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

返回值的规则

返回值有如下规则：
返回 1 个整数，使用 rax 寄存器。
返回 1 个浮点数，使用 xmm0 寄存器。
返回 1 个指针，使用 rax 寄存器。指针是特殊的整数。
返回 1 个 struct，使用寄存器或函数栈。依赖于 struct 的复杂度。
返回指针，表示引用传递。
返回非指针，表示值传递。
返回值支持自定义的寄存器、函数栈。

返回值的个数，可以有 0 个、1 个、多个。
0 个或 1 个返回值，是大多数编程语言支持的方式。
C、C++、Java 等，不支持多个返回值。
Go、Python 等，支持多个返回值。
从汇编语言角度，可以支持多个返回值。

用 C 和汇编分析简单的返回值

编写代码： return.c

```
// 函数没有返回值
void func_no_return()
{
}

// 函数返回 1 个整数
```

```
int func_1_return_int()
{
    return 666;
}

// 函数返回 1 个浮点数
double func_1_return_double()
{
    return 111.22D;
}
```

编译代码：

```
gcc return.c -S -o return.s
```

分析结果：

查看汇编文件 return.s，这里截取与返回值有关的部分。

场景	C 语言代码	汇编代码	分析

函数没有返回值	`void func_no_return()`
	`{`
	`}`
		`func_no_return:`	没有相关指令。

函数返回 1 个整数	`int func_1_return_int()`
	`{`
	`  return 666;`
	`}`
		`func_1_return_int:`
		`  movl  $666, %eax`
			使用寄存器 eax 传递返回值。

函数返回 1 个浮点数	`double func_1_return_double()`
	`{`
	`  return 111.22D;`
	`}`
		`func_1_return_double:`
		`  movabsq $4637526828664309678, %rax`
		`  movq  %rax, -8(%rbp)`
		`  movsd  -8(%rbp), %xmm0`
			使用寄存器 xmm0 传递返回值。
			$4637526828664309678 表示用 8 个字节的整数表示 double。

用 C 和汇编分析复杂的返回值

编写代码： return_hard.c

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

// 定义 struct
typedef struct
{
    long long age; // 8 个字节
    double height; // 8 个字节
} cat_t;

// 一个全局变量 struct
cat_t Tom = {.age = 33, .height = 55};

// 函数返回 1 个全局变量的指针
cat_t *func_1_return_pionter()
{
    return &Tom;
}

// 函数返回 1 个栈内变量的指针
void *func_1_return_pionter2()
{
    // 仅供测试，真实场景不要把栈内地址返回到外部
    int buf;
    return &buf;
}
```

```
// 函数返回 1 个外部 struct 的拷贝
cat_t func_1_return_struct()
{
    return Tom;
}

int main()
{
    // 分别查看 3 种返回值
    cat_t *ptr1 = func_1_return_pionter();
    void *ptr2 = func_1_return_pionter2();
    cat_t cat = func_1_return_struct();

    printf("函数返回 1 个全局变量的指针     = %p \n", ptr1);
    printf("函数返回 1 个栈内变量的指针     = %p \n", ptr2);
    printf("函数返回 1 个外部 struct 的拷贝   = %p \n", &cat);

    // 暂停进程，查看内存布局
    sleep(9999999);
    return 0;
}
```

编译代码：

```
gcc return_hard.c  -o return_hard
gcc return_hard.c -S -o return_hard.s
```

运行代码：

```
[root@192 func]# ./return_hard
函数返回 1 个全局变量的指针     = 0x601050
函数返回 1 个栈内变量的指针     = 0x7ffc9af6504c
函数返回 1 个外部 struct 的拷贝   = 0x7ffc9af65070
```

查看内存布局：

```
[root@192 func]# ps aux | grep return
root      10538  0.0  0.0   4216   352 pts/9    S+   02:54   0:00 ./return_hard
root      10620  0.0  0.0 112812   980 pts/0    S+   02:54   0:00 grep --color=auto return
[root@192 func]# cat /proc/10538/maps
00400000-00401000 r-xp 00000000 08:03 51425533                           /root/code/x86-asm/common2/func/return_hard
00600000-00601000 r--p 00000000 08:03 51425533                           /root/code/x86-asm/common2/func/return_hard
00601000-00602000 rw-p 00001000 08:03 51425533                           /root/code/x86-asm/common2/func/return_hard
7fe53282c000-7fe5329f0000 r-xp 00000000 08:03 15928                      /usr/lib64/libc-2.17.so
7fe5329f0000-7fe532bef000 ---p 001c4000 08:03 15928                      /usr/lib64/libc-2.17.so
7fe532bef000-7fe532bf3000 r--p 001c3000 08:03 15928                      /usr/lib64/libc-2.17.so
7fe532bf3000-7fe532bf5000 rw-p 001c7000 08:03 15928                      /usr/lib64/libc-2.17.so
7fe532bf5000-7fe532bfa000 rw-p 00000000 00:00 0
7fe532bfa000-7fe532c1c000 r-xp 00000000 08:03 611075                     /usr/lib64/ld-2.17.so
7fe532e11000-7fe532e14000 rw-p 00000000 00:00 0
7fe532e19000-7fe532e1b000 rw-p 00000000 00:00 0
7fe532e1b000-7fe532e1c000 r--p 00021000 08:03 611075                     /usr/lib64/ld-2.17.so
7fe532e1c000-7fe532e1d000 rw-p 00022000 08:03 611075                     /usr/lib64/ld-2.17.so
7fe532e1d000-7fe532e1e000 rw-p 00000000 00:00 0
7ffd51c48000-7ffd51c69000 rw-p 00000000 00:00 0                          [stack]
7ffd51d6f000-7ffd51d71000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

分析结果：

全局变量 cat_t Tom 的地址为 0x601050，在内存区间 `00601000-00602000 rw-p` ，此区间表示程序的可读可写数据段。

栈内变量的地址为 0x7ffc9af6504c，在内存区间 `[stack]` 下方，属于栈空间。

外部 struct 的拷贝的地址为 0x7ffc9af65070，在内存区间 `[stack]` 下方，属于栈空间。

查看汇编文件 return_hard.s，这里截取与返回值有关的部分。

如果把 struct 定义的更加复杂，然后用函数返回 struct，会发现使用的寄存器、函数栈也更加复杂。

场景	C 语言代码	汇编代码	分析

函数返回 1 个外部指针	`cat_t *func_1_return_pionter()`
	`{`
	`    return &Tom;`
	`}`
		`func_1_return_pionter:`
		`    movl    $Tom, %eax`
			把变量 Tom 的地址，赋给寄存器 eax，用 eax 返回指针。

函数返回 1 个栈内指针	void *func_1_return_pionter2()
	{
	    // 仅供测试，真实场景不要把栈内地址返回到外部
	    int buf;
	    return &buf;
	}
		func_1_return_pionter2:
		    leaq -4(%rbp), %rax
			把栈上的地址，赋给寄存器 rax，用 rax 返回指针。

函数返回 1 个外部 struct	cat_t func_1_return_struct()
	{
	    return Tom;
	}
		func_1_return_struct:
		    movq Tom(%rip), %rax
		    movq Tom+8(%rip), %rdx
		    movq %rdx, -16(%rbp)
		    movsd -16(%rbp), %xmm0
			返回一个 struct，值传递，把 struct 复制一份。
			整数 age，用寄存器 rax 返回。
			浮点数 height，用寄存器 xmm0 返回。

问题：上方返回 2 个指针，一个用 eax，一个用 rax，为什么不一样？
eax 表示 4 字节整数。rax 表示 8 字节整数。
进程的内存布局规定，程序的全局变量在比较低的地址空间，程序的栈在比较高的地址空间。
查看上文的内存布局，内存区间 `00601000-00602000 rw-p` ，此区间表示程序的可读可写数据段。
eax 可以表示程序的全局变量，因为高位 4 个字节都是 0。 `movl $Tom, %eax` 把 rax 的高位 4 个字节清零，低位 4 个字节赋值。

用汇编实现返回值使用自定义的函数栈

编写代码： return_stack.s

```
.global main

.data
out_str :
    .string "return values :  %d  %d  \n"

.text
```

```
# 返回值使用函数栈
func_self_stack :
    pushq %rbp
    movq %rsp, %rbp

    movl $777, 16(%rbp)  # 返回值 1，在函数栈写一个 int
    movl $555, 24(%rbp)  # 返回值 2，在函数栈写一个 int

    popq %rbp
    retq

main :
    pushq %rbp
    movq %rsp, %rbp

    subq $64, %rsp   # 扩大栈

    callq func_self_stack  # 调用函数，获得返回值

    movq $out_str, %rdi
    movl 0(%rsp), %esi   # 从栈上取返回值 1
    movl 8(%rsp), %edx   # 从栈上取返回值 2
    callq printf

    addq $64, %rsp   # 缩小栈。必须和扩大栈对应

    popq %rbp
    retq
```

编译代码：

```
gcc return_stack.s -o return_stack
```

运行代码：

```
[root@192 func]# ./return_stack
return values :  777  555
```

分析结果：

函数 main，调用函数 func_self_stack，从函数栈读取返回值，需要在栈帧预留足够的空间，使用 `subq $64, %rsp` 扩大栈帧。

函数 func_self_stack，使用函数栈传递返回值，把返回值写到函数 main 的栈帧， `movl $777, 16(%rbp)` ， `movl $555, 24(%rbp)` 。

函数的栈帧在函数调用退出后会销毁，所以不能把返回值写到函数自己的栈帧，必须写到调用方的栈帧。