云直播

SDK 实践

产品文档

腾讯云

文档目录

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：

功能模块		App 端	Web 端	文档指引
推流		iOS & Android	Web 推流	推流接入
播放		iOS & Android	-	播放接入
高级功能	播放 AV1 视频	接入指引		高级功能接入
	上行高质量传输	接入步骤	-	
	播放器支持快直播播放	接入步骤	-	
	弹幕及会话聊天集成指引	接入步骤	-	
	美颜特效	iOS & Android	-	

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：2022-11-21 16:56:28

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：

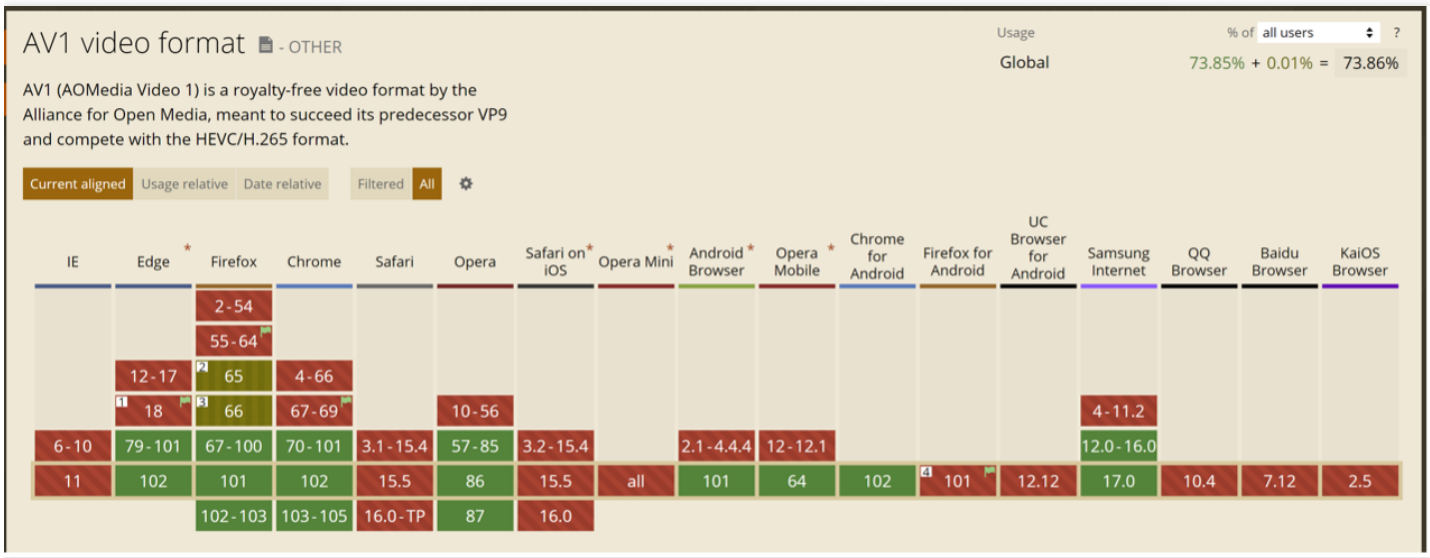
类型	品牌	处理器
手机	realme X7 Pro	天玑1000+
	oppo reno 5 pro	天玑1000+
	荣耀v40	天玑1000+
	Redmi k30 Ultra	天玑1000+
	vivo iQOO Z1	天玑1000+
	Redmi Note 10 Pro	天玑1000+
	vivo S9	天玑1100
	realme Q3 Pro	天玑1100

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。

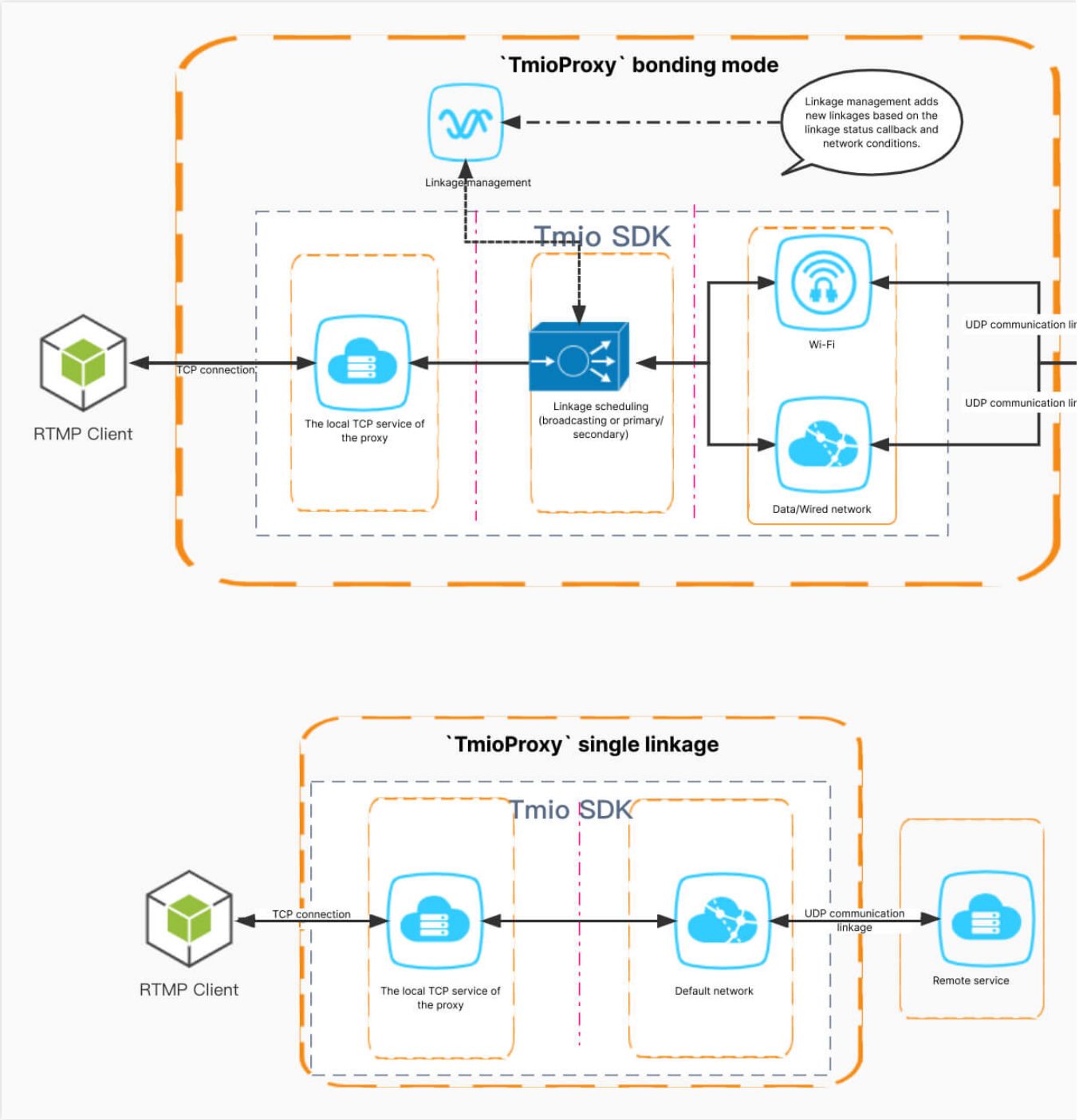
基于 QUIC 的流媒体传输功能
自适应拥塞控制算法。
支持网络连接迁移，平滑无感知。
支持下一代HTTP3基础传输协议。
带宽受限、抖动环境下，发送冗余数据更低，更加节约带宽成本，优势明显。
自研传输控制协议ETC
纯自研，轻量级，跨平台。
支持IoT设备，适合端到端通信。
快启动、低时延且高带宽利用率。
迅速、准确地感知链路状态变化，并及时调整到最佳传输策略。
与主流传输协议并存时，能更公平、更稳定地利用带宽。

接入方法

以 RTMP over SRT 协议为例：

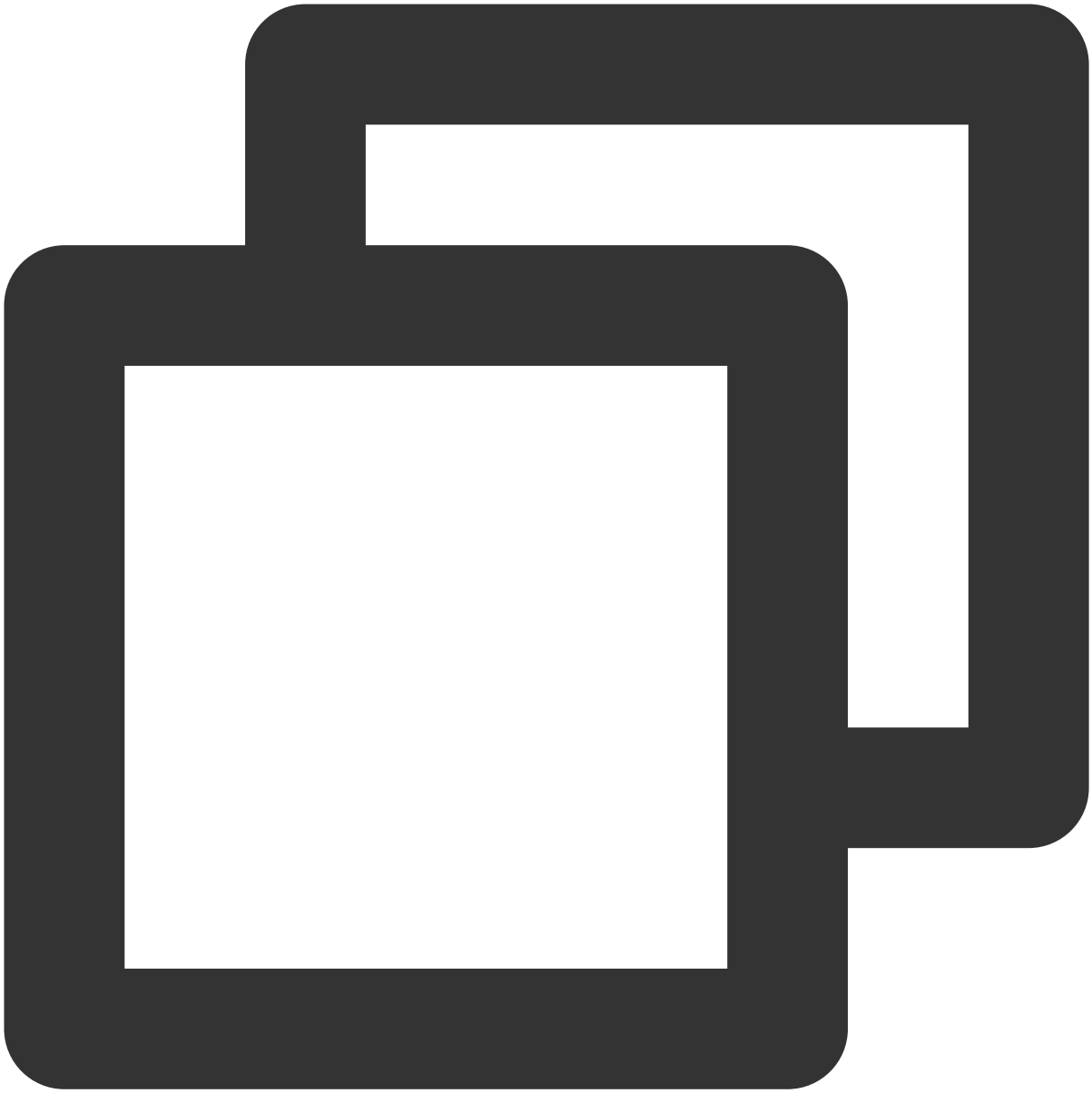
选择代理模式

Tmio Proxy 模式接入方式



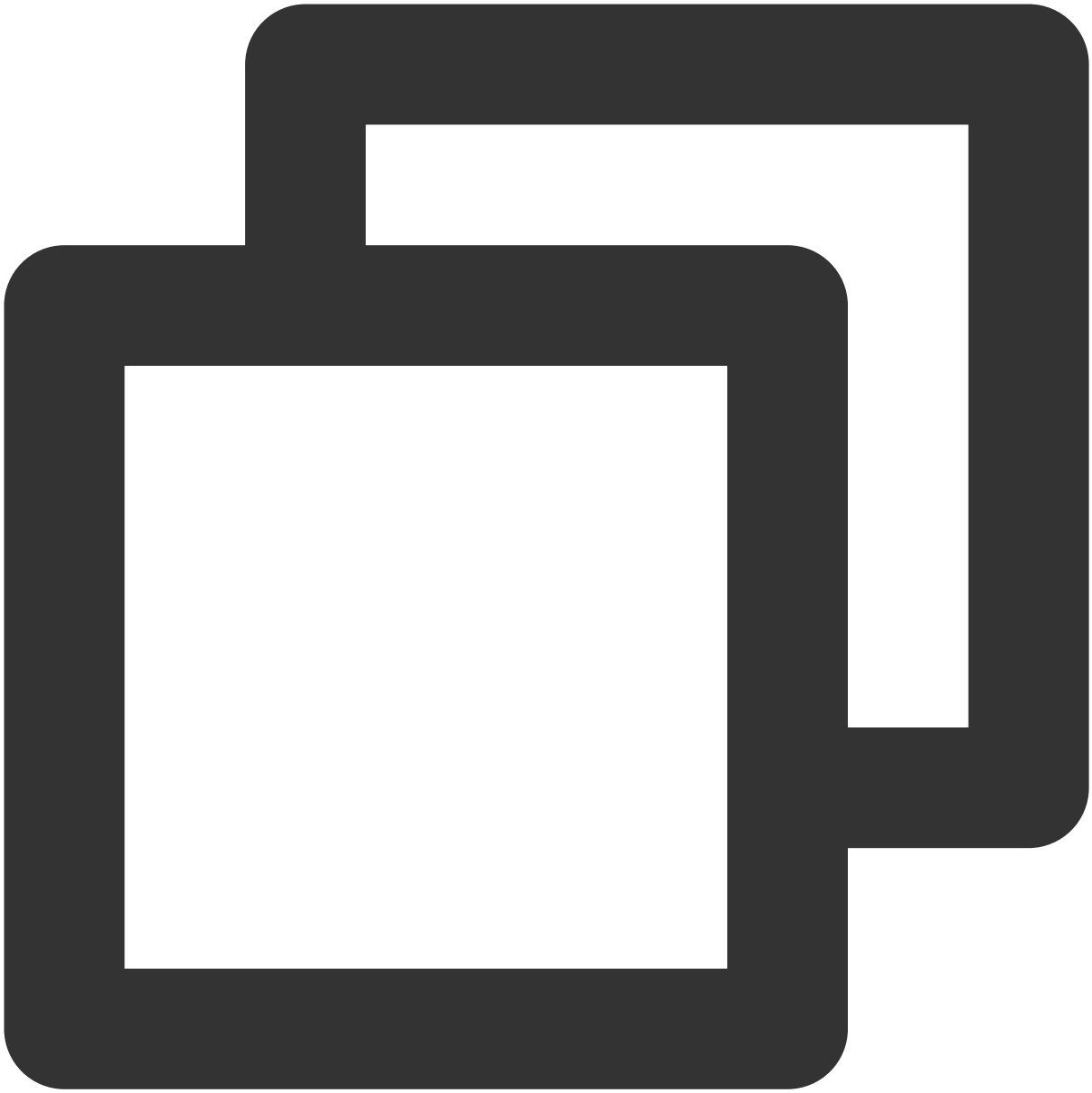
操作步骤

1. **创建 Tmio Proxy**：



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

2. **设置监听**：



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

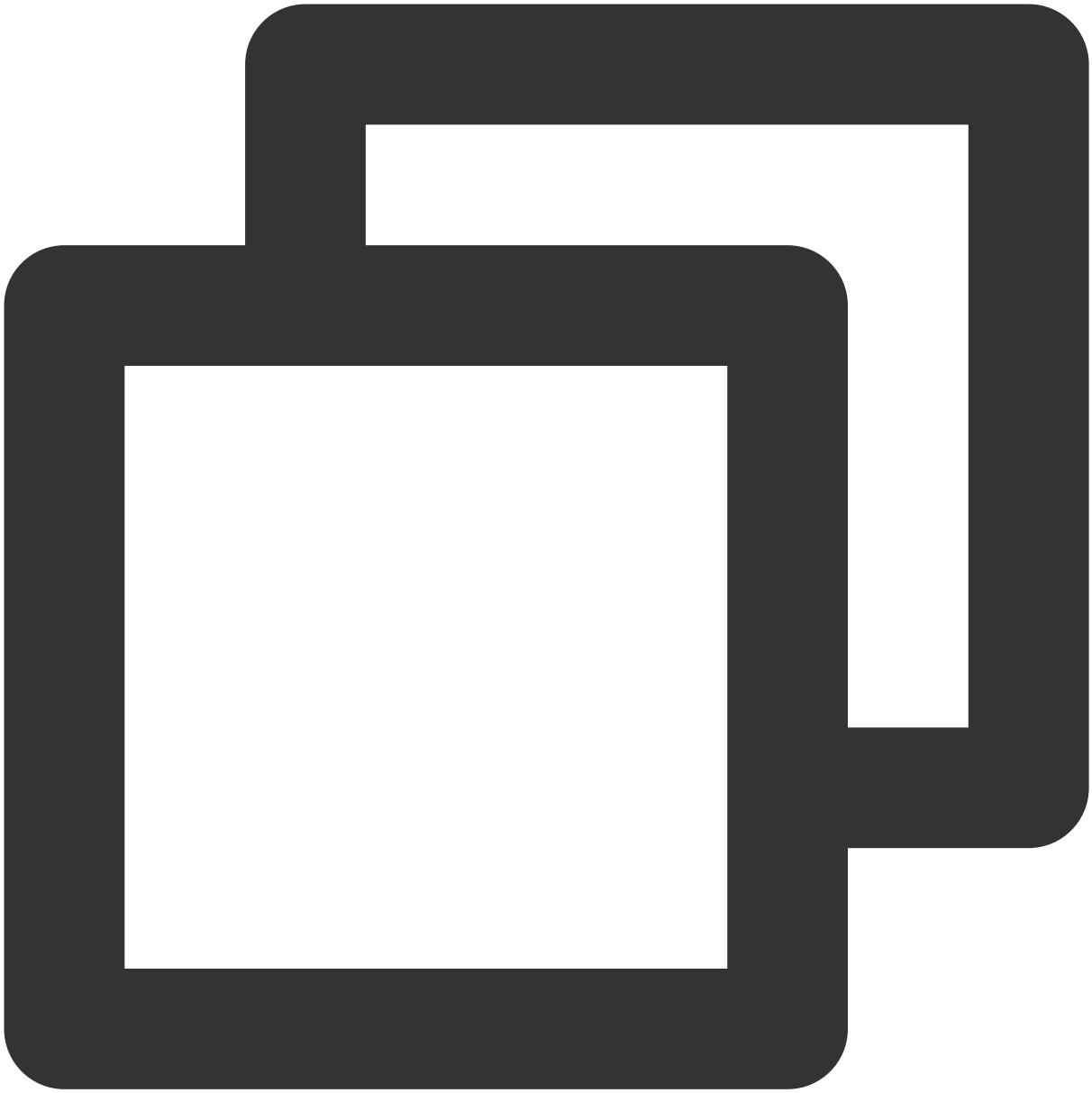
TmioProxyListener 监听接口如下：

Tmio 配置回调

TmioProxy 启动回调

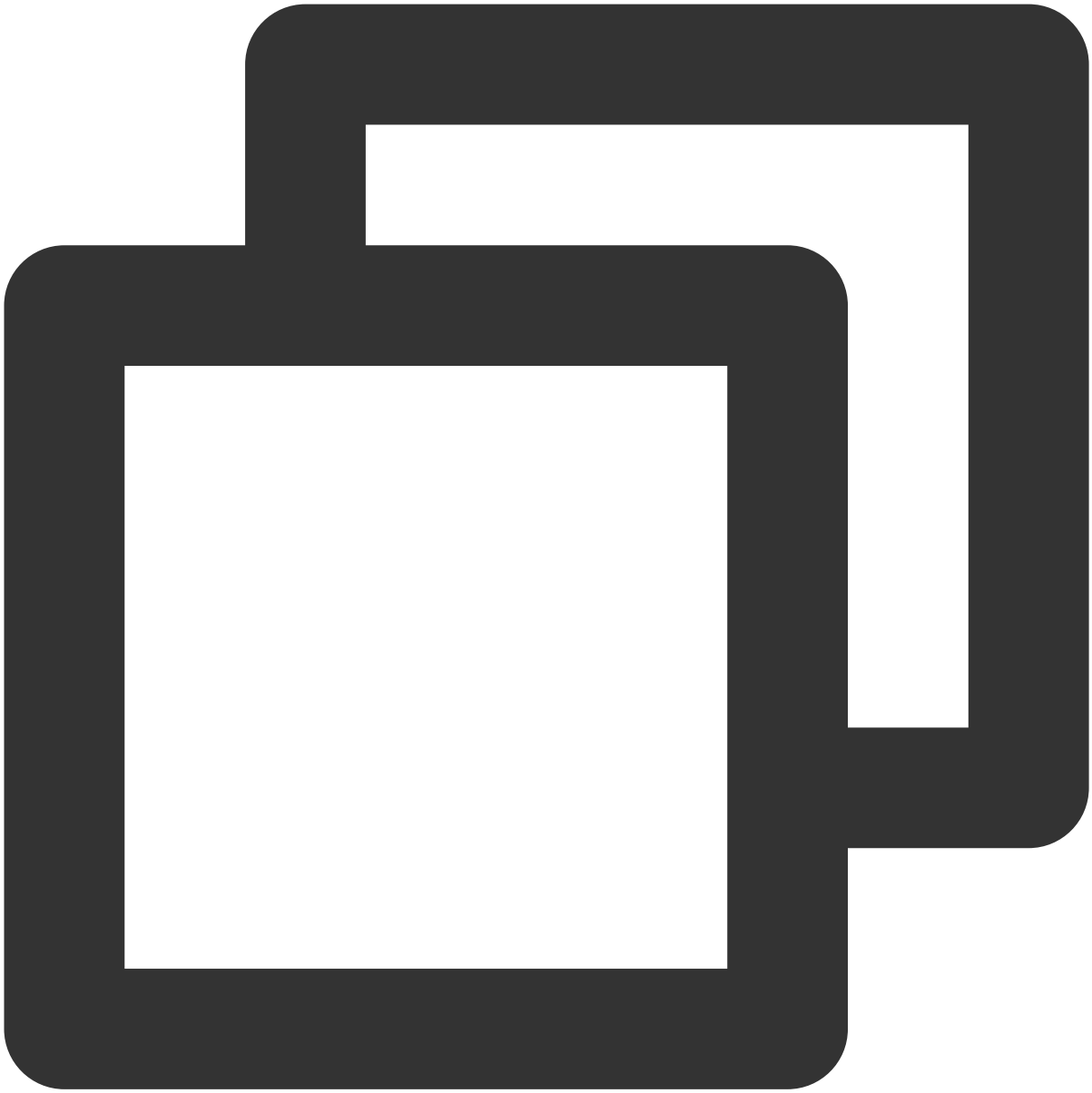
错误信息回调

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**。



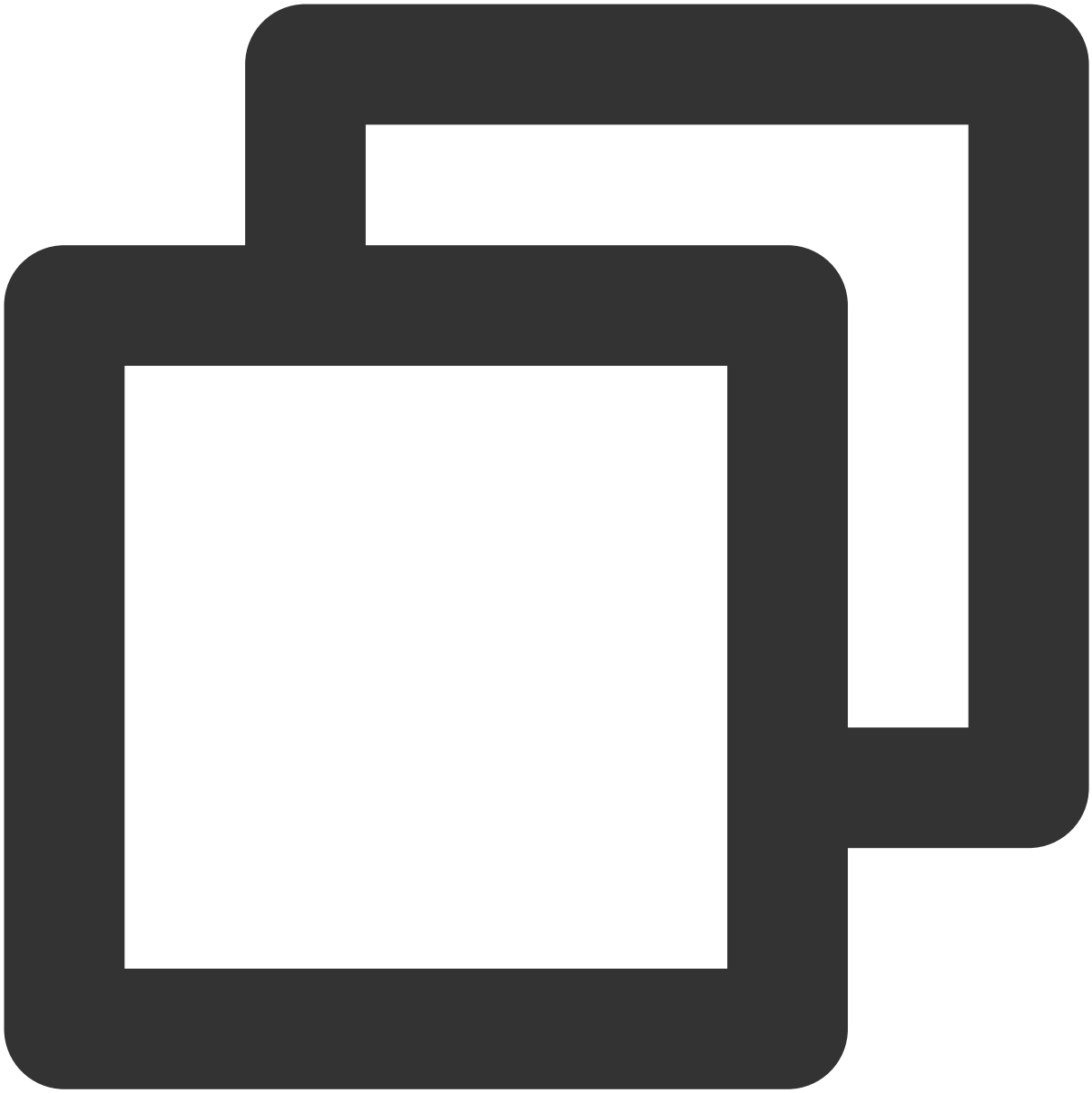
```
/*
void onTmioConfig(Tmio *tmio);
*/
void onTmioConfig(tmio::Tmio *tmio) override {
        auto protocol = tmio->getProtocol();
        if (protocol == tmio::Protocol::SRT) {
                tmio::SrtPreset::rtmp(tmio);
        } else if (protocol == tmio::Protocol::RIST) {
                tmio->setIntOption(tmio::base_options::RECV_SEND_FLAGS,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

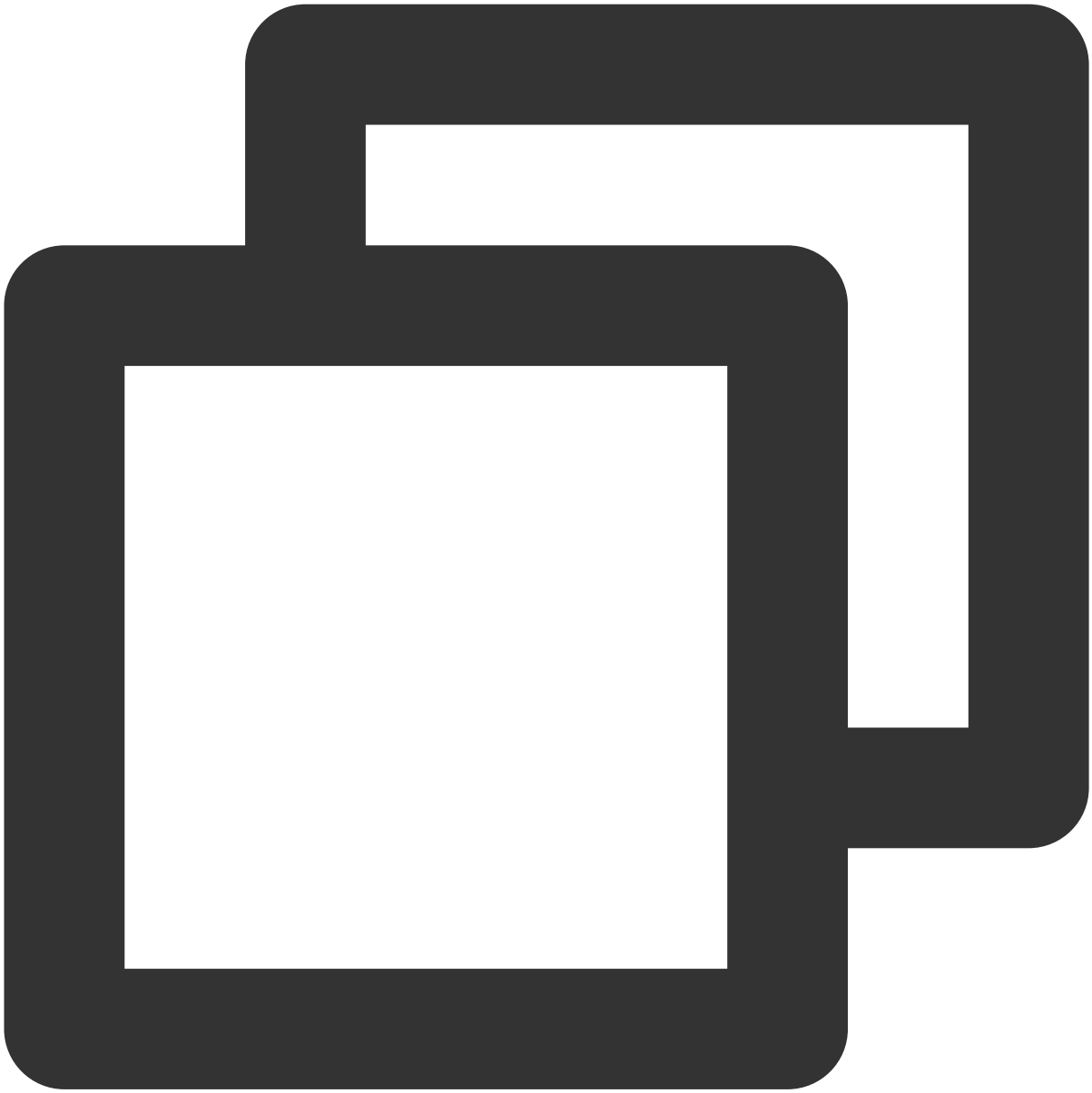
收到此回调代表连接远程服务器成功，并且 TCP 本地端口绑定成功，可以启动推流。



```
/*
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**：



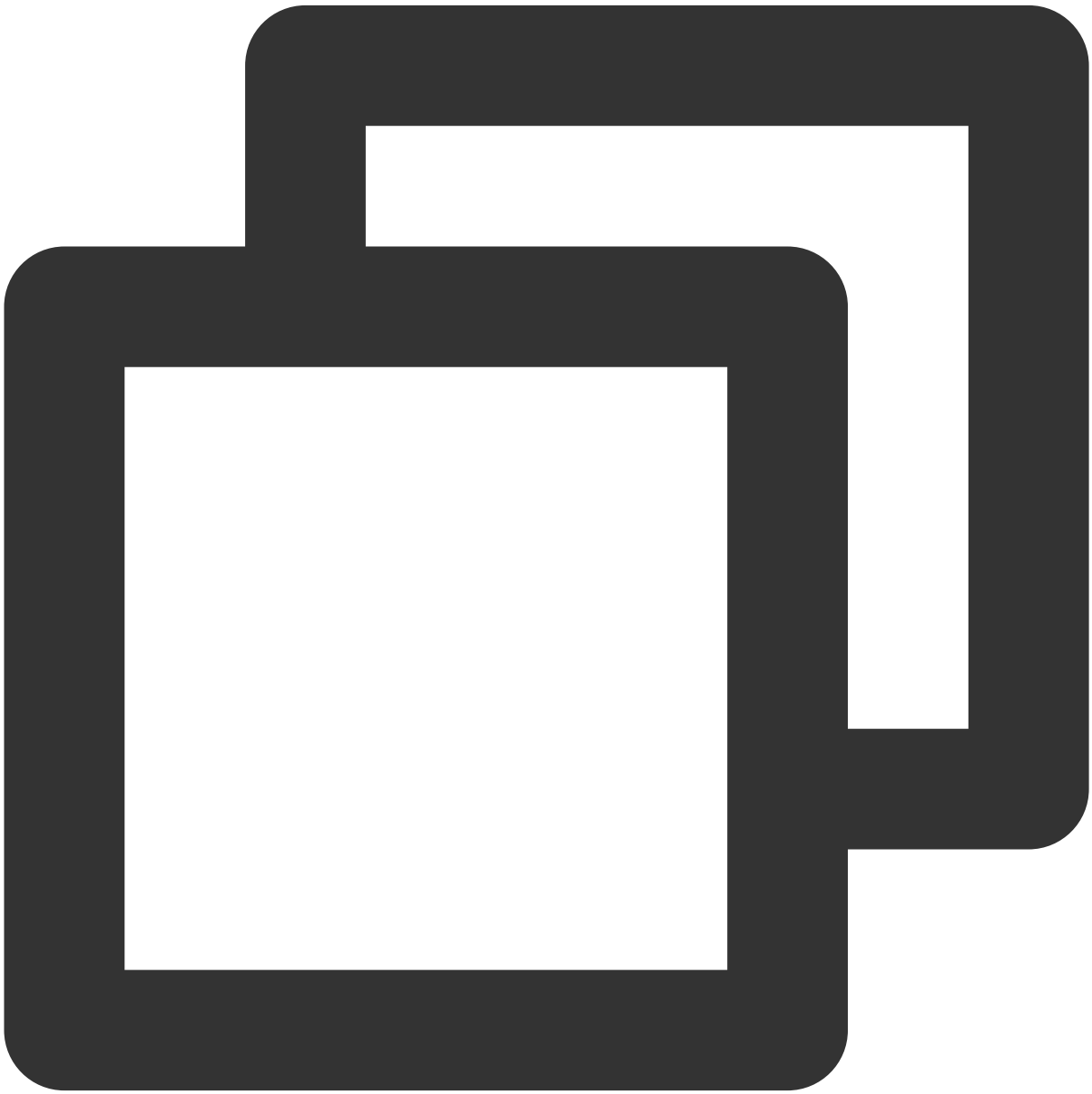
```
std::error_code start(const std::string &local_url, const std::string &remote_url,
```

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

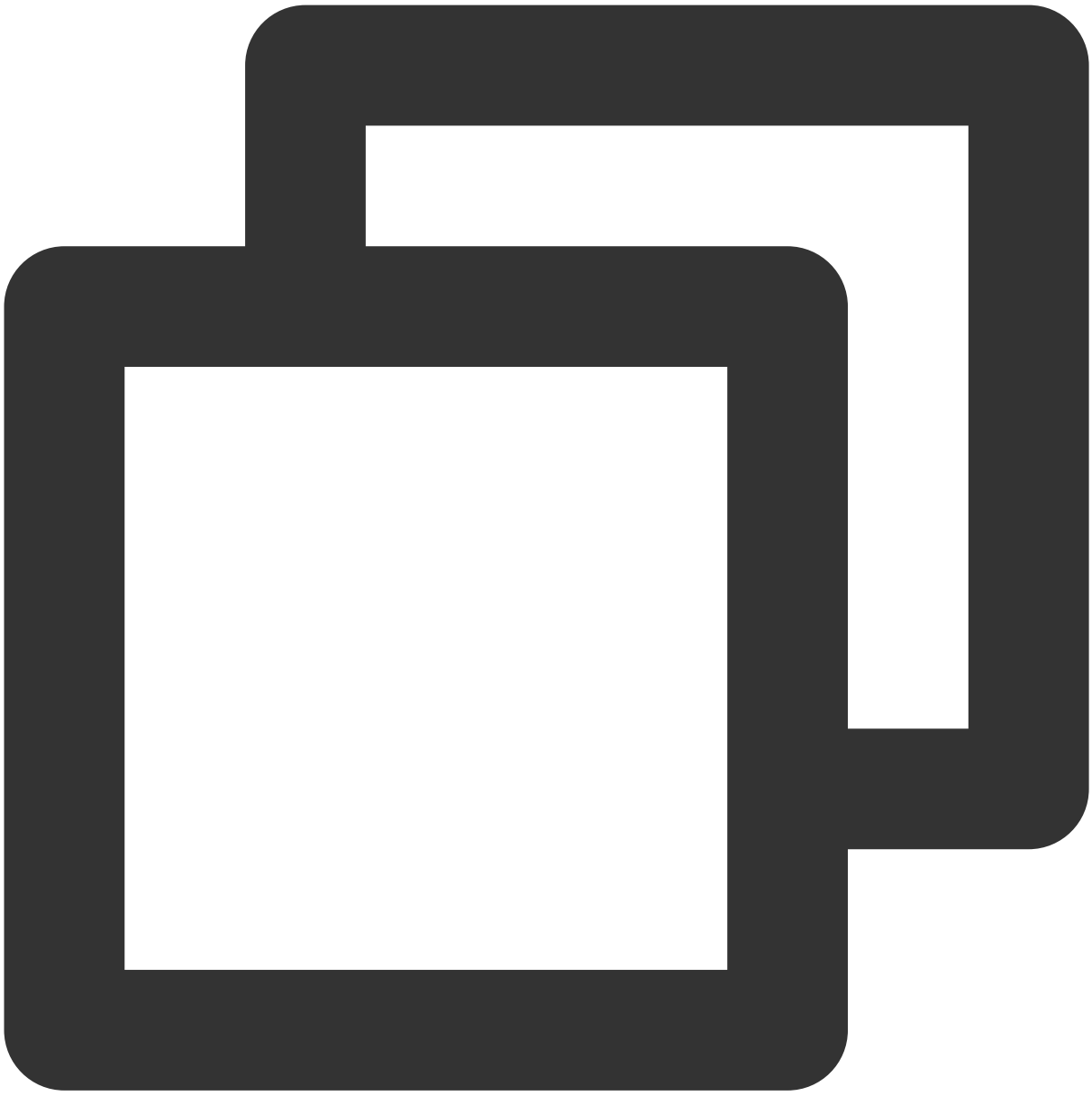
	随机端口，然后通过 onStart() 回调把绑定成功后的端口号返回给应用。使用0端口可以避免端口被占用、无权限等导致的绑定失败问题。
remote_url	远程服务器 URL。
config	配置参数，此参数在 SRT bonding 功能和 QUIC H3 协议启用时使用，具体定义请依据 tmio.h 下 TmioFeatureConfig 结构体定义。

单链路（代码示例）



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

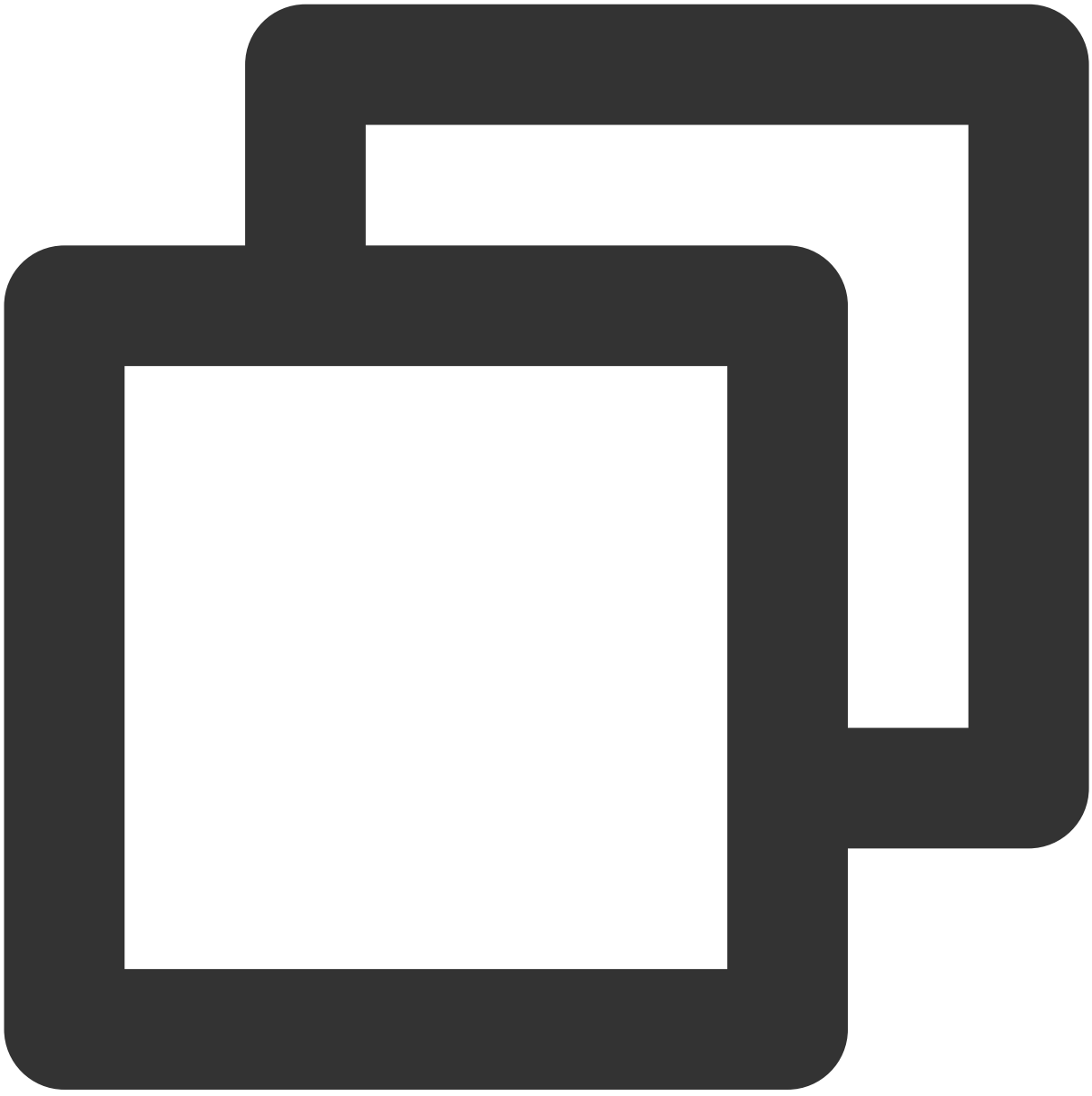
bonding 多链路（代码示例）



```
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_);
```

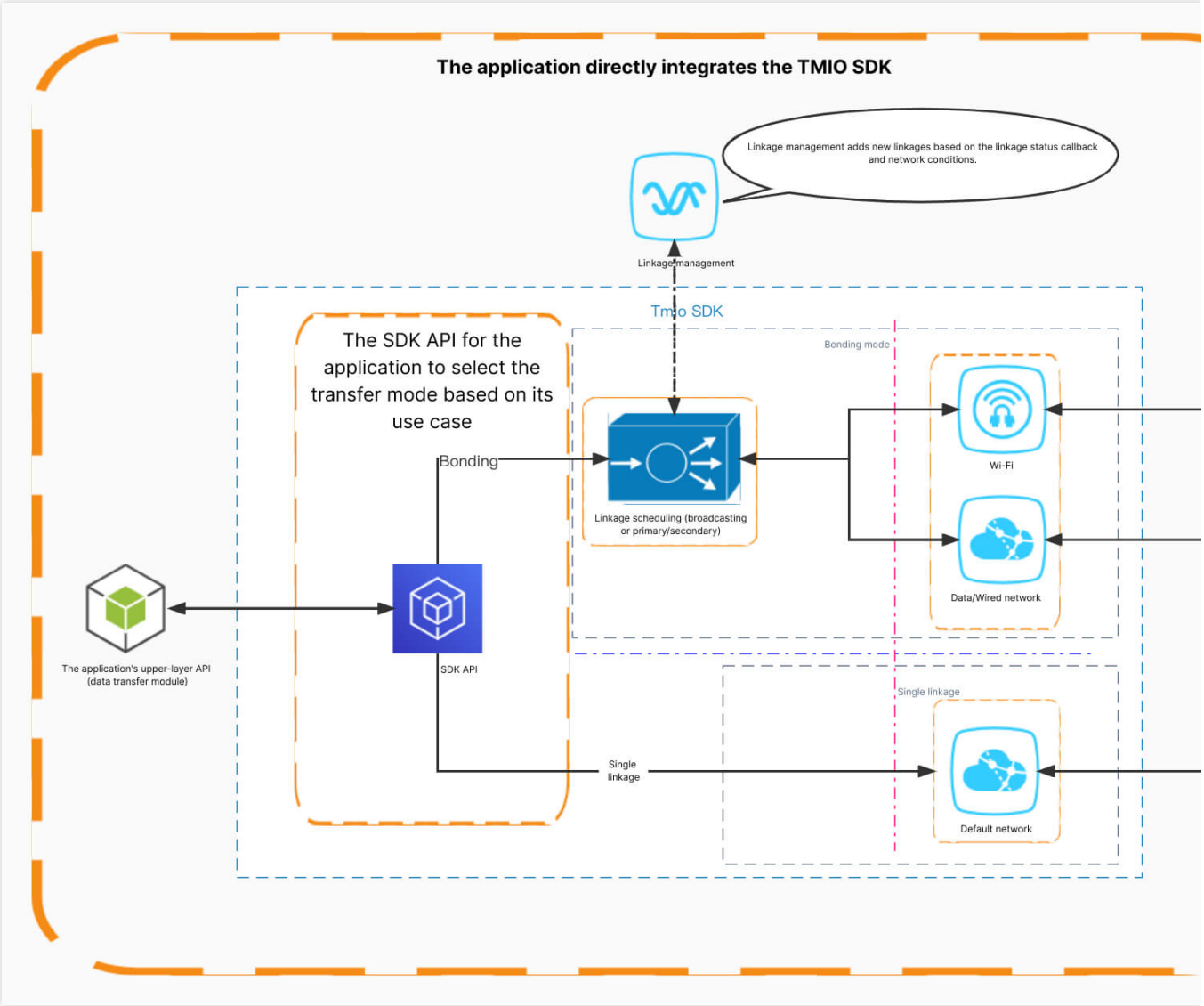
4. **停止**：



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

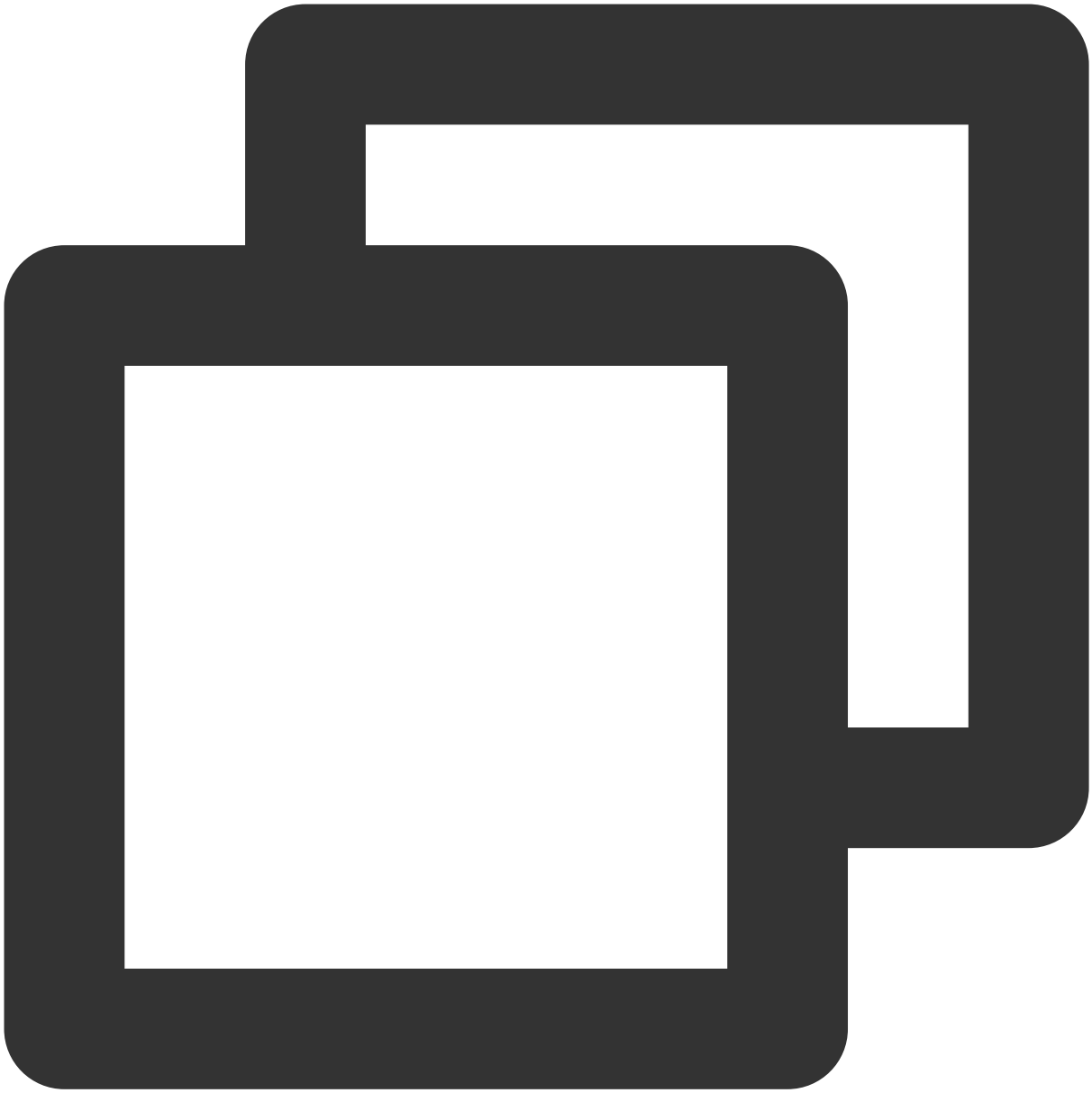
内部集成

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）：



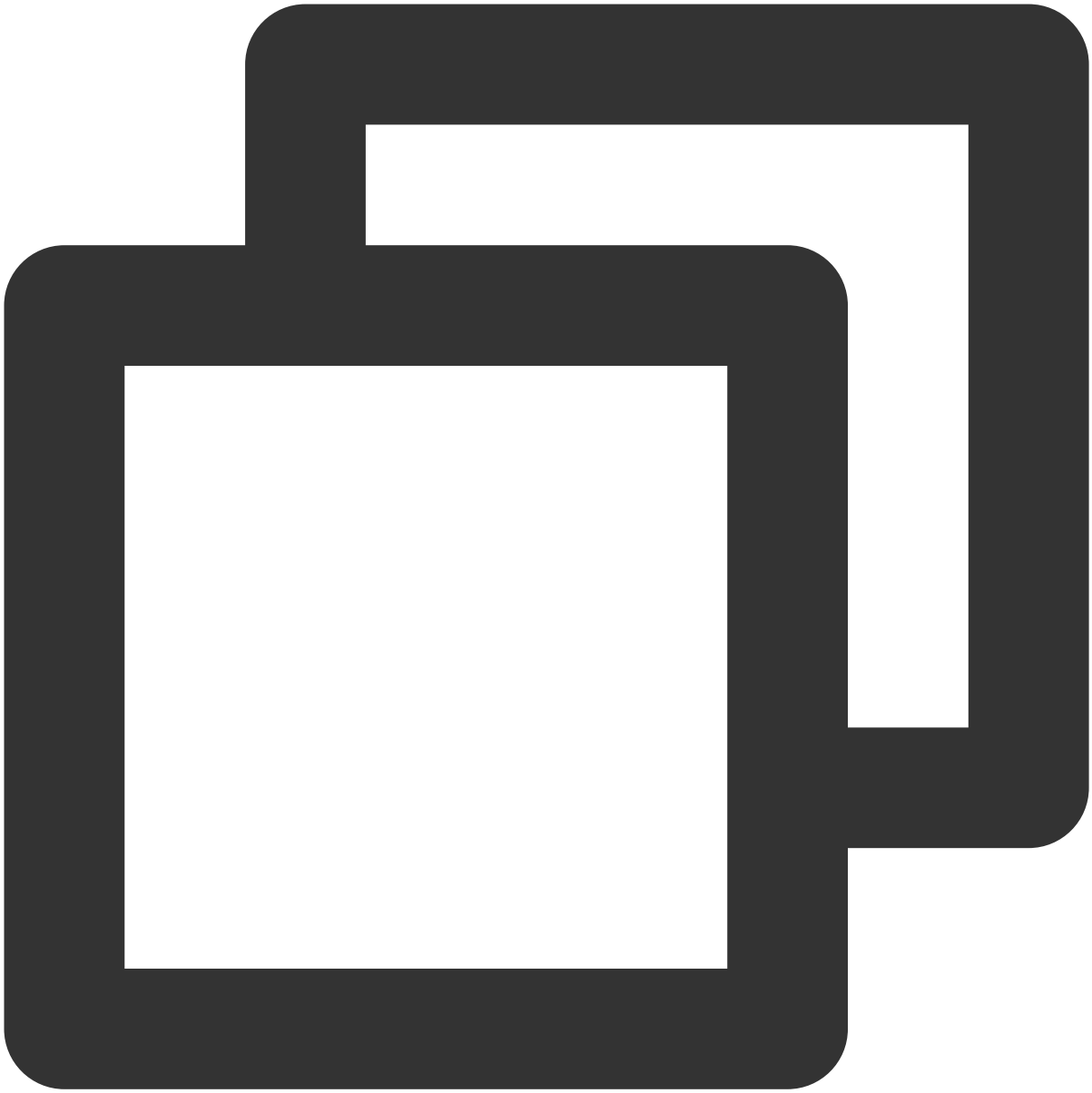
```
tmio_ = tmio::TmioFactory::createUnique(tmio::Protocol::SRT);
tmio::SrtPreset::mpegTsLossless(tmio_.get());
tmio_->setIntOption(tmio::srt_options::CONNECT_TIMEOUT, 4000);
tmio_->setBoolOption(tmio::base_options::THREAD_SAFE_CHECK, true);
```

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：

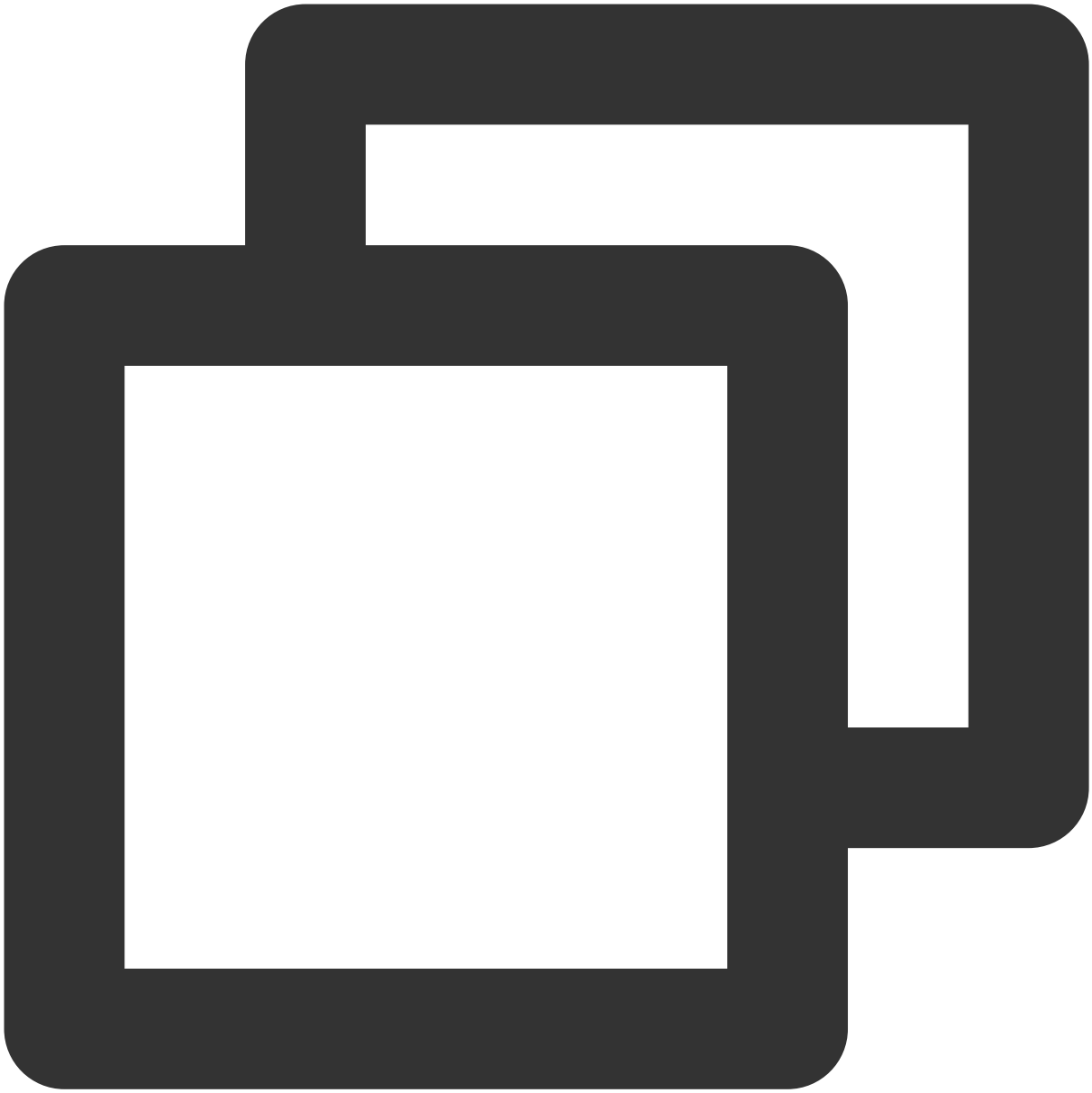
参数名：请参见 `tmio-option.h` 。

简单配置：请参见 `tmio-preset.h` 。



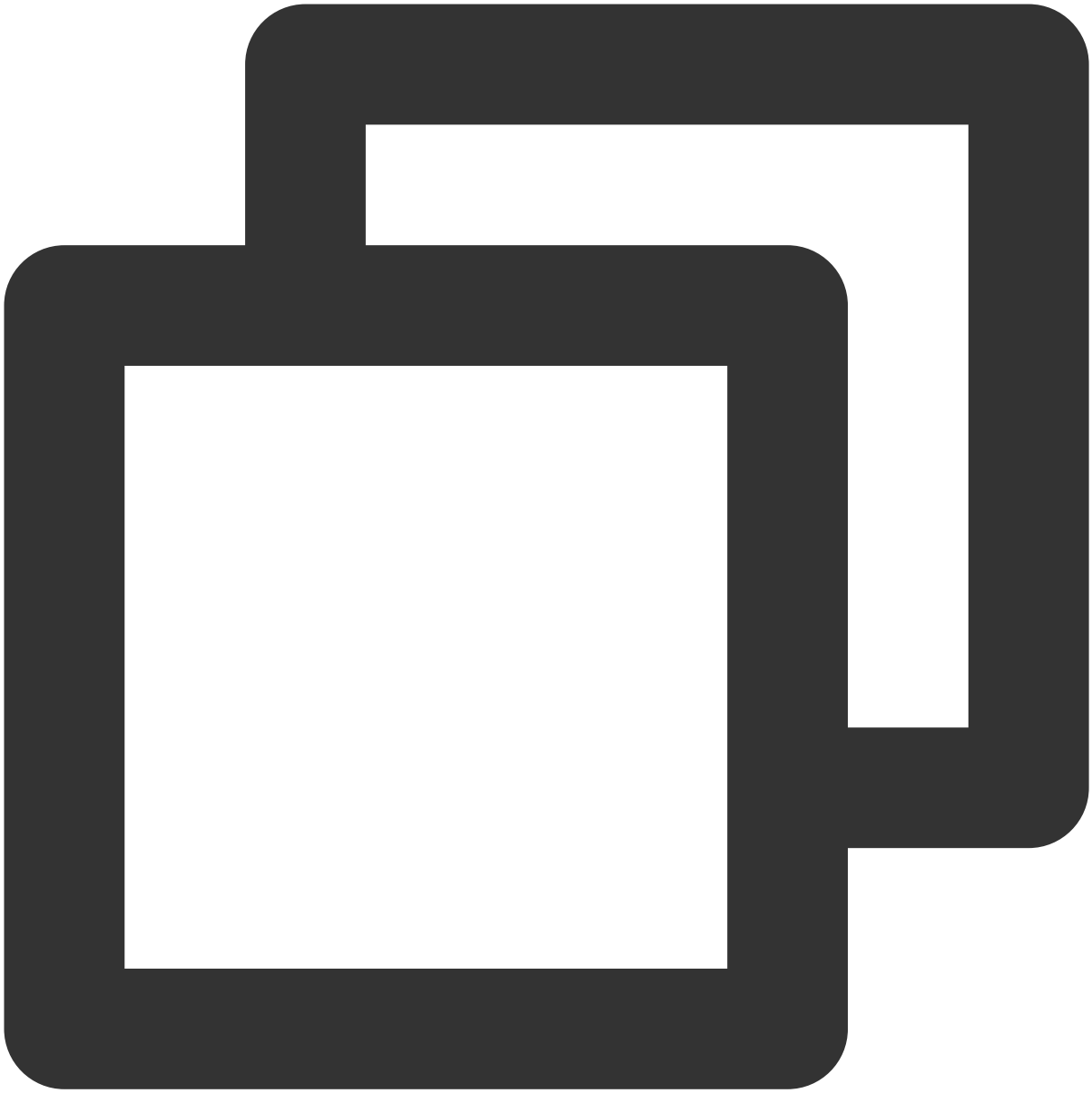
```
//根据不同参数属性选择合适的配置
bool setBoolOption(const std::string &optname, bool value);
bool setIntOption(const std::string &optname, int64_t value);
bool setDoubleOption(const std::string &optname, double value);
bool setStrOption(const std::string &optname, const std::string &value);
...
```

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;
```

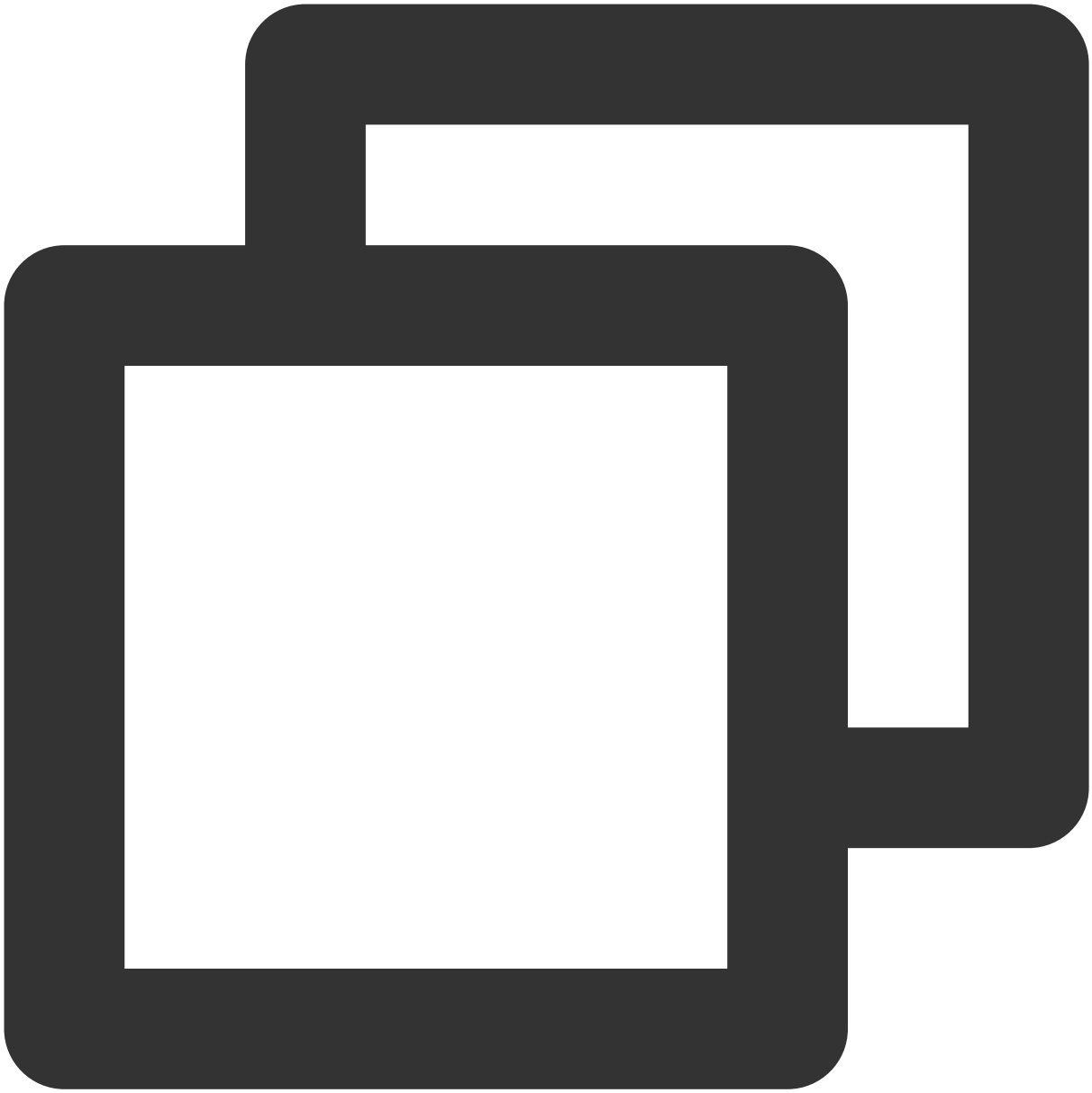
单链路（config 可为 NULL）



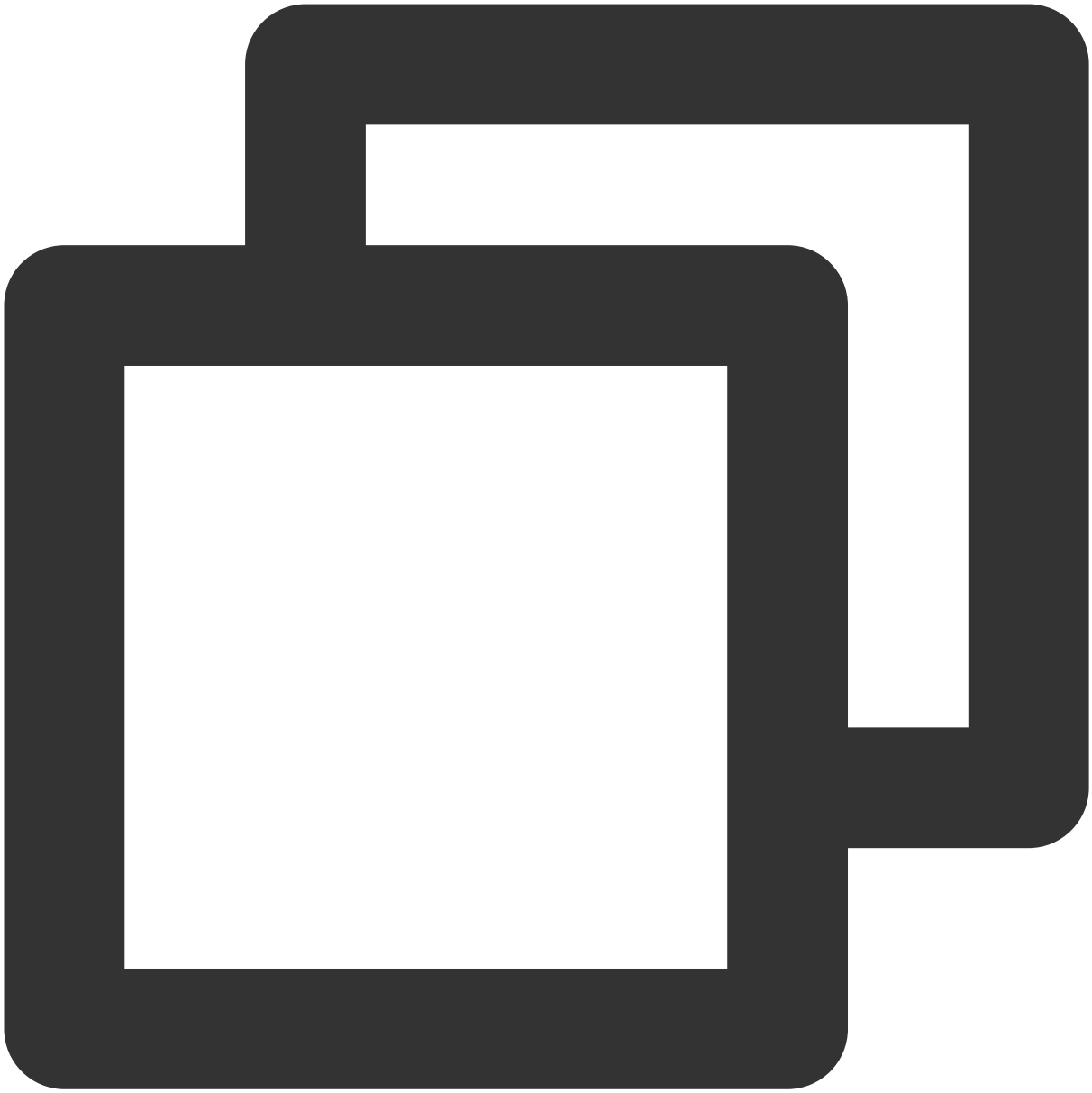
```
//默认单链路
auto err = tmio->open(TMIO_SRT_URL);
if (err) {
LOGE("open failed, %d, %s", err.value(), err.message().c_str());
}
```

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。



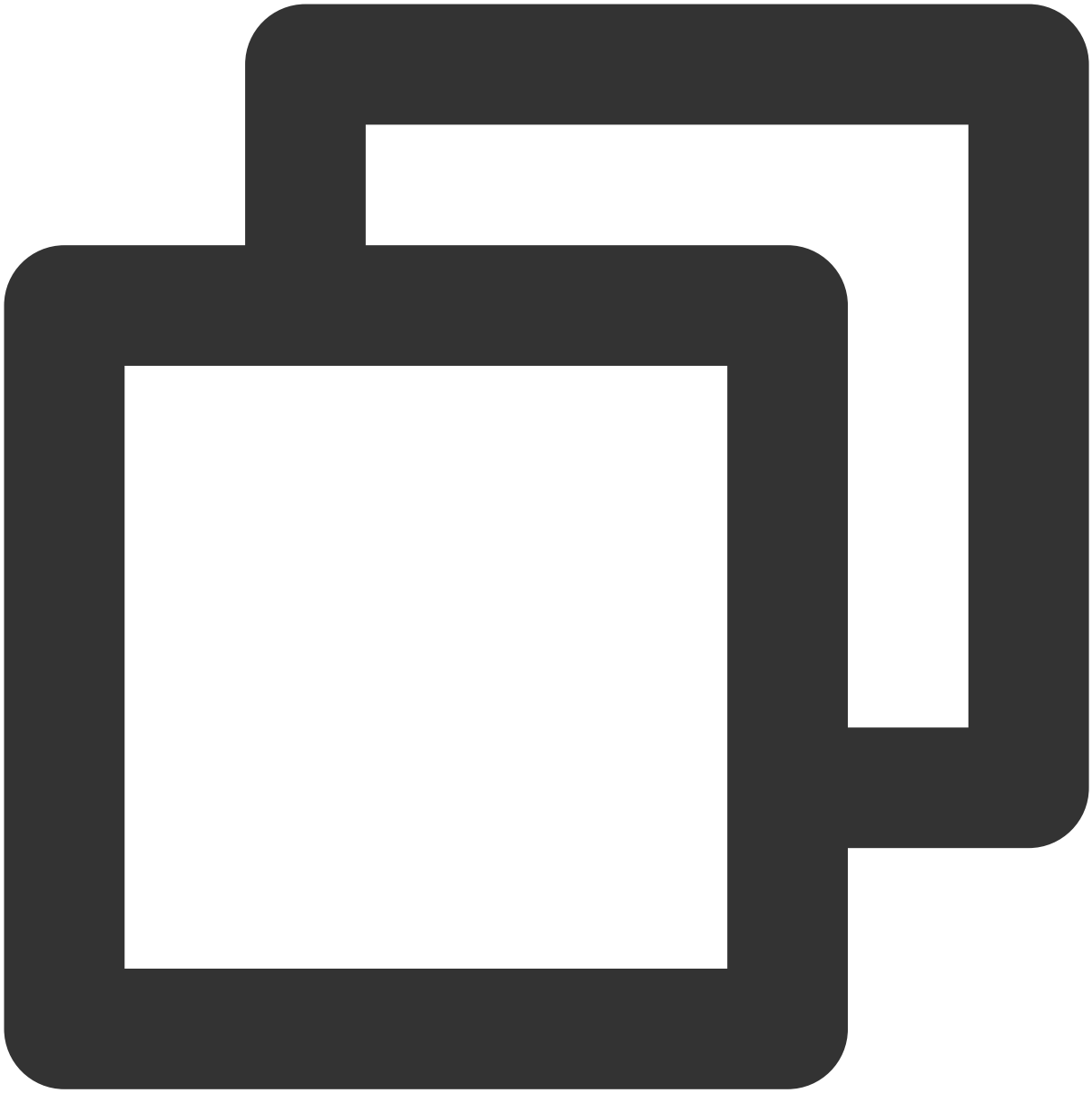
```
tmio::TmioFeatureConfig option_;
option_.protocol = tmio::Protocol::SRT;
option_.trans_mode = static_cast<int>(tmio::SrtTransMode::SRT_TRANS_BACKUP);
option_.addAvailableNet(net_name, local_addr, remote_url, 0, weight, -1);
```



```
//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。

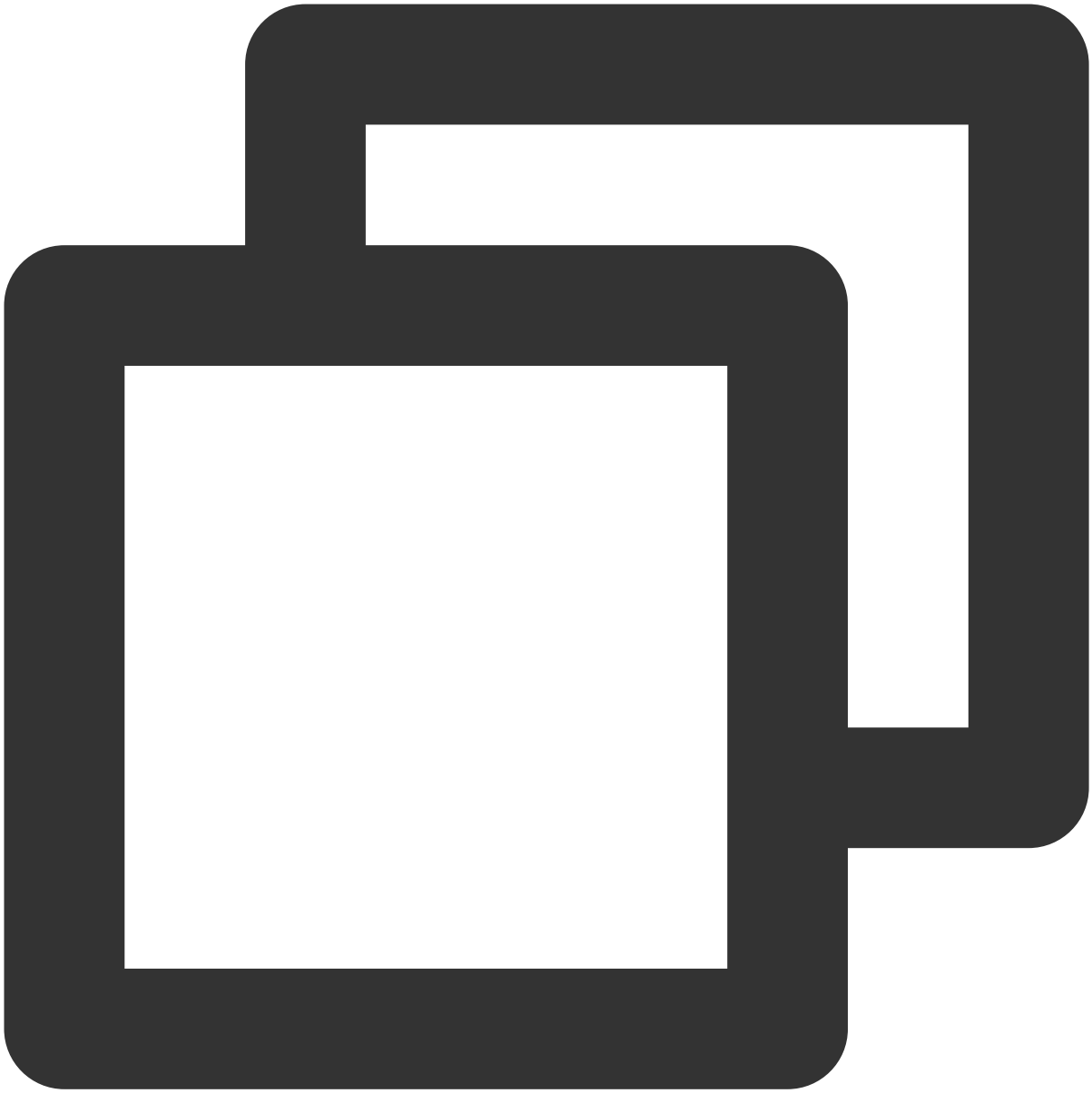
3. **发送数据**：



```
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**：

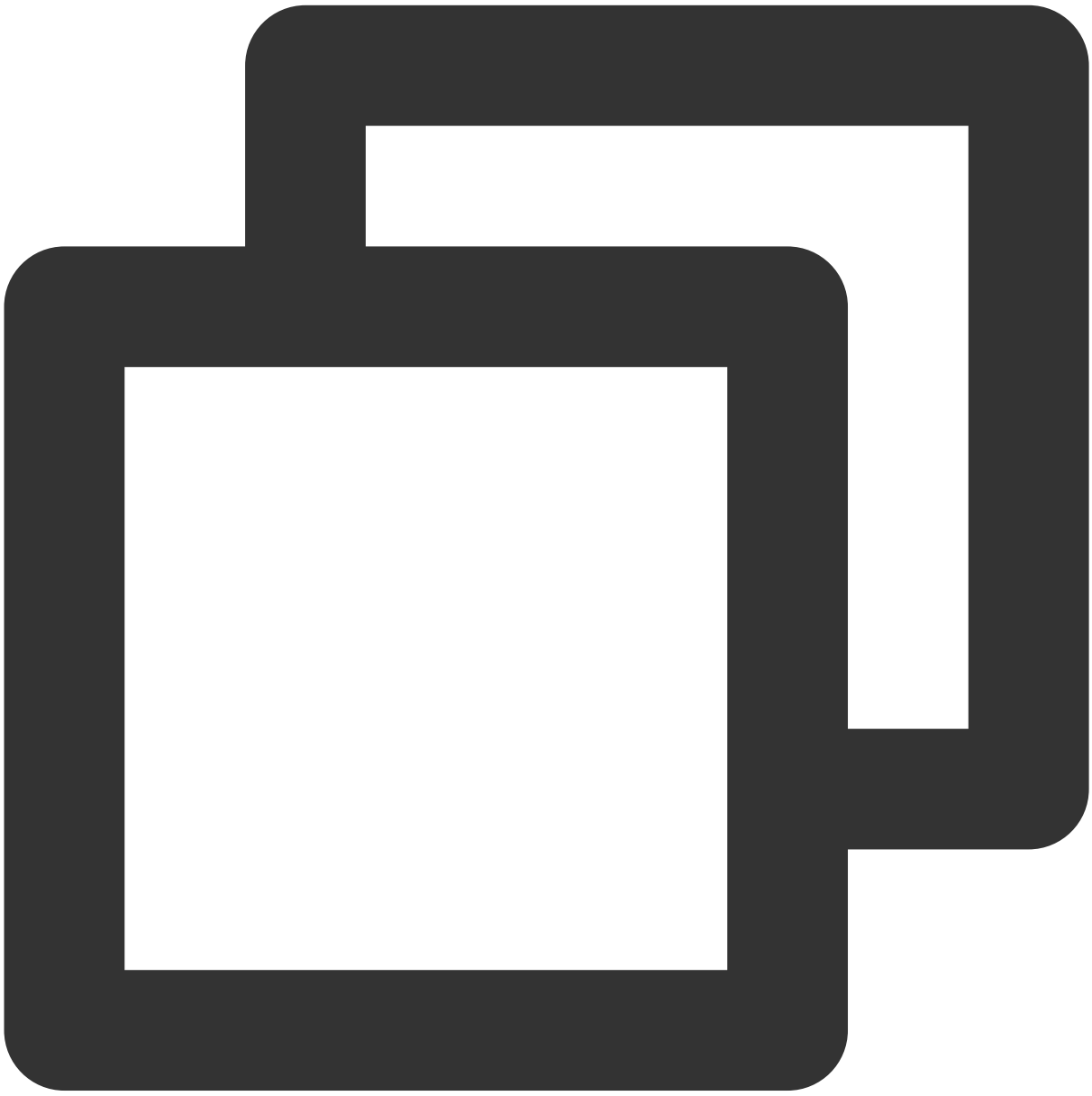
如果是需要交互的协议（如 RTMP）， 此时需要启用接收接口来读取数据完成协议交互，这里提供两个接口调用：



```
/**
* 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;
```

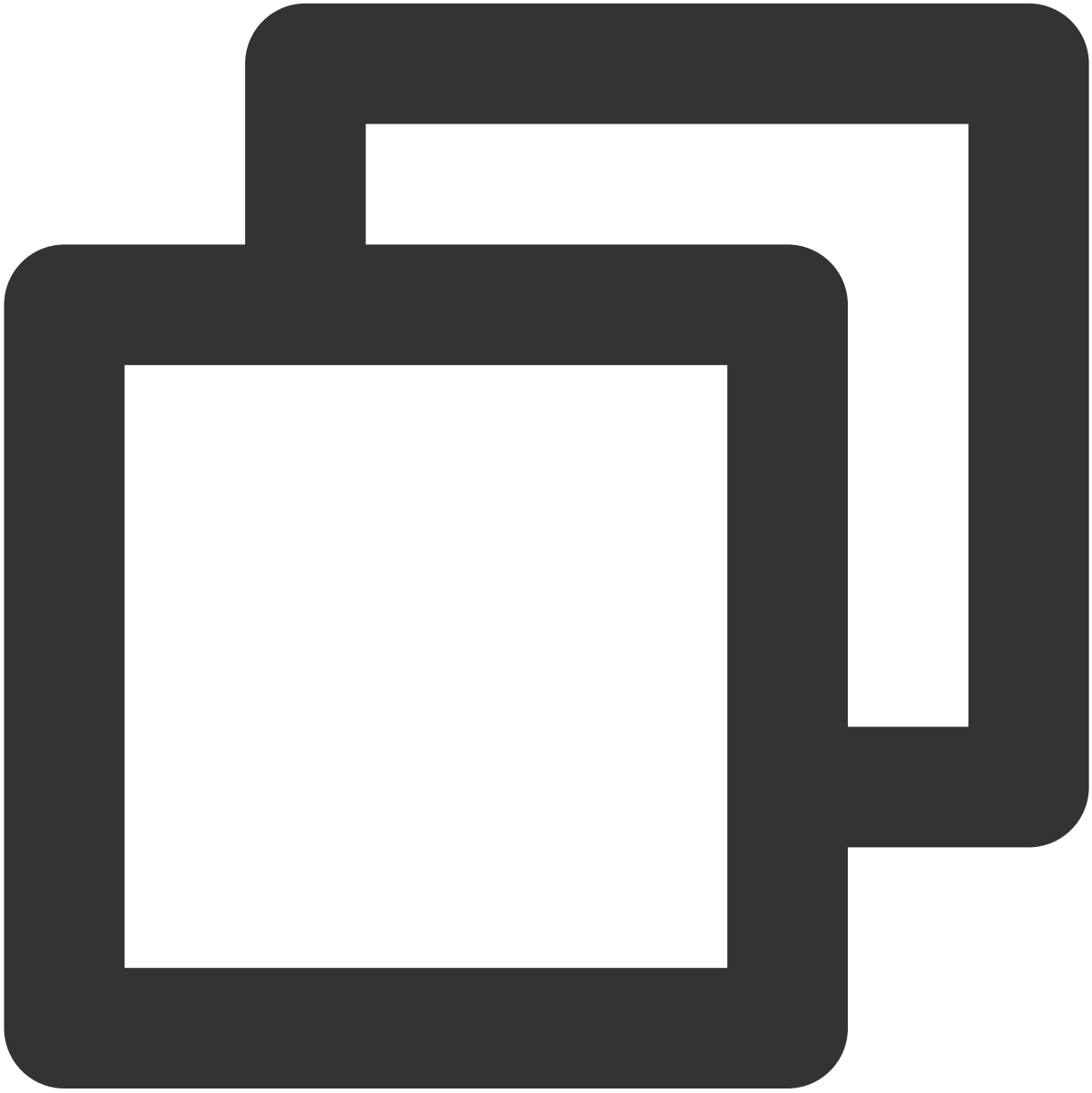
上层应用循环读取



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

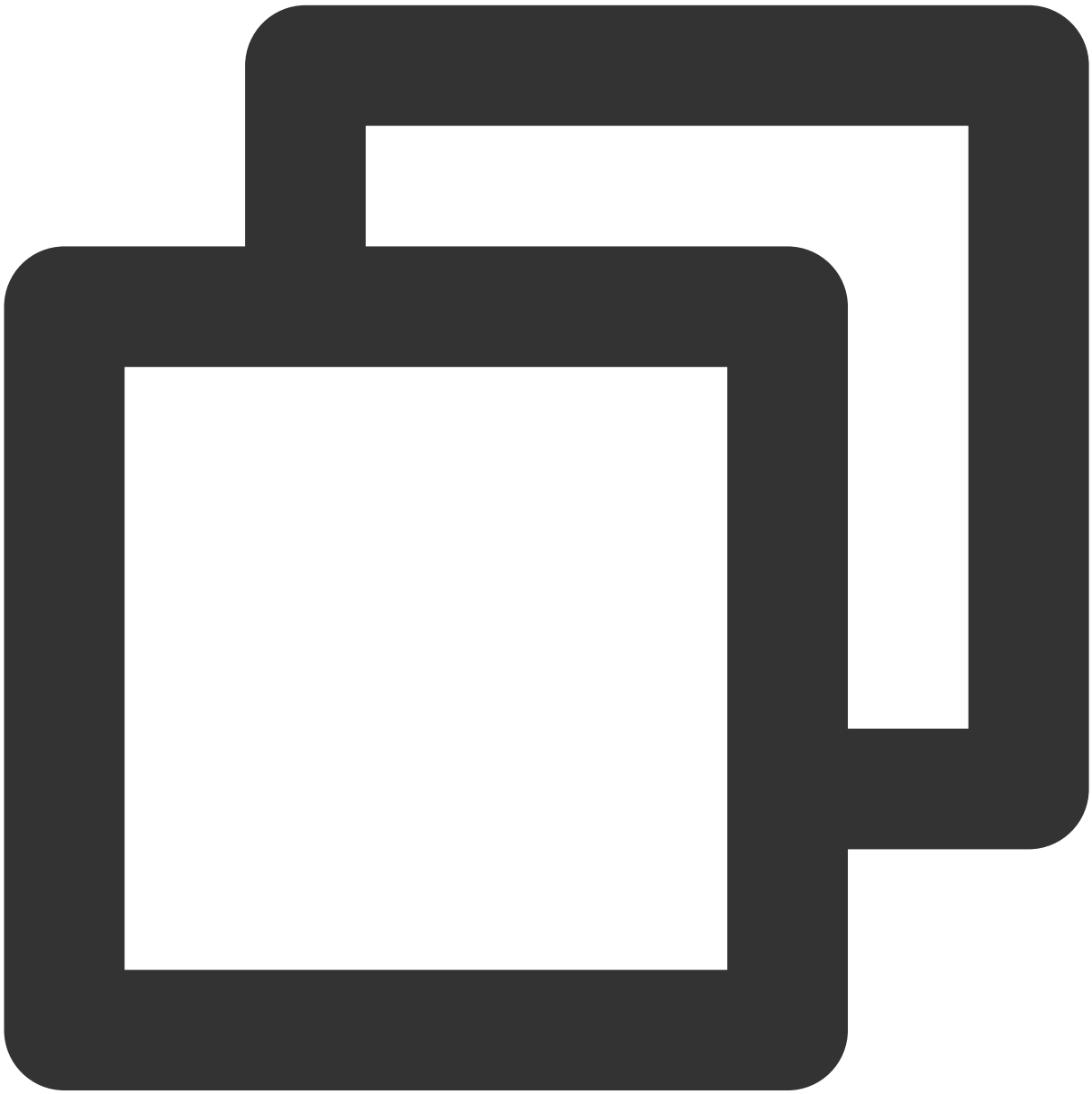
回调读取



```
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;
});
```

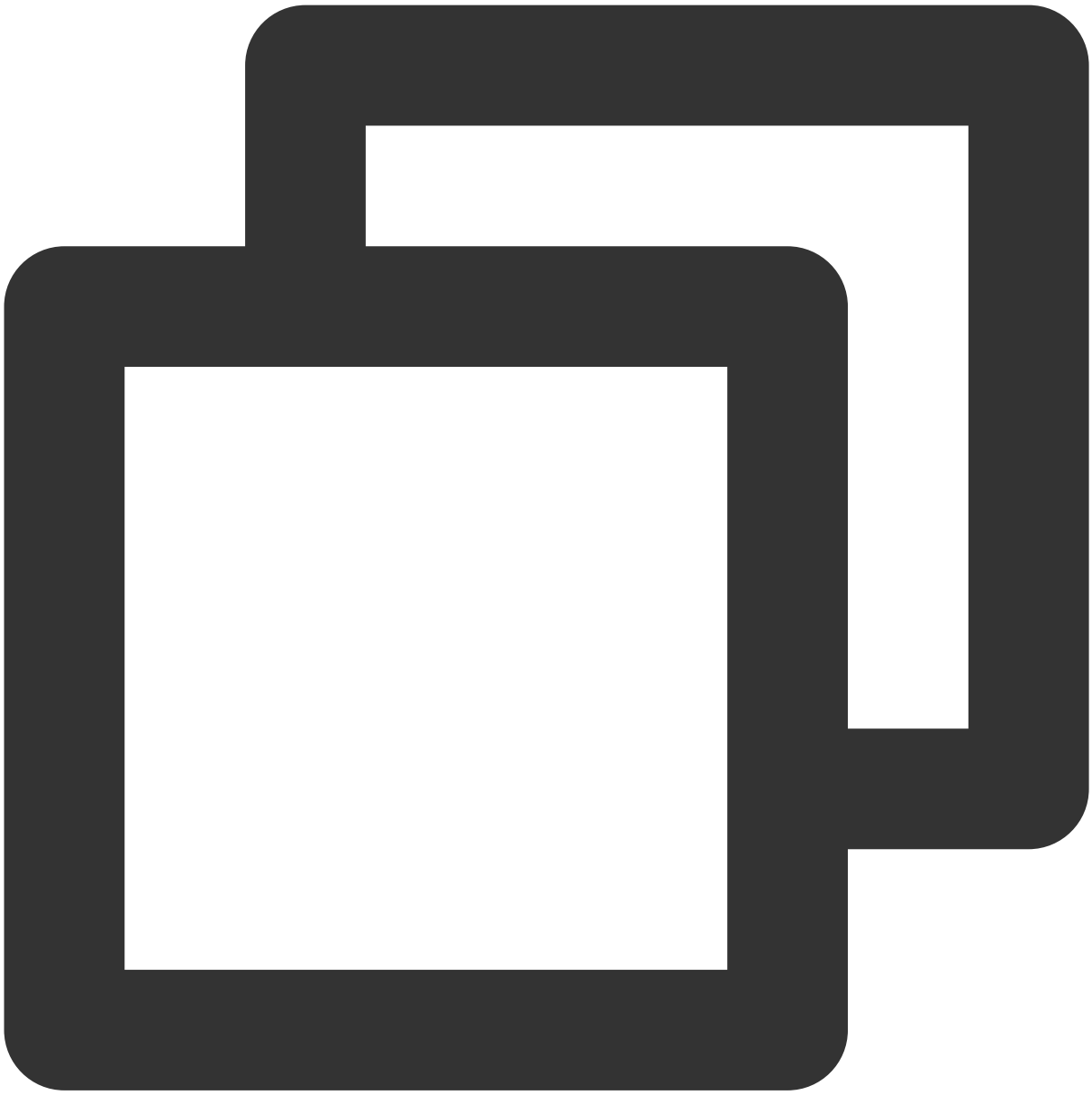
5. **关闭 Tmio**：



```
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。

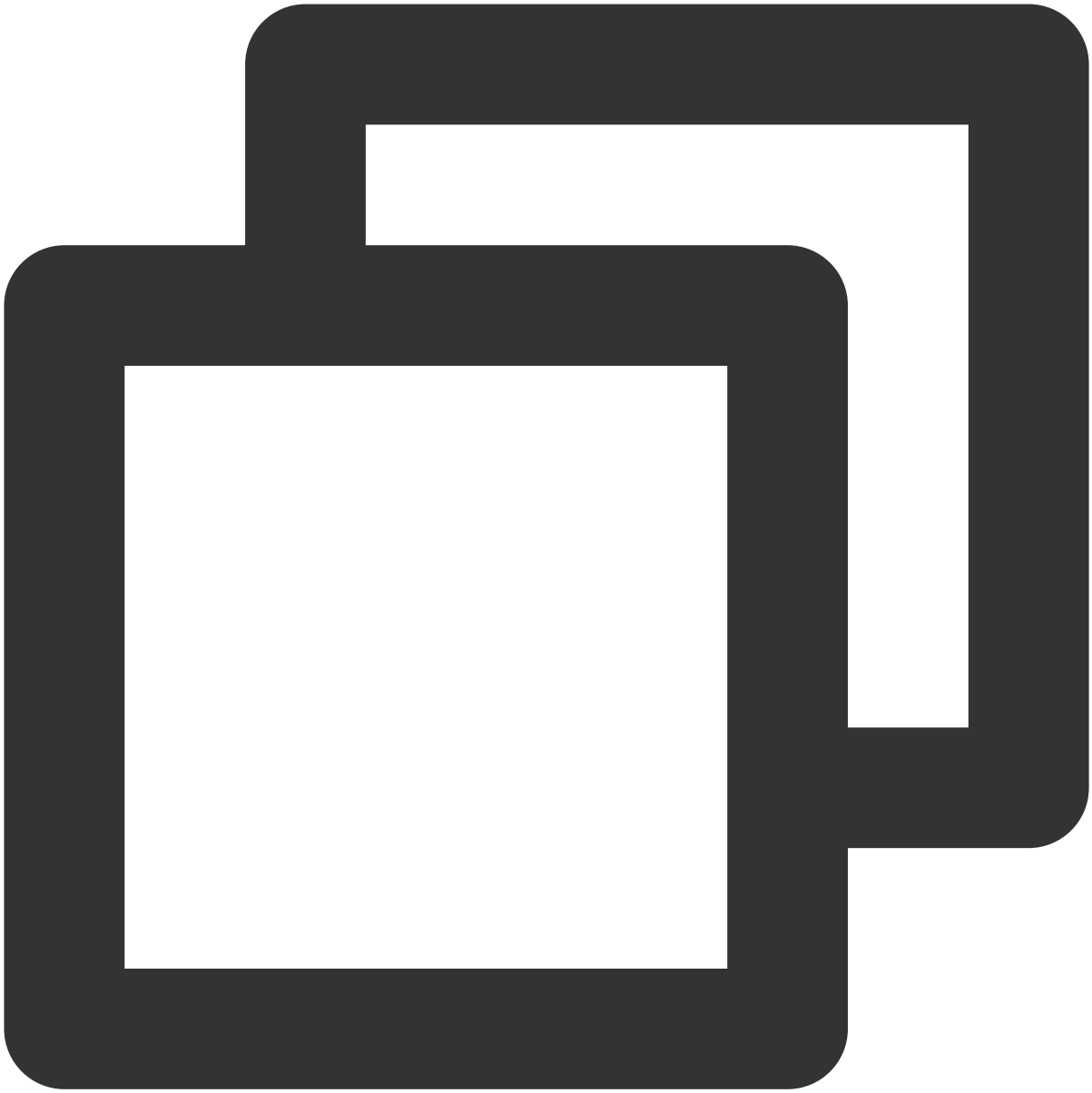
常见问题解答

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

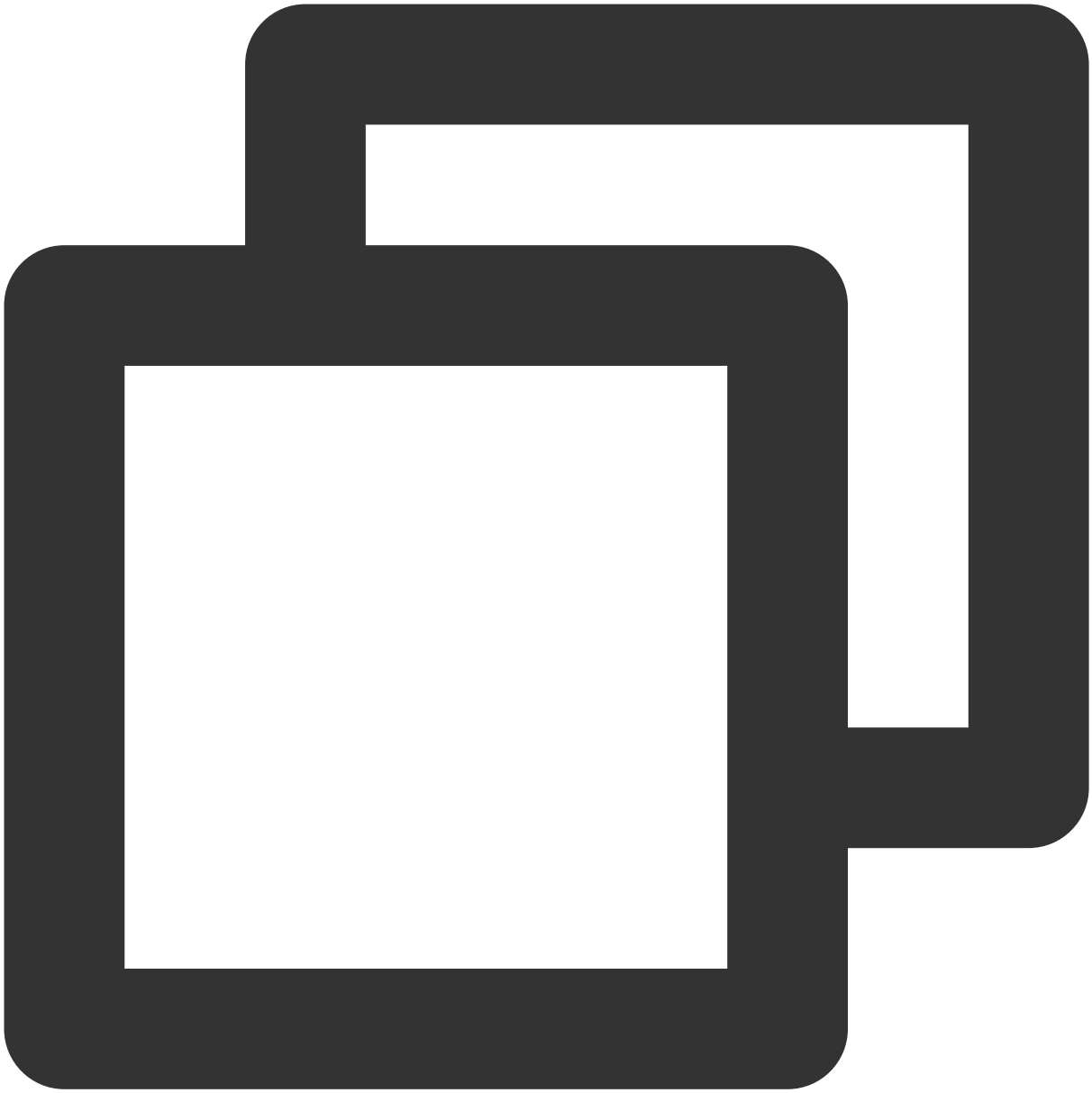
最近更新时间：2023-11-29 16:55:35

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。

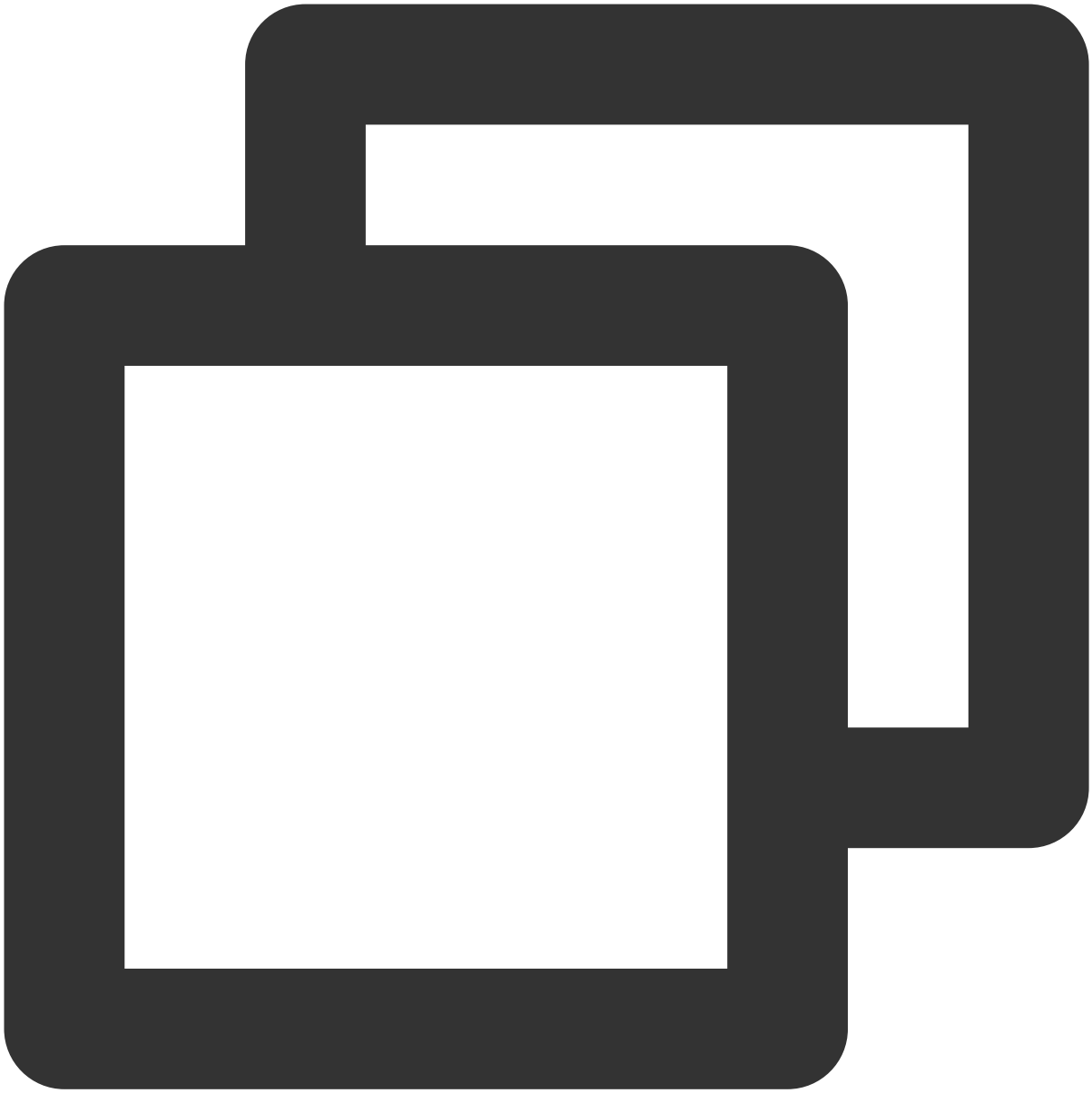
步骤1：获取视频效果管理实例



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

步骤2：开启本地混流

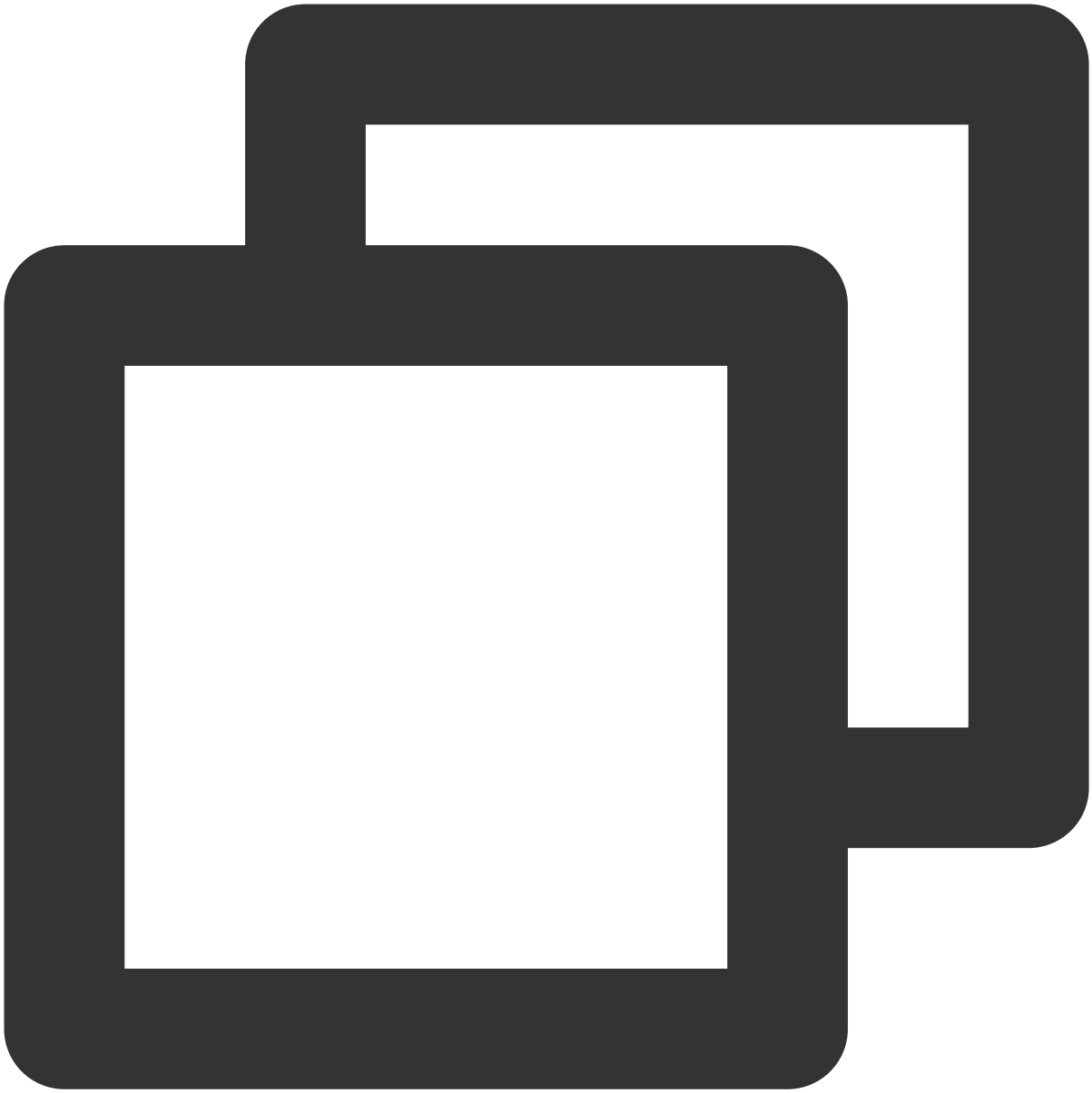
首先需要启用本地混流功能。默认情况下 SDK 只支持采集一路视频流和一路音频流，启用之后，就可以采集多路流，这些流将在浏览器本地进行混合处理。



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

步骤3：设置混流参数

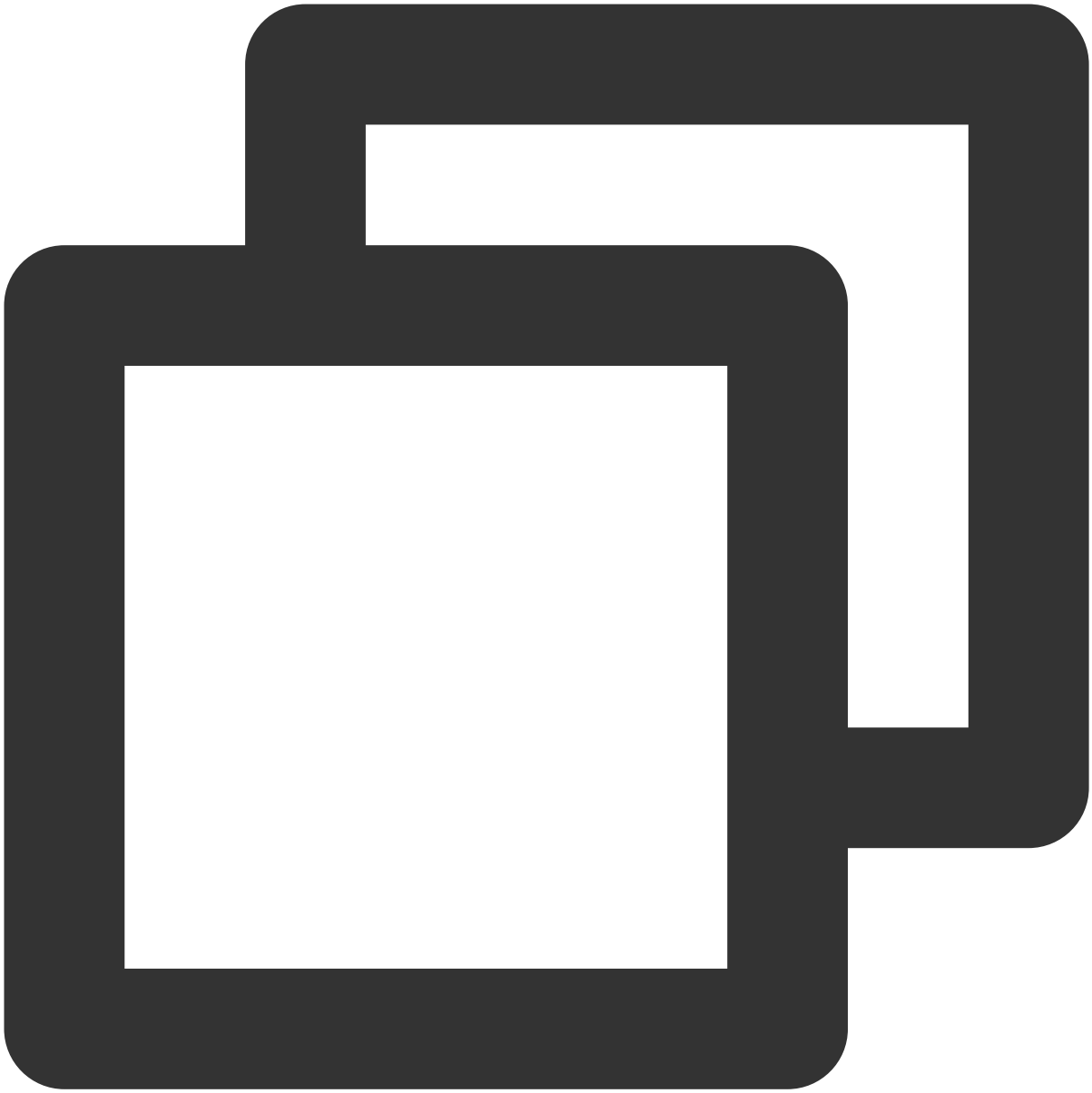
对混流参数进行设置，主要是设置最终混流后生成视频的分辨率和帧率。



```
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 。

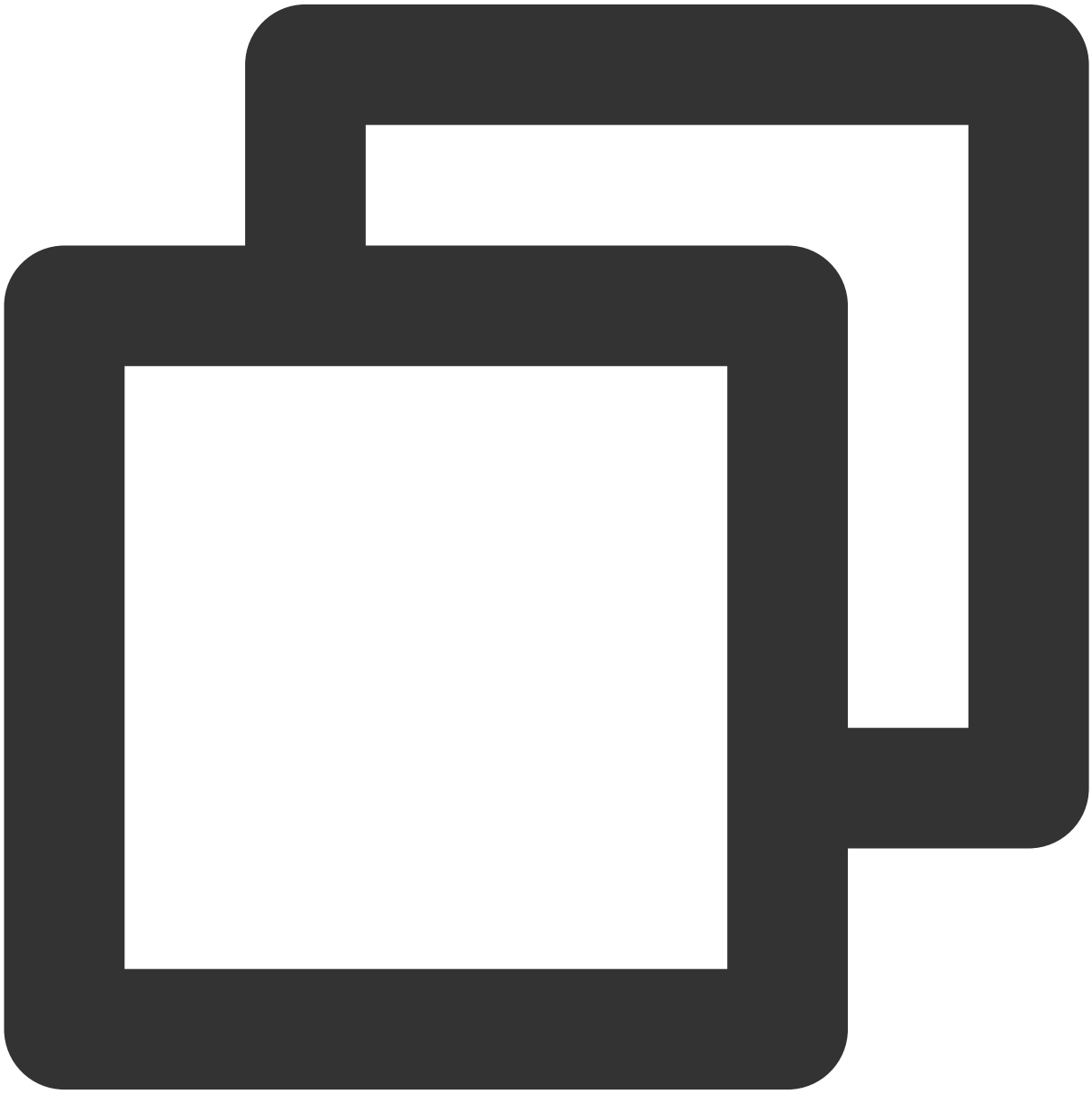


```
var cameraStreamId = null;
var screenStreamId = null;

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 。

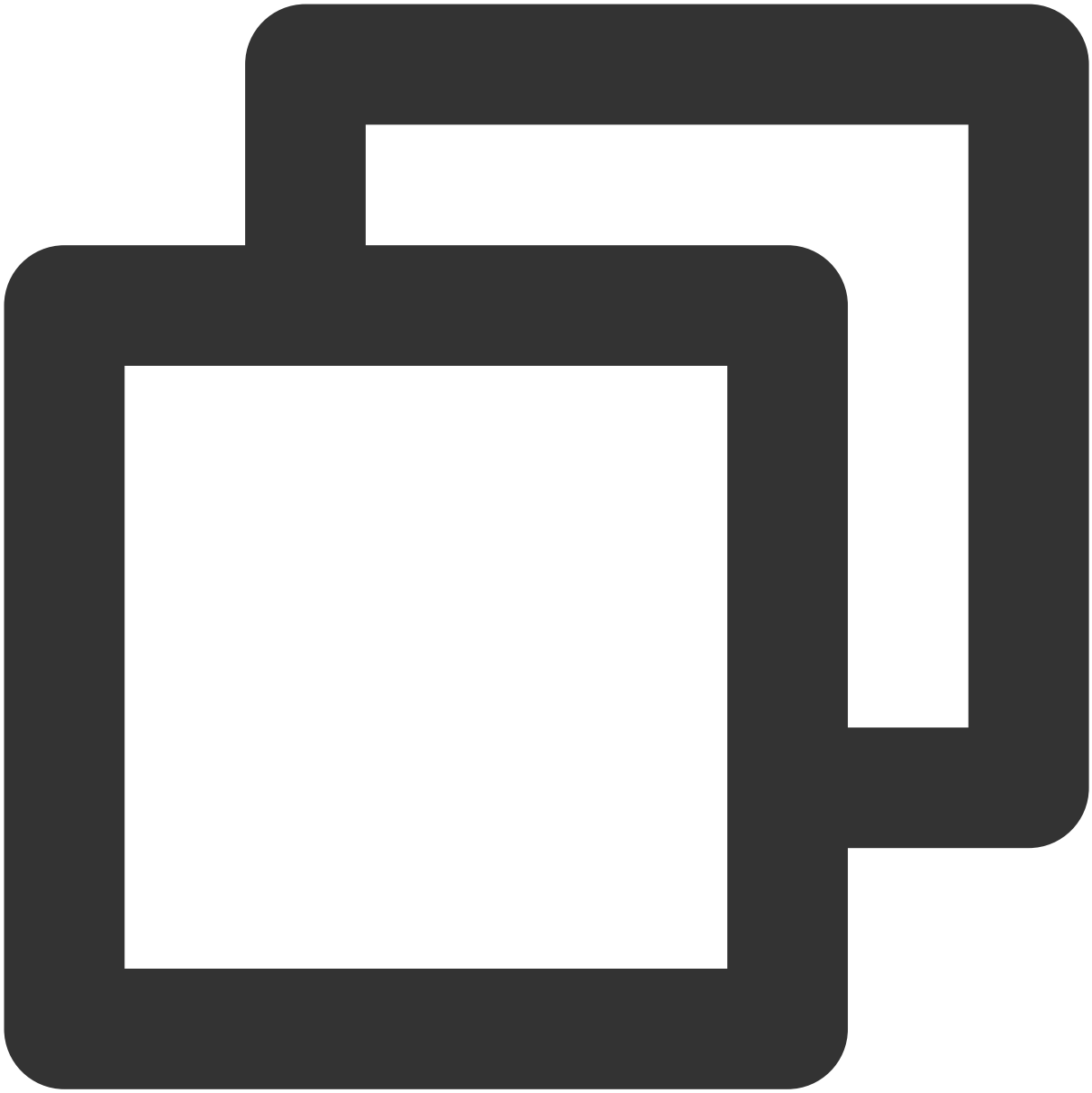


```
videoEffectManager.setLayout([{
    streamId: screenStreamId,
```

```
    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

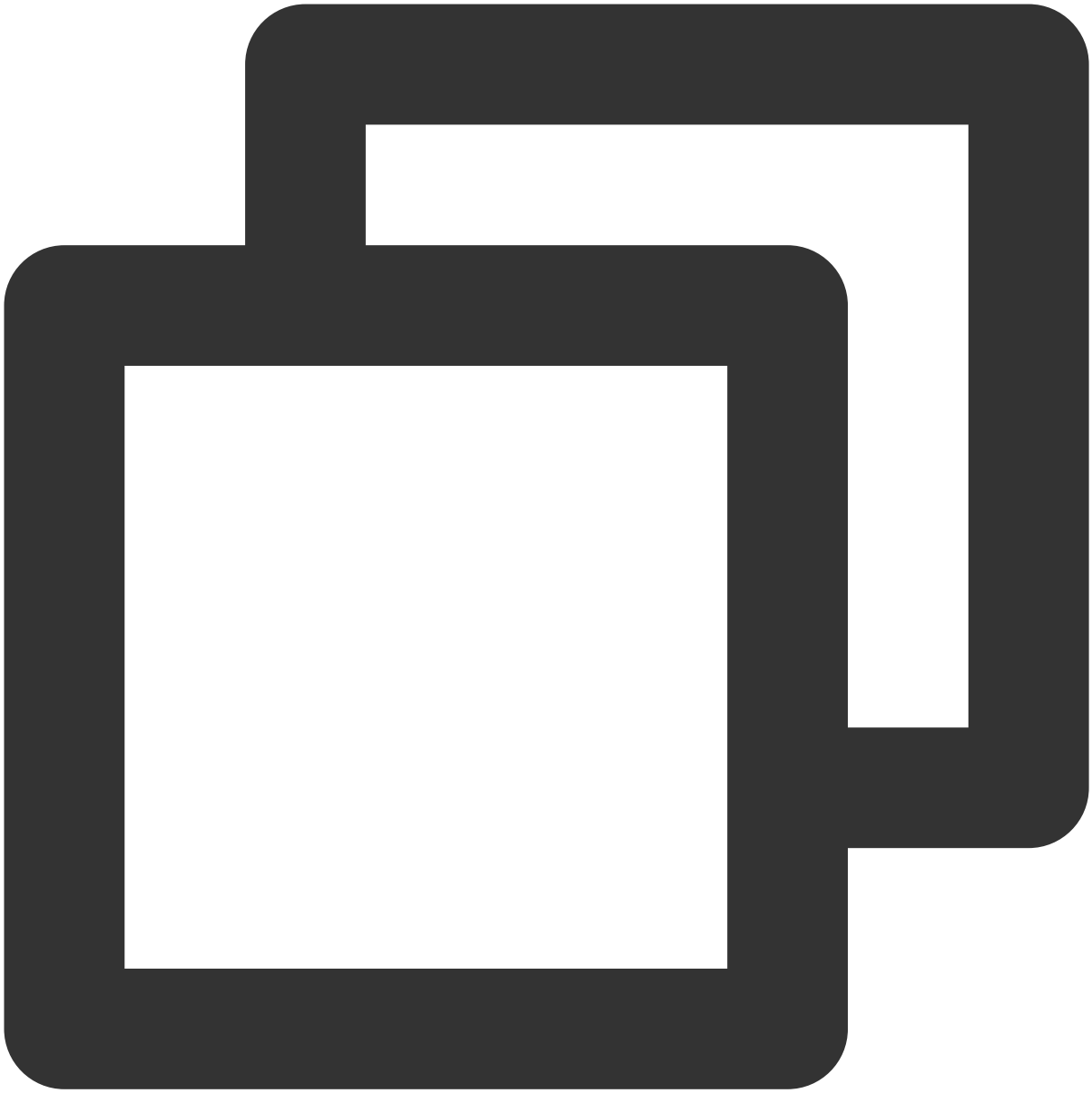
摄像头采集到的画面实际上是反的，这里对摄像头画面进行一次左右翻转。



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

步骤7：添加水印

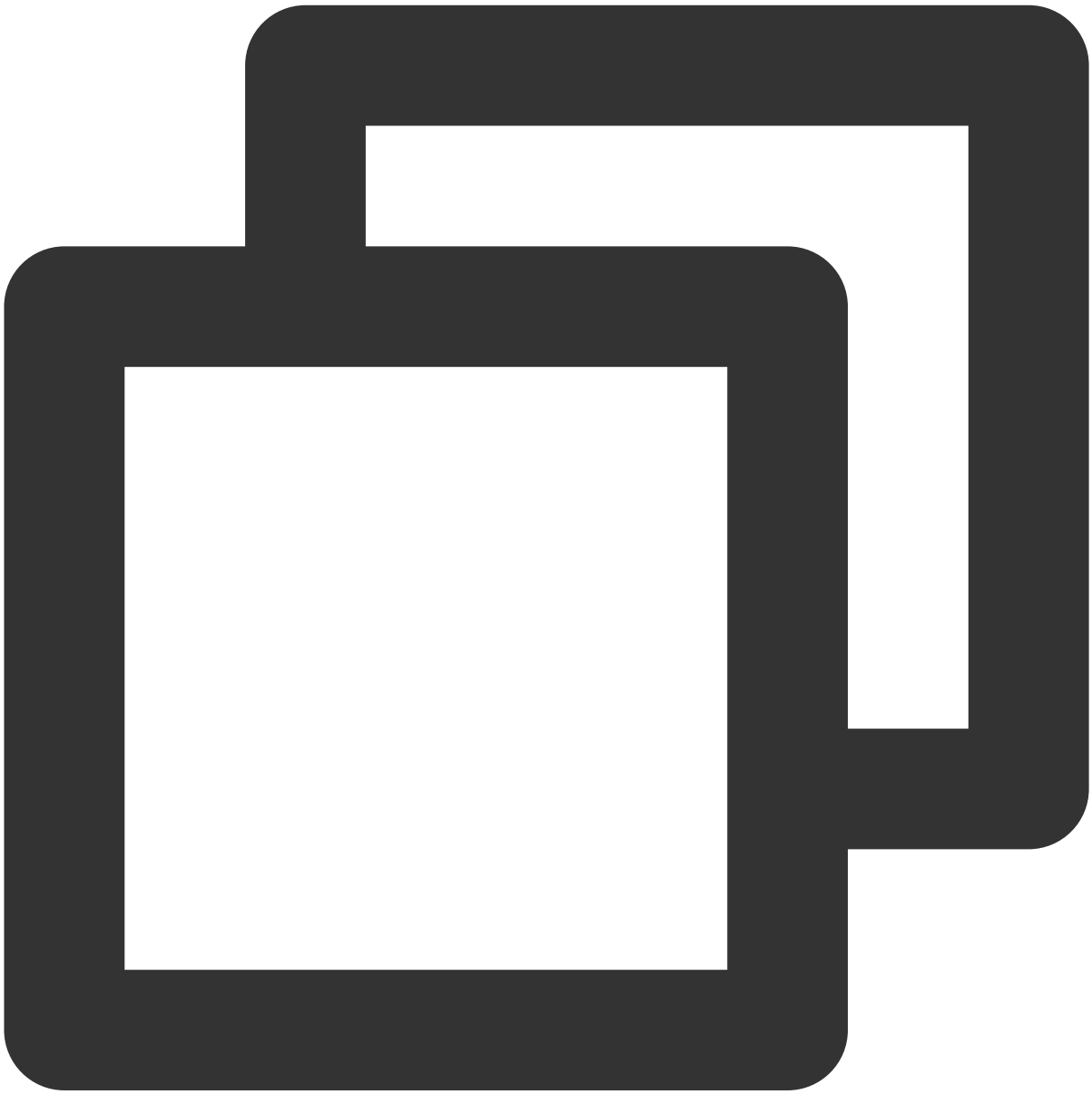
先准备好一个图片对象，然后将这个图片对象作为水印添加到视频流画面中，这里把水印图片放置在右上角。



```
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。

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

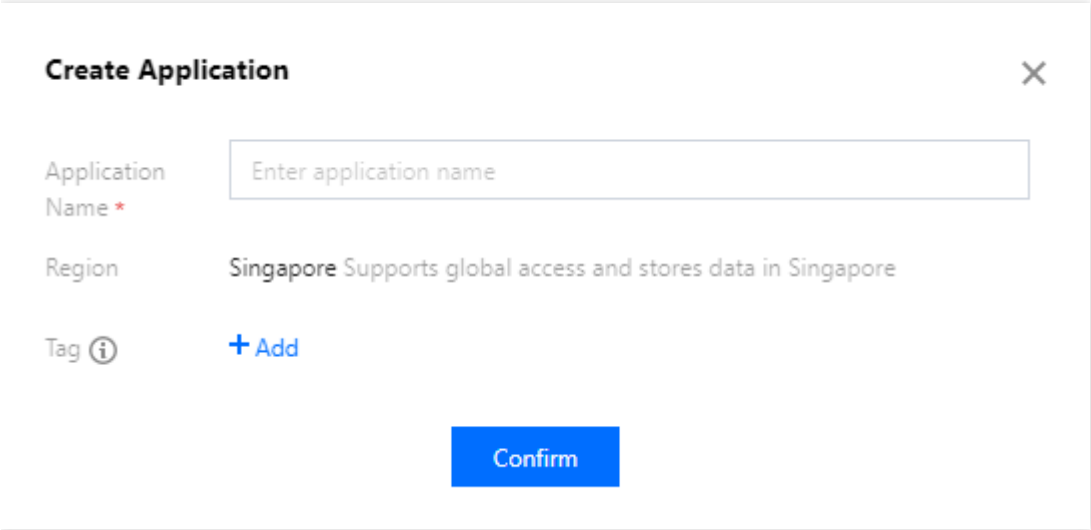
功能	说明
直播弹幕、送礼和点赞	支持亿级消息并发，轻松打造良好的直播聊天互动体验。
直播单聊、群聊等多种聊天模式	用户可以在这里发送自己想说的话并且可以收到同一个聊天室中其他成员的消息。支持文字、图片、语音、短视频等多种消息类型，实时消息推送，有效提升用户粘性与活跃度。

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：

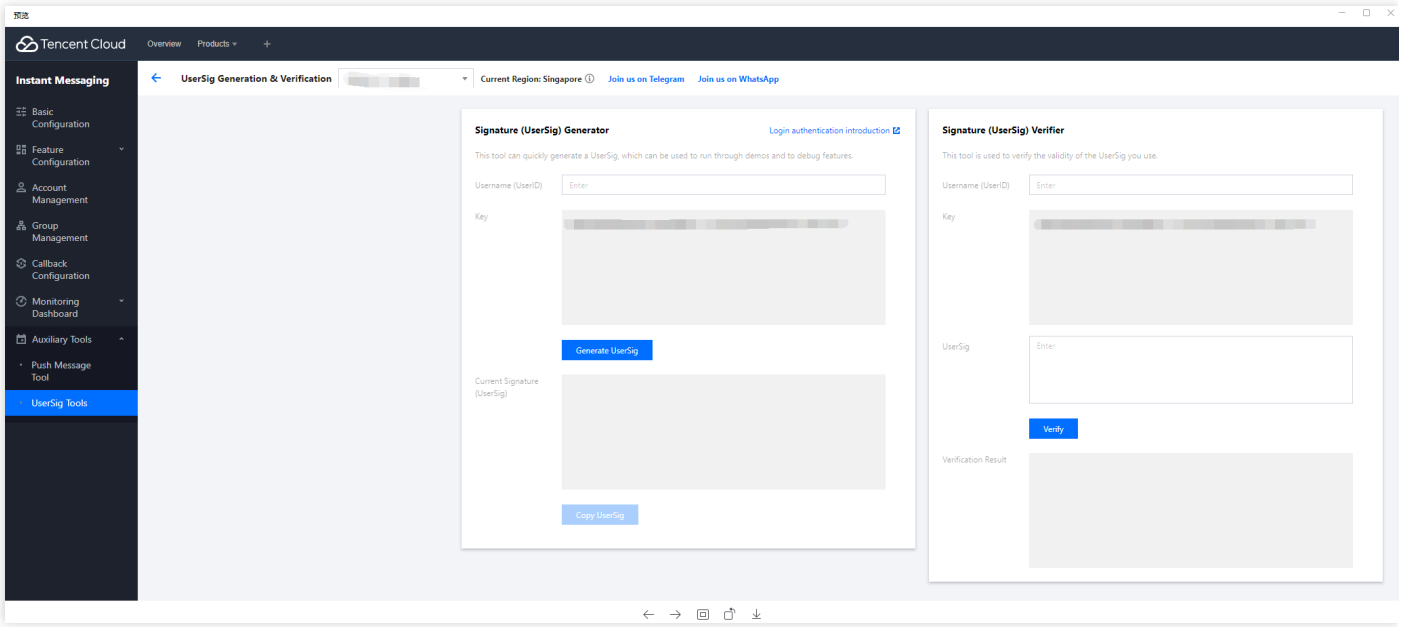


步骤2：完成相关配置

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：



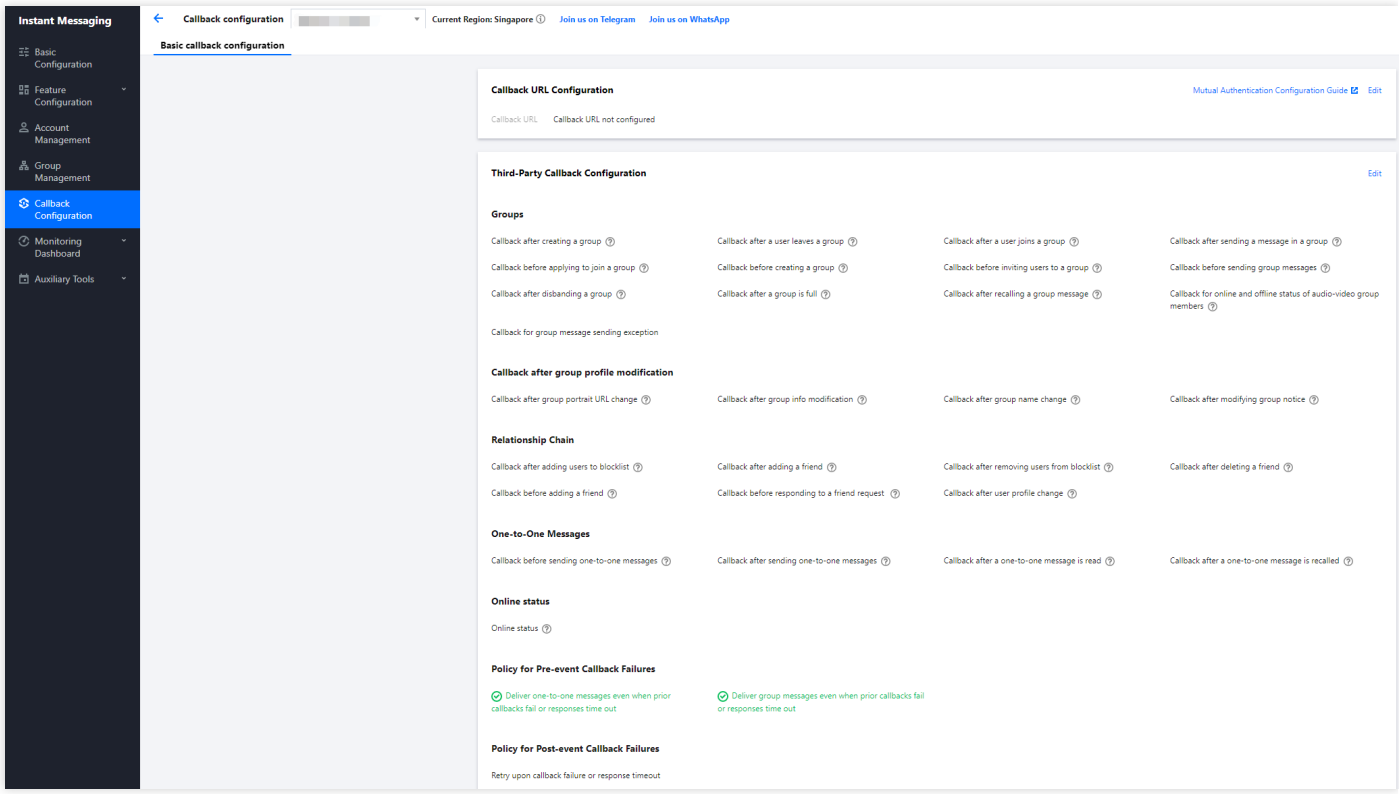
- **配置管理员账号**

  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**

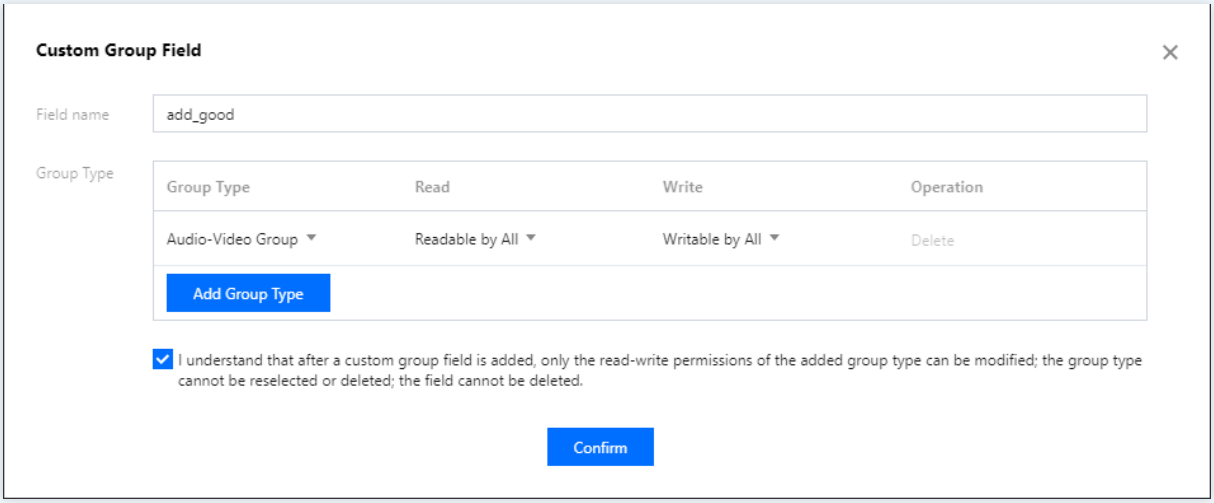
a. 登录 即时通信 IM 控制台，单击目标应用卡片，在左侧导航栏选择**功能配置 > 群组配置**。

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。