物联网开发平台

实时监控

腾讯云

【版权声明】

【商标声明】

【服务声明】

本文档意在向您介绍腾讯云全部或部分产品、服务的当时的相关概况，部分产品、服务的内容可能不时有所调整。您所购买的腾讯云产品、服务的种类、服务标准等应由您与腾讯云之间的商业合同约定，除非双方另有约定，否则，腾讯云对本文档内容不做任何明示或默示的承诺或保证。

【联系我们】

我们致力于为您提供个性化的售前购买咨询服务，及相应的技术售后服务，任何问题请联系 4009100100或95716。

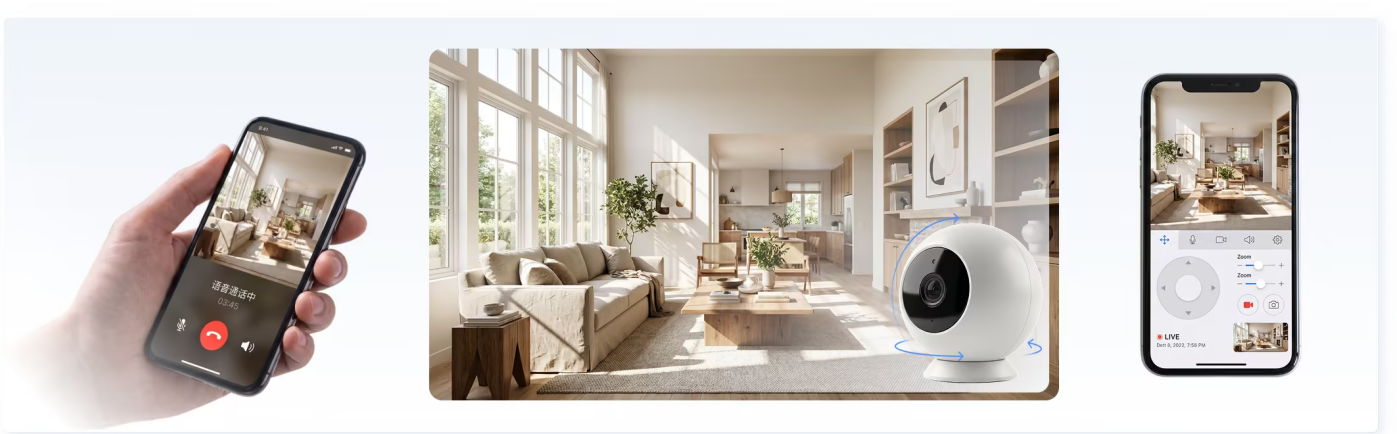
文档目录

实时监控
功能介绍

最近更新时间：2026-07-30 12:00:36

功能概述

实时监控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、低延时的实时监控服务。



产品优势

基本功能

功能	功能描述
实时监控	在 APP 或小程序等 C 端应用中，用户可实时观看 WiFi、蜂窝类 IPC、智能门铃门锁、宠物监控、机器人等设备的实时监控画面。
设备对讲	用户在移动端观看监控画面时，可通过设备对讲功能与设备进行实时语音对讲。
画质设置	用户在观看监控画面时，可对监控画面设置清晰度，如流畅、标清等模式。
云台控制	提供云台控制信令通道，如云台控制、电机控制等。满足开发者不同型号设备的个性化控制设备需求。
录制与截图	用户在查看监控时，可将实时监控录制为一个文件保存在手机上 ，也可将当前监控画面进行截图保存为一个图片文件。

计费说明

实时监控服务需集成设备 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

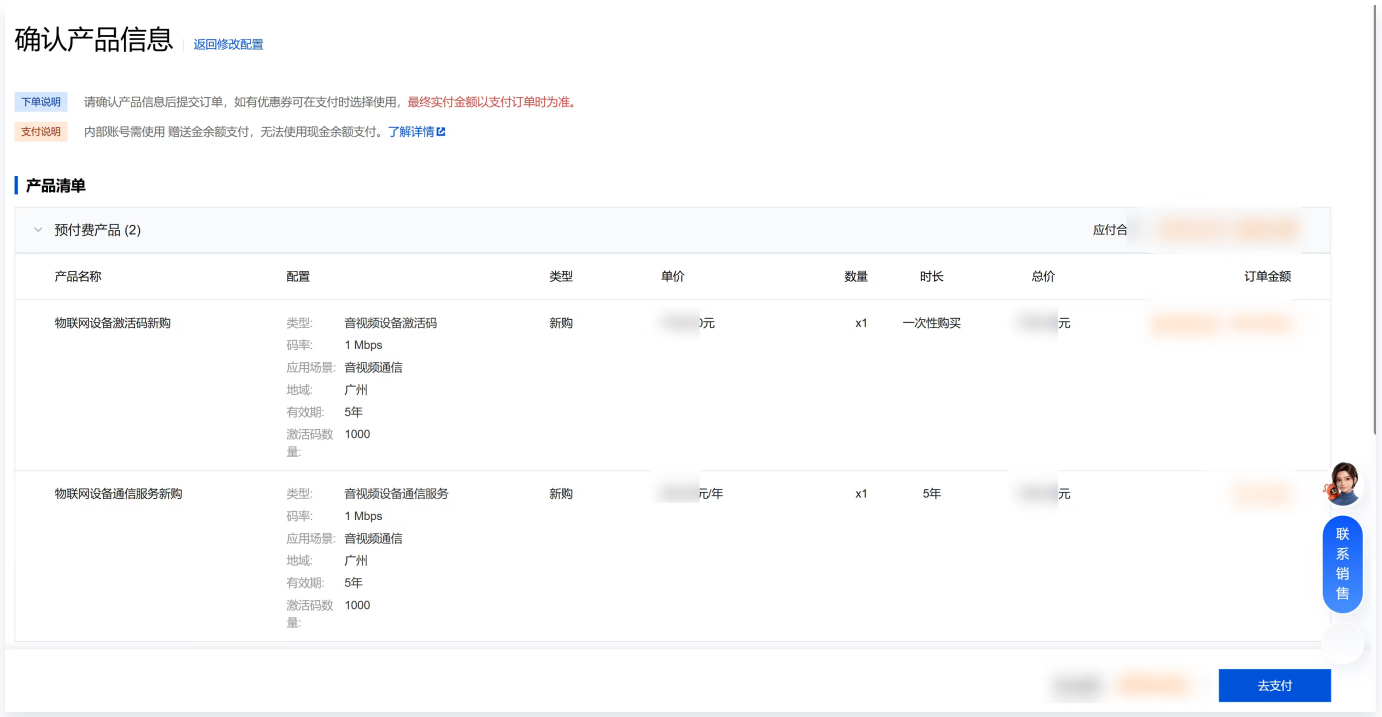
> **注意：**
> 设备接入视频通话服务涉及芯片适配工作，消费实例下的音视频设备接入服务规格目前需开白才可购买。需要该服务的用户可通过 下载中心 查看是否有已适配的芯片 SDK，若没有则可通过填写 芯片适配需求表 或者通过 提交工单 联系我们评估。视频通话服务必须购买音视频接入服务类型，若购买“基础设备接入”服务，则无法使用视频通话。

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”。



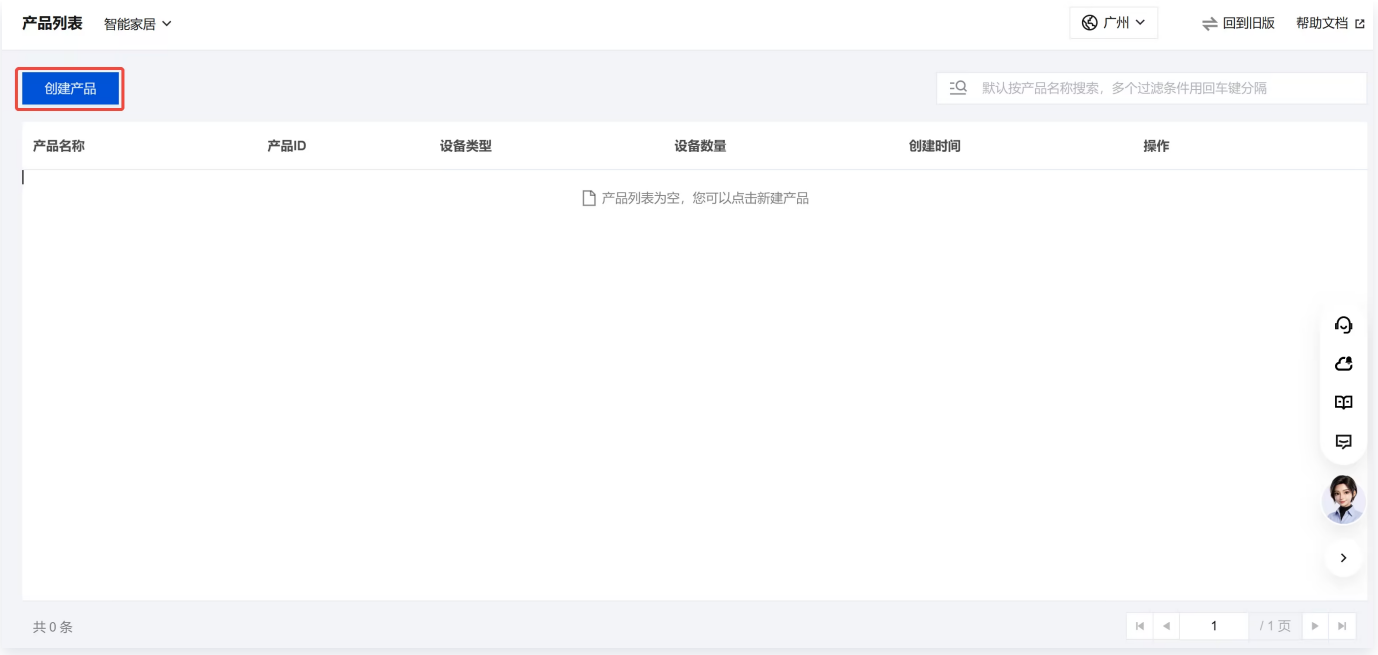
7. 确认后单击**去支付**进行付款，支付成功后系统将您支付订单的激活码数量累加到对应激活码规格的总可用量数。

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。

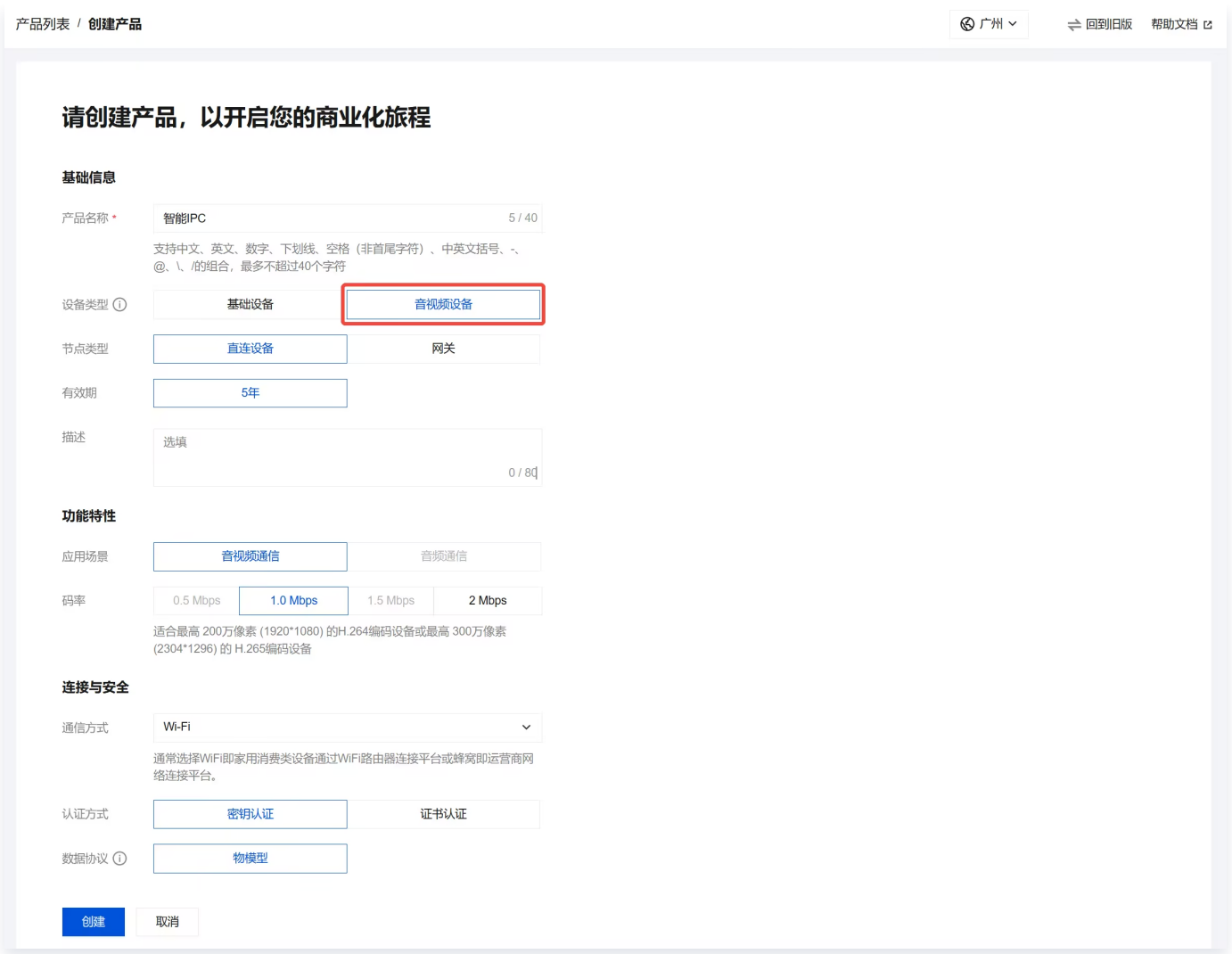
创建产品

1. 登录 物联网开发平台 ，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**。

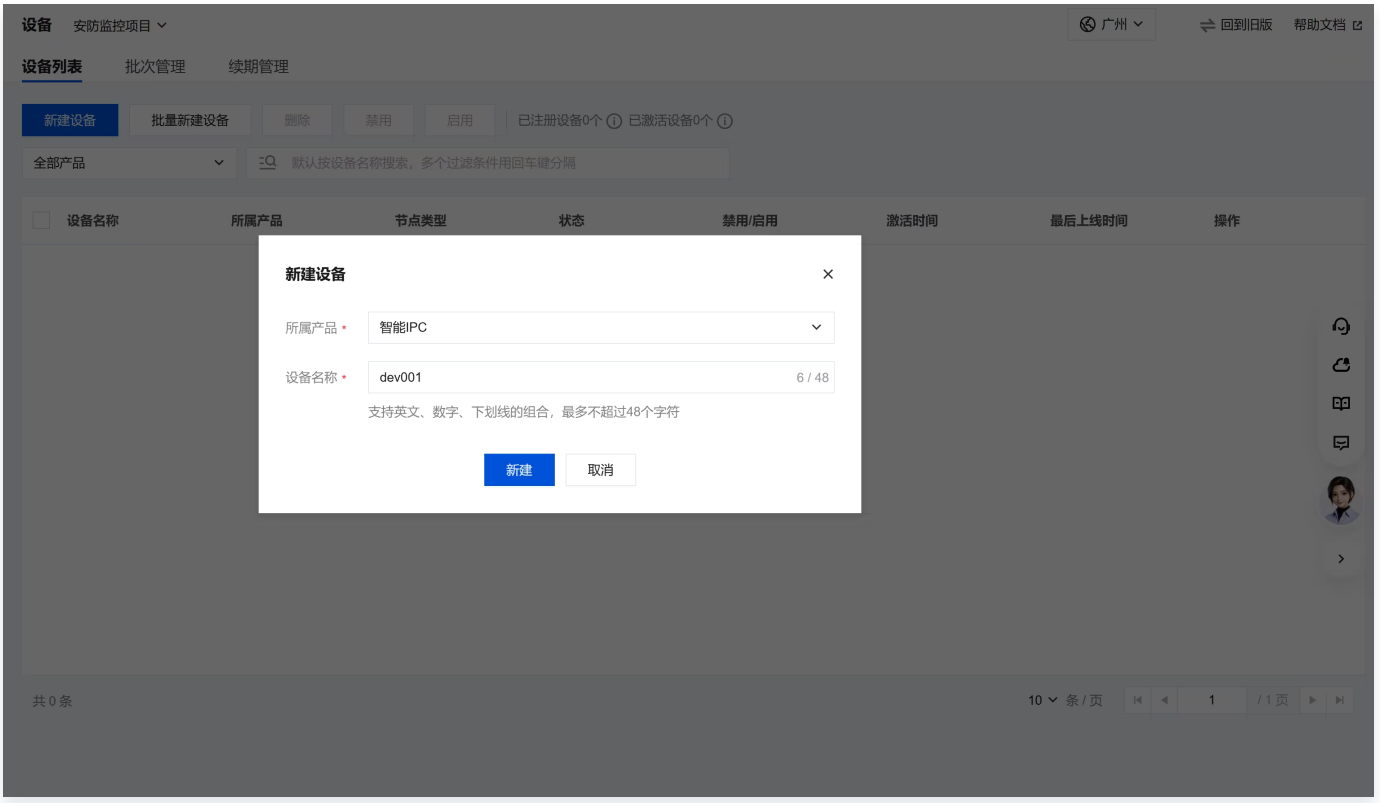
> **注意：**
> 创建产品时若码率规格显示为“不可选”，表示您的账号未购买该规格码率。码率决定了使用该款产品的用户切换清晰度的上限。



创建设备

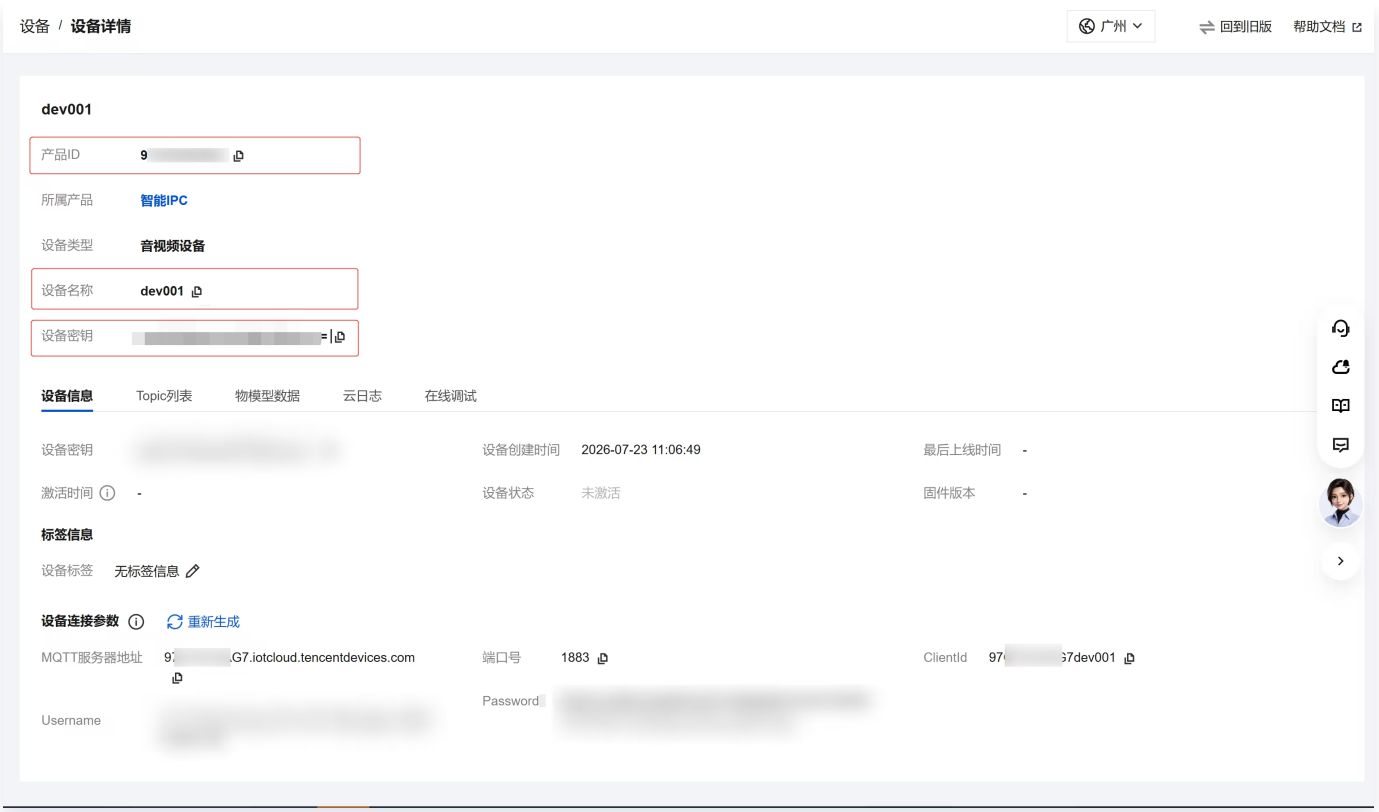
1. 选择消费实例左侧导航栏的**设备**，然后单击**新建设备**。

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**。



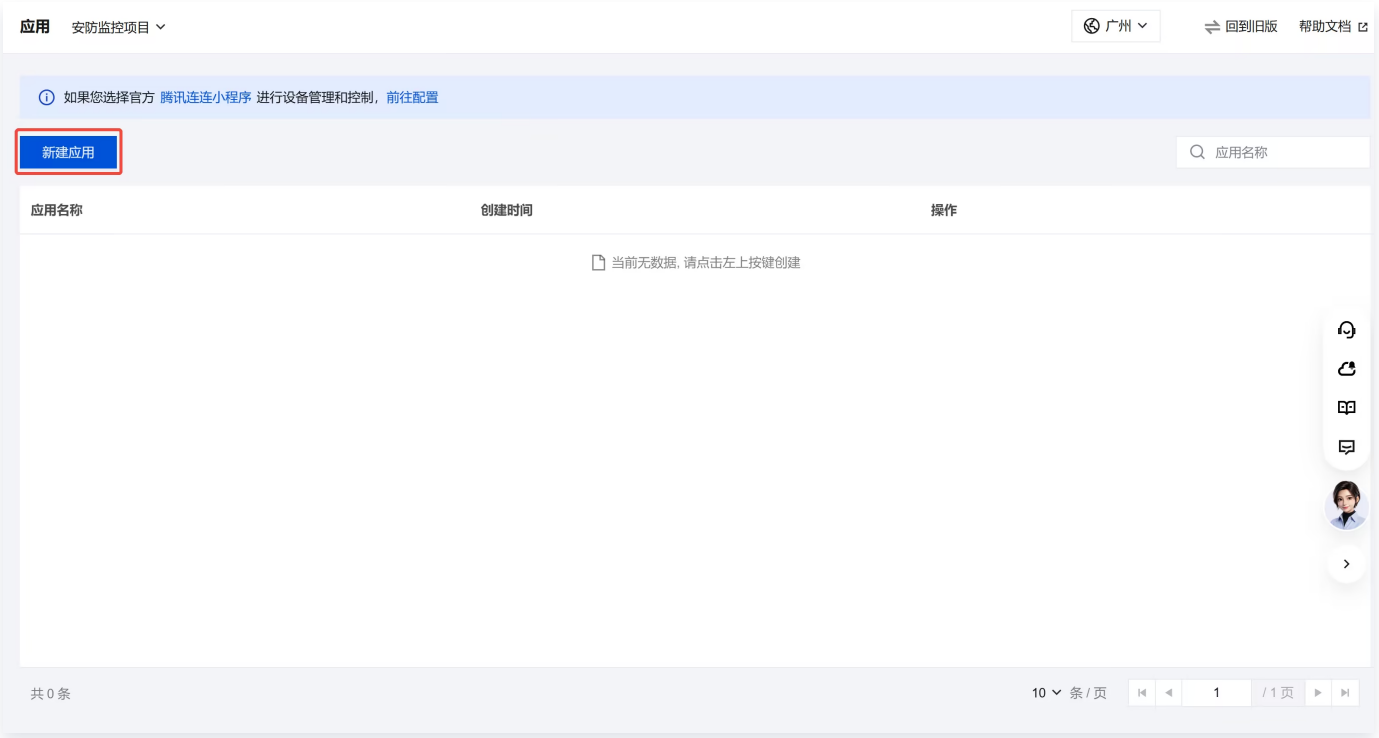
3. 单击创建成功的设备，进入设备详情页获取设备三元组信息。

> **注意：**
> 产品 ID、设备名称、设备密钥三元组信息作为后续实时监控的设备身份凭证。

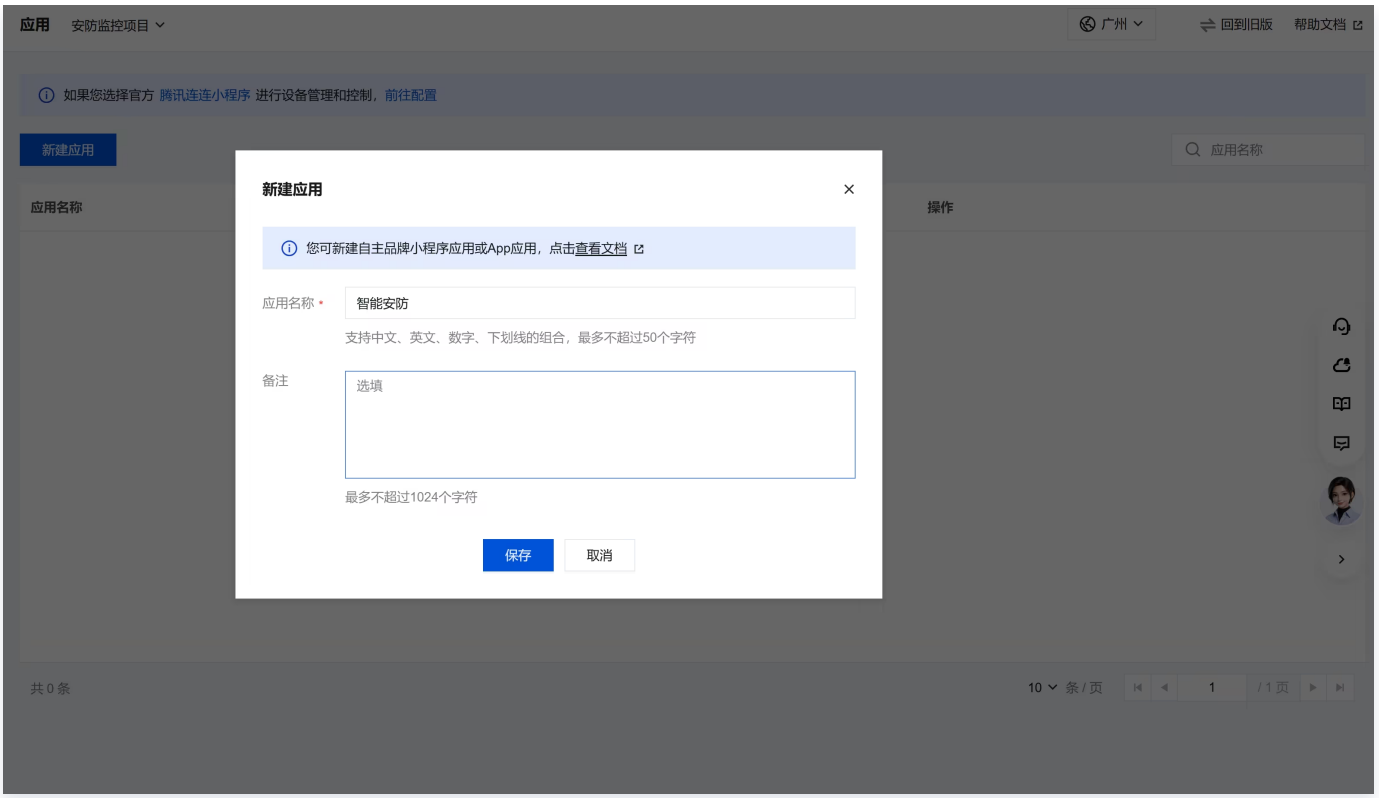


创建应用

1. 选择消费实例左侧导航栏的**应用**，单击**新建应用**。

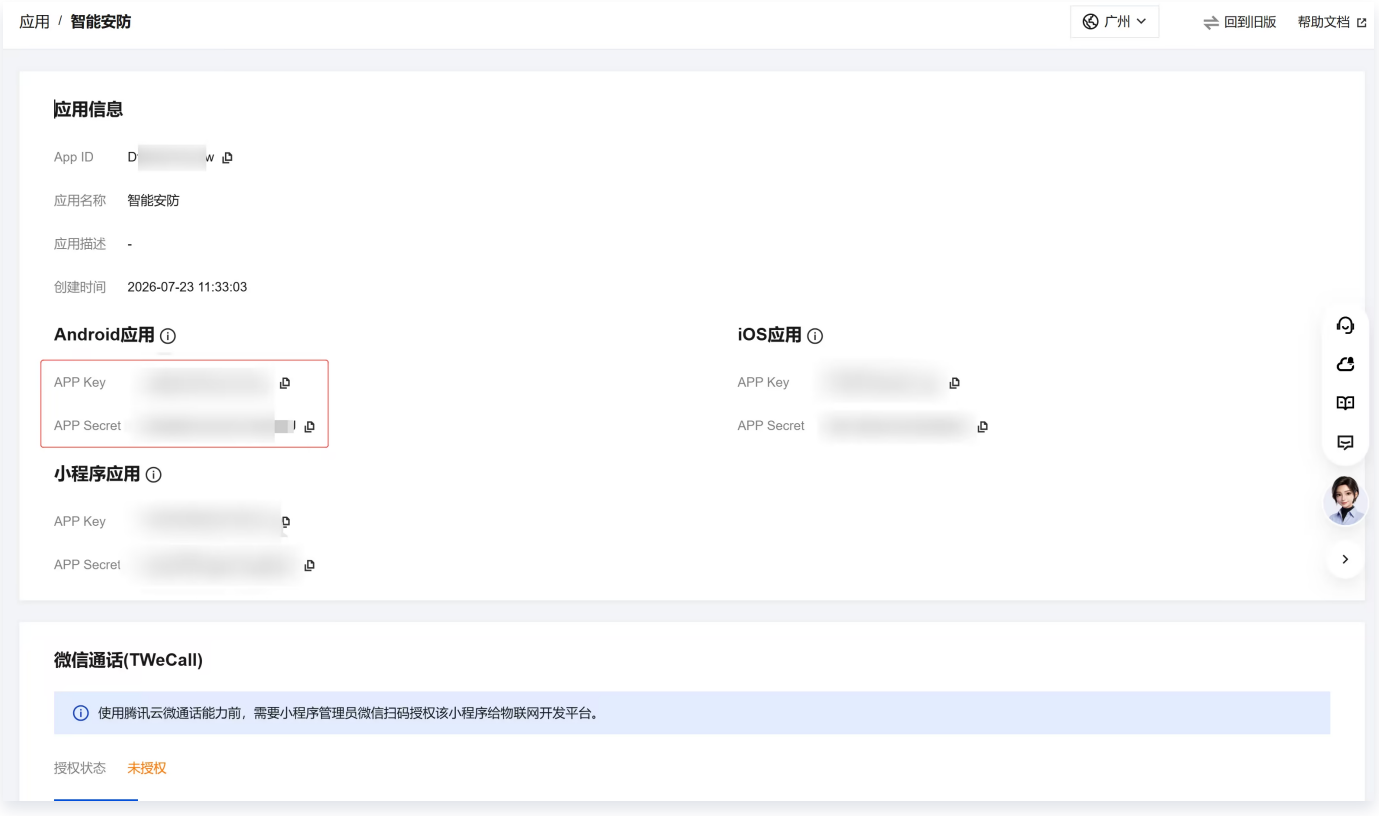


2. 输入应用名称，单击**保存**创建应用。



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”。

设备端接入

最近更新时间：2026-07-30 12:00:37

本文将介绍设备如何实现响应 APP 发起的**监控请求、推送音频帧与视频帧、 切换清晰度**，帮助您快速了解实时监控的功能。

前提条件

在开始之前，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：注册监控事件回调

登录成功后，调用 `tc_iot_av_init` 注册 `tc_iot_av_observer_s` ，实时监控相关的回调如下：

回调	说明
`on_monitor_begin`	APP 发起监控查看请求时触发，携带 `channel_id`（本路监控的通道标识）与 `option`（媒体类型、清晰度等）。
`on_monitor_end`	对端结束监控查看时触发，携带 `channel_id`。
`on_monitor_switch`	APP 在监控查看期间切换清晰度时触发，携带 `channel_id` 与切换后的 `quality`（可选实现，详见 步骤 5）。

```
#include "tc_iot_av.h"

static void on_monitor_begin(int channel_id, tc_iot_call_option_t
option);
static void on_monitor_end(int channel_id);
static void on_monitor_switch(int channel_id, tc_iot_video_quality_e
quality);

tc_iot_av_observer_s observer;
memset(&observer, 0, sizeof(observer));
observer.on_monitor_begin = on_monitor_begin;
observer.on_monitor_end = on_monitor_end;
observer.on_monitor_switch = on_monitor_switch;
```

```
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 on_monitor_begin，创建发送通道

根据 `option.media_content`，在 `on_monitor_begin` 回调中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，`channel_id` 需与回调中的 `channel_id` 一致，`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；多路视频监控同时在线时，可共享同一路音频发送通道，无需重复创建。

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

static void on_monitor_begin(int channel_id, tc_iot_call_option_t
option) {
    printf("[demo] on_monitor_begin channel_id=%d media=%d
quality=%d\n",
           channel_id, option.media_content, option.video_quality);

    if (option.media_content == TC_IOT_MEDIA_CONTENT_VIDEO ||
        option.media_content == TC_IOT_MEDIA_CONTENT_AUDIO_VIDEO) {
        g_video_channel = tc_iot_create_video_channel(channel_id,
option.video_quality);
        if (!g_video_channel) {
            printf("[demo] create video channel failed,
channel_id=%d\n", channel_id);
        }
    }
```

```
    if ((option.media_content == TC_IOT_MEDIA_CONTENT_AUDIO ||
         option.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 3/4 推流示例）
    // start_capture_and_push(...)
}
```

步骤 3：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 4：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 5：响应 on_monitor_switch，切换视频清晰度（可选）

APP 端在监控查看期间切换清晰度时，设备端会通过 `on_monitor_switch` 回调收到通知，携带 `channel_id`（对应的监控通道）与 `quality`（切换后的清晰度）。`quality` 的可选取值（`tc_iot_video_quality_e`）如下：

取值	说明
`TC_IOT_VIDEO_QUALITY_UNKNOWN`	未知/未设置。
`TC_IOT_VIDEO_QUALITY_LD`	流畅（低清晰度）。
`TC_IOT_VIDEO_QUALITY_SD`	标清。

`TC_IOT_VIDEO_QUALITY_HD`	高清。
`TC_IOT_VIDEO_QUALITY_FHD`	超清（全高清）。

视频发送通道创建时已固定清晰度，无法直接修改，因此收到 `on_monitor_switch` 后需要销毁旧通道、按新清晰度重新创建，后续视频帧继续推送到新通道即可：

```
static void on_monitor_switch(int channel_id, tc_iot_video_quality_e
quality) {
    printf("[demo] on_monitor_switch channel_id=%d quality=%d\n",
channel_id, quality);

    if (g_video_channel) {
        tc_iot_destroy_video_channel(g_video_channel);
        g_video_channel = NULL;
    }

    g_video_channel = tc_iot_create_video_channel(channel_id, quality);
    if (!g_video_channel) {
        printf("[demo] recreate video channel failed, channel_id=%d\n",
channel_id);
    }
    // 后续采集到的视频帧继续调用 tc_iot_push_video_frame 推送到新的
g_video_channel
}
```

> **注意：**
> 重建通道期间会短暂中断视频推流，建议尽快完成切换，并在重建后尽快推送一个 IDR 关键帧，保证画面能快速恢复。

步骤 6：响应 on_monitor_end，销毁发送通道

对端结束监控查看时触发 `on_monitor_end`，此时应停止采集并销毁对应的发送通道：

```
static void on_monitor_end(int channel_id) {
    printf("[demo] on_monitor_end channel_id=%d\n", channel_id);

    // 停止本地采集（示意，详见业务实现）
    // stop_capture(...)
```

```
    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;
    }
}
```

说明：
单个设备可同时存在多路监控发送通道（对应不同的 `channel_id`），本文仅演示单路场景；多路并发时建议用数组或列表管理各 `channel_id` 对应的通道指针与采集资源。

完整示例代码

```
#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;
}

static void on_monitor_begin(int channel_id, tc_iot_call_option_t
option) {
    printf("[demo] on_monitor_begin channel_id=%d media=%d
quality=%d\n",
           channel_id, option.media_content, option.video_quality);

    if (option.media_content == TC_IOT_MEDIA_CONTENT_VIDEO ||
        option.media_content == TC_IOT_MEDIA_CONTENT_AUDIO_VIDEO) {
        g_video_channel = tc_iot_create_video_channel(channel_id,
option.video_quality);
    }
    if ((option.media_content == TC_IOT_MEDIA_CONTENT_AUDIO ||
         option.media_content == TC_IOT_MEDIA_CONTENT_AUDIO_VIDEO) &&
        !g_audio_channel) {
        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);
    }
}

static void on_monitor_end(int channel_id) {
    printf("[demo] on_monitor_end channel_id=%d\n", channel_id);

    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;
    }
}

// 切换清晰度（本文步骤 5 核心内容，可选实现）
static void on_monitor_switch(int channel_id, tc_iot_video_quality_e
quality) {
    printf("[demo] on_monitor_switch channel_id=%d quality=%d\n",
channel_id, quality);

    if (g_video_channel) {
        tc_iot_destroy_video_channel(g_video_channel);
        g_video_channel = NULL;
    }
    g_video_channel = tc_iot_create_video_channel(channel_id, quality);
```

```
    if (!g_video_channel) {
        printf("[demo] recreate video channel failed, channel_id=%d\n",
channel_id);
    }
}

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");

    // 注册监控事件回调（本文核心内容）
    tc_iot_av_observer_s observer;
    memset(&observer, 0, sizeof(observer));
    observer.on_monitor_begin = on_monitor_begin;
    observer.on_monitor_end = on_monitor_end;
    observer.on_monitor_switch = on_monitor_switch;

    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 monitor request from app/cloud
(60s)...\n");
    sleep(60);

    // 释放资源
    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);
    }
    if (g_audio_channel) {
        tc_iot_destroy_audio_channel(g_audio_channel);
    }
    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 控制台对该设备发起监控查看请求，观察设备端日志输出 `on_monitor_begin` 及后续推流日志，即代表监控推流链路验证成功。

常见问题

现象	排查建议
发起监控查看后，设备端未收到 `on_monitor_begin`	确认设备已成功登录且 `tc_iot_av_init` 调用成功；确认 `on_monitor_begin` 回调已正确赋值（未被 `memset` 清零覆盖）。
APP 端看不到监控画面	确认已在 `on_monitor_begin` 中正确创建视频发送通道；确认推送的首帧为 IDR 关键帧；确认 `tc_iot_push_video_frame` 返回值为 `TC_IOT_ERR_SUCCESS`。
APP 端听不到声音	确认 `option.media_content` 包含音频（`TC_IOT_MEDIA_CONTENT_AUDIO` 或 `TC_IOT_MEDIA_CONTENT_AUDIO_VIDEO`）；确认已创建音频发送通道且 `tc_iot_push_audio_frame` 调用成功。
多路监控同时在线时声音异常（重复/杂音）	检查是否为每路监控重复创建了音频发送通道；音频发送通道应全局只创建一路，多路视频共享推送。
`tc_iot_push_video_frame` / `tc_iot_push_audio_frame` 返回失败	确认传入的发送通道指针非空且尚未被销毁；确认 `on_monitor_end` 触发后已停止继续推流。
切换清晰度后，设备端未收到 `on_monitor_switch`	确认设备已成功登录且 `tc_iot_av_init` 调用成功；确认 `on_monitor_switch` 已正确赋值（未被 `memset` 清零覆盖）；确认监控会话已建立（`on_monitor_begin` 已触发）。
收到 `on_monitor_switch` 回调但画面清晰度未变化	确认已先销毁旧的视频发送通道再按新 `quality` 创建；确认推送视频帧时使用的是重建后的新通道指针，而非旧指针。
切换清晰度时画面卡顿或短暂黑屏	通道重建期间存在短暂推流中断，属正常现象；建议重建后尽快推送一个 IDR 关键帧，保证画面能快速恢复。

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。

应用端接入

最近更新时间：2026-07-30 12:00:37

本文介绍腾讯云物联网（IoT）应用端 SDK 中实时监控（`TXIoTMonitorSession`）的使用方法。完成 快速接入 并登录 SDK 后，您可以获取监控会话对象，对已绑定的设备发起实时监控，播放设备实时音视频，并监听会话状态与错误回调。您还可以进一步使用云台控制、录制与截图、语音对讲等功能，详情参见 下一步：高级功能 。

前提条件

在调用本文 API 前，请确保已完成以下准备工作：

- 已开通腾讯云物联网相关服务，并在控制台完成实例、应用和设备的准备工作（可参见 开通服务 ）。
- 已在客户端工程中集成 IoT 应用端 SDK。
- 已通过业务后台生成登录签名，并调用 `TXIoTEngine.login` 完成登录。
- 目标设备已绑定到当前登录账户名下（请参考 设备管理 完成设备绑定）。
- 使用本地麦克风/摄像头对讲前，确保已在工程中申请并获取相应系统权限。

使用建议

- 调用顺序：获取监控会话对象 → 注册监听 → 尽早 startRemoteView 设置渲染视图（可在 startSession 之前调用）→ startSession → 会话建立后自动播放画面；结束时调用 stopSession 停止会话。
- 提前绑定设备：确保目标设备已绑定到当前账户，否则相关调用会返回权限类错误。
- 尽早注册监听：在 startSession 之前完成 addListener，避免错过 `onSessionEstablished` 等早期回调。
- 及时释放：不再需要监控时调用 stopSession 并 removeListener，避免资源泄漏与重复回调。
- 统一错误处理：在 `onError` 中集中处理错误码，结合错误码表给出用户可理解的提示。

获取监控会话对象

在 Android 和 iOS 上，登录成功后，均可通过 `TXIoTEngine` 单例实例调用 `getMonitorSession()` 获取监控会话对象 `TXIoTMonitorSession`，无需为每个设备单独创建。

> **警告：**
> 监控会话对象为单例，同一时刻只存在一个监控会话。若已有会话正在运行中（已调用 startSession 且尚未 stopSession），再次调用 startSession 会被直接忽略，不会打断或切换已有会话；无需也不能重复获取多个会话实例。

与视频通话的优先级关系：

实时监控的优先级低于视频通话。若设备当前正处于实时监控拉流状态，应用端对该设备发起视频通话时，视频通话会打断并抢占实时监控的拉流，监控会话会被中断（详见 视频通话 ）。

注册监听回调

监控会话的状态变化与错误均通过监听接口返回。请在调用 startSession 之前完成监听注册，避免错过 `onSessionEstablished` 等早期回调。

方法	说明
addListener	添加监控会话状态与错误监听器，回调接口定义见 监控事件回调（iOS：`addDelegate`）。
removeListener	移除监控会话监听器（iOS：`removeDelegate`）。

Android

```
TXIoTMonitorSessionListener monitorListener = new
TXIoTMonitorSessionListener() {
    @Override
    public void onSessionEstablished() {
        // 会话建立成功，可开始播放画面
    }

    // 其他回调...
};

session.addListener(monitorListener);
```

iOS

```
@interface ViewController () <TXIoTMonitorSessionDelegate>
@end

@implementation ViewController

- (void)onSessionEstablished {
    // 会话建立成功
```

```
}

- (void)onError:(NSInteger)channelId
      errorCode:(TXIoTErrorCode)errorCode
   errorMessage:(NSString *)errorMessage {
    // 发生错误，统一处理错误码与提示
}

@end

// 注册代理
[session addDelegate:self];
// 退出时注销
// [session removeDelegate:self];
```

开始与停止监控

通过 startSession 与目标设备建立监控会话，会话建立成功后即可播放实时画面。结束监控时调用 stopSession 释放资源。

方法	说明
startSession	与目标设备建立监控会话，`deviceId`（TXIoTDeviceId）为目标设备标识。
stopSession	停止当前监控会话并释放资源。

Android

```
TXIoTMonitorSession session =
TXIoTEngine.getInstance(context).getMonitorSession();
if (session == null) {
    // SDK 未登录或登录已过期，请先完成登录
    return;
}
// 监听已在“监听会话状态与错误回调”一节注册，此处直接开始监控
```

```
// 构造目标设备标识
TXIoTDeviceId deviceId = new TXIoTDeviceId();
deviceId.productId = "已绑定到当前账户的 productId";
deviceId.deviceName = "已绑定到当前账户的 deviceName";

// 开始监控
session.startSession(deviceId);

// 结束时停止监控
session.stopSession();
```

iOS

```
TXIoTMonitorSession *session = [[TXIoTEngine getInstance]
getMonitorSession];
if (session == nil) {
    // SDK 未登录或登录已过期，请先完成登录
    return;
}
// 监听已在“监听会话状态与错误回调”一节注册，此处直接开始监控

// 构造目标设备标识
TXIoTDeviceId *deviceId = [[TXIoTDeviceId alloc] init];
deviceId.productId = @"已绑定到当前账户的 productId";
deviceId.deviceName = @"已绑定到当前账户的 deviceName";

// 开始监控
[session startSession:deviceId];

// 结束时停止监控
[session stopSession];
```

播放设备实时画面

通过 startRemoteView 为指定通道设置渲染视图并播放实时音视频。您可以在调用 startSession 之前就调用 startRemoteView 设置好渲染视图。

方法	说明
startRemoteView	在指定视图播放 `channelId` 通道的实时画面， `streamType` 为码流类型（取值见「数据结构体」中的 TXIoTStreamType ）。 `channelId` 表示设备的视频通道，每个通道对应这个设备的一个摄像头，单通道设备使用 `0` 。
stopRemoteView	停止播放 `channelId` 通道的实时画面。

Android

```
// remoteView 为布局中的 TXCloudVideoView
TXCloudVideoView remoteView = findViewById(R.id.remote_view);

// 开始播放0号通道高清流
int channelId = 0; // channelId 表示设备的视频通道，需要与设备端的通道 Id 对齐，单通道设备使用 0
TXIoTStreamType streamType = TXIoTStreamType.HD; // 需要查看的视频清晰度，SDK 会将该信息同步给设备端
session.startRemoteView(channelId, streamType, remoteView);

// 停止播放0号通道
session.stopRemoteView(channelId);
```

iOS

```
// remoteView 为布局中的 UIView
UIView *remoteView = self.remoteVideoView;

// 开始播放0号通道高清流
NSInteger channelId = 0; // channelId 表示设备的视频通道，需要与设备端的通道 Id 对齐，单通道设备使用 0
```

```
TXIoTStreamType streamType = TXIoTStreamTypeHD; // 需要查看的视频清晰度，SDK 会将该信息同步给设备端
// 开始播放高清流
[session startRemoteView:channelId
              streamType:streamType
                    view:remoteView];

// 停止播放
[session stopRemoteView:0];
```

切换视频清晰度

查看监控期间，应用端可动态切换通道画面的清晰度（高清 HD / 流畅 SD）。切换通过 switchRemoteStream 完成，无需重新建立会话；应用端切换后，设备端会收到 `on_monitor_switch` 回调并推送对应清晰度的视频流（设备端处理逻辑见 切换清晰度）。

> ⚠ **注意：**
> 不同清晰度可能对应不同的计费标准，请确保应用端实际调用的清晰度与用户购买的套餐一致，避免因清晰度与计费套餐不匹配引发计费争议。

接口说明

方法	说明
switchRemotoeStream	将 `channelId` 通道的视频流切换为指定清晰度，`streamType` 取值见枚举类型 TXIoTStreamType。

调用示例

Android

```
// 切换为流畅流
session.switchRemoteStream(0,
TXIoTMonitorSession.TXIoTStreamType.SD);
// 切换为高清流
```

```
session.switchRemoteStream(0,
TXIoTMonitorSession.TXIoTStreamType.HD);
```

iOS

```
// 切换为流畅流
[session switchRemoteStream:0 streamType:TXIoTStreamTypeSD];
// 切换为高清流
[session switchRemoteStream:0 streamType:TXIoTStreamTypeHD];
```

音频控制

监控过程中可静音远端音频。通过 muteRemoteAudio（单通道）与 muteAllRemoteAudio（全部通道）控制远端音频播放。

方法	说明
muteRemoteAudio	`mute` 为 `true` 时静音指定通道的远端音频。
muteAllRemoteAudio	一键静音/恢复所有通道的远端音频。

Android

```
// 远端音频控制
session.muteRemoteAudio(0, true);
session.muteAllRemoteAudio(false);
```

iOS

```
// 远端音频控制
```

```
[session muteRemoteAudio:0 mute:YES];
[session muteAllRemoteAudio:NO];
```

状态与回调通知

监控会话通过监听接口（Android：`TXIoTMonitorSessionListener`，iOS：`TXIoTMonitorSessionDelegate`）回调状态与错误，回调接口如下：

回调	触发时机	关键参数
`onSessionEstablished`	监控会话建立成功。	无。
`onSessionReconnecting`	会话中断，正在重连。	无。
`onSessionRecovery`	会话重连成功。	无。
`onRemoteStreamAvailable`	远端音视频流可用状态变化。	`channelId, available`
`onRenderFirstFrame`	指定通道首帧已渲染。	`channelId`
`onPlayStateChanged`	播放状态变化。	`channelId, state`（PLAYING / LOADING / STOPPED）
`onError`	会话或某通道发生错误。	`channelId, errCode, errMsg`

> **注意：**
> 会话连接中断时会触发 `onSessionReconnecting`，SDK 会自动尝试重连；重连成功触发 `onSessionRecovery`，无需业务层主动重建会话。若重连持续失败，最终会通过 `onError` 回调错误。

错误处理

监控会话的所有错误均通过 `onError` 回调错误码与错误信息。常见错误码如下，Android 与 iOS 命名不同，请按所用平台查看：
完整错误码定义见 TXIoTErrorCode 。

Android

错误码	说明	建议处理方式
`ERR_INVALID_PARAMETER`	参数不合法	检查 `deviceId` 、 `channelId` 、路径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 后再发起监控。
`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	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_SPEAKER_START_FAIL`	扬声器启动失败	检查音频输出设备是否正常。
`ERR_DEVICE_SWITCH_TO_VOIP`	设备已切换到 VoIP 通话	结束通话或等待通话结束后再发起监控。

iOS

错误码	数值	说明	建议处理方式
`TXIoTErrorCodeInvalidParameter`	-2	参数不合法。	检查 `deviceId` 、 `channelId` 、路径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 后再发起监控。

`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。	确认目标设备已绑定到当前账户（见 前提条件 ）。
`TXIoTErrorCodeDeviceNotExist`	-10009	设备不存在。	确认 `deviceId` 是否正确、设备是否已绑定。
`TXIoTErrorCodeDeviceOffline`	-10010	设备离线。	确认设备在线后重试。
`TXIoTErrorCodeSpeakerStartFail`	-20006	扬声器启动失败。	检查音频输出设备是否正常。
`TXIoTErrorCodeDeviceSwitchToVoIP`	-2010	设备已切换到 VoIP 通话。	结束通话或等待通话结束后再发起监控。

下一步：高级功能

完成基础接入后，您可以进一步接入以下高级功能，丰富实时监控的使用场景：

- 云台控制： 通过 PTZ 指令控制摄像头转动、变焦，调整监控视角。
- 录制与截图：本地录制监控画面，并抓取实时截图保存。
- 语音对讲 ： 与设备端进行双向语音对讲。

云台控制

最近更新时间：2026-07-30 12:00:37

本文将介绍设备如何实现实时监控场景下**云台控制**功能。

应用端接入

对于支持云台（PTZ）的设备，应用端在查看监控期间可通过 sendPTZCommand 控制镜头方向、变焦与停止控制。指令下发后，设备端会收到 `on_ptz_command_received` 回调并驱动云台硬件（见下文 设备端接入 ）。指令类型见枚举类型 TXIoTPTZCommand，速度 `speed` 由设备能力决定（通常取值 1 ~ 10）。

> **注意：**
> 方向类操作通常为 “按住转动、松开停止”，按下时下发方向指令（如 `LEFT` ），松开时必须下发 `STOP` ，否则云台会持续转动无法停止。

接口说明

方法	说明
sendPTZCommand	向 `channelId` 通道发送云台指令， `command` 为指令类型（ TXIoTPTZCommand ）， `speed` 为速度。

调用示例

Android

```
// 控制云台向左转动，速度 5
session.sendPTZCommand(0,                                    // 通道 ID，如果仅有一个通道，则传 0
        TXIoTMonitorSession.TXIoTPTZCommand.LEFT, // 云台指令，例如：向左转动
        5);                                                  // 转动速度

// 松开时停止云台转动
session.sendPTZCommand(0,
        TXIoTMonitorSession.TXIoTPTZCommand.STOP,
```

```
        0);
```

iOS

```
// 控制云台向左转动，速度 5
[session sendPTZCommand:0                    // 通道 ID，如果仅有一个通
道，则传 0
                command:TXIoTPTZCommandLeft // 云台指令，例如：向左转动
                   speed:5];                 // 转动速度

// 松开时停止云台转动
[session sendPTZCommand:0
                command:TXIoTPTZCommandStop
                   speed:0];
```

设备端接入

接入步骤

步骤 1：注册 on_ptz_command_received 回调

调用 `tc_iot_av_init` 注册 `tc_iot_av_observer_s` 时，为 `on_ptz_command_received` 赋值。APP 端在查看监控期间发起云台操作时，设备端会通过该回调收到通知，携带 `channel_id`（对应的监控通道）、`ptz_command`（云台指令）与 `speed`（转动速度）：

```
#include "tc_iot_av.h"

static void on_ptz_command_received(int channel_id, ptz_command_e
ptz_command, int speed);

tc_iot_av_observer_s observer;
memset(&observer, 0, sizeof(observer));
observer.on_ptz_command_received = on_ptz_command_received;

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);
}
```

`ptz_command` 的可选取值（`ptz_command_e`）如下：

取值	说明
`PTZ_CMD_UP`	向上转动。
`PTZ_CMD_DOWN`	向下转动。
`PTZ_CMD_LEFT`	向左转动。
`PTZ_CMD_RIGHT`	向右转动。
`PTZ_CMD_ZOOM_IN`	镜头放大（拉近）。
`PTZ_CMD_ZOOM_OUT`	镜头缩小（拉远）。
`PTZ_CMD_STOP`	停止转动。

步骤 2：响应回调，驱动云台硬件

在回调中根据 `ptz_command` 调用对应的云台硬件驱动接口完成转动，`speed` 用于控制转动速度。云台驱动接口由具体硬件平台提供，以下以自定义 `ptz_motor_move` / `ptz_motor_zoom` / `ptz_motor_stop` 函数表示，请替换为您设备实际的云台驱动接口：

```
// 以下函数由业务自行实现，对接具体硬件平台的云台驱动接口
extern void ptz_motor_move(int direction, int speed);  // direction: 0=上
1=下 2=左 3=右
extern void ptz_motor_zoom(bool zoom_in, int speed);
extern void ptz_motor_stop(void);

static void on_ptz_command_received(int channel_id, ptz_command_e
ptz_command, int speed) {
    printf("[demo] on_ptz_command_received channel_id=%d cmd=%d
speed=%d\n",
           channel_id, ptz_command, speed);

    switch (ptz_command) {
        case PTZ_CMD_UP:
            ptz_motor_move(0, speed);
```

```
            break;
        case PTZ_CMD_DOWN:
            ptz_motor_move(1, speed);
            break;
        case PTZ_CMD_LEFT:
            ptz_motor_move(2, speed);
            break;
        case PTZ_CMD_RIGHT:
            ptz_motor_move(3, speed);
            break;
        case PTZ_CMD_ZOOM_IN:
            ptz_motor_zoom(true, speed);
            break;
        case PTZ_CMD_ZOOM_OUT:
            ptz_motor_zoom(false, speed);
            break;
        case PTZ_CMD_STOP:
            ptz_motor_stop();
            break;
        default:
            break;
    }
}
```

> ⚠ **注意：**
> APP 端云台方向操作通常是"按住转动、松开发送 `PTZ_CMD_STOP` "，请务必正确处理 `PTZ_CMD_STOP` ，否则云台会持续转动无法停止。

常见问题

现象	排查建议
发起云台操作后，设备端未收到 `on_ptz_command_received` 。	确认设备已成功登录且 `tc_iot_av_init` 调用成功；确认 `on_ptz_command_received` 已正确赋值（未被 `memset` 清零覆盖）；确认监控会话已建立（ `on_monitor_begin` 已触发）。
收到回调但云台无转动。	确认已正确对接云台硬件驱动接口；确认 `switch` 分支覆盖了实际下发的 `ptz_command` 取值。
云台持续转动不停止。	检查是否正确处理了 `PTZ_CMD_STOP` 分支并调用了停止接口。

语音对讲

最近更新时间：2026-07-30 12:00:37

本文将介绍设备如何实现实时监控场景下的**语音对讲**功能（设备端与 APP 进行语音对话）。

应用端接入

语音对讲需要设备端与应用端协同：设备端负责音频的采集与播放（见下文 设备端接入 ），应用端则在查看监控期间开启本地麦克风采集，将 APP 语音传输给设备端。如需双向画面，还可同时开启本地摄像头。

> ⚠ **注意：**
> 开启本地麦克风与摄像头前，请确保已在工程中申请并获取麦克风、摄像头权限，否则 startLocalAudio / startLocalVideo 会返回 `ERR_MIC_NOT_AUTHORIZED` / `ERR_CAMERA_NOT_AUTHORIZED` 等错误码。

接口说明

方法	说明
startLocalAudio	开始采集、传输本地麦克风音频。
stopLocalAudio	停止采集、传输本地麦克风音频。
muteLocalAudio	`mute` 为 `true` 时静音本地麦克风。
startLocalVideo	开始采集、传输本地摄像头画面，`view` 为本地渲染视图，编码参数为 TXIoTVideoEncoderParams 。
stopLocalVideo	停止采集、传输本地摄像头画面。

调用示例

Android

```
// 本地麦克风（对讲）
session.startLocalAudio();
```

```
session.muteLocalAudio(false);
// session.stopLocalAudio();

// 本地摄像头（可选，用于双向视频）
TXCloudVideoView localView = findViewById(R.id.local_view);
TXIoTVideoEncoderParams params = new TXIoTVideoEncoderParams();
params.resolution = TXIoTVideoResolution.RESOLUTION_640_360;
session.startLocalVideo(localView, params);
// session.stopLocalVideo();
```

iOS

```
// 本地麦克风（对讲）
[session startLocalAudio];
[session muteLocalAudio:NO];
// [session stopLocalAudio];

// 本地摄像头（可选，用于双向视频）
UIView *localView = self.localVideoView;
TXIoTVideoEncoderParams *params = [[TXIoTVideoEncoderParams alloc]
init];
params.resolution = TXIoTVideoResolution_640_360;
[session startLocalVideo:localView videoEncoderParams:params];
// [session stopLocalVideo];
```

错误处理

语音对讲需开启本地麦克风与摄像头，以下错误码在对讲过程中可能返回，完整错误码定义见 TXIoTErrorCode 。

Android

错误码	说明	建议处理方式

ERR_CAMERA_START_FAIL	摄像头启动失败。	检查设备摄像头是否被占用或硬件异常。
ERR_CAMERA_NOT_AUTHORIZED	摄像头权限未授权。	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	麦克风权限未授权。	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。

iOS

错误码	数值	说明	建议处理方式
TXIoTErrorCodeCameraStartFail	-2000	摄像头启动失败。	检查设备摄像头是否被占用或硬件异常。
TXIoTErrorCodeCameraNotAuthorized	-2001	摄像头权限未授权。	在工程中申请并获取摄像头权限。
TXIoTErrorCodeCameraOccupy	-2002	摄像头被占用。	停止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	麦克风权限未授权。	在工程中申请并获取麦克风权限。
TXIoTErrorCodeMicOccupy	-2005	麦克风被占用。	停止其他占用麦克风的功能后再试。

设备端接入

接入步骤

步骤 1：注册 on_audio_frame_received 回调

调用 `tc_iot_av_init` 注册 `tc_iot_av_observer_s` 时，为 `on_audio_frame_received` 赋值。APP 端在查看监控期间发起语音对讲时，设备端会通过该回调持续收到音频帧：

```
#include "tc_iot_av.h"

static void on_audio_frame_received(const tc_iot_audio_frame *frame);

tc_iot_av_observer_s observer;
memset(&observer, 0, sizeof(observer));
observer.on_audio_frame_received = on_audio_frame_received;

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：解析音频帧

`on_audio_frame_received` 回调携带的 `tc_iot_audio_frame` 结构体字段如下：

字段	说明
`codec`	音频编码格式，固定为 `TC_IOT_AUDIO_CODEC_PCM` 。
`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）。
`data` / `data_size`	PCM 音频数据指针与长度，可直接送入播放器。
`pts_ms`	帧时间戳（毫秒）。

> **说明：**
> 目前 `codec` 固定为 `TC_IOT_AUDIO_CODEC_PCM` ，无需考虑其他编码格式。

步骤 3：将音频数据送入播放器

将 `data` 指向的 PCM 数据写入设备的本地播放器（扬声器）完成播放。播放器接口由具体硬件平台提供，以下以自定义 `speaker_play_pcm` 函数表示，请替换为您设备实际的音频输出接口：

```
// speaker_play_pcm 由业务自行实现，对接具体硬件平台的音频输出接口
extern void speaker_play_pcm(const uint8_t *pcm_data, size_t pcm_size);

static void on_audio_frame_received(const tc_iot_audio_frame *frame) {
    if (!frame || !frame->data || frame->data_size == 0) {
        return;
    }
    speaker_play_pcm(frame->data, frame->data_size);
}
```

常见问题

现象	排查建议
开启语音对讲后，设备端未收到 `on_audio_frame_received` 。	确认设备已成功登录且 `tc_iot_av_init` 调用成功；确认 `on_audio_frame_received` 已正确赋值（未被 `memset` 清零覆盖）；确认监控会话已建立（ `on_monitor_begin` 已触发）。
设备端收到回调但播放无声音。	确认已正确对接本地音频播放硬件接口；检查 `sample_rate` 、 `channels` 与播放器初始化参数是否一致。
播放声音卡顿或有杂音。	检查 `on_audio_frame_received` 内是否有耗时阻塞操作（如同步写文件、网络请求等），建议将数据快速拷贝到缓冲队列后异步播放，避免阻塞回调影响后续帧接收。

录制与截图

最近更新时间：2026-07-30 12:00:37

本文将介绍安防监控场景下的录制与截图功能。录制与截图为纯应用端能力：应用端对正在拉取的监控画面进行截图或本地录制，设备端无需接入任何工作。

应用端接入

截图

在查看监控（ startRemoteView 成功）期间，调用 takeSnapshot 对指定通道的当前画面截图，截图结果通过 `onSnapshotComplete` 回调返回。

接口说明

方法 / 回调	说明
takeSnapshot	对 `channelId` 通道的当前画面发起截图。
`onSnapshotComplete`	截图完成回调，返回 `channelId` 、图片对象与错误码。

调用示例

Android

```
// 发起截图
session.takeSnapshot(0);

// 在 TXIoTMonitorSessionListener 中接收结果
@Override
public void onSnapshotComplete(int channelId, Bitmap image, int
errCode) {
    if (errCode == 0 && image != null) {
        // 保存或展示截图
        saveBitmapToAlbum(image);
    } else {
        Log.e(TAG, "snapshot failed: " + errCode);
    }
```

```
}
```

iOS

```
// 发起截图
[session takeSnapshot:0];

// 在 TXIoTMonitorSessionDelegate 中接收结果
- (void)onSnapshotComplete:(int)channelId image:(UIImage *)image
errCode:(int)errCode {
    if (errCode == 0 && image) {
        // 保存或展示截图
        UIImageWriteToSavedPhotosAlbum(image, nil, nil, nil);
    } else {
        NSLog(@"snapshot failed: %d", errCode);
    }
}
```

本地录制

在监控查看期间，调用 startLocalRecording 将当前通道画面录制为本地视频文件，调用 stopLocalRecording 结束录制。录制状态通过 `onLocalRecordBegin` / `onLocalRecording` / `onLocalRecordComplete` 三个回调返回。

接口说明

方法 / 回调	说明
startLocalRecording	开始本地录制，参数为 TXIoTLocalRecordingParams 。
stopLocalRecording	结束本地录制。
`onLocalRecordBegin`	录制开始回调，返回 `errCode` 与录制文件路径。
`onLocalRecording`	录制中进度回调，每秒钟回调一次已录制时长 `durationMs` 与文件路径。
`onLocalRecordComplet`	录制结束回调，返回 `errCode` 与最终录制文件路径。

e

调用示例

Android

```
// 开始录制
TXIoTLocalRecordingParams params = new TXIoTLocalRecordingParams();
params.channelId = 0;
params.outputPath = getExternalFilesDir(null) + "/record.mp4";
session.startLocalRecording(params);

// 结束录制
session.stopLocalRecording();

// 录制状态回调（TXIoTMonitorSessionListener）
@Override
public void onLocalRecordBegin(int errCode, String storagePath) { }

@Override
public void onLocalRecording(long durationMs, String storagePath) {
}

@Override
public void onLocalRecordComplete(int errCode, String storagePath) {
    if (errCode == 0) {
        // 录制成功，storagePath 为最终文件路径
    }
}
```

iOS

```
// 开始录制
TXIoTLocalRecordingParams *params = [[TXIoTLocalRecordingParams
alloc] init];
```

```
params.channelId = 0;
params.outputPath = [NSTemporaryDirectory()
stringByAppendingPathComponent:@"record.mp4"];
[session startLocalRecording:params];

// 结束录制
[session stopLocalRecording];

// 录制状态回调（TXIoTMonitorSessionDelegate）
- (void)onLocalRecordBegin:(int)errCode storagePath:(NSString
*)storagePath { }

- (void)onLocalRecording:(int64_t)durationMs storagePath:(NSString
*)storagePath { }

- (void)onLocalRecordComplete:(int)errCode storagePath:(NSString
*)storagePath {
    if (errCode == 0) {
        // 录制成功，storagePath 为最终文件路径
    }
}
```

错误处理

录制与截图涉及本地文件写入，以下错误码在录制或截图过程中可能返回，完整错误码定义见 TXIoTErrorCode 。

Android

错误码	说明	建议处理方式
`ERR_RECORD_DURATION_TOO_SHORT`	录像时长过短。	过短的录像可忽略。
`ERR_FILE_PATH_UNWRITABLE`	文件路径不可写。	确认保存目录存在且有写入权限。
`ERR_FILE_PATH_ALREADY_EXIST`	文件已存在。	使用新的文件路径，避免覆盖已有文

		件。

iOS

错误码	数值	说明	建议处理方式
`TXIoTErrorCodeRecordDurationTooShort`	-2007	录像时长过短。	过短的录像可忽略。
`TXIoTErrorCodeFilePathUnwritable`	-2008	文件路径不可写。	确认保存目录存在且有写入权限。
`TXIoTErrorCodeFilePathAlreadyExist`	-2009	文件已存在。	使用新的文件路径，避免覆盖已有文件。

常见问题

现象	排查建议
`onSnapshotComplete` 返回非 0 错误码。	确认调用 takeSnapshot 时该通道正在拉流（ startRemoteView 已成功且画面已渲染），无画面时无法截图。
录制文件无法生成或打开。	确认 `outputPath` 所在目录已存在且应用有写入权限。
录制过程中应用退到后台后录制中断。	本地录制依赖持续拉流，应用退后台被系统回收会中断录制，建议在退后台前自动停止录制。