go 语言

(main.go)

```go
package main // main package

func main() { // main function入口
    println("it's over 9000")
}
```

$ go **run** main.go ⇒ 必须要有 main package, main函数
↳ 编译成一个临时文件, 删除 ./main.

$ go **run --work** main.go (看到临时文件)

$ go **build** main.go ⇒ main (binary)

导入包

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    if len(os.Args) != 2 {
        os.Exit(1)
    }
    fmt.Println("it's over", os.Arg[1])
}
```

→ 导入, 但不用, 会报错 (因为编译变慢了)

fmt → 包名

$ go run main.go 9000

变量和声明

```go
var power int
power = 9000
```

(默认为 0 (int), false (bool), "" (string))

= var power int = 9000

= power := 9000 (自动推导类型), (声明变量并赋值)

name, power := "Goku", 9000. (同一作用域, 不能重名)

OK

```go
power := 9000
name, power := "Goku", 9001
```

新变量

var NAME TYPE

NAME := VALUE

(NAME = VALUE)

函数

```
func log (message string) { // 无返回值
}
func add (a int, b int) int { // 返回1个值
}
func power (name string) (int, bool) { // 返回多个值
}
```

(string: 类型)

```
func add (a, b int) int {
}
```

a, b 类型相同"int" (简写)

For example:

```
value, exists := power("goku")
if exists == false {
    ...
}
```

空白标识符 ← 没有真正的值.

```
_, exists := power("goku")
```

结构体

go 不是面向对象的, 没有对象和继承的概念, 也没有多态和重载

```
type Saiyan struct {
    Name string
    Power int
}
```

⇐ 类似 ⇒

```
goku := Saiyan {
    Name: "Goku",
    Power: 9000,
}
```

- `goku := Saiyan{}` // 字段有默认值.
- `goku := Saiyan{Name: "Goku"}`
  `goku.Power = 9000`
- `goku := Saiyan{"Goku", 9000}`

函数参数传递是传值, go 支持指针

```
func Super(s *Saiyan) {
    s.Power += 10000
}
```

⇐

```
goku := &Saiyan{}
goku.Name = "xx"
goku.Power = 10000.
```

⇐ 仍然用 "." 不是 →

结构体上的函数

```
type Saiyan struct{
    Name string
    Power int
}

func (s *Saiyan) Super() {
    s.Power += 10000
}
```

```
goku := &Saiyan{"Goku", 9001}
goku.Super()
fmt.Println(goku.Power)
```

*Saiyan 类型指针

表示第一个参数.

*Saiyan 是 Super() 方法的接收者.

构造函数

```
func NewSaiyan(name string, power int) *Saiyan {
    return &Saiyan{
        Name: name,
        Power: power,
    }
}
```

返回值 (1个指针)

⇒ 当然可直接返回一个结构体 return Saiyan{
...
}

new

内置的 new 函数 new(X) = &X{} 分配一个类型需要的内存.

```
goku := new(Saiyan)
goku := &Saiyan{}
```

```
type Saiyan struct{
    Name string
    Power int
    Father *Saiyan
}
```

```
gohan := &Saiyan{
    Name: "Gohan",
    Power: 1000,
    Father: &Saiyan{
        Name: "Goku",
        Power: 9001,
        Father: nil,
    },
}.
```

组合

```go
type Person struct{
    Name string
}
func (p *Person) Introduce() {
    fmt.Printf(.....)
}
type Saiyan struct {
    *Person   未命名的字段.
    Power int
}
```

```go
goku := &Saiyan{
    Person: &Person{"Goku"},
    Power: 9001,
}
goku.Introduce()            // 看上去像继承
```

可重载 Introduce
(实则覆盖)

```go
func (s *Saiyan) Introduce() {
    .....
}
```

数组

固定大小，类似 C.

```
var scores [10]int          scores := [4]int{9001, 9333, 212, 33}
scores[0] = 339
```

len 获取数组长度　**range** 遍历数组

```
for index, value := range scores {

}
```

效率高但不灵活

切片

轻量级的结构体封装.

```
scores := []int{1, 2, 3, 4}
scores := make([]int, 10)
```

切片长度和数组长度都是10.

```
scores := make([]int, 0, 10).
```

0 → 切片长度；10 → 数组容量（底层开辟分配的内存）

类似于 std::vector<int> v; v.reserve(10);

```
scores = append(scores, 5)
```

append '5' 作为第六个元素.
(push back)
如果 append 10个元素后，会自动 grow size，因此需要重新赋值给 scores.

```
names := []string{"leto", "jessica", "paul"}
checks := make([]int, 10)
var name []string
scores := make([]int, 0, 20)
```

⟹ 与 append 一起使用.

```
scores := []int{1, 2, 3, 4, 5}
slice := scores[2:4]      ← 不是 copy.
slice[0] = 999
fmt.Println(scores)
    [1, 2, 999, 4, 5].
```

```
func extractPowers(saiyan []*Saiyan) []int {
    powers := make([]int, len(saiyan))
    for index, value := range saiyans {
        powers[index] = value.Power
    }
    return powers
}
```

```
scores := make([]int, 0, 10)
scores[5] = 100   // 不work

scores = scores[0:6]   // work.
scores[5] = 100
```

映射 map

```go
lookup := make(map[string]int)   // string → key, int → value
lookup["goku"] = 9001
power, exists := lookup["vegeta"]   // power = 0, exists = false
total := len(lookup)
delete(lookup, "goku")
```

动态增长

```go
lookup := make(map[string]int, 100)   // 100 → 初始大小
```

```go
type Saiyan struct {
    Name string
    Friends map[string]*Saiyan   // string → key, *Saiyan → value
}
```

⇒

```go
goku := &Saiyan{
    Name: "Goku",
    Friends: make(map[string]*Saiyan),
}
goku.Friends["krillin"] = ...
```

```go
lookup := map[string]int{
    "goku": 9001,
    "gohan": 2044,
}

for key, value := range lookup {
    ...
}
```

map 都不是有序的（有序就是hashtable）

```go
a := make([]Saiyan, 10)
b := make([]*Saiyan, 10)
```

传递时是个切片，其是个引用，(指针一样)

package

$GOPATH/src/shopping ⇒ package shopping
$GOPATH/src/shopping/db ⇒ package db // 定义时只取 package name 与文件名相同

```
import(
  "shopping/db"   // 导入时需要 full name
)
```

```
package shopping
import(
  "shopping/db" // 相对 $GOPATH/src
)
```

......

可见性

命名类型/函数以大写字母开头. ⇒ 可见
小写字母开头 ⇒ 不可见.

数据结构体 类似

接口

```
type Logger interface {
    Log(message string)
}
```

```
type Server struct {
    Logger Logger
}
```

错误处理

go 没有异常处理. 一般通过返回值处理错误

也可以自己创建个错误类型, 继承实现内置类型 error 接口

```
type error interface {
    Error() string
}
```

初始化的if

```
if x:=10; count>x {
   ...
}
```

```
if error:=process(); error!=nil {
    return error
}
```

只对 if / else if / else 可见.

函数类型

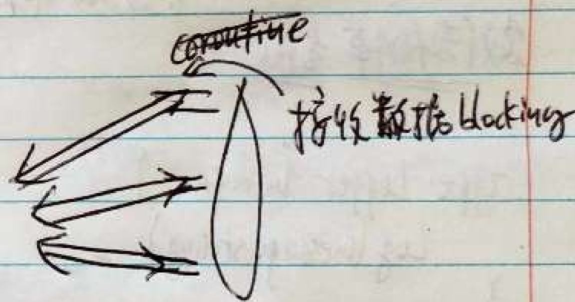
- first-class type

```
type Add func(a int, b int) int
```

```
type Add func(a int, b int) int

fmt.Println(process(func(a int, b int) int {
        return a+b
    }))
func process(adder Add) int {
    return adder(1, 2)
}
```

go并发 协程/通道 · channel



channel 用于数据的传递.

- 传送数据阻塞.
- 接收数据阻塞.

数据类型 for channel

```
c := make(chan int)
for i := 0; i<5; i++ {
    worker := &Worker{id: i}
    go worker.process(c)
}
for {
    c <- rand.Int()        (发送 blocking)
    time.sleep(time.Millisecond *50)
}
```

```
type Worker struct {
    id int
}
func (w *Worker) process(c chan int) {
    for {
        data := <-c        (只有1个协程)
                           //接收阻塞.
        fmt.Printf("worker %d got %d\n",
                        w.id, data)
    }
}
```

```
c := make(chan int, 100)     (不阻塞 发送方)
```

与 channel 长度. 扮演队列(最存100)