

Tao - Go语言实现的TCP网络编程框架



(/apps/download?utm_source=stc) + 关注

2016.05.09 16:14* 字数 4966 阅读 7556 评论 31 喜欢 49 赞赏 2 阅读 7556 评论 31 喜欢 49 赞赏 2

一. 什么是Tao

Tao (<https://github.com/leesper/tao>), 在英文中的意思是“The ultimate principle of universe”, 即“道”, 它是宇宙的终极奥义。

“道生一，一生二，二生三，三生无穷。” —— 《道德经》

Tao同时也是我用Go语言开发的一个异步的TCP服务器框架（TCP Asynchronous Go server Framework），秉承Go语言“Less is more”的极简主义哲学，它能穿透一切表象，带你一窥网络编程的世界，让你从此彻底摆脱只会写“socket-bind-listen-accept”的窘境。本文将简单讨论一下这个框架的设计思路以及自己的一些思考。

1. Tao解决什么问题

1.1 场景

你开发的产品有一套特有的业务逻辑，要通过互联网得到服务端的支持才能为你的客户提供服务。

1.2 问题

怎样快速稳定地实现产品的功能，而不需要耗费大量的时间处理各种底层的网络通信细节。

1.3 解决方案

Tao提供了一种用框架支撑业务逻辑的机制。你只需要与客户端定义好消息格式，然后将对应的业务逻辑编写成函数注册到框架中就可以了。

2. 50行启动一个聊天服务器

让我们举一个例子来看看如何使用Tao框架实现一个简单的群聊天服务器。服务器端代码可以这么写：



下载简书App
创作你的创作

(/apps/download?utm_source=stc)



```

package main

import (
    "fmt"
    "net"

    "github.com/leesper/holmes"
    "github.com/leesper/tao"
    "github.com/leesper/tao/examples/chat"
)

// ChatServer is the chatting server.
type ChatServer struct {
    *tao.Server
}

// NewChatServer returns a ChatServer.
func NewChatServer() *ChatServer {
    onConnectOption := tao.OnConnectOption(func(conn tao.WriteCloser) bool {
        holmes.Infof("on connect")
        return true
    })
    onErrorOption := tao.OnErrorOption(func(conn tao.WriteCloser) {
        holmes.Infof("on error")
    })
    onCloseOption := tao.OnCloseOption(func(conn tao.WriteCloser) {
        holmes.Infof("close chat client")
    })
    return &ChatServer{
        tao.NewServer(onConnectOption, onErrorOption, onCloseOption),
    }
}

func main() {
    defer holmes.Start().Stop()

    tao.Register(chat.ChatMessage, chat.DeserializeMessage, chat.ProcessMessage)

    l, err := net.Listen("tcp", fmt.Sprintf("%s:%d", "0.0.0.0", 12345))
    if err != nil {
        holmes.Fatalf("listen error", err)
    }
    chatServer := NewChatServer()
    err = chatServer.Start(l)
    if err != nil {
        holmes.Fatalf("start error", err)
    }
}

```

启动一个服务器只需要三步就能完成。首先注册消息和业务逻辑回调，其次填入IP地址和端口，最后Start一下就可以了。这时候客户端就能够发起连接，并开始聊天。业务逻辑的实现很简单，遍历所有的连接，然后发送数据：

```

// ProcessMessage handles the Message logic.
func ProcessMessage(ctx context.Context, conn tao.WriteCloser) {
    holmes.Infof("ProcessMessage")
    s, ok := tao.ServerFromContext(ctx)
    if ok {
        msg := tao.MessageFromContext(ctx)
        s.Broadcast(msg)
    }
}

```

3. Go语言的编程哲学

Go语言是“云计算时代的C语言”，适用于开发基础性服务，比如服务器。它语法类似C语言且标准库丰富，上手较快，所以开发效率高；编译速度快，运行效率接近C，所以运行效率高。

3.1 面向对象编程

Go语言面向对象编程的风格是“多用组合，少用继承”，以匿名嵌入的方式实现继承。比如上面的聊天服务器ChatServer：

```
// ChatServer is the chatting server.
type ChatServer struct {
    *tao.Server
}
```

于是ChatServer就自动继承了Server所有的属性和方法。当然，这里是以指针的方式嵌入的。

3.2 面向接口编程

Go语言的面向接口编程是“鸭子类型”的，即“如果我走起来像鸭子，叫起来像鸭子，那么我就是一只鸭子”。其他的编程语言需要显式地说明自己继承某个接口，Go语言却采取的是“隐式声明”的方式。比如Tao框架使用的多线程日志库Holmes (<https://github.com/leesper/holmes>)实现“每小时创建一个新日志文件”功能的核心代码如下：

```
func (ls *logSegment)Write(p []byte) (n int, err error) {
    if ls.timeToCreate != nil && ls.logFile != os.Stdout && ls.logFile != os.Stderr {
        {
            select {
            case current := <-ls.timeToCreate:
                ls.logFile.Close()
                ls.logFile = nil
                name := getLogFileName(current)
                ls.logFile, err = os.Create(path.Join(ls.logPath, name))
                if err != nil {
                    fmt.Fprintln(os.Stderr, err)
                    ls.logFile = os.Stderr
                } else {
                    next := current.Truncate(ls.unit).Add(ls.unit)
                    ls.timeToCreate = time.After(next.Sub(time.Now()))
                }
            default:
                // do nothing
            }
        }
    }
    return ls.logFile.Write(p)
}
```

而标准库中的io.Writer定义如下，那么这里的logSegment就实现了io.Writer的接口，所有以io.Writer作为形参的函数，我都可以传一个logSegment的实参进去。

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

3.3 一个中心，两个基本点

掌握Go语言，要把握“一个中心，两个基本点”。“一个中心”是Go语言并发模型，即“**不要通过共享内存来通信，要通过通信来共享内存**”；“两个基本点”是Go语言的并发模型的两块基石：channel和go-routine。理解了它们就能看懂大部分代码。下面让我们正式开始介绍Tao框架吧。

二. Tao的设计思路

1. 服务器的启动

Tao框架支持通过tao.TLSCredsOption()函数提供传输层安全的TLS Server。服务器的核心职责是“监听并接受客户端连接”。每个进程能够打开的文件描述符是有限制的，所以它还需要限制最大并发连接数，关键代码如下：

```
// Start starts the TCP server, accepting new clients and creating service
// go-routine for each. The service go-routines read messages and then call
// the registered handlers to handle them. Start returns when failed with fatal
// errors, the listener will be closed when returned.
func (s *Server) Start(l net.Listener) error {
```

```

s.mu.Lock()
if s.lis == nil {
    s.mu.Unlock()
    l.Close()
    return ErrServerClosed
}
s.lis[l] = true
s.mu.Unlock()

defer func() {
    s.mu.Lock()
    if s.lis != nil && s.lis[l] {
        l.Close()
        delete(s.lis, l)
    }
    s.mu.Unlock()
}()

    holmes.Infof("server start, net %s addr %s\n", l.Addr().Network(), l.Addr().S
tring())

s.wg.Add(1)
go s.timeOutLoop()

var tempDelay time.Duration
for {
    rawConn, err := l.Accept()
    if err != nil {
        if ne, ok := err.(net.Error); ok && ne.Temporary() {
            if tempDelay == 0 {
                tempDelay = 5 * time.Millisecond
            } else {
                tempDelay *= 2
            }
            if max := 1 * time.Second; tempDelay >= max {
                tempDelay = max
            }
            holmes.Errorf("accept error %v, retrying in %d\n", err, tempDelay
)

            select {
            case <-time.After(tempDelay):
            case <-s.ctx.Done():
            }
            continue
        }
        return err
    }
    tempDelay = 0

    // how many connections do we have ?
    sz := s.conns.Size()
    if sz >= MaxConnections {
        holmes.Warnf("max connections size %d, refuse\n", sz)
        rawConn.Close()
        continue
    }

    if s.opts.tlsCfg != nil {
        rawConn = tls.Server(rawConn, s.opts.tlsCfg)
    }

    netid := netIdentifier.GetAndIncrement()
    sc := NewServerConn(netid, s, rawConn)
    sc.SetName(sc.rawConn.RemoteAddr().String())

    s.mu.Lock()
    if s.sched != nil {
        sc.RunEvery(s.interv, s.sched)
    }
    s.mu.Unlock()

    s.conns.Put(netid, sc)
    addTotalConn(1)

    s.wg.Add(1)
    go func() {
        sc.Start()
    }()

    holmes.Infof("accepted client %s, id %d, total %d\n", sc.GetName(), netid
, s.conns.Size())
    s.conns.RLock()
    for _, c := range s.conns.m {
        holmes.Infof("client %s\n", c.GetName())
    }
    s.conns.RUnlock()

```

```
    } // for loop
}
```

如果服务器在接受客户端连接请求的时候发生了临时错误，那么服务器将等待最多1秒的时间再重新尝试接受请求，如果现有的连接数超过了MaxConnections（默认1000），就拒绝并关闭连接，否则启动一个新的连接开始工作。

2. 服务器的优雅关闭

Go语言在发布1.7版时在标准库中引入了context包。context包提供的Context结构能够在服务器，网络连接以及各相关线程之间建立一种相关联的“上下文”关系。这种上下文关系包含的信息是与某次网络请求有关的（request scoped），因此与该请求有关的所有Go线程都能安全地访问这个上下文结构，读取或者写入与上下文有关的数据。比如handleLoop线程会将某个网络连接的net ID以及message打包到上下文结构中，然后连同handler函数一起交给工作者线程去处理：

```
// handleLoop() - put handler or timeout callback into worker go-routines
func handleLoop(c WriteCloser, wg *sync.WaitGroup) {
    //... omitted ...

    for {
        select {
            //... omitted ...
            case msgHandler := <-handlerCh:
                msg, handler := msgHandler.message, msgHandler.handler
                if handler != nil {
                    if askForWorker {
                        WorkerPoolInstance().Put(netID, func() {
                            handler(NewContextWithNetID(NewContextWithMessage(ctx, msg), netID), c)
                        })
                    }
                }
                //... omitted ...
        }
    }
}
```

随后，在工作者线程真正执行时，业务逻辑代码就能在handler函数中获取到message或者net ID，这些都是与本次请求有关的上下文数据，比如一个典型的echo server就会这样处理：

```
// ProcessMessage process the logic of echo message.
func ProcessMessage(ctx context.Context, conn tao.WriteCloser) {
    msg := tao.MessageFromContext(ctx).(Message)
    holmes.Infof("receving message %s\n", msg.Content)
    conn.Write(msg)
}
```

使用context的另外一个场景是实现服务器及网络连接的“优雅关闭”。服务器在管理网络连接时会将自己的上下文传递给它，而网络连接启动新线程时同样也会将自己的上下文传递给这些线程，这些上下文都是可取消（cancelable）的。当服务器需要停机或者连接将要关闭时，只要调用cancel函数，所有这些线程就能收到通知并退出。服务器或者网络连接通过阻塞等待这些线程关闭之后再关闭，就能最大限度保证正确退出。服务器关闭的关键代码如下：

```
// Stop gracefully closes the server, it blocked until all connections
// are closed and all go-routines are exited.
func (s *Server) Stop() {
    // immediately stop accepting new clients
    s.mu.Lock()
    listeners := s.lis
    s.lis = nil
    s.mu.Unlock()

    for l := range listeners {
        l.Close()
        holmes.Infof("stop accepting at address %s\n", l.Addr().String())
    }

    // close all connections
    conns := map[int64]*ServerConn{}
    s.conns.RLock()
    for k, v := range s.conns.m {
        conns[k] = v
    }
    s.conns.Clear()
    s.conns.RUnlock()

    for _, c := range conns {
        c.rawConn.Close()
        holmes.Infof("close client %s\n", c.GetName())
    }

    s.mu.Lock()
    s.cancel()
    s.mu.Unlock()

    s.wg.Wait()

    holmes.Infoln("server stopped gracefully, bye.")
    os.Exit(0)
}
```

3. 网络连接模型

在其他的编程语言中，采用Reactor模式编写的服务器往往需要在一个IO线程异步地通过epoll进行多路复用。而因为Go线程的开销廉价，Go语言可以对每一个网络连接创建三个go-routine。readLoop()负责读取数据并反序列化成消息；writeLoop()负责序列化消息并发送二进制字节流；最后handleLoop()负责调用消息处理函数。这三个协程在连接创建并启动时就会各自独立运行：

```
// Start starts the server connection, creating go-routines for reading,
// writing and handling.
func (sc *ServerConn) Start() {
    holmes.Infof("conn start, <%v -> %v>\n", sc.rawConn.LocalAddr(), sc.rawConn.RemoteAddr())
    onConnect := sc.belong.opts.onConnect
    if onConnect != nil {
        onConnect(sc)
    }

    loopers := []func(WriteCloser, *sync.WaitGroup){readLoop, writeLoop, handleLoop}
    for _, l := range loopers {
        looper := l
        sc.wg.Add(1)
        go looper(sc, sc.wg)
    }
}
```

3.1 核心代码分析之readLoop

readLoop做了三件关键的工作。首先调用消息编解码器将接收到的字节流反序列化成消息；然后更新用于心跳检测的时间戳；最后，根据消息的协议号找到对应的消息处理函数，如果注册了消息回调函数，那么就调用该函数处理消息，否则将消息和处理函数打包发送到handlerCh中，注意其中的cDone和sDone分别是网络连接和服务器的上下文结构中的channel，分别用于监听网络连接和服务器的“关闭”事件通知（下同）。

```

/* readLoop() blocking read from connection, deserialize bytes into message,
then find corresponding handler, put it into channel */
func readLoop(c WriteCloser, wg *sync.WaitGroup) {
    var (
        rawConn      net.Conn
        codec          Codec
        cDone          <-chan struct{}
        sDone          <-chan struct{}
        setHeartBeatFunc func(int64)
        onMessage      onMessageFunc
        handlerCh      chan MessageHandler
        msg            Message
        err            error
    )

    switch c := c.(type) {
    case *ServerConn:
        rawConn = c.rawConn
        codec = c.belong.opts.codec
        cDone = c.ctx.Done()
        sDone = c.belong.ctx.Done()
        setHeartBeatFunc = c.SetHeartBeat
        onMessage = c.belong.opts.onMessage
        handlerCh = c.handlerCh
    case *ClientConn:
        rawConn = c.rawConn
        codec = c.opts.codec
        cDone = c.ctx.Done()
        sDone = nil
        setHeartBeatFunc = c.SetHeartBeat
        onMessage = c.opts.onMessage
        handlerCh = c.handlerCh
    }

    defer func() {
        if p := recover(); p != nil {
            holmes.Errorf("panics: %v\n", p)
        }
        wg.Done()
        holmes.Debugln("readLoop go-routine exited")
        c.Close()
    }()

    for {
        select {
        case <-cDone: // connection closed
            holmes.Debugln("receiving cancel signal from conn")
            return
        case <-sDone: // server closed
            holmes.Debugln("receiving cancel signal from server")
            return
        default:
            msg, err = codec.Decode(rawConn)
            if err != nil {
                holmes.Errorf("error decoding message %v\n", err)
                if _, ok := err.(ErrUndefined); ok {
                    // update heart beats
                    setHeartBeatFunc(time.Now().UnixNano())
                    continue
                }
                return
            }
            setHeartBeatFunc(time.Now().UnixNano())
            handler := GetHandlerFunc(msg.MessageNumber())
            if handler == nil {
                if onMessage != nil {
                    holmes.Infof("message %d call onMessage()\n", msg.MessageNumber())
                    onMessage(msg, c.(WriteCloser))
                } else {
                    holmes.Warnf("no handler or onMessage() found for message %d\n", msg.MessageNumber())
                }
                continue
            }
            handlerCh <- MessageHandler{msg, handler}
        }
    }
}

```

3.2 核心代码分析之writeLoop

writeLoop做了一件事情，从sendCh中读取已序列化好的字节流，然后发送到网络上。但是要注意，该协程在连接关闭退出执行之前，会非阻塞地将sendCh中的消息全部发送完毕再退出，**避免漏发消息**，这就是关键所在。

```
/* writeLoop() receive message from channel, serialize it into bytes,
then blocking write into connection */
func writeLoop(c WriteCloser, wg *sync.WaitGroup) {
    var (
        rawConn net.Conn
        sendCh   chan []byte
        cDone    <-chan struct{}
        sDone    <-chan struct{}
        pkt     []byte
        err     error
    )

    switch c := c.(type) {
    case *ServerConn:
        rawConn = c.rawConn
        sendCh = c.sendCh
        cDone = c.ctx.Done()
        sDone = c.belong.ctx.Done()
    case *ClientConn:
        rawConn = c.rawConn
        sendCh = c.sendCh
        cDone = c.ctx.Done()
        sDone = nil
    }

    defer func() {
        if p := recover(); p != nil {
            holmes.Errorf("panics: %v\n", p)
        }
        // drain all pending messages before exit
    OuterFor:
        for {
            select {
            case pkt = <-sendCh:
                if pkt != nil {
                    if _, err = rawConn.Write(pkt); err != nil {
                        holmes.Errorf("error writing data %v\n", err)
                    }
                }
            default:
                break OuterFor
            }
        }
        wg.Done()
        holmes.Debugln("writeLoop go-routine exited")
        c.Close()
    }()

    for {
        select {
        case <-cDone: // connection closed
            holmes.Debugln("receiving cancel signal from conn")
            return
        case <-sDone: // server closed
            holmes.Debugln("receiving cancel signal from server")
            return
        case pkt = <-sendCh:
            if pkt != nil {
                if _, err = rawConn.Write(pkt); err != nil {
                    holmes.Errorf("error writing data %v\n", err)
                    return
                }
            }
        }
    }
}
```

3.3 核心代码分析之handleLoop

readLoop将消息和处理函数打包发给了handlerCh，于是handleLoop就从handlerCh中取出消息和处理函数，然后交给工作者线程池，由后者负责调度执行，完成对消息的处理。这里很好的诠释了Go语言是如何通过channel实现Go线程间通信的。


```

// handleLoop() - put handler or timeout callback into worker go-routines
func handleLoop(c WriteCloser, wg *sync.WaitGroup) {
    var (
        cDone      <-chan struct{}
        sDone      <-chan struct{}
        timerCh     chan *OnTimeOut
        handlerCh   chan MessageHandler
        netID       int64
        ctx         context.Context
        askForWorker bool
    )

    switch c := c.(type) {
    case *ServerConn:
        cDone = c.ctx.Done()
        sDone = c.belong.ctx.Done()
        timerCh = c.timerCh
        handlerCh = c.handlerCh
        netID = c.netid
        ctx = c.ctx
        askForWorker = true
    case *ClientConn:
        cDone = c.ctx.Done()
        sDone = nil
        timerCh = c.timing.timeOutChan
        handlerCh = c.handlerCh
        netID = c.netid
        ctx = c.ctx
    }

    defer func() {
        if p := recover(); p != nil {
            holmes.Errorf("panics: %v\n", p)
        }
        wg.Done()
        holmes.Debugln("handleLoop go-routine exited")
        c.Close()
    }()

    for {
        select {
        case <-cDone: // connectin closed
            holmes.Debugln("receiving cancel signal from conn")
            return
        case <-sDone: // server closed
            holmes.Debugln("receiving cancel signal from server")
            return
        case msgHandler := <-handlerCh:
            msg, handler := msgHandler.message, msgHandler.handler
            if handler != nil {
                if askForWorker {
                    WorkerPoolInstance().Put(netID, func() {
                        handler(NewContextWithNetID(NewContextWithMessage(ctx, msg), netID), c)
                    })
                    addTotalHandle()
                } else {
                    handler(NewContextWithNetID(NewContextWithMessage(ctx, msg), netID), c)
                }
            }
        case timeout := <-timerCh:
            if timeout != nil {
                timeoutNetID := NetIDFromContext(timeout.Ctx)
                if timeoutNetID != netID {
                    holmes.Errorf("timeout net %d, conn net %d, mismatched!\n", timeoutNetID, netID)
                }
                if askForWorker {
                    WorkerPoolInstance().Put(netID, func() {
                        timeout.Callback(time.Now(), c.(WriteCloser))
                    })
                } else {
                    timeout.Callback(time.Now(), c.(WriteCloser))
                }
            }
        }
    }
}

```

4. 消息处理机制

4.1 消息上下文

任何一个实现了Message接口的类型，都是一个消息，它需要提供方法访问自己的协议号并将自己序列化成字节数组；另外，每个消息都需要注册自己的反序列化函数和处理函数：

```
// Handler takes the responsibility to handle incoming messages.
type Handler interface {
    Handle(context.Context, interface{})
}

// HandlerFunc serves as an adapter to allow the use of ordinary functions as handlers.
type HandlerFunc func(context.Context, WriteCloser)

// Handle calls f(ctx, c)
func (f HandlerFunc) Handle(ctx context.Context, c WriteCloser) {
    f(ctx, c)
}

// UnmarshalFunc unmarshals bytes into Message.
type UnmarshalFunc func([]byte) (Message, error)

// handlerUnmarshaler is a combination of unmarshal and handle functions for message.
type handlerUnmarshaler struct {
    handler      HandlerFunc
    unmarshaler UnmarshalFunc
}

func init() {
    messageRegistry = map[int32]messageFunc{}
    buf = new(bytes.Buffer)
}

// Register registers the unmarshal and handle functions for msgType.
// If no unmarshal function provided, the message will not be parsed.
// If no handler function provided, the message will not be handled unless you
// set a default one by calling SetOnMessageCallback.
// If Register being called twice on one msgType, it will panics.
func Register(msgType int32, unmarshaler func([]byte) (Message, error), handler f
unc(context.Context, WriteCloser)) {
    if _, ok := messageRegistry[msgType]; ok {
        panic(fmt.Sprintf("trying to register message %d twice", msgType))
    }

    messageRegistry[msgType] = handlerUnmarshaler{
        unmarshaler: unmarshaler,
        handler:      HandlerFunc(handler),
    }
}

// GetUnmarshalFunc returns the corresponding unmarshal function for msgType.
func GetUnmarshalFunc(msgType int32) UnmarshalFunc {
    entry, ok := messageRegistry[msgType]
    if !ok {
        return nil
    }
    return entry.unmarshaler
}

// GetHandlerFunc returns the corresponding handler function for msgType.
func GetHandlerFunc(msgType int32) HandlerFunc {
    entry, ok := messageRegistry[msgType]
    if !ok {
        return nil
    }
    return entry.handler
}

// Message represents the structured data that can be handled.
type Message interface {
    MessageNumber() int32
    Serialize() ([]byte, error)
}
```

对每个消息处理函数而言，要处理的消息以及发送该消息的客户端都是不同的，这些信息被称为“消息上下文”，用Context结构表示，每个不同的客户端用一个64位整数netid标识：

```

// Context is the context info for every handler function.
// Handler function handles the business logic about message.
// We can find the client connection who sent this message by netid and send back
responses.
type Context struct{
    message Message
    netid int64
}

func NewContext(msg Message, id int64) Context {
    return Context{
        message: msg,
        netid: id,
    }
}

func (ctx Context)Message() Message {
    return ctx.message
}

func (ctx Context)Id() int64 {
    return ctx.netid
}

```

4.2 编解码器

接收数据时，编解码器（Codec）负责按照一定的格式将网络连接上读取的字节数据反序列化成消息，并将消息交给上层处理（解码）；发送数据时，编解码器将上层传递过来的消息序列化成字节数据，交给下层发送（编码）：

```

// Codec is the interface for message coder and decoder.
// Application programmer can define a custom codec themselves.
type Codec interface {
    Decode(Connection) (Message, error)
    Encode(Message) ([]byte, error)
}

```

Tao框架采用的是“Type-Length-Data”的格式打包数据。Type占4个字节，表示协议类型；Length占4个字节，表示消息长度，Data为变长字节序列，长度由Length表示。反序列化时，由Type字段可以确定协议类型，然后截取Length长度的字节数据Data，并调用已注册的反序列化函数处理。核心代码如下：

```

// Codec is the interface for message coder and decoder.
// Application programmer can define a custom codec themselves.
type Codec interface {
    Decode(net.Conn) (Message, error)
    Encode(Message) ([]byte, error)
}

// TypeLengthValueCodec defines a special codec.
// Format: type-length-value |4 bytes|4 bytes|n bytes <= 8M|
type TypeLengthValueCodec struct{}

// Decode decodes the bytes data into Message
func (codec TypeLengthValueCodec) Decode(raw net.Conn) (Message, error) {
    byteChan := make(chan []byte)
    errorChan := make(chan error)

    go func(bc chan []byte, ec chan error) {
        typeData := make([]byte, MessageTypeBytes)
        _, err := io.ReadFull(raw, typeData)
        if err != nil {
            ec <- err
            close(bc)
            close(ec)
            holmes.Debugln("go-routine read message type exited")
            return
        }
        bc <- typeData
    }(byteChan, errorChan)

    var typeBytes []byte

    select {
    case err := <-errorChan:
        return nil, err

    case typeBytes = <-byteChan:
        if typeBytes == nil {
            holmes.Warnln("read type bytes nil")
            return nil, ErrBadData
        }
        typeBuf := bytes.NewReader(typeBytes)
        var msgType int32
        if err := binary.Read(typeBuf, binary.LittleEndian, &msgType); err != nil
    {
        return nil, err
    }

    lengthBytes := make([]byte, MessageLenBytes)
    _, err := io.ReadFull(raw, lengthBytes)
    if err != nil {
        return nil, err
    }
    lengthBuf := bytes.NewReader(lengthBytes)
    var msgLen uint32
    if err = binary.Read(lengthBuf, binary.LittleEndian, &msgLen); err != nil
    {
        return nil, err
    }
    if msgLen > MessageMaxBytes {
        holmes.Errorf("message(type %d) has bytes(%d) beyond max %d\n", msgTy
pe, msgLen, MessageMaxBytes)
        return nil, ErrBadData
    }

    // read application data
    msgBytes := make([]byte, msgLen)
    _, err = io.ReadFull(raw, msgBytes)
    if err != nil {
        return nil, err
    }

    // deserialize message from bytes
    unmarshaler := GetUnmarshalFunc(msgType)
    if unmarshaler == nil {
        return nil, ErrUndefined(msgType)
    }
    return unmarshaler(msgBytes)
    }
}

```

这里的代码存在一些微妙的设计，需要仔细解释一下。

TypeLengthValueCodec.Decode()函数会被readLoop协程用到。因为io.ReadFull()是同步调用，没有数据可读时会阻塞readLoop协程。此时如果关闭网络连接，readLoop协程

将无法退出。所以这里的代码用到了一个小技巧：专门开辟了一个新协程来等待读取最开始的4字节Type数据，然后自己select阻塞在多个channel上，这样就不会忽略其他channel传递过来的消息。一旦成功读取到Type数据，就继续后面的流程：读取Length数据，根据Length读取应用数据交给先前注册好的反序列化函数。注意，如果收到超过最大长度的数据就会关闭连接，这是为了防止外部程序恶意消耗系统资源。

5. 工作者协程池

为了提高框架的健壮性，避免因为处理业务逻辑造成的响应延迟，消息处理函数一般都会被调度到工作者协程池执行。设计工作者协程池的一个关键是如何将任务散列给池子中的不同协程。一方面，要避免并发问题，必须保证同一个网络连接发来的消息都被散列到同一个协程按顺序执行；另一方面，散列一定要是均匀的，不能让协程“忙的忙死，闲的闲死”。关键还是在散列函数的设计上。

5.1 核心代码分析

协程池是按照单例模式设计的。创建时会调用newWorker()创建一系列worker协程。

```
// WorkerPool is a pool of go-routines running functions.
type WorkerPool struct {
    workers    []*worker
    closeChan chan struct{}}

var (
    globalWorkerPool *WorkerPool
)

func init() {
    globalWorkerPool = newWorkerPool(WorkersNum)
}

// WorkerPoolInstance returns the global pool.
func WorkerPoolInstance() *WorkerPool {
    return globalWorkerPool
}

func newWorkerPool(vol int) *WorkerPool {
    if vol <= 0 {
        vol = WorkersNum
    }

    pool := &WorkerPool{
        workers:    make([]*worker, vol),
        closeChan: make(chan struct{})},
    }

    for i := range pool.workers {
        pool.workers[i] = newWorker(i, 1024, pool.closeChan)
        if pool.workers[i] == nil {
            panic("worker nil")
        }
    }

    return pool
}
```

5.2 给工作者协程分配任务

给工作者协程分配任务的方式很简单，通过hashCode()散列函数找到对应的worker协程，然后把回调函数发送到对应协程的channel中。对应协程在运行时就会从channel中取出然后执行，在start()函数中。

```
// Put appends a function to some worker's channel.
func (wp *WorkerPool) Put(k interface{}, cb func()) error {
    code := hashCode(k)
    return wp.workers[code&uint32(len(wp.workers)-1)].put(workerFunc(cb))
}

func (w *worker) start() {
    for {
        select {
        case <-w.closeChan:
            return
        case cb := <-w.callbackChan:
            before := time.Now()
            cb()
            addTotalTime(time.Since(before).Seconds())
        }
    }
}

func (w *worker) put(cb workerFunc) error {
    select {
    case w.callbackChan <- cb:
        return nil
    default:
        return ErrWouldBlock
    }
}
}
```

6. 线程安全的定时器

Tao框架设计了一个定时器TimingWheel，用来控制定时任务。Connection在此基础上进行了进一步封装。提供定时执行（RunAt），延时执行（RunAfter）和周期执行（RunEvery）功能。这里通过定时器的设计引出多线程编程的一点经验之谈。

6.1 定时任务的数据结构设计

6.1.1 定时任务结构

每个定时任务由一个timerType表示，它带有自己的id和包含定时回调函数的结构OnTimeOut。expiration表示该任务到期要被执行的时间，interval表示时间间隔，interval > 0意味着该任务是会被周期性重复执行的任务。

```
/* 'expiration' is the time when timer time out, if 'interval' > 0
the timer will time out periodically, 'timeout' contains the callback
to be called when times out */
type timerType struct {
    id          int64
    expiration  time.Time
    interval    time.Duration
    timeout     *OnTimeOut
    index       int // for container/heap
}

// OnTimeOut represents a timed task.
type OnTimeOut struct {
    Callback func(time.Time, WriteCloser)
    Ctx      context.Context
}

// NewOnTimeOut returns OnTimeOut.
func NewOnTimeOut(ctx context.Context, cb func(time.Time, WriteCloser)) *OnTimeOut {
    return &OnTimeOut{
        Callback: cb,
        Ctx:      ctx,
    }
}
```

6.1.2 定时任务的组织

定时器需要按照到期时间的顺序从最近到最远排列，这是一个天然的小顶堆，于是这里采用标准库container/heap创建了一个堆数据结构来组织定时任务，存取效率达到O(nlogn)。

```
// timerHeap is a heap-based priority queue
type timerHeapType []*timerType

func (heap timerHeapType) getIndexByID(id int64) int {
    for _, t := range heap {
        if t.id == id {
            return t.index
        }
    }
    return -1
}

func (heap timerHeapType) Len() int {
    return len(heap)
}

func (heap timerHeapType) Less(i, j int) bool {
    return heap[i].expiration.UnixNano() < heap[j].expiration.UnixNano()
}

func (heap timerHeapType) Swap(i, j int) {
    heap[i], heap[j] = heap[j], heap[i]
    heap[i].index = i
    heap[j].index = j
}

func (heap *timerHeapType) Push(x interface{}) {
    n := len(*heap)
    timer := x.(*timerType)
    timer.index = n
    *heap = append(*heap, timer)
}

func (heap *timerHeapType) Pop() interface{} {
    old := *heap
    n := len(old)
    timer := old[n-1]
    timer.index = -1
    *heap = old[0 : n-1]
    return timer
}
```

6.2 定时器核心代码分析

TimingWheel在创建时会启动一个单独协程来运行定时器核心代码start()。它在多个channel上进行多路复用操作：如果从cancelChan收到timerId，就执行取消操作：从堆上删除对应的定时任务；将定时任务数量发送给sizeChan，别的线程就能获取当前定时任务数；如果从quitChan收到消息，定时器就会被关闭然后退出；如果从addChan收到timer，就将该定时任务添加到堆；如果从tw.ticker.C收到定时信号，就调用getExpired()函数获取到期的任务，然后将这些任务回调发送到TimeOutChannel中，其他相关线程会通过该channel获取并执行定时回调。最后tw.update()会更新周期性执行的定时任务，重新调度执行。

```

func (tw *TimingWheel) update(timers []*timerType) {
    if timers != nil {
        for _, t := range timers {
            if t.isRepeat() {
                t.expiration = t.expiration.Add(t.interval)
                heap.Push(&tw.timers, t)
            }
        }
    }
}

func (tw *TimingWheel) start() {
    for {
        select {
        case timerID := <-tw.cancelChan:
            index := tw.timers.getIndexByID(timerID)
            if index >= 0 {
                heap.Remove(&tw.timers, index)
            }

        case tw.sizeChan <- tw.timers.Len():

        case <-tw.ctx.Done():
            tw.ticker.Stop()
            return

        case timer := <-tw.addChan:
            heap.Push(&tw.timers, timer)

        case <-tw.ticker.C:
            timers := tw.getExpired()
            for _, t := range timers {
                tw.GetTimeOutChannel() <- t.timeout
            }
            tw.update(timers)
        }
    }
}

```

6.3 定时器是怎么做到线程安全的

用Tao框架开发的服务器一开始总是时不时地崩溃。有时候运行了几个小时服务器就突然退出了。查看打印出来的调用栈发现。每次程序都在定时器上崩溃，原因是数组访问越界。这就是并发访问导致的问题，为什么呢？因为定时器的核心函数在一个协程中操作堆数据结构，与此同时其提供的添加，删除等接口却有可能在其他协程中调用。多个协程并发访问一个没有加锁的数据结构，必然会出现问题。解决方法很简单：将多个协程的并发访问转化为单个协程的串行访问，也就是将添加，删除等操作发送给不同的channel，然后在start()协程中统一处理：

```

// AddTimer adds new timed task.
func (tw *TimingWheel) AddTimer(when time.Time, interv time.Duration, to *OnTimeOut) int64 {
    if to == nil {
        return int64(-1)
    }
    timer := newTimer(when, interv, to)
    tw.addChan <- timer
    return timer.id
}

// Size returns the number of timed tasks.
func (tw *TimingWheel) Size() int {
    return <-tw.sizeChan
}

// CancelTimer cancels a timed task with specified timer ID.
func (tw *TimingWheel) CancelTimer(timerID int64) {
    tw.cancelChan <- timerID
}

```

6.4 应用层心跳

陈硕在他的《Linux多线程服务端编程》一书中说到，维护长连接的服务器都应该在应用层自己实现心跳消息：

“在严肃的网络程序中，应用层的心跳协议是必不可少的。应该用心跳消息来判断对方进程是否能正常工作。”

要使用一个连接来同时发送心跳和其他业务消息，这样一旦应用层因为出错发不出消息，对方就能够立刻通过心跳停止感知到。值得注意的是，在Tao框架中，定时器只有一个，而客户端连接可能会有很多个。在长连接模式下，每个客户端都需要处理心跳包，或者其他类型的定时任务。将框架设计为“每个客户端连接自带一个定时器”是不合适的——有十万个连接就有十万个定时器，会有较高的CPU占用率。定时器应该只有一个，所有客户端注册进来的定时任务都由它负责处理。但是如果所有的客户端连接都等待唯一一个定时器发来的消息，就又会存在并发问题。比如client 1的定时任务到期了，但它现在正忙着处理其他消息，这个定时任务就可能被其他client执行。所以这里采取了一种“先集中后分散”的处理机制：每一个定时任务都由一个TimeOut结构表示，该结构中除了回调函数还包含一个context。客户端启动定时任务的时候都会填入net ID。TCPServer统一接收定时任务，然后从定时任务中取出net ID，然后将该定时任务交给相应的ServerConn或ClientConn去执行：

```
// Retrieve the extra data(i.e. net id), and then redispach timeout callbacks
// to corresponding client connection, this prevents one client from running
// callbacks of other clients
func (s *Server) timeOutLoop() {
    defer s.wg.Done()

    for {
        select {
        case <-s.ctx.Done():
            return

        case timeout := <-s.timing.GetTimeOutChannel():
            netID := timeout.Ctx.Value(netIDCtx).(int64)
            if sc, ok := s.conns.Get(netID); ok {
                sc.timerCh <- timeout
            } else {
                holmes.Warnf("invalid client %d\n", netID)
            }
        }
    }
}
```

三. 也谈并发编程的核心问题和基本思路

当我们谈论并发编程的时候，我们在谈论什么？用一句话概括：**当多个线程同时访问一个未受保护的共享数据时，就会产生并发问题**。那么多线程编程的本质就是怎样避免上述情况的发生了。这里总结一些，有三种基本的方法。

1. 对共享数据结构进行保护

这是教科书上最常见的方法了。用各种信号量/互斥锁对数据结构进行保护，先加锁，然后执行操作，最后解锁。举个例子，Tao框架中用于网络连接管理的ConnMap就是这么实现的：

```
// ConnMap is a safe map for server connection management.
type ConnMap struct {
    sync.RWMutex
    m map[int64]*ServerConn
}

// NewConnMap returns a new ConnMap.
func NewConnMap() *ConnMap {
    return &ConnMap{
        m: make(map[int64]*ServerConn),
    }
}

// Clear clears all elements in map.
func (cm *ConnMap) Clear() {
    cm.Lock()
    cm.m = make(map[int64]*ServerConn)
    cm.Unlock()
}

// Get gets a server connection with specified net ID.
func (cm *ConnMap) Get(id int64) (*ServerConn, bool) {
    cm.RLock()
    sc, ok := cm.m[id]
    cm.RUnlock()
    return sc, ok
}

// Put puts a server connection with specified net ID in map.
func (cm *ConnMap) Put(id int64, sc *ServerConn) {
    cm.Lock()
    cm.m[id] = sc
    cm.Unlock()
}

// Remove removes a server connection with specified net ID.
func (cm *ConnMap) Remove(id int64) {
    cm.Lock()
    delete(cm.m, id)
    cm.Unlock()
}

// Size returns map size.
func (cm *ConnMap) Size() int {
    cm.RLock()
    size := len(cm.m)
    cm.RUnlock()
    return size
}

// IsEmpty tells whether ConnMap is empty.
func (cm *ConnMap) IsEmpty() bool {
    return cm.Size() <= 0
}
```

2 多线程并行转化为单线程串行

这种方法在前面已经介绍过，它属于无锁化的一种编程方式。多个线程的操作请求都放到一个任务队列中，最终由一个单一的线程来读取队列并串行执行。这种方法在并发量很大的时候还是会有性能瓶颈。

3 采用精心设计的并发数据结构

最好的办法还是要从数据结构上入手，有很多技巧能够让数据结构适应多线程并发访问的场景。比如Java标准库中的java.util.concurrent，包含了各种并发数据结构，其中ConcurrentHashMap的基本原理就是分段锁，对每个段（Segment）加锁保护，并发写入数据时通过散列函数分发到不同的段上面，在SegmentA上加锁并不影响SegmentB的访问。

处理并发多线程问题，一定要小心再小心，思考再思考，一不注意就会踩坑

四. 特别鸣谢

- 《Linux多线程服务端编程》以及Muduo网络编程库
(<https://github.com/chenshuo/muduo>) - by陈硕

- 《Go程序设计语言》 - Donovan & Kernighan
- gotcp - A Go package for quickly building tcp servers
(https://github.com/gansidui/gotcp)
- syncmap - A thread safe map implementation for Golang
(https://github.com/DeanThompson/syncmap)
- leaf - A pragmatic game server framework in Go (https://github.com/leesper/leaf)



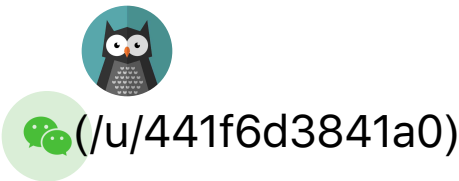
Leesper (/u/10d26e0194c7)

写了 51243 字, 被 100 人关注, 获得了 105 个喜欢
(/u/10d26e0194c7)写了 51243 字, 被 100 人关注, 获得了 105 个喜欢

+ 关注

感谢您的打赏, ^^

赞赏支持



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

49








更多分享

(http://cwb.assets.jianshu.io/notes/images/3868238)



下载简书 App ▶

随时随地发现和创作内容



(/apps/download?utm_source=nbc)




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

31条评论

只看作者

按喜欢排序 按时间正序 按时间倒序




KKiCC (/u/6e3ced9db766)

2楼 · 2016.06.12 10:48
(/u/6e3ced9db766)

可以

👍 1人赞 💬 回复



CodingTech (/u/ec271be1f5dd)

3楼 · 2016.09.01 09:18
(/u/ec271be1f5dd)

👍

👍 赞 💬 回复

Leesper (/u/10d26e0194c7): @CodingTech (/users/ec271be1f5dd) 😊

2016.09.01 09:19 💬 回复

✎ 添加新评论

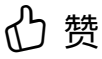


黑哥儿666 (/u/db8e833e40bd)

4楼 · 2016.11.01 23:51

(/u/db8e833e40bd)

一直在寻找一款框架。虽然还没看代码，但是感觉不错。请问这个框架对于峰值的流量是如何控制的？貌似没介绍



赞



回复

Leesper (/u/10d26e0194c7): 哈哈，还没来得及做呢，我也是刚学Go😁

2016.11.02 06:37 回复



添加新评论



shepherdlife (/u/77d7ba9d6fa7)

5楼 · 2016.12.22 10:20

(/u/77d7ba9d6fa7)

好文章



赞



回复

Leesper (/u/10d26e0194c7): @shepherdlife (/users/77d7ba9d6fa7) 谢谢支持！

2016.12.22 10:20 回复



添加新评论

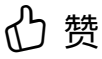


斗破苍穷 (/u/44ff4f832345)

6楼 · 2017.01.06 19:47

(/u/44ff4f832345)

很详细



赞



回复



zjh821 (/u/c203c8b1d48a)

7楼 · 2017.01.12 08:32

(/u/c203c8b1d48a)



赞



回复



楚客 (/u/db33dd26daef)

8楼 · 2017.06.05 19:24

(/u/db33dd26daef)

最近在学习你的tao网络框架，做了下压测，发现在压力大时handle数量和发出去的消息数量不一致

跟踪发现在handleLoop中将任务放到任务池子时可能返回ErrWouldBlock错误，但上层直接忽略了这个错误

调用代码为

```
WorkerPoolInstance().Put(netID, func() {
    handler(NewContextWithNetID(NewContextWithMessage(ctx, msg), netID), c)
})
```

想问一下这种情况该怎么处理，循环调用put直到成功？（这样会阻塞在循环中）

另外我还发现WriteCloser接口的Write方法也可能会发生ErrWouldBlock错误，这种情况建议上层应用怎么处理？



赞



回复

Leesper (/u/10d26e0194c7): @楚客 (/users/db33dd26daef) 好的! 谢谢您发现的这个bug! 这个问题我需要好好研究下再答复您, 因为我也是初学者 😊

2017.06.05 19:26 回复

楚客 (/u/db33dd26daef): @Leesper (/users/10d26e0194c7) 请问一下你这个框架对海量连接的处理效率怎么样, 比如10W客户端长连接, 每个连接起3个goroutine, 这样对调度会有影响吗?


2017.06.07 11:11 回复

Leesper (/u/10d26e0194c7): @楚客 (/users/db33dd26daef) 没有在海量连接的场景下测试过, 但据我所知, go语言做服务器开发一般都是按照这样一个连接起三个goroutine的方式实现的。它相当轻量级。然而单个进程能够建立的网络连接是受文件描述符限制的, 所以您说的这种场景应该是需要启动多个服务器做集群吧, 比如多个网关服务器

2017.06.07 15:45 回复

添加新评论

还有11条评论, 展开查看

爱家爱宝贝爱生活 (/u/7c536ba6bee4)


9楼 · 2017.06.24 18:40

(/u/7c536ba6bee4)

学习了👍

👍 赞

💬 回复

wiseAaron (/u/97c336a8df84)

10楼 · 2017.09.02 19:43

(/u/97c336a8df84)

谢谢作者分享, 学习了


👍 赞

💬 回复

Leesper (/u/10d26e0194c7): 😊

2017.09.04 13:25 回复

添加新评论

myZero1986 (/u/5c23479c74ec)

11楼 · 2017.09.29 11:17

(/u/5c23479c74ec)

开源了吗? 在哪里可以下载源码了

👍 赞

💬 回复

Leesper (/u/10d26e0194c7): @myZero1986 (/users/5c23479c74ec) github哈

2017.09.29 12:39 回复

myZero1986 (/u/5c23479c74ec): @Leesper (/users/10d26e0194c7) 正在学习, 不错。就是准备找一个网络通信架构


2017.09.29 14:42 回复

Leesper (/u/10d26e0194c7): @myZero1986 (/users/5c23479c74ec) 😊




2017.09.29 14:59 回复

添加新评论

被以下专题收入, 发现更多相似内容

Go语言 (/c/c7634b78294d?utm_source=desktop&utm_medium=notes-included-collection)

golang (/c/ae85506e775b?utm_source=desktop&utm_medium=notes-

-  golang (/c/b5c0c0cc7eb3?utm_source=desktop&utm_medium=notes-included-collection)
-  网络编程 (/c/b793190f6509?utm_source=desktop&utm_medium=notes-included-collection)
-  go Network (/c/b6ff180b6cb7?utm_source=desktop&utm_medium=notes-included-collection)

推荐阅读

更多精彩内容 > (/)

Tao Release Notes (/p/06de1d0d03ce?utm_campaign=maleskine&utm_content=note&utm_medium=pc_all_hots&utm_source=recommendation)

Tao 1.7.0 TODOs: HashedWheelTimer 使用反射改进Handler设计 read about teleport Annoucing Tao

1.6.0 Bugfix: writeLoop() drains all pending messages before exit;writeLoop()函数退出前将所有的网络数

Leesper (/u/10d26e0194c7?utm_campaign=maleskine&utm_content=user&utm_medium=pc_all_hots&utm_source=recommendation)

程序员学统计（一）：统计学框架 (/p/d8ff8c6afa09?utm_campaign=maleskine&utm_content=note&utm_medium=pc_all_hots&utm_source=recommendation)

“不要轻视简单，简单意味着坚固，整个数学大厦都是建立在这种简单到不能再简单，但在逻辑上坚如磐石的公理基础上。” ——《三体》 作为一个正在向数据

Leesper (/u/10d26e0194c7?utm_campaign=maleskine&utm_content=user&utm_medium=pc_all_hots&utm_source=recommendation)

陈道明：遇见，就是这辈子最大的幸运 (/p/d8cc4aef053f?utm_campaign=maleskine&utm_content=note&utm_medium=pc_all_hots&utm_source=recommendation)

文/麦大人 01 前段时间《我的前半生》余温不减，剧中参演配角老卓的陈道明，举手投足之间都是戏，可谓吸引了所有人的目光。有一次，马伊琍上《圆桌

麦大人 (/u/2b3ad4f2a058?utm_campaign=maleskine&utm_content=user&utm_medium=pc_all_hots&utm_source=recommendation)

合理分配月收入，投资自己，我们从月薪2000起步， ... (/p/065feced026b?utm_campaign=maleskine&utm_content=note&utm_medium=pc_all_hots&utm_source=recommendation)

作者：百灵 写这篇文的初衷是之前收到不少职场新人和师弟师妹的咨询，以及很多新人觉得自己收入太少，几百块钱不值得一存。我把自己的经历写出来，希

裂痕之光 (/u/5426929b9c20?utm_campaign=maleskine&utm_content=user&utm_medium=pc_all_hots&utm_source=recommendation)

这个超级实用的淘宝购物省钱攻略，99%的人居然都还... (/p/b73783355e73?utm_campaign=maleskine&utm_content=note&utm_medium=pc_all_hots&utm_source=recommendation)

大家好，我是小丸子~ 前几天我在简书给大家分享了电脑版的淘宝省钱攻略：《掌握了这个技巧，你每次在淘宝购物都可以省一大笔钱！》，很多人留言表示

小丸子的杂物集 (/u/24bca2bb387d?utm_campaign=maleskine&utm_content=user&utm_medium=pc_all_hots&utm_source=recommendation)