

课堂目标

- 掌握 HTTP 协议
- 掌握 http 服务使用
- 掌握前后端通信技术 ajax、websocket 等
- 能解决常见 web 问题：跨域

1. HTTP 协议

1.1-http 协议详解

1.2 创建接口，http-server.js

```
const http = require("http");
const fs = require("fs");
http.createServer((req, res) => {
  const { method, url } = req;
  if (method === "GET" && url === "/") {
    fs.readFile("./index.html", (err, data) => {
      res.setHeader("Content-Type", "text/html");
      res.end(data);
    });
  } else if (method === "GET" && url === "/users") {
    res.setHeader("Content-Type", "application/json");
    res.end(JSON.stringify({ name: "tom", age: 20 }));
  }
})
.listen(3000);
```

1.3 请求接口，index.html

```
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
```

```
<script>
axios
.get("/users")
.then(res => res.data)
.then(users => console.log(users));
</script>
```

1.4 跨域：浏览器同源策略引起的接口调用问题

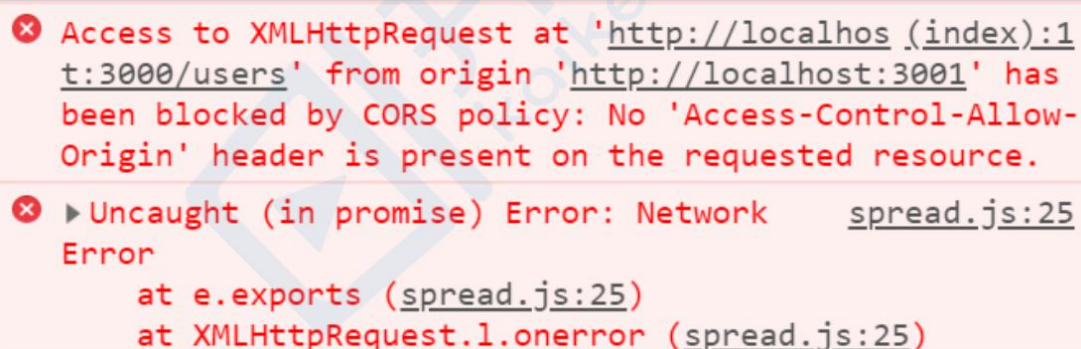
// 1.创建 http-server-2.js, 使用端口 3001

```
server.listen(3001);
```

// 2.index.html 中请求位于 3000 服务器的接口

```
axios.get("http://localhost:3000/users")
```

浏览器抛出跨域错误



```
✖ Access to XMLHttpRequest at 'http://localhost:3000/users' from origin 'http://localhost:3001' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.
✖ ▶ Uncaught (in promise) Error: Network Error      spread.js:25
    at e.exports (spread.js:25)
    at XMLHttpRequest.1.onerror (spread.js:25)
```

常用解决方案:

1. JSONP(JSON with Padding), 前端+后端方案, 绕过跨域

前端构造 script 标签请求指定 URL (由 script 标签发出的 GET 请求不受同源策略限制),

服务器返回一个函数执行语句, 该函数名称通常由查询参 callback 的值决定, 函数的参数为

服务器 返回的 json 数据。该函数在前端执行后即可获取数据。

2. 代理服务器

请求同源服务器, 通过该服务器转发请求至目标服务器, 得到结果再转发给前端。

前端开发中测试服务器的代理功能就是采用的该解决方案，但是最终发布上线时如果 web 应用和接口服务器不在一起仍会跨域。

3. CORS(Cross Origin Resource Share) - 跨域资源共享，后端方案，解决跨域

原理：cors 是 w3c 规范，真正意义上解决跨域问题。它需要服务器对请求进行检查并对响应头做相应处理，从而允许跨域请求。

具体实现：

- 响应简单请求: 动词为 get/post/head，没有自定义请求头，Content-Type 是 application/x-www-form-urlencoded, multipart/form-data 或 text/plain 之一，通过添加以下响应头解决：

```
res.setHeader('Access-Control-Allow-Origin', 'http://localhost:3001')
```

- 响应 preflight 请求，需要响应浏览器发出的 options 请求（预检请求），并根据情况设置响应头：

```
else if (method == "OPTIONS" && url == "/users") {
  res.writeHead(200, {
    "Access-Control-Allow-Origin": "http://localhost:3001",
    "Access-Control-Allow-Headers": "X-Token,Content-Type",
    "Access-Control-Allow-Methods": "PUT"
  });
  res.end();
}
```

该案例中可以通过添加自定义的 x-token 请求头使请求变为 preflight 请求

```
// index.html
axios.get("http://localhost:3000/users", {
  headers:{'X-Token':'jilei'}
})
```

则服务器需要允许 x-token，若请求为 post，还传递了参数：

```
// index.html
axios.post("http://localhost:3000/users", {foo:'bar'}, {headers:
{'X-Token':'jilei'}})
// http-server.js
else if ((method == "GET" || method == "POST") && url == "/users") {}
```

则服务器还需要允许 content-type 请求头

- 如果要携带 cookie 信息，则请求变为 credential 请求：

// 预检 options 中和/users 接口中均需添加

```
res.setHeader('Access-Control-Allow-Credentials', 'true');
```

<https://wangdoc.com/javascript/bom/cors.html>

观察 http 协议

```
curl -v http://www.baidu.com
```

https 协议

https 协议：超文本传输安全协议

是一种通过计算机网络进行安全通信的传输协议。HTTPS 经由 HTTP 进行通信，但利用 SSL/TLS 来加密数据包。HTTPS 开发的主要目的，是提供对网站服务器的身份认证，保护交换数据的隐私与完整性。这个协议由网景公司（Netscape）在 1994 年首次提出，随后扩展到互联网上。

简单来说，HTTPS 是 HTTP 的安全版，是使用 SSL/TLS 加密的 HTTP 协议。通过 TLS/SSL 协议的的身份验证、信息加密和完整性校验的功能，从而避免信息窃听、信息篡改和信息劫持的风险。

HTTPS 提供了加密 (Encryption)、认证 (Verification)、鉴定 (Identification) 三种功能。如下的解释中，假设是张三和李四在通讯。

- **私密性**(Confidentiality/Privacy):

也就是提供信息加密，保证数据传输的安全；保证信息只有张三和李四知道，而不会被窃听。

- **可信性**(Authentication):

身份验证，主要是服务器端的，确认网站的真实性，有些银行也会对客户端进行认证；用来证明李四就是李四。

- **完整性**(Message Integrity):

保证信息传输过程中的完整性，防止被修改；李四接收到的消息就是张三发送的。

HTTPS 就是在应用层和传输层中间加了一道验证的门槛以保证数据安全

SSL/TLS 协议

SSL(Secure Socket Layer) 安全套接层

TLS(Transport Layer Security) 传输层安全

HTTPS

利用 OpenSSL 创建 https 证书

```
mkdir https
cd https
openssl req -newkey rsa:2048 -new -nodes -x509 -days 3650 -keyout key.pem -out cert.pem
US
California
Francisco
noyanse
noyanse
localhost 这里可以填域名
noyanse@163.com
```

http 与 https 的区别

<https://www.ihuandu.com/help/zsk/46.html>

最简单的 socket 服务端

```
var net = require("net");
server1 = net.createServer(function(client){
    client.write('Hello World!\r\n');
});
server1.listen(9000);
```

什么是 Websocket

Websocket 协议也是基于 TCP 协议的，是一种双全工通信技术、复用 HTTP 握手通道。

Websocket 默认使用请求协议为:ws://,默认端口:80。对 TLS 加密请求协议为:wss://，端口:443。

特点

- 支持浏览器/Nodejs 环境
- 支持双向通信
- API 简单易用
- 支持二进制传输
- 减少传输数据量

长轮询

客户端向服务端发送 **xhr** 请求,服务端接收并 **hold** 该请求，直到有新消息 **push** 到客户端，

才会主动断开该连接。然后，客户端处理该 response 后再向服务端发起新的请求。以此类推。

HTTP1.1 默认使用长连接，使用长连接的 HTTP 协议，会在响应头中加入下面这行信息: Connection:keep-alive

短轮询:

客户端不管是否收到服务端的 response 数据，都会定时向服务端发送请求，查询是否有数据更新。

长连接

指在一个 TCP 连接上可以发送多个数据包，在 TCP 连接保持期间，如果没有数据包发送，则双方就需要发送心跳包来维持此连接。

连接过程: 建立连接——数据传输——...——(发送心跳包, 维持连接)——...——数据传输——关闭连接

短连接

指通信双方有数据交互时，建立一个 TCP 连接，数据发送完成之后，则断开此连接。

连接过程: 建立连接——数据传输——断开连接——...——建立连接——数据传输——断开连接

这里有一个误解，长连接和短连接的概念本质上指的是传输层的 TCP 连接，因为 HTTP1.1 协议以后，连接默认都是长连接。没有短连接说法(HTTP1.0 默认使用短连接)，网上大多数指的长短连接实质上说的就是 TCP 连接。

http 使用长连接的好处: 当我们请求一个网页资源的时候，会带有很多 js、css 等资源文件，如果使用的短连接的话，就会打开很多 tcp 连接，如果客户端请求数过大，导致 tcp 连接数量过多，对服务端造成压力也就可想而知了。

- 单工

数据传输的方向唯一，只能由发送方向接收方的单一固定方向传输数据。

- 半双工

即通信双方既是接收方也是发送方，不过，在某一时刻只能允许向一个方向传输数据。

- 全双工:

即通信双方既是接收方也是发送方，两端设备可以同时发送和接收数据。

单工、半双工和全双工 这三者都是建立在 TCP 协议(传输层上)的概念，不要与应用层进行混淆。

建立连接过程

Websocket 复用了 HTTP 的握手通道。指的是，客户端发送 HTTP 请求，并在请求头中带上 Connection: Upgrade、Upgrade: websocket，服务端识别该 header 之后，进行协议升级，使用 WebSocket 协议进行数据通信。

WebSocket 是基于 TCP 的一个独立的协议，它与 HTTP 协议的唯一关系就是它的握手请求可以作为一个 Upgrade request 经由 HTTP 服务器解析，且与 HTTP 使用一样的端口。WebSocket 默认对普通请求使用 80 端口，协议为 ws://，对 TLS 加密请求使用 443 端口，协议为 wss://

常用框架

- ws
- socket.io

Ws

前言

WebSocket 是一个通信的协议，分为服务器和客户端。服务器放在后台，保持与客户端的长连接，完成双方通信的任务。客户端一般都是实现在支持 HTML5 浏览器核心中，通过提供 JavascriptAPI 使用网页可以建立 websocket 连接。

WebSocket 事件：

WebSocket API 是纯事件驱动，通过监听事件可以处理到来的数据和改变的链接状态。客户端不需要为了更新数据而轮训服务器。服务端发送数据后，消息和事件会异步到达。

WebSocket 编程遵循一个异步编程模型，只需要对 WebSocket 对象增加回调函数就可以监听事件。你也可以使用 `addEventListener()` 方法来监听。而一个 WebSocket 对象分四类不同事件。

open:

一旦服务端响应 WebSocket 连接请求，就会触发 open 事件。响应的回调函数称为

`onopen`。

```
ws.onopen = function(e) {  
  console.log("Connection open...");  
};
```

open 事件触发的时候，意味着协议握手结束，WebSocket 已经准备好收发数据。如果你的应用收到 open 事件，就可以确定服务端已经处理了建立连接的请求，且同意和你的应用通信。

Message:

当消息被接受会触发消息事件，响应的回调函数叫做 onmessage。如下：

// 接受文本消息的事件处理实例：

```
ws.onmessage = function(e) {
    if(typeof e.data === "string"){
        console.log("String message received", e, e.data);
    } else {
        console.log("Other message received", e, e.data);
    }
};
```

除了文本消息，WebSocket 消息机制还能处理二进制数据，有 Blob 和 ArrayBuffer 两种类型，在读取到数据之前需要决定好数据的类型。

// 设置二进制数据类型为 blob（默认类型）

```
ws.binaryType = "blob";// Event handler for receiving Blob messages
```

```
ws.onmessage = function(e) {
    if(e.data instanceof Blob){
        console.log("Blob message received", e.data);
        var blob = new Blob(e.data);
    }
};
```

//ArrayBuffer

```
ws.binaryType = "arraybuffer";
```

```
ws.onmessage = function(e) {
    if(e.data instanceof ArrayBuffer){
        console.log("ArrayBuffer Message Received", + e.data);// e.data 即 ArrayBuffer 类型
        var a = new Uint8Array(e.data);
    }
};
```

Error

如果发生意外的失败会触发 error 事件，相应的函数称为 onerror,错误会导致连接关闭。

如果你收到一个错误事件，那么你很快会收到一个关闭事件，在关闭事件中也许会告诉你错

误的原因。而对错误事件的处理比较适合做重连的逻辑。

//异常处理

```
ws.onerror = function(e) {
```

```
        console.log("WebSocket Error: ", e);
    };
```

Close

不言而喻，当连接关闭的时候会触发这个事件，对应 `onclose` 方法，连接关闭之后，服务端和客户端就不能再收发消息。

当然你可以调用 `close` 方法断开与服务端的链接来触发 `onclose` 事件，

```
ws.onclose = function(e) {
    console.log("Connection closed", e);
};
```

WebSocket 方法：

WebSocket 对象有两个方法：`send()`

`send()`:

一旦在服务端和客户端建立了全双工的双向连接，可以使用 `send` 方法去发送消息，

```
//发送一个文本消息
ws.send("Hello WebSocket!");
```

当连接是 `open` 的时候 `send()` 方法传送数据，当连接关闭或获取不到的时候会抛出异常。一个通常的错误是人们喜欢在连接 `open` 之前发送消息。如下所示：

```
// 这不会工作
var ws = new WebSocket("ws://echo.websocket.org")
ws.send("Initial data");
```

正确的姿势如下，应该等待 `open` 事件触发后再发送消息。

```
var ws = new WebSocket("ws://echo.websocket.org")
ws.onopen = function(e) {
    ws.send("Initial data");
}
```

如果想通过响应别的事件去发送消息，可以检查 `readyState` 属性的值为 `open` 的时候来实现。

```
function myEventHandler(data) {  
  if (ws.readyState === WebSocket.OPEN) {  
    //open 的时候即可发送  
    ws.send(data);  
  } else {  
    // Do something else in this case.  
  }  
}
```

发送二进制数据：

```
// Send a Blob  
var blob = new Blob("blob contents");  
ws.send(blob);  
  
// Send an ArrayBuffer  
var a = new Uint8Array([8,6,7,5,3,0,9]);  
ws.send(a.buffer);
```

`Blob` 对象和 JavaScript File API 一起使用的时候相当有用，可以发送或接受文件，大部分的多媒体文件，图像，视频和音频文件。这一章末尾会结合 File API 提供读取文件内容来发送 WebSocket 消息的实例代码。

close()

使用 `close` 方法来关闭连接，如果连接以及关闭，这方法将什么也不做。调用 `close` 方法只后，将不能发送数据。

```
ws.close();
```

`close` 方法可以传入两个可选的参数，`code` (numerical) 和 `reason` (string)，以告诉服务端为什么终止连接。第三章讲到关闭握手的时候再详细讨论这两个参数。

```
// 成功结束会话  
ws.close(1000, "Closing normally");//1000 是状态码，代表正常结束。
```

websocket 服务端的搭建

WebSocket 是基于事件驱动，支持全双工通信。下面通过三个简单例子体验一下。

1.简单安装

安装 ws 模块，npm install ws

ws: 是 nodejs 的一个 WebSocket 库，可以用来创建服务。 <https://github.com/websockets/ws>

WebSocket 协议定义了两种 URL 方案, WS 和 WSS 分别代表了客户端和服务端之间未加密和加密的通信。WS(WebSocket)类似于 Http URL, 而 WSS (WebSocket Security) URL 表示连接是基于安全传输层 (TLS/SSL) 和 https 的连接是同样的安全机制。

2.服务端 server.js 配置

在项目里面新建一个 server.js，创建服务，指定 8181 端口，将收到的消息 log 出来。

```
var WebSocketServer = require('ws').Server,
wss = new WebSocketServer({ port: 8181 });
wss.on('connection', function (ws) {
  console.log('client connected');
  ws.on('message', function (message) {
    console.log(message);
  });
});
```

服务端完整代码如下：

```
var WebSocketServer = require('ws').Server,
wss = new WebSocketServer({ port: 8181 });
wss.on('connection', function (ws) {
  console.log('client connected');
  var timer = null;

  function repeatFun(str){
    clearInterval(timer);
    timer = setInterval(()=>{
      var ran = Math.random();
      ws.send(10 * ran+str);
    }, 1000);
  }
  repeatFun('');
});
```

```

    },2000);
  }
  // repeatFun();

  ws.on('message', function (message) {
    console.log('receive:',message);
    repeatFun(message);
  });
  ws.on('close',function(){
    clearInterval(timer);
  })
});

```

3.客户端 client.html 建立 WebSocket 链接

我们需要通过调用 WebSocket 构造函数来创建一个 WebSocket 连接，构造函数会返回一个 WebSocket 实例，**可以用来监听事件**。这些事件会告诉你什么时候连接建立，什么时候消息到达，什么时候连接关闭了，以及什么时候发生了错误。

WebSocket 的构造函数需要一个 URL 参数和一个可选的协议参数（一个或者多个协议的名字），协议的参数例如 XMPP (Extensible Messaging and Presence Protocol)、SOAP (Simple Object Access Protocol) 或者自定义协议。而 URL 参数需要以 WS://或者 WSS://开头，例如：ws://www.websocket.org，如果 URL 有语法错误，构造函数会抛出异常。

在页面上建立一个 WebSocket 的连接。用 send 方法发送消息。

```

var ws = new WebSocket("ws://localhost:8181");
ws.onopen = function (e) {
  console.log('Connection to server opened');
}

```

```
function sendMessage() {  
    ws.send($('#message').val());  
}
```

完整的页面代码如下：

```
<body>  
    <input class="form-control" type="text" name="message" id="message"  
        placeholder="Type text to echo in here" value="" />  
    <button type="button" id="send" class="btn btn-primary" onclick="sendMessage();">  
        Send!  
    </button>  
<script>  
var ws = new WebSocket("ws://localhost:8181");  
    ws.onopen = function (e) {  
        console.log('Connection to server opened');  
        ws.send('laney');  
        console.log("sened a laney");  
    }  
    function sendMessage() {  
        var message = document.getElementById('message');  
        ws.send(message.value);  
    }  
    //收到请求  
    ws.onmessage = function (e) {  
        console.log(e.data);  
    }  
</script>  
</body>
```

运行之后如下，服务端即时获得客户端的消息。

socket.io

socket.io 就是众多 websocket 库中的一种，它并不像其它库那样简单地实现了一下 websocket，而是在 websocket 外面包裹了厚厚的一层。普通的 websocket（例如 ws 库）只需要服务端就够了，socket.io 自定义了一种基于 websocket 的协议，所以 socket.io 的

服务端和客户端必须配套。简言之，如果服务端使用 socket.io，那么客户端就没得选了，必然也用 socket.io 的客户端。

概述：

socket.io 是 基于 engine.io 进行封装的库

socket.io 是基于 Websocket 的 Client-Server 实时通信库。

socket.io 底层使用 engine.io 封装了一层协议。

socket.io 与 engine.io 的一大区别在于，socket.io 并不直接提供连接功能，而是在 engine.io 层提供。

socket.io 提供了一个房间(Namespace)概念。当客户端创建一个新的长连接时，就会分配一个新的 Namespace 进行区分。

socket.io 事件

socket.io 服务器的系统事件

io.on('connection' , callback): 有新 Socket 连接建立时

socket.on('message' , callback): 当有 socket..send()方法发送完信息后触发

socket.on('disconnect' , callback): 当连接断开时触发

客户端的系统事件

socket.io.on('open' , callback): 当 socket 客户端开启与服务器的新连接时触发

socket.io.on('connect' , callback):当 socket 客户端连接到服务器后触发

socket.io.on('connect_timeout' , callback):当 socket 客户端与服务器连接超时时触发

socket.io.on('connec_errort' , callback):当 socket 客户端连接服务器失败时触发
socket.io.on('connec_attemptt' , callback):当 socket 客户端尝试重新连接到服务器后触发
socket.io.on('reconnect' , callback):当 socket 客户端重新连接到服务器后触发
socket.io.on('reconnect_error' , callback):当 socket 客户端重新连接服务器失败后触发
socket.io.on('reconnect_fail' , callback):当 socket 客户端重新连接到服务器失败后触发
socket.io.on('close' , callback):当 socket 客户端关闭与服务器的连接后触发

事件触发

- 服务端触发
- 客户端触发

```
/*服务端*/
io.on('connection', function (socket) {
  socket.emit('customEvent', message);
});
<!--客户端-->
<script>
var socket = io();
socket.emit('customEvent', mesaage);
</script>
```

事件处理

```
/*服务端*/
io.on('connection', function (socket) {
  socket.on('customEvent', (message) => {
    /*逻辑代码*/
  })
});

//通过 broadcast 属性可以给处理当前连接外的所有连接的 socket 客户端发送事件:
io.on('connection',(socket)=>{
  socket.broadcast.emit('customEvent', meassage);
});

<!--客户端-->
<script>
var socket =io();
socket.on('eventName',(message)=>{ /逻辑代码*/ });
</script>
```

socket.io 命名空间:对 socket 连接进行划分命名空间,可以对不同的 socket 的连接分类管理,而且不需要新建 socket 服务器实例.

```
//服务端命名空间
io.of('/someNameSpace').on('connection', (socket)=>{
  socket.on('customEvent', (message)=>{ /*逻辑代码*/ });
});
```

```
<!--客户端命名空间-->
<script>
  var someSocket = io('/someNameSpace');
  someSocket.on('customEvent', (message)=>{
    /*逻辑代码*/
  });
</script>
```

重要的 api

on() 和 jquery 一样用于绑定事件
emit() 用于绑定发送事件
httpServer (http.Server) 需要绑定的服务。

socket.io 几个重要的事件

connect 客户端的 socket 连接实例
connection 用法同 Event: 'connect'。
disconnect 关闭对客户端的链接,如果 close 的值为 true,则关闭下行连接,否则,仅仅关闭命名空间。

1.简单安装

npm install socket.io

2.服务端 server.js 配置

```
var app = require('http').createServer(handler)
var io = require('socket.io')(app);
var fs = require('fs');

app.listen(8083);

function handler(req, res) {
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end('<h1>Hello Socket Lover!</h1>');
}

io.on('connection', function (socket) {

  socket.emit('news', { data : 'server world' });

  socket.on('test_event', function (data) {
    console.log('receive data from client. data : ' + JSON.stringify(data));

    var str = data.data + "(from server by " + new Date().getTime() + ")";
    socket.emit('news', { data : str});
  });

  socket.on('disconnect', function () {
    console.log('websocket close. -- server log');
  });

});
```

3.客户端 client.html 建立 WebSocket 链接

HTML:

```
<!DOCTYPE <!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <title>websocket 测试</title>
```

```

    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" type="text/css" href="main.css" />
    <script src="./socket.io.dev.js"></script>
</head>
<body>

<div id="wrapper">
  <div id="main">
    <textarea readonly="readonly" id="txtConsole"></textarea>
  </div>
  <div id="footer">
    <div class="ipt_container">
      <input type='text' class="ipt" placeholder="请输入内容..." id="chatIpt"/>
    </div>
    <div class="btn_container">
      <a href="javascript:;" class="btn" onclick="sendMsgHandler()">发送</a>
    </div>
  </div>
</div>
<script type="text/javascript" src='./socket.js'></script>
</body>
</html>

```

注意需要引入 socket.io.dev.js 库，如果在项目中，可以通过 npm 安装

Socket.js

```

// Create SocketIO instance, connect
var socket = io.connect('ws://localhost:8083');

// Add a connect listener
socket.on('connect', function() {
  showMessage('Client has connected to the server!');
});

// Add a connect listener
socket.on('news', function(data) {
  var str = 'Received data from server. data : ' + JSON.stringify(data);
  showMessage(str);
});

```

```

// Add a disconnect listener
socket.on('disconnect', function() {
    showMessage('The client has disconnected');
});

// Sends a message to the server via sockets
function sendMessageToServer(msgData) {
    socket.emit('test_event', msgData);
}

var delayScrollTimer = null;
function showMessage(str) {
    var txt = document.getElementById('txtConsole');
    txt.value = txt.value + '\n' + str;

    if (delayScrollTimer) {
        clearTimeout(delayScrollTimer);
        delayTimer = null;
    }

    delayScrollTimer = setTimeout(function() {
        txt.scrollTop = txt.scrollHeight;
    }, 10);
}

// 向服务端发送数据
function sendMsgHandler() {
    var elem = document.getElementById('chatplpt');
    var iptValue = elem.value;
    sendMessageToServer({data : iptValue});
    elem.value = "";
    elem.focus();
}

```