物联网开发平台

视频通话

腾讯云

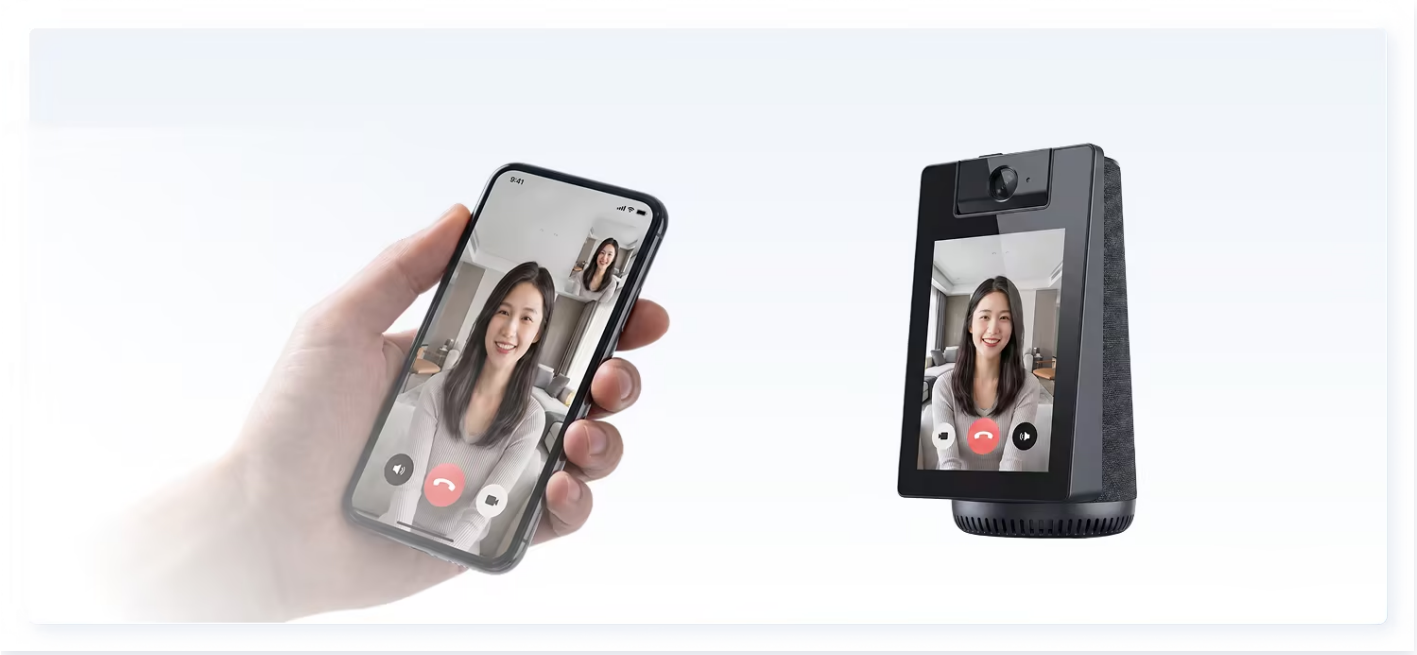
文档目录

视频通话
功能介绍

最近更新时间：2026-07-30 12:00:36

功能概述

视频通话服务是腾讯云物联网基于腾讯实时音视频能力，结合腾讯超过20年在网络与音视频技术上的深厚积累，与腾讯云物联网深度融合，面向各场景音视频类智能硬件开发者开放。该服务助力智能硬件开发者快速搭建高可靠、低延时的语音、视频通话服务。



产品优势

计费说明

视频通话服务需集成设备 SDK，需要用户先购买消费实例下的音视频设备激活码，计费方式可参考 音视频设备接入服务 。

应用场景

应用场景	说明
安防监控	支持各类家用 WiFi 监控摄像头、户外4G摄像头、婴幼儿看护摄像头及宠物摄像头的实时监控，并实现 APP 与设备的双向通话。
智能入户	支持各类可视门铃、低功耗智能门锁、多目双向对讲门铃的实时视频、语音通话及监控。

智慧屏显	支持家庭场景各类智慧屏显设备与 APP 实时语音、视频通话。
电话手表	支持各场景电话手表的音视频通话对讲。
行车记录仪	支持车载行车记录仪的实时监控。
机器人	支持各类家庭场景机器人与 APP 实时语音、视频通话及监控。

开通服务

最近更新时间：2026-07-30 12:00:36

开通消费实例正式版

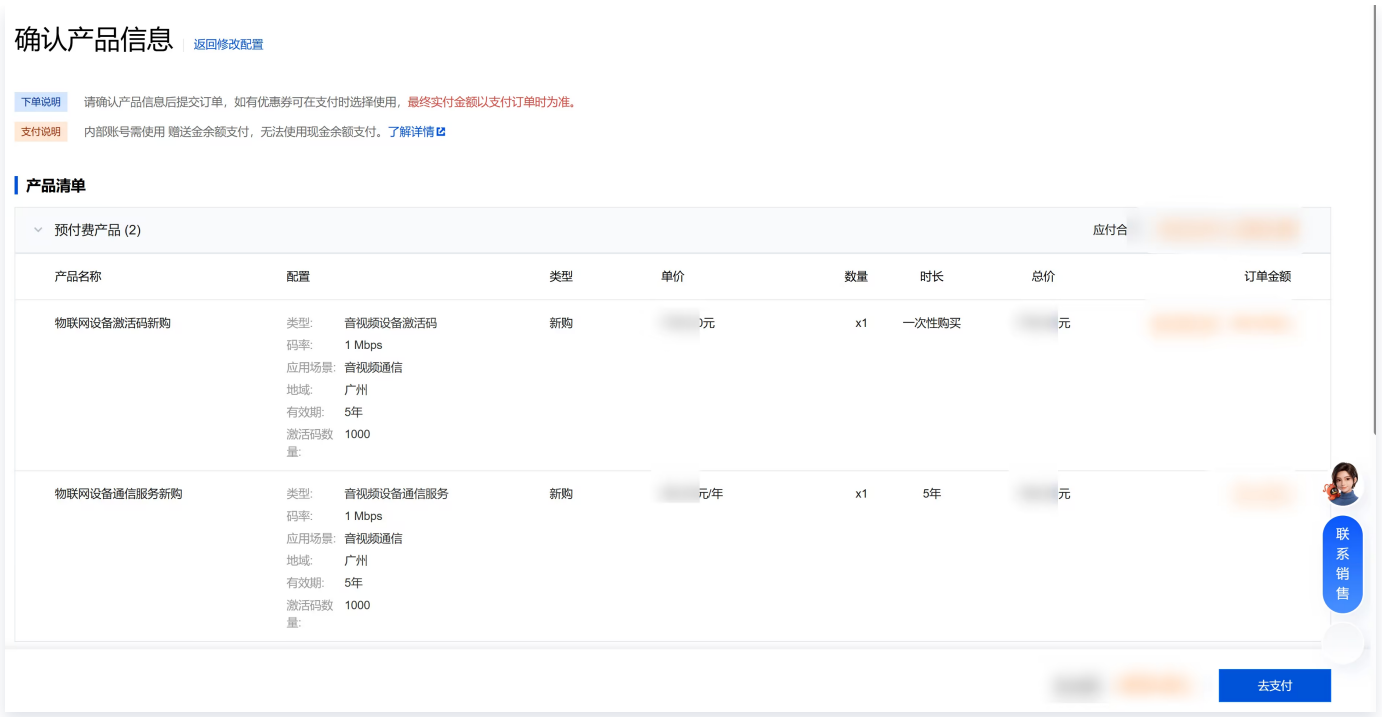
> **注意：**
> 设备接入视频通话服务涉及芯片适配工作，消费实例下的音视频设备接入服务规格目前需开白才可购买。需要该服务的用户可通过 下载中心 查看是否有已适配的芯片 SDK，若没有则可通过填写 芯片适配需求表 或者通过 提交工单 联系我们评估。视频通话服务必须购买音视频接入服务类型，若购买“基础设备接入”服务，则无法使用视频通话。

购买消费实例

音视频设备接入必须要有对应的音视频设备激活码额度，购买激活码成功后即表示开通了正式版。具体购买流程如下：

1. 在购买音视频设备接入服务之前，您需要先 注册腾讯云 账号，并完成 实名认证 。
2. 进入 物联网开发平台购买页 ，实例类型选择**消费实例**，服务类型选择**“音视频设备接入服务”**。
3. 生效地域：国内设备选择广州区域，出海设备选择新加坡；有效期默认规格5年。
4. 激活码数量：输入您要量产的设备数量，一个物理设备默认烧录一个激活码。
5. 勾选我已阅读并同意《腾讯云服务协议》，单击**立即购买**。

6. 单击立即购买后进入订单确认，音视频设备接入服务分别对应“音视频设备接入服务”和“音视频设备通信服务”。



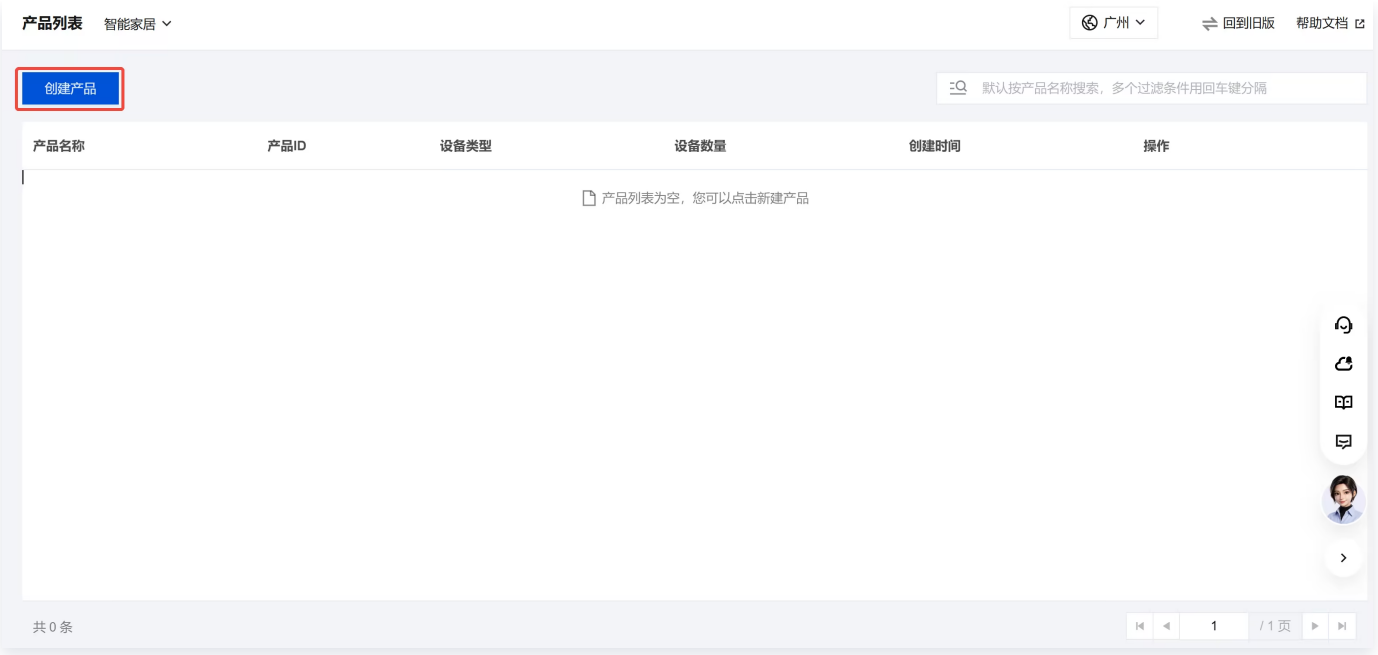
7. 确认后单击**去支付**进行付款，支付成功后系统将您支付订单的激活码数量累加到对应激活码规格的总可用数量。

查看激活码用量

支付完音视频设备接入服务后，用户可进入 物联网开发平台控制台 的**消费实例**，选择左侧导航栏的**用量统计** > **激活码概览**页面，可查看支付成功的订单对应的音视频设备激活码用量。

创建产品

1. 登录 物联网开发平台控制台 ，进入**实例管理**页面，单击**消费实例**的卡片进入详情页。
2. 选择消费实例左侧导航栏的**产品**，然后单击**创建产品**。



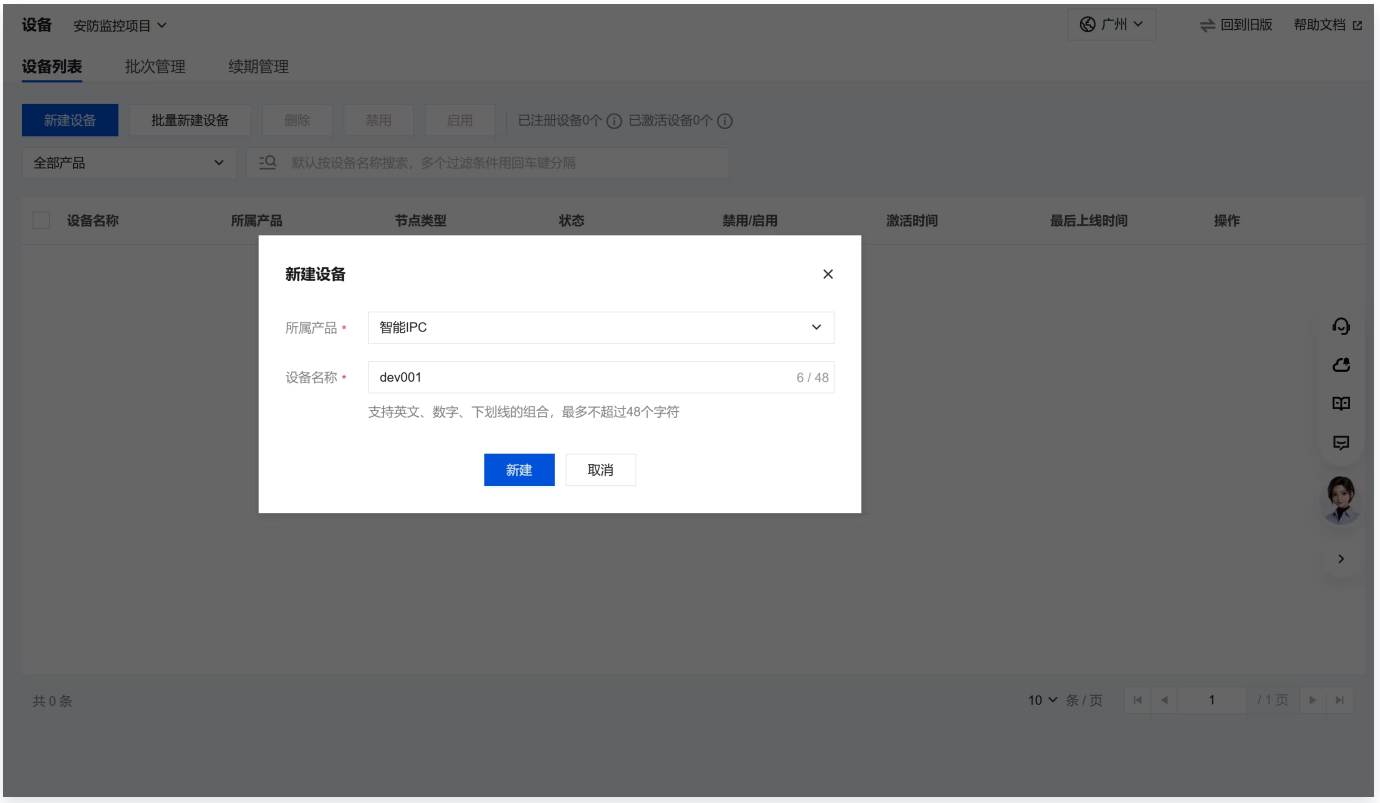
3. 在创建产品页面中，输入“产品名称”，设备类型必须是“**音视频设备**”、认证方式为“密钥认证”、数据协议选择“物模型”。信息配置好后，单击**创建**。

> **注意：**
> 创建产品时若码率规格显示为“不可选”，表示您的账号未购买该规格码率。码率决定了使用该款产品的用户切换清晰度的上限。

创建设备

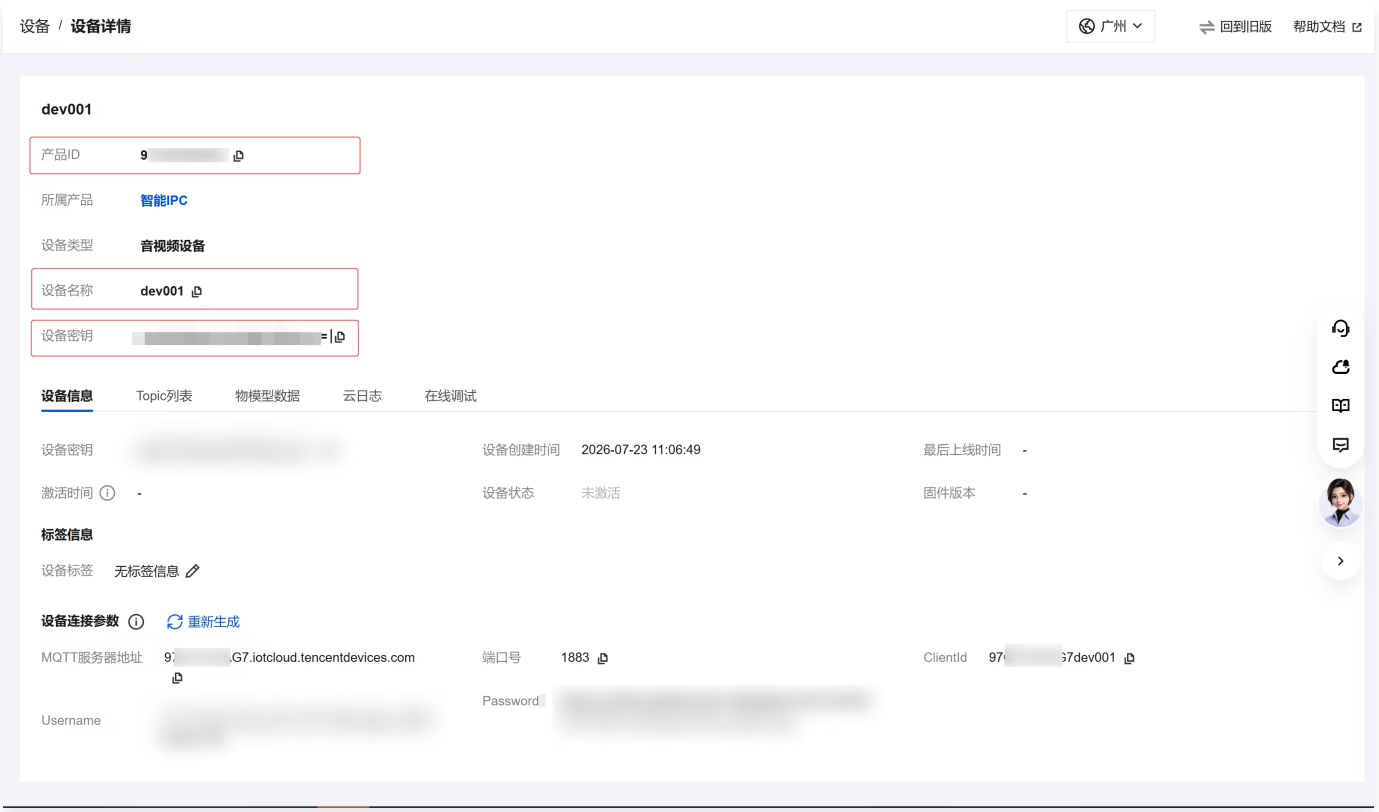
1. 选择消费实例左侧导航栏的**设备**，然后单击**新建设备**。

2. 所属产品选择前述步骤已创建成功的产品，输入设备名称并单击**新建**。



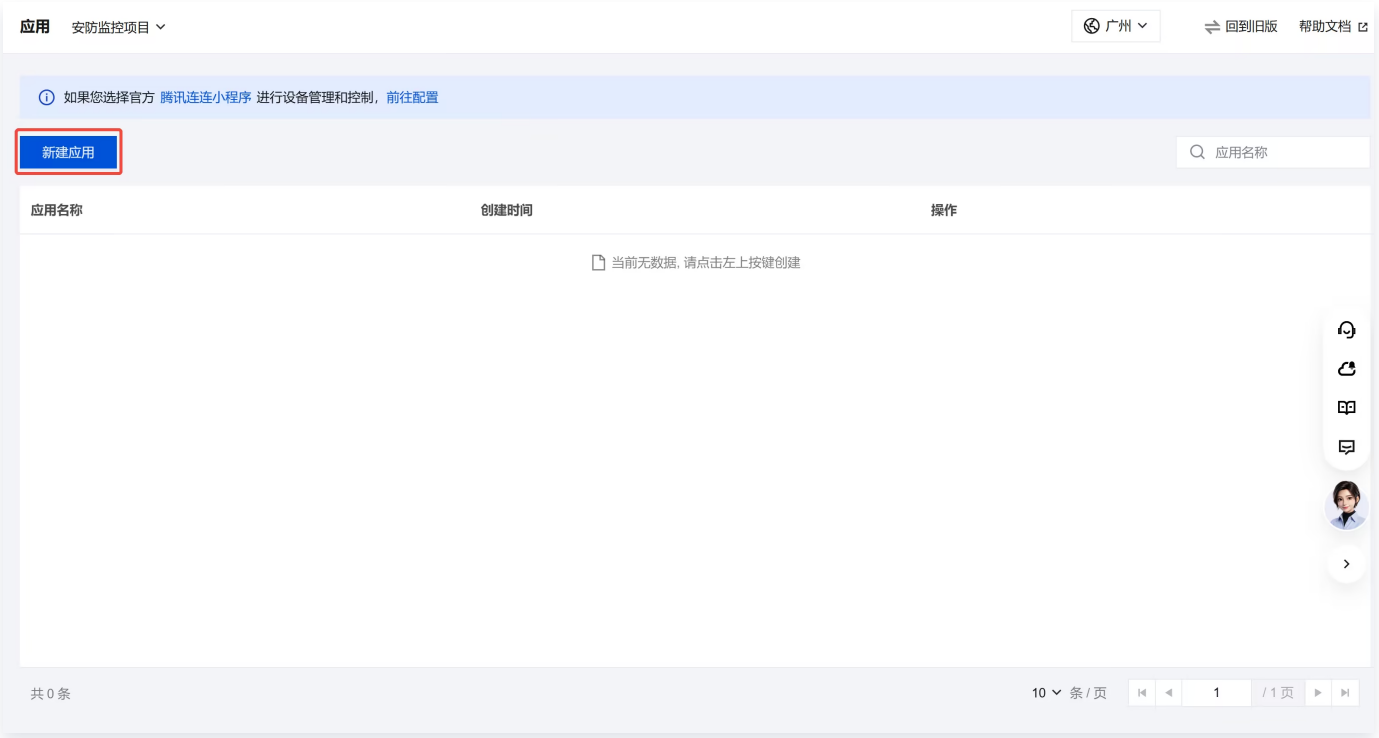
3. 单击创建成功的设备，进入设备详情页获取设备三元组信息。

> **注意：**
> 产品 ID、设备名称、设备密钥三元组信息作为后续视频通话的设备身份凭证。

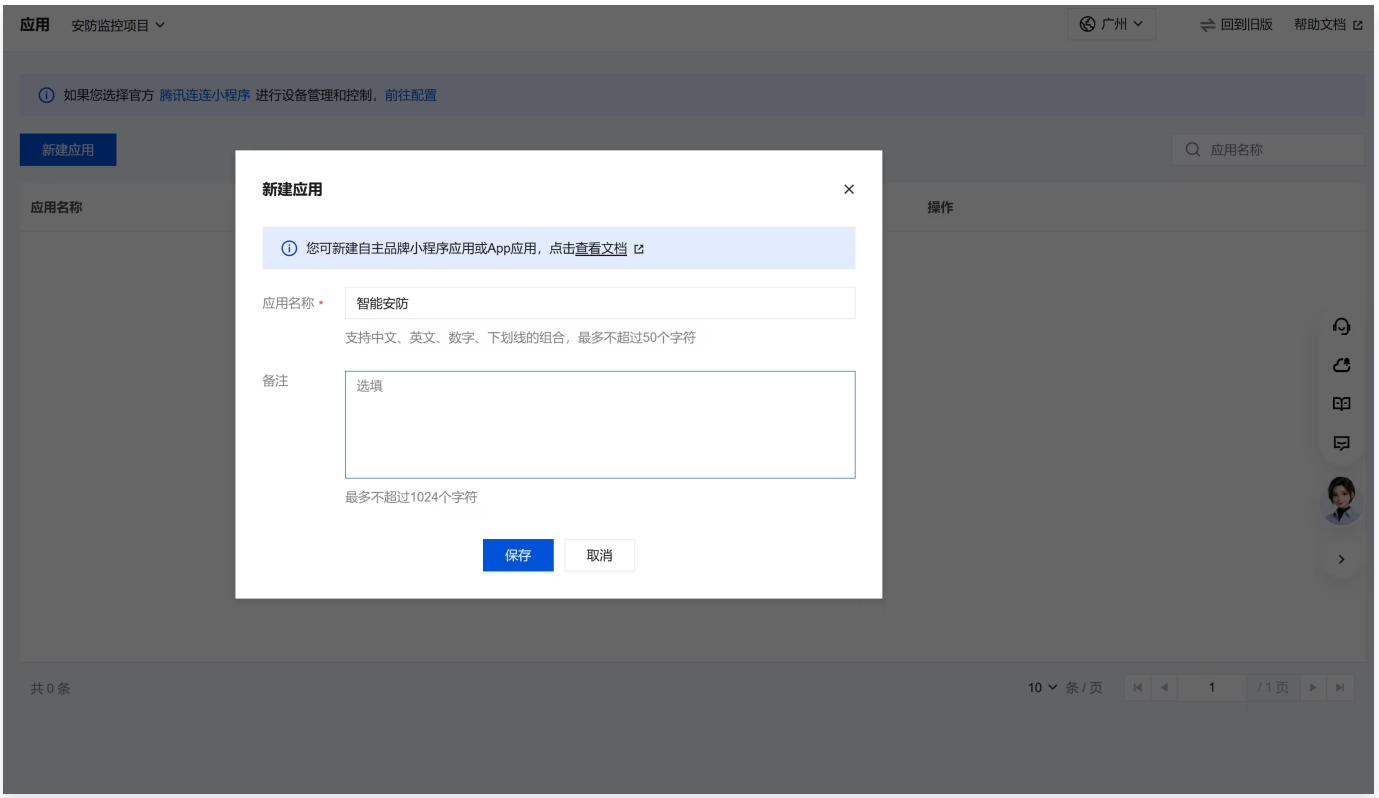


创建应用

1. 选择消费实例左侧导航栏的**应用**，单击**新建应用**。

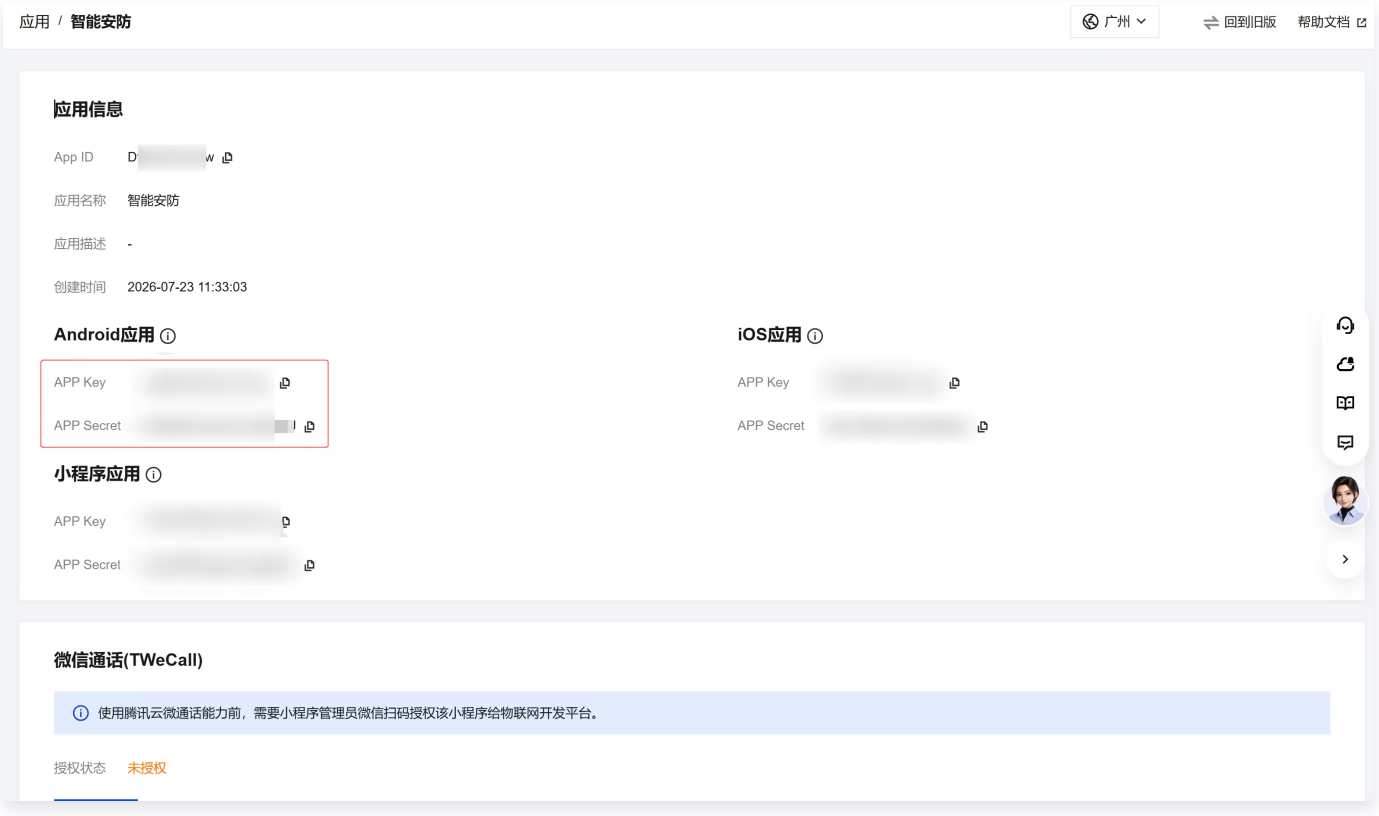


2. 输入应用名称，单击**保存**创建应用。



3. 单击创建的应用，可获取应用的 App Key 及 App Secret。

> **注意：**
> App Key 与 App Secret 会在后续 App SDK 开发时使用，可直接使用 Android 应用的 App Key 与 App Secret。



应用关联产品

创建应用后，需授权该应用管理某产品下的设备，需在应用详情页执行应用与产品关联的操作。如下图：

注意：

若应用未关联产品，APP 在绑定产品下的设备时会提示“APP对操作该产品无权限”。

应用端接入

最近更新时间：2026-07-30 12:00:38

本文介绍腾讯云物联网（IoT）应用端 SDK 中视频通话（`TXIoTCallSession`）相关 API 的使用方法。完成 快速接入 并登录 SDK 后，您可以通过本文实现设备与 App 之间的实时音视频通话。

前提条件

- 已集成 IoT 应用端 SDK 并完成初始化。
- 已调用 `TXIoTEngine.login` 登录，且登录未过期。
- 目标设备已绑定到当前登录账户名下（详见 设备管理）。未绑定的设备没有通话权限，相关接口会返回权限相关错误。

> **注意：**
> - **呼叫方向限制：**本套视频通话接口仅支持应用端主动呼叫设备端，**应用端只能发起通话（callDevice），不能接听来电。**
> - **通话优先级：**视频通话的优先级高于实时监控，若设备当前正处于实时监控拉流状态，应用端发起视频通话会打断该设备的实时监控（详见 实时监控），设备会切换到视频通话状态。

使用建议

- 发起通话前先完成设备绑定与登录，并检查设备在线状态。
- 在调用 callDevice 前注册监听，避免漏掉 `onCallBegin` 与错误回调。
- 视频通话务必设置本地预览视图与远端渲染视图，且确保视图可见。
- 通话结束后及时调用 hangup、停止远端视图并移除监听，释放摄像头与麦克风等资源。
- 摄像头/麦克风需在 `Info.plist`（iOS）或 `AndroidManifest`（Android）中声明并动态申请权限。

获取通话会话对象

视频通话通过 `TXIoTCallSession` 对象进行。该对象由 `TXIoTEngine` 管理，无需手动创建，直接通过引擎获取即可：

- Android：`TXIoTEngine.getInstance(context).getCallSession()`
- iOS：`[TXIoTEngine getInstance] getCallSession`

若返回 `null`（Android）或 `nil`（iOS），通常表示当前未登录或登录已过期，请先调用 `TXIoTEngine.login` 登录成功后再获取。建议在每次发起通话前重新获取，避免因会话失效导致调用失败。

> **警告：**
> 视频通话对象为单例，同一时刻只存在一个视频通话。若已有视频通话正在进行（已调用 callDevice 且尚未 hangup），再次调用 callDevice 会被直接忽略，不会打断或切换已有视频通话；无需也不能重复

> 获取多个会话实例。

设置通话监听

在发起通话前，请先注册通话监听器，用于接收通话开始/结束、对方状态变化、网络质量、错误等回调。通话结束或页面销毁时记得移除监听，避免内存泄漏。

方法	说明
addListener	添加通话状态与错误的监听器，回调接口定义见 通话事件回调（iOS 方法名为 `addDelegate`）。
removeListener	移除已注册的监听（iOS 方法名为 `removeDelegate`）。

Android

```
TXIoTCallSession callSession =
TXIoTEngine.getInstance(context).getCallSession();
if (callSession == null) {
    // 未登录或登录过期，请先登录。
    return;
}
callSession.addListener(new TXIoTCallListener() {
    @Override
    public void onError(TXIoTErrorCode code, String msg) {
        // 通话发生错误。
    }

    @Override
    public void onCallBegin(TXIoTCallMediaType mediaType) {
        // 通话开始。
    }

    @Override
    public void onCallEnd(TXIoTCallMediaType mediaType,
TXIoTCallEndReason reason) {
        // 通话结束。
    }
```

```
    // 其余回调省略，详见“通话状态与回调通知”。
});
```

iOS

```
TXIoTCallSession *callSession = [[TXIoTEngine getInstance]
getCallSession];
if (callSession == nil) {
    // 未登录或登录过期，请先登录。
    return;
}
[callSession addDelegate:self];
// 在 delegate 对象中实现 TXIoTCallDelegate 协议方法：
// - (void)onError:(TXIoTErrorCode)errorCode errorMessage:(NSString
*)errorMessage;
// - (void)onCallBegin:(TXIoTCallMediaType)mediaType;
// - (void)onCallEnd:(TXIoTCallMediaType)mediaType reason:
(TXIoTCallEndReason)reason;
// 其余回调省略，详见“通话状态与回调通知”。
```

⚠ 注意：
请在发起通话（callDevice）之前完成监听注册；否则可能漏掉 `onCallBegin`、`onError` 等关键回调。

发起与结束通话

调用 callDevice 向指定设备发起音视频通话，需传入目标设备的 TXIoTDeviceId 与通话类型（TXIoTCallMediaType）。通话过程中调用 hangup 结束通话。

方法	说明
callDevice	向 `deviceId`（TXIoTDeviceId）指定的设备发起通话，`callType`（TXIoTCallMediaType）为通话类型。
hangup	挂断并结束当前通话。

Android

```
// 通过 TXIoTDeviceManager.getDeviceList、
TXIoTDeviceManager.getDeviceListSharedWithMe
// 获取有权限通话的设备列表
TXIoTDeviceId deviceId = ...;
callSession.callDevice(deviceId, TXIoTCallMediaType.VIDEO);
```

iOS

```
// 通过 TXIoTDeviceManager.getDeviceList、
TXIoTDeviceManager.getDeviceListSharedWithMe
// 获取有权限通话的设备列表
TXIoTDeviceId *deviceId = ...;
[callSession callDevice:deviceId callType:TXIoTCallMediaTypeVideo];
```

调用 callDevice 向设备发起通话后，可能出现以下结果，对应不同回调：

- **设备接听本次通话 →** `onCallAccepted` **，随后媒体就绪 →** `onCallBegin` **。**
- **设备拒绝接听 →** `onCallRejected` **。**
- **设备在 30 秒内未接听 →** `onCallNoResponse` **。**
- **本地摄像头/麦克风未授权，通话无法建立 → 结束原因** `CALL_PERMISSION_DENIED` **，回调** `onCallEnd` **。**
- **远端挂断 →** `onCallEnd` **（原因对应** `REMOTE_HANGUP` **）。**
- **网络异常、设备离线等 →** `onCallEnd` **（原因对应** `NETWORK_ERROR` **等）或** `onCallUserOffline` **。**
- **若呼叫时设备已经在与另一个应用端账号在通话中，此时本账号再向同一台设备发送呼叫请求，设备侧也会收到本账号的呼叫请求。设备侧可以决定是拒绝后发的呼叫请求，还是结束正在进行的通话并接受本账号的呼叫请求。**

结束通话时，应用端调用 hangup，本机会收到 `onCallEnd` **收到结束通知（原因为** `LOCAL_HANGUP` **）。**

Android

```
callSession.hangup();
```

iOS

```
[callSession hangup];
```

设置渲染视图

视频通话需要本地预览与远端画面两类渲染视图：

- **本地预览：** `openCamera(cameraId, videoView)` **开启摄像头并指定预览视图。**
- **远端画面：** `startRemoteView(callUser, videoView)` **开始渲染指定对端的视频，** `stopRemoteView(callUser)` **停止渲染指定对端的视频。**

方法	说明
openCamera	开启并预览指定摄像头。 • `cameraId` 为枚举类型 TXIoTCamera（前置/后置摄像头）。 • `videoView` 为渲染视图（Android 为 `TXCloudVideoView`，iOS 为 `UIView`）。
startRemoteView	渲染对端画面。 • `callUser` 为 TXIoTCallUser 类型。 • `videoView` 为渲染视图（Android 为 `TXCloudVideoView`，iOS 为 `UIView`）。
stopRemoteView	停止渲染指定对端画面。 `callUser` 为 TXIoTCallUser 类型。

Android

```
// 本地预览（传入 TXCloudVideoView）
callSession.openCamera(TXIoTCamera.FRONT, localView);
```

```
// 远端渲染
callSession.startRemoteView(callUser, remoteView);
// 停止远端渲染
callSession.stopRemoteView(callUser);
```

iOS

```
// 本地预览（传入 UIView）
[callSession openCamera:TXIoTCameraFront view:localView];
// 远端渲染
[callSession startRemoteView:callUser view:remoteView];
// 停止远端渲染
[callSession stopRemoteView:callUser];
```

开关摄像头与切换

通过以下方法控制本地摄像头：开启、关闭、切换前后摄像头。

方法	说明
openCamera	开启并预览指定摄像头，参数 `cameraId` 为 TXIoTCamera 类型。
closeCamera	关闭本地摄像头。
switchCamera	切换前后摄像头，参数 `cameraId` 为 TXIoTCamera 类型。

Android

```
callSession.openCamera(TXIoTCamera.BACK, localView);
callSession.switchCamera(TXIoTCamera.FRONT);
callSession.closeCamera();
```

iOS

```
[callSession openCamera:TXIoTCameraBack view:localView];
[callSession switchCamera:TXIoTCameraFront];
[callSession closeCamera];
```

开关麦克风与切换

控制本地麦克风采集的开启与关闭，并选择通话音频的播放设备（听筒或扬声器）。

方法	说明
openMicrophone	开启本地麦克风采集、传输。
closeMicrophone	关闭本地麦克风采集、传输。
selectAudioPlaybackDevice	选择通话音频的播放设备，`device` 为 TXIoTAudioPlaybackDevice（听筒或扬声器）类型。

Android

```
callSession.openMicrophone();
callSession.closeMicrophone();
callSession.selectAudioPlaybackDevice(TXIoTAudioPlaybackDevice.SPEAKER_PHONE);
```

iOS

```
[callSession openMicrophone];
[callSession closeMicrophone];
```

```
[callSession
selectAudioPlaybackDevice:TXIoTAudioPlaybackDeviceSpeakerphone];
```

通话状态与回调通知

通过 `TXIoTCallListener`（iOS 为 `TXIoTCallDelegate`）监听通话全生命周期事件。注册方式见 设置通话监听。下方表格列出各回调的含义与触发时机：

回调	触发时机 / 说明
`onError`	通话发生错误时回调错误码与信息（如摄像头/麦克风权限被拒、设备离线、网络异常等）。
`onCallBegin`	通话建立。
`onCallAccepted`	设备接听本次通话。
`onCallEnd`	通话结束。参数中带结束原因（如 LOCAL_HANGUP、REMOTE_HANGUP、CALL_PERMISSION_DENIED、NETWORK_ERROR 等）。
`onCallRejected`	设备拒绝接听本次通话。
`onCallNoResponse`	设备在 30 秒内未接听，通话超时。
`onCallUserOffline`	对端设备通话过程中离线。
`onCallUserAudioAvailable`	对方音频可用状态变化。
`onCallUserVideoAvailable`	对方视频可用状态变化。
`onCallNetworkQualityChanged`	本地与远端网络质量变化。

错误处理

所有异步通话操作通过监听器 `onError` 返回错误，下面是通话场景可能回调的错误码表，完整错误码定义见 TXIoTErrorCode。

Android

错误码	说明	建议处理方式
`ERR_SUCCESS`	成功。	无需处理（调用成功）。
`ERR_FAILED`	通用失败。	结合 `onError` 返回的 `errMsg` 排查具体失败原因。
`ERR_INVALID_PARAMETER`	参数不合法。	检查 `deviceId`、`callType` 等参数是否为空或格式错误。
`ERR_INVALID_ACCESS_TOKEN`	登录凭证无效。	重新登录 SDK 后再发起通话。
`ERR_RATE_LIMITED`	请求被限频。	降低调用频率后重试。
`ERR_UNAUTHORIZED_OPERATION`	当前用户无权限（常见于设备未绑定）。	确认目标设备已绑定到当前账户（见 前提条件）。
`ERR_DEVICE_NOT_EXIST`	设备不存在。	确认 `deviceId` 是否正确、设备是否已绑定。
`ERR_DEVICE_OFFLINE`	设备离线。	确认设备在线后重试。
`ERR_CAMERA_START_FAIL`	摄像头启动失败。	检查摄像头是否被占用或硬件异常。
`ERR_CAMERA_NOT_AUTHORIZED`	摄像头无权限。	在工程中申请并获取摄像头权限。
`ERR_CAMERA_OCCUPY`	摄像头被占用。	停止其他占用摄像头的功能后再试。
`ERR_MIC_START_FAIL`	麦克风启动失败。	检查麦克风是否被占用或硬件异常。
`ERR_MIC_NOT_AUTHORIZED`	麦克风无权限。	在工程中申请并获取麦克风权限。
`ERR_MIC_OCCUPY`	麦克风被占用。	停止其他占用麦克风的功能后再试。
`ERR_SPEAKER_START_FAIL`	扬声器启动失败。	检查音频输出设备是否正常。

iOS

错误码	数值	说明	建议处理方式
`TXIoTErrorCodeSuccess`	0	成功	无需处理（调用成功）。
`TXIoTErrorCodeFailed`	-1	通用失败	结合 `onError` 返回的 `errMsg` 排查具体失败原因。
`TXIoTErrorCodeInvalidParameter`	-2	参数不合法	检查 `deviceId`、`callType` 等参数是否为空或格式错误。
`TXIoTErrorCodeInvalidAccessToken`	-3	登录凭证无效	重新登录 SDK 后再发起通话。
`TXIoTErrorCodeRateLimited`	-4	请求被限频	降低调用频率后重试。
`TXIoTErrorCodeUnauthorizedOperation`	-5	当前用户无权限（常见于设备未绑定）	确认目标设备已绑定到当前账户（见前提条件）。
`TXIoTErrorCodeDeviceNotExist`	-1009	设备不存在	确认 `deviceId` 是否正确、设备是否已绑定。
`TXIoTErrorCodeDeviceOffline`	-1010	设备离线	确认设备在线后重试。
`TXIoTErrorCodeCameraStartFail`	-2000	摄像头启动失败	检查摄像头是否被占用或硬件异常。
`TXIoTErrorCodeCameraNotAuthorized`	-2001	摄像头无权限	在工程中申请并获取摄像头权限。

`TXIoTErrorCodeCameraOccupy`	-2002	摄像头被占用	释放其他占用摄像头的功能后再试。
`TXIoTErrorCodeMicStartFail`	-2003	麦克风启动失败	检查麦克风是否被占用或硬件异常。
`TXIoTErrorCodeMicNotAuthorized`	-2004	麦克风无权限	在工程中申请并获取麦克风权限。
`TXIoTErrorCodeMicOccupy`	-2005	麦克风被占用	释放其他占用麦克风的功能后再试。
`TXIoTErrorCodeSpeakerStartFail`	-2006	扬声器启动失败	检查音频输出设备是否正常。

设备端接入

最近更新时间：2026-07-30 12:00:37

本文将介绍 App 呼叫设备，实现 App 和设备之间的**双向通话**，包括响应来电（接听/拒绝）和挂断通话。

前提条件

在接入通话功能前，请您先完成 快速接入 文档中跑通 `tc_iot_login` 登录功能。如果您已经完成，可以跳过该操作。

接入步骤

步骤 1：注册通话事件回调

登录成功后，调用 `tc_iot_av_init` 注册 `tc_iot_av_observer_s`，通话相关的回调如下：

回调	说明
`on_call_requested`	收到对端来电时触发，携带 `contact`（对端信息）与 `option`（通话媒体类型等）。
`on_call_timeout`	来电无人应答超时时触发。
`on_call_hangup`	对端挂断通话时触发。

```
#include "tc_iot_av.h"

static void on_call_requested(const tc_iot_contact_s *contact,
tc_iot_call_option_t option);
static void on_call_timeout(const tc_iot_contact_s *contact);
static void on_call_hangup(const tc_iot_contact_s *contact);

tc_iot_av_observer_s observer;
memset(&observer, 0, sizeof(observer));
observer.on_call_requested = on_call_requested;
observer.on_call_timeout = on_call_timeout;
observer.on_call_hangup = on_call_hangup;

tc_iot_error_e av_rc = tc_iot_av_init(&observer);
if (av_rc != TC_IOT_ERR_SUCCESS) {
```

```
    printf("tc_iot_av_init failed: %d\n", av_rc);
}
```

步骤 2：响应来电，接听或拒绝

收到 `on_call_requested` 后，根据业务逻辑调用 `tc_iot_av_accept` 接听，或调用 `tc_iot_av_reject` 拒绝：

```
static void on_call_requested(const tc_iot_contact_s *contact,
tc_iot_call_option_t option) {
    printf("[demo] on_call_requested user_id=%s media=%d\n", contact->user_id, option.media_content);

    // 示例：直接接听，实际业务可结合按键、APP 联动等方式决定接听或拒绝
    tc_iot_error_e rc = tc_iot_av_accept(contact, NULL);
    if (rc != TC_IOT_ERR_SUCCESS) {
        printf("[demo] tc_iot_av_accept failed: %d\n", rc);
    }

    // 拒绝来电：tc_iot_av_reject(contact, NULL);
}
```

步骤 3：创建发送通道

己方接听来电成功后，通话即建立，此时根据通话的 `media_content` 创建对应的发送通道：

接口	说明
`tc_iot_create_video_channel(channel_id, quality)`	创建视频发送通道，通话场景固定使用通道 `0`，`quality` 为视频清晰度。
`tc_iot_create_audio_channel()`	创建音频发送通道，无需传入 `channel_id`。

> ⚠ **注意：**
> 设备通常只有一路麦克风采集源，`tc_iot_create_audio_channel` 建议全局只创建一次。

```
static tc_iot_av_channel_t *g_video_channel = NULL;
static tc_iot_av_channel_t *g_audio_channel = NULL;

// 通话建立后，根据媒体类型创建对应的发送通道
```

```
static void create_call_channels(tc_iot_media_content_e media_content,
                                 tc_iot_video_quality_e quality) {
    if (media_content == TC_IOT_MEDIA_CONTENT_VIDEO ||
        media_content == TC_IOT_MEDIA_CONTENT_AUDIO_VIDEO) {
        g_video_channel = tc_iot_create_video_channel(0, quality);
        if (!g_video_channel) {
            printf("[demo] create video channel failed\n");
        }
    }

    if ((media_content == TC_IOT_MEDIA_CONTENT_AUDIO ||
         media_content == TC_IOT_MEDIA_CONTENT_AUDIO_VIDEO) &&
        !g_audio_channel) {
        g_audio_channel = tc_iot_create_audio_channel();
        if (!g_audio_channel) {
            printf("[demo] create audio channel failed\n");
        }
    }

    // 启动本地采集（示意，详见步骤 4/5 推流示例）
    // start_capture_and_push(...)
}
```

步骤 4：推送视频帧

采集到一帧编码后的视频数据后，填充 `tc_iot_video_frame` 并调用 `tc_iot_push_video_frame` 推送到视频发送通道：

字段	说明
`codec`	视频编码格式，`TC_IOT_VIDEO_CODEC_H264` 或 `TC_IOT_VIDEO_CODEC_H265`。
`type`	帧类型，关键帧 `TC_IOT_VIDEO_FRAME_TYPE_IDR` 或非关键帧 `TC_IOT_VIDEO_FRAME_TYPE_P`。
`rotation`	画面旋转角度，如 `TC_IOT_VIDEO_ROTATION_0`。
`width` / `height`	画面宽高（像素）。

`data` / `data_size`	编码后的视频裸流数据指针与长度。
`pts_ms`	帧时间戳（毫秒）。

说明：
建议推送第一个 IDR 关键帧之后才开始持续推流，以保证对端能正常解码；在 IDR 帧之前推送 P 帧，对端可能无法解码画面。

```
void push_video_frame_sample(tc_iot_av_channel_t *video_channel,
                             const uint8_t *encoded_data, size_t
encoded_size,
                             bool is_key_frame) {
    if (!video_channel) {
        return;
    }

    tc_iot_video_frame frame;
    memset(&frame, 0, sizeof(frame));
    frame.codec = TC_IOT_VIDEO_CODEC_H264;
    frame.type = is_key_frame ? TC_IOT_VIDEO_FRAME_TYPE_IDR :
TC_IOT_VIDEO_FRAME_TYPE_P;
    frame.rotation = TC_IOT_VIDEO_ROTATION_0;
    frame.width = 480;
    frame.height = 320;
    frame.data = (uint8_t *)encoded_data;
    frame.data_size = encoded_size;
    frame.pts_ms = current_time_ms(); // 由业务自行实现，返回当前毫秒时间戳

    tc_iot_error_e err = tc_iot_push_video_frame(video_channel, &frame);
    if (err != TC_IOT_ERR_SUCCESS) {
        printf("[demo] push video frame failed: %d\n", err);
    }
}
```

步骤 5：推送音频帧

采集到一帧编码后的音频数据后，填充 `tc_iot_audio_frame` 并调用 `tc_iot_push_audio_frame` 推送到音频发送通道：

字段	说明
`codec`	音频编码格式，支持 `TC_IOT_AUDIO_CODEC_PCM` / `AAC` / `OPUS` / `G722` 。
`sample_rate`	采样率，如 `TC_IOT_AUDIO_SAMPLE_RATE_16000` 。
`channels`	声道数， `TC_IOT_AUDIO_CHANNEL_MONO` （单声道）或 `TC_IOT_AUDIO_CHANNEL_STEREO` （双声道）。
`frame_duration_ms`	单帧时长（毫秒），如 20ms 一帧。
`data` / `data_size`	编码后的音频裸流数据指针与长度。
`pts_ms`	帧时间戳（毫秒）。

```
void push_audio_frame_sample(tc_iot_av_channel_t *audio_channel,
                             const uint8_t *pcm_data, size_t pcm_size) {
    if (!audio_channel) {
        return;
    }

    tc_iot_audio_frame frame;
    memset(&frame, 0, sizeof(frame));
    frame.codec = TC_IOT_AUDIO_CODEC_PCM;
    frame.sample_rate = TC_IOT_AUDIO_SAMPLE_RATE_16000;
    frame.channels = TC_IOT_AUDIO_CHANNEL_MONO;
    frame.frame_duration_ms = 20;
    frame.data = (uint8_t *)pcm_data;
    frame.data_size = pcm_size;
    frame.pts_ms = current_time_ms(); // 由业务自行实现，返回当前毫秒时间戳

    tc_iot_error_e err = tc_iot_push_audio_frame(audio_channel, &frame);
    if (err != TC_IOT_ERR_SUCCESS) {
        printf("[demo] push audio frame failed: %d\n", err);
    }
}
```

步骤 6：挂断通话

通话中主动挂断，调用 `tc_iot_av_hangup`；对端挂断时会触发 `on_call_hangup`，此时应停止本地推流并销毁发送通道：

```
void hangup_call(const char *user_id) {
    tc_iot_contact_s contact = {0};
    contact.user_id = user_id;

    tc_iot_error_e rc = tc_iot_av_hangup(&contact, NULL);
    if (rc != TC_IOT_ERR_SUCCESS) {
        printf("[demo] tc_iot_av_hangup failed: %d\n", rc);
    }
}

static void on_call_hangup(const tc_iot_contact_s *contact) {
    printf("[demo] on_call_hangup user_id=%s\n", contact->user_id);
    // 停止推流并销毁发送通道（同实时监控场景的 on_monitor_end 处理）
}
```

> **说明：**
> 己方来电未及时处理时会触发 `on_call_timeout`，处理方式与 `on_call_hangup` 一致：停止推流并释放资源。

完整示例代码

以下示例整合登录、通话事件响应、发起语音/视频呼叫、以及使用**模拟数据**推送音视频帧的完整流程，用于验证通话链路是否正常。实际接入时，请将模拟数据替换为真实的摄像头/麦克风采集数据。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/time.h>
#include "tc_iot.h"
#include "tc_iot_av.h"

static volatile int s_login_done = 0;
static tc_iot_error_e s_login_rc = TC_IOT_ERR_SUCCESS;
```

```
static tc_iot_av_channel_t *g_video_channel = NULL;
static tc_iot_av_channel_t *g_audio_channel = NULL;
static volatile bool g_pushing = false;
static pthread_t g_video_thread;
static pthread_t g_audio_thread;

static uint64_t current_time_ms(void) {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (uint64_t)tv.tv_sec * 1000 + tv.tv_usec / 1000;
}

// 登录回调：与快速接入一致
static void on_login(tc_iot_error_e error_code, const char
*error_message) {
    s_login_rc = error_code;
    s_login_done = 1;
    if (error_code != TC_IOT_ERR_SUCCESS) {
        printf("[demo] login failed: %d, msg=%s\n", error_code,
               error_message ? error_message : "");
    }
}

// 模拟视频采集：独立线程定时生成占位视频帧（首帧标记为 IDR）并推送。
// 实际接入时替换为真实摄像头采集与 H264/H265 编码逻辑。
static void *video_push_thread_func(void *arg) {
    static uint8_t dummy_video[64] = {0};
    bool first_frame = true;

    while (g_pushing) {
        if (g_video_channel) {
            tc_iot_video_frame vframe;
            memset(&vframe, 0, sizeof(vframe));
            vframe.codec = TC_IOT_VIDEO_CODEC_H264;
            vframe.type = first_frame ? TC_IOT_VIDEO_FRAME_TYPE_IDR
                                      : TC_IOT_VIDEO_FRAME_TYPE_P;
            vframe.rotation = TC_IOT_VIDEO_ROTATION_0;
            vframe.width = 480;
            vframe.height = 320;
```

```
            vframe.data = dummy_video;
            vframe.data_size = sizeof(dummy_video);
            vframe.pts_ms = current_time_ms();
            tc_iot_push_video_frame(g_video_channel, &vframe);
            first_frame = false;
        }
        usleep(40 * 1000); // 模拟 25fps
    }
    return NULL;
}

// 模拟音频采集：独立线程定时生成占位音频帧并推送。
// 实际接入时替换为真实麦克风采集与 PCM/AAC 编码逻辑。
static void *audio_push_thread_func(void *arg) {
    static uint8_t dummy_audio[320] = {0}; // 16kHz/mono/20ms 对应的 PCM
采样点数

    while (g_pushing) {
        if (g_audio_channel) {
            tc_iot_audio_frame aframe;
            memset(&aframe, 0, sizeof(aframe));
            aframe.codec = TC_IOT_AUDIO_CODEC_PCM;
            aframe.sample_rate = TC_IOT_AUDIO_SAMPLE_RATE_16000;
            aframe.channels = TC_IOT_AUDIO_CHANNEL_MONO;
            aframe.frame_duration_ms = 20;
            aframe.data = dummy_audio;
            aframe.data_size = sizeof(dummy_audio);
            aframe.pts_ms = current_time_ms();
            tc_iot_push_audio_frame(g_audio_channel, &aframe);
        }
        usleep(20 * 1000); // 20ms 一帧
    }
    return NULL;
}

// 根据通话媒体类型创建发送通道并启动推流线程
static void start_call_streams(tc_iot_media_content_e media_content) {
    if (media_content == TC_IOT_MEDIA_CONTENT_VIDEO ||
        media_content == TC_IOT_MEDIA_CONTENT_AUDIO_VIDEO) {
```

```
        g_video_channel = tc_iot_create_video_channel(0,
TC_IOT_VIDEO_QUALITY_SD);
    }
    if (media_content == TC_IOT_MEDIA_CONTENT_AUDIO ||
        media_content == TC_IOT_MEDIA_CONTENT_AUDIO_VIDEO) {
        g_audio_channel = tc_iot_create_audio_channel();
    }

    if (!g_pushing) {
        g_pushing = true;
        pthread_create(&g_video_thread, NULL, video_push_thread_func,
NULL);
        pthread_create(&g_audio_thread, NULL, audio_push_thread_func,
NULL);
    }
}

// 停止推流并销毁发送通道
static void stop_call_streams(void) {
    if (g_pushing) {
        g_pushing = false;
        pthread_join(g_video_thread, NULL);
        pthread_join(g_audio_thread, NULL);
    }
    if (g_video_channel) {
        tc_iot_destroy_video_channel(g_video_channel);
        g_video_channel = NULL;
    }
    if (g_audio_channel) {
        tc_iot_destroy_audio_channel(g_audio_channel);
        g_audio_channel = NULL;
    }
}

static void on_call_requested(const tc_iot_contact_s *contact,
tc_iot_call_option_t option) {
    printf("[demo] on_call_requested user_id=%s media=%d\n", contact-
>user_id, option.media_content);

    tc_iot_error_e rc = tc_iot_av_accept(contact, NULL);
```

```
    if (rc != TC_IOT_ERR_SUCCESS) {
        printf("[demo] tc_iot_av_accept failed: %d\n", rc);
        return;
    }
    start_call_streams(option.media_content);
}

static void on_call_timeout(const tc_iot_contact_s *contact) {
    printf("[demo] on_call_timeout user_id=%s\n", contact->user_id);
    stop_call_streams();
}

static void on_call_hangup(const tc_iot_contact_s *contact) {
    printf("[demo] on_call_hangup user_id=%s\n", contact->user_id);
    stop_call_streams();
}

int main(int argc, char **argv) {
    if (argc < 7) {
        printf("Usage: %s -p <product_id> -d <device_id> -s
<device_secret>\n", argv[0]);
        return 0;
    }

    const char *product_id = NULL;
    const char *device_id = NULL;
    const char *device_secret = NULL;
    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-p") == 0 && i + 1 < argc) {
            product_id = argv[++i];
        } else if (strcmp(argv[i], "-d") == 0 && i + 1 < argc) {
            device_id = argv[++i];
        } else if (strcmp(argv[i], "-s") == 0 && i + 1 < argc) {
            device_secret = argv[++i];
        }
    }
    if (!product_id || !device_id || !device_secret) {
        printf("device info error\n");
        return 1;
    }
```

```
// 初始化与登录：与快速接入一致
tc_iot_config_s config;
memset(&config, 0, sizeof(config));
config.storage_path = "./";
config.log_level = TC_IOT_LOG_LEVEL_INFO;

tc_iot_error_e rc = tc_iot_init(&config);
if (rc != TC_IOT_ERR_SUCCESS) {
    printf("tc_iot_init failed: %d\n", rc);
    return 1;
}

tc_iot_device_info_s device_info;
device_info.product_id = product_id;
device_info.device_id = device_id;
device_info.device_secret = device_secret;
device_info.region = "ap-guangzhou";

rc = tc_iot_login(&device_info, on_login);
if (rc != TC_IOT_ERR_SUCCESS) {
    printf("tc_iot_login call failed: %d\n", rc);
    tc_iot_deinit();
    return 1;
}

int waited_ms = 0;
while (!s_login_done && waited_ms < 5000) {
    usleep(100 * 1000);
    waited_ms += 100;
}
if (!s_login_done || s_login_rc != TC_IOT_ERR_SUCCESS) {
    printf("login timeout or failed\n");
    tc_iot_deinit();
    return 1;
}
printf("[demo] login success\n");

// 注册通话事件回调（本文核心内容）
tc_iot_av_observer_s observer;
```

```
    memset(&observer, 0, sizeof(observer));
    observer.on_call_requested = on_call_requested;
    observer.on_call_timeout = on_call_timeout;
    observer.on_call_hangup = on_call_hangup;

    rc = tc_iot_av_init(&observer);
    if (rc != TC_IOT_ERR_SUCCESS) {
        printf("tc_iot_av_init failed: %d\n", rc);
        tc_iot_deinit();
        return 1;
    }

    printf("[demo] ready, waiting for incoming call from app
(60s)...\n");
    sleep(60);

    // 释放资源
    stop_call_streams();
    tc_iot_av_deinit();
    tc_iot_logout();
    tc_iot_deinit();
    printf("[demo] main exit\n");
    return 0;
}
```

编译并运行

将上述代码覆盖到 快速接入 中已创建的项目 `main.c`， `CMakeLists.txt` 无需改动，重新编译并运行即可。

1. 重新编译。

```
cmake --build build -j
```

2. 运行可执行文件，传入设备三元组。

```
./build/iot_quickstart_demo -p YOUR_PRODUCT_ID -d YOUR_DEVICE_NAME -s
'YOUR_DEVICE_SECRET'
```

3. 通过 APP 对该设备发起语音或视频呼叫，观察设备端日志输出 `on_call_requested` 及后续推流日志；挂断后观察 `on_call_hangup` 日志，即代表通话链路验证成功。

常见问题

现象	排查建议
APP 发起呼叫后，设备端未收到 `on_call_requested`。	确认设备已成功登录且 `tc_iot_av_init` 调用成功；确认 `on_call_requested` 已正确赋值（未被 `memset` 清零覆盖）。
`tc_iot_av_accept` 返回失败。	确认设备已成功登录；确认 `contact.user_id` 有效；确认当前没有其他通话正在进行。
通话已接通，但对端看不到画面或听不到声音。	确认接通后已根据 `media_content` 创建对应的发送通道并开始推流；确认视频首帧为 IDR 关键帧。
多次通话后设备资源占用持续增长。	检查 `on_call_hangup` / `on_call_timeout` 是否都正确停止了推流并销毁了发送通道，避免通道未释放导致的资源泄漏。

联系我们

如果您在接入或使用过程中有任何疑问或者建议，欢迎 联系我们 提交反馈。