实时互动-工业能源版

开发指南

腾讯云

【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

开发指南
视频观看与切流

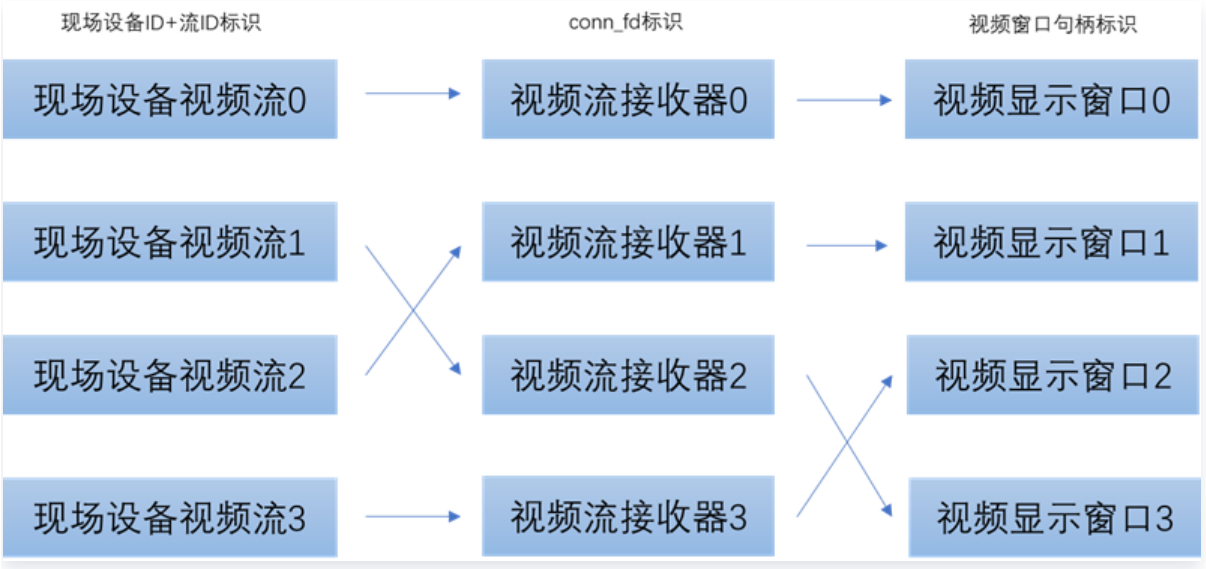
最近更新时间：2024-02-19 17:19:11

> **说明：**
> 视频观看与切流，用于控制远端设备上现场设备视频流的显示。目前支持对多个现场设备的多个视频流进行观看和切换。由远端设备 SDK 发起，无需对现场设备 SDK 进行操作。

用法介绍

视频流的接收

远端设备接收视频流时，调用 TRRO_connect 接口会通过视频流接收器来接收视频流，并关联到通过 TRRO_setWindows接口设置的视频显示窗口。下图是一组视频流和显示窗口关联关系示意：



- **现场设备视频流：**由现场设备 ID + 流 ID 唯一标识。流 ID 编号从0到 N，对应现场设备 streams_config 配置中视频流数组的元素编号。
- **视频显示窗口：**由视频窗口句柄唯一标识，例如 win32 的 HWND。内部渲染模式下，SDK 可根据视频窗口句柄在指定窗口上进行视频渲染显示。外部渲染模式下，由开发者自己维护视频显示窗口和视频的渲染显示。
- **远端设备视频流接收器：**由接收句柄（conn_fd）唯一标识。接收句柄编号从0开始递增，可由开发者自定义，一般可设置为窗口编号。

> **注意：**
> 同一时间，一个视频流接收器只能接收一个现场设备视频流。当视频流接收器连接新的视频流时，之前连接的视频流会自动断开。

可以看出，对于内部渲染，远端设备视频流接收器桥接了现场设备视频流和视频窗口，可让 SDK 自动完成视频流显示。对于外部渲染，远端设备视频流接收句柄可协助开发者来标识 SDK 回调的视频流数据应该在哪个窗口上进行渲染。

视频流的切换

远端设备进行视频流切换时，可通过 TRRO_connect 接口进行。根据新切换的视频流是否要在新的窗口显示，可分两类场景：

1. 切换后视频流在已有窗口显示：

使用已有窗口对应的接收句柄，通过 TRRO_connect 接口接收该窗口要切换的视频流。这时新视频流会切换到对应窗口显示，而窗口上原有视频流会被断开。

2. 切换后视频流在新建窗口显示：

使用新窗口对应的接收句柄，通过 TRRO_connect 接口接收切换后的视频流，并通过 TRRO_disconnect 接口断开切换前的视频流。

视频窗口的切换

内部渲染模式下，远端设备进行视频窗口切换时，可通过 TRRO_setWindows 接口进行。一般在 SDK 启动阶段，会将视频流接收器的接收句柄和视频窗口进行关联。在窗口布局需要调整时（例如交换两个视频流的所在窗口，而不涉及视频流本身的切换），可通过 TRRO_setWindows 接口更新视频流接收器的接收句柄和视频窗口的对应关系，从而在不断连视频流的情况下，完成视频流窗口的切换。

相关接口

连接视频流

使用说明：可通过该接口对现场设备视频流进行连接和切换。

```
/*
* @name :  TRRO_connect
* @brief : 发起视频连接，可多次调用连接不同流，异步模式，根据 onState 状态回调确认
视频连接成功
* @input : gwid          目标连接的现场设备 ID
*          record_config:
*                        缺省时，优先使用全局录制配置，当默认命名规则无法满足需求时
使用，json 字符串，需要对每一路进行配置 eg:"
{\"file_names\": [{\"file\":\"test\", \"duration\":15}, {\"file\":\"test
01\", \"duration\":15}]}"  ps:file 文件名 duration 分片时长单位秒
*          streams_num 要拉取现场设备视频流的个数， 与 conn_fds 和 streams_num
数组长度匹配， 值的范围 1 到 现场设备支持的device_streams数量
*          streams_id  现场设备视频流的 ID 数组， 现场设备视频流ID从0开始， 最大
值为现场设备支持的 device_streams 数量 -1
```

```
*          conn_fds     conn_fd 数组，conn_fd 为接收视频流的句柄标识，自行编号，
取值从0开始，最大值为 max_streams -1，max_streams在远端设备配置文件配置
* @return : 成功 1 失败 <= 0
*/
int TRRO_connect(const char* gwid, const char* record_config, int stream
s_num, int* streams_id, int* conn_fds);
```

参数	含义
gwid	现场设备 ID
record_config	视频录制配置。json 字符串，需要对连接的每一路视频流进行配置。建议缺省优先使用全局录制配置，无法满足时再使用该方式。 `{ "file_names": [{ "file": "test", "duration": 15 }, { "file": "test01", "duration": 15 }] }` 参数： • file：文件名 • duration：切片时长（单位：秒）
streams_num	要连接视频流的数量
streams_id	要连接的现场设备视频流 ID 数组
conn_fds	接收视频流使用的接收视频流句柄数组
返回值	• 成功 1 • 失败 <= 0

设置视频流渲染窗口

使用说明：可通过该接口设置渲染窗口，内部渲染时使用，外部渲染时设置空指针即可。

```
/*
* @name :  TRRO_setWindows
* @brief : 设置接收流句柄对应的显示窗口句柄
* @input : conn_fds 接收视频流句柄数组
*          windows  显示窗口句柄数组， 显示窗口句柄为显示窗体的句柄/指针，如
Windows 平台的 HWND;
*                   外部渲染时，数组中的显示窗口句柄配置为 nullptr
*          num      设置显示窗口数量，与 conn_fds 和 Windows 数组长度匹配
* @return :void
*/
void TRRO_setWindows(int* conn_fds, WindowIdType * windows, int num);
```

参数	含义
conn_fds	接收视频流句柄数组
windows	显示窗口句柄数组，外部渲染时可设置为空指针数组
num	与 conn_fds 和 windows 数组长度一致

关闭视频连接

使用说明：可通过该接口断开视频连接。

关闭指定视频连接

```
/*
* @name :  TRRO_disconnect
* @brief : 关闭conn_fds 对应的视频连接
* @input : conn_fds 要关闭视频连接对应的视频接收句柄数组
*          fd_num conn_fd数组长度
* @return : 成功 1 失败 <= 0
*/
int TRRO_disconnect(int* conn_fds, int fd_num);
```

参数	含义
conn_fd	视频接收句柄数组
fd_num	数组长度
返回值	• 成功 1

	• 失败 <= 0

关闭所有视频连接

```
/*
* @name :  TRRO_disconnectAll
* @brief : 关闭所有视频连接
* @input : void
* @return : 成功 1 失败 <= 0
*/
int TRRO_disconnectAll();
```

语音对讲

最近更新时间：2025-04-03 17:56:12

> **说明：**
> 语音对讲，用于远端设备和现场设备进行语音交互。 语音传输会随操控会话建立自动建议。SDK 默认只打开了上行音频。如需使用双向音频，请准备相关音频硬件。增加相关配置并重启 SDK。

硬件设备准备

设备端	上行音频： 现场设备->远端设备	下行音频： 远端设备->现场设备
现场设备	麦克风	扬声器
远端设备	扬声器	麦克风

软件配置修改

对 Linux 现场设备 SDK 及 Windows 远端设备 SDK 配置节点 config.json 文件修改，开启音频开关并设置音频设备。

开启音频开关

开启音频接收，将 audio_receive 配置为1； 开启音频采集，将audio_enable 配置为1。

设备端	上行音频： 现场设备 -> 远端设备	下行音频： 远端设备 -> 现场设备
现场设备	"audio_enable":1	"audio_receive":1
远端设备	"audio_receive":1	"audio_enable":1

Linux 系统使用配置文件设置音频设备

SDK 默认使用系统默认的音频播放和采集设备，Linux 平台下如果目标音频设备不是系统默认设备，可通过配置指定使用对应的音频设备。

配置项	配置说明	配置举例
音频播放设备	• `audio_play":"hw:#Card,#Device,#SubDevice` • 其中 #Card , #Device , #SubDevice 为声卡设备编号，Linux 平台可通过 `aplay -l` 查询目标设备	例如，目标设备是 Card 0， Device 0， SubDevice #0： `audio_play":"hw:0,0,0`

	的 card、device、subdevice 的编号。	
音频采集设备	• `audio_record":"hw:#Card,#Device,#SubDevice` • 其中 #Card , #Device , #SubDevice 为声卡设备编号，Linux 平台可通过 `arecord -l` 查询目标设备的 card、device、subdevice 的编号	例如，目标设备是 Card 0， Device 0，SubDevice #0： `audio_record":"hw:0,0,0`

示例：执行命令 "aplay -l ", 可以查看音频播放设备。

```
nvidia@nvidia-desktop:~$ aplay -l
**** List of PLAYBACK Hardware Devices ****
card 0: tegrahdagalent1 [tegra-hda-galen-t194], device 3: HDMI 0 [HDMI 0]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```

如图使用的播放设备为"card 0, device 3, subdevice 0"， 则在配置文件 config.json 中增加配置节点"audio_play":"hw:0,3,0"。

Linux 系统使用 API 设置音频设备

可使用 C/C++ 现场设备 SDK API 中音频控制类 API 接口来设置当前音频设备.

1. TRRO_getDeviceCount 获取目标类型的音频设备列表。
2. TRRO_getDeviceName 获取设备名称，并通过名称过滤, 找到目标名称的音频设备编号。
3. TRRO_setCurrentDevice 将对应设备编号的音频设备设置为当前使用的音频设备。

函数列表	描述
TRRO_getDeviceCount	获取音频采集/播放设备数量
TRRO_getDeviceName	获取音频采集/播放设备名称
TRRO_setCurrentDevice	更改当前音频采集/播放设备

外部输入 PCM 音频数据

当应用需要自行音频采集时，可使用外部输入PCM音频数据接口，输入音频数据。

- 需打开音频配置 "audio_enable":1 以及外部音频输入配置"audio_external":1。
- 每次音频数据输入长度对应10ms音频数据。

- 输入音频 PCM 数据格式须为 16bit 采样小端对齐，即 s16le 格式。
- 输入音频采样率支持 8k/16k/24k/32k/44.1k/48k 等，48k 采样对应输入为48000。

```
/**
* @name  TRRO_externAudioData
* @brief 外部音频数据输入(pcm数据), 16位音频采样
* @param[in]   data          源数据
* @param[in]   data_size     数据大小
* @param[in]   channel       音频声道数
* @param[in]   sample_rate   音频采样率
* @return 1 for success, other failed
*/
extern "C" TRRO_EXPORT int TRRO_externAudioData(const char* data, int
data_size, int channel, int sample_rate);
```

SDK 回调 PCM 音频数据

当应用需要自行音频播放时，可使用 PCM 音频数据接回调口，获取 SDK 混流后的音频数据。使用 PCM 音频数据回调时，SDK 内部音频播放会关闭。

1. 需打开音频配置 "audio_receive":1 以及外部音频播放配置"audio_play":“outside”。
2. 每次音频数据回调长度对应10ms音频数据。
3. 输出音频 PCM 数据为16bit采样小端对齐，即 s16le 格式。
4. 输出音频采样率为48kHz。

```
/**
* @name  TRRO_onRemoteMixAuidoFrame
* @brief 对端多路混音后音频原始数据回调
* @param[in] context     回调上下文指针, 返回注册回调函数时传入的context
* @param[in] data        音频PCM数据 10ms 16bits，小端对齐
* @param[in] length      数据长度
* @param[in] channel     音频声道数目，如单声道为1，双声道为2，多声道为N
* @param[in] sample_rate   音频采样率
* @return : void
*/

typedef void TRRO_onRemoteMixAuidoFrame(void* context, const char* data,
int length, int channel, int sample_rate);

/**
* @name TRRO_registerRemoteMixAuidoFrameCallback
* @brief 注册远端混音音频数据回调函数
```

```
* @param[in]    context              上下文
* @param[in]    callback             回调函数
* @return 1 for success, other failed
*/
extern "C" TRRO_EXPORT int
TRRO_registerRemoteMixAuidoFrameCallback(void* context,
TRRO_onRemoteMixAuidoFrame * callback);
```

控制数据传输

最近更新时间：2024-12-11 16:32:22

说明：
控制数据传输，用于现场设备和远端设备之间的二进制数据通信。两端SDK均提供一个数据发送接口和一个数据接收回调接口。

注意：
- 现场设备 SDK，数据发送接口会向与该现场设备处于会话连接中的所有远端设备发送数据消息。
- 远端设备 SDK，数据发送接口可以向会话连接中的指定现场设备发送数据消息，仅当该远端设备拥有现场设备控制权时，才可发送成功。

现场设备：

向远端设备发送数据

说明：
该接口会向与该现场设备处于会话连接中的所有远端设备发送数据消息。 发送消息单条长度限制为700字节，发送频率限制为100Hz。

```
/**
* @name   TRRO_sendControlData
* @brief  向远端设备发送数据
* @param[in] msg            消息体
* @param[in] len            消息体长度
* @param[in] qos            0:unreliable 1:reliable
* @return 1 for success, other failed
*/
extern "C" TRRO_EXPORT int TRRO_sendControlData(const char* msg, int
len, int qos = 0);
```

参数	含义
msg	消息内容
len	消息长度

qos	发送 qos 类型。 • 0：超低时延传输，网络不佳时可能会有丢失 • 1：可靠传输
返回值	• 成功：1 • 失败：<= 0

注册控制消息回调

```
/**
* @name TRRO_registerControlDataCallback
* @brief 注册远端设备消息回调函数
* @param[in]   context             上下文指针，回调时会返回该上下文指针
* @param[in]   callback            回调函数
* @return 1 for success, other failed
*/
extern "C" TRRO_EXPORT int TRRO_registerControlDataCallback(void*
context, TRRO_onControlData * callback);
```

参数	含义
context	回调上下文
TRRO_onControlData	回调函数

回调函数定义

```
/**
* @name TRRO_onControlData
* @brief 接收远端设备消息回调
* @param[in]     context         上下文指针，返回注册时传入的context
* @param[in]     controller_id   远端设备ID
* @param[in]     msg             消息体内容
* @param[in]     len             消息体长度
* @param[in]     qos             消息qos类型 0:unreliable, 1:reliable
* @return void
*/
typedef void TRRO_onControlData(void *context, const char
*controller_id, const char* msg, int len, int qos);
```

参数	含义

context	回调上下文
controller_id	控制端设备id
msg	控制消息字符串
len	字符串长度
qos	消息 qos 类型： • 0：超低时延传输，网络不佳时可能会有丢失 • 1：可靠传输

远端设备

向现场设备发送数据

使用说明：该接口会向会话连接中的指定现场设备发送数据消息，仅当该远端设备拥有现场设备控制权时（ 控制授权管理 ），才可发送成功。 发送消息单条长度限制为700字节，发送频率限制为100Hz。

```
/*
* @name :  TRRO_sendControlData
* @brief : 向现场设备发送控制数据
* @input : gwid      目标现场设备 ID
*          msg       发送消息，二进制透传
*          len       消息长度
*          qos       消息传输qos   0:超低时延传输  1:可靠传输
* @return : 成功 1 失败 <= 0
*/
extern "C" TRRO_EXPORT int TRRO_sendControlData(const char* gwid, const
char* msg, int len, int qos = 0);
```

参数	含义
gwid	现场设备 ID
msg	发送二进制数据
len	消息长度
qos	发送 qos： • 0：超低时延传输，网络不佳时可能会有丢失 • 1：可靠传输

返回值	• 成功：1 • 失败：<= 0

注册控制消息回调

使用说明：此接口用于注册现场设备发送的报告数据回调函数，可根据需要实现。

```
/**
* @name TRRO_registerReportDataCallback
* @brief 注册现场设备消息回调函数
* @param[in]   context              上下文指针，回调时会返回该上下文指针
* @param[in]   callback             回调函数
* @return 1 for success, other failed
*/
extern "C" TRRO_EXPORT int TRRO_registerReportDataCallback(void*
context, TRRO_onReportData * callback);
```

参数	含义
context	上下文指针，回调时会返回该上下文指标用于定位
callback	回调函数

```
/*
* @name :  TRRO_onReportData
* @brief : 接收来自现场设备信息回调
* @input : context      回调上下文指针，返回注册回调函数时传入的context
*          gwid         消息来源现场设备id
*          msg          消息，二进制透传
*          len          消息长度
*          qos          消息来源传输qos，  0:不可靠传输，1:可靠传输
* @return : void
*/
typedef void STD_CALL TRRO_onReportData(void* context, const char* gwid,
const char* msg, int len, int qos);
```

参数	含义
context	回调上下文，返回注册时传入的 context

gwid	现场设备 ID
msg	控制消息字符串
len	字符串长度
qos	消息 qos 类型。 • 0：超低时延传输，网络不佳时可能会有丢失 • 1：可靠传输

控制授权管理

最近更新时间：2024-02-07 10:08:21

> **说明：**
> 控制授权管理用于分时管理远端设备给现场设备发送消息的权限。当有多个远端设备有权限操控某个现场设备时，可通过控制授权来限制同一时刻只有一个远端设备可以给现场设备发送控制消息，保障操控的安全性。

基本概念

权限类型

1. guest 权限

   远端设备作为旁观者，只能观看现场设备音视频内容和收到现场设备的上报数据，无法给现场设备下发控制消息。
2. master 权限

   远端设备作为操控者，既可以观看现场设备音视频内容和收到现场设备的上报数据，又可以给现场设备下发控制消息。

> **注意：**
> - 目前现场设备 SDK 限制一个现场设备同时只能给一个远端设备授权为 master 权限。
> - 当现场设备 SDK 给某个远端设备授权为 master 权限时，其他连接的远端设备会自动转为 guest 权限。

基本流程

1. 远端设备向现场设备发送授权请求请求权限， 0 为 guest 权限，1为 master 权限。
2. 现场设备根据回调的远端设备授权请求，对远端设备进行授权处理，并给相关远端设备发送授权通知。
3. 远端设备可根据现场设备的授权通知，判断当前授权状态。

授权模式

远端设备 SDK 和现场设备 SDK 可以分别配置授权模式是否为自动模式。具体配置的 json 字段为 `auto_permission` 字段，设置为1时为自动授权模式，设置为0时是应用授权模式，缺省配置默认为自动授权模式。

自动授权模式

应用可无需关心授权的处理，SDK 会自动按以下默认逻辑进行授权处理：

- 远端设备设置自动授权模式时，在发起 connect 请求时，会自动给连接的现场设备发送 master 权限授权请求。
- 现场设备设置自动授权模式时，收到远端设备的授权请求时，会自动按远端设备请求的授权权限对远端设备进行授权处理。

应用授权模式

SDK 提供相关授权接口，应用需要进行接口调用进行授权处理：

- 远端设备 SDK 不会自动发送 master 权限授权请求，需要应用显式调用权限请求接口，向现场设备请求权限授权。
- 现场设备 SDK 不会自动处理授权请求，需要应用根据回调的授权请求内容，结合应用定义的授权规则，显式调用授权接口，对远端设备进行授权。

授权接口

远端设备

注册现场设备操控权限变化通知

```
/*
*   回调注册函数
*   context 上下文指针，回调时会返回注册时传入的该指针，
*   callback 注册的 TRRO_OnOperationPermissionState 回调函数
*/

extern "C" TRRO_EXPORT void TRRO_registerOnOperationPermissionState(void
* context, TRRO_OnOperationPermissionState* callback);


/*
* #name :   TRRO_OnOperationPermissionState
* @brief :  回调现场设备操控权限状态通知
* @input :  context              回调上下文指针，返回注册回调函数时传入的
context
*           field_devid          来源现场设备 ID
*           self_permission      本远端设备当前操控权限，参考 TrroPermission，
0 是 guest，只有观看权限， 1是 master，拥有完全控制权限
*           master_devid         拥有 master 权限的远端设备 ID
* @return : void
*/
typedef void STD_CALL TRRO_OnOperationPermissionState(void* context,
const char* field_devid, int self_permission, const char* master_devid);
```

请求现场设备操控权限操控权限

```
/*
* @name : TRRO_requestPermission
* @brief : 向网关发出权限请求，网关会反馈TRRO_OnOperationPermissionState 更新
权限
* @input : gwid          目标设备 ID
*          permisson     参考结构体 TrroPermission
* @return : 成功1 失败 <= 0
*/
extern "C" TRRO_EXPORT int TRRO_requestPermission(const char* gwid, int
permisson);
```

现场设备

注册远端设备操控权限请求通知回调

使用说明：此接口用于在现场设备接收权限控制请求，可按需选择使用。

```
/**
* @name  TRRO_registerOperationPermissionRequest
* @brief 注册远端设备操控权限请求通知回调
* @param[in]   context              上下文
* @param[in]   callback             回调函数
* @return 1 for success, other failed
*/
extern "C" TRRO_EXPORT int TRRO_registerOperationPermissionRequest(void
*context, TRRO_onOperationPermissionRequest *callback);
```

参数	含义
context	上下文指针，回调时会返回该指针用于定位
callback	TRRO_onOperationPermissionRequest 回调函数

```
/**
* @name TRRO_onOperationPermissionRequest
* @brief 远端设备操控权限申请通知
* @param[in]     remote_devid        请求权限的 remote deviceId
* @param[in]     permission          请求的权限，参考TrroPermission,   0:
guest 只有观看权限,  1: master 完全控制权限
* @return void
*/
typedef void TRRO_onOperationPermissionRequest(void* context, const
char* remote_devid, int permission);
```

参数	含义
context	上下文指针，回调时会返回该指针用于定位
remote_devid	请求权限的 remote deviceId
permission	请求的权限，参见 TrroPermission： • 0：guest 只有观看权限， • 1：master 完全控制权限

设置远端设备操作权限

使用说明：此接口用于在改现场设备设置远端设备的操作权限。

```
/**
 * @name  TRRO_setOperationPermission
 * @brief 设置远端设备操控权限，目前同时只能有一个远端设备有 master 权限，若已有远端设备是 master 权限，调用该接口设置 master权限，会自动取消之前设备的master权限然后设置新设备；
 * @param[in]  remote_devid          设置权限的对端设备 ID
 * @param[in]  permission            参考 TrroPermission， 0 guest，只有观看权、1 master， 完全控制权限
 * @return 1 for success, other failed
 */
extern "C" TRRO_EXPORT int TRRO_setOperationPermission(const char* remote_devid, int permission);
```

参数	含义
context	上下文指针，回调时会返回该指针用于定位
callback	回调函数

会话连接诊断

最近更新时间：2024-02-20 10:59:32

> **说明：**
> 会话连接诊断，用于远端设备诊断与指定现场设备会话连接的情况。当与现场设备会话连接出现问题时，可通过该功能定位会话链路问题。

诊断流程

> **注意：**
> 诊断前，需要先对目标设备发起会话连接，然后再对目标设备进行诊断，最后从诊断回调接口中获取诊断报告。诊断时间约为10秒。

步骤1：发起设备会话连接

调用远端设备 SDK 的 connect 接口，对指定现场设备视频进行会话连接。

```
/*
* @name : TRRO_connect
* @brief : 发起视频连接，可多次调用连接不同流，异步模式，根据 onState 状态回调确认
视频连接成功
* @input : gwid 目标连接的现场设备 ID
* record_config:
* 优先使用录制配置，当默认录制配置命名规则无法满足需求时使
用，json 字符串，需要对每一路进行配置 eg:"{\"file_names\":
[{\"file\":\"test\", \"duration\":15}, {\"file\":\"test01\",
\"duration\":15}]}" ps:file 文件名 duration 分片时长单位秒
* streams_num 要拉取现场设备视频流的个数， 与conn_fds和streams_num数
组长度匹配， 值的范围 1 到 现场设备支持的device_streams数量
* streams_id 现场设备视频流的 ID 数组， 现场设备视频流 ID 从0开始， 最
大值为现场设备支持的 device_streams数量 -1
* conn_fds conn_fd数组，conn_fd为接收视频流的句柄标识，自行编号，取
值从0开始，最大值为 max_streams -1，max_streams在远端设备配置文件配置
* @return : 成功 1 失败 <= 0
*/
int TRRO_connect(const char* gwid, const char* record_config, int
streams_num, int* streams_id, int* conn_fds);
```

步骤2：注册诊断回调接口

```
/*
*    回调注册函数
*    context 上下文指针，回调时会返回注册时传入的该指针，
*    callback 注册的回调函数，回调函数实现中请勿在回调线程长时间阻塞，若需要长时间处
理建议转移到其他线程处理
*/

extern "C" TRRO_EXPORT void TRRO_registerOnDiagReport(void* context,
TRRO_OnDiagReport * callback);
```

回调函数

```
/*
* #name :    TRRO_OnDiagReport
* @brief :   诊断信息回调
* @input :   context       回调上下文指针，返回注册回调函数时传入的 context
*            gwid          发起诊断的目标现场设备 ID
*            type          1 成功 ，< 0 相关错误信息
*            json          回调诊断详细信息 json 格式
* @return : void
*/
typedef void STD_CALL TRRO_OnDiagReport(void* context, const char* gwid,
int type, const char* json);
```

步骤3：对设备会话进行诊断

调用 TRRO_connect 接口后，保持视频会话连接，通过调用诊断接口，等待从诊断回调函数中获取诊断报告。诊断时间为10秒左右。

```
/*
* @name : TRRO_DiagRequest
* @brief : 需要主动出发，诊断网关、控制端当前状态输出报告
* @input : gwid      目标设备 ID
* @return : 成功1 失败 <= 0
*/
extern "C" TRRO_EXPORT int TRRO_diagRequest(const char* gwid);
```

参数	含义
gwid	目标现场设备 ID

诊断说明

诊断输出报告为 json 格式字符串，包含现场设备和远端设备的 mqtt 服务状态、信令服务连接状态和媒体服务连接状态以及媒体链路状态等信息。

JSON 字段	含义	正常值	异常值可能原因
remote_mqtt_connect	远端设备 mqtt 状态	ok	• 与 mqtt 服务网络或端口不通 • 同名设备登录被阻塞
diag_error	诊断错误状态	ok	• 现场设备 mqtt 服务连接异常 • 远端设备 mqtt 服务连接异常
device_id	现场设备 ID	—	—
remote_mode	远端设备模式	与现场设备模式相同	• 现场设备与远端设备配置模式不一致
result	诊断详情	JSON 数组，按视频流号排列	• 诊断失败，无法获取详情
result / stream_id	现场设备视频流 ID	—	—
result / conn_fd	远端设备接收句柄 ID	—	—
result / field	现场设备详情	JSON 字符串	• 获取现场设备诊断信息失败
result / field / signal_connect	现场设备信令连接	ok	• 与信令服务网络或端口不通
result / field / media_connect	现场设备媒体连接	ok	• 媒体服务网络或端口不通 • 现场设备未推流
result / field / video_capture	现场设备视频采集	ok	• 相机采集异常 • 没有输入视频数据 • 使用外部编码流输入
result / field / video_encode	现场设备视频解码	ok	• 编码失败 • 编码器没有输入视频数据
result / field / video_transfer	现场设备视频发送	ok	• 没有进行推流 • 媒体服务或信令服务连接出错

result / field / lost	现场设备视频发送丢包	<1	• 现场设备网络发送有丢包
result / field / fps	现场设备视频发送帧率	与配置帧率相近	• 网络受限导致降帧 • cpu 或编码器处理能力不足 • 相机或输入帧率不足
result / field / bps	现场设备视频发送码率	与配置码率范围相近	• 网络受限导致发送码率不足 • 编码器没有输出或输出码率不足
result / field / rtt	现场设备网络延迟	<100ms	• 现场设备网络延迟过大
result / remote	远端设备详情	JSON字符串	—
result / remote / signal_connect	远端设备信令连接	ok	• 与信令服务网络或端口不通
result / remote / media_connect	远端设备媒体连接	ok	• 媒体服务网络或端口不通 • 现场设备未推流
result / remote / video_transfer	远端设备视频接收	ok	• 现场设备没有推流数据
result / remote / video_decode	远端设备视频解码	ok	• 没有解码数据 • 解码失败
result / remote / latency	远端设备视频延迟	30 – 150	• 视频延迟异常
result / remote / lost	远端设备视频接收丢包	<1	• 远端设备视频网络接收有丢包
result / remote / fps	远端设备视频接收帧率	与现场设备配置帧率相近	• 网络受限导致接收帧率不足 • 现场设备实际发送帧率不足
result / remote / bps	远端设备视频接收码率	与现场设备发送码率相近	• 网络受限导致接收码率不足
result / remote / rtt	远端设备网络延迟	<100ms	• 远端设备网络延迟过大

网络摄像机接入

最近更新时间：2024-02-07 10:08:21

说明：
现场设备 SDK 支持基于 RTSP 方式拉取网络相机的 h264/h265/mjpeg 流，可支持透传和转码两种相机接入方式。调用前需确认相机厂商提供的 RTSP 拉流 URL 格式，并验证 URL 对应视频流可正常播放：

- 透传模式：SDK 直接采用网络相机编码流进行传输。该模式下，SDK 无法自适应调整视频流传输码率。
- 转码模式：SDK 会先将网络相机视频流解码后再重新编码进行传输。

透传模式

视频流配置中，protocol 采用 rtsp_enc 模式，并在 cameras 配置中给出网络相机接入配置。下图给出了 config.json 文件中 streams_config 网络相机流的配置示例：

文件名： config.json
文件位置： $(workspace)/config.json
文件类型： json
注意： "//"后注释在使用时要删除。

```
{
 "device_id":"dev1",   //修改为控制台中创建的现场设备 ID
 "device_name":"vin234",
 "device_streams":1,   //如果是多路输入,修改这里的流数目，并增加 streams_config
数组中的元素个数
 "cloud_mode":"public",
 "certificate":"./device.pem",
 "projectid" : "xxxxx", //修改为控制台中创建的项目 ID
 "password": "xxxxx",   //修改为控制台中创建的密码
 "streams_config": [
        {
            "fps":25,
            "bps":2000,
            "width":1920,
            "height":1080,
            "protocol":"rtsp_enc",   //网络相机透传模式
            "cameras": [
                {
                    "width":1920,
                    "height":1080,
```

```
                    "protocol":1,
                    "url":"rtsp://xxxxx" //获取厂商对应网络摄像头的拉取 url，
并填写在此字段中
                }
            ]
        }
    ]
}
```

转码模式

视频流配置中，protocol 采用 normal 模式，并在 cameras 配置中给出网络相机接入配置。下图给出了 config.json 文件中 streams_config 网络相机流的配置示例：

文件名： config.json
文件位置： $(workspace)/config.json
文件类型： json
注意： "//"后注释在使用时要删除。

```
{
 "device_id":"dev1",  //修改为控制台中创建的现场设备 ID
 "device_name":"vin234",
 "device_streams":1,  //如果是多路输入,修改这里的流数目，并增加 streams_config
数组中的元素个数
 "cloud_mode":"public",
 "certificate":"./device.pem",
 "projectid" : "xxxxx", //修改为控制台中创建的项目 ID
 "password": "xxxxx",  //修改为控制台中创建的密码
 "streams_config": [
        {
            "fps":25,
            "bps":2000,
            "width":1920,
            "height":1080,
            "protocol":"normal",  //转码接入模式
            "cameras": [
                {
                    "width":1920,
                    "height":1080,
                    "protocol":1,
                    "url":"rtsp://xxxxx" //获取厂商对应网络摄像头的拉取 url，
并填写在此字段中
                }
            ]
```

```
            }
        ]
    }
```

免注册登录与临时会话授权

最近更新时间：2025-04-03 17:56:12

> **说明：**
> 免注册登录与观看授权，可不需要预先对设备进行创建，基于项目共享密钥生成设备密钥进行登录和自动注册，并可选择会话设备进行临时授权。其安全性略低于设备独立密钥方式，但由于不需要预先注册，适合希望简化业务流程的客户。

开通项目共享密钥

通过云 API 接口 修改项目安全模式 开启项目共享密钥，设置32位项目密钥，并根据使用场景设置是否允许自动注册以及远端获取现场设备列表。

字段	说明	取值
Mode	安全模式	• 0：关闭项目共享密钥 • 1：开启项目共享密钥
Key	项目共享密钥	32位字符串，小写英文 + 数字
AutoRegister	自动注册方式	• 0：关闭自动注册 • 1：仅允许现场设备自动注册 • 2：仅允许远端设备自动注册 • 3：允许现场和远端设备均自动注册
FieldListEnable	是否允许远端获取现场设备列表（getGwList）	• 0：不允许 • 1：允许

> **注意：**
> - 开启项目共享密钥后，请注意保护项目共享密钥，并及时更新。建议项目共享密钥保存在服务器侧。由服务器生成设备登录密码下发给设备，避免密钥保存在客户端侧产生的密钥泄露风险。
> - 开启项目共享密钥后，对于已注册的设备，仍可使用原设备密码登录。若希望仅能通过共享密钥生成密码登录，请通过云 API 将设备密码更新为"USEPROJECTKEYPWD"。

设备登录密码生成

开启项目共享密钥后，服务允许设备采用基于项目共享密钥生成的设备密码登录。可基于设备 ID 以及项目共享密钥生成设备密码，具体计算方法如下：

python

```python
import hmac
import hashlib
import base64
def genDevicePassword(devId,Key,expiretime):
    content = devId+str(expiretime)
    hmac_digest = hmac.new(Key.encode('utf-8'), content.encode('utf-8'),
hashlib.sha256).digest()
    base64_str = base64.b64encode(hmac_digest).decode('utf-8')
    signature =
base64_str.replace('+','').replace('/','').replace('=','')
    return signature+"_"+str(expiretime);
```

go

```go
import (
    "crypto/hmac"
    "crypto/sha256"
    "encoding/base64"
    "fmt"
    "strings"
)

func GenDevicePassword(devid, secKey string, expiretime int64) string {
    h := hmac.New(sha256.New, []byte(secKey))
    h.Write([]byte(fmt.Sprintf("%s%d", devid, expiretime)))
    digest := h.Sum(nil)
    encoded := base64.StdEncoding.EncodeToString(digest)
    encoded = strings.ReplaceAll(encoded, "+", "")
    encoded = strings.ReplaceAll(encoded, "/", "")
    encoded = strings.ReplaceAll(encoded, "=", "")
    return encoded+fmt.Sprintf("_%d",expiretime)
}
```

java

```java
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;
```

```
public class DeviceTokenGenerator {
    public static String genToken(String key, String devId, Long
expiretime) {
        try {
            // 创建一个 HmacSHA256 密钥
            SecretKeySpec secretKeySpec = new
SecretKeySpec(key.getBytes(), "HmacSHA256");
            // 获取 Mac 对象来执行 HMAC-SHA256
            Mac mac = Mac.getInstance("HmacSHA256");
            mac.init(secretKeySpec);
            String data = devId + expiretime.toString();
            // 执行 HMAC-SHA256 计算
            byte[] hmacBytes = mac.doFinal(data.getBytes());
            // 转base64
            String encoded =
Base64.getEncoder().encodeToString(hmacBytes);
            // 去除 + 和/，转小写
            return encoded.replace('+',
'').replace('/','').replace('=','')  + "_" + expiretime.toString();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return "";
    }
}
```

js

```
async function genDevicePassword(devId, key, expiretime) {
  // 1. 准备数据
  const content = devId + expiretime.toString();
  const encoder = new TextEncoder();
  const keyData = encoder.encode(key);
  const contentData = encoder.encode(content);
  // 2. 生成 HMAC-SHA256
  const cryptoKey = await window.crypto.subtle.importKey(
        "raw",
        keyData,
        { name: "HMAC", hash: "SHA-256" },
        false,
```

```
        ["sign"]    );
  const signature = await window.crypto.subtle.sign(
        "HMAC",
        cryptoKey,
        contentData    );
   // 3. Base64 编码并处理特殊字符
   const base64 = btoa(String.fromCharCode(...new
Uint8Array(signature)));
   const cleaned = base64.replace(/\+/g, '').replace(/\//g,
'').replace(/=/g, '');
   return `${cleaned}_${expiretime}`;
}
```

⚠ 注意：

- 用于生成密码的设备 ID 字符串中带有项目 ID 前缀，形如项目 ID / 设备 ID。密码过期时间戳的精度为分钟，等于过期时 UNIX 时间戳 / 60。设备登录时，当服务器时间超过密码的过期时间，设备密码将失效。
- 设备初始化时，可使用上述生成的设备密码作为 password 进行登录。
- 请确认 SDK 版本支持免注册登录。

会话授权码生成

免注册登录时，若项目设置为白名单模式，而服务端并没有设备 ID 对应的白名单设置，会造成远端设备没有权限观看现场设备。此时需要基于项目共享密钥生成会话授权码，并将授权码分发给远端设备。远端设备通过会话授权接口设置授权码可获得对应的临时会话权限。具体会话授权码生成的计算方法如下：

python

```
import hmac
import hashlib
import base64
def genSessionAuthCode(remoteDevId,fieldDevId,Key,expiretime):
    content = remoteDevId+fieldDevId+str(expiretime)
    hmac_digest = hmac.new(Key.encode('utf-8'), content.encode('utf-8'),
hashlib.sha256).digest()
    base64_str = base64.b64encode(hmac_digest).decode('utf-8')
    signature =
base64_str.replace('+','').replace('/','').replace('=','')
    return signature+"_"+str(expiretime);
```

go

```go
import (
    "crypto/hmac"
    "crypto/sha256"
    "encoding/base64"
    "fmt"
    "strings"
)

func GenSessionAuthCode(remoteDevId, fieldDevId, secKey string,
expiretime int64) string {
    h := hmac.New(sha256.New, []byte(secKey))

h.Write([]byte(fmt.Sprintf("%s%s%d",remoteDevId,fieldDevId,expiretime)))
    digest := h.Sum(nil)
    encoded := base64.StdEncoding.EncodeToString(digest)
    encoded = strings.ReplaceAll(encoded, "+", "")
    encoded = strings.ReplaceAll(encoded, "/", "")
    encoded = strings.ReplaceAll(encoded, "=", "")
    return encoded+fmt.Sprintf("_%d",expiretime)
}
```

java

```java
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;

public class SessionAuthCodeGenerator {
    public static String genAuthCode(String key, String remoteDevId,
String fieldDevId, Long expiretime) {
        try {
            // 创建一个HmacSHA256密钥
            SecretKeySpec secretKeySpec = new
SecretKeySpec(key.getBytes(), "HmacSHA256");
            // 获取Mac对象来执行HMAC-SHA256
            Mac mac = Mac.getInstance("HmacSHA256");
            mac.init(secretKeySpec);
            String data = remoteDevId +  fieldDevId +
expiretime.toString();
            // 执行HMAC-SHA256计算
```

```
            byte[] hmacBytes = mac.doFinal(data.getBytes());
            // 转base64
            String encoded =
Base64.getEncoder().encodeToString(hmacBytes);
            // 去除 + 和/，转小写
            return encoded.replace('+',
'').replace('/','').replace('=','') + "_" + expiretime.toString();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return "";
    }
}
```

js

```
async function genSessionAuthCode(remoteDevId,fieldDevId,Key,expiretime)
{
  // 1. 准备数据
  const content = remoteDevId + fieldDevId + expiretime.toString();
  const encoder = new TextEncoder();
  const keyData = encoder.encode(key);
  const contentData = encoder.encode(content);
  // 2. 生成 HMAC-SHA256
  const cryptoKey = await window.crypto.subtle.importKey(
        "raw",
        keyData,
        { name: "HMAC", hash: "SHA-256" },
        false,
        ["sign"]    );
  const signature = await window.crypto.subtle.sign(
        "HMAC",
        cryptoKey,
        contentData    );
   // 3. Base64 编码并处理特殊字符
   const base64 = btoa(String.fromCharCode(...new
Uint8Array(signature)));
   const cleaned = base64.replace(/\+/g, '').replace(/\//g,
'').replace(/=/g, '');
   return `${cleaned}_${expiretime}`;
}
```

> **注意：**
> 用于生成密码的远端设备和现场设备 ID 字符串中均带有项目 ID 前缀，形如项目 ID / 设备 ID。授权码过期时间戳的精度为分钟，等于过期时 UNIX 时间戳 / 60。当服务器时间超过密码的过期时间，会话授权将失效。

远端设备设置会话授权码

远端设备可通过 TRRO_setSessionPermissionToken 接口设置临时会话授权。一个会话授权码，用于一对远端设备和现场设备会话授权。远端设备可针对不同的现场设备设置不同的会话授权码。针对同一个现场设备，后设置的临时会话授权将覆盖之前设置的。

```
/*
* @name :  TRRO_setSessionPermissionToken
* @brief : 项目白名单模式下，未设置白名单的设备，需要通过会话授权码才能观看
* @input : fieldDevId      目标现场设备 ID，格式为项目ID/设备ID
*          authcode        授权码
*          expiretime      授权码过期时间戳，精度为分钟
* @return : 成功 1 失败 <= 0
*/
int TRRO_setSessionPermissionToken(const char* fieldDevId, const char*
authcode);
```

参数	含义
fieldDevId	目标现场设备 ID。
authcode	授权码，需要基于项目共享密钥生成。
返回值	• 成功：1 。 • 失败：<= 0。

典型使用场景

自动注册

- **场景需求：**
  无需提前对设备进行注册，设备首次上线时，即可自动完成注册。
- **用法建议：**

开启项目共享密钥，并设置自动注册。设备首次登录时，基于项目共享密钥生成的登录密码上线并完成注册。后续仍然可设置设备专属密码或继续基于共享密钥生成密码登录。

免注册观看

- **场景需求：**

  远端设备无需注册，即可观看视频流，适合临时观看。
- **用法建议：**

  开启项目共享密钥，并关闭自动注册以及远端现场设备列表获取功能。对于远端设备，可使用临时生成的远端设备 ID 和共享密钥生成的登录密码登录，拉取目标现场设备视频流。如果项目设置为白名单模式，约束远端对现场设备视频的观看权限，则还需基于共享密钥生成临时观看授权给远端设备。

批量密码变更

- **场景需求：**

  服务端希望能够一次更新所有设备密码，无需单个设备更新。
- **用法建议：**

  开启共享密钥模式设置自动注册，并将已注册设备密码更新为“USEPROJECTKEYPWD”。定期更改项目共享密钥，实现批量密码更新。

多网络路径传输

最近更新时间：2024-08-26 15:23:42

> **说明：**
> 多网络路径传输，可支持对音视频流、控制数据流采用多网络路径进行传输。使用时需要确保设备网络环境符合多网要求，且 SDK 配置多个对应网卡进行绑定。

SDK 多网配置

SDK 初始化时，加入网卡 IP 绑定配置，绑定多个网卡 IP 即可开启多网功能，并可选配置视频多网传输模式。

- 网卡 IP 绑定配置

```
"network_bind":["IP1","IP2"]
```

如果采用 Web 工具生成配置，在高级配置中，勾选指定 IP 绑定网卡配置，填写多个网卡 IP 并用分号隔开。

- 视频多网传输模式（可选）

```
"media_trans_mode":2
```

1为 5G 增强模式，适合多 5G 网络场景；2为均衡模式； 3为 4G 增强模式，适合多 4G 网络场景。

> **注意：**
> 使用 SDK 的应用运行多网模式时，为确保能够成功绑定网卡，需要授权 sudo 权限。如果不方便授权 sudo 权限，也可为应用单独授权网卡绑定权限，在应用目录下执行以下内容：`sudo setcap cap_net_raw,cap_net_bind_service+ep ./应用名` 。

设备多网环境验证

使用 SDK 进行多网传输前，需要验证设备多网环境正常，满足 SDK 多网传输要求。具体验证时，可在设备上使用 curl 命令访问可返回客户端公网 IP 的网站URL（以下以使用 http://cip.cc 为例），验证 SDK 配置绑定的多个网卡 IP 请求的公网 IP 地址是否符合多网环境要求。

```
curl http://cip.cc --interface 本地网卡IP1
curl http://cip.cc --interface 本地网卡IP2
```

如果**不同网卡 IP 的 curl 请求能稳定返回不同的公网 IP**，且与对应网络运营商线路匹配，则验证通过。

设备路由及 IP 配置（供参考）

以下是满足设备多网环境验证要求的一种车端网络部署方案示例，可用于网络部署参考，非强制建议。实际设备网络只要符合设备多网环境验证要求即可。

CPE 配置要求

单个多网 CPE

同一个 CPE 接入不同运营商 WAN 网络。 通过 VLAN+路由配置方式创建多个不同网段的 LAN 网关指向不同的 WAN 网络， 例如192.168.67.1网段网关指向WAN1 网络（A 运营商）， 192.168.8.1网段网关指向 WAN2 网络（B 运营商）。

多个单网 CPE

不同 CPE 接入不同运营商 WAN 网络。将多个 CPE 的 LAN 网段设置为不同网段。例如 CPE1 为 192.168.8.1/24网段， CPE2 为 192.168.67.1/24网段。

设备 IP 及路由配置

单网卡连接 CPE

设备采用1个网口及网线，通过交换机与 CPE 多个 LAN 网段网关连接。

- IP 设置

  为网卡设置多个网段的 IP 以及相应的网关（若无法配置网关也可）。以下是 Ubuntu 系统 /etc/network/interfaces.d 设置举例，对 eth0 网卡创建 eth0:0，分配 IP 192.168.8.96 及网关 192.168.8.1， 创建 eth0:1 分配 IP 192.168.67.96 及网关192.168.67.1。

```
auto eth0
iface eth0 inet static

auto eth0:0
iface eth0:0 inet static
address 192.168.8.96
netmask 255.255.255.0
gateway 192.168.8.1

auto eth0:1
iface eth0:1 inet static
address 192.168.67.96
netmask 255.255.255.0
gateway 192.168.67.1
```

- 路由表设置

  为每个网络的 IP 创建独立路由表，并指定路由的网关。以下是 eth0 网卡不同 IP 路由设置的举例：

```
ip rule add from 192.168.8.96 table 10
ip rule add from 192.168.67.96 table 20
ip route add default via 192.168.8.1 dev eth0 proto static metric 0
table 10
ip route add default via 192.168.67.1 dev eth0 proto static metric 0
table 20
```

多网卡连接 CPE

设备采用多个网口及网线，分别与 CPE 多个 LAN 网段网关连接（可通过交换机）。

- IP 设置

  为每个网卡设置对应网段的IP以及相应的网关。以下是 Ubuntu 系统 /etc/network/interfaces.d 设置举例，对 eth0 网卡分配 IP 192.168.8.96 及网关192.168.8.1，对 eth1 网卡分配 IP 192.168.67.96 及网关192.168.67.1。

```
auto eth0
iface eth0 inet static
address 192.168.8.96
netmask 255.255.255.0
gateway 192.168.8.1

auto eth1
iface eth1 inet static
address 192.168.67.96
netmask 255.255.255.0
gateway 192.168.67.1
```

- 路由表设置

  为每个网络的 IP 创建独立路由表，并指定路由的网关。以下是对 eth0 和 eth1 网卡不同 IP 路由设置的举例：

```
ip rule add from 192.168.8.96 table 10
ip rule add from 192.168.67.96 table 20
ip route add default via 192.168.8.1 dev eth0 proto static metric 0
table 10
ip route add default via 192.168.67.1 dev eth1 proto static metric 0
table 20
```