ALIBABA CLOUD

阿里云

音视频通信
常用功能

文档版本：20210326

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.生成Token

RTC为您提供两种生成Token的方式，通过阅读本文，您可以了解控制台和服务端生成Token的方法。

前提条件

- 您已经开通RTC服务，具体操作，请参见开通服务。
- 您已经创建好应用，具体操作，请参见创建应用。
- 您已经获取AppKey，具体操作，请参见查询AppKey

背景信息

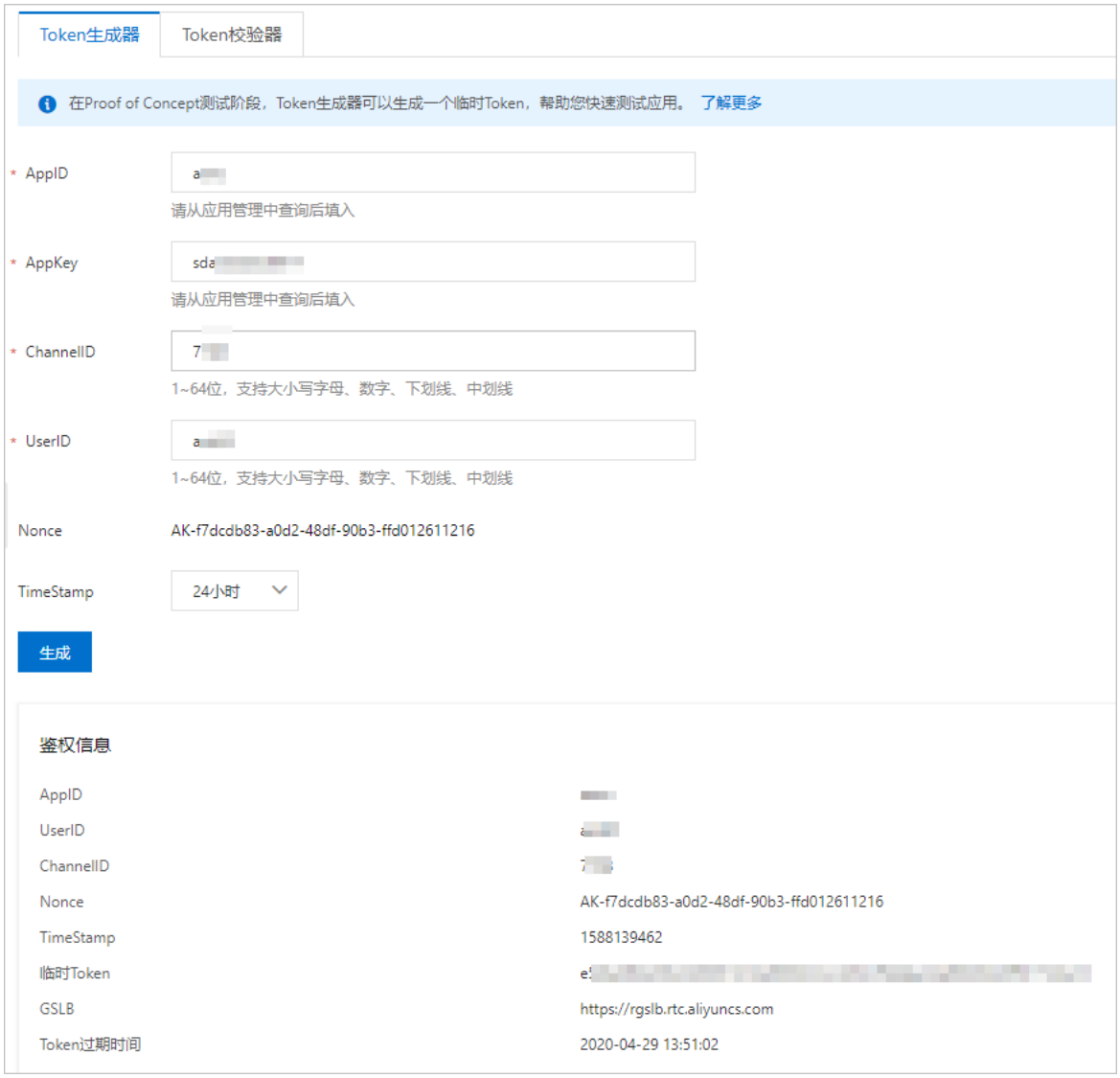
Token是阿里云设计的一种安全保护签名，目的是为了阻止恶意攻击者盗用您的云服务使用权。您需要在相应SDK的登录函数中提供AppID、UserID、ChannelId、Nonce、TimeStamp、GSLB和Token信息。其中AppID用于标识您的应用，UserID用于标识您的用户，而Token则是基于两者通过SHA256加密算法计算得出。因此，攻击者很难通过伪造Token盗用您的云服务流量。

控制台

1. 登录音视频通信RTC控制台。
2. 在左侧导航栏选择**接入工具**。
3. 在**Token生成器**页签下，输入生成Token所需要的参数。

参数	描述
AppID	应用ID，在控制台**应用管理**页面创建和查看。
AppKey	应用AppKey，在控制台**应用管理**页面查询。
ChannelId	频道ID。1~64位，由大小写字母、数字、下划线（_）、短划线（-）组成。
UserId	用户ID。1~64位，由大小写字母、数字、下划线（_）、短划线（-）组成。
Nonce	随机码。以前缀AK-开头，由大小写字母、数字组成，最大64字节。例如：AK-2b9be4b25c2d38c409c376ffd2372be1。
TimeStamp	过期时间戳。可以选择12小时、24小时、3天和7天，代表令牌有效时间。

4. 单击**生成**。



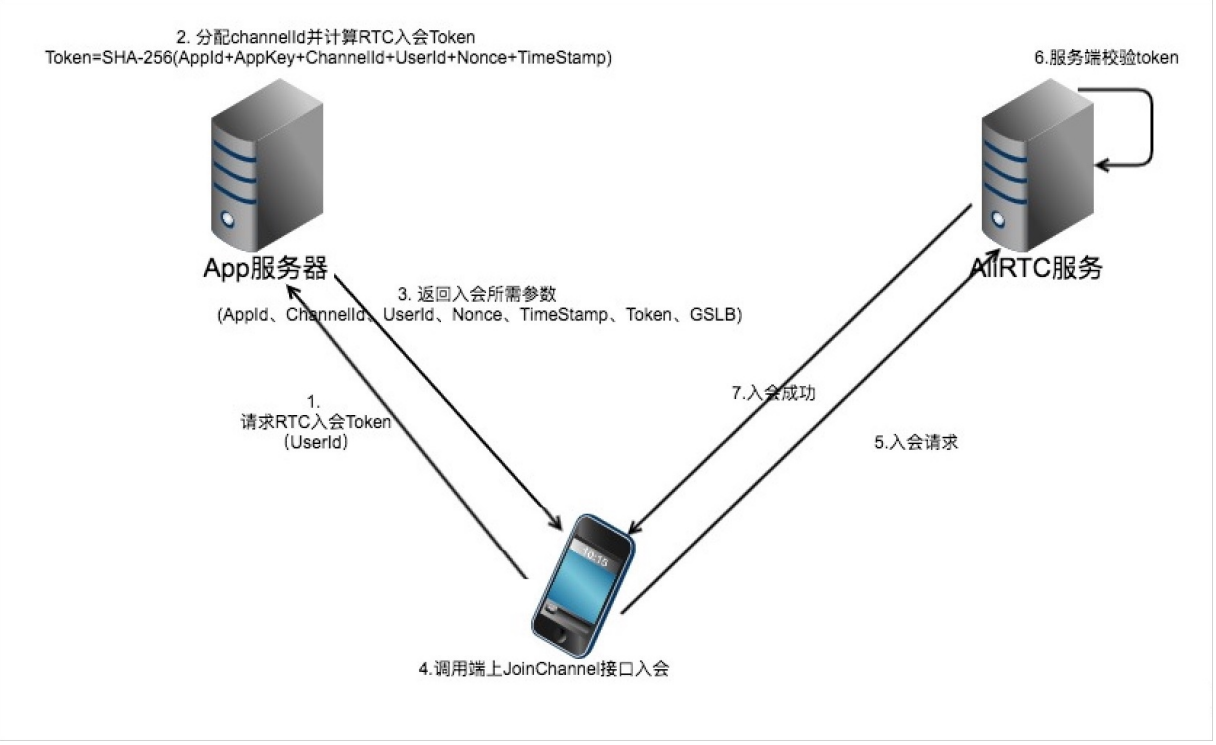
服务端

相对于控制台生成Token，服务端生成Token可以最大限度地保障计算Token的密钥不被泄露，具体的流程如下所示：

1. 您的App在调用SDK的初始化函数之前，首先要向您的服务器请求Token。
2. 您的服务器根据如下参数计算Token。

   ```
   token = sha256(appId + appKey + channelId + userId + nonce + timestamp)
   ```

3. 服务器将计算好的鉴权信息返回给您的App。
4. 您的App将获得的鉴权信息通过特定API传递给SDK。
5. SDK将鉴权信息提交给阿里云服务器进行校验。
6. 阿里云校验鉴权信息，确认合法性。
7. 校验通过后，即可开始提供实时音视频服务。



参数	说明
AppID	应用ID，通过控制台创建。
UserID	您的唯一标识，由AppServer生成。同一个UserId的用户在其他端登录，先入会的端会被后入会的端踢出房间。 由大小写字母、数字组成，最大64字节。例如：2b9be4b25c2d38c409c376ffd2372be1。
ChannelID	频道ID，AppServer生成。不支持设置ChannelID为0，并且ChannelID不可以重复，需要保持ChannelID的唯一。由大小写字母、数字、短划线（-）组成，最大64字节。例如：181-218-3406。
Nonce	令牌随机码，由AppServer生成。以前缀AK-开头，由大小写字母、数字组成，最大64字节。例如：AK-2b9be4b25c2d38c409c376ffd2372be1。
Timestamp	令牌过期时间戳，例如：1560588594代表过期时间为2019-06-15 16:49:54。
Token	加入频道的Token，由AppServer生成。实际算法为 `sha256(appId + appKey + channelId + userId + nonce + timestamp)` 。
GSLB	服务地址，该参数是数组类型，当前请使用：["https://rgslb.rtc.aliyuncs.com"]，请您通过业务服务器下发到客户端SDK，不建议您将该地址固化在客户端代码。

服务端生成Token的签名算法为SHA256，您可以参见如下版本的生成Token函数：

- Golang程序实例请查看 `CreateToken` 函数，更多信息，请参见Golang Demo。
- Java程序实例请查看 `createToken` 函数，更多信息，请参见Java Demo。
- Python程序实例请查看 `create_token` 函数，更多信息，请参见Python Demo。

- C#程序实例请查看 `CreateToken` 函数，更多信息，请参见C# Demo。
- Nodejs程序实例请查看 `CreateToken` 函数，更多信息，请参见Node.js Demo。
- PHP程序实例请参查看 `CreateToken` 函数，更多信息，请参见PHP Demo。

2.设置音频属性

RTC SDK您提供设置音质和场景的功能，您可以根据实际情况通过搭配音质和场景设置音频属性，以达到更好的产品体验。通过阅读本文，您可以了解设置音频属性的方法。

功能简介

RTC SDK提供了三种音质模式（音频Profile规格）和四种场景（音频Scene规格）供您选择，如下所示：

音频Profile规格

音频Profile关键字	名称	声道数/采样率/编码码率
ENGINE_LOW_QUALITY_MODE	低功耗音质模式	1/8kHz/12kbps
ENGINE_BASIC_QUALITY_MODE	标准音质模式（默认模式）	1/16kHz/24kbps
ENGINE_HIGH_QUALITY_MODE	高音质模式	1/48kHz/48kbps

音频Scene规格

场景关键字	名称	特性
SCENE_DEFAULT_MODE	默认场景	推荐一般的音视频通信场景使用。
SCENE_EDUCATION_MODE	教育场景	优先保证音频连续性与稳定性。
SCENE_MEDIA_MODE	媒体场景	保真人声与音乐音质，推荐连麦直播间使用。
SCENE_MUSIC_MODE	音乐场景	高保真音乐音质，推荐乐器教学等对音乐音质有要求的场景使用。

参数搭配推荐

您可以根据推荐进行参数搭配，也可以根据自身实际业务场景进行自定义搭配。例如社交场景中，有背景音乐的音质需求，Scene可以选择音乐场景，如果没有背景音乐需求，Scene场景可以选择默认背景。

业务场景	Profile设置	Scene设置	特性
音视频通话	标准音质模式（ENGINE_BASIC_QUALITY_MODE）	默认场景（SCENE_DEFAULT_MODE）	在保证高清音质的同时，保证传输稳定流畅。
音频社交场景	高音质模式（ENGINE_HIGH_QUALITY_MODE）	音乐场景（SCENE_MUSIC_MODE）	高保真音乐音质，乐器教学等对音乐音质有要求的场景推荐使用。
音频社交场景2	高音质模式（ENGINE_HIGH_QUALITY_MODE）	默认场景（SCENE_DEFAULT_MODE）	高保真人声，没有背景音乐需求。

业务场景	Profile设置	Scene设置	特性
教育场景	标准音质模式（ENGINE_BASIC_QUALITY_MODE）	教育场景（SCENE_EDUCATION_MODE）	优先保证音频连续性与稳定性，同时保证音质高清。

实现方法

Android、iOS、Mac和Windows SDK平台关于音频的Profile设置与Scene设置通过集成RTC SDK时传递的extras参数设置，extras参数以JSON格式体现，如下所示：

```
{"user_specified_ engine_mode" : "Profile设置关键字","user_specified_scene_mode" :"Scene场景关键字"}
```

示例代码如下所示：

> **说明** 示例代码中SDK的初始化extras参数只包含了音频设置，实际代码设置过程中，还可以包含其他模块功能的初始化设置。

- Android

```
JSONObject jsonObject = new JSONObject();
//配置音质模式
try {
  jsonObject.put("user_specified_engine_mode", "ENGINE_BASIC_QUALITY_MODE");
  //配置场景模式
  jsonObject.put("user_specified_scene_mode", "SCENE_DEFAULT_MODE");
  AliRtcEngine aliRtcEngine = AliRtcEngine.getInstance(getApplicationContext(),jsonObject);
} catch (JSONException e) {
  e.printStackTrace();
}
```

- iOS

```
@property (nonatomic, strong) AliRtcEngine *engine;
NSMutableDictionary *extraDic = [[NSMutableDictionary alloc] init];
[extraDic setValue:@"ENGINE_BASIC_QUALITY_MODE" forKey:@"user_specified_engine_mode"];
[extraDic setValue:@"SCENE_EDUCATION_MODE" forKey:@"user_specified_scene_mode"];
NSError *error = nil;
NSData *json = [NSJSONSerialization dataWithJSONObject:extraDic options:NSJSONWritingPrettyPrinted error:&error];
NSString *string = [[NSString alloc] initWithData:json encoding:NSUTF8StringEncoding];
_engine = [AliRtcEngine sharedInstance:self extras: string];
```

- Mac

```
@property (nonatomic, strong) AliRtcEngine *engine;
NSMutableDictionary *extraDic = [[NSMutableDictionary alloc] init];
[extraDic setValue:@"ENGINE_BASIC_QUALITY_MODE" forKey:@"user_specified_engine_mode"];
[extraDic setValue:@"SCENE_EDUCATION_MODE" forKey:@"user_specified_scene_mode"];
NSError *error = nil;
NSData *json = [NSJSONSerialization dataWithJSONObject:extraDic options:NSJSONWritingPrettyPrinte
d error:&error];
NSString *string = [[NSString alloc] initWithData:json encoding:NSUTF8StringEncoding];
_engine = [AliRtcEngine sharedInstance:self extras: string];
```

- Windows

```
std::string GetEngAndsceneMode(const std::string strEngMode, const std::string strSceneMode)
{
 std::string strParam = "{";
 if (strEngMode == "default")
 {
  strParam += "\"user_specified_engine_mode\" : \"ENGINE_BASIC_QUALITY_MODE\",";
 }
 if (strSceneMode == "default")
 {
  strParam += "\"user_specified_scene_mode\" : \"SCENE_DEFAULT_MODE\",";
 }
 strParam += "}";
 return strParam;
}
AliRtcEventListener* rtcEventlister;
std::string strEngMode = "default";
std::string strSceneMode = "default";
AliRtcEngine *mpEngine = AliRtcEngine::sharedInstance(rtcEventlister, GetEngAndsceneMode(strEngMod
e, strSceneMode).c_str());
```

3.设置视频属性

3.1. Android

RTC SDK为您提供设置视频流规格和类型的功能，您可以根据实际情况通过搭配视频流规格和类型设置视频属性，以达到更好的产品体验。通过阅读本文，您可以了解设置视频属性的方法。

功能简介

在音视频通信中，根据您的喜好和实际情况设置视频属性，调整视频画面的清晰度和流畅度。如果是一对一视频通信，您可以将分辨率和帧率调高，如果频道内有多个用户进行视频通信，您可以将分辨率和码率适当调低，以减少编解码的资源消耗和缓解下行带宽压力。视频属性包含视频流规格、视频流类型，如下所示：

视频流规格

枚举名	描述
AliRTCSDK_Video_Profile_Default	默认，分辨率480*640，帧率15。
AliRTCSDK_Video_Profile_180_240P_15	分辨率180*240，帧率15。
AliRTCSDK_Video_Profile_180_320P_15	分辨率180*320，帧率15。
AliRTCSDK_Video_Profile_180_320P_30	分辨率180*320，帧率30。
AliRTCSDK_Video_Profile_240_320P_15	分辨率240*320，帧率15。
AliRTCSDK_Video_Profile_360_480P_15	分辨率360*480，帧率15。
AliRTCSDK_Video_Profile_360_480P_30	分辨率360*480，帧率30。
AliRTCSDK_Video_Profile_360_640P_15	分辨率360*640，帧率15。
AliRTCSDK_Video_Profile_360_640P_30	分辨率360*640，帧率30。
AliRTCSDK_Video_Profile_480_640P_15	分辨率480*640，帧率15。
AliRTCSDK_Video_Profile_480_640P_30	分辨率480*640，帧率30。
AliRTCSDK_Video_Profile_720_960P_15	分辨率720*960，帧率15。
AliRTCSDK_Video_Profile_720_960P_30	分辨率720*960，帧率30。
AliRTCSDK_Video_Profile_720_1280P_15	分辨率720*1280，帧率15。
AliRTCSDK_Video_Profile_720_1280P_30	分辨率720*1280，帧率30。
AliRTCSDK_Video_Profile_360_640P_15_800Kb	360*640分辨率，帧率15，800Kb码率。
AliRTCSDK_Video_Profile_480_840P_15_500Kb	480*840分辨率，帧率15，500Kb码率。
AliRTCSDK_Video_Profile_480_840P_15_800Kb	480*840分辨率，帧率15，800Kb码率。

枚举名	描述
AliRTCSDK_Video_Profile_540_960P_15_800Kb	540*960分辨率，帧率15，800Kb码率。
AliRTCSDK_Video_Profile_540_960P_15_1200Kb	540*960分辨率，帧率15，1200Kb码率。
AliRTCSDK_Video_Profile_Max	占位值。

视频流类型

枚举名	描述
AliRtcVideoTrackNo	无视频流。
AliRtcVideoTrackCamera	相机流。
AliRtcVideoTrackScreen	屏幕共享流。
AliRtcVideoTrackBoth	相机流和屏幕共享流。

实现方法

RTC SDK通过 `setVideoProfile` 方法设置视频属性。

```
public abstract void setVideoProfile(AliRtcVideoProfile profile, AliRtcVideoTrack track)
```

名称	类型	描述
profile	AliRtcVideoProfile	视频流参数。默认分辨率480*640，帧率15的相机流。
track	AliRtcVideoTrack	需要设置的视频流类型，默认相机流。

3.2. iOS和Mac

RTC SDK为您提供设置视频流规格和类型的功能，您可以根据实际情况通过搭配视频流规格和类型设置视频属性，以达到更好的产品体验。通过阅读本文，您可以了解设置视频属性的方法。

功能简介

在音视频通信中，根据您的喜好和实际情况设置视频属性，调整视频画面的清晰度和流畅度。如果是一对一视频通信，您可以将分辨率和帧率调高，如果频道内有多个用户进行视频通信，您可以将分辨率和码率适当调低，以减少编解码的资源消耗和缓解下行带宽压力。视频属性包含视频流规格、视频流类型，如下所示：

视频流规格

枚举名	描述
AliRtcVideoProfile_Default	默认，分辨率480*640，帧率15。
AliRtcVideoProfile_180_240P_15	分辨率180*240，帧率15。

枚举名	描述
AliRtcVideoProfile_180_320P_15	分辨率180*320，帧率15。
AliRtcVideoProfile_180_320P_30	分辨率180*320，帧率30。
AliRtcVideoProfile_240_320P_15	分辨率240*320，帧率15。
AliRtcVideoProfile_360_480P_15	分辨率360*480，帧率15。
AliRtcVideoProfile_360_480P_30	分辨率360*480，帧率30。
AliRtcVideoProfile_360_640P_15	分辨率360*640，帧率15。
AliRtcVideoProfile_360_640P_30	分辨率360*640，帧率30。
AliRtcVideoProfile_480_640P_15	分辨率480*640，帧率15。
AliRtcVideoProfile_480_640P_30	分辨率480*640，帧率30。
AliRtcVideoProfile_720_960P_15	分辨率720*960，帧率15。
AliRtcVideoProfile_720_960P_30	分辨率720*960，帧率30。
AliRtcVideoProfile_720_1280P_15	分辨率720*1280，帧率15。
AliRtcVideoProfile_720_1280P_30	分辨率720*1280，帧率30。
AliRtcVideoProfile_1080_1920P_15	分辨率1080*1920，帧率15。
AliRtcVideoProfile_1080_1920P_30	分辨率1080*1920，帧率30。
AliRtcVideoProfile_480_640P_15_1500Kb	分辨率480*640，帧率15，1500Kb码率。
AliRtcVideoProfile_900_1600P_20	分辨率900*1600，帧率20。
AliRtcVideoProfile_360_640P_15_800Kb	分辨率360*640，帧率15，800Kb码率。
AliRtcVideoProfile_480_840P_15_500Kb	分辨率480*840，帧率15，500Kb码率。
AliRtcVideoProfile_480_840P_15_800Kb	分辨率480*840，帧率15，800Kb码率。
AliRtcVideoProfile_540_960P_15_800Kb	分辨率540*960，帧率15，800Kb码率。
AliRtcVideoProfile_540_960P_15_1200Kb	分辨率540*960，帧率15，1200Kb码率。
AliRtcVideoProfile_Max	占位值。

视频流类型

枚举名	描述
AliRtcVideoTrackNo	无摄像头和屏幕共享流。

枚举名	描述
AliRtcVideoTrackCamera	相机流。
AliRtcVideoTrackScreen	屏幕共享流。
AliRtcVideoTrackBoth	相机流和屏幕共享流。

实现方法

RTC SDK通过 setVideoProfile 方法设置视频属性。

```
- (void)setVideoProfile:(AliRtcVideoProfile)profile forTrack:(AliRtcVideoTrack)track;
```

名称	类型	描述
profile	AliRtcVideoProfile	视频流参数。默认为分辨率480*640，帧率15的相机流。
track	AliRtcVideoTrack	需要设置的视频Track类型。默认相机流。

3.3. Windows

RTC SDK为您提供设置视频流规格和类型的功能，您可以根据实际情况通过搭配视频流规格和类型设置视频属性，以达到更好的产品体验。通过阅读本文，您可以了解设置视频属性的方法。

功能简介

在音视频通信中，根据您的喜好和实际情况设置视频属性，调整视频画面的清晰度和流畅度。如果是一对一视频通信，您可以将分辨率和帧率调高，如果频道内有多个用户进行视频通信，您可以将分辨率和码率适当调低，以减少编解码的资源消耗和缓解下行带宽压力。视频属性包含视频流规格、视频流类型，如下所示：

视频流规格

枚举名	描述
AliRtcVideoProfile_Default	默认，分辨率480*640，帧率15。
AliRtcVideoProfile_180_240P_15	分辨率180*240，帧率15。
AliRtcVideoProfile_180_320P_15	分辨率180*320，帧率15。
AliRtcVideoProfile_180_320P_30	分辨率180*320，帧率30。
AliRtcVideoProfile_240_320P_15	分辨率240*320，帧率15。
AliRtcVideoProfile_360_480P_15	分辨率360*480，帧率15。
AliRtcVideoProfile_360_480P_30	分辨率360*480，帧率30。
AliRtcVideoProfile_360_640P_15	分辨率360*640，帧率15。

枚举名	描述
AliRtcVideoProfile_360_640P_30	分辨率360*640，帧率30。
AliRtcVideoProfile_480_640P_15	分辨率480*640，帧率15。
AliRtcVideoProfile_480_640P_30	分辨率480*640，帧率30。
AliRtcVideoProfile_720_960P_15	分辨率720*960，帧率15。
AliRtcVideoProfile_720_960P_30	分辨率720*960，帧率30。
AliRtcVideoProfile_720_1280P_15	分辨率720*1280，帧率15。
AliRtcVideoProfile_720_1280P_30	分辨率720*1280，帧率30。
AliRtcVideoProfile_1080_1920P_15	分辨率1080*1920，帧率15。
AliRtcVideoProfile_1080_1920P_30	分辨率1080*1920，帧率30。
AliRtcVideoProfile_480_640P_15_1500Kb	分辨率480*640，帧率15，1500Kb码率。
AliRtcVideoProfile_900_1600P_20	分辨率900*1600，帧率20。
AliRtcVideoProfile_360_640P_15_800Kb	分辨率360*640，帧率15，800Kb码率。
AliRtcVideoProfile_480_840P_15_500Kb	分辨率480*840，帧率15，500Kb码率。
AliRtcVideoProfile_480_840P_15_800Kb	分辨率480*840，帧率15，800Kb码率。
AliRtcVideoProfile_540_960P_15_800Kb	分辨率540*960，帧率15，800Kb码率。
AliRtcVideoProfile_540_960P_15_1200Kb	分辨率540*960，帧率15，1200Kb码率。
AliRtcVideoProfile_540_960P_20	分辨率540*960，帧率20。
AliRtcVideoProfile_720_1280P_20	分辨率720*1280，帧率20。
AliRtcVideoProfile_1080_1920P_20	分辨率1080*1920，帧率20。

视频流类型

枚举名	描述
AliRtcVideoTrackNo	无视频流。
AliRtcVideoTrackCamera	摄像头流。
AliRtcVideoTrackScreen	屏幕共享流。
AliRtcVideoTrackBoth	摄像头和屏幕共享。

实现方法

RTC SDK通过 setVideoProfile 方法设置视频属性。

```
void setVideoProfile(AliRtcVideoProfile profile, AliRtcVideoTrack track)
```

名称	类型	描述
profile	AliRtcVideoProfile	视频流参数。默认为AliRtcVideoProfile_Default。
track	AliRtcVideoTrack	需要设置的视频流类型。

3.4. Web

RTC SDK为您提供设置视频流规格和类型的功能，您可以根据实际情况通过搭配视频流规格和类型设置视频属性，以达到更好的产品体验。通过阅读本文，您可以了解设置视频属性的方法。

功能简介

在音视频通信中，根据您的喜好和实际情况设置视频属性，调整视频画面的清晰度和流畅度。如果是一对一视频通信，您可以将分辨率和帧率调高，如果频道内有多个用户进行视频通信，您可以将分辨率和码率适当调低，以减少编解码的资源消耗和缓解下行带宽压力。视频属性包含视频流规格、视频流类型。

实现方法

设置视频属性之前，您需要先调用getAvailableResolutions传入摄像头参数返回支持的分辨率和帧率，再通过 setVideoProfile 方法设置视频属性，调用publish才能生效。

```
aliWebrtc.setVideoProfile({
    width,
    height,
    frameRate,
  },type);
```

名称		类型	描述
	width	Number	宽度。取值： • 摄像头：640（默认值）。 • 屏幕共享：960（默认值）。
	height	Number	高度。取值： • 摄像头：480（默认值）。 • 屏幕共享：540（默认值）。

名称		类型	描述
config	frameRate	Number	帧率。取值范围：5~30。 默认取值： • 摄像头：15（默认值）。 • 屏幕共享：10（默认值）。
	maxBitrate	Number	最大码率。取值： • 摄像头：500000（默认值）。 • 屏幕共享：1500000（默认值）。 **注意** 1.13.2版本已删除该参数，SDK会根据设置的分辨率和帧率自动设置最大码率。1.13.2以下版本如果调用setVideoProfile，还需要配置该参数。
type		Number	1表示摄像头，2表示屏幕共享。

4.设备检测和管理

4.1. Android

RTC SDK为您提供了设备检测和管理的功能，您可以在加入频道之前检查硬件设备是否能正常工作。通过阅读本文，您可以了解设备检测和管理的方法。

功能简介

RTC SDK通过调用内部方法实现设备检测和管理。例如，您可以查询设备信息、检测摄像头是否正常工作、检测音频设备是否正常录音及播放、设置摄像头方向或者切换音频设备（麦克风和扬声器）等。

实现方法

以下为常用的设备检测和管理方法，更多信息，请参见AliRtcEngine接口。

- getCurrentCameraType：获取当前摄像头类型。

  返回摄像头类型AliRTCCameraType。

  ```
  public abstract AliRTCCameraType getCurrentCameraType()
  ```

- isCameraOn：检查摄像头是否打开。

  ```
  public abstract boolean isCameraOn()
  ```

  返回说明

  true表示摄像头已打开，false表示摄像头未打开。

- isSpeakerOn：查询是否开启扬声器。

  ```
  public abstract boolean isSpeakerOn()
  ```

  返回说明

  true表示已开启扬声器，false表示未开启扬声器。

- setPreCameraType：预设值摄像头方向。

  ```
  public abstract void setPreCameraType(int faceTo)
  ```

  参数说明

名称	类型	描述
faceTo	int	0表示后置，1表示前置（默认值为1）。

- getPreCameraType：获取预设值摄像头方向。

  ```
  public abstract int getPreCameraType()
  ```

  返回说明

  0表示后置摄像头，1表示前置摄像头。

- setCameraZoom：设置摄像头参数。

```
public abstract int setCameraZoom(float zoom, boolean flash, boolean autoFocus)
```

参数说明

名称	类型	描述
zoom	float	zoom变焦的级别（默认值：1.0）。
flash	boolean	true表示打开闪光灯，false表示不打开闪光灯。默认不打开闪光灯。
autoFocus	boolean	true表示打开自动对焦，false表示不打开自动对焦。默认不打开自动对焦。

返回说明

0表示设置成功，其他表示设置失败。

- enableSpeakerphone：切换听筒、扬声器输出。

```
public abstract int enableSpeakerphone(boolean enable)
```

参数说明

名称	类型	描述
enable	boolean	true为扬声器模式，false为听筒模式。默认扬声器模式。

返回说明

0表示方法调用成功，其他表示方法调用失败。

> 注意 该接口只能在主线程调用。

4.2. iOS

RTC SDK为您提供了设备检测和管理的功能，您可以在加入频道之前检查硬件设备是否能正常工作。通过阅读本文，您可以了解设备检测和管理的方法。

功能简介

RTC SDK通过调用内部方法实现设备检测和管理。例如，您可以查询设备信息、检测摄像头是否正常工作、检测音频设备是否正常录音及播放、设置摄像头方向或者切换音频设备（麦克风和扬声器）等。

实现方法

以下为常用的设备检测和管理方法，更多信息，请参见AliRtcEngine接口。

- switchCamera（仅iOS可用）：切换前后摄像头。

```
- (int)switchCamera;
```

返回说明

0表示切换成功，其他表示切换失败。

- setCameraZoom（仅iOS可用）：设置摄像头参数。

```
- (int)setCameraZoom:(float)zoom flash:(BOOL)flash autoFocus:(BOOL)autoFocus;
```

参数说明

名称	类型	描述
zoom	float	变焦的级别。取值范围：-3.0~3.0，默认取值1.0。
flash	BOOL	是否打开闪光灯。YES为打开闪光灯，NO为不打开闪光灯。默认不打开闪光灯。
autoFocus	BOOL	是否打开自动对焦。YES为打开自动对焦，NO为不打开自动对焦。默认不打开自动对焦。

返回说明

0表示设置成功，其他表示设置失败。

- isCameraOn（仅iOS可用）：检查摄像头是否打开。

```
- (BOOL)isCameraOn;
```

返回说明

YES表示摄像头已打开，NO表示摄像头没有打开。

- enableSpeakerphone（仅iOS可用）：设置音频输出为听筒还是扬声器。

```
- (int)enableSpeakerphone:(BOOL)enable;
```

参数说明

名称	类型	描述
enable	BOOL	YES为扬声器模式，NO为听筒模式（默认值）。

返回说明

0表示成功，其他返回错误码。

4.3. Mac

RTC SDK为您提供了设备检测和管理的功能，您可以在加入频道之前检查硬件设备是否能正常工作。通过阅读本文，您可以了解设备检测和管理的方法。

功能简介

RTC SDK通过调用内部方法实现设备检测和管理。例如，您可以查询设备信息、检测摄像头是否正常工作、检测音频设备是否正常录音及播放、设置摄像头方向或者切换音频设备（麦克风和扬声器）等。

实现方法

以下为常用的设备检测和管理方法，更多信息，请参见AliRtcEngine接口。

- getCameraList（仅Mac可用）：获取摄像头列表。

```
- (NSArray<AliRtcDeviceInfo *> *)getCameraList;
```

- getCurrentCamera（仅Mac可用）：获取当前使用的摄像头名称。

```
- (NSString *)getCurrentCamera;
```

- setCurrentCamera（仅Mac可用）：选择摄像头。

```
- (void)setCurrentCamera:(NSString *)camera;
```

参数说明

名称	类型	描述
camera	NSString *	摄像头名称。

> 注意 必须先调用getCameraList接口获取设备列表后再调用此接口设置。

- getAudioCaptures（仅Mac可用）：获取系统中的录音设备列表。

```
- (NSArray<AliRtcDeviceInfo *> *)getAudioCaptures;
```

- getCurrentAudioCapture（仅Mac可用）：获取当前使用的音频采集设备名称。

```
- (NSString *)getCurrentAudioCapture;
```

- setCurrentAudioCapture（仅Mac可用）：选择音频采集设备。

```
- (void)setCurrentAudioCapture:(NSString *)capture;
```

参数说明

名称	类型	描述
capture	NSString *	音频采集设备名称。

> 注意 必须先调用getCurrentAudioCapture接口获取设备列表后再调用此接口设置。

- getAudioRenderers（仅Mac可用）：获取系统中的扬声器列表。

```
- (NSArray<AliRtcDeviceInfo *> *)getAudioRenderers;
```

- getCurrentAudioRenderer（仅Mac可用）：获取当前使用的音频播放设备。

```
- (NSString *)getCurrentAudioRenderer;
```

- setCurrentAudioRenderer（仅Mac可用）：选择音频播放设备。

```
- (void)setCurrentAudioRenderer:(NSString *)renderer;
```

参数说明

名称	类型	描述
renderer	NSString *	音频播放设备名称。

> 注意 必须先调用getAudioRenderers接口获取设备列表后再调用此接口设置。

4.4. Windows

RTC SDK为您提供了设备检测和管理的功能，您可以在加入频道之前检查硬件设备是否能正常工作。通过阅读本文，您可以了解设备检测和管理的方法。

功能简介

RTC SDK通过调用内部方法实现设备检测和管理。例如，您可以查询设备信息、检测摄像头是否正常工作、检测音频设备是否正常录音及播放、设置摄像头方向或者切换音频设备（麦克风和扬声器）等。

实现方法

以下为常用的设备检测和管理方法，更多信息，请参见AliRtcEngine接口。

- getCameraList：获取摄像头列表。

```
void getCameraList(AliRtc::StringArray& array)
```

参数说明

名称	类型	描述
array	AliRtc::StringArray&	摄像头列表。

- getCurrentCamera：获取当前使用的摄像头名称。

```
AliRtc::String getCurrentCamera()
```

- setCurrentCamera：选择摄像头。

```
void setCurrentCamera(const AliRtc::String& camera)
```

参数说明

名称	类型	描述
camera	const AliRtc::String&	摄像头名称。

> 注意 该接口只可在调用getCameraList接口获取设备列表后才可调用。

- isCameraOn：检查摄像头是否打开。

```
bool isCameraOn()
```

返回说明

true表示摄像头已打开，false表示摄像头未打开。

- getAudioCaptures：获取系统中的录音设备列表。

```
void getAudioCaptures(AliRtc::StringArray& array)
```

参数说明

名称	类型	描述
array	AliRtc::StringArray&	音频采集设备列表。

- getCurrentAudioCapture：获取当前使用的音频采集设备名称。

```
AliRtc::String getCurrentAudioCapture()
```

- setCurrentAudioCapture：选择音频采集设备。

```
void setCurrentAudioCapture(const AliRtc::String& capture)
```

参数说明

名称	类型	描述
capture	String	音频采集设备名称。

> 注意 该接口只可在getAudioCaptures接口后调用。

- getAudioRenderers：获取系统中的扬声器列表。

```
void getAudioRenderers(AliRtc::StringArray& array)
```

参数说明

名称	类型	描述
array	AliRtc::StringArray&	音频播放设备列表。

- getCurrentAudioRenderer：获取当前使用的音频播放设备。

```
AliRtc::String getCurrentAudioRenderer()
```

- setCurrentAudioRenderer：选择音频播放设备。

```
void setCurrentAudioRenderer(const AliRtc::String &renderer)
```

参数说明

名称	类型	描述
renderer	String	音频播放设备名称。

注意 该接口只可在getAudioRenderers后调用。

4.5. Web

RTC SDK为您提供了设备检测和管理的功能，您可以在加入频道之前检查硬件设备是否能正常工作。通过阅读本文，您可以了解设备检测和管理的方法。

功能简介

RTC SDK通过调用内部方法实现设备检测和管理。例如，您可以查询设备信息、检测摄像头是否正常工作、检测音频设备是否正常录音及播放、设置摄像头方向或者切换音频设备（麦克风和扬声器）等。

实现方法

以下为常用的设备检测和管理方法，更多信息，请参见AliRtcEngine接口。

getDevices：获取设备信息，返回摄像头和音频输入设备。如果外接设备重新插拔后获取不到设备信息，请尝试重新启动电脑。

```
aliWebrtc.getDevices().then((re)=>{
}).catch((error)=>{
 console.log(error.message)
});
```

5.通话前的网络测速

5.1. Android

RTC SDK为您提供网络测速的接口方法，您可以在音视频通话前进行网络测速，并根据当前的网络质量切换网络，获取更高质量的音视频通话。通过阅读本文，您可以了解网络测速的方法。

操作步骤

1. 使用网络测速功能时，应用层需要实现 `AliRtcEngineEventListener` 的 `onNetworkQualityProbeTest` 回调，用于接收测试结果返回。

```
private AliRtcEngineEventListener mEventListener = new AliRtcEngineEventListener() {
    @Override
    public void onNetworkQualityProbeTest(AliRtcEngine.AliRtcNetworkQuality aliRtcNetworkQuality)
{
      //AliRtcNetworkQuality返回测速后的网络质量
    }
};
```

2. 创建SDK引擎实例后，在加入频道之前，调用接口 `startNetworkQualityProbeTest` 启动测速功能，开始测速后，测试结果将从 `onNetworkQualityProbeTest` 中回调返回。

> **说明** 网络测试功能必须在加入频道前使用，加入频道前您需要主动停止网络测速。

```
AliRtcEngine mAliRtcEngine =AliRtcEngine.getInstance(this);
int i = mAliRtcEngine.startNetworkQualityProbeTest();
```

3. 结束网络测速时，调用接口 `stopNetworkQualityProbeTest` 停止测速。

```
mAliRtcEngine.stopNetworkQualityProbeTest();
```

5.2. iOS和Mac

RTC SDK为您提供网络测速的接口方法，您可以在音视频通话前进行网络测速，并根据当前的网络质量切换网络，获取更高质量的音视频通话。通过阅读本文，您可以了解网络测速的方法。

操作步骤

1. 使用网络测速功能时，应用层需要继承并实现 `onLastmileDetectResultWithQuality` 回调，用于接收测试结果返回。

```
/*
* @brief 网络质量探测回调
* @param networkQuality 网络质量
*/
- (void)onLastmileDetectResultWithQuality:(AliRtcNetworkQuality)networkQuality;
```

2. 创建SDK引擎实例后，在加入频道之前，调用接口 `startLastmileDetect` 启动测速功能。开始测速后，测试结果将从 `onLastmileDetectResultWithQuality` 中回调返回。

> **说明** 网络测试功能必须在加入频道前使用，加入频道前您需要主动停止网络测速。

```
/**
 * @brief 开始网络质量探测
 * @return 成功返回0，失败返回非0值
 */
- (int)startLastmileDetect;
// 示例
[self.engine startLastmileDetect];
```

3. 结束网络测速时，调用接口 `stopLastmileDetect` 停止测速。

```
/**
 * @brief 停止网络质量探测
 * @return 成功返回0，失败返回非0值
 */
- (int)stopLastmileDetect;
// 示例
[self.engine stopLastmileDetect];
```

5.3. Windows

RTC SDK为您提供网络测速的接口方法，您可以在音视频通话前进行网络测速，并根据当前的网络质量切换网络，获取更高质量的音视频通话。通过阅读本文，您可以了解网络测速的方法。

操作步骤

1. 使用网络测速功能时，应用层需要继承并实现 `onLastmileDetectResultWithQuality` 回调，用于接收测试结果返回。

```
/**
 * @brief 最后一公里网络质量探测回调
 * @param networkQuality 网络质量
 */
virtual void onLastmileDetectResultWithQuality(AliRtcNetworkQuality networkQuality)
{
  // 接收网络测速
};
```

2. 创建SDK引擎实例后，在加入频道之前，调用接口 `startLastmileDetect` 启动测速功能。开始测速后，测试结果将从 `onLastmileDetectResultWithQuality` 中回调返回。

> **说明** 网络测试功能必须在加入频道前使用，加入频道前您需要主动停止网络测速。

```
/**
* @brief 开始最后一公里网络质量探测
* @return 成功返回0，失败返回非0值
*/
virtual int startLastmileDetect() = 0;
AliRtcEngine *pEngine = AliRtcEngine::sharedInstance(listener, "");
pEngine->startLastmileDetect();
```

3. 结束网络测速时，调用接口stopLastmileDetect停止测速。

```
/**
* @brief 停止最后一公里网络质量探测
*/
virtual int stopLastmileDetect() = 0;
pEngine->stopLastmileDetect();
```

6.音视频设备测试说明

6.1. Mac

RTC SDK为您提供音视频设备测试的方法，您可以在音视频通话前检查当前设备上的摄像头，麦克风以及扬声器等音视频设备是否正常工作，以保证音视频通话质量。通过阅读本文，您可以了解音视频设备测试的方法。

摄像头测试

1. 创建SDK实例后，应用层可以调用接口 `getCameraList` 获取当前设备上所有可用摄像头，返回设备列表中包含设备ID及设备名称，应用层可选择通过设备名称或设备ID进行判断，并通过接口 `setCurrentCamera` 或 `setCurrentCameraWithID` 选中需要测试的设备。

```
NSArray *camList = [self.engine getCameraList];
// 设备名称
NSString *matchDeviceName = /*测试设备名*/;
int camCount = (int)camList.count;
for(int i=0; i<camCount; i++) {
  AliRtcDeviceInfo *deviceInfo = [camList objectAtIndex:i];
  NSString *name = deviceInfo.deviceName;
  if ([name isEqualToString:matchDeviceName]) {
    [self.engine setCurrentCamera:matchDeviceName];
  }
}
// 设备ID
NSString *matchDeviceId = /*测试设备ID*/;
int camCount = (int)camList.count;
for(int i=0; i<camCount; i++) {
  AliRtcDeviceInfo *deviceInfo = [camList objectAtIndex:i];
  NSString *deviceID = deviceInfo.deviceId;
  if ([deviceID isEqualToString:matchDeviceId]) {
    [self.engine setCurrentCameraWithID:matchDeviceId];
  }
}
```

2. 设置测试摄像头设备之后，通过setLocalViewConfig接口可以设置预览显示窗口，再启动预览，即可通过预览画面是否正常显示，判断当前摄像头设备是否正常工作。

```
// 设置预览窗口
AliVideoCanvas *canvas = [[AliVideoCanvas alloc] init];
canvas.renderMode = renderMode;
canvas.view = (AliRenderView *)view;
canvas.mirrorMode = mirrorMode;
[self.engine setLocalViewConfig:canvas forTrack:AliRtcVideoTrackCamera];
[self.engine startPreview];
```

麦克风测试

1. 开始测试前，通过接口 `getAudioCaptures` 获取当前设备上所有可用麦克风设备，返回设备列表中将同时返回设备ID及设备名称，应用层可选择通过设备名称调用接口 `startTestAudioRecordWithName` 进行麦克风测试，测试开始后会收到 `onAudioDeviceRecordLevel` 回调，接收麦克风测试时返回的音量值。

```
// 开始测试
NSArray *audioList = [self.engine getAudioCaptures];
AliRtcDeviceInfo *deviceInfo = audioList[0];
[self.engine startTestAudioRecordWithName:deviceInfo.deviceName];
// 回调
- (void)onAudioDeviceRecordLevel:(int)level {
  dispatch_async(dispatch_get_main_queue(), ^{
    NSLog(@"麦克风音量值:%d", level);
  });
}
```

2. 测试完成后，调用接口stopTestAudioRecord停止麦克风测试。

```
[self.engine stopTestAudioRecord];
```

扬声器测试

1. 开始测试前，通过接口 getAudioRenderers 获取当前设备上所有可用扬声器设备，返回设备列表中将同时返回设备ID及设备名称，应用层可选择通过设备名称调用接口 startTestAudioPlayoutWithName 进行扬声器测试，测试开始后会收到 onAudioDevicePlayoutLevel 回调，接收扬声器测试时返回的音量值。

```
// 开始测试
NSArray *speakerList = [self.engine getAudioRenderers];
AliRtcDeviceInfo *deviceInfo = speakerList[0];
[self.engine startTestAudioPlayoutWithName:deviceInfo.deviceName filePath:path loopCycles:0];
// 回调
- (void)onAudioDevicePlayoutLevel:(int)level {
  dispatch_async(dispatch_get_main_queue(), ^{
    NSLog(@"扬声器音量值:%d", level);
  });
}
```

> **说明** 目前扬声器测试播放文件只支持Wave格式，传入路径需要为绝对路径，并保证可以被读取访问。

2. 测试完成或接收到文件播放结束事件回调后，调用stopTestAudioPlayout接口停止扬声器测试。

```
[self.engine stopTestAudioPlayout];
```

6.2. Windows

RTC SDK为您提供音视频设备测试的方法，您可以在音视频通话前检查当前设备上的摄像头，麦克风以及扬声器等音视频设备是否正常工作，以保证音视频通话质量。通过阅读本文，您可以了解音视频设备测试的方法。

摄像头测试

1. 创建SDK实例后，应用层可以调用接口 getCameraList ，获取当前设备上所有可用摄像头，返回设备列表中将同时返回设备ID及设备名称，应用层可选择通过设备名称或设备ID进行判断，并通过接口 setCurrentCamera 或 setCurrentCameraById 选中需要测试的设备。

```
AliRtcEngine *pEngine = AliRtcEngine::sharedInstance(listener, "");
AliRtcDeviceList cameraList;
pEngine->getCameraList(cameraList);
#if defined(USE_DEVIC_NAME)
// 可以通过遍历所有摄像头设备名，查找并设置需要测试的设备
std::string matchDeviceName = /*测试设备名*/;
for (size_t i = 0; i != cameraList.deviceNames.size(); ++i)
{
  AliRtc::String deviceName = cameraList.deviceNames.at(i);
  if(std::string(deviceName.c_str()) == matchDeviceName)
  {
    // 设置测试设备
    pEngine->setCurrentCamera(deviceName);
  }
}
#else
// 可以通过遍历所有摄像头设备ID并查找需要测试的设备
std::string matchDeviceId = /*测试设备ID*/;
for (size_t i = 0; i != cameraList.deviceIds.size(); ++i)
{
  AliRtc::String deviceId = cameraList.deviceIds.at(i);
  if(std::string(deviceId.c_str()) == matchDeviceId)
  {
    // 设置测试设备
    pEngine->setCurrentCameraById(deviceId);
  }
}
#endif
```

2. 设置测试摄像头设备之后，通过 `setLocalViewConfig` 接口可以设置预览显示窗口，然后启动预览，即可通过预览画面是否正常显示，判断当前摄像头设备是否正常工作。

```
// 设置预览窗口
AliVideoCanvas canvas;
canvas.hWnd = /*预览显示窗口句柄*/;
pEngine->setLocalViewConfig(canvas, AliRtcVideoTrackCamera);
// 开启预览检查，确认显示是否正常
pEngine->startPreview();
```

麦克风测试

1. 创建SDK实例后，应用层需要继承 `AliMediaDeviceTestEventListener` 接口，实现 `OnAudioDeviceRecordLevel` 回调，用于接收麦克风测试时返回的音量值。再通过SDK接口 `createMediaDeviceTestInterface` 创建设备测试实例，并在创建时传入回调监听实例。

```
// 继承实现设备测试事件回调
class DeviceTestEventListener : public AliMediaDeviceTestEventListener
{
public:
  virtual void OnAudioDeviceRecordLevel(int level)
  {
    // 处理麦克风测试音量回调
  }
};
// 创建SDK实例及设备测试实例
AliRtcEngine *pEngine = AliRtcEngine::sharedInstance(listener, "");
DeviceTestEventListener* deviceTestListener = new DeviceTestEventListener();
AliMediaDeviceTestInterface *pTestInterface = pEngine->createMediaDeviceTestInterface(deviceTestListener);
```

2. 开始测试前，通过接口 `getAudioCaptures` 获取当前设备上所有可用麦克风设备，返回设备列表中将同时返回设备ID及设备名称，应用层可选择通过设备名或设备ID判断并选择需要测试的设备。

```
AliRtcDeviceList audioCaptureList;
pEngine->getAudioCaptures(audioCaptureList);
#if defined(USE_DEVIC_NAME)
// 可以通过遍历所有麦克风设备名，查找需要测试的设备
std::string matchDeviceName = /*测试设备名*/;
for (size_t i = 0; i != audioCaptureList.deviceNames.size(); ++i)
{
  AliRtc::String deviceName = audioCaptureList.deviceNames.at(i);
  matchDeviceName = ...
}
#else
// 可以通过遍历所有麦克风设备ID，查找需要测试的设备
std::string matchDeviceId = /*测试设备ID*/;
for (size_t i = 0; i != audioCaptureList.deviceIds.size(); ++i)
{
  AliRtc::String deviceId = audioCaptureList.deviceIds.at(i);
  matchDeviceId = ...
}
#endif
```

3. 选中测试设备后，调用麦克风测试接口启动测试，接口中需要指明测试设备名称或者设备ID，以及音量回调频率（传入0为默认频率，每20ms回调一次音量）。测试开始后，提示您对麦克风设备说话，并将 `OnAudioDeviceRecordLevel` 回调中返回的采集音量值进行展示，判断当前麦克风设备是否正常工作。

```
// 启动麦克风设备测试
#if defined(USE_DEVIC_NAME)
pTestInterface->StartTestAudioRecord(matchDeviceName.c_str(), 0);
#else
pTestInterface->StartTestAudioRecordById(matchDeviceName.c_str(), 0);
#endif
```

4. 测试完成后，调用接口 `StopTestAudioRecord` 停止麦克风测试，并释放设备测试功能实例。

```
pTestInterface->StopTestAudioRecord();
pTestInterface->Release();
pTestInterface = nullptr;
```

扬声器测试

1. 创建SDK引擎实例后，应用层需要继承 AliMediaDeviceTestEventListener 接口，实现 OnAudioDevicePlayoutLevel 回调，用于接收扬声器测试时返回的音量值，同时实现 OnAudioDevicePlayoutEnd 回调，用于接收播放文件结束事件。然后通过SDK接口 createMediaDeviceTestInterface 创建设备测试实例，并在创建时传入回调监听实例。

```
// 继承实现设备测试事件回调
class DeviceTestEventListener : public AliMediaDeviceTestEventListener
{
public:
  virtual void OnAudioDevicePlayoutLevel(int level)
  {
    // 处理扬声器测试音量回调
  }
  virtual void OnAudioDevicePlayoutEnd()
  {
    // 处理扬声器测试播放结束事件
  }
};
// 创建SDK实例及设备测试实例
AliRtcEngine *pEngine = AliRtcEngine::sharedInstance(listener, "");
DeviceTestEventListener* deviceTestListener = new DeviceTestEventListener();
AliMediaDeviceTestInterface *pTestInterface = pEngine->createMediaDeviceTestInterface(deviceTestListener);
```

2. 开始测试前，通过接口 getAudioCaptures 获取当前设备上所有可用扬声器，返回设备列表中将同时返回设备ID及设备名称，应用层可选择通过设备名或设备ID判断并选中需要测试的设备。

```
AliRtcDeviceList audioRenderList;
pEngine->getAudioRenderers(audioRenderList);
#if defined(USE_DEVIC_NAME)
// 可以通过遍历所有麦克风设备名，查找需要测试的设备
std::string matchDeviceName = /*测试设备名*/;
for (size_t i = 0; i != audioRenderList.deviceNames.size(); ++i)
{
  AliRtc::String deviceName = audioRenderList.deviceNames.at(i);
  matchDeviceName = ...
}
#else
// 也可以通过遍历所有麦克风设备ID，查找需要测试的设备
std::string matchDeviceId = /*测试设备ID*/;
for (size_t i = 0; i != audioRenderList.deviceIds.size(); ++i)
{
  AliRtc::String deviceId = audioRenderList.deviceIds.at(i);
  matchDeviceId = ...
}
#endif
```

3. 选中测试设备后，调用扬声器测试接口启动测试，接口中需要指明测试设备名称或设备ID，音量回调频率（传入0为默认频率，每20ms回调一次音量），以及测试使用的音频文件路径。开始测试后，可以将 OnAudioDevicePlayoutLevel 回调中返回的采集音量值进行展示，同时关注扬声器中播放的测试音频，判断当前扬声器设备是否正常工作。

```
// 启动麦克风设备测试
#if defined(USE_DEVIC_NAME)
pTestInterface->StartTestAudioPlayout(matchDeviceName.c_str(), 0, wavPath.c_str());
#else
pTestInterface->StartTestAudioPlayoutById(matchDeviceId.c_str(), 0, wavPath.c_str());
#endif
```

> **说明** 目前扬声器测试播放文件只支持Wave格式，传入路径需要为绝对路径，并保证可以被读取访问。

4. 测试完成或接收到文件播放结束事件回调后，调用 `StopTestAudioRecord` 接口停止麦克风测试，并释放设备测试功能实例。

```
pTestInterface->StopTestAudioPlayout();
pTestInterface->Release();
pTestInterface = nullptr;
```

7.音频管理

7.1. 音量设置

RTC SDK为您提供了不同类型音量设置接口。通过阅读本文，您可以了解各类型音量的设置方法及关系。

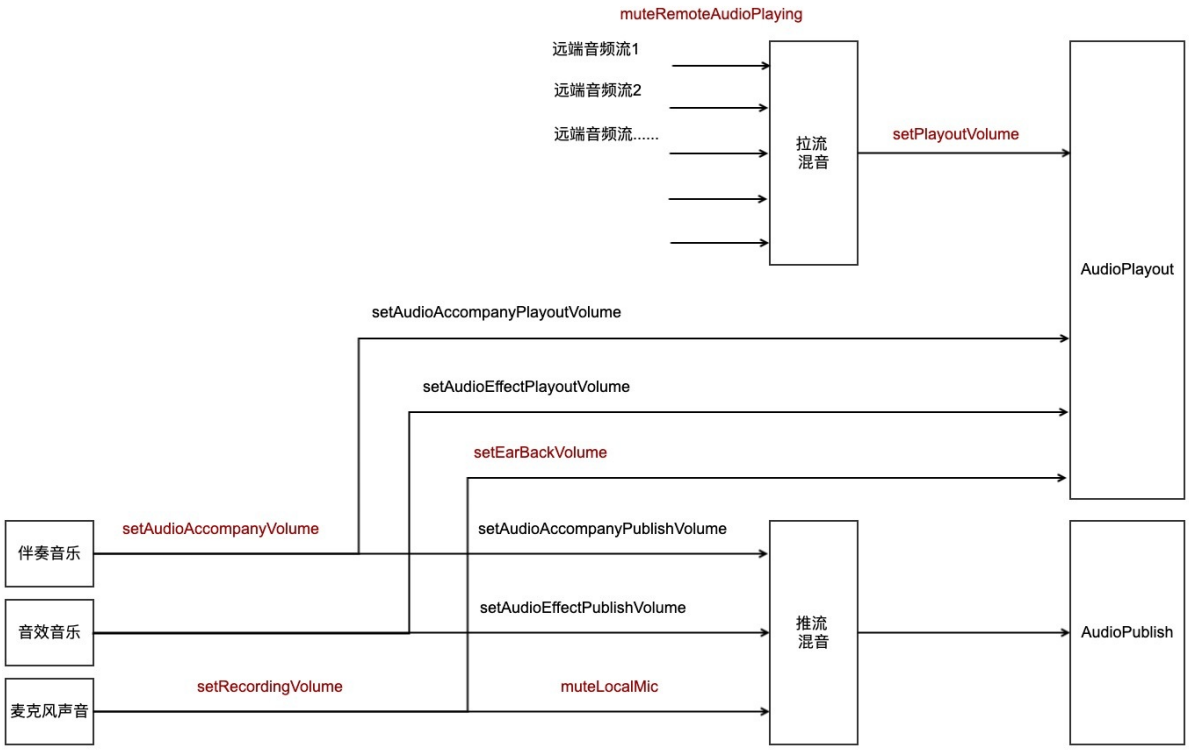
功能简介

SDK中不同音量设置的接口如下表所示：

类别	接口	备注
伴奏音量	setAudioAccompanyVolume setAudioAccompanyPlayoutVolume setAudioAccompanyPublishVolume	setAudioAccompanyVolume统一设置本地播放音量和推流音量，setAudioAccompanyPlayoutVolume和setAudioAccompanyPublishVolume分别设置本地播放音量和推流音量。 **注意** 分开设置音量后，会覆盖之前统一设置的音量。
音效音量	setAudioEffectPublishVolume setAudioEffectPlayoutVolume	前者设置音效推流音量，后者设置音效本地音量。
录音音量	setRecordingVolume	调整采集推出去的音频音量。
播放音量	setPlayoutVolume	设置远端推流的音频播放音量。
停止发布本地音频	muteLocalMic	该接口推空音频帧，音频正在采集的，还能听到耳返的声音。
停止播放远端音频	muteRemoteAudioPlaying	可以静音远端某个用户的混合音频。

音量设置关系

音量设置关系如下图所示：



7.2. 耳返设置

RTC SDK为您提供了设置耳返的接口。通过阅读本文，您可以了解设置耳返的方法。

背景信息

随着近些年连麦、多人在线K歌等互联网娱乐场景的兴起，音视频通信不仅可以支撑音视频会议、在线教育等场景，也向娱乐场景衍生出了各种各样的技术方案，耳返便是其中一种。在娱乐唱歌场景中，主播为了更好的展示自己，需要边唱歌边听到自己的声音效果，耳返在低延时的情况下可以给主播一个比较真实的反馈，阿里云RTC SDK支持耳返功能，同时支持调节耳返的音量。

实现方法

RTC SDK提供 `enableEarBack` 方法开启或关闭耳返功能，提供 `setEarBackVolume` 方法设置耳返音量。

- Android端

```
mAliRtcEngine = AliRtcEngine.getInstance(getApplicationContext());
// 设置开启耳返功能，默认为false
mAliRtcEngine.enableEarBack(true);
// 设置耳返的音量，volume的取值范围为0~100，默认值100，代表麦克风录到的原始音量
int volume = 80;
mAliRtcEngine.setEarBackVolume(volume);
```

接口详情，请参见enableEarBack和setEarBackVolume。

- iOS端

```
// 开启耳返
[self.engine enableEarBack:YES];
// 设置耳返音量，音量的取值范围为0~100，默认100
[self.engine setEarBackVolume:80];
```

接口详情，请参见enableEarBack和setEarBackVolume。

> **说明** 只有在耳返功能开启的情况下，才能设置耳返音量。

7.3. 音效设置

7.3.1. Android播放音效文件

阿里云RTC SDK为您提供伴奏文件和音效文件的相关接口方法，您可以通过本文了解其具体的实现方法。

伴奏文件

伴奏文件的接口方法如下所示。

- 调用startAudioAccompany开始混音伴奏。

```
//接口方法
public abstract int startAudioAccompany(String fileName, boolean onlyLocalPlay, boolean replaceMic, int loopCycles) ;
//示例方法
int ret = mAliRtcEngine.startAudioAccompany(audioFileName, localPlay, replaceMic, loopCycles);
```

参数	类型	描述
fileName	String	伴奏文件路径
onlyLocalPlay	boolean	是否只本地播放，true：只本地播放；false：本地播放和推流
replaceMic	boolean	是否替换麦克风采集，true：替换麦克风采集，只有伴奏声；false：与麦克风共存
loopCycles	int	循环次数

- 调用pauseAudioAccompany暂停伴奏，调用成功返回0，失败返回错误码。

> **说明** 您需要在startAudioAccompany接口之后调用。

```
//接口方法
public abstract int pauseAudioAccompany() ;
//示例方法mAliRtcEngine.pauseAudioAccompany();
```

- 调用resumeAudioAccompany恢复伴奏播放，调用成功返回0，失败返回错误码。

> **说明** 您可以与pauseAudioAccompany接口搭配使用。

```
//接口方法
public abstract int resumeAudioAccompany();
//示例方法mAliRtcEngine.resumeAudioAccompany();
```

- 调用stopAudioAccompany停止伴奏播放，调用成功返回0，失败返回错误码。

> **说明** 调用该接口后会立刻停止伴奏播放，您可以与startAudioAccompany接口搭配使用。

```
//接口方法
public abstract int stopAudioAccompany();
//示例方法
mAliRtcEngine.stopAudioAccompany();
```

- 您可以通过以下接口方法设置音量。

```
/**
 * 设置混音音量，需要在调用startAudioAccompany后才能生效
 * @param volume 混音音量，取值：0~100
 * @return 调用成功返回0，失败返回错误码
 */
public abstract int setAudioAccompanyVolume( int volume);
/**
 * 设置混音之后推流出去的音量，需要在调用startAudioAccompany后才能生效
 * @param volume 混音音量，取值：0~100
 * @return 调用成功返回0，失败返回错误码
 */
public abstract int setAudioAccompanyPublishVolume(int volume);
/**
 * 获取推流出去的混音音量
 * @return 调用成功返回0，失败返回错误码
 */
public abstract int getAudioAccompanyPublishVolume();
/**
 * 设置混音之后本地播放的音量，需要在调用startAudioAccompany后才能生效
 * @param volume 混音音量，取值：0~100
 * @return 调用成功返回0，失败返回错误码
 */
public abstract int setAudioAccompanyPlayoutVolume(int volume);
/**
 * 获取混音本地播放的音量
 * @return 调用成功返回0，失败返回错误码
 */
public abstract int getAudioAccompanyPlayoutVolume();
```

音效文件

音效文件的接口方法如下所示。

- 调用preloadAudioEffect预加载音效文件，调用成功返回0，失败返回错误码。

> **说明** 您设置的音效文件ID会进行后续操作。

```
//接口方法
public abstract int preloadAudioEffect(int soundId, String filePath) ;
//示例方法
mAliRtcEngine.preloadAudioEffect(1, filePath);
```

参数	类型	描述
filePath	String	伴奏文件路径（建议不包含中文）
soundId	int	指定的文件ID，个数无限制

- 调用unloadAudioEffect根据预加载的音效文件ID删除预加载音效，调用成功返回0，失败返回错误码。

```
//代码方法
public abstract int unloadAudioEffect(int soundId) ;
//示例方法
mAliRtcEngine.unloadAudioEffect(soundId);
```

- 调用playAudioEffect开始播放音效，调用成功返回0，失败返回错误码。

> **说明** 成功调用后立即播放音效，您需要在preloadAudioEffectWithSoundId接口之后调用。

```
//接口方法
public abstract int playAudioEffect(int soundId, String filePath, int cycles, boolean publish);
//示例方法
mAliRtcEngine.playAudioEffect(soundId, filePath, cycles, isPublish);
```

参数	类型	描述
soundId	int	预加载时指定的文件ID
filePath	String	音效文件路径（建议不包含中文）
cycles	int	循环次数
publish	boolean	是否推流，false：不推流并且仅本地播放；true：本地播放和推流

- 调用pauseAudioEffect暂停音效，调用成功返回0，失败返回错误码。

```
//代码方法
public abstract int pauseAudioEffect(int soundId) ;
//示例方法
mAliRtcEngine.pauseAudioEffect(soundId);
```

- 调用resumeAudioEffect恢复音效，调用成功返回0，失败返回错误码。

> **说明** 您可以与pauseAudioEffect接口搭配使用。

```
//接口方法
public abstract int resumeAudioEffect(int soundId) ;
//示例方法
mAliRtcEngine.resumeAudioEffect(soundId);
```

- 调用stopAudioEffect停止音效，调用成功返回0，失败返回错误码。

> **说明** 成功调用之后会立刻停止音效播放，您可以与playAudioEffect接口搭配使用。

```
//接口方法
public abstract int stopAudioEffect(int soundId) ;
//示例方法
mAliRtcEngine.stopAudioEffect(soundId);
```

- 您可以通过以下接口方法设置音效音量。

```
/**
 * 设置音效音量
 * @param soundId 音效文件ID
 * @param volume 混音音量，取值：0~100
 * @return 调用成功返回0，失败返回错误码
 */
public abstract int setAudioEffectPublishVolume(int soundId, int volume) ;
/**
 * 获取推流音效音量
 * @param soundId 音效文件ID
 * @return 调用成功返回0~100音量，失败返回错误码
 */
public abstract int getAudioEffectPublishVolume(int soundId) ;
/**
 * 设置音效本地播放音量
 * @param soundId 音效文件ID
 * @param volume 混音音量，取值：0~100
 * @return 调用成功返回0，失败返回错误码
 */
public abstract int setAudioEffectPlayoutVolume(int soundId, int volume) ;
/**
 * 获取音效本地播放音量
 * @param soundId 音效文件ID
 * @return 调用成功返回0~100音量，失败返回错误码
 */
public abstract int getAudioEffectPlayoutVolume(int soundId) ;
```

7.3.2. iOS播放音效文件

RTC SDK为您提供了伴奏文件和音效文件设置接口。通过阅读本文，您可以了解伴奏文件和音效文件的设置方法。

伴奏文件

- startAudioAccompanyWithFile（仅iOS可用）：开始播放伴奏。

```
- (int)startAudioAccompanyWithFile:(NSString *)filePath onlyLocalPlay:(BOOL)onlyLocalPlay replaceMic:(BOOL)replaceMic loopCycles:(NSInteger)loopCycles;
```

参数说明

名称	类型	描述
filePath	NSString *	文件路径。
onlyLocalPlay	BOOL	是否只本地播放。取值：YES|NO。
replaceMic	BOOL	是否替换MIC（麦克风）。取值：YES|NO。
loopCycles	NSInteger	循环次数。可以设置为-1或者正整数。

返回说明

0表示成功，其他返回错误码。

> 注意 支持本地资源和网络资源播放。

- pauseAudioAccompany（仅iOS可用）：暂停播放伴奏。

```
- (int)pauseAudioAccompany;
```

返回说明

0表示成功，其他返回错误码。

- resumeAudioAccompany（仅iOS可用）：恢复播放伴奏。

```
- (int)resumeAudioAccompany;
```

返回说明

0表示成功，其他返回错误码。

- stopAudioAccompany（仅iOS可用）：停止伴奏。

```
- (int)stopAudioAccompany;
```

返回说明

0表示成功，其他返回错误码。

- setAudioAccompanyVolume：设置伴奏音量。

```
- (int)setAudioAccompanyVolume:(NSInteger)volume;
```

参数说明

名称	类型	描述
volume	NSInteger	混音音量。取值范围：0~100。默认取值50。

返回说明

0表示成功，其他返回错误码。

> 注意 设置音量需要在调用startAudioAccompanyWithFile后才能生效。

- setAudioAccompanyPublishVolume（仅iOS可用）：设置伴奏之后推流出去的音量。

```
- (int)setAudioAccompanyPublishVolume:(NSInteger)volume;
```

参数说明

名称	类型	描述
volume	NSInteger	混音音量。取值范围：0~100。默认取值50。

返回说明

0表示成功，其他返回错误码。

> 注意 设置音量需要在调用startAudioAccompanyWithFile后才能生效。

- getAudioAccompanyPublishVolume（仅iOS可用）：获取伴奏推流音量。

```
- (int)getAudioAccompanyPublishVolume;
```

返回说明

0表示成功，其他返回错误码。

- setAudioAccompanyPlayoutVolume（仅iOS可用）：设置伴奏本地音量。

```
- (int)setAudioAccompanyPlayoutVolume:(NSInteger)volume;
```

参数说明

名称	类型	描述
volume	NSInteger	混音音量。取值范围：0~100。默认取值50。

返回说明

0表示成功，其他返回错误码。

> 注意 设置音量需要在调用startAudioAccompanyWithFile后才能生效。

- getAudioAccompanyPlayoutVolume（仅iOS可用）：获取伴奏本地音量。

```
- (int)getAudioAccompanyPlayoutVolume;
```

返回说明

0表示成功，其他返回错误码。

音效文件

- preloadAudioEffectWithSoundId（仅iOS可用）：预加载音效。

```
- (int)preloadAudioEffectWithSoundId:(NSInteger)soundId filePath:(NSString *)filePath;
```

参数说明

名称	类型	描述
soundId	NSInteger	用户给该音效文件分配的ID。
filePath	NSString *	音效文件路径。

返回说明

0表示成功，其他返回错误码。

- unloadAudioEffectWithSoundId（仅iOS可用）：清除预加载音效。

```
- (int)unloadAudioEffectWithSoundId:(NSInteger)soundId;
```

参数说明

名称	类型	描述
soundId	NSInteger	用户给该音效文件分配的ID。

返回说明

0表示成功，其他返回错误码。

- playAudioEffectWithSoundId（仅iOS可用）：开始播放音效。

```
- (int)playAudioEffectWithSoundId:(NSInteger)soundId filePath:(NSString *)filePath cycles:(NSInteger)cycles publish:(BOOL)publish;
```

参数说明

名称	类型	描述
soundId	NSInteger	用户给该音效文件分配的ID。
filePath	NSString *	音效文件路径。
cycles	NSInteger	循环次数。可以设置-1或者正整数。
publish	BOOL	是否发布。取值：YES|NO。

返回说明

0表示成功，其他返回错误码。

- pauseAudioEffectWithSoundId（仅iOS可用）：暂停播放音效。

```
- (int)pauseAudioEffectWithSoundId:(NSInteger)soundId;
```

参数说明

名称	类型	描述
soundId	NSInteger	用户给该音效文件分配的ID。

返回说明

0表示成功，其他返回错误码。

- resumeAudioEffectWithSoundId（仅iOS可用）：恢复播放音效。

```
- (int)resumeAudioEffectWithSoundId:(NSInteger)soundId;
```

参数说明

名称	类型	描述
soundId	NSInteger	用户给该音效文件分配的ID。

返回说明

0表示成功，其他返回错误码。

- stopAudioEffectWithSoundId（仅iOS可用）：停止播放音效。

```
- (int)stopAudioEffectWithSoundId:(NSInteger)soundId;
```

参数说明

名称	类型	描述
soundId	NSInteger	用户给该音效文件分配的ID。

返回说明

0表示成功，其他返回错误码。

- setAudioEffectPublishVolumeWithSoundId（仅iOS可用）：设置音效推流音量。

```
- (int)setAudioEffectPublishVolumeWithSoundId:(NSInteger)soundId volume:(NSInteger)volume;
```

参数说明

名称	类型	描述
soundId	NSInteger	用户给该音效文件分配的ID。
volume	NSInteger	混音音量。取值范围：0~100。默认取值50。

返回说明

0表示成功，其他返回错误码。

- getAudioEffectPublishVolumeWithSoundId（仅iOS可用）：获取音效推流音量。

```
- (int)getAudioEffectPublishVolumeWithSoundId:(NSInteger)soundId;
```

参数说明

名称	类型	描述
soundId	NSInteger	用户给该音效文件分配的ID。

返回说明

返回0~100为成功，否则返回错误码。

- setAudioEffectPlayoutVolumeWithSoundId（仅iOS可用）：设置音效本地音量。

```
- (int)setAudioEffectPlayoutVolumeWithSoundId:(NSInteger)soundId volume:(NSInteger)volume;
```

参数说明

名称	类型	描述
soundId	NSInteger	用户给该音效文件分配的ID。
volume	NSInteger	混音音量。取值范围：0~100。默认取值50。

返回说明

0表示成功，其他返回错误码。

- getAudioEffectPlayoutVolumeWithSoundId（仅iOS可用）：获取音效本地音量。

```
- (int)getAudioEffectPlayoutVolumeWithSoundId:(NSInteger)soundId;
```

参数说明

名称	类型	描述
soundId	NSInteger	用户给该音效文件分配的ID。

返回说明

0表示成功，其他返回错误码。

8.频道成员管理

8.1. Android

RTC SDK为您提供了频道成员管理的接口方法，您可以获取远端在线用户列表、查询远端用户信息、查询用户是否在线等功能。通过阅读本文，您可以了解频道成员管理的方法。

实现方法

以下为常用的频道成员管理方法，更多信息，请参见AliRtcEngine接口。

- getOnlineRemoteUsers：获取远端在线用户列表。

  ```
  public abstract String[] getOnlineRemoteUsers()
  ```

  返回说明

  返回用户ID列表。

  > 注意 该方法在加入频道时调用有延时，建议在调用onRemoteUserOnLineNotify后再调用此接口或自己维护一个远端用户列表。

- getUserInfo：查询远端用户信息。

  ```
  public abstract AliRtcRemoteUserInfo getUserInfo(String uid)
  ```

  参数说明

名称	类型	描述
uid	String	用户ID。

  返回说明

  返回远程用户信息AliRtcRemoteUserInfo。

- isUserOnline：查询用户是否在线。

  ```
  public abstract boolean isUserOnline(String uid)
  ```

  参数说明

名称	类型	描述
uid	String	用户ID。

  返回说明

  true表示在线，false表示不在线。

8.2. iOS和Mac

RTC SDK为您提供了频道成员管理的接口方法，您可以获取远端在线用户列表、查询远端用户信息、查询用户是否在线等功能。通过阅读本文，您可以了解频道成员管理的方法。

实现方法

以下为常用的频道成员管理方法，更多信息，请参见AliRtcEngine接口。

- getOnlineRemoteUsers：获取远端在线用户列表。

```
- (NSArray<NSString *> *)getOnlineRemoteUsers;
```

返回说明

返回用户ID列表。

- getUserInfo：查询远端用户信息。

```
- (NSDictionary *)getUserInfo:(NSString *)uid;
```

参数说明

名称	类型	描述
uid	NSString *	用户ID。

- isUserOnline：查询用户是否在线。

```
- (BOOL)isUserOnline:(NSString *)uid;
```

参数说明

名称	类型	描述
uid	NSString *	用户ID。

返回说明

YES表示在线，NO表示不在线。

8.3. Windows

RTC SDK为您提供了频道成员管理的接口方法，您可以获取远端在线用户列表、查询远端用户信息、查询用户是否在线等功能。通过阅读本文，您可以了解频道成员管理的方法。

实现方法

以下为常用的频道成员管理方法，更多信息，请参见AliRtcEngine接口。

- getOnlineRemoteUsers：获取远端在线用户列表。

```
void getOnlineRemoteUsers(AliRtc::StringArray& array)
```

参数说明

名称	类型	描述
array	AliRtc::StringArray&	用户列表（用户ID列表）。

- getUserInfo：查询远端用户信息。返回0表示成功获取，其他表示失败。

```
int getUserInfo(const AliRtc::String& uid, AliRtc::Dictionary& dict)
```

参数说明

名称	类型	描述
uid	const AliRtc::String&	用户ID。
dict	AliRtc::Dictionary&	用于存放用户数据。取值：

dict当中key值包括：userID、isOnline、sessionID、callID、displayName、hasAudio,hasCameraMaster、hasCameraSlave、hasScreenSharing、requestAudio,requestCameraMaster、requestCameraSlave、requestScreenSharing、preferCameraMaster subScribedAudio、subScribedCameraMaster,subScribedCamearSlave、subScribedScreenSharing、hasCameraView、hasScreenView、muteAudioPlaying。

返回说明

0表示方法调用成功，其它表示方法调用失败。

- isUserOnline：查询用户是否在线。

```
bool isUserOnline(const AliRtc::String& uid)
```

参数说明

名称	类型	描述
uid	const AliRtc::String&	用户ID。

返回说明

true表示用户在线，false表示用户不在线。

9.摄像头管理

9.1. Android

RTC SDK为您提供了摄像头管理的接口方法，您可以在音视频通话之前对摄像头进行管理，以确保设备是否正常工作。通过阅读本文，您可以了解摄像头管理的方法。

功能简介

阿里云RTC提供一系列摄像头管理方法，包括切换前后置摄像头、缩放镜头、曝光设置和对焦功能，您可以在加入频道前进行设置，帮助您在通话时使成像更清晰、大小与亮度更适宜。

实现方法

以下为常用的摄像头管理方法，更多信息，请参见AliRtcEngine接口。

- getCurrentCameraType：获取当前摄像头类型。

  返回摄像头类型AliRTCCameraType。

  ```
  public abstract AliRTCCameraType getCurrentCameraType()
  ```

- isCameraOn：检查摄像头是否打开。

  ```
  public abstract boolean isCameraOn()
  ```

  返回说明

  true表示摄像头已打开，false表示摄像头未打开。

- setPreCameraType：预设值摄像头方向。

  ```
  public abstract void setPreCameraType(int faceTo)
  ```

  参数说明

名称	类型	描述
faceTo	int	0表示后置，1表示前置（默认值为1）。

- getPreCameraType：获取预设值摄像头方向。

  ```
  public abstract int getPreCameraType()
  ```

  返回说明

  0表示后置摄像头，1表示前置摄像头。

- setCameraZoom：设置摄像头参数。

  ```
  public abstract int setCameraZoom(float zoom, boolean flash, boolean autoFocus)
  ```

  参数说明

名称	类型	描述
zoom	float	zoom变焦的级别（默认值：1.0）。
flash	boolean	true表示打开闪光灯，false表示不打开闪光灯。默认不打开闪光灯。
autoFocus	boolean	true表示打开自动对焦，false表示不打开自动对焦。默认不打开自动对焦。

返回说明

0表示设置成功，其他表示设置失败。

- isCameraSupportExposurePoint：相机是否支持手动曝光。

```
public abstract boolean isCameraSupportExposurePoint();
```

返回说明

true表示支持，false表示不支持。

- isCameraSupportFocusPoint：相机是否支持手动聚焦。

```
public abstract boolean isCameraSupportFocusPoint();
```

返回说明

true表示支持，false表示不支持。

- setCameraExposurePoint：设置手动曝光的坐标点。

```
public abstract int setCameraExposurePoint(float x, float y);
```

参数说明

名称	类型	描述
x	float	x坐标。
y	float	y坐标。

返回说明

0表示设置成功，其他表示设置失败。

- setCameraFocusPoint：设置手动聚焦的坐标点。

```
public abstract int setCameraFocusPoint(float x, float y);
```

参数说明

名称	类型	描述
x	float	x坐标。

名称	类型	描述
y	float	y坐标。

返回说明

0表示设置成功，其他表示设置失败。

- isCameraFlash：查看摄像头闪光灯是否开启。

```
public abstract boolean isCameraFlash();
```

返回说明

true表示开启，false表示未开启。

- getCameraZoom： 获取相机zoom（变焦）值。

```
public abstract float getCameraZoom();
```

返回说明

返回值范围：0.0~1.0。0表示相机支持的最小值，1表示相机支持的最大值。

9.2. iOS

RTC SDK为您提供了摄像头管理的接口方法，您可以在音视频通话之前对摄像头进行管理，以确保设备是否正常工作。通过阅读本文，您可以了解摄像头管理的方法。

功能简介

阿里云RTC提供一系列摄像头管理方法，包括切换前后置摄像头、缩放镜头、曝光设置和对焦功能，您可以在加入频道前进行设置，帮助您在通话时使成像更清晰、大小与亮度更适宜。

实现方法

以下为常用的摄像头管理方法，更多信息，请参见AliRtcEngine接口。

- switchCamera（仅iOS可用）：切换前后摄像头。

```
- (int)switchCamera;
```

返回说明

0表示切换成功，其他表示切换失败。

- setCameraZoom（仅iOS可用）：设置摄像头参数。

```
- (int)setCameraZoom:(float)zoom flash:(BOOL)flash autoFocus:(BOOL)autoFocus;
```

参数说明

名称	类型	描述
zoom	float	变焦的级别。取值范围：-3.0~3.0，默认取值1.0。

名称	类型	描述
flash	BOOL	是否打开闪光灯。YES为打开闪光灯，NO为不打开闪光灯。默认不打开闪光灯。
autoFocus	BOOL	是否打开自动对焦。YES为打开自动对焦，NO为不打开自动对焦。默认不打开自动对焦。

返回说明

0表示设置成功，其他表示设置失败。

- isCameraOn（仅iOS可用）：检查摄像头是否打开。

```
- (BOOL)isCameraOn;
```

返回说明

YES表示摄像头已打开，NO表示摄像头没有打开。

- isCameraFocusPointSupported（仅iOS可用）：相机是否支持手动聚焦。

```
- (BOOL)isCameraFocusPointSupported;
```

返回说明

YES表示支持手动聚焦，NO表示不支持手动聚焦。

- isCameraExposurePointSupported（仅iOS可用）：相机是否支持手动曝光。

```
- (BOOL)isCameraExposurePointSupported;
```

返回说明

YES表示支持手动曝光，NO表示不支持手动曝光。

- setCameraFocusPoint：设置手动聚焦的坐标点。

```
public abstract int setCameraFocusPoint(float x, float y);
```

参数说明

名称	类型	描述
x	float	x坐标。
y	float	y坐标。

返回说明

0表示设置成功，其他表示设置失败。

- setCameraExposurePoint（仅iOS可用）：设置手动曝光的坐标点。

```
- (int)setCameraExposurePoint:(CGPoint)point;
```

参数说明

名称	类型	描述
point	CGPoint	曝光点坐标。

返回说明

0表示成功，其他表示失败。

10.网络环境监控与弱网策略

10.1. Android

RTC SDK为您提供了网络质量监控的功能，您可以通过网络状况变化时回调获取网络质量，设置对应的音视频规格，以确保基础通信体验。通过阅读本文，您可以了解获取网络质量及设置音视频规格的方法。

功能简介

在网络质量不理想的情况下，音视频通信的质量受客观因素影响会下降。当监控到弱网环境时，为保证基础通信体验，建议您使用SDK对应的方法分别在发布端和订阅端进行如下优化。

- 调整视频流规格：通过设置较低档位规格的VideoProfile，减少视频通信的网络资源占用。
- 切换视频为小流：小流有着与大流相同的宽高比，但是分辨率和码率相对较低，网络资源占用的需求较低。
- 仅发布音频流：在极端网络环境下，可以选择只发送音频流，从而保证通信的持续。

您可以通过onNetworkQualityChanged（网络质量变化时回调）方法获得网络质量，然后在根据实际策略进行优化。

```
void onNetworkQualityChanged(String uid, AliRtcNetworkQuality upQuality, AliRtcNetworkQuality downQuality);
```

参数	类型	描述
downQuality	AliRtcNetworkQuality	下行网络质量。
upQuality	AliRtcNetworkQuality	上行网络质量。
uid	String	网络质量发生变化的用户ID，用户ID为空表示本地，其他表示远端。

实现方法

以下为常用的设置音视频流规格的方法，更多信息，请参见AliRtcEngine接口。

- setVideoProfile：设置视频流的参数。

  ```
  public abstract void setVideoProfile(AliRtcVideoProfile profile, AliRtcVideoTrack track)
  ```

  参数说明

名称	类型	描述
profile	AliRtcVideoProfile	视频流参数。默认分辨率480*640，帧率15的相机流。
track	AliRtcVideoTrack	需要设置的视频流类型，默认相机流。

- configRemoteCameraTrack：设置是否订阅远端相机流。

```
public abstract void configRemoteCameraTrack(String uid, boolean master, boolean enable)
```

参数说明

名称	类型	描述
uid	String	用户ID。
master	boolean	true为优先订阅大流，false为订阅次小流。默认为订阅大流。
enable	boolean	默认不订阅。true为订阅远端相机流，false为停止订阅远端相机流。

> 注意 该接口需要对流进行操作时（如手动订阅，关闭订阅），必须调用subscribe才能生效。

- configLocalCameraPublish：设置是否允许发布相机流。

```
public abstract void configLocalCameraPublish(boolean enable)
```

参数说明

名称	类型	描述
enable	boolean	true为允许发布相机流，false表示不允许。默认为允许发布相机流。

> 注意 手动发布时，需要调用publish才能生效。

10.2. iOS

RTC SDK为您提供了网络质量监控的功能，您可以通过网络状况变化时回调获取网络质量，设置对应的音视频规格，以确保基础通信体验。通过阅读本文，您可以了解获取网络质量及设置音视频规格的方法。

功能简介

在网络质量不理想的情况下，音视频通信的质量受客观因素影响会下降。当监控到弱网环境时，为保证基础通信体验，建议您使用SDK对应的方法分别在发布端和订阅端进行如下优化。

- 调整视频流规格：通过设置较低档位规格的VideoProfile，减少视频通信的网络资源占用。
- 切换视频为小流：小流有着与大流相同的宽高比，但是分辨率和码率相对较低，网络资源占用的需求较低。
- 仅发布音频流：在极端网络环境下，可以选择只发送音频流，从而保证通信的持续。

您可以通过onNetworkQualityChanged（网络质量变化时回调）方法获得网络质量，然后在根据实际策略进行优化。

```
- (void)onNetworkQualityChanged:(NSString *)uid
    upNetworkQuality:(AliRtcNetworkQuality)upQuality
    downNetworkQuality:(AliRtcNetworkQuality)downQuality;
```

参数	类型	描述
uid	NSString *	网络质量发生变化的用户ID，用户ID为空表示本地，其他表示远端。
upQuality	AliRtcNetworkQuality	上行网络质量。
downQuality	AliRtcNetworkQuality	下行网络质量。

实现方法

以下为常用的设置音视频流规格的方法，更多信息，请参见AliRtcEngine接口。

- setVideoProfile：设置视频流的参数。

```
- (void)setVideoProfile:(AliRtcVideoProfile)profile forTrack:(AliRtcVideoTrack)track;
```

参数说明

名称	类型	描述
profile	AliRtcVideoProfile	视频流参数。默认为分辨率480*640，帧率15的相机流。
track	AliRtcVideoTrack	需要设置的视频Track类型。默认相机流。

- configRemoteCameraTrack：设置是否订阅远端相机流。

```
- (void)configRemoteCameraTrack:(NSString *)uid preferMaster:(BOOL)master enable:(BOOL)enable;
```

参数说明

名称	类型	描述
uid	NSString *	用户ID。
master	BOOL	YES为订阅大流，NO为订阅次小流。默认为订阅大流。
enable	BOOL	YES为订阅远端相机流，NO为停止订阅远端相机流。默认为不订阅。

> 注意 该接口需要对流进行操作时（如手动订阅，关闭订阅），必须调用subscribe才能生效。

- configLocalCameraPublish：设置是否允许发布相机流。

```
- (void)configLocalCameraPublish:(BOOL)enable;
```

参数说明

名称	类型	描述
enable	BOOL	YES为允许发布相机流，NO为不允许。默认为允许发布相机流。

注意 手动发布时，需要调用publish才能生效。

11.设置频道模式

11.1. Android

通过阅读本文，您可以了解实现教师端和学生端互动模式的接入流程。

教师端

1. 在教师加入频道之前设置频道模式为互动模式。

```
aliRtcEngine.setChannelProfile(AliRTCSDK_Channel_Profile.AliRTCSDK_Interactive_live);
```

2. 在教师加入频道之前设置为非自动发布和自动订阅模式。

```
aliRtcEngine.setAutoPublishSubscribe(false, true);
```

3. 在加入频道之前设置角色为AliRTCSDK_Interactive。

```
aliRtcEngine.setClientRole(AliRTCSDK_Client_Role.AliRTCSDK_Interactive);
```

AliRTCSDK_Client_Role参数取值：

- AliRTCSDK_Interactive：参与互动角色，可以推拉流。
- AliRTCSDK_live：仅观看角色，只能拉流，适用于普通观众。

> **说明** setClientRole接口暂时不要处理接口返回值。

4. 加入频道。

```
aliRtcEngine.joinChannel(aliRtcAuthInfo, userName);
```

5. 调用publish接口开始发布。

```
// 配置推送音频流，可根据需要开启
aliRtcEngine.configLocalAudioPublish(true);
// 配置推送摄像头流，可根据需要开启
aliRtcEngine.configLocalCameraPublish(true);
// 配置推送屏幕流，可根据需要开启
aliRtcEngine.configLocalScreenPublish(true);
//启动推流,注意publish是异步接口需要收到onPublishResult之后且result为0才能算推流成功，请参考设置set
RtcEngineEventListener接口的相应回调
aliRtcEngine.publish();
```

> **注意** 在调用publish接口前必须确保加入频道已经成功，否则调用publish接口会失败。

6. 远端用户推流、停止推流消息回调（学生端收到远端用户推流消息的回调与教师端相同）。

```
/**
 * 当远端用户的流发生变化时，返回这个消息
 * @note 远方用户停止推流，也会发送这个消息
 * @param uid User ID
 * @param audioTrack 音频
 * @param videoTrack 视频
 */
void onRemoteTrackAvailableNotify(String uid, AliRtcAudioTrack audioTrack, AliRtcVideoTrack videoTrack);
```

说明

- audioTrack取值为AliRtcAudioTrackNo并且videoTrack取值为AliRtcVideoTrackNo时，表示远端用户停止推流；其他情况表示远端用户对应的推流状态。
- 加入频道之后也可设置角色，设置成功后可收到回调。

  ```
  void onUpdateRoleNotify(AliRtcEngine.AliRTCSDK_Client_Role oldRole,
  AliRtcEngine.AliRTCSDK_Client_Role newRole);
  ```

- 需要业务侧关注一下当前的角色参数值，例如当前是AliRtcClientRoleInteractive角色，再次调用以下接口将不会收到回调，建议业务侧将角色切换和推拉流的逻辑区分开。

  ```
  aliRtcEngine.setClientRole(AliRTCSDK_Client_Role.AliRTCSDK_Interactive);
  ```

学生端

学生加入频道之前设置频道模式为互动模式。

```
AliRtcEngine aliRtcEngine = AliRtcEngine.getInstance(getApplicationContext());
aliRtcEngine.setChannelProfile(AliRTCSDK_Channel_Profile.AliRTCSDK_Interactive_live);
```

普通观众：设置为互动模式后，默认角色类型为“仅观看角色”，即只可拉流观看直播，不能推流。此角色适用于普通观众，无需其他操作。

连麦观众：角色类型为“参与互动角色”，可以推流，操作步骤如下所示：

1. 开始上麦。
   i. 如果普通观众需要上麦切换为连麦观众的话，需要先切换用户角色。

ii. 切换角色为AliRTCSDK_Interactive（需要在调用publish接口进行推流前）。

```
aliRtcEngine.setClientRole(AliRTCSDK_Client_Role.AliRTCSDK_Interactive);
```

切换成功后可收到回调。

```
/**
   * 当前角色变化通知。当本地用户在加入频道后调用 setClientRole 切换角色成功时会触发此回调
   *
   * @ param oldRole 切换前的角色
   * @ param newRole 切换后的角色
   * @ return 无
   */
  void onUpdateRoleNotify(AliRtcEngine.AliRTCSDK_Client_Role oldRole, AliRtcEngine.AliRTCSDK_
Client_Role newRole);
```

iii. 收到回调后，调用publish接口开始推流。

```
// 配置推送音频流，可根据需要开启
aliRtcEngine.configLocalAudioPublish(true);
// 配置推送摄像头流，可根据需要开启
aliRtcEngine.configLocalCameraPublish(true);
// 配置推送屏幕流，可根据需要开启
aliRtcEngine.configLocalScreenPublish(true);
// 启动推流，publish是异步接口需要收到onPublishResult之后且result为0才能算推流成功，请参考设置s
etRtcEngineEventListener接口的相应回调
aliRtcEngine.publish();
```

2. 下麦。

与上麦逻辑相反，如果连麦观众需要下麦切换为普通观众的话，需要先停止推流，再进行角色切换。

i. 停止推流。

```
// 配置推送音频流，可根据需要关闭
aliRtcEngine.configLocalAudioPublish(false);
// 配置推送摄像头流，可根据需要关闭
aliRtcEngine.configLocalCameraPublish(false);
// 配置推送屏幕流，可根据需要关闭
aliRtcEngine.configLocalScreenPublish(false);
// 启动推流，publish是异步接口需要收到onPublishResult之后且result为0才能算推流成功，请参考设置s
etRtcEngineEventListener接口的相应回调
aliRtcEngine.publish();
```

ii. 停止推流收到回调之后，即可切换为普通观众模式。

```
aliRtcEngine.setClientRole(AliRTCSDK_Client_Role.AliRTCSDK_live);
```

iii. 切换成功后可收到onUpdateRoleNotify回调，继续保持拉流观看。

11.2. iOS

通过阅读本文，您可以了解实现教师端和学生端互动模式的接入流程。

教师端

1. 在教师加入频道之前设置频道模式为互动模式。

```
[self.engine setChannelProfile:AliRtcChannelProfileInteractivelive];
```

2. 在教师加入频道之前设置为非自动发布和自动订阅模式。

```
[self.engine setAutoPublish:NO withAutoSubscribe:YES];
```

3. 在教师加入频道之前设置角色为AliRtcClientRoleInteractive。

```
[self.engine setClientRole:AliRtcClientRoleInteractive];
```

AliRtcClientRole参数取值：

- AliRtcClientRoleInteractive：参与互动角色，可以推拉流。
- AliRtcClientRolelive：仅观看角色，只能拉流，适用于普通观众。

> **说明** setClientRole接口暂时不要处理接口返回值。

4. 加入频道。

```
[self.engine joinChannel:authInfo name:name onResult:^(NSInteger errCode){
  if (errCode == 0) {
  //加入频道成功
  } else {
  //加入频道失败
  }
}];
```

5. 调用publish接口开始发布。

```
[self.engine configLocalAudioPublish:YES];
[self.engine configLocalCameraPublish:YES];
[self.engine publish:^(int err) {
}];
```

> **注意** 在调用publish接口前必须确保加入频道已经成功，否则调用publish接口会失败。

6. 远端用户推流、停止推流消息回调（学生端收到远端用户推流消息的回调与教师端相同）。

```
/**
 * @brief 当远端用户的流发生变化时，返回这个消息
 * @note 远方用户停止推流，也会发送这个消息
 */
- (void)onRemoteTrackAvailableNotify:(NSString *)uid audioTrack:(AliRtcAudioTrack)audioTrack video
Track:(AliRtcVideoTrack)videoTrack;
```

> 说明
>
> - audioTrack取值为AliRtcAudioTrackNo并且videoTrack取值为AliRtcVideoTrackNo时，表示远端用户停止推流；其他情况表示远端用户对应的推流状态。
> - 加入频道之后也可设置角色，设置成功后可收到回调。
>
> ```
> - (void)onUpdateRoleNotifyWithOldRole:(AliRtcClientRole)oldRole newRole:(AliRtcClientRole)newRole;
> ```
>
> - 需要业务侧关注一下当前的角色参数值，例如当前是AliRtcClientRoleInteractive角色，再次调用以下接口将不会收到回调，建议业务侧将角色切换和推拉流的逻辑区分开。。
>
> ```
> [self.engine setClientRole:AliRtcClientRoleInteractive];
> ```

学生端

学生加入频道之前设置频道模式为互动模式。

```
[self.engine setChannelProfile:AliRtcInteractivelive];
```

普通观众：设置为互动模式后，默认角色类型为“仅观看角色”，也就是只可拉流观看直播，不能推流。此角色适用于普通观众，无需其他操作。

连麦观众：角色类型为“参与互动角色”，可以推流，操作步骤如下所示：

1. 开始上麦。
   i. 如果普通观众需要上麦切换为连麦观众的话，需要先切换用户角色。
   ii. 切换角色为AliRtcClientRoleInteractive（需要在调用publish接口进行推流前）。

   ```
   [self.engine setClientRole:AliRtcClientRoleInteractive];
   ```

   切换成功后可收到回调。

   ```
   - (void)onUpdateRoleNotifyWithOldRole:(AliRtcClientRole)oldRole newRole:(AliRtcClientRole)newRole;
   ```

   iii. 收到回调后，调用publish接口开始推流。

   ```
   [self.engine configLocalAudioPublish:YES];
   [self.engine configLocalCameraPublish:YES];
   [self.engine publish:^(int err) {
   }];
   ```

2. 下麦。

   与上麦逻辑相反，如果连麦观众需要下麦切换为普通观众的话，需要先停止推流，再进行角色切换。

   i. 停止推流。

   ```
   [self.engine configLocalAudioPublish:NO];
   [self.engine configLocalCameraPublish:NO];
   [self.engine publish:^(int err) {
   }];
   ```

ii. 停止推流收到回调之后，即可切换为普通观众角色。

```
[self.engine setClientRole:AliRtcClientRolelive];
```

iii. 切换成功后可收到回调。

```
- (void)onUpdateRoleNotifyWithOldRole:(AliRtcClientRole)oldRole newRole:(AliRtcClientRole)new
Role;
```

11.3. Windows

通过阅读本文，您可以了解实现教师端和学生端互动模式的接入流程。

教师端

1. 在教师加入频道之前设置频道模式为互动模式。

```
pEngine->setChannelProfile(AliRtcChannelProfileInteractiveLive);
```

2. 在教师加入频道之前设置为非自动发布和自动订阅模式。

```
pEngine->setAutoPublishSubscribe(false, true);
```

3. 在教师加入频道之前设置角色为AliRtcClientRoleInteractive。

```
pEngine->setClientRole(AliRtcClientRoleInteractive);
```

AliRtcClientRole参数取值：

- AliRtcClientRoleInteractive：参与互动角色，可以推拉流。
- AliRtcClientRolelive：仅观看角色，只能拉流，适用于普通观众。

> **说明** setClientRole接口暂时不要处理接口返回值。

4. 加入频道。

```
pEngine->joinChannel(aliRtcAuthInfo, userName);
```

5. 调用publish接口开始推流。

```
// 配置推送音频流，可根据需要开启
pEngine->configLocalAudioPublish(true);
// 配置推送摄像头流，可根据需要开启
pEngine->configLocalCameraPublish(true);
// 配置推送屏幕流，可根据需要开启
pEngine->configLocalScreenPublish(true);
//启动推流,注意publish是异步接口需要收到onPublishResult之后且result为0才能算推流成功，请参考设置set
RtcEngineEventListener接口的相应回调
pEngine->publish();
```

> **注意** 在调用publish接口前必须确保加入频道已经成功，否则调用publish接口会失败。

6. 远端用户推流、停止推流消息回调（学生端收到远端用户推流消息的回调与教师端相同）。

```
/**
 * @brief 当远端用户的流发生变化时，返回这个消息
 * @param uid      User ID。从App server分配的唯一标示符
 * @param audioTrack      音频流类型，详见AliRtcAudioTrack
 * @param videoTrack      视频流类型，详见AliRtcVideoTrack
 * @note 远方用户停止推流，也会发送这个消息
 */
virtual void onRemoteTrackAvailableNotify(const AliRtc::String &uid,
                                          AliRtcAudioTrack audioTrack,
                                          AliRtcVideoTrack videoTrack)
```

> 说明
>
> - audioTrack取值为AliRtcAudioTrackNo并且videoTrack取值为AliRtcVideoTrackNo时，表示远端用户停止推流；其他情况表示远端用户对应的推流状态。
> - 加入频道之后也可设置角色，设置成功后可收到回调。
>
>   ```
>   void onUpdateRoleNotify(const AliRtcClientRole old_role, const AliRtcClientRole new_role)
>   ```
>
> - 需要业务侧关注一下当前的角色参数值，例如当前是AliRtcClientRoleInteractive角色，再次调用以下接口将不会收到回调。
>
>   ```
>   pEngine->setClientRole(AliRtcClientRoleInteractive);
>   ```
>
>   建议业务侧将角色切换和推拉流的逻辑区分开。

学生端

学生加入频道之前设置频道模式为互动模式。

```
AliRtcEngine *pEngine = AliRtcEngine::sharedInstance(listener, "");
pEngine->setChannelProfile(AliRtcChannelProfileInteractiveLive);
```

普通观众：设置为互动模式后，默认角色类型为“仅观看角色”，即只可拉流观看直播，不能推流。此角色适用于普通观众，无需其他操作。

连麦观众：角色类型为“参与互动角色”，可以推流，操作步骤如下所示：

1. 开始上麦。
   i. 如果普通观众需要上麦切换为连麦观众的话，需要先切换用户角色。
   ii. 切换角色为AliRtcClientRoleInteractive（需要在调用publish接口进行推流前）。

   ```
   pEngine->setClientRole(AliRtcClientRoleInteractive);
   ```

   切换成功后可收到回调。

   ```
   void onUpdateRoleNotify(const AliRtcClientRole old_role, const AliRtcClientRole new_role) {};
   ```

iii. 收到回调后，调用publish接口开始推流。

```
pEngine->configLocalAudioPublish(true);
pEngine->configLocalCameraPublish(true);
auto callback = [](void *opaque, int errorCode) {
  if (errorCode == 0) {
    // 推流成功
  } else {
    // 推流失败
  }
};
pEngine->publish(callback, this);
```

2. 下麦。

与上麦逻辑相反，如果连麦观众需要下麦切换为普通观众的话，需要先停止推流，再进行角色切换。

i. 停止推流。

```
pEngine->configLocalAudioPublish(false);
pEngine->configLocalCameraPublish(false);
auto callback = [](void *opaque, int errorCode) {
};
pEngine->publish(callback, this);
```

ii. 停止推流收到回调之后，即可切换为普通观众角色。

```
pEngine->setClientRole(AliRtcClientRoleLive);
```

iii. 切换成功后可收到回调，继续保持拉流观看。

```
void onUpdateRoleNotify(const AliRtcClientRole old_role, const AliRtcClientRole new_role)
```

11.4. Web

通过阅读本文，您可以了解到Web端互动模式的接入流程。

操作步骤

1. 实例化对象。

```
var aliWebrtc = new AliRtcEngine();
```

2. 设置onUpdateRole（角色切换）监听。

> **说明** 在互动模式下，当调用setClientRole接口返回true，并且收到onUpdateRole回调获取到正确新值时才表示切换成功。

```
aliWebrtc.on("onUpdateRole",(data) => {
  console.log(data);
})
```

3. 设置用户角色。

该方法在加入频道前后都可以调用，调用前您必须先停止推流。

```
/**
 * @param role 角色，0：互动角色；1：观众角色
 * @return 调用成功返回true，失败返回false
 */
aliWebrtc.setClientRole(role);
```

4. 设置频道类型。

> 说明 接入互动模式，需要将channelProfile设置为1。

```
/**
 * @param channelProfile 频道类型，0：通信模式；1：互动模式；2：低延迟互动直播模式
 * @return 调用成功返回true，失败返回false
 */
aliWebrtc.setChannelProfile(channelProfile)
```

5. 检测浏览器是否支持RTC SDK。

```
aliWebrtc.isSupport().then(re => {
  console.log(re);
}).catch(err => {
 console.log(err)
})
```

更多信息，请参见isSupport。

6. 加入频道。

成功加入频道后，互动角色可以进行推流，观众角色不能进行推流。

```
/**
 * @param authInfo 鉴权信息
 * @param displayName 显示名称
 */
aliWebrtc.joinChannel(authInfo,displayName).then(()=>{
  //加入频道成功
}).catch((err)=>{
 if(err.errCode === 33622275){
  //加入频道失败，频道类型错误
 }else{
  //加入频道失败，其他错误
 }
})
```

> 说明 authInfo为服务端下发的鉴权信息，生成Token操作及authInfo参数，请参见服务端生成Token方式。

更多接口详情，请参见AliRtcEngine接口。

12.设置美颜

12.1. Android

阿里云RTC SDK为您提供基础美颜功能和第三方美颜接入功能的接口和回调，您可以通过本文了解相关流程。

基础美颜

- 功能简介

  目前阿里云RTC支持基础美颜的版本为1.17.9以上版本。

  在直播、视频通话、视频会议等场景中，美颜效果能更好的提升用户体验，阿里云RTC基础美颜提供了美白和磨皮两种功能。

- 实现方法

  阿里云RTC SDK通过setBeautyEffect方法设置是否启用基础美颜。

```
public abstract int setBeautyEffect(boolean enable, AliRtcEngine.AliRtcBeautyConfig config);
```

参数	类型	描述
enable	boolean	true表示开启，false表示关闭，默认为关闭。
config	AliRtcBeautyConfig	基础美颜参数。

第三方美颜

- 功能简介

  目前阿里云RTC支持美颜接入的版本为1.14.0以上版本。

  通常美颜SDK接受如下两种类型的数据回调处理：

  - YUV裸数据用于做人脸识别。
  - openGL纹理数据用于最终做美颜效果的处理。

- 接入流程

  为了避免完成集成SDK后，自己能看到美颜效果，对方却看不到美颜效果。在接入前，我们需要在SDK的实例extra字段中添加开关，设置user_specified_video_preprocess为TRUE。

  i. SDK的instance构造函数里面，扩展字段添加jsonObject.addProperty("user_specified_video_preprocess", "TRUE")，并且将jsonObject转化成string后传入到AliRtcEngine.getInstance的第二个字段extra里面。

  ii. 在调用本地预览开启接口startPreview之后，调用RegisterTexturePreObserver订阅openGL纹理数据。

  > **说明** 通常是对本地用户进行美颜，callid填写为双引号（""）字符串即可。

  iii. 需要对接YUV数据人脸识别功能时候：在调用本地预览开启接口startPreview之后，再调用RegisterPreprocessVideoObserver订阅采集前处理YUV数据（通常是对采集图像做人脸识别）。

iv. 需要对接YUV数据人脸识别功能时候：在AliDetectObserver的onData回调中做第三方算法的人脸识别操作，返回的long为人脸识别之后的该图像的人脸结构体指针。

v. 在传入observer函数回调onTextureCreate中第三方算法的初始化工作，其中context为openGL的上下文。

vi. 在传入observer函数回调onTexture做第三方算法的每一帧美颜处理工作，如果不需要美颜或者第三方算法处理不成功，请将输入的textureId返回给该函数。如果美颜处理成功，则返回第三方算法处理过的textureId。

vii. 在传入observer函数回调onTextureDestroy做第三方算法的销毁工作。

第三方美颜接口调用

- RTC SDK YUV裸数据人脸识别接入接口，人脸识别接入时，需要订阅采集之后的前处理buffer数据，所以在startPreview之后需要调用RegisterPreprocessVideoObserver接口获取采集前处理数据并处理。

```
///@brief register preprocess observer
///@param observer 回调接口
void RegisterPreprocessVideoObserver(AliDetectObserver observer);
```

订阅成功后，将会回调AliDetectObserver的onData参数。

```
///@brief 采集前处理回调接口
///@param dataFrameY Y分量指针
///@param dataFrameU U分量指针
///@param dataFrameV V分量指针，NV12和NV21该指针为null
///@param format，图像数据格式，Android输出的YUV数据格式为NV21
///@param width，图像宽度
///@param height，图像高度
///@param strideY，图像Y分量stride
///@param strideU，图像U分量stride
///@param strideV，图像V分量stride
///@param rotate，图像旋转角度
///@param extraData，附加字段（非定制化可忽略）
///@return 人脸识别结构体指针（第三方定义结构体），SDK只做传递 （该返回值传递人脸数据）
long onData(long dataFrameY, long dataFrameU, long dataFrameV, AliRTCImageFormat format, int width,
int height, int strideY, int strideU, int strideV, int rotate, long extraData);
```

- RTC SDK openGL纹理接口在美颜接入时，需要订阅视频的纹理数据，所以在startPreview之后需要调用接口获取openGL的纹理数据和openGL的线程环境。 需要调用的接口如下所示：
  - RegisterTexturePreObserver接口。

```
///@brief 订阅openGL的纹理数据
///@param[in] uid 订阅的用户，通常本地需要美颜，填写""或者本地uid都可以
///@param[in] observer 回调接口
void RegisterTexturePreObserver(String callId, AliTextureObserver observer);
```

  - onTextureCreate回调（相芯SDK不处理此回调）。

```
///@brief 表示本地视频流纹理创建
///@param[in] callId 订阅的用户，RegisterTexturePreObserver所填写的callId
///@param[in] context openGL的上下文EGLContext指针
void onTextureCreate(String callId, long context);
```

- onTexture回调。

```
///@brief 表示每一帧视频流处理
///@param[in] callId 订阅的用户，RegisterTexturePreObserver所填写的callId
///@param[in] textureId 视频流纹理输入的ID
///@param[in] width 视频流纹理的宽度
///@param[in] height 视频流纹理的高度
///@param[in] stride 视频流纹理的stride
///@param[in] rotate 视频流纹理的rotate角度
///@param[in] extraData 检测的人脸数据，带人脸识别时：为AliDetectObserver的OnData返回的人脸结构数据指针；不带人脸识别时：为0
///@param[out] 处理之后的texture id，如果不需要美颜，请把textureId输入的texture id传回
int onTexture(String callId, int textureId, int width, int height, int stride, int rotate, long extraData);
```

- onTextureDestroy回调（相芯SDK不处理此回调）。

```
///@brief 表示本地视频流纹理销毁
///@param[in] callId 订阅的用户，RegisterTexturePreObserver所填写的callId
void onTextureDestroy(String callId);
```

12.2. iOS

阿里云RTC SDK为您提供基础美颜功能和第三方美颜接入功能的接口和回调，您可以通过本文了解相关流程。

基础美颜

- 功能简介

  目前阿里云RTC支持基础美颜的版本为1.17.9以上版本。

  在直播、视频通话、视频会议等场景中，美颜效果能更好的提升用户体验，阿里云RTC基础美颜提供了美白和磨皮两种功能。

- 实现方法

  阿里云RTC SDK通过setBeautyEffect方法设置是否启用基础美颜。

```
/**
* 设置美颜
* @param enable 美颜开关
* @param config 美颜参数控制
*/
- (int)setBeautyEffect:(BOOL)enable config:(AliRtcBeautyConfig)config;
```

参数	类型	描述
enable	boolean	true表示开启，false表示关闭，默认为关闭。
config	AliRtcBeautyConfig	基础美颜参数。

第三方美颜

- 功能简介

目前阿里云RTC支持美颜接入的版本为1.14.0以上版本。

通常美颜SDK接受如下两种类型的数据回调处理。

- YUV裸数据用于做人脸识别。
- openGL纹理数据用于最终做美颜效果的处理。

- 接入流程

为了避免完成集成SDK后，自己能看到美颜效果，对方却看不到美颜效果。在接入前，我们需要在SDK的实例extra字段中添加开关，设置user_specified_video_preprocess为TRUE。

i. SDK的instance构造函数里面，扩展字段添加[extrasDic setValue:@"TRUE" forKey:@"user_specified_video_preprocess"]，并且将extraDic经过序列化之后传入到AliRtcEngine sharedInstance的第二个字段extra中。

ii. 在调用本地预览开启接口startPreview之后，调用subscribeVideoTexture订阅openGL纹理数据。

> **说明** 通常是对本地用户进行美颜，uid填写为英文半角双引号（""）字符串即可。

iii. 需要对接YUV数据人脸识别功能时候：在调用本地预览开启接口startPreview之后，再调用subscribeVideoPreprocessData订阅采集前处理YUV数据（通常是对采集图像做人脸识别）。

iv. 需要对接YUV数据人脸识别功能时候：在onVideoDetectCallback回调中做第三方算法的人脸识别操作，返回的long为人脸识别之后的该图像的人脸结构体指针。

v. 在onVideoTextureCreated中做第三方算法的初始化工作，其中context为openGL的上下文。

vi. 在onVideoTexture做第三方算法的每一帧美颜处理工作，如果不需要美颜或者第三方算法处理不成功，请将输入的textureId返回给该函数。如果美颜处理成功，则返回第三方算法处理过的textureId。

vii. 在onVideoTextureDestory做第三方算法的销毁工作。

第三方美颜接口调用

- RTC SDK YUV裸数据人脸识别接入接口，人脸识别接入时，需要订阅采集之后的前处理buffer数据，所以在startPreview之后需要调用subscribeVideoPreprocessData接口获取采集前处理数据并处理。

```
///@brief 订阅采集视频前处理裸数据
///@param videoSource 订阅本地采集的哪一路流，移动端场景填写AliRtcVideosourceCameraLargeType
- (void)subscribeVideoPreprocessData:(AliRtcVideoSource)videoSource;
```

onVideoDetectCallback回调方法如下：

> **说明** 订阅成功后，通过delegate回调本地采集数据。

```
///@brief RTC采集视频数据前处理回调
///@param type 视频流类型
///@param videoFrame 视频数据帧
///@return 人脸识别结构体指针（第三方定义结构体），SDK只做传递 （该返回值传递人脸数据）
- (long)onVideoDetectCallback:(AliRtcVideoSource)type videoFrame:(AliRtcVideoDataSample *)videoFrame;
```

AliRtcVideoDataSample对象定义如下：

```
@interface AliRtcVideoDataSample : NSObject
@property (nonatomic, assign) AliRtcImageFormat format; //iOS输出的YUV数据格式为NV12
@property (nonatomic, assign) AliRtcVideoDataType dataType;
@property (nonatomic, assign) long dataPtr;
@property (nonatomic, assign) long dataYPtr;
@property (nonatomic, assign) long dataUPtr;
@property (nonatomic, assign) long dataVPtr;
@property (nonatomic, assign) int strideY;
@property (nonatomic, assign) int strideU;
@property (nonatomic, assign) int strideV;
@property (nonatomic, assign) int height;
@property (nonatomic, assign) int width;
@property (nonatomic, assign) int rotation;
@property (nonatomic, assign) int stride;
@property (nonatomic, assign) long long timeStamp;
@end
```

- RTC SDK openGL 纹理接口美颜接入时，需要订阅视频的纹理数据，所以在startPreview之后需要调用接口获取openGL的纹理数据和openGL的线程环境。需要调用的接口如下所示：
  - subscribeVideoTexture接口。

```
///@brief 订阅openGL的纹理数据
///@param[in] uid 订阅的用户，通常本地需要美颜，填写""或者本地uid都可以
///@param[in] videoSource 订阅本地pub的哪一路流，移动端场景填写AliRtcVideosourceCameraLargeTyp
e
///@param[in] videoTextureType 美颜的场景选择AliRtcVideoTextureTypePre
- (void)subscribeVideoTexture:(NSString *)uid videoSource:(AliRtcVideoSource)videoSource videoText
ureType:(AliRtcVideoTextureType)videoTextureType;
```

  - onVideoTextureCreated回调（相芯SDK不处理此回调）。

```
///@brief 表示本地视频流纹理创建
///@param[in] uid 订阅的用户，subscribeVideoTexture所填写的uid
///@param[in] videoTextureType，subscribeVideoTexture所填写的videoTextureType
///@param[in] context openGL的上下文EAGLContext指针
- (void)onVideoTextureCreated:(NSString *)uid videoTextureType:(AliRtcVideoTextureType)videoText
ureType context:(void *)context
```

  - onVideoTexture回调。

```
///@brief 表示每一帧视频流处理
///@param[in] uid 订阅的用户，subscribeVideoTexture所填写的uid
///@param[in] videoTextureType，subscribeVideoTexture所填写的videoTextureType
///@param[in] textureId 视频流纹理的输入texture id
///@param[in] width 视频流纹理的宽度
///@param[in] height 视频流纹理的高度
///@param[in] extraData 检测的人脸数据， 带人脸识别时：为onVideoDetectCallback返回的人脸结构数据指
针；不带人脸识别时：为0
///@param[out] 处理之后的texture id，如果不需要美颜，请把textureId输入的texture id传回
- (int)onVideoTexture:(NSString *)uid videoTextureType:(AliRtcVideoTextureType)videoTextureType t
extureId:(int)textureId width:(int)width height:(int)height extraData:(long)extraData
```

- onVideoTextureDestory回调（相芯SDK不处理此回调）。

```
///@brief 表示本地视频流纹理销毁
///@param[in] uid 订阅的用户，subscribeVideoTexture所填写的uid
///@param[in] videoTextureType，subscribeVideoTexture所填写的videoTextureType
- (void)onVideoTextureDestory:(NSString *)uid videoTextureType:(AliRtcVideoTextureType)videoTextureType
```

- 外置美颜SDK相关接口调用。

  外置美颜SDK根据各自的设计，有对应的SDK自己的提供的接口，需要App对接时候正确使用，主要有如下：

  - 初始化、销毁。
  - 资源加载和释放（通常是人脸识别使用的资源文件）。
  - 美颜控制接口（美颜等级、强度的调节接口，美颜开关接口）。
  - 对接RTC SDK回调的接口。

13.旁路转推

13.1. 概述

阿里云RTC为您提供云端音视频的混流、混音服务，可以将实时音视频流转为标准直播流。通过本文，您可以快速了解旁路转推的基本概念和功能说明。

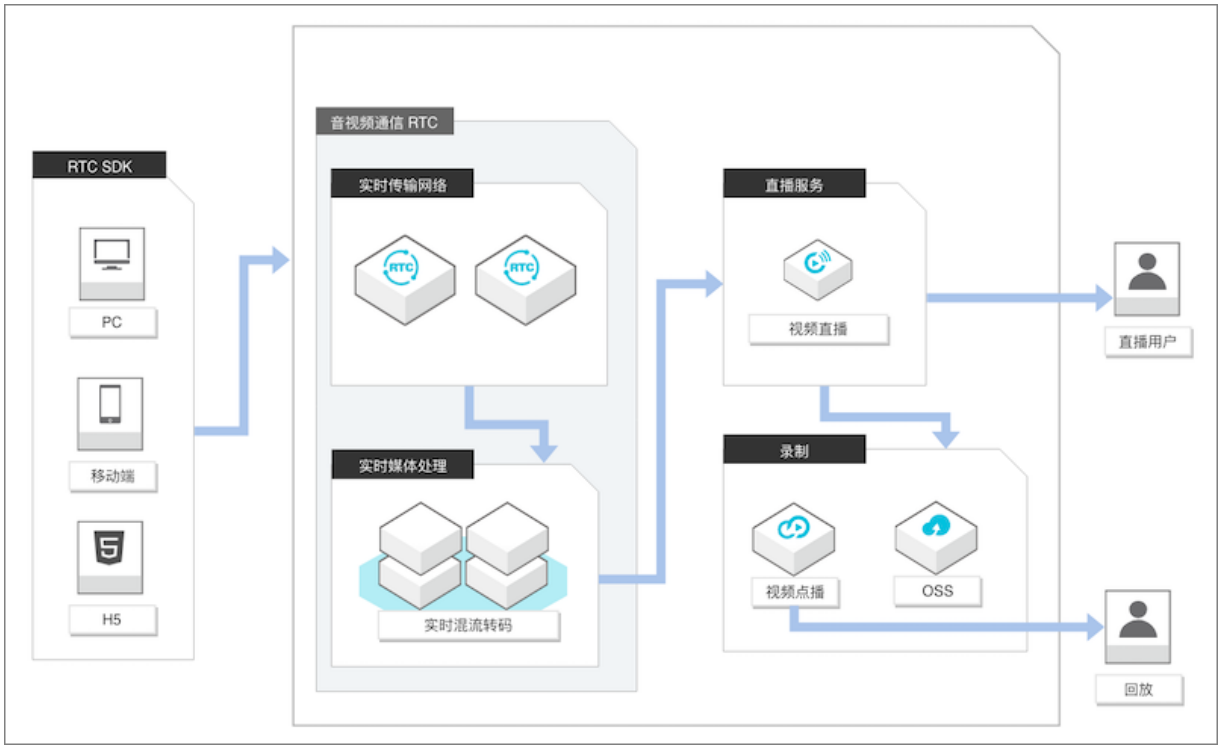
> 说明 旁路转推计费详情请参见旁路转推计费。

旁路转推

将正在进行实时音视频通话时频道的画面同步到云端进行云端混流，并将混流后的频道直播流推流给第三方CDN或者阿里云视频直播，这一过程称为旁路转推。

旁路转推架构图

当您在各端（PC、移动端、Web）集成阿里云RTC并实现音视频通话后，您可以将频道画面进行旁路转推。首先在云端进行混流、转码、编码等操作，然后生成旁路转推的视频流，并推送到视频直播中进行大规模下行加速的直播观看。



功能特性

阿里云RTC的旁路转推功能特性如下所示。

类别	功能	说明
媒体源支持	终端支持	支持Windows、Mac、iOS、Android、H5。

类别	功能	说明
混流、混音	视频布局	提供11组默认布局。具体详情，请参见默认布局。
	布局切换	更新布局中的视频源。
	事件触发	频道内首个终端加入自动触发转码，频道内无终端自动停止转码。
输出流	直播流	协议支持：RTMP。
	媒体规格	输出媒体规格支持： • 1920X1080 30fps。 • 1280X720 25fps。 • 720X540 24fps。 • 640X360 15fps。

13.2. 接入流程

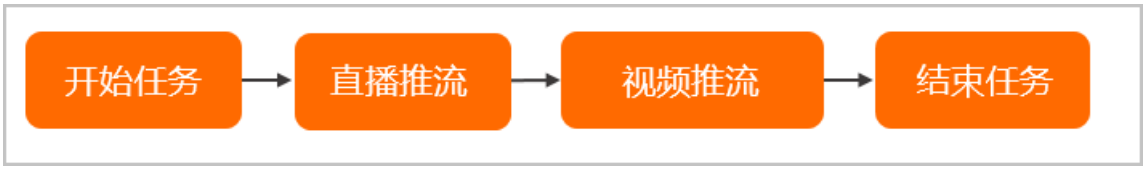
阿里云RTC为您提供全套旁路转推的接入流程。您可以参考本文，搭建旁路转推。

前提条件

接入旁路转推之前，您需要完成以下工作：

- 成功搭建阿里云RTC实时音视频通话SDK。具体详情，请参见入门概述。阿里云RTC为您提供多端SDK Demo，您可以快速跑通Demo。具体详情，请参见Demo。
- 开通视频直播服务并生成推流地址（StreamURL）。具体详情，请参见开通视频直播服务和推流地址和播放地址。

接入流程



步骤	操作	描述
1	开始任务	您需要调用StartMPUTask实现开始任务。
2	旁路转推（系统自动完成）	在视频直播控制台根据推流地址进行推流操作，请参见PC端推流与播流。
3	结束任务	当最后一个人离开频道后30秒自动停止任务，您也可以调用StopMPUTask主动结束任务。

> **说明** 成功开始旁路转推任务后，您可以调用GetMPUTaskStatus获取任务状态或调用UpdateMPULayout（建议您开始任务后调用该接口一次）更新任务布局。

当您成功进行旁路转推时，所输出的视频流会产生计费。阿里云产品只对流出流量进行计费。所以当您从视频直播服务进行拉流时会产生相关费用。计费文档，请参见直播计费。

14.布局说明

布局是指在画布（Canvas）上多个展示元素不同大小、不同位置和叠放关系的描述。本文为您介绍了阿里云12组视频布局的配置，您可以在旁路转推混流前选择适合您的布局配置，开始旁路转推任务，也可以调用UpdateMPULayout进行布局切换。

> **说明**
>
> 布局功能目前仅可以在旁路转推及云端录制中使用。自定义布局可通过控制台和OpenAPI使用。

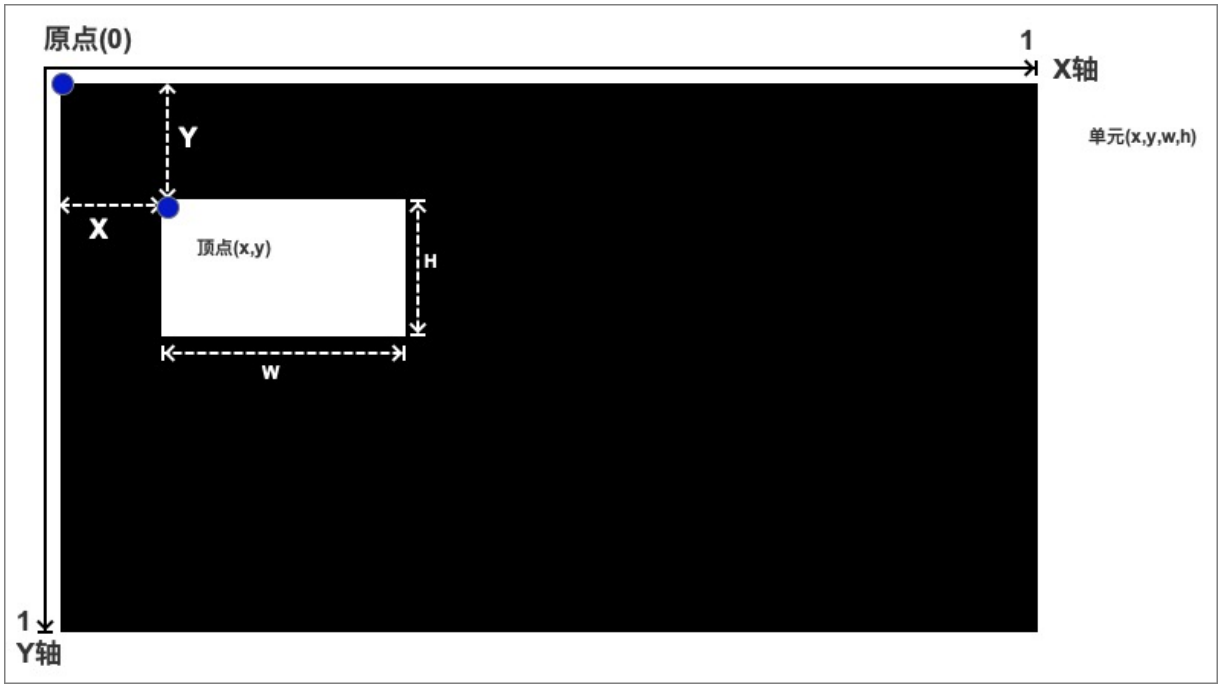
布局组成元素

参数	类型	描述
panes	object array	窗格信息，最多支持16组设置。
audio_mix_count	int	最大混音个数。

窗格信息相关参数如下表所示。

参数	类型	描述
paneid	int	窗格编号。
major_pane	int	1：主窗格。
x	float	坐标x，归一化百分比。
y	float	坐标y，归一化百分比。
width	float	窗格宽，归一化百分比。
height	float	窗格高，归一化百分比。
zorder	int	叠放顺序，0为最底层，1层在0层之上，以此类推。

布局坐标体系



默认布局

阿里云RTC为您提供如下布局参考。

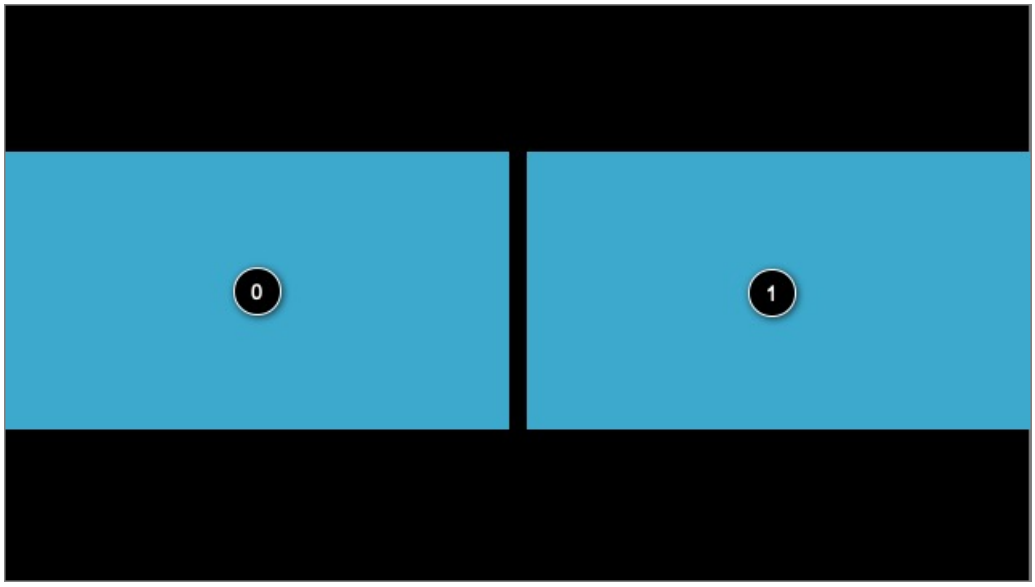
布局ID	视频源个数	布局名称
1	1	单画面1
2	2	左右平铺_1
3	2	画中画1
4	2	画中画2
5	3	画廊模式_3
6	4	画廊模式_4
7	5	画廊模式_5
8	6	画廊模式_6
9	7	画廊模式_7
10	8	画廊模式_8
11	9	画廊模式_9
13	2	左右平铺_2

单画面1

id	x	y	width	height	zorder
0	0	0	1	1	0

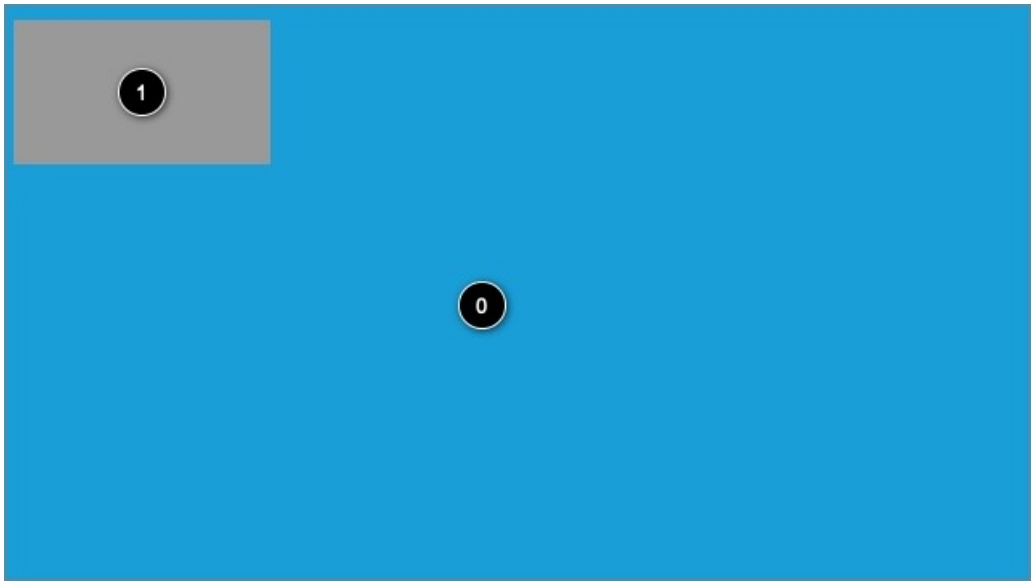
左右平铺_1

id	x	y	width	height	zorder
0	0.0021	0.2516	0.4968	0.4968	0
1	0.501	0.2516	0.4968	0.4968	0



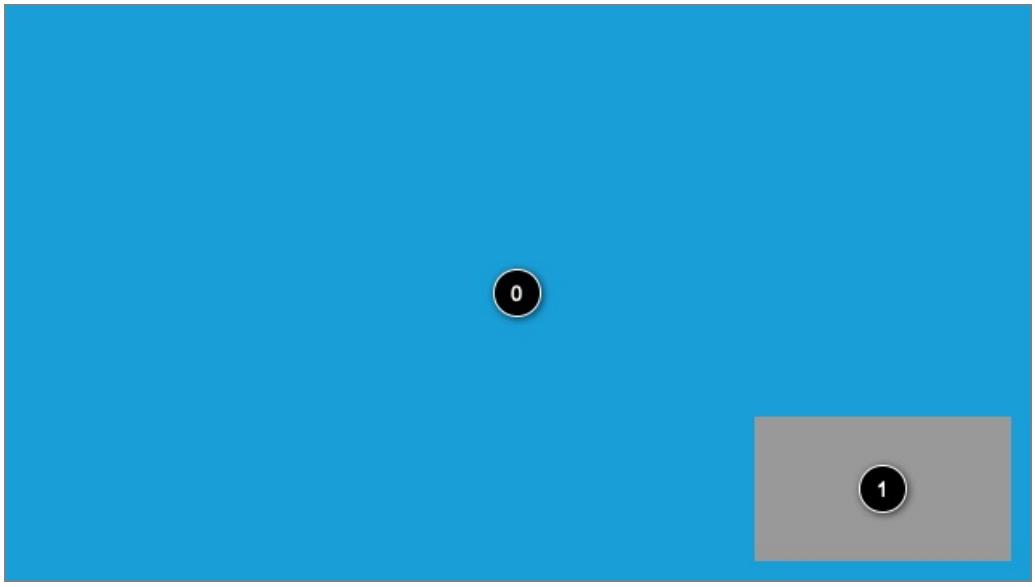
画中画1

id	x	y	width	height	zorder
0	0	0	1	1	0
1	0.00625	0.00625	0.2361	0.2361	1



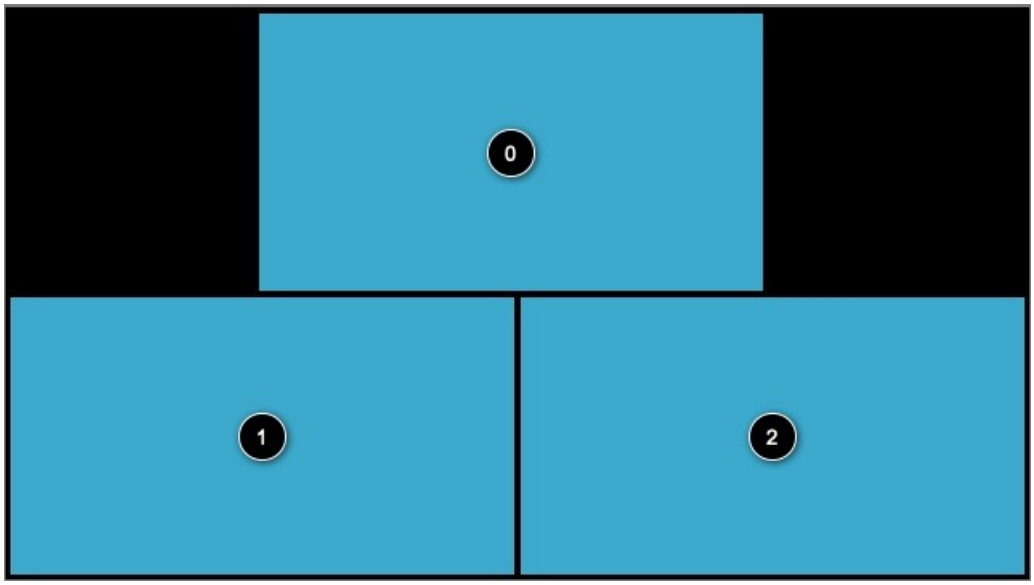
画中画2

id	x	y	width	height	zorder
0	0	0	1	1	0
1	0.7576	0.7576	0.2361	0.2361	1



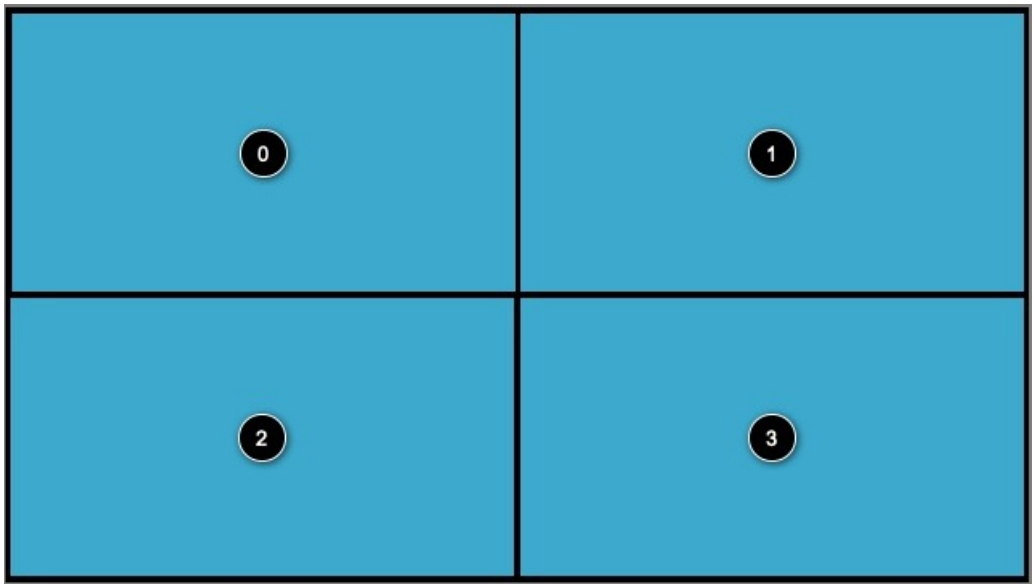
画廊模式_3

id	x	y	width	height	zorder
0	0.2547	0.00625	0.4906	0.4906	0
1	0.00625	0.5031	0.4906	0.4906	0
2	0.5031	0.5031	0.4906	0.4906	0



画廊模式_4

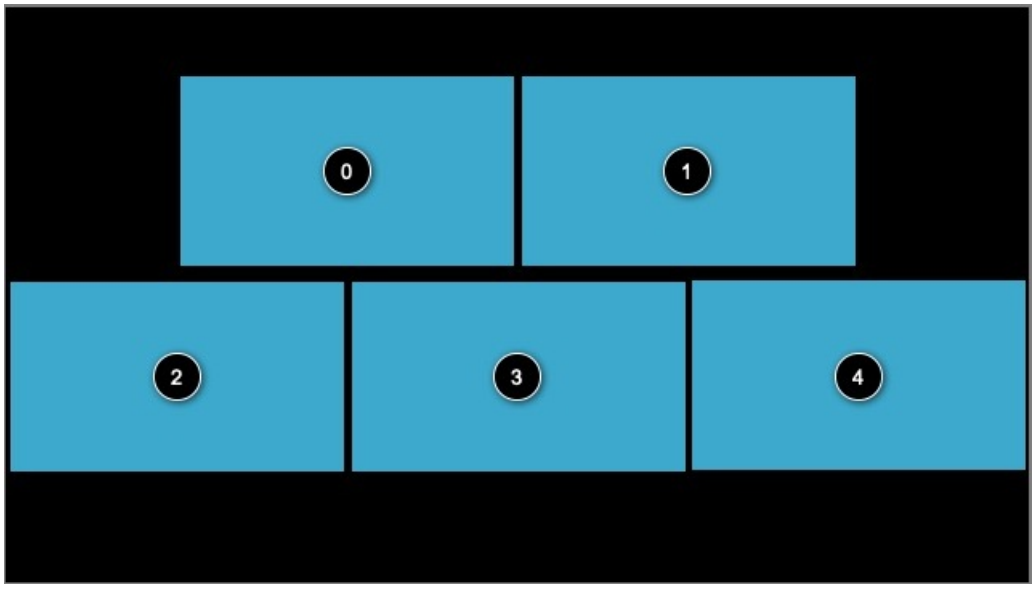
id	x	y	width	height	zorder
0	0.013	0.013	0.4805	0.4805	0
1	0.5065	0.013	0.4805	0.4805	0
2	0.013	0.5065	0.4805	0.4805	0
3	0.5065	0.5065	0.4805	0.4805	0



画廊模式_5

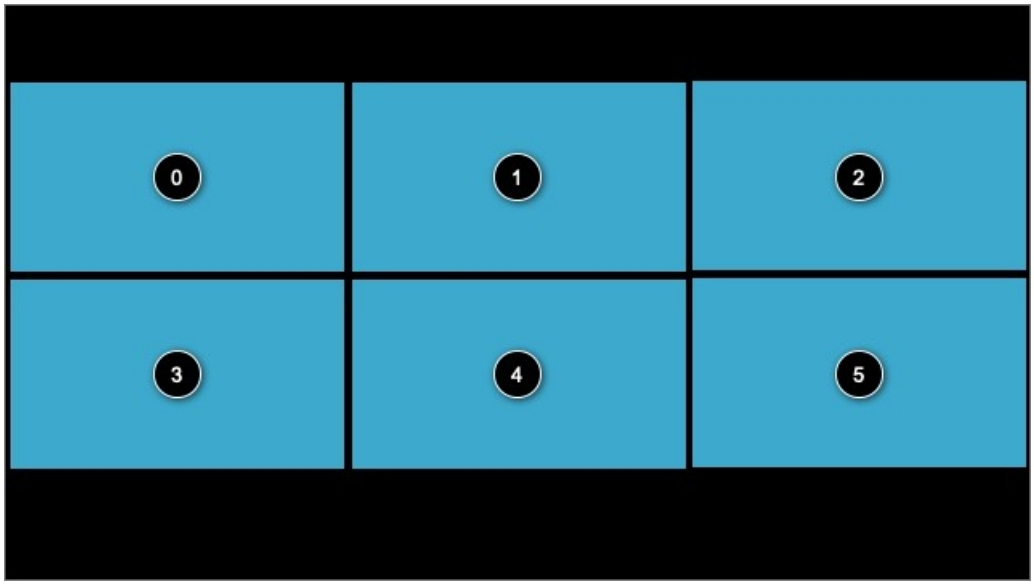
id	x	y	width	height	zorder
0	0.1718	0.1718	0.325	0.325	0

id	x	y	width	height	zorder
1	0.5031	0.1718	0.325	0.325	0
2	0.00625	0.503	0.325	0.325	0
3	0.3375	0.503	0.325	0.325	0
4	0.66875	0.503	0.325	0.325	0



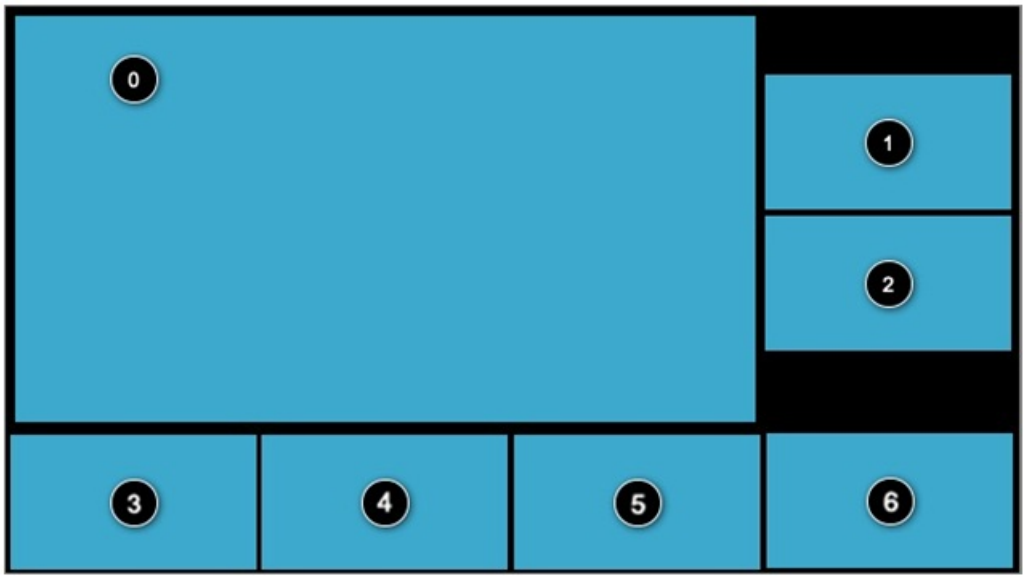
画廊模式_6

id	x	y	width	height	zorder
0	0.00625	0.1718	0.325	0.325	0
1	0.3375	0.1718	0.325	0.325	0
2	0.6688	0.1718	0.325	0.325	0
3	0.00625	0.5031	0.325	0.325	0
4	0.3375	0.5031	0.325	0.325	0
5	0.6688	0.5031	0.325	0.325	0



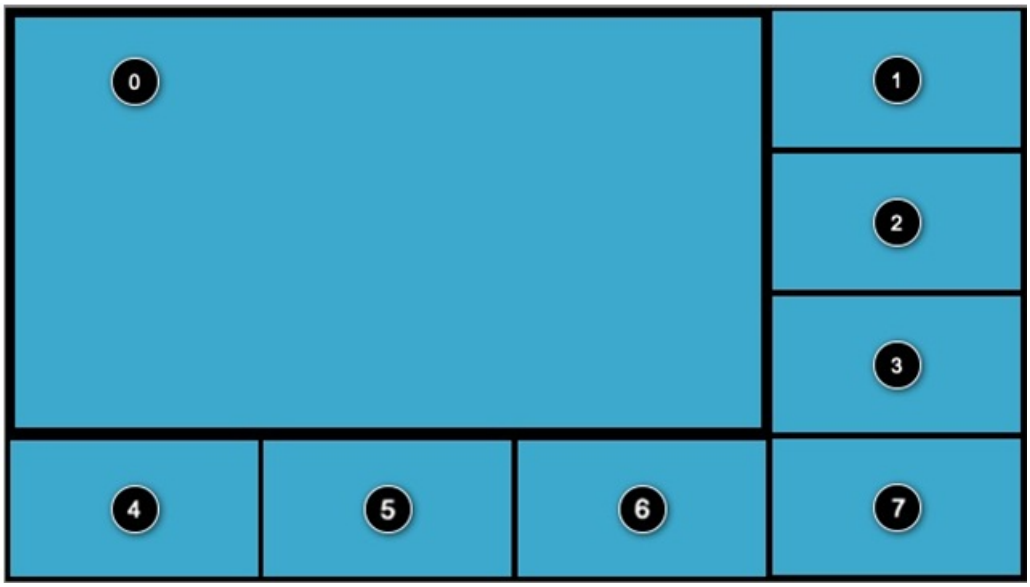
画廊模式_7

id	x	y	width	height	zorder
0	0.00625	0.00625	0.7391	0.7391	0
1	0.7516	0.1336	0.2422	0.2422	0
2	0.7516	0.3821	0.2422	0.2422	0
3	0.00625	0.7516	0.2422	0.2422	0
4	0.2547	0.7516	0.2422	0.2422	0
5	0.5032	0.7516	0.2422	0.2422	0
6	0.7516	0.7516	0.2422	0.2422	0



画廊模式_8

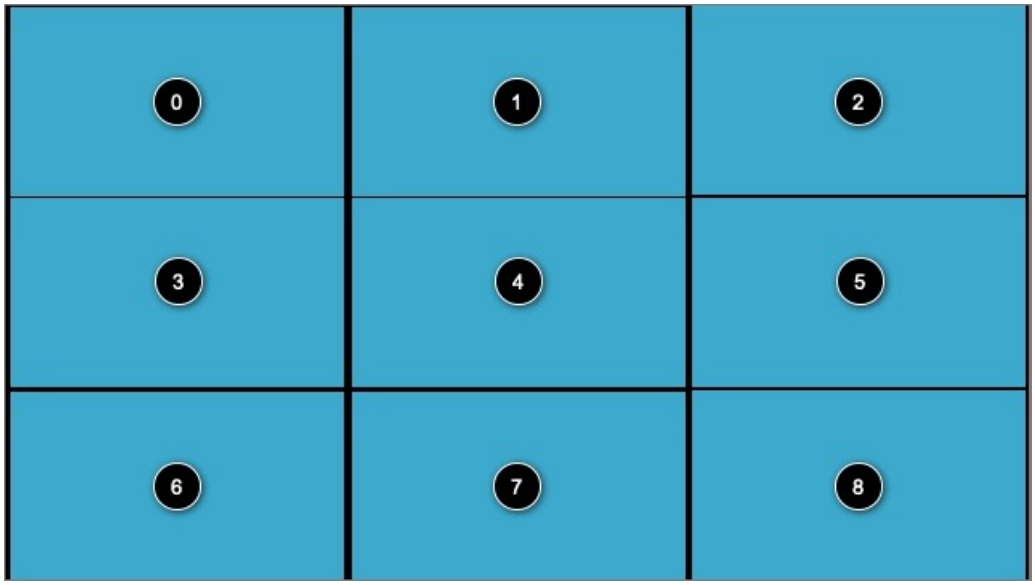
id	x	y	width	height	zorder
0	0.00625	0.00625	0.7391	0.7391	0
1	0.7516	0.00625	0.2422	0.2422	0
2	0.7516	0.2547	0.2422	0.2422	0
3	0.7516	0.5032	0.2422	0.2422	0
4	0.00625	0.7516	0.2422	0.2422	0
5	0.2547	0.7516	0.2422	0.2422	0
6	0.5032	0.7516	0.2422	0.2422	0
7	0.7516	0.7516	0.2422	0.2422	0



画廊模式_9

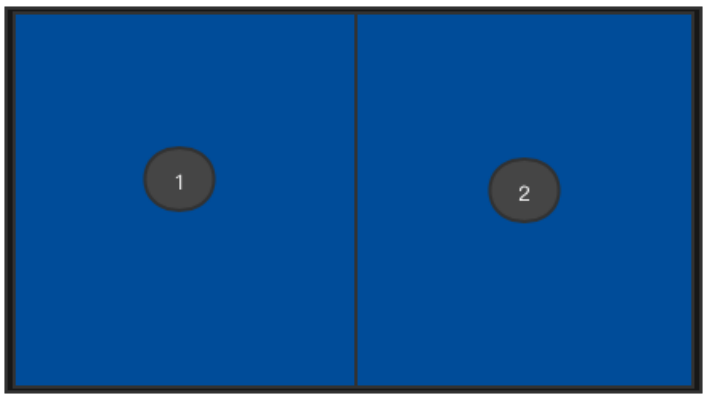
id	x	y	width	height	zorder
0	0.00625	0.00625	0.325	0.325	0
1	0.3375	0.00625	0.325	0.325	0
2	0.6688	0.00625	0.325	0.325	0
3	0.00625	0.3375	0.325	0.325	0
4	0.3375	0.3375	0.325	0.325	0
5	0.6688	0.3375	0.325	0.325	0

id	x	y	width	height	zorder
6	0.00625	0.6688	0.325	0.325	0
7	0.3375	0.6688	0.325	0.325	0
8	0.6688	0.6688	0.325	0.325	0



左右平铺_2

id	x	y	width	height	zorder
0	0	0	0.5	1	0
1	0.5	0	0.5	1	0



相关文档

配置旁路转推布局所涉及的API如下所示。

- StartMPUTask

- UpdateMPULayout

15.云端录制

阿里云RTC为您提供录制服务，将正在进行实时音视频通话时频道的画面同步到云端进行云端混流，并将混流后的频道内容进行录制，同时您也可以根据业务场景自由选择录制的内容。通过本文，您可以快速了解录制服务的基本概念和功能说明。

> **说明** 云端录制服务自2021年4月1日23:00起开始商业收费。价格详情请参见详细价格总览，计费方式请参见云端录制计费。

前提条件

在使用录制服务前，您需要完成以下操作：

- 您已经完成注册阿里云账号，并完成实名认证。具体操作，请参见阿里云账号注册流程
- 您已经开通音视频通信服务。具体操作，请参见开通服务
- 您需要在阿里云音视频通信RTC控制台创建应用。具体操作，请参见创建应用
- 您需要开通阿里云OSS服务存储录制文件。具体操作，请参见开通服务。
- 您需要开通阿里云MNS服务接收录制回调消息。具体操作，请参见开通服务。

流程图

录制存储到OSS

阿里云RTC服务支持将接收到的源视频流进行录制，支持M3U8（同时会有.ts分片文件） 格式，也支持周期录制时长的配置，视频文件会保存至您指定的OSS存储位置。录制任务结束时，自动生成本次的录制索引文件（M3U8文件）。还支持按您指定的录制开始时间和录制结束时间生成自定义录制索引文件。

> **说明**
> - 为了便于您对录制内容进行回看，录制的视频存储在OSS bucket中。本文以录制存储至OSS为例进行说明。
> - 如果将录制的视频存储至OSS中，您需要授权阿里云RTC可将视频内容写入OSS。授权后才能将视频存储至指定的OSS bucket中。
> - 为了避免录制时，因网络抖动或临时断流而导致录制文件被异常截断，系统会延迟断流180s，即如果断流之后在180s内重新推流，系统会默认是同一路录制流，超过180s则认为是两路录制流。

如何配置OSS

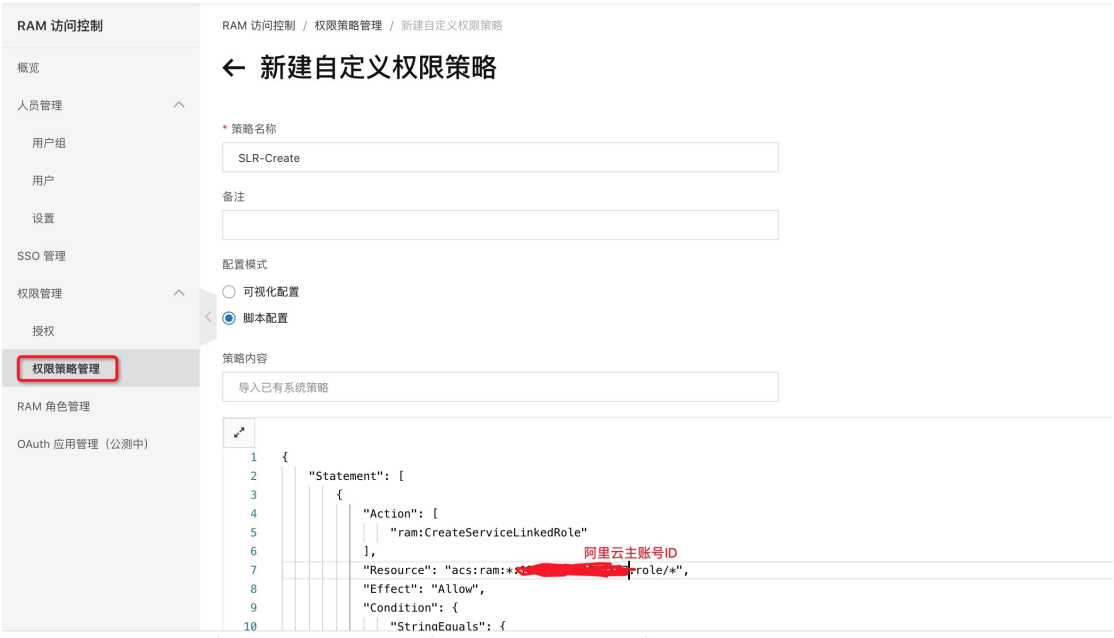
1. 如何配置OSS，请参见配置OSS。
2. 配置RTC写入OSS权限阿里云RTC录制文件保存到用户OSS，需要授权RTC访问用户OSS资源，RTC通过服务关联角色AliyunServiceRoleForRTC访问相应用户资源。

i. 您需要拥有指定的权限，才能自动创建或删除AliyunServiceRoleForRTC。因此在RAM用户无法自动创建AliyunServiceRoleForRTC时，您需为其添加以下权限策略。

```
{
  "Statement": [
    {
      "Action": [
        "ram:CreateServiceLinkedRole"
      ],
      "Resource": "acs:ram:*:主账号ID:role/*",
      "Effect": "Allow",
      "Condition": {
        "StringEquals": {
          "ram:ServiceName": [
            "rtc.aliyuncs.com"
          ]
        }
      }
    }
  ],
  "Version": "1"
}
```

> **说明** 请将主账号ID替换为您实际的阿里云主账号ID。

ii. 登录RAM访问控制，选择**权限策略管理 > 新建自定义权限策略**，为RAM用户创建新的权限策略。



主要接口及功能

接口	描述
AddRecordTemplate	添加录制配置模板

DeleteRecordTemplate	删除录制配置模板
UpdateRecordTemplate	更新录制配置模板
DescribeRecordTemplates	查询录制模板配置列表
DescribeRecordFiles	查询录制的文件列表
StartRecordTask	开始录制视频任务
StopRecordTask	停止视频录制任务
UpdateRecordTask	运行中任务参数更新

16.外部音视频输入

16.1. Android

本文为您介绍了Android的输入外部视频流和音频流的接口示例。

输入外部视频流

> **注意** 增加自定义视频输入时，建议您提前采集，RTC进入频道之前开始送入数据，避免看到摄像头数据。

1. 注册外部视频接口回调。

   接口说明如下所示。

   ```
   /**
    * 注册原始数据回调接口
    * @param streamType 流类型
    * @return dataInterface 视频原始数据接口
    */
   public abstract AliRtcEngine.VideoRawDataInterface registerVideoRawDataInterface(AliRtcEngine.AliRawDataStreamType streamType);
   ```

   > **说明** streamType支持AliRTCSdk_Streame_Type_Capture（视频流）和AliRTCSdk_Streame_Type_Screen（屏幕流）；SDK默认自动推拉流，例如：主动关闭将收不到对应回调。

   示例方法如下所示。

   ```
   //获取AliRtcEngine实例
   AliRtcEngine mAliRtcEngine = AliRtcEngine.getInstance(getApplicationContext());
   AliRtcEngine.VideoRawDataInterface videoRawDataInterface = mAliRtcEngine.registerVideoRawDataInterface(
           AliRtcEngine.AliRawDataStreamType.AliRTCSdk_Streame_Type_Capture);
   ```

2. 输入视频数据。

   接口说明如下所示。

   ```
   /**
    * 发送视频帧
    * @param frame AliRawDataFrame数据帧实体类
    * @param timeStamp 时间戳
    * @return
    */
   public abstract int deliverFrame(AliRawDataFrame frame, long timeStamp);
   ```

   参数说明，AliRawDataFrame必填项如下所示。

参数	说明

参数	说明
MediaStatesVideoFormat	数据类型，目前默认AndroidYUV数据返回NV21
width	视频流宽
height	视频流高
rotation	视频流角度
video_frame_length	数据buffer长度
lineSize	yuv数据数组，具体使用请参见下文示例
frame	buffer数据

示例方法如下所示。

```
private void decodeYUVRawData(AliRtcEngine.VideoRawDataInterface videoRawDataInterface) {
    String yuvPath = "<!-请输入yuv文件地址->";
    if (TextUtils.isEmpty(yuvPath)) {
      ToastUtils.LongToast("请先选择YUV文件！！！");
      return;
    }
    File yuvDataFile = new File(yuvPath);
    if (!yuvDataFile.exists()) {
      ToastUtils.LongToast(yuvPath + " 文件不存在！！！");
      return;
    }
    //此处仅为示例代码做法，请使用正确的流角度
    int rotation = 90;
    Log.d(TAG, "decodeYUVRawData: "+rotation+"__"+getVideoFormat());
    ToastUtils.LongToast("inputing yuv data");
    //此处仅为示例代码做法，请使用正确的流宽高
    int width = 480;
    int height = 640;
    //start push yuv
    new Thread() {
      @Override
      public void run() {
        File yuvDataFile = new File(yuvPath);
        RandomAccessFile raf = null;
        try {
          raf = new RandomAccessFile(yuvDataFile, "r");
        } catch (FileNotFoundException e) {
          e.printStackTrace();
          return;
        }
        try {
          byte[] buffer = new byte[width * height * 3 / 2];
          while (videoRawDataInterface != null) {
            int len = raf.read(buffer);
            if (len == -1) {
              raf.seek(0);
```

```
                }
                AliRtcEngine.AliRawDataFrame rawDataFrame
                    = new AliRtcEngine.AliRawDataFrame();
                rawDataFrame.format = MediaStatesVideoFormat.NV21;// 支持NV21和I420
                rawDataFrame.frame = buffer;
                rawDataFrame.width = width;
                rawDataFrame.height = height;
                rawDataFrame.lineSize[0] = width;
                rawDataFrame.lineSize[1] = width / 2;
                rawDataFrame.lineSize[2] = height / 2;
                rawDataFrame.lineSize[3] = 0;
                rawDataFrame.rotation = rotation;
                rawDataFrame.video_frame_length = buffer.length;
                if (videoRawDataInterface != null) {
                    videoRawDataInterface.deliverFrame(rawDataFrame, System.nanoTime() / 1000);
                }
                //主动sleep发送一次
                Thread.sleep(25);
              }
            } catch (IOException | InterruptedException ex) {
              ex.printStackTrace();
            } finally {
              try {
                raf.close();
              } catch (IOException e) {
                e.printStackTrace();
              }
            }
          }
        }.start();
    }
```

3. 反注册视频数据回调。

   接口方法如下所示。

```
/**
   * 反注册原始数据回调接口
   * @param streamType 流类型
   */
  public abstract void unRegisterVideoRawDataInterface(AliRtcEngine.AliRawDataStreamType streamT
ype);
```

   示例方法如下所示。

```
mAliRtcEngine.unRegisterVideoRawDataInterface(
            type);
```

输入外部音频流

1. 打开外部音频输入开关。

   接口如下所示。

```
/**
   * @brief 设置是否启用外部音频输入源
   * @param enable YES：开启，NO：关闭
   * @param sampleRate 采样率，16k、48k...
   * @param channels 外部音频源的通道数，可设置为1：单声道，2：双声道
   * @return 返回>=0表示成功,<0表示失败
   */
  public abstract  int setExternalAudioSource(boolean enable ,int sampleRate,int channels);
   /**
   * @brief 设置是否与麦克风采集音频混合
   * @param mixed YES：混音，NO：完全替换麦克风采集数据
   */
   public abstract int setMixedWithMic(boolean mixed);
```

> **说明** 目前仅支持PCM数据。

示例方法如下所示。

```
//获取mAliRtcEngine，
AliRtcEngine  mAliRtcEngine = AliRtcEngine.getInstance(getApplicationContext());
//设置开启外部音频输入源
mAliRtcEngine.setExternalAudioSource(true,44100,1);
//完全替代麦克风采集
mAliRtcEngine.setMixedWithMic(false);
```

2. 输入音频数据。

   接口方法如下所示。

```
/**
   * 输入外部音频数据
   * @param samples 音频数据 不建议超过40ms数据
   * @param samplesLength 采样长度
   * @param timestamp 时间戳
   * @return return>=0 Success, return<0 Failure
   */
  public abstract  int pushExternalAudioFrameRawData(byte[] samples,int samplesLength,long timesta
mp);
```

> **说明** 如果返回值为17236225，代表当前buffer队列塞满，需要等待后再继续输送数据，建议您等待20ms。

示例方法如下所示。

```
private void decodePCMRawData(){
    String pcmPath = "/sdcard/123.pcm";
    if (TextUtils.isEmpty(pcmPath)) {
      ToastUtils.LongToast("请先选择PCM文件！！！");
      return;
    }
    File pcmDataFile = new File(pcmPath);
    if (!pcmDataFile.exists()) {
      ToastUtils.LongToast(pcmPath + " 文件不存在！！！");
      return;
    }
    ToastUtils.LongToast("inputing pcm data");
    new Thread() {
      @Override
      public void run() {
        File pcmDataFile = new File(pcmPath);
        RandomAccessFile raf = null;
        try {
          raf = new RandomAccessFile(pcmDataFile, "r");
        } catch (FileNotFoundException e) {
          e.printStackTrace();
          return;
        }
        try {
          byte[] buffer = new byte[(44100/100)*4*1];
          while (true) {
            int len = raf.read(buffer);
            if (len == -1) {
              raf.seek(0);
            }
            mAliRtcEngine.pushExternalAudioFrameRawData(buffer,buffer.length,0);
            Thread.sleep(20);
          }
        } catch (IOException | InterruptedException ex) {
          ex.printStackTrace();
        } finally {
          try {
            raf.close();
          } catch (IOException e) {
            e.printStackTrace();
          }
        }
      }
    }.start();
  }
```

16.2. iOS

本章节为您介绍了iOS的输入外部视频流和音频流的接口示例。

输入外部视频流

注意 增加自定义视频输入时，建议您提前进行采集，RTC入会之前开始送入数据，避免看到摄像头数据。

1. 打开外部视频输入开关。

   接口说明如下所示。

```
/**
 * @brief 启用外部视频输入源
 * @param enable YES：开启，NO：关闭
 * @param useTexture 是否使用texture模式
 * @param type 流类型
 */
- (int)setExternalVideoSource:(BOOL)enable useTexture:(BOOL)useTexture sourceType:(AliRtcVideoSource)type;
```

   说明 目前userTexture暂不支持，type仅支持AliRtcVideosourceCameraLargeType（视频流）。

   示例方法如下所示。

```
int ret = [self.engine setExternalVideoSource:YES useTexture:NO sourceType:AliRtcVideosourceCameraLargeType];
```

2. 输入视频数据。

   接口说明如下所示。

```
/**
 * @brief 输入视频数据
 * @param frame 帧数据
 * @param type 流类型
 * @return 返回>=0表示成功，<0表示失败
 */
- (int)pushExternalVideoFrame:(AliRtcVideoDataSample *)frame sourceType:(AliRtcVideoSource)type;
```

   参数说明，AliRtcVideoDataSample必填项如下所示。

参数	说明
dataPtr	数据句柄
format	数据类型，目前仅支持AliRtcVideoFormat_I420
width	视频流宽
height	视频流高
datalength	数据buffer length

> 说明 需要APP开启线程进行push；type仅支持AliRtcVideosourceCameraLargeType（视频流）。

示例方法如下所示。

```
float pcmHzFloat;
int yuvDataWidth;
int yuvDataHeight;
NSThread * yuvInputThread;
FILE * yuvInputFile;
- (void)inputYUVRun {
  int width = yuvDataWidth;
  int height = yuvDataHeight;
  int dataSize = width*height*3/2;
  char *yuv_read_data = (char *)malloc(dataSize);
  while (true) {
    if ([yuvInputThread isCancelled]) {
      break;
    }
    size_t read = fread(yuv_read_data,dataSize, 1, yuvInputFile);
    if (read > dataSize) {
      break;
    }
    if (read == 0) {
      fseek(yuvInputFile, 0,SEEK_SET);
      NSLog(@"input yuv reset head!");
      if (yuvInputThread) {
        continue;
      } else {
        break;
      }
    }
    bool push_error = false;
    while (true) {
      if (![yuvInputThread isExecuting]) {
        push_error = YES;
        break;
      }
      AliRtcVideoDataSample *dataSample = [[AliRtcVideoDataSample alloc] init];
      dataSample.dataPtr = (long)yuv_read_data;
      dataSample.format = AliRtcVideoFormat_I420;
      dataSample.width = width;
      dataSample.height = height;
      dataSample.strideY = width;
      dataSample.strideU = width/2;
      dataSample.strideV = width/2;
      dataSample.dataLength = dataSample.strideY * dataSample.height * 3/2;
      //加判断是否开启输入屏幕共享
      int ret = 0;
      if (yuvShareState != YES) {
      } else {
        ret = [self.engine pushExternalVideoFrame:dataSample sourceType:AliRtcVideosourceScreenSh
areType];
```

```
        }
        if (ret == AliRtcErrAudioBufferFull &&
          [yuvInputThread isCancelled] == NO) {
        } else {
          if (ret < 0) {
            push_error = true ;
          }
        }
        //加判断频率, 默认为60hz(30毫秒)
        if (pcmHzFloat > 0) {
          float hz = 1.0/pcmHzFloat;
          [NSThread sleepForTimeInterval:hz]; // 指定频率
        } else {
          [NSThread sleepForTimeInterval:0.03]; // 默认30毫秒 (加延迟操作播放出流畅的画面)
        }
        break;
      }
      if (push_error) {
        break ;
      }
    }
    free(yuv_read_data);
    fclose(yuvInputFile);
    yuvInputFile = NULL ;
}
```

输入外部音频流

1. 打开外部音频输入开关。

   接口如下所示。

```
/**
 * @brief 设置是否启用外部音频输入源
 * @param enable YES 开启，NO 关闭
 * @param sampleRate 采样率 16k, 48k等
 * @param channelsPerFrame 声道数 1, 2
 * @return 返回>=0表示成功, <0表示失败
 */
- (int)setExternalAudioSource:(BOOL)enable withSampleRate:(NSUInteger)sampleRate channelsPerFra
me:(NSUInteger)channelsPerFrame;
/**
 * @brief 设置是否与麦克风采集音频混合
 * @param mixed YES：混音，NO：完全替换麦克风采集数据
 */
- (int)setMixedWithMic:(BOOL)mixed;
```

> **说明** 目前仅支持PCM数据。

   示例方法如下：

```
int ret = [self.engine setExternalAudioSource:YES withSampleRate:pcmSampleRate channelsPerFrame:pcmChannels];
// 完全替代麦克风采集
[self.engine setMixedWithMic:NO];
```

2. 输入音频数据。

   接口方法如下所示。

```
/**
 * @brief 输入音频数据
 * @param data 音频数据 不建议超过40ms数据
 * @param samples 采样率
 * @param timestamp 时间戳
 * @return 返回>=0表示成功,<0表示失败
 * @note 如果返回值为errorCode中的AliRtcErrAudioBufferFull，代表当前buffer队列塞满，需要等待后再继续输送数据，，建议等待20ms
 */
- (int)pushExternalAudioFrameRawData:(void *_Nonnull)data samples:(NSUInteger)samples timestamp:(NSTimeInterval)timestamp;
```

> **说明** 如果返回值为errorCode中的AliRtcErrAudioBufferFull，代表当前buffer队列塞满，需要等待后再继续输送数据，建议您等待20ms。

   示例方法如下所示。

```
int pcmSampleRate;
int pcmChannels;
NSThread * pcmInputThread;
FILE * pcmInputFile;
- (void)inputPCMRun {
  // 40ms
  int readbufSize = (pcmSampleRate/100)*4*sizeof(int16_t)*pcmChannels ;
  while (true) {
    if ([pcmInputThread isCancelled]) {
      break;
    }
    size_t read = fread(pcmData, 1, readbufSize, pcmInputFile) ;
    if (read > readbufSize) {
      break;
    }
    if (read == 0) {
      fseek(pcmInputFile, 0, SEEK_SET) ;
      NSLog(@"input pcm reset head!");
      if (pcmInputThread) {
        continue;
      } else {
        break;
      }
    }
    bool push_error = false;
    while (true) {
      if (![pcmInputThread isExecuting]) {
        push_error = YES;
        break;
      }
      int rc = [self.engine pushExternalAudioFrameRawData:pcmData samples:read timestamp:0];
      if ( rc == AliRtcErrAudioBufferFull && [pcmInputThread isCancelled ] == NO ) {
        [NSThread sleepForTimeInterval:0.08] ;
      } else {
        if (rc < 0) {
          push_error = true ;
        }
        break;
      }
    }
    if (push_error) {
      break;
    }
  }
  fclose(pcmInputFile);
  pcmInputFile = NULL ;
}
```

16.3. Windows

本文为您介绍了Windows的输入外部视频流和音频流的接口示例。

背景信息

Windows支持1.16.2以上版本。

使用场景包括但不限于以下：

- 需要将本地媒体文件（视频/音频）及第三方音视频数据，通过SDK传输到远端播放渲染，可使用音视频外部输入推流实现。
- 需要在使用音频外部输入的同时，本地播放（耳返）输入内容，可使用外部音频输入播放实现。
- 需要将通信过程中视频数据保存或输出处理（外部渲染，修改内容）时，可使用视频数据裸数据输出实现。

输入外部视频数据推流

外部视频输入推流涉及接口和方法如下所示：

```
/**
  * @brief 启用外部视频输入源
  * @param enable YES：开启，NO：关闭
  * @param useTexture 是否使用texture（纹理）模式 取值true|false
  * @param type 流类型
  * @note 启用后使用pushExternalVideoFrame接口输入视频数据
  */
  virtual int setExternalVideoSource(bool enable, bool useTexture, AliRtcVideoSource sourceType) = 0;
/**
  * @brief 输入外部输视频
  * @param frame 帧数据
  * @param type 流类型
  * @param 目前输入视频类型只支持I420
  */
  virtual int pushExternalVideoFrame(AliRtcVideoDataSample *frame, AliRtcVideoSource sourceType) = 0;
```

1. 调用接口setExternalVideoSource启用外部视频输入推流，通过参数enable设置开启，通过参数sourceType指定要推流的track类型（摄像头流/屏幕流）。

   > **说明** 目前Windows端不支持直接输入纹理，所以参数useTexture始终传入*false*。

2. 应用侧通过接口configLocalCameraPublish或configLocalScreenPublish配置指定track推流，然后调用接口publish开始推视频流。

   > **说明** SDK允许先推流然后开启外部视频输入，即步骤1和步骤2时序对换，但这种情况下，默认开始推流时，先推送出的是原始采集源（摄像头或屏幕捕获）的视频数据，直到启用外部输入。

3. 应用侧持续调用接口pushExternalVideoFrame，向SDK投递视频裸数据，进行推流，参数frame传入裸数据相关信息，参数sourceType指明推流track类型，与步骤1中设置开启的track类型保持一致。

> 说明
>
> - 投递视频帧数据的频率由应用方控制，依据视频源帧率保持间隔投递，直至输入停止。建议应用侧独立开启线程，进行数据投递，保证数据输入及时性。
> - 目前Windows端支持输入YUV数据（格式I420），需要在参数frame的裸数据信息中，指定format为*AliRtcVideoFormatI420*， bufferType为*AliRtcBufferTypeRawData*。
> - 赋值裸数据信息时，需要数据指针、视频宽高信息、stride信息等完整传入；旋转角度rotation目前暂未支持设置，请保持默认值0；时间戳取当期时间，单位为毫秒。

4. 数据源输入结束或应用中止外部视频输入，同样调用接口setExternalVideoSource关闭外部视频输入。

代码示例：

```
AliRtcEngine *pEngine = AliRtcEngine::sharedInstance(this, "");
.....
//1.启用外部视频输入
pEngine->setExternalVideoSource(true, false, AliRtcVideoSourceCamera); //camera track启用外部输入
bPushExternalVideo = true;
//2.配置开始推流
pEngine->configLocalCameraPublish(true);
pEngine->publish();
.....
//3.独立线程推送外部视频数据
int frameRate = 30;  // 30 fps
do
{
  size_t frameLength = videoWidth * videoHeight * 3 / 2;
  void* cacheBuf = (void*)malloc(frameLength);
  /*
  从外部数据源拷贝推送视频数据到cacheBuf
  */
  AliRtcVideoDataSample sample;
  sample.data = (unsigned char*)cacheBuf;
  sample.format = AliRtcVideoFormatI420;
  sample.width = videoWidth;
  sample.height = videoHeight;
  sample.strideY = videoWidth;
  sample.strideU = videoWidth / 2;
  sample.strideV = videoWidth / 2;
  sample.dataLen = frameLength;
  sample.rotation = 0;
  pEngine->pushExternalVideoFrame(&sample, AliRtcVideoSourceCamera);
  //控制帧数据投递频率
  Sleep(1000 / frameRate);
} while (bPushExternalVideo);
.....
//4.停止外部视频输入
pEngine->setExternalVideoSource(false, false, AliRtcVideoSourceCamera);
bPushExternalVideo = false;
```

视频裸数据输出

视频裸数据输出涉及接口和方法如下所示：

```
/**
  * @brief 订阅视频数据输出
  * @note 输出数据将通过onCaptureVideoSample及onRemoteVideoSample回调返回
  */
  virtual void r   egisterVideoSampleObserver() = 0;
  /**
  * @brief 取消订阅视频数据输出
  */
  virtual void unRegisterVideoSampleObserver() = 0;
  /**
  * @brief 本地采集视频数据回调
  * @param videoSource 视频数据类型
  * @param videoSample 视频数据
  */
  virtual void onCaptureVideoSample(AliRtcVideoSource videoSource, AliRtcVideoDataSample *videoSampl
e) {};
  /**
  * @brief 远端视频数据回调
  * @param uid user id
  * @param videoSource video source
  * @param videoSample video sample
  */
  virtual void onRemoteVideoSample(const AliRtc::String &uid, AliRtcVideoSource videoSource, AliRtcVideo
DataSample *videoSample) {};
```

1. 应用需先继承AliRtcEventListener接口，实现onCaptureVideoSample和onRemoteVideoSample回调，用于接收本地采集视频裸数据，以及订阅到的远端视频裸数据。

   > **说明** 目前Windows端只支持输出YUV（I420）格式数据，可通过SDK提供的AliConvertVideoData静态接口，转换到RGBA格式。

2. 调用接口registerVideoSampleObserver接口启动视频裸数据输出，启动后视频数据将通过onCaptureVideoSample和onRemoteVideoSample持续回调。

   > **说明** 本地采集视频数据需要在开启摄像头（打开预览或正在推送视频流）前提下才会有回调输出，远端视频数据同样需要在订阅其他用户的视频流成功后才会有回调输出。

3. 需要停止接收视频裸数据时，调用接口unRegisterVideoSampleObserver关闭视频裸数据输出即可。

代码示例：

```
AliRtcEngine *pEngine = AliRtcEngine::sharedInstance(this, "");
.....
//1. 注册视频裸数据输出
pEngine->registerVideoSampleObserver();
//开始预览/推流及订阅其他用户视频
.....
//2. 接收裸数据回调
void onCaptureVideoSample(AliRtcVideoSource videoSource, AliRtcVideoDataSample *videoSample)
{
  //处理本地采集数据回调
}
void onRemoteVideoSample(const AliRtc::String &uid, AliRtcVideoSource videoSource, AliRtcVideoDataSam
ple *videoSample)
{
  //处理远端数据回调
}
//3.停止视频裸数据输出
pEngine->unRegisterVideoSampleObserver();
```

输入外部音频数据推流

外部音频输入推流涉及接口和方法如下所示：

```
  /**
  * @brief 设置是否启用外部音频输入源
  * @param enable YES 开启，NO 关闭
  * @param sampleRate 采样率 16k 48k...
  * @param channelsPerFrame 采样率 16k 48k...
  * @return >=0表示成功， <0表示失败
  */
  virtual int setExternalAudioSource(bool enable, unsigned int sampleRate, unsigned int channelsPerFrame
) = 0;
  /**
  * @brief 输入音频数据
  * @param audioSamples 音频数据
  * @param sampleLength 音频数据长度
  * @param timestamp 时间戳
  * @return <0表示失败，返回值为ERR_AUDIO_BUFFER_FULL时，需要在间隔投递数据时间长度后再次重试投递
  */
  virtual int pushExternalAudioFrameRawData(const void* audioSamples, unsigned int sampleLength, long
long timestamp) = 0;
  /**
  * @brief 设置外部输入是否与麦克风采集音频混合
  * @param mixed YES 混音，NO 完全替换麦克风采集数据
  */
  virtual int setMixedWithMic(bool mixed) = 0;
  /**
  * @brief 设置外部输入音频混音音量
  * @param vol 音量 0-100
  */
  virtual int setExternalAudioVolume(int volume) = 0;
  /**
  * @brief 获取外部输入音频混音音量
  * @return vol 音量
  */
  virtual int getExternalAudioVolume() = 0;
```

1. 调用接口setExternalAudioSource启用外部音频输入推流，通过参数enable设置开启，通过参数sampleRate和参数channelsPerFrame指定要输入音频数据的采样率和声道数。

   > **说明** 目前仅支持输入音频PCM数据，数据编码为Signed 16-bit。

2. 应用侧通过接口configLocalAudioPublish配置音频推流，然后调用publish接口开始推音频流。

   > **说明** SDK允许先推流在开启外部音频输入，即步骤1和步骤2时序对换，但这种情况下，默认开始推流时，先推送出的是麦克风采集音频，直到启用外部输入。

3. 应用侧持续调用pushExternalAudioFrameRawData接口，向SDK投递音频PCM数据，参数audioSamples带入音频数据地址，参数sampleLength指明音频长度，参数timestamp为当前时间。

> 说明
>
> - 投递音频裸数据的频率由应用方控制，每次投递数据量不要超过240ms的音频数据量，建议每次投递20ms的音频数据，保持循环投递直到结束。当输入数据频率过快，SDK缓存已满暂时无法消费数据时，接口会返回错误码*ERR_AUDIO_BUFFER_FULL*，此时应用侧需要等待一个数据时间长度后，再次重试投递此数据，直到成功，否则将丢失输入音频数据。
> - 与视频输入一致，同样建议应用侧单独开启线程，投递音频裸数据，直到输入停止。
> - 由于外部输入音频数据的同时，可能同时还有麦克风在采集推流，应用可设置是否需要将外部输入音频与麦克风采集音频混音后一起推出，或单独只推送外部输入音频，通过调用接口setMixedWithMic可开启或关闭与麦克风采集音频的混音，同时可通过接口setExternalAudioVolume和getExternalAudioVolume设置和获取输入音频的混音音量，音量可调整范围为[0 - 100]，默认50。
> - 当应用同时推送麦克风采集音频与外部输入音频，接口muteLocalMic的默认行为是停止所有音频的推送（包括麦克风与外部输入），如果应用只需要停止麦克风音频（保持外部输入音频推送），可通过设置muteLocalMic接口中的第二个参数 mode，选择*AliRtcMuteOnlyMicAudioMode*模式即可。

4. 数据源输入结束或应用中止外部音频输入，同样调用接口setExternalAudioSource关闭外部视音频输入即可。

代码示例：

```
AliRtcEngine *pEngine = AliRtcEngine::sharedInstance(this, "");
.....
//1. 启用外部音频输入播放
unsigned int sampleRate = 44100; //44.1k
unsigned int channelsPerFrame = 1; //单声道
pEngine->setExteranlAudioRender(true, sampleRate, channelsPerFrame); //启用音频输入播放
bRenderExternalAudio = true;
//2.独立线程推送外部音频数据
unsigned int audioDataInterval = 20; //20 ms
size_t bytePerSample = 16 / 8; // Signed 16-bit
size_t dataSize = sampleRate * channelsPerFrame * bytePerSample / (1000 / audioDataInterval);
unsigned int* data = (unsigned int*)malloc(dataSize);
do
{
  /*
  从外部数据源拷贝推送音频数据到data
  */
  std::chrono::milliseconds now = std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::syste
m_clock::now().time_since_epoch());
  int ret = mpEngine->pushExternalAudioRenderRawData(data, dataSize, sampleRate, channelsPerFrame, n
ow.count());
  if (ret < 0)
  {
    //发生错误，中断播放
    break;
  }
  //返回结果不为0，需要进行错误判断，检查是否缓冲区满
  //返回结果如果为0，投递成功，继续读取并投递数据，无需间隔
  while (ret == ERR_AUDIO_BUFFER_FULL && bRenderExternalAudio)
  {
    Sleep(audioDataInterval); //缓冲区满，间隔一个数据长度后再次重试，直到成功
    if (!bRenderExternalAudio) break;
    ret = mpEngine->pushExternalAudioRenderRawData(data, read_size, sampleRate, channelsPerFrame, no
w.count());
    if (ret < 0)
    {
      //发生错误，中断推送
      break;
    }
  }
} while (bRenderExternalAudio);
.....
//4.停止外部音频输入播放
mpEngine->setExteranlAudioRender(false, 0, 0);
bRenderExternalAudio = false;
```

外部音频输入播放

外部音频输入涉及接口和方法如下所示：

```
/**
  * @brief 设置是否启用外部输入音频播放
  * @param enable YES 开启，NO 关闭
  * @param sampleRate 采样率 16k 48k...
  * @param channelsPerFrame 采样率 16k 48k...
  * @return >=0表示成功， <0表示失败
  */
  virtual int setExteranlAudioRender(bool enable, unsigned int sampleRate, unsigned int channelsPerFrame
) = 0;
  /**
  * @brief 输入音频播放数据
  * @param audioSamples 音频数据
  * @param sampleLength 音频数据长度
  * @param sampleRate 音频采样率
  * @param channelsPerFrame 音频声道数
  * @param timestamp 时间戳
  * @return <0表示失败
  */
  virtual int pushExternalAudioRenderRawData(const void* audioSamples, unsigned int sampleLength, unsi
gned int sampleRate, unsigned int channelsPerFrame, long long timestamp) = 0;
  /**
  * @brief 设置外部音频播放音量
  * @param vol 音量 0-100
  */
  virtual int setExternalAudioRenderVolume(int volume) = 0;
  /**
  * @brief 获取音频播放音量
  * @return vol 音量
  */
  virtual int getExternalAudioRenderVolume() = 0;
```

1. 调用接口setExteranlAudioRender启用外部音频输入播放，通过参数enable设置开启，通过参数sampleRate和参数channelsPerFrame指定要输入音频数据的采样率和声道数。

   > **说明** 目前仅支持输入音频PCM数据，数据编码为Signed 16-bit，输入播放音频的声道数与采样率，可以在推流过程中动态变更，下一步骤2中，投递接口pushExternalAudioRenderRawData中可指定当次音频数据的采样率和声道数。

2. 应用侧持续调用pushExternalAudioRenderRawData接口，向SDK投递音频PCM数据播放，参数audioSamples带入音频数据地址，参数sampleLength指明音频长度，参数sampleRate和参数channelsPerFrame指定要输入音频数据的采样率和声道数，参数timestamp为当前时间戳。

   > **说明**
   >
   > - 与外部输入音频数据推送相同，播放音频也需要由应用侧保持频率投递，建议应用侧独立开启线程，进行数据投递，保证数据输入及时性。
   > - 外部输入音频播放过程中，可通过setExternalAudioRenderVolume接口和getExternalAudioRenderVolume接口调整设置播放音频音量，音量可调整范围为 [0 - 100]，默认50。

3. 数据源输入结束或应用中止音频播放时，调用接口setExteranlAudioRender关闭外部音频播放即可。

代码示例：

```
AliRtcEngine *pEngine = AliRtcEngine::sharedInstance(this, "");
.....
//1. 启用外部音频输入播放
unsigned int sampleRate = 44100; //44.1k
unsigned int channelsPerFrame = 1; //单声道
pEngine->setExteranlAudioRender(true, sampleRate, channelsPerFrame); //启用音频输入播放
bRenderExternalAudio = true;
//2.独立线程推送外部音频数据
unsigned int audioDataInterval = 20; //20 ms
size_t bytePerSample = 16 / 8; // Signed 16-bit
size_t dataSize = sampleRate * channelsPerFrame * bytePerSample / (1000 / audioDataInterval);
unsigned int* data = (unsigned int*)malloc(dataSize);
do
{
  /*
  从外部数据源拷贝推送音频数据到data
  */
  std::chrono::milliseconds now = std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::syste
m_clock::now().time_since_epoch());
  int ret = mpEngine->pushExternalAudioRenderRawData(data, dataSize, sampleRate, channelsPerFrame, n
ow.count());
  if (ret < 0)
  {
    //发生错误，中断播放
    break;
  }
  //返回结果不为0，需要进行错误判断，检查是否缓冲区满
  //返回结果如果为0，投递成功，继续读取并投递数据，无需间隔
  while (ret == ERR_AUDIO_BUFFER_FULL && bRenderExternalAudio)
  {
    Sleep(audioDataInterval); //缓冲区满，间隔一个数据长度后再次重试，直到成功
    if (!bRenderExternalAudio) break;
    ret = mpEngine->pushExternalAudioRenderRawData(data, read_size, sampleRate, channelsPerFrame, no
w.count());
    if (ret < 0)
    {
      //发生错误，中断推送
      break;
    }
  }
} while (bRenderExternalAudio);
.....
//4.停止外部音频输入播放
mpEngine->setExteranlAudioRender(false, 0, 0);
bRenderExternalAudio = false;
```

17.阿里云RACE美颜

17.1. RACE美颜

通过阅读本文，您可以了解RACE美颜功能，并快速完成产品配置操作。

产品介绍

RACE是渲染计算平台（Render And Compute Everything Engine）的简写。RACE美颜包含图像及视频渲染能力，如滤镜、特效、贴纸，以及美颜美型、人脸检测等算法能力，覆盖多种拍摄、剪辑场景，满足客户丰富的产品需求。

快速集成

- Android端，支持maven与aar离线集成两种方式：
  - maven依赖集成方式：*maven { url "https://maven.aliyun.com/nexus/content/repositories/releases" } implementation 'com.aliyun.race:AliyunRace:1.1.1'*。
  - 本地集成方式：获取AliyunRace.aar文件并添加到工程中。
- iOS端，支持pod与framework离线集成两种方式：
  - pod依赖集成方式：pod 'AliyunRace', '1.1.1'。
  - 本地集成方式：将AliyunRace.framework、Face3D.framwork、opencv2.framework添加到工程中。

接入指南

- RACE美颜Android端的集成方式、接口说明和实现步骤请参照RACE美颜Android端接入指南。
- RACE美颜iOS端的集成方式、接口说明和实现步骤请参照RACE美颜Android端接入指南。

美颜美型功能

- 功能介绍。

  高级美颜包含美白、磨皮、锐化等功能，美型默认包含优雅、精致、网红、可爱、婴儿五种脸型。该功能提供包括眼睛、鼻子、嘴形等22种预置形状，便于客户根据产品需求扩展丰富的功能。
- 数据结构及接口介绍。

- 美型形状类型。

```
/**
 * 美型形状类型
 */
typedef enum ALRFaceShape
{
  ALR_FACE_TYPE_CUT_CHEEK    = 0, //颧骨   [0,1]
  ALR_FACE_TYPE_CUT_FACE     = 1, //削脸   [0,1]
  ALR_FACE_TYPE_THIN_FACE    = 2, //瘦脸   [0,1]
  ALR_FACE_TYPE_LONG_FACE    = 3, //脸长   [0,1]
  ALR_FACE_TYPE_LOWER_JAW    = 4, //下巴缩短 [-1,1]
  ALR_FACE_TYPE_HIGHER_JAW   = 5, //下巴拉长 [-1,1]
  ALR_FACE_TYPE_THIN_JAW     = 6, //瘦下巴  [-1,1]
  ALR_FACE_TYPE_THIN_MANDIBLE  = 7, //瘦下颌  [0,1]
  ALR_FACE_TYPE_BIG_EYE      = 8, //大眼   [0,1]
  ALR_FACE_TYPE_EYE_ANGLE1   = 9, //眼角1   [0,1]
  ALR_FACE_TYPE_CANTHUS      = 10,//眼距   [-1,1]
  ALR_FACE_TYPE_CANTHUS1     = 11,//拉宽眼距 [-1,1]
  ALR_FACE_TYPE_EYE_ANGLE2   = 12,//眼角2   [-1,1]
  ALR_FACE_TYPE_EYE_TDANGLE  = 13,//眼睛高度 [-1,1]
  ALR_FACE_TYPE_THIN_NOSE    = 14,//瘦鼻   [0,1]
  ALR_FACE_TYPE_NOSE_WING    = 15,//鼻翼   [0,1]
  ALR_FACE_TYPE_NASAL_HEIGHT  = 16,//鼻长   [-1,1]
  ALR_FACE_TYPE_NOSE_TIP_HEIGHT = 17,//鼻头长  [-1,1]
  ALR_FACE_TYPE_MOUTH_WIDTH   = 18,//唇宽   [-1,1]
  ALR_FACE_TYPE_MOUTH_SIZE    = 19,//嘴唇大小 [-1,1]
  ALR_FACE_TYPE_MOUTH_HIGH    = 20,//唇高   [-1,1]
  ALR_FACE_TYPE_PHILTRUM     = 21, //人中
  ALR_FACE_TYPE_MAX        = 22
} ALRFaceShape;
```

> **说明** 对于每一种形状，可设置参数值超出上述定义的[-1, 1]的范围综合调试效果。

- 创建高级美颜实例。

```
/**
 * 创建美颜美型实例
 * @param handle 美颜美型句柄指针
 * @param resDir 资源文件目录绝对路径
 * @param rid 数据上报标识符
 * @param lid 日志上报标识符
 * @return 成功返回 ALR_OK，否则返回 < 0
 */
int aliyun_beautify_create(race_t *handle, const char *resDir, int64_t rid = -1, int64_t lid = -1);
```

- 销毁高级美颜实例。

```
/**
 * 销毁美颜美型实例
 * @param handle 美颜美型句柄
 */
void aliyun_beautify_destroy(race_t handle);
```

- 高级美颜调试开关。

```
/**
 * 美颜美型调试开关
 * @param handle 美颜美型句柄
 * @param enable 调试开启标志
 * @return 成功返回 ALR_OK，否则返回 < 0
 */
int aliyun_beautify_setFaceDebug(race_t handle, bool enable);
```

- 美型开关。

```
/**
 * 美型开关
 * @param handle 美颜美型句柄
 * @param switchOn 是否开启人脸检测及美型处理
 */
void aliyun_beautify_setFaceSwitch(race_t handle, bool switchOn);
```

- 设置美型参数。

```
/**
 * 设置美型参数
 * @param handle 美颜美型句柄
 * @param level 美型等级参数，可以超出 ALRFaceShape 中定义的参数范围
 * @return 成功返回 ALR_OK，否则返回 < 0
 */
int aliyun_beautify_setFaceShape(race_t handle, ALRFaceShape type, float level);
```

- 高级美颜渲染。

```
/**
 * 高级美颜，输入是类型 CMSampleBufferRef 的图像数据，返回与输入同样大小的纹理 id
 * @param handle 创建成功的句柄
 * @param sampleBuffer 图像数据 CMSampleBufferRef
 * @return 成功则返回渲染纹理 id，失败返回 0
 */
int aliyun_beautify_processSampleBuffer(race_t handle, void *sampleBuffer);
/**
 * 单输入美颜美型渲染
 * @param handle 美颜美型句柄
 * @param buffer CPU图像内存地址
 * @param bufferSize CPU图像内存字节数
 * @param format 图像像素格式
 * @param width 图像宽度
 * @param height 图像高度
 * @param bytesPerRow CPU图像内存一行的字节数
 * @param rotation 图像旋转角度（顺进针方向）
 * @param range 颜色色域范围
 * @param standard 色彩标准
 * @param flags 详见 ALR_FLAG_XXX 定义，如 OES 纹理、镜像、输出图像旋转等
 * @return 成功则返回输出纹理句柄，否则返回 < 0
 */
int aliyun_beautify_processBufferToTexture(race_t handle,
                                           uint8_t *buffer,
                                           uint32_t bufferSize
```

```
                        uint32_t bufferSize,
                        aliyun_image_format_t format,
                        uint32_t width,
                        uint32_t height,
                        uint32_t bytesPerRow,
                        aliyun_rotation_t rotation,
                        aliyun_color_range_t range,
                        aliyun_color_standard_t standard,
                        uint8_t flags);
/**
 * 美颜美型纹理输入渲染
 * @param handle 美颜美型句柄
 * @param textureIn 输入图像纹理
 * @param width 输入图像纹理宽度
 * @param height 输入图像纹理高度
 * @param rotation 输入图像纹理旋转角度（顺时针）
 * @param flags 详见 ALR_FLAG_XXX 定义，如 OES 纹理、镜像、输出图像旋转等
 * @return 成功则返回输出纹理句柄，否则返回 < 0
 */
int aliyun_beautify_processTextureToTexture(race_t handle,
                        uint32_t textureIn,
                        uint32_t width,
                        uint32_t height,
                        aliyun_rotation_t rotation,
                        uint8_t flags);
/**
 * 双输入美颜美型渲染
 * @param handle 美颜美型句柄
 * @param textureIn GPU纹理 id
 * @param buffer CPU图像内存地址
 * @param bufferSize CPU图像内存字节数
 * @param format 图像像素格式
 * @param width 图像宽度
 * @param height 图像高度
 * @param bytesPerRow CPU图像内存一行的字节数
 * @param rotation 图像旋转角度（顺进针方向）
 * @param flags 详见 ALR_FLAG_XXX 定义，如 OES 纹理、镜像、输出图像旋转等
 * @return 成功则返回输出纹理句柄，否则返回 < 0
 */
int aliyun_beautify_processDualInputToTexture(race_t handle,
                        uint32_t textureIn,
                        uint8_t *buffer,
                        uint32_t bufferSize,
                        aliyun_image_format_t format,
                        uint32_t width,
                        uint32_t height,
                        uint32_t bytesPerRow,
                        aliyun_rotation_t rotation,
                        uint8_t flags);
```

- 接口集成示例。

```
#include "aliyun_beautify.h"
race_t beautify = nullptr;
// 创建高级美颜实例
// iOS: RACE内部自动从Framework读取，可传空字串。
// Android: 外部需将asserts/race_res目录拷贝到sdcard或者可访问的路径，并将此路径作为参数传入如/sdcard
aliyun_beautify_create(&beautify, "");
// 设置美颜参数
aliyun_beautify_setSkinBuffing(beautify, skinBuffingValue); //磨皮
aliyun_beautify_setSkinWhitening(beautify, skinWhiteningValue); //美白
aliyun_beautify_setSharpen(beautify, sharpenValue); //锐化
// 设置美型参数
aliyun_beautify_setFaceShape(beautify, ALR_FACE_TYPE_BIG_EYE, bigEyeValue);//大眼
aliyun_beautify_setFaceShape(beautify, ALR_FACE_TYPE_LONG_FACE, longFaceValue);//脸长
aliyun_beautify_setFaceShape(beautify, ALR_FACE_TYPE_CUT_FACE, cutFaceValue);//削脸
aliyun_beautify_setFaceShape(beautify, ALR_FACE_TYPE_THIN_FACE, thinFaceValue);//瘦脸
aliyun_beautify_setFaceShape(beautify, ALR_FACE_TYPE_LOWER_JAW, lowerJawValue);//下巴
aliyun_beautify_setFaceShape(beautify, ALR_FACE_TYPE_MOUTH_WIDTH, mouthWidthValue);//唇宽
aliyun_beautify_setFaceShape(beautify, ALR_FACE_TYPE_THIN_NOSE, thinNoseValue);//瘦鼻
aliyun_beautify_setFaceShape(beautify, ALR_FACE_TYPE_THIN_MANDIBLE, thinMandibleValue);//下颌
aliyun_beautify_setFaceShape(beautify, ALR_FACE_TYPE_CUT_CHEEK, cutCheekValue);//颧骨
// iOS: 美颜美型效果处理，返回渲染后的结果纹理id。
textureOut = aliyun_beautify_processSampleBuffer(beautify, sampleBuffer);
// Android & iOS: 单纹理输入
textureOut = aliyun_beautify_processTextureToTexture(beautify,
                                textureIn,
                                width, height,
                                rotation, flags);
// Android & iOS: CPU内存+GPU纹理双输入
textureOut = aliyun_beautify_processDualInputToTexture(beautify,
                                 textureIn,
                                 buffer, bufferSize,
                                 format, width, height,
                                 bytePerRow,
                                 rotation, flags);
// 销毁美颜美型实例
aliyun_beautify_destroy(beautify);
```

人脸检测

- 功能介绍。

  在静态的照片、图像或动态的视频流中，使用人脸检测技术可以快速准确地辨别出单张或多张人脸，并检测出人脸所在的位置。

- 数据结构及接口介绍。

```
// 宏定义
#define ALR_FACE_DETECT_MODE_VIDEO       0x10000000 // video
#define ALR_FACE_DETECT_MODE_IMAGE       0x20000000 // image
/// 网络模型
#define ALR_FACE_DETECT_NETWORK_HBN      0x00000001 // HBN
#define ALR_FACE_DETECT_NETWORK_FASTERRCNN  0x00000002 // Faster RCNN
// 人脸区域
typedef struct aliyun_rect_t
{
  int left;   //left of face rectangle
  int top;    //top of face rectangle
  int right;  //right of face rectangle
  int bottom; //bottom of face rectangle
} aliyun_rect_t;
// 点的数据结构
typedef struct aliyun_point_t
{
  float x;
  float y;
} aliyun_point_t;
// 一张人脸的数据结构
typedef struct aliyun_face_t
{
  aliyun_rect_t rect;            // 人脸区域
  float score;                   // 置信度
  aliyun_point_t landmarks_array[106]; // 106关键点
  float landmarks_visible_array[106]; // 106关键点遮挡情况
  float yaw;                     // 水平转角，真实度量的左负右正
  float pitch;                   // 俯仰角，真实度量的上负下正
  float roll;                    // 旋转角，真实度量的左负右正
  float eye_distance;            // 两眼间距
  int faceID;
} aliyun_face_t;
// 多张人脸的数据结构
typedef struct aliyun_face_info_t
{
  aliyun_face_t *p_faces;          //face info
  int face_count;                  //face detection num
} aliyun_face_info_t;
// 人脸检测参数类型
typedef enum
{
  ALR_FACE_PARAM_DETECT_INTERVAL     = 1, // 人脸检测的帧率（默认值30，即每隔30帧检测一次）
  ALR_FACE_PARAM_SMOOTH_THRESHOLD    = 2, // 人脸关键点平滑系数（默认值0.25）
  ALR_FACE_PARAM_POSE_SMOOTH_THRESHOLD = 4, // 姿态平衡系数(0,1]，越大平滑程度越大
  ALR_FACE_PARAM_DETECT_THRESHOLD    = 5, // 人脸检测阈值(0,1)，阈值越大，误检越少，但漏检测会增
加, default 0.95 for faster rcnn; default 0.3 for SSD
  ALR_FACE_PARAM_ALIGNMENT_INTERVAL  = 11, // 人脸检测对齐间隔，默认1，一般不要超过5
  ALR_FACE_PARAM_MAX_FACE_SUPPORT    = 12, // 最多支持检出的人脸个数，最大设为32, 主要针对faster rc
nn
  ALR_FACE_PARAM_DETECT_IMG_SIZE     = 13, // 人脸检测输入的图像大小，default： 240 for faster rcnn, r
ecommend set 320 for tiny face detection
} aliyun_face_param_type_t;
```

- 创建人脸检测实例。

  Android端C接口：

```
/**
 * 创建人脸检测句柄
 * @param handle
 * @param det_paraPath 人脸检测模型的路径
 * @param pts_paraPath 关键点检测模型的路径
 * @param config
 * @return == 0 OK; < 0 error
 */
RACE_EXTERN int aliyun_face_create(race_t* handle,
                    JNIEnv* env,
                    const char* det_paraPath,
                    const char* pts_paraPath,
                    unsigned int config);
```

  iOS端C接口：

```
/**
 * 创建人脸检测句柄
 * iOS端推荐使用接口
 * @param handle with initialed
 * @param config 检测图片：ALR_FACE_DETECT_MODE_IMAGE，检测视频：ALR_FACE_DETECT_MODE_V
IDEO
 * @return == 0 OK; < 0 error
 */
RACE_EXTERN int aliyun_face_default_create(race_t* handle, unsigned int config);
/**
 * 创建人脸检测句柄
 * @param handle with initialed
 * @param det_paraPath 人脸检测模型的绝对路径
 * @param pts_paraPath 关键点检测模型的绝对路径
 * @param config 检测图片：ALR_FACE_DETECT_MODE_IMAGE，检测视频：ALR_FACE_DETECT_MODE_V
IDEO，人脸检测模型选择模型1，则增加ALR_FACE_DETECT_NETWORK_HBN，选择模型3，则增加ALR_FACE
_DETECT_NETWORK_FASTERRCNN
 * @return == 0 OK; < 0 error
 */
RACE_EXTERN int aliyun_face_create(race_t* handle,
            const char* det_paraPath,
            const char* pts_paraPath,
            unsigned int config);
```

- 设置人脸检测参数。

```
/**
 * 设置人脸检测参数
 * @param handle with initialed
 * @param type face_param_type
 * @param value new threshold
 * @return = 0 OK; < 0 error
 */
RACE_EXTERN int aliyun_face_setParam(race_t handle, aliyun_face_param_type_t type, float value);
```

- 调用人脸检测接口。

```
/**
 * 人脸检测
 * @param handle with initialed
 * @param buffer input image
 * @param format support type BGR、RGBA、RGB、Y(GRAY)，推荐使用RGBA
 * @param width  width
 * @param height height
 * @param bytesPerRow 用于检测的图像的跨度(以像素为单位)，即每行的字节数
 * @param rotation rotate image to frontalization for face detection
 * @param config MOBILE_FACE_DETECT, or MOBILE_FACE_DETECT|MOBILE_EYE_BLINK et.al 默认值0
 * @param outRotation result process rotate specific angle first, angle =  0/90/180/270
 * @param outFlipAxis  flip x/y 0(no flip)/1(flip X axis)/2(flip Y axis)
 * @param faceInfo store face detetion result
 * @return = 0 OK; < 0 error
 */
RACE_EXTERN int aliyun_face_detect(race_t handle,
                        uint8_t *buffer,
                        aliyun_image_format_t format,
                        uint32_t width,
                        uint32_t height,
                        uint32_t bytesPerRow,
                        aliyun_rotation_t rotation,
                        uint32_t config,
                        aliyun_rotation_t outRotation,
                        uint32_t outFlipAxis,
                        aliyun_face_info_t* faceInfo);
```

- 销毁人脸检测实例。

```
/**
 * 销毁人脸检测句柄
 * @param handle  with initialed
 */
RACE_EXTERN void aliyun_face_destroy(race_t handle);
```

- 接口集成示例。

```
race_t handle = nullptr;
// for Android
aliyun_face_create(handle,
            "/sdcard/race_res/models/0_3/fd_00002_3",
            "/sdcard/race_res/models/0_3/fd_00002_12",
            ALR_FACE_DETECT_NETWORK_FASTERRCNN | ALR_FACE_DETECT_MODE_VIDEO);
// for iOS
aliyun_face_default_create(handle, ALR_FACE_DETECT_MODE_VIDEO);
// 设置检测间隔，推荐设置为5
aliyun_face_setParam(handle, ALR_FACE_PARAM_DETECT_INTERVAL, 5);
aliyun_face_info_t info;
aliyun_face_detect(handle,
            buffer,
            ALR_IMAGE_FORMAT_NV21,
            1280,
            720,
            1280,
            ALR_ROTATE_90_CW,
            0,
            ALR_ROTATE_90_CW,
            0,
            &info);
aliyun_face_destroy(handle);
```

17.2. Android

通过阅读本文，您可以了解阿里云RTC SDK对接RACE美颜的集成方式、接口说明和实现步骤。

集成RACE SDK

支持maven与aar离线集成两种方式。

- maven集成方式：
  i. 在项目的project目录的gradle添加以下maven仓库。

```
allprojects {
  repositories {
    google()
    jcenter()
    maven { url "https://maven.aliyun.com/nexus/content/repositories/releases" }
  }
}
```

  ii. 在App的gragle下添加RACE依赖。

```
implementation "com.aliyun.race:AliyunRace:1.1.0"
```

- 本地集成方式：

  获取AliyunRace.aar文件并添加到工程的libs目录中。

RTC SDK对接RACE接口说明

以下为RTC SDK对接RACE时需要使用的接口说明，RACE美颜SDK中的其他接口不在本文说明。

API	描述
AliyunBeautifyNative	创建美颜美型实例
initialize	初始化
setFaceDebug	美颜美型调试开关
setLogLevel	设置输出日志等级
setSkinBuffing	设置磨皮参数
setSharpen	设置锐化参数
setSkinWhitening	设置美白参数
setFaceSwitch	美型开关
setFaceShape	设置美型参数
processTexture	美颜美型纹理输入渲染
destroy	销毁美颜美型实例

- AliyunBeautifyNative：创建美颜美型实例。

```
AliyunBeautifyNative(Context context);
```

参数	类型	描述
context	Context	上下文

- initialize：初始化美型实例。

```
int initialize();
```

返回值：成功返回0，失败返回<0。

- setFaceDebug：美颜美型调试开关。

```
void setFaceDebug(boolean enable);
```

参数	类型	描述
enable	boolean	美颜美型Debug开关

- setLogLevel：设置输出日志等级。

```
void setLogLevel(int level);
```

参数	类型	描述
level	int	日志等级

- setSkinBuffing：设置磨皮等级。

```
int setSkinBuffing(float level);
```

参数	类型	描述
level	float	磨皮等级，取值：0~1

返回值：成功返回0，失败返回<0。

- setSharpen：设置锐化等级。

```
int setSharpen(float level);
```

参数	类型	描述
level	float	锐化等级，取值：0~1

返回值：成功返回0，失败返回<0。

- setSkinWhitening：设置美白等级。

```
int setSkinWhitening(float level);
```

参数	类型	描述
level	float	美白等级，取值：0~1

返回值：成功返回0，失败返回<0。

- setFaceSwitch：美型开关。

```
void setFaceSwitch(boolean enable);
```

参数	类型	描述
enable	boolean	是否开启人脸检测及美型处理，取值：true|false

- setFaceShape：设置美型等级。

```
int setFaceShape(int type, float level);
```

参数	类型	描述
type	int	美型类型
level	float	美型等级

返回值：成功返回0，失败返回<0。

type定义和建议范围如下：

```
public static final int ALR_FACE_SHAPE_TYPE_CUT_CHEEK    = 0; //颧骨，取值：0~1
public static final int ALR_FACE_SHAPE_TYPE_CUT_FACE     = 1; //削脸，取值：0~1
public static final int ALR_FACE_SHAPE_TYPE_THIN_FACE    = 2; //瘦脸，取值：0~1
public static final int ALR_FACE_SHAPE_TYPE_LONG_FACE    = 3; //脸长，取值：0~1
public static final int ALR_FACE_SHAPE_TYPE_LOWER_JAW    = 4; //下巴缩短，取值：-1~1
public static final int ALR_FACE_SHAPE_TYPE_HIGHER_JAW   = 5; //下巴拉长，取值：-1~1
public static final int ALR_FACE_TYPE_THIN_JAW         = 6; //瘦下巴，取值：-1~1
public static final int ALR_FACE_TYPE_THIN_MANDIBLE     = 7; //瘦下颌，取值：0~1
public static final int ALR_FACE_SHAPE_TYPE_BIG_EYE     = 8; //大眼，取值：0~1
public static final int ALR_FACE_SHAPE_TYPE_EYE_ANGLE1   = 9; //眼角1，取值：0~1
public static final int ALR_FACE_SHAPE_TYPE_CANTHUS     = 10;//眼距，取值：-1~1
public static final int ALR_FACE_SHAPE_TYPE_CANTHUS1    = 11;//拉宽眼距，取值：-1~1
public static final int ALR_FACE_SHAPE_TYPE_EYE_ANGLE2   = 12;//眼角2，取值：-1~1
public static final int ALR_FACE_TYPE_EYE_TDANGLE       = 13;//眼睛高度，取值：-1~1
public static final int ALR_FACE_SHAPE_TYPE_THIN_NOSE    = 14;//瘦鼻，取值：0~1
public static final int ALR_FACE_SHAPE_TYPE_NOSEWING     = 15;//鼻翼，取值：0~1
public static final int ALR_FACE_SHAPE_TYPE_NASAL_HEIGHT  = 16;//鼻长，取值：-1~1
public static final int ALR_FACE_TYPE_NOSE_TIP_HEIGHT    = 17;//鼻头长，取值：-1~1
public static final int ALR_FACE_SHAPE_TYPE_MOUTH_WIDTH   = 18;//唇宽，取值：-1~1
public static final int ALR_FACE_TYPE_MOUTH_SIZE        = 18;//嘴唇大小，取值：-1~1
public static final int ALR_FACE_SHAPE_TYPE_MOUTH_HIGH    = 20;//唇高，取值：-1~1
public static final int ALR_FACE_SHAPE_TYPE_PHILTRUM     = 21;//人中
```

- processTexture：美颜美型纹理输入渲染。

```
int processTexture(int texture, int width, int height, int rotation, int flag);
```

参数	类型	描述
texture	int	输入图像纹理
width	int	输入图像纹理宽度
height	int	输入图像纹理高度
rotation	int	输入图像纹理旋转角度（顺时针）
flags	int	RTC对接时该参数必须设置为0

返回值：成功返回美颜处理后的textureId，失败返回<0。

- destroy：销毁美颜美型实例。

```
void destroy();
```

RTC对接RACE美颜

1. 设置RTC Engine美颜通道。

   RTC中使用外置美颜功能时，必须打开Engine的美颜通道。美颜通道的打开是通过扩展字段添加`jsonObject.addProperty("user_specified_video_preprocess", "TRUE");`，jsonObject转化成string后传入到AliRtcEngine.getInstance的第二个字段extra里面。详细使用方式参见如下代码：

```
JSONObject jsonObject = new JSONObject();
//初始化支持美颜的SDK
jsonObject.put("user_specified_video_preprocess", "TRUE");
//实例化必须在主线程进行
mAliRtcEngine = AliRtcEngine.getInstance(getApplicationContext(), jsonObject.toString());
```

2. 订阅RTC本地预览Texture回调。 在调用本地预览开启接口startPreview之后，调用RegisterTexturePreObserver订阅opengl纹理数据。userId填写为英文半角双引号（""）。

   > **说明** observer为实例化接口类AliTextureObserver。

   ```
   //订阅美颜texture回调
   mAliRtcEngine.RegisterTexturePreObserver("", raceBeautyImp);
   ```

3. RTC SDK Texture回调对接RACE美颜。

   订阅本地预览Texture成功后， RTC SDK会触发以下三个回调：

   - onTextureCreate：Texture创建回调。在该回调中创建和初始化RACE美颜。

     ```
     @Override
     public void onTextureCreate(String callId, long context) {
       if (mBeautifyNative == null) {
         mBeautifyNative = new AliyunBeautifyNative(mContext);
       }
       //初始化
       mBeautifyNative.initialize();
       //美颜开关
       mBeautifyNative.setFaceSwitch(true);
     }
     ```

   - onTexture：Texture渲染回调。在该回调中调用RACE接口processTexture进行每一帧的美颜处理。

     > **注意** 请您一定要判断处理返回值texId是否有效，返回texId<0时该回调必须要返回参数中原始inputTexture。磨皮、锐化、美白等级的控制，请您在该回调中美颜处理前调用相关的接口。

```
@Override
public int onTexture(String callId, int inputTexture, int textureWidth, int textureHeight, int stride, int rotate, long extra) {
  int texId = inputTexture;
  if (!isOn) {
    return inputTexture;
  }
  mBeautifyNative.setSkinBuffing(mSkinBuffing);
  mBeautifyNative.setSkinWhitening(mSkinWhitening);
  mBeautifyNative.setSharpen(mSharpen);
  if (mBeautifyNative != null) {
    texId = mBeautifyNative.processTexture(inputTexture, textureWidth, textureHeight, rotate, 0);
    if (texId < 0) {
      texId = inputTexture;
    }
  }
  return texId;
}
```

- onTextureDestroy：Texture销毁回调。在该回调中执行RACE美颜SDK销毁相关的工作。

```
@Override
public void onTextureDestroy(String callId) {
  if (mBeautifyNative != null) {
    mBeautifyNative.destroy();
  }
}
```

4. RACE美颜控制。

   基础美颜控制：磨皮、锐化、美白。基础美颜控制必须在onTexture回调中调用processTexture之前进行设置。

   人脸检测和美型控制：如果需要打开人脸美型高级功能，通过接口setFaceSwitch打开开关，有关美型类别的等级设置可以通过接口setFaceShape进行设置。

   > **说明** 当您不再使用人脸美型功能时，请您关闭此功能，否则会有性能损耗。

5. 取消订阅RTC本地预览Texture回调。

   在关闭本地预览前取消订阅本地预览texture回调。

   ```
   mAliRtcEngine.UnRegisterTexturePreObserver("");
   ```

17.3. iOS

通过阅读本文，您可以了解阿里云RTC SDK对接RACE美颜的集成方式、接口说明和实现步骤。

集成RACE SDK

支持pods与本地集成两种方式。

- pods集成方式：

  ```
  pod 'AliyunRace', '1.0.0.12196201'
  ```

- 本地集成方式：

  获取AliyunRace.framework、Face3D.framwork、opencv2.framework文件并添加到工程中。

  > 说明 请您正确添加动态库Embed Frameworks。

RTC SDK对接RACE接口说明

以下为RTC SDK对接RACE时需要使用的接口说明，RACE美颜SDK中的其他接口不在本文说明。

API	描述
aliyun_beautify_create	创建美颜美型实例
aliyun_beautify_setFaceDebug	美颜美型调试开关
aliyun_setLogLevel	设置输出日志等级
aliyun_beautify_setSkinBuffing	设置磨皮等级
aliyun_beautify_setSharpen	设置锐化等级
aliyun_beautify_setSkinWhitening	设置美白等级
aliyun_beautify_setFaceSwitch	美型开关
aliyun_beautify_setFaceShape	设置美型等级
aliyun_beautify_processTextureToTexture	美颜美型纹理输入渲染
aliyun_beautify_destroy	销毁美颜美型实例

- aliyun_beautify_create：创建美颜美型实例。

```
int aliyun_beautify_create(race_t *handle);
```

参数	类型	描述
context	Context	上下文

返回值：成功返回0，失败返回<0。

- aliyun_beautify_setFaceDebug：美颜美型调试开关。

```
int aliyun_beautify_setFaceDebug(race_t handle, bool enable);
```

参数	类型	描述
handle	race_t	美颜美型句柄
enable	bool	调试开启标志

返回值：成功返回0，失败返回<0。

- aliyun_setLogLevel：设置输出日志等级。

```
void aliyun_setLogLevel(int level);
```

参数	类型	描述
level	int	日志等级

- aliyun_beautify_setSkinBuffing：设置磨皮等级。

```
int aliyun_beautify_setSkinBuffing(race_t handle, float level);
```

参数	类型	描述
handle	race_t	美颜美型句柄
level	float	磨皮等级，范围：0~1

返回值：成功返回0，失败返回<0。

- aliyun_beautify_setSharpen：设置锐化等级。

```
int aliyun_beautify_setSharpen(race_t handle, float level);
```

参数	类型	描述
handle	race_t	美颜美型句柄
level	float	锐化等级，范围：0~1

返回值：成功返回0，失败返回<0。

- aliyun_beautify_setSkinWhitening：设置美白等级。

```
int aliyun_beautify_setSkinWhitening(race_t handle, float level);
```

参数	类型	描述
handle	race_t	美颜美型句柄
level	float	美白等级，范围：0~1

返回值：成功返回0，失败返回<0。

- aliyun_beautify_setFaceSwitch：美型开关。

```
void aliyun_beautify_setFaceSwitch(race_t handle, bool switchOn);
```

参数	类型	描述
handle	race_t	美颜美型句柄

参数	类型	描述
switchOn	bool	是否开启人脸检测及美型处理，取值：true|false

- aliyun_beautify_setFaceShape：设置美型等级。

```
int aliyun_beautify_setFaceShape(race_t handle, ALRFaceShape type, float level);
```

参数	类型	描述
handle	race_t	美颜美型句柄
type	ALRFaceShape	美型类型
level	float	美型等级，可以超出ALRFaceShape中定义的参数范围

返回值：成功返回0，失败返回<0。

ALRFaceShape 定义和建议范围如下：

```
typedef enum ALRFaceShape
{
  ALR_FACE_TYPE_CUT_CHEEK    = 0,//颧骨，取值：0~1
  ALR_FACE_TYPE_CUT_FACE     = 1,//削脸，取值：0~1
  ALR_FACE_TYPE_THIN_FACE    = 2,//瘦脸，取值：0~1
  ALR_FACE_TYPE_LONG_FACE    = 3,//脸长，取值：0~1
  ALR_FACE_TYPE_LOWER_JAW    = 4,//下巴缩短，取值：-1~1
  ALR_FACE_TYPE_HIGHER_JAW   = 5,//下巴拉长，取值：-1~1
  ALR_FACE_TYPE_THIN_JAW     = 6,//瘦下巴，取值：-1~1
  ALR_FACE_TYPE_THIN_MANDIBLE = 7,//瘦下颌，取值：0~1
  ALR_FACE_TYPE_BIG_EYE      = 8, //大眼，取值：0~1
  ALR_FACE_TYPE_EYE_ANGLE1   = 9,//眼角1，取值：0~1
  ALR_FACE_TYPE_CANTHUS      = 10,//眼距，取值：-1~1
  ALR_FACE_TYPE_CANTHUS1     = 11,//拉宽眼距，取值：-1~1
  ALR_FACE_TYPE_EYE_ANGLE2   = 12,//眼角2，取值：-1~1
  ALR_FACE_TYPE_EYE_TDANGLE  = 13,//眼睛高度，取值：-1~1
  ALR_FACE_TYPE_THIN_NOSE    = 14,//瘦鼻，取值：0~1
  ALR_FACE_TYPE_NOSE_WING    = 15,//鼻翼，取值：0~1
  ALR_FACE_TYPE_NASAL_HEIGHT = 16,//鼻长，取值：-1~1
  ALR_FACE_TYPE_NOSE_TIP_HEIGHT = 17,//鼻头长，取值：-1~1
  ALR_FACE_TYPE_MOUTH_WIDTH  = 18,//唇宽，取值：-1~1
  ALR_FACE_TYPE_MOUTH_SIZE   = 19,//嘴唇大小，取值：-1~1
  ALR_FACE_TYPE_MOUTH_HIGH   = 20,//唇高，取值：-1~1
  ALR_FACE_TYPE_PHILTRUM     = 21, //人中
  ALR_FACE_TYPE_MAX          = 22
} ALRFaceShape;
```

- aliyun_beautify_processTextureToTexture：美颜美型纹理输入渲染。

```
int aliyun_beautify_processTextureToTexture(race_t handle,
                                            uint32_t textureIn,
                                            uint32_t width,
                                            uint32_t height,
                                            aliyun_rotation_t rotation,
                                            uint8_t flags);
```

参数	类型	描述
handle	race_t	美颜美型句柄
textureIn	uint32_t	输入图像纹理
width	uint32_t	输入图像纹理宽度
height	uint32_t	输入图像纹理高度
rotation	aliyun_rotation_t	输入图像纹理旋转角度（顺时针），RTC 对接时该参数必须设置为ALR_ROTATE_180_CW
flags	uint8_t	RTC对接时该参数必须设置为0

返回值：成功返回美颜处理后的textureId，失败返回<0。

- aliyun_beautify_destroy：销毁美颜美型实例。

```
void aliyun_beautify_destroy(race_t handle);
```

参数	类型	描述
handle	race_t	美颜美型句柄

对接RACE美颜

1. 设置RTC Engine美颜通道。

   RTC中使用外置美颜功能时，必须打开Engine的美颜通道。美颜通道的打开是通过在AliRtcEngine sharedInstance的第二个参数extra中添加 `user_specified_video_preprocess` 扩展字段，并设置为 *TRUE*。

```
#pragma mark extras配置
- (NSString *)serializeExtrasJson {
  NSMutableDictionary *extrasDic = [[NSMutableDictionary alloc] init];
  [extrasDic setValue:@"TRUE" forKey:@"user_specified_video_preprocess"];
  if (extrasDic.count == 0) {
    return @"";
  }
  NSError *parseError = nil;
  NSData *jsonData = [NSJSONSerialization dataWithJSONObject:extrasDic options:NSJSONWritingPr
ettyPrinted error:&parseError];
  return [[NSString alloc] initWithData:jsonData encoding:NSUTF8StringEncoding];
}
 _engine = [AliRtcEngine sharedInstance:self extras:extras];
```

2. 订阅RTC本地预览Texture回调。

   在调用接口startPreview（本地预览开启）之后，调用subscribeVideoTexture订阅opengl纹理数据。uid填写为英文半角双引号（""），videoSource填写为AliRtcVideosourceCameraLargeType，videoTextureType填写为AliRtcVideoTextureTypePre。

```
// 开启本地预览
[self.engine startPreview];
[self.engine subscribeVideoTexture:@"" videoSource:AliRtcVideosourceCameraLargeType videoTextur
eType:AliRtcVideoTextureTypePre];  /* 美颜订阅openGL纹理数据 */
```

3. RTC SDK Texture回调对接RACE美颜。

   订阅本地预览Texture成功后， RTC SDK会触发以下三个回调：

   - onVideoTextureCreated：Texture创建回调。在该回调中创建和初始化RACE美颜句柄。

```
- (void)onVideoTextureCreated:(NSString *)uid videoTextureType:(AliRtcVideoTextureType)videoTe
xtureType context:(void *)context
{
  if(aliRaceBuautyHandle){
    aliyun_beautify_destroy(aliRaceBuautyHandle);
    aliRaceBuautyHandle = nullptr;
  }
  aliyun_beautify_create(&aliRaceBuautyHandle);
  aliyun_beautify_setFaceSwitch(aliRaceBuautyHandle, true);
  aliyun_setLogLevel(ALR_LOG_LEVEL_ERROR);
  //如需初始化美颜参数请编写代码
  SkinBuffing = 1;
  Sharpen = 1;
  SkinWhitening = 0.5;
  aliyun_beautify_setSkinBuffing(aliRaceBuautyHandle, SkinBuffing);
  aliyun_beautify_setSharpen(aliRaceBuautyHandle, Sharpen);
  aliyun_beautify_setSkinWhitening(aliRaceBuautyHandle, SkinWhitening);
}
```

   - onVideoTexture：Texture渲染回调。

     在该回调中，调用RACE接口aliyun_beautify_processTextureToTexture进行每一帧的美颜处理。

> **注意** 请您判断处理返回值textureId是否有效，返回textureId<0时该回调必须要返回参数中原始textureId。磨皮、锐化、美白等级的控制，请您在该回调中美颜处理前调用相关的接口。

```
- (int)onVideoTexture:(NSString *)uid videoTextureType:(AliRtcVideoTextureType)videoTextureType textureId:(int)textureId width:(int)width height:(int)height rotate:(int)rotate extraData:(long)extraData{
  if (beautifySwitchOn == NO) {
    return textureId;
  }
  aliyun_beautify_setSkinBuffing(aliRaceBuautyHandle, SkinBuffing);
  aliyun_beautify_setSharpen(aliRaceBuautyHandle, Sharpen);
  aliyun_beautify_setSkinWhitening(aliRaceBuautyHandle, SkinWhitening);
  int texId = aliyun_beautify_processTextureToTexture(aliRaceBuautyHandle,textureId,width,height,ALR_ROTATE_180_CW,0);
  if(texId<0) {
    texId = textureId;
  }
  return texId;
}
```

- onVideoTextureDestory：Texture销毁回调。在该回调中执行RACE美颜SDK销毁相关的工作。

```
- (void)onVideoTextureDestory:(NSString *)uid videoTextureType:(AliRtcVideoTextureType)videoTextureType{
  if(aliRaceBuautyHandle){
    aliyun_beautify_destroy(aliRaceBuautyHandle);
    aliRaceBuautyHandle = nullptr;
  }
}
```

4. RACE美颜控制。

   基础美颜控制：磨皮、锐化、美白。基础美颜控制必须在onVideoTexture回调中调用aliyun_beautify_processTextureToTexture之前进行设置。

   人脸检测和美型控制：如果需要打开人脸美型高级功能，通过接口aliyun_beautify_setFaceSwitch打开开关，有关美型类别的等级设置可以通过接口aliyun_beautify_setFaceShape进行设置。

   > **说明** 当您不使用人脸美型功能时，请不要打开此功能，否则会有性能损耗。

5. 取消订阅RTC本地预览Texture回调。

   在关闭本地预览前取消订阅本地预览texture回调。

```
[self.engine unSubscribeVideoTexture:@"" videoSource:AliRtcVideosourceCameraLargeType videoTextureType:AliRtcVideoTextureTypePre];
[self.engine stopPreview];
```

18.音视频输出

18.1. Android

本文为您介绍Android端音视频输出功能调用接口的具体步骤。

输出视频媒体

1. 当应用需要输出视频媒体数据时，需先注册AliVideoObserver回调，实现onLocalVideoSample和onRemoteVideoSample回调，用于接收本地采集视频裸数据，以及订阅到的远端视频裸数据。

```
//接收本地数据回调 void onLocalVideoSample(AliVideoSourceType sourceType, AliVideoSample videoSa
mple);
//接收远端数据回调 void onRemoteVideoSample(String callId, AliVideoSourceType sourceType, AliVideo
Sample videoSample);
```

> **说明** 目前Android端数据输出默认本地数据回调只支持输出YUV（NV21）格式数据，远端数据只支持输出YUV（I420）。应用侧如果有其他格式需求，您可以自行转换到其他格式。

2. 应用在创建AliRtcEngine引擎实例之后，调用接口registerVideoSampleObserver注册视频裸数据输出，注册后开启预览或订阅拉流时，视频数据将通过onCaptureVideoSample和onRemoteVideoSample持续回调。

```
AliRtcEngine mAliRtcEngine = AliRtcEngine.getInstance(getApplicationContext());
// 注册视频裸数据输出
mAliRtcEngine.registerVideoSampleObserver(new AliRtcEngine.AliVideoObserver() {
        @Override
        public void onLocalVideoSample(AliRtcEngine.AliVideoSourceType sourceType, AliRtcEngine.Ali
VideoSample videoSample) {
        }
        @Override
        public void onRemoteVideoSample(String callId, AliRtcEngine.AliVideoSourceType sourceType,
AliRtcEngine.AliVideoSample videoSample) {
        }
    });
//开始预览/推流及订阅其他用户视频
.....
```

> **说明** 本地采集视频数据需要在开启摄像头（打开预览或推送视频流）前提下才会有回调输出，远端视频数据同样需要在订阅其他用户的视频流成功后才会有回调输出。

3. 需要停止接收视频裸数据时，调用接口unRegisterVideoSampleObserver关闭视频裸数据输出即可。

```
// 停止视频裸数据输出
mAliRtcEngine.unRegisterVideoSampleObserver();
```

输出音频媒体

1. 当应用需要输出音频媒体数据时，首先需要先注册registerAudioObserver接口，实现AliAudioObserver回调，用于接收音频媒体数据。数据格式为PCM数据，目前SDK支持输出不同环节的音频数据，注册通过AliAudioType参数指明当前回调音频数据类型。具体含义如下：

- PUBOBSERVER：经过音频3A处理后的音频数据。
- SUBOBSERVER：当前订阅到的远端用户混音后的音频数据。
- RAWDATAOBSERVER：本地采集的原始音频数据。

```
// AliAudioObserver接口数据回调
public interface AliAudioObserver {
/* @param dataPtr 音频数据
@param numSamples 采样数
@param bytesPerSample 采样位数（字节）
@param numChannels 声道数
@param sampleRate 采样率
@param samplesPerSec 采样位数（字节）
*/
void onCaptureRawData(long dataPtr, int numSamples, int bytesPerSample, int numChannels, int sam
pleRate, int samplesPerSec);
void onCaptureData(long dataPtr, int numSamples, int bytesPerSample, int numChannels, int sampleR
ate, int samplesPerSec);
void onRenderData(long dataPtr, int numSamples, int bytesPerSample, int numChannels, int sampleRa
te, int samplesPerSec);
}
```

2. 应用在创建AliRtcEngine引擎实例之后，注册音频数据回调，其中参数AliAudioType指定需要输出的音频数据类型（类型参见上一步骤说明），如果需要输出多种类型数据，需要分别调用注册。注册完成后，音频数据将通过AliAudioObserver持续回调。

```
AliRtcEngine mAliRtcEngine = AliRtcEngine.getInstance(getApplicationContext());
// 开始本地采集音频数据输出(type输出其他类型参见枚举AliAudioType)
mAliRtcEngine.registerAudioObserver(type,
        new AliRtcEngine.AliAudioObserver() {
          @Override
          public void onCaptureRawData(long dataPtr, int numSamples, int bytesPerSample, int numCh
annels, int sampleRate, int samplesPerSec) {
          }
          @Override
          public void onCaptureData(long dataPtr, int numSamples, int bytesPerSample, int numChann
els, int sampleRate, int samplesPerSec) {
          }
          @Override
          public void onRenderData(long dataPtr, int numSamples, int bytesPerSample, int numChanne
ls, int sampleRate, int samplesPerSec) {
          }
        });
// 开始音频推流及订阅其他用户视频
.....
```

> 说明
>
> - 本地采集音频数据需要在开启麦克风（开始推送音频流）前提下才会有回调输出，远端视频数据同样需要在订阅其他用户的音频流成功后才会有回调输出。
> - 音频类型枚举AliAudioType中的VOLUME_DATA_OBSERVER类型为注册用户音量值回调，目前已经废弃，从1.16版本开始请您使用registerAudioVolumeObserver接口。

3. 需要停止接收视频裸数据时，调用接口unRegisterAudioObserver关闭音频数据输出，停止类型需要对应步骤2中注册输出的音频数据类型。

```
// 停止本地采集音频数据输出，停止类型需要对应步骤2中注册输出的音频数据类型AliAudioType
pEngine->unRegisterAudioObserver(type);
```

18.2. iOS和Mac

本文为您介绍了iOS和Mac端音视频输出功能调用接口的具体步骤。

订阅音频裸数据

1. 设置入会模式为音乐模式。

   接口方法：

   ```
   + (instancetype)sharedInstance:(id<AliRtcEngineDelegate>)delegate extras:(NSString *)extras;
   ```

   通过设置extras参数来控制是否为音乐模式。

   ```
   {
    "user_specified_scene_mode" : "SCENE_MUSIC_MODE",
   }
   ```

   > **说明** extras为json字符串，如果当前使用了其他key值的extras配置，可以共存。

   音乐模式示例代码如下所示。

   ```
   NSMutableDictionary *extrasDic = [[NSMutableDictionary alloc] init];
   [extrasDic setValue:@"ENGINE_BASIC_QUALITY_MODE" forKey:@"user_specified_engine_mode"];
   [extrasDic setValue:@"SCENE_MUSIC_MODE" forKey:@"user_specified_scene_mode"];
   NSError *parseError = nil;
   NSData *jsonData = [NSJSONSerialization dataWithJSONObject:extrasDic options:NSJSONWritingPret
   tyPrinted error:&parseError];
   NSString *extras = [[NSString alloc] initWithData:jsonData encoding:NSUTF8StringEncoding];
   AliRtcEngine *engine = [AliRtcEngine sharedInstance:self extras:extras];
   ```

2. 获取音频推流裸数据。

i. 注册音频裸数据。

接口方法如下：

```
- (void)subscribeAudioData:(AliRtcAudioSource)audioSource;
```

> **说明** 该接口必须在音频流发布成功之后调用才能生效；调用此接口后，即可开始订阅音频裸数据，裸数据通过回调接口返回。

参数说明如下：

参数	说明
AliRtcAudioSource	AliRtcAudiosourcePub：推流端数据 AliRtcAudiosourceSub：拉流端数据

如果您想停止获取音频裸数据，调用以下接口：

```
- (void)unSubscribeAudioData:(AliRtcAudioSource)audioSource;
```

ii. 通过回调获取对应的音频裸数据。

回调接口如下：

```
- (void)onAudioSampleCallback:(AliRtcAudioSource)audioSource audioSample:(AliRtcAudioDataSample *)audioSample;
```

> **说明** 调用subscribeAudioData接口后，通过此回调获取对应的音频裸数据。

AliRtcAudioDataSample参数说明如下：

参数	说明
dataPtr	裸数据data
numOfSamples	音频样本点数
bytesPerSample	量化位数
numOfChannels	声道数
samplesPerSec	采样率

iii. 推流音频裸数据写入本地pcm文件。

```
{
// 开始订阅（在音频流publish成功之后）
[self.engine subscribeAudioData:(AliRtcAudiosourcePub)];
}
```

```
// 订阅的音频数据回调
- (void)onAudioSampleCallback:(AliRtcAudioSource)audioSource audioSample:(AliRtcAudioDataSample *)audioSample {
  if (audioSample.dataPtr == 0) {
    //没有音频订阅数据
    return;
  }
  if (audioSource == AliRtcAudiosourcePub) {
    // pub数据回调
  }
  // 获取buffer 和 bufferSize
  int bufferSize = audioSample.numOfSamples * audioSample.bytesPerSample * audioSample.numOfChannels;
  void *audioSampleBufferPtr = NULL;
  if(bufferSize) {
    audioSampleBufferPtr = malloc(bufferSize);
    if(audioSampleBufferPtr) {
      memcpy(audioSampleBufferPtr, (void *)audioSample.dataPtr, bufferSize);
    }
  }
  // 写入文件（此处省略FILE对象的初始化和销毁等，请写在外部）
  fwrite(audioSampleBufferPtr, 1, bufferSize, file);
  if (audioSampleBufferPtr) {
    free(audioSampleBufferPtr);
    audioSampleBufferPtr = NULL;
  }
}
```

订阅视频裸数据

1. 注册视频裸数据。

   接口方法如下：

   ```
   - (void)registerVideoSampleObserver;
   ```

   > **说明** 调用此接口后，即可开始订阅视频裸数据，裸数据通过回调接口给出。

   如果想停止获取音频裸数据，调用以下接口：

   ```
   - (void)unregisterVideoSampleObserver;
   ```

2. 通过回调获取视频裸数据。

   回调接口方法如下：

```
/**
* @brief 订阅的本地采集视频数据回调
* @param videoSource video source
* @param videoSample video sample
*/
- (void)onCaptureVideoSample:(AliRtcVideoSource)videoSource videoSample:(AliRtcVideoDataSample
*)videoSample;
/**
* @brief 订阅的远端视频数据回调
* @param uid user id
* @param videoSource video source
* @param videoSample video sample
*/
- (void)onRemoteVideoSample:(NSString *)uid videoSource:(AliRtcVideoSource)videoSource videoSam
ple:(AliRtcVideoDataSample *)videoSample;
```

> **说明** 调用registerVideoSampleObserver接口后，通过这两个回调获取对应的视频裸数据。
>
> - onCaptureVideoSample为预览数据回调，在开始预览之后可收到数据流。
> - onRemoteVideoSample为拉流数据回调，subscribe拉流成功后可收到数据流。

3. 预览视频裸数据写入本地yuv文件。

```
{
// 开始订阅
[_engine registerVideoSampleObserver];
}
```

```
- (void)onCaptureVideoSample:(AliRtcVideoSource)videoSource videoSample:(AliRtcVideoDataSample
*)videoSample {
  [self dumpVideoYuvData:@"" videoSource:videoSource videoSample:videoSample];
}
- (void)onRemoteVideoSample:(NSString *)uid videoSource:(AliRtcVideoSource)videoSource videoSam
ple:(AliRtcVideoDataSample *)videoSample {
  [self dumpVideoYuvData:uid videoSource:videoSource videoSample:videoSample];
}
- (void)dumpVideoYuvData:(NSString *)uid videoSource:(AliRtcVideoSource)videoSource videoSample:
(AliRtcVideoDataSample *)videoSample {
  //创建串行队列
    dispatch_queue_t testqueue = dispatch_queue_create("subVideo", NULL);
    //同步执行任务
    dispatch_sync(testqueue, ^{
      if(_dumpYUVVideoData || _remoteYUVVideoData){
        if (videoSource != AliRtcVideosourceCameraLargeType) {
          return;
        }
        if (!_subDataMutableDic[uid]) {
          NSString *time = [self getCurrentTimes];
          NSString *filePath = [NSString stringWithFormat:@"Documents/yuv/%@/",uid];
          if ([uid isEqualToString:@""]) {
            filePath = [NSString stringWithFormat:@"Documents/yuv/self/"];
          }
          NSString *fileName = [NSString stringWithFormat:@"sub_%@_video_%d*%d_%@.yuv",uid,vi
```

```
deoSample.width,videoSample.height,time];
            if ([uid isEqualToString:@""]) {
              fileName = [NSString stringWithFormat:@"pub_%@_video_%d*%d_%@.yuv",uid,videoSam
ple.width,videoSample.height,time];
            }
            NSString * dataPath = [SubFilePath getSubDataFilePath:filePath FileName:fileName];
            subYUVFile = fopen([dataPath UTF8String], "ab");
            SubDataModel *cache = [[SubDataModel alloc] init];
            cache.filePath = dataPath;
            cache.outputFile = subYUVFile;
            [_subDataMutableDic setObject:cache forKey:uid];
          }
          SubDataModel *cache = _subDataMutableDic[uid];
          subYUVFile = cache.outputFile;
          uint8_t *outputData = NULL;
          int32_t w = videoSample.width;
          int32_t h = videoSample.height;
          if (outputData == NULL) {
            outputData = (uint8_t*)malloc(w*h*3/2);
          }
          if(videoSample.type == AliRtcBufferType_Raw_Data) {
            if(videoSample.dataYPtr && videoSample.dataUPtr &&  videoSample.dataVPtr) {
              memcpy(outputData, (uint8_t*)videoSample.dataYPtr,w*h*3/2);
              memcpy(outputData + w*h, (uint8_t*)videoSample.dataUPtr, w*h/4);
              memcpy(outputData + w*h*5/4, (uint8_t*)videoSample.dataVPtr, w*h/4);
            }
          }else if(videoSample.type == AliRtcBufferType_CVPixelBuffer) {
            if(videoSample.pixelBuffer) {
              CVPixelBufferRef newBuffer = videoSample.pixelBuffer;
              if(newBuffer) {
                CVReturn cvRet = CVPixelBufferLockBaseAddress(newBuffer, 0);
                if ( cvRet != kCVReturnSuccess ) {
                  return;
                }
                void*  src_y = CVPixelBufferGetBaseAddressOfPlane(newBuffer, 0);
                void*  src_uv = CVPixelBufferGetBaseAddressOfPlane(newBuffer, 1);
                size_t src_y_stride = CVPixelBufferGetBytesPerRowOfPlane(newBuffer, 0);
                size_t src_uv_stride = CVPixelBufferGetBytesPerRowOfPlane(newBuffer, 1);
                uint8_t *destData = outputData;
                for (int row = 0; row < h; ++row) {
                  memcpy(destData, (uint8_t*)src_y + row * src_y_stride,w);
                  destData += w;
                }
                for (int row = 0; row < (h + 1) / 2; ++row) {
                  memcpy(destData, (uint8_t*)src_uv + row * src_uv_stride,w);
                  destData += w;
                }
                CVPixelBufferUnlockBaseAddress(newBuffer, 0);
              }
            }
          }
          fwrite(outputData, 1, w*h*3/2, subYUVFile);
          if (outputData) {
            free(outputData);
```

```
            outputData = NULL;
        }
    }
});
}
```

18.3. Windows

本文为您介绍Windows端音视频输出功能调用接口的具体步骤。

输出视频媒体

1. 当应用需要输出视频媒体数据时，需先继承AliRtcEventListener接口，实现onCaptureVideoSample和onRemoteVideoSample回调，用于接收本地采集视频裸数据，以及订阅到的远端视频裸数据。

```
//接收裸数据回调
void onCaptureVideoSample(AliRtcVideoSource videoSource, AliRtcVideoDataSample *videoSample) {
//处理本地采集视频数据
}
void onRemoteVideoSample(const AliRtc::String &uid, AliRtcVideoSource videoSource, AliRtcVideoDataSample *videoSample) {
//处理远端视频数据
}
```

> **说明** 目前Windows端数据输出只支持输出YUV（I420）格式数据，应用侧可通过SDK提供的AliConvertVideoData静态接口，转换到RGBA格式。

2. 应用在创建AliRtcEngine引擎实例之后，调用接口registerVideoSampleObserver注册视频裸数据输出，注册后开启预览或订阅拉流时，视频数据将通过onCaptureVideoSample和onRemoteVideoSample持续回调。

```
AliRtcEngine *pEngine = AliRtcEngine::sharedInstance(this, "");
// 注册视频裸数据输出
pEngine->registerVideoSampleObserver();
//开始预览/推流及订阅其他用户视频
.....
```

> **说明** 本地采集视频数据需要在开启摄像头（打开预览或推送视频流）前提下才会有回调输出，远端视频数据同样需要在订阅其他用户的视频流成功后才会有回调输出。

3. 需要停止接收视频裸数据时，调用接口unRegisterVideoSampleObserver关闭视频裸数据输出即可。

```
// 停止视频裸数据输出
pEngine->unRegisterVideoSampleObserver();
```

输出音频媒体

1. 当应用需要输出音频媒体数据时，首先需要先继承AliRtcEventListener接口，实现onAudioSampleCallback回调，用于接收音频媒体数据。

   音频数据通过回调中audioSample参数返回，数据格式为PCM数据，目前SDK支持输出不同环节的音频数据，回调中通过type参数指明当前回调音频数据类型。

具体含义如下：

- AliRtcAudiosourceRawData：本地采集的原始音频数据。
- AliRtcAudiosourcePub：经过音频3A处理后的音频数据。
- AliRtcAudiosourceRawData：当前订阅到的远端用户混音后的音频数据。

```
// 音频媒体数据结构
typedef struct tagAliRtcAudioDataSample { unsigned char *data{nullptr};
//音频数据 int numOfSamples{0};
//采样数 int bytesPerSample{0};
//采样位数（字节）int numOfChannels{0};
//声道数 int samplesPerSec{0};
//采样率 }AliRtcAudioDataSample;
// 接收并处理音频数据回调 void void onAudioSampleCallback(AliRtcAudioSource type, AliRtcAudioDataS
ample *audioSample) {
};
```

2. 应用在创建AliRtcEngine引擎实例之后，注册音频数据回调，其中参数AliAudioType指定需要输出的音频数据类型（类型参见上一步骤说明），如果需要输出多种类型数据，需要分别调用注册。注册完成后，音频数据将通过AliAudioObserver持续回调。

```
AliRtcEngine mAliRtcEngine = AliRtcEngine.getInstance(getApplicationContext());
// 开始本地采集音频数据输出(type输出其他类型参见枚举AliAudioType)
mAliRtcEngine.registerAudioObserver(type,
        new AliRtcEngine.AliAudioObserver() {
          @Override
          public void onCaptureRawData(long dataPtr, int numSamples, int bytesPerSample, int numCh
annels, int sampleRate, int samplesPerSec) {
          }
          @Override
          public void onCaptureData(long dataPtr, int numSamples, int bytesPerSample, int numChann
els, int sampleRate, int samplesPerSec) {
          }
          @Override
          public void onRenderData(long dataPtr, int numSamples, int bytesPerSample, int numChanne
ls, int sampleRate, int samplesPerSec) {
          }
        });
// 开始音频推流及订阅其他用户视频
.....
```

> **说明**
>
> - 本地采集音频数据需要在开启麦克风（开始推送音频流）前提下才会有回调输出，远端视频数据同样需要在订阅其他用户的音频流成功后才会有回调输出。
> - 音频类型枚举AliRtcAudioSource中的AliRtcAudioSourceVolume类型为注册用户音量值回调，注册订阅该类型时，用户音量将通过onAudioVolumeCallback回调返回，与音频数据输出回调不同。

3. 需要停止接收视频裸数据时，调用接口unsubscribeAudioData关闭音频数据输出，停止类型需要对应步骤2中注册输出的音频数据类型。

```
// 停止本地采集音频数据输出
pEngine->unsubscribeAudioData(AliRtcAudiosourceRawData);
```

18.4. Web

本文为您介绍Web端音频输出功能调用接口的具体步骤。

输出音频媒体

1. 开启音频数据接收回调。

```
aliWebrtc.enableAudioVolumeIndicator = true;
```

> 说明 该接口可以在实例化后任何时间开启。

2. 使用音频能量值回调。

```
aliWebrtc.on("onAudioLevel", (data) => {
  console.log(data)
})
```

> 说明 需要设置enableAudioVolumeIndicator后回调，每秒返回一次（返回每秒音频能量最大值）。

返回结果说明：

- 当您推了音频流，返回数组中userId为字符串0的一项，是自己的音频信息。
- 当您订阅了其他用户的音频流，该数组中会包含订阅用户的音频信息。
- 具体数组各项的信息如下所示：

返回值	类型	描述
userId	String	订阅用户userId，用户自己的userId为0。
displayName	String	用户名。
level	Number	音频能量值，取值范围0~100。
buffer	Array	每秒内音频PCM的数据。

3. 关闭音频数据接收回调。

当您需要停止接收音频数据，可以设置关闭音频数据接收。

```
aliWebrtc.enableAudioVolumeIndicator = false;
```

19.屏幕分享使用

19.1. Mac

阿里云RTC SDK为您提供屏幕分享使用的接口方法，通过本文档您可以了解实现的具体调用流程。

推流端

1. 启动屏幕分享。

```
// 配置屏幕分享推流
[self.engine configLocalScreenPublish:YES];
// 启动推流
[self.engine publish:^(int err) {
}];
.....
```

2. 停止屏幕分享。

```
// 配置屏幕分享停止
[self.engine configLocalScreenPublish:NO];
// 启动停推
[self.engine publish:^(int err) {
}];
```

订阅端

订阅端用户可通过自动或手动方式订阅推流端屏幕分享视频流，并设置对应View显示，详情请参见AliRtcEngine接口。

19.2. Windows

阿里云RTC SDK为您提供屏幕分享使用的接口方法，通过本文档您可以了解实现的具体调用流程。

背景信息

目前Windows SDK屏幕分享功能支持屏幕分享和窗口分享两种类型，其中屏幕分享还支持指定区域进行分享，应用侧可根据业务场景设置分享内容并推流。

> **说明** 本文将推流端的操作分为屏幕分享和窗口分享两种类型。

屏幕分享

1. 推流端创建SDK实例后，通过接口getScreenShareSourceInfo可以获取当前屏幕分享源，其中参数source_type指定AliRtcScreenShareDesktop，然后SDK通过source_list参数返回当前设备所有显示器设备。

```
AliRtcEngine *pEngine = AliRtcEngine::sharedInstance(this, "");
// 获取屏幕分享source
AliRtcScreenSourceList sourceList;
pEngine->getScreenShareSourceInfo(AliRtcScreenShareDesktop, sourceList);
// 遍历所有支持屏幕
for (int i = 0; i < sourceList.sourceNum; i++) {
}
```

> **说明** 屏幕分享只有在当前设备接入显示器超过1个时，会返回多个source。

2. 通过接口setScreenShareSource设置屏幕分享source，其中source参数中的sourceId设置为需要分享的屏幕source id（上一步骤中获取），同时如果业务需要只分享屏幕中的部分区域，则设置source参数中的isShareByRegion字段为true，同时将要分享区域的左上角坐标和宽高，赋值给source参数中的shareRegion变量。

```
// 设置屏幕分享源为屏幕分享，并指定分享屏幕source id
AliRtcScreenSource source;
source.eType = AliRtcScreenShareDesktop;
source.sourceId = sourceId;
// 如果需要按区域分享，则设置isShareByRegion为true，并指定分享区域，如分享屏幕左上角100 x 100的区域
source.isShareByRegion = true;
source.shareRegion.originX = 0.f;
source.shareRegion.originY = 0.f;
source.shareRegion.width = 100.f;
source.shareRegion.height = 100.f;
// 设置分享内容
pEngine->setScreenShareSource(source);
```

> **说明** 按区域分享时，设置分享区域最小分辨率为16x16，设置区域小于最小分辨率时重置为最小分辨率；按区域分享时，设置分享区域超过实际桌面分辨率时，将按照完整桌面分辨率分享，桌面分辨率可通过接口getDesktopResolution接口获取，用于分享区域设置校验。

3. 配置分享内容完成后，启动屏幕共享推流。

```
// 配置屏幕分享推流
pEngine->configLocalScreenPublish(true);
// 启动推流
pEngine->publish();
```

4. 结束分享时，配置屏幕共享流停推。

```
// 配置屏幕分享停止
pEngine->configLocalScreenPublish(false);
// 启动停推
pEngine->publish();
```

窗口分享

1. 推流端创建SDK实例后，通过接口getScreenShareSourceInfo可以获取当前屏幕分享源，其中参数source_type指定AliRtcScreenShareWindow，然后SDK通过source_list参数返回当前电脑上所有可见窗口的source id和source title。

```
AliRtcEngine *pEngine = AliRtcEngine::sharedInstance(this, "");
// 获取屏幕分享source
AliRtcScreenSourceList sourceList;
pEngine->getScreenShareSourceInfo(AliRtcScreenShareWindow, sourceList);
// 遍历所有支持分享窗口
for (int i = 0; i < sourceList.sourceNum; i++) {
}
```

> **说明** SDK只会返回所有当前可见（没有最小化且Size不为0）的窗口作为分享源。

2. 通过接口setScreenShareSource设置屏幕分享source，其中source参数的sourceId设置为需要分享窗口id（上一步骤中获取）。

```
// 设置屏幕分享源为屏幕分享，并指定分享屏幕source id
AliRtcScreenSource source;
source.eType = AliRtcScreenShareWindow;
source.sourceId = sourceId;
// 设置分享内容
pEngine->setScreenShareSource(source);
```

3. 配置分享内容完成后，启动屏幕共享推流。

```
// 配置屏幕分享推流
pEngine->configLocalScreenPublish(true);
// 启动推流
pEngine->publish();
```

> **说明** 设置分享源到启动推流过程中，如果指定的分享窗口发送变化（被关闭/不可见）导致无法分享，推流将会发生失败，业务方调用时需要关注推流结果是否成功；分享过程中，如果分享窗口被最小化，同样会导致SDK无法抓取数据，分享内容异常。

4. 结束分享时，配置屏幕共享流停推。

```
// 配置屏幕分享停止
pEngine->configLocalScreenPublish(false);
// 启动停推
pEngine->publish();
```

订阅端

订阅端用户可通过自动或手动方式订阅推流端屏幕分享视频流，并设置对应View显示，详情请参见AliRtcEngine接口。

19.3. Web

阿里云RTC SDK为您提供屏幕分享使用的接口方法，通过本文档您可以了解实现的具体调用流程。

前提条件

您需要调用isSupport接口根据返回参数supportScreenShare来检测是否支持屏幕分享。

环境要求

Web SDK屏幕共享功能的环境要求请参见环境要求。

推流端

1. 设置参数。

```
/**
 * 设置视频/屏幕流参数
 * @param {Number} width 宽度
 * @param {Number} height 高度
 * @param {Number} frameRate 帧率
 * @param {Number} type 类型 1：摄像头流 2：共享流
 */
aliWebrtc.setVideoProfile({
  width,
  height,
  frameRate
},type);
```

2. 启动屏幕分享。

```
// 配置屏幕共享推流
aliWebrtc.configLocalScreenPublish = true;
// 启动推流
aliWebrtc.publish().then(()=>{
  //推流成功
}).catch((err) => {
 //推流失败
})
```

3. 共享屏幕声音。

> **说明** 共享屏幕声音支持Windows端Chrome 75及以上版本或Edge 80及以上版本，Mac端仅支持分享标签页声音。

i. 勾选**分享音频**。

ii. 推音频流。

分享的音频会和麦克风混流，需要同时推音频流，此时订阅端只需订阅音频流就可以听到对方麦克风和屏幕分享音频。

4. 停止屏幕分享。

```
// 配置屏幕共享停止
aliWebrtc.configLocalScreenPublish = false;
// 启动停推
aliWebrtc.publish().then(()=>{
  //推流成功
}).catch((err) => {
 //推流失败
})
```

5. 错误码提示。

```
aliWebrtc.on("onError",(error)=>{
 //10010 屏幕共享未知错误
 //10011 屏幕共享在选择页面取消选择 屏幕共享被禁止
 //10012 屏幕共享在网页底部悬浮窗单击停止共享 屏幕共享已取消
  console.log(error.errorCode);
})
```

订阅端

订阅端用户可通过手动方式订阅推流端屏幕共享流，并设置对应video显示，详情请参见AliRtcEngine接口。

20.媒体扩展信息的使用

本文为您介绍媒体扩展信息的使用说明，您可以使用传递时间戳和位控制信息，以达到更好的产品体验。

使用场景

在以下两种场景下可以使用媒体扩展信息：

- 借用媒体链路传递时间戳，用于自己计算端到端网络延迟，或者跟自身其他业务做数据同步。
- 传递位控制信息，目前可以传递8Byte数据，共64位，每一位或者每几位数据可以表示一些控制信息，可以用于自身业务上的指令传输。

示例代码

以下代码均以时间戳为例。

- iOS
  - 发送：

```
NSTimeInterval now = [[NSDate dateWithTimeIntervalSinceNow:0] timeIntervalSince1970];
long long lNow = CFSwapInt64BigToHost((long long)(now * 1000));
NSData * timeData = [NSData dataWithBytes:&lNow length:sizeof(lNow)];
[self.engine sendMediaExtensionMsg:timeData repeatCount:5];
```

  - 接收：

```
- (void)onMediaExtensionMsgReceived:(NSString *)uid message:(NSData *)data {
   long long receivedLongLongValue = 0;
   [data getBytes:&receivedLongLongValue length:8];
   long long sentTime = CFSwapInt64BigToHost(receivedLongLongValue);
   NSTimeInterval now = [[NSDate dateWithTimeIntervalSinceNow:0] timeIntervalSince1970];
   long long nowTime = (long long)(now * 1000);
   NSLog(@"%@", [NSString stringWithFormat:@"uid: %@, now: %lld, sent: %lld, 差值: %lld", uid, nowT
ime, sentTime, (nowTime-sentTime)];)}
```

- Mac
  - 发送：

```
NSTimeInterval now = [[NSDate dateWithTimeIntervalSinceNow:0] timeIntervalSince1970];
long long lNow = CFSwapInt64BigToHost((long long)(now * 1000));
NSData * timeData = [NSData dataWithBytes:&lNow length:sizeof(lNow)];
[self.engine sendMediaExtensionMsg:timeData repeatCount:5];
```

  - 接收：

```
- (void)onMediaExtensionMsgReceived:(NSString *)uid message:(NSData *)data {
   long long receivedLongLongValue = 0;
   [data getBytes:&receivedLongLongValue length:8];
   long long sentTime = CFSwapInt64BigToHost(receivedLongLongValue);
   NSTimeInterval now = [[NSDate dateWithTimeIntervalSinceNow:0] timeIntervalSince1970];
   long long nowTime = (long long)(now * 1000);
   NSLog(@"%@", [NSString stringWithFormat:@"uid: %@, now: %lld, sent: %lld, 差值: %lld", uid, nowT
ime, sentTime, (nowTime-sentTime)];)
```

- Android
  - 发送：

```
long lNow = System.currentTimeMillis();
byte b[] = new byte[8];
ByteBuffer buf = ByteBuffer.wrap(b);
buf.putLong(l);
buf.flip();
mAliRtcEngine.sendMediaExtensionMsg(buf.array(), 5);
```

  - 接收：

```
public void onMediaExtensionMsgReceived(String uid, byte[] message) {
  super.onMediaExtensionMsgReceived(uid, message);
  ByteBuffer buffer = ByteBuffer.allocate(8);
  buffer.put(message, 0, message.length);
  buffer.flip();
  long sentTime = buffer.getLong();
}
```

- Windows
  - 发送：

```
char* long2charArr(long long num)
{
  char* arr = new char[8];
  int x = 56;
  for (int i = 7; i >= 0; i--)
  {
    long long temp = num << x;
    arr[i] = temp >> 56;
    x -= 8;
  }
  return arr;
}
SYSTEMTIME tmSys;
GetLocalTime(&tmSys);
CTime ctmNow(tmSys);
long long lNow = __int64(ctmNow.GetTime()) * 1000 + tmSys.wMilliseconds;
char* cNow = long2charArr(lNow);
mpEngine->sendMediaExtensionMsg((unsigned char*)cNow, 8, 5);
delete cNow;
```

- 接收：

```
long long charArr2long(char* p)
{
  long long value = 0;
  value = (((long long)p[0] << 56 & 0xFF00000000000000L) |
  ((long long)p[1] << 48 & 0xFF000000000000L) |
  ((long long)p[2] << 40 & 0xFF0000000000L) |
  ((long long)p[3] << 32 & 0xFF00000000L) |
  ((long long)p[4] << 24 & 0xFF000000L) |
  ((long long)p[5] << 16 & 0xFF0000L) |
  ((long long)p[6] << 8 & 0xFF00L) |
  ((long long)p[7] & 0xFFL));
  return value;
}
void CTutorialDlg::onMediaExtensionMsgReceived(const AliRtc::String &uid, unsigned char* message, i
nt size)
{
  long long sentTime = charArr2long((char*)message);
}
```

各个端使用注意点

因为复用音视频数据通道，必须控制自定义消息的发送频率和消息数据长度，使用限制如下：

- 为了不影响媒体数据的传输质量，自定义消息体长度限制为8Byte，可以用来传输时间戳，位控制信息等。
- 每秒最多发送30条消息。
- repeatCount为自定义消息冗余度，若大于1，则会发送多次，防止网络丢包导致的消息丢失。
- 若repeatCount大于1，房间里的其他人也会收到多次相同的消息，需要去重。
- 发送的自定义消息，在旁路直播时，房间里的订阅者也一样会收到。
- 目前H5不支持发送和接收。

21.语音数据处理

21.1. iOS和Mac

通过本文，您可以了解iOS和Mac端语音数据处理的功能。

场景

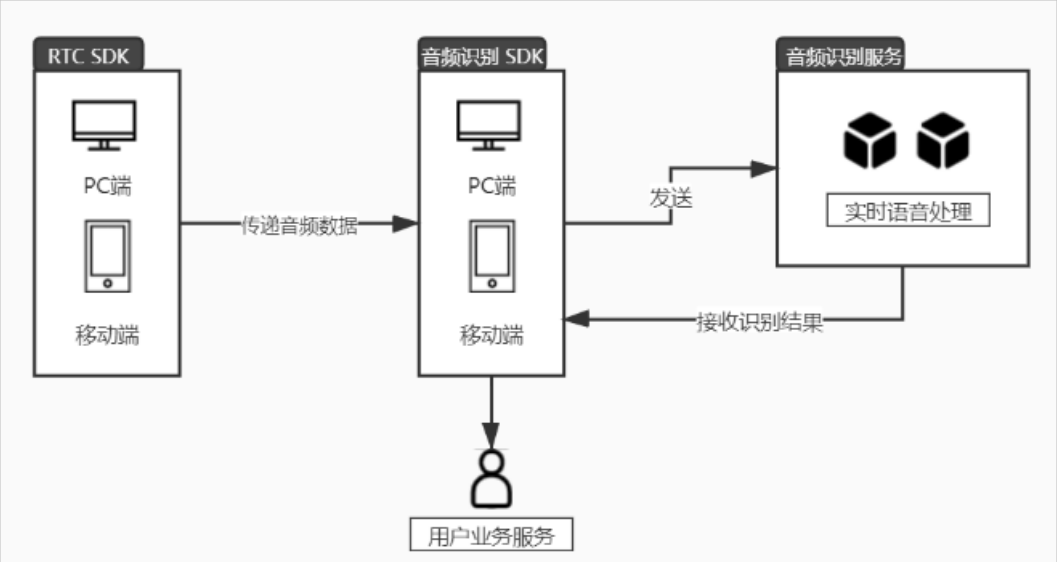
语音识别

阿里云的语音识别指的是将本地发布端或订阅端的音频数据转化为文字，实现流程如下：

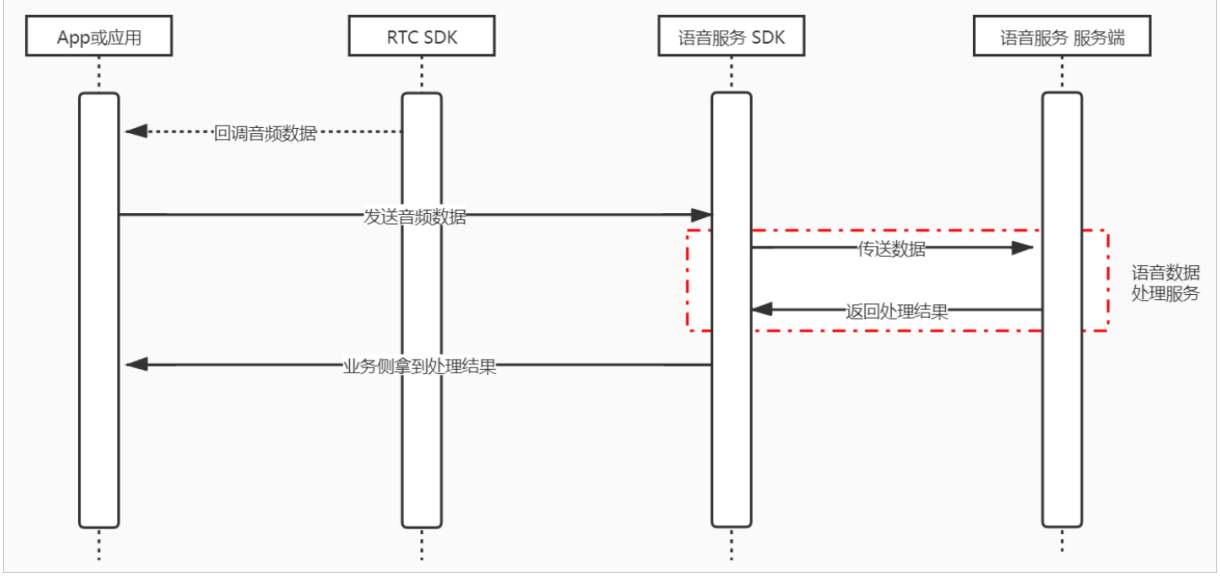
1. 阿里云RTC会将音频数据发送至音频识别SDK中。
2. 音频识别SDK将音频数据发送至音频识别服务进行实时语音处理并返回识别结果。
3. 音频识别SDK为用户提供识别结果。

详细请参见智能语音交互。

方案架构图



调用时序图



接口及使用

通过调用接口subscribeAudioData得到回调数据，从回调接口onAudioSampleCallback获取音频数据，并根据业务场景使用相应的数据源。

onAudioSampleCallback接口参数如下：

参数	类型	描述
audioSource	AliRtcAudioSource	音频裸数据源类型。
audioSample	AliRtcAudioDataSample *	音频裸数据。

AliRtcAudioSource音频裸数据源类型说明如下：

枚举名	描述
AliRtcAudiosourcePub	推流音频数据。
AliRtcAudiosourceSub	拉流音频数据。
AliRtcAudiosourceRawData	采集音频裸数据。
AliRtcAudiosourceVolume	音量。

> **说明** 采集音频裸数据为本地采集的原始音频数据，推流音频数据为经过音频3A处理后的音频数据。

AliRtcAudioDataSample音频裸数据说明如下：

参数	类型	描述
dataPtr	long	裸数据。
numOfSamples	int	音频样本点数。

参数	类型	描述
bytesPerSample	int	量化位数。
numOfChannels	int	声道数。
samplesPerSec	int	采样率。

语音数据处理

RTC获取音频数据方式如下：

1. 设置入会模式为音乐模式。

   接口方法：

   ```
   + (instancetype)sharedInstance:(id<AliRtcEngineDelegate>)delegate extras:(NSString *)extras;
   ```

   通过设置extras参数来控制是否为音乐模式：

   > 说明 extras为json字符串，如果当前使用了其他key值的extras配置，可以共存。

   ```
   {
    "user_specified_scene_mode" : "SCENE_MUSIC_MODE",
   }
   ```

   音乐模式示例代码如下所示：

   ```
   NSMutableDictionary *extrasDic = [[NSMutableDictionary alloc] init];
   [extrasDic setValue:@"ENGINE_BASIC_QUALITY_MODE" forKey:@"user_specified_engine_mode"];
   [extrasDic setValue:@"SCENE_MUSIC_MODE" forKey:@"user_specified_scene_mode"];
   NSError *parseError = nil;
   NSData *jsonData = [NSJSONSerialization dataWithJSONObject:extrasDic options:NSJSONWritingPret
   tyPrinted error:&parseError];
   NSString *extras = [[NSString alloc] initWithData:jsonData encoding:NSUTF8StringEncoding];
   AliRtcEngine *engine = [AliRtcEngine sharedInstance:self extras:extras];
   ```

2. 获取音频发布端裸数据。

   注册音频裸数据。接口方法如下：

   > 说明 该接口必须在音频流发布成功之后调用才能生效。调用此接口后，即可开始订阅音频裸数据，裸数据通过回调接口返回。

   ```
   - (void)subscribeAudioData:(AliRtcAudioSource)audioSource;
   ```

   如果您想停止获取音频裸数据，调用以下接口：

   ```
   - (void)unSubscribeAudioData:(AliRtcAudioSource)audioSource;
   ```

   通过回调获取对应的音频裸数据。回调接口如下：

   > 说明 调用subscribeAudioData接口后，通过此回调获取对应的音频裸数据。

```
{
// 开始订阅（在音频流publish成功之后）
[self.engine subscribeAudioData:(AliRtcAudiosourcePub)];
}
// 订阅的音频数据回调
- (void)onAudioSampleCallback:(AliRtcAudioSource)audioSource audioSample:(AliRtcAudioDataSampl
e *)audioSample {
  if (audioSample.dataPtr == 0) {
    //没有音频订阅数据
    return;
  }
  if (audioSource == AliRtcAudiosourcePub) {
    // pub数据回调
  }
  // 获取buffer 和 bufferSize
  int bufferSize = audioSample.numOfSamples * audioSample.bytesPerSample * audioSample.numOfC
hannels;
  void *audioSampleBufferPtr = NULL;
  if(bufferSize) {
    audioSampleBufferPtr = malloc(bufferSize);
    if(audioSampleBufferPtr) {
      memcpy(audioSampleBufferPtr, (void *)audioSample.dataPtr, bufferSize);
    }
  }
  // 写入文件（此处省略FILE对象的初始化和销毁等，请写在外部）
  fwrite(audioSampleBufferPtr, 1, bufferSize, file);
  if (audioSampleBufferPtr) {
    free(audioSampleBufferPtr);
    audioSampleBufferPtr = NULL;
  }
}
```

语音服务操作，详细请参见：智能语音交互。

21.2. Android

通过本文，您可以了解Android端语音数据处理的功能介绍。

场景

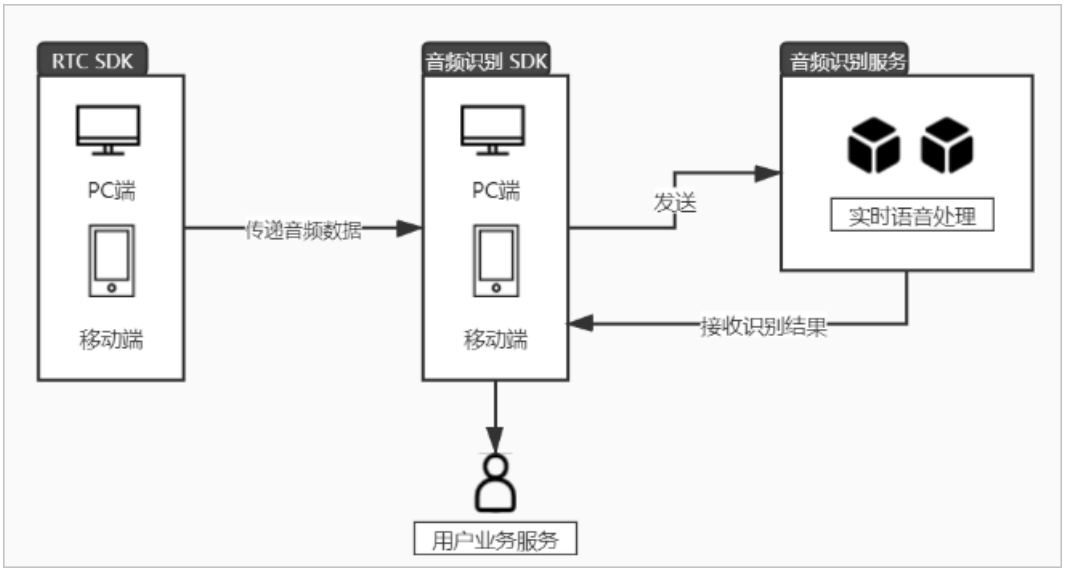
语音识别

阿里云的语音识别指的是将本地发布端或订阅端的音频数据转化为文字，实现流程如下：

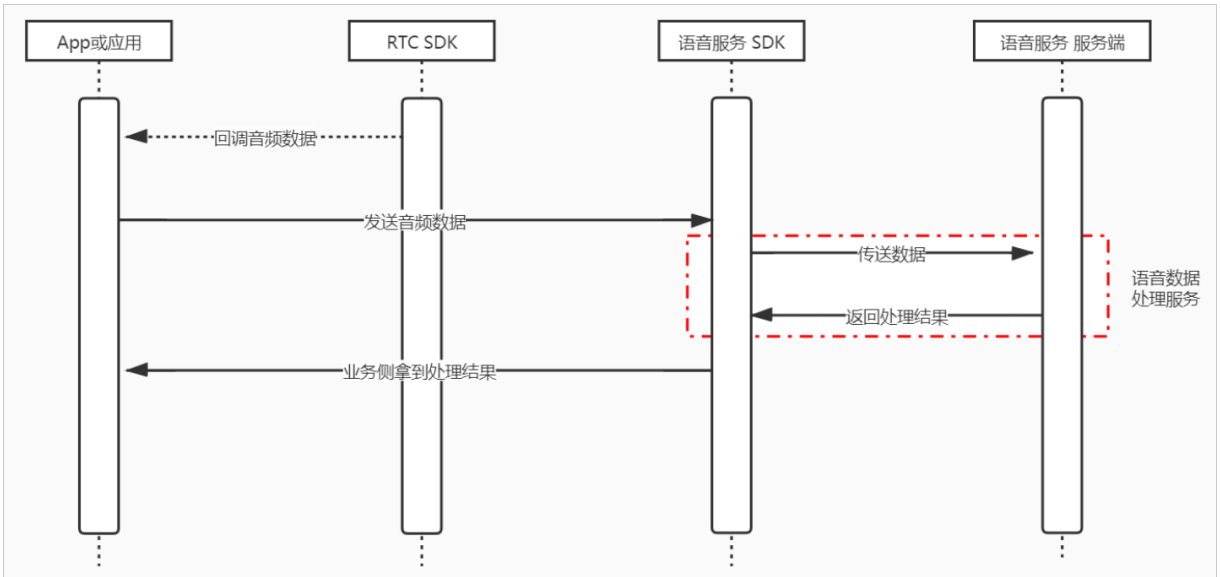
1. 阿里云RTC会将音频数据发送至音频识别SDK中。
2. 音频识别SDK将音频数据发送至音频识别服务进行实时语音处理并返回识别结果。
3. 音频识别SDK为用户提供识别结果。

详细请参见智能语音交互。

方案架构图



调用时序图



接口及使用

通过调用接口registerAudioObserver注册音频数据回调，注册时通过AliAudioType参数指明当前回调音频数据类型。使用音频回调AliAudioObserver接收音频媒体数据。并根据业务场景使用相应的数据源。

AliAudioObserver：音频数据回调。

API	描述	以上版本支持
onCaptureRawData	音频原始数据。	1.17
onCaptureData	本地推流音频数据。	1.17
onRenderData	远端音频数据。	1.17

AliAudioType：音频类型枚举。

枚举名	描述
PUB_OBSERVER	经过音频3A处理后的音频数据。
SUB_OBSERVER	当前订阅到的远端用户混音后的音频数据。
RAW_DATA_OBSERVER	本地采集的原始音频数据。

语音数据处理

RTC获取音频数据方式如下：

1. 应用在创建AliRtcEngine引擎实例之后，注册音频数据回调。

   > **说明** 指定参数AliAudioType输出指定的音频数据类型（类型参见下一步骤说明）。

```
AliRtcEngine mAliRtcEngine = AliRtcEngine.getInstance(getApplicationContext());
// 开始本地采集音频数据输出(type输出其他类型参见枚举AliAudioType)
mAliRtcEngine.registerAudioObserver(type,
        new AliRtcEngine.AliAudioObserver() {
          @Override
          public void onCaptureRawData(long dataPtr, int numSamples, int bytesPerSample, int numCh
annels, int sampleRate, int samplesPerSec) {
          }
          @Override
          public void onCaptureData(long dataPtr, int numSamples, int bytesPerSample, int numChann
els, int sampleRate, int samplesPerSec) {
          }
          @Override
          public void onRenderData(long dataPtr, int numSamples, int bytesPerSample, int numChanne
ls, int sampleRate, int samplesPerSec) {
          }
        });
// 开始音频推流及订阅其他用户视频
.....
```

   > **说明**
   >
   > - 本地采集音频数据需要在开启麦克风（开始推送音频流）前提下才会有回调输出，远端视频数据同样需要在订阅其他用户的音频流成功后才会有回调输出。
   > - 音频类型枚举AliAudioType中的VOLUME_DATA_OBSERVER类型为注册用户音量值回调，目前已经废弃，从1.16版本开始请您使用registerAudioVolumeObserver接口。

2. 注册音频数据回调，通过AliAudioType参数指明当前回调音频数据类型。

   > **说明** 如果需要输出多种类型数据，需要分别调用注册。注册完成后，音频数据将通过AliAudioObserver持续回调。

```
// AliAudioObserver接口数据回调
public interface AliAudioObserver {
/* @param dataPtr 音频数据
@param numSamples 采样数
@param bytesPerSample 采样位数（字节）
@param numChannels 声道数
@param sampleRate 采样率
@param samplesPerSec 采样位数（字节）
*/
void onCaptureRawData(long dataPtr, int numSamples, int bytesPerSample, int numChannels, int sampleRate, int samplesPerSec);
void onCaptureData(long dataPtr, int numSamples, int bytesPerSample, int numChannels, int sampleRate, int samplesPerSec);
void onRenderData(long dataPtr, int numSamples, int bytesPerSample, int numChannels, int sampleRate, int samplesPerSec);
}
```

3. 需要停止接收视频裸数据时，调用接口unRegisterAudioObserver关闭音频数据输出。

> **说明** 停止类型需要对应步骤一中注册输出的音频数据类型。

```
// 停止本地采集音频数据输出，停止类型需要对应步骤2中注册输出的音频数据类型AliAudioType
pEngine->unRegisterAudioObserver(type);
```

语音服务操作，详细请参见：智能语音交互。

21.3. Windows

通过本文，您可以了解Windows端语音数据处理的功能介绍。

场景

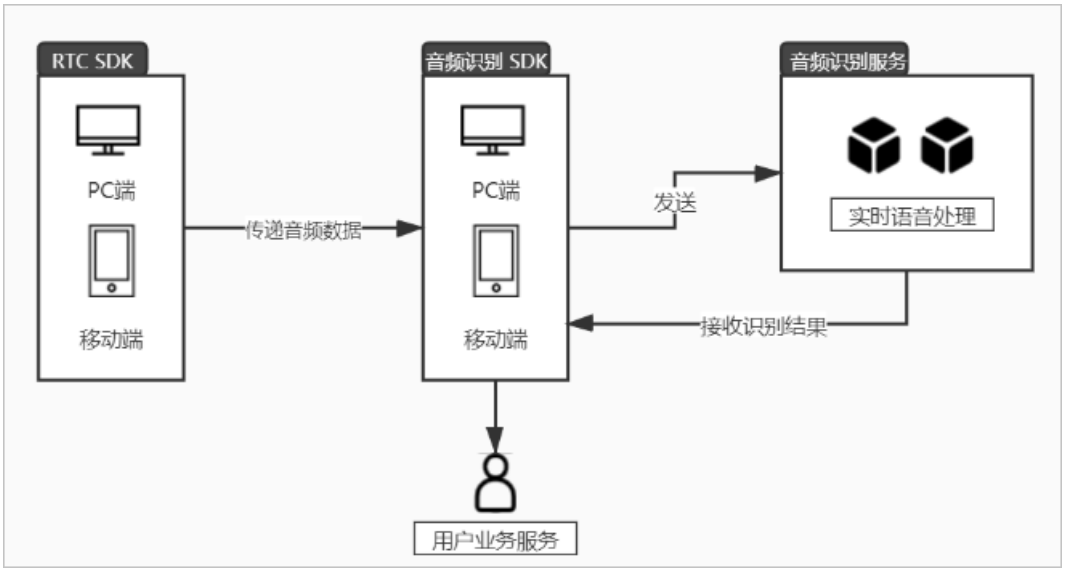
语音识别

阿里云的语音识别指的是将本地发布端或订阅端的音频数据转化为文字，实现流程如下：

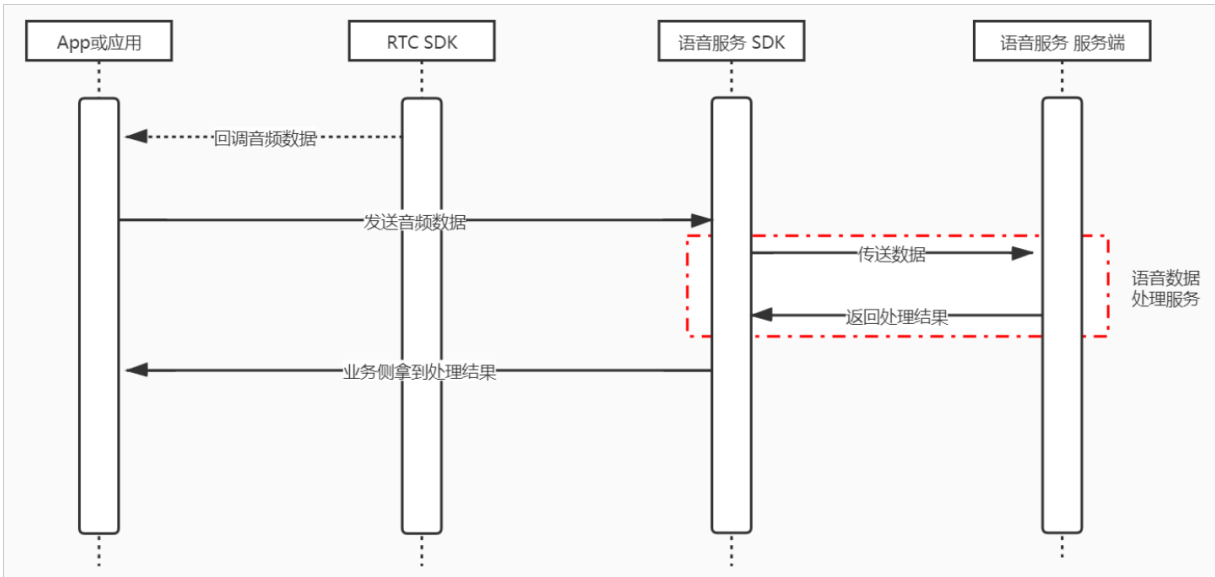
1. 阿里云RTC会将音频数据发送至音频识别SDK中。
2. 音频识别SDK将音频数据发送至音频识别服务进行实时语音处理并返回识别结果。
3. 音频识别SDK为用户提供识别结果。

详细请参见智能语音交互。

方案架构图



调用时序图



接口及使用

通过继承AliRtcEventListener回调类，实现onAudioSampleCallback回调接收音频媒体数据，并根据业务场景使用相应的数据源。

onAudioSampleCallback：订阅的音频数据回调。

参数	类型	描述
type	AliRtcAudioSource	音频数据源类型。
audioSample	AliRtcAudioDataSample *	音频数据。

AliRtcAudioSource：音频裸数据源类型。

枚举名	描述
AliRtcAudiosourceRawData	本地采集的原始音频数据。
AliRtcAudiosourcePub	经过音频3A处理后的音频数据。
AliRtcAudiosourceSub	当前订阅到的远端用户混音后的音频数据。

语音数据处理

RTC获取音频数据方式如下：

1. 应用在创建AliRtcEngine引擎实例之后，注册音频数据回调。

   > **说明** 指定参数AliAudioType输出指定的音频数据类型（类型参见下一步骤说明）。

   ```
   AliRtcEngine mAliRtcEngine = AliRtcEngine.getInstance(getApplicationContext());
   // 开始本地采集音频数据输出(type输出其他类型参见枚举AliAudioType)
   mAliRtcEngine.registerAudioObserver(type,
           new AliRtcEngine.AliAudioObserver() {
             @Override
             public void onCaptureRawData(long dataPtr, int numSamples, int bytesPerSample, int numCh
   annels, int sampleRate, int samplesPerSec) {
             }
             @Override
             public void onCaptureData(long dataPtr, int numSamples, int bytesPerSample, int numChann
   els, int sampleRate, int samplesPerSec) {
             }
             @Override
             public void onRenderData(long dataPtr, int numSamples, int bytesPerSample, int numChanne
   ls, int sampleRate, int samplesPerSec) {
             }
           });
   // 开始音频推流及订阅其他用户视频
   .....
   ```

   > **说明**
   > - 本地采集音频数据需要在开启麦克风（开始推送音频流）前提下才会有回调输出，远端视频数据同样需要在订阅其他用户的音频流成功后才会有回调输出。
   > - 音频类型枚举AliRtcAudioSource中的AliRtcAudioSourceVolume类型为注册用户音量值回调，注册订阅该类型时，用户音量将通过onAudioVolumeCallback回调返回，与音频数据输出回调不同。

2. 注册音频数据回调，通过audioSample参数指明当前回调音频数据类型。

   > **说明** 如果需要输出多种类型数据，需要分别调用注册。注册完成后，音频数据将通过AliAudioObserver持续回调。

   音频数据通过回调中audioSample参数返回，数据格式为PCM数据，目前SDK支持输出不同环节的音频数据，回调中通过type参数指定当前回调音频数据类型。

```
// AliRtcAudioDataSample音频媒体数据结构
typedef struct tagAliRtcAudioDataSample { unsigned char *data{nullptr}; //音频数据
int numOfSamples{0}; //采样数
int bytesPerSample{0}; //采样位数（字节）
int numOfChannels{0}; //声道数
int samplesPerSec{0}; //采样率
// 接收并处理音频数据回调
void void onAudioSampleCallback(AliRtcAudioSource type, AliRtcAudioDataSample *audioSample) {};
```

3. 需要停止接收视频裸数据时，调用接口unsubscribeAudioDat a关闭音频数据输出。

> **说明** 停止类型需要对应步骤一中注册输出的音频数据类型。

```
// 停止本地采集音频数据输出
pEngine->unsubscribeAudioData(AliRtcAudiosourceRawData);
```

语音服务操作，详细请参见：智能语音交互。

22.extras参数配置说明

本文为您介绍extras参数配置的说明、使用场景和实现方式。

功能说明

extras字符配置的功能列表如下表所示。

参数	功能描述	取值	适用平台
user_specified_aec	音频3A：回声消除	TRUE：开启，FALSE：关闭	Android、iOS、Mac、Windows
user_specified_ans	音频3A：自动降噪	TRUE：开启，FALSE：关闭	Android、iOS、Mac、Windows
user_specified_agc	音频3A：自动增益	TRUE：开启，FALSE：关闭	Android、iOS、Mac、Windows
user_specified_engine_mode	音质模式	TRUE：开启，FALSE：关闭	Android、iOS、Mac、Windows
user_specified_scene_mode	场景模式	TRUE：开启，FALSE：关闭	Android、iOS、Mac、Windows
user_specified_video_preprocess	使用三方美颜处理视频	TRUE：开启，FALSE：关闭	Android、iOS
user_specified_groupid	1对1模式	TRUE：开启，FALSE：关闭	Android、iOS、Mac、Windows

音频3A

使用场景

- 当移动端（Android和iOS）硬件效果不满足要求时，可以将这三个开关均设置为TRUE，表示启用阿里云RTC提供的软件音频处理算法。能达到效果与音乐模式或媒体模式一样。
- 当PC端或移动端有外接声卡设备，而且声卡设备自带音频信号处理功能时，为保真音质，建议手动将相关功能设置为FLASE。
- 如果不设置任何3A参数，即表示不激活3A开关的功能，系统会按照当前选择的模式（默认模式，音乐模式等）运行。

实现方式

```
JsonObject jsonObject = new JsonObject();
jsonObject.put("user_specified_aec","TRUE");
jsonObject.put("user_specified_ans","TRUE");
jsonObject.put("user_specified_agc","TRUE");
AliRtcEngine engine= AliRtcEngine.getInstance(context, jsonObject.toString());
```

音质模式与场景模式

功能说明

- 在一些比较专业的场景里，用户对声音的效果需求很高。阿里云RTC提供多种组合方案。
- 开发者根据对音质、场景等的不同需求，自由定制不同的音频属性，获得最佳实时互动效果。

模式说明

- 音质模式

音质模式值列举	模式名称	声道数	采样率	编码码率
ENGINE_LOW_QUALITY_MODE	低音质模式	1	8 kHz	12 Kbps
ENGINE_BASIC_QUALITY_MODE	标准音质模式	1	16 kHz	24 Kbps
ENGINE_HIGH_QUALITY_MODE	高音质模式	1	48 kHz	48 Kbps

- 场景模式

场景模式值列举	场景名称	特性
SCENE_DEFAULT_MODE	默认场景	一般的音视频通信场景推荐使用。
SCENE_EDUCATION_MODE	教育场景	优先保证音频连续性与稳定性。
SCENE_MEDIA_MODE	媒体场景	保真人声与音乐音质，连麦直播间推荐使用。
SCENE_MUSIC_MODE	音乐场景	高保真音乐音质，乐器教学等对音乐音质有要求的场景推荐使用。

场景搭配示例

场景	音质模式	场景模式	特性
普通语音聊天室	ENGINE_BASIC_QUALITY_MODE	SCENE_DEFAULT_MODE	音质较好，优先保证通话质量，传输流畅。适用于对音质没有极致追求的场景。
语音教学小班课	ENGINE_HIGH_QUALITY_MODE	SCENE_DEFAULT_MODE	音质高清，优先保证通话质量，传输流畅。适用于对语音音质有极致追求的场景。
乐器教学小班课	ENGINE_HIGH_QUALITY_MODE	SCENE_MUSIC_MODE	音质高清，优先音乐质量，传输流畅。适用于对音乐音质有极致追求的场景。
直播连麦（语聊）	ENGINE_HIGH_QUALITY_MODE	SCENE_MEDIA_MODE	传输流畅、音质高清，保证语音质量的同时，兼顾音乐音质。适用于既有语音又有音乐的聊天场景。

场景	音质模式	场景模式	特性
直播连麦（唱歌）	ENGINE_HIGH_QUALITY_MODE	SCENE_MUSIC_MODE	传输流畅、音质高清，优先音乐质量，配合提供的各种音效。适用于唱歌乐器弹奏为主的场景。
小型穿戴设备（如电话手表）	ENGINE_LOW_QUALITY_MODE	SCENE_DEFAULT_MODE	传输流畅、音质较好，优先保证语音可听可懂，功耗低。

实现方式

```
JsonObject jsonObject = new JsonObject();
//开启音乐场景下高音质模式
jsonObject.put("user_specified_engine_mode","ENGINE_HIGH_QUALITY_MODE");
jsonObject.put("user_specified_scene_mode","SCENE_MUSIC_MODE");
AliRtcEngine engine= AliRtcEngine.getInstance(context, jsonObject.toString());
```

使用三方美颜处理视频

功能说明

- 在接入前，我们需要在SDK的实例extra字段中添加开关：user_specified_video_preprocess：TRUE。
- 通常客户集成以后反馈自己能看到美颜，对方看不到，就是因为这个没有设置。

实现方式

```
JsonObject jsonObject = new JsonObject();
jsonObject.put("user_specified_video_preprocess","TRUE");
AliRtcEngine engine= AliRtcEngine.getInstance(context, jsonObject.toString());
```

1对1模式

1对1模式自动推流，只推相机大流，不推次要相机流。

场景描述

随着近些年音视频的应用越来越广泛，衍生了多样的在线教育、实时通信等场景。在教育场景下1对1指导更是广泛使用。下面是一些典型的1对1场景。 比如1对1线上课外辅导、1对1线上语言教学、1对1线上音乐陪练和1对1线上面试等。

实现方式

```
JsonObject jsonObject = new JsonObject();
jsonObject.put("user_specified_groupid","1v1");
AliRtcEngine engine= AliRtcEngine.getInstance(context, jsonObject.toString());
```