Tencent Smart Advisor–Tencent RTC Copilot

モジュール化ソリューション

製品ドキュメント

著作権声明

商標に関する声明

サービス声明

本ドキュメントは、お客様にTencent Cloudの全部また一部の製品・サービスの概要をご紹介することを目的としておりますが、一部の製品・サービス内容は変更される可能性があります。お客様がご購入されるTencent Cloud製品・サービスの種類やサービス基準などは、お客様とTencent Cloudとの間の締結された商業契約に基づきます。別段の合意がない限り、Tencent Cloudは本ドキュメントの内容に関して、明示または黙示の一切保証もしません。

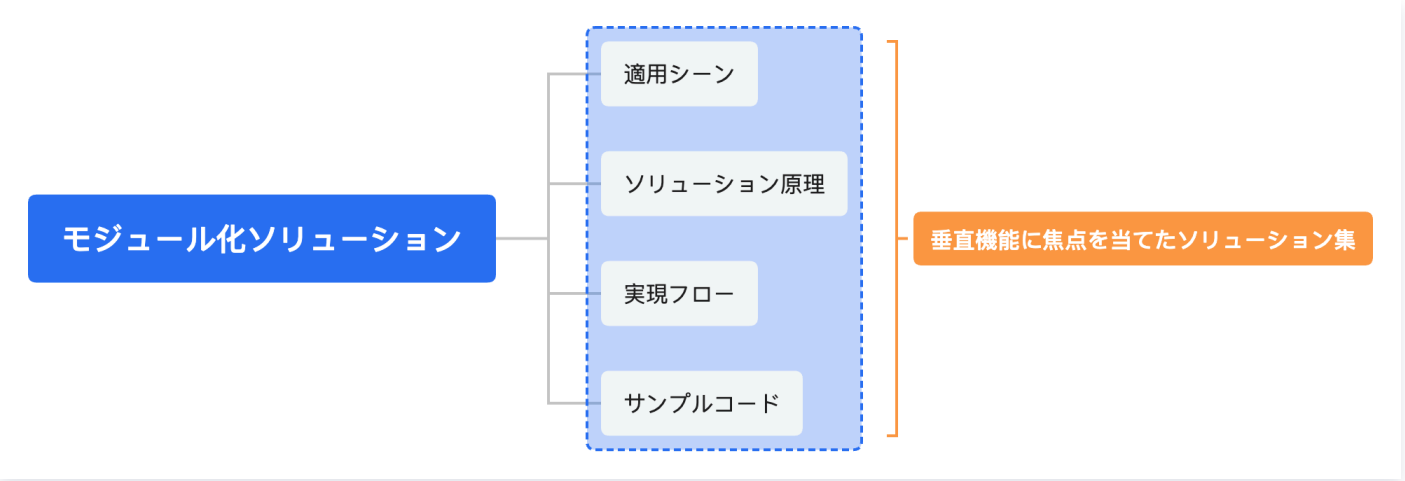
カタログ:

モジュール化ソリューション
モジュール化ソリューション概要

最終更新日：： 2025-11-18 15:15:32

概要

モジュール化ソリューションはシーン化ソリューションとは独立しており、オーディオ・ビデオ通信分野の垂直機能にさらに焦点を当て、対応するソリューション集を提供します。
現在、クロスルームPK通話ソリューション、AI対話Chatシグナリングソリューション、モバイル端末アプリケーション・キープアライブ・ソリューション、ネットワーク品質監視、ライブストリーミング上下スワイプ、ビデオピクチャーインピクチャーソリューションをカバーしており、単一機能モジュールの適用シーン、ソリューション原理、実現プロセスなどを深く理解し、業務の迅速なソリューションの選定と導入を進めるのに役立ちます。



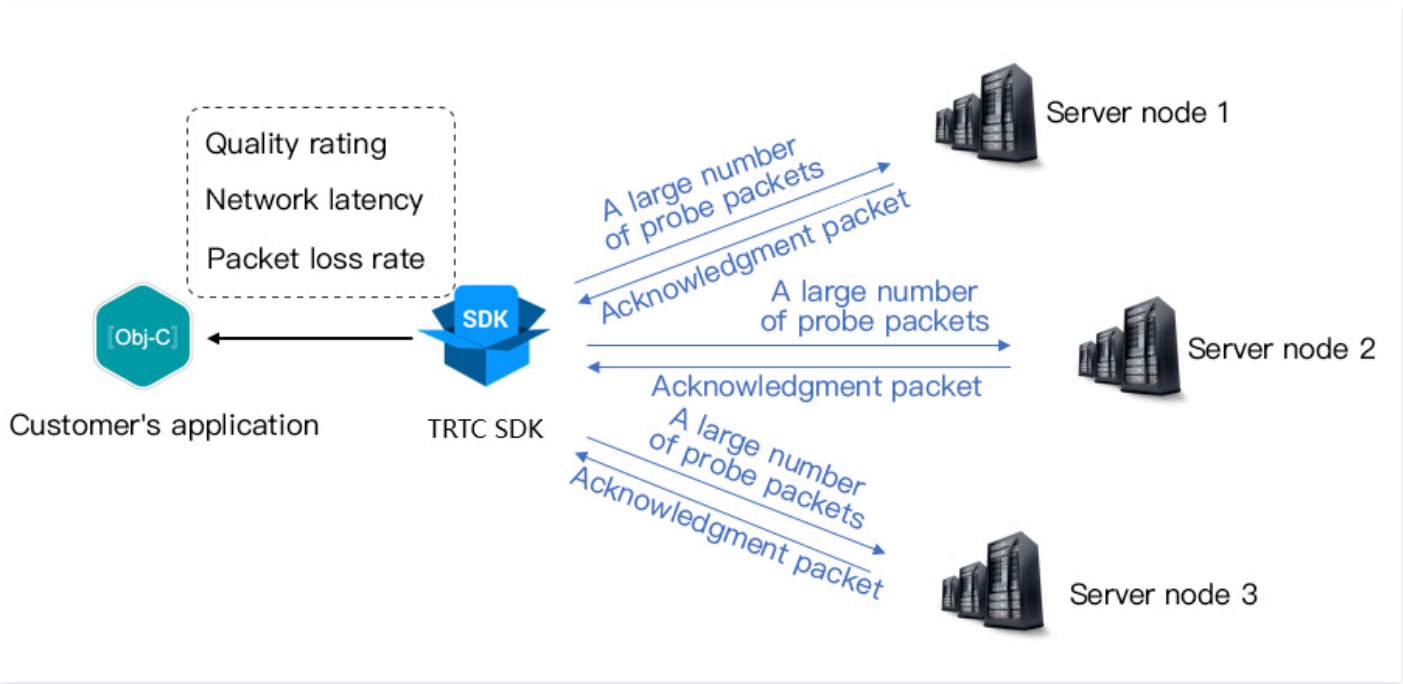
ネットワーク品質監視

最終更新日：： 2025-11-18 15:15:33

ユーザーがネットワーク品質をより良く認識できるように、通話前に速度測定を実施し、通話中はネットワーク品質と接続状況を監視し、対応するUIインターフェースのインタラクションと通知を提供することをお勧めします。これにより、ユーザーは現在のネットワーク品質を容易に評価し、ネットワーク設定を適時に調整して、最適な通話効果を達成できます。

通話前のネットワーク速度測定

一般ユーザーはネットワーク状況を認識するのが難しいため、通話前に速度測定を実施し、現在のネットワーク品質を評価して、ネットワークを適時に調整し、通話の最適な効果を達成することをお勧めします。
速度測定の原理は、SDKがサーバーノードに一連の探査パケットを送信し、返送パケットの品質を統計し、速度測定の結果をコールバックインターフェースを通じて通知することです。下図をご参照ください。



通話前の速度測定の利点：

- 速度測定の結果はSDKのサーバー選択戦略を最適化するために使用されるため、最適なサーバーを選択できるように、ユーザーが初めて通話する前に一度速度測定を実施することをお勧めします。
- テスト結果が非常に理想的でない場合、目立つUIでユーザーにより良いネットワーク環境を選択するよう促すことができます。

startSpeedTest 速度測定

Android

```
//ネットワーク速度測定を開始するサンプルコード、sdkAppIdとUserSigが必要（取得方法は基本機能を参照）
// ここではログイン後に測定を開始する例を示します
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 はコンソールで取得した実際のアプリケーションのAppIDです
    trtcCloud.startSpeedTest(params);
}

// 速度測定の結果を監視し、TRTCCloudListenerを継承して以下のメソッドを実現します
void onSpeedTestResult(TRTCCloudDef.TRTCSpeedTestResult result)
{
    // 速度測定が完了すると、速度測定の結果がコールバックされます
}Android
```

iOS

```
// ネットワーク速度測定を開始するサンプルコード、sdkAppIdとUserSigが必要です（取得方法は基本機能を参照）
// ここではログイン後に測定を開始する例を示します
- (void)onLogin:(NSString *)userId userSig:(NSString *)userSid
{
    TRTCSpeedTestParams *params;
    // sdkAppID はコンソールで取得した実際のアプリケーションのAppIDです
    params.sdkAppID = sdkAppId;
    params.userID = userId;
    params.userSig = userSig;
    // 予想されるアップリンク帯域幅（kbps、範囲：10～5000、0の場合は測定なし）
    params.expectedUpBandwidth = 5000;
    // 予想されるダウンリンク帯域幅（kbps、範囲：10～5000、0の場合は測定なし）
```

```
    params.expectedDownBandwidth = 5000;
    [trtcCloud startSpeedTest:params];
}
- (void)onSpeedTestResult:(TRTCSpeedTestResult *)result {
    // 速度測定が完了すると、速度測定の結果が返されます
}
```

Windows

```
// ネットワーク速度測定を開始するサンプルコード、sdkAppIdとUserSigが必要です（取得方法は基本機能を参照）
// ここではログイン後に測定を開始する例を示します
void onLogin(const char* userId, const char* userSig)
{
    TRTCSpeedTestParams params;
    // sdkAppID はコンソールで取得した実際のアプリケーションのAppIDです
    params.sdkAppID = sdkAppId;
    params.userId = userid;
    param.userSig = userSig;
    // 予想されるアップリンク帯域幅（kbps、範囲：10～5000、0の場合は測定なし）
    param.expectedUpBandwidth = 5000;
    // 予想されるダウンリンク帯域幅（kbps、範囲：10～5000、0の場合は測定なし）
    param.expectedDownBandwidth = 5000;
    trtcCloud->startSpeedTest(params);
}

// 速度測定の結果を監視します
void TRTCCloudCallbackImpl::onSpeedTestResult(
            const TRTCSpeedTestResult& result)
{
    // 速度測定が完了すると、速度測定の結果がコールバックされます
}
```

速度測定の結果（TRTCSpeedTestResult）には以下のフィールドが含まれます。

フィールド	意味	意味説明
success	成功したかどうか	今回のテストは成功したかどうか

errMsg	エラー情報	帯域幅テストの詳細なエラー情報
ip	サーバーIP	速度測定サーバーのIP
quality	ネットワーク品質スコア	評価アルゴリズムによって計算されたネットワーク品質は、lossが低く、rttが小さいほど、スコアが高くなります
upLostRate	アップリンク・パケット損失率	範囲は[0 – 1.0]で、例えば0.3はサーバーに10個のデータパケットを送信するたびに、3個が途中で損失する可能性があることを示します。
downLostRate	ダウンリンク・パケット損失率	範囲は[0 – 1.0]で、例えば0.2はサーバーから10個のデータパケットを受信するたびに、2個が途中で損失する可能性があることを示します。
rtt	ネットワーク遅延	SDKとサーバー間の往復にかかる時間を表し、この値は小さいほど良いです。通常の値は10ms～100msの間です。
availableUpBandwidth	アップリンク帯域幅	予測されるアップリンク帯域幅、単位はkbps、–1は無効値を示します
availableDownBandwidth	ダウンリンク帯域幅	予測されるダウンリンク帯域幅、単位はkbps、–1は無効値を示します

ツールによる速度測定

インターフェース呼び出し方式によるネットワーク速度測定を行いたくない場合、Real–Time Communication Engine （RTC Engine)はデスクトップ版のネットワーク速度測定ツールも提供しており、詳細なネットワーク品質情報を迅速に取得できます。ご利用のプラットフォームに応じて、以下のプログラムを選択してダウンロードしてください。このプログラムは、クイックテストと持続テストの2種類のテストオプションを提供しています。

- Mac
- Windows

指標	意味
WiFi Quality	Wi–Fi 信号品質

DNS RTT	Tencent Cloudの速度測定ドメイン名解決時間
MTR	MTRはネットワークテストツールで、クライアントからRTC Engineノードまでのパケット損失率と遅延を検出でき、ルートの各ホップの詳細情報も確認できます
UDP Loss	クライアントからRTC EngineノードまでのUDPパケット損失率
UDP RTT	クライアントからRTC EngineノードまでのUDP遅延
Local RTT	クライアントからローカルゲートウェイまでの遅延
Upload	アップリンク予測帯域幅
Download	ダウンリンク予測帯域幅

注意：

- ビデオ通話中は通話品質に影響を与えないよう、テストを行わないでください。
- 速度測定自体が一定のデータ通信量を消費するため、ごくわずかな追加通信料金が発生する可能性があります（基本的に無視できる程度です）。

通話中のネットワーク検出

ユーザーが通話中に発生する可能性のあるネットワークの変動をより明確に認識できるように、通話画面に適切なUIリマインダーを追加することをお勧めします。特にネットワーク状態が悪い場合、リマインダーによってユーザーが変化を感知し、ネットワーク状況を改善するための調整を迅速に行えるようにします。

onNetworkQuality でローカルネットワーク品質を監視します

onNetworkQualityのコールバックイベントは、2秒ごとに現在のネットワーク品質を報告します。そのパラメータには、localQualityとremoteQualityの2つの部分が含まれます。

- localQuality：現在のネットワーク品質を表し、Excellent、Good、Poor、Bad、VeryBad、Downの6つのレベルに分けられます。
- remoteQuality：リモートユーザーのネットワーク品質を表し、これは配列であり、配列内の各要素は1つのリモートユーザーのネットワーク品質を表します。

Quality	名称	说明
0	Unknown	検出されません

1	Excellent	現在のネットワークは非常に良好です
2	Good	現在のネットワークは比較的良好です
3	Poor	現在のネットワークは普通です
4	Bad	現在のネットワークはやや悪く、明らかなラグや通話遅延が発生する可能性があります
5	VeryBad	現在のネットワークは非常に悪く、RTC Engineはかろうじて接続を維持できますが、通信品質は保証できません
6	Down	現在のネットワークはRTC Engineの最低要件を満たしていないため、正常なオーディオ通話やビデオ通話を行うことができません

onNetworkQualityを監視し、インターフェースで適切な通知を表示するだけで十分です。

Android

```
// onNetworkQuality コールバックを監視し、現在のネットワーク状態の変化を感知します
@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

```
// onNetworkQuality コールバックを監視し、現在のネットワーク状態の変化を感知します
- (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

```
// onNetworkQuality コールバックを監視し、現在のネットワーク状態の変化を感知します
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 で完全なネットワーク品質を監視します

onStatistics 統計コールバックイベントは2秒間隔でスローされ、SDK内部のオーディオ、ビデオ、およびネットワーク関連の専門的な技術指標を通知するために使用されます。これらの情報はTRTCStatisticsにすべて記載されています。

- ビデオ統計情報：ビデオの解像度（resolution）、フレームレート（FPS）およびビットレート（bitrate）などの情報。

- オーディオ統計情報：オーディオのサンプルレート（samplerate）、チャンネル（channel）およびビットレート（bitrate）などの情報。
- ネットワーク統計情報：SDKとクラウド間の往復（SDK > Cloud > SDK）におけるネットワーク所要時間（rtt）、パケット損失率（loss）、アップリンクトラフィック（sentBytes）、ダウンリンクトラフィック（receivedBytes）などの情報。

列挙型	説明
appCpu	現在のアプリケーションのCPU使用率、単位（%）、Android 8.0以上はサポートされていません。
downLoss	クラウドからSDKまでのダウンリンクパケット損失率、単位（%）。 • この数値は小さいほど良いです。downLossが0%の場合、ダウンリンクのネットワーク品質が非常に良好であり、クラウドから受信したデータパケットが基本的に損失しないことを意味します。 • downLossが30%の場合、クラウドからSDKに送信されるオーディオ・ビデオデータパケットの30%が伝送リンクで損失することを意味します。
gatewayRtt	SDKからローカルルーターまでの往復遅延、単位はms。この数値は、SDKがネットワークパケットをローカルルーターゲートウェイに送信し、ゲートウェイからSDKにパケットを返送するまでの総所要時間、即ちネットワークパケットが「SDK > ゲートウェイ > SDK」を経由するのにかかる時間を表します。 • この数値は小さいほど良いです：gatewayRtt < 50msの場合、オーディオ・ビデオ通話の遅延が低いことを意味します；gatewayRtt > 200msの場合、オーディオ・ビデオ通話の遅延が高いことを意味します。 • ネットワークタイプがセルラーネットワークの場合、この値は無効です。
localArray	ローカルのオーディオ・ビデオ統計情報。 ローカルには3つのオーディオビデオストリーム（即ち高精細大画面、低精細小画面、および補助ストリーム画面）が存在する可能性があるため、ローカルのオーディオビデオ統計情報は配列形式となります。
receiveBytes	総受信バイト数（シグナリングデータとオーディオ・ビデオデータを含む）、単位はバイト数（Bytes）。
remoteArray	リモートのオーディオ・ビデオ統計情報。 複数のリモートユーザーが同時に存在する可能性があり、さらに各リモートユーザーは複数のオーディオ・ビデオストリーム（即ち高精細大画面、低精細小画面、および補助ストリーム画面）を同時に持つ可能性があるため、リモートのオーディオ・ビデオ統計情報は配列形式となります。
rtt	SDKからクラウドまでの往復遅延、単位はmsです。この数値は、SDKがネットワークパケットをクラウドに送信し、クラウドからSDKにネットワークパケットを返送するまでの総所要時間、即ちネットワークパケットが「SDK > クラウド > SDK」を経由する総時間を表します。

	この数値は小さいほど良いです：rtt < 50msの場合、オーディオ・ビデオ通話の遅延が低いことを意味します；rtt > 200msの場合、オーディオ・ビデオ通話の遅延が高いことを意味します。 **説明：** rttは「SDK＞クラウド＞SDK」の総所要時間を表すため、upRttとdownRttを区別する必要はありません。
sendBytes	総送信バイト数（シグナリングデータとオーディオ・ビデオデータを含む）、単位はバイト数（Bytes）。
systemCpu	現在のシステムのCPU使用率、単位（%）、Android 8.0以上はサポートされていません。
upLoss	SDKからクラウドへのアップリンクパケット損失率、単位（%）。 • この数値は小さいほど良いです。upLossが0%の場合、アップリンクのネットワーク品質が非常に良好であり、クラウドにアップロードしたデータパケットが基本的に損失しないことを意味します。 • upLossが30%の場合、SDKからクラウドに送信されるオーディオ・ビデオデータパケットの30%が伝送リンクで損失することを意味します。

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からクラウドへの往復遅延
    Log.d(TAG, "rtt:" + statistics.rtt);
    // upLoss アップリンク・パケット損失率
    Log.d(TAG, "upLoss:" + statistics.upLoss);
    // downLoss ダウンリンク・パケット損失率
    Log.d(TAG, "downLoss:" + statistics.downLoss);
    // gatewayRtt ゲートウェイからクラウドへの往復遅延
    Log.d(TAG, "gatewayRtt:" + statistics.gatewayRtt);
    // sendBytes 送信バイト数
    Log.d(TAG, "sendBytes:" + statistics.sendBytes);
```

```
    // receiveBytes 受信バイト数
    Log.d(TAG, "receiveBytes:" + statistics.receiveBytes);
    if(statistics.localArray != null) {
        for (int i = 0; i < statistics.localArray.size(); i++) {
            // ローカルビデオ幅
            Log.d(TAG, "localStatistics width:" +
statistics.localArray.get(i).width);
            // ローカルビデオ高さ
            Log.d(TAG, "localStatistics height:" +
statistics.localArray.get(i).height);
            // ローカルビデオフレームレート
            Log.d(TAG, "localStatistics frameRate:" +
statistics.localArray.get(i).frameRate);
            // ローカルビデオビットレート
            Log.d(TAG, "localStatistics videoBitrate:" +
statistics.localArray.get(i).videoBitrate);
            // ローカルオーディオビットレート
            Log.d(TAG, "localStatistics audioBitrate:" +
statistics.localArray.get(i).audioBitrate);
            // ローカルオーディオデバイスの収集状態（オーディオ周辺機器の健全性を検
出するために使用）0：収集デバイスの状態が正常；1：長時間の無音を検出；2：音割れを検
出；3：音声の異常な中断を検出。
            Log.d(TAG, "localStatistics audioCaptureState:" +
statistics.localArray.get(i).audioCaptureState);
            // ローカルオーディオサンプリングレート
            Log.d(TAG, "localStatistics audioSampleRate:" +
statistics.localArray.get(i).audioSampleRate);
            // ローカルストリームタイプ
            Log.d(TAG, "localStatistics streamType:" +
statistics.localArray.get(i).streamType);
        }
    }
    if(statistics.remoteArray != null) {
        for (int i = 0; i < statistics.remoteArray.size(); i++) {
            // リモートユーザーuserid
            Log.d(TAG, "remoteStatistics userId:" +
statistics.remoteArray.get(i).userId);
            // リモートユーザーストリームタイプ
            Log.d(TAG, "remoteStatistics streamType:" +
statistics.remoteArray.get(i).streamType);
```

```
            // リモートビデオ幅
            Log.d(TAG, "remoteStatistics width:" +
statistics.remoteArray.get(i).width);
            // リモートビデオ高さ
            Log.d(TAG, "remoteStatistics height:" +
statistics.remoteArray.get(i).height);
            // リモートビデオフレームレート
            Log.d(TAG, "remoteStatistics frameRate:" +
statistics.remoteArray.get(i).frameRate);
            // リモートビデオビットレート
            Log.d(TAG, "remoteStatistics videoBitrate:" +
statistics.remoteArray.get(i).videoBitrate);
            // リモートビデオブロック率 単位 (%) ビデオ再生ブロック率
(videoBlockRate) = ビデオ再生の累計ブロック時間 (videoTotalBlockTime) / ビデオ
再生の総時間。
            Log.d(TAG, "remoteStatistics videoBlockRate:" +
statistics.remoteArray.get(i).videoBlockRate);
            // ビデオ再生の累計ブロック時間、単位 ms
            Log.d(TAG, "remoteStatistics videoTotalBlockTime:" +
statistics.remoteArray.get(i).videoTotalBlockTime);
            // このビデオストリームの総パケット損失率
            // videoPacketLossは、このビデオストリームが 配信者>クラウド>視聴者
という完全な伝送リンクを経た後、最終的に視聴者側で統計されたパケット損失率を表します。
            // videoPacketLossは小さいほど良く、パケット損失率が0ということは、そ
のビデオストリームのすべてのデータが視聴者側に完全に到達したことを意味します。
            // downLoss == 0 だが videoPacketLoss != 0 の場合、そのビデオスト
リームは クラウド>視聴者 の伝送リンクではパケット損失が発生しなかったが、配信者>クラ
ウド の伝送リンクで回復不能なパケット損失が発生したことを示します。
            Log.d(TAG, "remoteStatistics videoPacketLoss:" +
statistics.remoteArray.get(i).videoPacketLoss);
            // リモートオーディオビットレート
            Log.d(TAG, "remoteStatistics audioBitrate:" +
statistics.remoteArray.get(i).audioBitrate);
            // リモートオーディオ再生のブロック率
            Log.d(TAG, "remoteStatistics audioBlockRate:" +
statistics.remoteArray.get(i).audioBlockRate);
            // リモートオーディオのサンプリングレート
            Log.d(TAG, "remoteStatistics audioSampleRate:" +
statistics.remoteArray.get(i).audioSampleRate);
            // リモートオーディオのパケット損失率
```

```
            Log.d(TAG, "remoteStatistics audioPacketLoss:" +
statistics.remoteArray.get(i).audioPacketLoss);
            // オーディオ再生の累計ブロック時間
            Log.d(TAG, "remoteStatistics audioTotalBlockTime:" +
statistics.remoteArray.get(i).audioTotalBlockTime);
            // 再生遅延　ネットワークジッターやネットワークパケットの順序乱れによる音声と映像のラグを防ぐため、RTC Engineは再生側で再生バッファを管理し、受信したネットワークパケットを整列します。このバッファのサイズは現在のネットワーク品質に応じて自動調整され、このバッファの大きさをミリ秒単位の時間長に換算したものがjitterBufferDelayです。
            Log.d(TAG, "remoteStatistics jitterBufferDelay:" +
statistics.remoteArray.get(i).jitterBufferDelay);
            // エンドツーエンド遅延、単位 ms
            //point2PointDelayは「配信者=>クラウド=>視聴者」の遅延を表し、より正確には「収集=>エンコード=>ネットワーク伝送=>受信=>バッファ=>デコード=>再生」という全リンクの遅延を意味します。
            //point2PointDelay は、ローカルとリモートのSDKがともにバージョン8.5以上の場合にのみ有効です。リモートユーザーが8.5以前のバージョンの場合、この数値は常に0になり、意味を持ちません。
            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からクラウドへの往復遅延
    NSLog(@"rtt: %d", statistics.rtt);
    // upLoss アップリンク・パケット損失率
    NSLog(@"upLoss: %u", statistics.upLoss);
    // downLoss ダウンリンク・パケット損失率
    NSLog(@"downLoss: %u", statistics.downLoss);
```

```
    // gatewayRtt ゲートウェイからクラウドへの往復遅延
    NSLog(@"gatewayRtt: %d", statistics.gatewayRtt);
    // sendBytes 送信バイト数
    NSLog(@"sendBytes: %llu", (unsigned long long)statistics.sentBytes);
    // receiveBytes 受信バイト数
    NSLog(@"receiveBytes: %llu", (unsigned long
long)statistics.receivedBytes);

    if (statistics.localStatistics != nil) {
        for (int i = 0; i < statistics.localStatistics.count; i++) {
            // ローカルビデオ幅
            NSLog(@"localStatistics width: %d",
statistics.localStatistics[i].width);
            // ローカルビデオ高さ
            NSLog(@"localStatistics height: %d",
statistics.localStatistics[i].height);
            // ローカルビデオフレームレート
            NSLog(@"localStatistics frameRate: %u",
statistics.localStatistics[i].frameRate);
            // ローカルビデオビットレート
            NSLog(@"localStatistics videoBitrate: %d",
statistics.localStatistics[i].videoBitrate);
            // ローカルオーディオビットレート
            NSLog(@"localStatistics audioBitrate: %d",
statistics.localStatistics[i].audioBitrate);
            // ローカルオーディオデバイスの収集状態
            NSLog(@"localStatistics audioCaptureState: %d",
statistics.localStatistics[i].audioCaptureState);
            // ローカルオーディオサンプリングレート
            NSLog(@"localStatistics audioSampleRate: %d",
statistics.localStatistics[i].audioSampleRate);
            // ローカルストリームタイプ
            NSLog(@"localStatistics streamType: %ld",
(long)statistics.localStatistics[i].streamType);
        }
    }

    if (statistics.remoteStatistics != nil) {
        for (int i = 0; i < statistics.remoteStatistics.count; i++) {
            // リモートユーザーuserid
```

```
            NSLog(@"remoteStatistics userId: %@",
statistics.remoteStatistics[i].userId);
            // リモートユーザーストリームタイプ
            NSLog(@"remoteStatistics streamType: %ld",
(long)statistics.remoteStatistics[i].streamType);
            // リモートビデオ幅
            NSLog(@"remoteStatistics width: %d",
statistics.remoteStatistics[i].width);
            // リモートビデオ高さ
            NSLog(@"remoteStatistics height: %d",
statistics.remoteStatistics[i].height);
            // リモートビデオフレームレート
            NSLog(@"remoteStatistics frameRate: %u",
statistics.remoteStatistics[i].frameRate);
            // リモートビデオビットレート
            NSLog(@"remoteStatistics videoBitrate: %d",
statistics.remoteStatistics[i].videoBitrate);
            // リモートビデオのブロック率
            NSLog(@"remoteStatistics videoBlockRate: %u",
statistics.remoteStatistics[i].videoBlockRate);
            // ビデオ再生の累計ブロック時間
            NSLog(@"remoteStatistics videoTotalBlockTime: %d",
statistics.remoteStatistics[i].videoTotalBlockTime);
            // このビデオストリームの総パケット損失率
            NSLog(@"remoteStatistics videoPacketLoss: %u",
statistics.remoteStatistics[i].videoPacketLoss);
            // リモートオーディオビットレート
            NSLog(@"remoteStatistics audioBitrate: %d",
statistics.remoteStatistics[i].audioBitrate);
            // リモートオーディオ再生のブロック率
            NSLog(@"remoteStatistics audioBlockRate: %u",
statistics.remoteStatistics[i].audioBlockRate);
            // リモートオーディオのサンプリングレート
            NSLog(@"remoteStatistics audioSampleRate: %d",
statistics.remoteStatistics[i].audioSampleRate);
            // リモートオーディオのパケット損失率
            NSLog(@"remoteStatistics audioPacketLoss: %u",
statistics.remoteStatistics[i].audioPacketLoss);
            // オーディオ再生の累計ブロック時間
```

```
            NSLog(@"remoteStatistics audioTotalBlockTime: %d",
statistics.remoteStatistics[i].audioTotalBlockTime);
            // 再生遅延
            NSLog(@"remoteStatistics jitterBufferDelay: %d",
statistics.remoteStatistics[i].jitterBufferDelay);
            // エンドツーエンド遅延
            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からクラウドへの往復遅延
    printf("rtt: %d\n", statistics.rtt);
    // upLoss アップリンク・パケット損失率
    printf("upLoss: %f\n", statistics.upLoss);
    // downLoss ダウンリンク・パケット損失率
    printf("downLoss: %f\n", statistics.downLoss);
    // gatewayRtt ゲートウェイからクラウドへの往復遅延
    printf("gatewayRtt: %d\n", statistics.gatewayRtt);
    // sentBytes 送信バイト数
    printf("sentBytes: %lld\n", statistics.sentBytes);
    // receiveBytes 受信バイト数
    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++)
```

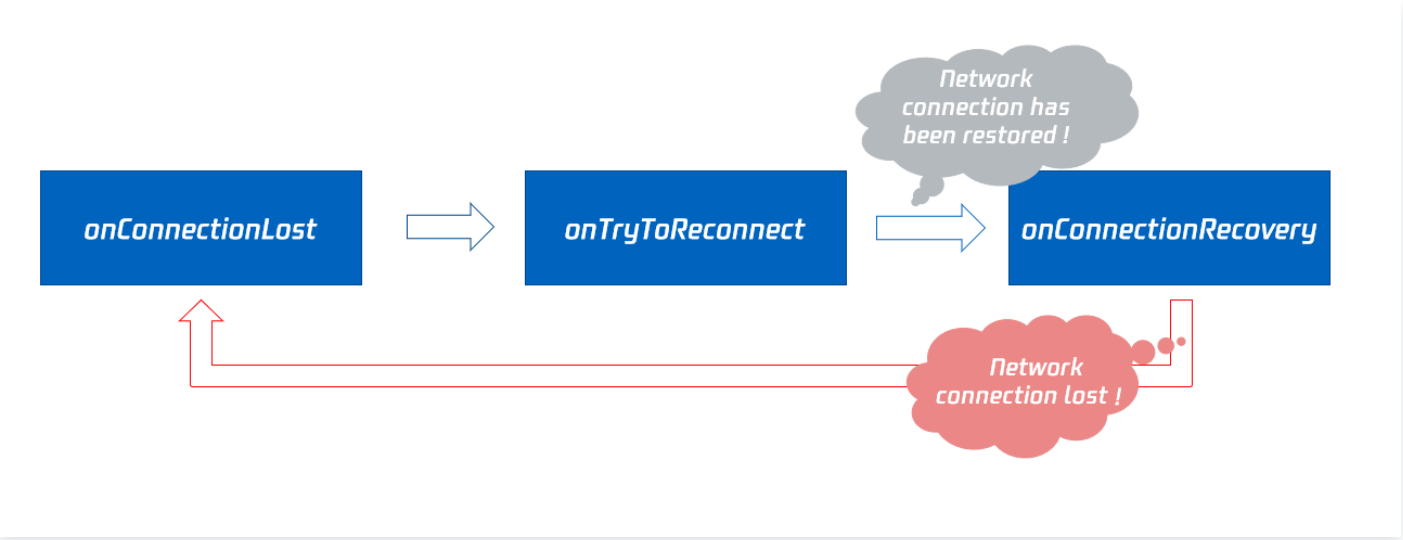
```
        {
            // ローカルビデオ幅
            printf("localStatistics width: %d\n",
statistics.localStatisticsArray[i].width);
            // ローカルビデオ高さ
            printf("localStatistics height: %d\n",
statistics.localStatisticsArray[i].height);
            // ローカルビデオフレームレート
            printf("localStatistics frameRate: %f\n",
statistics.localStatisticsArray[i].frameRate);
            // ローカルビデオビットレート
            printf("localStatistics videoBitrate: %d\n",
statistics.localStatisticsArray[i].videoBitrate);
            // ローカルオーディオビットレート
            printf("localStatistics audioBitrate: %d\n",
statistics.localStatisticsArray[i].audioBitrate);
            // ローカルオーディオデバイスの収集状態
            printf("localStatistics audioCaptureState: %d\n",
statistics.localStatisticsArray[i].audioCaptureState);
            // ローカルオーディオサンプリングレート
            printf("localStatistics audioSampleRate: %d\n",
statistics.localStatisticsArray[i].audioSampleRate);
            // ローカルストリームタイプ
            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++)
        {
            // リモートユーザーuserid
            printf("remoteStatistics userId: %s\n",
statistics.remoteStatisticsArray[i].userId);
            // リモートユーザーストリームタイプ
            printf("remoteStatistics streamType: %d\n",
statistics.remoteStatisticsArray[i].streamType);
```

```
            // リモートビデオ幅
            printf("remoteStatistics width: %d\n",
statistics.remoteStatisticsArray[i].width);
            // リモートビデオ高さ
            printf("remoteStatistics height: %d\n",
statistics.remoteStatisticsArray[i].height);
            // リモートビデオフレームレート
            printf("remoteStatistics frameRate: %f\n",
statistics.remoteStatisticsArray[i].frameRate);
            // リモートビデオビットレート
            printf("remoteStatistics videoBitrate: %d\n",
statistics.remoteStatisticsArray[i].videoBitrate);
            // リモートビデオのブロック率
            printf("remoteStatistics videoBlockRate: %f\n",
statistics.remoteStatisticsArray[i].videoBlockRate);
            // ビデオ再生の累計ブロック時間
            printf("remoteStatistics videoTotalBlockTime: %d\n",
statistics.remoteStatisticsArray[i].videoTotalBlockTime);
            // このビデオストリームの総パケット損失率
            printf("remoteStatistics videoPacketLoss: %f\n",
statistics.remoteStatisticsArray[i].videoPacketLoss);
            // リモートオーディオビットレート
            printf("remoteStatistics audioBitrate: %d\n",
statistics.remoteStatisticsArray[i].audioBitrate);
            // リモートオーディオ再生のブロック率
            printf("remoteStatistics audioBlockRate: %f\n",
statistics.remoteStatisticsArray[i].audioBlockRate);
            // リモートオーディオのサンプリングレート
            printf("remoteStatistics audioSampleRate: %d\n",
statistics.remoteStatisticsArray[i].audioSampleRate);
            // リモートオーディオのパケット損失率
            printf("remoteStatistics audioPacketLoss: %f\n",
statistics.remoteStatisticsArray[i].audioPacketLoss);
            // オーディオ再生の累計ブロック時間
            printf("remoteStatistics audioTotalBlockTime: %d\n",
statistics.remoteStatisticsArray[i].audioTotalBlockTime);
            // 再生遅延
            printf("remoteStatistics jitterBufferDelay: %d\n",
statistics.remoteStatisticsArray[i].jitterBufferDelay);
            // エンドツーエンド遅延
```

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

通話中の接続監視

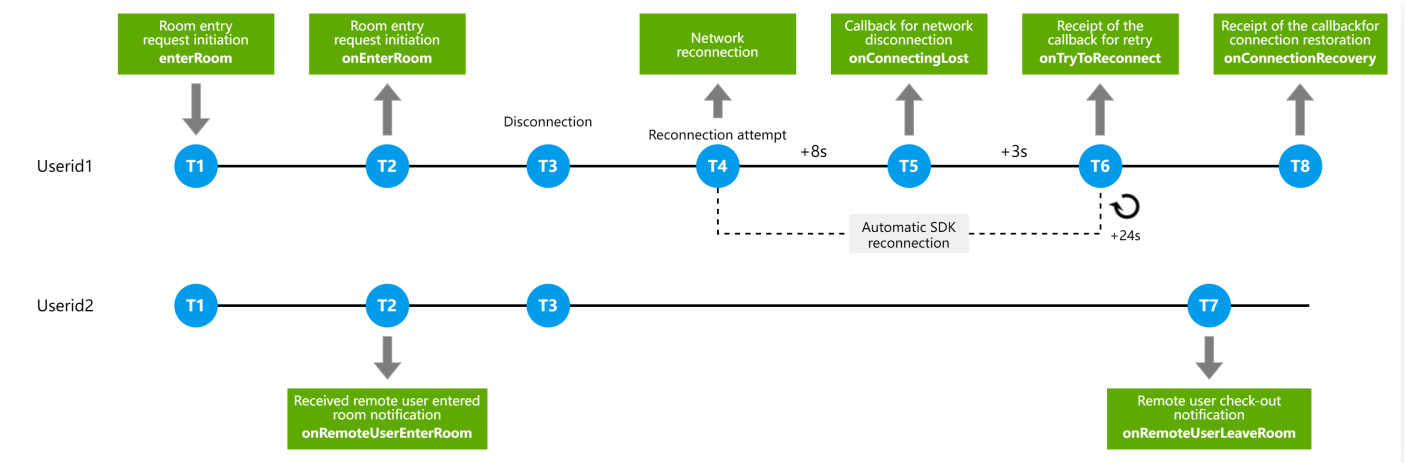
ユーザーは通話中にネットワークの変化を感知するだけでなく、現在のSDKとバックエンドの接続状態を把握する必要があります。これにより、ユーザーは現在のネットワークが調整済みかどうか、および正常に通話を行えるかどうかをより正確に判断できます。



コールバックインターフェース	インターフェース説明
onConnectionLost	SDKとクラウドの接続が切断されました
onTryToReconnect	SDKはクラウドへの再接続を試みています
onConnectionRecovery	SDKとクラウドの接続が復旧しました

切断再接続ロジック

SDKはユーザーの接続が切断された場合に自動的に再接続をサポートします（30分間再接続に成功しない場合、自動的に退室し、-3301エラーコードが返されます）。接続プロセス中の具体的な接続状態と処理ロジックは以下の通りです。下図は、ユーザーUserid1がチャンネルに参加し、接続が中断され、再度ルームに再参加する過程で受信したリスナーコールバックイベントを示しています。



具体的な説明:

- T1：ユーザー側が `enterRoom` インターフェースを呼び出して入室リクエストを開始します。
- T2：ユーザーUserid1が `onEnterRoom` コールバックを受信し、Userid2はUserid1の遅延を感知し、約300ms後にUserid2が `onRemoteUserEnterRoom` コールバックを受信します。
- T3：Userid1クライアントがネットワーク問題で切断され、SDKはルームへの再参加を試みます。
- T4：Userid1が**8秒間連続で**サーバーに接続できない場合、Userid1は `onConnectionLost` 切断コールバックを受信します。
- T5：Userid1が続いて**3秒間隔で**サーバーに接続できない場合、Userid1は `onTryToReconnect` 再試行コールバックを受信します。
- T6：Userid1が続いて**24秒ごとに**、 `onTryToReconnect` 再試行コールバックを受信します。
- T7：Userid2はUserid1の切断通知を受信してから90秒後に、SDKがリモートユーザーUserid1の切断を判断し、Userid2は `onRemoteUserLeaveRoom` コールバックを受信します。
- T8：Userid1が切断中にいつでも再接続に成功した場合、Userid1は `onConnectionRecovery` 回復コールバックを受信します。

Android

```
@Override
public void onConnectionLost() {
    // 接続が切断されました
    Log.d(TAG, "onConnectionLost");
}

@Override
public void onTryToReconnect() {
    // 再接続を試みます
    Log.d(TAG, "onTryToReconnect");
```

```
}

@Override
public void onConnectionRecovery() {
    // 再接続に成功しました
    Log.d(TAG, "onConnectionRecovery");
}
```

iOS&Mac

```
- (void)onConnectionLost {
    // 接続が切断されました
    NSLog(@"onConnectionLost");
}

- (void)onTryToReconnect {
    // 再接続を試みます
    NSLog(@"onTryToReconnect");
}

- (void)onConnectionRecovery {
    // 再接続に成功しました
    NSLog(@"onConnectionRecovery");
}
```

Windows

```
void onConnectionLost() {
    // 接続が切断されました
    printf("onConnectionLost\n");
}

void onTryToReconnect() {
    // 再接続を試みます
    printf("onTryToReconnect\n");
}
```

```
void onConnectionRecovery() {
    // 再接続に成功しました
    printf("onConnectionRecovery\n");
}
```

モバイルアプリケーション・キープアライブ・ソリューション

最終更新日：： 2025-11-18 15:15:33

オーディオ・ビデオの収集および再生に関わるモバイルアプリケーションの場合、通常は追加のキープアライブ処理が必要です。そうしないと、アプリケーションがバックエンドで実行されている際に一定の機能制限を受けるか、バックエンドでしばらく実行された後にシステムによって強制終了される可能性があります。以下では、Androidアプリケーション・キープアライブ・ソリューションとiOSアプリケーション・キープアライブ・ソリューションをそれぞれ紹介します。

Androidアプリケーション・キープアライブ・ソリューション

現在、Android端末でよく使われるキープアライブ方法は、フロントエンドサービスを起動することです。フロントエンドサービスは特殊なサービスで、実行時に持続的な通知を表示し、ユーザーにサービスが実行中であることを知らせます。フロントエンドサービスの通知は持続的に表示されるため、システムはこれを高優先度のタスクと見なし、アプリケーションはバックエンドで継続的に実行され、システムによって終了されにくくなります。以下に、フロントエンドサービスの実現方法と手順を紹介します。

ステップ1：権限を宣言

AndroidManifest.xmlファイルに以下の権限宣言を追加します。

```
<!-- アプリケーションがフロントエンドサービスを使用することを許可 -->
<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />

<!-- アプリケーションがフロントエンドサービスでカメラを使用する必要がある場合 -->
<uses-permission
android:name="android.permission.FOREGROUND_SERVICE_CAMERA" />

<!-- アプリケーションがフロントエンドサービスでマイクを使用する必要がある場合 -->
<uses-permission
android:name="android.permission.FOREGROUND_SERVICE_MICROPHONE" />

<!-- フロントエンドサービスが通知を送信することを許可 -->
<uses-permission android:name="android.permission.POST_NOTIFICATIONS"/>
```

> ⚠ 注意：

- プロジェクトのtargetSdkVersionが34以上に設定されており、かつフロントエンドサービスでカメラとマイクを使用する必要がある場合、`FOREGROUND_SERVICE_CAMERA` と `FOREGROUND_SERVICE_MICROPHONE` の権限宣言が必須です。
- Android 13以上では、フロントエンドサービスを通知バーに表示させたい場合、`POST_NOTIFICATIONS` の権限宣言が必要です。

ステップ2：サービスクラスを作成

Serviceを継承したクラスを作成し、その中でフロントエンドサービスのロジックを実現します。

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

    @Override
    public void onCreate() {
        super.onCreate();
        // 通知チャネルを作成
        Notification notification = createNotification();
        // サービス起動ロジックを処理
        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;
        //8.0の判断を行います
        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);
            }
        }
        // 通知バーをクリックしてアプリケーションに戻るように設定、オプション
        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("これはテストタイトルです")
                    .setContentIntent(pendingIntent)
                    .setContentText("これはテスト内容です")
                    .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("これはテストタイトルです")
                    .setContentText("これはテスト内容です")
                    .setContentIntent(pendingIntent)
                    .setPriority(NotificationCompat.PRIORITY_DEFAULT);
            return builder.build();
        }
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        // フロントエンドサービスを停止
        stopForeground(true);
    }
}
```

ステップ3：サービスを宣言

AndroidManifest.xmlファイルでサービスを宣言します。

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

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

> ⚠ 注意：

`android:foregroundServiceType` 属性を使用して、フロントエンドサービスが使用する必要のあるサービスタイプを指定し、バックエンドで正常なサービス機能を維持できるようにします。

- `mediaPlayback` サービスはメディア再生に使用。
- `mediaProjection` サービスはメディア投影に使用。
- `microphone` サービスはマイク有効化に使用。
- `camera` サービスはカメラ有効化に使用。

ステップ4：フロントエンドサービスを起動

フロントエンドサービスを起動する必要がある場合、必要に応じてフロントエンドサービスを起動します。

```
NotificationManagerCompat notificationManager =
NotificationManagerCompat.from(this);
boolean areNotificationsEnabled =
notificationManager.areNotificationsEnabled();
if (!areNotificationsEnabled) {
    // ユーザーに通知権限を有効化するよう促します
    Toast.makeText(this, "サービスが正常に動作するように通知権限を有効化してくださ
い", Toast.LENGTH_LONG).show();
    // ユーザーを設定ページに案内します
    Intent intent = new
Intent(Settings.ACTION_APP_NOTIFICATION_SETTINGS)
            .putExtra(Settings.EXTRA_APP_PACKAGE, getPackageName());
    startActivity(intent);
} else {
    // フロントエンドサービスを起動します
    Intent serviceIntent = new Intent(this, MyForegroundService.class);
    ContextCompat.startForegroundService(this, serviceIntent);
}
```

注意：
フロントエンドサービスが正常に動作するようにするため、フロントエンドサービスを起動する前に、通知権限が無効になっていないか確認することをお勧めします。無効になっている場合は、ユーザーに通知権限を有効化するよう促すことができます。

ステップ5：フロントエンドサービスを停止

外部コンポーネント（例：ActivityやBroadcastReceiver）からフロントエンドサービスを停止できます。

```
// サービスのIntentを作成
Intent serviceIntent = new Intent(this, MyForegroundService.class);

// サービスを停止
stopService(serviceIntent);
```

ほとんどのモバイルデバイスでは、フロントエンドサービスを起動したアプリケーションに対して、ユーザーが最近使用したアプリケーションのリストでスワイプして強制終了すると、フロントエンドサービスも同時に終了し、アプリケーションは完全に停止します。しかし、**一部の海外ブランドのモバイルデバイス（例：Google Pixel シリーズやSAMSUNG A シリーズ）では、スワイプして強制終了した後もアプリケーションは完全に停止せず、フロントエンドサービスが依然としてアクティブな状態になり、ユーザーがメディア再生を聞き続けることができます。**
このようなデバイスでは、以下の2つのソリューションを実現することで、アプリケーションが強制終了された後もメディアオーディオが再生される現象を回避できます。

ソリューション1：サービス宣言でstopWithTask属性を定義

`android:stopWithTask="true"` 属性値を追加すると、タスクが削除された時にサービスは直ちに停止します。

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

ソリューション2：Service 層で onTaskRemoved コールバックを監視

onTaskRemovedを監視し、タスクが削除されるとこのメソッドがコールバックされ、ここでクリーンアップ操作やデータ保存のロジックを実行できます。

```
@Override
public void onTaskRemoved(Intent rootIntent) {
    super.onTaskRemoved(rootIntent);
    // 例えば、ここでReal-Time Communication Engine (RTC Engine)の退室を実行
し、オーディオの収集と再生を続けないようにします
    TRTCCloud mTRTCCloud = TRTCCloud.sharedInstance(this);
    mTRTCCloud.exitRoom();
```

```
}
```

> ⚠ 注意：
> 上記の2つの方法のいずれかを選択してください。`android:stopWithTask="true"` を設定すると、`onTaskRemoved` メソッドはコールバックされなくなります。

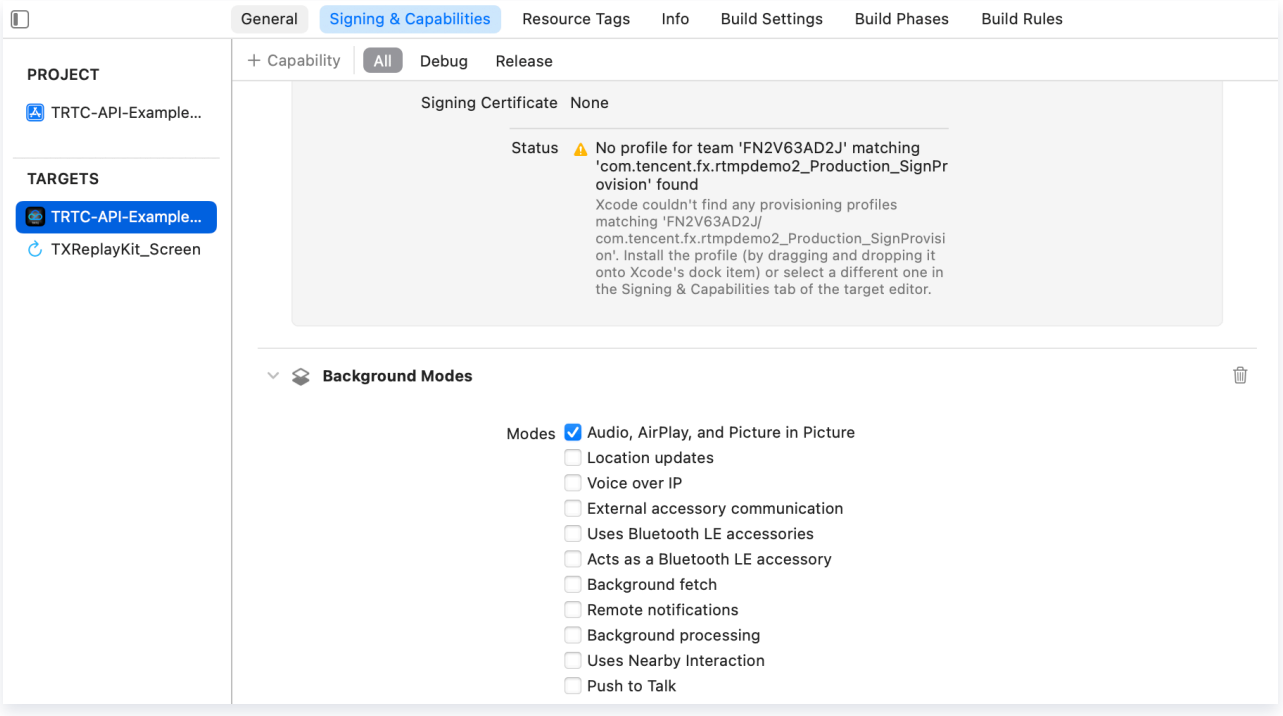
iOSアプリケーション・キープアライブ・ソリューション

Appleは、ユーザーのプライバシーとデバイスのバッテリー寿命を保護するため、アプリケーションのバックエンドでの動作に厳しい制限を設けています。通常、特定のバックエンドモード（Background Modes）を有効化することで、ある程度アプリケーション・キープアライブを実現し、アプリケーションのバックエンドでのオーディオやビデオの再生を許可することができます。

バックエンドモードを有効化

Xcodeでバックエンドモード（Background Modes）を有効化する手順は以下の通り。

1. Xcodeプロジェクトを開きます。
2. プロジェクトファイル（通常はプロジェクトナビゲータの上部にあります）を選択します。
3. プロジェクトファイルでターゲット（**Targets**）を選択します。
4. ターゲット設定で、**Signing&Capabilities**タブを選択します。
5. **Background Modes**オプションを見つけて、**Audio, AirPlay, and Picture in Picture**モードにチェックを入れます。



よくある質問

1. **音声通話またはライブストリーミングシーンで、配信者がアプリケーションをバックエンドに移動したり画面をロックしたりした場合、イヤホンの抜き差しによりオーディオの収集と再生が無音になります**

   SDKのデフォルトのオーディオポリシーでは、システム音量タイプは自動切り替えモード、即ち「通話中は通話音量、非通話中はメディア音量」になっています。また、音量タイプはオーディオルートの変化に伴って変化します。例えば、イヤホンを挿入すると、音量タイプは通話音量からメディア音量に切り替わります。したがって、自動切り替えモードでは、配信者がイヤホンを抜き差しすると、システム音量タイプが切り替わり、この時、システムはオーディオドライバーを再起動する必要があります。しかし、iOSシステムはバックエンドや画面ロック状態でオーディオドライバーを再起動する際、失敗する可能性があるため、オーディオの収集と再生が無音になる原因となります。
   このような問題に対しては、システムの音量タイプを固定することで回避できます。例えば、全過程の通話音量または全過程のメディア音量を指定します。

   ```
   // 全過程の通話音量を指定
   [self.trtcCloud setSystemVolumeType:TRTCSystemVolumeTypeVOIP];

   // 全過程のメディア音量を指定
   [self.trtcCloud setSystemVolumeType:TRTCSystemVolumeTypeMedia];
   ```

2. **ビデオ通話またはライブストリーミングシーンで、配信者がアプリケーションをバックエンドに移動したり画面をロックしたりした場合、リモート視聴者がストリームを受信すると、画面はブラックアウトするが、オーディオは正常です**

   Appleシステムは、アプリケーションがバックエンドでビデオを収集することを厳しく禁止しています。バックエンドモードを有効化した場合でも、アプリケーションがバックエンドに移行するとカメラは自動的に停止します。これはユーザーのプライバシーを保護し、アプリケーションがユーザーの同意なしにビデオを録画するのを防ぐためです。したがって、このようなシーンでのビデオ収集の問題は一時的に回避できず、オーディオの正常な収集と再生のみを実現できます。
3. **視聴者がルームに入室した際にルーム内で配信者がおらず、アプリケーションをバックエンドに移動したり画面をロックしたりすると、後続でルーム内で配信者が現れても正常に受信できません**

   iOSアプリケーションがバックエンドに切り替わる前に、AudioUnitの収集または再生を開始していない場合、すぐに中断され、フロントエンドに切り替わるまで人為的にウェイクアップすることはできません。このような問題を解決するには、アプリケーションがバックエンドに切り替わる前に、AudioUnitを常に実行状態に保つ（無音データを再生）だけで十分です。具体的な実現方法は、以下のサンプルコードをご参照ください。

   ```
   // ルームに入室後にカスタムオーディオトラックを有効化します
   [self.trtcCloud enableMixExternalAudioFrame:NO playout:YES];

   // ルームから退室する前にカスタムオーディオトラックを無効化します
   [self.trtcCloud enableMixExternalAudioFrame:NO playout:NO];
   ```

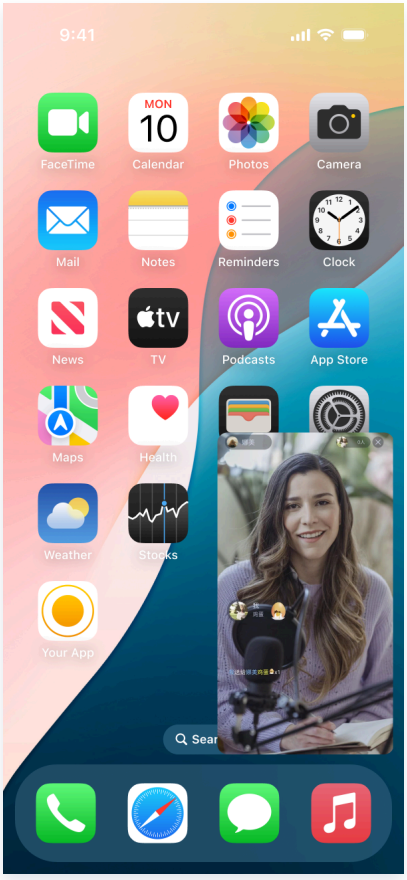
ビデオピクチャーインピクチャー アプリケーション・キープアライブ・ソリューション

ビデオピクチャーインピクチャー機能により、アプリケーションのビデオを常にフロントエンドで再生しつつ、アプリケーション・キープアライブを実現できます。具体的な実現方法については、ビデオピクチャーインピクチャーソリューション をご参照ください。

ビデオピクチャーインピクチャーソリューション

最終更新日：： 2025-11-18 15:15:33

インタラクションライブストリーミングなどのビデオシーンでは、モバイルデバイスの視聴者が長時間配信者の画面を視聴している際に、他のAPPを一時的に操作する必要がある場面があります。このような場合、配信者の画面を中断せずに他のAPPを操作できれば、視聴者により良いユーザー体験を提供できます。ビデオピクチャーインピクチャーはこのシーンに対するソリューションであり、実現効果は下図の通りです。本文では、iOS、Android、およびFlutter端末におけるピクチャーインピクチャーの実現についてそれぞれ紹介します。

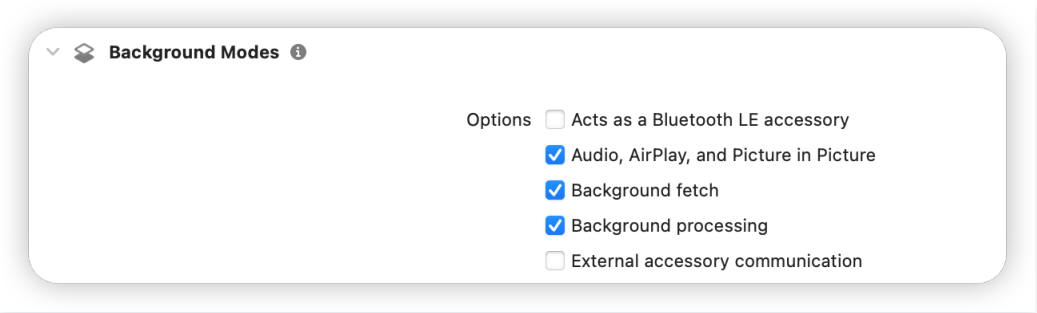


ピクチャーインピクチャーは、iOSとAndroidが提供するシステム機能に依存して実現されており、配信者側（カメラの収集とデータのアップロードが必要）と視聴者側（データのダウンロードのみ必要）に分けられます。iOSシステムは権限管理が比較的厳しいため、iOS端末では視聴者側のピクチャーインピクチャーのみを提供し、Android端末では配信者側と視聴者側の両方のピクチャーインピクチャーを提供しています。ビデオ再生に関しては、一般的にReal-Time Communication Engine （RTC Engine)を使用した再生とライブストリーミング再生の2つの方法があり、ピクチャーインピクチャーソリューションでもこの2つの状況について説明します。

iOS 端末の視聴者ピクチャーインピクチャー実現

対応する権限を有効化

iOSプロジェクトのSigning & Capabilitiesで以下の権限を有効化する必要があります。



SDKを呼び出して実現

iOS端末では、SDKはピクチャーインピクチャー機能を実現するためのインターフェースを提供しており、APIを呼び出すことで簡単にピクチャーインピクチャーを有効化できます（関連APIは以下のサンプルコードを参照）。ただし、SDKは単一の配信者を視聴するピクチャーインピクチャー機能のみを提供しており、複数の配信者PKピクチャーインピクチャーを視聴する機能をサポートする必要がある場合は、システムのAPIを呼び出して実現する必要があります。詳細はシステムAPIを呼び出して実現をご参照ください。

RTC Engine 再生

> **説明:**
> RTC EngineのSDKは12.1以降のバージョンが必要です。

視聴者側で以下のインターフェースを呼び出して有効化します。

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];
```

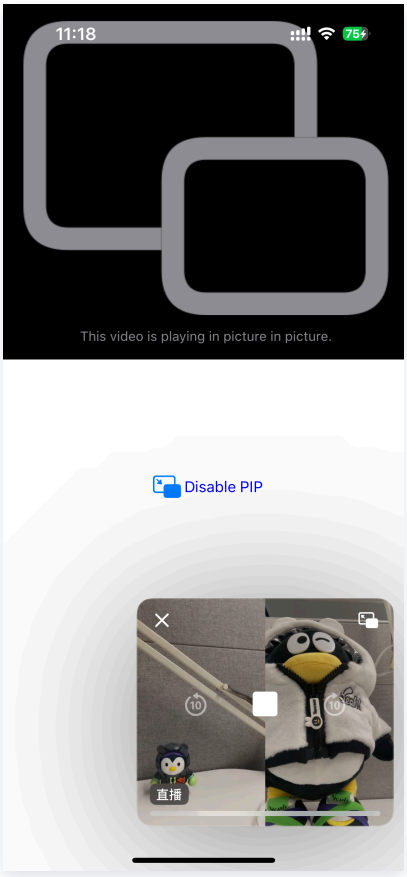
swift

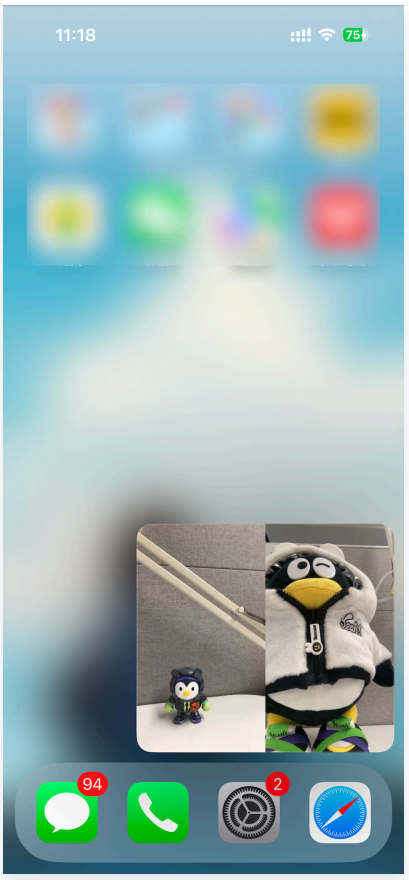
```
let param: [String : Any] = ["api":
"enablePictureInPictureFloatingWindow", "params": ["enable":true]]
if let jsonData = try? JSONSerialization.data(withJSONObject: param,
options: .fragmentsAllowed) {
    let paramJsonString = String.init(data: jsonData, encoding: .utf8)
?? ""
    trtcCloud.callExperimentalAPI(paramJsonString)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

無効化する必要がある場合は、対応するパラメータ位置にfalseを入力してください。

システムAPIを呼び出して実現します

ピクチャーインピクチャーはiOSシステムが提供する機能であり、システムAPIを呼び出すことで複雑なシーンのピクチャーインピクチャー機能を実現できます。iOSはピクチャーインピクチャー機能をサポートしていますが、この特性には多くの制限があり、レンダリングビデオのUIViewを直接使用してピクチャーインピクチャーを実現することはできません。カスタムレンダリングを使用し、ピクチャーインピクチャーに表示するビデオを要件に合ったコンポーネントにレンダリングする必要があります。以下では、2人の配信者PKピクチャーインピクチャーのシーンを例に、システムAPIを呼び出してピクチャーインピクチャーを実現する方法を紹介します。





説明:
単一の配信者または2人以上の配信者のピクチャーインピクチャーを実現する場合も、以下のソリューションを参考にすることができます。ここでは、2人の配信者PKのシーンについてのみ説明します。

1. ピクチャーインピクチャーに必要なコンポーネントを定義します。

   iOSシステムは、特定のコンポーネントのみがピクチャーインピクチャーにレンダリングできることを要求するため、ここではAVSampleBufferDisplayLayerを使用します。また、このコンポーネントが対応するビデオを直接レンダリングする必要があるため、2人の配信者のビデオデータを結合するためにcombinedPixelBufferを定義します。

```
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. RTC Engine ルームに入室。

```
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. オーディオを設定し、バックエンドデコードを有効化。

```
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. ピクチャーインピクチャーコンポーネントを初期化。

```
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. カスタムレンダリングを有効化。

> ⚠ **注意：**
> カスタムレンダリングを有効化する際に指定する形式 `._NV12` と ステップ6：左右2つの画面を結合 の方法は関連しており、異なる形式には異なる結合方法が必要です。この例では `._NV12` 形式の左右結合のみを表示しています。

```
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. 左右2つの画面を結合。

2人の配信者のビデオデータを結合する際、SDKがビデオデータをコールバックするタイミングが同期していないため、単一の配信者のビデオデータを受信するたびに対応するデータを更新し、ロックをかける必要があります。複数の配信者を実現する場合も、同様の方法で操作する必要があります。以下のコードは、左右2人の配信者がそれぞれ半分ずつ占めるレイアウト方法です。他のレイアウトが必要な場合は、業務上で必要に応じて実装する必要があり、ここではSDKには関わりません。

```
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. 結合後の画面を対応するコンポーネントにレンダリングします。

```
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. リモートユーザーのビデオデータを取得し、結合後に指定されたコンポーネントにレンダリングします。
   サンプルでは left を使用して左側に表示される配信者IDを識別していますが、実際の業務では業務ニーズに応じて変更する必要があります。

```
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. 関連プロトコルを実現します。

```
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. ピクチャーインピクチャーを開始/終了します。

```
// ピクチャーインピクチャーを終了します
pipController?.stopPictureInPicture()
// ピクチャーインピクチャーを開始します
pipController?.startPictureInPicture()
```

> ⚠ 注意：
> - ここでは実現ソリューションについてのみ説明し、実際の業務では様々な可能性のある異常ケースへの対応も必要です。
> - ピクチャーインピクチャーの上層制御ボタンの処理はiOSシステム関連の機能であり、SDKには関わらないため、ここでは説明しません。業務側で実際の必要に応じて対応する必要があります。

Android端末のピクチャーインピクチャーの実現

Android 8.0（APIレベル26）以降、Androidはactivityをピクチャーインピクチャー（PIP）モードで起動することを許可しています。ピクチャーインピクチャーは、特殊なタイプのマルチウィンドウモードで、主にビデオ再生に使用されます。このモードを使用すると、ユーザーは画面の隅に固定された小さなウィンドウでビデオを視聴しながら、アプリケーション間を移動したり、ホーム画面のコンテンツを閲覧したりできます。RTC Engine SDKはAndroidのピクチャーインピクチャー APIをさらにカプセル化しておらず、ピクチャーインピクチャー機能はAndroid APIを直接呼び出して実現されています。詳細については、Androidドキュメント ピクチャーインピクチャー（PIP）機能を使用してビデオを追加 をご参照ください。
Android端末はピクチャーインピクチャーを開始すると、xmlのレイアウトルールに基づき、ピクチャーインピクチャーのウィンドウサイズでmeasure、layoutを再度実行します。そのため、配信者側と視聴者側はこのルールに従ってピクチャーインピクチャーを実現できます。

ピクチャーインピクチャーの実現

以下はAndroidドキュメント ピクチャーインピクチャー（PIP）機能を使用してビデオを追加 に基づいて実現されます。

1. AndroidManifest.xmlで<activity>に対してピクチャーインピクチャー属性を宣言します。

```
<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"` はピクチャーインピクチャーをサポートすることを宣言します。
- ピクチャーインピクチャーモードの切り替え中にレイアウト変更が発生した場合、activityの再起動を防ぐには、android:configChanges属性に対応する値を設定する必要があります。

2. ピクチャーインピクチャーを開始します。

```
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);
        //ピクチャーインピクチャーを開始します
        enterPictureInPictureMode(pictureInPictureBuilder.build());
    } else {
        Toast.makeText(this,
R.string.picture_in_picture_not_supported, Toast.LENGTH_SHORT).show();
    }
}
```

- `pictureInPictureBuilder.setAspectRatio(aspectRatio);` ピクチャーインピクチャーのアスペクト比を設定します。ここでは再生ビデオViewのアスペクト比に設定します。
- `enterPictureInPictureMode(pictureInPictureBuilder.build());` ピクチャーインピクチャーを開始します。

3. ピクチャーインピクチャー開始/終了のコールバック。

```
@Override
```

```
public void onPictureInPictureModeChanged(boolean
isInPictureInPictureMode, Configuration configuration) {
    super.onPictureInPictureModeChanged(isInPictureInPictureMode,
configuration);
    if (isInPictureInPictureMode) {
        //ピクチャーインピクチャーを開始した後、非表示にする必要があるview
    } else{
        //ピクチャーインピクチャーを終了した後、表示する必要があるview
    }
}
```

ピクチャーインピクチャーで複数のビデオ画面を表示します

複数のビデオ画面を表示したい場合、ピクチャーインピクチャーを開始する時に、View Aに固定の幅と高さを設定し、他のViewはレイアウトルールに従って表示されるか、パーセンテージレイアウトを設定します。

> **説明:**
> ピクチャーインピクチャーで複数のビデオ画面を表示する方法はAndroidの規定された使い方ではなく、現在Android 12で使用可能ですが、今後のAndroidシステムの更新に伴い変更される可能性があります。リリース前に各バージョンのシステムとの互換性をテストする必要があります。

効果展示


```
// mTRTCCloudは左側のビデオView（TXCloudVideoView）に対応し、
TRTC_VIDEO_RENDER_MODE_FITを設定しています
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は右側のビデオビュー（TXCloudVideoView）に対応します
mTRTCCloud.startRemoteView(remoteUserIdB,
TRTCCloudDef.TRTC_VIDEO_STREAM_TYPE_BIG, mTXCloudRemoteView);
```

ピクチャーインピクチャーを開始した後のレイアウトでは、TXCloudVideoViewの幅と高さを計算して手動で設定するか、フィルモードを設定することで、ビデオ画面の完全な表示を保証できます。mTRTCCloud（TRTCCloudオブジェクト）のsetRemoteRenderParamsメソッドを呼び出して、ビデオ画面のフィルモードを設定します。

- ピクチャーインピクチャーの左側のTXCloudVideoViewは、TRTC_VIDEO_RENDER_MODE_FITを設定した効果です。
- ピクチャーインピクチャーの右側のTXCloudVideoViewは、TRTC_VIDEO_RENDER_MODE_FILLを設定した効果です。

この例では、2つのビデオ画面（TXCloudVideoView）のみがあり、左側のTXCloudVideoViewに幅と高さを設定すると、右側のTXCloudVideoViewはレイアウトルールに従って表示されます。複数のTXCloudVideoViewがある場合、レイアウトを適切に設計して目的の効果を達成できます。

実現手順

1. activity_picture_in_picture.xml に2つの TXCloudVideoView を並べて表示するように追加します。

```
<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. ピクチャーインピクチャー開始時と終了時に、video_viewの幅と高さを設定します。

```
@Override
public void onPictureInPictureModeChanged(boolean
isInPictureInPictureMode, Configuration configuration) {
    super.onPictureInPictureModeChanged(isInPictureInPictureMode,
configuration);
    if (isInPictureInPictureMode) {
        // mVideoViewの幅を100dpに設定します
        RelativeLayout.LayoutParams layoutParams =
(RelativeLayout.LayoutParams) mVideoView.getLayoutParams();
        layoutParams.width = (int)
TypedValue.applyDimension(TypedValue.COMPLEX_UNIT_DIP, 100,
```

```
getResources().getDisplayMetrics());
    } else {
        // ピクチャーインピクチャークを終了し、video_viewの幅を元に戻します
        RelativeLayout.LayoutParams layoutParams =
(RelativeLayout.LayoutParams) mVideoView.getLayoutParams();
        layoutParams.width = (int)
TypedValue.applyDimension(TypedValue.COMPLEX_UNIT_DIP, 192,
getResources().getDisplayMetrics());
    }
}
```

Flutter側の視聴者ピクチャーインピクチャー実現

Flutter側でピクチャーインピクチャーを有効化するには、プラットフォームごとに異なる実現方法があります。以下では、iOSとAndroidプラットフォームについてそれぞれ説明します。

iOSデバイスにリリース

SDKを呼び出して実現

Flutter側でも、SDKが提供するAPIを呼び出すことで簡単にピクチャーインピクチャーを有効化できます。ネイティブのiOS端末と同様に、SDKは単一の配信者のピクチャーインピクチャー表示機能のみを提供しています。複数の配信者のビデオをピクチャーインピクチャーで表示する必要がある場合は、システムAPIを呼び出して実現 をご参照ください。

> ⚠ 注意：
> 同様に、Flutterで生成されたiOSプロジェクトで対応する権限を有効化する必要があります。詳細は、本文の 対応する権限を有効化 部分をご参照ください。

RTC Engine 再生

Flutter SDKはバージョン2.9.1以降が必要で、視聴者側で以下のインターフェースを呼び出して実現します。

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

無効化する必要がある場合は、対応するパラメータ位置にfalseを入力してください。

ライブストリーミング再生

視聴者側で以下のインターフェースを呼び出して有効化します。

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

無効化する必要がある場合は、対応するパラメータ位置に**false**を入力してください。

システムAPIを呼び出して実現します

複雑なピクチャーインピクチャー機能（例：ピクチャーインピクチャーで複数の配信者のビデオを表示するなど）を実現する必要がある場合は、iOSシステムが提供するAPIを呼び出して実現する必要があります。iOS端末の視聴者ピクチャーインピクチャー実現–システムAPIを呼び出して実現の部分をご参照ください。以下では、Flutter側でiOSシステムAPIを呼び出す部分について説明します。

1. Flutter 側は MethodChannel を使用して iOS ネイティブ端末にメッセージを送信します。

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

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

2. Flutterでパッケージ化されたiOSプロジェクトで、対応するメッセージを受信し、適切な処理を行います。

   FlutterがシステムAPIを呼び出して複数配信者PKのピクチャーインピクチャーを実現する場合、実際にはiOSシステムAPIを呼び出し、カスタム収集機能を使用して2つの配信者PKの画面を再描画し、Flutterレイヤーの上に表示します。そのため、iOSシステムAPIを呼び出して描画するウィンドウのサイズと位置がFlutter側と一致する必要があり、MethodChannelで対応するレイアウトパラメータと配信者IDを渡すことができます。

```
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. ピクチャーインピクチャーを処理するクラスを定義します。

   ピクチャーインピクチャーを有効化する際、対応する配信者のストリームをカスタムレンダリングに変更し、レンダリング後の画面をルートビューに挿入して表示します。

```
import UIKit
import AVKit
import TXLiteAVSDK_Professional

class PipRender: NSObject {
    // その他の変数はiOSネイティブ端末でシステムAPIを呼び出して実現する部分をご参
照ください
    var mainView: UIView?
    var mt: CGFloat?
```

```
    // trtcCloudはシングルトンインスタンスであるため、コード内でこのように取得で
きます
    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);
        }
    }

    // このメソッドは、業務ニーズに応じてピクチャーインピクチャーの表示位置を調整
し、Flutter側の表示位置と一致させる必要があります
    func setupPipController() {
        let screenWidth = UIScreen.main.bounds.width
        let videoHeight = screenWidth / 2 / 9 * 16
        pipDisplayLayer = AVSampleBufferDisplayLayer()
        // ここでは実際のニーズに応じて、ピクチャーインピクチャーの表示位置を調整
します
```

```
        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)
        // ここではenablePipで渡されたmainViewを使用してピクチャーインピク
チャーの画面を追加します
        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")
        }
    }

    // その他のメソッドは、iOSネイティブ端末でシステムAPIを呼び出して実現する部分
と一致します
}
```

4. Flutter側はピクチャーインピクチャーを停止する際に対応する配信者のストリームを再取得し、Flutter側のレンダリングを復元する必要があります。

```
// ビジネス上でピクチャーインピクチャーを停止するトリガーが発生した場合
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側は現在のページを破棄する際にピクチャーインピクチャーを停止する必要があります。

   ピクチャーインピクチャーを開始すると、実際にはiOSシステムAPIを呼び出して新しいビューを再描画し、Flutterのビューの上に重ねるため、現在のページを破棄する際にはピクチャーインピクチャーを停止し、対応するビューをルートビューから削除する必要があります。

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

AndroidでFlutterを通じてピクチャーインピクチャーを実現

Flutterでピクチャーインピクチャーを実現する場合も、Androidのピクチャーインピクチャー APIを呼び出す必要があります。ピクチャーインピクチャーを開始すると、Flutter UIは既存のWidgetレイアウトルールに従って表示され、業務ルールに基づいてピクチャーインピクチャー開始後に一部のWidgetを非表示にし、ビデオWidgetの幅と高さを適切に設定することができます。
プラットフォームチャネルを使用してAndroidのコードを呼び出し、チャネルはクライアント（Flutter）と配信者（Android）に分かれており、以下にピクチャーインピクチャーの具体的な実現方法を示します。

1. Flutterクライアントのコード：チャネル名「samples.flutter.dev」を使用してチャネルメソッド「pictureInPicture」を呼び出します。このメソッドの具体的な実現はAndroid配信者側にあります。

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

2. Android配信者側のコード。

   2.1 FlutterActivityを継承したActivityで実現します：

```
private void startPictureInPicture() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        PictureInPictureParams.Builder pictureInPictureBuilder = new
PictureInPictureParams.Builder();
        //具体的な業務ニーズに応じて、指定されたピクチャーインピクチャーのサイズを
設定します
```

```
        Rational aspectRatio = new Rational(100, 100);
        pictureInPictureBuilder.setAspectRatio(aspectRatio);
        //ピクチャーインピクチャーを開始します
        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 AndroidManifest.xmlでactivityにピクチャーインピクチャーのパラメータ `android:supportsPictureInPicture="true"` を設定します。以下の通り：

```
<activity

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

android:configChanges="orientation|keyboardHidden|keyboard|screenSize|
```

```
smallestScreenSize|locale|layoutDirection|fontScale|screenLayout|densi
ty|uiMode"
    >
    ...
</activity>
```

ピクチャーインピクチャーで2つのビデオ画面（例：配信者同士のPK）を表示したい場合は、レイアウトルールとサイズを適切に設定することで実現できます。

ライブストリーミング上下スワイプ

最終更新日：：2025-11-18 15:15:34

ライブストリーミングシーンでは、多数の配信者による多様な動画コンテンツが提供されており、上下にスワイプして素早く閲覧し、好みのコンテンツを選択することで、ユーザーに優れた使用体験を提供できます。本文では、Android端末とiOS端末それぞれのソリューションについて紹介します。

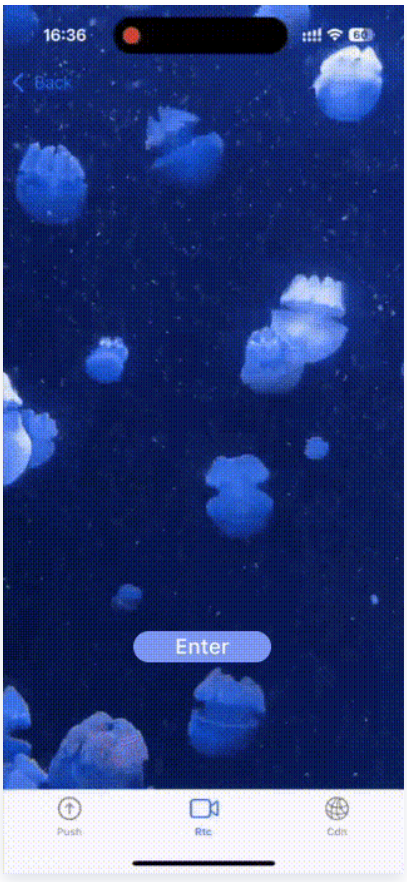
iOS端末のライブストリーミング上下スワイプソリューション

このシーンでは、ライブストリーミングルームを切り替える際、新しいルームに入室してコンテンツを受信するのに時間がかかるため、前後のルームの動画映像が途切れる可能性があります。この問題に対して、一般的に以下の3つのソリューションがあります。本文では、これら3つのソリューションについてそれぞれ詳しく説明します。

実現方法	ユーザー体験	リソース消費	実現ロジック
黒画面	一般、ライブストリーミングルームを切り替える際には、まず黒画面が表示され、その後新しい画面が表示されます	追加リソース消費なし	簡単、追加ロジックなし
プレースホルダー画像	やや良い、ライブストリーミングルームを切り替える際には、まず対応する配信者の固定プレースホルダー画像が表示され、その後新しい画面が表示されます	やや多い、各配信者に対してプレースホルダー画像を追加で保存し、クライアントに読み込む必要があります	やや複雑、切り替え前に非同期でプレースホルダー画像の読み込み完了が必要です
デュアルインスタンス	最も良い、ライブストリーミングルームを切り替える際には、前後の2人の配信者の画面がスムーズに表示されます	比較的に多い、リスト内では2つのストリームを同時に受信する必要があり、ライブストリーミングルームに入った後は1つのストリームを受信するだけで十分です	比較的に複雑、複数のインスタンスを使用し、異なるインスタンスのオーディオ・ビデオ受信を制御する必要があります

黒画面

システムのスワイプイベントを監視し、ルーム切り替えのインターフェースを呼び出してライブストリーミングルームを切り替えます。新しいライブストリーミングルームが読み込まれる前に、コンテンツが表示されず、一時的な黒画面が表示されます。説明の便宜上、ここでの黒画面時間は約1秒としますが、具体的な黒画面時間はネットワークとビデオビットレートの影響を受けます。効果は以下の通りです。



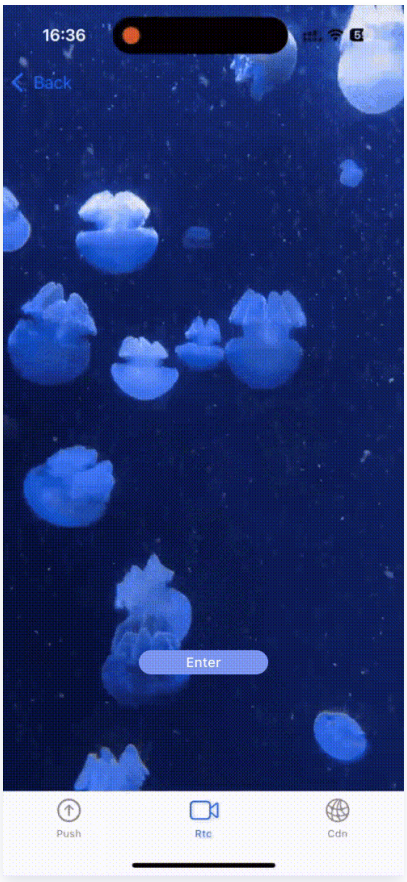
ルーム切り替えのコードスニペットは以下の通りです。

```
let src = TRTCSwitchRoomConfig()
// 業務に応じて対応するルーム番号と入室証明書を生成します。サンプルではクライアント側
で入室証明書を生成していますが、オンライン業務ではバックエンドから取得してください
src.strRoomId = strRoomId
src.userSig = GenerateTestUserSig.genTestUserSig(identifier: userId) as
String
trtcCloud.switchRoom(src)
```

プレースホルダー画像

システムのスワイプイベントを監視し、ルーム切り替えのインターフェースを呼び出してライブストリーミングルームを切り替えますが、黒画面ソリューションとは異なり、このソリューションでは各ライブストリーミングルームのプレースホルダー画像を事前に読み込む必要があります。ライブストリーミングルームのビデオストリームが表示される前に、対応するライブストリーミングルームのプレースホルダー画像が表示されます。説明の便宜上、ここではプレースホルダー画像の表示時間を約1秒としますが、具体的な時間はネットワークとビデオビットレートの影響を受けます。

効果図



実現手順

1. 最初の配信者の背景画像を設定します。

```
bgView = UIImageView(frame: self.view.bounds)
// この画像は対応するライブストリーミングルームのプレースホルダー画像である必要があり、業務上事前に取得しておく必要があります
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. ライブストリーミングルームを切り替える前に背景画像を切り替えます。

```
DispatchQueue.main.async {
    UIView.transition(
            with: self.bgView,
            duration: 0,
            options: .transitionCrossDissolve,
            animations: {
                // 業務上で対応するプレースホルダー画像を切り替えます
                self.bgView.image = UIImage(named: strRoomId)
            }, completion: nil)
}

let src = TRTCSwitchRoomConfig()
// 業務に応じて対応するルーム番号と入室証明書を生成します。サンプルではクライアント側で入室証明書を生成していますが、オンライン業務ではバックエンドから取得してください
src.strRoomId = strRoomId
src.userSig = GenerateTestUserSig.genTestUserSig(identifier: userId) as String
trtcCloud.switchRoom(src)
```

3. 新しいライブストリーミングルームの最初のフレームのビデオ画面がレンダリングを開始した時に、背景画像をビデオ画面に切り替えます。

```
// ビデオストリームを受信
func onUserVideoAvailable(_ userId: String, available: Bool) {
    if available {
        trtcCloud.startRemoteView(userId, streamType: .big, view: view)
    } else {
        trtcCloud.stopRemoteView(userId, streamType: .big)
    }
}

// 最初のフレームのビデオ画面がレンダリングを開始した時に、プレースホルダー画像を背景に切り替えて、ビデオ画面を表示します
func onFirstVideoFrame(_ userId: String, streamType: TRTCVideoStreamType, width: Int32, height: Int32) {
    // ここでは、背景画像とビデオレンダリングコントロールの前後順序を調整します。実際の業務では状況に応じて調整してください
    self.view.exchangeSubview(at: 1, withSubviewAt: 0)
```

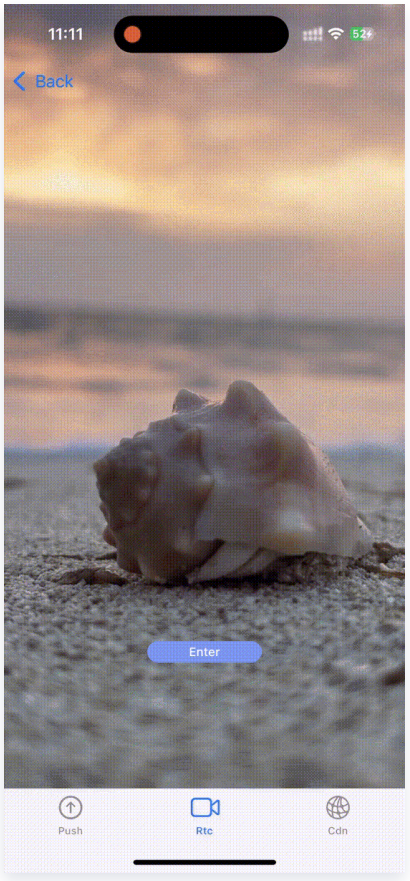
```
}
```

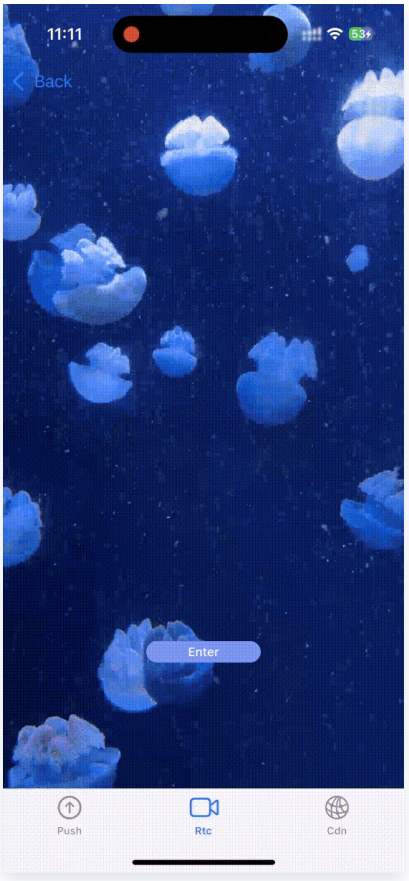
デュアルインスタンス

> **注意：**
> このソリューションは表示効果が最良で、ユーザー体験も最高ですが、スワイプリストでは前後の2つのライブストリーミングルームの2つのストリームを同時に受信する必要があります。ユーザーがライブストリーミングルームに入った時に次のライブストリーミングルームのストリームのプリロードを停止できますが、全体的なトラフィックとコスト消費はより多くなります。

効果図

最もスムーズな上下スワイプ効果を実現するためには、2つのインスタンスを同時に使用し、前のライブストリーミングルームを視聴しながら次のライブストリーミングルームのビデオ画面をプリロードし、UIPageViewControllerまたは手動でスワイプ位置に基づいて上下2つのビデオの表示位置を調整し、自然でスムーズなトランジションを実現します。次のライブストリーミングルームをスワイプして視聴する効果は下図左側の例の通りです。





このソリューションの全体的なロジックは、現在のライブストリーミングルームに入室後、すぐにサブインスタンスを使用して次のライブストリーミングルームに入室し、次のライブストリーミングルームのビデオストリームを受信し、次のライブストリーミングルームのビデオストリームをUIPageViewControllerの次のPageに表示することです。新しいライブストリーミングルームに入室した時に、新しいライブストリーミングルームのオーディオス

トリームを開始し、このように循環することで、最もスムーズな上下スワイプ効果を達成できます。同時に、過剰なリソース消費を避けるために、次のライブストリーミングルーム視聴のみをプリロードし、使用頻度の少ない前のライブストリーミングルーム視聴はプリロードせず、切り替え時には依然として黒画面を使用します。前のライブストリーミングルームを視聴する効果は上図右側の例の通りです。

実現手順

1. サブインスタンスツールクラスを定義します。

```
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. サブインスタンスを使用します。

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

override func viewDidLoad() {
    super.viewDidLoad()

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

3. 切り替え用のUIPageViewControllerを準備します。

```
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. pageViewControllerのページ切り替えを実現します。

```
// 次/前のPageを取得
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. メインインスタンスでルームに入室し、サブインスタンスを使用して次のライブストリーミングルームをプリロードします。

```
// メインインスタンスでルームに入室
// 実際の業務に応じてsdkAppId、roomID、strRoomId、userId、userSigを置き換えてください
// サンプルではクライアント側でuserSigを生成しますが、オンライン業務ではバックエンドから取得してください
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)

// サブインスタンスでプリロードし、次のルームに入室
// 実際の業務に応じてsdkAppId、roomID、strRoomId、userId、userSigを置き換えてください
// サンプルではクライアント側でuserSigを生成しますが、オンライン業務ではバックエンドから取得してください
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. コールバックに基づいてビデオストリームを受信し、対応する page にレンダリングします。

```
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. 新しいルームに切り替えた後、プリロードされたルームを更新するか、上にスワイプした時に、現在表示されているルームを更新します。

```
func updateCurRoomIdx(isNext: Bool) {
    if isNext {
        curRoomIdx += 1
        if curRoomIdx >= strRoomIdLst.count {
            curRoomIdx = 0
        }
    } else {
        curRoomIdx -= 1
        if curRoomIdx < 0 {
            curRoomIdx = strRoomIdLst.count - 1
        }
    }
}
// ここでは実際の業務ロジックに基づいてルーム番号を切り替える必要があります
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) {
                // 上下スワイプの判断基準を取得
                let iden = index - curPageIdx
                // 現在表示されているページのシーケンス番号を更新
                curPageIdx = index
                // 下にスワイプ
                if iden == 1 || iden == -2 {
                    // 現在いるルームのシーケンス番号を更新
                    updateCurRoomIdx(isNext: true)
                    // 現在表示されているページのインスタンスを更新
                    curIsSub.toggle()
                    // ルームを更新
                    updateNewRoom(isNext: true)
                }
                // 上にスワイプ
                if iden == -1 || iden == 2 {
                    // ルームを更新
                    updateNewRoom(isNext: false)
                    // 現在いるルームのシーケンス番号を更新
                    updateCurRoomIdx(isNext: false)
                    // 現在表示されているページのインスタンスを更新
                    curIsSub.toggle()
                    trtcCloud.muteAllRemoteAudio(true)
                    subCloudHelper.trtcCloud.muteAllRemoteAudio(true)
                }
                // 現在のルームのミュートを解除
                if curIsSub {
                    subCloudHelper.trtcCloud.muteAllRemoteAudio(false)
```

```
                } else {
                    trtcCloud.muteAllRemoteAudio(false)
                }
            }
        }
    }
}
```

さらに、業務上では「スワイプリスト内」と「ライブストリーミングルーム入室」という2つの状態を区別することも可能です。上下にスワイプする際には通常、ルームの詳細やチャットなどの情報は不要であり、「ライブストリーミングルーム入室」後にのみ表示が必要となるためです。このように区別することで、頻繁な上下スワイプによるライブストリーミングルーム状態の業務記録負荷を軽減し、同時にプリロードによるリソース消費を削減できます。
具体的な方法：「スワイプリスト内」でのみ上下スクロールによるライブストリーミングルーム切り替えを許可し、この時はライブストリーミングルームの画面と少量の情報のみを表示します。「ライブストリーミングルーム入室」時には完全なライブストリーミングルームとチャットなどの情報を表示し、この時は上下スワイプによるライブストリーミングルーム切り替えを禁止します。また、「ライブストリーミングルーム入室」時に次のライブストリーミングルームのビデオストリームのプリロードを停止し、「スワイプリスト内」に戻ると同時に次のライブストリーミングルームのビデオストリームのプリロードを再開します。効果は以下の通りです。

具体的な実現は次の通りです。

```
// ライブストリーミングルーム入室ロジックを処理
@objc func enterBtnClick() {
    // 上下スワイプリストのUIコンポーネントを非表示
    self.enterBtn.isHidden = true
    // ライブストリーミングルーム入室後のUIコンポーネントを表示
    self.exitBtn.isHidden = false
    // ...

    self.atRoom = true
    // ライブストリーミングルーム入室後、上下のスワイプを禁止
    pageViewController.dataSource = nil

    // プリロードを停止
    if curIsSub{
        trtcCloud.muteAllRemoteVideoStreams(true)
    } else {
        subCloudHelper.trtcCloud.muteAllRemoteAudio(true)
    }
}

// ライブストリーミングルーム退出ロジックを処理
@objc func exitBtnClick() {
    // ライブストリーミングルームのUIコンポーネントを非表示
    self.enterBtn.isHidden = false
    // 上下スワイプリストのUIコンポーネントを表示
    self.exitBtn.isHidden = true

    self.atRoom = false
    // 上下スワイプリストに進んだ後、上下スワイプを復元
    pageViewController.dataSource = self
    // プリロードを再開
    if curIsSub{
        trtcCloud.muteAllRemoteVideoStreams(false)
    } else {
        subCloudHelper.trtcCloud.muteAllRemoteAudio(false)
    }
}
```

Android端末のライブストリーミング上下スワイプソリューション

以下のデュアルインスタンスとシングルインスタンスのセクションでは、効果図とサンプルコードに3つのページがあり、ページの順序はA > B > Cで、Aはルーム1231、Bはルーム1232、Cはルーム1233に対応しています。

実現方法	ユーザー体験	リソース消費	実現ロジック
シングルインスタンス	一般、リストをスワイプする際、2つのライブストリーミングを同時に見ることはできず、ページが完全に切り替わった後に対応するライブストリーミングが表示されます。プレースホルダー画像を使用してユーザー体験を向上させることができます。	リストには1ストリームのトラフィック消費のみがあり、1つのビデオ再生オブジェクトが使用されています。	簡単、必要に応じてプレースホルダーを設定します。
デュアルインスタンス	比較的に良い、同時に2つのライブストリーミングルームに入室し、次のライブストリーミングを事前にロードし、リストをスワイプした時に、現在と次のライブストリーミングルームを同時に表示できます。	リストには2ストリームのトラフィック消費があり、2ストリームの費用が発生し、2つのビデオ再生オブジェクトが使用されます。	複雑、複数のインスタンスを使用し、異なるインスタンスのオーディオ・ビデオ受信を制御する必要があります。

シングルインスタンス

ライブストリーミングリストを上下にスワイプすると、スワイプ中は単一のライブストリーミング映像（シングルインスタンス）しか表示されず、コストを節約できます。

効果図

Aページをスワイプ中は、Bページのライブストリーミング映像を同時に表示できず、Bページに切り替えるとBページのライブストリーミング映像が表示され、Aページのライブストリーミング映像は表示されません。

ソリューション原理

ページをスワイプ中は、同時に表示できるライブストリーミング映像は1つだけで、ページを切り替えると前のライブストリーミング映像を停止し、次のライブストリーミング映像を受信します。
AページからBページへスワイプする際の各段階の操作と状態は以下の通りです。

1. Aページが画面に表示されている時、TRTCCloudインスタンス1を使用し、ルーム1231に入室してストリームを受信し、オーディオ・ビデオを再生し、Aページに表示します。
2. AページからBページへスワイプする過程で、Bページに切り替えるコールバックを受信していないため、Aページは引き続きルーム1231のオーディオ・ビデオストリームを正常に再生し、Bページはプレースホルダー画像または黒画面を表示します。
3. AページからBページへスワイプする過程で、Bページに切り替えるコールバックを受信し、TRTCCloudインスタンス1を使用し、ルーム1231のオーディオ・ビデオストリームの受信を停止して退室し、ルーム1232に入室してストリームを受信し、オーディオ・ビデオを再生し、Bページに表示します。Aページはプレースホルダー画像または黒画面を表示します。

実現コード

ViewPager2、RecyclerView.Adapterを使用して全画面スワイプ効果を実現します。ViewPager2のregisterOnPageChangeCallbackのonPageSelectedコールバックで、受信を停止し、退室し、入室し、受信を開始します。以下にScrollSwitchRoomActivityの完全なコードを示します。レイアウトファイルは前節と同じです。

ScrollSwitchRoomActivity のコードは以下の通りです。

```
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);

        //タイトルバーを非表示
        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);
        //viewPagerのスワイプ方向を設定
        viewPager.setOrientation(ViewPager2.ORIENTATION_VERTICAL);
        //viewPagerのプリロードを設定
        viewPager.setOffscreenPageLimit(1);
```

```
        // ページ切り替えのリスナーを追加
        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);
                // 先に現在のルームから退室
                exitRoom();
                // 次のルームに入室
                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 は以下の通り。

```
<?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 は以下の通り。

```
<?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>
```

デュアルインスタンス

ライブストリーミングリストを上下にスワイプすると、2つのTRTCCloudインスタンスが同時に2つのルーム（現在のルームと次のルーム）に入室し、スワイプ中に両方のルームのライブストリーミングを同時に視聴できます。現在のルームから下にスワイプする過程では、前のルームのライブストリーミングを同時に視聴することはできず、ページが完全に切り替わった後にのみ視聴可能です。必要に応じて、3つのTRTCCloudインスタンスを自ら使用して実現できます。

> **注意：**
> デュアルインスタンスは2つのルームのストリームを同時に受信し、2ストリームのトラフィック消費が発生するため、より多くの費用がかかります。3つのTRTCCloudインスタンスを自ら実現した場合、3ストリームのトラフィック消費が発生します。

効果図

Bページをスワイプ中は、Cページのライブストリーミング映像を直接見ることができます。

ソリューション原理

ページをスワイプ中は、最大2つのページを同時に表示でき、2つのTRTCCloudインスタンスを使用し、同時に2つのルームに入室し、2つのルームのストリームを同時に受信できます。
AページからBページへスワイプする際の各段階の操作と状態は以下の通りです。

1. Aページが画面に表示される時、TRTCCloudインスタンス1を使用し、ルーム1231に入室してストリームを受信し、オーディオ・ビデオを再生し、Aページに表示します。同時に、TRTCCloudインスタンス2を使用し、ルーム1232に入室してストリームを受信し、ビデオを再生し、Bページに表示し、オーディオをミュートします。
2. AページからBページへスワイプする過程で、この時、AページとBページが同時にビデオを再生しているのを見ることができ、ルーム1231のオーディオを聞くことができます。
3. Bページが完全に表示された後、TRTCCloudインスタンス2を使用し、ルーム1232のオーディオを開始し、ビデオを引き続き再生します。TRTCCloudインスタンス1を使用し、ルーム1231から退室し、ルーム1233に入室してストリームを受信し、ビデオを再生し、Cページに表示し、オーディオをミュートします。
4. BページからAページへスワイプする過程で、ルーム1232のビデオのみを見ることができます。この時、TRTCCloudインスタンス2はBページで使用され、TRTCCloudインスタンス1はCページで使用されているためです。Aページが完全に表示された後、TRTCCloudインスタンス1を使用してルーム1231に入室し、ストリームを受信し、オーディオとビデオを再生します。Bページでは引き続きTRTCCloudインスタンス2を使用してビデオストリームを受信し、オーディオをミュートします。

実現コード

ViewPager2、RecyclerView.Adapterを使用して全画面スワイプ効果を実現します。ScrollSwitchRoomActivityのコードは次の通りです。

```
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);

        //タイトルバーを非表示
        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);

        //viewPagerのスワイプ方向を設定
        viewPager.setOrientation(ViewPager2.ORIENTATION_VERTICAL);
        //viewPagerのプリロードを設定
        viewPager.setOffscreenPageLimit(1);

        // ページ切り替えのリスナーを追加
        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) {
                            // 画面上方向へスワイプ
                            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 {
                            //画面下方向へスワイプ
                            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();
    }

    /**
     * 初期化時のみ、ルーム入室を呼び出し、その後ルームを切り替える場合は、
switchRoomを呼び出します
     */
    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();
```

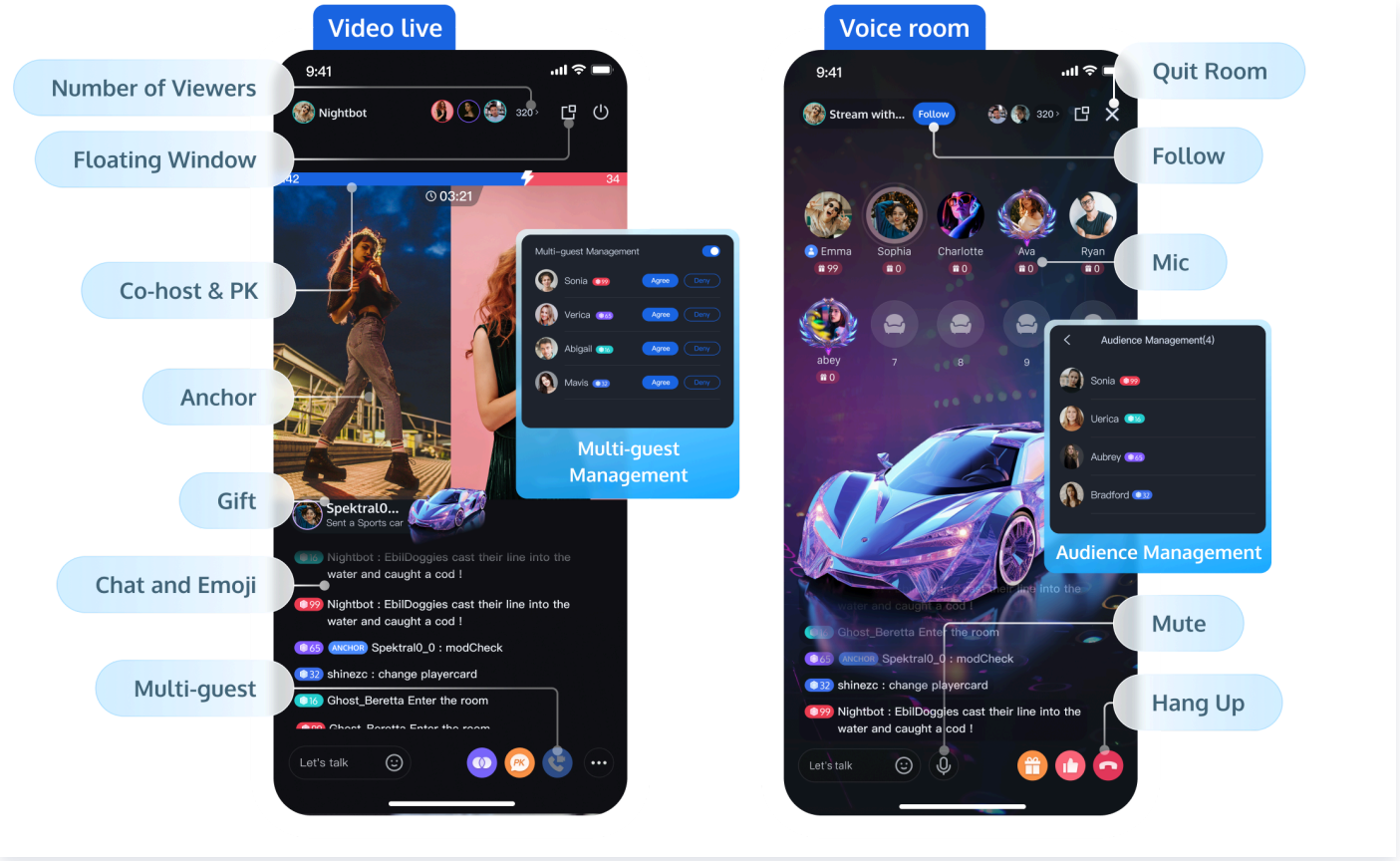
```
        mSubCloud.setListener(null);
    }


    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();
    }
}
```

クロスルーム PK 通話ソリューション

最終更新日：：2025-11-18 15:15:33

ライブストリーミングルームでは、ライブの雰囲気を盛り上げ、素早くファンを増やすために、配信者は他のライブストリーミングルームの配信者を招待して通話インタラクションやオンラインPKを行うことができます。通話中のライブストリーミングルームの視聴者は、複数の配信者のインタラクションを同時に聞いたり見たりできるため、インタラクションなライブストリーミングの楽しさを増し、視聴者がランキングに貢献したりギフトを送りたくなる意欲を刺激します。以下では、3つの異なるクロスルームPK通話ソリューションの具体的な実現方法を紹介します。



通常のクロスルーム PK 通話ソリューション

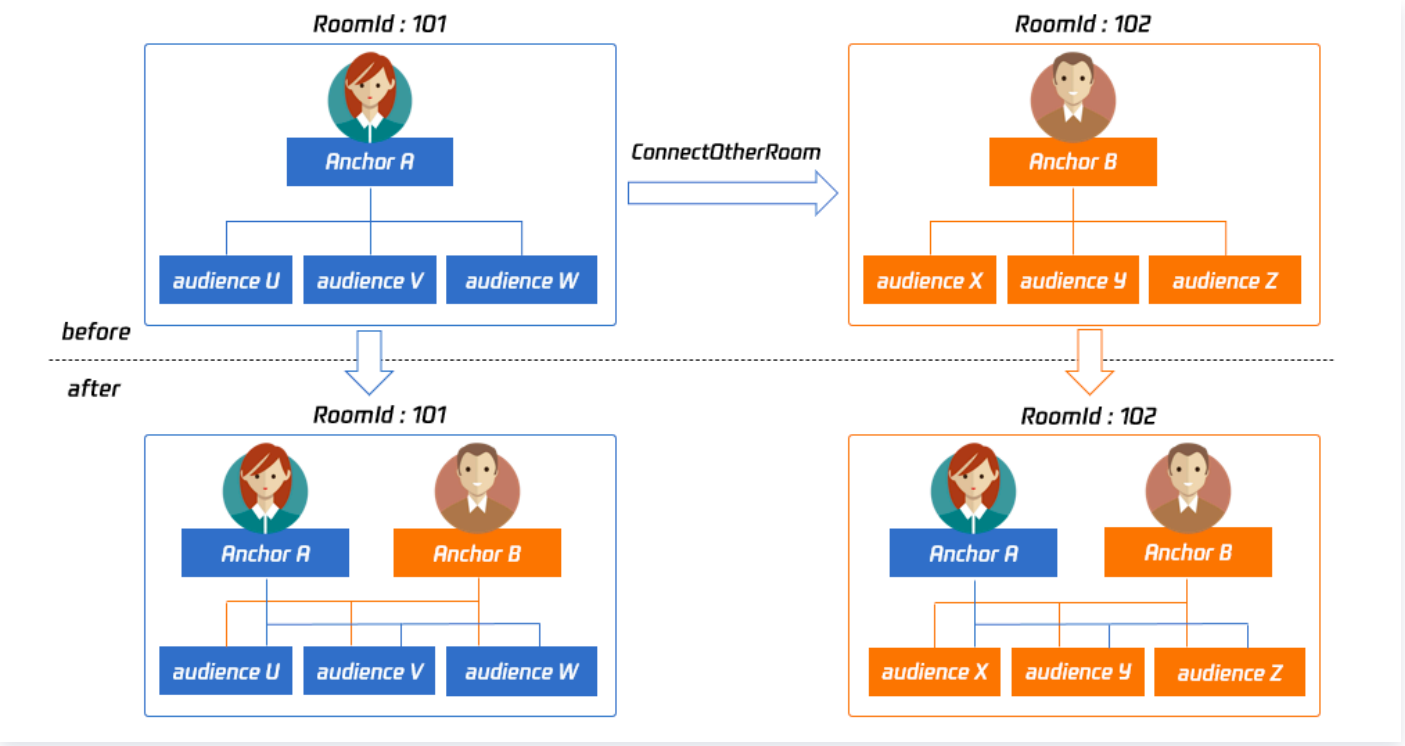
適用シーン

2つ以上のルームでPKを行い、ルーム内の配信者が少ない簡単なクロスルーム通話シーン。

ソリューション原理

デフォルトでは、同じルーム内のユーザー間でのみオーディオ・ビデオが相互接続され、異なるルーム間のオーディオ・ビデオストリームは相互に隔離されています。クロスルーム通話により、別のルームの特定の配信者のオーディオ・ビデオストリームを自分のいるルームに公開することができ、同時に自分のオーディオ・ビデオストリームも対象配信者のルームに公開されます。これにより、2つの異なるルームにいる配信者がルームを跨いで

オーディオ・ビデオストリームを共有し、各ルームの視聴者が両方の配信者のオーディオ・ビデオを視聴できるようになります。



実現フロー

ルーム「101」の配信者AがConnectOtherRoomを通じてルーム「102」の配信者Bとクロスルーム通話を確立した後：

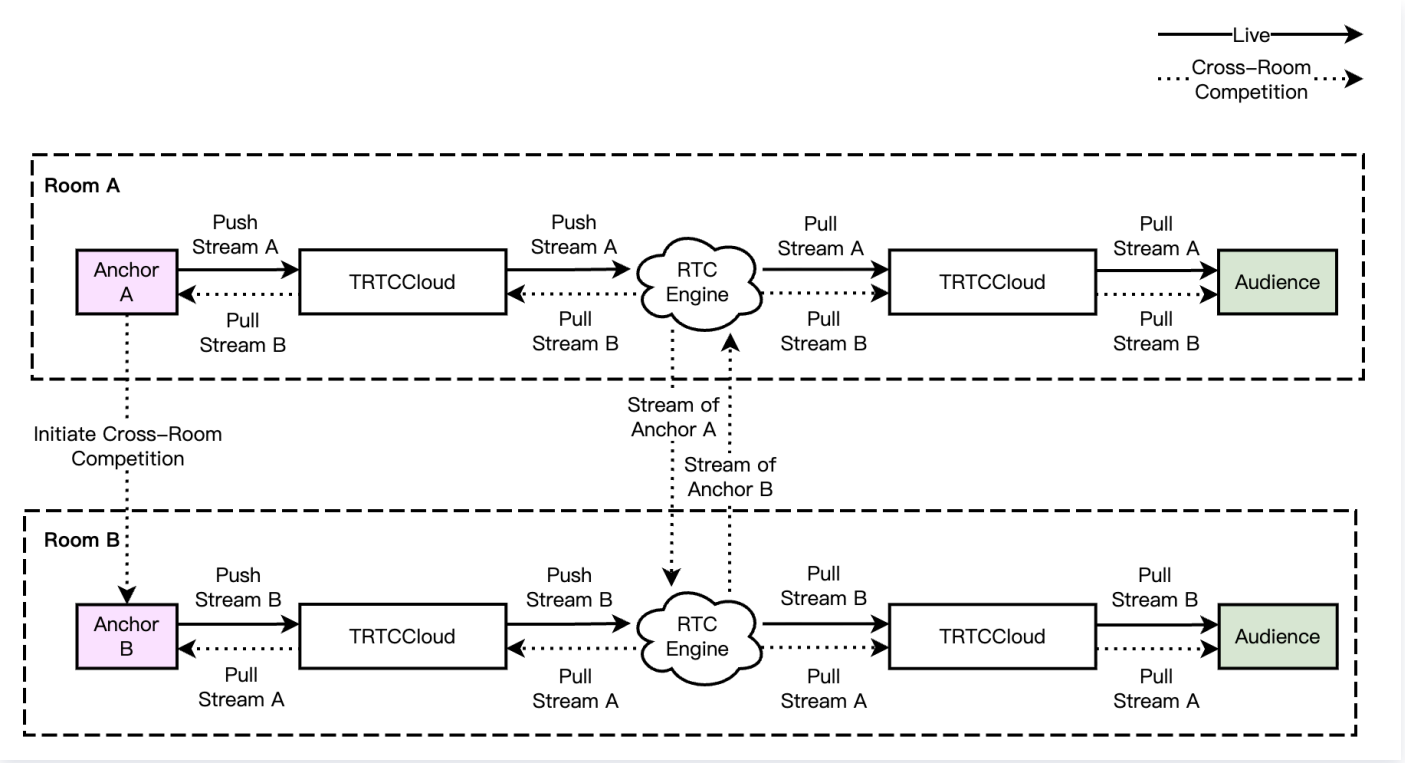
- ルーム「101」のユーザーは、配信者Bの `onRemoteUserEnterRoom(B)` と `onUserVideoAvailable(B,true)` という2つのイベントコールバックを受信します。即ち、ルーム「101」のユーザーは全員、配信者Bのオーディオ・ビデオをサブスクライブできます。
- ルーム「102」のユーザーは、配信者Aの `onRemoteUserEnterRoom(A)` と `onUserVideoAvailable(A,true)` という2つのイベントコールバックを受信します。即ち、ルーム「102」のユーザーは全員、配信者Aのオーディオ・ビデオをサブスクライブできます。

⚠ 注意：

- 2つのルームで単一配信者のクロスルームPKを行う場合、一方のルームの配信者が `ConnectOtherRoom` を呼び出してクロスルーム通話を確立するだけで十分です。双方向での呼び出しは行わないでください。
- 配信者は `ConnectOtherRoom` を複数回呼び出すことで、複数のルームの配信者とクロスルーム通話を確立できます。現在、単一配信者は他のルームの最大9人の配信者とクロスルーム通話を行うことができます。

リアルタイムインタラクションクロスルーム通話

RTCシーンにおけるクロスルーム通話PKフローは全体的に簡単で、配信者とクロスルーム通話配信者が互いにRTCシングルストリームを受信し、視聴者は同時に配信者とクロスルーム通話配信者のRTCシングルストリームを受信します。視聴者は配信者とクロスルーム通話配信者のメディアストリームサブスクリプションロジックを独立して制御できます。リアルタイムインタラクションシーンのクロスルーム通話フローは下図の通りです。



> **注意:**
> リアルタイムインタラクションクロスルーム通話シーンでは、ルーム内の視聴者はクロスルーム通話配信者のメディアストリームサブスクリプションロジックを独立して制御でき、また配信者がクロスルーム配信者の当該ルームにおけるアップリンク能力を変更することも可能です。

サンプルコード

1. いずれか一方がクロスルーム PK 通話を開始します。

Android

```
public void connectOtherRoom(String roomId, String userId) {
    try {
        JSONObject jsonObj = new JSONObject();
        // 文字列ルーム番号を例に、数字ルーム番号 key:roomId
        jsonObj.put("strRoomId", roomId);
        jsonObj.put("userId", userId);
        mTRTCCloud.ConnectOtherRoom(jsonObj.toString());
```

```
    } catch (JSONException e) {
        e.printStackTrace();
    }
}

// クロスルーム通話リクエストの結果コールバック
@Override
public void onConnectOtherRoom(String userId, int errCode, String
errMsg) {
    // userId:  クロスルーム通話するもう一方のルームの配信者のユーザーID
    // errCode: エラーコード、ERR_NULLはリクエスト成功を表します
    // errMsg:  エラーメッセージ
}
```

iOS

```
- (void)connectOtherRoom:(NSString *)roomId {
    NSMutableDictionary *jsonDict = [[NSMutableDictionary alloc] init];
    // 文字列ルーム番号を例に、数字ルーム番号 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];
}

// クロスルーム通話リクエストの結果コールバック
- (void)onConnectOtherRoom:(NSString *)userId errCode:
(TXLiteAVError)errCode errMsg:(NSString *)errMsg {
    // userId:  クロスルーム通話するもう一方のルームの配信者のユーザーID
    // errCode: エラーコード、ERR_NULLはリクエスト成功を表します
    // errMsg:  エラーメッセージ
}
```

⚠ 注意：

- クロスルームPK通話のローカルユーザーと相手側ユーザーは、両方とも配信者ロールであり、かつオーディオまたはビデオのアップリンクが必要です。
- 2つのルームで単一配信者のクロスルームPKを行う場合、一方のルームの配信者が `ConnectOtherRoom` を呼び出してクロスルーム通話を確立するだけで十分です。双方向での呼び出しは行わないでください。

2. 2つのルーム内のすべてのユーザーは、もう一方のルームのPK配信者からのオーディオ・ビデオストリーム利用可能コールバックを受信します。

Android

```
@Override
public void onUserAudioAvailable(String userId, boolean available) {
    // あるリモートユーザーが自身のオーディオを公開/キャンセルしました
    // 自動サブスクリプションモードでは、何も操作する必要はなく、SDKが自動的にリモートユーザーのオーディオを再生します
}

@Override
public void onUserVideoAvailable(String userId, boolean available) {
    // あるリモートユーザーがメインストリームのビデオ画面を公開/キャンセルしました
    if (available) {
        // リモートユーザーのビデオストリームをサブスクライブし、ビデオレンダリングコントロールをバインドします
        mTRTCCloud.startRemoteView(userId, TRTCCloudDef.TRTC_VIDEO_STREAM_TYPE_BIG, view);
    } else {
        // リモートユーザーのビデオストリームのサブスクリプションを停止し、レンダリングコントロールを解放します
        mTRTCCloud.stopRemoteView(userId, TRTCCloudDef.TRTC_VIDEO_STREAM_TYPE_BIG);
    }
}
```

iOS

```
- (void)onUserAudioAvailable:(NSString *)userId available:(BOOL)available {
```

```
    // あるリモートユーザーが自身のオーディオを公開/キャンセルしました
    // 自動サブスクリプションモードでは、何も操作する必要はなく、SDKが自動的にリモートユーザーのオーディオを再生します
}

- (void)onUserVideoAvailable:(NSString *)userId available:(BOOL)available {
    // あるリモートユーザーがメインストリームのビデオ画面を公開/キャンセルしました
    if (available) {
        // リモートユーザーのビデオストリームをサブスクライブし、ビデオレンダリングコントロールをバインドします
        [self.trtcCloud startRemoteView:userId streamType:TRTCVideoStreamTypeBig view:self.remoteView];
    } else {
        // リモートユーザーのビデオストリームのサブスクリプションを停止し、レンダリングコントロールを解放します
        [self.trtcCloud stopRemoteView:userId streamType:TRTCVideoStreamTypeBig];
    }
}
```

3. いずれか一方がクロスルームPK通話を終了します。

Android

```
// クロスルーム通話を終了
mTRTCCloud.DisconnectOtherRoom();

// クロスルーム通話終了結果のコールバック
@Override
public void onDisConnectOtherRoom(int errCode, String errMsg) {
    super.onDisConnectOtherRoom(errCode, errMsg);
}
```

iOS

```
// クロスルーム通話を終了
[self.trtcCloud disconnectOtherRoom];
```

```
// クロスルーム通話終了結果のコールバック
- (void)onDisconnectOtherRoom:(TXLiteAVError)errCode errMsg:(NSString
*)errMsg {
}
```

> ⚠ **注意：**
> クロスルームPK通話の開始側または受信側のいずれかがDisconnectOtherRoomを呼び出してクロスルームPK通話を終了できます。

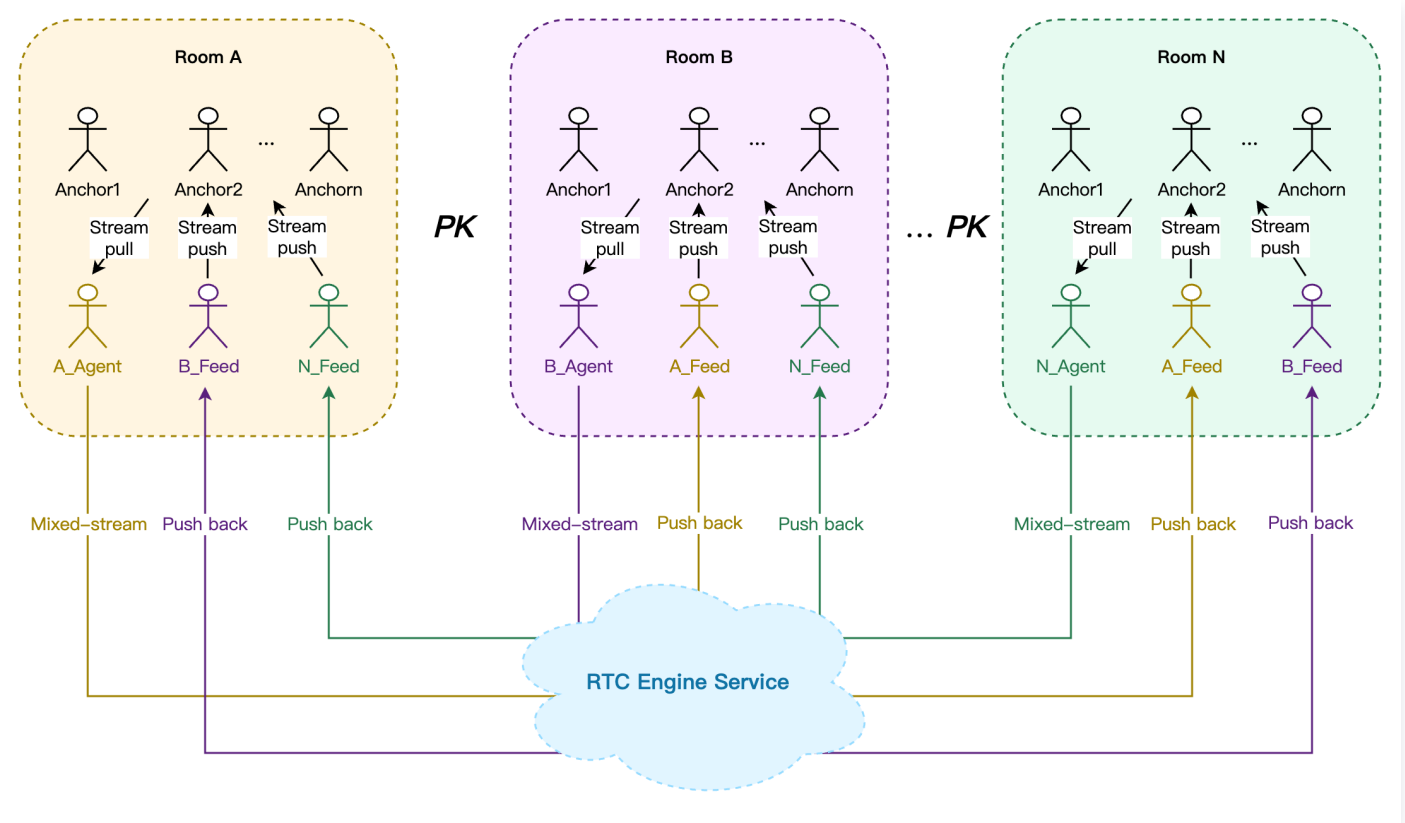
サーバー側クロスルームPK通話ソリューション

適用シーン

複数のルームでのPK、各ルームに複数の配信者がいる純粋なサーバー側クロスルーム通話シーン。

ソリューション原理

サーバー側で複数のミックスストリーミング転送タスクを起動し、各転送タスクはAgentボットユーザーを起動して自身のReal-Time Communication Engine （RTC Engine)ルームに入室させてストリームを受信し、同時に1つ以上のFeedボットユーザーを起動して混合されたオーディオ・ビデオストリームをクロスルームPK通話に参加している他のRTC Engineルームにフィードバックします。これにより、異なるルームのユーザーは他のルームのミックスストリーミングフィードバックをサブスクライブすることで、クロスルームPK通話を実現できます。



実現フロー

リアルタイムインタラクションクロスルーム通話

1. ルームAの配信者がルームBの配信者とルームNの配信者にクロスルームPKリクエスト（業務シグナリング）を送信します。
2. ルームBの配信者とルームNの配信者がクロスルームPKリクエスト（業務シグナリング）に同意します。
3. 業務バックエンドは同時にN個のミックスストリーミングフィードバックルームタスク StartPublishCdnStream を開始します。
   - タスク1：A_Agentボットが Aルームの配信者メディアストリームを受信し、RTC Engineバックエンドでミキシングされた後、A_FeedボットによってBルームとNルームにフィードバックされます。
   - タスク2：B_Agentボットが Bルームの配信者メディアストリームを受信し、RTC Engineバックエンドでミキシングされた後、B_FeedボットによってAルームとNルームにフィードバックされます。
   - タスク N：N_Agent ボットが N ルームの配信者メディアストリームを受信し、RTC Engine バックエンドでミキシングされた後、N_Feed ボットによって A ルームと B ルームにフィードバックされます。
4. ルームA、ルームB、ルームNのユーザーがルーム内のミックスストリームフィードバックされたオーディオ・ビデオストリームを互いに受信し、クロスルームPKを開始します。
5. クロスルームPKが終了し、業務バックエンドはTaskIdを通じてN個のミックスストリームフィードバックルームタスク StopPublishCdnStream を停止します。

> ⚠ 注意：

- 本ソリューションは最大11ルームでの同時クロスルームPK通話をサポートし、各ルームは最大16人の配信者が同時に通話に参加できます。
- ボットIDはルーム内の通常ユーザーIDと競合してはいけません。そうでないと、ボットユーザーがRTC Engineルームから退出させられるため、転送タスクが異常終了する可能性があります。

サンプルコード

以下は、純粋なオーディオシーンを例に、クロスルームPKミックスストリーミングフィードバックルームタスクのパラメータ例を示します。

タスク1

```
{
  "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..."
    }
  ]
}
```

タスク2

```
{
  "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..."
```

```
    }
  ]
}
```

タスク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：**
> 純粋なオーディオシーンでは、RTC Engineバックエンドはルーム内のすべての配信者のオーディオストリームをデフォルトでミックスします。また、オーディオパラメータ McuAudioParams を使用して、オーディオミックスストリーミングのブラックリスト/ホワイトリストを指定することもできます。

クロスルーム PK 通話ソリューションの比較分析

上記では、3つの異なるクロスルームPK通話の実現ソリューションを紹介しました。それぞれに異なる適用シーンがあります。以下では、4つの次元で異なるクロスルーム通話ソリューションを比較分析します。

ソリューションタイプ	ソリューション利点	ソリューション欠点	ルーム及び人数制限	おすすめの使用シーン
通常のクロスルーム PK 通話ソリューション	2人PKの呼び出しロジックは簡単です	複数人PKの呼び出しロジックは複雑です	単一配信者は他のルームの最大9人の配信者とクロスルームPK可能	2つのルーム、単一配信者（2人）のクロスルームPK
サーバー側クロスルームPK通話ソリューション	純粋なサーバー側ソリューションであり、クライアント側で追加処理不要	追加のボットの配信・受信およびミックスストリーミング費用が発生します	最大11ルームでの同時クロスルームPKをサポートし、各ルームは最大16人の配信者が同時にクロスルームPKに参加できます	複数のルーム、複数の配信者（複数人）によるクロスルームPK、純粋なサーバー側管理

AI 対話 IM シグナリング ソリューション

最終更新日：：2025-11-18 15:15:34

IM SDK の統合

iOS

IM SDK を統合

IM SDK の統合には、CocoaPodsを使用した自動ロード方式を選択することをお勧めします。

1. CocoaPodsをインストールします。ターミナルウィンドウで以下のコマンドを入力します（MacにRuby環境を事前にインストールする必要があります）。

```
sudo gem install cocoapods
```

2. Podfileファイルを作成します。プロジェクトのパスに移動し、次のコマンドラインを入力すると、プロジェクトパスにPodfileファイルが作成されます。

```
pod init
```

3. Podfileファイルを編集します。以下の方法でPodfileファイルを設定してください。

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

target 'App' do
    # 完全版のIM SDKを統合する場合、バージョン番号は'8.1.6129'以上にする必要があります
    pod 'TXIMSDK_Plus_iOS','8.1.6129'
    # または、ボリュームを削減したIM SDK（AIシグナリング関連機能のみを含む）を統合する場合、バージョン番号は'8.2.6361'以上にする必要があります
    pod 'TXIMSDK_Plus_SignalingSDK','8.2.6361'
end
```

4. SDKを更新してインストールします。

ターミナルウィンドウで以下のコマンドを入力して、ローカルライブラリファイルを更新し、IM SDKをインストールします。

```
pod install
```

または、以下のコマンドを使用してローカルライブラリのバージョンを更新します。

```
pod update
```

> ⚠ 注意：
> 上記の操作を行っても問題が解決しない場合は、Xcode 統合のよくある問題 ドキュメントをご参照ください。

IM SDK を引用

プロジェクトコードでSDKを使用するには2つの方法があります。

- `Xcode > Build Setting > Header Search Paths` でSDKヘッダーファイルのパスを設定し、プロジェクトでSDK APIを使用する必要があるファイルに具体的なヘッダーファイルを導入します。

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

- プロジェクトでSDK APIを使用する必要があるファイルに具体的なヘッダーファイルを導入します。

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

Android

SDK（aar）を統合

IM SDK の統合には、Gradle を使用した自動ロード方式を選択することをお勧めします。

1. SDK依存関係を追加。
   1.1 appのbuild.gradleを見つけて、repositoriesにmavenCentral()の依存関係を追加します。

```
repositories {
    google()
    // mavenCentral リポジトリを追加
    mavenCentral()
}
```

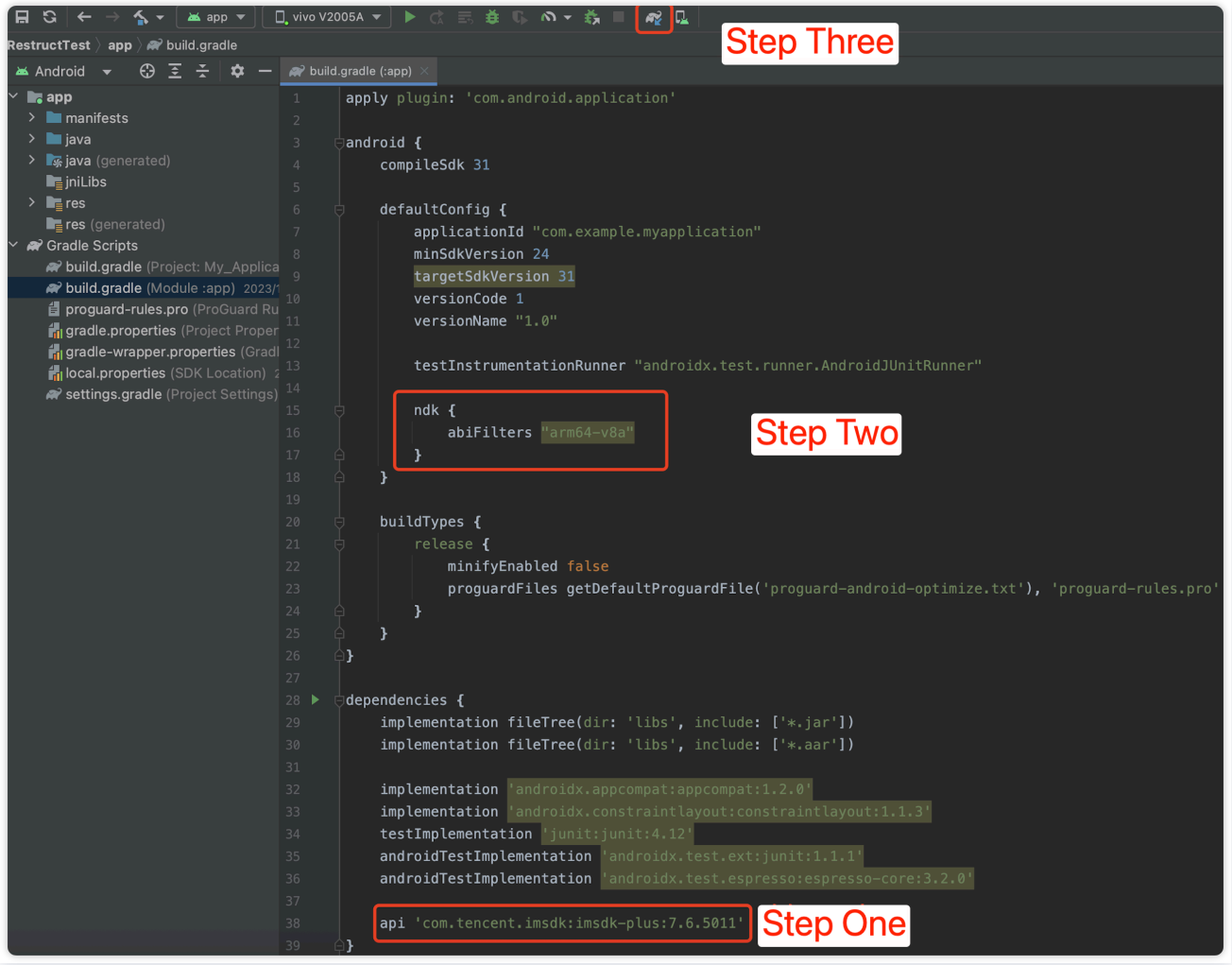
1.2 次に、dependenciesにIM SDKの依存関係を追加します。

```
dependencies {
    // 完全版のIM SDKを統合する場合、バージョン番号は'8.1.6129'以上にする必要があります
    api 'com.tencent.imsdk:imsdk-plus:8.1.6129'

    // または、ボリュームを削減したIM SDK（AIシグナリング関連機能のみを含む）を統合する場合、バージョン番号は'8.2.6361'以上にする必要があります
    api 'com.tencent.imsdk:signalingsdk:8.2.6361'
}
```

2. Appで使用するアーキテクチャを指定します。defaultConfigで、Appで使用するCPUアーキテクチャを指定します（IM SDK 4.3.118バージョン以降から、armeabi-v7a、arm64-v8a、x86、x86_64をサポート）。

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

3. SDKの同期。ネットワークがmavenに接続されていることを確認し、Sync ボタンをクリックすると、SDKが自動的にダウンロードされてプロジェクトに統合されます。



App 権限の設定

AndroidManifest.xmlで App の権限を設定します。IM SDK には以下の権限が必要です。

```
    <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" />
```

難読化ルールの設定

proguard-rules.proファイルで、IM SDK関連のクラスを難読化除外リストに追加します。

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

Web & ミニプログラム

IM SDK 統合

npm 方式で IM SDK を Web やミニプログラムに統合することをお勧めします。

```
// バージョン番号v3.4.5以上
npm install @tencentcloud/chat
```

説明:
依存関係の同期中に問題が発生した場合は、npmのソースを切り替えてから再度お試しください。

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

モジュールを導入

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

SDKを初期化

iOS

1. 初期化インターフェースを呼び出します。

```
// 1. Chatコンソールからアプリケーションの SDKAppID を取得します。
// 2. configオブジェクトを初期化します
V2TIMSDKConfig *config = [[V2TIMSDKConfig alloc] init];
// 3. log出力レベルを指定します。
config.logLevel = V2TIM_LOG_INFO;
// 4. V2TIMSDKListenerのイベントリスナーを追加します。selfは
id<V2TIMSDKListener>の実装クラスです。IM SDKのイベントを監視する必要がない場
合、このステップを省略できます。
[[V2TIMManager sharedInstance] addIMSDKListener:self];
// 5. IM SDKを初期化し、このインターフェースを呼び出した後、すぐにログインイン
ターフェースを呼び出すことができます。
```

```
[[V2TIMManager sharedInstance] initSDK:sdkAppID config:config];
```

2. ログイン。

```
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) {
    // 以下のエラーコードが返された場合、UserSigが期限切れであることを示します。新しく発行された UserSig を使用して再度ログインしてください。
    // 1. ERR_USER_SIG_EXPIRED (6206)
    // 2. ERR_SVR_ACCOUNT_USERSIG_EXPIRED (70001)
    // 注意：他のエラーコードの場合は、ここでログインインターフェースを呼び出さないでください。IM SDK のログインが無限ループになるのを避けるためです。
    NSLog(@"failure, code:%d, desc:%@", code, desc);
}];
```

Android

1. 初期化インターフェースを呼び出します。

```
// 1. Chatコンソールからアプリケーションの SDKAppIDを取得します。
// 2. configオブジェクトを初期化します。
V2TIMSDKConfig config = new V2TIMSDKConfig();
// 3. log出力レベルを指定します。
config.setLogLevel(V2TIMSDKConfig.V2TIM_LOG_INFO);
// 4. V2TIMSDKListenerのイベントリスナーを追加します。sdkListenerは V2TIMSDKListenerの実現クラスです。IM SDKのイベントを監視する必要がない場合、このステップを省略できます。
V2TIMManager.getInstance().addIMSDKListener(sdkListener);
// 5. IM SDKを初期化し、このインターフェースを呼び出した後、すぐにログインインターフェースを呼び出すことができます。
V2TIMManager.getInstance().initSDK(context, sdkAppID, config);
```

2. ログイン。

```
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) {
        // 以下のエラーコードが返された場合、UserSigが期限切れであることを示し
ます。新しく発行された UserSig を使用して再度ログインしてください。
        // 1. ERR_USER_SIG_EXPIRED (6206)
        // 2. ERR_SVR_ACCOUNT_USERSIG_EXPIRED (70001)
        // 注意: 他のエラーコードの場合は、ここでログインインターフェースを呼び
出さないでください。IM SDK のログインが無限ループになるのを避けるためです。
        Log.i("imsdk", "failure, code:" + code + ", desc:" + desc);
    }
});
```

Web & ミニプログラム

1. 初期化インターフェースを呼び出します。

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

let options = {
  SDKAppID: 0 // 導入時には0をChatアプリケーションのSDKAppIDに置き換えてくだ
さい
};
// SDKインスタンスを作成します。`TencentCloudChat.create()`メソッドは同じ
`SDKAppID`に対して同じインスタンスのみを返します
let chat = TencentCloudChat.create(options); // SDKインスタンスは通常
chatで表されます
```

```
chat.setLogLevel(0); // 通常レベル、ログ量が多く、導入時にはこの設定をお勧めします
// chat.setLogLevel(1); // release レベル、SDKは重要な情報を出力し、本番環境でこの設定をお勧めします
```

2. ログイン。

```
let promise = chat.login({userID: 'your userID', userSig: 'your userSig'});
promise.then(function(imResponse) {
  console.log(imResponse.data); // ログイン成功
  if (imResponse.data.repeatLogin === true) {
    // アカウントがすでにログイン済みであり、今回のログイン操作は重複ログインです。
    console.log(imResponse.data.errorInfo);
  }
}).catch(function(imError) {
  console.warn('login error:', imError); // ログイン失敗に関する情報
});
```

⚠ 注意：

- Real-Time Communication Engine（RTC Engine） と Chat の `sdkAppId` と `secretKey` は同じでなければなりません。
- メッセージを受信する受信者のIMはログイン成功、即ちオンライン状態である必要があります。
- 受信者のRTC EngineアカウントとChatアカウントは、同じ `userId` でなければなりません（即ち、同じ `userId` を使用してRTC Engineルームに入室し、IMにログインする必要があります）。

サーバー側からのダウンリンクメッセージを受信します

IM SDK の1対1チャットのカスタムメッセージ受信機能（iOS & Android / Web & ミニプログラム）を通じて、クライアント側でコールバックを監視することで、リアルタイム字幕とAI状態のデータを受信します。

type	说明
10000	リアルタイム字幕、翻訳の配信
10001	AI対話リアルタイム状態の配信

10010	大規模モデルメッセージのパススルー

リアルタイム字幕を受信

```
{
  "type": 10000, // 10000はリアルタイム字幕の配信を示します
  "sender": "user_a", // 発言者のuserid
  "receiver": [], // 受信者useridリスト、このメッセージは実際にはルーム内でブロードキャストされます
  "payload": {
     "text":"", // 音声認識されたテキスト
     "translation_text":"", // 翻訳されたテキスト
     "start_time":"00:00:01", // この文の開始時間
     "end_time":"00:00:02", // この文の終了時間
     "roundid": "xxxxx" // 一連の対話を一意に識別します
     "end": true // trueの場合、これは完全な文であることを示します
  }
}
```

ボットの状態を受信

```
{
  "type": 10001, // ボットの状態
  "sender": "user_a", // 送信者userid、ここはボットのidです
  "receiver": [], // 受信者useridリスト、このメッセージは実際にはルーム内でブロードキャストされます
  "payload": {
    "roundid": "xxx", // 一連の対話を一意に識別します
    "timestamp": 123
    "state": 1,       //   1 聴取中 2 思考中 3 発言中 4 中断された
  }
}
```

大規模モデルメッセージのパススルー受信

```
{
  "type": 10010, // 大規模モデルメッセージのパススルー・ダウンリング
  "sender": "user_a", // 送信者userid、ここはボットのidです
```

```
  "receiver": [], // 受信者useridリスト、このメッセージは実際にはルーム内でブロードキャストされます
  "payload": {
    "id": "uuid", // メッセージID、uuidを使用可能、問題の調査に使用 オプション
    "taskid":"xxxxxx", // このai対話のtaskid、必須
    "timestamp": 123 // タイムスタンプ、問題の調査に使用、オプション
        "data": {
            "key": "value" //業務でカスタマイズされたjson形式
        }
  }
}
```

サンプルコード

iOS

```
// addSimpleMsgListenerを呼び出してイベントリスナーを設定します
V2TIMManager.sharedInstance().addSimpleMsgListener(listener: self)

/// 1対1チャットのカスタムメッセージを受信します
/// @param msgID メッセージID
/// @param info 送信者情報
/// @param data カスタムメッセージのバイナリ内容
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

```
// addSimpleMsgListenerを呼び出してイベントリスナーを設定します
V2TIMManager.getInstance().addSimpleMsgListener(sdkListener);

/**
 * 1対1チャットのカスタムメッセージを受信します
 * @param msgID メッセージID
 * @param sender 送信者情報
 * @param customData 送信内容
 */
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 & ミニプログラム

```
const onMessageReceived = (event) => {
  const messageList = event.data;
  messageList?.forEach((msg) => {
    if (msg.type === TencentCloudChat.TYPES.MSG_CUSTOM) {
      console.log('カスタムメッセージを受信しました', 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('字幕メッセージ', jsonData);
          return;
        }
        if (jsonData.type === 10001) {
          console.log('ボットの状態', jsonData);
          return;
        }
        if (jsonData.type === 10010) {
          console.log('大規模モデルメッセージのパススルー・ダウンリンク',
jsonData);
          return;
        }
      } catch (error) {
        console.error('receive custom msg', data, error);
      }
    }
  });
}

// メッセージを監視
chat.on(TencentCloudChat.EVENT.MESSAGE_RECEIVED, onMessageReceived);
```

説明:
デフォルトでは、1対1チャットのカスタムメッセージでリアルタイム字幕とAI状態のデータを受信します。1対1チャットが要件を満たさない場合、グループチャットのカスタムメッセージチャネルを開通する必要があります。その場合は当社までお問い合わせください。

端末からアップリンクシグナリングを送信

カスタムシグナリングを送信することで、ASRプロセスをスキップし、直接AIとテキストコミュニケーションを行ったり、中断シグナリングを送信して中断を行ったり、大規模モデルにパススルー情報を直接送信したりできます。

type	说明
20000	ai_conversation_chat：AI対話テキストを送信

20001	ai_conversation_interrupt：手動で中断
20010	大規模モデルにパススルー情報を送信

アップリンクシグナリングを送信し、ASRプロセスをスキップして直接AIとテキストコミュニケーションを行います

```
{
  "type": 20000,
  "sender": "user_a", // 送信者useridであり、サーバー側はこのuseridが有効かどうかをチェックします
  "receiver": ["user_bot"], // 受信者 userid リストであり、ボットのuseridを記入するだけで十分です。サーバー側でuseridが有効かどうかをチェックします
  "payload": {
    "id": "uuid", // メッセージidであり、uuidを使用可能、問題の調査に使用
    "message": "xxx", // メッセージ内容
    "timestamp": 123, // タイムスタンプ、問題の調査に使用
    "taskid": "v2_20240920_xxxxx",
  }
}
```

中断信号を送信して中断します

```
{
  "type": 20001,
  "sender": "userid", // 送信者のuseridであり、サーバー側はこのuseridが有効かどうかをチェックします
  "receiver": ["user_bot"], // 受信者 userid リストであり、ボットのuseridを記入するだけで十分です
  "payload": {
    "id": "uuid", // メッセージidであり、uuidを使用可能、問題の調査に使用
    "timestamp": 123 // タイムスタンプ、問題の調査に使用
    "taskid": "v2_20240920_xxxxx",
  }
}
```

大規模モデルにパススルー情報を送信

```
{
    "type": 20010,
    "sender": "userid",
    "receiver": [
        "robotid"
    ],
    "payload": {
        "id": "uuid",
        "taskid": "v2_20240920_xxxxx",
        "timestamp": 1234,
        "data": {
            "key": "value" //業務でカスタマイズされたjson形式
        }
    }
}
```

サンプルコード

iOS

```
@IBAction func interruptAi(_ sender: UIButton) {
    let timestamp = Int(Date().timeIntervalSince1970 * 1000)
    let payload = [
        "id": userId + "_\(roomId)" + "_\(timestamp)", // メッセージidであり、uuidを使用可能、問題の調査に使用
        "timestamp": timestamp, // タイムスタンプ、問題の調査に使用
        "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); // メッ
セージid、uuidを使用可能、問題の調査に使用
    payload.put("timestamp", timestamp); // タイムスタンプ、問題の調査に使用
    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 & ミニプログラム

```
// 中断シグナルを送信
chat.callExperimentalAPI('sendTRTCCustomData', {
serviceCommand: 'trtc_ai_service.SendCustomCmdMsg',
data: {
  type: 20001,
  sender: "user_a", // 送信者useridであり、サーバー側はこのuseridが有効かどうか
をチェックします
  receiver: ["user_bot"], // 受信者 userid リストであり、ボットのuseridを記入
するだけで十分です
  payload: {
    id: "uuid", // メッセージid、uuidを使用可能、問題の調査に使用
    timestamp: 123, // タイムスタンプ、問題の調査に使用
    taskid: "タスクのtaskid"
```

```
    }
}
});
```

注意：

`type` 、 `sender` 、 `receiver` および `payload` の `taskid` 、 `id` 、 `timestamp` は必須フィールドです。