【视频客服–访客】Juphoon SDK for WeChat 开发集成指南

Juphoon

开发指导手册（WeChat）

宁波菊风系统软件有限公司

2023年1月

版本	作者	日期	说明
JRTC v2203.0	刘正治	2022.08	初版
v2301.0	刘正治	2023.01	增加在线消息接口说明

一、 系统概述

非常感谢您使用菊风系统软件的产品，我们将为您提供最好的服务。本手册可能包含技术上不准确的地方或排版错误。本手册的内容将做定期的更新，恕不另行通知；更新的内容将会在本手册的新版本中加入。我们随时会改进或更新本手册中描述的产品或程序。

1.1 系统介绍

菊风视频能力平台在实际的项目中定位为音视频能力的提供方，除此之外还包装了一些和音视频通讯强相关的业务。以银行项目为例可分为视频客服业务、视频会议业务、视频双录业务、AI双录业务、一对一通话及消息业务等。上述业务

需要客户渠道类系统或者客户业务类系统集成我们 Juphoon RTC SDK 或者插件才能形成完整的业务，在整个完整的业务中我们提供基础的音视频通讯能力和一些对应业务上所需的特色能力。如视频客服业务的智能排队服务，视频会议业务的增强会控服务等。

菊风视频能力平台提供标准 Juphoon RTC SDK 用于给客户渠道类系统和客户业务类系统集成并通过 Juphoon RTC SDK 接入到视频能力平台进行音视频通讯。

菊风为开发者提供 JRTC SDK 功能开发包，涵盖了音视频引擎终端、服务器和业务模块，支持实现智能排队、全景录像、多人音视频等业务功能。

Juphoon RTC SDK支持 iOS、Android、Windows、微信小程序、H5 等操作系统平台。对于银行的其他公共平台或其他第三方平台，视频能力平台可提供标准第三方接口和其他平台进行对接。实现和银行环境的整体融入。

1.2 系统特性

菊风视频能力平台（Juphoon Video Capability Platform）提供高可用、高品质、超低延时的实时音视频通信服务，为远程银行、视频双录、视频会议、AI 双录、VoLTE 视频通话等泛金融场景化方案提供平台支撑。具有业界领先的实时音视频编码技术，以及抗啸叫降噪、ARS 码率自适应、SPo 视频甜点、智能路由等技术，应对网络质量非均衡性、网络异构性、多类型终端的接入的挑战，保证高音质、高画质。Juphoon RTC for Android SDK 专为 Android 平台设计，支持 arm、arm64-v8a 等常见 Android 平台的 CPU 架构。整个平台由宁波菊风系统软件有限公司独立研发，具有自主知识产权。

二、 关于菊风软件

宁波菊风系统软件有限公司（简称“菊风 ”，英文简称“Juphoon”）成立于2005年，现有员工200余 人，注册资金2050万元，总部位于宁波，在北京、广州、长沙设有区域中心（研发、销售和交付），在郑州和杭州设有交付中心，是一家提供实时音视频通信和RCS融合通信解决方案的供应商。宁波总部研发中心主要负责客户端SDK 和 APP、音视频引擎、服务器等产品的研发；云平台和服务器系统的运维、网管等支撑系统研发，现中心成员有180名。

菊风经过15年+音视频底层技术积累，为众多行业合作伙伴提供了超优音视频通信服务。凭借卓越的产品以及优质的服务，迄今为止，已有数十亿终端用户以及众多企业用户通过菊风云实现了音视频场景化沟通，涉及社交、教育、医疗、智能硬件、金融、电商等多个行业领域，为其提供了有针对性的行业化解决方案。

菊风为开发者提供的优而小的 SDK 极简接入，快速助其实现实时音视频通信能力。基于客户不同需求，菊风云提供灵活的部署模式——公有云，专有云，私有云，海外云以及混合云。对主流系统平台全覆盖，支持Windows、Android、iOS、macOS、Web-OCX、H5-WebRTC、微信小程序等。支持各移动设备（电脑、手机、平板）、VTM 机等多终端设备的适配。

2.1 技术支持

在您使用 Juphoon RTC SDK 的过程中，遇到任何困难，请与我们联系，我们将热忱为您提供帮助。

您可以通过如下方式与我们取得联系：

公司官网：https://rtc.juphoon.com

产品咨询：sales@juphoon.com

加急热线：13056832331

咨询电话：400-800-8708 / 0574-87901227

售前工程师微信二维码：

2.2 版权申明

“ Juphoon RTC for Android SDK ”是由宁波菊风系统软件有限公司开发，拥有自主知识产权（软著正式编号2020SR0369466号）的系统平台，宁波菊风系统软件有限公司拥有与本产品所用技术相关的知识产权。这些知识产权包括但不限于一项或多项发明专利或者正在进行申请的专利（ZL202010288867.7 、ZL201911393580.4 ）。

本产品发行所依照的许可协议限制其使用、复制分发和反编译。未经宁波菊风系统软件有限公司事先书面授权，不得以任何形式或借助任何手段复制本产品的任何部分。随本 SDK 一同发布的 Demo 演示程序源代码版权归宁波菊风系统软件有限公司。Juphoon 是宁波菊风系统软件有限公司的商标。

三、快速集成 SDK

本文为您介绍了 Wechat 端集成 SDK 的操作步骤，帮助您快速集成 SDK 并实现视频客服（访客端）的基本功能。

操作步骤

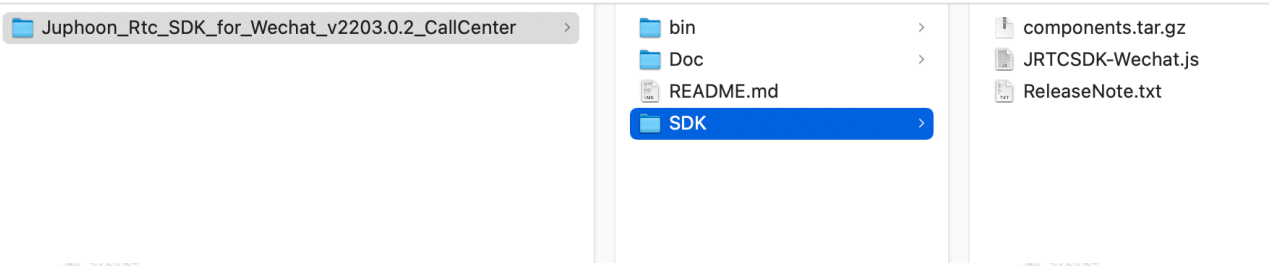
步骤一：获取 Juphoon_Rtc_SDK_for_Wechat

您可在 Juphoon 的产品官方网站下载到最新版的 Juphoon RTC SDK ，

访问下载地址，示例如下：

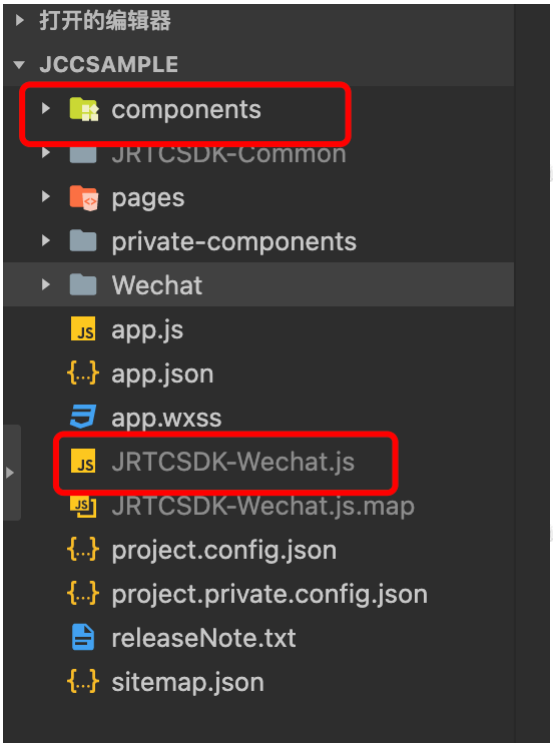
注：首次访问，请先注册后登录。

Juphoon_Rtc_SDK_for_Wechat_版本号_CallCenter 包里面提供了所有支持开发语言 demo 程序的编译程序、开发指南、demo 程序源码和 SDK 文件，其解压之后的目录结构如下所示：



步骤二：导入 SDK

1、拷贝 SDK 文件夹内的 JRTCSDK-Wechat.js 到您的工程目录中及将解压 components 出来的文件夹拷贝到程序上。如下图所求：



步骤三：导入工程需要使用到的相关模块

```
var JRTCSDK = require(”../../JRTCSDK-Wechat.js”);

const {
  JRTCClient,
  JRTCClientInitParam,
  JRTCClientLoginParam,
  ClientState,
  JRTCMediaDevice,
  RenderType,
  JRTCGuest,
  JRTCCallCenterInitParam,
  JRTCCallCenterCallParam,
  GuestCallStateChangeType,
  JRTCTracking
} = JRTCSDK;
```

步骤四：引用视频组件

在需展示视频画面的 json 文件中添加视频组件

步骤五：编译运行

以上步骤进行完后，编译工程，如果没有报错，恭喜您，您已经成功配置 SDK ，可以进行下一步了。

四、实现视频通话（访客）

本文档为您展示通过 SDK 实现视频通过（访客）的相关步骤，帮助您在远程银行和视频客服的场景下实现智能排队、屏幕共享、全景录像、访客管理的相关能力。

4.1 前提条件

请确认您已完成以下操作：

- 已获取 App Key

AppKey 作为同个环境的分域依据，同一个域的终端才能实现互通，AppKey 由 Juphoon 视频平台提供。

- 集成 SDK

4.2 快速跑通 Sample

1、在 Juphoon RTC SDK 文档中心，选择视频客服-访客选择项，在 SDK 下载页面下载体验 Sample 示例项目。

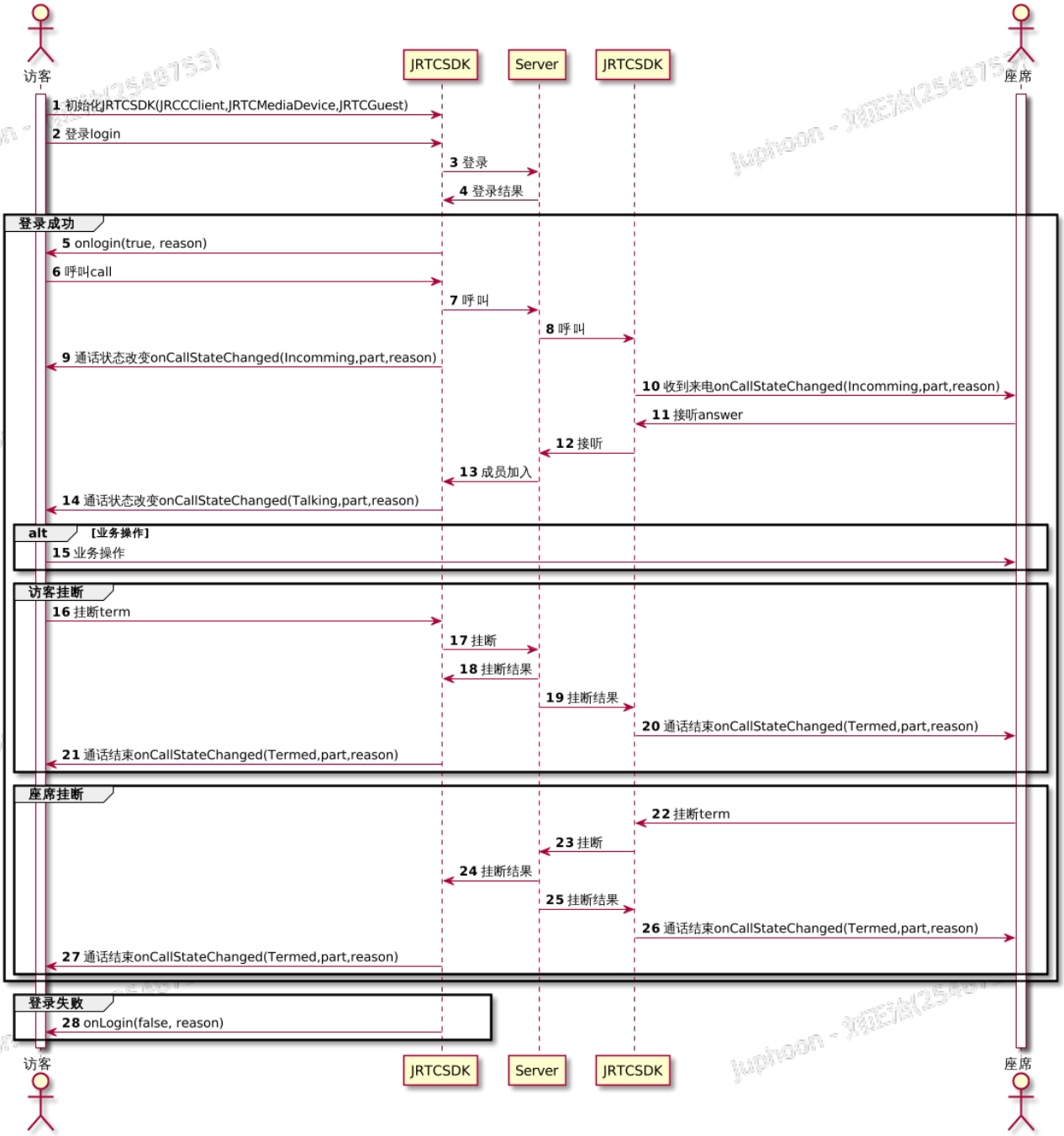
访问下载地址，示例如下：

2、下载完成后，解压出 JCCSample 。

3、使用微信开发者工具导入项目，如下图所求：

4、编译项目运行，如下图所求：

4.3 实现视频通话



4.3.1 初始化

在使用业务接口前，需对 Juphoon RTC SDK 进行初始化操作。

JRTCClient	基础模块	负责视频平台的登录登出，只有登录到视频平台才可以使用视频相关的业务，AppKey 作为同个环境的分域依据，同一个域的终端才能实现互通。
JRTCMediaDevice	媒体模块	负责本地的媒体设备操作，视频画面渲染等功能。
JRTCGuest	访客模块	负责访客的视频业务的处理

AppKey 作为同个环境的分域依据，同一个域的终端才能实现互通，AppKey 由 Juphoon 视频平台提供。

在使用业务接口前，需对 Juphoon SDK 进行初始化操作。

初始化参数详见 JRTCClientInitParam

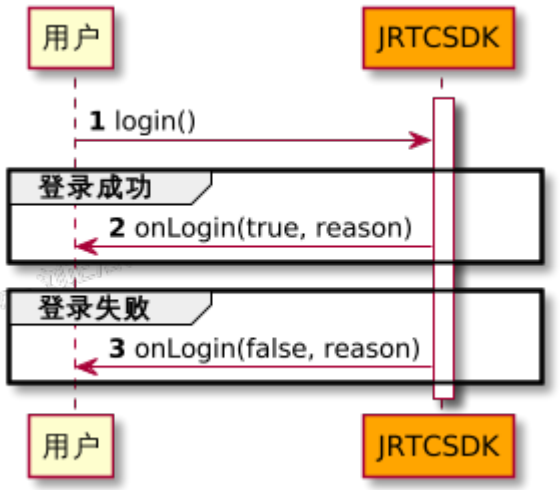
示例代码：

```
const clientInitParam = new JRTCClientInitParam();
clientInitParam.appName = "JCCSample-Wechat";
clientInitParam.server = "server";
clientInitParam.appKey = "appKey";
clientState = ClientState.IDLE;
initClientCallback();
client = JRTCClient.create(clientCallback, clientInitParam);

initMediaDeviceCallback();
mediaDevice = JRTCMediaDevice.create(mediaDeviceCallback, {});

initGuestCallback();
let initParam = new JRTCCallCenterInitParam();
initParam.roomServer = "roomServer";
initParam.protocol = 'RTMP';
guest = JRTCGuest.create(client, mediaDevice, initParam, guestCallback);

/**
*       增加JRTCClientCallback监听回调
*/
initClientCallback() {
  clientCallback = {
    onLogin: (result, reason) => {
    },
    onLogout: (reason) => {
    },
    onClientStateChanged: (state, oldState) => {
    },
    onSDKEvent: (event) => {
    }
  }
},

/**
*       增加JRTCMediaDeviceCallback监听回调
*/
initMediaDeviceCallback() {
  mediaDeviceCallback = {
    onCameraUpdate: () => {
    },
    onVolumeChanged: (volume, userId) => {
    }
  }
},

/**
*       增加JRTCGuestCallback监听回调
*/
initGuestCallback() {
  guestCallback = {
```

```
    onGetAllGroups: (result, groups) => {
    },
    onCallStateChanged: (type, incomingType, inviter, reason) => {
    },
    onCallQueueCount: (count, time) => {
    },
    onCallPropertyChanged: (propChangeParam) => {
    },
    onMemberJoin: (participant) => {
    },
    onMemberLeave: (participant) => {
    },
    onMemberUpdate: (participant, changeParam) => {
    },
    onUrgentResultResponse: (agree) => {
    },
    onHoldStateChanged: (hold) => {
    },
    onCallTypeChanged: (callType) => {
    },
    onMessageReceived: (content, contentType, messageType, fromUserId) => {
    },
    onOnewayVideoChanged: (turnOn) => {
    },
    onDeliveryAbort: (isShutDown, deliveryUri, reason) => {
    },
    onCallForwarding: () => {
    },
    onNotifyMessageReceived: (notifyMessage, fromUserId) => {
    },
    onSignRequest: (userId, extraInfo) => {
    }
  }
},
```

4.3.2 登录

SDK 初始化之后，即可进行登录的集成，登录接口调用流程如下所示：



登录到 Juphoon 视频平台主要调用的是 JRTCClient 的登录接口 login

```
/**
 * 登录 Juphoon RTC 平台，只有登录成功后才能进行平台上的各种业务
 *
 * 登录结果通过 {@link JRTCClientCallback.onLogin} 回调通知
 *
 * @param userId            用户ID
 * @param password          密码，不能为空
 * @param clientLoginParam  登录参数，一般不需要设置，不设置则按默认值
 * @returns 接口调用结果
 * – true: 接口调用成功
 * – false: 接口调用异常
 * @note 目前只支持免鉴权模式，服务器不校验账号密码，免鉴权模式下当账号不存在时会自动去创建该账号
 * @note 用户名为英文数字和'+' '–' '_' '.'，长度不要超过64字符，'–' '_' '.'不能作为第一个字符
 */
public login(userId: string, password: string, clientLoginParam?: JRTCClientLoginParam): boolean;
```

JRTCClientLoginParam 参数介绍

token	token
tokenType	token校验类型

token认证服务，主要用于登录时 token 验证，由第三方服务获取 token，将 token 下发给集成的终端，由 SDK 发起登录时带上 token ，进行认证。详见 token 流程介绍。

允许用户登录时，带入 token 。如果未使用，可以不带。

登录的结果将会通过 JRTCClientCallback 中的 onLogin 接口上报。

```
/**
 * 登录结果回调
 *
 * @param result true 表示登录成功，false 表示登录失败
 * @param reason 登录失败原因，当 result 为 false 时该值有效
 */
onLogin(result: boolean, reason: ReasonCode): void;
```

示例代码：

```
// 创建登录配置参数
let loginParam = new JRTCClientLoginParam();
loginParam.token = "token";
loginParam.tokenType = "tokenType";
client.setServer("server");
client.setAppKey("appKey");
// 登录
client.login("userId", "123456", loginParam);

/**
* 调用登录接口返回回调消息
*/
onLogin: (result, reason) => {
  if (result) {
    //登录成功
  } else {
```

```
    //登录失败
  }
},
```

4.3.3 获取业务号列表

在发起呼叫前，可以先调用 queryAllGroups 接口获取业务号、业务描述等详细信息；业务号一般由业务管理人员在业务管理平台上配置业务，然后将业务号给开发人员，或者也可以通过该接口查询到所有的业务号列表进行业务。

```
/**
 * 获取业务号列表
 *
 * @returns 接口调用结果
 * – true: 接口调用成功，返回结果通过 {@link JRTCGuestCallback.onGetAllGroups} 回调上报
 * – false: 接口调用异常
 */
public queryAllGroups(): boolean {}
```

获取结果通过实现 JRTCGuestCallback 中的 onGetAllGroups 接口获取。

```
/**
 * 获取业务号列表结果回调
 *
 * 访客调用 {@link JRTCGuest.queryAllGroups} 接口获取业务号列表，会收到此回调。
 * @param groups 座席业务实体对象列表，获取失败时为 undefined
 * @param result 获取结果，true 表示获取成功，false 表示获取失败
 */
onGetAllGroups(result: boolean, groups: Array<JRTCCallCenterGroupItem>): void;
```

示例代码：

```
//获取业务号列表
guest.queryAllGroups();
/**
* 获取业务号列表回调消息
*/
onGetAllGroups: (result, groups)=> {
    if(result) {
        //获取成功
    } else {
        //获取失败
    }
},
```

4.3.4 发起呼叫

在登录成功之后，访客就可以通过发起呼叫的接口来呼叫自己要做的业务；也可以发起呼叫的同时，设置呼叫参数；JRTCCallCenterCallParam 包含了很多呼叫参数，比如随路参数、SVC 等。

访客有两个呼叫方法，分别为 call 和 oneToOneCall 如下：

```
/**
 * 呼叫指定业务
 *
```

```
 * @param number 业务号，如10087，一般由业务管理人员在业务管理平台上配置业务，然后将业务号给开发人员
 * @param param 呼叫参数设置，可以设置通话分辨率、全局宽高比等参数，此参数不传则使用默认配置
 * @returns 接口调用结果
 * – true: 接口调用成功，通话状态会通过 {@link JRTCGuestCallback.onCallStateChanged} 回调上报
 * – false: 接口调用异常
 */
public call(number: string, param?: JRTCCallCenterCallParam): boolean;

/**
 * 呼叫指定座席
 *
 * @param userId 座席 id，如agent1，一般由业务管理人员在业务管理平台上配置座席id，然后将座席id给开发人员
 * @param param 呼叫参数设置，可以设置通话分辨率、全局宽高比等参数，此参数不传则使用默认配置
 * @returns 接口调用结果
 * – true: 接口调用成功，通话状态会通过 {@link JRTCGuestCallback.onCallStateChanged} 回调上报
 * – false: 接口调用异常
 */
public oneToOneCall(userId:string, param?: JRTCCallCenterCallParam): boolean;
```

示例代码：

```
// 创建呼叫配置参数
let callParam = new JRTCCallCenterCallParam();
//呼叫随路参数
callParam.extraInfo = ”extraInfo”;
// 呼叫指定业务
guest.call(”10086”,callParam);
// 呼叫指定座席
guest.oneToOneCall(”agent1”,callParam);
```

4.3.5 通话状态改变通知

当访客发起呼叫、通话建立或者通话结束，都会通过 JRTCGuestCallback 里的 onCallStateChanged 接口进行上报。

```
/**
 * 通话状态改变回调
 *
 * @param type 访客通话状态改变类型，即以下情况会收到此回调：
 * – {@link GuestCallStateChangeType#CALLING} 访客呼叫成功
 * – {@link GuestCallStateChangeType#WAITING} 访客呼叫成功后等待座席接听
 * – {@link GuestCallStateChangeType#INCOMING} 作为第三方访客收到通话邀请
 * – {@link GuestCallStateChangeType#TALKING} 通话接通（第三方访客）或被接通（主访客）
 * – {@link GuestCallStateChangeType#TERMED} 通话挂断或被挂断
 * @param incomingType 来电类型，当 type == {@link GuestCallStateChangeType#INCOMING} 时有效
 * @param inviter 邀请成员对象，当 type == {@link GuestCallStateChangeType#INCOMING} 时有效
 * @param reason 挂断原因，只在 type 为 {@link GuestCallStateChangeType#TERMED} 时需要关注
 */
onCallStateChanged(type: GuestCallStateChangeType, incomingType: CallIncomingType, inviter: JRTCInviter |
undefined, reason: CallTermReason): void;
```

访客通话状态详见 GuestCallStateChangeType

通话结束原因详见 CallTermReason

示例代码：

```
onCallStateChanged: (type, incomingType, inviter, reason)=> {
  switch (type) {
    case GuestCallStateChangeType.CALLING:
      //呼叫成功
      break;
    case GuestCallStateChangeType.TALKING:
      //通话建立
      break;
    case GuestCallStateChangeType.TERMED:
      //通话挂断
      break;
  }
},
```

4.3.6 创建本地视频画面

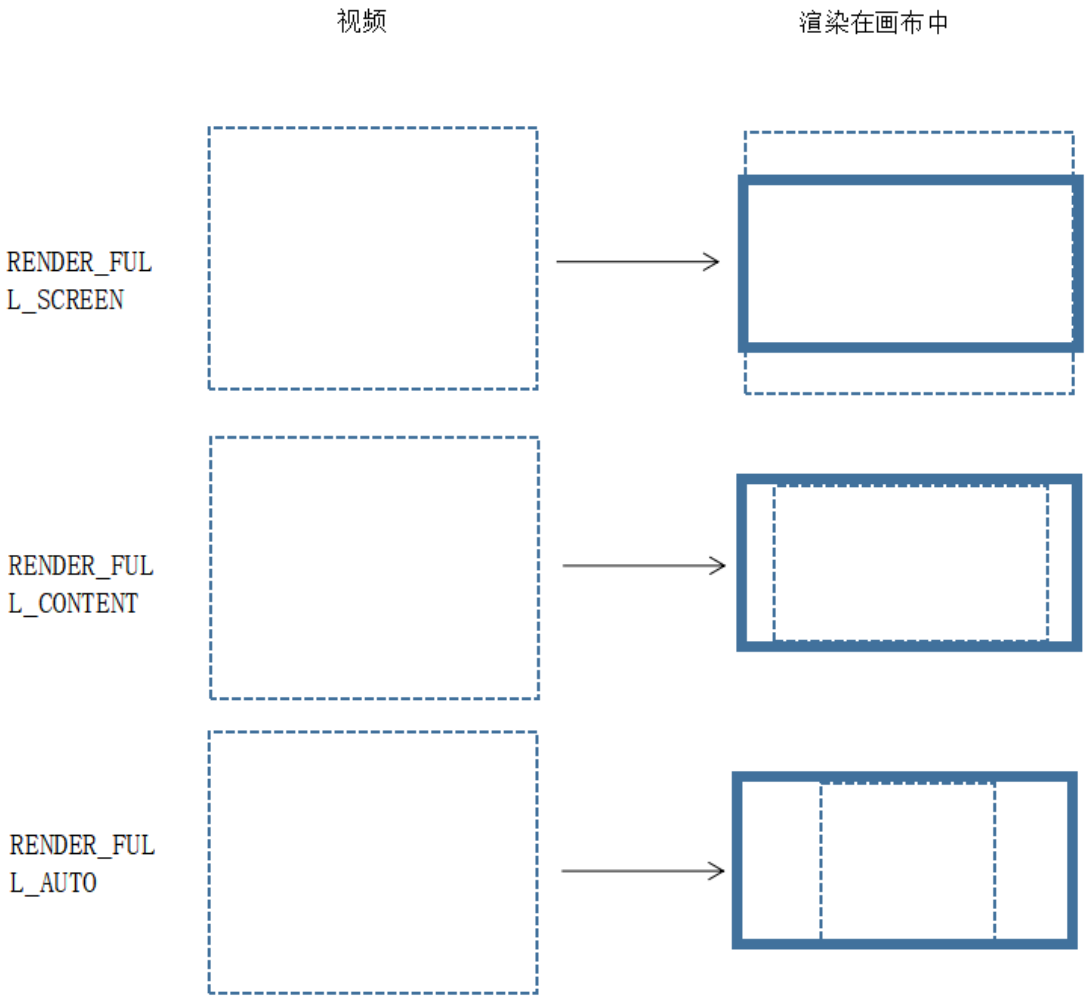
初始化成功后，可以创建本地的视频画面，创建本地视频画面的时机没有具体要求，在通话前通话中皆可。

1. 通过调用 JRTCMediaDevice 里的 startCameraVideo 方法获取本地的视频对象。
2. 通过 startCameraVideo 方法里传入渲染的画布的句柄进行视频画面渲染。

```
/**
 * 开始本端视频渲染
 *
 * 获取本端视频预览对象 JRTCMediaDeviceVideoCanvas，通过此对象能获得视图用于UI显示
 * @param renderType 视频渲染模式
 * @param viewId 本端视频视图组件（local-stream）id
 * @param pageTarget 页面节点
 * @param enableMic 是否打开麦克风，默认打开
 * @returns Promise
 */
public async startCameraVideo(renderType: RenderType, viewId: string, pageTarget: WechatMiniprogram.Component.
TrivialInstance, enableMic?: boolean): Promise<JRTCMediaDeviceVideoCanvas> {}
```

RenderType 决定了视频的渲染模式：

- RENDER_FULL_SCREEN 为填充模式：即将画面内容居中等比缩放以充满整个显示区域，超出显示区域的部分将会被裁剪掉，此模式下画面可能不完整；
- RENDER_FULL_CONTENT 为适应模式：即按画面长边进行缩放以适应显示区域，短边部分会被填充为黑色，此模式下图像完整但可能留有黑边。



示例代码：

```
// 本端视频视图组件（local-stream）ID
let viewId = 'local-stream';
// 传入viewId
mediaDevice.startCameraVideo(RenderType.RENDER_FULL_SCREEN, viewId, this, true)
  .then((canvas) => {});
```

4.3.7 创建远端视频画面

当通话建立后，除了本地的视频画面，还有通话成员的远端视频画面，如果通话成员有视频流的上传，访客端可以获取到座席的视频流并进行渲染。

访客可在收到以下回调时进行界面处理：

- 收到 onCallStateChanged 回调且通话状态改为 GuestCallStateChangeType. TALKING 时，表示座席加入了通话。
- 收到 onMemberJoin 回调时，表示有其他成员加入通话。
- 收到 onMemberUpdate 回调时，表示通话中用户的通话状态变化。例如，可通过 JRTCRoomParticipantChangeParam 的 video 属性判断用户视频流状态是否发送改变；当 JRTCRoomParticipant. isVideo() 的值为 true ，表示远端用户已上传视频流，本端通过订阅远端视频流，获取远端视图渲染对象。

调用 startVideo 接口在界面画布上渲染远端视频对象画面。

```
/**
 * 开始远端视频渲染
 *
 * @param streamUrl 远端视频拉流地址
 * @param renderType 视频渲染模式
 * @param viewId 视频视图组件（remote-stream）id
 * @param pageTarget 页面节点
 * @returns
 * - JCMediaDeviceVideoCanvas 对象: 开始远端视频渲染成功
 * - undefined: 开始远端视频渲染失败
 */
public startVideo(streamUrl: string, renderType: RenderType, viewId: string, pageTarget: WechatMiniprogram.
Component.TrivialInstance): JRTCMediaDeviceVideoCanvas | undefined {}
```

渲染其他用户视图画面的方法法与渲染摄像头画面的用法基本一致，需关注 RenderType 渲染模式。

示例代码：

```
guest.requestVideo(participant, { width: 100, height: 100 }).then((result) => {
        // 视频视图组件（local-stream）ID
  let viewId = 'remote-stream';
        let selector = '.live-player';
        remoteCanvas = mediaDevice.startVideo(result, RenderType.RENDER_FULL_CONTENT, viewId, this, selector);
    });
```

4.3.8 结束通话

访客可以在呼叫等待或者通话中调用 term 接口主动取消或者结束通话。

```
/**
 * 结束通话
 *
 * @note
 * - 主访客调用此接口会结束通话，通话中所有成员都会离开，此通通话销毁，所有成员会收到 {@link JRTCGuestCallback.
onCallStateChanged} 通话结束回调。
 * - 第三方访客调用此接口仅自身离开通话，通话中其他成员会收到该成员离开的回调 {@link JRTCGuestCallback.
onMemberLeave} 回调，通话继续进行。
 * @returns 接口调用结果
 * - true: 接口调用成功，会收到 {@link JRTCGuestCallback.onCallStateChanged} 回调
 * - false: 接口调用异常
 */
public term():boolean;
```

主动和被动的挂断事件与来电、接通一样，都通过 onCallStateChanged 接口上报，其中 CallTermReason 可以用来判断挂断原因，如：主动挂断、对端挂断等，详细可参考 API 文档。

示例代码：

```
// 访客主动结束通话
guest.term();

// 通话状态改变回调
onCallStateChanged: (type, incomingType, inviter, reason) => {
```

```
    if (type == GuestCallStateChangeType.GUEST_CHANGE_TYPE_TERMED) {
            // 通话结束，结束原因查看 reason
    }
}
```

4.3.9 销毁本地和远端画面

当不再需要查看视频画面，包括通话其他成员离开，或者通话结束，需要调用 JRTCMediaDevice 中的 stopVideo 方法来释放渲染的资源；该方法需传入要释放的 JRTCMediaDeviceVideoCanvas 对象；必须进行这步操作，不然会造成渲染内存不释放。

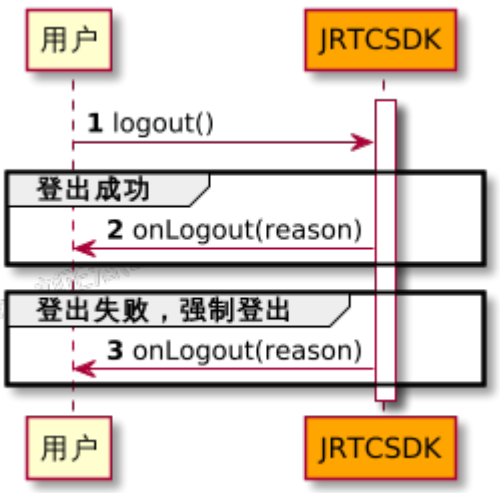
```
/**
 * 停止视频渲染
 *
 * @param canvas JCMediaDeviceVideoCanvas 对象，由 {@link #startVideo} 或 {@link #startCameraVideo} 接口返回
 */
public stopVideo(canvas: JRTCMediaDeviceVideoCanvas): void;
```

示例代码：

```
// 通话结束
onCallStateChanged: (type, incomingType, inviter, reason) => {
        if (type == GuestCallStateChangeType.GUEST_CHANGE_TYPE_TERMED) {
                mediaDevice.stopVideo(localCanvas);
    mediaDevice.stopVideo(remoteCanvas);
        }
}
// 成员离开
onMemberLeave: (participant) => {
        // 停止该成员画面渲染
        mediaDevice.stopVideo(remoteCanvas);
 }
```

4.3.10 登出

访客结束通话后，可以做登出操作；登出接口调用流程如下所示：



访客可以登出视频平台，与平台断开一切连接。

```
/**
 * 登出 Juphoon RTC 平台，登出后不能进行平台上的各种业务
 *
 * 登出结果通过 {@link JRTCClientCallback.onLogout} 回调通知
 * @returns 接口调用结果
 * – true: 接口调用成功
 * – false: 接口调用异常
 */
public logout(): boolean {}
```

登出结果通过 JRTCClientCallback 中的 onLogout 接口上报：

```
/**
 * 登出回调
 *
 * @param reason 登出原因
 */
onLogout(reason: ReasonCode): void;
```

示例代码：

```
// 调用登出接口
client.logout();

// 监听登出结果回调
onLogout: (reason) =>{
        // 登出完成
}
```

4.3.11 登录状态改变通知

登录状态通过 JRTCClientCallback 中的 onClientStateChanged 接口上报。

```
/**
 * 登录状态变化通知
 *
 * @param state    当前状态值
 * @param oldState 之前状态值
 */
onClientStateChanged(state: ClientState, oldState: ClientState): void;
```

登录状态详见 ClientState

示例代码：

```
// 登录状态改变通知
onClientStateChanged: (state, oldState) => {
    // state 当前状态
    // oldState 之前状态
}
```

4.3.12 销毁 SDK

每个模块都有对应的销毁接口。如不需再使用 SDK 的相关功能，可以强制释放 SDK 的资源。

注：该方法为同步调用，调用此方法后，你将无法再使用该模块的其它方法和回调。 我们不建议在 JRTCSDK 的所有回调方法中调用此方法销毁对象，否则可能出现崩溃现象。

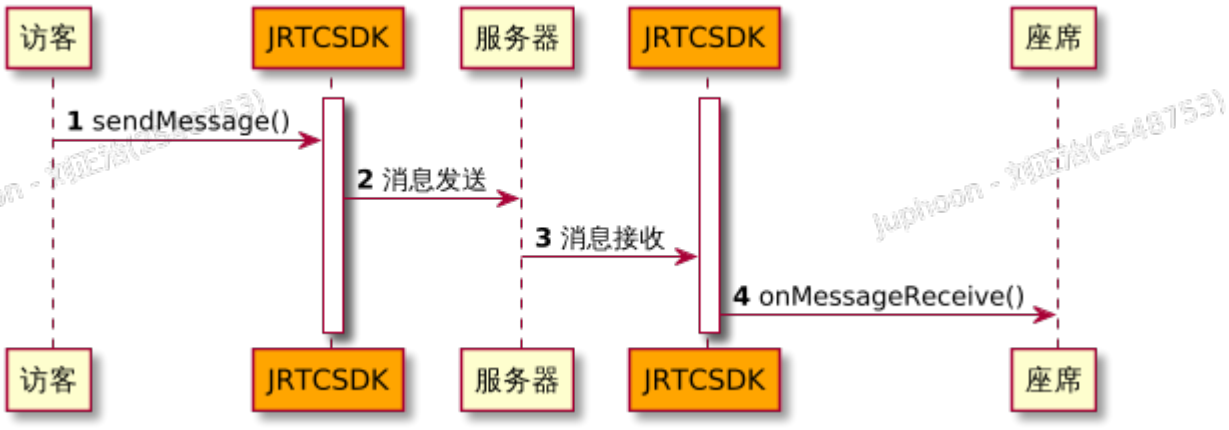
```
JRTCClient.destroy();
JRTCMediaDevice.destroy();
JRTCGuest.destroy();
```

五、消息通道

透明通道消息主要为通话内消息，具体使用要求和场景举例参考下表：

	通话内消息
一对一发送	支持
群发消息	支持
是否支持异步发送结果上报	不支持
是否需要登录	是
是否需要建立通话	是
使用场景举例	1、实现通话中的一些自定义信令、通知等； 2、实现通话中单聊和群聊；
消息内容支持	只支持文本消息
消息内容大小最大支持（bit）	4K

5.1 通话内消息



调用 JRTCGuest 下的 sendMessage 方法

```
/**
 * 发送消息，消息内容不能大于4K
 *
 * 指定成员会收到 {@link JRTCGuestCallback.onMessageReceived} 回调
 * @param contentType 消息内容类型
 * @param content 消息内容
 * @param toUserId 指定成员的用户ID，传 null 给通话中全部成员发送消息
 * @returns 接口调用结果
 * – true: 接口调用成功
 * – false: 接口调用异常
 */
public sendMessage(contentType: string, content: string, toUserId: string):boolean
```

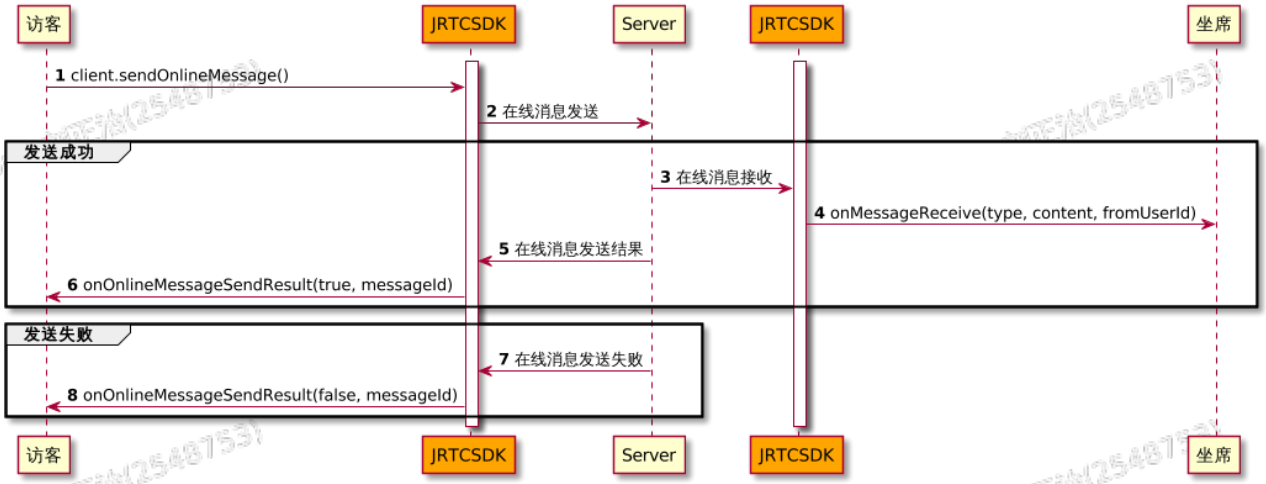
接收消息通过实现 JRTCGuestCallback 中的 onMessageReceived 接口上报。

```
/**
 * 收到消息回调
 *
 * 通话中的访客和座席可分别调用 {@link JRTCGuest.sendMessage} 接口给通话中的指定成员或全体成员发送文本消息，接收消息的成员会收到此回调，由此获取消息具体信息。
 * @param content 消息内容
 * @param contentType 消息内容类型
 * @param messageType 消息归属类型
 * – {@link MessageType#TYPE_1TO1} : 一对一消息
 * – {@link MessageType#TYPE_GROUP} : 群发消息(发送给通话中所有成员)
 * @param fromUserId 发送方的用户ID
 */
onMessageReceived(content: string, contentType: string, messageType: MessageType, fromUserId: string): void;
```

示例代码：

```
// 给成员 agent1 发送消息
guest.sendMessage(”messageType”, ”content”, ”agent1”);
// 给通话中所有成员发送消息
guest.sendMessage(”messageType”, ”content”, null);
// 收到消息
onMessageReceived(content, contentType, messageType, fromUserId) {
    // 收到消息，消息内容为 content，消息来自 fromUserId
}
```

5.2 在线消息



只要登录到 Juphoon RTC 平台就可以通过 JRTCClient 的 sendOnlineMessage 实现在线消息的收发，消息内容不能大于4K。

```
/**
 * 发送在线消息
 *
 * @note 消息大小不超过4k
 * @param message 消息内容
 * @param userId 对端的用户名
 * @returns 接口调用结果
 * – 操作id: 接口调用成功，对应 {@link JRTCClientCallback.onOnlineMessageSendResult} 回调的 operatorId 参数
 * – –1: 接口调用异常，不会收到回调
 */
 public sendOnlineMessage(message: string, userId: string): number
```

在线消息发送结果通过 onOnlineMessageSendResult 回调通知。

```
/**
 * 在线消息发送结果回调
 *
 * @param result     发送结果是否成功
 *                   – true：发送成功
 *                   – false：发送失败
 * @param operatorId 操作id，对应 {@link JRTCClient.sendOnlineMessage} 的返回值
 */
 onOnlineMessageSendResult(result: boolean, operatorId: number): void;
```

在线消息接收者会收到 onOnlineMessageReceived 回调通知。

```
/**
 * 收到在线消息回调
 *
 * @param message 消息内容
 * @param userId  对方用户ID
 */
 onOnlineMessageReceived(message: string, userId: string): void;
```

示例代码：

```
// 给成员 agent1 发送在线消息
client.sendOnlineMessage(”content”, ”agent1”);
```

```
// 在线消息发送结果回调
onOnlineMessageSendResult(result,operatorId) {}
// 收到消息
onMessageReceived(content, fromUserId) {
// 收到消息，消息内容为 content，消息来自 fromUserId
}
```

六、智能排队

Juphoon 提供的智能排队服务是纯软件解决方案，相较于传统的排队机，智能排队服务轻量化的纯软件灵活部署模式、完美兼容行方现有的集中作业平台、呼叫中心等系统。

智能排队服务是在座席管理后台里通过对业务组、技能组和座席进行灵活配置，将座席资源按照实际的业务所需进行分组，再通过智能调度中心按照不同调度策略结合当前座席资源使用情况，合理的将来自访客终端的呼叫请求分配至相应的座席。实现大流量访客的合理分流，降低排队流失，优化顾客体验。

6.1 排队人数与预计等待时长

在呼叫发起尚未接通的时间段每隔一定时间上报一次，结果通过 JRTCGuestCallback 里的 onCallQueueCount 接口进行上报：

```
/**
 * 当前排队人数上报回调
 *
 * 在呼叫发起尚未接通的时间段每隔一定时间上报一次，通话接通后将停止上报。
 * @param count 当前排队人数
 * @param time 预计等待时长，单位秒
 */
onCallQueueCount(count: number, time: number): void;
```

6.2 请求加急

访客呼叫时 requestUrgent 申请加急优先进行通话：

注：只有管理员权限的座席才能收到加急请求，可以在业务管理平台上配置座席权限。

```
/**
 * 请求加急
 * 请求加急流程: <br>
 * 1. 访客在排队过程中调用此接口发起加急请求 <br>
 * 2. 管理员权限的座席（业务管理平台配置）收到 onUrgentRequest 回调 <br>
 * 3. 座席收到回调后调用 responseUrgent 接口对加急请求进行处理 <br>
 * 4. 座席处理后，访客会收到 {@link JRTCGuestCallback.onUrgentResultResponse} 加急请求处理结果回调，如果座席同意加急请求，则将会插队到队列最前
 * @return 接口调用结果
 * – true: 接口调用成功
 * – false: 接口调用异常
 */
public requestUrgent(): boolean {}
```

座席处理加急请求后，结果通过 JRTCGuestCallback 中的 onUrgentResultResponse 接口上报：

```
/**
 * 座席处理加急的结果回调
 *
 * 访客调用 {@link JRTCGuest.requestUrgent} 接口请求加急后，座席同意或拒绝加急请求，访客会收到此回调获得加急请求应答结果。
 * @param agree 加急是否通过，true 表示座席同意了访客的加急请求，false 表示不同意
 */
onUrgentResultResponse(agree:boolean): void;
```

示例代码：

```
// 访客请求加急
guest.requestUrgent();
// 请求加急处理结果
onUrgentResultResponse:(agree) => {
        if (agree) {
                // 座席同意加急请求
        } else {
                // 座席不同意加急请求
        }
}
```

七、通话操作

本文将介绍访客可基于 Juphoon RTC SDK 提供的通话操作能力实现的功能。

7.1 通话属性变化

通话属性变化通过实现 JRTCGuestCallback 中的 onCallPropertyChanged 接口上报。

```
/**
 * 通话属性改变回调
 *
 * @note
 * 重点关注屏幕共享，即当{@link 多方通话模块!JRTCRoomPropChangeParam.screenShare} 属性为 true 时，去处理屏幕共享相关事件。<br>
 * 可根据 {@link JRTCGuest.getShareStreamId} 和 {@link JRTCGuest.getShareUserId} 方法进行屏幕共享画面的渲染和停止渲染。
 * @param propChangeParam 通话改变的属性
 */
onCallPropertyChanged(propChangeParam: JRTCRoomPropChangeParam): void;
```

通话属性变化详见 JRTCRoomPropChangeParam。

其中对应通话属性的 **boolean** 值，有变化的为 **true**，没有变化为 **false**。

7.2 成员属性变化

成员属性变化通过实现 JRTCGuestCallback 中的 onMemberUpdate 接口上报。

```
/**
 * 通话中成员属性更新回调
 *
 * 常用的有 {@link 多方通话模块!JRTCRoomParticipantChangeParam.audio}、{@link 多方通话模块!
JRTCRoomParticipantChangeParam.video}等。<br>
 * 例如当通话中有成员关闭视频传输，通话中所有成员都会收到此回调。
 * @param participant 属性更新的成员对象
 * @param changeParam 更新的属性对象
 */
onMemberUpdate(participant: JRTCRoomParticipant, changeParam: JRTCRoomParticipantChangeParam): void;
```

代码示例：

```
/**
 * 成员更新
 *
 * @param part        成员
 * @param changeParam 更新的属性
 */
onMemberUpdate: (participant, changeParam) => {
    if (changeParam.video) {
        if (part.video) {
                // 视频流打开
        } else {
                // 视频流关闭
        }
    }
    if (changeParam.audio) {
        if (part.audio) {
                // 音频流打开
        } else {
                // 音频流关闭
        }
    }
}
```

成员属性变化详见 JRTCRoomParticipantChangeParam 。

其中有对应成员属性的 **boolean** 值，有变化的为 **true**，没有变化为 **false** 。

7.3 获取所有通话成员

通过 getParticipants 获取所有参会者成员，JRTCRoomParticipant 类见 API 文档。

```
/**
 * 获取所有成员（包含自己、座席和其他访客）
 */
public getParticipants():Array<JRTCRoomParticipant> {}
```

示例代码：

```
//获取通话中所有成员
guest.getParticipants();
```

7.4 获取座席成员

```
/**
 * 获取主座席成员
 *
 * @returns
 * – 只有在通话中且通话中存在座席成员才能获得座席成员对象，否则为 undefined
 * – 座席成员发起转接成功后，在新的座席对象接听通话前这段时间内，获取的值为 undefined
 */
public getMainAgentParticipant(): JRTCRoomParticipant | undefined {}

/**
 * 获取座席成员列表
 *
 * @returns
 * – 只有在通话中且通话中存在座席成员才能获得含有座席成员对象的数组，返回空数组
 * – 当通话中不存在第三方座席时，数组中仅包含一个主座席成员对象
 * – 当通话中存在第三方座席时，数组中包含主座席和第三方座席成员对象
 * – 当通话中仅存在一个座席时，座席成员发起转接成功后，在新的座席对象接听通话前这段时间内，返回空数组
 */
public getAgentParticipants(): Array<JRTCRoomParticipant> {}
```

示例代码：

```
// 获取主座席对象
let mainAgent = guest.getMainAgentParticipant();

// 获取座席成员列表
let agents = guest.getAgentParticipants();
```

7.5 获取自己的对象

```
/**
 * 获取自己对象
 *
 * @returns 自己对象
 */
public getSelfParticipant(): JRTCRoomParticipant | undefined {}
```

示例代码：

```
//获取自己
let self = guest.getSelfParticipant();
```

八、视频管理

8.1视频属性设置

8.1.1 设置请求分辨率

在通话中修改对端画面的分辨率，这个参数结合访客 SVC 来使用可以实现通话中切换设置的分辨率。

```
/**
 * 获取视频请求尺寸
 *
 * 影响自己看其他成员的视频分辨率
 *
 * @returns 视频请求尺寸
 */
public getRequestSize(): JRTCVideoSize;

/**
 * 设置视频请求尺寸
 *
 * 在渲染画面前设置才有效，建议在通话开始前设置。
 * @param requestSize 视频尺寸大小
 */
public setRequestSize(requestSize: JRTCVideoSize):void;

/**
 * 订阅通话中其他成员的视频流
 *
 * @param participant   成员对象
 * @param videoSize   视频请求的尺寸
 * @returns 接口调用结果
 * – true: 接口调用成功
 * – false: 接口调用异常
 */
public requestVideo(participant: JRTCRoomParticipant, videoSize: JRTCVideoSize): Promise<boolean | string |
MediaStream>;

/**
 * 取消订阅通话中其他成员的视频流
 * @param participant 成员对象
 * @returns 接口调用结果
 * – true: 接口调用成功，会收到 {@link JRTCGuestCallback.onMemberUpdate} 回调，具体可关注 {@link 多方通话模块!
JRTCRoomParticipantChangeParam.videoSize} 属性
 * – false: 接口调用异常
 */
public unRequestVideo(participant: JRTCRoomParticipant): boolean;
```

示例代码：

```
// 订阅视频流
guest.requestVideo(participant, {width: 360, height: 640});

//取消订阅视频流
guest.unRequestVideo(participant);
```

8.1.2 SVC 设置说明

根据实际订阅需求和网络状况动态调整视频发送分辨率是视频通话的特性之一，SVC 可用于设置通话视频的每一层编码分辨率。该参数在发起呼叫时设置，且全局统一。

具体使用详见 SVC 说明。

可在访客发起呼叫时，通过呼叫参数 JRTCCallCenterCallParam 的 svcResolution 属性进行设置，通话全局属性，只有发起呼叫用户设置有效。

```
/**
 * 设置 svc 分辨率，默认为 ”1 180 250 360 600 720 1400”
 *
 * @note 当参数 {@link videoDefinition} 为 {@link 多方通话模块!VideoDefinition#CUSTOM} 时有效
 *
 * 用于自定义分层参数和码率
 *
 * 格式:
 * 高度公约数 第一层高倍数 第一层码率 第二层高倍数 第二层码率 第三层高倍数 第三层码率 第四层高倍数 第四层码率 <br>
 * 说明 <br>
 * 1）默认宽高比16:9 <br>
 * 2）编码宽高最后被裁成16整除 <br>
 * 例如 ”1 180 250 360 600 720 1400” <br>
 * 第一层 分辨率 宽320（180*1/9*16）高 180（180*1）； 码率250kbps <br>
 * 第二层 分辨率 宽640（360*1/9*16）高 360（360*1）； 码率600kbps <br>
 * 第三层 分辨率 宽1280（720*1/9*16）高 720（720*1）； 码率1400kbps <br>
 * 此情况下只有三层，若需要四层，则需补充为 ”1 180 250 360 600 720 1400 1080 1600” <br>
 * 第四层 分辨率 宽1920（1080*1/9*16）高 1080（1080*1）； 码率1600kbps <br>
 */
public set svcResolution(svcResolution:string) {}
```

示例代码：

```
// 创建呼叫配置参数
let callParam = new JRTCCallCenterCallParam();
//配置SVC
callParam.svcResolution = ”1 180 250 360 600 720 1400 1080 1600”;
// 发起呼叫
this.guest.call(”10086”,callParam);
```

8.1.3 设置房间视频清晰度

如果觉得设置 SVC 不好理解，可以直接调用 videoDefinition 来设置通话视频清晰度（一组已经定义的 SVC 和帧率），VideoDefinition 详见 API 文档。

```
/**
 * 设置房间视频清晰度，主要通过修改 {@link svcResolution} 参数调整清晰度，
 * 默认为 {@link 多方通话模块!VideoDefinition.CUSTOM}
 */
public set videoDefinition(videoDefinition:VideoDefinition) {}
```

示例代码：

```
// 创建呼叫配置参数
let callParam = new JRTCCallCenterCallParam();
// 设置通话视频清晰为流畅模式，低帧率
callParam.videoDefinition = VideoDefinition.DEFINITION_FLUENCY_FRAME_LOW;
// 发起呼叫
this.guest.call(”10086”,callParam);
```

8.2 视频渲染管理

8.2.1 渲染视频画面

```
/**
 * 开始本端视频渲染
 *
 * 获取本端视频预览对象 JRTCMediaDeviceVideoCanvas，通过此对象能获得视图用于UI显示
 * @param renderType 视频渲染模式
 * @param viewId 本端视频视图组件（local-stream）id
 * @param pageTarget 页面节点
 * @param enableMic 是否打开麦克风，默认打开
 * @returns Promise
 */
public async startCameraVideo(renderType: RenderType, viewId: string, pageTarget: WechatMiniprogram.Component.
TrivialInstance, enableMic?: boolean): Promise<JRTCMediaDeviceVideoCanvas> {}

/**
 * 开始远端视频渲染
 *
 * @param streamUrl 远端视频拉流地址
 * @param renderType 视频渲染模式
 * @param viewId 视频视图组件（remote-stream）id
 * @param pageTarget 页面节点
 * @returns
 * – JCMediaDeviceVideoCanvas 对象: 开始远端视频渲染成功
 * – undefined: 开始远端视频渲染失败
 */
public startVideo(streamUrl: string, renderType: RenderType, viewId: string, pageTarget: WechatMiniprogram.
Component.TrivialInstance): JRTCMediaDeviceVideoCanvas | undefined {}
```

示例代码：

```
// 本端视频视图组件（local-stream）ID
let viewId = 'local-stream';
// 传入viewId
mediaDevice.startCameraVideo(RenderType.RENDER_FULL_SCREEN, viewId, this, true)
  .then((canvas) => {});

//渲染成员视频画面
guest.requestVideo(participant, { width: 100, height: 100 }).then((result) => {
        // 视频视图组件（local-stream）ID
  let viewId = 'remote-stream';
        let selector = '.live-player';
        remoteCanvas = mediaDevice.startVideo(result, RenderType.RENDER_FULL_CONTENT, viewId, this, selector);
    });
```

8.2.2 停止视频渲染

```
/**
 * 停止视频渲染
 *
```

```
 * @param canvas JCMediaDeviceVideoCanvas 对象，由 {@link #startVideo} 或 {@link #startCameraVideo} 接口返回
 */
public stopVideo(canvas: JRTCMediaDeviceVideoCanvas): void
```

注：开启渲染后必须在不需要时进行关闭，否则会造成内存泄漏。

示例代码：

```
mediaDevice.stopVideo(localCanvas);
```

8.3 视频截图

视频业务存在对当前通话截屏的业务操作，以便于记录用户的操作行为。

```
/**
 * 截图
 *
 * @param type 图片的质量，默认原图。有效值为 raw（原图）、compressed（压缩）
 * @returns Promise
 * 截图成功后返回 base64 编码图片数据
 */
snapshot(type?: string): Promise<string | ArrayBuffer>{}
```

示例代码：

```
localCanvas.snapshot('raw').then(res => {
        const fileManager = wx.getFileSystemManager();
        let imgBase64 = res;
        //将base64数据写入到相册
        var filePath = wx.env.USER_DATA_PATH + '/test.png';
        fileManager.writeFile({
                filePath: filePath,
                data: imgBase64,
                encoding: 'base64',
                success: res => {
                        //保存图片到相册
      wx.saveImageToPhotosAlbum({
              filePath: filePath,
        success: function (data) {
                console.log('图片已保存到相册');
        },
                                fail: function (err) {
                console.log('截图失败',err);
        }
      })
    }
  })
}).catch(e => {
        console.log('截图失败',e);
});
```

九、设备管理

9.1 视频设备管理

在视频场景中，您可能需要根据实际的情况选择视频的采集设备，以及相关的采集参数。

9.1.1 获取摄像头列表

```
/**
 * 获取摄像头列表
 *
 * @returns 摄像头列表
 */
public getCameras(): JRTCMediaDeviceCamera[];
```

示例代码：

```
// 获取所有可用的摄像头列表
mediaDevice.getCameras();
```

9.1.2 摄像头采集属性

```
/**
 * 设置摄像头采集属性
 *
 * 在调用开启摄像头视频预览接口之前设置即可生效
 * @param width     采集宽度，默认为 640
 * @param height    采集高度，默认为 360
 * @param frameRate 采集帧速率，默认为 24
 */
public setCameraProperty(width: number, height: number, frameRate: number): void {}
```

示例代码：

```
// 设置摄像头采集属性
mediaDevice.setCameraProperty(640,360,24);
```

9.1.3 切换摄像头

切换摄像头，内部会根据当前摄像头类型来进行切换

```
/**
 * 切换摄像头
 *
 * @note 内部会根据当前摄像头类型来进行切换
 *
 * – 调用此方法时要保证摄像头已打开，否则将直接返回 false
 * – 设备拥有两个以上摄像头，否则将直接返回 false
 *
 * @returns
 * – true: 切换摄像头成功
 * – false: 切换摄像头失败
```

```
 *
 * @override
 */
public switchCamera(): boolean {}
```