## Go

Strconv errors fmt time encoding/json

import 下划线（如：import \_ hello/imp）的作用：当导入一个包时，该包下的文件里所有init()函数都会被执行，然而，有些时候我们并不需要把整个包都导入进来，仅仅是是希望它执行init()函数而已。这个时候就可以使用 import \_ 引用该包。即使用【import \_ 包路径】只是引用该包，仅仅是为了调用init()函数，所以无法通过包名来调用包中的其他函数。

Import 前面是个. 这样的意思就是在调用这个包下面的函数不用加包名字了

如果import前面是个字符串,就是为了避免相同的包名字冲突把这个包给重命名了.

这个是比较快的json

"[github.com/json-iterator/go](http://github.com/json-iterator/go)"

var json\_iterator = jsoniter.ConfigCompatibleWithStandardLibrary

Time.Now().Add(time.Durtation(60) \* time.Second)

这里要说的时候,要操作的时间的时候,要这样把数字强转成time.Duration()类型.

api的路径后面如果少一个/的话就会触发307帮你重定向到正确的地方.

如果一个时间的unix的时间是0,那么他的time对象是0001年0001月01日,并不是0时,但是这个时间由于在1970年之前,那么转出来的时间戳就是个负数的啊.

字符串转时间

t1, err := time.ParseInLocation(“2006-01-02 15:04:05”, 时间的串, time.Local )

一个tasklist append一个东西后,如果触发了扩容,那么地址就变了啊

如果一个普通的结构体你不打上json的tag,那么序列化出来的字段名就是结构体元素的名字.

type Product struct {

    Name      string  `json:"name"`

    ProductID int64   `json:"-"` // 表示不进行序列化

    Number    int     `json:"number"`

    Price     float64 `json:"price"`

IsOnSale  bool    `json:"is\_on\_sale,string"`

指定类型为string,这样这个字段打印出来就是"is\_on\_sale":"false"

而不是这样"IsOnSale":true

ProductID int64   `json:"product\_id,omitempty"`

这样的话就是如果这个字段为空就不会序列化了.

 ProductID int64   `json:"product\_id,string"`

Number    int     `json:"number,string"`

这样的话int的元素就会按照字符串打印,

"product\_id":"10","number":"10000"

strings repalce必要用1.2

import 下划线（如：import \_ hello/imp）的作用：当导入一个包时，该包下的文件里所有init()函数都会被执行，然而，有些时候我们并不需要把整个包都导入进来，仅仅是是希望它执行init()函数而已。这个时候就可以使用 import \_ 引用该包。即使用【import \_ 包路径】只是引用该包，仅仅是为了调用init()函数，所以无法通过包名来调用包中的其他函数。

go语言的字符串是可以直接比较大小的.

match, err := regexp.MatchString("11", "11")

如果你想自定义一个error的话,只需要定义一个结构体实现Error() string这个函数就可以了,结构体的元素随便是啥,一般都是code和msg.因为error本身就是个接口啊.

对于把接口对象转成真正的实现类对象,这样操作就可以了.

If res,ok:= err.(\*xxxError);ok{}

有的时候你用go mod拉一个工程,拉不下来看提示可能是你没有这个工程的权限,但是你确依赖了他. 平时我们依赖的包可以拉下来说明人家是对外开放的.

New(struct)出来是一个指针

给了Maintainer,scm你就可以配置了,这也会影响到.

go的结构体如果你要弄成json的话,结构体名字要大写,后面还要补上json的注释,不然是识别不出来的哦.

json还是比较只能的,多一个少一个的,顺序什么的都无所谓的,它好像就是用key当作map去捞的.

结构体也是可以用%v打印的,万物皆可用%v打印.

json是结构 还是json是字符串是不一样的.

Json unmarl的时候数组和map都必须加上地址!!!!!!!!!!!!!

数组元素的内容是结构体和是指针都是可以的.

如果实现了结构体中的String方法，在使用fmt打印时候会调用该方法，  类似于java中的tostring

结构体会转化为JSON对象，并且只有结构体里边以大写字母开头的可被导出的字段才会

被转化输出，而这些可导出的字段会作为JSON对象的字符串索引,也就是结构体想要可以序列化成json的话,元素必须的是大写的那部分才会被转化.

如果JSON中的字段在Go目标类型中不存在，json.Unmarshal()函数在解码过程中会丢弃 该字段,而不是报错,所以json的转换是根据key去填的,多几个少几个,都不会影响其他字段的解码与编码.

3 转化一个map类型的数据结构时，该数据的类型必须是 map[string]T(T可以是

encoding/json 包支持的任意数据类型

type Transport struct {

// Proxy指定用于针对特定请求返回代理的函数。

// 如果该函数返回一个非空的错误，请求将终止并返回该错误。

// 如果Proxy为空或者返回一个空的URL指针，将不使用代理

Proxy func(\*Request) (\*url.URL, error)

// Dial指定用于创建TCP连接的dail()函数。

// 如果Dial为空，将默认使用net.Dial()函数

Dial func(net, addr string) (c net.Conn, err error)

看到没有,上面的两个元素,就是函数类型,go直接有就有一个函数类型的哦!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

go func() {

defer close(exit)

go func() {

fmt.Println("b")

}()

}()

Go 运行一个不要定义的func是这样子操作的

go中的方法和传统的类的方法不太一样，方法和类并非组织在一起，传统的oop方法和类放在一个文件里面，而go语言只要在同一个包里就可，可分散在不同文件里。go的理念就是数据和实现分离，

go的结构体和属于这个结构体的方法可以在不同的文件里，但是要在同一个包里

把一个interface{}类型的东西转成他真正的类型。

if res, ok := data.TfPlanDH.Data.(\*data.TfPlanData); ok {

if err = conn.Table("tf\_plan").Where("plan\_id =?", tfPlan.PlanId).Updates(tfPlan).Error; err != nil {

  logs.CtxWarn(ctx, "UpdateTfPlan Updates error: %s", err)

  return err

 }这种更新的时候传进来的结构体需要有id的

这种就会像java那样，你的结构体的元素不是空就会去更新，是null的就会保持原来的。

func test(arr \*[]string) (s string){

这样的话，看到了么，返回值列表既定义了一个类型，顺便也创建了这个返回的变量，代码里对s进行复制操作就可以了，但是这种返回方式必须要用小括号扩起来

go func(){fmt.Println("Hello inline")}()

        go printHello()

但是呢，go运行一个用户线程的时候，函数名后面必须是个（）

所以即使第一行的是一个匿名函数，后面也要加一个（）

:= 结构不能使用在函数外，函数外的每个语法块都必须以关键字开始

var (

    a int

    b string

    c []float32

    d func() bool

    e struct {

        x int

    }

)          批量定义变量

go语言很多的那流程操作都是在init里面，如果不看init单撸代码的话可能看不到

channel是一个定义比较简单的携程之间的消息队列啊，因为channel是多个的。

看到了么，函数指针的执行。

        if (eventActFun)

        {

            eventActFun();

声明时的\*eventActFun的\*只是声明作用，下面用的时候还是要用eventActFun

返回值（\*函数名）（参数列表）

是在函数名前面顶了一个\*，这个就是函数指针。

var ptr \*\*int; 指针的指针

go的携程是用户线程。

numbers = append(numbers, 2,3,4) 还可以追加切片和数组。

slice1 := make([]type, len) len是切片的初始i长度，指定0就可以，我们可以往上在append。

/\* 拷贝 numbers 的内容到 numbers1 \*/

copy(numbers1,numbers) 切片复制

range就是foreach呗

**那种既不是大写也没有指定结构体的方法就是包内可以调用，那种大写的是包名调用（这种是类似与静态）。 而实现接口的方法或者结构体自己的方法就是结构体指针调用。**

**包名调用的还有就是那种文件var定义的全局变量，还有就是包下的结构体信息，其他包想创建这个包定义的结构体的时候。**

**还有就是如果一个结构体的元素名字不是大写的,但是结构体名字是大写的,那么出了这个包后其他包即使创建了这个结构体(因为结构体的名字是大写的),但是也是无法访问这个结构体的元素的,因为结构体的名字是小写的.**



go的依赖会先去vendor里找，然后在去gopath里面找。

Go的数组会如果长度超了会扩容和java的listy差不多，关键的是lsit的扩容是list对象里面的buff变成新的，list指针不会变，而go的扩容后数组的地址都变了！！！！

注意，init的顺序，上面的还没跑到他的init函数，你就去调用他。

Fmt.printf(“%p”)就是按照指针打印

经过验证,那种string init的啥遍历的时候指针是同一个.

经过验证,那种结构体数组在遍历的时候指针也是同一个,对于结构体指针数组在遍历的时候指针就不是同一个了.

还有就是在遍历数组的数组的时候,其中的一个item就是一个指针了,不要在加&这东西了.

Fmt.Printf(“%T”)是打印类型,如果你还要加个&的话,就变成数组的指针了,以后就把数组当作指针来看了.

对于那种创建一个数组并塞一个元素并赋值的代码,不要那样先创建一个切片,在给切片append然后再赋值,而是要xxx = []\*model.xxx{ aaa } 这样就可以了

结构体切片的根据结构体的某一个元素排序代码

sort.Slice(results, func(i, j int) bool {

        return results[i].Value < results[j].Value

})

这里就是根据Value元素的大小排序results这个数组.

## go代码

抽象能力.

对于从数据库的查的数据,一定要把两个返回值都判断下,不要只判断error,因为err不为null,但是数据可能是有问题的,数据库嘛.

数据耦合,而不是参数耦合,不是程序耦合,有没有发现经常一个参数层层的传下去...,如果通过数据耦合就可以避免这个问题.

数据结构的定义要复杂而不是算法复杂.

各种计算不应该是在每次请求的时候,而是提前就算好,请求的时候直接给数据.

一定要注意接口的幂等性, 幂等性最好的方法还是加唯一索引.

把db层只做简单的更新,插入等,事务啊,我们放在logic层.组合的db操作我们也放在上面的层.

有些东西是基础服务,比如班课啊,很多东西都是依赖着班课,基础服务的概念要知道.

对于什么东西我们要冗余呢,对于那种不方便循环rpc的我么就冗余在表里避免查询时间太长.

函数内容一定不能过长.

函数不要全关联到包,而是要关联到logic对象上面.

不要在代码里面的所有的结构都用thrift的或者是pb的,而是要多用自己定义的数据结构,也就是定义在logic里面的数据结构.

logic一般都会有一个newlogic的代码,这个newlogic看起来是没什么用,但是类似于上次的那种选择题的数组默认有abcd的这种,这个时候newlogic就可以初始化一些东西了.

尽量把一些方法都关联在logic结构体上,这样的话可以方便使用logic结构体的元素数据,而不是用参数传递来传递去,不要经常搞那些大写开头的代码,那种是类似于静态的代码包名来调用.

以全局数据为中心去操作来维护,其他的各种函数都是来操作这个全局数据的,而不是数据的传递,围绕一个很大的数据结构把代码给串起来.

Api-service-dao的关系是这样的,层与层之间是互相不接触的,所有的操作的数据都是互相抽象的.

对于service层来说,只认订单,并不认你下面有多少表,(这就是互相抽象的意义)

而上文所说的定义的复杂的数据结构就是在service层.

服务的边界,服务的强自治,也就是一个服务最好能单独的就提供功能,不要依赖太多其他服务,不然可能导致rpc调用环,引起rpc风暴,所以服务的边界的划分就十分的重要.除了上面所说的基础服务,还有那种聚合服务,比如trade服务就是一个聚合服务,他背后还有优惠价啊库存啊等服务.

服务的菱形分层,api层也就是入口少,dao层也就是和数据的交互少,中间的service层要多.



常量的定义不要只想着定义字符串,数字啥的,就是这种数组啊,map啊,也是可以定义成常量的, 主要还是对数组的常量的定义.

下面的代码是在协程中用的

在我们普通的代码中,如果出现了panic,我们的代码一般也不会写defer和recover还抓住这个panic,那么这个panci就会跑到最外面的我们的一个recover的地方,就会被抓住. 但是呢,如果我们自己起的那种协程panic了,外面是抓不住的,所以在我们自己起的协程里面一定要加上下面这段代码,一旦panic了也是可以知道的,而且还可以安全的把资源或者业务给处理好.

defer func(id int64) {

  if e := recover(); e != nil {

If e != nil 就说明来这里已经出现来panic了,就可以处理了.

   const size = 64 << 10

   buf := make([]byte, size)

   buf = buf[:runtime.Stack(buf, false)] 这里可以用runtime.Stack()函数来打印堆栈信息

   logs.CtxError(ctx, "KITE: panic in handler: %s: %s", e, buf)

   metrics.EmitCounter("GetReport.panic", 1, map[string]string{"source": "homework"})

   res.err = errors.New("panic in processSingleXiaoban")

  }

  ch <- res

但是即使上面panic了,也只是会在上面打印些东西,这个地方该跑还是会跑的,因为外面的select会因为这个东西而阻塞,这时候defer的好处就出来了,即使panic了,也不会让有些代码跑不到

 }(xiaobanID)

上面的代码要注意理解

## Sql

Select where amount < 3000 这样是有问题的,因为如果amount是null的话,是捞不出来的,所以也侧面反应来,default的重要性!!!!!

update xxx= xx +1 这样是原子的.

db的分页是走的offset,偏移量不会把所有的数据都捞出来.

但是如果这个操作在两个事务中还是会有问题

Left join xxxxx on a.user\_id = b.user\_id where b.user\_id is not null;

用on的话后面得是一个where.

SELECT \* FROM table WHERE zongbu NOT REGEXP'北京|上海|深圳|天津|香港|沈阳';

多个not like无论是or还是and都不好使，要用这个。

## 数据库

如果是not null的话就必须要有默认值。而text没有默认值的,所以给text设置成not null就会出问题

这是一个用lua脚本的地方,redis的eval语句就是用来执行lua脚本的,语句中的内容就是先get,如果get到了就去delete

var script string = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end"

 if result, err := redisClient.Eval(script, []string{lockKey(roomId)}, uuid).Result(); err != nil {

  logs.CtxWarn(ctx, "RedisUnLock Eval error: %v", err)

  return false, err

 } else if result.(int64) != 1 {

  logs.CtxWarn(ctx, "RedisUnLock result error: %v", result)

  return false, nil

 }

 return true, nil

读写分离 分了主从的话就一定会有那种双机房问题,写到了主里去从读就读不到,但是如果你setnx这是个写操作就会全到主里面.

那么这种双机房问题就咋办呢,电商公司一般都不会读写分离的,一是容易出问题,数据量大的话还是要走分库分表,头条这种feed流起家的一般都是走读写分离.

public static void usePipeline(){

        ShardedJedis jedis = getShardedJedis();

        ShardedJedisPipeline pipeline = jedis.pipelined();

        long begin = System.currentTimeMillis();

        for(int i = 0;i<count;i++){

            pipeline.set("key\_"+i,"value\_"+i);

        }

        pipeline.sync();

        jedis.close();

pieline报错可能是因为pipeline的长度是0

幂等就是同一个消息来了重复了两次,不会重复执行两次,保持幂等就是执行前去查一下咯,有没有执行过,有没有用过.

opentsdb是基于Hbase的时序数据库[时间序列数据库]。不具备通用性，主要针对具有时间特性和需求的数据，如监控数据、温度变化数据等。opentsdb说是数据库，但并不能称作为数据库，他是在Hbase(HBase才是具有存储功能的)的基础上，进行数据结构的优化和处理，从而适合存储具有时间特性的数据，同时提供特定的工具进行查询等操作。

## Protobuf

protobuf比json更加的块 速度。时间效率空间效率都是json的3-5倍

repeated GradeInfo grade\_list = 5;//年级列表  在pro里面这个repeated表示的意思是list

可以向前兼容向后兼容

有代码生成机制

## Thrift

include "../base.thrift"

include "./paper.thrift"

include "./examination.thrift"

应用别的thrift文件

## Gorm

如果是个查询语句是可以用con.Raw(xxx).Rows()来查询的

并且可以用scan来取数据

如果是个update语句就不能用raw的啊 而是要是con.Exec(xxx)



这种的话,updates是不会更新0值的,如果你把一个本来是1的列改成0的话是不会生效的.

对于普通的con.table.where(xx)的查询后面就用Find(xx)就可以了

如果是sql直写的查询,就用Raw()了 raw里面写内容

db.Raw("SELECT name, age FROM users WHERE name = ?", 3).Scan(&result)

那么上面的scan()是咋回事呢,scan一般用于不查询一列的全部数据,而是查询其中某几列数据的时候使用,你看上面只查询了name和age,而reuslt这个结构体里的参数正是name和age.

但是如果查询出来的是多条

rows, err := db.Model(&User{}).Where("name = ?", "jinzhu").Select("name, age, email").Rows() // (\*sql.Rows, error)defer rows.Close()

for rows.Next() {

...

rows.Scan(&name, &age, &email)

...

}

用Select(“列名s”)来确定查询哪几个语句,然后用Rows()找到多条数据,然后在for循环去scan()

// Raw SQL

rows, err := db.Raw("select name, age, email from users where name = ?", "jinzhu").Rows() // (\*sql.Rows, error)

defer rows.Close() close row

for rows.Next() { row.Next()

...

rows.Scan(&name, &age, &email)

...

}

下面这个sql直写的也同理,由于是多个,后面也得是Rows(),然后再for循环.

conn.Model(&ExamRule{})灵活一点 相对于conn.Table() 这个都是上来都选了一个表

mysql事务

conn, err := GetWriteConn(ctx)

 if err != nil {

  return fmt.Errorf("CreatePaper GetConnection error: %v", err.Error())

 }

 tx := conn.Begin()

 defer tx.Rollback() 如果出panic了,理论上panci会一直往上抛就会导致事务无法回滚,但是呢,这里加了defer 即使panic了,事务也会被回滚.

 if err = tx.Error; err != nil {

  return fmt.Errorf("CreatePaper get tx error: %v", err.Error())

这样可以保证在开启事务的时候出错也能捕获到.

 }

下面这样可以保证提交事务的时候的错误也会被捕获到

if err = tx.Commit().Error; err != nil {

  return fmt.Errorf("CreatePaper commit error: %v", err.Error())

 }

mysql加锁

var res Result

 err = tx.Table("cron\_job").Raw("select exec\_state, exec\_time from cron\_job where id = ? for update", id).Scan(&res).Error

 if err != nil {

解锁

tx.Commit().Error

## 虚函数

首先,java中的函数除了构造和静态哪些都是虚函数,虚函数的意义和我们以前想的是一样的,有什么动态分派啊,虚函数表啊,这里主要讲一下c++中的虚函数,因为c++是有一个visual关键字的,会把一个函数标记成虚函数.

如果你不标记成虚函数的话就不会有多态的特性了,你的静态类型是什么,就会调用那个函数,一旦你标记成visual后就会有类似于java的多态的效果了.

所以c++默认是没有虚函数的.

## Tips

所以经过我的验证,我知道了我们的每个日志,不管是多长,其实都是一行的啊,所以我们可以用grep和awk操作啊,因为都是同一行,也就是换行符只有那么一个

str2 := "hello" + " " + "world"   字符串可以这样拼接 但是加上整数不行

Thrift 会生成method。idl生产的是pb——gen,不是idl也是会生成method的,如果你的pb没有生成method说明你忘记写service了.

加了代理就用不了 postman了 加了代理就用不了 postman了

那么序列化的本质是把结构体编程btye[],所以我们看到go语言的json的这两个函数的参数和返回值都是byte[] 而不是字符串哦。但是go的string和byte[]之间是可以非常方便的转换的

json也是一种序列化方式,序列化的结果是一定是byte[]

Echo $GOPATH

Echo $SHELL 这样可以打印一些生效的环境变量咯.

如果你要是用咯zsh,以后哪些环境变量相关的东西就挪到.zshrc里面咯

售卖流程都是会有createorder和confim order的.

Sd lookup 就是去consul走服务发现流程了.

Tlv type length value

Overload 限流

kerberos. 就是一个单点登录的协议,有一个票据服务,先去这个票据服务获取票据,然后你就可以,访问其他服务不用登陆了,kinit的k就是她

if len(os.Args) != 2 {

fmt.Fprintf(os.Stderr, "Usage: %s host:port", os.Args[0])

os.Exit(1)

}

service := os.Args[1]

这个os.Args os.Args[1]啥的乱七八糟的东西,就是你在执行这个bin文件的时候后面跟的参数

接口去抖就是防止用户手抖点击多次，导致瞬间调用接口两次。

大力这里用的也不是protobuf。如果真用的是protobuf的话，那么你的http请求响应体李的内容你就看不懂了，因为那里都是乱码了

centos yum mac brew debain apt

ssh的生成是由git登陆的邮箱生成的，每输入一次生成的都不一样，但是私钥文件是在你本地的，公钥配置在远端。

设置ssh key免密登陆服务器，把自己的公钥配置在服务器的.ssh文件夹下的一个文件里。

Reflect.TypeOf()这个是可以查看变量类型的,也可以看到go语言也是有指针的指针的,但是呢,对于json的unmarshal来说,这个你传指针,还是指针的指针都是能够处理的.

句柄（handle），有多种意义，第一种解释：句柄是一种特殊的[智能指针](https://baike.baidu.com/item/%E6%99%BA%E8%83%BD%E6%8C%87%E9%92%88/10784135" \t "/Users/yangwenshuo/Documents\\x/_blank) 。当一个应用程序要引用其他系统（如数据库、操作系统）所管理的内存块或对象时，就要使用句柄。 [1]  第二种解释：整个Windows编程的基础。一个句柄是指使用的一个唯一的整数值，即一个4字节(64位程序中为8字节)长的数值，来标识[应用程序](https://baike.baidu.com/item/%E5%BA%94%E7%94%A8%E7%A8%8B%E5%BA%8F/5985445" \t "/Users/yangwenshuo/Documents\\x/_blank)中的不同[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1/17158" \t "/Users/yangwenshuo/Documents\\x/_blank)和同类中的不同的实例，诸如，一个窗口，按钮，图标，滚动条，输出设备，控件或者文件等。

句柄泄漏是进程在调用系统文件之后，没有释放已经打开的文件句柄。



过载就是服务器抗不住了，你的下游找你让你别调用他了，你就可以配置下限流。



熔断的意思和以前差不多。

限流和熔断都是保护下游不被打挂的。



集成了一些公共的告警比如cpu过高，错误日志过多，协程太多。