高级篇：51-57

51. 使用指针作为方法的 receiver

只要值是可寻址的，就可以在值上直接调用指针方法。即是对一个方法，它的 receiver 是指针就足矣。

但不是所有值都是可寻址的，比如 map 类型的元素、通过 interface 引用的变量：

```
type data struct {
    name string
}

type printer interface {
    print()
}

func (p *data) print() {
    fmt.Println("name: ", p.name)
}

func main() {
    d1 := data{"one"}
    d1.print()    // d1 变量可寻址，可直接调用指针 receiver 的方法

    var in printer = data{"two"}
    in.print()    // 类型不匹配

    m := map[string]data{
        "x": data{"three"},
    }
    m["x"].print()    // m["x"] 是不可寻址的    // 变动频繁
}
```

```
cannot use data literal (type data) as type printer in assignment:
data does not implement printer (print method has pointer receiver)

cannot call pointer method on m["x"]
cannot take the address of m["x"]
```

52. 更新 map 字段的值

如果 map 一个字段的值是 struct 类型，则无法直接更新该 struct 的单个字段：

```
// 无法直接更新 struct 的字段值
type data struct {
    name string
```

```
}

func main() {
    m := map[string]data{
        "x": {"Tom"},
    }
    m["x"].name = "Jerry"
}
```

> cannot assign to struct field m["x"].name in map

因为 map 中的元素是不可寻址的。需区分开的是，slice 的元素可寻址：

```
type data struct {
    name string
}

func main() {
    s := []data{{"Tom"}}
    s[0].name = "Jerry"
    fmt.Println(s)    // [{Jerry}]
}
```

注意：不久前 gccgo 编译器可更新 map struct 元素的字段值，不过很快便修复了，官方认为是 Go1.3 的潜在特性，无需及时实现，依旧在 todo list 中。

更新 map 中 struct 元素的字段值，有 2 个方法：

- 使用局部变量

```
// 提取整个 struct 到局部变量中，修改字段值后再整个赋值
type data struct {
    name string
}

func main() {
    m := map[string]data{
        "x": {"Tom"},
    }
    r := m["x"]
    r.name = "Jerry"
    m["x"] = r
    fmt.Println(m)    // map[x:{Jerry}]
}
```

- 使用指向元素的 map 指针

```
func main() {
    m := map[string]*data{
        "x": {"Tom"},
    }

    m["x"].name = "Jerry"    // 直接修改 m["x"] 中的字段
    fmt.Println(m["x"])    // &{Jerry}
}
```

但是要注意下边这种误用：

```
func main() {
    m := map[string]*data{
        "x": {"Tom"},
    }
    m["z"].name = "what???"
    fmt.Println(m["x"])
}
```

```
panic: runtime error: invalid memory address or nil pointer dereference
```

53. nil interface 和 nil interface 值

虽然 interface 看起来像指针类型，但它不是。interface 类型的变量只有在类型和值均为 nil 时才为 nil

如果你的 interface 变量的值是跟随其他变量变化的（雾），与 nil 比较相等时小心：

```
func main() {
    var data *byte
    var in interface{}

    fmt.Println(data, data == nil)    // <nil> true
    fmt.Println(in, in == nil)    // <nil> true

    in = data
    fmt.Println(in, in == nil)    // <nil> false    // data 值为 nil，但 in 值不为
nil
}
```

如果你的函数返回值类型是 interface，更要小心这个坑：

```
// 错误示例
func main() {
```

```
    doIt := func(arg int) interface{} {
        var result *struct{} = nil
        if arg > 0 {
            result = &struct{}{}
        }
        return result
    }

    if res := doIt(-1); res != nil {
        fmt.Println("Good result: ", res)    // Good result:  <nil>
        fmt.Printf("%T\n", res)            // *struct {}    // res 不是 nil，它的值为 nil
        fmt.Printf("%v\n", res)            // <nil>
    }
}


// 正确示例
func main() {
    doIt := func(arg int) interface{} {
        var result *struct{} = nil
        if arg > 0 {
            result = &struct{}{}
        } else {
            return nil    // 明确指明返回 nil
        }
        return result
    }

    if res := doIt(-1); res != nil {
        fmt.Println("Good result: ", res)
    } else {
        fmt.Println("Bad result: ", res)    // Bad result:  <nil>
    }
}
```

54. 堆栈变量 你并不总是清楚你的变量是分配到了堆还是栈。

在 C++ 中使用 new 创建的变量总是分配到堆内存上的，但在 Go 中即使使用 new()、make() 来创建变量，变量为内存分配位置依旧归 Go 编译器管。

Go 编译器会根据变量的大小及其 "escape analysis" 的结果来决定变量的存储位置，故能准确返回本地变量的地址，这在 C/C++ 中是不行的。

在 go build 或 go run 时，加入 -m 参数，能准确分析程序的变量分配位置：

```
PS D:\work\godemo> go run -gcflags -m  test.go
# command-line-arguments
./test.go:13:6: can inline (*data).pointerFunc
./test.go:17:6: can inline data.valueFunc
./test.go:27:12: inlining call to fmt.Printf
./test.go:29:15: inlining call to (*data).pointerFunc
```

```
./test.go:30:12: inlining call to fmt.Printf
./test.go:32:13: inlining call to data.valueFunc
./test.go:33:12: inlining call to fmt.Printf
<autogenerated>:1: inlining call to data.valueFunc
./test.go:13:7: this does not escape
./test.go:17:7: this does not escape
./test.go:24:2: moved to heap: key
./test.go:26:25: make(map[string]bool) escapes to heap
./test.go:27:12: ... argument does not escape
./test.go:27:44: d.num escapes to heap
./test.go:27:50: *d.key escapes to heap
./test.go:30:12: ... argument does not escape
./test.go:30:44: d.num escapes to heap
./test.go:30:50: *d.key escapes to heap
./test.go:33:12: ... argument does not escape
./test.go:33:44: d.num escapes to heap
./test.go:33:50: *d.key escapes to heap
num=1  key=key1  items=map[]
num=7  key=key1  items=map[]
num=7  key=valueFunc.key  items=map[valueFunc:true]
```

55. GOMAXPROCS、Concurrency（并发）and Parallelism（并行）

Go 1.4 及以下版本，程序只会使用 1 个执行上下文 / OS 线程，即任何时间都最多只有 1 个 goroutine 在执行。

Go 1.5 版本将可执行上下文的数量设置为 runtime.NumCPU() 返回的逻辑 CPU 核心数，这个数与系统实际总的 CPU 逻辑核心数是否一致，取决于你的 CPU 分配给程序的核心数，可以使用 GOMAXPROCS 环境变量或者动态的使用 runtime.GOMAXPROCS() 来调整。

误区：GOMAXPROCS 表示执行 goroutine 的 CPU 核心数，参考文档

GOMAXPROCS 的值是可以超过 CPU 的实际数量的，在 1.5 中最大为 256

```go
func main() {
    fmt.Println(runtime.GOMAXPROCS(-1))    // 4
    fmt.Println(runtime.NumCPU())    // 4
    runtime.GOMAXPROCS(20)
    fmt.Println(runtime.GOMAXPROCS(-1))    // 20
    runtime.GOMAXPROCS(300)
    fmt.Println(runtime.GOMAXPROCS(-1))    // Go 1.9.2 // 300
}
```

56. 读写操作的重新排序 Go 可能会重排一些操作的执行顺序，可以保证在一个 goroutine 中操作是顺序执行的，但不保证多 goroutine 的执行顺序：

```go
var _ = runtime.GOMAXPROCS(3)

var a, b int
```

```
func u1() {
    a = 1
    b = 2
}

func u2() {
    a = 3
    b = 4
}

func p() {
    println(a)
    println(b)
}

func main() {
    go u1()    // 多个 goroutine 的执行顺序不定
    go u2()
    go p()
    time.Sleep(1 * time.Second)
}
```

运行效果：

```
PS D:\work\godemo> go run test.go
0
4
PS D:\work\godemo> go run test.go
3
4
PS D:\work\godemo> go run test.go
0
4
```

如果你想保持多 goroutine 像代码中的那样顺序执行，可以使用 channel 或 sync 包中的锁机制等。

57. 优先调度

你的程序可能出现一个 goroutine 在运行时阻止了其他 goroutine 的运行，比如程序中有一个不让调度器运行的 for 循环：

```
func main() {
    done := false

    go func() {
        done = true
    }()
```

```
    for !done {
    }

    println("done !")
}
```

for 的循环体不必为空，但如果代码不会触发调度器执行，将出现问题。

调度器会在 GC、Go 声明、阻塞 channel、阻塞系统调用和锁操作后再执行，也会在非内联函数调用时执行：

```
func main() {
    done := false

    go func() {
        done = true
    }()

    for !done {
        println("not done !")    // 并不内联执行
    }

    println("done !")
}
```

可以添加 -m 参数来分析 for 代码块中调用的内联函数：

```
PS D:\work\godemo> go run -gcflags -m  test.go
# command-line-arguments
./test.go:6:5: can inline main.func1
./test.go:4:2: moved to heap: done
./test.go:6:5: func literal escapes to heap
not done !
done !
```

你也可以使用 runtime 包中的 Gosched() 来 手动启动调度器：

```
func main() {
    done := false

    go func() {
        done = true
    }()

    for !done {
        runtime.Gosched()
    }
```

```
    println("done !")
}
```

运行效果:

```
PS D:\work\godemo> go run -gcflags -m  test.go
# command-line-arguments
./test.go:8:5: can inline main.func1
./test.go:13:18: inlining call to runtime.Gosched
./test.go:13:18: inlining call to runtime.checkTimeouts
./test.go:6:2: moved to heap: done
./test.go:8:5: func literal escapes to heap
done !
```