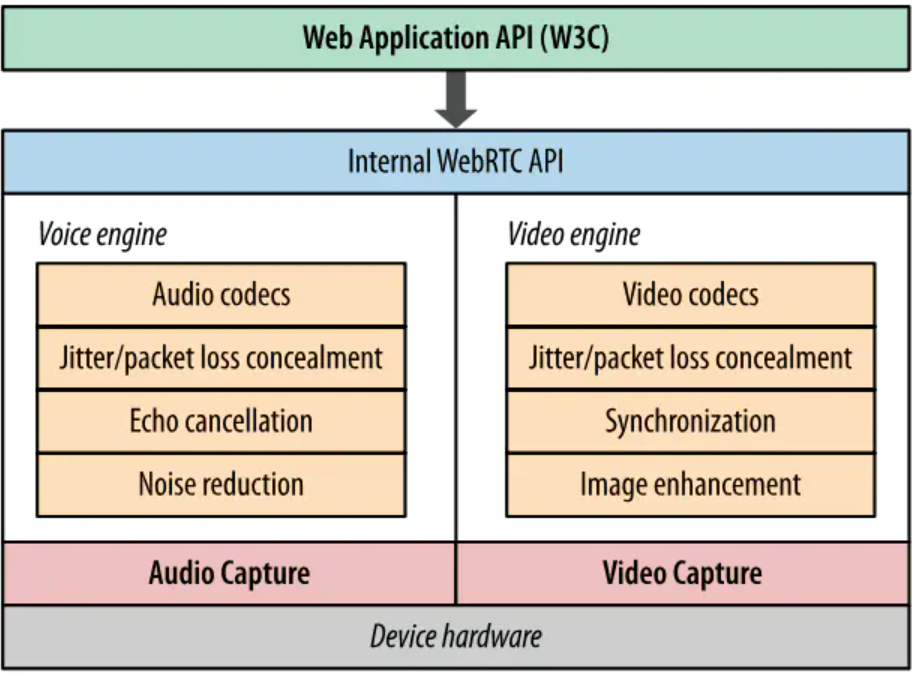
# 什么是WebRTC

简单来说，WebRTC 是一个可以在 Web 应用程序中实现音频，视频和数据的实时通信的开源项目。在实时通信中，音视频的采集和处理是一个很复杂的过程。比如音视频流的编解码、降噪和回声消除等，但是在 WebRTC 中，这一切都交由浏览器的底层封装来完成。我们可以直接拿到优化后的媒体流，然后将其输出到本地屏幕和扬声器，或者转发给其对等端。

WebRTC 的音视频处理引擎：



所以，我们可以在不需要任何第三方插件的情况下，实现一个浏览器到浏览器的点对点（P2P）连接，从而进行音视频实时通信。当然，WebRTC 提供了一些 API 供我们使用，在实时音视频通信的过程中，我们主要用到以下三个：

* getUserMedia：获取音频和视频流（MediaStream）
* RTCPeerConnection：点对点通信
* RTCDataChannel：数据通信

不过，虽然浏览器给我们解决了大部分音视频处理问题，但是从浏览器请求音频和视频时，我们还是需要特别注意流的大小和质量。因为即便硬件能够捕获高清质量流，CPU 和带宽也不一定可以跟上，这也是我们在建立多个对等连接时，不得不考虑的问题。

# 实现

接下来，我们通过分析上文提到的 API，来逐步弄懂 WebRTC 实时通信实现的流程。

## getUserMedia

### MediaStream

getUserMedia 这个 API 大家可能并不陌生，因为常见的 H5 录音等功能就需要用到它，主要就是用来获取设备的媒体流（即 MediaStream）。它可以接受一个约束对象 constraints 作为参数，用来指定需要获取到什么样的媒体流。

navigator.mediaDevices.getUserMedia({ audio: true, video: true })

// 参数表示需要同时获取到音频和视频

.then(stream => {

// 获取到优化后的媒体流

let video = document.querySelector('#rtc');

video.srcObject = stream;

})

.catch(err => {

// 捕获错误

});

我们简单看一下获取到的 MediaStream。



可以看到它有很多属性，我们只需要了解一下就好，更多信息可以查看 [MDN](https://developer.mozilla.org/zh-CN/docs/Web/API/MediaStream)。

\* id [String]: 对当前的 MS 进行唯一标识。所以每次刷新浏览器或是重新获取 MS，id 都会变动。

\* active [boolean]: 表示当前 MS 是否是活跃状态（就是是否可以播放）。

\* onactive: 当 active 为 true 时，触发该事件。

这里也可以通过 getAudioTracks()、getVideoTracks() 来查看获取到的流的某些信息，更多信息查看 [MDN](https://developer.mozilla.org/zh-CN/docs/Web/API/MediaStreamTrack)。



\* kind: 是当前获取的媒体流类型（Audio/Video）。

\* label: 是媒体设备，我这里用的是虚拟摄像头。

\* muted: 表示媒体轨道是否静音。

### 兼容性

继续来看 getUserMedia，navigator.mediaDevices.getUserMedia 是新版的 API，旧版的是 navigator.getUserMedia。为了避免兼容性问题，我们可以稍微处理一下（其实说到底，现在 WebRTC 的支持率还不算高，有需要的可以选择一些适配器，如 adapter.js）。

// 判断是否有 navigator.mediaDevices，没有赋成空对象

if (navigator.mediaDevices === undefined) {

navigator.mediaDevices = {};

}

// 继续判断是否有 navigator.mediaDevices.getUserMedia，没有就采用 navigator.getUserMedia

if (navigator.mediaDevices.getUserMedia) {

navigator.mediaDevices.getUserMedia = function(prams) {

let getUserMedia = navigator.webkitGetUserMedia || navigator.mozGetUserMedia;

// 兼容获取

if (!getUserMedia) {

return Promise.reject(new Error('getUserMedia is not implemented in this browser'));

}

return new Promise(function(resolve, reject) {

getUserMedia.call(navigator, prams, resolve, reject);

});

};

}

navigator.mediaDevices.getUserMedia(constraints)

.then(stream => {

let video = document.querySelector('#Rtc');

if ('srcObject' in video) { // 判断是否支持 srcObject 属性

video.srcObject = stream;

} else {

video.src = window.URL.createObjectURL(stream);

}

video.onloadedmetadata = function(e) {

video.play();

};

})

.catch((err) => { // 捕获错误

console.error(err.name + ': ' + err.message);

});

### Constraints

对于 constraints 约束对象，我们可以用来指定一些和媒体流有关的属性。比如指定是否获取某种流：

navigator.mediaDevices.getUserMedia({ audio: false, video: true });

// 只需要视频流，不要音频

指定视频流的宽高、帧率以及理想值：

// 获取指定宽高，这里需要注意：在改变视频流的宽高时，

// 如果宽高比和采集到的不一样，会直接截掉某部分

{ audio: false,

video: { width: 1280, height: 720 }

}

// 设定理想值、最大值、最小值

{

audio: true,

video: {

width: { min: 1024, ideal: 1280, max: 1920 },

height: { min: 776, ideal: 720, max: 1080 }

}

}

对于移动设备来说，还可以指定获取前摄像头，或者后置摄像头：

{ audio: true, video: { facingMode: "user" } } // 前置

{ audio: true, video: { facingMode: { exact: "environment" } } } // 后置

// 也可以指定设备 id，

// 通过 navigator.mediaDevices.enumerateDevices() 可以获取到支持的设备

{ video: { deviceId: myCameraDeviceId } }

还有一个比较有意思的就是设置视频源为屏幕，但是目前只有火狐支持了这个属性。

{ audio: true, video: {mediaSource: 'screen'} }

这里就不接着做搬运工了，更多精彩尽在 [MDN](https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia)，^\_^!

## RTCPeerConnection

RTCPeerConnection 接口代表一个由本地计算机到远端的 WebRTC 连接。该接口提供了创建，保持，监控，关闭连接的方法的实现。—— MDN

### 概述

RTCPeerConnection 作为创建点对点连接的 API，是我们实现音视频实时通信的关键。在点对点通信的过程中，需要交换一系列信息，通常这一过程叫做 — 信令（signaling）。在信令阶段需要完成的任务：

\* 为每个连接端创建一个 RTCPeerConnection，并添加本地媒体流。

\* 获取并交换本地和远程描述：SDP 格式的本地媒体元数据。

\* 获取并交换网络信息：潜在的连接端点称为 ICE 候选者

我们虽然把 WebRTC 称之为点对点的连接，但并不代表在实现过程中不需要服务器的参与。相反，在点对点的信道建立起来之前，二者之间是没有办法通信的。这也就意味着，在信令阶段，我们需要一个通信服务来帮助我们建立起这个连接。WebRTC 本身没有指定某一个信令服务，所以，我们可以但不限于使用 XMPP、XHR、Socket 等来做信令交换所需的服务。我在工作中采用的方案是基于 XMPP 协议的Strophe.js来做双向通信，但是在本例中则会使用Socket.io 以及 Koa 来做项目演示。

### NAT 穿越技术

我们先看连接任务的第一条：为每个连接端创建一个 RTCPeerConnection，并添加本地媒体流。事实上，如果是一般直播模式，则只需要播放端添加本地流进行输出，其他参与者只需要接受流进行观看即可。

因为各浏览器差异，RTCPeerConnection 一样需要加上前缀。

let PeerConnection = window.RTCPeerConnection ||

window.mozRTCPeerConnection ||

window.webkitRTCPeerConnection;

let peer = new PeerConnection(iceServers);

我们看见 RTCPeerConnection 也同样接收一个参数 — iceServers，先来看看它长什么样：

{

iceServers: [

{ url: "stun:stun.l.google.com:19302"}, // 谷歌的公共服务

{

url: "turn:\*\*\*",

username: \*\*\*, // 用户名

credential: \*\*\* // 密码

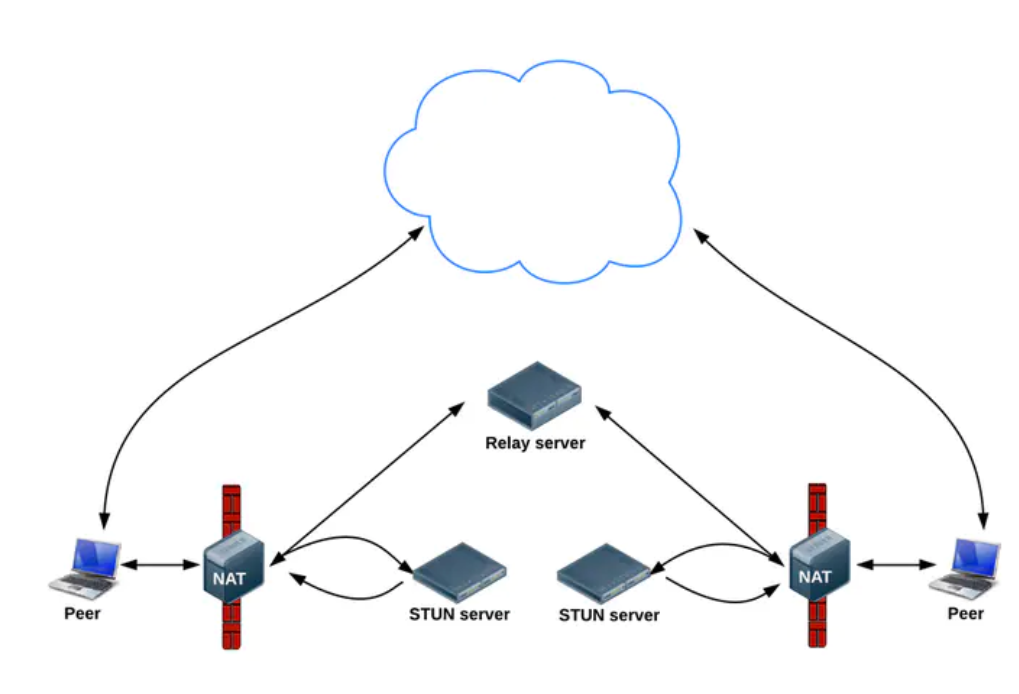
}

]

}

参数配置了两个 url，分别是 STUN 和 TURN，这便是 WebRTC 实现点对点通信的关键，也是一般 P2P 连接都需要解决的问题：NAT穿越。

NAT（Network Address Translation，网络地址转换）简单来说就是为了解决 IPV4 下的 IP 地址匮乏而出现的一种技术，也就是一个 公网 IP 地址一般都对应 n 个内网 IP。这样也就会导致不是同一局域网下的浏览器在尝试 WebRTC 连接时，无法直接拿到对方的公网 IP 也就不能进行通信，所以就需要用到 NAT 穿越（也叫打洞）。以下为 NAT 穿越基本流程：



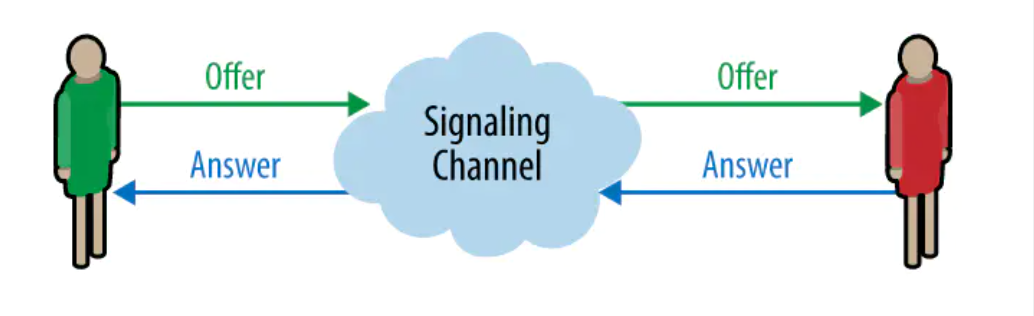
一般情况下会采用 ICE 协议框架进行 NAT 穿越，ICE 的全称为 Interactive Connectivity Establishment，即交互式连接建立。它使用 STUN 协议以及 TURN 协议来进行穿越。关于 NAT 穿越的更多信息可以参考 [ICE协议下NAT穿越的实现（STUN&TURN）](https://www.jianshu.com/p/84e8c78ca61d)、[P2P通信标准协议(三)之ICE](https://www.cnblogs.com/pannengzhi/p/5061674.html" \t "_blank)。

到这里，我们可以发现，WebRTC 的通信至少需要两种服务配合：

* 信令阶段需要双向通信服务辅助信息交换。
* STUN、TURN辅助实现 NAT 穿越。

### 建立点对点连接

WebRTC 的点对点连接到底是什么样的过程呢，我们通过结合图例来分析连接。



显而易见，在上述连接的过程中：

**呼叫端**（在这里都是指代浏览器）需要给 **接收端** 发送一条名为 offer 的信息。

**接收端** 在接收到请求后，则返回一条 answer 信息给 **呼叫端**。

这便是上述任务之一 ，SDP 格式的本地媒体元数据的交换。sdp 信息一般长这样：

v=0

o=- 1837933589686018726 2 IN IP4 127.0.0.1

s=-

t=0 0

a=group:BUNDLE audio video

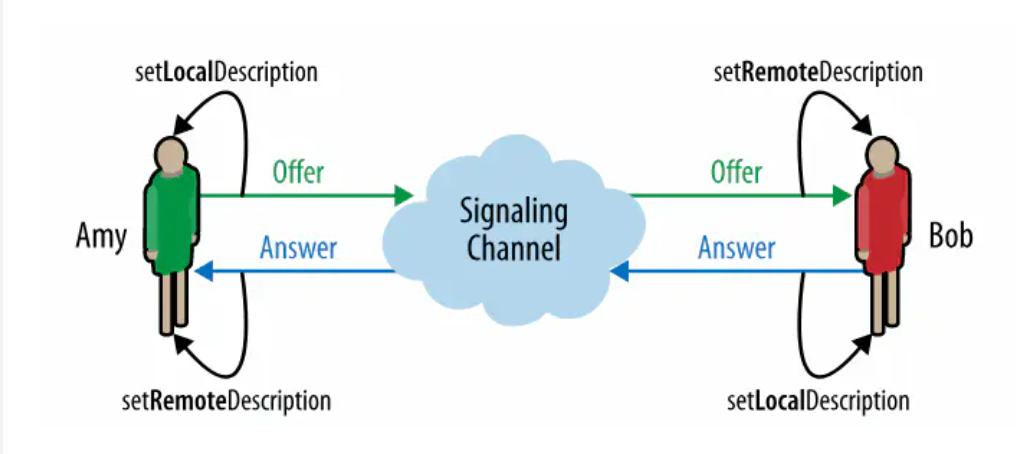
a=msid-semantic: WMS yvKeJMUSZzvJlAJHn4unfj6q9DMqmb6CrCOT

m=audio 9 UDP/TLS/RTP/SAVPF 111 103 104 9 0 8 106 105 13 110 112 113 126

...

...

但是任务不仅仅是交换，还需要分别保存自己和对方的信息，所以我们再加点料：

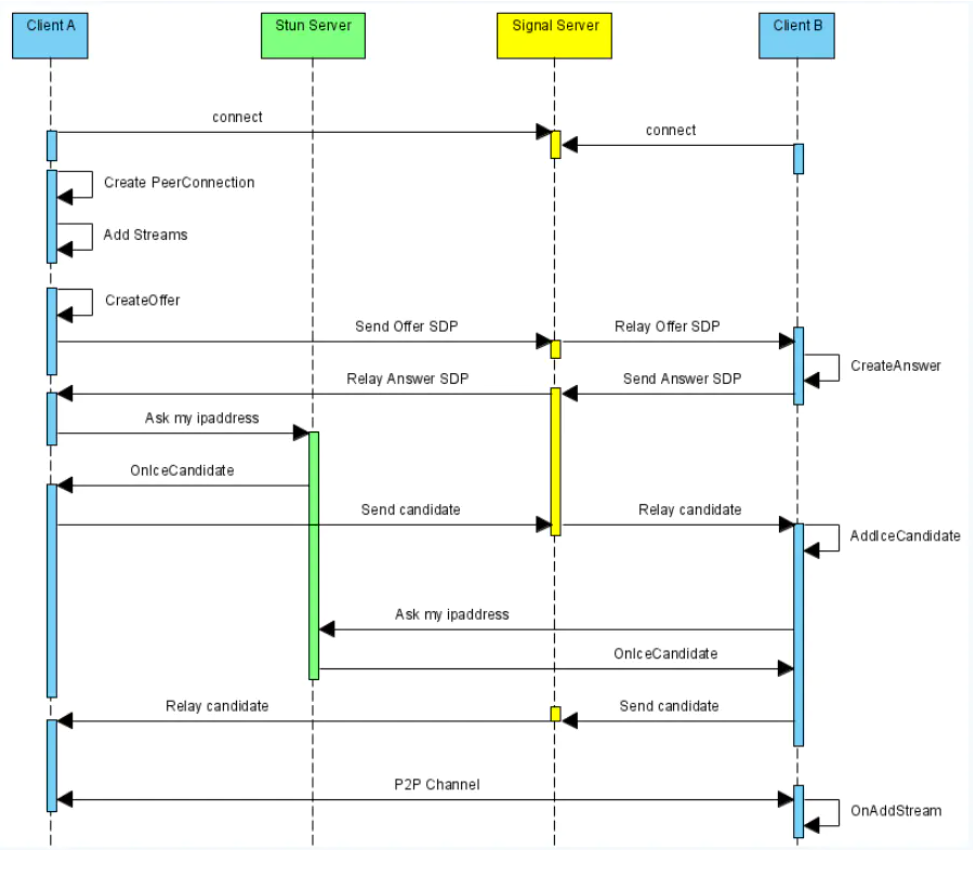


\* \*\*呼叫端\*\* 创建 offer 信息后，先调用 setLocalDescription 存储本地 offer 描述，再将其发送给 \*\*接收端\*\*。

\* \*\*接收端\*\* 收到 offer 后，先调用 setRemoteDescription 存储远端 offer 描述；然后又创建 answer 信息，同样需要调用 setLocalDescription 存储本地 answer 描述，再返回给 \*\*呼叫端\*\*

\* \*\*呼叫端\*\* 拿到 answer 后，再次调用 setRemoteDescription 设置远端 answer 描述。

到这里点对点连接还缺一步，也就是网络信息 ICE 候选交换。不过这一步和 offer、answer 信息的交换并没有先后顺序，流程也是一样的。即：在**呼叫端**和**接收端**的 ICE 候选信息准备完成后，进行交换，并互相保存对方的信息，这样就完成了一次连接。



这张图是我认为比较完善的了，详细的描述了整个连接的过程。正好我们再来小结一下：

\* 基础设施：必要的信令服务和 NAT 穿越服务

\* clientA 和 clientB 分别创建 RTCPeerConnection 并为输出端添加本地媒体流。如果是视频通话类型，则意味着，两端都需要添加媒体流进行输出。

\* 本地 ICE 候选信息采集完成后，通过信令服务进行交换。

\* 呼叫端（好比 A 给 B 打视频电话，A 为呼叫端）发起 offer 信息，接收端接收并返回一个 answer 信息，呼叫端保存，完成连接。

## 本地 1 v 1 对等连接

### 创建媒体流

页面布局很简单，就是两个 video 标签，分别代表 A 和 B。所以我们直接看代码，虽然源码是用 Vue 构建的，但是并没有用到特别的 API，整体上和 es6 的 class 语法相差不大，而且都有详细的注释，所以建议没有 Vue 基础的同学可以直接当成 es6 来阅读。示例 **源码库** [webrtc-stream](https://github.com/wuyawei/webrtc-stream)

async createMedia() {

// 保存本地流到全局

this.localstream = await navigator.mediaDevices.getUserMedia({ audio: true, video: true })

let video = document.querySelector('#rtcA');

video.srcObject = this.localstream;

this.initPeer(); // 获取到媒体流后，调用函数初始化 RTCPeerConnection

}

### 初始化RTCPeerConnection

initPeer() {

...

this.peerA.addStream(this.localstream); // 添加本地流

this.peerA.onicecandidate = (event) => {

// 监听 A 的ICE候选信息 如果收集到，就添加给 B 连接状态

if (event.candidate) {

this.peerB.addIceCandidate(event.candidate);

}

};

...

// 监听是否有媒体流接入，如果有就赋值给 rtcB 的 src

this.peerB.onaddstream = (event) => {

let video = document.querySelector('#rtcB');

video.srcObject = event.stream;

};

this.peerB.onicecandidate = (event) => { 连接状态

// 监听 B 的ICE候选信息 如果收集到，就添加给 A

if (event.candidate) {

this.peerA.addIceCandidate(event.candidate);

}

};

}

这部分主要就是分别创建 peer 实例，并互相交换 ICE 信息。不过有一个属性需要在这里提一下，就是 iceConnectionState。

peer.oniceconnectionstatechange = (evt) => {

console.log('ICE connection state change: ' + evt.target.iceConnectionState);

};

我们可以通过 oniceconnectionstatechange 方法来监测 ICE 连接的状态，它一共有七种状态：

new ICE代理正在收集候选人或等待提供远程候选人。

checking ICE代理已经在至少一个组件上接收了远程候选者，并且正在检查候选但尚未找到连接。除了检查，它可能还在收集。

connected ICE代理已找到所有组件的可用连接，但仍在检查其他候选对以查看是否存在更好的连接。它可能还在收集。

completed ICE代理已完成收集和检查，并找到所有组件的连接。

failed ICE代理已完成检查所有候选对，但未能找到至少一个组件的连接。可能已找到某些组件的连接。

disconnected ICE 连接断开

closed ICE代理已关闭，不再响应STUN请求。

我们需要注意的是 completed 和 disconnected，一个是完成连接时触发，一个在断开连接时触发。

### 创建连接

async call() {

if (!this.peerA || !this.peerB) { // 判断是否有对应实例，没有就重新创建

this.initPeer();

}

try {

let offer = await this.peerA.createOffer(this.offerOption); // 创建 offer

await this.onCreateOffer(offer);

} catch (e) {

console.log('createOffer: ', e);

}

}

这里需要判断是否有对应实例，是为了挂断之后又重新呼叫做的处理。

async onCreateOffer(desc) {

try {

await this.peerB.setLocalDescription(desc); // 呼叫端设置本地 offer 描述

} catch (e) {

console.log('Offer-setLocalDescription: ', e);

}

try {

await this.peerA.setRemoteDescription(desc); // 接收端设置远程 offer 描述

} catch (e) {

console.log('Offer-setRemoteDescription: ', e);

}

try {

let answer = await this.peerA.createAnswer(); // 接收端创建 answer

await this.onCreateAnswer(answer);

} catch (e) {

console.log('createAnswer: ', e);

}

},

async onCreateAnswer(desc) {

try {

await this.peerA.setLocalDescription(desc); // 接收端设置本地 answer 描述

} catch (e) {

console.log('answer-setLocalDescription: ', e);

}

try {

await this.peerB.setRemoteDescription(desc); // 呼叫端端设置远程 answer 描述

} catch (e) {

console.log('answer-setRemoteDescription: ', e);

}

}

这基本就是之前重复过好几次的流程用代码写出来而已，看到这里，思路应该比较清晰了。不过有一点需要说明一下，就是现在这种情况，A 作为呼叫端，B 一样是可以拿到 A 的媒体流的。因为连接一旦建立了，就是双向的，只不过 B 初始化 peer 的时候没有添加本地流，所以 A 不会有 B 的媒体流。

## 网络 1 v 1 对等连接

想必基本流程大家都已经熟悉了，通过图解、实例来来回回讲了好几遍。所以趁热打铁，我们这次把服务加上，做一个真正的点对点连接。在看下面的文章之前，我希望你有一点点 Koa 和 Scoket.io 的基础，了解一些基本 API 即可。不熟悉的同学也不要紧，现在看也来得及，[Koa](https://koa.bootcss.com/" \t "_blank)、[Socke.io](https://www.w3cschool.cn/socket/socket-ulbj2eii.html" \t "_blank)，或者可以参考我之前的文章 [Vchat - 一个社交聊天系统（vue + node + mongodb）](https://juejin.im/post/6844903734145712135)。

### 需求

连接过程涉及到多个环节，这里就不一一截图了，可以直接去演示地址查看。简单分析一下我们要做的事情： \* 加入房间后，获取到房间的所有在线成员。 \* 选择任一成员进行通话，也就是呼叫动作。这时候就有一些细节问题要处理：不能呼叫自己、同一时刻只允许呼叫一个人且需要判断对方是否是通话中、呼叫后回复需要有相应判断（同意、拒绝以及通话中） \* 拒绝或通话中，都没有后续动作，可以换个人再呼叫。同意之后，就要开始建立点对点连接。

### 加入房间

简单看一下加入房间的流程：

// 前端

join() {

if (!this.account) return;

this.isJoin = true; // 输入框弹层逻辑

window.sessionStorage.account = this.account; // 刷新判断是否登录过

socket.emit('join', {roomid: this.roomid, account: this.account}); // 发送加入房间请求

}

// 后端

const sockS = {}; // 不同客户端对应的 sock 实例

const users = {}; // 成员列表

sock.on('join', data=>{

sock.join(data.roomid, () => {

if (!users[data.roomid]) {

users[data.roomid] = [];

}

let obj = {

account: data.account,

id: sock.id

};

let arr = users[data.roomid].filter(v => v.account === data.account);

if (!arr.length) {

users[data.roomid].push(obj);

}

sockS[data.account] = sock; // 保存不同客户端对应的 sock 实例

// 将房间内成员列表发给房间内所有人

app.\_io.in(data.roomid).emit('joined', users[data.roomid], data.account, sock.id);

});

});

后端成员列表的处理，是因为做了多房间的逻辑，按每个房间的成员表返回的。你们如果做的时候没有多房间，则不需要这么考虑。sockS 的处理，是为了发送私聊消息。

### 呼叫

前面已经说了呼叫的注意事项，所以这里就一起来讲。需要注意的就是消息中需要带有自己和对方的 account，因为这是判断成员 sock 的标识，也就是之前存储在 socks 中的用来发私聊消息的。然后是前面说的三种状态，在这里用 type 值 1, 2, 3 来区分，然后给出不同的回复

// 前端

apply(account) { // 发送请求

// account 对方account self 是自己的account

this.loading = true;

this.loadingText = '呼叫中'; // 呼叫中 loading

socket.emit('apply', {account: account, self: this.account});

},

reply(account, type) { // 处理回复

socket.emit('reply', {account: account, self: this.account, type: type});

}

// 收到请求

socket.on('apply', data => {

if (this.isCall) { // 判断是否在通话中

this.reply(data.self, '3');

return;

}

this.$confirm(data.self + ' 向你请求视频通话, 是否同意?', '提示', {

confirmButtonText: '同意',

cancelButtonText: '拒绝',

type: 'warning'

}).then(async () => {

this.isCall = data.self;

this.reply(data.self, '1');

}).catch(() => {

this.reply(data.self, '2');

});

});

// 后端

sock.on('apply', data=>{ // 转发申请

sockS[data.account].emit('apply', data);

});

后端比较简单，仅仅是转发一下请求，给对应的客户端。其实我们这个例子的后端，基本都是这个操作，所以后面的后端代码就不贴了，可以去源码直接看。

### 回复

回复和和呼叫是一样的逻辑，分别处理不同的回复就好了。

// 前端

socket.on('reply', async data =>{ // 收到回复

this.loading = false;

switch (data.type) {

case '1': // 同意

this.isCall = data.self; // 存储通话对象

break;

case '2': //拒绝

this.$message({

message: '对方拒绝了你的请求！',

type: 'warning'

});

break;

case '3': // 正在通话中

this.$message({

message: '对方正在通话中！',

type: 'warning'

});

break;

}

});

### 创建连接

呼叫和回复的逻辑基本清楚了，那我们继续思考，应该在什么时机创建 P2P 连接呢？我们之前说的，拒绝和通话中都不需要处理，只有同意需要，那就应该在同意请求的位置开始创建。需要注意的是，同意请求有两个地方：一个是你点了同意，另一个是对方知道你点了同意之后。

本例采取的是呼叫方发送 Offer，这个地方一定得注意，只要有一方创建 Offer 就可以了，因为一旦连接就是双向的。

socket.on('apply', data => { // 你点同意的地方

...

this.$confirm(data.self + ' 向你请求视频通话, 是否同意?', '提示', {

confirmButtonText: '同意',

cancelButtonText: '拒绝',

type: 'warning'

}).then(async () => {

await this.createP2P(data); // 同意之后创建自己的 peer 等待对方的 offer

... // 这里不发 offer

})

...

});

socket.on('reply', async data =>{ // 对方知道你点了同意的地方

switch (data.type) {

case '1': // 只有这里发 offer

await this.createP2P(data); // 对方同意之后创建自己的 peer

this.createOffer(data); // 并给对方发送 offer

break;

...

}

});

和微信等视频通话一样，双方都需要进行媒体流输出，因为你们都要看见对方。所以这里和之前本地对等连接的区别就是都需要给自己的 RTCPeerConnection 实例添加媒体流，然后连接后各自都能拿到对方的视频流。在 初始化 RTCPeerConnection 时，记得加上 onicecandidate 函数，用以给对方发送 ICE 候选。

async createP2P(data) {

this.loading = true; // loading动画

this.loadingText = '正在建立通话连接';

await this.createMedia(data);

},

async createMedia(data) {

... // 获取并将本地流赋值给 video 同之前

this.initPeer(data); // 获取到媒体流后，调用函数初始化 RTCPeerConnection

},

initPeer(data) {

// 创建输出端 PeerConnection

...

this.peer.addStream(this.localstream); // 都需要添加本地流

this.peer.onicecandidate = (event) => {

// 监听ICE候选信息 如果收集到，就发送给对方

if (event.candidate) { // 发送 ICE 候选

socket.emit('1v1ICE',

{account: data.self, self: this.account, sdp: event.candidate});

}

};

this.peer.onaddstream = (event) => {

// 监听是否有媒体流接入，如果有就赋值给 rtcB 的 src，改变相应loading状态，赋值省略

this.isToPeer = true;

this.loading = false;

...

};

}

createOffer 等信息交换和之前一样，只是需要通过 Socket 转发给对应的客户端。然后各自接收到消息后分别采取对应的措施。

socket.on('1v1answer', (data) =>{ // 接收到 answer

this.onAnswer(data);

});

socket.on('1v1ICE', (data) =>{ // 接收到 ICE

this.onIce(data);

});

socket.on('1v1offer', (data) =>{ // 接收到 offer

this.onOffer(data);

});

// 这里只贴一个 createOffer 的代码，因为和之前的思路都一样，只是写法有些区别

// 建议大家都自己敲一遍，有问题可以交流，也可以去源码查看。

async createOffer(data) { // 创建并发送 offer

try {

// 创建offer

let offer = await this.peer.createOffer(this.offerOption);

// 呼叫端设置本地 offer 描述

await this.peer.setLocalDescription(offer);

// 给对方发送 offer

socket.emit('1v1offer', {account: data.self, self: this.account, sdp: offer});

} catch (e) {

console.log('createOffer: ', e);

}

}

### 挂断

挂断的思路依然是将各自的 peer 关闭，但是这里挂断方还需要借助 Socket 告诉对方，你已经挂电话了，不然对方还在痴痴地等。

hangup() { // 挂断通话 并做相应处理 对方收到消息后一样需要关闭连接

socket.emit('1v1hangup', {account: this.isCall, self: this.account});

this.peer.close();

this.peer = null;

this.isToPeer = false;

this.isCall = false;

}

# WebRTC + Canvas 实现一个双人协作的共享画板

## RTCDataChannel

### 介绍

简单来说，RTCDataChannel 就是在点对点连接中建立一个双向的数据通道，从而获得文本、文件等数据的点对点传输能力。它依赖于流控制传输协议（SCTP），SCTP 是一种传输协议，类似于 TCP 和 UDP，可以直接在 IP 协议之上运行。但是，在 WebRTC 的情况下，SCTP 通过安全的 DTLS 隧道进行隧道传输，该隧道本身在 UDP 之上运行。

另外总的来说 RTCDataChannel 和 WebSocket 很像，只不过 WebSocket 不是 P2P 连接，需要服务器做中转。

### 使用

// 创建

let Channel = RTCPeerConnection.createDataChannel('messagechannel', options);

// messagechannel 可以看成是给 DataChannel 取的别名，限制是不得超过65,535 字节。

// options 可以设置一些属性，一般默认就好。

// 接收

RTCPeerConnection.ondatachannel = function(event) {

let channel = event.channel;

}

RTCDataChannel 只需要在一端使用 createDataChannel 来创建实例，在接收端只需要给 RTCPeerConnection 加上 ondatachannel 监听即可。但是有一点需要注意的是，一定要是 **呼叫端** 也就是创建 createOffer 的那端来 createDataChannel 创建通道。

RTCDataChannel 的一些属性，更多可以查看 [MDN](https://developer.mozilla.org/zh-CN/docs/Web/API/RTCPeerConnection/createDataChannel#RTCDataChannelInit_dictionary)

* label：创建时提到的别名。
* ordered：指发送的消息是否需要按照它们的发送顺序到达目的地（true），或者允许它们无序到达（false）。默认值：true。
* binaryType：是一个 DOMString 类型，表示发送的二进制数据的类型。值为 blob 或 arraybuffer，默认值为 "blob"。
* readyState：表示数据连接的状态：
  + connecting 等待连接，也是创建初始状态。
  + open 连接成功并且运行。
  + closing 连接关闭中，不会接受新的发送任务，但是缓冲队列中的消息还是会被继续发送或者接收。也就是没发送完的会继续发送。
  + closed 连接完全被关闭。

一个简单的收发消息的功能，我们已经知道了在 **呼叫端** 和 **接收端** 分别拿到 RTCDataChannel 实例，但是还不知道怎么接收和发送消息，现在就来看一下。

// this.peerB 呼叫端 RTCPeerConnection 实例

this.channelB = this.peerB.createDataChannel('messagechannel'); // 创建 Channel

this.channelB.onopen = (event) => { // 监听连接成功

console.log('channelB onopen', event);

this.messageOpen = true; // 连接成功后显示消息框

};

this.channelB.onclose = function(event) { // 监听连接关闭

console.log('channelB onclose', event);

};

// 发送消息

send() {

this.channelB.send(this.sendText);

this.sendText = '';

}

// this.peerA 接收端 RTCPeerConnection 实例

this.peerA.ondatachannel = (event) => {

this.channelA = event.channel; // 获取接收端 channel 实例

this.channelA.onopen = (e) => { // 监听连接成功

console.log('channelA onopen', e);

};

this.channelA.onclose = (e) => { // 监听连接关闭

console.log('channelA onclose', e);

};

this.channelA.onmessage = (e) => { // 监听消息接收

this.receiveText = e.data; // 接收框显示消息

console.log('channelA onmessage', e.data);

};

};

建立对等连接的过程这里就省略了，通过这两段代码就可以实现简单的文本传输了。

## 白板演示

### 需求



this.localstream = this.$refs['canvas'].captureStream();

一句话就可以把 Canvas 变成媒体流了，所以演示画面仍然是 video 标签在播放媒体流，只是这次不是从摄像头获取的流，而是 Canvas 转换的。

### 封装 Canvas 类

现在点对点连接我们有了，白板流我们也有了，好像就缺一个能画画的 Canvas 了。说时迟那时快，看，Canvas 来了。[源码地址](https://github.com/wuyawei/webrtc-stream/blob/master/webrtc-main/src/utils/palette.js)

* 功能点

从图上我们可以看见这个画板类需要哪些功能：绘制圆形、绘制线条、绘制矩形、绘制多边形、橡皮擦、撤回、前进、清屏、线宽、颜色，这些是功能可选项。

再往细分析：

1. 绘制各种形状，肯定要用到鼠标事件，来记录鼠标移动的位置从而进行绘图；
2. 绘制多边形，需要用户选择到底是几边形，最少当然是 3 边，也就是三角形；
3. 线宽和颜色也是用户可以改变的东西，所以我们需要提供一个接口，用来修改这些属性；
4. 撤回和前进，意味着我们需要保存每次绘制的图像，保存时机在鼠标抬起的时候；而且撤回和前进不是无限制的，有边界点；
5. 试想一下：当你绘制了 5 步，现在撤回到了第 3 步，想在第 3 步的基础上再次进行绘制，这时候是不是应该把第 4 步和第 5 步清除掉？如果不清除，新绘制的算第几步？

综上，我们可以先列出大体的框架。

// Palette.js

class Palette {

constructor() {

}

gatherImage() { // 采集图像

}

reSetImage() { // 重置为上一帧

}

onmousedown(e) { // 鼠标按下

}

onmousemove(e) { // 鼠标移动

}

onmouseup() { // 鼠标抬起

}

line() { // 绘制线性

}

rect() { // 绘制矩形

}

polygon() { // 绘制多边形

}

arc() { // 绘制圆形

}

eraser() { // 橡皮擦

}

cancel() { // 撤回

}

go () { // 前进

}

clear() { // 清屏

}

changeWay() { // 改变绘制条件

}

destroy() { // 销毁

}

}

* 绘制线条

任何绘制，都需要经过鼠标按下，鼠标移动，鼠标抬起这几步；

onmousedown(e) { // 鼠标按下

this.isClickCanvas = true; // 鼠标按下标识

this.x = e.offsetX; // 获取鼠标按下的坐标

this.y = e.offsetY;

this.last = [this.x, this.y]; // 保存每次的坐标

this.canvas.addEventListener('mousemove', this.bindMousemove); // 监听 鼠标移动事件

}

onmousemove(e) { // 鼠标移动

this.isMoveCanvas = true; // 鼠标移动标识

let endx = e.offsetX;

let endy = e.offsetY;

let width = endx - this.x;

let height = endy - this.y;

let now = [endx, endy]; // 当前移动到的坐标

switch (this.drawType) {

case 'line' :

this.line(this.last, now, this.lineWidth, this.drawColor); // 绘制线条的方法

break;

}

}

onmouseup() { // 鼠标抬起

if (this.isClickCanvas) {

this.isClickCanvas = false;

this.canvas.removeEventListener('mousemove', this.bindMousemove); // 移除鼠标移动事件

if (this.isMoveCanvas) { // 鼠标没有移动不保存

this.isMoveCanvas = false;

this.gatherImage(); // 保存每次的图像

}

}

}

代码中鼠标移动事件用的是 this.bindMousemove，这是因为我们需要绑定 this，但是 bind 后每次返回的并不是同一个函数，而移除事件和绑定的不是同一个的话，无法移除。所以需要用变量保存一下 bind 后的函数。

this.bindMousemove = this.onmousemove.bind(this); // 解决 eventlistener 不能用 bind

this.bindMousedown = this.onmousedown.bind(this);

this.bindMouseup = this.onmouseup.bind(this);

在 this.line 方法中，我们将所有的参数采用函数参数的形式传入，是为了共享画板时需要同步绘制对方绘图的每一步。在绘制线条的时候，采取将每次移动的坐标点连接成线的方式，这样画出来比较连续。如果直接绘制点，速度过快会出现较大的断层。

line(last, now, lineWidth, drawColor) { // 绘制线性

this.paint.beginPath();

this.paint.lineCap = "round"; // 设定线条与线条间接合处的样式

this.paint.lineJoin = "round";

this.paint.lineWidth = lineWidth;

this.paint.strokeStyle = drawColor;

this.paint.moveTo(last[0], last[1]);

this.paint.lineTo(now[0], now[1]);

this.paint.closePath();

this.paint.stroke(); // 进行绘制

this.last = now; // 更新上次的坐标

}

* 撤回、前进

在鼠标抬起的时候，用到了一个 gatherImage 方法，用来采集图像，这也是撤回和前进的关键。

gatherImage() { // 采集图像

this.imgData = this.imgData.slice(0, this.index + 1);

// 每次鼠标抬起时，将储存的imgdata截取至index处

let imgData = this.paint.getImageData(0, 0, this.width, this.height);

this.imgData.push(imgData);

this.index = this.imgData.length - 1; // 储存完后将 index 重置为 imgData 最后一位

}

回想一下之前提到的一个问题，在撤退到某一步且从这一步开始作画的话，我们需要把这一步后续的图像都删除，以免造成混乱。所以我们用一个全局的 index 作为当前绘制的是第几帧图像的标识，在每次保存的图像的时候，都截取一次图像缓存数组 imgData，用以跟 index 保持一致，储存完后将 index 重置到最后一位。

cancel() { // 撤回

if (--this.index <0) { // 最多重置到 0 位

this.index = 0;

return;

}

this.paint.putImageData(this.imgData[this.index], 0, 0); // 绘制

}

go () { // 前进

if (++this.index > this.imgData.length -1) { // 最多前进到 length -1

this.index = this.imgData.length -1;

return;

}

this.paint.putImageData(this.imgData[this.index], 0, 0);

}

* 橡皮擦

橡皮擦我们用到了 Canvas 的一个属性，clip 裁切。简单来说，就是将图像绘制一个裁剪区域，后续的操作便都只会作用域该区域。所以当我们把裁剪区域设置成一个小圆点的时候，后面就算清除整个画板，实际也只清除了这个圆点的范围。清除完以后，再将其还原。

eraser(endx, endy, width, height, lineWidth) { // 橡皮擦

this.paint.save(); // 缓存裁切前的

this.paint.beginPath();

this.paint.arc(endx, endy, lineWidth / 2, 0, 2 \* Math.PI);

this.paint.closePath();

this.paint.clip(); // 裁切

this.paint.clearRect(0, 0, width, height);

this.paint.fillStyle = '#fff';

this.paint.fillRect(0, 0, width, height);

this.paint.restore(); // 还原

}

* 矩形

在绘制矩形等这种形状是，因为其并不是一个连续的动作，所以应该以鼠标最后的位置为坐标进行绘制。那么这个时候应该不断清除画板并重置为上一帧的图像（这里的上一帧是指，鼠标按下前的，因为鼠标抬起才会保存一帧图像，显然，移动的时候没有保存）。

看一下不做重置的现象，应该更容易理解。下面，就是见证奇迹的时刻：



rect(x, y, width, height, lineWidth, drawColor) { // 绘制矩形

this.reSetImage();

this.paint.lineWidth = lineWidth;

this.paint.strokeStyle = drawColor;

this.paint.strokeRect(x, y, width, height);

}

reSetImage() { // 重置为上一帧

this.paint.clearRect(0, 0, this.width, this.height);

if(this.imgData.length >= 1){

this.paint.putImageData(this.imgData[this.index], 0, 0);

}

}

### 建立连接

这下准备工作都做好了，对等连接该上了。我们不需要获取媒体流，而是用 Canvas 流代替。

async createMedia() {

// 保存canvas流到全局

this.localstream = this.$refs['canvas'].captureStream();

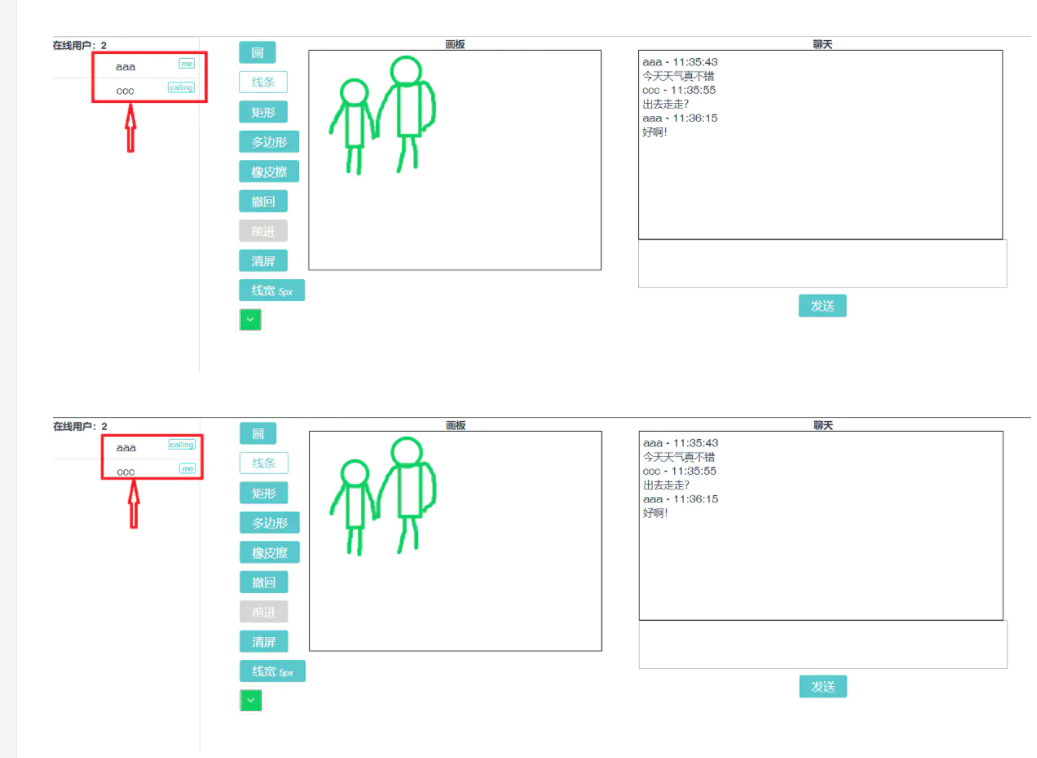
this.initPeer(); // 获取到媒体流后，调用函数初始化 RTCPeerConnection

}

## 共享画板

### 需求

做了这么多铺垫，一切都是为了今天的终极目标，完成一个多人协作的共享画板。实际上，在共享画板中要用到的知识点，我们都已经讲完了。我们基于上期的 1 v 1 网络连接做一些改造，先重温一下前言中的那张图。



仔细看一下我圈住的地方，从登录人可以看出，这是我在两个浏览器打开的页面截图。当然你们也可以直接去线上地址实际操作一下。两个页面，两个画板，两个人都可以操作，各自的操作也会分别同步到对方的画板上。右边是一个简单的聊天室，所有的数据同步以及聊天消息都是基于今天讲的 RTCDataChannel 来做的。

### 建立连接

这次不需要视频流，也不需要 Canvas 流，所以我们在点对点连接时直接建立数据通道。

createDataChannel() { // 创建 DataChannel

try{

this.channel = this.peer.createDataChannel('messagechannel');

this.handleChannel(this.channel);

} catch (e) {

console.log('createDataChannel:', e);

}

},

onDataChannel() { // 接收 DataChannel

this.peer.ondatachannel = (event) => {

// console.log('ondatachannel', event);

this.channel = event.channel;

this.handleChannel(this.channel);

};

},

handleChannel(channel) { // 处理 channel

channel.binaryType = 'arraybuffer';

channel.onopen = (event) => { // 连接成功

console.log('channel onopen', event);

this.isToPeer = true; // 连接成功

this.loading = false; // 解除 loading

this.initPalette();

};

channel.onclose = function(event) { // 连接关闭

console.log('channel onclose', event)

};

channel.onmessage = (e) => { // 收到消息

this.messageList.push(JSON.parse(e.data));

// console.log('channel onmessage', e.data);

};

}

分别在 **呼叫端** 和 **接收端** 创建 channel。

// 呼叫端

socket.on('reply', async data =>{ // 收到回复

this.loading = false;

switch (data.type) {

case '1': // 同意

this.isCall = data.self;

// 对方同意之后创建自己的 peer

await this.createP2P(data);

// 建立DataChannel

await this.createDataChannel();

// 并给对方发送 offer

this.createOffer(data);

break;

···

}

});

// 接收端

socket.on('apply', data => { // 收到请求

···

this.$confirm(data.self + ' 向你请求视频通话, 是否同意?', '提示', {

confirmButtonText: '同意',

cancelButtonText: '拒绝',

type: 'warning'

}).then(async () => {

await this.createP2P(data); // 同意之后创建自己的 peer 等待对方的 offer

await this.onDataChannel(); // 接收 DataChannel

···

}).catch(() => {

···

});

});

### 聊天

连接成功后，就可以进行简单的聊天了，和之前讲 API 时的栗子基本一样。本次只实现了简单的文本聊天，DataChannel 还支持文件传输

send(arr) { // 发送消息

if (arr[0] === 'text') {

let params = {account: this.account, time: this.formatTime(new Date()), mes: this.sendText, type: 'text'};

this.channel.send(JSON.stringify(params));

this.messageList.push(params);

this.sendText = '';

} else { // 处理数据同步

this.channel.send(JSON.stringify(arr));

}

}

### 画板同步

一直说需要将各自的画板操作同步给对方，那到底什么时机来触发同步操作呢？又需要同步哪些数据呢？在之前封装画板类的时候我们提到过，所有绘图需要的数据都通过参数形式传递。

this.line(this.last, now, this.lineWidth, this.drawColor);

所以很容易想到，我们只需要在每次自己绘图也就是鼠标移动时，将绘图所需的数据、操作的类型（也许是撤回、前进等操作）都发送给对方就可以了。在这里我们利用一个回调函数去通知页面什么时候开始给对方发送数据。

// 有省略

constructor(canvas, {moveCallback}) {

···

this.moveCallback = moveCallback || function () {}; // 鼠标移动的回调

}

onmousemove(e) { // 鼠标移动

this.isMoveCanvas = true;

let endx = e.offsetX;

let endy = e.offsetY;

let width = endx - this.x;

let height = endy - this.y;

let now = [endx, endy]; // 当前移动到的位置

switch (this.drawType) {

case 'line' : {

let params = [this.last, now, this.lineWidth, this.drawColor];

this.moveCallback('line', ...params);

this.line(...params);

}

break;

case 'rect' : {

let params = [this.x, this.y, width, height, this.lineWidth, this.drawColor];

this.moveCallback('rect', ...params);

this.rect(...params);

}

break;

case 'polygon' : {

let params = [this.x, this.y, this.sides, width, height, this.lineWidth, this.drawColor];

this.moveCallback('polygon', ...params);

this.polygon(...params);

}

break;

case 'arc' : {

let params = [this.x, this.y, width, height, this.lineWidth, this.drawColor];

this.moveCallback('arc', ...params);

this.arc(...params);

}

break;

case 'eraser' : {

let params = [endx, endy, this.width, this.height, this.lineWidth];

this.moveCallback('eraser', ...params);

this.eraser(...params);

}

break;

}

}

看起来挺丑，但是这么写是有原因的。首先 moveCallback 不能放在相应操作函数的下面，因为都是同步操作，有些值在绘图完成后会发生改变，比如 last 和 now ，绘图完成后，二者相等。

其次，不能将 moveCallback 写在相应操作函数内部，否则会无限循环。你想，你画了一条线，Callback 通知对方也画一条，对方也要调用 line 方法绘制相同的线。结果倒好，Callback 在 line 方法内部，它立马又得反过来告诉你，这样你来我往，一回生二回熟，来而不往非礼也，额，不好意思，说快了。反正会造成一些麻烦。

页面收到 Callback 通知以后，直接调用 send 方法，将数据传递给对方。moveCallback(...arr) { // 同步到对方

this.send(arr);

},

send(arr) { // 发送消息

if (arr[0] === 'text') {

···

} else { // 处理数据同步

this.channel.send(JSON.stringify(arr));

}

}

接收到数据后，调用封装类相应方法进行绘制。

handleChannel(channel) { // 处理 channel

···

channel.onmessage = (e) => { // 收到消息 普通消息类型是 对象

if (Array.isArray(JSON.parse(e.data))) { // 如果收到的是数组，进行结构

let [type, ...arr] = JSON.parse(e.data);

this.palette[type](...arr); // 调用相应方法

} else {

this.messageList.push(JSON.parse(e.data)); // 接收普通消息

}

// console.log('channel onmessage', e.data);

};

}

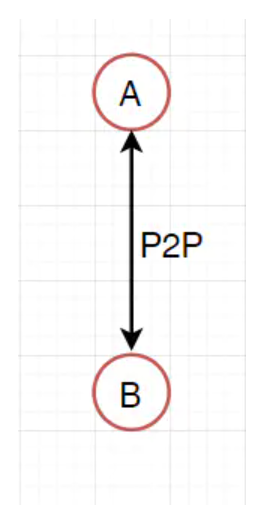
# 多人视频聊天

## 三种模式

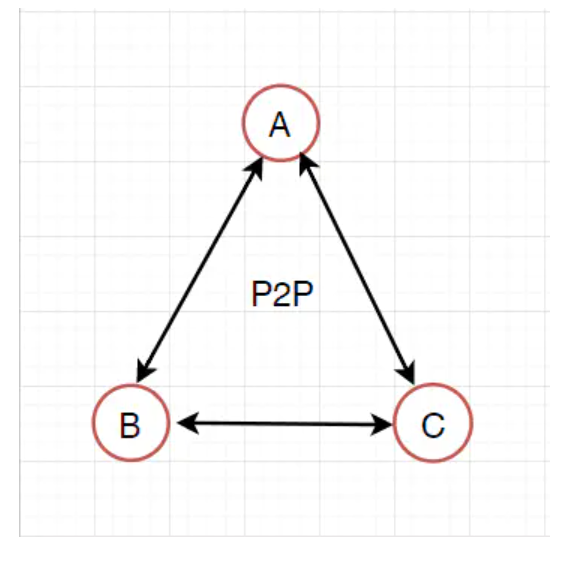
简单介绍一下基于 WebRTC 的多人通信的几种架构模式。

* Mesh 架构

我们之前写过几个 1 v 1 的栗子，它们的连接模式如下：



这是典型的端到端对等连接，所以当我们要实现多人视频（实际上也就是多端通信）的时候，我们会很自然的想到在 1 v 1 的基础上扩充，给每个客户端创建多个 1 v 1 的对等连接：

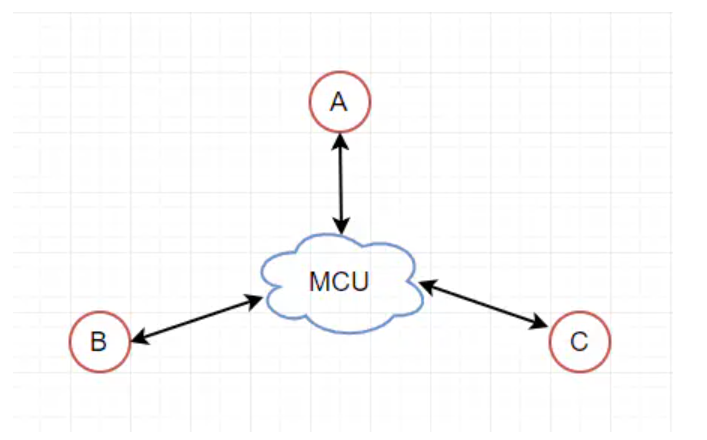


这就是所谓的 Mesh 模式，不需要额外的服务器处理媒体数据（当然，信令服务器是不可少的），仅仅是基于 WebRTC 自身的点对点连接进行通信，本期的实例也是采用这种模式。

但是这种架构的缺点也是十分明显的，如果连接的客户端过多，上行带宽面临的压力将会非常大，相应的视频通话 。

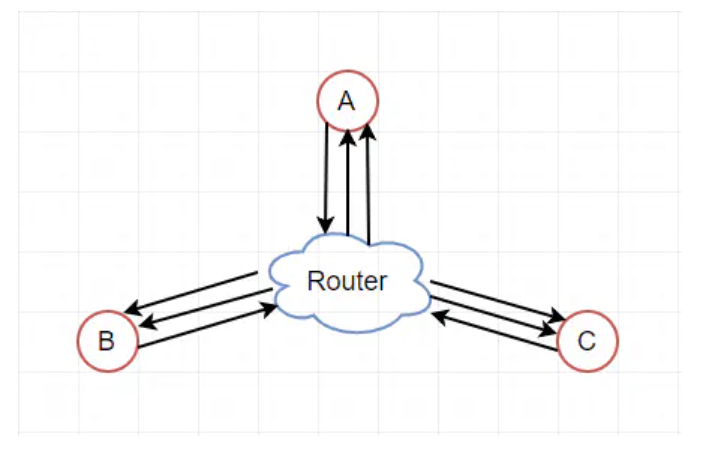
* Mixer 架构

传统的视频会议，一般都是采用 Mixer 架构。以录播摄像为例，会利用 MCU (多点控制单元) 接收并混合每个客户端传入的媒体流。也就是将多个客户端的音视频画面合成单个流，再传输给每个参与的客户端。这样也可以保证客户端始终是 1 对 1 的连接，有效缓解了 Mesh 架构的问题。缺点则是依赖服务端，成本比较大，而且服务端处理过多也更容易导致视频流的延迟。



* Router 架构

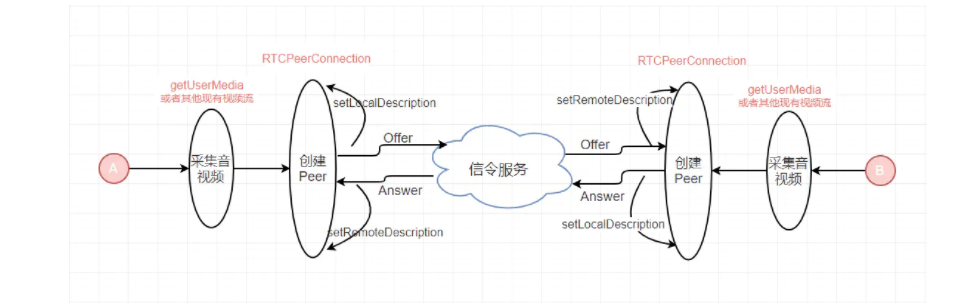
Router 模式和 Mixer 很类似，比较来说，它只是单纯的进行数据流的转发，而不用合成、转码等操作。



## 多人视频

### 1 v 1

我们基于 Mesh 模式来做多人视频的演示，所以需要给每个客户端创建多个 1 v 1 的对等连接。除了 WebRTC 的基础知识，还需要用到 Socket.io 和 Koa 来做信令服务。



先复习一下 1 v 1 的连接过程：

A 创建 offer 信息后，先调用 setLocalDescription 存储本地 offer 描述，再将其发送给 B。

B 收到 offer 后，先调用 setRemoteDescription 存储远端 offer 描述；

然后又创建 answer 信息，同样需要调用 setLocalDescription 存储本地 answer 描述，再返回给 A

A 拿到 answer 后，再次调用 setRemoteDescription 设置远端 answer 描述。

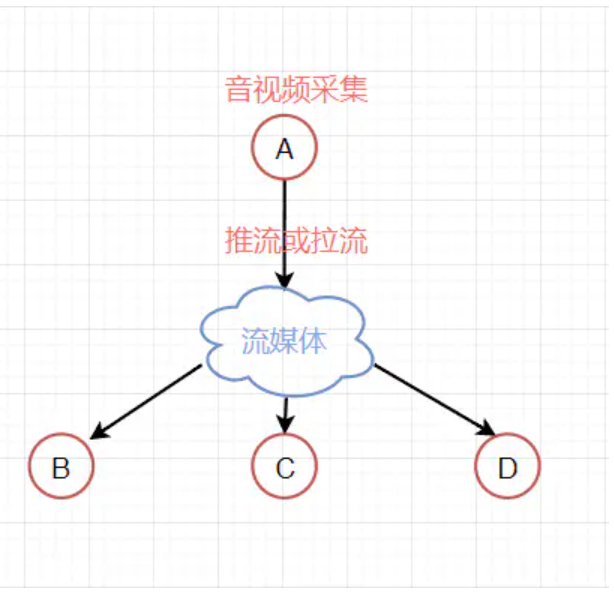
当然，NAT 穿越和候选信息交换也是必不可少的。

本地 ICE 候选信息采集完成后，通过信令服务进行交换。

这一步也是在创建 Peer 之后，但与 offer 的发送没有先后关系。

### 1 v 多

我们平时观看直播实际上就是 1 v 多，也就是只有一端输出视频流，其他观看端只需要接收就好了。但是这种形式，一般不会采用点对点连接，而是用传统的直播方式，服务端进行媒体流的转发。有些直播可以和主播进行互动，这里的原理大致和上篇文章中的共享画板类似。



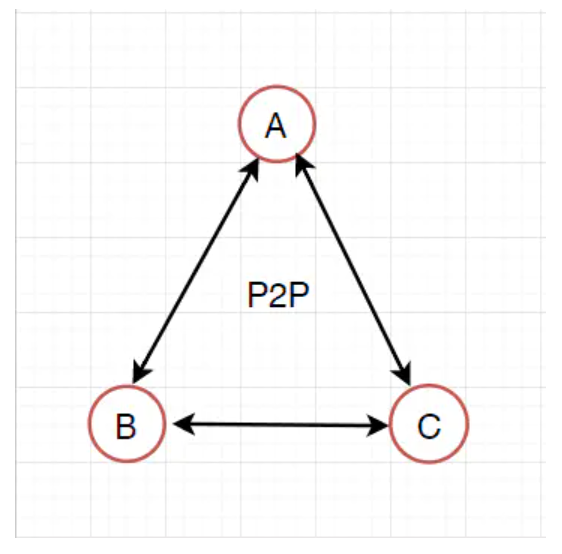
### 多路通话

其实这种情况，主要用于视频会议或者多人视频通话，类似于微信的视频通话一样。

#### 注意事项

我们刚刚回忆过 1 v 1 的连接流程，也知道要基于 Mesh 架构来做，那么到底该如何去做呢？这里先提炼两个要点：

* 如何给每个客户端创建多个点对点连接？
* 如何确认连接的顺序？



我们以 3 个客户端 A、B、C 为例。A 最先打开浏览器或者说 A 是第一个加入房间的，那么 A 进入的时候房间内没有其他人，这个时候要做什么？只需要初始化一下自己的视频画面就好，并不需要进行任何连接操作，因为这个时候没有第二个人，也就没有连接的对象。

什么时候需要进行连接？等 B 加入房间的时候。这里又一个问题，B 加入房间时，谁发送 Offer ？ 因为都参与通话，B 加入的时候首先也会初始化自己的视频流，那么此时 A 和 B 都可以 createOffer 。这也是和之前 1 v 1 的区别所在，因为 1 v 1 我们有明确的 **呼叫端** 和 **接收端**，不需要考虑这个问题。**所以，为了避免连接混乱，我们只用后加入的成员，向房间内所有已加入成员分别发送 Offer，也就是说 B 加入时，给 A 发；C 加入时，再给 A 和 B 分别发。** 以此来保证连接的有序性，这是第二个问题。

那么如何在一个端建立多个点对点连接呢？我采用的策略是，**两两之间的连接，都是单独创建的 Peer 实例**。也就是说，A ——> B 、A ——> C 的连接中，A 会创建两个 Peer 实例，用来分别与 B、C 做连接，同样的 B、C 也会创建多个 Peer 实例。但是我们需要确保每个端之间的 Peer 是一一对应的，简单来说，就是 A 的 PeerA-B 必须和 B 的 peerA-B 连接。很明显，这里需要一个唯一性标识。

// loginname 唯一

// 假设 A 的 loginname 是 A；B 的 loginname 是 B；

// 在客户端 A 中

let arr = ['A', 'B'];

let id = arr.sort().join('-'); // 排序后再连接 A-B

this.PeerList[id] = Peer; // 将创建的 peer 以键值对形式都存放到 PeerList 中

// PS: 在客户端 B 中，操作一样

#### 代码写起来

其实实现多人通信的主要思路刚刚已经讲完了，我习惯于先将思路理清楚，再讲代码实现。个人觉得这样比大家直接看代码注释效果要好，大家有什么好的意见也可以在评论区提出，我们一起讨论。

我们先做一个加入房间的过渡页，简单的 Vue 写法，没啥好说的。

<div class="center">

登录名：<input type="text" v-model="account"> <br>

房间号：<input type="text" v-model="roomid"> <br>

<button @click="join">加入房间</button>

</div>

// ···

methods: {

join() {

if (this.account && this.roomid) {

this.$router.push({name: 'room',

params: {roomid: this.roomid, account: this.account}})

}

// 参数是路由形式的，如 room/id/account

}

}

初始化步骤和前两期 1 v 1 的栗子没有区别，视频通话首先当然是获取视频流。

getUserMedia() { // 获取媒体流

let myVideo = this.$refs['video-mine']; // 默认播放自己视频流的 video

let getUserMedia = (navigator.getUserMedia ||

navigator.webkitGetUserMedia ||

navigator.mozGetUserMedia ||

navigator.msGetUserMedia);

//获取本地的媒体流，并绑定到一个video标签上输出

return new Promise((resolve, reject) => {

getUserMedia.call(navigator, {

"audio": true,

"video": true

}, (stream) => {

//绑定本地媒体流到video标签用于输出

myVideo.srcObject = stream;

this.localStream = stream;

resolve();

}, function(error){

reject(error);

// console.log(error);

//处理媒体流创建失败错误

});

})

}

大家还记不记得，在 1 v 1 中，我们创建 Peer 实例的时机是： **接收端** 点击同意通话后，初始化自己的 Peer 实例；**呼叫端** 收到对方同意申请的通知后，初始化 Peer 实例，并向其发送 Offer。刚刚分析过，多人通信思路有些不一样，但是 初始化方法是差不多的，我们先写个初始化方法。

getPeerConnection(v) {

let videoBox = this.$refs['video-box']; // 用于向 box 中添加新加入的成员视频

let iceServer = { // stun 服务，如果要做到 NAT 穿透，还需要 turn 服务

"iceServers": [

{

"url": "stun:stun.l.google.com:19302"

}

]

};

let PeerConnection = (window.RTCPeerConnection ||

window.webkitRTCPeerConnection ||

window.mozRTCPeerConnection);

// 创建 peer 实例

let peer = new PeerConnection(iceServer);

//向PeerConnection中加入需要发送的流

peer.addStream(this.localStream);

// 如果检测到媒体流连接到本地，将其绑定到一个video标签上输出

// v.account 就是上面提到的 A-B

peer.onaddstream = function(event){

let videos = document.querySelector('#' + v.account);

if (videos) { // 如果页面上有这个标识的播放器，就直接赋值 src

videos.srcObject = event.stream;

} else {

let video = document.createElement('video');

video.controls = true;

video.autoplay = 'autoplay';

video.srcObject = event.stream;

video.id = v.account;

// video加上对应标识，这样在对应客户端断开连接后，可以移除相应的video

videoBox.append(video);

}

};

// 发送ICE候选到其他客户端

peer.onicecandidate = (event) => {

if (event.candidate) {

// ··· 发送 ICE

}

};

this.peerList[v.account] = peer; // 存储 Peer

}

创建 Peer 的时候用到了 account 标识来做保存，这里也涉及到我们建立点对点连接的时机问题。现在我们来看看，之前分析的第二个问题如何体现在代码上呢？

// data 是后端返回的房间内所有成员列表

// account 是本次新加入成员 loginname

socket.on('joined', (data, account) => {

// joined 在每次有人加入房间时触发，自己加入时，自己也会收到

if (data.length> 1) { // 成员数大于1，也就是前面提到的从第二个开始，每个新加入成员发送 Offer

data.forEach(v => {

let obj = {};

let arr = [v.account, this.$route.params.account];

obj.account = arr.sort().join('-'); // 组合 Peer 的标识

if (!this.peerList[obj.account] && v.account !== this.$route.params.account) {

// 如果列表中没有这个标识的 Peer ，则创建 Peer实例

// 如果是自己，就不创建，否则就重复了

// 比如所有成员列表中，有 A 和 B，我自己就是 A，如果不排除，就会创建两个 A-B

this.getPeerConnection(obj);

}

});

if (account === this.$route.params.account) {

// 如果新加入成员是自己，则给所有已加入成员发送 Offer

for (let k in this.peerList) {

this.createOffer(k, this.peerList[k]);

}

}

}

});

我们在初始化 Peer 实例的时候，还做了一个发送 ICE 的操作。那我们就以 ICE 接收为例，看一下这种加了唯一标识的处理和之前有什么区别。

getPeerConnection(v) {

// ··· 部分代码省略

// 发送ICE候选到其他客户端

peer.onicecandidate = (event) => {

if (event.candidate) {

socket.emit('\_\_ice\_candidate',

{candidate: event.candidate,

roomid: this.$route.params.roomid,

account: v.account});

// 将标识 v.account 也放进数据中转发给对方，用于匹配对应的 Peer

}

};

}

// 在mounted 方法中接收

socket.on('\_\_ice\_candidate', v => {

//如果是一个ICE的候选，则将其加入到PeerConnection中

if (v.candidate) {

// 利用传过来的唯一标识匹配对应的 Peer，并添加 Ice

this.peerList[v.account] && this.peerList[v.account].addIceCandidate(v.candidate).catch((e) => { console.log('err', e)

});

}

});

其实区别就是，我们把标识（A-B）也放进了信令交互的数据中，这样才能在两端之前匹配到对应的 Peer 实例，而不至于混乱。

最后，后端代码比较简单，看一下需要注意的点就好。

const users = {};

app.\_io.on( 'connection', sock => {

sock.on('join', data=>{

sock.join(data.roomid, () => {

if (!users[data.roomid]) {

users[data.roomid] = [];

}

// 因为多房间，采用了这种格式保存房间成员

// {'room1': [userA, userB, userC]} userA 包含loginname 和 sock.id

let obj = {

account: data.account,

id: sock.id

};

let arr = users[data.roomid].filter(v => v.account === data.account);

if (!arr.length) {

users[data.roomid].push(obj);

}

app.\_io.in(data.roomid).emit('joined', users[data.roomid], data.account, sock.id);

// 新成员加入时，把房间内成员列表发给房间内所有人

});

});

sock.on('offer', data=>{ // 转发 Offer

sock.to(data.roomid).emit('offer',data);

});

// 这里转发是直接转发到房间了，也可以转发到指定的客户端

// 看过上一篇共享画板的同学应该有印象，没看过的可以去看看，这里就不再多说

sock.on('answer', data=>{ // 转发 Answer

sock.to(data.roomid).emit('answer',data);

});

sock.on('\_\_ice\_candidate', data=>{ // 转发ICE

sock.to(data.roomid).emit('\_\_ice\_candidate',data);

});

})

app.\_io.on('disconnect', (sock) => { // 断开连接时，删除对应的客户端数据

for (let k in users) {

users[k] = users[k].filter(v => v.id !== sock.id);

}

console.log(`disconnect id => ${users}`);

});

# <https://juejin.cn/post/6844903830140747789>

# 遇到的问题

1. Uncaught (in promise) TypeError: Cannot read property 'getUserMedia' of undefined

解释：

1. Uncaught (in promise) DOMException: Could not start video source

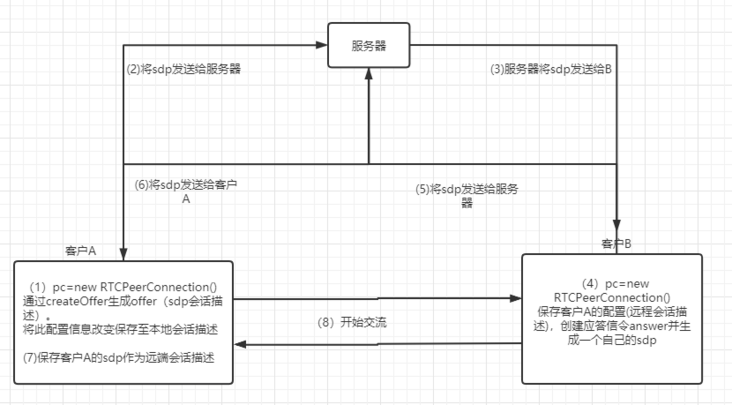
解释：不可以不同浏览器同时占用摄像头

1. 两两用户分别建立独立的peer，两个peer联系后形成一个通道。通话和演示建立在一个通道的基础上

# [webRTC PC端API使用](https://segmentfault.com/a/1190000022863639)



#### webRTC通信过程：

  
步骤：  
1：clientA创建peer对象，打开本地媒体流，将媒体流添加到peer中。调用peer对象的createOffer方法创建一个用于offer的sdp  
2：clientA将sdp传送到服务端  
3：服务端将sdp发送给clientB  
4：clientB创建peer对象,保存clientA的sdp并生成answer的sdp  
5：clientB将sdp传送到服务端  
6：服务端将sdp发送给clientA  
7：clientA保存clientB的sdp到本地  
8：clientA和clientB进行点对点通信

其他补充：  
1：webRTC通信使用tcp协议  
2：点对点通信路由查找相关：ice、iceServers  
3：NAT传透参考博客：[https://www.jianshu.com/p/373...](https://www.jianshu.com/p/3737724bc216?tdsourcetag=s_pctim_aiomsg" \t "_blank)  
4：sdp:通用的会话描述协议，主要用来描述多媒体会话，用途包括会话声明、会话邀请、会话初始化

#### 三个主要API:

1:RTCPeerConnection:构建两个终端连接的api  
2:getUserMedia:访问设备的摄像头及麦克风，获取音频，视频流  
3:RTCDataChannel:传输任意数据，此处用的少

##### RTCPeerConnection相关API:

###### 通信过程使用api及参数：

createOffer():带有特定的配置信息寻找远端匹配机器（peer）的请求  
createAnswer():在协调一条连接中的两端offer/answers时，根据从远端发来的offer生成一个answer  
addStream():添加媒体流到远端  
removeStream():删除媒体流  
onaddstream():监听远端传来的媒体流

###### 通话质量检测api及参数：

getSenders()->getStats():获取发送出去的通话质量数据  
type:发送方向（主要检测inBound-rtp）  
bytesSent:发送的字节数(累计)  
meidaType:媒体类型  
packageSent:发送的包数  
retransmittedPacketsSent:重传包数  
retransmittedPacketsSent:重传字节数

getReceivers()->getStats():获取接收的通话质量数据  
type:发送方向（主要检测inBound-rtp）  
bytesReceived:接收的字节数(累计)  
meidaType:媒体类型  
packetsReceived :接收的包数（累计）  
packetsLost:丢包数  
jitter:抖动值

##### getUserMedia相关API:

###### 获取媒体流：

navigator.mediaDevices.getUserMedia({  
audio:true || {},  
video:true || {}  
}).then((stream)=>{})  
getUserMedia传递的参数为媒体流的配置：如麦克风，摄像头的配置

###### 获取屏幕共享媒体流：

navigator.mediaDevices.getDisplayMedia({  
audio:true || {},  
video:true || {}  
}).then((stream)=>{})  
获取终端设备信息：  
navigator.mediaDevices.enumerateDevices().then(arr=>{})  
获取摄像头，麦克风型号等，常用户摄像头前后切换前获取摄像头数据

###### 媒体流的设置：

stream中视频流和音频流是两个不同的track  
方法一：  
stream.getVideoTracks():获取视频流  
stream.getAudioTracks()：获取音频流  
stream. addTrack():添加某个流  
stream.removeTrack(track):删除某个流  
方法二：  
stream.getTracks().forEach((track)=>{  
track信息：track.kind(track类型)  
设置api:track.stop()，track.replaceTrack()  
})

###### 视频通话过程中摄像头的打开关闭：

stream代表媒体流  
stream.getVideoTracks()[0].enabled=true/false

###### 视频通话过程中切换前后摄像头

1：根据摄像头id重新获取媒体流  
2: 切换原媒体流（replaceTrack），切换代码示例如下：

onsuccess: function(stream){

**let** videoTracks = stream.getVideoTracks();

self.localStream.addTrack(videoTracks[0]);

**if** (!RTCRtpTransceiver.prototype.setDirection) {

pc.getSenders()[1].replaceTrack(videoTracks[0]);

pc.getTransceivers()[1].direction = 'sendrecv';

} **else** {

**if** (pc.getTransceivers().length >= 3) {

p1.getTransceivers()[2].stop();

}

pc.getTransceivers()[1].setDirection('sendrecv'); }

},

# 切换视频流

**实现思路**

1. **首先建立一对一视频通话基础**
2. **在本地用主动触发操作获取屏幕分享流并保存一份在本地**
3. **获取到屏幕分享流之后将PeerConnection中的视频流信息更改，移除本地摄像头流，绑定屏幕分享流**
4. **新的流绑定完成之后，重新和对面建立新的通讯连接，即重新创建offer，对面监听offer以及发送响应**
5. **切换本地摄像头画面和屏幕分享流原理都一样，就是将绑定在peerConnection中的媒体信息更改，然后重新建立链接**

创建视频分享流

//分享屏幕

shareScreenStream(){

const that = this;

const displayMediaStreamConstraints = {

video: {

cursor: "always"

},

audio: true

};

//获取分享窗口流

if (navigator.mediaDevices.getDisplayMedia) {

navigator.mediaDevices.getDisplayMedia(displayMediaStreamConstraints).then(function (mediaStream) {

console.log("mediaStream", mediaStream)

that.shareStream = mediaStream;

//移除本地摄像头流媒体

that.pc.removeStream(that.localStream)

//添加屏幕分享流

that.pc.addStream(mediaStream)

//重新发送offer建立通讯

that.onCreateOffer();

//改变本地视频窗口信息为屏幕分享信息

let video = document.querySelector('#local');

if ("srcObject" in video) {

video.srcObject = that.shareStream;

} else {

video.src = window.URL.createObjectURL(that.shareStream);

video.volume = 0

}

// eslint-disable-next-line no-unused-vars

video.onloadedmetadata = function(e) {

video.play();

};

that.isLocalStream = false;

}).catch(error=>{

console.log("error",error)

that.$message.error("媒体设备获取异常")

});

} else {

console.log("navigator.mediaDevices.getDisplayMedia false");

that.$message.error("浏览器不不支持")

}

切换流

//切回本地摄像头

changeLocalStream(){

const that = this;

//移除视频分享流

that.pc.removeStream(that.shareStream)

//添加本地摄像头流

that.pc.addStream(that.localStream)

//重新建立链接

that.onCreateOffer();

//切换本地媒体展示

that.nativeMedia();

that.isLocalStream = true;

},

