

一个简单的golang TCP通信例子



getyouyou (/u/457e34ba02a9) [+ 关注](#)

2016.09.30 23:27* 字数 583 阅读 1756 评论 0 喜欢 4 阅读 1756 评论 0 喜欢 4

(/u/457e34ba02a9)

前言

新工作接手了公司的一个使用golang编写的agent程序，用于采集各个机器的性能指标和监控数据，之前使用http实现数据的上传，最近想把它改成tcp上传的方式，由于是新手上路，顺手写了一个小demo程序。

这个程序中包含：

- 简单的TcpServer服务程序：侦听，数据收发与解析
- 简单的客户端程序：数据收发与解析

服务器

与正常的其他语言一样，go中也提供了丰富的网络相关的包，按照正常的套路，它是这样的：

1. 绑定端口，初始化套接字
2. 启动侦听，开启后台线程接收客户端请求
3. 接收请求，针对每个请求开启一个线程来处理通信
4. 资源回收

golang的套路也是如此，不同的地方在于它可以使用goroutine来替换上面的线程；

整体的代码很简单，可以参考文档和api手册，示例代码如下：

```
package main

import (
    "fmt"
    "net"
    "os"
    "encoding/json"
    "bufio"
    "hash/crc32"
    "io"
)
//数据包的类型
const (
    HEART_BEAT_PACKET = 0x00
    REPORT_PACKET = 0x01
)

var (
    server = "127.0.0.1:8080"
)
//这里是包的结构体，其实是可以不需要的
type Packet struct {
    PacketType      byte
    PacketContent   []byte
}

//心跳包，这里用了json来序列化，也可以用github上的gogo/protobuf包
//具体见(https://github.com/gogo/protobuf)
type HeartPacket struct {
```

```
Version      string`json:"version"`
Timestamp    int64`json:"timestamp"`
}
//正式上传的数据包
type ReportPacket struct {
    Content      string`json:"content"`
    Rand         int`json:"rand"`
    Timestamp    int64`json:"timestamp"`
}
//与服务器相关的资源都放在这里面
type TcpServer struct {
    listener      *net.TCPListener
    hawkServer    *net.TCPAddr
}

func main() {
    //类似于初始化套接字，绑定端口
    hawkServer, err := net.ResolveTCPAddr("tcp", server)
    checkErr(err)
    //侦听
    listen, err := net.ListenTCP("tcp", hawkServer)
    checkErr(err)
    //记得关闭
    defer listen.Close()
    tcpServer := &TcpServer{
        listener:listen,
        hawkServer:hawkServer,
    }
    fmt.Println("start server successful.....")
    //开始接收请求
    for {
        conn, err := tcpServer.listener.Accept()
        fmt.Println("accept tcp client %s",conn.RemoteAddr().String())
        checkErr(err)
        // 每次建立一个连接就放到单独的协程内做处理
        go Handle(conn)
    }
}
//处理函数，这是一个状态机
//根据数据包来做解析
//数据包的格式为|0xFF|0xFF|len(高)|len(低)|Data|CRC高16位|0xFF|0xFE
//其中len为data的长度，实际长度为len(高)*256+len(低)
//CRC为32位CRC，取了最高16位共2Bytes
//0xFF|0xFF和0xFF|0xFE类似于前导码
func Handle(conn net.Conn) {
    // close connection before exit
    defer conn.Close()
    // 状态机状态
    state := 0x00
    // 数据包长度
    length := uint16(0)
    // crc校验和
    crc16 := uint16(0)
    var recvBuffer []byte
    // 游标
    cursor := uint16(0)
    bufferReader := bufio.NewReader(conn)
    //状态机处理数据
    for {
        recvByte,err := bufferReader.ReadByte()
        if err != nil {
            //这里因为做了心跳，所以就没有加deadline时间，如果客户端断开连接
            //这里ReadByte方法返回一个io.EOF的错误，具体可考虑文档
            if err == io.EOF {
                fmt.Printf("client %s is close!\n",conn.RemoteAddr().String())
            }
            //在这里直接关闭这个连接退出goroutine
            conn.Close()
            return
        }
        //进入状态机，根据不同的状态来处理
        switch state {
        case 0x00:
            if recvByte == 0xFF {
                state = 0x01
                //初始化状态机
                recvBuffer = nil
                length = 0
                crc16 = 0
            }else{
                state = 0x00
            }
            break
        case 0x01:
            if recvByte == 0xFF {
                state = 0x02
```

```

        }else{
            state = 0x00
        }
        break
    case 0x02:
        length += uint16(recvByte) * 256
        state = 0x03
        break
    case 0x03:
        length += uint16(recvByte)
        // 一次申请缓存, 初始化游标, 准备读数据
        recvBuffer = make([]byte, length)
        cursor = 0
        state = 0x04
        break
    case 0x04:
        //不断地在这个状态下读数据, 直到满足长度为止
        recvBuffer[cursor] = recvByte
        cursor++
        if(cursor == length){
            state = 0x05
        }
        break
    case 0x05:
        crc16 += uint16(recvByte) * 256
        state = 0x06
        break
    case 0x06:
        crc16 += uint16(recvByte)
        state = 0x07
        break
    case 0x07:
        if recvByte == 0xFF {
            state = 0x08
        }else{
            state = 0x00
        }
    case 0x08:
        if recvByte == 0xFE {
            //执行数据包校验
            if (crc32.ChecksumIEEE(recvBuffer) >> 16) & 0xFFFF == uint32(crc1
6) {
                var packet Packet
                //把拿到的数据反序列化出来
                json.Unmarshal(recvBuffer,&packet)
                //新开协程处理数据
                go processRecvData(&packet,conn)
            }else{
                fmt.Println("丢弃数据!")
            }
        }
        //状态机归位,接收下一个包
        state = 0x00
    }
}

//在这里处理收到的包, 就和一般的逻辑一样了, 根据类型进行不同的处理, 因人而异
//我这里处理了心跳和一个上报数据包
//服务器往客户端的数据包很简单地以\n换行结束了, 偷了一个懒:), 正常情况下也可根据自己的协议来封装好
//然后在客户端写一个状态来处理
func processRecvData(packet *Packet,conn net.Conn) {
    switch packet.PacketType {
    case HEART_BEAT_PACKET:
        var beatPacket HeartPacket
        json.Unmarshal(packet.PacketContent,&beatPacket)
        fmt.Printf("recieve heat beat from [%s] ,data is [%v]\n",conn.RemoteAddr().String(),beatPacket)
        conn.Write([]byte("heartBeat\n"))
        return
    case REPORT_PACKET:
        var reportPacket ReportPacket
        json.Unmarshal(packet.PacketContent,&reportPacket)
        fmt.Printf("recieve report data from [%s] ,data is [%v]\n",conn.RemoteAddr().String(),reportPacket)
        conn.Write([]byte("Report data has recive\n"))
        return
    }
}

//处理错误, 根据实际情况选择这样处理, 还是在函数调之后不同的地方不同处理
func checkErr(err error) {
    if err != nil {
        fmt.Println(err)
        os.Exit(-1)
    }
}

```

```
}
}
```

特别需要注意：

Handle方法在一个死循环中使用了一个无阻塞的buff来读取套接字中的数据，因此当客户端主动关闭连接时，如果不对这个io.EOF进行处理，会导致这个goroutine空转，疯狂吃cpu，在这里io.EOF的处理非常重要:)

客户端

客户端与一般的TCP通信程序一样，它需要完成的工作有：

- 1. 向服务器发送心跳包
- 2. 向服务器发送数据包
- 3. 接收服务器的数据包

需要注意的就是客户端与服务端的数据协议保持一致，请在开始发送数据之前启动数据接收

上面的3个工作我分别用了goroutine来做，整体的代码如下：

```
package main

import (
    "os"
    "fmt"
    "net"
    "time"
    "math/rand"
    "encoding/json"
    "bufio"
    "hash/crc32"
    "sync"
)
//数据包类型
const (
    HEART_BEAT_PACKET = 0x00
    REPORT_PACKET = 0x01
)
//默认的服务器地址
var (
    server = "127.0.0.1:9876"
)
//数据包
type Packet struct {
    PacketType      byte
    PacketContent    []byte
}
//心跳包
type HeartPacket struct {
    Version      string`json:"version"`
    Timestamp    int64`json:"timestamp"`
}
//数据包
type ReportPacket struct {
    Content      string`json:"content"`
    Rand         int`json:"rand"`
    Timestamp    int64`json:"timestamp"`
}

//客户端对象
type TcpClient struct {
    connection    *net.TCPConn
    hawkServer    *net.TCPAddr
    stopChan      chan struct{}
}

func main() {
    //拿到服务器地址信息
    hawkServer,err := net.ResolveTCPAddr("tcp", server)
    if err != nil {
        fmt.Printf("hawk server [%s] resolve error: [%s]",server,err.Error())
    }
}
```

```

        os.Exit(1)
    }
    //连接服务器
    connection,err := net.DialTCP("tcp",nil,hawkServer)
    if err != nil {
        fmt.Printf("connect to hawk server error: [%s]",err.Error())
        os.Exit(1)
    }
    client := &TcpClient{
        connection:connection,
        hawkServer:hawkServer,
        stopChan:make(chan struct{}),
    }
    //启动接收
    go client.receivePackets()

    //发送心跳的goroutine
    go func() {
        heartBeatTick := time.Tick(2 * time.Second)
        for{
            select {
            case <-heartBeatTick:
                client.sendHeartPacket()
            case <-client.stopChan:
                return
            }
        }
    }()

    //测试用的, 开300个goroutine每秒发送一个包
    for i:=0;i<300;i++ {
        go func() {
            sendTimer := time.After(1 * time.Second)
            for{
                select {
                case <-sendTimer:
                    client.sendReportPacket()
                    sendTimer = time.After(1 * time.Second)
                case <-client.stopChan:
                    return
                }
            }
        }()
    }
    //等待退出
    <-client.stopChan
}

// 接收数据包
func (client *TcpClient)receivePackets() {
    reader := bufio.NewReader(client.connection)
    for {
        //承接上面说的服务器端的偷懒, 我这里读也只是以\n为界限来读区分包
        msg, err := reader.ReadString('\n')
        if err != nil {
            //在这里也请处理如果服务器关闭时的异常
            close(client.stopChan)
            break
        }
        fmt.Print(msg)
    }
}

//发送数据包
//仔细看代码其实这里做了两次json的序列化, 有一次其实是不需要的
func (client *TcpClient)sendReportPacket() {
    reportPacket := ReportPacket{
        Content:getRandString(),
        Timestamp:time.Now().Unix(),
        Rand:rand.Int(),
    }
    packetBytes,err := json.Marshal(reportPacket)
    if err!=nil{
        fmt.Println(err.Error())
    }

    //这一次其实可以不需要, 在封包的地方把类型和数据传进去即可
    packet := Packet{
        PacketType:REPORT_PACKET,
        PacketContent:packetBytes,
    }
    sendBytes,err := json.Marshal(packet)
    if err!=nil{
        fmt.Println(err.Error())
    }
    //发送
    client.connection.Write(EnPackSendData(sendBytes))
    fmt.Println("Send metric data success!")
}

```

```
}

//使用的协议与服务器端保持一致
func EnPackSendData(sendBytes []byte) []byte {
    packetLength := len(sendBytes) + 8
    result := make([]byte,packetLength)
    result[0] = 0xFF
    result[1] = 0xFF
    result[2] = byte(uint16(len(sendBytes)) >> 8)
    result[3] = byte(uint16(len(sendBytes)) & 0xFF)
    copy(result[4:],sendBytes)
    sendCrc := crc32.ChecksumIEEE(sendBytes)
    result[packetLength-4] = byte(sendCrc >> 24)
    result[packetLength-3] = byte(sendCrc >> 16 & 0xFF)
    result[packetLength-2] = 0xFF
    result[packetLength-1] = 0xFE
    fmt.Println(result)
    return result
}

//发送心跳包，与发送数据包一样
func (client *TcpClient)sendHeartPacket() {
    heartPacket := HeartPacket{
        Version:"1.0",
        Timestamp:time.Now().Unix(),
    }
    packetBytes,err := json.Marshal(heartPacket)
    if err!=nil{
        fmt.Println(err.Error())
    }
    packet := Packet{
        PacketType:HEART_BEAT_PACKET,
        PacketContent:packetBytes,
    }
    sendBytes,err := json.Marshal(packet)
    if err!=nil{
        fmt.Println(err.Error())
    }
    client.connection.Write(EnPackSendData(sendBytes))
    fmt.Println("Send heartbeat data success!")
}

//拿一串随机字符
func getRandString()string {
    length := rand.Intn(50)
    strBytes := make([]byte,length)
    for i:=0;i<length;i++ {
        strBytes[i] = byte(rand.Intn(26) + 97)
    }
    return string(strBytes)
}
```

后记


测试过程中，一共开了7个client，共计2100个goroutine，本机启动服务器端，机器配置为i-5/8G的情况下，整体的资源使用情况如下：

PID	COMMAND	%CPU	TIME	#TH	#WQ	#PORT	MEM	PURG	CMPRS	PGRP	PPID	STATE	BOOSTS	%CPU_ME	%CPU_OTHR	UID	FAULTS	COW	MSGSENT	MSGRECV	SY
69069	screencaptur	0.0	00:00.17	2	0	52	2276K	20K	0B	269	269	sleeping	*0[1]	0.00000	0.00000	501	11151	288	4478	1476	16
69022	mdworker	0.0	00:00.03	3	0	43	2940K	0B	0B	69022	1	sleeping	*0[1]	0.00000	0.00000	501	3430	170	608	271	14
69021	mdworker	0.0	00:00.03	3	0	43	2980K	0B	0B	69021	1	sleeping	*0[1]	0.00000	0.00000	501	3435	170	587	263	12
69020	top	3.4	00:01.43	1/1	0	24	4344K	0B	0B	69020	51061	running	*0[1]	0.00000	0.00000	0	9783+	96	526314+	263120+	33
68989	ReportCrash	0.0	00:00.01	4	1	22	948K	0B	0B	68989	1	sleeping	*0[1]	0.00000	0.00000	0	1924	136	94	37	38
68973	tcpClient	3.0	00:01.55	125	0	386	9464K	0B	0B	68973	68698	sleeping	*0[1]	0.00000	0.00000	501	2981	86	285	142	12
68956	mdworker	0.0	00:00.04	3	0	43	2972K	0B	0B	68956	1	sleeping	*0[1]	0.00000	0.00000	501	3463	170	621	277	15
68955	mdworker	0.0	00:00.04	3	0	43	3504K	0B	0B	68955	1	sleeping	*0[1]	0.00000	0.00000	501	4014	170	599	268	14
68698	bash	0.0	00:00.02	1	0	17	792K	0B	0B	68698	68697	sleeping	*0[1]	0.00000	0.00000	501	1278	374	105	45	13
68697	login	0.0	00:00.03	2	0	29	1124K	0B	0B	68697	51031	sleeping	*0[9]	0.00000	0.00000	0	1408	162	192	68	17
68681	tcpClient	3.5	00:02.06	176	0	539	10M	0B	0B	68681	51433	sleeping	*0[1]	0.00000	0.00000	501	3189	79	387	193	14
68665	tcpClient	3.5	00:02.29	111	0	344	8616K	0B	0B	68665	51197	sleeping	*0[1]	0.00000	0.00000	501	2775	82	258	128	16
68649	tcpClient	4.1	00:02.31	209	0	638	11M	0B	0B	68649	51254	sleeping	*0[1]	0.00000	0.00000	501	3554	82	454	226	15
68646	tcpClient	4.1	00:02.55	156	0	479	9560K	0B	0B	68646	51287	sleeping	*0[1]	0.00000	0.00000	501	3001	76	347	173	19
68645	tcpClient	3.4	00:02.53	150	0	461	9580K	0B	0B	68645	51045	sleeping	*0[1]	0.00000	0.00000	501	3005	80	335	166	18
68629	tcpClient	3.0	00:02.76	143	0	440	10M	0B	0B	68629	51044	sleeping	*0[1]	0.00000	0.00000	501	3270	84	323	160	21
68319	bash	0.0	00:00.01	1	0	17	824K	0B	0B	68319	68318	sleeping	*0[1]	0.00000	0.00000	501	955	301	74	31	85
68318	login	0.0	00:00.26	2	0	29	1132K	0B	0B	68318	51031	sleeping	*0[9]	0.00000	0.00000	0	1389	167	192	68	76
58287	tcpServer	18.1	00:11.07	120	0	371	9860K+	0B	0B	68287	51742	sleeping	*0[1]	0.00000	0.00000	501	3116+	83	278	137	64

测试结果.png

需要改进的地方，也是后两篇的主题：

- 引入内存池
- 服务无缝重启



getyouyou (/u/457e34ba02a9)

写了 1657 字，被 5 人关注，获得了 4 个喜欢

(/u/457e34ba02a9) 写了 1657 字，被 5 人关注，获得了 4 个喜欢

+ 关注

阿里，互联网

如果觉得我的文章对您有用，请随意赞赏。您的支持将鼓励我继续创作！

赞赏支持

♡ 喜欢 (/sign_in?utm_source=desktop&utm_medium=not-signed-in-like-button)

4







更多分享

(http://cwb.assets.jianshu.io/notes/images/6086965




发表评论

(/sign_in?utm_source=desktop&utm_medium=not-signed-in-comment-form)


评论

智慧如你，不想发表一点想法 (/sign_in?utm_source=desktop&utm_medium=not-signed-in-nocomments-text)咩~

 登录/注册

为你个性化推荐内容

(/sign_in?utm_source=desktop&utm_medium=not-signed-in-bind)

 下载简书App

随时随地发现和创作内容

(/apps/download?utm_source=desktop&utm_medium=click-note-bottom-bind)