 917904                     /lib/x86_64-linux-gnu/ld-2.27.so
7ffff7ffe000-7ffff7fff000 rw-p 00000000 00:00 0
7ffffffde000-7ffffffff000 rw-p 00000000 00:00 0                          [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

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。

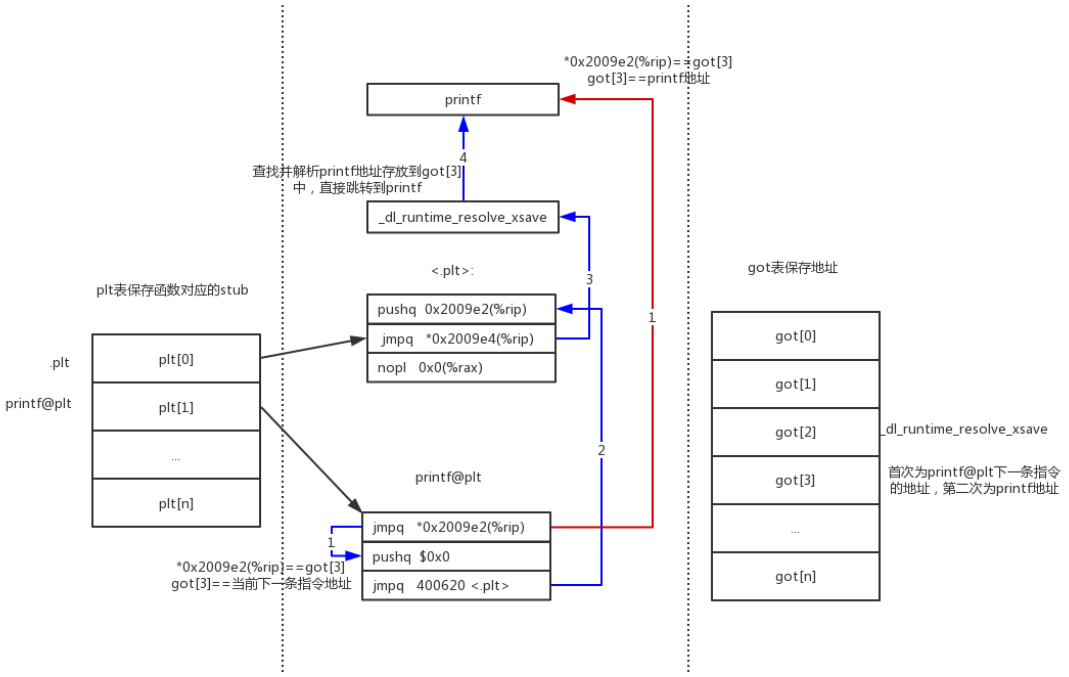
寄存器名（64位环境）	用　途	寄存器名（32位环境）	用　途
rax	第一函数值，操作用	eax	函数值，操作用
rbx	寄存器变量	ebx	寄存器变量
rcx	操作用，第四函数变量	ecx	操作用
rdx	操作用，第三函数变量，第二函数值	edx	操作用
rsi	操作用，第二函数变量	esi	寄存器变量
rdi	操作用，第一函数变量	edi	寄存器变量
r8	操作用，第五函数变量	—	
r9	操作用，第六函数变量	—	
r10~r11	操作用	—	
r12~r15	寄存器变量	—	
rbp	基指针，寄存器变量	ebp	基指针，寄存器变量
rip	程序计数器（提示下一条指令）	eip	程序计数器（提示下一条指令）

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

```
0000000000400620 <.plt>:
  400620:       ff 35 e2 09 20 00       pushq  0x2009e2(%rip)        # 601008 <_GLOBAL_OFFSET_TABLE_+0x8>
  400626:       ff 25 e4 09 20 00       jmpq   *0x2009e4(%rip)       # 601010 <_GLOBAL_OFFSET_TABLE_+0x10>
  40062c:       0f 1f 40 00             nopl   0x0(%rax)
0000000000400630 <printf@plt>:
  400630:       ff 25 e2 09 20 00       jmpq   *0x2009e2(%rip)       # 601018 <printf@GLIBC_2.2.5>
  400636:       68 00 00 00 00          pushq  $0x0
  40063b:       e9 e0 ff ff ff          jmpq   400620 <.plt>
00000000004007ab <main>:
  4007ab:       55                      push   %rbp
  4007ac:       48 89 e5                mov    %rsp,%rbp
  4007af:       48 83 ec 30             sub    $0x30,%rsp
                   ...
  4007e7:       89 c6                   mov    %eax,%esi
  4007e9:       48 8d 3d f5 00 00 00    lea    0xf5(%rip),%rdi       # 4008e5 <std::piecewise_construct+0x1>
  4007f0:       b8 00 00 00 00          mov    $0x0,%eax
  4007f5:       e8 36 fe ff ff          callq  400630 <printf@plt>
  4007fa:       b8 00 00 00 00          mov    $0x0,%eax
  4007ff:       c9                      leaveq
  400800:       c3                      retq
```

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》
《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》