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

用代码说明汇编文件的组成部分

汇编文件，使用 ELF 文件格式，主要包括数据段、代码段等。
数据段，包括变量、常量、GOT 代理等。
代码段，包括函数、PLT 代理等。

编写代码： part.s

```
# ----------------------------
.data       # 可读可写数据段

.global num_int64   # 全局的符号

    .align 16    # 对齐
num_int64:  # 单值变量
    .quad 2000  # 一个 64 位整数

num_int32_array: # 数组变量
    .long 100   # 数组元素
    .long 200   # 数组元素

# ----------------------------
.section .rodata # 只读数据段

str_int64:  # 字符串。打印变量的地址和值
    .string "num_int64 addr = %#llX    value = %lld \n"

str_ret:    # 字符串。打印函数的地址和返回值
    .string "func_add  addr = %#llX   return = %lld \n"

# ----------------------------
.text       # 代码段

.global main        # 全局的符号
.local  func_add    # 本地的符号

    .align 1   # 对齐
main:       # 定义 main 函数
    pushq %rbp          # 入栈。把 rbp 的值压入栈顶
    movq %rsp, %rbp     # 把 rsp 的值赋给 rbp
    subq $64, %rsp      # 栈帧扩容。

    movq $str_int64, %rdi       # 入参 1。绝对寻址
    leaq num_int64(%rip), %rsi  # 入参 2。相对寻址
    movq num_int64(%rip), %rdx  # 入参 3。相对寻址
```

```
    callq printf                # 调用函数

    movq num_int64(%rip), %rdi  # 入参 1。相对寻址
    callq func_add              # 调用函数
    movq %rax, -8(%rbp)         # 函数的返回值。

    leaq str_ret(%rip), %rdi    # 入参 1。相对寻址
    movq $func_add, %rsi        # 入参 2。绝对寻址
    movq -8(%rbp), %rdx         # 入参 3。函数的返回值
    callq printf                # 调用函数

    addq $64, %rsp      # 栈帧缩容。
    popq %rbp           # 出栈。把栈顶的值赋给 rbp
    retq                # 退出函数

    .align 8   # 对齐
func_add:   # 定义函数
    pushq %rbp
    movq %rsp, %rbp

    movq %rdi, %rcx     # 取入参 1
    addq $46, %rcx      # 加法。增加值
    movq %rcx, %rax     # 返回值

    popq %rbp
    retq
```

编译代码：

```
gcc part.s -o part

readelf -a part > part.elf.txt
objdump -D part > part.dump.txt
```

运行代码：

```
[root@local part]# ./part
num_int64 addr = 0X601040    value = 2000
func_add  addr = 0X400580   return = 2046
```

分析结果：
汇编文件 part.s，编译为可执行文件 part。使用 readelf 查看 part 的 ELF 信息。使用 objdump 查看 part 的 dump 信息。
函数 func_add，给入参加上 46，再返回。入参为 2000，返回值为 2046，说明函数功能正常。
查看文件 part.s、part.elf.txt、part.dump.txt。这里截取主要的部分。

符号表

```
Symbol table '.symtab' contains 68 entries:
    37: 0000000000601048     0 NOTYPE  LOCAL  DEFAULT   24 num_int32_array
    38: 0000000000400620     0 NOTYPE  LOCAL  DEFAULT   15 str_int64
    39: 0000000000400649     0 NOTYPE  LOCAL  DEFAULT   15 str_ret
```

```
    40: 0000000000400580     0 NOTYPE  LOCAL  DEFAULT   13 func_add
    53: 0000000000601040     0 NOTYPE  GLOBAL DEFAULT   24 num_int64
    55: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@@GLIBC_2.2.5
    65: 0000000000400530     0 NOTYPE  GLOBAL DEFAULT   13 main
```

数据段

```
Disassembly of section .data:
0000000000601040 <num_int64>:
0000000000601048 <num_int32_array>:

Disassembly of section .got:
0000000000600ff8 <.got>:

Disassembly of section .got.plt:
0000000000601000 <_GLOBAL_OFFSET_TABLE_>:

Disassembly of section .rodata:
0000000000400620 <str_int64>:
0000000000400649 <str_ret>:
```

代码段

```
Disassembly of section .text:
0000000000400530 <main>:
0000000000400580 <func_add>:

Disassembly of section .plt:
0000000000400400 <.plt>:
0000000000400410 <printf@plt>:
```

自定义汇编文件的各个部分

定义段 section
格式为 `.section section_name` 。
特殊的段.data、.text,可以省略前缀.section。

```
.data
.section .rodata
.text
```

定义单值变量
格式为

```
param_name :
    param_type param_value
```

比如，定义一个 64 位整数的变量。

```
num_int64:  # 变量
    .quad 2000        # 一个 64 位整数
```

定义数组变量

格式为

```
param_name :
    param_type param_value
    param_type param_value
```

或

```
param_name :
    param_type param_value , param_value
```

比如，定义 1 个数组，包含 2 个 32 位整数。

```
num_int32_array: # 数组变量
    .long 100   # 数组元素
    .long 200   # 数组元素
```

定义字节对齐

格式为 `.align num` 。必须为 2 的 N 次方。

比如，在符号 num_int64 的前面加了对齐 16 字节。

```
    .align 16    # 对齐
num_int64:  # 单值变量
```

查看 dump 文件，符号 num_int64 的地址为 0000000000601040，是 16 的倍数。

```
0000000000601040 <num_int64>:
```

定义函数

格式为

```
func_name :
    op_stack
    some_code
    op_stack
    retq
```

比如，定义一个函数，把一个数加上某值，再返回。

```
func_add:   # 定义一个函数
    pushq %rbp
    movq %rsp, %rbp

    movq %rdi, %rcx     # 取入参 1
    addq $46, %rcx      # 加法。增加值
    movq %rcx, %rax     # 返回值

    popq %rbp
    retq
```

定义作用范围

```
.global symbol_name  # 全局的符号
.local  symbol_name  # 本地的符号。默认。
```

比如，符号 num_int64 使用.global 修饰 `.global num_int64` 。符号 num_int32_array 没有.global 修饰。

查看 ELF 文件，符号 num_int64 有 GLOBAL 标记，符号 num_int32_array 有 LOCAL 标记 。

```
    53: 0000000000601040     0 NOTYPE  GLOBAL DEFAULT   24 num_int64
    37: 0000000000601048     0 NOTYPE  LOCAL  DEFAULT   24 num_int32_array
```

指定 main 函数

格式为 `.global main` 。

main 函数，是程序运行的入口。
main 函数，从函数的格式来看，是一个普通的函数，和普通函数的主要区别是函数名称为 main。

调用函数
格式为

```
    put_func_param
    callq func_name
    get_func_return
```

比如，调用函数 func_add

```
    movq num_int64(%rip), %rdi  # 入参 1。相对寻址
    callq func_add              # 调用函数
    movq %rax, -8(%rbp)         # 函数的返回值。
```

绝对寻址与相对寻址

功能	汇编指令	dump 文件
绝对寻址，取地址	movq $str_int64, %rdi	400538: mov $0x400620,%rdi
相对寻址，取地址	leaq num_int64(%rip), %rsi	40053f: lea 0x200afa(%rip),%rsi # 601040 <num_int64>
相对寻址，取值	movq num_int64(%rip), %rdx	400546: mov 0x200af3(%rip),%rdx # 601040 <num_int64>
调用函数	callq printf	40054d: callq 400410 <printf@plt>

绝对寻址，使用$str_int64，被替换为地址$0x400620，对应符号 str_int64 的地址 0000000000400620。

```
    38: 0000000000400620     0 NOTYPE  LOCAL  DEFAULT   15 str_int64
```

相对寻址，使用 num_int64(%rip)，被替换为相对于 rip 的偏移，0x200af1(%rip)、0x200aea(%rip)。

```
    53: 0000000000601040     0 NOTYPE  GLOBAL DEFAULT   24 num_int64
```

计算过程：RIP + 偏移 = 符号的绝对地址。

```
    uint64_t tmp2 = 0x400546 + 0x200afa;
    uint64_t tmp3 = 0x40054d + 0x200af3;
    printf("  %#llX  %#llX  \n", tmp2, tmp3);
```

输出为 `0X601040  0X601040` ，对应符号 num_int64 的地址 0000000000601040。