

## 课程目标

01

工作日常

02

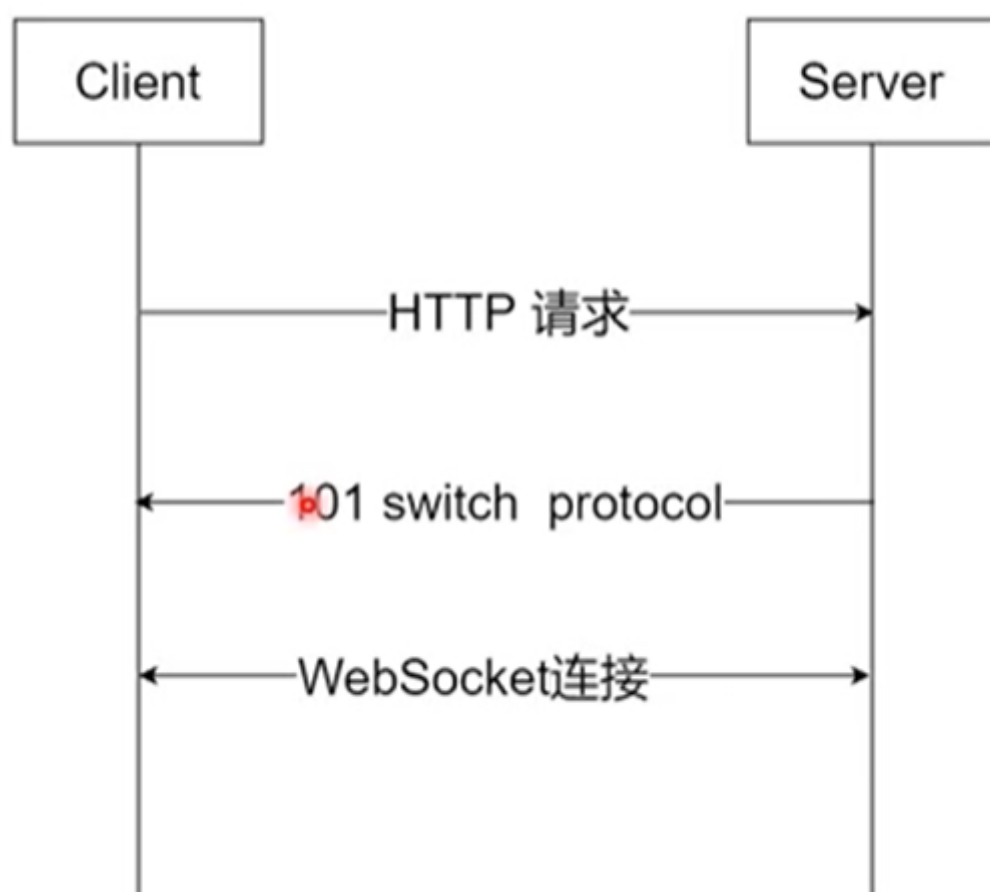
面试重点

03

程序训练

## WebSocket协议初探

websocket是一个基于TCP的通信协议，和http协议一样都是在应用层。但是websocket要复用HTTP的握手



client先以HTTP请求向Server发送一次请求，这个请求会带上一个头部，**客户端希望以websok cet协议，如果服务端实现了webSocket协议的,会给的client端返回一个101 切换的协议的头，客户端收到协议之后，以websocket的方式以跟服务端建立连接。下面就**

实现实战一个Socket连接的建立过程。观察浏览器/node端websocket连接建立的过程。观察对握手的处理和协议转换的过程。

### 客户端demo:

```
1 //简单的demo
2 <!DOCTYPE html>
3 <html lang="en">
4 <head>
5   <meta charset="UTF-8">
6   <title>webSocket聊天室</title>
7 </head>
8 <body>
9 <h1 style="text-align: center">web聊天</h1>
10 <button onclick="chat()">开始聊天</button>
11 <script>
12   function chat(){
13     let websocket = new WebSocket('ws://dev.websocket:8090')
14     //wss ->websocket Secure
15     //https- tls/ssl - tcp/ip
16     // wss - tls- tcp/ip
17     websocket.onopen = function () {
18       console.log('connection open');
19     }
20   }
21 </script>
22 </body>
23 </html>
```

### node原生实现服务端demo:

```
1 //网页服务
2 const express = require('express')
3 const path = require('path')
4 const app = express()
5
6 app.get('/', (req, res) => {
7   res.sendFile(path.resolve(__dirname, 'index.html'))
8 })
9 app.listen(3000, () => {
10   console.log('进入网页服务');
```

```

11 })
12
13
14 // websocket server
15 const net = require('net') //node自带的一个包
16 //net 这个服务是tcp/IP的一个服务和express的http服务还是有一定差别的
17 const server = net.createServer()
18 //解析头部的包
19 const parseHeader = require('parse-headers')
20
21 //crypto 加密包
22 const crypto = require('crypto')
23 server.on('connection', socket => {
24   //监听socket的数据来源
25   socket.on('data', (buffer) => {
26     //数据都是二进制的buffer
27     const str = buffer.toString()
28     //查看传输了几次消息 第一次是握手 后面是websokcet 封包后的传输
29     console.log("----message----");
30     //解析头部
31     const header = parseHeader(str)
32     console.log(header)
33
34     //crypto 加密
35     const sha1 = crypto.createHash('sha1')
36     sha1.update(header['sec-websocket-key'] + '258EAF5-E914-47DA-95CA-C5AB0DC85B11')
37     const acceptKey = sha1.digest('base64')
38     let response = `HTTP/1.1 101 Switch protocols
39 Upgrade:websocket
40 Connection:Upgrade
41 Sec-WebSocket-Accept:${acceptKey}
42
43 `
44     socket.write(response)
45   })
46 })
47
48
49 server.listen(8090, () => {
50   console.log('连接聊天服务')

```

## node原生实现服务端相关日志

```
[nodemon] restarting due to changes...
[nodemon] starting `node nodeWeb.js`

进入网页服务
连接聊天服务

从client发送websocket连接, 经过了两次消息传递

---message---
{
  'get / http/1.1': 'GET / http/1.1',
  host: 'dev.websocket:8090',
  connection: 'Upgrade',
  pragma: 'no-cache',
  'cache-control': 'no-cache',
  'user-agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.149 Safari/537.36',
  upgrade: 'websocket',
  origin: 'http://dev.websocket:3000',
  'sec-websocket-version': '13',
  'accept-encoding': 'gzip, deflate',
  'accept-language': 'zh-CN,zh;q=0.9,en;q=0.8',
  'sec-websocket-key': '2M61hRvaXm50gUIcnICckw==',
  'sec-websocket-extensions': 'permessage-deflate; client_max_window_bits'
}

---message---
{
  '0000 00 \t 0001 0001 0001 0001 00 00': '0000 00 \t 0001 0001 0001 0001 00 00'
}

[nodemon] restarting due to changes...
[nodemon] starting `node nodeWeb.js`
```

## 聊天室:node端

## 使用socket.io实现的node服务端demo

```
1 //网页服务
2 const express = require('express')
3 const path = require('path')
4 const app = express()
5
6 app.get('/', (req, res) => {
7   res.sendFile(path.resolve(__dirname, 'index.html'))
8 })
9
10 app.listen(3000, () => {
11   console.log('进入网页服务');
12 })
13
14 const socketIO = require('socket.io')(8090)
15
16 function broadcast(type, message, sender) {
17   for (let socket of users.keys()) {
```

```

17  socket.send({type,message,sender}) //每个socket对应每个用户的 发送相关给到
    客户端
18  }
19  }
20  const users = new Map()
21  socketIO.on('connect',(socket)=>{
22    // 创建socket的成本很低的,维护的成本不高,进来一个用户创建一个io
23    // 高成本的是计算 一个聊天室的能支撑多少个用户的,得看后面数据得存储 计算得效率
24    socket.on('message',data=>{
25      //这里用得switch 只是用作简单得demo 真正的登录还得需要用户token进行校验
26      switch (data.type) {
27        case 'LOGIN':
28          users.set(socket,{name:data.name}) //map中的每一个key都是一个socket
29          broadcast('LOGIN',`${data.name}加入了聊天`)
30          break
31        case 'CHAT':
32          const user = users.get(socket) //聊天的时候的时候 先获取的当前socket
33          // socket关联的map的值 是之前存的{name:data.name}
34          broadcast('CHAT',data.message,user.name)
35          break
36        }
37      })
38    })
39

```

## 观察node端的实现过程

思考：下载全部聊天记录应该如何实现？

1. 在demo实例情况下 用数组缓存起来即可
2. 是以什么样的形式给到client端 是以http协议 还是socket直接发呢？ 其实都可以 但是一个socket的复用的是一个TCP链接，如果聊天记录数据很多的时候,用socket 就会造成阻塞。这是可以使用另外一个连接，这个用socket的还是http这个就无关紧要了。

## 聊天室:web端(demo)

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>

```

```
4 <meta charset="UTF-8">
5 <title>websocket聊天室</title>
6 </head>
7 <body>
8 <h1 style="text-align: center">web聊天</h1>
9 <input type="text" id="chatValue" onkeydown="calAge(event)"; >
10 <button onclick="send()">发送</button>
11 <div id="textContent">
12
13 </div>
14 <script src="/socket.io.js"></script>
15 <script>
16 var socket = io.connect('ws://dev.websocket:8090',{transports: ['websoc
ket']});
17 const name = "user" +new Date().getTime()
18 socket.send({
19 type:'LOGIN',
20 name
21 })
22 socket.on('message',(data)=>{
23 console.log(data)
24 const {message,sender} = data
25 let senderName = sender
26 if(!sender){
27 senderName = '系统'
28
29 }else if(sender === name){
30 senderName = "我"
31 }
32 Message(senderName,message)
33 })
34
35
36 function Message(sender,message) {
37 let textDiv = document.createElement('div')
38 textDiv.innerHTML =`<span>${sender}</span><span>${message}</span>`
39 document.getElementById('textContent').append(textDiv)
40
41 }
42 function calAge(e) {
43 var evt = window.event || e;
```

```
44  if (evt.keyCode == 13) {
45    //回车后要干的业务代码
46    send()
47  }
48  }
49  function send() {
50
51    let chatInpt = document.getElementById("chatValue")
52    socket.send({
53      type: 'CHAT',
54      message: chatInpt.value
55    })
56    chatInpt.value = ""
57  }
58  </script>
59  </body>
60  </html>
```

## 课程小结

websocket握手和协议转换的过程很自然[我们可以看到用原生的node和用socket.io] 比较  
socket网络插座 为客户端/服务端提供通信机制