

# 菊风视频能力平台SDK开发集成指南

---

## 一、系统概述

### 1.1 系统介绍

### 1.2 系统特性

## 二、关于菊风软件

### 2.1 技术支持

### 2.2 版权申明

## 三、快速集成 SDK

### 3.1 前提条件

### 3.2 操作步骤

步骤一： 获取 Juphoon\_Rtc\_SDK\_for\_Harmony

步骤二： 导入 SDK

步骤三： 添加权限

关于蓝牙权限

步骤四： 混淆规则

步骤五： 编译运行

## 四、实现视频通话

### 4.1 前提条件

### 4.2 快速跑通 Sample

### 4.3 功能实现

#### 4.3.1 初始化

#### 4.3.2 登录

#### 4.3.3 加入通话

#### 4.3.4 邀请

#### 4.3.5 取消邀请

#### 4.3.6 收到邀请

##### 4.3.6.1 接收邀请

##### 4.3.6.2 拒绝邀请

#### 4.3.7 视频渲染

4.3.8 新成员加入

4.3.9 成员更新

4.3.10 成员离开

4.3.11 结束离开

4.3.12 登出

4.3.13 登录登出状态改变通知

4.3.14 销毁SDK

## 五、通话管理

5.1 设置用户角色

5.2 获取统计信息

5.3 获取通话唯一标识

5.4 获取业务流水号

## 六、消息通道

6.1 通话内消息

6.2 在线消息

## 七、音频管理

7.1 发送本地音频流

7.2 音频输出

7.3 自定义音频输入

7.4 本地音频播放

7.5 音频异常回调

7.6 音频数据回调

7.6.1 输入音频数据回调

7.6.2 输出音频数据回调

## 八、视频管理

8.1 发送本地视频流

8.2 订阅/取消订阅视频流

8.3 SVC 设置说明

8.4 设置本地视频宽高比

8.5 视频截图

8.6 视频采集回调

8.7 视频异常回调

## 九、设备管理

### 9.1 音频设备管理

#### 9.1.1 音频参数设置

#### 9.1.2 扬声器的开启关闭

#### 9.1.3 获取麦克风音量级别

#### 9.1.4 获取扬声器音量级别

#### 9.1.5 获取当前噪声强度

#### 9.1.6 获取当前信噪比强度

#### 9.1.7 设置是否开启自动增益控制

#### 9.1.8 设置开启自适应回音消除

### 9.2 视频设备管理

#### 9.2.1 获取摄像头列表

#### 9.2.2 指定摄像头/指定摄像头采集角度

#### 9.2.3 摄像头采集属性

#### 9.2.4 开启/关闭摄像头

#### 9.2.5 切换摄像头

## 十、体验提升

### 10.1 通话中质量检测

#### 10.1.1 网络质量检测

#### 10.1.2 音频质量检测

#### 10.1.3 剩余可用内存检测

### 10.2 文件上传

## 十一、屏幕共享

### 11.1 开启/关闭屏幕共享

### 11.2 共享视频采集

### 11.3 暂停/恢复屏幕共享

### 11.4 订阅/取消订阅屏幕共享的视频流

### 11.5 渲染共享画面

## 十二、音视频录制

### 12.1 本地录制

### 12.2 本地录制(不需要建立通信)

### 12.3 远程录制

12.3.1 开启/关闭远程录制

12.3.2 远程录制异常回调

12.3.3 水印

12.3.3.1 注意事项

12.3.3.2 添加、修改或删除水印

12.3.4 自定义布局

12.3.5 更新远程录制自定义布局

12.3.6 更新远程录制水印信息

十三、加密传输

13.1 国密加密

13.2 Token 校验

十四、视频多流



## 开发指导手册（Harmony）

宁波菊风系统软件有限公司

2025年1月

版权所有©宁波菊风系统软件有限公司 2025。保留一切权利。



版本	作者	日期	说明
v2401.0	杨象坤	2024.4	• 鸿蒙SDK音频通话集成文档初版
v2501.0	王乐凯	2025.1	• 更新 api 接口链接

## 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、视频房间业务、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 Jphoon 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，视频房间业务的增强会控服务等。

菊风视频能力平台提供标准 Jphoon 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 Jphoon 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。

Jphoon RTC SDK 支持 iOS、Android、鸿蒙Next、Windows、UOS、Kylin、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（Jphoon 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、视频房间、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。Jphoon RTC for Harmony SDK 专为 鸿蒙Next 平台设计，适用于鸿蒙手机等华为公司移动终端设备。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，私有云，海外云以及混合云。对主流系统平台全覆盖，支持 iOS、Android、鸿蒙Next、Windows、UOS、Kylin、微信小程序、H5 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](mailto: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 Harmony 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Harmony 端集成 SDK 的操作步骤，帮助您快速集成 SDK 并实现多方视频通话的基本功能。

### 3.1 前提条件

- HarmonyOS 5.0.0(API 12) Release 及以上
- DevEco Studio 5.0.0 Release (5.0.3.906) 及以上
- Command Line Tools for HarmonyOS 5.0.0 Release (5.0.3.906) 及以上

### 3.2 操作步骤

## 步骤一：获取 Juphoon\_Rtc\_SDK\_for\_Harmony

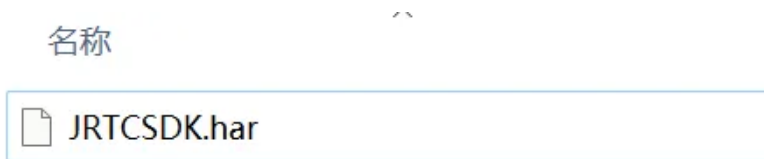
您可在 Juphoon 的产品官方网站下载到最新版的 Juphoon RTC SDK，访问下载地址，示例如下：

注：首次访问，请先注册后登录。

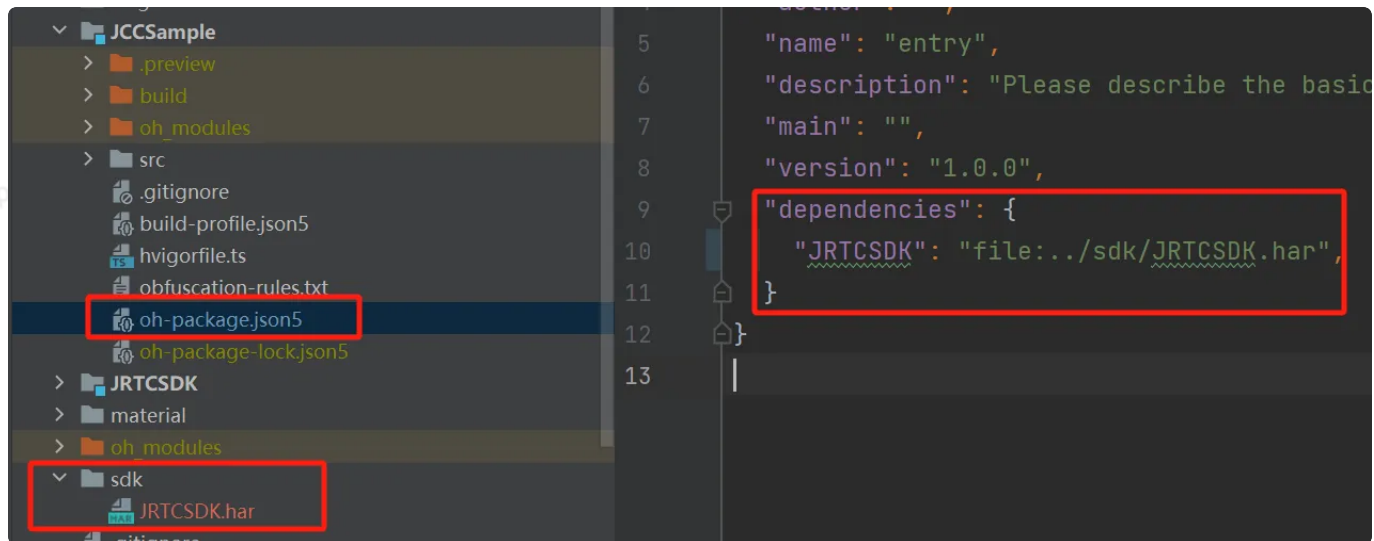
Juphoon\_Rtc\_SDK\_for\_Harmony\_版本号\_CallCenter 包里面提供了所有支持开发语言 demo 程序的编译程序、开发指南、demo 程序源码和 SDK 文件，其解压之后的目录结构如下所示：

## 步骤二：导入 SDK

1. 拷贝 SDK 文件夹内的 JRTCSDK.har 到您工程目录中的 sdk 目录下，并打开工程，如下图所示



2. 为能连接到我们的 so 库，在您工程 oh-package.json5 文件中确保增加以下配置，如图：



## 步骤三：添加权限

根据工程需要，打开 src/main/module.json5 文件，配置权限。

```
1  "requestPermissions": [  
2    {  
3      "name" : "ohos.permission.INTERNET",  
4    },  
5    {  
6      "name" : "ohos.permission.GET_NETWORK_INFO",  
7    },  
8    {  
9      "name" : "ohos.permission.GET_WIFI_INFO",  
10   },  
11   {  
12     "name" : "ohos.permission.MODIFY_AUDIO_SETTINGS",  
13   },  
14   {  
15     "name" : "ohos.permission.USE_BLUETOOTH",  
16   },  
17   {  
18     "name" : "ohos.permission.MICROPHONE",  
19     "reason": "$string:reasonUseMicrophone",  
20     "usedScene": {  
21       "abilities": [  
22         "EntryAbility"  
23       ],  
24       "when": "always"  
25     }  
26   },  
27   {  
28     "name" : "ohos.permission.CAMERA",  
29     "reason": "$string:reasonUseCamera",  
30     "usedScene": {  
31       "abilities": [  
32         "EntryAbility"  
33       ],  
34       "when": "always"  
35     }  
36   }  
37 ],
```

权限	介绍
ohos.permission.INTERNET	网络权限，登录与通话必需
ohos.permission.GET_NETWORK_INFO	访问网络状态权限，登录与通话必需

ohos.permission.GET_WIFI_INFO	访问wifi状态权限，登录与通话必需
ohos.permission.MODIFY_AUDIO_SETTINGS	修改音量权限，音频控制需要
ohos.permission.MICROPHONE	麦克风权限，音频通话必需， <b>需要动态申请</b>
ohos.permission.CAMERA	相机权限，视频通话必需， <b>需要动态申请</b>
ohos.permission.USE_BLUETOOTH	蓝牙通话切换需要

您在 module.json5 中进行权限配置时，请确保您能够获得打开摄像头、音视频录制等权限

## 关于蓝牙权限

## 步骤四：混淆规则

## 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

## 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 (Harmony)。

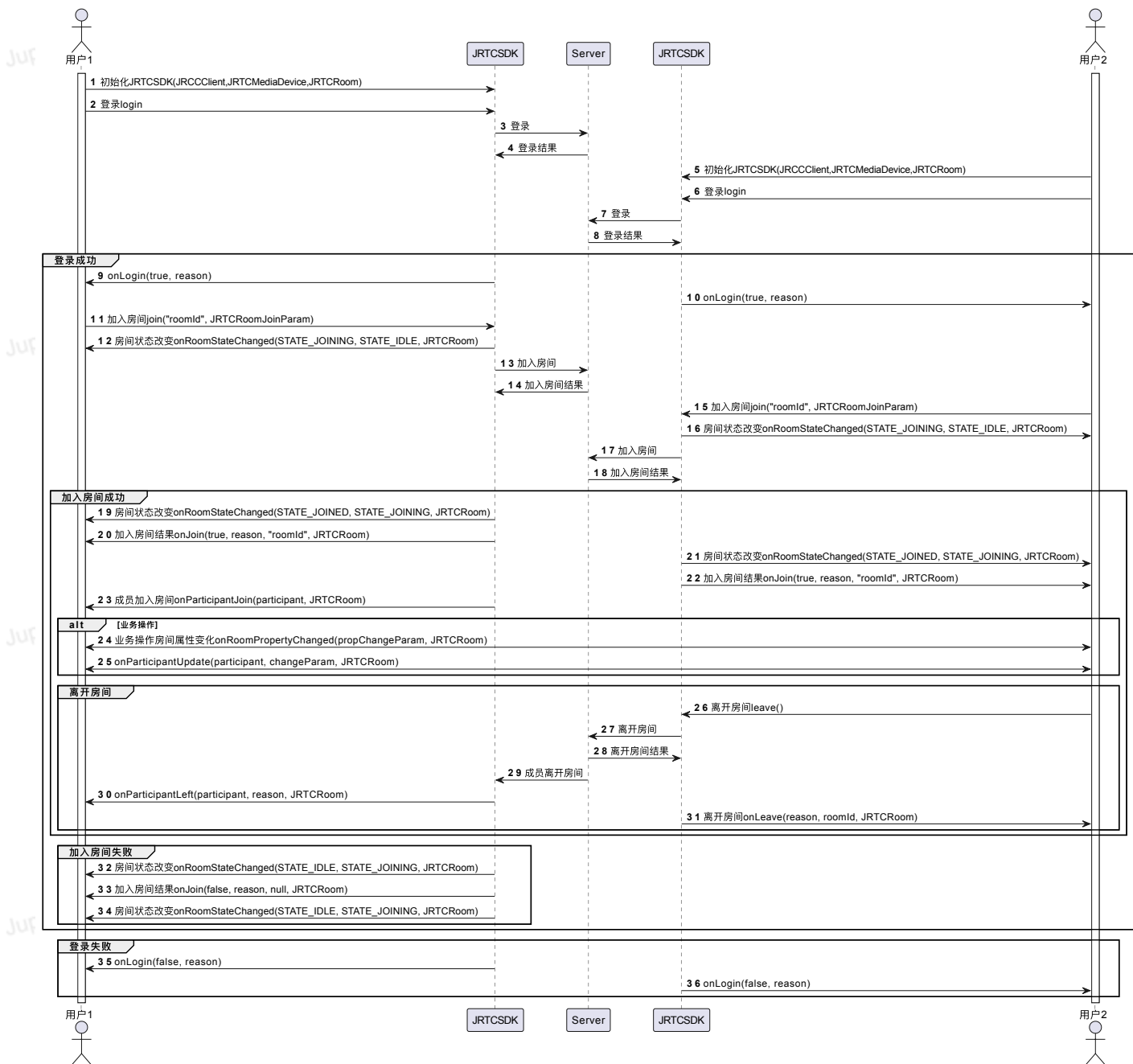
## 4.2 快速跑通 Sample

1. 在 Juphoon RTC SDK 文档中心，选择 Harmony 平台下载体验 **JCCSample** 示例项目。

访问下载地址，示例如下：

2. 下载完成后，打开安装包，解压 JCCSample，然后安装 JCCSample.apk
3. 打开应用程序后，设置正确的 appkey，环境地址以及账号。
  - a. 首先点击初始化按钮，成功后，登入按钮变为可点击；
  - b. 确认账号输入无误之后，点击登入按钮，按钮字样变为登出，即登录成功；
  - c. 点击多方通话(MpCall)按钮，即可进入多方体验相关的功能；
  - d. 进入网络电话页面后，输入房间号，如果有房间密码输入密码，点击加入，即可进入房间。

## 4.3 功能实现



### 4.3.1 初始化

注：我们所有的方法都建议在主线程调用，否则可能会出现异常无法正常使用，回调接口也都在主线程上报。

在使用业务接口前，需对 Juphoon RTC SDK 进行初始化操作。

类	模块	描述
JRTCClient	登录模块	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



<b>JRTCMediaDevice</b>	媒体模块	负责本地的媒体设备操作，视频画面渲染等功能
<b>JRTCCall</b>	通话模块	可以通过流水号加入通话，邀请其他成员加入通话
<b>JRTCRecord</b>	录制模块	实现本地音视频录制（不需要建立通话），本地分片录制等功能

```

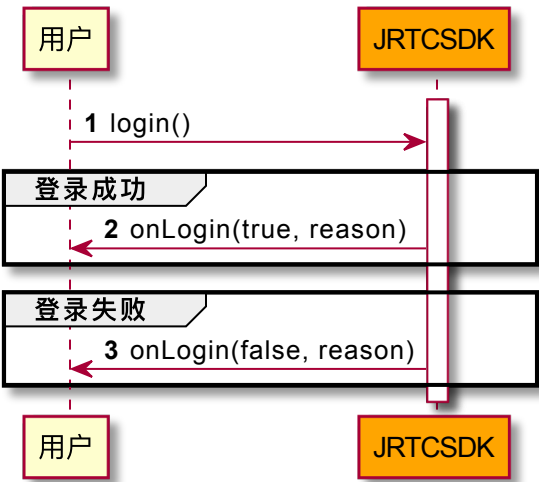
1  class JRTCManager implements JRTCClientCallback, JRTCCallCallback, JRTCMediaDeviceCallback {
2      private client: JRTCClient;
3      private mediaDevice: JRTCMediaDevice;
4      private call: JRTCCall;
5      private record: JRTCRecord;
6
7      public init(context: Context) {
8          let param: JRTCClientInitParam = new JRTCClientInitParam();
9
10         param.appName = "appName"           // 设置应用名称
11         param.SDKInfoDir = "SDKInfoDir"      // 设置SDK信息存储目录
12         param.appKey = "appKey"              // 设置AppKey
13         param.server = "server"              // 设置接入服务器地址
14         param.logConsole = true               // 设置是否控制台日志输出，默认true
15         param.logLocalFile = true            // 设置是否本地文件日志输出，默认true
16         param.looseTimeoutControl = true     // 设置是否开启 RPC 抗信令丢包控制（70%的上下行信令丢包），默认false
17
18         this.client = JRTCClient.create(context.getApplicationContext(), this, initParam);
19         this.mediaDevice = JRTCMediaDevice.create(this.mClient, this, undefined);
20         this.call = JRTCCall.create(this.client, this.mediaDevice, this);
21         this.record = JRTCRecord.create(this.client, this.mediaDevice, this);
22
23         // 设置基本参数
24         this.client.setServer("server"); // 设置接入服务器地址
25         this.client.setAppKey("appKey"); // 设置AppKey
26         this.client.setDisplayName("displayName"); // 设置显示名称
27         this.client.setAppName("appName"); // 设置应用名称
28     }
29 }

```

根据需求实现 对应 [JRTCCallCallback](#) 的接口即可。

### 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`。

```
ArkTS
1  /**
2   * 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
3   * <p>
4   * 登录结果通过 {@link JRTCClientCallback.onLogin onLogin} 回调通知
5   *
6   * @param { string } userId 用户ID
7   * @param { string } password 密码，不能为空
8   * @param { JRTCClientLoginParam? } clientLoginParam 登录参数，一般不需要设置，
9   * 如需设置请问问客服，传 undefined 则按默认值
10  * @return { boolean } 接口调用结果
11  *   - true: 接口调用成功
12  *   - false: 接口调用异常
13  * @warning 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
14  * 去创建该账号
15  * @warning 用户名为英文数字和 '+' '-' '_' '.'，长度不要超过64字符， '-' '_' '.' 不能
16  * 作为第一个字符
17  */
18  public abstract login(userId: string, password: string, clientLoginParam?: JRTCClientLoginParam): boolean;
```

#### [JRTCClientLoginParam](#) 属性介绍

属性	描述
----	----

accountEntry	设置账户分录参数，如果支持国密S3则需要设置 certificate 参数，否则可以不设置 certificate 参数
certificate	设置 S3 国密证书 Base64 编码内容
acceptExpiredCertificate	设置是否允许过期证书校验通过
token	设置token
tokenType	设置 token 校验类型
deviceId	设置设备id
logFilter	设置日志过滤标签（用于日志管理平台过滤终端日志使用）
terminalType	设置终端登录类型，支持多终端登录，默认所有终端相同会导致互踢
autoCreateAccount	是否自动创建账号（免鉴权使用），默认true
accelerateKey	设置加速云KEY
accelerateKeySecret	设置加速云KEY密钥
optimizeDataRouter	设置是否开启数据路由优化，默认true

登录的结果将会通过 [JRTCClientCallback](#) 的接口上报：

```

1  /**
2   * 登录结果回调
3   *
4   * @param { boolean } result 登录结果
5   *                                - true: 表示登录成功,
6   *                                - false: 表示登录失败
7   * @param { JRTCReasonCode } reason 登录失败原因, 当 result 为 false 时该值有效
8   */
9   onLogin?: (result: boolean, reason: JRTCReasonCode) => void;

```

示例代码：

```

1  // 创建登录配置参数
2  const loginParam: JRTCCClientLoginParam = new JRTCCClientLoginParam();
3  // 登录
4  client.login("juphoon", "123456", loginParam);
5
6  public onLogin(result: boolean, reason: number): void {
7      if (result) {
8          // 登录成功
9      } else {
10         // 登录失败，具体原因查询 reason 错误码
11     }
12 }
13

```

### 4.3.3 加入通话

```

1  /**
2  * 加入通话
3  * @note 需要已经登录
4  * @note 该接口支持加入通话并且邀请其他用户加入，通过 {@link JRTCCallJoinParam#set
   InviteCalleeUserId(string)} setInviteCalleeUserId}
5  * 和 {@link JRTCCallJoinParam#setInviteCalleeUserType(int)} setInviteCalle
   eUserType} 设置需要邀请的用户ID和用户类型
6  *
7  * <p>
8  * 该方法让用户加入通话，在同一个通话内的用户可以互相视频语音。<br>
9  * 如果用户已在通话中，必须退出当前通话，即处于空闲状态，才能进入其他通话，否则将直接返
   回 false，且不会收到回调通知。
10 *
11 * @param serialId 业务流水号，保证唯一，必选
12 * @param param 加入通话参数，传 undefined 则使用默认参数
13 * @see JRTCCallJoinParam
14 * @return 接口调用结果
15 * - true: 接口调用成功，会收到 {@link JRTCCallCallback#onJoin onJoin} 回调
16 * - false: 接口调用异常
17 */
18 public abstract join(serialId: string, param: JRTCCallJoinParam | undefine
   d): boolean;

```

[JRTCCallJoinParam](#) 参数介绍

inviteCalleeUserId	被邀请者用户ID。如果不为空，会在自身加入通话同时邀请用户加入
inviteCalleeUserType	被邀请者用户类型，默认APP用户。当被邀请者用户ID不为空时，该值有效
routeId	线路ID。当被邀请用户ID不为空，并且被邀请者用户类型是SIP用户的时候有效
extraInfo	随路参数。当被邀请用户ID不为空时，该值有效

示例代码：

```

1  let param:JRTCCallJoinParam = new JRTCCallJoinParam()
2
3  ...
4  param.video = true;
5  ...
6
7  call.join("serialId",param))
8
9  onJoin: (result: boolean, reasonCode: JRTCReasonCode) => {
10    if (result) {
11      //登录成功
12    }
13  }

```

#### 4.3.4 邀请

```

1  /**
2   * 邀请其他成员加入通话
3   * @note 需要已经加入通话
4   *
5   * @param calleeUserId 被邀请者用户ID
6   * @param param        邀请参数
7   * @return
8   * - 操作id: 接口调用成功, 对应 {@link JRTCCallCallback#onInviteResult onInviteResult} 回调的 operatorId 参数
9   * - -1: 接口调用异常, 不会收到回调
10  */
11 public abstract invite(calleeUserId: string, param: JRTCCallExtraParam | undefined): number;

```

### JRTCCallExtraParam参数介绍

video	是否视频通话
serialId	业务流水号
callerUserId	邀请人用户ID
callerDisplayName	邀请人昵称
calleeUserId	被邀请者用户ID
calleeDisplayName	被邀请者昵称
calleeUserType	被邀请者用户类型
routeId	线路ID
extraInfo	随路参数

如果类型是 `JRTCCallUserType.App`, 被邀请人会收到 `onInviteReceived` 回调

```

1  /**
2   * 收到邀请通知
3   *
4   * @param param 其他参数
5   * @see JRTCCallExtraParam
6   */
7  onInviteReceived?: (param: JRTCCallExtraParam) => void;

```

示例代码

```

1  const param: JRTCCallExtraParam = new JRTCCallExtraParam();
2  param.callerUserId = "userId";
3  param.video = true;
4  param.calleeUserType = JRTCCallUserType.APP;
5  param.routeId = "routeId";
6
7  call.invite("userId", param);
8

```

### 4.3.5 取消邀请

在被邀请人未回应邀请前，可以取消当前的邀请。

```

1  /**
2   * 取消邀请
3   *
4   * @return
5   * - 操作id: 接口调用成功，对应 {@link JRTCCallCallback#onCancelInviteResult(int, boolean, string)} onCancelInviteResult} 回调的 operatorId 参数
6   * - -1: 接口调用异常，不会收到回调
7   */
8  public abstract cancelInvite(): number;

```

取消结果回调

```

1  /**
2   * 取消邀请结果回调
3   * @param operatorId 操作id, 对应 {@link JRTCCall#cancelInvite() invite} 的返回值
4   * @param result 加入通话是否成功
5   *               - true: 成功
6   *               - false: 失败
7   * @param reason 取消邀请失败原因
8   */
9  onCancelInviteResult?: (operatorId: number, result: boolean, reason: string) => void;

```

示例代码

```

1  call.cancelInvite()

```

同时对方将收到取消邀请通知

```

1  /**
2   * 对方取消邀请通知
3   *
4   * @param param 其他参数
5   * @see JRTCCallExtraParam
6   */
7  onInviteCanceled?: (param: JRTCCallExtraParam) => void;

```

### 4.3.6 收到邀请

被邀请人收到邀请处理，可以接收或者拒绝邀请

示例代码：



```

1  /**
2   * 收到邀请通知
3   *
4   * @param param 其他参数
5   * @see JRTCCallExtraParam
6   */
7  onInviteReceived?: (param: JRTCCallExtraParam) => void;

```

#### 4.3.6.1 接收邀请

```

1  /**
2   * 接受邀请
3   *
4   * @param video 是否需要视频
5   *
6   * @return
7   * - 操作id: 接口调用成功, 对应 {@link JRTCCallCallback#onAcceptInviteResult(int, boolean, string)} onAcceptInviteResult} 回调的 operatorId 参数
8   * - -1: 接口调用异常, 不会收到回调
9   */
10 public abstract acceptInvite(video: boolean): number;

```

#### 邀请处理结果回调

```

1  /**
2   * 接受邀请结果回调
3   * @param operatorId 操作id, 对应 {@link JRTCCall#acceptInvite(boolean) acceptInvite} 的返回值
4   * @param result 加入通话是否成功
5   *               - true: 成功
6   *               - false: 失败
7   * @param reason 接受邀请失败原因
8   */
9  onAcceptInviteResult?: (operatorId: number, result: boolean, reason: string) => void;

```

示例代码:

```

1 // 接受邀请
2 call.acceptInvite(true)

```

同时对方能收到接受邀请通知

```

1 /**
2  * 对方接受邀请通知
3  *
4  * @param param 其他参数
5  * @see JRTCCallExtraParam
6  */
7 onInviteAccepted?: (param: JRTCCallExtraParam) => void;
8

```

#### 4.3.6.2 拒绝邀请

```

1 /**
2  * 拒绝邀请
3  *
4  * @return
5  * - 操作id: 接口调用成功, 对应 {@link JRTCCallCallback#onAcceptInviteResult(int, boolean, string)} onAcceptInviteResult} 回调的 operatorId 参数
6  * - -1: 接口调用异常, 不会收到回调
7  */
8 public abstract rejectInvite(): number;

```

示例代码

```

1 // 拒绝邀请
2 call.rejectInvite();

```

同时对方能收到拒绝邀请通知

```
1  /**
2   * 对方拒绝邀请通知
3   *
4   * @param param 其他参数
5   * @see JRTCCallExtraParam
6   */
7  onInviteRejected?: (param: JRTCCallExtraParam) => void;
```

### 4.3.7 视频渲染

当加入房间后，除了本地的视频画面，还有房间内其他成员的视频画面，如果房间内其他成员有视频流上传，本端可以获取到其他成员的视频流并进行渲染；

当成员视频状态变化，比如该成员开始上传视频，此时可以去渲染该成员视频，该成员结束上传视频，则可以去停止渲染该成员视频。

通过调用 [requestVideo](#) 方法订阅该视频，当成员离开需要调用 [unRequestVideo](#) 及时取消订阅该视频流。

渲染视频画面需要调用 JRTCVideoComponent 组件传入视频流id即可渲染

```

1  /**
2   * 订阅房间中其他用户的视频流
3   *
4   * @param participant JRTCRoomParticipant 成员对象
5   * @param videoSize 视频请求的尺寸, 详见 {@link JRTCVideoSize}
6   * @return 接口调用结果
7   * - true: 接口调用成功, 会收到 {@link JRTCRoomCallback#onParticipantUpdate} 回调
8   * - false: 接口调用异常
9   */
10 public abstract requestVideo(participant: JRTCRoomParticipant, videoSize: JRTCVideoSize): boolean;
11
12 /**
13 * 取消订阅房间中其他用户的视频流
14 *
15 * @param participant JRTCRoomParticipant 房间中其他成员对象
16 * @return 调用是否正常
17 * - true: 正常执行调用流程, 会收到 {@link JRTCRoomCallback#onParticipantUpdate} 回调
18 * - false: 调用失败, 不会收到回调通知
19 */
20 public abstract unRequestVideo(participant: JRTCRoomParticipant): boolean;

```

### 示例代码

```

1  @State participantArray: Array<JRTCRoomParticipant> = [];
2  onParticipantUpdate: (participant: JRTCRoomParticipant | undefined, changeParam: JRTCRoomParticipantChangeParam | undefined) => {
3    for (const participant of call.getParticipants() || []) {
4      this.participantArray.push(participant);
5    }
6  }
7
8  ForEach(this.participantArray, (participant: JRTCRoomParticipant) => {
9    JRTCVideoComponent({
10      streamId: participant.streamId,
11      mediaDevice: JRTCManager.getInstance().mMediaDevice
12    })
13  })

```

### 4.3.8 新成员加入

当新成员加入房间后，其他成员会收到 [onParticipantJoin](#) 成员加入的回调。

```
1  /**
2  * 成员加入回调
3  * <p>
4  * 当有用户调用 {@link JRTCRoom#join join} 接口加入房间成功时，已在房间中的成员会收到
   此回调。
5  *
6  * @param participant JRTCRoomParticipant 成员对象
7  * @param room        当前 JRTCRoom 对象
8  */
9  onParticipantJoin?: (participant: JRTCRoomParticipant | undefined) => void;
```

### 4.3.9 成员更新

当房间内成员状态发生改变时，其他成员能收到该成员状态变化通知 [onParticipantUpdate](#)，具体成员变化属性参考 [JRTCRoomParticipantChangeParam](#)，包含成员音量、网络状态、音视频上传状态、成员类型、视频订阅尺寸变化等。

```
1  /**
2  * 成员更新回调
3  *
4  * @param participant 成员对象
5  * @param changeParam 更新标识类
6  */
7  onParticipantUpdate?: (participant: JRTCRoomParticipant, changeParam: C
8  JRTCRoomPropChangeParam) => void;
```

### 4.3.10 成员离开

当成员离开房间后，其他成员会收到成员离开的回调 [onParticipantLeft](#) 回调通知。

```

1  /**
2   * 成员离开回调
3   *
4   * @param participant 成员对象
5   * @param reason      成员离开原因
6   */
7  onParticipantLeft?: (participant: JRTCRoomParticipant, reason: JRTCReasonCode) => void;

```

### 4.3.11 结束离开

成员可以自己离开通话或者结束通话。

```

1  /**
2   * 离开通话
3   * @note 仅自己离开
4   * @note 需要已经加入通话
5   *
6   * @return 接口调用结果
7   * - true: 接口调用成功, 非空闲状态下, 会收到 {@link JRTCCallCallback#onLeave onLeave} 回调
8   * - false: 接口调用异常
9   */
10 public abstract leave(): boolean;
11
12 /**
13 * 结束通话
14 * @note 自己离开并踢出通话中的其他成员
15 * @note 需要已经加入通话
16 *
17 * @return 接口调用结果
18 * - true: 接口调用成功, 非空闲状态下, 会收到 {@link JRTCCallCallback#onLeave onLeave} 回调
19 * - false: 接口调用异常
20 */
21 public abstract stop(): boolean;

```

离开或者结束通话会收到 [onLeave](#) 回调

```

1  /**
2   * 离开通话结果回调
3   * <p>
4   * 调用 {@link JRTCCall#leave leave} 接口成功后，会收到此回调。
5   *
6   * @param serialId    业务流水号
7   * @param reasonCode  离开原因，参见：{@link JRTCEnum.JRTCReasonCode 离开原因}
8   */
9   onLeave?: (serialId: string, reasonCode: JRTCReasonCode) => void;

```

示例代码

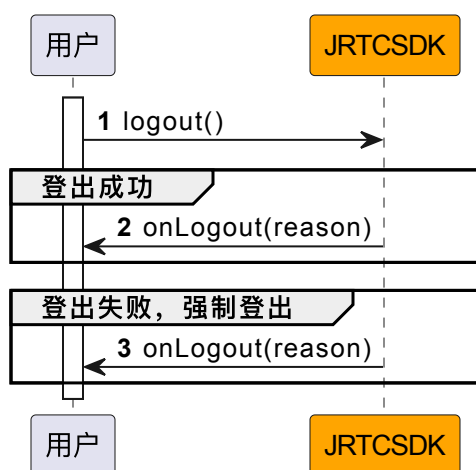
```

1  // 离开通话
2  call.leave();
3
4  // 结束通话
5  call.stop();
6
7  // 离开或者结束回调
8  onLeave:(serialId: string, reasonCode: number) => {
9
10 }

```

### 4.3.12 登出

离开房间后，可以做登出操作，登出接口调用流程如下所示：



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

```

1  /**
2   * 登出 Juphoon RTC 平台，登出后不能进行平台上的各种业务
3   * <p>
4   * 登出结果通过 {@link JRTCClientCallback.onLogout onLogout} 回调通知
5   *
6   * @return { boolean } 接口调用结果
7   *   - true: 接口调用成功
8   *   - false: 接口调用异常
9   */
10 public abstract logout(): boolean;

```

登出结果通过 [JRTCClientCallback](#) 中的 [onLogout](#) 接口进行上报。

```

1  /**
2   * 登出回调
3   *
4   * @param { JRTCReasonCode } reason 登出原因
5   */
6  onLogout?: (reason: JRTCReasonCode) => void;

```

示例代码：

```

1  // 调用登出接口
2  client.logout();
3
4  // 监听登出结果回调
5  public onLogout(reason: number) {
6      // 登出完成
7  }
8

```

### 4.3.13 登录登出状态改变通知

登录状态通过 [JRTCClientCallback](#) 中的 [onClientStateChanged](#) 接口上报



```

1  /**
2   * 登录状态变化通知
3   *
4   * @param { JRTCClientState } state 当前状态值
5   * @param { JRTCClientState } oldState 之前状态值
6   */
7  onClientStateChanged?: (state: JRTCClientState, oldState: JRTCClientState)
  => void;

```

示例代码

```

1  //登录状态改变通知
2  onClientStateChanged:(state: JRTCClientState, oldState: JRTCClientState) =
    > {
3      //state 当前状态
4      //oldState 之前状态
5  }

```

### 4.3.14 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 JRTC SDK 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，如果一定要在回调方法中调用，需要异步调用。**

```

1  /**
2   * 销毁 JRTCAll 对象
3   *
4   * @note 该方法为同步调用，需要等待 JRTCAll 实例资源释放后才能执行其他操作，调用此方法
      后，你将无法再使用 JRTCAll 的其它方法和回调。<br>
5   * 我们 **不建议** 在 JRTCSDK 的回调中调用此方法销毁 JRTCAll 对象，否则可能出现崩溃现
      象。<br>
6   * 如需在销毁后再次创建 JRTCAll 实例，需要等待 destroy 方法执行结束后再创建实例。
7   */
8  public static destroy(): void;

```

示例代码：

```

1  //建议按照初始化顺序反顺序销毁
2  JRTCCall.destroy();
3  JRTCMediaDevice.destroy();
4  JRTCClient.destroy();
5
6  //异步调用示例
7  // 登出回调
8  public onLogout(reason: number) {
9      setTimeout(() => {
10         JRTCCall.destroy();
11         JRTCMediaDevice.destroy();
12         JRTCClient.destroy();
13     });
14 }

```

## 五、通话管理

本文将介绍多方视频中通话管理的相关功能。

### 5.1 设置用户角色

由于通用化录制需要区分不同用户角色，那么需要终端以特定的角色加入到通话中；

如果是终端直接调用加入接口加入通话，那么可以在通话参数里面设置用户角色，如下：

示例代码：

```

1  let joinParam:JRTCCallJoinParam = new JRTCCallJoinParam();
2  //设置访客身份加入
3  joinParam.role = JRTCCallCenterRole.GUEST
4
5  call.join("serialId", joinParam);

```

如果终端是通过被邀请加入，那么需要在接受邀请设置用户角色，如下：

```
1  /**
2   * 设置用户角色
3   * @note 该接口只有在接受邀请前设置有效，主动加入通话，请使用加入通话参数设置
4   *
5   * @param role 用户角色
6   */
7  public abstract setRole(role: JRTCCallCenterRole);
```

示例代码：

```
1  onInviteReceived: (param: JRTCExtraParam) => {
2      //收到邀请
3      //设置访客身份加入
4      call.setRole(JRTCCallCenterRole.MAIN_GUEST);
5      //接受邀请
6      call.acceptInvite(true);
7  }
```

## 5.2 获取统计信息

实时统计信息用于在通话中查看音视频收发情况，以及分辨率、帧率、码率，网络情况等。

```

1  /**
2  * 获取统计信息
3  *
4  * 以字符串形式返回，其中包含 "Config", "Network", "Transport" 和 "Participant
5  * s" 4个节点：
6  * @verbatim
7  * {
8  *     "Config": // 音视频设置信
9  *         {
10 *             "Audio Config: // 音频设置
11 *             {
12 *                 "SRTTP": off, // 是否对音频RT
13 *                 P数据加密，以及加密会显示使用的加密协议，加密协议两端一致才会音频互通正常
14 *                 "Codec": opus, // 本端设置的音
15 *                 频编码
16 *                 "Payload": 116, // 音频payload
17 *                 的大小
18 *                 "Bitrate": 16000, // 音频码率
19 *                 "Pkt Len": 60, // 音频包长
20 *                 "Nack": off, // 丢包是否允许
21 *                 数据包重传
22 *                 "RTX": off, // 是否允许RTX
23 *                 技术
24 *                 "FEC/RED": off, // 是否开启FEC
25 *                 "AEC": on, // 是否开启回声
26 *                 消除
27 *                 "Mode": 0S, // AEC模式
28 *                 "HowlSup": Auto, // AEC HowlSup
29 *                 p模式
30 *                 "Sts": Auto, // AEC Sts模式
31 *                 "AGC": on, // 是否开启发送
32 *                 端自动增益
33 *                 "Mode": Fixed, // 发送端AGC Mo
34 *                 de
35 *                 "Target": 3, // 发送端AGC Ta
36 *                 rget
37 *                 "Gain": 9, // 接收端AGC Ga
38 *                 in
39 *                 "Rx AGC": off, // 是否开启接收
40 *                 端自动增益
41 *                 "Mode": Fixed, // 接收端AGC Mo
42 *                 de
43 *                 "Target": 3, // 接收端AGC Ta
44 *                 rget

```

```

30 *          "Gain": 9,                                // 接收端AGC Ga
31 in
32 *          "VAD": off,                                // 是否开启VAD
33 *          "Mode": Mid,                                // VAD Mode
34 *          "ANR": off,                                // 是否开发送
端噪音抑制
35 *          "Mode": High,                              // ANR mode
36 *          "Noise": N/A,                              // 噪音音量
37 *          "SNR": N/A,                                // 信噪比
38 *          "Rx ANR": off,                              // 是否开启接收
端噪音抑制
39 *          "Mode": Low,                                // 接收端ANR mo
de
40 *          "ARS": off,                                // 是否开启音频
码率控制
41 *          "BR Min": N/A,                              // ARS码率最小
值
42 *          "BR Max": N/A                              // ARS码率最大
值
43 *          },
44 *          "Video Config":                            // 视频设置
45 *          {
46 *              "SRTTP": off,                          // 是否对音频RT
P数据加密, 以及加密会显示使用的加密协议, 加密协议两端一致才会音频互通正常
47 *              "Codec": H264-SVC,                    // 双方通话采用
的编解码类型
48 *              "Payload": 125,                        // 视频Payload
的大小
49 *              "Bitrate": 2250,                      // 视频码率, 单
位kbps
50 *              "Framerate": 24,                      // 视频帧率, 单
位fps
51 *              "Resolution": 1280x720,               // 视频分辨率
52 *              "FEC": on|124|123,                   // FEC是否打开
和payload的类型号
53 *              "FIR": off,                            // 是否允许重发
关键帧
54 *              "Key Interval": 0,                    // 允许的最小关
键帧间隔
55 *              "Repeat": 0,                          // 关键帧丢失是
否允许重发
56 *              "NACK": off,                          // 丢包是否允许
数据包重传
57 *              "RTX": off,                            // 是否允许RTX
技术, RTX的payload类型
58 *              "TMMBR": off,                         // 是否允许带宽
估计

```

```

59  *          "RPSI": off,                      // 是否允许RPSI
    技术
60  *          "Small NALU": on,                  // 是否允许NALU
    技术
61  *          "ARS": off,                        // 是否开启ARS
    自动码率检测
62  *          "BR Min": 10,                      // ARS发送码率
    下限
63  *          "BR Max": 2000,                    // ARS发送码率
    上限
64  *          "FR Min": 1,                       // ARS发送帧速
    率下限
65  *          "FR Max": 30,                      // ARS发送帧速
    率上限
66  *          "Res. Ctrl": off,                  // 是否允许分辨
67  *          "Res. Mode": 0,                    // 分辨率Mode
    率控制
68  *          "Fr Ctrl": on,                     // 是否允许帧速
    率控制
69  *          "CPU Load Ctrl": off,              // 是否允许CPU
    控制
70  *          "Target": 80,                      // CPU控制的最
    大使用率
71  *          "Bw Efficient": off,               // 是否采用节省
    带宽模式
72  *          "Error Conceal": off,              // 是否允许错误
    隐藏技术, 在解码出错的时候采用
73  *          "Enhance color": off,              // 是否采用颜色
    增强技术
74  *          "Boost bright": off,               // 是否采用亮度
    增强技术
75  *          "Boost contrast": off,             // 是否采用对比
    度增强技术
76  *          "RTP Ext": CV0,                    // 使用的RTP扩
    展的类型
77  *          "Render Name": N/A,                // 渲染图像的名
    字
78  *          "SVC": "320 180 250 640 360 600 1280 720 1400",
    // 会议SVC配置
79  *          "TemporalLayers": 4,               // 取值1、2、
    3、4, 会议时间层设置
80  *          "PreferMode": Clear                // 偏好设置
81  *          }
82  *          },
83  *          "Network":                          // 网络统计信息
84  *          {
85  *          "Send Statistic:                    // 数据发送统计信息
86  *          {

```

```

87  *           "Packets": 181|1305|0|0,           // 发送的数据包的个
    数。正常包个数 | 探测包个数 | RED包个数 | NACK包个数
88  *           "RTT": 4,                           // 网络双向延时的时
    间, 单位为毫秒
89  *           "Jitter": 2,                         // 网络的扰动, 表征数
    据包抖动的时间, 单位毫秒
90  *           "Lost": 2,                           // 丢失的数据包的个数
91  *           "LostRate": 0,                       // 当前的丢包率, 单位
    百分比
92  *           "RelayLost": 0,                      // 服务器转发丢包率
93  *           "RelayRtt": 0,                      // 服务器转发往返时
    延, 单位为毫秒
94  *           "BitRate/BWE": 16/1345,             // BitRate表示当前
    发送的数据包的码率, 单位kbps; BWE表示当前发送带宽的估计值
95  *           "AudioSend": 0|0,                   // 实际发送音频包次
    数|估计发送音频包次数
96  *           "VideoSend": 0|0,                   // 实际发送视频包次
    数|估计发送视频包次数
97  *           "ScreenSend": 0|0,                  // 实际发送屏幕共享包
    次数|估计发送屏幕共享包次数
98  *           "MaxPredKbps": 100,                  // 发送最大需求码率
99  *           "Server(102679111220103708)": [2211(1): BWE(1345|697)
100 LOSS(0|0) OUT(A:37) IN(A:0;)] // 选用的第一个服务器
101 *           },
102 *           "Recv Statistic":                     // 数据接收统计信息
103 *           {
    *           "Packets": 1423|675|0|0,           // 收到的数据包个
    数。正常包个数 | 探测包个数 | RED包个数 | NACK包个数
104 *           "Jitter": 1,                         // 网络的扰动, 表征数
    据包乱序的时间, 单位毫秒
105 *           "Lost": 0,                           // 丢失的数据包的个数
106 *           "Lost Ratio": 0,                     // 当前的丢包率, 单位
    百分比
107 *           "BitRate/BWE": 178/2291,             // BitRate表示当前
    接收的数据包的码率, 单位kbps; BWE表示当前接收带宽的估计值
    *           "Server(102679111220103708)": [2211(3): BWE(1979|215
108 0) LOSS(0|0) OUT(A:37;FPS:24,FEC:10,SUB:00f0=3456) IN(A:17;V:2273=2211[00
109 f0]2273)] // 选用的第一个服务器
110 *           },
111 *           }
112 *           "Transport":                          // 运输通道
113 *           {
    *           "Local": 2.1923737535:32414,        // 本地地址
114 *           "Remote": 2:11023,                   // 远端地址
115 *           "LastPaths": 2,2,                    // 最后使用通道
116 *           "Path": 2 [udp],                     // 通道名
117 *           "Step1": Delay/Loss(S/R): 4/0/0,      // 通道质量
118 *           "Cost": 7*(best: -1)                 // 通道分数

```

```

119 *      },
120 *      "Participants":
121 *      {
122 *      "2333":                                // 成员为自己
123 *      {
124 *          "Audio Sending Stats":            // 音频发送数据统计
125 *          {
126 *              "Packets": 143,                // 发送的数据包的个数
127 *              "BitRate": 18.5,              // 发送的数据包的码
128 *                                              // 率, 单位kbps
129 *              "FecPrecent": 0                // 音频Fec保护百分
130 *                                              // 比,N/A表示未开启FEC保护
131 *          },
132 *          "Video Sending Stats":            // 视频发送数据统计
133 *          {
134 *              "Packets": 19502,              // 发送的数据包的个数
135 *              "Capture Res": 640x360,        // 视频采集分辨率
136 *              "Capture Fr": 30,              // 视频采集帧率
137 *              "FPS/IDR": [0|0|24|0]/3,       // 当前视频发送帧速/
138 *                                              // 已发送的视频关键帧数
139 *              "Resolution": 1280x720[0|0|0], // 当前发送图像最大尺
140 *                                              // 寸。[]中为每种尺寸的帧率, 取值范围为0到f(十六进制), 0表示该层视频未被发送, 值越大表示
141 *                                              // 该层视频帧率越高;
142 *              "Bitrate/Setrate": 0/2250,     // Bitrate表示当前
143 *                                              // 发送的数据包的码率, 单位kbps; Setrate表示视频编码的目标码率, 单位kbps。
144 *              "QP": 20,                      // 发送当前图像的量化
145 *                                              // 步长(0-51), 越小图像画质越好。
146 *              "EncodeTime": 10,              // 当前编码时间, 可以
147 *                                              // 体现终端编码时占用的CPU性能, 越大表示CPU占有越高, 单位毫秒
148 *              "Codec": H264-SVC,             // 采用的编解码类型
149 *              "FecPrecent": 20               // 视频Fec保护百分
150 *                                              // 比,N/A表示未开启FEC保护
151 *          },
152 *          "Be Subscribed Stats":            // 被订阅统计信息
153 *          {
154 *              "Audio": true,                 // 音频是否被订阅
155 *              "Video": [0|0|F|0]            // [S0|S1|S2|S3]表
156 *                                              // 示4个空间层被订阅
157 *          },
158 *          "Publish Stats":                  // 当前音视频发布状态
159 *          {
160 *              "Audio": true,                 // 当前音频发布状态
161 *              "Video": true                  // 当前视频发布状态
162 *          }
163 *      },
164 *      "6666":                                // 成员不是自己
165 *      {
166 *          "Audio Receiving Stats":          // 音频接收统计信息

```



```

157 *      {
158 *          "Packets": 40243,           // 接收的数据包的个数
159 *          "BitRate": 18.5,           // 当前接收的数据包的
码率, 单位kbps。
160 *          "EpdRate/lr/dc": 0/0/0,    // expand rate/los
s rate/discard rate。neteq buffer中的扩展比例/丢包比例/丢弃比例
161 *      },
162 *      "Video Receiving Stats":       // 视频接收统计信息
163 *      {
164 *          "Packets": 19502,           // 接收的数据包的个数
165 *          "BitRate": 161,             // 当前发送的数据包的
码率, 单位kbps
166 *          "FPS/FIR": 24/0,            // 当前视频接收帧率/
视频关键帧请求个数
167 *          "Resolution": 1280x720,     // 当前接收分辨率
168 *          "Render FR": 24,            // 当前渲染帧速率
169 *          "Codec": H264-SVC,          // 采用的编解码类型
170 *          "PvMos": 4.9,               // 表示过去5s平均流
畅度MOS分, 每5s更新一次。体现视频画面的流畅程度。1到5分, 1分最差, 5分最好
171 *          "SMOS": 5,                 // 表示当前清晰度MOS
分。体现视频画面的清晰程度。1到5分, 1分最差, 5分最好。前5s是0, 是正常现象, 因为PvMos
还没有值
172 *      },
173 *      "Subscribed Stats":            // 订阅统计信息
174 *      {
175 *          "Channel Audio": true,      // 当前是否发布音频
176 *          "Audio": true,              // 当前音频订阅状态
177 *          "Video": [0|0|F|0]          // [S0|S1|S2|S3]表
示4个空间层被订阅
178 *      }
179 *  }
180 *  }
181 *  }
182 @endverbatim
183 */
184 public abstract getStatistics(): string | undefined;
185
186 /**
187 * 获取实时统计信息
188 *
189 * 以Json字符串形式返回, 包含以下信息:
190 @verbatim
191 *      {
192 *          "localActor": "[username:2333@100645.cloud.justalk.com]", // a
ctorID
193 *          "sendBWE": "1440",      // 发送带宽估计
194 *          "recvBWE": "929",       // 接收带宽估计
195 *          "sendBr": "16",         // 发送码率

```

```

196 *      "recvBr": "772",          // 接收码率
197 *      "sendJitter": "1",        // 发送jitter
198 *      "recvJitter": "0",        // 接收jitter
199 *      "sendLossRate": "0",      // 发送丢包率
200 *      "recvLossRate": "0",      // 接收丢包率
201 *      "encodeTime": "0",        // 编码时长
202 *      "rtt": "5",               // 往返延时
203 *      "audioSendBr": "19",      // 音频发送码率
204 *      "videoSendBr": "0",       // 视频发送码率
205 *      "audioLevel": "58",       // 音量
206 *      "event": ""
207 *    }
208 @endverbatim
209 */
public abstract getJsonStats(): string | undefined;

```

## 5.3 获取通话唯一标识

通话唯一标识 `getCallId` 对应于业务管理平台上的 `callid`，可用于查询录像数据、查询录像上传结果等等。

```

1 /**
2  * 获取唯一标识（服务器生成）
3  *
4  * @return 房间唯一标识
5  */
6 public abstract getCallId(): string | undefined;

```

## 5.4 获取业务流水号

该业务流水号，用户可以在加入通话接口通过入参设置，也可以不传，不传则由服务端生成，且进入通话后，可以由以下接口获取

```

1  /**
2   * 获取业务流水号
3   *
4   * @return 业务流水号
5   */
6  public abstract getSerialId(): string | undefined;

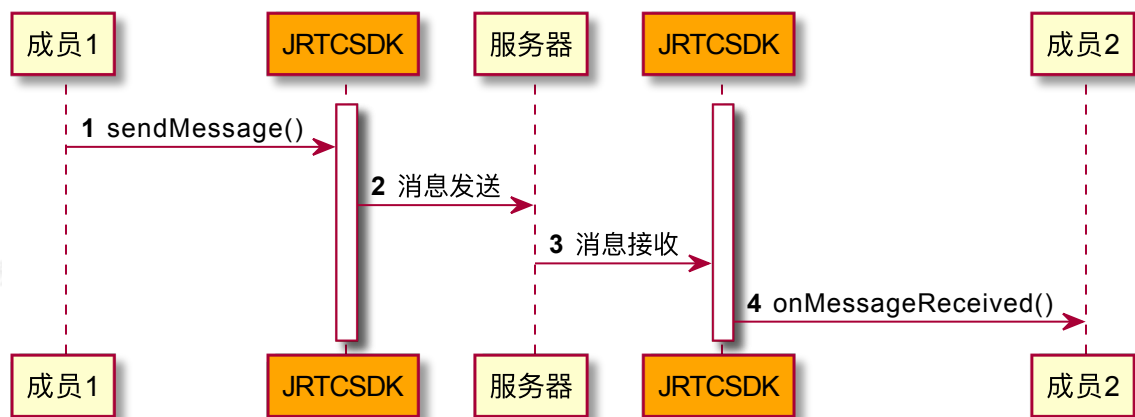
```

## 六、消息通道

透明通道消息主要包含房间内消息（接口集成使用请查看章节6.1）和在线消息（接口集成使用请查看章节6.2），具体使用要求和场景举例参考下表：

	在线消息	房间内消息
一对一发送	支持	支持
群发消息	不支持	支持
是否支持异步发送结果上报	支持	不支持
是否需要登录	是	是
是否需要建立通话	否	是
使用场景举例	1、实现不依赖通话的一对一聊天； 2、实现一些通话前的自定义信令交互，比如呼叫、拒接/接听等等；	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	只支持文本消息
消息内容大小最大支持（bit）	4K	4K

### 6.1 通话内消息



如果想在通话内给其他成员发送消息，可以调用 `sendMessage` 接口：

```

1  /**
2  * 发送消息，消息内容不能大于4K
3  *
4  * 指定成员会收到 {@link JRTCAllCallback#onMessageReceived onMessageReceived} 回调
5  * @param contentType 消息内容类型
6  * @param content 消息内容
7  * @param toUserId 指定成员的用户ID，传 undefined 给通话中全部成员发送消息
8  * @return 接口调用结果
9  * - true: 接口调用成功
10 * - false: 接口调用异常
11 */
12 public abstract sendMessage(contentType: string, content: string, toUserId: string | undefined): boolean;
  
```

消息通过实现 `JRTCAllCallback` 的 `onMessageReceived` 接口上报。

```

1  /**
2   * 收到消息回调
3   *
4   * 通话中的成员可调用 {@link JRTCCall#sendMessage(string, string, string | undefined)} 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。
5   * @param content 消息内容
6   * @param contentType 消息内容类型
7   * @param messageType 消息归属类型
8   * - {@link JRTCCallCenterMessageType#ONE_TO_ONE} : 一对一消息
9   * - {@link JRTCCallCenterMessageType#GROUP} : 群发消息(发送给通话中所有成员)
10  * @param fromUserId 发送方的用户ID
11  */
12  onMessageReceived?: (content: string | undefined, contentType: string | undefined, messageType: JRTCCallCenterMessageType, fromUserId: string) => void;
13

```

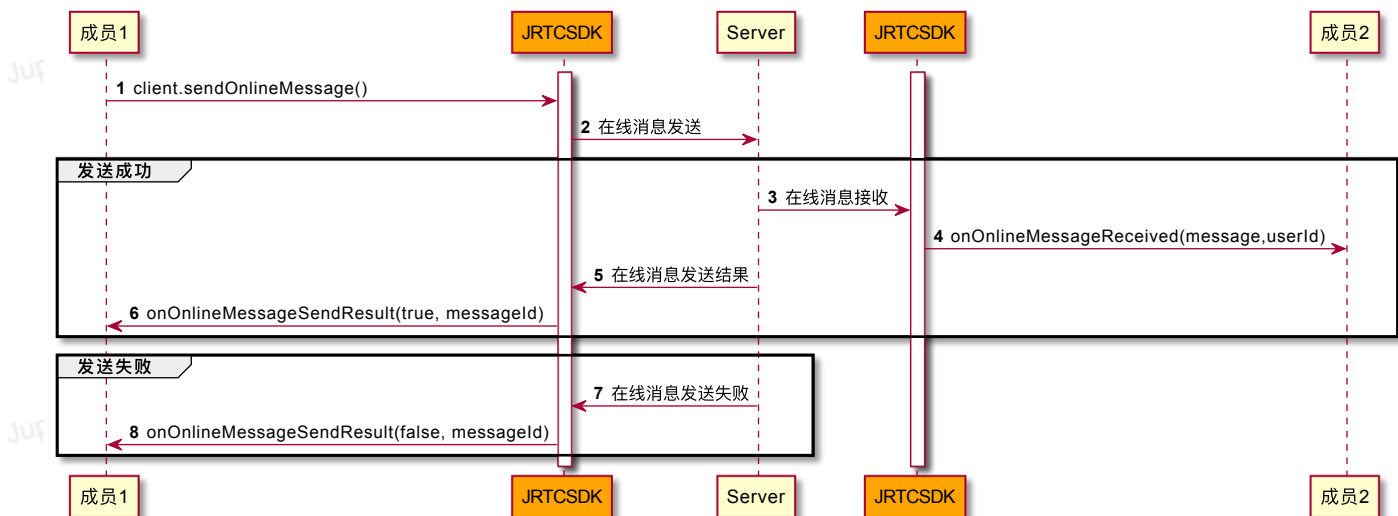
示例代码:

```

1  //成员1发送消息给房间内所有成员
2  call.sendMessage(this.confMessageType, this.confMessageContent, "");
3  //成员1发送消息给成员2
4  call.sendMessage(this.confMessageType, this.confMessageContent, "成员2");
5
6  onMessageReceived: (content: string | undefined, contentType: string | undefined, messageType: JRTCMessageType, fromUserId: string) => {
7    //成员2接收到来自成员1发送的消息
8  }

```

## 6.2 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。

```

1  /**
2  * 发送在线消息
3  *
4  * @param { string } message 消息内容
5  * @param { string } userId 对端的用户名
6  * @return { number } 接口调用结果
7  * - 操作id: 接口调用成功，对应 {@link JRTCClientCallback.onOnlineMessageSendResult onOnlineMessageSendResult} 回调的 operatorId 参数
8  * - -1: 接口调用异常，不会收到回调
9  * @note 消息大小不超过4k
10 */
11 public abstract sendOnlineMessage(message: string, userId: string): number;
  
```

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

```

1  /**
2   * 在线消息发送结果回调
3
4   *
5   * @param { boolean } result 发送结果是否成功
6   *                               - true: 发送成功
7   *                               - false: 发送失败
8   * @param { number } operatorId 操作id, 对应 {@link JRTCClient.sendOnlineMes
9   *                               sage sendOnlineMessage} 的返回值
10  */
10 onOnlineMessageSendResult?: (result: boolean, operatorId: number) => void;

```

示例代码:

```

1  @State operatorId
2
3  // 给用户 7777 发送在线消息
4  this.operatorId = client.sendOnlineMessage("消息内容", "777");
5
6  // 给用户 7777 发送在线消息结果
7  onOnlineMessageSendResult:(result: boolean, operatorId: number) => {
8      if (this.operatorId === operatorId) {
9          if(result) {
10             // 在线消息发送成功
11          } else {
12             // 在线消息发送失败
13          }
14      }
15  }
16
17  // 收到在线消息
18  onOnlineMessageReceived:(message: string, userId: string) => {
19      // 收到来自 userId 的消息, 消息内容为 message
20  }
21

```

## 七、音频管理

Juphoon 音视频能力平台及终端的媒体引擎支持音频质量保障能力，Juphoon RTC SDK 提供视频通话过程中访客端实现音频管理的功能。

## 7.1 发送本地音频流

通话中的成员可通过调用 `enableUploadAudioStream` 方法来开启关闭发送本地音频流。

```
ArkTS |
1  /**
2   * 开启/关闭发送本地音频流
3   *
4   * 通话中调用该方法可开启或关闭发送本地音频流。开启后，通话中的成员将听见本端声音；关闭
   后，频道成员将听不见本端声音 <br>
5   * 通话中调用此方法成功后，服务器会更新状态并同步给通话中所有成员，即所有成员会收到 {@link JRTCCallCallback#onParticipantUpdate onParticipantUpdate} 回调，具体可关
   注 {@link JRTCRoomParticipant#audio audio} 和 {@link JRTCRoomParticipant#audio audio} <br>
6   * 通话中调用此方法不影响接收其他成员的音频流
7   * @param enable 开启/关闭发送本地音频流
8   * - true: 开启，即发送本地音频流
9   * - false: 关闭，即不发送本地音频流
10  * @return 接口调用结果
11  * - true: 接口调用成功
12  * - false: 接口调用异常
13  */
14 public abstract enableUploadAudioStream(enable: boolean): boolean;
```

1. 在多方通话中，`enableUploadAudioStream` 的作用是开启或关闭发送本地音频流。开启后，房间成员将听见本端声音；关闭后，房间成员将听不见本端声音。房间中调用此方法不影响接收远端音频。
2. 初始化 `JRTCCall` 时，默认不发送本地音频流。若要加入房间时让房间内其他成员听见本端声音，需要在调用 `join` 加入房间前设置，或者在 `JRTCCallJoinParam` 设置。
3. 房间中调用此方法开启或关闭发送本地音频流，服务器会更新状态并同步给其他房间成员同时，房间中的其他成员会收到该成员“是否上传音频”的状态变化回调 `onParticipantUpdate`。
4. 此外，此方法还可以实现开启或关闭静音的功能。当 `enable` 值为 `false`，将会停止发送本地音频流，此时其他成员将听不到您的声音，从而实现静音功能。



```
1  /**
2   * 成员更新回调
3   *
4   * @param participant 成员对象
5   * @param changeParam 更新标识类
6   */
7  onParticipantUpdate?: (participant: JRTCRoomParticipant, changeParam: JRTCRoomParticipantChangeParam) => void;
```

示例代码：

```
1  // 关闭音频流发送
2  call.enableUploadAudioStream(false);
3
4  // 开启音频流发送
5  call.enableUploadAudioStream(true);
6
7  // 成员属性更新回调
8  onParticipantUpdate:(participant: JRTCRoomParticipant, changeParam: JRTCRoomParticipantChangeParam) => {
9      if(changeParam.audio){
10         // 成员音频上传状态发生改变
11         if(participant.audio){
12             // 该成员音频流打开
13         } else{
14             // 该成员音频流关闭
15         }
16     }
17 }
18
```

## 7.2 音频输出

```

1  /**
2  * 开启/关闭音频输出
3  * <p>
4  * - 该方法可实现本地静音功能。关闭时听不到房间内其他成员的声音，不影响其他成员；开启时可以听到其他成员声音
5  * - 初始化 JRTCRoom 时，音频输出功能默认是开启的。若要加入房间时听不见其他成员的声音，建议在调用 {@link #join join} 加入房间前设置
6  *
7  * @param enable 是否开启音频输出
8  *             - true: 开启音频输出
9  *             - false: 关闭音频输出
10 * @return 接口调用结果
11 * - true: 接口调用成功，会收到 {@link JRTCCallCallback#onCallPropertyChanged onCallPropertyChanged} 回调
12 * - false: 接口调用异常
13 */
14 public abstract enableAudioOutput(enable: boolean): boolean;

```

1. 该方法可实现本地静音功能。关闭时听不到房间内其他成员的声音，不影响其他成员；开启时可以听到其他成员声音。
2. 初始化 `JRTCCall` 时，音频输出功能默认是开启的。若要加入房间时听不见其他成员的声音，建议在调用 `join` 加入房间前设置。
3. 该方法可以关闭或重新开启音频输出功能，在房间内和房间外均可调用，且在离开房间后该设置仍然有效，也就是说这一次设置了关闭音频输出，那么下一次加入房间时也是默认关闭音频输出。

```

1  /**
2  * 通话属性改变，重点关注屏幕共享
3  *
4  * @param propChangeParam 通话改变的属性
5  */
6  onCallPropertyChanged?: (propChangeParam: JRTCRoomPropChangeParam) => void;

```

示例代码：

```
1 // 关闭音频输出
2 call.enableAudioOutput(false);
3
4 // 开启音频输出
5 call.enableAudioOutput(true);
6
7 // 房间属性变化回调
8 onCallPropertyChanged:(changeParam: PropChangeParam) => {
9   /**
10    * 输出声音状态是否变化
11    * - true: 变化
12    * - false: 没变化
13    */
14   if (changeParam.audioOutput) {
15   }
16 }
```

## 7.3 自定义音频输入

通话中可以自定义从外部音频文件作为音频源输入，使用场景举例：比如共享本地音频。

```

1  /**
2   * 开始/结束播放本地音频文件作为音频源输入
3   * @note 如果用户正在通话中，该音频将播放到通话内，通话中所有成员包括自己都能听到
4   *
5   * @param { boolean } enable 开始或者结束
6   * @param { string } filePath 音频文件路径，支持pcm, wav的格式（需要单声道，采样率
   16K音频文件）
7   * @param { boolean } loop 是否循环播放
8   * @note 重复调用会覆盖
9   * @return { boolean } 接口调用结果
10  * - true: 接口调用成功
11  * - false: 接口调用异常
12  */
13  public abstract enableAudioInputFromFile(enable: boolean, filePath: string, loop: boolean): boolean;
14
15  /**
16   * 暂停/继续播放语音文件作为音频源输入
17   *
18   * @param { boolean } suspend
19   * - true: 暂停播放
20   * - false: 继续播放
21   * @return { boolean } 调用是否正常
22   * - true: 正常执行调用流程
23   * - false: 调用异常
24   */
25  public abstract suspendAudioInputFromFile(suspend: boolean): boolean;

```

音频输入播放结束（非循环播放结束或者主动停止播放），会收到 [JRTCMediaDeviceCallback](#) 的 [onFileAudioInputDidFinish](#) 回调通知

```

1  /**
2   * 本地文件音频源输入完成回调
3   */
4  onFileAudioInputDidFinish?: () => void;

```

示例代码

```

1 // 开始分享本地文件音频输入到通话内
2 mediaDevice.enableAudioInputFromFile(true, "/sdcard/1.pcm", true);
3 mediaDevice.enableAudioInputFromFile(false, "", true);
4
5 onFileAudioInputFinish():void {
6     // 分享本地文件音频输入到通话结束
7 }

```

## 7.4 本地音频播放

通话中可以播放一段本地音频，使用场景举例：比如播放来电铃声。

```

1 /**
2  * 开始播放音频
3  * @note 不管是否在通话中，该音频播放只有本地可以听到
4  *
5  * - 当播放音频文件完成后会收到 {@link JRTCMediaDeviceCallback#onRingPlayFinish() onRingPlayFinish} 回调通知
6  * @param { string } filePath 音频文件路径，支持pcm, wav的格式（需要单声道，采样率16K音频文件）
7  * @param { boolean } isLoop 是否循环播放
8  * @return { boolean } 接口调用结果
9  * - true: 接口调用成功
10 * - false: 接口调用异常
11 */
12 public abstract startRing(filePath: string, isLoop: boolean): boolean;
13
14 /**
15 * 结束播放音频
16 *
17 * - 会收到 {@link JRTCMediaDeviceCallback#onRingPlayFinish() onRingPlayFinish} 回调通知
18 * @return { boolean } 接口调用结果
19 * - true: 接口调用成功
20 * - false: 接口调用异常
21 */
22 public abstract stopRing(): boolean;

```

音频播放结束会收到 [JRTCMediaDeviceCallback](#) 的 [onRingPlayFinish](#) 回调通知

```

1  /**
2  * 音频播放完成
3  */
4  onRingPlayFinish?: () => void;

```

示例代码：

```

1  mediaDevice.startRing("/sdcard/1.pcm", true);
2
3  // 结束播放本地音频
4  mediaDevice.stopRing();
5
6  onRingPlayFinish:() => {
7      // 本地音频播放结束
8  }
9

```

## 7.5 音频异常回调

通过实现 [JRTCMediaDeviceCallback](#) 的 [onAudioError](#) 接口监听音频异常回调，具体错误查看参数 [error](#) 描述。

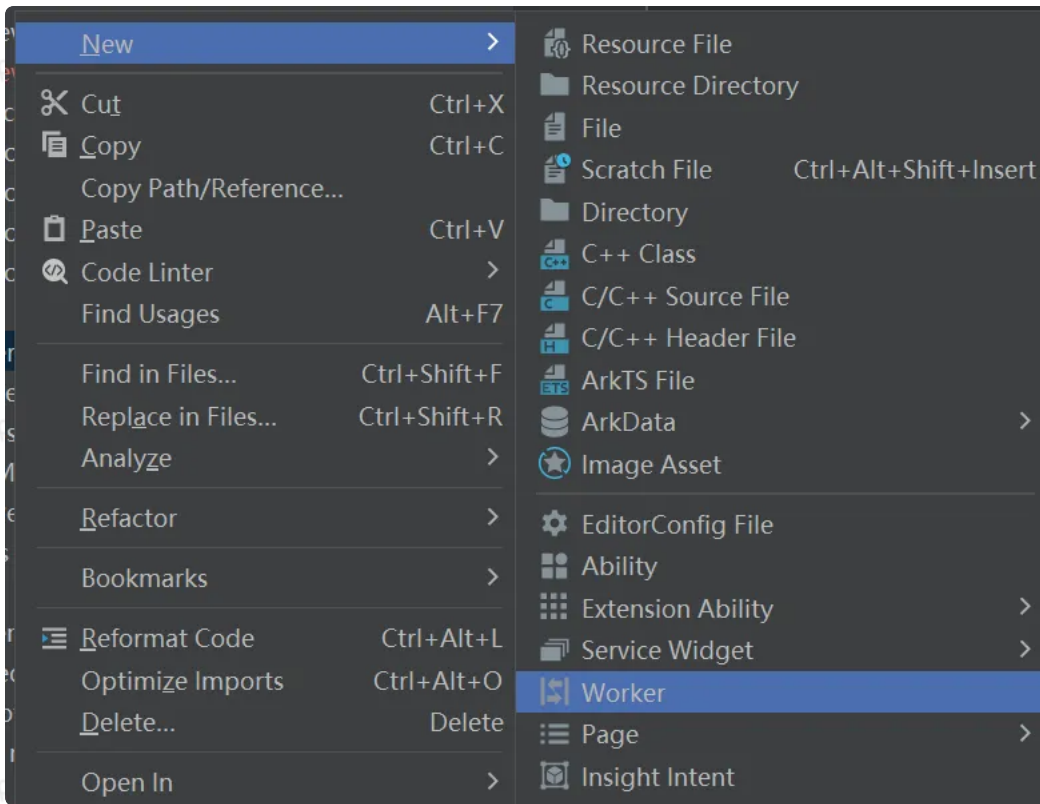
```

1  /**
2  * 音频异常
3  *
4  * @param { string } error 异常信息
5  */
6  onAudioError?: (error: string) => void;

```

## 7.6 音频数据回调

获取音频数据建议创建worker线程接收回调数据



### 7.6.1 输入音频数据回调

```
ArkTS |  
  
1  /**  
2   * 设置音频输入数据回调  
3   * @note 因为回调数据比较频繁，建议创建独立的Worker线程调用该接口  
4   * 当Worker线程结束时，内部会自动清除回调，也可以手工设置null来主动删除回调  
5   * setAudioInputFrameCallback(null)  
6   *  
7   * @param callback 全局唯一的回调函数，回调参数说明如下：  
8   * @param { string } inputId 输入源的自定义字符串  
9   * @param { number } sampleRateHz 输入源的采样频率  
10  * @param { number } channels 输入源的频道数量  
11  * @param { ArrayBuffer } data 该帧的采样数据  
12  * @return { boolean }  
13  * -true 设置成功  
14  * -false 设置失败  
15  */  
16 export function setAudioInputFrameCallback(callback: ((inputId: string, sampleRateHz: number, channels: number, data: ArrayBuffer) => void) | null): boolean
```

示例代码

```

1 //主线程
2 let worker: worker.ThreadWorker = new worker.ThreadWorker("子线程文件路径");
3 worker.postMessage({"option":"StartInput"})
4 worker.postMessage({"option":"StopInput"})
5
6
7 //子线程
8 workerPort.onmessage = (e: MessageEvents) => {
9     if (e.data.option as string == "StartInput"){
10         setAudioInputFrameCallback((inputId: string, sampleRateHz: number, channels: number, data: ArrayBuffer) => {
11             //具体操作
12         });
13     }else if(e.data.option as string == "StopInput"){
14         setAudioInputFrameCallback(null);
15     }
16 }

```

## 7.6.2 输出音频数据回调

```

1 /**
2  * 设置音频输出数据回调
3  * @note 因为回调数据比较频繁，建议创建独立的Worker线程调用该接口
4  * 当Worker线程结束时，内部会自动清除回调，也可以手工设置null来主动删除回调
5  * setAudioOutputFrameCallback(null)
6  *
7  * @param callback 全局唯一的回调函数，回调参数说明如下：
8  * @param { string } inputId 输入源的自定义字符串
9  * @param { number } sampleRateHz 输入源的采样频率
10 * @param { number } channels 输入源的频道数量
11 * @param { ArrayBuffer } data 该帧的采样数据
12 * @return { boolean }
13 * -true 设置成功
14 * -false 设置失败
15 */
16 export function setAudioOutputFrameCallback(callBack: ((inputId: string, sampleRateHz: number, channels: number, data: ArrayBuffer) => void) | null): boolean

```



```
1 //主线程
2 let worker: worker.ThreadWorker = new worker.ThreadWorker("子线程文件路径");
3 worker.postMessage({"option":"StartOutput"})
4 worker.postMessage({"option":"StopOutput"})
5
6
7 //子线程
8 workerPort.onmessage = (e: MessageEvents) => {
9     if (e.data.option as string == "StartOutput"){
10         setAudioOutputFrameCallback((inputId: string, sampleRateHz: number, channels: number, data: ArrayBuffer) => {
11             //具体操作
12         });
13     }else if(e.data.option as string == "StopOutput"){
14         setAudioOutputFrameCallback(null);
15     }
16 }
```

## 八、视频管理

### 8.1 发送本地视频流

房间内的成员可通过调用 [enableUploadVideoStream](#) 方法来开启关闭发送本地视频流。

```

1  /**
2   * 开启/关闭发送本地视频流
3   * <p>
4   * - 调用该方法可开启或关闭发送本地视频流。开启后，房间成员将可以看见本端视频画面；关闭后，房间成员将看不见本端视频画面
5   * - 房间中调用此方法不影响接收远端视频
6   * - 初始化 JRTCRoom 时，默认发送本地视频流。若要加入房间时，让房间内其他成员看见本端视频画面，建议在调用 {@link #join join} 加入房间前设置
7   * - 该方法在房间内和房间外均可调用，且在离开房间后该设置仍然有效。也就是说这一次设置了关闭发送本地视频流，那么在下次加入房间时默认会关闭发送本地视频流
8   * - 通话中也可调用此方法开启或关闭发送本地视频流，服务器会更新状态并同步给其他房间成员，即房间中所有成员都会收到 {@link JRTCRoomCallback#onParticipantUpdate onParticipantUpdate} 回调
9   *
10  * @param enable 是否发送本地视频流
11  *             - true: 开启，即发送本地视频流
12  *             - false: 关闭，即不发送本地视频流
13  * @return 接口调用结果
14  * - true: 接口调用成功
15  * - 在调用此方法时，用户不在房间中，不会收到回调
16  * - 在调用此方法时，用户在房间中，会收到 {@link JRTCRoomCallback#onRoomPropertyChanged onRoomPropertyChanged} 回调
17  * - false: 接口调用异常
18  */
19 public abstract enableUploadVideoStream(enable: boolean): boolean;

```

1. `enableUploadVideoStream` 的作用是设置“是否上传视频流数据”。调用该方法可开启或关闭发送本地视频流。开启后，房间成员将可以看见本端视频画面；关闭后，房间成员将看不见本端视频画面。房间中调用此方法不影响接收远端音频。
2. 初始化 `JRTCCall` 时，默认发送本地视频流。若要加入房间时，让房间内其他成员看见本端视频画面，建议在调用 `join` 加入房间前设置。房间中调用此方法不影响接收远端视频。
3. 房间中也可调用此方法开启或关闭发送本地视频流，服务器会更新状态并同步给其他房间成员，房间中的其他成员会收到该成员“是否上传音频”的状态变化回调 `onParticipantUpdate`。
4. 该方法在房间内和房间外均可调用，且在离开房间该设置仍然有效。也就是说这一次设置了关闭发送本地视频流，那么在下次加入房间时默认会关闭发送本地视频流。

此外，调用该方法发送本地视频流数据还要依赖摄像头是否已经打开。

```

1  /**
2   * 成员属性更新回调
3   * <p>
4   * 当房间中有成员的属性发生变化时，房间中的其他成员会收到此回调，例如音频上传状态、视频上传状态、网络状态等发生变化。
5   *
6   * @param participant JRTCRoomParticipant 成员对象
7   * @param changeParam {@link ChangeParam} 更新标识类对象
8   * @param room        当前 JRTCRoom 对象
9   */
10 onParticipantUpdate?: (participant: JRTCRoomParticipant, changeParam: JRTCRoomParticipantChangeParam) => void;

```

示例代码

```

1  // 关闭视频流发送
2  call.enableUploadVideoStream(false);
3  // 开启视频流发送
4  call.enableUploadVideoStream(true);
5  // 通话中成员属性更新回调
6  onParticipantUpdate:(participant: JRTCRoomParticipant, changeParam: JRTCRoomParticipantChangeParam) => {
7      if(participant.video){
8          //该成员视频流打开
9      } else{
10         //该成员视频流关闭
11     }
12 }
13
};

```

## 8.2 订阅/取消订阅视频流

```

1  /**
2   * 订阅房间中其他用户的视频流
3   *
4   * @param participant JRTCRoomParticipant 成员对象
5   * @param videoSize 视频请求的尺寸, 详见 {@link JRTCVideoSize}
6   * @return 接口调用结果
7   * - true: 接口调用成功, 会收到 {@link JRTCRoomCallback#onParticipantUpdate o
nParticipantUpdate} 回调
8   * - false: 接口调用异常
9   */
10 public abstract requestVideo(participant: JRTCRoomParticipant, videoSize:
JRTCVideoSize): boolean;
11
12 /**
13 * 取消订阅房间中其他用户的视频流
14 *
15 * @param participant JRTCRoomParticipant 房间中其他成员对象
16 * @return 调用是否正常
17 * - true: 正常执行调用流程, 会收到 {@link JRTCRoomCallback#onParticipantUpdat
e onParticipantUpdate} 回调
18 * - false: 调用失败, 不会收到回调通知
19 */
20 public abstract unRequestVideo(participant: JRTCRoomParticipant): boolean;

```

示例代码

```

1  onParticipantUpdate: (participant: JRTCRoomParticipant | undefined, changePa
ram: JRTCRoomParticipantChangeParam | undefined) => {
2    if (participant!.video) {
3      room.requestVideo(participant, new JRTCVideoSize(width, height));
4    }
5  }

```

## 8.3 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 JSM 房间的特性之一, SVC 可用于设置房间视频的每一层编码分辨率。该参数在房间创建时设置, 且全局统一。

具体使用详见 [SVC 说明](#)。

可在加入房间时，通过加入通话参数 `JRTCCallJoinParam` 的 `svcResolution` 属性进行设置，房间全局属性，只有第一个加入房间用户设置有效。

```
1  /**
2  * svc分辨率，默认为 "1 180 250 360 600 720 1400"
3  *
4  * @note 当参数 {@link #setVideoDefinition(int)} videoDefinition} 为 {@link
5  * JRTCRoomVideoDefinition#CUSTOM CUSTOM} 时有效
6  *
7  * 用于自定义分层参数和码率
8  *
9  * 格式：
10 * 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
11 * 四层高倍数 第四层码率 <br>
12 * 说明 <br>
13 * 1) 默认宽高比16:9，即 @ref wholeRatio <br>
14 * 2) 编码宽高最后被裁成16整除 <br>
15 * 例如 "1 180 250 360 600 720 1400" <br>
16 * 第一层 分辨率 宽320 (180*1/9*16) 高 180 (180*1) ; 码率250kbps <br>
17 * 第二层 分辨率 宽640 (360*1/9*16) 高 360 (360*1) ; 码率600kbps <br>
18 * 第三层 分辨率 宽1280 (720*1/9*16) 高 720 (720*1) ; 码率1400kbps <br>
19 * 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 160
20 * 0" <br>
21 * 第四层 分辨率 宽1920 (1080*1/9*16) 高 1080 (1080*1) ; 码率1600kbps
22 *
23 * @note 房间全局属性，第一个加入房间成员设置成效
24 */
25 public set svcResolution(value: string);
```

示例代码

```
1  let param:JRTCRoomJoinParam = new JRTCRoomJoinParam();
2  // 设置svc参数
3  param.setSvcResolution("1 180 250 360 600 720 1400 1080 1600");
4  // 加入房间
5  call.join("10086", param);
```

## 8.4 设置本地视频宽高比

在通话内设置本地视频宽高比，会影响视频画面宽高比，用于适配不同屏幕的显示需求，需在进入房间后调用。

```
ArkTS |
1  /**
2   * 设置本端视频宽高比
3   * <p>
4   * 将自己的视频采集根据宽高比裁剪后进行发送，通话中其他成员收到的画面将是裁剪后的比例。<b
   r>
5   * 该方法不影响其他成员的画面在本端的显示比例，也不影响其他成员相互之间的画面显示比例。<b
   r>
6   * 必须 ***开始通话后*** 设置才能生效，即收到 {@link JRTCAgentCallback#onCallSta
   teChanged onCallStateChanged} 回调且 type == {@link JRTCCallCenterAgentCall
   StateChangeType#TALKING} 时设置才生效。
7   *
8   * @param ratio 视频宽高比
9   * @return 接口调用结果
10  * - true: 接口调用成功
11  * - false: 接口调用异常
12  */
13 public abstract setRatio(ratio: number): boolean;
```

示例代码

```
ArkTS |
1  onJoin: (result: boolean, reasonCode: JRTCReasonCode) => {
2      call.setRatio(0.5625);
3  }
```

## 8.5 视频截图

```

1  /**
2   * 截图
3   *
4   * @param { string } streamId 要截图的视频流ID
5   * @param { string } path 要存放截图的文件路径
6   * @return 接口调用结果
7   * - true: 接口调用成功
8   * - false: 接口调用异常
9   */
10 public abstract snapshotWithStreamId(streamId: string, path: string): boolean;

```

结果通过实现 [JRTCMediaDeviceCallback](#) 中的 [onSnapshotComplete](#) 接口上报

```

1  /**
2   * 截图完成回调
3   *
4   * @param { string } file 截图路径
5   * @param { number } width 图片像素宽
6   * @param { number } height 图片像素高
7   */
8  onSnapshotComplete?: (file: string, width: number, height: number) => void;

```

示例代码：

```

1  // 截取指定 视频流ID的帧图片并且保存到指定路径
2  // 视频流，可以是本地视频流、对端视频流或者屏幕共享视频流
3  mediaDevice.snapshotWithStreamId("user_renderId", "file_save_path");
4  // 截图完成回调
5  public void onSnapshotComplete(String file, int width, int height) {
6  //截图完成，可以从文件路径获取截图文件进行后续操作
7  }
8  onSnapshotComplete:(file: string, width: number, height: number) => {
9
10 }

```

## 8.6 视频采集回调

当打开本端摄像头视频预览或者打开本地屏幕采集时，通过实现 `JRTCMediaDeviceCallback` 的 `onVideoCaptureDidStart` 接口能收到采集开始通知

```
1  /**
2   * 视频采集开始回调
3   *
4   * @param { string } streamId 视频流ID
5   * @param { number } ratio 视频宽高比
6   */
7  onVideoCaptureDidStart?: (streamId: string, ratio: number) => void;
```

## 8.7 视频异常回调

通过实现 `JRTCMediaDeviceCallback` 的 `onVideoError` 接口来监听视频异常、采集异常、渲染错误等事件，具体原因查看参数 `error` 描述。

```
1  /**
2   * 视频异常，渲染错误，包括摄像头采集错误、屏幕采集错误等回调
3   *
4   * @param { JRTCMediaDeviceVideoErrorType } errorType 异常类型
5   * @see JRTCMediaDeviceVideoErrorType.OTHER 其他未知异常
6   * @see JRTCMediaDeviceVideoErrorType.CAMERA 摄像头异常
7   * @see JRTCMediaDeviceVideoErrorType.SCREEN 屏幕采集异常
8   * @see JRTCMediaDeviceVideoErrorType.RENDER 视频渲染异常
9   * @param { string } errorDetail 异常详细描述
10  */
11  onVideoError?: (errorType: JRTCMediaDeviceVideoErrorType, errorDetail: string) => void;
```

# 九、设备管理

Juphoon RTC SDK 支持设备设置摄像头相关参数与配置，并在提供进入房间前的视频设备测试方法。

## 9.1 音频设备管理

在音频场景中，您可能需要根据实际的场地情况选择音频的采集设备和播放设备。



### 9.1.1 音频参数设置

设置 `audioParam` 音频参数，在加入通话前设置生效，若不设置参数，使用 SDK 默认值

参数	描述
<code>audioInputDevice</code>	音频输入设备
<code>audioOutputDevice</code>	音频输出设备
<code>audioInputSamplingRate</code>	音频输入采样率
<code>audioOutputSamplingRate</code>	音频输出采样率
<code>audioInputChannelNumber</code>	音频输入通道数量
<code>audioOutputChannelNumber</code>	音频输出通道数量

### 9.1.2 扬声器的开启关闭

ArkTS |

```
1  /**
2   * 开启/关闭扬声器
3   * @note 只有在音频已经启动的情况下调用才会生效
4   *
5   * @param { boolean } enable 开启或关闭扬声器
6   * - true: 开启
7   * - false: 关闭
8   */
9  public abstract enableSpeaker(enable: boolean): void
```

示例代码

ArkTS |

```
1  //听筒模式
2  mediaDevice.enableSpeaker(false);
3  //外放模式
4  mediaDevice.enableSpeaker(true);
```

### 9.1.3 获取麦克风音量级别

打开音频采集后就可以实时调用 `getMicLevel` 接口获取当前的采集音量级别，与通话状态无关。

```

1  /**
2   * 获取当前本地音量级别，音量级别范围为0-100，用以测试设备
3   * 目前只在开始麦克风检测，或者当房间内有输入音频时，才能获取到有效的音量级别
4   *
5   * @return { number } 麦克风音量级别，返回-1获取失败
6   */
7  public abstract getMicLevel(): number;

```

示例代码

```

1  //获取本地音量采集级别
2  mediaDevice.getMicLevel();

```

### 9.1.4 获取扬声器音量级别

打开音频输出后就可以实时调用 [getSpkLevel](#) 接口获取当前扬声器的音量级别，与通话状态无关。

```

1  /**
2   * 获取当前扬声器音量级别，音量级别范围为0-100，用以测试设备
3   * 目前只在开始扬声器检测，或者当房间内有输出音频时，才能获取到有效的音量级别
4   *
5   * @return { number } 扬声器音量级别，返回-1获取失败
6   */
7  public abstract getSpkLevel(): number;

```

示例代码

```

1  //获取本地扬声器的音量级别
2  mediaDevice.getSpkLevel();

```

### 9.1.5 获取当前噪声强度

```

1  /**
2   * 获取当前噪声强度
3   * 环境平均噪声强度（1s），检测需要打开麦克风 {@link startAudio startAudio} 或者
   * {@link startAudioInput startAudioInput}
4   * @return { number } 噪声强度
5   * - -1: 获取失败
6   * - 0-50dB: 噪声非常微弱
7   * - 50-60dB: 噪声较弱
8   * - 60-70dB: 噪声较强
9   * - 70dB以上: 噪声非常强
10  */
11  public abstract getAnrNoiseLevel(): number;

```

### 9.1.6 获取当前信噪比强度

```

1  /**
2   * 获取当前信噪比强度
3   * 环境平均信噪比强度（1s），检测需要打开麦克风 {@link startAudio startAudio} 或
   * 者 {@link startAudioInput startAudioInput}
4   * @return { number } 噪声强度
5   * - -1: 获取失败
6   * - 0-20dB: 噪声明显，语音含糊，较难听清
7   * - 20-40dB: 语音基本能听清，但有一定的噪声
8   * - 40dB以上: 语音非常清晰
9   */
10  public abstract getAnrNoiseRatio(): number;

```

### 9.1.7 设置是否开启自动增益控制

```

1  /**
2   * 设置是否开启自动增益控制
3   * @note 需要在打开音频输入设备 {@link startAudioInput} 或者 {@link startAudio}
   前调用才生效
4   *
5   * @param { boolean } agc0n 是否开启自动增益控制
6   */
7   public abstract setAgc(agc0n: boolean): void;

```

### 9.1.8 设置开启自适应回音消除

```

1  /**
2   * 设置开启自适应回声消除
3   * @note 需要在打开音频输入设备 {@link startAudioInput} 或者 {@link startAudio}
   前调用才生效
4   *
5   * @param { boolean } aec0n 是否开启自适应回声消除
6   */
7   public abstract setAec(aec0n: boolean): void;

```

## 9.2 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。

### 9.2.1 获取摄像头列表

```

1  /**
2   * 获取摄像头列表
3   *
4   * @return 摄像头列表
5   */
6   public abstract getCameras(): ArrayList<JRTCMediaDeviceCamera>;

```

示例代码

```

1 // 获取所有可用的摄像头列表
2 const cameras: ArrayList<JRTCMediaDeviceCamera> = mediaDevice.getCameras();

```

## 9.2.2 指定摄像头/指定摄像头采集角度

```

1 /**
2  * 指定要开启的摄像头, 在 {@link startCamera} 之前调用
3  *
4  * @param { JRTCMediaDeviceCamera } camera 摄像头对象
5  */
6 public abstract specifyCamera(camera: JRTCMediaDeviceCamera): void;
7
8 /**
9  * 指定摄像头采集角度
10 *
11 * @param { JRTCMediaDeviceVideoAngle } angle 角度
12 */
13 public abstract specifyCameraAngle(angle: JRTCMediaDeviceVideoAngle): void;

```

示例代码

```

1 // 获取所有可用的摄像头列表
2 let cameras:ArrayList<JRTCMediaDeviceCamera> = mediaDevice.getCameras();
3 // 指定要开启的摄像头
4 mediaDevice.specifyCamera(cameras[0]);
5 // 指定摄像头采集角度为90度
6 mediaDevice.specifyCameraAngle(90);

```

## 9.2.3 摄像头采集属性

```
1  /**
2   * 设置摄像头采集属性
3   *
4   * 在调用 {@link startCamera} 接口开启摄像头前设置即可生效
5   * @param { number } width 采集宽度, 默认为 640
6   * @param { number } height 采集高度, 默认为 360
7   * @param { number } frameRate 采集帧速率, 默认为 24
8   */
9  public abstract setCameraProperty(width: number, height: number, frameRate: number): void;
```

示例代码

```
1  mediaDevice.setCameraProperty(640, 360, 24);
```

## 9.2.4 开启/关闭摄像头

```

1  /**
2   * 开启摄像头
3   *
4   * @note 调用此方法时需要保证默认摄像头不为空，即 {@link defaultCamera} 不为空，否则将直接返回 false
5   *
6   * @return 接口调用结果
7   * - true: 接口调用成功
8   * - 若调用此方法前摄像头已打开，不会收到回调通知
9   * - 若调用此方法前摄像头未打开，会收到 {@link JRTCMediaDeviceCallback#onCameraUpdate onCameraUpdate} 回调
10  * - false: 接口调用异常
11  */
12  public abstract startCamera(): boolean;
13
14  /**
15   * 关闭摄像头
16   *
17   * @return 接口调用结果
18   * - true: 接口调用成功
19   * - 调用此方法前摄像头未打开，不会收到回调通知
20   * - 调用此方法前摄像头已打开，会收到 {@link JRTCMediaDeviceCallback#onCameraUpdate onCameraUpdate} 回调
21   * - false: 接口调用异常
22  */
23  public abstract stopCamera(): boolean;

```

#### 示例代码

```

1  // 打开本地摄像头
2  mediaDevice.startCamera();
3  // 关闭本地摄像头
4  mediaDevice.stopCamera();

```

### 9.2.5 切换摄像头

```

1  /**
2   * 切换摄像头
3   * @note 内部会根据当前摄像头类型来进行切换
4   *
5   * - 调用此方法时要保证摄像头已打开，否则将直接返回 false
6   * - 设备拥有两个以上摄像头，否则将直接返回 false
7   * - 满足以上两个条件后，内部会调用 {@link switchCamera switchCamera} 接口并提供返回
   返回值
8   * @return 接口调用结果
9   * - true: 接口调用成功
10  * - false: 接口调用异常
11  */
12  public abstract switchCamera(): Promise<boolean>;
13
14  /**
15   * 切换到指定摄像头
16   * @note调用此方法时需要保证摄像头已打开并且摄像头数大于0，否则将直接返回 false
17   *
18   * @param { JRTCMediaDeviceCamera } camera 摄像头对象
19   * @return 接口调用结果
20   * - true: 接口调用成功
21   * - 摄像头个数 == 1，不会收到回调
22   * - 摄像头个数 > 1，会收到 {@link JRTCMediaDeviceCallback#onCameraUpdate onCameraUpdate} 回调
23   * - false: 接口调用异常，不会收到回调
24   */
25  public abstract switchCamera(camera: JRTCMediaDeviceCamera): Promise<boolean>;
26
27  /**
28   * 切换摄像头，用于手机前置和后置摄像头的切换
29   *
30   * @return 接口调用结果
31   * - true: 接口调用成功
32   * - false: 接口调用异常
33   */
34  public abstract switchCameraBetweenFrontAndBack(): Promise<boolean>;

```

示例代码



```

1  // 切换摄像头
2  mediaDevice.switchCamera();
3  // 切换至指定摄像头
4  mediaDevice.switchCamera(mediaDevice.getCameras[0]);
5  // 前后置摄像头切换
6  mediaDevice.switchCameraBetweenFrontAndBack();

```

## 十、体验提升

### 10.1 通话中质量检测

在通话场景中，开发者经常需要了解当前通话的通话质量、设备状态等信息，监测通话的整体体验；也可将部分质量数据在 UI 层面展示给用户，使用户能够及时了解当前通话的整体质量。Juphoon RTC SDK 支持将关键的音视频状况、网络状况、设备状态的相关指标实时回调给 APP 应用层，应用层可以将收到的数据进行展示或统计。

#### 10.1.1 网络质量检测

视频通话过程中，通话中成员的网络状态发生变化导致视频通话出现质量波动的时候，SDK 会通过 [JRTCCallCallback](#) 的 [onParticipantUpdate](#) 回调进行上报。

```

1  /**
2   * 成员更新回调
3   *
4   * @param participant 成员对象
5   * @param changeParam 更新标识类
6   */
7  onParticipantUpdate?: (participant: JRTCRoomParticipant, changeParam: JRTCRoomParticipantChangeParam) => void;

```

示例代码

```
1  onParticipantUpdate(participant: JRTCRoomParticipant, changeParam: JRTCRoomParticipantChangeParam) {
2      if (changeParam.netStatus) {
3          switch (participant.netStatus) {
4              case JRTCNetStatus.DISCONNECTED:
5                  // 断开
6                  break;
7              case JRTCNetStatus.VERY_BAD:
8                  // 非常差
9                  break;
10             case JRTCNetStatus.BAD:
11                 // 差
12                 break;
13             case JRTCNetStatus.NORMAL:
14                 // 一般
15                 break;
16             case JRTCNetStatus.GOOD:
17                 // 好
18                 break;
19             case JRTCNetStatus.VERY_GOOD:
20                 // 非常好
21                 break;
22             }
23         }
24     }
25 }
```

### 10.1.2 音频质量检测

视频通话过程中，通话中成员的说话声音状态发生变化，SDK 会通过 [JRTCCallCallback](#) 的 [onParticipantUpdate](#) 回调进行上报。

示例代码：

```

1  onParticipantUpdate:(participant: JRTCRoomParticipant, changeParam: JRTCRoomParticipantChangeParam) => {
2      if (changeParam.volumeStatus) {
3          switch (participant.volumeStatus) {
4              case JRTCVolumeStatus.NONE:
5                  // 无声音 1-30
6                  break;
7              case JRTCVolumeStatus.VERY_LOW:
8                  // 很低 30-40
9                  break;
10             case JRTCVolumeStatus.LOW:
11                 // 低 40-50
12                 break;
13             case JRTCVolumeStatus.MID:
14                 // 中 50-70
15                 break;
16             case JRTCVolumeStatus.HIGH:
17                 // 高 70-80
18                 break;
19             case JRTCVolumeStatus.VERY_HIGH:
20                 // 很高 >80
21                 break;
22         }
23     }
24 }
25

```

### 10.1.3 剩余可用内存检测

通话建立后，[JRTCMediaDeviceCallback](#) 的 [onMemoryAvailable](#) 将定时上报系统中的内存剩余情况，检测系统的运行情况。

```

1  /**
2   * 上报剩余可用内存回调
3   *
4   * 周期性上报一次内存剩余情况
5   * @param { number } memorySize 当前剩余可用内存空间 (MB)
6   */
7  onMemoryAvailable?: (memorySize: number) => void;

```

示例代码

```
1  onMemoryAvailable(memorySize: number) {  
2      if (memorySize < 100) {  
3          // 内存已严重不足，已不足100M，可能影响软件正常使用  
4      } else if (memorySize < 200) {  
5          // 剩余内存紧张，已不足200M  
6      } else if (memorySize < 300) {  
7          // 剩余内存低，已不足300M  
8      } else {  
9          return;  
10     }  
11 }  
12
```

## 10.2 文件上传

获取文件上传或断点续传信息 [requestFileUploadInfo](#)

```

1  /**
2   * 获取文件上传或断点续传信息
3   *
4   * @param { string } serialId 业务id, 必选, 如果是通话业务相关文件, 需要传通话唯一
   标识 callId
5   * @param { JRTCRequestFileUploadParam } requestFileUploadParam 请求文件上传
   信息参数, 必选
6   * @return { boolean } 接口调用结果
7   * - 操作id: 接口调用成功, 对应 {@link JRTCClientCallback.onRequestFileUpload
   InfoResponse onRequestFileUploadInfoResponse } 回调的 operatorId 参数
8   * - -1: 接口调用异常, 不会收到回调
9   * @note 目前仅支持视频和图片类型文件上传, 服务端会通过文件后缀名判断
10  */
11  public abstract requestFileUploadInfo(serialId: string, requestFileUploadP
   aram: JRTCRequestFileUploadParam): number;
12
13
14  /**
15   * 获取文件上传或断点续传信息响应
16   *
17   * @param { number } operatorId 操作id, 对应 {@link JRTCClient.requestFileUp
   loadInfo requestFileUploadInfo} 的返回值
18   * @param { boolean } result 请求是否成功
19   * - true: 请求成功
20   * - false: 请求失败
21   * @param { string } url 上传地址, 分片录制文件上传场景, 首次请求分片上传信息时有效
22   * @param { string } token 文件上传所需token, 用于校验上传合法性, 需要在上传文件的
   时候携带
23   * @param { number } requestTimestamp 本次请求发起时间戳, 用于控制上传地址有效期,
   需要在上传文件的时候携带
24   * @param { string } extraInfo 随路参数
25   * @param { number } fileSize 文件大小
26   * @param { number } offset 偏移量
27   * @param { string } fileType 文件类型
28   * @param { string } server0id 上传目标服务0id
29   * @param { string } reason 请求失败原因描述, 当 result 为 false 时有效
30   */
31  onRequestFileUploadInfoResponse?: (operatorId: number, result: boolean, ur
   l: string, token: string, requestTimestamp: number, extraInfo: string, fil
   eSize: number, offset: number, fileType: string, server0id: string, reaso
   n: string) => void;
32

```

```
1 // 创建上传文件参数对象
2 const fileUploadParam = new JRTCRequestFileUploadParam();
3 fileUploadParam.setFileName("");
4 // ...
5 // 获取文件上传或断点续传信息
6 client.requestFileUploadInfo("serialId", fileUploadParam);
7
8 onRequestFileUploadInfoResponse: (operatorId: number, result: boolean, url: string, token: string, requestTimestamp: number, extraInfo: string, fileSize: number, offset: number, fileType: string, serverId: string, reason: string) => {
9   if (result) {
10     // 根据返回 url 进行文件上传
11   } else {
12     // 查看 reason 值(请求失败原因)
13   }
14 }
15
```

文件上传成功后，再调用 [completeFileUpload](#) 接口确认文件已上传

```

1  /**
2   * 文件上传完成确认
3   *
4   * @param { string } serialId 业务id, 必选, 如果是通话业务相关文件, 需要传通话唯一
   标识 callId
5   * @param { JRTCCompleteFileUploadParam } completeFileUploadParam 文件上传完
   成确认参数, 必选
6   * @return { boolean } 接口调用结果
7   * - 操作id: 接口调用成功, 对应 {@link JRTCClientCallback.onCompleteFileUploa
   dResponse onCompleteFileUploadResponse} 回调的 operatorId 参数
8   * - -1: 接口调用异常, 不会收到回调
9   * @note 通过 http 上传文件完成后, 需要调用该接口确认完成, 否则上传文件将无法在平台查询
   到
10  */
11  public abstract completeFileUpload(serialId: string, completeFileUploadPar
   am: JRTCCompleteFileUploadParam): number;
12
13  /**
14   * 文件上传完成确认响应
15   *
16   * @param { number } operatorId 操作id, 对应 {@link JRTCClient.completeFileU
   pload completeFileUpload} 的返回值
17   * @param { boolean } result 请求是否成功
18   * - true: 请求成功
19   * - false: 请求失败
20   * @param { string } fileName 服务器合并后的文件名
21   * @param { string } extraInfo 随路参数
22   * @param { string } fileType 文件类型
23   * @param { string } reason 请求失败原因描述, 当 result 为 false 时有效
24   */
25  onCompleteFileUploadResponse?: (operatorId: number, result: boolean, fileN
   ame: string, extraInfo: string, fileType: string, reason: string) => void;
26

```

示例代码

```
1 // 文件上传完成确认
2 const fileUploadParam = new JRTCCompleteFileUploadParam();
3 client.completeFileUpload("serialId", fileUploadParam);
4
5 // 文件上传完成确认响应
6 onCompleteFileUploadResponse: (operatorId: number, result: boolean, fileName: string, extraInfo: string, fileType: string, reason: string) => {
7     if (result) {
8         // 处理上传成功的情况
9     }
10 }
11
```

## 十一、屏幕共享

通过 Juphoon RTC SDK 可以在视频通话过程中实现屏幕共享，坐席可以将自己的屏幕内容，以视频的方式分享给远端参会者，从而提升沟通效率，一般适用于一对一或多人视频通话、在线通话等在线金融场景。

- 视频房间场景中，参会者可以在通话中将本地的文件、数据、网页、PPT 等画面分享给其他与会者，让其他与会者更加直观的了解讨论的内容和主题。
- 在线金融场景中，坐席可以通过屏幕共享或者窗口共享将风险揭示等画面展示给远端的访客观看，移动端访客也可将屏幕共享给坐席观看，提升沟通效率。

### 11.1 开启/关闭屏幕共享



```

1  /**
2   * 开启/关闭屏幕共享
3   *
4   * @param enable 开启或关闭屏幕共享
5   *               - true: 开启屏幕共享
6   *               - false: 关闭屏幕共享
7   * @return 接口调用结果
8   * - true: 接口调用成功
9   * - false: 接口调用异常
10  * @note 如果 {@link #setUseExternalScreenCaptureControl(boolean) setUse
    ExternalScreenCaptureControl} 为 true,
11  * 则该接口只负责信令通知, 请确保开启屏幕共享前, 已经开启了屏幕采集, 否则远端用户收到
    屏幕共享画面为黑屏
12  */
13  public abstract enableScreenShare( enable:boolean, sendScreenParam?:JRTC
    SendScreenParam ):boolean;

```

开启/关闭屏幕共享通过实现 [JRTCCallCallback](#) 中的 [onCallPropertyChanged](#) 接口上报。

可通过 [getShareUserId](#) 获取当前正在屏幕共享的成员用户ID;

可通过 [getShareStreamId](#) 获取当前屏幕共享的视频流ID。

```

1  /**
2   * 获取屏幕共享时的视频流ID, 无屏幕共享时为 undefined
3   * <p>
4   * 调用 {@link JRTCMediaDevice#startVideo startVideo} 接口渲染通话中其他成员的屏
    幕共享画面时使用。
5   *
6   * @return 屏幕共享时的视频流ID
7   */
8  public abstract getShareStreamId(): string;
9  /**
10  * 获取发起屏幕共享者的用户ID, 无屏幕共享时为 undefined
11  * <p>
12  * 可用来判断当前通话中是否有成员发起屏幕共享。
13  *
14  * @return 发起屏幕共享者的用户ID
15  */
16  public abstract getShareUserId(): string;

```

示例代码:

```
1 call.enableScreenShare(true);
```

## 11.2 共享视频采集

您可以调用 `JRTCMediaDevice` 类中的 `setScreenCaptureProperty` 方法设置屏幕共享采集属性，包括采集的高度、宽度和帧速率。该方法可以在开启屏幕共享前调用，也可以在屏幕共享中调用；如果在屏幕共享中调用，则设置的采集属性要在下次屏幕共享开启时生效。

```
1  /**
2   * 设置屏幕共享采集属性
3   *
4   * 在调用 {@link enableScreenCapture} 接口开启屏幕共享前设置即可生效
5   * @param { number } width 采集宽度，默认1280
6   * @param { number } height 采集高度，默认720
7   * @param { number } frameRate 采集帧速率，默认10
8   */
9   public abstract setScreenCaptureProperty(width: number, height: number, frameRate: number): void;
10  /**
11   * 开启/关闭屏幕采集
12   *
13   * @param { boolean } enable 是否开启
14   * @return { boolean } 开启/关闭 是否成功
15   */
16  public abstract enableScreenCapture(enable: boolean): boolean;
```

示例代码：

```
1  //开启/关闭采集
2  mediaDevice.enableScreenCapture(true);
```

## 11.3 暂停/恢复屏幕共享

```

1  /**
2   * 暂停/继续屏幕共享
3   *
4   * @param suspend true 暂停屏幕共享, false 继续屏幕共享
5   * @param tip      暂停屏幕共享后提示文字
6   * @return 接口调用结果
7   * - true: 接口调用成功, 会收到 {@link JRTCRoomCallback#onRoomPropertyChanged
8   * onRoomPropertyChanged} 回调, 可通过{@link #isSuspendScreenShare isSuspendScreenShare} 判断当前屏幕共享是否暂停
9   * - false: 接口调用异常
10  * @note 只有自己发起的屏幕共享可以使用该接口暂停, 多次调用会覆盖
11  */
12 public abstract suspendScreenShare(suspend: boolean, tip: string): boolean;

```

查询屏幕共享是否暂停

```

1  /**
2   * 是否屏幕共享暂停
3   *
4   * @return - true: 暂停屏幕共享
5   * - false: 未暂停屏幕共享
6   */
7  public abstract isSuspendScreenShare(): boolean;

```

暂停/恢复屏幕共享变化事件通过实现 `JRTCCallCallback` 中的 `onCallPropertyChanged` 接口上报。

示例代码：

```

1  // 暂停屏幕共享
2  call.suspendScreenShare(true,"屏幕共享暂停中");
3
4  onCallPropertyChanged: (changeParam: JRTCRoomPropChangeParam | undefined)
    => {
5      if (changeParam!.screenShare) {
6          // 屏幕共享状态发生改变
7          if(call.isSuspendScreenShare()) {
8              // 屏幕共享暂停中
9          }
10     }
11 }

```

## 11.4 订阅/取消订阅屏幕共享的视频流

如果通话中有成员开启了屏幕共享，其他成员将收到 `onCallPropertyChanged` 的回调，并通过 `getShareUserId` 获得发起屏幕共享的成员用户 ID。

```

1  /**
2   * 通话属性改变，重点关注屏幕共享
3   *
4   * @param propChangeParam 通话改变的属性
5   */
6  onCallPropertyChanged?: (propChangeParam: JRTCRoomPropChangeParam) => void;

```

此时可以调用 `requestScreenVideo` 方法请求订阅屏幕共享的视频流。

```

1  /**
2   * 订阅屏幕共享的视频流
3   *
4   * @param videoSize 视频请求的尺寸
5   * @return 接口调用结果
6   * - true: 接口调用成功
7   * - false: 接口调用异常
8   */
9  public abstract requestScreenVideo(videoSize: JRTCVideoSize): boolean;

```

取消订阅屏幕共享的视频流，如果不需要屏幕共享视频流，此时可以调用 [unRequestScreenVideo](#) 方法取消订阅屏幕共享的视频流，建议不使用时取消订阅屏幕共享的视频流，否则可能造成资源浪费。

```
1  /**
2   * 取消订阅屏幕共享的视频流
3   *
4   * @return 接口调用结果
5   * - true: 接口调用成功
6   * - false: 接口调用异常
7   */
8  public abstract unRequestScreenVideo(): boolean;
```

## 11.5 渲染共享画面

获取屏幕共享相关参数 [getShareUserId](#) 和 [getShareStreamId](#)

```
1  /**
2   * 获取屏幕共享时的视频流ID，无屏幕共享时为 undefined
3   * <p>
4   * 调用 {@link JRTCMediaDevice#startVideo startVideo} 接口渲染通话中其他成员的屏
5   * 幕共享画面时使用。
6   *
7   * @return 屏幕共享时的视频流ID
8   */
9  public abstract getShareStreamId(): string;
10 /**
11 * 获取发起屏幕共享者的用户ID，无屏幕共享时为 undefined
12 * <p>
13 * 可用来判断当前通话中是否有成员发起屏幕共享。
14 *
15 * @return 发起屏幕共享者的用户ID
16 */
17 public abstract getShareUserId(): string;
```

屏幕共享开始/结束均通过实现 [onCallPropertyChanged](#) 接口上报。

示例代码：

```
1 onCallPropertyChanged: (changeParam: JRTCRoomPropChangeParam | undefined) =  
  > {  
2   if (changeParam?.screenShare) {  
3     if (room.getShareUserId() !== undefined && room.getShareStreamId().length > 0) {  
4       // 屏幕共享打开, 可通过guest.getShareStreamId 渲染共享视频画面  
5     }else {  
6       // 屏幕共享关闭, 可停止渲染共享视频画面  
7     }  
8   }  
9 }
```

## 十二、音视频录制

在开展在线理财、开户、面签等业务时，应国家监管要求，必须提供录音录像服务，形成交易记录的视频，存档备查。

Juphoon RTC SDK 在音视频通话过程中，支持全程进行实时的服务器录制或本地录制，录制的场景画面覆盖座席画面和所有终端类型的访客画面，按需可支持多摄像头、多屏幕合成录制，满足用户记录业务办理全过程录制的需求。

### 12.1 本地录制

本地录制支持实时的通话过程音频录制，录制文件保存在用户本地设备中，适用于通话过程录音录像场景等其他音视频相关场景。优点不受网络影响，录制画面质量高，减少带宽压力。

录制通话在本地生成视频文件。

```

1  /**
2   * 开启/关闭本地录制
3   *
4   * @param enable      开启或关闭本地录制
5   *                    - true: 开启本地录制
6   *                    - false: 关闭本地录制
7   * @param recordParam 本地录制参数配置, 当 enable == true 时, {@link JRTCRecord
   *                    LocalParam#filePath} 必须设置, 其余参数不设置则使用默认配置; 当 enable == false
   *                    时, recordParam 可传 undefined
8   * @return 接口调用结果
9   * - true: 接口调用成功
10  * - false: 接口调用异常
11  * @note 确保调用接口前本地录制文件所在目录已经存在, 否则会录制失败
12  * @see JRTCRecordLocalParam
13  */
14  public abstract enableLocalRecord(enable: boolean, recordParam: JRTCRecord
   *                    LocalParam | undefined): boolean;
15
16  /**
17   * 获取是否正在本地录制
18   *
19   * @return 是否正在本地录制
20   * - true: 本地录制中
21   * - false: 未进行本地录制
22   */
23  public abstract isLocalRecording(): boolean;

```

本地录制参数详见 [JRTCRecordLocalParam](#)。

示例代码：

```

1 let param: JRTCRecordLocalParam = new JRTCRecordLocalParam();
2 param.recVideo = true; // 设置是否录制视频
3 param.recAudio = true; // 设置是否录制音频
4 param.includeSelf = true; // 设置录制是否包含自己
5 param.mergeMode = JRTCVideoMergeMode.INTELLIGENT_LAYOUT; // 设置媒体录制视频合并模式，默认智能分屏模式，当使用配置文件时，该参数无效
6 param.intelligentMergeMode = JRTCIntelligentMergeMode.FREE_LAYOUT; // 智能分屏模式下的布局样式（无屏幕共享）
7 param.scsMergeMode = JRTCScsMergeMode.SCREEN_SHARE; // 智能分屏模式下的布局样式（有屏幕共享）
8 param.frameRate = 18; // 设置录制帧率，默认15
9 param.iBitrate = 0; // 设置录制码率
10 param.videoWidth = 640; // 设置录制视频的宽度
11 param.videoHeight = 360; // 设置录制视频的高度
12 param.filePath = ""; // 设置保存的文件路径

```

更新本地录制自定义布局 [updateLocalRecordLayout](#)，当本地录制已经在进行时，可以通过该接口实时更新本地录制布局

```

1 /**
2  * 更新本地录制自定义布局
3  *
4  * @param layoutList 需要更新的布局列表
5  * @return 接口调用结果
6  * - true: 接口调用成功
7  * - false: 接口调用异常
8  */
9 public abstract updateLocalRecordLayout(layoutList: ArrayList<JRTCRecordLocalLayout>): boolean;

```

示例代码：



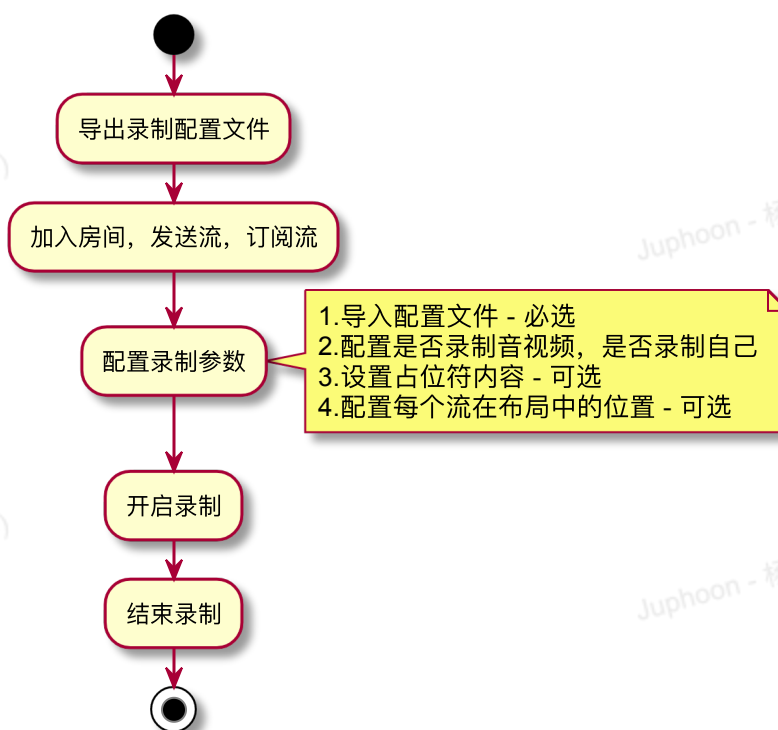
```

1  let layoutList:ArrayList<JRTCRecordLocalLayout> = new ArrayList();
2  let layout1:JRTCRecordLocalLayout = new JRTCRecordLocalLayout();
3  layout1.setId("streamId0",false);
4  layout1.position = 0;
5  layoutList.add(layout1);
6
7  let layout2:JRTCRecordLocalLayout = new JRTCRecordLocalLayout();
8  layout1.setId("streamId1", false);
9  layout1.position = 1;
10 layoutList.add(layout2);
11
12 call.updateLocalRecordLayout(layoutList);

```

通过配置文件实现本地录制

流程图如下



本地录制配置文件示例如下：

```

1  {
2    "default" : {    //录制配置类型，固定值。
3      "videoRecord" : {    //视频录制配置，固定值。
4        "layout" : [    //自定义布局，mergeMode 字段为自定义布局模式 {@link J
RTCVideoMergeMode#CUSTOM_LAYOUTCUSTOM_LAYOUT(4)} 时生效，JSON数组格式，可同时
设置多个，实际应用时通过窗口号区分。
5          {
6            "posX" : 0.25,    //画面左上角的横坐标(双精度浮点数类
型)，代表占画面总宽度的比例。
7            "posY" : 0.26805556000000003,    //画面左上角的纵坐标(双精度浮点数类
型)，代表占画面总高度的比例。
8            "width" : 0.1140625,    //画面的宽度(双精度浮点数类型)，代表
占画面总宽度的比例。
9            "height" : 0.47916666000000002, //画面的高度(双精度浮点数类型)，代表
占画面总高度的比例。
10           "window" : 0    //窗口号(大于或等于的整数)，在使用
{@link RecordLayout#setLayoutList(List) setLayoutList} 自定义视频窗口布局时需
要，对应于用户自定义的位置信息 {@link RecordLayout#setPosition(int)} setPositi
on} 设置布局位置。
11         },
12         {
13           "posX" : 0.49609375,    //画面左顶点的横坐标(双精度浮点数类
型)，范围为[0, 1]，代表占画面总宽度的比例。
14           "posY" : 0.033333334999999999,    //画面左顶点的纵坐标(双精度浮点数类
型)，范围为[0, 1]，代表占画面总高度的比例。
15           "width" : 0.364062500000000001,    //画面的宽度(双精度浮点数类型)，范围
为[0, 1]，代表占画面总宽度的比例。
16           "height" : 0.94027775999999996, //画面的高度(双精度浮点数类型) 范围
为[0, 1]，代表占画面总高度的比例。
17           "window" : 1    //窗口号(大于或等于的整数)，在使用
{@link RecordLayout#setLayoutList(List) setLayoutList} 自定义视频窗口布局时需
要，对应于用户自定义的位置信息 {@link RecordLayout#setPosition(int)} setPositi
on} 设置布局位置。
18         }
19       ],
20       "mergeBitrate" : 0, //设置录制视频的码率，单位为千比特每秒(kbps)，默认值
为0，此时码率由内部媒体算法进行自适应调节
21       "mergeFPS" : 30,    //设置录制视频的帧率，单位为每秒传输帧数(fps)，默认值
为20。
22       "mergeWidth" : 1280, //设置录制视频的宽度，默认值为 640
23       "mergeHeight" : 720, //设置录制视频的高度，默认值为 360
24       "mergeMode" : 4,    //设置媒体推流的视频合并模式，默认值为 {@link JRTCVideoMergeMode#INTELLIGENT_LAYOUT INTELLIGENT_LAYOUT(5)}，目前可支持自定义布
局 {@link JRTCVideoMergeMode#CUSTOM_LAYOUTCUSTOM_LAYOUT(4)} 和智能布局 {@link JRTCVideoMergeMode#INTELLIGENT_LAYOUT INTELLIGENT_LAYOUT(5)}。

```

```

25     "mergeModeI" : 1,    //设置智能分屏模式下的布局样式（无屏幕共享），默认值为
自由布局 {@link JRTCIntelligentMergeMode#FREE_LAYOUT FREE_LAYOUT(1)}, 有效值
参考 {@link JRTCIntelligentMergeMode. 智能分屏模式下的布局样式（无屏幕共享）}, 本
字段仅在 mergeMode 为智能布局 {@link JRTCVideoMergeMode#INTELLIGENT_LAYOUT IN
26     TELLIGENT_LAYOUT(5)} 时生效.
    "screenShareType" : 1 //设置智能分屏模式下的布局样式（有屏幕共享），默认值
为屏幕共享独占 {@link JRTCScsMergeMode#SCREEN_SHARE SCREEN_SHARE(1)}, 有效值
参考 {@link JRTCScsMergeMode 智能分屏模式下的布局样式（有屏幕共享）}, 本字段仅在 m
ergeMode 为智能布局 {@link JRTCVideoMergeMode#INTELLIGENT_LAYOUT INTELLIGEN
T_LAYOUT(5)} 时生效.
27 },
28     "watermark" : { //水印配置，固定值.
29         "picture" : [ //图片水印配置，固定值，图片水印目前只支持 png 格式.
30             {
31                 "enable" : true,    //是否启用(布尔类型).
32                 "pcUrl" : "http://192.168.17.60:10042/protected_files/6afa960e-f
e4a-49ae-a939-48c788122192?attname=juphoon.png",    //图片水印链接地址.
33                 "index" : 1,        //水印的序号.
34                 "state" : 1,        //水印的状态. 设置值为1表示使用水印，设置值为2表示
关闭水印.
35                 "posX" : 0,          //相对于基准位置的水平偏移值(整数类型)，负值向左偏
移，正值向右偏移，实际大小非比例值.
36                 "posY" : 600        //相对于基准位置的垂直偏移值(整数类型)，负值向上偏
移，正值向下偏移，实际大小非比例值.
37             }
38         ],
39         "text" : { //文本水印配置，固定值.
40             "enable" : true,    //是否启用(布尔类型).
41             "memo" : [          //文本水印样式配置(JSON数组类型)，如果不设置就使用
默认全局样式.
42                 "Dialogue: 0,0:00:00.0,60:00:00.0,Default,,0,0,0,,{\pos(34, 56)
\\an7}菊风文本水印1$name@$",
43                 "Dialogue: 0,0:00:00.0,60:00:00.0,Default,,0,0,0,,{\pos(50, 10
0)\an7}菊风文本水印2"
44                 //数组的每个元素为一条 ASS 字幕的 event 字符串，支持通过标签来设置各种文字
特效，添加的event字符串必须是utf-8编码，否则中文会出现乱码。ASS 格式规范下载地址: ht
tp://www.perlfu.co.uk/projects/asa/ass-specs.doc
45                 //其中以 "$@" 开始并且以 "@$" 结束的部分内容"$@name@$"将提取出关键字"na
me"，通过解析用户通过 {@link #setWatermarkTextMap(Map) setWatermarkTextMap}
设置的自定义文本水印信息获取其对应值，"$@name@$"格式的内容被其对应值替换后就是实际应用的
event 字符串，"$@xxx@$" 格式的内容支持同时设置多个.
46             ],
47             "style" : {          //文本水印格式，固定值.
48                 "enable" : true,    //格式是否启用(布尔类型).
49                 "alignment" : 0,     //对齐方式，有效值参考 0: 左对齐; 1: 居中对
齐; 2: 右对齐;
50                 "backColor" : 16777215, //背景颜色(整数类型)，10进制颜色代码.
51                 "blod" : false,     //是否使用粗体(布尔类型).

```

```

52         "fontColor" : 16777215, //字体颜色(整数类型), 10进制颜色代码.
53         "fontFile" : "SourceHanSansCN-Normal.otf", //字体文件路径, Windows
        s系统上只能用相对路径(相对配置文件所在路径), 不可以带盘符, 建议和配置文件放在同个目录
        下.
54         "fontSize" : 36,           //字体尺寸(整数类型).
55         "italic" : true,           //是否使用斜体(布尔类型).
56         "underline" : false        //是否带有下划线(布尔类型).
57     }
58 },
59     "timestamp" : { //时间戳水印配置, 固定值.
60         "enable" : true,           //是否启用(布尔类型).
61         "basePosType" : 0,         //水印基准位置类型(整数类型), 有效值参考 0: 左上;
        1: 左下; 2: 右上; 3: 右下; 4: 居中;
62         "borderWidth" : 2,         //字体边界宽度(整数类型), 取值范围为[0, 5].
63         "fontFile" : "SourceHanSansCN-Normal.otf", //字体文件路径, Windows
        系统上只能用相对路径(相对配置文件所在路径), 不可以带盘符, 建议和配置文件放在同个目录下.
64         "fontColor" : 0,           //字体颜色(整数类型), 有效值参考 0: 红色; 1: 黄色; 2:
        绿色; 3: 青色; 4: 蓝色; 5: 洋红色; 6: 白色; 7: 中和色; 8: 黑色;
65         "fontSize" : 36,           //字体尺寸(整数类型).
66         "isMs" : true,             //是否显示毫秒值(布尔类型).
67         "posX" : 0,                //相对于基准位置的水平偏移值(整数类型), 负值向左偏
        移, 正值向右偏移, 实际大小非比例值.
68         "posY" : 0                //相对于基准位置的垂直偏移值(整数类型), 负值向上偏
        移, 正值向下偏移, 实际大小非比例值.
69     }
70 }
71 }
72 }

```

配置样例文件(不带注解): [record.cfg.zip](#)

## 12.2 本地录制(不需要建立通信)

视频录制参数详见: [JRTCRecordVideoCaptureParam](#)

```

1  /**
2  * 开启视频录制（本地录制，不需要建立通信，不能和音频录制 {@link startAudioRecord startAudioRecord} 同时开启）
3  *
4  * @param { string } streamId 视频流ID，（包括摄像头ID、文件视频源ID、屏幕采集流ID等）
5  * @param { JRTCRecordVideoCaptureParam } recordParam 录制参数
6  * @return { boolean } 接口调用结果
7  * - true: 接口调用成功
8  * - false: 接口调用异常
9  */
10 public abstract startVideoCaptureRecord(streamId: string, recordParam: JRTCRecordVideoCaptureParam): boolean;
11
12 /**
13 * 关闭视频录制（本地录制，不需要建立通信）
14 *
15 * @param { string } streamId 视频流ID（包括摄像头ID、文件视频源ID、屏幕采集流ID等）
16 * @return { boolean } 关闭视频录制是否成功
17 */
18 public abstract stopVideoCaptureRecord(streamId: string): boolean;
19 /**
20 * 开启音频录制（本地录制，不需要建立通信，不能和视频录制 {@link startVideoCaptureRecord startVideoCaptureRecord} 同时开启）
21 *
22 * @param { string } filePath 保存的文件路径，必须包含文件名（xxx.wav或者xxx.pcm）
23 * @param { JRTCMediaDeviceRecordAudioSource } audioSource 录制文件音频源
24 * @param { JRTCMediaDeviceAudioRecordFileType } fileType 录制文件编码封装类型
25 * @return { boolean } 接口调用结果
26 * - true: 接口调用成功
27 * - false: 接口调用异常
28 */
29 public abstract startAudioRecord(filePath: string, audioSource: JRTCMediaDeviceRecordAudioSource, fileType: JRTCMediaDeviceAudioRecordFileType): boolean;
30
31 /**
32 * 关闭音频录制（本地录制，不需要建立通信）
33 *
34 * @return { boolean } 接口调用结果
35 * - true: 接口调用成功
36 * - false: 接口调用异常
37 */

```

```
38 public abstract stopAudioRecord(): boolean;  
39
```

录制水印(**目前不支持**): 本地录制支持图片、文字、时间戳水印, 通过录制参数设置, 详见: Image  
Text TimeStamp

示例代码:

```
1 // 打开摄像头  
2 mediaDevice.startCamera();  
3 // 打开麦克风  
4 mediaDevice.startAudioInput();  
5 let param: JRTCRecordVideoCaptureParam = new JRTCRecordVideoCaptureParam  
6 ();  
7 param.filePath = "filePath";  
8 param.audioSource = JRTCMediaDeviceRecordAudioSource.FROM_MICROPHONE;  
9 param.fileType = JRTCMediaDeviceVideoRecordFileType.MP4_H264;  
10 param.width = 640  
11 param.height = 340  
12 mMediaDevice.startVideoCaptureRecord(mMediaDevice.getCurrentCamera().cameraId!, param)  
13 // 开启本地音频录制  
14 mediaDevice.startAudioRecord("/sdcard/1.wav", param.audioSource, param.file  
15 Type);  
16 // 关闭本地音频录制  
17 mediaDevice.stopAudioRecord();
```

## 12.3 远程录制

在线上金融的应用场景中, 考虑取证、质检、审核、存档和回放等需求, 常需要将整个视频通话过程录制, 并存储。

Juphoon RTC SDK 的远程录制, 通过会场服务将收到的所有终端数据发送给录制服务器, 进行实时录制。在实际的集成中, 当 SDK 初始化完成后, 集成方通过加入会场的方式将需要进行录制的音视频流上传至服务器并进行实时录制。

### 12.3.1 开启/关闭远程录制

访客在发起呼叫的时候可以携带 `JRTCCallCenterCallParam.autoRecord` 参数来设置是否在通话开始后就进行远程录制。

如果该次通话没有设置自动远程录制，则可以通过终端来触发录制，在服务端形成通话过程的视频文件。

录制的配置（如水印、时间戳等）与自动录制的相同，都来自系统管理平台后台的配置。

```
ArkTS |
1  /**
2   * 开启/关闭远程视频录制
3   *
4   * 当呼叫参数 {@link JRTCCallCenterCallParam#autoRecord autoRecord} == false 时，可通过此接口开启服务端录制。<br>
5   * 可用过 {@link #getRemoteRecordState} 接口获取当前服务器录制状态。
6   * @param enable 开启或关闭视频录制
7   * - true: 开启视频录制
8   * - false: 关闭视频录制
9   * @param recordParam 录制参数，当 enable == false 时，可传 undefined；当 enable == true 且按照默认配置进行录制可传 undefined
10  * @return 接口调用结果
11  * - true: 接口调用成功，录制状态通过 {@link JRTCGuestCallback#onCallPropertyChanged onCallPropertyChanged} 回调获得，具体可关注 {@link JRTCRoom.PropChangeParam#remoteRecordState remoteRecordState}
12  * - false: 接口调用异常
13  */
14  public abstract enableRemoteRecord(enable: boolean, recordParam: JRTCRecordRemoteParam): boolean;
```

注意：该远程录制接口不经过排队机服务。

远端录制参数详见 [JRTCRecordRemoteParam](#)。

录制状态改变通过 [onCallPropertyChanged](#) 回调通知到访客。

通话属性改变详见 [JRTCRoomPropChangeParam](#)，录制状态具体可关注 [getRemoteRecordState](#)。

```
ArkTS |
1  /**
2   * 通话属性改变回调
3   * @note
4   * 重点关注屏幕共享，即当{@link PropChangeParam#screenShare screenShare} 属性为 true 时，去处理屏幕共享相关事件。<br>
5   * 可根据 {@link JRTCGuest#getShareStreamId shareRenderId} 和 {@link JRTCGuest#getShareUserId shareUserId} 属性进行屏幕共享画面的渲染和停止渲染。
6   * @param propChangeParam 通话改变的属性
7   */
8  onCallPropertyChanged?: (propChangeParam: JRTCRoomPropChangeParam) => void;
```

示例代码：

```
1 let param:JRTCRecordRemoteParam = new JRTCRecordRemoteParam();
2 param.recordVideo = true
3 ...
4 let map:HashMap<string,string> = new HashMap()
5 map.set("guest","juphoon001")
6 param.watermarkTextMap = map
7 if (guest.getRemoteRecordState() == JRTCRemoteRecordState.READY) {
8     //开启远程录制
9     call.enableRemoteRecord(true, param);
10 }
11
12 //关闭远程录制
13 call.enableRemoteRecord(false, new JRTCRecordRemoteParam());
14
15 //远程录制状态发生改变
16 onCallPropertyChanged: (propChangeParam: JRTCRoomPropChangeParam) => {
17     if (propChangeParam.recordState) {
18         if (call.getRemoteRecordState == JRTCRemoteRecordState.RUNNING) {
19             //当前正在远程录制
20         }
21     }
22 }
```

### 12.3.2 远程录制异常回调

详见 [onDeliveryAbort](#)。



```

1  /**
2   * 录制异常回调
3   * <p>
4   * 远程录制异常退出时会上报此回调。
5   *
6   * @param isShutDown 录制异常时服务器是否自动结束通话
7   *                   - true: 自动结束通话
8   *                   - false: 不自动结束通话
9   * @param deliveryUserId 录制异常的用户ID
10  * @param reason 录制异常的原因
11  * @param room 当前 JRTCRoom 对象
12  */
13  onDeliveryAbort?: (isShutDown: boolean, deliveryUserId: string, reason: string) => void;

```

### 12.3.3 水印

开启服务器音视频录制接口，支持配置文字水印，录制完成后视频保存于服务端。

Juphoon RTC SDK 支持以下三种画布水印设置：

- 文字水印：使用一段文字信息作为水印，支持设置字体和字号。
- 动态时间戳水印：使用当前时间戳作为水印，显示格式为“2021-03-18 14:30:35”。
- 静态图片水印：使用图片作为水印。

#### 12.3.3.1 注意事项

- 其中自定义水印内容通过 `JRTCRecordRemoteParam` 对象 `watermarkTextMap` 方法配置，服务端进行录制时将会把水印内容中的占位符替换成键值对中占位符对应Key的Value值。

```

1  /**
2   * 设置录制视频水印串
3   */
4  public set watermarkTextMap(watermarkTextMap: HashMap<string, string> | undefined);

```

#### 12.3.3.2 添加、修改或删除水印

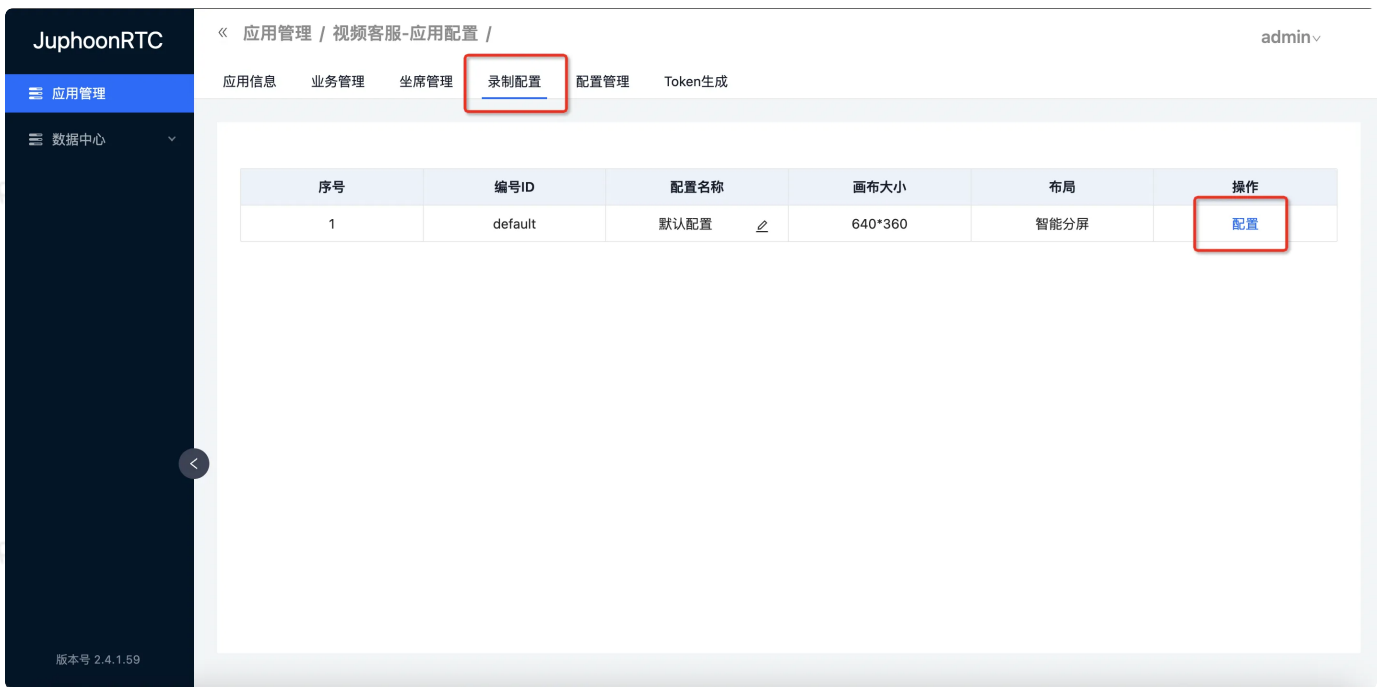
- 文字水印，如要求水印内容需具备客户经理工号、姓名、地理位置、经纬度；则需要终端开启录制时通过 `watermarkTextMap` 设置：{"userInfo": "工号+姓名"}、{"location": "地理位置"}、{"JWD": "经纬度"}这样的键值对；具体体现和系统管理平台端配置如下,红框内就是文字水印，支持在业务管理平台调整位置和字体颜色等，而`$_xxx$`最终会替换成键xxx对应的值，如`$_userInfo$`，替换成“工号+姓名”，`$_location$`，替换成“地理位置”，`$_JWD$`，替换成“经纬度”。
- 图片水印，在业务管理平台支持配置图片水印，目前知支持 png 格式图片，可以自由调整位置。
- 时间戳水印，在业务管理平台支持配置时间戳水印，可以自由调整位置，字体颜色、大小等。



### 12.3.4 自定义布局

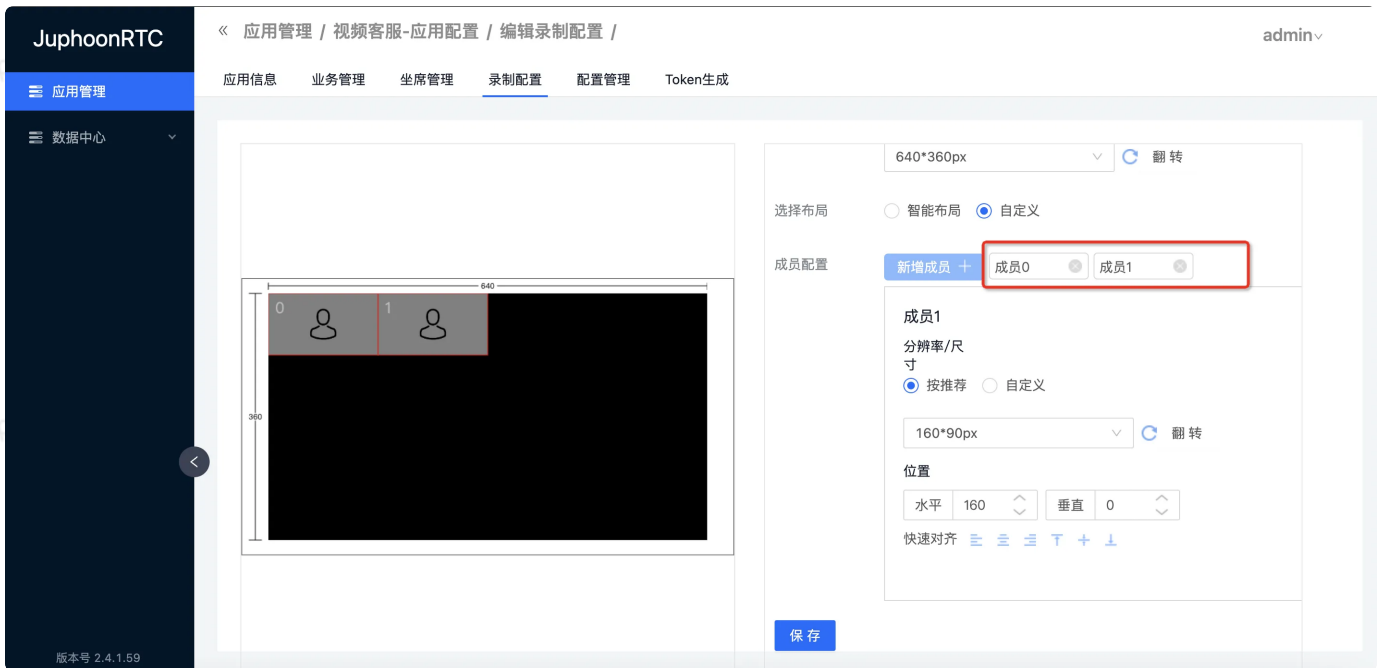
在系统管理平台端中的录制配置目录下，可以配置录制的一些参数，为了录制布局更加灵活，还提供了自定义布局，自定义布局可以配置每个录制流的位置和大小。

系统管理平台端默认使用的是智能分屏，如需切换自定义布局，在系统管理平台端修改如下：



配置每个成员的布局位置和大小,配置完成后点保存:

图中的0, 1表示每个窗口对应的窗口位置, 终端开启远程录制绑定视频流和窗口位置需要用到。



以上就是系统管理平台端的配置，实现自定义布局还需要在代码中绑定对应的窗口位置，参考代码如下：

```
ArkTS |  
  
1 let param:JRTCRecordRemoteParam = new JRTCRecordRemoteParam();  
2 param.frameRate = 24;  
3 param.iBitrate = 500;  
4 // 设置录制合并模式为自定义布局模式  
5 param.mergeMode = JRTCVideoMergeMode.CUSTOM_LAYOUT;  
6 param.videoWidth = 640;  
7 param.videoHeight = 360;  
8 param.recordVideo = true;  
9 param.layoutType = "default";//设置录制样式，对应业务管理平台上录制配置中的编号ID，不传则用默认  
10 // 开启远程录制  
11 call.enableRemoteRecord(true, param);
```

### 12.3.5 更新远程录制自定义布局

如果需要在远程录制已经开启的情况下更改录制布局，使用场景：比如录制中有新成员加入，需要调整布局位置，可以使用如下接口，需要在远程录制进行中调用，详见：[updateRemoteRecordLayout](#)

```

1  /**
2   * 更新远程录制自定义布局
3   *
4   * @param layoutList 需要更新的布局列表
5   * @return 接口调用结果
6   * - true: 接口调用成功
7   * - false: 接口调用异常
8   */
9  public abstract updateRemoteRecordLayout(layoutList: ArrayList<JRTCRecordRemoteLayout>): boolean;

```

示例代码：

```

1  let param:JRTCRecordRemoteParam = new JRTCRecordRemoteParam();
2  // .....远程录制参数设置
3  // 开启远程录制
4  call.enableRemoteRecord(true, param);
5  let layoutList:ArrayList<JRTCRecordRemoteLayout> = new ArrayList();
6  let layout:JRTCRecordRemoteLayout = new JRTCRecordRemoteLayout();
7  layout.setId("成员用户ID", true);
8  // 设置系统管理平台中对应的窗口位置
9  layout.position = 0;
10 layoutList.add(layout);
11 //.....设置其他成员视频流或者屏幕共享流对应窗口位置
12 //更新录制自定义布局
13 let result:boolean = call.updateRemoteRecordLayout(layoutList);
14 // 关闭远程录制
15 call.enableRemoteRecord(false, new JRTCRecordRemoteParam());

```

## 12.3.6 更新远程录制水印信息

如果需要在远程录制已经开启的情况下更改水印信息，使用场景：比如录制中有成员离开，需要修改该成员原先位置水印标签（可能是用户名字），可以使用如下接口，需要在远程录制进行中调用,详见：

[updateRemoteRecordWatermark](#)

```

1  /**
2   * 更新远程录制水印信息
3   *
4   * @param watermarkTextMap 水印信息
5   * @return 接口调用结果
6   * - true: 接口调用成功
7   * - false: 接口调用异常
8   */
9  public abstract updateRemoteRecordWatermark(watermarkTextMap: HashMap<string, string>): boolean;

```

示例代码：

```

1  let param:JRTCRecordRemoteParam = new JRTCRecordRemoteParam();
2  let watermarkTextMap1:HashMap<string, string> = new HashMap();
3  watermarkTextMap1.set("w1", "张三");
4  watermarkTextMap1.set("w2", "李四");
5  param.watermarkTextMap = watermarkTextMap1;//初始水印
6  // .....远程录制参数设置
7  // 开启远程录制
8  call.enableRemoteRecord(true, param);
9  let watermarkTextMap2:HashMap<string, string> = new HashMap();
10 watermarkTextMap2.set("w1", "李四");//修改水印信息
11 watermarkTextMap2.set("w2", "王五");//修改水印信息
12 let result:boolean = call.updateRemoteRecordWatermark(watermarkTextMap2);//更改录制水印信息
13 // 关闭远程录制
14 call.enableRemoteRecord(false, new JRTCRecordRemoteParam());

```

## 十三、加密传输

为了建立安全通道，就是通信双方建立连接通道后，在这个通道中传递的信息不可被第三方窃取，或者即使窃取后，也不能识别信息内容。但是仅通信双方建立安全通道是不够的，还需要进行合法性确认。

通常终端的合法性是基于用户名密码实现的，即服务器通过客户端提交的用户名和密码来进行鉴权，确认此用户的合法性。

在建立安全通道以及身份合法性验证后，后续大量的信息交互需要高效率的加密和解密，通常会使用对称加密方式进行处理。

## 13.1 国密加密

- 国密 SM4 分组密码算法是我国自主设计的分组对称密码算法，用于实现数据的加密/解密运算，以保证数据和信息的机密性。
- 要保证一个对称密码算法的安全性的基本条件是其具备足够的密钥长度，SM4 算法与 AES 算法具有相同的密钥长度分组长度128比特，因此在安全性上高于 3DES 算法。

在登录的接口参数里设置 CA 证书（Certificate）和账户分录（AccountEntry）。

通过 [certificate](#) 设置 Base64 编码后的证书字符串，通过 [accountEntry](#) 设置账户分录。

示例代码：

```
1  const loginParam: JRTCClientLoginParam = new JRTCClientLoginParam();
2  loginParam.certificate = "证书内容";
3  // 用户登录
4  client.login("juphoon", "123456", loginParam);
```

在加入会话房间时设置加密方式为 JRTCRoomSecurityType.SM4。

示例代码

```
1  let callJoinParam: JRTCJoinParam = new JRTCJoinParam();
2  callJoinParam.securityType = JRTCRoomSecurityType.SM4
3  mCall.join("10086", callJoinParam);
```

## 13.2 Token 校验

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。

```

1  /**
2   * 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
3   * <p>
4   * 登录结果通过 {@link JRTCClientCallback.onLogin onLogin} 回调通知
5   *
6   * @param { string } userId 用户ID
7   * @param { string } password 密码，不能为空
8   * @param { JRTCClientLoginParam? } clientLoginParam 登录参数，一般不需要设置，
   如需设置请问客服，传 undefined 则按默认值
9   * @return { boolean } 接口调用结果
10  *   - true: 接口调用成功
11  *   - false: 接口调用异常
12  * @warning 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
13  * @warning 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
14  */
15  public abstract login(userId: string, password: string, clientLoginParam?: JRTCClientLoginParam): boolean;

```

示例代码

```

1  let loginParam:JRTCClientLoginParam = new JRTCClientLoginParam();
2  loginParam.tokenType = "token校验类型";
3  loginParam.token = "token字符串";
4  client.login("userId", "密码", loginParam)

```

## 十四、视频多流

SDK 提供了可以自行创建/删除视频流通道的接口

使用场景举例：鉴于目前的房间模式只能由一人发起屏幕共享，当多个成员想在同个房间内发起屏幕共享时，可以使用该接口实现。



```
1  /**
2   * 创建额外视频流
3   *
4   * @param captureStreamId 本地视频流采集源流Id
5   * @param screenShare      是否屏幕共享（包括窗口共享、区域共享）
6   * @return 通话中的视频流ID 创建成功后，通话内其他成员将收到 {@link JRTCRoomCall
7   back#onParticipantUpdate} 回调
8   */
9  public abstract createExtraStream(captureStreamId: string, screenShare: bo
10 olean): string | undefined;
11
12  /**
13   * 删除额外视频流
14   *
15   * @param captureStreamId 本次视频流采集源流Id
16   * @return 接口调用结果
17   * - true: 接口调用成功，通话内其他成员将收到 {@link JRTCRoomCallback#onPartici
18   pantUpdate} 回调
19   * - false: 接口调用异常
20   */
21  public abstract deleteExtraStream(captureStreamId: string): boolean;
```

订阅/取消订阅额外视频流

```
1  /**
2   * 订阅房间中其他用户的额外视频流
3   *
4   * @param participant JRTCRoomParticipant 成员对象
5   * @param streamId    视频流ID
6   * @param videoSize   视频请求的尺寸
7   * @return 接口调用结果
8   * - true: 接口调用成功
9   * - false: 接口调用异常
10  */
11  public abstract requestExtraStreamVideo(participant: JRTCRoomParticipant,
12    streamId: string,
13    videoSize: JRTCVideoSize): boolean;
14  /**
15   * 取消订阅房间中其他用户的额外视频流
16   *
17   * @param participant JRTCRoomParticipant 成员对象
18   * @param streamId    视频流ID
19   * @return 接口调用结果
20   * - true: 接口调用成功
21   * - false: 接口调用异常
22   */
23  public abstract unRequestExtraStreamVideo(participant: JRTCRoomParticipant,
24    streamId: string): boolean;
```

示例代码（以屏幕共享举例）：

```
1 //发起端
2 //开启本地屏幕采集
3 mediaDevice.enableScreenCapture(true);
4
5 onScreenSharePermissionResult(result: boolean): void {
6     if (result) {
7         //创建视频额外流通道, 将本地屏幕采集流id绑定到该通道
8         let callStreamId:string = call.createExtraStream(mediaDevice.getScreenCaptureId(), true);
9         if (!TextUtils.isEmpty(callStreamId)) {
10             //开启多流屏幕共享成功
11         } else {
12             //开启多流屏幕共享失败
13         }
14     }
15 }
16
17 //删除视频额外流通道
18 call.deleteExtraStream(mediaDevice.getScreenCaptureId());
19 //关闭本地屏幕采集
20 mediaDevice.enableScreenCapture(false);
21
22 onParticipantUpdate: (participant: JRTCRoomParticipant, changeParam: JRTCRoomParticipantChangeParam) => {
23
24     if (changeParam.extraStream) {
25         //判断不是本端发起的多流
26         if (participant.userId != JRTCManager.getInstance().client.getUserId()) {
27             //遍历新增的视频流
28             for (const stream of changeParam.addedExtraStreams) {
29                 //订阅该视频流
30                 call.requestExtraStreamVideo(participant, stream, new JRTCVideoSize(1280, 720));
31             }
32             this.updateVideoUi();
33             //删除额外流不为空
34             if (changeParam.removedExtraStreams.length > 0 ) {
35                 //遍历移除的视频流
36                 for (const stream of changeParam.removedExtraStreams) {
37                     //这里不需要去取消订阅
38                     //可有将该视频画面从视图容器移除
39                 }
40             }
41         }
42     }
```

```
42 }  
43 }
```

杨象坤(1664699)

杨象坤(1664699)