go+区块链培训 讲师:张长志

管道channel

gorountine 运行在相同地址，之间的**通信**通过channel完成

channel类型

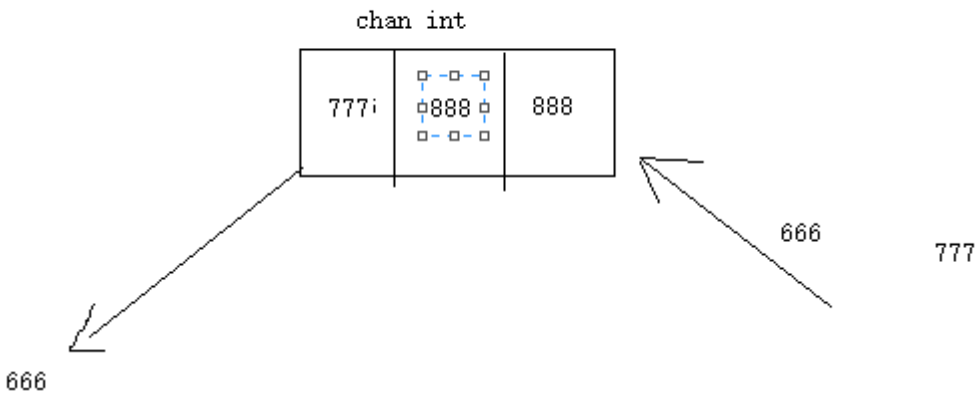
与map类似，make创建的底层数据结构的引用

当我们复制一个channel或用于函数参数传递时，我们只是拷贝了一个channel**引用**，和其它的引用类型一样，channel的零值也是nil

定义一个channel时，也需要定义发送到channel的值的类型。channel可以使用内置的make()函数来创建：

```
make(chan Type) //Type int string
make(chan Type,capacity) //容量
```

当 capacity= 0 时，channel 是**无缓冲阻塞读写**的，当capacity> 0 时，channel 有缓冲、是非阻塞的，直到写满capacity个元素才阻塞写入。



channel 定义

channel会通过 <- 来接收和发送数据

channel通过操作符<-来接收和发送数据，发送和接收数据语法：

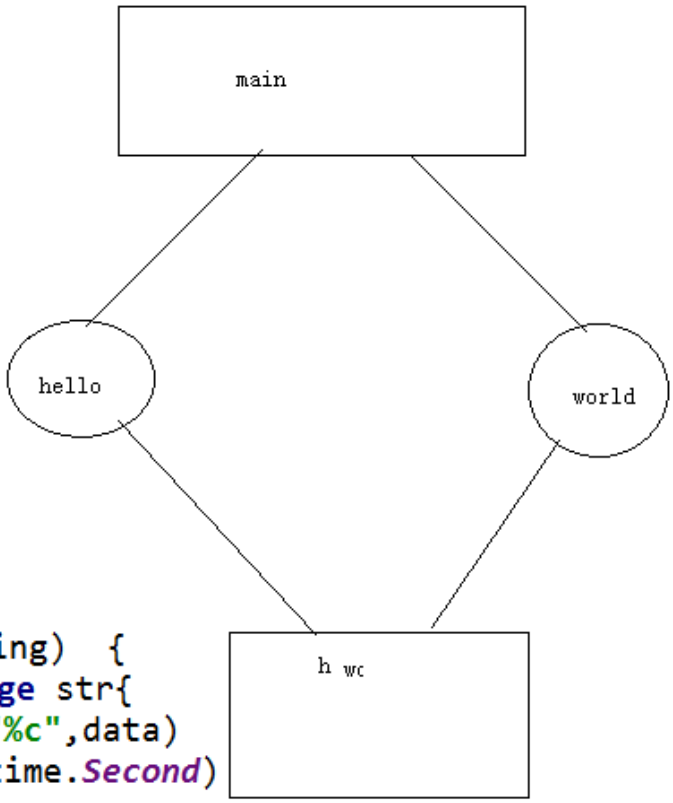
```
    channel <- value      //发送value到channel
    <-channel             //接收并将其丢弃
    x := <-channel        //从channel中接收数据，并赋值给x
    x, ok := <-channel    //功能同上，同时检查通道是否已关闭或者是否为空
```

默认情况下，channel接收和发送数据都是阻塞的

channel阻塞代码实例

```
package main

import (
   "fmt"
   "time"
)

func main(){
   //创建channel
   ch := make(chan string)
   defer fmt.Println("主协程也结束")

   go func() {
      defer fmt.Println("子协程结束")
      for i:=0;i<2;i++{
         fmt.Println("子协程i=",i)
         time.Sleep(time.Second)
      }
          ch <- "我是子协程，子协程工作完毕"
   }()

   str := <-ch //没有数据，阻塞
   fmt.Println("str=",str)

}
```

解决打印资源抢占问题




```
func Printer(str string)  {
    for _,data :=range str{
        fmt.Printf("%c",data)
        time.Sleep(time.Second)
    }
```

```go
package main

import (
   "fmt"
   "time"
)

//定义全局的变量，创建一个channel
var ch = make(chan int)
//定义一个打印机，参数字符串，按照每个字符打印
func Printer(str string)  {
   for _,data :=range str{
      fmt.Printf("%c",data)
      time.Sleep(time.Second)
   }
   fmt.Printf("\n")

}

func Person1()  {
  Printer("hello")
  ch <- 66666

}

func Person2()  {
```

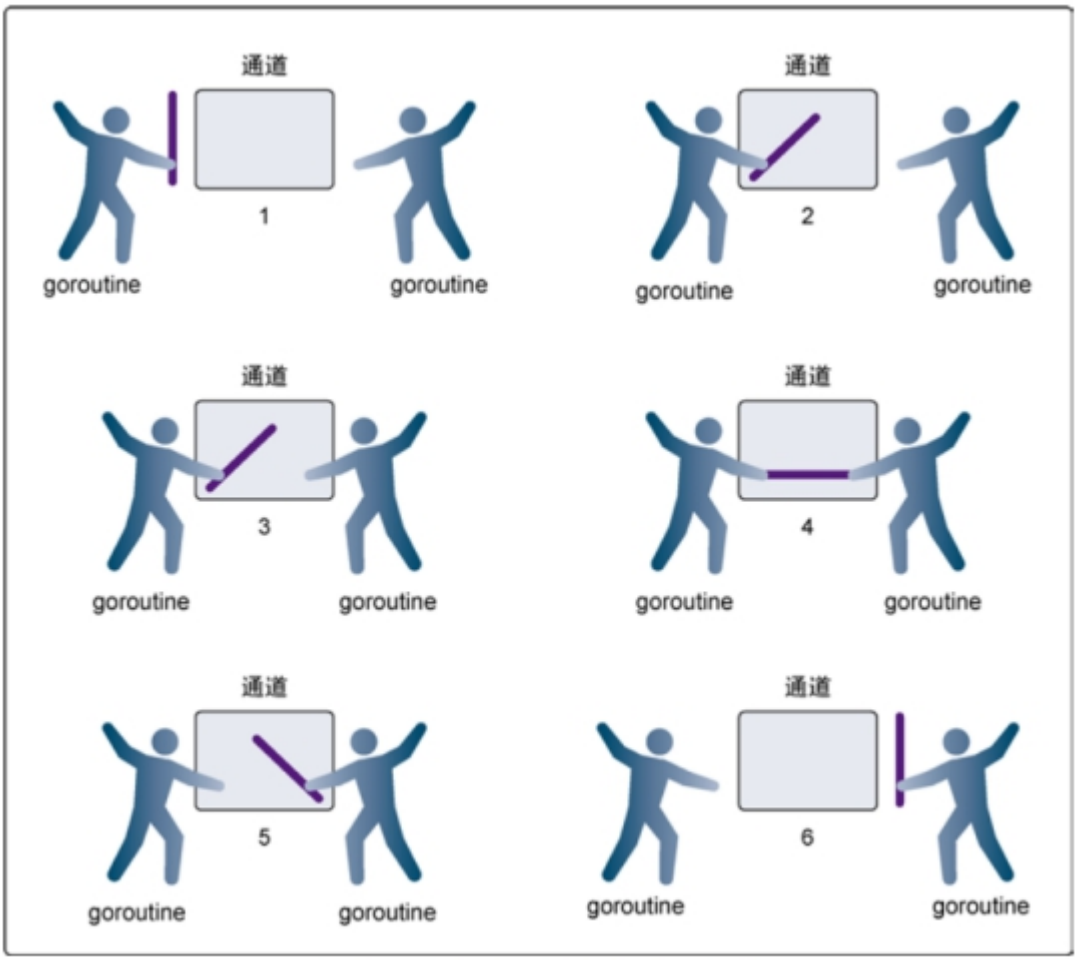
```
     <- ch //从管道读取数据，接收，如果没有数据它会阻塞
    Printer("world")
}
func main(){
    //2个协程共有一个资源
    go Person1()
    go Person2()

    for{

    }

}
```

无缓冲的channel

无缓冲的通道（unbuffered channel）是指在接收前没有能力保存任何值的通道。

这种类型的通道要求发送 goroutine 和接收 goroutine 同时准备好，才能完成发送和接收操作。如果两个goroutine没有同时准备好，通道会导致先执行发送或接收操作的 goroutine **阻塞等待**。

这种对通道进行发送和接收的交互行为本身就是同步的。其中任意一个操作都无法离开另一个操作单独存在。



使用无缓冲的通道在 goroutine 之间同步

l 在第 1 步，两个 goroutine 都到达通道，但哪个都没有开始执行发送或者接收。

l 在第 2 步，左侧的 goroutine 将它的手伸进了通道，这模拟了向通道发送数据的行为。这时，这个 goroutine 会在通道中被锁住，直到交换完成。

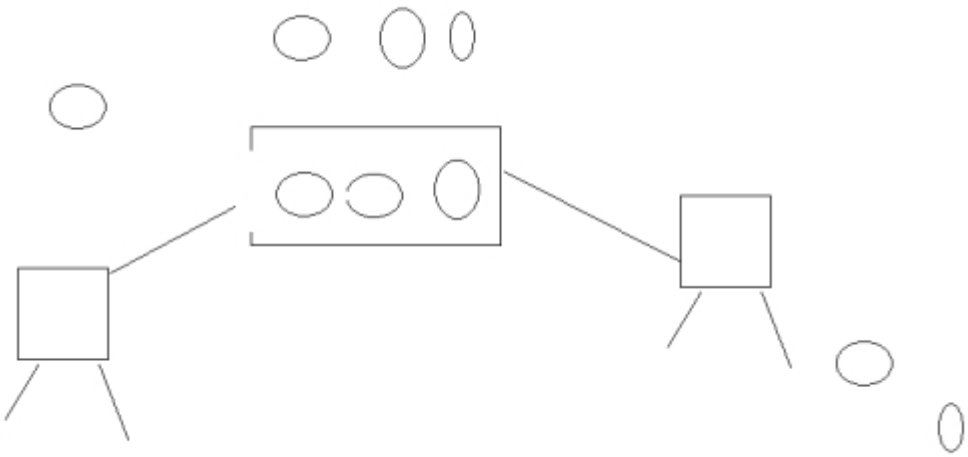
l 在第 3 步，右侧的 goroutine 将它的手放入通道，这模拟了从通道里接收数据。这个 goroutine 一样也会在通道中被锁住，直到交换完成。

l 在第 4 步和第 5 步，进行交换，并最终，在第 6 步，两个 goroutine 都将它们的手从通道里拿出来，这模拟了被锁住的 goroutine 得到释放。两个 goroutine 现在都可以去做别的事情了。

```
package main

import (
   "fmt"
   "time"
)

func main()  {
   //创建一个无缓存区的channel
   ch := make(chan int,0)
```

```
    //len(ch)缓冲区使用数据个数，cap(ch)缓冲区大小
    fmt.Printf("len(ch)=%d,cap(ch)=%d\n",len(ch),cap(ch))


    go func() {
       for i:=0;i<3;i++{
          fmt.Printf("子协程:i=%d\n",i)
          ch <- i//往chan写内容
          fmt.Printf("len(ch)=%d,cap(ch)=%d\n",len(ch),cap(ch))
       }
    }()
     //延时2秒
    time.Sleep(2*time.Second)

    for i:=0;i<3;i++{
       num := <-ch //读管道内容，没有内容前，阻塞
       fmt.Println("num=",num)
    }
}
```

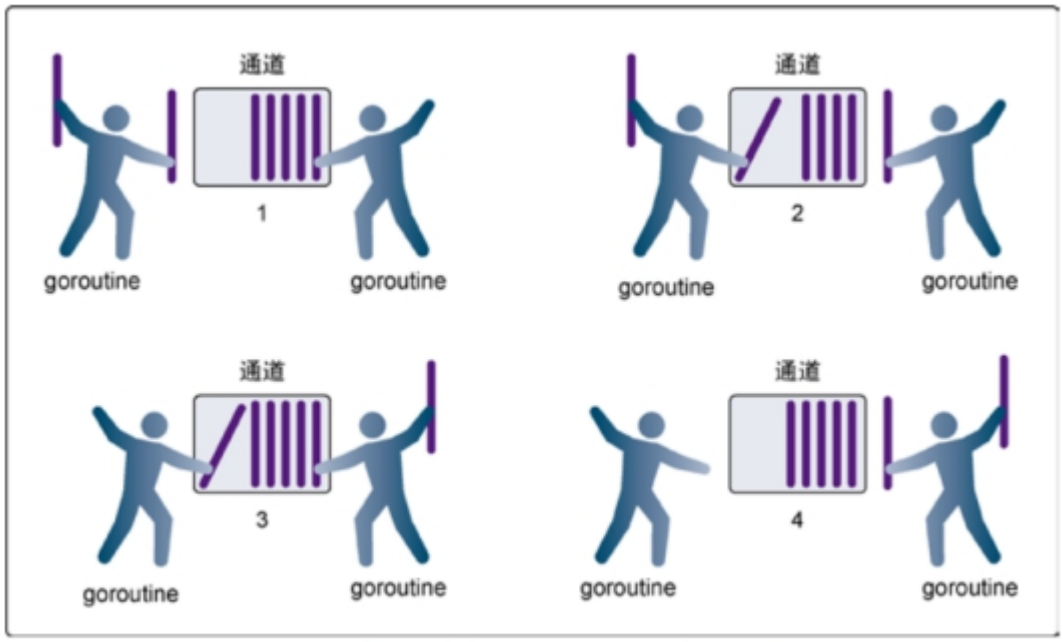
有缓冲的channel



有缓冲的通道（buffered channel）是一种在被接收前能存储一个或者多个值的通道。

这种类型的通道并不强制要求 goroutine 之间必须同时完成发送和接收。通道会阻塞发送和接收动作的条件也会不同。只有在通道中没有要接收的值时，接收动作才会阻塞。只有在通道没有可用缓冲区容纳被发送的值时，发送动作才会阻塞。

这导致有缓冲的通道和无缓冲的通道之间的一个很大的不同：**无缓冲的通道保证进行发送和接收的 goroutine 会在同一时间进行数据交换；有缓冲的通道没有这种保证。**



使用有缓冲的通道在 goroutine 之间同步数据

l 在第 1 步，右侧的 goroutine 正在从通道接收一个值。

l 在第 2 步，右侧的这个 goroutine独立完成了接收值的动作，而左侧的 goroutine 正在发送一个新值到通道里。

l 在第 3 步，左侧的goroutine 还在向通道发送新值，而右侧的 goroutine 正在从通道接收另外一个值。这个步骤里的两个操作既不是同步的，也不会互相阻塞。

l 最后，在第 4 步，所有的发送和接收都完成，而通道里还有几个值，也有一些空间可以存更多的值。

有缓存区的channel

```
package main

import (
   "fmt"

   "time"
)

func main()  {
   //创建一个有缓存区的管道
   ch := make(chan int,3)
    //len(ch) 缓存区里面有多个数据，cap(ch )缓存区大小
   fmt.Printf("len(ch)=%d,cap(ch)=%d\n",len(ch),cap(ch))

   go func() {

      for i:=0;i< 10;i++{
         ch <- i //往chan写内容
         fmt.Printf("子协程[%d]:len(ch)=%d,cap(ch)=%d\n",i,len(ch),cap(ch))
      }
```

```
    }()

    time.Sleep(2*time.Second)

    for i:=0;i<10;i++{
        num := <-ch
        fmt.Println("num=",num)
    }
}
```

关闭channel方法close

如果发送者知道，没有更多的值需要发送到channel的话，那么让接收者也能及时知道没有多余的值可接收将是有用的，因为接收者可以停止不必要的接收等待。这可以通过内置的close函数来关闭channel实现。

```
package main

import (
    "fmt"
)

func main()  {
    ch := make(chan int) //创建一个无缓存的channle

    go func() {
      for i:= 0;i<5;i++{
         ch <- i //往通道写数据
      }
        close(ch) //不需要写数据的时候 我们关闭channel
       // ch <-666  //send on closed channel 关闭channel后无法在发送数据
    }()

    for {
       //如果ok为true,说明管道没有关闭
       if num,ok := <-ch;ok==true{
         fmt.Println("num=",num)
       }else {//管道关闭
         break
       }
    }
}
```

注意点：

l channel不像文件一样需要经常去关闭，只有当你确实没有任何发送数据了，或者你想显式的结束range循环之类的，才去关闭channel；

l 关闭channel后，无法向channel 再发送数据(引发 panic 错误后导致接收立即返回零值)；

l 关闭channel后，可以继续向channel接收数据；

l 对于nil channel，无论收发都会被阻塞。

通过range读取对应的值

```
package main

import "fmt"

func main()  {
   ch := make(chan int)

   go func() {
       for i:=0;i<5;i++{
         ch <- i //往通道写数据
       }
       //不需要写数据的时候，关闭channel
       close(ch)
   }()

   for num := range ch{
      fmt.Println("num=",num)
   }
}
```

单方向的channel

读 或者 写

默认情况下，通道是双向的，也就是，既可以往里面**发送数据**也可以同里面**接收数据**。

但是，我们经常见一个通道作为参数进行传递而值希望对方是单向使用的，要么只让它发送数据，要么只让它接收数据，这时候我们可以指定通道的方向。

单向channel变量的声明非常简单，如下：

var ch1 **chan** int // ch1是一个正常的channel，不是单向的

var ch2 **chan**<- float64 // ch2是单向channel，只用于写float64数据

var ch3 <-**chan** int // ch3是单向channel，只用于读取int数据

l chan<- 表示数据进入管道，要把数据写进管道，对于调用者就是输出。

l <-chan 表示数据从管道出来，对于调用者就是得到管道的数据，当然就是输入。

```
package main

func main()  {
   //创建一个管道
   ch := make(chan int)
    //双向channel能隐式的转换为单向的channel
   var writeCh chan<- int = ch //只能写 不能读
   var readch <-chan int = ch //只能读 不能写

   writeCh <- 666
   //<- writeCh  //invalid operation: <-writeCh (receive from send-only type chan<-
int)

   <-readch //读
   //readch <- 666 //invalid operation: readch <- 666 (send to receive-only type <-chan
int)

    //单向的无法转换成双向的
   //var ch2 chan int =writeCh //cannot use writeCh (type chan<- int) as type chan int
in assignment
   }
```

单向管道的应用

```
package main

import (
   "fmt"
   //"time"
)

//创建通道只能写，不能读\
//out chan<- int
func producer(out chan<- int){ //out chan<- int = ch
    for i :=0;i<10;i++{
       out <- i*i
   }
   close(out)
}

func consumer(in <-chan  int)  {
   for num:=range in{
          fmt.Println("num=",num)
   }
}

func main()  {
   //创建一个双向管道
   ch := make(chan int)
   //生成者，生成数据,开启一个协程
```

```
    go producer(ch)
     //消费者，从channel管道读取内容，打印
     consumer(ch)

    // time.Sleep(2*time.Second)

}
```

定时器

Timer 是一个定时器，代表未来的一个单一事件，你可以告诉timer你要等待多长时间，它提供一个**channel**，在将来的那个时间那个channel提供了一个时间值。

```
package main

import (
    "time"
   "fmt"
)

func main(){
   //创建一个定时器，设置时间为2s，2s后，往time通道写内容
   timer := time.NewTimer(3*time.Second)
   fmt.Println("当前时间：",time.Now())

    t :=<- timer.C //channel 没有数据前后阻塞
    fmt.Println("t=",t)

}
```

time.NewTImer 时间到了只会响应一次

```
package main

import (
   "time"
   "fmt"
)

func main(){
   //验证time.newTimer（）时间到了 只会响应一次
   timer := time.NewTimer(1*time.Second)

   for{
      <-timer.C
      fmt.Println("时间到")
```

```
    }
}
```

延时的3种方式

```
package main

import (
    "time"
    "fmt"
)
//time.NewTimer(2*time.Second).C
func main()  {
    <- time.After(2*time.Second) //定时2s 阻塞2s 2s后产生一个事假往channel写内容，和第一种一样
    fmt.Println("时间到")
    }

/*
func main()  {
    time.Sleep(2*time.Second)
    fmt.Println("时间到")
}
*/

/*
func main(){
    //延时2秒方法
    timer := time.NewTimer(2*time.Second).c

    <-timer.C
    fmt.Println("时间到")
}
*/
```

定时器的停止操作

```
package main

import (
    "time"
    "fmt"
)

func main(){
    timer := time.NewTimer(3*time.Second)

    go func() {
        <- timer.C
        fmt.Println("子协程可以打印了，因为定时器的时间到了")
```

```
    }()

    timer.Stop()

    for{

    }
}
```

定时器的时间重置

```
package main

import (
    "time"
    "fmt"
)

func main()  {
    timer := time.NewTimer(3*time.Second)
    ok := timer.Reset(1*time.Second) //把以前3秒重置1s
    fmt.Println("ok=",ok)

    <-timer.C
    fmt.Println("时间到")
}


/*func main(){
    timer := time.NewTimer(3*time.Second)

    go func() {
        <- timer.C
        fmt.Println("子协程可以打印了，因为定时器的时间到了")
    }()

    timer.Stop()

    for{

    }
}*/
```

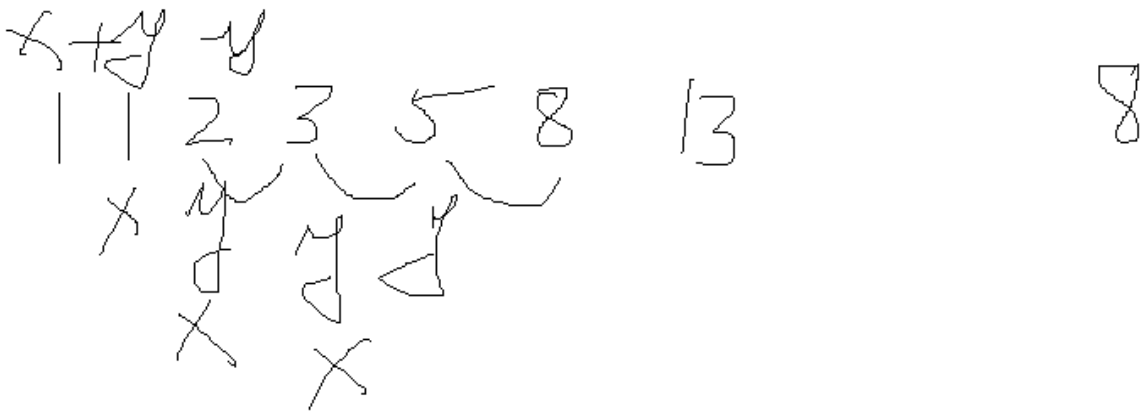
Ticker

Ticker是一个定时触发的计时器，它会以一个**间隔**(interval)往channel发送一个事件(当前时间)，而channel的接收者可以以固定的时间间隔从channel中读取事件。（**周期性的**）

```
```

```
package main

import (
   "time"
   "fmt"
)

func main(){
   ticker := time.NewTicker(1*time.Second)

   i := 0

   for{
       <- ticker.C  //1s钟的间隔
       i++
       fmt.Println("i=",i)

       if i==5{
         ticker.Stop()
         break
       }

   }

}
```

案例



```
package main

import "fmt"

func fibonacci(ch chan<- int,quit <-chan  bool) {
    x,y := 1,1
    for{
```

```
        //监听channel数据流动
        select {
        case ch <-x:
            x,y = y,x+y
        case  flag:= <-quit:
            fmt.Println("flag=",flag)
            return

        }
    }
}

func main(){
    ch := make(chan int) //数字通信

    quit := make(chan bool) //程序是否退出 true

    go func() {
        for i :=0;i<8;i++{
            num := <-ch
            fmt.Println(num)
        }
        quit <- true

    }()

    fibonacci(ch,quit)


}
```

select作用

Go里面提供了一个关键字select，通过select可以监听channel上的数据流动。

select的用法与switch语言非常类似，由select开始一个新的选择块，每个选择条件由case语句来描述。

与switch语句可以选择任何可使用相等比较的条件相比， select有比较多的限制，其中最大的一条限制就是每个case语句里**必须是一个IO操作**，大致的结构如下：

```
    select {
     case <-chan1:
         // 如果chan1成功读到数据，则进行该case处理语句
     case chan2 <- 1:
         // 如果成功向chan2写入数据，则进行该case处理语句

     }
```

在一个select语句中，Go语言会按顺序从头至尾评估每一个**发送和接收**的语句。

如果其中的**任意一语句可以继续执行**(即没有被阻塞)，那么就从那些可以执行的语句中**任意选择一条**来使用。

如果没有任意一条语句可以执行(即所有的通道都被阻塞)，那么有两种可能的情况：

- 如果给出了default语句，那么就会执行default语句，执行完default会从select语句继续恢复执行
- 如果没有default语句，select阻塞，直到至少有一个语句可以执行下去

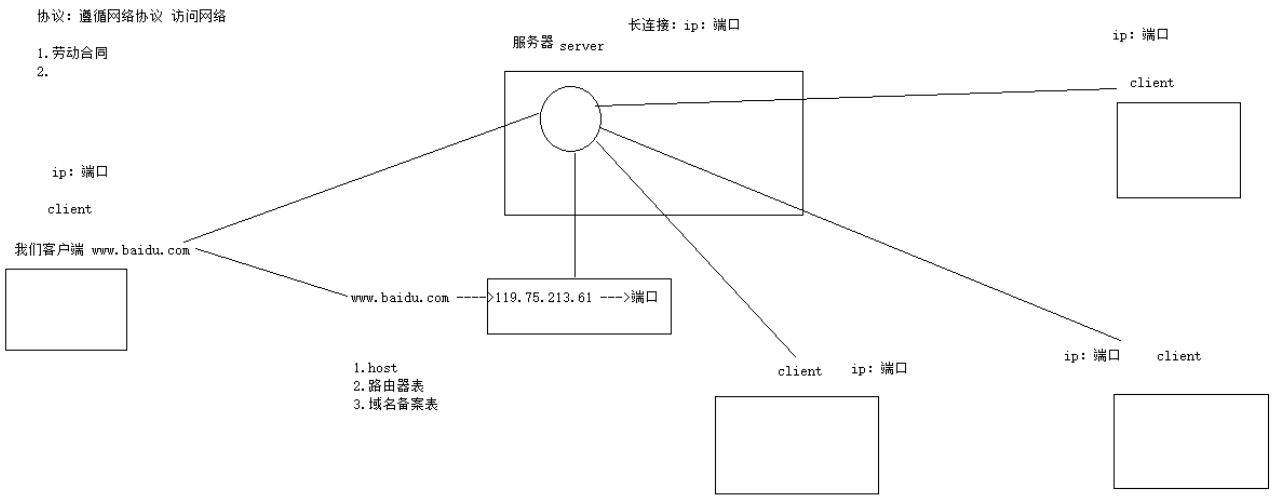
开发中 select语句中一般没有default，这样可以减少开销

超时问题

有时候会出现goroutine阻塞的情况，那么我们如何避免整个程序进入阻塞的情况呢？我们可以利用select来设置超时，通过如下的方式实现：

```
package main

import (
   "fmt"
   "time"
)

func main()  {
   ch := make(chan int)
   quit := make(chan bool)

   go func() {
      for{
         select {
               case num := <-ch:
                fmt.Println("num=",num)
                  case <-time.After(3*time.Second):
                   fmt.Println("超时")
                    quit <- true
                    //break
         }
      }
   }()

   for i:= 0;i<5;i++{
      ch <- i
      time.Sleep(time.Second)
   }
   qt:= <-quit
   fmt.Println("程序结束:qt=",qt)

}
```

网络协议



网络协议

从应用的角度出发，协议可理解为“规则”，是数据传输和数据的解释的规则。

假设，A、B双方欲传输文件。规定：

l 第一次，传输文件名，接收方接收到文件名，应答OK给传输方；

l 第二次，发送文件的尺寸，接收方接收到该数据再次应答一个OK；

l 第三次，传输文件内容。同样，接收方接收数据完成后应答OK表示文件内容接收成功。

由此，无论A、B之间传递何种文件，都是通过三次数据传输来完成。A、B之间形成了一个最简单的数据传输规则。双方都按此规则发送、接收数据。A、B之间达成的这个相互遵守的规则即为协议。

这种仅在A、B之间被遵守的协议称之为**原始协议**。

当此协议被更多人使用，不断增加 改进 维护 完善 最终形成一个稳定的 完整的文件传输协议，被广泛的应用各种文件传输过程中，这种原始协议就形成一个**标准协议**。最早ftp就是这样形成的。

网络分层架构

为了减少协议的复杂性，大多数网络模型均采用分层模式来组织。每一层都有自己的功能，就像建筑物，每一层都靠下一层支持。每一层都利用下一层提供的服务来为上一层提供服务，本层服务的实现细节对上层屏蔽。每一层只要关心本层的协议和功能既可以。

OSI/RM(理论上的标准)	TCP/IP(事实上的标准)
应用层	应用层
表示层	
会话层	
传输层	传输层
网络层	网络层
数据链路层	链路层
物理层	

越下面的越接近硬件，越上面的越接近用户，至于每一层叫什么名字，其实并不重要，只要知道，互联网分层多个即可。

1)物理层：主要定义物理设备标准，如网线接口类型，光纤的接口类型，各种传输介质的传输速率。主要作用就是传输比特流（1和0转换成电流强弱来进行传输，达到目的地把1.0转换成用户想要的信息） 比特层

2）数据链路层：定义了如果让格式化的数据以祯为单位进行传输，以及如何让控制对物理介质的访问，这一层通常还提供错误的检测和纠正，以确保数据可靠传输。串口通信：8 N

3）网络层：在位于不同地理位置的网络中的俩个主机之间提供链接和路径选择。Internet发展使得从世界各站访问信息的用户数大大增加，而网络层这是管理这种链接层的

4）传输层：定义了一些传输的协议和端口号（www 80） tcp（传输控制协议，传输效率低 可靠性强，用于传输可靠性要求高 数据量大的数据） udp（通过这种） 只要将下层接收的数据进行分段和传输，达到目的地进行重组。有人叫段

5）会话层：通过传输层（端口号：传输端口和接收端口）建立数据传输通路 主要在系统之间发起会话或者接收会话

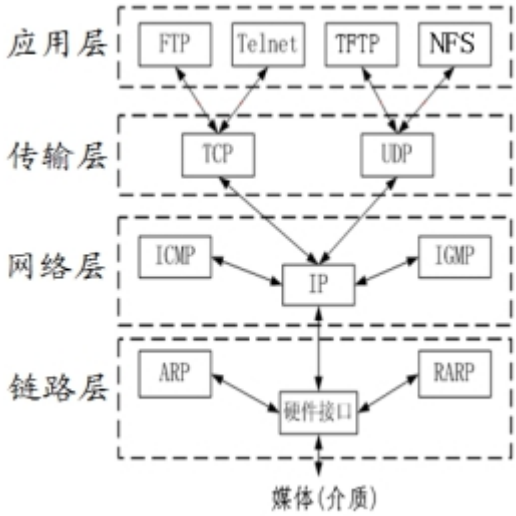
6）表示层：可以确保一个系统的应用层所有发送的信息可以被另一个系统的应用层读取，比如：pc程序与另一个计算机通信，其中一台计算机使用扩展A编码 ASCII编码，进行转换

7）应用层:这一层是应用程序提供网络服务（电子邮箱和终端仿真）

层与协议

每一层都是为了完成一种功能，为了实现这些功能，就需要大家都遵守共同的规则。大家都遵守这规则，就叫做“协议”（protocol）。

网络的每一层，都定义了很多协议。这些协议的总称，叫“TCP/IP协议”。TCP/IP协议是一个大家族，不仅仅只有TCP和IP协议，它还包括其它的协议，如下图：



每层协议的功能

1) 链路层

以太网规定，连入网络的所有设备，都必须具有“网卡”接口。数据包必须是从一块网卡，传送到另一块网卡。通过网卡能够使不同的计算机之间连接，从而完成数据通信等功能。网卡的地址——MAC 地址，就是数据包的物理发送地址和物理接收地址。

2) 网络层

网络层的作用是引进一套新的地址，使得我们能够区分不同的计算机是否属于同一个子网络。这套地址就叫做“网络地址”，这是我们平时所说的IP地址。这个IP地址好比我们的手机号码，通过手机号码可以得到用户所在的归属地。

192.168.75.201

网络地址帮助我们确定计算机所在的子网络，MAC 地址则将数据包送到该子网络中的目标网卡。网络层协议包含的主要信息是源IP和目的IP。

于是，“网络层”出现以后，每台计算机有了两种地址，一种是 MAC 地址，另一种是网络地址。**两种地址之间没有任何联系**，MAC 地址是绑定在网卡上的，网络地址则是管理员分配的，它们只是随机组合在一起。

网络地址帮助我们确定计算机所在的子网络，MAC 地址则将数据包送到该子网络中的目标网卡。因此，从逻辑上可以推断，必定是先处理网络地址，然后再处理 MAC 地址。

3) 传输层

当我们一边聊QQ，一边聊微信，当一个数据包从互联网上发来的时候，我们怎么知道，它是来自QQ的内容，还是来自微信的内容？

也就是说，我们还需要一个参数，表示这个数据包到底供哪个程序（进程）使用。这个参数就叫做“端口”（port），它其实是每一个使用网卡的程序的编号。每个数据包都发到主机的特定端口，所以不同的程序就能取到自己所需要的数据。

端口特点：

l 对于同一个端口，在不同系统中对应着不同的进程

l 对于同一个系统，一个端口只能被一个进程拥有

4) 应用层

应用程序收到“传输层”的数据，接下来就要进行解读。由于互联网是开放架构，数据来源五花八门，必须事先规定好格式，否则根本无法解读。“应用层”的作用，就是规定应用程序的数据格式。

画图解析

网络通信的条件

1）网卡mac地址(不需要用户处理)

2）逻辑地址 ip地址（需要用户绑定）

3）端口

- 同一个系统，一个程序只能绑定一个端口
- 不同系统同一个端口对应程序可能不一样 80 qq 80 微信