

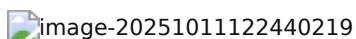
Golang

Google Go 语言是一种由 Google 开发的开源编程语言，具有简洁、高效的特点。

docker k8s go

- 语言特性
 - 并行处理
 - 垃圾回收
 - 简洁易学
 - 高效稳定
- 编程模型
 - 事件驱动
 - 异步非阻塞
 - 多线程
 - 多协程
 - CPU 指令
- 运行时
 - runtime 机制
 - 并行垃圾回收
 - 垃圾回收
 - 并行垃圾回收
 - 并行垃圾回收
- 性能
 - 25% C
 - 80% C
 - 90% C
 - 95% C

性能优势



Golang

命令行

命令行工具

- go run 命令行 go 语句
 - 打印 main() 函数值

- go build 建立可执行文件
 - -o 定义输出文件名
 - go version 显示go版本
 - go get path 下载指定的Github仓库
 - go env 显示环境变量

1

Hello World

```
package main // 亂子

import "fmt" // 亂子

// 亂子
import(
    "fmt"
    "time"
)

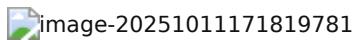
// 亂子
func main(){
    fmt.Println("Hello World")
}
```

1

- package main プログラム名
 ◦ フォルダ package main フォルダ名
 ◦ フォルダ名/main.go .a フォルダ
 ◦ main フォルダ名
 - フォルダ名/main.go
 - フォルダ名/main package フォルダ名

1

- `import` `fmt` `println`
 - ◦ `import` `io`
 - ◦ `import` `io.TextReader`
 - `go` `fmt.Println`
 - ◦ `fmt`
 - `io.TextReader` `init()` `func`



- `__init__` _ `__init__(self, name)` `init()` `__init__`
 - `__init__(self, name)`
 - `name` . `__init__(self, name)`

```
package main

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

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

    // ...
    Lib3Test()
}
```

•

- `mylib2.Lib2Test()`

•

```
// ...
bool

// ...
string

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

// uint8...
byte

// int32...
rune

// ...
float32 float64

// ...
complex64 complex128

// ...
string
```

•

- `%v` `string`

- `%T` BOOLEAN
 - `%t` BOOLEAN true | false
 - `%d` INTEGER
 - `%x` HEXADECIMAL
 - `%b` BINARY
 - `%c` CHAR Unicode \u0000
 - `%s` STRING
 - `%q` QUOTED STRING JSON
 - `%p` PTR
 - `%f` FLOAT
 - `%e` EXPONENT
 - `%g` EXPONENTIAL %e | %f
 - `%#v` Go PRINTFLIKE
 - `%#x` HEXADECIMAL 0x INTEGER

1

□ □ □ □ □ □ □ □

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

var

- `var` `var` `var`
 - `var` `var` `var`
 - `var` `var` `var` **0**

- `fmt` `Printf` `Format`
 - `fmt.Printf("type of a = %T",a)` `Format` `%T`
 - `Scan`
 - `Scanf`

1

- $\square \square := \square \square$
 - $\square \square \square \square \square \square \square$
 - $\square \square \square$
 - $\square \square \square := \square \square \square \square \square \square \square \square \square$

1

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

const

- $\text{Bijection} \rightarrow \text{Bijection}$
 - $\text{Bijection} \rightarrow \text{Bijection}$
 - $\text{Bijection} \rightarrow \text{Bijection}$
 - $\text{Bijection} \rightarrow \text{Bijection} \rightarrow \text{Bijection}$

11

```
import "fmt"

// 111
func fool1(a string,b int) int{
    fmt.Println("a = ",a)
    fmt.Println("b = ",b)

    c := 100
    return c
}

// 111111111
func fool1(a string,b int) (int,int){
    return 666,777
}
```

1

- go ဂုဏ်ဆိုင်များ
 - ဂုဏ်ဆိုင်ရေးလုပ်နည်းလမ်း
- ဂုဏ်ဆိုင်ရေးလုပ်နည်းလမ်း
- ဂုဏ်ဆိုင်ရေး
- ဂုဏ်ဆိုင်ရေးလုပ်နည်းလမ်း
- ဂုဏ် _ ဂုဏ်ဆိုင်

ဂုဏ်ဆိုင်

- ဂုဏ်ဆိုင်ရေးလုပ်နည်း
 - ဂုဏ်ဆိုင်ရေး
- ဂုဏ်returnဂုဏ်ဆိုင်ရေးလုပ်နည်း

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

၁၁

ဂုဏ်ဆိုင်ရေးလုပ်နည်းလမ်း

- ဂုဏ်ဆိုင်ရေးလုပ်နည်း * ဂုဏ်ဆိုင်ရေးလုပ်နည်း
- & & ဂုဏ်ဆိုင်ရေး

```
package main

import "fmt"

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

func addptr(n *int){
    // အောက်ဖြစ်ပါသည်
    *n += 2
}

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

ဂုဏ်ဆိုင်

- if ဂုဏ်ဆိုင်ရေးလုပ်နည်း if ဂုဏ်ဆိုင်
- if-else ဂုဏ်ဆိုင်ရေး

- `if` 语句 ; `else`

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

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

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

switch 语句

- `switch` 语句
- `switch` \square `case` 语句
- `switch` 语句 `break`
- `else` `fallthrough` 语句 `case`

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

循环语句

- `go` 循环语句 `for`
- `continue` \square `break` 语句

```
for{
    //语句
}

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

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

defer

- defer
- defer
- defer
- **defer return**

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

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

Slice

slice

slice

- var slice slice {int} slice
- slice := []

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

- len(slice) slice

```
var myArray1 [10]int
```

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

- range slice index slice value slice
 ◦ _ slice _ slice

```
// slice
var myArray1 [10]int

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

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

- `slice` と `make` の関係

```
// []
func method(arr [5]int){

}

// []
func method(arr [4]int){

}

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

問題

- `slice` と `make` の関係
 - `nil`
 - `nil` を `make` で作成する
 - `slice` が `make` で作成される
 - `nil` を `make` で作成
 - `nil := nil make`

```
// 例題
slice1 := [int]{1,2,3}

// 例題`slice`の作り方
var slice1 []int
slice1 = make([]int,3)

// 例題`make`の作り方
var slice1 []int = make([]int,3)

// 例題`:=`の作り方
var slice1 := make([]int,3)
```

- `slice` の書き方

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

func main() {
```

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

- nil 100 nil 100

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

- nil
 - make
 - append a = append(a,value)
 - append
 - nil

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

// nil
numbers = append(numbers,1)
```

- nil
 - s[i:]
 - s[:j] i:j
 - nil
 - copy()
 - copy(s1,s2) s2 nil s1

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

// s1 1 2
s1 = s[0:2]
```

Map

Map

- [] key value
 - make
 - :=
 - nil
 - nil
- key value nil

```
// マップ
var myMap1 map[string]string
// マップの作成
myMap1 = make(map[string]string),10)
// マップの初期化
myMap1["one"] = "php"
myMap2["two"] = 'js'
myMap3["three"] = "go"

// マップの取得
var myMap2 := make(map[int]string,10)

// マップの構造
myMap3 := map[string][string]{
    "one":"php",
    "two":"js",
    "three":"go"
}
```

Map操作

- マップの range ループ

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

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

- マップの delete 操作
 - マップのmap構造
 - マップの構造からkeyを削除

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

delete(myMap3,"one")
```

- マップ構造の key の確認

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

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

- `map` は構造体

Struct

構造体

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

構造体宣言

- `var` 構造体宣言

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

構造体

- `copy` 構造体複数
- `指针` 構造体

```
// コピー
func changeStruct(person Person){
    // ...
}

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

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

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

5

- 二〇一〇年十一月三十日
 - 二〇一〇年十二月三十一日

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

1

10 / 10

- 二〇二〇年九月二十一日
 - 二〇二〇年九月二十二日

5


```
type Person struct{
    Name string
    Age int
}
person := Person{name: "Alice", age: 25}
```

1

- **this** **指向** **对象**
 - **this** **指向** **对象**
 - **对象**

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

```
}
```

2

1

- 
 - 

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

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

// 人物
type SuperMan struct{
    Human // SuperMan继承Human
    level int
}

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

11

2

- 空闲 interface 例題
 - 空闲 interface
 - 空闲 interface
 - 空闲 interface
 - 空闲 interface
 - 空闲 interface
 - 空闲 interface

```
type Animal interface {
    Speak()
}

type Cat struct{}

type Dog struct{}
```

```

// 犬の鳴き声
func (d *Dog) Speak() {
    fmt.Println("Woof")
}

// 猫の鳴き声
func (c Cat) Speak() {
    fmt.Println("Meow")
}

func main() {
    var a Animal

    // 犬と猫の関係性
    a = Cat{}           // 犬
    a = &Cat{}          // 猫のGo 関係性
    a = &Dog{}          // 犬
    a = Dog{}           // 犬のDog プロパティ Animalの *Dog プロパティ
}

```

□□

- ・ 犬と猫の関係性
- ・ 犬と猫の関係性を明確にするために Animal という名前

```

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

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

// 猫の構造体
type Cat struct{
    color string
}

// 犬の構造体

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(){
    // ...
    var animal AnimalIF

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

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

```

interface

- `interface`
- `interface` \times `(T)` \rightarrow `x` \in `T` \rightarrow `x`
- `interface` \rightarrow `T` \rightarrow `x` \in `T` \times `interface` \rightarrow `T` \rightarrow `x` \in `T`
- `interface`
- `value` \rightarrow `ok`
 - `value` \rightarrow `ok` \rightarrow `true` \rightarrow `value` \rightarrow `ok`

```

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

```

```

}

type book struct{
    // ...
}

func main(){
    book := Book{}
    // ...
    MyFunc(book)
}

```

二

pair 二

- type
 - static type
 - concrete type
- value
- pair

```

var a string
a = "aceld"

var allType interface{}
// allType 二 value type a 二
allType = a

```

二

- reflect 二
 - ValueOf() 二
 - TypeOf() 二
- Field 二
 - NumField() 二
 - Field() 二
 - Field().Interface() 二
 - NumMethod() 二
 - Method() 二

```

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)
}

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

    // 例
    for i := 0; i < inputType.NumField(); i++ {
        field := inputType.Field(i)           // 例
        value := inputValue.Field(i).Interface() // 例

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

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

```

例

- .Field().Tag.Get("key") 例
- .Elem() 例

```

type resume struct{
    Name string `info:name` `doc:简历`
    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")
    }
}

```

JSON

JSON

- `encoding/json`
- `json`
 - key 在 json
 - value 在 json 中 key
- `json.Marshal()` 将对象转换为 `json` 字符串
 - 将 json 转换为字符串
 - 将字符串转换为 json
- `json.Unmarshal()` 将 `json` 字符串转换为对象
 - 将 json 转换为对象
 - 将对象转换为 json

```
import "encoding/json"

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

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

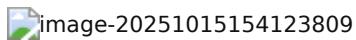
    movie := Movie{}
    err = json.Unmarshal(jsonStr,&movie)

}
```

goroutine

协程

- 协程是线程A和CPU线程B
- CPU线程
- CPU线程是线程A和线程B的统称



协程

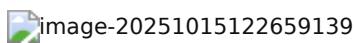
goroutine

- `goroutine` 语句
- `goroutine` 语句

- goroutine `** channel` `**`

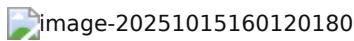
•

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



Go GMP

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

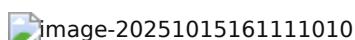


•

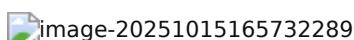
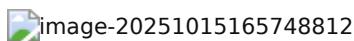
- GOMAXPROCS
- P LRQ
- P GRQ
 - P

•

- work stealing hand off
 - work stealing thread Processor
 - hand off thread thread thread thread Processor



- GOMAXPROCS CPU/2
- thread goroutine thread thread thread thread thread thread goroutine



- G
- thread thread thread thread

- GRQ Work Stealing LRQ Work Stealing
- LRQ は GRQ

goroutine

main.go は

- main メソッド goroutine は main メソッド goroutine
- main メソッド goroutine は

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

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

- main.go は
 - 実行するgoroutine
 - 実行するgoroutine () も実行する
 - 実行するgoroutine
 - も go 実行するgoroutine runtime.Goexit() も実行する goroutine
- 実行するgoroutine
 - 実行するgoroutine
 - 実行する channel も

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

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

    fmt.Println("A")      // 実行
}()

// 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) //channel声明语句 int`
- `channel <- value //value向channel写入值`
- `<- channel //读取值`
- `x,ok := <-channel //channel接收语句 x ok 表示接收成功与否`

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

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

    num := <- c

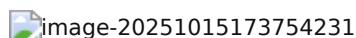
}
```

- `num:= <- c // c <- 666 表示从channel读取值`
 - `// num:= <- c 表示从channel向thread写入666值`
 - `// c <- 666 表示从666向channel写入channel向thread写入 num:= <- c`

channel向channel

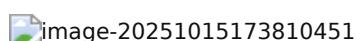
通道

- `两个 goroutine 通过channel goroutine 通信`



通道

- `两个 goroutine 通过channel向channel goroutine 通信`
- `通过channel向channel通信`



channel向channel

- `make(chan, int, 3)` 3つのchannelを返す
 - `len(c)` channelの長さ
 - `cap(c)` channelの容量

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

channelの操作

- `close(chan)` channelを閉じる
- `x,ok := <-channel` リーチャンネルを受信する。xはokがtrueなら値、falseならnil
- `channel1,channel2 := <->channel` channel1とchannel2を対向チャネルとして作成
- `channel1,channel2 := <-><->channel` channel1とchannel2を対向チャネルとして作成

channelのrange

- `range` ループ用キーワード

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

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

channelのselect

- `select` ブラケットで囲む channel
- `select { case ... case ... case ... case ... }`

```
select{
case <- chan1:
    // channel1にデータが来た時の処理
case chan2 <- 1:
    // channel2にデータが来た時の処理
default:
    // デフォルト処理
}
```

Go Modules

Goモジュールの仕組み

GoPath

- `$GOROOT`
- `$GOPATH`
- `$GOBIN`

go mod

- go mod init 项目名 go.mod 项目
- 生成模块文件
- go mod download 项目名 go.mod 下载所有依赖
- go mod tidy 项目名 整理模块
- go mod graph 项目名 生成模块图
- go mod edit 项目名 go.mod 编辑
- go 项目名go命令
- -require 项目名
- -droprequire 项目名
- -replace 项目名
- -exclude 项目名
- go mod vendor 项目名生成vendor vendor 项目
- go mod verify 项目名验证模块

go mod命令

- G0111MODULE 项目名 Go modules 项目
- auto 项目名生成 go.mod 项目名 Go modules
- on 项目 Go modules
- off 项目 Go modules

环境变量设置

```
go env -w G0111MODULE=on
```

- GOPROXY 项目名Go代理设置为直接连接源
 - 项目名 https://proxy.golang.org,direct

项目

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

- GOSUMDB 项目名Go模块缓存设置
 - 项目名 sum.golang.org
 - 项目名 GOPROXY 项目名
- GONOPROXY/GONOSUMDB/GOPRIVATE 项目名Go模块缓存设置
 - 项目名 GOPRIVATE 项目名设置 GONOSUMDB 为 GONOPROXY 项目名
 - 项目名GONOPROXY 项目名设置

go.mod文件

```
module github.com/yourname/project // 项目名

go 1.21 // 项目名 Go 版本号

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

```
replace (                                // 旣存するモジュール
    golang.org/x/sync => ./local/sync  // リロケーション
)

exclude (                                // 排除するモジュール
    github.com/old/lib v1.2.3
)

retract (                                // 削除するモジュール
    v1.0.0 // バージョン
)
```

go.sum

- ・ 既存するモジュールを削除する
- ・ 新規モジュールを登録する