**基于WebRTC的在线音视频会议系统**

**摘 要**

随着互联网技术的飞速发展，人们对实时通讯的要求不断提高。从文字通信、声音通信、一路演变到如今的视频通信。人们希望通过互联网技术，跨越空间限制面对面交流。这是以互联网技术以及计算机硬件系统飞速迭代为基础的。而如今，人们又对实时通讯的主要标准—实时性产生了更高的要求。低延迟、高质量、高并发等等标准不断提高，这对计算机网络和硬件以及音视频技术又是一道挑战。

目前的实时音视频解决方案可以分为两类：客户端和网页端。对于网页版的实时通讯应用，通信双方首先要在浏览器中安装诸如 Adobe Flash、Microsoft Sliverlight 等插件，除去安装和更新的不便，开发者不仅需要对插件进行开发，也需要对不同浏览器和操作系统做兼容处理。对于客户端版的实时通讯应用，开发软件时也需要考虑不同的平台，并且开发出的软件，相互之间一般不能相互通信，比如: 腾讯会议和钉钉用户之间不能直接进行通信，通信的双方需要下载安装同一款软件。从更新和维护的角度来说，二者差不太多。但是从用户的使用体验来说，网页端应用的优势在于：用户永远不需要主动更新，所有的更新以及维护都是在服务端完成的，用户甚至感知不到变化。

WebRTC(Web Real-Time Communication) 技术，是基于网页端的一项实时通讯技术。它解决了浏览器之间互不兼容、插件繁多、性能消耗严重的问题。对Web应用的开发者来说，它极大的节约了开发成本、维护成本和运营成本。对于用户来说，它降低了用户的使用门槛，用户无需安装任何插件，打开网站即能运行。

本文对WebRTC技术以及其相关API进行了深入的研究以及验证。利用这项技术，配合现有的流行框架：Vue、Node.js、WebSocket、Express等，实现了一套完整的音视频会议系统。该系统不依赖任何插件，打开浏览器即能运行，支持多路音视频同时通讯、多人聊天、文件共享等功能。

【关键词】WebRTC 实时通信 视频会议 WebSocket Vue Express Node.js

目 录

[第一章 绪论 4](#_Toc38057339)

[1.1 课题背景 4](#_Toc38057340)

[1.2 国内外发展现状 4](#_Toc38057341)

[1.3 研究内容及结构 6](#_Toc38057342)

[第二章 相关技术介绍 7](#_Toc38057343)

[2.1 整体架构简述 7](#_Toc38057344)

[2.2 WebRTC技术 7](#_Toc38057345)

[2.3 WebSocket 技术 9](#_Toc38057346)

[2.4 Node.js 介绍 9](#_Toc38057347)

[2.5 Vue框架 10](#_Toc38057348)

[第三章 基于WebRTC的视频会议系统实现 11](#_Toc38057349)

[3.1 首页 11](#_Toc38057350)

[3.2 登录模块 12](#_Toc38057351)

[3.3 注册模块 15](#_Toc38057352)

[3.4 创建会议 18](#_Toc38057353)

[3.5 加入会议 20](#_Toc38057354)

[3.6 会议室模块 23](#_Toc38057355)

[3.7 实时聊天 27](#_Toc38057356)

[3.8 文件共享 28](#_Toc38057357)

[3.9 屏幕分享 31](#_Toc38057358)

[第四章 打包上线 32](#_Toc38057359)

[4.1 前端项目打包 32](#_Toc38057360)

[4.2 后端项目部署 32](#_Toc38057361)

[4.3 nginx静态资源服务器配置 33](#_Toc38057362)

[第五章 总结与展望 33](#_Toc38057363)

[6.1 总结 33](#_Toc38057364)

[6.2 个人展望 34](#_Toc38057365)

[参考文献 35](#_Toc38057366)

[致 谢 36](#_Toc38057367)

第一章 绪论

### 1.1 课题背景

随着互联网技术和通信技术的飞速发展，人们对通讯的要求也越来越高，已经不仅限于传统的纸笔书信联系或是固定的手机语音通信。人们的通信正在由传统的语音通信向多媒体 通信发展，如文本，语音，视频。近几年，web 实时通讯已经成为人们互相交流的一种最主要的方式，比传统的运营商语音通信有天然优势，web 实时通信的实现较易，信息类别多， 价格低廉，依赖运营商程度小。即时通信的这些优点将促进它迅速发展，成为互联网的第一大应用。

以浏览器为客户端提供实时通信，扩大浏览器的应用，节省存储量，提升用户体验。现在 HTML5 技术应用已普及，其有着页面的渲染 速度更快，画面更炫，音视频流畅等优势，在未来的发展中，Web 页面已经不仅仅是获取信息功能，还会具有多种应用，一个浏览器可以用来接收邮件、文本消息、音视频、游戏等。

WebRTC 技术是嵌入在浏览器中实时音视频通信技术，实质上是在浏览器中集成了即时通信中所用的音视频捕捉及处理、会话控制和网络传输等功能，因为 WebRTC 的多媒体通信软终端可以使用户避免安装插件的繁琐过程，直接在浏览器上进行实时的音视频通信，用户的体验非常好。另外，基于 WebRTC 的多媒体软终端使用 Javascript 和 HTML5 开发，开发者不必为好多版本来适应操作系统的差异，只需要熟悉业务本身而不必太了解媒体信息本身。

更为重要的是，WebRTC技术通过建立浏览器之间点对点的通信来实现数据传输，在参会用户较少的情况下（3-4人），有着非常出色的性能表现。它不需要流媒体服务器就能实现音视频数据传输，极大的降低了延时、节省了服务器资源。

由此，通过WebRTC技术，我们可以实现一个跨平台的、低成本的、低延时的、易于维护的音视频会议系统。

### 1.2 国内外发展现状

WebRTC技术由谷歌公司在2011年推出并被纳入W3C推荐标准。WebRTC虽然冠以“web”之名，但并不受限于传统互联网应用或浏览器的终端运行环境。实际上无论终端运行环境是浏览器、桌面应用、移动设备（Android或iOS）还是IoT设备，只要IP连接可到达且符合WebRTC规范就可以互通。这一点释放了大量智能终端（或运行在智能终端上的app）的实时通信能力，打开了许多对于实时交互性要求较高的应用场景的想象空间，譬如在线教育、视频会议、视频社交、远程协助、远程操控等等都是其合适的应用领域。

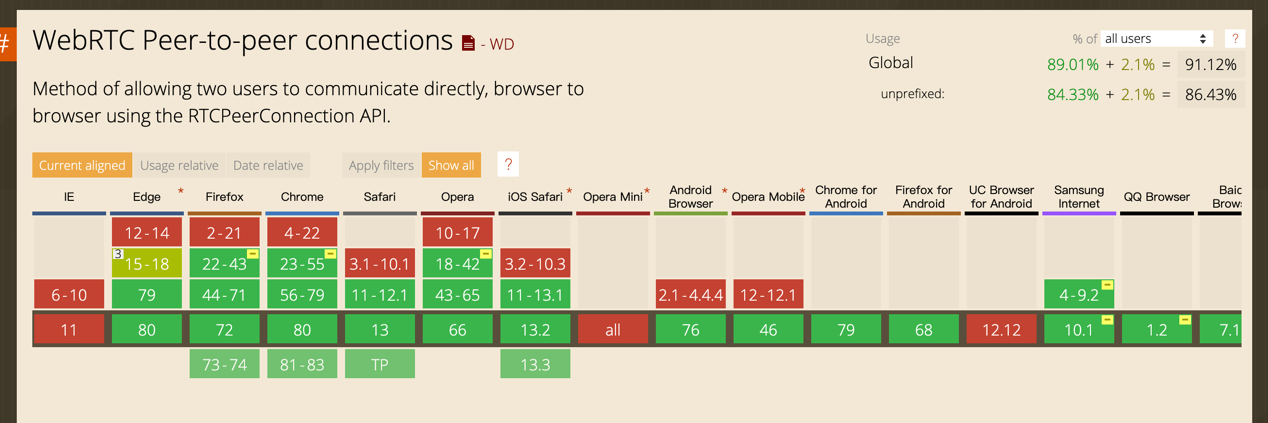
截止目前各大浏览器的支持情况：

图1.2-1 各浏览器对WebRTC技术的支持情况

无论是 Web 还是 Native，都非常依赖 WebRTC 提供的音视频引擎，尤其是在 Web 端，几乎所有浏览器厂商的实现都是基于 Google WebRTC 项目。随着 WebRTC 1.0 标准的定稿，各大浏览器的 WebRTC 接口已经基本得到统一。

2017年，Google 推出了 KITE 开源项目，用于帮助开发者检测 WebRTC 应用在不同浏览器的互通性。对于标准化社区来讲，下一步工作主要会围绕提供一组更完备的测试套件，不仅可以帮助开发者测试 WebRTC 应用在 Web 端、Native 端的互通性与体验，还有助于保证各厂商浏览器 WebRTC 接口功能的一致性，并逐步完善 WebRTC 缺失的功能。

与之前的实时通讯技术相比， WebRTC具有以下显著优势：

（1）用户不再需要安装插件和各种客户端，操作简单，使用方便。

（2）用户不用总是更新客户端，升级在服务器侧完成，提升用户体验。

（3）超文本标记语言（HTML）和JavaScript开发快捷简单、成本低。

（4）跨操作系统，不用兼容操作系统，不需要考虑pc或移动，应用只需开发一个版本即可。

（5）屏蔽了媒体处理，开发者只用关注于应用本身业务逻辑。

全球领先的技术研究和咨询公司Technavio最近发布了题为“全球网络实时通讯（WebRTC）市场， 2017-2021”的报告。报告显示，2017-2021年期间，全球网络实时通信（WebRTC）市场将以34.37％的年均复合增长率增长，增长十分迅速。增长主要来自北美、欧洲及亚太地区。

阿里云、腾讯云、声网Agora 等都有基于WebRTC的音视频服务。

自2014年以来，声网每年都有举办RTC大会：



图 1.2-2 历年WebRTC大会主题

可以看到，WebRTC的发展紧跟互联网主流技术的步伐。另一方面，随电子设备的配置越来越高、在5G、Wi-Fi6等技术即将商用的加持下，RTC技术的短板得到弥补。实时通信技术将得到进一步的发展。

### 1.3 研究内容及结构

本文深入研究了 WebRTC 技术及相关的 Web 实时通信技术， 与当前主流的前后端分离架构结合，实现了一套完整的视频会议系统。

第一章 绪论。阐述了课题的研究背景以及当前国内外的现状，分析了WebRTC技术目前的优缺点以及对其将来的应用前景做了简要分析。

第二章 相关技术知识介绍。这一章将对该系统实现所需要的技术手段以及框架做出详细介绍。尤其是WebRTC技术，从获取本地媒体流的MeidaStream API、 到建立WebRTC连接的RTCPeerConnectin API、再到WebRTC数据传输的RTCDataChannel API。 这些接口是实现该系统必不可少的，本文将对其进行详尽的分析。

第三章 基于 WebRTC 的视频会议系统的设计与实现。本章节是本文的核心内容，介绍了该系统从前端到后端的整体逻辑。依次介绍了登录注册模块、创建会议模块、加入会议模块、会议室模块、实时聊天模块、文件传输模块、屏幕分享模块的具体实现方式。

第四章 打包上线。本章介绍了如何将该系统从本地打包发送到腾讯云服务器上部署，使得用户可以通过网址访问。包括搭建nginx服务器、MySQL配置、nodeJS环境配置、https证书配置（该系统需要使用到摄像头，需要https协议才能正常使用）、域名配置等。

第五章 总结与展望。

第二章 相关技术介绍

### 2.1 整体架构简述

本课题研究过程以及实现将在macOS Mojave系统上完成。开发工具将使用VScode（Visual Studio Code）,利用git实现版本控制，Chrome浏览器调试代码，后端服务依赖于Node.js，前端框架使用Vue全家桶，UI框架使用Vuetify，数据库使用mysql。开发模式为前后端分离模式，由前端驱动，即根据前端的功能驱动开发后端接口。

使用Vue搭建前端页面和组件，通过Vue Router来控制路由和页面之间的对应关系，使用VueX进行统一状态管理。

后端服务使用Node.js来实现，通过当前主流的express框架，不仅可实现快速搭建http服务的需求，还能搭配Socket.io实现websocket服务，完成信令服务器的搭建。

数据库使用MySQL，可与后端接口配合，实现数据的持久化。

### 2.2 WebRTC技术

WebRTC (Web Real-Time Communications) 是一项实时通讯技术。与传统音视频传输方案不同的是，它允许网络应用或者站点，在不借助中间媒介的情况下，建立浏览器之间点对点（Peer-to-Peer）的连接，实现视频流和（或）音频流或者其他任意数据的传输。

传统音视频传输方案与WebRTC音视频传输方案：

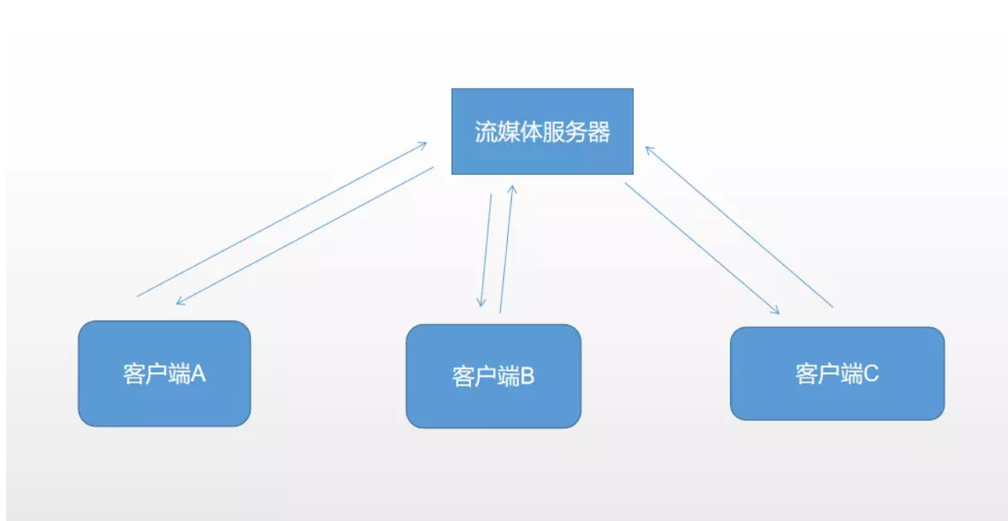


图 2.2-1 传统音视频传输方案

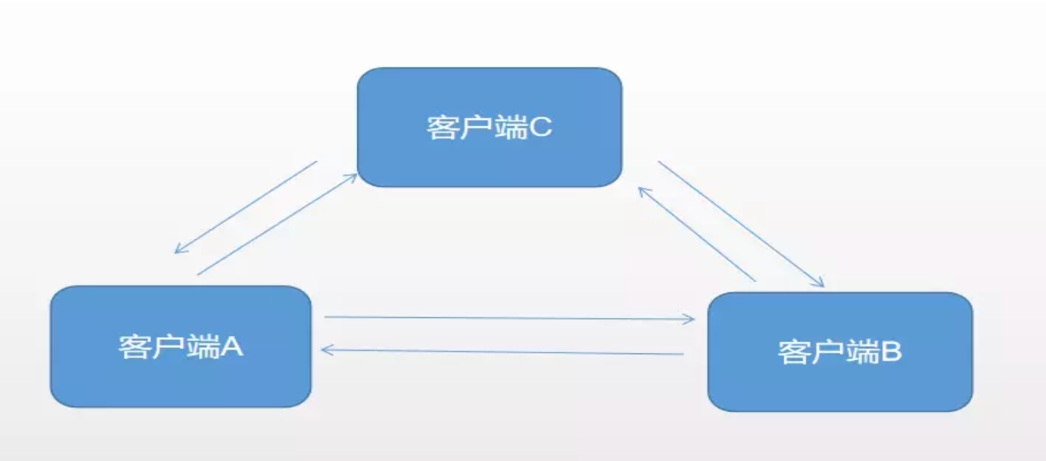
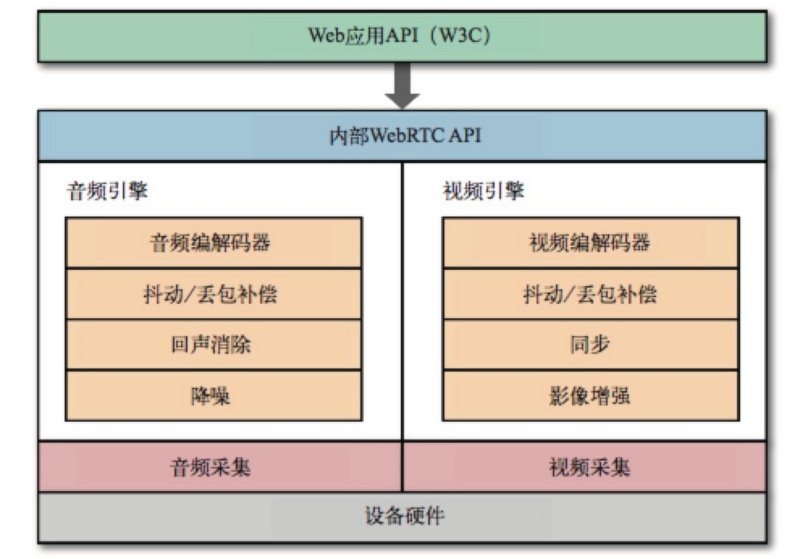


图 2.2-2 WebRTC音视频传输方

可以看到，WebRTC音视频传输不依赖流媒体服务器，这将极大的节省服务器资源。

WebRTC技术将计算机系统层的硬件设备与浏览器进行打通，使得开发者无需关心底层的实现，只需按一定的步骤来调用各个封装好的接口，即可在多个客户端之间进行数据传输。

如下图：

图 2.2-3 WebRTC结构简图

作为开发者，我们只需要关注Web的应用层API（上图绿色部分），而不用去管底层包括音视频采集、降噪、网络、编解码等复杂的工作。

WebRTC技术是基于浏览器的，而且是跨平台的，因此用户可以不需要下载任何APP就能在手机以及其他移动设备或电脑上使用它。简单的说，使用WebRTC进行音视频通信只需要以下步骤：

1、通过浏览器自带的MediaStream接口获取本地摄像头和麦克风捕捉到的视频流和音频流；

2、通过浏览器自带的RTCPeerConnection接口在用户之间建立WebRTC连接；

3、连接建立完成后将本地媒体流挂载到连接上。

至此，用户之间就能进行音视频通信。

### 2.3 WebSocket 技术

WebSocket是HTML5的重要组成部分，WebSocket通信协议是为了解决客户端和服务器的即时通讯问题，它是由IETF根据HTTP协议制订的WebSocket通信协议，和标准的Socket协议有差异，他是在OSI七层模型的上层，是HTTP/1.1协议的升级版[4]。WebSocket 是在浏览器与服务器之间建立一条全双工的通信信道，这个信道上实现浏览器的“拉”信息和服务器的“推”信息的功能。这样可以消除轮询中的可移植和连接难题，也还能为 Ajax轮询提供了更优的解决方案。WebSocket的两个重要组成部分分别是W3C制订的 WebSocket API 和WebSocket，其中在WebSocketAPI中，浏览器与服务器之间做一个握手的信息交换，就可以在浏览器与服务器之间形成相互传送数据的快速通道。

Socket.io是一个基于WebSocket技术进行二次封装的、专门为实时通信Web APP设计的JavaScript库。它支持Web客户端和服务器之间的实时双向通信。它包含两个部分：在浏览器中运行的客户端库，以及用于Node.js的服务端库。二者的API几乎一样，使得开发者可以更轻松的完成开发。值得注意的是，它是事件驱动的。也就是说客户端的服务端需要监听相应的事件才能完成各种逻辑的处理。

本系统中的聊天系统、建立WebRTC连接所用到信令系统，都将基于Socket.io来实现。

### 2.4 Node.js 介绍

Node（正式名称 Node.js）是一个开源的、跨平台的JavaScript运行时环境，有了它，开发人员可以使用 JavaScript 创建各种服务器端工具和应用程序。此运行时主要用于浏览器上下文之外（即可以直接运行于计算机或服务器操作系统上）。据此，该环境省略了一些浏览器专用的 JavaScript API，同时添加了对更传统的 OS API（比如 HTTP 库和文件系统库）的支持。

从 web 服务器开发的角度来看，Node 有很多好处：

（1）卓越的性能表现！Node 为优化 web 应用的吞吐量和扩展度而生，对常见的 web 开发问题是一套绝佳方案（比如实时 web 应用）。

（2）使用JavaScript进行编码，这意味着在客户端和服务器端“上下文切换”的时间成本更低。

（3）与传统的 web 服务器语言（例如 Python、PHP 等）相比，JavaScript 理念更新，语言设计的改进带来了诸多好处。许多其它新近流行的语言也可编译/转换成 JavaScript，所以TypeScript、CoffeeScript、ClojureScript、Scala、LiveScript 等等也可以使用。

（4）Node 包管理工具（node package manager，NPM）提供了数十万个可重用的工具包。它还提供了一流的依赖解决方案，可实现自动化工具链构建。

（5）Node.js 是可移植的，可运行于 Microsoft Windows、macOS、Linux、Solaris、FreeBSD、OpenBSD、WebOS 和 NonStop OS。此外，许多 web 主机供应商对其提供了良好支持（包括专用的基础框架和构建 Node 站点的文档）。

（6）它有一个非常活跃的第三方生态系统和开发者社区：npm，很多人愿意提供帮助。

Express.js 是一个免费的、开源的、基于Node.js的Web应用程序框架。它是许多其它流行 Node 框架的底层库。它提供了以下机制：

（1）为不同 URL 路径中使用不同 HTTP 动词的请求（路由）编写处理程序。

（2）集成了“视图”渲染引擎，以便通过将数据插入模板来生成响应。

（3）设置常见 web 应用设置，比如用于连接的端口，以及渲染响应模板的位置。

（4）在请求处理管道的任何位置添加额外的请求处理“中间件”。

虽然 Express 本身是极简风格的，但是开发人员通过创建各类兼容的中间件包解决了几乎所有的 web 开发问题。这些库可以实现 cookie、会话、用户登录、URL 参数、POST 数据、安全头等功能。

我们可以通过express框架，快速搭建一个功能完善的http服务，在本系统中，它将为我们提供所有的后端api服务。

### 2.5 Vue框架

Vue.js（读音 /vjuː/, 类似于 view） 是一套构建用户界面的 渐进式框架。与其他重量级框架不同的是，Vue 采用自底向上增量开发的设计。Vue 的核心库只关注视图层，并且非常容易学习，非常容易与其它库或已有项目整合。另一方面，Vue 完全有能力驱动采用单文件组件和 Vue 生态系统支持的库开发的复杂单页应用。Vue.js 的目标是通过尽可能简单的 API 实现响应的数据绑定和组合的视图组件。

Vue不仅自身功能强大，而且还有丰富的周边生态：Vuex、Vue Router、Vue-cli等。这些由Vue官方开发和维护的工具极大的简化了项目搭建、路由管理、状态管理、http请求等前端开发流程。使得我们可以快速搭建符合需要的前端页面。

本系统的前端页面将基于Vue以及其周边工具来搭建。

第三章 基于WebRTC的视频会议系统实现

本章节将分别从以下几个小节描述基于WebRTC的视频会议系统的实现逻辑：首页、登录模块、注册模块、创建会议模块、加入会议模块、会议室模块、聊天模块、参会人员列表模块、文件传输模块、屏幕分享模块。

注意，为了节省篇幅，本章所有的代码部分只展示部分重要逻辑（js）代码，结构（html）和样式（css）代码省略。

### 3.1 首页

实现效果：



图 3.1-1 系统首页

首页主要由3个功能按钮组成：右上角的登录按钮、banner里面的加入会议按钮和创建会议按钮，点击不同的按钮可以跳转到不同的页面。

实现代码：

methods: {

joinRoom() {

this.$refs.joinModal.show()

},

createRoom() {

this.$refs.createModal.show()

},

logout() {

this.updateLoginUser(null)

},

goLogin() {

this.$router.push({

name: 'Login'

})

},

goHome() {

this.$router.push({

name: 'Home'

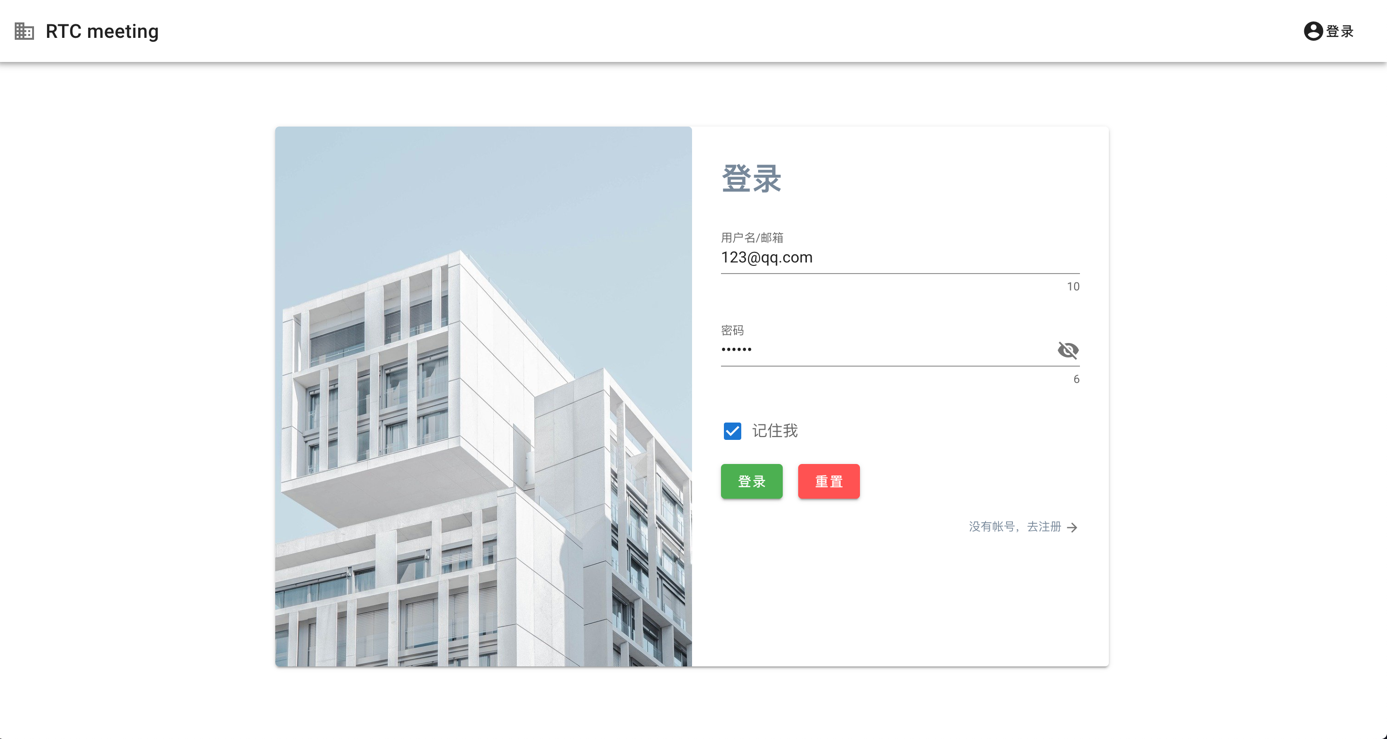
})

}

}

### 3.2 登录模块

实现效果：

图3.2-1 登录页面

登录页面由一个简单的登录组件构成，用户输入的用户名和密码将通过/user/login接口发送到后台，后台将用户名和经过md5加密后的密码通过数据库进行查询和比对，此时有以下几种情况：

1、用户不存在。即数据库中没有与用户输入的用户名相同的记录，会提示用户先进行注册。

2、密码错误。即用户输入的用户名与密码不匹配，会提示用户输入正确的密码。

3、登录成功。此时后台会根据登录信息查询到数据库中的用户信息，并用这些信息生成一个有有效期的token，用作下一次请求接口的凭证，并将这个token返回给前端。前端拿到这个token后存入local storage中，然后跳转至首页。在该token的有效期内以后每次请求接口的时候都会带上这个信息，向后台提供自己的身份。

实现代码：

前端：输入字段验证、请求登录接口、跳转至注册

passwordRules: [

*v* => !!v || '请输入密码',

*v* => (v && v.length <= 16 && v.length >= 6) || '请输入6-16位密码'

],

nameRules: [

*v* => !!v || '请输入用户名/邮箱',

*v* => /.+@.+\..+/.test(v) || '请输入格式正确的邮箱'

]

methods: {

login() {

login({

name: this.username,

password: this.password

}).then(*res* => {

this.$emit('tip', {

code: res.data.code || 0,

msg: res.data.msg

})

// 登录成功

if (res.data && res.data.code === 0) {

this.updateLoginUser(res.data.user)

this.$router.push({

name: 'Home'

})

}

})

},

reset() {

this.$refs.loginForm.reset()

},

goRegister() {

this.$router.push({

name: 'Register'

})

}

}

后台登录接口：

/\*\*

\* @description 用户登录接口

\* @params Object

\* @params name/email,password

\*/

user.post('/login', (*req*, *res*) => {

const user = req.body

if (!user.name || !user.password) {

res.json({ code: -1, msg: '用户名或密码不得为空' })

} else {

query('SELECT \* FROM user WHERE name = ? OR email = ?', [user.name, user.name]).then(*row* => {

if (row.length < 1) {

res.json({ code: -1, msg: '用户不存在' })

} else {

let findUser = row[0]

if (md5(user.password) === findUser.password) {

// 生成token

const token = jwt.sign({

user\_id: findUser.id,

user\_name: findUser.name,

user\_avatar: findUser.avatar,

user\_role: findUser.super

}, secretKey, {

expiresIn: 60 \* 60 \* 24 \* 3

})

res.json({

code: 0,

msg: '登录成功',

token,

user: {

id: findUser.id,

name: findUser.name,

role: findUser.super,

avatar: findUser.avatar

}

})

} else {

res.json({

code: -1,

msg: '密码错误'

})

}

}

})

}

})

### 3.3 注册模块

实现效果：



图3.3-1 注册页面

注册页面提供四个输入框和一个必要的注册按钮，用户输入对应信息后，首先会在前端进行判空和密码一致验证，验证通过后相关数据会通过user/register接口发送给后台，后台拿到注册信息后有以下几种情况返回：

1、用户所填信息不完整。此时后台不会将这些信息写入到数据库，而是会返回错误码，并提示用户缺少必填字段。

2、帐号已经注册。在向数据库插入记录之前，回台会先进行一次查询，如果数据库中已经存在对应的帐号信息，会提示前端用户已经注册。

3、注册成功。后台会将相关信息插入到数据库中，然后走与上一小节登录成功后的逻辑：生成token，返回必要信息，跳转到首页。

前端代码：输入字段验证、请求注册接口、跳转至登录

nameRules: [*v* => !!v || '请输入用户名'],

emailRules: [

*v* => !!v || '邮箱不得为空',

*v* => /.+@.+\..+/.test(v) || '邮箱格式不正确'

],

passwordRules: [

*v* => !!v || '请输入密码',

*v* => (v && v.length <= 16 && v.length >= 6) || '请输入6-16位密码'

],

passwordRules2: [

*v* => !!v || '请输入确认密码',

*v* => this.password2 === this.password1 || '两次密码不一致'

]

methods: {

register() {

register({

name: this.username,

email: this.email,

password: this.password1

}).then(*res* => {

this.$emit('tip', {

code: res.data.code,

msg: res.data.msg

})

if (res.data && res.data.code === 0) {

this.updateLoginUser(res.data.user)

this.$router.push({

name: 'Home'

})

}

})

},

reset() {

this.$refs.registerForm.reset()

},

goLogin() {

this.$router.push({

name: 'Login'

})

}

}

后台注册接口：

/\*\*

\* @description 注册接口

\* @params Object

\* @params name,email,password

\*/

user.post('/register', (*req*, *res*) => {

const user = req.body

if (!user.name || !user.email || !user.password) {

res.json({ code: -1, msg: '缺少必填字段' })

} else {

// 先查询是否已经注册

query('SELECT \* FROM user WHERE email = ?', [user.email]).then(*row* => {

if (row.length > 0) {

res.json({ code: 1001, registered: 1, msg: '该账号已注册，请直接登录' })

} else {

const encryptedPassword = md5(user.password)

query('INSERT INTO user (name, email, password) VALUES (?, ?, ?)',

[

user.name,

user.email,

encryptedPassword

]).then(*row* => {

console.log(row)

const token = jwt.sign({

user\_id: row.insertId,

user\_name: user.name,

user\_avatar: '',

user\_role: 0

}, secretKey, {

expiresIn: 60 \* 60 \* 24 \* 3

})

res.json({ code: 0, msg: '注册成功',token, user: {

id: row.insertId,

name: user.name,

avatar: '',

role: 0

}})

}).catch(*err* => {

console.log(err)

res.json({ code: -1, msg: '注册失败' })

})

}

})

}

})

### 3.4 创建会议

实现效果：



图3.4-1 创建会议弹框

创建会议需要先登录。点击创建会议后会弹出一个表单对话框，用户填入相关信息后点击创建会议，此时会通过/room/create向后台发送创建会议请求，后端会生成参加码（参加码是进入会议室所需要的必要信息，并且是后端websocket服务用以区分不同会议室的唯一标志）等会议信息，并将会议信息插入数据库的会议记录表。会议创建成功后会直接进入会议室，作为会议发起人，进入会议室后只需要连接至wss服务器，然后等待其他人加入。后端会在创建会议返回的接口中返回会议参加码和密码等信息，并在前端展示出来，方便分享给其他人。

前端代码：

createRoom({

owner: 'hhh',

owner\_id: '123',

title: this.title,

need\_password: this.needPassword,

password: this.joinPassword

}).then(*response* => {

let res = response.data

if (res.code === 0 && res.data && res.data.code) {

this.$router.push({

path: '/room',

query: {

password: this.joinPassword || '',

code: res.data.code,

name: this.nickName,

// as the initiator

title: this.title,

role: 0

}

})

}

})

后端接口：

/\*\*

\* @description: 创建会议

\* @param {owner,owner\_id,need\_password,title,password}

\* @return:

\*/

room.post('/create', (*req*, *res*) => {

const data = req.body

if (!data.owner || !data.owner\_id) {

res.json({ code: -1, msg: 'Nice try.' })

}

// 需要密码但是没有密码

if (data.need\_password && !data.password) {

res.json({ code: -1, msg: '请输入密码' })

}

// 生成参加码

let code = Math.random().toString().substring(2, 8)

let needPassword = data.need\_password || 0

let password = needPassword ? data.password : 'null'

// 查一下当前有没有正在进行中的会议参加码跟这个一样

try {

query('SELECT \* FROM room WHERE code = ? AND status = 1', [code]).then(*row* => {

if (!row.length) {

query(

'INSERT INTO room (code, owner, owner\_id, title, need\_password, password) VALUES (?,?,?,?,?,?)',

[

code,

data.owner,

data.owner\_id,

data.title,

needPassword,

password

]

).then(*row* => {

if (row.affectedRows > 0) {

res.json({

code: 0,

data: {

code: code,

password: password

},

msg: '创建成功'

})

}

})

}

})

} catch (e) {

res.json({ code: -1, msg: '系统错误' })

console.log(e)

}

})

### 3.5 加入会议

实现效果：



图3.5-1 加入会议弹框

加入会议不需要登录。与创建会议的逻辑一样，点击加入会议按钮后会弹出一个表单对话框。用户需要输入会议参加码和昵称，并通过/room/join接口发送给后端，后端通过查询数据库将会议信息返回给前端。前端收到响应后先将自己的临时身份存入local storage中，然后带着会议信息进入会议室。

前端代码：

join() {

if (this.validate()) {

JoinRoom({

room\_id: this.code,

password: this.password

}).then(*res* => {

if(res.data && res.data.code === 0){

this.$router.push({

path: '/room',

query: {

code: res.data.room\_id,

password: this.password,

name: this.nickName,

title: res.data.title,

// as the participator

role: 1,

type: 'screen'

}

})

} else {

this.$emit('tip', {

code: -1,

msg: res.data.msg

})

}

}).catch(*error* => {

this.$emit('tip', {

code: -1,

msg: res.data.msg

})

})

}

},

加入会议后台接口：

/\*\*

\* @description: 加入会议

\* @param {type}

\* @return:

\*/

room.post('/join', (*req*, *res*) => {

try {

const data = req.body

if (!data.room\_id){

res.json({code: 1001,msg:'请输入参加码'})

}

query('SELECT \* FROM room WHERE code = ?', [data.room\_id]).then(*row* => {

// 会议不存在

if(row.length < 1) {

res.json({

code: 1002,

msg: '会议不存在'

})

} else {

let meeting = row[0]

// 会议存在但已结束

if(meeting.status == 0) {

res.json({

code: 1003,

msg: '当前会议已结束'

})

} else if(meeting.need\_password && meeting.password !== data.password){

res.json({

code: 1004,

msg: '参会密码错误'

})

} else {

res.json({

code: 0,

room\_id: meeting.code,

title: meeting.title,

need\_password: meeting.need\_password

})

}

}

})

} catch (error) {

res.json({

code: -1,

msg: 'system error'

})

}

})

### 3.6 会议室模块

会议室是本系统的核心模块。用户通过创建会议或者加入会议后会进入此模块。针对用户角色的不同（会议发起者和参与会议者）会做出不同的反应。

对于加入会议室的用户来说，无论是会议发起者还是加入会议者，首先会先都会调用getUserMedia接口获取本地音视频流，并通过video标签展现本地音视频（如图3.6-1）。在获取到本地音视频流后会与websocket服务器建立连接，然后向wss发送user\_join信令。会议室里的用户都将会收到这个信令。并对加入的用户作出一定的响应。

在参与会议的成员之间开始共享音视频数据之前，会先进行WebRTC的连接。也就是本系统的核心接口：RTCPeerConnection APIs的相关实现。

为了阐述WebRTC连接建立的逻辑关系，下面以A（发起会议）和B（加入会议）进行会议为例：

1. A创建一个RTCPeerConnection对象。

2. A使用RTCPeerConnection .createOffer()方法产生一个offer。（一个SDP会话描述）

3. A用生成的offer调用setLocalDescription()，设置成自己的本地会话描述。

4. A将offer通过信令服务器发送给B。

5. B用A的offer调用setRemoteDescription()，设置成自己的远端会话描述，以便他的RTCPeerConnection知道A的设置。

6. B调用createAnswer()生成answer。

7. B通过调用setLocalDescription()将其answer设置为本地会话描述。

8. B然后使用信令机制将他的answer发回给A。

9. A使用setRemoteDescription()将B的应答设置为远端会话描述。

此时，A和B之间的连接建立完毕。但此时还不能进行数据交互，原因是这样生成的offer和answer是不被信赖的。在实际应用中，在创建RTCPeerConnection对象的时候需要传入一个configuration参数，这个参数会提供一个可信赖的STUN/TURN 服务器列表。

通过STUN/TURN 服务器来完成 Candidate 地址的收集，进行 NAT/防火墙穿越，也就是我们常说的ICE打洞连接，简单的说就是通过一个中间层服务器在客户端与客户端之间找出一条通信链路，以便它们进行点对点通信。本系统使用了自己搭建的TURN服务器和谷歌官方提供的STUN服务器作为ICE备选列表，以保证系统的稳定。

pcList: [],

// ICE服务器列表

pcConfig: {

iceServers: [

{

urls: 'turn:129.211.64.178:3478',

username: 'kefeng',

credential: '123456'

},

{

urls: 'stun:stun.l.google.com:19302'

},

{

urls: 'stun:stun1.l.google.com:19302'

},

{

urls: 'stun:stun2.l.google.com:19302'

}

]

},

RTCPeerConnection对象依赖于事件驱动，其中一个icecandidate，该事件在为RTCPeerConnection对象设置了正确的STUN/TURN服务器并且连接创建成功后触发，对应的回调中可以拿到一个ice备选icecandidate，此时需要将这个备选icecandidate通过信令服务器发送给远端。远端在拿到这个icecandidate后，调用PeerConnection的addIceCandidate方法，将icecandidate添加到连接中。

以下代码是icecandidate事件触发后，向信令服务器发送icecandidate信令的过程：

// 给peerConnection绑定事件

peerConnection.onicecandidate = *event* => {

console.log('ice 成功获取:')

if (event && event.candidate) {

socket.send({

type: 'icecandidate',

icecandidate: event.candidate,

from: this.name,

to: data.user\_name,

room\_id: this.code,

user\_role: this.role,

user\_number: this.number

})

}

}

服务端对此信令进行监听，然后将收到的ice信息发送给远端用户：

// icecandidate处理

if (data.type === 'icecandidate') {

io.to(data.room\_id).emit('message', {

type: 'icecandidate\_res',

icecandidate: data.icecandidate,

from: data.from,

to: data.to,

room\_id: data.room\_id,

user\_role: data.user\_role,

user\_number: data.user\_number

})

}

远端用户需要对icecandidate\_res信令进行监听，然后调用findAndSetIceCandidate方法进行处理：

if (data.type === 'icecandidate\_res') {

try {

if (data.to === this.name) {

this.findAndSetIceCandidate(data)

}

} catch (e) {

console.log(e)

}

}

至此，客户端之间的链路已经联通，数据开始交换。各方的PeerConnection都能将本地的媒体流通过addStream方法挂载到PeerConection上，各端都能看到对方的画面。

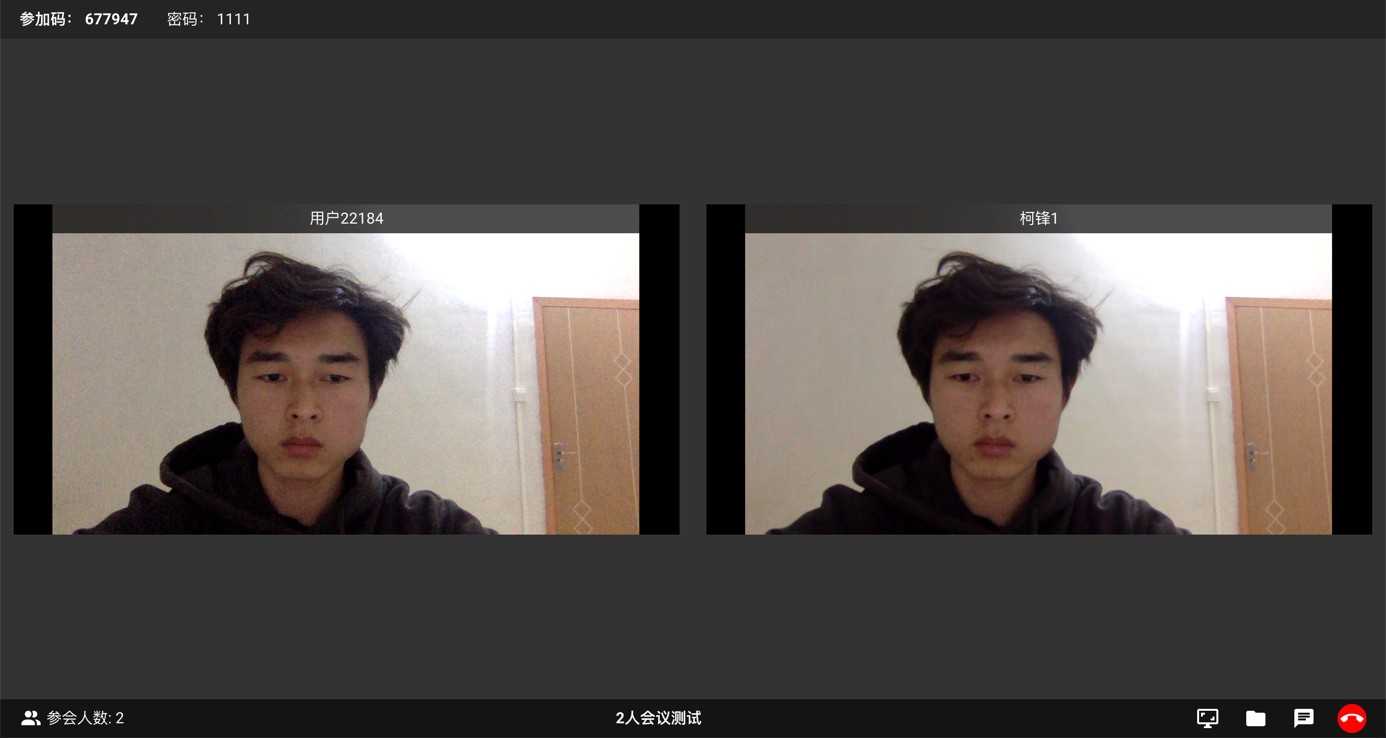


图3.6-1 2人会议效果

值得注意的是，在多人会议（大于2人）的情况中，情况就不是这么简单了。举一个例子来说，一场3人的实时会议，会议各方共需要建立6条peerConnection连接：任何参会者都需要与除自己以外的其他人建立一条peerConnection通道。而且，多人会议中还需要涉及到用户加入时间的不确定性。此时，就需要更加复杂的逻辑来处理这些问题。

在本系统中，参会列表的维护不是在服务器端而是在前端，这样可以避免服务器因为需要维护太多的数据造成内存和处理速度的下降。将用户列表放在前端维护可以分担原本处于服务器单方的压力。本系统的用户列表由发起会议的用户维护，具体逻辑是这样的：

1、任何用户加入会议都会给wss发送user\_join信令；这个信令包含用户的name，role，number，room\_id等信息；

2、wss会将user\_join信令转发给会议室内的所有用户，但是只有会议发起者会收集这些信息并存在客户端的内存中；

3、会议发起者收到user\_join信令后会立马更新内存中的userList数组，然后通过user\_update信令将这个存有当前会议室所有用户信息的数组发送给wss；

4、wss收到user\_update信令后会通过user\_update\_res信令将最新的用户列表发送给会议室的每一位成员；

5、房间内的所有成员都会监听到user\_update\_res信令，此时，各端都会对这个由wss发来的userList进行处理：如果此userList与上次的一样，则不进行任何处理。如果userList与原来的相比有更新，则会对更新项分别进行处理。对每一项（即每一个新加入的用户）都建立一个peerConnection并且对每一个peerConnection都产生一个offer并通过setLocalDescription设置为该peerConnection的本地会话描述；

根据上文中A和B的连接建立过程完成后续过程。

至此，多人连接逻辑处理完毕，此时可进行2人以上的视频会议。

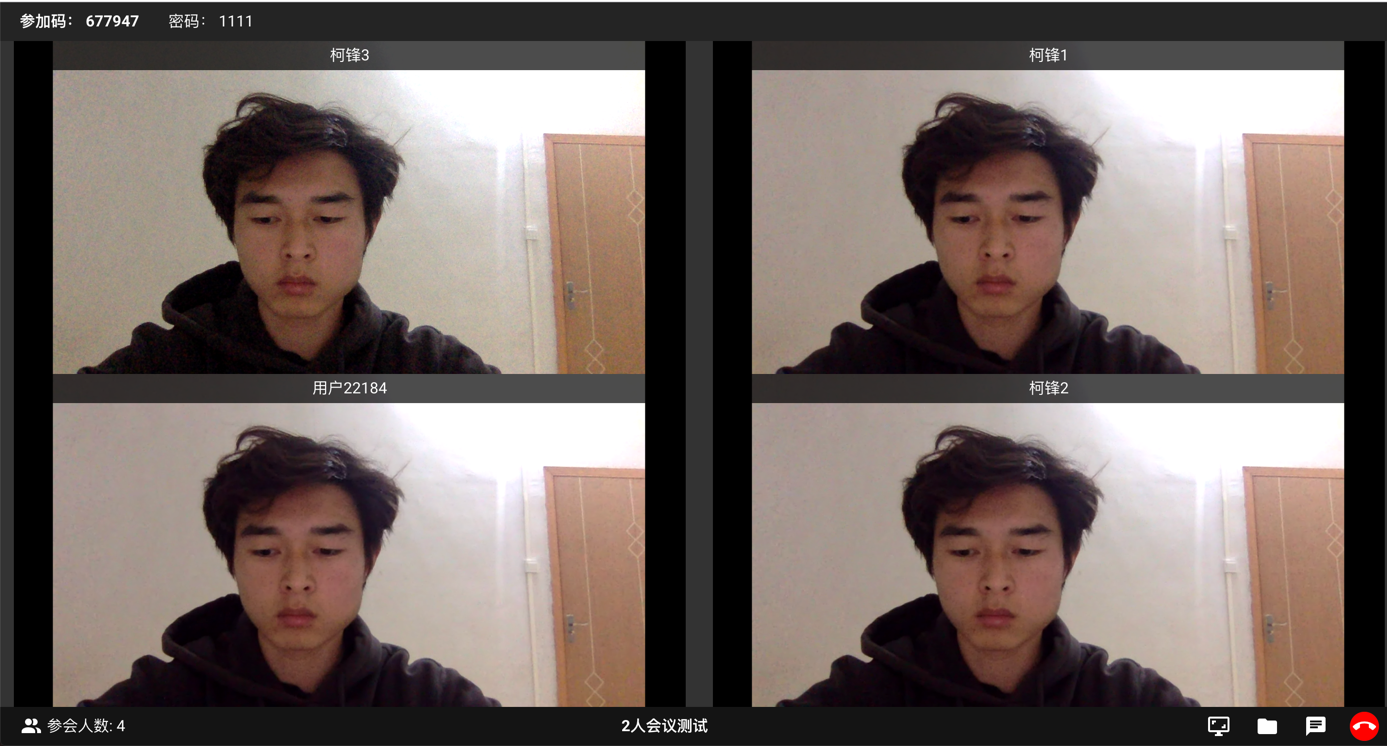


图3.6-3 多人会议效果

### 3.7 实时聊天

实现效果：

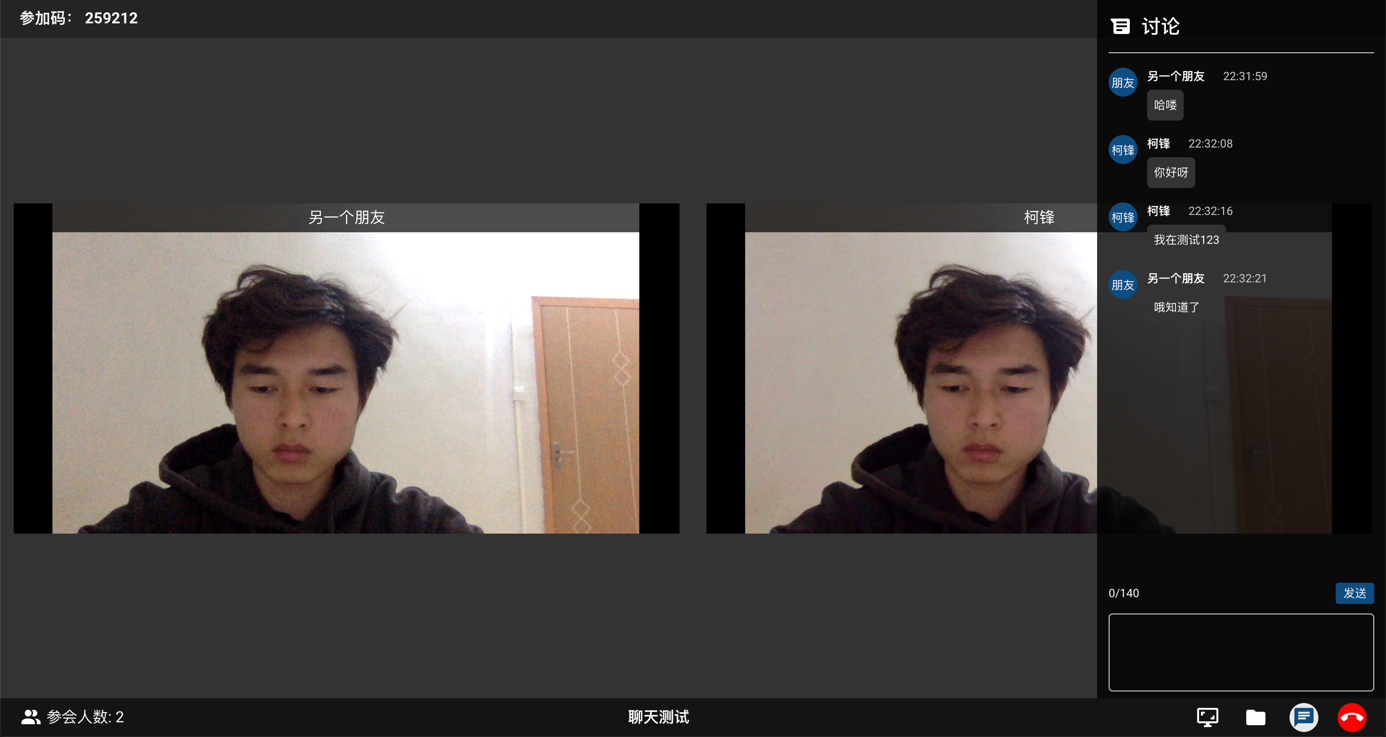


图3.7-1 实时聊天效果

聊天组件由底部的按钮控制，点击可以打开或关闭。用户在聊天框输入的信息，将通过信令服务器发送到会议室，每个参与会议的用户都能收到这个信息并在聊天框中显示出来。

前端发送chat信令：

chatSend() {

if (!this.content.trim()) {

return

}

socket.send({

type: 'chat',

from: this.user.name,

time: this.getTime(),

content: this.content,

room\_id: this.user.code,

number: this.user.number

})

},

后台处理chat信令：

// 聊天

if (data.type === 'chat') {

console.log(data)

io.to(data.room\_id).emit('message', {

type: 'chat\_res',

from: data.from,

content: data.content,

time: data.time,

number: data.number

})

}

### 3.8 文件共享

实现效果：

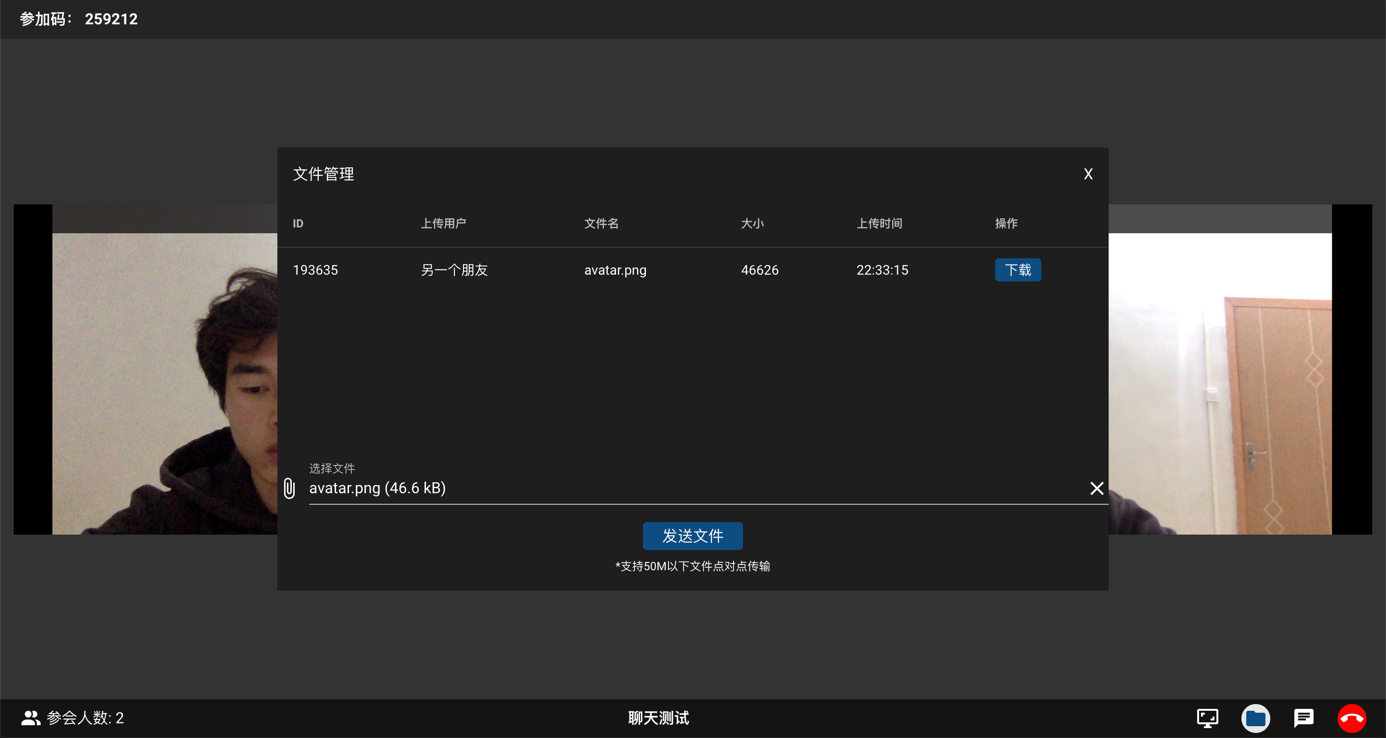


图3.8-1 文件传输组件

文件共享组件集成在会议室页面中，同一个会议室内的各个成员之间可以在一个公共的文件库中上传或下载文件。文件的传输依赖RTCPeerConnection的dataChannel API。这个API可以在端对端之间已经建立连接的情况下，建立一条额外的dataChannel信道，可以传输任何数据格式的文件。在本系统中，利用input标签的file属性可以从系统中选择一个文件，当用户点击发送文件按钮后，将遍历与当前用户的所有连接，并利用连接上的sendChannel对象将选择的文件分块传输给远端。远端在接受文件时，会新建一个fileReader对象，并将接受到的文件快放入当前文件对象的receivedBuffer中，并利用一个offset变量存储当前文件块的大小。一旦offset变量与文件本身的大小相等，说明文件已传输完毕，此时将接受到的完整receivedBuffer对象转换成为Blob对象，然后利用URL.createObjectURL方法将Blob对象转化为可供下载的链接，添加在下载按钮的downloadUrl属性中。此时，文件传输结束，可以关闭dataChannel。点击下载按钮即可将文件下载到本地。

前端实现代码：

sendFileByRtc() {

const chunkSize = 16384;

let fileReader = new FileReader();

let offset = 0;

fileReader.addEventListener('error', *error* => console.error('Error reading file:', error));

fileReader.addEventListener('abort', *event* => console.log('File reading aborted:', event));

fileReader.addEventListener('load', *e* => {

console.log('FileRead.onload ', e);

this.pcList.forEach(*item* => {

console.log(item)

item.sendChannel.send(e.target.result)

offset += e.target.result.byteLength;

// sendProgress.value = offset;

if (offset < this.selectFile.size) {

readSlice(offset);

}

})

});

const readSlice = *o* => {

console.log('readSlice ', o);

console.log(this.selectFile)

const slice = this.selectFile.slice(offset, o + chunkSize);

fileReader.readAsArrayBuffer(slice);

};

readSlice(0);

},

receiveChannel.onmessage = *event* => {

// 查找出此连接下的file并更新

let currentFile = this.findCurrentFile(item.number)

currentFile.receiveBuffer.push(event.data)

currentFile.receivedSize += event.data.byteLength

// 文件传输完成

if (currentFile.receivedSize === currentFile.size) {

const received = new Blob(currentFile.receiveBuffer);

currentFile.downloadUrl = URL.createObjectURL(received);

receiveChannel.close()

}

}

// receiveChannel.onopen = onReceiveChannelStateChange;

// receiveChannel.onclose = onReceiveChannelStateChange;

})

由于远端在文件开始传输之前并不知道文件的信息，所以在开始利用dataChannel传输文件之前，需要通过信令服务器先把文件信息发送给远端。

后端文件发送信令代码：

// 发送文件

if (data.type === 'send\_file') {

console.log(data)

io.to(data.room\_id).emit('message', {

type: 'send\_file\_res',

from: data.from,

room\_id: data.room\_id,

number: data.number,

time: data.time,

file\_size: data.file\_size,

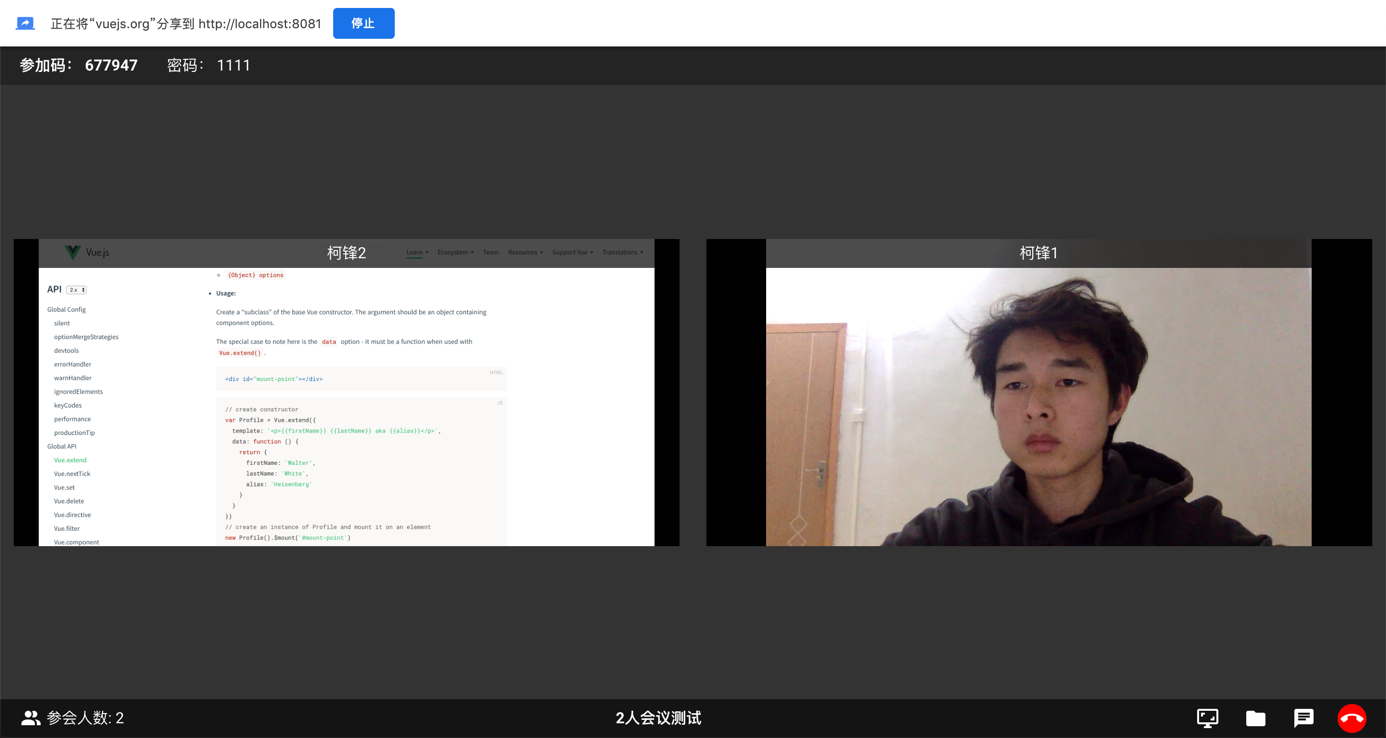
file\_name: data.file\_name

})

}

### 3.9 屏幕分享

实现效果：



3.9-1 屏幕分享效果

屏幕分享功能与普通的视频传输逻辑一样，唯一不同的是获取媒体源的方式不同。

普通视频获取通过以下代码实现：

navigator.mediaDevices

.getUserMedia(this.mediaStreamConstraints)

// 获取本地媒体流

.then(this.gotLocalMediaStream)

.catch(this.handleLocalMediaStreamError)

屏幕分享获取媒体源通过以下方法实现：

navigator.mediaDevices

.getDisplayMedia(option)

.then(this.gotLocalMediaStream)

.catch(this.handleLocalMediaStreamError)

由于屏幕分享需要相应API的支持。因此此功能仅在Chrome和fireFox浏览器中可用。并且由于WebRTC的传输依赖底层的编码以及其他音视频数据处理的过程，无法做到在视频传输的时候切换为屏幕分享。将摄像头获取到的视频流切换为屏幕分享流，需要重新建立rtc连接。

第四章 打包上线

### 4.1 前端项目打包

利用npm工具的npm run build命令可以实现将使用vue构建的项目快速打包为静态文件。这会在根目录下生成一个dist文件夹，将此文件夹放到服务器中的对应目录下，再通过配置nginx静态资源服务器，即完成前端项目打包部署。

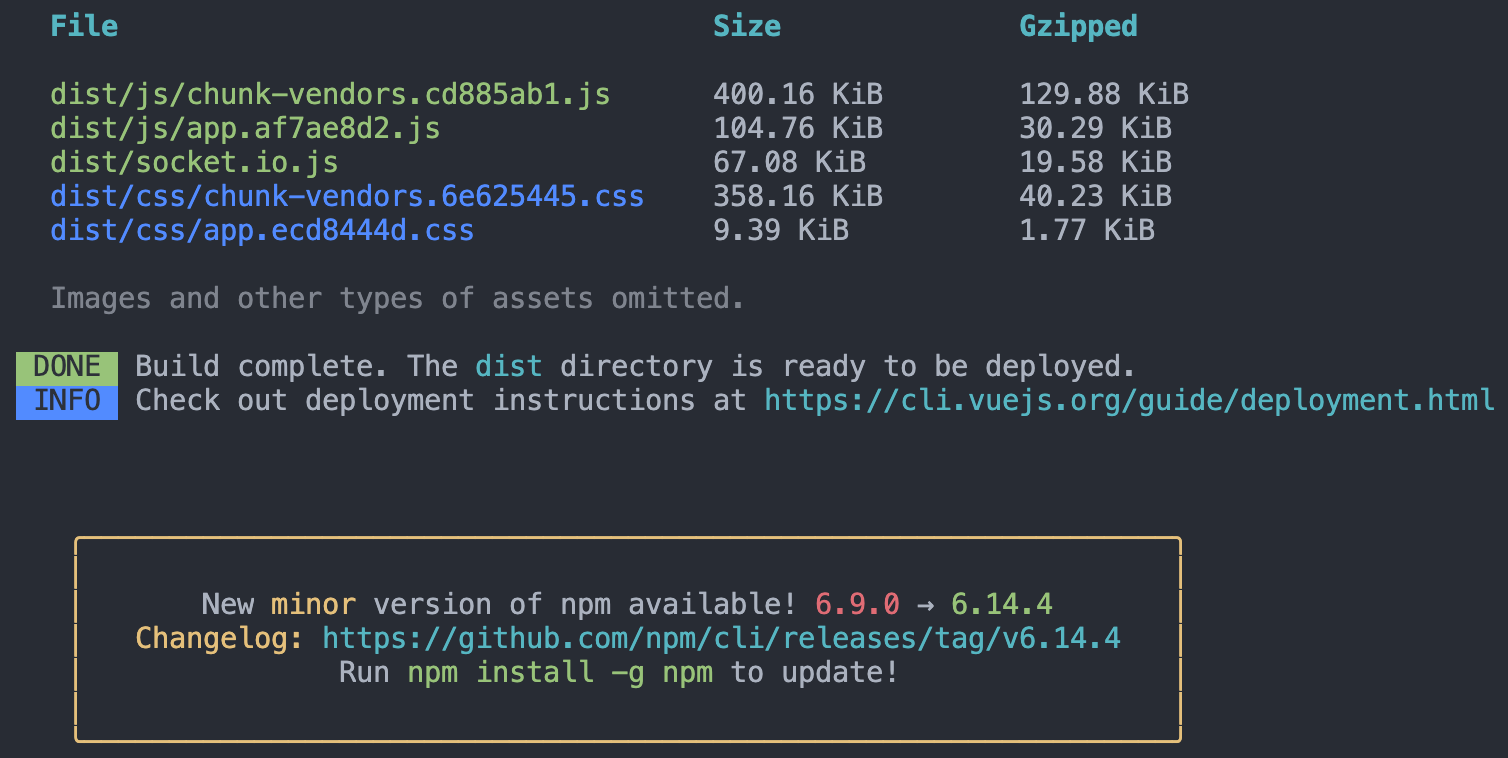


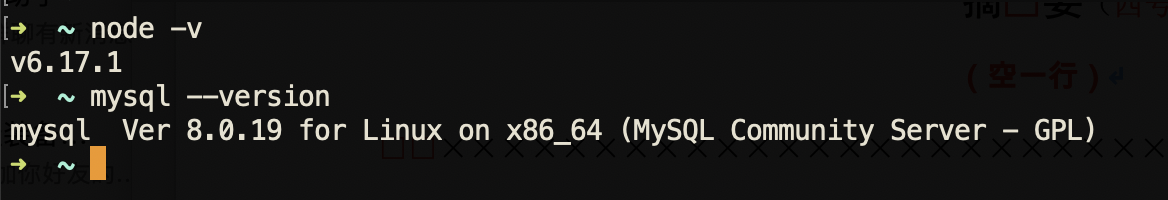
图4.1-1 前端文件打包

### 4.2 后端项目部署

后端项目依赖于node.js环境，在运行后端代码之前，先要确保服务器中安装有node.js环境和mysql应用程序。本项目部署在CentOS 7.0环境中，使用yum命令可快速安装node.js和mysql：

yum install nodejs

yum install mysql



4.2-1 服务器配置环境

将后端文件整个迁移到服务器中，后端代码无需打包压缩，直接放到服务器中。配置好数据库信息以及涉及到的目录信息，然后在后端项目根目录运行nodemon index.js即可启动后端服务，本项目中使用pm2工具来管理后端服务。

### 4.3 nginx静态资源服务器配置

前端项目转移到服务器之后，需要使用nginx来启动一个静态资源服务器，将域名信息等配置在nginx的配置文件nginx.conf中，然后在配置文件中指出静态资源的访问目录，再通过：

nginx -s start 命令，即可启动nginx服务，此时，便可通过域名访问网站了。

第五章 总结与展望

互联网技术发展至今，计算机硬件配置和网络条件不断提高，极大的刺激了软件开发进程。WebRTC技术正是在计算机软硬件技术极速发展的时候横空出世，并不断优化至今。如今各大视频技术服务商都有基于WebRTC技术的音视频服务，并且是收费服务，足以看出WebRTC技术带来的优势：更短的网络延时、更高的画质、更低的维护成本。基于浏览器的通信不仅不需要用户下载额外的软件，也不需要用户主动更新软件。除此之外，由于web应用的跨平台特点，用户可以在不同的终端设备使用它，极大的提升了用户体验。但不可否认的是，Web通信也有一些存在至今的技术以及安全层面的问题。

### 6.1 总结

本文通过对WebRTC技术以及目前主流的Web应用框架的研究，实现了一套基于浏览器的多路实时音视频会议系统。不需要下载安装客户端和各种插件，就可以实现多路音视频实时互传、文件互传、实时聊天、屏幕分享等功能。

1、前端框架使用了目前主流的三大框架之一：Vue，它极易上手开发，拥有Vuex,VueRouter,VueCli等官方工具，使得开发者可以专注于逻辑处理，而无需过多的注意视图变化，极大的降低了开发和维护成本。

2、后端基于Node.js环境，它与JavaScript开发语言几乎一样，并且可以利用express框架实现快速搭建http服务器，静态资源服务器，webSocket信令服务器等。

3、数据库使用了MySQL，它是目前最主流的数据库服务之一，在安全性、可移植性、性能等方面都有不错的表现。并且与MySQL配合使用的插件也非常多，可以非常方便的在后端与数据库之间建立连接。

4、本文对WebRTC以及WebRTC的主要接口进行了详细的介绍以及实际应用，利用RTCPeerConnection对象在端对端之间传输文件、音视频等信息，不需要流媒体服务器提供服务就可以将用户之间的视频流传输给对方。极大的降低了网络延时，节省了服务器成本。

5、利用Socket.io实现了信令服务器的搭建，给rtc连接的建立提供了必要的基础。通过webSocket传输offer、answer、chat、icecandidate、user\_join等信令，使得用户之间的连接能正常建立，实现了用户聊天，文件传输等。

但是由于本人能力有限，技术功底不足，以及投入到开发上的时间不够充分，该系统还有很多不完善的地方，以下几点是我认为有待改进的地方：

1、系统兼容性不够好，尤其是移动端，没有投入时间去做移动端适配工作；

2、系统稳定性有待提高，在本地测试条件下，没有针对一些特殊使用环境下做处理，例如断网处理，url限制等；

3、没有做用户设备检测，由于WebRTC技术的特殊性，对设备性能以及型号有一定要求，因此，当不满足使用条件的设备进入网站时，会出现不可预料的错误。

### 6.2 个人展望

随着5G、WI-FI6时代的到来，网络延时已经优化到了一个新的高度。基于Web的实时应用不断普及，例如各类小程序、快应用等，都是不需要下载安装客户端就能使用的web程序。相比于传统工具型应用来说，用户似乎更青睐于这类‘web应用’，因为它们不仅可以节省流量，还可以节省用户的时间，不需要漫长的等待过程，也不需要频繁更新，用户体验更好。

另外，随着计算机硬件的‘摩尔定律’,各种计算终端的配置水平提升飞速，类似WebRTC这类技术的软肋：更高的硬件配置，也已经不再是阻碍。现如今的任意一款主流配置的机器，都能流畅运行此类应用。

WebRTC作为一项开源技术，由全世界顶级的技术团队：谷歌公司进行维护和更新，并且被纳入W3C联盟标准，必将在未来将得到进一步的完善和发展。尽管在目前来看，它还不为更多的人所知，但鉴于此如此巨大的潜力以及高速发展的互联网软硬件技术，我相信一定会有越来越多的开发者们去了解它、使用它。同时也希望能有更多有能力的开发者利用WebRTC技术开发出质量更高的、功能更丰富的应用，加快互联网技术的发展。

参考文献

[1] Alan B.Johnston，Daniel C.Burnett，声网Agora.io（译）. WebRTC权威指南[M] . 北京：机械工业出版社 . 2016-08

[2] 何明亮 . WebRTC技术的研究与应用[D] . 南京：南京邮电大学 . 2014

[3] 霍丽娟 . 基于WebRTC技术的音视频聊天室系统的设计与实现[D] . 南京：南京邮电大学 . 2016

[4] 谷歌在线开发实验室 . Real time communication with WebRTC官方文档 . [EB/OL] . https://codelabs.developers.google.com/codelabs/WebRTC-web/#1

[5] Vuetify.js官方文档 . [EB/OL] . https://vuetifyjs.com/zh-Hans/

[6] 王 (Vanessa Wang) . 萨利姆 (Frank Salim)，HTML5 WebSocket权威指南[M] .机械工业出版社 . 2014-03

致 谢

首先要感谢我的论文指导老师肖蓉导师。肖老师是一位知识渊博、积极年轻的老师，对生活和工作充满热情。在我进行毕业设计和撰写论文期间一直给予了积极的支持，是她一直以来的督促和指导帮助我完成了毕业设计和论文的撰写。

其次我要感谢我的班主任肖老师。在本科的最后半年时间里，由于特殊情况无法到校参与学业学习，他耐心的通过线上指导我们就业和毕业，举行线上班会，为我们请来学业有成的学长指导我们学习和就业，非常负责。

然后要感谢我身边的同学朋友以及同事们。通过与他们的学习和合作，不仅我个人的学习能力、工作能力、技术水平得到了提升，还教会了我更多的为人处事的道理。他们每一个人都对我有着或多或少的影响，使我能够快速的融入团队，提升自我。

我还要感谢我目前实习的单位。在本科的最后阶段，一直积极的支持和配合我在实习和学业之间切换。尤其是我的直属上级湛敏斯导师，对我个人的技术能力以及团队合作能力产生了非常大的影响，一直以来是他帮助我解决工作上的各种问题，给予我充分的时间完成学业。

最后要感谢stack overflow、github、csdn等技术交流论坛以及各类技术的官方机构。他们的网站内容对我完成毕业设计和论文有着非常多的帮助，感谢这些开源机构和开发者们。

感谢我的父母和家人们一直以来对我学业和工作的支持，让我一路走到现在，感谢你们的无私付出！

由衷的感谢以上各位老师、同学、同事、领导、家人朋友以及各类机构！