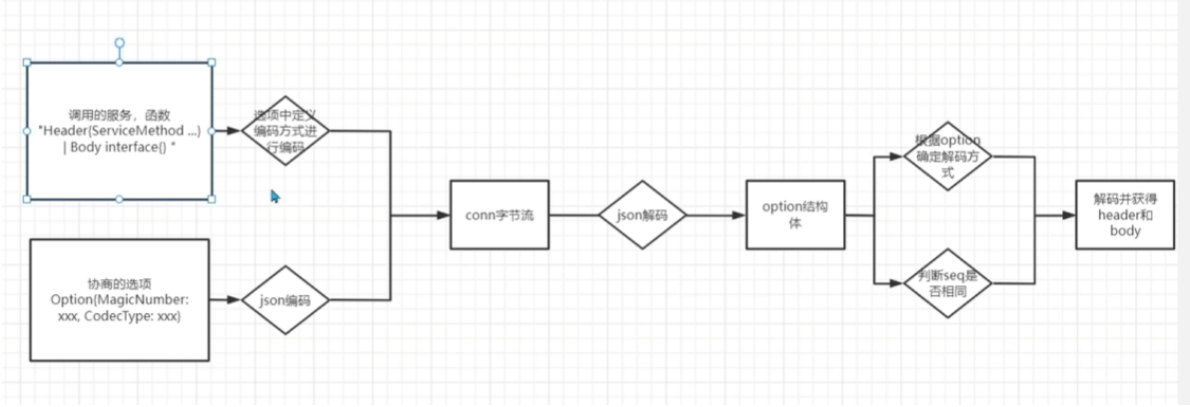
第一章 项目架构



|----codec：服务端和客户端之间的编解码
  |----codec.go：对消息体进行编解码的接口
  |----gob.go：使用gob包实现对消息体的编解码
|----loadBalancer：负载均衡组件
  |----discovery.go：负载均衡服务端
  |----wgRegistryDiscovery.go：与注册中心连接的负载均衡服务端
  |----xclient.go：支持负载均衡的客户端
|----registry：注册中心
  |----registry.go
|----client.go：rpc客户端
|----server.go：rpc服务端
|----service.go：rpc服务
![[Pasted image 20240703201927.png]]

第二章 代码讲解

RPC的调用方式：`err = client.Call("Arith.Multiply", args, &reply)`

1. 客户端传三个参数：服务名.方法名、参数args、返回值reply。
2. 服务端将处理结果写入reply返回，同时返回error。

一、 /codec - 消息的序列化和反序列化

1. Header

将请求和响应中的参数和返回值（args、reply）抽象为body，剩余信息放在header中。

```
type Header struct {
    ServiceMethod string //服务名和方法名，与GO中的结构体和方法相映射
    Seq           uint64 //请求序号
    Error         string //错误信息
}
```

2. Codec

用于对消息体进行编解码的接口

```
type Codec interface {
    io.Closer
    ReadHeader(*Header) error
    ReadBody(interface{}) error
    Write(*Header, interface{}) error
}
```

为了能够让用户自己选择使用哪种编解码方式，抽象出Codec构造函数。客户端和服务端通过Codec的Type得到构造函数，从而创建Codec实例。

```
type NewCodecFunc func(io.ReadWriteCloser) Codec
type Type string
const (
//gob和json两种编解码方式
    GobType  Type = "application/gob"
    JsonType Type = "application/json"
)
var NewCodecFuncMap map[Type]NewCodecFunc
func init() {
    NewCodecFuncMap = make(map[Type]NewCodecFunc)
    NewCodecFuncMap[GobType] = NewGobCodec
}
```

3. GobCodec

使用gob包进行消息编解码

```
type GobCodec struct {
    conn io.ReadWriteCloser //链接实例
    buf  *bufio.Writer      //缓冲区
    dec  *gob.Decoder       //decoder
    enc  *gob.Encoder       //encoder
}
```

实现Codec接口

```
var _ Codec = (*GobCodec)(nil)

func (g *GobCodec) Close() error {
    return g.conn.Close()
}
func (g *GobCodec) ReadHeader(header *Header) error {
    return g.dec.Decode(header)
}
func (g *GobCodec) ReadBody(body interface{}) error {
    return g.dec.Decode(body)
}

func (g *GobCodec) Write(header *Header, body interface{}) (err error) {
    defer func() {
       g.buf.Flush()
```

```
        if err != nil {
            g.Close()
        }
    }()
    if err := g.enc.Encode(header); err != nil {
        log.Println("codec.GobCoder.Write:Encoding header ERR:", err)
        return err
    }
    if err := g.enc.Encode(body); err != nil {
        log.Println("codec.GobCoder.Write:Encoding body ERR:", err)
        return err
    }
    return nil
}
```

实现上一小节NewCodecFunc中的的NewGobCodec函数（工厂模式）

```
func NewGobCodec(conn io.ReadWriteCloser) Codec {
    buf := bufio.NewWriter(conn)
    return &GobCodec{
        conn: conn,
        buf:  buf,
        dec:  gob.NewDecoder(conn),
        enc:  gob.NewEncoder(buf),
    }
}
```

二、/service.go 服务注册

用于将结构体映射为服务。

对 net/rpc 而言，一个函数需要能够被远程调用，需要满足如下五个条件:
1.方法所属类型是导出的
2.方式是导出的
3.两个入参均为导出或内置类型
4.第二个入参必须是一个指针
5.返回值为 error 类型
即：`func (t *T) MethodName(argType T1,replyType *T2)error  {}`

借助反射来使映射过程自动化，获取某个结构体的所有方法，获取该方法的所有参数类型和返回值

1. methodType

结构体

```
type methodType struct {
    method    reflect.Method //方法本身
    ArgType   reflect.Type   //方法的第一个参数的类型
    ReplyType reflect.Type   //第二个参数的类型
    numCalls  uint64         //用于统计方法调用次数
}

// NumCalls 原子获取调用次数
func (m *methodType) NumCalls() uint64 {
    return atomic.LoadUint64(&m.numCalls)
}
```

创建Argv的实例

```
func (m *methodType) newArgv() reflect.Value {
    var argv reflect.Value
    if m.ArgType.Kind() == reflect.Ptr {
       argv = reflect.New(m.ArgType.Elem())
    } else {
       argv = reflect.New(m.ArgType).Elem()
    }
    return argv
}
```

创建Replyv的实例

```
func (m *methodType) newReplyv() reflect.Value {
    replyv := reflect.New(m.ReplyType.Elem())
    switch m.ReplyType.Elem().Kind() {
    case reflect.Map:
       replyv.Elem().Set(reflect.MakeMap(m.ReplyType.Elem()))
    case reflect.Slice:
       replyv.Elem().Set(reflect.MakeSlice(m.ReplyType.Elem(), 0, 0))
    }
    return replyv
}
```

2. service

结构体

```
type service struct {
    name   string                 //映射的结构体的名称
    typ    reflect.Type           //结构体的类型
    rcvr   reflect.Value          //结构体的实例本身
    method map[string]*methodType //存储映射的结构体的所有符合条件的方法。
}

func newService(rcvr interface{}) *service {
    s := new(service)
    s.rcvr = reflect.ValueOf(rcvr)
    s.name = reflect.Indirect(s.rcvr).Type().Name()
    s.typ = reflect.TypeOf(rcvr)
    //判断结构体是否外部可见
    if !ast.IsExported(s.name) {
       log.Fatalf("service.newService: %s is not a valid service name",
s.name)
    }
    s.registerMethods()
    return s
}
```

注册方法

筛选出符合条件的方法，放入service.method中

```
func (service *service) registerMethods() {
    service.method = make(map[string]*methodType)
    //遍历该结构体的所有方法
    for i := 0; i < service.typ.NumMethod(); i++ {
       method := service.typ.Method(i)
       mType := method.Type
       //判断入参是否等于3
       if mType.NumIn() != 3 || mType.NumOut() != 1 {
          continue
       }
       if mType.Out(0) != reflect.TypeOf((*error)(nil)).Elem() {
          continue
       }
       //0是它自身（即this），1是第一个参数，2是第二个参数
       argType, replyType := mType.In(1), mType.In(2)
       if !isExportedOrBuiltinType(argType) ||
!isExportedOrBuiltinType(replyType) {
          continue
       }
       //放入service.method中
       service.method[method.Name] = &methodType{
          method:    method,
          ArgType:   argType,
          ReplyType: replyType,
       }
       log.Printf("rpc server: register %s.%s\n", service.name, method.Name)
    }
}

func isExportedOrBuiltinType(t reflect.Type) bool {
    return ast.IsExported(t.Name()) || t.PkgPath() == ""
}
```

调用方法

通过反射调用方法

```
func (service *service) call(m *methodType, argv, replyv reflect.Value) error
{
    atomic.AddUint64(&m.numCalls, 1)
    f := m.method.Func
    returnValues := f.Call([]reflect.Value{service.rcvr, argv, replyv})
    if errInter := returnValues[0].Interface(); errInter != nil {
       return errInter.(error)
    }
    return nil
}
```

三、/server.go - 服务端

1. 通信过程

客户端与服务端之间的通信，需要协商一部分内容。对于 RPC 协议来说，这部分协商是需要自主设计的。为了提升性能，一般在报文的最开始会规划固定的字节，来协商相关的信息。比如第1个字节用来表示序列化方式，第2个字节表示压缩方式，第3-6字节表示 header 的长度，7-10 字节表示 body 的长度。
对于本项目来说，只需要协商消息的编解码方式、过期时间。我们将这部分信息放在Option结构体中承载。
![[Pasted image 20240629213857.png]]

```
const MagicNumber      = 0x03719666

type Option struct {
    MagicNumber    int32
    CodecType      codec.Type
    ConnectTimeout time.Duration //time.Duration用于表示持续时间`
    HandleTimeout  time.Duration
}

var DefaultOption = &Option{
    MagicNumber:    MagicNumber,
    CodecType:      codec.GobType,
    ConnectTimeout: time.Second * 10, //设置默认值为10s
    //HandleTimeout不设置默认值，即为0秒
}
```

2. 集成service

从接收到请求到回复还差以下几个步骤：

1. 根据入参类型，将请求的 body 反序列化
2. 调用 `service.call`，完成方法调用
3. 将 reply 序列化为字节流，构造响应报文，返回

结构体

只包含一个参数：serviceMap，是并发安全的map，用于保存service

```
type Server struct {
    serviceMap sync.Map
}

func NewServer() *Server {
    return &Server{}
}

var DefaultServer = NewServer()

```

将方法注册到服务端

```go
func (server *Server) Register(rcvr interface{}) error {
    s := newService(rcvr)
    if _, dup := server.serviceMap.LoadOrStore(s.name, s); dup {
        return errors.New("server.Register: service already defined: " + s.name)
    }
    return nil
}

// Register 公共接口，用于注册方法
func Register(rcvr interface{}) error {
    return DefaultServer.Register(rcvr)
}
```

服务端寻找对应的服务

1. 将ServiceMethod分割成两部分：Service名称、方法名。
2. 再serviceMap中找到对应的service实例。
3. 从service实例的method中，找到对应的methodType。

```go
func (server *Server) findService(serviceMethod string) (svc *service, mtype *methodType, err error) {
    dot := strings.LastIndex(serviceMethod, ".")
    if dot < 0 {
        err = errors.New("server.Register: service/method request ill-formed: " + serviceMethod)
        return
    }
    serviceName, methodName := serviceMethod[:dot], serviceMethod[dot+1:]
    //读取对应的service
    svci, ok := server.serviceMap.Load(serviceName)
    if !ok {
        err = errors.New("server.findService: can't find service: " + serviceName)
        return
    }
    svc = svci.(*service)
    mtype = svc.method[methodName]
    if mtype == nil {
        err = errors.New("server.findService: can't find method: " + methodName)
    }
    return
}
```

3. 与客户端client连接

3.1 建立tcp连接

Accept传入net.Listener，for循环等待socket简历，并开启协程，然后将处理过程交给ServerConn方法。

```go
func Accept(listener net.Listener) { DefaultServer.Accept(listener) }

func (server *Server) Accept(listner net.Listener) {
    //建立socket连接
```

```
    for {
        conn, err := listner.Accept()
        if err != nil {
            log.Println("server.Accept:", err)
            return
        }
        //开启子携程处理连接
        go server.ServeConn(conn)
    }
}
```

启动服务示例

```
lis, _ := net.Listen("tcp", ":9999")
wgRPC.Accept(lis)
```

3.2 解析请求

ServeConn

将Option解码出来

1. 反序列化得到Option，并进行验证
2. 根据CodeType得到对应的消息编解码器
3. 将处理交给serverCodec

```
func (server *Server) ServeConn(conn io.ReadWriteCloser) {
    defer conn.Close()
    var opt Option
    //根据CodeType得到对应的消息编解码器
    if err := json.NewDecoder(conn).Decode(&opt); err != nil {
        log.Println("server.ServeConn: option ERR:", err)
        return
    }
    //验证妙妙数字
    if opt.MagicNumber != MagicNumber {
        log.Printf("server.ServeConn: magic number ERR:%x \n",
opt.MagicNumber)
        return
    }
    //创建map并验证数据类型
    f := codec.NewCodecFuncMap[opt.CodecType]
    if f == nil {
        log.Printf("server.ServeConn: invalid codec type ERR: %s \n",
opt.CodecType)
        return
    }
    //使用serveCodec处理消息
    server.serveCodec(f(conn), &opt)
}
```

serveCodec

循环处理Option后面的各个Header-Body
主要包含以下三个步骤：

1. 读取请求 readRequest
2. 处理请求 handleRequest
3. 回复请求 sendRequest

请求的处理是并发的，但是回复必须是串行的，这里使用锁`sending`保证。
当header解析失败时才会终止循环。

```
// 如果发生错误则发送这个空body给客户端
var invalidRequest = struct{}{}

func (server *Server) serveCodec(codec codec.Codec, opt *Option) {
    sending := new(sync.Mutex)
    wg := new(sync.WaitGroup) //等待所有请求被处理完，
    //循环直到发生错误（例如连接被关闭，接收到的报文有问题等），这使得一次链接可以接收多个请求
    for {
       //读取请求
       req, err := server.readRequest(codec)
       if err != nil {
          if req == nil {
             break
          }
          req.header.Error = err.Error()
          //发生错误时回复 invalidRequest
          server.sendResponse(codec, req.header, invalidRequest, sending)
          continue
       }
       wg.Add(1)
       //处理请求
       go server.handleRequest(codec, req, sending, wg, opt.HandleTimeout)
    }
    wg.Wait()
    codec.Close()
}
```

4 处理请求

请求体

```
type request struct {
    header       *codec.Header
    argv, replyV reflect.Value //反射获得类型
    mtype        *methodType
    svc          *service //服务
}
```

读取请求头

```go
func (server *Server) readRequestHeader(c codec.Codec) (*codec.Header,
error) {
    var header codec.Header
    if err := c.ReadHeader(&header); err != nil {
       if err != io.EOF && err != io.ErrUnexpectedEOF {
          log.Println("server.readRequestHeader: read header ERR:", err)
       }
       return nil, err
    }
    return &header, nil
}
```

读取请求

1. 通过 `findService()` 找到对应服务
2. 通过 `newArgv()` 和 `newReplyv()` 两个方法创建出两个入参实例
3. 通过 `codec.ReadBody()` 将请求报文反序列化为第一个入参 argv
    - 注意argv可能是值类型或指针类型，所以处理方式不同

```go
func (server *Server) readRequest(c codec.Codec) (*request, error) {
    header, err := server.readRequestHeader(c)
    if err != nil {
       return nil, err
    }
    //从server中读取出request
    req := &request{
       header: header,
    }
    req.svc, req.mtype, err = server.findService(header.ServiceMethod)
    if err != nil {
       return req, err
    }
    //创建入参实例
    req.argv = req.mtype.newArgv()
    req.replyV = req.mtype.newReplyv()
    argvi := req.argv.Interface()
    //确保argvi是指针类型
    if req.argv.Type().Kind() != reflect.Ptr {
       argvi = req.argv.Addr().Interface()
    }
    //将请求报文反序列化为第一个入参argv
    err = c.ReadBody(argvi)
    if err != nil {
       log.Println("server.readRequest: read body ERR: ", err)
       return req, err
    }
    return req, nil
}
```

处理请求

- service
  - 通过`req.svc.call`完成方法调用
  - 将replyv传递给sendResponse完成序列化
- 超时处理：[[#2.3 服务端处理超时]]

```
func (server *Server) handleRequest(c codec.Codec, req *request, sending *sync.Mutex, wg *sync.WaitGroup, timeout time.Duration) {
    defer wg.Done()
    //将过程拆为call和sent两个阶段，以确保sendResponse仅调用一次
    called := make(chan struct{})
    sent := make(chan struct{})
    go func() {
       err := req.svc.call(req.mtype, req.argv, req.replyV)
       called <- struct{}{}
       if err != nil {
          req.header.Error = err.Error()
          server.sendResponse(c, req.header, invalidRequest, sending)
          sent <- struct{}{}
          return
       }
       server.sendResponse(c, req.header, req.replyV.Interface(), sending)
       sent <- struct{}{}
    }()
    if timeout == 0 {
       <-called
       <-sent
       return
    }
    select {
    //处理超时，则阻塞called和sent，调用sendResponse
    case <-time.After(timeout):
       req.header.Error = fmt.Sprintf("server.handleRequest: request handle timeout: expect within %s", timeout)
       server.sendResponse(c, req.header, invalidRequest, sending)
    case <-called:
       <-sent
    }
}
```

发送响应

```
func (server *Server) sendResponse(c codec.Codec, header *codec.Header, body interface{}, sending *sync.Mutex) {
    sending.Lock()
    defer sending.Unlock()
    if err := c.Write(header, body); err != nil {
       log.Println("server.sendResponse: write response ERR: ", err)
    }
}
```

5. 支持HTTP协议

阅读http包的源码，我们可以看到：

```
package http
// Handle registers the handler for the given pattern
// in the DefaultServeMux.
// The documentation for ServeMux explains how patterns are matched.
func Handle(pattern string, handler Handler) {
     DefaultServeMux.Handle(pattern, handler)
}

type Handler interface {
    ServeHTTP(w ResponseWriter, r *Request)
}
```

只需要实现接口 Handler 即可作为一个 HTTP Handler 处理 HTTP 请求。接口 Handler 只定义了一个方法 `ServeHTTP`，实现该方法即可。

```
const (
    connected        = "200 Connected to Wg RPC"
    defaultRPCPath   = "/_wgprc_"
    defaultDebugPath = "/debug/wgrpc"  //为后续DEBUG页面预留的地址
)

// 实现http包中的Handler，将requests发送给RPC
func (server *Server) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    if req.Method != "CONNECT" {
       w.Header().Set("Content-Type", "text/plain; charset=utf-8")
       w.WriteHeader(http.StatusMethodNotAllowed)
       io.WriteString(w, "405 must CONNECT\n")
       return
    }
    conn, _, err := w.(http.Hijacker).Hijack()
    if err != nil {
       log.Print("server.serveHTTP: hijacking ERR: ", req.RemoteAddr, ": ",
err.Error())
       return
    }
    io.WriteString(conn, "HTTP/1.0 "+connected+"\n\n")
    server.ServeConn(conn)
}

func (server *Server) HandleHTTP() {
    http.Handle(defaultRPCPath, server)
    http.Handle(defaultDebugPath, debugHTTP{server})
    log.Println("server.HandleHTTP: server debug path: ", defaultDebugPath)
}

func HandleHTTP() {
    DefaultServer.HandleHTTP()
}
```

四、/client.go - 客户端

1. Call

用于承载一次RPC调用所需要的信息

```
type Call struct {
    Seq           uint64
    ServiceMethod string
    Args          interface{} //函数的参数
    Reply         interface{} //回复
    Error         error
    Done          chan *Call //Call完成时放入chan中
}

// 调用结束后调用此函数通知调用方。用于支持异步调用。
func (call *Call) done() {
    call.Done <- call
}
```

2. Client

2.1 Client结构体

结构体

```
type Client struct {
    c        codec.Codec //消息编解码器
    opt      *Option
    header   codec.Header //每个消息的请求头
    sending  sync.Mutex   //互斥锁，保证请求有序发送
    mutex    sync.Mutex
    seq      uint64           //用于给发送的请求编号，每个请求拥有唯一编号。
    pending  map[uint64]*Call //未处理完的请求（key：编号，val：Call实例）
    //closing或shutdown为true时，表示Client不可用。
    closing  bool             //用户决定停止
    shutdown bool             //服务器通知停止（有错误发生）
}

// 超时处理包装
type clientResult struct {
    client *Client
    err    error
}
```

创建Client

```
func NewClient(conn net.Conn, option *Option) (*Client, error) {
    //协议交换（发送Option信息给服务端，协商编解码方式）
    f := codec.NewCodecFuncMap[option.CodecType]
    if f == nil {
       err := fmt.Errorf("invalid codec type %s", option.CodecType)
       log.Println("client.NewClient: codec ERR: ", err)
       conn.Close()
       return nil, err
    }
    if err := json.NewEncoder(conn).Encode(option); err != nil {
       log.Println("client.NewClient: options ERR: ", err)
```

```
        conn.Close()
        return nil, err
    }
    client := &Client{
        c:       f(conn),
        opt:     option,
        seq:     1,
        pending: make(map[uint64]*Call),
    }

    //创建协程,调用receive()接收响应
    go client.receive()
    return client, nil
}
```

关闭Client

```
var _ io.Closer = (*Client)(nil)
var ErrShutdown = errors.New("connection is shut down")

// Close 关闭连接
func (client *Client) Close() error {
    client.mutex.Lock()
    defer client.mutex.Unlock()
    if client.closing {
        return ErrShutdown
    }
    client.closing = true
    return client.c.Close()
}

// IsAvailable 判断是否还在工作(是则返回True)
func (client *Client) IsAvailable() bool {
    client.mutex.Lock()
    defer client.mutex.Unlock()
    return !client.shutdown && !client.closing
}
```

接收响应

```
func (client *Client) receive() {
    var err error
    for err == nil {
        var header codec.Header
        if err = client.c.ReadHeader(&header); err != nil {
            break
        }
        call := client.removeCall(header.Seq)
        switch {
        case call == nil: //call不存在
            err = client.c.ReadBody(nil)
        case header.Error != "": //call存在但是服务端处理错误，即header.Error不为空
            call.Error = fmt.Errorf(header.Error)
            err = client.c.ReadBody(nil)
            call.done()
        default: //服务端处理正常
```

```
            err = client.c.ReadBody(call.Reply)
            if err != nil {
                call.Error = errors.New("reading body" + err.Error())
            }
            call.done()
        }
    }
    //发生错误，则关掉pending中的所有Call
    client.terminateCalls(err)
}
```

2.2 用户创建Client入口

Dial - 入口

用户传入服务端地址，创建Client实例。
使用了`net.DialTimeout`进行超时处理，利用channel捕获超时

```
func Dial(network, address string, options ...*Option) (client *Client, err error) {
    //解析option
    option, err := parseOptions(options...)
    if err != nil {
        return nil, err
    }
    //用net.DialTimeout防止超时（传入设置的时间）
    conn, err := net.DialTimeout(network, address, option.ConnectTimeout)
    if err != nil {
        return nil, err
    }
    defer func() {
        if err != nil {
            conn.Close()
        }
    }()

    ch := make(chan clientResult)
    go func() {
        client, err := NewClient(conn, option)
        ch <- clientResult{
            client: client,
            err:    err,
        }
    }()
    if option.ConnectTimeout == 0 {
        result := <-ch
        return result.client, result.err
    }

    select {
    case <-time.After(option.ConnectTimeout):
        return nil, fmt.Errorf("client.Dial: connect timeout: expect within %s", option.ConnectTimeout)
    case result := <-ch:
        return result.client, result.err
    }
}
```

解析Option

通过`...*Option`将`Option`实现为可选参数

```
func parseOptions(options ...*Option) (*Option, error) {
    if len(options) == 0 || options[0] == nil {
       return DefaultOption, nil
    }
    if len(options) != 1 {
       return nil, errors.New("number of options is more than 1")
    }
    option := options[0]
    option.MagicNumber = DefaultOption.MagicNumber
    if option.CodecType == "" {
       option.CodecType = DefaultOption.CodecType
    }
    return option, nil
}
```

2.3 处理Call

注册Call

将参数call添加到client.pending中，并更新client.seq

```
func (client *Client) registerCall(call *Call) (uint64, error) {
    client.mutex.Lock()
    defer client.mutex.Unlock()
    if client.closing || client.shutdown {
       return 0, ErrShutdown
    }
    call.Seq = client.seq
    client.pending[call.Seq] = call
    client.seq++
    return call.Seq, nil
}
```

获取Call

根据seq从pending中获取对应的call

```
func (client *Client) removeCall(seq uint64) *Call {
    client.mu.Lock()
    defer client.mu.Unlock()
    call := client.pending[seq]
    delete(client.pending, seq)
    return call
}
```

关闭所有Call

客户端或服务端发生错误时调用，将客户端shutdown然后通知所有pending状态的call

```go
func (client *Client) terminateCalls(err error) {
    client.sending.Lock()
    defer client.sending.Unlock()
    client.mu.Lock()
    defer client.mu.Unlock()
    client.shutdown = true
    for _, call := range client.pending {
        call.Error = err
        call.done()
    }
}
```

发送响应

```go
func (client *Client) send(call *Call) {
    client.sending.Lock()
    defer client.sending.Unlock()
    //注册这个call
    seq, err := client.registerCall(call)
    if err != nil {
       call.Error = err
       call.done()
       return
    }
    //组装header
    client.header.ServiceMethod = call.ServiceMethod
    client.header.Seq = seq
    client.header.Error = ""
    //编码并发送请求
    if err := client.c.Write(&client.header, call.Args); err != nil {
       call := client.removeCall(seq)
       if call != nil {
          call.Error = err
          call.done()
       }
    }
}
```

2.4 用户调用入口

Go - 异步接口

Go是一个异步接口，返回call实例。

```go
func (client *Client) Go(serviceMethod string, args, reply interface{}, done
chan *Call) *Call {
    if done == nil {
       done = make(chan *Call, 10)
    } else if cap(done) == 0 {
       log.Panic("client.go: done channel")
    }
    call := &Call{
       ServiceMethod: serviceMethod,
       Args:          args,
       Reply:         reply,
       Done:          done,
```

```
    }
    client.send(call)
    return call
}
```

Call - 同步接口

Call是一个同步接口，是对Go的封装，阻塞call.Done，等待响应返回。

```
func (client *Client) Call(ctx context.Context, serviceMethod string, args, reply interface{}) error {
    call := client.Go(serviceMethod, args, reply, make(chan *Call, 1))
    //用context包实现超时处理，控制权交给用户
    select {
    case <-ctx.Done():
       client.removeCall(call.Seq)
       return errors.New("client.Call: call failed: " + ctx.Err().Error())
    case call := <-call.Done:
       return call.Error
    }
}
```

3. 支持HTTP协议

连接服务端

客户端发起CONNETC请求，检查返回的状态码

```
func NewHTTPClient(conn net.Conn, option *Option) (*Client, error) {
    io.WriteString(conn, fmt.Sprintf("CONNECT %s HTTP/1.0\n\n", defaultRPCPath))
    resp, err := http.ReadResponse(bufio.NewReader(conn), &http.Request{Method: "CONNECT"})
    //连接上了的话创建新客户端
    if err == nil && resp.Status == connected {
       return NewClient(conn, option)
    }
    if err == nil {
       err = errors.New("unexpected HTTP response: " + resp.Status)
    }
    return nil, err
}

// DialHTTP 通过HTTP CONNECT请求建立连接，连接上HTTP RPC服务器
func DialHTTP(network, address string, opts ...*Option) (*Client, error) {
    return dialTimeout(NewHTTPClient, network, address, opts...)
}
```

用户入口

```
// XDial 使用不同的方法去连接RPC server
func XDial(rpcAddr string, opts ...*Option) (*Client, error) {
    //根据rpcAddr
    parts := strings.Split(rpcAddr, "@")
    if len(parts) != 2 {
```

```
        return nil, fmt.Errorf("client.XDial: client ERR: wrong format '%s', expect protocol@addr", rpcAddr)
    }
    protocol, addr := parts[0], parts[1]
    switch protocol {
    case "http":
        return DialHTTP("tcp", addr, opts...)
    default:
        //tcp、unix或其他传输协议
        return Dial(protocol, addr, opts...)
    }
}
```

五、/xclient - 负载均衡

负载均衡是接下来要实现的注册中心的基础，主要有以下作用：

1. 提高系统负载
2. 避免单点故障
3. 提高系统可用
4. 提高响应速度

常见的负载均衡策略有：

- 随机选择策略 - 从服务列表中随机选择一个。
- 轮询算法(Round Robin) - 依次调度不同的服务器，每次调度执行 i = (i + 1) mode n。
- 加权轮询(Weight Round Robin) - 在轮询算法的基础上，为每个服务实例设置一个权重，高性能的机器赋予更高的权重，也可以根据服务实例的当前的负载情况做动态的调整，例如考虑最近5分钟部署服务器的 CPU、内存消耗情况。
- 哈希/一致性哈希策略 - 依据请求的某些特征，计算一个 hash 值，根据 hash 值将请求发送到对应的机器。一致性 hash 还可以解决服务实例动态添加情况下，调度抖动的问题。一致性哈希的一个典型应用场景是分布式缓存服务。

本框架只做了RoundRobin和Random，通过SelectMode来选择负载均衡策略：

```
type SelectMode int

const (
    RandomSelect     SelectMode = iota // select randomly
    RoundRobinSelect                   // select using Robbin algorithm
)
```

负载均衡通过一个基础的服务发现模块discovery + 一个支持负载均衡的客户端xclient来实现

1. discovery.go - 服务发现

负载均衡功能的服务端，保存了服务列表，通过负载均衡策略找到合适的服务实例。

Discovery接口

Discovery 是一个接口类型，包含了服务发现所需要的最基本的接口。

```go
type Discovery interface {
    Refresh() error                      //从注册中心更新服务表
    Update(servers []string) error       //手动更新服务列表
    Get(mode SelectMode) (string, error) //根据负载均衡策略，选择服务实例
    GetAll() ([]string, error)           //返回所有服务实例
}
```

实现接口

创建MultiServerDiscovery用于实现Discovery接口
用户需要提供服务的地址
构造函数会初始化一个随机数种子，以及用于RoundRobin算法的index

```go
// MultiServerDiscovery 一个不需要注册中心，服务列表由手工维护的服务发现结构体
// 用户显示提供服务器地址
type MultiServerDiscovery struct {
    r       *rand.Rand // 用于生成随机数
    mutex   sync.RWMutex
    servers []string
    index   int //用于记录Robin算法的选定位置
}

// NewMultiServerDiscovery 构造函数
func NewMultiServerDiscovery(servers []string) *MultiServerDiscovery {
    m := &MultiServerDiscovery{
       servers: servers,
       //用时间戳设定随机数种子，以免每次生成相同随机数序列
       r: rand.New(rand.NewSource(time.Now().UnixNano())),
    }
    //index初始化时随机设定一个值
    m.index = m.r.Intn(math.MaxInt32 - 1)
    return m
}
```

实现接口

```go
// 实现接口(将结构体赋给接口，完成实例化接口的交接仪式)
var _ Discovery = (*MultiServerDiscovery)(nil)

// Refresh 刷新对MultiServerDiscovery没有意义
func (m *MultiServerDiscovery) Refresh() error {
    return nil
}

func (m *MultiServerDiscovery) Update(servers []string) error {
    m.mutex.RLock()
    defer m.mutex.RUnlock()
    m.servers = servers
    return nil
}

func (m *MultiServerDiscovery) Get(mode SelectMode) (string, error) {
    m.mutex.Lock()
    defer m.mutex.Unlock()
    n := len(m.servers)
    if n == 0 {
```

```
        return "",
errors.New("ERR:xclient.discovery.MultiServerDiscovery.Get: no available
servers")
    }
    //根据负载均衡策略，选择合适的服务实例
    switch mode {
    case RandomSelect:
        return m.servers[m.r.Intn(n)], nil
    case RoundRobinSelect:
        s := m.servers[m.index%n] //server可能更新，所以%n一下确保安全
        m.index = (m.index + 1) % n
        return s, nil
    default:
        return "",
errors.New("ERR:xclient.discovery.MultiServerDiscovery.Get: not supported
select mode")
    }
}

func (m *MultiServerDiscovery) GetAll() ([]string, error) {
    m.mutex.Lock()
    defer m.mutex.Unlock()
    servers := make([]string, len(m.servers), len(m.servers))
    copy(servers, m.servers)
    return servers, nil
}
```

2. xclient.go - 负载均衡客户端

负载均衡功能的客户端，面向用户。
同时具备Client的复用和自动关闭的特性：XClient会保存创建成功的Client实例以复用，并提供Close方法在结束后关闭已经建立的连接。

XClient结构体

XClient构造时需要传入：

1. 服务发现实例Discovery
2. 负载均衡策略SelectMode
3. 协议选项

```
type XClient struct {
    discovery Discovery
    mode      SelectMode
    opt       *Option
    mutex     sync.Mutex
    //保存创建好的Client实例，以复用socket
    clients map[string]*Client //key:rpcAddr，val:*Client
}

func NewXClient(discovery Discovery, mode SelectMode, option *Option)
*XClient {
    return &XClient{
        discovery: discovery,
        mode:      mode,
        opt:       option,
        clients:   make(map[string]*Client),
```

```
    }
}
```

Close方法

通过实现io.Closer接口来提供Close方法。从而提供结束后关闭已建立的连接的功能。

```
// 实现io.Closer接口
var _ io.Closer = (*XClient)(nil)

func (x *XClient) Close() error {
    x.mutex.Lock()
    defer x.mutex.Unlock()
    for k, v := range x.clients {
        //关闭客户端
        v.Close()
        delete(x.clients, k)
    }
    return nil
}
```

用户入口（调用服务）

Call()传入的参数和普通的客户端一致。

1. 向Discovery获取合适的服务端地址
2. 调用下文的dial()方法，传入服务端地址，获取合适的客户端
3. 客户端向服务端发起rpc调用

```
func (x *XClient) call(ctx context.Context, rpcAddr string, serviceMethod
string, args, reply interface{}) error {
    client, err := x.dial(rpcAddr)
    if err != nil {
        return err
    }
    //调用client.Call
    return client.Call(ctx, serviceMethod, args, reply)
}

func (x *XClient) Call(ctx context.Context, serviceMethod string, args,
reply interface{}) error {
    rpcAddr, err := x.discovery.Get(x.mode)
    if err != nil {
        return err
    }
    return x.call(ctx, rpcAddr, serviceMethod, args, reply)
}
```

Client复用

检查 `xc.clients` 是否有缓存的 Client，即已经连接了传入的服务地址。

1. 有：检查是否是可用状态，
    1. 可用：返回缓存的 Client。
    2. 不可用：从缓存中删除。
2. 没有：创建新的 Client，缓存并返回。

```go
func (x *XClient) dial(rpcAddr string) (*Client, error) {
    x.mutex.Lock()
    defer x.mutex.Unlock()
    client, ok := x.clients[rpcAddr]
    //检查x.clients是否有缓存的Client，有则检查其可用状态
    if ok && !client.IsAvailable() {
       //不可用，从缓存中删除
       client.Close()
       delete(x.clients, rpcAddr)
       client = nil
    }
    if client == nil {
       var err error
       client, err = XDial(rpcAddr, x.opt)
       if err != nil {
          return nil, err
       }
       x.clients[rpcAddr] = client
    }
    //可用，返回缓存的Client
    return client, nil
}
```

广播

Broadcast 将请求广播到所有的服务实例，如果任意一个实例发生错误，则返回其中一个错误；如果调用成功，则返回其中一个的结果。

有以下几点需要注意：

1. 为了提升性能，请求是并发的。
2. 并发情况下需要使用互斥锁保证 error 和 reply 能被正确赋值。
3. 借助 context.WithCancel 确保有错误发生时，快速失败。

```go
func (x *XClient) Broadcast(ctx context.Context, serviceMethod string, args,
reply interface{}) error {
    servers, err := x.discovery.GetAll()
    if err != nil {
       return err
    }
    var wg sync.WaitGroup
    var mu sync.Mutex
    var e error
    replyDone := reply == nil //如果reply==nil，replyDone=true
    //context.WithCancel 确保有错误发生时，快速失败。
    ctx, _ = context.WithCancel(ctx)
    for _, rpcAddr := range servers {
       wg.Add(1)
       go func(rpcAddr string) {
          defer wg.Done()
          var clonedReply interface{}
          if reply != nil {
             clonedReply =
reflect.New(reflect.ValueOf(reply).Elem().Type()).Interface()
          }
          err := x.call(ctx, rpcAddr, serviceMethod, args, clonedReply)
          mu.Lock()
```

```
            if err != nil && e == nil {
                e = err
                //cancel()
                runtime.Goexit()
            }
            if err == nil && !replyDone {

reflect.ValueOf(reply).Elem().Set(reflect.ValueOf(clonedReply).Elem())
                replyDone = true
            }
            mu.Unlock()
        }(rpcAddr)
    }
    wg.Wait()
    return e
}
```

六、注册中心

注册中心位于客户端和服务端中间。

1. 服务端启动后，将自己注册到注册中心。服务端定期向注册中心发送心跳，证明自己还活着。
2. 客户端调用服务时，向注册中心询问哪些服务可用，注册中心将可用的服务列表返回客户端。
3. 客户端根据注册中心得到的服务列表，选择其中一个发起调用。

注册中心通过心跳机制保证服务可用，通过与负载均衡结合保证性能。

1. WgRegistry - 注册中心

目录：/registry/registry.go

1.1 主要功能

WgRegistry：一个支持心跳保活的简易注册中心
ServerItem：记录服务信息，包括服务地址和启动时间

```
type WgRegistry struct {
    timeout time.Duration
    mutex   sync.Mutex
    servers map[string]*ServerItem //服务器
}

type ServerItem struct {
    Addr  string
    start time.Time
}

const (
    defaultPath    = "/wgrpc/registry"
    defaultTimeout = time.Minute * 5
)

func NewWgRegistry(timeout time.Duration) *WgRegistry {
    return &WgRegistry{
        servers: make(map[string]*ServerItem),
        timeout: timeout,
    }
```

```
}

var DefaultWgRegister = NewWgRegistry(defaultTimeout)
```

添加服务实例

添加服务实例，若服务存在则更新start时间

```
func (w *WgRegistry) putServer(addr string) {
    w.mutex.Lock()
    defer w.mutex.Unlock()
    s := w.servers[addr]
    if s == nil {
       w.servers[addr] = &ServerItem{
          Addr:  addr,
          start: time.Now(),
       }
    } else {
       s.start = time.Now() //存在的话，更新start time
    }
}
```

获取可用的服务列表

```
func (w *WgRegistry) aliveServers() []string {
    w.mutex.Lock()
    defer w.mutex.Unlock()
    var alive []string
    for addr, s := range w.servers {
       if w.timeout == 0 || s.start.Add(w.timeout).After(time.Now()) {
          alive = append(alive, addr)
       } else {
          delete(w.servers, addr)
       }
    }
    sort.Strings(alive)
    return alive
}
```

1.2 通信

采用HTTP协议提供服务，所有信息承载于HTTP Header中

注册服务和获取服务

GET：返回所有可用服务列表
POST：注册服务

```
func (w *WgRegistry) ServeHTTP(rw http.ResponseWriter, req *http.Request) {
    // 为了更简洁，用HTTP协议提供服务，将所有信息承载于HTTP Header
    switch req.Method {
    case "GET":
       rw.Header().Set("X-Wgrpc-Servers", strings.Join(w.aliveServers(),
","))
    case "POST":
       addr := req.Header.Get("X-Wgrpc-Server")
```

```
        if addr == "" {
            rw.WriteHeader(http.StatusInternalServerError)
            return
        }
        w.putServer(addr)
    default:
        rw.WriteHeader(http.StatusMethodNotAllowed)
    }
}
```

打开HTTP服务

```
// HandleHTTP 在registryPath上为WgRegistry注册HTTP处理程序
func (w *WgRegistry) HandleHTTP(registryPath string) {
    http.Handle(registryPath, w)
    log.Println("rpc registry path:", registryPath)
}

//使用默认地址开启
func HandleHTTP() {
    DefaultWgRegister.HandleHTTP(defaultPath)
}
```

心跳

server可以使用此函数向注册中心发送心跳
默认心跳的发送周期逼注册中心设置的过期时间少1min

```
func Heartbeat(registry, addr string, duration time.Duration) {
    if duration == 0 {
        duration = defaultTimeout - time.Duration(1)*time.Minute
    }
    var err error
    err = sendHeartbeat(registry, addr)
    go func() {
        t := time.NewTicker(duration)
        for err == nil {
            <-t.C
            err = sendHeartbeat(registry, addr)
        }
    }()
}
func sendHeartbeat(registry, addr string) error {
    log.Println(addr, " send heart beat to registry", registry)
    httpClient := &http.Client{}
    req, _ := http.NewRequest("POST", registry, nil)
    req.Header.Set("X-Wgrpc-Server", addr)
    if _, err := httpClient.Do(req); err != nil {
        log.Println("ERR: registry.registry.sendHeartbeat: ", err)
        return err
    }
    return nil
}
```

2. WgRegistryDiscovery - 与负载均衡结合

目录：/xclient/WgRegistryDiscovery.go

结构体

WgRegistryDiscovery 嵌套了之前写过的MultiServersDiscovery，很多能力可以复用

```
type WgRegistryDiscovery struct {
    *MultiServerDiscovery
    registry   string        //注册中心地址
    timeout    time.Duration //服务列表的过期时间
    lastUpdate time.Time     //最后从注册中心更新服务列表的时间
}

// 默认十秒过期
const defaultUpdateTimeout = time.Second * 10

func NewWgRegistryDiscovery(registerAddr string, timeout time.Duration)
*WgRegistryDiscovery {
    if timeout == 0 {
        timeout = defaultUpdateTimeout
    }
    d := &WgRegistryDiscovery{
        MultiServerDiscovery: NewMultiServerDiscovery(make([]string, 0)),
        registry:             registerAddr,
        timeout:              timeout,
    }
    return d
}
```

Update和Refresh

超时获取逻辑在Refresh中实现。

```
func (receiver *WgRegistryDiscovery) Update(servers []string) error {
    receiver.mutex.Lock()
    defer receiver.mutex.Unlock()
    receiver.servers = servers
    receiver.lastUpdate = time.Now()
    return nil
}

// Refresh 超时重新获取
func (receiver *WgRegistryDiscovery) Refresh() error {
    receiver.mutex.Lock()
    defer receiver.mutex.Unlock()
    if receiver.lastUpdate.Add(receiver.timeout).After(time.Now()) {
        return nil
    }
    log.Println("ERR: xclient.xclient.Refresh: refresh servers from
registry: ", receiver.registry)
    resp, err := http.Get(receiver.registry)
    if err != nil {
        log.Println("ERR: xclient.xclient.Refresh: refresh err: ", err)
        return err
    }
```

```
    servers := strings.Split(resp.Header.Get("X-Wgrpc-Servers"), ",")
    receiver.servers = make([]string, 0, len(servers))
    for _, server := range servers {
       if strings.TrimSpace(server) != "" {
          receiver.servers = append(receiver.servers,
strings.TrimSpace(server))
       }
    }
    receiver.lastUpdate = time.Now()
    return nil
}
```

Get和GetAll

与MultiServersDiscovery唯一不同的是：需要先调用Refresh确保服务列表没有过期

```
func (receiver *WgRegistryDiscovery) Get(mode SelectMode) (string, error) {
    if err := receiver.Refresh(); err != nil {
       return "", err
    }
    return receiver.MultiServerDiscovery.Get(mode)
}

func (receiver *WgRegistryDiscovery) GetAll() ([]string, error) {
    if err := receiver.Refresh(); err != nil {
       return nil, err
    }
    return receiver.MultiServerDiscovery.GetAll()
}
```

第三章 其他机制

一、超时处理

1. 超时处理的地方

纵观整个远程过程调用：

- 客户端处理超时的地方有：
  - 与服务端建立连接，导致的超时
  - 发送请求到服务端，写报文导致的超时
  - 等待服务端处理时，等待处理导致的超时（比如服务端已挂死，迟迟不响应）
  - 从服务端接收响应时，读报文导致的超时
- 服务端处理超时的地方有：
  - 读取客户端请求报文时，读报文导致的超时
  - 发送响应报文时，写报文导致的超时
  - 调用映射服务的方法时，处理报文导致的超时

WgRPC 在 3 个地方添加了超时处理机制。分别是：

1. 客户端创建连接时
2. 客户端 `Client.Call()` 整个过程导致的超时（包含发送报文，等待处理，接收报文所有阶段）
3. 服务端处理报文，即 `Server.handleRequest` 超时。

2. 超时处理的实现

2.1 创建连接超时

设定超时时间

[[#1. 通信过程]]
在option中，ConnectTimeout和HandleTimeout参数用于设定超时
同时，给了一个默认的超时设置

```
type Option struct {
   MagicNumber    int
   CodecType      codec.Type
   ConnectTimeout time.Duration  //默认为10s
   HandleTimeout  time.Duration  //默认为0s
}
```

检测超时

[[#Dial - 入口]]

1. 在Dial中使用`net.DialTimeout`，传入Option中的ConnectTimeout。如果创建连接超时，则会返回错误
2. 使用协程执行NewClient，通过channel进行超时处理。使用`time.After()`并传入Option中的ConnectTime参数。如果`time.After()`信道先收到消息，说明NewClient执行超时，返回错误。

2.2 Client.Call 超时

[[#Call - 同步接口]]

使用context包实现控制，将控制权交给用户。

用户使用`context.WithTimeout`创建具备超时检测能力的context对象，并传入Client.Call()进行超时控制。

使用select关键字，当ctx.Done()先完成时，则触发超时处理。

2.3 服务端处理超时

[[#处理请求]]

与客户端相似，使用`time.After()`结合`select+chan`完成。

为了确保`sendResponse`仅调用一次，将整个过程拆分为`called`和`sent`两个阶段：

- called信道收到消息，说明没有超时，继续执行sendresponse
- time.After() 先收到消息，说明已经超时，阻塞called和sent，在`case <- time.After(timeout)`处调用`sendResponse`。

二、支持HTTP协议

框架设计之初即支持TCP协议和unix协议，HTTP协议的支持是在TCP协议之上套了一层外壳，用于HTTP的连接。

1. 服务端

[[#5. 支持HTTP协议]]

服务端需要能够处理HTTP请求。而在GO语言中，处理HTTP请求十分简单。
阅读只需要实现标准库中的http包，http.Handle实现如下：

```
package http

func Handle(pattern string, handler Handler) {
DefaultServeMux.Handle(pattern, handler) }
```

包含两个入参：

1. 支持通配的字符串 pattern，在这里，我们固定传入 `/_wgrpc_`
2. Handler 类型，Handler 是一个接口类型，定义如下：

```
type Handler interface {
    ServeHTTP(w ResponseWriter, r *Request)
}
```

也就是说，我们只需要实现接口 Handler 即可作为一个 HTTP Handler 处理 HTTP 请求。
接口 Handler 只定义了一个方法 `ServeHTTP`，实现该方法即可。

在服务端中我们实现了该接口，同时预留了开启HTTP功能的方法HandleHTTP()

2. 客户端

[[#3. 支持HTTP协议]]

客户端只需要向服务端发起HTTP CONNECT请求建立链接。建立链接后其他处理交给NewClient。

主要通过以下三个函数实现

1. `NewHTTPClient()` 函数：创建一个连接HTTP的客户端
2. `DialHTTP()` 函数：连接到指定的地址
3. `XDial()` 函数：一个统一入口。会判断是否是HTTP客户端，如果是则进行HTTP连接，否则TCP连接，或使用unix协议进行socket连接。

三、注册中心和负载均衡

![[Pasted image 20240703193252.png]]

1. 注册中心和负载均衡器相连接，注册中心负责保证服务端的活性，负载均衡器负责为客户端选择合适的服务端
2. 服务端启动后，向注册中心注册自己，同时使用HeatBeat()方法向注册中心发送心跳
3. 客户端需要服务时：
    1. 客户端向负载均衡器发送请求
    2. 负载均衡器从注册中心获取服务列表，然后根据负载均衡策略选出合适的服务端地址发送给客户端。
    3. 客户端获取到服务端地址，根据地址向服务端发送请求

第四章 错误与debug过程

一、client.mutex.Lock()出错

1. 首先，看到锁，以为客户端发生死锁了。调试后发现是指向了空指针，即client = nil
2. main函数找到client创建处，进入上一级函数一步步调试，发现是其中一个函数调用后创建client=nil
3. 该函数抽象后作为参数传入，找到该函数发现是一个验证器，验证是否获得了与服务端的连接（即受到CONNECT消息）
4. 于是开始排除网络问题，进入浏览器” http://localhost:9999/debug/wgrpc “页面发现正常访问，排除服务端网络问题
5. 仔细调试该函数，发现虽然正确连接了，但是还是打印了错误信息，从而导致检验没通过，导致返回值client为空
6. 查看错误信息，发现错误信息显示的是”不符合格式的http响应：connected“，发现是server的问题
7. 进入server代码，查看返回response的函数，发现响应的语句`io.WriteString(conn, "HTTP/1.0"+connected+"\n\n")`，在HTTP/1.0和connected之间少了一个空格，导致connected被判为不符合格式所以报错。
8. 解决问题后，在client调用链路上增加client结构体判空以及错误报告的语句。

二、TCP粘包

1. 在测试多线程并发时，重复测试时有几率卡住不动
2. 仔细调试和重复运行，发现有时服务端会报错：gob格式不正确。于是以为是和错误1一样的错误，仔细查看发送的消息是否正确，发现没有问题。
3. 于是思考：可能是传输过程中发生了问题
4. 查阅相关资料后得证应该是TCP粘包问题，仔细学习相应原理和知识
5. 猜测应该是结构体Option传输时过多地取出字节，导致后面的结构体Header不完整
6. 将Option中的字段类型int指定为int32，问题得到解决。