如何进行网络编程

张华

64174234@qq.com

内容

1.1 如何进行TCP编程
1.2 如何进行UDP编程
1.3 如何进行WEB编程
1.4 如何在本地查看Go文档
1.5 总结
1.6 思考

1.1如何进行TCP编程

- 什么是TCP编程
  - 使用TCP协议(（Transmission Control Protocol)来传输数据的编程
- TCP协议——用于端到端,大量数据的可靠传输
  - 可靠传输：有确认重传机制，接收方收到后会发个确认，如果规定时间内收不到确认，发送方会重发
  - 面向连接的：通讯前建立连接，通讯完毕拆除连接
  - 面向字节流的：根据当前的网络状态决定以多少字节为单位组装数据，并决定何时发送这些数据
  - 流量控制：采用一种称为“滑动窗口”的方式进行流量控制，所谓窗口实际表示接收能力，用以限制发送方的发送速度。
  - 拥塞控制：发现网络堵车后，停止发送或减量发送
  - 用于FTP,TELNET,SMTP,POP3,HTTP等应用层协议
- Go的net包提供了对TCP操作的支持

1.1如何进行TCP编程(续)

- 应用1：如何返回本机的网络地址
  - func InterfaceAddrs() ([]Addr, error)
    - 返回值类型[]addr——如果成功获取,存放返回的地址
    - 返回值类型error——若获取失败,存放错误信息,否则为nil

```
1  package main
2
3  import (
4      "fmt"
5      "net"
6  )
7
8  func main() {
9      addr, err := net.InterfaceAddrs()
10     if err != nil {
11         fmt.Println(err)
12     }
13     fmt.Println(addr)
14 }
```

```
D:\go-dev\src>go run net1.go
[0.0.0.0 0.0.0.0 192.168.1.104 0.0.0.0]
```

1.1如何进行TCP编程(续)

- 应用2：如何获取主机所对应的IP地址
  - func LookupIP(host string) (addrs []IP, err error)
    - IP用来表示一个ip地址，定义为type IP []byte
    - addrs 为[]IP就可以表示多个ip地址了
    - err是erro类型r——若获取失败,存放错误信息,否则为nil

```
1  package main
2
3  import (
4      "fmt"
5      "net"
6  )
7
8  func main() {
9      ips, err := net.LookupIP("www.baidu.com")
10     if err != nil {
11         fmt.Println(err)
12     }
13     fmt.Println(ips)
14 }
```

```
D:\go-dev\src>go run net2.go
[61.135.169.121 61.135.169.125]
```

1.1如何进行TCP编程(续)

- 应用3：如何解析出TCP地址(带端口号的ip)

```go
package main

import (
    "fmt"
    "net"
)

func main() {
    ip, err := net.ResolveTCPAddr("tcp", "www.baidu.com:80")
    if err != nil {
        fmt.Println(err)
    }
    fmt.Println(ip)
}
```

```
D:\go-dev\src>go run net3.go
115.239.210.27:80
```

1.1如何进行TCP编程(续)

- 应用3：代码中的解析函数说明
  - func ResolveTCPAddr(net, addr string) (*TCPAddr, os.Error)
  - 该函数用来解析一个TCPAddr,第一个参数为,tcp,tcp4 或者tcp6,addr 是一个字符串，由主机名或IP 地址，以及":"后跟随着端口号组成，
    - 例如： "www.baidu.com:80" 或'127.0.0.1:8080"。如果地址是一个IPv6 地址，由于已经有冒号，主机部分，必须放在方括号内, 例如："[::1]:8080"。
  - TCPAddr 包含一个IP 和Port 端口号

```
type TCPAddr struct {
    IP   IP
    Port int
}
```

1.1如何进行TCP编程(续)

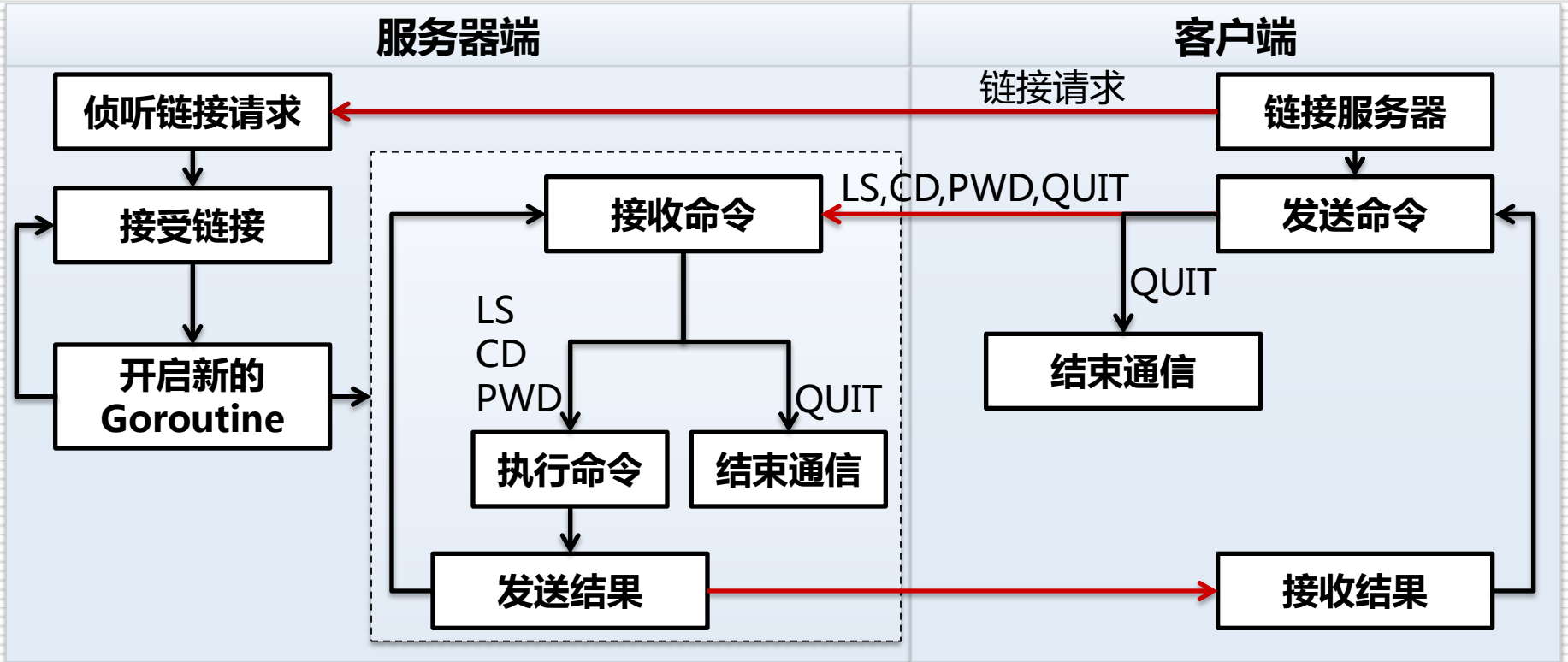
- 如何监听端口并接受客户端的请求
  - TCP程序分为服务端和客户端。服务端程序在某一端口监听客户端的链接请求，有客户端的链接请求时，读取客户端发了的数据，进行相关处理，然后关闭链接，ListenTCP函数就是在指定的端口监听，等待客户端的链接
    - func ListenTCP(net string, laddr *TCPAddr) (*TCPListener, error)
  - 监听成功后，接收数据前，先要接受客户端的请求
    - func (l *TCPListener) AcceptTCP() (*TCPConn, error)
      - 用来接受客户端的请求，返回一个Conn 链接，通过这个Conn 来与客户端的进行通信。
    - func (l *TCPListener) Accept() (Conn, error)
      - 与AcceptTCP 相同，这两个方法，用哪一个都可以。

1.1如何进行TCP编程(续)

- 如何发送数据和接收数据
  - func (c *TCPConn) Write(b []byte) (int, error)
    - 向TCPConn 网络链接发送数据，b 是要发送的内容，返回值int 为实际发送的字节数。
  - func (c *TCPConn) Read(b []byte) (int, error)
    - 从TCPConn 网络链接接收数据。返回值int 是实际接收的字节数。b 是接收的数据。
- 如何链接远程服务器
  - func DialTCP(net string, laddr, raddr *TCPAddr) (*TCPConn, error)
    - net 可以是tcp、tcp4、tcp6 中的一个。Laddr 为本地地址，通常传null，raddr 是要链接的远端服务器的地址。成功返回TCPConn，用返回的TCPConn 可以向服务器发送消息，读取服务器的响应信息。

1.1如何进行TCP编程(续)

- 应用4：一个简单的服务器客户端通信程序
  - 客户端可向服务器发送LS,CD,PWD,QUIT四种命令
    - LS：列出服务器端当前路径下的文件清单和子目录清单
    - CD：改变服务器端的当前路径
    - PWD：查看服务器端的当前路径
    - QUIT：结束与服务器的通信
  - 服务器端：接收命令,执行命令



1.1如何进行TCP编程(续)

- 应用4：一个简单的服务器客户端通信程序
  - 服务器端代码→包及全局变量

```
1  package main
2  import (
3      "bytes"
4      "fmt"
5      "io/ioutil"
6      "net"
7      "os"
8  )
9  const ( //将命令关键字定义为常量
10     LS   = "LS"
11     CD   = "CD"
12     PWD  = "PWD"
13     QUIT = "QUIT"
14 )
```

1.1如何进行TCP编程(续)

- 应用4：一个简单的服务器客户端通信程序
  - 服务器端代码→主函数

```go
func main() {
    //解析tcp地址
    tcpAddr, err := net.ResolveTCPAddr("tcp", ":7076")
    if err != nil { //解析出错
        fmt.Println(err)
        os.Exit(0)
    }
    //解析正确则侦听，在7076端口侦听
    listener, err := net.ListenTCP("tcp", tcpAddr)
    if err != nil { //监听失败
        fmt.Println(err)
        os.Exit(0)
    }
    //监听成功，等待连接提示
    fmt.Println("等待客户端的链接")
    for {
        //接收客户端的链接
        conn, err := listener.Accept()
        if err != nil { //接收失败
            /*通常服务端为一个服务，不会因为错误而退出。
            出错后，继续等待下一个链接请求*/
            fmt.Println(err)
            continue
        }
        //成功接收链接，并显示远程客户端的地址
        fmt.Println("收到链接，自客户端:", conn.RemoteAddr())
        //开启一个goroutine处理来自客户指令
        go ProcessClient(conn)
    }
}
```

1.1如何进行TCP编程(续)

- 应用4：一个简单的服务器客户端通信程序
  - 服务器端代码→处理客户端函数

```
//处理来自客户端的指令
func ProcessClient(conn net.Conn) {
    for {
        cmdstr := ReceiveCMD(conn)      //接收命令/获取命令
        res := ExecCMD_S(conn, cmdstr) //执行命令
        if !res {
            fmt.Println("与客户端:", conn.RemoteAddr(), "的通信结束")
            conn.Close() //关闭与客户端的链接
            break
        }
    }
}
```

1.1如何进行TCP编程(续)

- 应用4：一个简单的服务器客户端通信程序
  - 服务器端代码→接收命令函数

```
//接收命令/获取命令
func ReceiveCMD(conn net.Conn) string {
    cmdstr := ReadData(conn) //读取命令/接收数据
    //读取命令失败，则告诉客户接收数据出错
    if cmdstr == "" {
        SendData(conn, "获取命令失败，通信结束")
        fmt.Println("与客户端:", conn.RemoteAddr(), "的通信结束")
        conn.Close() //关闭链接
    }
    //读取成功
    fmt.Println("收到命令: ", cmdstr)
    return cmdstr
}
```

1.1如何进行TCP编程(续)

- 应用4：一个简单的服务器客户端通信程序
  - 服务器端代码→执行命令函数

```
//执行命令
func ExecCMD_S(conn net.Conn, cmdstr string) bool {
    var res bool = true //res返回值，默认true
    switch cmdstr {
    case LS: //列出当前目录的文件及文件夹清单
        ListDir(conn)
    case PWD: //给出当前目录
        Pwd(conn)
    case QUIT: //客户端结束链接指令
        res = false
    default: //其他情况
        if cmdstr[0:2] == CD { //前两个字节为CD的情况
            Chdir(conn, cmdstr[3:])
        } else { //命令不正确
            SendData(conn, "命令错误")
        }
    }
    return res
}
```

1.1如何进行TCP编程(续)

- 应用4：一个简单的服务器客户端通信程序
  - 服务器端代码→执行LS命令函数

```
//LS命令：列出当前路径下的文件及子文件夹
func ListDir(conn net.Conn) {
    //读出当期路径下的文件信息
    files, err := ioutil.ReadDir(".")
    if err != nil { //读取失败，并告知客户端
        SendData(conn, err.Error())
    }
    //读取成功
    var str string
    for i, j := 0, len(files); i < j; i++ {
        f := files[i]
        str += f.Name() + "\t"
        if f.IsDir() { //如果是文件夹
            str += "dir\r\n"
        } else { //如果是文件
            str += "file\r\n"
        }
    }
    //发送给客户端文件信息及子文件夹信息
    SendData(conn, str)
}
```

1.1如何进行TCP编程(续)

- 应用4：一个简单的服务器客户端通信程序
  - 服务器端代码→执行CD命令函数

```
//CD命令：改变当前路径
func Chdir(conn net.Conn, s string) {
    err := os.Chdir(s) //改变目录到指定目录
    if err != nil {    //改变失败，发送错误信息给客户端
        SendData(conn, err.Error())
    } else { //改变成功，发送OK给客户端
        SendData(conn, "OK")
    }
}
```

1.1如何进行TCP编程(续)

- 应用4：一个简单的服务器客户端通信程序
  - 服务器端代码→执行PWD命令函数

```
//PWD命令：获取服务器当前路径
func Pwd(conn net.Conn) {
    //获取当前工作目录
    s, err := os.Getwd()
    if err != nil { //获取失败，发送错误信息给客户端
        SendData(conn, err.Error())
    } else { //获取成功，将当前工作目录发给客户端
        SendData(conn, s)
    }
}
```

1.1如何进行TCP编程(续)

- 应用4：一个简单的服务器客户端通信程序
  - 服务器端代码→从客户端读取数据函数

```
//从客户端读取数据
func ReadData(conn net.Conn) string {
    var data bytes.Buffer
    var buf [512]byte
    for {
        n, err := conn.Read(buf[0:]) //n为字节数
        if err != nil {
            fmt.Println(err)
            return ""
        }
        //我们的数据以0做为结束的标记
        if buf[n-1] == 0 {
            //n-1去掉结束标记0
            data.Write(buf[0 : n-1])
            break
        } else {
            data.Write(buf[0:n])
        }
    }
    return string(data.Bytes())
}
```

1.1如何进行TCP编程(续)

- 应用4：一个简单的服务器客户端通信程序
  - 服务器端代码→向客户端发送数据函数

```
//发送数据给客户端
func SendData(conn net.Conn, data string) {
    buf := []byte(data)
    /*向byte 字节里添加结束标记*/
    buf = append(buf, 0)
    _, err := conn.Write(buf)
    if err != nil {
        fmt.Println(err)
    }
}
```

1.1如何进行TCP编程(续)

- 应用4：一个简单的服务器客户端通信程序
  - 客户端代码→包及全局变量

```
1  package main
2  import (
3      "bufio"
4      "bytes"
5      "fmt"
6      "net"
7      "os"
8      "strings"
9  )
10 const (
11     LS    = "LS"
12     CD    = "CD"
13     PWD   = "PWD"
14     QUIT = "QUIT"
15 )
```

1.1如何进行TCP编程(续)

- 应用4：一个简单的服务器客户端通信程序
  - 客户端代码→主函数

```
func main() {
    //录入服务器的TCP地址
    tcpAddrStr := InputTCPAddr()
    //TCP地址合法性检查
    tcpAddr, err := net.ResolveTCPAddr("tcp", tcpAddrStr)
    if err != nil { //TCP地址无效，退出系统
        fmt.Println(err)
        os.Exit(0)
    }
    //服务器TCP地址有效，则链接服务器。客户机地址为nil
    conn, err := net.DialTCP("tcp", nil, tcpAddr)
    if err != nil { //链接失败，退出系统
        fmt.Println(err)
        os.Exit(0)
    }
    //链接成功，conn为链接对象
    for { //循环录入并处理命令，直至退出
        cmd := InputCMD()                  //录入命令
        res := ProcessCMD_C(conn, cmd) //处理命令
        //res==false 时候，退出系统
        if !res {
            fmt.Println("与服务器:", conn.RemoteAddr(), "的通信结束")
            conn.Close() //关闭连接
            break
        }
    }
}
```

1.1如何进行TCP编程(续)

- 应用4：一个简单的服务器客户端通信程序
  - 客户端代码→录入服务器地址

```
//输入服务器的TCP地址
func InputTCPAddr() string {
    var TCPAddr string //TCP地址串
    fmt.Printf("请输入服务器的TCP地址(default: 127.0.0.1:7076):\n")
    fmt.Scanln(&TCPAddr)
    if len(TCPAddr) == 0 {
        TCPAddr = "127.0.0.1:7076"
    }
    return TCPAddr
}
```

1.1如何进行TCP编程(续)

- 应用4：一个简单的服务器客户端通信程序
  - 客户端代码→录入命令函数

```
//录入命令
func InputCMD() string {
    fmt.Printf("请输入命令:\n")
    //建立一个buffer reader (带缓冲的读取器)
    reader := bufio.NewReader(os.Stdin)
    //录入数据，以回车结束
    cmdline, err := reader.ReadString('\n')
    //录入失败，退出系统
    if err != nil {
        fmt.Println(err)
        os.Exit(0)
    }
    //录入成功，去掉两端的空格
    cmdline = strings.TrimSpace(cmdline)
    //统一转换成大写字母
    cmdline = strings.ToUpper(cmdline)
    return cmdline
}
```

1.1如何进行TCP编程(续)

- 应用4：一个简单的服务器客户端通信程序
  - 客户端代码→处理命令函数

```
//处理命令
func ProcessCMD_C(conn net.Conn, cmd string) bool {
    var res bool = true //res返回值，默认true
    //将命令按空格分隔成2部分
    cmdarr := strings.SplitN(cmd, " ", 2)
    switch cmdarr[0] { //根据第一部分决定执行哪些命令
    case LS:
        //连接成功后，发送命令
        SendData(conn, LS)
        //读取服务器端返回的结果，并输出
        fmt.Println(ReadData(conn))
    case CD:
        SendData(conn, CD+" "+strings.TrimSpace(cmdarr[1]))
        fmt.Println(ReadData(conn))
    case PWD:
        SendData(conn, PWD)
        fmt.Println(ReadData(conn))
    case QUIT:
        SendData(conn, QUIT)
        res = false
    default:
        fmt.Println("命令错误！")
    }
    return res
}
```

1.1如何进行TCP编程(续)

- 应用4：一个简单的服务器客户端通信程序
  - 客户端代码→向服务器发送数据/命令函数

```
//向服务器端发送数据
func SendData(conn net.Conn, data string) {
    buf := []byte(data)
    /*向byte 字节里添加结束标记*/
    buf = append(buf, 0)
    _, err := conn.Write(buf) //发送数据，使用conn的write方法
    if err != nil {           //发送数据出错
        fmt.Println(err)
    }
}
```

1.1如何进行TCP编程(续)

- 应用4：一个简单的服务器客户端通信程序
  - 客户端代码→接收服务器执行结果/数据函数

```
/*读取服务器端返回的数据*/
func ReadData(conn net.Conn) string {
    //data是个缓冲byte类型的缓冲器，这个缓冲器里存放着都是byte
    var data bytes.Buffer
    var buf [512]byte
    for {
        //接收数据到buf，使用conn.Read方法，n存储接收的字节数量
        n, err := conn.Read(buf[0:])
        if err != nil { //接收失败
            fmt.Println(err)
            return ""
        }
        //我们的数据以0做为结束的标记
        if buf[n-1] == 0 {
            //n-1去掉结束标记0
            data.Write(buf[0 : n-1])
            break
        } else {
            data.Write(buf[0:n])
        }
    }
    return string(data.Bytes())
}
```

1.2如何进行UDP编程

- 什么是UDP编程
  - 使用UDP协议(User Datagram Protocol)来传输数据的编程
- UDP协议→是面向无连接的、不可靠的数据报投递服务。
  - 当使用UDP 协议传输信息流时，用户应用程序必须负责解决数据报丢失、重复、排序，差错确认等问题。
  - 资源消耗小，处理速度快，通常音频、视频和普通数据在传送时使UDP 较多。如QQ 使用的就是UDP协议。
  - UDP 适用于一次只传少量数据的环境
    - 从理论上说，包含报头在内的数据报的最大长度为65535字节。不过，一些实际应用往往会限制数据报的大小，有时会降低到8192 字节。

1.2如何进行UDP编程(续)

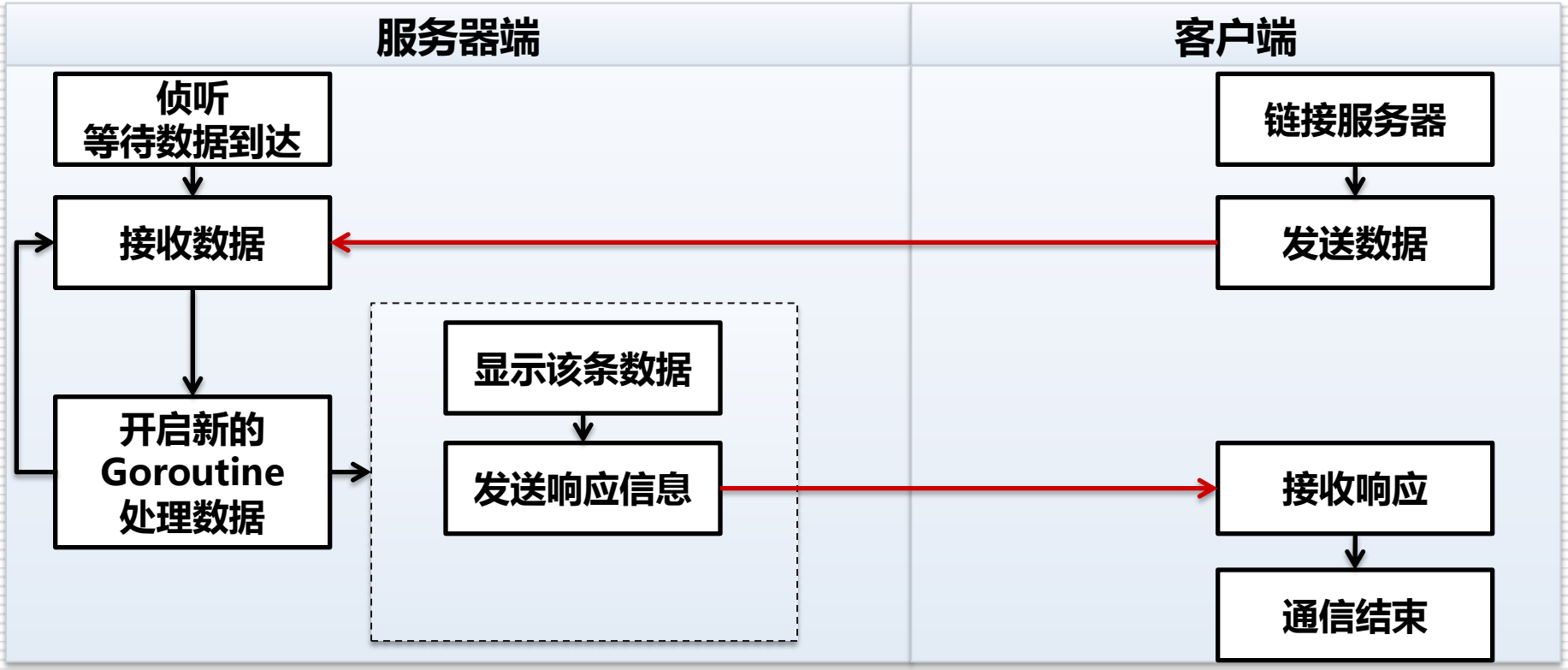
- net包常用的UDP库函数
  - func ResolveUDPAddr(net, addr string) (*UDPAddr, error)
    - 把addr 地址字符串，解析成UDPAddr 地址。
    - net 可以是”udp”,”udp4”,”udp6”
    - addr 是一个地址字符串，由主机名或IP 地址，以及“:”后面跟着的端口号组成。如果是IPv6，主机部分必须在方括内，如[::1]8080
  - func ListenUDP(net string, laddr *UDPAddr) (*UDPConn, error)
    - 在指定的地址(laddr)监听，等待UDP 数据包的到达。
    - 返回*UDPConn，可以使用连接的ReadFrom函数来读取UDP 数据，用WriteTo 来向客户端发送数据。
  - func (c *UDPConn) ReadFrom(b []byte) (int, Addr, error)
    - 服务端用来读取UDP 数据。Addr 是发送方的地址。

1.2如何进行UDP编程(续)

- net包常用的UDP库函数
  - func (c *UDPConn) WriteTo(b []byte, addr Addr) (int, error)
    - 向addr 发数据时用。b 是要发送的数据，addr 是接收方的地址。
  - func DialUDP(net string, laddr, raddr *UDPAddr) (*UDPConn, error)
    - 连接到远端服务器raddr，net 参数必须是“udp”,“udp4”,“udp6”中的一个
    - Laddr 通常为nil,如果不是nil将使用laddr来连接到服务端
  - func (c *UDPConn) Write(b []byte) (int, error)
    - 用来向服务端发送数据。
  - func (c *UDPConn) ReadFromUDP(b []byte) (n int, addr *UDPAddr, err error)
    - 与ReadFrom 相同，用来读取发来的UDP 数据。

1.2如何进行UDP编程(续)

- 简单的UDP应用举例
  - 客户端向服务端发送一条数据
  - 服务端收到数据后,给客户端一个响应,告知收到数据
  - 客户端收到响应后，关闭链接



1.2如何进行UDP编程(续)

- 简单的UDP应用举例
  - 服务器端代码——包及全局变量

```
1  package main
2
3  import (
4      "fmt"
5      "net"
6  )
```

1.2如何进行UDP编程(续)

- 简单的UDP应用举例
  - 服务器端代码——主函数

```
func main() {
    //解析udp地址
    addr, err := net.ResolveUDPAddr("udp", ":7070")
    if err != nil {
        fmt.Println(err)
        return
    }
    //在7070端口监听
    conn, err := net.ListenUDP("udp", addr)
    if err != nil {
        fmt.Println(err)
        return
    }
    for { //循环接收数据，处理数据
        var buf [1024]byte
        n, addr, err := conn.ReadFromUDP(buf[0:])
        if err != nil {
            fmt.Println(err)
            return
        }
        //开启新线程处理客户端的数据
        go HandleClient(conn, buf[0:n], addr)
    }
}
```

1.2如何进行UDP编程(续)

- 简单的UDP应用举例
  - 服务器端代码——处理客户端数据函数

```
//处理客户端数据
func HandleClient(conn *net.UDPConn, data []byte, addr *net.UDPAddr) {
    fmt.Println("收到数据: " + string(data))
    conn.WriteToUDP([]byte("OK,数据已到"), addr)
}
```

1.2如何进行UDP编程(续)

- 简单的UDP应用举例
  - 客户端代码——主函数

```
func main() {
    //解析服务器UDP地址
    addr, err := net.ResolveUDPAddr("udp", "127.0.0.1:7070")
    if err != nil {
        fmt.Println(err)
        return
    }
    //链接服务器
    conn, err := net.DialUDP("udp", nil, addr)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer conn.Close() //关闭链接
    //向服务器发送数据
    conn.Write([]byte("Hello Server"))
    var buf [1024]byte
    //读取服务器的响应信息
    n, _, err := conn.ReadFromUDP(buf[0:])
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(buf[0:n]))
}
```

1.2如何进行UDP编程(续)

- 简单的UDP应用举例
  - 客户端代码——包及全局变量

```
package main

import (
    "fmt"
    "net"
)
```

1.3如何进行WEB编程

- Go 可以用来开发WEB 应用程序。Go 内置实现了一个WEB服务，net/http 包提供了相应的实现。通常Go WEB 程序以反向代理的方式发布。
- 两个基本函数
  - func HandleFunc(pattern string, handler func(ResponseWriter, *Request))
    - 用来注册http 路由的处理函数，partten 是http 的地址，handler 是对应的处理函数。
  - func ListenAndServe(addr string, handler Handler) error
    - 在指定端口监听HTTP 请求，并阻塞程序，直到退出。

1.3如何进行WEB编程

- 小应用1

```
D:\go-dev\src>go run web1.go

D:\go-dev\src>_
```


```go
package main

import (
    "net/http"
)

func main() {
    http.HandleFunc("/test", processReq)
    http.ListenAndServe(":8888", nil)
}
func processReq(rw http.ResponseWriter, req *http.Request) {
    rw.Write([]byte("第一个WEB应用"))
    rw.Write([]byte(req.URL.Path))
}
```

1.3如何进行WEB编程

- ListenAndServe(addr string, handler Handler) 中的参数 Handler其实是一个接口，
  type Handler **interface** {
  ServeHTTP(ResponseWriter, *Request)
  }

```
D:\go-dev\src>go run web2.go
```

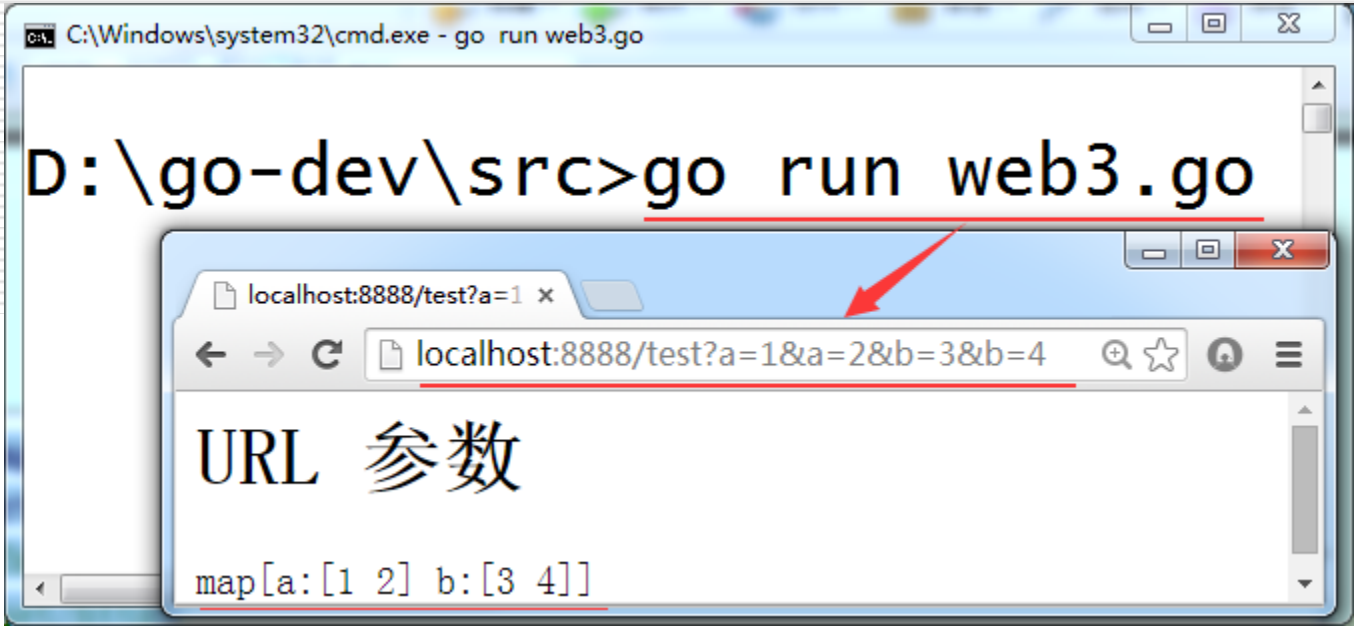
```go
package main
import "net/http"
type HttpHandler struct { //定义一个空类HttpHandler
}
//实现接口Handler中的ServeHTTP
func (this *HttpHandler) ServeHTTP(w http.ResponseWriter,
    r *http.Request) {
    w.Write([]byte("<h1>在ServeHTTP 里</h1>"))
    w.Write([]byte(r.URL.Path))
}
func main() {
    handler := &HttpHandler{}
    http.ListenAndServe(":8899", handler)
}
```

1.3如何进行WEB编程

- 小应用2——URL参数与form表单处理
- form表单：用于向服务器传送数据
  <form>…</form>
- http.Request.URL.Query()可以获取地址栏中的参数，返回Values 类型，即map[string][]string
  - 比如在地址栏中，先设置a=1，再设置a=10，这两个值将按先后顺序存到string 数组中，即map[“a”]=[]string{“1”,”2”}。

1.3如何进行WEB编程

- http.Request.URL.Query() 举例




```
package main
import (
    "fmt"
    "net/http"
)
func main() {
    http.HandleFunc("/test", HandleRequest)
    http.ListenAndServe(":8888", nil)
}
func HandleRequest(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("<h1>URL 参数</h1>"))
    w.Write([]byte(fmt.Sprintf("%v", r.URL.Query())))
}
```

1.3如何进行WEB编程

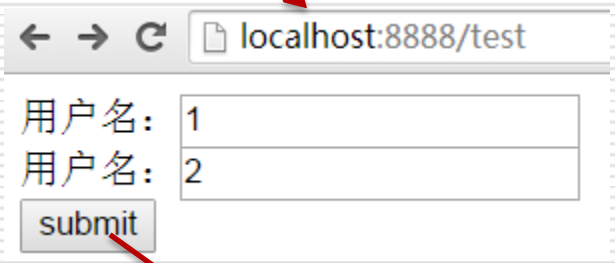
- func (r *Request) ParseForm() error
  - 解析URL 请求的参数并更新r.Form
  - 对于Post,Put 请求，还将解析POST 的内容，将结果放到r.PostForm 和r.Form 中
  - r.Form 中存放了URL 的参数值,和Post 的Form 值
  - r.PostForm 只存放了Post 的Form 值。
  - r.Form,r.PostForm 都是url.Values 类型，前面提到url.Values 是map[string][]string 类型。

1.3如何进行WEB编程

- func (r *Request) ParseForm()举例
  - 完整代码及结果

```
package main
import (
    "fmt"
    "net/http"
)
func main() {
    http.HandleFunc("/test", HandleRequest)
    http.ListenAndServe(":8888", nil)
}
func HandleRequest(w http.ResponseWriter, r *http.Request) {
    w.Header().Add("Content-Type", " text/html;charset=utf-8")
    if r.Method == "POST" {
        r.ParseForm()
        /*username 有两个值，默认取的是第一个的*/
        w.Write([]byte("用户名：" + r.FormValue("username") + "<br/>"))
        w.Write([]byte("<hr/>"))
        names := r.Form["username"]
        w.Write([]byte("username 有两个：" + fmt.Sprintf("%v", names)))
        w.Write([]byte("<hr/>r.Form 的内容：  " + fmt.Sprintf("%v",
            r.Form)))
        w.Write([]byte("<hr/>r.PostForm 的内容：" + fmt.Sprintf("%v",
            r.Form)))
        //r.Form
    } else {
        strBody := `<form action="` + r.URL.RequestURI() + `"
method="post">
用户名：<input name="username" type="text" /><br />
用户名：<input name="username" type="text" /><br />
<input id="Submit1" type="submit" value="submit" />
</form>`
        w.Write([]byte(strBody))
        r.ParseForm()
    }
}
```

```
D:\go-dev\src>go run web4.go
```



1.3如何进行WEB编程

- 主函数

```
package main

import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/test", HandleRequest)
    http.ListenAndServe(":8888", nil)
}
```

1.3如何进行WEB编程

- HandleRequest函数

```
func HandleRequest(w http.ResponseWriter, r *http.Request) {
    w.Header().Add("Content-Type", " text/html;charset=utf-8")
    if r.Method == "POST" {
        r.ParseForm()
        /*username 有两个值，默认取的是第一个的*/
        w.Write([]byte("用户名： " + r.FormValue("username") + "<br/>"))
        w.Write([]byte("<hr/>"))
        names := r.Form["username"]
        w.Write([]byte("username 有两个： " + fmt.Sprintf("%v", names)))
        w.Write([]byte("<hr/>r.Form 的内容：  " + fmt.Sprintf("%v",
            r.Form)))
        w.Write([]byte("<hr/>r.PostForm 的内容： " + fmt.Sprintf("%v",
            r.Form)))
        //r.Form
    } else {
        strBody := `<form action="` + r.URL.RequestURI() + `"
method="post">
用户名: <input name="username" type="text" /><br />
用户名: <input name="username" type="text" /><br />
<input id="Submit1" type="submit" value="submit" />
</form>`
        w.Write([]byte(strBody))
        r.ParseForm()
    }
}
```

1.3如何进行WEB编程

- 小应用3——文件上传功能的实现
  - Go 的文件上传处理十分方便，Request.FormFile返回一个multipart.File 对像，可以直接读取文件内容，并保存。定义如下
  - func (r *Request) FormFile(key string) (multipart.File, *multipart.FileHeader, error)
  - multipart.File 是一个接口，继承了io.Reader 接口，可以通过该接口读取上传文件的内容。
  - multipart.FileHeader 是一个结构体，可以通过该结构体取到上传文件的名称，文件类型，

```
type File interface {
    io.Reader
    io.ReaderAt
    io.Seeker
    io.Closer
}
```

```
type FileHeader struct {
    Filename string
    Header   textproto.MIMEHeader
    // contains filtered or unexported fields
}
```

1.3如何进行WEB编程

- 小应用3——主函数

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
)

func main() {
    http.HandleFunc("/", HelloServer)
    err := http.ListenAndServe(":12345", nil)
    if err != nil {
        fmt.Println(err)
    }
}
```

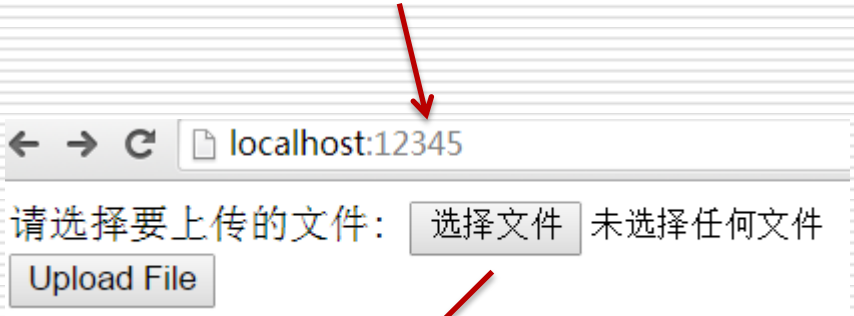
1.3如何进行WEB编程

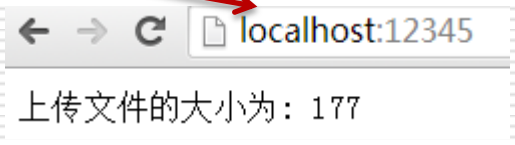
小应用3——HelloServer函数

```
func HelloServer(w http.ResponseWriter, r *http.Request) {
    if "POST" == r.Method { //如果提交了form
        //接收文件 file: 文件句柄，Handler: 文件的信息，
        file, handler, err := r.FormFile("file")
        if err != nil { //获取上传文件错误
            http.Error(w, err.Error(), 500)
            return
        }
        fmt.Println(handler.Header) //输出消息头部
        defer file.Close()
        f, err := os.OpenFile("./"+handler.Filename,
            os.O_WRONLY|os.O_CREATE, os.ModePerm)
        // ./ 当前路径 即当前项目的GOPATH
        //O_WRONLY以只写方式打开 WR==WRITE
        //O_CREATE 创建文件
        //os.Modeperm 每个人都有读和写以及执行的权限 0777
        if err != nil {
            fmt.Println(err)
            return
        }
        defer f.Close()
        size, err := io.Copy(f, file)
        //io.Copy适合于实时传输占用资源大
        //可以用ioutil.ReadAll
        if err != nil {
            fmt.Println(err)
            return
        }
        fmt.Fprintf(w, "上传文件的大小为: %d", size)
        return
    }
    // 上传页面
    //向http的响应头中加入内容
    w.Header().Add("Content-Type", "text/html;charset=UTF-8")
    //根据HTTP State Code来写Response的Header
    w.WriteHeader(200) //写入http的状态码到Header
    //200 代表请求已成功，请求所希望的响应头或数据体将随此响应返回。
    html := `
    <form enctype="multipart/form-data" action="/" method="POST">
    请选择要上传的文件: <input name="file" type="file" /><br/>
    <input type="submit" value="Upload File" />
    </form>`
    io.WriteString(w, html) //将html串写入w

}
```

```
D:\go-dev\src>go run web5.go
```



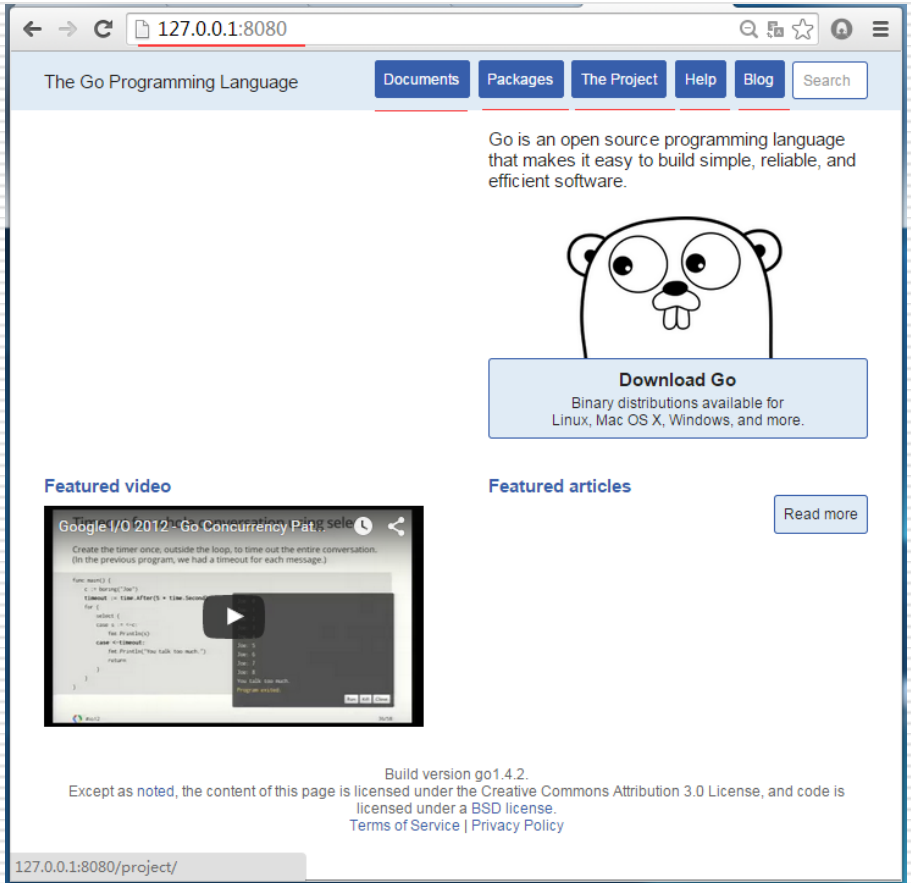



```
D:\go-dev\src>go run web5.go
map[Content-Disposition:[form-data; name="fi
nt-Type:[application/octet-stream]]
map[Content-Disposition:[form-data; name="fi
e:[application/octet-stream]]
```

1.4如何在本地查看Go文档

- CMD下运行 godoc -http=:8080然后在浏览器上输入 127.0.0.1:8080就可以访问本地Go DOC文档库

```
C:\Users\think>godoc -http=:8080
```



1.5总结

- TCP 协议需要通信双方约定数据的传输格式，否则接收方无法判断数据是否接收完成。
  - 1.1的代码中SendData 用来发送数据，发送的数据末尾要添加结束标记， buf= append(buf, 0)，用来表示数据发送结束。接收方收到0 说明数据接收完成。否则接收无法判断数据是否接收完。
  - 如果发送端，发送数据后，调用Close 关闭连接，不等待服务端的返回数据，服务端可以用ioutil.ReadAll 来读取数据，这时可以判断出EOF，读取结束。但如果客户端发送数据后，没有关闭，而是等待服务端的数据返回，用ReadAll 是不行的。所以1.1的例子中，用0 来示数据的发送完成

1.5总结(续)

- 在UDP 里，不需要有约定结束标记，但需要约定，UDP 报文的最大长度。UDP 的数据，必须一次接收完成。
  - 比如1.2的例子中，我们用，var buf [1024]byte，定义了1KB 的缓冲区来接收数据。因为我们发送的数据量不大，所以可以一次读取。
    - 如果把服务器端缓冲区改为两个字节var buf [2]byte 会怎么样？大家可以自己试一下，会报一个错误：
    - WSARecvFrom udp 0.0.0.0:7070: More data is available.
  - 所以，UDP 通信的双方需要约定报文的最大长度。

1.6思考

- 在1.1列子中，客户端和服务器端通信过程中，服务器端为客户端开启了专门的Goroutine，如果长时间不通信，对系统资源是一种浪费，请思考如何加入超时退出Goroutine的机制，解决长时间不通信带来的资源浪费问题。
- 在1.2中的客户端和服务器端的通信图中，每发送一条数据，服务器就要为客户端开启一个线程来处理数据，如果这个客户端再发送下一条数据的时候，服务器还是按照新客户来对待，还会再开启一个线程来处理客户的数据，如何解决将同一客户发送来的数据，放在同一个线程来处理的问题呢？

Thank you very much

Any comments and suggestions are beyond welcome