

37 | 代码操练：怎么实现一个TCP服务器？（中）

2022-01-26 Tony Bai

《Tony Bai · Go语言第一课》

[课程介绍 >](#)



讲述: Tony Bai

时长 14:53 大小 13.65M



你好，我是 Tony Bai。

上一讲中，我们讲解了解决 Go 语言学习“最后一公里”的实用思路，那就是“理解问题”->“技术预研与储备”->“设计与实现”的三角循环，并且我们也完成了“理解问题”和“技术预研与储备”这两个环节，按照“三角循环”中的思路，这一讲我们应该针对实际问题进行一轮设计与实现了。

今天，我们的目标是实现一个基于 TCP 的自定义应用层协议的通信服务端，要完成这一目标，我们需要建立协议的抽象、实现协议的打包与解包、服务端的组装、验证与优化等工作。一步一步来，我们先在程序世界建立一个对上一讲中自定义应用层协议的抽象。

领资料

建立对协议的抽象

程序是对现实世界的抽象。对于现实世界的自定义应用协议规范，我们需要在程序世界建立起对这份协议的抽象。在进行抽象之前，我们先建立这次实现要用的源码项目 tcp-server-

demo1，建立的步骤如下：

复制代码

```
1 $mkdir tcp-server-demo1
2 $cd tcp-server-demo1
3 $go mod init github.com/bigwhite/tcp-server-demo1
4 go: creating new go.mod: module github.com/bigwhite/tcp-server-demo1
```

为了方便学习，我这里再将上一讲中的自定义协议规范贴出来对照参考：

表：请求消息包定义

序号	字符名	类型	长度	描述
1	totalLength	uint32	4字节	消息总长度（含自身及后面消息体）
2	commandID	uint8	1字节	消息或响应的类型
3	ID	数字类型string	8字节	消息流水号（顺序累加，步长为1，循环使用（一对请求和应答消息的流水号必须相同））
4	payload	字节序列	任意长度	消息的有效载荷，应用层需要的有效数据

表：响应消息包定义

序号	字符名	类型	长度	描述
1	totalLength	uint32	4字节	消息总长度（含自身及后面消息体）
2	commandID	uint8	1字节	消息或响应的类型
3	ID	数字类型string	8字节	消息流水号（顺序累加，步长为1，循环使用（一对请求和应答消息的流水号必须相同））
4	result	uint8	1字节	响应状态（0：正常；1：错误）



深入协议字段

上一讲，我们没有深入到协议规范中对协议的各个字段进行讲解，但在建立抽象之前，我们有必要了解一下各个字段的具体含义。

这是一个高度简化的、基于二进制模式定义的协议。二进制模式定义的特点，就是采用长度字段标识独立数据包的边界。



在这个协议规范中，我们看到：请求包和应答包的第一个字段（totalLength）都是包的总长度，它就是用来标识包边界的那个字段，也是在应用层用于“分割包”的最重要字段。

请求包与应答包的第二个字段也一样，都是 `commandID`，这个字段用于标识包类型，这里我们定义四种包类型：

- 连接请求包（值为 `0x01`）
- 消息请求包（值为 `0x02`）
- 连接响应包（值为 `0x81`）
- 消息响应包（值为 `0x82`）

换为对应的代码就是：

 复制代码

```
1 const (  
2     CommandConn    = iota + 0x01 // 0x01, 连接请求包  
3     CommandSubmit          // 0x02, 消息请求包  
4 )  
5  
6 const (  
7     CommandConnAck    = iota + 0x81 // 0x81, 连接请求的响应包  
8     CommandSubmitAck          // 0x82, 消息请求的响应包  
9 )
```

请求包与应答包的第三个字段都是 `ID`，`ID` 是每个连接上请求包的消息流水号，顺序累加，步长为 `1`，循环使用，多用来请求发送方后匹配响应包，所以要求一对请求与响应消息的流水号必须相同。

请求包与响应包唯一的不同之处，就在于最后一个字段：请求包定义了有效载荷

（`payload`），这个字段承载了应用层需要的业务数据；而响应包则定义了请求包的响应状态字段（`result`），这里其实简化了响应状态字段的取值，成功的响应用 `0` 表示，如果是失败的响应，无论失败原因是什么，我们都用 `1` 来表示。

明确了应用层协议的各个字段定义之后，我们接下来就看看如何建立起对这个协议的抽象。

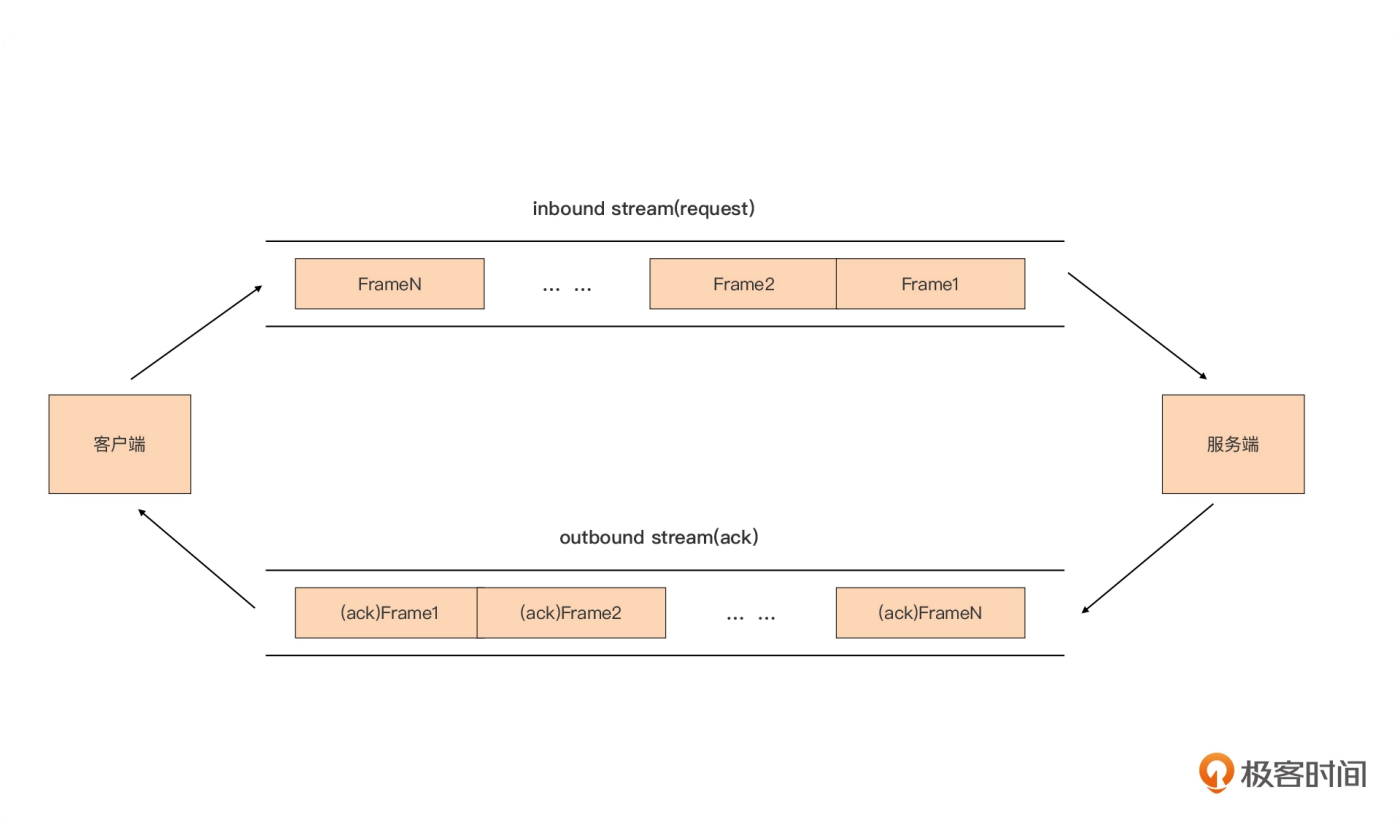


建立 Frame 和 Packet 抽象

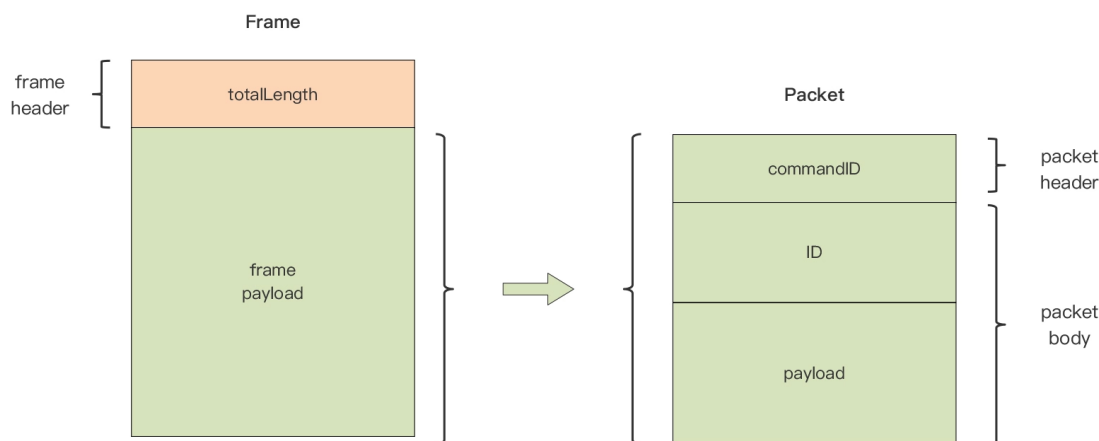
首先我们要知道，`TCP` 连接上的数据是一个没有边界的字节流，但在业务层眼中，没有字节流，只有各种协议消息。因此，无论是从客户端到服务端，还是从服务端到客户端，业务层在

连接上看到的都应该是一个挨着一个的协议消息流。

现在我们建立第一个抽象：**Frame**。每个 **Frame** 表示一个协议消息，这样在业务层眼中，连接上的字节流就是由一个接着一个 **Frame** 组成的，如下图所示：



我们的自定义协议就封装在这一个个的 **Frame** 中。协议规定了将 **Frame** 分割开来的方法，那就是利用每个 **Frame** 开始处的 **totalLength**，每个 **Frame** 由一个 **totalLength** 和 **Frame** 的负载（**payload**）构成，比如你可以看看下图中左侧的 **Frame** 结构：



这样，我们通过 **Frame header: totalLength** 就可以将 **Frame** 之间隔离开来。

在这个基础上，我们建立协议的第二个抽象：**Packet**。我们将 **Frame payload** 定义为一个 **Packet**。上图右侧展示的就是 **Packet** 的结构。

Packet 就是业务层真正需要的消息，每个 **Packet** 由 **Packet 头** 和 **Packet Body** 部分组成。**Packet 头** 就是 **commandID**，用于标识这个消息的类型；而 **ID** 和 **payload**（**packet payload**）或 **result** 字段组成了 **Packet** 的 **Body** 部分，对业务层有价值的数据都包含在 **Packet Body** 部分。

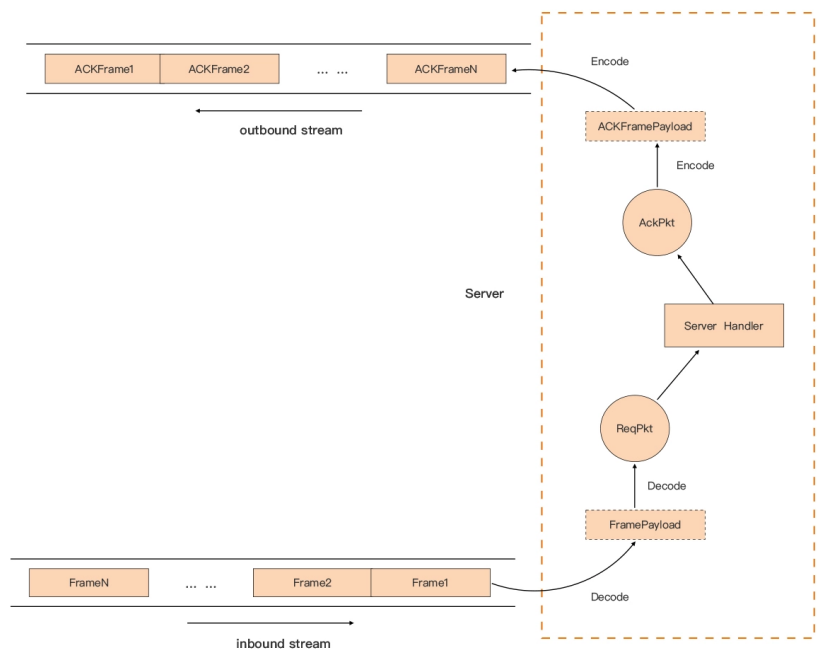
那么到这里，我们就通过 **Frame** 和 **Packet** 两个类型结构，完成了程序世界对我们私有协议规范的抽象。接下来，我们要做的就是基于 **Frame** 和 **Packet** 这两个概念，实现对我们私有协议的解包与打包操作。

协议的解包与打包

所谓协议的**解包**（**decode**），就是指识别 **TCP** 连接上的字节流，将一组字节“转换”成一个特定类型的协议消息结构，然后这个消息结构会被业务处理逻辑使用。

而**打包**（**encode**）刚刚好相反，是指将一个特定类型的消息结构转换为一组字节，然后这组字节数据会被放在连接上发送出去。

具体到我们这个自定义协议上，解包就是指字节流 -> Frame，打包是指Frame -> 字节流。你可以看一下针对这个协议的服务端解包与打包的流程图：



我们看到，TCP 流数据先后经过 frame decode 和 packet decode，得到应用层所需的 packet 数据，而业务层回复的响应，则先后经过 packet 的 encode 与 frame 的 encode，写入 TCP 数据流中。

到这里，我们实际上已经完成了协议抽象的设计与解包打包原理的设计过程了。接下来，我们先来看看私有协议部分的相关代码实现。

Frame 的实现

前面说过，协议部分最重要的两个抽象是 Frame 和 Packet，于是我们就在项目中建立 frame 包与 packet 包，分别与两个协议抽象对应。frame 包的职责是提供识别 TCP 流边界的编解码器，我们可以很容易为这样的编解码器，定义出一个统一的接口类型 StreamFrameCodec：



复制代码

```
1 // tcp-server-demo1/frame/frame.go
2
3 type FramePayload []byte
4
5 type StreamFrameCodec interface {
6     Encode(io.Writer, FramePayload) error // data -> frame, 并写入io.Writer
7     Decode(io.Reader) (FramePayload, error) // 从io.Reader中提取frame payload, 并
```

```
8 }
```

`StreamFrameCodec` 接口类型有两个方法 `Encode` 与 `Decode`。`Encode` 方法用于将输入的 `Frame payload` 编码为一个 `Frame`，然后写入 `io.Writer` 所代表的输出（outbound）TCP 流中。而 `Decode` 方法正好相反，它从代表输入（inbound）TCP 流的 `io.Reader` 中读取一个完整 `Frame`，并将得到的 `Frame payload` 解析出来并返回。

这里，我们给出一个针对我们协议的 `StreamFrameCodec` 接口的实现：

 复制代码

```
1 // tcp-server-demo1/frame/frame.go
2
3 var ErrShortWrite = errors.New("short write")
4 var ErrShortRead = errors.New("short read")
5
6 type myFrameCodec struct{}
7
8 func NewMyFrameCodec() StreamFrameCodec {
9     return &myFrameCodec{}
10 }
11
12 func (p *myFrameCodec) Encode(w io.Writer, framePayload FramePayload) error {
13     var f = framePayload
14     var totalLen int32 = int32(len(framePayload)) + 4
15
16     err := binary.Write(w, binary.BigEndian, &totalLen)
17     if err != nil {
18         return err
19     }
20
21     n, err := w.Write([]byte(f)) // write the frame payload to outbound stream
22     if err != nil {
23         return err
24     }
25
26     if n != len(framePayload) {
27         return ErrShortWrite
28     }
29
30     return nil
31 }
32
33 func (p *myFrameCodec) Decode(r io.Reader) (FramePayload, error) {
34     var totalLen int32
35     err := binary.Read(r, binary.BigEndian, &totalLen)
36     if err != nil {
37         return nil, err
```

领资料


```

38     }
39
40     buf := make([]byte, totalLen-4)
41     n, err := io.ReadFull(r, buf)
42     if err != nil {
43         return nil, err
44     }
45
46     if n != int(totalLen-4) {
47         return nil, ErrShortRead
48     }
49
50     return FramePayload(buf), nil
51 }

```

在这段实现中，有三点事项需要注意：

- 网络字节序使用大端字节序（**BigEndian**），因此无论是 **Encode** 还是 **Decode**，我们都是用 **binary.BigEndian**；
- **binary.Read** 或 **Write** 会根据参数的宽度，读取或写入对应的字节个数的字节，这里 **totalLen** 使用 **int32**，那么 **Read** 或 **Write** 只会操作数据流中的 4 个字节；
- 这里没有设置网络 I/O 操作的 **Deadline**，**io.ReadFull** 一般会读满你所需的字节数，除非遇到 **EOF** 或 **ErrUnexpectedEOF**。

在工程实践中，保证打包与解包正确的最有效方式就是**编写单元测试**，**StreamFrameCodec** 接口的 **Decode** 和 **Encode** 方法的参数都是接口类型，这让我们可以很容易为 **StreamFrameCodec** 接口的实现编写测试用例。下面是我为 **myFrameCodec** 编写了两个测试用例：

复制代码

```

1 // tcp-server-demo1/frame/frame_test.go
2
3 func TestEncode(t *testing.T) {
4     codec := NewMyFrameCodec()
5     buf := make([]byte, 0, 128)
6     rw := bytes.NewBuffer(buf)
7
8     err := codec.Encode(rw, []byte("hello"))
9     if err != nil {
10         t.Errorf("want nil, actual %s", err.Error())
11     }
12

```

领资料


```

13 // 验证Encode的正确性
14 var totalLen int32
15 err = binary.Read(rw, binary.BigEndian, &totalLen)
16 if err != nil {
17     t.Errorf("want nil, actual %s", err.Error())
18 }
19
20 if totalLen != 9 {
21     t.Errorf("want 9, actual %d", totalLen)
22 }
23
24 left := rw.Bytes()
25 if string(left) != "hello" {
26     t.Errorf("want hello, actual %s", string(left))
27 }
28 }
29
30 func TestDecode(t *testing.T) {
31     codec := NewMyFrameCodec()
32     data := []byte{0x0, 0x0, 0x0, 0x9, 'h', 'e', 'l', 'l', 'o'}
33
34     payload, err := codec.Decode(bytes.NewReader(data))
35     if err != nil {
36         t.Errorf("want nil, actual %s", err.Error())
37     }
38
39     if string(payload) != "hello" {
40         t.Errorf("want hello, actual %s", string(payload))
41     }
42 }

```

我们看到，测试 `Encode` 方法，我们其实不需要建立真实的网络连接，只要用一个满足 `io.Writer` 的 `bytes.Buffer` 实例“冒充”真实网络连接就可以了，同时 `bytes.Buffer` 类型也实现了 `io.Reader` 接口，我们可以很方便地从中读取 `Encode` 后的内容，并进行校验比对。

为了提升测试覆盖率，我们还需要尽可能让测试覆盖到所有可测的错误执行分支上。这里，我模拟了 `Read` 或 `Write` 出错的情况，让执行流进入到 `Decode` 或 `Encode` 方法的错误分支中：



复制代码



```

1 type ReturnErrorWriter struct {
2     W io.Writer
3     Wn int // 第几次调用Write返回错误
4     wc int // 写操作次数计数
5 }
6
7 func (w *ReturnErrorWriter) Write(p []byte) (n int, err error) {
8     w.WC++

```

```

9         if w.Wc >= w.Wn {
10             return 0, errors.New("write error")
11         }
12         return w.W.Write(p)
13     }
14
15     type ReturnErrorReader struct {
16         R io.Reader
17         Rn int // 第几次调用Read返回错误
18         rc int // 读操作次数计数
19     }
20
21     func (r *ReturnErrorReader) Read(p []byte) (n int, err error) {
22         r.rc++
23         if r.rc >= r.Rn {
24             return 0, errors.New("read error")
25         }
26         return r.R.Read(p)
27     }
28
29     func TestEncodeWithWriteFail(t *testing.T) {
30         codec := NewMyFrameCodec()
31         buf := make([]byte, 0, 128)
32         w := bytes.NewBuffer(buf)
33
34         // 模拟binary.Write返回错误
35         err := codec.Encode(&ReturnErrorWriter{
36             W: w,
37             Wn: 1,
38         }, []byte("hello"))
39         if err == nil {
40             t.Errorf("want non-nil, actual nil")
41         }
42
43         // 模拟w.Write返回错误
44         err = codec.Encode(&ReturnErrorWriter{
45             W: w,
46             Wn: 2,
47         }, []byte("hello"))
48         if err == nil {
49             t.Errorf("want non-nil, actual nil")
50         }
51     }
52
53     func TestDecodeWithReadFail(t *testing.T) {
54         codec := NewMyFrameCodec()
55         data := []byte{0x0, 0x0, 0x0, 0x9, 'h', 'e', 'l', 'l', 'o'}
56
57         // 模拟binary.Read返回错误
58         _, err := codec.Decode(&ReturnErrorReader{
59             R: bytes.NewReader(data),
60             Rn: 1,

```

```

61     })
62     if err == nil {
63         t.Errorf("want non-nil, actual nil")
64     }
65
66     // 模拟io.ReadFull返回错误
67     _, err = codec.Decode(&ReturnErrorReader{
68         R: bytes.NewReader(data),
69         Rn: 2,
70     })
71     if err == nil {
72         t.Errorf("want non-nil, actual nil")
73     }
74 }

```

为了实现错误分支的测试，我们在测试代码源文件中创建了两个类型：**ReturnErrorWriter** 和 **ReturnErrorReader**，它们分别实现了 **io.Writer** 与 **io.Reader**。

我们可以控制在第几次调用这两个类型的 **Write** 或 **Read** 方法时，返回错误，这样就可以让 **Encode** 或 **Decode** 方法按照我们的意图，进入到不同错误分支中去。有了这两个用例，我们的 **frame** 包的测试覆盖率（通过 **go test -cover** . 可以查看）就可以达到 **90%** 以上了。

Packet 的实现

接下来，我们再看看 **Packet** 这个抽象的实现。和 **Frame** 不同，**Packet** 有多种类型（这里只定义了 **Conn**、**submit**、**connack**、**submit ack**）。所以我们要先抽象一下这些类型需要遵循的共同接口：

```

1 // tcp-server-demo1/packet/packet.go
2
3 type Packet interface {
4     Decode([]byte) error // []byte -> struct
5     Encode() ([]byte, error) // struct -> []byte
6 }

```

 复制代码

领资料

其中，**Decode** 是将一段字节流数据解码为一个 **Packet** 类型，可能是 **conn**，可能是 **submit** 等，具体我们要根据解码出来的 **commandID** 判断。而 **Encode** 则是将一个 **Packet** 类型编码为一段字节流数据。

考虑到篇幅与复杂性，我们这里只完成 `submit` 和 `submitack` 类型的 `Packet` 接口实现，省略了 `conn` 流程，也省略 `conn` 以及 `connack` 类型的实现，你可以课后自己思考一下有 `conn` 流程时代码应该如何调整。

 复制代码

```
1 // tcp-server-demo1/packet/packet.go
2
3 type Submit struct {
4     ID      string
5     Payload []byte
6 }
7
8 func (s *Submit) Decode(pktBody []byte) error {
9     s.ID = string(pktBody[:8])
10    s.Payload = pktBody[8:]
11    return nil
12 }
13
14 func (s *Submit) Encode() ([]byte, error) {
15     return bytes.Join([][]byte{[]byte(s.ID[:8]), s.Payload}, nil), nil
16 }
17
18 type SubmitAck struct {
19     ID      string
20     Result  uint8
21 }
22
23 func (s *SubmitAck) Decode(pktBody []byte) error {
24     s.ID = string(pktBody[0:8])
25     s.Result = uint8(pktBody[8])
26     return nil
27 }
28
29 func (s *SubmitAck) Encode() ([]byte, error) {
30     return bytes.Join([][]byte{[]byte(s.ID[:8]), []byte{s.Result}}, nil), nil
31 }
```

这里各种类型的编解码被调用的前提，是明确数据流是什么类型的，因此我们需要在包级提供一个导出的函数 `Decode`，这个函数负责从字节流中解析出对应的类型（根据 `commandID`），并调用对应类型的 `Decode` 方法：

 领资料



 复制代码


```
1 // tcp-server-demo1/packet/packet.go
2
3 func Decode(packet []byte) (Packet, error) {
```

```

4  commandID := packet[0]
5  pktBody := packet[1:]
6
7  switch commandID {
8  case CommandConn:
9      return nil, nil
10 case CommandConnAck:
11     return nil, nil
12 case CommandSubmit:
13     s := Submit{}
14     err := s.Decode(pktBody)
15     if err != nil {
16         return nil, err
17     }
18     return &s, nil
19 case CommandSubmitAck:
20     s := SubmitAck{}
21     err := s.Decode(pktBody)
22     if err != nil {
23         return nil, err
24     }
25     return &s, nil
26 default:
27     return nil, fmt.Errorf("unknown commandID [%d]", commandID)
28 }
29 }

```

同样，我们也需要包级的 `Encode` 函数，根据传入的 `packet` 类型调用对应的 `Encode` 方法实现对象的编码：

 复制代码

```

1  // tcp-server-demo1/packet/packet.go
2
3  func Encode(p Packet) ([]byte, error) {
4      var commandID uint8
5      var pktBody []byte
6      var err error
7
8      switch t := p.(type) {
9      case *Submit:
10         commandID = CommandSubmit
11         pktBody, err = p.Encode()
12         if err != nil {
13             return nil, err
14         }
15      case *SubmitAck:
16         commandID = CommandSubmitAck
17         pktBody, err = p.Encode()
18         if err != nil {

```

领资料

```

19         return nil, err
20     }
21     default:
22         return nil, fmt.Errorf("unknown type [%s]", t)
23     }
24     return bytes.Join([][]byte{[]byte{commandID}, pktBody}, nil), nil
25 }


```

不过，对 `packet` 包中各个类型的 `Encode` 和 `Decode` 方法的测试，与 `frame` 包的相似，这里我就把为 `packet` 包编写单元测试的任务就交给你自己完成了，如果有什么问题欢迎在留言区留言。

好了，万事俱备，只欠东风！下面我们就来编写服务端的程序结构，将 `tcp conn` 与 `Frame`、`Packet` 连接起来。

服务端的组装

在上一讲中，我们按照每个连接一个 `Goroutine` 的模型，给出了典型 `Go` 网络服务端程序的结构，这里我们就以这个结构为基础，将 `Frame`、`Packet` 加进来，形成我们的第一版服务端实现：

 复制代码

```

1 // tcp-server-demo1/cmd/server/main.go
2
3 package main
4
5 import (
6     "fmt"
7     "net"
8
9     "github.com/bigwhite/tcp-server-demo1/frame"
10    "github.com/bigwhite/tcp-server-demo1/packet"
11 )
12
13 func handlePacket(framePayload []byte) (ackFramePayload []byte, err error) {
14     var p packet.Packet
15     p, err = packet.Decode(framePayload)
16     if err != nil {
17         fmt.Println("handleConn: packet decode error:", err)
18         return
19     }
20
21     switch p.(type) {
22     case *packet.Submit:
23         submit := p.(*packet.Submit)

```

领资料

```

24     fmt.Printf("recv submit: id = %s, payload=%s\n", submit.ID, string(submit.P
25     submitAck := &packet.SubmitAck{
26         ID:      submit.ID,
27         Result: 0,
28     }
29     ackFramePayload, err = packet.Encode(submitAck)
30     if err != nil {
31         fmt.Println("handleConn: packet encode error:", err)
32         return nil, err
33     }
34     return ackFramePayload, nil
35 default:
36     return nil, fmt.Errorf("unknown packet type")
37 }
38 }
39
40 func handleConn(c net.Conn) {
41     defer c.Close()
42     frameCodec := frame.NewMyFrameCodec()
43
44     for {
45         // decode the frame to get the payload
46         framePayload, err := frameCodec.Decode(c)
47         if err != nil {
48             fmt.Println("handleConn: frame decode error:", err)
49             return
50         }
51
52         // do something with the packet
53         ackFramePayload, err := handlePacket(framePayload)
54         if err != nil {
55             fmt.Println("handleConn: handle packet error:", err)
56             return
57         }
58
59         // write ack frame to the connection
60         err = frameCodec.Encode(c, ackFramePayload)
61         if err != nil {
62             fmt.Println("handleConn: frame encode error:", err)
63             return
64         }
65     }
66 }
67
68 func main() {
69     l, err := net.Listen("tcp", ":8888")
70     if err != nil {
71         fmt.Println("listen error:", err)
72         return
73     }
74
75     for {

```



```

76     c, err := l.Accept()
77     if err != nil {
78         fmt.Println("accept error:", err)
79         break
80     }
81     // start a new goroutine to handle the new connection.
82     go handleConn(c)
83 }
84

```

这个程序的逻辑非常清晰，服务端程序监听 8888 端口，并在每次调用 `Accept` 方法后得到一个连接，服务端程序将这个新连接交到一个新的 `Goroutine` 中处理。

新 `Goroutine` 的主函数为 `handleConn`，有了 `Packet` 和 `Frame` 这两个抽象的加持，这个函数同样拥有清晰的代码调用结构：

 复制代码

```

1 // handleConn的调用结构
2
3 read frame from conn
4     ->frame decode
5     -> handle packet
6         -> packet decode
7         -> packet(ack) encode
8     ->frame(ack) encode
9 write ack frame to conn

```

到这里，一个基于 `TCP` 的自定义应用层协议的经典阻塞式的服务端就完成了。不过这里的服务端依旧是一个简化的实现，比如我们这里没有考虑支持优雅退出、没有捕捉某个链接上出现的可能导致整个程序退出的 `panic` 等，这些我也想作为作业留给你。

接下来，我们就来验证一下这个服务端实现是否能正常工作。

验证测试

要验证服务端的实现是否可以正常工作，我们需要实现一个自定义应用层协议的客户端。这里，我们同样基于 `frame`、`packet` 两个包，实现了一个自定义应用层协议的客户端。下面是客户端的 `main` 函数：

 领资料



 复制代码

```

1 // tcp-server-demo1/cmd/client/main.go
2
3 func main() {
4     var wg sync.WaitGroup
5     var num int = 5
6
7     wg.Add(5)
8
9     for i := 0; i < num; i++ {
10         go func(i int) {
11             defer wg.Done()
12             startClient(i)
13         }(i + 1)
14     }
15     wg.Wait()
16 }

```

我们看到，客户端启动了 5 个 Goroutine，模拟 5 个并发连接。startClient 函数是每个连接的主处理函数，我们来看一下：

 复制代码

```

1 func startClient(i int) {
2     quit := make(chan struct{})
3     done := make(chan struct{})
4     conn, err := net.Dial("tcp", ":8888")
5     if err != nil {
6         fmt.Println("dial error:", err)
7         return
8     }
9     defer conn.Close()
10    fmt.Printf("[client %d]: dial ok", i)
11
12    // 生成payload
13    rng, err := codename.DefaultRNG()
14    if err != nil {
15        panic(err)
16    }
17
18    frameCodec := frame.NewMyFrameCodec()
19    var counter int
20
21    go func() {
22        // handle ack
23        for {
24            select {
25            case <-quit:
26                done <- struct{}{}
27                return
28            default:
29

```

领资料

```

30     conn.SetReadDeadline(time.Now().Add(time.Second * 1))
31     ackFramePayload, err := frameCodec.Decode(conn)
32     if err != nil {
33         if e, ok := err.(net.Error); ok {
34             if e.Timeout() {
35                 continue
36             }
37         }
38         panic(err)
39     }
40
41     p, err := packet.Decode(ackFramePayload)
42     submitAck, ok := p.(*packet.SubmitAck)
43     if !ok {
44         panic("not submitack")
45     }
46     fmt.Printf("[client %d]: the result of submit ack[%s] is %d\n", i,
47 }
48 }()
49
50 for {
51     // send submit
52     counter++
53     id := fmt.Sprintf("%08d", counter) // 8 byte string
54     payload := codename.Generate(rng, 4)
55     s := &packet.Submit{
56         ID:      id,
57         Payload: []byte(payload),
58     }
59
60     framePayload, err := packet.Encode(s)
61     if err != nil {
62         panic(err)
63     }
64
65     fmt.Printf("[client %d]: send submit id = %s, payload=%s, frame length
66         i, s.ID, s.Payload, len(framePayload)+4)
67
68     err = frameCodec.Encode(conn, framePayload)
69     if err != nil {
70         panic(err)
71     }
72
73     time.Sleep(1 * time.Second)
74     if counter >= 10 {
75         quit <- struct{}{}
76         <-done
77         fmt.Printf("[client %d]: exit ok", i)
78         return
79     }
80 }
81 }

```

关于 `startClient` 函数，我们需要简单说明几点。

首先，`startClient` 函数启动了两个 `Goroutine`，一个负责向服务端发送 `submit` 消息请求，另外一个 `Goroutine` 则负责读取服务端返回的响应；

其次，客户端发送的 `submit` 请求的负载（`payload`）是由第三方包 `github.com/lucasepe/codename` 负责生成的，这个包会生成一些对人类可读的随机字符串，比如：`firm-iron`、`moving-colleen`、`game-nova` 这样的字符串；

另外，负责读取服务端返回响应的 `Goroutine`，使用 `SetReadDeadline` 方法设置了读超时，这主要是考虑该 `Goroutine` 可以在收到退出通知时，能及时从 `Read` 阻塞中跳出来。

好了，现在我们就来构建和运行一下这两个程序。

我在 `tcp-server-demo1` 目录下提供了 `Makefile`，如果你使用的是 `Linux` 或 `macOS` 操作系统，可以直接敲入 `make` 构建两个程序，如果你是在 `Windows` 下构建，可以直接敲入下面的 `go build` 命令构建：

 复制代码

```
1 $make
2 go build github.com/bigwhite/tcp-server-demo1/cmd/server
3 go build github.com/bigwhite/tcp-server-demo1/cmd/client
```

构建成功后，我们先来启动 `server` 程序：

 复制代码

```
1 $./server
2 server start ok(on *.8888)
```

领资料

然后，我们启动 `client` 程序，启动后 `client` 程序便会向服务端建立 5 条连接，并发送 `submit` 请求，`client` 端的部分日志如下：

 复制代码

```

1 $. /client
2 [client 5]: dial ok
3 [client 1]: dial ok
4 [client 5]: send submit id = 00000001, payload=credible-deathstrike-33e1, frame
5 [client 3]: dial ok
6 [client 1]: send submit id = 00000001, payload=helped-lester-8f15, frame length
7 [client 4]: dial ok
8 [client 4]: send submit id = 00000001, payload=strong-timeslip-07fa, frame leng
9 [client 3]: send submit id = 00000001, payload=wondrous-expediter-136e, frame l
10 [client 5]: the result of submit ack[00000001] is 0
11 [client 1]: the result of submit ack[00000001] is 0
12 [client 3]: the result of submit ack[00000001] is 0
13 [client 2]: dial ok
14 ...
15 [client 3]: send submit id = 00000010, payload=bright-monster-badoon-5719, fram
16 [client 4]: send submit id = 00000010, payload=crucial-wallop-ec2d, frame lengt
17 [client 2]: send submit id = 00000010, payload=pro-caliban-c803, frame length =
18 [client 1]: send submit id = 00000010, payload=legible-shredder-3d81, frame len
19 [client 5]: send submit id = 00000010, payload=settled-iron-monger-bf78, frame
20 [client 3]: the result of submit ack[00000010] is 0
21 [client 4]: the result of submit ack[00000010] is 0
22 [client 1]: the result of submit ack[00000010] is 0
23 [client 2]: the result of submit ack[00000010] is 0
24 [client 5]: the result of submit ack[00000010] is 0
25 [client 4]: exit ok
26 [client 1]: exit ok
27 [client 3]: exit ok
28 [client 5]: exit ok
29 [client 2]: exit ok

```

client 在每条连接上发送 10 个 submit 请求后退出。这期间服务端会输出如下日志：

```

1 recv submit: id = 00000001, payload=credible-deathstrike-33e1
2 recv submit: id = 00000001, payload=helped-lester-8f15
3 recv submit: id = 00000001, payload=wondrous-expediter-136e
4 recv submit: id = 00000001, payload=strong-timeslip-07fa
5 recv submit: id = 00000001, payload=delicate-leatherneck-4b12
6 recv submit: id = 00000002, payload=certain-deadpool-779d
7 recv submit: id = 00000002, payload=clever-vapor-25ce
8 recv submit: id = 00000002, payload=causal-guardian-4f84
9 recv submit: id = 00000002, payload=noted-tombstone-1b3e
10 ...
11 recv submit: id = 00000010, payload=settled-iron-monger-bf78
12 recv submit: id = 00000010, payload=pro-caliban-c803
13 recv submit: id = 00000010, payload=legible-shredder-3d81
14 handleConn: frame decode error: EOF
15 handleConn: frame decode error: EOF
16 handleConn: frame decode error: EOF

```

 复制代码

领资料

```
17 handleConn: frame decode error: EOF
18 handleConn: frame decode error: EOF
```

从结果来看，我们实现的这一版服务端运行正常！

小结

好了，今天的课讲到这里就结束了，现在我们一起回顾一下吧。

在上一讲完成对 **socket** 编程模型、网络 **I/O** 操作的技术预研后，这一讲我们正式进入基于 **TCP** 的自定义应用层协议的通信服务端的设计与实现环节。

在这一环节中，我们首先建立了对协议的抽象，这是实现通信服务端的基石。我们使用 **Frame** 的概念来表示 **TCP** 字节流中的每一个协议消息，这使得在业务层的视角下，连接上的字节流就是由一个接着一个 **Frame** 组成的。接下来，我们又建立了第二个抽象 **Packet**，来表示业务层真正需要的消息。

在这两个抽象的基础上，我们实现了 **frame** 与 **packet** 各自的打包与解包，整个实现是低耦合的，我们可以在对 **frame** 编写测试用例时体会到这一点。

最后，我们把上一讲提到的、一个 **Goroutine** 负责处理一个连接的典型 **Go** 网络服务端程序结构与 **frame**、**packet** 的实现组装到一起，就实现了我们的第一版服务端。之后，我们还编写了客户端模拟器对这个服务端的实现做了验证。

这个服务端采用的是 **Go** 经典阻塞 **I/O** 的编程模型，你是不是已经感受到了这种模型在开发阶段带来的好处了呢！

思考题

在这讲的中间部分，我已经把作业留给你了：

1. 为 **packet** 包编写单元测试；
2. 为我们的服务端增加优雅退出机制，以及捕捉某个链接上出现的可能导致整个程序退出的 **panic**。


[🔗 项目的源代码在这里！](#)


领资料



分享给需要的人，Ta订阅超级会员，你最高得 50 元

Ta单独购买本课程，你将得 20 元

 生成海报并分享

 赞 6  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 36 | 打稳根基：怎么实现一个TCP服务器？（上）

下一篇 38 | 成果优化：怎么实现一个TCP服务器？（下）

更多课程推荐

陈天 · Rust 编程第一课

实战驱动，快速上手 Rust

陈天

Tubi TV 研发副总裁



会员免费

¥199

200+ 原理图，详解 Rust 设计理念

领资料

精选留言 (7)

 写留言



罗杰 

2022-01-26

还是老师实现的代码优雅，我们项目的这块代码是刚开始学 Go 时实现的，只能说可以用。但对比老师的实现，我觉得我们的代码可以好好优化一下了。

作者回复:



2



骨汤鸡蛋面

2022-01-26

老师，一些rpc框架学习 http2 的stream概念，在connection与协议之间加了一个stream层，这块主要抽象了啥，很想听一下老师的看法。

作者回复: 我觉得更多是对通信两端交互模式的抽象。

共 2 条评论 >



1



酒醒何处

2022-03-06

CommandConnAck = iota + 0x80 // 0x81，连接请求的响应包
这儿应该是：

CommandConnAck = iota + 0x81 // 0x81，连接请求的响应包

作者回复: 看的真细致！ 感谢指出笔误，后续让编辑老师改一下。



顷

2022-03-02

老师，Makefile的相关知识考虑在加餐要说下吗

作者回复: Makefile在本专栏中不会展开说了。一方面，Makefile与Go并非紧耦合，也并非必须，只是构建辅助工具的一种。另一方面，本专栏中使用的Makefile的内容都是最简单的，一般只有一个target，容易理解，不需要展开大家应该都能理解。

学习过程中，有有关makefile的问题，可以在留言区提出，我尽力解答。



左耳朵东

2022-02-12

client 代码中的 done chan 好像没必要吧，去掉它也能正常退出

领资料



作者回复: 这里的确没必要。但是如果handle ack的goroutine在退出前需要执行一些清理工作, 那么done就有必要了。否则可能会出现handle ack的goroutine没有执行完清理工作, send goroutine就退出的进而导致main goroutine退出前某handle ack的goroutine都没有执行完清理工作。

共 2 条评论 >



陈东

2022-01-27

老师能加餐一节单元测试吗? 查很多资料, 没有学到测试的入门。谢谢。

作者回复: 这个需要后续和编辑老师商量一下。

共 2 条评论 >



Calvin

2022-01-26

老师, Conn 和 ConnAck 要实现的话, 请问从业务中来讲, 一般会需要发送一些什么 Payload 呢? 我看这里的例子没有他们也可以正常运行整个流程, 是类似 需要认证的系统中的登录账号和密码 的这种内容吗?

作者回复: 对的, conn流程一般有一个身份验证的过程。考虑到篇幅, 文中没有加conn和connack, 如果加上, 篇幅就要超出许多。想想思路就好, 如果要实现, 可以自定义一个conn消息体, 然后练练手。



领资料

