# golang特点

* 类型检查：编译时
* 运行环境：编译成机器代码直接运行
* 编程范式：面向接口，函数式编程（可以在函数体内定义函数），并发编程
* 计算能力强
* 从语言层面上支持大并发
* C++：必须有性能保障的部分，如搜索引擎
* java：复杂业务逻辑，如adwords, google docs
* Python：大量内部工具
* go：新的内部工具，及其他业务模块，如dl.google.com

golang的应用：

区块链的开发、服务器端开发、分布式/云计算

Go语言保证了既能到达静态编译语言的安全和性能，又达到了动态语言开发速度和易维护性，使用一个表达式来形容Go语言：Go = C + Python , 说明Go语言既有C静态语言程序的运行速度，又能达到Python动态语言的快速开发。

**学习资料：**

|  |  |
| --- | --- |
| api下载地址： | https://github.com/astaxie/godoc |
| 资料下载网站： | https://studygolang.com/dl |
| golang中国： | https://www.golangtc.com |
| 网站：算法题 | https://leetcode.com/ |
| 题：中国 | https://leetcode-cn.com/ |
| 学习资料 | <http://docscn.studygolang.com/doc/> |
| 学习资料 | go tool tour |

# 基本语法

## 变量

* golang中定义的变量都需要使用，引入的包也必须都用到，不然报错
* 定义一个不用的变量：使用 \_ 做变量名，不会报错

### 变量定义

在文件中，必须要以var/func等关键字开头，所以不能在函数外使用 := 来定义变量赋值。

函数外定义的变量叫做包内变量，不是全局变量。

|  |
| --- |
| **package** main  **import** "fmt"  **func** main() {  fmt.Println("hello,world")  } |
| **func** variableZeroValue(){  //每种类型都有初始值，int->0,string->""  **var** a int  **var** b string  fmt.Printf("%d %q\n",a,b)  } |
| **func** variableInitialValue(){  **var** a int = 10  **var** b string = "aa"  fmt.Println(a,b)  } |
| **func** variableShort1(){  //省略类型定义  **var** a, b, c = 10, 20, "kkk"  fmt.Println(a,b,c)  } |
| **func** variableShort2(){  //省略类型定义和var，此种方式只能在函数中使用，函数外不能使用  a, b, c := 10, 20, "kkk"  fmt.Println(a,b,c)  } |
| //包内变量  **var** aa = 2  **var** bb = "bb"  **var** kk = true  //包内变量  **var** (  cc = 2  dd = "dd"  mm = false  ) |
| **func** plx() {  //复数表示的时候虚部写成ni，不能写成n\*i  a := 3 + 4i  fmt.Println(cmplx.Abs(a))  } |

### 内建变量类型

|  |  |
| --- | --- |
| bool, string |  |
| (u)int, (u)int8, (u)int16,  (u)int32, (u)int64,uintptr | (u)→无负数；  不加长度就是和操作系统位数一样；  uintptr→指针 |
| byte, rune | byte→8位  rune→32位，为了表示utf-8中的字符 |
| float32, float64,  complex64, complex128 | complex64→复数，实部float32位，虚部float32位 |

go只有强制类型转换，没有隐式类型转换

|  |
| --- |
| **func** tran() {  **var** a, b int= 3, 4  **var** c int  c = int(math.Sqrt(float64(a\*a+b\*b)))  fmt.Println(c)  } |

## 常量与枚举

|  |
| --- |
| //go中的常量名一般不大写，大写在go中表示public  //常量可以定义在函数内，也可以定义在包内  **const** filename = "abc.txt"  **const** a int = 3  **const** b, c = 4, 5 |
| //golang中没有专有的枚举关键字，使用const块来定义枚举，常量必须赋初值  **const** (  cpp = 0  java = 1  python = 2  golang = 3  ) |
| //iota代表给变量依次自增赋值  **const** (  cpp = **iota**  java  python  golang  )  //iota可以参与运算  **const** (  b = 1 << (10 \* **iota**)  kb  mb  gb  tb  pb  ) |

## 条件语句

### if

|  |
| --- |
| **const** filename = "abc.txt"  contents, err := ioutil.ReadFile(filename)  //if的用法一  **if** err != nil {  fmt.Println(err)  } **else** {  fmt.Printf("%s\n",contents)  } |
| **const** filename = "abc.txt"  //if的用法二，这时定义的变量是if块中的  **if** contents, err := ioutil.ReadFile(filename); err != nil {  fmt.Println(err)  } **else** {  fmt.Printf("%s\n",contents)  } |

### switch

func eval(a, b int, op string) int {

var result int

switch op {

case “+”:

result = a + b

case “-”:

result = a - b

case “\*”:

result = a \* b

case “/”:

result = a / b

default:

panic(“unsupported operator:” + op)

}

return result

}

switch 会自动break，除非使用fallthrough

func grade(score int) string {

g := “”

switch {

case score < 0 || score > 100:

panic(fmt.Sprintf(“wrong score: %d”, score))

case score < 60:

g = “D”

case score < 80:

g = “C”

case score < 90:

g = “B”

case score <= 100:

g = “A”

}

return g

}

此种switch的使用方式相当于if-elseif

## 循环

### for

sum := 0

for i := 1; i <= 100; i++ {

sum += i

}

* for的条件里不需要括号
* for的条件里可以省略初始条件，结束条件，递增表达式

func convertToBin(n int) string {

result := “”

for ; n > 0; n /= 2 {

lsb := n % 2

result = strconv.Itoa(lsb) + result

}

return result

}

func main() {

fmt Println(convertToBin(5), convertToBin(13))

// 101 1101

}

### 只有判断的for

此处的for相当于while，go中没有while

func printFile(filename string) {

file, err := os.Open(filename)

if err != nil {

panic(err)

}

scanner := bufio.NewScanner(file)

for scanner.Scan() {

fmt.Println(scanner.Text())

}

}

### forever

这时相当与死循环

for {

fmt.Println(“abc”)

}

## 基本语法回顾

* for, if 后面的条件没有括号
* if 条件里也可以定义变量
* 没有while
* switch不需要break，也可以直接switch多个条件

# 函数

## 函数定义

func eval(a, b int, op string) int

## 返回多个值

* **返回多个值**

//13 / 3 = 4 ... 1

func div(a, b int) (int, int) {

return a/b, a%b

}

* **推荐使用**

//13 / 3 = 4 ... 1

func div(a, b int) (q, r int) {

return a/b, a%b

}

* **函数体长的话，这种写法容易搞混**

//13 / 3 = 4 ... 1

func div(a, b int) (q, r int) {

q = a/b

r = a%b

return

}

* 函数返回多个值时可以起名字
* 仅用于非常简单的函数
* 对于调用者而言没有区别
* **golang的函数写法：**

两个返回值不要随意用，一般第二个返回值返回的是error信息；这种写法程序不会中断，打印error信息后，继续执行

func eval(a, b int, op string) (int, error) {

switch op {

case “+”:

return a + b, nil

case “-”:

return a - b, nil

case “\*”:

return a \* b, nil

case “/”:

return a / b, nil

default:

return 0, fmt.Errorf(“unsupported operator: %s” , op)

}

}

## 复合函数

把函数当做参数传入函数：

func apply(op func(int, int) int, a, b int) int {

p := reflect.ValueOf(op).Pointer()

opName := runtime.FuncForPC(p).Name()

fmt.Printf(“Callint function %s with args (%d, %d)”, opName, a, b)

return op(a, b)

}

调用1：

func pow(a, b int) int {

return int(math.Pow(float64(a), float64(b)))

}

fmt.Println(apply(pow, 3, 4))

调用2：定义匿名函数

fmt.Println(apply(

func(a int, b int) int {

return int(math.Pow(float64(a), float64(b)))

}, 3, 4 ))

## 可变参数

func sum(numbers ...int) int {

s := 0

for i := range numbers {

s += numbers[i]

}

return s

}

## 函数语法回顾

* 返回值类型写在最后面
* 可返回多个值
* 函数作为参数
* 没有默认参数，可选参数，也没有函数重载

# 指针

## 指针

var a int = 2

var pa \*int = &a

\*pa = 3

fmt.Println(a)

**go的指针不能运算**

1. 不要操作没有合法指向的指针

|  |
| --- |
| //p为int类型变量的地址  **var** p \*int  //这样也可以使p指向安全的内存，相当于动态内存分配  p = new(int)  \*p = 10  fmt.Println(\*p) |

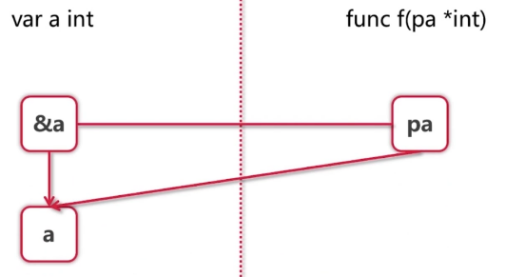
## 函数参数的传递

go函数的参数传递，都是值传递

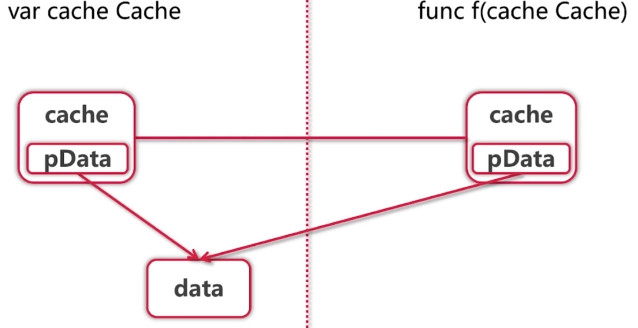
* **值传递**



* **指针传递**



* object传递



* **例子**

交换连个参数的值

方法一：

func swap(a, b \*int) {

\*b, \*a = \*a, \*b

}

方法二：

func swap(a, b int) (int, int) {

return b, a

}

# 数组/切片/容器

## 数组

* **定义数组**

var arr1 [5]int

arr2 := [3]int{1, 3, 5}

arr3 := [...]int{2, 4, 6, 8, 10}

var grid [4][5]int

如果不给初始值，则自动初始化

* **数组的遍历**

arr := [...]int{2, 4, 6, 8, 10}

for i, a := range arr {

fmt.Printf(“numbers[%d] = %d”, i, a)

}

**range**:可以获得下标，也可以获得下标和数据

* 可通过 \_ 省略变量
* 不仅range, 任何地方都可以通过 \_ 省略变量
* 如果只要i , 可以写成

for i := range arr

* **数组是值类型**
* [10]int和[20]int是不同的类型
* 调用func f(arr [10]int)会copy数组
* 在go语言中一般不直接使用数组

在函数中对数组的数据进行改变，不改变外边数组的值

func printArray(arr [5]int) {

arr[0] = 100

}

传指针进去，在函数中改变数组的值，外边的数组一起改变

func printArray(arr \*[5]int) {

arr[0] = 100

}

* **数组的比较**

//数组的比较,比较的时候，数组元素的值及顺序完全一致：true，其他均为false

func arr2() {

    a := [5]int{1, 2, 3, 4, 5}

    a1 := [5]int{1, 2, 3, 5, 4}

    b := [5]int{1, 2, 3, 4, 5}

    c := [5]int{1, 2, 3}

    fmt.Println(a == b, a1 == b, a == c) //true false false

    //使用的是不同的内存空间

    fmt.Printf("%p,%p,%p\n", &a, &a1, &b)

    //同类型的数组可以赋值

    var d [5]int

    d = a

    fmt.Println(d)

}

## 切片Slice

1. go语言中一般不直接使用数组，而是使用切片；
2. 切片是数组的一个视图，倒映的是数组底层的元素；
3. 可以reslice操作
4. 切片当做函数的参数传入的时候，函数中对数据进行改变，原数组的值也会改变，相当于传入的是数组的某部分的指针

|  |
| --- |
| //切片为数组的view，修改切片的数据，数组中的数据也会跟着改变  **func** slice() {      a := [10]int{1, 2, 3, 4, 5, 6, 7, 8, 9, 0}        //s1：只去特定长度作为切片的容量      s1 := a[2:4:5]      fmt.Println(s1) //[3 4]      fmt.Println(len(s1)) //2      fmt.Println(cap(s1)) //3      //如果添加的元素超出切片的容量，将会复制数据到另一个数组，原数组数据不变      fmt.Println(append(s1, 12, 13, 14)) //[3 4 12 13 14]      fmt.Println(a) //[1 2 3 4 5 6 7 8 9 0]      //在容量内修改数据，会直接改变原数组的数据      fmt.Println(append(s1, 12)) //[3 4 12]      fmt.Println(s1) //[3 4]      fmt.Println(a) //[1 2 3 4 12 6 7 8 9 0]  fmt.Println("--------------s2-----------------")  //s2：cap为数组的 len-切片开始位置  s2 := a[2:4]      fmt.Println(append(s2, 12, 13, 14)) //[3 4 12 13 14]      fmt.Println(s2) //[3 4]      fmt.Println(a) //[1 2 3 4 12 13 14 8 9 0]  } |
| //切片的长度是自动扩展的，每次扩展为原来的两倍  **func** add() {      a := []int{}      fmt.Printf("%d, %d\n", len(a), cap(a)) //0 0      a = append(a, 11)      fmt.Printf("%d, %d\n", len(a), cap(a)) //1 1      a = append(a, 12)      fmt.Printf("%d, %d\n", len(a), cap(a)) //2 2      a = append(a, 13)      fmt.Printf("%d, %d\n", len(a), cap(a)) //3 4  } |
| **func** create() {      a := make([]int, 5, 10)      fmt.Printf("%d, %d\n", len(a), cap(a)) //5 10      b := make([]int, 5)      fmt.Printf("%d, %d\n", len(b), cap(b)) //5 5      //向后边添加元素      b = append(b, 1)      fmt.Printf("%d, %d\n", len(b), cap(b)) //6 10      //这样生成的是指针类型      c := new([]int)      fmt.Printf("%p\n", c)      fmt.Printf("%d, %d\n", len(\*c), cap(\*c)) //0 0  } |
| **func** copyDemo() {      src := []int{1, 2}      dst1 := []int{0, 0, 0, 0, 0, 0}      dst2 := []int{}      //把数据源的数据copy到目的的相应位置      copy(dst1, src)      fmt.Println(dst1) //[1 2 0 0 0 0]      //如果目的cap < 源len，源的数据只会copy对应的数量到目的中      copy(dst2, src)      fmt.Println(dst2) //[]  } |

**Delete：**

fmt.Println(“Deleting elements from slice”)

s2 = append(s2[:3],s2[4:]...) //delete s2[3]

fmt.Println(“Popping from front”)

front := s2[0]

s2 = s2[1:]

fmt.Println(“Popping from back”)

tail := s2[len(s2)-1]

s2 := s2[:len(s2)-1]

## Map

|  |
| --- |
| **func** main() {      //map的创建+初始化      m := **map**[int]string{1: "java", 2: "go", 3: "c++"}      //增加元素      m[4] = "php"      //修改元素      m[2] = "python"      //删除键值对      delete(m, 1)        find(m)      args(m)  } |
| **func** find(m **map**[int]string) {      //map的遍历  **for** k, v := **range** m {          fmt.Printf("%d===>%s\n", k, v)      }      //查看某个key是否存在，如果key存在，则返回value，ok为true      value, ok := m[1]  **if** ok {          fmt.Println(value)      } **else** {          //如果key不存在value拿到的是零值          fmt.Println("key不存在", value)      }  } |
| //map当做参数传递到函数中，为引用传递  **func** args(m **map**[int]string) {      m[10] = "c"  } |
| key的使用：   * map使用哈希表，key必须可以比较相等 * 除了slice,map,function的内建类型都可以作为key * struct类型不包含上述字段，也可作为key |
| //map的定义  m2 := make(**map**[int]string) //m2 == empty map  m4 := make(**map**[int]string, 3) //定义一个空的map，并给定初始容量  **var** m3 **map**[int]string //m3 == nil |

**例：寻找最长不含重复字符的子串**

|  |
| --- |
| **func** lengthOfNonRepeatingSubStr(s string) int {  lastOccurred := make(**map**[byte]int)  start := 0  maxLength := 0  **for** i, ch := **range** []byte(s) {  lastI, ok := lastOccurred[ch]  **if** ok && lastI >= start {  start = lastI + 1  }  **if** i-start+1 > maxLength {  maxLength = i - start + 1  }  lastOccurred[ch] = i  }  **return** maxLength } |

## 字符和字符串

斜的单引号中定义的字符串会原样输出；

rune相当与go的char：

* 使用range遍历pos，rune对
* 使用utf8.RuneCountInString获得字符数量
* 使用len获得字节长度
* 使用 []byte 获得字节

**字符串操作的方法：**

* Fields, Split, Join
* Contains, Index
* ToLower, ToUpper
* Trim, TrimRight, TrimLeft

**例：寻找最长不含重复字符的子串 - 国际版**

|  |
| --- |
| **func** lengthOfNonRepeatingSubStr(s string) int {  lastOccurred := make(**map**[rune]int)  start := 0  maxLength := 0  **for** i, ch := **range** []rune(s) {  lastI, ok := lastOccurred[ch]  **if** ok && lastI >= start {  start = lastI + 1  }  **if** i-start+1 > maxLength {  maxLength = i - start + 1  }  lastOccurred[ch] = i  }  **return** maxLength } |

# 面向对象

* go语言仅支持封装，不支持继承和多态
* go语言没有class，只有struct

## 结构体

|  |
| --- |
| //结构体创建与赋值  **func** comm() {      //使用点(.)运算符操作成员  **var** s student      s.id = 1      s.name = "make"      s.sex = 'm'      s.age = 18      s.addr = "xa"      fmt.Println("s = ", s)      //顺序赋值，必须给所有的成员赋值      s1 := student{2, "jack", 'm', 12, "xa"}      fmt.Println("s1 = ", s1)      //给部分成员赋值，其他的成员默认为零值      s2 := student{name: "black", age: 15}      fmt.Println("s2 = ", s2)  } |
| //必须操作有合法指向的指针，使用 . 运算符操作成员  **func** point() {      //1、先定义一个结构体变量，再定义一个指针指向它  **var** s student  **var** p \*student      p = &s      p.id = 11      p.name = "lisi"      fmt.Println("p = ", p)      //用new直接申请一块内存      p1 := new(student)      p1.name = "wangwu"      fmt.Println("p1 = ", p1)  } |
| 不管是指针还是对象，都使用 . 运算符操作成员 |

使用工厂函数

func createNode(value int) \*treeNode {

return &treeNode{value: value}

}

局部变量的创建：是在堆上分配还是在栈上分配由编译器决定；

* 如果只是局部使用的，就分配到堆上；
* 如果返回地址供外部使用，就分配到栈上。这是参与垃圾回收机制；

type treeNode struct {

value int

left, right \*treeNode

}

func (node treeNode) print(){

fmt.Print(node.value)

}

func (node \*treeNode) setValue(value int) {

node.value = value

}

方法是定义在结构体外边的，这里的node相当于this

* go的所有的参数传递都是值传递
* nil指针也可以调用方法
* 要改变结构内容使用指针接收者
* 结构过大考虑使用指针接收者
* 一致性：如有指针接收者，最好都使用指针接收者

## 包

1. 名字一般使用CamelCase（驼峰命名）
2. 一个目录下只能有一个包；
3. 一个项目中有且只能有一个main函数；
4. 同一包下的 .go 文件中的函数调用时，直接调用，不用导包；
5. 为结构定义的方法必须放在同一个包内，可以是不同的文件；
6. 函数或包变量的首字母大写为public，小写为private；
7. 不同包，调用函数需要导入包，且只能调用public的函数；调用时需要包名+函数名 调用
8. 配置GOPATH 和 GOBIN 环境变量

注意：

导入一个包，运行程序的时候，首先会自动调用包中的 init() 函数

**如何扩展系统类型或者别的类型：**

* 定义别名
* 使用组合

## GOPATH

* 默认在 ~/go(unix,linux) , %USERPROFILE%\go(windows)
* 官方推荐：所有的项目和第三方库都放在同一个GOPATH下
* 也可以将每个项目放在不同的GOPATH

获取一个镜像：go get命令的使用，要先安装git

$ go get -v github.com/gpmgo/gopm

* go get命令演示
* 使用gopm来获取无法下载的包
* go build 来编译
* go install 产生pkg文件和可执行文件
* go run直接编译运行

GOPATH下的目录结构

* src
* git repository 1
* git repository2
* pkg
* git repository 1
* git repository2
* bin
* 执行文件1、2、3 ......

# 接口

接口是使用者定义；实现的时候只要实现接口的方法，就默认实现了此接口，不用显式的表达出来；

## 接口基本使用

type Traversal interface {

Traverse()

}

func main() {

traversal := getTraversal()

traversal.Traverse()

}

**duck typing:**

* 像鸭子走路，像鸭子叫(长得像鸭子)，那么就是鸭子
* 描述事物的外部行为而非内部结构
* 严格说go属于结构化类型系统，类似duck typing

**go的duck typing:**

* 同时具有python, c++的duck typing的灵活性
* 又具有java的类型检查
* 接口由使用者定义，实现者实现

|  |
| --- |
| //定义接口  **type** myin **interface** {      printInfo()  }  **type** persion **struct** {      name string      age int      sex byte  }  //实现接口  **func** (p \*persion) printInfo() {      fmt.Println(p)  }  **type** cat **struct** {      name string      money float32      speed int  }  //实现接口  **func** (c \*cat) printInfo() {      fmt.Println(c)  }  **func** pri(m myin) {      m.printInfo()  }  **func** main() {      //只要实现了接口的方法的自定义类型，都可以使用多态      //如果方法使用的是指针，需要传入指针      p := persion{"black", 10, 'm'}      c := cat{"bm", 1222.00, 128}      pri(&p)      pri(&c)  } |

接口的实现：

* 接口的实现是隐式的
* 只要实现接口里的方法

### 类型检查

使用方式一：

if mockRetriever, ok := r.(mock.Retriever); ok {

fmt.Println(mockRetriever.Contents)

} else {

fmt.Println(“not a mock retriever”)

}

使用方式二：

function inspect(r Retriever) {

fmt.Printf(“%T %v\n”, r, r)

switch v := r.(type) {

case mock.Retriever:

fmt.Println(“Contents:”, v.Contents)

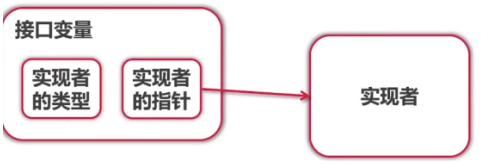
case \*real.Retriever:

fmt.Println(“UserAgent:”, v.UserAgent)

}

}

**接口变量里边有什么：**



* 接口变量自带指针
* 接口变量同样采用值传递，几乎不需要使用接口的指针
* 指针接收者实现只能以指针方式使用；值接收者都可

查看接口变量：

* 表示任何类型：interface{}
* Type Assertion
* Type Switch

支持任何类型的队列：

|  |
| --- |
| **type** Queue []**interface**{} **func** (q \*Queue) Push(v **interface**{}) {  \*q = append(\*q, v) } **func** (q \*Queue) Pop() **interface**{} {  head := (\*q)[0]  \*q = (\*q)[1:]  **return** head } **func** (q \*Queue) IsEmpty() bool {  **return** len(\*q) == 0 } |

## 接口组合

|  |
| --- |
| **type** dy **interface** {      printInfo()  }  **type** xg **interface** {      setName(name string)  }  **type** per **interface** {      dy //匿名字段表示继承      xg      setAge(age int)  } |

使用的时候，可以调用组合的接口的所有方法：

## 标准的接口

|  |
| --- |
| **type** Stringer **interface** {  String() string }  //相当于java的toString |
| **type** Reader **interface** {  Read(p []byte) (n int, err error) } |
| **type** Writer **interface** {  Write(p []byte) (n int, err error) } |

# json

## 编码：生成json

通过struct生成

|  |
| --- |
| **type** IT **struct** {      Company string `json:"-"` //转换json串的时候跳过此字段      Subject []string `json:"subject"` //二次编码，可以使json的字段名首字母小写      IsOk bool `json:",string"` //把数据的格式转换成string格式      Price float64  } |
| **func** main() {      //定义并初始化一个结构体变量      s := IT{"glzj", []string{"go", "java", "python", "c++"}, true, 66.66}      fmt.Println(s)  //生成json串  //js, err := json.Marshal(s)      js, err := json.MarshalIndent(s, "", " ") //转换成格式化的json串，带换行和缩进  **if** err != nil {          fmt.Println(err)  **return**      }      fmt.Println(string(js))  } |

通过map生成

|  |
| --- |
| **func** main() {      //定义一个map，并初始化      m := make(**map**[string]**interface**{}, 4)      m["company"] = "glzj"      m["subject"] = []string{"go", "java", "python", "c++"}      m["isok"] = true      m["price"] = 66.66      //生成json串      //result, err := json.Marshal(m)      result, err := json.MarshalIndent(m, "", "    ") //格式化输出  **if** err != nil {          fmt.Println(err)  **return**      }      fmt.Println(string(result))  } |

## 解码

|  |
| --- |
| **type** IT **struct** {      Company string `json:"company"`      Subject []string `json:"subject"` //二次编码      IsOk bool `json:"isok"`      Price float64 `json:"price"`  }  **func** main() {      jsonBuf := `      {      "company": "glzj",      "subject":["go", "java", "python", "c++"],      "isok":true,      "price":66.66      }`      //解析到struct，定义struct麻烦，但是解析之后，使用简单      //定义几个字段，解析几个字段  **var** s IT      err := json.Unmarshal([]byte(jsonBuf), &s) //第二个参数要传地址  **if** err != nil {          fmt.Println(err)  **return**      }      fmt.Printf("%+v\n", s) //详细打印      //解析到map中，解析方便，使用的时候需要通过类型断言来反推类型，才能赋值      m := make(**map**[string]**interface**{}, 4)      err = json.Unmarshal([]byte(jsonBuf), &m) //第二个参数传地址  **if** err != nil {          fmt.Println(err)  **return**      }      fmt.Printf("%+v\n", m) //详细打印  **var** str string      //类型错误，不能把 interface{}类型直接转换为string，需要通过类型断言反推类型      //str = string(m["company"])  **for** key, value := **range** m {  **switch** data := value.(**type**) {  **case** string:              str = data              fmt.Printf("%s===>%s,类型为string\n", key, str)  **case** bool:              fmt.Printf("%s===>%v,类型为bool\n", key, data)  **case** []string:              fmt.Printf("%s===>%v,类型为[]string\n", key, data)  **case** float64:              fmt.Printf("%s===>%f,类型为float64\n", key, data)  //切片的类型为[]interface{}类型，使用的时候还要类型断言反推  **case** []**interface**{}:           fmt.Printf("%s===>%v,类型为[]interface{}\n", key, data)          }      }  } |

# I/O 操作+读取命令行参数

|  |
| --- |
| **package** main  **import** (      "fmt"      "io"      "os"  )  **func** copyFile(dest, src string) {      //fmt.Println(dest, src)      rf, err := os.Open(src)  **if** err != nil {          fmt.Println("err = ", err)  **return**      }      wf, err := os.Create(dest)  **if** err != nil {          fmt.Println("err = ", err)  **return**      }  **defer** rf.Close()  **defer** wf.Close()      buf := make([]byte, 1024\*4)  **for** {          n, err1 := rf.Read(buf)  **if** err1 != nil {  **if** err1 == io.EOF {                  fmt.Println("end")  **break**              }              fmt.Println("err1 = ", err1)  **return**          }          \_, err2 := wf.Write(buf[:n])  **if** err2 != nil {              fmt.Println("err2 = ", err2)  **return**          }      }  }  **func** main() {      //读取命令行参数      list := os.Args  **if** len(list) != 3 {          fmt.Println("请输入正确的参数！！！")  **return**      }      src := list[1]      dest := list[2]  **if** src == dest {          fmt.Println("原路径和目标路径相同，请重新输入！！！")  **return**      }      copyFile(dest, src)  } |

# 资源管理和出错处理

## 资源管理

* defer调用
* 确保调用在函数结束时发生
* 参数在defer语句时计算
* defer列表为后进先出
* 何时使用defer调用
* Open/Close
* Lock/Unlock
* PrintHeader/PrintFooter

|  |
| --- |
| **func** tryDefer() {  **for** i := 0; i < 100; i++ {  **defer** fmt.Println(i)  **if** i == 30 {  panic("printed too many")  }  } } |

## error及处理

|  |
| --- |
| **func** writeFile(filename string) {  //OpenFile方法可以选择[打开文件][操作文件]的方式类型  file, err := os.OpenFile(filename, os.*O\_EXCL*|os.*O\_CREATE*, 0666)  **if** err != nil {  //PathError为OpenFile 定义的 error，实现了error接口  **if** panicErr, ok := err.(\*os.PathError); !ok {  panic(panicErr)  } **else** {  fmt.Println(panicErr.Op, panicErr.Path, panicErr.Err)  }  **return** }  **defer** file.Close()   writer := bufio.NewWriter(file)  **defer** writer.Flush()   **for** i := 0; i < 20; i++ {  fmt.Fprintln(writer, i)  } } |

## panic及处理

* panic
* 停止当前函数的执行
* 一直向上返回，执行每一层的defer
* 如果没有遇见recover，程序退出
* recover
* 仅在defer调用中使用
* 获取panic的值
* 如果无法处理，可以重新panic

|  |
| --- |
| **defer func**() {  //recover()只能在defer语句的函数中调用，返回panic的信息  r := recover()  **if** err, ok := r.(error); ok {  fmt.Println("Error occurred: ", err.Error())  } **else** {  panic(fmt.Sprintf("I don't know what to do : %v", r))  } }()  //产生抛出error的panic panic(errors.New("This is an error"))  //产生一个panic //panic("This is a panic") |

* error vs panic
* 意料之中的：使用error。如：文件打不开
* 意料之外的：使用panic。如：数组越界

## 错误处理综合示例(统一错误处理)

* defer + panic + recover
* Type Assertion
* 函数式编程的应用
* 基础示例

|  |
| --- |
| **package** main  **import** (  "github.com/gpmgo/gopm/modules/log"  "goDemo/filelistingserver/filelisting"  "net/http"  "os" )  //定义函数类型 **type** handleFunction **func**(writer http.ResponseWriter, request \*http.Request) error  //利用函数式编程，把处理函数包装一下，使error汇总到此函数中集中处理 **func** handleError(handler handleFunction) **func**(writer http.ResponseWriter, request \*http.Request) {  **return func**(writer http.ResponseWriter, request \*http.Request) {  err := handler(writer, request)  **if** err != nil {  log.Warn("Error handling request: %s", err.Error())  code := http.*StatusOK* **switch** {  **case** os.IsExist(err):  code = http.*StatusNotFound* **case** os.IsPermission(err):  code = http.*StatusForbidden* **default**:  code = http.*StatusInternalServerError* }  http.Error(writer, http.StatusText(code), code)  }  } }  **func** main() {  http.HandleFunc("/list/", handleError(filelisting.HandleFileList))  err := http.ListenAndServe(":8888", nil)  **if** err != nil {  panic(err)  } } |
| **package** filelisting  **import** (  "io/ioutil"  "net/http"  "os" )  //如果出错，把错误抛出去，集中处理 **func** HandleFileList(writer http.ResponseWriter, request \*http.Request) error {  filepath := request.URL.Path[len("/list/"):]  file, err := os.Open(filepath)  **if** err != nil {  **return** err  }  **defer** file.Close()   all, err := ioutil.ReadAll(file)  **if** err != nil {  **return** err  }   writer.Write(all)  **return** nil } |

* 进阶示例（添加了panic、UserError的处理）

|  |
| --- |
| //定义用户可知道错误的接口 **type** UserError **interface** {  error  Message() string } //定义函数类型 **type** handleFunction **func**(writer http.ResponseWriter, request \*http.Request) error  //利用函数式编程，把处理函数包装一下，使error汇总到此函数中集中处理 **func** handleError(handler handleFunction) **func**(writer http.ResponseWriter, request \*http.Request) {  **return func**(writer http.ResponseWriter, request \*http.Request) {  err := handler(writer, request)  //如果是panic，在此处recover  **defer func**() {  r := recover()  **if** r != nil {  log.Warn("panic : %v", r)  http.Error(writer, http.StatusText(http.*StatusInternalServerError*), http.*StatusInternalServerError*)  }  }()  //如果是error，在此处处理  **if** err != nil {  log.Warn("Error occurred handling request: %s", err.Error())  //判断此错误是否为用户可知错误，如果是直接返回结果  **if** userErr, ok := err.(UserError); ok {  http.Error(writer, userErr.Message(), http.*StatusBadRequest*)  **return**  }  //如果非用户可知道的错误，在此处处理  code := http.*StatusOK* **switch** {  **case** os.IsNotExist(err):  code = http.*StatusNotFound* **case** os.IsPermission(err):  code = http.*StatusForbidden* **default**:  code = http.*StatusInternalServerError* }  http.Error(writer, http.StatusText(code), code)  }  } } **func** main() {  http.HandleFunc("/", handleError(filelisting.HandleFileList))  err := http.ListenAndServe(":8888", nil)  **if** err != nil {  panic(err)  } } |
| //定义一个用户错误，此错误的出错信息可以反馈到页面上被用户查看 **type** UserError string **func** (u UserError) Error() string {  **return** u.Message() } **func** (u UserError) Message() string {  **return** string(u) } **const** *prefix* = "/list/" //如果出错，把错误抛出去，集中处理 **func** HandleFileList(writer http.ResponseWriter, request \*http.Request) error {  //验证url是否以prefix开头  **if** strings.Index(request.URL.Path, *prefix*) != 0 {  **return** UserError("path must start with " + *prefix*)  }  filepath := request.URL.Path[len(*prefix*):]  file, err := os.Open(filepath)  **if** err != nil {  **return** err  }  all, err := ioutil.ReadAll(file)  **if** err != nil {  **return** err  }  **defer** file.Close()  writer.Write(all)  **return** nil } |

# 测试

## 基础测试、代码覆盖率、性能

* 测试

$ go test . //运行测试文件

* 代码覆盖率

$ go test -coverprofile=c.out

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

$ go tool cover -html=c.out

|  |
| --- |
| **func** TestLengthOfNonRepeatingSubStr(t \*testing.T) {  tt := []**struct** {  str string  count int  }{  {"abcdebfgh", 7},  {"我是中国人", 5},  }  **for** \_, c := **range** tt {  **if** n := lengthOfNonRepeatingSubStr(c.str); n != c.count {  t.Errorf("{%s , %d} : %d", c.str, c.count, n)  }  } } |

* 性能

$ go test -bench .

|  |
| --- |
| **func** BenchmarkLengthOfNonRepeatingSubStr(b \*testing.B) {  c := **struct** {  str string  count int  }{"我是中国人", 5}  **for** i := 0; i < b.N; i++ {  **if** n := lengthOfNonRepeatingSubStr(c.str); n != c.count {  b.Errorf("{%s , %d} : %d", c.str, c.count, n)  }  } } |

* pprof

性能测试资料：<https://studygolang.com/articles/1443>

查看性能图：

$ go test -bench . -cpuprofile cpu.out //在Test文件目录中使用此命令

$ go tool pprof cpu.out

(pprof) web //需要安装graphviz [配置环境变量]

|  |
| --- |
| **func** BenchmarkLengthOfNonRepeatingSubStr(b \*testing.B) {  s := "我是中国人"  **for** i := 0; i < 13; i++ {  s = s + s  }  b.Logf("len(s) = %d", len(s))  c := **struct** {  str string  count int  }{s, 5}  //重置时间，除去前边测试内容准备时间  b.ResetTimer()  **for** i := 0; i < b.N; i++ {  **if** n := lengthOfNonRepeatingSubStr(c.str); n != c.count {  b.Errorf("{%s , %d} : %d", c.str, c.count, n)  }  } } |

## Http测试

|  |
| --- |
| //定义一个用户错误,用于测试 **type** testUserError string **func** (u testUserError) Error() string {  **return** u.Message() } **func** (u testUserError) Message() string {  **return** string(u) }  //create error function **func** appHandle(writer http.ResponseWriter, request \*http.Request) error {  **return** errors.New("new error") } //create error function **func** userErrorHandle(writer http.ResponseWriter, request \*http.Request) error {  **return** testUserError("user error") } **func** notFindError(writer http.ResponseWriter, request \*http.Request) error {  **return** os.ErrNotExist } **func** noError(writer http.ResponseWriter, request \*http.Request) error {  fmt.Fprintln(writer, "no error")  **return** nil } //定义测试用例 **var** tests = []**struct** {  f HandleFunction  code int  message string }{  {appHandle, 500, "Internal Server Error"},  {userErrorHandle, 400, "user error"},  {notFindError, 404, "Not Found"},  {noError, 200, "no error"}, } |
| //模拟访问服务测试，假的request和response；只是测试了handleError函数,更像一个单元测试 **func** TestHandleError(t \*testing.T) {  **for** \_, tt := **range** tests {  h := handleError(tt.f)  request := httptest.NewRequest(http.*MethodGet*, "http://www.localhost:8888", nil)  response := httptest.NewRecorder()  h(response, request)  verifyResponse(response.Result(), tt.code, tt.message, t)  } } |
| //启动一个测试服务器来测试；测试整个服务器 **func** TestHandleErrorInServe(t \*testing.T) {  **for** \_, tt := **range** tests {  h := handleError(tt.f)  server := httptest.NewServer(http.HandlerFunc(h))  resp, \_ := http.Get(server.URL)  verifyResponse(resp, tt.code, tt.message, t)  } } |
| // **func** verifyResponse(resp \*http.Response, expectedCode int, expectedMsg string, t \*testing.T) {  all, \_ := ioutil.ReadAll(resp.Body)  body := strings.Trim(string(all), "\n")  **if** resp.StatusCode != expectedCode || body != expectedMsg {  t.Errorf("Expect (%d, %s) ; Got (%d, %s)", expectedCode, expectedMsg, resp.StatusCode, body)  } } |

import \_ “net/http/pprof” → 在服务器中导入这个包后启动，服务器具有Debug的能力

URL/Debug/pprof → 在客户端访问此路径，可以得到服务器的调试数据

使用go tool pprof分析性能

|  |  |
| --- | --- |
| 查看内存使用情况 | go tool pprof http://localhost:8000/debug/pprof/heap |
|  | web //打开内存使用分析图 |

## go文档的编写

* 用注释写文档
* 在测试中加入Example，在文档中会添加进去
* 使用go doc → 查看/生成文档

$ go doc

$ go doc Queue

$ go doc IsEmpty

$ godoc -http :6060

|  |
| --- |
| // pushes the element into the queue // e.g. q.Push(123) **func** (q \*Queue) Push(v **interface**{}) {  \*q = append(\*q, v) } |
| 上表示例的注释代码，会在文档中体现出来；前边添加多个空格的注释，在文档中会自动添加底色 |

* 示例代码

|  |
| --- |
| 写到queue\_test.go文件中的代码，也有测试的功能  写入test文件中的Example代码，会体现在文档中 |
| **func** ExampleQueue\_Pop() {  q := Queue{1}  q.Push(2)  q.Push(3)  fmt.Println(q.Pop())  fmt.Println(q.Pop())  fmt.Println(q.IsEmpty())   // Output:  // 1  // 2  // false } |

# 并发编程

* 主协程结束，其他协程也会结束

## 多协程goroutine

|  |
| --- |
| **func** task() {  **for** {          fmt.Println("this is task")          time.Sleep(time.Second)      }  } |
| **func** main() {      //需要关键字 go 开启一个新的协程  **go** task()  **for** {          fmt.Println("this is main gorontine")          time.Sleep(time.Second)      }  } |

|  |
| --- |
| **func** printInfo(str string) {  **for** \_, data := **range** str {          fmt.Printf("%c", data)          time.Sleep(time.Second)      }      fmt.Println()  }  **func** main() {      //同步问题  **go** printInfo("hello")  **go** printInfo("world")  **for** {      }  } |

## runtime协程操作

|  |
| --- |
| **func** main1() {      //此函数设置运行的cpu数量，数量越多，时间片段切换的越快      n := runtime.GOMAXPROCS(8)      fmt.Println(n)  **go** **func**() {  **for** {              fmt.Print(1)          }      }()  **for** {          fmt.Print(0)      }  } |
| **func** test() {      fmt.Println("ddddddddddddddddddd")      //退出当前协程      runtime.Goexit()  } |
| **func** main() {  **go** **func**() {          fmt.Println("bbbbbbbbbbbbbbbbbb")          test()          fmt.Println("cccccccccccccccccc")      }()      //让出当前协程时间片段      runtime.Gosched()      fmt.Println("aaaaaaaaaaaaaa")  **for** {      }  } |

**goroutine的定义：**

* 任何函数只需加上go就能送给调度器运行
* 不需要在定义时区分是否是异步函数
* 调度器在合适的点进行切换
* 使用 -race 来检测数据访问的冲突

**goroutine可能的切换点**：

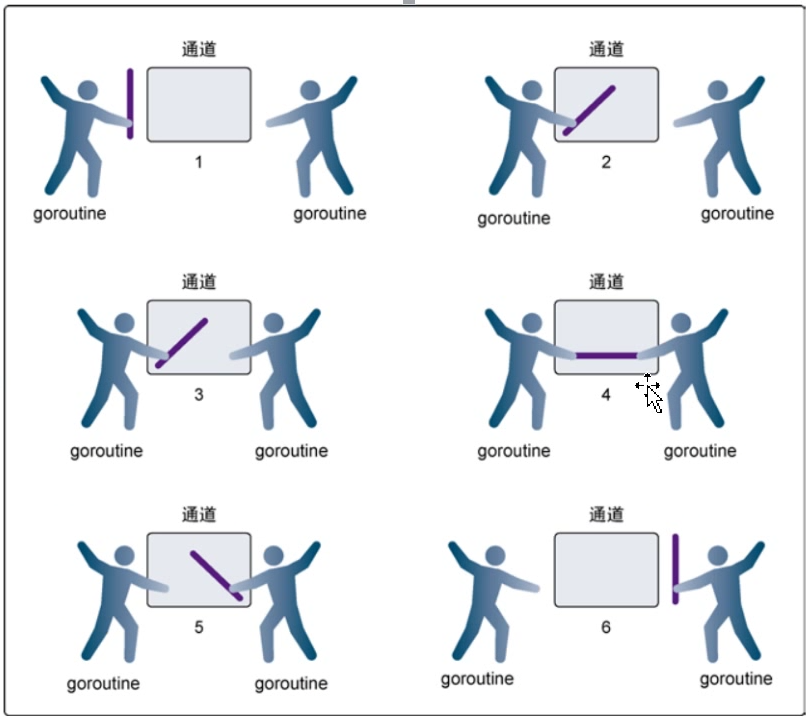
I/O, select 函数调用（有时） channel

runtime.Gosched() 等待锁

**只是参考，不能保证切换，不能保证在气头地方不切换**

# channel

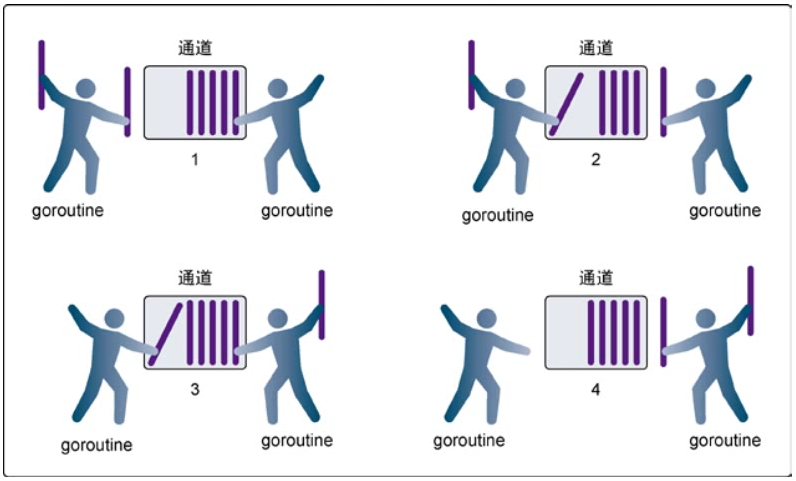
## 无缓存的channel



* 无缓存的channel，在向里边放数据和取数据的整个过程没完成的时候都是处于阻塞状态
* 通道中的信息传递是同步的

|  |
| --- |
| **func** main() {      ch := make(**chan** int)      fmt.Printf("len(ch)=%d, cap(ch)=%d\n", len(ch), cap(ch))  **go** **func**() {  **for** i := 0; i < 3; i++ {              fmt.Println("i = ", i)              ch <- i              //由于无缓存区，所以len(ch)和cap(ch)的值都是0              //fmt.Printf("len(ch)=%d, cap(ch)=%d\n", len(ch), cap(ch))          }      }()      time.Sleep(2 \* time.Second)  **for** i := 0; i < 3; i++ {          num := <-ch          fmt.Println("num = ", num)      }  } |

## 有缓存的channel



* 当channel的缓存区满的时候，不能向里边存东西 -> 阻塞
* 当channel的缓存区空的时候，不能从中取数据 -> 阻塞
* 通道中的信息传递是异步的

|  |
| --- |
| **func** main() {      ch := make(**chan** int, 3)      fmt.Printf("len(ch)=%d , cap(ch)=%d\n", len(ch), cap(ch))  **go** **func**() {  **for** i := 0; i < 10; i++ {              fmt.Printf("i = %d ==>", i)              ch <- i              fmt.Printf("go: len(ch)=%d , cap(ch)=%d\n", len(ch), cap(ch))          }      }()      time.Sleep(2 \* time.Second)  **for** i := 0; i < 10; i++ {          n := <-ch          fmt.Printf("n = %d ==>", n)          fmt.Printf("main: len(ch)=%d , cap(ch)=%d\n", len(ch), cap(ch))      }  } |

## 关闭channel

* channel不像文件一样需要经常关闭，只有当你确实没有任何数据发送了，或者想显式的接收range循环之类的，才去关闭channel；
* 关闭channel后，无法向channel再发送数据（引发panic错误后导致接收立即返回零值）
* 关闭channel后，可以继续向channel接收数据
* 对于nil channel，无论收发都会被阻塞

|  |
| --- |
| **func** main() {      ch := make(**chan** int)  **go** **func**() {  **for** i := 0; i < 5; i++ {              ch <- i          }  //不需要再写数据时关闭channel          close(ch)      }()  **for** {  //ok为true时，未关闭channel  **if** num, ok := <-ch; ok {              fmt.Println("num = ", num)          } **else** {  **break**          }      }  } |
| **func** main() {      ch := make(**chan** int)  **go** **func**() {  **for** i := 0; i < 5; i++ {              ch <- i          }          //不需要再写数据时关闭channel          close(ch)      }()      //使用range遍历ch，当关闭时，跳出循环  **for** num := **range** ch {          fmt.Println("num = ", num)      }  } |

## 单向channel

* chan<- int //只能写，不能读
* <-chan int //只能读，不能写
* chan int //双向，可读可写
* 双向可以转换为单向，单向不能转换为双向

|  |
| --- |
| //生产者  **func** producter(write **chan**<- int) {  **for** i := 0; i < 10; i++ {          write <- i \* i      }      close(write)  } |
| //消费者  **func** consumer(read <-**chan** int) {  **for** num := **range** read {          fmt.Println("num = ", num)      }  } |
| /\*生产者消费者模式\*/  **func** main() {      ch := make(**chan** int)      //生产者生产数字  **go** producter(ch)      //消费者打印数字      consumer(ch)  } |
| channel传递参数，是引用传递 |

## 定时器

* timer：时间到了，只执行一次调用，即向通道中写一次时间
* ticker：会循环调用，向通道中循环写入时间

### timer和ticker

|  |
| --- |
| //验证timer只有一次写入  **func** main4() {      timer := time.NewTimer(3 \* time.Second)      timer.Reset(time.Second)      i := 0  **for** {          i++          //由于timer不能重复写入，一次调用后死锁          <-timer.C          fmt.Println("i = ", i)  **if** i == 5 {  **break**          }      }  } |
| //重置时间  **func** main3() {      timer := time.NewTimer(3 \* time.Second)      timer.Reset(time.Second)      //重置时间后，会按照重置的时间调用      <-timer.C      fmt.Println("end")  } |
| //timer停止  **func** main2() {      timer := time.NewTimer(3 \* time.Second)      timer.Stop()      //关闭之后，此处就是死锁了      <-timer.C      fmt.Println("end")  } |
| //创建timer  **func** main1() {      //创建一个计时器，3s后向管道中写入当前的时间      timer := time.NewTimer(3 \* time.Second)      fmt.Println(time.Now())      //timer.C，取出timer中写入的时间      time := <-timer.C      fmt.Println("time: ", time)  } |
| ticker: |
| **func** main() {      ticker := time.NewTicker(time.Second)      i := 0  **for** {          i++          //ticker 可以重复写入          <-ticker.C          fmt.Println("i = ", i)  **if** i == 5 {  **break**          }      }  } |

### 延时功能

|  |
| --- |
| **func** test01() {      //通过sleep实现延时加载  **for** i := 0; i < 5; i++ {          time.Sleep(time.Second)          fmt.Println("i=", i)      }  } |
| **func** test02() {      timer := time.NewTimer(time.Second)      //通过timer实现延时加载      <-timer.C      fmt.Println("end")  } |
| **func** test03() {      //通过after实现延时加载      ti := time.After(time.Second)      t := <-ti      fmt.Println("t: ", t)  } |

## sync.WaitGroup:等待任务执行完成

系统提供的等待多任务执行，协作的类

wg.add(int i) //有多少个协程被添加到到了其中

wg.done() //在worker的协程中

wg.wait() //等待完成的协程中

## select：调度channel

* 监听管道中的数据流动
* 相当于switch，但是case的判断只能是io操作
* 如果所有的case中的条件都满足，会任意执行一个case
* **斐波那契数列**

|  |
| --- |
| **func** fbnc(ch **chan**<- int, flag <-**chan** bool) {      x, y := 1, 1  **for** {  **select** {  **case** ch <- x:              x, y = y, x+y  **case** f := <-flag:              fmt.Println(f)  **return**          }      }  }  **func** main() {      //定义两个channel      ch := make(**chan** int)      flag := make(**chan** bool)      //消费者打印数列  **go** **func**() {  **for** i := 0; i < 8; i++ {              num := <-ch              fmt.Println(num)          }          flag <- true      }()      //生产者向channel中写数据      fbnc(ch, flag)  } |

* **实现超时机制**

|  |
| --- |
| **func** main() {      ch := make(**chan** int)      flag := make(**chan** bool)  **go** **func**() {          //检测管道中数据的流动  **select** {  **case** c := <-ch:              fmt.Println(c)  **case** <-time.After(3 \* time.Second):              fmt.Println("程序超时了")              flag <- true          }      }()      <-flag      fmt.Println("main is end!!!")  } |

## 传统的同步机制

* WaitGroup
* Mutex
* Cond

包：sync中的Lock

# reflect反射

|  |
| --- |
| **package** main **import** (  "fmt"  "reflect" ) **type** User **struct** {  Id int  Name string  Age int } **func** (u User) SeyHello(name string) {  fmt.Printf("%s sey hello %s", u.Name, name) } **type** manage **struct** {  User  title string } **func** (m manage) seyHi() {  fmt.Println("sey hi") }  **func** main1() {  //getPublicFieldsAndMethods(User{1, "ok", 12})  //getPrivateFieldAndMethods(manage{User{1, "ok", 12}, "111"})  //updateBasicValue()   //user := User{1, "ok", 12}  //updateStructValue(&user)  //fmt.Println(user)   //此处可以传入object也可以是point  callFunction(&User{1, "ok", 12}) }  **func** callFunction(i **interface**{}) {  v := reflect.ValueOf(i)  m := v.MethodByName("SeyHello")  args := []reflect.Value{reflect.ValueOf("joe")}  m.Call(args) }  **func** updateStructValue(i **interface**{}) {  v := reflect.ValueOf(i)  **if** v.Kind() == reflect.*Ptr* && !v.Elem().CanSet() {  fmt.Println("params is no point")  **return** } **else** {  v = v.Elem()  }  f := v.FieldByName("Name")  **if** !f.IsValid() {  fmt.Println("this field is not exist")  **return** }  **if** f.Kind() == reflect.*String* {  f.SetString("byebye")  } }  **func** updateBasicValue() {  x := 123  v := reflect.ValueOf(&x)  vv := v.Elem() //返回的是值  fmt.Println("vv:", vv)  vv.SetInt(999) //重新设置值  fmt.Println(x) }  **func** getPrivateFieldAndMethods(i **interface**{}) {  t := reflect.TypeOf(i)  //按照序号可取出匿名的字段和private的字段  f := t.Field(1)  fmt.Printf("%#v\n", f)  //可以取得public的method，取不到private的method  m := t.Method(0)  fmt.Printf("%#v\n", m) }  **func** getPublicFieldsAndMethods(i **interface**{}) {  t := reflect.TypeOf(i)  fmt.Println("type: ", t.Name())  //判断传入的类型是否为结构体，如果传入指针则退出  **if** k := t.Kind(); k != reflect.*Struct* {  fmt.Println("in type of ", k)  **return** }  v := reflect.ValueOf(i)  //获取属性的值及类型  fmt.Println("Fileds:")  **for** i := 0; i < t.NumField(); i++ {  f := t.Field(i)  val := v.Field(i)  fmt.Printf("%6s: %v = %v\n", f.Name, f.Type, val)  }  //获取方法的值和类型  fmt.Println("Methods:")  **for** i := 0; i < t.NumMethod(); i++ {  m := t.Method(i)  fmt.Printf("%6s: %v\n", m.Name, m.Type)  } } |

# 网络编程

## tcp

|  |
| --- |
| serve: |
| **func** main() {      //监听指定端口      listen, err := net.Listen("tcp", ":8000")  **if** err != nil {          fmt.Println("err = ", err)  **return**      }  **defer** listen.Close()      //阻塞等待连接      conn, err := listen.Accept()  **if** err != nil {          fmt.Println("err = ", err)  **return**      }  **defer** conn.Close()      //接收信息      buf := make([]byte, 1024)      n, err := conn.Read(buf)  **if** err != nil {          fmt.Println("err = ", err)  **return**      }      fmt.Println("client:", string(buf[:n]))  } |
| client: |
| **func** main() {      //客户端连接服务器      conn, err := net.Dial("tcp", "127.0.0.1:8000")  **if** err != nil {          fmt.Println("err = ", err)  **return**      }  **defer** conn.Close()      \_, err1 := conn.Write([]byte("are u ok?"))  **if** err1 != nil {          fmt.Println("err1 = ", err1)  **return**      }  } |

## 并发访问serve和client

|  |
| --- |
| serve |
| **func** clientAccept(conn net.Conn) {  **defer** conn.Close()      //clientIP      addr := conn.RemoteAddr().String()      fmt.Printf("%s connection\n", addr)      buf := make([]byte, 2048)  **for** {          n, err := conn.Read(buf)  **if** err != nil {              fmt.Println("conn err: ", err)  **return**          }          str := string(buf[:n])  **if** "exit" == strings.Trim(str, "\n") || "exit" == strings.Trim(str, "\r\n") {              fmt.Printf("%s exit", addr)  **return**          }          fmt.Printf("[%s]: %s", addr, str)          conn.Write([]byte(strings.ToUpper(str)))      }  }  **func** main() {      //监听端口      listen, err := net.Listen("tcp", ":8888")  **if** err != nil {          fmt.Println("listen err: ", err)  **return**      }  **defer** listen.Close()  **for** {          //连接端口          conn, err := listen.Accept()  **if** err != nil {              fmt.Println("listen err: ", err)  **continue**          }          //处理客户信息  **go** clientAccept(conn)      }  } |
| client |
| **func** main() {      conn, err := net.Dial("tcp", "127.0.0.1:8888")  **if** err != nil {          fmt.Println("conn err: ", err)  **return**      }  **defer** conn.Close()  **go** **func**() {          //向服务器发送数据          buf := make([]byte, 1024)  **for** {              n, err := os.Stdin.Read(buf)  **if** err != nil {                  fmt.Println("os read err :", err)  **return**              }              \_, err1 := conn.Write(buf[:n])  **if** err1 != nil {                  fmt.Println("conn write err1 :", err1)  **return**              }          }      }()      //接收服务器数据  **for** {          buf := make([]byte, 1024)          n, err := conn.Read(buf)  **if** err != nil {              fmt.Println("os read err :", err)  **return**          }          fmt.Printf("receive:%s\n", string(buf[:n]))      }  } |

## http服务器/客戶端

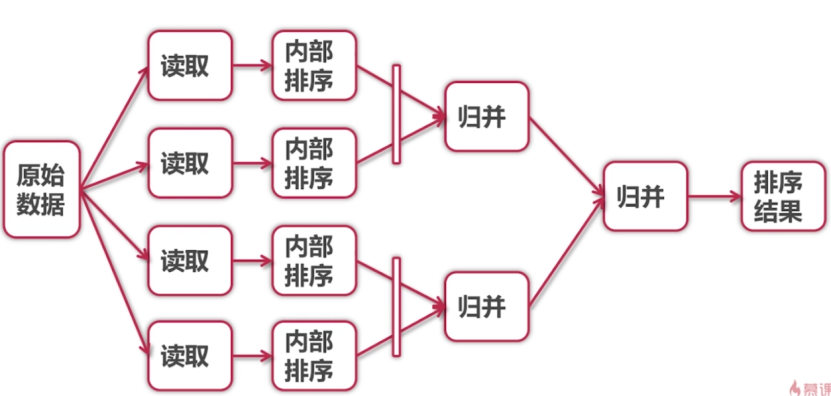
|  |
| --- |
| http serve |
| **import** (      "fmt"      "net/http"  )  //第一个参数为响应，第二个参数包含请求的参数  **func** myHandle(w http.ResponseWriter, r \*http.Request) {      w.Write([]byte("hello go"))  }  **func** main() {      //第一个参数为请求的资源名，第二个参数为处理请求的函数      http.HandleFunc("/go", myHandle)      //第一个参数为监听的ip及端口，第二个参数一般写nil，默认调用处理函数      err := http.ListenAndServe(":8000", nil)  **if** err != nil {          fmt.Println("http listen err:", err)  **return**      }  } |
| http client |
| **import** (      "fmt"      "net/http"  )  **func** main() {      resp, err := http.Get("http://www.baidu.com")  **if** err != nil {          fmt.Println("get err:", err)  **return**      }  **defer** resp.Body.Close()      fmt.Println(resp.Status)      buf := make([]byte, 1024\*4)  **var** temp string  **for** {          n, err := resp.Body.Read(buf)  **if** n == 0 {              fmt.Println("resp err:", err)  **return**          }          temp += string(buf[:n])      }      fmt.Println(temp)  } |
| **对http-server的底层的探索：** |
| //自己编写处理函数  **func** main() {  mux := http.NewServeMux()  mux.Handle("/", &myHandel{})  mux.HandleFunc("/hello",seyHello)  err := http.ListenAndServe(":8888",mux)  **if** err != nil {  log.Fatal(err)  } } **type** myHandel **struct** {} **func** (\*myHandel) ServeHTTP(w http.ResponseWriter,r \*http.Request){  io.WriteString(w,"url:"+r.URL.String()) } **func** seyHello(w http.ResponseWriter,r \*http.Request){  io.WriteString(w,"hello world this is session 2") } |
| //自己编写处理函数  **type** handlerFunc **func**(http.ResponseWriter, \*http.Request) **var** mux **map**[string]handlerFunc **func** main() {  server := http.Server{  Addr: ":8888",  Handler: &myHandel{},  ReadTimeout: 5 \* time.*Second*,  }   //注册路由  mux = make(**map**[string]handlerFunc)  mux["/hello"] = seyHello  mux["/bye"] = seyBye  err := server.ListenAndServe()  **if** err != nil {  log.Fatal(err)  } } **type** myHandel **struct**{} **func** (\*myHandel) ServeHTTP(w http.ResponseWriter, r \*http.Request) {  **if** h, ok := mux[r.URL.String()]; ok {  h(w, r)  **return** }  io.WriteString(w, "url:"+r.URL.String()) }  **func** seyHello(w http.ResponseWriter, r \*http.Request) {  io.WriteString(w, "hello world this is version 3") }  **func** seyBye(w http.ResponseWriter, r \*http.Request) {  io.WriteString(w, "Bey bey this is version 3") } |
| //对静态文件的访问  **func** main() {  mux := http.NewServeMux()  mux.Handle("/", &myHandel{})  mux.HandleFunc("/hello", seyHello)  //获取相对路径  dir, err := os.Getwd()  **if** err != nil {  log.Fatal(err)  }  //fmt.Println(dir)  //定义静态文件访问  mux.Handle("/static/",  http.StripPrefix("/static/",  http.FileServer(http.Dir(dir))))  err = http.ListenAndServe(":8080", mux)  **if** err != nil {  log.Fatal(err)  } } **type** myHandel **struct**{} **func** (\*myHandel) ServeHTTP(w http.ResponseWriter, r \*http.Request) {  io.WriteString(w, "url:"+r.URL.String()) } **func** seyHello(w http.ResponseWriter, r \*http.Request) {  io.WriteString(w, "hello world this is session 2") } |

# 例

## 广度优先算法走迷宫

|  |
| --- |
| **package** main  **import** (      "fmt"      "os"  )  **func** createMaze(fileName string) [][]int {      file, err := os.Open(fileName)  **if** err != nil {          fmt.Println("os.open err:", err)      }  **defer** file.Close()  **var** row, col int      fmt.Fscanf(file, "%d %d", &row, &col)      maze := make([][]int, row)  **for** i := **range** maze {          maze[i] = make([]int, col)          fmt.Fscanf(file, "%d", &row)  **for** j := **range** maze[i] {              fmt.Fscanf(file, "%d", &maze[i][j])          }      }  **return** maze  }  **type** point **struct** {      i, j int  }  **var** dirs []point = []point{{-1, 0}, {0, -1}, {1, 0}, {0, 1}}  **func** (p point) add(r point) point {  **return** point{p.i + r.i, p.j + r.j}  }  **func** (p point) at(result [][]int) (int, bool) {  **if** p.i < 0 || p.i >= len(result) {  **return** 0, false      }  **if** p.j < 0 || p.j >= len(result[0]) {  **return** 0, false      }  **return** result[p.i][p.j], true  }  **func** game(maze [][]int, start, end point) [][]int {      result := make([][]int, len(maze))  **for** i := **range** maze {          result[i] = make([]int, len(maze[i]))      }      Q := []point{start}  **for** len(Q) > 0 {          cur := Q[0]          Q = Q[1:]  **for** \_, dir := **range** dirs {              next := cur.add(dir)              v, ok := next.at(maze)  **if** !ok || v != 0 {  **continue**              }  **if** next == start {  **continue**              }              v, ok = next.at(result)  **if** !ok || v != 0 {  **continue**              }              Q = append(Q, next)              c, \_ := cur.at(result)              result[next.i][next.j] = c + 1          }      }  **return** result  }  **func** main() {      maze := createMaze("maze.txt")      printInfo(game(maze, point{0, 0}, point{len(maze) - 1, len(maze[0]) - 1}))  }  **func** printInfo(info [][]int) {  **for** \_, row := **range** info {  **for** \_, val := **range** row {              fmt.Printf("%3d", val)          }          fmt.Println()      }  } |
| maze.txt文件  6 5  0 1 0 0 0  0 0 0 1 0  0 1 0 1 0  1 1 1 0 0  0 1 0 0 1  0 1 0 0 0 |

## 并行管道计算



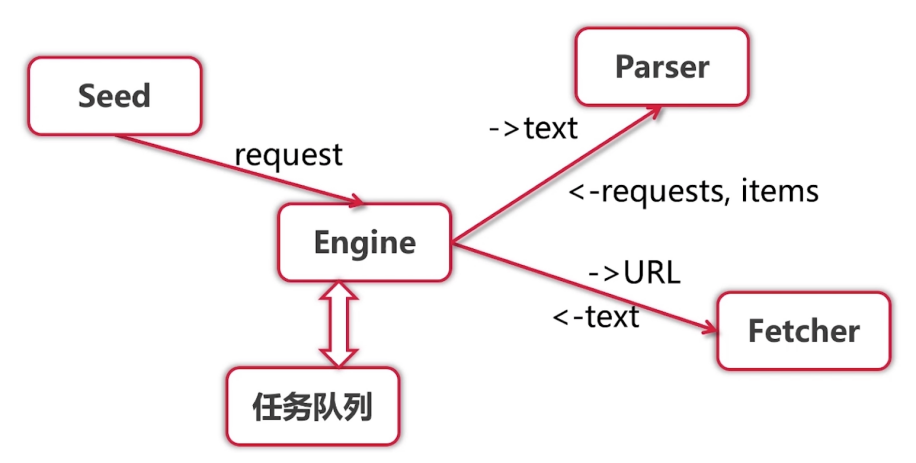
## 分布式爬虫

|  |  |
| --- | --- |
| go语言的爬虫库/框架 |  |
| henrylee2cn/pholcus | gocrawl |
| colly | hu17889/go\_spider |

### 爬虫项目架构

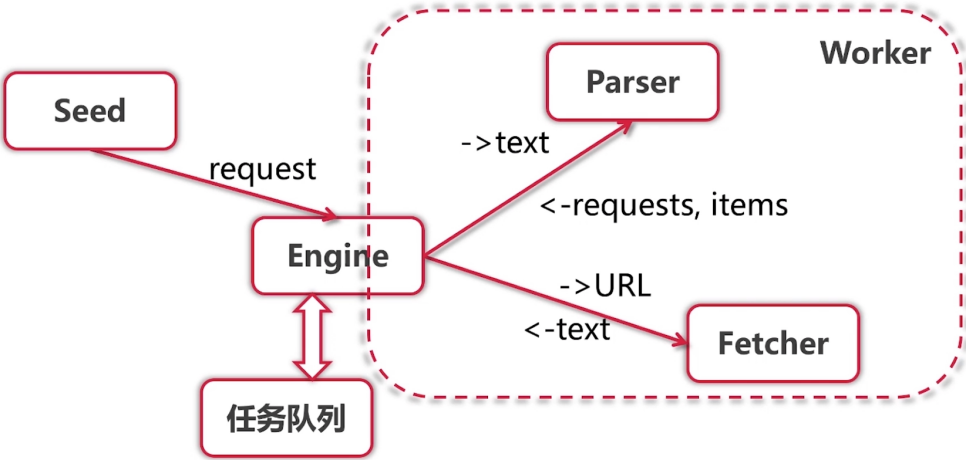
1. 不使用现成爬虫库/框架
2. 使用ElasticSearch作为数据储存
3. 使用Go语言标准模板库实现http数据展示部分

* **单机爬虫的架构**

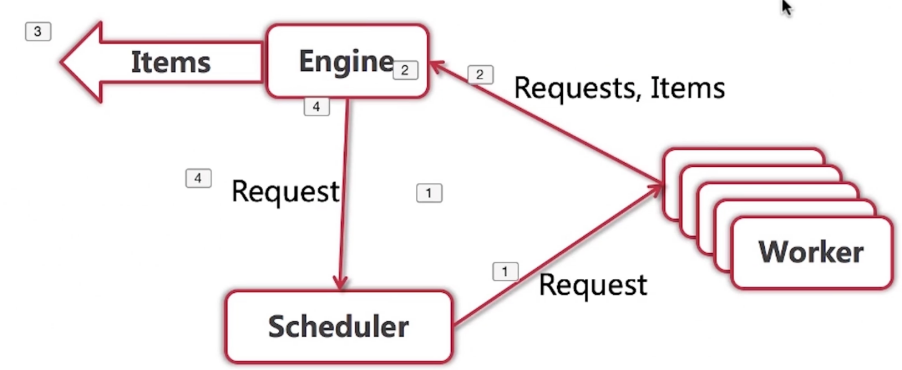


此处的箭头：代表函数的参数或返回值

* **并发版爬虫**



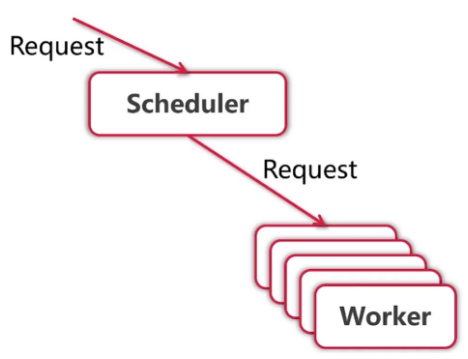
↓



并发版的箭头：代表channel

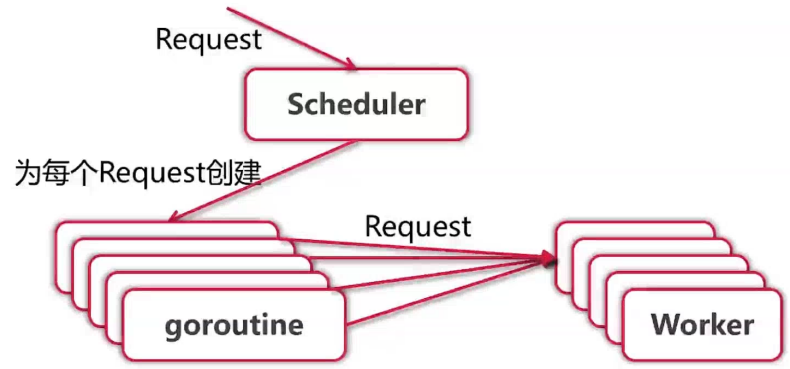
**Scheduler实现I：所有Worker公用一个输入：**

**问题：会出现循环等待的问题**



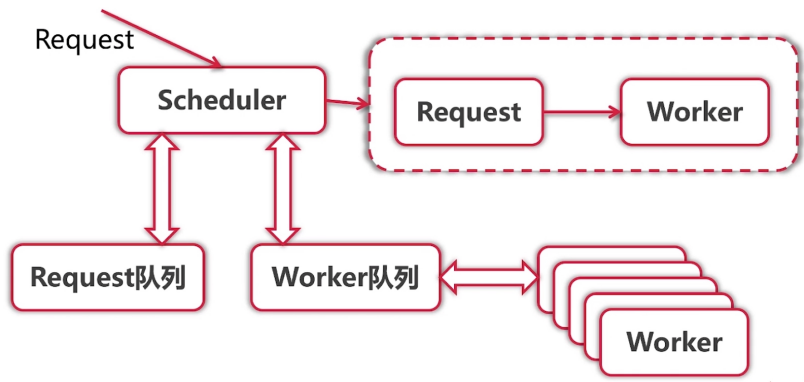
**Scheduler实现II：并发分发Request**

**问题：对创建的Request协程和Worker协程，不能更好的控制**

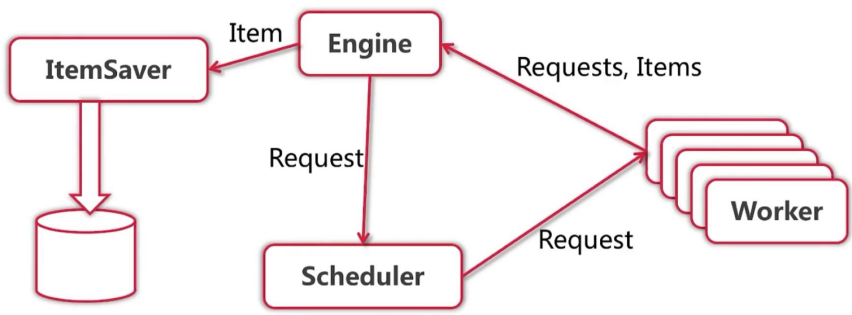


**Scheduler实现III：Request队列和Worker队列**

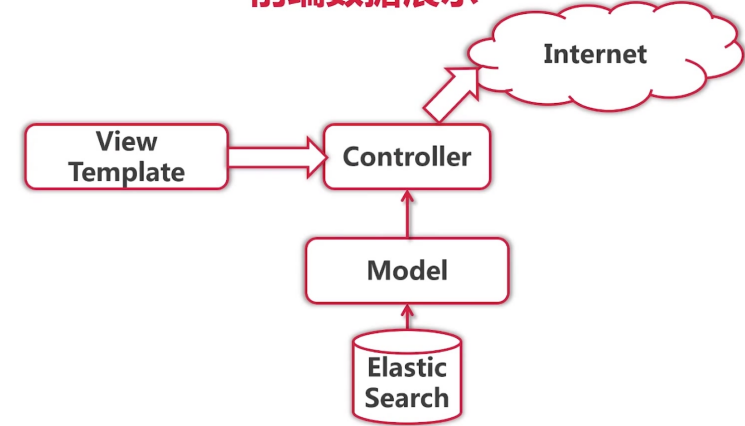
**可以选择接收Request的Worker，有更好的控制性**



**并发版爬虫架构**



* **前端数据展示**：



**html/template:**

* 模板引擎
* 服务器终端生成最终网页
* 适合做后台或者维护页面

**需要掌握：**

取值 循环 选择 函数调用

### 爬虫的实现

* **获取网页**

**转换读取数据的编码方式：**

golang.org/x/text包的获得：

使用gopm直接获得：

$ gopm get -g -v golang.org/x/text

单独下载包：

1. 在GOPATH/src/中建立golang.org/x目录
2. 下载包放进文件夹 > git clone <https://github.com/golang/text.git> --depth 1

**读取网页数据猜网页的编码格式：**

1. $ gopm get -g -v golang.org/x/net/html
2. > git clone https://github.com/golang/net --depth 1

* **过滤网页中的数据**

1. css选择器；golang官方包不支持css选择器，可以找第三方的库导入
2. 使用xpath
3. **使用正则表达式**

## 流量统计系统

视频：https://www.bilibili.com/video/av26613496/?p=3



## 千万级消息推送系统

视频：https://www.imooc.com/learn/1025

* 问题：
* 内核瓶颈：

推送**量大**：100万在线\*10条/秒 = 1000万条/秒

内核瓶颈：linux内核发送TCP的 **极限包频** ≈ 100万/秒

* 锁瓶颈：

需要维护在线用户集合（100万在线），通常是一个字典结构

推送消息即**遍历整个集合**，顺序发送消息，耗时极长

推送期间，客户端仍旧正常上/下线，所以**集合需要上锁**

* CPU瓶颈：

浏览器与服务端通常采取json格式通讯

json**编码**非常**耗费CPU**资源

向100万在线推送1次，需要100万次json encode

* 解决：
* 内核瓶颈：

-优化原理

减少网络小包的发送

-优化方案

将同一秒内的N条消息，合并成1条消息

合并后每秒**推送次数**只等于**在线连接数**

* 锁瓶颈：

-优化原理

大拆小

-优化方案

连接**打散**到多个集合中，每个集合都有自己的锁

**多线程并发**推送多个集合，避免锁竞争

**读写锁**取代互斥锁，多个推送任务可以并发遍历相同集合

* CPU瓶颈：

-优化原理

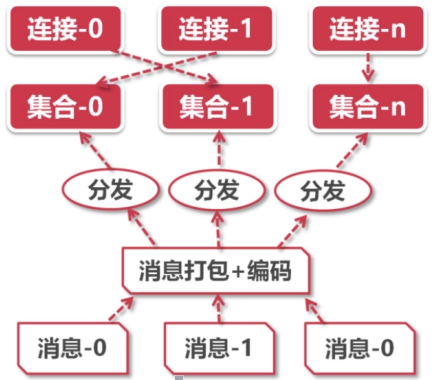
减少重复计算

-优化方案

json编码前置，**1次**消息编码 + **100万次**推送

消息合并前置，N条消息合并后只编码1次

* 单机架构



单机瓶颈：

* 维护海量长连接会花费**不少内存**
* 消息推送瞬时消耗**大量CPU**资源
* 消息推送瞬时带宽高达400~600MB（4~6Gbits），是**主要瓶颈**！
* 集群架构：
* 网关集群：



* 逻辑集群：

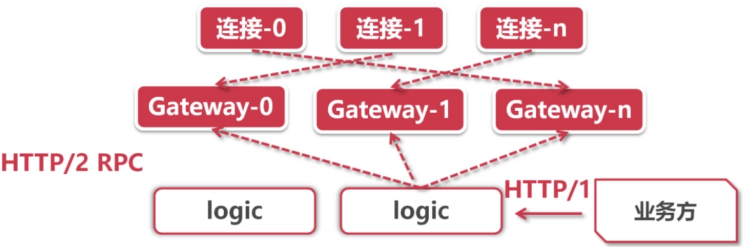
基于HTTP/2协议向gateway集群分发消息

**HTTP/2支持连接复用，用作RPC性能更佳**

基于HTTP/1协议对外提供推送API

**HTTP/1更加普及，对业务方更加友好**

* 整体架构：



* 其他编程知识
* 技术

生产工具，就像斧子、铲子一样

开发语言、框架都是技术，**努力学习大多可以掌握**

* 思想

来自于**常年累月**对技术的积累和理解

合理设计、架构经验、来自于不断的技术**实践与总结**

* 精神

不在局限于特定的技术体系与思想体系

凭直觉，能够快速抓住**问题本质**，在众多**纷扰中做出正确选择**

## 日志监控系统

* Influxdb简介：

Infuxdb是一个开源的时序型的数据库，使用Go语言编写，被广泛应用于储存系统的监控数据，IoT行业的实时数据等场景。

部署简单，无外部依赖；

内置http支持，使用http读写

类sql的灵活查询(max , min , sum 等)

* Infuxdb安装：

<https://docs.influxdata.com/influxdb/v0.9/introduction/installation/>

docker pull influxdb

* Influxdb关键概念：



写入：

curl -i XPOST ‘http://localhost:8086/write?db=mydb’ --data-binary

‘cpu\_usage,host=server01,region=us-west value=0.64 143405556200000000’

读取：

curl -G ‘http://localhost:8086/query?pretty=true’ --data-urlencode

“db=mydb” --data-urlencode “q=SELECT \”value\” FROM \”cpu\_usage\” WHERE \”region\”=’us-west’”

* Golang的客户端

使用原始的http不方便；

Golang客户端使用：

<https://github.com/influxdata/influxdb/tree/master/client>

# web框架

* Docker
* Kubernetes
* Caddy → nginx
* CockroachDB 数据库
* beego/gin/echo → web框架

beegoAPI: https://beego.me/quickstart

* react → 前端框架

## beego快速入门

安装或者升级 Beego 和 [Bee](http://beego.me/docs/install/bee.md) 的开发工具

|  |
| --- |
| $ go get -u github.com/astaxie/beego  $ go get -u github.com/beego/bee |

bee new hello → 创建一个项目

bee run hello → 启动项目

beego运行方式 → 深度优先

beego特点 → 热更新：修改代码后不用人为的重新编译重启服务器，服务器会自动重启

## orm的使用

安装orm

|  |
| --- |
| $ go get github.com/astaxie/beego/orm |

操作数据库-mysqlqudong

|  |
| --- |
| $ go get -u github.com/go-sql-driver/mysql |

|  |
| --- |
| **func** init() {  // set default database  orm.RegisterDataBase("default", "mysql",  "root:root@tcp(127.0.0.1:3306)/godemo?charset=utf8", 30)   // register model  orm.RegisterModel(new(Category), new(Topic))   // create table  orm.RunSyncdb("default", *false*, *true*) } |

# 附录1：golang的命令

Go常用命令简介：

go get：获取远程包（需 提前安装 git或hg）

go run：直接运行程序

go build：测试编译，检查是否有编译错误

当文件的名称为main.go的时候会产生可执行文件main.exe；

如果不是将不会生成任何文件；

go fmt：格式化源码（部分IDE在保存时自动调用）

go install：编译包文件并编译整个程序

go test：运行测试文件

go doc：查看文档（CHM手册）

godoc fmt.println → 可以查看单个的函数

godoc -http=:8080 → 可以在本地的8080端口生成一个go的api

go run -race xxx.go ：可以检测 xxx.go 中的数据访问冲突(报错)

# 附录2：idea项目上传到github上

* 创建 README.md 文件
* 创建 .gitignore 文件，在此文件中配置的项目不会推送到git仓库

\*.class

\*.war

.idea/

...

* 初始化项目为git版本控制

$ git init

Initialized empty Git repository in path //此提示信息表示初始化完成

* 查看文件变化

$ git status

* 添加变更文件，查看文件变化

$ git add .

$ git status

* 提交到本地仓库

$ git commit -am ‘first commit init project’

* 连接远程仓库

$ git remote add origin https://github.com/nyfblack/crawler\_one\_thread.git

* 查看分支

$ git branch

* 提交到远程仓库

$ git push -u origin master

运行以上命令后报错：第一次提交需要先把远程的拉取下来。执行以下命令把git上的项目拉取过来

$ git pull

$ git push -u origin master

执行以上命令后，继续报错，错误信息是本地仓库版本落后于远程仓库，因为这个是新项目，我们执行以下命令，强制推到远程仓库就可以了

$ git push -u -f origin master

刷新github对应的项目页可以看到文件已提交成功

* 创建版本

查看本地分支：$ git branch

查看远程分支：$ git branch -r

创建新的分支：$ git checkout -b v1.0 origin/master

把当前分支推送到远程：$ git push origin HEAD -u