南京大学

研 究 生 毕 业 论 文

（申请工程硕士学位）

论文题目	面向企业管理平台的会话子系统前端的设计与实现
作者姓名	陈 硕
学科、专业名称	工程硕士
研究方向	软件工程
指导教师	苏丰 副教授

2018 年 4 月 1 日

学　　号：**MF1632008**

论文答辩日期：　　年　月　日

指 导 教 师：　　（签字）

面向企业管理平台的会话子系统前端的设计与实现

作　　者：　　陈硕

指导教师：　　苏丰　副教授

南京大学研究生毕业论文

（申请工程硕士学位）

南京大学软件学院

2018 年 **04** 月

The Design and Implementation of Front-end of communication subsystem for Enterprise Management Platforms

Chen, Shuo

Submitted in partial fulfillment of the requirements for the degree of Master of Engineering

Supervised by

Professor **Su, Feng**

Software Institute

NANJING UNIVERSITY

Nanjing, China

June, 2011

摘 要

随着互联网的普及和信息技术的不断发展，企业各种对外的业务活动也已经延伸到了互联网上，这也促使越来越多的企业管理者从新的角度思维来探索企业管理和经营的进步。互联网的出现不仅直接破除了企业经营中的地理边界和信息流通障碍，还间接为组织提供了更加细致、强力的管理手段。

而随着办公自动化和信息数字化，企业管理系统会涉及到一个企业的方方面面，规模也越来越复杂，同时企业内部沟通的需求也应运而生，但一些个人通讯软件或第三方企业即时通讯平台在安全性和可用性方面无法完全满足企业管理的需求，所以南京某公司决定在其企业协同管理平台中加入会话子系统，在系统内部提供企业即时通讯服务。企业内部沟通的目的往往是针对某些问题，抑或围绕某个主题进行深度讨论，在每次会话展开之前都可以确定此次沟通的主题和参与这一事务的相关人员。

本文所介绍的会话子系统是基于企业协同管理系统，在解决企业内部管理问题时，面向不同的事务、发布、组织结构或不同的工作主题提供基于工作上下文的群组会话聊天服务，同时又能与企业管理模块相兼容。本文主要研究的方面是会话系统前端的设计与实现，该系统整体采用前后端分离架构，后端总体采用微服务架构和 Spring 框架，对外提供数据接口，前端为基于 React 框架开发的单页面应用，在开发方面以前端工程化的思想规范整个开发流程，构建前端的编译生产发布流程。在 Web 即时通信方面，使用 HTML5 提供的新技术 WebSocket，实现浏览器与服务端的持久连接，提供稳定、高效的会话服务。

本论文详细介绍了作者在会话子系统的设计和实现中的相关工作，包括明确会话需求，进行需求分析，设计会话子系统前端各模块，以及各模块的编码实现，最后总结了系统的待改进之处以及进一步工作展望。

关键词：企业即时通讯、前后端分离、React 框架、HTML5、WebSocket

Abstract

With the development of the Internet and information technology, more and more managers to explore the progress of business management and management from a new perspective. The enterprise management system will involve all aspects of a company, and the internal communication needs of the company will come into being, but some personal communication software or third-party enterprise instant messaging platforms can not fully meet the needs of business management because of security and availability. One problem that needs to be solved in an instant communication within an enterprise is that the purpose of the communication is to solve a problem, or to conduct in-depth discussions around a topic, and to assign corresponding employees to the transaction each time such communication is initiated.

The communication subsystem introduced in this paper is to solve this problem. It provides a group conversation chat service based on work context for different issues, organizational structures, or work topics. The main research topic in this paper is the design and implementation of the front end of the conversation system. The system adopts a front-end and back-end separation architecture as a whole. The back end generally adopts the micro service framework and the Spring framework to provide external data interfaces. The front end is a single page application developed based on the React framework. In terms of Web instant messaging, the HTML5 new technology WebSocket is used to implement long-term client-server connections and provide stable and efficient session services.

Keywords：Internal communications,Frontend development, React, HTML5, WebSocket

目 录

图目录

表目录

第一章 引言

1.1 项目背景

随着互联网的普及和信息技术的不断发展，企业各种对外的业务活动也已经延伸到了互联网上，互联网经济快速发展，这也促使越来越多的企业管理者从新的角度思维来探索企业管理和经营的进步。在企业传统经济活动中，地理、组织、部门、角色、资源等边界的存在带来了不必要的管理成本，制约了经济活动的更有效地开展。此外，发挥“互联网+”的时代优势，保证企业资源的有效整合，为企业管理提供科学的指导，促使企业稳步持续进步[马兰红, 2015]。互联网的出现不仅直接破除了企业经营中的地理边界和信息流通障碍，还间接为组织提供了更加细致、强力的管理手段。

“超级账号”是“思目创意科技有限公司”旗下的一款产品，该系统采用“人”，“财”，“物”，“事务”这四个要素作为核心概念，以事务模型为中心，将企业经营中涉及任务分配、决策、人财物流动等方面的经济行为组织起来，实现一个灵活型的企业管理系统，捕获经济活动中的关键决策和各项（人财物）资源流动，帮助企业破除各种隐形和有形的边界，助力于企业的灵活性管理和互联网时代的组织结构改革。

而随着办公自动化和信息数字化，企业管理系统会涉及到一个企业的方方面面，规模也越来越复杂，同时企业内部沟通的需求也应运而生。企业内部沟通的目的往往是针对某些问题，抑或围绕某个主题进行深度讨论，在每次会话展开之前都可以确定此次沟通的主题和参与这一事务的相关人员。

本文所研究的会话子系统为“超级账号”企业协同管理平台在企业即时通讯方面提供基于上下文的聊天会话服务。在特定的企业组织结构下，基于工作上下文主题去构建会话服务，企业管理者和普通工作人员可以根据部门的组织结构或工作主题快速找到相关人员。在部门组织下进行沟通会话，会话人员可以实时掌握部门动态，而排除其他会话群组的干扰。在主题下进行沟通，会话人员可以针对具体的工作内容，了解实时进展，以及历史沟通成果，避免了使用个人通讯软件而产生的工作效率低下的问题。

1.2 国内外的研究现状

传统企业管理软件经过长时间的积累，已经有了相当的基础，产生了以 ERP 为代表的典型软件产品。ERP 软件本身的建模、架构、技术支撑都非常出色，ERP 的可定制性又使得它能够非常适用于企业的特殊情况，而且已占领大部分的企业管理市场。但它的缺点依然存在，互联网飞速发展的今天，新形势下的改革受制于传统商业和管理机制，各个企业更加内向，形成了“孤岛”，没有最大化发挥互联网的效应。在企业内部沟通方面，很少提供完备的即时通讯服务工具。

不少公司在即时通讯方面仍会使用电话、传真、E-mail 为主的工具，但传统通讯工具实时性较低，沟通方式和内容单一，已经很难满足人们在通讯方面的需求。随着移动互联网和智能终端的普及，人们习惯于通过手机中的即时通讯工具相互沟通交流。很多企业内部员工的日常沟通也会使用的功能丰富的个人聊天软件，如微信、QQ 等，但这些聊天软件大多是通过创建群或临时讨论组这种扁平化的方法构建企业通讯架构，缺少符合企业组织架构的会话通讯录，同时个人即时通讯工具在使用过程中会让员工进入到非工作状态的聊天中，严重影响了企业办公效率，同时也带来了许多安全问题。

目前市面上也有很多第三方企业即时通讯平台，如阿里旗下的钉钉，腾讯的企业微信、TIM，网易的易信等，它们为企业打造免费沟通和协同的多端平台，面向企业终端使用者提供沟通服务，沟通方式包括文本、图片、文件、语音以及视频等，它们的优点在于可以基于的企业组织结构提供稳定、使用的即时通讯服务和扁平化的沟通渠道。而使用第三方平台也意味着一定的不安全性，公司内部交流信息或共享文件可能暴露于外部，会话数据也由第三方平台存储，并且大部分主体为实时通讯功能，只提供简单的企业管理功能，对于企业内部已有系统难以兼容。而一些可部署私有云的企业即时通讯工具在引进之初，需要对其现有的内部信息系统进行二次开发以实现二者的兼容，且这部分软件大都版本更新迭代停滞，需求难以满足现代信息化办公的需要。

近些年来，随着外包、开源、社群等地理分布办公需求的普及，各类协同工作软件例如（Teambition）开始出现并得到广泛应用。这类软件的关注点都是“分布式协同”工作，适合中小型团队的协同管理，提供简单的即时通讯服务，即使包含了管理机制也非常简单，对整个企业管理并没有提供一个很好的解决方案。

而本文所介绍的会话系统是基于“超级账号”企业协同管理系统，以事务模型为中心，在解决企业内部管理问题时，面向不同的事务、发布、组织结构或不同的工作主题提供基于工作上下文的群组会话聊天服务，同时又能与企业管理模块相兼容。本文主要研究的方面是会话系统前端的设计与实现，在前端框架方面对比目前国内外比较流行的 UI 框架，选用 React 框架进行组件化。React 是 Facebook 开发的一个为数据提供渲染为 HTML 视图的开源框架。React 以声明式编写 UI，能够以组件化的方式快速搭建前端页面，它是根据数据渲染页面，并且在数据更新时它的渲染机制能够高效地更新渲染界面[Facebook, 2017]。从前端框架的角度来说，目前的前端框架基本上都基于两个关注点：模块化和组件化。模块化解决了分而治之的问题，组件化解决了代码复用的问题，前端框架主要在这个基础上，实现其各自不同的解决方案。

在 Web 实时通信方面，早期的浏览器和服务端之间的通讯大多基于 HTTP 请求，传统 B/S 架构的 Web 应用在数据更新还需要通过刷新浏览器来完成。Ajax 技术的出现使得浏览器可以在不刷新的情况下异步获取服务端的数据，本质上是通过 XMLHTTPRequest 和 JS 脚本在客户端发送 HTTP 请求。但这种通讯方式仍存在一定的局限性，在通信过程中，只能浏览器主动请求数据，服务端根据请求作出响应，返回数据，而服务端无法根据后台数据变化主动推送来通知浏览器端，在一些实时性要求较高的应用场景中，难以满足开发者的需求。

在 HTML5 出现之前，开发者尝试通过各种 Hacks 方式来实现 Web 的实时通信方案，有基于 Ajax 的短轮询，模拟长连接的 Comet 技术。HTML5 规范了下一代的 Web 标准，同时也为 Web 端即时通讯带来了新技术 WebSocket。WebSocket 是 HTML5 开始提供的一种浏览器与服务器间进行全双工通讯的网络技术。WebSocket 可以用于建立客户端和服务器端的长连接，减少了互联网通信开销，并提供了 Web 服务器和客户端之间高效，有状态的通信[Pimental, 2012]。WebSocket 与 HTTP 协议相比，其优点在于控制开销少、实时性强、支持二进制，能够更好地建立长连接，实现双向通信，同时与 HTTP 协议有着良好的兼容性。相比于之前提到的 Hacks 方式，WebSocket 实现了真正意义上的 Web 双向通信。

1.3 本文主要研究的工作

本文所研究的会话子系统，主要是为“超级账号”企业管理平台提供会话聊天服务。在“超级账号”企业管理平台中，针对不同的场景，需要开发相应的会话模块，以达到在不同的组织或活动中能实现随时随地的交流沟通。应用场景和会话入口的多样化，所以会话子系统的前端需要以组件化构建，整个项目使用前后端分离架构，这也意味着前端在数据展示、交互体验、逻辑处理需要做更多的工作。会话子系统是基于“超级账号”企业管理系统进行开发，主系统在前端方面也是基于 React 的单页面应用，已经积累了一定量的前端业务组件。为了能在原系统中灵活嵌入，并保证在不同场景和入口下的可复用性，会话子系统在前端部分需要开发灵活通用的 UI 组件。在交互体验方面，由于受限于浏览器性能，而会话相关的主要功能又需要较高的实时性和稳定性，用户需要实时收到消息或聊天通知，前端需要对一部分会话数据进行缓存，避免频繁调用接口。

此外，Web 实时通信是系统最主要面对的一个问题。Web 实时通信的实现方案随着 HTML5 技术的发展也变得越来越完善，会话系统在 Web 实时通信方面使用的是 HTML5 提供的新技术 WebSocket，通过在 Web 端封装基于 WebSocket 的 SDK，用于建立浏览器端与服务端的长连接，保证会话服务的稳定性。

结合上述用户的需求，会话子系统前端采用 React 框架进行组件化开发，采用单向数据流和数据决定视图的设计思想，提高了前端的开发效率。使用 Redux 管理前端应用的数据状态，以数据渲染视图，处理组件之间的交互。结合 Webpack 实现前端工程化和模块化，对开发项目进行编译构建，对静态资源进行合并打包，优化了 Web 前端的性能体验。在前端通信方面，会话系统使用 HTTP 和 WebSocket 协议与后端交换数据，实时性较低、数据量较大的数据通过 HTTP 接口获取，而会话聊天中更新频繁的聊天信息等数据使用 WebSocket 交互，同时也在前端封装 WebSocket SDK，与后端建立双向实时通信通道，封装会话通信接口，实现断线重连和心跳检测机制，保证会话系统的稳定性和可靠性。

1.4 本文的组织结构

本文的组织结构如下：

第一章 引言部分。介绍了项目的整体背景，以及会话子系统的选题意义，目前国内外企业内部即时通讯工具的现状，以及在 Web 实时通信方面的技术研究现状，概述了本文的主要研究工作。

第二章 技术综述。将项目所要涉及的技术和框架做了介绍，包括前后端分离架构、HTML5 相关技术、前端工程化所使用的工具、前端框架 React 等，以及这些技术和框架在项目中的具体应用。

第三章 会话子系统的分析与设计。介绍超级账号企业管理系统整体概述，明确并分析会话子系统的需求，并阐述会话系统前端的概要设计和架构设计，设计前端会话缓存数据结构，描述了会话系统中各前端模块的详细设计。

第四章 会话子系统的具体实现。在需求分析的基础上，重点阐述了会话系统的前端各模块的具体实现，包括会话通用组件的实现、事务会话模块的实现、互动白板模块的实现、视频会议模块的实现。

第五章 总结与展望。总结论文期间所做的工作以及不足之处，并且就该会话系统的未来扩展作了进一步展望。

第二章 技术综述

2.1 前后端分离

在传统的 Web 项目开发中，几乎所有程序员都会将浏览器作为前端与后端的边界。前端指在浏览器中为用户进行页面展示的部分，而后端是指运行在服务器，为前端页面提供展示数据、处理业务逻辑以及持久化数据存储的部分。随着不同终端的兴起以及 Web 技术的发展，富客户端化的单页面应用也开始流行起来，同时用户也越来越注重 Web 端应用的交互体验。早期前后端之间的协作点是 Web 页面，前端工程师编写好静态页面之后交由后端开发人员动态嵌套数据，后端同时也需要完成前后端代码合并、发布部署等环节，承担项目全部的业务逻辑开发。随着 Web 应用的规模越来越大，前端负责的交互处理也随之变多，项目整体开发流程却依赖后端环境，导致前端工程师无法进行独立的测试和开发，协作和联调成本也很高，而且由于前后端代码耦合度教大，业务系统的重构与维护变得较为困难。因此，为了降低前端开发对后端的依赖程度，提高前端开发的效率，更好地实现前后端专业化分工，前后端分离的需求越来越受到重视[温馨, 2017]。

前后端分离作为 Web 应用的一种架构模式，与传统的 Web 开发也有所不同。在前后端分离的开发模式中，前后端工程师根据需求规范设计数据接口，编写接口文档，后端根据文档开发数据接口，前端根据文档开发页面，做到前后端并行开发，两端只通过数据接口交互，这样后端工程师不需要接触前端代码，前端开发者也不需要搭建后台运行环境；在应用部署方面，前端可以与后端部署在不同的服务器，两端之间可通过 HTTP 进行通信。我们主要在交互形式、代码组织方式、开发模式、数据接口规范流程这几个方面对前后端分离的架构模式进行详细地介绍。

在交互形式上，以 SPA 单页面应用为例，前端页面在初始化之初通过 Ajax 请求获取数据，然后根据数据和 JS 脚本组装 DOM，并渲染页面，后端不再需要将数据动态嵌入页面中，前后端之间的交互方式只有 HTTP 接口，在某些应用中还可能会有基于 WebSocket 协议的交互方式。

在代码组织方式上，前后端代码不再耦合在一个项目中，而是分成各自独立的项目或代码库，前端可以按照工程化思想搭建完整的开发和发布环境，可以使用 NPM 管理整个前端项目的依赖，使用 Webpack 构建工具对前端代码进行打包合并，输出所要发布的资源。同时根据前后端定义的接口文档，前端可以自行搭建 Mock 测试接口，支持前端独立的测试和开发。

在开发模式上，前后端分离使得开发之间分工更加明确，前后端之间只需定义好接口文档，前端工程师专注于页面渲染和处理用户交互，做好前端性能优化，后端也只需专注于 Model 层的开发，提供基于数据模型的实用接口。

在数据接口规范流程上，数据接口规范的定义变得越来越重要。一套良好的接口规范可以提升工作效率，减少沟通障碍。在接口定义时通常都会采用 RESTful 风格的接口，最关键的是接口的数据格式和结构，对于展示数据来说，一般会使用 JSON 格式，而明确了具体的数据结构，后端就可以根据接口编写测试用例，前端也可以 Mock 接口数据，实现并行开发，对后期的联调测试有很大帮助作用[Fielding, 2017]。

随着移动互联网和 HTML5 技术的发展，Web 移动端应用也越来越来多。前后端分离不仅仅为 Web 应用的开发带来了方便，IOS 端和安卓端也可以使用相应的接口，后端不需要针对不同终端开发定制的接口。同时这样的开发方式也能更好的应对复杂多变的前端需求和需求变更，增强代码可维护性和可读性。

在“超级账号”项目和会话系统中，前端是基于 React 开发的单页面应用，后端分为基于 Node 的会话后端和基于 Spring 框架的业务系统后端，后端都只提供数据和接口，前后端之间通过 HTTP 和 WebSocket 通信。在开发过程中，前后端根据需求预先定义好接口文档，后端根据文档开发接口，前端根据文档定义 Mock 接口，添加测试数据，联调时只需切断接口调用地址，整个过程都是并行开发的。

2.2 HTML5

HTML5 是用于在 Web 上构建和呈现内容的标记语言。它是 HTML 标准的第五个和当前主要版本。HTML5 既是一个单一的规范，也是一套完整的技术，通过不断的更新技术，添加新特性，使 Web 标准在互联网应用飞速发展的同时，

能提供更好的技术支持，满足开发者的开发需求，同时也旨在帮助浏览器能提供更原生的应用服务[Anthes, 2012]。

会话系统中主要使用到的 HTML5 新技术：多媒体标签<video>、WebSocket 通信、WebRTC、Canvas 元素绘图、拖拽 API。

2.2.1 WebSocket

WebSocket 是 HTML5 开始提供的一种浏览器与服务器间进行全双工通讯的网络技术。WebSocket 可以用于建立客户端和服务器端的长连接，减少了互联网通信开销，并提供了 Web 服务器和客户端之间高效，有状态的通信。在实际使用过程中，通过 WebSocket 完成一次握手，浏览器和服务端之间就可以建立持久性的连接，实现双向通信[Wang, 2013]。

在 WebSocket 出现之前，很多开发者会使用轮询的方法去模拟 Web 推送技术的实现，有基于 Ajax 的短轮询长轮询，长连接的 Comet 技术。轮询的实现原理其实就是每隔一段时间就向服务器发送请求，这种方式带来的缺点也很明显，客户端会不断发送请求，浪费带宽，网络开销大。而 Comet 技术虽然可以实现双向通信，但仍需不断发送请求。在这种情况下，HTML5 定义了 WebSocket 协议，能更好的节省服务器资源和带宽。WebSocket 的优点在于控制开销少、实时性强、支持二进制，能够更好地建立长连接，实现双向通信，同时与 HTTP 协议有着良好的兼容性。相比于之前提到的 Hacks 方式，WebSocket 实现了真正意义上的 Web 双向通信[刘峰, 2016]。

目前大多数主流浏览器都支持 WebSocket 协议，包括 Google Chrome，Microsoft Edge，Internet Explorer，Firefox，Safari 和 Opera。WebSocket 还需要服务器上的 Web 应用程序来支持它。

WebSocket 在会话系统中的使用主要分以下几个部分：建立长连接、注册事件、监听事件、推送消息、心跳检测、重连机制。会话系统中只有对实时性要求较高的接口使用的是 WebSocket 连接，例如消息的发送和接收、群组聊天和消息推送功能等，在开发过程中使用了 Socket.io——一款基于 WebSocket 的 JavaScript 开源库，会在后面的章节中详细介绍使用概况。

2.2.2 WebRTC

WebRTC 全称 Web Real-Time Communication 网页实时通讯技术，为浏览器提供实时语音或视频对话功能。WebRTC 作为一个免费开源的项目，由 Google、Mozilla 和 Opera 等团队提供维护支持，封装底层实现，提供支持实时通信功能的简单 API。WebRTC 通过制定一套通用的通信协议，使得开发者可以在移动平台、PC 浏览器和物联网设备等多终端开发高质量的实时应用程序[Loreto S, 2012]。

现在 WebRTC 技术内置于大多数现代浏览器中，对于开发者来说，在开发实时通信应用时不需要安装其他插件，而在 Google 团队开源 WebRTC 项目之前，浏览器实时通讯技术大多由大企业掌握，普通开发者难以使用，而目前 WebRTC 已成为 HTML5 标准之一，Web 开发者可以使用原生 JavaScript API 来开发功能多样的实时多媒体应用[张向辉, 2015]。

目前，开源社区上也有很多基于 WebRTC 开发的开源项目，用于方便开发者搭建 Web 实时通信应用。比如多媒体处理框架 Kurento，它支持 WebRTC 协议栈。轻量级 WebRTC 通信平台 Licode，提供音视频流的纯转发服务。还有 WebRTC 通信网关 Janus，以及用于移动端开发的 React-Native-WebRTC 框架。

会话系统中主要是视频会议模块使用了 WebRTC 技术搭配 Licode 框架，业务后端单独使用 Licode 开发 Node 服务端，拆分出了独立的视频转发服务，同时前端使用 Licode 中的 Erizo 组件，创建会话房间，处理 Licode 视频流之间的通信。

2.2.3 Canvas

HTML5 新增的 Canvas 标签在网页实时绘图方面功能强大，往往被看做是代替 Flash 的技术[Geary, 2012]。Canvas 基于位图技术，Canvas 有一个矩形画布区域，所有的操作都是基于画布的，通过 JavaScript 可以修改其每一像素上的绘图数据，其中每一像素都由四位数据表示，对应当前点的 RGBA 值。Canvas 的画图数据存储结构为点阵，按画图顺序有一定的层级，在将绘图数据覆盖到屏幕数据之前，可使用代码修改画图数据而不影响显示输出。Canvas API

提供了多种绘制线条、形状、图片等的方法[王宝丹, 2017]。

Canvas 的兼容性良好，除了 IE 上的 8 及以下版本，所有现代主流浏览器都支持 Canvas API，不需要使用任何插件。

Canvas 在会话系统中的应用主要在互动白板模块，提供视频会议的参与者绘制图形、插入文字的互动交互。

2.3 前端工程化

Web 前端开发这几年发展非常迅速，由于现代浏览器的出现，以及性能的大大提升，前端可以开发出功能完善、具有复杂界面的 Web 应用。而前端开发的特性使得它在面对复杂的产品需求变更时，可以进行快速迭代。这就使得前端规模越来越大，在团队协作开发中，没有规范和模块化的 JS、CSS 代码也使得前端项目变得难以维护，而在代码编译和打包的环节也没有很好的工具可以使用，所以前端开发者们尝试从工程学的角度去解决这一系列的前端问题。任何一个通向工程化的道路上都不可避免的会走过一段工具化的道路[占东明, 2016]。Node.js 的出现为前端开发者带来了快速构建平台，其自带的 NPM 插件既是一个 Node 模块化管理工具，也是目前最大的 JS 第三方库管理平台，管理前端项目的整个依赖。

前端开发者也在开源社区贡献了诸多推进前端工程化的项目和工具，模块化方面有 CommonJS 和 AMD 规范，以及现在比较流行的 ES6 的 class 关键字和模块化，构建工具有基于任务流的 grunt 和 gulp 工具，也有基于配置化的 Webpack 去帮助前端开发者对静态资源的重新构建，测试框架有可在前端进行单元测试的 Mocha 框架，在组件化开发方面目前比较流行的有 React、Vue、Angular 框架。前端工程化旨在帮助前端工程师开发可维护、高质量的项目代码，规范开发流程，根据业务特点，将流程标准化，提升前端工程师的开发效率，专注于业务代码的开发[张志飞, 2016]。

前端工程化大致可以分为这几个部分：技术选型、构建发布、模块化开发、组件化开发。

在技术选型方面，优先考虑项目的整体需求和项目日程，结合现有团队的技

术基础和新框架的学习曲线，做出最合理的技术选型方案。过去，前端开发者使用最多的库是 jQuery，这是一个快速、简洁的 JavaScript 代码库，提供了一系列 HTML 操作和事件处理 API。jQuery 简单易上手，普通的页面使用 jQuery 搭配 CSS 和 HTML，能够实现快速开发。而随着 Web 应用越来越复杂，产品交互体验越来越高，以及移动互联网的飞速发展，SPA 单页应用越来越受到前端开发者的青睐，React、Vue、Angular 等框架随之推出，项目的技术选型变得丰富多样。在项目初期，公司根据团队掌握概况和项目自身需求特点，选择 React 作为项目前端主要开发框架，团队成员开发不同的 React 组件，并组装成页面[Gallaba, 2015]。

在构建发布方面，由于前后端分离所带来的前后端团队的横向切分，前端开发团队需要自己对前端所使用的包和依赖做统一管理，并且对前端相关的资源包括 HTML、CSS、JS 进行合理的构建、打包、压缩操作。在开发阶段，这些静态资源都是以项目和模块分目录进行管理，而上线到生产环境中时，为了兼容旧版本的浏览器以及尽可能地优化前端页面的性能，需要对这些资源进行编译打包压缩，比如原先使用 ECMAScript6 语言编写的 JS 代码需要转换成兼容低版本浏览器的 JS 代码，LESS 模块化语言需要转换成 CSS Style 标签插入 HTML 中，JS 文件需要合并压缩成单个文件以减少 HTTP 请求数和资源大小。这些构建过程都是使用 Webpack 配置实现的，具体实现会在下面的章节详细地阐述。

在模块化开发方面，当项目复杂度到达一定规模时，JS 代码越来越庞大，Web 页面趋向于应用程序化，需要将一些功能明确、职责单一的代码交给不同的团队进行管理，分工协作开发，这就是模块化的来源。模块就是实现特定功能并且相互之间独立的一组方法，在前端模块化中，模块可以指一组 UI 组件，包含 HTML、JS 和 CSS 代码，独立负责渲染页面上的某一部分，也可以是一组纯 JS 代码的工具类，提供一些功能强大的函数。然而，早期 JS 由于语言的特性不支持模块化，开发者们通过制定模块化规范 CommonJS 和 AMD 来实现模块化，现在 ES6 中引入了 Class 的概念，同时支持像 Java 语言一样 import 和 export，而对于 CSS 的模块化，也有很多预编译语言如 Less 和 Sass 可以使用[Fink, 2014]。

在组件化方面，如果将一个页面的代码逻辑都写在一个文件中，上千行的代

码对于后续的维护和新功能的添加都是很困难的。组件化开发就是将整体划分为功能完整的组件，单独开发和维护。前端的组件化开发，更多的将页面也划入组件中，组件负责控制某一部分的数据层、视图层和控制层，管理这部分的 HTML、CSS 和 JS 代码，并向外提供一些可使用的属性和函数。React 组件根据数据渲染页面，这也意味着在页面的相同部分不必重复开发代码，只需开发通用性良好的组件，传入不同的数据即可。这里的组件之间的引用的方式也是借助了模块化开发的思想，待引用组件类通过 ES6 的 export 暴露出去，外部组件通过 import 引入。会话系统就是基于 React 开发 UI 组件，同一个组件可以在不同的页面容器中使用，只要设计好相应的组件接口 API 即可。

2.3.1 npm

npm 全称 Node Package Manager，是 Node.js 默认的，它既是一个 Node 模块化管理工具，也是目前最大的 JS 第三方库管理平台。npm 可以管理本地项目的所需模块并自动维护依赖情况，同时开发者也可以在 npm 平台上发布自己的 Node 模块供其他人使用。

在创建前端项目时，开发者可以通过 npm init 命令在文件夹中添加 package.json 文件，之后就可以直接使用 npm install 命令自动安装和维护当前项目所需的所有模块。在 package.json 文件中，开发者可以指定每个依赖项的版本范围，这样既可以保证模块自动更新，又不会因为所需模块功能大幅变化导致项目出现问题。开发者也可以选择将模块固定在某个版本之上[Npm, 2018]。

在会话系统的开发中，前端项目就是一个 npm 项目，使用 package.json 管理项目中使用的依赖，包括开发中的构建工具以及项目中使用的前端代码库，这样团队成员可以随时随地 clone 项目，安装依赖，搭建开发环境。

2.3.2 ECMAScript 6

ECMAScript 6 简称 ES6，由 ECMA 组织在 2015 年正式发布，是 JavaScript 语言的一次重大更新。ES6 旨在不断扩展 JavaScript 语言，使其满足开发复杂的大型应用程序的需要，成为企业级开发语言。

ES6 其实就是 JavaScript 语言的一种新标准，提供了一些新特性和关键字，

包括 Promise 异步规范和 class 关键字。JavaScript 本身没有类的概念，ES6 引入了 class 和 extends 关键字，并实现了类的继承。

JavaScript 由于语言的局限性没有设计模块体系，在开发大型应用程序时，无法使用模块化开发对整个项目分而治之。ES6 提供的模块化方案主要是通过 class、extends、import 和 export 关键字实现[Kerchove, 2016]。

```
import React from 'react' // 引入外部类
// 声明自定义类
class CustomComponent extends React.Component {
  constructor(props) {
    super(props)
  }
}
// 暴露类
export default CustomComponent
```

图 2.1 ES6 模块化代码示例

ES6 模块化代码如上图所示，通过 import 引入外部类，使用 class 关键字声明类，通过 export 输出需要暴露给外部的类。ES6 实现的模块化方案，简单通用，大大方便了前端模块化的实现。同时 ECMAScript 仍在不断更新规范，加入新特性，完善 JavaScript 语言，不久的将来也会有 ES7、ES8 出现。

在实际项目开发中，虽然 ES6 已经兼容主流浏览器，但是为了保证前端页面在旧版浏览器中的可运行性，会使用 Babel 对开发者编写的 ES6 代码进行一次转译，变成兼容性较高的 JS 代码。ES6 引入的 Class 和 Module 概念，使得开发者可以用面向对象的思想去设计编写 JS 代码，为 JS 模块化提供了一个很好的实现方案[阮一峰, 2014]。

2.3.3 Babel

上文提到的 ES6 是指新版本的 JavaScript 语言规范，它功能强大，但也带来一个大问题，大多数浏览器只支持旧版本的 ES5 代码，而不支持 ES6 代码。这个时候就需要使用 Babel 来编译。

Babel 是一种 JavaScript 转码器，它将使用特定语法（例如 ES6 代码，JSX）的 JavaScript 代码转换为可以被旧浏览器识别兼容的代码。Babel 提供的 babel-loader 可以搭配 Webpack 使用，实现对项目中.js 文件资源的转码和编译

[Babel, 2018]。

在前端项目开发中，会根据使用的 JS 语言版本对 Babel 进行预设，Babel 可以灵活地将各个版本的代码转化成 ES5 代码，保证前端代码的兼容性。

2.3.4 Webpack

Webpack 是一个开源的前端打包工具，它的主要作用就是将你的开发代码转换成生产环境中的代码。Webpack 提供了前端开发缺乏的模块化开发方式，将各种静态资源视为模块，并从它生成优化过的代码。它有四个核心概念：入口、输出、loader、插件[Tobias, 2017]。loader 转换模块的源代码，比如 babel-loader 将 ES6 代码转换成 ES5 代码，css-loader 将 CSS 代码转换成 style 标签插入 HTML 中，file-loader 对项目中引用到的图片、字体等静态资源进行统一管理，同时 Webpack 也可以根据文件内容生成 Hash，做到对静态资源的版本控制[Clow, 2018]。Webpack 由开源团队维护，目前已经更新到了第 4 版本，一直在不断优化前端构建工具。

在实际项目开发中，团队使用了开发和部署两套 Webpack 配置文件。开发模式下的 Webpack 会配置一个前端项目服务器，通过监听本地目录文件的变动而进行实时编译反馈给前端页面，实现浏览器的热更新。而在部署模式下的 Webpack 会将一些公用或者第三方库代码统一编译打包合并，将整个项目的文件拆分成库代码和业务代码，同时对代码进行丑化和压缩，更符合生产环境的要求。

2.4 前端框架

2.4.1 React

React 是一个用于创建 Web 用户界面的开源 Javascript 库，由 Facebook 工程师开发并维护。它旨在解决开发复杂用户界面时所面临的挑战，这些界面随着时间而变化，这在前端开发中被视为一大难题。React 会在任何给定的时间点根据应用中的状态数据渲染页面。当数据发生变化时，React 会自动处理所有的 UI 更新，从而避免开发者操纵 DOM，在开发单页应用和负责的交互页面时尤为

方便和实用。

React 使用组件描述用户界面，可以将组件看作简单的函数，调用函数时传入参数，获取输出，也可以根据需要重用函数，使用较小的函数组合更大的函数。React 组件中输入对应的是它的 props 和 state，而组件输出则是对用户界面的描述，在 Web 中可以理解为 HTML。开发者可以在多个页面重复使用单个组件，并且组件可以包含其他组件[Archer, 2015]。

React 组件具有被动更新的性质，当组件（输入）的状态发生变化是，它所表示的用户界面（输出）也会发生变化。在浏览器中，如果我们想改变 HTML 视图，需要编写大量 DOM 操作代码，重新生成文档对象模型 DOM。而借助 React，开发者无需担心如何更新 DOM，只需要在需要时更新组件状态，React 会在自动更新 DOM。React 组件的输入包括 props 和 state，props 是外部组件调用时传入的参数，外部组件可以修改传参来触发子组件的更新，state 是组件自身维护的私有状态，更新时需要通过 setState 方法来触发组件的重新渲染。

由上文可知，React 状态更新会带来视图 DOM 的更新，而频繁的 DOM 操作会大大损耗浏览器性能，降低用户体验。React 在性能优化方面的解决方案是 Virtual DOM，在一般的网页中，HTML 是以一个以各种节点组成的树状结构，React 的渲染函数根据开发者所编写的 JSX 代码在渲染前可以拿到整个 HTML 的结构，在 JavaScript 内存中生成一个虚拟 DOM 树，当数据更新时，React 渲染机制会根据变更数据生成新的虚拟 DOM 树，然后通过团队改进的 diff 算法比较新旧 DOM 树之间的最小差异，基于这些差异对真实 DOM 进行最小化重绘。Virtual DOM 避免了不必要的 DOM 更新，同时还简化了应用的维护成本。

React 是 Facebook 内部的一个 JavaScript 类库中，可用于创建 Web 用户交互界面。它引入一种新的 DOM 处理方式，你只需要声名式的定义各个时间点的用户界面，React 就能在指定时间点，以最小的 DOM 修改来更新整个用户界面[Staff, 2016]。

在组件化开发方面，React 使用 JXS 编写 HTML，在组件类中可以使用 ES6 的 import 引入 CSS，面向外部只暴露出一个组件类，囊括了 UI 组件所需的三元素：简单的 HTML 标签、JS 操作 DOM 的能力、CSS 描述布局的功能。组件化开发的优势有可扩展，通过组件之间的合理组合搭配，可以构建出满足业务需求

的新组件[Gackenheimer, 2015]。可复用，组件内部高度内聚，只给外部提供功能，对外部的修改关闭，组件之间低耦合，组件与组件只需要监听事件、触发事件，子组件不依赖与父组件。

2.4.2 Redux

Redux 是一个可预测的 JavaScript 状态容器，用于解决 React 没有解决的问题。传统的 Web 应用通常采用 MVC 的模式以结构化方式构建，实现关注点分离，但这样由于数据的双向绑定，开发者就失去了对数据流的控制。Redux 通过在应用中引入中央数据存储来解决这些问题。

Redux 有三个主要部分：Actions，Reducers 和 Store。Actions 用于将数据从主程序发送至 Store。在用户交互，内部事件或 API 调用之后，需要向 Store 发送数据以修改当前应用状态。Reducers 作为 JavaScript 纯函数，用于根据 Action 修改 Store 中的数据。Store 是管理整个应用状态数据的中心对象，页面组件订阅 Store 中的数据，并随着数据的更新重新渲染页面。

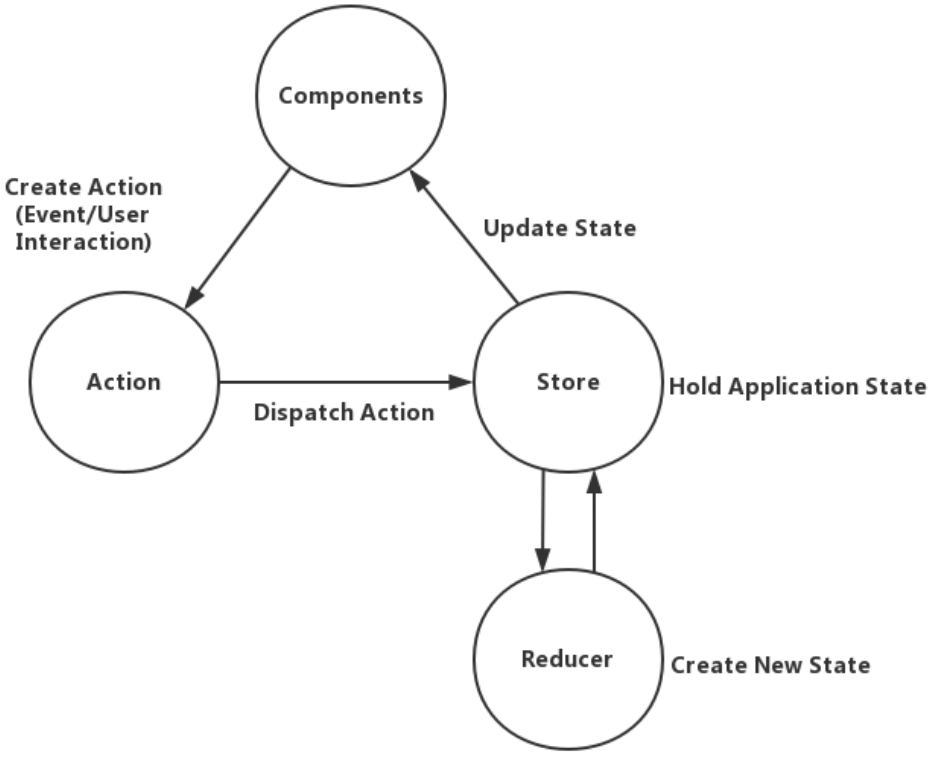

图 2.2 redux 状态更新数据流图

如上图所示，Components 负责渲染页面，在 MVC 中担任的是 View 的角色，在订阅 Store 中数据的同时，又创建 Action 去修改应用状态数据。在开发大型复杂多交互的应用时，React 需要搭配 Redux 使用，React 组件就扮演上图 2.1 中 Components 的角色。

Redux 在项目中有两个角色，一个是作为前端的数据源缓存，存储一些前端页面需要长期使用的数据，一个是作为 React 组件间通信的桥梁，组件订阅 Redux Store 中的数据并基于数据渲染 UI，并且组件也可以更新 Store 中的数据，更新数据会触发其他订阅组件的重新渲染操作，这样组件之间就建立了以 Redux 为基础的通信。

2.4.3 Ant Design

Ant Design 是基于 ReactJS 库构建的 UI 设计框架，用于开发 Web 和 React JS 应用程序，由阿里巴巴蚂蚁金服推出，专注服务于企业级产品。在公司项目中，我们主要使用了 Ant Design 提供的一套开箱即用的高质量 React 组件库，因为项目是针对企业管理系统，所以 Ant Design 的很多组件符合项目的开发场景和需求，其次也可以基于 Ant Design 的组件做一些定制化的开发，组装自己的前端页面。

2.4.4 Socket.io

Socket.io 是一个 JavaScript 库，用于在客户端和服务器之间建立双向通信。它是基于 WebSocket 实现，包含两部分，在浏览器上运行的客户端库和在 Node.js 上运行的服务器端库[Rai, 2013]。

在会话系统中，Socket.io 主要应用于两个部分。在浏览器端，实现会话通信所需的 Web SDK，封装 Socket 连接以及消息发送和监听的接口。在服务器端，利用 Socket.io 的 Node.js 库实现会话端口监听，管理会话房间，转发和推送会话消息。

2.4.5 Licode

Licode 是一个基于 WebRTC 的轻量级通讯平台，由开源社区维护，与 Google Chrome 的最新稳定版本兼容。尽管 WebRTC 的出现减少了搭建 Web 音视频通信应用的难度，但是开发者仍需要掌握简单的视频编解码、网络传输、媒体信息、浏览器、服务端技术。Licode 就是在 WebRTC 的基础上提供一整套搭建 Web 实时通信应用的开发环境，包括 Web 端 SDK、服务端房间管理框架、流转发服务端等，可以帮助开发者快速开发基于 HTML5 的视频会议。

在会话系统中的视频会议模块，由于前期团队开发的遗留问题，这部分的架构仍然使用的是 Licode 提供的房间后台逻辑和 Web 实时通信信道。

2.5 本章小结

本章先介绍了“超级账号”项目整体架构中所使用到的一些相关技术和设计思想，其次介绍了会话系统中所用到的核心技术，包括前端工程化和主要用到前端框架，详细介绍了基于 React 的前端组件化开发和会话系统核心通信方面所要用的相关技术。

第三章 会话子系统前端的分析与设计

3.1 会话子系统总体规划

会话子系统主体分为三个部分：会话子系统前端部分、Node 会话服务后端、与主系统对接的会话业务后端。本论文主要介绍会话子系统前端部分的设计和实现。会话子系统前端部分主要有事务会话模块、发布会话模块、视频会议模块、互动白板模块。其中事务和发布只是作为会话的不同入口，在上下文环境上有所差别，在会话的通用需求上有一部分是重合的，比如聊天框组件的开发、消息列表显示、消息的发送监听、聊天历史消息的管理等。所以先基于会话通用需求设计会话通用组件，而上层入口模块根据会话通用组件再搭建页面，上下文入口约束了参与会话的用户人群和会话依赖的相关实体，通用组件提供会话的基础服务，视频会议所包含的功能主要有邀请会话讨论组中的成员加入视频会议，通过摄像头设备实现视频通话，以及视频源的相互切换，和保存视频历史数据。互动白板模块主要是在视频会议的同时提供一个供会议成员互动交流的板块，视频会议的成员不仅可以通过音视频交流，还可以通过白板画图交流，由于互动白板大部分的功能都是与 Web 交互相关，所以白板模块基本上都由前端实现，后端只提供历史数据存储服务。

3.2 会话子系统需求分析

会话系统是“超级账号”主系统的重要组成部分，是在各个模块中提供基于上下文的聊天群组功能，如基于发布的讨论组功能。会话系统的入口处在主系统的事务模块和发布模块，会话系统中的会话都是基于事务和发布的，在企业管理模型中可能对应的就是公司、部门和活动实体，会话页面根据用户所处在的事务或发布进行随时切换，进入到某一事务或发布下的会话时，用户只能看到与上下文环境相关的会话讨论组和历史数据。会话基础需求会在下面的具体章节进行详细介绍，用户除了能在会话系统中就特定内容主题进行交流和决策，还要能在上述过程中基于会话成员快速地发起业务相关的沟通，如基于公司组织部门之间的

资金往来交易、或者物资登记交易等。基础会话功能包括单聊功能、群聊功能、聊天记录管理。视频会议和互动白板在功能上属于会话的辅助功能，在实现上视频会议和互动白板只利用会话提供的发起入口，模块相对独立。视频会议模块提供多人视频语音聊天功能、禁言与主动闭麦（或视频）、视频会议记录功能。参与视频会议的成员都可以进入互动白板，基于本次视频会议的主题对白板内容进行编辑，并保存历史记录。会话系统的整体用例如下图 3.1 所示：

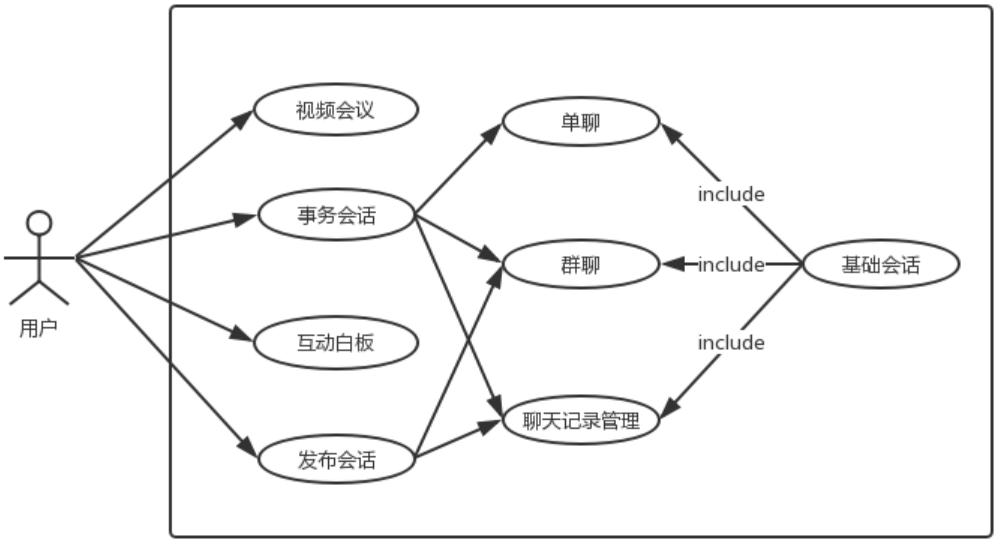

图 3.1 会话系统整体用例图

用户在“超级账号”企业管理平台中通过事务或发布入口进入会话中，使用会话基础功能进行基于上下文的交流和沟通，同时也可以通过视频会议入口进入视频会议模块和互动白板模块。

3.2.1 会话基础需求分析

会话基础需求包括单聊功能、群聊功能、聊天记录管理三个部分。会话基础需求用例如下图 3.2 所示：

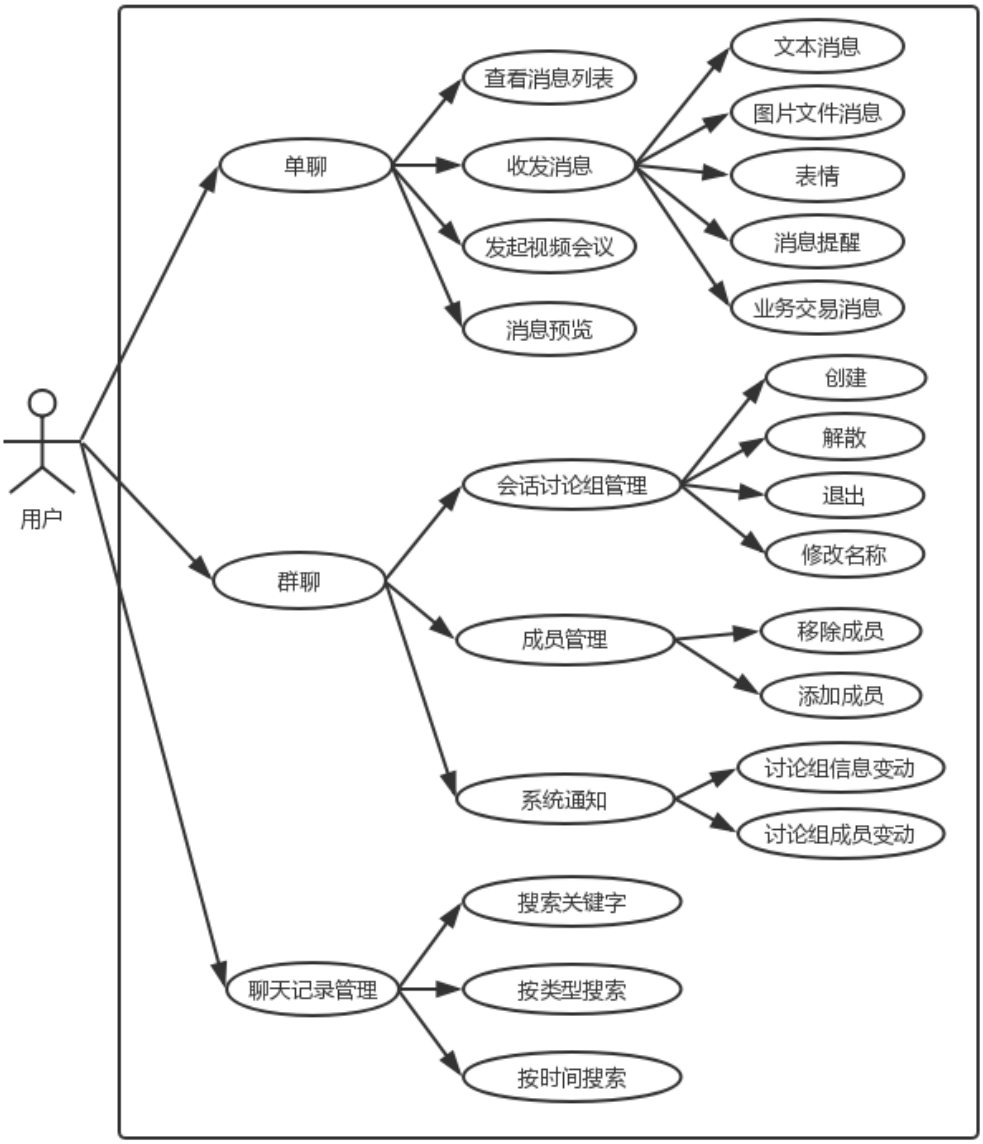

图 3.2 会话基础需求用例图

单聊主要功能介绍：

1. 查看消息列表：打开某一次单聊会话之后，可以看到一定数量的历史消息，通过向上滚动可以看到更多历史消息，实时收到的消息和发送出去的消息也会在消息列表中显示。
2. 收发消息：发送和接收消息，具体的消息类型有：文本消息、表情、图片消息、文件消息、含有消息提醒的消息、业务交易相关消息，根据不同的消息类型，构造或解析相同的消息体数据结构，渲染至页面。
3. 发起视频会议：用户可以在会话中发起特定主题的视频会议，邀请当前

会话中的成员，被邀请者可以在当前会话中点击加入会议。

4. 消息预览：可以对不同类型的消息进行预览，图片消息可以查看大图，文件消息根据文件类型提供合理的预览方案，业务交易相关消息提供查看业务详情的功能。

群聊的基础会话功能与单聊一致，不同的是会提供讨论组相关功能：

1. 会话讨论组管理：具有权限的角色可以创建、解散、退出、修改讨论组。
2. 成员管理：在讨论组中可以移除或添加成员
3. 系统通知：讨论组信息和成员的变动会产生相应的系统通知

聊天记录管理功能提供的是针对某一特定的会话查找相关的聊天记录，用户可以限定消息的类型或时间区间，根据关键词搜索相关聊天记录。

3.2.2 事务内会话模块需求分析

事务会话模块主要是为用户提供基于事务上下文的会话服务，用户通过事务进入相应的会话，当用户切换不同事务时，需要展现不一样的会话页面，事务会话在会话功能方面与基础功能一致，不同的地方主要在最近会话和通讯录管理方面。最近会话是指最近活跃的单聊或群聊会话列表，根据最后一条消息的回复时间排序，用户可以随时点击切换打开会话，当接收到实时消息时，最近会话会根据最后活跃时间重新排序，用户还可以通过搜索关键词过滤当前会话列表。通讯录存储了当前事务中用户所加入的讨论组和所有角色，用户可以打开任一会话，并更新最近会话列表。事务会话中还提供从主系统的其他模块开启单聊会话的入口，例如用户查看事务中的某一角色时，可以切换到会话页面，开启与该角色的单聊会话，并保留会话上下文。事务会话模块需求如下图 3.3 所示：

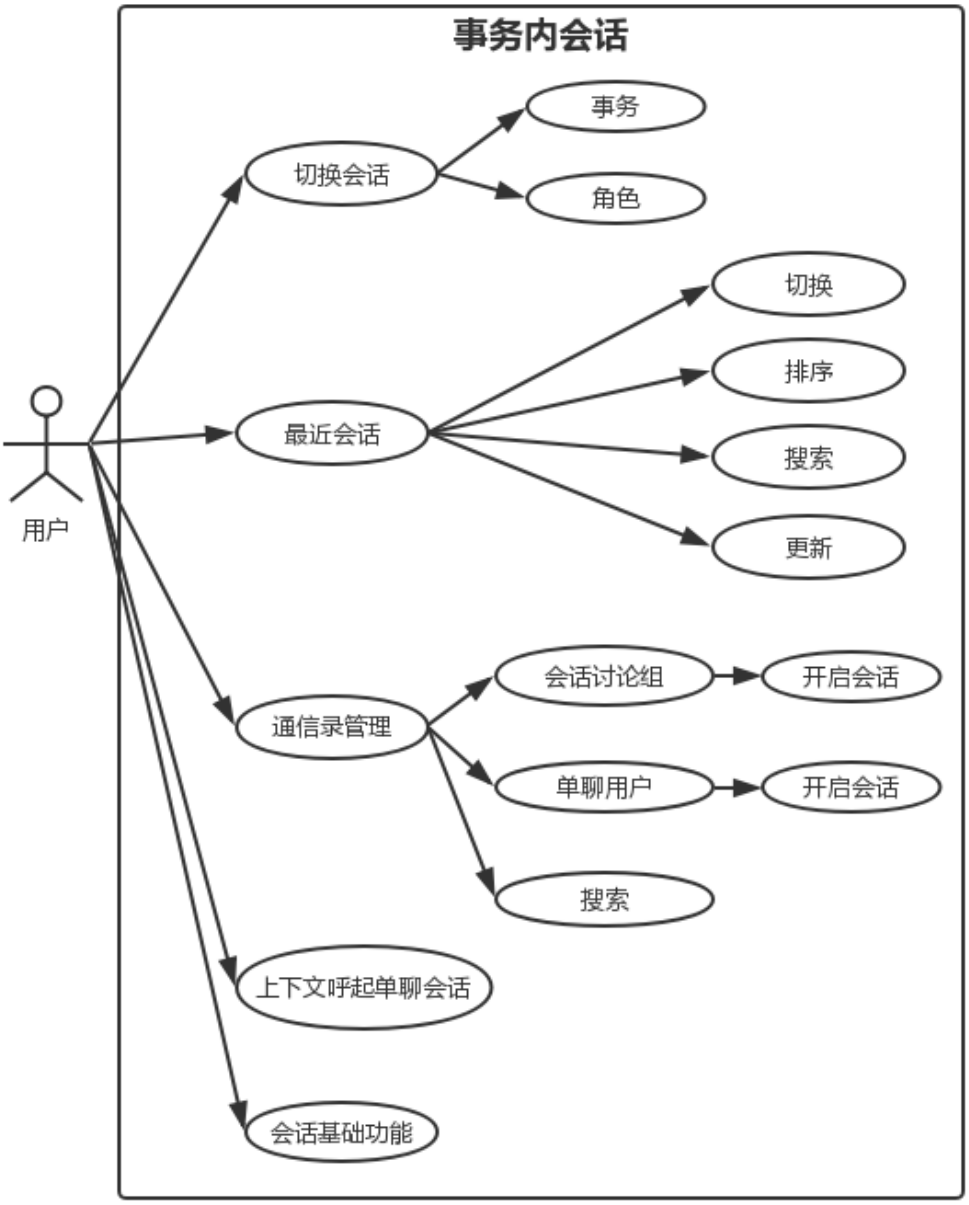

图 3.3 事务会话需求用例图

3.2.3 发布内会话模块需求分析

发布会话主要是为用户提供基于某一发布的上下文会话服务，同样也包括切换发布之后的会话切换，发布内会话会根据发布内成员自动创建相关的讨论组，发布内会话没有单聊只有讨论组聊天，其他会话需求跟会话基础需求一致。发布会话需求如下图 3.4 所示：

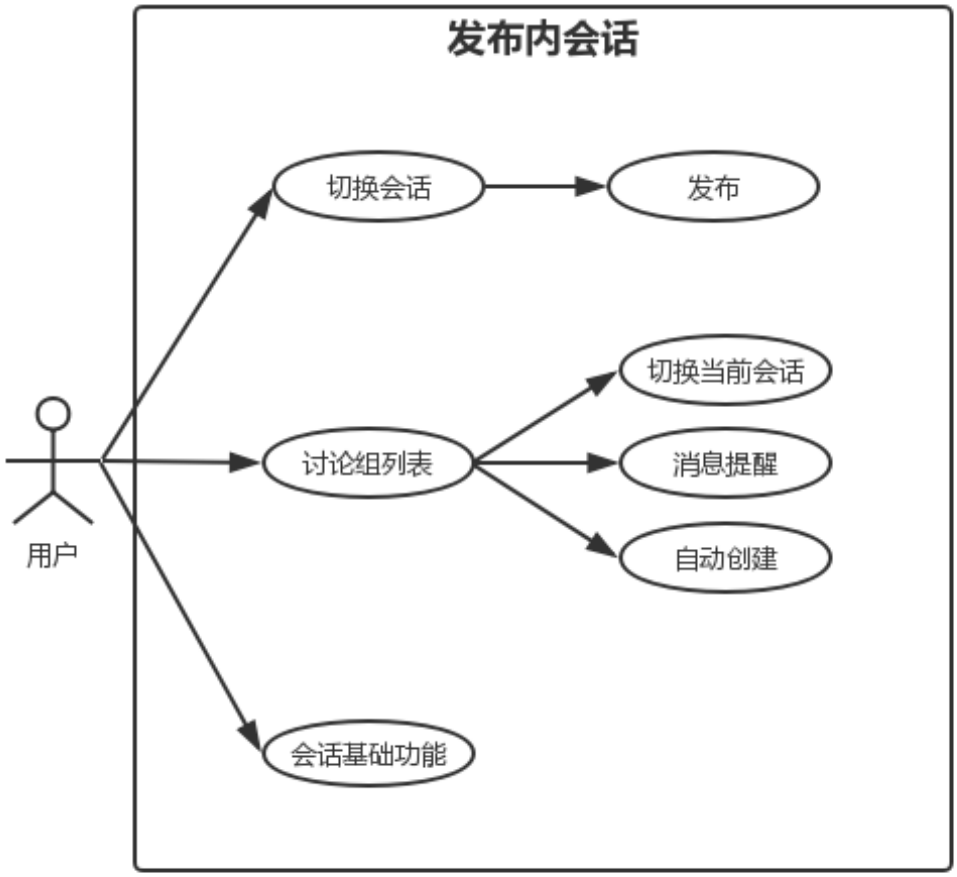

图 3.4 发布会话需求用例图

3.2.4 视频会议模块需求分析

视频会议模块主要是为用户提供一个多人视频语音的场景。用户在会话中可以创建视频会议，并邀请成员进入，视频会议同时只允许有一个用户使用屏幕播放视频，其他用户可以同申请屏幕改变当前发言人，视频会议的视频内容会保存下来，用户也可将视频会议切换成悬浮窗模式，在会议同时处理系统中的其他事务。视频会议模块具体需求用例如下图 3.5 所示：

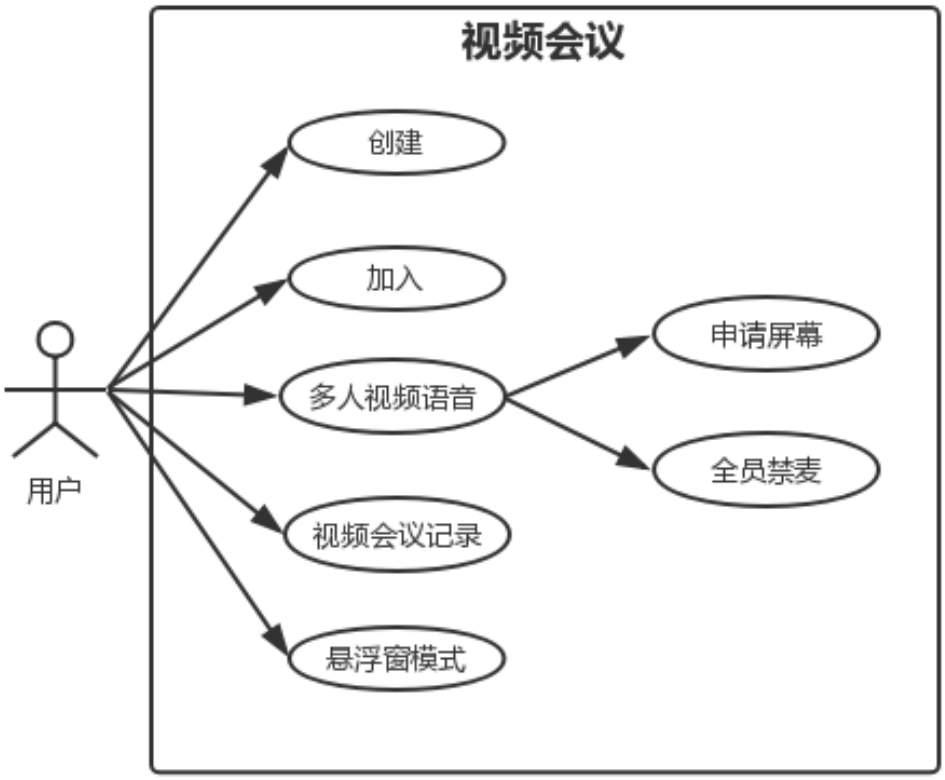

图 3.5 视频会议需求用例图

3.2.5 互动白板模块需求分析

互动白板模块是为用户在视频会议的同时提供辅助在线协同工具。用户可以在视频会议过程中使用白板，在初次进入白板时，需要展示之前用户的白板进行的编辑操作。视频会议的当前发言人可以给白板加锁，加锁后只有当前发言人可以对白板进行操作，所有人都可以查看白板，并且可以在白板和视频之间自由切换。

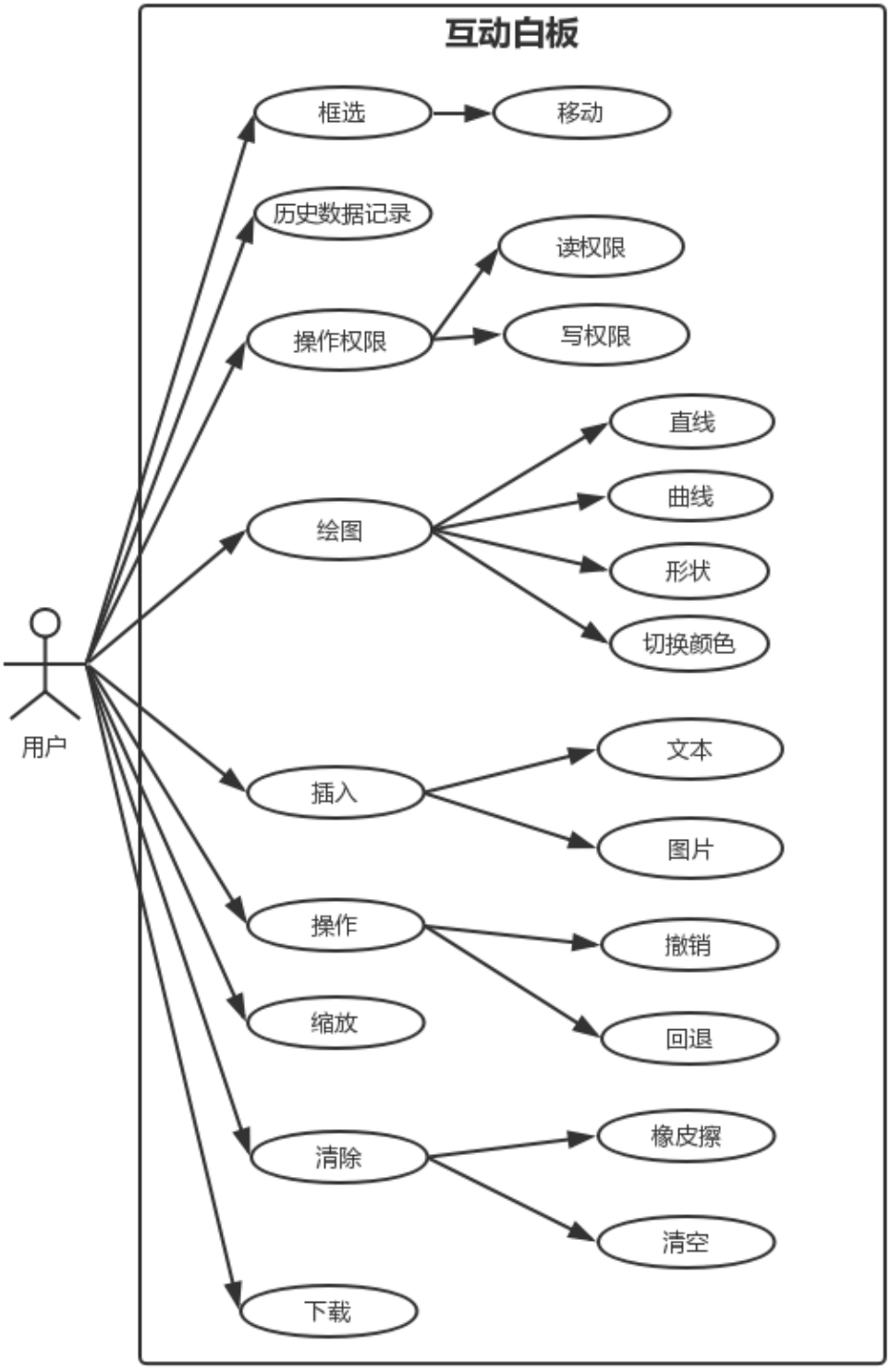

图 3.6 互动白板需求用例图

如上图 3.6 所示，互动白板的需求主要集中在用户交互部分，具体操作需求有：

1. 框选：在选择工具下，可以通过点击、框选组件，来对组件进行平移。
2. 画笔：在画笔工具下，可以绘制线条和选择粗细，分为直线、曲线。
3. 颜色：在颜色工具下，可以选择一个颜色，并应用到画笔绘制的线条颜色、文本的文字颜色、图形的边框颜色。
4. 文本：在文本工具下，用户可以通过拉动鼠标创建一个输入框输入文本
5. 清除：在工具下包含橡皮擦和清空两类工具，橡皮擦使用一个默认大小去擦除已绘内容；清空会清空白板所有内容，只有当前发言人可以使用。
6. 图形：在该工具下可以选择矩形、圆形两类图形进行插入，可使用鼠标拖放决定位置和大小。
7. 上传：用户可以在该工具下上传图片文件到白板中，可以支持 png、jpg、jpeg、svg 等格式。
8. 撤销：用户可以撤销自己进行过的操作，但不可以撤销其他人的操作。
9. 回退：用户可以撤销自己刚刚进行的撤销，如果上一次撤销之后用户进行了操作，那么就无法进行返回。

3.3 会话子系统前端概要设计

3.3.1 总体结构

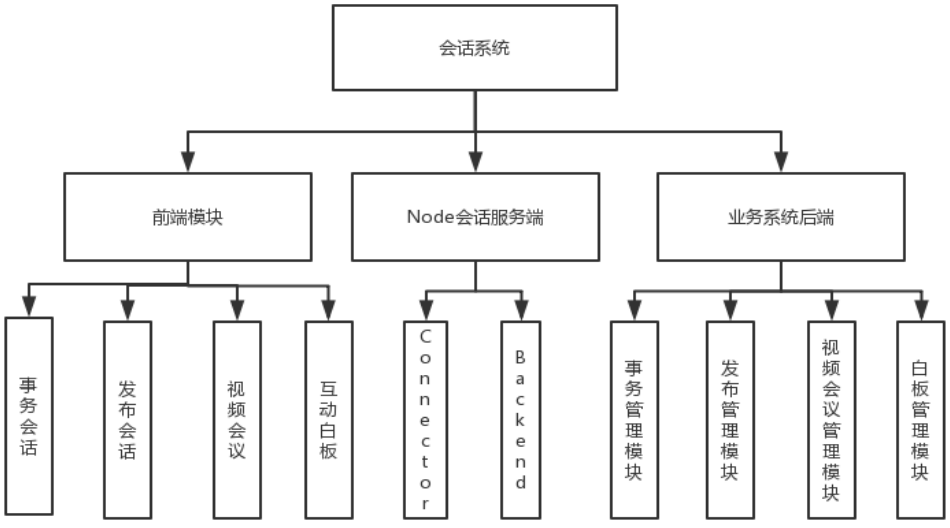

图 3.7 会话系统总体结构

如图 3.7 所示，会话系统主要由前端模块、Node 会话服务端、业务系统后端组成。前端模块根据功能拆分成事务会话、发布会话、视频会议、互动白板四个部分。Node 会话服务端分成 Connector 和 Backend，Connector 负责与前端保持长连接，Backend 负责处理会话逻辑。业务系统后端处理与主系统事务和发布相关的逻辑处理，前端与服务端之间使用 HTTP 或者 Socket 通信。

3.3.2 前端架构设计

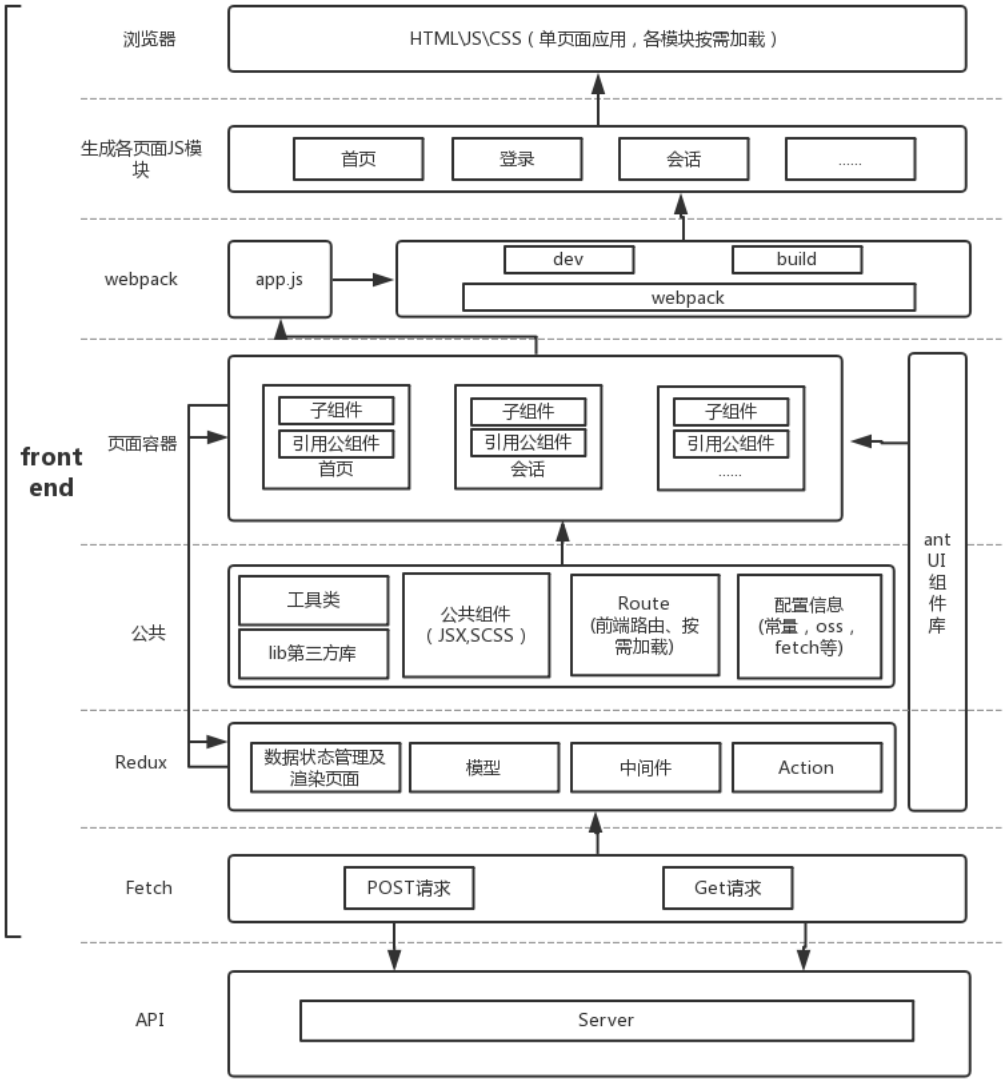

图 3.8 会话系统前端架构设计图

如上图 3.8 前端架构设计图所示，会话系统的前端是基于 React 组件构建的单页面应用，前端路由方面使用的是 React 生态圈中路由组件 react-router，将

路由定位到不同的页面容器。页面容器层负责使用公共组件或子组件组装页面，同时也会引用 Ant Design 基础组件。公共层负责前端项目中的工具类以及第三方类库使用，以及配置前端路由属性，封装 fetch 接口请求组件，配置后台 API 参数。Redux 层负责单页面应用的数据状态管理，控制前端页面状态数据的缓存和更新，同时管理页面的重新渲染。

Fetch 是一个现代的概念，等同于 XMLHttpRequest，它的核心在于对 HTTP 接口的抽象，但被设计成更具可扩展性和高效性。前端架构中的 Fetch 层主要封装整个应用对后台接口的调用，管理后台 API 接口，提供灵活的请求 header 参数配置，对频繁请求进行前端过滤，根据后端定义的响应码进行统一的错误处理。Server 层为后端服务层，提供数据服务，与前端通过 HTTP 请求和 Socket 通信。

Webpack 层负责对项目中的 JS、CSS 等资源进行打编译打包合并，可根据前端路由和页面模块配置，生成不同路由页面所需的静态 JS 和 CSS 资源包，方便页面的按需加载。同时 Webpack 也负责开发环境和生产环境的转换，根据参数可以灵活配置，将开发代码编译构建成生产环境下部署的资源。

Webpack 输出的生产环境下的 HTML 已配置好相应的 JS 和 CSS 资源标签。用户在浏览器端方法 HTML 单文件，加载打包合并过的 app.js 文件，执行 JS 代码，生成页面模块，根据后台接口获取数据，渲染可显示的页面。

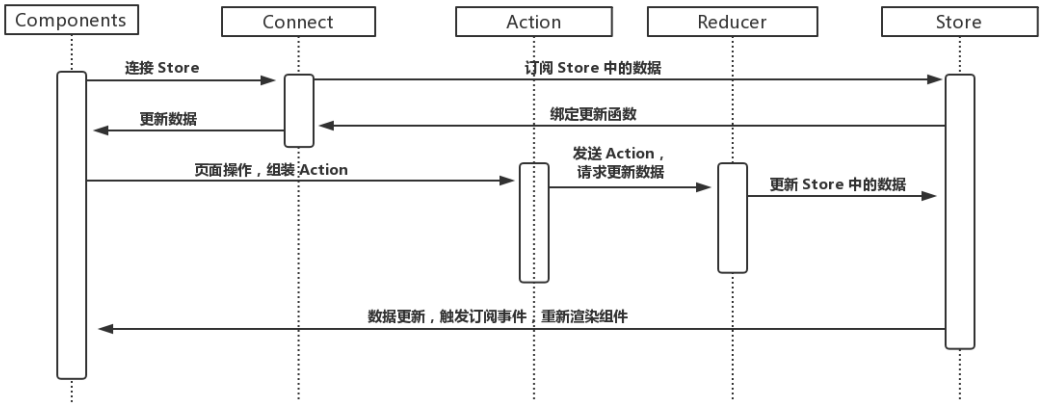

图 3.9 Redux 数据更新顺序图

如上图 3.9 所示为 Redux 的数据更新顺序图，Components 为 React 组件，负责页面展示和渲染，以及处理用户交互逻辑，Store 为 Redux 中存储前端数据状态的模块，Connect 为 react-redux 库中提供的方法，可以让 Component 订

阅 Store 中某些指定的数据，同时这些数据的更新也会触发 Compoent 的重新渲染。所以在 React 和 Redux 应用中，数据更新需要重新渲染页面时，只要去更新相应的 Store 中的数据就可触发页面的重绘。Component 在订阅 Store 数据的同时也是数据的修改者，用户的交互需要去修改页面时，Component 会调用 Action 中的方法，组装成一定结构的数据发送给 Reducer，Reducer 就是一个 Action 过滤器，收集当前应用的所有数据修改信息 Action，根据 Action 的参数决定修改哪一部分 Store 数据，Store 中数据的变更又会反馈给 Component 组件，更新页面，这就是单向数据流的设计原则。

3.3.3 会话消息体数据结构设计

会话系统中传输量最大最频繁的就是会话消息，消息体数据结构需要承载不同类型的消息，包括文本、表情、图片、文件、业务交易消息、系统通知、视频邀请。消息体数据结构的初步设计成两个部分：公共字段、特殊字段。公共字段存放如 id、time 这类通用的数据，特殊字段根据消息类型不同进行区分。具体的数据结构设计如下表所示：

表 3.1 会话消息体数据结构表

字段	使用
id	消息的唯一标识
time	消息发送时间，timestamp
key	消息所在会话（单聊或群聊）的唯一标识
sub	消息类型字段
type	区分消息是单聊还是群聊的字段
name	发送者昵称
fromUserId	发送者 UserId
groupId	若是群聊，则表示讨论组 id
apns	数组，消息提醒@的对象的 id 数组
content	json 格式，不同类型消息的内容字段

消息体中的 content 字段表示的是消息的具体内容，前端在发送 content 时需要将它转换成 json 字符串，后端不需要了解 content 的格式，只需要将它作为 string 变量存储。

表 3.2 会话消息体 content 数据结构表

消息类型	content 格式	说明
文本	content: string	含有表情的文本消息需要用 emoji 进行转换
图片	content:{ name, url, size, ext }	图片文件通过上传阿里云获取资源 url，再存入消息体中
文件	content:{ name, url, size, ext }	与图片相同
业务交易消息	content:{ orderId, remark}	记录交易 id
视频邀请	content:{ conferenceId, topic }	记录会议 id,会议主题
系统通知	content:{senderId, receiverId }	业务交易消息处理反馈
	content:{groupId, name, avatar }	讨论组信息变动（修改名称、创建、加入、退出、解散）
	content:{inviterName, inviteeName, removerName,removeeName }	讨论组成员变动（添加、移除）

content 中的数据结构设计如上表 3.2 所示，处理含有 emoji 表情的文本消息时，前端会有一个 emoji 表情 unicode 与 shortname 映射表，content 中存储的是某一个 emoji 表情的 shortname，当需要渲染至页面上时，将 shortname 转换成对应表情的 unicode，便于系统识别。

除系统通知类型的消息外，其他类型的消息只需要直接存入对应的消息列表中渲染即可。对于系统通知类型的消息，前端需要对不同类型的系统通知做出不同的交互反馈。业务交易处理反馈消息中需要标明当前交易的状态，以及交易人的相关 id，同时去更新之前交易信息的状态。讨论组信息变动消息需要记录变动的讨论组 id，前端收到消息之后，需要去更新相应的讨论组的数据或者请求新的讨论组数据。讨论组成员变动信息会调用接口去更新成员列表。

3.3.4 会话数据前端缓存设计

会话系统用户交互复杂，数据量大，频繁的 HTTP 的请求会导致前端性能较低，影响用户体验，在会话模块设计之初，就决定会话页面的初始化数据通过调用 HTTP 接口，而之后的数据传输通过 WebSocket 建立的长连接获取，在保证数据足够页面响应反馈的同时，减少网络开销，提高用户体验。React 组件通过

订阅全局 Store 中的数据渲染页面，所以通过设计 Store 中的会话数据源结构，缓存部分会话数据，同时 Store 中的数据又可作为会话组件之前通信的桥梁，或者是会话组件与主系统之间通信的桥梁。下面以事务内会话的数据缓存设计描述前端缓存数据的关注点。下表 3.3 为事务会话 Store 数据设计表：

表 3.3 事务会话 Store 数据设计表

变量	使用
recentChats	最近会话列表
isFetchingRecentChats	是否正在请求最近会话数据
chatGroups	事务内群聊讨论组列表
affairRoles	事务内单聊会话角色
selectedChatKey	正在会话的讨论组 key 标识
chatRole	外部环境发起的单聊会话角色

数据设计的依据是界面设计稿和交互需求，Store 中的数据部分是通过 HTTP 接口调用获取，部分是前端为了页面组件之间的通信而进行标记的。

表 3.4 RecentChat 数据结构设计表

变量	使用
key	会话（包括群聊和单聊）的唯一标识
type	区别会话是群聊还是单聊的标识
lastMsg	当前会话的最后一条消息
msgList	会话消息列表
opened	会话是否已经被开启过
unread	当前会话的未读消息数
chatInfo	会话的信息： 在单聊中，此字段会记录聊天角色的昵称、头像、id 在群聊中，此字段会记录讨论组的名称、头像、id

RecentChat 中的初始数据是通过 HTTP 接口获取，包括 key、type、lastMsg、unread、chatInfo 字段。当用户第一次开启某一会话聊天时，会通过 Socket 接口调用会话服务的数据，获取到对应的 msgList、最新的未读消息数，并将 opened 设置为 true，这样用户在切换最近会话时就不会频繁调用接口。当 Socket 消息监听收到一条会话消息时，只需要去更新 RecentChat 中的 lastMsg、unread、msgList 字段，然后对最近会话列表重新排序就可实现会话的实时更新。

3.4 会话前端模块详细设计

3.4.1 会话通用组件详细设计

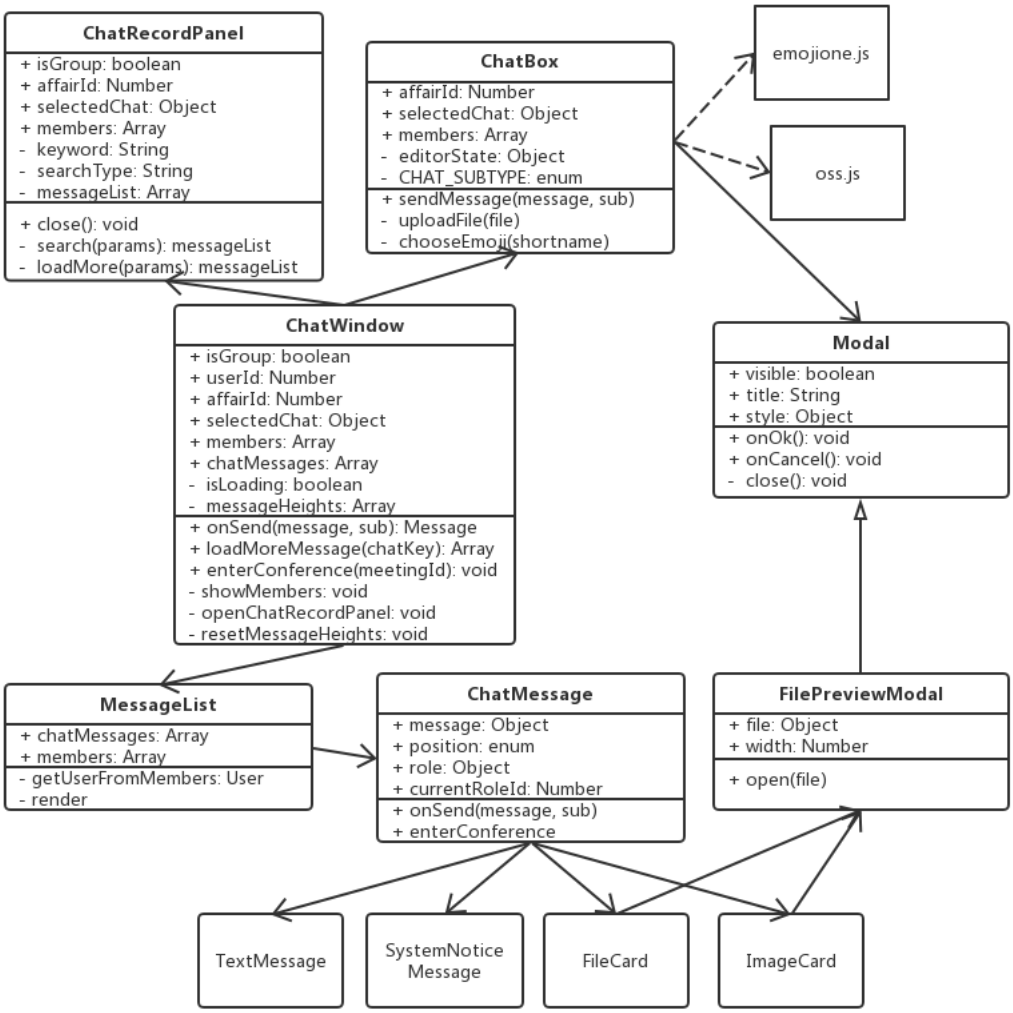

图 3.10 会话通用组件详细设计类图

如上图 3.10 所示展示了会话通用聊天框组件的详细设计类图，本类图主要对聊天框组件中的窗口、消息列表、消息输入框、查找历史消息面板、消息展示组件和它们之间的组织关系进行了详细描述，其中的组件类都是包含 HTML、CSS 和 JS 的 React 组件。ChatWindow 作为聊天框组件的最外层容器，负责渲染当前选中会话的历史消息和消息输入框，上下文环境向 ChatWindow 中更新历史消息列表 chatMessages 和传入消息发送回调函数 onSend，子组件 MessageList 和 ChatMessage 负责根据不同消息的类型，引用不同的消息类型

卡片组件渲染消息内容，消息类型卡片目前有：TextMessage 普通文本消息、SystemNoticeMessage 系统通知消息、FileCard 文件消息、ImageCard 消息。ChatBox 类提供有一个文本编辑框和各种消息的选项栏，用户的输入和选择操作通过文本编辑框和选项栏获取，此类指在将用户的各种不同类型的输入构造成会话消息规定的数据结构，通过父组件 ChatWindow 传入的 onSend 方法发送出去。ChatBox 组件中也会使用一些第三方库的静态方法，如 oss.js 是用于上传文件至阿里云的方法封装，emojione.js 是用于转换 emoji 表情的工具类。ChatBox 中有一些业务方面的消息发送会使用到公共组件弹出框 Modal，在 FileCard 和 ImageCard 中由于需要实现文件预览功能，也引用了继承自 Model 组件的 FilePreviewModal 文件预览模态框组件。FilePreviewModal 组件接收不同类型的文件资源地址，根据不同文件类型，输出不同的渲染界面。

会话聊天框组件的主要职责就是获取上下文中的会话数据并渲染出来，还有根据用户的操作向外输出会话消息，并管理用户聊天过程中的所有交互，快速响应数据更新，重绘页面。每个组件设计单一职责，分工明确，利用 React 的单向数据流思想，组件设计只需考虑上层输入的数据和自己输出的数据，数据决定视图。

3.4.2 事务内会话模块详细设计

如下图 3.11 所示展示了事务内会话模块的详细设计类图，事务内会话模块主要由 UI 组件事务会话容器 AffairChatContainer、通讯录组件 ChatGroupsPanel、聊天框组件 ChatWindow，Redux 数据相关类 ChatAction、ChatReducer，接口服务类 ChatService 和封装 Socket 的 SDK。

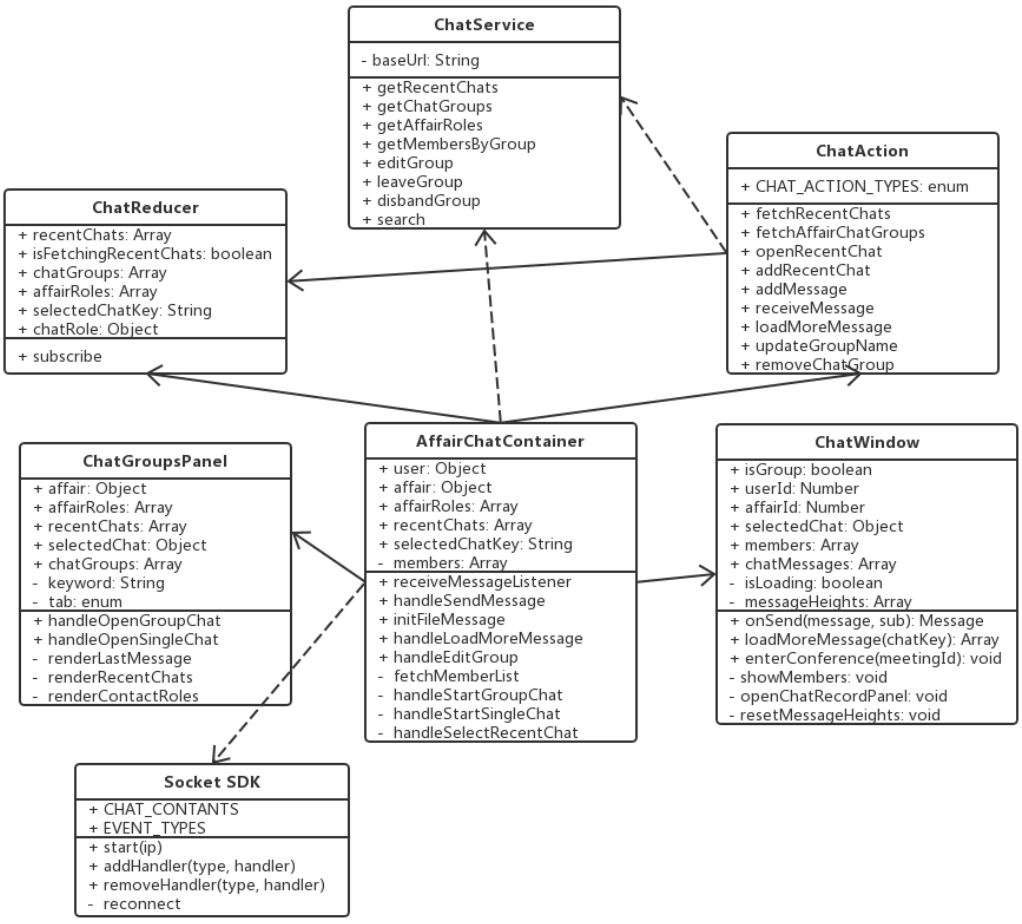

图 3.11 事务会话模块详细设计类图

AffairChatContainer 组件作为事务会话的页面容器，一方面是订阅 ChatReducer 类中的数据，根据数据渲染 ChatGroupsPanel 通信录组件和 ChatWindow 组件，此数据与 3.4.5 小节中介绍的会话前端缓存数据相一致，数据的更新也是由 AffairChatContainer 组件中绑定的 ChatAction 中的方法实现。另一方面，它还负责在用户进入会话界面时去向 Socket SDK 注册消息监听，负责接收服务端推送的会话实时聊天数据，并更新会话列表，在用户离开会话界面时，组件也负责将消息监听去除，并清空 ChatReducer 中的缓存数据。

ChatGroupsPanel 组件负责渲染事务会话中的最近会话列表和通信录页面，实现最近会话列表的更新、排序、切换和消息提醒，在通讯录页面实现开启群聊最近会话和单聊最近会话，通过 tab 变量在最近会话列表之间切换。

ChatService 类主要负责封装与后端交互的 API 方法。

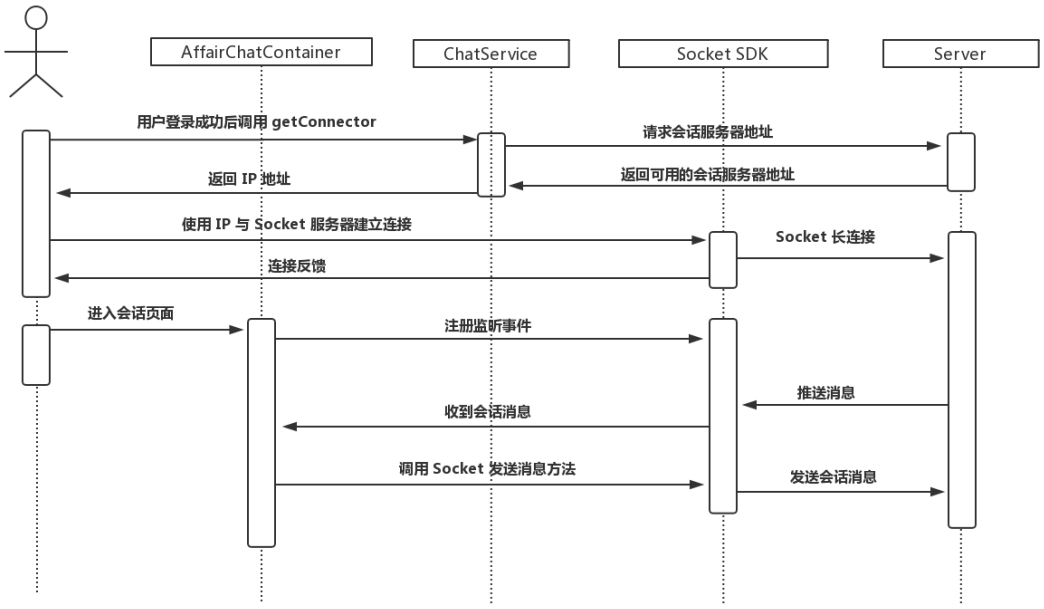

图 3.12 Socket 连接顺序图

如上图 3.12 所示为 Socket 连接顺序图，用户在主系统中登录成功后会调用 ChatService 的 getConnector 接口，向后端查询可用的会话服务器地址，收到 IP 地址后会根据 userId 初始化一个 Socket 连接，Socket 会话消息也是根据 userId 推送的，这样就建立起客户端与服务器的 Socket 长连接。当用户进入会话页面时，AffairChatContainer 组件会向 Socket SDK 注册消息监听，实现会话消息的接收，同时 Socket SDK 也向客户端提供发送消息的接口。用户退出会话页面时，AffairChatContainer 组件会自动移除监听。

3.4.3 视频会议模块详细设计

如下图 3.13 所示展示了视频会议模块前端的详细设计，视频会议模块主要由视频会议页面容器组件 ConferenceContainer、全局入口 VideoConferenceHOC 组件、创建视频会议组件 StartConferenceModal 和用于实现 WebRTC 音视频通信的第三方库 erizo.js 组成。

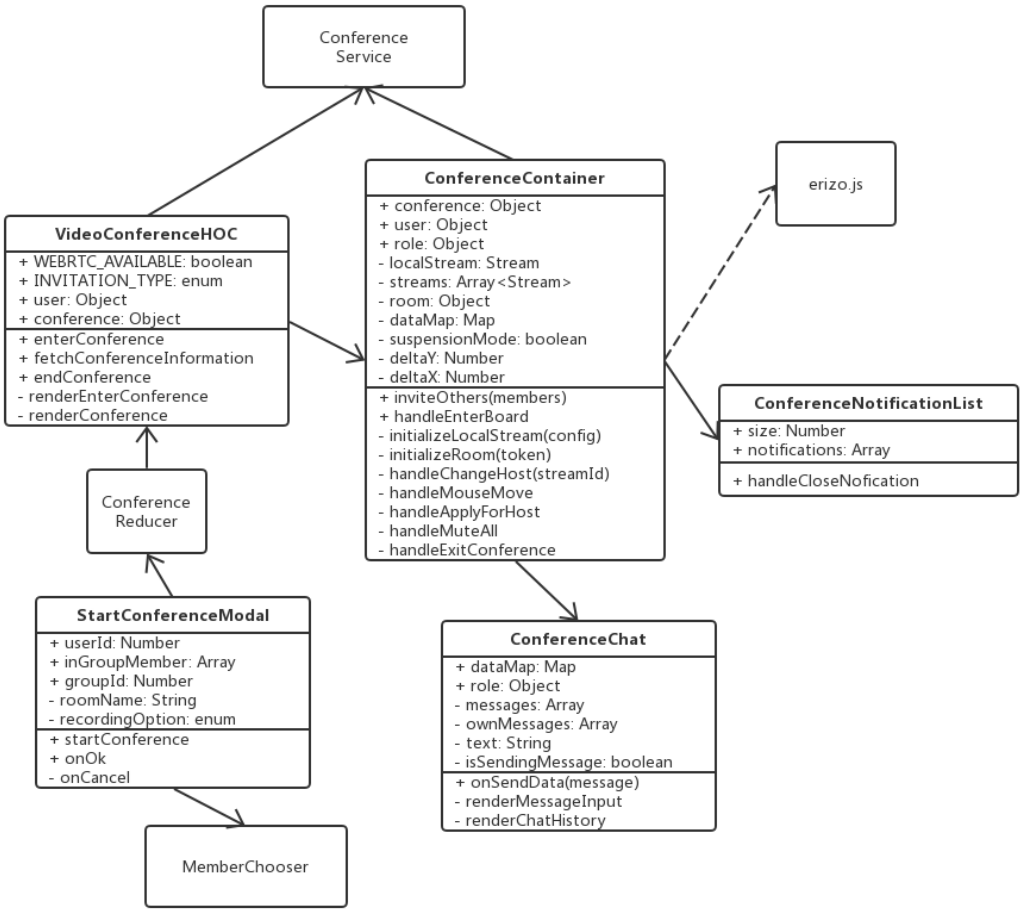

图 3.13 视频会议模块详细设计类图

视频会议模块在通信方面使用的是基于 WebRTC 的 Licode 第三方框架，后端基于 Licde 实现会议房间相关的逻辑，负责管理视频会议 sessions，管理服务器资源（会议房间，与会用户，加入凭证），前端使用基于 webRTC 针对视屏会议场景的一对多组件 erizo.js 初始化本地流，订阅会议房间远程流和注册监听事件。下图为会议房间初始化流程图：

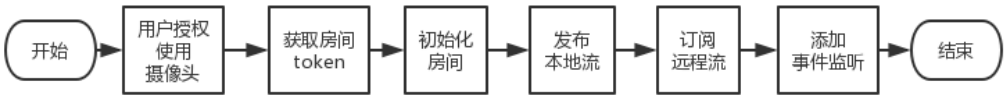

图 3.14 会议房间初始化流程图

视频会议的关键部分是如何在前端维护整个房间状态和房间逻辑。视频会议的服务端分为业务服务端和 Licode 服务端，业务服务端管理会议房间的创建和记录并与主系统交互，Licode 服务端可以看成是拆分出来的一个视频微服务，

只负责当前在线视频会议的 sessions 和通信。进入视频会议后，ConferenceContainer 会先获取用户 token，在 Licode 服务端标记当前用户已进入的房间，使用 token 和 Erizo 组件初始化房间，并向房间中发布本地流，同时订阅房间中的其他远程流，然后通过在远程流上绑定消息监听事件，实现各个流在房间中的通信。

VideoConferenceHOC 组件中订阅全局状态管理中心中 ConferenceReducer 的数据，视频会议的入口一个是用户通过 StartConferenceModal 组件创建视频会议，另一个是收到别人的邀请通知时，VideoConferenceHOC 组件弹出全局提醒，可以由此进入视频会议。

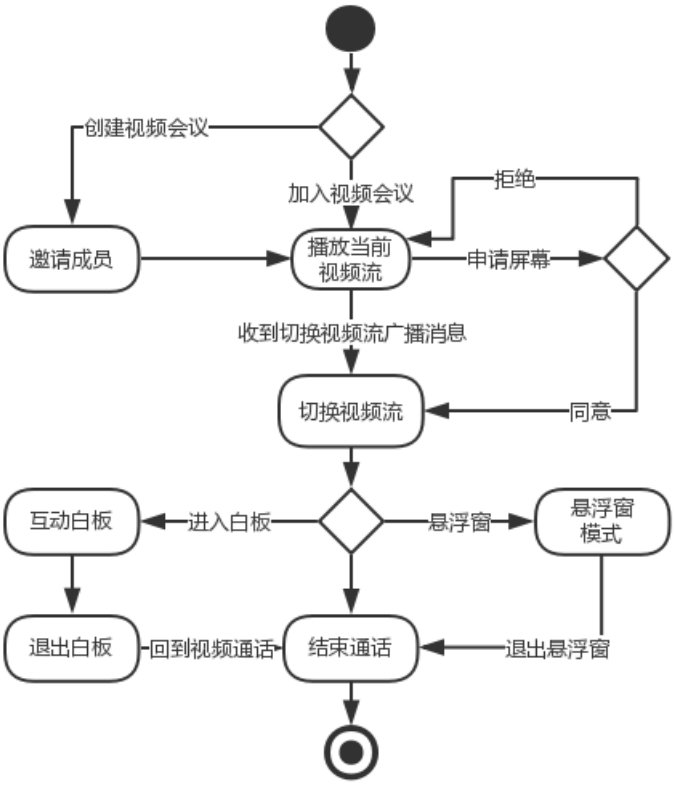

图 3.15 视频会议功能活动图

如上图 3.15 视频会议功能活动图所示，视频会议中有加入成员、申请屏幕、切换视频流、简单的文字聊天等功能，存在消息的传输和通信需求。其中，需要向远程用户广播的消息有：成员加入退出通知、申请屏幕以及其接收和拒绝消息、简单文本消息，全员禁麦通知。可以本地存储的状态信息则是进入白板和悬浮窗模式。广播消息只在前端传输和使用，所以消息的格式由前端定义，ConferenceContainer 通过监听函数收到其他客户端发送的消息，并根据相应的消息类型响应处理，包括切换当前视频流、申请屏幕等。

3.4.4 互动白板模块详细设计

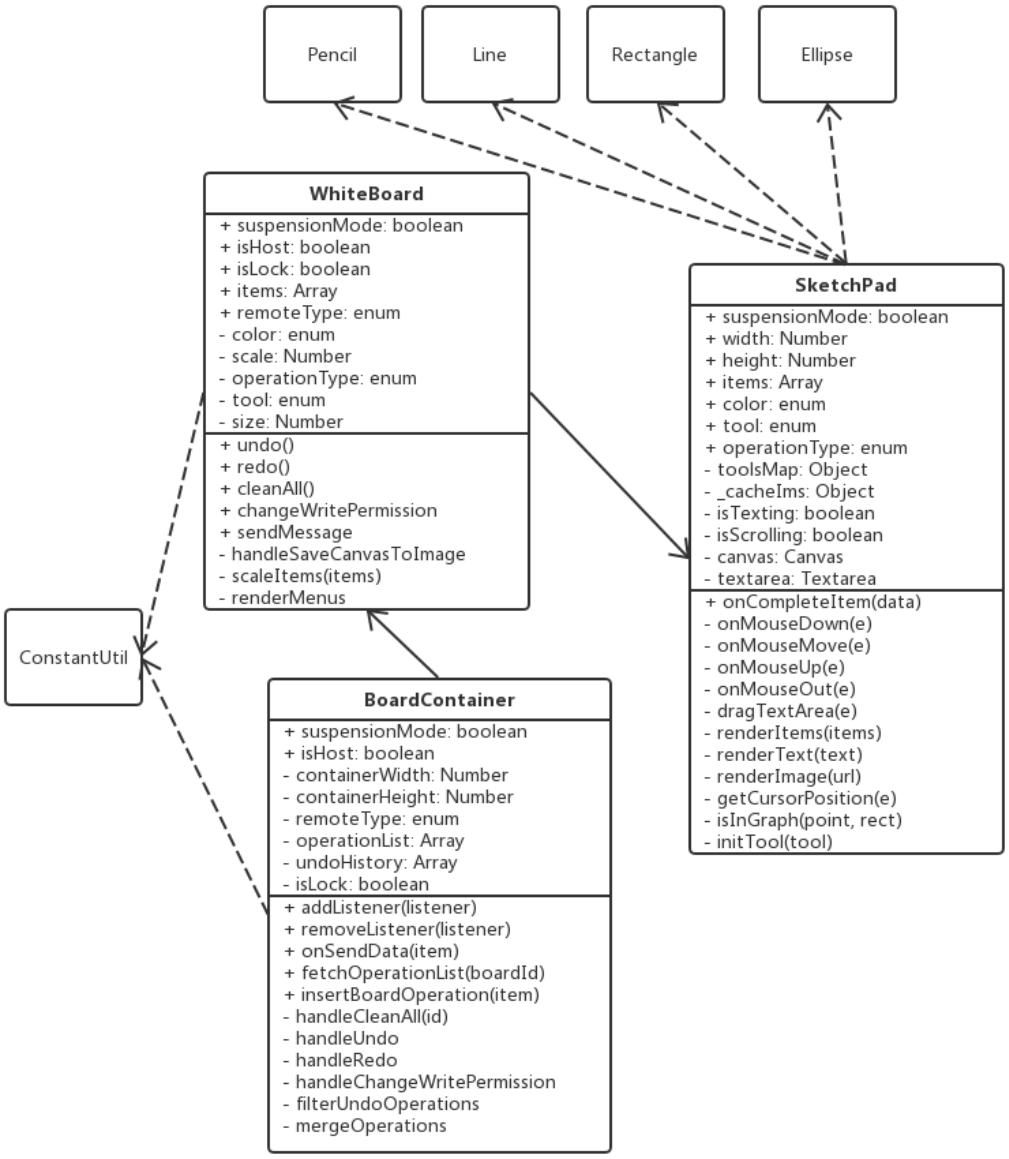

图 3.16 互动白板模块详细设计类图

如图所示为互动白板模块前端详细设计类图，BoardContainer 组件作为白板页面容器，负责注册上层消息监听事件及提供消息发送方法，用户在白板上的操作都会转化为一条操作数据，从最外层转发出去，同时该类也负责进入白板时历史数据的初始化和新数据的插入，在本地维护当前白板的所有操作数据，并根据消息监听实时更新数据。WhiteBoard 组件负责管理白板的操作工具栏，以及用户当前所选中的工具。SketchPad 组件中实例化一个 canvas 对象，作为白板的

画布，所有的用户操作包括绘图、插入文本图片等都是使用 canvas 提供的 API 实现的。为了响应用户操作，SketchPad 组件在 canvas 画布上会绑定各种鼠标监听事件，通过对用户鼠标事件在不同阶段的处理，来模拟画图工具。SketchPad 不仅是将用户操作转化为固定格式的数据转发出去，还需实现将操作数据转化成图形的方法。上层组件维护的白板操作数组传入 SketchPad 组件中，需要根据不同操作类型和具体的内容数据，将操作在 canvas 上渲染出来。ConstantsUtil 类中定义的是一些白板数据的类型常量，Pencil、Line、Rectangle、Ellipse 类是作为绘图画笔的工具类，分别对应画笔、直线、矩形、圆的画图工具。

表 3.5 白板操作数据结构表

字段	使用
id	前端根据操作数据生成的唯一标识（hash）
timestamp	消息发送时间，Date 对象
userId	发送者 id
op	消息类型字段
pos	当前操作所占画布的位置和大小
data	不同类型操作的内容字段

如上表 3.5 所示，不同类型的白板操作只在 data 字段上会有区分。

在互动白板的设计中，后端只提供数据存储、查询和删除服务，且存储的只是白板操作数据 json 格式化之后的字符串，对白板操作的具体数据格式进行了隐藏，因为白板数据只在前端传输并使用，在初始化白板时，前端会根据 boardId 去获取当前白板的历史数据数组，并在本地维护。

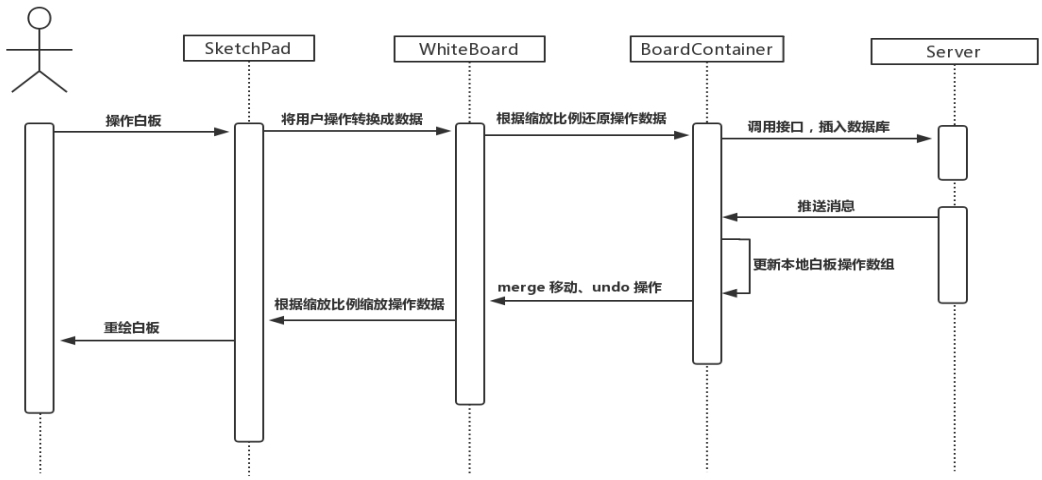

图 3.17 互动白板消息发送和监听顺序图

互动白板消息发送和监听顺序图如上图 3.17 所示，在用户完成一步绘图操作时，SketchPad 组件收集用户操作转化为数据发送给 WhiteBoard 组件，WhiteBoard 根据当前白板缩放比例还原操作数据，BoardContainer 会将此操作数据广播给当前参与白板的其他客户端用户，同时像后台插入一条白板历史数据，以此来实现白板历史记录的保存功能。当收到一条远程流推送的消息时，BoardContainer 更新本地维护的操作数组，WhiteBoard 将操作数组中的移动和 undo 操作 merge 一遍，SketchPad 组件获取更新之后的操作数组重新绘制白板。

3.5 本章小结

本章主要阐述了会话系统的需求分析和基本功能需求，简单介绍了“超级账号”主系统的功能概述和架构设计。在了解会话系统需求的基础上，对系统进行了概要设计，对前端进行了架构设计和模块划分，描述每个模块可能遇到的技术难点以及解决思路。最后对前端各模块进行详细设计，给出细化的实现方案，描述相应的详细类图和顺序图。本文将在下一章从每个模块的技术难点和具体实现方案进行详细的阐述。

第四章 会话子系统的前端实现

4.1 会话通用组件的实现

4.1.1 消息发送

消息发送是会话组件的关键功能，需要实现将用户的操作转化为同一标准格式的数据，具体的消息类型有文本消息、图片、文件、业务消息消息、视频会议，文本消息中还会包含表情，数据结构依据上文 3.4.4 小节中的消息体数据结构设计表实现，发送逻辑由 ChatBox 组件内部实现。

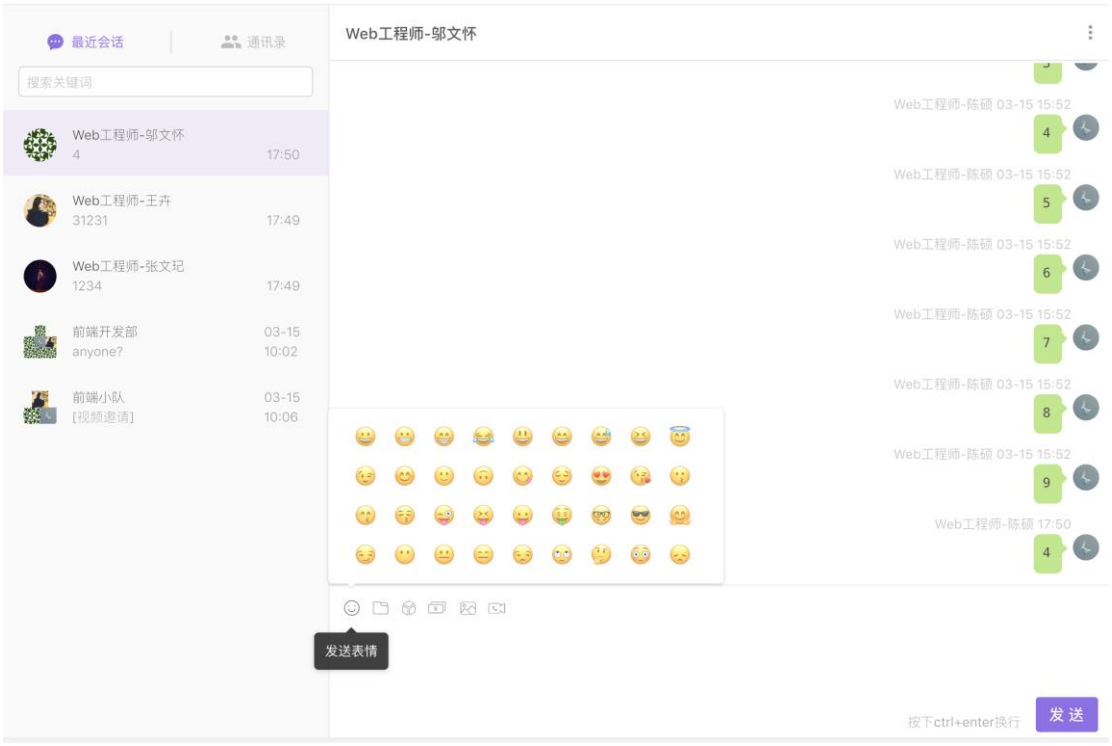

图 4.1 会话通用组件实现界面

会话通用组件具体界面如上图 4.1 所示，文本消息通过使用第三方库 draft.js 提供的 React 组件 Editor，在页面上提供一个文本输入区域，同时将输入内容作为 ChatBox 的组件 state 存储在变量 editorState 中，对 Editor 添加键盘绑定事件，根据键盘事件的 keyCode 去判断用户是否按了 Enter 键，如果是则触发消息发送处理逻辑，发送完成后。Editor 中的内容由 editorState 决定，用户在进行输入时，组件会可以通过 onChange 事件获取用户的具体输入，然后通过 React

组件提供的 setState 方法去修改 editorState 的值，触发界面更新，这也是 React 单向数据流的基本原则。其实用户在界面输入文本看到文本显示在输入框中的过程，已经经过了一层输入验证，这就为之后的发送内容长度限制和发送表情提供了预留的实现接口。其他类型的消息则是通过一组输入选项栏给用户提供操作入口，比如表情面板、发送文件、发送图片按钮等。如下图 4.2 所示为 ChatBox 组件中发送消息和键盘监听的相关代码实现：

```
// Editor 组件 UI 部分
<div className={styles.textarea} onClick={this.focus}>
  <Editor
    blockStyleFn={getBlockStyle}
    editorState={editorState}
    handleKeyCommand={this.handleKeyCommand}
    keyBindingFn={this.sendKeyBindingFn}
    onChange={this.onChange}
    plugins={plugins}
    ref={(element) => { this.editor = element }}
  />
</div>

// 键盘监听事件
sendKeyBindingFn = (e) => {
  if (e.keyCode === 13 && !(e.metaKey || e.ctrlKey || e.shiftKey)){
    //换行，metaKey || ctrlKey + enter
    return 'send-message'
  }
  return getDefaultKeyBinding(e)
}

//发送普通消息
handleSendMessage = () => {
  let { editorState } = this.state
  let contentState = editorState.getCurrentContent()
  if (contentState.hasText()) {
    let message = ''
    contentState.getBlocksAsArray().map((block) => {
      if (block.text === '') {
        message += '\n'
      } else {
        message += block.text
      }
    })
```

```
      this.props.onSend(emojione.toShort(message), CHAT_SUBTYPE.DEFAULT)
      // clear editorState 清空输入框
    }
}
```

图 4.2 ChatBox 组件代码

发送表情方面使用的是目前比较流行的 emoji 表情，emoji 是指 unicode 存在对应编码的系统内置的的小图表情。在页面上，只有使用 emoji 表情所对应的 unicode 编码系统才能正常识别。前端会存储一张 emoji 表情的 shortName 和 unicode 对应表。用户打开表情面板选中表情之后，组件获取当前选中的表情的 unicode 码，修改 editorState 的值，将表情在输入框中渲染出来。在用户发送消息时，会对当前输入内容进行检索，将其中属于表情的 unicode 转化成 shortName 的格式（:shortName:），这样在会话消息中传递的就是字符串，而不会出现编码问题，当在不同的系统中时，只需配置相同 shortName 的表情映射表，即可实现表情的灵活替换。

发送文件和图片的实现与其它消息发送有稍许差别，就是会有一个上传文件的过程。在这里暂时先把图片合并到文件中一起阐述，因为它们的上传和发送逻辑一样，只是消息展示时有所区别。发送文件需要把文件上传到云服务器，并将上传之后的文件服务器地址放入消息体中，发送出去的文件会话消息只包含文件的 url，在展示时使用 a 标签标记提供文件下载的功能。

```
// FileCard 组件上传文件代码
  componentDidMount() {
    const { state, position, file } = this.props
    if (state === FILE_STATE.NOT_UPLOADED && file) {
      //发送文件
      this.setState({
        state: FILE_STATE.UPLOADING
      })
      oss.uploadChatFile(file, affairId, roleId, (progress) => {
        this.setState({ progress })
        if (progress === 100) {
          this.setState({
            state: FILE_STATE.UPLOADED
          })
        }
      }, (xhr) => {
        this.setState({ XHR: xhr })
      }).then((res) => {
```

```
            if (res) {
              // 上传成功
              const url = res.host + '/' + res.path
              let fileContent = {
                name: file.name,
                size: file.size,
                url: url,
                ext: file.type
              }
              this.props.updateMessage(JSON.stringify(fileContent))
              this.setState({
                url: url
              })
            } else {
              this.props.cancelMessage()
            }
          })
      }
  }
```

图 4.3 FileCard 组件上传文件代码

如上图 4.3 所示，在发送文件中，关键实现是 FileCard 组件。FileCard 组件有两个功能，渲染本地发送的文件消息，渲染收到的文件消息，提供文件上传的功能。用户选中要发送的文件之后，ChatBox 中通过 type 为 file 的 input 标签获取到所要发送文件的 File 对象，在组装本地 message 时将 File 对象传入，ChatMessage 组件在渲染消息时将 message 传给 FileCard 组件。FileCard 组件在 React 的组件生命周期方法 componentDidMount 方法中检测是否存在 file 变量，若存在则调用 oss 提供的方法上传文件获取 url，将 url 组装成标准的文件消息传递给 Socket 进行广播，同时更新 FileCard 组件的状态。FileCard 组件在渲染收到的文件消息时，则是使用 a 标签结合消息中的 url 生成文件卡片，提供下载按钮。progress 参数用于实时渲染上传文件的进度条，若用户取消上传或上传失败，则通过 cancelMessage 方法取消已发送的消息。

4.1.2 响应式布局

会话通用组件因为需要在不同场景中使用，所以需要根据容器的尺寸去渲染自身的大小，在 ChatWindow 组件中，宽度是与容器一致，可以设置成 100%，

而高度则有一些细微调整，首先需占满整个容器高度，其次聊天框的 header 高度需要固定，ChatBox 的高度也是固定的，中间的消息列表根据容器高度拉伸，这里的实现使用了 CSS Flexbox 布局，将 ChatWindow 容器设置为 display:flex，header、ChatBox 和 MessageList 横向占满，纵向根据属性分配高度，header 和 ChatBox 分配固定高度，MessageList 设置自增长属性，自动填充剩余容器高度，相关代码如图 4.4 所示：

```
:local(.chatContainer) {
  height: 100%;
  width: 100%;
  display: flex;
  flex-direction: column;

  :local(.header) {
    font-weight: 500;
    display: flex;
    justify-content: space-between;
    padding: 15px 8px 15px 15px;
    font-size: 14px;
    flex: 0 0 48px; // 0 表示固定高度
  }

  :local(.messageList) {
    flex: 1 0 300px; // 1 为表示自增长
    margin-top: 2px;
    overflow: hidden;
  }

  :local(.chatBox) {
    flex: 0 0 140px;
  }
}
```

图 4.4 ChatWindow 组件 CSS 代码

4.1.3 消息提醒的实现

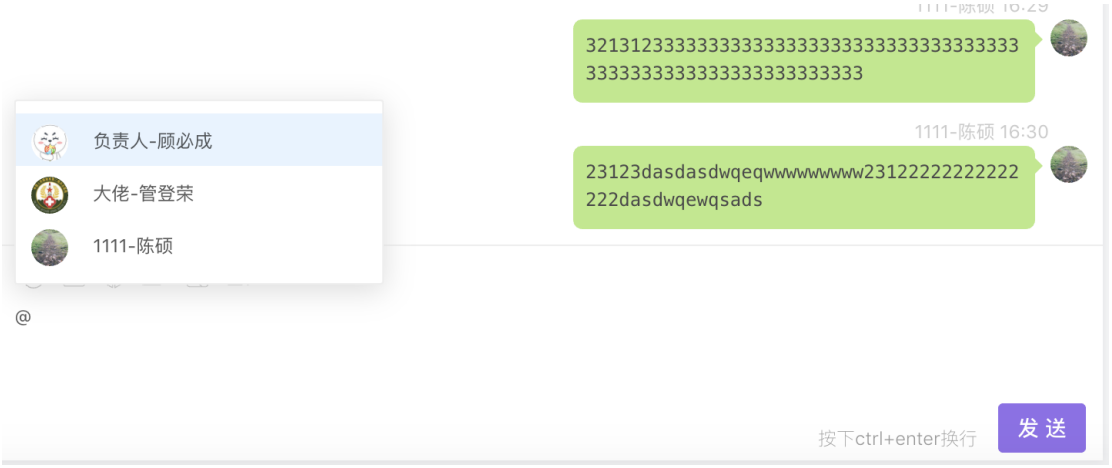

图 4.5 消息提醒@实现界面

消息提醒的实现页面如上图 4.5 所示，用户输入@时弹出用户列表，选中需要@的用户。

用户列表的数据来源于上层组件传入的讨论组成员列表，已经选中的用户列表存储在 ChatBox 的 state 变量 models 数组中，可以用于过滤已选中的用户。在 Editor 组件上绑定键盘监听事件，当用户按下@键时，计算光标位置，并在相应的位置弹出 mentionSuggestions 浮层框，在用户列表中绑定键盘上下选择操作事件和选中事件，当选中某一 mention 时，获取用户昵称组装成‘@’+ 昵称 + 空格的字符串更新 editorState，并使用用户 id 更新 models 数组。考虑到用户可能会有自动输入完整昵称和删减已选中用户的操作，在发送消息之前会对输入内容进行一次全文匹配，检测内容中是否有‘@’+ 昵称 + 空格的字符串能匹配讨论组成员数据中的成员，然后生成@的用户 id 数组 apns，就可以获取当前消息中所要@的用户，相关的代码实现如下图所示：

```
// 使用讨论组成员数据组装 mentions
const createMentions = (members) => {
  return members.map((role) => {
    return {
      id: role.id,
      name: role.roleTitle + role.username,
      avatar: role.avatar,
    }
  })
}
```

```
// Mentions UI 部分代码
<MentionSuggestions
   onSearchChange={this.onSearchChange}
   suggestions={this.state.suggestions}
   onAddMention={this.onAddMention}
/>

// 选中 mention 之后逻辑处理
onAddMention = (value) => {
   const { models } = this.state
   let mention = value.toJS()
   if (models.has(mention.id)) {
      return
   }
   models.set(mention.id, mention)
   this.setState({ models })
}
// 发送消息之前的全文检测匹配代码
let models = [...this.state.models.values()]
let apns = models.filter((m) => message.includes(m.name + ' ')).map((m) => m.id)
```

图 4.6 消息提醒@实现代码

4.1.4 上拉加载

上拉加载是用户已打开某个会话，在消息列表中想查看更多消息时的操作。通过监听 MessageList 的滚动条事件，当用户滚动到消息列表顶部 e.scrollTop 小于 0 时就触发获取更多消息的逻辑，同时使用 isInfiniteLoading 变量去记录当前是否正在获取数据。loadMoreMessage 方法是一个异步函数，返回的是一个 Promise 对象，函数中使用当前会话列表的最后一条消息的 endTime 字段获取之前的更多消息，当异步请求完成后使用 resolve 方法通知 ChatWindow 改变 isInfiniteLoading 状态，更新 MessageList 页面，相关的代码实现如下图所示：

```
// 上拉加载更多
handleScrollTop(e) {
   if (e.scrollTop < 0) {
      if (this.props.chatMessages.size === 0 || this._isInfiniteLoading) return
      let load = () => {
         this.props.loadMoreMessage().then(() => {
            setTimeout(() => {
```

```
                this.setState({ isInfiniteLoading: false })
            }, 100)
          })
      }
      this.setState({
        isInfiniteLoading: true
      }, load)
    }
}
// 异步请求获取更多消息
handleLoadMoreMessage = () => {
    const { affair, recentChats, selectedChatKey } = this.props
    let selectedChat = recentChats.find((c) => c.get('_key') === selectedChatKey)
    let params = null
    let endTime = selectedChat.getIn(['msgList', 0]) ? selectedChat.getIn(['msgList',
0]).time : null
    return new Promise((resolve) => {
      Client.groupChatService.loadGroupMsg(params, (success) => {
        this.props.loadMoreMessage({
          key: selectedChatKey,
          msgList: success.msgList
        })
        resolve()
      })
    })
}
```

图 4.7 上拉加载实现代码

4.1.5 搜索聊天记录的实现

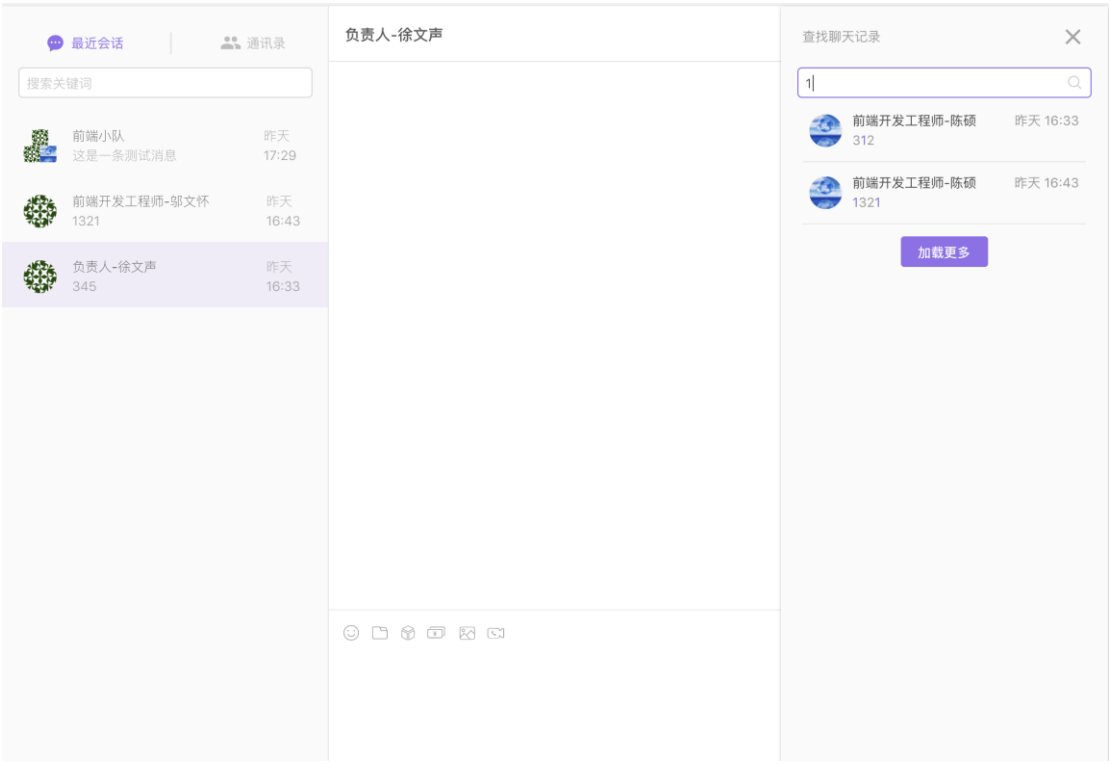

图 4.8 搜索聊天记录界面运行图

搜索聊天记录的功能由 ChatRecordPanel 组件实现，上层只需传入讨论组 groupId 和讨论组成员数据。搜索面板可以通过点击切换搜索不同类型的消息，默认是搜索普通文本消息，同时还可根据时间范围搜索消息。搜索消息的分页是通过底部的“加载更多”按钮实现，组件内部使用 state 变量 isLoadingMore 和 hasMore、pageNumber 实现对分页历史的记录，“加载更多”按钮默认显示，当用户点击时会调一次搜索异步接口，每次请求消息数量为 25，pageNumber 递增 1，当返回的消息数量小于 25 时，则隐藏“加载更多”按钮，并将 hasMore 设为 false。只要在搜索栏中输入内容就会触发搜索，这是通过在搜索的 input 上绑定 onChange 事件，监听内容变化实现的。renderHighlight 方法根据搜索 keyword 高亮搜索出来的消息，相关的代码实现如下：

```
// ChatRecordPanel 类 state 变量
state = {
  keyword: '', // 搜索关键字
  searchType: MESSAGE_TYPES.TEXT, // 搜索消息类型
  placeholder: '搜索关键词',
  messageList: [], // 搜索消息显示列表
```

```
    hasMore: true, // 是否加载更多
    isLoadingMore: false, // 正在加载
    pageNumber: 0, // 当前页数
    startTime: 0, // 起始时间
    endTime: 0, // 结束时间
}

// 加载更多消息
handleLoadMore = () => {
    const { affair, group } = this.props
    const { keyword, pageNumber, messageList } = this.state

    this.setState({ isLoadingMore: true })
    this.fetchSearch(keyword, pageNumber + 1).then((res) => {
        this.setState({
            isLoadingMore: false,
            messageList: messageList.concat(res.data.filter((msg) => msg.sub !==
CHAT_SUBTYPE.IMAGE)),
            pageNumber: pageNumber + 1,
            hasMore: res.data.length === PAGE_SIZE
        })
    })
}
```

图 4.9 搜索聊天记录实现代码

4.2 事务会话模块的实现

4.2.1 最近会话列表的实现

最近会话列表是事务会话中最核心用户操作最频繁的区域，最近会话列表的初始数据由后台接口提供，之后的数据更新就不在通过 HTTP 接口而是根据 Socket 通信。用户打开某一会话时，调用 Socket SDK 提供的 openGroupChat 方法获取会话 msgList 和更新未读消息，将消息列表传入 ChatWindow 组件显示。最近会话列表的更新逻辑较为复杂，主要分为三种情况：用户点击切换会话、消息监听收到消息更新最近会话、通讯录开启会话，具体的更新流程如下图所示：

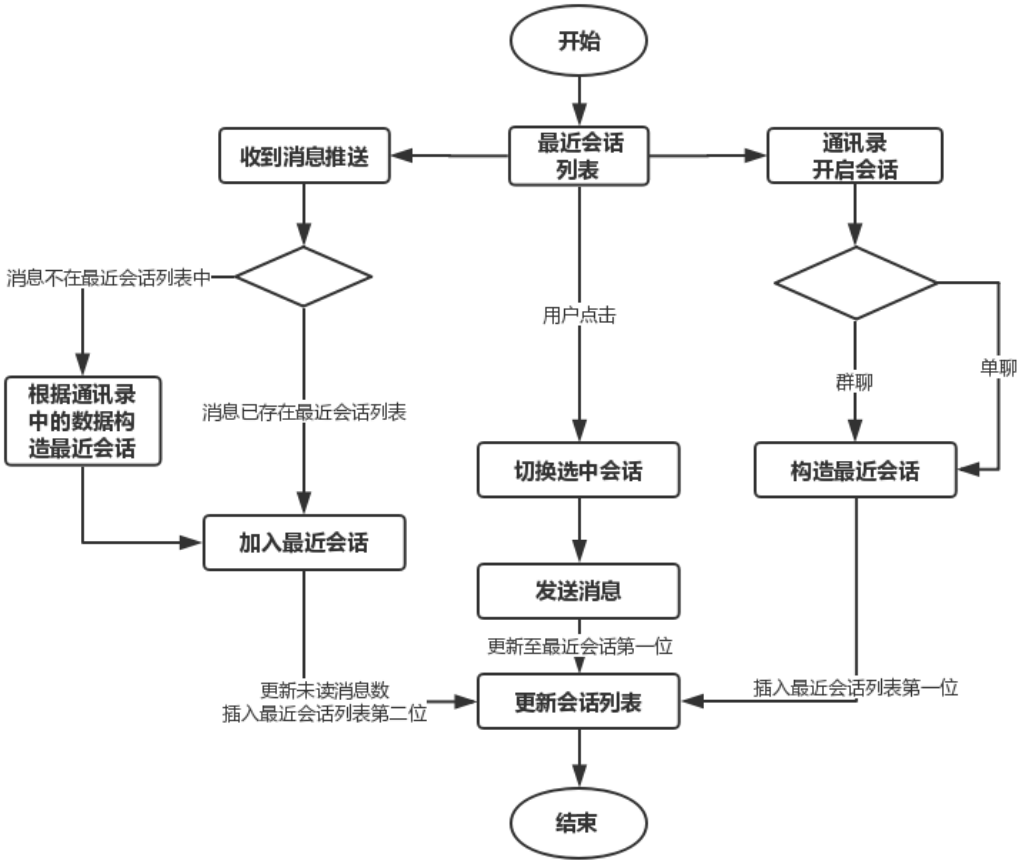

图 4.10 最近会话列表更新流程图

React 组件遵循的设计原则是数据决定视图，最近会话列表的页面组件订阅的数据源是 ChatReducer 中的 recentChats，详细的数据结构在 3.4.5 小节中已经详细阐述过了，所以当需要更新最近会话时，只需要去更新 recentChats 数组，保持单向数据流。recentChat 中会影响页面渲染的字段有：recentChats 数组的排序方式、unread 会话未读消息数、lastMsg 最后一条消息。最近会话列表实现相关代码如下图所示：

```
// 选择最近会话
handleSelectRecentChat = (chat) => {
  if (chat.get('type') === CHAT_TYPE.GROUP) {
    this.openGroupChat(chat.get('groupInfo').groupId).then((success) => {
      let msgList = List(success.msgList)
      chat = chat.set('msgList', msgList).set('unreadCount', 0)
      this.props.openRecentChat(chat)
      this.fetchMemberList(chat.get('groupInfo').groupId)
    })
  } else {
```

```
      this.openPrivateChat(chat.get('toUserInfo').toRoleId).then((success) => {
        let msgList = List(success.msgList)
        chat = chat.set('msgList', msgList).set('unreadCount', 0)
        this.props.openRecentChat(chat)
      })
    }
}

// 开启群聊
handleStartGroupChat = (group) => {
  const { recentChats } = this.props
  this.openGroupChat(group.id).then((success) => {
    let msgList = List(success.msgList)
    // 判断是否已经在最近会话中
    let chat = recentChats.filter((c) => c.get('type') === CHAT_TYPE.GROUP).find((c) =>
c.get('groupInfo').groupId === group.id)
    if (chat) {
      this.props.openRecentChat(chat)
    } else {
      chat = Map(this.createChatData(group, true))
      this.props.addRecentChat(chat)
    }
    this.fetchMemberList(group.id)
  })
}
```

图 4.11 最近会话列表实现代码

当用户在最近会话列表中选中某一会话时，会调用 handleSelectRecentChat 方法，通过 Socket 获取会话最近消息，更新未读消息数为 0，页面上会修改会话选中状态和重新渲染 ChatWindow 界面，如果发送一条消息，则将选中会话更新至最近会话列表第一位。当用户在通讯录中想开启单聊或群聊时，调用 handleStartGroupChat 和 handleStartSingleChat，判断是否已在最近会话列表中，若已存在则与上一情况处理逻辑一致，若不在，则通过 createChatData 方法构造最近会话对象，添加至最近会话列表中，并更新至列表第一位。当收到一条推送消息时，根据消息体中的 key 判断是否属于最近会话列表中的某一会话，若属于则直接加入该会话的 msgList 中，并更新未读消息数，将位置更新至列表第二位。若不属于则根据消息的 fromUserId 或者 groupId 找到通讯录中的数据，通过 createChatData 构造最近会话，加入最近会话列表。

4.2.2 消息监听分发

上一小节中提到收到会话推送消息，利用消息体数据构造最近会话。但是消息并不仅仅只有普通的文本消息，当收到一些讨论组内的系统通知消息，如人员变动、讨论组信息变动时，需要有一些额外的处理，本小节准备就收到的消息类型分别阐述不同情况下的消息处理实现，下图 4.12 所示为消息监听分发的整个流程图：

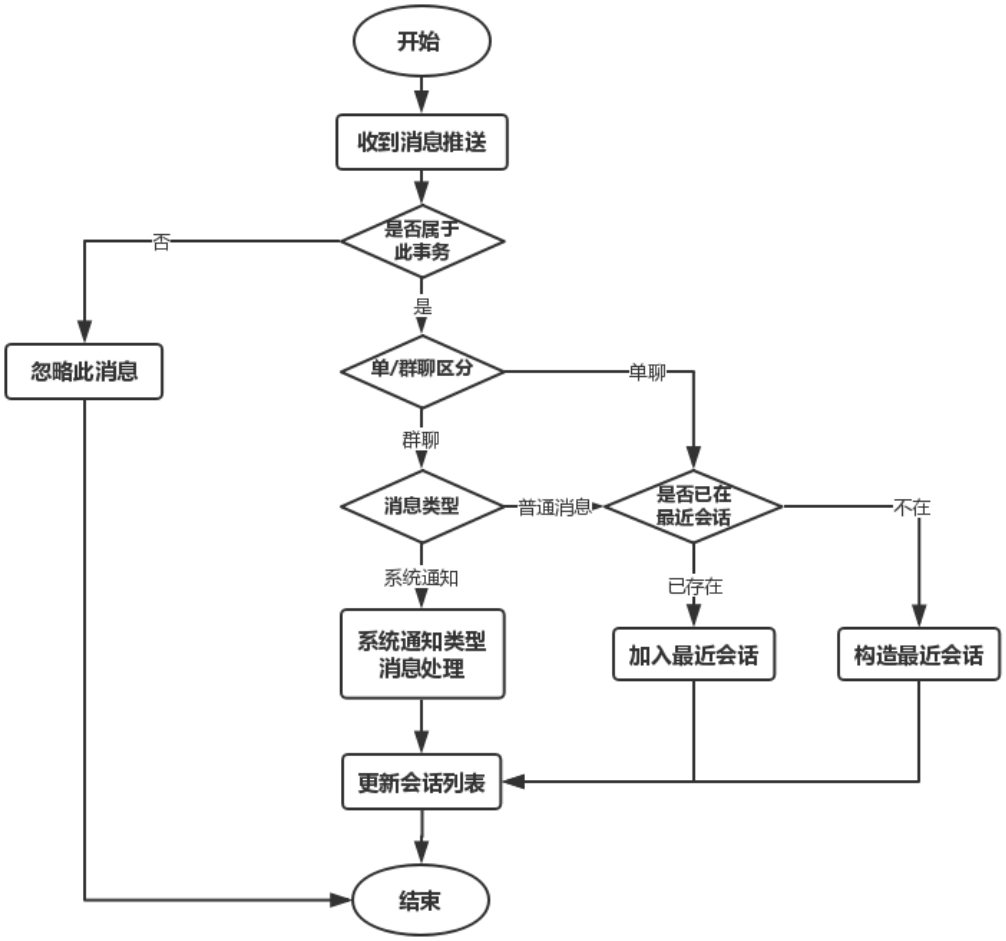

图 4.12 消息监听分发流程图

对于系统通知类型的消息，大部分只会出现在群聊中，具体有以下几种类型：创建讨论组、邀请成员进组、移除讨论组成员、退出讨论组、解散讨论组。需要根据不同的消息类型，更新讨论组信息和成员信息。

```
groupMessageListener = (message) => {
  switch (message.sub) {
```

```
      case groupMessageType.CREATE:
        // 创建讨论组的系统通知,更新群聊列表
        this.props.fetchAffairChatGroups(affairId, roleId)
      case groupMessageType.INVITATION:
        // 邀请讨论组成员, 两种视角
        content = JSON.parse(message.content)
        if (content.inviteeRoleId === roleId) {
          // 已经在讨论组中的成员视角：
          this.props.fetchAffairChatGroups(affairId, roleId)
        } else {
          // 不在讨论组中的成员：
          const newRecentChat = this.createRecentChatByGroupId(message.groupId,
message)
          this.props.createRecentChat(newRecentChat)
        }
      case groupMessageType.REMOVE:
        // 移除讨论组成员，两种视角
        content = JSON.parse(message.content)
        if (content.removeeRoleId === roleId) {
          // 被移除的成员视角：
          if (isInRecentChat) {
            this.props.removeChatGroup({ key: message._key, groupId:
message.groupId })
          } else {
            this.props.fetchAffairChatGroups(affairId, roleId)
          }
        } else {
          // 仍在讨论组中的成员视角：
          // 普通接收消息处理
        }
      case groupMessageType.EXIT:
        // 退出讨论组,仍在讨论组中的成员视角：
        // 普通接收消息处理
      case groupMessageType.DISMISS:
        // 解散讨论组
        // 删除最近会话中的对象，删除被解散讨论组
        this.props.removeChatGroup({ key: message._key, groupId: message.groupId })
    }
}
```

图 4.13 系统通知消息处理代码

收到创建讨论组的消息时，调用 fetchAffairChatGroups 异步请求更新群聊列表。收到成员邀请消息时，若用户是被邀请者，则更新群聊列表，若用户是群组

内其他成员，则正常显示系统消息。收到移除讨论组成员消息时，若用户是被移除成员则删除最近会话列表中的该会话，其他则按系统消息显示。收到退出讨论组的消息时，只按系统消息显示即可。收到解散讨论组的消息时，调用 removeChatGroup 删除群聊数据，若当前讨论组在最近会话中已存在，一并移除。

4.3 视频会议模块的实现

4.3.1 会议房间初始化

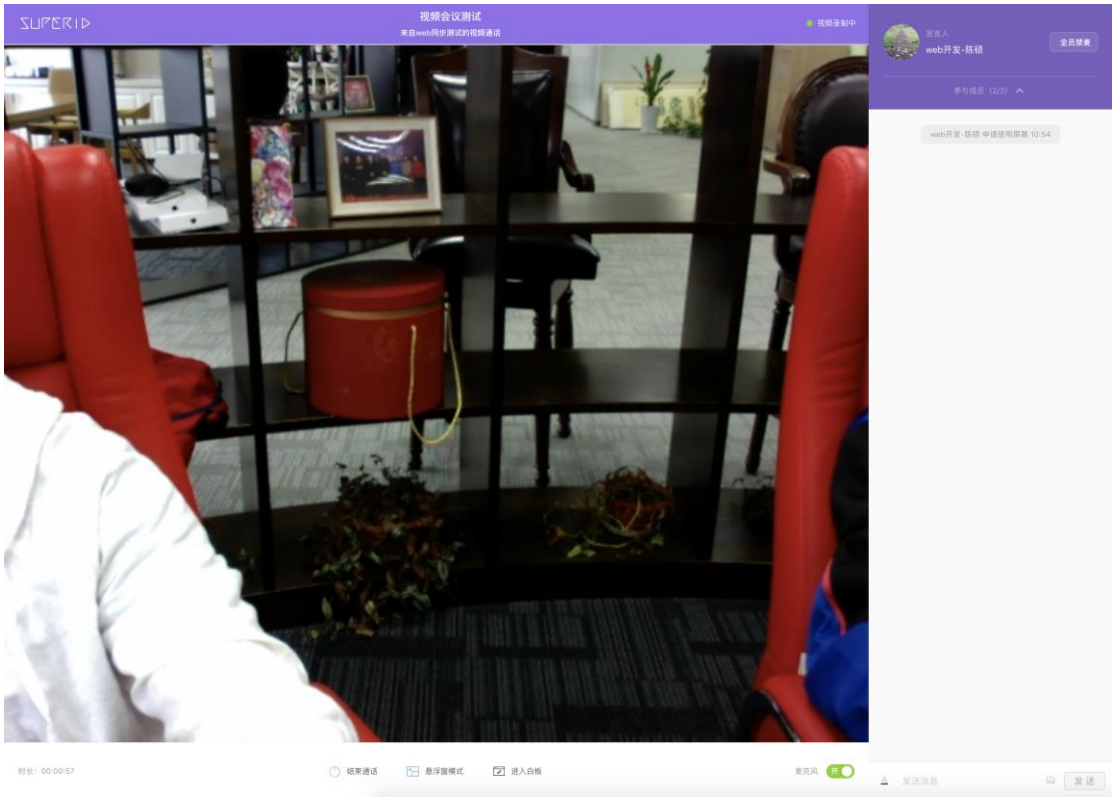

图 4.14 视频会议界面运行图

视频会议的实际运行图如上图所示，视频会议的房间初始化是维持整个视频会议通信的关键。在进入视频会议之前，根据 conferenceId 和 username 访问后台服务获取 token。ConferenceContainer 使用 Erizo 初始化本地流，调用 navigator.getUserMedia 请求用户使用媒体设备，当用户授权之后，使用 token 初始化房间调用 initializeRoom 方法，发布本地流，订阅房间中已存在的远程流，并绑定监听事件。由于用户授权、房间连接、订阅远程流等操作都是异步操作，

所以代码中都是以回调函数的形式实现操作成功后的处理，具体的监听事件有如下几种：

表 4.1 视频会议房间初始化事件表

Event	绑定对象	说明
access-accepted	localStream	用户授权浏览器使用设备
access-denied	localStream	用户拒绝使用设备授权
room-connected	room	房间连接成功
stream-added	room	有新的远程流加入 room
stream-subscribed	room	成功订阅远程流
stream-removed	room	移除远程流
stream-data	Stream	远程流推送消息

下图为房间初始化的具体实现代码，使用 token 初始化房间对象 room，调用 room.connect()连接房间，在房间连接成功之后的回调中调用 room.publish()发布本地流，同时订阅房间中已存在的远程流。此后当房间中有其他远程流加入时，会触发 stream-add 事件，并订阅该远程流。订阅远程流也是一个异步的操作，订阅成功事件为 stream-subscribed，回调中注册该远程流的消息监听事件，实现房间内各个流之间的消息通信。

```
initializeRoom(token) { // 初始化房间
  const room = Erizo.Room({ token: token })
  const { localStream } = this.state
  this.setState({
    room: room,
  })
  function subscribeToStreams(streams) { // 订阅远程流的方法
    streams.filter((v) => (v.getID() !== localStream.getID()))
      .filter((v) => v.getAttributes())
      .forEach((v) => room.subscribe(v))
  }
  room.addEventListener('room-connected', (roomEvent) => { // 房间连接成功后的回调
    const options = { metadata: { type: 'publisher' } }
    room.publish(localStream, options)
    subscribeToStreams(roomEvent.streams)
  })
  room.addEventListener('stream-added', (streamEvent) => { // 远程流加入的回调
    subscribeToStreams([streamEvent.stream])
  })
  room.addEventListener('stream-subscribed', (streamEvent) => {
```

```
      const stream = streamEvent.stream // 订阅远程流成功的回调
      const streams = this.state.streams.push(stream)
      this.setState({ streams,})
      stream.addEventListener('stream-data', (evt) => {
        // 监听消息推送回调
      })
    })
    room.addEventListener('stream-removed', (streamEvent) => { // 移除远程流
      const stream = streamEvent.stream
      stream._removed = true
    })
    room.connect()
},
```

图 4.15 会议房间初始化代码

4.3.2 消息订阅与推送

视频会议中需要记录的历史数据较少，通信的数据格式主要由前端定义，去实现申请屏幕、全员禁麦等会议操作，本地向房间中广播消息时需调用 localStream.sendData 方法发送消息，在接收消息方面，需要手动在订阅每个远程流成功后通过 stream.addEventListener 添加 stream-data 事件监听，当远程流有数据推送时，就会调用监听中的回调函数。

```
stream.addEventListener('stream-data', (evt) => {
    const { msg } = evt
    const attributes = stream.getAttributes()
    const boardListener = this.boardListener

    switch (msg.type) {
      case 'APPLY_FOR_HOST':
        // 申请屏幕
      case 'MEMBERS_UPDATE':
        // 成员变动
        this.props.fetchConferenceInformation(this.props.conference.get('conferenceId'),
this.props.affair.get('roleId'))
      case 'I_AM_THE_HOST':
        // 切换本地播放的视频流
        this.handleChangeHost(msg.payload.streamId)
      case 'MUTE_AUDIO':
        // 全员禁麦通知
        this.handleMuteAudio(msg.payload.streamId, msg.payload.muteAudi)
```

```
      case 'I_HAVE_BEEN_REJECTED':
        // 申请屏幕被拒绝
      case 'I_WILL_BE_THE_HOST':
        // 申请屏幕成功后的通知
      case 'BOARD_MSG':
        // 白板消息
        boardListener && boardListener(msg.content)
      default:
        // 其他自定义的数据（聊天）
        this.setState({ dataMap })
  }
})
```

图 4.16 消息订阅实现代码

房间中远程流的消息类型如上图所示，有申请屏幕、成员变动、切换发言人、禁麦、拒绝申请、同意申请、白板消息等处理。

4.3.3 悬浮窗模式的实现

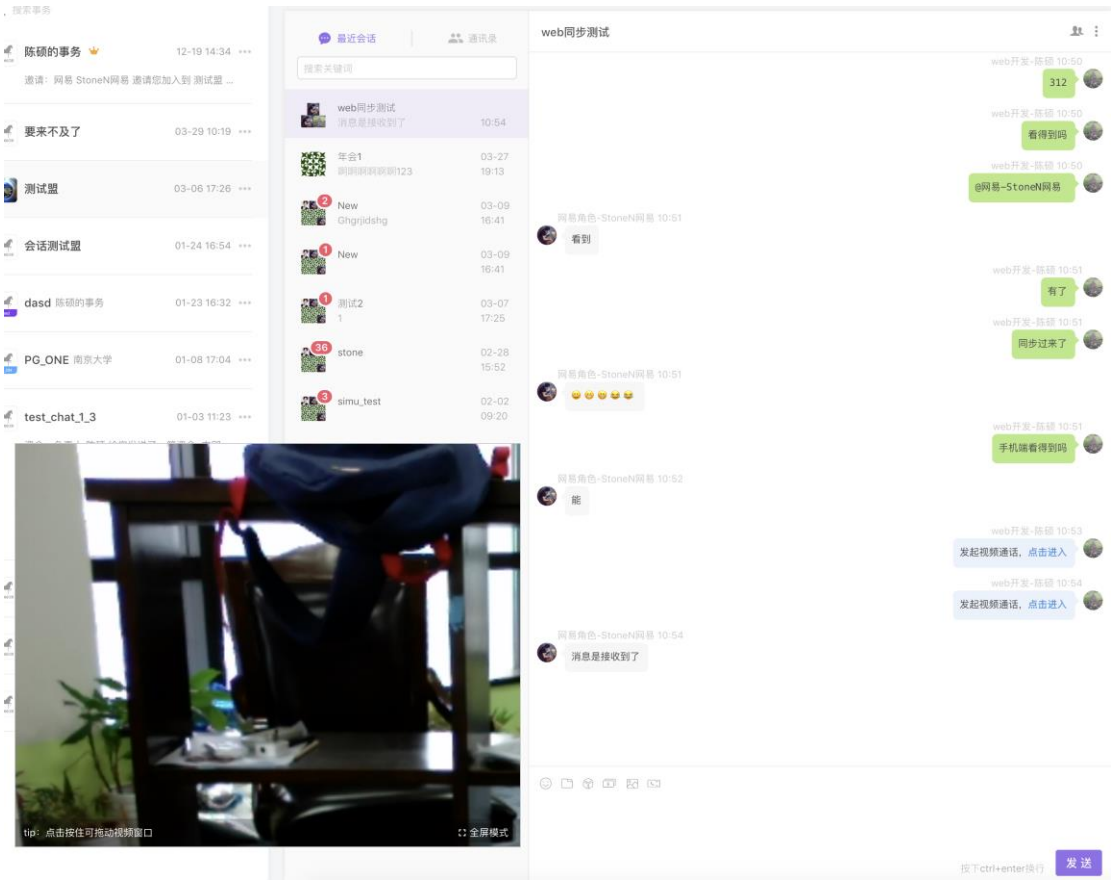

图 4.17 视频会议悬浮窗运行图

悬浮窗模式的实现主要分两步，因为视频会议页面容器的 DOM 元素是作为

全局的元素加在主系统界面上，第一步需要将容器 DOM 缩小，同时视频流播放 DOM 也缩小至相应尺寸，悬浮窗中只播放当前视频流，第二步在容器 DOM 上绑定 MouseDown、MouseMove 和 MouseUp 事件，监听鼠标移动，然后重新计算悬浮窗位置，实现拖拽功能。

```
handleMouseDown(evt) {
  const { clientX, clientY } = evt
  this._isDragging = true
  this._currentX = clientX
  this._currentY = clientY
},
// 鼠标移动事件监听
handleMouseMove(evt) {
  evt.preventDefault()
  evt.stopPropagation()

  const { clientX, clientY } = evt

  if (!this._isDragging) return

  // 计算鼠标移动距离
  this.setState({
    deltaX: this.state.deltaX + (clientX - this._currentX),
    deltaY: this.state.deltaY + (clientY - this._currentY),
  })
  this._currentX = clientX
  this._currentY = clientY
},
// 更新拖拽状态
handleMouseUp() {
  this._isDragging = false
},
handleMouseLeave() {
  this._isDragging = false
},
```

图 4.18 悬浮窗模式实现代码

上图为悬浮窗模式的具体实现代码，_isDragging 标记拖拽是否开始，_currentX 和_currentY 标记拖拽起始位置，deltaX 和 deltaY 为悬浮窗 DOM 左上角的位置，鼠标移动时计算移动距离，并更新悬浮窗位置，松开鼠标后将 _isDragging 设为 false，标记拖拽操作结束。

4.4 互动白板模块的实现

4.4.1 绘图操作的实现

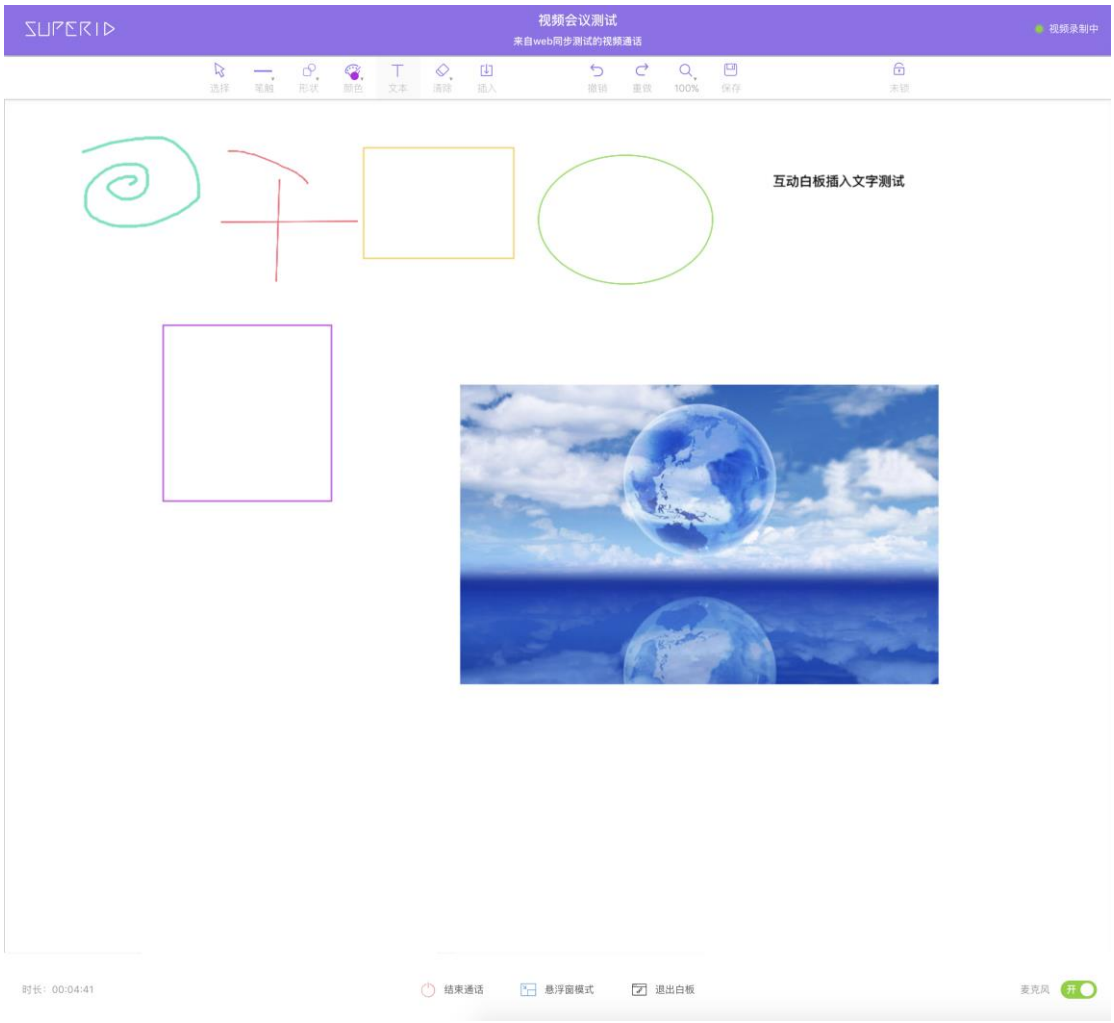

图 4.19 互动白板运行图

互动白板中的绘图操作分为曲线、直线、椭圆和矩形这四类。在画法上有一些一致的部分，都是在画板上监听 MouseDown 事件也就是用户按下鼠标的操作作为画笔的起始点，同时使用 Canvas 的 getImageData 获取当前白板快照，在 MouseMove 时根据鼠标移动的距离实时绘图，在绘制之前并使用 putImageData 还原初始画板，在 MouseUp 事件监听中将绘图数据整理出来，构成白板数据。下图为绘制矩形的具体代码：

```
const onMouseDown = (x, y, color, size, fill) => {
  rectangle = { color, size, fill, start: { x, y }, end: null }
  imageData = context.getImageData(0, 0, canvas.clientWidth, canvas.clientHeight)
```

```
    return [rectangle]
}

const drawRectangle = (item, mouseX, mouseY) => {
    const startX = mouseX < item.start.x ? mouseX : item.start.x
    const startY = mouseY < item.start.y ? mouseY : item.start.y
    const widthX = Math.abs(item.start.x - mouseX)
    const widthY = Math.abs(item.start.y - mouseY)

    context.beginPath()
    context.rect(startX, startY, widthX, widthY) // canvas 绘制矩形 API
    context.stroke()
}
// 鼠标移动时重绘图形
const onMouseMove = (x, y) => {
    if (!rectangle) return
    context.putImageData(imageData, 0, 0)
    context.save()
    drawRectangle(rectangle, x, y)
    context.restore()
}

const onMouseUp = (x, y, tag = true) => {
    if (!rectangle) return
    onMouseMove(x, y)
    const item = rectangle
    !tag && context.putImageData(imageData, 0, 0)
    imageData = null
    rectangle = null
    item.end = { x, y }
    return [item]
}
```

图 4.20 绘制矩形实现代码

如上图代码所示，imageData 保存绘图操作前的白板快照数据，在 MouseMove 时不断地使用 putImageData 重绘白板，在将当前移动位置的画笔画上去，最后得到矩形的数据。

这样直线、曲线、矩形、椭圆的数据格式已经可以分析得出，如果把这个白板画布 Canvas 看成是一个二维坐标系的话，线和形状都可以用其中的 x、y 坐标点表示。直线可以看成是两个点 start、end，曲线其实只是若干条直线的集合，就是一组点集 points，如果点越密集，则线条越平滑。矩形可以用左上和右下的

坐标点固定，椭圆是由圆心坐标点和长轴半径、短轴半径确定的。

4.4.2 插入文字、图片

插入文字的实现依赖于 Canvas 提供 fillText 方法，此方法传入一个文本字符串和起始点坐标，就可在 Canvas 上插入文字。插入文字的实现主要集中在交互部分，用户选中文本工具，点击画布上的任一位置，弹出一个 textarea 输入框，用户可以自由拖拽，当按下 Enter 键时，即在 textarea 所在位置将输入框内的文本渲染至画布上。

插入图片与插入文字在交互上类似，不过图片的渲染性能开销大，需要在缓存方面做一些特殊的处理，插入图片的上传和渲染实现代码如下图所示：

```
/*
* 图片 file-input 的 onchange 事件
* 创建 Image 对象插入图片（缓存图片对象），同步插入操作
*/
onFileChange(e) {
  const file = e.target.files[0]
  if (file) {
    const pos = this.fileInput.pos
    let reader = new window.FileReader()
    reader.readAsDataURL(file)
    reader.onloadend = () => {
      let base64data = reader.result
      const img = new Image()
      const mid = 'img_' + v4()
      img.src = base64data
      this._cacheImgs[mid] = img
      img.onload = () => {
        this.ctx.drawImage(img, pos[0], pos[1])
      }
    }
  }
}
```

图 4.21 插入图片实现代码

Canvas 提供的绘制图片的 API drawImage 只支持 Image 对象的传参，图片先转化为 base64 数据放在白板消息中传输。

首先使用 type 为 file 的 input 标签可以获取到用户上传的文件对象

e.target.files，之后使用浏览器内置的 FileReader 对象读取该文件，获取到该图片的 base64 数据，创建一个 Image 对象 img，img.src 设为 base64 数据即可渲染图片。由于图片的加载是一个异步的过程，频繁地重绘白板时加载图片会出现抖动、卡顿，所以在渲染白板图片时会使用 _cacheImgs 对象缓存已经加载过的 Image 对象，操作 id 组成标识字符串作为 Image 对象的 key 值，之后在 renderImage 时，先根据操作 id 去 _cacheImgs 中查找是否有已加载的 Image 对象，如果有可以直接使用。

4.4.3 框选、移动操作

框选和移动是一组连贯的操作，其中框选又分为画框和识别。在用户视角就是鼠标点击框住一片区域，该区域中的所有白板操作被识别，鼠标可以对已框选操作进行拖拽，鼠标点击未框选区域时即放弃此次框选。画框操作的实现很容易，监听鼠标事件可以获取到框的位置和大小信息，为了实现识别操作，在白板消息的数据结构中加入一个变量 pos，它有 5 个字段：x、y、w、h、center。x 和 y 表示左上角坐标，w 和 h 表示宽高，center 表示中心点坐标。在每次用户操作转化为白板数据时，将可覆盖每个操作的最小矩形计算出来，同时得到此操作的中心点坐标。

```
// 识别被框选操作，计算最小覆盖矩阵
function getSelectedItems {
  const a = this.tool.onMouseUp(...this.getCursorPosition(e), false)
  if (a && a[0]) {
    const data = a[0]
    let rect = { xMin, xMax, yMin, yMax }
    // 绘制框选大矩阵
    let resultRect = { xMin, xMax, yMin, yMax }
    const { items } = this.props
    const selectedItems = []
    items.forEach((item) => {
      if (item.op !== OPERATION_TYPE.CLEAR && item.op !==
OPERATION_TYPE.SELECT) {
        const position = item.data.position
        if (this.isInGraph(position.center, rect)) {
          selectedItems.push(item)
          if (position.x < resultRect.xMin ) {
            resultRect.xMin = position.x
```

```
            }
            if (position.x + position.w > resultRect.xMax) {
              resultRect.xMax = position.x + position.w
            }
            if (position.y < resultRect.yMin ) {
              resultRect.yMin = position.y
            }
            if (position.y + position.h > resultRect.yMax) {
              resultRect.yMax = position.y + position.h
            }
          }
        }
      })
      return [selectedItems, resultRect]
  }
}
```

图 4.22 框选操作实现代码

如上图所示为框选操作中识别被框选操作和计算最小覆盖矩阵的代码，根据用户框选的矩阵 rect 计算有哪些操作的中心点落在框选矩阵，同时根据这些操作的数据计算出最小的覆盖矩阵。移动操作的实现只需要根据被框选操作 selectedItems 和移动 diff 距离不断去计算移动之后的数据，同时注意区分不同类型的操作数据有不同的计算方式。移动操作需要记入历史数据，表示移动操作的内容字段有 ops 和 diff，ops 为选中操作的 id 数组，diff 为横向和纵向移动距离。

4.4.4 撤销、回退操作

白板操作的数据化和移动操作的设计都是为了实现撤销功能，当把所有操作都数据化之后，操作就变成了一条记录，撤销的实现只需要删除最后一条数据记录。撤销操作也会作为一条记录插入后台白板历史数据中，关键的数据结构部分为 opid，表示所要撤销的那条操作的 id。这样实现的原因是为了方便后端，在白板的后端服务中，只需实现创建白板、记录的插入、白板清空接口，不需要了解白板操作具体数据。撤销操作删除的那条数据会暂存在 BoardContainer 类的 undoHistory 数组中，回退操作则是使用这条数据再进行一次插入。

```
// 撤销操作的实现
handleUndo = (msg) => {
```

```
  const { affair } = this.props
  const { operationList, isLock } = this.state
  const roleId = affair.get('roleId')
  const message = Object.assign({}, msg, { id: v4(), roleId,
    timestamp: Date.now(),
  })
  // 找到被撤销操作记录
  const undoItem = operationList.findLast((msg) => msg.roleId === message.roleId)
  message.did = undoItem.id

  // 向房间中其他客户端广播
  this.props.onSendData({
    type: BOARD_MSG_TYPE,
    content: message
  })

  // 修改本地 state 中的数据
  this.insertBoardOperation(message, true)

  // 调用后端接口插入撤销数据
  this.fetchInsertOperation(message)

}
```

图 4.23 撤销操作实现代码

如上图实现代码所示，在获取到被撤销操作的 id 之后，构造撤销操作并广播给房间中的其他客户端，修改本地维护的白板历史数据 operationList，更新 undoHistory 数组，同时调用后台接口插入数据。

4.4.5 历史数据保存

历史数据保存的功能来源于视频会议的参与者进入白板时，需要看到之前互动白板上的数据。BoardContainer 组件在初始化时，调用获取白板历史数据的接口，返回得到一组白板操作数据。根据之前章节中的设计和实现，白板历史数据中会插入一些撤销、移动操作，这些操作本身不产生绘图效果，而是通过修改其他操作的数据来影响视图。所以在获取历史数据和更新白板视图时，可以对这部分数据进行一层过滤和合并操作。

```
// 过滤 undo 操作
filterUndoOperations = (oList) => {
```

```
  const undoIds = oList.filter((o) => o.op === OPERATION_TYPE.UNDO).map((o) =>
o.did)
  const newList = oList.filter((o) => (undoIds.indexOf(o.id) === -1) && (o.op !==
OPERATION_TYPE.UNDO))
  return newList
}
// 合并移动操作
mergeOperations = (operationList) => {
  const data = operationList.toJS()
  let relocate = (pos, diff) => {
    pos.x = pos.x + diff.x
    pos.y = pos.y + diff.y
    const center = pos.center
    pos.center = [center[0] + diff.x, center[1] + diff.y]
    return pos
  }

  let normalItems = []
  let moveItems = []

  data.forEach((item) => {
    switch (item.op) {
      case OPERATION_TYPE.MOVE:
        moveItems.push(item)
        break
      default:
        normalItems.push(item)
    }
  })

  normalItems.sort((a, b) => a.timestamp - b.timestamp)
  const resultItems = normalItems.map((item) => {
    moveItems.forEach((m) => {
      if (m.data.ops.indexOf(item.id) !== -1) {
        item = _moveItem(item, m.data.diff)
        item.data.position = relocate(item.data.position, m.data.diff)
      }
    })
    return item
  })
  return resultItems
}
```

图 4.24 过滤合并白板历史数据实现代码

如上图所示，filterUndoOperations 方法中先找出所有撤销操作，再根据撤销操作的内容找出那些被撤销操作的数据。mergeOperations 方法中先区分出移动操作数组和普通操作数组，根据移动操作的内容，对普通操作的数据进行 move 和 relocate 操作。

4.5 本章小结

本章按照系统的模块划分，详细介绍了会话子系统前端各模块的实现，包括会话通用组件、事务会话模块、视频会议模块和互动白板模块，展示了会话子系统的部分界面运行图。对于各模块中的实现难点，结合模块设计说明解决思路，并结合具体代码阐述了实现方案。

第五章 总结与展望

5.1 总结

随着互联网技术和电子商务技术的发展，企业各种对外的业务活动也已经延伸到了 Internet 上，在这个网络发展迅速的年代，公司内部工作联系不再局限于通信设施（如电话、手机、传真等），而是更多的借助于电脑上的各种通讯软件取得实时联系，同时企业内部沟通和即时通讯的需求也在不断增长。目前市场上的企业即时通讯应用大多是独立运行的应用，有简单的企业部门架构，但脱离于真实的企业管理。所以公司决定在现用的企业协同管理系统中开发会话子系统，为企业管理中各个模块提供基于上下文的聊天群组功能。

本文设计并实现的会话子系统在企业管理中提供不同场景以及不同层级的聊天服务，用户只有在进入相应的上下文环境时，才能开启会话聊天。目前针对“超级账号”主系统，只在事务和发布中加入了会话入口，用户在切换事务或发布时，会更新会话页面，同时保存历史会话记录。会话系统中还提供基于音视频通讯的视频会议服务，以及在 Web 端的多人协助互动白板功能。

围绕会话系统，本文首先概要介绍了该系统的项目背景和研究意义，旨在解决企业管理系统中内部通讯问题。得益于前端技术的飞速发展，本文所设计的会话子系统在前端开发方面，贯彻前端工程化和组件化开发的思想，使用目前最流行的 React 框架搭建前端单页面应用，并开发 React 业务组件库，将大量的逻辑处理和交互交由前端处理，后端面向数据模型提供稳定的接口，同时在 Web 即时通讯方面，会话系统使用了 HTML5 提出的实时通信规范 WebSocket，后端拆分出基于 Node 的会话微服务，构建稳定的会话通信服务。本文的第三章节先对会话子系统的需求进行了具体分析，给出了各个模块的用例图和活动图，对整个系统的架构设计和概要设计进行了阐述，并对关键模块的详细设计进行了深入说明。在第四章节，根据设计给出各实现难点的解决方案，阐述各个功能实现的技术细节，同时展示了关键代码和系统运行界面截图。

5.2 进一步工作展望

本文设计并实现的基于上下文的会话子系统还有很大的改进空间，将在以后的开发中逐步完善。首先，在会话数据的前端缓存方面，对于 Web 应用，缓存的数据不宜过多，且外部环境更新过于频繁需要定期清除历史缓存。还有对于会话系统的稳定性方面，目前会话服务针对网络波动和掉线的重连机制还不完善，影响用户的使用体验，还需要对这一方面进一步开发，优化 Socket 重连机制。

其次，在业务需求方面，会话的应用场景和部分需求还不是很完善，需要通过用户使用反馈不断改进会话的业务需求，对会话系统进行迭代开发。

参 考 文 献

[Pimental, 2012] Pimentel, Victoria, and B. G. Nickerson, Communicating and Displaying Real-Time Data with WebSocket, *IEEE Internet Computing* 16.4(2012):45-53.

[Pahl, 2016] Pahl, Claus, and P. Jamshidi. "Microservices: A Systematic Mapping Study." *International Conference on Cloud Computing and Services Science* SCITEPRESS - Science and Technology Publications, Lda, 2016:137-146.

[Facebook, 2018] https://facebook.github.io/react, Introduction to React maintained by Facebook, 2018.

[Fielding, 2017] Fielding, Roy T, et al. "Reflections on the REST architectural style and principled design of the modern web architecture (impact paper award)." *Joint Meeting on Foundations of Software Engineering ACM*, 2017:4-14.

[Archer, 2015] Archer, Ralph. ReactJS: *For Web App Development*. CreateSpace Independent Publishing Platform, 2015.

[Gackenheimer, 2015] Gackenheimer, Cory. *JSX Fundamentals. Introduction to React*. Apress, 2015:43-64.

[Anthes, 2012] Anthes, Gary. "HTML5 leads a web revolution." *Communications of the Acm* 55.7(2012):16-17.

[Rai, 2013] Rai, Rohit. *Socket.IO Real-time Web Application Development*. Packt Publishing, 2013.

[Loreto S, 2012] Loreto, Salvatore, and S. P. Romano. "Real-Time Communications in the Web: Issues, Achievements, and Ongoing Standardization Efforts." *IEEE Internet Computing* 16.5(2012):68-73.

[Wang, 2013] Wang, Vanessa, F. Salim, and P. Moskovits. *The Definitive Guide to HTML5 WebSocket*. Apress, 2013.

[Geary, 2012] Geary, David M. "Core HTML5 canvas : graphics, animation, and game development." *Pearson Schweiz Ag* (2012).

[Kerchove, 2016] Kerchove, Florent Marchand De. "Extensible modules for JavaScript." *ACM Symposium on Applied Computing ACM*, 2016:2010-2012.

[Staff, 2016] Staff, Cacm. *React: Facebook's functional turn on writing Javascript*. ACM, 2016.

[Npm, 2018] https://docs.npmjs.com/getting-started/what-is-npm, npm Documentation, 2018.

[Babel, 2018] https://babeljs.io, A detailed overview of Babel maintained by Babel Community, 2018

[Fink, 2014] G. Fink, I. Flatow, Modular JavaScript Development, Apress, 2014:35-48

[Clow, 2018] Clow, Mark. *Introducing Webpack. Angular 5 Projects*. 2018.

[Gallaba, 2015] Gallaba, Keheliya, A. Mesbah, and I. Beschastnikh. "Don't Call Us, We'll Call You: Characterizing Callbacks in Javascript." ACM/IEEE International Symposium on Empirical Software Engineering and Measurement IEEE, 2015:1-10.

[Tobias, 2017] https://github.com/webpack/webpack, webpack document maintained by K. Tobias, E. Johannes, T. L. Sean, 2017

[Edan, 2017] Edan, Naktal Moaid, A. Al-Sherbaz, and S. Turner. "Design and evaluation of browser-to-browser video conferencing in WebRTC." *Global Information Infrastructure and NETWORKING Symposium IEEE*,

2017:75-78.

[马兰红, 2015] 马红兰，“互联网+”下企业财务管理模式的探讨，*税收经济研究*，20.6(2015):92-94.

[温馨, 2017] 温馨，*基于Node.js 的Web 前端框架的研究与实现*[D]，东南大学，2017.

[王纪军, 2017] 王纪军等，云环境中 Web 应用的微服务架构评估，*计算机系统应用*，26.5(2017):9-15.

[李春阳, 2017] 李春阳等. "基于微服务架构的统一应用开发平台." *计算机系统应用* 26.4(2017):43-48.

[刘峰, 2016] 刘峰, 陈朴, 贾军营. "WebSocket 与 MQTT 在 Web 即时通信系统中的应用." *计算机系统应用* 25.5(2016):28-33.

[张向辉, 2015] 张向辉等. "基于 WebRTC 的实时视音频通信研究综述." *计算机科学* 42.2(2015):1-6.

[王宝丹, 2017] 王宝丹. "基于 HTML5 Canvas 的画图板设计与实现." *科学与信息化* 36(2017).

[张志飞, 2016] 张志飞. "前端工程化的研究与实践." *电脑知识与技术* 12.25(2016):224-226.

[占东明, 2016] 占东明等. "Web 新兴前端框架与模式研究." 电子商务 10(2016):65-66.

[阮一峰, 2014] 阮一峰. *ECMAScript 6 入门*. 电子工业出版社, 2014.

[严新巧, 2017] 严新巧, 白俊峰. "基于 Dom Diff 算法分析 React 刷新机制." 电脑知识与技术 13.18(2017):76-78.

致　　谢

首先感谢自己的指导老师，感谢与本文相关的其他工作人员，感谢他们在本文工作过程中提供的帮助与鼓

版权及论文原创性说明

任何收存和保管本论文的单位和个人，未经作者本人授权，不得将本论文转借他人并复印、抄录、拍照或以任何方式传播，否则，引起有碍作者著作权益的问题，将可能承担法律责任。

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含其他个人或集体已经发表或撰写的作品成果。本文所引用的重要文献，均已在文中以明确方式标明。本声明的法律结果由本人承担。

作者签名：

日期： 年 月 日