1、基本介绍

net包提供了可移植的网络I/O接口，包括TCP/IP、UDP、域名解析和Unix域socket。

虽然本包提供了对网络原语的访问，大部分使用者只需要Dial、Listen和Accept函数提供的基本接口；以及相关的Conn和Listener接口。crypto/tls包提供了相同的接口和类似的Dial和Listen函数。

Dial函数和服务端建立连接：

```
conn, err := net.Dial("tcp", "google.com:80")
if err != nil {
        // handle error
}
fmt.Fprintf(conn, "GET / HTTP/1.0\r\n\r\n")
status, err := bufio.NewReader(conn).ReadString('\n')
// ...
```

Listen函数创建的服务端：

```
ln, err := net.Listen("tcp", ":8080")
if err != nil {
        // handle error
}
for {
        conn, err := ln.Accept()
        if err != nil {
                // handle error
                continue
        }
        go handleConnection(conn)
}
```


2、Go实现TCP通信

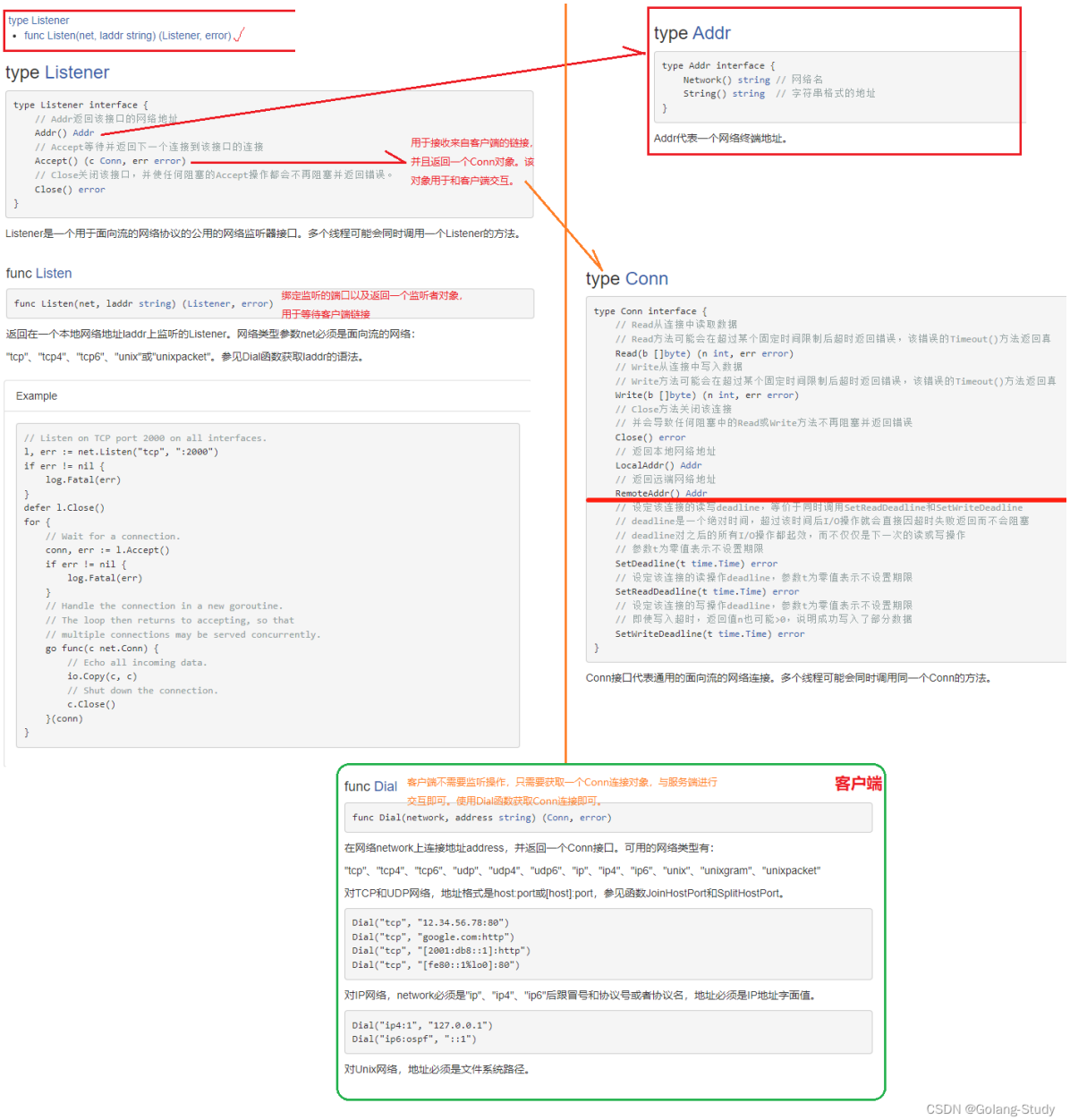
2.1 TCP协议

TCP/IP(Transmission Control Protocol/Internet Protocol) 即传输控制协议/网间协议，是一种面向连接（连接导向）的、可靠的、基于字节流的传输层（Transport layer）通信协议，因为是面向连接的协议，数据像水流一样传输，会存在黏包问题。

2.2 TCP服务端

一个TCP服务端可以同时连接很多个客户端，例如：世界各地的用户使用自己电脑上的浏览器访问淘宝网。因为Go语言中创建多个goroutine实现并发非常方便和高效，所以我们可以每建立一次链接就创建一个goroutine去处理。

TCP服务端程序的处理流程：
首先：监听端口
然后：接收客户端请求建立链接
最后：创建goroutine处理链接



*** 客户端

```
package main

import (
    "bufio"
    "fmt"
    "net"
)

// tcp/server/main.go

// TCP server端

// 处理函数
func process(conn net.Conn) {

    defer conn.Close() // 关闭连接

    for {

        reader := bufio.NewReader(conn)
```

```
        var buf [128]byte

        n, err := reader.Read(buf[:]) // 读取数据
        if err != nil {
            fmt.Println("read from client failed, err:", err)
            break
        }

        recvStr := string(buf[:n])
        fmt.Printf("收到客户端 %s 信息--> %s\n", conn.RemoteAddr().String(),
recvStr)
        conn.Write([]byte("ret--> " + recvStr)) // 发送数据
    }
}

func main() {

    // 监听
    listen, err := net.Listen("tcp", ":8888")

    fmt.Println("服务端已经开启, 正在监听, ip&port --> ", listen.Addr())

    if err != nil {
        fmt.Println("listen failed, err:", err)
        return
    }

    for { // 循环等待

        // 等待建立连接
        conn, err := listen.Accept()
        if err != nil {

            fmt.Println("accept failed, err:", err)
            continue
        }
        go process(conn) // 启动一个goroutine处理连接
    }
}
```

*** 客户端

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
)

func main() {
```

```
    // 第二个参数是服务端的网络地址和端口号
    conn, err := net.Dial("tcp", "192.168.1.13:8888")

    if err != nil {
        fmt.Println("err :", err)
        return
    }

    fmt.Println("已经建立连接，可以进行通信.......")

    defer conn.Close() // 关闭连接

    inputReader := bufio.NewReader(os.Stdin)

    for {
        input, _ := inputReader.ReadString('\n') // 读取用户输入

        inputInfo := strings.Trim(input, "\r\n")

        if strings.ToUpper(inputInfo) == "Q" { // 如果输入q就退出
            return
        }
        _, err = conn.Write([]byte(inputInfo)) // 发送数据
        if err != nil {
            return
        }

        buf := [512]byte{}
        n, err := conn.Read(buf[:])
        if err != nil {
            fmt.Println("recv failed, err:", err)
            return
        }

        fmt.Println(string(buf[:n]))
    }
}
```

2.3 TCP粘包问题

*** 粘包示例

Server3.go > process

```go
func process(conn net.Conn) {

    defer conn.Close()

    reader := bufio.NewReader(conn)
    var buf [1024]byte

    for {
        n, err := reader.Read(buf[:])

        if err == io.EOF {
            fmt.Println("EOF")
            break
        }
        if err != nil {
            fmt.Println("read from client failed, err:", err)
            break
        }
        recvStr := string(buf[:n])
        fmt.Println("收到client发来的数据:", recvStr)
    }
}

func main() {

    listen, err := net.Listen("tcp", "127.0.0.1:30000")
    if err != nil {
        fmt.Println("listen failed, err:", err)
        return
    }
    defer listen.Close()
    for {
        conn, err := listen.Accept()
        if err != nil {
            fmt.Println("accept failed, err:", err)
            continue
        }
        go process(conn)
```

tcp > Client3.go > ...

```go
package main

import (
    "fmt"
    "net"
)

// socket_stick/client/main.go

func main() {
    conn, err := net.Dial("tcp", "127.0.0.1:30000")
    if err != nil {
        fmt.Println("dial failed, err", err)
        return
    }
    defer conn.Close()
    for i := 0; i < 20; i++ {
        msg := "Hello, Hello. How are you?"
        conn.Write([]byte(msg))
    }
}
```


C:\Windows\System32\cmd.exe - go run Server3.go

```
F:\tools\golang\tcp>go run Server3.go
收到client发来的数据： Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?
收到client发来的数据： Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?
收到client发来的数据： Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?
EOF
```

C:\Windows\System32\cmd.exe

```
F:\tools\golang\tcp>go run Client3.go

F:\tools\golang\tcp>
```


*** 出现 粘包原因

主要原因就是tcp数据传递模式是流模式，在保持长连接的时候可以进行多次的收和发。

“粘包”可发生在发送端也可发生在接收端：

由`Nagle算法造成的发送端的粘包`：Nagle算法是一种改善网络传输效率的算法。简单来说就是当我们提交一段数据给TCP发送时，TCP并不立刻发送此段数据，而是等待一小段时间看看在等待期间是否还有要发送的数据，若有则会一次把这两段数据发送出去。

`接收端接收不及时造成的接收端粘包`：TCP会把接收到的数据存在自己的缓冲区中，然后通知应用层取数据。当应用层由于某些原因不能及时的把TCP的数据取出来，就会造成TCP缓冲区中存放了几段数据。

*** 解决办法

出现"粘包"的关键在于接收方不确定将要传输的数据包的大小，因此我们可以对数据包进行封包和拆包的操作。

封包：封包就是给一段数据加上包头，这样一来数据包就分为包头和包体两部分内容了(过滤非法包时封包会加入"包尾"内容)。包头部分的长度是固定的，并且它存储了包体的长度，根据包头长度固定以及包头中含有包体长度的变量就能正确的拆分出一个完整的数据包。

我们可以自己定义一个协议，比如数据包的前4个字节为包头，里面存储的是发送的数据的长度。

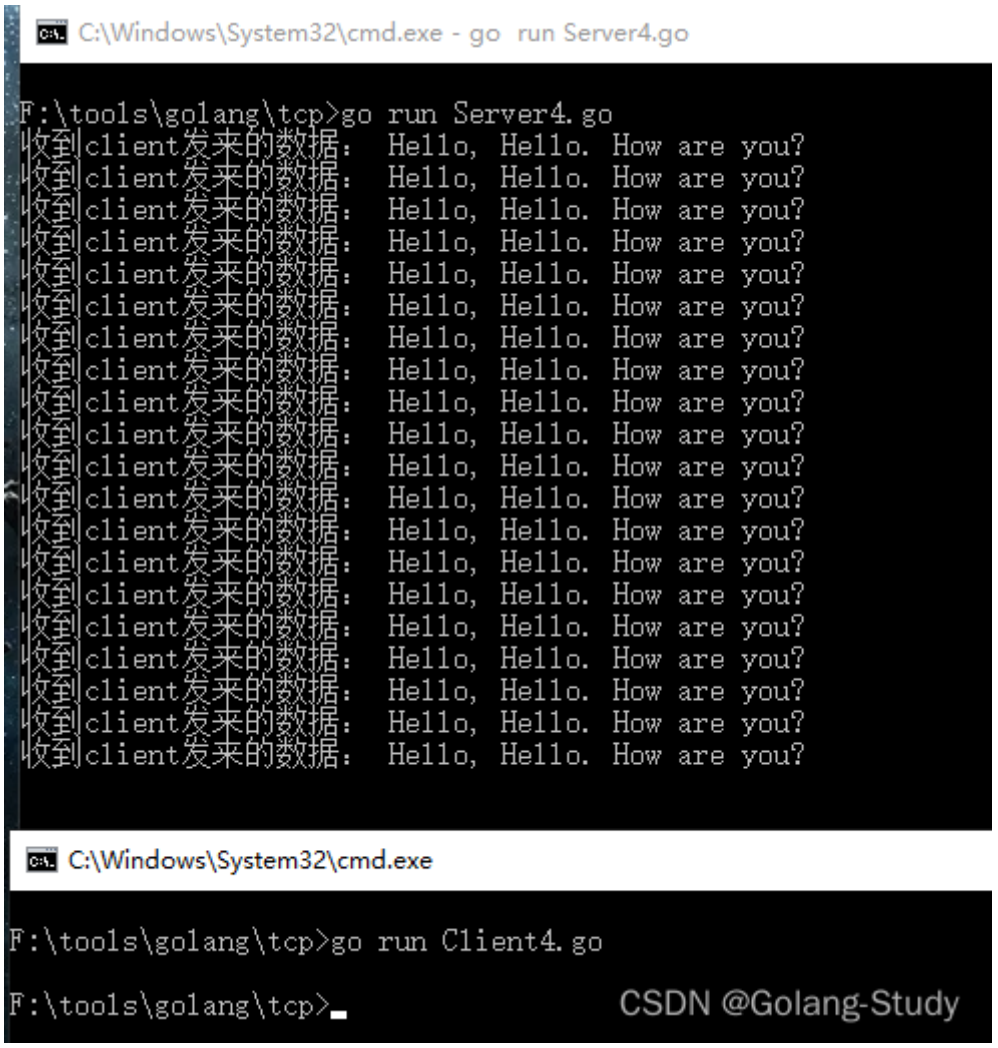
```go
// socket_stick/proto/proto.go
package proto

import (
    "bufio"
    "bytes"
    "encoding/binary"
)

// Encode 将消息编码
func Encode(message string) ([]byte, error) {
    // 读取消息的长度，转换成int32类型（占4个字节）
    var length = int32(len(message))
    var pkg = new(bytes.Buffer)
    // 写入消息头
    err := binary.Write(pkg, binary.LittleEndian, length)
    if err != nil {
        return nil, err
    }
    // 写入消息实体
    err = binary.Write(pkg, binary.LittleEndian, []byte(message))
    if err != nil {
        return nil, err
    }
    return pkg.Bytes(), nil
}

// Decode 解码消息
func Decode(reader *bufio.Reader) (string, error) {
    // 读取消息的长度
    lengthByte, _ := reader.Peek(4) // 读取前4个字节的数据
    lengthBuff := bytes.NewBuffer(lengthByte)
    var length int32
    err := binary.Read(lengthBuff, binary.LittleEndian, &length)
    if err != nil {
        return "", err
    }
    // Buffered返回缓冲中现有的可读取的字节数。
    if int32(reader.Buffered()) < length+4 {
        return "", err
    }

    // 读取真正的消息数据
    pack := make([]byte, int(4+length))
    _, err = reader.Read(pack)
```

```
    if err != nil {
        return "", err
    }
    return string(pack[4:]), nil
}
```

```
package main

import (
    "bufio"
    "bytes"
    "encoding/binary"
    "fmt"
    "io"
    "net"
)

// socket_stick/server2/main.go

func process(conn net.Conn) {
    defer conn.Close()
    reader := bufio.NewReader(conn)
    for {
        msg, err := Decode(reader)
        if err == io.EOF {
            return
        }
        if err != nil {
            fmt.Println("decode msg failed, err:", err)
            return
        }
        fmt.Println("收到client发来的数据：", msg)
    }
}

func main() {

    listen, err := net.Listen("tcp", "127.0.0.1:30000")
    if err != nil {
        fmt.Println("listen failed, err:", err)
        return
    }
    defer listen.Close()
    for {
        conn, err := listen.Accept()
        if err != nil {
            fmt.Println("accept failed, err:", err)
            continue
        }
        go process(conn)
    }
}
```

```
package main
```

```go
import (
    "bufio"
    "bytes"
    "encoding/binary"
    "fmt"
    "net"
)

// socket_stick/client2/main.go

func main() {
    conn, err := net.Dial("tcp", "127.0.0.1:30000")
    if err != nil {
        fmt.Println("dial failed, err", err)
        return
    }
    defer conn.Close()
    for i := 0; i < 20; i++ {
        msg := `Hello, Hello. How are you?`
        data, err := Encode(msg)
        if err != nil {
            fmt.Println("encode msg failed, err:", err)
            return
        }
        conn.Write(data)
    }
}
```



3、Go实现UDP通信

UDP协议（User Datagram Protocol）中文名称是用户数据报协议，是OSI（Open System Interconnection，开放式系统互联）参考模型中一种无连接的传输层协议，`不需要建立连接就能直接进行数据发送和接收`，属于不可靠的、没有时序的通信，但是UDP协议的实时性比较好，通常用于视频直播相关领域。

type UDPConn

```
type UDPConn struct {
    // 内含隐藏或非导出字段
}
```

UDPConn代表一个UDP网络连接，实现了Conn和PacketConn接口。

func DialUDP

客户端获取UDP连接对象，发送和接收数据都由该UDPConn的方法进行操作

```
func DialUDP(net string, laddr, raddr *UDPAddr) (*UDPConn, error)
```

DialTCP在网络协议net上连接本地地址laddr和远端地址raddr。net必须是"udp"、"udp4"、"udp6"；如果laddr不是nil，将使用它作为本地地址，否则自动选择一个本地地址。

func ListenUDP

发送方(服务端)由该函数获取一个UDPConn连接对象，发送和接收数据都由该UDPConn的方法进行操作

```
func ListenUDP(net string, laddr *UDPAddr) (*UDPConn, error)
```

ListenUDP创建一个接收目的地是本地地址laddr的UDP数据包的网络连接。net必须是"udp"、"udp4"、"udp6"；如果laddr端口为0，函数将选择一个当前可用的端口，可以用Listener的Addr方法获得该端口。返回的*UDPConn的ReadFrom和WriteTo方法可以用来发送和接收UDP数据包（每个包都可获得来源地址或设置目标地址）。

方法

func (*UDPConn) LocalAddr

```
func (c *UDPConn) LocalAddr() Addr
```

LocalAddr返回本地网络地址

func (*UDPConn) RemoteAddr

```
func (c *UDPConn) RemoteAddr() Addr
```

RemoteAddr返回远端网络地址

func (*UDPConn) Read

```
func (c *UDPConn) Read(b []byte) (int, error)
```

Read实现Conn接口Read方法

func (*UDPConn) ReadFrom

```
func (c *UDPConn) ReadFrom(b []byte) (int, Addr, error)
```

ReadFrom实现PacketConn接口ReadFrom方法

func (*UDPConn) ReadFromUDP

读取数据，通过参数获取，然后返回一个通信交互的连接，然后通过获取的连接，利用WriteToUDP作为参数写入

```
func (c *UDPConn) ReadFromUDP(b []byte) (n int, addr *UDPAddr, err error)
```

ReadFromUDP从c读取一个UDP数据包，将有效负载拷贝到b，返回拷贝字节数和数据包来源地址。

ReadFromUDP方法会在超过一个固定的时间点之后超时，并返回一个错误。

func (*UDPConn) Write

```
func (c *UDPConn) Write(b []byte) (int, error)
```

Write实现Conn接口Write方法

func (*UDPConn) WriteTo

```
func (c *UDPConn) WriteTo(b []byte, addr Addr) (int, error)
```

WriteTo实现PacketConn接口WriteTo方法

func (*UDPConn) WriteToUDP

```
func (c *UDPConn) WriteToUDP(b []byte, addr *UDPAddr) (int, error)
```

WriteToUDP通过c向地址addr发送一个数据包，b为包的有效负载，返回写入的字节。

WriteToUDP方法会在超过一个固定的时间点之后超时，并返回一个错误。在面向数据包的连接上，写入超时是十分罕见的。

Send.go > main

```go
// UDP Send端
func main() {
                        服务端获取UDPConn对象
    listen, err := net.ListenUDP("udp", &net.UDPAddr{
        IP:   net.IPv4(0, 0, 0, 0),
        Port: 30000,
    })

    if err != nil {
        fmt.Println("listen failed, err:", err)
        return
    }
    defer listen.Close()

    for {
        var data [1024]byte  读取数据，并且返回远程连接地址
        n, addr, err := listen.ReadFromUDP(data[:]) // 接收数据
        if err != nil {
            fmt.Println("read udp failed, err:", err)
            continue
        }
        fmt.Printf("data:%v addr:%v count:%v\n", string(data[:n]), addr, n)

                                向远程地址发送数据
        _, err = listen.WriteToUDP(data[:n], addr) // 发送数据
        if err != nil {
            fmt.Println("write to udp failed, err:", err)
            continue
        }
    }
}
```

tcp > Receive.go > main

```go
    "fmt"
    "net"
)

// UDP 客户端
func main() {

    // 客户端绑定服务端的ip地址和端口，进行连接   客户但获取UDPConn连接
    socket, err := net.DialUDP("udp", nil, &net.UDPAddr{
        IP:   net.IPv4(192, 168, 1, 13),
        Port: 30000,    确定服务端的ip和端口
    })
    if err != nil {
        fmt.Println("连接服务端失败,err:", err)
        return
    }

    defer socket.Close()

    sendData := []byte("Hello server")
    _, err = socket.Write(sendData) // 发送数据   通过连接发送数据
    if err != nil {
        fmt.Println("发送数据失败,err:", err)
        return
    }

    data := make([]byte, 4096) 返回远程连接的ip地址
    n, remoteAddr, err := socket.ReadFromUDP(data) // 接收数据
    if err != nil {
        fmt.Println("接收数据失败,err:", err)
        return
    }
    fmt.Printf("recv:%v addr:%v count:%v\n", string(data[:n]), remoteAddr, n)
}
```