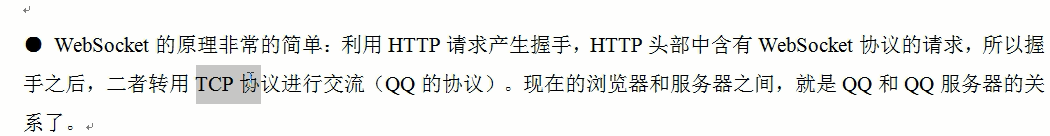
# 一 websocket使用

## 1 websocket简介

websocket性能高，且支持双向通信。



## 2 使用io.socket

使用 npm install socket.io安装该第三方模块，且该模块需要与http模块配合使用（原因见原理）。

客户端支持：

sock.on(‘connect’)

sock.on(‘disconnect’)

服务端：

sock.on(‘connection’)

sock.on(‘disconnect’)

### 2.1 客户端代码

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>Title</title>  
 <style>  
 #listText {  
 list-style: none;  
 border: solid 1px #2aabd2;  
 width: 400px;  
 height: 300px;  
 position: relative;  
 }  
 #listText .myText {  
 color: #5cb85c;  
 }  
 #listText #connectStat {  
 position: absolute;  
 left: auto;  
 bottom: 20px;  
 color: red;  
 display: none;  
 }  
 </style>  
</head>  
<body>  
<ul id="listText">  
 <span id="connectStat">服务器已断开，请检查网络....</span>  
</ul>  
<textarea rows="4" cols="60" id="sendText"></textarea>  
<button id="sendBtn">发送</button>  
<script src="http://localhost:8000/socket.io/socket.io.js" charset="utf-8"></script>  
<script>  
  
 let listText = document.querySelector('#listText');  
 let sendText = document.querySelector('#sendText');  
 let connectStat = document.querySelector('#connectStat');  
 let sendBtn = document.querySelector('#sendBtn');  
  
 let sock = io.connect('ws://localhost:8000');  
 sendBtn.onclick = function () {  
 *//发送一个名称为msg的消息* sock.emit('msg',sendText.value);  
 let myLi = document.createElement('li');  
 myLi.innerHTML = sendText.value;  
 myLi.className = 'myText';  
 listText.appendChild(myLi);  
 sendText.value = '';  
 }  
  
 *//已经连接* sock.on('connect',function () {  
 console.log('已连接');  
 connectStat.style.display = 'none';  
 });  
  
 *//接收消息* sock.on('serverMsg',function (str) {  
 let oLi = document.createElement('li');  
 oLi.innerHTML = str;  
 listText.appendChild(oLi);  
 });  
  
 *//断开连接* sock.on('disconnect',function () {  
 console.log('已断开');  
 connectStat.style.display = 'block';  
 });  
  
</script>  
</body>  
</html>

### 2.2 服务端代码

const http = require('http');  
const io = require('socket.io');  
  
let hs = http.createServer((req,res)=>{  
  
});  
hs.listen(8000);  
  
let ws = io.listen(hs);  
  
let sockArr = [];  
ws.on('connection',sock=>{  
  
 sockArr.push(sock);  
  
 *//接收消息* sock.on('msg',function (str) {  
 *//分发消息给所有客户端，除了发消息的客户端* sockArr.forEach(function (s) {  
 if(s != sock){  
 s.emit('serverMsg',str);  
 }  
 });  
 });  
  
 *//断开连接* sock.on('disconnect',function () {  
 *//从数组中删除该链接* let n = sockArr.indexOf(sock);  
 if(n != -1){  
 sockArr.splice(n,1);  
 }  
 });  
  
});

# 二 原生websocket

## 1 简介

websocket其实是前端H5的内容，Node等后台是自带socket服务的，而Node本身的socket很底层，在上述案例中使用了第三方包io.socket来处理。

使用源生net包来处理socket，得到的数据经过打印是：

|  |
| --- |
| GET / HTTP/1.1  Host: localhost:8000  Connection: Upgrade  Pragma: no-cache  Cache-Control: no-cache  Upgrade: websocket  Origin: http://localhost:63342  Sec-WebSocket-Version: 13  User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.119 Safari/537.36  Accept-Encoding: gzip, deflate, br  Accept-Language: zh-CN,zh;q=0.9,en;q=0.8  Cookie: \_ga=GA1.1.574841553.1517937000; Webstorm-63f64cd9=acfa6107-17dd-497f-85f8-29856fe9a6b2; io=-hHwrF9Hh\_HRNJ8LAAAB  Sec-WebSocket-Key: Z6eN3mB4Ip+FHChXL+jQ+g==  Sec-WebSocket-Extensions: permessage-deflate; client\_max\_window\_bits |

我们需要对该数据进行解析：每行数据都是以 : + 空格形式 存在的键值对

## 2 服务端代码

*//http,socket.io都是基于net模块制作的*const net = require('net');  
const crypto = require('crypto');  
*//前台不使用socket.io时，使用源生webscoket连接服务，会被http服务拒绝，所以这里使用net创建服务*net.createServer(sock=>{ *//使用http接收会拒绝* console.log('已经连接');  
  
 *//发现数据传输* sock.once('data',data=>{ *//握手的过程只有一次  
 //该过程即 握手 此时接收到了http头数据，但是我们没有http模块来解析* console.log('开始握手...');  
 *// console.log(data.toString());//打印该数据* let str = data.toString();  
 let lines = str.split('\r\n');  
 *//舍弃第一行和最后两行* lines = lines.slice(1,lines.length - 2);  
 *//用：+ 空格切割* let headers = {};  
 lines.forEach(line=>{  
 let [key,value] = line.split(': ');  
 headers[key] = value;  
 });  
  
 if(headers['Upgrade'] != 'websocket'){  
 console.log('其他协议',headers['Upgrade']);  
 sock.end();  
 } else if(headers['Sec-WebSocket-Version'] != 13){  
 console.log('只支持13版本的webscoket');  
 sock.end();  
 } else {  
  
 *// 此处为官方规定：sha1(key+mask)->base64=>client* let key=headers['Sec-Websocket-Key'];  
 let mask='258EAFA5-E914-47DA-95CA-C5AB0DC85B11';  
 let hash=crypto.createHash('sha1');  
 hash.update(key+mask);  
 let key2=hash.digest('base64');  
  
 *//发送key2 给客户端* sock.write(`HTTP/1.1 101 Switching Protocols\r\nUpgrade: websocket\r\nConnection: Upgrade\r\nSec-WebSocket-Accept: ${key2}\r\n\r\n`);  
  
 console.log('握手结束');  
  
 *//真正的数据，以后每次来数据都只走这一步* sock.on('data', data=>{  
  
 console.log('真正的数据' + data);  
  
 let FIN=data[0]&0x001;  
 let opcode=data[0]&0x0F0;  
 let mask=data[1]&0x001;  
 let payload=data[1]&0x0FE;  
  
 console.log(FIN, opcode);  
 console.log(mask, payload);  
 }  
 }  
 });  
  
 *//断开连接* sock.on('end',()=>{  
  
 });  
}).listen(8000);

## 3 客户端代码

<script>  
 let sock = new WebSocket('ws://localhost:8000');  
  
 *//模仿socket.io手工封装一个emit方法* sock.emit = function (name,...args) {  
 console.log('发送了：' + JSON.stringify({name,data:[...args]}));  
 sock.send(JSON.stringify({name,data:[...args]}));  
 };  
 *//已经连接* sock.onopen = function () {  
 console.log('连接上了');  
 sock.emit('msg',12,5);  
 };  
   
 *//发现数据传输* sock.onmessage = function () {  
 console.log('有数据传输');  
 };  
   
 *//断开连接* sock.onclose = function () {  
 console.log('断开连接');  
 };  
</script>

## 3 数据帧解析数据

计算机的数据都是由位构成的，1个位占据8。

0位 1位 2位 3位

0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7

+-+-+-+-+-------+-+-------------+-------------------------------+

|F|R|R|R| opcode|M| Payload len | Extended payload length |

|I|S|S|S| (4) |A| (7) | (16/64) |

|N|V|V|V| |S| | (if payload len==126/127) |

| |1|2|3| |K| | |

+-+-+-+-+-------+-+-------------+-------------------------------+

| Extended payload length continued, if payload len == 127 |

+-------------------------------+-------------------------------+

| |Masking-key, if MASK set to 1 |

+-------------------------------+-------------------------------+

| Masking-key (continued) | Payload Data |

+-------------------------------+-------------------------------+

| Payload Data continued ... |

+---------------------------------------------------------------+

| Payload Data continued ... |

+---------------------------------------------------------------+

FIN 1bit 是否最后一帧

RSV 3bit 预留

Opcode 4bit 帧类型

Mask 1bit 掩码，是否加密数据，默认必须置为1

Payload 7bit 长度

Masking-key 1 or 4 bit 掩码

Payload data (x + y) bytes 数据

Extension data x bytes 扩展数据

Application data y bytes 程序数据

# 三 点对点思路

上述案例中，所有人都在同一个请求下工作，属于一对多的通信方式，而单对单的模式，可以采用不同的请求地址来实现。比如所有人的通信在/chat下，A与B的单对单通信在/chat/ab下