Go语言

视频：**BV1fz4y1m7Pm**

BV1CX4y1G7FS

gin框架学习：BV1pz4y1D7Sx

安装配置

链接：**下载链接**

输入`go version`验证安装

环境变量中添加`GOPATH`，对应目录下添加三个文件夹`src`(源代码)，`pkg`，`bin`

编译运行文件

`go build [-path] [-o name]`：path为`GOPATH`的`src`目录下的项目路径，编译为exe文件，`-o`后面输出名字

`go run file.go` 运行go文件

`go install [go file]` 相当于先`go build`，再将exe文件移动到`GOPATH`的`bin`目录

`go fmt [gofile]` 格式化go代码

go也支持跨平台编译

初步startup

```go
package main
// package为main表示最终编译的文件

import "fmt"

// 程序入口函数
func main() {
    fmt.Println("Hello world")
}
```

用户输入输出

输出：

```go
fmt.Println("a")
s1 := "QAQAQQA"
fmt.Printf("type:%T, value:%v\n", s1, s1)
```

输入：

```go
var a1 int
var a2 int
n, err := fmt.Scanf("%d %d", &a1, &a2)
if err != nil {
      return
}
fmt.Println(n)
fmt.Println(a1, a2)
```

与bufio输入方式

```go
func c() {
    reader := bufio.NewReader(os.Stdin)
    s, err := reader.ReadString('\n')   // 这种方法可以识别输入中的空格
    if err != nil {
        return
    }
    fmt.Println(s)
}
```

Go Mod管理

切换国内环境：

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

变量与常量

`var 变量名 变量类型`

分为单独声明和批量声明，并且支持类型推导、简短声明（只能在函数里面使用）、匿名变量(`_`)

匿名变量相当于占位符，不分配内存

```
var nickname string //单独

var (   // 批量，由默认初始值
    name string  // ""
    age int // 0
    flag bool   // false
)

var a1, a2, a3 = 1, true, "good"

var pcName = "HP"   // 类型推导
n1 := 1     // 简短声明

x, _ = foo()

const PI = 3.1415926

const (
    n2 = 100
    n3  // 后面的值都是n2
    n4
)
```

iota关键字

初始定义值为0，域内常量**每添加一行**，iota自增1

```
const (
    a = iota    //  0
    b           //  1
    c = 100     // 100
    d = iota    // 3
    e = iota + 2    // 6
)
```

基本数据类型

整型：int8, int16, int32, int64, uint8....

特殊类型：int, uint, uintptr(指针)，由操作系统指定

浮点型：float32, float64(default)

复数：complex64, complex128

布尔值：bool

进制：`077`，`0xff`，不支持二进制

查看变量类型：`fmt.Printf("%T", a)` 或者 `reflect.TypeOf(a).String()`

字符串类型

串使用双引号，字符使用单引号，多行用反引号。

字符串处理函数都放在`strings`包中

```
var (
    s1 = "你好啊"
    s2 = '你'
)

func main() {
    //fmt.Println(a, b, c, d, e)
    fmt.Println(reflect.TypeOf(s2).String())
    fmt.Println(strings.Contains(s1,"好"))
    fmt.Println(strings.Split(s1,"好"))
    fmt.Println(strings.Count(s1,"好"))
    fmt.Println(strings.Repeat(s1,2))
}
```

对于中文等字符，对应的字符类型为`rune`类型，相当于`int32`类型

字符串修改：

一般来说字符串不可以修改，通常经过将string类型转为rune类型来进行修改

```
s3 := "北京市朝阳区"
s4 := []rune(s3)
s4[5] = '市'
fmt.Println(string(s4))
```

流程控制

if语句

```
if a:=1; a > 1{ // 可以先在分号前进行变量定义，在进行判断，减少内存占用
      //...
}
```

循环只有for，没有while

```
//1
for i:=0; i<10; i++{
        fmt.Printf("%v ", i)
}

//2
var a = 2
for a < 10 {
      //...
}

//3
for {
      // 死循环
}
```

for range键值循环

```
s := "Aoli给"
for i, v := range s{
	fmt.Printf("%d, %c\n" ,i, v)
}
```

switch语句

```
s := 1
switch s {
	case 1:
	fmt.Println(1)
	case 2:
	fmt.Println(2)
	default:
	fmt.Println("None")
}
```

变种1：

```
s := 13
switch a:=10;s {    // 赋值
	case 1,3,5,7,a+2:   //多情况判断
	   fmt.Println(1)
	case 2:
	   fmt.Println(2)
	default:
	   fmt.Println("None")
}
```

变种2：

```
switch {    // 跟表达式的时候不再需要加变量
	case age < 50:
	   fmt.Println("OK")
	case age < 100 && age > 50:
	   fmt.Println(2)
	   fallthrough // 与C语言中的case兼容，不判断直接执行下一句
	default:
	   fmt.Println("None")
}
```

goto语句

运算

位运算：& | >> << ^

逻辑运算：&& || !

复杂数据类型

数组Array

定义必须指定其长度和元素类型

不同长度同一元素类型的数组不是同一个类型，长度也是数组类型的一部分

数组属于值类型！不是引用类型

```
var a1[3]bool
var a2[4]int
a1 = [3]bool{true, false, true}
a2 = [...]int{1,5,4,8}  //自动推断长度
a3 := [5]float64{1,2}   // {1,2,0,0,0}
a4 := [4]int{0:2, 3:1} // {2,0,0,1}
```

多维数组的定义：

```
a5 := [3][2]int{
     {1,2},
     {3,4},
     {5,6},
}   // [[1 2] [3 4] [5 6]]
```

数组的复制：

```
a3 = [5]float64{1, 2}
a6 := a3
a6[0] = 4
fmt.Println(a3) //[1 2 0 0 0]
fmt.Println(a6) //[4 2 0 0 0]
```

数组逻辑运算：支持运算符 "==" "!="

```
a3 = [5]float64{1, 2}
a6 := a3
fmt.Println(a3 == a6)   // true
a6[0] = 4
fmt.Println(a3) //[1 2 0 0 0]
fmt.Println(a6) //[4 2 0 0 0]
fmt.Println(a3 == a6)   // false
```

切片Slice

> 拥有可变长度并且相同元素类型的序列，相当于数组不需要指定长度即可

是引用类型，可以与`nil`进行比较，nil的切片没有底层数组，切片不能比较

```
var a = []int{1,2,3}
fmt.Printf("%T, %v, %v\n", a, a, a==nil)
a = append(a, 4, 5)
```

切片的长度和容量：`len(s)  cap(s)`

数组转切片，引用类型，因此会更改所有的切片：

```
var a = [...]int{1,2,3,4,5}
a1 := a[0:2]    //由数组得到的切片，[1,2]
a2 := a[1:]     //{2,3,4,5}
```

```
a3 := a[:2]     //{1,2}
a4 := a[:]      //全拿过来
a1[0] = 11
a2[1] = 12
a[2] = 13
fmt.Println(a, a1, a2)
//a [11,12,13,4,5]
//a1 [11,12,13]
//a2 [12,13,4,5]
fmt.Println(len(a1), cap(a1), cap(a2))  // 2 5 4

// 从某个位置开始，后移1个数量
a1 = append(a1, 0)
copy(a[i+1:], a[i:])
a1[i] = 666
```

切片容量是底层数组从**切片第一个元素**到**数组的最后一个元素**的长度

切片初始化make(T type, len int, cap int)

```
ints := make([]int, 5, 10)  // 长度为5，容量为10
```

切片扩容append与复制copy:

```
s1 := []int{1}
fmt.Printf("%p cap:%v, %v\n",s1,cap(s1), s1)
s1 = append(s1, 2)
fmt.Printf("%p cap:%v,%v\n",s1,cap(s1), s1)
s1 = append(s1, 3)
fmt.Printf("%p cap:%v,%v\n",s1,cap(s1), s1)
s1 = append(s1, 4)
fmt.Printf("%p cap:%v,%v\n",s1,cap(s1), s1)
s1 = append(s1, 5)
fmt.Printf("%p cap:%v,%v\n",s1,cap(s1), s1)
0xc00000a098 cap:1, [1]
0xc00000a0e0 cap:2, [1 2]
0xc00000e1c0 cap:4, [1 2 3]
0xc00000e1c0 cap:4, [1 2 3 4]
0xc000010380 cap:8, [1 2 3 4 5]
```

当扩容超过原来切片的容量的时候，切片指向的底层数组地址会发生变化。

```
s2 := []int{1,2,3,4}
s3 := []int{1,2,3,4}
s2 = append(s2, s3...)  // 切片之间append需要进行展开
s3[0]=5
// 切片之间的添加是向s2中添加，不会同链表一样
fmt.Println(s2)     //[1 2 3 4 1 2 3 4]
fmt.Println(s3)     //[5,2,3,5]
```

对于copy相当于对底层数组进行copy，不会互相影响

```
q1 := []int{6,6,6,6}
q2 := make([]int, 6, 6) // 首先要进行分配空间，否则无法复制
copy(q2,q1)
q1[0]=5
fmt.Println(q1,q2)  // [5 6 6 6] [6 6 6 6 0 0]
```

索引删除元素:

> Go中没有对索引进行删除的操作

```
q1 := [1,3,5]
q1 := append(q1[:1],q1[2:]...)  // 删除索引为1的元素
fmt.Println(q3, cap(q1)) // [1 5] 3
```

进行元素删除后，其容量还是根据从切片开始元素在底层数组的索引到最后元素的索引。

本质上相当于删除索引将后面的元素往前移动一位

避免切片内存泄漏：

```
func findTel(file string) []byte{
	b, _ := ioutil.ReadFile(file)
	return regexp.MustCompile("[0-9]+").find(b)
}
// 这个可能只需要小部分的数据，但是会一直保存着整个文件的数据，算是一种“非传统”的内存泄漏。
// 可以改为下面的写法，重新分配一个切片的底层数组
func findTel(file string) []byte{
	b, _ := ioutil.ReadFile(file)
	return append([]byte{}, regexp.MustCompile("[0-9]+").find(b))
}
```

指针

> go中不存在对指针进行操作

`&`：取地址

`*`：根据地址内容

```
var a *int
b := 9
a = &b
*a = 5
fmt.Println(b)  // 5
```

make和new的区别：make直接返回内存，new之间返回内存的指针。make用于返回`slice` `map` `chan`的类型

```
a1 := new(int)
*a1 = 20
fmt.Println(a1, *a1)    // 0xc00000a0c0 20
```

map

> 使用hash实现

```
m1 := make(map[string]int, 10)  // map需要进行初始化分配空间
m1["Age"] = 10
m1["Name"] = 98
fmt.Println(m1)
```

```
for i, v := range m1{   // 遍历
      fmt.Println(i,v)
}

a1, o := m1["s"]    // 对于一个不存在的键值对，使用bool值进行接收，对应键值为初始数据类型的值
fmt.Println(a1, o)  // 0 false

delete(m1, "Age")
fmt.Println(m1)
delete(m1, "Age1")  // 删除不存在的键值对不会提示
fmt.Println(m1)
```

函数

函数定义方式

```
func ok(){
      print('s')
}

func sum(a float64, b float64) float64 {
    return a-b
}

func add(a int, b int) (c int){
    c = a+b
    return  // 如果由定义返回值变量，则可以空return
}

func multi() (int, string) {    // 多个返回值
    return 1, ""
}

func sub(a, b, c int, m, n string) int {        // 参数类型相同时候，前面的参数类型定义可以省略
    return a-b-c
}

func auto(a string, b ...int){  // 可变长参数，必须放在函数最好
    fmt.Println(a)
    fmt.Println(b)  // slice
}
```

Go语言没有默认参数

defer语句

将其后面的语句在函数即将返回的时候再执行。

```
func DeferDemo(){
    fmt.Println("a")
    defer fmt.Println("e")
    defer fmt.Println("d")
    defer fmt.Println("c")
    fmt.Println("b")
}
// 顺序 a b c d e
```

对于都是defer修饰的语句，越靠后的语句越先执行。

※对于Go语言中，函数返回分为两步：1. 给返回值赋值 2. 执行RET指令。而defer语句就是在这两句之间执行。

defer的面试题：

```
func f1() int {
    x := 5
    defer func() {
        x++
    }()
    return x
} // 在defer之前已经赋予返回值了

func f2() (x int) {
    defer func() {
        x++
    }()
    return 5
} // 对返回值变量操作

func f3() (y int) {
    x := 5
    defer func() {
        x++
    }()
    return x
} // 同f1

func f4() (x int) {
    defer func(x int) {
        x++
    }(x)
    return 5
} // 还是5，改变的是返回值参数的副本

func main() {
    fmt.Println(f1())
    fmt.Println(f2())
    fmt.Println(f3())
    fmt.Println(f4())
}
```

函数类型

函数也可以作为一种参数和返回值

```
func f1() {
    fmt.Println("s")
}

func f2(int, int) string{
    return ""
}

func main() {
    a1 := f1
    a2 := f2
    fmt.Printf("%T,%T,%v,%v",a1, a2,a1,a2)  // func(),func(int, int) string,0x4e94e0,0x4e9580
      a1()
}
```

匿名函数

```
func f2(int, int) string{
    ff2 := func() { //匿名函数，用在函数内部
        fmt.Println("Good Func")
    }
    ff2()
    return ""
}
```

闭包

> 闭包让你可以在一个内层函数中访问到其外层函数的作用域。

举个JS的例子：

```
function makeAdder(x) {
   return function(y) {
      return x + y;
  };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);

console.log(add5(2));   // 7
console.log(add10(2)); // 12
```

JS使用闭包可以绑定事件：

```
function makeSizer(size) {
   return function() {
      document.body.style.fontSize = size + 'px';
  };
}

var size12 = makeSizer(12);
var size14 = makeSizer(14);
var size16 = makeSizer(16);

document.getElementById('size-12').onclick = size12;
document.getElementById('size-14').onclick = size14;
document.getElementById('size-16').onclick = size16;
```

GO闭包：

```
func f1(x int)int {
    fmt.Println(x , "sad")
    return x + 1
}

func f3(f func(int)int) func(int)int {
    x := f(4)
    return func(y int) int{
        return x + y
    }
}

func main() {
    q1 := f3(f1)
    q2 := q1(6)
    fmt.Println(q2) // 11
}
```

结构体

> 结构体是值类型，不是引用类型，占用连续的一块内存

类型别名和自定义类型

```
type myInt int  // 自定义类型
type a = int    // 类型别名

func main() {
    var n myInt
    n = 5
    fmt.Printf("%T\n",n)        // main.myInt
    var m a
    m = 6
    fmt.Printf("%T\n",m)        // int
}
```

构建结构体

> 首字母大写表示对外公开的，小写为私有的

```
type Person struct {
    name string
    age int
    hobby []string
}

var p Person
p.name = "John"
p.age = 18
p.hobby = []string{"Play", "Suck", "Pack"}
fmt.Println(p, reflect.TypeOf(p))   // {John 18 [Play Suck Pack]} main.Person

//或者使用new进行创建

var a = new(Person)

// key-value初始化
p2 := Person{
      "Jack",
      2,
     []string{"Sell", "Song", "Sing"},
}

fmt.Println(p2)
```

匿名结构体（只使用一次）：

```
var k struct{
      x int
      y int
}
k.x = 1
k.y = 2
fmt.Println(k)
```

结构体参数

```
func f(x *Person){  // 需要这样赋值
    x.age = 20
}

f(&p)
fmt.Println(p)
```

方法和接收者

> 有点类似面向对象中的this

```
type person struct {
    name string
    age int
```

```
    pets []string
}

func newPerson(name string, age int) *person {
    return &person{name: name, age: age}
}

func (p person) sing() {
    fmt.Println(p, "is sing")
}

func main() {
    p1 := newPerson("Sarah", 30)
    p1.sing()
}
```

使用接收者对结构体“对象”属性进行操作：

```
func (p *person) addAge(){
    p.age++
}

func main(){
      fmt.Println(p1.age)
    p1.addAge()
    fmt.Println(p1.age)
}
```

结构体匿名字段

> 不常用，也不建议使用

```

```

嵌套结构体

```
type person struct {
    name string
    age int
    pets []string
}

type Group struct {
    header person
    members int
}


g1 := Group{
      header:   person{
            "Gaye",
            30,
           []string{""},
     },
      members: 10,
}
```

结构体模拟继承

```
type animal struct {
    name string
    age int
}

type dog struct {
    animal  // 使用匿名变量可直接使用dog.name
    vaccine uint8
}

func (a animal)move() {
    fmt.Println(a.name, "is moving")
}

func (d dog) bark() {
    fmt.Println(d.name, "get vaccined")
}

func main() {
    d1 := dog{
        animal{
            "Tommy",
            4,
        },
        6,
    }

    d1.move()   // 继承得到
    d1.bark()   // 自身方法
}
```

JSON处理

json处理官方文档

结构体转json

```
type animal struct {
    Name string // 必须首字母大写保证可见
    Age   int
}

func main() {
    a := animal{
        Name: "Sarah",
        Age:   2,
    }
    by, _ := json.Marshal(a)
    fmt.Println(string(by))
}
```

反序列化:

```
type an2 struct {
    Name string `json:"Name"`   // 反序列化使用这种方法进行指定json的字段
    Age   int      `json:"Age"`
}

func main() {
    s1 := `{"Name":"Sarah","Age":2}`
    var a2 an2
    json.Unmarshal([]byte(s1), &a2)
    fmt.Printf("%v", a2)
}
```

接口

> 接口是一种类型

接口定义

```
type name interface{
      method1(arg1, arg2)(ret1, ret2)
      method2(arg1, arg2)(ret1, ret2)
}
```

案例

```
type obj interface {    // 定义接口类型
    speak()
}

type dog struct {}
type cat struct {}
type person struct {}

func (d dog)speak() { fmt.Println("aoaoao")}
func (c cat)speak() { fmt.Println("miaomiaomiao")}
func (p person)speak() { fmt.Println("aaa")}

func snoozee(x obj) {
    x.speak()
}

func main() {
    var d dog
    var c cat
    var p person
    snoozee(d)
    snoozee(c)
    snoozee(p)
}
```

向下转型

```
func main() {
    var d dog
    var c cat
    var p person
    snoozee(d)
    snoozee(c)
    snoozee(p)
    // 向下转型
    var x obj
      fmt.Printf("%T\n", x)   // nil
    x = d
    fmt.Printf("%T\n", x)   // main.dog
    x = c
    fmt.Printf("%T\n", x)   // main.cat
}
```

值接收者和指针接收者的区别

上例

指针接收者可以对原来的结构体对象进行属性修改。

使用值接收者和指针接收者实现接口向下指针需要取地址。 `&`

多接口与多接收者

一个结构体可以实现多个接口，接口也可以进行嵌套

空接口

> 多用于存储各种类型的数据

定义为

```
interface{}
```

通常不需要写名字，任意类型的变量都可以设置为空接口。

```
func show(a interface{}){
    fmt.Printf("Value: %v, type: %T\n", a, a)
}

func main() {
    var m1 map[string]interface{}
    m1 = make(map[string]interface{}, 16)
    m1["name"] = "Jonny"
    m1["age"] = 16
    m1["married"] = true
    m1["wives"] = []string{"Marry", "Lisa", "Linda"}
    show(nil)   // Value: <nil>, type: <nil>
    show(false) // Value: false, type: bool
    show(m1)    //Value: map[age:16 married:true name:Jonny wives:[Marry Lisa Linda]], type:
map[string]interface {}
}
```

类型断言以及判断实现接口类型

> 对于interface类型使用类型断言进行“向下转型”

```
value, flag := a.(string)
```

`value`为转换后的值， `flag`为是否转换成功。

```
func show(a interface{}) {
    fmt.Printf("Value: %v, type: %T\n", a, a)
    switch a.(type) {   // 使用这个分
    case string:
            fmt.Println("Is string", a.(string))
    case int:
        fmt.Println("Is Int", a.(int))
    case bool:
        fmt.Println("Is Bool", a.(bool))
    default:
        fmt.Printf("%v\n", "Other")
    }
}
```

包package

包中的方法名，大写表示公开方法，小写表示私有方法

引用包：

```
package main

import nick "awesomeProject/16-package/lib" // 别名  从gopath目录开始计算

func main() {
    nick.Add(4,2)
}
```

GO语言中禁止循环导入包

init初始函数

> 在导入包的时候会自动触发包中定义的init函数

```
func init() {
    fmt.Println("本包init")
}

func main() {
    nick.Add(4,2)
}
//导入包init
//本包init
```

省略前缀

```
import (
    ."fmt"
)

func init() {
    Println("本包init")   // 可以直接调用包下的函数
}
```

文件操作

打开读取文件

OS读取

```
file, err := os.Open("F:/a.txt")
if err != nil {
      fmt.Println("打开文件失败")
      return
}
var data [1024]byte
n, err := file.Read(data[:])
if err != nil {
      fmt.Println("Read Error")
      return
}
if err == io.EOF{   // 读取完毕
      fmt.Println("Read Over")
      return
}
fmt.Println(n, string(data[:n]))
file.Close()    // 重要
```

bufio读取，适用于每次读取一行

```
func bufioRead(){
    file, err := os.Open("F:/win10应用.txt")
    if err != nil {
        fmt.Println("文件打开失败", err)
        return
    }
    defer file.Close()
    // 创建读取内容的对象
    reader := bufio.NewReader(file)
    line, err := reader.ReadString('\n')
    if err != nil{
        fmt.Printf("err:%v\n", err)
        return
    }
    if err == io.EOF{
        fmt.Printf("%v\n", "Read Over")
        return
    }
    fmt.Println(line)
```

```
}
```

IOUtil

这个会自动帮忙进行打开和关闭文件，适用于读取整个文件。

```
func ioutilRead() {
    content, err := ioutil.ReadFile("F:/win10应用.txt")
    if err != nil {
        fmt.Printf("\"File Read Err, err: %v\n", err)
        return
    }
    fmt.Println(string(content))
}
```

写入文件

使用`os.OpenFile(name string, flag int, perm FileMode)`

其中`flag`有以下几种：可以通过`|`连接符进行连接多个状态

```
O_RDONLY int = syscall.O_RDONLY // open the file read-only.
O_WRONLY int = syscall.O_WRONLY // open the file write-only.
O_RDWR   int = syscall.O_RDWR   // open the file read-write.
// The remaining values may be or'ed in to control behavior.
O_APPEND int = syscall.O_APPEND // append data to the file when writing.
O_CREATE int = syscall.O_CREAT  // create a new file if none exists.
O_EXCL   int = syscall.O_EXCL   // used with O_CREATE, file must not exist.
O_SYNC   int = syscall.O_SYNC   // open for synchronous I/O.
O_TRUNC  int = syscall.O_TRUNC  // truncate regular writable file when opened.
```

`perm`为linux下的文件权限，"0644"， “0777”等

普通写入：

```
func writeFile1(){
    file, err := os.OpenFile("./mock.txt", os.O_WRONLY | os.O_CREATE | os.O_TRUNC, 0644)
    if err != nil {
        fmt.Println("File not exists")
        return
    }
    file.Write([]byte{97,98,99,'\n'})   // Write写入byte
    file.WriteString("666牛批")       // Write写入字符串
    file.Close()
}
```

bufio写入：

```go
func writeFile2(){
    file, err := os.OpenFile("./mock.txt", os.O_WRONLY | os.O_CREATE | os.O_TRUNC, 0644)
    if err != nil {
        fmt.Println("File not exists")
        return
    }
    writer := bufio.NewWriter(file)
    writer.Write([]byte{99,100,101,'\n'})
    writer.WriteString("奥里给")
    writer.Flush()  // 重要，必须清空缓冲区
    file.Close()
}
```

ioutil直接写入：

> 直接替换原来文件中的所有内容

```go
func writeFile3(){
    err := ioutil.WriteFile("./mock.txt", []byte("asdsss"), 0644)
    if err != nil {
        return
    }
}
```

Time包

时间戳转换

```go
fmt.Printf("%v\n", time.Now().Unix())    // 获取当前时间戳
fmt.Println(time.Unix(1623906946,0))     // 时间戳转换为时间
```

定时器

```go
func main() {
    timer := time.Tick(time.Second)  // 一秒钟重新赋一次值
    for i := range timer{
         fmt.Println(i)    // 一秒钟执行一次
    }
}
//2021-06-17 13:22:25.0036191 +0800 CST m=+1.028786501
//2021-06-17 13:22:26.0021729 +0800 CST m=+2.027340301
//2021-06-17 13:22:26.9983606 +0800 CST m=+3.023528001
//2021-06-17 13:22:27.9986587 +0800 CST m=+4.023826101
//2021-06-17 13:22:28.9948536 +0800 CST m=+5.020021001
```

时间格式化

```go
time.Now().Format("2006-01-02 15:04:05")
```

strconv库

其他类型转string

```
b1 := true
s1 := fmt.Sprintf("%v", b1)
fmt.Println(s1)
```

string转其他类型：

```
func strconv.ParseInt(s string, base int, bitSize int)(i int64, err error)
func strconv.ParseUint(s string, base int, bitSize int) (uint64, error)
func strconv.Atoi(s string) (int, error)
func strconv.ParseFloat(s string, bitSize int) (float64, error)
func strconv.ParseBool(str string) (bool, error)
```

异常处理

自定义异常

1. `errors.New("txt")` 不支持自定义内容
2. `fmt.Errorf("Errors: %v \n", err)`

并发编程

问题

- goroutine与线程的关系：

  goroutine是用户级线程（协程），OS线程默认初始栈为2MB，而goroutine刚开始只有2KB，大的时候可以达到1GB

-

goroutine

主线程的goroutine结束的话，由其生成的goroutine也会随之结束。

当一个goroutine执行的函数结束的话，goroutine的生命周期也结束了。

开启goroutine

```
func hello (i int) {
    fmt.Println("Hello routine", i)
}

func main() {
    for i := 0; i < 5000; i++ {
        go hello(i)
    }
    fmt.Println("main")
    time.Sleep(1000)
}
```

闭包问题

闭包使用goroutine会出现问题

```
func bug() {
    for i := 0; i < 1000; i++ {
        go func() {
            fmt.Println(i)
        }()
    }
}

func main() {
    bug()
    time.Sleep(1000)
}
/*
...
823
819
587
823
823
823
823
453
823
...
*/
```

由于需要访问外部for中的`i`变量，启动goroutine需要耗费时间

改正方式：

```
func bug() {
    for i := 0; i < 1000; i++ {
        go func(i int) {
            fmt.Println(i)
        }(i)
    }
}
```

waitgroup

> 等待所有的非主goroutine执行完毕，主goroutine才能结束。

```
var wg sync.WaitGroup

func thread() {
    for i := 0; i < 10; i++ {
        wg.Add(1)   // signal
        go func(i int) {
            fmt.Println(i)
            wg.Done()   // await
        }(i)
    }
}
```

```
func main() {
    thread()
    wg.Wait()
}
```

调度模型GMP

G：goroutine　M：machine　P：processor

- `G`很好理解，就是个goroutine的，里面除了存放本goroutine信息外 还有与所在P的绑定等信息。
- `P`管理着一组goroutine队列，P里面会存储当前goroutine运行的上下文环境（函数指针，堆栈地址及地址边界），P会对自己管理的goroutine队列做一些调度（比如把占用CPU时间较长的goroutine暂停、运行后续的goroutine等等）当自己的队列消费完了就去全局队列里取，如果全局队列里也消费完了会去其他P的队列里抢任务。
- `M（machine）`是Go运行时（runtime）对操作系统内核线程的虚拟， M与内核线程一般是一一映射的关系， 一个groutine最终是要放到M上执行的；

runtime包

- `runtime.Gosched()`：让出时间片，重新等待安排任务。

  让当前协程让出CPU让其他协程执行，当时间片轮到它的时候再执行。

  ```
  func main() {
      runtime.GOMAXPROCS(10)
      go func(s string) {
          for i := 0; i < 10; i++ {
              fmt.Println(s)
          }
      }("world")
      // 主协程
      for i := 0; i < 2; i++ {
          // 切一下，再次分配任务
          runtime.Gosched()
          fmt.Println("hello")
      }
  }
  ```

- `runtime.Goexit()`： 结束当前协程

  ```
  func main() {
      var wg sync.WaitGroup
      wg.Add(1)
      go func() {
          defer wg.Done()
          defer fmt.Println("A.defer")
          func() {
              defer fmt.Println("B.defer")
              // 结束协程
              runtime.Goexit()
              defer fmt.Println("C.defer")
              fmt.Println("B")
          }()
          fmt.Println("A")
      }()
  ```

```
    wg.Wait()
}
```

- `runtime.GOMAXPROCS(n int)`：Go运行时的调度器使用GOMAXPROCS参数来确定需要使用多少个OS线程来同时执行Go代码。

Channel

引用类型

单纯地将函数并发执行是没有意义的。函数与函数间需要交换数据才能体现并发执行函数的意义。

GO的共享模型是CSP(Communicating Sequential Process)，是通过**通信来进行共享内存**，而不是通过共享内存来进行通信。

`goroutine`是GO程序并发的执行体，`channel`是它们之间的连接，是`goroutine`之间通信的连接。channel数据结构相当于一个队列，因此需要指定其元素的基本类型。

`channel`是一种特殊的类型，通道像一个队列，遵循FIFO规则

创建channel：

需要进行初始化才能使用，否则为`nil`

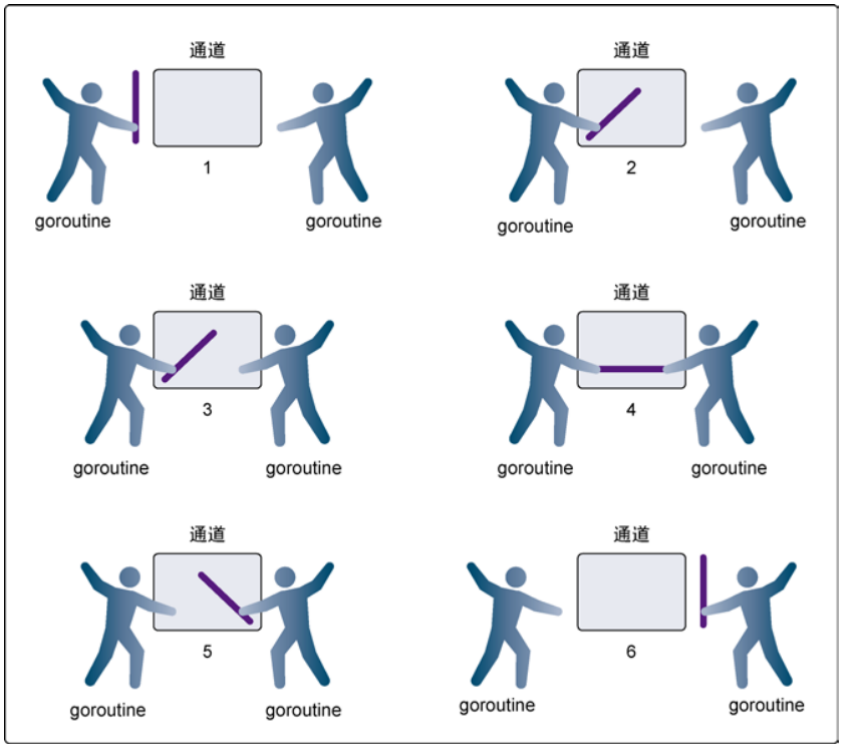
```
var ch chan int
ch = make(chan int, [缓冲大小])
```

通道操作：

`<-`

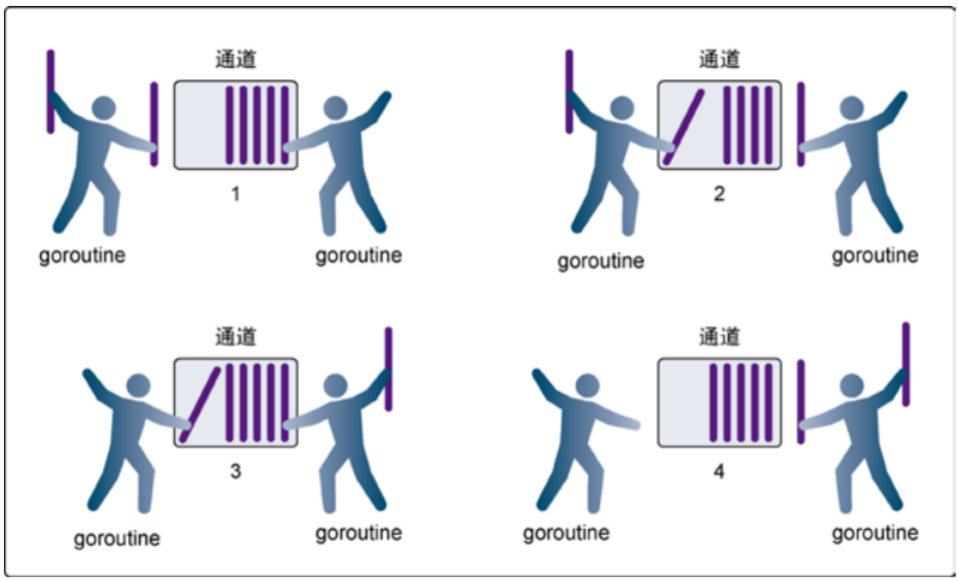
- 发送操作：`channel <- 1`。将1数值保存到channel中
- 接收操作：`x <- channel`。从channel中取出数值到`x`变量中
- 关闭：`close(channel)`，channel关闭之后只能读不能写入。

无缓冲的通道又称为**阻塞通道，同步通道**，因此必须要有接收信息和发送信息的goroutines。



使用无缓冲的通道在 goroutine 之间同步

有缓冲通道，



使用有缓冲的通道在 goroutine 之间同步数据

无缓冲区channel实例

```
func noBufChannel(){
    wg := sync.WaitGroup{}
    wg.Add(1)
    ch := make(chan int)
    go func() {
        x := <- ch  // 如果未收到会一直阻塞
        fmt.Println("We get ",x)
        wg.Done()
    }()

    fmt.Println("We Send")
    time.Sleep(time.Second*5)
    ch <- 2
    wg.Wait()
}
//We Send
//We get  2     %(5 seconds later)
```

含缓冲区的channel

```
func bufChannel(){
    ch := make(chan int, 1)
    ch <- 1
    fmt.Println("放入了1")
    ch <- 2  // 会报错，无接收的goroutine
    fmt.Println("放入了2")
}
```

需要进一步了解，样例比较简要

遍历channel：

```
for v := intChan{
      fmt.Println(v)
}
```

如果管道未关闭，则会报deadlock。

只读channel类型：

```
func channelF1Reader(mes <-chan string) {
    msg := <-mes
    fmt.Println(msg)
}
```

只写channel类型：

```
func channelF2Writer(mes chan<- string) {
    mes <- "www.ydook.com"
}
```

channel与goroutine的结合：

```
package main

import (
    "fmt"
    "sync"
)

var wg sync.WaitGroup
var arr []int

func writeData(c *chan int) {
    for i := 0; i < 5; i++ {
        *c <- i
    }
    close(*c)   // 关闭管道
    wg.Done()
}

func readData(c *chan int){
    for i := 0; i < 6; i++ {
        num, ok := <-*c     // 读取管道的值
        fmt.Println("over:", ok)
        if ok{
            arr = append(arr, num)
        }

    }
    wg.Done()
}

func main() {
    wg.Add(2)
    c := make(chan int, 0)
    go writeData(&c)
    go readData(&c)
```

```
    wg.Wait()
    for _, v := range arr {
        fmt.Printf("Pipe recv:%v\n", v)
    }
}

//输出:
/*
over: true
over: true
over: true
over: true
over: true
over: false
Pipe recv:0
Pipe recv:1
Pipe recv:2
Pipe recv:3
Pipe recv:4
*/
```

如果不关闭管道，继续读取空管道会报错；已关闭后会返回 `ok` 的值为false， `num` 数值为0。

定时器

`time.NewTimer(d)` ，过一段时间d定时器会发送其当前时间到管道 `<-chan Time` 中，且只能响应一次。

```
func t1() {
    timer := time.NewTimer(time.Second)
    t1 := time.Now()
    fmt.Printf("t1:%v\n", t1)
    t2 := <-timer.C
    fmt.Println(t2)
}
```

三个方法实现延时功能:

```
func t2() {
    time.Sleep(time.Second * 2)                 // (1)
    timer := time.NewTimer(time.Second * 2)     // (2)
    <-timer.C
    fmt.Println("2 seconds")
    <- time.After(time.Second * 2)              // (3)
    fmt.Println("2 seconds")
}
```

停止定时器:

```
func t3() {
    timer := time.NewTimer(time.Second * 10)
    go func() {
        <-timer.C
        fmt.Println("wait OK!") // STOP之后，chan后面的语句不会执行
    }()
    time.Sleep(time.Second * 3)
    fmt.Println("STOP 10 seconds timer.")
    b := timer.Stop()
    if b{
        fmt.Println("Stop success")
    }
}
```

重置定时器：

```
func t4() {
    timer := time.NewTimer(time.Hour)
    b := timer.Reset(time.Second * 10)
    if b {
        fmt.Println("重新设置成功")
    }else{ fmt.Println("Timer或者已经过期") }
    <-timer.C
    fmt.Println("OK")
}
```

循环定时器`time.Ticker()`：

```
func t5() {
    ticker := time.NewTicker(time.Second * 2)
    var wg sync.WaitGroup
    wg.Add(1)
    go func() {
        i := 0
        for {
            <-ticker.C
            i++
            fmt.Printf("Once ticker:%v\n", i)
            if i == 5 {
                break
            }
        }
        wg.Done()
    }()
    wg.Wait()
}
```

select

> 多路复用

对于同时都向管道输入的数据，select会随机选择一个进行执行。如果想要全部执行，需要一直执行select语句。

```
func g1(ch *chan int) {
    *ch <- rand.Int()
```

```
    time.Sleep(time.Second * 2)
    fmt.Println("g1 ok")
}

func g2(ch *chan string) {
    *ch <- "2"
    time.Sleep(time.Second * 2)
    fmt.Printf("g2 recv:%v\n", <-*ch)
}

func main() {
    var ch1 = make(chan int)
    var ch2 = make(chan string)
    go g2(&ch2)
    go g1(&ch1)
    go func() {
        for  {
            select {
            case s1 := <-ch1:
                fmt.Println("ch1 recv:",s1)
            case s2 := <-ch2:
                fmt.Println("write ok:", s2)
            default:
                fmt.Println("Nothing Channel")
            }
            time.Sleep(time.Second * 2)
        }
    }()
    for {}
}
```

上下文context

> 上下文 context.Context是Go 语言中用来设置截止日期、同步信号，传递请求相关值的结构体。

Go 语言并发编程与 Context

context.Context结构体

```
type Context interface {
    Deadline() (deadline time.Time, ok bool)
    Done() <-chan struct{}
    Err() error
    Value(key interface{}) interface{}
}
```

- `Deadline` — 返回 context.Context 被取消的时间，也就是完成工作的截止日期；
- `Done` — 返回一个 Channel，这个 Channel 会在当前工作完成或者上下文被取消后关闭，多次调用 Done 方法会返回同一个 Channel；
- `Err` — 返回 context.Context 结束的原因，它只会在 Done 方法对应的 Channel 关闭时返回非空的值；如果 context.Context 被取消，会返回 Canceled 错误；如果 context.Context 超时，会返回 DeadlineExceeded 错误；
- `Value` — 从 context.Context 中获取键对应的值，对于同一个上下文来说，多次调用 Value 并传入相同的 Key 会返回相同的结果，该方法可以用来传递请求特定的数据；

使用context同步信号

```
func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 1*time.Second)
    defer cancel()

    go handle(ctx, 500*time.Millisecond)    // ctx时限为1s，设置请求时间为500ms
    select {
    case <-ctx.Done():
        fmt.Println("main", ctx.Err())
    }
}

func handle(ctx context.Context, duration time.Duration) {
    select {
    case <-ctx.Done():
        fmt.Println("handle", ctx.Err())
    case <-time.After(duration):
        fmt.Println("process request with", duration)
    }
}

// 输出
/*
process request with 500ms
main context deadline exceeded
*/

// 当请求时间为1500ms的时候

/*
main context deadline exceeded
handle context deadline exceeded
*/
```

反射

> 在运行的时候可以动态的获取变量的信息

有两大接口：`reflect.Type` 和 `reflect.Value`

获取变量类型：`reflect.TypeOf(v)` 和 `reflect.ValueOf(v)`

举例

```
import (
    "fmt"
    "reflect"
)

func test(v interface{}) {
    vRet := reflect.ValueOf(v)  // interface{} cast to reflect.Value type
```

```
    iRet := vRet.Interface()    // reflect.Value cast to interface{} type
      num := iRet.(int)             // interface{} type assert to int
    fmt.Println(num)
}

func main() {
    test(2)
}
```

GoWeb

简单服务器

```
package main

import (
    "fmt"
    "net/http"
)

func sayHello(w http.ResponseWriter, r *http.Request) {
    _, _ = fmt.Fprint(w, "Hello, Simple Server")
}

func main() {
    http.HandleFunc("/hello", sayHello)
    err := http.ListenAndServe(":80", nil)
    if err != nil {
        fmt.Println("HTTP Server Start Failed!", err)
    }
}
```

gin框架

> 可替换国产框架 GoFrame

安装gin：`go get -u github.com/gin-gonic/gin`

简易使用

```
func main() {
    engine := gin.Default()

    engine.GET("/hello", func(context *gin.Context) {   // 处理GET请求
        fmt.Println(context.FullPath()) //打印访问根目录后的路径
        _, _ = context.Writer.Write([]byte("6666")) // 输出到网页中
    })

      if err := engine.Run(":8090"); err != nil{  // 自定义端口
        log.Fatal(err.Error())
    }
}
```

创建engine

在gin框架中engine定义为一个结构体，其中包含了路由组，中间件，页面渲染接口，框架配置等相关内容。可以使用两种方式进行创建：

```
gin.Default()
gin.New()
```

`Default()` 底层是使用了 `New()` 方法，并且使用了 `Logger` 和 `Recovery` 两个中间件。

处理HTTP请求

通用处理函数 `handle`：

```
package main

import (
    "fmt"
    "github.com/gin-gonic/gin"
)

func main() {
    e := gin.Default()

    // GET localhost:8899/hello?name=Jack&age=20&gender=
    e.Handle("GET", "/hello", func(context *gin.Context) {
        name := context.DefaultQuery("name", "John")
        age := context.Query("age")
        gender := context.Query("gender")
        s := fmt.Sprintf("name: %v, age: %v, gender:%v==", name, age, gender)
        fmt.Println(s)
        _, _ = context.Writer.Write([]byte(s))
    })

    // GET localhost:8899/lgin
    e.Handle("POST", "/login", func(context *gin.Context) {
        acc := context.PostForm("acc")
        pwd := context.PostForm("pwd")
        context.Writer.Write([]byte("Hello" + acc + ", Pwd:" + pwd))
    })

      //DELETE
      e.DELETE("/del/user/:id", func(context *gin.Context) {
        id := context.Param("id")
```

```
        context.Writer.Write([]byte(id))
    })

    err := e.Run(":8899")
    if err != nil {
        return
    }
}
```

或者直接使用 `GET` 和 `POST` 方法：

```
e.GET("/hello", func(context *gin.Context) {
	name := context.DefaultQuery("name", "John")
	age := context.Query("age")
	gender := context.Query("gender")
	s := fmt.Sprintf("name: %v, age: %v, gender:%v==", name, age, gender)
	fmt.Println(s)
	_, _ = context.Writer.Write([]byte(s))
})

// GET localhost:8899/login
e.POST("/login", func(context *gin.Context) {
	acc := context.PostForm("acc")
	pwd := context.PostForm("pwd")
	context.Writer.Write([]byte("Hello" + acc + ", Pwd:" + pwd))
})

e.DELETE("/del/user/:id", func(context *gin.Context) {
	id := context.Param("id")
	context.Writer.Write([]byte(id))
})
```

获取 `GET` 请求参数： `Query(key)` 或者 `DefaultQuery(key, DefaultValue)` ，后者为设定默认参数

获取 `POST` 请求参数： `PostForm(key)` 或者 `GETPostForm(key)` ，后者可以回传第二参数bool值表示串是否传入

获取 `DELETE` 请求参数：(RestfulAPI形式，比如： `"/del/user/:id"`) `Param(key)`

多数据处理和数据绑定

在开发的时候可能会面临很多的参数，单个处理效率较低。因此gin提供**表单实体绑定**

使用一个结构体进行接收：

> 注意这里结构体里的数据必须是大写

```
type Student struct {
	Name   string `form:"name" binding:"required"`
	Age    int    `form:"age"`
	Gender string `form:"gender"`
}
```

主要程序如下：

```
func main() {
    e := gin.Default()
    e.GET("/hello", func(context *gin.Context) {
```

```
        var s Student
        err := context.ShouldBindQuery(&s)  //注意这里是取地址
        fmt.Println(s)
        if err != nil {return}
        _, _ = context.Writer.Write([]byte(fmt.Sprintf("Name: %v, Age: %v, Gender:%v==", s.Name, s.Age,
s.Gender)))
    })

    // GET localhost:8899/lgin
    e.POST("/login", func(context *gin.Context) {
        var s Student
        err := context.ShouldBind(&s)
        if err != nil {return}
        context.Writer.Write([]byte("Hello " + s.Name + ", Age:" + s.Gender))
    })

    err := e.Run(":8899")
}
```

处理JSON的POST请求：

`BindJSON()`

```
e.POST("/login", func(context *gin.Context) {
      var s Student
      err := context.BindJSON(&s)
      if err != nil {
            return
     }
      context.Writer.Write([]byte("Hello " + s.Name + ", Age:" + s.Gender))
})
```

多类型返回数据结果

- []byte类型

  ```
  context.Writer.Write()
  ```

- String类型：

  ```
  context.Writer.WriteString("\n奥里给")
  ```

- JSON类型

  使用结构体类型：

  ```
  type msg struct {
      Code int
      Name string
      Hobbies []string
  }

  func main() {
      e := gin.Default()
      e.GET("/json", func(c *gin.Context) {
          m := msg{Code:0,Name: "AKA", Hobbies: []string{"a","b"}}
          c.JSON(200, m)
      })
      err := e.Run(":8899")
  ```

```
    if err != nil {return
}
```

使用 `map[string]interface{}` 类型

```
func main() {
    e := gin.Default()
    e.GET("/json", func(c *gin.Context) {
        c.JSON(200, map[string]interface{}{
            "name": "Jack",
            "age": 18,
            "hobbies": []string{"Eat", "Fuck"},
        })
    })
    err := e.Run(":8899")
    if err != nil { return  }
}
```

- HTML类型

> 支持插值表达式，进行对html中的变量进行赋值，对于图片等静态资源需要进行目录映射，否则会出错。

html文件:

```
<!DOCTYPE html>
<html lang="en">
<head>
      <meta charset="UTF-8">
      <title>{{.title}}</title>
</head>
<body>
      <h1>{{.header}}</h1>
      <img src="../img/2-3.png" alt="">
</body>
</html>
```

Go文件

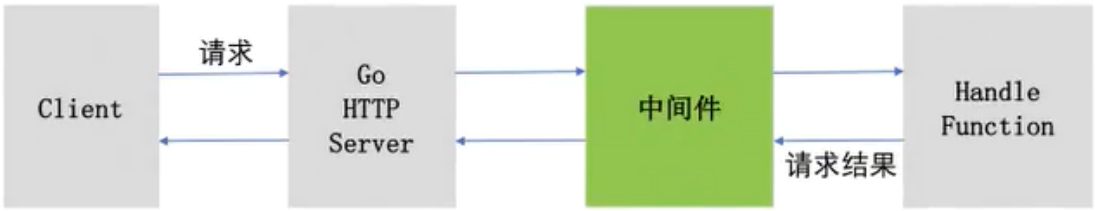
```
e := gin.Default()
e.LoadHTMLGlob("./html/*") // 从GOPATH路径开始计算
e.Static("/img", "./img")   // 将本地资源映射到访问链接中
e.GET("/html", func(c *gin.Context) {
    c.HTML(200,"index.html", gin.H{
         "title":"我的Title",
         "header":"我的路径"+c.FullPath(),
    })
})
err := e.Run(":8899")
if err != nil { return }
```

路由组分类

```
func main() {
    e := gin.Default()

    userGroup := e.Group("/user")

    userGroup.GET("/register", reg)
    userGroup.GET("/login", login)
    userGroup.GET("/del", del)

    err := e.Run(":8899")
    if err != nil {
        return
    }
}
```

中间件

> 一般用于鉴权处理，权限管理，安全验证，日志记录



gin中有两个中间件 `Logger` 和 `Recovery`

自定义中间件：

- 给全局使用中间件，使用 `Use()` 方法：

```
func main() {
    e := gin.Default()
    e.Use(reqInfo())    // 含括号
    e.GET("/mid", func(c *gin.Context) {
        _, _ = c.Writer.WriteString("middle ware demo")
    })
    err := e.Run(":8899")
    if err != nil {return}
}

func reqInfo() gin.HandlerFunc {
    return func(c *gin.Context) {
        path := c.FullPath()
        method := c.Request.Header
        fmt.Println(path, "\n", method)
    }
}
```

- 给单个路由使用中间件，放在第二个 `handleFunc` 的位置上：

```
func main() {
```

```
    e := gin.Default()
    e.GET("/mid", reqInfo() ,func(c *gin.Context) { // 放在第二个位置
        _, _ = c.Writer.WriteString("middle ware demo")
    })

    err := e.Run(":8899")
    if err != nil {return}
}

func reqInfo() gin.HandlerFunc {
    return func(c *gin.Context) {
        path := c.FullPath()
        method := c.Request.Header
        fmt.Println(path, "\n", method)
    }
}
```

中间件Next()

> 之前的方法都是在请求处理前进行调用，还需要对请求处理后进行调用

使用Next方法可以将中间件的处理函数分成两个部分：

相当于一个中断函数。

```
func reqInfo() gin.HandlerFunc {
    return func(c *gin.Context) {
        path := c.FullPath()
        method := c.Request.Header
        fmt.Println(path, "\n", method)
        c.Next()    // 相当于一个中断
        fmt.Println("请求处理后。。。")
    }
}
```

跨域请求

> 当请求的通信协议，主机地址，主机端口不相同的时候，就属于跨域请求(CORS, cross origin resources share)。

需要一个中间件进行设置服务器跨域请求：

```
func CORS() gin.HandlerFunc {
    return func(c *gin.Context) {
        origin := c.Request.Header.Get("Origin")
        if origin!=""{
            c.Header("Access-Control-Allow-Origin","*")
            c.Header("Access-Control-Allow-Methods", "POST, GET, OPTIONS, PUT, DELETE, UPDATE")
            c.Header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept,
Authorization")
            c.Header("Access-Control-Expose-Headers", "Content-Length, Access-Control-Allow-Origin,
Access-Control-Allow-Headers, Cache-Control, Content-Language, Content-Type")
            c.Header("Access-Control-Allow-Credentials", "true")
            c.Set("content-type","application/json")    // 设置返回数据类型
        }
```

```
        if c.Request.Method == "OPTIONS"{
            utils.Success(c, "OPTIONS SUCCESS")
        }

        c.Next()
    }
}
```

图片上传

gin框架使用 `SaveUploadedFile(file *multipart.FileHeader, pathName string)`

案例：

```
file, err := c.FormFile("aviator")  // 获取post的文件
// 获取后缀名，过滤文件类型
temp := strings.Split(file.Filename, ".")
fileSuffix := temp[len(temp)-1]
b := utils.AllowedUploadImgType[fileSuffix]
if !b{
      utils.Failed(c, "不允许上传该类型图片:"+fileSuffix)
      return
}
// md5重新命名文件名，并且使用函数SaveUploadedFile
hash := md5.Sum([]byte(fmt.Sprintf("%v_%v", mem.Id, time.Now().Unix())))
filename := fmt.Sprintf("%x.%v", hash, fileSuffix)
filepath := fmt.Sprintf("./upload/%v",filename)
err = c.SaveUploadedFile(file, filepath)
```

session管理

> http框架没有session管理就离谱，这里配合redis进行session管理

安装：`go get github.com/gin-contrib/sessions`

这个框架是在gin框架基础上进行开发的。

初始化SESSION，使用redis进行保存session信息：

```
// InitSession Init Session Operation
func InitSession(e *gin.Engine) {
    cfg := GetConfig()
    store, err := redis.NewStore(10, "tcp", cfg.Redis.Addr+":"+cfg.Redis.Port, cfg.Redis.Port,
[]byte("secret"))
    if err != nil {
        fmt.Println(err.Error())
        return
    }
    e.Use(sessions.Sessions("mySession", store))
}
```

设置session：

```go
func SetSession(c *gin.Context, key interface{}, value interface{}) error {
    session := sessions.Default(c)
    if session == nil {return nil}
    session.Set(key, value)
    return session.Save()
}
```

获取session：

```go
func GetSession(c *gin.Context, key interface{}) (value interface{}) {
    return sessions.Default(c).Get(key)
}
```

JWT验证

> JSON Web Token，使用的库为**dgrijalva/jwt-go**，推荐学习网址**JSON Web Tokens - jwt.io**

流程：

设置JWT参数：

```go
type MyClaims struct {  // 设置自己jwt想要保存的董事
    Username string `json:"username"`
    Id          string `json:"id"`
    Stamp      int64   `json:"stamp"`
    jwt.StandardClaims
}

const (
    TokenExpireDuration = time.Minute * 10
)

var JwtAuthorizationWhiteList = map[string]bool{    // JWT验证白名单
    "/api/register": true,
    "/api/login":      true,
}
```

```
var secretKey = []byte("`147258u.B%^OqpSt;~")   //密钥
```

生成JWT：

```
// GenToken Generate JWT by username and id
func GenToken(username, id string) (string, error) {
    c := MyClaims{
        Username: username,
        Id:           id,
        Stamp:      time.Now().Unix() * 2,
        StandardClaims: jwt.StandardClaims{
            ExpiresAt: time.Now().Add(TokenExpireDuration).Unix(),
            Issuer:      "PipePlanAdmin",
            NotBefore: time.Now().Unix(),
        },
    }
    token := jwt.NewWithClaims(jwt.SigningMethodHS256, c)
    return token.SignedString(secretKey)
}
```

解密JWT：

```
// DecryptToken tokenString comes from Request Header: Authorization:Bearer ...
func DecryptToken(BearerString string) (*MyClaims, int) {
    arr := strings.SplitN(BearerString, " ", 2)
    if len(arr) != 2 || arr[0] != "Bearer" {
        log.Println(fmt.Errorf("Authorization Token is invalid "))
        return nil, status.TokenFormatError
    }
    tokenString := arr[1]
    token, err := jwt.ParseWithClaims(tokenString, &MyClaims{}, func(token *jwt.Token) (interface{},
error) {
        return secretKey, nil
    })
    if err != nil {
        log.Println(fmt.Errorf("Token Parse Error Or has expired\n"))
        return nil, status.TokenParseErrorOrHasExpired
    }
    if claims, ok := token.Claims.(*MyClaims); ok && token.Valid { // examine token if is valid
        return claims, status.TokenStatusOK
    }
    log.Println(fmt.Errorf("Token expire or invalid. "))
    return nil, status.TokenHasExpired
}
```

验证码生成

使用的库：`go get github.com/mojocn/base64Captcha`

配置`base64Captcha`参数：

```
type CaptchaResult struct {
    Id            string `json:"id"`   // 必须
    Base64Blob string `json:"base_64_blob"`
```

```
    VerifyCode string `json:"code"`
}

type CaptchaConfig struct {
    Id              string `json:"id"`
    CaptchaType     string `json:"captcha_type"`
    VerifyValue     string `json:"code"`
    DriverString *base64Captcha.DriverString
}

var CaptchaDriverString1 = base64Captcha.DriverString{
    Height:              30,
    Width:               60,
    NoiseCount:          0,
    ShowLineOptions: 2,
    Length:              4, // 4个字符验证码
    Source:              "1234567890qwertyuioplkjhgfdsazxcvbnm",
    Fonts: []string{"wqy-microhei.ttc"},
}
```

生成验证码：

这个 `Id` 会自动生成，使用 `Id` 和验证值 `VerifyCode` 对一个验证码图片进行标识和验证。

```
var store = base64Captcha.DefaultMemStore

func GenerateCaptcha(c *gin.Context) {
    captchaCfg := model.CaptchaConfig{
        CaptchaType:    "string",
    }
    captchaCfg.DriverString = &model.CaptchaDriverString1
    capt := base64Captcha.NewCaptcha(captchaCfg.DriverString.ConvertFonts(), store)
    id, b64s, err := capt.Generate()
    if err != nil {
        Failed(c, "生成验证码失败:" + err.Error())
        return
    }
    res := model.CaptchaResult{Id: id,Base64Blob: b64s}
    c.Header("Content-Type", "application/json; charset=utf-8")
    Success(c, map[string]interface{}{"captcha_result": res})
}
```

验证验证码：

```
func VerifyCaptcha(id, verifyCode string) bool {
    if store.Verify(id, verifyCode, true){  // store是上面的store
        return true
    }else {
        return false
    }
}
```

HTTPS

安装支持库：`github.com/unrolled/secure`

启动使用方法`RunTLS()`，并且指定证书：

```
err = app.RunTLS(fmt.Sprintf(":%v", cfg.AppPort), "./cert/ssl.pem", "./cert/ssl.key")
```

使用中间件：

```
// TLSHandler https handler
func TLSHandler() gin.HandlerFunc {
    return func(c *gin.Context) {
         cfg := utils.GetConfig()
         if cfg.IfHttps { // 判断是否使用https
             secureMiddleware := secure.New(secure.Options{
                  SSLRedirect: true,
                  SSLHost:       fmt.Sprintf("%v:%v", cfg.AppHost, cfg.AppPort),
             })
             err := secureMiddleware.Process(c.Writer, c.Request)
             // If there was an error, do not continue.
             if err != nil {
                  log.Println(err.Error())
                  return
             }
             c.Next()
        }

    }
}
```

Redis

安装支持库：`go get github.com/go-redis/redis/v8`

数据库操作

安装mysql驱动：`go get github.com/go-sql-driver/mysql`

执行使用`Exec()`，查询使用`Query()`

```
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/go-sql-driver/mysql"  // 启用init函数
    "log"
)

func main() {
    conn := "root:@tcp(127.0.0.1:3306)/gosqldemo"   // 冒号后面跟密码
    db, err := sql.Open("mysql", conn)  // 连接数据库，设置驱动
    if err != nil {
        log.Fatal("DB Connect Error")
    }
```

```
    fmt.Println("DB connect Successfully")

    // create table
    _, err = db.Exec("CREATE TABLE person(" +
        "id int auto_increment primary key," +
        "name varchar(128) not null," +
        "age int not null default 20" +
        ")charset=utf8;")
    if err != nil {
        fmt.Println("创建失败或者表已存在", err.Error())
    }else{
        fmt.Println("Create table successfully")
    }

    // Insert
    res, err := db.Exec("Insert into person values (null, ?, ?)", "kk", 18)
    if err != nil {
        return
    }
    a, _ := res.RowsAffected()
    fmt.Println("已经插入记录数：", a)

    // Query
    rows, err := db.Query("SELECT * from person")
    if err != nil {
        return
    }

    type person struct {
        id int
        name string
        age int
    }

    for rows.Next(){    // for循环读取数据
        var p person
        err := rows.Scan(&p.id, &p.name, &p.age)
        if err != nil {
            return
        }
        fmt.Println(p)
    }

    // Close connection
    _ = db.Close()
}
```

XORM使用

> 有个缺陷就是不能使用外键

安装：`go get xorm.io/xorm`

然后安装对应数据库的驱动：比如mysql：`go get github.com/go-sql-driver/mysql`

创建引擎：

```
engine, _ := xorm.NewEngine(db.Driver, "root:123@tcp(127.0.0.1:3306)/db?charset=utf8")
```

测试数据库是否连接：

```
err := engine.Ping()
if err != nil {
      fmt.Println("数据库未连接")
      return nil, err
}
```

是否在测试的时候展示SQL语句：

```
engine.ShowSQL(db.ShowSQL)  // 表示是否在运行的时候展示SQL语句
```

创建一个数据表，使用 `Sync2(struct)` ：

表名为结构体名字，字段类型根据 `xorm:""` 内容指定， `pk` 表示主键， `autoincr` 表示自增

```
type SmsCode struct {
    Id             int64   `xorm:"pk autoincr" json:"id"`
    Phone          string `xorm:"varchar(11)" json:"phone"`
    BizId          string `xorm:"varchar(30)" json:"biz_id"`
    Code           string `xorm:"varchar(6)" json:"code"`
    CreateTime int64   `xorm:"bigint" json:"create_time"`
}
// ================================================
// 将一个结构体映射到一个数据表，相当于创建一个表
err = engine.Sync2(new(model.SmsCode))
if err != nil {
      fmt.Println("创建表失败", err)
      return nil, err
}
```

插入一条记录：

```
//使用InsertOne函数
func (d MemberDao) InsertCode(sms model.SmsCode) int64 {
    res, err := d.InsertOne(&sms)
    if err != nil {
        panic(err)
    }
    return res
}
```

查询记录：

> 根据函数 `Where()` 来进行查询，使用 `Get()` 函数来对数据进行获取传给 `m` 。

```
// 会员数据结构定义
type Member struct {
    Id               int64     `xorm:"pk autoincr" json:"id"`
    Username       string    `xorm:"varchar(20) notnull" json:"username"`
    Phone            string    `xorm:"varchar(11) notnull" json:"phone"`
    Password       string    `xorm:"varchar(255)" json:"password"`
    RegisterTime int64     `xorm:"bigint notnull" json:"register_time"`
    Avatar          string    `xorm:"varchar(255)" json:"avatar"`
    Balance         float64 `xorm:"double" json:"balance"`
    IsActive        int8      `xorm:"tinyint" json:"is_active"`
```

```
    City            string    `xorm:"varchar(10)" json:"city"`
}
var m model.Member
f, err := d.Where("phone=?", phone).Get(&m)
```

Update操作：

`Where("id=?", coll...).Update(&obj)`

```
func (d *MemberDao) UploadUserAviator(id int64, filename string) int64 {
    m := model.Member{Aviator: filename}
    res, err := d.Where("id = ?", id).Update(&m)
    if err != nil {
        fmt.Println("图片更新失败"+err.Error())
        return 0
    }
    return res
}
```

GORM使用

安装：

```
go get gorm.io/gorm
go get gorm.io/driver/mysql
```

连接数据库：

> gorm需要进行配置`gorm.config`：log的配置`logger2`、数据库命名的配置`NamingStrategy`、查询的配置`QueryFields`。

在一般情况下，根据logger级别可以选取**默认**等级也可以自己配置（例子中是自己配置的）：

- Debug模式可以选择：`logger2.Default.LogMode(logger2.Info)`
- Release模式可以选择：`logger2.Default.LogMode(logger2.Error)`

```
func connectDb() {
    dsn := "root:password@tcp(localhost:3306)/demo?charset=utf8mb4"

    newlogger := logger2.New(                                   // 自定义配置logger
        log.New(os.Stdout, "\r\n", log.LstdFlags),
        logger2.Config{
            SlowThreshold:                  0,                  // 超过某个毫秒阈值就打印SQL语句
            Colorful:                      true,               // 是否显示颜色
            IgnoreRecordNotFoundError: true,
            LogLevel:                      logger2.Info, // Info Warning Error Silent
        },
    )

    nameStra := schema.NamingStrategy{                          // 命名策略
        TablePrefix:    "",
        SingularTable: true,                                    // 是否为单数
    }

    db, err := gorm.Open(mysql.Open(dsn), &gorm.Config{
        Logger:              newlogger,
        NamingStrategy: nameStra,
```

```
        QueryFields:      true,       // 查询时候显示所有字段，而不是select *
    })
    if err != nil {
        log.Fatalf("Conn Error: %v\n", err)      // 数据库连接失败，端口错误，密码错误，网络原因
    }
    if db != nil && db.Error != nil {
        log.Printf("==Error==: %v\n", db.Error)
    }

    log.Println("Connect Success.")
}
```

模型构建

参考模型关键字：**模型定义 | GORM**

- 定义的方式要使用 `gorm:""` 包围
- 对于可能为零值（`0`，`false`，`''`）需要使用指针的形式进行存储。
- 对于外键，需要在结构体内嵌套对应表，字段名为 `外键表名+对应字段名` 。如 `CustomerID` 。

```
type Merchant struct {
    // 主键 自增，这里设置ID为int类型，gorm会自动转为bigint类型
    ID          int     `gorm:"type:int;primaryKey;autoIncrement"`
    // 设置索引
    Name        string  `gorm:"type:varchar(126);not null;index"`
    // 设置check约束
    Age         byte    `gorm:"type:tinyint;not null;check:age > 12"`
    Address     string  `gorm:"type:varchar(255)"`
    // 设置默认值，可能为零值，只用指针存储
    IsCredit    *bool   `gorm:"type:bool;default:true;not null"`
    // 使用autoCreateTime，在插入记录的时候，gorm会自动补全时间戳
    RegisterTime int64  `gorm:"autoCreateTime;not null"`
}

type Customer struct {
    ID          int       `gorm:"type:int;primaryKey;autoIncrement"`
    Nickname    string    `gorm:"type:varchar(255);not null;index"`
    Age         byte      `gorm:"type:tinyint;not null;check:age> 12"`
    RecvAddress string    `gorm:"type:varchar(255)"`
    Credit  float64 `gorm:"type:float(3);not null; default:0; check credit > 0"`
    RegisterTime int64    `gorm:"autoCreateTime;not null"`
}

type Order struct {
    // 设置备注
    ID      int         `gorm:"type:int;primaryKey;autoIncrement;comment:order id"`
    MerchantID   int         `gorm:"not null"`
    CustomerID   int         `gorm:"not null"`
    ProductName string    `gorm:"type:varchar(255);not null;"`
    // 定义外键
    Customer      Customer `gorm:"constraint"`
    Merchant      Merchant `gorm:"constraint"`
}
```

自动迁移表：

> 相当于xorm中的 `sync2()`

```go
func TestCreateTable(t *testing.T) {
    connectDb()
    // 使用上面的结构体
    err := Orm.DB.AutoMigrate(&model.Merchant{}, &model.Customer{}, &model.Order{})
    if err != nil {
        log.Fatal(err)
    }
    log.Println("Auto migrate successfully.")
}
```

增（insert）

> 支持插入单条记录和多条记录

```go
func TestInsert(t *testing.T) {
    connectDb()
    b := t == nil
    user := model.Merchant{
        Name:       "ganster",
        Age:          18,
        Address:   "Beijing",
        IsCredit:   &b,
    }
    // INSERT INTO `merchant` (`name`,...,`register_time`) VALUES ('ganster',...,1628161718)
    re := Orm.DB.Create(&user)
    if re.Error != nil {
        log.Fatal(re.Error)
    }
    fmt.Println(user.ID, re.RowsAffected)
}

func TestInsertBatches(t *testing.T) {
    connectDb()
    var grp []model.Merchant
    /*
    INSERT INTO `merchant` (`name`,...,`register_time`) VALUES ('ganster',...,1628161718),
    ('ganster2',...,1628161718),('ganster3',...,1628161718),('ganster4',...,1628161718);
    */
    for i := 0; i < 10; i++ {
        b := i%2==0
        mer := model.Merchant{
            Name:       fmt.Sprintf("test_%c", 'A'+i),
            Age:          byte(i + 20),
            Address:   fmt.Sprintf("Addr_%c", 'a'+i),
            IsCredit: &b,
        }
        grp = append(grp, mer)
    }
    re := Orm.DB.CreateInBatches(grp, len(grp))
    log.Printf("%v\n", re.RowsAffected)
}
```

删(delete)

```
func TestDel(t *testing.T) {
    connectDb()
    // DELETE FROM `merchant` WHERE name = 'john'
    re := Orm.DB.Delete(&model.Merchant{}, "name = ?", "john")
    log.Printf("Rows: %v\n", re.RowsAffected)
}
```

改（update）

`Update()` 更新单列，`Updates()` 更新多列

```
func TestUpdate(t *testing.T) {
    connectDb()
    b := false
    u := model.Merchant{
        Name:       "john",
        IsCredit:   &b,
    }
    // 更新单列
    re := Orm.DB.Model(&u).Where("name = ?", "john").Update("name", u.Name)
    log.Printf("Rows:%v, Error:%v\n", re.RowsAffected, re.Error)

    // 更新多列
    re = Orm.DB.Model(&u).Where("name = ?", "johny").Updates(&u)
    log.Printf("2nd: Rows:%v, Error:%v\n", re.RowsAffected, re.Error)

    // 选取字段更新，只更新name字段
    Orm.DB.Model(&u).Select("name").Where("name = ?", "johny").Updates(&u)
}
```

查:

`First()` `Last()` 根据主键排序后取第一个或者最后一个记录。

`Take()` 取出一个记录

`Find()` 找到所有的记录

```
func TestQuery(t *testing.T) {
    connectDb()
    // 查询单条记录
    var u model.Merchant
    type APIUser struct {
        Name string
        Age  int
    }
    re := Orm.DB.First(&u, "name = ?", "ganster")
    log.Printf("Get one: %v\n", u)
    log.Printf("%v\n", re.RowsAffected)
    // 查询多条记录
    var us []model.Merchant
    // SELECT name, age FROM `merchant` WHERE name regexp 'jo+'
    re = Orm.DB.Select("name, age").Find(&us, "name regexp ?", "jo+")
    log.Printf("Get all %v\n", us)
    log.Printf("%v\n", re.RowsAffected)
    // 另一种
    var uas []APIUser
    re2 := Orm.DB.Model(&model.Merchant{}).Find(&uas, "name regexp ?", "jo+")
    log.Printf("Get all 2:%v\n", uas)
```

```
    log.Printf("%v\n", re2.RowsAffected)
}
```

🔘错误处理：

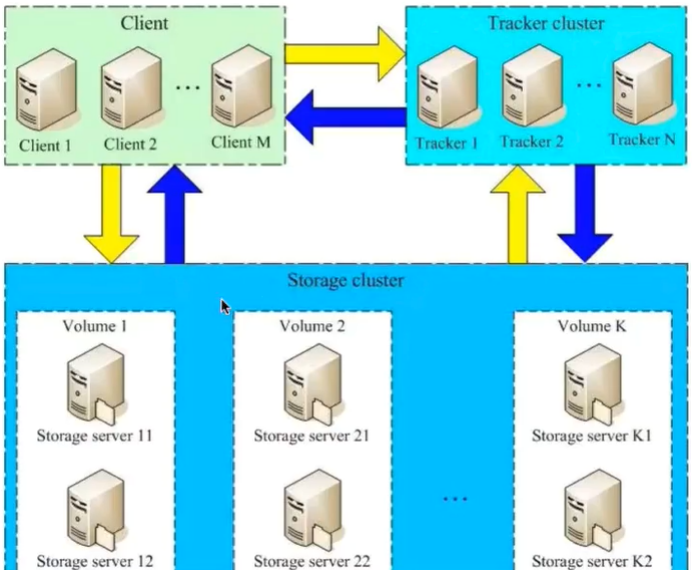
```
if err := db.Where("name = ?", "jinzhu").First(&user).Error; err != nil {
   // 处理错误...
}
```

分布式文件系统FastDFS

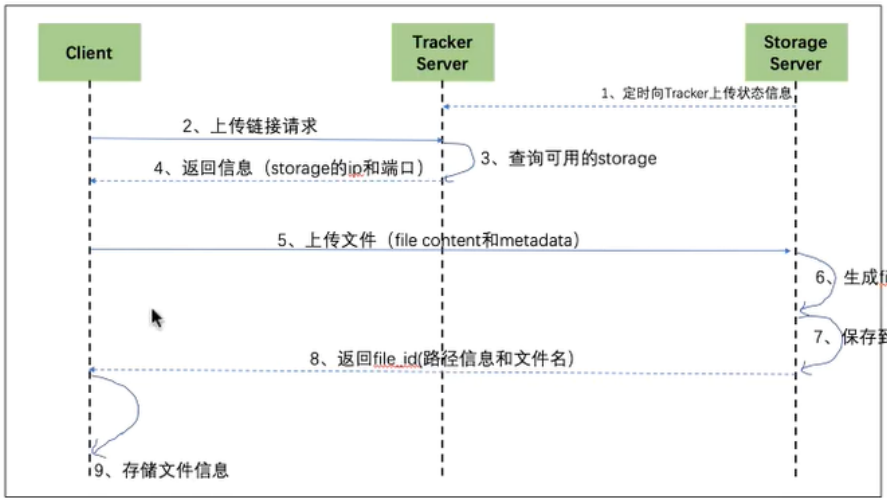
FastDFS分为三个角色：跟踪服务器(Tracker Server)，存储服务器(Storage Server)和客户端(Client)

每个角色组件都有各自的作用和功能:

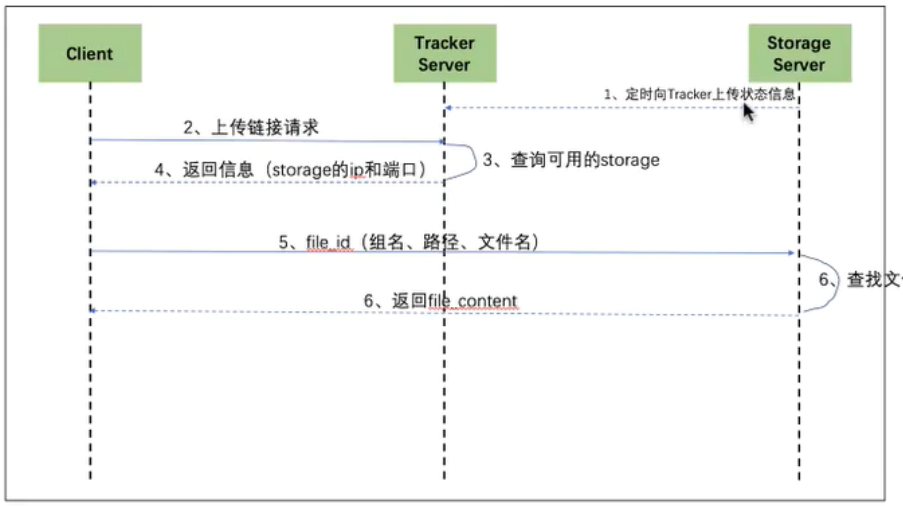
- 存储服务器:即Storage Server。存储服务器顾名思义就是用于存储数据的服务器，主要提供存储容量和备份服务。存储服务器为了保证数据安全不丢失，会多台服务器组成一个组，又称group。**同一个组内的服务器上的数据互为备份。**
- 跟踪服务器:英文称之为Tracker Server。跟踪服务器的主要作用是做调度工作，担负起均衡的作用;跟踪服务器主要负责管理所有的存储服务器，以及服务器的分组情况。存储服务器启动后会与跟踪服务器保持链接,并进行周期性信息同步。
- 客户端:主要负责上传和下载文件数据。客户端所部署的机器就是实际项目部署所在的机器。



FastDFS文件上传和下载过程



文件下载：



环境搭建

1. 首先进行fastdfs安装配置和服务启动
    - 下载 **libfastcommon** 库

```
wget https://github.com/happyfish100/libfastcommon/archive/refs/tags/V1.0.52.tar.gz
```

    - 解压：

```
tar -zxvf V1.0.52.tar.gz
```

    - 编译并且安装：

```
cd ibfastcommon-1.0.52
```

编译：`./make.sh`

安装：`./make.sh install`

    - 下载 **fastdfs** 文件：

```
wget https://github.com/happyfish100/fastdfs/archive/refs/tags/V6.07.tar.gz
```

同样进行编译安装。

    - 配置Tracker服务器：

进入`/etc/fdfs`目录，发现四个模板文件，将后缀名去掉

```
[root@VM-16-14-centos fdfs]# ll
total 32
-rw-r--r-- 1 root root  1909 Jul  1 19:00 client.conf.sample
-rw-r--r-- 1 root root 10246 Jul  1 19:00 storage.conf.sample
-rw-r--r-- 1 root root   620 Jul  1 19:00 storage_ids.conf.sample
-rw-r--r-- 1 root root  9138 Jul  1 19:00 tracker.conf.sample
```

2. 配置Nginx服务器模块

gRPC

BV1Xv411t7h5

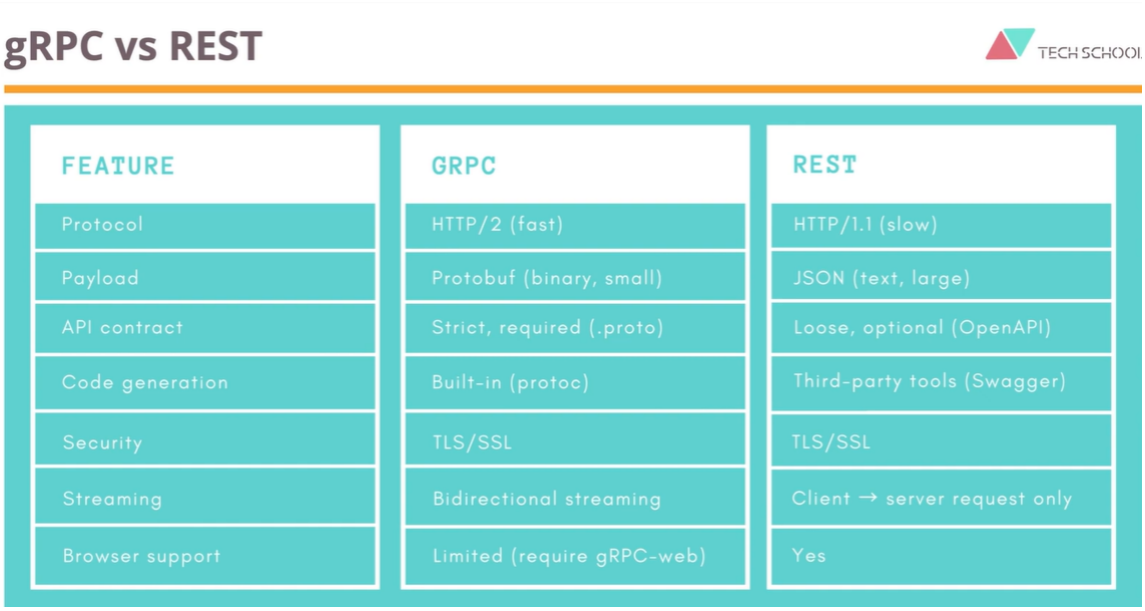
BV1GE411A7kp

支持多种语言之间的信息交互。

gRPC使用http2的优点：

- 使用二进制框架：
  - 高性能；更轻便安全；与protocal buffer更好结合
- 使用HPACK压缩头
- 可以支持多路复用
  - 在单个TCP连接中可以并行发送多个请求和接受多个响应。
  - 低延迟
- 单个用户请求，多个服务器响应，减少延迟。

查看http2样例：**http://http2demo.io/**



因此，微服务才是grpc使用的主场，其可以实现低延迟和高吞吐量的通信。

gRPC的四种数据类型

- unary：客户端发送一个请求，服务器端+回复一个请求。类似普通的HTTP API
- client streaming：客户端发送多个信息流，服务器值返回一个回复。
- server streaming：客户端发送一个请求，服务器值返回多个信息流。
- bidirectional streaming：双方无阻塞的互相发送信息流。

安装并且编译代码

1. 首先需要`protoc`来编译`.proto`文件，并且添加环境变量：

   win: **github 下载界面**

   linux: `apt install -y protobuf-compiler`

2. go获取包：

   ```
   go get -u google.golang.org/grpc
   go get -u google.golang.org/grpc/cmd/protoc-gen-go-grpc
   go get -u github.com/golang/protobuf/protoc-gen-go
   ```

3. 编译：

   在编译之前需要在对应文件中加入：`option go_package = "/path";`一句话表示安装的路径

   ```
   protoc -I=$proto_path --go_out=./ --go-grpc_out=./ $proto_path/*.proto
   ```

proto—message类型

```
syntax = "proto3";

package tech.pcbook;

option go_package = "/pb";

message Screen{
   message Resolution{
      uint32 width = 1;
      uint32 height = 2;
  }

   enum Panel{
      UNKNOWN = 0;
      IPS = 1;
      OLED = 2;
  }

   float size_inch = 1;
   Resolution resolution = 2;    // 自定义类型
   Panel panel = 3;
   bool multitouch = 4;
   repeated GPU gpus = 5;        // 数组类型
   repeated Storage storages = 6;
   oneof weight{
      double weight_kg = 7;
      double weight_lb = 8;
  }
   google.protobuf.Timestamp update_at = 9;  // 时间戳类型，使用timestamppb.Now()
}
```

protobuf序列化与反序列化

> 序列化成二进制文件大小要比JSON格式的文档要小很多，因此传输更加方便。

信息的类型为 `proto.Message` 类型

将信息序列化到二进制文件中：

```
func WritePB2Binary(msg proto.Message, filename string) error {
    data, err := proto.Marshal(msg)
    if err != nil {
        return err
    }

    err = ioutil.WriteFile(filename, data, 0644)
    if err != nil {
        return err
    }
    return nil
}
```

从二进制文件中反序列化：

```
func ReadPBFromBin(filename string, msg proto.Message) error {
    file, err := ioutil.ReadFile(filename)
    if err != nil {
        return err
    }
    err = proto.Unmarshal(file, msg)
    if err != nil {
        return err
    }
    return nil
}
```

将信息序列化到JSON格式中：

```
func WritePB2JSON(msg proto.Message, filename string) error {
    json, err := PB2JSON(msg)
    if err != nil {
        return err
    }
    err = ioutil.WriteFile(filename, []byte(json), 0644)
    if err != nil {
        return err
    }
    return nil
}

func PB2JSON(msg proto.Message)(string, error){
    marshaler := jsonpb.Marshaler{
        OrigName:       true,     // 是否使用proto源文件中的名称
        EnumsAsInts:   true, // 是否设置枚举类型为数字或者字符串
        EmitDefaults: true,     // 是否设置默认初始值
        Indent:           "\t",     // 格式缩进
    }
```

```
    return marshaler.MarshalToString(msg)
}
```

Unary

> 客户端发送一个请求，服务器端回复一个请求。类似普通的HTTP API

样例：

proto文件：

```
syntax = "proto3";
package tech.pcbook;
option go_package = "/pb";

import "laptop_message.proto";

message CreateLaptopRequest{
   Laptop laptop = 1;
}

message CreateLaptopResponse{
   string id = 1;
}

service CreateLaptopService{
   rpc CreateLaptop(CreateLaptopRequest) returns (CreateLaptopResponse){};
}
```

经过 `protoc` 文件编译后会生成两个文件：一个是message文件 `laptop_service.pb.go` ，一个是服务文件 `laptop_service_grpc.pb.go` 。在grpc文件中存在两个接口，分别是 `XXXClient` 和 `XXXServer` 的interface需要去实现。

实现rpc服务接口：

```
type LaptopServer struct {
    pb.UnimplementedCreateLaptopServiceServer   // 由于兼容性，必须要包含
    // 其他自定义
    Store LaptopStore
}

// 实现接口一
func (s *LaptopServer) CreateLaptop(ctx context.Context, req *pb.CreateLaptopRequest)
(*pb.CreateLaptopResponse, error) {
    laptop := req.GetLaptop()   // 根据proto文件中的rpc中的参数获取
    if len(laptop.Id) > 0 {
        log.Printf("Get laptop:%v", laptop)
        _, err := uuid.Parse(laptop.Id)
        if err != nil {
            return nil, err
        }
        err = s.Store.Save(laptop)
        if err != nil {
            return nil, err
        }
    }
```

```
    res := &pb.CreateLaptopResponse{    // 创建响应
        Id: laptop.Id,
    }
    return res, nil
}

func NewLaptopServer(store LaptopStore) *LaptopServer {
    return &LaptopServer{Store: store}
}
```

开启服务端：

```
func main() {
    port := flag.Int("port", 0, "The Server Port")  // 命令行参数
    flag.Parse()
    log.Printf("Start Server at port: %v\n", *port)

      // 1. 初始化grpc服务器
    grpcServer := grpc.NewServer()
      // 2. 实现protobuf的Server对象
      laptopServer := service.NewLaptopServer(service.NewInMemoryLaptopStore())
      // 3. 注册绑定二者
    pb.RegisterCreateLaptopServiceServer(grpcServer, laptopServer)
    addr := fmt.Sprintf("192.168.1.7:%v", *port)
    fmt.Println(addr)
      // 4. 开始监听
    listener, err := net.Listen("tcp", addr)
    if err != nil {
        log.Fatal("Cannot start server 1,", err)
    }
      // 5. 创建服务
    err = grpcServer.Serve(listener)
    if err != nil {
        log.Fatal("Cannot start server 2,", err)
    }
}
```

开启客户端：

```
func main() {
    serverAddr := "192.168.1.7:8080"
      // 1. 连接服务器
    conn, err := grpc.Dial(serverAddr)
    if err != nil {
        return
    }
    // 2. 创建rpc的client
    lpClient := pb.NewCreateLaptopServiceClient(conn)
      // 3. 构造请求参数
    laptop := sample.NewLaptop()
    req := pb.CreateLaptopRequest{Laptop: laptop}
      // 4. 发送请求
    res, err := lpClient.CreateLaptop(context.Background(), &req)
    if err != nil {
        fmt.Println(err.Error())
        return
    }
    fmt.Println(res)
```

```
}
```

其他：

本项目中用到的其他包：

```
# uuid generator
go get github.com/google/uuid
# deep copy
go get github.com/jinzhu/copier
# test
go get github.com/stretchr/testify
```

Server Streaming

> 客户端发送一个请求，服务器端回复多个信息流

定义rpc：

```
service LaptopService{
   // stream
   rpc SearchLaptop(SearchLaptopRequest) returns (stream SearchLaptopResponse){};
}
```

实现对应的grpc接口：（`stream.Send(response)`）

```
func (s *InMemoryLaptopStore) Search(filter *pb.Filter, found func(lp *pb.Laptop) error) error {
    s.mutex.RLock()
    defer s.mutex.RUnlock()

    for _, lp := range s.data {
        if filter.MaxPriceUsd > lp.PriceUsd && filter.MinCpuGhz < lp.Cpu.MinGhz && filter.MinCpuCores <
lp.Cpu.NumberCores && filter.MinRam.Value < lp.Ram.Value {
            other := &pb.Laptop{}
            err := copier.Copy(other, lp)
            if err != nil {
                return err
            }

            err = found(other)  // 在循环内一直发送
            if err != nil {
                return err
            }
        }
    }
    return nil
}

// 实现接口
func (s *LaptopServer) SearchLaptop(req *pb.SearchLaptopRequest, stream
pb.LaptopService_SearchLaptopServer) error {
    filter := req.GetFilter()
    log.Printf("Recv a search laptop req, filter : %v\n", filter)

    err := s.Store.Search(filter, func(lp *pb.Laptop) error {
        rsp := &pb.SearchLaptopResponse{Laptop: lp}
        err := stream.Send(rsp) // 使用此方法发送流
```

```
        if err != nil {
            return err
        }
        log.Printf("We have send a search laptop ID:%v\n", lp.Id)
        return nil
    })
    if err != nil {
        return err
    }
    return nil
}
```

客户端读取：

```
func main(){
      // 连接
      serverAddr := "127.0.0.1:8080"
    conn, err := grpc.Dial(serverAddr, grpc.WithInsecure())
    if err != nil {
        log.Fatal("Dial failed", err)
    }
      // grpc客户端绑定连接
    lpClient := pb.NewLaptopServiceClient(conn)
      // 构造参数
      req2 := pb.SearchLaptopRequest{Filter: filter}
      // 获取stream
      stream, err := lpClient.SearchLaptop(context.Background(), &req2)
      if err != nil {
            return
     }

      i := 1
      for true {
            // 循环获取
            rsp, err := stream.Recv()   // 阻塞
            if err != nil {
                  if err == io.EOF{   // 末尾推出
                        break
                 }
                  log.Fatal(err)
                  return
           }
            log.Printf("%v: stream recv: %v\n", i, rsp.GetLaptop().Name)
            i++
     }
}
```

Client Streaming

> 从某个文件中读取陆续上传

proto文件：

```
message UploadImgReq{
   oneof data{
      ImageInfo info = 1;
      bytes chunk_data = 2;
```

```
  }
}

message ImageInfo{
   string laptop_id = 1;
   string image_type = 2;
}

message UploadImgResp{
   string id = 1;
   uint32 size = 2;
}

service LaptopService{
   rpc UploadImage(stream UploadImgReq) returns (UploadImgResp){};
   // other
}
```

grpc实现接口：`Recv()` `SendAndClose()`

```
func (s *LaptopServer) UploadImage(stream pb.LaptopService_UploadImageServer) error {
    req, err := stream.Recv()   // 第一次接受图片名字和类型
    if err != nil {
        log.Println(err)
        return err
    }
    id := req.GetInfo().GetLaptopId()
    imageType := req.GetInfo().GetImageType()
    log.Printf("Recv img info %v.%v\n", id, imageType)
    imageData := bytes.Buffer{}
    imageSize := 0

    for {
        req, err := stream.Recv()   // 后面n次接受图片数据
        if err == io.EOF {
            log.Printf("%v\n", "Recv img Over")
            break
        }
        if err != nil {
            return err
        }
        chunk := req.GetChunkData()
        size := len(chunk)
        imageSize += size

        imageData.Write(chunk)
    }

    imgId, err := s.ImgStore.Save(id, imageType, imageData) // 保存接口
    if err != nil {
        return err
    }

    rsp := &pb.UploadImgResp{   // 构造response
        Id:    imgId,
        Size: uint32(imageSize),
    }
```

```
    err = stream.SendAndClose(rsp)  // 发送关闭
    log.Printf("Saved img, id:%v, size:%v\n", imgId, imageSize)
    return nil
}
```

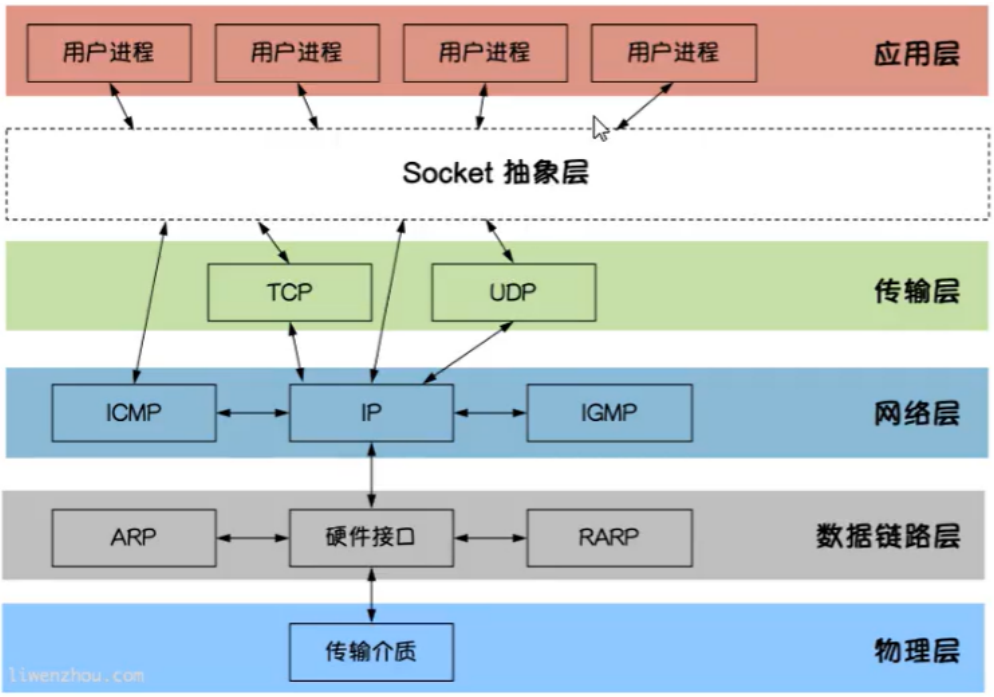
Bidireactional Streaming

类似与client/server streaming的结合

Golang网络编程

http官方文档

golang常用的http请求操作



TCP编程

> 简单样例，复杂网络编程详见IO多路复用

服务器端：

```
func main() {
    // 创建连接
    listener, err := net.Listen("tcp", "127.0.0.1:20000")
    if err != nil { log.Fatal("Start Server err:", err) }

    // 等待连接
    log.Println("Server is listening.")
    conn, err := listener.Accept()
    if err != nil { log.Fatal("Listener start err:", err)}
    var msg [128]byte
      // 读取数据
    n, err := conn.Read(msg[:])
```

```
    if err != nil { log.Fatal("Read msg err:", err) }
    log.Printf("Read from client msg: %v\n", string(msg[:n]))

    err = conn.Close()
    if err != nil { log.Fatal("Close Server error: ", err) }

}
```

客户端：

```
func main() {
    conn, err := net.Dial("tcp", "127.0.0.1:20000")
      if err != nil { log.Fatal("Dial Server failed: ", err) }

    _, err = conn.Write([]byte("Hello TCP"))
    if err != nil { log.Fatal("Send Message failed: ", err) }

    err = conn.Close()
    if err != nil { log.Fatal("Close client failed: ", err) }

}
```

TCP粘包问题

> 粘包：发送端为了将多个发往接收端的包，更加高效的的发给接收端，于是采用了优化算法（Nagle算法），将多次间隔较小、数据量较小的数据，合并成一个数据量大的数据块，然后进行封包。

造成TCP粘包的原因：

- 发送方：Nagel算法，较小的数据包合并发送。
- 接收方：TCP接受到数据包的时候不马上交给应用层处理，TCP接受的包会保存到缓存里；如果TCP接收方收到缓存的速度要比应用层从缓存中读取的速度要快，就会产生粘包的问题。

什么时候需要处理粘包问题：

- 如果同一个文件分组发送，就不用处理
- 如果多个分组数据毫不相关，就一定要处理粘包问题了。

怎么处理粘包问题：

- 发送方：
- 接收方：1. 使用分隔符协议：`reader.ReadSlice('\n')` 2. 使用自定义长度协议，每个数据前n位表示长度。

UDP通信编程

服务器端：

```
func main() {
    conn, err := net.ListenUDP("udp", &net.UDPAddr{ // 监听端口
        IP:    net.IPv4(127, 0, 0, 1),
        Port: 8888,
    })
    if err != nil {
        log.Fatal("Udp Listen error: ", err)
    }
```

```
    log.Printf("Start listening\n")
    var msg [128]byte
    for {
            // 读取信息
        n, addr, err := conn.ReadFromUDP(msg[:])
        if err != nil {
            log.Fatal("Read from udp error: ", err)
        }
        log.Printf("Data from %v msg: %v\n", addr.String(), string(msg[:n]))
             // 回复客户端信息
        n, err = conn.WriteToUDP([]byte("Recv OK\n"), addr)
        if err != nil {
            log.Fatal("Send udp error: ", err)
        }
    }
}
```

客户端:

```
func main() {
      // 连接UDP服务器
    conn, err := net.DialUDP("udp", nil, &net.UDPAddr{
        IP:    net.IPv4(127, 0, 0, 1),
        Port: 8888,
    })
    if err != nil {
        log.Fatal(err)
        return
    }
      // 直接发送信息
    _, err = conn.Write([]byte("ABS"))
    if err != nil {
        log.Fatal(err)
    }
    var msg [128]byte
      // 获取有服务器发来的信息
    n, _, err := conn.ReadFromUDP(msg[:])
    if err != nil {
        return
    }
    log.Printf("Recv from server: %v\n", string(msg[:n]))
}
```

获取自己出口IP地址

```
func getMyIP() (ip string){
    conn, err := net.Dial("udp", "210.2.1.6:80")    // 连接对应的IP
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()
    localAddr := conn.LocalAddr().(*net.UDPAddr)
    return localAddr.IP.String()
}
```

Cobra

> 快速生成命令行界面

链接： **spf13/cobra**

简单介绍

cobra指令分为两种，一种是command，一种是flag。

形式为：

```
demo.exe test --account jack
```

其中 `test` 为command， `--account` 为flag， `jack` 为参数。

定义指令

根指令是由 `cobra.Command` 结构体创建而成。

- `Use` 表示使用输入的具体命令，
- `Short` 表示对整体程序使用 `demo.exe -h` 时，指令后面的说明文字
- `Long` 表示对该指令使用 `demo.exe main -h` 时候，所展示的具体文字。
- `Run` 表示运行此指令的时候执行的函数。
- `TraverseChildren` 设置为 `true` 表示可以在主程序上使用flag，默认不可以在主程序后面使用flag，比如 `demo.exe -name jack` 。
- `AddCommand(...*Command)` ，指令之间可以进行嵌套，添加子指令
- `cobra.MousetrapHelpText = ""` ，可以在直接点击exe程序的时候运行，cobra默认设置不可以直接运行，必须采用命令行的形式运行。参考： **cobra package**

```
var rootCmd = &cobra.Command{
    Use:                   "main",
    Short:                 "Short text for program",
    Long:                  "Illustration of AirJ Login program",
    TraverseChildren: true, // 允许在主命令上使用flag
    Run: func(cmd *cobra.Command, args []string) {
        acc, pass := service.ReadLoginData() // 获取配置文件
        if len(acc) > 0 && len(pass) > 0 {
            service.Login(acc, pass)
        } else {
            log.Println("Account parameter error.")
        }
        time.Sleep(time.Second * 2)
    },
}

func Execute() {    // 在main程序中调用即可
    if err := rootCmd.Execute(); err != nil {
         log.Println("Input command format error.")
         os.Exit(1)
    }
}

func init() {
    rootCmd.AddCommand(versionCmd, changeCmd, listCmd, delCmd)
```

```
    cobra.MousetrapHelpText = ""
}
```

定义flag标志

标志的类型可以分为两种：一种是持久性标志`PersistentFlags()`，一种是局部标志`Flags()`。持久性标志可以在所有的command下指定，局部表示只能在当前command下指定。

`String()`：指令名称无短名称，返回String类型

`StringP()`：指令名称有短名称，返回String类型

`StringVarP()`：将数据返回到对应变量，可以指定短名称，返回String类型

`MarkFlagRequired()`：将对应的指令标识为**必需**参数。

其他类型变量类似

```
var changeCmd = &cobra.Command{
    Use:   "change",
    Short: "Change AirJ account",
    Long:  "Change a new account of airJ, then login with this account, if this account exists, then change to this.",
    Run: func(cmd *cobra.Command, args []string) {
        // 业务逻辑
    },
}

func init() {
    changeCmd.Flags().StringVarP(&account, "account", "a", "", "your account (required)")
    changeCmd.Flags().StringVarP(&password, "password", "p", "", "your password")
    _ = changeCmd.MarkFlagRequired("account")
}
```

Viper

> 可以将传入的参数写入到配置文件中，并且下次可读取。

设计模式

设计模式之美 -> 对应资源 -> **BDPan: 密码6666**

Go设计模式24-总结　极客时间对于的go实现

其他

CentOS安装mariadb：

安装：`yum -y install mariadb mariadb-server`

启动：`systemctl start mariadb`

简单配置：`mysql_secure_installation`

配置字符集：`vi /etc/my.cnf`

在`/etc/my.cnf`中配置，在`[mysqld]`标签下添加：

```
init_connect='SET collation_connection = utf8_unicode_ci'
init_connect='SET NAMES utf8'
character-set-server=utf8
collation-server=utf8_unicode_ci
skip-character-set-client-handshake
```

在`/etc/my.cnf.d/client.cnf`中配置，在`[client]`中添加：

```
default-character-set=utf8
```

在`/etc/my.cnf.d/mysql-clients.cnf`配置，在`[mysql]`中添加:

```
default-character-set=utf8
```

重启：`systemctl restart mariadb`

查看配置的字符集：

```
mysql> show variables like "%character%";show variables like "%collation%";
```