云顾问-Tencent RTC 云助手

模块化方案

产品文档

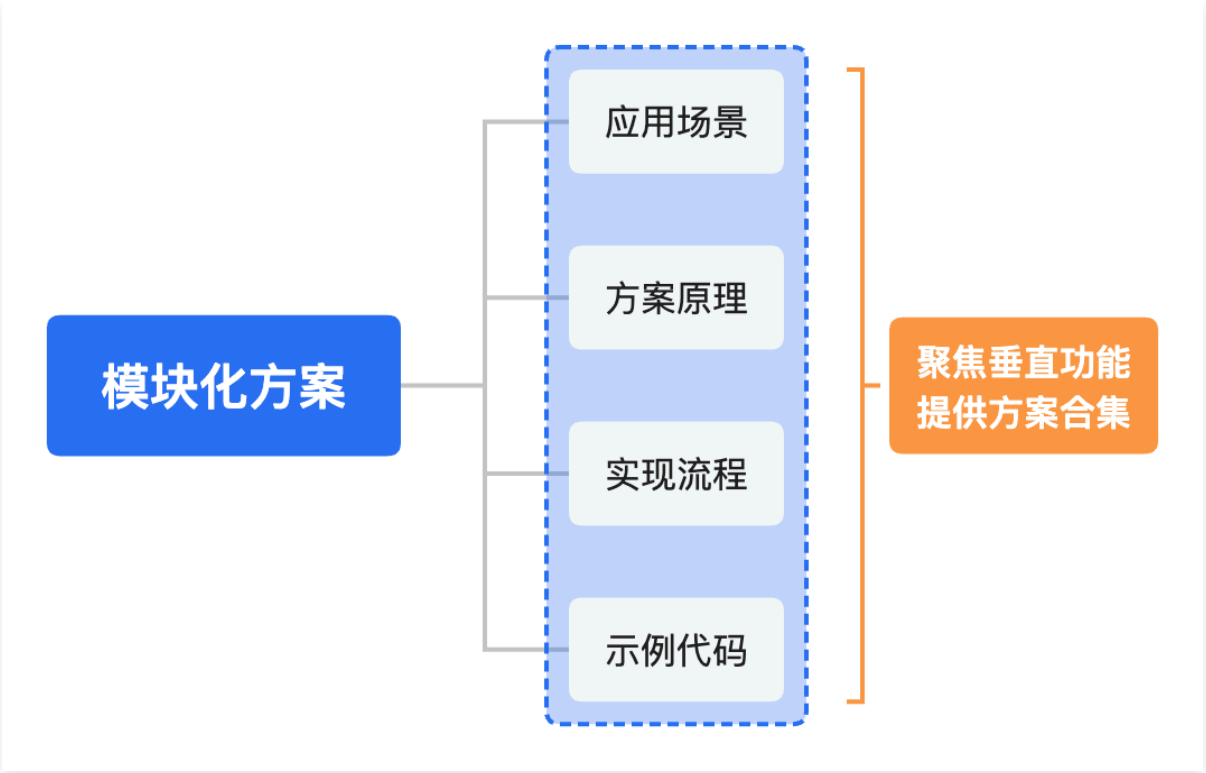
文档目录

模块化方案
模块化方案概述

最近更新时间：2025-11-18 15:15:32

概述

模块化方案独立于场景化方案之外，更加聚焦音视频通信领域的垂直功能，提供对应的解决方案合集。
目前涵盖了 跨房 PK 连麦方案 、AI 对话 IM 信令方案、移动端应用保活方案、网络质量监控、直播上下滑、视频画中画方案，可以帮助您深入理解单个功能模块的应用场景、方案原理和实现流程等，助力业务快速进行方案的选型和落地。



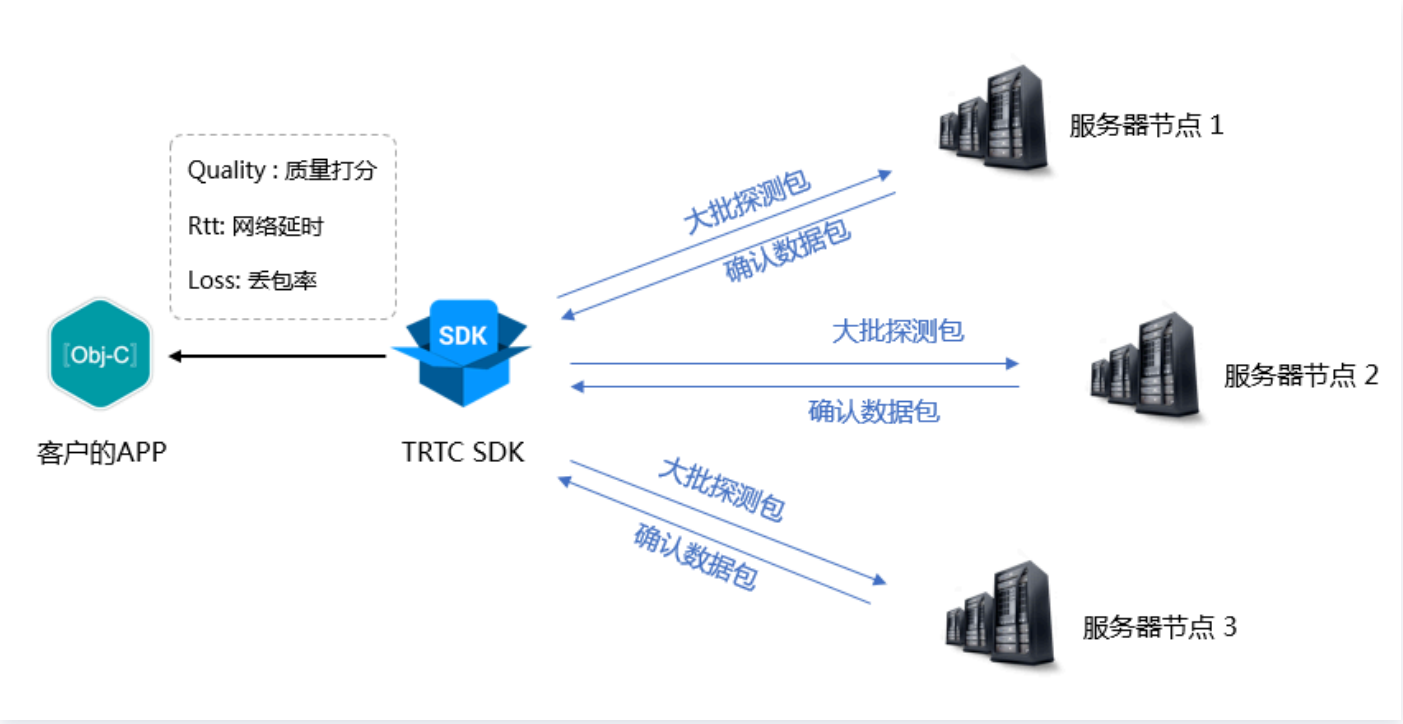
网络质量监控

最近更新时间：2025-11-18 15:15:33

为了让用户更好地感知网络质量，建议业务在通话前进行测速，通话中监测网络质量和连接情况，并提供相应的 UI 界面交互与提示。这样可以方便用户评估当前网络质量，并及时调整网络设置，以达到最佳通话效果。

通话前网络测速

普通用户很难去感知网络情况，建议在通话前进行测速，以便于评估当前网络质量，及时调整网络以达到通话的最优的效果。
测速的原理是 SDK 向服务器节点发送一批探测包，然后统计回包的质量，并将测速的结果通过回调接口进行通知，如下图所示：



通话前测速的优势：

- 测速结果将用于优化 SDK 的服务器选择策略，因此建议在用户首次通话前进行一次测速，以帮助我们选择最佳服务器。
- 如果测试结果非常不理想，您可以通过显眼的 UI 提示用户选择更好的网络环境。

startSpeedTest 测速

Android

```
//启动网络测速的示例代码，需要 sdkAppId 和 UserSig，（获取方式参考基本功能）
// 这里以登录后开始测试为例
```

```
public void onLogin(String userId, String userSig)
{
  TRTCCloudDef.TRTCSpeedTestParams params = new
TRTCCloudDef.TRTCSpeedTestParams();
  params.sdkAppId = GenerateTestUserSig.SDKAPPID;
  params.userId = mEtUserId.getText().toString();
  params.userSig = GenerateTestUserSig.genTestUserSig(params.userId);
  params.expectedUpBandwidth = Integer.parseInt(expectUpBandwidthStr);
  params.expectedDownBandwidth =
Integer.parseInt(expectDownBandwidthStr);
    // sdkAppID 为控制台中获取的实际应用的 AppID
    trtcCloud.startSpeedTest(params);
}

// 监听测速结果，继承 TRTCCloudListener  并实现如下方法
void onSpeedTestResult(TRTCCloudDef.TRTCSpeedTestResult result)
{
    // 测速完成后，会回调出测速结果
}Android
```

iOS

```
// 启动网络测速的示例代码，需要 sdkAppId 和 UserSig，(获取方式参考基本功能)
// 这里以登录后开始测试为例
- (void)onLogin:(NSString *)userId userSig:(NSString *)userSid
{
    TRTCSpeedTestParams *params;
    // sdkAppID 为控制台中获取的实际应用的 AppID
    params.sdkAppID = sdkAppId;
    params.userID = userId;
    params.userSig = userSig;
    // 预期的上行带宽（kbps，取值范围： 10 ～ 5000，为 0 时不测试）
    params.expectedUpBandwidth = 5000;
    // 预期的下行带宽（kbps，取值范围： 10 ～ 5000，为 0 时不测试）
    params.expectedDownBandwidth = 5000;
    [trtcCloud startSpeedTest:params];
}
- (void)onSpeedTestResult:(TRTCSpeedTestResult *)result {
    // 测速完成后，返回测速结果
```

```
}
```

Windows

```
// 启动网络测速的示例代码，需要 sdkAppId 和 UserSig，（获取方式参考基本功能）
// 这里以登录后开始测试为例
void onLogin(const char* userId, const char* userSig)
{
    TRTCSpeedTestParams params;
    // sdkAppID 为控制台中获取的实际应用的 AppID
    params.sdkAppID = sdkAppId;
    params.userId = userid;
    param.userSig = userSig;
    // 预期的上行带宽（kbps，取值范围： 10 ～ 5000，为 0 时不测试）
    param.expectedUpBandwidth = 5000;
    // 预期的下行带宽（kbps，取值范围： 10 ～ 5000，为 0 时不测试）
    param.expectedDownBandwidth = 5000;
    trtcCloud->startSpeedTest(params);
}

// 监听测速结果
void TRTCCloudCallbackImpl::onSpeedTestResult(
            const TRTCSpeedTestResult& result)
{
    // 测速完成后，会回调出测速结果
}
```

测速的结果（TRTCSpeedTestResult）包含如下几个字段：

字段	含义	含义说明
success	是否成功	本次测试是否成功
errMsg	错误信息	带宽测试的详细错误信息
ip	服务器 IP	测速服务器的 IP
quality	网络质量评分	通过评估算法测算出的网络质量，loss 越低，rtt 越小，得分也就越高

upLostRate	上行丢包率	范围是[0 – 1.0]，例如0.3代表每向服务器发送10个数据包，可能有3个会在中途丢失
downLostRate	下行丢包率	范围是[0 – 1.0]，例如0.2代表从服务器每收取10个数据包，可能有2个会在中途丢失
rtt	网络延时	代表 SDK 跟服务器一来一回之间所消耗的时间，这个值越小越好，正常数值在10ms – 100ms之间
availableUpBandwidth	上行带宽	预测的上行带宽，单位为kbps， –1表示无效值
availableDownBandwidth	下行带宽	预测的下行带宽，单位为kbps， –1表示无效值

工具测速

如果您不想通过调用接口的方式进行网络测速，RTC Engine 还提供了桌面端的网络测速工具程序，帮助您快速获取详细的网络质量信息，您可以根据自己平台选择以下程序进行下载。该程序提供了快速测试和持续测试两种测试选择：

- Mac
- Windows

指标	含义
WiFi Quality	Wi-Fi 信号质量
DNS RTT	腾讯云的测速域名解析耗时
MTR	MTR 是一款网络测试工具，能探测客户端到 RTC Engine 节点的丢包率与延时，还可以查看路由中每一跳的具体信息
UDP Loss	客户端到 RTC Engine 节点的 UDP 丢包率
UDP RTT	客户端到 RTC Engine 节点的 UDP 延时
Local RTT	客户端到本地网关的延时
Upload	上行预估带宽

Download	下行预估带宽

> ⚠ 注意：
> - 视频通话期间请勿测试，以免影响通话质量。
> - 测速本身会消耗一定的流量，从而产生极少量额外的流量费用（基本可以忽略）。

通话中网络检测

为了让用户更清楚地感知到通话过程中可能出现的网络波动，建议在通话界面添加相应的 UI 提醒。特别是在网络状况较差时，通过提醒使用户能够感知到变化，并及时调整以改善网络情况。

onNetworkQuality 监听本地网络质量

onNetworkQuality 的回调事件，它会每两秒钟一次向您汇报当前的网络质量，其参数包括 localQuality 和 remoteQuality 两个部分：

- localQuality：代表您当前的网络质量，分为 6 个等级，分别是 Excellent、Good、Poor、Bad、VeryBad 和 Down。
- remoteQuality：代表远端用户的网络质量，这是一个数组，数组中的每个元素代表一个远端用户的网络质量。

Quality	名称	说明
0	Unknown	未检测到
1	Excellent	当前网络非常好
2	Good	当前网络比较好
3	Poor	当前网络一般
4	Bad	当前网络较差，可能会出现明显的卡顿和通话延迟
5	VeryBad	当前网络很差，RTC Engine 只能勉强保持连接，但无法保证通讯质量
6	Down	当前网络不满足 RTC Engine 的最低要求，无法进行正常的音视频通话

通过对 onNetworkQuality 进行监听并在界面上做相应地提示即可：

Android

```
// 监听 onNetworkQuality 回调并感知当前网络状态的变化
@Override
public void onNetworkQuality(TRTCCloudDef.TRTCQuality localQuality,
```

```
                                ArrayList<trtcclouddef.trtcquality>
remoteQuality)
{
    // Get your local network quality
    switch(localQuality) {
        case TRTCQuality_Unknown:
            Log.d(TAG, "SDK has not yet sensed the current network
quality.");
            break;
        case TRTCQuality_Excellent:
            Log.d(TAG, "The current network is very good.");
            break;
        case TRTCQuality_Good:
            Log.d(TAG, "The current network is good.");
            break;
        case TRTCQuality_Poor:
            Log.d(TAG, "The current network quality barely meets the
demand.");
            break;
        case TRTCQuality_Bad:
            Log.d(TAG, "The current network is poor, and there may be
significant freezes and call delays.");
            break;
        case TRTCQuality_VeryBad:
            Log.d(TAG, "The current network is very poor, the
communication quality cannot be guaranteed");
            break;
        case TRTCQuality_Down:
            Log.d(TAG, "The current network does not meet the minimum
requirements.");
            break;
        default:
            break;
    }
    // Get the network quality of remote users
    for (TRTCCloudDef.TRTCQuality info : arrayList) {
        Log.d(TAG, "remote user : = " + info.userId + ", quality = " +
info.quality);
    }
```

```
}
```

iOS&Mac

```
// 监听 onNetworkQuality 回调并感知当前网络状态的变化
- (void)onNetworkQuality:(TRTCQualityInfo *)localQuality remoteQuality:
(NSArray<trtcqualityinfo *=""> *)remoteQuality {
    // Get your local network quality
    switch(localQuality.quality) {
        case TRTCQuality_Unknown:
            NSLog(@"SDK has not yet sensed the current network
quality.");
            break;
        case TRTCQuality_Excellent:
            NSLog(@"The current network is very good.");
            break;
        case TRTCQuality_Good:
            NSLog(@"The current network is good.");
            break;
        case TRTCQuality_Poor:
            NSLog(@"The current network quality barely meets the
demand.");
            break;
        case TRTCQuality_Bad:
            NSLog(@"The current network is poor, and there may be
significant freezes and call delays.");
            break;
        case TRTCQuality_VeryBad:
           NSLog(@"The current network is very poor, the communication
quality cannot be guaranteed");
            break;
        case TRTCQuality_Down:
            NSLog(@"The current network does not meet the minimum
requirements.");
            break;
        default:
            break;
    }
    // Get the network quality of remote users
```

```
    for (TRTCQualityInfo *info in arrayList) {
        NSLog(@"remote user : = %@, quality = %@", info.userId,
@(info.quality));
    }
}

iOSMac
```

Windows

```
// 监听 onNetworkQuality 回调并感知当前网络状态的变化
void onNetworkQuality(liteav::TRTCQualityInfo local_quality,
    liteav::TRTCQualityInfo* remote_quality, uint32_t
remote_quality_count) {
    // Get your local network quality
    switch (local_quality.quality) {
    case TRTCQuality_Unknown:
        printf("SDK has not yet sensed the current network quality.");
        break;
    case TRTCQuality_Excellent:
        printf("The current network is very good.");
        break;
    case TRTCQuality_Good:
        printf("The current network is good.");
        break;
    case TRTCQuality_Poor:
        printf("The current network quality barely meets the demand.");
        break;
    case TRTCQuality_Bad:
        printf("The current network is poor, and there may be
significant freezes and call delays.");
        break;
    case TRTCQuality_Vbad:
        printf("The current network is very poor, the communication
quality cannot be guaranteed");
        break;
    case TRTCQuality_Down:
        printf("The current network does not meet the minimum
requirements.");
```

```
        break;
    default:
        break;
    }
    // Get the network quality of remote users
    for (int i = 0; i < remote_quality_count; ++i) {
        printf("remote user : = %s, quality = %d",
remote_quality[i].userId, remote_quality[i].quality);
    }
}
```

onStatistics 监听完整网络质量

onStatistics 统计回调事件，它会每间隔2秒抛出一次，用于通知 SDK 内部音频、视频以及网络相关的专业技术指标，这些信息在 TRTCStatistics 均有罗列：

- 视频统计信息：视频的分辨率（resolution）、帧率（FPS）和比特率（bitrate）等信息。
- 音频统计信息：音频的采样率（samplerate）、声道（channel）和比特率（bitrate）等信息。
- 网络统计信息：SDK 和云端一次往返（SDK > Cloud > SDK）的网络耗时（rtt）、丢包率（loss）、上行流量（sentBytes）和下行流量（receivedBytes）等信息。

枚举类型	描述
appCpu	当前应用的 CPU 使用率，单位 (%)，Android 8.0 以上不支持。
downLoss	从云端到 SDK 的下行丢包率，单位 (%)。 • 该数值越小越好，如果 downLoss 为 0%，则意味着下行链路的网络质量很好，从云端接收的数据包基本不发生丢失。 • 如果 downLoss 为 30%，则意味着云端向 SDK 传输的音视频数据包中，会有 30% 丢失在传输链路中。
gatewayRtt	从 SDK 到本地路由器的往返时延，单位ms。该数值代表从 SDK 发送一个网络包到本地路由器网关，再从网关回送一个网络包到 SDK 的总计耗时，也就是一个网络包经历“SDK > 网关 > SDK”的总耗时。 • 该数值越小越好：如果 gatewayRtt < 50ms，意味着较低的音视频通话延迟；如果 gatewayRtt > 200ms，则意味着较高的音视频通话延迟。 • 当网络类型为蜂窝网时，该值无效。
localArray	本地的音视频统计信息。 由于本地可能有三路音视频流（即高清大画面，低清小画面，以及辅流画面），因此本地的音视频统计信息是一个数组。

receiveBytes	总接收字节数（包含信令数据和音视频数据），单位为字节数（Bytes）。
remoteArray	远端的音视频统计信息。 因为同时可能有多个远端用户，而且每个远端用户同时可能有多路音视频流（即高清大画面，低清小画面，以及辅流画面），因此远端的音视频统计信息是一个数组。
rtt	从 SDK 到云端的往返延时，单位ms。该数值代表从 SDK 发送一个网络包到云端，再从云端回送一个网络包到 SDK 的总计耗时，也就是一个网络包经历“SDK > 云端 > SDK”的总耗时。该数值越小越好：如果 rtt < 50ms，意味着较低的音视频通话延迟；如果 rtt > 200ms，则意味着较高的音视频通话延迟。 **说明：** rtt 代表“SDK > 云端 > SDK”的总耗时，所以不需要区分 upRtt 和 downRtt。
sendBytes	总发送字节数（包含信令数据和音视频数据），单位为字节数（Bytes）。
systemCpu	当前系统的 CPU 使用率，单位 (%)，Android 8.0 以上不支持。
upLoss	从 SDK 到云端的上行丢包率，单位 (%)。 • 该数值越小越好，如果 upLoss 为 0%，则意味着上行链路的网络质量很好，上传到云端的数据包基本不发生丢失。 • 如果 upLoss 为 30%，则意味着 SDK 向云端发送的音视频数据包中，会有 30% 丢失在传输链路中。

Android

```
@Override
public void onStatistics(TRTCStatistics statistics) {
    super.onStatistics(statistics);
    // appCpu使用率
    Log.d(TAG, "appCpu:" + statistics.appCpu);
    // systemCpu使用率
    Log.d(TAG, "systemCpu:" + statistics.systemCpu);
    // rtt SDK 到云端的往返延时
    Log.d(TAG, "rtt:" + statistics.rtt);
    // upLoss 上行丢包率
    Log.d(TAG, "upLoss:" + statistics.upLoss);
    // downLoss 下行丢包率
```

```
    Log.d(TAG, "downLoss:" + statistics.downLoss);
    // gatewayRtt 网关到云端的往返延时
    Log.d(TAG, "gatewayRtt:" + statistics.gatewayRtt);
    // sendBytes 发送字节数
    Log.d(TAG, "sendBytes:" + statistics.sendBytes);
    // receiveBytes 接收字节数
    Log.d(TAG, "receiveBytes:" + statistics.receiveBytes);
    if(statistics.localArray != null) {
        for (int i = 0; i < statistics.localArray.size(); i++) {
            // 本地视频宽度
            Log.d(TAG, "localStatistics width:" +
statistics.localArray.get(i).width);
            // 本地视频高度
            Log.d(TAG, "localStatistics height:" +
statistics.localArray.get(i).height);
            // 本地视频帧率
            Log.d(TAG, "localStatistics frameRate:" +
statistics.localArray.get(i).frameRate);
            // 本地视频码率
            Log.d(TAG, "localStatistics videoBitrate:" +
statistics.localArray.get(i).videoBitrate);
            // 本地音频码率
            Log.d(TAG, "localStatistics audioBitrate:" +
statistics.localArray.get(i).audioBitrate);
            // 本地音频设备采集状态（用于检测音频外设的健康度）0：采集设备状态正常；
1：检测到长时间静音；2：检测到破音；3：检测到声音异常间断。
            Log.d(TAG, "localStatistics audioCaptureState:" +
statistics.localArray.get(i).audioCaptureState);
            // 本地音频采样率
            Log.d(TAG, "localStatistics audioSampleRate:" +
statistics.localArray.get(i).audioSampleRate);
            // 本地流类型
            Log.d(TAG, "localStatistics streamType:" +
statistics.localArray.get(i).streamType);
        }
    }
    if(statistics.remoteArray != null) {
        for (int i = 0; i < statistics.remoteArray.size(); i++) {
            // 远端用户userid
```

```
            Log.d(TAG, "remoteStatistics userId:" +
statistics.remoteArray.get(i).userId);
            // 远端用户流类型
            Log.d(TAG, "remoteStatistics streamType:" +
statistics.remoteArray.get(i).streamType);
            // 远端视频宽度
            Log.d(TAG, "remoteStatistics width:" +
statistics.remoteArray.get(i).width);
            // 远端视频高度
            Log.d(TAG, "remoteStatistics height:" +
statistics.remoteArray.get(i).height);
            // 远端视频帧率
            Log.d(TAG, "remoteStatistics frameRate:" +
statistics.remoteArray.get(i).frameRate);
            // 远端视频码率
            Log.d(TAG, "remoteStatistics videoBitrate:" +
statistics.remoteArray.get(i).videoBitrate);
            // 远端视频卡顿率 单位 (%) 视频播放卡顿率 (videoBlockRate) = 视频
播放的累计卡顿时长 (videoTotalBlockTime) / 视频播放的总时长。
            Log.d(TAG, "remoteStatistics videoBlockRate:" +
statistics.remoteArray.get(i).videoBlockRate);
            // 视频播放的累计卡顿时长，单位 ms
            Log.d(TAG, "remoteStatistics videoTotalBlockTime:" +
statistics.remoteArray.get(i).videoTotalBlockTime);
            // 该路视频流的总丢包率
            // videoPacketLoss 代表该路视频流历经 主播>云端>观众 这样一条完整的
传输链路后，最终在观众端统计到的丢包率。
            // videoPacketLoss 越小越好，丢包率为0即表示该路视频流的所有数据均已
经完整地到达了观众端。
            // 如果出现了 downLoss == 0 但 videoPacketLoss != 0 的情况，说明
该路视频流在 云端>观众 这一段链路上没有出现丢包，但是在 主播>云端 这一段链路上出现了不
可恢复的丢包。
            Log.d(TAG, "remoteStatistics videoPacketLoss:" +
statistics.remoteArray.get(i).videoPacketLoss);
            // 远端音频码率
            Log.d(TAG, "remoteStatistics audioBitrate:" +
statistics.remoteArray.get(i).audioBitrate);
            // 远端音频播放的卡顿率
            Log.d(TAG, "remoteStatistics audioBlockRate:" +
statistics.remoteArray.get(i).audioBlockRate);
```

```
            // 远端音频的采样率
            Log.d(TAG, "remoteStatistics audioSampleRate:" +
statistics.remoteArray.get(i).audioSampleRate);
            // 远端音频丢包率
            Log.d(TAG, "remoteStatistics audioPacketLoss:" +
statistics.remoteArray.get(i).audioPacketLoss);
            // 音频播放的累计卡顿时长
            Log.d(TAG, "remoteStatistics audioTotalBlockTime:" +
statistics.remoteArray.get(i).audioTotalBlockTime);
            // 播放延迟  为了避免网络抖动和网络包乱序导致的声音和画面卡顿，RTC
Engine 会在播放端管理一个播放缓冲区，用于对接收到的网络数据包进行整理，该缓冲区的大小
会根据当前的网络质量进行自适应调整，该缓冲区的大小折算成以毫秒为单位的时间长度，也就是
jitterBufferDelay。
            Log.d(TAG, "remoteStatistics jitterBufferDelay:" +
statistics.remoteArray.get(i).jitterBufferDelay);
            // 端到端延迟，单位 ms
            //point2PointDelay 代表“主播=>云端=>观众”的延迟，更准确地说，它代
表了“采集=>编码=>网络传输=>接收=>缓冲=>解码=>播放”全链路的延迟。
            //point2PointDelay 需要本地和远端的 SDK 均为 8.5 及以上的版本才生
效，若远端用户为 8.5 以前的版本，此数值会一直为0，代表无意义。
            Log.d(TAG, "remoteStatistics point2PointDelay:" +
statistics.remoteArray.get(i).point2PointDelay);
        }
    }
}
```

iOS&Mac

```
- (void)onStatistics:(TRTCStatistics *)statistics {
    // appCpu使用率
    NSLog(@"appCpu: %u", statistics.appCpu);
        // systemCpu使用率
    NSLog(@"systemCpu: %u", statistics.systemCpu);
    // rtt SDK 到云端的往返延时
    NSLog(@"rtt: %d", statistics.rtt);
    // upLoss 上行丢包率
    NSLog(@"upLoss: %u", statistics.upLoss);
    // downLoss 下行丢包率
```

```
    NSLog(@"downLoss: %u", statistics.downLoss);
    // gatewayRtt 网关到云端的往返延时
    NSLog(@"gatewayRtt: %d", statistics.gatewayRtt);
    // sendBytes 发送字节数
    NSLog(@"sendBytes: %llu", (unsigned long long)statistics.sentBytes);
    // receiveBytes 接收字节数
    NSLog(@"receiveBytes: %llu", (unsigned long
long)statistics.receivedBytes);

    if (statistics.localStatistics != nil) {
        for (int i = 0; i < statistics.localStatistics.count; i++) {
            // 本地视频宽度
            NSLog(@"localStatistics width: %d",
statistics.localStatistics[i].width);
            // 本地视频高度
            NSLog(@"localStatistics height: %d",
statistics.localStatistics[i].height);
            // 本地视频帧率
            NSLog(@"localStatistics frameRate: %u",
statistics.localStatistics[i].frameRate);
            // 本地视频码率
            NSLog(@"localStatistics videoBitrate: %d",
statistics.localStatistics[i].videoBitrate);
            // 本地音频码率
            NSLog(@"localStatistics audioBitrate: %d",
statistics.localStatistics[i].audioBitrate);
            // 本地音频设备采集状态
            NSLog(@"localStatistics audioCaptureState: %d",
statistics.localStatistics[i].audioCaptureState);
            // 本地音频采样率
            NSLog(@"localStatistics audioSampleRate: %d",
statistics.localStatistics[i].audioSampleRate);
            // 本地流类型
            NSLog(@"localStatistics streamType: %ld",
(long)statistics.localStatistics[i].streamType);
        }
    }

    if (statistics.remoteStatistics != nil) {
        for (int i = 0; i < statistics.remoteStatistics.count; i++) {
```

```
            // 远端用户userid
            NSLog(@"remoteStatistics userId: %@",
statistics.remoteStatistics[i].userId);
            // 远端用户流类型
            NSLog(@"remoteStatistics streamType: %ld",
(long)statistics.remoteStatistics[i].streamType);
            // 远端视频宽度
            NSLog(@"remoteStatistics width: %d",
statistics.remoteStatistics[i].width);
            // 远端视频高度
            NSLog(@"remoteStatistics height: %d",
statistics.remoteStatistics[i].height);
            // 远端视频帧率
            NSLog(@"remoteStatistics frameRate: %u",
statistics.remoteStatistics[i].frameRate);
            // 远端视频码率
            NSLog(@"remoteStatistics videoBitrate: %d",
statistics.remoteStatistics[i].videoBitrate);
            // 远端视频卡顿率
            NSLog(@"remoteStatistics videoBlockRate: %u",
statistics.remoteStatistics[i].videoBlockRate);
            // 视频播放的累计卡顿时长
            NSLog(@"remoteStatistics videoTotalBlockTime: %d",
statistics.remoteStatistics[i].videoTotalBlockTime);
            // 该路视频流的总丢包率
            NSLog(@"remoteStatistics videoPacketLoss: %u",
statistics.remoteStatistics[i].videoPacketLoss);
            // 远端音频码率
            NSLog(@"remoteStatistics audioBitrate: %d",
statistics.remoteStatistics[i].audioBitrate);
            // 远端音频播放的卡顿率
            NSLog(@"remoteStatistics audioBlockRate: %u",
statistics.remoteStatistics[i].audioBlockRate);
            // 远端音频的采样率
            NSLog(@"remoteStatistics audioSampleRate: %d",
statistics.remoteStatistics[i].audioSampleRate);
            // 远端音频丢包率
            NSLog(@"remoteStatistics audioPacketLoss: %u",
statistics.remoteStatistics[i].audioPacketLoss);
            // 音频播放的累计卡顿时长
```

```
            NSLog(@"remoteStatistics audioTotalBlockTime: %d",
statistics.remoteStatistics[i].audioTotalBlockTime);
            // 播放延迟
            NSLog(@"remoteStatistics jitterBufferDelay: %d",
statistics.remoteStatistics[i].jitterBufferDelay);
            // 端到端延迟
            NSLog(@"remoteStatistics point2PointDelay: %d",
statistics.remoteStatistics[i].point2PointDelay);
        }
    }

}
```

Windows

```
void onStatistics(const TRTCStatistics& statistics) {
    // appCpu使用率
    printf("appCpu: %f\n", statistics.appCpu);
    // systemCpu使用率
    printf("systemCpu: %f\n", statistics.systemCpu);
    // rtt SDK 到云端的往返延时
    printf("rtt: %d\n", statistics.rtt);
    // upLoss 上行丢包率
    printf("upLoss: %f\n", statistics.upLoss);
    // downLoss 下行丢包率
    printf("downLoss: %f\n", statistics.downLoss);
    // gatewayRtt 网关到云端的往返延时
    printf("gatewayRtt: %d\n", statistics.gatewayRtt);
    // sentBytes 发送字节数
    printf("sentBytes: %lld\n", statistics.sentBytes);
    // receiveBytes 接收字节数
    printf("receivedBytes: %lld\n", statistics.receivedBytes);

    //for (const auto& localStat : statistics.localStatisticsArray) {
    if (statistics.localStatisticsArray != nullptr &&
statistics.localStatisticsArraySize != 0)
    {
        for (int i = 0; i < statistics.localStatisticsArraySize; i++)
```

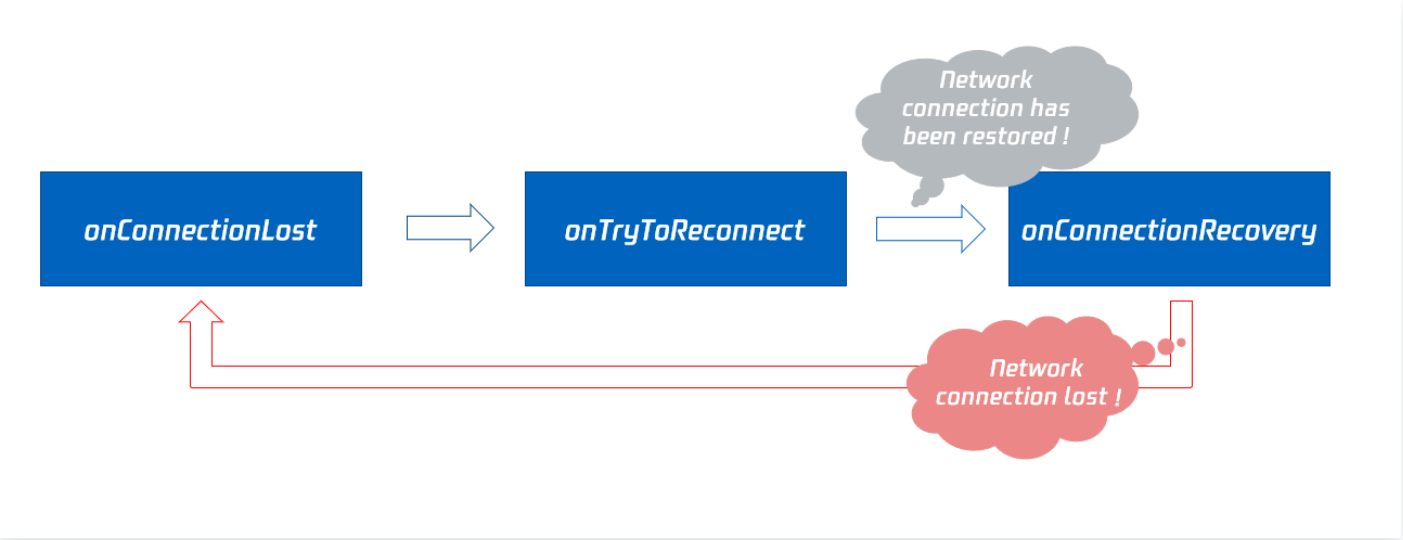
```
        {
            // 本地视频宽度
            printf("localStatistics width: %d\n",
statistics.localStatisticsArray[i].width);
            // 本地视频高度
            printf("localStatistics height: %d\n",
statistics.localStatisticsArray[i].height);
            // 本地视频帧率
            printf("localStatistics frameRate: %f\n",
statistics.localStatisticsArray[i].frameRate);
            // 本地视频码率
            printf("localStatistics videoBitrate: %d\n",
statistics.localStatisticsArray[i].videoBitrate);
            // 本地音频码率
            printf("localStatistics audioBitrate: %d\n",
statistics.localStatisticsArray[i].audioBitrate);
            // 本地音频设备采集状态
            printf("localStatistics audioCaptureState: %d\n",
statistics.localStatisticsArray[i].audioCaptureState);
            // 本地音频采样率
            printf("localStatistics audioSampleRate: %d\n",
statistics.localStatisticsArray[i].audioSampleRate);
            // 本地流类型
            printf("localStatistics streamType: %d\n",
statistics.localStatisticsArray[i].streamType);
        }
    }

    //for (const auto& remoteStat : statistics.remoteStatisticsArray) {
    if (statistics.remoteStatisticsArray != nullptr &&
statistics.remoteStatisticsArraySize != 0)
    {
        for (int i = 0; i < statistics.remoteStatisticsArraySize; i++)
        {
            // 远端用户userid
            printf("remoteStatistics userId: %s\n",
statistics.remoteStatisticsArray[i].userId);
            // 远端用户流类型
            printf("remoteStatistics streamType: %d\n",
statistics.remoteStatisticsArray[i].streamType);
```

```
            // 远端视频宽度
            printf("remoteStatistics width: %d\n",
statistics.remoteStatisticsArray[i].width);
            // 远端视频高度
            printf("remoteStatistics height: %d\n",
statistics.remoteStatisticsArray[i].height);
            // 远端视频帧率
            printf("remoteStatistics frameRate: %f\n",
statistics.remoteStatisticsArray[i].frameRate);
            // 远端视频码率
            printf("remoteStatistics videoBitrate: %d\n",
statistics.remoteStatisticsArray[i].videoBitrate);
            // 远端视频卡顿率
            printf("remoteStatistics videoBlockRate: %f\n",
statistics.remoteStatisticsArray[i].videoBlockRate);
            // 视频播放的累计卡顿时长
            printf("remoteStatistics videoTotalBlockTime: %d\n",
statistics.remoteStatisticsArray[i].videoTotalBlockTime);
            // 该路视频流的总丢包率
            printf("remoteStatistics videoPacketLoss: %f\n",
statistics.remoteStatisticsArray[i].videoPacketLoss);
            // 远端音频码率
            printf("remoteStatistics audioBitrate: %d\n",
statistics.remoteStatisticsArray[i].audioBitrate);
            // 远端音频播放的卡顿率
            printf("remoteStatistics audioBlockRate: %f\n",
statistics.remoteStatisticsArray[i].audioBlockRate);
            // 远端音频的采样率
            printf("remoteStatistics audioSampleRate: %d\n",
statistics.remoteStatisticsArray[i].audioSampleRate);
            // 远端音频丢包率
            printf("remoteStatistics audioPacketLoss: %f\n",
statistics.remoteStatisticsArray[i].audioPacketLoss);
            // 音频播放的累计卡顿时长
            printf("remoteStatistics audioTotalBlockTime: %d\n",
statistics.remoteStatisticsArray[i].audioTotalBlockTime);
            // 播放延迟
            printf("remoteStatistics jitterBufferDelay: %d\n",
statistics.remoteStatisticsArray[i].jitterBufferDelay);
            // 端到端延迟
```

```
            printf("remoteStatistics point2PointDelay: %d\n",
statistics.remoteStatisticsArray[i].point2PointDelay);
        }
    }
}
```

通话中连接监测

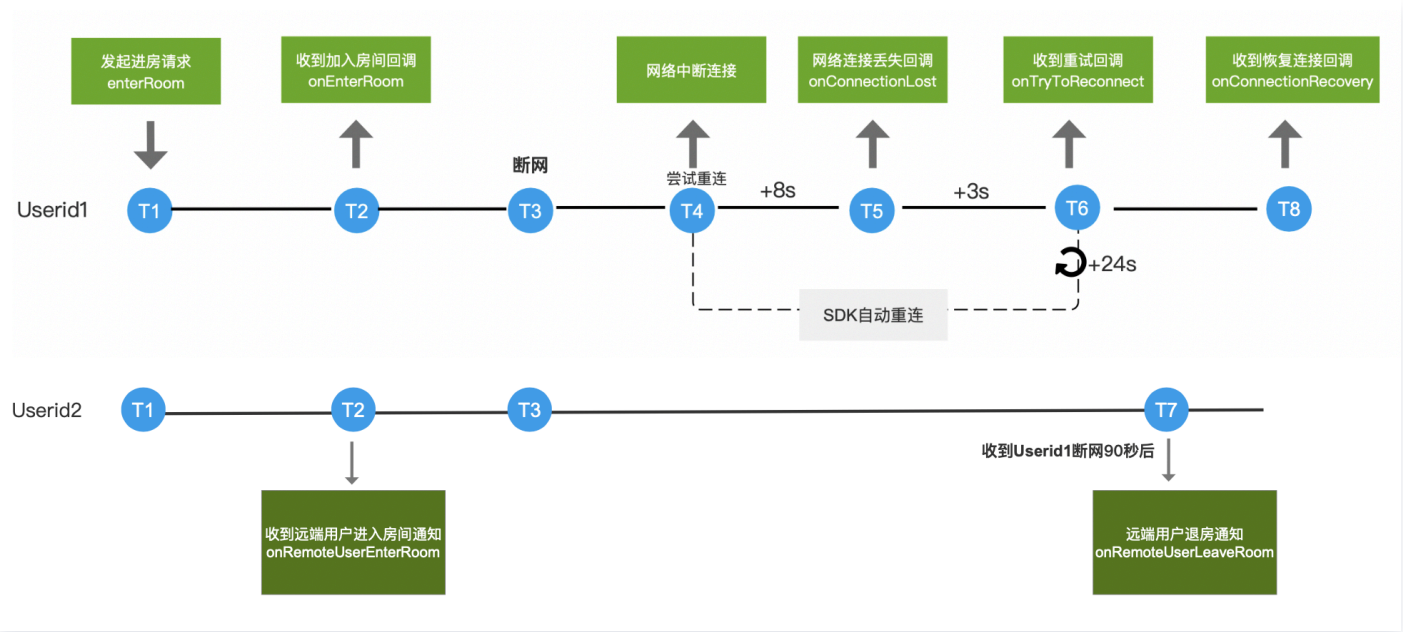
用户在通话过程中，除了要感知网络变化，还需要了解当前 SDK 与后台的连接状态。这样用户可以更准确地判断当前网络是否已经调整完成，以及是否可以正常进行通话。



回调接口	接口说明
onConnectionLost	SDK 与云端的连接已经断开
onTryToReconnect	SDK 正在尝试重新连接到云端
onConnectionRecovery	SDK 与云端的连接已经恢复

断线重连逻辑

SDK 支持用户断线情况下自动重连（若持续30分钟都未重连成功，则自动退房并返回-3301错误码），连接过程中具体的连接状态和处理逻辑如下说明。下图展示了从用户 Userid1 加入频道，到连接中断，再到重新加入房间过程中，收到的监听回调事件：



具体说明：

- T1：用户侧发起调用 `enterRoom` 接口发起进房请求。
- T2：用户 Userid1 收到 `onEnterRoom` 回调，Userid2 感知 Userid1 存在延迟，大约300ms后，Userid2 收到 `onRemoteUserEnterRoom` 回调。
- T3：Userid1 客户端因网络问题断网，SDK 会尝试重新加入房间。
- T4：Userid1 如果连续8秒没有连接上服务端，Userid1 收到 `onConnectionLost` 断连回调。
- T5：Userid1 接着隔3秒没有连接上服务端，Userid1 收到 `onTryToReconnect` 重试回调。
- T6：Userid1 接着每隔24秒，收到 `onTryToReconnect` 重试回调。
- T7：Userid2 会在收到 Userid1 掉线通知90s后，SDK 判断远端用户 Userid1 掉线，Userid2 收到 `onRemoteUserLeaveRoom` 回调。
- T8：如果 Userid1 断连期间任意时刻重连成功，Userid1 收到 `onConnectionRecovery` 恢复回调。

Android

```
@Override
public void onConnectionLost() {
    // 连接断开
    Log.d(TAG, "onConnectionLost");
}

@Override
public void onTryToReconnect() {
    // 尝试重连
    Log.d(TAG, "onTryToReconnect");
```

```
}

@Override
public void onConnectionRecovery() {
    // 重连成功
    Log.d(TAG, "onConnectionRecovery");
}
```

iOS&Mac

```
- (void)onConnectionLost {
    // 连接断开
    NSLog(@"onConnectionLost");
}

- (void)onTryToReconnect {
    // 尝试重连
    NSLog(@"onTryToReconnect");
}

- (void)onConnectionRecovery {
    // 重连成功
    NSLog(@"onConnectionRecovery");
}
```

Windows

```
void onConnectionLost() {
    // 连接断开
    printf("onConnectionLost\n");
}

void onTryToReconnect() {
    // 尝试重连
    printf("onTryToReconnect\n");
}
```

```
void onConnectionRecovery() {
    // 重连成功
    printf("onConnectionRecovery\n");
}
```

移动端应用保活方案

最近更新时间：2025-11-18 15:15:33

对于涉及音视频采集和播放的移动端应用，通常需要进行额外的保活处理，否则应用在后台运行时会受到一定的功能性限制，甚至在后台运行一段时间后被系统强制终止运行。下面我们分别介绍 Android 应用保活方案，以及 iOS 应用保活方案。

Android 应用保活方案

目前 Android 端常用的保活方式是启动前台服务。前台服务是一种特殊的服务，它在运行时会显示一个持续的通知，以告知用户该服务正在运行。由于前台服务的通知是持续可见的，系统会认为这是一个高优先级的任务，应用可以在后台持续运行，而不容易被系统终止。下面介绍前台服务的实现方式及步骤。

步骤1：声明权限

在 AndroidManifest.xml 文件中添加以下权限声明。

```
<!-- 允许应用使用前台服务 -->
<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />

<!-- 如果应用需要在前台服务中使用摄像头 -->
<uses-permission
android:name="android.permission.FOREGROUND_SERVICE_CAMERA" />

<!-- 如果应用需要在前台服务中使用麦克风 -->
<uses-permission
android:name="android.permission.FOREGROUND_SERVICE_MICROPHONE" />

<!-- 允许前台服务发送通知 -->
<uses-permission android:name="android.permission.POST_NOTIFICATIONS"/>
```

⚠ 注意：

- 若您的项目设置 targetSdkVersion 34及以上，且需要在前台服务中使用摄像头和麦克风，则 `FOREGROUND_SERVICE_CAMERA` 和 `FOREGROUND_SERVICE_MICROPHONE` 权限声明是必须的。
- 在 Android 13及以上，如果想让前台服务显示在通知栏上，则需要声明 `POST_NOTIFICATIONS` 权限。

步骤2：创建服务类

创建一个继承自 Service 的类，并在其中实现前台服务的逻辑。

```
public class MyForegroundService extends Service {
    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    @Override
    public void onCreate() {
        super.onCreate();
        // 创建通知渠道
        Notification notification = createNotification();
        // 处理服务启动逻辑
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
            startForeground(1024, notification,
ServiceInfo.FOREGROUND_SERVICE_TYPE_MICROPHONE);
        } else {
            startForeground(1024, notification);
        }
    }

    private Notification createNotification() {
        String CHANNEL_ONE_ID = "CHANNEL_ONE_ID";
        String CHANNEL_ONE_NAME = "CHANNEL_ONE_ID";
        NotificationChannel notificationChannel;
        //进行8.0的判断
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            notificationChannel = new
NotificationChannel(CHANNEL_ONE_ID,
                    CHANNEL_ONE_NAME,
NotificationManager.IMPORTANCE_HIGH);
            notificationChannel.enableLights(true);
            notificationChannel.setLightColor(Color.RED);
            notificationChannel.setShowBadge(true);

notificationChannel.setLockscreenVisibility(Notification.VISIBILITY_PUBL
IC);
```

```
            NotificationManager manager = (NotificationManager)
getSystemService(NOTIFICATION_SERVICE);
            if (manager != null) {
                manager.createNotificationChannel(notificationChannel);
            }
        }
        // 设置点击通知栏回到应用，可选
        Intent intent = new Intent(this, MainActivity.class);
        ActivityOptions options = null;
        if (android.os.Build.VERSION.SDK_INT >=
android.os.Build.VERSION_CODES.M) {
            options = ActivityOptions.makeBasic();
        }
        if (Build.VERSION.SDK_INT >=
Build.VERSION_CODES.UPSIDE_DOWN_CAKE) {

options.setPendingIntentBackgroundActivityStartMode(ActivityOptions.MODE
_BACKGROUND_ACTIVITY_START_ALLOWED);
        }
        PendingIntent pendingIntent;
        if (options != null) {
            pendingIntent = PendingIntent.getActivity(this, 0, intent,
PendingIntent.FLAG_IMMUTABLE, options.toBundle());
        } else {
            pendingIntent = PendingIntent.getActivity(this, 0, intent,
PendingIntent.FLAG_IMMUTABLE);
        }

        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            Notification notification = new Notification.Builder(this,
CHANNEL_ONE_ID).setChannelId(CHANNEL_ONE_ID)
                    .setSmallIcon(R.mipmap.videocall_float_logo)
                    .setContentTitle("这是一个测试标题")
                    .setContentIntent(pendingIntent)
                    .setContentText("这是一个测试内容")
                    .build();
            notification.flags |= Notification.FLAG_NO_CLEAR;
            return notification;
        }else {
```

```
            NotificationCompat.Builder builder = new
NotificationCompat.Builder(this, CHANNEL_ONE_ID)
                    .setSmallIcon(R.mipmap.videocall_float_logo)
                    .setContentTitle("这是一个测试标题")
                    .setContentText("这是一个测试内容")
                    .setContentIntent(pendingIntent)
                    .setPriority(NotificationCompat.PRIORITY_DEFAULT);
            return builder.build();
        }
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        // 停止前台服务
        stopForeground(true);
    }
}
```

步骤3：声明服务

在 AndroidManifest.xml 文件中声明服务。

```
<service
    android:name=".MyForegroundService"
    android:enabled="true"
    android:exported="false"

android:foregroundServiceType="mediaPlayback|mediaProjection|microphone|
camera" />
```

注意：

您可以通过 `android:foregroundServiceType` 属性指定前台服务需要使用的服务类型，以确保后台可以保持正常的服务功能。

- `mediaPlayback` 服务用于媒体播放。
- `mediaProjection` 服务用于媒体投影。
- `microphone` 服务用于使用麦克风。
- `camera` 服务用于使用摄像头。

步骤4：启动前台服务

在需要启动前台服务的地方，按需启动前台服务。

```
NotificationManagerCompat notificationManager =
NotificationManagerCompat.from(this);
boolean areNotificationsEnabled =
notificationManager.areNotificationsEnabled();
if (!areNotificationsEnabled) {
    // 提示用户启用通知权限
    Toast.makeText(this, "请启用通知权限以确保服务正常运行",
Toast.LENGTH_LONG).show();
    // 引导用户到设置页面
    Intent intent = new
Intent(Settings.ACTION_APP_NOTIFICATION_SETTINGS)
            .putExtra(Settings.EXTRA_APP_PACKAGE, getPackageName());
    startActivity(intent);
} else {
    // 启动前台服务
    Intent serviceIntent = new Intent(this, MyForegroundService.class);
    ContextCompat.startForegroundService(this, serviceIntent);
}
```

> ⚠ **注意：**
> 为了确保前台服务能够正常运行，建议在启动前台服务之前，检查通知权限是否被禁用。如果被禁用，可以提示用户启用通知权限。

步骤5：停止前台服务

可从外部组件（例如 Activity 或 BroadcastReceiver）中停止前台服务。

```
// 创建服务的Intent
Intent serviceIntent = new Intent(this, MyForegroundService.class);

// 停止服务
stopService(serviceIntent);
```

在大部分移动设备上，针对启动前台服务的应用，用户在最近应用列表里划卡强杀，前台服务会同时终止，应用会完全停止运行。但在部分境外品牌移动设备上（例如 Google Pixel 系列和 SAMSUNG A 系列），划卡强杀后应用并

不会完全停止运行，前台服务仍然处于活跃状态，这会导致用户还能够听到媒体播放。
针对此类设备，可以通过实现以下两种方案，从而避免应用被强杀后仍然播放媒体声音的现象。

方案一：在服务声明中定义 stopWithTask 属性

添加 `android:stopWithTask="true"` 属性值，服务会在任务被移除时立即停止。

```
<service
    android:name=".MyForegroundService"
    android:enabled="true"
    android:exported="false"
    android:stopWithTask="true"
    android:foregroundServiceType="mediaPlayback|microphone" />
```

方案二：在 Service 层监听 onTaskRemoved 回调

监听 onTaskRemoved，当任务被移除时，该方法会被回调，您可以在这里执行清理操作或保存数据的逻辑。

```
@Override
public void onTaskRemoved(Intent rootIntent) {
    super.onTaskRemoved(rootIntent);
    // 例如在这里执行 RTC Engine 退房，避免继续进行音频采集和播放
    TRTCCloud mTRTCCloud = TRTCCloud.sharedInstance(this);
    mTRTCCloud.exitRoom();
}
```

> ⚠ 注意：
> 以上两种方案任选其一即可，当设置 `android:stopWithTask="true"` 之后，`onTaskRemoved` 方法将不会被回调。

iOS 应用保活方案

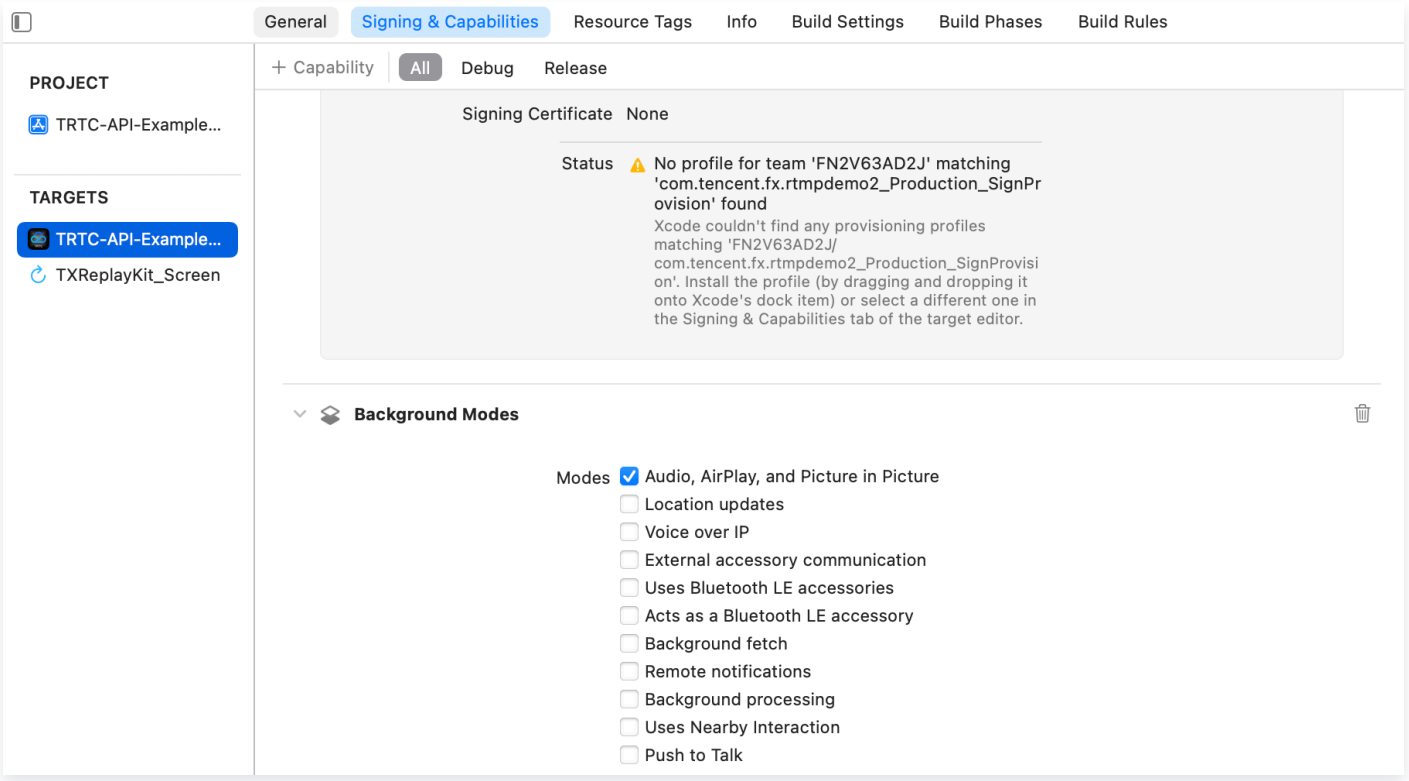
苹果对应用在后台的行为有严格的限制，以保护用户的隐私和设备的电池寿命。通常可以通过启用特定的后台模式（Background Modes），从而在一定程度上实现应用保活，允许应用在后台播放音频或视频。

启用后台模式

Xcode 启用后台模式（Background Modes）的步骤如下：

1. 打开您的 Xcode 项目。
2. 选择您的项目文件（通常在项目导航器的顶部）。

3. 在项目文件中，选择您的目标（Targets）。
4. 在目标设置中，选择 Signing & Capabilities 选项卡。
5. 找到 Background Modes 选项，勾选 Audio, AirPlay, and Picture in Picture 模式。



常见问题

1. 在语音通话或直播场景中，主播将应用退至后台或锁屏，插拔耳机导致音频采集和播放无声

   在 SDK 默认的音频策略下，系统音量类型处于自动切换模式，即“麦上通话，麦下媒体”。同时音量类型也会随音频路由的变化而变化，例如插入耳机音量类型会由通话音量切换至媒体音量。因此，在自动切换模式下，主播插拔耳机会导致系统音量类型的切换，此时系统需要重启音频驱动。而 iOS 系统在后台或锁屏状态下重启音频驱动有概率会失败，所以会导致音频采集和播放无声。
   针对这类问题，可以通过固定系统音量类型来规避，例如指定全程通话音量或全程媒体音量。

   ```
   // 指定全程通话音量
   [self.trtcCloud setSystemVolumeType:TRTCSystemVolumeTypeVOIP];

   // 指定全程媒体音量
   [self.trtcCloud setSystemVolumeType:TRTCSystemVolumeTypeMedia];
   ```

2. 在视频通话或直播场景中，主播将应用退至后台或锁屏，远端观众拉流画面黑屏但声音正常

   苹果系统严格禁止应用在后台采集视频。即使您启用了后台模式，应用在进入后台后，摄像头仍然会自动停止工作。这是为了保护用户的隐私，防止应用在未经用户同意的情况下录制视频。因此，这类场景下的视频采集问题

暂时无法避免，只能实现音频的正常采集和播放。

3. 观众进房时房间内无人推流，将应用退至后台或锁屏，后续房间内有人推流也无法正常接收

   iOS 应用在切换至后台之前，如果没有启动 AudioUnit 采集或播放，则很快会被挂起，且无法人为唤醒，直到切换至前台。要解决此类问题，只需要在应用切换至后台之前，保持 AudioUnit 一直运行即可（播放静音数据）。具体实现方法可参考下方示例代码。

```
// 进房之后启用自定义音轨
[self.trtcCloud enableMixExternalAudioFrame:NO playout:YES];

// 退房之前关闭自定义音轨
[self.trtcCloud enableMixExternalAudioFrame:NO playout:NO];
```

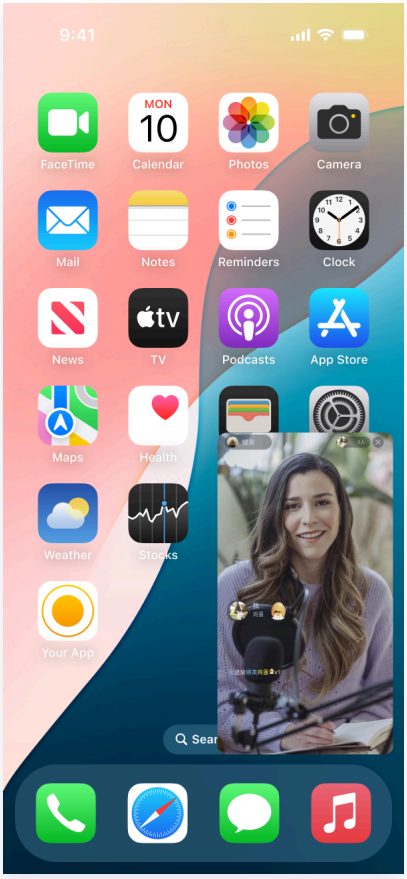
视频画中画 应用保活方案

通过视频画中画功能，可以确保应用视频始终在前台播放，同时实现应用保活。具体实现方法请参考 视频画中画方案 。

视频画中画方案

最近更新时间：2025-11-18 15:15:33

在互动直播等视频场景中，移动端设备观众在长时间观看主播画面时，存在需要临时操作其他 APP 的场景，此时如果能够不中断主播的画面，同时操作其他 APP，会给观众带来更好的用户体验。视频画中画就是针对此场景的解决方案，实现效果如下图。本文将分别对 iOS、Android 和 Flutter 端画中画的实现进行介绍。

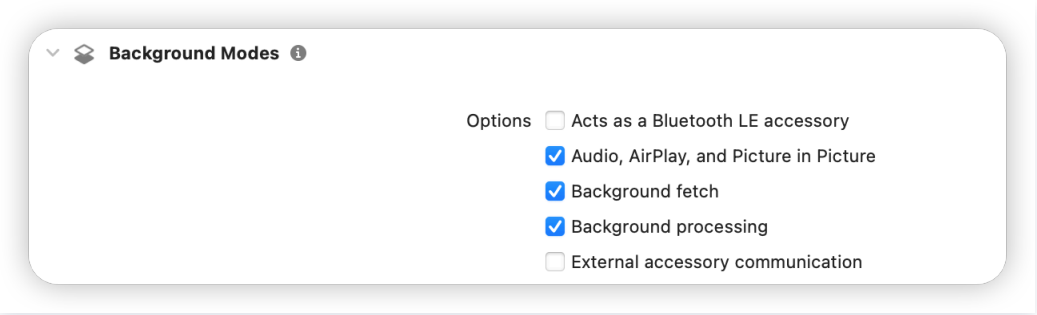


画中画是依赖 iOS 和 Android 提供的系统能力实现的，可分为主播端（需要采集摄像头和上行数据）和观众端（仅需下行数据） 。由于 iOS 系统对权限管控比较严格，因此 iOS 端只提供观众端画中画实现，Android 端提供主播和观众端的画中画实现。对于视频播放，一般有使用 RTC Engine 播放和直播播放两种方案，画中画方案中也分别对这两种情况进行说明。

iOS 端观众画中画实现

开启对应权限

需要在 iOS 工程的 Signing & Capabilities 处开启以下权限：



调用 SDK 实现

iOS 端 SDK 提供了接口来实现画中画的功能，通过调用 API 可以方便的开启画中画（相关 API 见下面的示例代码），但 SDK 只提供观看单个主播的画中画能力，如果需要支持观看多个主播 PK 画中画的能力，需要调用系统的 API 来实现，详情请参见 调用系统 API 实现 。

RTC Engine 播放

> 说明：
> 需要 RTC Engine 的 SDK 在12.1及以上版本。

在观众端调用如下接口开启。

objectivec

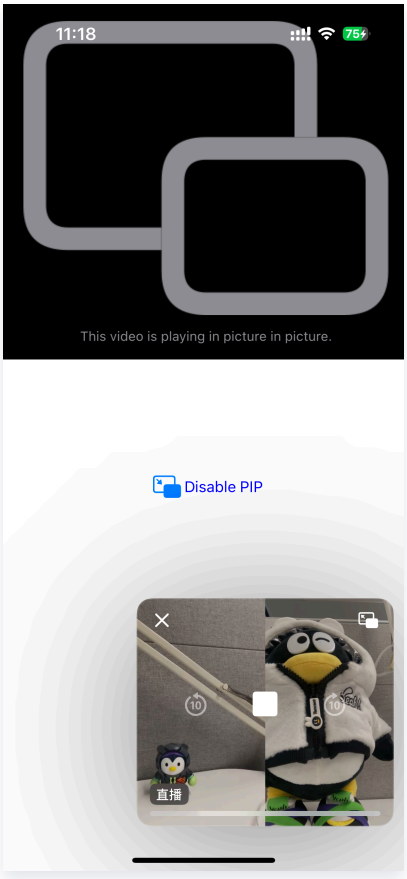
```
NSDictionary *param = @{
    @"api" : @"enablePictureInPictureFloatingWindow",
    @"params" : @{
        @"enable" : @(true)
    }
};
NSError *err = nil;
NSData *jsonData = [NSJSONSerialization dataWithJSONObject:param
options:0 error:&err];
if (err) {
    NSLog(@"error: %@", err);
}
NSString *paramJsonString = [[NSString alloc] initWithData:jsonData
encoding:NSUTF8StringEncoding];
[self.trtcCloud callExperimentalAPI:paramJsonString];
```

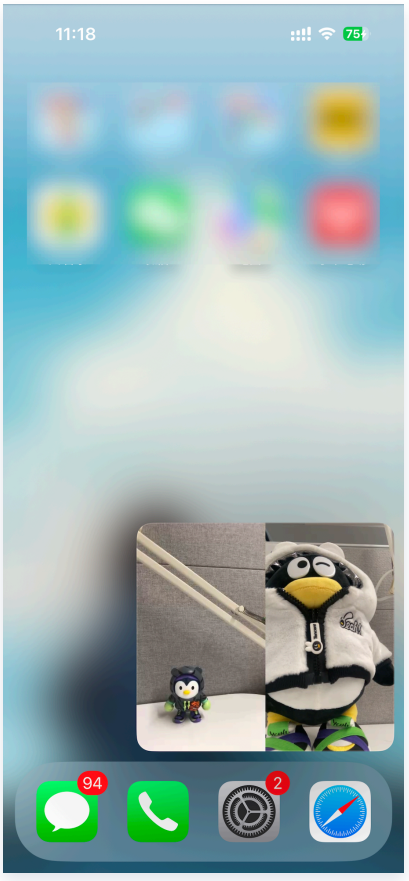
```swift
let param: [String : Any] = ["api":
"enablePictureInPictureFloatingWindow", "params": ["enable":true]]
if let jsonData = try? JSONSerialization.data(withJSONObject: param,
options: .fragmentsAllowed) {
    let paramJsonString = String.init(data: jsonData, encoding: .utf8)
?? ""
    trtcCloud.callExperimentalAPI(paramJsonString)
}
```

如果需要关闭，在对应参数位置传入 false 即可。

调用系统 API 实现

画中画是 iOS 系统提供的能力，通过调用系统 API 能实现复杂场景的画中画能力。iOS 虽然支持画中画能力，但对该特性有较多的限制，无法直接使用渲染视频的 UIView 来实现画中画，需要使用自定义渲染，将需要展示在画中画的视频渲染到符合要求的组件上。下面以2个主播 PK 画中画的场景为例，介绍调用系统 API 对画中画的实现。





> 说明：

> 实现单个主播或多于2个主播画中画依旧可以参考如下方案实现，这里仅对2个主播 PK 的场景进行说明。

1. 定义画中画需要的组件。

   因为 iOS 系统要求只能特定的组件才能渲染到画中画中，这里使用 AVSampleBufferDisplayLayer，并且需要该组件直接渲染对应视频，所以定义一个 combinedPixelBuffer 用于合并2个主播的视频数据。

```
import UIKit
import AVKit
import CoreFoundation
import TXLiteAVSDK_Professional

class PipVC: UIViewController {
    let trtcCloud = TRTCCloud()
    var pipController: AVPictureInPictureController?
    var combinedPixelBuffer: CVPixelBuffer?
    let pixelBufferLock = DispatchQueue(label: "com.demo.pip")
    var pipDisplayLayer: AVSampleBufferDisplayLayer!
}
```

2. 进入 RTC Engine 房间。

```
func enterTrtcRoom() {
    let params = TRTCParams()
    params.sdkAppId = UInt32(SDKAppID)
    params.roomId = UInt32(roomId)
    params.userId = userId
    params.role = .audience
    params.userSig = GenerateTestUserSig.genTestUserSig(identifier:
userId) as String

    trtcCloud.addDelegate(self)
    trtcCloud.enterRoom(params, appScene: .LIVE)
}
```

3. 设置音频并开启后台解码。

```
func setupAudioSession() {
    do {
        try AVAudioSession.sharedInstance().setCategory(.playback)
```

```
    } catch let error {
        print("+> error: \(error)")
        return
    }

    do {
        try AVAudioSession.sharedInstance().setActive(true)
    } catch let error {
        print("+> error: \(error)")
        return
    }
}

func enableBGDecode() {
    let param: [String : Any] = ["api": "enableBackgroundDecoding",
                                 "params": ["enable":true]]
    if let jsonData = try? JSONSerialization.data(withJSONObject:
param, options: .fragmentsAllowed) {
        let paramJsonString = String.init(data: jsonData, encoding:
.utf8) ?? ""
        trtcCloud.callExperimentalAPI(paramJsonString)
    }
}
```

4. 初始化画中画组件。

```
func setupPipController() {
    let screenWidth = UIScreen.main.bounds.width
    let videoHeight = screenWidth / 2 / 9 * 16

    pipDisplayLayer = AVSampleBufferDisplayLayer()
    pipDisplayLayer.frame = CGRect(x: 0, y: 0, width: screenWidth,
height: videoHeight) // Adjust size as needed
    pipDisplayLayer.videoGravity = .resizeAspect
    pipDisplayLayer.isOpaque = true
    pipDisplayLayer.backgroundColor = CGColor(red: 0, green: 0, blue:
0, alpha: 1)
    view.layer.addSublayer(pipDisplayLayer)
```

```
    if AVPictureInPictureController.isPictureInPictureSupported() {
        let contentSource =
AVPictureInPictureController.ContentSource(
            sampleBufferDisplayLayer: pipDisplayLayer,
            playbackDelegate: self
        )
        pipController = AVPictureInPictureController(contentSource:
contentSource)
        pipController?.delegate = self
        pipController?.canStartPictureInPictureAutomaticallyFromInline
= true
    } else {
        print("+> error")
    }
}
```

5. 开启自定义渲染。

> ⚠ **注意：**
> 开启自定义渲染时指定的格式 `._NV12` 和 `步骤6：拼接左右2个画面` 中的方法是相关的，不同的格式需要不同的方法来拼接，该示例仅展示 `._NV12` 格式的左右拼接。

```
extension PipVC: TRTCCloudDelegate {
    func onUserVideoAvailable(_ userId: String, available: Bool) {
        if available {
            trtcCloud.startRemoteView(userId, streamType: .big, view:
nil)
            trtcCloud.setRemoteVideoRenderDelegate(userId, delegate:
self, pixelFormat: ._NV12, bufferType: .pixelBuffer);
        }else{
            trtcCloud.stopRemoteView(userId, streamType: .big)
        }
    }
}
```

6. 拼接左右2个画面。

在拼接2个主播的视频数据时，因为 SDK 回调出视频数据的时间不同步，所以需要在每次收到单个主播的视频数据时去更新对应的数据，并加锁，如果需要实现多个主播的情况，也需要按照类似的方法操作。以下代码是按照

左右2个主播各占一半的方法布局，如果需要其他布局，需要业务上根据需要进行实现，这里不涉及 SDK。

```
func createCombinedPixelBuffer(from sourceBuffer: CVPixelBuffer) {
    let width = CVPixelBufferGetWidth(sourceBuffer) * 2
    let height = CVPixelBufferGetHeight(sourceBuffer)
    let pixelFormat = CVPixelBufferGetPixelFormatType(sourceBuffer)

    let attributes: [CFString: Any] = [
        kCVPixelBufferWidthKey: width,
        kCVPixelBufferHeightKey: height,
        kCVPixelBufferPixelFormatTypeKey: pixelFormat,
        kCVPixelBufferIOSurfacePropertiesKey: [:]
        ]
    CVPixelBufferCreate(kCFAllocatorDefault, width, height,
pixelFormat, attributes as CFDictionary, &combinedPixelBuffer)
}

func updateCombinedPixelBuffer(with sourceBuffer: CVPixelBuffer,
forLeft: Bool) {
    guard let combinedBuffer = combinedPixelBuffer else { print("+>
error"); return}

    CVPixelBufferLockBaseAddress(combinedBuffer, [])
    CVPixelBufferLockBaseAddress(sourceBuffer, [])

    // Plane 0: Y/luma plane
    let combinedLumaBaseAddress =
CVPixelBufferGetBaseAddressOfPlane(combinedBuffer, 0)!
    let sourceLumaBaseAddress =
CVPixelBufferGetBaseAddressOfPlane(sourceBuffer, 0)!
    let combinedLumaBytesPerRow =
CVPixelBufferGetBytesPerRowOfPlane(combinedBuffer, 0)
    let sourceLumaBytesPerRow =
CVPixelBufferGetBytesPerRowOfPlane(sourceBuffer, 0)
    let widthLuma = CVPixelBufferGetWidthOfPlane(sourceBuffer, 0)
    let heightLuma = CVPixelBufferGetHeightOfPlane(sourceBuffer, 0)

    // Plane 1: UV/chroma plane
    let combinedChromaBaseAddress =
CVPixelBufferGetBaseAddressOfPlane(combinedBuffer, 1)!
```

```
    let sourceChromaBaseAddress =
CVPixelBufferGetBaseAddressOfPlane(sourceBuffer, 1)!
    let combinedChromaBytesPerRow =
CVPixelBufferGetBytesPerRowOfPlane(combinedBuffer, 1)
    let sourceChromaBytesPerRow =
CVPixelBufferGetBytesPerRowOfPlane(sourceBuffer, 1)
    let widthChroma = CVPixelBufferGetWidthOfPlane(sourceBuffer, 1)
    let heightChroma = CVPixelBufferGetHeightOfPlane(sourceBuffer, 1)

    for row in 0..<heightLuma {
        let combinedRow = combinedLumaBaseAddress.advanced(by: row *
combinedLumaBytesPerRow + (forLeft ? 0 : widthLuma))
        let sourceRow = sourceLumaBaseAddress.advanced(by: row *
sourceLumaBytesPerRow)
        memcpy(combinedRow, sourceRow, widthLuma)
    }

    // ._nv12 the chroma plane is subsampled 2:1 horizontally and
vertically
    for row in 0..<heightChroma {
        let combinedRow = combinedChromaBaseAddress.advanced(by: row *
combinedChromaBytesPerRow + (forLeft ? 0 : 2 * widthChroma))
        let sourceRow = sourceChromaBaseAddress.advanced(by: row *
sourceChromaBytesPerRow)
        memcpy(combinedRow, sourceRow, 2 * widthChroma)
    }

    CVPixelBufferUnlockBaseAddress(sourceBuffer, [])
    CVPixelBufferUnlockBaseAddress(combinedBuffer, [])
}
```

7. 将合并后的画面渲染到对应的组件上。

```
func displayPixelBuffer(_ pixelBuffer: CVPixelBuffer, in layer:
AVSampleBufferDisplayLayer) {
    var timing = CMSampleTimingInfo.init(duration: .invalid,
                                         presentationTimeStamp:
.invalid,
                                         decodeTimeStamp: .invalid)
```

```
    var videoInfo: CMVideoFormatDescription? = nil
    var result =
CMVideoFormatDescriptionCreateForImageBuffer(allocator: nil,

imageBuffer: pixelBuffer,

formatDescriptionOut: &videoInfo)
    if result != 0 {
        return
    }
    guard let videoInfo = videoInfo else {
        return
    }
    var sampleBuffer: CMSampleBuffer? = nil
    result = CMSampleBufferCreateForImageBuffer(allocator:
kCFAllocatorDefault,
                                                imageBuffer:
pixelBuffer,
                                                dataReady: true,
                                                makeDataReadyCallback:
nil,
                                                refcon: nil,
                                                formatDescription:
videoInfo,
                                                sampleTiming: &timing,
                                                sampleBufferOut:
&sampleBuffer)
    if result != 0 {
        return
    }
    guard let sampleBuffer = sampleBuffer else {
        return
    }
    guard let attachments =
CMSampleBufferGetSampleAttachmentsArray(sampleBuffer,

createIfNecessary: true) else {
        return
    }
    CFDictionarySetValue(
```

```
        unsafeBitCast(CFArrayGetValueAtIndex(attachments, 0), to:
CFMutableDictionary.self),

Unmanaged.passUnretained(kCMSampleAttachmentKey_DisplayImmediately).to
Opaque(),
        Unmanaged.passUnretained(kCFBooleanTrue).toOpaque())
        layer.enqueue(sampleBuffer)
    if layer.status == .failed {
        if let error = layer.error as? NSError {
            if error.code == -11847 {
                print("+> error")
            }
        }
    }
}
```

8. 拿到远端用户的视频数据，拼接后渲染到指定的组件。

   示例中使用 left 标识显示在左边的主播 ID，实际业务中需要根据业务需要进行修改。

```
extension PipVC: TRTCVideoRenderDelegate {
    func onRenderVideoFrame(_ frame: TRTCVideoFrame, userId: String?,
streamType: TRTCVideoStreamType) {
        guard let newPixelBuffer = frame.pixelBuffer else { print("+>
error"); return}
        pixelBufferLock.sync {
            if combinedPixelBuffer == nil {
                createCombinedPixelBuffer(from: newPixelBuffer)
            }
            if userId == "left" {
                updateCombinedPixelBuffer(with: newPixelBuffer,
forLeft: true)
            } else {
                updateCombinedPixelBuffer(with: newPixelBuffer,
forLeft: false)
            }
        }

        if let combinedBuffer = combinedPixelBuffer {
            DispatchQueue.main.async {
```

```
                self.displayPixelBuffer(combinedBuffer, in:
self.pipDisplayLayer)
            }
        }
    }
}
```

9. 实现相关协议。

```
extension PipVC: AVPictureInPictureControllerDelegate {
    func pictureInPictureControllerWillStartPictureInPicture(_
pictureInPictureController: AVPictureInPictureController) {
    }
    func pictureInPictureControllerDidStartPictureInPicture(_
pictureInPictureController: AVPictureInPictureController) {
    }
    func pictureInPictureControllerDidStopPictureInPicture(_
pictureInPictureController: AVPictureInPictureController) {
    }
    func pictureInPictureController(_ pictureInPictureController:
AVPictureInPictureController,
restoreUserInterfaceForPictureInPictureStopWithCompletionHandler
completionHandler: @escaping (Bool) -> Void) {
        completionHandler(true)
    }
    func pictureInPictureController(_ pictureInPictureController:
AVPictureInPictureController, failedToStartPictureInPictureWithError
error: any Error) {
    }
}

extension PipVC: AVPictureInPictureSampleBufferPlaybackDelegate {
    func pictureInPictureControllerTimeRangeForPlayback(_
pictureInPictureController: AVPictureInPictureController) ->
CMTimeRange {
        return CMTimeRange.init(start: .zero, duration:
.positiveInfinity)
    }
```

```
    func pictureInPictureControllerIsPlaybackPaused(_
pictureInPictureController: AVPictureInPictureController) -> Bool {
        return false
    }
    func pictureInPictureController(_ pictureInPictureController:
AVPictureInPictureController, setPlaying playing: Bool) {
    }
    func pictureInPictureController(_ pictureInPictureController:
AVPictureInPictureController, didTransitionToRenderSize newRenderSize:
CMVideoDimensions) {
    }
    func pictureInPictureController(_ pictureInPictureController:
AVPictureInPictureController, skipByInterval skipInterval: CMTime)
async {
    }
}
```

10. 开启/关闭画中画。

```
// 关闭画中画
pipController?.stopPictureInPicture()
// 开启画中画
pipController?.startPictureInPicture()
```

> ⚠ 注意：
> - 这里仅对实现方案进行说明，实际业务中还需要对各种可能的异常情况进行处理。
> - 对于画中画上层控制按钮的处理是 iOS 系统相关能力，不涉及到 SDK，这里不做说明，需要业务根据实际需要进行处理。

Android 端画中画的实现

从 Android 8.0（API 级别26）开始，Android 允许以画中画（PIP）模式启动 activity。画中画是一种特殊类型的多窗口模式，最常用于视频播放。使用该模式，用户可以通过固定到屏幕一角的小窗口观看视频，同时在应用之间进行导航或浏览主屏幕上的内容。RTC Engine SDK没有对 Android 画中画 API 进一步封装，画中画的功能是直接调用 Android API 实现的。如需了解更多信息，请参见 Android 文档 使用画中画（PIP）功能添加视频 。
Android 端在进入画中画时，会根据 xml 的布局规则，以画中画的窗口大小，重新进行 measure、layout。所以主播端和观众端都可按照此规则实现画中画。

画中画的实现

下面是根据 Android 文档 使用画中画（PIP）功能添加视频 来实现。

1. 在 AndroidManifest.xml 中对<activity>声明画中画属性。

```
<activity

android:name="com.tencent.trtc.pictureinpicture.PictureInPictureActivi
ty"
    android:theme="@style/Theme.AppCompat.Light.NoActionBar"

android:configChanges="screenSize|smallestScreenSize|screenLayout|orie
ntation"
    android:supportsPictureInPicture="true"
```

   - `android:supportsPictureInPicture="true"` 表示声明支持画中画。
   - 在画中画模式转换期间出现布局更改，如果不希望 activity 重启动，需要配置 android:configChanges 属性的对应值。

2. 进入画中画。

```
private void startPictureInPicture() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        PictureInPictureParams.Builder pictureInPictureBuilder = new
PictureInPictureParams.Builder();
        Rational aspectRatio = new Rational(mVideoView.getWidth(),
mVideoView.getHeight());
        pictureInPictureBuilder.setAspectRatio(aspectRatio);
        //进入画中画
        enterPictureInPictureMode(pictureInPictureBuilder.build());
    } else {
        Toast.makeText(this,
R.string.picture_in_picture_not_supported, Toast.LENGTH_SHORT).show();
    }
}
```

   - `pictureInPictureBuilder.setAspectRatio(aspectRatio);` 设置画中画的宽高比，此处设置为播放视频 View 的宽高比。
   - `enterPictureInPictureMode(pictureInPictureBuilder.build());` 进入画中画。

3. 进出画中画的回调。

```
@Override
public void onPictureInPictureModeChanged(boolean
isInPictureInPictureMode, Configuration configuration) {
    super.onPictureInPictureModeChanged(isInPictureInPictureMode,
configuration);
    if (isInPictureInPictureMode) {
        //进入画中画后，需要隐藏的view
    } else{
        //退出画中画后，需要显示的view
    }
}
```

在画中画中显示多个视频画面

想要显示多个视频画面，可以在进入画中画时，对 View A 设置固定的宽高，其他 View 会根据布局规则来显示或者设置百分比布局。

> 说明：
> 在画中画中显示多个视频画面方式不是 Android 的规定用法，且当前可在 Android 12上使用，后续可能会随着 Android 系统的更新而变化，需要在发布前测试各版本系统的兼容性。

效果展示


```
// mTRTCCloud 对应左边的视频View（TXCloudVideoView），设置了
TRTC_VIDEO_RENDER_MODE_FIT
TRTCCloudDef.TRTCRenderParams param = new
TRTCCloudDef.TRTCRenderParams();
param.fillMode      = TRTCCloudDef.TRTC_VIDEO_RENDER_MODE_FIT;
mTRTCCloud.setRemoteRenderParams(remoteUserIdA,TRTCCloudDef.TRTC_VIDEO_S
TREAM_TYPE_BIG, param);
mTRTCCloud.startRemoteView(remoteUserIdA,
TRTCCloudDef.TRTC_VIDEO_STREAM_TYPE_BIG, mTXCloudRemoteView);


// mTRTCCloud 对应右边的视频View（TXCloudVideoView）
mTRTCCloud.startRemoteView(remoteUserIdB,
TRTCCloudDef.TRTC_VIDEO_STREAM_TYPE_BIG, mTXCloudRemoteView);
```

进入画中画后的布局，可以计算并手动设置 TXCloudVideoView 的宽高或者设置填充方式，来保障视频画面的完整显示。调用 mTRTCCloud（TRTCCloud 对象）的 setRemoteRenderParams 方法，来设置视频画面的填充方式。

- 画中画左边 TXCloudVideoView 是设置了 TRTC_VIDEO_RENDER_MODE_FIT 的效果。

- 画中画右边 TXCloudVideoView 是设置了 TRTC_VIDEO_RENDER_MODE_FILL 的效果。

此示例中，只有两个视频画面（TXCloudVideoView），对左边的 TXCloudVideoView 设置宽高，右边的 TXCloudVideoView 会根据布局规则来显示。如果您有多个 TXCloudVideoView，可以合理设计布局，达到目标效果。

实现步骤

1. 在 activity_picture_in_picture.xml 中添加两个 TXCloudVideoView 并排显示。

```
<com.tencent.rtmp.ui.TXCloudVideoView
    android:id="@+id/video_view"
    android:layout_width="192dp"
    android:layout_height="108dp"
    android:layout_alignParentStart="true"
    android:background="#00BCD4"/>

<com.tencent.rtmp.ui.TXCloudVideoView
    android:id="@+id/video_view2"
    android:layout_width="192dp"
    android:layout_height="108dp"
    android:layout_alignTop="@+id/video_view"
    android:layout_toEndOf="@+id/video_view"
    android:background="#3F51B5"/>
```

2. 在进入和退出画中画的时候，设置 video_view 的宽高。

```
@Override
public void onPictureInPictureModeChanged(boolean
isInPictureInPictureMode, Configuration configuration) {
    super.onPictureInPictureModeChanged(isInPictureInPictureMode,
configuration);
    if (isInPictureInPictureMode) {
        // 设置mVideoView 的宽为 100dp
        RelativeLayout.LayoutParams layoutParams =
(RelativeLayout.LayoutParams) mVideoView.getLayoutParams();
        layoutParams.width = (int)
TypedValue.applyDimension(TypedValue.COMPLEX_UNIT_DIP, 100,
getResources().getDisplayMetrics());
    } else {
        // 退出画中画，还原video_view的宽度
```

```
        RelativeLayout.LayoutParams layoutParams =
(RelativeLayout.LayoutParams) mVideoView.getLayoutParams();
        layoutParams.width = (int)
TypedValue.applyDimension(TypedValue.COMPLEX_UNIT_DIP, 192,
getResources().getDisplayMetrics());
    }
}
```

Flutter 端观众画中画实现

Flutter 端开启画中画在不同平台有不同的实现，下面分别对 iOS 和 Android 平台进行说明。

发布到 iOS 设备

调用 SDK 实现

Flutter端同样可以通过调用 SDK 提供的 API 方便的开启画中画，和原生 iOS 端一样，SDK 只提供观看单个主播的画中画能力，如果需要画中画中显示多个主播的视频，请参见 调用系统 API 实现 。

> ⚠ 注意：
> 同样需要在 Flutter 生成的 iOS 工程中开启对应的权限，可参见本文 开启对应权限 部分。

RTC Engine 播放

Flutter SDK 需要2.9.1版本及以后，在观众端调用如下接口实现。

```
trtcCloud.callExperimentalAPI(jsonEncode({
    "api": "enablePictureInPictureFloatingWindow",
    "params": {"enable": true}
}));
```

如果需要关闭，在对应参数位置传入 false 即可。

直播播放

在观众端调用如下接口开启。

```
var pipCode = await _livePlayer!.enablePictureInPicture(true);
if (pipCode != V2TXLIVE_OK) {
  print("error: $pipCode");
}
```

如果需要关闭，在对应参数位置传入 false 即可。

调用系统 API 实现

如果需要实现复杂的画中画能力，例如在画中画中显示多个主播的视频，需要调用 iOS 系统提供的 API 来实现，请参见 iOS 端观众画中画实现-调用系统 API 实现 部分。下面对 Flutter 端调用 iOS 系统 API 部分进行说明。

1. Flutter 端使用 MethodChannel 发消息到 iOS 原生端。

```
final channel = MethodChannel('flutter_ios_pip_demo');

await channel.invokeMethod('enablePip', {
                              'marginTop': appBarHeight +
topSafeAreaHeight,
                              'pkLeft': pkLeftUserId,
                              'pkRight': pkRightUserId,
                            });
```

2. 在 Flutter 打包出的 iOS 工程中，接收对应的消息，并做相应处理。

   Flutter 调用系统 API 实现多主播 PK 画中画时，实际上是调用 iOS 系统 API，使用自定义采集重新绘制2个主播 PK 的画面，并显示在 Flutter 层之上，所以需要在调用 iOS 系统 API 绘制的窗口大小和位置与 Flutter 端一致，可以在 MethodChannel 中传递相应的布局参数和主播 ID。

```
var channel: FlutterMethodChannel?
let pipListener = PipRender()

guard let controller = window?.rootViewController as?
FlutterViewController else {
    fatalError("Invalid root view controller")
}

channel = FlutterMethodChannel(name: "flutter_ios_pip_demo",
binaryMessenger: controller.binaryMessenger)
channel?.setMethodCallHandler({ [weak self] call, result in
    guard let self = self else { return }
    switch (call.method) {
    case "enablePip":
        if let arg = call.arguments as? [String: Any] {
            let marginTop = arg["marginTop"] as? CGFloat ?? 0
            let pkLeft = arg["pkLeft"] as? String ?? ""
            let pkRight = arg["pkRight"] as? String ?? ""
```

```
            pipListener.enablePip(mainView: vc.view, mt: mt, pkLeft:
pkLeft, pkRight: pkRight)
        }
        result(nil)
        break
    case "disablePip":
        pipListener.disablePip()
        result(nil)
        break
    default:
        break
    }
})
```

3. 定义处理画中画的类。

   在开启画中画时，将对应主播的流改为自定义渲染，并将渲染后的画面插入到根视图中，用于显示。

```
import UIKit
import AVKit
import TXLiteAVSDK_Professional

class PipRender: NSObject {
    // 其余变量参考iOS原生端调用系统API实现的部分
    var mainView: UIView?
    var mt: CGFloat?

    // 因为trtcCloud是单实例的，可以在代码中这样获取
    let trtcCloud = TRTCCloud.sharedInstance()

    func disablePip() {
        pipDisplayLayer?.removeFromSuperlayer()
        pipController?.stopPictureInPicture()
    }

    func enablePip(mainView: UIView, mt: CGFloat, pkLeft: String,
pkRight: String) {
        self.mainView = mainView
        self.mt = mt
        trtcCloud.addDelegate(self)
```

```
        enableBGDecode()
        setupAudioSession()
        setupPipController()
        pipController?.startPictureInPicture()
        if pkLeft.count > 0 {
            trtcCloud.startRemoteView(pkLeft, streamType: .big, view:
nil)
            trtcCloud.setRemoteVideoRenderDelegate(pkLeft, delegate:
self, pixelFormat: ._NV12, bufferType: .pixelBuffer);
        }
        if pkRight.count > 0 {
            trtcCloud.startRemoteView(pkRight, streamType: .big, view:
nil)
            trtcCloud.setRemoteVideoRenderDelegate(pkRight, delegate:
self, pixelFormat: ._NV12, bufferType: .pixelBuffer);
        }
    }

    // 该方法需要根据业务需要，调整画中画显示的位置，保证和Flutter端显示的位置一
致
    func setupPipController() {
        let screenWidth = UIScreen.main.bounds.width
        let videoHeight = screenWidth / 2 / 9 * 16
        pipDisplayLayer = AVSampleBufferDisplayLayer()
        // 这里根据实际需要，调整画中画显示的位置
        let tsa = self.mainView?.safeAreaInsets.top ??
        let vmt = tsa + (self.mt ?? 0)
        pipDisplayLayer.frame = CGRect(x: 0, y: vmt, width:
screenWidth, height: videoHeight) // Adjust size as needed
        pipDisplayLayer.videoGravity = .resizeAspect
        pipDisplayLayer.isOpaque = true
        pipDisplayLayer.backgroundColor = CGColor(red: 0, green: 0,
blue: 0, alpha: 1)
        // 这里使用enablePip传递进来的mainView添加画中画的画面
        mainView?.layer.addSublayer(pipDisplayLayer)

        if AVPictureInPictureController.isPictureInPictureSupported()
{
            let contentSource =
AVPictureInPictureController.ContentSource(
```

```
                sampleBufferDisplayLayer: pipDisplayLayer,
                playbackDelegate: self
            )
            pipController =
AVPictureInPictureController(contentSource: contentSource)
            pipController?.delegate = self

pipController?.canStartPictureInPictureAutomaticallyFromInline = true
        } else {
            print("+> PiP not supported")
        }
    }

    // 其余的方法，和iOS原生端调用系统API实现的部分一致
}
```

4. Flutter 端在停止画中画时需要重新拉对应的主播的流，恢复 Flutter 端的渲染。

```
// 业务上触发停止画中画时
trtcCloud.startRemoteView(pkLeftUserId,
TRTCCloudDef.TRTC_VIDEO_STREAM_TYPE_BIG, pkLeftId);
trtcCloud.startRemoteView(pkRightUserId,
TRTCCloudDef.TRTC_VIDEO_STREAM_TYPE_BIG, pkRightId);

await channel.invokeMethod('disablePip');
```

5. Flutter 端在销毁当前页面时需要停止画中画。

   因为开始画中画后，实际是调用 iOS 系统 API 重新绘制一个视图，覆盖在 Flutter 的视图之上，所以在销毁当前页面时，需要停止画中画，将对应的视图从根视图中移除。

```
@override
dispose() {
    channel.invokeMethod('disablePip');
    super.dispose();
}
```

在 Android 上通过 Flutter 实现画中画

在 Flutter 中实现画中画，也是需要调用 Android 画中画的 API。进入画中画后，Flutter UI 会按照已有的 Widget 布局规则进行显示，可以根据自己的业务规则，在进入画中画后，隐藏部分 Widget，合理设置视频 Widget 的宽高。

使用 平台通道调用 Android 的代码，通道分为客户端（Flutter）和宿主端（Android）， 下面来看画中画的具体实现：

1. Flutter 客户端的代码：使用通道名"samples.flutter.dev" 调用通道方法"pictureInPicture"，这个方法的具体实现在 Android 宿主端。

```
MethodChannel _channel = MethodChannel('samples.flutter.dev');
final int? result = await _channel.invokeMethod('pictureInPicture');
```

2. Android 宿主端的代码。

   2.1 在继承了 FlutterActivity 的 Activity 中实现：

```
private void startPictureInPicture() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        PictureInPictureParams.Builder pictureInPictureBuilder = new
PictureInPictureParams.Builder();
        //根据具体的业务需求，设置指定的画中画大小
        Rational aspectRatio = new Rational(100, 100);
        pictureInPictureBuilder.setAspectRatio(aspectRatio);
        //进入画中画
        enterPictureInPictureMode(pictureInPictureBuilder.build());
    } else {
        Toast.makeText(this,
R.string.picture_in_picture_not_supported, Toast.LENGTH_SHORT).show();
    }
}

@Override
public void configureFlutterEngine(@NonNull FlutterEngine
flutterEngine) {
    super.configureFlutterEngine(flutterEngine);
    MethodChannel channel = new
MethodChannel(flutterEngine.getDartExecutor().getBinaryMessenger(),
"samples.flutter.dev");
    channel.setMethodCallHandler(
            (call, result) -> {
                if (call.method.equals("pictureInPicture")) {
```

```
                    startPictureInPicture();
                } else {
                    result.notImplemented();
                }
            }
    );
}
```

2.2 在 AndroidManifest.xml 中对 activity 配置画中画参数 `android:supportsPictureInPicture="true"`，如下：

```
<activity

android:name="example.android.app.src.main.java.com.tencent.live.example.MainActivity"
    android:supportsPictureInPicture="true"

android:configChanges="orientation|keyboardHidden|keyboard|screenSize|smallestScreenSize|locale|layoutDirection|fontScale|screenLayout|density|uiMode"
    >
    ...
</activity>
```

想要在画中画中显示两个视频画面，例如主播 PK，可以通过合理设置布局规则和大小来实现。

直播上下滑

最近更新时间：2025-11-18 15:15:34

直播场景中，面对大量主播提供的多种多样的视频内容，通过上下滑动的方式快速浏览和选择自己喜爱的内容可以给用户带来较好的使用体验。本文将分别介绍 Android 和 iOS 端的方案。

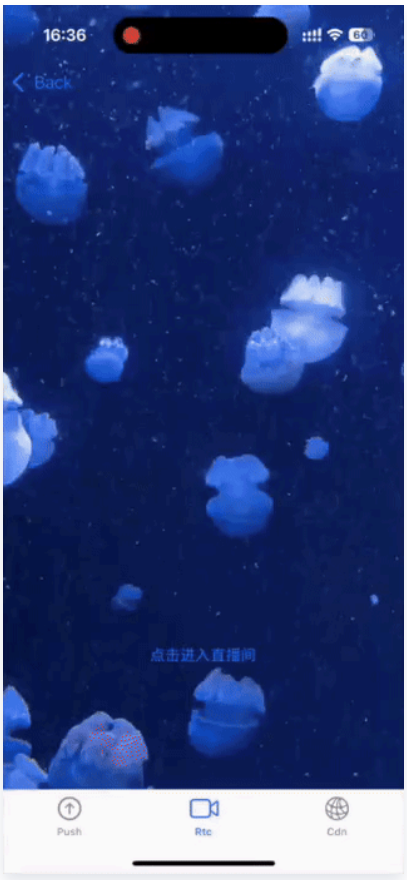
iOS 端直播上下滑方案

在这种场景下，切换直播间时，因为进新的房间和拉流需要时间，会导致前后两个房间的视频画面不连贯，对于该问题，一般有以下3种解决方案，本文将分别对这3种方案进行详细说明。

实现方式	用户体验	资源消耗	实现逻辑
黑屏	一般， 切换直播间时先显示黑屏，再出现新的画面	无额外资源消耗	简单，无额外逻辑
占位图	稍好，切换直播间时，先显示对应主播的固定占位图，再出现新的画面	稍多，需额外为每个主播存储一个占位图，并加载到客户端	稍复杂，切换前需额外异步加载完成占位图
双实例	最好，切换直播间时，流畅显示前后2个主播的画面	较多，在列表中需同时拉2路流，进入直播间后可以只拉1路流	较复杂，需使用多个实例，并控制不同实例的音视频拉取

黑屏

通过监听系统的滑动事件，调用切换房间的接口来切换直播间，在新的直播间加载出来之前，没有内容显示，体现为短暂黑屏，为了方便说明，这里黑屏时间为1秒左右，具体黑屏时间受网络和视频码率影响，效果如下。



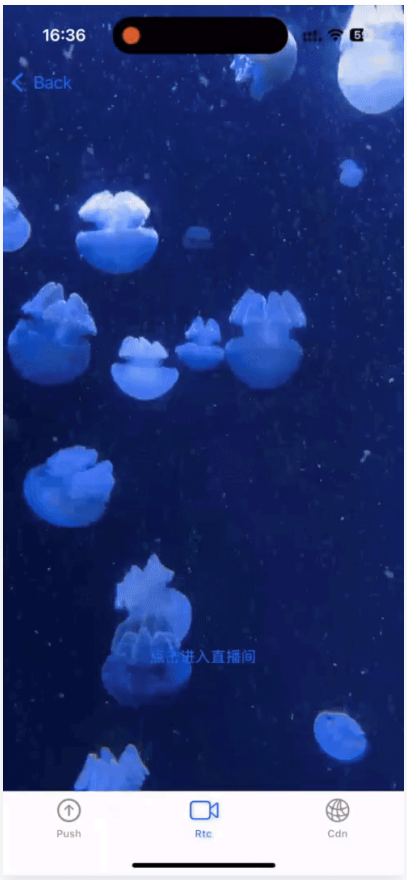
其中切换房间的代码片段如下：

```
let src = TRTCSwitchRoomConfig()
// 根据业务生成对应的房间号以及进房凭证，示例中使用客户端生成进房凭证，线上业务请从后台获取
src.strRoomId = strRoomId
src.userSig = GenerateTestUserSig.genTestUserSig(identifier: userId) as String
trtcCloud.switchRoom(src)
```

占位图

通过监听系统的滑动事件，调用切换房间的接口来切换直播间，但和黑屏方案不同，该方案需要提前加载每个直播间的占位图，在直播间视频流显示之前，显示对应直播间的占位图，为了方便说明，这里占位图持续时间1秒左右，具体时间受网络和视频码率影响。

效果图



实现步骤

1. 设置第一个主播的背景图。

```
bgView = UIImageView(frame: self.view.bounds)
// 该图片需为对应直播间的占位图，需业务上提前获取
bgView.image = UIImage(named: "1.png")
bgView.contentMode = .scaleAspectFill
bgView.translatesAutoresizingMaskIntoConstraints = false
self.view.insertSubview(bgView, at: 0)

NSLayoutConstraint.activate([
    bgView.topAnchor.constraint(equalTo: view.topAnchor),
    bgView.bottomAnchor.constraint(equalTo: view.bottomAnchor),
    bgView.leadingAnchor.constraint(equalTo: view.leadingAnchor),
    bgView.trailingAnchor.constraint(equalTo: view.trailingAnchor),
])
```

2. 在切换直播间之前切换背景图。

```
DispatchQueue.main.async {
    UIView.transition(
            with: self.bgView,
            duration: 0,
            options: .transitionCrossDissolve,
            animations: {
                // 业务上切换对应的占位图
                self.bgView.image = UIImage(named: strRoomId)
            }, completion: nil)
}

let src = TRTCSwitchRoomConfig()
// 根据业务生成对应的房间号以及进房凭证，示例中使用客户端生成进房凭证，线上业务请从后台获取
src.strRoomId = strRoomId
src.userSig = GenerateTestUserSig.genTestUserSig(identifier: userId) as String
trtcCloud.switchRoom(src)
```

3. 在新直播间第一帧视频画面开始渲染时，切换背景图到视频画面。

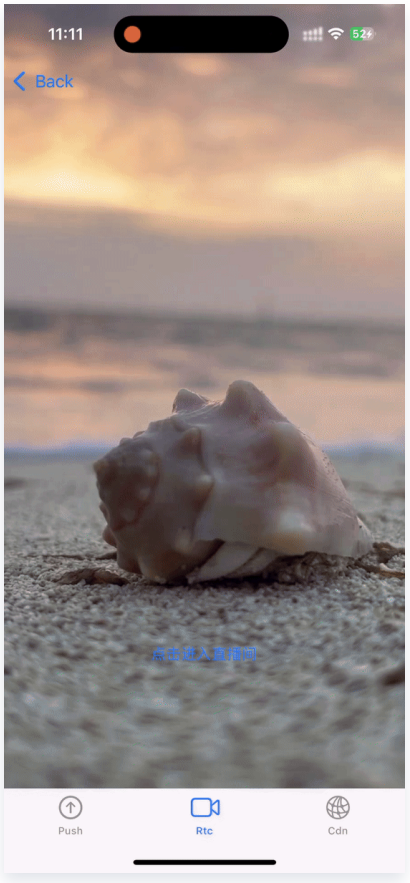
```
// 拉视频流
func onUserVideoAvailable(_ userId: String, available: Bool) {
    if available {
        trtcCloud.startRemoteView(userId, streamType: .big, view: view)
    } else {
        trtcCloud.stopRemoteView(userId, streamType: .big)
    }
}

// 在开始渲染第一帧视频画面时，切换占位图到背景，显示视频画面
func onFirstVideoFrame(_ userId: String, streamType: TRTCVideoStreamType, width: Int32, height: Int32) {
    // 这里是调整背景图和视频渲染控件的前后顺序，实际业务中根据实际情况调整
    self.view.exchangeSubview(at: 1, withSubviewAt: 0)
}
```

双实例

> ⚠ 注意：
> 该方案虽显示效果最佳，用户体验最好，但在滑动列表中需要同时拉前后2个直播间的2路流，尽管可以在用户进入直播间时停止预加载下一个直播间的流，但整体流量和费用消耗会更多。

效果图

为了实现最丝滑流畅的上下滑效果，需要同时使用2个实例，在观看前一个直播间的同时，预加载下一个直播间的视频画面，并借助 UIPageViewController 或手动根据滑动位置调整上下2个视频的显示位置，实现自然流畅的过渡，滑动观看下一个直播间的效果如下图左侧示例。



该方案的整体逻辑为，进入当前直播间后，立即使用子实例进入下一个直播间，拉下一个直播间的视频流，并将下一个直播间的视频流显示到UIPageViewController 的下一个 Page 中。在进入新的直播间时，开启新的直播间的音频流，如此循环，可以达到最流畅的上下滑效果；同时为了避免过多的资源消耗，只对观看下一个直播间进行预加载，使用场景较少的观看上一个直播间不做预加载，切换时依旧使用黑屏，观看上一个直播间的效果如上图右侧示例。

实现步骤

1. 定义子实例工具类。

```
import Foundation
import ObjectiveC
```

```
import TXLiteAVSDK_Professional

@objc protocol SubCloudHelperDelegate : NSObjectProtocol {
    @objc optional func onUserVideoAvailableWithSubId(subId: Int,
userId: String, available: Bool)
}

class SubCloudHelper:NSObject,TRTCCloudDelegate {
    var trtcCloud: TRTCCloud!
    var subId: Int!
    weak var delegate : SubCloudHelperDelegate? = nil

    func initWithSubId(subId: Int, trtcIns: TRTCCloud) {
        self.subId = subId
        self.trtcCloud = trtcIns
        self.trtcCloud.addDelegate(self)
    }

    func getCloud()->TRTCCloud {
        return trtcCloud
    }

    func onUserVideoAvailable(_ userId: String, available: Bool) {
        if self.delegate?.onUserVideoAvailableWithSubId?(subId: subId,
userId: userId, available: available) == nil {
            return
        }
    }
}
```

2. 使用子实例。

```
let trtcCloud = TRTCCloud()
let subCloudHelper = SubCloudHelper()

override func viewDidLoad() {
    super.viewDidLoad()
```

```
    subCloudHelper.initWithSubId(subId: 0, trtcIns:
trtcCloud.createSub())
    subCloudHelper.delegate = self
}
```

3. 准备好用于切换的 UIPageViewController。

```
private var atRoom: Bool = false

var pageViewController: UIPageViewController!
var pageZero: UIViewController!
var pageOne: UIViewController!
var pageTwo: UIViewController!
var pages: [UIViewController] = []

var curPageIdx = 0
var curIsSub = false


func setupPages() {
    pageViewController = UIPageViewController(
        transitionStyle: .scroll,
        navigationOrientation: .vertical
    )
    pageViewController.dataSource = self
    pageViewController.delegate = self
    addChild(pageViewController)
    view.addSubview(pageViewController.view)
    pageViewController.didMove(toParent: self)

    // pages
    pageZero = UIViewController()
    pageZero.view.backgroundColor = .black
    pageOne = UIViewController()
    pageOne.view.backgroundColor = .black
    pageTwo = UIViewController()
    pageTwo.view.backgroundColor = .black
    pages = [pageZero, pageOne, pageTwo]
```

```
    pageViewController.setViewControllers([pages[curPageIdx]],
direction: .forward, animated: false)
}
```

4. 实现 pageViewController 的页面切换。

```
// 获取下/上一个Page
func getShowPage(isNext: Bool) -> UIViewController {
    var newPageIdx = 0
    if isNext {
        newPageIdx = curPageIdx + 1
    } else {
        newPageIdx = curPageIdx - 1
    }
    if newPageIdx >= pages.count {
        newPageIdx = 0
    } else if newPageIdx < 0 {
        newPageIdx = pages.count - 1
    }
    return pages[newPageIdx]
}

extension RtcDuplexVC: UIPageViewControllerDataSource {
    func pageViewController(_ pageViewController:
UIPageViewController, viewControllerBefore viewController:
UIViewController) -> UIViewController? {
        return getShowPage(isNext: false)
    }

    func pageViewController(_ pageViewController:
UIPageViewController, viewControllerAfter viewController:
UIViewController) -> UIViewController? {
        return getShowPage(isNext: true)
    }
}
```

5. 主实例进房，并使用子实现预加载下一个直播间。

```
// 主实例进房
```

```
// 根据实际业务替换sdkAppId,roomID,strRoomId,userId和userSig
// 示例中使用客户端生成userSig，线上业务请从后台获取
let params = TRTCParams()
params.sdkAppId = UInt32(SDKAppID)
params.roomId = 0
params.strRoomId = strRoomIdLst.first ?? "1"
params.userId = userId
params.role = .anchor
params.userSig = GenerateTestUserSig.genTestUserSig(identifier:
userId) as String
trtcCloud.addDelegate(self)
trtcCloud.enterRoom(params, appScene: .LIVE)

// 子实例预加载进入下一个房间
// 根据实际业务替换sdkAppId,roomID,strRoomId,userId和userSig
// 示例中使用客户端生成userSig，线上业务请从后台获取
let subParams = TRTCParams()
subParams.sdkAppId = UInt32(SDKAppID)
subParams.roomId = 0
subParams.strRoomId = strRoomIdLst[1]
subParams.userId = userId
subParams.role = .anchor
subParams.userSig = GenerateTestUserSig.genTestUserSig(identifier:
userId) as String
subCloudHelper.trtcCloud.enterRoom(subParams, appScene: .LIVE)
subCloudHelper.trtcCloud.muteAllRemoteAudio(true)
```

6. 根据回调拉视频流并渲染到对应的 page 上。

```
func getPageByIdx(isNext: Bool) -> UIViewController {
    var newPageIdx = curPageIdx
    if isNext {
        newPageIdx += 1
    }
    if newPageIdx >= pages.count {
        newPageIdx = 0
    }
    return pages[newPageIdx]
}
```

```
extension RtcDuplexVC: TRTCCloudDelegate {
    func onUserVideoAvailable(_ userId: String, available: Bool) {
        if available {
            trtcCloud.startRemoteView(userId, streamType: .big, view:
getPageByIdx(isNext: curIsSub).view)
        } else {
            trtcCloud.stopRemoteView(userId, streamType: .big)
        }
    }
}
extension RtcDuplexVC: SubCloudHelperDelegate {
    func onUserVideoAvailableWithSubId(subId: Int, userId: String,
available: Bool) {
        if available {
            subCloudHelper.trtcCloud.startRemoteView(userId,
streamType: .big, view: getPageByIdx(isNext: !curIsSub).view)
        } else {
            subCloudHelper.trtcCloud.stopRemoteView(userId,
streamType: .big)
        }
    }
}
```

7. 切换到新的房间后，更新预加载的房间，或者在向上滑时，更新当前显示的房间。

```
func updateCurRoomIdx(isNext: Bool) {
    if isNext {
        curRoomIdx += 1
        if curRoomIdx >= strRoomIdLst.count {
            curRoomIdx = 0
        }
    } else {
        curRoomIdx -= 1
        if curRoomIdx < 0 {
            curRoomIdx = strRoomIdLst.count - 1
        }
    }
}
```

```
// 这里需要根据实际的业务逻辑切换房间号
func updateNewRoom(isNext: Bool) {
    var newRoomIdx = 0
    if isNext{
        newRoomIdx = curRoomIdx + 1
    } else {
        newRoomIdx = curRoomIdx - 1
    }
    if newRoomIdx >= strRoomIdLst.count {
        newRoomIdx = 0
    } else if newRoomIdx < 0 {
        newRoomIdx = strRoomIdLst.count - 1
    }
    let newRoomStrId = strRoomIdLst[newRoomIdx]

    let src = TRTCSwitchRoomConfig()
    src.strRoomId = newRoomStrId
    src.userSig = GenerateTestUserSig.genTestUserSig(identifier:
userId) as String
    if curIsSub {
        trtcCloud.switchRoom(src)
        trtcCloud.muteAllRemoteAudio(true)
    } else {
        subCloudHelper.trtcCloud.switchRoom(src)
        subCloudHelper.trtcCloud.muteAllRemoteAudio(true)
    }
}

extension RtcDuplexVC: UIPageViewControllerDelegate {
    func pageViewController(
        _ pageViewController: UIPageViewController,
        didFinishAnimating finished: Bool,
        previousViewControllers: [UIViewController],
        transitionCompleted completed: Bool
    ) {
        if completed {
            guard let currentVC =
pageViewController.viewControllers?.first else {return}
            if let index = pages.firstIndex(of: currentVC) {
                // 获取上下滑的判断依据
```

```
                let iden = index - curPageIdx
                // 更新当前显示页面的序号
                curPageIdx = index
                // 向下滑
                if iden == 1 || iden == -2 {
                    // 更新当前所在房间的序号
                    updateCurRoomIdx(isNext: true)
                    // 更新当前显示页面的实例
                    curIsSub.toggle()
                    // 更新房间
                    updateNewRoom(isNext: true)
                }
                // 向上滑
                if iden == -1 || iden == 2 {
                    // 更新房间
                    updateNewRoom(isNext: false)
                    // 更新当前所在房间的序号
                    updateCurRoomIdx(isNext: false)
                    // 更新当前显示页面的实例
                    curIsSub.toggle()
                    trtcCloud.muteAllRemoteAudio(true)
                    subCloudHelper.trtcCloud.muteAllRemoteAudio(true)
                }
                // 解除当前房间的静音
                if curIsSub {
                    subCloudHelper.trtcCloud.muteAllRemoteAudio(false)
                } else {
                    trtcCloud.muteAllRemoteAudio(false)
                }
            }
        }
    }
}
```

此外，业务上还可以区分“在滑动列表中”和“进入直播间”这2个状态，因为在上下滑动时一般不需要房间的详情和聊天等信息，“进入直播间”之后才需要显示，这样区分之后可以减轻频繁上下滑对业务记录直播间状态的压力，同时减少预加载对资源的消耗。

具体做法：只允许“在滑动列表中”时可以上下滑切换直播间，此时只显示直播间画面和少量信息； 在“进入直播间”时显示完整的直播间和聊天等信息，此时不允许上下滑切换直播间，同时在“进入直播间”时停止预加载下一个直播间的视频流，在回到“滑动列表中”同时恢复预加载下一个直播间的视频流，效果如下：

具体实现如下：

```
// 处理进入直播间逻辑
@objc func enterBtnClick() {
    // 隐藏上下滑列表中的UI组件
    self.enterBtn.isHidden = true
    // 显示进入直播间后的UI组件
    self.exitBtn.isHidden = false
    // ...

    self.atRoom = true
    // 进入直播间后禁止上下滑动
    pageViewController.dataSource = nil

    // 停止预加载
    if curIsSub{
        trtcCloud.muteAllRemoteVideoStreams(true)
    } else {
        subCloudHelper.trtcCloud.muteAllRemoteAudio(true)
    }
```

```
}

// 处理离开直播间逻辑
@objc func exitBtnClick() {
    // 隐藏直播间中的UI组件
    self.enterBtn.isHidden = false
    // 显示上下滑列表中的UI组件
    self.exitBtn.isHidden = true

    self.atRoom = false
    // 进入上下滑列表后恢复上下滑动
    pageViewController.dataSource = self
    // 恢复预加载
    if curIsSub{
        trtcCloud.muteAllRemoteVideoStreams(false)
    } else {
        subCloudHelper.trtcCloud.muteAllRemoteAudio(false)
    }
}
```

Android 端直播上下滑方案

以下的双实例和单实例章节中，其中的效果图和示例代码中，有三个页面，页面顺序是： A > B > C， A 对应房间1231，B 对应房间1232，C 对应房间1233。

实现方式	用户体验	资源消耗	实现逻辑
单实例	一般，滑动列表时，不能同时看到两个直播间，需页面完全切换后，才显示对应的直播间。可用占位图提升用户体验。	列表中只有一路流量消耗，有一个视频播放对象被使用。	简单，根据需要设置占位图。
双实例	较好，同时进入两个直播间，提前加载下一个直播，滑动列表时，能同时看到当前和下一个直播间。	列表中会有两路流量消耗，会产生两路的费用，有两个视频播放对象被使用。	复杂，需使用多个实例，并控制不同实例的音视频拉取。

单实例

直播列表上下滑动，在滑动过程中只能看到单个直播画面（单实例）， 可节省费用。

效果图

在 A 页滑动过程中，无法同时看到 B 页的直播画面，再切换到 B 页，可看到 B 页的直播画面，无法看到 A 页的直播画面。

方案原理

在页面滑动中，只能同时看到一个直播画面，在页面切换后，停止上一个直播画面，拉取下一个直播画面。
在从 A 页滑动到 B 页，各阶段的操作和状态如下：

1. 在 A 页显示到屏幕上时，用 TRTCCloud 实例1，进入房间1231并拉流，播放音视频，在 A 页面显示。
2. 从 A 页滑动到 B 页的过程中，没有收到切换到 B 页的回调，A 页面还是正常播放房间1231的音视频流，B 页面显示占位图或黑屏。
3. 从 A 页滑动到 B 页的过程中，收到切换到 B 页的回调，用 TRTCCloud 实例1，停止拉取房间1231的音视频流并退房，进入房间1232并拉流，播放音视频，在 B 页面显示，A 页面显示占位图或黑屏。

实现代码

使用 ViewPager2、RecyclerView.Adapter 来实现整屏滑动效果。在 ViewPager2的 registerOnPageChangeCallback 的 onPageSelected 回调中，停止拉流、退出房间、进入房间、开始拉流。下面给出 ScrollSwitchRoomActivity 的完整代码，布局文件与上节的相同。

ScrollSwitchRoomActivity 代码如下：

```
public class ScrollSwitchRoomActivity extends TRTCBaseActivity {


    PageAdapter mAdapter;
    public String[] mRoomIds;
    private TRTCCloud mTRTCCloud;
    private TXCloudVideoView mRemoteVideoView;
    private int mCurPos = -1;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_scroll_switch_room);

        //隐藏标题栏
        getSupportActionBar().hide();
        mRoomIds = new String[]{"1231", "1232", "1233"};

        if (checkPermission()) {
            initView();
        }

    }

    @Override
    protected void onPermissionGranted() {
        initView();
    }

    private void initView() {

        mAdapter = new PageAdapter(this, mRoomIds);
        ViewPager2 viewPager = findViewById(R.id.viewPager);
        viewPager.setAdapter(mAdapter);
        //设置 viewPager 滑动方向
        viewPager.setOrientation(ViewPager2.ORIENTATION_VERTICAL);
        // 设置 viewPager 预加载
        viewPager.setOffscreenPageLimit(1);
```

```
        // 添加页面切换监听
        viewPager.registerOnPageChangeCallback(new
ViewPager2.OnPageChangeCallback() {

            public void onPageSelected(int position) {
                Log.d("ScrollSwitchRoom", "onPageSelected: " +
position);

                if (mCurPos == position) {
                    return;
                }

                RecyclerView recyclerViewImpl = (RecyclerView)
viewPager.getChildAt(0);
                // 先退出当前房间
                exitRoom();
                // 进入下一个房间
                View itemView = recyclerViewImpl.getChildAt(position);
                mRemoteVideoView =
itemView.findViewById(R.id.txcvv_main_local);
                enterRoom(position);
                mCurPos = position;
            }
        });

        // Initialize your views here
        mTRTCCloud = TRTCCloud.sharedInstance(getApplicationContext());
        mTRTCCloud.addListener(mTRTCCloudListener);
    }

    private void enterRoom(int roomIdIndex) {
        TRTCCloudDef.TRTCParams mTRTCParams = new
TRTCCloudDef.TRTCParams();
        mTRTCParams.sdkAppId = GenerateTestUserSig.SDKAPPID;
        mTRTCParams.userId = "123";
        mTRTCParams.strRoomId = mRoomIds[roomIdIndex];
        mTRTCParams.userSig =
GenerateTestUserSig.genTestUserSig(mTRTCParams.userId);
        mTRTCParams.role = TRTCCloudDef.TRTCRoleAudience;
```

```
        mTRTCCloud.enterRoom(mTRTCParams,
TRTCCloudDef.TRTC_APP_SCENE_LIVE);
    }


    private TRTCCloudListener mTRTCCloudListener = new
TRTCCloudListener() {
        public void onEnterRoom(long result) {
            if (result == 0) {
                // Enter room success
            } else {
                // Enter room failed
            }
        }

        public void onExitRoom(int reason) {
            // Exit room
        }


        @Override
        public void onUserVideoAvailable(String userId, boolean
available) {
            super.onUserVideoAvailable(userId, available);
            if (available) {
                mTRTCCloud.startRemoteView(userId,
TRTCCloudDef.TRTC_VIDEO_STREAM_TYPE_BIG, mRemoteVideoView);
            } else {
                mTRTCCloud.stopRemoteView(userId,
TRTCCloudDef.TRTC_VIDEO_STREAM_TYPE_BIG);
            }
        }

        public void onError(int errCode, String errMsg, Bundle
extraInfo) {
            // print Error
            Log.e("ScrollSwitchRoom", "Error: " + errCode + " " +
errMsg);
        }
```

```
    };

    private void exitRoom() {

        mTRTCCloud.stopAllRemoteView();
        mTRTCCloud.exitRoom();
//        mTRTCCloud.setListener(null);
    }


    public class PageAdapter extends
RecyclerView.Adapter<PageAdapter.PageViewHolder> {

        private Context context;
        public String[] mRoomIds;

        public PageAdapter(Context context, String[] roomIds) {
            this.context = context;
            this.mRoomIds = roomIds;
        }

        public PageViewHolder onCreateViewHolder(@NonNull ViewGroup
parent, int viewType) {

            View view =
LayoutInflater.from(context).inflate(R.layout.item_scroll_page, parent,
false);
            return new PageViewHolder(view);
        }

        public void onBindViewHolder(@NonNull PageViewHolder holder, int
position) {
            TextView textView =
holder.itemView.findViewById(R.id.tv_room_number);
            textView.setText(getString(R.string.switchroom_roomid) + ":"
+ mRoomIds[position]);
        }

        public int getItemCount() {
            return mRoomIds.length;
```

```
        }

        public int getItemViewType(int position) {
            return position;
        }

        class PageViewHolder extends RecyclerView.ViewHolder {
            PageViewHolder(@NonNull View itemView) {
                super(itemView);
            }
        }
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        exitRoom();
    }
}
```

activity_scroll_switch_room.xml 如下：

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.viewpager2.widget.ViewPager2
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/viewPager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

item_scroll_page.xml 如下：

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/item_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
```

```
    <ImageView
        android:id="@+id/iv_placeholder"
        android:scaleType="centerCrop"
        android:background="@drawable/placeholder_img"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>

    <com.tencent.rtmp.ui.TXCloudVideoView
        android:id="@+id/txcvv_main_local"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <TextView
        android:id="@+id/tv_room_number"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintEnd_toEndOf="@id/item_layout"
        app:layout_constraintStart_toStartOf="@id/item_layout" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

双实例

直播列表上下滑动，两个 TRTCCloud 实例同时进入两个房间（当前房间和下一个房间），在滑动过程中可同时看到这两个房间的直播。如果从当前房间向下滑动的过程中，是不能同时看到上一个房间的直播，只有在页面完全切换后，才能看到。如果需要可自行使用三个 TRTCCloud 实例实现。

> **注意：**
> 双实例会同时拉取两个房间的流，会造成两路的流量消耗，会产生更多的费用。如果自实现三个 TRTCCloud 实例，则会有三路的流量消耗。

效果图

在 B 页滑动过程中，可直接看到 C 页的直播画面。

方案原理

在页面滑动中，最多可同时看到两个页面，使用两个 TRTCCloud 实例，同时进两个房间，同时拉取两个房间的流。
在从 A 页滑动到 B 页，各阶段的操作和状态如下：

1. 在 A 页显示到屏幕上时，用 TRTCCloud 实例1，进入房间1231并拉流，播放音视频，在 A 页面显示。同时用 TRTCCloud 实例2，进入房间1232并拉流，播放视频，在 B 页面显示，静音音频。
2. 从 A 页滑动到 B 页的过程中，此时可同时看到 A、B 页面在同时播放视频，听到房间1231的音频。
3. B 页完全显示后，用 TRTCCloud 实例2，开启房间1232的音频，视频继续播放。用 TRTCCloud 实例1，退出房间1231，进入房间1233并拉流，播放视频，在 C 页面显示，静音音频。
4. 从 B 页滑动到 A 页的过程中，只能看到房间1232的视频，因为此时 TRTCCloud 实例2在 B 页面使用，TRTCCloud 实例1在 C 页面使用。在 A 页面完全显示后，用 TRTCCloud 实例1，进入房间1231并拉流，播放音视频，B 页面继续用 TRTCCloud 实例2拉取视频流，静音音频。

实现代码

使用 ViewPager2、RecyclerView.Adapter 来实现整屏滑动效果。ScrollSwitchRoomActivity 代码如下：

```
public class ScrollSwitchRoomDualActivity extends TRTCBaseActivity {

    PageAdapter mAdapter;

    public String[] mRoomIds;

    private TRTCCloud mTRTCCloud;
    private TRTCCloud mSubCloud;
    private TXCloudVideoView mRemoteVideoView;
    private TXCloudVideoView mSubRemoteVideoView;

    private Boolean mIsInMainRoom = null;

    private int mCurPos = 0;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_scroll_switch_room);

        //隐藏标题栏
        getSupportActionBar().hide();
        mRoomIds = new String[]{"1231", "1232", "1233"};

        if (checkPermission()) {
            initView();
        }

    }

    @Override
    protected void onPermissionGranted() {
        initView();
    }

    private void initView() {

        mAdapter = new PageAdapter(this,mRoomIds);
```

```
        ViewPager2 viewPager = findViewById(R.id.viewPager);
        viewPager.setAdapter(mAdapter);

        //设置 viewPager 滑动方向
        viewPager.setOrientation(ViewPager2.ORIENTATION_VERTICAL);
        // 设置 viewPager 预加载
        viewPager.setOffscreenPageLimit(1);

        // 添加页面切换监听
        viewPager.registerOnPageChangeCallback(new
ViewPager2.OnPageChangeCallback() {
            public void onPageSelected(int position) {

                Log.d("ScrollSwitchRoom", "onPageSelected: " +
position);
                RecyclerView recyclerViewImpl = (RecyclerView)
viewPager.getChildAt(0);
                if (mIsInMainRoom == null) {
                    View itemView =
recyclerViewImpl.getChildAt(position);
                    mRemoteVideoView =
itemView.findViewById(R.id.txcvv_main_local);
                    View subItemView =
recyclerViewImpl.getChildAt(position + 1);
                    mSubRemoteVideoView =
subItemView.findViewById(R.id.txcvv_main_local);
                    enterRoom();
                    mIsInMainRoom = true;
                } else {

                    if (mIsInMainRoom) {
                        mTRTCCloud.muteAllRemoteAudio(true);
                        mSubCloud.muteAllRemoteAudio(false);
                    } else {
                        mTRTCCloud.muteAllRemoteAudio(false);
                        mSubCloud.muteAllRemoteAudio(true);
                    }

                    if (position != (mRoomIds.length - 1)) {
                        String roomId;
```

```
                        TRTCCloud trtcCloud;
                        if (mCurPos < position) {
                            // 屏幕向上滑
                            roomId = mRoomIds[position + 1];
                            trtcCloud = mIsInMainRoom ? mTRTCCloud :
mSubCloud;
                            View itemView =
recyclerViewImpl.getChildAt(position + 1);
                            if (mIsInMainRoom) {
                                mRemoteVideoView =
itemView.findViewById(R.id.txcvv_main_local);
                            } else {
                                mSubRemoteVideoView =
itemView.findViewById(R.id.txcvv_main_local);
                            }

                        } else {
                            //屏幕向下滑
                            roomId = mRoomIds[position];
                            trtcCloud = mIsInMainRoom ? mSubCloud :
mTRTCCloud;

                            View itemView =
recyclerViewImpl.getChildAt(position);
                            if (mIsInMainRoom) {
                                mSubRemoteVideoView =
itemView.findViewById(R.id.txcvv_main_local);
                            } else {
                                mRemoteVideoView =
itemView.findViewById(R.id.txcvv_main_local);
                            }
                        }
                        switchRoom(roomId, trtcCloud);
                    }

                    mIsInMainRoom = !mIsInMainRoom;
                }

                mCurPos = position;
```

```
            }
        });

        // Initialize your views here
        mTRTCCloud = TRTCCloud.sharedInstance(getApplicationContext());
        mSubCloud = mTRTCCloud.createSubCloud();
    }

    /**
     * 只有在初始化时，才调用进入房间，后续切换房间，调用 switchRoom
     */
    private void enterRoom() {

        mTRTCCloud.addListener(mTRTCCloudListener);
        mSubCloud.addListener(mSubCloudListener);
        TRTCCloudDef.TRTCParams mTRTCParams = new
TRTCCloudDef.TRTCParams();
        mTRTCParams.sdkAppId = GenerateTestUserSig.SDKAPPID;
        mTRTCParams.userId = "123";
//        mTRTCParams.roomId = Integer.parseInt(roomId);
        mTRTCParams.strRoomId = mRoomIds[0];
        mTRTCParams.userSig =
GenerateTestUserSig.genTestUserSig(mTRTCParams.userId);
        mTRTCParams.role = TRTCCloudDef.TRTCRoleAudience;
        mTRTCCloud.enterRoom(mTRTCParams,
TRTCCloudDef.TRTC_APP_SCENE_LIVE);

        mTRTCParams.strRoomId = mRoomIds[1];
        mSubCloud.muteAllRemoteAudio(true);
        mSubCloud.enterRoom(mTRTCParams,
TRTCCloudDef.TRTC_APP_SCENE_LIVE);

    }


    private TRTCCloudListener mTRTCCloudListener = new
TRTCCloudListener() {
        public void onEnterRoom(long result) {
            if (result == 0) {
                // Enter room success
```

```
            } else {
                // Enter room failed
            }
        }

        public void onExitRoom(int reason) {
            // Exit room
        }


        @Override
        public void onUserVideoAvailable(String userId, boolean
available) {
            super.onUserVideoAvailable(userId, available);
            if (available) {
                mTRTCCloud.startRemoteView(userId,
TRTCCloudDef.TRTC_VIDEO_STREAM_TYPE_BIG, mRemoteVideoView);
            } else {
                mTRTCCloud.stopRemoteView(userId,
TRTCCloudDef.TRTC_VIDEO_STREAM_TYPE_BIG);
            }
        }

        public void onError(int errCode, String errMsg, Bundle
extraInfo) {
            // print Error
            Log.e("ScrollSwitchRoom", "Error: " + errCode + " " +
errMsg);


        }

        public void onSwitchRoom(long err, String errMsg) {
            // Switch room
        }
    };


    private TRTCCloudListener mSubCloudListener = new
TRTCCloudListener() {
```

```
        public void onEnterRoom(long result) {
            if (result == 0) {
                // Enter room success
            } else {
                // Enter room failed
            }
        }

        public void onExitRoom(int reason) {
            // Exit room
        }

        @Override
        public void onUserVideoAvailable(String userId, boolean
available) {
            super.onUserVideoAvailable(userId, available);
            if (available) {
                mSubCloud.startRemoteView(userId,
TRTCCloudDef.TRTC_VIDEO_STREAM_TYPE_BIG, mSubRemoteVideoView);
            } else {
                mSubCloud.stopRemoteView(userId,
TRTCCloudDef.TRTC_VIDEO_STREAM_TYPE_BIG);
            }
        }

    };

    private void exitRoom() {

        mTRTCCloud.stopAllRemoteView();
        mTRTCCloud.exitRoom();
        mTRTCCloud.setListener(null);

        mSubCloud.stopAllRemoteView();
        mSubCloud.exitRoom();
        mSubCloud.setListener(null);
    }


    private void switchRoom(String roomId, TRTCCloud trtcCloud) {
```

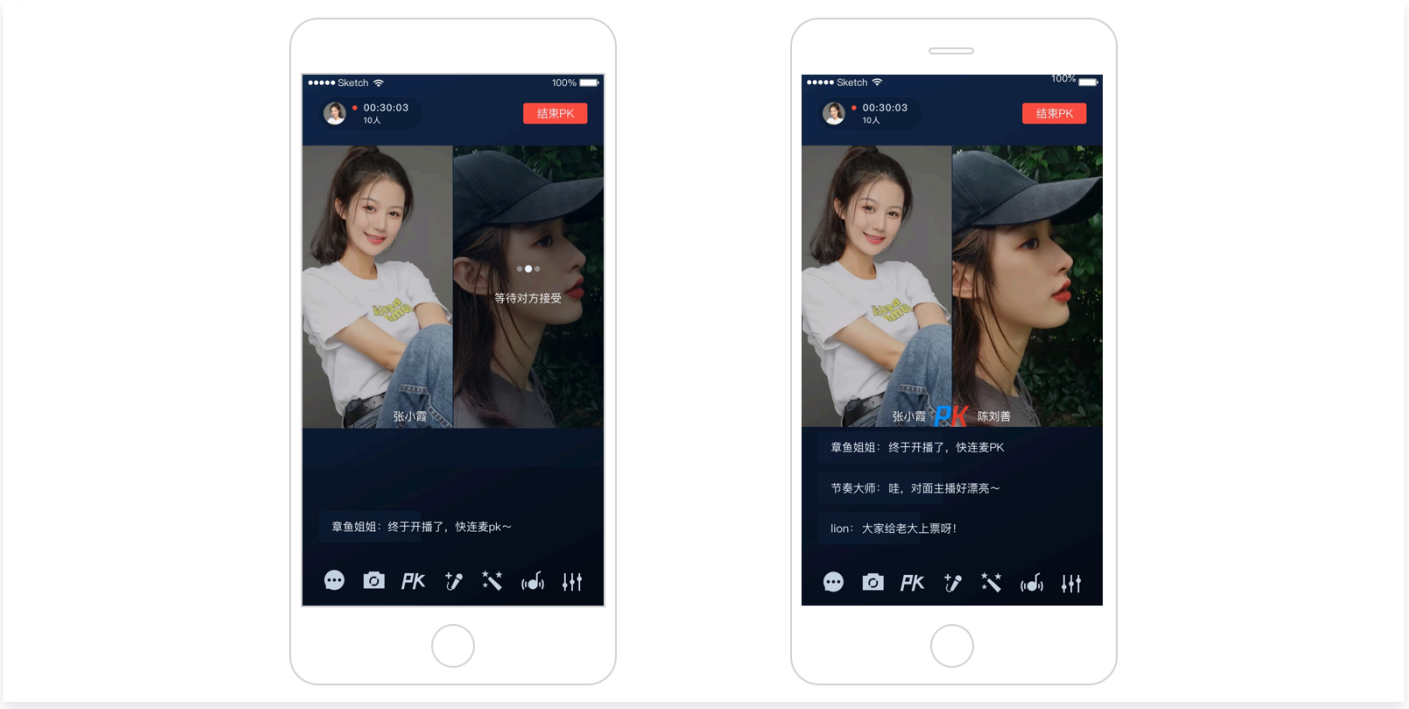
```
        TRTCCloudDef.TRTCSwitchRoomConfig config = new
TRTCCloudDef.TRTCSwitchRoomConfig();
//        config.roomId = Integer.parseInt(roomId);
        config.strRoomId = roomId;
        trtcCloud.switchRoom(config);
    }


    public class PageAdapter extends
RecyclerView.Adapter<PageAdapter.PageViewHolder> {


        private Context context;
        public String[] mRoomIds;


        public PageAdapter(Context context, String[] roomIds) {
            this.context = context;
            this.mRoomIds = roomIds;
        }

        public PageViewHolder onCreateViewHolder(@NonNull ViewGroup
parent, int viewType) {

            View view =
LayoutInflater.from(context).inflate(R.layout.item_scroll_page, parent,
false);
            return new PageViewHolder(view);
        }

        public void onBindViewHolder(@NonNull PageViewHolder holder, int
position) {
            TextView textView =
holder.itemView.findViewById(R.id.tv_room_number);
            textView.setText(getString(R.string.switchroom_roomid) + ":"
+ mRoomIds[position]);
        }

        public int getItemCount() {
            return mRoomIds.length;
```

```
        }

        public int getItemViewType(int position) {
            return position;
        }

        class PageViewHolder extends RecyclerView.ViewHolder {
            PageViewHolder(@NonNull View itemView) {
                super(itemView);
            }


        }
    }


    @Override
    protected void onDestroy() {
        super.onDestroy();
        exitRoom();
    }
}
```

跨房 PK 连麦方案

最近更新时间：2025-11-18 15:15:33

直播间里，为了增进直播气氛、快速吸粉，主播可以邀请其他直播间的主播进行连麦互动或在线 PK。连麦直播间内的观众可以同时收听或观看多个主播互动，能够增强互动直播的趣味性，激发观众刷榜送礼物的欲望。下面介绍三种不同跨房 PK 连麦方案的具体实现。



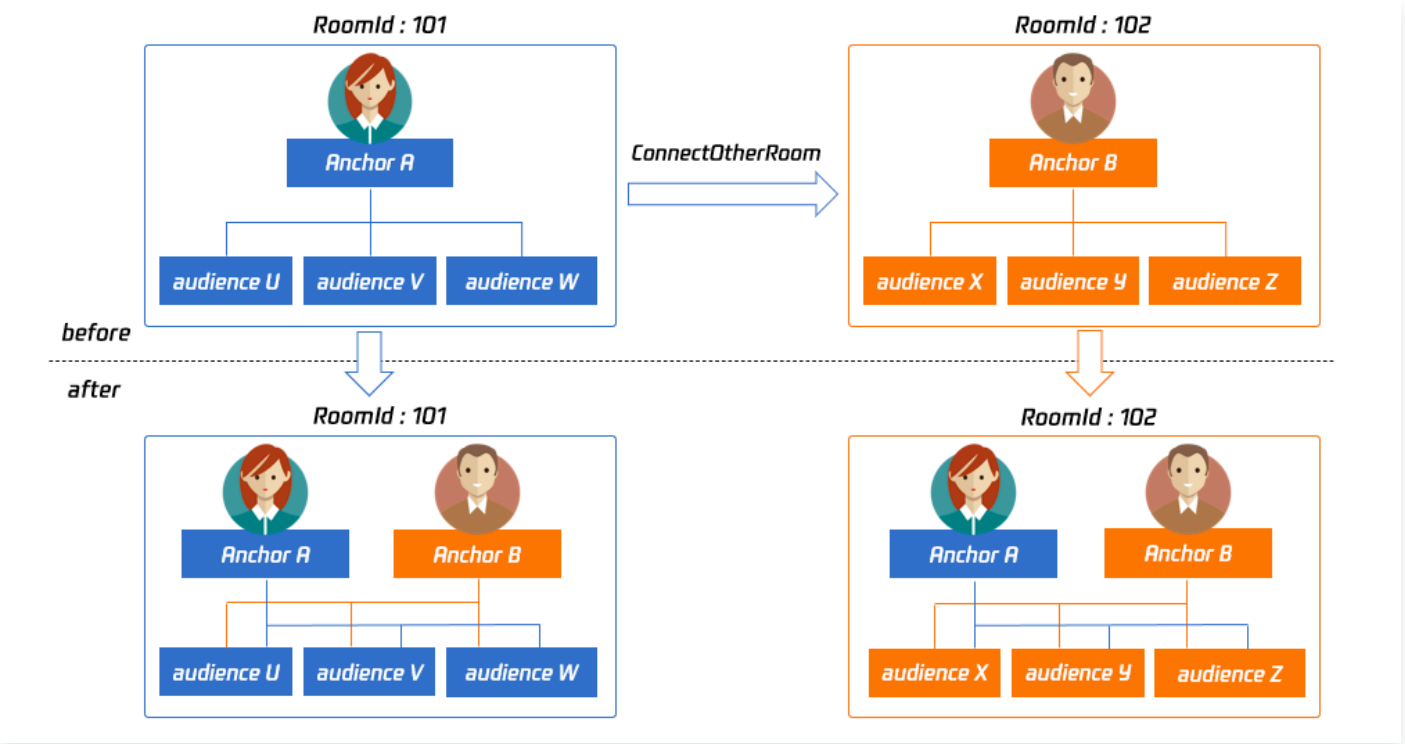
常规跨房 PK 连麦方案

适用场景

两个或多个房间 PK，房间主播数量较少的简单跨房连麦场景。

方案原理

默认情况下，只有同一个房间内的用户之间音视频可以互通，不同的房间之间的音视频流是相互隔离的。通过跨房连麦，可以将另一个房间中某个主播音视频流发布到自己所在的房间中，与此同时也会将自己的音视频流发布到目标主播的房间中。让身处两个不同房间中的主播进行跨房间的音视频流分享，从而让每个房间中的观众都能观看到这两个主播的音视频。



实现流程

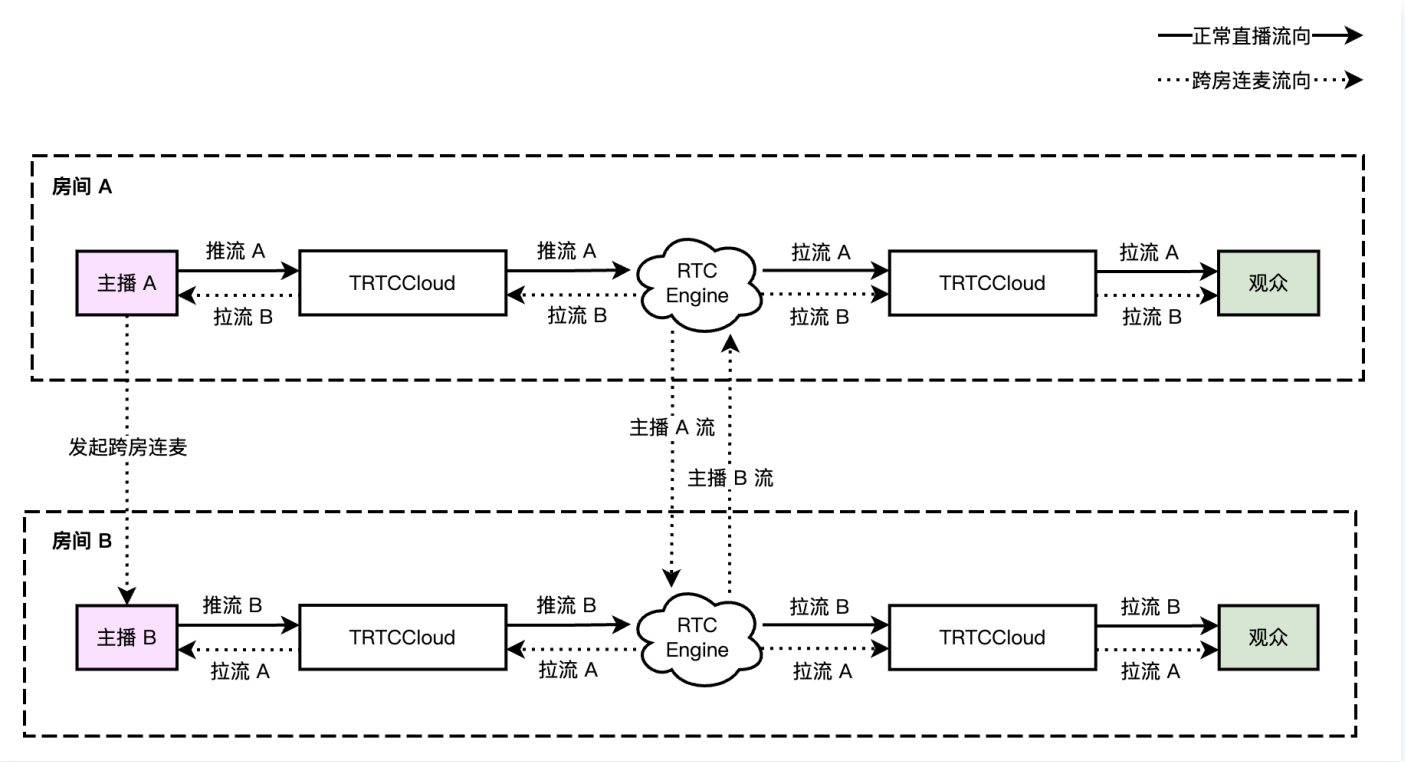
当房间“101”中的主播 A 通过 ConnectOtherRoom 跟房间“102”中的主播 B 建立跨房通话后：

- 房间“101”中的用户都会收到主播 B 的 `onRemoteUserEnterRoom(B)` 和 `onUserVideoAvailable(B,true)` 这两个事件回调，即房间“101”中的用户都可以订阅主播 B 的音视频。
- 房间“102”中的用户都会收到主播 A 的 `onRemoteUserEnterRoom(A)` 和 `onUserVideoAvailable(A,true)` 这两个事件回调，即房间“102”中的用户都可以订阅主播 A 的音视频。

> ⚠ 注意：
>
> - 两个房间单主播跨房 PK，只需要其中一个房间的主播调用 `ConnectOtherRoom` 建立跨房连麦即可，请勿双向调用。
> - 主播可通过多次调用 `ConnectOtherRoom` 与多个房间的主播建立跨房连麦，目前限制单个主播最多和其他房间的 9 个主播进行跨房连麦。

实时互动跨房连麦

RTC 场景下的跨房连麦 PK 流程整体简单，主播和跨房连麦主播互相拉取 RTC 单流，观众同时拉取主播和跨房连麦主播的 RTC 单流，观众可独立控制主播和跨房连麦主播媒体流订阅逻辑。实时互动场景跨房连麦流程如下图所示。



> **注意：**
> 实时互动跨房连麦场景下，房间内观众可独立控制跨房连麦主播媒体流的订阅逻辑，也可由主播 更改跨房主播在本房间的上行能力 。

示例代码

1. 任意一方发起跨房 PK 连麦。

Android

```
public void connectOtherRoom(String roomId, String userId) {
    try {
        JSONObject jsonObj = new JSONObject();
        // 以字符串房间号为例，数字房间号 key:roomId
        jsonObj.put("strRoomId", roomId);
        jsonObj.put("userId", userId);
        mTRTCCloud.ConnectOtherRoom(jsonObj.toString());
    } catch (JSONException e) {
        e.printStackTrace();
    }
}
```

```
// 请求跨房连麦的结果回调
@Override
public void onConnectOtherRoom(String userId, int errCode, String
errMsg) {
    // userId:  要跨房连麦的另一个房间中的主播的用户 ID
    // errCode: 错误码，ERR_NULL 代表请求成功
    // errMsg:  错误信息
}
```

iOS

```
- (void)connectOtherRoom:(NSString *)roomId {
    NSMutableDictionary *jsonDict = [[NSMutableDictionary alloc] init];
    // 以字符串房间号为例，数字房间号 key:roomId
    [jsonDict setObject:roomId forKey:@"strRoomId"];
    [jsonDict setObject:self.userId forKey:@"userId"];
    NSData *jsonData = [NSJSONSerialization dataWithJSONObject:jsonDict
options:NSJSONWritingPrettyPrinted error:nil];
    NSString *jsonString = [[NSString alloc] initWithData:jsonData
encoding:NSUTF8StringEncoding];
    [self.trtcCloud connectOtherRoom:jsonString];
}

// 请求跨房连麦的结果回调
- (void)onConnectOtherRoom:(NSString *)userId errCode:
(TXLiteAVError)errCode errMsg:(NSString *)errMsg {
    // userId:  要跨房连麦的另一个房间中的主播的用户 ID
    // errCode: 错误码，ERR_NULL 代表请求成功
    // errMsg:  错误信息
}
```

⚠ 注意：

- 跨房 PK 连麦的本地用户和对端用户必须都为主播角色，且必须都有音频或视频上行。
- 两个房间单主播跨房 PK，只需要其中一个房间的主播调用 `ConnectOtherRoom` 建立跨房连麦即可，请勿双向调用。

2. 两个房间中的所有用户都会收到来自另一个房间中的 PK 主播的音视频流可用回调。

Android

```
@Override
public void onUserAudioAvailable(String userId, boolean available) {
    // 某远端用户发布/取消了自己的音频
    // 在自动订阅模式下，您无需做任何操作，SDK 会自动播放远端用户音频
}

@Override
public void onUserVideoAvailable(String userId, boolean available) {
    // 某远端用户发布/取消了主路视频画面
    if (available) {
        // 订阅远端用户的视频流，并绑定视频渲染控件
        mTRTCCloud.startRemoteView(userId,
TRTCCloudDef.TRTC_VIDEO_STREAM_TYPE_BIG, view);
    } else {
        // 停止订阅远端用户的视频流，并释放渲染控件
        mTRTCCloud.stopRemoteView(userId,
TRTCCloudDef.TRTC_VIDEO_STREAM_TYPE_BIG);
    }
}
```

iOS

```
- (void)onUserAudioAvailable:(NSString *)userId available:
(BOOL)available {
    // 某远端用户发布/取消了自己的音频
    // 在自动订阅模式下，您无需做任何操作，SDK 会自动播放远端用户音频
}

- (void)onUserVideoAvailable:(NSString *)userId available:
(BOOL)available {
    // 某远端用户发布/取消了主路视频画面
    if (available) {
        // 订阅远端用户的视频流，并绑定视频渲染控件
        [self.trtcCloud startRemoteView:userId
streamType:TRTCVideoStreamTypeBig view:self.remoteView];
    } else {
```

```
        // 停止订阅远端用户的视频流，并释放渲染控件
        [self.trtcCloud stopRemoteView:userId
streamType:TRTCVideoStreamTypeBig];
    }
}
```

3. 任意一方退出跨房 PK 连麦。

Android

```
// 退出跨房连麦
mTRTCCloud.DisconnectOtherRoom();

// 退出跨房连麦的结果回调
@Override
public void onDisConnectOtherRoom(int errCode, String errMsg) {
    super.onDisConnectOtherRoom(errCode, errMsg);
}
```

iOS

```
// 退出跨房连麦
[self.trtcCloud disconnectOtherRoom];

// 退出跨房连麦的结果回调
- (void)onDisconnectOtherRoom:(TXLiteAVError)errCode errMsg:(NSString
*)errMsg {
}
```

> ⚠ **注意：**
> 跨房 PK 连麦的发起端和接收端任意一端均可调用 DisconnectOtherRoom 结束跨房 PK 连麦。

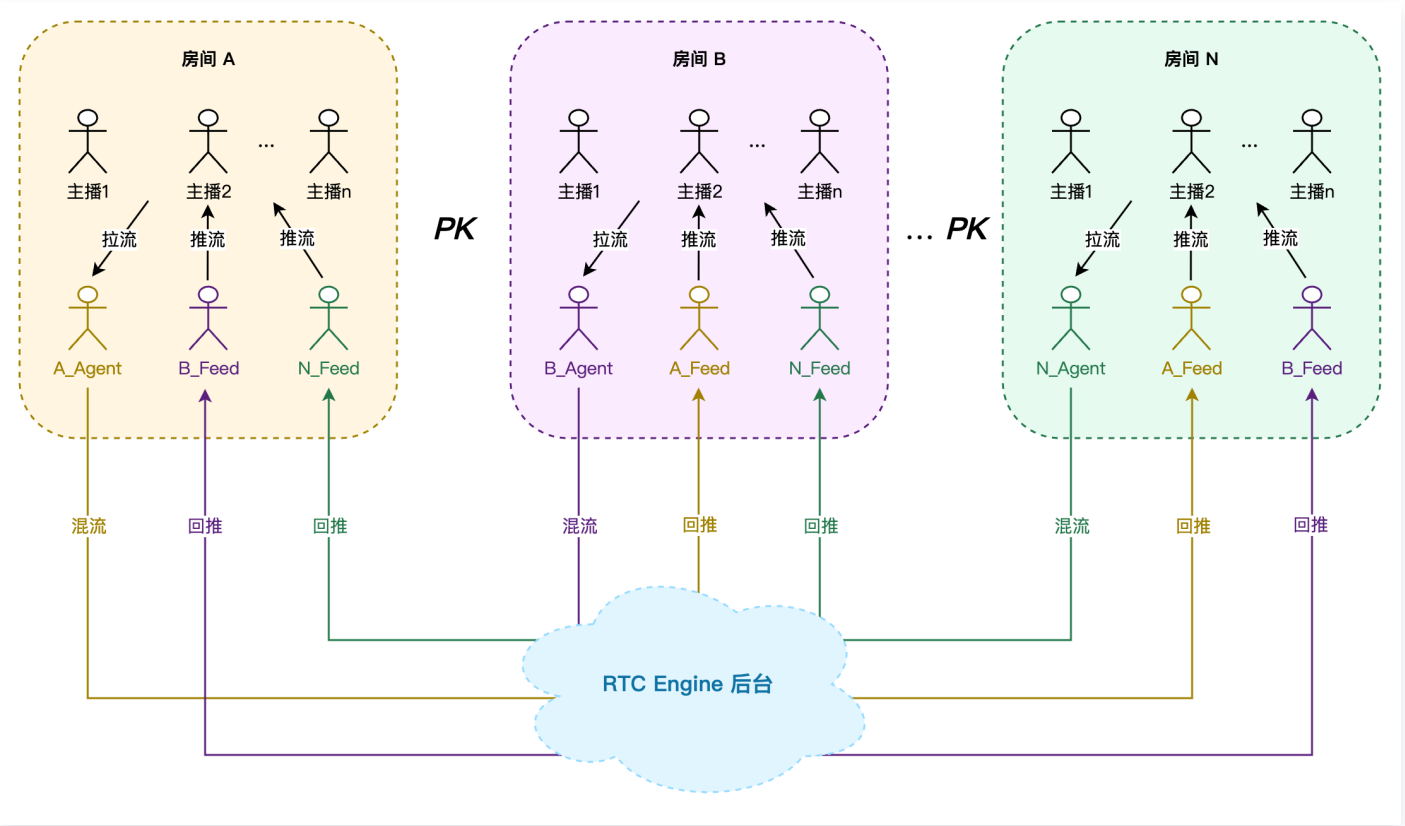
服务端跨房 PK 连麦方案

适用场景

多个房间 PK，每个房间有多个主播的纯服务端跨房连麦场景。

方案原理

服务端启动多个混流转推任务，每个转推任务都会拉起一个 Agent 机器人用户进入己方 RTC Engine 房间进行拉流，同时会拉起一个或多个 Feed 机器人用户将混合的音视频流回推到其他参与跨房 PK 连麦的 RTC Engine 房间。这样不同房间里的用户就可以通过订阅其他房间混流回推的音视频流，从而实现跨房 PK 连麦。



实现流程

实时互动跨房连麦

1. 房间 A 主播向房间 B 主播和房间 N 主播发起跨房 PK 请求（业务信令）。
2. 房间 B 主播和房间 N 主播同意跨房 PK 请求（业务信令）。
3. 业务后台同时启动 N 个混流回推房间任务 StartPublishCdnStream 。
   - 任务一：A_Agent 机器人接收 A 房间主播媒体流，经 RTC Engine 后台混流后由 A_Feed 机器人回推到 B 房间和 N 房间。
   - 任务二：B_Agent 机器人接收 B 房间主播媒体流，经 RTC Engine 后台混流后由 B_Feed 机器人回推到 A 房间和 N 房间。
   - 任务 N：N_Agent 机器人接收 N 房间主播媒体流，经 RTC Engine 后台混流后由 N_Feed 机器人回推到 A 房间和 B 房间。
4. 房间 A、房间 B、房间 N 的用户互相拉取房间中混流回推的音视频流，开始进行跨房 PK。
5. 跨房 PK 结束，业务后台通过 TaskId 停止 N 个混流回推房间任务 StopPublishCdnStream 。

> ⚠ 注意：
> - 本方案最多支持 11 个房间同时进行跨房 PK 连麦，每个房间最多支持 16 个主播同时参与连麦。
> - 机器人 ID 不能与房间内的普通用户 ID 冲突，否则会导致转推任务由于机器人用户被踢出 RTC Engine 房间而异常结束。

示例代码

下面以纯音频场景为例，展示跨房 PK 混流回推房间任务的参数体示例。

任务一

```
{
  "SdkAppId": 1400000000,
  "RoomId": "A",
  "RoomIdType": 1,
  "AgentParams": {
    "UserId": "A_Agent",
    "UserSig": "eJwtjMEKgkAUAP9lz2Hv6b40oU...",
    "MaxIdleTime": 50
  },
  "WithTranscoding": 1,
  "AudioParams": {
    "AudioEncode": {
      "Codec": 0,
      "SampleRate": 48000,
      "Channel": 2,
      "BitRate": 64
    }
  },
  "FeedBackRoomParams": [
    {
      "RoomId": "B",
      "RoomIdType": 1,
      "UserId": "A_Feed",
      "UserSig": "eJwtzEELgkAUBOD-sldD3745..."
    },
    {
      "RoomId": "N",
      "RoomIdType": 1,
```

```
      "UserId": "A_Feed",
      "UserSig": "eJwtzEELgkAUBOD-sldD3745..."
    }
  ]
}
```

任务二

```
{
  "SdkAppId": 1400000000,
  "RoomId": "B",
  "RoomIdType": 1,
  "AgentParams": {
    "UserId": "B_Agent",
    "UserSig": "eJwtjMEKgkAUAP9lz2Hv6b40oU...",
    "MaxIdleTime": 50
  },
  "WithTranscoding": 1,
  "AudioParams": {
    "AudioEncode": {
      "Codec": 0,
      "SampleRate": 48000,
      "Channel": 2,
      "BitRate": 64
    }
  },
  "FeedBackRoomParams": [
    {
      "RoomId": "A",
      "RoomIdType": 1,
      "UserId": "B_Feed",
      "UserSig": "eJwtzEELgkAUBOD-sldD3745..."
    },
    {
      "RoomId": "N",
      "RoomIdType": 1,
      "UserId": "B_Feed",
      "UserSig": "eJwtzEELgkAUBOD-sldD3745..."
    }
```

```
  ]
}
```

任务 N

```
{
  "SdkAppId": 1400000000,
  "RoomId": "N",
  "RoomIdType": 1,
  "AgentParams": {
    "UserId": "N_Agent",
    "UserSig": "eJwtjMEKgkAUAP9lz2Hv6b40oU...",
    "MaxIdleTime": 50
  },
  "WithTranscoding": 1,
  "AudioParams": {
    "AudioEncode": {
      "Codec": 0,
      "SampleRate": 48000,
      "Channel": 2,
      "BitRate": 64
    }
  },
  "FeedBackRoomParams": [
    {
      "RoomId": "A",
      "RoomIdType": 1,
      "UserId": "N_Feed",
      "UserSig": "eJwtzEELgkAUBOD-sldD3745..."
    },
    {
      "RoomId": "B",
      "RoomIdType": 1,
      "UserId": "N_Feed",
      "UserSig": "eJwtzEELgkAUBOD-sldD3745..."
    }
  ]
}
```

> ⚠ 注意：
> 纯音频场景下，RTC Engine 后台会默认混合房间内所有主播音频流，亦可通过音频参数 McuAudioParams 指定音频混流黑白名单。

跨房 PK 连麦方案对比分析

上面介绍了三种不同的跨房 PK 连麦实现方案，它们各自有不同的适用场景。下面分四个维度对不同的跨房连麦方案进行对比分析。

方案类型	方案优势	方案劣势	房间及人数限制	推荐的使用场景
常规跨房 PK 连麦方案	两人 PK 调用逻辑简单	多人 PK 调用逻辑复杂	单个主播最多和其他房间的 9 个主播跨房 PK	两个房间，单主播（双人）跨房 PK
服务端跨房 PK 连麦方案	纯服务端方案，客户端无需额外处理	存在额外的机器人推拉流及混流费用	最多支持 11 个房间同时进行跨房 PK，每个房间最多支持 16 个主播同时参与跨房 PK	多个房间，多主播（多人）跨房 PK，纯服务端管理

AI 对话 IM 信令方案

最近更新时间：2025-11-18 15:15:34

IM SDK 集成

iOS

集成 IM SDK

建议选择使用 CocoaPods 自动加载的方式集成 IM SDK。

1. 安装 CocoaPods。在终端窗口中输入如下命令（需要提前在 Mac 中安装 Ruby 环境）：

```
sudo gem install cocoapods
```

2. 创建 Podfile 文件。进入项目所在路径输入以下命令行，之后项目路径下会出现一个 Podfile 文件。

```
pod init
```

3. 编辑 Podfile 文件。请您按照如下方式设置 Podfile 文件：

```
platform :ios, '8.0'
source 'https://github.com/CocoaPods/Specs.git'

target 'App' do
    # 集成完整版本的 IM SDK，版本号要大于 '8.1.6129'
    pod 'TXIMSDK_Plus_iOS','8.1.6129'
    # 或者集成裁剪体积后的 IM SDK（仅包含 AI 信令相关能力），版本号要大于
'8.2.6361'
    pod 'TXIMSDK_Plus_SignalingSDK','8.2.6361'
end
```

4. 更新并安装 SDK。

在终端窗口中输入如下命令以更新本地库文件，并安装 IM SDK。

```
pod install
```

或使用以下命令更新本地库版本：

```
pod update
```

> ⚠ 注意：
> 若您进行上述操作后仍遇到问题，请参见 Xcode 集成常见问题 文档。

引用 IM SDK

项目代码中使用 SDK 有两种方式：

- 在 `Xcode > Build Setting > Header Search Paths` 设置 SDK 头文件的路径，然后在项目需要使用 SDK API 的文件里，引入具体的头文件。

```
#import "ImSDK_Plus.h"
```

- 在项目需要使用 SDK API 的文件里，引入具体的头文件。

```
#import <ImSDK_Plus/ImSDK_Plus.h>
```

Android

集成 SDK（aar）

建议选择使用 Gradle 自动加载的方式集成 IM SDK。

1. 添加 SDK 依赖。

1.1 找到 app 的 build.gradle，在 repositories 中添加 mavenCentral() 的依赖。

```
repositories {
    google()
    // 增加 mavenCentral 仓库
    mavenCentral()
}
```

1.2 然后在 dependencies 中添加 IM SDK 的依赖。

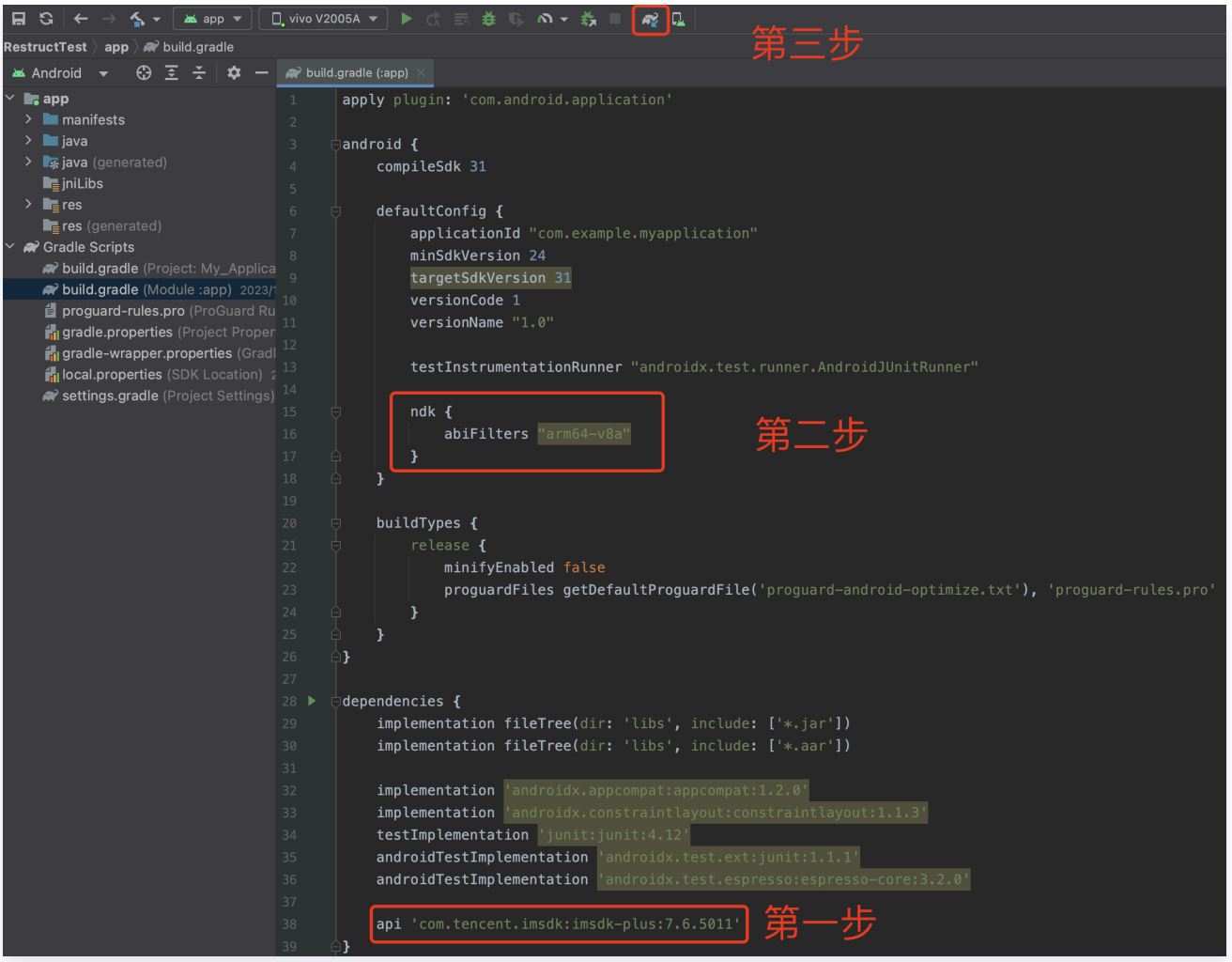
```
dependencies {
```

```
    // 集成完整版本的 IM SDK，版本号要大于 '8.1.6129'
    api 'com.tencent.imsdk:imsdk-plus:8.1.6129'

    // 或者集成裁剪体积后的 IM SDK（仅包含 AI 信令相关能力），版本号要大于
'8.2.6361'
    api 'com.tencent.imsdk:signalingsdk:8.2.6361'
}
```

2. 指定 App 使用架构。在 defaultConfig 中，指定 App 使用的 CPU 架构（从 IM SDK 4.3.118版本开始支持 armeabi-v7a，arm64-v8a，x86，x86_64）。

```
defaultConfig {
    ndk {
      abiFilters "arm64-v8a"
     }
 }
```

3. 同步 SDK。请保证您的网络已连接 maven，单击 Sync 按钮，SDK 就会自动下载集成到工程里。



配置 App 权限

在 AndroidManifest.xml 中配置 App 的权限，IM SDK 需要以下权限：

```
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission
android:name="android.permission.ACCESS_WIFI_STATE" />
```

设置混淆规则

在 proguard-rules.pro 文件，将 IM SDK 相关类加入不混淆名单。

```
-keep class com.tencent.imsdk.** { *; }
```

Web & 小程序

集成 SDK

建议通过 npm 方式将 IM SDK 集成到您的 Web、小程序中。

```
// 版本号v3.4.5或更高
npm install @tencentcloud/chat
```

> 说明：
> 若同步依赖过程中出现问题，请切换 npm 源后再次重试。

```
npm config set registry http://r.cnpmjs.org/
```

引入模块

```
import TencentCloudChat from '@tencentcloud/chat';
```

初始化 SDK

iOS

1. 调用初始化接口。

```
// 1. 从即时通信 IM 控制台获取应用 SDKAppID。
// 2. 初始化 config 对象
V2TIMSDKConfig *config = [[V2TIMSDKConfig alloc] init];
// 3. 指定 log 输出级别。
config.logLevel = V2TIM_LOG_INFO;
// 4. 添加 V2TIMSDKListener 的事件监听器，self 是 id<V2TIMSDKListener>
的实现类，如果您不需要监听 IM SDK 的事件，这个步骤可以忽略。
[[V2TIMManager sharedInstance] addIMSDKListener:self];
// 5. 初始化 IM SDK，调用这个接口后，可以立即调用登录接口。
[[V2TIMManager sharedInstance] initSDK:sdkAppID config:config];
```

2. 登录。

```
NSString *userID = @"your user id";
NSString *userSig = @"userSig from your server";
[[V2TIMManager sharedInstance] login:userID userSig:userSig succ:^{
    NSLog(@"success");
} fail:^(int code, NSString *desc) {
    // 如果返回以下错误码，表示使用 UserSig 已过期，请您使用新签发的 UserSig 进行再次登录。
    // 1. ERR_USER_SIG_EXPIRED (6206)
    // 2. ERR_SVR_ACCOUNT_USERSIG_EXPIRED (70001)
    // 注意：其他的错误码，请不要在这里调用登录接口，避免 IM SDK 登录进入死循环。
    NSLog(@"failure, code:%d, desc:%@", code, desc);
}];
```

Android

1. 调用初始化接口。

```
// 1. 从即时通信 IM 控制台获取应用 SDKAppID。
// 2. 初始化 config 对象。
V2TIMSDKConfig config = new V2TIMSDKConfig();
// 3. 指定 log 输出级别。
config.setLogLevel(V2TIMSDKConfig.V2TIM_LOG_INFO);
// 4. 添加 V2TIMSDKListener 的事件监听器，sdkListener 是 V2TIMSDKListener 的实现类，如果您不需要监听 IM SDK 的事件，这个步骤可以忽略。
V2TIMManager.getInstance().addIMSDKListener(sdkListener);
// 5. 初始化 IM SDK，调用这个接口后，可以立即调用登录接口。
V2TIMManager.getInstance().initSDK(context, sdkAppID, config);
```

2. 登录。

```
String userID = "your user id";
String userSig = "userSig from your server";
V2TIMManager.getInstance().login(userID, userSig, new
V2TIMCallback() {
```

```
    @Override
    public void onSuccess() {
        Log.i("imsdk", "success");
    }

    @Override
    public void onError(int code, String desc) {
        // 如果返回以下错误码，表示使用 UserSig 已过期，请您使用新签发的
UserSig 进行再次登录。
        // 1. ERR_USER_SIG_EXPIRED (6206)
        // 2. ERR_SVR_ACCOUNT_USERSIG_EXPIRED (70001)
        // 注意：其他的错误码，请不要在这里调用登录接口，避免 IM SDK 登录进入
死循环。
        Log.i("imsdk", "failure, code:" + code + ", desc:" + desc);
    }
});
```

Web & 小程序

1. 调用初始化接口。

```
import TencentCloudChat from '@tencentcloud/chat';

let options = {
  SDKAppID: 0 // 接入时需要将0替换为您的即时通信 IM 应用的 SDKAppID
};
// 创建 SDK 实例，`TencentCloudChat.create()`方法对于同一个 `SDKAppID` 只
会返回同一份实例
let chat = TencentCloudChat.create(options); // SDK 实例通常用 chat 表
示

chat.setLogLevel(0); // 普通级别，日志量较多，接入时建议使用
// chat.setLogLevel(1); // release 级别，SDK 输出关键信息，生产环境时建议
使用
```

2. 登录。

```
let promise = chat.login({userID: 'your userID', userSig: 'your
userSig'});
promise.then(function(imResponse) {
  console.log(imResponse.data); // 登录成功
  if (imResponse.data.repeatLogin === true) {
    // 标识账号已登录，本次登录操作为重复登录。
    console.log(imResponse.data.errorInfo);
  }
}).catch(function(imError) {
  console.warn('login error:', imError); // 登录失败的相关信息
});
```

⚠ 注意：

- TRTC 和 IM 的 `sdkAppId` 和 `secretKey` 必须相同。
- 收消息的接收者的 IM 必须要登录成功，既是在线状态。
- 接收者的 TRTC 账号和 IM 账号必须是同一个 `userId`（即使用同一个 `userId` 进入 TRTC 房间和 IM 登录）。

接收服务端下行消息

通过 IM SDK 接收单聊自定义消息功能（iOS & Android / Web & 小程序），在客户端上监听回调，来接收实时字幕与 AI 状态的数据。

type	说明
10000	实时字幕、翻译下发
10001	AI 对话实时状态下发
10010	大模型消息透传

接收实时字幕

```
{
  "type": 10000, // 10000表示是下发的实时字幕
  "sender": "user_a", // 说话人的userid
  "receiver": [], // 接收者userid列表，该消息实际是在房间内广播
  "payload": {
```

```
      "text":"", // 语音识别出的文本
      "translation_text":"", // 翻译的文本
      "start_time":"00:00:01", // 这句话的开始时间
      "end_time":"00:00:02", // 这句话的结束时间
      "roundid": "xxxxx" // 唯一标识一轮对话
      "end": true // 如果为true，代表这是一句完整的话
    }
}
```

接收机器人状态

```
{
  "type": 10001, // 机器人的状态
  "sender": "user_a", // 发送者userid，这里是机器人的id
  "receiver": [], // 接受者userid列表，该消息实际是在房间内广播
  "payload": {
    "roundid": "xxx", // 唯一标识一轮对话
    "timestamp": 123
    "state": 1,       //   1 聆听中   2 思考中   3 说话中   4 被打断
  }
}
```

接收大模型消息透传

```
{
  "type": 10010, // 下行大模型消息透传
  "sender": "user_a", // 发送者userid，这里是机器人的id
  "receiver": [], // 接受者userid列表，该消息实际是在房间内广播
  "payload": {
    "id": "uuid", // 消息id，可以使用uuid，排查问题使用 可选
    "taskid":"xxxxxx", // 该ai 对话的 taskid， 必选
    "timestamp": 123 // 时间戳，排查问题使用， 可选
        "data": {
            "key": "value" //业务自定义的json格式
        }
  }
}
```

代码示例

iOS

```
// 调用 addSimpleMsgListener 设置事件监听器
V2TIMManager.sharedInstance().addSimpleMsgListener(listener: self)

/// 接收单聊自定义消息
/// @param msgID 消息 ID
/// @param info 发送者信息
/// @param data 自定义消息二进制内容
func onRecvC2CCustomMessage(_ msgID: String!, sender info:
V2TIMUserInfo!, customData data: Data!) {
    do {
        if let jsonObject = try JSONSerialization.jsonObject(with: data,
options: []) as? [String: Any] {
            print("onRecvGroupCustomMessage: \(jsonObject)")
            handleMessage(jsonObject)
        } else {
            print("The data is not a dictionary.")
        }
    } catch {
        print("Error parsing JSON: \(error)")
    }
}
```

Android

```
// 调用 addSimpleMsgListener 设置事件监听器
V2TIMManager.getInstance().addSimpleMsgListener(sdkListener);

/**
 * 接收单聊自定义消息
 * @param msgID 消息 ID
 * @param sender 发送方信息
 * @param customData 发送内容
 */
public void onRecvC2CCustomMessage(String msgID, V2TIMUserInfo sender,
byte[] customData) {
```

```
    Log.i("onRecvC2CCustomMessage", "msgID:" + msgID + ", from:" +
sender.getNickName() + ", content:" + new String(customData));
    try {
        String jsonString = new String(customData, "UTF-8");
        JSONObject jsonObject = new JSONObject(jsonString);
        System.out.println("onRecvGroupCustomMessage: " + jsonObject);
        handleMessage(jsonObject);
     } catch (UnsupportedEncodingException e) {
        System.out.println("The data is not a dictionary.");
     } catch (JSONException e) {
        System.out.println("Error parsing JSON: " + e);
     }
}
```

Web & 小程序

```
const onMessageReceived = (event) => {
  const messageList = event.data;
  messageList?.forEach((msg) => {
    if (msg.type === TencentCloudChat.TYPES.MSG_CUSTOM) {
      console.log('收到自定义消息', event);
      const { data } = msg.payload;
      try {
        const jsonData = JSON.parse(data);
        console.log(`receive custom msg from ${msg.from} data:
${data}`);
        if (jsonData.type === 10000) {
          console.log('字幕消息', jsonData);
          return;
        }
        if (jsonData.type === 10001) {
          console.log('机器人的状态', jsonData);
          return;
        }
        if (jsonData.type === 10010) {
          console.log('下行大模型消息透传', jsonData);
          return;
        }
      } catch (error) {
```

```
        console.error('receive custom msg', data, error);
      }
    }
  });
}

// 监听消息
chat.on(TencentCloudChat.EVENT.MESSAGE_RECEIVED, onMessageReceived);
```

> 说明：
> 默认通过单聊自定义消息来接收实时字幕与 AI 状态的数据。如果单聊无法满足需求，需要开通群聊自定义消息通道，请 联系我们。

端上发送上行信令

可以通过发送自定义信令，跳过 ASR 过程，直接跟 AI 进行文字沟通，或通过发送打断信令来进行打断，或直接发送透传信息给到大模型。

type	说明
20000	ai_conversation_chat：发送 AI 对话文本
20001	ai_conversation_interrupt：手动打断
20010	发送透传信息给到大模型

发送上行信令，跳过 ASR 过程，直接跟 AI 进行文字沟通

```
{
  "type": 20000,
  "sender": "user_a", // 发送者userid，服务端会 check 该 userid 是否有效
  "receiver": ["user_bot"], // 接受者 userid 列表，只需要填写机器人 userid，服务端会 check 该 userid 是否有效
  "payload": {
    "id": "uuid", // 消息 id，可以使用 uuid，排查问题使用
    "message": "xxx", // 消息内容
    "timestamp": 123, // 时间戳，排查问题使用
    "taskid": "v2_20240920_xxxxx",
  }
}
```

发送打断信令来进行打断

```
{
  "type": 20001,
  "sender": "userid", // 发送者userid，服务端会 check 该 userid 是否有效
  "receiver": ["user_bot"], // 接受者 userid 列表，只需要填写机器人 userid
  "payload": {
    "id": "uuid", // 消息 id，可以使用 uuid，排查问题使用
    "timestamp": 123 // 时间戳，排查问题使用
    "taskid": "v2_20240920_xxxxx",
  }
}
```

发送透传信息给到大模型

```
{
    "type": 20010,
    "sender": "userid",
    "receiver": [
        "robotid"
    ],
    "payload": {
        "id": "uuid",
        "taskid": "v2_20240920_xxxxx",
        "timestamp": 1234,
        "data": {
            "key": "value" //业务自定义的json格式
        }
    }
}
```

代码示例

iOS

```
@IBAction func interruptAi(_ sender: UIButton) {
    let timestamp = Int(Date().timeIntervalSince1970 * 1000)
    let payload = [
```

```
        "id": userId + "_\(roomId)" + "_\(timestamp)", // 消息id，可以使用
uuid，排查问题使用
        "timestamp": timestamp, // 时间戳，排查问题使用
        "taskid": aiTaskId,
    ] as [String : Any]
    let content = [
        "type": 20001,
        "sender": userId,
        "receiver": [botId],
        "payload": payload
    ] as [String : Any]
    let contentData = try! JSONSerialization.data(withJSONObject:
content, options: [])
    let contentString = String(data: contentData, encoding: .utf8)!
    let dataDict = [
        "service_command": "trtc_ai_service.SendCustomCmdMsg",
        "request_content": contentString
    ] as [String : Any]
    do {
        let jsonData = try JSONSerialization.data(withJSONObject:
dataDict, options: [])

V2TIMManager.sharedInstance().callExperimentalAPI("sendTRTCCustomData",
param: jsonData as NSObject) { _ in
            print("sendTRTCCustomData success")
        } fail: { code, desc in
            print("sendTRTCCustomData error, \(code), \(desc ??
"null")")
        }
    } catch {
        print("Error serializing dictionary to JSON: \(error)")
    }
}
```

Android

```
public void interruptAi() {
    long timestamp = System.currentTimeMillis();
    Map<String, Object> payload = new HashMap<>();
```

```
    payload.put("id", userId + "_" + roomId + "_" + timestamp); // 消息
id，可以使用uuid，排查问题使用
    payload.put("timestamp", timestamp); // 时间戳，排查问题使用
    payload.put("taskid", aiTaskId);

    Map<String, Object> content = new HashMap<>();
    content.put("type", 20001);
    content.put("sender", userId);
    content.put("receiver", Collections.singletonList(botId));
    content.put("payload", payload);

    String contentString = new JSONObject(content).toString();

    Map<String, Object> dataDict = new HashMap<>();
    dataDict.put("service_command", "trtc_ai_service.SendCustomCmdMsg");
    dataDict.put("request_content", contentString);

    try {
        byte[] jsonData = new
JSONObject(dataDict).toString().getBytes("UTF-8");

V2TIMManager.getInstance().callExperimentalAPI("sendTRTCCustomData",
jsonData, new V2TIMValueCallback() {
            @Override
            public void onSuccess(Object o) {
                System.out.println("sendTRTCCustomData success");
            }
            @Override
            public void onError(int code, String desc) {
                System.out.println("sendTRTCCustomData error, " + code +
", " + (desc != null ? desc : "null"));
            }
        });
    } catch (UnsupportedEncodingException e) {
        System.out.println("Error serializing dictionary to JSON: " +
e);
    }
}
```

Web & 小程序

```
// 发送打断信令
chat.callExperimentalAPI('sendTRTCCustomData', {
serviceCommand: 'trtc_ai_service.SendCustomCmdMsg',
data: {
  type: 20001,
  sender: "user_a", // 发送者userid， 服务端会check该userid是否有效
  receiver: ["user_bot"], // 接受者userid列表，只需要填写机器人userid
  payload: {
    id: "uuid", // 消息id，可以使用uuid，排查问题使用
    timestamp: 123, // 时间戳，排查问题使用
    taskid: "任务的taskid",
  }
}
});
```

注意：

`type` 、 `sender` 、 `receiver` 以及 `payload` 中的 `taskid` 、 `id` 、 `timestamp` 是必填字段。