

# WebSocket

WebSocket 是一种网络通信协议，它能在单个 TCP 连接上建立全双工通信 (Full-duplex communication)

简单来说，它让客户端（比如浏览器）和服务器之间可以**双向、持续地实时通信**，而不需要像 HTTP 那样每次都发送一个新的请求。

对比传统的HTTP请求

## 传统的 HTTP 请求

- **单向通信**：客户端发起请求，服务器返回响应。
- **无状态**：每次请求都是独立的，服务器不会保留之前的连接信息。
- **开销大**：每次请求都要携带完整的 HTTP 请求头，这会产生额外的网络开销。

## WebSocket 连接

### 1. 建立连接（握手）：

- 客户端通过一个特殊的 HTTP 请求（带有 Upgrade: websocket 请求头）向服务器发起连接请求。
- 服务器如果支持 WebSocket，会返回一个特殊的 HTTP 响应，表示同意升级协议。
- 这次“握手”成功后，客户端和服务器之间的协议就从 HTTP 升级到了 WebSocket，一个持久的 TCP 连接就此建立。

### 2. 数据传输（帧）：

- 连接建立后，客户端和服务器可以随时、双向地发送和接收数据。
- 数据以“帧”（frame）的形式传输，这比 HTTP 请求的数据包小得多，开销很小。

### 3. 关闭连接：

- 客户端或服务器可以主动发送一个控制帧来关闭连接。

## WebSocket 的优点

- **实时性高：**一旦连接建立，服务器可以主动向客户端推送数据，无需客户端发起请求，非常适合实时聊天、在线游戏、股票行情等场景。
- **双向通信：**客户端和服务端可以同时发送和接收数据，非常灵活。
- **开销小：**握手成功后，数据传输的开销极小，大大节省了带宽和服务端资源。
- **持久连接：**一次连接可以长时间保持，避免了重复建立连接的开销。

### 适用场景

- **实时聊天应用：**消息能即时推送到所有在线用户。
- **在线多人游戏：**玩家的实时操作和位置能快速同步。
- **股票行情、金融交易：**价格变动能毫秒级更新。
- **协同编辑：**多个用户同时编辑一个文档时，能实时看到对方的修改。
- **通知系统：**服务器有新通知时，能立即推送给客户端。

## WebSocket 的基本使用

浏览器内置了 WebSocket 对象

### 1. 创建 web socket 实例

你需要创建一个 `WebSocket` 对象，并传入你后端 WebSocket 服务器的 URL。URL 的协议是 `ws` (非加密) 或 `wss` (加密)。

JavaScript

```
const ws = new WebSocket('ws://localhost:8080'); // 替换成你的服务器地址
```

## 2. 监听事件

WebSocket 对象有几个重要的事件，你需要监听它们来处理不同的状态和数据。

**onopen**：当连接成功建立时触发。这是你可以开始发送数据的地方。

```
ws.onopen = function() {  
  console.log('WebSocket 连接已打开');  
  ws.send('你好，服务器! '); // 连接成功后，可以向服务器发送数据  
};
```

**onmessage**：当从服务器接收到数据时触发。数据包含在 `event.data` 中。

```
ws.onmessage = function(event) {  
  console.log('收到服务器消息:', event.data);  
};
```

**onclose**：当连接关闭时触发。

```
ws.onclose = function() {  
  console.log('WebSocket 连接已关闭');  
};
```

**onerror**：当发生错误时触发。

```
ws.onerror = function(error) {  
  console.error('WebSocket 错误:', error);  
};
```

## 3. 发送和接收数据

**发送数据**：使用 `ws.send()` 方法向服务器发送数据。你可以发送字符串、Blob、ArrayBuffer 等。

**接收数据**：在 `onmessage` 事件中接收数据。服务器通常会返回 JSON 格式的字符串，你需要使用 `JSON.parse()` 来解析它。

## 完整示例

```
// 替换成你的 WebSocket 服务器地址
const ws = new WebSocket('ws://localhost:8080');

ws.onopen = function() {
  console.log('连接成功');
  // 连接成功后发送一个消息
  ws.send(JSON.stringify({
    type: 'message',
    payload: '前端连接成功, 准备接收数据!'
  }));
};

ws.onmessage = function(event) {
  // 接收到服务器消息, 通常是 JSON 字符串
  const data = JSON.parse(event.data);
  console.log('收到消息:', data);

  if (data.type === 'chat') {
    // 渲染聊天消息
    console.log('新聊天消息:', data.content);
  }
};

ws.onclose = function() {
  console.log('连接已断开');
};

ws.onerror = function(error) {
  console.error('发生错误:', error);
};

// 示例: 在某个按钮点击事件中发送消息
document.getElementById('sendButton').onclick = function() {
  const input = document.getElementById('messageInput');
  const message = {
    type: 'chat',
    content: input.value
  };
  ws.send(JSON.stringify(message));
  input.value = '';
};
```

## 后端服务器

以Node.js(ws库)为例

```
// 安装: npm install ws
const WebSocket = require('ws');

// 启动 WebSocket 服务端, 监听 8080 端口
const wss = new WebSocket.Server({ port: 8080 });

wss.on('connection', (ws) => {
  console.log('客户端已连接');

  // 收到消息
  ws.on('message', (message) => {
    console.log('收到客户端消息:', message);

    // 回复客户端
    ws.send(`服务端已收到: ${message}`);
  });

  // 连接关闭
  ws.on('close', () => {
    console.log('客户端已断开');
  });
});
```

通信示意图

浏览器(前端)

```
const socket =  
  new WebSocket("ws://...")
```

WebSocket API

send("你好")



消息经过网络

WebSocket 协议



Node.js (后端)

```
npm install ws  
const WebSocket = require('ws')
```

ws 服务端对象

on('message')



收到: "你好"

回复: "服务端已收到"



浏览器(前端)

```
socket.onmessage(...)  
收到: "服务端已收到"
```

# SSE

## Server Sent Events

SSE (Server-Sent Events, 服务端推送事件) 也是前端实时通信里很重要的一种方式。

- SSE 是一种基于 **HTTP 协议** 的单向通信机制。
- **特点**: 服务端可以主动把消息推送给客户端, 但客户端不能主动发消息给服务端。
- 你可以把它理解为 **浏览器自动保持一个长连接, 服务端随时能推送数据**。

## 2. 和 WebSocket 的区别

特点	SSE (Server-Sent Events)	WebSocket
方向	单向 (服务端 → 客户端)	双向 (客户端 ↔ 服务端)
协议	HTTP ( <code>text/event-stream</code> )	独立协议 (ws:// 或 wss://)
兼容性	原生支持, 基于 HTTP	需要 WebSocket API 支持
适用场景	消息推送、通知、股票/天气更新	聊天、游戏、协作编辑、交易系统

只需要服务端 → 客户端推送 → 用 SSE 就够了。

需要双向通信 (比如聊天室) → 必须用 WebSocket。

## SSE的基本使用

在前端, 你可以使用浏览器内置的 `EventSource` API 来接收 SSE 数据。和 `WebSocket` 一样, 这个 API 是原生支持的, 不需要安装任何库。



## 1. 创建EventSource示例

你需要创建一个 `EventSource` 对象，并传入你后端 SSE 服务器的 URL。

JavaScript

```
const eventSource = new EventSource('http://localhost:3000/events'); // 替换成你的服务器URL
```

这里的 URL 指向的是服务器上的一个端点，这个端点会以 `text/event-stream` 的 MIME 类型持续发送数据流。

## 2. 监听对象

`EventSource` 对象有几个重要的事件，你需要监听它们来处理不同的状态和数据。

`onopen`: 当连接成功建立时触发。

JavaScript

```
eventSource.onopen = function() {  
  console.log('SSE 连接已打开');  
};
```

`onmessage`: 当从服务器接收到**无名事件**（默认事件）的数据时触发。数据包含在 `event.data` 中。

```
eventSource.onmessage = function(event) {  
  console.log('收到服务器消息:', event.data);  
  // 通常服务器会发送 JSON 字符串，你需要解析它  
  const data = JSON.parse(event.data);  
  console.log('解析后的数据:', data);  
};
```

`onerror`: 当连接发生错误时触发。浏览器会自动尝试重新连接。

```
eventSource.onerror = function(error) {  
  console.error('SSE 错误:', error);  
  // 浏览器会自动重连，你也可以在这里做一些错误处理  
};
```

### 3.处理具名事件

SSE 还有一个非常实用的功能，就是可以发送**具名事件**（named events）。服务器可以在数据流中指定事件的名称，前端就可以监听特定的事件。

• 服务器发送的数据格式（示例）：

```
event: update-user-count
data: {"count": 100}

event: new-message
data: "Hello world!"
```

前端监听具名事件：

```
// 监听名为 'update-user-count' 的事件
eventSource.addEventListener('update-user-count', function(event) {
  const data = JSON.parse(event.data);
  console.log('当前在线用户数:', data.count);
});

// 监听名为 'new-message' 的事件
eventSource.addEventListener('new-message', function(event) {
  console.log('收到新消息:', event.data);
});
```

SSE vs WebSocket vs 长轮询			
特性	SSE (Server-Sent Events)	WebSocket	长轮询 (Long Polling)
通信方向	服务器 -> 客户端 (单向)	双向	服务器 -> 客户端 (单向)
协议	基于 HTTP	独立于 HTTP	基于 HTTP
实现难度	非常简单，前端只需 EventSource	较复杂，需要独立的服务 器	较简单，但需处理超时和重连
适用场景	服务器推送：通知、直播数据、股票行情等	双向实时通信：聊天、在线游戏、协同编辑	数据更新不频繁，且服务器无法支持 WebSocket 的场景
优缺点	优点：简单、兼容性好；缺点：无法双向通信，二进制数据支持不佳	优点：双向、高效；缺点：实现复杂，协议独立	优点：兼容性好；缺点：效率低、延迟高

## 服务端

```
const express = require("express");
const app = express();

app.get("/sse", (req, res) => {
  // 设置 SSE 响应头
  res.setHeader("Content-Type", "text/event-stream");
  res.setHeader("Cache-Control", "no-cache");
  res.setHeader("Connection", "keep-alive");

  // 每隔 2 秒推送一条消息
  let count = 0;
  setInterval(() => {
    count++;
    res.write(`data: 这是第 ${count} 条消息\n\n`);
  }, 2000);
});

app.listen(3000, () => console.log("SSE 服务运行在 http://localhost:3000"));
```

前端只要访问 /sse，就会源源不断收到消息。

## WebRTC

### WebRTC 是什么？

全称：Web Real-Time Communication（网页实时通信）。

是浏览器内置的一套 **点对点（P2P）通信协议和 API**。

**最大特点**：浏览器之间可以直接传输 **音视频流** 和 **数据**，不需要中间服务器转发（节省延迟和带宽）。

简单说，WebRTC = 浏览器自带的“实时语音/视频/数据通道”。

# WebRTC 能做什么？

视频通话（Zoom、Google Meet、腾讯会议）

语音通话（微信语音、Discord）

实时协作（共享白板、远程桌面）

**P2P 文件传输**（比如 Firefox Send 以前就是用的 WebRTC DataChannel）

## 3. WebRTC 和 WebSocket / SSE 有啥区别？

技术	主要用途	特点	
WebSocket	实时消息（聊天、游戏）	双向，基于 TCP	
SSE	服务端推送消息	单向，基于 HTTP	
WebRTC	实时音视频、P2P 数据	P2P，支持音视频流，延迟低	

- 👉 如果只是 **文字消息**，WebSocket 足够；
- 👉 如果要 **视频通话 / 实时语音**，必须用 WebRTC。

## P2P协议

对于WebRTC来说，**理解P2P（点对点）协议非常重要，因为它和我们前端日常接触的HTTP协议完全不同。**

## 什么是P2P（点对点）协议？

P2P，全称 Peer-to-Peer，中文译为**点对点**。它是一种网络架构模型，在这种模型中，网络中的所有设备（或称为“对等端”，英文是 peers）都具有同等的地位，既是服务的**提供者**，也是服务的**消费者**。

你可以想象一个文件分享网络，比如老式的BT下载。当你下载一个电影时，你不是从一个中央服务器下载，而是**同时从其他正在下载或已经下载完电影的用户那里获取数据块**。同时，你也在把你已经下载好的数据块分享给其他用户。

**P2P 协议的核心特点：**

- **去中心化或弱中心化**：P2P网络没有一个唯一的中心服务器来管理所有数据和连接。这使得网络更加健壮，因为它不会因为一个中心服务器的故障而崩溃。
- **直接通信**：对等端之间可以直接相互通信和交换数据，无需通过一个中心服务器作为中转站。这大大提高了通信效率，尤其是对于音视频这样的实时数据流。

区别

P2P协议和HTTP协议有什么不同？		
HTTP（超文本传输协议）是互联网的基石，但它的工作模式和P2P协议完全相反。		
特性	P2P（点对点）协议	HTTP（客户端-服务器）协议
架构模型	去中心化或弱中心化：所有设备都是对等的，既能提供服务也能请求服务。	中心化：严格的客户端-服务器模型。客户端发起请求，服务器提供响应。
通信模式	点对点直接通信：两个对等端（例如，两个浏览器）之间直接建立连接并传输数据。	中转通信：所有通信都必须通过一个中心服务器。客户端向服务器请求资源。
连接生命周期	长连接：为了维持实时通信，连接通常需要保持长时间。	短连接：通常是无状态的，一次请求-响应后连接就关闭。
数据流向	双向：对等端之间可以自由地发送和接收数据，数据流是双向的。	单向：数据流通常从服务器流向客户端（响应），客户端只能通过请求来获取。
主要应用	实时通信：VoIP、视频会议、文件分享、在线游戏。	网页浏览：获取网页、图片、API数据、下载文件。

WebRTC中的P2P

在WebRTC中，P2P协议是实现实时音视频通话的关键。当你使用WebRTC进行视频通话时：

1. 信令服务器（Signal Server）：首先，你需要一个中心服务器（信令服务器）来帮助双方交换信息，例如IP地址、网络类型等。这就像是你们互相留联系方式的过程。这个阶段是中心化的。

2. 建立P2P连接：一旦双方获得了联系方式，它们就会尝试直接建立P2P连接。

3. 直接通信：P2P连接建立后，音视频数据流将直接在两个浏览器之间传输，完全绕过了信令服务器。

这就是为什么WebRTC的通话质量和效率都非常高，因为它消除了中心服务器作为中转的延迟。理解了P2P，你就理解了WebRTC的核心优势。

P2P的数据结构

P2P 协议的数据结构是去中心化的。它的核心是网络中的节点（对等端）\*和\*节点间直接传输的数据块。

**没有统一的请求-响应格式：**P2P 协议不遵循 HTTP 那种严格的“请求-响应”模式。相反，它的数据是以**小块（chunks）**的形式在对等端之间直接传输的。

**元数据和身份标识：**在 WebRTC 这类 P2P 应用中，数据流（如音视频流）被分割成一个个小的 **UDP 数据包**。每个数据包都包含**元数据（如序列号、时间戳）**，以及一个**标识符来确定它属于哪个流**。这些元数据是用于**重组和同步数据的**，而不是像 HTTP 那样用于请求或路由。

**信令数据：**虽然 P2P 通信本身没有中心服务器，但在连接建立阶段，会有一个**信令服务器**来帮助**对等端交换元数据**，例如彼此的 IP 地址和网络能力。这些**信令数据通常是 JSON 格式**，通过 WebSockets 或其他实时协议进行传输。但这部分数据是**用于建立连接的**，而不是实际的 P2P 内容数据。

## 前端的基本使用

**WebRTC 在前端的使用比前面提到的 WebSocket 和 SSE 要复杂得多**，因为它涉及多个步骤和浏览器 API，但掌握了它，你就能实现强大的音视频实时通信功能。

WebRTC 的核心思想是**点对点（P2P）通信**，这意味着音视频数据直接在两个浏览器之间传输。但要实现这一点，**你需要一个“中间人”来帮助建立连接**。

整体流程可以分为三个主要阶段：

1. **信令（Signaling）**：交换元数据，让双方知道彼此的存在和网络信息。
2. **连接建立（Connection）**：利用信令信息，建立 P2P 连接。
3. **数据传输（Data Transfer）**：在 P2P 连接上直接传输音视频数据。

### 1. 创建信令

WebRTC 本身没有内置的信令机制。你需要自己实现一个**信令服务器**来协调两个对等端（Peers）。通常，前端会使用 WebSocket 来实现信令，因为它能提供双向、持久的通信。

#### 信令的作用：

- **交换会话描述（SDP）**：包含对等端的音视频能力、支持的编解码器等信息。
- **交换 ICE Candidates**：包含对等端的网络地址信息（IP 地址、端口等），用于寻找最佳的 P2P 连接路径。

## 前端代码示例（伪代码）：

JavaScript



```
// 连接到信令服务器，这里使用 WebSocket
const signaling = new WebSocket('wss://your-signaling-server.com');
```

```
signaling.onmessage = async (message) => {
  const data = JSON.parse(message.data);
  // 步骤 2 和 3 会用到
  // 处理 SDP Offer、Answer 和 ICE Candidates
};
```

```
// 发送信令消息
function sendSignalingMessage(message) {
  signaling.send(JSON.stringify(message));
}
```

发送信令消息

## 2. 媒体和连接的准备

在信令之后，你需要准备好本地的音视频流，并创建 `RTCPeerConnection` 对象来处理 P2P 连接。

**获取本地媒体流：**使用 `navigator.mediaDevices.getUserMedia()` API 来请求用户的摄像头和麦克风权限。

**创建 `RTCPeerConnection`：**这是 WebRTC 的核心 API，它负责管理连接、媒体流和传输。

```

let localStream;
let peerConnection;

// 1. 获取本地音视频流
async function getLocalStream() {
  try {
    localStream = await navigator.mediaDevices.getUserMedia({ video: true, audio: true });
    document.getElementById('localVideo').srcObject = localStream;
  } catch (err) {
    console.error('获取媒体流失败:', err);
  }
}

// 2. 创建 PeerConnection
function createPeerConnection() {
  // `stun` 和 `turn` 服务器用于 NAT 穿透
  const iceServers = [
    { urls: 'stun:stun.l.google.com:19302' }
  ];
  peerConnection = new RTCPeerConnection({ iceServers });

  // 将本地媒体流添加到 PeerConnection
  localStream.getTracks().forEach(track => {
    peerConnection.addTrack(track, localStream);
  });
}

```

### 3.信令交换与连接建立

这个阶段是 WebRTC 最复杂的部分，涉及到 SDP Offer/Answer 和 ICE Candidates 的交换。

1. **创建 Offer**：A 端（发起方）**创建 offer**，这是一个包含本地会话信息的 SDP。
2. **发送 Offer**：A 端通过**信令服务器**将 Offer **发送给 B 端**。
3. **创建 Answer**：B 端接收 Offer，**然后创建 Answer**，这是一个包含 B 端会话信息的 SDP。
4. **发送 Answer**：B 端通过**信令服务器**将 Answer **发送回 A 端**。
5. **ICE Candidates 交换**：在整个过程中，两个浏览器会**不断生成 ICE Candidates（网络地址）**，并通过信令服务器交换给对方。这些信息用于**寻找最佳的连接路径**。



```

// A 端（发起方）
async function startCall() {
  createPeerConnection();

  // 1. 创建 Offer
  const offer = await peerConnection.createOffer();
  await peerConnection.setLocalDescription(offer);

  // 2. 将 Offer 发送给 B 端
  sendSignalingMessage({ type: 'offer', sdp: offer.sdp });
}

// B 端（接收方）
async function handleOffer(sdp) {
  createPeerConnection();

  // 3. 设置 Offer, 然后创建 Answer
  await peerConnection.setRemoteDescription({ type: 'offer', sdp });
  const answer = await peerConnection.createAnswer();
  await peerConnection.setLocalDescription(answer);

  // 4. 将 Answer 发送回 A 端
  sendSignalingMessage({ type: 'answer', sdp: answer.sdp });
}

// 接收 ICE Candidates
peerConnection.onicecandidate = (event) => {
  if (event.candidate) {
    // 交换 ICE Candidate
    sendSignalingMessage({
      type: 'ice-candidate',
      candidate: event.candidate
    });
  }
};

// 接收远程媒体流
peerConnection.ontrack = (event) => {
  const [remoteStream] = event.streams;
  document.getElementById('remoteVideo').srcObject = remoteStream;
};

```

作为前端开发者，使用 WebRTC 的基本步骤如下：

1. **搭建信令机制**：使用 WebSocket 实现信令服务器，这是 WebRTC 正常工作的前提。
2. **获取本地媒体流**：使用 `navigator.mediaDevices.getUserMedia` 获取摄像头和麦克风权限。
3. **创建 RTCPeerConnection**：这是核心对象，用于管理连接。

4. **SDP 交换**：通过信令服务器发送 Offer 和 Answer。
5. **ICE Candidate 交换**：通过信令服务器交换网络地址信息。
6. **监听媒体流**：使用 ontrack 事件来接收远程媒体流并渲染到页面上。

WebRTC 的复杂性主要在于其**状态机**和**信令**部分，你需要仔细处理各种事件和回调，但一旦掌握，它将为你开启一个全新的实时通信世界。