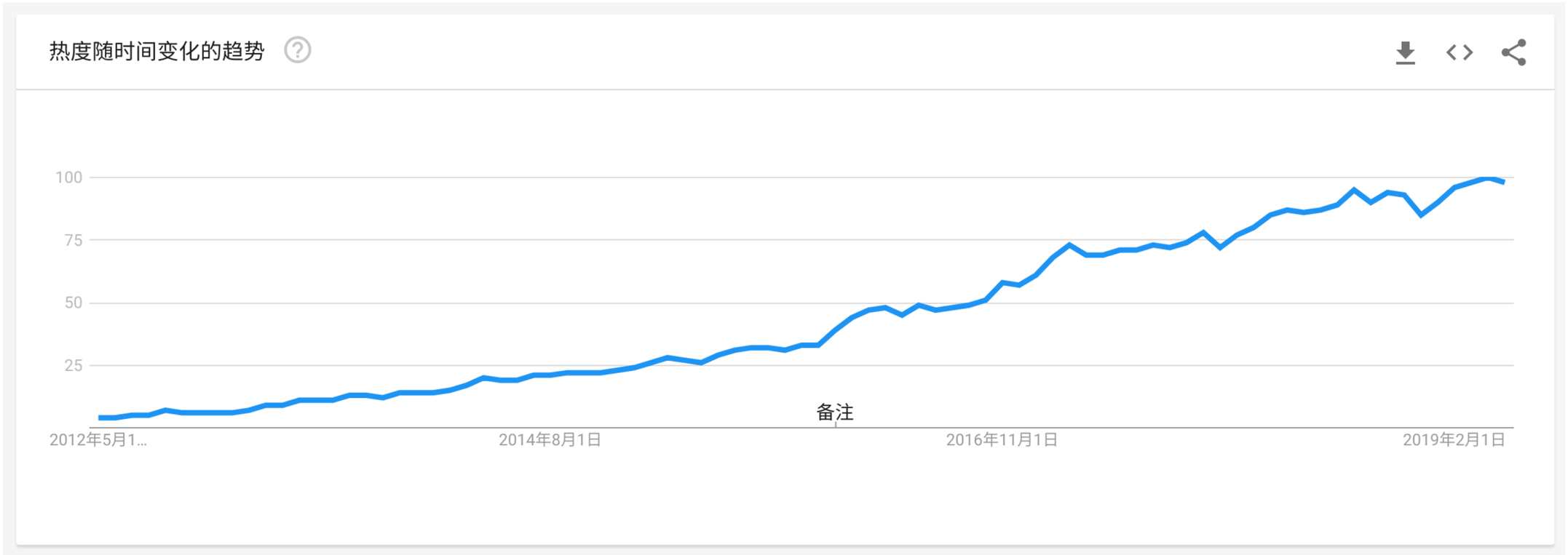
Golang 实战

优点知识 - 阳明

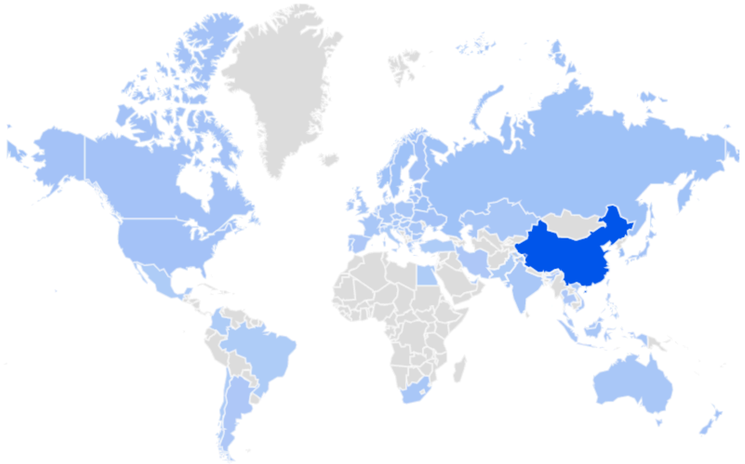
https://youdianzhishi.com


热度随时间变化的趋势
100
75
50
25
备注
2012年5月1...
2014年8月1日
2016年11月1日
2019年2月1日

Golang 趋势

按区域显示的搜索热度

区域



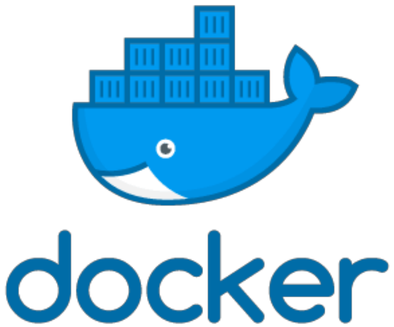
1	中国	100
2	圣赫勒拿	16
3	新加坡	16
4	韩国	12
5	香港	11

包括搜索量较低的区域

当前显示的是第 1-5 个地区（共 61 个）

Golang 中国

Golang 项目



课程设计

- 基本语法
- 面向接口编程
- 函数式编程
- 并发编程（goroutine+channel）
- 文本处理（工具）
- Nginx 日志处理（协程）
- Go Web（web）

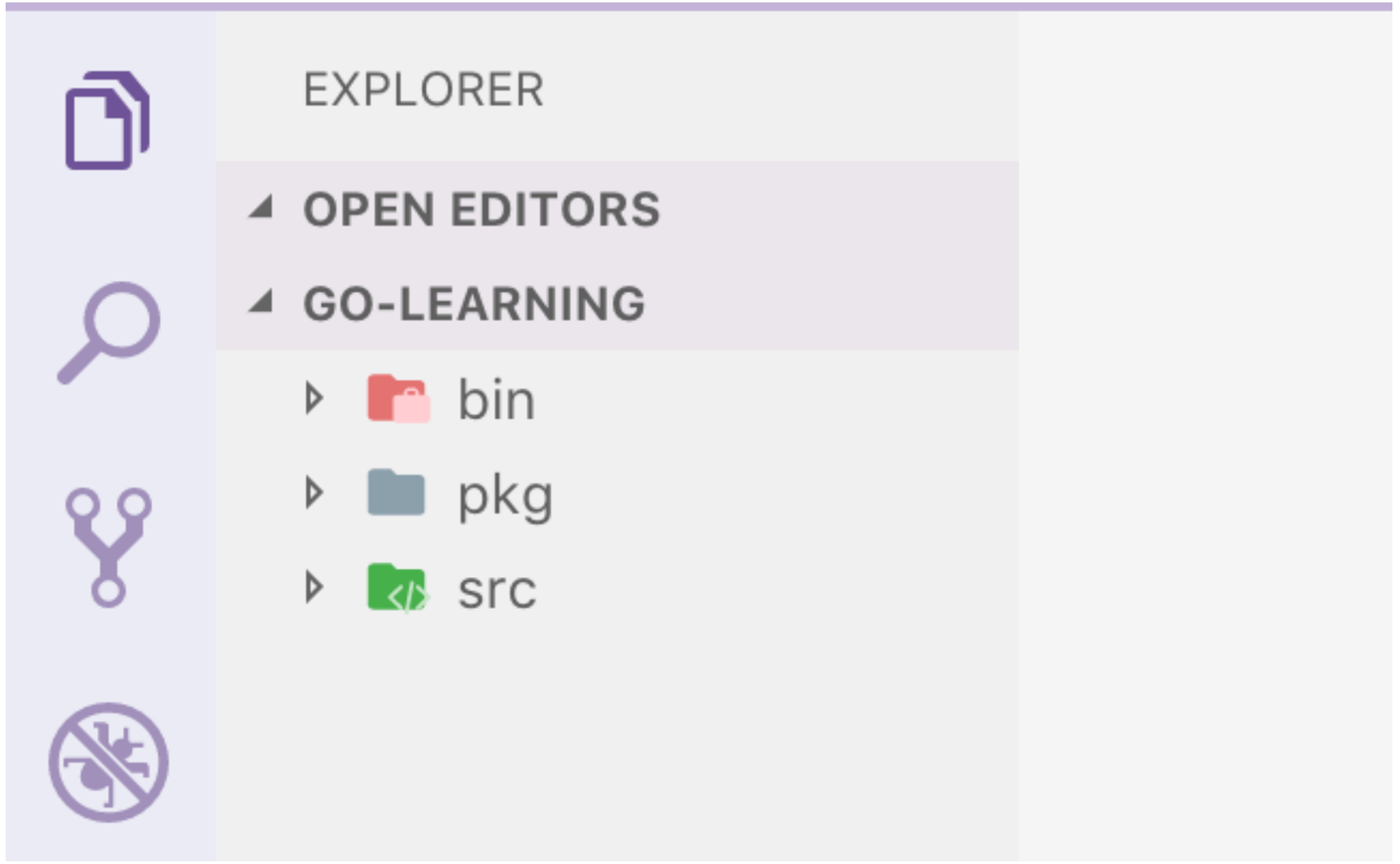
Golang 安装&开发工具

- 安装：https://golang.google.cn/dl/
- 开发工具：vim、emacs、sublime text、atom、vscode(推荐1)、GoLand(推荐2)

Golang 开发环境

- GOROOT：go 的安装目录，设置这个环境变量自定义 go 路径
- GOPATH：go 的工作目录（项目目录），编译或运行时从这个环境变量中去查找包、依赖

Go 项目基本结构



Go语言基本形式

```go
package main ❶

import "fmt" ❷

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

❶ 使用 package 关键字定义包

❷ 导入打印需要用到的系统包：fmt

❸ 调用系统包 fmt 中的 Println 方法打印字符串

构建并运行

- go build：编译指定的源文件或代码包以及依赖包
- go install：安装自身包和依赖包
- go run：编译并运行 go 程序
- vscode 插件：Code Runner
- GoLand IDE

Go tools 配置

```
{
    "go.gopath": "${workspaceFolder}",
    "go.inferGopath": true,
    "go.autocompleteUnimportedPackages": true,
    "go.gocodePackageLookupMode": "go",
    "go.gotoSymbol.includeImports": true,
    "go.useCodeSnippetsOnFunctionSuggest": true,
    "go.useCodeSnippetsOnFunctionSuggestWithoutType": true,
    "go.docsTool": "gogetdoc"
}
```

1. "cmd+shift+p": “go:install/update tools” 安装插件
2. "go.toolsGopath": “指定tools包的路径，不指定默认在 GOPATH”

变量

```
var a int、var a, b int = 1, 2
var a, b, c = 1, "go", true
a, b, c := 1, "go", true ❶
var (
    a = 1
    b = "go"
    c = true
)
```

❶ 只能在函数内使用，不能用在函数外

系统变量类型

- bool
- string
- (u)int、(u)int8、(u)int16、(u)int32、(u)int64、uintptr
- byte(uint8)、rune(int32 unicode)
- float32、float64、complex64、complex128

1. Go语言不允许隐式类型转换，只能强制转换
2. Go语言中不支持指针运算

常量

```
const s string = "Hello"
const a, b = 3, 4  ❶
const (
    s1  = "golang"
    c   = 5
    MAX = 10        ❷
)
```

.....

❶ 不指定类型的常量，则它的类型是不确定的,可以做各种类型使用

❷ Go语言中定义常量一般不用大写，大写表示public，可导出的，有特殊含义

枚举

```
const (
    Monday = 1 + iota  ❶
    Tuesday
    Wednesday
    Thursday
    Friday
    Staurday
    Sunday
)
```

❶ 枚举是一种特殊的常量，可以通过iota快速设置连续的值

类型定义与别名

```
type MyInt1 int                ❶
type MyInt2 = int              ❷

var i int = 1
var i1 MyInt1 = MyInt1(i)      ❸
var i2 MyInt2 = i
```

❶ 类型定义：基于int创建的一个新类型，主要提高代码可读性

❷ 类型别名：基于int创建的一个别名，和原类型完全一样，主要用于包兼容

❸ 类型定义是一个新的类型了，所以类型转换的时候必须强制类型转换

算术运算符

下表列出了所有Go语言的算术运算符。假定 A 值为 10，B 值为 20。

运算符	描述	实例
+	相加	A + B 输出结果 30
-	相减	A - B 输出结果 -10
*	相乘	A * B 输出结果 200
/	相除	B / A 输出结果 2
%	求余	B % A 输出结果 0
++	自增	A++ 输出结果 11 [1]
--	自减	A-- 输出结果 9 [1]

[1] Go语言中没有前置的++和--，++a、--a是错误的

关系运算符

下表列出了所有Go语言的关系运算符。假定 A 值为 10，B 值为 20。

运算符	描述	实例
==	检查两个值是否相等，如果相等返回 True 否则返回 False。	(A == B) 为 False
!=	检查两个值是否不相等，如果不相等返回 True 否则返回 False。	(A != B) 为 True
>	检查左边值是否大于右边值，如果是返回 True 否则返回 False。	(A > B) 为 False
<	检查左边值是否小于右边值，如果是返回 True 否则返回 False。	(A < B) 为 True
>=	检查左边值是否大于等于右边值，如果是返回 True 否则返回 False。	(A >= B) 为 False
<=	检查左边值是否小于等于右边值，如果是返回 True 否则返回 False。	(A <= B) 为 True

逻辑运算符

下表列出了所有Go语言的逻辑运算符。假定 A 值为 True，B 值为 False。

运算符	描述	实例
&&	逻辑 AND 运算符。 如果两边的操作数都是 True，则条件 True，否则为 False。	(A && B) 为 False
||	逻辑 OR 运算符。 如果两边的操作数有一个 True，则条件 True，否则为 False。	(A || B) 为 True
!	逻辑 NOT 运算符。 如果条件为 True，则逻辑 NOT 条件 False，否则为 True。	!(A && B) 为 True

位运算符

下表列出了位运算符 &, |, 和 ^ 的计算：

p	q	p & q	p	q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Go 语言支持的位运算符如下表所示。假定 A 为60，B 为13：

运算符	描述	实例
&	按位与运算符"&"是双目运算符。 其功能是参与运算的两数各对应的二进位相与。	(A & B) 结果为 12, 二进制为 0000 1100
|	按位或运算符"|"是双目运算符。 其功能是参与运算的两数各对应的二进位相或	(A | B) 结果为 61, 二进制为 0011 1101
^	按位异或运算符"^"是双目运算符。 其功能是参与运算的两数各对应的二进位相异或，当两对应的二进位相异时，结果为1。	(A ^ B) 结果为 49, 二进制为 0011 0001
<<	左移运算符"<<"是双目运算符。左移n位就是乘以2的n次方。 其功能把"<<"左边的运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。	A << 2 结果为 240 ， 二进制为 1111 0000
>>	右移运算符">>"是双目运算符。右移n位就是除以2的n次方。 其功能是把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数。	A >> 2 结果为 15 ， 二进制为 0000 1111

赋值运算符

运算符	描述	实例
=	简单的赋值运算符，将一个表达式的值赋给一个左值	C = A + B 将 A + B 表达式结果赋值给 C
+=	相加后再赋值	C += A 等于 C = C + A
-=	相减后再赋值	C -= A 等于 C = C - A
*=	相乘后再赋值	C *= A 等于 C = C * A
/=	相除后再赋值	C /= A 等于 C = C / A
%=	求余后再赋值	C %= A 等于 C = C % A
<<=	左移后赋值	C <<= 2 等于 C = C << 2
>>=	右移后赋值	C >>= 2 等于 C = C >> 2
&=	按位与后赋值	C &= 2 等于 C = C & 2
^=	按位异或后赋值	C ^= 2 等于 C = C ^ 2
|=	按位或后赋值	C |= 2 等于 C = C | 2

条件语句

```
if 布尔表达式 {
    /* 在布尔表达式为 true 时执行 */
}
```

```
if 布尔表达式 {
    /* 在布尔表达式为 true 时执行 */
} else if 另外一个布尔表达式 {
    /* 在布尔表达式为 true 时执行 */
} else {
    /* 在布尔表达式为 false 时执行 */
}
```

条件语句

```
switch {
case true:                          ❶
        fmt.Println("1、case 条件语句为 false”)
        fallthrough                 ❷
case false:
        fmt.Println("2、case 条件语句为 false")
case true:
        fmt.Println("3、case 条件语句为 true")
case true:
        fmt.Println("4、case 条件语句为 true")
default:
        fmt.Println(“5、默认 case”)
}
```

❶ 直到找到匹配项，匹配项后面也不需要加 break，相当于默认就有 break

❷ 默认情况下 case 匹配成功后就不会执行其他 case，如果我们需要执行后面的 case，可以使用 fallthrough，fallthrough 不会判断下一条 case 的表达式结果是否为 true。

循环语句

```go
sum := 0
for i := 1; i <= 100; i++ {    ❶
    sum += i
}

n := 0
for n < 10 {    ❷
    n++
}

for {    ❸
    fmt.Println("dead loop")
}
```

❶ 不需要括号包裹起来

❷ 没有初始值，相当于 while 循环

❸ 没有初始值，没有循环条件，表示死循环

循环控制语句

- break语句：用于中断当前 for 循环
- continue语句：跳过当前循环的剩余语句，然后继续进行下一轮循环
- goto语句：将控制转移到被标记的语句

函数

```
func operate(a, b int, op string) int
func swap(a, b int) (x, y int)
func compute(op func(int, int) int, a, b int) int
func sum(nums ...int) int
```

1. 函数返回值类型写在最后面
2. 函数可以返回多个值，也可以给返回值命名，一般和error结合使用
3. 函数式编程，函数也可以作为参数传递给其他函数
4. go语言中没有默认参数，可选参数，但是可以使用可变参数列表

变量作用域

- 局部变量：函数内定义的变量（作用域只在函数体内，参数和返回值变量也是局部变量）
- 全局变量：函数外定义的变量（全局变量可以在整个包甚至外部包（被导出后）使用）
- 形式参数：函数定义中的变量（作为函数的局部变量来使用）

指针

```
var p int = 20 /* 声明实际变量 */
var ip *int    /* 声明指针变量 */    ❶
ip = &p        /* 指针变量的存储地址 */ ❷
*ip = 30       /*给指针变量赋值*/    ❸
fmt.Println(p)
```

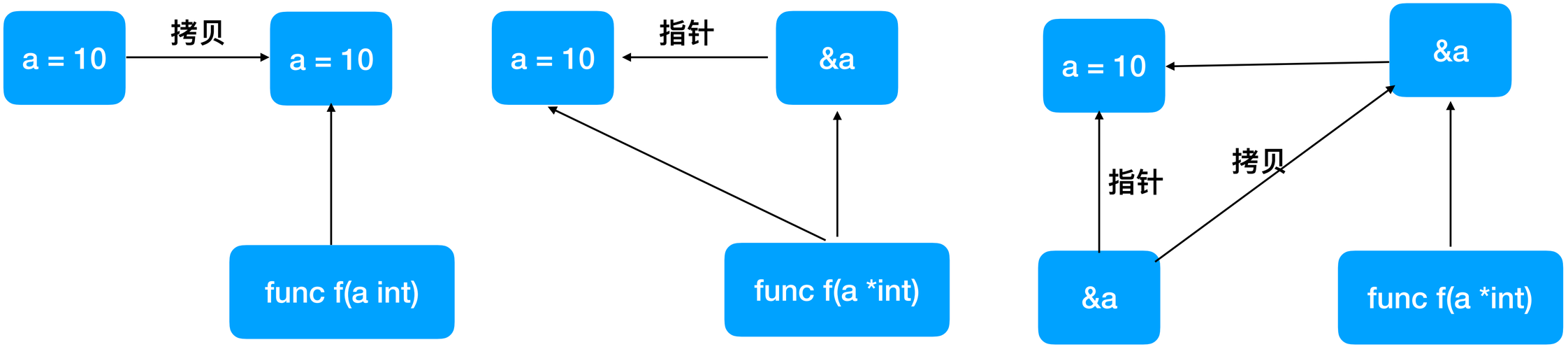
❶ 指针变量指向一个值的内存地址，* 号用于指定变量是作为一个指针

❷ & 符号是用于取后面的变量的内存地址

❸ 将指针 ip 指向的内容更改，则变量 p 也会更改，因为是同一块内存地址

值传递和引用传递

go 语言中只有值传递一种方式



数组

```
var arr1 [5]int                    ❶

arr2 := [3]int{1, 2, 3}            ❷

arr3 := [...]int{4, 5, 6, 7, 8}    ❸

arr4 := [3][4]int{                 ❹
    {0, 1, 2, 3},    /*  第一行索引为 0 */
    {4, 5, 6, 7},    /*  第二行索引为 1 */
    {8, 9, 10, 11}, /* 第三行索引为 2 */
}
```

❶ 数组声明需要指定元素类型及元素个数，var var_name [SIZE] var_type

❷ 初始化数组中 {} 中的元素个数不能大于 [] 中的数

❸ 可以使用 . . . （不能省略）来忽略数组大小，会根据元素的个数来设置数组的大小

❹ 多维数组，本质上也是一维数组

切片

```
var s0 []int // s0 = []int{1, 2, 3}      ❶
s1 := make([]int, 5, 5)    ❷

arr := [...]int{0, 1, 2, 3, 4, 5, 6, 7}
s2 := arr[2:6]
s3 := arr[2:]
s4 := arr[:6]    ❸
s5 := arr[:]
s6 := s4[3:5]    ❹
```

.....

❶ 初始化切片，[]表示是切片类型，不需要指定大小

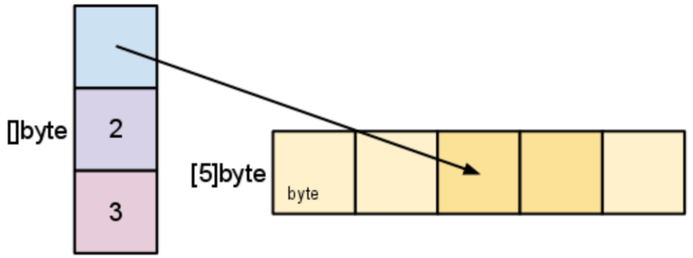
❷ 也可以通过make关键字进行初始化，make([]T, length, capacity)

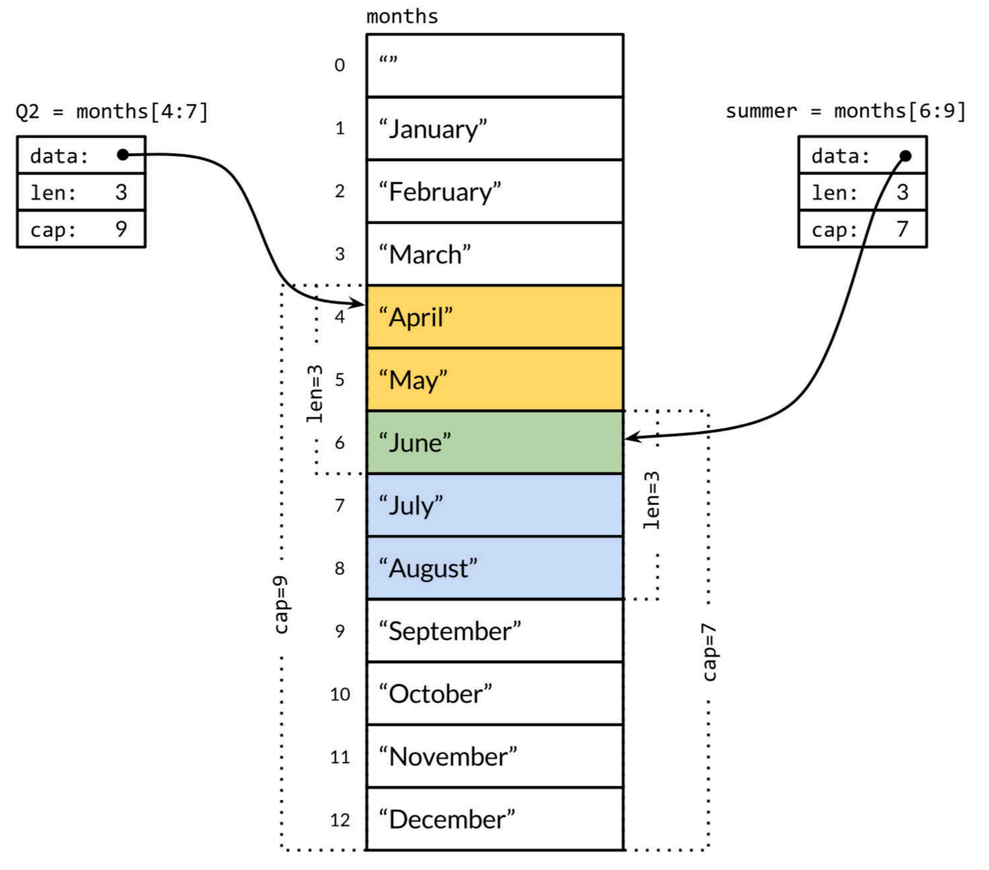
❸ 指定数组的索引初始化切片，是数组的引用

❹ 切片之上还可以继续指定切片

切片的数据结构

```
type slice struct {
    array unsafe.Pointer
    len   int
    cap   int
}
```





切片的操作

```
s1 = append(s1, 50)          1
var s2 []int                 2
copy(s1, s2)                 3
s2 = append(s2[:1], s2[2:]...)   4
```

1. 添加元素时，如果超过cap，会重新分配一个新的底层数组
1. 因为是值传递，所以需要用一个变量接收append后的值
2. 切片在未初始化之前默认为nil，长度为0，可以通过s == nil进行判断
3. 将切片s2拷贝到切片s1
4. 可以利用append来达到删除元素的功能

Map

```
var m1 map[string]int          ❶
m2 := make(map[string]int)     ❷
m3 := map[string]int{
    "a": 1,
    "b": 2,
    "c": 3,
}
val, ok := m4["key"]           ❸
delete(m3, "a")                ❹
```

❶ 直接通过var来声明Map，不初始化就是零值：nil，不能存放k/v

❷ 通过make关键字来初始化Map，这个时候是空Map，可以存放k/v;

❸ 通过 val, ok := map[“key”]来判断键值是否存在

❹ 直接使用delete可以删除Map中的key

Map

- 可以使用range关键字遍历Map
- Map是无序的，哈希表，可以通过对 key 进行排序进行取值
- 除了slice、map、function之外的内建类型都可以作为key

结构体

```
type Book struct {
    id      int
    title   string
    author  string
    subject string
}

book := new(Book)

func (book *Book) String() string {
    return fmt.Sprintf("id=%d,title=%s,author=%s,subject=%s",
        book.id, book.title,
        book.author, book.subject)
}
```

1. go语言不支持继承和多态，但是有接口的概念
2. 使用new函数给一个新的结构体变量分配内存，它返回指向已分配内存的指针
3. 给结构体定义的函数叫方法，前面需要带上结构体指针的声明

结构体扩展

```
type Book struct { // 标签(tag)
    id      int    "书籍编号"
    title   string "书籍标题"
    author  string "书籍作者"
    subject string "书籍主题"
}

type TechBook struct {
    cat  string
    int  // 匿名字段
    Book // 匿名字段
}
```

1

2

3

1. 结构体中的字段除了有名字和类型外，还可以有一个可选的标签（tag）
2. 结构体可以包含一个或多个匿名（或内嵌）字段
3. 匿名类型的可见方法也同样被内嵌，这在效果上等同于 继承

接口

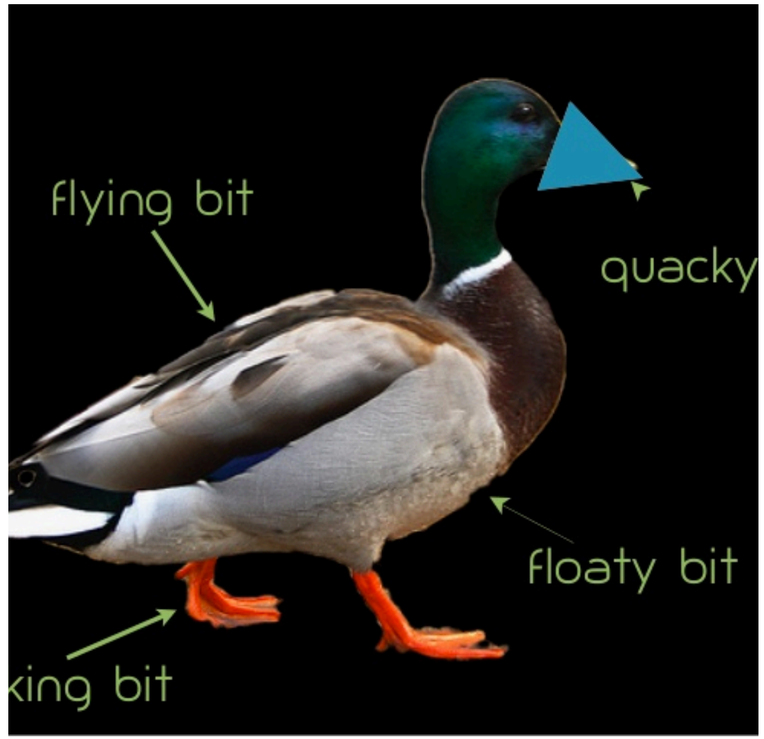
```
package main

type Phone interface {
    Call()
}
type MiPhone struct {
}
func (mp *MiPhone) Call() {
    fmt.Println("I am iPhone, I can
call you!")
}

func main() {
    phone := getPhone()
    phone.Call()
}
```

1. 接口把所有的具有共性的方法定义在一起
2. 任何其他类型只要实现了这些方法就是实现了这个接口

duck typing

- 当一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子

- 我们并不关心对象是什么类型，到底是不是鸭子，只关心行为（描述事物的外部行为，而非内部结构）



类型断言和判断

```go
if v, ok := varI.(T); ok {
    // varI is type T
    Process(v)
}
// varI is not of type T

switch t := varI.(type) {
case *xxx:
    fmt.Printf("Type XXX %T with value %v\n", t, t)
default:
    fmt.Printf("Unexpected type %T\n", t)
}

type Any interface {}
```

1

2

3

1. varI 必须是一个接口变量；应该总是使用该方式进行类型断言
2. 接口变量的类型也可以使用 type-swtich 来检测
3. 不包含任何方法的空接口，类似于Java中的Object基类，可以表示任何类型

接口方法定义

- 指针方法可以通过指针调用
- 值方法可以通过值调用
- 接收者是值的方法可以通过指针调用，因为指针会首先被解引用
- 接收者是指针的方法不可以通过值调用，因为存储在接口中的值没有地址

常用系统接口

```
type Stringer interface {
    String() string
}

type Interface interface {
    Len() int
    Less(i, j int) bool
    Swap(i, j int)
}

type Reader interface {
    Read(p []byte) (n int, err error)
}
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

错误处理

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

if value, err := Function1(param1); err != nil {
    fmt.Printf("An error occured in Function1
with parameter %v", param1)
    return err
}
// 未发生错误，继续执行
```

defer

```
func ReadFile(filename string) (string, error) {
    file, err := os.Open(filename)
    defer file.Close()          ❶
    defer fmt.Println(1)
    defer fmt.Println(2)        ❷
    if err != nil {
        return "", err
    }
    bt, err := ioutil.ReadAll(file)
    if err != nil {
        return "", err
    }
    return string(bt), nil
}
```

❶ defer的内容在当前函数结束之后执行（如果有return，就是在return之前执行）

❷ 多个defer函数是一个栈的形式，先进后出

panic/recover

```
func tryPanic() {
    defer func() {
        if e := recover(); e != nil {
            fmt.Printf("catch panic from
recover: %s\n", e)
        }
    }()
    panic("call a panic")
}
```

3 2 1

1. panic停止当前函数的执行，一直向上返回，执行每一层的defer
2. 如果没有遇见recover，则退出程序
3. recover只能在defer调用中使用，获取panic的值

govendor

- govendor是类似于npm的包管理工具
- 安装govendor： go get -u github.com/kardianos/govendor
- 将项目依赖的外部包拷贝到项目下的vendor目录下，并通过vendor.json文件来记录依赖包的版本

指令	含义
init	创建 `vendor` 文件夹和 `vendor.json` 文件
list	列出已经存在的依赖包
add	从 `$GOPATH` 中添加依赖包，会加到 vendor.json
update	从 `$GOPATH` 升级依赖包
remove	从 `vendor` 文件夹删除依赖
status	列出本地丢失的、过期的和修改的 `package`
fetch	从远端库添加或者更新 `vendor` 文件中的依赖包
sync	本地存在 `vendor.json` 时候拉取依赖包，匹配所记录的版本
get	等同于 `go get`