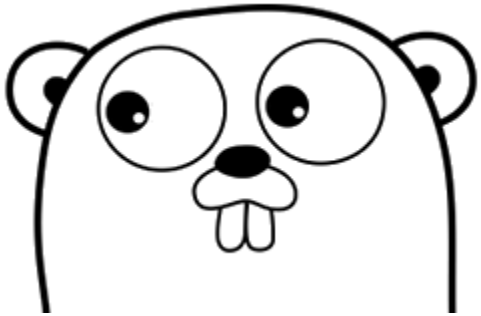
# Go语言编程实践

## 为软件工程而生，为Java程序员而写



撰写人：郝林（@特价萝卜）

# 目录

# Go语言基础——初看

- ✓ 通用编程语言，开源，跨平台
- ✓ 类C的、简介的语法，集多编程范式之大成者
- ✓ 静态类型、编译型语言
  （却看起来像动态类型、解释型语言）
- ✓ 自动垃圾回收，内置多核并发机制，强大的运行时反射
- ✓ 高生产力，高运行效率，体现优秀软件工程原则

# Go语言基础——再看

- ✓ 来自Google，2009年诞生，当前版本：1.1
- ✓ 主页：http://golang.org 和 https://code.google.com/p/go
- ✓ 语言规范： http://tip.golang.org/ref/spec
- ✓ API文档： http://godoc.org
- ✓ Go语言中文社区：
  http://www.golang.tc 和 http://bbs.mygolang.com

# Go语言基础——运算符

| 优先级 | 运算符 |
| --- | --- |
| 最高 | * / % << >> & &^ |
| | + - \| ^ |
| | == != < <= > >= |
| | <- |
| | && |
| 最低 | \|\| |

# Go语言基础——保留字

| | | | | |
|---|---|---|---|---|
| break | default | func | interface | select |
| case | defer | go | map | struct |
| chan | else | goto | package | switch |
| const | fallthrough | if | range | type |
| continue | for | import | return | var |

# Go语言基础——基本数据类型

| 类型 | 长度（字节） | 零值 | 说明 |
| --- | --- | --- | --- |
| bool | 1 | false | true，false。不能把非非零值当作 true。 |
| byte | 1 | 0 | 等同于uint8。 |
| rune | 4 | 0 | 等同于int32。存储 Unicode Code Point。 |
| int/uint | | 0 | 与平台有关，在 AMD64/X86-64 平台是 64 位整数。 |
| int8/uint8 | 1 | 0 | 范围：-128 ~ 127；0 ~ 255。 |
| int16/uint16 | | 0 | 范围：-32768 ~ 32767；0 ~ 65535。 |
| int32/uint32 | 4 | 0 | 范围：-21亿 ~ 21亿；0 ~ 42亿。 |
| int64/uint64 | 8 | 0 | |
| float32 | 4 | 0.0 | 精确到 7 个小小数位。 |
| float64 | 8 | 0.0 | 精确到 15 个小小数位。 |
| complex64 | 8 | 0.0 | |
| complex128 | 16 | 0.0 | |

# Go语言基础——基本数据类型（续）

| 类型 | 零值 | 说明 |
| --- | --- | --- |
| uintptr | nil | 足足够保存指针的 32 位或 64 位整数。 |
| array | nil | 值类型，如：[2]int。 |
| struct | | 结构体，值类型。无零值，自动实例化。 |
| string | “” | 值类型。多行时可用“`”包裹。 |
| slice | nil | 引用类型，如：[]int。 |
| map | nil | 引用类型。 |
| channel | nil | 引用类型。 |
| interface | nil | 接口类型。 |
| function | nil | 函数类型。 |

# Go语言基础——常量

声明：

```
const (
    LANG   = "Go"
    TOPIC  = "Practice"
    METHOD = "Coding"
)
```

# Go语言基础——变量

声明：

```
var i int64
var m map[string]int
var c chan
```

声明并赋值：

```
var i1, s1 = 123, "hello"
i2, s2 := 123, "hello"  //仅限函数内使用
array1 := [...]{1,2,3}  //仅限函数内使用
```

# Go语言基础——字符串

string :

```
s1 := "abcdefg"
fmt.Printf("s1: %s\n", s1[2:3])
// => s1 part: c
ba1 := []byte(s1)
ba1[2] = 'C'
s2 := string(ba1)
fmt.Printf("s2: %s\n", s2)
// => s2: abCdefg
```

# Go语言基础——字符串（续）

string :

```
// => s2: abCdefg
fmt.Printf("s2 (rune array):
%v\n",[]rune(s2))
// => s2 (rune array): [97 98 67 100
101 102 103]
fmt.Printf("Raw string:\n%s\n", `a\t
b`)
// => Raw string: a\t
// => b
```

这儿有一个回车

这儿也有一个回车

# Go语言基础——切片

slice :

```
var arr1 []int
fmt.Printf("arr1 (1):%v\n", arr1)
// => arr1 (1):[]
arr1 = append(arr1, 1)
arr1 = append(arr1, []int{2, 3, 4}...)
fmt.Printf("arr1 (2):%v\n", arr1)
// => arr1 (2):[1 2 3 4]
```

# Go语言基础——切片（续）

slice :

```
arr2 := make([]int, 5)
fmt.Printf(“arr2 (1):%v\n”, arr2)
// => arr2 (1):[0 0 0 0 0]
n := copy(arr2, arr1[1:4])
fmt.Printf("%d copied, arr2 (2):%v\n",
n, arr2)
// => 3 copied, arr2 (2):[2 3 4 0
0]
```

# Go语言基础——切片（续2）

slice :

```
// => arr2 (2):[2 3 4 0 0]
n = copy(arr2, arr1)
fmt.Printf("%d copied, arr2 (3):%v\n", n,
arr2)
// => 4 copied, arr2 (3):[1 2 3 4 0]
arr3 := []int{6, 5, 4, 3, 2, 1}
n = copy(arr2, arr3)
fmt.Printf("%d copied, arr2 (4):%v\n", n,
arr2)
// => 5 copied, arr2 (4):[6 5 4 3 2]
```

# Go语言基础——字典

map：

```
m1 := map[string]int{"A": 1, "B": 2}
fmt.Printf("m1 (1): %v\n", m1)
// => m1 (1): map[A:1 B:2]
delete(m1, "B")
fmt.Printf("m1 (2): %v\n", m1)
// => m1 (2): map[A:1]
v, ok := m1["a"]
fmt.Printf("v: %v, ok? %v\n", v, ok)
// => v: 0, ok? false
```

# Go语言基础——控制语句

if :

```
var i1 int
if i1 == 0 {
    fmt.Println("Zero value!")
} else {
    fmt.Println("Nonzero value")
}
if1 := interface{}(i1)
if i2, ok := if1.(int32); ok {
    fmt.Printf("i2: %d\n", i2) // 未被打印？
}
// => Zero value!
```

# Go语言基础——控制语句（续）

switch：

```
var n int8
switch n {
case 0:
    fallthrough  // 继续执行下面的case
case 1:
    n = (n + 1) * 2
default:
    n = -1
}
fmt.Printf("I: %d\n", n)  // => I: 2
```

# Go语言基础——控制语句（续2）

for :

```
var n uint8
for n < 100 {
    n++
}
fmt.Printf("N: %d\n", n)  // => N: 100

for i := 0; i < 100; i++ {
    n++
}
fmt.Printf("N: %d\n", n) )  // => N: 100
```

# Go语言基础——控制语句（续3）

for :

```
strings := []string{"A", "B", "C"}
for i, e := range strings {
    fmt.Printf("%d: %s\n", i, e)
}
// => 0: A
// => 1: B
// => 2: C
stringMap := map[int]string{1: "A", 2: "B", 3: "C"}
for k, v := range stringMap {
   fmt.Printf("%d: %s\n", k, v)
}
// 会打印出什么？
```

# Go语言基础——函数

函数声明（First Class Style）：

函数可作为参数

函数可作为返回值

```
func GenMyFunc(hash func(string) int64, content string) func() string {
        return func() string {
                return fmt.Sprintf("Content Hash: %v", hash(content))
        }
}
```

函数允许有多返回值。这里返回的是error实例，但是我们用占位符“_”扔掉了它。

调用代码：

```
myFunc := GenMyFunc(func(s string) int64 {
        result, _ := strconv.ParseInt(s, 0, 64)
        return result
}, "0x10")
fmt.Printf("%s\n", myFunc())
```

输出结果： Content Hash: 16

# Go语言基础——defer

defer的常用法：

```
func ReadFile(filePath string) error {
    file, err := os.Open(filePath)
    if err != nil {
        return err
    }
    defer file.Close()
    ……
    return nil
}
```

在退出函数ReadFile前，defer后的语句会被执行。

# Go语言基础——defer（续）

defer后也可以是一个匿名函数：

```
func ReadFile(filePath string) error {
    file, err := os.Open(filePath)
    if err != nil {
        return err
    }
    defer func() {
        file.Close()
    }()
    ……
    return nil
}
```

# Go语言基础——异常处理

panic：

```
func ReadInputs(exitMark string, buffer bytes.Buffer) {
    end := false
    reader := bufio.NewReader(os.Stdin)
    fmt.Println("Please input:\n")
    for !end {
        line, err := reader.ReadString('\n')
        if err != nil {
            panic(err)
        }
        if (exitMark + "\n") == line {
            break
        }
        buffer.WriteString(line)
    }
}
```

普通错误常常被作为返回值，而不是被抛出。

如果你认为某类错误或异常是不可容忍的，甚至需要终止程序，那么你可以用panic制造一个“恐慌”并附上错误信息！

# Go语言基础——异常处理（续）

recover：

当然，你可以在调用处添加“保护层”。这是defer的另一个常用法。

```
func main() {
    defer func() {
        if err := recover(); err != nil {
            debug.PrintStack()
            fmt.Printf("Fatal Error: %s\n", err)
        }
    }()
    var buffer bytes.Buffer
    ReadInput("exit", buffer)
    fmt.Printf("Inputs: %s\n", buffer.String())
}
```

如果“恐慌”发生了，我们可以“平息”它，以防程序终止。

我们为致命的异常打印一下调用栈吧。

panic不一定会被调用方recover，只有在确认有必要的时候才应该这么做！

# Go语言基础——结构体与方法

一个简单的struct以及它的一个方法：

```
type Dept struct {
    name string
    building string
    floor uint8
}

func (self Dept) Name() string {
    return self.name
}
```

是不是有些眼熟？（如果你写过Python代码的话）。这里也相当于Java中的“this”。

# Go语言基础——结构体与方法（续）

再看看这个struct的其他几个方法：

```
func (self Dept) SetName(name string) {
    self.name = name
}

func (self *Dept) Relocate(building string,
floor uint8) {
    self.building = building
    self.floor = floor
}
```

注意这个星号！这意味将Dept实例的指针赋值给了“self”，后面我们将会看到它们的不同之处。

# Go语言基础——结构体与方法（续2）

struct的使用方法：

```
dept1 :=
    Dept{
        name: "MySohu",
        building: "Internet",
        floor: 7,
    }
fmt.Printf("dept (1): %v\n", dept1)
// => dept (1): {MySohu Internet 7}
dept1.Relocate("Media", 12)
fmt.Printf("dept (3): %v\n", dept1)
// => dept (2): {MySohu Media 12}
```

# Go语言基础——结构体与方法（续3）

struct方法中的传值与传引用（指针）：

```
dept1.SetName("Other")
fmt.Printf("dept (3): %v\n", dept1)
// => dept (3): {MySohu Media 12}
```

看这里，说明SetName方法没起作用，为什么？

回顾一下两个设置方法的签名：

```
func (self Dept) SetName…

func (self *Dept) Relocate…
```

说明：

- “(self Dept)”相当于把本Dept实例的副本赋值给了“self”。
- “(self *Dept)”相当于把本Dept实例的指针的副本赋值给了“self”。

# Go语言基础——接口


interface：

```
type DeptModeFull interface { //包含了结构Dept及其指针上的所有方法
    Name() string
    SetName(name string)
    Relocate(building string, floor uint8)
}

type DeptModeA interface { //仅包含了结构Dept上的方法
    Name() string
    SetName(name string)
}

type DeptModeB interface { //仅包含了结构Dept的指针上的方法
    Relocate(building string, floor uint8)
}
```

# Go语言基础——接口（续）

结构Dept实例实现了哪个接口：

```
dept1 :=
    Dept{
        name: "MySohu",
        building: “Media",
        floor: 7}
switch v := interface{}(dept1).(type) {
case DeptModeFull:
    fmt.Printf("The dept1 is a DeptModeFull.\n")
case DeptModeB:
    fmt.Printf("The dept1 is a DeptModeB.\n")
case DeptModeA:
    fmt.Printf("The dept1 is a DeptModeA.\n")
default:
    fmt.Printf("The type of dept1 is %v\n", v)
} // => The dept1 is a DeptModeA.
```

# Go语言基础——接口（续2）

结构Dept实例的指针实现了哪些接口：

```
deptPtr1 := &dept1
if _, ok := interface{}(deptPtr1).(DeptModeFull); ok {
    fmt.Printf("The deptPtr1 is a DeptModeFull.\n")
}
if _, ok := interface{}(deptPtr1).(DeptModeA); ok {
    fmt.Printf("The deptPtr1 is a DeptModeA.\n")
}
if _, ok := interface{}(deptPtr1).(DeptModeB); ok {
    fmt.Printf("The deptPtr1 is a DeptModeB.\n")
}
// => The deptPtr1 is a DeptModeFull.
// => The deptPtr1 is a DeptModeA.
// => The deptPtr1 is a DeptModeB.
```

# Go语言基础——接口（续3）

为什么deptPtr1被判定为全部三个接口DeptModeFull、DeptModeA和DeptModeB的实现？而dept1只实现了接口DeptModeA？

依据Go语言规范：

- 结构Dept的方法集中仅包含方法接收者为Dept的方法，即：Name()和SetName()。所以，结构Dept的实例仅为DeptModeA的实现。
- 结构的指针*Dept的方法集包含了方法接受者为Dept和*Dept的方法，即：Name()、SetName()和Relocate()。所以，接口Dept的实例的指针为全部三个接口—DeptModeFull、DeptModeA和DeptModeB的实现。

# Go语言基础——接口（续4）

继续延伸：调用方法时发生的隐形转换：

```
dept1.Relocate("Media", 12)
fmt.Printf("Dept: %v\n", dept1)
fmt.Printf("Dept name: %v\n", deptPtr1.Name())

// => Dept: {MySohu Media 12 }
// => Dept name: MySohu
```

？ 那为什么结构Dept的实例却可以调用其指针方法集中的方法？

依据Go语言规范：

- 如果结构的实例x是“可被寻址的”，且&x的方法集中包含方法m，则x.m()为(&x).m()的速记（快捷方式）。

即：dept1是可被寻址的，且&dept1的方法集中包含方法Relocate()，则dept1.Relocate()为&dept1.Relocate()的快捷方式。

# To be continue...

Talk is cheap, show me the code!