a8m/golang-cheat-sheet的中文版本

English

golang-cheat-sheet是目前GitHub上最流行的golang代码速查表。

作者Ariel Mashraki也是Facebook著名ORM框架ent(2019年开源)的作者和首席布道师。

本文是该速查表的中文版本，会根据原版实时更新。

版本

- 当前更新版本：2021-04-04 版本地址：commit#master
- 如果您发现更新、问题或改进，欢迎随时提issue或PR。
- please feel free to open an issue or PR if you find any updates, issues or improvement.

目录

致谢

大多数代码示例来源于A Tour of Go，它对Go做了非常棒的介绍。
很认真地说，如果你是Go新手，一定要看完A Tour of Go。

Go特性一览

- 命令式编程语言
- 静态类型
- 语法类似C语言(但是和C语言相比，圆括号更少，没有分号)，结构类似Oberon-2编程语言
- 编译成机器代码(没有JVM)
- 没有类，但是有可以带方法的结构体
- 接口类型Interfaces
- 没有继承，但是有类型嵌套
- 函数是一等公民
- 函数可以返回多个值
- 有闭包
- 有指针，但是没有指针运算
- 内置并发原语：协程goroutine和管道channel

基础语法

Hello World

文件 `hello.go`:

```
package main

import "fmt"

func main() {
    fmt.Println("Hello Go")
}
```

`$ go run hello.go`

操作符

算术操作符

Operator	Description
`+`	加
`-`	减
`*`	乘
`/`	除
`%`	取余
`&`	按位与
`|`	按位或
`^`	按位异或，这个也可以当做”按位取反“操作符
`&^`	bit clear (and not)操作，x&^y表示把y按位取反，再和x做位与&操作，也即x&(^y)
`<<`	按位左移
`>>`	按位右移

比较操作符

Operator	Description
`==`	等于
`!=`	不等于
`<`	小于
`<=`	小于等于
`>`	大于
`>=`	大于等于

逻辑操作符

Operator	Description
`&&`	逻辑与
`||`	逻辑或
`!`	逻辑非

其它

Operator	Description
`&`	取变量地址或者创建指针
`*`	取指针指向的变量的值
`<-`	管道channel的发送和接收操作符

声明

类型在标识符后面

```
var foo int // 只声明，不做初始化
var foo int = 42 // 声明的同时做初始化
var foo, bar int = 42, 1302 // 一次声明和初始化多个变量
var foo = 42 // 忽略类型，编译器自行推导
foo := 42 // 简写，只能在函数或者方法体内使用，没有var关键字，变量类型也是隐式推导而来
const constant = "This is a constant"

// iota的值从0开始，用于常量的数值递增
const (
    _ = iota
    a
    b
    c = 1 << iota
    d
)
fmt.Println(a, b) // 1 2 (0被赋值给了_，相当于被跳过了)
fmt.Println(c, d) // 8 16 (2^3, 2^4)
```

函数

```
// 一个简单的函数
func functionName() {}

// 带参数的函数，参数的类型在标识符后面
func functionName(param1 string, param2 int) {}

// 多个参数有相同的类型
func functionName(param1, param2 int) {}

// 返回值类型声明
func functionName() int {
    return 42
}

// 可以返回多个值
func returnMulti() (int, string) {
    return 42, "foobar"
}
var x, str = returnMulti()

// 函数返回值有标识符，可以在函数体内对返回标识符赋值
```

```
func returnMulti2() (n int, s string) {
    n = 42
    s = "foobar"
    // 只需要return即可，n和s的值会被返回
    return
}
var x, str = returnMulti2()
```

函数作为值和闭包

```
func main() {
    // 把函数赋值给变量add，add是一个函数类型变量
    add := func(a, b int) int {
        return a + b
    }
    // 使用函数变量来调用函数
    fmt.Println(add(3, 4))
}

// 闭包是匿名函数，闭包可以访问当前作用域可以访问到的变量
func scope() func() int{
    outer_var := 2
    foo := func() int { return outer_var}
    return foo
}

func another_scope() func() int{
    // 编译失败，因为outer_var和foo没有在another_scope里定义
    outer_var = 444
    return foo
}


// 闭包
func outer() (func() int, int) {
    outer_var := 2
    inner := func() int {
        outer_var += 99 // 如果执行了闭包，闭包外面的outer_var的值会被修改
        return outer_var
    }
    inner()
    return inner, outer_var // outer_var的值被改变，这里返回inner函数和101
}
```

参数可变的函数

```
func main() {
    fmt.Println(adder(1, 2, 3))      // 6
    fmt.Println(adder(9, 9))    // 18

    nums := []int{10, 20, 30}
    fmt.Println(adder(nums...)) // 60
}

// 在最后一个参数的类型前面加...表示函数的最后一个传参可以有0个或者多个
```

```go
// 函数调用和普通函数一样，只是我们可以传递任意多个参数
func adder(args ...int) int {
    total := 0
    for _, v := range args { // 遍历传进来的参数，args是一个slice类型变量
        total += v
    }
    return total
}
```

内置类型

```go
bool

string

int  int8  int16  int32  int64
uint uint8 uint16 uint32 uint64 uintptr

byte // uint8的别名

rune // int32的别名，主要用来表示字符类型

float32 float64

complex64 complex128
```

Go所有的内置类型都被定义在标准库的builtin这个包里。

类型转换

```go
var i int = 42
var f float64 = float64(i)
var u uint = uint(f)

// 下面这种语法也可以，用于局部作用域
i := 42
f := float64(i)
u := uint(f)
```

包

- 在每个Go源文件的最开头(不包括注释)添加包声明
- 可执行文件在main包里
- 惯例：包名 == 包导入路径的最后一个名称(导入路径 `math/rand` => 包名 `rand`)
- 大写字母开头的标识符: 表示可导出(exported)标识符， 对其它包可见
- 小写字母开头的标识符: 表示私有(private)标识符，对其它包不可见

控制结构

If

```
func main() {
    // 基本语法
    if x > 10 {
        return x
    } else if x == 10 {
        return 10
    } else {
        return -x
    }

    // 在if条件前面可以加一条代码语句
    if a := b + c; a < 42 {
        return a
    } else {
        return a - 42
    }

    // 在if里做类型判断
    var val interface{} = "foo"
    if str, ok := val.(string); ok {
        fmt.Println(str)
    }
}
```

循环

```
    // Go只有for，没有while和until关键字
    for i := 1; i < 10; i++ {
    }
    for ; i < 10;  { // 相当while循环的效果
    }
    for i < 10  { // 如果只有一个条件，可以省略分号，也相当于while循环
    }
    for { // 可以忽略条件，相当于while (true)
    }

    // 循环里可以使用break/continue来控制循环执行逻辑
    // break/continue还可以和循环外的label一起使用，用于控制外层循环的执行逻辑
    // continue here表示外层的for循环继续执行，继续执行时外层for循环里的i会++
    // break there表示退出外层循环，也就是退出整个循环了
here:
    for i := 0; i < 2; i++ {
        for j := i + 1; j < 3; j++ {
            if i == 0 {
                continue here
            }
            fmt.Println(j)
            if j == 2 {
                break
            }
        }
    }

there:
```

```
    for i := 0; i < 2; i++ {
        for j := i + 1; j < 3; j++ {
            if j == 1 {
                continue
            }
            fmt.Println(j)
            if j == 2 {
                break there
            }
        }
    }
```

Switch

```
// switch语句
switch operatingSystem {
case "darwin":
    fmt.Println("Mac OS Hipster")
    // case分支里的代码执行完后会自动退出switch，默认没有fallthrough
case "linux":
    fmt.Println("Linux Geek")
default:
    // Windows, BSD, ...
    fmt.Println("Other")
}

// 和if一样，switch的value之前可以添加一条赋值语句
switch os := runtime.GOOS; os {
case "darwin": ...
}

// switch的case条件还可以是比较语句
number := 42
switch {
    case number < 42:
        fmt.Println("Smaller")
    case number == 42:
        fmt.Println("Equal")
    case number > 42:
        fmt.Println("Greater")
}

// case分支后还可以带多个值，用逗号分隔，任意一个匹配即可
var char byte = '?'
switch char {
    case ' ', '?', '&', '=', '#', '+', '%':
        fmt.Println("Should escape")
}
```

数组，切片和range迭代

数组

```
var a [10]int // 声明一个长度为10的int数组，数组长度也是数组类型的一部分
a[3] = 42     // 设置数组元素的值
i := a[3]     // 读数组元素的值

// 声明和初始化
var a = [2]int{1, 2}
a := [2]int{1, 2} //简写
a := [...]int{1, 2} // 编译器自行推导数组长度
```

切片

```
var a []int                              // 声明切片，和数组类型声明类似，不需要指定长度
var a = []int {1, 2, 3, 4}               // 声明和初始化切片
a := []int{1, 2, 3, 4}                    // 简写
chars := []string{0:"a", 2:"c", 1: "b"}  // ["a", "b", "c"]

var b = a[lo:hi]    // 通过下标索引从已有的数组或切片创建新切片，下标前闭后开，取值从lo到
hi-1
var b = a[1:4]      // 取切片a的下标索引从1到3的值赋值给新切片b
var b = a[:3]       // :前面没有值表示起始索引是0，等同于a[0:3]
var b = a[3:]       // :后面没有值表示结束索引是len(a)，等同于a[3:len(a)]
a =  append(a,17,3) // 往切片里添加新元素
c := append(a,b...) // 把切片a和b的值拼接起来，组成新切片

// 使用make来创建切片
a = make([]byte, 5, 5)  // make的第2个参数是切片长度，第3个参数是切片容量
a = make([]byte, 5) // 第3个切片容量参数可选，即可以不传值

// 根据数组来创建切片
x := [3]string{"Лайка", "Белка", "Стрелка"}
s := x[:] // 切片s指向了数组x的内存空间，改变切片s的值，也会影响数组x的值
```

数组和切片上的操作

`len(a)`可以用来计算数组或切片的长度，len()是Go的内置函数，不是数组或者切片的方法

```
// 循环遍历数组或切片
for i, e := range a {
    // i是下标索引，从0开始，e是具体的元素
}

// 如果你只需要元素，不需要下标索引，可以按照下面的方式做:
for _, e := range a {
    // e是元素
}

// 如果你只需要下标索引，可以按照下面的方式做
for i := range a {
}

// Go 1.4之前，如果range的前面不按照上面2个示例那样带上i和e，会编译报错
// Go 1.4开始，可以不用带上i和e，直接for range遍历
for range time.Tick(time.Second) {
```

```
    // 每秒执行一次
}
```

集合

```
m := make(map[string]int)
m["key"] = 42
fmt.Println(m["key"])

delete(m, "key")

elem, ok := m["key"] // 判断key是否存在：如果存在，ok就是true，elem是对应value，否则ok是false，elem是map的value的类型的零值

// map字面值，声明的同时做初始化
var m = map[string]Vertex{
    "Bell Labs": {40.68433, -74.39967},
    "Google":    {37.42202, -122.08408},
}

// 遍历map
for key, value := range m {
}
```

结构体

Go没有class，只有结构体struct，结构体可以有自己的方法。

```
// 结构体是一种类型，也是一系列字段的集合

// 声明
type Vertex struct {
    X, Y int
}

// 创建结构体变量
var v = Vertex{1, 2}
var v = Vertex{X: 1, Y: 2} // 通过字段名称:值的形式来创建结构体变量
var v = []Vertex{{1,2},{5,2},{5,5}} // 初始化结构体切片

// 访问结构体的字段
v.X = 4

// 给结构体定义方法，在func关键字和方法名称之间加上结构体声明(var_name StructName)即可
// 调用方法时，会把结构体的值拷贝一份
func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

// 调用结构体方法
v.Abs()

// 如果想调用方法时改变外部结构体变量的值，方法需要使用指针接受者
// 下面的方法，每次调用add方法时就不会拷贝结构体的值
```

```
func (v *Vertex) add(n float64) {
    v.X += n
    v.Y += n
}
```

匿名结构体:
比使用`map[string]interface{}`更轻量、更安全。

```
point := struct {
    X, Y int
}{1, 2}
```

指针

```
p := Vertex{1, 2}  // p是结构体Vertex的变量或者说实例
q := &p            // q是指向Vertex的指针
r := &Vertex{1, 2} // r也是指向Vertex的指针

// Vertex指针的类型是*Vertex

var s *Vertex = new(Vertex) // new函数创建一个指向Vertex实例的指针
```

接口

```
// 接口声明
type Awesomizer interface {
    Awesomize() string
}

// 结构体不会在声明的时候指定要实现某个接口
type Foo struct {}

// 相反，结构体如果实现了接口里的所有方法，那就隐式表明该结构体满足了该接口
// 可以通过接口变量来调用结构体方法
func (foo Foo) Awesomize() string {
    return "Awesome!"
}
```

接口和结构体嵌套

Go没有子类的概念，不过Go有接口嵌套和结构体嵌套。

```
// 接口嵌套，ReadWriter的实现一定要同时实现Reader和Writer这2个接口类型里的所有方法
type ReadWriter interface {
    Reader
    Writer
}

// 结构体嵌套，Server同时有了log.Logger的所有方法
type Server struct {
    Host string
    Port int
    *log.Logger
```

```
}

// 初始化嵌套结构体变量，和普通结构体初始化一样
server := &Server{"localhost", 80, log.New(...)}

// 被嵌套的结构体log.Logger的方法也自然成为了结构体Server的方法
server.Log(...) // 相当于调用了server.Logger.Log(...)

// 被嵌套的类型的字段名称是它的类型名称(在本代码示例里，被嵌套的类型*log.Logger的字段名称是它的类型名称Logger)
var logger *log.Logger = server.Logger
```

错误处理

Go没有异常处理。函数如果可能产生错误只需要在函数返回值里额外增加一个类型为error的返回值。error接口类型的定义如下:

```
// error接口类型是Go内置类型，用于表示错误
// 值为nil时表示没有错误
type error interface {
    Error() string
}
```

这里有一个示例:

```
func sqrt(x float64) (float64, error) {
    if x < 0 {
        return 0, errors.New("negative value")
    }
    return math.Sqrt(x), nil
}

func main() {
    val, err := sqrt(-1)
    if err != nil {
        // 处理错误
        fmt.Println(err) // negative value
        return
    }
    // 没有错误，打印结果
    fmt.Println(val)
}
```

并发

协程Goroutine

Goroutines是轻量级线程(由Go运行时来管理，不是操作系统线程)。 `go f(a, b)` 语句会开启了一个新的goroutine，这个goroutine执行 `f(a, b)` 这个函数调用。

```go
// 这里只是定义一个函数，后面用于goroutine执行
func doStuff(s string) {
}

func main() {
    // 在goroutine里使用有名称的函数
    go doStuff("foobar")

    // 在goroutine里使用匿名函数(闭包)
    go func (x int) {
        // 函数体定义
    }(42)
}
```

管道Channel

```go
ch := make(chan int) // 创建类型为int的管道
ch <- 42             // 发送数据到管道ch
v := <-ch            // 从管道ch接收数据

// 没有缓冲区的管道会阻塞。
// 如果没有往管道发送值，读操作会阻塞，如果没有从管道接收值，写操作会阻塞

// 创建带缓冲区的管道
// 如果缓冲区未满，往有缓冲区的管道发送数据不会阻塞
ch := make(chan int, 100)

close(ch) // 关闭管道(只有往管道发送数据的发送者才应该执行close操作)

// 从管道读数据，并判断管道是否已经被关闭
v, ok := <-ch

// 如果ok是false就表示管道已经被关闭了

// 从管道读数据，直到管道被关闭
for i := range ch {
    fmt.Println(i)
}

// select关键字在多个管道操作上阻塞，只要有1个不阻塞了，对应case分支就会被执行
func doStuff(channelOut, channelIn chan int) {
    select {
    case channelOut <- 42:
        fmt.Println("We could write to channelOut!")
    case x := <- channelIn:
        fmt.Println("We could read from channelIn")
    case <-time.After(time.Second * 1):
        fmt.Println("timeout")
    }
}
```

Channel原则

- 给值为nil的管道发送数据会一直阻塞

```
var c chan string
c <- "Hello, World!"
// fatal error: all goroutines are asleep - deadlock!
```

- 从值为nil的管道接收数据会一直阻塞

```
var c chan string
fmt.Println(<-c)
// fatal error: all goroutines are asleep - deadlock!
```

- 往被关闭的管道发送数据会panic

```
var c = make(chan string, 1)
c <- "Hello, World!"
close(c)
c <- "Hello, Panic!"
// panic: send on closed channel
```

- 从被关闭的管道接收数据会立即返回零值

```
var c = make(chan int, 2)
c <- 1
c <- 2
close(c)
for i := 0; i < 3; i++ {
    fmt.Printf("%d ", <-c)
}
// 1 2 0
```

打印

```
fmt.Println("Hello, 你好, नमस्ते, Привет, 🌍🌍🌍") //基本的打印，会自动换行
p := struct { X, Y int }{ 17, 2 }
fmt.Println( "My point:", p, "x coord=", p.X ) // 打印结构体和字段值
s := fmt.Sprintln( "My point:", p, "x coord=", p.X ) // 打印内容到字符串变量里

fmt.Printf("%d hex:%x bin:%b fp:%f sci:%e",17,17,17,17.0,17.0) // C风格的格式化打印
s2 := fmt.Sprintf( "%d %f", 17, 17.0 ) // 字符串格式化

hellomsg := `
 "Hello" in Chinese is 你好 ('Ni Hao')
 "Hello" in Hindi is नमस्ते ('Namaste')
` // 跨越多行的字符串，使用``
```

反射

类型Switch

类型switch类似普通的switch语句，只是case分支的判断条件是类型，而不是具体的值。使用场景主要是用于判断接口变量的值类型。

```
func do(i interface{}) {
    switch v := i.(type) {
    case int:
        fmt.Printf("Twice %v is %v\n", v, v*2)
    case string:
        fmt.Printf("%q is %v bytes long\n", v, len(v))
    default:
        fmt.Printf("I don't know about type %T!\n", v)
    }
}

func main() {
    do(21)
    do("hello")
    do(true)
}
```

代码片段

文件嵌入

Go程序可以使用embed包嵌入静态文件

```
package main

import (
    "embed"
    "fmt"
    "io"
    "log"
    "net/http"
)

// content持有服务器static目录下的所有文件
//go:embed static/*
var content embed.FS

func main() {
    http.Handle("/", http.FileServer(http.FS(content)))
    go func() {
        log.Fatal(http.ListenAndServe(":8080", nil))
    }()
    // 读取服务器static目录下的内容
    entries, err := content.ReadDir("static")
    if err != nil {
        log.Fatal(err)
    }
    for _, e := range entries {
        resp, err := http.Get("http://localhost:8080/static/" + e.Name())
        if err != nil {
```

```
            log.Fatal(err)
        }
        body, err := io.ReadAll(resp.Body)
        if err != nil {
            log.Fatal(err)
        }
        if err := resp.Body.Close(); err != nil {
            log.Fatal(err)
        }
        fmt.Printf("%q: %s", e.Name(), body)
    }
}
```

Go Playground代码示例

HTTP服务器

运行下面的代码，在浏览器访问 http://127.0.0.1:4000 会显示"hello"

```
package main

import (
    "fmt"
    "net/http"
)

// 定义处理http请求的Handler
type Hello struct{}

// 结构体Hello实现接口类型http.Handler里的方法ServeHTTP
// 这样结构体Hello的实例就可以作为http的Handler来接收http请求，返回http响应结果
func (h Hello) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "Hello!")
}

func main() {
    var h Hello
    http.ListenAndServe("localhost:4000", h)
}

// 下面是http.ServerHTTP的方法签名
// type Handler interface {
//     ServeHTTP(w http.ResponseWriter, r *http.Request)
// }
```