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

值传递与引用传递的含义

值传递，把旧值拷贝一份，获得新值，然后传递使用新值。旧值和新值是 2 个变量，修改新值，不会影响旧值。
引用传递，核心在于引用二字。引用表示地址，即指针。使用指针传递，表示引用传递。否则，表示值传递。

使用技巧：
数字变量，并且函数不修改原始变量，建议使用值传递，也可以使用引用传递。
数字变量，并且函数修改原始变量，需要使用引用传递。
复杂变量，并且函数不修改原始变量，建议使用引用传递。
复杂变量，并且函数修改原始变量，需要使用引用传递。

用 C 和汇编分析数字变量的传递

编写代码： refer_num.c

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

// 值传递，简单数字
int func_value_num(int day)
{
    day = 3;
    return day;
}
```

```
// 引用传递，简单数字。使用指针
int *func_refer_num_pointer(int *month)
{
    (*month) = 11;
    return month;
}

// 引用传递，简单数字。使用 64 位地址
uint64_t func_refer_num_u64(uint64_t year_addr)
{
    // 把地址转为指针，再操作
    int *tmp = (int *)year_addr;
    *tmp = 2023;
    return year_addr;
}

int day = 2;
int month = 9;
int year = 1970;
int main()
{
    printf("值传递：  \n");
    printf("  调用前 = %d  \n", day);
    int day_out = func_value_num(day);
    printf("  调用后 = %d  \n", day);
    printf("  返回值 = %d  \n", day_out);
    printf("\n");

    printf("引用传递，指针模式：  \n");
    printf("  调用前 = %d  \n", month);
    // 直接传指针
    int *month_out = func_refer_num_pointer(&month);
    printf("  调用后 = %d  \n", month);
```

```
    printf("   返回值 = %p  \n", month_out);
    printf("\n");

    printf("引用传递，64 位地址模式：  \n");
    printf("   调用前 = %d  \n", year);
    // 获得 64 位地址
    uint64_t year_addr = (uint64_t)(&year);
    uint64_t year_out = func_refer_num_u64(year_addr);
    printf("   调用后 = %d  \n", year);
    printf("   返回值 = %#llx  \n", year_out);
    printf("\n");

    return 0;
}
```

编译代码：

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

运行代码：

```
[root@192 func]# ./refer_num
值传递：
   调用前 = 2
   调用后 = 2
   返回值 = 3

引用传递，指针模式：
   调用前 = 9
   调用后 = 11
   返回值 = 0x601048

引用传递，64 位地址模式：
   调用前 = 1970
```

```
调用后 = 2023
返回值 = 0x60104c
```

分析结果：
值传递，函数调用前后，没有改变变量的值。
引用传递，函数调用前后，改变变量的值。

分析函数定义。查看 refer_num.c、refer_num.s，这里截取核心部分。

分类	源码	汇编代码	分析

值传递	`int func_value_num(int day)`
	`{`
	`  day = 3;`
	`  return day;`
	`}`
		`func_value_num:`
		`  movl  %edi, -20(%rbp)`
		`  movl  $3, -4(%rbp)`
		`  movl  -4(%rbp), %eax`
			edi 传入 1 个 32 位数字，使用 movl 指令。
			eax 返回 1 个 32 位数字。

引用传递
指针模式
	`int *func_refer_num_pointer(int *month)`
	`{`
	`  (*month) = 11;`
	`  return month;`
	`}`
		`func_refer_num_pointer:`
		`  movq  %rdi, -8(%rbp)`
		`  movq  -8(%rbp), %rax`
		`  movl  $11, (%rax)`
		`  movq  -8(%rbp), %rax`
			rdi 传入 1 个 64 位地址，使用 movq 指令。
			rax 返回 1 个 64 位地址。

引用传递
64 位地址模式
	`uint64_t func_refer_num_u64(uint64_t year_addr)`
	`{`
	`  // 把地址转为指针，再操作`
	`  int *tmp = (int *)year_addr;`
	`  *tmp = 2023;`
	`  return year_addr;`
	`}`
		`func_refer_num_u64:`
		`  movq  %rdi, -24(%rbp)`
		`  movq  -24(%rbp), %rax`
		`  movq  %rax, -8(%rbp)`
		`  movq  -8(%rbp), %rax`
		`  movl  $2023, (%rax)`
		`  movq  -24(%rbp), %rax`
			rdi 传入 1 个 64 位地址，使用 movq 指令。
			rax 返回 1 个 64 位地址。

入参的区别：
引用传递，使用 64 位寄存器传递 64 位地址。比如，`movq %rdi, -8(%rbp)`。
值传递，按需使用寄存器。比如，`movl %edi, -20(%rbp)`。

返回值的区别：

引用传递，使用 64 位寄存器传递 64 位地址。比如， `movq -8(%rbp), %rax` 。
值传递，按需使用寄存器。比如， `movl -4(%rbp), %eax` 。

函数 func_refer_num_pointer 和函数 func_refer_num_u64 的汇编代码非常相似。使用 rdi 传入地址，使用 rax 返回地址。因为指针和地址本质一样。

分析函数调用。查看 refer_num.c、refer_num.s，这里截取核心部分。

分类	源码	汇编代码	分析

值传递	`int day_out = func_value_num(day);`	`movl day(%rip), %eax`
		`movl %eax, %edi`
		`call func_value_num`
			取变量 day 的 32 位值，放入 edi。
			可以简化为 `movl day(%rip), %edi` 。

引用传递 指针模式	`int *month_out =` `func_refer_num_pointer(&month);`	`movl $month, %edi` `call func_refer_num_pointer`	取变量 month 的地址，放入 edi。 等价于 `movq $month, %rdi` 。

引用传递
64 位地址模式
	`uint64_t year_addr = (uint64_t)(&year);`
	`uint64_t year_out =`
	`func_refer_num_u64(year_addr);`
		`movq $year, -24(%rbp)`
		`movq -24(%rbp), %rax`
		`movq %rax, %rdi`
		`call func_refer_num_u64`
			取变量 year 的地址，放入 rdi。
			可以简化为 `movq $year, %rdi` 。

问题：为什么变量 month 使用 edi 传参，变量 year 使用 rdi 传参？
变量 month、year 在程序的数据区，高 32 位是 0，低 32 位能够表示内存地址。
`movl $month, %edi` 和 `movq $month, %rdi` 功能等价。

函数 func_refer_num_pointer 和函数 func_refer_num_u64 的函数调用的汇编代码非常相似。使用 rdi 传入地址 `movq $param, %rdi` ，因为指针和地址本质一样。

用 C 和汇编分析 struct 变量的传递

编写代码： refer_struct.c

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

// 结构体
typedef struct
{
    // 每个属性使用 64 位，更直观
    // 设置多个属性，使用栈传递 struct，对比效果更明显
    int64_t age;
    int64_t speed;
    int64_t height;
    int64_t weight;
} cat_t;

// 值传递。作为入参
void func_value_as_param(cat_t cat)
{
    cat.age = 100;
    cat.speed = 200;
    cat.height = 300;
    cat.weight = 400;
}

// 值传递。作为返回值
cat_t func_value_as_return()
{
    cat_t cat;
    cat.age = 500;
    cat.speed = 600;
    cat.height = 700;
    cat.weight = 800;
    return cat;
```

```
}

// 引用传递。指针类型
cat_t *func_value_pointer(cat_t *cat)
{
    cat->age = 1000;
    cat->speed = 2000;
    cat->height = 3000;
    cat->weight = 4000;
    return cat;
}

void print_cat(char *title, cat_t *cat)
{
    printf("  %s  addr = %p  age = %lld  speed = %lld  height = %lld  weight = %lld \n",
           title, cat, cat->age, cat->speed, cat->height, cat->weight);
}

// 全局变量
cat_t cat_tom;

int main()
{
    // 初始化
    cat_tom.age = 1;
    cat_tom.speed = 2;
    cat_tom.height = 3;
    cat_tom.weight = 4;

    printf("值传递：作为入参 \n");
    print_cat("调用前", &cat_tom);
    func_value_as_param(cat_tom);
    print_cat("调用后", &cat_tom);
```

```
    printf("\n");

    printf("值传递：作为返回值 \n");
    cat_t cat_red = func_value_as_return();
    print_cat("调用后", &cat_red);
    printf("\n");

    printf("引用传递：指针 \n");
    print_cat("调用前", &cat_tom);
    func_value_pointer(&cat_tom);
    print_cat("调用后", &cat_tom);
    printf("\n");

    return 0;
}
```

编译代码：

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

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

运行代码：

```
[root@192 func]# ./refer_struct
值传递：作为入参
  调用前  addr = 0x601080  age = 1  speed = 2  height = 3  weight = 4
  调用后  addr = 0x601080  age = 1  speed = 2  height = 3  weight = 4

值传递：作为返回值
  调用后  addr = 0x7ffef8478600  age = 500  speed = 600  height = 700  weight = 800

引用传递：指针
```

```
调用前  addr = 0x601080  age = 1  speed = 2  height = 3  weight = 4
调用后  addr = 0x601080  age = 1000  speed = 2000  height = 3000  weight = 4000
```

分析结果：
值传递，函数调用前后，没有改变变量的值。
引用传递，函数调用前后，改变变量的值。
值传递，返回 1 个 struct，其地址 0x7ffef8478600 在栈上。

分析函数定义。查看 refer_struct.c、refer_struct.s，这里截取核心部分。

分类	源码	汇编代码	分析

值传递
作为入参
	void func_value_as_param(cat_t cat)
	{
	    cat.age = 100;
	    cat.speed = 200;
	    cat.height = 300;
	    cat.weight = 400;
	}
		func_value_as_param:
		    movq $100, 16(%rbp)
		    movq $200, 24(%rbp)
		    movq $300, 32(%rbp)
		    movq $400, 40(%rbp)
			在外层函数的栈帧，依次写入 cat 的 4 个属性。

值传递
作为返回值
	cat_t func_value_as_return()
	{
	    cat_t cat;
	    cat.age = 500;
	    cat.speed = 600;
	    cat.height = 700;
	    cat.weight = 800;
	    return cat;
	}
		func_value_as_return:
		    movq %rdi, -40(%rbp)
		    movq $500, -32(%rbp)
		    movq $600, -24(%rbp)
		    movq $700, -16(%rbp)
		    movq $800, -8(%rbp)
		    movq -40(%rbp), %rax
		    movq -32(%rbp), %rdx
		    movq %rdx, (%rax)
		    movq -24(%rbp), %rdx
		    movq %rdx, 8(%rax)
		    movq -16(%rbp), %rdx
		    movq %rdx, 16(%rax)
		    movq -8(%rbp), %rdx
		    movq %rdx, 24(%rax)
		    movq -40(%rbp), %rax
			入参 rdi，传入返回值需要拷贝到的地址。
			在 func_value_as_return 的栈帧，初始化 cat 变量，依次写入 cat 的 4 个属性。
			把 cat 变量，复制到入参 rdi 指向的地址，依次复制 cat 的 4 个属性。

引用传递
指针类型
	cat_t *func_value_pointer(cat_t *cat)
	{
	cat->age = 1000;
	cat->speed = 2000;
	cat->height = 3000;
	cat->weight = 4000;
	return cat;
	}
		func_value_pointer:
		movq %rdi, -8(%rbp)
		movq -8(%rbp), %rax
		movq $1000, (%rax)
		movq -8(%rbp), %rax
		movq $2000, 8(%rax)
		movq -8(%rbp), %rax
		movq $3000, 16(%rax)
		movq -8(%rbp), %rax
		movq $4000, 24(%rax)
		movq -8(%rbp), %rax
			入参 rdi，传入变量 cat 的地址。
			依次写入 cat 的 4 个属性。
			返回值 rax，返回变量 cat 的地址。

分析函数调用。查看 refer_struct.c、refer_struct.s，这里截取核心部分。

分类	源码	汇编代码	分析

给 struct 赋值	cat_tom.age = 1;
	cat_tom.speed = 2;
	cat_tom.height = 3;
	cat_tom.weight = 4;
		movq $1, cat_tom(%rip)
		movq $2, cat_tom+8(%rip)
		movq $3, cat_tom+16(%rip)
		movq $4, cat_tom+24(%rip)
			依次设置 cat_tom 的 4 个属性。
			使用 rip 相对寻址。

值传递
作为入参
	func_value_as_param(cat_tom);	movq cat_tom(%rip), %rax
		movq %rax, (%rsp)
		movq cat_tom+8(%rip), %rax
		movq %rax, 8(%rsp)
		movq cat_tom+16(%rip), %rax
		movq %rax, 16(%rsp)
		movq cat_tom+24(%rip), %rax
		movq %rax, 24(%rsp)
		call func_value_as_param
			把 cat_tom 拷贝一份，放在 main 的栈上。
			依次拷贝 cat_tom 的 4 个属性。
			新值的起始地址在 rsp 指向的地址。

值传递
作为返回值
	cat_t cat_red =
	func_value_as_return();
		leaq -32(%rbp), %rax
		movq %rax, %rdi
		movl $0, %eax
		call func_value_as_return
			在 main 的栈帧，分配一块空间。
			使用 rdi，传入刚才分配的栈空间。

引用传递 指针类型	func_value_pointer(&cat_tom);	movl $cat_tom, %edi call func_value_pointer	使用 edi，传入 cat_tom 的地址。

值传递，对于复杂的 struct，需要借助函数栈，操作复杂。
把 struct 变量复制到栈上，依次复制 struct 的每个属性。然后，把新值所在的栈地址，传入被调用的函数。
频繁复制 struct，使用很多指令，影响性能。

引用传递，使用 struct 变量的地址，操作同一个 struct 变量。
从汇编代码角度看，引用传递的代码量比值传递少许多。

问题：引用传递一定比值传递效率高吗？
不一定。如果是简单的数字变量，引用传递会导致更多的地址操作，效率反而更低。

用 C 和汇编分析只读场景的数字变量

编写代码： refer_read_only.c

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

// 值传递
void func_value(int64_t speed, double height)
{
    int64_t tmp_int = speed + 666;
    double tmp_double = height + 666;
}

// 引用传递
```

```
void func_pointer(int64_t *speed, double *height)
{
    int64_t tmp_int = *speed + 666;
    double tmp_double = *height + 666;
}

// 2 个变量
int64_t speed = 120;
double height = 33.55F;

int main()
{
    func_value(speed, height);
    func_pointer(&speed, &height);
    return 0;
}
```

编译代码：

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

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

分析结果：
对于数字变量且只读的场景，引用传递比值传递多一个操作，使用地址获得值。此时，值传递的指令更少，效率更高。

分析函数定义。查看 refer_read_only.c、refer_read_only.s，这里截取核心部分。

分类	值传递	引用传递	分析

整数	func_value:
	movq %rdi, -24(%rbp)
	movq -24(%rbp), %rax
	addq $666, %rax
		func_pointer:
		movq %rdi, -24(%rbp)
		movq -24(%rbp), %rax
		movq (%rax), %rax
			值传递，使用 rdi 传值。
			引用传递，使用 rdi 传地址。
			引用传递比值传递多一个操作,使用地址获得值， `movq (%rax), %rax` 。

		addq $666, %rax	

浮点数	func_value:
	movsd %xmm0, -32(%rbp)
	movsd -32(%rbp), %xmm1
	movsd .LC0(%rip), %xmm0
	addsd %xmm1, %xmm0
		func_pointer:
		movq %rsi, -32(%rbp)
		movq -32(%rbp), %rax
		movsd (%rax), %xmm1
		movsd .LC0(%rip), %xmm0
		addsd %xmm1, %xmm0
			值传递，使用 xmm0 传值。
			引用传递，使用 rsi 传地址。
			引用传递比值传递多一个操作,使用地址获得值， `movsd (%rax), %xmm1` 。