# Socket编程

本章重点为大家介绍如下的内容：

什么是Socket通信

Dail（）函数

网络七层协议

TCP/IP五层模型

TCP Socket通信

聊天室案例

网络上的两个程序通过一个双向的通信连接实现数据的交换，这个连接的一端称为一个socket。建立网络通信连接至少要一对端口号。比如甲要给乙打电话，通话双方甲乙当于相互通信的2个进程x，电话号码相当于网络地址。任何用户在通话之前，首先要占有一部手机，相当于申请一个Socket；同时要知道对方的号码，相当于对方有一个Socket。然后向对方拨号呼叫，相当于发出连接请求。假如对方空闲（相当于通信的另一主机开机且可以接受连接请求），接听电话，双方就可以正式通话，相当于连接成功。双方通话的过程，是一方向手机发出信号和对方从手机接收信号的过程，相当于向Socket发送数据和从socket接收数据。通话结束后，一方挂断电话相当于关闭Socket，撤消连接。一般用户只能感受到本地和对方号码的存在，建立通话的过程，话音传输的过程以及整个电话系统的技术细节对他都是透明的，这也与Socket机制非常相似。Socket利用网间网通信设施实现进程通信，但它对通信设施的细节毫不关心，只要通信设施能提供足够的通信能力，它就满足了。

## Dial()函数

Go语言标准库里提供的net包，支持基于IP层、TCP/UDP层及更高层面（如HTTP、FTP、SMTP）的网络操作，其中用于IP层的称为Raw Socket。net包对于网络I/O提供了便携式接口，包括TCP/IP,UDP，域名解析以及Unix Socket。尽管net包提供了大量访问底层的接口，但是大多数情况下，客户端仅仅只需要最基本的接口，例如Dial，LIsten，Accepte以及分配的conn连接和listener接口。 crypto/tls包使用相同的接口以及类似的Dial和Listen函数。下面对net包进行具体分析。

首先介绍其中常量：

const ( IPv4len = 4 IPv6len = 16 )

很容易看出这表示ip地址的长度（bytes），其中ipv4长度是4，ipv6地址长度是16

接下来讲一下IPv4和IPv6相关知识，了解的请直接跳过。

IP地址（Internet Protocol Address，称为互联网协议地址，简写为IP Address），是一种给主机在Internet上进行编址的方式。常见的IP地址，分为IPv4和IPv6地址两类。

IPv4地址：目前的全球因特网所采用的协议族是TCP/IP协议。IP是TCP/IP协议中网络层的协议，是TCP/IP协议族的核心协议。目前主要采用的IP协议的版本号是4(简称为IPv4)，发展至今已经使用了30多年。

IPv4

IPv4的地址位数为32位，也就是最多有2的32次方的网络设备可以联到Internet上。近十年来由于互联网的蓬勃发展，IP位址的需求量愈来愈大，使得IP位址的发放愈趋紧张，前一段时间，据报道IPV4的地址已经发放完毕，我们公司目前很多服务器的IP都是一个宝贵的资源。

地址格式类似这样：127.0.0.1 172.122.121.111。IPv4是由四位0~255的十进制数字组成，即从0.0.0.0~255.255.255.255，每组XXX代表小于或等于255的10进制数，IPv4地址总数为232，不过，一些地址是为特殊用途所保留的，如专用网络（约18百万个地址）和多播地址（约270百万个地址）。

此外还有几个特殊IP地址：

0.0.0.0只能做源地址

255.255.255.255是广播地址

127.x.x.x为环回地址，本机使用

专用地址：

10/8 地址范围：10.0.0.0——10.255.255.255，

172.16/12 地址范围：172.16.0.0——172.31.255.255，

192.168/16地址范围：192.168.0.0——192.168.255.255。

IPv6

IPv6是下一版本的互联网协议，也可以说是下一代互联网的协议，它是为了解决IPv4在实施过程中遇到的各种问题而被提出的，IPv6采用128位地址长度，几乎可以不受限制地提供地址。按保守方法估算IPv6实际可分配的地址，整个地球的每平方米面积上仍可分配1000多个地址。在IPv6的设计过程中除了一劳永逸地解决了地址短缺问题以外，还考虑了在IPv4中解决不好的其它问题，主要有端到端IP连接、服务质量（QoS）、安全性、多播、移动性、即插即用等。

地址格式类似这样：ac:ed:32:aa:12:bf

net包的Dial()函数用于创建网络连接，函数原型如下：

func Dial(net, addr string) (Conn, error)

其中net参数是网络协议的名字，addr参数是IP地址或域名；如连接成功，返回连接对象，否则返回error。

目前，Dial()函数支持如下几种网络协议："tcp"、"udp"、"ip"、"ip6"等，例如：

conn, err := net.Dial("tcp", "192.168.0.123:1234") 　　 // TCP连接

conn, err := net.Dial("udp", "192.168.0.132:4321") 　　// UDP连接

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

在成功连接连接后，可以使用conn的Write()和Read()方法读写数据。

实际上，Dial()函数是对DialTCP()、DialUDP()、DialIP()、DialUnix()函数的封装：

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) (c \*IPConn, err error)

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

下面是一个TCP示例程序：

func checkError(err error) {

if err != nil {

fmt.Fprintf(os.Stderr, "致命错误: %s", err.Error()) os.Exit(1)

}

}

func readFully(conn net.Conn) ([]byte, error) {

defer conn.Close() result := bytes.NewBuffer(nil)

var buf [512]byte

for { n, err := conn.Read(buf[0:]) result.Write(buf[0:n])

if err != nil {

if err == io.EOF {

Break

}

return nil, err

}

}

return result.Bytes(), nil

}

func main() {

addr := "127.0.0.1:80"

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

checkError(err)

\_, err = conn.Write([]byte("GET /api/v3/get HTTP/1.1\r\n\r\n")) checkError(err)

result, err := readFully(conn)

checkError(err)

fmt.Println(string(result))

os.Exit(0)

}

## TCP Socket 通信

网络七层协议

网络七层协议由下往上分别为物理层、数据链路层、网络层、传输层、会话层、表示层和应用层。其中物理层、数据链路层和网络层通常被称作媒体层，是网络工程师所研究的对象；传输层、会话层、表示层和应用层则被称作主机层，是用户所面向和关心的内容。

HTTP协议对应于应用层，TCP协议对应于传输层，IP协议对应于网络层，HTTP协议是基于TCP连接的,三者本质上没有可比性。 TCP/IP是传输层协议，主要解决数据如何在网络中传输；而HTTP是应用层协议，主要解决如何包装数据。Socket是应用层与TCP/IP协议族通信的中间软件抽象层，是它的一组接口。

TCP/IP五层模型

TCP/IP五层模型的协议分为：应用层、传输层、网络层、数据链路层和物理层。中继器、集线器、还有我们通常说的双绞线也工作在物理层；网桥（现已很少使用）、以太网交换机（二层交换机）、网卡（其实网卡是一半工作在物理层、一半工作在数据链路层）在数据链路层；路由器、三层交换机在网络层；传输层主要是四层交换机、也有工作在四层的路由器。

TCP/IP协议中的应用层处理七层模型中的第五层、第六层和第七层的功能。TCP/IP协议中的传输层并不能总是保证在传输层可靠地传输数据包，而七层模型可以做到。TCP/IP协议还提供一项名为UDP（用户数据报协议）的选择。UDP不能保证可靠的数据包传输。

以前使用Socket编程时，一般是如下步骤：

建立连接

绑定socket

监听

接受连接

接受/发送

服务器端代码：

package main

import (

    "net"

    "fmt"

)

func main() {

    //第一步：提供本机的地址

    tcpAddr, \_ := net.ResolveTCPAddr("tcp4", "10.0.154.238:9527")

    //第二步：提供监听对象

    listener, \_ := net.ListenTCP("tcp", tcpAddr)

    //第三步：监听端口，等待客户端链接

    fmt.Println("服务器端已经准备就绪，等待客户端链接......")

    for{

        conn, \_ := listener.Accept()

        fmt.Println("已有客户端%s接入......", conn.RemoteAddr())

        //第四步：读写数据

        //服务器读取客户端

        go handleData(conn)

    }

}

func handleData(conn net.Conn){

    for {

        bs := make([] byte, 512)

        n, \_ := conn.Read(bs) //阻塞的

        content:=string(bs[:n])

        if content=="over" ||n == 0{

            fmt.Println("客户端已经断开连接......")

            break

        }

        fmt.Println("客户端说：", content, n)

        //写给客户端

        conn.Write([] byte(content))

    }

//第五步：关闭

    conn.Close()

}

客户端代码：

package main

import (

    "net"

    "fmt"

)

func main() {

    //第一步：提供服务器地址

    tcpAddr, \_ := net.ResolveTCPAddr("tcp4", "10.0.154.238:9527")

    //第二步：连接服务器

    tcpConn, \_ := net.DialTCP("tcp", nil, tcpAddr)

    //\*TCPConn

    //数据交互

    handleData2(tcpConn)

}

func handleData2(tcpConn \*net.TCPConn){

    //第三步：读写

    for {

        //1）读取键盘输入

        line := ""

        fmt.Scanln(&line) //键盘输入，阻塞的

        fmt.Println("--",line,"--")

        //2）发送给服务器

        n,\_:=tcpConn.Write([] byte(line))

        fmt.Println("客户端写出数据：",n)

        if line == "over"{

            fmt.Println("客户端即将结束......")

            break

        }

        //读取服务器

        bs := make([] byte, 512)

        n,\_= tcpConn.Read(bs) //阻塞式

        fmt.Println("服务器说：", string(bs[:n]))

    }

//第四步：关闭

    tcpConn.Close()

}

## 聊天室案例

场景一：一对一聊天

服务器端代码：

package main

import (

    "net"

    "fmt"

    "os"

)

func main() {

    //第一步：定义服务端的端口：

    ip := ":1234"

    //第二步：获取服务器端的地址

    tcpAddr, err := net.ResolveTCPAddr("tcp4", ip)

    checkErr(err)

    fmt.Printf("%T\n", tcpAddr) //\*net.TCPAddr

    fmt.Println(tcpAddr) // :1234

    // 第三步：服务端需要监听该端口

    listener, err := net.ListenTCP("tcp4", tcpAddr)

    fmt.Printf("%T\n", listener) //\*net.TCPListener

    fmt.Println(listener)

    // 第四步：等待客户端的链接

    fmt.Println("等待客户端的链接......")

    map1 := make(map[string]net.Conn)

    done:=make(chan bool)

    for i := 0; i < 2; i++ {

        conn, \_ := listener.Accept()

        // 获取远程客户端的信息

        remoAddr := conn.RemoteAddr()

        fmt.Println("已有客户端接入：", i,remoAddr)

        map1[remoAddr.String()] = conn

        fmt.Println(map1)

        go sendData3(conn, map1,done)

    }

}

func sendData3(conn net.Conn, map1 map[string]net.Conn, done chan bool) {

    bs := make([] byte, 512)

    sendName := conn.RemoteAddr().String()

    for {

        count, \_ := conn.Read(bs)

        content := string(bs[:count])

        fmt.Println(sendName, "发来消息：", content)

        if content == "over" {

            //conn.Close()

            fmt.Println(sendName, "已经断开于服务器连接!")

            delete(map1, sendName)

            fmt.Println(map1)

            break

        }

        for key, v := range map1 {

            if key == sendName {

                continue

            }

            msg := sendName + "说：" + content

            v.Write([] byte(msg))

        }

    }

    if len(map1) ==0{

        done<-true

    }

}

func checkErr(err error) {

    if err != nil {

        fmt.Println("连接失败......")

        os.Exit(1) //程序退出，非0表示不正常退出

    }

}

客户端代码：

package main

import (

    "net"

    "fmt"

    "os"

)

func main() {

    //1.确定要访问的服务端的ip和端口

    service := "127.0.0.1:1234"

    tcpAddr,err:=net.ResolveTCPAddr("tcp4",service)

    fmt.Printf("%T\n",tcpAddr) //\*net.TCPAddr

    fmt.Println(tcpAddr) //192.168.1.128:1234

    checkErr(err)

    // 2.客户端申请链接服务器

    tcpConn, err:=net.DialTCP("tcp4",nil,tcpAddr)

    fmt.Printf("%T\n",tcpConn) //\*net.TCPConn

    fmt.Println(tcpConn) //\*net.TCPConn

    // 3.向服务器发送消息

    go sendData(tcpConn)

    line:=""

    for{

        fmt.Scanln(&line)

        tcpConn.Write([] byte(line))

        fmt.Println("客户端数据写完!")

        if line==""||line=="over"{

            fmt.Println("客户端已经断开于服务器连接!")

            break

        }

    }

    tcpConn.Close()

}

func sendData(tcpConn \*net.TCPConn){

    bs:=make([] byte, 512)

    for{

        count,\_:=tcpConn.Read(bs)

        if count ==0{

            fmt.Println("退出！")

            break

        }

        fmt.Println(string(bs[:count]) ,count)

    }

}

func checkErr(err error){

    if err != nil{

        fmt.Println("连接服务器失败！", err)

        os.Exit(0)

    }

}

场景二：一对多聊天

服务器端代码：

package main

import (

    "net"

    "fmt"

    "os"

)

func main() {

    //1.定义服务端的端口：

    ip := ":1234"

    //2：获取服务器端的地址

    tcpAddr, err := net.ResolveTCPAddr("tcp4", ip)

    checkErr(err)

    fmt.Printf("%T\n", tcpAddr) //\*net.TCPAddr

    fmt.Println(tcpAddr) // :1234

    //3：服务端需要监听该端口

    listener, err := net.ListenTCP("tcp4", tcpAddr)

    fmt.Printf("%T\n", listener) //\*net.TCPListener

    //4：等待客户端的链接

    fmt.Println("服务器准备就绪，等待客户端连接......")

    map1 := make(map[string]net.Conn)

    for {

        conn, \_ := listener.Accept()

        // 获取远程客户端的信息

        remoAddr := conn.RemoteAddr()

        fmt.Println("已有客户端接入：", remoAddr)

        map1[remoAddr.String()] = conn

        fmt.Println(map1)

        go sendData3(conn, map1)

    }

}

func sendData3(conn net.Conn, map1 map[string]net.Conn) {

    bs := make([] byte, 512)

    sendName := conn.RemoteAddr().String()

    for {

        count, \_ := conn.Read(bs)

        content := string(bs[:count])

        fmt.Println(sendName,"发来消息：", content)

        if content == "over" {

            fmt.Println(sendName, "已经断开于服务器连接")

            delete(map1, sendName)

            fmt.Println(map1)

            break

        }

        for key, v := range map1 {

            if key == sendName {

                continue

            }

            msg := sendName + "说：" + content

            v.Write([] byte(msg))

        }

    }

}

func checkErr(err error) {

    if err != nil {

        fmt.Println("服务器发生错误！")

        os.Exit(1) //程序退出，非0表示不正常退出

    }

}

客户端代码：

package main

import (

    "net"

    "fmt"

    "os"

)

func main() {

    //1.确定要访问的服务端的ip和端口

    service := "127.0.0.1:1234"

    tcpAddr,err:=net.ResolveTCPAddr("tcp4",service)

    fmt.Printf("%T\n",tcpAddr) //\*net.TCPAddr

    fmt.Println(tcpAddr)

    checkErr(err)

    // 2.客户端申请链接服务器

    tcpConn, err:=net.DialTCP("tcp4",nil,tcpAddr)

    fmt.Printf("%T\n",tcpConn) //\*net.TCPConn

    fmt.Println(tcpConn) //\*net.TCPConn

    // 3.向服务器发送消息

    go sendData(tcpConn)

    line:=""

    for{

        fmt.Scanln(&line)

        tcpConn.Write([] byte(line))

        fmt.Println("客户端数据写完!")

        if line==""||line=="over"{

            fmt.Println("客户端退出程序!")

            break

        }

    }

//关闭

    tcpConn.Close()

}

func sendData(tcpConn \*net.TCPConn){

    bs:=make([] byte, 512)

    for{

        count,\_:=tcpConn.Read(bs)

        if count ==0{

            fmt.Println("退出！")

            break

        }

        fmt.Println(string(bs[:count]) ,count)

    }

}

func checkErr(err error){

    if err != nil{

        fmt.Println("错误了！", err)

        os.Exit(0)

    }

}

# HTTP编程

本章重点为大家介绍如下的内容：

什么是HTTP协议

HTTP状态码有哪些

HTTP客户端请求方法

GET方法和POST方法的区别

HTTP服务器端实现

HTTP客户端实现

服务器端如何使用ServeMux分发任务

服务器端如何获取客户端数据

常见案例

## HTTP协议

HTTP协议（HyperText Transfer Protocol，超文本传输协议）是用于从WWW服务器传输超文本到本地浏览器的传送协议，定义了客户端和服务端之间请求和响应的传输标准。它可以使浏览器更加高效，使网络传输减少。它不仅保证计算机正确快速地传输超文本文档，还确定传输文档中的哪一部分，以及哪部分内容首先显示(如文本先于图形)等。HTTP是一个应用层协议，由请求和响应构成，是一个标准的客户端服务器模型。HTTP是一个无状态的协议。

HTTP协议通常承载于TCP协议之上，有时也承载于TLS或SSL协议层之上，这个时候，就成了我们常说的HTTPS。HTTPS协议是以安全为目标的HTTP通道，简单讲是HTTP的安全版，即HTTP下加入SSL层，HTTPS的安全基础是SSL，因此加密的详细内容就需要SSL。HTTPS协议的主要作用可以分为两种：一种是建立一个信息安全通道，来保证数据传输的安全；另一种就是确认网站的真实性。

　　HTTP协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安全，为了保证这些隐私数据能加密传输，于是网景公司设计了SSL（Secure Sockets Layer）协议用于对HTTP协议传输的数据进行加密，从而就诞生了HTTPS。简单来说，HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全。

　　HTTPS和HTTP的区别主要如下：

https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。

http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

HTTP默认的端口号为80，HTTPS的端口号为443。

HTTP协议是一个无状态的协议，同一个客户端的这次请求和上次请求是没有对应关系。

HTTP状态码

HTTP状态码（HTTP Status Code）是用以表示网页[服务器](file://localhost/item/%25E6%259C%258D%25E5%258A%25A1%25E5%2599%25A8)HTTP响应状态的数字代码。当我们从客户端向服务器发送请求时，服务器向我们返回状态码，状态码就是告诉我们服务器响应的状态 ，通过它，我们就可以知道当前请求是成功了还是出现了什么问题。

状态码是由3位数字和原因短语组成的（比如最常见的：200 OK） ，其中第一位数字表示响应类别，响应类别从1到5分为五种 。

|  |  |  |
| --- | --- | --- |
| 状态码 | 响应类别 | 原因短语 |
| 1XX | 信息性状态码（Informational） | 服务器正在处理请求 |
| 2XX | 成功状态码（Success） | 请求已正常处理完毕 |
| 3XX | 重定向状态码（Redirection） | 需要进行额外操作以完成请求 |
| 4XX | 客户端错误状态码（Client Error） | 客户端原因导致服务器无法处理请求 |
| 5XX | 服务器错误状态码（Server Error） | 服务器原因导致处理请求出错 |

状态码响应类别的的定义是必须遵守的，后面的两个数字服务器自己DIY也没什么问题

下面列举了一些状态码，作为程序员应该了解。

2开头 （请求成功）表示成功处理了请求的状态代码。

200   （成功）  服务器已成功处理了请求。 通常，这表示服务器提供了请求的网页。

201   （已创建）  请求成功并且服务器创建了新的资源。

202   （已接受）  服务器已接受请求，但尚未处理。

203   （非授权信息）  服务器已成功处理了请求，但返回的信息可能来自另一来源。

204   （无内容）  服务器成功处理了请求，但没有返回任何内容。

205   （重置内容） 服务器成功处理了请求，但没有返回任何内容。

206   （部分内容）  服务器成功处理了部分 GET 请求。

3开头 （请求被重定向）表示要完成请求，需要进一步操作。 通常，这些状态代码用来重定向。

300   （多种选择）  针对请求，服务器可执行多种操作。 服务器可根据请求者 (user agent) 选择一项操作，或提供操作列表供请求者选择。

301   （永久移动）  请求的网页已永久移动到新位置。 服务器返回此响应（对 GET 或 HEAD 请求的响应）时，会自动将请求者转到新位置。

302   （临时移动）  服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。

303   （查看其他位置） 请求者应当对不同的位置使用单独的 GET 请求来检索响应时，服务器返回此代码。

304   （未修改） 自从上次请求后，请求的网页未修改过。 服务器返回此响应时，不会返回网页内容。

305   （使用代理） 请求者只能使用代理访问请求的网页。 如果服务器返回此响应，还表示请求者应使用代理。

307   （临时重定向）  服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。

4开头 （请求错误）这些状态代码表示请求可能出错，妨碍了服务器的处理。

400   （错误请求） 服务器不理解请求的语法。

401   （未授权） 请求要求身份验证。 对于需要登录的网页，服务器可能返回此响应。

403   （禁止） 服务器拒绝请求。

404   （未找到） 服务器找不到请求的网页。

405   （方法禁用） 禁用请求中指定的方法。

406   （不接受） 无法使用请求的内容特性响应请求的网页。

407   （需要代理授权） 此状态代码与 401（未授权）类似，但指定请求者应当授权使用代理。

408   （请求超时）  服务器等候请求时发生超时。

409   （冲突）  服务器在完成请求时发生冲突。 服务器必须在响应中包含有关冲突的信息。

410   （已删除）  如果请求的资源已永久删除，服务器就会返回此响应。

411   （需要有效长度） 服务器不接受不含有效内容长度标头字段的请求。

412   （未满足前提条件） 服务器未满足请求者在请求中设置的其中一个前提条件。

413   （请求实体过大） 服务器无法处理请求，因为请求实体过大，超出服务器的处理能力。

414   （请求的 URI 过长） 请求的 URI（通常为网址）过长，服务器无法处理。

415   （不支持的媒体类型） 请求的格式不受请求页面的支持。

416   （请求范围不符合要求） 如果页面无法提供请求的范围，则服务器会返回此状态代码。

417   （未满足期望值） 服务器未满足"期望"请求标头字段的要求。

5开头（服务器错误）这些状态代码表示服务器在尝试处理请求时发生内部错误。 这些错误可能是服务器本身的错误，而不是请求出错。

500   （服务器内部错误）  服务器遇到错误，无法完成请求。

501   （尚未实施） 服务器不具备完成请求的功能。 例如，服务器无法识别请求方法时可能会返回此代码。

502   （错误网关） 服务器作为网关或代理，从上游服务器收到无效响应。

503   （服务不可用） 服务器目前无法使用（由于超载或停机维护）。 通常，这只是暂时状态。

504   （网关超时）  服务器作为网关或代理，但是没有及时从上游服务器收到请求。

505   （HTTP 版本不受支持） 服务器不支持请求中所用的 HTTP 协议版本。

## HTTP客户端

Go语言中提供了net/http库来实现网络请求，我们在import中需要导入我们用到的库。

HTTP请求方法:

HTTP请求方法指定了客户端想对指定的资源/服务器作何种操作，本书重点讲解其中两种GET方法和POST方法。

 GET方法用来请求已被URI识别的资源。指定的资源经服务器端解析后返回响应内容（也就是说，如果请求的资源是文本，那就保持原样返回；如果是CGI[通用网关接口]那样的程序，则返回经过执行后的输出结果）。最常用于向服务器查询某些信息。必要时，可以将查询字符串参数追加到URL末尾，以便将信息发送给服务器。golang要请求远程网页，可以使用net/http包中的client提供的方法实现。

GET方法：

1、http.Get示例：

func httpGet() {

    resp, err := http.Get("http://www.webxml.com.cn/WebServices/WeatherWebService.asmx/getWeatherbyCityName?theCityName=北京")

    if err != nil {

        // handle error

    }

    defer resp.Body.Close()

    body, err := ioutil.ReadAll(resp.Body)

    if err != nil {

        // handle error

    }

     fmt.Println(string(body))

}

2、http.NewRequest示例：

package \_client

import (

    "net/http"

    "log"

    "fmt"

    "io/ioutil"

)

func main() {

    urlStr := "http://www.webxml.com.cn/WebServices/WeatherWebService.asmx/getWeatherbyCityName?theCityName=北京"

    client := http.Client{}

    request, err := http.NewRequest("GET", urlStr, nil)

    if err != nil {

        log.Fatal(err)

    }

    response, err := client.Do(request)

    if err != nil {

        log.Fatal(err)

    }

    defer response.Body.Close()

    if response.StatusCode == 200 {

        body, err := ioutil.ReadAll(response.Body)

        if err != nil {

            log.Fatal(err)

        }

        fmt.Println(string(body))

    }

    fmt.Printf("response:%+v\n", response)

    fmt.Printf("response.Header:%+v\n", response.Header)

    fmt.Printf("response.Cookies:%+v\n", response.Cookies())

}

package main

import (

    "net/http"

    "log"

    "fmt"

    "io/ioutil"

)

3、http.clinet.Get示例：

func main() {

    urlStr := "http://www.webxml.com.cn/WebServices/WeatherWebService.asmx/getWeatherbyCityName?theCityName=北京"

    clinet := http.Client{}

    response, err := clinet.Get(urlStr)

    if err != nil {

        log.Fatal(err)

    }

    defer response.Body.Close()

    if response.StatusCode == 200 {

        body, err := ioutil.ReadAll(response.Body)

        if err != nil {

            log.Fatal(err)

        }

        fmt.Println(string(body))

    }

    fmt.Printf("%+v", response)

}

2、POST方法：

POST方法用来传输实体的主体。虽然用GET方法也可以传输实体的主体，但一般不用GET方法进行传输，而是用POST方法；虽然GET方法和POST方法很相似，但是POST的主要目的并不是获取响应的主体内容。POST请求的主体可以包含非常多的数据，而且格式不限。

1、http.Post示例1：

func httpPost() {

    resp, err := http.Post("http://www.baidu.com/test/post.html",

        "application/x-www-form-urlencoded",

        strings.NewReader("id=1"))

    if err != nil {

        fmt.Println(err)

    }

    defer resp.Body.Close()

    body, err := ioutil.ReadAll(resp.Body)

    if err != nil {

        // handle error

    }

    fmt.Println(string(body))

}

注意：使用这个方法的话，第二个参数要设置成”application/x-www-form-urlencoded”，否则post参数无法传递。

http.Post示例2：

package main

import (

    "net/http"

    "log"

    "fmt"

    "io/ioutil"

    "net/url"

    "bytes"

)

func main() {

    urlStr := "http://www.webxml.com.cn/WebServices/WeatherWebService.asmx/getWeatherbyCityName"

    param := url.Values{

        "theCityName": {"北京"},

    }

    requestBody := bytes.NewBufferString(param.Encode())

    response, err := http.Post(urlStr, "application/x-www-form-urlencoded", requestBody)

    if err != nil {

        log.Fatal(err)

    }

    defer response.Body.Close()

    if response.StatusCode == 200 {

        body, err := ioutil.ReadAll(response.Body)

        if err != nil {

            log.Fatal(err)

        }

        fmt.Println(string(body))

    }

    fmt.Printf("%+v\n", response)

    fmt.Printf("%+v\n", response.Body)

    fmt.Printf("%+v\n", response.Header)

    fmt.Printf("%+v\n", response.StatusCode)

    fmt.Printf("%+v\n", response.Status)

    fmt.Printf("%+v\n", response.Request)

}

2、http.NewRequest示例：

package main

import (

    "net/http"

    "log"

    "fmt"

    "io/ioutil"

    "bytes"

    "net/url"

)

func main() {

    urlStr := "http://www.webxml.com.cn/WebServices/WeatherWebService.asmx/getWeatherbyCityName"

    client := http.Client{}

    param := url.Values{

        "theCityName": {"北京"},

    }

    requestBody := bytes.NewBufferString(param.Encode())

    request, err := http.NewRequest("POST", urlStr, requestBody)

    if err != nil {

        log.Fatal(err)

    }

    cookId := &http.Cookie{Name:"id", Value:"1", Domain:"baidu.com"}

    cookName := &http.Cookie{Name:"name", Value:"go"}

    request.AddCookie(cookId)

    request.AddCookie(cookName)

    //使用text/html就会出错

    request.Header.Set("Content-Type", "application/x-www-form-urlencoded")

    response, err := client.Do(request)

    if err != nil {

        log.Fatal(err)

    }

    defer response.Body.Close()

    if response.StatusCode == 200 {

        body, err := ioutil.ReadAll(response.Body)

        if err != nil {

            log.Fatal(err)

        }

        fmt.Println(string(body))

    }

    fmt.Printf("response:%+v\n", response)

    fmt.Printf("response.Header:%+v\n", response.Header)

    fmt.Printf("response.Cookies:%+v\n", response.Cookies())

    fmt.Printf("request.Header:%+v\n", request.Header)

    fmt.Printf("request.Cookies:%+v\n", request.Cookies())

}

3、http.client示例：

package \_client

import (

    "net/http"

    "log"

    "fmt"

    "io/ioutil"

    "bytes"

    "net/url"

)

func main() {

    urlStr := "http://www.webxml.com.cn/WebServices/WeatherWebService.asmx/getWeatherbyCityName"

    clinet := http.Client{}

    param := &url.Values{

        "theCityName": {"北京"},

    }

    requestBody := bytes.NewBufferString(param.Encode())

    response, err := clinet.Post(urlStr, "application/x-www-form-urlencoded", requestBody)

    if err != nil {

        log.Fatal(err)

    }

    defer response.Body.Close()

    if response.StatusCode == 200 {

        body, err := ioutil.ReadAll(response.Body)

        if err != nil {

            log.Fatal(err)

        }

        fmt.Println(string(body))

    }

    fmt.Printf("%+v", response)

}

4、http.PostForm方法示例

func httpPostForm() {

    resp, err := http.PostForm("http://www.baidu.com/test/post.html",

        url.Values{"key": {"Value"}, "id": {"1"}})

    if err != nil {

        // handle error

    }

    defer resp.Body.Close()

    body, err := ioutil.ReadAll(resp.Body)

    if err != nil {

        // handle error

    }

    fmt.Println(string(body))

 }

5、复杂的请求

有时需要在请求的时候设置头参数、cookie之类的数据，就可以使用http.Do方法。

func httpDo() {

    client := &http.Client{}

    req, err := http.NewRequest("POST", "http://www.baidu.com", strings.NewReader("id=1"))

    if err != nil {

        // handle error

    }

    req.Header.Set("Content-Type", "application/x-www-form-urlencoded")

    req.Header.Set("Cookie", "name=go")

    resp, err := client.Do(req)

    defer resp.Body.Close()

    body, err := ioutil.ReadAll(resp.Body)

    if err != nil {

        // handle error

    }

    fmt.Println(string(body))

}

同上面的post请求，必须要设定Content-Type为application/x-www-form-urlencoded，post参数才可正常传递。

**3、GET方法和POST方法区别**

下面的表格比较了两种 HTTP 方法：GET 和 POST

|  |  |  |
| --- | --- | --- |
|  | GET | POST |
| 后退按钮/刷新 | 无害 | 数据会被重新提交（浏览器应该告知用户数据会被重新提交） |
| 书签 | 可收藏为书签 | 不可收藏为书签 |
| 缓存 | 能被缓存 | 不能缓存 |
| 编码类型 | application/x-www-form-rulencoded | application/x-www-form-urlencoded 或 multipart/form-data。为二进制数据使用多重编码 |
| 历史 | 参数保留在浏览器历史中 | 参数不会保留在浏览器历史中 |
| 对数据长度的限制 | 是的。当发送数据时，GET方法向URL添加数据；URL的长度是受限制的（URL的最大长度是2048个字符） | 无限制 |
| 对数据类型的限制 | 只允许ASCII字符 | 没有限制。也允许二进制数据 |
| 安全性 | 与POST相比，GET安全性较差，因为发送的数据是URL的一部分。在发送密码或其他敏感信息时绝不要使用GET | POST比GET更安全，因为参数不会被保存在浏览器历史或WEB服务器日志中 |

## HTTP服务端

Go语言标准库内建提供了net/http包，涵盖了HTTP客户端和服务端的具体实现。使用net/http包，我们可以很方便地编写HTTP客户端或服务端的程序。

下面看以下几个服务器端的例子：

示例1：

package main

import (

"fmt"

"net/http"

)

//服务端编写的业务逻辑处理程序

//hander函数： 具有func(w http.ResponseWriter, r \*http.Requests)签名的函数

func myHandler(w http.ResponseWriter, r \*http.Request) {

fmt.Println(r.RemoteAddr, "连接成功") //r.RemoteAddr远程网络地址

fmt.Println("method = ", r.Method) //请求方法

fmt.Println("url = ", r.URL.Path)

fmt.Println("header = ", r.Header)

fmt.Println("body = ", r.Body)

w.Write([]byte("hello world")) //给客户端回复数据

}

func main() {

http.HandleFunc("/go", myHandler)

//该方法用于在指定的 TCP 网络地址 addr 进行监听，然后调用服务端处理程序来处理传入的连接请求。

//该方法有两个参数：第一个参数 addr 即监听地址；第二个参数表示服务端处理程序，通常为空

//第二个参数为空意味着服务端调用 http.DefaultServeMux 进行处理

http.ListenAndServe("127.0.0.1:1234", nil)

}

客户端测试1:

在浏览器中输入：<http://127.0.0.1:1234/go>

结果： hello world

客户端测试2:

package main

import (

"fmt"

"io/ioutil"

"net/http"

)

func main() {

response, \_ := http.Get("<http://127.0.0.1:1234/go>")

defer response.Body.Close()

body, \_ := ioutil.ReadAll(response.Body)

fmt.Println(string(body))

}

结果： hello world

示例2：

package main

import (

    "fmt"

    "net/http"

    "strings"

    "log"

)

func sayhelloName(w http.ResponseWriter, r \*http.Request) {

    r.ParseForm() //解析参数，默认是不会解析的

    fmt.Println(r.Form) //这些信息是输出到服务器端的打印信息

    fmt.Println("path", r.URL.Path)

    fmt.Println("scheme", r.URL.Scheme)

    fmt.Println(r.Form["url\_long"])

    for k, v := range r.Form {

        fmt.Println("key:", k)

        fmt.Println("val:", strings.Join(v, ""))

    }

    fmt.Fprintf(w, "Hello Wrold!") //这个写入到w的是输出到客户端的

}

func main() {

    http.HandleFunc("/", sayhelloName) //设置访问的路由

    err := http.ListenAndServe(":9090", nil) //设置监听的端口

    if err != nil {

        log.Fatal("ListenAndServe: ", err)

    }

}

在浏览器中输入了 [http://localhost:9090](http://localhost:9090/)

结果： 浏览器页面中输入出 Hello World!

测试在浏览器地址后面加一些参数试试：[http://localhost:9090?url\_long=111&url\_long=222](http://localhost:9090/?url_long=111&url_long=222)

服务器端使用ServeMux分发任务

ServeMux大致作用是，有一张map表，map里的key记录的是r.URL.String()，而value记录的是一个方法，这个方法和ServeHTTP是一样的，这样ServeMux是实现Handler接口的。这个方法有一个别名，叫HandlerFunc。

func main() {

    mux := http.NewServeMux()

    mux.HandleFunc("/h", func(w http.ResponseWriter, r \*http.Request) {

        io.WriteString(w, "hello")

    })

    mux.HandleFunc("/bye", func(w http.ResponseWriter, r \*http.Request) {

        io.WriteString(w, "byebye")

    })

    mux.HandleFunc("/baidu", func(w http.ResponseWriter, r \*http.Request) {

        http.Redirect(w, r, "http://www.baidu.com", http.StatusTemporaryRedirect)

    })

    mux.HandleFunc("/hello", sayhello)

    http.ListenAndServe(":1234", mux)

}

func sayhello(w http.ResponseWriter, r \*http.Request) {

    io.WriteString(w, "hello world")

}

当http.ListenAndServe(":1234", nil)的第2个参数是nil时，http内部会自己建立一个叫DefaultServeMux的ServeMux，因为这个ServeMux是http自己维护的，如果要向这个ServeMux注册的话，就要用http.HandleFunc这个方法。

func FileServer

func FileServer(root FileSystem) Handler

FileServer返回一个使用FileSystem接口root提供文件访问服务的HTTP处理器。要使用操作系统的FileSystem接口实现，可使用http.Dir：

    log.Fatal(http.ListenAndServe(":8080", http.FileServer(http.Dir("/usr/share/doc"))))

服务端如何获取客户端请求数据

1. 通过GET参数获取

比较常见的是如下方式获取：

r.ParseForm()

if len(r.Form["id"]) > 0 {

fmt.Fprintln(w, r.Form["id"][0])

}

其中r表示\*http.Request类型，w表示http.ResponseWriter类型。

r.Form是url.Values字典类型，r.Form["id"]取到的是一个数组类型。因为http.request在解析参数的时候会将同名的参数都放进同一个数组里，所以这里要用[0]获取到第一个。

2. 通过POST参数获取

这里要分两种情况：

普通的post表单请求，Content-Type=application/x-www-form-urlencoded

有文件上传的表单，Content-Type=multipart/form-data

第一种情况比较简单，直接用PostFormValue就可以取到了。

fmt.Fprintln(w, r.PostFormValue("id"))

3. 通过COOKIE参数获取

cookie, err := r.Cookie("id")

if err == nil {

fmt.Fprintln(w, "Domain:", cookie.Domain)

fmt.Fprintln(w, "Expires:", cookie.Expires)

fmt.Fprintln(w, "Name:", cookie.Name)

fmt.Fprintln(w, "Value:", cookie.Value)

}

r.Cookie返回\*http.Cookie类型，可以获取到domain、过期时间、值等数据。

application/x-www-form-urlencoded与 text/html 的区别

若需使用r.PostForm、r.Form等方法，必须先调用r.ParseForm()

客户端使用post提交数据到body时，若服务端需使用r.PostForm、r.Form等方法，客户端的请求体必须使用"Content-Type":"application/x-www-form-urlencoded") 类型，并且r.ParseForm()须放在ioutil.ReadAll(r.Body)解析body之前。否则数据将在body中。

http包的运行机制

首先了解服务端的几个概念：

Request：用户请求的信息，用来解析用户的请求信息，包括post，get，Cookie，url等信息。

Response:服务器需要反馈给客户端的信息。

Conn：用户的每次请求链接。

Handle:处理请求和生成返回信息的处理逻辑。

Go实现web服务的流程

第一步：创建Listen Socket，监听指定的端口，等待客户端请求到来。

第二步：Listen Socket接受客户端的请求，得到Client Socket，接下来通过Client Socket与客户端通信。

第三步：处理客户端请求，首先从Client Socket读取HTTP请求的协议头，如果是POST方法，还可能要读取客户端提交的数据，然后交给相应的handler处理请求，handler处理完，将数据通过Client Socket返回给客户端。

## 案例

案例1. 使用Get方式实现登录功能

要求：服务端根据客户端的请求判断是否能成功登陆

其中：用户名 username长度为4-10，密码 password长度为6-16

若满足条件返回"登录成功"，否则返回"登录失败"。

服务器端代码：

package main

import (

    "net/http"

    "fmt"

)

var userDatas = make(map[string]string)

/\*

给客户端响应数据之后，最好return，可以保证不会重复发送数据到客户端

\*/

func login(responseWriter http.ResponseWriter, request \*http.Request) {

    //解析提交的form表单数据（若要使用FormValue、form等方法，必须要调用ParseForm方法）

    request.ParseForm()

    fmt.Println(request.Header)

    //判断客户端请求的方式（包括GET、POST、HEAD、PUT、DELETE）

    if request.Method == "GET" {

        //获取客户端提交的username这个字段的值

        username := request.FormValue("username")

        password := request.FormValue("password")

        fmt.Println(username, password)

        //服务端的业务处理

        if len(username) < 4 || len(username) > 10{

            responseWriter.Write([]byte("用户名不符合要求"))

            return

        }

        if len(password) < 6 || len(password) > 16 {

            responseWriter.Write([]byte("密码不符合要求"))

            return

        }

        //判断客户端发送的username存不存在

        //passw是存在服务端系统里面的数据

        //password是客户端登录时发送过来的数据

        passw := userDatas[username]

        if len(passw) <= 0 {

            responseWriter.Write([]byte("用户名不存在"))

            return

        }

        if passw != password {

            responseWriter.Write([]byte("密码不对"))

            return

        }

        responseWriter.Write([]byte("登录成功"))

        return

    } else {

        responseWriter.Write([]byte("请求方式不对"))

        return

    }

    responseWriter.Write([]byte("登录失败"))

}

func register(responseWriter http.ResponseWriter, request \*http.Request) {

    request.ParseForm()

    if request.Method == "POST" {

        //获取客户端提交的username这个字段的值

        username := request.FormValue("username")

        password := request.FormValue("password")

        fmt.Println(username, password)

        //服务端的业务处理

        if len(username) < 4 || len(username) > 10{

            responseWriter.Write([]byte("用户名不符合要求"))

            return

        }

        if len(password) < 6 || len(password) > 16 {

            responseWriter.Write([]byte("密码不符合要求"))

            return

        }

        //判断系统里面有么有该用户名

        if len(userDatas[username]) > 0{

            responseWriter.Write([]byte("用户名已经存在"))

            return

        }

        responseWriter.Write([]byte("注册成功"))

        userDatas[username] = password

        return

    } else {

        responseWriter.Write([]byte("请求方式不对"))

        return

    }

    responseWriter.Write([]byte("注册失败"))

}

func other(w http.ResponseWriter, r \*http.Request){

    w.Write([]byte("找不到方法"))

}

func main() {

    mux := http.NewServeMux()

    mux.HandleFunc("/login", login)

    mux.HandleFunc("/register", register)

    mux.HandleFunc("/", other)

    http.ListenAndServe(":2000", mux)

}

客户端代码：

package main

import (

    "net/http"

    "bytes"

    "net/url"

    "goland/ketang/0604socket/help"

    "fmt"

    "io/ioutil"

)

func main() {

    //构建参数

    param := url.Values{

        "username":{"xiaoming"},

        "password":{"123456"},

    }

    //参数转化成body

    buffer := bytes.NewBufferString(param.Encode())

    //发起post请求

    response, err := http.Post("http://192.168.149.138:2000/register",

        "application/x-www-form-urlencoded", buffer)

    help.CheckErr(err)

    //获取到服务端的响应数据

    if response.StatusCode == 200 {

        //操作响应数据

        defer response.Body.Close()

        //读取body里面的数据

        responseBody,err := ioutil.ReadAll(response.Body)

        if err != nil {

            fmt.Println(err)

        }

        //打印响应体里面的数据

        fmt.Println(string(responseBody))

    } else {

        fmt.Println("请求失败", response.Status)

    }

}

案例2. 实现用户注册和登录功能

要求：

1）实现注册功能

服务端根据客户端的请求判断是否能成功注册，注册成功要求如下：

用户名 username长度为4-10

密码 password长度为6-16

若满足条件返回"注册成功"，否则返回"注册失败"

2） 若请求其它url则返回"找不到方法"

3） 服务端将注册后的数据保存在全局变量中，登录时需要判断usernam是否存在，并且判断密码是否正确。

服务器端代码：

package main

import (

"net/http"

"fmt"

)

var userDatas = make(map[string]string)

/\*

给客户端响应数据之后，最好return，可以保证不会重复发送数据到客户端

\*/

func login(responseWriter http.ResponseWriter, request \*http.Request) {

//解析提交的form表单数据（若要使用FormValue、form等方法，必须要调用ParseForm方法）

request.ParseForm()

fmt.Println(request.Header)

//判断客户端请求的方式

if request.Method == "GET" {

//获取客户端提交的username这个字段的值

username := request.FormValue("username")

password := request.FormValue("password")

fmt.Println(username, password)

//服务端的业务处理

if len(username) < 4 || len(username) > 10{

responseWriter.Write([]byte("用户名不符合要求"))

return

}

if len(password) < 6 || len(password) > 16 {

responseWriter.Write([]byte("密码不符合要求"))

return

}

//判断客户端发送的username存不存在

//passw是存在服务端系统里面的数据

//password是客户端登录时发送过来的数据

passw := userDatas[username]

if len(passw) <= 0 {

responseWriter.Write([]byte("用户名不存在"))

return

}

if passw != password {

responseWriter.Write([]byte("密码不对"))

return

}

responseWriter.Write([]byte("登录成功"))

return

} else {

responseWriter.Write([]byte("请求方式不对"))

return

}

responseWriter.Write([]byte("登录失败"))

}

func register(responseWriter http.ResponseWriter, request \*http.Request) {

request.ParseForm()

if request.Method == "POST" {

//获取客户端提交的username这个字段的值

username := request.FormValue("username")

password := request.FormValue("password")

fmt.Println(username, password)

//服务端的业务处理

if len(username) < 4 || len(username) > 10{

responseWriter.Write([]byte("用户名不符合要求"))

return

}

if len(password) < 6 || len(password) > 16 {

responseWriter.Write([]byte("密码不符合要求"))

return

}

//判断系统里面有么有该用户名

if len(userDatas[username]) > 0{

responseWriter.Write([]byte("用户名已经存在"))

return

}

responseWriter.Write([]byte("注册成功"))

userDatas[username] = password

return

} else {

responseWriter.Write([]byte("请求方式不对"))

return

}

responseWriter.Write([]byte("注册失败"))

}

func other(w http.ResponseWriter, r \*http.Request){

w.Write([]byte("找不到方法"))

}

func main() {

mux := http.NewServeMux()

mux.HandleFunc("/login", login)

mux.HandleFunc("/register", register)

mux.HandleFunc("/", other)

http.ListenAndServe(":1234", mux)

}

客户端代码：

package main

import (

"net/http"

"bytes"

"net/url"

"goland/ketang/0604socket/help"

"fmt"

"io/ioutil"

)

func main() {

//构建参数

param := url.Values{

"username":{"admin"},

"password":{"123456"},

}

//参数转化成body

buffer := bytes.NewBufferString(param.Encode())

//发起post请求

response, err := http.Post("http://localhost：1234/register",

"application/x-www-form-urlencoded", buffer)

help.CheckErr(err)

//获取到服务端的响应数据

if response.StatusCode == 200 {

//操作响应数据

defer response.Body.Close()

//读取body里面的数据

responseBody,err := ioutil.ReadAll(response.Body)

if err != nil {

fmt.Println(err)

}

//打印响应体里面的数据

fmt.Println(string(responseBody))

} else {

fmt.Println("请求失败", response.Status)

}

}

​

# JSON处理

本章重点为大家介绍如下的内容：

什么是JSON

JSON语法规则

JSON的优点和缺点

对象转换为JSON

JSON转换为对象

JSON转换为map

JSON流解析

JSON是一种轻量级的数据交换格式(JavaScript Object Notation)，因其简单、可读性强广泛使用。

JSON语法规则

对象是一个无序的“‘名称/值’对”集合。一个对象以“{”（左括号）开始，“}”（右括号）结束。每个“名称”后跟一个“:”（冒号）；“‘名称/值’ 对”之间使用“,”（逗号）分隔。

数组是值（value）的有序集合。一个数组以“[”（左中括号）开始，“]”（右中括号）结束。值之间使用“,”（逗号）分隔。

值（value）可以是双引号括起来的字符串（string）、数值(number)、true、false、 null、对象（object）或者数组（array）。这些结构可以嵌套。

字符串（string）是由双引号包围的任意数量Unicode字符的集合，使用反斜线转义。一个字符（character）即一个单独的字符串（character string）。

简单的json：

{name:"张三",age:23},

复杂的json：

{

"employee" :

{

"firstName": "张",

"lastName" : "三",

"employeeNumber" : 101,

"age":23

}

}

JSON的优点:

数据格式比较简单, 易于读写, 格式都是压缩的, 占用带宽小

易于解析这种语言, 客户端JavaScript可以简单的通过eval\_r()进行JSON数据的读取

支持多种语言, 包括ActionScript, C, C#, ColdFusion, Java,JavaScript, Perl, PHP, Python, Ruby等语言服务器端语言, 便于服务器端的解析

因为JSON格式能够直接为服务器端代码使用, 大大简化了服务器端和客户端的代码开发量, 但是完成的任务不变, 且易于维护

JSON的缺点:

没有XML格式这么推广的深入人心和使用广泛, 没有XML那么通用性

JSON格式目前在Web Service中推广还属于初级阶段

## 对象转换为JSON

Go的标准包encoding/json对JSON的编解码提供了完整的支持。

将Go数据类型转换为JSON。用到的函数：

func Marshal(v interface{}) ([]byte, error)

该函数递归遍历v的结构，生成对应的JSON。

字典转JSON

package main

import (

"encoding/json"

"fmt"

)

func main() {

m := map[string][]string{

"level": {"debug"},

"message": {"File not found", "Stack overflow"},

}

if data, err := json.Marshal(m); err == nil {

fmt.Printf("%s\n", data)

}

}

输出结果：

{"level":["debug"],"message":["File not found","Stack overflow"]}

大家可以看到Marshal函数返回的JSON字符串是没有空白字符和缩进的，这种紧凑的表示形式是最常用的传输形式，但是不好阅读。如果需要为前端生成便于阅读的格式，可以调json.MarshaIndent，该函数有两个参数表示每一行的前缀和缩进方式：

if data, err := json.MarshalIndent(m, "", " "); err == nil {

fmt.Printf("%s\n", data)

}

输出结果：

{

"level": [

"debug"

],

"message": [

"File not found",

"Stack overflow"

]

}

Go数据类型转换为JSON规则

在编码过程中，json包会将Go的类型转换为JSON类型，转换规则如下：

bool -> JSON boolean

浮点数, 整数, Number -> JSON number

string -> JSON string

数组、切片 -> JSON数组

[]byte -> base64 string

struct、map -> JSON object

结构体转JSON

经常会使用结构体来转换成JSON。json包是通过反射机制来实现编解码的，因此结构体必须导出所转换的字段，不导出的字段不会被json包解析：

package main

import (

"encoding/json"

"fmt"

)

type DebugInfo struct {

Level string

Msg string

author string // 未导出字段不会被json解析(首字母小写)

}

func main() {

dbgInfs := []DebugInfo{

DebugInfo{"debug", `File: "test.txt" Not Found`, "Cynhard"},

DebugInfo{"", "Logic error", "Gopher"},

}

if data, err := json.Marshal(dbgInfs); err == nil {

fmt.Printf("%s\n", data)

}

}

输出结果如下：

[{"Level":"debug","Msg":"File: \"test.txt\" Not Found"},{"Level":"","Msg":"Logic error"}]

结构体字段标签：

json包在解析结构体时，如果遇到key为“json”的字段标签，则会按照一定规则解析该标签：第一个出现的是字段在JSON串中使用的名字，之后为其他选项，例如“omitempty”指定空值字段不出现在JSON中。如果整个value为'"-"'，则不解析该字段。例如将上例中的结构体改为如下：

type DebugInfo struct {

Level string 'json:"level,omitempty"' // Level解析为level,忽略空值

Msg string 'json:"message"' // Msg解析为message

Author string ‘'json:"-"' // 忽略Author

}

则输出为：

[{"level":"debug","message":"File: \"test.txt\" Not Found"},{"message":"Logic error"}]

结构体中匿名字段：

json包在解析匿名字段时，会将匿名字段的字段当成该结构体的字段处理：

package main

import (

"encoding/json"

"fmt"

)

type Point struct{ X, Y int }

type Circle struct {

Point

Radius int

}

func main() {

if data, err := json.Marshal(Circle{Point{50, 50}, 25}); err == nil {

fmt.Printf("%s\n", data)

}

}

输出结果：

{"X":50,"Y":50,"Radius":25}

注意：

Marshal函数只有在转换成功的时候才会返回数据，在转换的过程中我们需要注意几点：

JSON对象只支持string作为key，所以要编码一个map，那么必须是map[string]T这种类型(T是Go语言中任意的类型)

Channel, complex和function是不能被编码成JSON的

指针在编码的时候会输出指针指向的内容，而空指针会输出null

## JSON转换为对象

将JSON转换为Go数据类型。用到的函数：

func Unmarshal(data []byte, v interface{}) error

JSON转数组

package main

import (

"encoding/json"

"fmt"

)

func main() {

data := '[{"Level":"debug","Msg":"File: \"test.txt\" Not Found"},{"Level":"","Msg":"Logic error"}]'

var dbgInfos []map[string]string

json.Unmarshal([]byte(data), &dbgInfos)

fmt.Println(dbgInfos)

}

输出为：

[map[Level:debug Msg:File: "test.txt" Not Found] map[Level: Msg:Logic error]]

在解码过程中，json包会将JSON类型转换为Go类型，转换规则如下：

JSON boolean -> bool

JSON number -> float64

JSON string -> string

JSON数组 -> []interface{}

JSON object -> map

null -> nil

JSON转结构体

JSON可以转换成结构体。同编码一样，json包是通过反射机制来实现解码的，因此结构体必须导出所转换的字段，不导出的字段不会被json包解析，另外解析时不区分大小写：

package main

import (

"encoding/json"

"fmt"

)

type DebugInfo struct {

Level string

Msg string

author string // 未导出字段不会被json解析

}

func (dbgInfo DebugInfo) String() string {

return fmt.Sprintf("{Level: %s, Msg: %s}", dbgInfo.Level, dbgInfo.Msg)

}

func main() {

data := '[{"level":"debug","msg":"File Not Found","author":"Cynhard"},{"level":"","msg":"Logic error","author":"Gopher"}]'

var dbgInfos []DebugInfo

json.Unmarshal([]byte(data), &dbgInfos)

fmt.Println(dbgInfos)

}

输出结果如下：

[{Level: debug, Msg: File Not Found} {Level: , Msg: Logic error}]

结构体字段标签

解码时依然支持结构体字段标签，规则和编码时一样：

package main

import (

"encoding/json"

"fmt"

)

type DebugInfo struct {

Level string 'json:"level"' // level 解码为 Level

Msg string 'json:"message"' // message 解码为 Msg

Author string 'json:"-"' // 忽略Author

}

func (dbgInfo DebugInfo) String() string {

return fmt.Sprintf("{Level: %s, Msg: %s}", dbgInfo.Level, dbgInfo.Msg)

}

func main() {

data := '[{"level":"debug","message":"File Not Found","author":"Cynhard"},{"level":"","message":"Logic error","author":"Gopher"}'

var dbgInfos []DebugInfo

json.Unmarshal([]byte(data), &dbgInfos)

fmt.Println(dbgInfos)

}

则结果为：

[{Level: debug, Msg: File Not Found} {Level: , Msg: Logic error}]

匿名字段

编码时，和解码类似，在解码JSON时，如果找不到字段，则查找字段的字段：

package main

import (

"encoding/json"

"fmt"

)

type Point struct{ X, Y int }

type Circle struct {

Point

Radius int

}

func main() {

data := '{"X":80,"Y":80,"Radius":40}'

var c Circle

json.Unmarshal([]byte(data), &c)

fmt.Println(c)

}

输出结果：

{{80 80} 40}

## JSON转换为map

// 当前程序的包名

package main

// 导入其它的包

import (

"encoding/json"

"fmt"

)

func main() {

map2json2map()

}

func map2json2map() {

map1 := make(map[string]interface{})

map1["1"] = "hello"

map1["2"] = "world"

//return []byte

str, err := json.Marshal(map1)

if err != nil {

fmt.Println(err)

}

fmt.Println("map to json", string(str))

//json([]byte) to map

map2 := make(map[string]interface{})

err = json.Unmarshal(str, &map2)

if err != nil {

fmt.Println(err)

}

fmt.Println("json to map ", map2)

fmt.Println("The value of key1 is", map2["1"])

}

运行结果：

   map to json {"1":"hello","2":"world"}

    json to map  map[1:hello 2:world]

    The value of key1 is hello

## JSON流解析

json 包提供了Decoder和Encoder用来支持JSON 数据流的读写。函数NewDecoder 和 NewEncoder封装了io.Reader和io.Writer 接口类型。

func NewDecoder(r io.Reader) \*Decoder

func NewEncoder(w io.Writer) \*Encoder

由于读写操作的普遍性，类型Encoder和Decoder可以用于多种场合，例如读写HTTP 链接，WebSockets或者文件。

func main() {

data := `{"name":"go语言学习", "Count":34, "student":["张三", "里斯"], "address":{"floor":2, "classroom":"201"}}`

//流式编码

dec := json.NewDecoder(bytes.NewBufferString(data))

var v map[string]interface{}

if err := dec.Decode(&v); err != nil {

log.Println(err)

return

}

for k := range v {

if k == "name" {

delete(v, k)

}

}

fmt.Printf("%T %v\n", v, v)

//流式解码

buffer := bytes.Buffer{}

enc := json.NewEncoder(&buffer)

if err := enc.Encode(&v); err != nil {

log.Println(err)

}

fmt.Printf("%v", buffer.String())

}

# Web开发

本章重点为大家介绍如下的内容：

网页模板

模板的使用

Web交互

## 网页模版

Go的两个包text/template和html/template提供了将变量的值替换到文本的机制。通常用在一些printf应付不了的、需要复杂的格式化的场合。

模板的使用

模板规则

Action：模板是一个包含了一个或多个双花括号标记（{{...}}）的字符串，这种双花括标记号叫做Action。模板中所有的除Action的字符串都按照原样打印，Action则表现了特殊的行为。通过Action可以打印变量的值、字段的值、调用函数或方法、表达控制流程等。

Dot（.）：Action中的点（.）表示当前值。初始值为传递给模板的参数。

API：text/template与html/template提供了相同的API来创建、解析和执行模板，区别是html/template提供了HTML转义。

// 创建一个名字为name的模板

func New(name string) \*Template

// 解析模板字符串

func (t \*Template) Parse(text string) (\*Template, error)

// 解析文件

func ParseFiles(filenames ...string) (\*Template, error)

// 执行模板，将结果写入wr

func (t \*Template) Execute(wr io.Writer, data interface{}) error

在确定模板可以正确解析的场合，可以使用Must()来处理Parse()的结果。如果Parse()失败，则调用panic()，如果成功，返回Template对象。

func Must(t \*Template, err error) \*Template

Funcs()用来注册函数给模板，注册之后模板就可以通过名字调用外部函数了（见下面例子）。

func (t \*Template) Funcs(funcMap FuncMap) \*Template type FuncMap map[string]interface{}

最基本的使用

package main

import (

"log"

"os"

"text/template" // 引入包，如果处理HTML，则为 "html/template"

)

func main() {

// 创建一个名字为report的模板，并解析一个字符串

t, err := template.New("report").Parse("I am {{.}} years old.")

if err != nil {

log.Fatal(err)

}

// 执行模板，5作为参数传递，将结果写到标准输出

t.Execute(os.Stdout, 5)

// 打印结果：I am 5 years old.

}

访问复合数据结构的值

访问结构体字段需要在点（.）后面加字段名。

访问Map值需要在点（.）后面加Key的值。

package main

import (

"os"

"text/template"

)

func main() {

t := template.Must(

template.New("report").Parse("I am {{.Age}} years old.\n"))

t.Execute(os.Stdout, struct{ Age int }{5})

t.Execute(os.Stdout, map[string]int{"Age": 5})

}

需要注意的是template访问结构体字段时利用了[反射](http://blog.csdn.net/u011304970/article/details/69908641" \l "reflect)机制，因此必须导出字段才能在模板里访问。

循环

使用{{range}}和{{end}}来表示循环。每次迭代中点（.）代表当前迭代的值。

package main

import (

"os"

"text/template"

)

func main() {

const temStr = "{{range .}}{{.}}\n{{end}}"

t := template.Must(template.New("report").Parse(temStr))

t.Execute(os.Stdout, []string{"Hello", "Gopher"})

}

管道

用|将其左边的输出作为其右边的输入。

package main

import (

"os"

"text/template"

)

func main() {

const temStr = ‘{{. | printf "% 10s"}}’

t := template.Must(template.New("report").Parse(temStr))

t.Execute(os.Stdout, "Gopher")

}

调用函数

如果在模板中需要调用函数，则在Parse()之前需要通过Funcs()来注册。

package main

import (

"os"

"text/template"

)

func greet(name string) string {

return "Hello, " + name + "!"

}

func main() {

const temStr = ‘{{. | greet}}’

t := template.Must(template.New("report")

Funcs(template.FuncMap{"greet": greet})

Parse(temStr))

t.Execute(os.Stdout, "Gopher")

}

生成动态HTML

HTML文件：time.html

<html>

<body>

<p>Current time is: {{.}}</p>

</body>

</html>

go文件：

package main

import (

"html/template"

"log"

"net/http"

"time"

)

func main() {

http.HandleFunc("/time", timeHandler)

log.Fatal(http.ListenAndServe(":9090", nil))

}

func timeHandler(w http.ResponseWriter, r \*http.Request) {

t := template.Must(template.ParseFiles("time.html"))

t.Execute(w, time.Now().Format("2019-01-02 15:04:05"))

}

请看下面的示例：

package main

import (

"os"

"html/template"

"log"

)

const templ = ‘<!DOCTYPE html>

<html>

<meta charset="utf-8"/>

<head>

<title>主页</title>

</head>

<body>

<form>

<table align="center">

{{range .}}

<tr>

<td>Name：</td>

<td>{{.Name}}</td>

</tr>

<tr>

<td>Age:</td>

<td>{{.Age}}</td>

</tr>

<tr>

<td>Sex:</td>

<td>{{.Sex}}</td>

</tr>

<tr>

<td>Habbits:</td>

{{range $\_, $habbit := .Habbits}}

<td>{{$habbit}}</td>

{{end}}

</tr>

<tr>

<td>{{.Normal | addString}}</td>

</tr>

{{end}}

</table>

</form>

<body>

</html>’

type Student struct {

Name string

Age int

Sex string

Habbits []string

Normal string

}

func addString(normal string) string {

return normal + "模式"

}

var students = []Student{

{Name:"Marco", Age:19, Sex:"男", Habbits:[]string{"sing", "write","basketball"},Normal:"正常"},

{Name:"Mike", Age:20, Sex:"男", Habbits:[]string{"basketball", "paino"},Normal:"正常"},

{Name:"Marry", Age:17, Sex:"女", Habbits:[]string{"art", "eat"},Normal:"正常 "},

}

func main() {

report, err := template.New("report").Funcs(template.FuncMap{"addString": addString}).Parse(templ)

if err != nil {

log.Fatal(err)

}

if err := report.Execute(os.Stdout,students); err != nil {

log.Fatal(err)

}

}

## Web交互示例

利用Go语言构建Web应用程序，实质上是构建HTTP服务器。HTTP是一个简单的请求-响应协议，通常运行在TCP之上。它指定了客户端可能发送给服务器什么样的消息以及得到什么样的响应。下图为最简化的HTTP协议处理流程。

HTTP请求和响应流程

构建在服务器端运行的Web程序的基本要素包括：

如何分析和表示HTTP请求；

如何根据HTTP请求以及程序逻辑生成HTTP响应（包括生成HTML网页）；

如何使服务器端一直正确地运行以接受请求并生成响应。

Go语言有关Web程序的构建主要涉及net/http包，因此这里所给的各种函数、类型、变量等标识符，除了特别说明外，都是属于net/http包内的。

请求和响应信息的表示在 HTTP 1.1中，请求和响应信息都是由以下四个部分组成，两者之间格式的区别是开始行不同。

开始行。位于第一行。在请求信息中叫请求行，在响应信息中叫状态行。

请求行：构成为请求方法 URI 协议/版本，例如GET /images/logo.gif HTTP/1.1；

响应行：构成为协议版本 状态代码 状态描述，例如HTTP/1.1 200 OK。

头。零行或多行。包含一些额外的信息，用来说明浏览器、服务器以及后续正文的一些信息。

空行。

正文。包含客户端提交或服务器返回的一些信息。请求信息和响应信息中都可以没有此部分。

开始行和头的各行必须以<CR><LF>作为结尾。空行内必须只有<CR><LF>而无其他空格。在HTTP/1.1协议中，开始行和头都是以ASCII编码的纯文本，所有的请求头，除Host外，都是可选的。

HTTP请求信息由客户端发来，Web程序要做的首先就是分析这些请求信息，并用Go语言中响应的数据对象来表示。在net/http包中，用Request结构体表示HTTP请求信息。

当收到并理解（将请求信息解析为Request类型变量）了请求信息之后，就需要根据相应的处理逻辑，构建响应信息。net/http包中，用Response结构体表示响应信息。

如何构建响应信息

客户端发来的请求信息是符合HTTP协议的，因此net/http包已经能够根据请求信息，自动帮我们创建Request结构体对象了。

对于每个服务器程序，其行为是不同的，也即需要根据请求构建各样的响应信息，因此我们只能自己构建这个Response了。不过在这个过程中，net/http包还是竭尽所能地为我们提供帮助，从而帮我们隐去了许多复杂的信息。甚至如果不仔细想，我们都没有意识到我们是在构建Response结构体对象。

为了能更好地帮助我们，net/http包首先为我们规定了一个构建Response的标准过程。该过程就是要求我们实现一个Handler接口：

type Handler interface { ServeHTTP(ResponseWriter, \*Request) }

现在，我们编写Web程序的主要工作就是编写各种实现该Handler接口的类型，并在该类型的ServeHTTP方法中编写服务器响应逻辑。这样一来，我们编写的Web服务器程序可能主要就是由各种各样的fooHandler、barHandler构成；Handler接口就成为net/http包中最重要的东西。可以说，每个Handler接口的实现就是一个小的Web服务器。以往由许多人将“handler”翻译为“句柄”，这里将其翻译为处理程序，或不做翻译。

如何实现Handler接口

方法1：

显式地编写一个实现Handler接口的类型

package main

import (

"fmt"

"io/ioutil"

"net/http"

)

type Page struct {

Title string

Body []byte

}

func loadPage(title string) (\*Page, error) {

filename := title + ".txt"

body, err := ioutil.ReadFile(filename)

if err != nil {

return nil, err

}

return &Page{Title: title, Body: body}, nil }

type viewHandler struct{}

func (viewHandler) ServeHTTP(w http.ResponseWriter, r \*http.Request) {

title := r.URL.Path[len("/view/"):]

p, \_ := loadPage(title)

fmt.Fprintf(w, "<h1>%s</h1><div>%s</div>", p.Title, p.Body)

}

func main() {

http.Handle("/view/", viewHandler{})

http.ListenAndServe(":8080", nil)

}

假设该程序的当前目录中有一个abc.txt的文本文件，若访问http://localhost:8080/view/abc，则会显示该文件的内容。

在该程序main函数的第一行使用了Handle函数，其定义为：

func Handle(pattern string, handler Handler)

该函数的功能就是将我们编写的Handler接口的实现viewHandler传递给net/http包，并由net/http包来调用viewHandler的ServeHTTP方法。

不过有一点还是要注意，该viewHandler只对URL的以/view/开头的路径才起作用，如果我们访问http://localhost:8080/或http://localhost:8080/edit，则都会返回一个404 page not found页面；而如果访问http://localhost:8080/view/xyz，则浏览器什么数据也得不到。对于后一种情况，很显然是因为我们编写的viewHandler.ServeHTTP方法没有对Wiki页面文件不存在时loadPage函数返回的错误进行处理造成的；而对前一种情况，则是net/http包帮我们完成的。

方法2：

将一个普通函数转换为请求处理函数

我们可能已经注意到了，方法1中程序的viewHandler结构体中没有一个字段，我们构建它主要是为了使用其ServeHTTP方法。很显然，这有点绕了。因为在大多数时候，我们只需要使Handler成为一个函数就足够了。为此，http 包中提供了一个替代Handle函数的HandleFunc函数：

func HandleFunc(pattern string, handler func(ResponseWriter, \*Request))

即HandleFunc函数不再像Handle那样接受一个Handler接口对象，而是接受一个具有特定签名的函数。而原来由Handler接口对象的ServeHTTP方法所实现的功能，现在需要该函数来实现。

package main

import (

"fmt"

"io/ioutil"

"net/http"

)

type Page struct {

Title string

Body []byte

}

func loadPage(title string) (\*Page, error) {

filename := title + ".txt"

body, err := ioutil.ReadFile(filename)

if err != nil {

return nil, err

}

return &Page{Title: title, Body: body}, nil }

func viewHandler(w http.ResponseWriter, r \*http.Request) {

title := r.URL.Path[len("/view/"):]

p, \_ := loadPage(title)

fmt.Fprintf(w, "<h1>%s</h1><div>%s</div>", p.Title, p.Body)

}

func main() {

http.HandleFunc("/view/", viewHandler)

http.ListenAndServe(":8080", nil)

}

可以看出，该示例程序中的viewHandler函数实际上并没有实现Handler接口，因此它是一个伪Handler。不过其所实现的功能正是Handler接口对象需要实现的功能，我们可称像viewHandler这样的函数为Handler函数。我们会在方法3中通过类型转换轻易地将这种Handler函数转换为一个真正的Handler。

多数情况下，使用HandleFunc比使用Handle更加简便，这也是我们所常用的方法。

方法3：

利用闭包功能编写一个返回Handler的请求处理函数。

在Go语言中，函数是一等公民，函数字面可以被赋值给一个变量或直接调用。同时函数字面（实际上就是一段代码块）也是一个闭包，它可以引用定义它的外围函数（即该代码块的作用域环境）中的变量，这些变量会在外围函数和该函数字面之间共享，并且在该函数字面可访问期间一直存在。

那么，我们可以定义一个这样的函数类型，该函数类型具有和我们在方法2中定义的viewHandler函数具有相同的签名，因而可以通过类型转换把viewHandler函数转换为此函数类型；同时该函数类型本身实现了Handler接口。net/http包中的HandlerFunc就是这样的函数类型。

首先，HandlerFunc是一个函数类型：

type HandlerFunc func(ResponseWriter, \*Request)

其次，HandlerFunc同时也实现了Handler接口：

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r \*Request)

这里ServeHTTP的实现很简单，即调用其自身f(w, r)。

任何签名为func(http.ResponseWriter, \*http.Request)函数都可以被转换为HandlerFunc。的事实上，方法2中的main函数中第一行的HandleFunc函数就是将viewHandler转换为HandlerFunc再针对其调用Handle的。即http.HandleFunc("/view/", viewHandler)相当于http.Handle("/view/", http.HandlerFunc(viewHandler{}))。

既然如此，能不能更直接地编写一个返回HandlerFunc函数的函数？借助于Go语言函数的灵活性，这一点是可以实现的。可对方法2中的viewHandler函数做如下改写：

func viewHandler()

http.HandlerFunc {

return http.HandlerFunc(func(w http.ResponseWriter, r \*http.Request) {

title := r.URL.Path[len("/view/"):]

p, \_ := loadPage(title)

fmt.Fprintf(w, "<h1>%s</h1><div>%s</div>", p.Title, p.Body)

})

}

由于viewHandler函数返回的HandlerFunc对象既实现了Handler接口，又具有和方法2中的Handler函数相同的签名。因此此例中main函数的第一行既可以使用http.Handle，又可以使用http.HandleFunc。另外，该viewHandler函数中的return可以不用http.HandlerFunc进行显式类型转换，而是自动地将返回的函数字面转换为HandlerFunc类型。

现在理解起来可能变得困难点了。为什么要这样做呢？对比方法2和方法3的viewHandler函数签名就可以看出来了：方法2中的viewHandler函数签名必须是固定的，而方法3则是任意的。这样我们可以利用方法3向viewHandler函数中传递任意的东西，如数据库连接、HTML模板、请求验证、日志和追踪等东西，这些变量在闭包函数中是可访问的。而被传递的变量可以是定义在main函数内的局部变量；要不然，在闭包函数中能访问的外界变量就只能是全局变量了。另外，利用闭包的性质，被闭包函数引用的外部自由变量将与闭包函数一同存在，即在同样的引用环境中调用闭包函数时，其所引用的自由变量仍保持上次运行后的值，这样就达到了共享状态的目的。让我们对本例中的代码进行修改：

func viewHandler(n int)

http.HandlerFunc {

return http.HandlerFunc(func(w http.ResponseWriter, r \*http.Request) {

title := r.URL.Path[len("/view/"):]

p, \_ := loadPage(title)

fmt.Fprintf(w, "<h1>%s</h1><div>%s</div>", p.Title, p.Body)

n++

fmt.Fprintf(w, "<div>%v</div>", n)

})

}

func main() {

var n

int http.HandleFunc("/view/", viewHandler(n))

http.HandleFunc("/page/", viewHandler(n))

http.ListenAndServe(":8080", nil)

}

现在，分别访问http://localhost:8080/view/abc和http://localhost:8080/page/abc两个地址，每次刷新页面，则显示的n值增加1，但两个地址页面内的n值得变化是相互独立的。

方法4：

用封装器函数封装多个Handler的实现，我们就可以编写一个具有如下签名的HandlerFunc封装器函数：

wrapperHandler(http.HandlerFunc) http.HandlerFunc

该封装器是这样一个函数，它具有一个输入参数和一个输出参数，两者都是HandlerFunc类型。该函数通常按如下方式进行定义：

func wrapperHandler(f http.HandlerFunc) http.HandlerFunc {

return http.HandlerFunc(func(w http.ResponseWriter, r \*http.Request) {

do\_something\_before\_calling\_f() f(w, r) do\_something\_after\_calling\_f()

})

}

与方法3一样，在封装器函数中，我们使用了Go语言闭包的功能构建了一个函数变量，并在返回时将该函数变量转换为HandlerFunc。与方法3不一样的地方在于，我们通过一个参数将被封装的Handler函数传递给封装器函数，并在封装器函数中定义的闭包函数中通过通过f(w, r)调用被封装的HandlerFunc的功能。而在执行f(w, r)之前或之后，我们可以额外地做一些事情，甚至可以根据情况决定是否执行f(w, r)。

这样一来，可以在方法2的示例程序的基础上，添加wrapperHandler函数，并修改main函数：

func wrapperHandler(f http.HandlerFunc)

http.HandlerFunc {

return http.HandlerFunc(func(w http.ResponseWriter, r \*http.Request) {

fmt.Fprintf(w, "<div>Do something <strong>before</strong> calling a handler.</div>")

f(w, r)

fmt.Fprintf(w, "<div>Do something <strong>after</strong> calling a handler.</div>")

})

}

func main() {

http.HandleFunc("/view/", wrapperHandler(viewHandler)) http.ListenAndServe(":8080", nil)

}

我们真是绕了一个大圈，但这样绕有其自身的好处：

共享代码：将多个Handler函数（如viewHandler、editHandler和saveHandler）中共同的代码放进此封装器函数中，并在封装器中实现一些公用的代码，具体请见Writing Web Applications一文的末尾部分。

共享状态：除了本例中向wrapperHandler函数传递各种Handler函数外，我们可以增加参数个数，即传递其他自由变量给闭包（例如：func wrapperHandler(f http.HandlerFunc, n int) http.HandlerFunc），从而达到与方法3相同的共享状态效果。注意，这里说的共享状态实际上只是在同一个闭包函数（也即Handler）及其运行环境中共享状态，在某一运行环境下传递到某个闭包型Handler的自由变量并不能自动再被传出去，这与以后将要讲得在多个Handler间共享状态是不同的。

需要补充说明一下。在net/http包中，Handle和HandleFunc，Handler和HandlerFunc，都是对同一问题的具体两种方法。当我们处理的东西较简单时，为求简便，一般会用带Func后缀的后一类方法，尤其是HandlerFunc给我们带来了很大的灵活性。当需要定义一个包含较多字段的Handler实现时，就会像方法1那样正正经经地定义一个Handler类型。因此，不管是方法3和方法4，你都可以看到不同的写法，如使方法4封装的是Handler结构体变量而非这里的HandlerFunc，但其原理都是相通的。

ResponseWriter接口

尽管知道了Handler的多种写法，但我们还没有完全弄明白如何构建Response。net/http包将构建Response的过程也标准化了，即通过各种Handler操作ResponseWriter接口来构建Response。

type ResponseWriter interface {

Header()

Header Write([]byte) (int, error)

WriteHeader(int)

}

ResponseWriter实现了io.Writer接口，因此，该接口可被用于各种打印函数，如fmt.Fprintf。WriteHeader方法用于向HTTP响应信息写入状态码（一般是错误代码），它必须先于Write调用。若不调用WriteHeader，使用Write方法会自动写入状态码http.StatusOK。Header方法返回一个Header结构体对象，可以通过该结构体的方法对HTTP响应消息的头进行操作。但这种操作必须在WriteHeader和Write执行之前进行，除非所操作的Header字段在执行WriteHeader或Write之前已经被标记为"Trailer"。有点复杂，这里就不再多讲了。其实对于大部分人只要调用WriteHeader和Write就够了。

示例：

用go语言做后台一个简易的聊天。

go代码：

// WebChat project main.go

package main

import (

"fmt"

"net/http"

"time"

"encoding/json"

"strings"

"golang.org/x/net/websocket"

)

//全局信息

var datas Datas

var users map[\*websocket.Conn]string

func main() {

fmt.Println("启动时间")

fmt.Println(time.Now())

//初始化

datas = Datas{}

users = make(map[\*websocket.Conn]string)

//绑定效果页面

http.HandleFunc("/", h\_index)

//绑定socket方法

http.Handle("/webSocket", websocket.Handler(h\_webSocket))

//开始监听

http.ListenAndServe(":8", nil)

}

func h\_index(w http.ResponseWriter, r \*http.Request) {

http.ServeFile(w, r, "index.html")

}

func h\_webSocket(ws \*websocket.Conn) {

var userMsg UserMsg

var data string

for {

//判断是否重复连接

if \_, ok := users[ws]; !ok {

users[ws] = "匿名"

}

userMsgsLen := len(datas.UserMsgs)

fmt.Println("UserMsgs", userMsgsLen, "users长度：", len(users))

//有消息时，全部分发送数据

if userMsgsLen > 0 {

b, errMarshl := json.Marshal(datas)

if errMarshl != nil {

fmt.Println("全局消息内容异常...")

break

}

for key, \_ := range users {

errMarshl = websocket.Message.Send(key, string(b))

if errMarshl != nil {

//移除出错的链接

delete(users, key)

fmt.Println("发送出错...")

break

}

}

datas.UserMsgs = make([]UserMsg, 0)

}

fmt.Println("开始解析数据...")

err := websocket.Message.Receive(ws, &data)

fmt.Println("data：", data)

if err != nil {

//移除出错的链接

delete(users, ws)

fmt.Println("接收出错...")

break

}

data = strings.Replace(data, "\n", "", 0)

err = json.Unmarshal([]byte(data), &userMsg)

if err != nil {

fmt.Println("解析数据异常...")

break

}

fmt.Println("请求数据类型：", userMsg.DataType)

switch userMsg.DataType {

case "send":

//赋值对应的昵称到ws

if \_, ok := users[ws]; ok {

users[ws] = userMsg.UserName

//清除连接人昵称信息

datas.UserDatas = make([]UserData, 0)

//重新加载当前在线连接人

for \_, item := range users {

userData := UserData{UserName: item}

datas.UserDatas = append(datas.UserDatas, userData)

}

}

datas.UserMsgs = append(datas.UserMsgs, userMsg)

}

}

}

type UserMsg struct {

UserName string

Msg string

DataType string

}

type UserData struct {

UserName string

}

type Datas struct {

UserMsgs []UserMsg

UserDatas []UserData

}

HTML代码：

<!DOCTYPE html>

<html lang="zh-CN">

<head>

<title></title>

<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, user-scalable=no">

<!-- 新 Bootstrap 核心 CSS 文件 -->

<link rel="stylesheet" href="//cdn.bootcss.com/bootstrap/3.3.5/css/bootstrap.min.css">

<script src="//cdn.bootcss.com/jquery/1.11.3/jquery.min.js"></script>

<!-- <script src="//cdn.bootcss.com/bootstrap/3.3.5/js/bootstrap.min.js"></script>-->

</head>

<body>

<div class="container">

<div>内容：</div>

<div class="list-group" id="divShow">

<!--<div class="list-group-item list-group-item-success">1111</div>

<div class="list-group-item list-group-item-info">1111</div>

<div class="list-group-item list-group-item-warning">1111</div>

<div class="list-group-item list-group-item-danger">1111</div>-->

</div>

<div class="list-group" id="divUsers">

在线：<br />

<!--<div class="btn btn-default">111</div>-->

</div>

<div>

昵称：<input class="form-control" id="txtUserName" value="红领巾" type="text" maxlength="20" style="width: 30%; margin-bottom: 15px" />

聊聊：<textarea class="form-control" id="txtContent" autofocus rows="6" placeholder="想聊的内容" maxlength="200" required style="width: 60%; "></textarea>

<button class="btn btn-default" id="btnSend" style="margin-top:15px">发 送</button>

</div>

</div>

</body>

</html>

<script>

var tool = function () {

var paperLoopNum = 0;

var paperTempleArr = [

'<div class="list-group-item list-group-item-success">{0}</div>',

'<div class="list-group-item list-group-item-info">{0}</div>',

'<div class="list-group-item list-group-item-warning">{0}</div>',

'<div class="list-group-item list-group-item-danger">{0}</div>'

];

return {

paperDiv: function (val) {

var hl = paperTempleArr[paperLoopNum];

paperLoopNum++;

if (paperLoopNum >= paperTempleArr.length) { paperLoopNum = 0; }

return this.formart(hl, [val])

},

formart: function (str, arrVal) {

for (var i = 0; i < arrVal.length; i++) {

str = str.replace("{" + i + "}", arrVal[i]);

}

return str;

}

}

}

function showMsg(id, hl, isAppend) {

if (!isAppend) { $("#" + id).html(hl); } else {

$("#" + id).append(hl);

}

}

$(function () {

//初始化工具方法

var tl = new tool();

var wsUrl = "ws://172.16.9.6:8/webSocket";

ws = new WebSocket(wsUrl);

try {

ws.onopen = function () {

//showMsg("divShow", tl.paperDiv("连接服务器-成功"));

}

ws.onclose = function () {

if (ws) {

ws.close();

ws = null;

}

showMsg("divShow", tl.paperDiv("连接服务器-关闭"), true);

}

ws.onmessage = function (result) {

//console.log(result.data);

var data = JSON.parse(result.data);

$(data.UserMsgs).each(function (i, item) {

showMsg("divShow", tl.paperDiv("【" + item.UserName + "】：" + item.Msg), true);

});

var userDataShow = [];

$(data.UserDatas).each(function (i, item) {

userDataShow.push('<div class="btn btn-default">' + item.UserName + '</div>');

});

showMsg("divUsers", userDataShow.join(''), false);

}

ws.onerror = function () {

if (ws) {

ws.close();

ws = null;

}

showMsg("divShow", tl.paperDiv("连接服务器-关闭"), true);

}

} catch (e) {

alert(e.message);

}

$("#btnSend").on("click", function () {

var tContentObj = $("#txtContent");

var tContent = $.trim( tContentObj.val()).replace("/[\n]/g", "");

var tUserName = $.trim( $("#txtUserName").val()); tUserName = tUserName.length <= 0 ? "匿名" : tUserName;

if (tContent.length <= 0 || $.trim(tContent).length <= 0) { alert("请输入发送内容!"); return; }

if (ws == null) { alert("连接失败，请F5刷新页面!"); return; }

var request = tl.formart('{"UserName": "{0}", "DataType": "{1}", "Msg": "{2}" }',

[tUserName, "send", tContent]);

ws.send(request);

tContentObj.val("");

tContentObj.val($.trim(tContentObj.val()).replace("/[\n]/g", ""));

});

$("#txtContent").on("keydown", function (event) {

if (event.keyCode == 13) {

$("#btnSend").trigger("click");

}

});

})

</script>