集成音视频 Realtime Web SDK

本文档介绍如何集成扣子 Realtime Web SDK，将你在扣子编程中搭建的 AI 智能体集成到你的 Web 应用中。

扣子 Realtime Web SDK 封装了火山引擎 Web RTC 音视频链路相关 API，接入流程简洁高效。

准备工作

在开始集成 Realtime SDK 之前，你需要先完成以下准备工作。

操作	说明
发布智能体	已成功搭建并发布智能体为 API 服务。搭建步骤可参考搭建可视化智能体手，发布步骤请参见发布低代码智能体为 API 服务。 **说明** 若需使用视频理解能力，请为智能体配置一个视觉模型，例如**豆包视觉理**
获取访问密钥	获取访问密钥，用于身份认证与鉴权。 - **体验或调试场景**：建议生成短期的个人访问令牌（PAT），以快速完成 I 程。个人访问令牌的获取方法请参见添加个人访问令牌。 - **线上环境**：在线上环境中，应使用服务访问令牌（SAT）或 OAuth 鉴权 说明请参考鉴权方式概述。 **说明** 扣子 SDK 封装了多种鉴权方式，能够有效简化鉴权流程，你可以参考鉴 的 OAuth 认证，以获取和管理访问扣子 API 所需的令牌。
准备开发环境	- 安装Node.js 及 npm，具体请参见 Node.js 及 npm，Node.js 版本需 - 安装最新稳定版本的 Google Chrome 浏览器。

跑通示例项目

项目源码

扣子提供基于 React 框架的示例项目源码和 Vue 框架的示例项目源码。通过扣子的 Realtime API 和 Ant Design 的 UI 组件库，实现一个简单的语音通话应用。项目的核心功能包括实时语音交互、消息显示、视频通话（可选）以及麦克风控制等。

你可根据自身需求选择合适的框架。

创建并运行示例项目

1. 创建项目。

   在本地终端执行以下命令，创建一个基于 Vite 和 React 的新项目，并进行初始化和安装依赖。

准备工作
跑通示例项目
  项目源码
  创建并运行示例项目
  源码解析
    基础设置和依赖
    状态管理设置
    鉴权配置
    语音配置
    客户端初始化
    消息处理
    核心功能实现
    UI 实现

```
npm create vite@latest my-react-app -- --template react-ts
cd my-react-app
npm install
```

2. 安装依赖。

   运行以下命令，安装项目所需的依赖包。其中， `antd` 是 Ant Design 的 React UI 组件库，用于构建高质量的用户界面：

```
npm install @coze/realtime-api @coze/api antd
```

3. 创建基础文件。
   a. 进入本地项目目录。
   b. 在 `src` 目录下创建新文件 `hooks.ts` ，并添加以下内容：

```
export const useTokenWithPat = () => {
  // 替换成你的 PAT
  const token = "你的PAT令牌"; // Access Token
  return {
    getToken: () => token
  };
};
```

4. 在 `src` 目录中找到 `App.tsx` ，并用以下内容完全替换原文件内容。

> **说明**
>
> 请将 `botId` 参数的值替换为你的智能体 ID，并在 `App()` 中补全代码。完整的代码示例请参见示例项目中的App.tsx。

```
import { useRef, useState } from 'react';
import {
  EventNames,
  RealtimeAPIError,
  RealtimeClient,
  RealtimeError,
  RealtimeUtils,
} from '@coze/realtime-api';
import { Button, Space, List, message } from 'antd';
import { CozeAPI, COZE_CN_BASE_URL, ChatEventType } from '@coze/api';
import { useTokenWithPat } from './hooks';

// ⚠ 替换成你的智能体ID
const botId = '在这里填入你的智能体ID';

```

```
function App() {
  // ... [这里的代码请参考：https://github.com/coze-dev/coze-js/blob/main/exa
}

export default App;
```

5. 编辑文件 `src/main.tsx`，新增 `antd` 样式，删除 `import './index.css'` 或清空 `index.css` 内容。

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
~~import './index.css'~~
import 'antd/dist/reset.css';
import App from './App.tsx'

createRoot(document.getElementById('root')!).render(
  <StrictMode>
    <App />
  </StrictMode>,
)
```

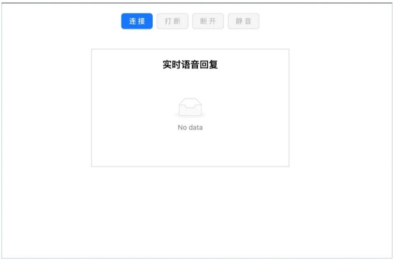
6. 运行项目。
    a. 执行以下命令，在开发环境中启动项目。

    ```
    npm run dev
    ```

    b. 等待浏览器自动打开项目页面。

    项目页面通常部署在本地主机的 3000 端口，须保证此端口未被占用。浏览器会自动访问 `http://localhost:3000` 。

    c. 在打开的项目页面中单击**连接**按钮，连接成功后即可开始对话。



源码解析

本项目的核心代码位于 src/App.tsx，其中包含了 Demo 项目的主要业务逻辑。Demo 项目结合 React、Coze API 和 Ant Design 构建了一个简单的语音通话界面。

基础设置和依赖

以下代码是 React 组件的导入部分，通过以下代码引入必要的依赖和类型定义：

- 使用 React 的状态管理和引用功能来管理组件状态和 DOM 操作。
- 通过 `@coze/realtime-api` 和 `@coze/api` 与扣子智能体进行实时通信和交互。
- 使用 Ant Design 的 UI 组件来构建用户界面，提供良好的用户体验。

```
import { useRef, useState } from 'react';
import {
  EventNames,
  RealtimeAPIError,
  RealtimeClient,
  RealtimeError,
  RealtimeUtils,
} from '@coze/realtime-api';
import { Button, Space, List, message } from 'antd';
import { CozeAPI, COZE_CN_BASE_URL, ChatEventType } from '@coze/api';
```

状态管理设置

智能语音通话项目通常需要管理客户端的状态和行为。示例项目定义了一个名为 `App` 的 React 函数组件，通过 `useRef` 和 `useState` 钩子来管理组件状态和 DOM 操作。

```
function App() {
  // 客户端引用
  const clientRef = useRef<RealtimeClient | null>(null);

  // 核心状态管理
  const [messageList, setMessageList] = useState<string[]>([]); // 消息列表
  const [isConnecting, setIsConnecting] = useState(false);     // 连接状态
  const [isConnected, setIsConnected] = useState(false);        // 已连接状态
  const [audioEnabled, setAudioEnabled] = useState(true);       // 麦克风状态
  const [isSupportVideo, setIsSupportVideo] = useState(false); // 视频支持状态
  // ...其它代码
}
```

鉴权配置

开发者可以选择不同的认证方式来获取访问令牌（Token）。本示例中通过个人访问令牌（PAT）来鉴权，生产环境请切换为更安全的 OAuth 2.0 认证方式。

```
  // 选择一种认证方式
  const { getToken } = useTokenWithPat();
  // 其他认证方式：
  // const { getToken } = useTokenWithDevice();
```

```
// const { getToken } = useTokenWithJWT();
// const { getToken } = useTokenWithPKCE();
// const { getToken } = useTokenWithWeb();
```

语音配置

定义一个异步函数 `getVoices`，通过 OpenAPI 获取可用的音色列表。该步骤为可选操作，你使用系统默认音色，也可以在代码中输入音色 ID 修改为其他音色。

```
async function getVoices() {
  const api = new CozeAPI({
    token: getToken,
    baseURL: COZE_CN_BASE_URL,
    allowPersonalAccessTokenInBrowser: true,
  });
  const voices = await api.audio.voices.list();
  return voices.voice_list;
}
```

客户端初始化

定义一个异步函数 `initClient`，用于初始化一个实时通话客户端（`RealtimeClient`）。该函数包括检查设备权限、获取音色列表、配置客户端参数，并最终创建客户端实例。

```
async function initClient() {
  // 检查设备权限
  const permission = await RealtimeUtils.checkDevicePermission(true);
  if (!permission.audio) {
    throw new Error('需要麦克风访问权限');
  }
  setIsSupportVideo(permission.video);

  // 获取音色列表
  const voices = await getVoices();

  // 初始化客户端
  const client = new RealtimeClient({
    accessToken: getToken,
    botId,
    connectorId: '1024',
    voiceId: voices.length > 0 ? voices[0].voice_id : undefined,
    allowPersonalAccessTokenInBrowser: true,
    debug: true,
    videoConfig: permission.video
      ? { renderDom: 'local-player' }
```

```
      : undefined,
  });

  clientRef.current = client;
  handleMessageEvent(); // 下面会实现
}
```

消息处理

定义了一个异步函数 `handleMessageEvent`，它是一个消息事件监听器，用于处理从实时通话服务器接收到的消息事件，并根据事件类型更新消息列表。

本示例代码仅对部分消息类型进行了处理，主要用于展示实时语音交互中的文本消息内容。开发者可根据实际需求对消息处理逻辑进行扩展或调整。

```
const handleMessageEvent = async () => {
  let lastEvent: any;

  clientRef.current?.on(EventNames.ALL_SERVER, (eventName, event: any) => {
    // 只处理消息增量更新和完成事件
    if (
      event.event_type !== ChatEventType.CONVERSATION_MESSAGE_DELTA &&
      event.event_type !== ChatEventType.CONVERSATION_MESSAGE_COMPLET
    ) {
      return;
    }

    const content = event.data.content;
    setMessageList(prev => {
      // 处理增量更新
      if (lastEvent?.event_type === ChatEventType.CONVERSATION_MESSAGE_DE
        return [...prev.slice(0, -1), prev[prev.length - 1] + content];
      }
      // 处理新消息
      if (event.event_type === ChatEventType.CONVERSATION_MESSAGE_DELTA)
        return [...prev, content];
      }
      return prev;
    });
    lastEvent = event;
  });
};
```

核心功能实现

语音通话的核心功能包括连接、断开、打断以及麦克风控制。以下是相关功能函数的实现：

```
// 连接
const handleConnect = async () => {
  try {
    if (!clientRef.current) {
      await initClient();
    }
    await clientRef.current?.connect();
    setIsConnected(true);
  } catch (error) {
    // 错误处理...
  }
};

// 打断
const handleInterrupt = () => {
  try {
    clientRef.current?.interrupt();
  } catch (error) {
    message.error('打断失败：' + error);
  }
};

// 断开连接
const handleDisconnect = () => {
  try {
    clientRef.current?.disconnect();
    clientRef.current?.clearEventHandlers();
    clientRef.current = null;
    setIsConnected(false);
  } catch (error) {
    message.error('断开失败：' + error);
  }
};

// 麦克风控制
const toggleMicrophone = async () => {
  try {
    await clientRef.current?.setAudioEnable(!audioEnabled);
    setAudioEnabled(!audioEnabled);
  } catch (error) {
    message.error('切换麦克风状态失败：' + error);
  }
};
```

UI 实现

通过以下代码构建一个实时通话应用的用户界面，包括控制按钮、视频显示区域和消息显示区域。

整个组件的布局是一个居中的 `div`，包含以下部分：

- **控制按钮**：用于管理实时通话的连接状态和麦克风状态。
- **视频显示区域**：如果设备支持视频功能，则显示本地视频流。
- **消息显示区域**：显示实时语音回复的消息列表。

```
return (
  <div style={{ textAlign: 'center' }}>
    {/* 控制按钮 */}
    <Space style={{ padding: '20px' }}>
      <Button type="primary" onClick={handleConnect}>连接</Button>
      <Button onClick={handleInterrupt}>打断</Button>
      <Button danger onClick={handleDisconnect}>断开</Button>
      <Button onClick={toggleMicrophone}>
        {audioEnabled ? '静音' : '取消静音'}
      </Button>
    </Space>

    {/* 视频显示区域 */}
    {isSupportVideo && (
      <div id="local-player" style={{width: 400, height: 400}}></div>
    )}

    {/* 消息显示区域 */}
    <div style={{width: '400px', overflowY: 'auto'}}>
      <h3>实时语音回复</h3>
      <List
        dataSource={messageList}
        renderItem={(message, index) => (
          <List.Item key={index}>{message}</List.Item>
        )}
      />
    </div>
  </div>
);
```

集成 SDK

步骤一：安装依赖

在开始集成之前，请确保项目环境已正确配置。运行以下命令安装项目所需的依赖包：

```
npm install @coze/realtime-api @coze/api
```

步骤二：检查设备权限

在初始化 SDK 之前，需确保当前项目已获取设备的麦克风访问权限。如果项目涉及视频通话功能，还需申请摄像头权限。以下是检查设备权限的代码示例：

```
import { RealtimeUtils } from "@coze/realtime-api";

// 检查设备权限
const checkVideo = false; // 如需申请摄像头权限，请设置为 true
const result = await RealtimeUtils.checkDevicePermission();

// 检查麦克风权限
if (!result.audio) {
    throw new Error("需要麦克风访问权限");
}
// 如果申请摄像头权限，还需检查摄像头权限
if (checkVideo && !result.video) {
    throw new Error("需要摄像头访问权限");
}
```

步骤三：初始化 SDK

参考以下示例代码初始化 SDK。初始化时需要传入访问密钥 `accessToken` 、智能体 ID `botId` 和渠道 ID `connectorId` 。

```
import { RealtimeClient } from "@coze/realtime-api";

const client = new RealtimeClient({
    accessToken: 'your access token',   // 替换为准备工作中获取的访问密钥
    botId: 'your bot id',               // 替换为智能体 ID
    connectorId: '1024',                // 渠道 ID，固定为 1024
    allowPersonalAccessTokenInBrowser: true, // 可选：允许在浏览器中使用个人访
});
```

关键参数说明如下表所示。

参数	说明
accessToken	请替换为准备工作中获取的访问密钥，用于身份认证与鉴权。
botId	智能体 ID，获取方法如下： 进入智能体的开发页面，开发页面 URL 中 `bot` 参数后的数字即为智能体 ID。`https://www.coze.cn/space/341****/bot/73428668*****` ，则智能体 ID

connectorId	渠道 ID，固定为 1024。

步骤四：监听事件

调用 `client.on` 监听事件。

SDK 提供了一系列信令事件，建议监听所有信令事件。以下是监听事件的代码示例：

```
import { EventNames } from "@coze/realtime-api";

// 监听所有事件
client.on(EventNames.ALL, (eventName, data) => {
    console.log(eventName, data);
});
```

步骤五：建立连接

在开始对话前，调用 `client.connect` 方法建立客户端和服务端之间的连接。

```
await client.connect();
```

步骤六：断开连接

当结束对话时，调用 `client.disconnect` 方法，断开客户端和服务端之间的连接。

```
await client.disconnect();
```

示例代码

完整的示例代码如下。你也可以参考跑通示例项目获取更多示例代码。

```
import { EventNames, RealtimeClient, RealtimeUtils } from "@coze/realtime-api";

async function initClient() {
    try {
        // 1. 检查设备权限
        const result = await RealtimeUtils.checkDevicePermission();
        if (!result.audio) {
            throw new Error("需要麦克风访问权限");
        }

        // 2. 初始化客户端
        const client = new RealtimeClient({
            accessToken: 'your access token',
            botId: 'your bot id',
            connectorId: '1024',
            allowPersonalAccessTokenInBrowser: true,
```

```
        });

        // 3. 配置事件监听
        client.on(EventNames.ALL, (eventName, data) => {
            console.log(eventName, data);
        });

        // 4. 建立与服务端的连接
        await client.connect();

        // 返回初始化完成的客户端实例
        return client;
    } catch (error) {
        console.error('初始化失败:', error);
        throw error;
    }
}
```

进阶功能

客户端初始配置

Realtime SDK 支持丰富的初始配置选项，初始化时除了必选的 accessToken、botId 及 connectorId 之外，还支持设置音色、开启调试模式、配置噪声抑制等功能。

说明

各个配置项的详细说明，可参考示例项目源码中的构造函数注释。

```
interface RealtimeClientConfig {
  // 必填项
  accessToken: string;    // 访问令牌
  botId: string;          // Bot ID
  connectorId: string;    // 渠道 ID

  // 可选项
  voiceId?: string;       // 音色 ID
  conversationId?: string; // 会话 ID
  baseURL?: string;       // Base URL，默认为 https://api.coze.cn
  userId?: string;        // 自定义用户Id
  workflowId?: string;    // 工作流Id
  debug?: boolean;        // 调试模式，建议开发模式下启用
  allowPersonalAccessTokenInBrowser?: boolean; // 允许在浏览器中使用个人访问令
  audioMutedDefault?: boolean; // 是否默认开启音频，默认 true
  suppressStationaryNoise?: boolean;    // 静态噪声抑制，默认 false
  suppressNonStationaryNoise?: boolean; // 非静态噪声抑制，默认 false
```

```
  videoConfig?: VideoConfig;    // 视频配置
}
```

音视频控制

Realtime SDK 提供了完整的音频控制功能，可控制音频状态、视频状态，设置输入设备和输出设备。

- 控制音频状态：

  你可以通过如下代码控制音频状态，例如开启或关闭麦克风。

  ```
  // 开启/关闭麦克风
  await client.setAudioEnable(true);  // 开启麦克风
  await client.setAudioEnable(false); // 关闭麦克风
  ```

- 控制视频状态：

  你可以通过如下代码实现视频功能，但需要在初始化时配置视频相关的设置。

  ```

  import { RealtimeClient, RealtimeUtils } from "@coze/realtime-api";

  // 需要检查视频设备权限
  const result = await RealtimeUtils.checkDevicePermission(true);
  if (!result.video) {
    throw new Error("需要视频访问权限");
  }

  // 首先，初始化时需要增加视频配置
  const client = new RealtimeClient({
      accessToken: 'your access token',
      botId: 'your bot id',
      connectorId: '1024',
      // ... 其它配置
      videoConfig: { // 视频配置
          renderDom: 'local-player', // 视频渲染dom
          videoOnDefault: true, // 默认开启视频
      }
  });

  // 其次，需要将视频渲染到指定的dom
  <div
  style={{
    width: '300px',
    height: '300px',
    position: 'relative',
  ```

```
  background: '#000',
}}
id={'local-player'}
></div>


// 最后，才可以使用启用/禁用视频功能
client.setVideoEnable(true); // 启用视频
client.setVideoEnable(false); // 禁用视频
```

- 控制输入设备、输出设备：

  你可以通过如下代码获取和设置音频输入设备和输出设备。

```
import { RealtimeUtils } from "@coze/realtime-api";

const devices = await RealtimeUtils.getAudioDevices(); // 获取输入、输出设备
client.setAudioInputDevice(devices.audioInputs[0].deviceId); // 设置音频输入设备
client.setAudioOutputDevice(devices.audioOutputs[0].deviceId); // 设置音频输出
```

调试功能

在调试模式（debug: true）下，可以使用以下功能来帮助开发和问题排查。

- 开启音频属性报告：

  定期输出音频相关的调试信息，帮助开发者了解音频状态。

```
client.enableAudioPropertiesReport({
    interval: 100  // 报告间隔(毫秒)
});
```

- 设备测试：

  播放测试音频，帮助用户确认音频输出设备是否正常工作。

```
// 开始音频播放设备测试
await client.startAudioPlaybackDeviceTest();

// 停止音频播放设备测试
await client.stopAudioPlaybackDeviceTest();
```

- 打印控制台日志：

  在调试模式下，`RealtimeClient` 会输出详细的日志信息，日志以 `[RealtimeClient]` 开头，后面跟着事件名称和相关数据，帮助开发者了解客户端和服务端的交互情况。

```
[RealtimeClient] on realtime.event event
[RealtimeClient] on realtime.event event
```

```
[RealtimeClient] dispatch server.bot.join event
[RealtimeClient] dispatch server.session.created event
[RealtimeClient] dispatch server.conversation.created event
[RealtimeClient] dispatch client.connected event
[RealtimeClient] dispatch server.audio.user.speech_started event
```

噪声抑制

扣子 Realtime API 提供了两种噪声抑制模式，即静态噪声抑制和非静态噪声抑制。这两种模式可以同时启用，系统会根据具体场景自动选择最佳的抑制策略。

静态噪声抑制技术旨在通过多种技术手段降低或消除那些在一定时间段内保持恒定水平的噪声，这类噪声通常被称作静态噪声，例如持续的机器运转声或稳定的道路交通背景声。该技术的主要目标是增强信号的清晰度，特别是在语音通信、音频编辑和语音识别等场景中，通过减少背景噪声来提高语音信号与噪声的比例（信噪比），从而提升语音的清晰度和可理解性。

- 开启静态噪声抑制

  ```
  const client = new RealtimeClient({
    // ... 其他配置
    suppressStationaryNoise: true,  // 启用静态噪声抑制
  });
  ```

- 开启非静态噪声抑制

  ```
  const client = new RealtimeClient({
    // ... 其他配置
    suppressNonStationaryNoise: true,  // 启用非静态噪声抑制
  });
  ```

音色配置

系统默认采用 `柔美女友` 音色，音色 ID 为 7426720361733046281。在初始化 Realtime SDK 时，你可以通过指定 `voiceId` 来自定义智能体的音色。

说明

- 每个音色都有不同的特征，例如性别、语言、风格等特征。
- 如果不指定 `voiceId` 或值为空，智能体将使用系统默认音色。

配置方式如下：

1. 获取可用的音色 ID。

   你可以通过查看音色列表获取可用的音色列表。

   ```
   import { CozeAPI, COZE_CN_BASE_URL } from '@coze/api';
   const api = new CozeAPI({
       token: 'your-access-token',
       baseURL: COZE_CN_BASE_URL,
   ```

```javascript
});
// 获取可用的音色列表
const voices = await api.audio.voices.list();
```

2. 在 Realtime SDK 中使用音色 ID。

   获取到想要使用的音色 ID 后，你可以在初始化 Realtime 客户端时指定 `voiceId`，以使用这个音色。

```javascript
import { RealtimeClient } from "@coze/realtime-api";

const client = new RealtimeClient({
  accessToken: 'your-access-token',
  botId: 'your-bot-id',
  connectorId: '1024',
  voiceId: '7426725529589661723', // 你想使用的音色 ID
  // ... 其他配置项
});
```

信令事件

信令事件指的是在通话过程中，硬件端和服务端通过信令通道交互的事件，例如硬件端可以通过事件去设置自己的降噪模型，服务端会通过信令事件发送对话文字内容等。通过信令事件可在硬件设备上实现丰富的交互效果，例如获取设备端的数据、解析超链接或图片内容、控制设备去播放音乐、动态调整音色、设置智能体说话速度等。

> **说明**
>
> - 事件 ID：每个上行事件 ID 建议不要重复，故障排查场景下便于定位问题。
> - 事件顺序：确保在监听到 Realtime SDK 的 onUserJoined 回调智能体进房后再发送上行事件。
> - 插件模式：根据需求选择 `blocking` 或 `nonblocking` 模式，控制插件执行是否阻塞对话。
> - 信令事件需要发送给房间内的智能体，传入的 UID 就是建房的 BotID，广播是不生效的。
> - 每个事件有 ID 和 EventType， 通过 EventType 可以区分具体的事件类型，每个事件类型对应的 Payload 在 Data 中，开发者可以按需去提取需要的内容。

事件类型

智能语音信令事件包括上行事件和下行事件。每个事件有 ID 和 EventType， 通过 EventType 可以区分具体的事件类型，每个事件类型对应的 Payload 在 Data 中，开发者可以按需去提取需要的内容。

- **上行事件**：设备端上报给服务端的事件。应用程序需要根据扣子编程提供的事件结构，在触发事件时填充字段内容并上报事件。设备端需要在监听到 Realtime SDK 的 onUserJoined 回调智能体进房后才能发送上行事件，可通过 Realtime SDK 的 `sendUserMessage` 发送，如果是嵌入式设备集成音视频，可通过 Realtime SDK 的 `byte_rtc_rts_send_message` 发送。详细信息可参考Realtime 上行事件。
- **下行事件**：服务端下发给设备端的事件，可订阅 Realtime SDK的

`onUserMessageReceived` 回调，如果是嵌入式设备集成音视频，可订阅 Realtime SDK 的 `on_message_received` 回调，接收下行事件。应用程序需要解析下行事件，并根据业务需求进行下一步操作。详细信息可参考Realtime 下行事件。

公共参数

智能语音信令事件的公共参数如下：

参数名称	**类型**	**描述**
id	String	事件 ID，也就是事件的唯一标识。由客户端或服务端生成， 下用于定位具体的事件，便于排查问题。
event_type	String	事件的类型。
data	JSON	事件的详细信息，其中包含具体事件的业务字段。

信令事件的使用流程

信令事件的使用流程如下：

1. 初始化。通过监听 `bot.join` 事件，确认房间初始化完成。
2. 发送请求。使用上行事件（如 `conversation.message.create`）向智能体发送消息。
3. 处理响应。监听下行事件（如 `conversation.message.delta`）获取智能体的增量回复。
4. 插件交互。在收到事件 `conversation.chat.requires_action` 后，执行插件操作并通过事件 `conversation.chat.submit_tool_outputs` 提交结果。
5. 错误处理。通过监听 `error` 事件捕获和处理异常情况。

使用示例如下：

```
// 使用示例
client.on(EventNames.ALL_SERVER, (eventName, event: any) => {
  if (eventName === 'server.bot.join') { // 这里需要加个 server. 前缀
    client.sendMessage({
      "id": "",
      "event_type": "conversation.message.create",
      "data": {
        "role": "user",
        "content_type": "text",
        "content": "你好"
      }
    });
  } else if (eventName === 'server.conversation.message.delta') {
    console.log('delta', event);
  }
});
```

信令事件的处理思路

在处理信令事件时，建议采用以下思路：

- **监听所有事件**：捕获客户端和服务端的所有交互，便于调试和监控。扣子编程推荐你监听所有的事件，你也可以根据实际的业务需求，监听不同类型的事件，例如客户端事件、服务端事件等。
- **处理特定事件**：根据业务需求，监听和处理特定的事件，例如连接成功、中断、断开、音频状态变化等。
- **错误处理**：捕获和处理错误事件，确保应用的稳定性和用户体验。

以下是支持的事件类型及其说明：

> **说明**
>
> 事件的详细说明，可参考示例项目源码。

```
// 事件监听示例
client.on(EventNames.ALL, (eventName, data) => {
  console.log(`收到事件: ${eventName}`, data);
});

// 支持的事件类型
enum EventNames {
  ALL = 'realtime.event',            // 所有事件
  ALL_CLIENT = 'client.*',           // 所有客户端事件
  ALL_SERVER = 'server.*',           // 所有服务端事件
  CONNECTED = 'client.connected',    // 客户端已连接
  INTERRUPTED = 'client.interrupted', // 客户端已中断
  DISCONNECTED = 'client.disconnected', // 客户端已断开
  AUDIO_UNMUTED = 'client.audio.unmuted', // 音频已取消静音
  AUDIO_MUTED = 'client.audio.muted',     // 音频已静音
  ERROR = 'client.error'             // 客户端发生错误
}
```

错误处理

系统定义了一个错误处理机制，用于处理 `RealtimeClient` 在运行过程中可能遇到的各种错误。通过定义一个枚举 `RealtimeError` ，系统将不同的错误类型进行了分类，便于开发者根据具体的错误类型进行针对性的处理。

```
enum RealtimeError {
  DEVICE_ACCESS_ERROR = 'DEVICE_ACCESS_ERROR',         // 设备访问错误
  CONNECTION_ERROR = 'CONNECTION_ERROR',               // 连接错误
  DISCONNECTION_ERROR = 'DISCONNECTION_ERROR',         // 断开连接错误
  INTERRUPT_ERROR = 'INTERRUPT_ERROR',                 // 中断错误
  EVENT_HANDLER_ERROR = 'EVENT_HANDLER_ERROR',         // 事件处理错误
  NETWORK_ERROR = 'NETWORK_ERROR',                     // 网络错误
```

```
  CREATE_ROOM_ERROR = 'CREATE_ROOM_ERROR'           // 创建房间错误
}
```

以下是一个完整的示例，展示如何在实际应用中使用错误处理机制：

```
import { AuthenticationError } from '@coze/api';

try {
  await client.connect();
} catch (error) {
      console.error(error);
      if (error instanceof RealtimeAPIError) {
        switch (error.code) {
          case RealtimeError.CREATE_ROOM_ERROR:
            console.log(`创建房间失败: ${error.message}`);
            break;
          case RealtimeError.CONNECTION_ERROR:
            console.log(`加入房间失败: ${error.message}`);
            break;
          case RealtimeError.DEVICE_ACCESS_ERROR:
            console.log(`获取设备失败: ${error.message}`);
            break;
          default:
            console.log(`发生错误: ${error.message}`);
        }
      } else {
        console.error('未知错误：', error);
      }
}
```

上一篇
接入流程

下一篇
集成音视频 Realtime iOS SDK