

golang | 是返回struct还是返回struct的指针

原创 KINGYT 卯时卯刻 2021-08-02 08:00

收录于话题
#golang

1个


当我们定义一个函数时，是返回结构体呢，还是返回指向结构体的指针呢？

对于这个问题，我想大部分人的回答，肯定都是返回指针，因为这样可以避免结构体的拷贝，使代码的效率更高，性能更好。

但真的是这样吗？

在回答这个问题之前，我们先写几个示例，来确定一些基本事实：


```
→ test_go cat -n hello.go
1 package main
2
3 type S struct {
4     a1, a2, a3 int
5     b1, b2    string
6     c1        []int
7 }
8
9 func main() {
10     s := f()
11     if s.a2 != 2 {
12         panic("err")
13     }
14 }
15
16 func f() *S {
17     return &S{a1: 1, a2: 2, b1: "b1"}
18 }
→ test_go _
```

 微信号: ytcode

上图中，函数f返回的是结构体S的指针，即一个地址，这个可以通过其汇编来确认：

```
→ test_go go tool compile -l hello.go && go tool objdump hello.o
TEXT ".main(SB) gofile../home/yt/test_go/hello.go
hello.go:9      0xd7f      493b6610      CMPQ 0x10(R14), SP      [2:2]R_USEIFACE:type.string
hello.go:9      0xd83      763a         JBE 0xdbf
hello.go:9      0xd85      4883ec18      SUBQ $0x18, SP
hello.go:9      0xd89      48896c2410    MOVQ BP, 0x10(SP)
hello.go:9      0xd8e      488d6c2410    LEAQ 0x10(SP), BP
hello.go:10     0xd93      e800000000    CALL 0xd98      [1:5]R_CALL:"".f
hello.go:11     0xd98      4883780802    CMPQ $0x2, 0x8(AX)
hello.go:11     0xd9d      6690         NOPW
hello.go:11     0xd9f      750a         JNE 0xdab
hello.go:14     0xda1      488b6c2410    MOVQ 0x10(SP), BP
hello.go:14     0xda6      4883c418      ADDQ $0x18, SP
hello.go:14     0xdaa      c3          RET
hello.go:12     0xdab      488d0500000000 LEAQ 0(IP), AX      [3:7]R_PCREL:type.string
hello.go:12     0xdb2      488d1d00000000 LEAQ 0(IP), BX      [3:7]R_PCREL:""..stmp_0<1>
hello.go:12     0xdb9      e800000000    CALL 0xdbe      [1:5]R_CALL:runtime.gopanic<1>
hello.go:12     0xdbe      90          NOPL
hello.go:9      0xdbf      e800000000    CALL 0xdc4      [1:5]R_CALL:runtime.morestack_noctxt
hello.go:9      0xdc4      ebb9        JMP ".main(SB)

TEXT ".f(SB) gofile../home/yt/test_go/hello.go
hello.go:16     0xdc6      493b6610      CMPQ 0x10(R14), SP
hello.go:16     0xdca      764b         JBE 0xe17
hello.go:16     0xdcc      4883ec18      SUBQ $0x18, SP
hello.go:16     0xdd0      48896c2410    MOVQ BP, 0x10(SP)
hello.go:16     0xdd5      488d6c2410    LEAQ 0x10(SP), BP
hello.go:17     0xdda      488d0500000000 LEAQ 0(IP), AX      [3:7]R_PCREL:type."".S
hello.go:17     0xde1      0f1f440000    NOPL 0(AX)(AX*1)
hello.go:17     0xde6      e800000000    CALL 0xdeb      [1:5]R_CALL:runtime.newobject<1>
hello.go:17     0xdeb      48c70001000000 MOVQ $0x1, 0(AX)
hello.go:17     0xdf2      48c7400802000000 MOVQ $0x2, 0x8(AX)
hello.go:17     0xdfa      48c7402002000000 MOVQ $0x2, 0x20(AX)
hello.go:17     0xe02      488d0d00000000 LEAQ 0(IP), CX      [3:7]R_PCREL:go.string."b1"
hello.go:17     0xe09      48894818      MOVQ CX, 0x18(AX)
hello.go:17     0xe0d      488b6c2410    MOVQ 0x10(SP), BP
hello.go:17     0xe12      4883c418      ADDQ $0x18, SP
hello.go:17     0xe16      c3          RET
hello.go:16     0xe17      e800000000    CALL 0xe1c      [1:5]R_CALL:runtime.morestack_noctxt
hello.go:16     0xe1c      eba8        JMP ".f(SB)
→ test_go _
```

 微信号: ytcode

看上图中的选中行。

第一行是调用函数f，其结果，即结构体S的指针，或结构体S的地址，是放到ax寄存器中返回的。

第二行用0x8(ax)，即ax中的地址加8的形式，来获得结构体S中a2字段的值，然后将该值和0x2相比，以进行后续逻辑。

由此可见，返回结构体指针的形式，确实是只传递了一个地址。

我们再来看下返回结构体的情况：

```
→ test_go cat -n hello.go
1 package main
2
3 type S struct {
4     a1, a2, a3 int
5     b1, b2    string
6     c1        []int
7 }
8
9 func main() {
10     s := f()
11     if s.a2 != 2 {
12         panic("err")
13     }
14 }
15
16 func f() S {
17     return S{a1: 1, a2: 2, b1: "b1"}
18 }
→ test_go _
```

微信号: ytcode

这次函数f返回的是S，而不是*S，看看这样写其汇编是什么样子：

```
→ test_go go tool compile -l hello.go && go tool objdump hello.o
TEXT ".main(SB) gofile../home/yt/test_go/hello.go
hello.go:9      0xe32      4c8d6424d8      LEAQ -0x28(SP), R12      [3:3]R_USEIFACE:type.string
hello.go:9      0xe37      4d3b6610      CMPQ 0x10(R14), R12
hello.go:9      0xe3b      7674      JBE 0xeb1
hello.go:9      0xe3d      4881eca8000000      SUBQ $0xa8, SP
hello.go:9      0xe44      4889ac24a0000000      MOVQ BP, 0xa0(SP)
hello.go:9      0xe4c      488dac24a0000000      LEAQ 0xa0(SP), BP
hello.go:10     0xe54      e800000000      CALL 0xe59      [1:5]R_CALL:"".f
hello.go:10     0xe59      488d7c2450      LEAQ 0x50(SP), DI
hello.go:10     0xe5e      4889e6      MOVQ SP, SI
hello.go:10     0xe61      660f1f840000000000      NOPW 0(AX)(AX*1)
hello.go:10     0xe6a      0f1f84000000000000      NOPL 0(AX)(AX*1)
hello.go:10     0xe72      48896c24f0      MOVQ BP, -0x10(SP)
hello.go:10     0xe77      488d6c24f0      LEAQ -0x10(SP), BP
hello.go:10     0xe7c      e800000000      CALL 0xe81      [1:5]R_CALL:runtime.duffcopy+826
hello.go:10     0xe81      488b6d00      MOVQ 0(BP), BP
hello.go:11     0xe85      48837c245802      CMPQ $0x2, 0x58(SP)
hello.go:11     0xe8b      7510      JNE 0xe9d
hello.go:14     0xe8d      488bac24a0000000      MOVQ 0xa0(SP), BP
hello.go:14     0xe95      4881c4a800000000      ADDQ $0xa8, SP
hello.go:14     0xe9c      c3      RET
hello.go:12     0xe9d      488d050000000000      LEAQ 0(IP), AX      [3:7]R_PCREL:type.string
hello.go:12     0xea4      488d1d0000000000      LEAQ 0(IP), BX      [3:7]R_PCREL:""..stmp_0<1>
hello.go:12     0xeab      e800000000      CALL 0xeb0      [1:5]R_CALL:runtime.gopanic<1>
hello.go:12     0xeb0      90      NOPL
hello.go:9      0xeb1      90      NOPL
hello.go:9      0xeb2      e800000000      CALL 0xeb7      [1:5]R_CALL:runtime.morestack_noctxt
hello.go:9      0xeb7      e976ffffff      JMP ".main(SB)

TEXT ".f(SB) gofile../home/yt/test_go/hello.go
hello.go:16     0xebc      4883ec08      SUBQ $0x8, SP
hello.go:16     0xec0      48892c24      MOVQ BP, 0(SP)
hello.go:16     0xec4      488d2c24      LEAQ 0(SP), BP
hello.go:16     0xec8      488d7c2410      LEAQ 0x10(SP), DI
hello.go:16     0xecd      488d7fd0      LEAQ -0x30(DI), DI
hello.go:16     0xed1      660f1f840000000000      NOPW 0(AX)(AX*1)
hello.go:16     0xeda      6690      NOPW
hello.go:16     0xedc      48896c24f0      MOVQ BP, -0x10(SP)
hello.go:16     0xee1      488d6c24f0      LEAQ -0x10(SP), BP
hello.go:16     0xee6      e800000000      CALL 0xeeb      [1:5]R_CALL:runtime.duffzero+336
hello.go:16     0xeeb      488b6d00      MOVQ 0(BP), BP
hello.go:17     0xeef      48c744241001000000      MOVQ $0x1, 0x10(SP)
hello.go:17     0xef8      48c744241802000000      MOVQ $0x2, 0x18(SP)
hello.go:17     0xf01      488d050000000000      LEAQ 0(IP), AX      [3:7]R_PCREL:go.string."b1"
hello.go:17     0xf08      4889442428      MOVQ AX, 0x28(SP)
hello.go:17     0xf0d      48c744243002000000      MOVQ $0x2, 0x30(SP)
hello.go:17     0xf16      488b2c24      MOVQ 0(SP), BP
hello.go:17     0xf1a      4883c408      ADDQ $0x8, SP
hello.go:17     0xf1e      c3      RET
→ test_go _
```

微信号: ytcode

上图main函数的汇编中，通过调用函数f，初始化了main函数栈中，0x0(sp)到0x50(sp)的内存段，该内存段共有80个字节，正好对应于结构体S的大小。

在函数f返回后，sp寄存器存放的，正是函数f初始化的结构体S的地址。

接着，我们看上图中的选中行，该段逻辑通过runtime.duffcopy函数，将栈中内存段0x0(sp)到0x50(sp)的值，拷贝到了内存段0x50(sp)到0xa0(sp)的部分，即将函数f初始化的结构体S，从内存地址0x0(sp)，拷贝到了0x50(sp)。

然后，通过0x58(sp)，即sp中的地址加上0x58的形式，获得拷贝后的结构体S中，a2字段的值，最后将其和0x2比较，以进行后续逻辑。


由上可见，当函数返回结构体时，确实存在着一次结构体的拷贝操作。

对比以上两个示例我们看到，返回指针的确会更好些，因为这样节省了一次结构体的拷贝操作。

但这样性能就真的更好吗？


写个benchmark测试下：

```
→ test_go cat -n hello.go
1  package main
2
3  type S struct {
4      a1, a2, a3 int
5      b1, b2    string
6      c1        []int
7  }
8
9  func f1() *S {
10     return &S{a1: 1, a2: 2, b1: "b1"}
11 }
12
13 func f2() S {
14     return S{a1: 1, a2: 2, b1: "b1"}
15 }
→ test_go cat -n hello_test.go
1  package main
2
3  import "testing"
4
5  func BenchmarkF1(b *testing.B) {
6      for i := 0; i < b.N; i++ {
7          s1 := f1()
8          if s1.a2 != 2 {
9              panic("f1")
10         }
11     }
12 }
13
14 func BenchmarkF2(b *testing.B) {
15     for i := 0; i < b.N; i++ {
16         s2 := f2()
17         if s2.a2 != 2 {
18             panic("f2")
19         }
20     }
21 }
→ test_go _
```

 微信号: ytcode

执行看下结果：

```
→ test_go go test -bench=. -benchmem
goos: linux
goarch: amd64
pkg: ytcode.io/hello
cpu: Intel(R) Core(TM) i7-7600U CPU @ 2.80GHz
BenchmarkF1-4      1000000000    0.2596 ns/op    0 B/op    0 allocs/op
BenchmarkF2-4      1000000000    0.2591 ns/op    0 B/op    0 allocs/op
PASS
ok      ytcode.io/hello 0.586s
→ test_go _
```

 微信号: ytcode

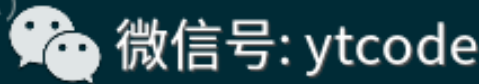
这两个benchmark的时间几乎是相等的，其结果并不像我们预料的那样，返回指针的形式会更快些。

为什么呢？

看下这两个benchmark对应的汇编：

```
→ test_go go test -c && go tool objdump -s ytcode.io hello.test
TEXT ytcode.io/hello.BenchmarkF1(SB) /home/yt/test_go/hello_test.go
hello_test.go:5      0x4e38c0      31c9      XORL CX, CX
hello_test.go:6      0x4e38c2      eb03      JMP 0x4e38c7
hello_test.go:6      0x4e38c4      48ffc1      INCQ CX
hello_test.go:6      0x4e38c7      48398890010000      CMPQ CX, 0x190(AX)
hello_test.go:6      0x4e38ce      7ff4      JG 0x4e38c4
hello_test.go:12     0x4e38d0      c3      RET

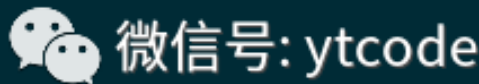
TEXT ytcode.io/hello.BenchmarkF2(SB) /home/yt/test_go/hello_test.go
hello_test.go:14     0x4e38e0      31c9      XORL CX, CX
hello_test.go:15     0x4e38e2      eb03      JMP 0x4e38e7
hello_test.go:15     0x4e38e4      48ffc1      INCQ CX
hello_test.go:15     0x4e38e7      48398890010000      CMPQ CX, 0x190(AX)
hello_test.go:15     0x4e38ee      7ff4      JG 0x4e38e4
hello_test.go:21     0x4e38f0      c3      RET
→ test_go _
```



它们居然都被优化成了空跑for循环了，难怪这两个测试耗时是一样的。

加上编译器指令//go:noinline，防止f1/f2函数被内联，进而被过度优化：

```
→ test_go cat -n hello.go
1  package main
2
3  type S struct {
4      a1, a2, a3 int
5      b1, b2    string
6      c1        []int
7  }
8
9  //go:noinline
10 func f1() *S {
11     return &S{a1: 1, a2: 2, b1: "b1"}
12 }
13
14 //go:noinline
15 func f2() S {
16     return S{a1: 1, a2: 2, b1: "b1"}
17 }
→ test_go _
```



如上图的第9行和第14行。

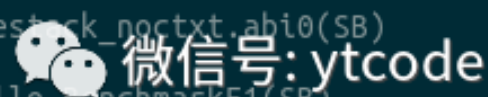
再来看下测试程序的汇编，确保以上操作是有效的。

先看下函数f1及其对应的benchmark：


```
→ test_go go test -c && go tool objdump -s 'ytcode.io/hello*[f|F]1' hello.test
TEXT ytcodes.io/hello.f1(SB) /home/yt/test_go/hello.go
hello.go:10      0x4e38c0      493b6610      CMPQ 0x10(R14), SP
hello.go:10      0x4e38c4      764b         JBE 0x4e3911
hello.go:10      0x4e38c6      4883ec18      SUBQ $0x18, SP
hello.go:10      0x4e38ca      48896c2410     MOVQ BP, 0x10(SP)
hello.go:10      0x4e38cf      488d6c2410     LEAQ 0x10(SP), BP
hello.go:11      0x4e38d4      488d0565000200 LEAQ 0x20065(IP), AX
hello.go:11      0x4e38db      0f1f440000     NOPL 0(AX)(AX*1)
hello.go:11      0x4e38e0      e85b95f2ff     CALL runtime.newobject(SB)
hello.go:11      0x4e38e5      48c70001000000 MOVQ $0x1, 0(AX)
hello.go:11      0x4e38ec      48c7400802000000 MOVQ $0x2, 0x8(AX)
hello.go:11      0x4e38f4      48c7402002000000 MOVQ $0x2, 0x20(AX)
hello.go:11      0x4e38fc      488d0db7980200 LEAQ 0x298b7(IP), CX
hello.go:11      0x4e3903      48894818      MOVQ CX, 0x18(AX)
hello.go:11      0x4e3907      488b6c2410     MOVQ 0x10(SP), BP
hello.go:11      0x4e390c      4883c418      ADDQ $0x18, SP
hello.go:11      0x4e3910      c3           RET
hello.go:10      0x4e3911      e86adc7f7ff     CALL runtime.morestack_noctxt.abi0(SB)
hello.go:10      0x4e3916      eba8         JMP ytcodes.io/hello.f1(SB)

TEXT ytcodes.io/hello.BenchmarkF1(SB) /home/yt/test_go/hello_test.go
hello_test.go:5  0x4e39a0      493b6610      CMPQ 0x10(R14), SP
hello_test.go:5  0x4e39a4      7660         JBE 0x4e3a06
hello_test.go:5  0x4e39a6      4883ec20      SUBQ $0x20, SP
hello_test.go:5  0x4e39aa      48896c2418     MOVQ BP, 0x18(SP)
hello_test.go:5  0x4e39af      488d6c2418     LEAQ 0x18(SP), BP
hello_test.go:5  0x4e39b4      4889442428     MOVQ AX, 0x28(SP)
hello_test.go:5  0x4e39b9      31c9         XORL CX, CX
hello_test.go:6  0x4e39bb      eb0e         JMP 0x4e39cb
hello_test.go:6  0x4e39bd      488b542410     MOVQ 0x10(SP), DX
hello_test.go:6  0x4e39c2      488d4a01      LEAQ 0x1(DX), CX
hello_test.go:6  0x4e39c6      488b442428     MOVQ 0x28(SP), AX
hello_test.go:6  0x4e39cb      48398890010000 CMPQ CX, 0x190(AX)
hello_test.go:6  0x4e39d2      7e13         JLE 0x4e39e7
hello_test.go:6  0x4e39d4      48894c2410     MOVQ CX, 0x10(SP)
hello_test.go:7  0x4e39d9      e8e2feffff     CALL ytcodes.io/hello.f1(SB)
hello_test.go:8  0x4e39de      4883780802     CMPQ $0x2, 0x8(AX)
hello_test.go:8  0x4e39e3      74d8         JE 0x4e39bd
hello_test.go:8  0x4e39e5      eb0a         JMP 0x4e39f1
hello_test.go:12 0x4e39e7      488b6c2418     MOVQ 0x18(SP), BP
hello_test.go:12 0x4e39ec      4883c420      ADDQ $0x20, SP
hello_test.go:12 0x4e39f0      c3           RET
hello_test.go:9  0x4e39f1      488d0588c70000 LEAQ 0xc788(IP), AX
hello_test.go:9  0x4e39f8      488d1d611b0500 LEAQ 0x51b61(IP), BX
hello_test.go:9  0x4e39ff      90           NOPL
hello_test.go:9  0x4e3a00      e8dbf7cf4fff     CALL runtime.gopanic(SB)
hello_test.go:9  0x4e3a05      90           NOPL
hello_test.go:5  0x4e3a06      4889442408     MOVQ AX, 0x8(SP)
hello_test.go:5  0x4e3a0b      e870dbf7fff     CALL runtime.morestack_noctxt.abi0(SB)
hello_test.go:5  0x4e3a10      488b442408     MOVQ 0x8(SP), AX
hello_test.go:5  0x4e3a15      eb89         JMP ytcodes.io/hello.BenchmarkF1(SB)
→ test_go _
```

再看下函数f2及其对应的benchmark：



```
→ test_go go test -c && go tool objdump -s 'ytcode.io/hello*[f|F]2' hello.test
TEXT ytcode.io/hello.f2(SB) /home/yt/test_go/hello.go
hello.go:15      0x4e3920      4883ec08      SUBQ $0x8, SP
hello.go:15      0x4e3924      48892c24      MOVQ BP, 0(SP)
hello.go:15      0x4e3928      488d2c24      LEAQ 0(SP), BP
hello.go:15      0x4e392c      488d7c2410    LEAQ 0x10(SP), DI
hello.go:15      0x4e3931      488d7fd0      LEAQ -0x30(DI), DI
hello.go:15      0x4e3935      660f1f8400000000 NOPW 0(AX)(AX*1)
hello.go:15      0x4e393e      6690         NOPW
hello.go:15      0x4e3940      48896c24f0    MOVQ BP, -0x10(SP)
hello.go:15      0x4e3945      488d6c24f0    LEAQ -0x10(SP), BP
hello.go:15      0x4e394a      e88103f8ff    CALL 0x463cd0
hello.go:15      0x4e394f      488b6d00      MOVQ 0(BP), BP
hello.go:16      0x4e3953      48c744241001000000 MOVQ $0x1, 0x10(SP)
hello.go:16      0x4e395c      48c744241802000000 MOVQ $0x2, 0x18(SP)
hello.go:16      0x4e3965      488d054e980200 LEAQ 0x2984e(IP), AX
hello.go:16      0x4e396c      4889442428    MOVQ AX, 0x28(SP)
hello.go:16      0x4e3971      48c744243002000000 MOVQ $0x2, 0x30(SP)
hello.go:16      0x4e397a      488b2c24      MOVQ 0(SP), BP
hello.go:16      0x4e397e      4883c408      ADDQ $0x8, SP
hello.go:16      0x4e3982      c3           RET

TEXT ytcode.io/hello.BenchmarkF2(SB) /home/yt/test_go/hello_test.go
hello_test.go:14 0x4e3a20      4c8d6424d0    LEAQ -0x30(SP), R12
hello_test.go:14 0x4e3a25      4d3b6610      CMPQ 0x10(R14), R12
hello_test.go:14 0x4e3a29      0f8697000000 JBE 0x4e3ac6
hello_test.go:14 0x4e3a2f      4881ecb0000000 SUBQ $0xb0, SP
hello_test.go:14 0x4e3a36      4889ac24a8000000 MOVQ BP, 0xa8(SP)
hello_test.go:14 0x4e3a3e      488dac24a8000000 LEAQ 0xa8(SP), BP
hello_test.go:14 0x4e3a46      48898424b8000000 MOVQ AX, 0xb8(SP)
hello_test.go:14 0x4e3a4e      31c9         XORL CX, CX
hello_test.go:15 0x4e3a50      eb11         JMP 0x4e3a63
hello_test.go:15 0x4e3a52      488b542450    MOVQ 0x50(SP), DX
hello_test.go:15 0x4e3a57      488d4a01      LEAQ 0x1(DX), CX
hello_test.go:15 0x4e3a5b      488b8424b8000000 MOVQ 0xb8(SP), AX
hello_test.go:15 0x4e3a63      48398890010000 CMPQ CX, 0x190(AX)
hello_test.go:15 0x4e3a6a      7e31         JLE 0x4e3a9d
hello_test.go:15 0x4e3a6c      48894c2450    MOVQ CX, 0x50(SP)
hello_test.go:16 0x4e3a71      e8aafeffff    CALL ytcode.io/hello.f2(SB)
hello_test.go:16 0x4e3a76      488d7c2458    LEAQ 0x58(SP), DI
hello_test.go:16 0x4e3a7b      4889e6        MOVQ SP, SI
hello_test.go:16 0x4e3a7e      6690         NOPW
hello_test.go:16 0x4e3a80      48896c24f0    MOVQ BP, -0x10(SP)
hello_test.go:16 0x4e3a85      488d6c24f0    LEAQ -0x10(SP), BP
hello_test.go:16 0x4e3a8a      e8ab05f8ff    CALL 0x46403a
hello_test.go:16 0x4e3a8f      488b6d00      MOVQ 0(BP), BP
hello_test.go:17 0x4e3a93      48837c246002  CMPQ $0x2, 0x60(SP)
hello_test.go:17 0x4e3a99      74b7         JE 0x4e3a52
hello_test.go:17 0x4e3a9b      eb10         JMP 0x4e3aad
hello_test.go:21 0x4e3a9d      488bac24a8000000 MOVQ 0xa8(SP), BP
hello_test.go:21 0x4e3aa5      4881c4b0000000 ADDQ $0xb0, SP
hello_test.go:21 0x4e3aac      c3           RET
hello_test.go:18 0x4e3aad      488d05ccc60000 LEAQ 0xc6cc(IP), AX
hello_test.go:18 0x4e3ab4      488d1db51a0500 LEAQ 0x51ab5(IP), BX
hello_test.go:18 0x4e3abb      0f1f440000    NOPL 0(AX)(AX*1)
hello_test.go:18 0x4e3ac0      e81bfcf4ff    CALL runtime.gopanic(SB)
hello_test.go:18 0x4e3ac5      90           NOPL
hello_test.go:14 0x4e3ac6      4889442408    MOVQ AX, 0x8(SP)
hello_test.go:14 0x4e3acb      e8b0daf7ff    CALL runtime.morestack_noctxt.abi0(SB)
hello_test.go:14 0x4e3ad0      488b442408    MOVQ 0x8(SP), AX
hello_test.go:14 0x4e3ad5      e946ffff      JMP ytcode.io/hello.BenchmarkF2(SB)
→ test_go _
```

这次这两个都没有问题。

再来跑下benchmark：

```
→ test_go go test -bench=. -benchmem
goos: linux
goarch: amd64
pkg: ytcode.io/hello
cpu: Intel(R) Core(TM) i7-7600U CPU @ 2.80GHz
BenchmarkF1-4      28938547      38.94 ns/op      80 B/op      1 allocs/op
BenchmarkF2-4      140408112     8.516 ns/op      0 B/op      0 allocs/op
PASS
ok      ytcode.io/hello 3.234s
→ test_go _
```

这次结果显示，f2函数，即返回结构体形式，比f1函数，即返回指针的形式，居然快了将近5倍，意不意外？

这是为什么呢？

其实在上图中，就有一些线索。

看BenchmarkF1那行，其最后两列显示，每次调用f1函数，都会有一次堆内存分配操作，其分配内存的大小为80字节，正好对应于结构体S的大小，也就是说，f1函数中结构体S的内存，都是在堆上分配的。

而在BenchmarkF2中，就没有发生堆内存的分配操作，f2函数中的结构体S，都是在栈上分配的。

这个也可以通过上面展示的，f1/f2函数的汇编代码看到。

f1函数的汇编是通过runtime.newobject在堆上分配内存的，而f2函数则是直接就在栈上把内存分配好了，并没有调用runtime.newobject函数。

那为什么在堆上分配内存，会比在栈上分配内存慢这么多呢？

有两点原因，一是在堆上分配内存的函数runtime.newobject，其本身逻辑就比较复杂，二是堆上分配的内存，后期还要通过gc来对其进行内存回收，这些逻辑加起来，远比在栈上分配内存，外加一次拷贝操作要耗时的多。

有关go内存是在堆上分配的，还是在栈上分配的，这个是在编译过程中，通过逃逸分析来确定的，其主体思想是：

假设有变量v，及指向v的指针p，如果p的生命周期大于v的生命周期，则v的内存要在堆上分配。

其实逃逸分析的具体逻辑，远比上面说的复杂，如果有兴趣研究代码，可以从下面开始入手：

```
> src > cmd > compile > internal > gc > main.go > Main
53 // arguments, type-checks the parsed Go package, compiles functions to machine
54 // code, and finally writes the compiled package definition to disk.
55 func Main(archInit func(*ssagen.ArchInfo)) {
56     // ... (197 lines)
253     escape.Funcs(typecheck.Target.Decls)
254     // ... (76 lines)
330 }
331
332 func writebench(filename string) error {
1-UU-:----F1 /home/yt/goroot/src/cmd/compile/internal/gc/main.go 12% (253,0) Git-main-Go Compiler Projectle[
```

当然，我们也可以在编译时，通过加上-m参数，来让编译器告诉我们，一个变量到底是分配在堆上，还是在栈上：

```
→ test_go cat -n hello.go
1 package main
2
3 type S struct {
4     a1, a2, a3 int
5     b1, b2    string
6     c1        []int
7 }
8
9 func f1() *S {
10     return &S{a1: 1, a2: 2, b1: "b1"}
11 }
12
13 func f2() S {
14     return S{a1: 1, a2: 2, b1: "b1"}
15 }
→ test_go go tool compile -l -m hello.go
hello.go:10:9: &S{...} escapes to heap
→ test_go _
```

看上图，f1函数中的&S{...}逃逸到了堆上，即是在堆上分配的。

以上是对80字节大小的结构体，返回指针和返回值情况的比较，那如果结构体字节数更小或更大会怎么样呢？

经过测试，1MiB字节以下，返回结构体都更有优势。

那返回指针的方式是不是没用了呢？也不是，如果你最终的结构体，就是要存放到堆里，比如要存放到全局的map里，那返回指针优势就更大些，因为其省去了返回结构体时的拷贝操作。