云直播

SDK 实践

产品文档

腾讯云

版权所有：腾讯云计算（北京）有限责任公司

【版权声明】

©2013-2024 腾讯云版权所有

本文档著作权归腾讯云单独所有，未经腾讯云事先书面许可，任何主体不得以任何形式复制、修改、抄袭、传播全部或部分本文档内容。

【商标声明】

及其它腾讯云服务相关的商标均为腾讯云计算（北京）有限责任公司及其关联公司所有。本文档涉及的第三方主体的商标，依法由权利人所有。

【服务声明】

本文档意在向客户介绍腾讯云全部或部分产品、服务的当时的整体概况，部分产品、服务的内容可能有所调整。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。

文档目录

SDK 实践
- 0. SDK 接入引导
- 1. 推流
  - 推流接入详情
  - Web 推流 SDK API
- 2. 播放
- 3. 高级功能
  - 概述
  - 媒体传输 SDK（TMIO SDK）
  - 快直播传输层 SDK 播放器集成指引
  - Web 端本地混流
  - 弹幕及会话聊天集成指引

SDK 实践

0. SDK 接入引导

最近更新时间：2022-11-21 16:56:28

本章节为您介绍在自己的业务程序中快速集成直播推流、拉流的方案，不同业务涉及的 SDK 具体如下：

<table>
<tr><th colspan="2">功能模块</th><th>App 端</th><th>Web 端</th><th>文档指引</th></tr>
<tr><td colspan="2">推流</td><td>iOS & Android</td><td>Web 推流</td><td>推流接入</td></tr>
<tr><td colspan="2">播放</td><td>iOS & Android</td><td>-</td><td>播放接入</td></tr>
<tr><td rowspan="5">高级功能</td><td>播放 AV1 视频</td><td colspan="2">接入指引</td><td rowspan="5">高级功能接入</td></tr>
<tr><td>上行高质量传输</td><td>接入步骤</td><td>-</td></tr>
<tr><td>播放器支持快直播播放</td><td>接入步骤</td><td>-</td></tr>
<tr><td>弹幕及会话聊天集成指引</td><td>接入步骤</td><td>-</td></tr>
<tr><td>美颜特效</td><td>iOS & Android</td><td>-</td></tr>
</table>

版权所有：腾讯云计算（北京）有限责任公司

1. 推流

推流接入详情

最近更新时间：2023-12-25 14:05:57

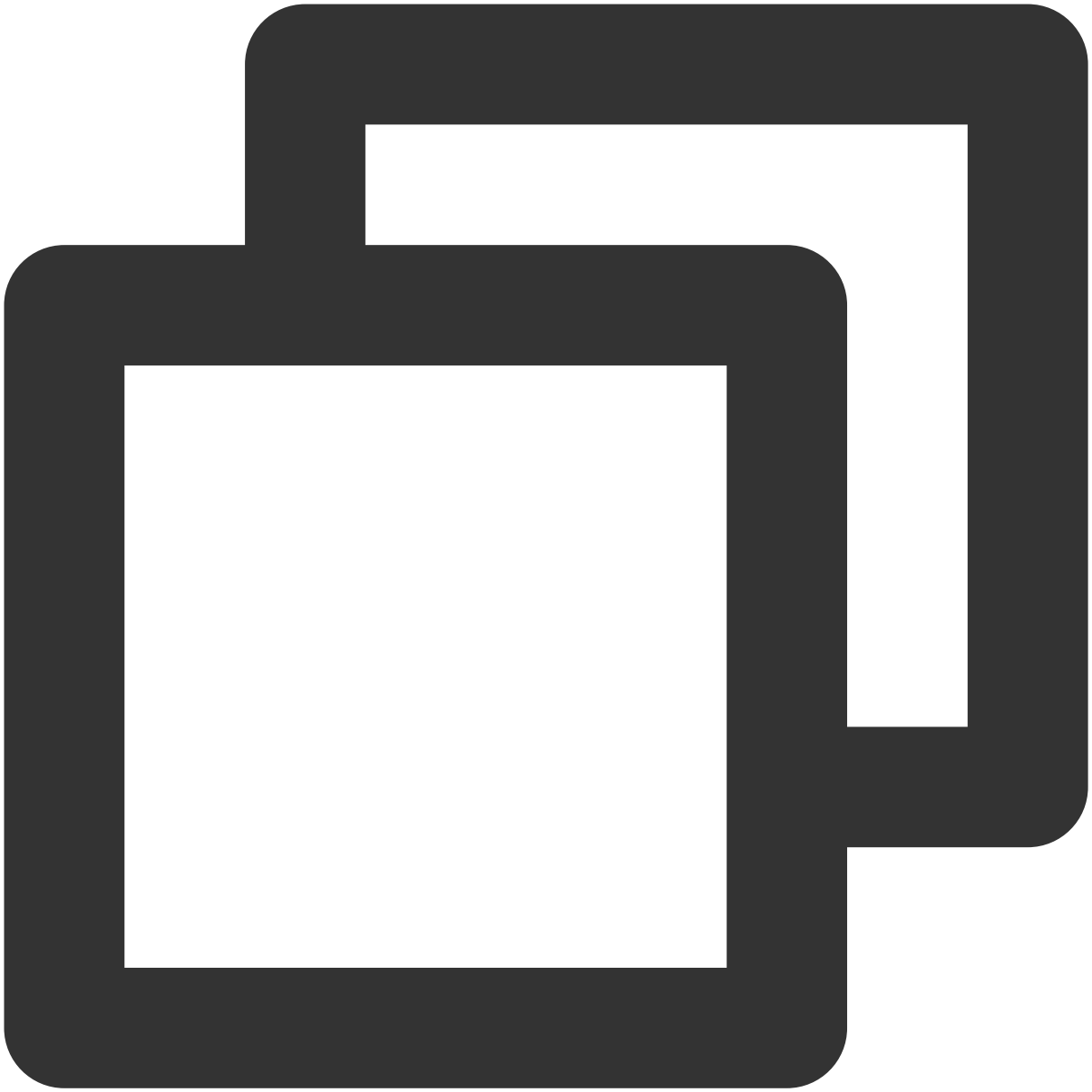
通过阅读本文，您可以了解到如何在自己的程序中通过集成 SDK 或插件，实现云直播推流功能。

准备工作

开通 腾讯云直播服务。
选择 域名管理，单击 **添加域名** 添加推流域名，详细请参见 添加自有域名。
进入云直播控制台的 **直播工具箱** > 地址生成器生成推流地址，详情请参见 地址生成器。接下来根据业务场景使用以下方式在自己的业务中实现直播推流。

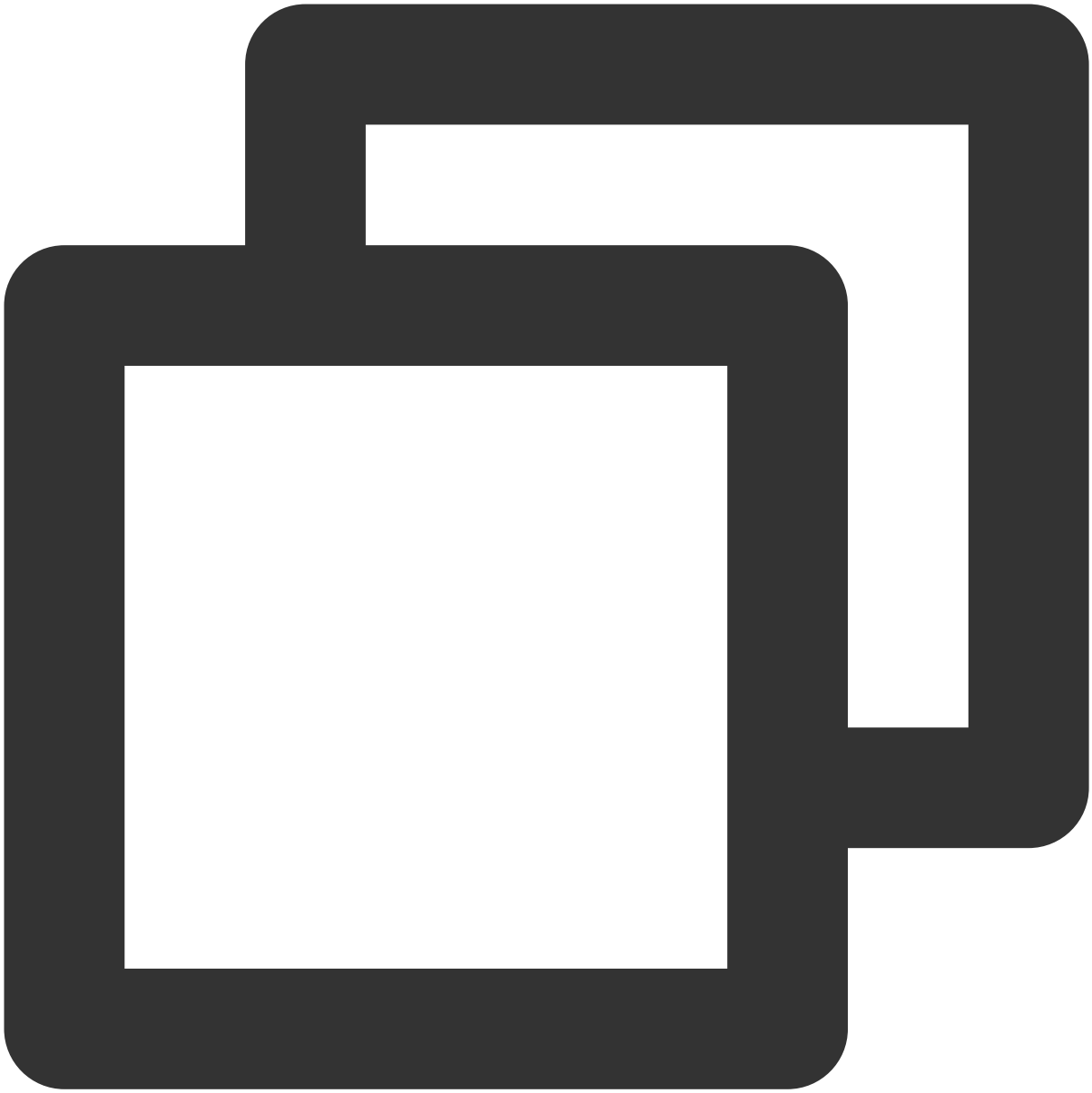
Native App 接入

下载并集成直播SDK(MLVB)，具体可参考对接文档（iOS & Android）完成接入。
注意：
开启 RTMP 推流需在初始化 V2TXLivePusher 组件时，创建 TXLivePusher 对象指定对应 V2TXLiveMode 为 `_RTMP` ，iOS 和 Android 的处理方式分别如下：
iOS



```
V2TXLivePusher *pusher = [[V2TXLivePusher alloc] initWithLiveMode:V2TXLiveMode_RTMP
```

Android



```
V2TXLivePusher mLivePusher = new V2TXLivePusherImpl(this, V2TXLiveDef.V2TXLiveMode.
```

Web 接入

Web 接入目前仅支持的推流协议是 WebRTC，请参考对接文档 WebRTC 完成接入。若需进一步实现本地混流，具体请参见 Web 端本地混流。

注意：

您也可以在云直播控制台的 Web 推流 直接进行 Web 网页推流。

PC 接入

使用 OBS 接入

在 PC（Windows/Mac）端可通过 OBS 直接进行推流， OBS 支持 Windows/Mac/Linux 等系统，是一个免费的开源的视频录制和视频实时流软件。
如果推流协议为 WebRTC，具体请参见 OBS WebRTC 推流。

使用 FFmpeg 接入

在 PC 端您同样可以使用 FFmpeg 工具进行推流，FFmpeg 是一个可以进行多种格式音频/视频录制、转码、串流的开源软件。
如果推流协议为 WebRTC，则需使用支持 WebRTC 推流的 FFmpeg 工具， 具体请参见 使用 FFmpeg 进行 WebRTC 推流。

更多

在使用腾讯云视立方·直播 SDK 的过程中需要付费，若您需要了解腾讯云视立方·直播 SDK 相关计费说明，详情请参见 价格总览。

版权所有：腾讯云计算（北京）有限责任公司

Web 推流 SDK API

最近更新时间：2023-12-25 15:18:29

API 概览

TXLivePusher

腾讯云直播推流器，主要用于浏览器 Web 推流。通过浏览器采集用户的画面和声音，通过 WebRTC 将视频流和音频流传输推送到腾讯云直播服务端。

API	描述
checkSupport	静态函数，检查浏览器支持性。
setRenderView	设置本地视频画面的预览容器。
setVideoQuality	设置推流视频质量。
setAudioQuality	设置推流音频质量。
setProperty	调用高级 API 接口。
startCamera	打开摄像头设备。
stopCamera	关闭摄像头设备。
startMicrophone	打开麦克风设备。
stopMicrophone	关闭麦克风设备。
startScreenCapture	开启屏幕采集。
stopScreenCapture	关闭屏幕采集。
startVirtualCamera	开始采集本地媒体文件流。//验证到了这里
stopVirtualCamera	停止采集本地媒体文件流。
startCustomCapture	使用用户自定义的音视频流。
stopCustomCapture	关闭用户自定义的音视频流。

版权所有：腾讯云计算（北京）有限责任公司

startPush	开始推流。
stopPush	停止推流。
isPushing	查询当前是否正在推流中。
getMediaStream	根据流 ID 获取采集到的音视频流。
getDeviceManager	获取设备管理对象。
getVideoEffectManager	获取视频效果管理对象。
getAudioEffectManager	获取音频效果管理对象。
setVideoMute	设置是否禁用视频流。
setAudioMute	设置是否禁用音频流。
pauseVideo	禁用视频流。
pauseAudio	禁用音频流。
resumeVideo	恢复视频流。
resumeAudio	恢复音频流。
setVideoContentHint	设置视频内容提示，用于提升在不同内容场景下的视频编码质量。
setObserver	设置推流事件回调通知。
destroy	离开页面或者退出时，清理 SDK 实例。

TXDeviceManager

设备管理接口，主要用于管理摄像头、麦克风设备，进行设备的获取和切换操作。

API	描述
getDevicesList	获取设备列表。
getCurrentDevice	获取当前流的设备信息。
switchDevice	切换当前正在使用的设备。
switchCamera	切换摄像头设备。
switchMicrophone	切换麦克风设备。

版权所有：腾讯云计算（北京）有限责任公司

TXAudioEffectManager

音频效果管理接口，主要用于调整音量的操作。

API	描述
setVolume	设置音频流的音量大小。

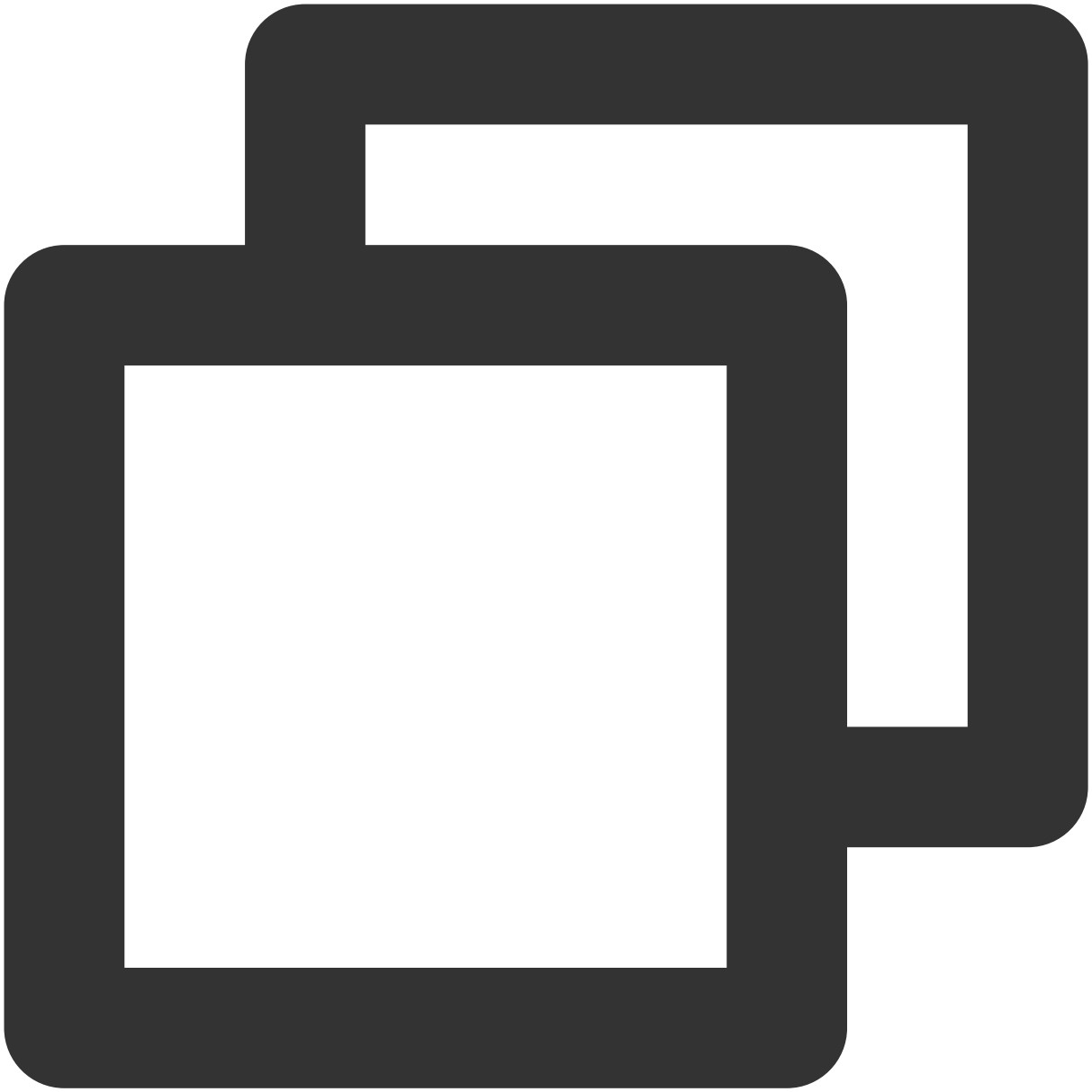
TXVideoEffectManager

视频效果管理接口，主要用于设置画中画、镜像、滤镜、水印、文本等操作。

API	描述
enableMixing	开启本地视频画面混流功能。
setMixingConfig	设置混流参数。
getMixingConfig	获取最终采用的混流参数。
setLayout	设置视频流的画中画布局参数。
getLayout	获取指定流的画中画布局参数。
setMirror	设置视频流的镜像效果。
setNormalFilter	设置视频流的普通滤镜效果。
setWatermark	设置水印。
setText	设置文本。

TXLivePusher

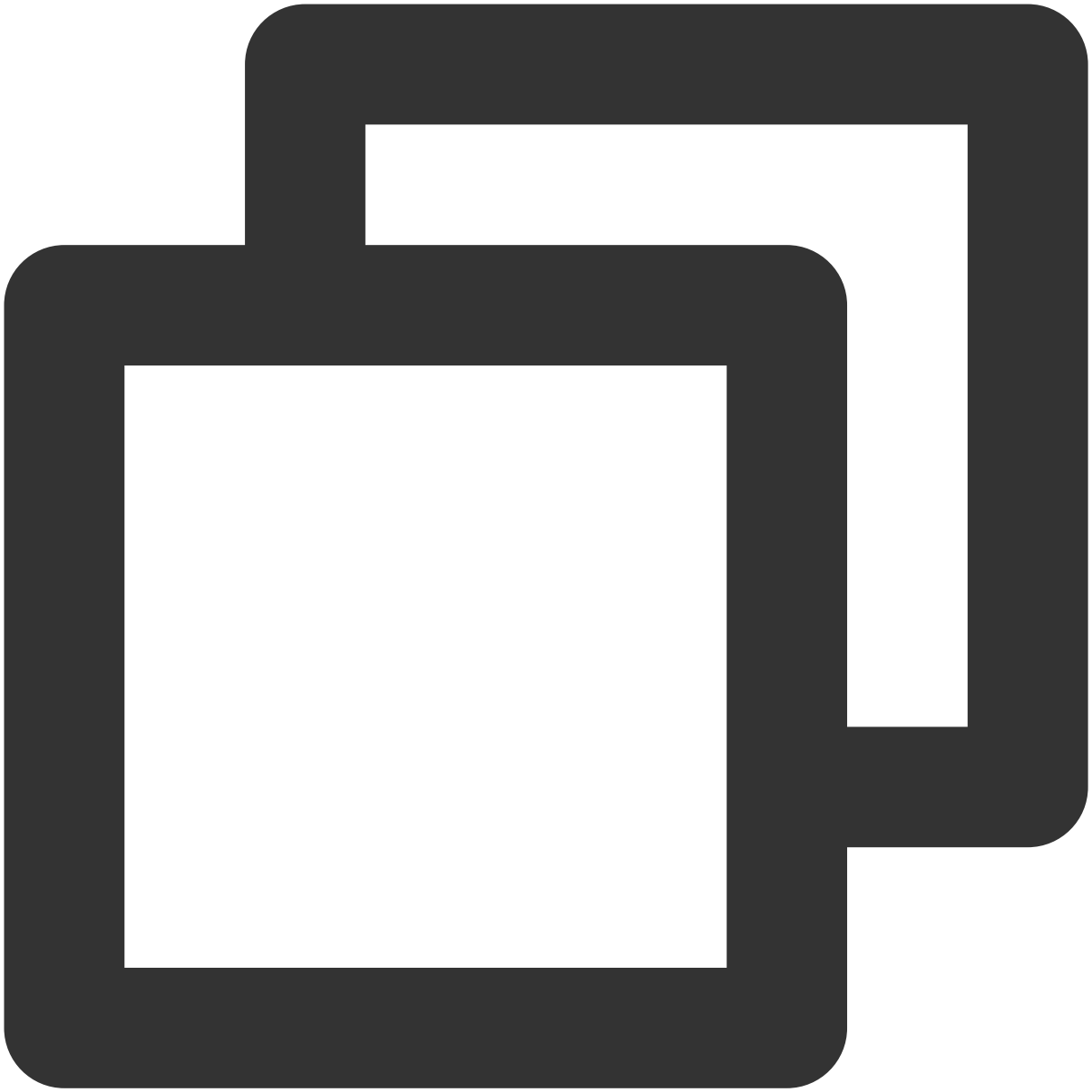
腾讯云直播推流器，主要用于浏览器 Web 推流。通过浏览器采集用户的画面和声音，通过 WebRTC 将视频流和音频流传输推送到腾讯云服务端。如果需要开启本地混流功能，调用 `TXVideoEffectManager` 方法 enableMixing() 来启用。
请先创建实例对象，用于后续所有操作。



```
const livePusher = new TXLivePusher();
```

checkSupport

静态函数，检查浏览器支持性。



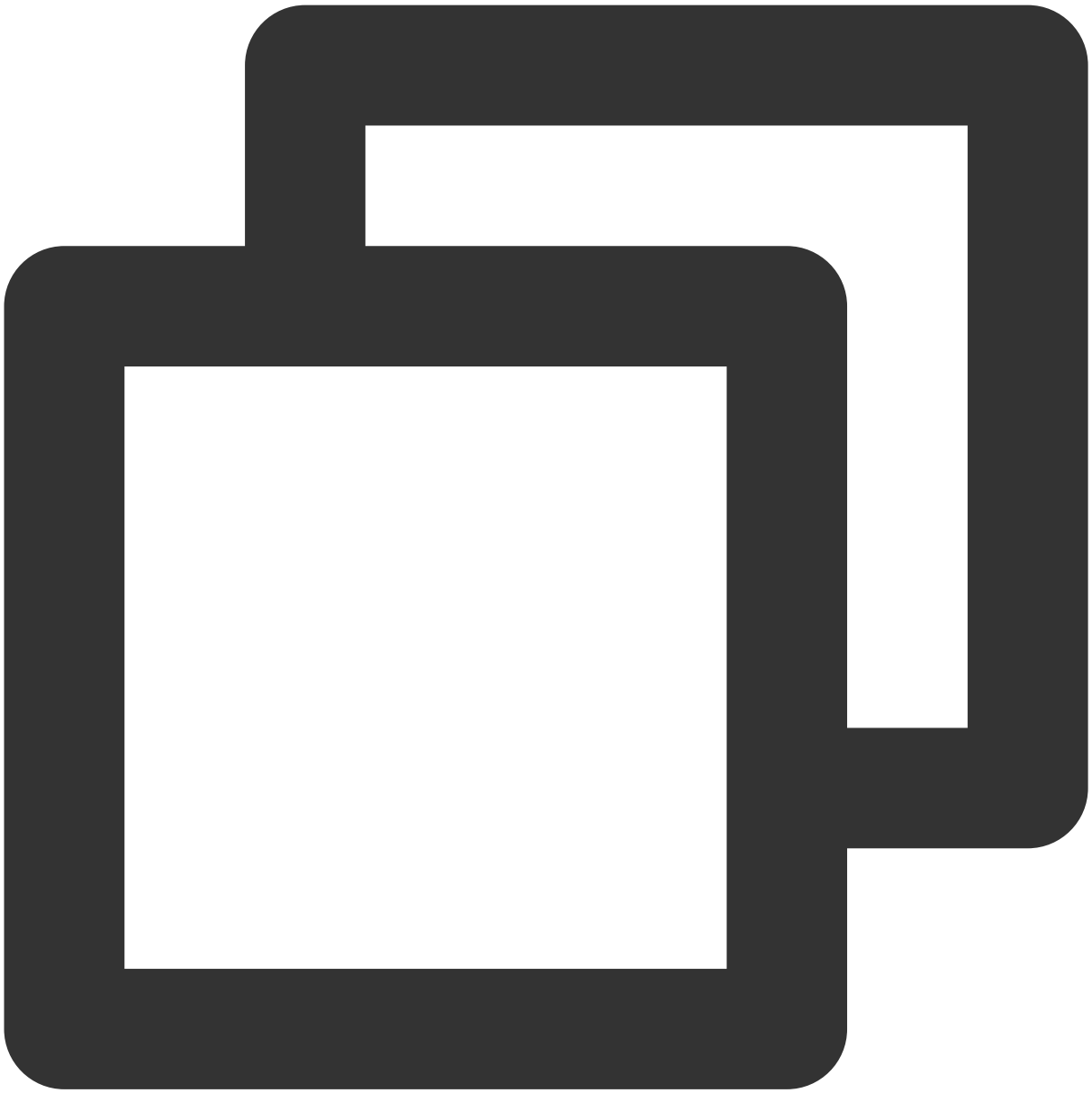
```
static checkSupport(): Promise<TXSupportResult>;
```

返回：

返回 Promise 对象，其中检查结果数据结构请参考 TXSupportResult 。

setRenderView

设置本地视频画面的预览容器，需提供一个 div 节点，本地采集的视频会在容器里渲染。如果开启了本地混流功能，容器里面会渲染混流处理之后的音视频。



```
setRenderView(container: string | HTMLDivElement): void;
```

参数：

字段	类型	说明
container	string | HTMLDivElement	容器的 ID 或者 dom 节点。

setVideoQuality

设置推流视频质量，SDK 已经内置了视频质量模板，直接通过预定义的模板来设置推流视频质量。

```
setVideoQuality(quality: string): void;
```

参数：

字段	类型	说明
quality	string	预定义的视频质量模板名称。

内置的视频质量模板如下所示：

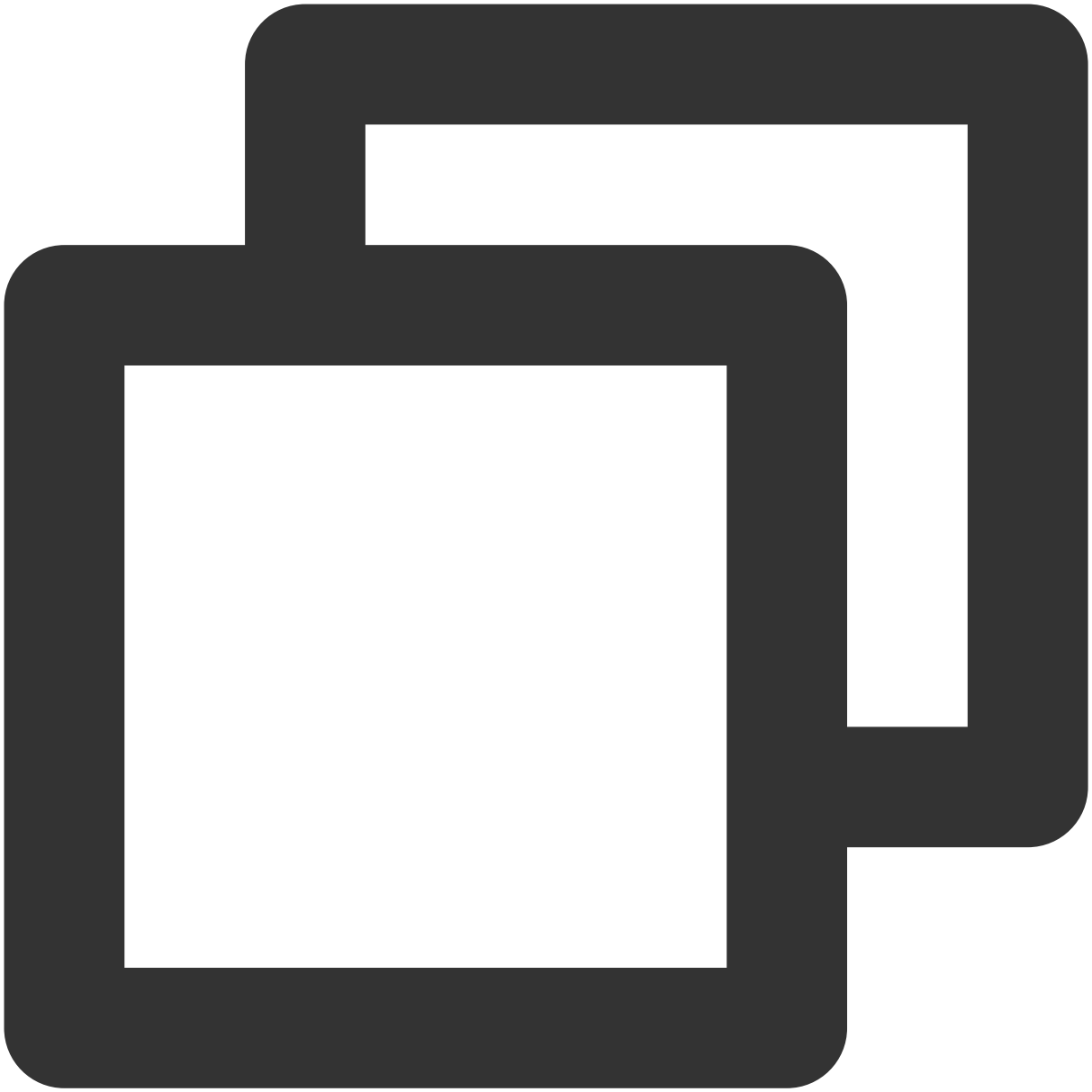
模板名	分辨率（宽 x 高）	帧率（fps）	码率（kbps）
120p	160 x 120	15	200
180p	320 x 180	15	350
240p	320 x 240	15	400
360p	640 x 360	15	800
480p	640 x 480	15	900
720p	1280 x 720	15	1500
1080p	1920 x 1080	15	2000
2K	2560 x 1440	30	4860
4K	3840 x 2160	30	9000

说明：

1. 由于设备和浏览器的限制，视频分辨率不一定能够完全匹配，在这种情况下，浏览器会自动调整分辨率使其接近对应的分辨率。
2. 如果视频质量参数（分辨率、帧率和码率）不符合您的要求，您可以通过 setProperty() 单独设置自定义的值。
3. 此处视频分辨率主要表示本地采集的视频分辨率，推流时分辨率可能会低于采集的分辨率，浏览器会根据网络带宽等情况自动调整推流分辨率。
4. 默认使用 720p ，即 `setVideoQuality('720p')` 。

setAudioQuality

设置推流音频质量，SDK 已经内置了音频质量模板，直接通过预定义的模板来设置推流音频质量。



```
setAudioQuality(quality: string): void;
```

参数：

字段	类型	说明
quality	string	预定义的音频质量模板名称。

内置的音频质量模板如下所示：

模板名	采样率	码率（kbps）

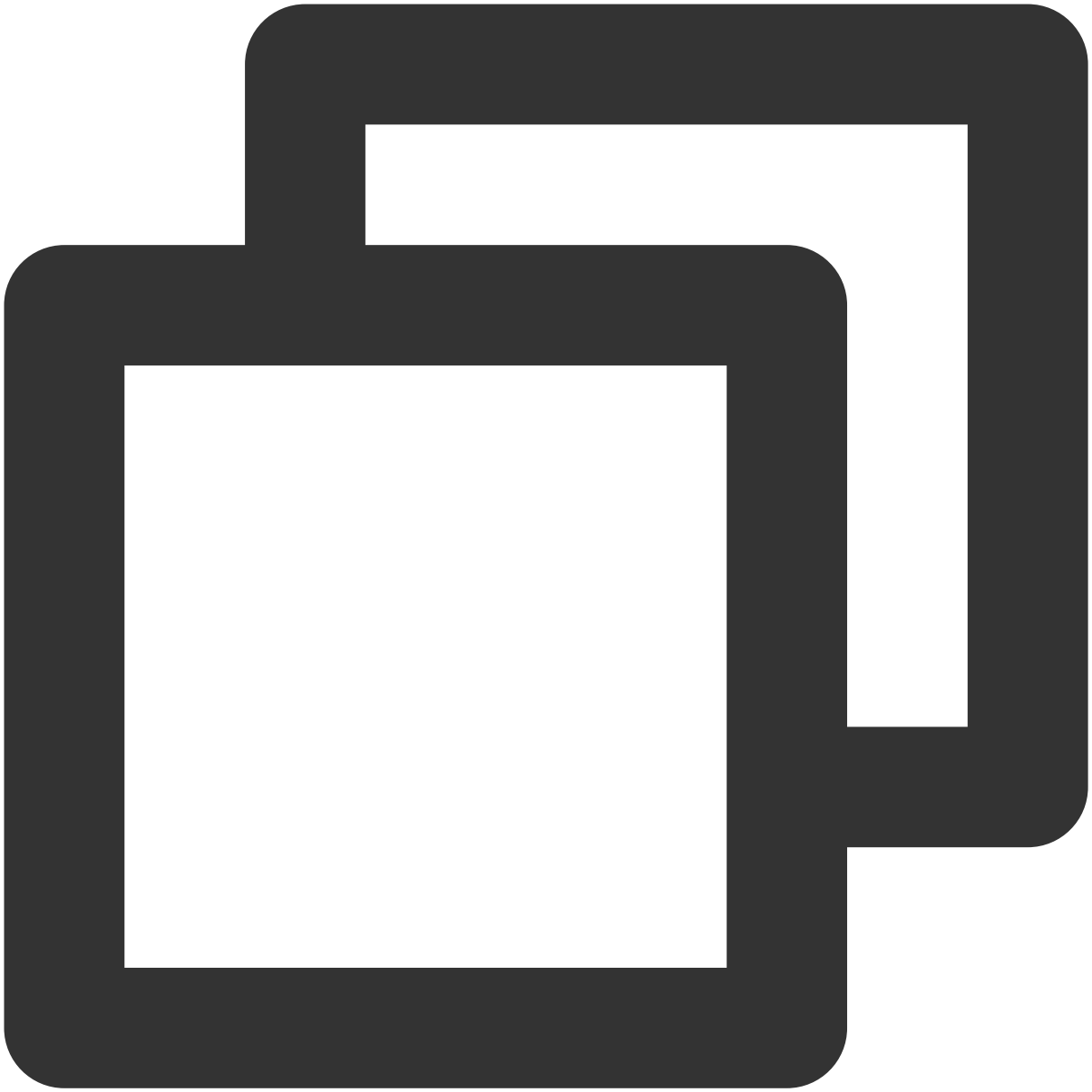
standard	48000	40
high	48000	128

说明：

1. 如果音频质量参数（采样率和码率）不符合您的要求，您可以通过 setProperty() 单独设置自定义的值。
2. 默认使用 standard ，即 `setAudioQuality('standard')` 。

setProperty

主要用于调用一些高级功能，比如设置视频的分辨率、帧率和码率，设置音频的采样率和码率等。



```
setProperty(key: string, value: any): void;
```

参数：

字段	类型	说明
key	string	高级 API 对应的 key。
value	*	调用 key 所对应的高级 API 时需要的参数。

目前支持以下高级功能：

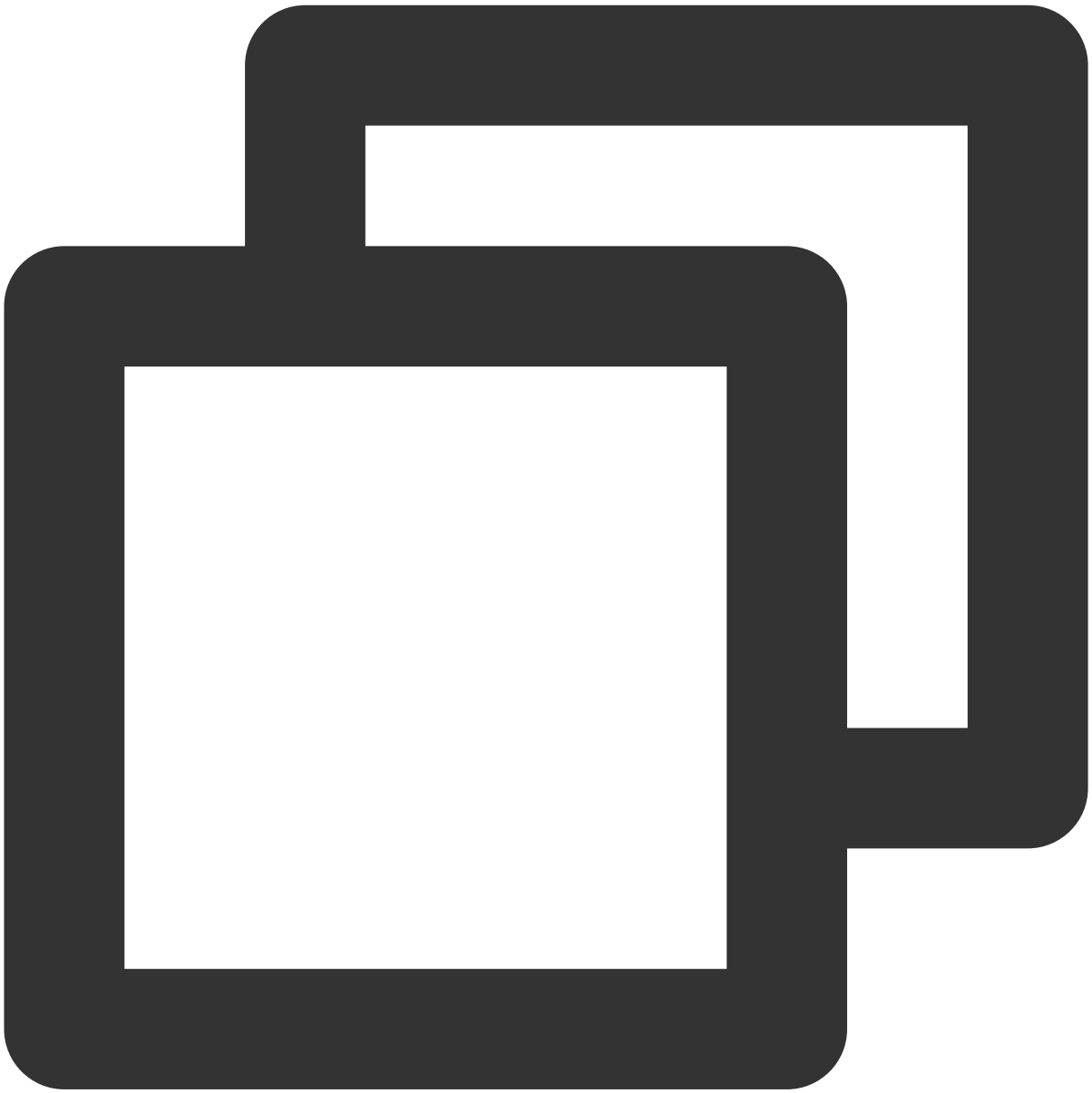
Key	Value	描述	示例
setVideoResolution	{ width: number; height:number; }	设置视频的分辨率	setProperty('setVideoResolution', { width: 1920, height: 1080 })
setVideoFPS	number	设置视频的帧率	setProperty('setVideoFPS', 25)
setVideoBitrate	number	设置视频的码率	setProperty('setVideoBitrate', 2000)
setAudioSampleRate	number	设置音频的采样率	setProperty('setAudioSampleRate', 44100)
setAudioBitrate	number	设置音频的码率	setProperty('setAudioBitrate', 200)
setConnectRetryCount	number	设置连接重试次数，默认值：3；取值范围：0 - 10。当 SDK 与服务器异常断开连接时，SDK 会尝试与服务器重连。	setProperty('setConnectRetryCount', 5)
setConnectRetryDelay	number	设置连接重试延迟，默认值：1，单位为秒；取值范围：0 - 10。当 SDK 与服务器异常断开连接时，SDK 会尝试与服务器重连。	setProperty('setConnectRetryDelay', 2)
enableAudioAEC	boolean	启用回声消除	setProperty('enableAudioAEC', true)
enableAudioAGC	boolean	启用自动增益	setProperty('enableAudioAGC', true)
enableAudioANS	boolean	启用噪声抑制	setProperty('enableAudioANS', true)
enableLog	boolean	是否在控制台打印日志	setProperty('enableLog', true)

说明：

1. 回声消除、自动增益和噪声抑制默认全部启用，最终是否起效依赖于设备和浏览器。这三个功能建议要么全部启用，要么全部禁用。
2. 请在采集流和推流之前进行设置。

startCamera

打开摄像头设备。需要用户授权允许浏览器访问摄像头，授权失败或者访问设备失败，返回的 Promise 对象会抛出错误。



```
startCamera(deviceId?: string): Promise<string>;
```

参数：

字段	类型	说明

deviceId	string	摄像头设备 ID，可选参数，指定打开的摄像头设备。设备 ID 可通过 `TXDeviceManager` 中的方法 getDevicesList() 获取。在移动设备上，可以通过传入 'user' 和 'environment' 来指定打开前置和后置摄像头。

返回：

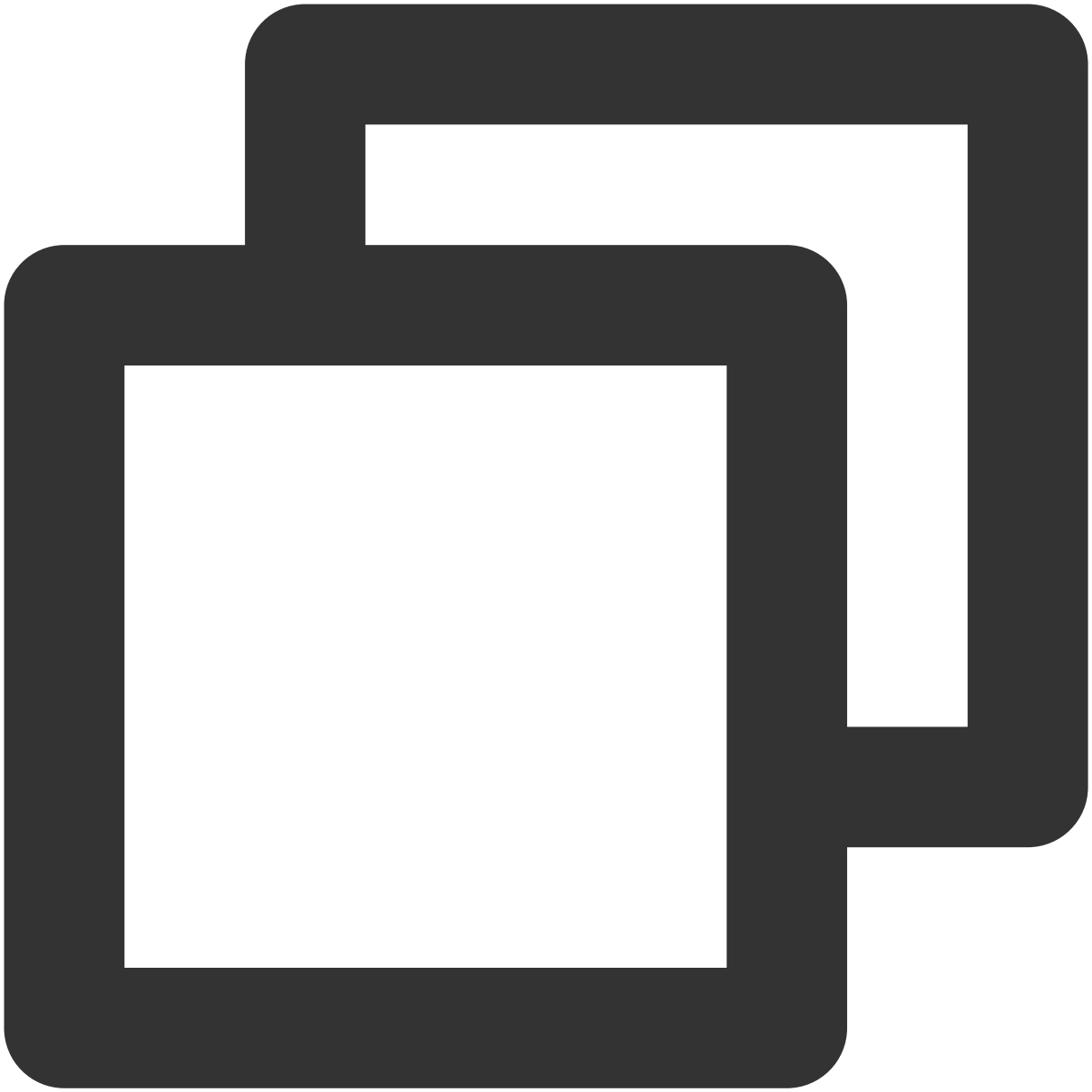
返回 Promise 对象，成功时返回流 ID 作为流在 SDK 内部的唯一标识，失败时抛出对应的错误信息。

说明：

1. 该接口不支持在 http 协议下使用，请使用 https 协议部署您的网站。
2. 打开摄像头失败时返回的错误信息，可参考 getUserMedia 异常 。

stopCamera

关闭摄像头设备。



```
stopCamera(streamId?: string): void;
```

参数：

字段	类型	说明
streamId	string	流 ID，可选参数，指定需要关闭的摄像头流。启用本地混流之后，如果采集了多路摄像头流，可通过流 ID 关闭指定的摄像头流，否则关闭所有的摄像头流。

startMicrophone

打开麦克风设备。需要用户授权允许浏览器访问麦克风，授权失败或者访问设备失败，返回的 Promise 对象会抛出错误。

```
startMicrophone(deviceId?: string): Promise<string>;
```

参数：

字段	类型	说明

deviceId	string	麦克风设备 ID，可选参数，指定打开的麦克风设备。设备 ID 可通过 `TXDeviceManager` 中的方法 getDevicesList() 获取。

返回：

返回 Promise 对象，成功时返回流 ID 作为流在 SDK 内部的唯一标识，失败时抛出对应的错误信息。

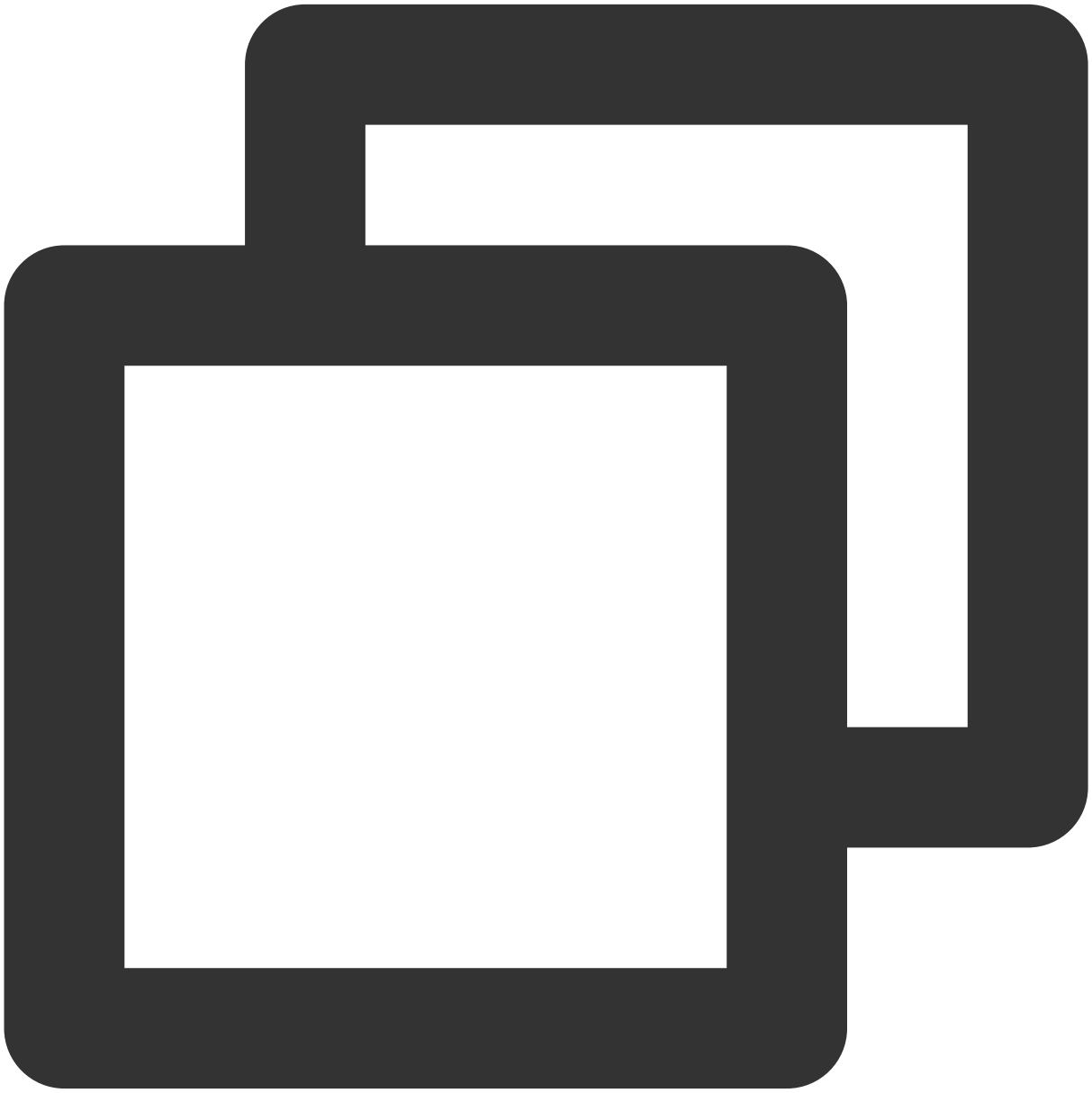
说明：

1. 该接口不支持在 http 协议下使用，请使用 https 协议部署您的网站。
2. 打开麦克风失败时返回的错误信息，可参考 getUserMedia 异常 。
3. 如果出现回声现象，可以将本地用于播放预览的视频元素 video 静音，避免回声现象的出现。

```
livePusher.videoView.muted = true;
```

stopMicrophone

关闭麦克风设备。



```
stopMicrophone(streamId?: string): void;
```

参数：

字段	类型	说明
streamId	string	流 ID，可选参数，指定需要关闭的麦克风流。启用本地混流之后，如果采集了多路麦克风流，可通过流 ID 关闭指定的麦克风流，否则关闭所有的麦克风流。

startScreenCapture

开启屏幕采集。需要用户授权允许浏览器访问屏幕，授权失败或者访问屏幕失败，返回的 Promise 对象会抛出错误。

```
startScreenCapture(audio?: boolean): Promise<string>;
```

参数：

字段	类型	说明
audio	boolean	是否采集系统声音或者标签页声音，true - 采集声音，false - 不采集声音，默认 false。

返回：

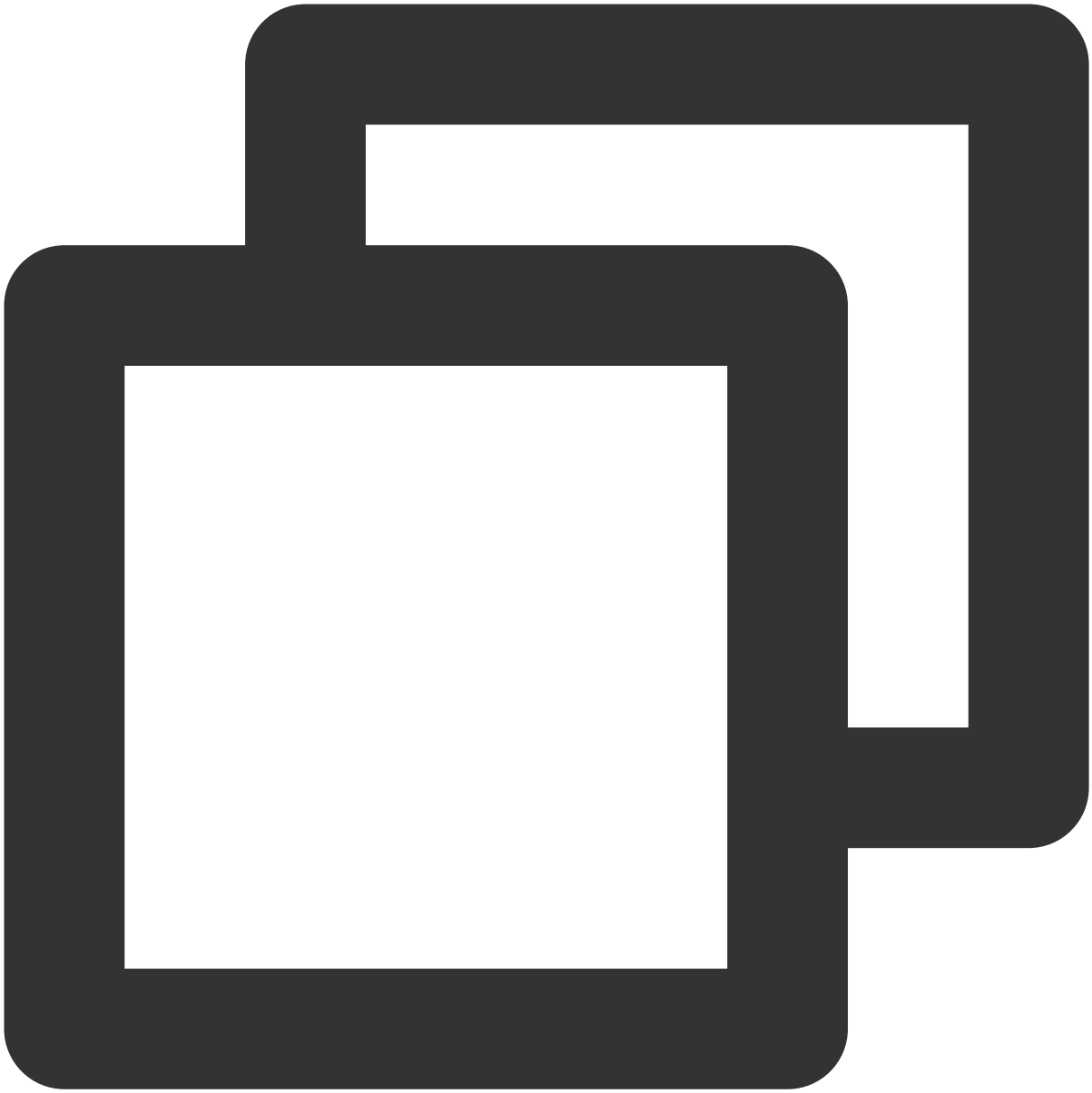
返回 Promise 对象，成功时返回流 ID 作为流在 SDK 内部的唯一标识，失败时抛出对应的错误信息。

说明：

1. 该接口不支持在 http 协议下使用，请使用 https 协议部署您的网站。
2. 打开屏幕采集失败时返回的错误信息，可参考 getDisplayMedia 异常 。
3. 目前只有 Chrome 74+ 和 Edge 79+ 支持采集声音，在 windows 系统可以采集整个系统的声音，在 Linux 和 Mac 上面只能采集标签页的声音。
4. 如果设置了 audio 为 true，在浏览器的屏幕分享弹窗中还需要确保弹窗最下面的采集声音选项是勾选状态，否则也不会采集声音。如果设置了 audio 为 false，浏览器的屏幕分享弹窗中不会出现采集声音的选项。

stopScreenCapture

关闭屏幕采集。



```
stopScreenCapture(streamId?: string): void;
```

参数：

字段	类型	说明
streamId	string	流 ID，可选参数，指定需要关闭的屏幕分享流。启用本地混流之后，如果采集了多路屏幕分享流，可通过流 ID 关闭指定的屏幕分享流，否则关闭所有的屏幕分享流。

版权所有：腾讯云计算（北京）有限责任公司

startVirtualCamera

开始采集本地媒体文件流。目前支持的文件格式有视频 mp4，音频 mp3 和图片 jpg、png、bmp。

```
startVirtualCamera(file: File): Promise<string>;
```

参数：

字段	类型	说明
file	File	本地媒体文件，必传。文件格式必须是以下几种：mp4、mp3、jpg、png、bmp。

返回：

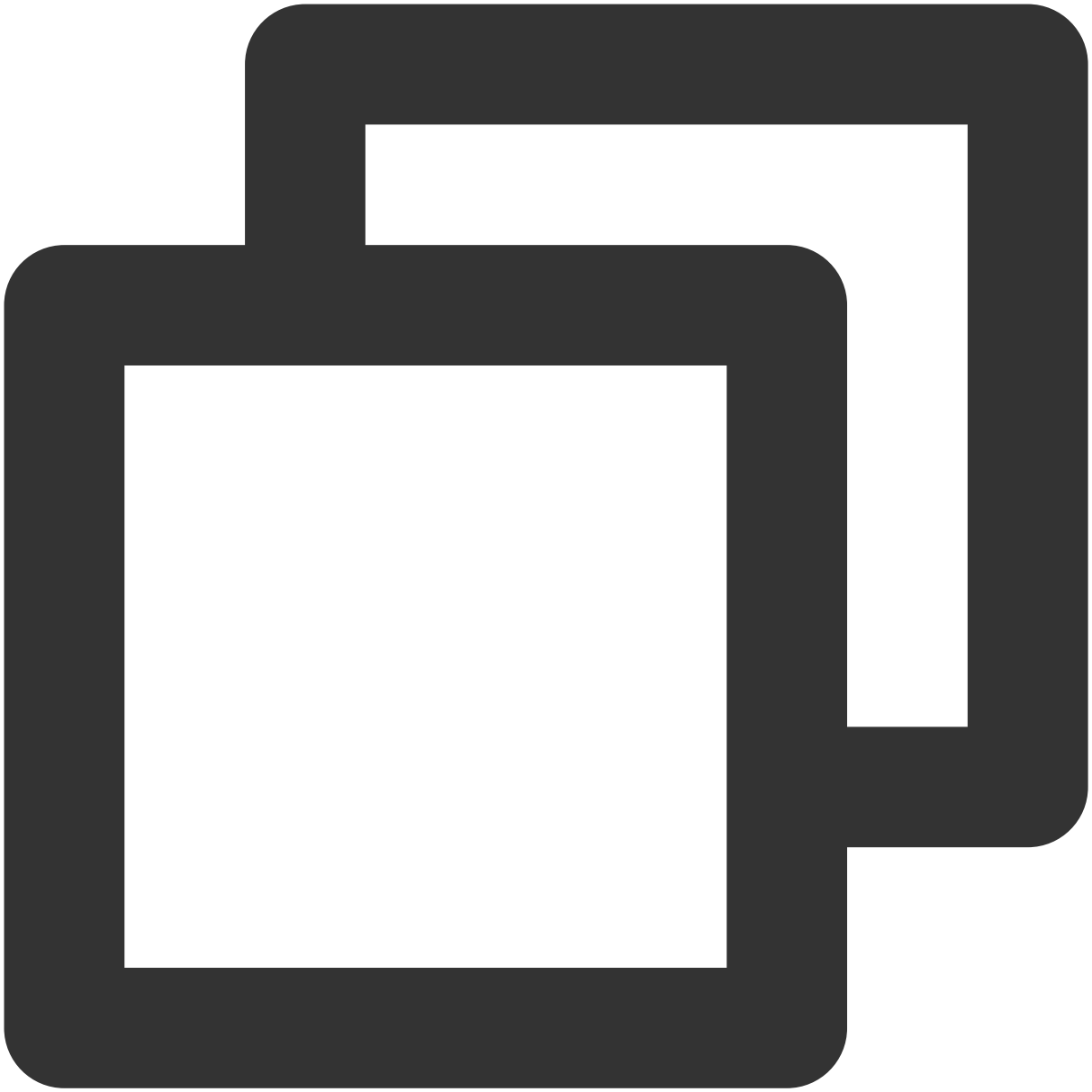
返回 Promise 对象，成功时返回流 ID 作为流在 SDK 内部的唯一标识，失败时抛出对应的错误信息。

说明：

1. 本地文件支持视频、音频和图片。视频文件采集视频流和音频流，音频文件只采集音频流，图片文件只采集视频流。
2. 必须手动传入 file 对象，需要提前使用 `<input type="file">` 引导用户选择本地文件。

stopVirtualCamera

停止采集本地媒体文件流。



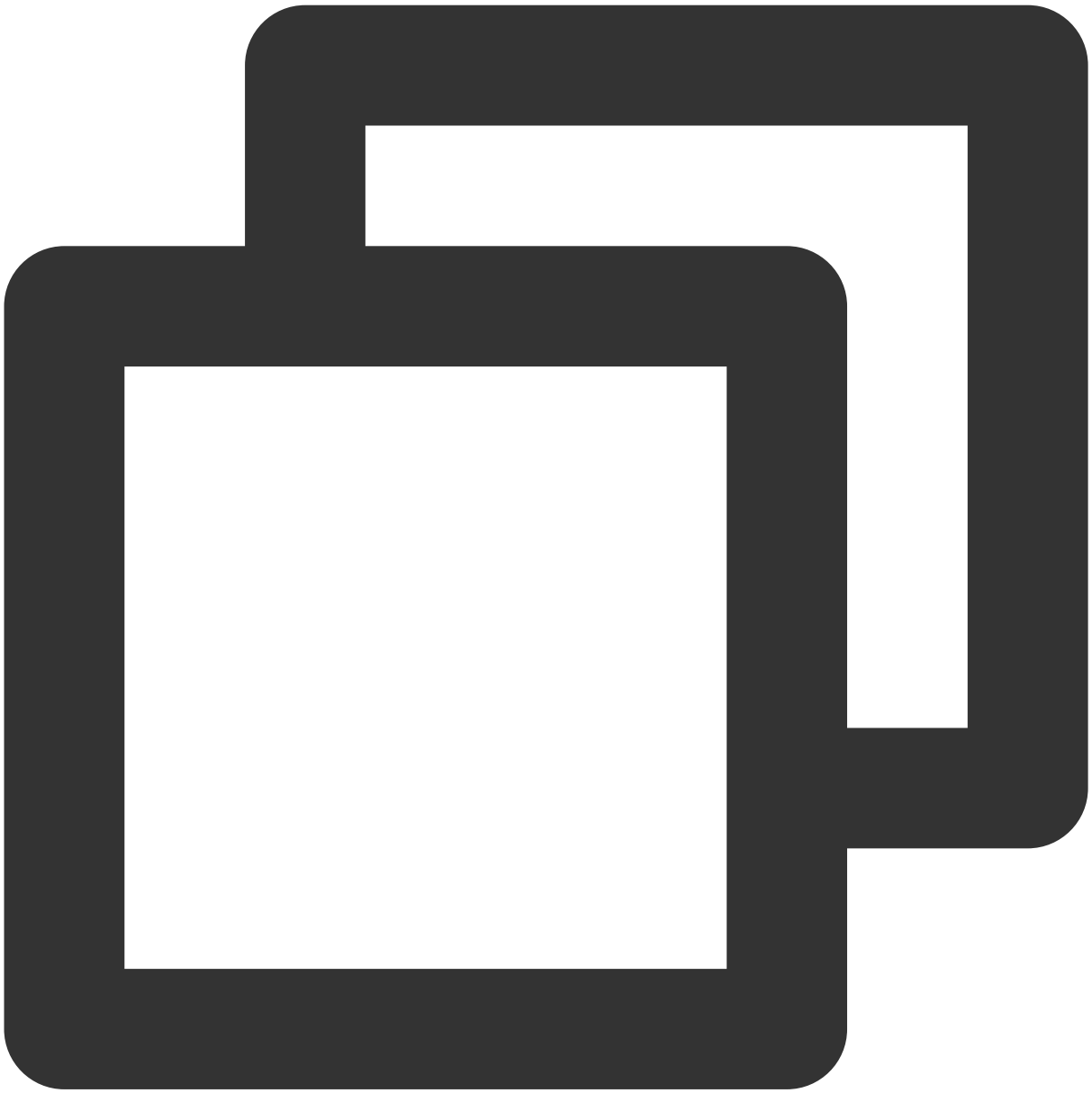
```
stopVirtualCamera(streamId?: string): void;
```

参数：

字段	类型	说明
streamId	string	流 ID，可选参数，指定需要关闭的媒体文件流。启用本地混流之后，如果采集了多路媒体文件流，可通过流 ID 关闭指定的媒体文件流，否则关闭所有的媒体文件流。

startCustomCapture

使用用户自定义的音视频流。将用户自己采集的流用于本地混流和推送。



```
startCustomCapture(stream: MediaStream): Promise<string>;
```

参数：

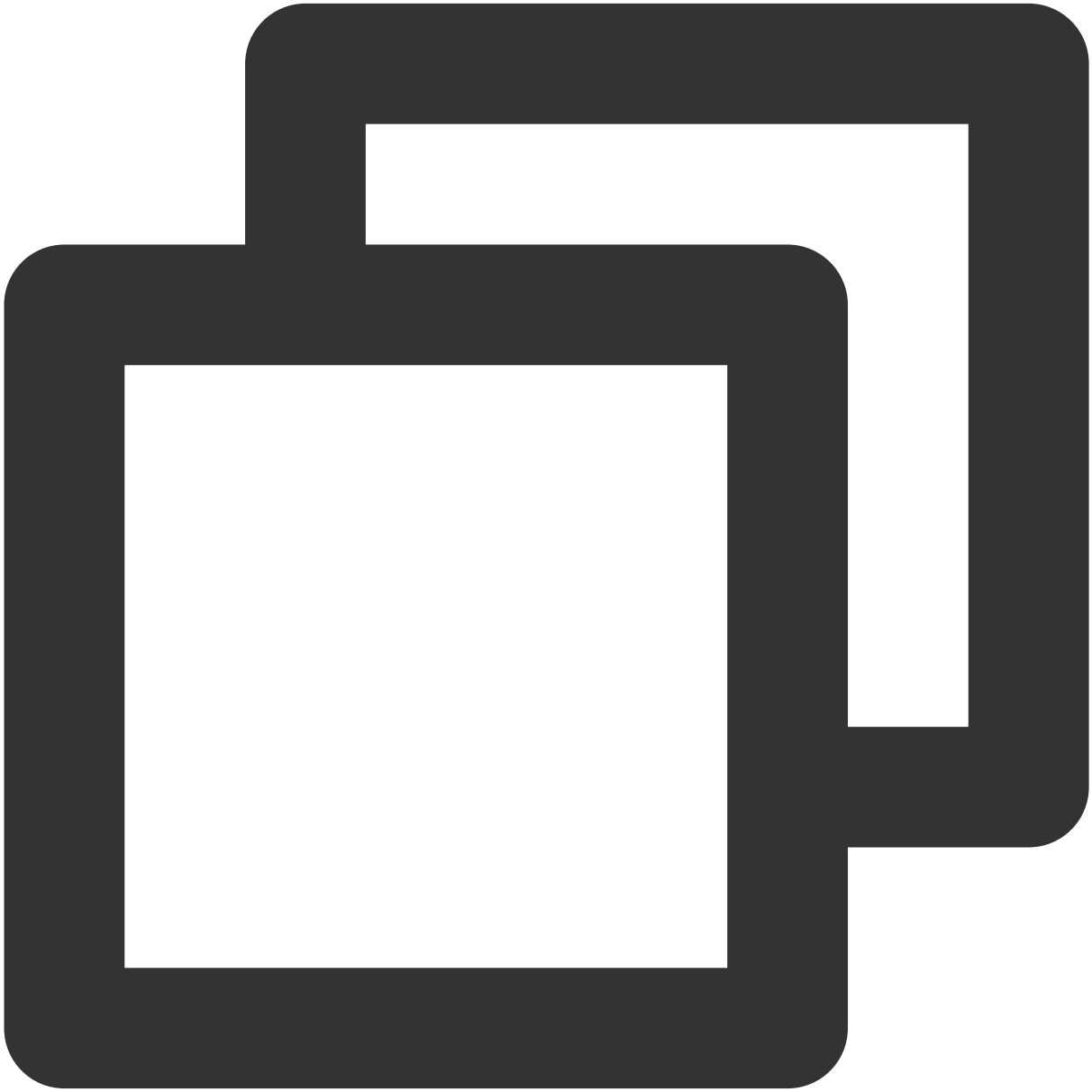
字段	类型	说明
stream	MediaStream	用户自定义的流。

返回：

返回 Promise 对象，成功时返回流 ID 作为流在 SDK 内部的唯一标识，失败时抛出对应的错误信息。

stopCustomCapture

关闭自定义的音视频流，仅移除自定义流，不会停止自定义流。



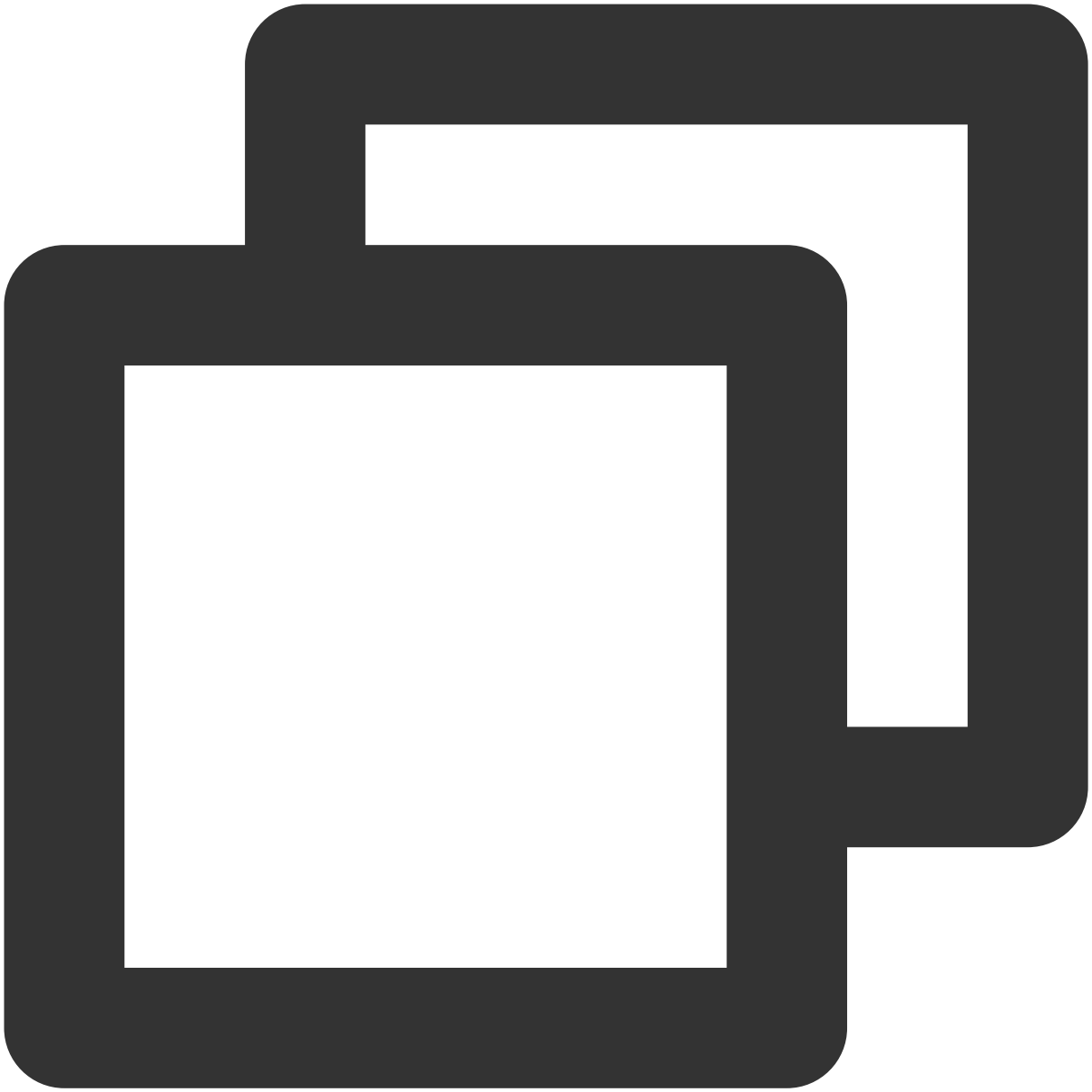
```
stopCustomCapture(streamId?: string): void;
```

参数：

字段	类型	说明
streamId	string	流 ID，可选参数，指定需要移除的自定义流。启用本地混流之后，如果添加了多路自定义流，可通过流 ID 移除指定的自定义流，否则移除所有的自定义流。

startPush

开始推流，建立 WebRTC 连接，往腾讯云服务器推送音视频流。如果开启了本地混流功能，推送的则是混流处理之后的流数据。



```
startPush(pushUrl: string): Promise<void>;
```

参数：

字段	类型	说明
pushUrl	string	WebRTC 推流地址。

返回：

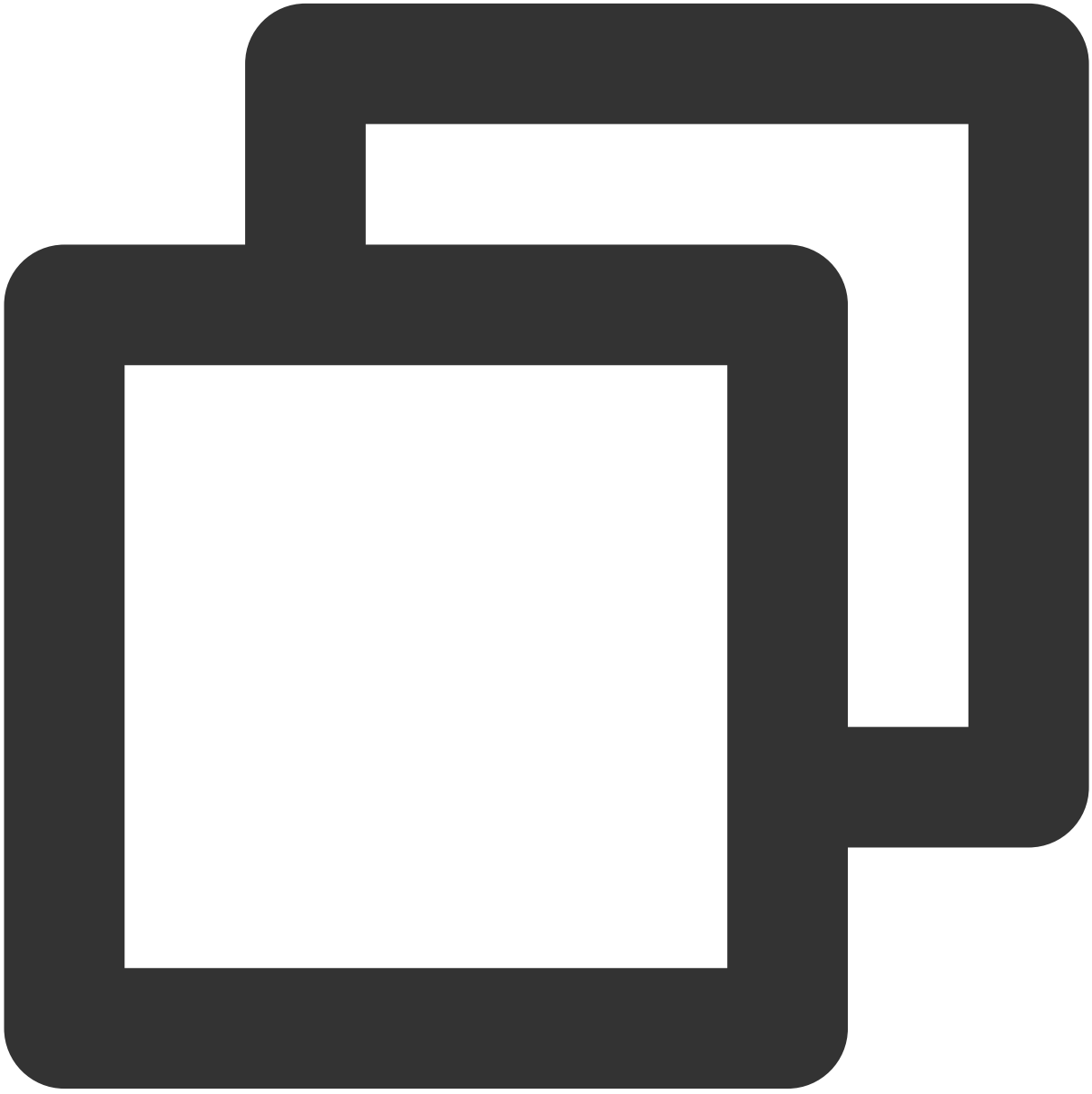
返回 Promise 对象。

说明：

推流地址的格式参考 拼装推流 URL 。

stopPush

停止推送音视频流，关闭 WebRTC 连接。



```
stopPush(): void;
```

isPushing

查询当前是否正在推流中。

```
isPushing(): boolean;
```

返回：

布尔值，true - 正在推流，false - 未推流。

getMediaStream

根据流 ID 获取采集到的音视频流。

```
getMediaStream(streamId: string): MediaStream;
```

参数：

字段	类型	说明
streamId	string	流 ID，由 startCamera() 、 startMicrophone() 、 startScreenCapture() 等接口调用成

		功之后返回。

返回：

采集到的流对象，可以通过传给 video 标签的 srcObject 属性进行播放。

getDeviceManager

获取设备管理对象。通过设备管理，可以进行查询设备列表，切换设备等操作。

```
getDeviceManager(): TXDeviceManager;
```

返回：

设备管理对象，具体用法请参考 TXDeviceManager 。

getVideoEffectManager

获取视频效果管理对象。通过视频效果管理，可以进行画中画、镜像、滤镜、水印、文本等操作。

```
getVideoEffectManager(): TXVideoEffectManager;
```

返回：

视频效果管理对象，具体用法请参考 TXVideoEffectManager 。

getAudioEffectManager

获取音频效果管理对象。通过音频效果管理，可以进行调整音量的操作。

```
getAudioEffectManager(): TXAudioEffectManager;
```

返回：

音频效果管理对象，具体用法请参考 TXAudioEffectManager 。

setVideoMute

设置是否禁用视频流。如果开启了本地混流功能，禁用的是最终生成的视频流。

```
setVideoMute(mute: boolean): void;
```

参数：

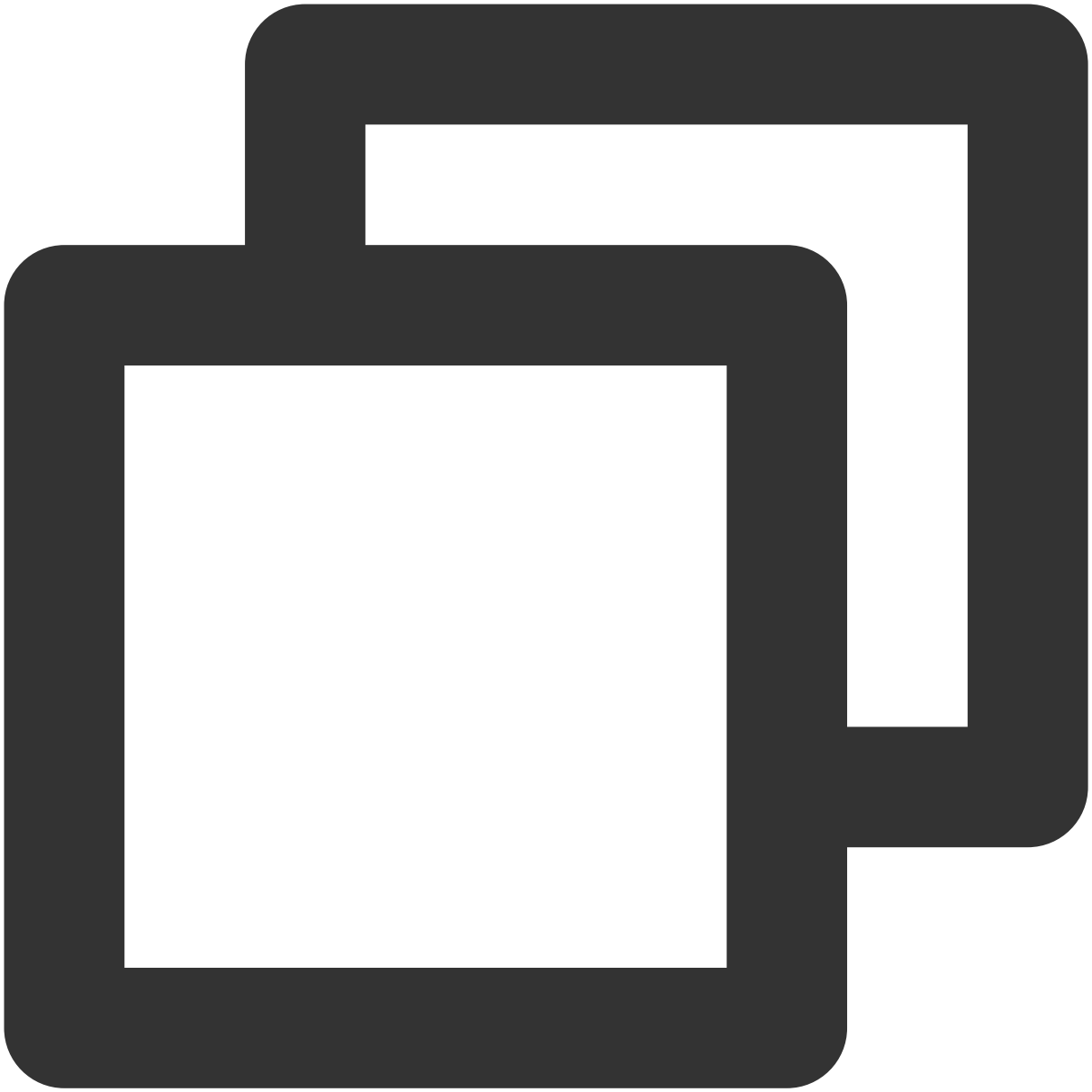
字段	类型	说明
mute	boolean	true - 禁用，false - 启用。

说明：

1. 当前有视频流时，设置才会生效。
2. 禁用之后每一帧都会用黑色像素填充，实际上仍在采集视频流。
3. 建议直接使用 pauseVideo() 和 resumeVideo() 。

setAudioMute

设置是否禁用音频流。如果开启了本地混流功能，禁用的是最终生成的音频流。



```
setAudioMute(mute: boolean): void;
```

参数：

字段	类型	说明
mute	boolean	true - 禁用，false - 启用。

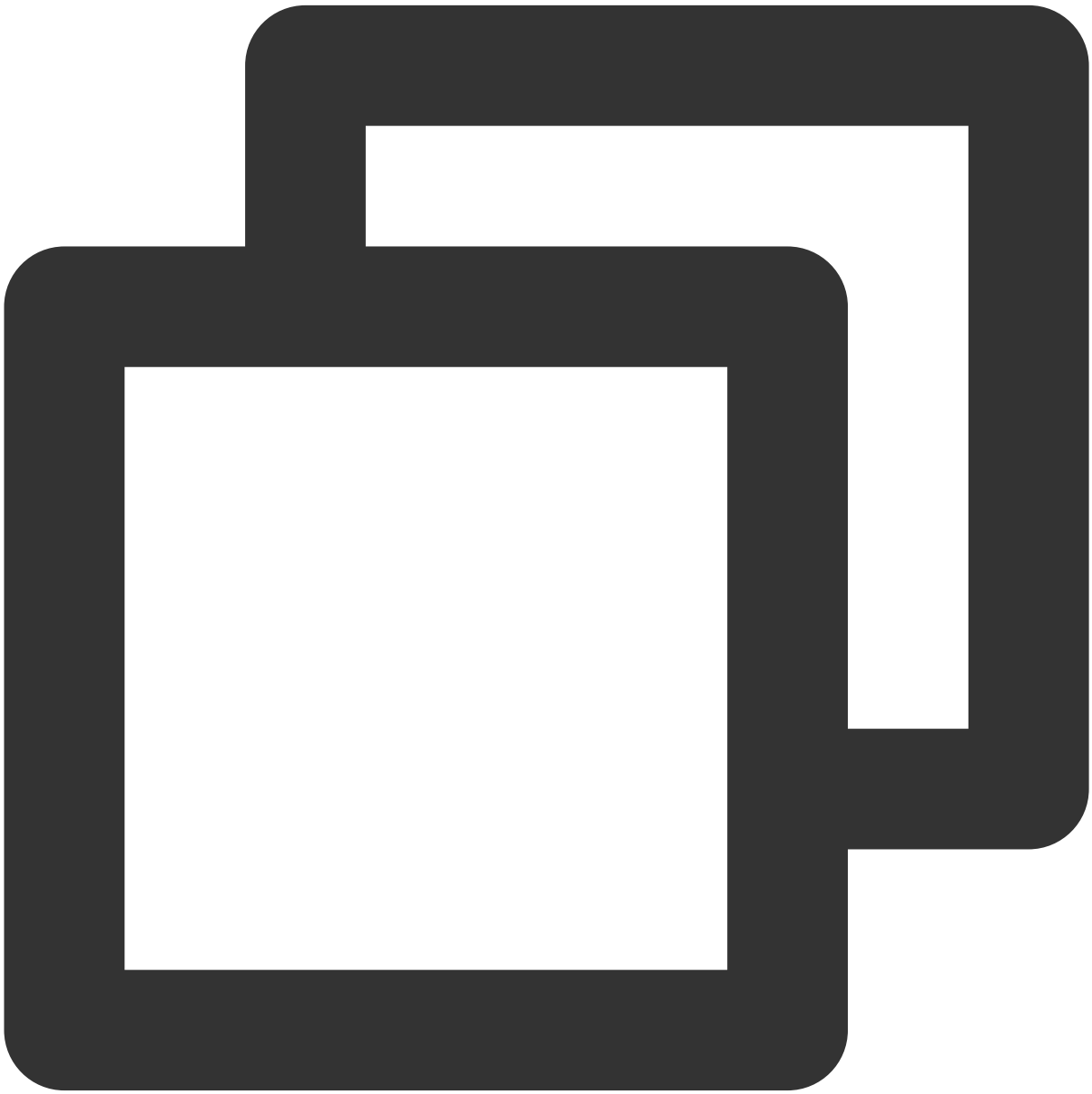
说明：

1. 当前有音频流时，设置才会生效。

2. 禁用之后是没有声音的，实际上仍在采集音频流。
3. 建议直接使用 pauseAudio() 和 resumeAudio() 。

pauseVideo

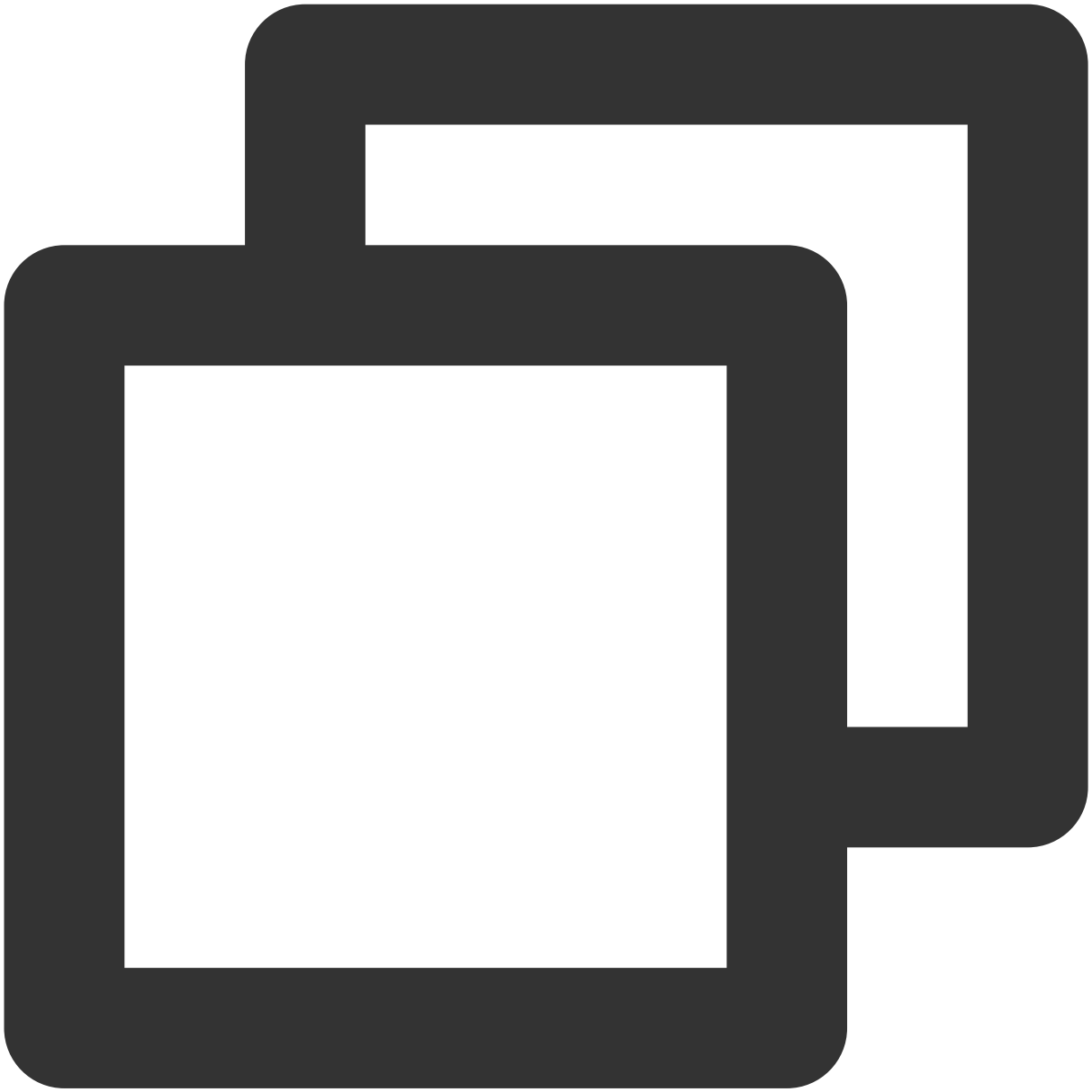
禁用视频流。等同于 `setVideoMute(true)` 。



```
pauseVideo(): void;
```

pauseAudio

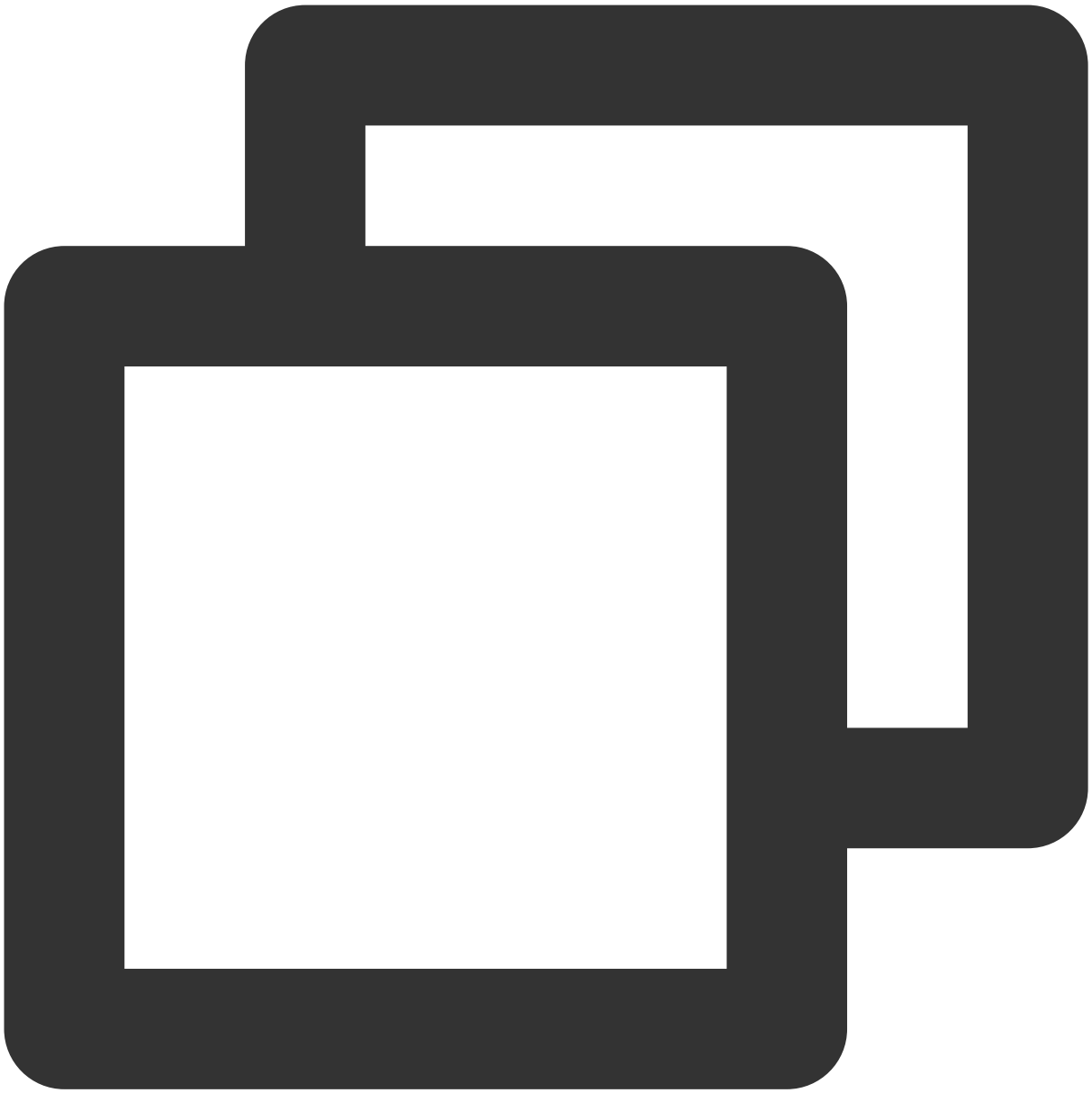
禁用音频流。等同于 `setAudioMute(true)` 。



```
pauseAudio(): void;
```

resumeVideo

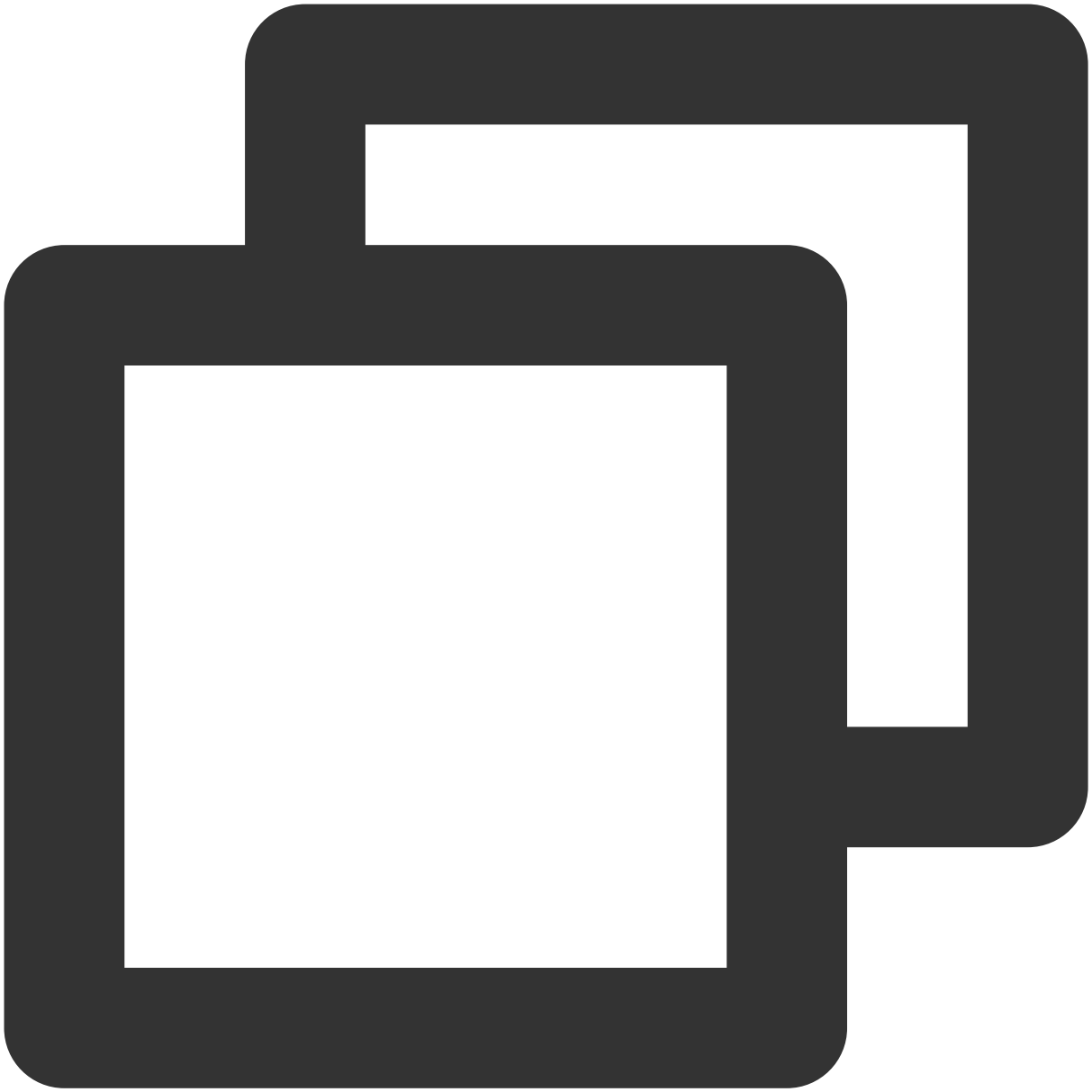
恢复视频流。等同于 `setVideoMute(false)` 。



```
resumeVideo(): void;
```

resumeAudio

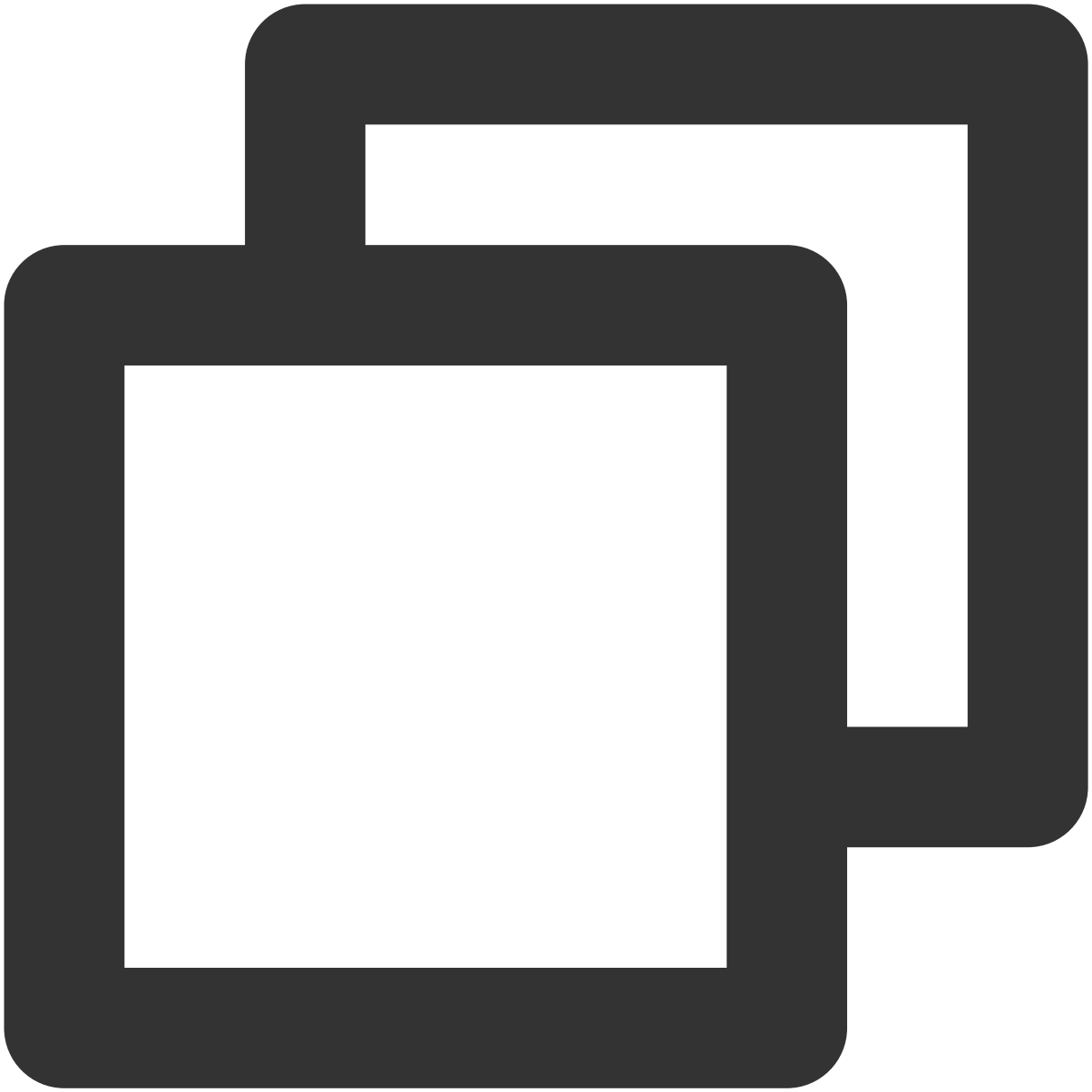
恢复音频流。等同于 `setAudioMute(false)` 。



```
resumeAudio(): void;
```

setVideoContentHint

设置视频内容提示，用于提升在不同内容场景下的视频编码质量。



```
setVideoContentHint(contentHint: string): void;
```

参数：

字段	类型	说明
contentHint	string	内容提示，参考 MediaStreamTrack.contentHint 。

内容提示取值范围如下：

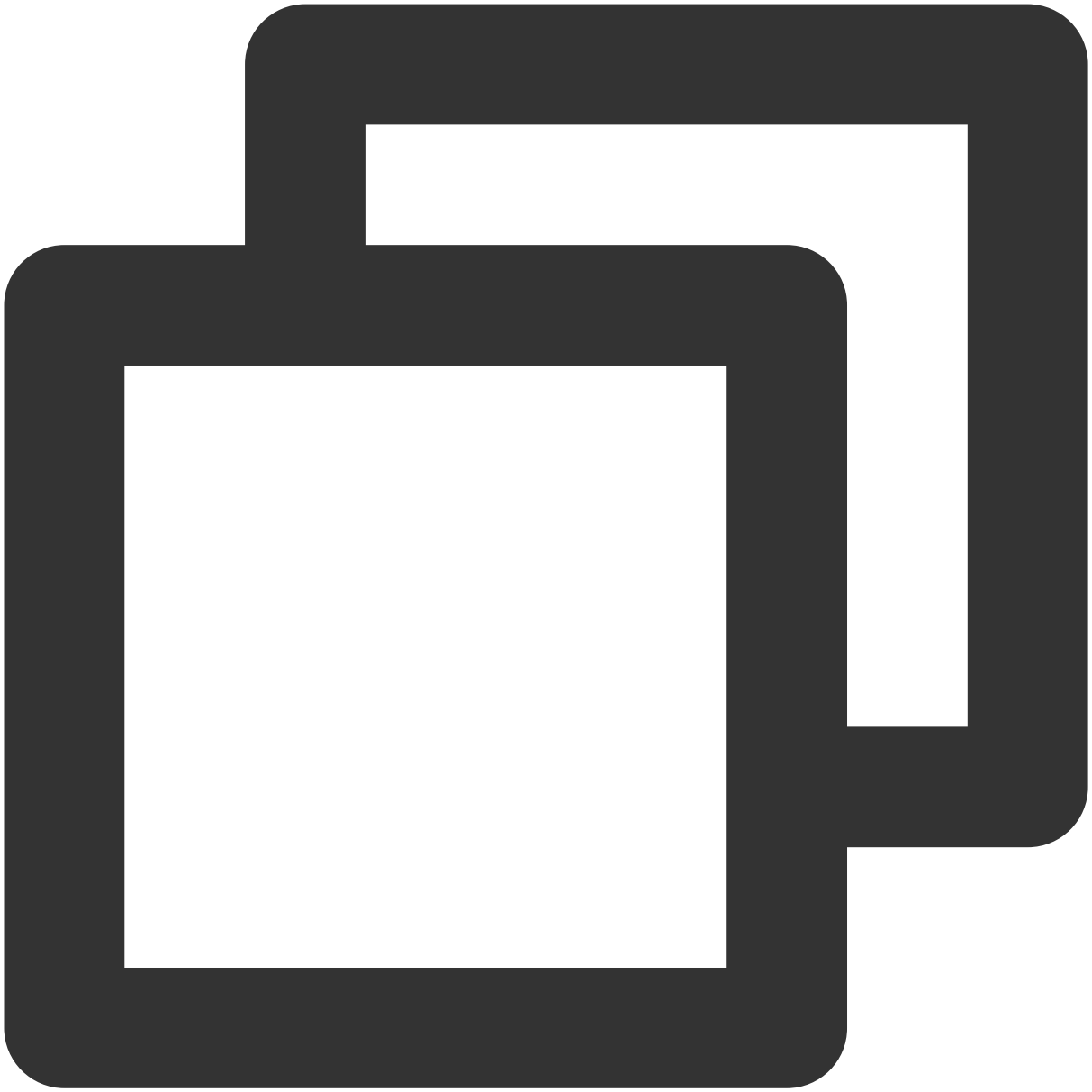
取值	说明

''	默认值，浏览器会自动评估视频内容，并选择合适的提示配置进行编码。
'motion'	表现为流畅度优先，用于视频内容为摄像头采集、电影、视频、游戏的情况。
'detail'	表现为清晰度优先，用于视频内容包含图片、文本混排的情况。在进行屏幕分享时，建议使用这个提示。
'text'	表现为清晰度优先，用于视频内容只包含大量文本的情况。

说明：
当前有视频流时，设置才会生效。

setObserver

设置推流事件回调通知。通过设置回调，可以监听推流的一些事件通知，包括推流状态、统计数据、警告和错误信息等。



```
setObserver(observer: TXLivePusherObserver): void;
```

参数：

字段	类型	说明
observer	TXLivePusherObserver	推流的回调目标对象。

说明：

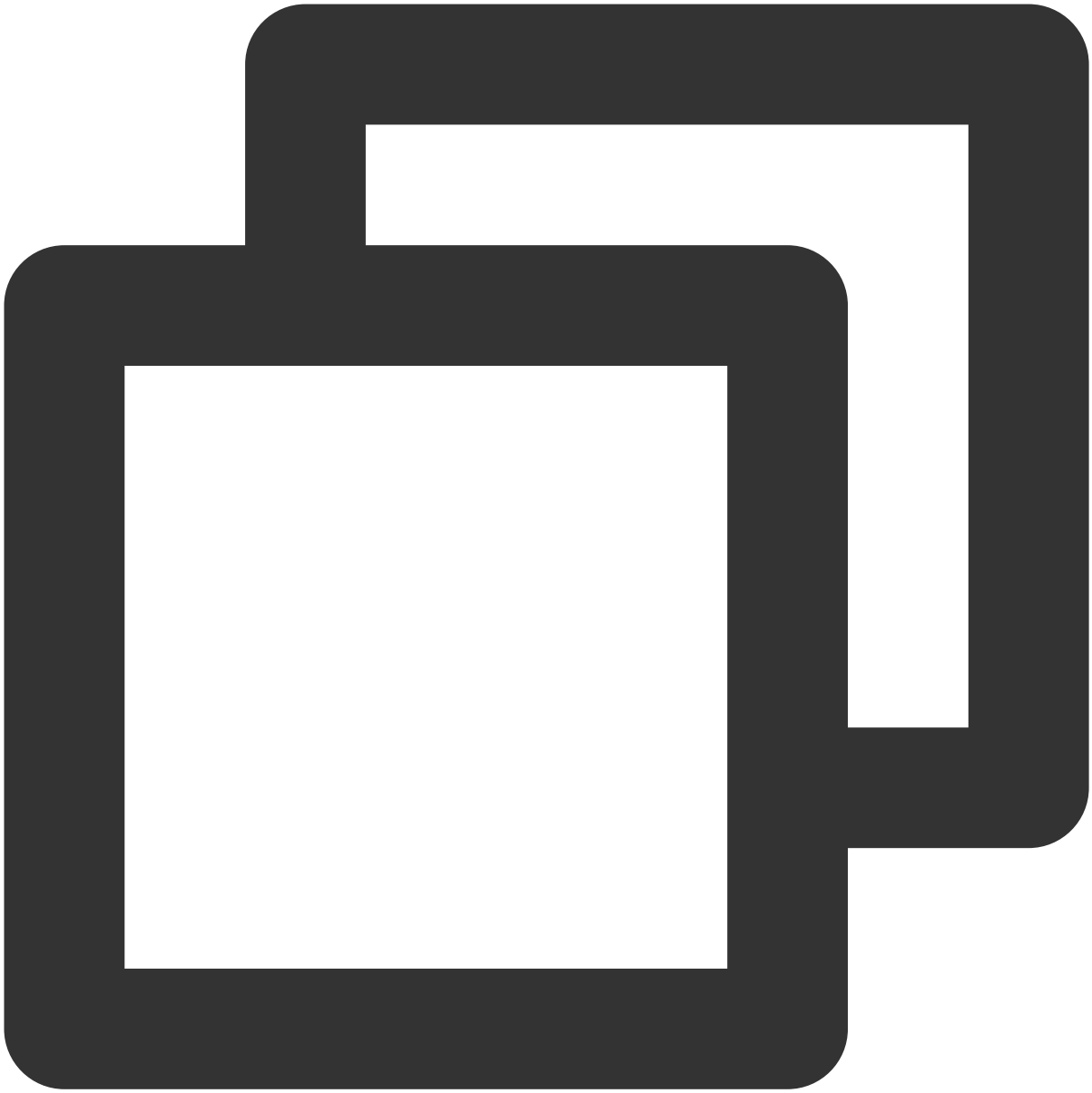
目前部分回调事件通知，比如 `onError` 、 `onWarning` 、 `onCaptureFirstAudioFrame` 、 `onCaptureFirstVideoFrame` 也可以通过调用对应接口返回的 Promise 对象来获取。用户可以根据自己的使用习惯，自由选择获取相应事件通知的方式。 例如：

`startCamera().then()` 等同于 `onCaptureFirstVideoFrame()` ，同样可以获取采集视频首帧成功的状态。

`startCamera().catch()` 等同于 `onWarning()` ，同样可以获取打开摄像头失败的错误。

destroy

离开页面或者退出时，清理 SDK 实例，避免可能会产生的内存泄露，调用前先执行 stop 相关的方法。

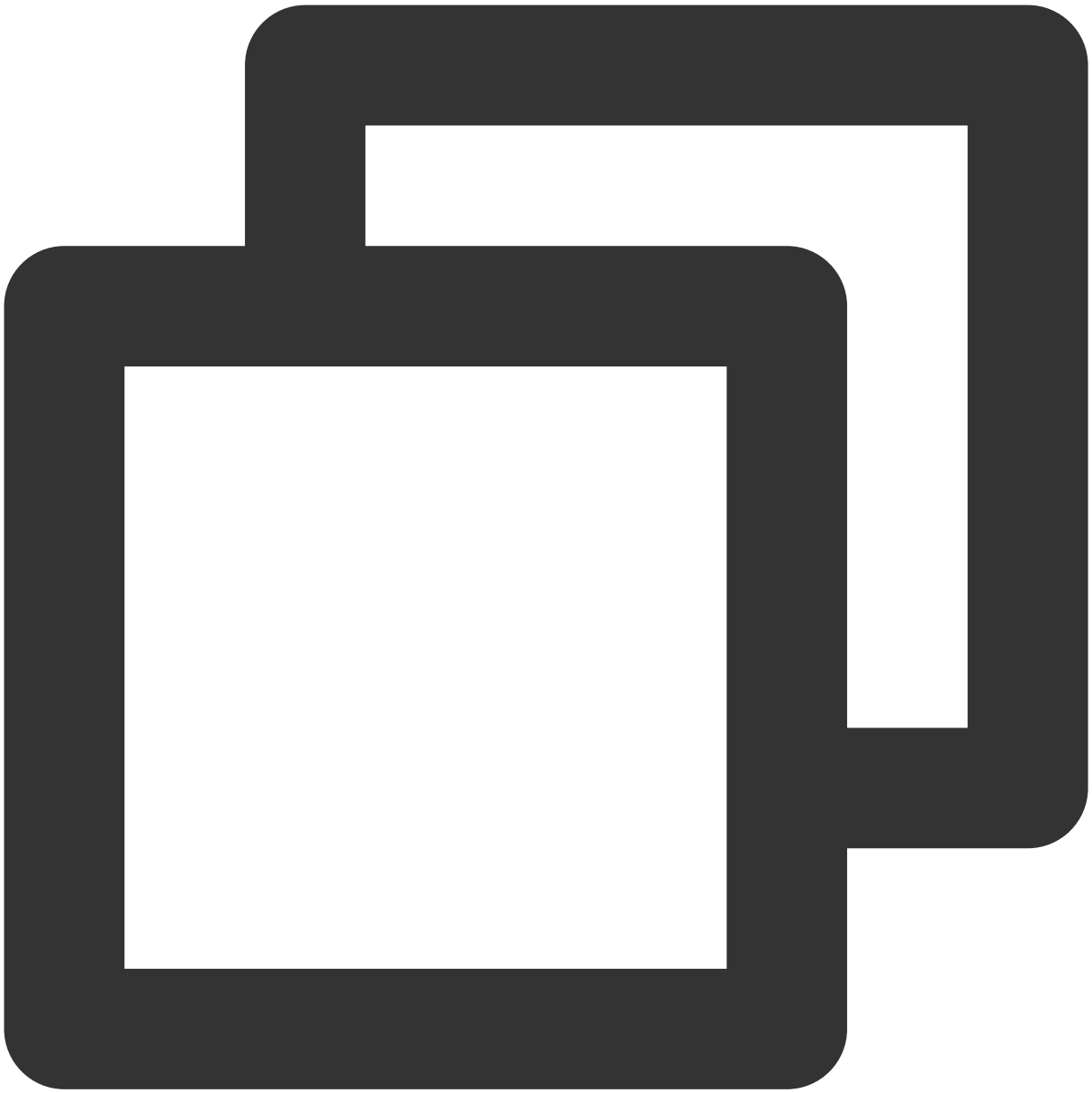


```
destroy(): void;
```

TXDeviceManager

设备管理接口。主要用于管理摄像头、麦克风设备。如果开启了本地混流功能，需要传入对应的流 ID 进行操作。

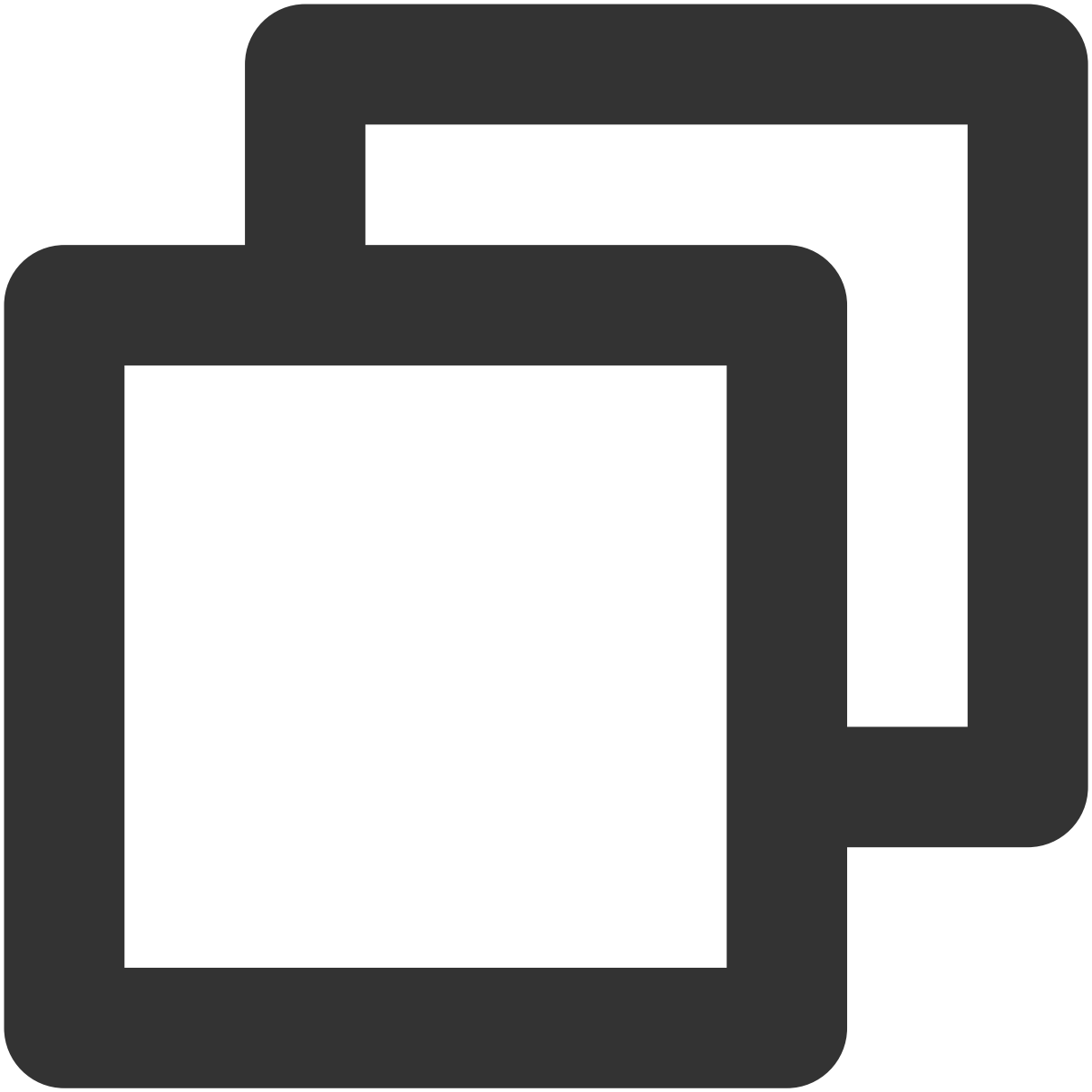
通过 `TXLivePusher` 方法 getDeviceManager() 来获取对象实例。



```
const deviceManager = livePusher.getDeviceManager();
```

getDevicesList

获取设备列表。



```
getDevicesList(type?: string): Promise<TXMediaDeviceInfo[]>;
```

参数：

字段	类型	说明
type	string	取值 **video** 或者 **audio**，可选参数。不传返回所有设备列表，传 video 返回摄像头设备列表，传 audio 返回麦克风设备列表。

返回：

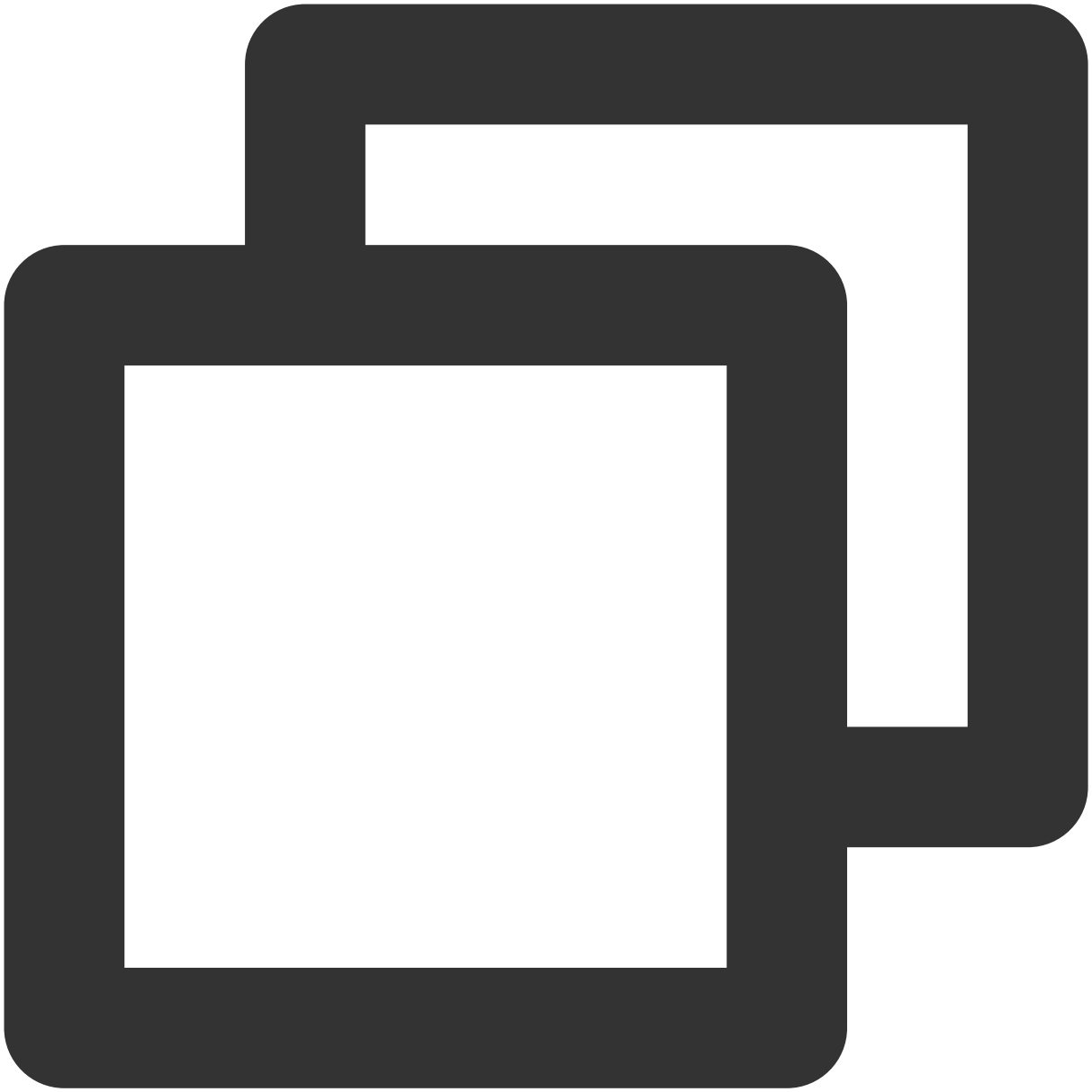
返回 Promise 对象，其中设备信息结构请参考 TXMediaDeviceInfo 。

说明：

1. 该接口不支持在 http 协议下使用，请使用 https 协议部署您的网站。
2. 浏览器出于安全的考虑，在用户未授权摄像头或麦克风访问权限前，deviceId 及 deviceName 字段可能都是空的。因此建议在用户授权访问后，再调用该接口获取设备详情。

getCurrentDevice

获取当前流的设备信息。如果启用了本地混流，必须指定流 ID。



```
getCurrentDevice(type: string, streamId?: string): Promise<TXMediaDeviceInfo>;
```

参数：

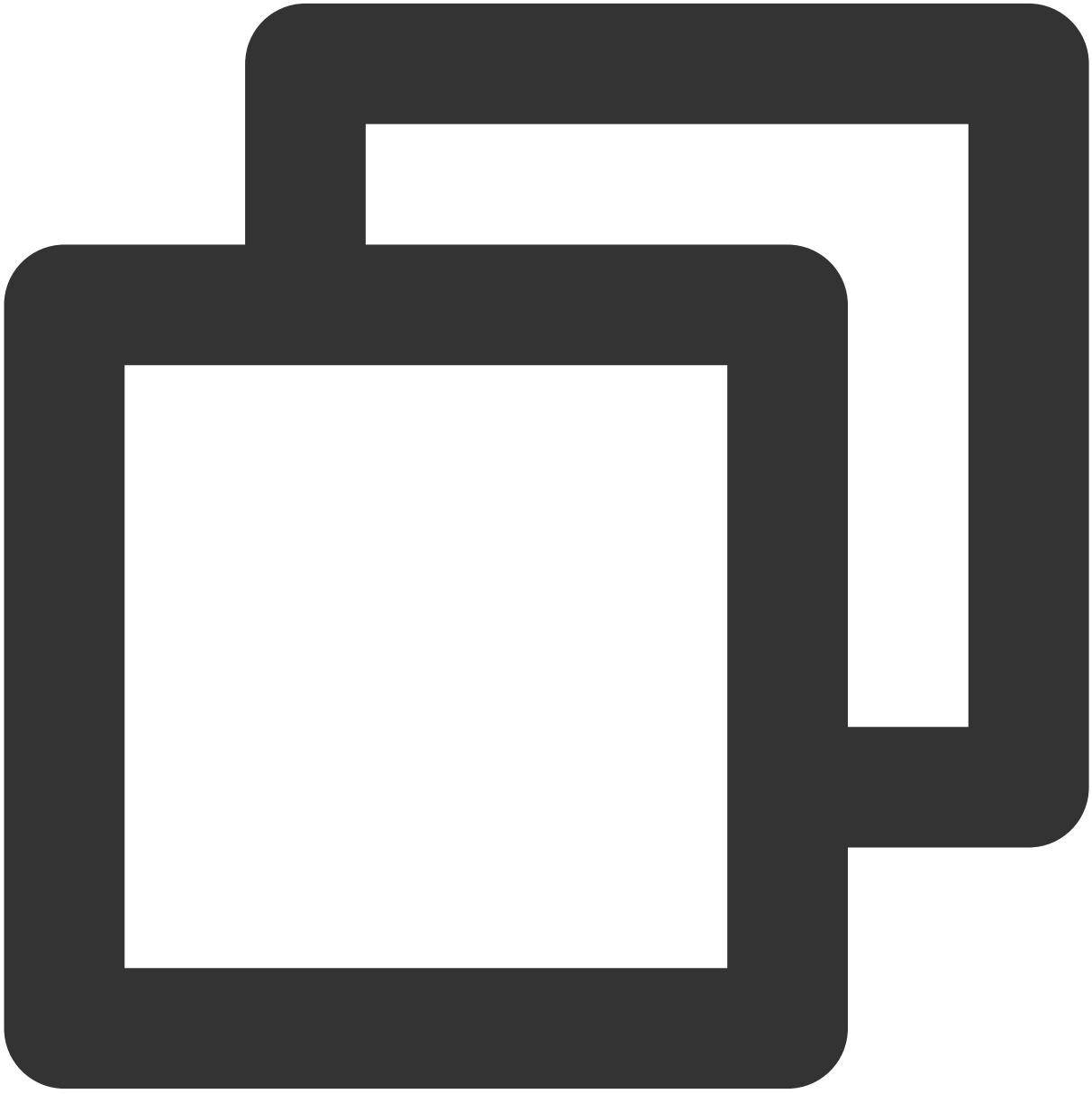
字段	类型	说明
type	string	设备类型：video - 摄像头设备，audio - 麦克风设备。
streamId	string	流 ID，指定要获取设备信息的流，启用本地混流后必须传，对应的流必须是摄像头或者麦克风设备采集的流。

返回：

返回 Promise 对象，其中设备信息结构请参考 TXMediaDeviceInfo 。

switchDevice

切换当前正在使用的设备。如果启用了本地混流，必须指定流 ID。



```
switchDevice(type: string, deviceId: string, streamId?: string): Promise<void>;
```

参数：

字段	类型	说明
type	string	设备类型：video - 摄像头设备，audio - 麦克风设备。
deviceId	string	设备 ID，可以通过调用 getDevicesList() 获取设备 ID。
streamId	string	流 ID，指定要切换设备的流，启用本地混流后必须传，对应的流必须是摄像头或者麦克风设备采集的流。

返回：

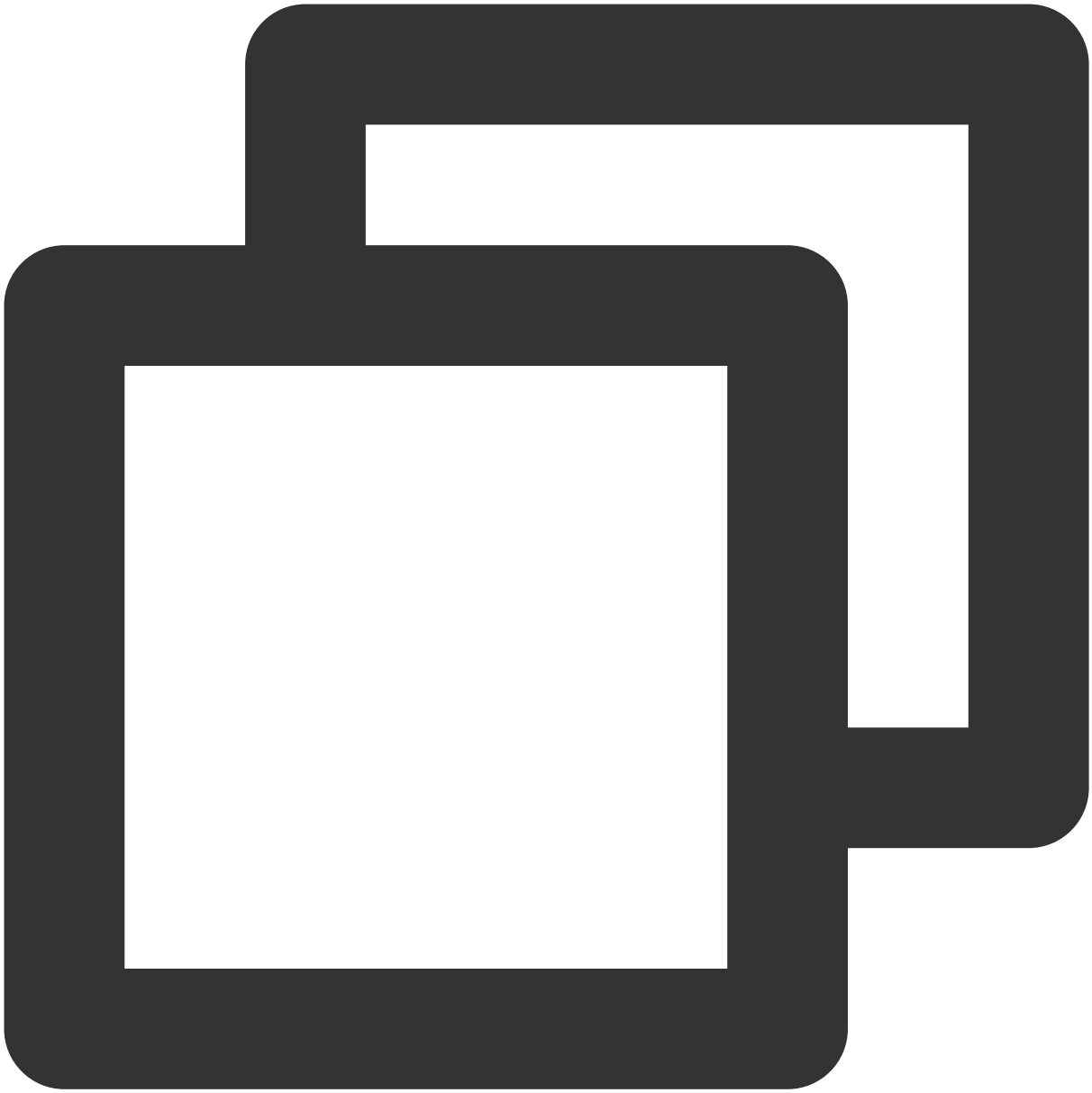
返回 Promise 对象。

说明：

1. 该方法仅适用于从摄像头和麦克风采集音视频时调用，其他采集方式采集的流不支持调用该接口。
2. 如果还没开始推流，则只更新本地流；如果已经开始推流，同步更新推到服务器的音视频流。
3. 切换流的设备时，对应流的 ID 不会发生变化。
4. 指定流 ID 时，对应流的类型必须和 type 匹配，比如 type 是 video，对应的流必须是摄像头采集的流。
5. 建议直接使用 switchCamera() 和 switchMicrophone() 。

switchCamera

切换摄像头设备。等同于 `switchDevice('video', deviceId, streamId)` 。



```
switchCamera(deviceId: string, streamId?: string): Promise<void>;
```

参数：

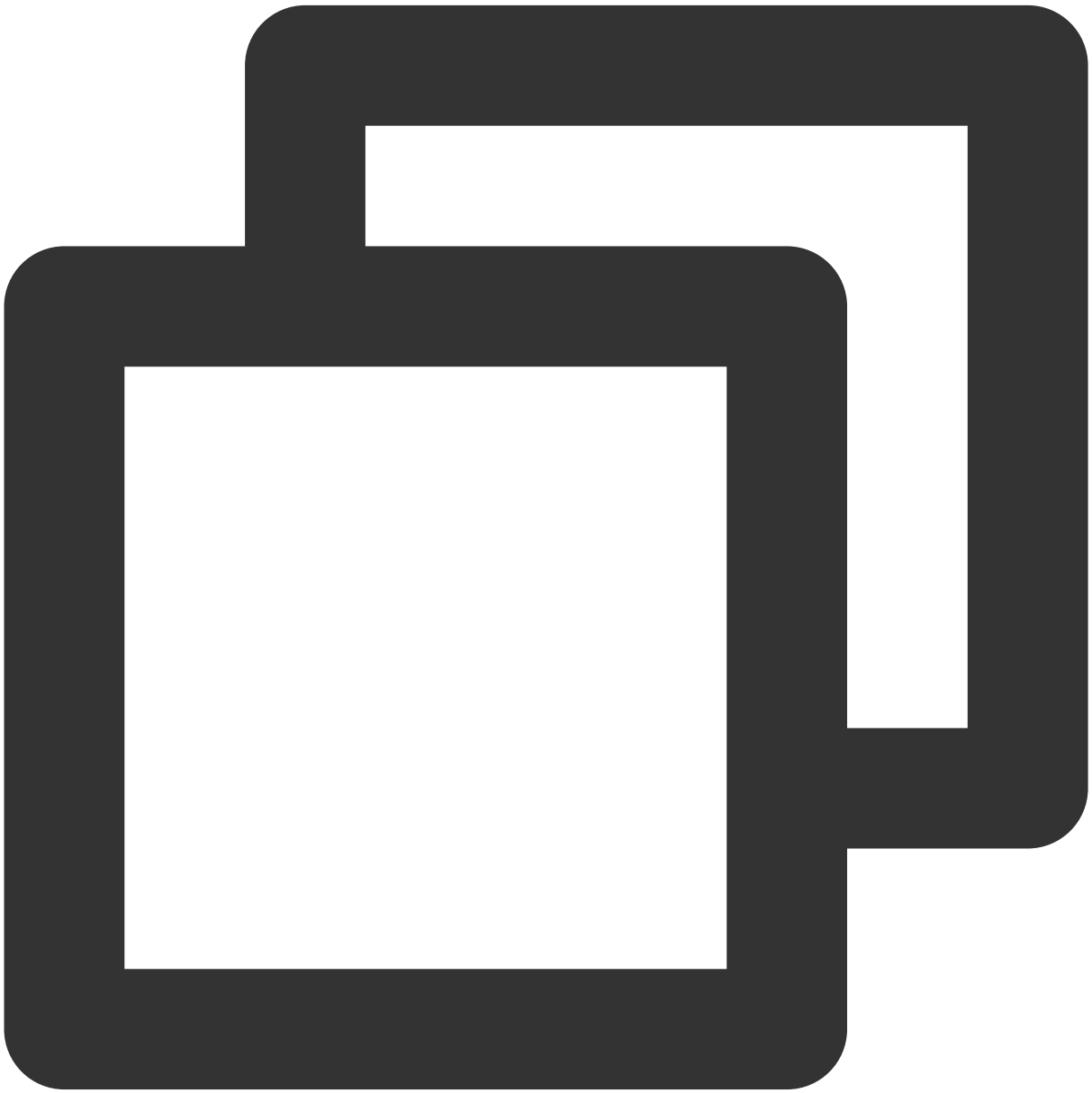
字段	类型	说明
deviceId	string	设备 ID，可以通过调用 getDevicesList() 获取设备 ID。在移动设备上，可以通过传入 'user' 和 'environment' 来切换前置和后置摄像头。
streamId	string	流 ID，指定要切换摄像头设备的流，启用本地混流后必须传，对应的流必须是摄像头设备采集的流。

返回：

返回 Promise 对象。

switchMicrophone

切换麦克风设备。等同于 `switchDevice('audio', deviceId, streamId)` 。



```
switchMicrophone(deviceId: string, streamId?: string): Promise<void>;
```

参数：

字段	类型	说明
deviceId	string	设备 ID ， 可以通过调用 getDevicesList() 获取设备 ID 。
streamId	string	流 ID，指定要切换麦克风设备的流，启用本地混流后必须传，对应的流必须是麦克风设备采集的流。

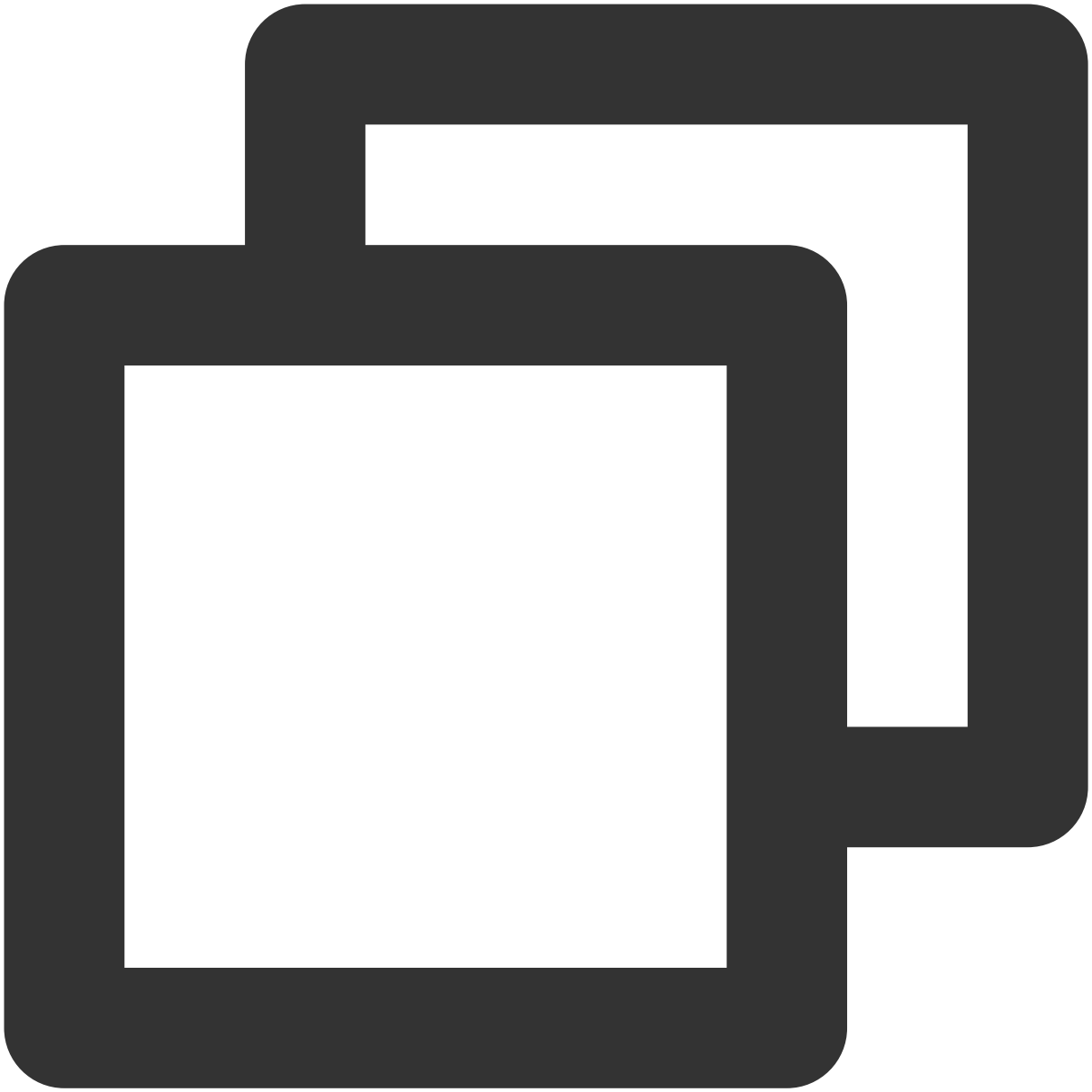
返回：

返回 Promise 对象。

TXAudioEffectManager

音频效果管理接口。主要用于调整音量的操作。如果开启了本地混流功能，需要传入对应的流 ID 进行操作。

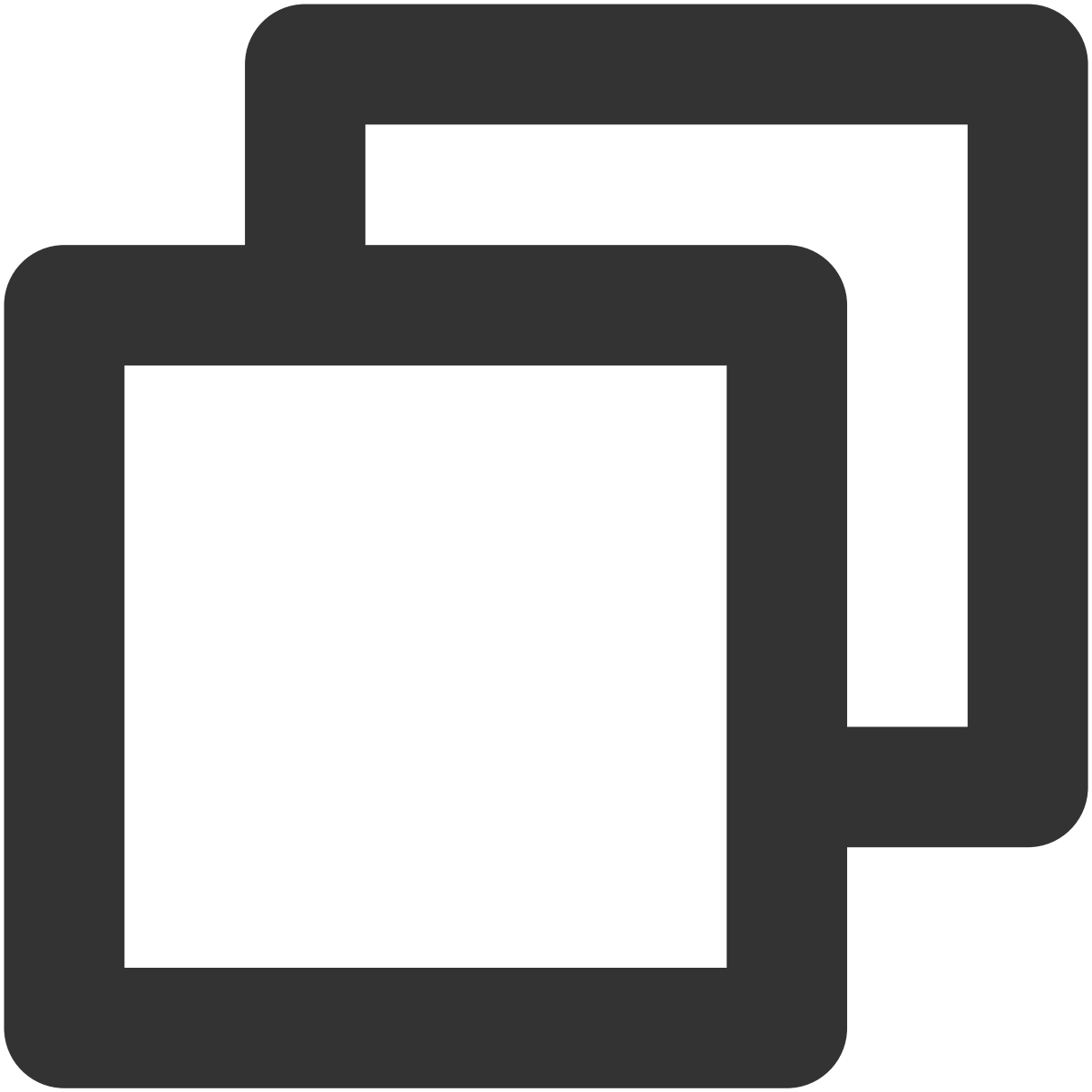
通过 `TXLivePusher` 方法 getAudioEffectManager() 来获取对象实例。



```
const audioEffectManager = livePusher.getAudioEffectManager();
```

setVolume

设置音频流的音量大小。如果启用了本地混流，必须指定流 ID。



```
setVolume(volume: number, streamId?: string): void;
```

参数：

字段	类型	说明
volume	number	音量大小，取值范围为 0 - 100，默认值是 100。
streamId	string	流 ID，指定要设置的流，启用本地混流后必须传。

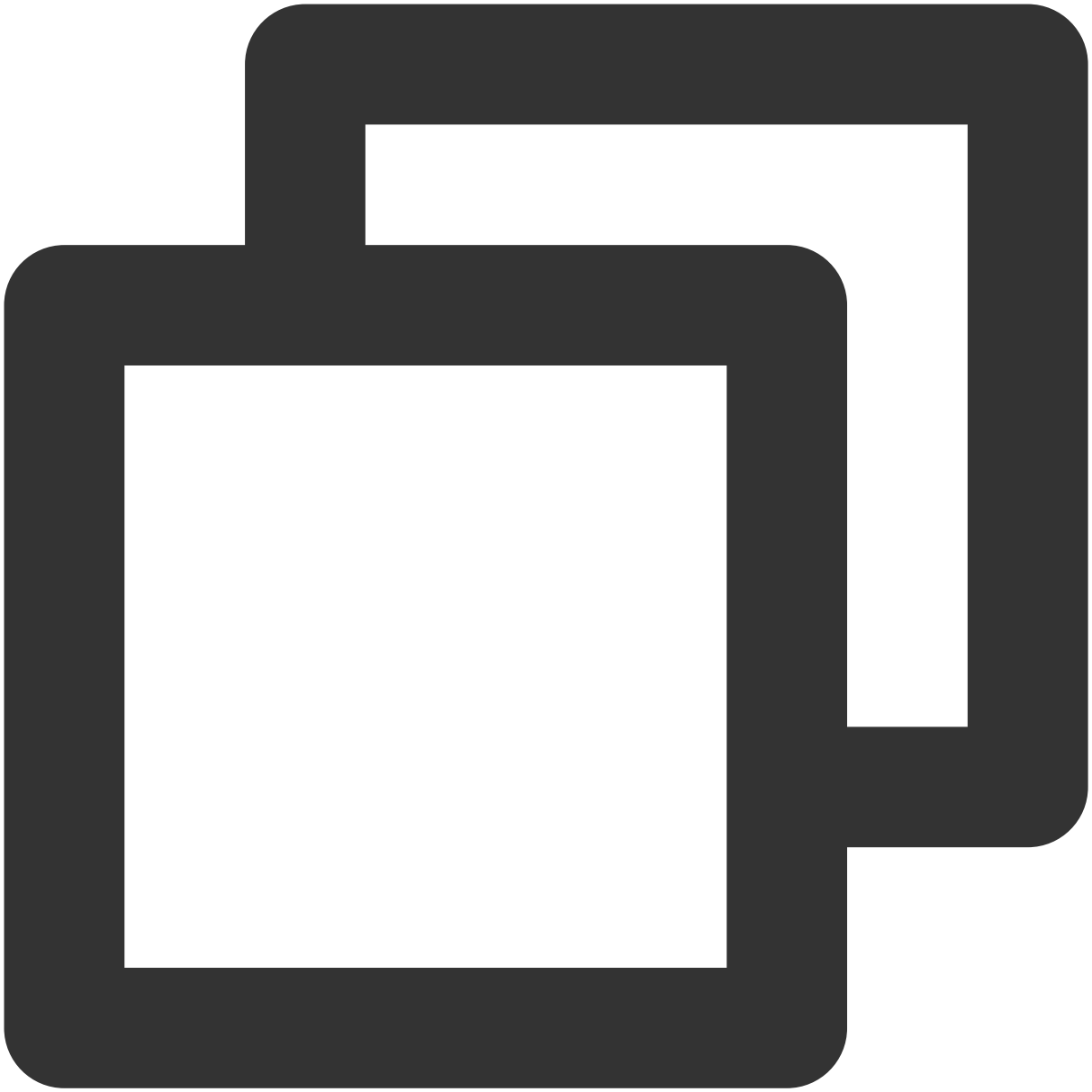
说明：

如果将 volume 设置成 100 之后感觉音量还是太小，可以将 volume 设置超过 100，但超过 100 的 volume 会有爆音的风险，请谨慎操作。

TXVideoEffectManager

视频效果管理接口。主要用于设置画中画、镜像、滤镜、水印、文本等操作。使用前需要先调用接口 enableMixing() 启用功能。

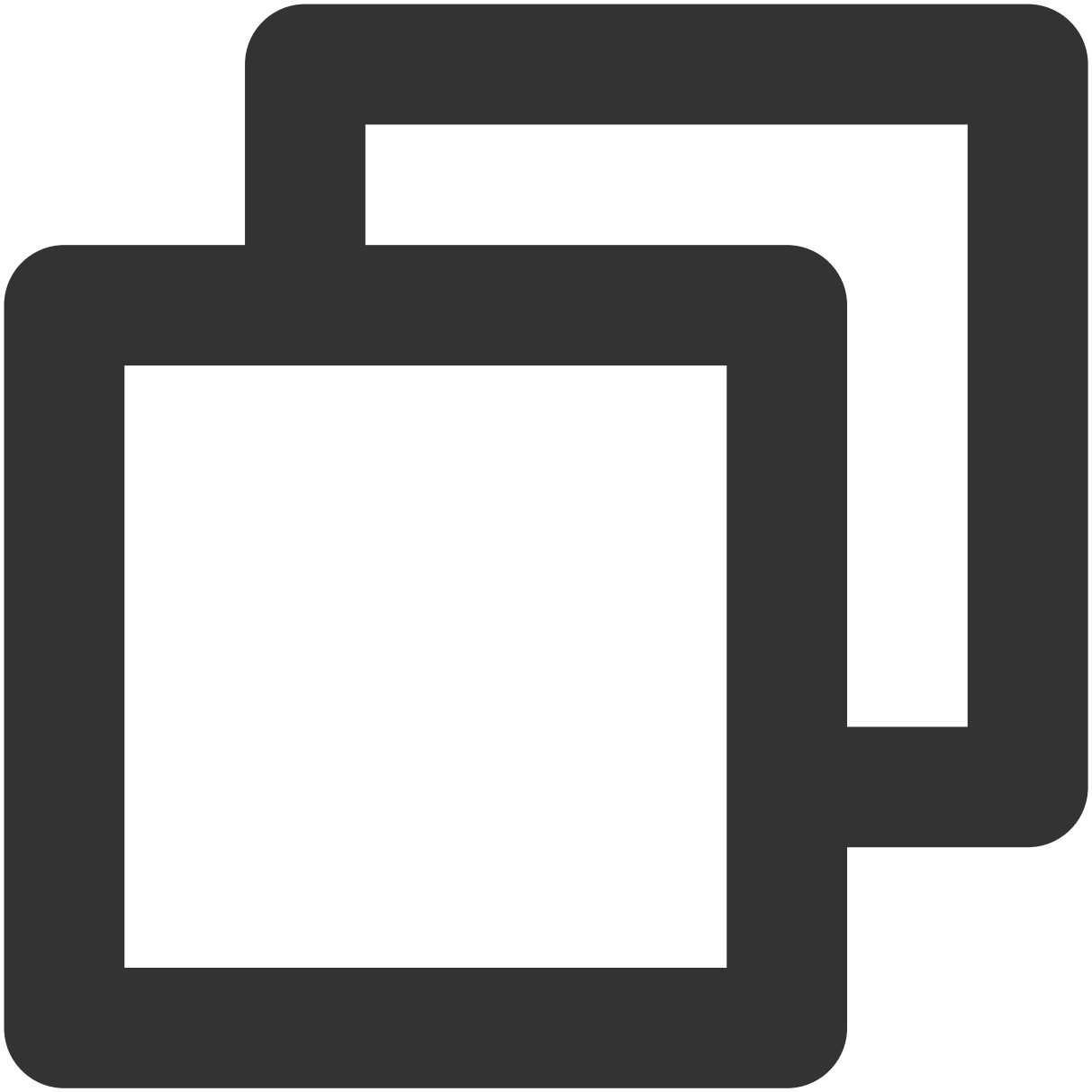
通过 `TXLivePusher` 方法 getVideoEffectManager() 来获取对象实例。



```
const videoEffectManager = livePusher.getVideoEffectManager();
```

enableMixing

开启本地视频画面混流功能。



```
enableMixing(enabled: boolean): void;
```

参数：

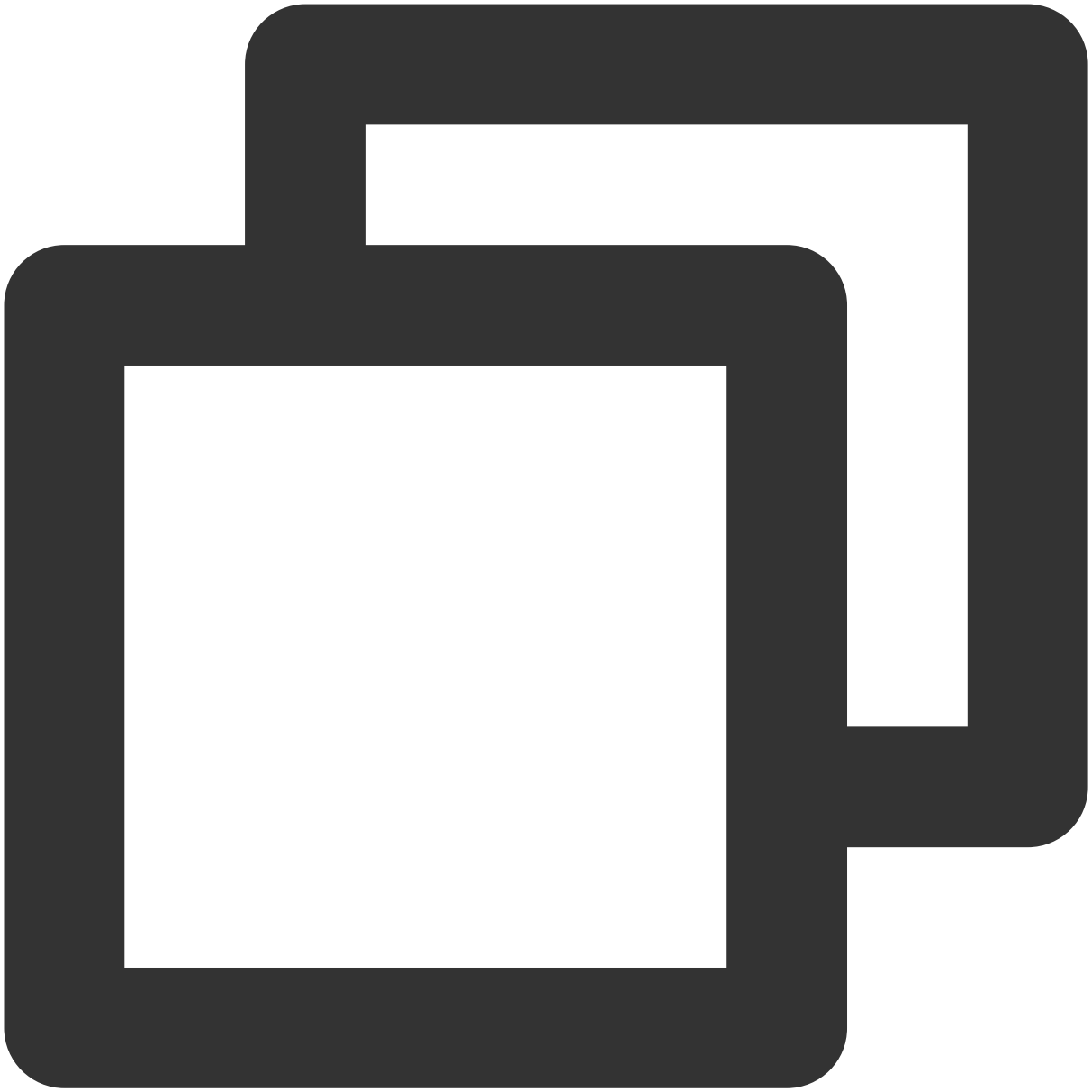
字段	类型	说明
enabled	boolean	是否开启本地混流功能，默认关闭。

说明：

1. 启用本地混流功能之前，预览和推送的流都是原始采集的流，启用之后，预览和推送的流都是经过浏览器本地混流处理之后的流，会有一定的浏览器性能开销。
2. 在调用 `TXVideoEffectManager` 其他方法前必须先调用该接口启用本地混流功能。
3. 没启用本地混流功能时，只能采集一路视频流和一路音频流。
4. 启用本地混流功能成功后，可以采集多路流，比如执行两个 `startCamera` 采集两个不同的摄像头画面，或者先执行 `startCamera` 采集摄像头画面，再执行 `startScreenCapture` 采集屏幕画面。采集的多路流的画面和声音都会出现在最终的混流输出结果中。

setMixingConfig

设置混流参数。不调用该接口进行设置时，默认混流参数直接使用 `TXLivePusher` 中方法 setVideoQuality() 和 setProperty() 设置之后的结果。



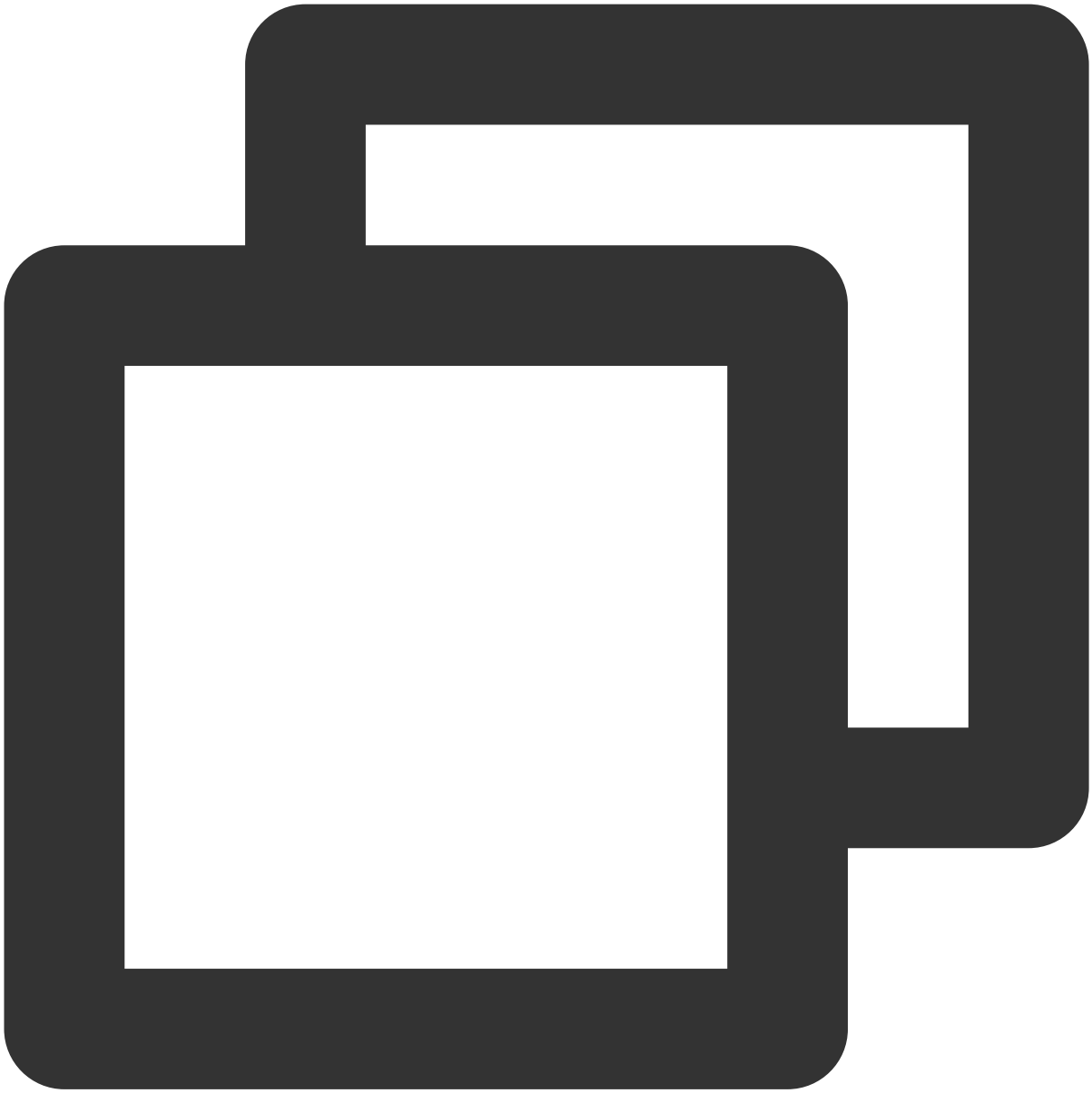
```
setMixingConfig(config: TXMixingConfig): void;
```

参数：

字段	类型	说明
config	TXMixingConfig	混流参数配置。

getMixingConfig

获取最终采用的混流参数，优先使用 setMixingConfig() 设置的结果，其次使用 `TXLivePusher` 中方法 setVideoQuality() 和 setProperty() 设置之后的结果。



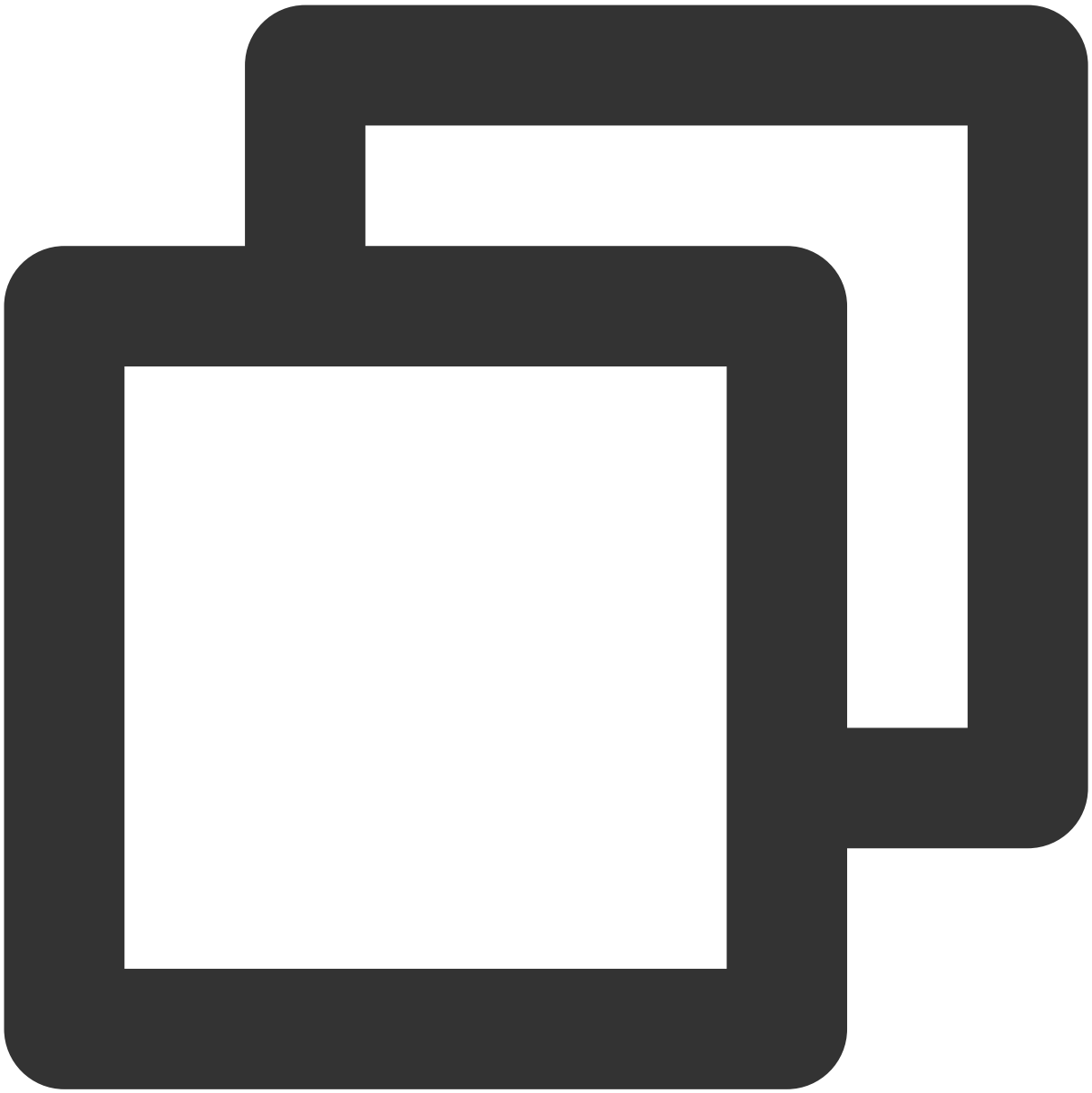
```
getMixingConfig(): TXMixingConfig;
```

返回：

混流参数配置，数据结构请参考 TXMixingConfig 。

setLayout

设置视频流的画中画布局参数。



```
setLayout(config: TXLayoutConfig | TXLayoutConfig[]): void;
```

参数：

字段	类型	说明
config	TXLayoutConfig | TXLayoutConfig[]	画中画布局配置，支持传对象或者对象数组。

说明：

1. 开启本地混流之后，所有采集的流都会自动添加到最终输出的视频流当中，默认的布局参数是保证视频流画面紧贴左上角原点出现。
2. 配置参数可以传对象数组，批量设置多个流的画中画布局效果，也可以只传对象单独设置指定的流。
3. 如果不想显示采集流的画面，只想保留声音，可以将布局宽度和高度设置为 0 。

getLayout

获取指定流的画中画布局参数。

```
getLayout(streamId: string): TXLayoutConfig | null;
```

参数：

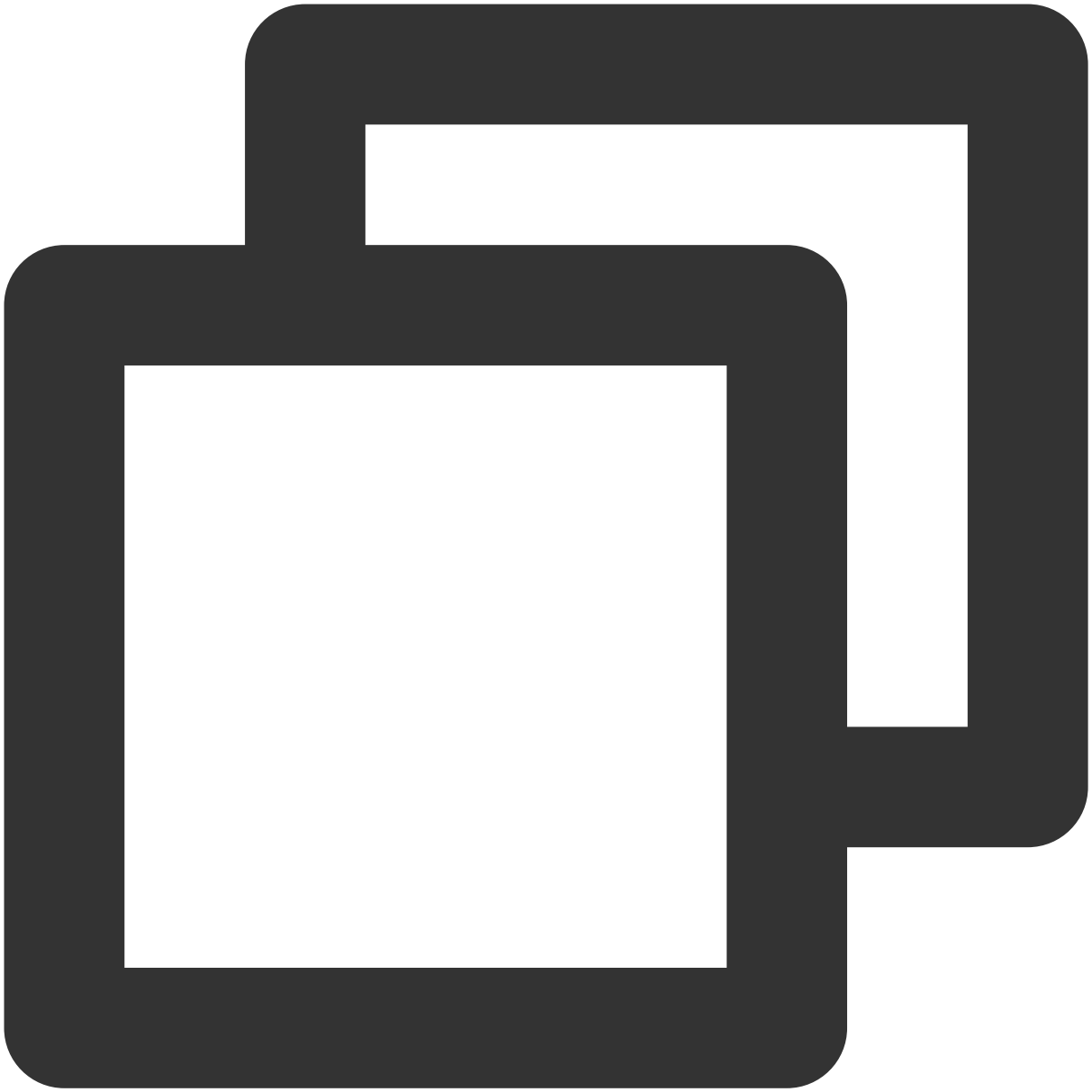
字段	类型	说明
streamId	string	流 ID，指定要获取画中画布局参数的流。

返回：

返回指定流的画中画布局参数，数据结构请参考 TXLayoutConfig 。如果流不存在，返回 null 。

setMirror

设置视频流的镜像效果，包括左右镜像和上下镜像。



```
setMirror(config: TXMirrorConfig | TXMirrorConfig[]): void;
```

参数：

字段	类型	说明
config	TXMirrorConfig | TXMirrorConfig[]	镜像效果配置，支持传对象或者对象数组。

说明：

配置参数可以传对象数组，批量设置多个流的镜像效果，也可以只传对象单独设置指定的流。

setNormalFilter

设置视频流的普通滤镜效果，包括对比度、亮度和饱和度。

```
setNormalFilter(config: TXNormalFilterConfig | TXNormalFilterConfig[]): void;
```

参数：

字段	类型	说明
config	TXNormalFilterConfig |	普通滤镜效果配置，支持传对象或者对象数组。

	TXNormalFilterConfig[]	

说明：

配置参数可以传对象数组，批量设置多个流的普通滤镜效果，也可以只传对象单独设置指定的流。

setWatermark

设置水印，支持同时设置多个水印。

```
setWatermark(config: TXWatermarkConfig | TXWatermarkConfig[] | null): void;
```

参数：

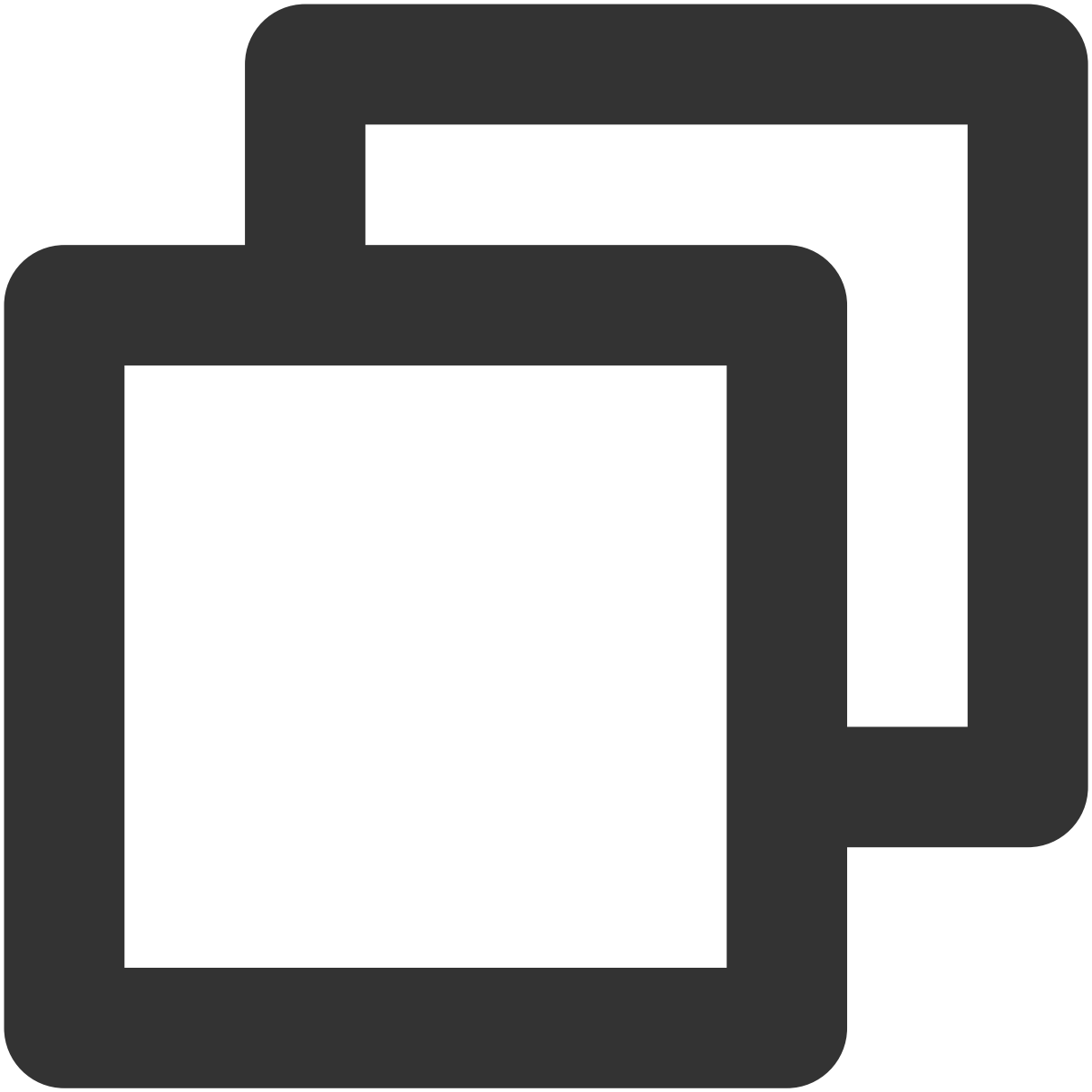
字段	类型	说明
config	TXWatermarkConfig | TXWatermarkConfig[] | null	水印配置参数，支持传对象、对象数组或者 null 。

说明：

1. 配置参数可以传对象数组，同时设置多个水印，也可以只传对象单独设置一个水印。
2. 配置参数传 null 或者空数组表示删除已有水印。

setText

设置文本，支持同时设置多个文本。



```
setText(config: TXTextConfig | TXTextConfig[] | null): void;
```

参数：

字段	类型	说明
config	TXTextConfig | TXTextConfig[] | null	文本配置参数，支持传对象、对象数组或者 null 。

说明：

1. 配置参数可以传对象数组，同时设置多个文本，也可以只传对象单独设置一个文本。

2. 配置参数传 null 或者空数组表示删除已有文本。

类型定义

TXSupportResult

浏览器支持性检查结果。
数据结构：

字段	类型	说明
isWebRTCSupported	boolean	是否支持 WebRTC
isH264EncodeSupported	boolean	是否支持 H264 编码
isH264DecodeSupported	boolean	是否支持 H264 解码
isMediaDevicesSupported	boolean	是否支持获取媒体设备及媒体流
isScreenCaptureSupported	boolean	是否支持屏幕采集
isMediaFileSupported	boolean	是否支持获取本地媒体文件流

TXLivePusherObserver

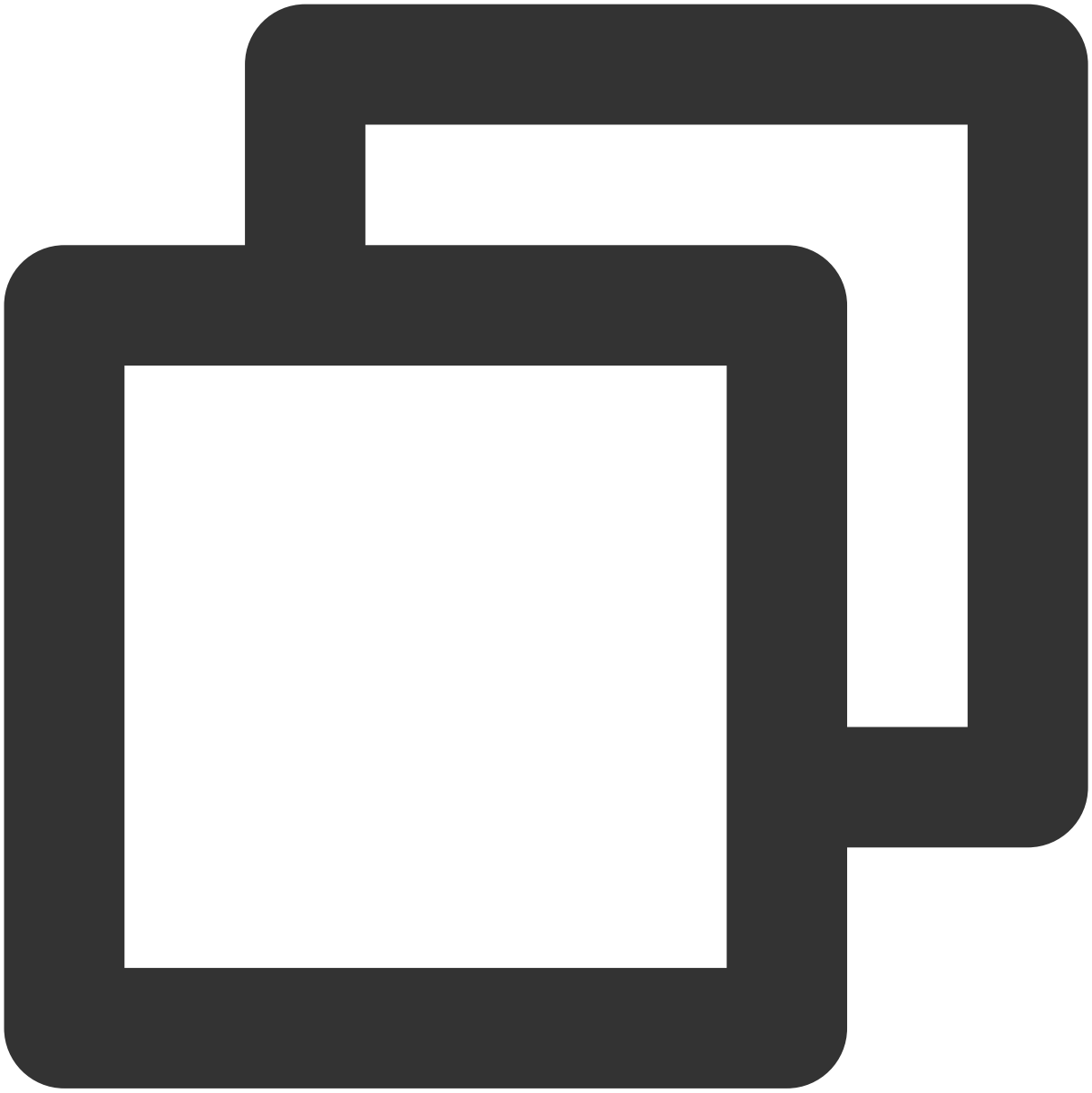
推流的回调通知，回调包括推流器状态，统计信息，警告以及错误信息。
数据结构：

字段	类型	说明
onError	onError	推流错误通知。
onWarning	onWarning	推流警告通知。
onCaptureFirstAudioFrame	onCaptureFirstAudioFrame	首帧音频采集完成的回调通知。
onCaptureFirstVideoFrame	onCaptureFirstVideoFrame	首帧视频采集完成的回调通知。
onPushStatusUpdate	onPushStatusUpdate	推流连接状态回调通知。
onStatisticsUpdate	onStatisticsUpdate	推流统计数据回调通知。

版权所有：腾讯云计算（北京）有限责任公司

onError

推流错误通知，推流出现错误时，会回调该通知。



```
onError(code: number, message: string, extraInfo: object): void;
```

参数：

字段	类型	说明
code	number	错误码。
message	string	错误信息。

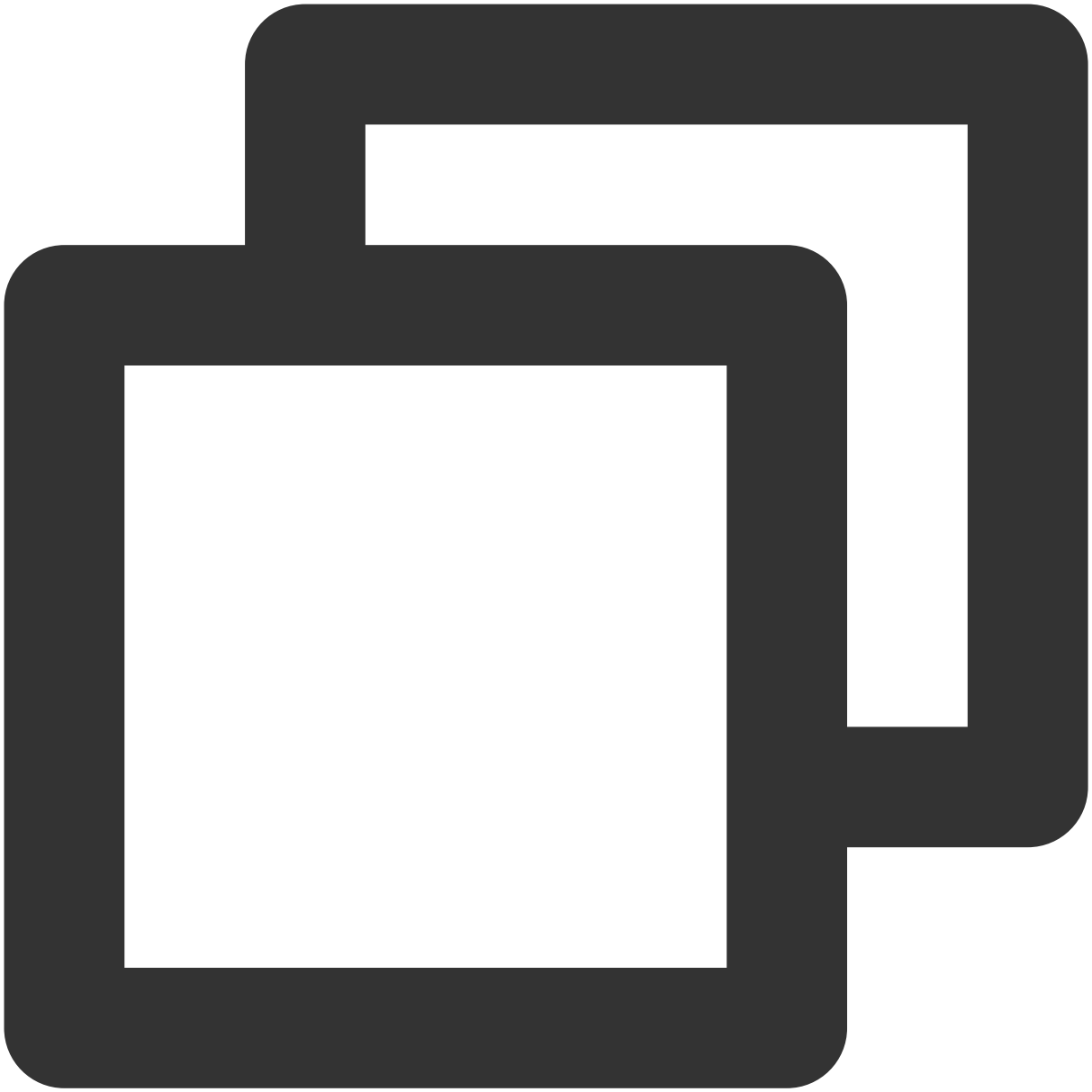
extraInfo	object	扩展信息。

错误码参考如下：

枚举值	数值	描述
TXLIVE_ERROR_WEBRTC_FAILED	-1	WebRTC 接口调用失败。
TXLIVE_ERROR_REQUEST_FAILED	-2	请求服务器推流接口返回报错。

onWarning

推流警告通知。



```
onWarning(code: number, message: string, extraInfo: object): void;
```

参数：

字段	类型	说明
code	number	错误码。
message	string	错误信息。
extraInfo	object	扩展信息。

错误码参考如下：

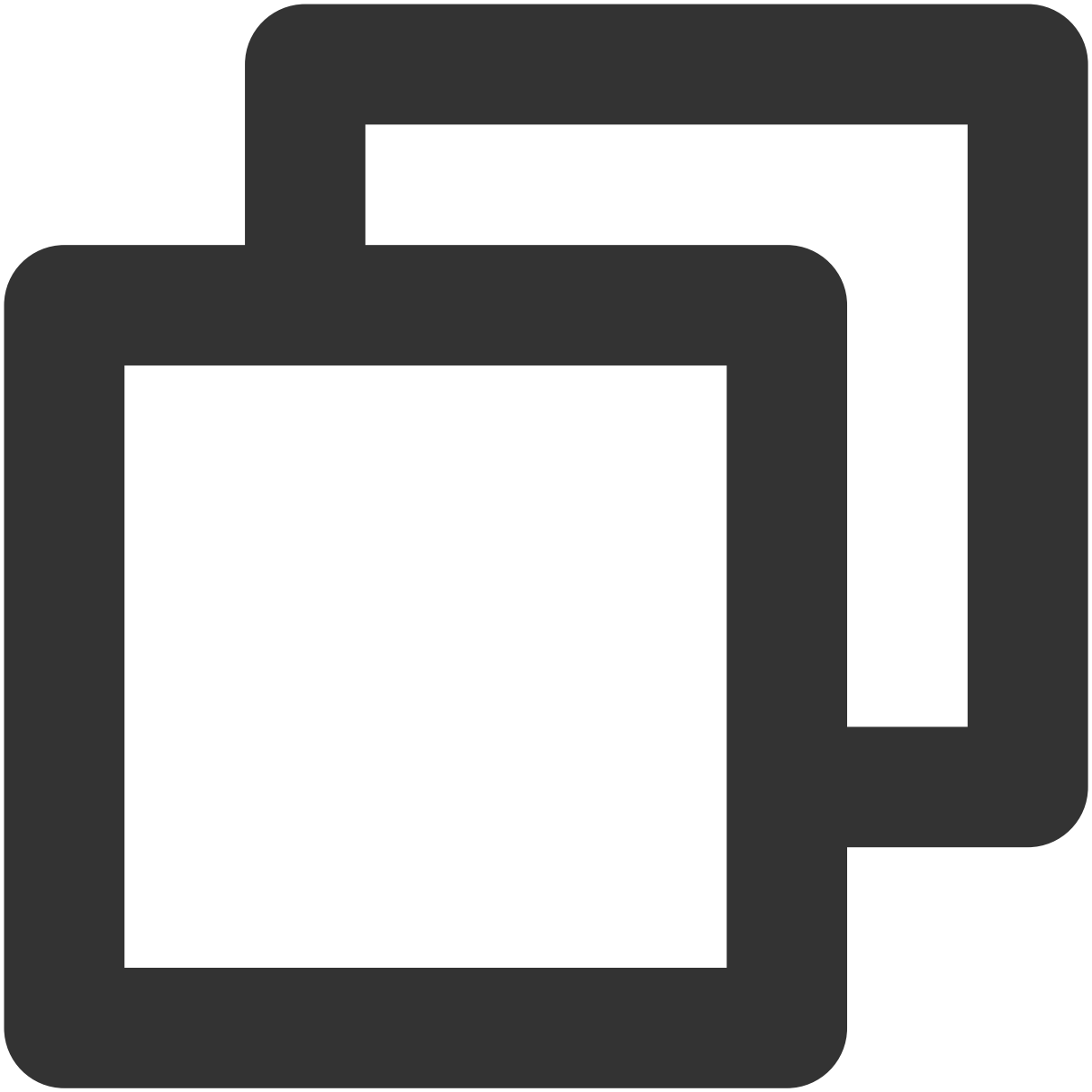
枚举值	数值	描述
TXLIVE_WARNING_CAMERA_START_FAILED	-1001	打开摄像头失败。
TXLIVE_WARNING_MICROPHONE_START_FAILED	-1002	打开麦克风失败。
TXLIVE_WARNING_SCREEN_CAPTURE_START_FAILED	-1003	打开屏幕分享失败。
TXLIVE_WARNING_VIRTUAL_CAMERA_START_FAILED	-1004	打开本地媒体文件失败。
TXLIVE_WARNING_CAMERA_INTERRUPTED	-1005	摄像头被中断（设备被拔出或者权限被用户取消）。
TXLIVE_WARNING_MICROPHONE_INTERRUPTED	-1006	麦克风被中断（设备被拔出或者权限被用户取消）。
TXLIVE_WARNING_SCREEN_CAPTURE_INTERRUPTED	-1007	屏幕分享被中断（Chrome 浏览器点击自带的停止共享按钮）。

说明：

1. 打开摄像头、麦克风、屏幕分享失败时返回的错误信息，可参考 getUserMedia 异常 和 getDisplayMedia 异常 。
2. 摄像头、麦克风、屏幕分享异常中断时返回的扩展信息，会包含对应流的流 ID。

onCaptureFirstAudioFrame

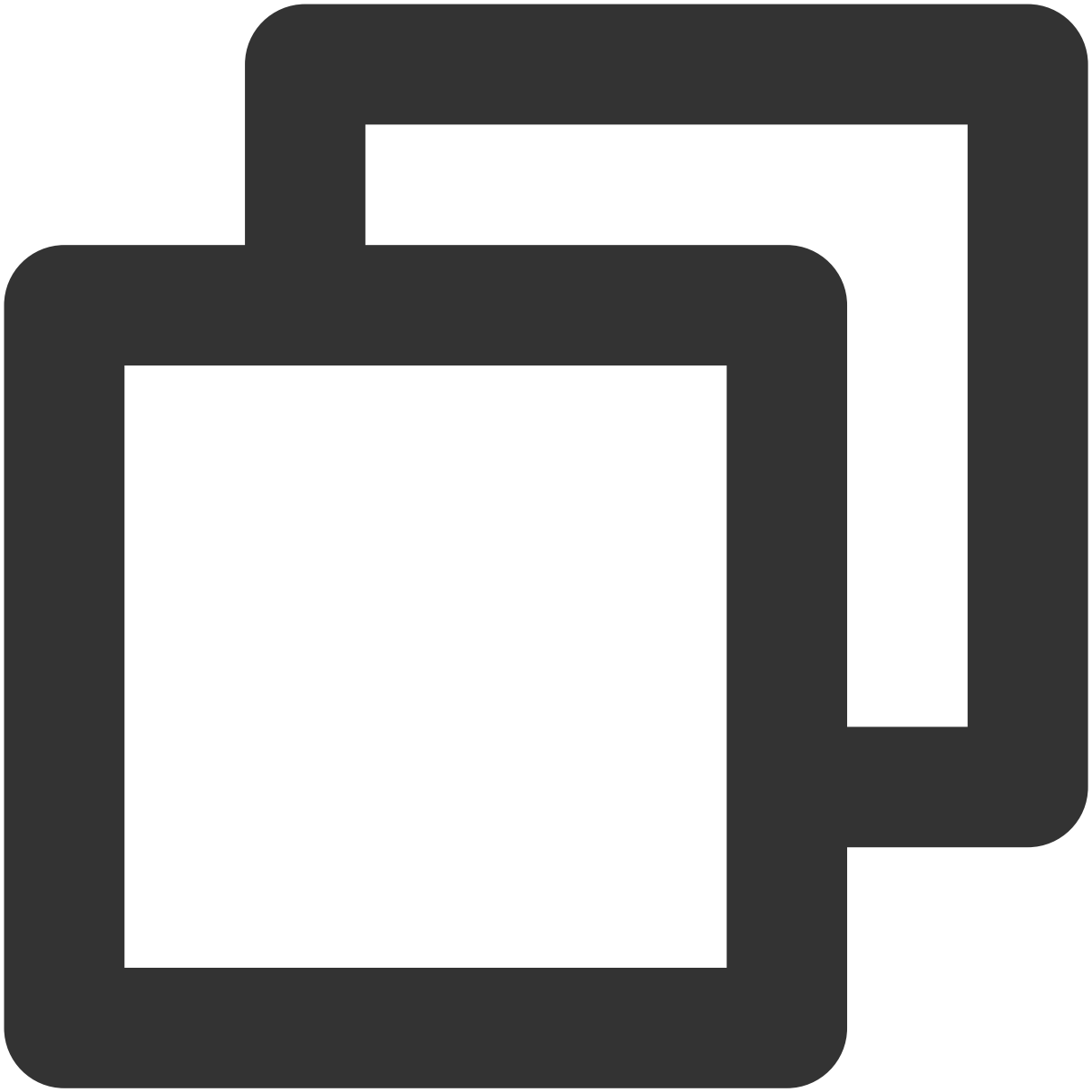
首帧音频采集完成的回调通知。如果启用本地混流功能，则在最终混流生成音频流时进行回调通知。



```
onCaptureFirstAudioFrame(): void;
```

onCaptureFirstVideoFrame

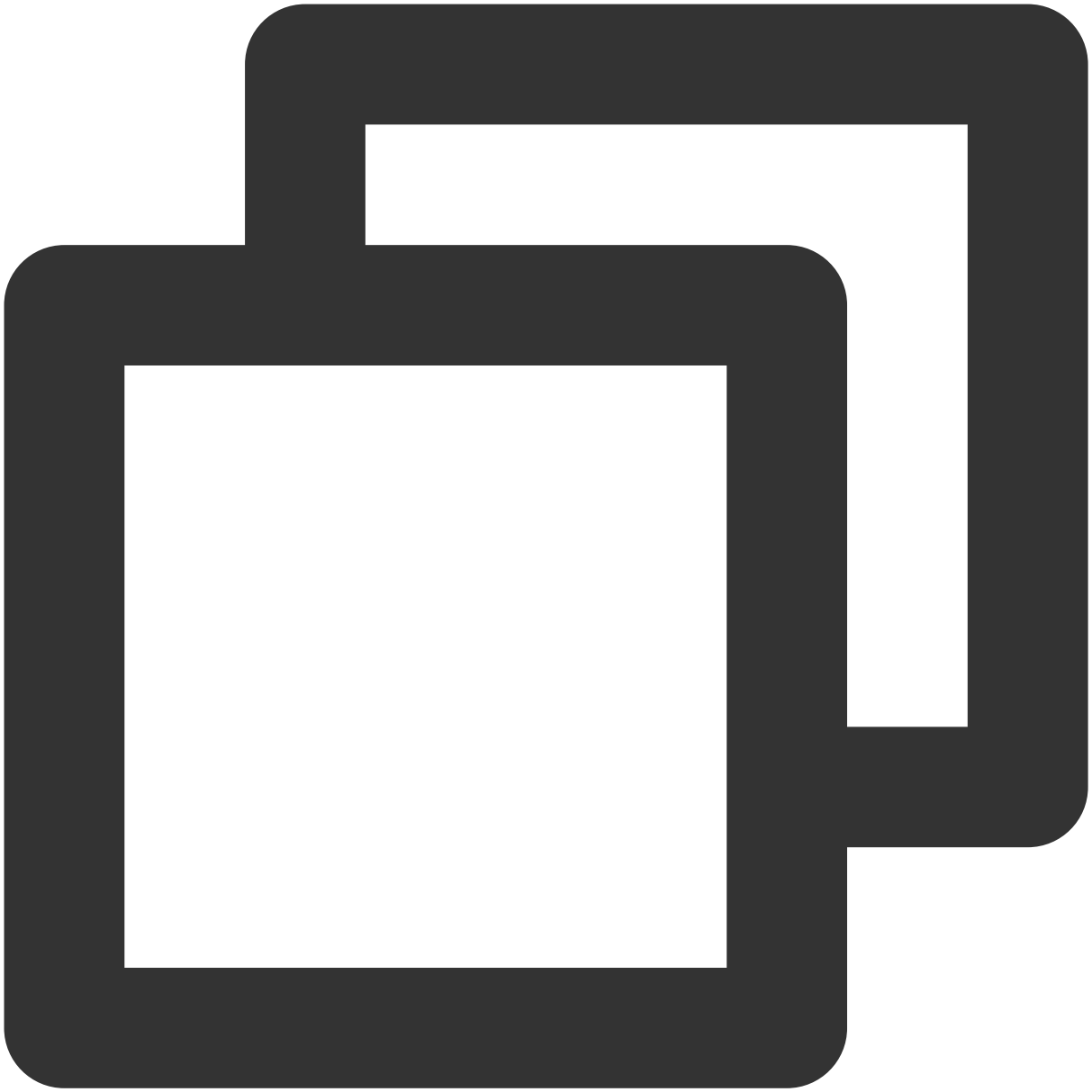
首帧视频采集完成的回调通知。如果启用本地混流功能，则在最终混流生成视频流时进行回调通知。



```
onCaptureFirstVideoFrame(): void;
```

onPushStatusUpdate

推流连接状态回调通知。



```
onPushStatusUpdate(status: number, message: string, extraInfo: object): void;
```

参数：

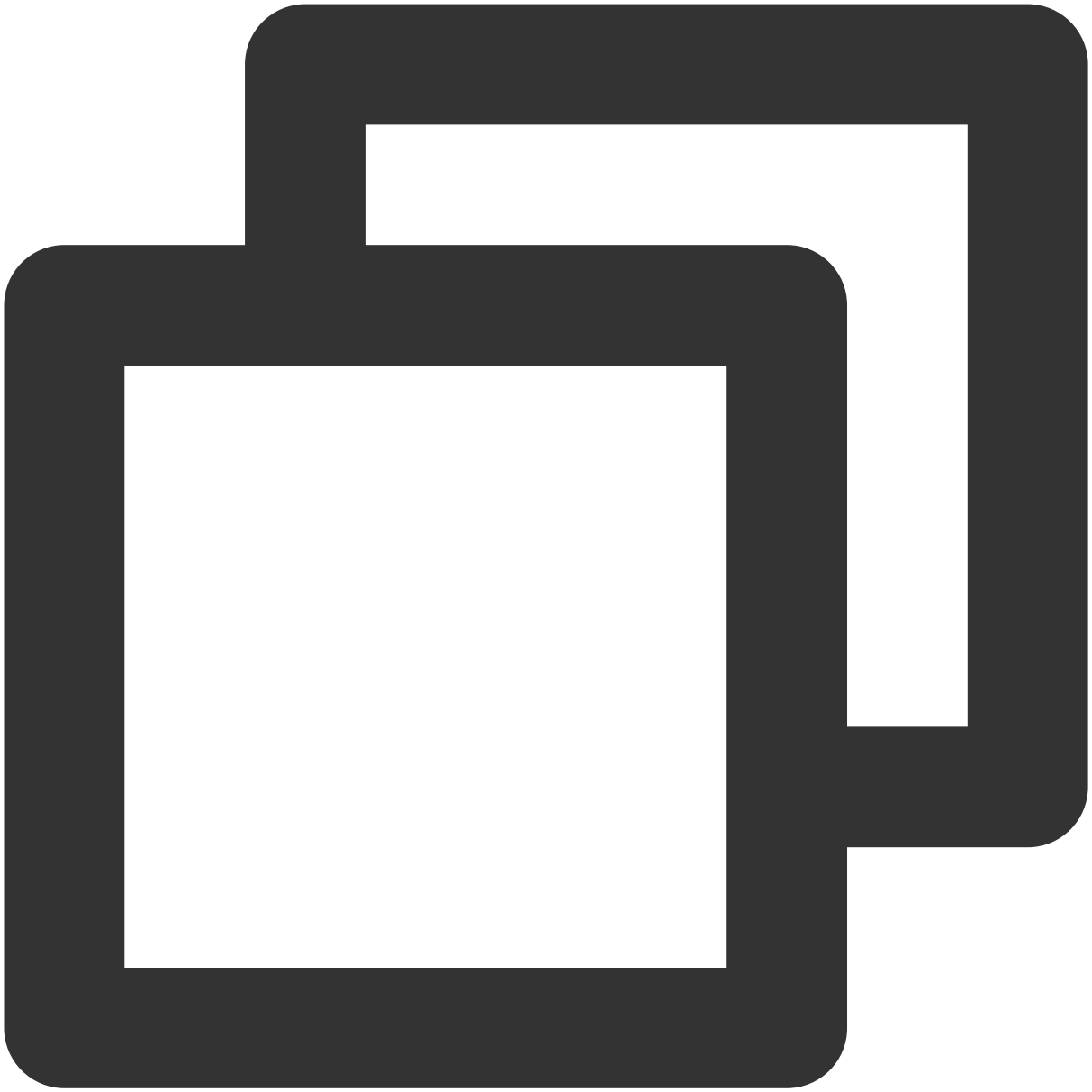
字段	类型	说明
status	number	连接状态码。
message	string	连接状态信息。
extraInfo	object	扩展信息。

连接状态码参考如下：

枚举值	数值	描述
TXLIVE_PUSH_STATUS_DISCONNECTED	0	与服务器断开连接
TXLIVE_PUSH_STATUS_CONNECTING	1	正在连接服务器
TXLIVE_PUSH_STATUS_CONNECTED	2	连接服务器成功
TXLIVE_PUSH_STATUS_RECONNECTING	3	重连服务器中

onStatisticsUpdate

推流统计数据回调，主要是 WebRTC 相关的统计数据。



```
onStatisticsUpdate(statistics: object): void;
```

参数：

字段	类型	说明
statistics	object	推流统计数据。
statistics.timestamp	number	数据采样的时间，自 1970年1月1日（UTC）起经过的毫秒数。

statistics.video	object	视频流相关的数据。
statistics.video.bitrate	number	视频码率，单位：bit/s 。
statistics.video.framesPerSecond	number	视频帧率。
statistics.video.frameWidth	number	视频宽度。
statistics.video.frameHeight	number	视频高度。
statistics.video.framesEncoded	number	编码帧数。
statistics.video.framesSent	number	发送帧数。
statistics.video.packetsSent	number	发送包数。
statistics.video.nackCount	number	NACK（Negative ACKnowledgement）数。
statistics.video.firCount	number	FIR（Full Intra Request），关键帧重传请求数。
statistics.video.pliCount	number	PLI（Picture Loss Indication），视频帧丢失重传数。
statistics.video.frameEncodeAvgTime	number	平均编码时间，单位：ms 。
statistics.video.packetSendDelay	number	数据包发送之前本地缓存的平均时间，单位：ms 。
statistics.audio	object	音频流相关的数据。
statistics.audio.bitrate	number	音频码率，单位：bit/s 。
statistics.audio.packetsSent	number	发送包数。

说明：

1. 直播推流过程中，SDK 会以一秒一次的频率统计 WebRTC 相关的数据，然后调用该回调接口返回数据。
2. 如果返回的字段数据是空（undefined）的话，说明当前浏览器获取不到对应的数据，目前只有 Chrome 浏览器可以拿到全部的数据。

TXMediaDeviceInfo

设备信息。

数据结构：

字段	类型	说明

type	string	设备类型，video - 摄像头，audio - 麦克风。
deviceId	string	设备 ID。
deviceName	string	设备名称。

TXMixingConfig

混流参数配置。

数据结构：

字段	类型	说明
videoWidth	number	最终混流后的视频宽度。
videoHeight	number	最终混流后的视频高度。
videoFramerate	number	最终混流后的视频帧率。
backgroundColor	number	混合后画面的底色颜色，格式为十六进制数字，默认黑色，即 0x000000 。

TXLayoutConfig

画中画布局参数。

数据结构：

字段	类型	说明
streamId	string	流 ID，指定要设置的流。
x	number	布局 x 坐标。
y	number	布局 y 坐标。
width	number	布局宽度。
height	number	布局高度。
zOrder	number	布局层级。

说明：

1. 以左上角为原点 (0, 0)，元素的坐标属性描述元素中心点。例如一个分辨率为 100*100 的视频流紧贴原点完整出现在最终生成视频流中，那么它的 x 、 y 坐标都是50。
2. zOrder 值越大，视频流会出现在越上方，覆盖其他视频流画面。

TXMirrorConfig

镜像参数。
数据结构：

字段	类型	说明
streamId	string	流 ID，指定要设置的流。
mirrorType	number	镜像类型：0 - 无镜像效果，1 - 左右镜像，2 - 上下镜像，3 - 左右+上下镜像。

TXNormalFilterConfig

普通滤镜参数。
数据结构：

字段	类型	说明
streamId	string	流 ID，指定要设置的流。
contrast	number	对比度，取值范围 [-100, 100]，0 表示不处理。
brightness	number	亮度，取值范围 [-100, 100]，0 表示不处理。
saturation	number	饱和度，取值范围 [-100, 100]，0 表示不处理。

TXWatermarkConfig

水印配置参数。
数据结构：

字段	类型	说明
image	HTMLImageElement	水印图片对象。
x	number	水印 x 坐标。

y	number	水印 y 坐标。
width	number	水印宽度。
height	number	水印高度。
zOrder	number	水印层级。

说明：

1. 以左上角为原点 (0, 0)，元素的坐标属性描述元素中心点。例如一个分辨率为 100*100 的水印图片紧贴原点完整出现在最终生成视频流中，那么它的 x 、 y 坐标都是50。
2. zOrder 值越大，水印图片会出现在越上方，覆盖其他视频流画面。

TXTextConfig

文本配置参数。

数据结构：

字段	类型	说明
text	string	文本内容，不能为空字符串。
style	object	文本样式，可以参考对应的 css 样式设置。
style.font	string	字体名。
style.font_size	number	字体大小。
style.font_color	string	字体颜色，十六进制表示，例如 #000000 。
style.font_alpha	number	字体透明度，范围 [0, 100]，默认 100（不透明）。
style.bold	number	字体加粗，0 - 不加粗，1 - 加粗，默认 0 。
style.italic	number	字体倾斜，0 - 正常，1 - 斜体，默认 0 。
style.shadow_color	string	文字阴影颜色，十六进制表示，例如 #000000 。
style.shadow_alpha	number	文字阴影透明度，范围 [0, 100]，shadow_color 存在时有效，默认 100（不透明）。
style.stroke_color	string	文字描边颜色，十六进制表示，例如 #000000 。
style.stroke_thickness	number	文字描边粗细，默认 0，即没有描边。
style.background_color	string	背景颜色，十六进制表示，例如 #000000 。

版权所有：腾讯云计算（北京）有限责任公司

style.background_alpha	number	背景透明度，范围 [0, 100]，background_color 存在时有效，默认 100（不透明）。
x	number	文本 x 坐标。
y	number	文本 y 坐标。
zOrder	number	文本层级。

说明：

1. 以左上角为原点 (0, 0)，元素的坐标属性描述元素中心点。假设文本的最终宽度是 100px，高度是 50px，如果文本要紧贴原点出现在最终生成视频流中的话，它的 x 坐标是 50， y 坐标是 25。
2. zOrder 值越大，文本会出现在越上方，覆盖其他视频流画面。

版权所有：腾讯云计算（北京）有限责任公司

2. 播放

最近更新时间：2022-11-21 16:56:28

准备工作

- 开通 腾讯云直播服务。
- 选择 域名管理，单击 **添加域名**，选择类型为播放域名，详细请参见 添加自有域名。
- 进入云直播控制台的 **直播工具箱** > 地址生成器生成播放地址，详情请参见 地址生成器。接下来根据业务场景使用以下方式在自己的业务中实现直播播放。

App 接入

下载并集成 腾讯云·直播SDK(MLVB)具体可参考对接文档（iOS & Android）完成接入。

注意：
开启 拉流播放，iOS 和 Android 的处理方式分别如下：

- iOS

```
V2TXLivePlayer *_txLivePlayer = [[V2TXLivePlayer alloc] init];
>
```

- Android

```
V2TXLivePlayer mLivePlayer = new V2TXLivePlayerImpl(mContext);
```

更多

- 在使用腾讯云·直播 SDK 的过程中需要付费，若您需要了解腾讯云·直播 SDK 相关计费说明，详情请参见 价格总览。

3. 高级功能

概述

最近更新时间：2024-07-24 10:32:13

通过阅读本文，您可以了解到如何在自己的程序中引入直播高级功能。

播放AV1格式视频

AV1是一款开源、免版权费的视频压缩格式，相比上一代H.265[HEVC]编码，在相同画质下码率可以再降低30%+，这就意味着在同等带宽下可以传输更高清的画质。目前云直播已经具备AV1编码能力，但要播放AV1格式的视频，需要播放器可以解码AV1 格式的视频。
如果要在自有的播放器中支持AV1 解码，可参考如下步骤处理：

容器格式与分发协议

AV1 in FLV 腾讯目前做了私有化扩展（in T-FFmpeg），需要对播放器进行改造，代码部分可以基于 T-FFmpeg 提供的 Patch 文件 做扩展，具体说明请参见 FLV 扩展支持AV1编码格式。

解码

- **硬解码**
  PC 生态，AMD、Intel、Nvidia 的较新款 GPU 基本都支持了 AV1 硬解码。
  已支持AV1硬解码设备如下所示：

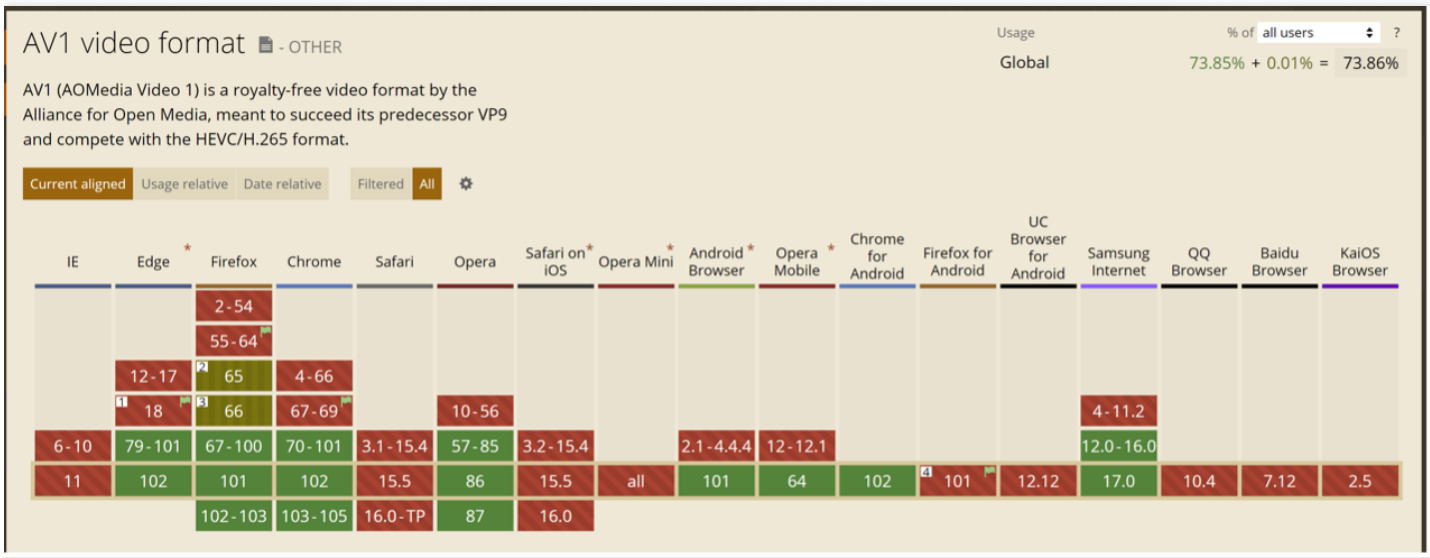
类型	品牌	处理器
手机	realme X7 Pro	天玑1000+
	oppo reno 5 pro	天玑1000+
	荣耀v40	天玑1000+
	Redmi k30 Ultra	天玑1000+
	vivo iQOO Z1	天玑1000+
	Redmi Note 10 Pro	天玑1000+
	vivo S9	天玑1100
	realme Q3 Pro	天玑1100

类型	品牌	处理器
	vivo s10	天玑1100
	vivo s12	天玑1100
	vivo s12 pro	天玑1200
	OPPO Reno6 Pro	天玑1200
	OPPO Reno7 Pro	天玑1200
	红米K40 pro	天玑1200
	realme GT Neo	天玑1200
	荣耀X20	天玑1200
	一加Nord 2	天玑1200
	realme GT Neo2	天玑1200
	OPPO K9 Pro	天玑1200
	OPPO Find X5 Pro天玑版	天玑9000
	Redmi K50	天玑9000
	Galaxy S21(三星芯片版)	Exynos 2100
	Galaxy S22(三星芯片版)	Exynos 2200
电视机	三星 新旗舰 8K 液晶电视 Q950TS	-

- **软解码**
  - av1d（腾讯优化版本的 AV1 解码器，性能优于 dav1d，可以对外提供闭源的库，集成方法请参见 av1d 集成说明，T-FFmpeg 提供 FFmpeg 部分的集成 Patch 和 av1d 库
  - dav1d
  - libgav1
  - Android 10.0 集成了AV1 解码器
  - Chrome 体系支持了AV1 解码

浏览器支持情况

Chrome 体系已经支持，iOS 体系不支持。



注意：

本数据为 AVI 官网 于2022年07月的统计，可前往该网站查询最新统计结果。

媒体传输 SDK（TMIO SDK）

TMIO SDK 通过对流媒体协议 SRT、QUIC 等的定制封装优化，为上行推流传输进行保驾护航，实现低延时传输、优秀的抗丢包能力、多链路传输优化、超时重传机制，对于推流数据源稳定性要求较高的场景，以及远距离传输有广阔的应用前景。

功能介绍

- 适用于远距离传输以及广电领域。
- 支持 Android、iOS、Windows、MacOS、Linux 等主流平台。

接入方式

接入 SDK，详情请参见 接入步骤。

快直播传输层 SDK

快直播传输层 SDK（libLebConnection）提供基于原生 WebRTC 升级版的传输能力，用户仅需对已有播放器进行简单改造，即可接入快直播。在完全兼容标准直播的推流、云端媒体处理能力的基础上，实现高并发低延迟直播，帮

助用户实现从现有的标准直播平滑地迁移到快直播上来。也可以帮助用户在现有 RTC 场景中快速实现低成本的大房间低延迟旁路直播。

功能介绍

- 音视频拉流，兼具优异的低延迟性能和抗弱网能力。
- 视频支持H.264、H.265和 AV1，支持 B 帧，视频输出格式为视频帧裸数据（H.264/H.265为 AnnexB，AV1 为 OBU）。
- 音频支持 AAC 和 OPUS，音频输出格式为音频帧裸数据。
- 支持 Android、iOS、Windows、Linux 和 Mac 平台。

接入方式

接入 SDK，详情请参见 接入步骤。

美颜特效

在直播过程中如果想接入美颜特效功能，引入美颜、滤镜、贴纸等，可以接入腾讯云·腾讯特效引擎（Tencent Effect）SDK 。

App 接入

下载并集成 腾讯特效引擎（Tencent Effect）SDK，具体可参考对接文档（iOS & Android）完成接入完成接入。

更多

使用 腾讯特效引擎（Tencent Effect）SDK 会产生费用，详情请参见 价格总览。

媒体传输 SDK（TMIO SDK）

最近更新时间：2023-11-24 15:01:48

简单介绍

TMIO SDK（Tencent Media IO SDK）是针对当前日益丰富的流媒体传输协议，对主流协议进行整合优化和扩展，为用户可以开发出稳定可用的媒体应用提供服务方便，摆脱繁重的多种协议的开发调试工作。
TMIO SDK 当前已对 SRT、QUIC 等主流媒体协议 进行了优化扩展，同时新增自研传输控制协议 ETC（全称：Elastic Transmission Control），后续将会持续增加其他主流协议的优化扩展。

能力优势

多平台支持：包括 Android、iOS、Linux、Mac 和 windows。
灵活的选择接入方式：
零代码入侵的代理模式， 请参见 接入简介。
依托简单的 API 设计，可实现快速接入替换原有传输协议，请参见 接入简介。
API 接口设计简单，兼容性灵活性强：
提供简单易用的接口设计。
可根据业务需求场景选择合适的模式和策略。
可扩展其他协议的定制优化。
整合多种传输协议扩展优化：
SRT、QUIC 等新一代主流传输协议和自研传输控制协议 ETC 支持
可适用于多种业务场景的不同需求选择
基于 UDP 的低延时、安全可靠的传输设计
多链路加速优势，保证传输更稳定、流畅

效果展示（以TMIO SRT 为例）

TMIO 支持 SRT 协议，可用于弱网、远距离传输场景中，提高上行稳定性和下行流畅度。
如下测试场景中，RTMP推流在10%丢包率时已播放卡顿，SRT推流在10%甚至30%丢包率时仍能保持稳定和低延迟。

功能介绍

基于 SRT 的流媒体传输功能
抗随机丢包能力强。
基于 ARQ 及超时策略的重传机制。
基于 UDT 的低延时、安全可靠的传输设计。

多链路传输功能，在原基础上新增扩展链路聚合功能：

多链路传输功能可配置多条链路来实现数据的传输，特别是在当前4G/5G网络普遍的情况下，移动终端设备既可以使用 Wi-Fi，又可以使用数据网络来进行数据的传输，这样即使单一网络突然中断，只要有一条链路可用就可以保证链路的稳定性。

功能模式	说明
广播模式	可配置多个链接实现冗余发送，保证数据的完整性和连接可靠性。
主备模式	基于链路稳定性和可靠性做参考，同一时间只有一路链接的活跃，实时选择更优质的链路来实现数据传输。既保证了链路的稳定性和可靠性，又能够减少冗余数据带来的带宽消耗。
聚合模式	对于高码率、带宽要求的场景，当单一链路带宽无法满足其需求时，聚合模式可将数据通过多链路来拆分发送，同时在接收侧重组，以达到增大带宽的目的。

基于 QUIC 的流媒体传输功能

自适应拥塞控制算法。

支持网络连接迁移，平滑无感知。

支持下一代HTTP3基础传输协议。

带宽受限、抖动环境下，发送冗余数据更低，更加节约带宽成本，优势明显。

自研传输控制协议ETC

纯自研，轻量级，跨平台。

支持IoT设备，适合端到端通信。

快启动、低时延且高带宽利用率。

迅速、准确地感知链路状态变化，并及时调整到最佳传输策略。

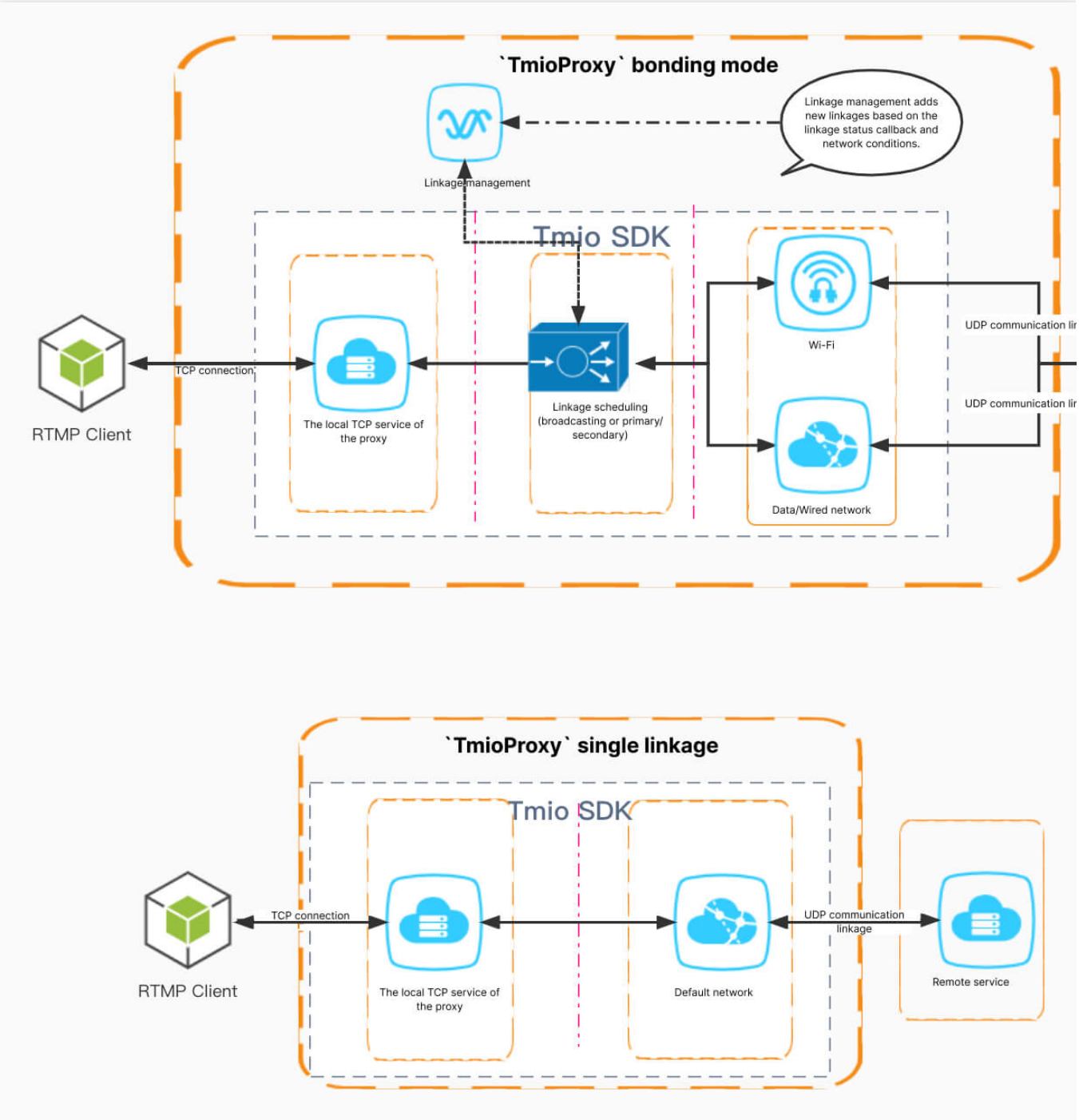
与主流传输协议并存时，能更公平、更稳定地利用带宽。

接入方法

以 RTMP over SRT 协议为例：

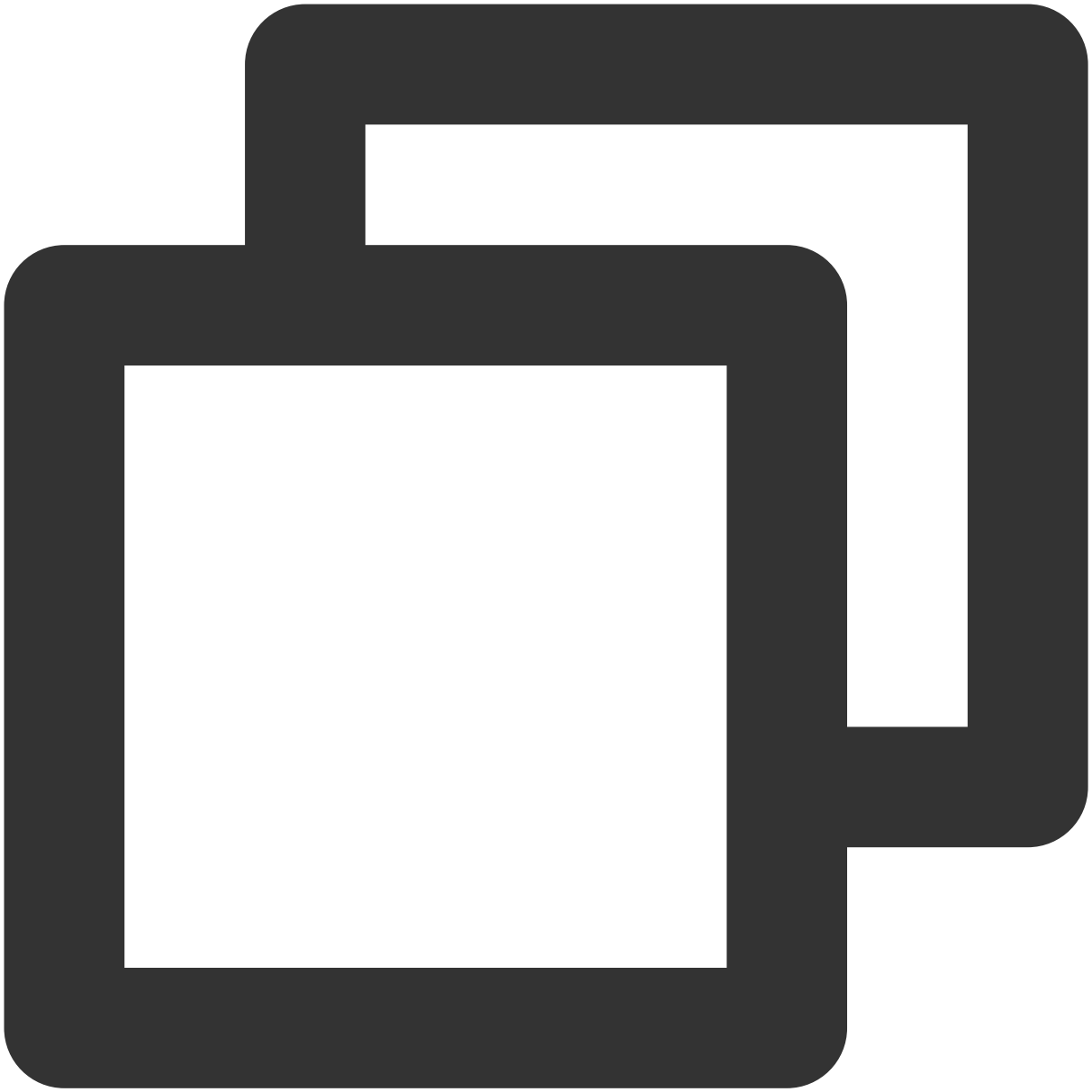
选择代理模式

Tmio Proxy 模式接入方式



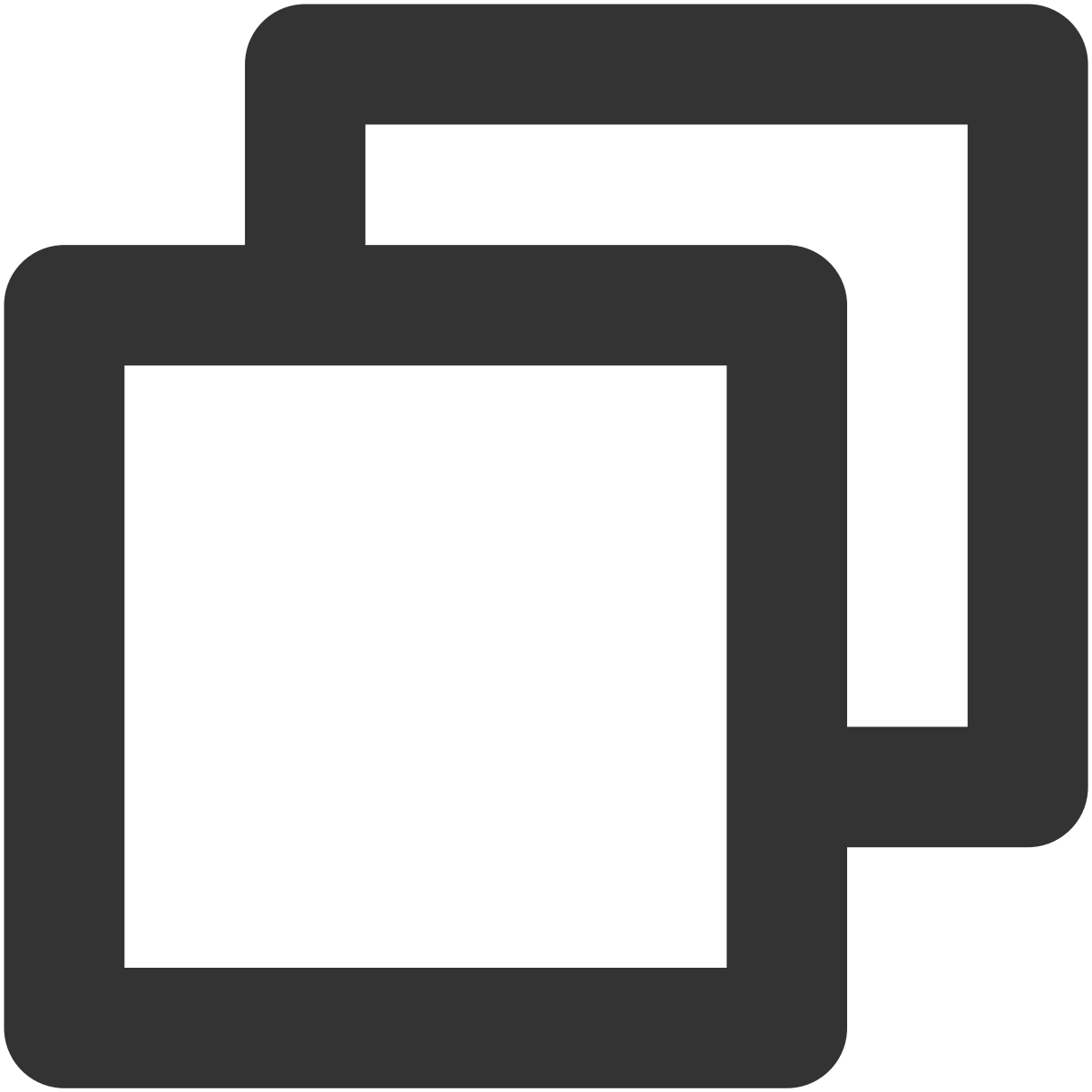
操作步骤

1. **创建 Tmio Proxy**：



```
std::unique_ptr<tmio::TmioProxy> proxy_ = tmio::TmioProxy::createUnique();
```

2. **设置监听**：



```
void setListener(TmioProxyListener *listener);
```

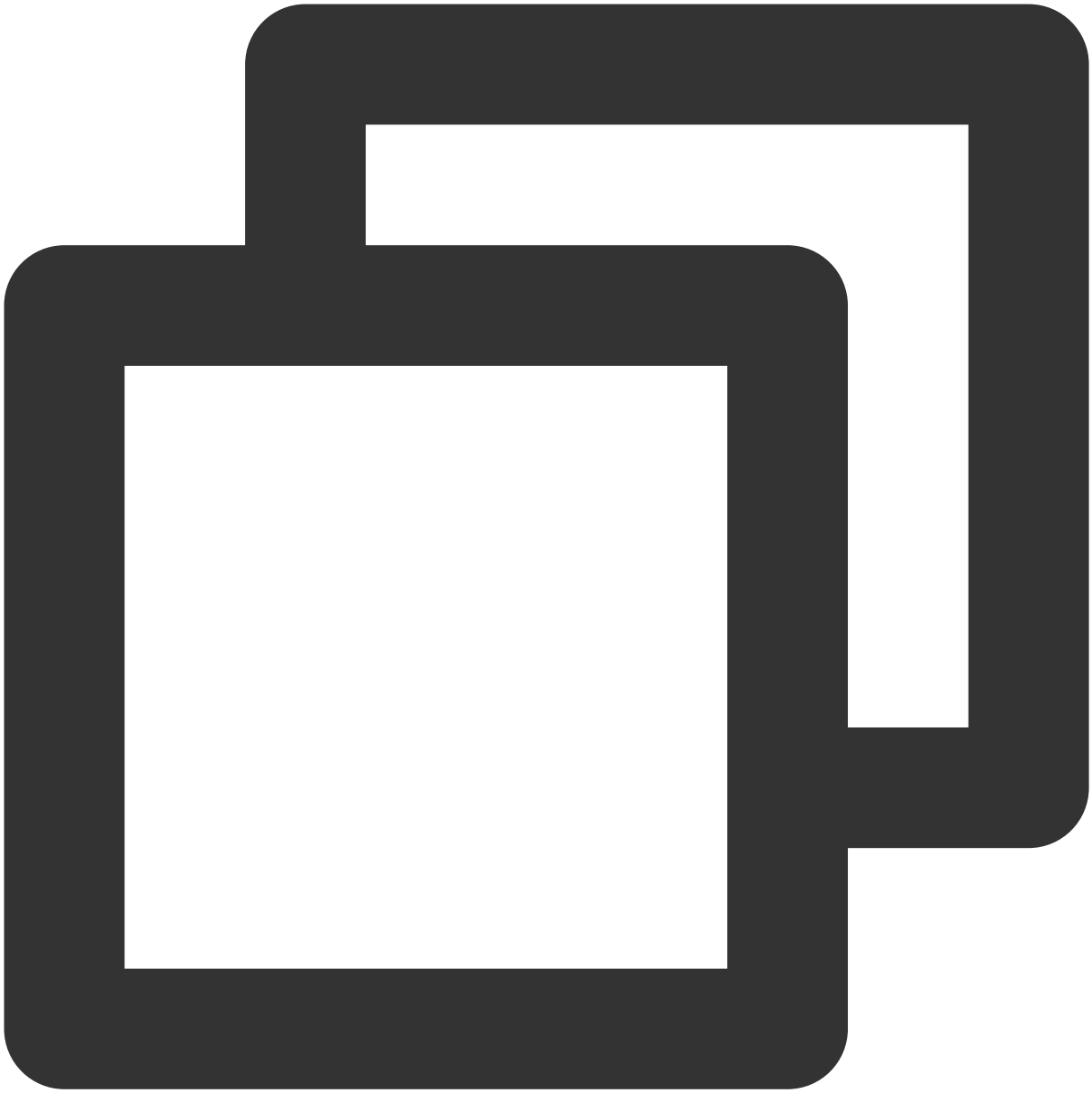
TmioProxyListener 监听接口如下：

Tmio 配置回调

TmioProxy 启动回调

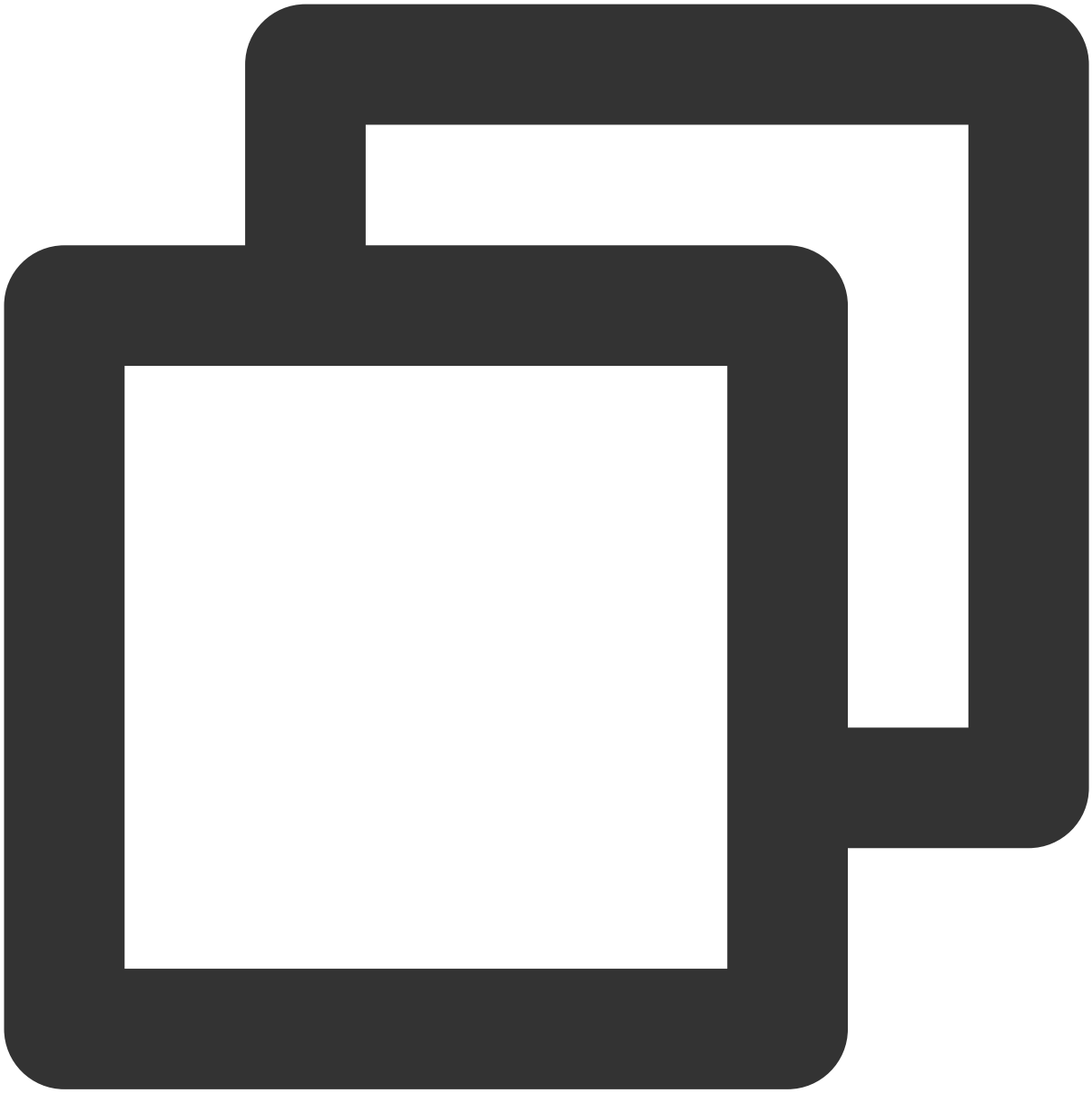
错误信息回调

用户可在此回调内对 Tmio 做参数配置，**简单配置可使用 `tmio-preset.h` 提供的辅助方法**。



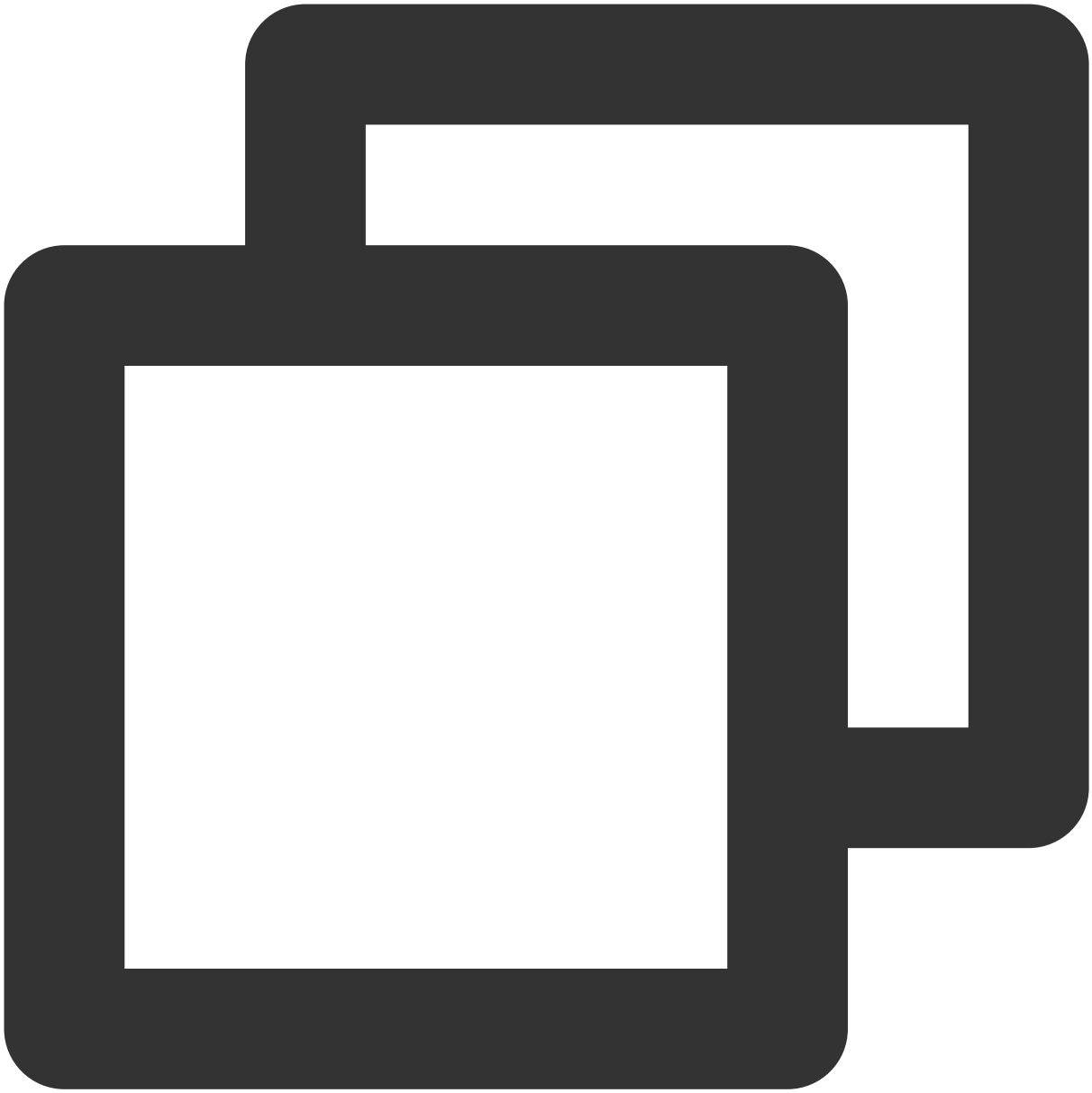
```
/*
void onTmioConfig(Tmio *tmio);
*/
void onTmioConfig(tmio::Tmio *tmio) override {
        auto protocol = tmio->getProtocol();
        if (protocol == tmio::Protocol::SRT) {
                tmio::SrtPreset::rtmp(tmio);
        } else if (protocol == tmio::Protocol::RIST) {
                tmio->setIntOption(tmio::base_options::RECV_SEND_FLAGS,
                                                        tmio::base_options::FLAG_SEND)
        }
```

```
}
```



```
/*
void onStart(const char *local_addr, uint16_t local_port);
*/
void onStart(const char *addr, uint16_t port) override {
        LOGFI("ip %s, port %" PRIu16, addr, port);
}
```

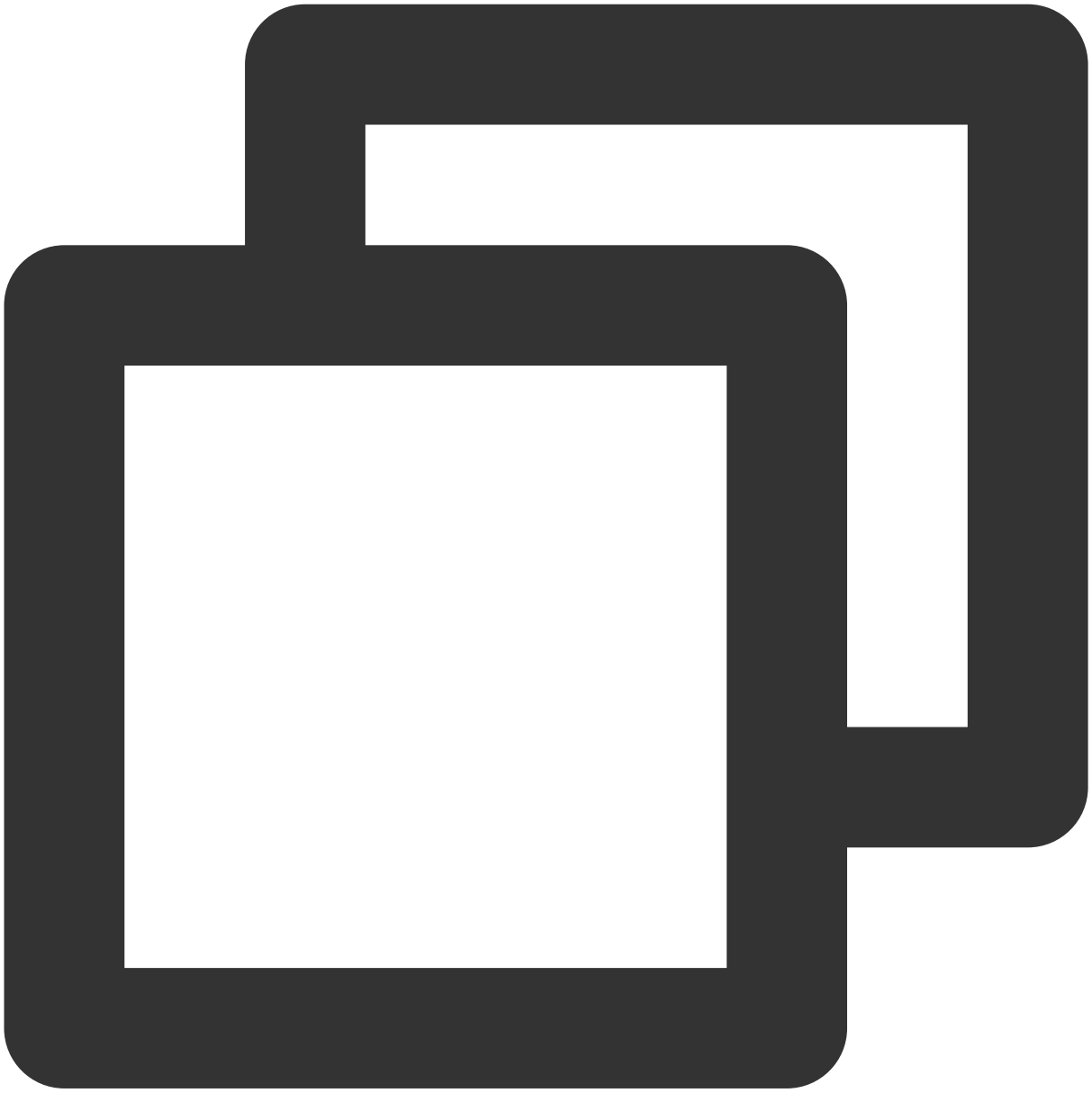
收到此回调代表连接远程服务器成功，并且 TCP 本地端口绑定成功，可以启动推流。



```
/*
void onError(ErrorType type, const std::error_code &err);
*/
void onError(tmio::TmioProxyListener::ErrorType type,
                        const std::error_code &err) override {
        LOGFE("error type %s, %s, %d", tmio::TmioProxyListener::errorType(type),
                    err.message().c_str(), err.value());
}
```

用户可通过 ErrorType 来区分是本地 IO 错误还是远程 IO 错误。本地 IO 通常是 RTMP 推流主动触发的，如结束推流，一般可忽略，而远程 IO 错误一般不可忽略。

3. **启动代理**：



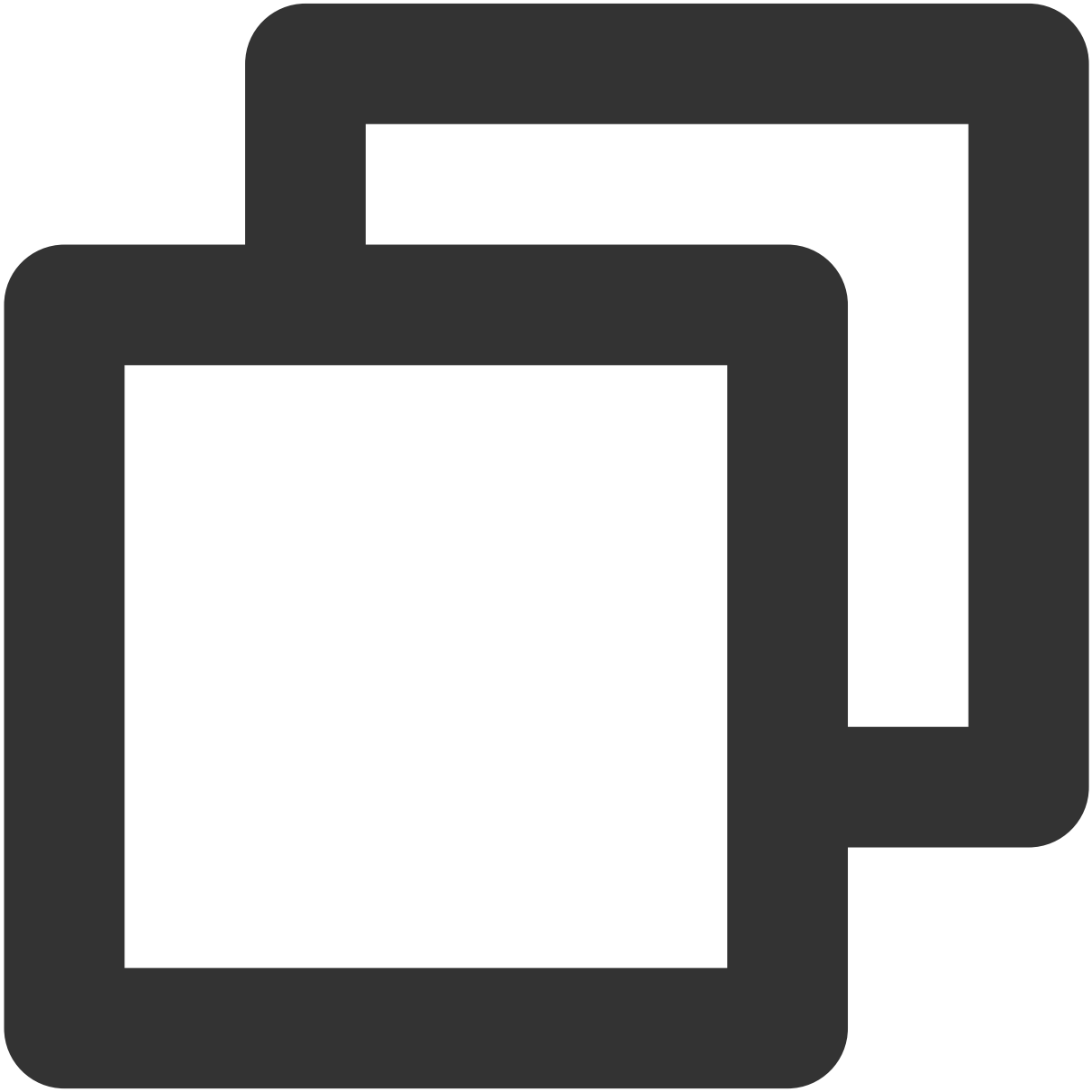
```
std::error_code start(const std::string &local_url, const std::string &remote_url,
```

接口参数

参数	说明
local_url	只支持 TCP Scheme，格式为 `tcp://${ip}:${port}` 。port 可以为0，为0时会绑定到

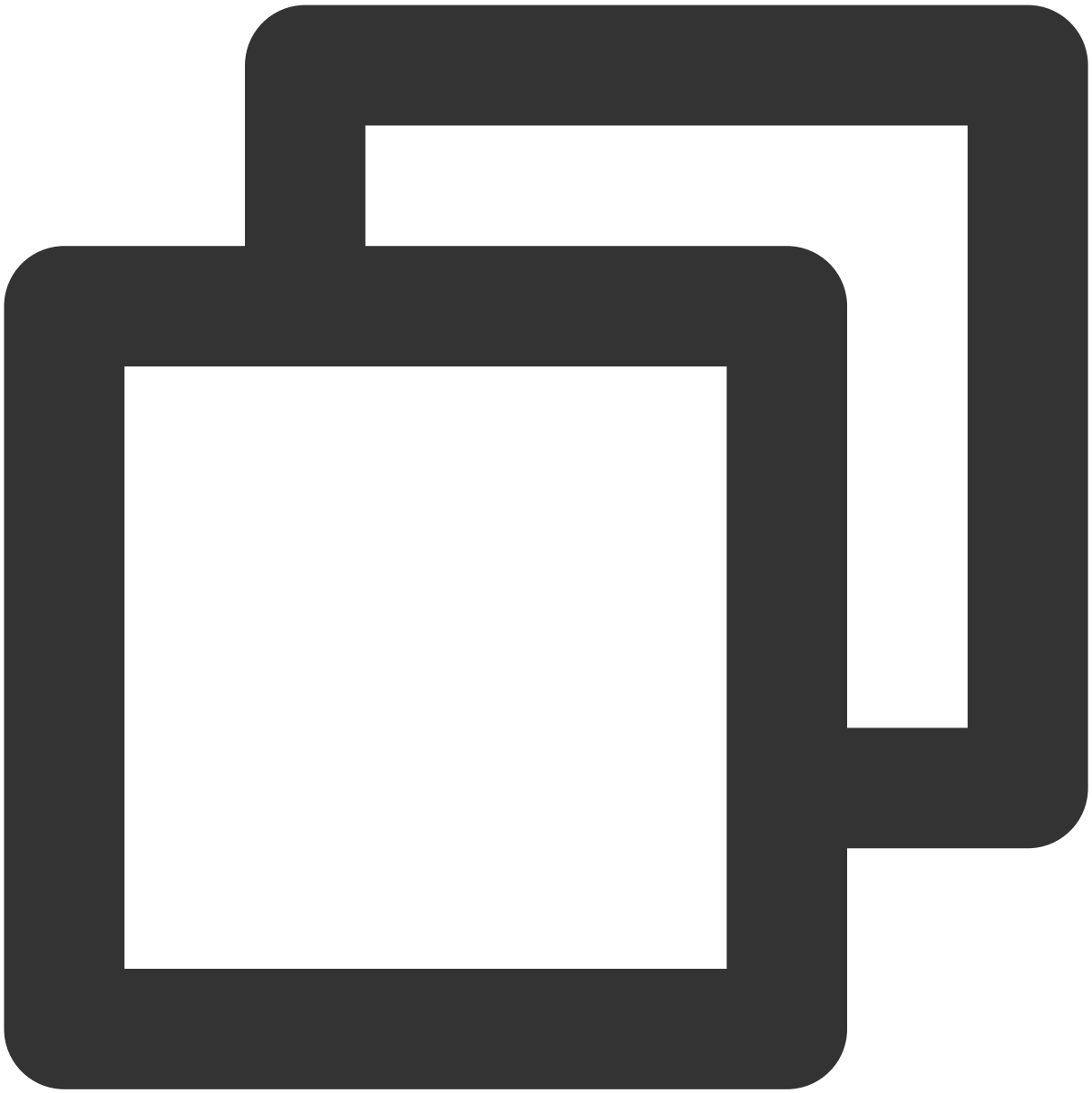
	随机端口，然后通过 onStart() 回调把绑定成功后的端口号返回给应用。使用0端口可以避免端口被占用、无权限等导致的绑定失败问题。
remote_url	远程服务器 URL。
config	配置参数，此参数在 SRT bonding 功能和 QUIC H3 协议启用时使用，具体定义请依据 tmio.h 下 TmioFeatureConfig 结构体定义。

单链路（代码示例）



```
proxy_->start(local_url, remote_url, NULL);
```

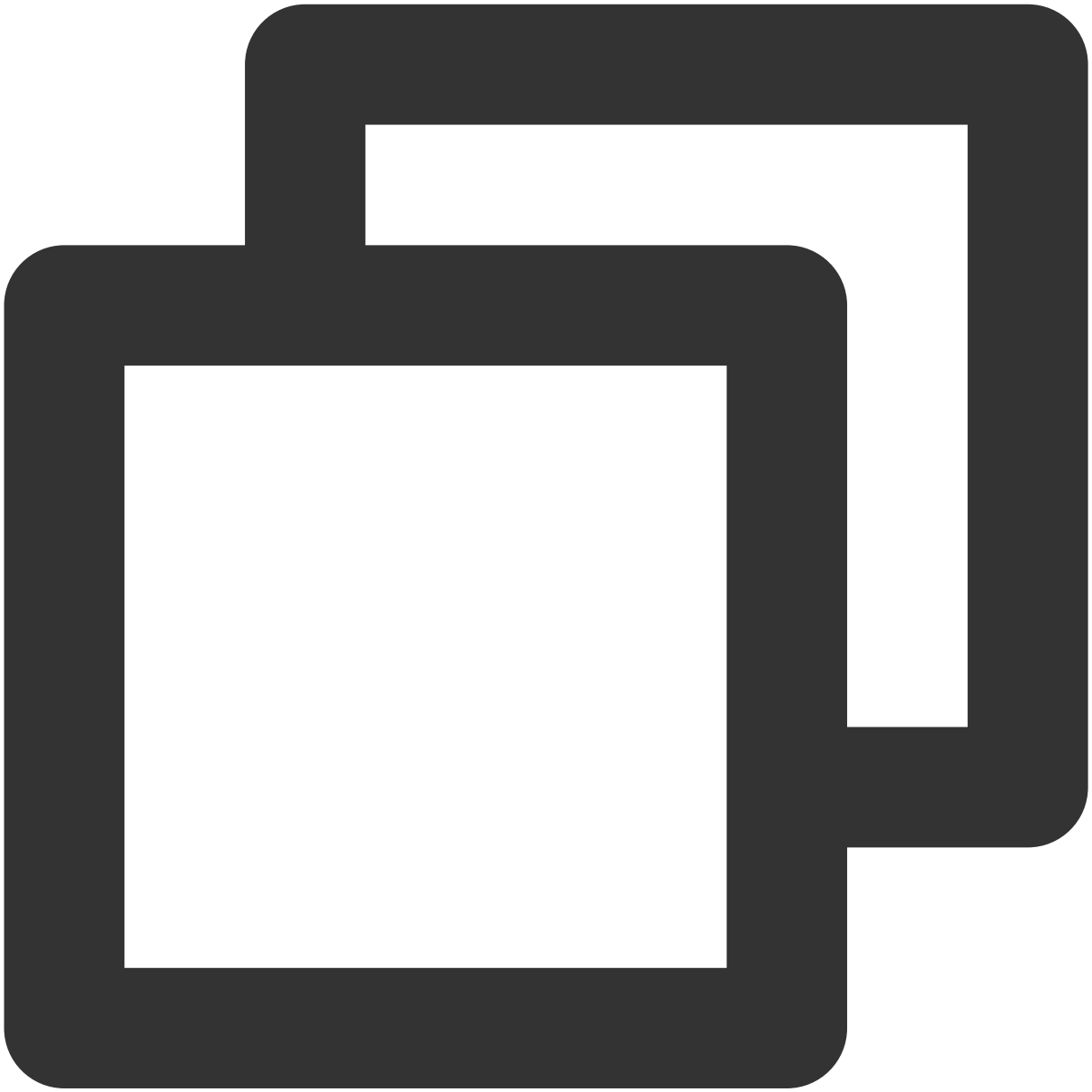
bonding 多链路（代码示例）



```
tmio::TmioFeatureConfig option;
option_.protocol = tmio::Protocol::SRT;
option_.trans_mode = static_cast<int>(tmio::SrtTransMode::SRT_TRANS_BACKUP);
/*-------------------------------------------------------------*/
{
//根据当前需要建立的链路数可添加多个链路
option_.addAvailableNet(net_name, local_addr, remote_url, 0, weight, -1);
}
/*-------------------------------------------------------------*/
```

```
proxy_->start(local_url, remote_url, &option_);
```

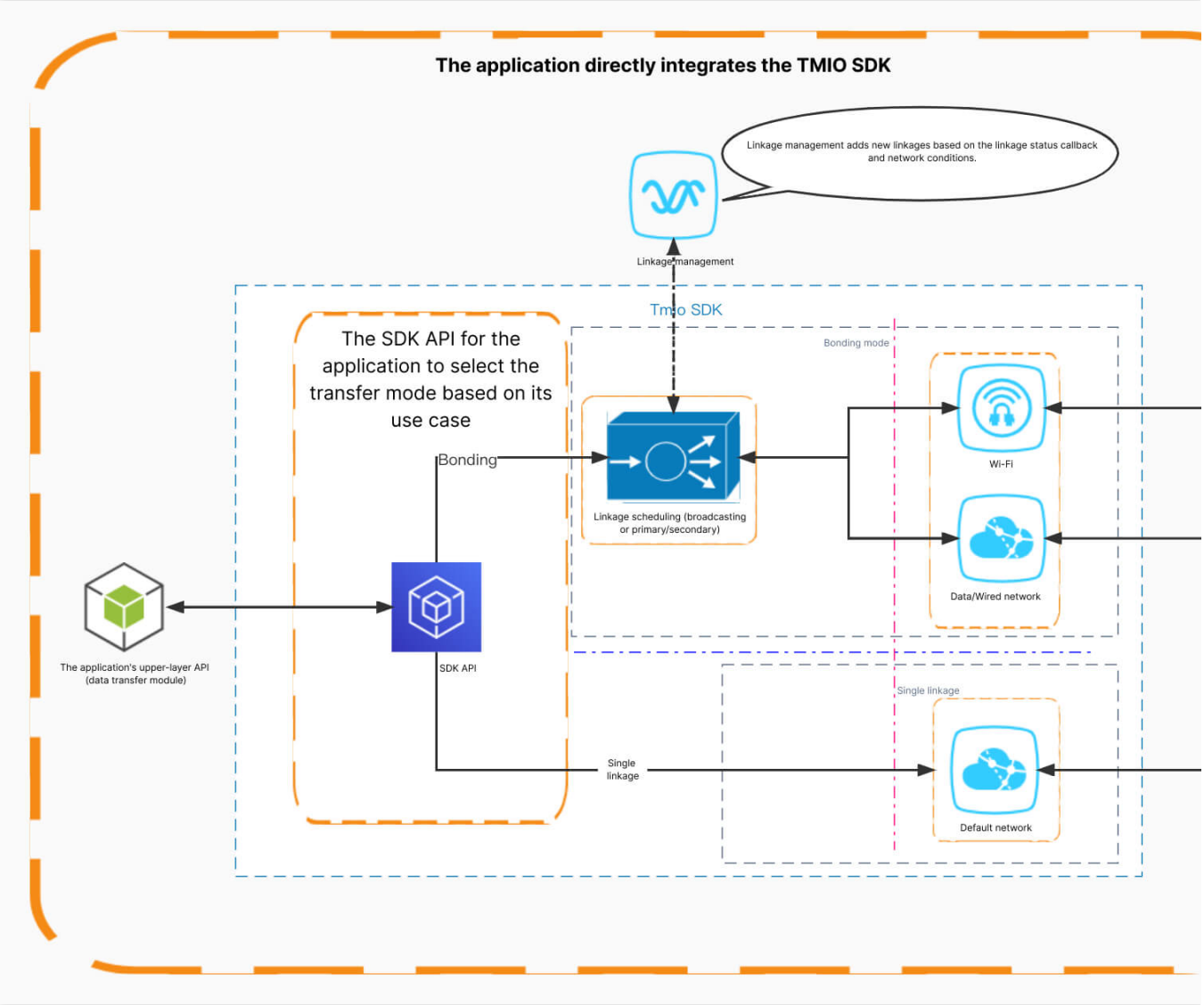
4. **停止**：



```
/*
void stop();
*/
proxy_.stop();
```

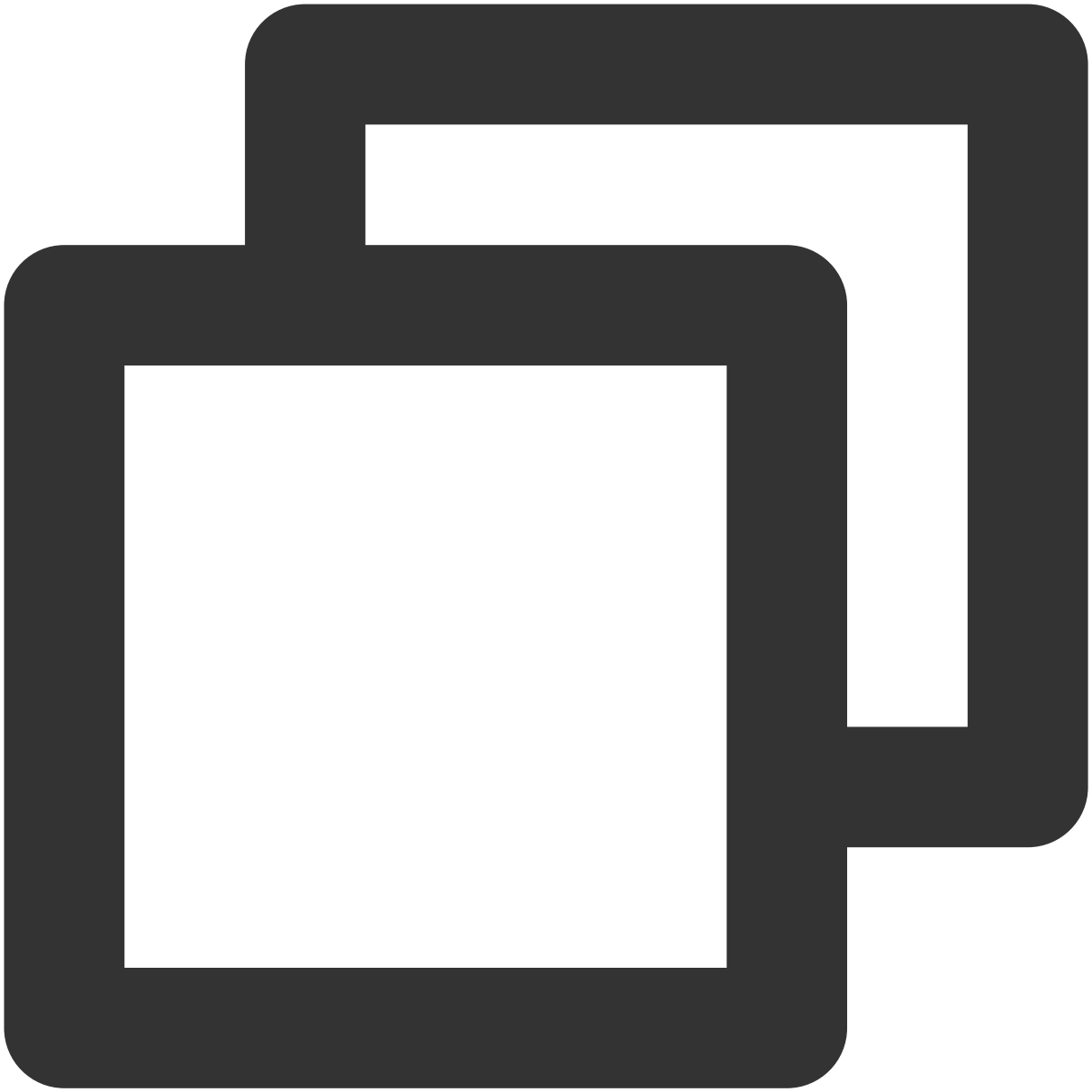
内部集成

Tmio SDK 内部集成接入方式



接入流程

1. **创建 Tmio&配置参数**（代码示例）：



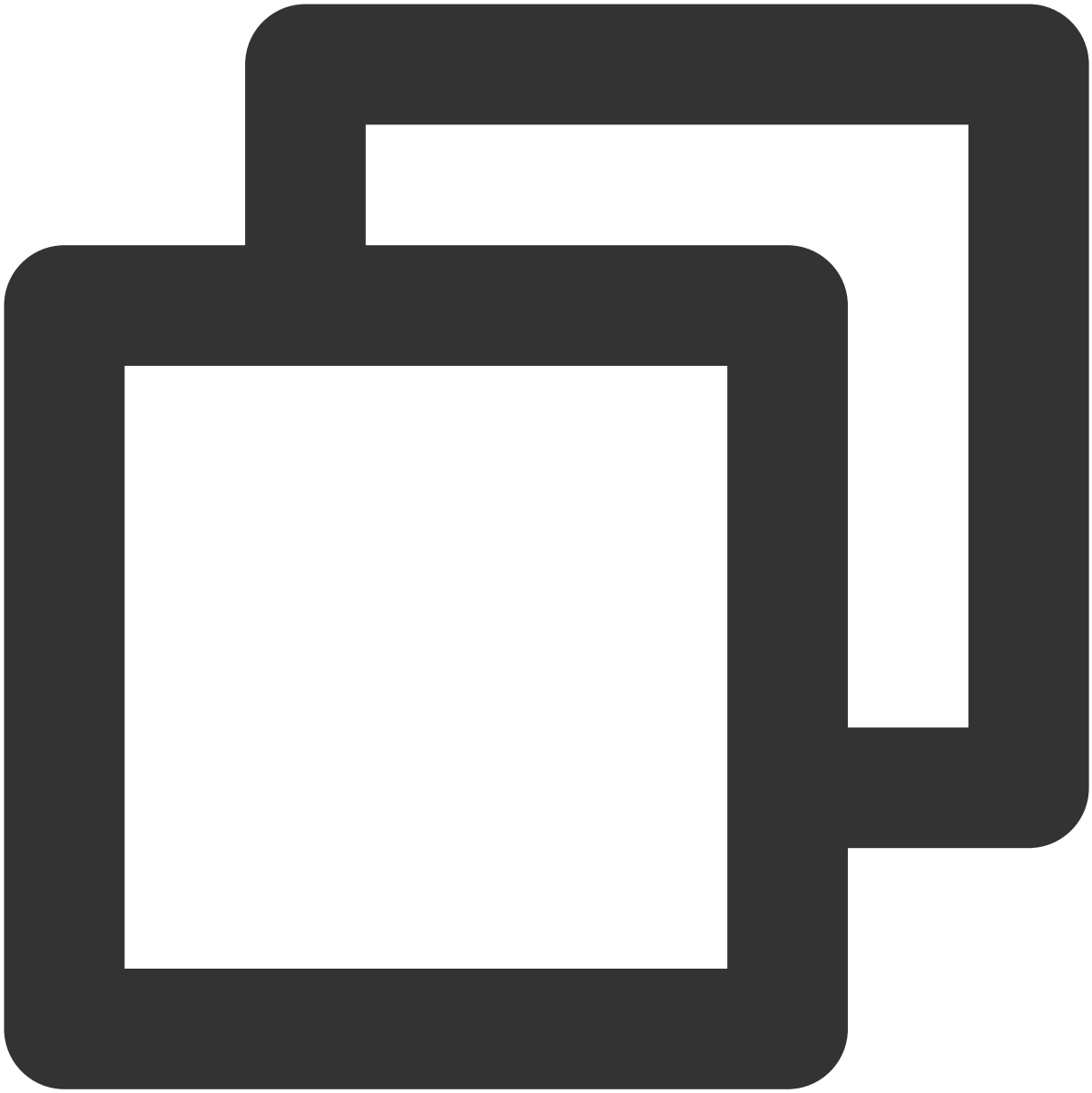
```
tmio_ = tmio::TmioFactory::createUnique(tmio::Protocol::SRT);
tmio::SrtPreset::mpegTsLossless(tmio_.get());
tmio_->setIntOption(tmio::srt_options::CONNECT_TIMEOUT, 4000);
tmio_->setBoolOption(tmio::base_options::THREAD_SAFE_CHECK, true);
```

创建 Tmio：通过 `TmioFactory` 来创建。

参数配置：根据不同参数选择不同的接口来实现配置：

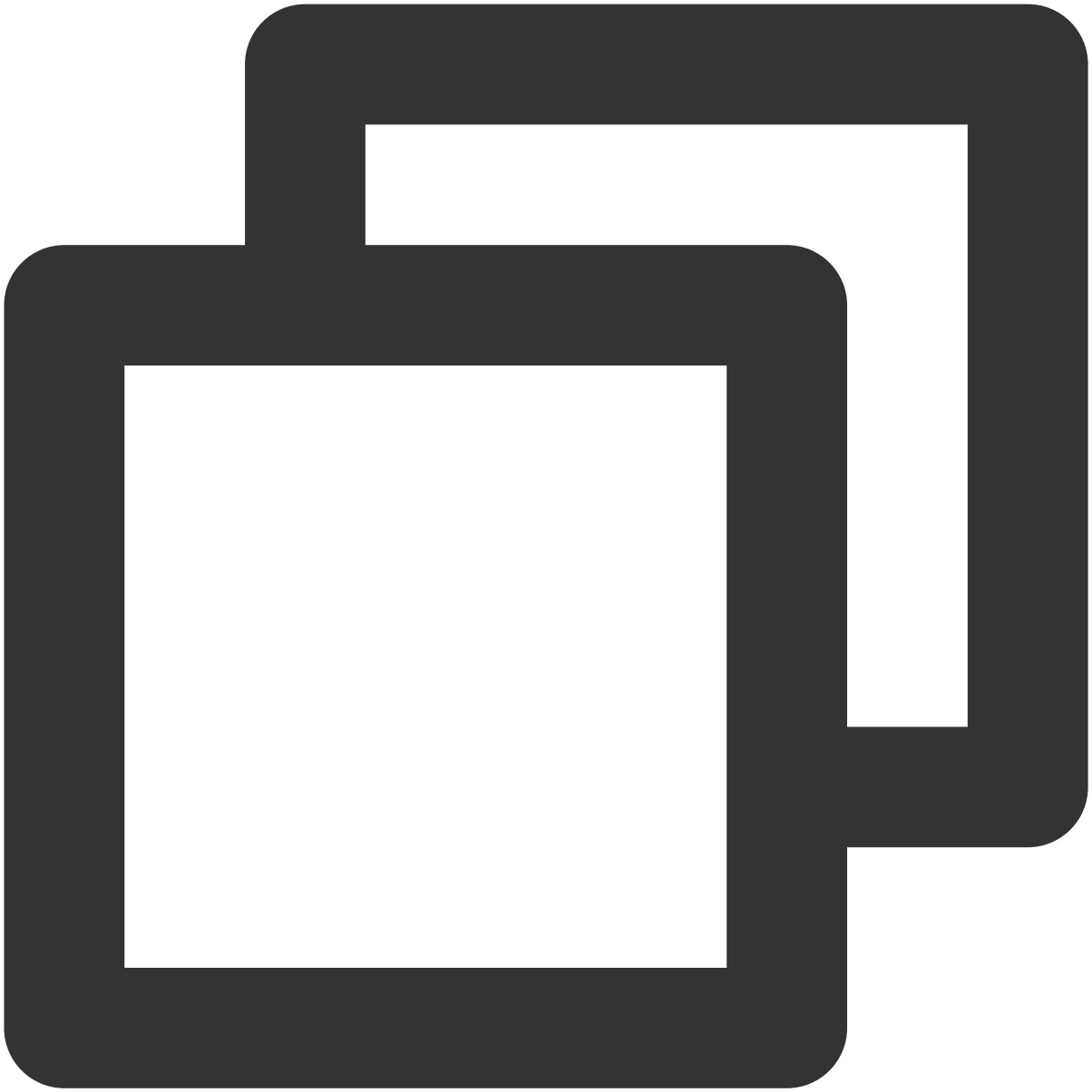
参数名：请参见 `tmio-option.h` 。

简单配置：请参见 `tmio-preset.h` 。



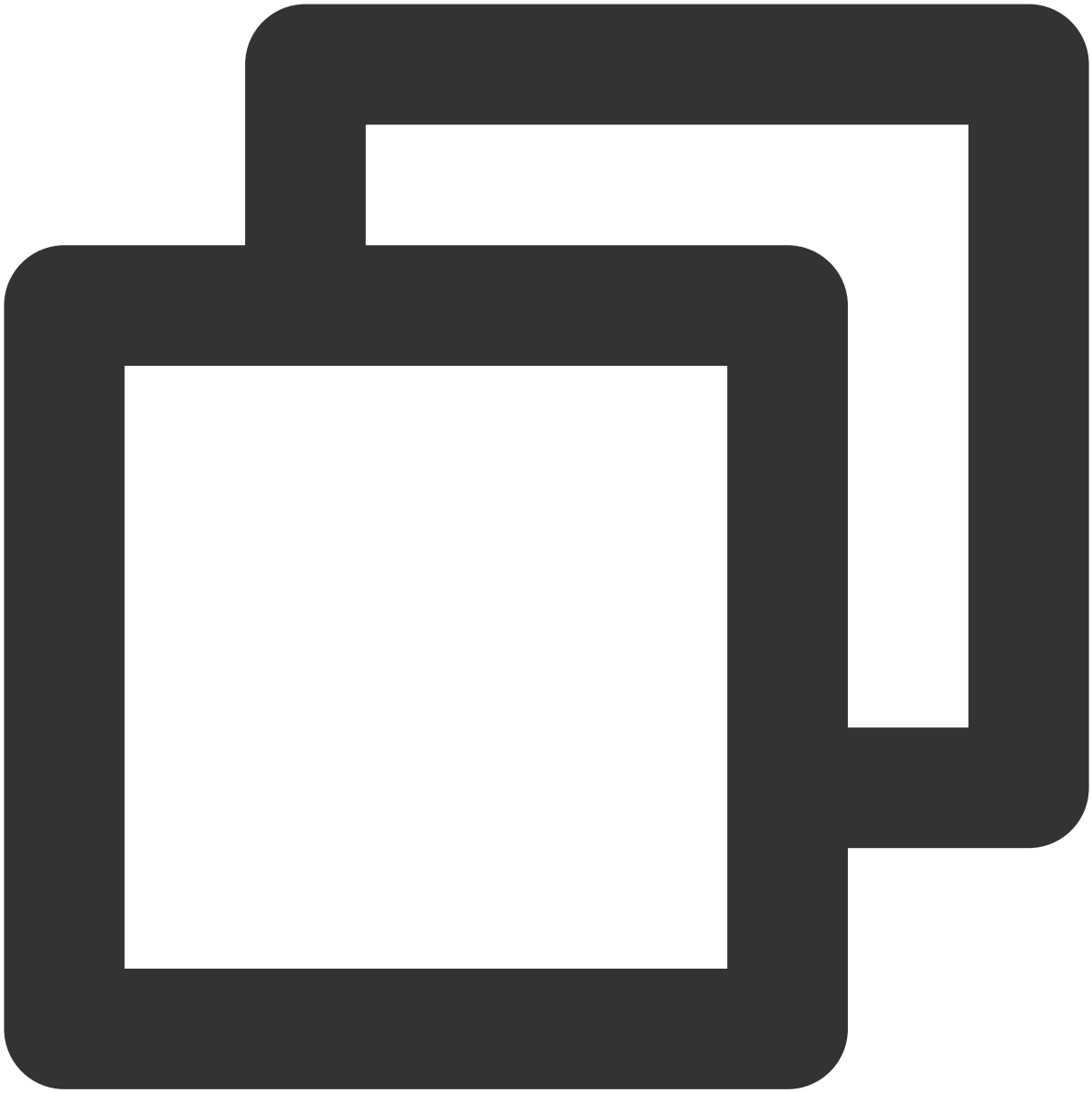
```
//根据不同参数属性选择合适的配置
bool setBoolOption(const std::string &optname, bool value);
bool setIntOption(const std::string &optname, int64_t value);
bool setDoubleOption(const std::string &optname, double value);
bool setStrOption(const std::string &optname, const std::string &value);
...
```

2. **开始连接**（代码示例）：



```
/**
 * open the stream specified by url
 *
 * @param config protocol dependent
 */
virtual std::error_code open(const std::string &url,
                             void *config = nullptr) = 0;
```

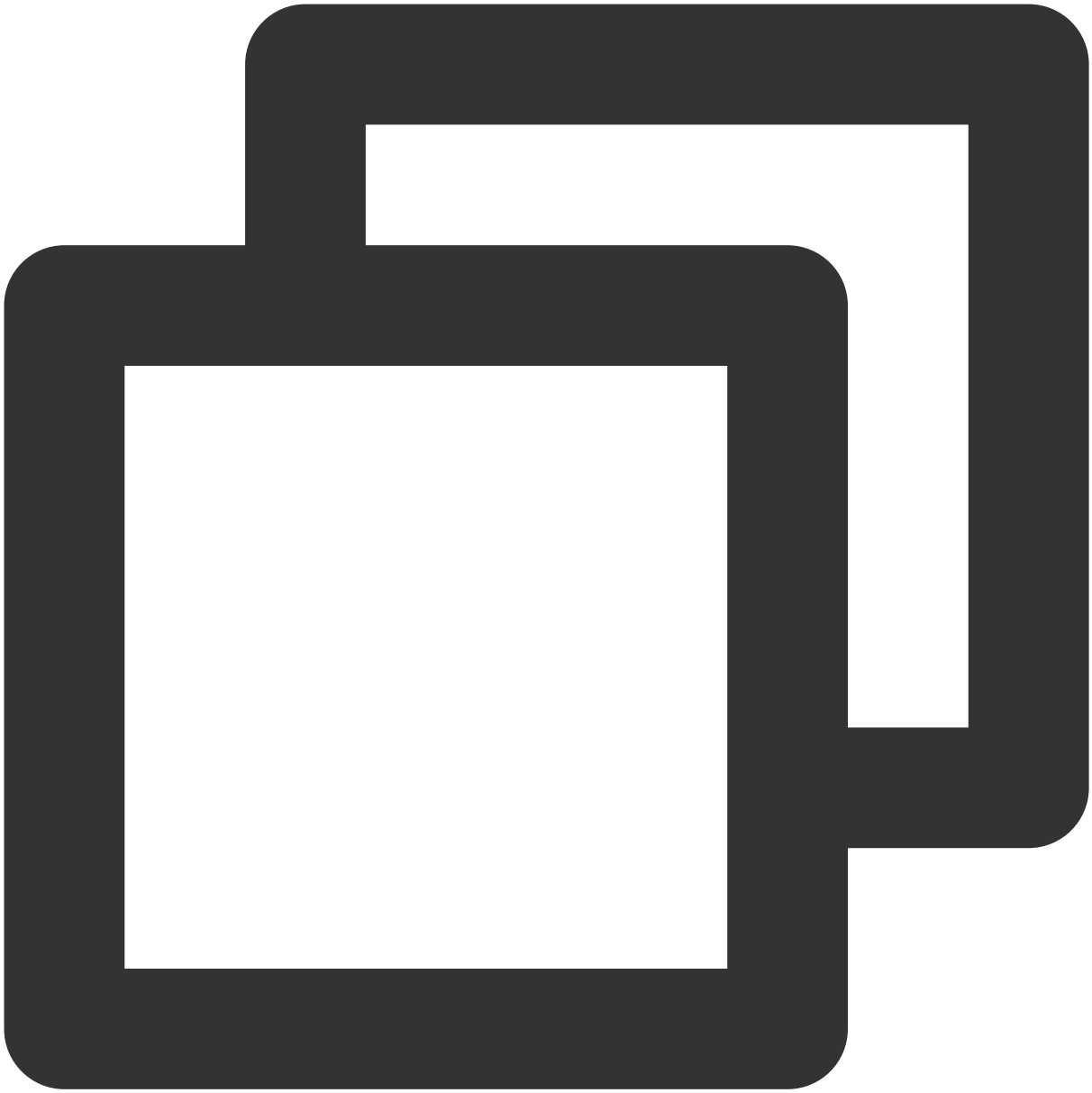
单链路（config 可为 NULL）



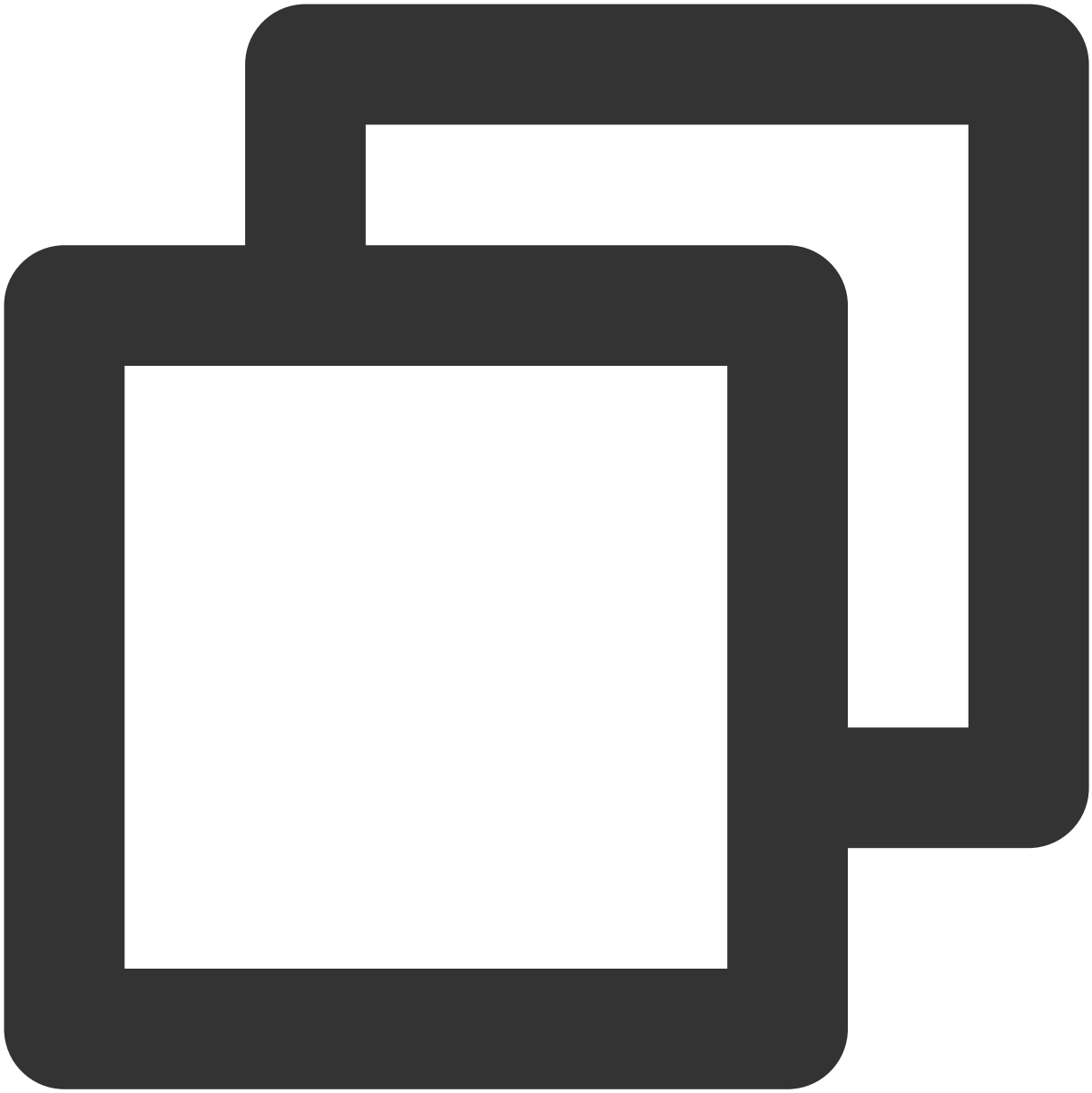
```
//默认单链路
auto err = tmio->open(TMIO_SRT_URL);
if (err) {
LOGE("open failed, %d, %s", err.value(), err.message().c_str());
}
```

多链路 bonding（当前仅支持 SRT 协议）

config 设置时 SRT bonding 配置参数可参见 `tmio.h` 文件结构中 TmioFeatureConfig 定义。



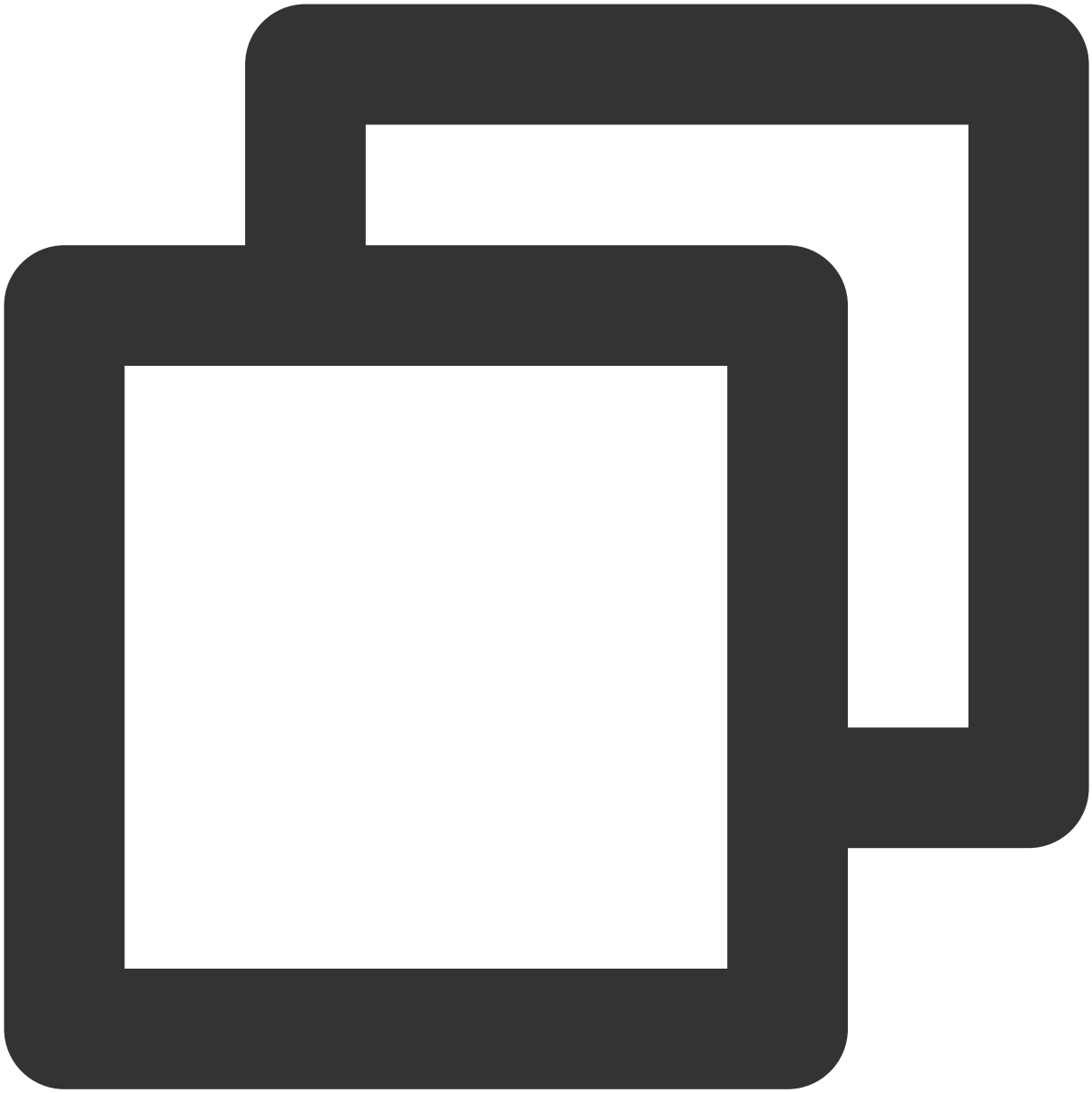
```
tmio::TmioFeatureConfig option_;
option_.protocol = tmio::Protocol::SRT;
option_.trans_mode = static_cast<int>(tmio::SrtTransMode::SRT_TRANS_BACKUP);
option_.addAvailableNet(net_name, local_addr, remote_url, 0, weight, -1);
```



```
//bonding 多链路
auto err = tmio_->open(TMIO_SRT_URL, &option_);
if (err) {
 LOGE("open failed, %d, %s", err.value(), err.message().c_str());
}
```

多链路 bonding open 接口还可以用来对 group 组添加新的链路用于传输。

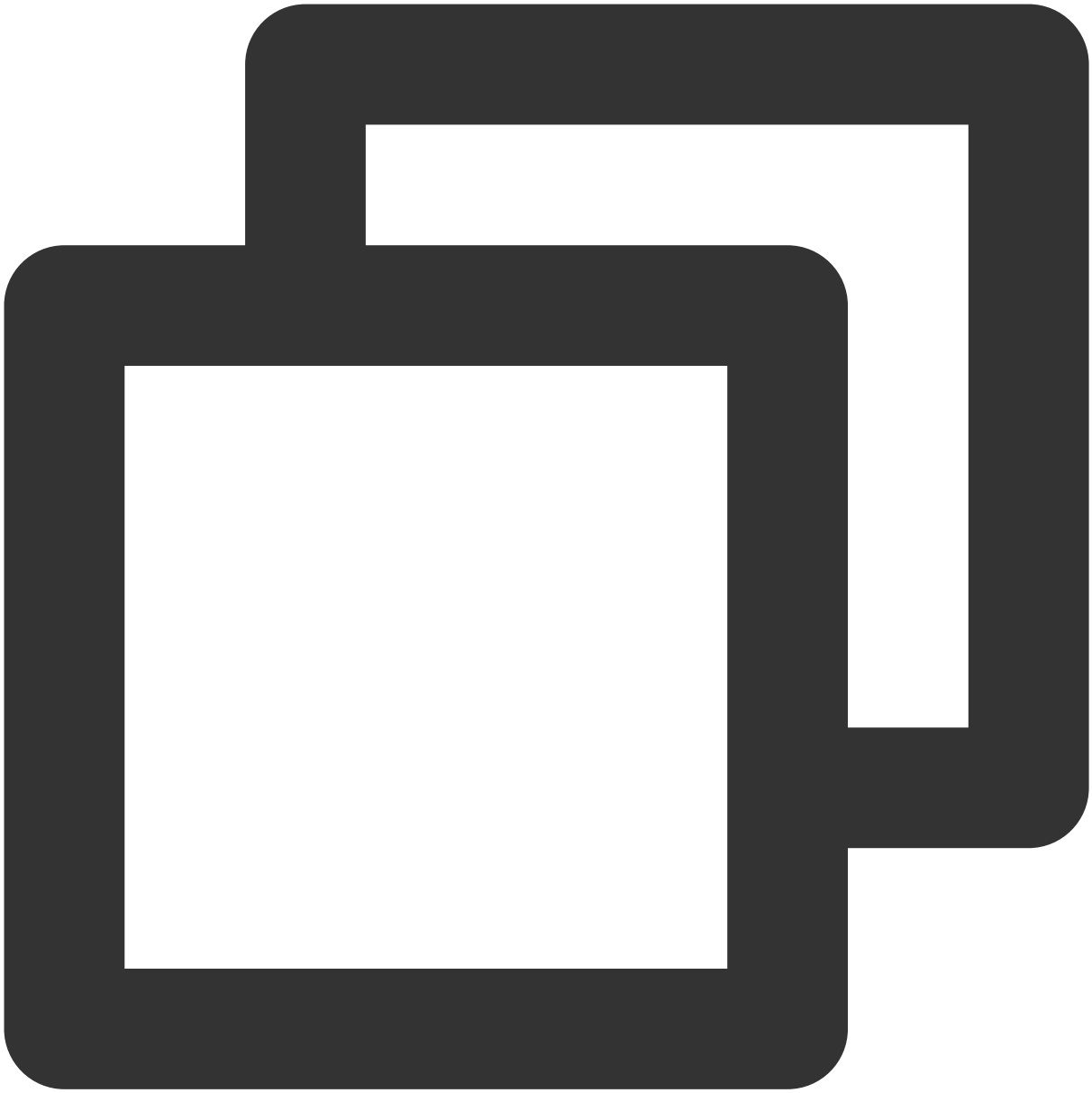
3. **发送数据**：



```
int ret = tmio_->send(buf.data(), datalen, err);
if (ret < 0) {
        LOGE("send failed, %d, %s", err.value(), err.message().c_str());
        break;
}
```

4. **接收数据**：

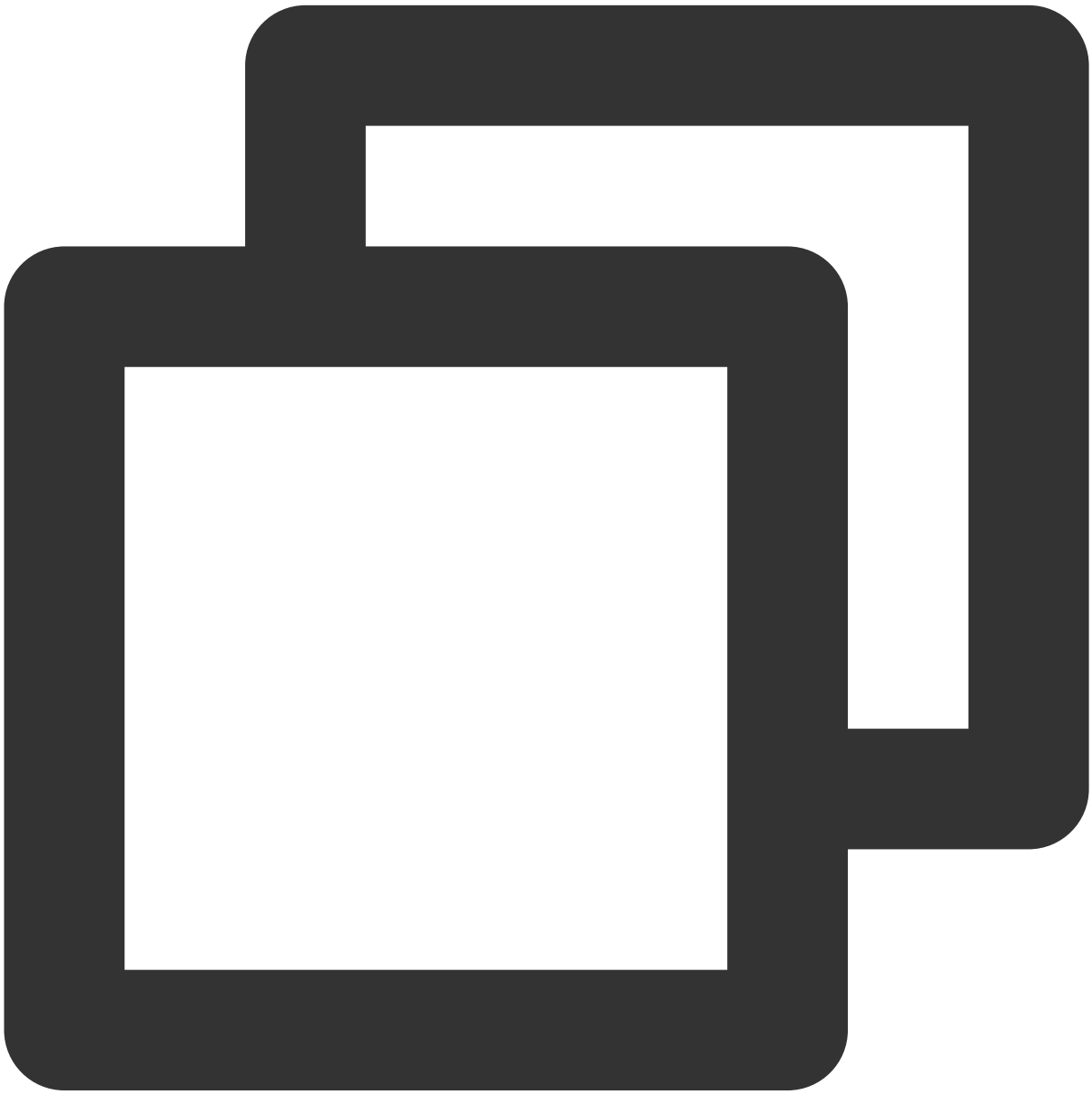
如果是需要交互的协议（如 RTMP）， 此时需要启用接收接口来读取数据完成协议交互，这里提供两个接口调用：



```
/**
* receive data
*
* @param err return error details
* @return number of bytes which were received, or < 0 to indicate error
*/
virtual int recv(uint8_t *buf, int len, std::error_code &err) = 0;

using RecvCallback = std::function<bool(const uint8_t *buf, int len, const std::err
/**
* receive data in event loop
```

```
 *
 * recvLoop() block current thread, receive data in a loop and pass the data to recv
 * @param recvCallback return true to continue the receive loop, false for break
 */
virtual void recvLoop(const RecvCallback &recvCallback) = 0;
```

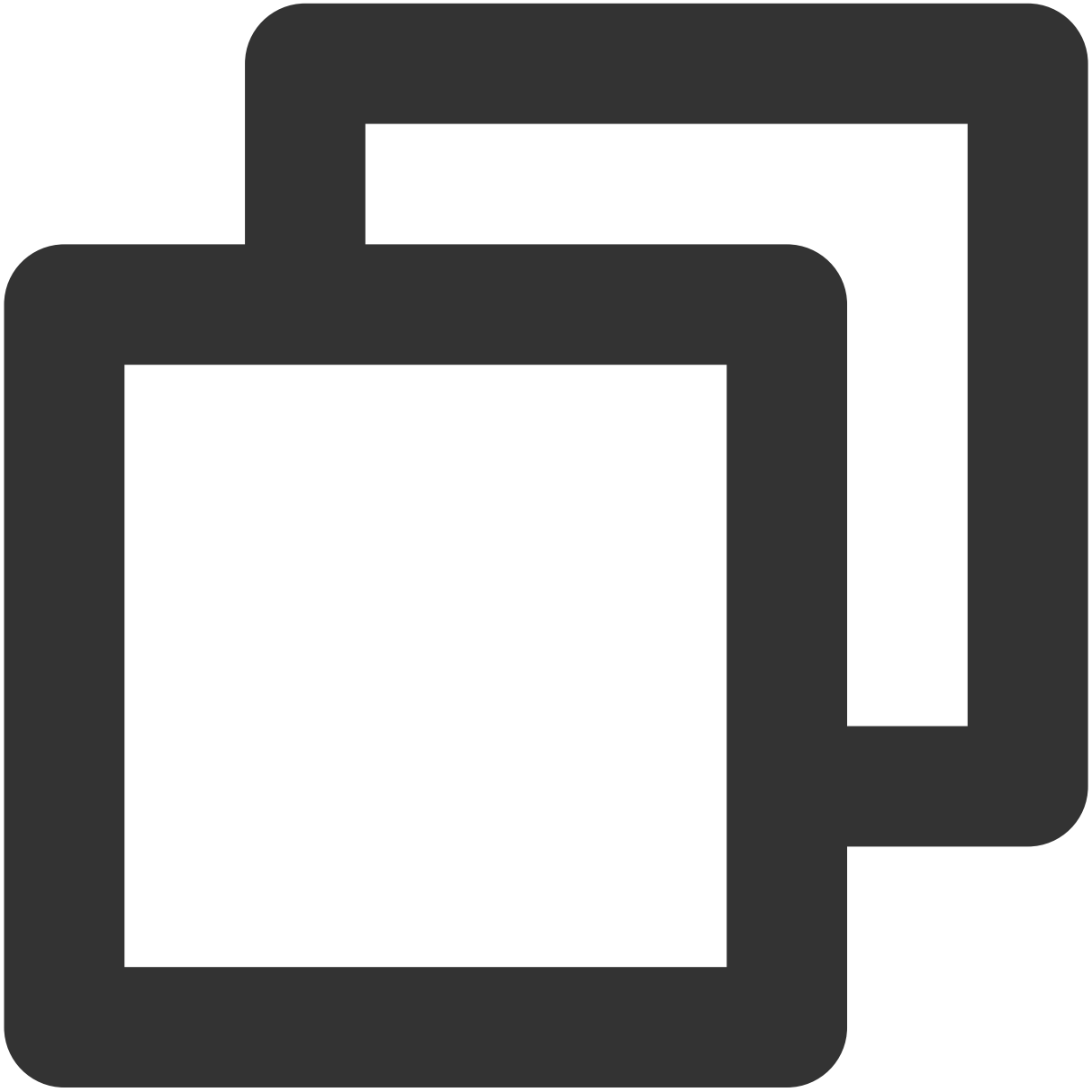
上层应用循环读取



```
    while (true) {
    ret = tmio_->recv(buf.data(), buf.size(), err);
    if (ret < 0) {
        LOGE("recv error: %d, %s", err.value(), err.message().c_str());
```

```
        break;
    }
    ...
}
```

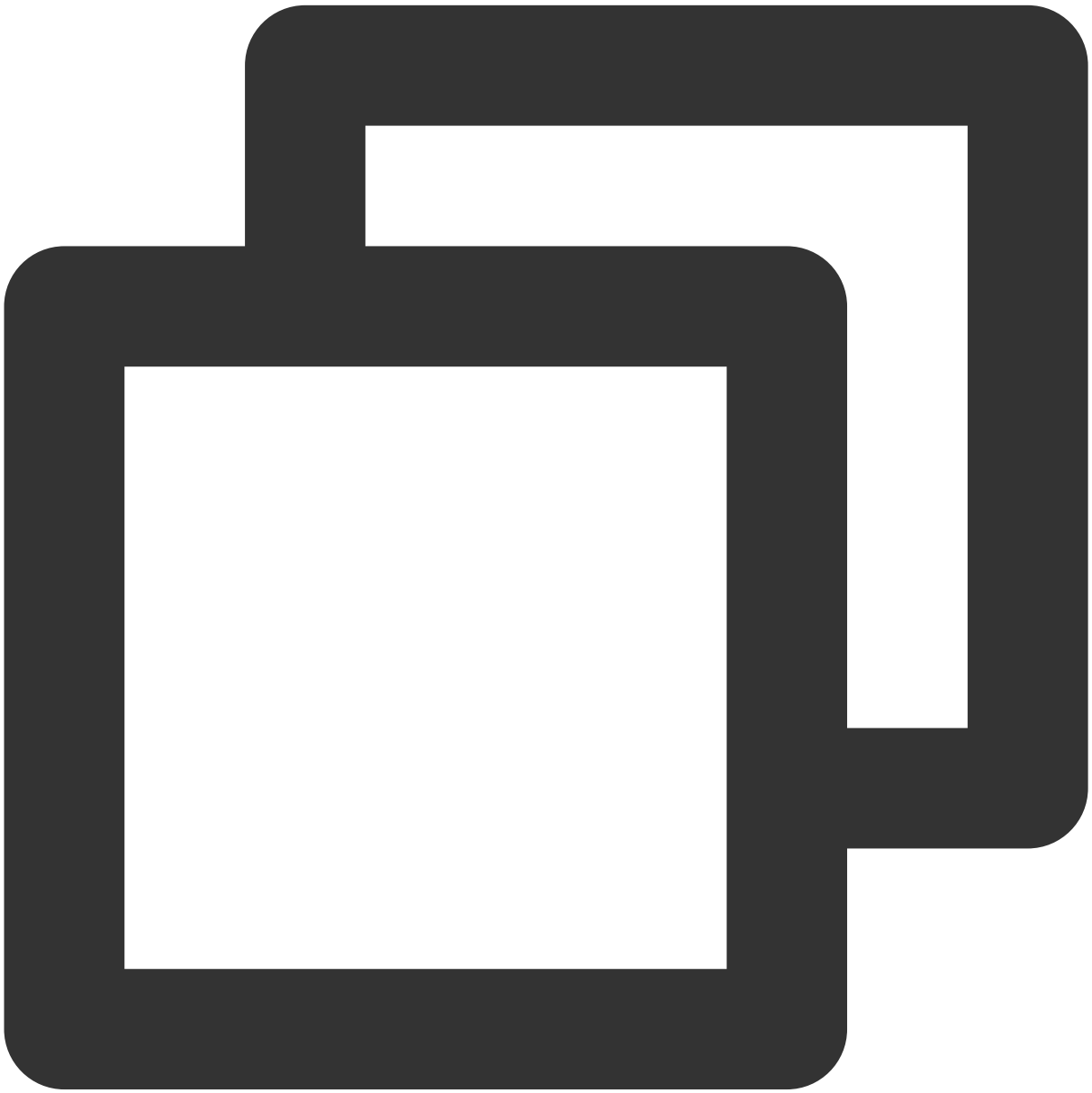
回调读取



```
FILE *file = fopen(output_path, "w");
tmio_->recvLoop([file](const uint8_t *buf, int len,
                                                const std::error_code &err) {
    if (len < 0) {
            fwrite(buf, 1, len, file);
```

```
    } else if (len < 0) {
            LOGE("recv error: %d, %s", err.value(), err.message().c_str());
    }
    return true;
});
```

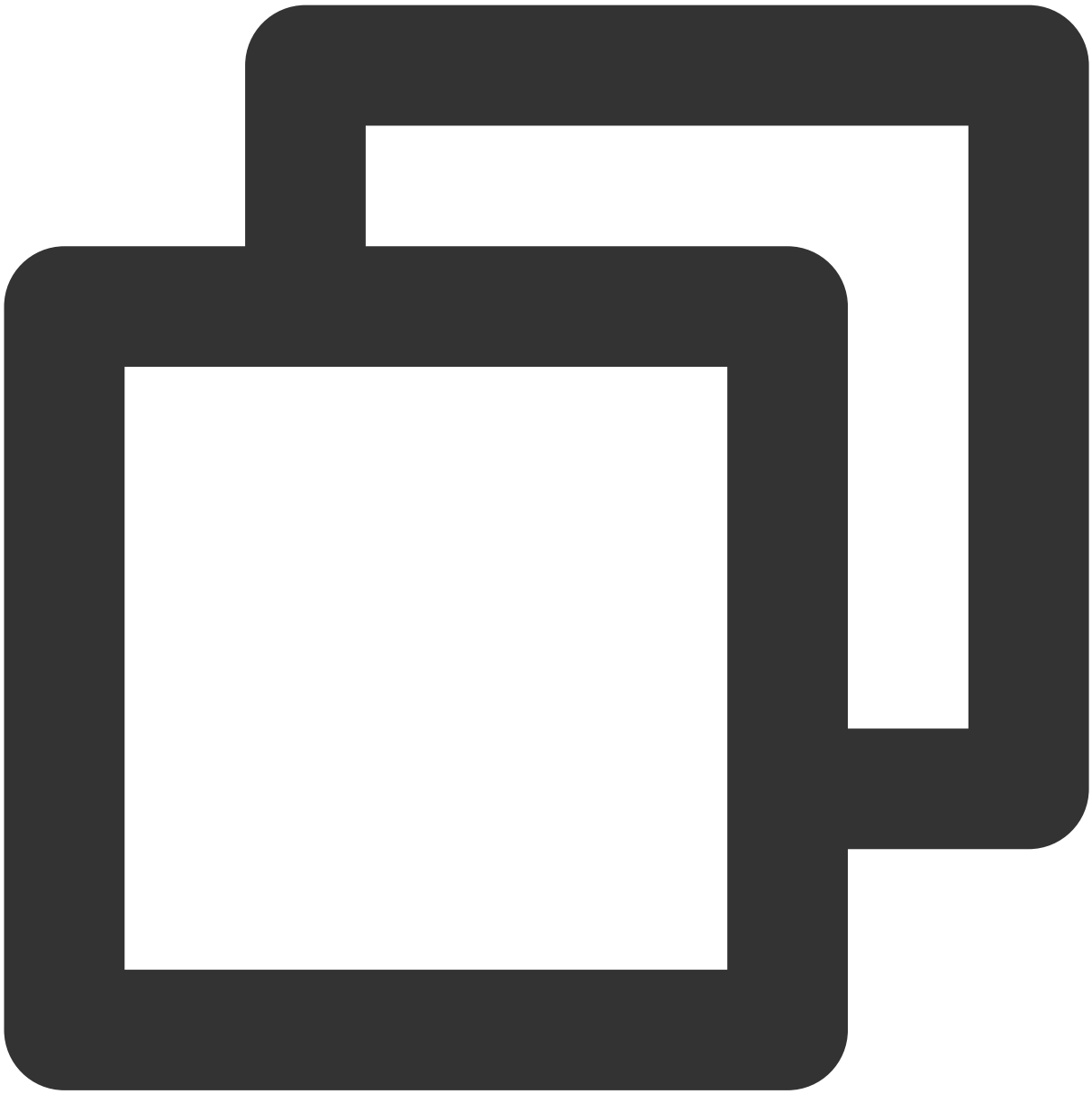
5. **关闭 Tmio**：



```
tmio_->close();
```

6. **其他**：

获取当前链路状态（应用可根据此状态信息调整推流策略）。



```
tmio::PerfStats stats_;
tmio_->control(tmio::ControlCmd::GET_STATS, &stats_);
```

最新 API 接口及 Demo 详细说明

接入 TMIO SDK，可参考最新 API 接口和 Demo 说明， 详情请参见 TMIO 接入详情。

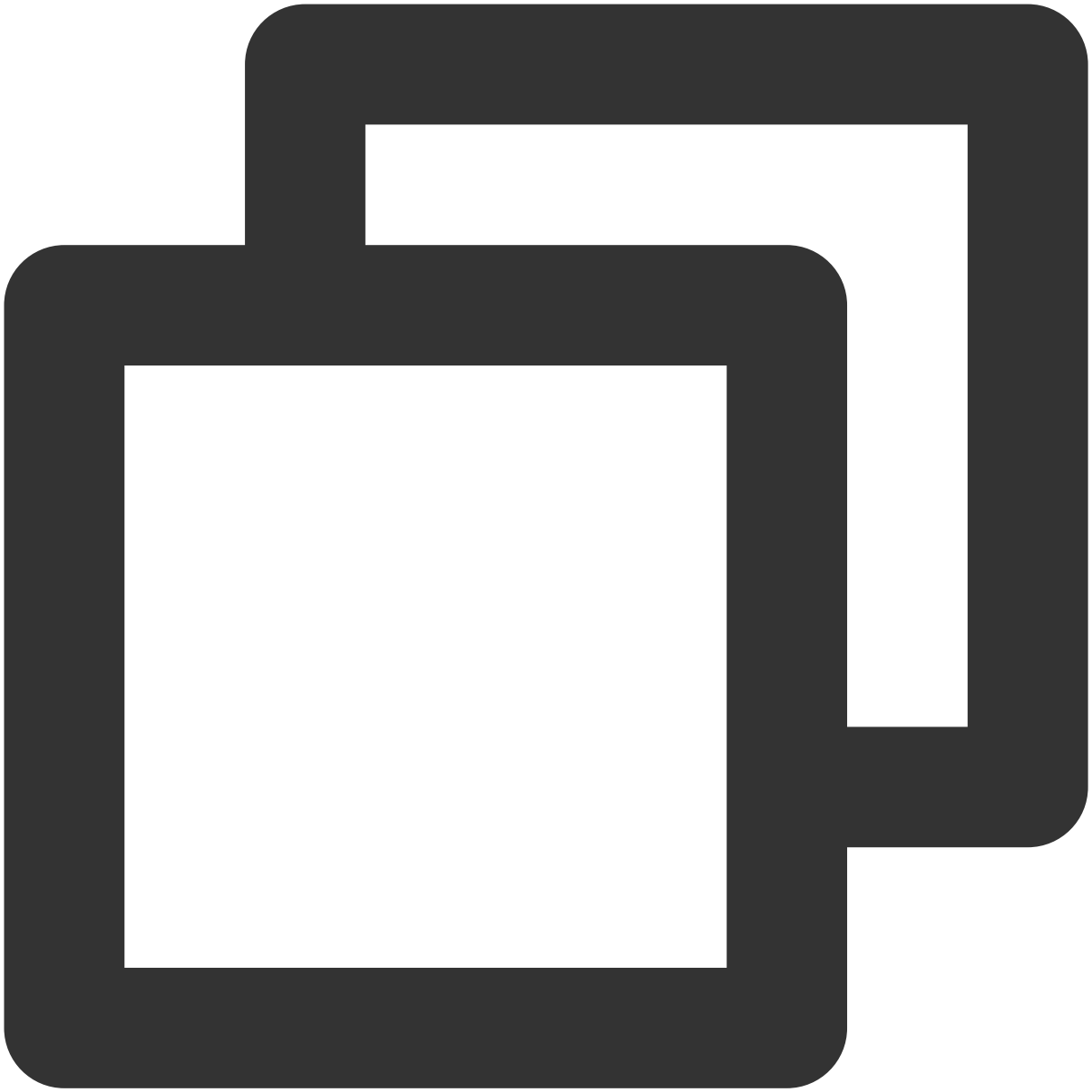
常见问题解答

是不是所有的设备都可以使用 SRT bonding 多链路功能？

多链路功能的前提是设备有多个可用的网络接口，同时针对 Android 设备其系统需要在6.0（api level >=23）以上才可使用。

Android 手机在已连接 Wi-Fi 的情况下，如何启用4G/5G数据网络？

Android 手机在已连接 Wi-Fi 的情况下，是无法直接使用4G/5G网络来实现数据传输的，此时如果想启用数据网络则需要申请数据网络权限，代码如下：



```
ConnectivityManager connectivityManager = (ConnectivityManager) context.getSystemSe
NetworkRequest request = new NetworkRequest.Builder().addTransportType(NetworkCapab
                                            .addCapability(NetworkCapabilities.NET_C
                                            .build();

ConnectivityManager.NetworkCallback networkCallback = new ConnectivityManager.Netwo
    @Override
    public void onAvailable(@NonNull Network network) {
                Log.d(TAG, "移动数据网络通道已开启.");
        super.onAvailable(network);
    }
```

```
}
```

快直播传输层 SDK 播放器集成指引

最近更新时间：2022-12-23 10:55:22

简单介绍

快直播传输层 SDK（libLebConnection）提供基于原生 WebRTC 升级版的传输能力，用户仅需对已有播放器进行简单改造，即可接入快直播。在完全兼容标准直播的推流、云端媒体处理能力的基础上，实现高并发低延迟直播，帮助用户实现从现有的标准直播平滑地迁移到快直播上来。也可以帮助用户在现有 RTC 场景中快速实现低成本的大房间低延迟旁路直播。

功能介绍

- 音视频拉流，兼具优异的低延迟性能和抗弱网能力
- 视频支持H.264、H.265和 AV1，支持 B 帧，视频输出格式为视频帧裸数据（H.264/H.265为 AnnexB，AV1 为 OBU）
- 音频支持 AAC 和 OPUS，音频输出格式为音频帧裸数据
- 支持 Android、iOS、Windows、Linux 和 Mac 平台

接入方式

基础接口说明

- 创建快直播连接

```
LEB_EXPORT_API LebConnectionHandle* OpenLebConnection(void* context, LebLogLeve
l loglevel);
```

- 注册回调函数

```
LEB_EXPORT_API void RegisterLebCallback(LebConnectionHandle* handle, const LebC
allback* callback);
```

- 开始连接拉流

```
LEB_EXPORT_API void StartLebConnection(LebConnectionHandle* handle, LebConfig c
onfig);
```

- 停止连接

```
LEB_EXPORT_API void StopLebConnection(LebConnectionHandle* handle);
```

- 关闭连接

```
LEB_EXPORT_API void CloseLebConnection(LebConnectionHandle* handle);
```

回调接口说明

```
typedef struct LebCallback {
// 日志回调
OnLogInfo onLogInfo;
// 视频信息回调
OnVideoInfo onVideoInfo;
// 音频信息回调
OnAudioInfo onAudioInfo;
// 视频数据回调
OnEncodedVideo onEncodedVideo;
// 音频数据回调
OnEncodedAudio onEncodedAudio;
// MetaData回调
OnMetaData onMetaData;
// 统计信息回调
OnStatsInfo onStatsInfo;
// 错误回调
OnError onError;
} LebCallback;
```

> 注意：
> 详细数据结构定义请见头文件 `leb_connection_api.h` 。

接口调用流程

1. **创建 LEB 连接**：OpenLebConnection()

2. **注册各种回调函数**：RegisterXXXXCallback()
3. **开始连接拉流**：StartLebConnection()
4. **音视频裸数据回调输出**：

- OnEncodedVideo()
- OnEncodedAudio()

5. **停止连接**：StopLebConnection()
6. **关闭连接**：CloseLebConnection

接入示例

本 示例 基于 Android 端使用广泛的具有代表性开源播放器 ijkplayer，介绍接入快直播传输层 SDK 的方法及流程，其他平台可参考进行集成。

最新 SDK 下载

快直播传输层 libLebConnection SDK 请参见 SDK 下载。

集成常见问题解答

开发卡顿统计功能

由于关闭了 buffering，现可以通过统计渲染刷新时间间隔来统计卡顿参数。当视频渲染时间间隔大于一定阈值，记一次卡顿次数，并累计进卡顿时长。
以 ijkplayer 为例，以下内容演示卡顿统计的开发流程。

1. **代码修改**
   i. 在 VideoState、FFPlayer 结构体中添加卡顿统计需要用到的变量。

```
diff --git a/ijkmedia/ijkplayer/ff_ffplay_def.h b/ijkmedia/ijkplayer/ff_ffplay_de
f.h
index 00f19f3c..f38a790c 100755
--- a/ijkmedia/ijkplayer/ff_ffplay_def.h
+++ b/ijkmedia/ijkplayer/ff_ffplay_def.h
@@ -418,6 +418,14 @@ typedef struct VideoState {
SDL_cond *audio_accurate_seek_cond;
volatile int initialized_decoder;
int seek_buffering;
+
+ int64_t stream_open_time;
+ int64_t first_frame_display_time;
```

```
+ int64_t last_display_time;
+ int64_t current_display_time;
+ int64_t frozen_time;
+ int frozen_count;
+ float frozen_rate;
} VideoState;
/* options specified by the user */
@@ -720,6 +728,14 @@ typedef struct FFPlayer {
char *mediacodec_default_name;
int ijkmeta_delay_init;
int render_wait_start;
int low_delay_playback;
+ int frozen_interval;
int high_level_ms;
int low_level_ms;
int64_t update_plabyback_rate_time;
int64_t update_plabyback_rate_time_prev;
} FFPlayer;
#define fftime_to_milliseconds(ts) (av_rescale(ts, 1000, AV_TIME_BASE))
@@ -844,6 +860,15 @@ inline static void ffp_reset_internal(FFPlayer *ffp)
ffp->pf_playback_volume = 1.0f;
ffp->pf_playback_volume_changed = 0;
ffp->low_delay_playback = 0;
ffp->high_level_ms = 500;
ffp->low_level_ms = 200;
+ ffp->frozen_interval = 200;
ffp->update_plabyback_rate_time = 0;
ffp->update_plabyback_rate_time_prev = 0;
av_application_closep(&ffp->app_ctx);
ijkio_manager_destroyp(&ffp->ijkio_manager_ctx);
```

ii. **添加卡顿统计逻辑**

```
diff --git a/ijkmedia/ijkplayer/ff_ffplay.c b/ijkmedia/ijkplayer/ff_ffplay.c
index 714a8c9d..c7368ff5 100755
--- a/ijkmedia/ijkplayer/ff_ffplay.c
+++ b/ijkmedia/ijkplayer/ff_ffplay.c
@@ -874,6 +874,25 @@ static void video_image_display2(FFPlayer *ffp)
VideoState *is = ffp->is;
Frame *vp;
Frame *sp = NULL;
+ int64_t display_interval = 0;
+
+ if (!is->first_frame_display_time){
+ is->first_frame_display_time = SDL_GetTickHR() - is->stream_open_time;
+ }
+
```

```
+ is->last_display_time = is->current_display_time;
+ is->current_display_time = SDL_GetTickHR() - is->stream_open_time;
+ display_interval = is->current_display_time - is->last_display_time;
+ av_log(NULL, AV_LOG_DEBUG, "last_display_time:%"PRId64" current_display_tim
e:%"PRId64" display_interval:%"PRId64"\n", is->last_display_time, is->current_dis
play_time, display_interval);
+
+ if (is->last_display_time > 0) {
+ if (display_interval > ffp->frozen_interval) {
+ is->frozen_count += 1;
+ is->frozen_time += display_interval;
+ }
+ }
+ is->frozen_rate = (float) is->frozen_time / is->current_display_time;
+ av_log(NULL, AV_LOG_DEBUG, "frozen_interval:%d frozen_count:%d frozen_time:%"PR
Id64" is->current_display_time:%"PRId64" frozen_rate: %f ", ffp->frozen_interval,
is->frozen_count, is->frozen_time, is->current_display_time, is->frozen_rate);
vp = frame_queue_peek_last(&is->pictq);
```

> 注意：
> 本示例中卡顿阈值 frozen_interval 初始化值为 `200(ms)`，可根据业务需要进行调整。

2. **卡顿参数统计测试**
   使用 QNET 模拟弱网环境进行测试，步骤如下：
   i. 下载 QNET 网络测试工具。
   ii. 打开QNET，单击**新增** > **模版类型** > **自定义模版**，根据需要配置弱网模版和参数（下图配置为下行网络30%随机丢包）。
   iii. 在程序列表中选择测试程序。
   iv. 开启弱网进行测试。

> 注意：
> 为便于测试，可将上述卡顿参数的修改，通过 jni 将数据传递到 Java 层进行显示。

解决播放有噪音问题（Android soundtouch 优化）

根据 buffer 水位调整播放速率的修改，会用到 soundtouch 进行音频变速实现。但在网络波动较大、buffer 水位调整频率高需要多次变速处理时，原生 ijkplayer 对于 soundtouch 的调用可能出现噪音问题，可参考如下代码进行优化：

在调用 soundtouch 进行变速处理时，如果是低延时播放模式，则全部 buffer 都是用 soundtouch 进行 translate。

```
diff --git a/ijkmedia/ijkplayer/ff_ffplay.c b/ijkmedia/ijkplayer/ff_ffplay.c
index 714a8c9d..c7368ff5 100755
--- a/ijkmedia/ijkplayer/ff_ffplay.c
+++ b/ijkmedia/ijkplayer/ff_ffplay.c
@@ -2579,7 +2652,7 @@ reload:
int bytes_per_sample = av_get_bytes_per_sample(is->audio_tgt.fmt);
resampled_data_size = len2 * is->audio_tgt.channels * bytes_per_sample;
#if defined(__ANDROID__)
- if (ffp->soundtouch_enable && ffp->pf_playback_rate != 1.0f && !is->abort_reque
st) {
+ if (ffp->soundtouch_enable && (ffp->pf_playback_rate != 1.0f || ffp->low_delay_
playback) && !is->abort_request) {
av_fast_malloc(&is->audio_new_buf, &is->audio_new_buf_size, out_size * translate_
time);
for (int i = 0; i < (resampled_data_size / 2); i++)
{
```

解决打开 MediaCodec 后，H.265不使用 MediaCodec 解码问题

快直播传输层的特色是支持 H.265 格式的视频流，但是原生 ijkplayer 在 **Settings** 中勾选打开 `Using MediaCodec` 后，H.265的视频流不会使用 MediaCodec 进行解码。可参考如下代码进行优化：

```
diff --git a/ijkmedia/ijkplayer/ff_ffplay_options.h b/ijkmedia/ijkplayer/ff_ffpla
y_options.h
index b021c26e..958b3bae 100644
--- a/ijkmedia/ijkplayer/ff_ffplay_options.h
+++ b/ijkmedia/ijkplayer/ff_ffplay_options.h
@@ -178,8 +178,8 @@ static const AVOption ffp_context_options[] = {
OPTION_OFFSET(vtb_handle_resolution_change), OPTION_INT(0, 0, 1) },

// Android only options
- { "mediacodec", "MediaCodec: enable H.264 (deprecated by 'mediacodec-avc')",
- OPTION_OFFSET(mediacodec_avc), OPTION_INT(0, 0, 1) },
+ { "mediacodec", "MediaCodec: enable all_videos (deprecated by 'mediacodec_all_v
ideos')",
+ OPTION_OFFSET(mediacodec_all_videos), OPTION_INT(0, 0, 1) },
{ "mediacodec-auto-rotate", "MediaCodec: auto rotate frame depending on meta",
OPTION_OFFSET(mediacodec_auto_rotate), OPTION_INT(0, 0, 1) },
{ "mediacodec-all-videos", "MediaCodec: enable all videos",
```

Web 端本地混流

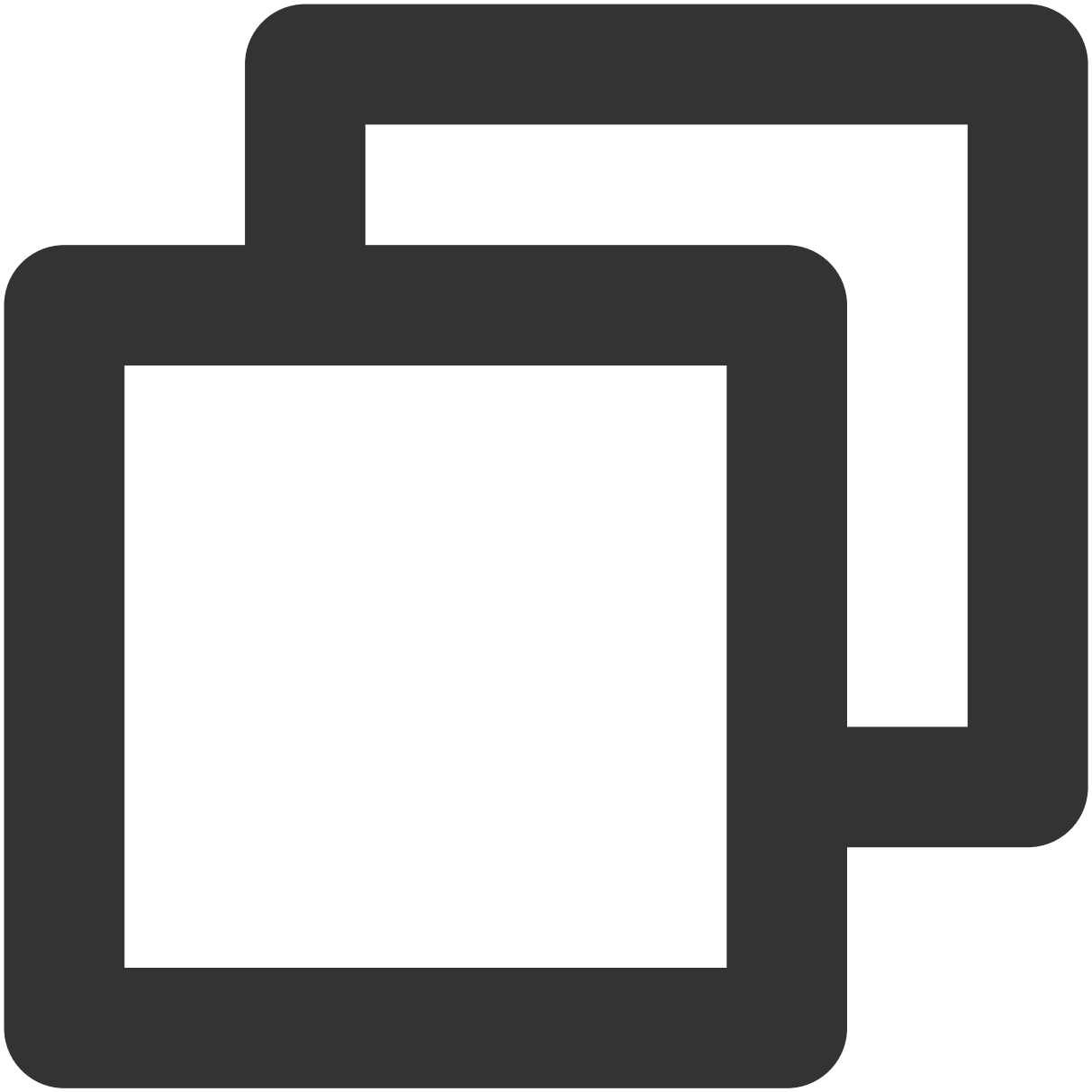
最近更新时间：2023-11-29 16:55:35

SDK 提供了对视频流画面的处理功能，包括多路视频流的混合（画中画）、画面效果的处理（镜像、滤镜）和其他元素的添加（水印、文本）。基本步骤是为：SDK 首先采集多路流，然后对多路流进行本地混流处理，对画面进行合并，声音进行混合，最后再进行其他效果处理。这些都依赖于浏览器本身功能的支持，因此对浏览器的性能有一定要求。具体的接口协议可以参考 TXVideoEffectManager，下面简单介绍本地混流的基础用法。

基础使用

使用本地混流功能需要完成SDK的初始化并获取 SDK 实例 livePusher，初始化代码请参见 对接攻略。

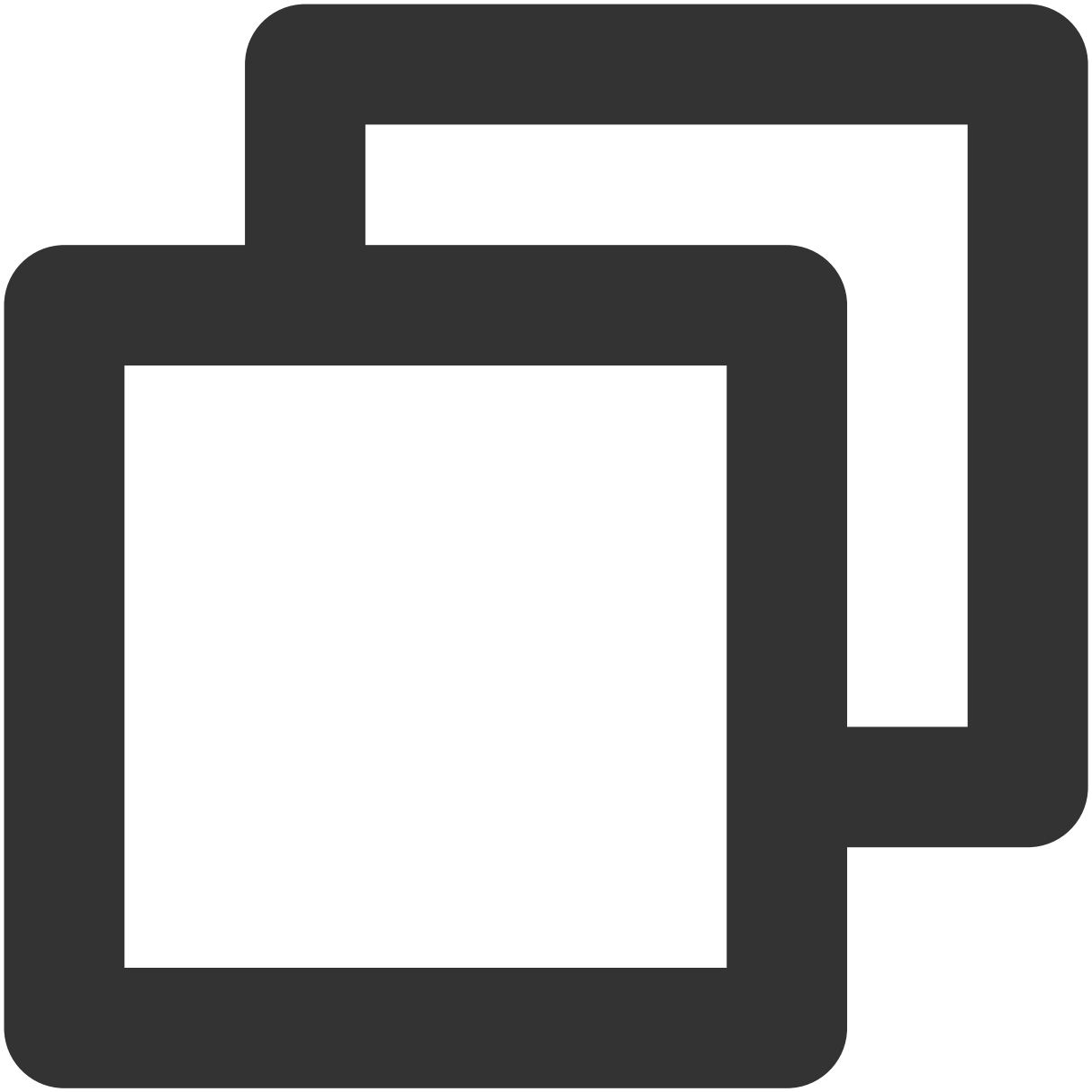
步骤1：获取视频效果管理实例



```
var videoEffectManager = livePusher.getVideoEffectManager();
```

步骤2：开启本地混流

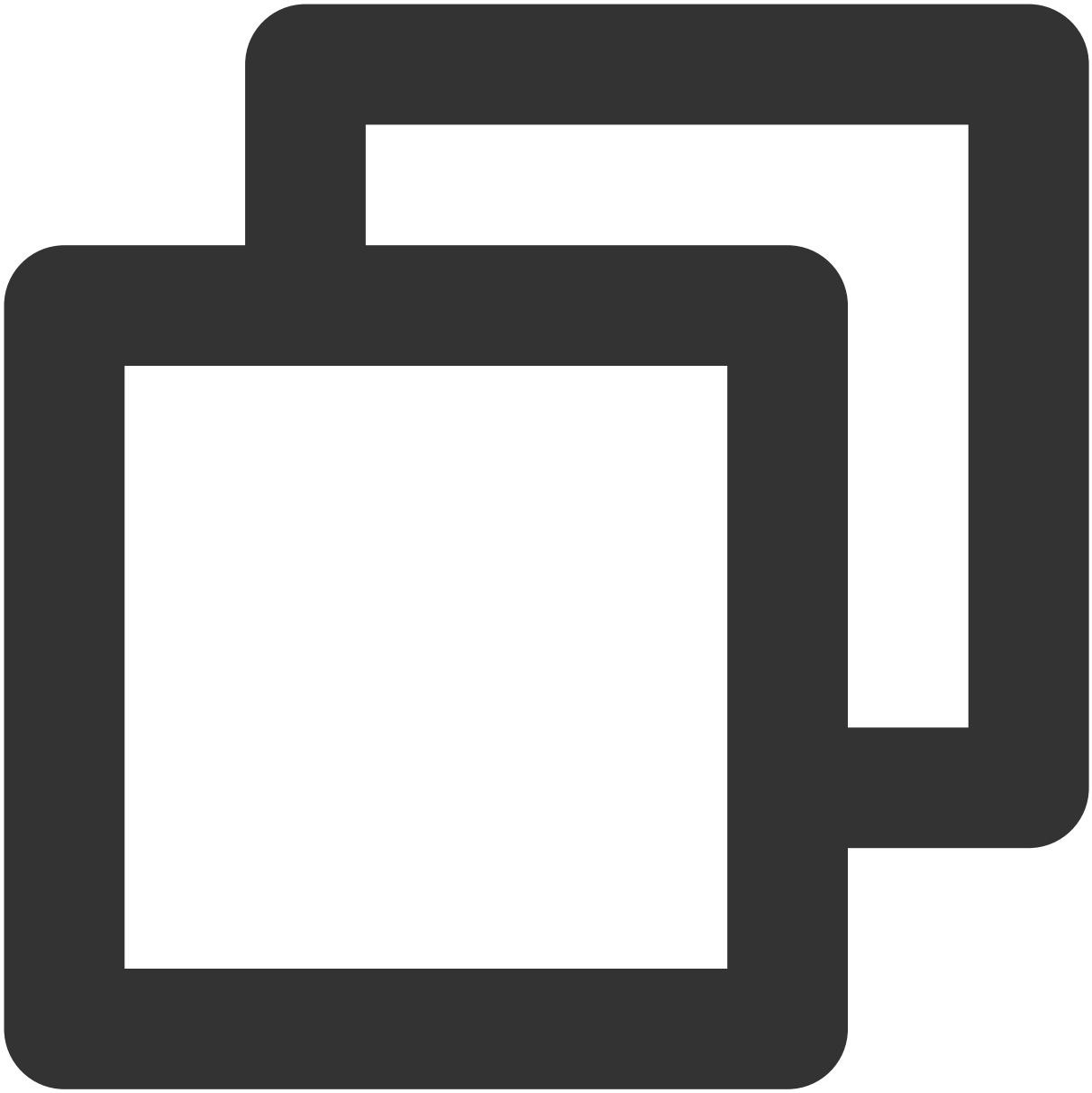
首先需要启用本地混流功能。默认情况下 SDK 只支持采集一路视频流和一路音频流，启用之后，就可以采集多路流，这些流将在浏览器本地进行混合处理。



```
videoEffectManager.enableMixing(true);
```

步骤3：设置混流参数

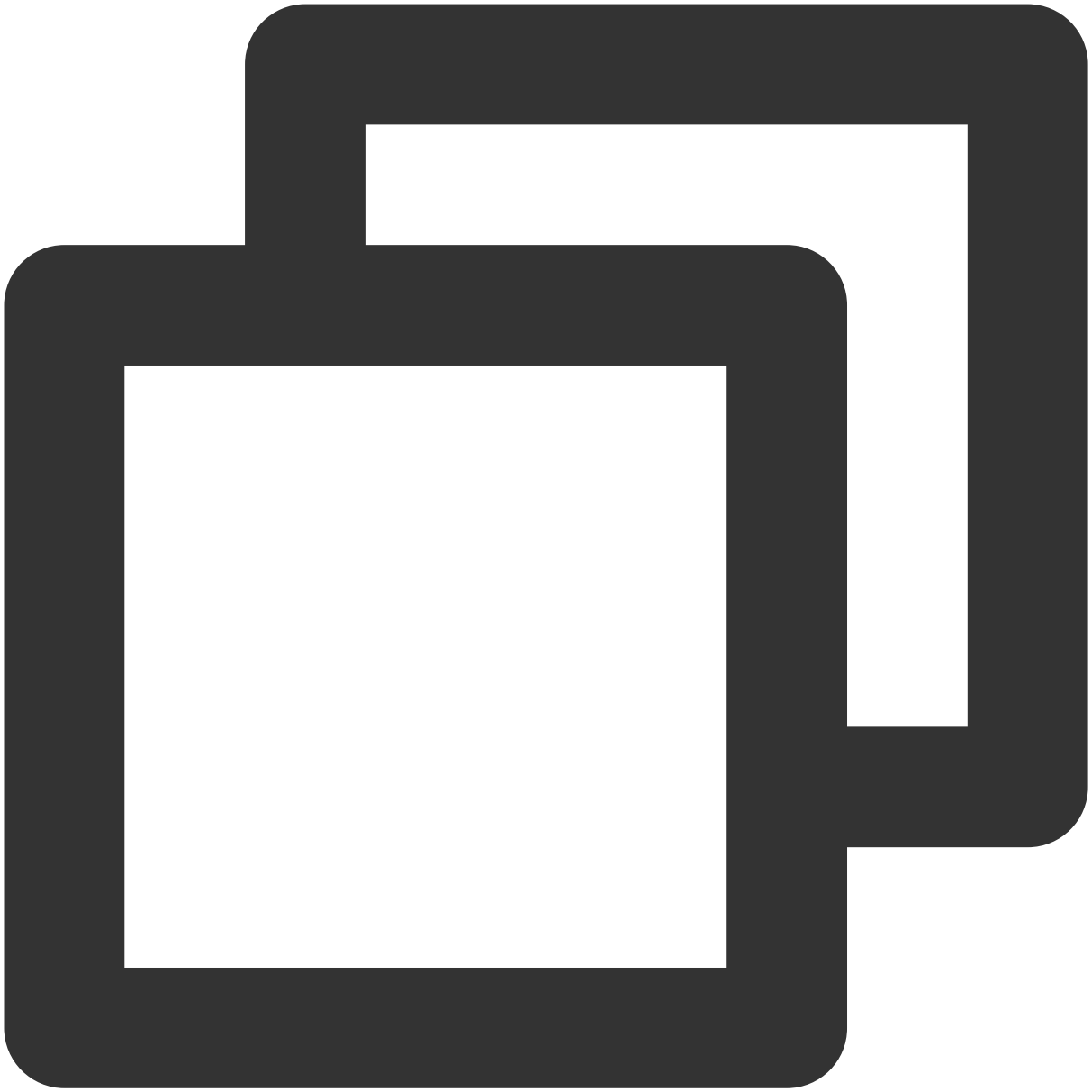
对混流参数进行设置，主要是设置最终混流后生成视频的分辨率和帧率。



```
videoEffectManager.setMixingConfig({
    videoWidth: 1280,
    videoHeight: 720,
    videoFramerate: 15
});
```

步骤4：采集多路流

启用本地混流之后，开始采集多路流，例如先打开摄像头，再进行屏幕分享。注意保存流 ID ，后续操作都需要使用流 ID 。

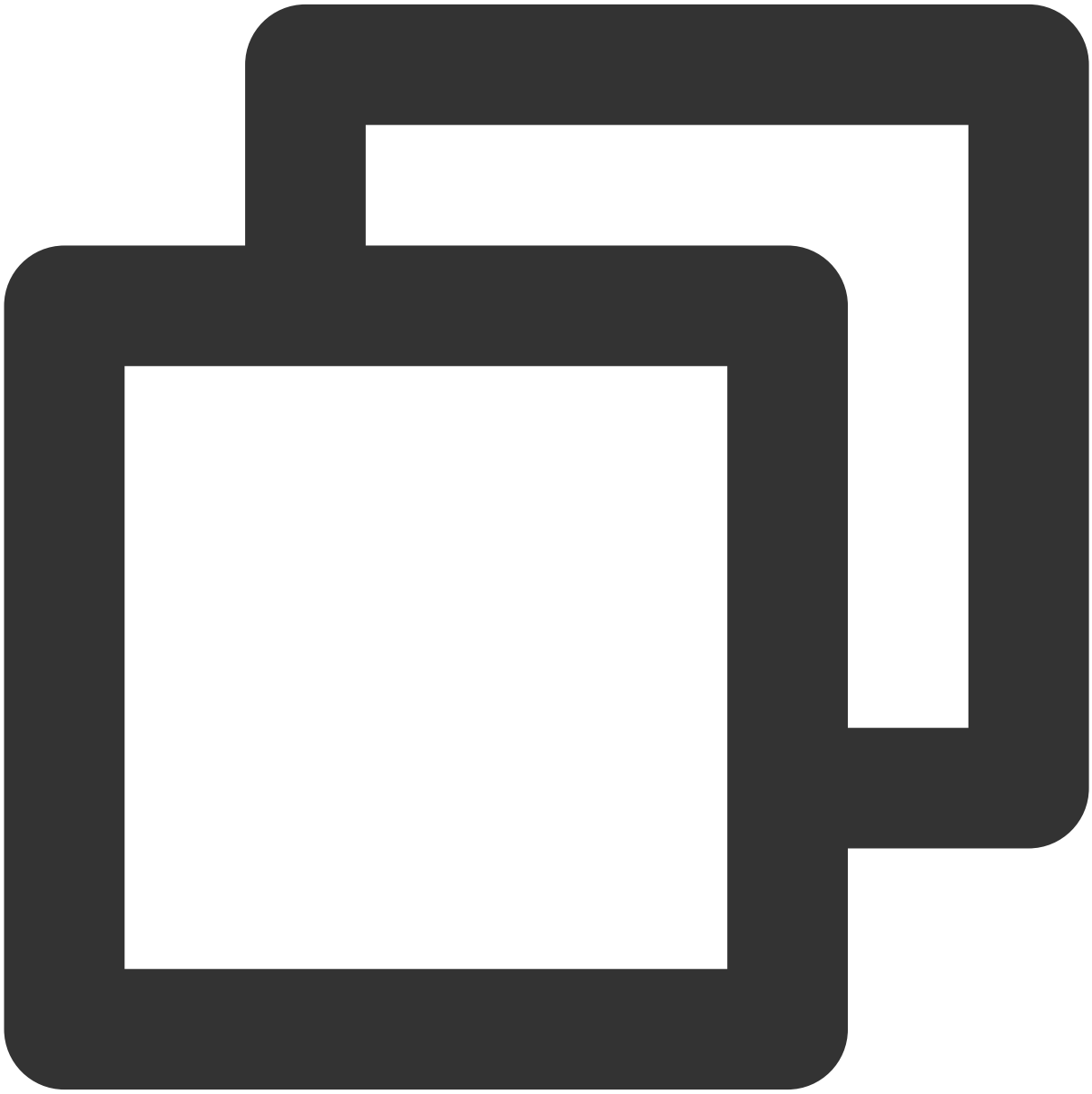


```
var cameraStreamId = null;
var screenStreamId = null;

livePusher.startCamera().then((streamId) => {
    cameraStreamId = streamId;
}).catch((error) => {
    console.log('打开摄像头失败：'+ error.toString());
});

livePusher.startScreenCapture().then((streamId) => {
    screenStreamId = streamId;
```

```
}).catch((error) => {
    console.log('屏幕分享失败：'+ error.toString());
});
```

步骤5：设置画面布局

对采集的两路画面进行布局设置。这里我们主要显示屏幕分享画面，摄像头画面出现在左上角。具体参数配置请参见 TXLayoutConfig 。

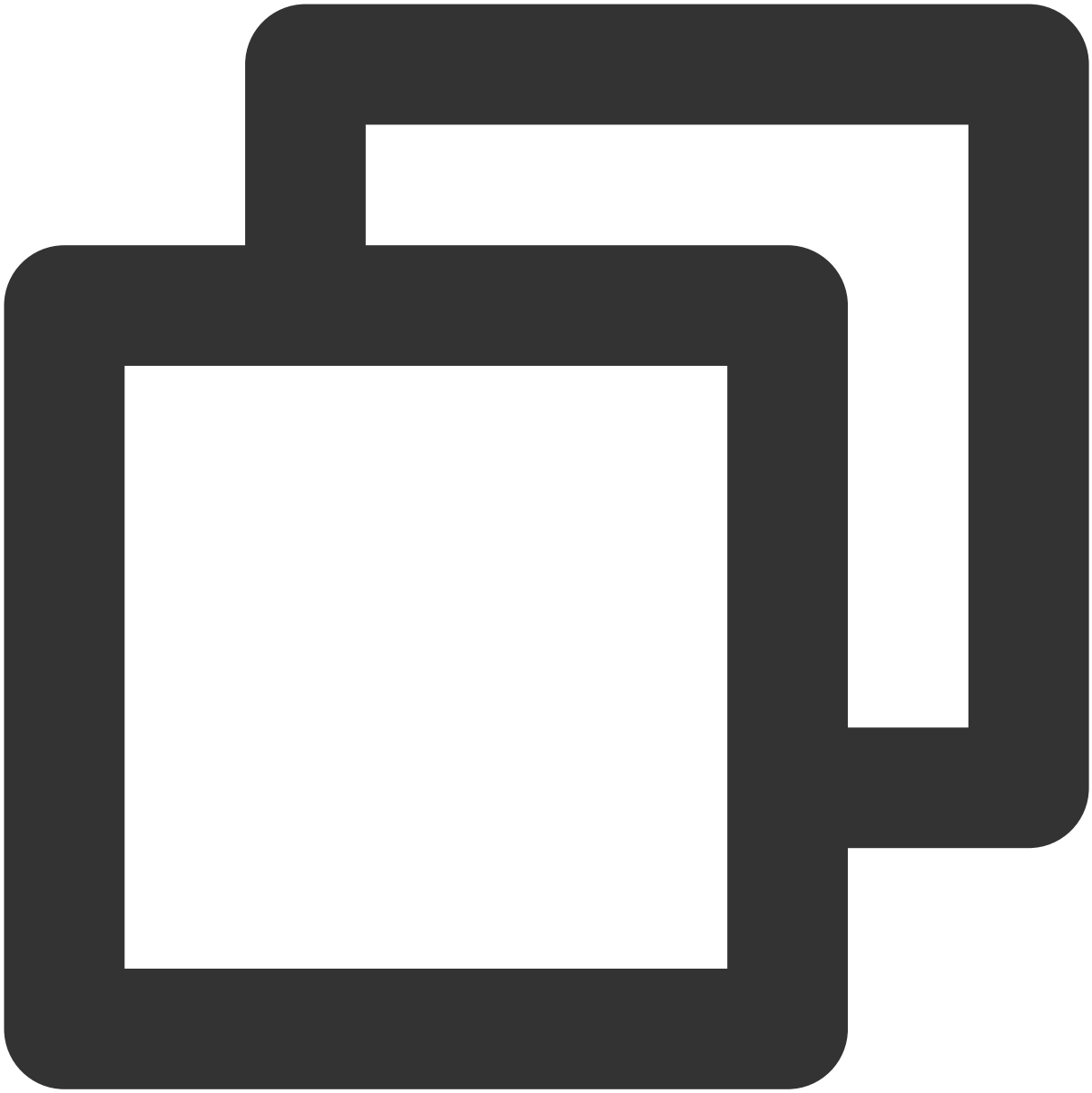


```
videoEffectManager.setLayout([{
    streamId: screenStreamId,
```

```
    x: 640,
    y: 360,
    width: 1280,
    height: 720,
    zOrder: 1
}, {
    streamId: cameraStreamId,
    x: 160,
    y: 90,
    width: 320,
    height: 180,
    zOrder: 2
}]);
```

步骤6：设置镜像效果

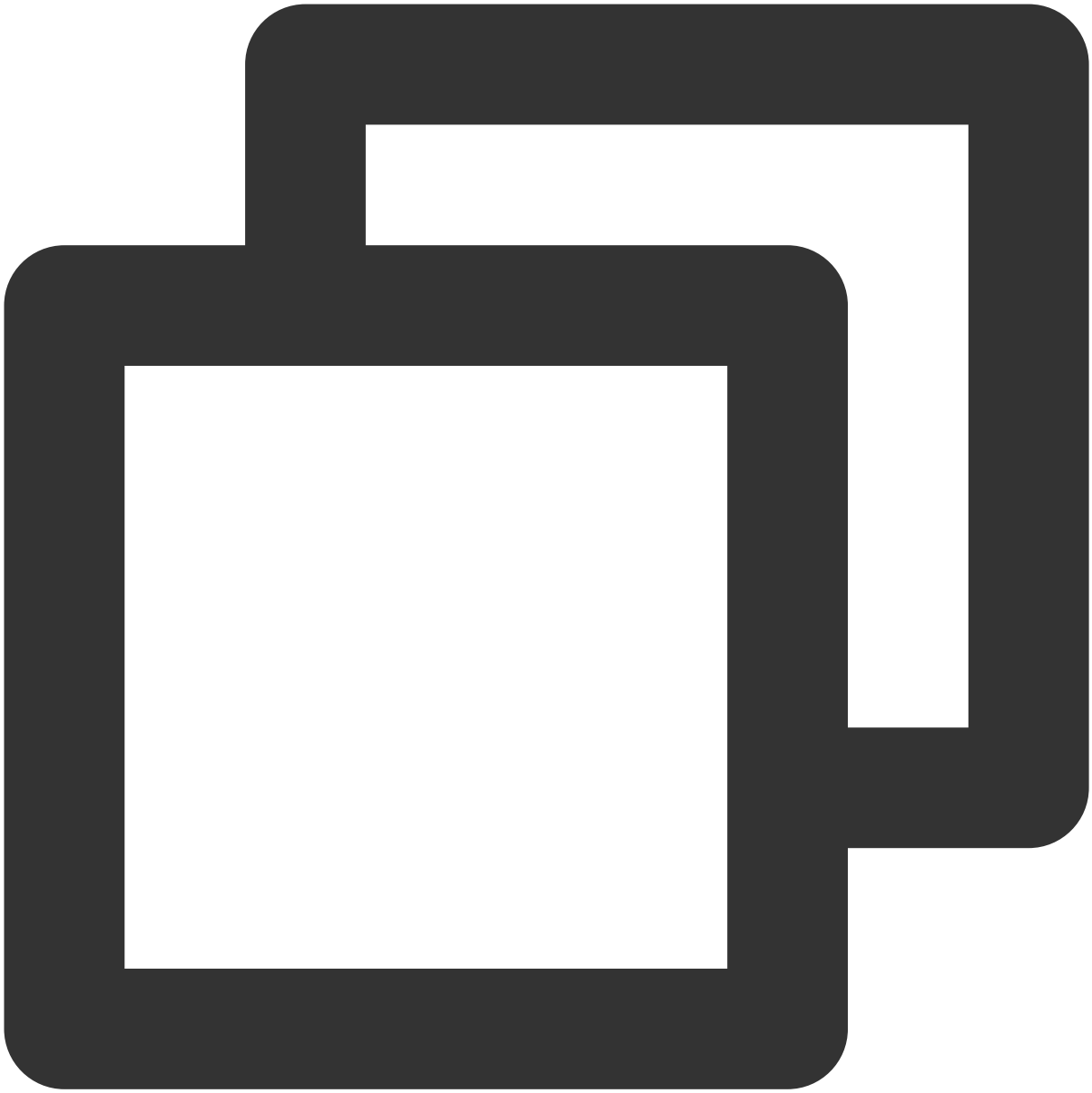
摄像头采集到的画面实际上是反的，这里对摄像头画面进行一次左右翻转。



```
videoEffectManager.setMirror({
    streamId: cameraStreamId,
    mirrorType: 1
});
```

步骤7：添加水印

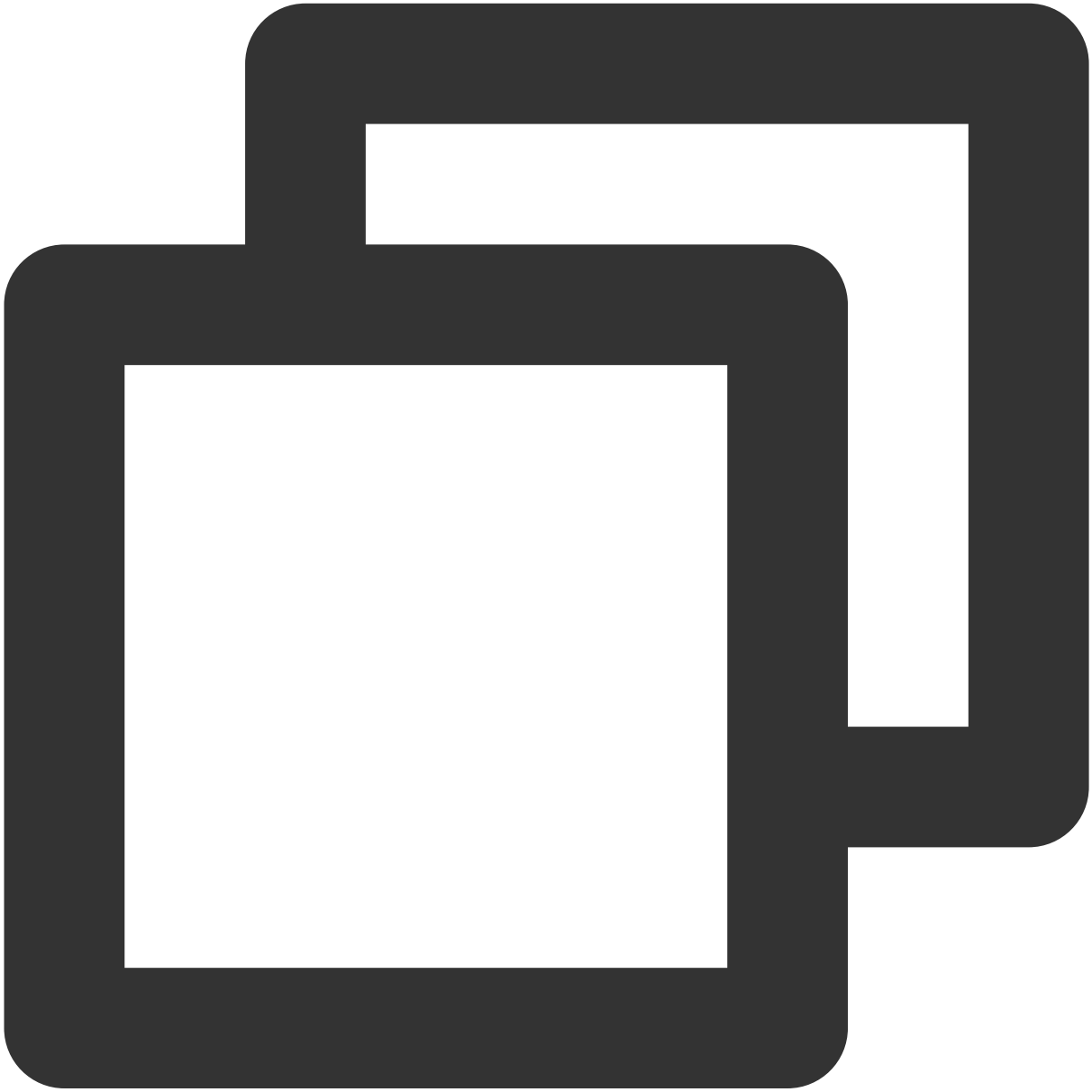
先准备好一个图片对象，然后将这个图片对象作为水印添加到视频流画面中，这里把水印图片放置在右上角。



```
var image = new Image();
image.src = './xxx.png'; // 图片地址注意不要跨域，否则会有跨域问题

videoEffectManager.setWatermark({
    image: image,
    x: 1230,
    y: 50,
    width: 100,
    height: 100,
    zOrder: 3
});
```

步骤8：开始推流

上面操作都完成后，我们最终得到了一个由画中画布局、镜像效果和水印组成的视频流，然后把处理之后的视频流推送到服务器。



```
livePusher.startPush('webrtc://domain/AppName/StreamName?txSecret=xxx&txTime=xxx');
```

说明：

WebRTC 混流相关接口说明，请参见 API 概览。

版权所有：腾讯云计算（北京）有限责任公司

弹幕及会话聊天集成指引

最近更新时间：2022-11-21 16:56:28

概述

在直播业务中，往往存在主播与观众间实时交互的场景需求如弹幕、会话聊天等，接入往往比较复杂，本文以腾讯云即时通信 IM为基础，梳理了在直播过程中弹幕、会话聊天、商品推送等需求的实现方案，以及可能遇到的问题、需要注意的细节点等，帮助开发者们快速的理解业务，实现需求。

重点功能介绍

功能	说明
直播弹幕、送礼和点赞	支持亿级消息并发，轻松打造良好的直播聊天互动体验。
直播单聊、群聊等多种聊天模式	用户可以在这里发送自己想说的话并且可以收到同一个聊天室中其他成员的消息。支持文字、图片、语音、短视频等多种消息类型，实时消息推送，有效提升用户粘性与活跃度。

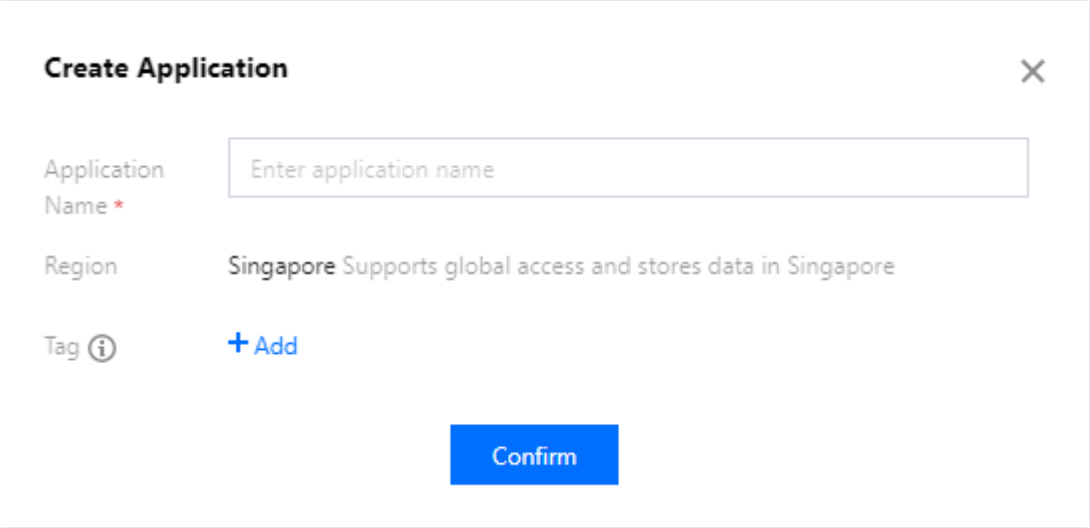
版权所有：腾讯云计算（北京）有限责任公司

功能	说明
直播带货商品推送	在直播带货场景中，主播推荐某款产品时，屏幕下方的商品位需要立即更新为当前产品，并需要通知所有在直播中的用户有新的商品上线，商品上新消息一般由小助手触发。
直播间大喇叭	相当于一个直播间维度的系统公告功能，当系统管理员下发大喇叭消息时，SDKAppID 下所有直播间均可收到该大喇叭消息。

接入方法

步骤1：创建应用

使用腾讯云搭建直播间首先要在 控制台 创建一个即时通信 IM 应用，如下图所示：

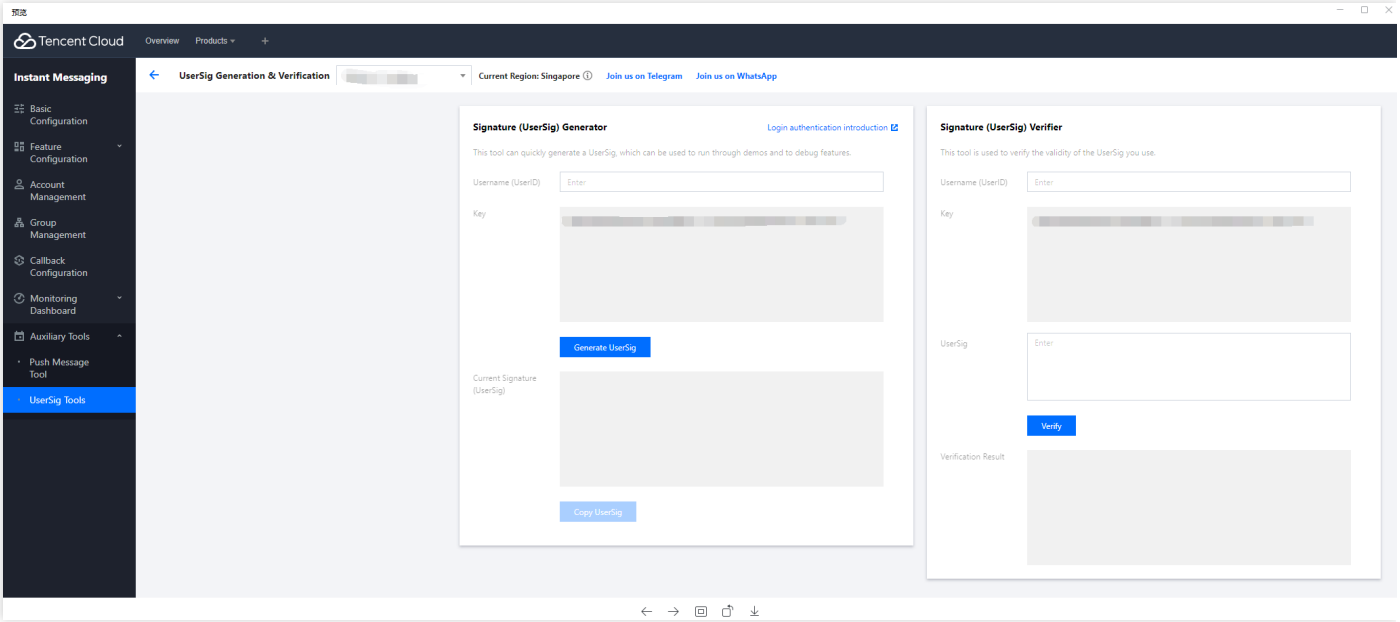


步骤2：完成相关配置

准备工作步骤一中创建的应用为体验版，只适合在开发阶段使用，正式环境用户可结合自己业务需求，开通专业版或旗舰版。不同版本之间的差异可参见 即时通信 IM 价格文档。
在直播场景中，除了创建应用之外，还需要一些额外的配置：

- **使用密钥计算 UserSig**
  在 IM 的账号体系中，用户登录需要的密码由用户服务端使用 IM 提供的密钥计算，用户可参见 UserSig计算 文

档，在开发阶段，为了不阻塞客户端开发，也可在 控制台计算 UserSig ，如下图所示：



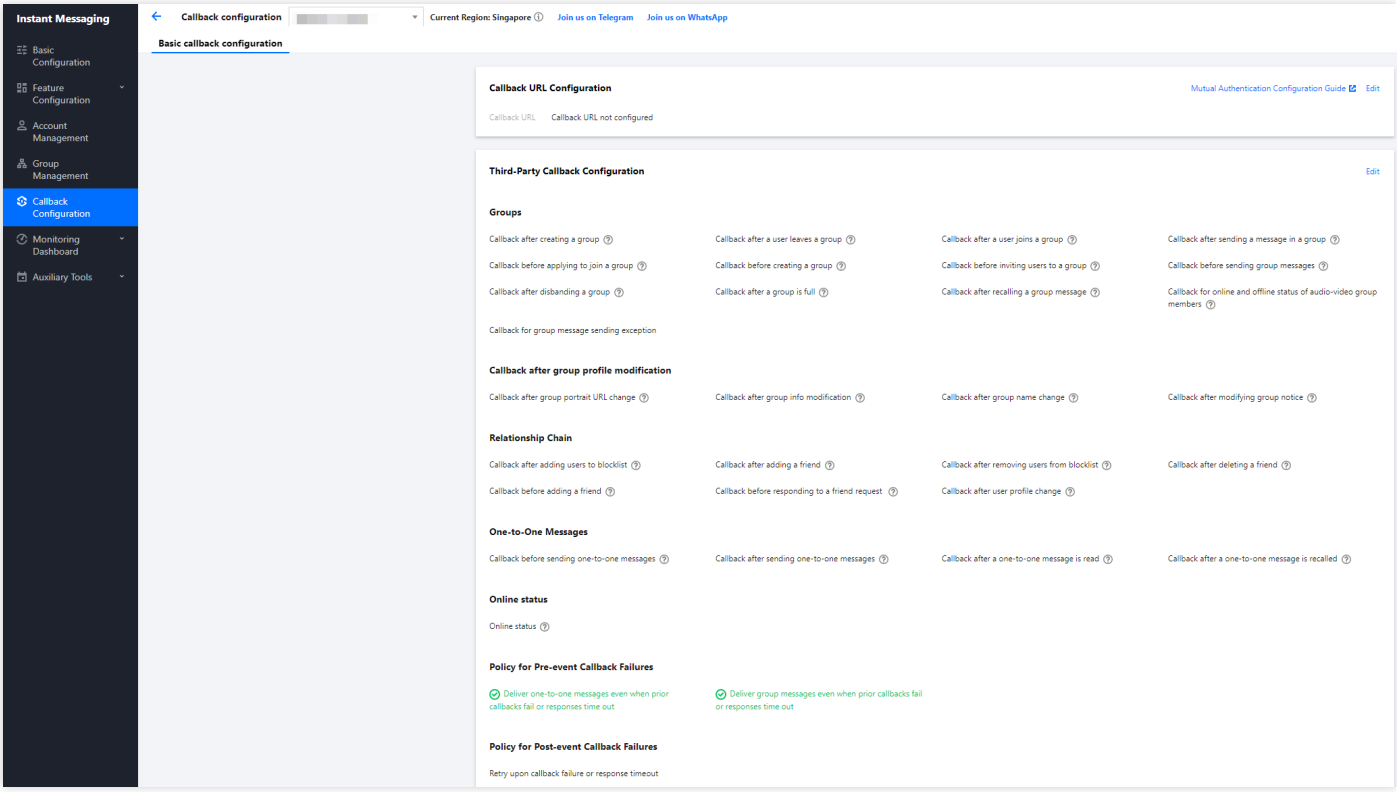
- **配置管理员账号**

  在直播过程中，可能需要管理员向直播间发送消息、禁言（踢出）违规用户等，这时就需要使用 即时通信 IM 服务端 API 来进行相应的处理，调用服务端api前需要 创建 IM 管理员账号，IM 默认提供一个 UserID 为 administrator 的账号供开发者使用，开发者也可以根据业务的场景，创建多个管理员账号。需要注意的是，IM 最多创建五个管理员账号。

- **配置回调地址以及开通回调**

  在实现直播间弹幕抽奖、消息统计、敏感内容检测等需求时，需要用到 IM 的回调模块，即 IM 后台在某些特定的场景回调开发者业务后台。开发者只需要提供一个 HTTP 的接口并且配置在 控制台 > 回调配置 模块即可，如下

图所示：



步骤3：集成客户端 SDK

在准备工作都完成好后，需要将即时通信 IM 以及实时音视频 TRTC 的客户端 SDK 集成到用户项目中去。开发者可以根据自己业务需要，选择不同的集成方案。请参见 快速集成系列文档。

接下来文章梳理了直播间中常见的功能点，提供最佳实践方案供开发者参见，并附上相关实现代码。

步骤4：直播间重要功能开发指引

1. **选择群类型**

   在直播这个场景，用户聊天区域有以下特点：

   - 用户进出群频繁，且不需要管理该群会话信息（未读、lastMessage等）
   - 用户自动进群不需要审核
   - 用户临时发言，不关注群历史聊天记录
   - 群人数通常比较多
   - 可以不用存储群成员信息

所以根据 IM 的 群特性，这里选择 AVChatRoom 作为直播间的群类型。即时通信 IM 的直播群（AVChatRoom）有以下特点：

- **无人数限制，可实现千万级的互动直播场景**。
- 支持向全体在线用户推送消息（群系统通知）。

- 申请加群后，无需管理员审批，直接加入。

> 说明：
> IM Web 限制同一用户在同一时间内，只能进入一个 AVChatRoom，在 IM 的多端登录场景，如果用户登录终端一在直播间 A 观看直播，在 控制台配置 允许多端登录情况下，该用户登录终端二进入直播间 B 观看，这时终端一的直播间 A 会被退群。

2. **设置直播间弹幕、送礼、点赞**

- **弹幕**
  AVChatRoom 支持弹幕、 送礼和点赞等多消息类型，轻松打造良好的直播聊天互动体验。
  弹幕消息的构建，可以通过使用IM的API创建文本或自定义消息，在发送成功后根据您需要在直播间展示弹幕的方式，通过接收 新消息的回调，获取直播间新消息的文本或者自定义等属性，之后使用不同的UI进行上屏展示。
- **送礼**
  - 客户端短连接请求到自己的业务服务器，涉及到计费逻辑。
  - 计费后，发送人直接看到 XXX 送了 XXX 礼物。（以确保发送人自己看到自己发的礼物，消息量大的时候，可能会触发抛弃策略）
  - 计费结算后，调用服务端接口发送发送自定义消息（礼物）。
  - 如果遇到连刷礼物的场景需要进行消息合并。
    - 如果直接选择礼物数量的刷礼物，如：直接选择 99 个礼物，1 条消息发送，参数带入礼物数量 99。
      - 如果连击的礼物，不确定停留在多少个，可以合并每 20 个（数量自己调整）或者连击超过 1 秒，发送一个。按照上述逻辑，例如连击 99 个礼物，优化后仅需要发送 5 条。
- **点赞**
  - 点赞与礼物略不同，点赞往往不用计费，所以一般采用端上直接发送的方式。
  - 针对与有需要服务端计数的点赞消息，在客户端节流之后，统计客户端点赞次数，将短时间内多次点赞消息合并之后，仅发送一次消息即可。业务方服务端在发消息前回调中拿到点赞次数进行统计。
  - 针对与不需要计数的点赞消息，与步骤2中的逻辑一致，业务要在客户端对点赞消息节流后发送，不需要在发消息前回调中计数。

3. **直播带货商品推送**
   主播推荐某款产品时，屏幕下方的商品位需要立即更新为当前产品，并需要通知所有在直播中的用户有新的商品上线，商品上新消息一般由小助手触发。推荐采用管理员修改群自定义字段的方式实现商品上新消息通知，具体步骤如下：

   i. **添加群自定义字段**

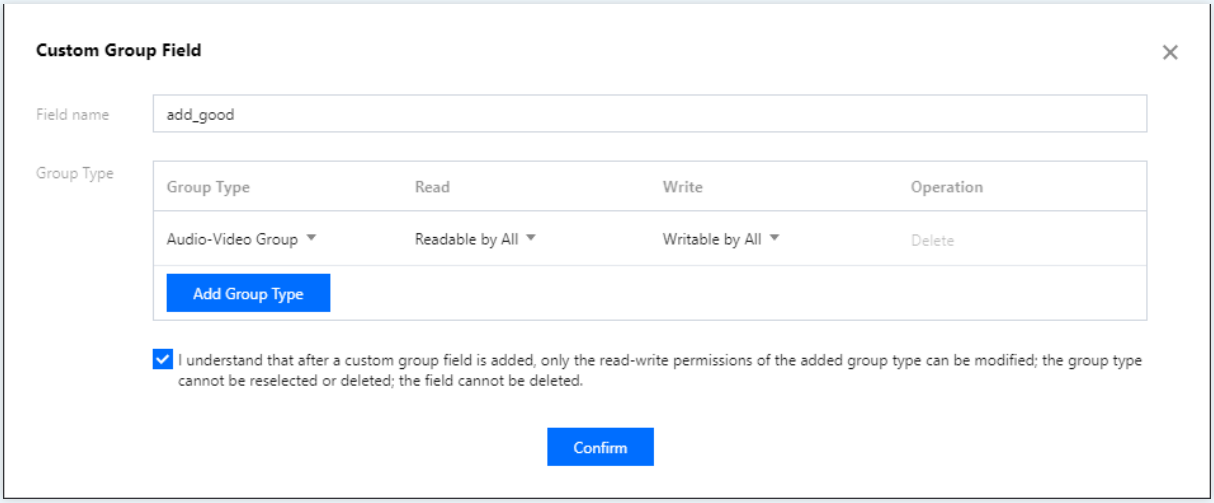
a. 登录 即时通信 IM 控制台，单击目标应用卡片，在左侧导航栏选择**功能配置** > **群组配置**。

b. 在**群自定义字段**页面，单击**添加群维度自定义字段**。

c. 在弹出的群维度自定义字段对话框中，输入字段名称，设置群组类型及其对应的读写权限。

说明：

- 字段名称只能由字母、数字以及下划线（_）组成，不能以数字开头，且长度不能超过16个字符。
- 群自定义字段名称不允许与群成员自定义字段名称一致。



d. 勾选**我已知道"群维度自定义字段"添加后，仅可修改已添加群组类型的读写权限；无法删除该字段，无法重新选择也无法删除已添加的群组类型。**单击**确定**保存设置。

说明：
群维度自定义字段配置后大约10分钟左右生效。

ii. **使用群自定义字段**

小助手以群管理员的角色身份，适时调用 修改群组基础资料 REST API 更新对应的群自定义字段，从而实现直播间的商品上新通知和直播状态改变消息通知。

4. **直播间大喇叭**

大喇叭功能相当于一个直播间维度的系统公告功能，但有别于公告，大喇叭功能属于消息。当系统管理员下发大

喇叭消息时，SDKAppID 下所有直播间均可收到该大喇叭消息。
大喇叭功能目前属于旗舰版功能，且需要在控制台开通。业务后台发送大喇叭消息请参见 直播群广播消息。

> 说明：
> 若开发者版本非旗舰版，可在服务端发送群自定义消息进行实现。

相关文档

若您还有其他直播间功能例如：实现直播间用户身份、等级；获取直播间历史消息；展示直播间在线人数等功能，请参见 直播间各功能开发指引。