程序员的自我修养之CSAPP

tips

- `gcc main.c -o main`，main 肯定是 elf 文件的类型，但是使用 `readelf -h main` 可以看到 main 的类型是这个，`DYN (Shared object file)`，原因是 gcc 开启了一个默认的配置选项 `--enable-default-pie`（使用 `gcc -v` 可以查看），so gcc create a dynamically linked position independent executable。如果期待的结果是可执行文件的elf，可以采用对应的参数禁止该选项 `gcc main.c -no-pie -o main`
- **符号** 包括
  - **函数名**
  - **全局变量名**
- **link** 处理的模块间 **变量的访问** 与 **函数的调用**
  - 静态link会把所有需要的符号都打包成一个elf，位置都是确定的，执行操作起来简单
  - 动态link，比如找printf的位置，在一个进程中，符号printf的位置保存在GOT中，但是第一次是没有的，所以call printf的时候先去PLT，再调用到GOT，GOT表如果有就完事了，如果GOT表没有的话（第一次调用），再次回到PLT表去 **调用linker** 确定printf的位置，并写到GOT表中
    - `GOT: global offset table`
      - 共享库中符号的绝对地址
        - 被延迟绑
    - `PLT: procedure linkage table`
      - **call printf -> PLT -> GOT**
- elf文件有两个视图，一个是link视图，一个是load视图，load视图中的那个段就是进程vm中的那个段
- 一个 hello world 程序的符号表中会包含 `printf`，`_init`，`_start` 等内容（**就是包括字符串的**）
  - 符号表 存在于 **可执行文件 elf 中的某 section 中**
    - 包括app与vmlinux
    - 都可以**通过 readelf -s elf/objdump -t elf 来查看**
  - 局部变量不算是符号，自然就不在符号表中，**对应的符号字符串自然也不在elf文件中**
  - 所以在学习反射的时候，下面的话是合理的
    - 反射是指在**程序运行期对程序本身进行访问和修改的能力**
    - 程序在编译时，变量被转换为内存地址，变量名不会被编译器写入到可执行部分
    - 在运行程序时，程序无法获取自身的信息
    - **所以需要额外的考量**
- gdb调试为什么可以调试局部变量呢?
  - 首先先要debug调试必须在编译阶段增加调试信息，否则是无法调试的
  - 其次函数的调用最后需要的是**返回值到rax**，中间的过程变量确实无需考虑
  - dwarf是debug文件格式，**被编译到elf中debug section中**
- `c/c++` **就是由main函数开始的**，**可以通过gdb判断**
  - **当然前面还有一些init/start等wrapper的启动位置**
- `objdump -j .text -s a.out`
  - 只显示某一个 section 的完整信息

1. hello world是如何跑起来的

```c
// sample.c
#include<stdio.h>
int main()
{
    printf("hello world!\n");
    return 0;
}
```

以上代码段的内存地址空间示意图，以及关于以下结果的一些说明

- https://www.runoob.com/w3cnote/cpp-header.html
- 首先，注释 `include<stdio.h>`， `gcc sample.c -o sample` 是会报错的，但是不影响执行
  - 实际上就是少了一个声明；但是在最后一步链接，是一定会link libc库的，printf这个符号的在libc库中是找的到的，是能够替换掉的；即运行是没有问题的
    - 原因
      - c/c++独立编译，如果遇到不认识的符号（**其它模块中定义的符号**），**就放到符号表中**
        - 例如，`main.c` 函数中，printf 就是在符号表中，连接 libc 库的时候，linker 会去找 libc 中找 printf 符号的地址，然后就知道 printf 的地址了

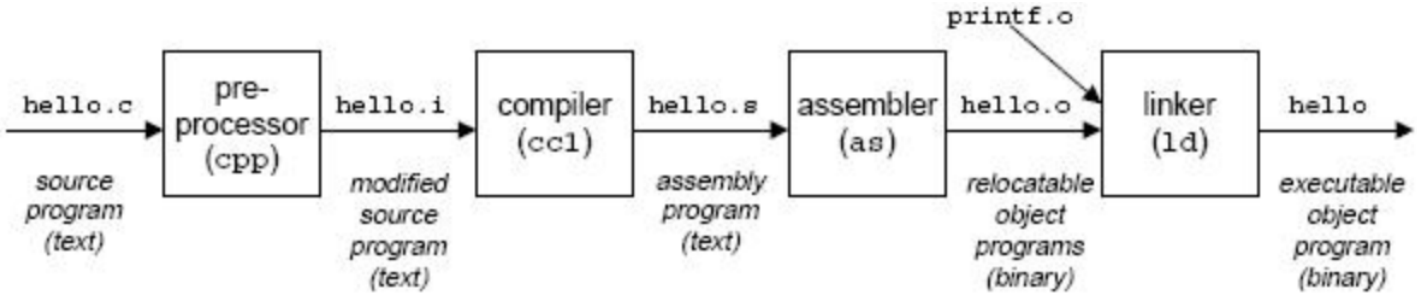
        - 声明的含义
            - 而 **声明** 则只是 **声明这个符号的存在**，即告诉编译器，**这个符号是在其他文件中定义的，我这里先用着，你链接的时候再到别的地方去找找看它到底是什么吧**
            - `#include<stdio.h>` 只是告诉编译器 printf 在其它地方是有定义的，别担心，放心的放到符号表中，link的时候我帮你搞定
    - 是否注释 `#include<stdio.h>` 仅仅影响预处理这一步， `no.i` 与 `yes.i` 是不一样的，因为预处理会展开，但是编译为 `.s` 的汇编码的时候，就完全相同了。包括最后编译为 `.o` 文件，二者没有任何区别
- `/proc` 是一个伪文件系统，导出进程运行时的状态
- 如上一个非常简单的 c 程序，加入 `sleep(-1)` 或 `for(;;){;}` 之后，就可以观察该进程的 maps 文件，除了映射程序代码本身之外，还额外包括了 `ld.so` 与 `libc.so`
    - 第一列：VMA
    - 第二列：Authority（读写执行，private or shared）
    - 第三列：offset
        - 如果有image文件，则表示该段在image文件中的偏移
    - 第四列：Image所在设备的主设备号与从设备号
    - 第五列：Image文件的inode号
    - 第六列：Image文件的路径，没有的是**匿名映射**

```
double_d@dd:/proc/14942$ cat maps
564942fba000-564942fbb000 r--p 00000000 08:13 133092                     /home/double_d/QuickTest/comp/no  # 有Image文件
564942fbb000-564942fbc000 r-xp 00001000 08:13 133092                     /home/double_d/QuickTest/comp/no
564942fbc000-564942fbd000 r--p 00002000 08:13 133092                     /home/double_d/QuickTest/comp/no
564942fbd000-564942fbe000 r--p 00002000 08:13 133092                     /home/double_d/QuickTest/comp/no
564942fbe000-564942fbf000 rw-p 00003000 08:13 133092                     /home/double_d/QuickTest/comp/no
564944b10000-564944b31000 rw-p 00000000 00:00 0                          [heap]  # 匿名映射
7f116add8000-7f116adfd000 r--p 00000000 08:13 7612494                    /lib/x86_64-linux-gnu/libc-2.29.so  # 有Image文件
7f116adfd000-7f116af44000 r-xp 00025000 08:13 7612494                    /lib/x86_64-linux-gnu/libc-2.29.so
7f116af44000-7f116af8d000 r--p 0016c000 08:13 7612494                    /lib/x86_64-linux-gnu/libc-2.29.so
7f116af8d000-7f116af8e000 ---p 001b5000 08:13 7612494                    /lib/x86_64-linux-gnu/libc-2.29.so
7f116af8e000-7f116af91000 r--p 001b5000 08:13 7612494                    /lib/x86_64-linux-gnu/libc-2.29.so
7f116af91000-7f116af94000 rw-p 001b8000 08:13 7612494                    /lib/x86_64-linux-gnu/libc-2.29.so
7f116af94000-7f116af9a000 rw-p 00000000 00:00 0
7f116afb4000-7f116afb5000 r--p 00000000 08:13 7612436                    /lib/x86_64-linux-gnu/ld-2.29.so  # 有Image文件
7f116afb5000-7f116afd3000 r-xp 00001000 08:13 7612436                    /lib/x86_64-linux-gnu/ld-2.29.so
7f116afd3000-7f116afdb000 r--p 0001f000 08:13 7612436                    /lib/x86_64-linux-gnu/ld-2.29.so
7f116afdb000-7f116afdc000 r--p 00026000 08:13 7612436                    /lib/x86_64-linux-gnu/ld-2.29.so
7f116afdc000-7f116afdd000 rw-p 00027000 08:13 7612436                    /lib/x86_64-linux-gnu/ld-2.29.so
7f116afdd000-7f116afde000 rw-p 00000000 00:00 0
7fffb4202000-7fffb4223000 rw-p 00000000 00:00 0                          [stack]  # 匿名映射
7fffb4254000-7fffb4257000 r--p 00000000 00:00 0                          [vvar]
7fffb4257000-7fffb4259000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

some tips:

- c语言的每一个单独的脚本称为一个**模块**，这个概念之前一直都没有建立起来
    - 可以类比 python 中的模块
    - 内核代码也是有模块的概念的
- **符号与变量的区别**：
    - 例如， `int a=1` 函数内部的定义，是栈去管理这个符号的，**并不是我们这里讨论的符号**
        - 栈上对象的生命周期是有限的，可以用来实现RALL
    - 符号的类型
        - 由模块 m 定义，能够被其他模块引用的符号，**global symbol**
        - 由其它模块定义并且被模块 m 引用的全局符号，称为外部符号，including **var** and **fun**
            - **func默认是全局的**，如果不希望被引用，**需要 static 来修饰，避免默认的 extern**
            - **var 默认是模块局部的的**，need **extern**声明
        - local symbol use **static**
            - **static** means private, otherwise public
            - 静态局部变量 in func
                - 存储位置， `.data` or `.bss`
                - 生命周期，不随着函数调用栈的结束而结束
- 链接程序 **ld** 始终将 **libc.so** C 标准库以及其它用于启动程序的库与用户的 `.o` 链接在一起生成最后的可执行文件
    - `libc.so`
        - glibc，C 标准库，操作系统API以及常用函数的封装

        - 发布一般包含三部分：
            - 头文件，例如 `stdio.h` in `/usr/lib/include`
            - 二进制的库文件
                - 动态库 `libc.so`， `/usr/lib/x86_64-linux-gnu/libc.so`
                - 静态库 `libc.a`，`/usr/lib/x86_64-linux-gnu/libc.a`
                    - 一组目标文件的集合，是一种特殊的文件类型，archive，即存档，存档文件 `.a`
                    - ar 指令将这些目标文件压缩到一起后形成的备存文件 `libc.a`
                    - 可以用 `ar -t libc.a` 可以查看都包含哪些.o
    - ld 链接程序
        - objdump -s no，elf 文件中会有一个 section，指明加载器的路径
            - 是 disk 上的一个路径
            - `Contents of section .interp: 02a8 2f6c6962 36342f6c 642d6c69 6e75782d 02b8 7838362d 36342e73 6f2e3200 /lib64/ld-linux-x86-64.so.2` 这个符号链接指向了 `ld-2.29.so`
        - `ld-2.29.so` 会先执行，把 `libc.so` 等动态库加载到该进程中，**符号处理完成之后**；将控制权交还给程序本身（后面详细描述）
        - **链接**概述
            - 最本质的功能是 **链接符号与符号的地址**
            - 将各个模块拼接成一个文件的过程，这个文件可以被**加载，或称为复制**到内存中执行



程序被按照某种指令集（ISA）的规范翻译为CPU可识别的底层代码的过程叫做编译，这个CPU可以识别的底层代码，就可以由磁盘加载到内存中执行。然后典型的几个步骤吧

1. 预处理
    - 文本到文本：去注释，展开include以及宏等等，处理编译条件，替换常量等等
    - `gcc -E main.c -o main.i`
        - -E means Preprocess only; do not compile, assemble or link
2. 编译
    - 文本到汇编代码
    - `gcc -S main.c -o main.s`
        - -S means Compile only; do not assemble or link
    - parse tree，语义分析，对于a=1（=是root，a是left leave，1是right leave）
3. 汇编
    - elf文件的生成
    - `gcc -c main.s -o main.o`
        - -c means Compile and assemble, but do not link
    - `as main.s -o main.o`
        - as是汇编指令
4. 链接
    - **可执行的 elf 文件的生成**
    - 静态链接以及动态链接
    - `ld main.o x1.o x2.o ...`
        - 想要通过 ld 连接起一个可以运行的 hello word 是**非常困难的**
    - 简单说一下静态连接
        - main.o 称为 relocatable file，包含对**变量的访问**，**对函数的调用**，地址还是未知的
        - printf 本来应该是一个 callq 指令，但是在没有链接之前，是无法知道 printf 的地址的，所以是凉凉的，所以 link 的这个过程，就是要重新计算打印 hello world 的这一条指令中 printf 的位置，这个过程叫**重定位**
        - 一个很重要的概念就是**符号**
            - 模块（不同.o）间的变量访问
            - 模块（不同.o）间的函数调用

        - 独立的.o中，可能对有些变量的访问，或函数的调用是没有准确的地址的，因为这些**符号**位于其它模块中；link的作用就是把所有的模块link到一起之后，以上帝视角遍历一遍，把之前不确定的替换掉，即重定位的过程
        - 最后这样的一个可执行文件，是在磁盘上的；**但是它是一个有顺序的二进制文件**。即将来被加载到线性地址空间中的地址现在就已经是确定的了，即VMA是已经确定好了的
    - 动态link后面描述

加深理解一下高级语言到二进制文件的转换过程

```
# cpu所执行的就是二进制文件，指令集负责划分与识别指令
01010000010100001001100101001010100101

# 将二进制转化为16进制
e5894855
03fc45c7
b8000000
    c35d

# 将上述二进制所转为的16进制转化为指令集表示
           55  pushq %rbp
        e58948  movq  %rsp, %rbp
00000003fc45c7  movl  $3, -4(%rbp)
   000000000b8  movl  $0, %eax
           5d  popq  %rbp
           c3  ret
```

来看一下elf文件里有什么

- elf: executable linkable format
- 四种类型
    1. relocatable file: .o file
        - `file main.o`
    2. executable file
        - `file /bin/bash`
    3. shared object file: .so file
        - `file /lib/x86_64-linux-gnu/ld-2.29.so`
    4. coredump file
        - **gdb app core**

来看一下elf文件的细节与结构，将以下代码编译为 `.o`，并且利用readelf或objdump查看 `.o` 文件的一些细节

- 利用 `gcc -c main.c -o main.o` 将其编译成为一个relocated的elf文件
- 这个 elf 文件是不能被直接 load 到 memory 中去执行的，是link view，不是为了load to memory准备的
- 如果要 load to memory，即load到线性地址空间中，如果以 section 为 标准去加载，则可能会有碎片。而加载到内存的时候，这里关注的是内存中每一段虚拟地址空间的属性（**读写，执行，以及是否private**），所以会把相同属性的section组合，形成一个segment的view。然后以segment为单位加载到内存中；**代码段，数据段等**。所以这里要了解一些elf文件的两种视图：
    - link view：section, for relocatable
    - execution view：segment, for load and execution

```
// 没有printf函数的定义，是一个外部量，printf的调用地址是需要load完成之后重定位的
int printf(const char* format, ...);  //函数声明，c语言允许定义参数数量可变的函数
int global_init_var = 84;
int global_uninit_var;

void func1(int i){
        printf("%d\n", i);
}
int main()
{
        static int static_var = 85;
        static int static_var2;
        int a=1;
        int b;
        func1(static_var + static_var2 + a + b);
    return a;
}
```

先用 `readelf -h main.o` 看一下这个 `.o` 的header，实际上就是这个elf文件的概述了

- elf header结构以及常数定义在 `/usr/include/elf.h`
- Use header to find section table, and through section table to view all elf file，其实就是一个多级索引的文件存储结构
- 它能够直接索引到的内容主要有下面2个：
  - Start of section headers：index section table directly
  - Section header string table index

```
double_d@dd:~/QuickTest/comp$ readelf -h main.o
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00  # Magic number，魔数可以用于确定可执行文件的类型，os可以根据魔数字判断文件类型是否正确，以及
  Class:                             ELF64  # 机器字节长度，64
  Data:                              2's complement, little endian  //字节序
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0  # 程序入口的虚拟地址，64地址空间中的那个地址；execute type会有，.o没有为0
  Start of program headers:          0 (bytes into file)  # segment的load view，.o是没有的，所以是0
  Start of section headers:          1096 (bytes into file)  # the offset of section table
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)  # the size of section table descriptor，如果这里有13个section，那么section header table的size就是1
  Number of section headers:         13  # num of sections
  Section header string table index: 12  # 段表字符串表在section table中的下标
```

there is a graph of a elf file with link view:

```
| ELF Header          |  # first index
| .text               |
| .data               |
| .bss                |  # block storage start, but better save space is better，仅标记，不占位
| ...other sections   |
| section header table|  # section index
| string table        |
| symbol table        |
```

再用 `objdump -h main.o` 来查看一下main.o中各个**基本**header的基本信息，。一些辅助性的section， `objdump -h` 是不会显示的，需用用 `readelf -S` 来查看，会将所有的辅助性段全部show出来

- 第一段内容与第二段内容是可以很好的对应上的
- `size main.o` 能够打印一些数据出来，方便展示
  - text: bytes of code segment
  - data: size of data
  - bss: size of bss
  - dec/hex: total bytes of main.o
- we have no need to care about string table and string table of section header, they are just string, like ".text" in .shstrtab and "hello world" in .strtab. what we should care are symbol table and relocatable table

```
double_d@dd:~/QuickTest/comp$ size main.o
text    data     bss     dec     hex filename
179       8       4     191      bf main.o

# Display the contents of the section headers
double_d@dd:~/QuickTest/comp$ objdump -h main.o

main.o:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000057  0000000000000000  0000000000000000  00000040  2**0  # code
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE                  # CONTENTS代表要实际占据elf文件的位置
  1 .data         00000008  0000000000000000  0000000000000000  00000098  2**2  # data
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  0000000000000000  0000000000000000  000000a0  2**2  # block by started by symbol，未初始化的全局变量与局部静态变量会在这
                  ALLOC
  3 .rodata       00000004  0000000000000000  0000000000000000  000000a0  2**0  # read only
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      00000027  0000000000000000  0000000000000000  000000a4  2**0  # 注释信息段
                  CONTENTS, READONLY
  5 .note.GNU-stack 00000000  0000000000000000  0000000000000000  000000cb  2**0  # 堆栈提示段
                  CONTENTS, READONLY
  6 .eh_frame     00000058  0000000000000000  0000000000000000  000000d0  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA

# Display the full contents of all sections requested
double_d@dd:~/QuickTest/comp$ objdump -s main.o

main.o:     file format elf64-x86-64

Contents of section .text:
offset in section   binary                 ASCII
 0000 554889e5 4883ec10 897dfc8b 45fc89c6  UH..H....}..E...
 0010 488d3d00 000000b8 00000000 e8000000  H.=.............
 0020 0090c9c3 554889e5 4883ec10 c745fc01  ....UH..H....E..
 0030 0000008b 15000000 008b0500 00000001  ................
 0040 c28b45fc 01c28b45 f801d089 c7e80000  ..E....E........
 0050 00008b45 fcc9c3                      ...E...
Contents of section .data:
 0000 54000000 55000000                    T...U...  # 8个字节，分别是84与85，即global_init_var=84; static_var=85;
Contents of section .rodata:
 0000 25640a00                             %d..  # "%d\n"
Contents of section .comment:
 0000 00474343 3a202844 65626961 6e20392e  .GCC: (Debian 9.
 0010 322e312d 31392920 392e322e 31203230  2.1-19) 9.2.1 20
 0020 31393131 303900                      191109.
Contents of section .eh_frame:
 0000 14000000 00000000 017a5200 01781001  .........zR..x..
 0010 1b0c0708 90010000 1c000000 1c000000  ................
 0020 00000000 24000000 00410e10 8602430d  ....$....A....C.
 0030 065f0c07 08000000 1c000000 3c000000  ._..........<...
 0040 00000000 33000000 00410e10 8602430d  ....3....A....C.
 0050 066e0c07 08000000                    .n......

# Display the sections' header
# section table is a array
double_d@dd:~/QuickTest/comp$ readelf -S main.o
There are 13 section headers, starting at offset 0x448:

Section Headers:
  [Nr] Name              Type      Address           file Offset Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL      0000000000000000  00000000    0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS  0000000000000000  00000040    0000000000000057  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA      0000000000000000  00000338    0000000000000078  0000000000000018   I      10     1     8
  [ 3] .data             PROGBITS  0000000000000000  00000098    0000000000000008  0000000000000000  WA       0     0     4
  [ 4] .bss              NOBITS    0000000000000000  000000a0    0000000000000004  0000000000000000  WA       0     0     4
  [ 5] .rodata           PROGBITS  0000000000000000  000000a0    0000000000000004  0000000000000000   A       0     0     1
  [ 6] .comment          PROGBITS  0000000000000000  000000a4    0000000000000027  0000000000000001  MS       0     0     1
  [ 7] .note.GNU-stack   PROGBITS  0000000000000000  000000cb    0000000000000000  0000000000000000           0     0     1
  [ 8] .eh_frame         PROGBITS  0000000000000000  000000d0    0000000000000058  0000000000000000   A       0     0     8
  [ 9] .rela.eh_frame    RELA      0000000000000000  000003b0    0000000000000030  0000000000000018   I      10     8     8
  [10] .symtab           SYMTAB    0000000000000000  00000128    0000000000000198  0000000000000018          11    11     8
  [11] .strtab           STRTAB    0000000000000000  000002c0    0000000000000071  0000000000000000           0     0     1
  [12] .shstrtab         STRTAB    0000000000000000  000003e0    0000000000000061  0000000000000000           0     0     1
Key to Flags:
```

```
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
l (large), p (processor specific)
```

重点关注一下符号表与重定位相关表的内容

relocation table

- in main.c, only the address of printf is unknow, so just rela.text, no rel.date
  - 代码段如果有需要重定位的内容，那么就有 `rela.text` ，如果数据段有需要重定位的内容，则 `rel.data` ，本例中是没有的
- 首先用一个单独的小例子来说一下静态链接
  - 模块 a 中引用了 shared 与 swap
    - 变量的访问
    - 函数的调用
  - `a.o` 与 `b.o` 链接的结果就是一个可执行文件：
    1. 两个 `.o` 中的相似段去合并（**符号表会拼接成为为全局的符号表**）
    2. 根据segment，分配VM（VM本身是按照更细粒度的page去管理的）
    3. **parse symbol and relocate**
       - **链接器**会根据重定位表去check每一个重定位入口（对符号的引用）
       - 拿着这个符号去全局的符号表中查到这个符号确定的位置（**符号表中是有这个符号被映射之后的确切位置的**）
       - 然后就可以替换了
  - 可以用ld指令将 `a.o` 与 `b.o` 链接起来的说 `ld a.o b.o -e main -o ab`
    - `-e main` 表示以 `main` 函数作为程序的 entry，否则默认是 `_start` ，但是这里是没有链接 `libs.so` 的，根本没有 `_start` 这个符号
  - `objdump -r a.o` 可用于分析a.o中的relocate tabe
    - 对于每一个需要被重定位的段，elf都会有一个与之对应的重定位表
      - 如果指令部分.text需要被重定位，那么就需要有一个section named .rel.text
      - 如果数据部分.data需要被重定位，那么就需要有一个section named .rel.data
    - 一个重定位表会包含若干重定位项，每一个重定位项都是对一个符号的引用

```c
//a.c
//shared, swap两个符号，定义在b模块，这两个符号总归是要被替换掉的
extern int shared;
int main()
{
    int a = 100;
    swap(&a, &shared);
}

//b.c
int shared = 1;
void swap(int* a, int* b)
{
    *a ^= *b ^= *a ^= *b;  //按位异或
}
```

- gcc -c a.c b.c
- objdump -d a.o
- objdump -d b.o

```
# 先看一下objdump -d的数据
double_d@dd:~/QuickTest$ objdump -d a.o

a.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
offset  ins
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 10             sub    $0x10,%rsp
   8:   c7 45 fc 64 00 00 00    movl   $0x64,-0x4(%rbp)
   f:   48 8d 45 fc             lea    -0x4(%rbp),%rax
  13:   48 8d 35 00 00 00 00    lea    0x0(%rip),%rsi        # 1a <main+0x1a>  //对shared的访问也需要地址，没有地址，先用0顶上先
  1a:   48 89 c7                mov    %rax,%rdi
  1d:   b8 00 00 00 00          mov    $0x0,%eax
  22:   e8 00 00 00 00          callq  27 <main+0x27>  # 调用swap函数的时候，swap的地址是未知的，所以直接去调下一条指令了，相当于没有call直接执行下去
  27:   b8 00 00 00 00          mov    $0x0,%eax
  2c:   c9                      leaveq
  2d:   c3                      retq

# ld a.o b.o -e main -o ab
# 这个结果不同于可重定位的.o文件，这里的VMA与LMA都已经有值了，即程序启动后，对应的segment会加载到程序的VM的对应位置
double_d@dd:~/QuickTest$ objdump -h ab

ab:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000079  0000000000401000  0000000000401000  00001000  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .eh_frame     00000058  0000000000402000  0000000000402000  00002000  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00000004  0000000000404000  0000000000404000  00003000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .comment      00000026  0000000000000000  0000000000000000  00003004  2**0
                  CONTENTS, READONLY

# objdump -r a.o 检查一下a.o的重定位表（部分关键内容）
# 用于说明这个.o中有2个内容是要在link的时候去重定位的
# Type指的是位置修正的策略或者说是类型
double_d@dd:~/QuickTest$ objdump -r a.o

a.o:     file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:  # 这是一个代码段的重定位表
OFFSET           TYPE              VALUE
0000000000000016 R_X86_64_PC32     shared-0x0000000000000004  # 重定位入口，该位置相对于.text段中的偏移就是0x16，对应mov指令
0000000000000023 R_X86_64_PLT32    swap-0x0000000000000004  # 重定位入口，该位置相对于.text段中的偏移就是0x23，对应call指令


Relocation section '.rela.text' at offset 0x338 (file offset) contains 5 entries:
Offset in .text   Info         Type            Sym. Value       Sym. Name + Addend
000000000013      000500000002 R_X86_64_PC32   0000000000000000 .rodata - 4
00000000001d      000f00000004 R_X86_64_PLT32  0000000000000000 printf - 4
000000000035      000300000002 R_X86_64_PC32   0000000000000000 .data + 0
00000000003b      000400000002 R_X86_64_PC32   0000000000000000 .bss - 4
00000000004e      000d00000004 R_X86_64_PLT32  0000000000000000 func1 - 4

# i can not understand this part, but just show here, maybe after sometimes, i will back to learn it
Relocation section '.rela.eh_frame' at offset 0x3b0 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000020  000200000002 R_X86_64_PC32     0000000000000000 .text + 0
000000000040  000200000002 R_X86_64_PC32     0000000000000000 .text + 24
```

symbol table

- what is symbol
  - **func name and var name**
- **可执行文件中的一部分**
  - **某section**
- 内核的符号表可以在这里看

    - `/proc/kallsyms`
- all based on symbol, so how to manager symbol is so important
- for var and func, the value of symbol is **address**
- 关于以下内容的一些说明
    - 符号种类有很多：
        - 定义在本目标文件中的全局符号，可以被其他目标文件引用
        - 定义在其它目标文件中全局符号，本文件引用了它, **like prinf here**, printf is a symbol here
        - other type
- `nm main.o` 也可以用于查看一个 `.o` 符号表
- 一些说明
    - Num，表示 section header table 中数组符号的下标，0-16，共计17项
    - value：符号值
        - for var and func, it is the adress, actually is offset
        - 不是common类型的，表示在对应段中的偏移，主要还是地址吧
        - common类型的，是符号的对齐属性
    - size: 符号大小：
        - 如果是变量，是变量类型的大小，即sizeof(int)
        - 函数的话，**是指令的字节数**
        - 0表示未知，或就是0
    - bind，绑定信息：
        - local局部符号
        - global全局符号
        - weak or strong symbol
    - Ndx，符号所在段：
        - 如果是一个数字下标，说明当前符号定义在当前的目标文件中，the index of section header table
        - 还有一些未定义在当前的目标文件中，或特殊的类型：
            - ABS，比如文件名
            - UNDEF，本文件仅引用，但是**未定义**，定义在外部模块
            - **COMMON，一般来说，未初始化的全局变量就在这里**
- **所以符号表是可执行文件的一部分**

```
double_d@dd:~/QuickTest/comp$ readelf -s main.o

Symbol table '.symtab' contains 17 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND  # 未知类型符号，第一个永远是这个样子的，忽略掉吧
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS no.c  # 文件名，符号所在段是ABS
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1  # .text
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3  # .data
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4  # .bss
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5  # .rodata
     6: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    3 static_var.1919  # .data，偏移4字节是它，int占4字节，so size=4
     7: 0000000000000000     4 OBJECT  LOCAL  DEFAULT    4 static_var2.1920  # .bss，offset is 0 and sizeof(int) is 4
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    7  # .note.GNU-stack
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT    8  # .eh_frame
    10: 0000000000000000     0 SECTION LOCAL  DEFAULT    6  # .comment
    11: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 global_init_var  # .data，偏移0字节，int占4字节，size=4
    12: 0000000000000004     4 OBJECT  GLOBAL DEFAULT  COM global_uninit_var  # value=4是该值的对齐属性 and sizeof(int)=4
    13: 0000000000000000    36 FUNC    GLOBAL DEFAULT    1 func1  # ndx is 1, in .text, 36是func1指令字节数，value是相对于.text的偏移，位于代码段的he
    14: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSET_TABLE_
    15: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf  # 未定义的符号，是要去外部重定位的
    16: 0000000000000024    51 FUNC    GLOBAL DEFAULT    1 main  # as same as func1, main if after func1, and the size of func1 is 24, so the val

double_d@dd:~/QuickTest/comp$ nm main.o
0000000000000000 T func1  # .text
0000000000000000 D global_init_var  # .data
                 U _GLOBAL_OFFSET_TABLE_  # undefine
0000000000000004 C global_uninit_var # common
0000000000000024 T main  # .text
                 U printf  # undefine
0000000000000004 d static_var.1919  # .data
0000000000000000 b static_var2.1920  # .bss
```

关于调试:

- `gcc -g` 可以增加调试信息，otherwise，`strip main.o` 可以去掉 debug info
- 关键在于 `file filename` 的时候有如下信息
  - **with debug_info**

可执行文件的加载，./main 的时候发生了什么呢

- shell 认为 main 是一个可执行的目标文件，通过调用 exec（**是一个系统调用，内存中的常驻的，kernel中的代码**）来加载其到内存
- 程序由磁盘复制到内存的过程称为 **加载**
  - 一定是有文件系统以及IO参与的
- 加载的概述（**静态的简单描述**）：
  - 父进程 shell fork 一个子进程，子进程是父进程的一个复制
  - **子进程被调度后，执行系统调用exec**：
    - **加载elf header and program header等一些header数据**
      - 仅此而已，**不会在这个阶段就去加载具体的代码段数据段等内容**
    - 加载器 delete 进程当前（继承自父进程）的vm映射关系，即删除代码段与数据段 （内核态）
    - 然后建立新的映射关系，堆栈等初始化为0（匿名映射都省了），**.text等映射到对应的磁盘文件上** （内核态）
    - exec执行完毕后，返回到用户空间，这个地方有两种情况
      1. 可执行文件是静态的
         - 返回用户空间侯，rip 指向可执行文件的 `e_entry`
         - 加载器跳转到 `_start` （该进程的上下文中，rip指向这里）
           - elf文件 header 中是保存有这样的一个项的 `Entry point address: 0x1050` ，这个就是 `_start` 的地址
      2. 可执行文件是 dynamic link 的
         - rip 指向 `.interp` , the addr of linker and finish the map of ld, libc, etc
         - linker:
           - dynamic linker先自举
           - link create got, including main, ld, libc; 所有的 `.so` 在这一步都要被映射
           - 遍历 main, ld.so, libc.so 等的重定位表与符号表，修正一些它们对应的GOT/PLT
           - 然后第一次需要的时候，就会真正的去修正地址了
         - linker完成之后，rip也会指向 `_start`
  - 再强调一下：
    - 这个过程中，除了header info (elf header and program header), 没有**任何磁盘到内存的数据复制**
    - 直到CPU第一次访问到这片内存的代码的时候，触发缺页中断，才将数据由磁盘读到内存中
- program header的解读
  - type：LOAD类型是需要被加载到内存中的
  - offset：elf文件中的偏移
  - VirtAddr：VM中的位置
  - PhysAddr：一般与VM是一致的
  - FileSize：二进制elf文件中的大小
  - MemSize：内存中的大小
  - Flags：读写执行权限等等
  - Align：对齐要求

```
double_d@dd:~/QuickTest/comp$ readelf -l main

Elf file type is DYN (Shared object file)  # gcc自动带了位置无关代码的优化参数
Entry point 0x1050  # 可执行程序的入口点，是_start的入口点
There are 11 program headers, starting at offset 64

Program Headers:
  Type          Offset             VirtAddr           PhysAddr           FileSiz            MemSiz              Flags  Align
  PHDR          0x0000000000000040 0x0000000000000040 0x0000000000000040 0x0000000000000268 0x0000000000000268  R      0x8
  INTERP        0x00000000000002a8 0x00000000000002a8 0x00000000000002a8 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD          0x0000000000000000 0x0000000000000000 0x0000000000000000 0x0000000000000560 0x0000000000000560  R      0x1000
  LOAD          0x0000000000001000 0x0000000000001000 0x0000000000001000 0x00000000000001bd 0x00000000000001bd  R E    0x1000
  LOAD          0x0000000000002000 0x0000000000002000 0x0000000000002000 0x0000000000000158 0x0000000000000158  R      0x1000
  LOAD          0x0000000000002de8 0x0000000000003de8 0x0000000000003de8 0x0000000000000248 0x0000000000000250  RW     0x1000
  DYNAMIC       0x0000000000002df8 0x0000000000003df8 0x0000000000003df8 0x00000000000001e0 0x00000000000001e0  RW     0x8
  NOTE          0x00000000000002c4 0x00000000000002c4 0x00000000000002c4 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_EH_FRAME 0x0000000000002010 0x0000000000002010 0x0000000000002010 0x000000000000003c 0x000000000000003c  R      0x4
  GNU_STACK     0x0000000000000000 0x0000000000000000 0x0000000000000000 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO     0x0000000000002de8 0x0000000000003de8 0x0000000000003de8 0x0000000000000218 0x0000000000000218  R      0x1

 # link view to load view
 # 减少碎片
 # 对于相同装载属性的section将整合为一个segment
 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt
   03     .init .plt .plt.got .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
   05     .init_array .fini_array .dynamic .got .got.plt .data .bss
   06     .dynamic
   07     .note.gnu.build-id .note.ABI-tag
   08     .eh_frame_hdr
   09
   10     .init_array .fini_array .dynamic .got
```

先学习一下 .so 的格式中 dynamic 的这一部分

- gcc -fPIC -shared -o Lib.so Lib.c
  - 生成地址无关代码 -fPIC
  - -shared, output .so
- 重要的就是符号表与relocate1 table

```
#.dynamic对于动态链接而言，is header
double_d@dd:~/QuickTest$ readelf -d Lib.so

Dynamic section at offset 0x2e20 contains 24 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x1000  # .init offset
 0x000000000000000d (FINI)               0x1118  # .fini offset
 0x0000000000000019 (INIT_ARRAY)         0x3e10
 0x000000000000001b (INIT_ARRAYSZ)       8 (bytes)
 0x000000000000001a (FINI_ARRAY)         0x3e18
 0x000000000000001c (FINI_ARRAYSZ)       8 (bytes)
 0x000000006ffffef5 (GNU_HASH)           0x260  # .gnu.hash offset
 0x0000000000000005 (STRTAB)             0x330
 0x0000000000000006 (SYMTAB)             0x288
 0x000000000000000a (STRSZ)              119 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000003 (PLTGOT)             0x4000
 0x0000000000000002 (PLTRELSZ)           24 (bytes)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0x480
 0x0000000000000007 (RELA)               0x3d8
 0x0000000000000008 (RELASZ)             168 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000006ffffffe (VERNEED)            0x3b8
 0x000000006fffffff (VERNEEDNUM)         1
 0x000000006ffffff0 (VERSYM)             0x3a8
 0x000000006ffffff9 (RELACOUNT)          3
 0x0000000000000000 (NULL)               0x0

# 看一眼动态符号表
# symbol符号表包含静态符号以及动态符号
Symbol table '.dynsym' contains 4 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__

# 看一眼重定位表
double_d@dd:~/QuickTest$ readelf -r main

Relocation section '.rela.dyn' at offset 0x3f0 contains 2 entries:  # for var
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000403ff0  000200000006 R_X86_64_GLOB_DAT 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
000000403ff8  000300000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0

Relocation section '.rela.plt' at offset 0x420 contains 1 entry:  # for func
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000404018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
```

动态链接

- 符号重定位的过程推迟到 load time or run time
- some tips:
  - PLT: procedure linkage table
    - 延迟绑定的核心内容
    - 多一次间接寻址，但是只有第一次这样搞，省内存，后面的开销很小
  - GOT: global offset table
    - 假如 hello world，其中 printf 是一个符号，其地址相对于 libc.so 加载到内存中的起始地址是固定的
    - 如果两个程序中都有对 printf 的调用，但是对于两个进程呢， `libc.so` 的加载位置显然是不同的，但是两个进程都需要 printf 的符号地址，VM 中，这两个 printf 的符号地址是不一样的。只能存放在 private 区了，即 `.data` 区，两个进程有各自的副本
    - GOT表就是位于 private `.data` 区的，link完成之后，存放了 printf 在VM中的地址
  - `ld.so` 是链接器，**它才是一个可执行文件的第一入口**，**自举**，本身也是以 `.so` 存在的
  - 地址无关代码（PIC）：数据与指令分离；指令不变；数据在不同进程中，可以有自己的副本，public ins，private data
    - 核心：与地址无关的内容放到 `.data` 中
    - 每一个 `.so` 都有自己的 `.text, .data, .got. plt` 等等
    - 而且一个 `.so` 内部是固定的，**相对位置是固定的**
  - 共享对象需要重定位的原因是因为**导入符号**的存在，例如printf；但是动态链接的时候，导入符号的地址在第一次使用的时候才确定（延迟绑定）
  - 做插件plug-in，dlopen，运行时加载的过程，reload可能就是调用dlopen，可能还有一个卸载的过程

```c
//main.c最简单的代码用例，用于说明动态链接的原理
#include<stdio.h>
int main()
{
        printf("hello world\n");
        return 0;
}
```

```
# 首先 objdump -s main 观察一下重要的几个 section 的位置以及二进制代码表示
section   VMA         Contents of section
.plt      0x401020    ff35e22f 0000ff25 e42f0000 0f1f4000  .5./...%./....@.
          0x401030    ff25e22f 00006800 000000e9 e0ffffff  .%./..h.........
.got      0x403ff0    00000000 00000000 00000000 00000000  ................
.got.plt  0x404000    203e4000 00000000 00000000 00000000   >@.............
          0x404010    00000000 00000000 36104000 00000000  ........6.@.....

# 展示反汇编的大部分内容，非可执行部分尽管反汇编出了指令，但是真的不是指令，不能当指令去看的说
double_d@dd:~/QuickTest$ objdump -D main

main:     file format elf64-x86-64

.interp:
00000000004002a8 <.interp>  # the location of loader image
.note.gnu.build-id
00000000004002c4 <.note.gnu.build-id>
note.ABI-tag
00000000004002e8 <.note.ABI-tag>
.gnu.hash
0000000000400308 <.gnu.hash>  # a k-v of symbol, more quick
.dynsym
0000000000400328 <.dynsym>  # dynamic symbol
.dynstr
0000000000400388 <.dynstr>  # dynamic string
.gnu.version
00000000004003c6 <.gnu.version>
.gnu.version_r
00000000004003d0 <.gnu.version_r>
.rela.dyn
00000000004003f0 <.rela.dyn>  # relocate data, 重定位表中存放的是需要被重定位的符号
.rela.plt
0000000000400420 <.rela.plt>  # relocate func
# reserve some key real execute part
.init
0000000000401000 <_init>

# procedure link table, 这就是lazy bind的关键部分
# 本例中, there are two entry in plt, every entry is 16 bytes
# every entry for a func
# the first entry in plt is to call dynamic linker
# 尽管多了一次间接引用，但是省内存，而且第二次就只有jump的开销了
.plt:
0000000000401020 <.plt>:  # call dynamic linker
  401020:       ff 35 e2 2f 00 00       pushq  0x2fe2(%rip)        # 404008 <_GLOBAL_OFFSET_TABLE_+0x8>
  401026:       ff 25 e4 2f 00 00       jmpq   *0x2fe4(%rip)        # 404010 <_GLOBAL_OFFSET_TABLE_+0x10> dl_runtime_resolve, 这个是linker中的函数
  40102c:       0f 1f 40 00             nopl   0x0(%rax)

# 404018 hold addr of 401036
0000000000401030 <puts@plt>:  # put is printf
  401030:       ff 25 e2 2f 00 00       jmpq   *0x2fe2(%rip)        # 404018 <puts@GLIBC_2.2.5>
  401036:       68 00 00 00 00          pushq  $0x0
  40103b:       e9 e0 ff ff ff          jmpq   401020 <.plt>  # goto first entry in plt to call dynamic linker

.text
0000000000401040 <_start>
0000000000401070 <_dl_relocate_static_pie>
0000000000401080 <deregister_tm_clones>
00000000004010b0 <register_tm_clones>
00000000004010f0 <__do_global_dtors_aux>
0000000000401120 <frame_dummy>

0000000000401122 <main>:
  401122:       55                      push   %rbp
  401123:       48 89 e5                mov    %rsp,%rbp
  401126:       48 8d 3d d7 0e 00 00    lea    0xed7(%rip),%rdi        # 402004 <_IO_stdin_used+0x4>
  40112d:       e8 fe fe ff ff          callq  401030 <puts@plt>  # 直接寻址到 PLT 中的对应项
  401132:       b8 00 00 00 00          mov    $0x0,%eax
  401137:       5d                      pop    %rbp
  401138:       c3                      retq
  401139:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)

0000000000401140 <__libc_csu_init>
00000000004011a0 <__libc_csu_fini>
.fini
00000000004011a4 <_fini>
```

```
.rodata
0000000000402000 <_IO_stdin_used>
.eh_frame_hdr:
0000000000402010 <__GNU_EH_FRAME_HDR>
.eh_frame
0000000000402050 <__FRAME_END__-0xfc>
000000000040214c <__FRAME_END__>
.init_array
0000000000403e10 <__frame_dummy_init_array_entry>
.fini_array
0000000000403e18 <__do_global_dtors_aux_fini_array_entry>
.dynamic
# 专用于动态链接，类似与header of elf
0000000000403e20 <_DYNAMIC>

# global offset table
# elf将GOT拆分成了两个表，.got hold the addr of global var，.got.plt hold the addr of func
.got:
# 00000000 00000000 00000000 00000000  16个字节
0000000000403ff0 <.got>

.got.plt:
# 203e4000 00000000 00000000 00000000  16 bytes
# 00000000 00000000 36104000 00000000  16 bytes
# there are some reserved entry in the start of .got
0000000000404000 <_GLOBAL_OFFSET_TABLE_>:
  404000: 20 3e 40 00 00 00 00 00  # addr of .dynamic
  404008: 00 00 00 00 00 00 00 00
  404010: 00 00 00 00 00 00 00 00  # 需要dynamic linker自举完才有内容看来
  404018  36 10 40 00 00 00 00 00  # printf的位置，目前prinft是调用了plt中的linker的位置，再由linker去确定下位置来printf真正的位置，再一次调用在got中就能

.data:
0000000000404020 <__data_start>
0000000000404028 <__dso_handle>
.bss
0000000000404030 <__bss_start>
.comment
0000000000000000 <.comment>
```

2. 寄存器

X86部分寄存器以及其使用惯例

- 调用者保存寄存器（**你随便用，我兜着**）：P调用Q时，Q可以覆盖这些寄存器，不会破坏任何P所需要的数据
- 被调用者保存寄存器（**我不管，你自己保证原样给还给我**）：Q必须在覆盖这些寄存器的值之前，先把它们保存到栈中，并在返回前恢复它们
- `-0x14(%rbp)`，内存间接寻址，表示rbp寄存器中的值减去**0x14**的内存位置
- 以mov指令为例：`mov A B`
  - 自己编译出来的采用的是AT&T（跨平台）汇编语法，上述指令表示将A中的值移到B中
  - Intel汇编语法，即Intel指令集手册，就是将B中的数移到A
  - R/M/I
    - R-寄存器
    - M-内存地址
    - I-立即数
- rax always hold the return value of a func

0-63	0-31	0-15	8-15	0-7	使用惯例
%rbp	%ebp	%bp	无	%bpl	被调用者保存，base pointer
%rsp	%esp	%sp	无	%spl	栈顶指针，stack pointer，当前进程的栈顶，函数调用以及变量定义均会导致栈指针减小
%rip	%eip	%ip			指令指针寄存器，指向下一个指令的地址
%rax	%eax	%ax	%ah	%al	accumulator即累加寄存器，用于保存函数的返回值
%rbx	%ebx	%bx	%bh	%bl	被调用者保存
%rdi	%edi	%di	无	%dil	第1个参数 or d means destination
%rsi	%esi	%si	无	%sil	第2个参数 or s means source

0-63	0-31	0-15	8-15	0-7	使用惯例
%rdx	%edx	%dx	%dh	%dl	第3个参数
%rcx	%ecx	%cx	%ch	%cl	第4个参数
%r8	%r8d	%r8w	无	%r8b	第5个参数
%r9	%r9d	%r9w	无	%r9b	第6个参数
%r10	%r10d	%r10w	无	%r10b	调用者保存
%r11	%r11d	%r11w	无	%r11b	调用者保存
%r13	%r13d	%r13w	无	%r13b	被调用者保存
%r14	%r14d	%r14w	无	%r14b	被调用者保存
%r15	%r15d	%r15w	无	%r15b	被调用者保存

一些指令的典型用法

- push %rbp: rbp寄存器中的值压栈
  - sub $0x8,%rsp
  - mov %rbp,(%rsp)
- pop %rbp: 将栈顶元素的值弹出到指定寄存器，恢复rbp寄存器，之前上一个栈的寄存器是被压到这里来的
  - mov (%rsp), %rbp
  - add $0x8, %rsp
- call: 将下一条指令压栈，并跳转到目标地址
  - push
  - jump
- ret: 把之前压栈的那条指令pop出来并且跳过去
  - pop
  - jump
- leave: 速度放弃当前的函数调用栈，快速恢复到上一个栈帧
  - mov %rbp, %rsp
  - pop %rbp

```c
#include<stdio.h>
#include<unistd.h>

int bar(int c, int d){
    int e = c + d;
    return e;
}

int foo(int a, int b){
    return bar(a, b);
}

int main(int argc, char *argv[]){
        int a=2, b=5, rtn;
        rtn = foo(2, 5);
        printf("rtn is %d\n", rtn);
    return 0;
}

// 程序执行部分的汇编代码以及与源码的对应关系
#include<stdio.h>
#include<unistd.h>

int bar(int c, int d){
    1135:       55                      push   %rbp  //将rbp中的值压栈，保存上一个base pointer的地址
    1136:       48 89 e5                mov    %rsp,%rbp  //rsp中的值给到rbp，这个是新的函数调用栈的起点
    1139:       89 7d ec                mov    %edi,-0x14(%rbp)  //edi中存放的是bar函数的第一个参数，即将c入栈，将edi寄存器中的值写入到%rbp寄存器-0x14的
    113c:       89 75 e8                mov    %esi,-0x18(%rbp)  //esi存放的是第二个参数，与上同理
    int e = c + d;
    113f:       8b 55 ec                mov    -0x14(%rbp),%edx  //将参数c赋值给寄存器edx，将内存位置%rbp-0x14的值写入%edx
    1142:       8b 45 e8                mov    -0x18(%rbp),%eax  //将参数d赋值给寄存器eax，同上
    1145:       01 d0                   add    %edx,%eax  //加法计算，结果保存在eax中
    1147:       89 45 fc                mov    %eax,-0x4(%rbp)  //eax的值保存到rbp-4的位置
    return e;
    114a:       8b 45 fc                mov    -0x4(%rbp),%eax  //rbp-4位置上的数据写入到eax中，作为返回值
}
    114d:       5d                      pop    %rbp  //push的逆动作
    //这个函数调用栈实际上是没有生长的，只有最开始的push有增长8字节
    //没有入栈的操作，全部在寄存器完成，就可以很快
    //当前的栈顶就是上一个函数调用栈的rbp的值，即将当前的栈顶元素写入rbp寄存器中
    //即恢复上一个调用栈，返回值保存在寄存器rax中
    114e:       c3                      retq
    //当前栈顶所保存的是call bar之后的下一条指令的地址
    //ret指令将该栈顶元素出栈并赋值给rip，即回到上一个函数的栈帧中运行

000000000000114f <foo>:

int foo(int a, int b){
    114f:       55                      push   %rbp
    1150:       48 89 e5                mov    %rsp,%rbp
    1153:       48 83 ec 08             sub    $0x8,%rsp  //预分配8个字节
    1157:       89 7d fc                mov    %edi,-0x4(%rbp)
    115a:       89 75 f8                mov    %esi,-0x8(%rbp)
    return bar(a, b);
    115d:       8b 55 f8                mov    -0x8(%rbp),%edx
    1160:       8b 45 fc                mov    -0x4(%rbp),%eax
    1163:       89 d6                   mov    %edx,%esi
    1165:       89 c7                   mov    %eax,%edi
    1167:       e8 c9 ff ff ff          callq  1135 <bar>
}
    116c:       c9                      leaveq  //快速放弃当前的函数调用栈，直接恢复到push rbp之前的状态
    116d:       c3                      retq  //pop and jump

000000000000116e <main>:

int main(int argc, char *argv[]){
    116e:       55                      push   %rbp
    116f:       48 89 e5                mov    %rsp,%rbp
    1172:       48 83 ec 20             sub    $0x20,%rsp  //在栈上预分配32个字节先
    1176:       89 7d ec                mov    %edi,-0x14(%rbp)
    1179:       48 89 75 e0             mov    %rsi,-0x20(%rbp)
        int a=2, b=5, rtn;
    117d:       c7 45 fc 02 00 00 00    movl   $0x2,-0x4(%rbp) //立即数2入栈
    1184:       c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%rbp)  //立即数5如栈
        rtn = foo(2, 5);
    118b:       be 05 00 00 00          mov    $0x5,%esi  //为函数调用准备参数
```

```
1190:       bf 02 00 00 00          mov    $0x2,%edi  //为函数调用准备参数
1195:       e8 b5 ff ff ff          callq  114f <foo>  //将下一条指令（即119a）的地址入栈，并跳转到新函数的起始位置
119a:       89 45 f4                mov    %eax,-0xc(%rbp)
    printf("rtn is %d\n", rtn);
119d:       8b 45 f4                mov    -0xc(%rbp),%eax
11a0:       89 c6                   mov    %eax,%esi  //为printf函数准备参数
11a2:       48 8d 3d 5b 0e 00 00    lea    0xe5b(%rip),%rdi        // 2004 <_IO_stdin_used+0x4>  //为printf函数准备参数
11a9:       b8 00 00 00 00          mov    $0x0,%eax
11ae:       e8 7d fe ff ff          callq  1030 <printf@plt>
return 0;
11b3:       b8 00 00 00 00          mov    $0x0,%eax
11b8:       c9                      leaveq
11b9:       c3                      retq
11ba:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
```