ALIBABA CLOUD

阿里云

物联网智能视频服务
设备端开发指南

文档版本：20220406

阿里云

法律声明

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

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

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

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

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

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

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

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

目录

1.概述 06
2.Linux LinkVisual SDK 08
2.1. SDK Demo 08
2.2. SDK获取 11
2.3. 使用流程 11
2.4. 接口列表 14
2.5. 生命周期 16
2.6. 消息交互 17
2.7. 图片功能 17
2.8. 视频播放 19
2.8.1. 使用说明 19
2.8.2. 直播和云端录像存储 20
2.8.3. 本地录像播放 22
2.8.4. 预录功能 25
2.9. 语音对讲 26
2.10. 其他接口 27
2.11. 接口详情 28
2.11.1. 生命周期 28
2.11.1.1. lv_init 28
2.11.1.2. lv_destroy 31
2.11.2. 消息交互 31
2.11.2.1. lv_message_adapter 31
2.11.2.2. lv_message_publish_cb 33
2.11.3. 音视频播放 34
2.11.3.1. lv_start_push_streaming_cb 34
2.11.3.2. lv_stop_push_streaming_cb 35

2.11.3.3. lv_on_push_streaming_cmd_cb 36
2.11.3.4. lv_on_push_streaming_data_cb 37
2.11.3.5. lv_stream_send_config 38
2.11.3.6. lv_stream_send_media 39
2.11.3.7. lv_stream_send_cmd 40
2.11.3.8. lv_query_record_cb 40
2.11.3.9. lv_post_query_record 41
2.11.4. 图片功能 41
2.11.4.1. lv_trigger_picture_cb 41
2.11.4.2. lv_post_trigger_picture 42
2.11.4.3. lv_post_alarm_image 42
2.11.4.4. lv_post_intelligent_alarm 43
2.11.5. 其他接口 43
2.11.5.1. lv_control 43
2.11.5.2. lv_cloud_event_cb 44
2.11.5.3. lv_feature_check_cb 45
3.Andriod LinkVisual SDK 47
3.1. SDK Demo 47
3.2. 获取SDK 47
3.3. 初始化 48
3.4. 直播功能 51
3.5. 录像播放 58
3.6. 语音对讲 65

1.概述

物联网视频服务为您提供LinkVisual SDK，供您将IPC设备接入视频型实例。本文介绍LinkVisual SDK的基本信息，以及设备端的开发流程。

背景信息

LinkVisual SDK依赖于设备接入Link SDK。Link SDK提供物联网通道能力，LinkVisual SDK通过响应Link SDK的控制消息来处理流媒体业务。

LinkVisual SDK和Link SDK提供的功能如下：

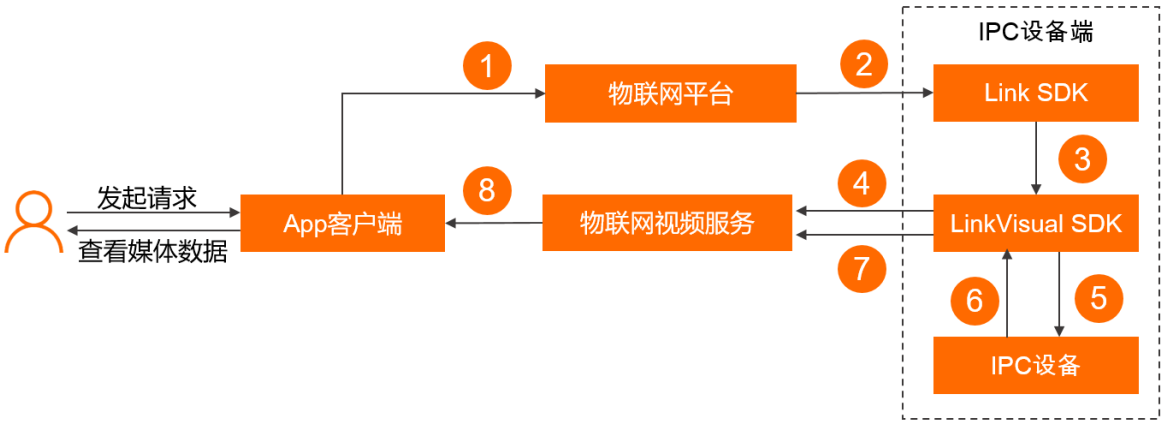
- LinkVisual SDK：音视频流的通道能力，视频直播、视频文件播放、图片上传、智能告警事件上报、PTZ控制等。
- Link SDK：物联网控制通道能力，包括长连接、消息通知、事件上报等。Link SDK的详细信息，请参考Link SDK文档。

SDK类型

物联网视频服务提供Linux和Android版本的设备端LinkVisual SDK。您可根据使用场景，选择需要的SDK。

- Linux LinkVisual SDK
- Android LinkVisual SDK

数据流转过程示意图



图号	描述
①	App客户端转交用户请求至物联网平台。
②	物联网平台建立信令通道，接收用户请求。
③	转交用户请求至LinkVisual SDK。

图号	描述
④	建立媒体通道，生成媒体文件的链接。
⑤	通知IPC设备发送媒体数据。
⑥	IPC设备发送媒体数据至LinkVisual SDK。
⑦	LinkVisual SDK发送媒体数据至物联网视频服务。
⑧	物联网视频服务的视频型实例，将媒体数据发送给App客户端。

2.Linux LinkVisual SDK

2.1. SDK Demo

视频型实例提供Ubuntu和Docker两种类型的设备端SDK Demo，供您快速体验视频型实例功能。

前提条件

已创建产品和设备，具体操作，请参见设备接入。

下载SDK Demo

您可根据需要，选择适合的Demo进行试用。

> **注意**
>
> 下载LinkVisual SDK Demo表示您遵守软件许可协议，请您仔细阅读软件许可协议。

名称	描述	下载地址
Ubuntu Demo	必须在Ubuntu18.04版本下运行。推荐您安装与阿里云已测试版本一致的发行版，以免碰到兼容性方面的问题。	LinkVisual SDK Ubuntu Demo
Docker Demo	跨平台能力好，您可在Windows、Mac或Linux等系统上安装Docker，即可使用该Demo。	LinkVisual SDK Docker Demo

Ubuntu Demo操作步骤

1. 下载LinkVisual SDK Ubuntu Demo。

   > **说明**
   >
   > 下载后的Demo文件名为 `lv_2.1.2-lk_2.3.0-ubuntu.tar.gz` 。

2. 执行命令 `tar -xf lv_2.1.2-lk_2.3.0-ubuntu.tar.gz` 解压文件夹。

3. 执行命令 `cd lv_2.1.2-lk_2.3.0-ubuntu` 打开文件夹，文件夹的目录结构如下。

```
1.jpg
avc_aac_1000k
avc_aac_1000k.index
avc_aac_1000k.meta
avc_aac_500k
avc_aac_500k.index
avc_aac_500k.meta
link_visual_ipc
```

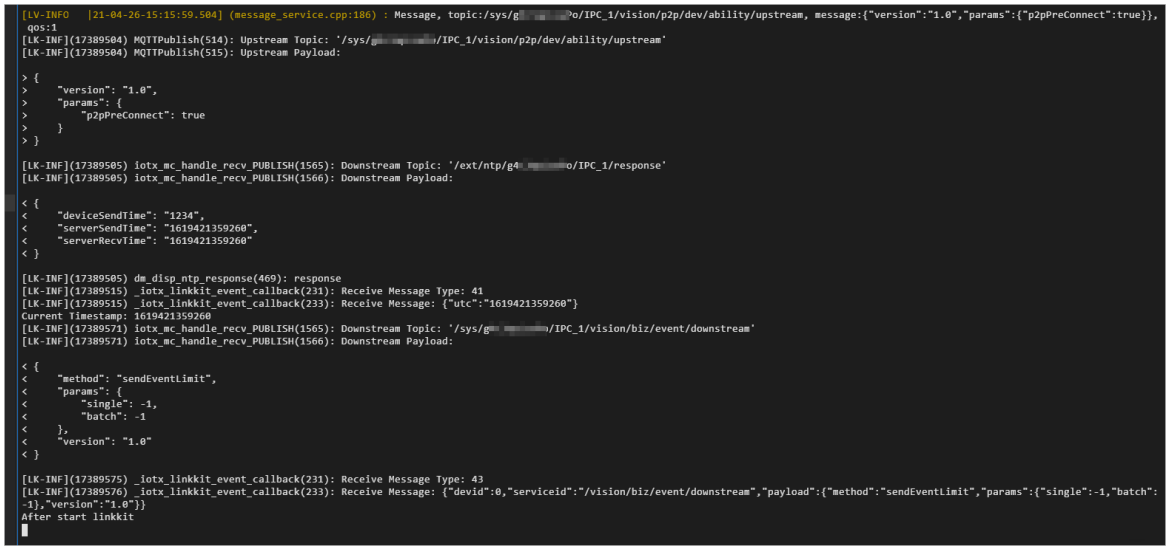
4. 执行命令

   `./link_visual_ipc -p ${YourProductKey} -n ${YourDeviceName} -s ${YourDeviceSecret}` 使设备在线并接入视频服务。

   其中，变量参数说明如下：

参数	对应设备证书的参数	示例	说明
${YourProductKey}	ProductKey	g18l****	您添加设备后，保存的设备证书信息，详细信息，请参见执行结果。您也可在物联网平台控制台中设备的**设备详情**页面查看。
${YourDeviceName}	DeviceName	Device1	
${YourDeviceSecret}	DeviceSecret	b2e6e4f1058d8****	

   显示如下日志，表示执行成功。



5. 按照如下步骤，在物联网平台控制台的**设备详情**页面，查看设备状态。

   i. 登录物联网平台控制台。

   ii. 在**实例概览**页面，找到对应的实例，单击实例进入**实例详情**页面。

   iii. 在左侧导航栏，选择**设备管理** > **设备**。

   iv. 在**设备**页面，单击设备名称，进入**设备详情**页面。

v. 单击**物模型数据**页签，可在**运行状态**页签下查看物模型数据；几分钟后，单击**事件管理**页签，可查看生成的智能告警事件。

Docker Demo操作步骤

1. 安装Docker。

   > **说明**
   >
   > 安装Docker的具体操作请您自行完成。

2. 下载LinkVisual SDK Docker Demo。

   > **说明**
   >
   > 下载后的Demo文件名为 `lv_2.1.2-lk_2.3.0-docker.tar.gz` 。

3. 执行命令 `docker load -i lv_2.1.2-lk_2.3.0-docker.tar.gz` 导入Docker镜像。

   成功后显示 `Loaded image: ubuntu:lv_2.1.2-lk_2.3.0` 。

4. 执行命令 `docker run -it --rm ubuntu:lv_2.1.2-lk_2.3.0 bash` 运行镜像，进入到镜像生成的容器中。

5. 执行命令 `cd /root` 进入/root目录。

6. 依次执行如下命令解压并打开Demo文件。

   ```
   tar -xvf lv_2.1.2-lk_2.3.0-ubuntu.tar.gz
   cd lv_2.1.2-lk_2.3.0-ubuntu
   ```

7. 执行 `ls` 命令，Demo文件内容如下。

   ```
   1.jpg
   avc_aac_1000k
   avc_aac_1000k.index
   avc_aac_1000k.meta
   avc_aac_500k
   avc_aac_500k.index
   avc_aac_500k.meta
   link_visual_ipc
   ```

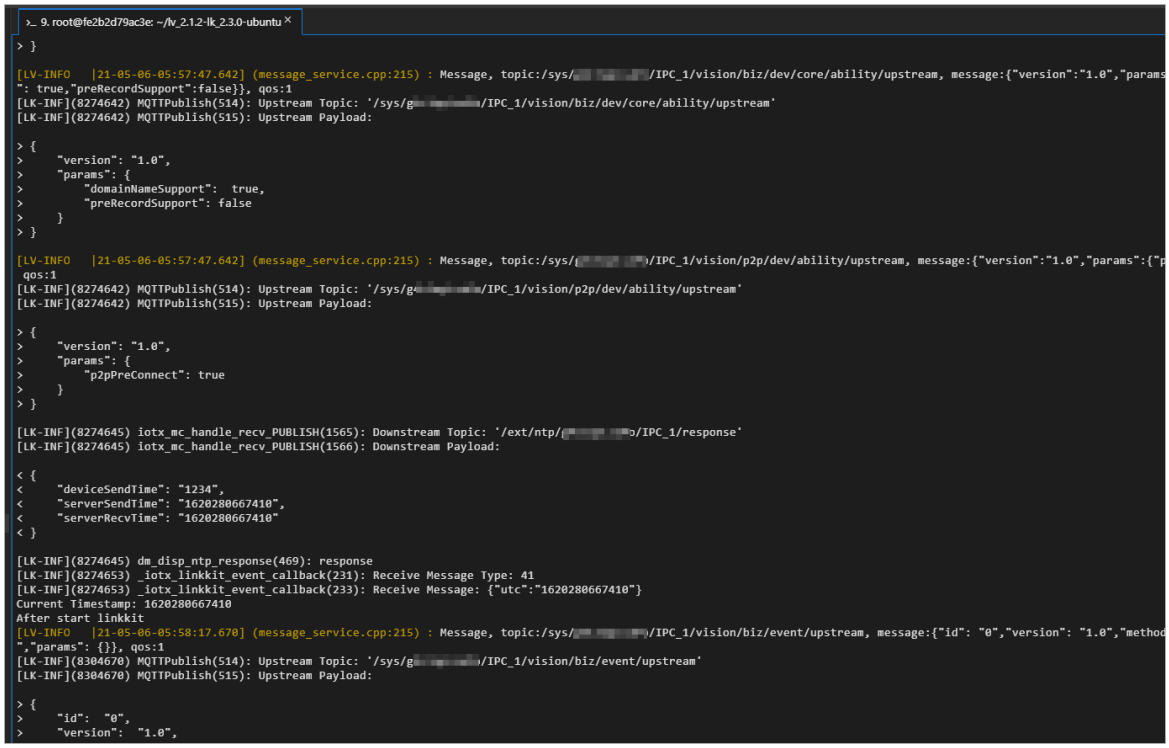
8. 执行命令

   `./link_visual_ipc -p ${YourProductKey} -n ${YourDeviceName} -s ${YourDeviceSecret}` 使设备在线并接入视频服务。

   其中，变量参数说明如下：

参数	对应设备证书的参数	示例	说明
${YourProductKey}	ProductKey	g18l****	您添加设备后，保存的设备证书信息，详细信息，请参见执行结果。您也可在物联网平台控制台中设备的**设备详情**页面查看。
${YourDeviceName}	DeviceName	Device1	
${YourDeviceSecret}	DeviceSecret	b2e6e4f1058d8****	

显示如下日志，表示执行成功。



9. 按照如下步骤，在物联网平台控制台的**设备详情**页面，查看设备状态。
   i. 登录物联网平台控制台。
   ii. 在**实例概览**页面，找到对应的实例，单击实例进入**实例详情**页面。
   iii. 在左侧导航栏，选择**设备管理** > **设备**。
   iv. 在**设备**页面，单击设备名称，进入**设备详情**页面。
   v. 约十分钟后，单击**物模型数据**页签，然后单击**事件管理**页签，可查看生成的智能告警事件。

2.2. SDK获取

LinkVisual设备端SDK以静态库的形式提供，我们可以编译SDK支持不同芯片平台的接入。

若需要获取LinkVisual设备端SDK，您可以点击咨询。

2.3. 使用流程

获取SDK后，您可参考本文，使用LinkVisual SDK将您的IPC设备接入视频型实例。

环境要求

LinkVisual SDK的资源占用和平台支持如下所示：

- 资源占用：
  - RAM：1 MB的码流，预计占用500 KB的RAM内存。
  - ROM：占用1.4 MB的ROM内存。
- 平台支持：支持在C++11标准的Linux平台中使用，且需确保gcc为4.8.1以上版本。

前提条件

- 已创建产品和设备，具体操作，请参见设备接入。
- 已获取LinkVisual SDK，详细信息，请参见获取SDK。
- 本文使用Linux下的设备端C语言SDK。该SDK的开发编译环境推荐使用Ubuntu18.04。

  SDK的开发编译环境会用到cmake、git等软件，可以使用如下命令行安装。

```
sudo apt-get install -y build-essential make git gcc g++ cmake tree
```

背景信息

- LinkVisual SDK软件包中包含Link SDK，您需要依次编译Link SDK和LinkVisual SDK，使IPC设备接入视频型实例。
- 您可参考下文中的操作步骤，实现编译过程。若您自行实现SDK集成编译，编译前请参见如下说明：
  - 编译选项中，请添加 `-std=c++11` 。
  - 链接时，库的连接顺序依次为：*link_visual_device*、*iot_sdk cjson*、*iot_hal*、*iot_tls*、*pthread*、*rt*。
  - 链接时，请在链接选项中添加 `-lstdc++` 。

操作步骤

1. 使用命令 `tar -xf lv_2.1.2-lk_2.3.0-xxx-xxx.tar.gz` 解压LinkVisual SDK压缩包。

   > **说明**
   >
   > 压缩包文件名含有版本等可变信息，执行命令时以实际压缩包名称为准。

2. 进入 `lv_2.1.2-lk_2.3.0-xxx-xxx` 文件的third_party目录，解压*cJSON-1.7.7*文件，并进入解压后的文件夹。

```
cd third_party
# 解压代码压缩包
tar -xf cJSON-1.7.7.tar.gz
cd cJSON-1.7.7
```

3. 打开*Makefile*文件，在文件最开始加入工具链的声明。

> **说明**
>
> 请您根据实际情况替换成对应的交叉编译工具链。

```
CC = arm-linux-gcc
LD = arm-linux-ld
AR = arm-linux-ar
```

4. 执行 `make` 命令编译文件后，确认生成*libcjson.a*和c*JSON.h*文件。

```
make
# 确认libcjosn.a和相关头文件已存在
ls lib*.a
ls *.h
```

5. 编译Link SDK。
   i. 进入*linkkit*目录，解压*linkkit-sdk-c*，然后进入解压后的文件夹。

   ```
   cd linkkit
   tar -xf linkkit-sdk-c.tar.gz #解压代码压缩包
   cd linkkit-sdk-c
   ```

   ii. 进入*src/board*目录，打开*config.ubuntu.x86*文件，在文件最后加上 `CROSS_PREFIX:=交叉编译工具链路径前缀` ，例如 `CROSS_PREFIX:=arm-linux-` 。

   > **说明**
   >
   > 请您根据实际情况，替换成对应的交叉编译工具链。

   iii. 执行 `make reconfig` ，然后输入config.ubuntu.x86对应的数字，一般为数字6。

   iv. 编译并确认生成了库文件*libiot_tls.a*、*libiot_sdk.a*和*libiot_hal.a*。

   ```
   make
   # 查询是否有库文件libiot_tls.a、libiot_sdk.a和libiot_hal.a
   ls output/lib/*.a
   # 确认有iot_import.h等头文件
   tree include
   ```

6. 整体编译。
   i. 返回LinkVisual SDK文件的根目录。
   ii. 打开*CMakeLists.txt*文件，在 `TOOLCHAINS_PREFIX` 参数后填写您的交叉编译工具链的前缀，例如 `arm-linux-` 。

   ```
   set(TOOLCHAINS_PREFIX "arm-linux-" CACHE STRING "set the toolchain")
   ```

iii. 依次执行命令编译文件。

```
# 建立一个build文件夹，用于归类编译产物
mkdir -p build
# 进入build目录，使用根目录的CMakeLists.txt进行cmake
cd build
cmake ..
# 编译并安装运行所需相关文件
make
make install
```

7. 执行命令 `./link_visual_ipc -p ${YourProductKey} -n ${YourDeviceName} -s ${YourDeviceSecret}` 运行设备Demo。

   其中，变量参数说明如下：

参数	对应设备证书的参数	示例	说明
${YourProductKey}	ProductKey	g18l****	您添加设备后，保存的设备证书信息，详细信息，请参见执行结果。您也可在物联网平台控制台中设备的**设备详情**页面查看。
${YourDeviceName}	DeviceName	Device1	
${YourDeviceSecret}	DeviceSecret	b2e6e4f1058d8****	

8. 按照如下步骤，在物联网平台控制台的**设备详情**页面，查看设备状态。
   i. 登录物联网平台控制台。
   ii. 在**实例概览**页面，找到对应的实例，单击实例进入**实例详情**页面。
   iii. 在左侧导航栏，选择**设备管理** > **设备**。
   iv. 在**设备**页面，单击设备名称，进入**设备详情**页面。
   v. 单击**物模型数据**页签，查看设备数据。
9. 若设备在线，且**物模型数据**页签下可查看设备的相关数据，则设备接入完成。

2.4. 接口列表

本文介绍LinkVisual SDK提供的接口。

生命周期

接口	描述
lv_init	初始化SDK。

接口	描述
lv_destroy	销毁SDK。

消息交互

接口	描述
lv_message_adapter	在该函数回调中，向LinkVisual SDK注入所有物联网平台消息。
lv_message_publish_cb	在该函数回调中，发送LinkVisual SDK所需要发送的物联网消息。

音视频播放

接口	描述
lv_start_push_streaming_cb	开启通知服务。
lv_stop_push_streaming_cb	发送直播或本地录像播放的链路断开通知。
lv_stream_send_config	推送音视频配置参数。
lv_stream_send_media	推送音视频数据。
lv_stream_send_cmd	发送控制视频播放的相关命令。
lv_on_push_streaming_cmd_cb	在建立直播链接或本地录像播放链接期间，发送控制命令。
lv_on_push_streaming_data_cb	在建立语音对讲链接期间，接收播放端的音频数据。
lv_query_record_cb	查询本地IPC存储卡或者NVR中的视频文件。
lv_post_query_record	发送本地IPC设备存储卡或者NVR设备中，录像文件的查询结果。

图片功能

接口	描述
lv_post_alarm_image	IPC设备主动发送侦测报警事件并上传图片。
lv_post_intelligent_alarm	获取智能告警事件并上传图片。
lv_trigger_picture_cb	通知IPC设备开始拍摄图片并上传。
lv_post_trigger_picture	发送IPC设备拍摄图片的结果。

其他接口

接口	描述
lv_control	动态调节SDK功能。
lv_cloud_event_cb	云端事件通知。
lv_feature_check_cb	确认功能是否实现。

2.5. 生命周期

使用LinkVisual SDK时，需要先配置生命周期相关接口进行初始化操作。本文介绍生命周期的相关接口，以及接口使用说明。

相关接口

功能描述	接口
初始化SDK	lv_init
销毁SDK	lv_destroy

使用说明

- 建议您在成功调用IOT_Linkkit_Connect前调用lv_init。
- 设备是否入网不影响调用lv_init。

- lv_init注册了大量的回调函数，调用lv_init前，请检查回调函数中的代码是否会导致长时间或者永久阻塞。回调函数共用一个消息队列线程，一个回调的永久阻塞将导致所有回调无法抛出。
- lv_init定义了日志类型log_level，建议您在对接初期，将日志类型设置为LV_LOG_DEBUG。
- lv_init会打印版本号信息，请您在对接技术支持人员时，注明此版本信息，用于追溯问题。
- 若调用lv_init失败，请先检查请求参数是否正确。

2.6. 消息交互

信息交互接口用于LinkVisual SDK和Link SDK进行信息交互。本文介绍信息交互的相关接口，及接口的使用说明。

相关接口

功能描述	接口
注入物联网平台SDK的消息	lv_message_adapter
抛出物联网平台SDK的消息	lv_message_publish_cb

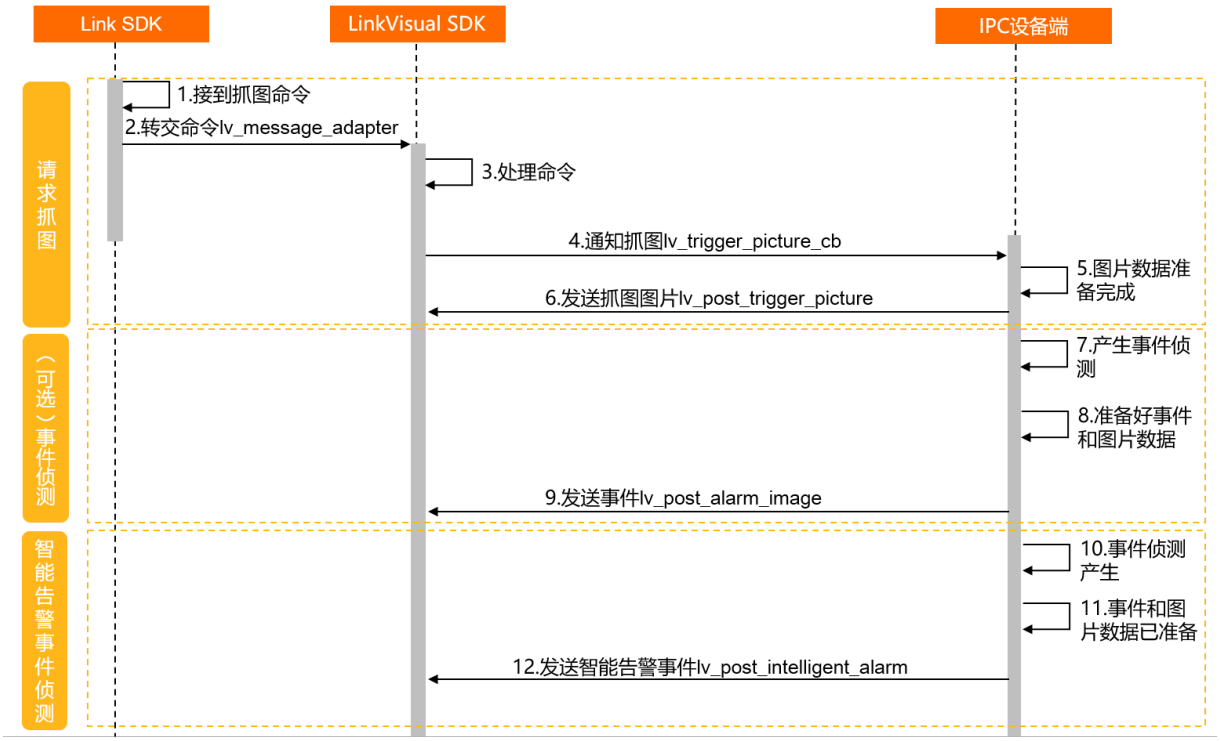
使用说明

- 请您直接使用LinkVisual SDK中的相关源码，避免因自定义消息或者物模型消息缺失，影响SDK的功能。
- IOT_RegisterCallback注册的回调函数，共用一个消息队列线程，请您认真检查回调函数中的代码，不要在回调中进行过于耗时的操作，而引起长时间或者永久阻塞。否则可能会因永久阻塞导致所有回调无法抛出。
- IOT_RegisterCallback注册的ITE_SERVICE_REQUST回调函数中，参数id、serviceid和request指向同一个字符串的不同位置，因此比较serviceid时，需要附带serviceid_len参数。
- 您可登录物联网平台控制台，使用**在线调试**功能，从云端下发测试数据至设备端，进行设备功能调试。详细信息，请参考在线调试。

2.7. 图片功能

LinkVisual SDK提供图片功能，当触发告警事件或您主动触发抓图时，IPC设备会拍摄图片并上传至云端。本文为您介绍使用图片功能时，使用的接口以及数据流转的过程。

数据流转流程图



相关接口

阶段	功能描述	相关接口
请求抓图	消息转交命令	lv_message_adapter
	通知抓图	lv_trigger_picture_cb
	发送抓图图片	lv_post_trigger_picture
（可选）事件侦测	发送侦测报警事件	lv_post_alarm_image
智能告警事件侦测	发送智能告警事件	lv_post_intelligent_alarm

> **说明**
>
> lv_post_intelligent_alarm接口是lv_post_alarm_image接口的升级版本。若初次使用该功能，请您使用lv_post_intelligent_alarm接口。

使用说明

- 最大支持上传大小为5 MB的图片。
- 上传图片的最小间隔为1秒，若频繁触发，图片将会被丢弃。

- 上传图片时SDK会拷贝一份图片，并形成待发送的图片队列。图片队列的长度默认为5，网络情况较差时，可能因图片满队列而导致后生成的图片被丢弃，但图片关联的事件会被保留。
- SDK不限制上传图片的格式，只需从物联网平台拉取图片的设备（例如您的手机），支持解析该图片格式即可。推荐您使用常见的图片格式，如JPEG。

2.8. 视频播放

2.8.1. 使用说明

LinkVisual SDK提供了视频播放功能，开发视频播放功能前，请参考本文，了解开发过程中的相关信息。

视频播放包括以下几种形式，其实现过程可参考相应的文档：

- 直播
- 云端录像播放
- 本地录像播放

使用说明

在设备端实现视频播放的功能时，请参考以下要求，完成设备端开发。

- 强制I帧命令发出时：
  - 请在300毫秒内产生I帧。
  - 请重新调用lv_stream_send_config发送音视频流配置。
- 进行本地录像播放的定位操作后：
  - 应该尽快发出I帧，确保尽快显示定位后的数据。
  - SDK在定位操作后，直至I帧到达前，不再接收音视频的P帧数据。
- 视频播放时对H.264和H.265的帧结构有如下要求，您可参考下面的代码，打印I帧的前256个字节查看帧结构。

```
for (int i = 0; i < ((buffer_size > 256)?256:buffer_size); i++) {
 printf("%02x ", buffer[i]);
 if ((i + 1) % 30 == 0) {
  printf("\n");
 }
}
printf("\n");
```

- H.264的I帧格式要求为 `帧分隔符+SPS+帧分隔符+PPS+帧分隔符+IDR` 。

  以下图为例，帧分隔符为0x000001或者0x00000001；序列参数集SPS（Sequence Parameter Set）起始数据为0x67；图像参数集PPS（Picture Parameter Set）起始数据为0x68；即时解码器刷新IDR（Instantaneous Decoding Refresh）起始数据为0x65。

```
00 00 01 67 42 c0 1e d9 00 a0 2f f9 3f f2 57 92 58 01 00 00 03 00 01 00 00 03 00 32 0f 16
2e 48 00 00 00 01 68 cb 8c b2 00 00 01 65 88 84 1f c9 81 0b 40 00 24 c7 19 c5 87 40 65 89
c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 ff ff
77 c2 a1 6e 00 16 c4 d5 6f e8 1b 0d 2c af b8 00 d1 72 9b 8d 81 28 7d 2c 8c 18 5d 50 9d 09
8e 79 94 b4 d8 9d e9 7e 0b ad 99 ca 06 a7 ff c7 84 56 15 91 7b 50 01 9f a0 8a f2 75 64 67
ab c2 b2 11 61 6f 23 00 03 4a d5 93 e4 1c 11 8b 2f f8 23 0d 26 cf d4 01 5a 2c c2 67 23 bd
1d 0e ff bf ff f0 42 5e 00 09 9a 88 33 36 eb 90 00 0f e0 33 ff 80 60 b0 42 4e 00 13 59 0a
ef 4a fc 89 bf ff 80 42 bc 30 00 4c e9 0c 89 ba e4 00 1f 40 8c 1a ff f0 f5 85 78 00 73 93
75 f4 bc fc 00 0f 17 21 9c e2 40 7f cc 3c c0
```

- H.265的I帧格式要求为 `帧分隔符+VPS+帧分隔符+SPS+帧分隔符+PPS+帧分隔符+IDR` 。

  以下图为例，帧分隔符为0x000001或者0x00000001；视频参数集VPS（Video Parameter Set）起始数据为0x40；SPS起始数据为0x42；PPS起始数据为0x44；IDR起始数据为0x26。

```
00 00 00 01 40 01 0c 01 ff ff 01 40 00 00 03 00 80 00 00 03 00 00 03 00 7b ac 09 00 00 00 00 01
42 01 01 01 40 00 00 03 00 80 00 00 03 00 00 03 00 7b a0 03 c0 80 10 e5 96 b9 2c ad 9a e5 51 36
00 80 00 00 00 00 00 01 44 01 c0 e3 0f 03 32 40 00 00 00 00 01 26 01 af 13 80 5b ff ff ff a9 59
16 ca fe 88 26 26 11 cc 9a 40 61 2f 2d 47 37 39 09 25 e4 6d a2 96 33 cb 99 ff c6 74 73 cd 15 0b
98 45 db fe 98 68 e2 80 bd f6 99 b9 37 c0 a3 5b 9f 38 70 82 8a 20 b8 44 a6 d7 35 35 5c 6e ed ce
25 28 10 00 88 21 0c e0 4b fd aa 47 9b a4 e7 ad 27 f0 f5 83 e3 a6 6e 89 2b 2a 7b 18 65 1d 4a 05
ff a1 fd d3 44 fe c3 34 ad ae a5 1a b5 a2 61 83 c3 ca 6a 1a c0 7b 08 44 42 f9 bf b7 ec 88 11 01
39 33 1e 96 b9 2d 91 72 63 e0 d5 e2 af a0 82 c0 46 b7 bd 43 cf f5 36 b5 f1 09 4f b6 2e e9 e3 00
```

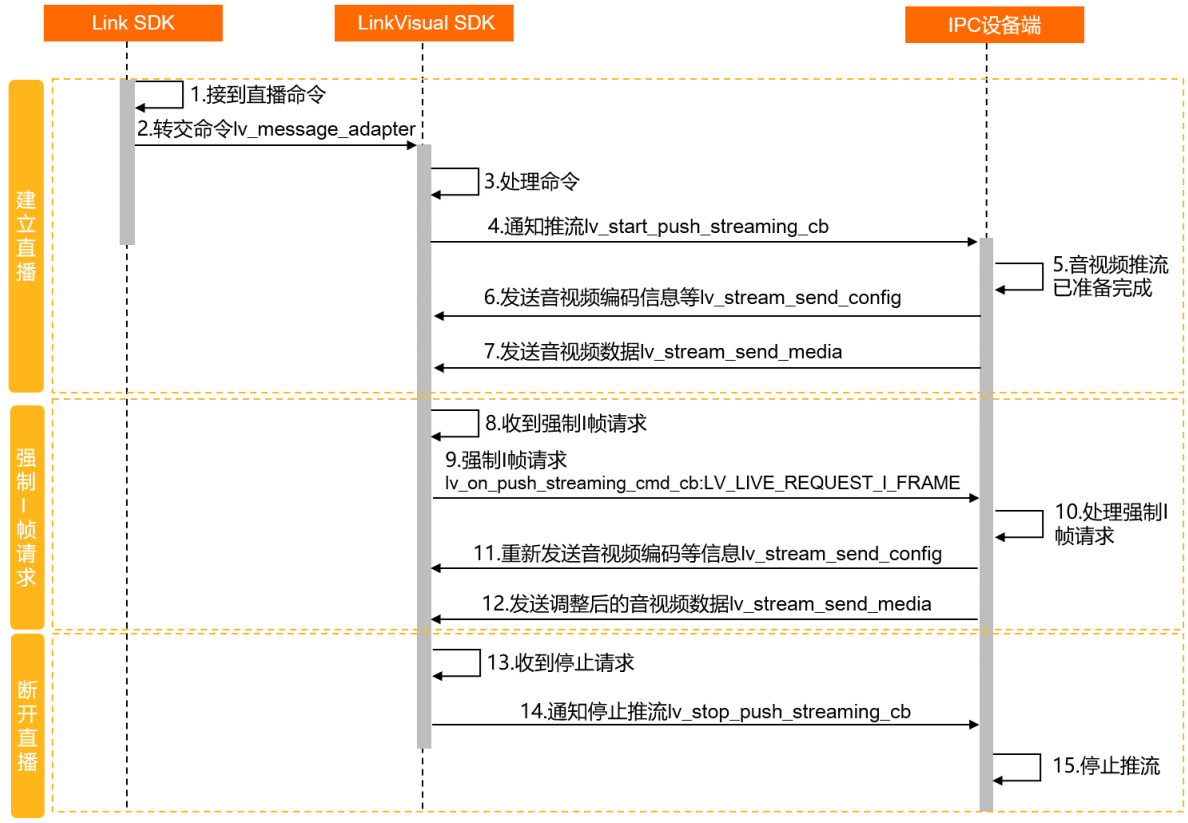
- 同一路视频流切换视频码流时，如主码流切换为子码流、修改码流分辨率、H.264和H.265互相切换等，请确保切换后的第一帧为I帧，否则可能会引发花屏等现象。

2.8.2. 直播和云端录像存储

LinkVisual SDK提供直播和云端录像存储功能，本文介绍使用直播和云端录像存储功能时，需要调用的接口以及数据流转的过程。

> **说明**
>
> 云端录像存储与直播的相关接口和流程相同，本文以直播为例进行介绍。

数据流转流程图



相关接口

阶段	功能描述	相关接口
建立直播	转交命令	lv_message_adapter
	通知开始推流	lv_start_push_streaming_cb
	发送音视频编码信息	lv_stream_send_config
	发送音视频数据	lv_stream_send_media
强制I帧请求	强制I帧请求	lv_on_push_streaming_cmd_cb **说明** 使用该接口中的LV_LIVE_REQUEST_I_FRAME方法。
强制I帧请求		

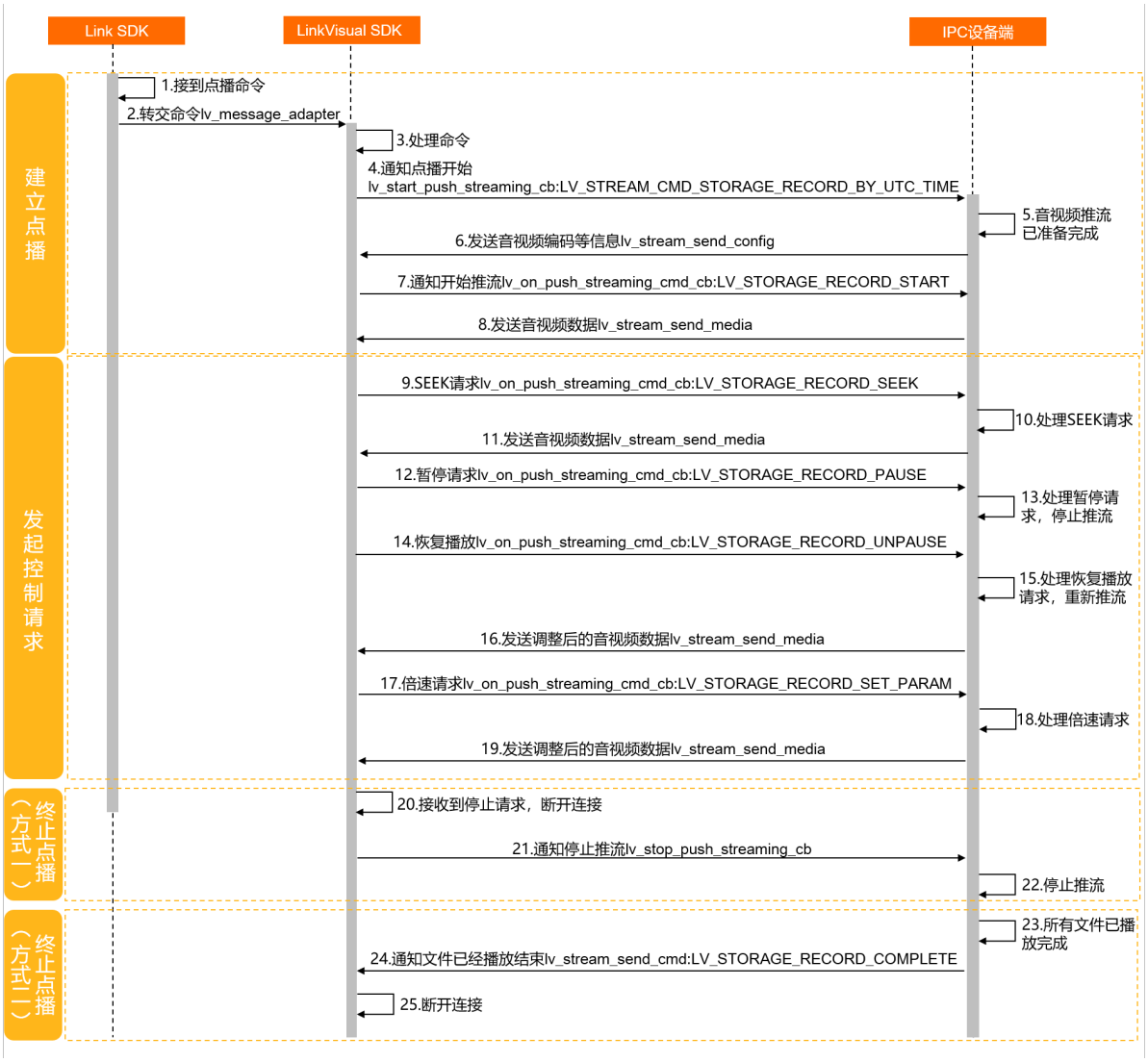
阶段	功能描述	相关接口
	发送音视频编码信息	lv_stream_send_config
	发送音视频数据	lv_stream_send_media
断开直播	通知停止推流	lv_stop_push_streaming_cb

2.8.3. 本地录像播放

LinkVisual SDK提供本地录像播放功能，本文介绍使用本地录像播放功能时，需要调用的接口以及数据流转的过程。

下文中简称本地录像播放功能为点播。

数据流转流程图



相关接口

阶段	功能描述	相关接口
建立点播	转交命令	lv_message_adapter
	点播开始	lv_start_push_streaming_cb
	发送音视频编码信息	lv_stream_send_config

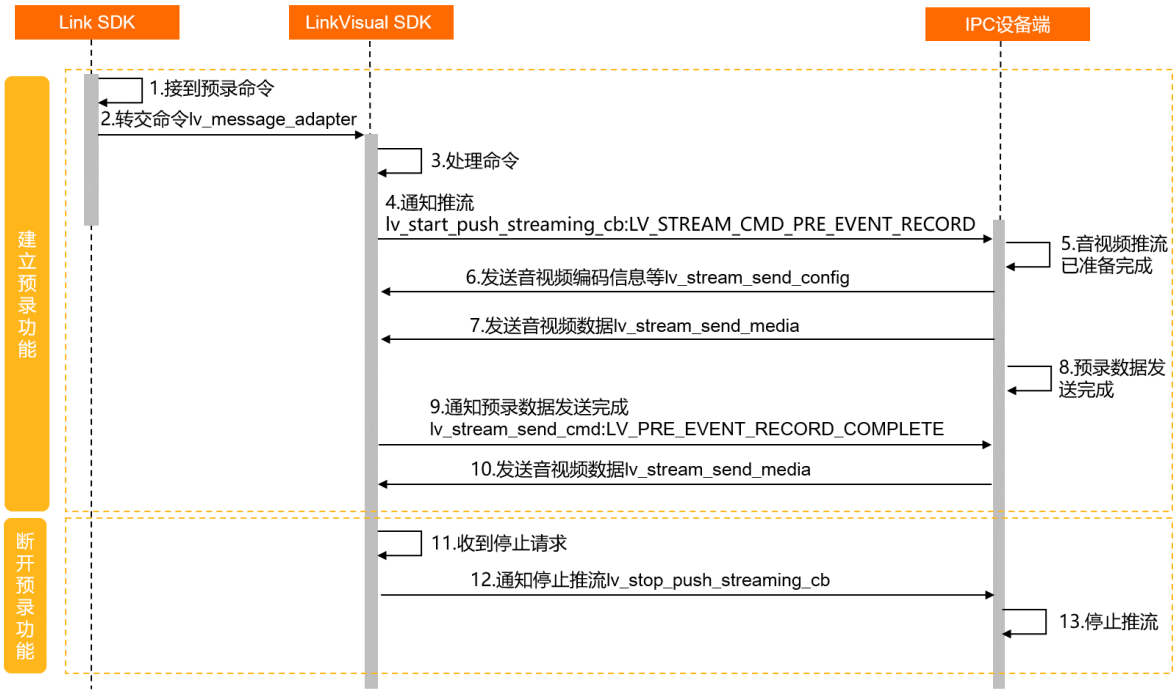
阶段	功能描述	相关接口
	通知开始推流	lv_on_push_streaming_cmd_cb **说明** 使用该接口中的LV_STORAGE_RECORD_START方法。
	发送音视频数据	lv_stream_send_media
发起控制请求	SEEK请求	lv_on_push_streaming_cmd_cb **说明** 使用该接口中的LV_STORAGE_RECORD_SEEK方法。
	暂停请求	lv_on_push_streaming_cmd_cb **说明** 使用该接口中的LV_STORAGE_RECORD_PAUSE方法。
	恢复播放请求	lv_on_push_streaming_cmd_cb **说明** 使用该接口中的LV_STORAGE_RECORD_UNPAUSE方法。
	倍速请求	lv_on_push_streaming_cmd_cb **说明** 使用该接口中的LV_STORAGE_RECORD_SET_PAPAM方法。

阶段	功能描述	相关接口
	发送音视频数据	lv_stream_send_media
终止点播（方式一）	使用远程命令结束点播	lv_stop_push_streaming_cb
终止点播（方式二）	视频播放完成触发结束点播	lv_stream_send_cmd **说明** 使用该接口中的LV_STORAGE_RECORD_COMPLETE方法。

2.8.4. 预录功能

LinkVisual SDK提供预录功能，当您的IPC设备支持预录功能时，可参考本文实现视频预录功能。本文介绍使用预录功能时，需要调用的接口以及数据流转的过程。

数据流转流程图

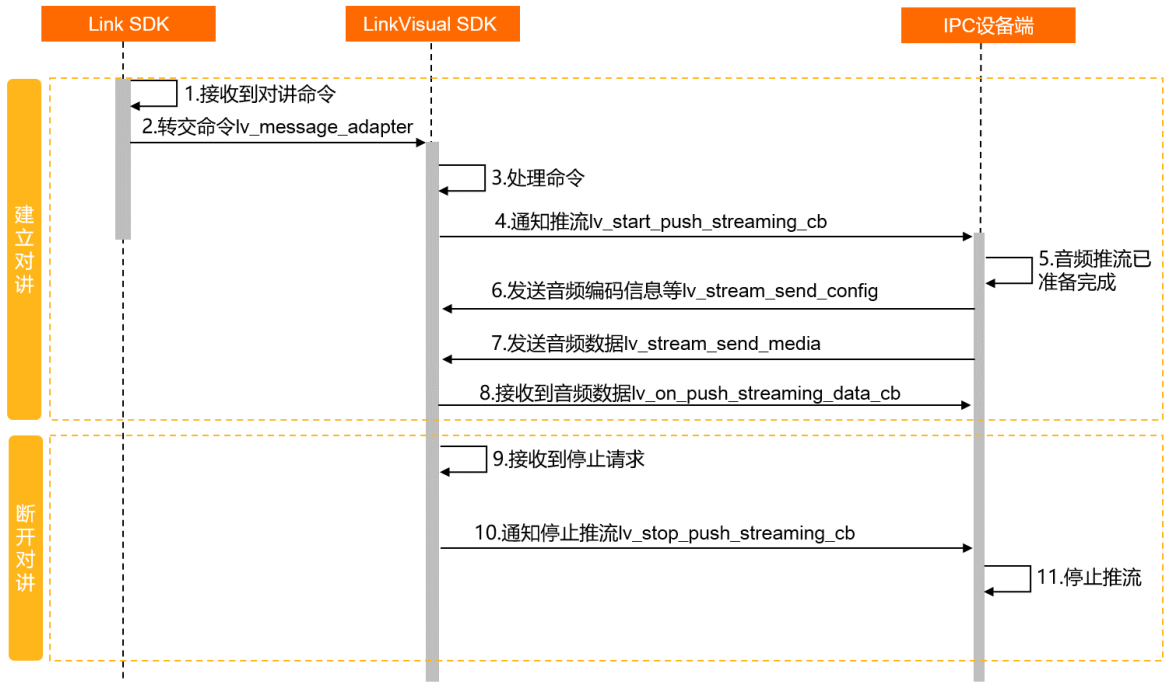


阶段	功能描述	相关接口
建立预录功能	转交命令	lv_message_adapter
	通知开始推流	lv_start_push_streaming_cb **说明** 使用接口中的LV_STREAM_CMD_PRE_EVENT_RECORD方法。
	发送音视频编码信息	lv_stream_send_config
	发送音视频数据	lv_stream_send_media
	通知预录数据发送完成	lv_stream_send_cmd **说明** 使用接口中的LV_PRE_EVENT_RECORD_COMPLETE方法。
断开预录功能	通知停止推流	lv_stop_push_streaming_cb

2.9. 语音对讲

LinkVisual SDK提供语音对讲功能，本文介绍使用语音对讲时，需要调用的接口以及数据流转的过程。

数据流转流程图



相关接口

阶段	功能描述	相关接口
建立对讲	转交命令	lv_message_adapter
	通知推流	lv_start_push_streaming_cb
	发送音频编码信息	lv_stream_send_config
	发送音频数据	lv_stream_send_media
	接收到音频数据	lv_on_push_streaming_data_cb
断开对讲	通知停止推流	lv_stop_push_streaming_cb

2.10. 其他接口

LinkVisual SDK提供了动态控制、回调云端事件、确认功能实现的相关接口。本文介绍涉及的接口和其使用说明。

相关接口

功能描述	接口
动态控制SDK功能	lv_control
回调云端事件	lv_cloud_event_cb
确认功能是否实现	lv_feature_check_cb

使用说明

- 请您直接使用LinkVisual SDK中的相关源码，避免因自定义消息或者物模型消息缺失，影响SDK的功能。
- IOT_RegisterCallback注册的回调函数，共用一个消息队列线程，请您认真检查回调函数中的代码，不要在回调中进行过于耗时的操作，而引起长时间或者永久阻塞。否则可能会因永久阻塞导致所有回调无法抛出。
- IOT_RegisterCallback注册的ITE_SERVICE_REQUST回调函数中，参数id、serviceid和request指向同一个字符串的不同位置，因此比较serviceid时，需要附带serviceid_len参数。
- 您可登录物联网平台控制台，使用**在线调试**功能，从云端下发测试数据至设备端，进行设备功能调试。详细信息，请参考在线调试。

2.11. 接口详情

2.11.1. 生命周期

2.11.1.1. lv_init

调用该接口初始化SDK。

接口详情

```
int lv_init(const lv_init_config_s *config, const lv_init_callback_s *callback, const lv_init_system_s *system);
```

接口中相关参数说明如下。

参数	类型	说明
config	lv_init_config_s *	配置参数结构体。
callback	lv_init_callback_s *	回调结构体。

参数	类型	说明
system	lv_init_system_s *	系统参数结构体。

示例代码

> **说明**
>
> 示例代码仅供参考，完整内容，请参见SDK中的实际代码。

```
//新建一个配置结构体，并置空
lv_init_config_s config;
memset(&config, 0, sizeof(lv_init_config_s));
lv_init_callback_s callback;
memset(&callback, 0, sizeof(lv_init_callback_s));
lv_init_system_s system;
memset(&system, 0, sizeof(lv_init_system_s));

/* SDK的类型配置 */
config.device_type = device_type;

/* SDK的日志配置 */
config.log_level = log_level;
config.log_dest = LV_LOG_DESTINATION_STDOUT;

/* 码流路数限制 */
config.storage_record_source_num = 1;

/* 码流检查功能 */
config.stream_auto_check = 1;
config.stream_auto_save = 0;

/* das默认开启 */
config.das_close = 0;

callback.message_publish_cb = linkkit_message_publish_cb;

//音视频推流服务
callback.start_push_streaming_cb = start_push_streaming_cb;
callback.stop_push_streaming_cb = stop_push_streaming_cb;
callback.on_push_streaming_cmd_cb = on_push_streaming_cmd_cb;
callback.on_push_streaming_data_cb = on_push_streaming_data_cb;

//获取存储录像录像列表
callback.query_storage_record_cb = query_storage_record_cb;

callback.trigger_picture_cb = trigger_picture_cb;

/* 云端事件通知 */
callback.cloud_event_cb = cloud_event_cb;

callback.feature_check_cb = feature_check_cb;

//先准备好LinkVisual相关资源
int ret = lv_init(&config, &callback, &system);
if (ret < 0) {
    printf("lv_init failed, result = %d\n", ret);
    return -1;
}

return 0;
```

2.11.1.2. lv_destroy

调用该接口销毁SDK。

接口详情

```
int lv_destroy(void);
```

示例代码

> 说明
> 示例代码仅供参考，完整内容，请参见SDK中的实际代码。

```
//运行结束后
lv_destroy();
```

2.11.2. 消息交互

2.11.2.1. lv_message_adapter

调用该接口向LinkVisual SDK注入物联网平台消息。

接口详情

```
int lv_message_adapter(const lv_device_auth_s *auth, const lv_message_adapter_param_s *para
m)
```

接口中相关参数说明如下。

参数	类型	说明
auth	lv_device_auth_s	设备认证信息。
param	const lv_message_adapter_param_s*	消息结构体。

示例代码

```
//user_link_visual_handler为IOT_RegisterCallback使用ITE_LINK_VISUAL订阅的回调，
//在回调中，所有消息都使用lv_message_adapter注入
static int user_link_visual_handler(const int devid, const char *service_id,
                                    const int service_id_len, const char *payload,
                                    const int payload_len) {
    /* LinkVisual自定义的消息，直接全交由LinkVisual来处理 */
```

```
    if (payload == NULL || payload_len == 0) {
        return 0;
    }

/* 设备证书信息，其中：
product_key：在阿里云物联网平台创建的产品ProductKey。
device_name：物联网平台为设备颁发的设备名称。
device_secret：物联网平台为设备颁发的设备密钥，用于认证加密。*/

    char *product_key = NULL;
    char *device_name = NULL;
    char *device_secret= NULL;
    iotx_dm_get_triple_by_devid(devid, &product_key, &device_name, &device_secret);

    lv_device_auth_s auth;
    GetAuth(devid, &auth);

    lv_message_adapter_param_s in = {0};
    in.type = LV_MESSAGE_ADAPTER_TYPE_LINK_VISUAL;
    in.service_name = (char *)service_id;
    in.service_name_len = service_id_len;
    in.request = (char *)payload;
    in.request_len = payload_len;
    int ret = lv_message_adapter(&auth, &in);
    if (ret < 0) {
        printf("LinkVisual process service request failed, ret = %d\n", ret);
        return -1;
    }

    return 0;
}


//user_service_request_handler为IOT_RegisterCallback使用ITE_SERVICE_REQUST订阅的回调
//在回调中，满足一定规则的命令需要使用lv_message_adapter注入
static int user_service_request_handler(const int devid, const char *id, const int id_len,
                                        const char *serviceid, const int serviceid_len,
                                        const char *request, const int request_len,
                                        char **response, int *response_len) {
    printf("Service Request Received, Devid: %d, ID %.*s, Service ID: %.*s, Payload: %s\n",
             devid, id_len, id, serviceid_len, serviceid, request);

    /* 部分物模型服务消息由LinkVisual处理，部分需要自行处理。 */
    int link_visual_process = 0;
    for (unsigned int i = 0; i < sizeof(link_visual_service)/sizeof(link_visual_service[0])
; i++) {
        /* 这里需要根据字符串的长度来判断 */
        if (!strncmp(serviceid, link_visual_service[i], strlen(link_visual_service[i]))) {
            link_visual_process = 1;
            break;
        }
    }

    if (link_visual_process) {
        /* ISV将某些服务类消息交由LinkVisual来处理，不需要处理response */
```

```
        /* 设备证书信息，其中：
            product_key：在阿里云物联网平台创建的产品ProductKey。
            device_name：物联网平台为设备颁发的设备名称。
            device_secret：物联网平台为设备颁发的设备密钥，用于认证加密。 */
        char *product_key = NULL;
        char *device_name = NULL;
        char *device_secret= NULL;
        iotx_dm_get_triple_by_devid(devid, &product_key, &device_name, &device_secret);

        lv_device_auth_s auth;
        GetAuth(devid, &auth);

        lv_message_adapter_param_s in = {0};
        in.type = LV_MESSAGE_ADAPTER_TYPE_TSL_SERVICE;
        in.msg_id = (char *)id;
        in.msg_id_len = id_len;
        in.service_name = (char *)serviceid;
        in.service_name_len = serviceid_len;
        in.request = (char *)request;
        in.request_len = request_len;
        int ret = lv_message_adapter(&auth, &in);
        if (ret < 0) {
            printf("LinkVisual process service request failed, ret = %d\n", ret);
            return -1;
        }
    } else {
        /* 非LinkVisual处理的消息 */
    }

    return 0;
}
```

2.11.2.2. lv_message_publish_cb

调用该接口，在该函数回调中，发送SDK所需要发送的物联网消息。

接口详情

```
typedef int (*lv_message_publish_cb)(const lv_message_publish_param_s *message)
```

接口中相关参数说明如下。

参数	类型	说明
message	const lv_message_publish_param_s*	消息结构体。

示例代码

> 说明
>
> 示例代码仅供参考，完整内容，请参见SDK中的实际代码。

```
//Demo中定义了回调函数linkkit_message_publish_cb，作为lv_message_publish_cb的实现
callback.message_publish_cb = linkkit_message_publish_cb;

//Demo中定义了回调函数linkkit_message_publish_cb
int linkkit_message_publish_cb(const lv_message_publish_param_s *param) {
    iotx_mqtt_topic_info_t topic_msg;

    /* Initialize topic information */
    memset(&topic_msg, 0x0, sizeof(iotx_mqtt_topic_info_t));
    topic_msg.qos = param->qos;
    topic_msg.retain = 0;
    topic_msg.dup = 0;
    topic_msg.payload = param->message;
    topic_msg.payload_len = strlen(param->message);
    int rc = IOT_MQTT_Publish(NULL, param->topic, &topic_msg);
    if (rc < 0) {
        printf("Publish msg error:%d\n", rc);
        return -1;
    }

    return 0;
}
```

2.11.3. 音视频播放

2.11.3.1. lv_start_push_streaming_cb

调用该接口开启通知服务。

该接口是回调函数。用于通知直播、点播、云存储等链路已经建立成功，并附带一些配置信息。在收到此回调后，您应该根据配置信息初始化编码器，并开始推送音视频数据。

接口详情

```
typedef int (*lv_start_push_streaming_cb)(const lv_device_auth_s *auth, const lv_start_push
_stream_param_s *param);
```

接口中相关参数说明如下。

参数	类型	说明
auth	lv_device_auth_s *	设备认证信息。

参数	类型	说明
param	const lv_start_push_stream_param_s*	附加参数，例如直播的主码流或子码流信息。

示例代码

> 说明
>
> 示例代码仅供参考，完整内容，请参见SDK中的实际代码。

```
//Demo中定义了回调函数start_push_streaming_cb，作为lv_start_push_streaming_cb的实现
lv_start_push_streaming_cb = start_push_streaming_cb;

//demo中定义了回调函数startPushStreamingCallback
static int start_push_streaming_cb(const lv_device_auth_s *auth, const lv_start_push_stream
_param_s *param) {
{
    if (param->common.cmd_type == LV_STREAM_CMD_LIVE) {
       //使用lv_stream_send_media推送音视频数据;
       //实际使用中建议新建线程发送数据
        ......
        return 0;
    } else if (param->common.cmd_type == LV_STREAM_CMD_STORAGE_RECORD_BY_UTC_TIME) {
       //使用lv_stream_send_media推送音视频数据
       //实际使用中建议新建线程发送数据
       ......
       return 0;
    }

    return 0;
}
```

2.11.3.2. lv_stop_push_streaming_cb

调用该接口，发送直播或者本地录像播放链路已断开的通知。

该接口为回调函数，收到该回调后，需要停止推送音视频数据。

接口详情

```
typedef int (*lv_stop_push_streaming_cb)(const lv_device_auth_s *auth, const lv_stop_push_s
tream_param_s *param);
```

接口中相关参数说明如下。

参数	类型	说明
auth	lv_device_auth_s *	设备认证信息。
param	const lv_stop_push_stream_param_s*	附加参数，例如直播的主码流或子码流信息。

示例代码

> 说明
> 示例代码仅供参考，完整内容，请参见SDK中的实际代码。

```
//Demo中定义了回调函数stop_push_streaming_cb，作为lv_start_push_streaming_cb的实现
lv_stop_push_streaming_cb = stop_push_streaming_cb;

//Demo中定义了回调函数stopPushStreamingCallback
static int stop_push_streaming_cb(const lv_device_auth_s *auth, const lv_stop_push_stream_p
aram_s *param) {
{
    if (param->service_id == g_live_service_id) {
        //停止推直播流
    } else if (param->service_id == g_vod_service_id) {
        //停止推点播流
    }

    return 0;
}
```

2.11.3.3. lv_on_push_streaming_cmd_cb

调用该接口，在建立直播播放链接或本地录像播放链接期间，发送控制类命令，例如发送帧命令。

该接口为回调函数。

接口详情

```
typedef int (*lv_on_push_streaming_cmd_cb)(const lv_device_auth_s *auth, const lv_on_push_s
tream_cmd_param_s *param);
```

参数	类型	说明
auth	lv_device_auth_s *	设备认证信息。

参数	类型	说明
param	const lv_on_push_stream_cmd_param_s *	附加参数，如本地录像播放的暂停命令。

示例代码

> 说明
>
> 示例代码仅供参考，完整内容，请参见SDK中的实际代码。

```
//Demo中定义了回调函数stopPushStreamingCallback，作为lv_start_push_streaming_cb的实现
on_push_streaming_cmd_cb = on_push_streaming_cmd_cb;

static int on_push_streaming_cmd_cb(const lv_device_auth_s *auth, const lv_on_push_stream_cmd_param_s *param) {
    printf("on_push_streaming_cmd_cb service_id:%d, cmd:%d %d\n", param->common.service_id, param->common.cmd_type, param->seek.timestamp);

    DummyIpcMediaParam ipc_param = {0};
    ipc_param.service_id = param->common.service_id;
    if (param->common.cmd_type == LV_LIVE_REQUEST_I_FRAME) {
        //对于直播，需要强制生成一个I帧
    } else if (param->common.cmd_type == LV_STORAGE_RECORD_SEEK) {
        //本地录像播放定位到某一段
    } else if (param->common.cmd_type == LV_STORAGE_RECORD_PAUSE) {
        //本地录像播放暂停推流
    } else if (param->common.cmd_type == LV_STORAGE_RECORD_UNPAUSE) {
        //本地录像播放恢复推流
    } else if (param->common.cmd_type == LV_STORAGE_RECORD_START) {
        //本地录像播放开始推流
    } else if (param->common.cmd_type == LV_STORAGE_RECORD_SET_PARAM) {
         //本地录像播放倍速推流
    }

    return 0;
}
```

2.11.3.4. lv_on_push_streaming_data_cb

调用该接口，在建立语音对讲链接期间，接收播放端的音频数据。

该接口是回调函数。

接口详情

```
typedef int (*lv_on_push_streaming_data_cb)(const lv_device_auth_s *auth, const lv_on_push_streaming_data_param_s *param);
```

接口中相关参数说明如下。

参数	类型	说明
auth	lv_device_auth_s *	设备认证信息。
param	const lv_on_push_streaming_data_param_s *	附加参数，如音频数据、音频格式等。

示例代码

> **说明**
> 示例代码仅供参考，完整内容，请参见SDK中的实际代码。

```
//Demo中定义了回调函数on_push_streaming_data_cb，作为lv_on_push_streaming_data_cb的实现
on_push_streaming_data_cb = on_push_streaming_data_cb;

static int on_push_streaming_data_cb(const lv_device_auth_s *auth, const lv_on_push_streaming_data_param_s *param) {
 //播放语音对讲的音频
    return 0;
}
```

2.11.3.5. lv_stream_send_config

调用该接口发送音频或视频文件的配置信息。

接口详情

```
int lv_stream_send_media(int service_id, const lv_stream_send_media_param_s *param);
```

接口中相关参数说明如下。

参数	类型	说明
service_id	int	请求ID。
param	const lv_stream_send_media_param_s *	音视频数据、附加信息等。

示例代码

> 说明
>
> 示例代码仅供参考，完整内容，请参见SDK中的实际代码。

```
lv_video_param_s video_param;
lv_audio_param_s audio_param;
memset(&video_param, 0, sizeof(lv_video_param_s));
memset(&audio_param, 0, sizeof(lv_audio_param_s));
//获取video_param、audio_param的值
lv_stream_send_config_param_s config_param = {0};
config_param.audio_param = &audio_param;
config_param.video_param = &video_param;
config_param.bitrate_kbps = 1000;
lv_stream_send_config(param->common.service_id, &config_param);
```

2.11.3.6. lv_stream_send_media

调用该接口发送音频或视频的相关数据。

接口详情

```
int lv_stream_send_media(int service_id, const lv_stream_send_media_param_s *param);
```

接口中相关参数说明如下。

参数	类型	说明
service_id	int	请求ID。
param	const lv_stream_send_media_param_s *	音频数据、视频数据或附加信息。

示例代码

> 说明
>
> 示例代码仅供参考，完整内容，请参见SDK中的实际代码。

```
//Demo中定义这个函数用于回调输入的音频帧数据
void linkvisual_client_video_handler(int service_id, lv_video_format_e format, unsigned cha
r *buffer, unsigned int buffer_size,
                                     unsigned int present_time, int nal_type) {
    //printf("video service_id:%d, format:%d, present_time:%u nal_type:%d size %u\n", servi
ce_id, format, present_time, nal_type, buffer_size);
    lv_stream_send_media_param_s param = {{0}};
    param.common.type = LV_STREAM_MEDIA_VIDEO;
    param.common.p = (char *)buffer;
    param.common.len = buffer_size;
    param.common.timestamp_ms = present_time;
    param.video.format = format;
    param.video.key_frame = nal_type;
    lv_stream_send_media(service_id, &param);
}
```

2.11.3.7. lv_stream_send_cmd

调用该接口发送视频的相关命令。

接口详情

```
int lv_stream_send_cmd(int service_id, lv_push_stream_cmd_s cmd);
```

接口中相关参数说明如下。

参数	类型	说明
service_id	int	请求ID。
cmd	lv_push_stream_cmd_s	命令类型。 请参考SDK中的实际代码，根据业务需要进行配置。

示例代码

> **说明**
>
> 示例代码仅供参考，完整内容，请参见SDK中的实际代码。

```
lv_stream_send_cmd(1, LV_STORAGE_RECORD_COMPLETE);
```

2.11.3.8. lv_query_record_cb

调用该接口查询本地IPC存储卡或者NVR设备中的视频文件。

该接口是回调函数。

接口详情

```
typedef void (*lv_query_record_cb)(const lv_device_auth_s *auth, const lv_query_record_para
m_s *param);
```

接口中相关参数说明如下。

参数	类型	说明
auth	const lv_device_auth_s *	设备认证信息。
param	const lv_query_record_param_s *	查询录像文件时的参数。

2.11.3.9. lv_post_query_record

调用该接口，发送本地IPC设备存储卡或者NVR设备中录像文件的查询结果。

接口详情

```
int lv_post_query_record(int service_id, const lv_query_record_response_param_s *response);
```

接口中相关参数说明如下。

参数	类型	说明
service_id	int	请求ID。
param	const lv_query_record_response_param_s*	本地录像文件查询的结果。

2.11.4. 图片功能

2.11.4.1. lv_trigger_picture_cb

调用该接口通知IPC设备开始拍摄图片。

接口详情

```
typedef void (*lv_trigger_picture_cb)(const lv_device_auth_s *auth, const lv_trigger_pictur
e_param_s *param);
```

接口中相关参数说明如下。

参数	类型	说明
auth	const lv_device_auth_s *	设备认证信息。
param	const lv_trigger_picture_param_s *	拍摄的图片参数。

2.11.4.2. lv_post_trigger_picture

调用该接口发送IPC设备拍摄图片的结果。

接口详情

```
int lv_post_trigger_picture(int service_id, const lv_trigger_picture_response_param_s *response);
```

接口中相关参数说明如下。

参数	类型	说明
service_id	int	请求ID。
param	const lv_trigger_picture_response_param_s *	拍摄的图片数据。

2.11.4.3. lv_post_alarm_image

调用该接口，主动发送IPC设备的侦测报警事件。

> **说明**
>
> 该接口已升级为lv_post_intelligent_alarm，建议新接入用户使用升级版接口。

接口详情

```
int lv_post_alarm_image(const lv_device_auth_s *auth, const lv_alarm_event_param_s *param, int *service_id);
```

接口中相关参数说明如下。

参数	类型	说明
auth	const lv_device_auth_s *	设备认证信息。
param	const lv_alarm_event_param_s *	侦测报警事件的参数。
servicd_id	int	请求ID。

2.11.4.4. lv_post_intelligent_alarm

调用该接口，使IPC设备主动发送智能告警事件。

> 说明
>
> 该接口为lv_post_alarm_image的升级版，建议新接入用户使用该接口。

接口详情

```
int lv_post_intelligent_alarm(const lv_device_auth_s *auth, const lv_intelligent_alarm_para
m_s *param, int *service_id);
```

接口中相关参数说明如下。

参数	类型	说明
auth	const lv_device_auth_s *	设备认证信息。
param	const lv_intelligent_alarm_param_s *	智能告警事件的参数。
servicd_id	int	请求ID。

2.11.5. 其他接口

2.11.5.1. lv_control

调用该接口动态控制SDK功能。

> 注意
>
> 该接口仅供您在调试设备阶段使用。

接口详情

```
int lv_control(lv_control_type_e type, ...)
```

接口中相关参数说明如下。

参数	类型	说明
type	lv_control_type_e	控制类型。
...	无	可变长参数，具体含义与type值有关，详细内容，请参考下文的示例代码。

示例代码

> 说明
>
> 示例代码仅供参考，完整内容，请参见SDK中的实际代码。

```
# 日志级别调整
        lv_control(LV_CONTROL_LOG_LEVEL, log_level);
# 码流自检功能开关
        lv_control(LV_CONTROL_STREAM_AUTO_CHECK, check);
# 码流自动保存开关
        lv_control(LV_CONTROL_STREAM_AUTO_SAVE, save, path);
```

2.11.5.2. lv_cloud_event_cb

调用该接口，回调云智能告警事件的通知。

接口详情

```
typedef int (*lv_cloud_event_cb)(const lv_device_auth_s *auth, const lv_cloud_event_param_s *param)
```

接口中相关参数说明如下。

参数	类型	说明
auth	const lv_device_auth_s*	设备认证信息。
message	const lv_message_publish_param_s*	消息结构体。

示例代码

> 说明
> 示例代码仅供参考，完整内容，请参见SDK中的实际代码。

```
//Demo中定义了回调函数cloud_event_cb，作为lv_cloud_event_cb的实现
callback.cloud_event_cb = cloud_event_cb;

//Demo中定义了回调函数linkkit_message_publish_cb
static int cloud_event_cb(const lv_device_auth_s *auth, const lv_cloud_event_param_s *param
) {
    printf("cloud_event_cb: %d \n", param->event_type);
    if (param->event_type == LV_CLOUD_EVENT_DOWNLOAD_FILE) {
        printf("cloud_event_cb %d %s %s\n", param->file_download.file_type, param->file_dow
nload.file_name, param->file_download.url);
    } else if (param->event_type == LV_CLOUD_EVENT_UPLOAD_FILE) {
        printf("cloud_event_cb %d %s %s\n", param->file_upload.file_type, param->file_uploa
d.file_name, param->file_upload.url);
    }
    return 0;
}
```

2.11.5.3. lv_feature_check_cb

调用该接口确认功能是否实现。

接口详情

```
typedef int (*lv_feature_check_cb)(void);
```

示例代码

> 说明
> 示例代码仅供参考，完整内容，请参见SDK中的实际代码。

```
static int feature_check_cb(void) {
    /* Demo中未实现预录功能的录像文件 */
    return LV_FEATURE_CHECK_CLOSE;
}
```

3.Andriod LinkVisual SDK

3.1. SDK Demo

物联网视频服务提供Android版本的设备端SDK Demo，供您快速体验物联网视频服务功能。

前提条件

已创建产品和设备，具体操作，请参见设备接入。

SDK获取

单击Android SDK Demo进行下载。

> 注意
>
> 下载Android SDK Demo表示您遵守软件许可协议，请您仔细阅读软件许可协议。

运行Demo

将已获取的Android SDK Demo导入至您的Android开发工具，例如Android Studio，运行Demo即可试用。

3.2. 获取SDK

物联网视频服务提供Android版本的设备端LinkViusal SDK，实现视频设备的直播、录像播放、语音对讲和图片获取等功能。本文介绍SDK的开发环境以及配置步骤。

前提条件

- 视频设备已接入视频型实例，详细操作，请参见设备接入。
- 开发前的环境要求如下表所示。

类别	说明
系统版本	支持Android 4.4及以上版本。
Java版本	支持Java 1.7及以上版本。
API LEVEL	支持Android SDK API LEVEL 18及以上版本。
Android Studio版本	支持Android Studio 2.3以上版本。

配置步骤

1. 创建Android Studio工程。
2. 配置依赖。

i. 在您创建的Android Studio工程，根目录下的*build.gradle*文件中，添加Aliyun Maven仓库地址。

```
maven {
                url "http://maven.aliyun.com/nexus/content/repositories/releases"
            }
```

ii. 在*app*文件夹下的*build.gradle*文件中添加如下依赖。

```
implementation 'com.aliyun.iotx:linkvisual-ipc:1.4.4'
```

3. 混淆配置。

   在*proguard-rules.pro*文件中添加如下混淆配置。

```
# keep linkvisual

-keep class com.aliyun.iotx.linkvisualipc.** { *; }
```

3.3. 初始化

使用设备端Android版本LinkVisual SDK，您需要先初始化SDK。本文为您介绍初始化SDK的方法。

下文简称设备端Android版本LinkVisual SDK为LinkVisual SDK。

前提条件

- 已创建产品和设备，具体操作，请参见设备接入。
- 已获取LinkVisual SDK，具体操作，请参见获取SDK。

背景信息

LinkVisual SDK依赖于设备接入Link SDK提供的，设备端与云端的双向数据通道能力，进行消息监听和处理。初始化LinkVisual SDK前，需要先完成Link SDK的初始化。

- 有关LinkVisual SDK和Link SDK，在物联网视频服务场景下各自功能及其关系的详细信息，请参见设备端开发指南概述。
- 有关Link SDK的详细信息，请参见Android Link SDK。

操作步骤

步骤一：传入设备证书信息。

```
LinkKit.getInstance().init(this, params, new ILinkKitConnectListener() {
            @Override
            public void onError(AError error){
                Log.d(TAG,
                    "onError() called with: error = [" + (error == null ? "null" : (error.getCode() + error.getMsg()))
                        + "]");
            }

            @Override
            public void onInitDone(Object data){
                Log.d(TAG, "onInitDone() called with: data = [" + JSON.toJSONString(data) + "]");
                // 初始化SDK
                IPCDev.getInstance().init(context,${YourProductKey}, ${YourDeviceName}, ${YourDeviceSecret});
            }
```

其中，变量参数说明如下：

参数	对应设备证书的参数	示例	说明
${YourProductKey}	ProductKey	g18l****	您添加设备后，保存的设备证书信息，详细信息，请参见执行结果。您也可在物联网平台控制台中设备的**设备详情**页面查看。
${YourDeviceName}	DeviceName	Device1	
${YourDeviceSecret}	DeviceSecret	b2e6e4f1058d8****	

步骤二：在物模型的设备服务中，注册异步服务调用的监听器。

设备服务的详细信息，请参考设备服务调用。

```
//注册异步服务调用的监听器
LinkKit.getInstance().getDeviceThing().setServiceHandler(service.getIdentifier(),
                itResRequestHandler);
//异步服务调用监听器
private ITResRequestHandler itResRequestHandler = new ITResRequestHandler() {
        @Override
        public void onProcess(String identify, Object result, ITResResponseCallback
            itResResponseCallback) {
            Log.d(TAG,
                "ITResRequestHandler  onProcess() called with: identify = [" + identify + "
], result = ["
                        + JSON.toJSONString(result) + "], itResResponseCallback = ["
                        + itResResponseCallback + "]");
            /**
             * 添加SDK对异步服务调用的监听
             */
            IPCDev.getInstance().notifyAsyncTopicReceived(identify, result, itResResponseCa
llback);
        }

        @Override
        public void onSuccess(Object o, OutputParams outputParams) {
            Log.d(TAG,
                "onSuccess() called with: o = [" + JSON.toJSONString(o) + "], outputParams
= [" + JSON
                        .toJSONString(outputParams) + "]");
        }

        @Override
        public void onFail(Object o, ErrorInfo errorInfo) {
            Log.d(TAG, "onFail() called with: o = [" + JSON.toJSONString(o) + "], errorInfo
= [" + JSON
                    .toJSONString(errorInfo) + "]");
        }
    };
```

步骤三：注册同步服务调用的监听器。

```
/**
 * 注册同步服务调用的监听器
 */
LinkKit.getInstance().registerOnPushListener(connectNotifyListener);
//同步服务调用监听器
private IConnectNotifyListener connectNotifyListener = new IConnectNotifyListener() {
        @Override
        public void onNotify(String connectId, String topic, AMessage aMessage) {
            /**
             * 添加SDK对同步服务调用的监听
             */
            IPCDev.getInstance().notifySyncTopicReceived(connectId, topic, aMessage);

            if (CONNECT_ID.equals(connectId) && !TextUtils.isEmpty(topic) &&
                topic.startsWith("/sys/" + productKey + "/" + deviceName + "/rrpc/request")
) {
                Log.d(TAG, "IConnectNotifyListener   onNotify() called with: connectId = ["
+ connectId + "], topic = ["
                    + topic + "], aMessage = ["
                    + new String((byte[])aMessage.data) + "]");
            }
        }

        @Override
        public boolean shouldHandle(String connectId, String topic){
            return true;
        }

        @Override
        public void onConnectStateChange(String connectId, ConnectState connectState){

        }
    };
```

3.4. 直播功能

Android版本设备端LinkVisual SDK提供直播功能，本文介绍实现直播功能的过程。

下文简称设备端Android版本LinkVisual SDK为LinkVisual SDK。

前提条件

- 已创建产品和设备，具体操作，请参见设备接入。
- 已获取LinkVisual SDK，具体操作，请参见获取SDK。
- 已完成初始化LinkVisual SDK，具体操作，请参见初始化SDK。

背景信息

直播功能通过RTMP协议推流。其支持的视频编码格式和音频编码格式如下：

- 视频编码格式：H.264和H.265。
- 音频编码格式：G711a、G711u和AAC_LC。

操作步骤

步骤一：注册直播事件监听器和流错误监听器。

LinkVisual SDK收到服务端下发的开始推流指令后，通过已注册的直播流事件监听器 `OnLiveStreamListener` 通知何时开始推流、结束推流或下发强制帧等指令。

详细开发流程如下：

1. 设置直播流事件监听。

```
// 设置直播流事件监听
IPCDev.getInstance().getIpcStreamManager().setOnLiveStreamListener(MainActivity.this);
// 设置流错误监听
IPCDev.getInstance().getIpcStreamManager().setOnStreamErrorListener(MainActivity.this);
```

2. 接收服务端发送的开始推直播流请求。

```
public interface OnLiveStreamListener{
    /**
     * 收到开始推直播流请求
     *
     * @param streamId 流ID
     * @param streamType 码流类型：0为主码流，1为辅码流
     * @param preTimeInS 预先录制时间，单位秒
     */
    void onStartPushLiveStreaming(final int streamId, final int streamType, final int p
reTimeInS);

    /**
     * 收到停止推流请求
     *
     * @param streamId 流ID
     */
    void onStopPushStreaming(final int streamId);

    /**
     * 收到强制I帧请求
     * 需立即构造一个I帧并发送
     * @param streamId 流ID
     */
    void onForceIFrame(int streamId);
}
```

步骤二：处理开始直播推流请求。

1. 当服务端下发推流请求时：通过回调

```
 OnLiveStreamListener.onStartPushLiveStreaming(int streamId, int streamType, int
preTimeInS)
```

方法，通知设备端需要开始采流并推流。

2. 处理开始直播推流请求时，一般需要同时开启IPC设备和录音机进行采流。

- 若开启，请调用 `MediaCodec` 方法对IPC设备采集的数据进行H264编码，对录音机采集的数据进行音频编码，并设置对应格式的音视频参数。
- 若不开启，请跳过此步骤。

```
@Override
    public void onStartPushLiveStreaming(int streamId, int streamType, int preTimeInS
){
        this.streamId = streamId;
        try {

            // 构造视频参数
            VideoStreamParams videoStreamParams = new VideoStreamParams();
            // 直播流该参数始终为0
            videoStreamParams.setDurationInS(0);
            videoStreamParams.setVideoFormat(VideoStreamParams.VIDEO_FORMAT_H264);

            // 构造音频参数
            AudioStreamParams audioStreamParams = new AudioStreamParams();
            audioStreamParams.setAudioChannel(AudioStreamParams.AUDIO_CHANNEL_MONO);
            audioStreamParams.setAudioFormat(AudioStreamParams.AUDIO_FORMAT_G711A);
            audioStreamParams.setAudioEncoding(AudioStreamParams.AUDIO_ENCODING_16BIT
);
            audioStreamParams.setAudioSampleRate(AudioStreamParams.AUDIO_SAMPLE_RATE_
8000);

            // 设置推流参数
            IPCDev.getInstance().getIpcStreamManager().setStreamParams(streamId, vide
oStreamParams, audioStreamParams);

            // TODO 开始采流、编码并发送音视频数据
        } catch (NoSuchStreamException e) {
            e.printStackTrace();
        }
    }
```

3. 调用 `IPCStreamManager` 方法发送音视频数据。

```
     /**
     * 发送音频帧数据
     *
     * @param streamId 流ID
     * @param directByteBuffer 源数据
     * @param length 数据长度
     * @param timeStampInMs  音频帧时间戳，单位为毫秒
     */
     void sendAudioData(int streamId, ByteBuffer directByteBuffer, int length, long tim
eStampInMs) throws NoSuchStreamException
    /**
     * 发送视频帧数据
     *
     * @param streamId 流ID
     * @param directByteBuffer 源数据
     * @param length  数据长度
     * @param isIFrame  是否为I帧
     * @param timeStampInMs 视频帧时间戳，单位为毫秒
     */
    void sendVideoData(int streamId, ByteBuffer directByteBuffer, int length, boolean i
sIFrame, long timeStampInMs) throws NoSuchStreamException

     /**
     * 发送音频帧数据
     *
     * @param streamId 流ID
     * @param data 源数据
     * @param offset 偏移量
     * @param length 数据长度
     * @param timeStampInMs 音频帧时间戳，单位为毫秒
     * @deprecated 使用 {@link #sendAudioData(int, ByteBuffer, int, long)}来替换
     */
    @Deprecated
    void sendAudioData(int streamId, byte[] data, int offset, int length, long timeStam
pInMs) throws NoSuchStreamException

    /**
     * 发送视频帧数据
     *
     * @param streamId  流ID
     * @param data 源数据
     * @param offset  偏移量
     * @param length  数据长度
     * @param isIFrame   是否为I帧
     * @param timeStampInMs 视频帧时间戳，单位为毫秒
     * @deprecated 使用 {@link #sendVideoData(int, ByteBuffer, int, boolean, long)}来替
换
     */
    @Deprecated
    public void sendVideoData(int streamId, byte[] data, int offset, int length, boolea
n isIFrame, long timeStampInMs) throws NoSuchStreamException
```

4. 打印帧的前256个字节，并查看结果。

视频播放时对H.264和H.265的帧结构有如下要求，您可参考下面的代码，打印帧的前256个字节查看帧结构。

```
for (int i = 0; i < ((buffer_size > 256)?256:buffer_size); i++) {
    printf("%02x ", buffer[i]);
    if ((i + 1) % 30 == 0) {
        printf("\n");
    }
}
printf("\n");
```

- H.264的I帧格式要求为 `帧分隔符+SPS+帧分隔符+PPS+帧分隔符+IDR` 。

  以下图为例，帧分隔符为0x000001或者0x00000001；序列参数集SPS（Sequence Parameter Set）起始数据为0x67；图像参数集PPS（Picture Parameter Set）起始数据为0x68；即时解码器刷新IDR（Instantaneous Decoding Refresh）起始数据为0x65。

```
00 00 01 67 42 c0 1e d9 00 a0 2f f9 3f f2 57 92 58 01 00 00 03 00 01 00 00 03 00 32 0f 16
2e 48 00 00 00 01 68 cb 8c b2 00 00 01 65 88 84 1f c9 81 0b 40 00 24 c7 19 c5 87 40 65 89
c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 c9 ff ff
77 c2 a1 6e 00 16 c4 d5 6f e8 1b 0d 2c af b8 00 d1 72 9b 8d 81 28 7d 2c 8c 18 5d 50 9d 09
8e 79 94 b4 d8 9d e9 7e 0b ad 99 ca 06 a7 ff c7 84 56 15 91 7b 50 01 9f a0 8a f2 75 64 67
ab c2 b2 11 61 6f 23 00 03 4a d5 93 e4 1c 11 8b 2f f8 23 0d 26 cf d4 01 5a 2c c2 67 23 bd
1d 0e ff bf ff f0 42 5e 00 09 9a 88 33 36 eb 90 00 0f e0 33 ff 80 60 b0 42 4e 00 13 59 0a
ef 4a fc 89 bf ff 80 42 bc 30 00 4c e9 0c 89 ba e4 00 1f 40 8c 1a ff f0 f5 85 78 00 73 93
75 f4 bc fc 00 0f 17 21 9c e2 40 7f cc 3c c0
```

- H.265的I帧格式要求为 `帧分隔符+VPS+帧分隔符+SPS+帧分隔符+PPS+帧分隔符+IDR` 。

  以下图为例，帧分隔符为0x000001或者0x00000001；视频参数集VPS（Video Parameter Set）起始数据为0x40；SPS起始数据为0x42；PPS起始数据为0x44；IDR起始数据为0x26。

```
00 00 00 01 40 01 0c 01 ff ff 01 40 00 00 03 00 80 00 00 03 00 00 03 00 7b ac 09 00 00 00 00 01
42 01 01 01 40 00 00 03 00 80 00 00 03 00 00 03 00 7b a0 03 c0 80 10 e5 96 b9 2c ad 9a e5 51 36
00 80 00 00 00 00 00 01 44 01 c0 e3 0f 03 32 40 00 00 00 00 01 26 01 af 13 80 5b ff ff ff a9 59
16 ca fe 88 26 26 11 cc 9a 40 61 2f 2d 47 37 39 09 25 e4 6d a2 96 33 cb 99 ff c6 74 73 cd 15 0b
98 45 db fe 98 68 e2 80 bd f6 99 b9 37 c0 a3 5b 9f 38 70 82 8a 20 b8 44 a6 d7 35 35 5c 6e ed ce
25 28 10 00 88 21 0c e0 4b fd aa 47 9b a4 e7 ad 27 f0 f5 83 e3 a6 6e 89 2b 2a 7b 18 65 1d 4a 05
ff a1 fd d3 44 fe c3 34 ad ae a5 1a b5 a2 61 83 c3 ca 6a 1a c0 7b 08 44 42 f9 bf b7 ec 88 11 01
39 33 1e 96 b9 2d 91 72 63 e0 d5 e2 af a0 82 c0 46 b7 bd 43 cf f5 36 b5 f1 09 4f b6 2e e9 e3 00
```

步骤三：处理结束推流请求。

1. 服务端下发停止推流请求时，回调 `OnLiveStreamListener.onStopPushLiveStreaming()` 方法通知设备端停止推流。
2. 处理停止推流时，一般情况下需要同时停止采集IPC设备数据和录音机数据。
   - 若需要，请调用 `IPCStreamManager的stopStreaming(int streamId)` 方法实现。
   - 若不需要，请跳过此步骤。

```
/**
     * 收到停止推流请求
     *
     * @param streamId 流ID
     */
    @Override
    public void onStopPushStreaming(int streamId){
        // TODO 停止音视频数据的发送
        try {
            // 调用停止推流接口
            IPCDev.getInstance().getIpcStreamManager().stopStreaming(streamId);
        } catch (NoSuchStreamException e) {
            e.printStackTrace();
        }
    }
```

步骤四：处理流错误。

推流过程使用`OnStreamErrorListener.onError(int streamId, StreamError error)`方法接收和处理流错误。错误码详细信息，请参考本文下方的**错误码**。

```
public interface OnStreamErrorListener{
    /**
     * 流异常时回调
     * @param streamId
     * @param error 参考StreamError定义
     */
    void onError(int streamId, StreamError error);
}
```

错误码

流错误码

错误码	标志符	描述	解决方法
1	StreamError.ERROR_STREAM_CREATE_FAILED	创建流实例失败。	该错误通常由系统资源不足引起，请您申请内存后重试。
2	StreamError.ERROR_STREAM_START_FAILED	建立RTMP链接失败。	请检查网络是否正常然后重试。
3	StreamError.ERROR_STREAM_STOP_FAILED	停止流失败。	因引入了无效的StreamId而引发的错误，该错误可忽略。

错误码	标志符	描述	解决方法
4	StreamError.ERROR_STREAM_SEND_VIDEO_FAILED	发送视频数据失败。	请根据RTMP错误码判断具体的出错原因。RTMP错误码的详细信息，请参考本文下方的**RTMP错误码**。
5	StreamError.ERROR_STREAM_SEND_AUDIO_FAILED	发送音频数据失败。	请根据RTMP错误码判断具体的出错原因。RTMP错误码的详细信息，请参考本文下方的**RTMP错误码**。
6	StreamError.ERROR_STREAM_INVALID_PARAMS	无效的流参数。	`setStreamParams`接口设置的参数无效，请检查并修改后重试。

RTMP错误码

> **说明**
> RTMP错误码只出现在日志中，仅用于排查详细问题。

错误码	标志符	描述	解决方法
-1	RTMP_ILLEGAL_INPUT	输入不合法。	请检查并修改输入参数后重试。
-2	RTMP_MALLOC_FAILED	内存分配失败。	请检查视频设备当前内存占用情况后重试。
-3	RTMP_CONNECT_FAILED	RTMP建立连接失败。	请检查网络是否正常后重试。
-4	RTMP_IS_DISCONNECTED	RTMP连接未建立。	该错误通常因服务端断开导致，可忽略。
-5	RTMP_UNSUPPORT_FORMAT	不支持的音视频格式。	`setStreamParams`接口设置的参数无效，请检查并修改后重试。

错误码	标志符	描述	解决方法
-6	RTMP_SEND_FAILED	RTMP数据包发送失败。	与服务端断开连接后再调用send接口会导致报该错误，可忽略。
-7	RTMP_READ_MESSAGE_FAILED	RTMP消息读取失败。	该错误通常因服务端断开导致，可忽略。
-8	RTMP_READ_TIMESTAMP_ERROR	输入时间戳错误。	直播推流时需保证时间戳未出现回退。请检查时间戳是否合法后重试。

3.5. 录像播放

设备端Android版本LinkVisual SDK提供录像播放功能，本文介绍实现录像播放功能的过程。

下文简称设备端Android版本LinkVisual SDK为LinkVisual SDK。

前提条件

- 已创建产品和设备，具体操作，请参见设备接入。
- 已获取LinkVisual SDK，具体操作，请参见获取SDK。
- 已完成初始化LinkVisual SDK，具体操作，请参见初始化SDK。

背景信息

录像播放功能通过RTMP协议推流。其支持的视频编码格式和音频编码格式如下：

- 视频编码格式：H.264和H.265。
- 音频编码格式：G711a、G711u和AAC_LC。

操作步骤

步骤一：注册录像播放事件监听器和流错误监听器。

- 录像播放事件监听器`OnVodStreamListener`：通知开始推流、结束推流、暂停、恢复、Seek等事件。
- 流错误监听器`OnStreamErrorListener`：通知推流中发送的错误。

步骤二：处理查询设备端录像文件列表的请求。

1. 应用端App发起查询设备端录像文件列表的请求。
2. 设备端收到调用查询设备录像文件列表的请求。

   查询设备录像文件列表的请求，由调用IPC设备的同步设备服务完成。设备服务的详细信息，请参考设备服务调用。
3. 设备端响应请求，并将查询范围内的文件列表返回至应用端App。

> **注意**
>
> 录像文件名需进行Base64编码。

```
@Override
        public void onNotify(String connectId, String topic, AMessage aMessage){
            Log.d(TAG, "onNotify() called with: connectId = [" + connectId + "], topic = ["
+ topic + "], aMessage = ["
                    + new String((byte[]) aMessage.data) + "]");
            /**
             * 添加SDK的监听
             */
            IPCDev.getInstance().notifySyncTopicReceived(connectId, topic, aMessage);

            // 处理同步服务调用
            if (CONNECT_ID.equals(connectId) && !TextUtils.isEmpty(topic) &&
                    topic.contains("rrpc")) {
                Log.d(TAG, "IConnectNotifyListener   onNotify() called with: connectId = ["
+ connectId + "], topic = ["
                        + topic + "], aMessage = ["
                        + new String((byte[]) aMessage.data) + "]");

                int code = 200;
                String data = "{}";

                JSONObject json = JSON.parseObject(new String((byte[]) aMessage.data));
                if (json != null) {
                    String method = json.getString("method");
                    JSONObject params = json.getJSONObject("params");
                    switch (method) {
                        // 查询设备录像列表请求
                        case "thing.service.QueryRecordList":
                            int beginTime = params.getIntValue("BeginTime");
                            int endTime = params.getIntValue("EndTime");
                            int querySize = params.getIntValue("QuerySize");
                            int type = params.getIntValue("Type");
                            appendLog("收到查询设备录像列表的请求: beginTime=" + beginTime +
                                    "\tendTime=" + endTime + "\tquerySize=" + querySize + "
\ttype=" + type);

                            JSONArray resultArray = new JSONArray();
                            JSONObject item1 = new JSONObject();
                            item1.put("FileName", Base64.encode("file1".getBytes(), Base64.
DEFAULT));
                            item1.put("BeginTime", System.currentTimeMillis() / 1000 - 200)
;
                            item1.put("EndTime", System.currentTimeMillis() / 1000 - 100);
                            item1.put("Size", 1024000);
                            item1.put("Type", 0);
                            resultArray.add(item1);

                            JSONObject item2 = new JSONObject();
                            item2.put("FileName", Base64.encode("file2".getBytes(), Base64.
```

```
DEFAULT));
                            item2.put("BeginTime", System.currentTimeMillis() / 1000 - 100)
;
                            item2.put("EndTime", System.currentTimeMillis() / 1000);
                            item2.put("Size", 1024000);
                            item2.put("Type", 0);
                            resultArray.add(item2);

                            JSONObject result = new JSONObject();
                            result.put("RecordList", resultArray);

                            code = 200;
                            data = result.toJSONString();
                            break;
                        default:
                            break;
                    }
                }

                MqttPublishRequest request = new MqttPublishRequest();
                request.isRPC = false;
                request.topic = topic.replace("request", "response");
                String resId = topic.substring(topic.indexOf("rrpc/request/") + 13);
                request.msgId = resId;
                request.payloadObj = "{\"id\":\"" + resId + "\", \"code\":" + code + ",\"da
ta\":" + data + "}";
                LinkKit.getInstance().publish(request, new IConnectSendListener() {
                    @Override
                    public void onResponse(ARequest aRequest, AResponse aResponse) {
                        appendLog("上报成功");
                    }

                    @Override
                    public void onFailure(ARequest aRequest, AError aError) {
                        appendLog("上报失败:" + aError.toString());
                    }
                });
            }
        }
```

步骤三：处理开始推流指令。

1. 应用端App请求步骤二返回的录像文件列表中的文件。
2. 服务端下发推流指令，提供回调

   `OnVodStreamListener.onStartPushVodStreaming(int streamId, String fileName)` 或

   ```
   OnVodStreamListener.onStartPushVodStreaming(int streamId, int beginTimeUtc, int
   endTimeUtc)
   ```

   通知视频设备端进行音视频数据推流，即录像播放。LinkVisual SDK提供了下面两种录像播放的方式：

   - 根据文件名播放设备端录像

```
    @Override
    public void onStartPushVodStreaming(int streamId, String fileName){
        appendLog("开始推点播流 " + streamId + " 文件名:" + new String(Base64.decode(fileName, Base64.NO_WRAP)));

        try {
            // 构造视频参数
            VideoStreamParams videoStreamParams = new VideoStreamParams();
            // 该视频文件的时长，单位为秒
            videoStreamParams.setDurationInS(H264_DURATION_IN_S);
            videoStreamParams.setVideoFormat(VideoStreamParams.VIDEO_FORMAT_H264);

            // 构造音频参数
            AudioStreamParams audioStreamParams = new AudioStreamParams();
            audioStreamParams.setAudioChannel(AudioStreamParams.AUDIO_CHANNEL_MONO);
            audioStreamParams.setAudioFormat(AudioStreamParams.AUDIO_FORMAT_G711A);
            audioStreamParams.setAudioEncoding(AudioStreamParams.AUDIO_ENCODING_16BIT);
            audioStreamParams.setAudioSampleRate(AudioStreamParams.AUDIO_SAMPLE_RATE_8000);

            // 设置推流参数
            IPCDev.getInstance().getIpcStreamManager().setStreamParams(streamId, videoStreamParams, audioStreamParams);

            // TODO 读取fileName文件，调用发送音视频数据接口进行推流
            // 文件推流完毕后应调用 IPCDev.getInstance().getIpcStreamManager().notifyVodComplete(streamId) 通知推流完成

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

- 根据录像时间播放设备端录像

```
    @Override
    public void onStartPushVodStreaming(int streamId, int beginTimeUtc, int endTimeUtc){
        appendLog("开始推点播流 " + streamId + " beginTimeUtc: "+beginTimeUtc + " endTimeUtc:"+endTimeUtc);

        //TODO 推流逻辑需要添加：
        // 1. beginTimeUtc和endTimeUtc是一天的开始和结束时间
        // 2. 当收到onStartPushVodStreaming回调后，应从beginTimeUtc开始向后最近的I帧开始推流，时间戳应使用对应帧的UTC时间
        // 3. 若beginTimeUtc到endTimeUtc范围内没有录像或范围内推流已经完成了，则应调用 IPCDev.getInstance().getIpcStreamManager().notifyVodComplete(streamId) 通知推流完成
        // 4. 只要是beginTimeUtc到endTimeUtc范围内有数据，即使跨文件，推流应该持续不断
    }
```

步骤四：处理暂停推流指令或恢复推流指令。

通过响应暂停推流指令或恢复推流指令 `OnVodStreamListener` ，暂停或恢复发送音视频数据。

```
/**
     * 收到暂停推流请求
     *
     * @param streamId 流ID
     */
    void onPausePushVodStreaming(int streamId);

    /**
     * 收到恢复推流的请求
     *
     * @param streamId 流ID
     */
    void onResumePushVodStreaming(int streamId);
```

步骤五：处理Seek指令。

响应Seek指令时，会回调 `onSeekTo` 方法，然后从 `timeStampInS` 时间点最近的I帧开始继续推流。例如应用端App的播放器进度条Seek到80秒时，会从80秒最近的I帧开始继续推流。

```
/**
     * 收到重新定位请求
     *
     * @param streamId     流ID
     * @param timeStampInS 时间偏移量，相对于视频开始时间，单位为秒
     */
    void onSeekTo(int streamId, long timeStampInS);
```

步骤六：处理停止推流指令。

当服务端下发停止推流请求时：

1. 通过回调 `OnVodStreamListener.onStopPushLiveStreaming()` 方法通知设备端停止推流。
2. 通常情况下需要停止调用音视频发送接口，关闭视频文件，并调用 `IPCStreamManager.getInstance().stopStreaming(int streamId)` 方法。

```
/**
     * 收到停止推流请求
     *
     * @param streamId 流ID
     */
    @Override
    public void onStopPushStreaming(int streamId){
        // TODO 停止音视频数据的发送
        try {
            // 调用停止推流接口
            IPCDev.getInstance().getIpcStreamManager().stopStreaming(streamId);
         catch (NoSuchStreamException e) {
            e.printStackTrace();
        }
    }
```

步骤七：处理流错误。

推流过程中通过 `OnStreamErrorListener.onError(int streamId, StreamError error)` 方法接收和处理流错误。错误码详细信息，请参考本文下方**错误码**。

错误码

流错误码

错误码	标志符	描述	解决方法
1	StreamError.ERROR_STREAM_CREATE_FAILED	创建流实例失败。	该错误通常由系统资源不足引起，请您申请内存后重试。
2	StreamError.ERROR_STREAM_START_FAILED	建立RTMP链接失败。	请检查网络是否正常然后重试。
3	StreamError.ERROR_STREAM_STOP_FAILED	停止流失败。	因引入了无效的StreamId而引发的错误，该错误可忽略。
4	StreamError.ERROR_STREAM_SEND_VIDEO_FAILED	发送视频数据失败。	请根据RTMP错误码判断具体的出错原因。RTMP错误码的详细信息，请参考本文下方的**RTMP错误码**。
5	StreamError.ERROR_STREAM_SEND_AUDIO_FAILED	发送音频数据失败。	请根据RTMP错误码判断具体的出错原因。RTMP错误码的详细信息，请参考本文下方的**RTMP错误码**。

错误码	标志符	描述	解决方法
6	StreamError.ERROR_STREAM_INVALID_PARAMS	无效的流参数。	`setStreamParams` 接口设置的参数无效，请检查并修改后重试。

RTMP错误码

> 说明
> RTMP错误码只出现在日志中，仅用于排查详细问题。

错误码	标志符	描述	解决方法
-1	RTMP_ILLEGAL_INPUT	输入不合法。	请检查并修改输入参数后重试。
-2	RTMP_MALLOC_FAILED	内存分配失败。	请检查视频设备当前内存占用情况后重试。
-3	RTMP_CONNECT_FAILED	RTMP建立连接失败。	请检查网络是否正常后重试。
-4	RTMP_IS_DISCONNECTED	RTMP连接未建立。	该错误通常因服务端断开导致，可忽略。
-5	RTMP_UNSUPPORT_FORMAT	不支持的音视频格式。	`setStreamParams` 接口设置的参数无效，请检查并修改后重试。
-6	RTMP_SEND_FAILED	RTMP数据包发送失败。	与服务端断开连接后再调用send接口会导致报该错误，可忽略。
-7	RTMP_READ_MESSAGE_FAILED	RTMP消息读取失败。	该错误通常因服务端断开导致，可忽略。
-8	RTMP_READ_TIMESTAMP_ERROR	输入时间戳错误。	直播推流时需保证时间戳未出现回退。请检查时间戳是否合法后重试。

3.6. 语音对讲

设备端Android版本LinkVisual SDK提供语音对讲功能，本文介绍实现语音对讲功能的过程。

下文简称设备端Android版本LinkVisual SDK为LinkVisual SDK。

前提条件

- 已创建产品和设备，具体操作，请参见设备接入。
- 已获取LinkVisual SDK，具体操作，请参见获取SDK。
- 已完成初始化LinkVisual SDK，具体操作，请参见初始化SDK。

音频类型

语音对讲支持G711a、G711U以及AAC_LC编码方式，三种音频类型详细信息为：

> 说明
>
> 选择编码方式前，请确认您的设备端IPC设备是否支持。

- 采样率：支持8 kHz和16 kHz。
- 支持编码。
- 支持解码。

对讲类型

语音对讲的类型及其注意事项如下：

类型	说明	注意事项
单向对讲	应用端App采集并发送音频数据到设备端进行播放。	应用端App采集音频期间手机保持静音。
双向对讲	应用端App和设备端同时采集音频和播放音频。	设备端必须支持声学回声消除AEC（Acoustic Echo Cancellation），否则不建议使用该功能。

操作步骤

服务端下发开始推流指令后，执行如下操作：

- 通过事先注册的语音对讲事件监听器 `OnLiveIntercomListener` 通知开始语音对讲、结束语音对讲、接收对端的音频参数和接收对端的语音数据。
- 通过流错误监听器监听推流中发生的错误，然后通知应用端App。

> 注意
>
> 请不要在回调接口中执行阻塞任务。

详细实现过程如下：

1. 选择对讲模式。

```
// 设置双向对讲模式
IPCDev.getInstance().setLiveIntercomModeBeforeInit(IPCLiveIntercomV2.LiveIntercomMode.DoubleTalk);
```

2. 设置监听。

```
// 设置语音对讲事件监听
IPCDev.getInstance().getIpcLiveIntercom().setOnLiveIntercomListener(MainActivity.this);
// 设置语音对讲错误回调
IPCDev.getInstance().getIpcLiveIntercom().setOnLiveIntercomErrorListener(MainActivity.this);
```

示例代码

注册语音对讲事件监听器和错误监听器的完整代码如下。

```
public interface OnLiveIntercomListener {

    /**
     * 收到应用端App发起的开始语音对讲请求
     *
     * @return 返回当前设备端上行音频参数格式，如采样率、通道数、采样位宽、音频格式，请确保对应用端App能支持该音频参数配置
     */
    AudioParams onStartVoiceIntercom();

    /**
     * 收到结束语音对讲请求
     */
    void onStopVoiceIntercom();

    /**
     * 收到应用端App音频参数，表示与应用端App的通道已建立，可以开始对讲
     * @param audioParams  应用端App的音频参数
     */
    void onAudioParamsChange(AudioParams audioParams);

    /**
     * 收到应用端App发送的PCM数据，一般用来做UI展示，比如绘制音量大小
     * @param buffer
     * @param size
     */
    void onAudioBufferReceive(byte[] buffer, int size);
}

public interface OnLiveIntercomErrorListener{

    /**
     * 语音对讲发生错误
     * @param error 见{@link LiveIntercomError}
     */
    void onError(LiveIntercomError error);
}
```

错误码

错误码	错误描述	解决方法
LiveIntercomError.INVALID_AUDIO_PARAMS	无效的设备端音频参数。	请检查设备端和应用端App的以下项目后重试： • 支持的音频编码格式为G711a、G711U或ACC_LC。 • 均支持单通道。 • 支持的音频采样频率为8 kHz或16 kHz。

错误码	错误描述	解决方法
LiveIntercomError.CONNECTION_STREAM_FAILED	建立语音对讲流通道失败。	请确保网络连接正常后重试。
LiveIntercomError.SEND_STREAM_DATA_FAILED	发送音频数据失败。	该错误通常由网络故障，或语音对讲对端主动关闭对讲导致，可忽略。
LiveIntercomError.INIT_RECORD_FAILED	初始化录音机错误。	请检查并修复以下异常后重试： • 检查是否录音权限未授权。 • 检查是否有其它应用占用了录音机，终止应用进程并重试。
LiveIntercomError.START_RECORD_FAILED	启动录音机错误。	请检查是否有其它应用占用了录音机，终止应用进程并重试。
LiveIntercomError.READ_RECORD_BUFFER_FAILED	读取录音机数据错误。	录音机异常，请重启设备后重试。
LiveIntercomError.INIT_AUDIO_PLAYER_FAILED	创建音频播放器失败。	音频播放器异常，请重启设备后重试。