# go语言编程

## Tips

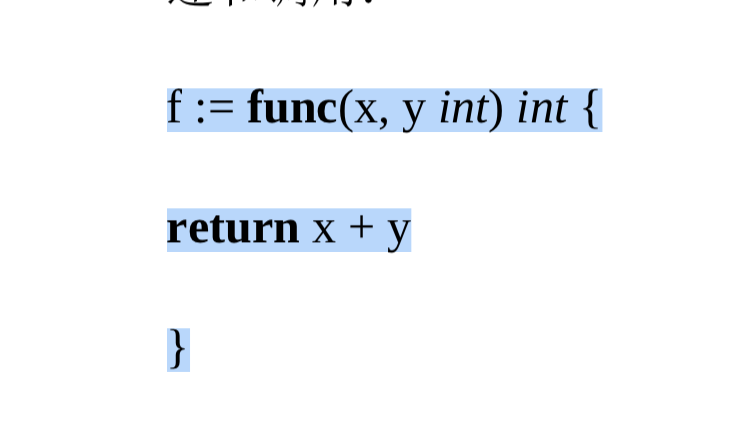
由于java的线程是和内核线程一对一的,那么java线程的运行与否,阻塞与否都是会被内核的线程调度所影响的.如果操作系统不然这个内核线程执行了,那么势必会影响到与之关联的java线程,而对于go的协程,协程是分配在线程上执行的,一旦这个线程不行了,协程就会分配到其他的线程去执行,反正就是无论内核你怎么折腾都不会影响我的协程的执行.

go语言没有方法重载.

go语言反对继承,反对虚函数,反对多态,也放弃了构造函数.

go语言的接口实现是没有侵入性的,你看java一个类要实现一个接口,必须要用关键字来声明才可以.而go语言呢,你只需需要这个结构体实现这个接口的所有的方法,那么这个结构体天就算作了这个接口的实现的.

还有就是比如有两个接口i1 和 i2 他俩都有函数read和write,那么需要i1的接口的实现类,你传个i2接口的实现类也是一样的.



在这里可以看到,匿名函数是可以用 :=func(){} 获取到一个函数的值的.

Go run xxx 执行完毕之后是看不到可执行文件的,因为都删掉了.

Go build是可以看到可执行文件的.

Go test -run 函数名

注意写test函数的时候函数名字一定是Test开头,参数是Test111(t \*testing.T)

%v 是按照类型的默认格式打印,什么都可以打印,%+v,的话,如果是结构体的话会把字段名字一起也打印出来.

go语言早期是通过makefile标记依赖的,现在通过import关键字就能处理依赖,进行正确的编译链接等操作,

Var v3[10] int 这就是定义了一个数组

Var v3[] int 这就是定义了一个切片 区别就是有没有长度.

Var xxx map[string]int

Var v3 function(a int) int 这就是定义了一个函数,你看定义函数的变量只需要确定参数和返回值就可以了.

go语言是强类型语言,只不过带了类型推导的功能.

多重赋值, i,j = 1,1 就是这样一下子就赋值了多个.

go语言还有复数类型, go语言也有uint32 uint8 uint64这种类型.

go语言是有那种位运算的,按位与 按位或 啥的 左移 右移啥的.

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)

字符串在go里面是基本类型,所以字符串的零值是“”空字符串,而不是nil

Len()函数是一个内置函数,就跟c语言的size()函数似的,并不是属于某一个类的函数,所以就是直接调用就行.

字符串操作

拼接就是直接用+号.

Len(“dd”) 获取字符串长度.

“对方的饭”[2] 获取字符串的索引值的字符 那么通过len和xx[]取某个字符的方法就可以遍历某个字符串的所有的字符了,但是要注意的是,字符串的字符是不能赋值的.

对于字符来说,可以 byte表示,也可以用rune来表示.

func main() {

arr := [4]int{1, 2, 3, 4}

注意这定义数组的时候就这样定义, 不要用var,用var定义的那种一般都是不直接赋值的,而这种直接赋值的都要用:=来定义,

注意在定义数组的时候一定要指定长度.

testArr(arr)

for \_, item := range arr {

fmt.Println(item)

}

}

func testArr(arr [4]int) {

arr[0] = 10

还有就是这里在接受的时候也要指定长度,并且和上面的长度是一样的才可以.

for \_, item := range arr {

fmt.Println(item)

}

}

上面这个例子证明了,数组 也就是长度是固定的东西,是按照值来传递的!!!!!!!!!!!!

package main

import "fmt"

func main() {

arr := []int{1, 2, 3, 4} 这样不指定长度的时候也是可以直接这样指定一个切片的

testArr(arr)

for \_, item := range arr {

fmt.Println(item)

}

}

func testArr(arr []int) {

arr[0] = 10

for \_, item := range arr {

fmt.Println(item)

}

}

而注意,一旦我们将长度去掉,那么数组就变成切片了,而切片不是值传递的

对于切片来说 make([]int, 5) 这就是创建了一个切片,没有指定用量,但是指定了长度,也就是这个切片的前5个元素已经被占用了,值全都是0啊

所以平时我们都是这个make([]int , 0, 10) 这样的话会创建一个容量为10的,并且元素都是没有使用过的切片.

对于切片来说是有cap()函数和len()函数的.

切片的append函数是可以是可以一次append多个元素的append(arr, a ,b c, d),也可以一次性就append一个切片,就这样 append(arr, arr2...)

go还可以copy()函数用来复制数组的!!!!! 把大的往小的复制还是把小的往大的复制都是可以的.

Mymap := Make(map[string]string, 100)

delete(myMap, “1234”) 这样删除一个key,如果key找不到的话不会有问题,但是key如果是nil的话就会panic

Switch 是不用break的,自动就会跳出的,想要用穿透用fallthough语句.

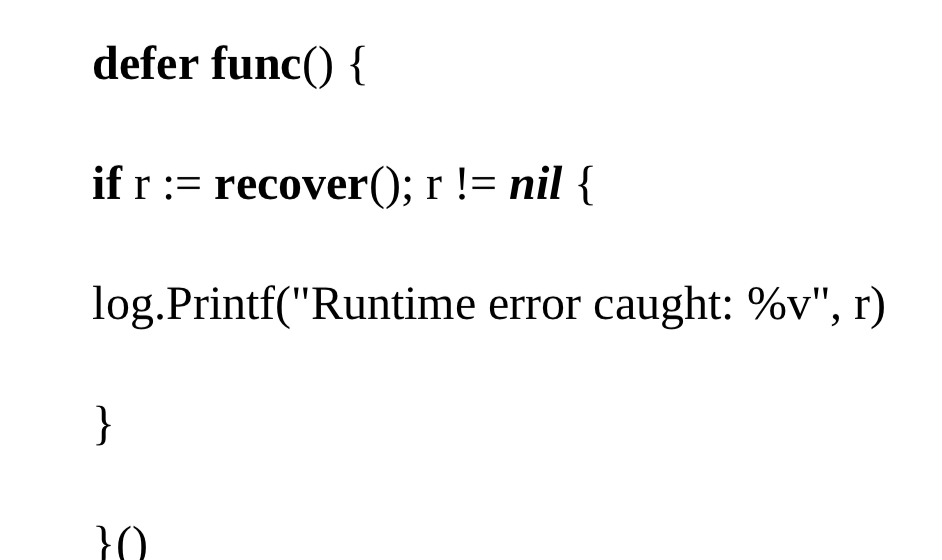
go语言的自定义的error只要实现了Error()函数就可以了,那么你这个结构体就可以当作一个类型为error的返回值了.

Defer 后面可以接一个表达式, 也可以接一个function啥的,都是可以的,

Panic()函数的参数类型是interface{},可以接受任何语句,一旦你调用了panic,那么么所属的goroutine所有的函数都会层层的被终止,其中的defer语句是会在panic之前被执行的

也就是即使你的程序panic了,defer语句还是会执行的,也就是panic并不会导致你的defer语句执行失败.

Recover()函数的作用呢,是和panic()函数相克的,他是会抓到panic的异常流(也就是干掉整个协程的企图的) 所以呢,recover最好是放在defer里面,这样的话可以保证被抓到,



## l并发

go的调度器还会有这种操作也就是如果一个线程下的其中一个goroutinue阻塞了,go会把这个线程下的所有携程移到其他的线程里面,避免这些携程也阻塞了.

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.

协程本质上是一种用户态线程,不需要操作系统来进行抢占式调度,且在真正的实现寄存于线程中,因此,系统开销极小,可以有效提高线程任务的并发性,而避免多线程的缺点(感觉就是在线程内部又进行了调度,保持cpu一直保持高效率的利用),.缺点就是需要语言的原生支持,如果不支持需要用户在程序中自行实现调度器,

消息传递系统和共享内存系统,其实不就是第一个,每一个线程都有一个自己的消息队列么,第二个就是通过共享内存(估计也就是锁对象之类的,搞的这么高大上)

协程的最大优势在于轻量级,可以轻松创建上百万个也不至于系统资源枯竭,而线程和进程最多也不能超过一万个.

多数语言在语法层面不支持携程,也就是无法通过关键字创建携程出来,必须要依赖别的库才能玩携程,而且用库的方式支持携程也不完整,仅仅能提供创建 切换 销毁 sleep的能力(比如mme的虚拟线程,不就是创建,切换 销毁么) 比如本地文件读写啊,网络io啊这些非cpu密集型的业务都会阻塞住这个携程进而阻塞住整个线程了.

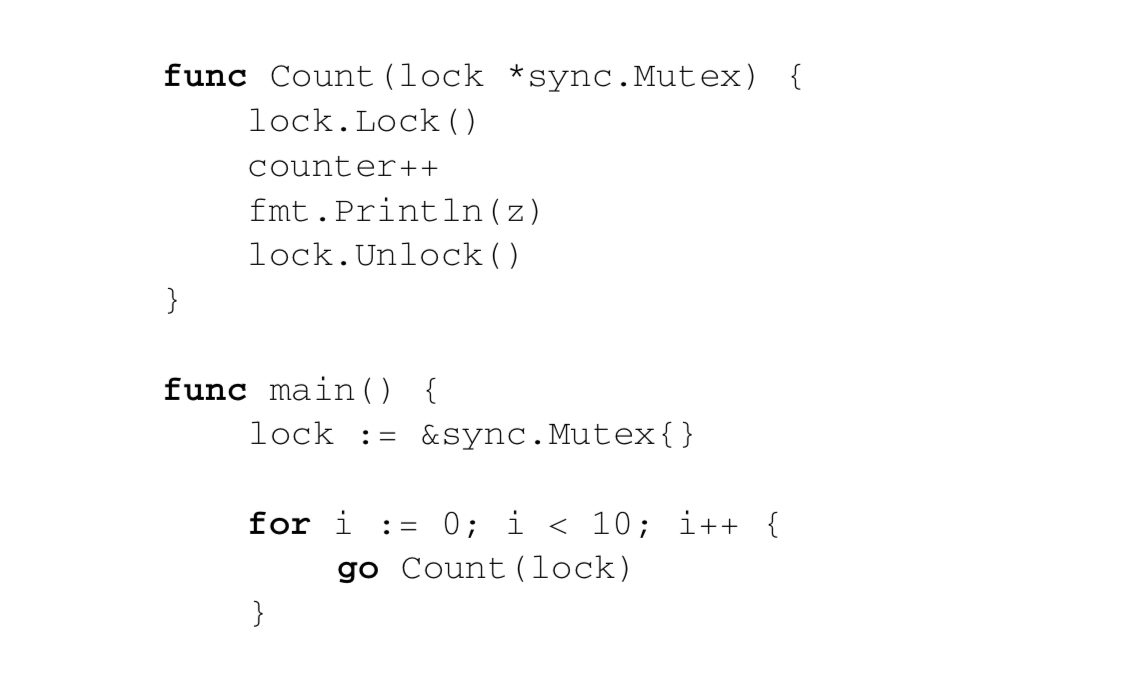
但是go的携程调度就不会出现这个问题,他会直接把执行权限让给其他的携程.

Go func func的返回值会被丢弃,如果想要玩那种带返回值的,为啥不用chan呢.

下面来一个c语言的多线程的简单例子



看到了里面又pthread\_create和 pthread\_join



可以看到go里面的lock := &sync.Mutex{} 这个Mutex结构体就是一个锁咯,这个结构体就有lock.Lock()方法和lokc.Unlock()方法咯.

**go语言社区的口号!!!!!!!!!!不要通过共享内存来通信,而应该通过通信来共享内存. 是啊,以前也怀疑过,用锁的把大量的线程给弄成阻塞状态是不是效率太低了,你还得调用操作系统的关键字,性能超级差,所以才有了后面的自旋锁,锁粗化等各种优化方式,如果你直接用chan来通信那不就是直接ok了.**

channel的消息传递类似于函数的参数传递,都是进程内的消息通信.channel是类型相关的,和unix的管道差不多,可以认为channel是一种类型安全的管道.

go语言的channel天生就是线程安全的，可以被多协程同时读写。

协同程序（coroutine）与多线程情况下的线程比较类似：有自己的堆栈，自己的局部变量，有自己的指令指针（IP，instruction pointer），但与其它协同程序共享全局变量等很多信息。

也就是操作系统执行起来肯定还是一个个线程的跑，只不过多个协程在同一个线程中被调度，一会跑跑协程a，一会跑跑协程b，一会跑跑协程c，这三个轮着来的，这样的好处就是一个协程阻塞了其他线程也可以跑？

所以线程的调度是操作系统在做，但是一个线程中协程的调度是go的源码在做，所以我们也可以一定程度上认为协程是用户线程。

线程中协程的调度是go源码在做，线程的调度是操作系统在做。

定义chan的时候 一定是chan int关键字加上类型.

Make(chan int)



上面可以看到,chan是一个阻塞队列,那么就可以当作锁来用了,read和write都是阻塞的.

注意channle 无论write和read都是 <-这个符号.

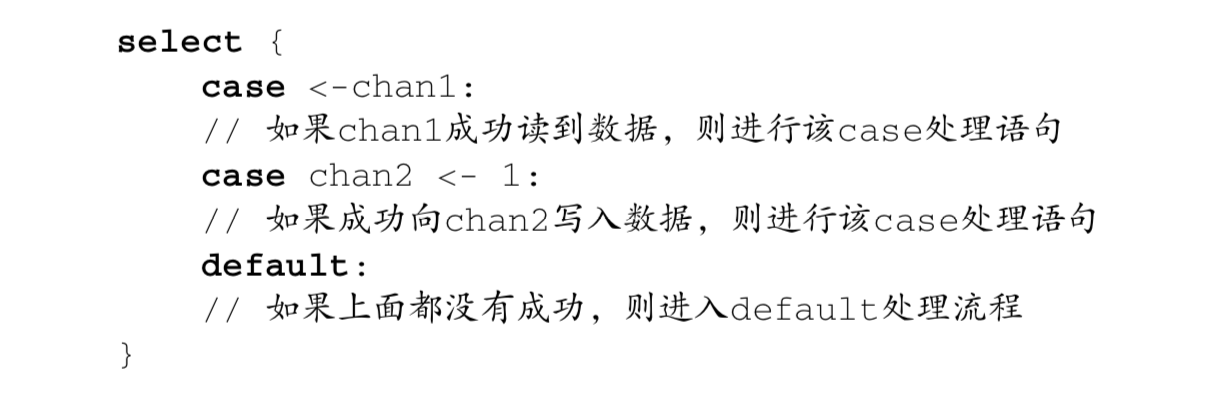
早在unix时代,select机制就已经被引入,通过调用select()函数来监控一系列的文件句柄,一旦其中一个文件句柄发送来IO动作,该select()调用就会被返回.后来该机制也被用于实现高并发的socket服务器程序.

go语言直接在语言级别支持select关键字,用于处理异步IO问题.

select的用户与switch语句非常类似,由select开始一个新的选择块,每个选择条件由case语句来描述.

select语句有比较多的限制,最大的一条限制就是每个case语句里必须是一个IO操作

也就是select语句里必须有一个IO操作.



这个select的意思就是case里面的哪个io操作可以执行就执行哪个,

比如对于上图来说,如果可以从chan1里读到数据,就进行该case,如果能往chan2里写数据,就执行第二个,如果这两个都没有成功就走default.如果都可以执行就会,随机的执行一个.

但是如果没有default的话,就select不到的话就会阻塞在这里

C := make(chan int, 1024) 就是一个make带缓冲的channel

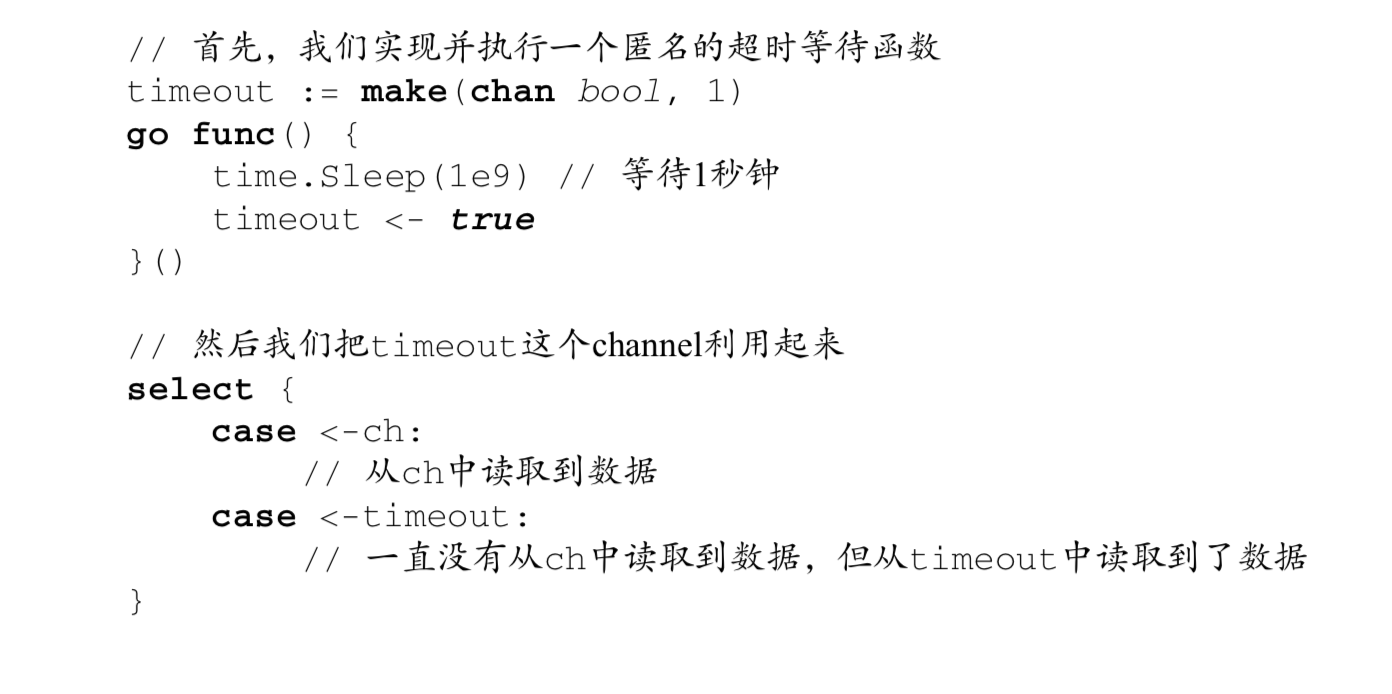
channel的超时问题,上面已经说过了,channel就是一个阻塞队列,对于一个阻塞队列来说,那么肯定就会有写满了或者读空了导致goroutinue一直阻塞的问题.你可以用超时还解决这个问题但是超时的话其实也是弄了线程这样压力还是会很大.

select可以解决这个问题,注意select的作用是监控,也就是说你普通的一个语句,

i := <- ch 这样如果读不到是会阻塞在这里的.

但是呢 select {  
 case <- chan1 这样的话是不会阻塞的,只是不会走case里面的内容了,这就是监控的意思啊

}



这样就弄了一个防止阻塞的channel的东西.

但是注意的是 case <-ch:

这样写的话,只能看到有没有消息来,消息的内容是拿不到的,这样就把这个消息给消耗掉了.

For data := range chs是可以这样玩的.

func main() {  
 ch := make(chan int, 10)  
  
 go func() {  
 time.Sleep(10 \* time.Second)  
 for i:=0; i<10; i++ {  
 ch <- i  
 }  
 close(ch) 这里要close,不然下面会死锁  
 }()  
  
 for v := range ch { 注意这里只有一个v  
 fmt.Println(v)  
 }  
}

结果会打印0123456789

func main() {  
 ch := make(chan int, 10)  
  
 go func() {  
 time.Sleep(10 \* time.Second)  
 for i:=0; i<10; i++ {  
 ch <- i  
 }  
 }()  
  
 for {  
 select {  
 case c := <- ch:  
 fmt.Println(c)  
 }  
 }  
}

这样的话会报死锁

ch := make(chan int, 10)  
  
go func() {  
 time.Sleep(10 \* time.Second)  
 for i:=0; i<10; i++ {  
 ch <- i  
 }  
 close(ch)  
}()  
  
for {  
 select {  
 case c := <- ch:  
 fmt.Println(c)  
 }  
}

这样的话则会疯狂打印0

func main() {  
 ch := make(chan int, 10)  
  
 go func() {  
 time.Sleep(10 \* time.Second)  
 for i:=0; i<10; i++ {  
 ch <- i  
 }  
 close(ch)  
 }()  
  
 for {  
 select {  
 case c, ok := <- ch:  
 if ok {  
 fmt.Println(c)  
  
 }  
 }  
 fmt.Println(11111)  
 }  
}

这样的话则会疯狂打印111,这个循环还是会有的

for {  
 select {  
 case c, ok := <- ch:  
 if ok {  
 fmt.Println(c)  
  
 }else {  
 return  
 }  
 }  
 fmt.Println(11111)  
}

你先结束循环的话可以直接return

for {  
 select {  
 case c, ok := <- ch:  
 if !ok {  
 ch = nil  
 }  
 fmt.Println(c)  
  
 case \_, ok := <- ch2:  
 if !ok {  
 return  
 }  
 fmt.Println(2222)  
  
 }  
 fmt.Println(11111)  
}

如果你这个select监听了多个chan,你不想return,可以当这个发现不是ok的时候把ch赋值成nil,这样的话就select就不会扫描他了.

而且for这里会阻塞,直到channel被关闭,在关闭之前,只要有一个东西被塞进来就会被打印出来的...还是用selectfor吧

单向channel就是只让写或者只让读的channel.

关闭不再需要使用的 channel 并不是必须的。跟其他资源比如打开的文件、socket 连接不一样，这类资源使用完后不关闭后会造成句柄泄露，channel 使用完后不关闭也没有关系，channel 没有被任何协程用到后最终会被 GC 回收。

也可以close(ch)关闭

X, ok := <-ch 判断一个携程是否已经被关闭,这个用法和map中按键获取value的过程比较类似,只需要看第二个bool返回值即可,如果false就表示ch已经被关闭.

Runtime包, go语言的调度器

runtime 调度器是个非常有用的东西，关于 runtime 包几个方法:

Gosched：让当前线程让出 cpu 以让其它线程运行,它不会挂起当前线程，因此当前线程未来会继续执行

NumCPU：返回当前系统的 CPU 核数量

GOMAXPROCS：设置最大的可同时使用的 CPU 核数

***GO默认是使用一个CPU核的，除非设置runtime.GOMAXPROCS***

        那么在多核环境下，什么情况下设置runtime.GOMAXPROCS会比较好的提高速度呢？

          适合于CPU密集型、并行度比较高的情景。如果是IO密集型，CPU之间的切换也会带来性能的损失

go的锁 sync.Mutex 和sync.RWMutex 返回值就是阻塞锁和读写锁.

返回值有Lock unlock Rlock Runlock Wlock WUnlock等

## 网络编程

以前我们使用Socket编程时， 会按照如下步骤展开。

(1) 建立Socket:使用socket()函数。

(2) 绑定Socket:使用bind()函数。

(3) 监听:使用listen()函数。或者连接:使用connect()函数。

(4) 接受连接:使用accept()函数。

(5) 接收:使用receive()函数。或者发送:使用send()函数

go的网络编程在net包里面

go的Dial()函数对上面的socket步骤进行了多次的封装.

func Dial(net, add string)(Conn, error) 直接这样弄一下就可以了.

TCP链接:

conn, err := net.Dial("tcp", "192.168.0.10:2100")

UDP链接:

conn, err := net.Dial("udp", "192.168.0.12:975")

ICMP链接: 相当于ping了百度一下

conn, err := net.Dial("ip4:icmp", "www.baidu.com")

返回的conn有Write()和Read()函数.

实际上，Dial()函数是对DialTCP()、DialUDP()、DialIP()和DialUnix()的封装。我

们也可以直接调用这些函数，它们的功能是一致的。这些函数的原型如下:

func DialTCP(net string, laddr, raddr \*TCPAddr) (c \*TCPConn, err error) func DialUDP(net string, laddr, raddr \*UDPAddr) (c \*UDPConn, err error) func DialIP(netProto string, laddr, raddr \*IPAddr) (\*IPConn, error)

func DialUnix(net string, laddr, raddr \*UnixAddr) (c \*UnixConn, err error)

net.ResolveTCPAddr()，用于解析地址和端口号;  net.DialTCP()，用于建立链接。

这两个函数在Dial()中都得到了封装。

此外，net包中还包含了一系列的工具函数，合理地使用这些函数可以更好地保障程序的 质量。

验证IP地址有效性的代码如下: func net.ParseIP()

创建子网掩码的代码如下:

func IPv4Mask(a, b, c, d byte) IPMask 获取默认子网掩码的代码如下:

func (ip IP) DefaultMask() IPMask 根据域名查找IP的代码如下:

func ResolveIPAddr(net, addr string) (\*IPAddr, error)

func LookupHost(name string) (cname string, addrs []string, err error);

也就是还有很多这方面的函数.

上面那些都是传输层级别的函数,谁跟你玩tcp对tcp啊,也就是mme总跟你这么搞.

那么http的网络编程,

Net/http包里面

resp, err := http.Get("http://example.com/") if err != nil {

// 处理错误 ...

return

}

defer resp.Body.close() io.Copy(os.Stdout, resp.Body)

也就是http.Get(“URL/xx/xxxxxxx”)

要以POST的方式发送数据，也很简单，只需调用http.Post()方法并依次传递下面的3个

参数即可:

 请求的目标 URL **将要 POST 数据的资源类型(MIMEType)**

 数据的比特流([]byte形式)

下面的示例代码演示了如何上传一张图片:

***resp, err := http.Post("http://example.com/upload", "image/jpeg", &imageDataBuf) if err != nil {***

// 处理错误

return

}

if resp.StatusCode != http.StatusOK { // 处理错误

return

}

 http.PostForm() http.PostForm()方法实现了标准编码格式为application/x-www-form-urlencoded

的表单提交。下面的示例代码模拟HTML表单提交一篇新文章:

resp, err := http.PostForm("http://example.com/posts",

{"article title"}, "content": {"article body"}})

if err != nil { // 处理错误

return

}// ...

req, err := http.NewRequest("GET", "http://example.com", nil) // ...

req.Header.Add("User-Agent", "Gobook Custom User-Agent")

// ...

client := &http.Client{ //... }

resp, err := client.Do(req)

除了上面的哪些比较方便的,也可以像上面那样,

先new一个req,然后req可以加一些操作,set些header啊,自定义的也可以,也可以设置cookie,然后在新建一个client,然后再调用client的Do()方法

所以说直接http.Get() http.Post()这种方法,这些都是直接用的默认的httpclinet,我们也可以自己定义client咯.

type Client struct {

// Transport用于确定HTTP请求的创建机制。

// 如果为空，将会使用DefaultTransport

Transport RoundTripper

// CheckRedirect定义重定向策略。

// 如果CheckRedirect不为空，客户端将在跟踪HTTP重定向前调用该函数。

// 两个参数req和via分别为即将发起的请求和已经发起的所有请求，最早的

// 已发起请求在最前面。

// 如果CheckRedirect返回错误，客户端将直接返回错误，不会再发起该请求。 // 如 果CheckRedirect为空，Client将采用一种确认策略，将在10个连续

// 请求后终止

CheckRedirect func(req \*Request, via []\*Request) error

// 如果Jar为空，Cookie将不会在请求中发送，并会

// 在响应中被忽略

Jar CookieJar

}

实际上，我们一般都用 http.SetCookie() 方法来设定 Cookie。

使用自定义的http.Client及其Do()方法，我们可以非常灵活地控制 HTTP 请求，比如发 送自定义 HTTP Header 或是改写重定向策略等。创建自定义的 HTTP Client 非常简单，具体代码 如下:

client := &http.Client {

CheckRedirect: redirectPolicyFunc,

}

resp, err := client.Get("http://example.com")

// ...

req, err := http.NewRequest("GET", "http://example.com", nil) // ...

req.Header.Add("User-Agent", "Our Custom User-Agent") req.Header.Add("If-None-Match", `W/"TheFileEtag"`)

resp, err := client.Do(req)

就是client对象去Do一下咯.

这是transpor的结构

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)

// TLSClientConfig指定用于tls.Client的TLS配置。

// 如果为空则使用默认配置

TLSClientConfig \*tls.Config

DisableKeepAlives bool

DisableCompression bool

// 如果MaxIdleConnsPerHost为非零值，它用于控制每个host所需要

// 保持的最大空闲连接数。如果该值为空，则使用DefaultMaxIdleConnsPerHost MaxIdleConnsPerHost int

// ... 8

}

瞅瞅,上面标黄的都是函数类型的数据,看到没有

除了上面哪些还有这几个

func(t \*Transport) CloseIdleConnections()。该方法用于关闭所有非活跃的

连接。

 func(t \*Transport) RegisterProtocol(scheme string, rt RoundTripper)。

该方法可用于注册并启用一个新的传输协议，比如 WebSocket 的传输协议标准(ws)，或

者 FTP、File 协议等。

 func(t \*Transport) RoundTrip(req \*Request) (resp \*Response, err error)。

用于实现 http.RoundTripper 接口。

综上示例讲解可以看到，Go语言标准库提供的 HTTP Client 是相当优雅的。一方面提供了极

其简单的使用方式，另一方面又具备极大的灵活性。

Go语言标准库提供的HTTP Client 被设计成上下两层结构。一层是上述提到的 http.Client

类及其封装的基础方法，我们不妨将其称为“业务层”。之所以称为业务层，是因为调用方通常 只需要关心请求的业务逻辑本身，而无需关心非业务相关的技术细节，这些细节包括:

 HTTP 底层传输细节  HTTP 代理

 gzip 压缩

 连接池及其管理

 认证(SSL或其他认证方式)

之所以HTTP Client可以做到这么好的封装性，是因为HTTP Client 在底层抽象了

http.RoundTripper 接口，而http.Transport 实现了该接口，从而能够处理更多的细节，我 们不妨将其称为“传输层”。HTTP Client 在业务层初始化 HTTP Method、目标URL、请求参数、 请求内容等重要信息后，经过“传输层”，“传输层”在业务层处理的基础上补充其他细节，然后 再发起 HTTP 请求，接收服务端返回的 HTTP 响应。

反正就是clinet这个结构体函数比较少,它饮用了Transport这个结构体,这个Transport里面封装了很多传输层的函数,就是你的do函数内部调用的.

使用 net/http 包提供的http.ListenAndServe()方法，可以在指定的地址进行监听， 开启一个HTTP.

net/http 包还提供 http.ListenAndServeTLS() 方法，用于处理 HTTPS 连接请求:

func ListenAndServeTLS(addr string, certFile string, keyFile string, handler Handler) error

Rpc remote procedure call 远程过程调用

rpc是一种通过网络从远程计算程序上请求服务,而不需要了解底层网络细节的应用程序通信协议. Rpc协议可以构建于TCP或者UDP之上,也可以构建.