

# Golang


Google

**docker** **k8s** **go**

- **인터프리터**
  - **인터프리터**
  - **비트코인**
  - **인터프리터**
- **인터프리터**
  - **인터프리터**
  - **인터프리터**
  - **인터프리터**
  - **인터프리터 CPU**
- **인터프리터**
  - **runtime 인터프리터**
    - **인터프리터**
    - **인터프리터 GC**
    - **인터프리터**
  - **인터프리터**
- **인터프리터**
  - **25 인터프리터**
  - **인터프리터**
  - **인터프리터**
  - **인터프리터**

--	--	--	--	--	--	--	--	--



 image-20251011122440219

# Golang

111

□ □ □ □ □ □ □ □ □

- `go run` `main.go`
  - `main()` `main()`

- `go build` 编译go程序
- `-o` 指定输出文件
- `go version` 查看go版本
- `go get path` 安装G 包
- `go env` 查看环境

编译

## Hello World

```
package main // 包名

import "fmt" // 导入包

// 主函数
import(
    "fmt"
    "time"
)

// 主函数
func main(){
    fmt.Println("Hello World")
}
```

编译

- `package main` 包名
- `package main` 包名
- `.a` 包名
- `main` 包名
- 编译选项
- `package` 包名

编译

- `import` 包名 `fmt` 包名
- `import` 包名
- `import` 包名
- `go` 编译选项
- 编译选项
- `init()` 包名

image-20251011171819781

- `_` 包名 `init()` 包名
- 包名
- `.` 包名

```

package main

import(
    _ "./lib1"
    mylib2 "./lib2"
    . "./lib3"
)

func main(){
    // 测试lib2
    mylib2.Lib2Test()

    // 测试lib3
    Lib3Test()
}

```

测试

- 测试lib2和lib3的函数

数据类型

```

// 布尔
bool

// 字符串
string

// 整数
int int8 int16 int32 int64
uint uint8 uint16 uint32 uint64 uintptr

// uint8的别名
byte

// int32的别名Unicode的别名
rune

// 浮点数
float32 float64

// 复数
complex64 complex128

// 字符串字面量"abc" `abc`
string

```

格式

- `%v` 打印

- `%T` 打印类型
- `%t` 打印 `true` 或 `false`
- `%d` 打印十进制
- `%X` 打印十六进制
- `%b` 打印二进制
- `%c` 打印 Unicode 码点
- `%S` 打印字符串
- `%q` 打印 JSON 字符串
- `%p` 打印内存地址
- `%f` 打印浮点数
- `%e` 打印科学计数法
- `%g` 打印科学计数法或浮点数 `%e` 或 `%f`
- `%#v` 打印 Go 类型名称
- `%#x` 打印十六进制 0x 前缀

变量

变量声明

```
func main(){
    // 变量
    var a int
    // 常量
    var b int = 100
    // 数组
    var b,c int = 1,2
    var bb,cc = 100,"zxb"
    var(
        bbb int = 100
        ccc string = "zxb"
    )
    // 常量
    var c = 100
    // 变量var
    e := 100
}
```

**var**

- 变量 `var` 声明变量
- 变量 `var` 声明变量
- 变量 `var` 声明变量 **0**

- `fmt` 包中的 `Printf` 函数
  - `fmt.Printf("type of a = %T",a)` 输出 `%T`
- 常量表达式
- 常量表达式只能用于常量表达式

`:=`

- `:=` 赋值语句
- 常量表达式
- 常量表达式只能用于常量表达式
- `:=` 赋值语句

常量

```
const(
    BEIJING = 10*iota // iota=0
    SHANGHAI      // iota=1
    SHENZHEN      // iota=2
)
func main(){
    const length int = 10
}
```

`const`

- 常量表达式
- 常量表达式
- 常量表达式只能用于常量表达式
  - `iota` 常量表达式

常量

```
import "fmt"

// 常量
func fool1(a string,b int) int{
    fmt.Println("a = ",a)
    fmt.Println("b = ",b)

    c := 100
    return c
}

// 常量表达式
func fool1(a string,b int) (int,int){
    return 666,777
}
```

常量

- `go` 開發環境
  - 安裝開發環境
- 變數與常數
- 資料型態
- 算術運算與比較運算
- 條件 \_ 迴圈

變數

- 變數的宣告與使用
  - 變數的宣告
- 變數的 `return` 值

```
func split(sum int) (x, y int){
    x = sum*4/9
    y = sum-x
    return
}
```

陣列

C語言的陣列

- 陣列的宣告與使用 \* 陣列的初始化
- 陣列的 & 地址

```
package main

import "fmt"

func add(n int) {
    n += 2
}

func addptr(n *int){
    // 陣列的宣告與使用
    *n += 2
}

func main(){
    p := 5
    add(p)
    fmt.Println(p) // p = 5
    addptr(&p)
    fmt.Println(p) // p = 7
}
```

條件

- `if` 條件判斷
- `if-else` 條件判斷

- `if v > 10{ work() }` ; `if v > 10 work()`

```
// 1
if v > 10 work()
if v > 10{ work() }

// 2
if v > 10{
    work()
}

if st:=0 ;v > 10{
    st = 1
    work()
}
```

## switch

- `switch` 语句
- `switch` 的 `case` 语句
- `switch` 语句中的 `break`
- `fallthrough` 语句

```
switch{
case t < 12:
    fmt.Println("")
default:
    fmt.Println("")
}
```

## for

- `go` 语言中的 `for`
- `continue` 和 `break` 语句

```
for{
    // ...
}

for j:=7;j < 9;j++){
    continue
    break
}

i:=1
for i<=3{
    ++i
}
```

## defer

- defer 语句只能用在函数内部
- defer 语句只能用在函数内部
- 多个 defer 语句会按照从后到前的顺序依次执行
- defer 语句只能用在函数内部

```
import "fmt"
func main(){
    defer fmt.Println("world")

    fmt.Println("hello")
} // 输出 hello world
```

## Slice

切片是 Go 语言中一种特殊的数组类型

切片

- var 切片 切片类型 {切片} 切片初始化
- 切片 := 切片

```
var myArray1 [10]int
myArray2 := [10]int
```

- len(切片) 切片长度

```
var myArray1 [10]int

for i:=1; i < len(myArray1);i++){
    fmt.Println(i)
}
```

- range 切片遍历切片 index 切片 value 切片
- \_ 切片遍历切片 \_ 切片

```
// 切片遍历
var myArray1 [10]int

// range 切片
for index,value := range myArray1{
    fmt.Println("index = ",index,"value=",value)
}

// 切片遍历
for _,value := range myArray1{
    fmt.Println("value=",value)
}
```



- 常量的使用
  - 常量的使用

```
// 常量
func method(arr [5]int){

}

// 常量
func method(arr [4]int){

}

func main(){
    arr := [5] int
    method(arr)
}
```

常量

- 常量的使用
  - 常量的使用
  - 常量 nil 常量 make 常量
    - 常量的使用
  - 常量 make 常量
  - 常量 := 常量 make 常量

```
// 常量
slice1 := [int]{1,2,3}

// 常量slice常量slice常量
var slice1 []int
slice1 = make([]int,3)

// 常量`make`常量
var slice1 []int = make([]int,3)

// 常量:=常量
var slice1 := make([]int,3)
```

- 常量的使用

```
// 常量
func modifySlice(s []int) {
    s[0] = 100 // 常量
}

func main() {
```

```
a := []int{1, 2, 3, 4, 5} // 00000000
modifySlice(a)
fmt.Println(a[0]) // 00 100
}
```

- nil [] nil 0

```
func main() {  
    var phone []int // nil  
}
```

- 列表
  - 列表 make 列表
  - 列表 append 列表 `a = append(a,value)`
    - 列表 append 列表
    - 列表

```
// 初始化
var numbers = make([]int,3,5)

// 添加
numbers = append(numbers,1)
```

- 字符串
  - `s[i:]` 从i开始
  - `s[:j]` 到j结束(不含j)
  - 字符串切片时, 超出范围也不会报错
  - 字符串 `copy()` 方法
    - `copy(s1,s2)` 将 `s2` 复制到 `s1`

```
s := []int{1,2,3}

// s1[] []2
s1 = s[0:2]
```

## Map

## Map

- [] 通过 key 来访问 value 值
  - 通过 make 来初始化
  - 通过 := 来赋值
  - 通过删除操作
    - 通过 delete 来删除
- 通过 key 来访问 value 值

```
// 1. map
var myMap1 map[string]string
// 2. make
myMap1 = make(map[string]string, 10)
// 3. set
myMap1["one"] = "php"
myMap2["two"] = "js"
myMap3["three"] = "go"

// 4. range
var myMap2 := make(map[int]string, 10)

// 5. map[string]string
myMap3 := map[string]string{
    "one": "php",
    "two": "js",
    "three": "go"
}
```

## Map 操作

- 遍历 range 遍历

```
myMap3 := map[string]string{
    "one": "php",
    "two": "js",
    "three": "go"
}

for key, value := range myMap3{
    fmt.Println("key = ", key)
    fmt.Println("value = ", value)
}
```

- delete 删除
  - map 删除
  - key 删除

```
myMap3 := map[string]string{
    "one": "php",
    "two": "js",
    "three": "go"
}

delete(myMap3, "one")
```

- key 删除

```
myMap3 := map[string]string{
    "one": "php",
    "two": "js",
    "three": "go"
}

myMap3["one"] = "python"
```

- `map` 자료형

## Struct

구조체

```
type Person struct{
    Name string
    Age int
}
```

변수 선언

- `var` 키워드 사용

```
var p person
p.name = "jhwang"
p.age = 20
```

함수

- 구조체 변수를 인자로 전달
- 구조체 포인터를 인자로 전달

```
// 구조체
func changeStruct(person Person){
    // ...
}

func main(){
    var p person
    p.name = "jhwang"
    p.age = 20
    changeStruct(person)
}

// 구조체 포인터
func changeStruct(person *Person){
    // ...
}
```

```
func main(){
    var p person
    p.name = "jhwang"
    p.age = 20
    changeStruct(&person)
}
```

5/5

- [illegible]

```
type resume struct{
    Name string `info:name` `doc:姓名`
    Sex  string `info:sex`
}
```

11

□□□□□□□□

- [illegible]

555

- $\{ \}$   $\{ \}$

```
type Person struct{
    Name string
    Age  int
}

person := Person{name: "Alice", age: 25}
```

5/5

- `__init__` this `__dict__`
  - this `__dict__`
  - `__dict__`

```
// 000
func (this Person) SayHello() {
    fmt.Printf("Hello, my name is %s\n", this.name)
}

person.SayHello()

// 0000
func (this *Person) SayHello() {
    fmt.Printf("Hello, my name is %s\n", this.name)
```

```
}
person.SayHello()
```

11

1111

- □□□□□□□□□□□□
- □□□□□□□□□□□□

```
type Human struct{
    name string
    sex string
}

func (this *Human) Eat(){
    // ...
}

// 测试
type SuperMan struct{
    Human // SuperMan包含Human的所有成员
    level int
}

// 测试
func (this *SuperMan) Eat(){
    // ...
}
```

11

11

- interface 接口
  - 数据类型
  - 方法
  - 常量
    - 静态常量
    - 非静态常量

```
type Animal interface {  
    Speak()  
}  
  
type Cat struct{}  
  
type Dog struct{}
```

```
// 动物接口
func (d *Dog) Speak() {
    fmt.Println("Woof")
}
// 动物接口
func (c Cat) Speak() {
    fmt.Println("Meow")
}

func main() {
    var a Animal

    // 动物接口实现
    a = Cat{}      // 猫
    a = &Cat{}     // 猫Go 语言实现
    a = &Dog{}     // 狗
    a = Dog{}      // 狗Dog 语言实现 Animal接口 *Dog 实现
```

接口

- 动物接口实现
- 动物接口实现

```
type AnimalIF interface{
    Sleep()
    GetColor() string
    GetType() string
}

func ShowAnimal(animal AnimalIF){
    // ...
}

// 动物接口
type Cat struct{
    color string
}

// 猫接口

func (this *Cat) Sleep(){
    // ...
}

func (this *Cat) GetColor() string{
    // ...
}

func (this *Cat) GetType() string{
```

```

    // ...
}

type Dog struct{
    color string
}

func (this *Dog) Sleep(){
    // ...
}

func (this *Dog) GetColor() string{
    // ...
}

func (this *Dog) GetType() string{
    // ...
}

func main(){
    // 测试
    var animal AnimalIF

    // 测试
    animal = &Cat{"green"}
    animal.Sleep()
    fmt.Println(animal.GetColor())
    fmt.Println(animal.GetType())

    // 测试
    animal = &Dog{"blue"}
    animal.Sleep()
    fmt.Println(animal.GetColor())
    fmt.Println(animal.GetType())
}

```

测试

- 测试
- 测试 `x.(T)` 与 `x` 类型 `T` 测试
  - 测试 `x.(T)` 与 `x` 类型 `T` 测试
  - 测试
  - 测试 `value` 与 `ok`
    - 测试 `ok` 与 `true` 与 `value` 测试

```

// 测试
func MyFunc(arg interface{}){
    // ...
    // 测试
    value,ok = arg.(string)
}

```



```

}

type book struct{
    // ...
}

func main(){
    book := Book{}
    // 構造関数book構造体
    MyFunc(book)
}

```

00

```
pair <
```

- `type`
  - `static type` `int`
  - `concrete type` `1`
- `value`
- `pair` `(1, 2)`

```
var a string
a = "aceld"

var allType interface{}
// allType[]value[]type[]a[]
allType = a
```

00

- `reflect`
  - `ValueOf()` 構造関数
  - `TypeOf()` 型取得関数
- 構造体
- 配列
  - 宣言
  - `.NumField()` フィールド数
  - `.Field()` フィールド名
  - `.Field().Interface()` フィールド型
  - `.NumMethod()` メソッド数
  - `.Method()` メソッド名

```
func reflectNum(arg interface{}){
    fmt.Println("Type=", reflect.TypeOf(arg))
    fmt.Println("Value=", reflect.Valueof(arg))
}
```

```

}

func main(){
    var num float64 = 3.14
    reflectNum(num)
}

// 测试
func DoFiledAndMethod(input interface{}) {
    // 测试类型
    inputType := reflect.TypeOf(input)
    fmt.Println("inputType is :", inputType.Name())
    // 测试值
    inputValue := reflect.ValueOf(input)
    fmt.Println("inputValue is:", inputValue)

    // 测试字段
    for i := 0; i < inputType.NumField(); i++ {
        field := inputType.Field(i) // 测试字段名
        value := inputValue.Field(i).Interface() // 测试值

        fmt.Printf("%s\tv=%v\n", field.Name, field.Type, value)
    }

    // 测试方法
    for i := 0; i < inputType.NumMethod(); i++ {
        m := inputType.Method(i)
        fmt.Printf("%s\tv\n", m.Name, m.Type)
    }
}

```

测试tag

- `t.Field().Tag.Get("key")` 测试tag
- `t.Elem()` 测试tag的value

```

type resume struct{
    Name string `info:name` `doc:姓名`
    Sex string `info:sex`
}

func findTag(str interface{}){
    t := reflect.TypeOf(str).Elem()

    for i:=0 ;i < t.NumField();i++){
        taginfo = t.Field(i).Tag.Get("info")
        tagdoc = t.Field(i).Tag.Get("doc")
    }
}

```

encoding/json

encoding/json

- encoding/json
- - key json
  - value json key
- json.Marshal() json
- - json
  - json
- json.Unmarshal() json
- - json
  - json

```
import "encoding/json"

type Movie struct{
    Title string `json:"title"`
    Year int `json:"year"`
}

func main(){
    movie := Movie{"",2000}
    jsonStr,err = json.Marshal(movie)

    movie := Movie{}
    err = json.Unmarshal(jsonStr,&movie)

}
```

## goroutine

- A CPU
- CPU
  - CPU

image-20251015154123809


goroutine

- goroutine
- goroutine

- `goroutine` `**** channel` `*****`


\*\*\*\*\*

- `N:M`
- `N*****M*****`
  - `N*****M*****`
  - `*****M*****N*****`

 image-20251015122659139


## Go GMP


- `G` `goroutine`
- `M***** thread ****`
- `P*****`

 image-20251015160120180


- \*\*\*\*\* `GOMAXPROCS` \*\*\*\*\*
- `P` \*\*\*\*\* `LRQ`
- `P` \*\*\*\*\* `GRQ`
  - `P*****`


- \*\*\*\*\* `work stealing` \*\*\*\*\* `hand off`
- `work stealing` \*\*\*\*\* `thread` \*\*\*\*\* `Processor` \*\*\*\*\*
- `hand off` \*\*\*\*\* `thread` \*\*\*\*\* `thread` \*\*\*\*\* `thread` `Processor` \*\*\*\*\* `thread`

 image-20251015161152465

 image-20251015161111010

- \*\*\*\*\* `GOMAXPROCS` `P*****CPU/2`
- \*\*\*\*\* `thread` \*\*\*\*\* `goroutine` \*\*\*\*\* `thread` \*\*\*\*\* `thread` \*\*\*\*\* `thread` \*\*\*\*\* `goroutine`

 image-20251015165748812

 image-20251015165732289

- \*\*\*\*\* `G*****`
  - `thread` \*\*\*\*\* `thread` \*\*\*\*\* `thread` \*\*\*\*\* `thread` \*\*\*\*\*

- **LRQ** **Go** **LRQ** **Work Stealing** **LRQ** **GRQ**

## goroutine

go

- **main** **goroutine** **goroutine**
  - **main** **goroutine**

```
func newTask(){
    i := 0
    for{
        i++
        fmt.Printf("Hello")
    }
}

func main(){
    go newTask()
}
```

- **go**
  - **runtime.Goexit()** **goroutine**
  - **channel**

```
func main(){
    // 1.
    go func() {
        defer fmt.Println("A.defer")

        func() {
            defer fmt.Println("B.defer")
            runtime.Goexit() // goroutine
            fmt.Println("B") // 
        }()

        fmt.Println("A") // 
    }()

    // 2.
    go func(a int, b int) bool {
        fmt.Println("a =", a, ", b = ", b)
        return true
    }(10, 20)
```

```

    for{
        // ...
    }
}

```

## Channel

通道

- `c:=make(chan int)` 创建channel 类型 `int`
- `channel <- value` 向value 写入channel
- `<- channel` 从channel 读取
- `x,ok := <-channel` 从channel 读取 x 是否 ok 是否成功

```

func main(){
    c := make(chan,int)

    go func(){
        c <- 666
    }()

    num := <- c
}


```

- `num:= <- c` 从 `c` 读取 `666` 成功
- `num:= <- c` 从 `c` 读取 `thread` 成功 `666`
- `c <- 666` 向 `666` 写入channel 成功 `thread` 成功 `num:= <- c`

## 通道channel 和 channel


通道

- 通道 `goroutine` 和 `goroutine` 通道

 image-20251015173754231

通道

- 通道 `goroutine` 和 `goroutine` 通道
- 通道 `goroutine` 和 `goroutine` 通道

 image-20251015173810451

## 通道channel

- `chan` `make(chan, int, 3)` 3個の要素を保持できるチャンネル
  - `len(c)` チャンネルの現在の長さ
  - `cap(c)` チャンネルの容量

```
func main(){
    c := make(chan int, 3)
}
```

### channelの操作

- `close(chan)` チャンネルを閉じる
- `x, ok := <-chan` `chan` から値を取り出す。x が値、ok が true かどうか
- `chan` は一度に1つの値しか受け取れない
- `chan` は一度に1つの値しか送れない

### channelとrange

- `range` はチャンネルに対して使われる

```
c := make(chan int, 3)

for data := range c{
    fmt.Println(data)
}
```

### channelとselect

- `select` は複数のチャンネルの中から1つを選択する
- `select` は複数のチャンネルの中から1つを選択する。case が複数ある場合、どれか1つが実行される

```
select{
case <- chan1:
    // channel1から値を取り出す
case chan2 <- 1:
    // channel2から値を取り出す
default:
    // default case
```

## Go Modules

Go Modulesの基本的な使い方

### GoPath

- `go get` コマンド
- `go install` コマンド
- `go mod` コマンド

### go mod

- `go mod init` 创建 `go.mod` 文件
  - 初始化模块
- `go mod download` 下载 `go.mod` 依赖的模块
- `go mod tidy` 清理模块
- `go mod graph` 显示模块依赖图
- `go mod edit` 编辑 `go.mod` 文件
  - `go` 添加/删除模块
  - `-require` 添加/删除模块
  - `-droprequire` 删除模块
  - `-replace` 替换模块
  - `-exclude` 排除模块
- `go mod vendor` 将模块下载到 `vendor` 目录
- `go mod verify` 验证模块

## go mod 配置

- `GOM11MODULE` 控制 Go modules 是否启用
  - `auto` 根据 `go.mod` 文件决定是否启用 Go modules
  - `on` 始终启用 Go modules
  - `off` 始终禁用 Go modules

配置方法

```
go env -w GOM11MODULE=on
```

- `GOPROXY` 指定 Go modules 的代理地址
  - 默认地址 `https://proxy.golang.org,direct`

配置

```
go env -w GOPROXY=https://goproxy.cn.direct
```

- `GOSUMDB` 指定 Go modules 的摘要数据库
  - 默认地址 `sum.golang.org`
  - 指定 `GOPROXY` 地址
- `GONOPROXY/GONOSUMDB/GOPRIVATE` 控制 Go modules 的代理地址
  - 指定 `GOPRIVATE` 地址 `GONOSUMDB` 和 `GONOPROXY` 地址
  - 指定 `GONOSUMDB` 地址

## go.mod 文件

```
module github.com/yourname/project // 模块名称

go 1.21 // 指定 Go 版本

require (
    github.com/gin-gonic/gin v1.9.1 // 依赖库
    golang.org/x/sync v0.3.0
)
```



```
replace (                                // 替换源
    golang.org/x/sync => ./local/sync // 本地源
)

exclude (                                // 排除源
    github.com/old/lib v1.2.3
)

retract (                                // 回退源
    v1.0.0 // 回退版本
)
```

## go.sum

- 替换源
- 排除源