# Golang

## Tips

所以说在锁和waitgroup里面 信号量和cas操作是经常用到的.

信号量（[semaphore](https://so.csdn.net/so/search?q=semaphore&spm=1001.2101.3001.7020" \t "/Users/yangwenshuo/Documents\\x/_blank)）是操作系统用来解决并发中的互斥和同步问题的一种方法

`{"service\_name":"%s"}` 字符串这样定义里面的双引号可以不用转义了

数组 也就是长度是固定的东西,是按照值来传递的!!!!!!!!!!!!但是切片不是

可以看到源码里面是没有什么defer释放锁的, 那样还是不太好

math.MaxFloat64

SetLogger() 和 Logger() 不要GetLogger()

var builder1 strings.Builder

builder1.WriteString("A Builder is used to efficiently build a string using Write methods.") 字符串拼接用strings.Builder

for finshnum < len(urls) {

这里这样写可以代替里面的判断长度然后break 这样的for是正确的方式  
 select {  
 case v := <- ch:  
 result[v.url] = v.result  
 finshnum++  
 case <- cc:  
 fmt.Println("怎么还不来啊")  
 return result  
 }  
}  
fmt.Println("chu l xunhuan")



当你想要把一个函数传进去的时候,参数不要直接写这个函数,而是要先把这个函数给type了,无论你创建参数还是函数定义都是比较好的

改过一个问题后,一定要想想还有没有其他地方要改.

出问题了先百度问题提示.

代码一定要一次性写完,不要拖到明天,不然有的东西真的会忘记.

第三方的库类似于ping++ 阿里云的源代码在github可以找到.

尽量用>>和<<的位移操作来代替乘和除.

接收一个消息队列的数据 一定要保证幂等性 已经什么时候err 什么时候丢弃 是否主从等问题, 解决消息队列的主从可以在消息体里加一个字段就做消费时间,如果consume消费到的时间大于了消费时间就去消费.

大结构尽量用指针, 如果是结构会拷贝.

进程的定时任务是不靠谱的,因为进程死了,定时任务就没了.

删东西的时候要全局grep下,不然有可能有的地方用到

A, b:= 如果之前没有b,那么就是一个新建b,如果之前有了b但是没有a, 那么b就是重新赋值.

func main() {  
  
 var err error  
  
 defer func() {  
 if err != nil {  
 fmt.Println("来了")  
 }  
 }()  
  
 if err := ceshi(); err != nil {  
 fmt.Println(err)  
 }  
}  
  
func ceshi() error {  
 return errors.New("啊啊啊啊")  
}

这样是不会走到defer的

func main() {  
  
 var err error  
  
 defer func() {  
 if err != nil {  
 fmt.Println("来了")  
 }  
 }()  
  
 s, err := ceshi()  
 fmt.Println(s)  
}  
  
func ceshi() (string, error) {  
 return "aaa", errors.New("啊啊啊啊")  
}

这样是会走到defer的

func main() {  
 n := 0  
 f := func() int {  
 n += 1  
 return n  
 }  
 fmt.Println(f()) *// 别忘记括号，不加括号相当于地址*

闭包,就是匿名函数用了外面的变量 这个函数不仅是描述了执行操作,还记住了外面的变量的地址.

*//由 main 函数作为程序入口点启动*func main() {  
 x, y := 1,2  
  
 defer func(a int){  
 fmt.Println("defer x, y = ", a, y) *//y为闭包引用* }(x) *//x值拷贝 调用时传入参数* x += 100  
 y += 200  
 fmt.Println(x, y)  
}

这个的打印结果是 101 202 和 1 202

注意defer的匿名函数如果传参数的话,那么在这个defer注册的时候就已经把这拷贝了,也就是a是被当作参数传进来的,他就是1, 而y是做为闭包弄进来的,那么他就会实时感受到y的变化.

SLA：Service-Level Agreement的缩写，意思是服务等级协议

过去10秒内最慢的x%的请求的平均延迟.其中X是数字与100的差.

例如:p99 1.403 表示过去的10秒内最慢的1%请求的平均延时为1.403秒

p95 过去的10秒内最慢的5%的请求平均延时.

为什么我们换域名 前端都不用改代码,不用改域名,因为前端的那些js,html图片的资源其实就是通过域名先拉到的,那么代码里面肯定可以获取到当前域名,而app的的域名是写在自己的代码里面的,所以一旦域名改了老的app就用不了了.

10M带宽=1.25MB/s网速：

跨域 值得说的是虽然浏览器禁止用户对请求返回数据的显示和操作，但浏览器确实是去请求了，如果服务器没有做限制的话会返回数据的，在调试模式的network中可以看到返回状态为200，且可看到返回数据

客户端发送syn 自己进入synsended状态,

服务端收到后返回syn+ack 服务端进入syn-recved状态

此时就进入了半链接状态

如果这个时候客户端不回复响应,服务端还会重试,还会等超时,这样肯定会消耗服务端用于维护半链接的资源就阻塞服务端了.

浮点数用%v方式打印超过一百万会变成科学记数法.

浮点型数据的大小比较不能直接用等号,而是要用func isEqual(p1, p2, float64)

https的服务也是可以控制台看到明文的,因为在http层的数据已经被解码了,在具体传输的时候是加密的.

为啥bd内部起的各种服务都是http的服务,因为https的东西已经被bgw给做了.

http.HandleFunc("/hello", hello) http.ListenAndServeTLS(":8080", certPath, keyPath, nil)

还有就是mme的虚拟线程,和go的携程还是有点不一样的,因为mme的虚拟线程的运行哪个的调度完全是有我们的代码控制的,而go语言是由go的调度器控制的.



## 协程池

github.com/panjf2000/ants

gopool

## IM

type client chan<- string *// send only channel*var (  
 entering = make(chan client)  
 leaving = make(chan client)  
 messages = make(chan string)  
)

func main() {  
 listener, err := net.Listen("tcp", "0.0.0.0:8888")  
 if err != nil {  
 log.Fatal(err)  
 }  
  
 go broadcaster()  
  
 for{  
 conn, err := listener.Accept()  
 if err != nil {  
 fmt.Fprintf(os.Stdout, "you got something wrong %v", err)  
 continue  
 }  
 go handleConn(conn)  
 }  
}

func broadcaster() {  
 clients := make(map[client]bool) *//all connected clients* for {  
 select {  
 case msg := <- messages:  
 *// Broadcast incoming message to all  
 // clients' outgoing message channels.* for cli := range clients{  
 cli <- msg  
 }  
 case cli := <- entering:  
 clients[cli] = true  
 case cli := <- leaving:  
 delete(clients, cli)  
 close(cli)  
 }  
 }  
}

## 编译原理

golang的源码是可以自己修改重新编译的。

抽象语法树 静态单赋值(SSA)

X86指令集 ARM指令集

编译器前端:承担着词法分析语法分析(生成抽象语法树),类型检查,中间码生成.

编译器后端:目标代码的生成和优化,翻译成中间码到机器码.

go源码包里面包含了不同指令集机器码生成所需的包,比如生成x86机器码的包, ARM机器码的包.

有了抽象语法树,那么类型检查的时候就会类似操作树一样什么left right node之类的了.

中间代码的存在是必要的,不能直接源码到机器码的.

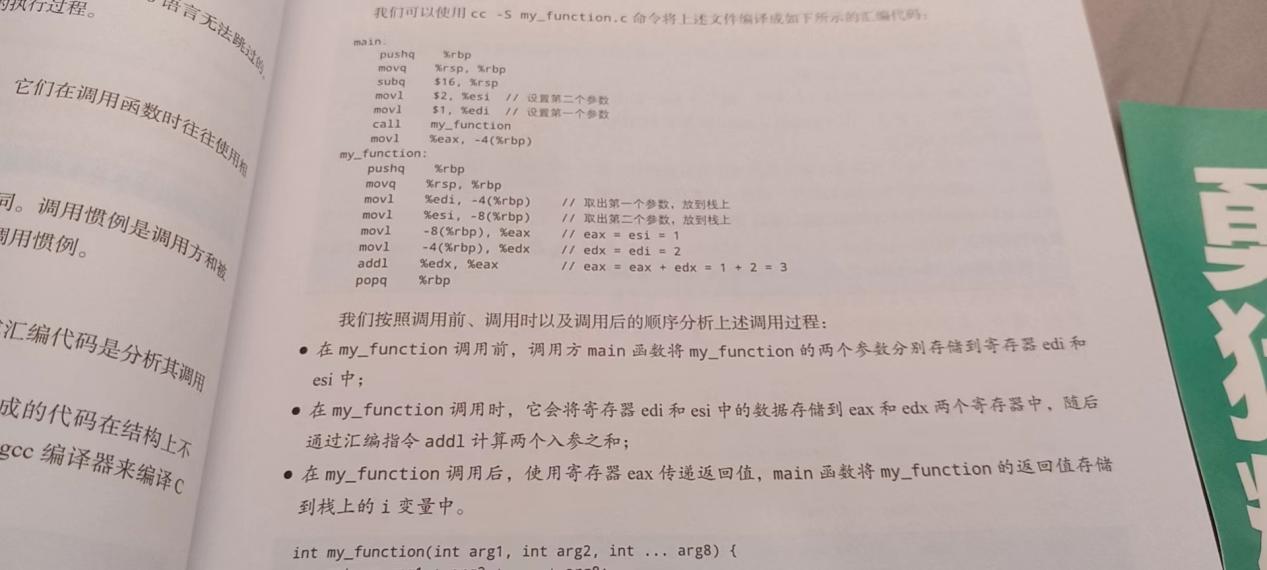
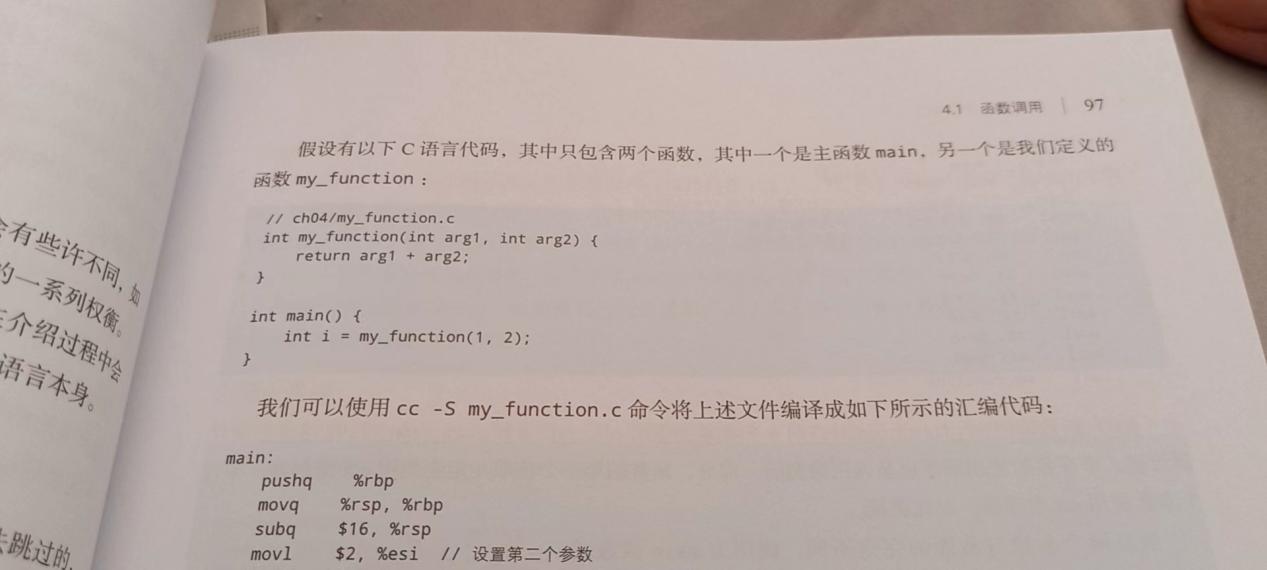
中间代码的生成过程就是抽象语法树到SSA中间代码的转换过程.

指令集是软硬件之间的桥梁,所以指令集架构是计算机的抽象模型,所以指令集架构也是计算机架构,包括x86和ARM.

X86就是复杂指令集,ARM是精简指令集.由于复杂指令集指令很多,指令长度也就不同,需要消耗额外的性能去判断指令长度,而简单指令集是用固定长度的基础指令组合使用,所以更好,性能更好.

中间代码会转成汇编代码, 汇编代码经过汇编器就会转成机器码

静态语法树 - SSA中间代码 - 汇编代码 - 机器码



可以看到上面的汇编,确实是值传递,里面的函数会复制一份外面传进来的参数使用.

是这样的,外面的1和2的实参,是main函数放到寄存器中的,myfunc用的时候会从寄存器里面复制数据放到自己的栈里,所以是值传递.

最后在用eax寄存器传递返回值,这个寄存器是专门用来传递返回值的,只有一个所以c语言只有一个返回值.

看来寄存器是有个数限制的,而且都有自己独特的名字.

c语言和go语言在设计函数调用选择不同的.

C语言使用寄存器和栈传递参数(如果参数数量小于6个全用的寄存器)

go使用栈传递参数和返回值.

c语言这样更快,cpu访问寄存器的速度比访问栈快几十倍.

go这样比较简单,不需要适配不同的寄存器,可以返回多个值,因为返回值也是在栈上,栈上多开空间不就可以返回多个了,但是寄存器却只有几个,就不能返回多个参数了.

for循环的实现基本都是用到了汇编的

JUMP命令跳转(跳到循环开始的地方) INCQ命令++ CMPQ命令比较.

## Chan



一个chan同时只能被一个协程读或者写。

1. func makechan(t \*chantype, size int) \*hchan {

2. var c \*hchan

3. c = new(hchan)

4. c.buf = malloc(元素类型大小\*size)

5. c.elemsize = 元素类型大小

6. c.elemtype = 元素类型

7. c.dataqsiz = size

8.

9. return c

10. }

可以看出chan里面是有两个队列的，一个存的是想读的阻塞在这里的协程list,还有一个是想写的阻塞在这里的协程list, 这俩会根据情况来唤醒调度。

就是一个阻塞消息队列.

可以有多个consumer和多个producer

## Slice

7. func main() {

8. var array [10]int

9.

10. var slice = array[5:6]

11.

12. fmt.Println("lenth of slice: ", len(slice))

13. fmt.Println("capacity of slice: ", cap(slice))

14. fmt.Println(&slice[0] == &array[5])

15. }

这个len是1， 但是cap是5，后面的比较是true

注意，第二个是cap是5， 这个cap就是从起始位置，到真正的空间的结尾的位置。

append如果扩容了地址才会变的。

使用append向Slice追加元素时，如果Slice空间不足，将会触发Slice扩容，扩容实际上重新一配一块更大的内存，将原Slice数据拷贝进新Slice，然后返回新Slice，扩容后再将数据追加进去。

slice := array[start:end]

1. type slice struct {

2. array unsafe.Pointer

3. len int

4. cap int

5. }

使用make来创建Slice时，可以同时指定长度和容量，创建时底层会分配一个数组，数组的长度即容量。

那种通过下标创造的切片是一个新的切片,他也有自己的len cap属性,但是data还是指向原来的切片,所以原来的改了他也会改.

切片的扩容也会用到memove

## Map

map的实现是hash表。

1. type hmap struct {

2. count int // 当前保存的元素个数

3. ...

4. B uint8 // 指示bucket数组的大小

5. ...

6. buckets unsafe.Pointer // bucket数组指针，数组的大小为2^B

7. ...

8. }

buckets就是个bucket的数组，所以这里也是用的unsafe的指针。

1. type bmap struct {

2. tophash [8]uint8 //存储哈希值的高8位

3. data byte[1] //key value数据:key/key/key/.../value/value/value...

4. overflow \*bmap //溢出bucket的地址

5. }

这是其中一个的bucket的结构

可以看到，一个bucket并不是只存一个k-v，而是存8个，后面也是bucket做的拉链表。

注意放kv的时候也是key和value分开放，这样可以防止字节对不齐。

负载因子 = 键数量/bucket数量

哈希表需要将负载因子控制在合适的大小，超过其阀值需要进行rehash，也即键值对重新组织：

哈希因子过小，说明空间利用率低

哈希因子过大，说明冲突严重，存取效率低

每个哈希表的实现对负载因子容忍程度不同，比如Redis实现中负载因子大于1时就会触发rehash，而Go则在在负载因子达到6.5时才会触发rehash，因为Redis的每个bucket只能存1个键值对，而Go的bucket可能存8个键值对，所以Go可以容忍更高的负载子。

当负载因子过大时，就新建一个bucket，新的bucket长度是原来的2倍，然后旧bucket数据搬迁到新的bucket。

考虑到如果map存储了数以亿计的key-value，一次性搬迁将会造成比较大的延时，Go采用逐步搬迁策略，即每次访问map时都会触发一次搬迁，每次搬迁2个键值对。

如果查找不到，也不会返回空值，而是返回相应类型的0值。

如果当前处于搬迁过程，则优先从oldbuckets查找

## String

1. type stringStruct struct {

2. str unsafe.Pointer

3. len int

4. }

string数据结构跟切片有些类似，只不过切片还有一个表示容量的成员，事实上string和切片，准确的说是byte切片经常发生转换。

字符串构建过程是先跟据字符串构建stringStruct，再转换成string

string在runtime包中就是stringStruct，对外呈现叫做string。

1. func gostringnocopy(str \*byte) string { // 跟据字符串地址构建string

2. ss := stringStruct{str: unsafe.Pointer(str), len: findnull(str)} // 先构造stringStruct

3. s := \*(\*string)(unsafe.Pointer(&ss)) // 再将stringStruct转换成string

4. return s

5. }

这个unsafe.pointer是可以转换成各种类型的指针的。

字符串和[]byte 互相转换其实都是内存拷贝的。

频繁的字符串+拼接这样效率是很低的，如果要是正经拼接的话的是strings.builder。

像C++语言中的string，其本身拥有内存空间，修改string是支持的。但Go的实现中，string不包含内存空间，只有一个内存的指针，这样做的好处是string变得非常轻量，可以很方便的进行传递而不用担心内存拷贝。

因为string通常指向字符串字面量，而字符串字面量存储位置是只读段，而不是堆或栈上，所以才有了string不可修改的约定。

字符串的结构和切片的区别就是字符串没有cap字段.字符串的拼接也是需要用到copy函数的,而且新拼出来的字符串是个新的空间,不是追加, 所以不挺的拼接,那么就会不停copy越来越大的空间性能就会下降.

字符串是不可写的,转成[]byte是可写的,其实这个转换也是copy了一份,如果很大的话也是会损耗性能的.

byte切片转换成string的场景很多，为了性能上的考虑，有时候只是临时需要字符串的场景下，byte切片转换成string时并不会拷贝内存，而是直接返回一个string，这个string的指针(string.str)指向切片的内存。

比如，编译器会识别如下临时场景：

使用m[string(b)]来查找map（map是string为key，临时把切片b转成string）；

字符串拼接，如”<” + “string(b)” + “>”；

字符串比较：string(b) == “foo”

因为是临时把byte切片转换成string，也就避免了因byte切片同容改成而导致string引用失败的情况

## Defer

defer语句用于延迟函数的调用，每次defer都会把一个函数压入栈中，函数返回前再把延迟的函数取出并执行。 为了方便描述，我们把创建defer的函数称为主函数，defer语句后面的函数称为延迟函数。

1. func deferFuncParameter() {

2. var aInt = 1

3.

4. defer fmt.Println(aInt)

5.

6. aInt = 2

7. return

8. }

这个输出的是1，因为这个是走的参数，我们以前的那种走的是闭包defer引用了外面的变量，而不是参数传进来的。

1. func deferFuncReturn() (result int) {

2. i := 1

3.

4. defer func() {

5. result++

6. }()

7.

8. return i

9. }

这个最终会返回2.

return不是原子的，会先把i赋值给result，然后在执行defer里面的内容。

注意是这个流程。

这种函数带有具名返回值的就是这个流程。

1. func foo() int {

2. var i int

3.

4. defer func() {

5. i++

6. }()

7.

8. return i

9. }

注意这里最后是0，

流程还是一样的，先把i赋个一个匿名的result，defer又去把i给++了，最后返回的是匿名result，是这个流程的。

1. type \_defer struct {

2. sp uintptr //函数栈指针

3. pc uintptr //程序计数器

4. fn \*funcval //函数地址

5. link \*\_defer //指向自身结构的指针，用于链接多个defer

6. }

这是一个defer的链表

deferproc()： 在声明defer处调用，其将defer函数存入goroutine的链表中；

deferreturn()：在return指令，准确的讲是在ret指令前调用，其将defer从goroutine链表中取出并执行。

## Select

select是Golang在语言层面提供的多路IO复用的机制

## 并发

锁那些玩意本来就肯定会用到信号量的啊

1. type Mutex struct {

2. state int32

3. sema uint32

4. }

Mutex.state表示互斥锁的状态，比如是否被锁定等。

Mutex.sema表示信号量，协程阻塞等待该信号量，解锁的协程释放信号量从而唤醒等待信号量的协程



Locked: 表示该Mutex是否已被锁定，0：没有锁定 1：已被锁定。

Woken: 表示是否有协程已被唤醒，0：没有协程唤醒 1：已有协程唤醒，正在加锁过程中。

Starving：表示该Mutex是否处理饥饿状态， 0：没有饥饿 1：饥饿状态，说明有协程阻塞了超过1ms。 这里就可以统计有多少饥饿的协程。

Waiter: 表示阻塞等待锁的协程个数，协程解锁时根据此值来判断是否需要释放信号量。

协程之间抢锁实际上是抢给Locked赋值的权利，能给Locked域置1，就说明抢锁成功。抢不到的话就阻塞等待Mutex.sema信号量，一旦持有锁的协程解锁，等待的协程会依次被唤醒。



自旋对应于CPU的”PAUSE”指令，CPU对该指令什么都不做，相当于CPU空转，对程序而言相当于sleep了一小段时间，时间非常短，当前实现是30个时钟周期。

自旋过程中会持续探测Locked是否变为0，连续两次探测间隔就是执行这些PAUSE指令，它不同于sleep，不需要将协程转为睡眠状态。

Mutx lock方法

先是cas操作看看能不能直接锁上,不能的话就自旋,再看能不能用cas锁上.如果还不能就通过信号量的方获取锁了.

Unlock 先看能不能cas操作释放锁,不能的话就信号量.

所以golang也是有自旋锁的.

waitgroup也是用到了信号量和cas操作的

1. type RWMutex struct {

2. w Mutex //用于控制多个写锁，获得写锁首先要获取该锁，如果有一个写锁在进行，那么再到来的写锁将会阻塞于此

3. writerSem uint32 //写阻塞等待的信号量，最后一个读者释放锁时会释放信号量

4. readerSem uint32 //读阻塞的协程等待的信号量，持有写锁的协程释放锁后会释放信号量

5. readerCount int32 //记录读者个数

6. readerWait int32 //记录写阻塞时读者个数

7. }

可以看到，里面还是有一个mutex的。

## Gmp

空闲的P会将其他P中的G偷取一部分过来，一般每次偷取一半。

## Timer

1. type timer struct {

2. tb \*timersBucket // the bucket the timer lives in // 当前定时器寄存于系统timer堆的地址

3. i int // heap index // 当前定时器寄存于系统timer堆的下标

4.

5. when int64 // 当前定时器下次触发时间

6.

period int64 // 当前定时器周期触发间隔（如果是Timer，间隔为0，表示不重复触

发）

7. f func(interface{}, uintptr) // 定时器触发时执行的函数

8. arg interface{} // 定时器触发时执行函数传递的参数一

9. seq uintptr // 定时器触发时执行函数传递的参数二(该参数只在网络收发场景下使用)

10. }

会有一个叫做timerbucket的结构，所有的timer都放在这里面，然后有专门的系统协程去调度这些timer。

这些timer会被存到一个小顶堆里面，然后距离执行时间最近的timer放在堆顶，这样去调度就可以了。

## 反射

switch str := value.(type) {

case string:

return str

这样interface.(type) 配合switch case

## Context

Cancelctx cancel了父, 会一层层给子也发消息,那么cancelctx一定会有一个字段存的childlist.

valuectx会先查自己,自己查不到就去查父,那么一定会有一个字段是存的parent.

为了能cancel父的时候把子也cancel了,所以cancelctx肯定有个地方存了自己的childmap.那么valuectx肯定也存了自己的parent.

## GORM

if err := Db.Model(xy).Where("id = ? ", id).Update("sign\_up\_num", gorm.Expr("sign\_up\_num+ ?", 1)).Error; err != nil {

return false

}

Save是会更新0值的,但是save的参数只能是个结构体,因为他会更新所有值.

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

## Sql

IF(process\_status = 2, 1, 0) as process\_status

select user\_name, count(DISTINCT tel) from user group by user\_name;

多个or条件要用括号扩起来

多个not in not like 要用or来链接.

INSERT INTO t\_stock\_chg(f\_market, f\_stockID, f\_name) VALUES('SH', '600000', '白云机场') ON DUPLICATE KEY UPDATE f\_market='SH', f\_name='浦发银行';

Insert on dunplicate key 就是有了就更新 没有就insert

那么到底怎么判断这个是不是已经存在呢,需要这个表有一个主键id或者uniquekey, 如果你的参数中包含了已经存在的主键id或者唯一key,那么就用这个找到了已有的数据进行更新.注意的是,如果最终是insert,effectrow是1, 最后是更新,effectrow是2, 如果你传的参数和表里的各个字段都一样,是不是做任何操作的,udpatetime也不会变的.

查看表的大小

select concat(round(sum(data\_length/1024/1024),2),'MB') as data from information\_schema.tables where table\_schema='ev\_courseware' and table\_name='user\_banke\_wrong\_item\_record' limit 10;

select record\_id, ea\_item\_id, count(\*), max(update\_time) from user\_question\_record where subject = 2 and user\_id = 1 group by ea\_item\_id;

mysqldump -h[47.92.202.123](/Users/yangwenshuo/Documents\\x/47.92.202.123) --skip-lock-tables --skip-add-locks -uhuamath\_dragon\_w -pebe00208299ad3bcb8  -d huamath\_dragon > dump.sql

只复制表结构

mysqldump -h[10.225.88.235](/Users/yangwenshuo/Documents\\x/10.225.88.235) -P3306 -uev\_boe\_w -p6abPHgXPTzxTB9P\_3NM9WeQvshLVKNLY --skip-lock-tables --skip-add-locks --databases ev\_boe --tables chat\_info --where "create\_time>\"2020\"" | mysql -h[10.225.124.243](/Users/yangwenshuo/Documents\\x/10.225.124.243) -P3306 -uev\_classroom\_w -plno3sGjYGd81TS2\_GJxHP66VQuHLBaWt

复制所有数据

数据库里面是null的话查出来的是空串 并不是null这个字符串的啊

update xxx= xx +1 这样是没有问题的

show index  from user\_examination;

drop index xxx on table name

Find(&userExamList).  必须要取地址

show create table banke;

tx := conn.Begin()

 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

加锁



没有就create,有就update, assgin和firstorcreate

select examination\_id, count(\*) count from examination group by examination\_id having count > 1;

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。

所以解决幻读问题也可以用lock in share model配合nextkeylock

唯一键冲突是导致自增主键id不连续的第一种原因

事务回滚是导致自增主键id不连续的第二种原因

es不要权量存es 存id就可以,因为默认es会给所有字段都加上索引,查询的时候会把索引都加载到内存中,如果你索引内容太多内存加载不上,就会走硬盘,所以最好es里面只存id,通过一些条件检索出来后再用唯一id去db或者hive等库里根据id查询内容就ok了.

countid属实要比count(\*)快

text是可以变长存储的,就是容量比较大的varchar

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.

## 文章

<https://mp.weixin.qq.com/s/gVNKibDQ6UsX_q8_CHvg1A> 稳定性规范

<https://mp.weixin.qq.com/s/JQqex6kON6ixJUwodRZByA> es

<https://books.studygolang.com/The-Golang-Standard-Library-by-Example/> go语言标准库

<https://zhuanlan.zhihu.com/p/95056679?utm_source=wechatMessage_article_bottom&from=singlemessage>

<https://gocn.vip/topics/9611> go1.4

<https://cloud.tencent.com/developer/article/1434134> 限流算法

<https://mp.weixin.qq.com/s/0IKxbt8MDH6Yqu1f00cwSA> kv

<https://zhuanlan.zhihu.com/p/79772089?utm_source=wechat_session&utm_medium=social&utm_oi=54330396770304> 切换上下文开销

[https://zhuanlan.zhihu.com/p/22557362?utm\_source=wechat\_session&utm\_medium=social&utm\_oi=54330396770304&from=singlemessage&s\_s\_i=EuzWvW5ULdKasS50J8NyYlWM5Vs%2FPJi1BiU4UfbOeDg%3D&s\_r=1](https://zhuanlan.zhihu.com/p/22557362?utm_source=wechat_session&utm_medium=social&utm_oi=54330396770304&from=singlemessage&s_s_i=EuzWvW5ULdKasS50J8NyYlWM5Vs/PJi1BiU4UfbOeDg=&s_r=1) 无锁queue

<https://blog.csdn.net/hil2000/article/details/51638751?depth_1-utm_source=distribute.pc_relevant.none-task&utm_source=distribute.pc_relevant.none-task> 性能测试

<https://blog.csdn.net/xiaosongluo/article/details/80292894> 代码覆盖率

<https://blog.csdn.net/liumiaocn/article/details/54922487> 代码覆盖率

[https://item.jd.com/11152134.html#comment](https://item.jd.com/11152134.html" \l "comment) 量化研究方法

<https://mp.weixin.qq.com/s/o2oMMh0PF5ZSoYD0XOBY2Q> go gc

<https://mp.weixin.qq.com/s/24AvtcL8_cz_pQg2RYEaiQ> cpu如何识别代码

<https://www.xttblog.com/?p=2943> redis为何这么块

<https://blog.csdn.net/w372426096/article/details/88352833> rpc原理

<https://tech.meituan.com/2016/12/02/performance-tunning.html> 常见性能优化侧罗

<https://mp.weixin.qq.com/s/QMSQG-pPElPhI457g6OBqQ> 为服务中台架构

<https://maimai.cn/article/detail?fid=1372321387&efid=pm8WOnFAPtUMV4ivWMP4Tw&use_rn=1> epoll 本质

<https://juejin.im/post/5dce54ca51882510ba1cbee6> 编程规范[https://juejin.im/post/5c4f16dbe51d454f342fb7e7#heading-19](https://juejin.im/post/5c4f16dbe51d454f342fb7e7" \l "heading-19) flink

<https://mp.weixin.qq.com/s/VQGF1vNvv-3GjdS6viuMIA> shell