Go语言核心编程读书笔记

mp.weixin.qq.com/s/NDnusVr7xdVN4tfmxgQhBw

> 以下内容来源于公众号： Go生态

Go语言核心编程中文版怎么样?

《Go语言核心编程》是一本系统介绍Go 语言编程的书籍。首先介绍Go 语言的基础知识，使读者对Go 语言有一个整体的认知。接着围绕Go 语言的三大语言特性：类型系统、接口和并发展开论述，《Go语言核心编程》不单单介绍每个语言特性怎么使用，在章节的最后还对重要语言特性的底层实现原理做了介绍。接着介绍反射这个高级语言特征。此外，《Go语言核心编程》专门用一章的篇幅来介绍Go 语言的陷阱。最后介绍Go 语言的工程实践和编程思想。相信《Go语言核心编程》能够帮助读者快速、深入地了解和学习这门语言。

《Go语言核心编程》适合各个层次的Go 语言开发者阅读，初学者可以系统地从头学习，有一定的编程经验者可以选择性地阅读本书。

作者简介:

李文塔，现就职于腾讯，FiT（前身为财付通）支付账户核心DBA，高级工程师。近十年金融系统开发、运维经验，先后参与2016年到2018年微信春节红包项目和支付账户核心多地多活容灾项目。 2012年开始接触Go语言，是国内较早的Go语言实践者之一。

书籍目录:

读书笔记目录

    - gRPC流
    - 基于docker的pub/sub与grpc流实现发布订阅模型
- Go-Web
    - httprouter
    - 中间件
- 服务流量限制
    - 问题
    - 漏桶限流
    - 令牌桶
- 分布式系统
    - 分布式ID-雪花算法
    - 分布式锁
    - 延时任务系统
- 负载均衡

      基于洗牌算法的负载均衡
- 分布式配置管理
- Go语言中使用时的坑
    - 可变参数是空接口类型
    - 数组是值传递
    - recover必须在defer函数中运行
    - 独占cpu导致其他的goroutine饿死
    - 不同的goroutine之间不满足顺序一致性内存模型
    - 闭包错误引用同一个变量
    - 在循环内部执行defer语句
    - 切片会导致整个底层数组被锁定

数组，字符串，切片

数组

数组是值类型，虽然数组元素的值可以被修改，但是数组本身的赋值和函数传参，都是以整体复制的方式处理的。

数组是一个由**固定长度的特定类型的元素组成**的序列，一个数组可以由零个或者多个元素组成。

数组的长度是数组类型的组成部分，因为数组的长度是数组类型的一个部分，不同长度的或者不同类型的数据组成的数组都是不同类型的数组，无法直接赋值。

数组的定义方式

```
var a [3]int
var b = [...]int{1,2,3}
```

数组内存结构

Go语言中的数组是值语义，一个数组变量既表示整个数组，并不是隐式的指向第一个元素的指针。

当一个数组变量被赋值或者传递时，实际上会复制整个数组。

空数组

长度为o的数组在内存中不占用空间，空数组可以用于强调某种特有类型的操作时避免分配额外的内存

```
c1 := make(chan [0]int)
go func() {
  c1 <- [0]int{}
}()
```

内置函数

```
len() //计算数组的长度
cap() //计算数组的容量
```

字符串

Go语言字符串的底层数据也是对应的字节数组，但是**字符串的只读属性禁止了在程序中对底层字节数组的元素进行修改**。

字符串赋值只是复制了数据地址和对应的长度，而不会导致底层数据的复制

底层结构

```
type StringHeader struct {
  Data uintptr //指向底层的字节数组
  Len int //字符串的字节长度
}
```

切片

切片的底层数据也是数组，但是**每个切片还有独立的长度和容量信息**，切片赋值和函数传参时也是将切片头信息部分按传值方式处理。因为切片头含有底层数组的指针，所以它的赋值也不会导致底层数组的复制

底层结构

```
type SliceHeader struct {
  Data uintptr
  Len int
  Cap int
}
```

切片的定义

```
var (
 a []int //nil切片，和nil相等，一般用来表示一个不存在的切片
  b = []int{} //空切片，和nil不相等，一般用来表示一个空的集合
  c = []int{1,2,3} //有三个元素的切片，len和cap都为3
  d = c[:2] //有两个元素的切片 len=2 cap=3
  e = c[:0] //0个元素的切片 len=0 cap=3
  f = make([]int,3) //有三个元素的切片 len=3 cap=3
  g = make([]int,3,6) //有三个元素的切片 len=3 cap=6
)
```

相关知识

切片可以和nil进行比较，只有当切片底层数据指针为空时，切片本身为nil，这时候切片的长度和容量信息将是无效的

在对切片本身赋值或参数传递时，和数组指针的操作类似，只是复制切片头信息，并不会复制底层的数据

内置函数

```
append() // 增加切片元素，在容量不足的情况下，append操作会导致重新分配内存，可能导致巨大的内存分配和复制数据代价
```

切片内存技巧

> 切片高效操作的要点是降低内存分配的次数，尽量保证append操作不会超出cap的容量，降低触发内存分配的次数和每次分配内存的大小

```
// 扩容切片
a = append(a, 0) //切片扩展一个空间
copy(a[i+1:], a[i:]) //a[i:] 向后移一位
a[i] = x //设置新添加的元素

//删除切片元素
a = []int{1,2,3}
a = a[:len(a)-1]
a = a[1:]

//利用空切片的特性删除切片元素
func TrimSpace(s []byte) []byte {
  b := [s:0]
for _,x := range s {
if x != ' ' {
      b = append(b, x)
    }
  }
return b
}

        //提高GC效率
func FindPhoneNum(filename string) []byte {
  b, _ := ioutil.ReadFile(filename)
  return regexp.MustCompile("[0-9]+").Find(b)
}
/*
这段代码返回的[]byte指向保存整个文件的数组，因为切片引用了整个原始数组，导致自动垃圾回收器不能及时释放底层数组的空间，一个小的需求可能导致需要长时间保存整个文件数据。通过下面的方式进行优化
*/
func FindPhoneNum(filename string) []byte {
  b, _ := ioutil.ReadFile(filename)
  regexp.MustCompile("[0-9]+").Find(b)
  return append([]byte{}, b...)
}
```

函数，方法与接口

函数

在Go语言中，函数是第一类对象，我们可以将函数保存到变量中。

```
//具名函数
func Add(a, b int) int {
return a+b
}

  //匿名函数
var Add = func(a, b int) int {
  return a+b
}
```

闭包

> 一般我们把匿名函数捕获了外部函数的局部变量的函数称为匿名函数

```
func main() {
  for i:=0; i < 3; i++ {
    defer func(){println(i)}()
  }
}
//3
//3
//3
//因为闭包，每个defer语句延迟执行的函数引用的都是同一个i迭代变量，在循环结束后，i都为3，可以通过以下方法解决该问题
func main() {
  for i:=0; i<3; i++ {
    defer func(){println(i)}(i)
  }
}
```

切片作为入参

Go语言中，如果以切片作为参数调用函数时，有时候会给人一种参数采用了引用方式传递的假象：因为在被调用函数内部可以修改传入的切片元素。其实，任何可以通过函数参数修改调用参数的情形，都是因为**函数参数中显示或者隐式的传入了指针参数**。

因为**切片中的底层数组部分是通过隐式指针传递**（指针本身依然是传值的，但是指针指向的却是同一份数据）,所以被调用函数是可以通过指针修改调用参数切片中的数据。除了数据之外，切片结构还包含了切片长度和切片容量的信息，这两个信息也是传值的，如果被调用函数中修改了Len或Cap信息的话，就无法反映到调用参数的切片中，这时候我们一般会通过返回修改后的切片来更新之前的切片，这也是为什么内置的append必须要返回一个切片的原因

```
func twice(x []int) {
for i := range x {
   x[i] *= 2
  }
}



type IntSliceHeader struct {
  Data []int
  Len int
  Cap int
}

func twice(x IntSliceHeader) {
  for i:=0; i < x.Len; i++ {
    x.Data[i] *= 2
  }
}
```

递归

Go语言函数的递归调用深度逻辑上没有限制，函数调用的栈是不会出现溢出错误的，因为Go语言运行时会根据需要动态的调整函数栈的大小。每个goroutine刚启动时只会分配很小的栈，根据需要动态调整栈的大小。

方法

Go语言中方法是关联到类型的，这样可以在编译阶段完成方法的静态绑定。

组合

```
type Point struct{X, Y float64}

type ColorPoint struct {
  Point
  Color color.RGBA
}
```

接口

> Go的接口类型是对其他类型行为的抽象和概括，Go语言的接口类型独特之处在于它是满足隐式实现的鸭子类型

这种设计可以让你创建一个新的接口类型满足已经存在的的具体类型却不用去破坏这些类型原有的定义，当我们使用的类型来自于不受我们控制的包时，这种设计尤其灵活有用。

接口在Go语言中无处不在

```
func Fprintf(w io.Writer, format string, args ...interface{}) (int, error)

//用于输出的接口
type io.Writer interface {
  Write(p []byte) (n int,err error)
}

//内置的错误接口
type error interface {
  Error() string
}

//下面的代码我们可以通过定制自己的输出对象，将每个字符转为大写字符后输出
type UpperWriter struct {
  io.Writer
}

func (p *UpperWriter) Write(data []byte) (n int, err error) {
return p.Writer.Write(bytes.ToUpper(data))
}

  func main() {
  fmt.Fprintln(&UpperWriter{os.Stdoout}, "hello, world")
}
```

Go语言中，对于基础类型（非接口类型）不支持隐式转换，我们无法将一个int类型的值直接赋给int64类型的变量，也无法将int类型的值赋值给底层是int类型的新定义命名类型的变量。

虚拟继承

```
type animal interface {
func hello()
func run()
}

    type xxx struct {
  animal //加入了接口，但是没有实现具体的方法，实现了虚拟继承
}
```

Goroutine和系统线程

线程

每个系统级的线程都会有一个固定大小的栈（默认2MB），这个栈主要用来保存函数调用时的参数和局部变量。

固定了栈的大小导致了两个问题，一个是对于只需要很小的栈空间的线程，是一种浪费，另一个是对于少数需要巨大栈空间的线程存在栈溢出的风险。Goroutine解决了这个问题。

协程

一个Goroutine会以一个很小的的栈启动（可能是2kb或4kb），当遇到深度递归导致当前栈空间不足时，Goroutine会根据需要动态的伸缩栈的大小，因为启动的代价很小，所以我们可以轻易的启动成千上万个Goroutine。

Go的运行时还包含了自己的调度器，这个调度器使用了一些技术手段，可以在n个操作系统上多工调度m个Goroutine。Go调度器的工作和内核的调度是相似的，但是这个调度器只关注单独的Go程序中的Goroutine。

Goroutine采用半抢占式的协作调度，只有在当前Goroutine发生阻塞时才会调度；同时发生在用户态，调度器会根据具体函数只保存必要的寄存器，切换的大家要比系统级线程低的多

原子操作

原子操作是指并发编程中，最小且不可并行化的操作。

如果多个并发体对同一个共享资源进行的操作是原子的话，那么同一时刻最多只能有一个并发体对该资源进行操作。

一般情况下原子的操作都是通过互斥来保证的，如下代码

互斥锁实现原子操作

```
package main

import (
"fmt"
"sync"
)



var total struct{
 sync.Mutex
 value int
}

func worker(wg *sync.WaitGroup) {
defer wg.Done()
for i:=0; i<100; i++ {
  total.Lock()
  total.value += 1
  total.Unlock()
 }
}



func main() {
var wg sync.WaitGroup
 wg.Add(2)
go worker(&wg)
go worker(&wg)
 wg.Wait()

    fmt.Println(total.value)
}
```

使用golang中自带的atomic

```
package main

import (
"fmt"
"sync"
"sync/atomic"
)



var total uint64

func worker(wg *sync.WaitGroup) {
defer wg.Done()



for i:=0; i<100; i++ {
  total = atomic.AddUint64(&total, 1)
 }
}

func main() {
 wg := &sync.WaitGroup{}
 wg.Add(2)
go worker(wg)
go worker(wg)
 wg.Wait()

   fmt.Println(total)
}
```

利用atomic和互斥锁实现单例模式

```
package main

import (
"sync"
"sync/atomic"
)



type singleton struct {

}

var (
 instance *singleton
 initialized uint32
 mu sync.Mutex
)

func Instance() *singleton{
if atomic.LoadUint32(&initialized) == 1 {
return instance
 }



 mu.Lock()
 defer mu.Unlock()



if instance == nil {
  defer atomic.StoreUint32(&initialized, 1)
  instance = &singleton{}
 }

 return instance
}
```

利用golang内置库Once实现单例模式

```
var (
 instance *singleton
  once sync.Once
)

func Instance() *singleton {
  Once.Do(func() {
    instance = &singleton
  })
  return instance
}
```

并发模型

> 并发不是并行，并发更多关注的是程序设计层面，并发的程序完全可以顺序执行，只有在真正的多核cpu上才可能真正的同时运行

不要通过共享内存来通信，而用通信来共享内存

在并发编程中，对共享资源的正确访问需要精确的控制，在目前绝大多数语言中，都是通过加锁等线程同步方案来解决这一问题，在go语言中将共享的值通过channel传递，在任意给定时刻，最好只有一个goroutine能够拥有该资源，数据竞争从设计层面就被杜绝了。

生产者消费者模型

```
package main

import (
"fmt"
"time"
)



func Producer(basic int,c chan int) {
 res := basic*2
 c <- res
}

func Consumer(c chan int) {
for v := range c {
  fmt.Println(v)
 }



}

func main() {
 c := make(chanint)
go Producer(2, c)
go Producer(3,c)

    go Consumer(c)
 time.Sleep(time.Second)
}
```

发布订阅模型

```
package main

import (
"fmt"
"strings"
"sync"
"time"
)



type (
 subscriber chaninterface{}
 topicFunc func(v interface{}) bool
)



type Publisher struct {
 m sync.Mutex //读写锁
 buffer int//订阅队列的缓存大小
 timeout time.Duration //发布超时时间
 subscribers map[subscriber] topicFunc //订阅者信息
}



// 构建一个发布者对象，可以设置发布超时时间和缓存队列的长度
func NewPublisher(publishTimeout time.Duration, buffer int) *Publisher {
return &Publisher{
  buffer: buffer,
  timeout: publishTimeout,
  subscribers: make(map[subscriber]topicFunc),
 }
}



//添加一个订阅者，订阅全部主题
func (p *Publisher) Subscribe() chan interface{}{
return p.SubscribeTopic(nil)
}



// 添加一个订阅者，订阅过滤器筛选后的主题
func (p *Publisher) SubscribeTopic(topic topicFunc) chan interface{} {
 ch := make(chaninterface{}, p.buffer)
 p.m.Lock()
 p.subscribers[ch] = topic
 p.m.Unlock()



return ch
}

// 退出订阅
func (p *Publisher) Exit(sub chan interface{}) {
 p.m.Lock()
defer p.m.Unlock()
```

```
delete(p.subscribers, sub)
close(sub)
}



// 发布一个主题
func (p *Publisher) Publish(v interface{}) {
 p.m.Lock()
defer p.m.Unlock()



var wg sync.WaitGroup
for sub, topic := range p.subscribers {
  wg.Add(1)
go p.sendTopic(sub, topic, v, &wg)
 }



 wg.Wait()
}

// 关闭发布者对象，同时关闭所有订阅者对象
func (p *Publisher) Close() {
 p.m.Lock()
defer p.m.Unlock()



for sub := range p.subscribers {
delete(p.subscribers, sub)
close(sub)
 }
}



// 发送主题， 可以容忍一定超时
func (p *Publisher) sendTopic(
 sub subscriber, topic topicFunc, v interface{}, wg *sync.WaitGroup,
 ) {
defer wg.Done()
if topic != nil && !topic(v) {
return
 }



select {
case sub <- v:
case <-time.After(p.timeout):
 }
}
```

```
func main() {
 p := NewPublisher(100 * time.Millisecond, 10)
defer p.Close()



 all := p.Subscribe()
 golang := p.SubscribeTopic(func (v interface{}) bool {
if s, ok := v.(string); ok {
return strings.Contains(s, "golang")
  }



returnfalse
 })



 p.Publish("hello, world")
 p.Publish("hello, golang")



gofunc() {
for msg := range all {
   fmt.Println("all:", msg)
  }
 }()



gofunc() {
for msg := range golang {
   fmt.Println("golang:", msg)
  }
 }()

    time.Sleep(3 * time.Second)
}
```

Select

基于select实现管道的超时判断

```
select {
  case v := <- in :
   fmt.Println(v)
  case <-time.After(time.Second):
   return //超时
}
```

基于select的default分支实现非阻塞的管道发送或接收操作

```
select {
 case v := <-in :
   fmt.Println(v)
  default:
   //无数据
}
```

通过select来阻止main退出

```
func main() {
  select{}
}
```

基于select来实现随机数

```
func main() {
 ch := make(chan int)
 go func() {
for {
   select {
case ch <- 0:
case ch <- 1:
   }
  }
 }()

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

基于select实现goroutine的退出

```
package main

import (
"fmt"
"time"
)



func worker(close chan int) {
for {
  select {
case  <- close:
break
   default:
    fmt.Println(123)



  }
 }
}

func main() {
 c := make(chan int)
 go worker(c)
 time.Sleep(time.Second)
 c <- 1
}
```

基于select实现goroutine退出 V2

```
package main

import (
"fmt"
"time"
)



func worker(close chan int) {
for {
  select {
case  <- close:
break
   default:
    fmt.Println(123)



  }
 }
}

func main() {
 c := make(chan int)
 for i:=0; i< 100; i++ {
  go worker(c)
 }
 time.Sleep(time.Second)
 close(c)
}
```

由于管道的接收和发送是一一对应的，如果要停止多个goroutine那么需要创建同样数量的管道，性能差，所以可以通过一个管道来实现广播的操作。

基于select实现goroutine退出 V3

```
package main

import (
"fmt"
"sync"
"time"
)



func worker(wg *sync.WaitGroup, close chan int) {
 defer wg.Done()



for {
  select {
case  <- close:
return
   default:
    fmt.Println(123)



  }
 }
}

func main() {
 c := make(chan int)
 wg := &sync.WaitGroup{}
for i:=0; i< 100; i++ {
  wg.Add(1)
  go worker(wg, c)
 }
 time.Sleep(time.Second)
 close(c)

  wg.Wait()
}
```

当每个goroutine收到退出指令时，一般会进行一定的清理工作，但是退出清理的工作无法保证一定完成，因为main线程没有等待各个goroutine工作完成的机制，所以结合sync.WaitGroup增强代码的健壮性

Context

```
package main

import (
"context"
"fmt"
"sync"
"time"
)



func worker(ct context.Context, wg *sync.WaitGroup) error{
 defer wg.Done()



for {
  select {
   default:
    fmt.Println(123)
case <-ct.Done():
return ct.Err()
  }
 }
}



func main() {
 wg := &sync.WaitGroup{}
 ct, cancel := context.WithTimeout(context.Background(), time.Second)



for i:=0; i<100; i++ {
  wg.Add(1)
  go worker(ct, wg)
 }

 time.Sleep(time.Second)
 cancel()
 wg.Wait()
}
```

错误和异常

错误处理策略

```
package main

import"fmt"

 func main() {
 defer func() {
  if p := recover(); p != nil {
   fmt.Printf("%v", p)
  }
 }()
 panic("test")
}
```

捕获异常不是最终目的，如果异常不可预测，直接输出异常信息是最好的处理方式。

RPC

RPC基本示例

服务器端

```
package main

import (
"fmt"
"net"
"net/rpc"
)



type HelloService struct {

}

func (this *HelloService) Hello(name string,reply *string) error {
 *reply = "hello" + name



return nil
}

func main() {
 rpc.RegisterName("HelloService", new(HelloService))

 listener, err := net.Listen("tcp", ":6062")
if err != nil {
  panic(err)
 }

  conn,err := listener.Accept()
 if err != nil {
  panic(err)
 }
 fmt.Printf("listening")
 rpc.ServeConn(conn)
}
```

客户端

```
package main

import (
"fmt"
"net/rpc"
)



func main() {
 conn,err := rpc.Dial("tcp",":6062")
if err != nil {
panic(err)
 }
var reply string
 err = conn.Call("HelloService.Hello","world", &reply)
if err != nil {
panic(err)
 }

        fmt.Printf("%v",reply)
}
```

深入了解RPC

客户端rpc实现原理

Call

```
/**
通常我们在客户端调用rpc框架中的call方法进行同步阻塞调用
该方法首先通过client.Go方法进行了一次异步调用,返回一个表名这次调用的Call结构体。
然后等待Call结构体的Done
**/
func (client *Client) Call(serviceMethod string, args interface{}, reply interface{
 call := <-client.Go(serviceMethod, args, reply, make(chan *Call, 1)).Done
 return call.Error
}
```

Go

```
/**
首先构造了一个表示当前调用的call变量
然后通过client.send方法将call的完整参数发送到rpc框架，send方法是线程安全的
当调用完成或者发生错误时，将调用call.done()方法
**/
func (client *Client) Go(serviceMethod string, args interface{}, reply interface{},
 call := new(Call)
 call.ServiceMethod = serviceMethod
 call.Args = args
 call.Reply = reply
 if done == nil {
  done = make(chan *Call, 10) // buffered.
 } else {
  // If caller passes done != nil, it must arrange that
  // done has enough buffer for the number of simultaneous
  // RPCs that will be using that channel. If the channel
  // is totally unbuffered, it's best not to run at all.
  if cap(done) == 0 {
   log.Panic("rpc: done channel is unbuffered")
  }
 }
 call.Done = done
 client.send(call)
 return call
}
```

Done

```
func (call *Call) done() {
 select {
 case call.Done <- call:
  // ok
 default:
  // We don't want to block here. It is the caller's responsibility to make
  // sure the channel has enough buffer space. See comment in Go().
  if debugLog {
   log.Println("rpc: discarding Call reply due to insufficient Done chan capacity")
  }
 }
}
```

Protobuf

> Protobuf作为接口规范的描述语言，可以作为设计安全的跨语言RPC接口的基础工具

grpc

> 如果从Protobuf的角度看，gRPC只不过是一个针对service接口生成代码的生成器

示例

```
syntax = "proto3";

package hello;

message String {
string value = 1;
}

    service HelloService {
    rpc Hello(String) returns (String);
}
```

运行

```
protoc --proto_path=. --go_out=plugins=grpc:. hello.proto
```

生成

```
func RegisterHelloServiceServer(s *grpc.Server, srv HelloServiceServer) {
 s.RegisterService(&_HelloService_serviceDesc, srv)
}

func NewHelloServiceClient(cc grpc.ClientConnInterface) HelloServiceClient {
return &helloServiceClient{cc}
}



// HelloServiceServer is the server API for HelloService service.
type HelloServiceServer interface {
 Hello(context.Context, *String) (*String, error)
}

// HelloServiceClient is the client API for HelloService service.
//
// For semantics around ctx use and closing/ending streaming RPCs, please refer to
type HelloServiceClient interface {
 Hello(ctx context.Context, in *String, opts ...grpc.CallOption) (*String, error)
}
```

重写HelloService服务端

```
package main

import (
 hello "code.yuhaos.com/studygo/grpc/protobuf"
"context"
"google.golang.org/grpc"
"net"
)



type HelloService struct {

}

func (p *HelloService) Hello(ctx context.Context, name *hello.String) (*hello.Strin
 var ret hello.String
 ret.Value = "hello" + name.Value
return &ret, nil
}



func main() {
 grpcServer := grpc.NewServer()
 hello.RegisterHelloServiceServer(grpcServer, new(HelloService))

 listener, err := net.Listen("tcp", ":1234")
if err != nil {
  panic(err)
 }

  grpcServer.Serve(listener)
}
```

重写HelloService客户端

```
package main

import (
 hello "code.yuhaos.com/studygo/grpc/protobuf"
"context"
"fmt"
"google.golang.org/grpc"
)



func main() {
 conn, err := grpc.Dial(":1234",grpc.WithInsecure())
if err != nil {
  panic(err)
 }
 defer conn.Close()



 client := hello.NewHelloServiceClient(conn)
 res, err := client.Hello(context.Background(), &hello.String{Value:"world"})
if err != nil {
  panic(err)
 }

  fmt.Println(res.Value)
}
```

gRPC流

> RPC是远程调用，因此每次调用的函数参数和返回值不能太大，否则将严重影响每次调用的响应时间，因此传统的RPC方法调用对于上传和下载大数据量的场景并不合适，为此gRPC框架针对服务器和客户端分别提供了流特性

protobuf

```
syntax = "proto3";

package hello;

message String {
    string value = 1;
}

service HelloService {
    rpc Hello(String) returns (String);
    rpc Channel (stream String) returns (stream String);
}
/**
HelloSerivceServer和HelloServiceClient两个接口都添加了Channel方法的定义
该方法可以用于客户端和服务端的双向通信
**/
type HelloServiceServer interface {
 Hello(context.Context, *String) (*String, error)
 Channel(HelloService_ChannelServer) error
}

type HelloServiceClient interface {
 Hello(ctx context.Context, in *String, opts ...grpc.CallOption) (*String, error)
 Channel(ctx context.Context, opts ...grpc.CallOption) (HelloService_ChannelClient,
}

/**
HelloService_ChannelServer和HelloService_ChannelClient都为接口
**/

type HelloService_ChannelServer interface {
 Send(*String) error
 Recv() (*String, error)
 grpc.ServerStream
}

type HelloService_ChannelClient interface {
 Send(*String) error
 Recv() (*String, error)
 grpc.ClientStream
}
```

基于流的Server

```
package main

import (
 hello "code.yuhaos.com/studygo/grpc/protobuf"
"context"
"google.golang.org/grpc"
"io"
"net"
)



type HelloService struct {

}

func (p *HelloService) Hello(ctx context.Context, name *hello.String) (*hello.Strin
 var ret hello.String
 ret.Value = "hello" + name.Value
return &ret, nil
}



func (p *HelloService) Channel(stream hello.HelloService_ChannelServer) error {
for {
  args, err := stream.Recv()
if err != nil {
if err == io.EOF {
return nil
   }
return err
  }



  reply := &hello.String{Value:"Hello" + args.GetValue()}
  err = stream.Send(reply)
if err != nil {
return err
  }
 }
}



func main() {
 grpcServer := grpc.NewServer()
 hello.RegisterHelloServiceServer(grpcServer, new(HelloService))

 listener, err := net.Listen("tcp", ":1234")
if err != nil {
  panic(err)
 }

  grpcServer.Serve(listener)
}
```

基于流的client

```
package main

import (
 hello "code.yuhaos.com/studygo/grpc/protobuf"
"context"
"fmt"
"google.golang.org/grpc"
"io"
"time"
)



func main() {
 conn, err := grpc.Dial(":1234",grpc.WithInsecure())
if err != nil {
  panic(err)
 }
 defer conn.Close()



 client := hello.NewHelloServiceClient(conn)
 stream, err := client.Channel(context.Background())
if err != nil {
  panic(err)
 }



 go func() {
for {
if err := stream.Send(&hello.String{Value:"hi"}); err != nil {
    panic(err)
   }
   time.Sleep(time.Second)
  }
 }()



for {
  res, err := stream.Recv()
if err != nil {
if err == io.EOF {
break
   }
   panic(err)
  }

           fmt.Println(res.GetValue())
 }
}
```

基于docker的pub/sub与grpc流实现发布订阅模型

protobuf

```
syntax = "proto3";

package pub_sub_service;

message String {
    string value=1;
}

service PubsubService {
    rpc Publish(String) returns (String);
    rpc Subscribe(String) returns (stream String);
}
```

service

```
package main

import (
 pub_sub_service "code.yuhaos.com/studygo/grpc-pub-sub/protobuf"
"context"
"github.com/docker/docker/pkg/pubsub"
"google.golang.org/grpc"
"net"
"strings"
"time"
)



type PubsubService struct {
 pub *pubsub.Publisher
}

func NewPubSubService() *PubsubService{
return &PubsubService{
  pub:pubsub.NewPublisher(100*time.Millisecond, 10),
 }
}



func (p *PubsubService) Publish(ctx context.Context, arg *pub_sub_service.String) (
{
 p.pub.Publish(arg.GetValue())
return &pub_sub_service.String{}, nil
}



func (p *PubsubService) Subscribe(arg *pub_sub_service.String, stream pub_sub_servi
 ch := p.pub.SubscribeTopic(func(v interface{}) bool {
if key, ok := v.(string); ok {
if strings.HasPrefix(key, arg.GetValue()) {
returntrue
   }
  }
returnfalse
 })



for v := range ch {
if err := stream.Send(&pub_sub_service.String{Value:v.(string)}); err != nil {
return err
  }
 }



return nil
}

func main() {
 grpcServer := grpc.NewServer()

 pub_sub_service.RegisterPubsubServiceServer(grpcServer, NewPubSubService())
```

```
 listener, err := net.Listen("tcp", ":1235")
if err != nil {
  panic(err)
 }

  grpcServer.Serve(listener)
}
```

client_pub

```
package main

import (
 pub_sub_service "code.yuhaos.com/studygo/grpc-pub-sub/protobuf"
"context"
"google.golang.org/grpc"
)



func main() {
 conn, err := grpc.Dial("localhost:1235", grpc.WithInsecure())
if err != nil {
  panic(err)
 }
 defer conn.Close()



 client := pub_sub_service.NewPubsubServiceClient(conn)

 _, err = client.Publish(context.Background(), &pub_sub_service.String{Value:"golan
if err != nil {
  panic(err)
 }

  _, err = client.Publish(context.Background(), &pub_sub_service.String{Value:"php:
 if err != nil {
  panic(err)
 }
}
```

client_sub

```
package main

import (
 pub_sub_service "code.yuhaos.com/studygo/grpc-pub-sub/protobuf"
"context"
"fmt"
"google.golang.org/grpc"
"io"
)



func main() {
 conn, err := grpc.Dial(":1235", grpc.WithInsecure())
if err != nil {
  panic(err)
 }
 defer conn.Close()



 client := pub_sub_service.NewPubsubServiceClient(conn)
 stream, err := client.Subscribe(context.Background(), &pub_sub_service.String{Valu
if err != nil {
  panic(err)
 }

  for {
  reply, err := stream.Recv()
  if err != nil {
   if err == io.EOF {
    break
   }
   fmt.Printf("err")
   continue
  }
  fmt.Println("reply:%v", reply.GetValue())
 }
}
```

Go-Web

httprouter

路由冲突

```
conflict:
GET /user/info/:name
GET /user/:id

no conflict:
GET  /user/info/:name
POST /user/:id
```

因为httprouter使用的是显示匹配，所以在设计路由时需要规避一些会导致路由冲突的情况

如果两个路由拥有一致的http方法和请求路径前缀，切在某个位置出现了A路由是wildcard(:id)参数，B路由是普通字符串那么就会发生路由冲突

对特殊情况进行定制

```
package main

import (
"github.com/julienschmidt/httprouter"
"net/http"
)



func main() {
 r := httprouter.New()

 r.NotFound = http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
  w.Write([]byte("404"))
 })

 r.PanicHandler = func(writer http.ResponseWriter, request *http.Request, i interfa
  writer.Write([]byte("500"))
 }

 http.ListenAndServe(":8081", r)

}
```

原理

HTTProuter和众多衍生router使用的数据结构被称为压缩字典树。

发生路由冲突的情况也是由于构建压缩字典树时造成的。

中间件

```
package main

import (
"github.com/sirupsen/logrus"
"net/http"
)



func LogMiddleware(next http.Handler) http.Handler{
return http.HandlerFunc(func (w http.ResponseWriter, r *http.Request){
  logrus.Println("test")
  next.ServeHTTP(w, r)
 })
}



func Hello(w http.ResponseWriter, r *http.Request) {
 w.Write([]byte("hello"))
}

func main() {

 http.Handle("/test", LogMiddleware(http.HandlerFunc(Hello)))



 http.ListenAndServe(":8081", nil)

}
```

中间件的原理

```
//中间件的原理在于http.handle方法中需要传入Handler参数
//handler为一个接口，既只要实现了ServeHTTP方法的即为Handler
type Handler interface {
 ServeHTTP(ResponseWriter, *Request)
}

//HandlerFunc可以将普通的方法强转为Handler类型
type HandlerFunc func(ResponseWriter, *Request)
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
 f(w, r)
}
```

更优雅的使用中间件

```
package main

import (
"github.com/sirupsen/logrus"
"net/http"
)



type middleware func(http.Handler) http.Handler

type Router struct {
 middlewareChain [] middleware
 mux map[string] http.Handler
}

func NewRouter() *Router {
return &Router{
  middlewareChain:[]middleware{},
  mux: make(map[string] http.Handler),
 }
}



func (r *Router) Use(m middleware) {
 r.middlewareChain = append(r.middlewareChain, m)
}

func (r *Router) Add(route string, h http.Handler) {
 var mergeHandler = h



for i := len(r.middlewareChain)-1; i>=0; i-- {
  mergeHandler = r.middlewareChain[i](mergeHandler "i")
 }

 r.mux[route] = mergeHandler
}

func (r *Router) ServeHTTP(writer http.ResponseWriter, request *http.Request) {
}

func LogMiddleware(next http.Handler) http.Handler{
return http.HandlerFunc(func (w http.ResponseWriter, r *http.Request){
  logrus.Println("test")
  next.ServeHTTP(w, r)
 })
}



func Hello(w http.ResponseWriter, r *http.Request) {
 w.Write([]byte("hello"))
}

func main() {
```

```
 r := NewRouter()
 r.Use(LogMiddleware)
 r.Add("/test", http.HandlerFunc(Hello))

 http.ListenAndServe(":8081", r)

}
```

服务流量限制

问题

有些程序偏网络IO瓶颈，例如CDN服务，Proxy服务

有些程序偏CPU/GPU瓶颈，例如登录校验服务，图像处理服务

有些程序偏磁盘，例如存储系统，数据库

对于IO/Network瓶颈类的程序，其表现是网课/磁盘IO会先于CPU打满，这种情况即使优化CPU的使用也无法提高整个系统的吞吐量，只有提高磁盘的读写速度，增加内存大小，提升网卡带宽来提升整体性能

漏桶限流

> 我们有一个，每过一段时间，向外漏一滴水，如果你接到了这滴水，那么就可以继续服务请求，否则需要等待下一滴水

令牌桶

> 匀速向桶中添加令牌，服务请求时需要从桶中获取令牌，令牌的数目可以按照需要消耗的资源进行相应的调整，如果没有令牌，可以选择等待或者放弃。该模型支持并发。

```
/**
令牌桶模拟
**/
package main

import (
"fmt"
"time"
)



func main() {
 var fillInterval = time.Millisecond * 10
 var capacity = 100
 var tokenBucket = make(chan struct{}, capacity)

 fillToken := func() {
  ticker := time.NewTicker(fillInterval)
for {
   select {
case <-ticker.C:
     select {
case tokenBucket <- struct{}{}:
     default:
     }
     fmt.Println("current token cnt:", len(tokenBucket), time.Now())
   }
  }
 }

            go fillToken()
 time.Sleep(time.Hour)
}
/**
模拟取令牌
**/
TakeAvailable := func(block bool) bool {
  var takenResult bool
  if block {
   select {
   case <-tokenBucket:
    takenResult = true
   }
  } else {
   select {
   case <- tokenBucket:
    takenResult = true
   default:
    takenResult = false
   }
  }
```

上述方法是一般令牌桶的实现，下面是优化算法

```
/**
思考一下，令牌桶每隔一段固定时间向桶中放令牌，我们记下上一次放令牌的时间为t1和当时的令牌数为k1，放令牌的间隔时间为ti，每次向桶中放x个令牌，令牌容量为cap。现在如果有人来获取令牌时间为t2，在t2时刻应该存在的令牌为多少呢？
**/
cur = k1 + ((t2-t1)/ti)*x
cur = cur > cap ? cap : cur
```

通过这种方式，只需要在取令牌的时候去获取令牌数量，在得到正确的令牌数之后，再进行实际的take即可。

分布式系统

分布式ID-雪花算法

首先确定数值是64位，将其划分为4个部分，不含开头的第一个bit，这个bit是符号位

第一部分，用41位来代表收到请求的时间戳，单位为毫秒

第二部分，用5位来表示数据中心id

第三部分，用5位来表示机器实例id

第四部分，用12位的循环自增id

这样一台机器，同一毫秒可以生产 2 ^ 12 = 4096条消息，一秒409.6万条

数据中心加上实例id共10位，可以支持我们每个数据中心部署32台机器，所有数据中心1024台实例

分布式锁

redis setnx

基于zooKeeper

```
func main() {
  c, _, err := zk.Connect([]string{"127.0.0.1"}, time.Second)
if err != nil {
    panic(err)
  }



  l := zk.NewLock(c, "/lock", zk.WorldACL(zk.PermAll))
  err = l.Lock()
if err != nil {
    panic(err)
  }
  //lock succ

      l.Unlock()
  //unlock succ
}
```

基于zooKeeper的锁与基于redis的锁的不同之处在于Lock成功之前会一直阻塞

该种方式的原理是基于临时的Sequence节点和Watch API。

比如使用了/lock节点，Lock会在该节点下的节点列表中插入自己的值，只要节点下的子节点发生变化，就会通知所有watch该节点的程序。这时候程序会检查当前节点下最小的子节点的id是否与自己的一致。如果一致说明加锁成功了。

这种分布式的阻塞锁比较适合分布式的任务调度场景，但不适合高频次的持锁时间短的抢锁场景。

Google的Chubby论文里的阐述，基于强一致协议的锁适用于粗粒度的加锁操作。这里的粗粒度指锁的占用时间长。

基于etcd

延时任务系统

时间堆

小顶堆

小顶堆就是一种特殊的二叉树，对于定时器来讲，如果堆顶的元素比当前时间还大，说明堆内的所有元素都比当前时间打

四叉堆

Go内置的定时器利用了四叉堆

时间轮

用时间轮来实现定时器时，我们需要定义每一个格子的刻度，中心有秒针顺时针转动。每转动到一个刻度时，我们就需要去查看该刻度挂载的任务列表是否有已到期的任务。

任务分发

每一个实例，每个小时就去数据库里把下一个小时需要处理的任务捞出来。

可以通过：

1. 将任务触发的信息封装为一条消息，发送到消息队列，由用户对消息队列进行监听。
2. 对用户预先配置的回调函数进行调用。

数据再平衡

当一台实例出现故障时，我们可以通过类似es的方式将数据同步到副本节点以保证数据再平衡。

负载均衡

基于洗牌算法的负载均衡

> 设计y一个大小和节点数组一致的索引素组，每次新的请求来的时候，我们对索引数组洗牌，然后取第一个元素作为选中的服务节点，如果请求失败，选择下一个节点重试。

算法1

```
var endpoints = []string {
"xxx.xxx.xxx.x:1",
"xxx.xxx.xxx.x:2",
"xxx.xxx.xxx.x:3",
"xxx.xxx.xxx.x:4",
}

               func shuffle(slice []int) {
  for i:=0; i<len(slice); i++ {
    a := rand.Intn(len(slice))
    b := rand.Intn(len(slice))
    slice[a], slice[b] = slice[b], slice[a]
  }
}
```

算法2（更均衡）

```
func shuffle(indexs []int) {
  for i:=len(indexs); i>0; i-- {
    lastIdx := i-1
    idx := rand.Int(i)
    indexs[lastIdx], indexs[idx] = indexs[idx], indexs[lastIdx]
  }
}
```

算法2简化

```
func shuffle(n int) []int {
  b := rand.Perm(n)
  return b
}
```

分布式配置管理

基于etcd实现

优点

开箱即用

缺点

不利于管理，无法承载高qps

Go语言中使用时的坑

可变参数是空接口类型

当参数的可变参数是空接口类型时，传入空参数的切片时需要注意参数展开的问题

```
func main() {
  var a = []interface{}{1,2,3}
  fmt.Println(a) // [1,2,3]
  fmt.Println(a...) // 1 2 3
}
```

数组是值传递

在函数调用的过程中，数组是值传递

```
func main() {
 x := [3]int{1,2,3}

 func (arr [3]int) {
for i:=0; i<len(arr); i++ {
   arr[i] = arr[i]*2
  }
 }(x)

   fmt.Println(x) //1 2 3
}
```

map遍历是顺序不固定的

map是一种hash表实现，每次遍历的顺序都可能不一样

recover必须在defer函数中运行

```
func main() {
  defer func () {
    recover()
  }()
  panic("recover test")
}
```

独占cpu导致其他的goroutine饿死

Goroutine是协作式抢占调度，Goroutine本身不会主动放弃CPU:

```
func main() {
  runtime.GOMAXPROCS(1)



  go func() {
for i:=0; i<10; i++ {
      fmt.Println(i)
    }
  }()

        for {} //占用cpu
}
```

解决方案1：利用runtime.Gosched()

```
func main() {
  runtime.GOMAXPROCS(1)



  go func() {
for i:=0; i<10; i++ {
      fmt.Println(i)
    }
  }()

        for {
    runtime.Gosched()
  }
}
```

解决方案2: 通过阻塞方式

```
func main() {
  runtime.GOMAXPROCS(1)



  go func() {
for i:=0; i<10; i++ {
      fmt.Println(i)
    }
    os.Exit(0)
  }()

        select{}
}
```

不同的goroutine之间不满足顺序一致性内存模型

```
package main

import "fmt"

var msg string
var done bool

func setup() {
 msg = "hello"
done = true
}



func main() {
 go setup()
for !done {



 }

 fmt.Println(msg)
}
```

在该程序中，对于main来说msg=”hello”和done=true的执行顺序是不确定的，所以可能无法输出hello，做以下修改保证了一致性

```
package main

import "fmt"

var msg string
var done = make(chan bool)

func setup() {
 msg = "hello"
done <- true
}



func main() {
 go setup()
 <-done

 fmt.Println(msg)
}
```

闭包错误引用同一个变量

```
func main() {
  for i:=0; i<5; i++ {
    defer func() {
      println(i)
    }()
  }
}
//5 5 5 5 5
```

修正后

```
//每轮迭代中生成一个局部变量通过参数传入
func main() {
  for i:=0; i<5; i++ {
    defer func() {
      println(i)
    }(i)
  }
}
```

在循环内部执行defer语句

defer在循环退出时才能执行，在for执行defer会导致资源延迟释放

```
func main() {
  for i:=0; i<5; i++ {
    f, err := os.Open('/path/to/file')
    if err != nil {
      log.Fatal(err)
    }
    defer f.Close()
  }
}
```

可以通过在for中构造一个局部函数，在局部函数内部defer

```
func main() {
  for i:=0; i<5; i++ {
    func() {
      f, err := os.Open('/path/to/file')
     if err != nil {
       log.Fatal(err)
     }
     defer f.Close()
    }()
  }
}
```

切片会导致整个底层数组被锁定

切片会导致整个底层数组被锁定，底层数组无法释放内存。

```
func main() {
  headerMap := make(map[string][]byte)
  for i:=0; i<5; i++ {
    name := "/path/to/file"
    data, err := ioutil.ReadFile(name)
    if err != nil {
      log.Fatal(err)
    }
    headerMap[name]  = data[:1] //导致底层数组被锁定
  }
}
```

优化，将结果克隆一份

```
func main() {
headerMap := make(map[string][]byte)
  for i:=0; i<5; i++ {
    name := "/path/to/file"
    data, err := ioutil.ReadFile(name)
    if err != nil {
      log.Fatal(err)
    }
    headerMap[name] = append([]byte{}, data[:1])
  }
}
```

欢迎关注Go生态。生态君会不定期分享 Go 语言生态相关内容。

Go生态
专注分享Go语言相关技术生态

3篇原创内容

公众号