GO 提供原生的 websocket API，使用时 go get 然后引用即可

golang.org/x/net/websocket

使用起来也很方便，直接上代码吧。

一个 echo server 的代码

```
package main

import(
    "golang.org/x/net/websocket"
    "fmt"
    "net/http"
    "flag"
)

type WSServer struct {
    ListenAddr string
}

func (this *WSServer)handler(conn *websocket.Conn){
    fmt.Printf("a new ws conn: %s->%s\n", conn.RemoteAddr().String(),
conn.LocalAddr().String())
    var err error
    for {
        var reply string
        err = websocket.Message.Receive(conn, &reply)
        if err != nil {
            fmt.Println("receive err:",err.Error())
            break
        }
        fmt.Println("Received from client: " + reply)
```

```
        if err = websocket.Message.Send(conn, reply); err != nil {
            fmt.Println("send err:", err.Error())
            break
        }
    }
}
func (this *WSServer)start()(error){
    http.Handle("/ws", websocket.Handler(this.handler))
    fmt.Println("begin to listen")
    err := http.ListenAndServe(this.ListenAddr, nil)
    if err != nil {
        fmt.Println("ListenAndServe:", err)
        return err
    }
    fmt.Println("start end")
    return nil
}

func main(){
    addr  := flag.String("a", "127.0.1.1:12345", "websocket server listen address")
    flag.Parse()
    wsServer := &WSServer{
        ListenAddr : *addr,
    }
    wsServer.start()
    fmt.Println("------end-------")
}
```

上述代码中，每来一个新的 websocket client，server 会起一个 goroutine 执行 WSSever 的 handler 函数。

websocket client 代码实例

```
package main

import (
    "flag"
    "fmt"
    "time"
    "golang.org/x/net/websocket"
)

var addr = flag.String("addr", "127.0.0.1:12345", "http service address")

func main() {
    flag.Parse()

    url := "ws://"+ *addr + "/ws"
    origin := "test://1111111/"
    ws, err := websocket.Dial(url, "", origin)
    if err != nil {
        fmt.Println(err)
    }
    go timeWriter(ws)

    for {
        var msg [512]byte
        _, err := ws.Read(msg[:])//此处阻塞，等待有数据可读
        if err != nil {
            fmt.Println("read:", err)
            return
```

```
        }

        fmt.Printf("received: %s\n", msg)
    }
}

func timeWriter(conn *websocket.Conn) {
    for {
        time.Sleep(time.Second * 2)
        websocket.Message.Send(conn, "hello world")
    }
}
```

client 的代码，每隔 2 秒钟发送 hello world 到 server，然后阻塞在 Read 函数。需要注意的是 origin 必须以“http://1111111/” 这种标准的 URI 格式，否则报错“invalid URI for request”。

关闭进程、网络断线等异常情况

关闭进程

无论 server 还是 client，关闭进程，对端 Read 都会立刻收到 EOF，对 EOF 做处理即可。

网络断线

测试

client 和 server 部署在不同机器上，client 每隔 2 秒中向 server 发送数据，server 收到后回吐给客户端。这个过程中，拔掉 client 的网线。

测试结果

断网后，client 一定时间内写都能成功返回，但是因为断网实际没有发送出去，数据写到了底层 tcp 的缓冲区。

过一段时间后，1 分钟左右，client Read 返回错误“read: operation timed out”。Write 会返回“write: broken pipe”。这个可能是 Go 中 websocket 实现时加了超时机制，也有可能是设置了底层 TCP SO_KEEPALIVE，检测到了网络不可用。

在 Read/Write 返回错误之前，重新连上网络，可以继续发送和接受数据。这个可以从 TCP 的实现上解释。TCP 连接并不是物理连接，本质上就是连接两端各自系统内核维护的一个四元组。客户端断线，在一定时间内并不会导致四元组的释放。所以当连上网线后此 TCP 连接可以自动恢复，继续进行正常的网络操作。

断线重连到其他网络，相当于断网。这个很好解释，连上其他网络，IP 地址都改变了，之前的四元组不可用。

作者：阿冬哥

来源：CSDN

原文：https://blog.csdn.net/c359719435/article/details/78845719