

蓝山工作室——Golang 第一节课

关于 Golang 语言

Go 语言由 Robert Griesemer, Rob Pike, Ken Thompson 于 2007 年末开发，并于 2009 年开源。它旨在提供一种简单、可靠且高效的编程语言，特别适合构建高性能网络服务和分布式系统。

认识一下 Go 程序

一个基础的 Go 程序通常包含以下几个部分：

codeGo

```
package main // 声明主包，表示这是一个可执行程序

import "fmt" // 导入 fmt 包，用于格式化输入输出

func main() { // 程序入口函数
    fmt.Println("Hello, World!") // 调用 fmt 包的 Println 函数打印字符串
}
```

- `package main`: 定义了程序的入口点。
- `import "fmt"`: 导入标准库中的 `fmt` 包，提供打印等功能。
- `func main()`: 程序的执行从这里开始。

Go 语法基础

数据类型

Go 语言支持以下基本数据类型：

1. 布尔型 (`bool`): `true` 或 `false`。
2. 数字类型:
 - 整型: `int8`, `int16`, `int32`, `int64` (有符号); `uint8`, `uint16`, `uint32`, `uint64` (无符号)。
 - `int`: 根据操作系统架构，可以是 32 或 64 位。
 - `byte`: `uint8` 的别名。
 - `rune`: `int32` 的别名，用于表示 Unicode 码点。
 - 浮点型: `float32`, `float64`。
 - 复数型: `complex64`, `complex128`。

3. 字符串类型 (`string`): 底层是不可变的 UTF-8 编码字节数组。
4. 派生类型: 包括指针、数组、结构体、`Channel`、函数、切片、接口和 `Map`。

变量声明

Go 语言变量必须先声明再使用, 且同一作用域内不可重复声明, 声明后也必须使用。

- 完整声明: `var 变量名 类型 = 表达式`

codeGo

```
var a = "first"      // 类型推导
var d int8 = 1       // 指定类型
var e float64        // 声明
e = 1.0              // 赋值
```

- 短变量声明: `变量名 := 表达式` (只能用于函数内部)

codeGo

```
f := 114514 // 自动推导类型并赋值
```

常量

常量是程序运行期间不可改变的值, 使用 `const` 关键字声明, 定义时必须赋值。

codeGo

```
const s string = "constant"
const h = 500000000 // 类型自动推导
```

循环

Go 语言只有 `for` 一个循环关键字。

1. 三段式: `for 初始化; 条件; 循环表达式 { ... }`

codeGo

```
for i := 0; i < 3; i++ {
    fmt.Println(i)
}
```

- 各部分可省略, 如 `for ; i < 5; { ... }`

2. 一段式 (只带条件): `for 条件 { ... }`

codeGo

```
i := 0
for i < 5 {
    fmt.Println(i)
    i++
}
```

3. 无限循环: `for { ... }`

codeGo

```
for {
    fmt.Println("loop")
    break // 需要手动跳出, 否则无限循环
}
```

- `break`: 跳出当前循环。
- `continue`: 跳过当前循环的剩余部分, 进入下一次循环。

条件判断 `if`

codeGo

```
if 条件表达式 {
    // 条件为 true 时执行
} else {
    // 条件为 false 时执行
}
```

- 支持 `else if` 多分支。
- `if` 语句可以带一个短声明, 声明的变量作用域仅限于 `if/else if/else` 块内。

codeGo

```
if num := 9; num < 0 {
    fmt.Println(num, "is negative")
} else if num < 10 {
    fmt.Println(num, "has 1 digit")
} else {
    fmt.Println(num, "has multiple digits")
}
```

多分支选择 `switch`

用于处理多个条件分支，比 `if-else if` 更简洁。

- `case` 匹配成功后会**自动跳出**，无需 `break` (与 C/C++ 不同)。
- 使用 `fallthrough` 可以强制执行下一个 `case`。
- 可以同时匹配多个值: `case 4, 5: ...`
- `switch` 后可不跟表达式，直接在 `case` 中写条件。

codeGo

```
a := 2
switch a {
case 1:
    fmt.Println("one")
case 2:
    fmt.Println("two") // 匹配到此并跳出
    // fallthrough // 如果加上这句，会继续执行 case 3
case 3:
    fmt.Println("three")
default:
    fmt.Println("other")
}

t := time.Now()
switch { // 没有 switch 表达式，case 直接是条件
case t.Hour() < 12:
    fmt.Println("It's before noon")
default:
    fmt.Println("It's after noon")
}
```

函数 `func`

函数是组织代码的基本单位，用于实现特定功能。

1. 声明与定义: `func 函数名(参数列表) 返回值列表 { ... }`

codeGo

```
func add(x int, y int) int {
    return x + y
}
```

- ```
}
 ○ 参数类型相同时可简写: func add(x, y int) int { ... }
 ○ 可返回多个值: func swap(x, y int) (int, int) { return y, x }
```
2. 调用: `result := add(3, 5)`
  3. 函数是一等公民: 函数可以像普通变量一样被赋值、作为参数传递、作为返回值返回。这使得高阶函数和闭包成为可能。

## codeGo

```
type MathOperation func(int, int) int // 定义函数类型

func add(a, b int) int { return a + b }

func main() {
 var op MathOperation = add // 将函数赋值给变量
 fmt.Println(op(1, 2)) // 调用函数变量
}
```

### `fmt`包

`fmt`包提供了格式化输入输出功能。

- `fmt.Println()`: 打印值后换行。
- `fmt.Printf()`: 格式化打印。
  - `%v`: 默认格式打印。
  - `%+v`: 打印结构体时显示字段名。
  - `%#v`: 打印值的 Go 语法表示。
  - `%.2f`: 浮点数格式，保留两位小数。

## codeGo

```
type point struct { x, y int }
p := point{1, 2}
fmt.Printf("p=%v\n", p) // p={1 2}
fmt.Printf("p=%+v\n", p) // p={x:1 y:2}
fmt.Printf("p=%#v\n", p) // p=main.point{x:1, y:2}
f := 3.14159
fmt.Printf("%.2f\n", f) // 3.14
```

---

## 第 02 节课

# 数组 & 切片

数组和切片是 Go 语言中用于存储同类型元素序列的两种数据结构。

## 数组-array

- **固定长度**: 数组的长度在定义后不可改变，是类型的一部分。  
`[3]int` 和 `[4]int` 是不同类型。
- **值类型**: 数组是值类型，传递时会创建一份完整的副本。
- **初始化**:

### codeGo

```
var a [3]int // 声明一个长度为 3 的 int 数组，元素默认为零值
[0 0 0]

b := [3]int{1, 2} // [1 2 0]，未初始化元素为零值
c := [...]int{1, 2, 3, 4} // 通过初始化值确定长度为 4
d := [5]int{2: 100, 4: 200} // [0 0 100 0 200]，通过索引初始化
```

- **多维数组**: 支持多维数组。

### codeGo

```
e := [2][2]int{{1, 2}, {3, 4}} // [[1 2] [3 4]]
```

- **访问**: 通过索引 `arr[index]` 访问，索引从 0 到 `len-1`。

### codeGo

```
arr := [3]string{"a", "b", "c"}
fmt.Println(arr[0]) // 输出: a
```

- **遍历**: `for i, v := range arr { ... }`

### codeGo

```
for i, v := range arr {
 fmt.Printf("arr[%d] = %s\n", i, v)
}
```

## 切片-slice

- **可变长度**: 切片是动态可变的，底层引用一个数组。

- **引用类型**: 切片是引用类型，传递时只复制切片头信息（指针、长度、容量），指向同一个底层数组。
- **切片三要素**: 一个切片由**指针**（指向底层数组的起始位置）、**长度**（`len`，当前可见元素数量）和**容量**（`cap`，从指针位置到底层数组末尾的元素数量）组成。
- **初始化**:

## codeGo

```
var s1 []int // 声明一个 nil 切片，长度和容量都为 0
s2 := []int{} // 声明一个空切片，长度和容量都为 0
s3 := []int{1, 2, 3} // 初始化并赋值
s4 := make([]int, 5) // 创建一个长度为 5，容量为 5 的切片，元素为零值
s5 := make([]int, 0, 10) // 创建一个长度为 0，容量为 10 的切片
```

- **操作**:
  - `len(s)`: 获取切片长度。
  - `cap(s)`: 获取切片容量。
  - `s[n]`: 访问索引为 `n` 的元素。
  - `s[low:high]`: 切片操作，从 `low` 到 `high-1`。新切片长度 `high-low`，容量 `原 cap-low`。
  - `s[low:high:max]`: 切片操作，指定底层数组的最大容量。新切片长度 `high-low`，容量 `max-low`。

## codeGo

```
nums := []int{10, 20, 30, 40, 50}
subSlice := nums[1:3] // subSlice: [20 30], len: 2, cap: 4 (从 20 开始到末尾)
fullSlice := nums[0:len(nums):len(nums)] // 完整切片，限制容量等于长度
```

- **append()**: 向切片末尾添加元素，返回新切片。当容量不足时，会重新分配一个更大的底层数组（通常是 2 倍），并复制旧数据。

## codeGo

```
s := []int{1, 2}
s = append(s, 3, 4) // s: [1 2 3 4]
```

## 切片与数组的关系

切片是底层数组的一个“视图”。多个切片可以引用同一个底层数组的不同部分。对一个切片的修改可能会影响到引用同一底层数组的其他切片。

## codeGo

```
s3 := []int{1, 2, 3, 4, 5, 6, 7, 8} // len=8, cap=8
s4 := s3[3:6] // s4: [4 5 6], len=3, cap=5 (从索引 3 到 8)
s6 := append(s4, 9) // s6: [4 5 6 9], len=4, cap=5。如果 s4 的 cap 不足，append 会创建新底层数组。
// 否则，s6 会修改 s3 中 s4 底层数组对应位置的值。
```

## 字典-map

Map 是键值对的无序集合，通过键快速查找对应的值。键必须是可比较的类型（不能是函数、切片、map 本身）。

- 初始化:

### codeGo

```
// 使用 make 函数
ages := make(map[string]int) // 创建一个 string 到 int 的 map

// 使用字面量
grades := map[string]string{
 "Alice": "A",
 "Bob": "B",
}
```

- 增/改元素: `map[key] = value`

### codeGo

```
ages["Alice"] = 25
ages["Bob"] = 30
```

- 查询元素: `value, exists := map[key]`

### codeGo

```
aliceAge, exists := ages["Alice"]
if exists {
```

```

 fmt.Printf("Alice's age: %d\n", aliceAge)
} else {
 fmt.Println("Alice not found.")
}

```

- 遍历: `for key, value := range map { ... }`

## codeGo

```

for name, age := range ages {
 fmt.Printf("%s is %d years old.\n", name, age)
}

```

- 删除元素: `delete(map, key)`

## codeGo

```

delete(ages, "Bob")

```

# 函数进阶

## 函数是一等公民

Go 语言中，函数可以被赋值给变量、作为参数传递给其他函数、作为返回值返回。

- 函数类型: 可以像其他类型一样定义函数类型。

## codeGo

```

type Printer func(contents string) (n int, err error) // 定义一个函数类型
func printToStd(contents string) (bytesNum int, err error) {
 return fmt.Println(contents)
}
func main() {
 var p Printer = printToStd // 将函数赋值给函数类型变量
 p("something")
}

```

- 函数签名: 指函数的参数列表和结果列表的类型和顺序。只要签名一致，函数就可以互相赋值。

## 高阶函数

满足以下任一条件即可称为高阶函数：

1. 接受其他函数作为参数。
2. 把其他函数作为结果返回。

## codeGo

```
type Operate func(x, y int) int // 定义一个操作函数类型

// 接受函数作为参数的高阶函数
func calculate(x, y int, op Operate) (int, error) {
 if op == nil { return 0, errors.New("invalid operation") }
 return op(x, y), nil
}

// 返回函数作为结果的高阶函数（通常伴随闭包）
type CalculateFunc func(x, y int) (int, error)
func genCalculator(op Operate) CalculateFunc {
 return func(x int, y int) (int, error) { // 这是一个闭包
 if op == nil { return 0, errors.New("invalid operation") }
 return op(x, y), nil
 }
}
```

## 闭包

闭包是一个函数值，它引用了其函数体外部的变量。这个外部变量被称为“自由变量”。闭包会“记住”它被创建时的环境，即使该环境已不再存在。

- 作用：延迟逻辑实现、动态生成函数、状态共享（如迭代器）、回调函数、函数工厂等。

## codeGo

```
// 例子：函数工厂
func CalculationFactory(operation string) func(int, int) int {
 switch operation {
 case "add":
 return func(a, b int) int { return a + b }
 case "subtract":
 return func(a, b int) int { return a - b }
 default:
```

```

 return nil
 }
}

func main() {
 addFunc := CalculationFactory("add")
 fmt.Println(addFunc(5, 3)) // 输出: 8
}

```

## 引用类型 & 值类型

- 值类型: `int`, `float`, `bool`, `string`, `array`, `struct`。函数传参时会复制一份完整的副本。修改副本不会影响原值。
- 引用类型: `slice`, `map`, `channel`。函数传参时会复制其引用/指针信息（如切片的底层数组指针），但它们引用的底层数据是共享的。修改引用类型可能影响原数据。

### codeGo

```

func modifyArray(a [3]string) { // 数组是值类型，传入的是副本
 a[1] = "x"
}

func main() {
 array1 := [3]string{"a", "b", "c"}
 modifyArray(array1)
 fmt.Println(array1) // 输出: [a b c] (未改变)
}

```

```

func modifySlice(s []string) { // 切片是引用类型，底层数据共享
 s[1] = "x"
}

func main() {
 slice1 := []string{"a", "b", "c"}
 modifySlice(slice1)
 fmt.Println(slice1) // 输出: [a x c] (已改变)
}

```

## Go Test

Go 语言内置了丰富的测试支持，用于编写单元测试、基准测试和示例测试。

- 测试文件命名: `xxx_test.go`。
- 测试函数命名:
  - 功能测试: `func TestXxx(t *testing.T)`

- **基准测试:** `func BenchmarkXxx(b *testing.B)`
- **示例测试:** `func ExampleXxx()` (通常用于展示用法和预期输出)
- **运行测试:** 在终端运行 `go test` 或 `go test -v` (显示详细信息)。

## codeGo

```
// my_math.go
func Add(a, b int) int {
 return a + b
}

// my_math_test.go
package main

import "testing"

func TestAdd(t *testing.T) {
 if Add(1, 2) != 3 {
 t.Errorf("Add(1, 2) = %d; want 3", Add(1, 2))
 }
}
```

## Git

Git 是分布式版本控制系统，用于管理代码版本和团队协作。

- **安装:** 访问 Git 官网下载安装。
- **基本操作:** `git init`, `git add`, `git commit`, `git push`, `git pull`, `git clone`。
- **IDE 集成:** 大部分 IDE（如 Goland）都内置了 Git 图形界面或命令行工具，方便操作。
- **学习资源:** Learn Git Branching (一个交互式 Git 教程)。

---

## 包

### 包介绍

Go 语言使用包 (package) 来组织代码，实现模块化和代码复用。每个 `.go` 文件都必须属于一个包。

## 定义包

- 一个包通常对应一个文件夹，文件夹内的所有`.go`文件都声明同一个包名。
- 包名为`main`的包是可执行程序的入口点。
- 包名通常应简洁且有意义，例如`config`, `logger`, `utils`。
- **禁止循环导入**：包之间不能形成循环依赖。

## 可见性

Go 语言通过标识符（变量、常量、函数、类型等）的**首字母大小写**来控制可见性：

- **首字母大写**：对外可见（可导出），其他包可以访问。
- **首字母小写**：包内可见（不可导出），只能在当前包内部访问。

### codeGo

```
var PublicVar string // 对外可见
var privateVar string // 包内可见
func PublicFunc() {} // 对外可见
func privateFunc() {} // 包内可见
```

## 包的引入

使用 `import` 关键字在当前文件引入其他包。

- 语法: `import "path/to/package"`

### codeGo

```
import "fmt" // 导入标准库包
import "myproject/mypackage" // 导入自定义模块的包（路径基于 go.mod）
```

- **批量引入**:

### codeGo

```
import (
 "fmt"
 "net/http"
)
```

- 别名导入: `import alias "path/to/package"`, 用于解决包名冲突或简化长包名。

codeGo

```
import f "fmt" // 使用 f.Println() 代替 fmt.Println()
```

- 匿名导入: `import _ "path/to/package"`。
  - 只执行包的 `init` 函数, 不导入包中的其他标识符。
  - 常用于引入数据库驱动等, 只为初始化其内部状态。
  - `init` 函数会在 `main` 函数之前执行, 多个 `init` 函数按声明顺序执行。

codeGo

```
import _ "github.com/go-sql-driver/mysql" // 注册数据库驱动
```

- 获取外部包: `go get "module/path"`。
- 清理依赖: `go mod tidy`。

---

## 指针

### 指针地址和指针类型

- 地址: 内存中存储变量的位置。
- 取地址符 `&`: 获取变量的内存地址。
- 指针类型: `*T` 表示指向 `T` 类型变量的指针。

codeGo

```
a := 10
b := &a // b 是指向 a 的指针, 类型为 *int
fmt.Printf("a 的值: %d, a 的地址: %p\n", a, &a)
fmt.Printf("b 的值 (a 的地址): %p, b 的类型: %T\n", b, b)
```

### 指针取值

- 取值符 `*`: 根据指针指向的地址, 获取存储在该地址上的值。

codeGo

```
a := 10
```

```
b := &a // b 存储 a 的地址
fmt.Println(*b) // 输出: 10 (通过 b 获取 a 的值)
*b = 20 // 通过 b 修改 a 的值
fmt.Println(a) // 输出: 20
```

- `&`和`*`是一对互补操作符。

## 指针的作用

- **修改原变量:** 函数接收指针作为参数时, 可以直接修改原变量的值, 而非其副本。

### codeGo

```
func updateValue(p *int) {
 *p++ // 直接修改指针 p 指向的变量的值
}

func main() {
 x := 0
 updateValue(&x) // 传入 x 的地址
 fmt.Println(x) // 输出: 1
}
```

## “指针就做指针的事”

Go 语言中的指针不能进行偏移和运算 (如 `p+1`)。这是为了:

- **内存安全:** 避免越界访问, Go 通过切片和边界检查提供更安全的内存操作。
- **可移植性:** 不同架构下的指针算术可能表现不同。
- **GC (垃圾回收) 友好:** GC 需要精确追踪哪些是有效指针, 指针运算会使 GC 难以判断。
- **简化复杂性:** 鼓励使用切片等更高级的抽象来处理内存块。

---

# 结构体, 方法, 接口——面向对象思想在 Go 中的浪漫诠释

Go 语言没有传统意义上的“类”和“继承”, 但通过**结构体**、**方法**和**接口**的组合, 实现了强大的面向对象编程 (OOP) 思想。

## 前置知识

## 自定义类型

使用 `type` 关键字定义一个全新的类型，它具有原类型的特性但被视为不同类型。

### codeGo

```
type MyInt int // MyInt 是一个新类型，底层是 int
```

## 类型别名

使用 `type 别名 = 类型` 定义一个类型别名，它与原类型是完全相同的。

### codeGo

```
type MyString = string // MyString 只是 string 的一个别名
```

- **区别：**自定义类型是新类型，别名是同类型。自定义类型可用于为基础类型添加方法。

## 结构体 (`struct`)

结构体是自定义的复合数据类型，用于将多个不同类型的数据字段组合在一起，表示一个事物的完整属性。

- **定义：**

### codeGo

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

- **初始化：**

### codeGo

```
// 键值对初始化
p1 := Person{Name: "Alice", Age: 30}
// 字段顺序初始化（不推荐，容易出错）
p2 := Person{"Bob", 25}
// 声明后赋值
var p3 Person
```

```
p3.Name = "Charlie"
```

```
p3.Age = 22
```

- **零值**: 未初始化的结构体字段会自动赋值为对应类型的零值（如 `string` 为`""`，`int`为`0`）。
- **嵌套结构体**: 一个结构体可以包含另一个结构体作为字段。

## codeGo

```
type Address struct {
 City string
 Street string
}

type Employee struct {
 Person // 匿名嵌套，Employee 直接拥有 Person 的字段
 Address // 匿名嵌套
 ID string
}

func main() {
 emp := Employee{
 Person: Person{Name: "David", Age: 35},
 Address: Address{City: "Beijing", Street: "Main St"},
 ID: "E001",
 }
 fmt.Println(emp.Name) // 直接访问内嵌结构体的字段（字段提升）
 fmt.Println(emp.City) // 也可以直接访问 Address 的字段
}
```

## 方法 (`Method`)

方法是作用于特定类型的函数，这种特定类型被称为“接收者”。

- **定义**: `func (接收者变量 接收者类型) 方法名(参数列表) (返回值列表) { ... }`

## codeGo

```
type Circle struct {
 Radius float64
}

// 值接收者方法（不会修改 Circle 本身）
```

```
func (c Circle) Area() float64 {
 return 3.14 * c.Radius * c.Radius
}

// 指针接收者方法（可以修改 Circle 本身）
func (c *Circle) SetRadius(r float64) {
 c.Radius = r
}
```

- **接收者类型:**
  - **直接接收者:** 方法内部操作的是接收者变量的副本，不会修改原变量。适用于读操作或接收者是小对象。
  - **指针接收者:** 方法内部操作的是接收者变量的指针，可以修改原变量。适用于写操作、接收者是大型对象（避免拷贝开销）或需要保持一致性时。
- **调用:** `myCircle.Area()` 或 `myCircle.SetRadius(10)`。
- **非本地类型不能定义方法:** 不能为其他包定义的类型添加方法。

## 接口 (`Interface`)

接口定义了一组方法的集合。它关注“能做什么”，而不是“是什么”。

- **定义:**

codeGo

```
type Shape interface {
 Area() float64
 Perimeter() float64
}
```

- **实现接口:** 一个类型只要实现了接口中定义的所有方法，就被认为实现了该接口。Go 语言的接口实现是**隐式的**，无需显式声明。

codeGo

```
type Rectangle struct {
 Width, Height float64
}

// Rectangle 实现了 Shape 接口的所有方法
func (r Rectangle) Area() float64 { return r.Width * r.Height }
func (r Rectangle) Perimeter() float64 { return 2 * (r.Width + r.Height) }
```

```
func main() {
 var s Shape // s 是一个接口类型变量
 s = Rectangle{Width: 3, Height: 4} // Rectangle 类型的值可以赋值给 Shape
 接口变量
 fmt.Println("Area:", s.Area())
}
```

- **接口是动态类型**: 接口变量可以持有任何实现了该接口的具体类型的值。
- **接口的意义**:
  1. **多态性**: 不同类型实现相同接口, 可被统一处理。
  2. **解耦合**: 模块间通过接口而非具体实现进行交互, 降低依赖。
  3. **扩展性**: 添加新功能时, 只需实现接口即可, 无需修改原有代码。
  4. **接口断言**: 在运行时检查接口变量持有的具体类型。

---

## 面向对象思想 (OOP)

Go 语言的 OOP 核心思想是: 用结构体承载状态, 用方法定义行为, 用接口抽象能力。

OOP 三要素在 Go 中的体现:

### 1. 封装 (Encapsulation)

- **定义**: 隐藏内部实现细节, 只通过公开接口暴露必要功能。
- **Go 实现**: 通过标识符首字母大小写 (大写导出, 小写不导出) 控制字段和方法的可见性。外部只能通过大写的方法访问或修改小写 (私有) 的字段。

codeGo

```
type Account struct {
 balance int64 // 小写, 包外不可直接访问
}

func (a *Account) Deposit(amount int64) { // 大写方法, 对外暴露
 a.balance += amount
}

func (a *Account) Balance() int64 { // 大写方法, 对外暴露
 return a.balance
}
```

### 2. 组合代替继承 (Composition over Inheritance)

- **定义:** Go 没有传统的类继承，通过**结构体内嵌**来实现代码复用和功能扩展。
- **Go 实现:** 将一个结构体作为另一个结构体的匿名字段嵌入，被嵌入的结构体字段和方法会自动“提升”到外层结构体。

## codeGo

```

type Logger struct {
 prefix string
}

func (l *Logger) Log(msg string) {
 fmt.Printf("%s: %s\n", l.prefix, msg)
}

type Server struct {
 Logger // 匿名内嵌 Logger，Server 拥有 Logger 的字段和方法
 Addr string
}

func main() {
 s := Server{
 Logger: Logger{prefix: "[SERVER]"},
 Addr: ":8080",
 }
 s.Log("Server started.") // 直接调用内嵌 Logger 的方法
 // 等同于 s.Logger.Log("Server started.")
}

```

### 3. 多态 (Polymorphism)

- **定义:** 不同类型的对象，通过实现同一接口，能够以统一的方式被操作。
- **Go 实现:** 通过**接口**和**隐式实现**来达到。函数或方法可以接收接口类型参数，从而接受任何实现了该接口的具体类型。

## codeGo

```

type Greeter interface {
 Greet() string
}

type EnglishSpeaker struct { Name string }

func (e EnglishSpeaker) Greet() string { return "Hello, " + e.Name }

```

```
type SpanishSpeaker struct { Name string }
func (s SpanishSpeaker) Greet() string { return "Hola, " + s.Name }

func SayHello(g Greeter) { // 接收接口类型
 fmt.Println(g.Greet())
}

func main() {
 eng := EnglishSpeaker{Name: "Alice"}
 spa := SpanishSpeaker{Name: "Bob"}
 SayHello(eng) // 输出: Hello, Alice
 SayHello(spa) // 输出: Hola, Bob
}
```

---

## 泛型

Go 语言在 1.18 版本引入了泛型，使得代码可以更灵活地处理多种类型，同时保持类型安全和性能。泛型是“语法糖级别的工具”，让你能够写一份代码，适配多种类型。

### 类型参数列表 / 泛型函数

泛型函数通过类型参数列表定义，通常用 `T` 表示一个或多个类型。类型参数列表放在函数名后的方括号 `[]` 中。

- **类型约束**: 使用接口或自定义的类型集来约束类型参数所能代表的具体类型。

#### codeGo

```
// Max 函数接受任何实现了 constraints.Ordered 接口的类型 T (int, float 等)
func Max[T comparable](a, b T) T { // T 必须是可比较类型 (==, !=)
 if a > b { // 这里要求 T 支持 > 运算符
 return a
 } else {
 return b
 }
}

// 自定义类型约束
type Number interface {
```

`~int | ~int8 | ~float32` // `~`表示包含底层类型为 `int/int8/float32` 的所有类型，包括自定义类型

}

`func Sum[T Number](nums []T) T` { // `Sum` 函数接受切片，元素类型 `T` 需满足 `Number` 约束

```
var total T
for _, n := range nums {
 total += n
}
return total
```

}

`func main()` {

```
fmt.Println(Max(10, 20)) // T 被推导为 int
fmt.Println(Max(3.14, 2.71)) // T 被推导为 float64
fmt.Println(Sum([]int{1, 2, 3})) // T 被推导为 int
```

}

- `comparable`: 内置接口，约束类型必须支持`==`和`!=`操作符。
- `~` (波浪号): 在类型约束中，`~`表示允许包含底层类型为指定类型的所有类型，包括基于该底层类型定义的自定义类型。

## 泛型结构体

结构体也可以使用类型参数，使其可以存储不同类型的数据。

### codeGo

```
type List[T any] struct { // T 可以是任何类型 (any 是 interface{} 的别名)
 data []T
}
```

```
type Map[K comparable, V any] struct { // K 必须是可比较类型，V 可以是任何类型
 hashMap map[K]V
}
```

```
func main() {
 myIntList := List[int]{data: []int{1, 2, 3}}
 myStringMap := Map[string, string]{hashMap: make(map[string]string)}
 myStringMap.hashMap["name"] = "Alice"
}
```

## 泛型底层简单解析（看看即可）

- **编译时优化 (Shape Stenciling):** Go 编译器会为具有相同内存布局和 GC 指针位图的类型参数（即同一“shape”）生成一份**共享的机器码**，从而实现性能接近手写代码。例如，`int`, `int32`, `float64` 等标量类型通常可以共享同一份泛型代码。
- **运行时字典:** 对于需要特定类型操作（如 `map` 的哈希函数、接口方法调用、比较操作 `==`/`<` 等）的类型参数，编译器会传入一个**算法字典**或生成**专用辅助函数**。这使得泛型在保持类型安全的同时，仍能处理复杂类型特有的行为。