ALIBABA CLOUD

阿里云

视频直播

低延时直播

文档版本：20220422

阿里云

法律声明

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

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

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

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

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

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

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

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

目录

1.低延时直播简介 06
2.实现低延时直播的基本流程 08
3.超低延迟直播与标准直播的区别 10
4.Demo体验 11
5.SDK下载 14
6.发布日志 16
7.Native RTS SDK 21
7.1. Native RTS SDK简介 21
7.2. 阿里云播放器SDK集成Native RTS SDK实现说明 22
7.2.1. Android端 22
7.2.2. iOS端 26
7.3. 依赖FFmpeg的其他播放器集成Native RTS SDK实现说明 28
7.3.1. Android端 28
7.3.2. iOS端 37
7.3.3. Windows端 44
7.3.4. Mac端 45
7.4. Native RTS SDK编程手册 47
7.4.1. Native RTS SDK编程简介 47
7.4.2. 编程模型 48
7.4.3. 参数设置 50
7.4.4. API 52
7.4.4.1. 数据类型 52
7.4.4.2. 接口说明 55
7.4.5. 消息回调 60
7.4.6. 日志回调 62
8.Web RTS SDK 65

8.1. Web RTS SDK简介 65
8.2. 接口说明 66
8.3. 错误码说明 69
9.OBS插件SDK 71
9.1. OBS插件SDK简介 71
9.2. OBS插件SDK集成与实现 71
9.3. 实现OBS OutPut插件功能 74
9.4. API 78
9.4.1. 数据类型 78
9.4.2. 接口说明 82
9.4.3. 回调说明 85
10.RTS信令协议规范 87

1.低延时直播简介

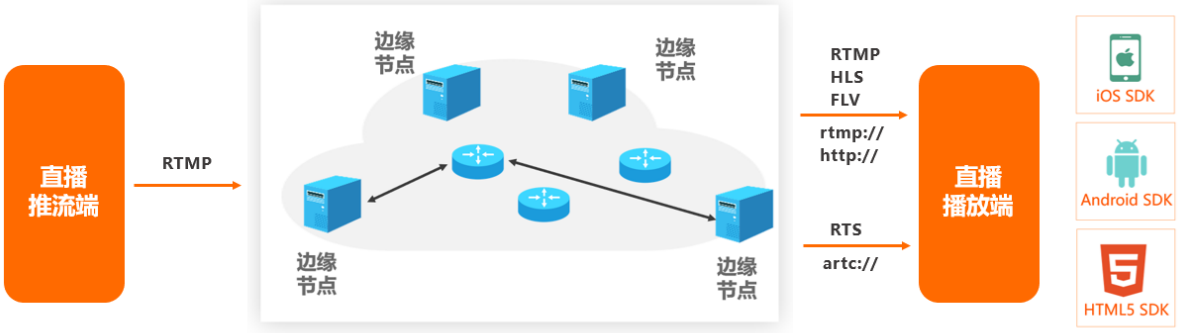
低延时直播RTS（Real-time Streaming）是视频直播的重要增值功能，可以提供客户端易接入、超低延迟、高并发、高清流畅的视频直播服务。通过阅读本文，您可以快速了解低延时直播的基本信息及使用方法。

产品特性

RTS在阿里云视频直播的基础上进行全链路延时监控、CDN传输协议改造、UDP等底层技术优化，通过集成阿里云播放器SDK，支持在千万级并发场景下节点间毫秒级延时直播的能力，弥补了传统直播存在3~6秒延时的问题，确保了低延时、低卡顿、秒开流畅的直播观看体验。

架构方案

低延时直播仅需在现有的直播业务上，新增一个RTS播流域名，实现多种协议拉流。架构如下所示：



> 说明
>
> - 直播推流端沿用推RTMP流的方式推流。推流端网络质量、流GOP大小、观众播放器缓存等都会影响全链路延迟效果。
> - 标准直播拉流（RTMP、FLV、HLS）使用原有rtmp://和http://格式；低延时直播拉流（UDP）使用artc://格式。

效果展示

低延时直播与普通直播效果如下所示：

适用场景

场景	描述
教育直播	应用于大班课，可以支持超大数量规模的同学同时在线低延时与老师互动。
电商直播	卖家与买家实时互动答疑，交流商品信息。
体育直播	直播精彩体育、电竞等赛事，让观众实时了解现场情况。
互动娱乐	主播可以在观众送礼物时及时反馈，增强与观众之间的互动。

计费说明

不同于标准直播，低延时直播按新价格体系计费，规则如下所示：

- 按低延时直播收费项计费，不再叠加标准直播计费项。
- 与标准直播计费方式一致，低延时直播分为按流量方式和按带宽方式计费。
- 标准直播计费方式变更后，低延时直播计费方式也会同步变更。

更多关于低延时直播计费信息，请参见低延时直播费用。

Demo体验

您可以扫描并安装（或复制扫描后的地址至浏览器上进行下载并安装）RTS低延时直播Demo体验。

Andriod iOS

2.实现低延时直播的基本流程

通过阅读本文，您可以了解实现低延时直播的基本流程。

操作视频

操作步骤

1. 开通低延时直播功能。
   i. 登录视频直播控制台。
   ii. 在左侧导航栏选择**域名管理**，进入域名管理页面。
   iii. 单击指定播流域名**操作**列的**域名配置**，进入域名管理页面。
   iv. 选择**直播管理 > 低延时直播**。
   v. 打开**低延时直播**开关。

   > 说明
   > - RTS配置全网下发需要数分钟时间，请等待后测试。
   > - 如果您需要一个推流域名对应多个播流域名时，请参见关联主、子播流域名。
   > - RTS支持和普通直播协议（RTMP、FLV、HLS）共用一个播流域名。但是鉴于RTS使用UDP协议，普通直播使用TCP协议，两者技术有所区别，建议您为RTS单独开通一个播流域名。

2. 推流。具体操作，请参见推流、拉流与播流。

   低延时直播Web RTS SDK对音视频解码存在以下限制：
   - 音频不支持AAC编码（会出现噪音）。
   - 视频不支持B帧（会出现画面跳跃）、H.265编码，GOP不大于3秒（会增加延迟）。

   如果您无法控制源码流参数，可以使用RTS转码服务来调整参数，转码可能会增加数百毫秒的延时，详情请参见RTS转码。

3. 播流。

   播RTS流时，RTS SDK配套的播放器和播流地址相关说明如下所示：
   - 播放器

播放器类型	相关文档
阿里云移动端播放器	阿里云播放器SDK集成Native RTS SDK实现说明
依赖FFmpeg的其他播放器	依赖FFmpeg的其他播放器集成Native RTS SDK实现说明
阿里云Web端播放器	Web RTS SDK简介

   - 播流地址：相对于普通直播播流地址，RTS播流地址使用的协议头为 `artc://` 。例如： `artc://播放域名/AppName/StreamName?鉴权串` 。更多信息，请参见推流地址和播放地址。

> **说明** Web RTS SDK音频不支持AAC编码格式，需要使用RTS转码功能转成Opus格式，因此Web RTS SDK播流地址为转码播流地址。例如：`artc://播放域名/AppName/StreamName{_转码模板id}?鉴权串`。RTS转码，请参见RTS转码。

4. （可选）直播录制。

   RTS录制方式同普通直播的录制。具体操作，请参见录制存储至VOD、录制存储至OSS。

3.超低延迟直播与标准直播的区别

标准直播支持平滑升级至超低延迟直播。如果您正在使用标准直播服务，可以通过打开超低延迟直播开关，同时使用标准直播和超低延迟直播服务；如果您在使用超低延迟直播播放失败，也可以降级至标准直播，进一步保障业务的可靠性。通过阅读本文，您可以了解超低延迟直播与标准直播的区别。

指标说明

指标	超低延迟直播	普通直播
播放协议	支持ARTC（基于WebRTC）协议流的播放。	支持RTMP、FLV、HLS协议流的播放。
端到端延迟	ARTC协议的流（通过UDP协议传输）延迟小于1秒。	• RTMP、FLV协议的流（通过TCP协议传输）延迟大于3秒。 • HLS协议的流（通过TCP协议传输）延迟大于6秒。
抗弱网能力	抗弱网能力强，在丢包30%的情况下可以流畅播放。	抗弱网能力有限。
兼容性	• Native端：兼容iOS、Andriod、Mac、Windows平台。 • Web端：终端兼容率大于94%。	视协议而定。
卡顿率	相对于标准直播，超低延迟直播大幅优化端到端延迟，同时保持卡顿率与标准直播持平或略有优化。	
并发能力	千万级并发能力。	

使用场景

直播类型	使用场景
超低延迟直播	• 强互动场景：常用于电商直播、群直播、教育直播、企业直播等，低延迟和强互动的特性有助于提升商业转化率和用户参与度。 • 多用户画面一致场景：常用于赛事直播，让观众实时了解赛事现场情况，避免观众间画面不同步的现象。
普通直播	弱互动场景：以观众观看为主，且允许多用户画面由于网络差异导致画面不同步现象。

4.Demo体验

阿里云低延时直播提供移动端和Web端Demo供您体验，建议您优先使用移动端Demo体验低延时直播服务。通过阅读本文，您可以了解低延时直播Demo的运行方法。

前提条件

- 您已经生成已鉴权的推流地址。具体操作，请参见生成推流地址和播放地址。
- 您已经开通低延时直播功能。具体操作，请参见开通低延时直播功能。
- 您已下载并安装OBS推流工具。详情请参见OBS推流工具。

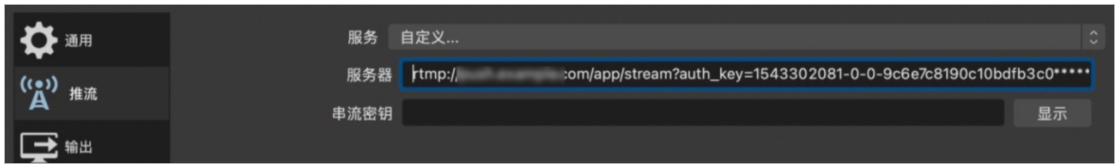
环境要求

移动端环境要求如下所示：

平台	环境要求
iOS	支持iOS 9.0及以上版本。
Android	支持Android 4.3及以上版本。

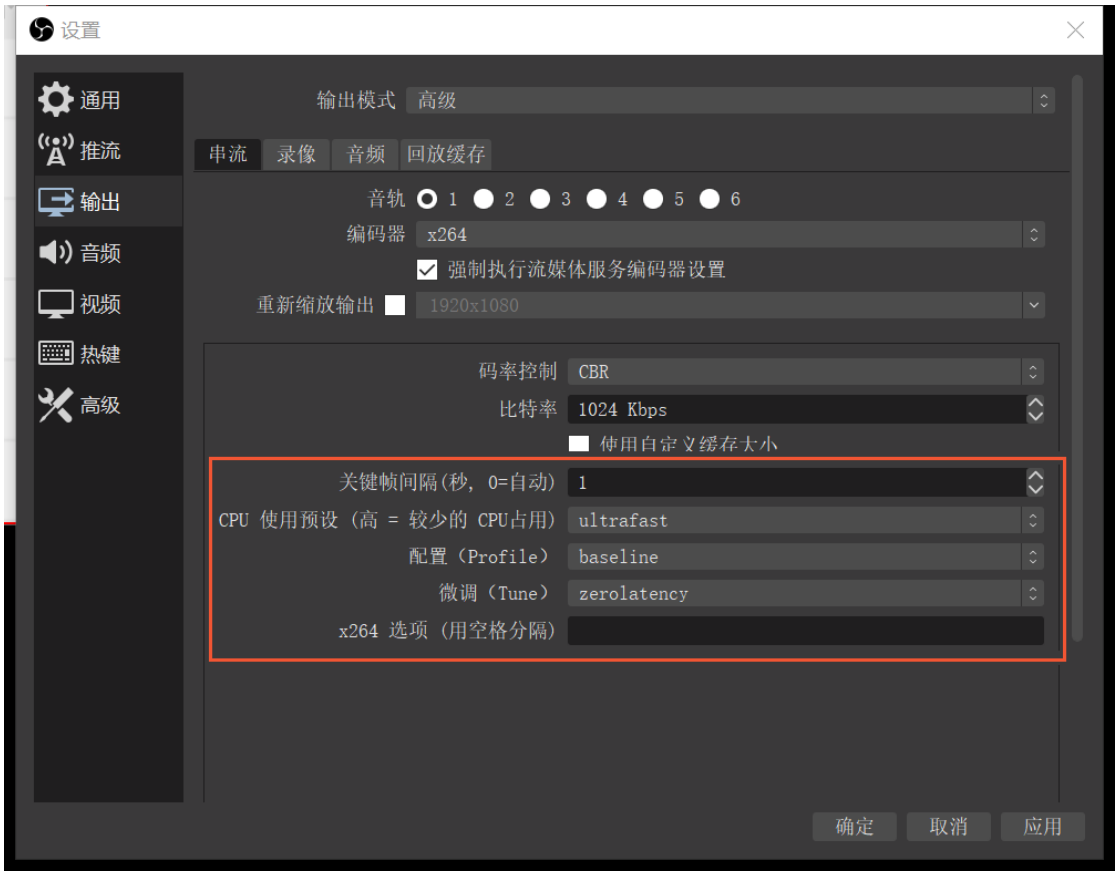
操作步骤

1. 使用OBS工具推流。
   i. 打开OBS推流工具。
   ii. 在左侧导航栏选择**推流**，**服务器**输入已生成推流地址，例如：`rtmp://example.aliyundoc.com/liveApp****/liveStream****?auth_key=1543302081-0-0-9c6e7c8190c10bdfb3c0****`。



> **说明**
>
> - 推流地址由推流域名、AppName、StreamName和鉴权串组成，您需要根据实际情况输入自己的推流地址。
> - 您可以根据实际情况输入**串流密钥**或者为空。

iii. 在左侧导航栏选择**输出**，**关键帧间隔**设置不超过3秒，**配置**设置为baseline。



iv. 在左侧导航栏选择**高级**，在**网络**配置区域选中**动态调整比特率以应对网络拥堵**。

2. 播RTS流。

方法一：使用移动端播RTS流。

i. 扫描并安装（或复制扫描后的地址至浏览器上进行下载并安装）**RTS低延时直播demo**。

Andriod iOS

ii. 打开**RTS低延时直播demo**，输入或扫描已开通低延时直播的播流地址URL。

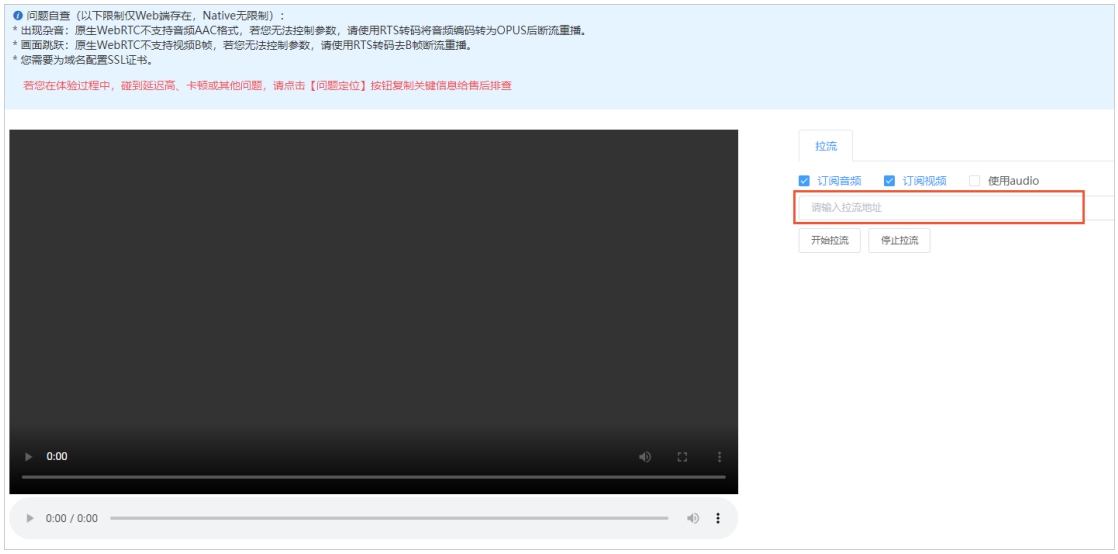
> **说明** iOS端在打开App前需要在**设置** > **通用** > **设备管理**中找到Taobao对应的信任描述，并选择信任。

iii. 单击**开始体验**，播放RTS直播流。

方法二：使用Web端播RTS流。

i. 进入RTS Web播放器。

ii. 输入已开通低延时直播的播流地址URL。



> **说明**
>
> - 由于浏览器针对WebRTC安全策略限制，RTS播流域名需要配置HTTPS证书，否则可能导致直播流无法正常播放。
> - Web RTS SDK不支持含B帧的视频和AAC编码的音频。如果您的视频包含B帧或音频为AAC编码，请先进行转码，播放地址为转码流地址。关于如何转码，请参见RTS转码。

iii. 单击**开始拉流**，播放RTS直播流。

5.SDK下载

通过阅读本文，你可以了解低延时直播相关SDK的下载方式。

Native RTS SDK

客户端	下载地址	配套阿里云播放器版本的限制
iOS	下载地址： V2.2.0 pod版本号： 2.2.0	支持V5.4.5.0或以上版本。
Android	下载地址： V2.2.0 Maven依赖： com.aliyun.rts.android:RtsSDK:2.2.0	
Mac	V2.2.0	不涉及。
Windows	V2.2.0	

> **说明**
>
> - 移动端（Android和iOS端）RTS SDK需要集成播放器使用，推荐使用阿里云播放器。更多信息，请参见Native RTS SDK简介。
> - Windows端阿里云播放器内置RTS SDK，无需单独集成。

Web RTS SDK

客户端	下载地址	npm地址
Web	V2.0.3	npm

OBS插件SDK

客户端	OBS插件SDK下载地址	Demo下载地址
Windows	V1.2.1	• x86版本 • x64版本
Mac		Mac版本

说明 此处Demo为OBS推流工具（已经集成OBS插件SDK），您可以下载此Demo使用OBS工具快速体验低延时推流。

6.发布日志

通过阅读本文，您可以了解低延时直播相关SDK的版本更新日志。

目录

- Native RTS SDK
- Web RTS SDK
- OBS插件SDK

Native RTS SDK

- 2022-03-03

客户端	下载地址	配套阿里云播放器版本的限制	发布日志
iOS	V2.2.0	支持V5.4.5.0或以上版本。	○ 优化首帧耗时。 ○ 拆分首帧耗时的埋点。 ○ 拆分起播失败的埋点。 ○ 修复Bug，提升SDK稳定性。
Android	V2.2.0		
Mac	V2.2.0	不涉及。	
Windows	V2.2.0		

- 2021-12-31

客户端	下载地址	配套阿里云播放器版本的限制	发布日志
iOS	V2.1.0	支持V5.4.2或以上版本。	○ 优化缓存策略。 ○ 优化日志埋点。 ○ 增加SFU、IP地址查询接口。
Android	V2.1.0		
Mac	V2.1.0	不涉及。	

- 2021-09-27

客户端	下载地址	配套阿里云播放器版本限制	发布日志
iOS	V1.9.0	支持V5.4.2或以上版本。	○ 优化视频卡顿。 ○ 优化首帧耗时。 ○ 起播阶段支持不同模式：快加速、慢加速、不加速。 ○ 支持AAC-HE。
Android	V1.9.0		
Mac	V1.9.0	不涉及。	
Windows	V1.9.0		

- 2021-07-30

客户端	下载地址	配套阿里云播放器版本限制	发布日志

客户端	下载地址	配套阿里云播放器版本限制	发布日志
iOS	V1.8.0	支持V5.4.2或以上版本。	○ 支持私有NAL的传输。 ○ 支持H.265编码格式推拉流。 ○ 支持统计全链路延迟。
Android	V1.8.0		
Mac	V1.8.0	不涉及。	
Windows	V1.8.0		

- 2021-04-23

客户端	下载地址	配套阿里云播放器版本限制	发布日志
iOS	V1.6.1	支持V5.3.0或以上版本。 **说明** 如果配套阿里云播放器版本为V5.3.0~V5.3.3，Android端阿里云播放器SDK中的AlivcArtc请使用V5.3.0p，iOS端阿里云播放器SDK中的artcSource请使用V5.3.0p。	○ 优化部分视频卡顿问题。 ○ 修复部分bug。
Android	V1.6.1		

- 2021-03-01

客户端	下载地址	配套阿里云播放器版本限制	发布日志
iOS	V1.6.0	支持V5.2.3或以上版本。	○ 优化弱网卡顿问题。 ○ 支持ARTC推流。
Android	V1.6.0		

- 2021-01-06

客户端	下载地址	配套阿里云播放器版本限制	发布日志
iOS	V1.5.0	支持V5.2.3或以上版本。	○ 降低延迟。 ○ 支持在播流过程中改变推流的分辨率。 ○ 优化抗卡顿。
Android	V1.5.0		

- 2020-11-20

客户端	下载地址	配套阿里云播放器版本限制	发布日志

客户端	下载地址	配套阿里云播放器版本限制	发布日志
iOS	V1.4.1	支持V5.2.2或以上版本。 **说明** 如果配套阿里云播放器版本为V5.2.2，Android端阿里云播放器SDK中的AlivcArtc请使用V5.2.2p，iOS端阿里云播放器SDK中的artcSource请使用V5.2.2p。	○ 裁剪包的大小。 ○ 支持Android x86架构。 ○ 完善回调事件。
Android	V1.4.1		

- 2020-11-05

客户端	下载地址	配套阿里云播放器版本限制	发布日志
iOS	○ 完整版本V1.4.0 ○ 裁剪版本V1.4.0	支持V5.2.1或以上版本。 **说明** 如果配套阿里云播放器版本为V5.2.1，Android端阿里云播放器SDK中的AlivcArtc请使用V5.2.1p，iOS端阿里云播放器SDK中的artcSource请使用V5.2.1p。	○ 优化特定场景卡顿问题。 ○ 支持纯音频、纯视频拉流。 ○ 支持外部调整缓存Buffer。 **说明** 裁剪版本可以有效减少包的体积，但是需要依赖外部的OpenSSL和cURL库，不能配合阿里云播放器SDK使用。
Android	○ 完整版本V1.4.0 ○ 裁剪版本V1.4.0		

- 2020-09-29

客户端	下载地址	配套阿里云播放器版本限制	发布日志
iOS	○ 完整版本V1.3.0 ○ 裁剪版本V1.3.0	支持V5.2.1或以上版本。	○ 独立发版。 ○ 增加回调实时状态。 ○ 整理错误码。 ○ 修复稳定性问题。 **说明** 裁剪版本可以有效减少包的体积，但是需要依赖外部的OpenSSL和cURL库，不能配合阿里云播放器SDK使用。
Android	○ 完整版本V1.3.0 ○ 裁剪版本V1.3.0		

Web RTS SDK

- 2022-03-23

下载地址	npm	发布日志
V2.0.3	npm	支持Edge浏览器。

- 2021-08-05

下载地址	npm	发布日志
V2.0.0	npm	○ 支持RTS推流。 ○ 更新接口。

- 2020-12-22

下载地址	npm	发布日志
V1.2.4	npm	○ 增加播放失败错误事件。 ○ 支持可配置播放时间超时。

- 2020-11-17

下载地址	npm	发布日志
V1.2.3	npm	支持参数设置拉纯音频流或纯视频流。

- 2020-10-10

下载地址	npm	发布日志
V1.2.2	npm	修复npm引入失败的问题。

- 2020-09-09

下载地址	npm	发布日志
V1.2.1	npm	○ 新增支持iOS端钉钉浏览器。 ○ 修复错误码抛出错误的问题。 ○ 增加onPlayEvent回调。

- 2020-08-05

下载地址	npm	发布日志
V1.1.0	npm	Web RTS SDK初版，具备RTS低延时直播拉流能力。

OBS插件SDK

- 2020-08-17

客户端	下载地址	Demo	发布日志
Mac	V1.2.1	OBS-studio	○ 优化视频编码器提升压缩率。 ○ 支持SEI信息发送。 ○ 支持视频图像质量提升。 ○ 修复其他稳定性问题。
Windows		○ OBS-Studio（x86） ○ OBS-Studio（x64）	

- 2020-07-07

客户端	下载地址	Demo	发布日志
Mac	V1.2.0	OBS-Studio	OBS插件SDK初版发布。
Windows		○ OBS-Studio（x86） ○ OBS-Studio（x64）	

7.Native RTS SDK

7.1. Native RTS SDK简介

实现Native端低延时直播功能，需要集成播放器引擎。通过阅读本文，您可以了解Native RTS SDK目录结构及各播放器通过集成Native RTS SDK实现低延时直播的方法。

实现低延时直播基本流程

播放器引擎分为阿里云播放器SDK、依赖FFmpeg的其他播放器引擎、不依赖FFmpeg的播放器引擎三类。各播放器通过集成Native RTS SDK实现低延时直播的基本流程如下所示：

播放器引擎	实现低延时直播的基本流程
阿里云播放器SDK	1. 阿里云播放器SDK集成Native RTS SDK作为插件。 2. 用户工程集成阿里云播放器SDK。 3. 调用阿里云播放器SDK接口实现低延时直播功能。 Mac和Windows端RTS SDK内置阿里云播放器，无需单独集成（同时Mac端没有单独的阿里云播放器）；Android和iOS端阿里云播放器SDK集成Native RTS SDK实现低延时直播，详情请参见阿里云播放器SDK集成Native RTS SDK实现说明。
依赖FFmpeg的其他播放器引擎	1. 依赖FFmpeg的其他播放器集成Native RTS SDK作为插件。 2. 用户工程集成其他播放器引擎。 3. 调用播放器引擎接口实现低延时直播功能。 通过依赖FFmpeg的其他播放器集成Native RTS SDK实现低延时直播，详情请参见依赖FFmpeg的其他播放器集成Native RTS SDK实现说明。 为了降低开发难度，对于依赖FFmpeg的其他播放器引擎，您也可以参见Native RTS SDK中的 `rtsdec.c` 文件将Native RTS SDK封装成了FFmpeg demuxer插件。
不依赖FFmpeg的播放器引擎	1. 不依赖FFmpeg的播放器集成Native RTS SDK作为插件。 2. 用户工程集成播放器引擎。 3. 调用播放器引擎的接口使用RTS服务。 不依赖FFmpeg的播放器集成Native RTS SDK适用于拥有完全自研的播放器引擎。您需要将Native RTS SDK动态库和头文件复制到工程中，参见Native RTS SDK中的*rtsdec.c*或*rtsenc.c*完成demuxer的开发。

Native RTS SDK目录说明

操作系统	目录	描述
	aar	Maven依赖库，配合阿里云播放器SDK使用。

操作系統	目录	描述
Android	• *lib* • *include*	需要用户自己开发artc demuxer，如果使用FFmpeg，可以基于*source/rtsdec.c*开发。
	jar	1.4.1版本之后开始提供，使用本地方式集成需要引入jar包。
iOS	*source*	无论是否使用阿里云播放器SDK都可以使用此依赖库。如果使用自有播放器，需要自己开发artc demuxer，如果自有播放器使用了FFmpeg，可以基于*source/rtsdec.c*开发。
	RtsSDK.framework	
Mac	*source*	• *source/rtsdec.c*：对接到FFmpeg拉流的示例。 • *source/rtsenc.c*：对接到FFmpeg推流的示例。
	lib	动态库文件libRtsSDK.dylib。
	include	RTS SDK头文件。
Windows	*source*	• *source/rtsdec.c*：对接到FFmpeg拉流的示例。 • *source/rtsenc.c*：对接到FFmpeg推流的示例。
	lib	动态库文件RtsSDK.dll、RtsSDK.lib。
	include	RTS SDK头文件。

7.2. 阿里云播放器SDK集成Native RTS SDK实现说明

7.2.1. Android端

通过阅读本文，您可以了解Android端阿里云播放器SDK集成Native RTS SDK实现低延时直播的方法。

操作步骤

1. 使用Maven方式添加阿里云播放器SDK和Native RTS SDK的依赖文件。

   在工程的*app/build.gradle*文件中添加阿里云播放器SDK和Native RTS SDK的依赖文件。

```
implementation 'com.aliyun.sdk.android:AlivcArtc:5.4.4.0'
implementation 'com.aliyun.rts.android:RtsSDK:2.1.0'
implementation 'com.aliyun.sdk.android:AliyunPlayer:5.4.4.0-full'
```

> 说明
>
> - `AlivcArtc` 和 `RtsSDK` 对应的依赖文件添加后，阿里云播放器SDK会自动将RTS SDK加载为插件。
> - 此处Maven依赖的版本仅供参考。获取最新的版本，阿里云播放器SDK请参见Android播放器SDK，低延时直播SDK请参见SDK下载。
> - 集成阿里云播放器SDK和Native RTS SDK时需要满足对应的配套关系，详情请参见发布日志。

2. 添加Maven仓库地址。

   在根目录的*build.gradle*中添加Maven仓库地址。

```
//MPChart maven仓库地址，如果不需要曲线图，可以删除
maven { url 'https://jitpack.io' }
//阿里云相关SDK（阿里云播放器、RTS SDK）Maven仓库地址
maven { url 'http://maven.aliyun.com/nexus/content/repositories/releases' }
```

3. 调用阿里云播放器SDK接口实现低延时直播功能。
   - 创建播放器

     通过AliPlayerFactory类可创建两种播放器：AliPlayer和AliListPlayer。单个播放功能使用AliPlayer，创建方法如下所示：

```
AliPlayer aliyunVodPlayer;
.....
aliyunVodPlayer = AliPlayerFactory.createAliPlayer(getApplicationContext());
```

   - 设置播放器监听事件

     播放器提供了多种监听事件，例如：onPrepared、onCompletion等。设置方法如下所示：

```
aliyunVodPlayer.setOnErrorListener(new IPlayer.OnErrorListener() {
    @Override
    public void onError(ErrorInfo errorInfo) {
        //出错事件
    }
});
aliyunVodPlayer.setOnPreparedListener(new IPlayer.OnPreparedListener() {
    @Override
    public void onPrepared() {
        //准备成功事件
    }
});
aliyunVodPlayer.setOnVideoSizeChangedListener(new IPlayer.OnVideoSizeChangedListener(
) {
    @Override
    public void onVideoSizeChanged(int width, int height) {
        //视频分辨率变化回调
    }
});
aliyunVodPlayer.setOnRenderingStartListener(new IPlayer.OnRenderingStartListener() {
    @Override
    public void onRenderingStart() {
```

```
        //首帧渲染显示事件
    }
});
aliyunVodPlayer.setOnInfoListener(new IPlayer.OnInfoListener() {
    @Override
    public void onInfo(int type, long extra) {
        //其他信息的事件，type包括循环播放开始、缓冲位置、当前播放位置、自动播放开始等
    }
});
aliyunVodPlayer.setOnLoadingStatusListener(new IPlayer.OnLoadingStatusListener() {
    @Override
    public void onLoadingBegin() {
        //缓冲开始
    }
    @Override
    public void onLoadingProgress(int percent, float kbps) {
        //缓冲进度
    }
    @Override
    public void onLoadingEnd() {
        //缓冲结束
    }
});
aliyunVodPlayer.setOnSubtitleDisplayListener(new IPlayer.OnSubtitleDisplayListener()
{
    @Override
    public void onSubtitleShow(long id, String data) {
        //显示字幕
    }
    @Override
    public void onSubtitleHide(long id) {
        //隐藏字幕
    }
});
aliyunVodPlayer.setOnTrackChangedListener(new IPlayer.OnTrackChangedListener() {
    @Override
    public void onChangedSuccess(TrackInfo trackInfo) {
        //切换音视频流或者清晰度成功
    }
    @Override
    public void onChangedFail(TrackInfo trackInfo, ErrorInfo errorInfo) {
        //切换音视频流或者清晰度失败
    }
});
aliyunVodPlayer.setOnStateChangedListener(new IPlayer.OnStateChangedListener() {
    @Override
    public void onStateChanged(int newState) {
        //播放器状态改变事件
    }
});
aliyunVodPlayer.setOnSnapShotListener(new IPlayer.OnSnapShotListener() {
    @Override
    public void onSnapShot(Bitmap bm, int with, int height) {
        //截图事件
```

```
    }
});
```

- 创建DataSource

  播放器支持4种播放源：VidSts、VidAuth、VidMps和UrlSource。其中UrlSource是直接URL播放，需要将URL设置为 `artc://` 协议，才可以使用RTS服务。

```
UrlSource urlSource = new UrlSource();
urlSource.setUri("artc://<播流地址>");
aliyunVodPlayer.setDataSource(urlSource);
```

- 设置显示的View

  如果源有画面，需要设置显示的View（支持SurfaceView和TextureView）到播放器中，用来显示画面。此处以SurfaceView举例说明，如下所示：

```
surfaceView = (SurfaceView) findViewById(R.id.playview);
surfaceView.getHolder().addCallback(new SurfaceHolder.Callback() {
    @Override
    public void surfaceCreated(SurfaceHolder holder) {
        aliyunVodPlayer.setDisplay(holder);
    }
    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int width, int heigh
t) {
        aliyunVodPlayer.surfaceChanged();
    }
    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
        aliyunVodPlayer.setDisplay(null);
    }
});
```

- 播放控制

  您需要自行创建播放器的控制按钮，在按钮事件里面实现播放器控制接口。基本控制功能有播放、暂停、停止、拖动（seek），其中seek功能仅对点播有效；直播使用暂停功能时会使画面停留在当前画面，使用恢复后会开始播放当前画面。使用示例如下所示：

```
//准备，在准备完成的回调接口中，调用start()可以开始播放
aliyunVodPlayer.prepare();
//开始播放
aliyunVodPlayer.start();
//暂停播放
aliyunVodPlayer.pause();
//停止播放
aliyunVodPlayer.stop();
//跳转到（不精准）
aliyunVodPlayer.seekTo(long position);
//重置
aliyunVodPlayer.reset();
//释放（释放后播放器将不可再被使用）
aliyunVodPlayer.release();
```

- 播放参数配置

配置播放参数，提升低延迟直播效果。

```
//获取配置
PlayerConfig config = mAliyunVodPlayer.getConfig();
//设置最大延迟为1秒，延迟控制由ARTC控制
config.mMaxDelayTime = 1000;
//设置播放器启播缓存为10毫秒，数据控制由ARTC控制。
config.mHighBufferDuration = 10;
config.mStartBufferDuration = 10;
....//其他设置
//给播放器加载配置
mAliyunVodPlayer.setConfig(config);
```

- 加载库

  根据实际需求，在适当的Activity内加入 `System.loadLibrary("RtsSDK");` 。

```
static {
    System.loadLibrary("RtsSDK");
}
```

更多阿里云播放器SDK功能，请参见进阶功能和接口说明。

7.2.2. iOS端

通过阅读本文，您可以了解iOS端阿里云播放器SDK集成Native RTS SDK实现低延时直播的方法。

前提条件

环境中已安装CocoaPods。

操作步骤

1. 使用pod方式添加阿里云播放器SDK和Native RTS SDK的依赖文件。
   i. 打开终端窗口。
   ii. 进入项目所在路径，创建Podfile文件。

   ```
   pop init
   ```

iii. 编辑Podfile文件，添加最新版本的依赖。

```
pod 'RtsSDK'
pod 'AliPlayerSDK_iOS'
pod 'AliPlayerSDK_iOS_ARTC'
```

> 说明
>
> - `AliPlayerSDK_iOS_ARTC` 和 `RtsSDK` 对应的依赖文件添加后，阿里云播放器SDK会自动将RTS SDK加载为插件。
> - 如果您需要添加指定版本的依赖，请自行补充版本号（例如：`pod 'RtsSDK','2.1.0'`）。获取最新的版本，阿里云播放器SDK请参见iOS播放器SDK，低延时直播SDK请参见SDK下载。
> - 集成阿里云播放器SDK和Native RTS SDK时需要满足对应的配套关系，详情请参见发布日志。

iv. 安装SDK。执行之后，会生成*.xcworkspace文件，表示SDK集成完毕。

```
pod install
```

2. 调用阿里云播放器SDK接口实现低延时直播功能。

- 初始化Aliplayer

```
- (AliPlayer *)aliPlayer{
    if (!_aliPlayer) {
        _aliPlayer = [[AliPlayer alloc] init];
        _aliPlayer.scalingMode =  AVP_SCALINGMODE_SCALEASPECTFIT;
        _aliPlayer.rate = 1;
    //如需实现AVPDelegate代理，添加此行
        _aliPlayer.delegate = self;
    //设置播放的视图，将您的播放视图赋值给aliplayer
        _aliPlayer.playerView = self.basePlayerView.playerView;
        _aliPlayer.autoPlay = YES;
    }
    return _aliPlayer;
}
```

- 设置播放URL

```
AVPUrlSource *source = [[AVPUrlSource alloc] urlWithString:_url];
[self.aliPlayer setUrlSource:source];
```

- 播放参数配置

  配置播放参数，提升低延迟直播效果。播放器参数配置需要在调用 `prepare` 之前才能生效。

```
AVPConfig *config = self.aliPlayer.getConfig;
//直播最大延迟
[config setMaxDelayTime:1000];
//卡顿恢复时长
[config setHighBufferDuration:10];
//起播最大延迟
[config setStartBufferDuration:10];
//重连次数
[config setNetworkRetryCount:2];
//重连时长
[config setNetworkTimeout:15000];
 [_aliPlayer setConfig:config];
//默认为硬解，如播放器在准备过程中发现硬解失败，会自动切换到软解
 _aliPlayer.enableHardwareDecoder = YES;
```

- 打开日志

```
[AliPlayer setEnableLog:YES];
[AliPlayer setLogCallbackInfo:LOG_LEVEL_DEBUG callbackBlock:nil];
```

- 播放控制

```
[self.aliPlayer prepare];
[self.aliPlayer stop];
[self.aliPlayer destroy];
[self.aliPlayer reload];
```

更多阿里云播放器SDK功能，请参见进阶功能和接口说明。

7.3. 依赖FFmpeg的其他播放器集成Native RTS SDK实现说明

7.3.1. Android端

通过阅读本文，您可以了解Android端依赖FFmpeg的其他播放器（本文以ijkplayer tag k0.8.8为例）集成Native RTS SDK实现低延时直播的方法。

前提条件

您已完成ijkplayer源码的编译。具体操作，请参见ijkplayer中README.md介绍。

操作步骤

1. 下载并解压ijkplayer源码。下载地址，请参见ijkplayer。
2. 下载并解压Native RTS SDK。下载地址，请参见SDK下载。
3. ijkplayer集成Native RTS SDK作为插件。

   ijkplayer集成Native RTS SDK有以下两种方式：

集成方式	描述	优点	缺点
拓展FFmpeg	拓展ijk中FFmpeg的demuxer。	使用更加简单，不需要根据ARTC的URL做逻辑区分。	需要重新编译FFmpeg库。
拓展ijk	ijkplayer中添AVInputFormat。	不需要编译FFmpeg。	ff_ffplay.c中需要添加部分逻辑代码。

拓展FFmpeg

i. 修改*ijkplayer/init-android.sh*中的`pull_fork`，只保留`armv7a`和`arm64`，并执行./init-android.sh运行该脚本。

```
pull_fork "armv7a"
pull_fork "arm64"
```

> **说明** 目前Native RTS SDK只提供armv7a和arm64的动态库，因此只需保留`armv7a`和`arm64`的描述。

ii. 修改*ijkplayer/android/contrib/compile-ffmpeg.sh*中的`FF_ACT_ARCHS_64`，只保留`armv7a`和`arm64`。

```
FF_ACT_ARCHS_64="armv7a arm64"
```

iii. 复制Native RTS SDK中的rtsdec.c文件至*ijkplayer/android/contrib/ffmpeg-arm64/libavformat*和*ijkplayer/android/contrib/ffmpeg-armv7a/libavformat*目录下。

iv. 修改Makefile文件并编译rtsdec.c文件。

修改*ijkplayer/android/contrib/ffmpeg-arm64/libavformat/Makefile*和*ijkplayer/android/contrib/ffmpeg-armv7a/libavformat/Makefile*，并编译rtsdec.c文件。

```
--- a/libavformat/Makefile
+++ b/libavformat/Makefile
@@ -24,6 +24,7 @@ OBJS = allformats.o                     \
         sdp.o                          \
         url.o                          \
         utils.o                        \
+        rtsdec.o                               \
```

v. 修改allformats.c文件。

修改*ijkplayer/android/contrib/ffmpeg-arm64/libavformat/allformats.c*和*ijkplayer/android/contrib/ffmpeg-armv7a/libavformat/allformats.c*，默认支持ARTC协议。

```
diff --git a/libavformat/allformats.c b/libavformat/allformats.c
index 405ddb5ad9f..a136a7836b0 100644
--- a/libavformat/allformats.c
+++ b/libavformat/allformats.c
@@ -385,6 +385,8 @@ static void register_all(void)
     REGISTER_DEMUXER (LIBGME,           libgme);
     REGISTER_DEMUXER (LIBMODPLUG,       libmodplug);
     REGISTER_DEMUXER (LIBOPENMPT,       libopenmpt);
+    extern AVInputFormat ff_rtc_demuxer;
+    av_register_input_format(&ff_rtc_demuxer);
 }
```

```
    extern AVInputFormat ff_rtc_demuxer;
    av_register_input_format(&ff_rtc_demuxer);
```

vi. 修改FFmpeg编译脚本*config/module-lite.sh*，使其支持PCM解码（Native RTS SDK输出为PCM数据）。

```
# aliyun rts
export COMMON_FF_CFG_FLAGS="$COMMON_FF_CFG_FLAGS --enable-decoder=pcm_s16be_planar"
export COMMON_FF_CFG_FLAGS="$COMMON_FF_CFG_FLAGS --enable-decoder=pcm_s16le"
export COMMON_FF_CFG_FLAGS="$COMMON_FF_CFG_FLAGS --enable-decoder=pcm_s16le_planar"
```

vii. 编译。

添加Android NDK环境变量，并在*ijkplayer/android/contrib*目录下执行**./compile-ffmpeg.sh all**运行该脚本。编译完成之后，检查并确保*ijkplayer/android/contrib/build*目录下有对应的FFmpeg编译输出文件。

viii. 修改*ijkplayer/Android/compile-ijk.sh*中的 `ACT_ABI_64` ，只保留 `armv7a` 和 `arm64` 。

```
ACT_ABI_64="armv7a arm64"
```

ix. 加入Native RTS SDK的动态库。

复制Native RTS SDK的动态库至对应的*ijkplayer/android/contrib/build/ffmpeg-arm64/output*和*ijkplayer/android/contrib/build/ffmpeg-armv7a/output*目录中。

复制Native RTS SDK的头文件rts_api.h和rts_messages.h分别至*ijkplayer/android/contrib/build/ffmpeg-arm64/output/include*和*ijkplayer/android/contrib/build/ffmpeg-armv7a/output/include*目录中。

x. 引入Native RTS SDK的动态库。

修改ijkplayer/android/ijkplayer/ijkplayer-armv7a/src/main/jni/ffmpeg/Android.mk和ijkplayer/android/ijkplayer/ijkplayer-arm64/src/main/jni/ffmpeg/Android.mk文件。

```
diff --git a/android/ijkplayer/ijkplayer-armv7a/src/main/jni/ffmpeg/Android.mk b/android/ijkplayer/ijkplayer-armv7a/src/main/jni/ffmpeg/Android.mk
index ec33908f..f99654f3 100644
--- a/android/ijkplayer/ijkplayer-armv7a/src/main/jni/ffmpeg/Android.mk
+++ b/android/ijkplayer/ijkplayer-armv7a/src/main/jni/ffmpeg/Android.mk
@@ -3,4 +3,9 @@ LOCAL_PATH := $(call my-dir)
 include $(CLEAR_VARS)
 LOCAL_MODULE := ijkffmpeg
 LOCAL_SRC_FILES := $(MY_APP_FFMPEG_OUTPUT_PATH)/libijkffmpeg.so
+include $(PREBUILT_SHARED_LIBRARY)
+
+include $(CLEAR_VARS)
+LOCAL_MODULE := rtssdk
+LOCAL_SRC_FILES := $(MY_APP_FFMPEG_OUTPUT_PATH)/libRtsSDK.so
 include $(PREBUILT_SHARED_LIBRARY)
```

```
include $(CLEAR_VARS)
LOCAL_MODULE := rtssdk
LOCAL_SRC_FILES := $(MY_APP_FFMPEG_OUTPUT_PATH)/libRtsSDK.so
include $(PREBUILT_SHARED_LIBRARY)
```

xi. 修改*ijkplayer/ijkmedia/ijkplayer/Android.mk*文件，使ijkplayer依赖Native RTS SDK动态库。

```
--- a/ijkmedia/ijkplayer/Android.mk
+++ b/ijkmedia/ijkplayer/Android.mk
@@ -75,7 +75,7 @@ LOCAL_SRC_FILES += ijkavutil/ijktree.c
 LOCAL_SRC_FILES += ijkavutil/ijkfifo.c
 LOCAL_SRC_FILES += ijkavutil/ijkstl.cpp

-LOCAL_SHARED_LIBRARIES := ijkffmpeg ijksdl
+LOCAL_SHARED_LIBRARIES := ijkffmpeg ijksdl rtssdk
 LOCAL_STATIC_LIBRARIES := android-ndk-profiler ijksoundtouch

 LOCAL_MODULE := ijkplayer
```

xii. 在ff_ffplay.c中添加RTS逻辑。

修改*ijkplayer/ijkmedia/ijkplayer/ff_ffplay.c*文件，设置ARTC的AVInputFormat函数指针。

```
diff --git a/ijkmedia/ijkplayer/ff_ffplay.c b/ijkmedia/ijkplayer/ff_ffplay.c
index 52aba31f..036fde6c 100755
--- a/ijkmedia/ijkplayer/ff_ffplay.c
+++ b/ijkmedia/ijkplayer/ff_ffplay.c
@@ -3868,6 +3868,19 @@ void ffp_global_init()
         return;

     ALOGD("ijkmediaplayer version : %s", ijkmp_version());
+
+    extern AVInputFormat ff_rtc_demuxer;
+    extern int artc_reload(AVFormatContext *ctx);
+    extern void av_set_rts_demuxer_funcs(const struct rts_glue_funcs *funcs);
+    extern void artc_set_rts_param(char* key, char* value);
+    extern long long artc_get_state(AVFormatContext *ctx, int key);
+
+    int version = 2;
+    const struct rts_glue_funcs* rts_funcs = get_rts_funcs(version);
+    // set to ffmpeg plugin
+    av_set_rts_demuxer_funcs(rts_funcs);
+    artc_set_rts_param((char*)"AutoReconnect", (char*)"false");
+
     /* register all codecs, demux and protocols */
     avcodec_register_all();
 #if CONFIG_AVDEVICE
```

```
    extern AVInputFormat ff_rtc_demuxer;
    extern int artc_reload(AVFormatContext *ctx);
    extern void av_set_rts_demuxer_funcs(const struct rts_glue_funcs *funcs);
    extern void artc_set_rts_param(char* key, char* value);
    extern long long artc_get_state(AVFormatContext *ctx, int key);
    int version = 2;
    const struct rts_glue_funcs* rts_funcs = get_rts_funcs(version);
    // set to ffmpeg plugin
    av_set_rts_demuxer_funcs(rts_funcs);
    artc_set_rts_param((char*)"AutoReconnect", (char*)"false");
```

xiii. 编译。

在*ijkplayer/android*目录下执行**./compile-ijk.sh all**运行该脚本。

拓展ijk

i. 修改并运行FFmpeg编译脚本*/config/module-lite.sh*。

修改FFmpeg编译脚本*/config/module-lite.sh*，使其支持PCM解码（Native RTS SDK输出为PCM数据），并执行**./module-lite.sh**编译FFmpeg。

```
# aliyun rts
export COMMON_FF_CFG_FLAGS="$COMMON_FF_CFG_FLAGS --enable-decoder=pcm_s16be_planar"
export COMMON_FF_CFG_FLAGS="$COMMON_FF_CFG_FLAGS --enable-decoder=pcm_s16le"
export COMMON_FF_CFG_FLAGS="$COMMON_FF_CFG_FLAGS --enable-decoder=pcm_s16le_planar"
```

ii. 加入Native RTS SDK的动态库。

复制Native RTS SDK的动态库至对应的*ijkplayer/android/contrib/build/ffmpeg-arm64/output*和*ijkplayer/android/contrib/build/ffmpeg-armv7a/output*目录中。

复制Native RTS SDK的头文件rts_api.h和rts_messages.h分别至*ijkplayer/android/contrib/build/ffmpeg-arm64/output/include*和*ijkplayer/android/contrib/build/ffmpeg-armv7a/output/include*目录中。

iii. 引入Native RTS SDK的动态库。

修改ijkplayer/android/ijkplayer/ijkplayer-armv7a/src/main/jni/ffmpeg/Android.mk和ijkplayer/android/ijkplayer/ijkplayer-arm64/src/main/jni/ffmpeg/Android.mk文件。

```
diff --git a/android/ijkplayer/ijkplayer-armv7a/src/main/jni/ffmpeg/Android.mk b/android/ijkplayer/ijkplayer-armv7a/src/main/jni/ffmpeg/Android.mk
index ec33908f..f99654f3 100644
--- a/android/ijkplayer/ijkplayer-armv7a/src/main/jni/ffmpeg/Android.mk
+++ b/android/ijkplayer/ijkplayer-armv7a/src/main/jni/ffmpeg/Android.mk
@@ -3,4 +3,9 @@ LOCAL_PATH := $(call my-dir)
 include $(CLEAR_VARS)
 LOCAL_MODULE := ijkffmpeg
 LOCAL_SRC_FILES := $(MY_APP_FFMPEG_OUTPUT_PATH)/libijkffmpeg.so
+include $(PREBUILT_SHARED_LIBRARY)
+
+include $(CLEAR_VARS)
+LOCAL_MODULE := rtssdk
+LOCAL_SRC_FILES := $(MY_APP_FFMPEG_OUTPUT_PATH)/libRtsSDK.so
 include $(PREBUILT_SHARED_LIBRARY)
```

```
include $(CLEAR_VARS)
LOCAL_MODULE := rtssdk
LOCAL_SRC_FILES := $(MY_APP_FFMPEG_OUTPUT_PATH)/libRtsSDK.so
include $(PREBUILT_SHARED_LIBRARY)
```

iv. 修改*ijkplayer/ijkmedia/ijkplayer/Android.mk*文件，使ijkplayer依赖Native RTS SDK动态库。

```
--- a/ijkmedia/ijkplayer/Android.mk
+++ b/ijkmedia/ijkplayer/Android.mk
@@ -75,7 +75,7 @@ LOCAL_SRC_FILES += ijkavutil/ijktree.c
 LOCAL_SRC_FILES += ijkavutil/ijkfifo.c
 LOCAL_SRC_FILES += ijkavutil/ijkstl.cpp

-LOCAL_SHARED_LIBRARIES := ijkffmpeg ijksdl
+LOCAL_SHARED_LIBRARIES := ijkffmpeg ijksdl rtssdk
 LOCAL_STATIC_LIBRARIES := android-ndk-profiler ijksoundtouch

 LOCAL_MODULE := ijkplayer
```

v. 在ff_ffplay.c中添加RTS逻辑。

修改*ijkplayer/ijkmedia/ijkplayer/ff_ffplay.c*文件，设置ARTC的AVInputFormat函数指针。

```
diff --git a/ijkmedia/ijkplayer/ff_ffplay.c b/ijkmedia/ijkplayer/ff_ffplay.c
index 52aba31f..2beac440 100755
--- a/ijkmedia/ijkplayer/ff_ffplay.c
+++ b/ijkmedia/ijkplayer/ff_ffplay.c
@@ -3114,8 +3114,24 @@ static int read_thread(void *arg)
         av_dict_set_int(&ffp->format_opts, "skip-calc-frame-rate", ffp->skip_calc_frame_rate, 0);
     }

-    if (ffp->iformat_name)
-        is->iformat = av_find_input_format(ffp->iformat_name);
+    if(strncmp(is->filename, "artc://", 7) == 0) {
+        extern AVInputFormat ff_rtc_demuxer;
+        extern int artc_reload(AVFormatContext *ctx);
+        extern void av_set_rts_demuxer_funcs(const struct rts_glue_funcs *funcs);
+        extern void artc_set_rts_param(char* key, char* value);
+        extern long long artc_get_state(AVFormatContext *ctx, int key);
+
+        int version = 2;
+        const struct rts_glue_funcs* rts_funcs = get_rts_funcs(version);
+        // set to ffmpeg plugin
+        av_set_rts_demuxer_funcs(rts_funcs);
+        artc_set_rts_param((char*)"AutoReconnect", (char*)"false");
+        is->iformat = &ff_rtc_demuxer;
+    }
+    else {
+        if(ffp->iformat_name)
+            is->iformat = av_find_input_format(ffp->iformat_name);
+    }
     err = avformat_open_input(&ic, is->filename, is->iformat, &ffp->format_opts);
     if (err < 0) {
```

```
if(strncmp(is->filename, "artc://", 7) == 0) {
        extern AVInputFormat ff_rtc_demuxer;
        extern int artc_reload(AVFormatContext *ctx);
        extern void av_set_rts_demuxer_funcs(const struct rts_glue_funcs *funcs);
        extern void artc_set_rts_param(char* key, char* value);
        extern long long artc_get_state(AVFormatContext *ctx, int key);
        int version = 2;
        const struct rts_glue_funcs* rts_funcs = get_rts_funcs(version);
        // set to ffmpeg plugin
        av_set_rts_demuxer_funcs(rts_funcs);
        artc_set_rts_param((char*)"AutoReconnect", (char*)"false");
        is->iformat = &ff_rtc_demuxer;
    }
    else {
        if(ffp->iformat_name)
            is->iformat = av_find_input_format(ffp->iformat_name);
    }
```

vi. 复制Native RTS SDK中的rtsdec.c文件至*ijkplayer/ijkmedia/ijkplayer*目录下，并修改该目录中*android.mk*文件添加rtsdec.c文件的编译描述。

```
LOCAL_SRC_FILES += rtsdec.c
```

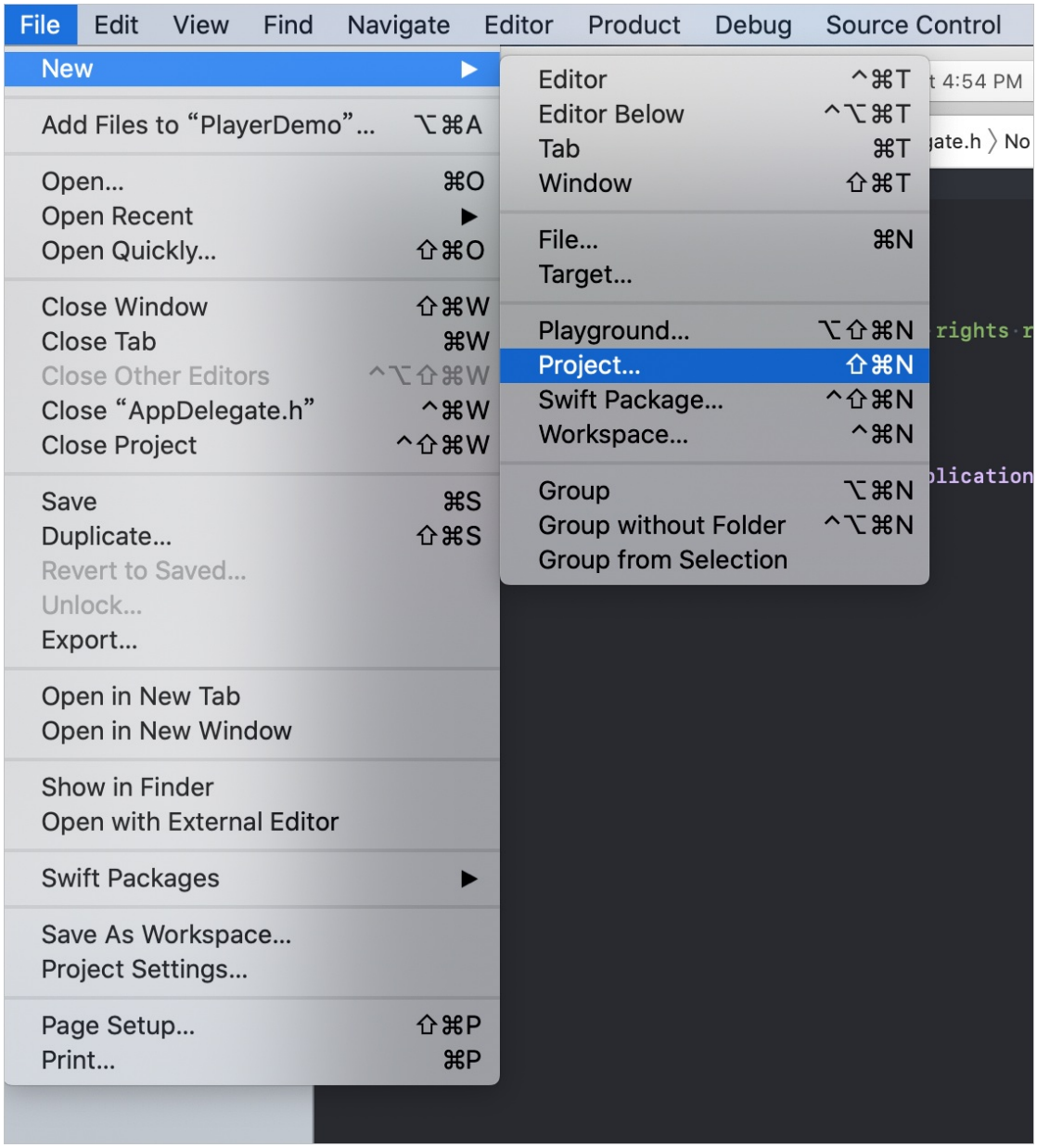
4. 用户工程集成ijkplayer。

i. 复制Native RTS SDK中的*RtsNetSDK.jar*文件至工程中。

ii. 在使用RTS播放的Activity内引入Native RTS SDK动态库。

```
static {
    System.loadLibrary("RtsSDK");
}
```

iii. 导入ijkplayer-arm64、ijkplayer-armv7a、ijkplayer-java模块。



5. 调用ijkplayer接口实现低延时直播功能。
   - 创建ijkplayer

     ```
     mIjkPlayer = new IjkMediaPlayer();
     ```

   - 设置回调

```
mIjkPlayer.setOnPreparedListener(new IMediaPlayer.OnPreparedListener() {
    @Override
    public void onPrepared(IMediaPlayer iMediaPlayer) {
    }
});
mIjkPlayer.setOnInfoListener(new IMediaPlayer.OnInfoListener() {
    @Override
    public boolean onInfo(IMediaPlayer iMediaPlayer, int arg1, int arg2) {
});
mIjkPlayer.setOnVideoSizeChangedListener(new IMediaPlayer.OnVideoSizeChangedListener(
) {
    @Override
    public void onVideoSizeChanged(IMediaPlayer iMediaPlayer, int width, int height,
int sarNum, int sarDen) {
    }
});
mIjkPlayer.setOnErrorListener(new IMediaPlayer.OnErrorListener() {
    @Override
    public boolean onError(IMediaPlayer iMediaPlayer, int framework_err, int impl_err
) {
        return false;
    }
});
```

- 设置显示窗口

```
mIjkPlayer.setSurface(surface);
```

- 设置播放源

```
mIjkPlayer.setDataSource("artc://<播流地址>");
```

- 状态控制

```
public void prepare() {
    if(mIjkPlayer != null){
        mIjkPlayer.prepareAsync();
    }
}
@Override
public void start() {
    if(mIjkPlayer != null){
        mIjkPlayer.start();
    }
}
public void pause() {
    if(mIjkPlayer != null){
        mIjkPlayer.pause();
    }
}
public void stop() {
    if(mIjkPlayer != null){
        mIjkPlayer.stop();
    }
}
public void release() {
    if(mIjkPlayer != null){
        mIjkPlayer.release();
    }
}
```

7.3.2. iOS端

通过阅读本文，您可以了解iOS端依赖FFmpeg的其他播放器（本文以ijkplayer tag k0.8.8为例）集成Native RTS SDK实现低延时直播的方法。

前提条件

您已完成ijkplayer源码的编译。具体操作，请参见ijkplayer中README.md介绍。

操作步骤

1. 下载并解压ijkplayer源码。下载地址，请参见ijkplayer。
2. 下载并解压Native RTS SDK。下载地址，请参见SDK下载。
3. ijkplayer集成Native RTS SDK作为插件。

   ijkplayer集成Native RTS SDK有以下两种方式：

集成方式	描述	优点	缺点
拓展FFmpeg	拓展ijk中FFmpeg的demuxer。	使用更加简单，不需要根据ARTC的URL做逻辑区分。	需要重新编译FFmpeg库。
拓展ijk	ijkplayer中添AVInputFormat。	不需要编译FFmpeg。	ff_ffplay.c中需要添加部分逻辑代码。

   拓展FFmpeg

i. 在ijkplayer根目录下，执行**./init-ios.sh**进行初始化。

ii. 复制Native RTS SDK中的rtsdec.c文件至*ijkplayer/ios/ffmpeg-$arch/libavformat*目录。

iii. 修改Makefile文件并编译rtsdec.c文件。

修改*ijkplayer/ios/ffmpeg-$arch/libavformat/Makefile*，并编辑rtsdec.c文件。

```
--- a/libavformat/Makefile
+++ b/libavformat/Makefile
@@ -24,6 +24,7 @@ OBJS = allformats.o          \
         sdp.o                \
         url.o                \
         utils.o              \
+        rtsdec.o                    \
```

iv. 修改allformats.c文件。

修改*ijkplayer/ios/ffmpeg-$arch/libavformat/allformats.c*，默认支持ARTC协议。

```
diff --git a/libavformat/allformats.c b/libavformat/allformats.c
index 405ddb5ad9f..a136a7836b0 100644
--- a/libavformat/allformats.c
+++ b/libavformat/allformats.c
@@ -385,6 +385,8 @@ static void register_all(void)
     REGISTER_DEMUXER (LIBGME,           libgme);
     REGISTER_DEMUXER (LIBMODPLUG,       libmodplug);
     REGISTER_DEMUXER (LIBOPENMPT,       libopenmpt);
+    extern AVInputFormat ff_rtc_demuxer;
+    av_register_input_format(&ff_rtc_demuxer);
 }
```

```
    extern AVInputFormat ff_rtc_demuxer;
    av_register_input_format(&ff_rtc_demuxer);
```

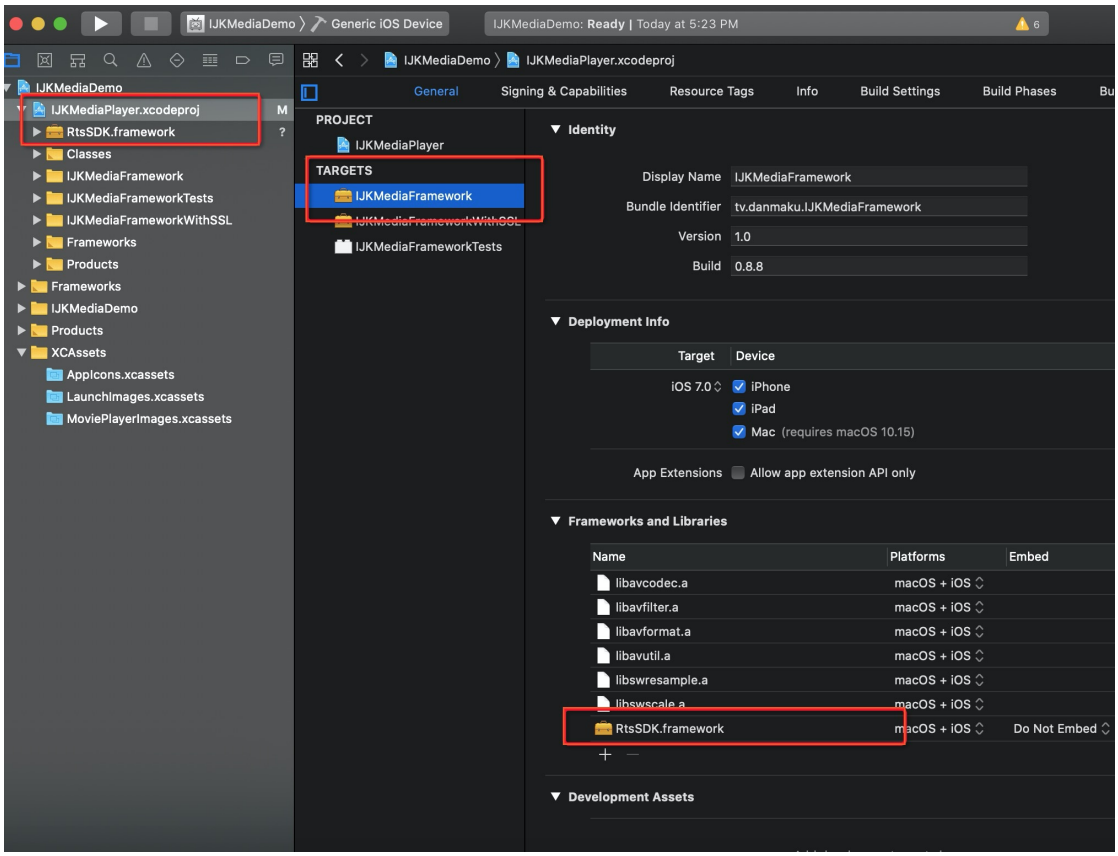
v. 编译。

在*ijkplayer/ios*目录下，执行**./compile-ffmpeg.sh all**运行该脚本。编译完成之后，检查并确保*ijkplayer/ios/build/universal*目录下有对应的FFmpeg编译输出文件。

vi. 复制Native RTS SDK中的*RtsSDK.framework*文件至*ijkplayer/ios/IJKMediaDemo/IJKMediaDemo*目录中。

vii. 使用Xcode打开*ios/IJKMediaDemo/IJKMediaDemo.xcodeproj*。

viii. 添加RtsSDK.framework的依赖。



ix. 在ff_ffplay.c中添加RTS逻辑。

修改*ijkplayer/ijkmedia/ijkplayer/ff_ffplay.c*文件，设置ARTC的AVInputFormat函数指针。

```
diff --git a/ijkmedia/ijkplayer/ff_ffplay.c b/ijkmedia/ijkplayer/ff_ffplay.c
index 52aba31f..036fde6c 100755
--- a/ijkmedia/ijkplayer/ff_ffplay.c
+++ b/ijkmedia/ijkplayer/ff_ffplay.c
@@ -3868,6 +3868,19 @@ void ffp_global_init()
         return;

     ALOGD("ijkmediaplayer version : %s", ijkmp_version());
+
+    extern AVInputFormat ff_rtc_demuxer;
+    extern int artc_reload(AVFormatContext *ctx);
+    extern void av_set_rts_demuxer_funcs(const struct rts_glue_funcs *funcs);
+    extern void artc_set_rts_param(char* key, char* value);
+    extern long long artc_get_state(AVFormatContext *ctx, int key);
+
+    int version = 2;
+    const struct rts_glue_funcs* rts_funcs = get_rts_funcs(version);
+    // set to ffmpeg plugin
+    av_set_rts_demuxer_funcs(rts_funcs);
+    artc_set_rts_param((char*)"AutoReconnect", (char*)"false");
+
     /* register all codecs, demux and protocols */
     avcodec_register_all();
 #if CONFIG_AVDEVICE
```

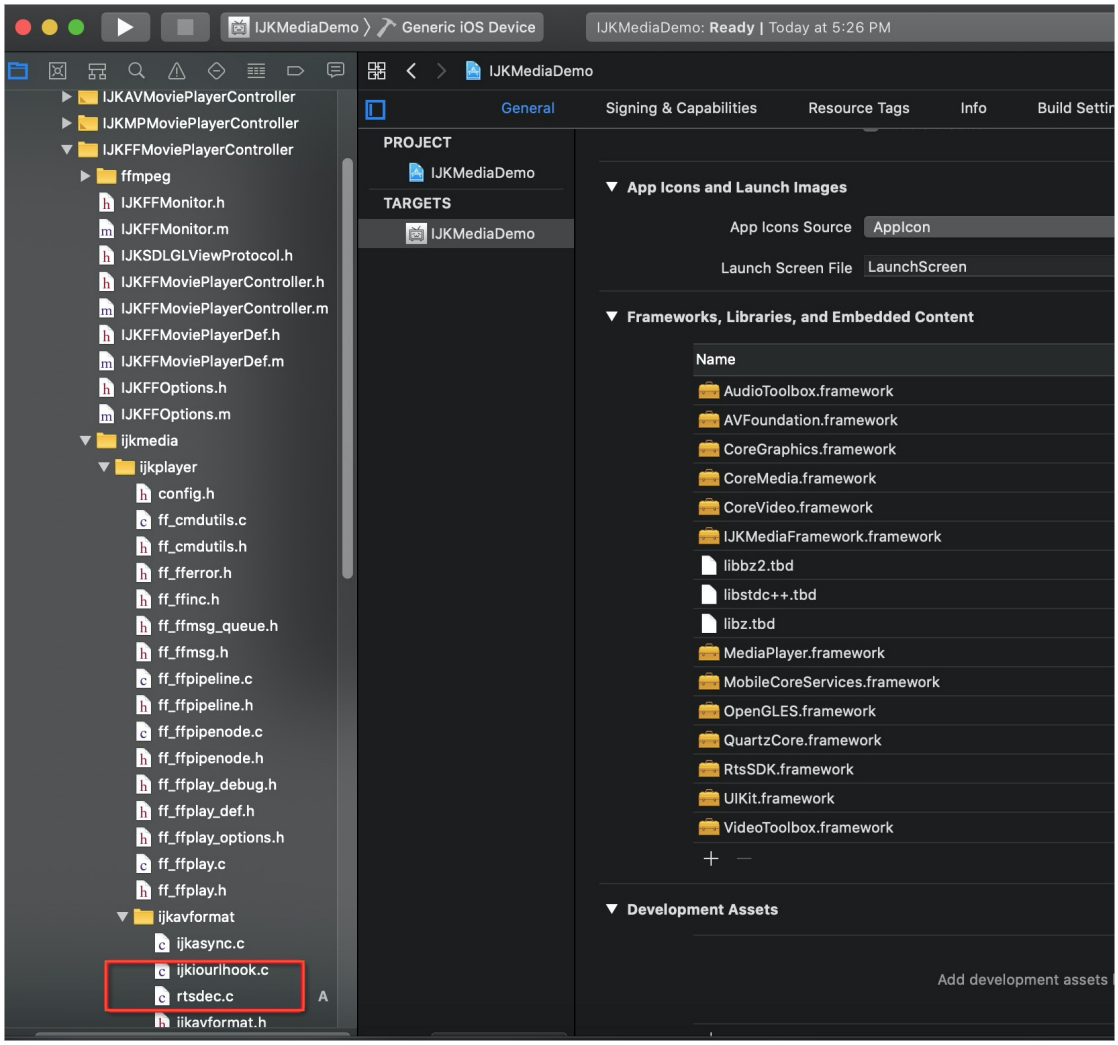
```
    extern AVInputFormat ff_rtc_demuxer;
    extern int artc_reload(AVFormatContext *ctx);
    extern void av_set_rts_demuxer_funcs(const struct rts_glue_funcs *funcs);
    extern void artc_set_rts_param(char* key, char* value);
    extern long long artc_get_state(AVFormatContext *ctx, int key);
    int version = 2;
    const struct rts_glue_funcs* rts_funcs = get_rts_funcs(version);
    // set to ffmpeg plugin
    av_set_rts_demuxer_funcs(rts_funcs);
    artc_set_rts_param((char*)"AutoReconnect", (char*)"false");
```

拓展ijk

i. 运行FFmpeg编译脚本*ijkplayer/config/module-lite.sh*。

ii. 复制Native RTS SDK中的*RtsSDK.framework*文件至*ijkplayer/ios/IJKMediaDemo/IJKMediaDemo*目录中。

iii. 工程中导入Native RTS SDK中的rtsdec.c文件。



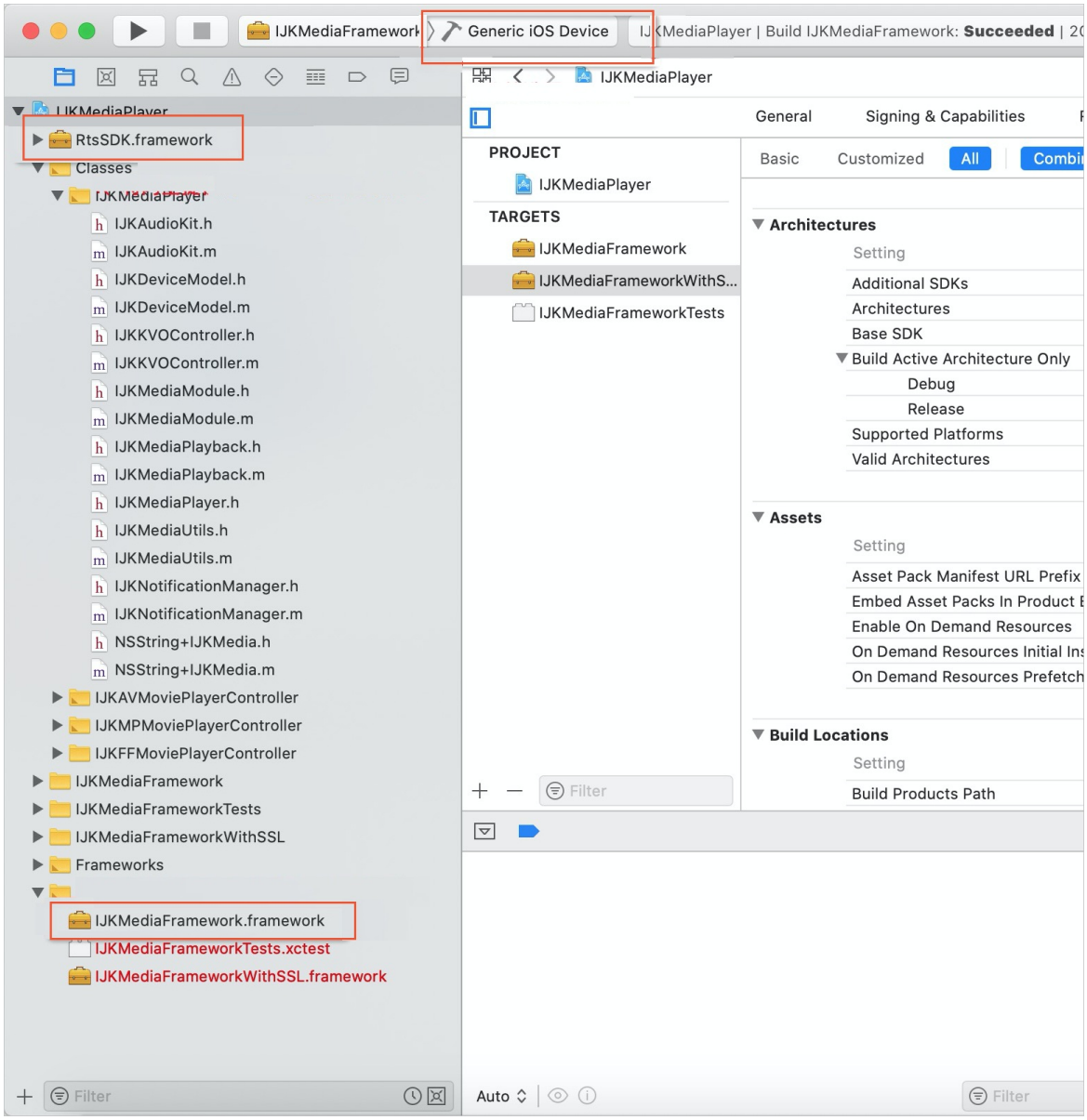
iv. 在ff_ffplay.c中添加RTS逻辑。

修改*ijkplayer/ijkmedia/ijkplayer/ff_ffplay.c*文件，设置ARTC的AVInputFormat函数指针。

```
diff --git a/ijkmedia/ijkplayer/ff_ffplay.c b/ijkmedia/ijkplayer/ff_ffplay.c
index 52aba31f..2beac440 100755
--- a/ijkmedia/ijkplayer/ff_ffplay.c
+++ b/ijkmedia/ijkplayer/ff_ffplay.c
@@ -3114,8 +3114,24 @@ static int read_thread(void *arg)
         av_dict_set_int(&ffp->format_opts, "skip-calc-frame-rate", ffp->skip_calc_frame_rate, 0);
     }

-    if (ffp->iformat_name)
-        is->iformat = av_find_input_format(ffp->iformat_name);
+    if(strncmp(is->filename, "artc://", 7) == 0) {
+        extern AVInputFormat ff_rtc_demuxer;
+        extern int artc_reload(AVFormatContext *ctx);
+        extern void av_set_rts_demuxer_funcs(const struct rts_glue_funcs *funcs);
+        extern void artc_set_rts_param(char* key, char* value);
+        extern long long artc_get_state(AVFormatContext *ctx, int key);
+
+        int version = 2;
+        const struct rts_glue_funcs* rts_funcs = get_rts_funcs(version);
+        // set to ffmpeg plugin
+        av_set_rts_demuxer_funcs(rts_funcs);
+        artc_set_rts_param((char*)"AutoReconnect", (char*)"false");
+        is->iformat = &ff_rtc_demuxer;
+    }
+    else {
+        if(ffp->iformat_name)
+            is->iformat = av_find_input_format(ffp->iformat_name);
+    }
     err = avformat_open_input(&ic, is->filename, is->iformat, &ffp->format_opts);
     if (err < 0) {
```

```
if(strncmp(is->filename, "artc://", 7) == 0) {
        extern AVInputFormat ff_rtc_demuxer;
        extern int artc_reload(AVFormatContext *ctx);
        extern void av_set_rts_demuxer_funcs(const struct rts_glue_funcs *funcs);
        extern void artc_set_rts_param(char* key, char* value);
        extern long long artc_get_state(AVFormatContext *ctx, int key);
        int version = 2;
        const struct rts_glue_funcs* rts_funcs = get_rts_funcs(version);
        // set to ffmpeg plugin
        av_set_rts_demuxer_funcs(rts_funcs);
        artc_set_rts_param((char*)"AutoReconnect", (char*)"false");
        is->iformat = &ff_rtc_demuxer;
    }
    else {
        if(ffp->iformat_name)
            is->iformat = av_find_input_format(ffp->iformat_name);
    }
```

4. 导入Native RTS SDK中*RtsSDK.framework*和ijkplayer中*IJKMediaFramework.framework*文件至工程中。



5. 调用ijkplayer接口实现低延时直播功能。
   - 创建ijkplayer

```
_ijkPlayer = [[IJKFFMoviePlayerController alloc] initWithContentURL:[NSURL URLWithStr
ing:_url] withOptions:options];
_ijkPlayer.view.autoresizingMask = UIViewAutoresizingFlexibleWidth|UIViewAutoresizing
FlexibleHeight;
_ijkPlayer.view.frame = self.basePlayerView.playerView.bounds;
_ijkPlayer.scalingMode = IJKMPMovieScalingModeAspectFit; //缩放模式
_ijkPlayer.shouldAutoplay = YES; //开启自动播放
```

   - 播放控制

```
- (void)prepareToPlay;
- (void)play;
- (void)pause;
- (void)stop;
```

7.3.3. Windows端

通过阅读本文，您可以了解Windows端依赖FFmpeg的其他播放器集成Native RTS SDK实现低延时直播的方法。

操作步骤

1. 下载并解压Native RTS SDK。下载地址，请参见SDK下载。
2. 复制rtsdec.c和rtsenc.c文件至FFmpeg播放器的*libavformat*目录中。
3. 修改Makefile文件。

   在Makefile文件中增加rtsdec.o和rtsenc.o的描述，如下图所示：

```
Makefile
1  NAME = avformat
2  DESC = FFmpeg container format library
3
4  HEADERS = avformat.h
5            avio.h
6            version.h
7
8  OBJS = allformats.o         \
9         avio.o               \
10        aviobuf.o            \
11        cutils.o             \
12        dump.o               \
13        format.o             \
14        id3v1.o              \
15        id3v2.o              \
16        metadata.o           \
17        mux.o                \
18        options.o            \
19        os_support.o         \
20        qtpalette.o          \
21        protocols.o          \
22        riff.o               \
23        sdp.o                \
24        url.o                \
25        utils.o              \
26        rtsdec.o             \
27        rtsenc.o             \
28
29 OBJS-$(HAVE_LIBC_MSVCRT)                   += file_open.o
30
```

4. 修改allformats.c文件，默认支持ARTC协议。

```
diff --git a/libavformat/allformats.c b/libavformat/allformats.c
index 405ddb5ad9f..a136a7836b0 100644
--- a/libavformat/allformats.c
+++ b/libavformat/allformats.c
@@ -385,6 +385,8 @@ static void register_all(void)
     REGISTER_DEMUXER (LIBGME,           libgme);
     REGISTER_DEMUXER (LIBMODPLUG,       libmodplug);
     REGISTER_DEMUXER (LIBOPENMPT,       libopenmpt);
+    extern AVInputFormat ff_rtc_demuxer;
+    av_register_input_format(&ff_rtc_demuxer);
 }
```

```
extern AVInputFormat ff_rtc_demuxer;
av_register_input_format(&ff_rtc_demuxer);
```

5. 编译运行。

   ./ffplay -i "artc://<播流地址>"

7.3.4. Mac端

通过阅读本文，您可以了解Mac端依赖FFmpeg的其他播放器集成Native RTS SDK实现低延时直播的方法。

操作步骤

1. 下载并解压Native RTS SDK。下载地址，请参见SDK下载。
2. 复制rtsdec.c和rtsenc.c文件至FFmpeg播放器的*libavformat*目录中。
3. 修改Makefile文件。

   在Makefile文件中增加rtsdec.o和rtsenc.o的描述，如下图所示：

Makefile

```
NAME = avformat
DESC = FFmpeg container format library

HEADERS = avformat.h
          avio.h
          version.h

OBJS = allformats.o         \
       avio.o               \
       aviobuf.o            \
       cutils.o             \
       dump.o               \
       format.o             \
       id3v1.o              \
       id3v2.o              \
       metadata.o           \
       mux.o                \
       options.o            \
       os_support.o         \
       qtpalette.o          \
       protocols.o          \
       riff.o               \
       sdp.o                \
       url.o                \
       utils.o              \
       rtsdec.o             \
       rtsenc.o             \

OBJS-$(HAVE_LIBC_MSVCRT)                 += file_open.o

```

4. 修改allformats.c文件，默认支持ARTC协议。

```
diff --git a/libavformat/allformats.c b/libavformat/allformats.c
index 405ddb5ad9f..a136a7836b0 100644
--- a/libavformat/allformats.c
+++ b/libavformat/allformats.c
@@ -385,6 +385,8 @@ static void register_all(void)
     REGISTER_DEMUXER (LIBGME,           libgme);
     REGISTER_DEMUXER (LIBMODPLUG,       libmodplug);
     REGISTER_DEMUXER (LIBOPENMPT,       libopenmpt);
+    extern AVInputFormat ff_rtc_demuxer;
+    av_register_input_format(&ff_rtc_demuxer);
 }
```

```
extern AVInputFormat ff_rtc_demuxer;
av_register_input_format(&ff_rtc_demuxer);
```

5. 编译运行。

```
./ffplay -i "artc://<播流地址>"
```

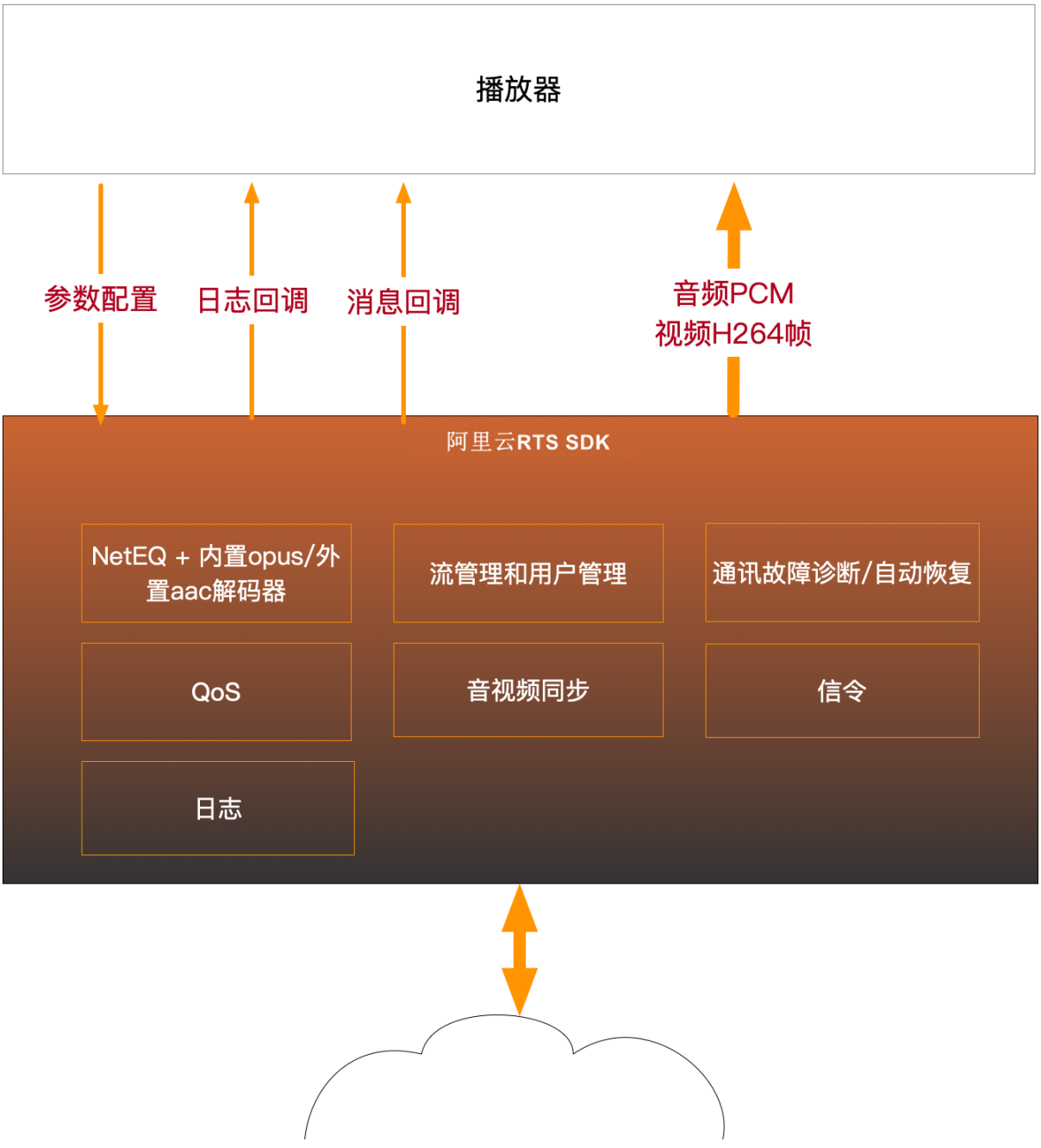
7.4. Native RTS SDK编程手册

7.4.1. Native RTS SDK编程简介

Native RTS SDK是介于播放器和阿里云GRTN网络之间的客户端SDK，用来从GRTN网络实时拉取音视频流，处理后将音频PCM和视频H.264帧提供给播放器进行解码或渲染。

架构方案

Native RTS SDK从阿里云GRTN网络拉取音视频，经过demuxer、jitterbuffer、音频解码和视频组帧后送到一个frame队列中，供播放器读取（PULL方式）。播放器将音频帧直接送至renderer，视频帧送至解码器，然后将解码输出的帧送到renderer，视频renderer通过pts和音频renderer同步。架构图如下所示：



相对于使用移动端RTS SDK中的rtsdec.c文件引入播放器，您还可以通过编程手册中的API实现集成播放器功能。详情请参见API。

Native RTS SDK提供参数配置接口，播放器可以对其参数进行调整。更多信息，请参见参数设置。

7.4.2. 编程模型

通过阅读本文，你可以了解Native RTS SDK的编程模型。

操作步骤

1. 编写以下代码。

> **说明** 此处以Mac端为例，其他系统请根据实际情况编写。

```
//test_rtssdk.c
#include "rts_api.h"
#include <stdarg.h>
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <sys/time.h>
#include <unistd.h>
static char *addr_to_string(const void *v, char buf[64])
{
    sprintf(buf, "%llu", (unsigned long long)v);
    return buf;
}
//日志回调函数
static int output_log(void *s, int level, const char *fmt, va_list args)
{
    (void) s;
    char str[1024];
    vsnprintf(str, 1000, fmt, args);
    printf("> %d %s\n", level, str);
    return 0;
```

```
}
//消息回调函数
static int on_event(void *s,
    int type,
    void *data, // temp data, do not cache it for later use!
    long long data_size)
{
    (void) s;
    printf("msg: %d %s\n", type, (data == NULL ? "null" : (const char *)data));
    return 0;
}
int main(int argc, const char **argv)
{
    const struct rts_glue_funcs *fs = get_rts_funcs(2); // 2: current version
    if(fs == NULL) {
        printf("Error: failed to get rts functions. check version.\n");
        return -1;
    }
    if(argc != 2) {
        printf("Usage: %s <artc url>\n", argv[0]);
        return -1;
    }
    const char *url = argv[1];
    char buf[64];
    //设置support id prefix
    fs->preconfig("HelpSupportIDPrefix", "yourInstanceID");
    //设置外置log function
    fs->preconfig("LogCallback",         addr_to_string(output_log, buf));
    fs->preconfig("LogCbParam",          addr_to_string(NULL, buf));
    //设置消息回调
    fs->preconfig("MessageCallback",     addr_to_string(on_event, buf));
    fs->preconfig("MessageCbParam",      addr_to_string(NULL, buf));
    //打开连接
    void *h = fs->open(url, "r");
    if(h == NULL) {
        printf("Error: failed to open %s!\n", url);
        return -1;
    }
    while(1) {
        struct rts_frame *frm = NULL;
        int c = fs->read(&frm, h);
        if(c == 1) {
            printf("got one frame\n");
            assert(frm != NULL);
            //TODO：将该帧送给解码器
            frm->free_ptr(frm);
            frm = NULL;
        }
        else if(c == 0) {
            assert(frm == NULL);
            printf("try again\n");
            //sleep a while，防止过于频繁的read占用太多cpu
            usleep(5 * 1000);
        }
```

```
        else {
            assert(frm == NULL);
            printf("Error: unknown\n");
            break;
        }
    }
    //关闭连接
    fs->close(h);
    return 0;
}
```

2. 执行以下命令编译上述*test_rtssdk.c*文件。

   gcc -I <path to header files> -o test_rtssdk test_rtssdk.c -L<path to library file> -lRtsSDK

3. 运行上述生成的可执行文件*test_rtssdk*。

   export DYLD_LIBRARY_PATH=<path to library file>

   ./test_rtssdk artc://<domain/app/name>

```
> 2 Start @1599029585691, Net sdk version 0.0.1_networkfd7ced1_LIBd84a462_sophoncb55d0435c_sophontb55d0435c_builde5fbe24_date2020.09.02.14:26-tag_rtssdk_1.3.0_startpoint-17-gfd7c
> 2 OS: Mac
> 2 Command queue thread running
> 2 help support id: <yourInstanceID-pull%2Ertsdemo%2Egrtn%2Ealiyunlive%2Ecom-mac-20200902065305-wxnArRGbLt5gGgJN>
try again
msg: 100 code=100,when=1599029585691,where=,who=0,desc="yourInstanceID-pull%2Ertsdemo%2Egrtn%2Ealiyunlive%2Ecom-mac-20200902065305-wxnArRGbLt5gGgJN"
<此处省略大量的打印信息>
got one frame
got one frame
got one frame
try again
try again
got one frame
try again
got one frame
got one frame
try again
try again
统计数据消息的内容:
msg: 105 code=105,when=1599029589707,where=,who=0,desc="haveVideo:1,videoCodecType:1,width:720,height:1280,receivedKeyFramePerSec:0.50,receivedVideoKbps:735,videoPktLostPerSec:0,receivedVideoPktPerSec:90,receivedVideoFramePerSec:12,videoNackPerSec:0,haveAudio:1,channels:2,sampleRate:48000,codecType:1,receivedAudioKbps:70,audioPktLostPerSec:0,receivedAudioPktPerSec:31,receivedAudioFramePerSec:31,audioNackPerSec:45,cacheDuration:0"
```

7.4.3. 参数设置

通过阅读本文，你可以了解全局参数和局部参数的设置方法。

全局参数设置

全局参数不针对特定的实例，设置时确保没有实例在运行（在open和close期间之外执行）。设置消息回调函数和日志回调函数请参见Native RTS SDK中的*rtsdec.c*文件。

```
__rts_funcs->preconfig("LogCallback", addr_to_string(output_log, buf));
__rts_funcs->preconfig("LogCbParam", addr_to_string(s, buf));
__rts_funcs->preconfig("MessageCallback", addr_to_string(format_control_message, buf));
__rts_funcs->preconfig("MessageCbParam", addr_to_string(s, buf));
...
__rts_funcs->open(...);
//这个期间不要调用preconfig
__rts_funcs->close(...);
//现在可以调用preconfig
```

局部参数设置

局部参数针对特定的实例，设置时需确保实例在运行（在open和close期间执行）。局部参数设置函数可实现接口调用、参数获取等功能。函数声明如下所示：

```
long long (* ioctl)(void *handle, const char *cmd, void *arg);
```

参数说明

参数	类型	描述
handle	void *	播放流时获取的句柄。
cmd	char *	命令，详情请参见下表。
arg	void *	命令所对应的参数。

命令	说明	命令对应的参数	返回值
get_stream_info	获取流参数。	无。	流参数指针 rts_worker_demux_info *。
reload	重连。	无。	返回0表示调用成功，小于0表示调用失败。
get_state	查询key-value。	ARTC_VIDEO_CACHE_SIZE（视频缓存时长），类型：int *。	缓存时长，单位：毫秒。
set_aes_key	设置解密密钥。	阿里云视频加密keyinfo，类型：char *。	返回0表示调用成功，小于0表示调用失败。
unsubscribe	停止订阅流。	无。	返回0表示调用成功，小于0表示调用失败。
subscribe	订阅流。	URL地址，类型：char *。	返回0表示调用成功，小于0表示调用失败。

7.4.4. API

7.4.4.1. 数据类型

通过阅读本文，你可以了解Native RTS SDK所涉及的数据类型。

目录

数据类型	描述
rts_worker_demux_info	媒体信息。
rts_frame	封装音频帧和视频帧的信息类型。
rts_glue_funcs	函数指针。
pusher_delay	推流延迟信息。
player_delay	拉流延迟信息。

详情

- rts_worker_demux_info：媒体信息。

```
struct rts_worker_demux_info
{
    int audio_flag;
    int audio_channels;
    int audio_sample_rate;
    int video_flag;
    int video_codec;
    int video_width;
    int video_height;
    int video_profile;
    int video_level;
    unsigned char spspps[10 * 1024];
    int spspps_len;
};
```

数据成员	描述
audio_flag	指示跟随其后的audio信息是否valid的标志。取值： ○ 1：valid。 ○ 0：invalid。
audio_channels	音频channel数。当audio_flag等于1的时候才有意义。
audio_sample_rate	音频采样率。当audio_flag等于1的时候才有意义。

数据成员	描述
video_flag	指示跟随其后的video信息是否valid的标志。取值： ○ 1：valid。 ○ 0：invalid。
video_codec	视频帧类型。当video_flag等于1（h264）的时候才有意义。
video_width	视频分辨率宽度。当video_flag等于1的时候才有意义。
video_height	视频分辨率高度。当video_flag等于1的时候才有意义。
video_profile	视频编码使用的profile。当video_flag等于1，且video_codec等于1的时候才有意义。
video_level	视频编码使用的level。当video_flag等于1，且video_codec等于1的时候才有意义。
spspps	存放sps和pps数据，可用于提前初始化解码器，可以为空。当video_flag等于1，且video_codec等于1的时候才有意义。
spspps_len	spspps的长度（单位：字节）。当video_flag等于1，且video_codec等于1的时候才有意义。

- rts_frame：封装音频帧和视频帧的信息类型。

```
struct rts_frame
{
    void *buf;              // where frame data is stored
    int size;               // size of frame data in bytes
    int is_audio;           // 1 for audio frame, 0 for video frame
    unsigned long long pts; // presentation time stamp, in ms
    unsigned long long dts; // decoding time stamp, in ms
    int flag;               // for video frame (is_audio == 0)
                            // bit 0: key frame
                            // bit 1: corruption
                            // bit 2: sps
                            // bit 3: sps change
    int duration;           // in ms
    // use this function to free rts_frame object
    void (*free_ptr)(struct rts_frame *);
    unsigned int uid; // reserved.
    struct pusher_delay delay;
    // msid-value = msid-id [ SP msid-appdata ]
    char msid[NET_MSID_MAX_LENGTH + 1];
};
```

数据成员	描述
buf	frame数据buffer。
size	buf的字节数。

数据成员	描述
is_audio	是否是音频帧。取值： ○ 1：音频帧。 ○ 0：视频帧。
pts	演示时间戳。单位：毫秒。
dts	解码时间戳。单位：毫秒。
flag	当is_audio等于0时有效。取值： ○ 1：损坏标志。 ○ 0：关键帧标志。
duration	帧持续时间。单位：毫秒。
free_ptr	函数指针，用来释放当前的rts_frame对象，例如 `frm->free_ptr(frm);` 。
uid	预留参数，标记数据属于哪个流。
delay	用于记录统计推流延迟信息的数据类型。详情请参见pusher_delay。

- rts_glue_funcs：函数指针。

```
struct rts_glue_funcs
{
    int api_version;
    int (* preconfig)(const char *key, const char *val);
    void (* open)(const char *url, const char mode);
    void (* close)(void handle);
    long long (* ioctl)(void *handle, const char *cmd, void arg);
    int (* read)(struct rtsframe **frame, void handle);
    int (* write)(struct rtsframe **frame, void *handle);
};
```

数据成员	描述
api_version	API版本号（必须是2）。
preconfig	全局参数配置函数。调用顺序：在open前调用，不要在open后调用。详情请参见preconfig。
open	打开一个流。详情请参见open。
close	关闭一个流。详情请参见close。
ioctl	参数配置、查询。详情请参见ioctl。
read	读取一帧数据。详情请参见read。
write	发送一帧数据。详情请参见write。

- pusher_delay：推流延迟信息。

```
struct pusher_delay
{
    long long cap_time_ms;  //current utc time when capture the video frame
    long long enc_time_ms;  //current utc time when the videoframe send to encoder
};
```

> **说明**
>
> 对于RTS的RTMP推流，可以通过插入SEI信息，统计全链路延迟信息。gdelay为全部延迟的总和，并非当前时间，单位：毫秒。

```
{
    "gdelay":100
}
```

数据成员	描述
cap_time_ms	采集该图像的系统时间，使用UNIX时间表示，单位：毫秒。
enc_time_ms	采集图像送入编码器的系统时间，使用UNIX时间表示，单位：毫秒。

- player_delay：拉流延迟信息。

```
struct player_delay
{
    long long decoder_time_ms;  //current utc time when the video frame send to decoder
    long long render_time_ms;   //current utc time when the video frame send to render
    unsigned long long pts;     //the video frame's pts
};
```

数据成员	描述
decoder_time_ms	采集该图像的UTC时间，单位：毫秒。
render_time_ms	送入编码器的UTC时间，单位：毫秒。
pts	演示时间戳，单位：毫秒。

7.4.4.2. 接口说明

通过阅读本文，你可以了解Native RTS SDK所涉及的接口详情。

目录

API	描述
get_rts_funcs	唯一对外可见的API，获取其他API的指针。

API	描述
preconfig	全局参数配置函数。
open	打开流。
close	关闭流。
ioctl	访问Native RTS SDK参数。
read	从Native RTS SDK读取一帧数据。
write	从Native RTS SDK发送一帧数据。

详情

- get_rts_funcs：唯一对外可见的API，获取其他API的指针。

```
const struct rts_glue_funcs *get_rts_funcs(
    int version
);
```

参数说明

参数	类型	描述
version	int	API兼容版本（必须是2）。

返回说明

返回rts_glue_funcs指针表示调用成功，返回NULL表示调用失败（一般情况为version参数不匹配）。

- preconfig：全局参数配置函数。在open和close期间之外执行。

```
int (* preconfig)(
    const char *key,
    const char *val
);
```

参数说明

参数	类型	描述
key	const char *	参数名称，大小写敏感，详情请参见下表。
val	const char *	参数值。

参数	描述
AutoReconnect	是否允许Native RTS SDK发现网络断开后自动重联，默认值为允许。
BufferingDuration	Jitter buffer缓存时间，默认值为500ms。
LogCallback	设置外部的日志回调函数，需要同时设置LogCbParam才能生效。

参数	描述
LogCbParam	设置外部的日志的回调函数的参数，需要同时设置LogCallback才能生效。
LogToConsole	设置内置的日志输出到控制台，默认值为不输出。
LogToFile	设置内置的日志输出到文件，默认值为不输出。
LogToServer	设置内置的日志输出到服务器，默认值为不输出。
MessageCallback	设置消息回调函数，需要同时设置MessageCbParam才能生效。
MessageCbParam	设置消息回调函数的参数，需要同时设置MessageCallback才能生效。
AacdCreateCallback	设置创建外部AAC解码器回调。
AacdDecodeCallback	设置外部AAC解码器解码回调。
AacdCloseCallback	设置关闭外部AAC解码器回调。
HelpSupportIDPrefix	设置Help Support ID的前缀。
LogLevel **说明** 此参数仅支持RTS SDK 2.1.0及以上版本。	设置日志回调的级别，对应的参数值： ○ 0：ERR。 ○ 1：WARN。 ○ 2：INFO。 ○ 3：DEBUG。 ○ 100：NONE。
SigTimeoutMs **说明** 此参数仅支持RTS SDK 2.2.0及以上版本。	设置信令超时时间，默认值为10000ms，建议设置不小于6000ms。

返回说明

返回0表示调用成功，其他表示调用失败。

- open：打开流。

```
void *(* open){
    const char *url,
    const char *mode
);
```

参数说明

参数	类型	描述
url	const char *	流的URL，支持 `artc://` 格式的URL。

参数	类型	描述
mode	const char *	模式。取值： ○ r：播放流。 ○ w：推送流。

返回说明

返回句柄表示调用成功，返回NULL表示调用失败。

- close：关闭流。

```
void (* close)(
    void *handle
);
```

参数说明

参数	类型	描述
handle	void *	open返回的句柄。

- ioctl：访问Native RTS SDK参数。

```
long long (* ioctl)(
    void *handle,
    const char *cmd,
    void *arg
);
```

参数说明

参数	类型	描述
handle	void *	open返回的句柄。
cmd	const char *	发送的命令名称，详情请参见下表。
arg	void *	发送的命令参数。如果命令不带参数，填NULL。

命令名称	命令参数	描述
get_stream_info	rts_worker_demux_info变量地址	从Native RTS SDK获得流信息，存储到arg指向的rts_worker_demux_info变量中（rts_worker_demux_info变量不能为NULL）。
reload	NULL	提示Native RTS SDK重新连接。

命令名称	命令参数	描述
skip_avformat_find_stream_info 说明 此命令仅支持RTS SDK 2.2.0及以上版本。	NULL	基于FFmpeg的播放器起播阶段是否可以跳过 `avformat_find_stream_info()` 。 返回0表示不能跳过，返回1表示可以跳过。 注意 如果您关注视频秒开，建议调用此命令。

返回说明

返回0表示调用成功，其他表示调用失败。

- read：从Native RTS SDK读取一帧数据。

```
int (* read)(
    struct  **frame,
    void *handle
);
```

参数说明

参数	类型	描述
frame	struct **	返回的音视频帧的存储位置。
handle	void *	open返回的句柄。

返回说明

- 返回1表示读入frame一帧。
- 返回0表示稍后再试。
- 返回-1表示EOF。
- 返回除1、0、-1以外的负数表示其他致命错误。

说明

如果返回值为1，返回的frame需要调用者释放。释放的示例代码如下所示：

```
struct rts_frame *f = NULL;
int r = __rts_funcs->read(&f, handle);
...
if(f != NULL)
    f->free_ptr(f);
```

- write：从Native RTS SDK发送一帧数据。

```
int (* write)(
    struct  **frame,
    void *handle
);
```

参数说明

参数	类型	描述
frame	struct **	要发送的音视频帧。
handle	void *	open返回的句柄。

返回说明

- 返回1表示调用成功。
- 返回0表示稍后再试。
- 返回-1表示EOF。
- 返回除1、0、-1以外的负数表示其他致命错误。

> **说明** 如果返回值为1，返回的frame所有权转移至Native RTS SDK并由其释放。

7.4.5. 消息回调

通过阅读本文，您可以了解Native RTS SDK消息回调函数详情及其示例。

函数说明

```
int (* event_callback)(
    void *opaque,
    int type,
    void *data,
    long long data_size
);
```

参数说明

参数	类型	描述
opaque	void *	通过preconfig("MessageCbParam")设置的值，RTS SDK将原值回传给 `event_callback` 。
type	int	消息ID，详情请参见下表。
data	void *	消息内容，格式为 `when=<time_in_ms>,where=<place>,who=<id>,desc=<extra text>` 。该参数为临时变量，出回调函数会释放。
data_size	long long	消息内容长度，单位：字节。

消息ID	支持的最低版本	说明
100	无限制	Help Support ID消息，发生在open函数中的connect之前。
102	无限制	Help Support ID消息，发生在connect操作中。
103	无限制	Help Support ID消息，发生在推流操作中。
104	无限制	Help Support ID消息，发生在拉流操作中。
105	无限制	每4秒统计一次数据消息。 **说明** 当消息ID是105（统计数据消息）时，desc的内容格式为 `\"key1:val1,key2:val2,...\"` 。
121	2.2.0	停止拉流。
20000 ~ 30000	无限制	错误码。
20001	无限制	DNS出错，无法解析域名。
20002	无限制	域名访问权限验证失败。
20003	2.2.0	订阅URL协议头不为artc。
20004	2.2.0	推流URL的协议头不为artc。
20005	2.2.0	RTS SDK内存分配失败。
20009	无限制	成功连接到流媒体服务器。
20010	无限制	连接流媒体服务器失败。
20011	2.2.0	建连信令超时。
20012	2.2.0	订阅信令超时。
20013	无限制	订阅流不存在。
20014	无限制	订阅流没有音频。
20015	无限制	订阅流没有视频。
20016	2.2.0	订阅信令成功后，|x|秒没有收到音频RTP包，|x|默认10，可通过preconfig("SigTimeoutMs", "10000")配置。
20017	2.2.0	订阅信令成功后，|x|秒没有收到视频RTP包，|x|默认10，可通过preconfig("SigTimeoutMs", "10000")配置。
20020	无限制	其他未知订阅错误。
20050	无限制	丢包率高，拥塞严重。

消息ID	支持的最低版本	说明
20051	无限制	丢包率降到正常水平，拥塞恢复。
20052	无限制	流中断，没有任何音频或者视频包。
20053	无限制	流中断恢复。
20054	无限制	流结束。
20055	无限制	连接丢失。
20056	无限制	检测到流重新推送。
20057	无限制	无法播放，需要降级到RTMP。

示例说明

消息回调函数通过preconfig设置给Native RTS SDK，示例如下所示：

```
static int on_message(void *s,
                      int type,
                      void *data,
                      long long data_size)
{
    (void) s;
    //TODO: process message, do not take too long
    switch(type) {
    case 105:
        printf("Profiling message %s.\n", (const char *)data);
        break;
    default:
        break;
    }
    return 0;
}
__rts_funcs->preconfig("MessageCallback", addr_to_string(on_message, buf));
__rts_funcs->preconfig("MessageCbParam", addr_to_string(s, buf));
```

关于消息回调函数更多信息，请参见Native RTS SDK中的rtsdec.c文件。

7.4.6. 日志回调

通过阅读本文，你可以了解外部日志回调和内置日志回调函数的详情。

外部日志回调

外部日志回调函数声明如下所示：

```
int (* output_log)(
    void *s,
    int level,
    const char *fmt,
    va_list args
);
```

参数说明

参数	类型	描述
s	void *	通过preconfig（"LogCbParam"）设置值，Native RTS SDK将原值回传给 `output_log` 。
level	int	日志等级，取值： • 0：Error。 • 1：Warning。 • 2：Info。 • 3：Debug。
fmt	const char *	格式化字符串，和 `printf` 中的 `format` 参数一致。
args	va_list	可变参数列表。

示例说明

外部日志回调函数通过preconfig设置给Native RTS SDK，示例如下所示：

```
//提供一个回调函数
static int output_log(struct AVFormatContext *s, int level, const char *fmt, va_list args)
{
    //TODO: 处理log
    return 0;
}
//注册这个回调函数给RTS SDK
__rts_funcs->preconfig("LogCallback", addr_to_string(output_log, buf));
__rts_funcs->preconfig("LogCbParam", addr_to_string(s, buf));
```

关于外部日志回调函数更多信息，请参见Native RTS SDK中的rtsdec.c文件。

内置日志回调

内置日志回调功能通过preconfig接口实现，函数声明如下所示：

```
int (* preconfig)(
    const char *key,
    const char *val
);
```

参数说明

参数	类型	描述
key	const char *	参数名称，大小写敏感，详情请参见下表。
val	const char *	参数值。

参数	解释
LogToConsole	是否允许输出到控制台。
LogToFile	是否允许输出到文件。
LogToServer	是否允许输出到服务器。
LogLevel	日志过滤条件。符合日志过滤条件或者严重等级更高的日志才会输出。日志严重等级如下所示： • 0：ERROR。 • 1：WARN。 • 2：INFO。 • 3：DEBUG。

示例说明

```
__rts_funcs->preconfig("LogToConsole", "true"); //output to console
__rts_funcs->preconfig("LogToFile",    "true"); //output to file
__rts_funcs->preconfig("LogToServer",  "true"); //output to server
__rts_funcs->preconfig("LogLevel",     "2");    //LOG_INFO
```

8.Web RTS SDK

8.1. Web RTS SDK简介

通过阅读本文，您可以了解Web RTS SDK基本信息和浏览器要求。

产品特性

Web RTS SDK是通过HTML网页加载的JavaScript库，不包含普通直播Web推流与播放功能，并且不依赖用户安装App实现直播推流播放。Web RTS SDK只用于RTS低延时直播协议推流与拉流，可以解决在移动端、桌面端和微信上浏览器的推流与播放问题。

浏览器要求

Web RTS SDK视频推流与播放能力不依赖网页代码，而是靠浏览器支持，浏览器具体要求如下所示：

操作系统	浏览器	支持的最低版本
Windows	Chrome	63
	Firefox	62
	Opera	15
	Edge	79
Mac	Chrome	63
	Safari	11
	Firefox	62
	Opera	15
	Edge	79
Android	Chrome	63
	Edge	79
	微信内置浏览器	7.0.9（微信版本）
iOS	Safari	11
	Edge	79
	微信内置浏览器	7.0.9（微信版本）
	钉钉内置浏览器	11.2.5（钉钉版本）

说明 由于浏览器针对WebRTC安全策略限制，RTS拉流域名需要配置HTTPS证书，否则可能导致直播流无法正常播放。

接口说明

相对于旧版本（V1.1.0~V1.2.4），最新版本Web RTS SDK对接口进行了变更，并且新增RTS推流功能，推荐您使用最新版本的SDK，详情请参见接口说明。

如果您需要查看旧版本Web RTS SDK（V1.1.0~V1.2.4）的接口详情，请参见接口说明（旧版：V1.1.0~V1.2.4）。

错误码说明

最新版本Web RTS SDK错误码详情请参见错误码说明。

如果您需要查看旧版本Web RTS SDK（V1.1.0~V1.2.4）的错误码详情，请参见错误码说明（旧版：V1.1.0~V1.2.4）。

8.2. 接口说明

通过阅读本文，您可以了解Web RTS SDK的接口详情。

目录

API	描述
createClient	实例化对象。
isSupport	检测浏览器是否可用。
subscribe	开始拉RTS流。
unsubscribe	停止RTS播放。
muted	静音。
createStream	获取本地流。
publish	开始推流。
unpublish	停止推流。
on	回调监听。

详情

- createClient：实例化对象。

```
var aliRts = AliRTS.createClient();
```

- isSupport：检测浏览器是否可用。

```
/**
 * isSupport检测是否可用
 * @param {Object} supportInfo 检测信息
 * @param {Boolean} supportInfo.isReceiveVideo 是否拉视频流
 * @return {Promise}
 */
aliRts.isSupport(supportInfo).then(re=> {
  // 可用
}).catch(err=> {
  // 不可用
  console.log(`not support errorCode: ${err.errorCode}`);
  console.log(`not support message: ${err.message}`);
})
```

- subscribe：开始拉RTS流。

```
/**
 * rts开始拉流接口
 * @param {String} pullStreamUrl 拉流地址,在地址后添加@subaudio=no或者@subvideo=no来表示不订阅音频流或视频流
 * @return {Promise}
 */
aliRts.subscribe(pullStreamUrl).then((remoteStream) => {
  // mediaElement是媒体标签audio或video
  remoteStream.play(mediaElement);
}).catch((err) => {
  // 订阅失败
})
```

新增说明，用户使用aac音频格式可进行转码配置即可。

> **注意**
>
> 低延时直播（RTS）对音视频解码存在以下限制：
>
> - 音频不支持AAC编码（会出现噪音）。
> - 视频不支持B帧（会出现画面跳跃）、H.265编码，GOP不大于3秒（会增加延迟）。
>
> 如有需求可进行RTS转码解除限制，具体操作请参见RTS转码。

- unsubscribe：停止RTS播放。

```
aliRts.unsubscribe();
```

- muted：静音。

```
remoteStream.muted = true;
```

- createStream：获取本地流。

```
/**
 * 获取本地流localStream
 */
AliRTS.createStream({
  audio: true,
  video: true,
  screen: false,
}).then((localStream) => {
  // 预览推流内容，mediaElement是媒体标签audio或video
  localStream.play(mediaElement);
}).catch((err) => {
  // 创建本地流失败
})
```

- publish：开始推流。

```
/**
 * 开始推流
 */
aliRts.publish(pushUrl, localStream)).then(() => {
  // 推流成功
}).catch((err) => {
  // 推流失败
})
```

- unpublish：停止推流。

```
aliRts.unpublish();
```

- on：回调监听。

```
/*
 * 在onError中获取到错误码10201时，此时网页的音频是静音的，
 * 需要用户在网页上手动触发事件（必须有用户交互，不能直接通过代码控制）
 * 调用remoteStream.muted = false取消静音
 */
aliRts.on("onError", (err)=> {
  console.log(`errorCode: ${err.errorCode}`);
  console.log(`message: ${err.message}`);
})
const PLAY_EVENT = {
  CANPLAY: "canplay",
  WAITING: "waiting",
  PLAYING: "playing"
}
aliRts.on('onPlayEvent', (play)=>{
  if(play.event === PLAY_EVENT.CANPLAY){
    //拉流可以播放
  }else if(play.event === PLAY_EVENT.WAITING){
    //拉流卡顿等待缓冲中 （仅Chrome）
  }else if(play.event === PLAY_EVENT.PLAYING){
    //拉流卡顿结束恢复播放 （仅Chrome）
  }
})
```

8.3. 错误码说明

通过阅读本文，您可以了解Web RTS SDK的错误码详情。

Web RTS SDK错误码如下所示：

错误码	错误信息	描述
10000	device unknown error	设备未知错误。
10001	audio device not found	没有找到音频设备。
10002	video device not found	没有找到视频设备。
10003	audio device not allowed	不允许使用音频设备。
10004	video device not allowed	不允许使用视频设备。
10005	audio device not readable	没有使用音频设备权限。
10006	video device not readable	没有使用视频设备权限。
10007	device constrained error	获取设备参数错误。
10010	screenshare unknown error	屏幕共享未知错误。
10011	screenshare not allowed	不允许屏幕共享。
10012	screenshare ended	屏幕共享结束。
10013	screenshare no permission	应用没有屏幕共享的权限。
10018	screenshare not support	不支持屏幕共享。
10101	http request error	信令请求失败。
10102	play fail	播放失败（默认5秒没有返回成功表示播放失败）。
10110	not support webrtc	不支持WebRTC。
10111	browser not support	不支持此浏览器。
10112	browser version too low	浏览器版本过低。
10113	not support h264	不支持H.264。
10114	create offer error	create offer失败。
10125	html element not match	订阅内容和传入的媒体类型不匹配。
10201	auto play failed	自动播放失败。
10202	play url error	播放URL协议错误。

错误码	错误信息	描述
10203	subscribe nothing	没有订阅任何内容。
10204	html element error	订阅传入的HTML Element不是audio或video。
10205	http request error	HTTP请求失败。
10206	http answer error	answer失败。
10300	publish url error	推流URL错误。
10301	publish no audio	推流缺少音频。
11000	track type audio	track type不存在。
11001	track kind error	media stream track类型错误，例如添加audio，但是传入的是video track。
11002	no audio track error	没有音频流，至少需要一个audio track。
12000	peerconnection error	peer connection异常。

9.OBS插件SDK

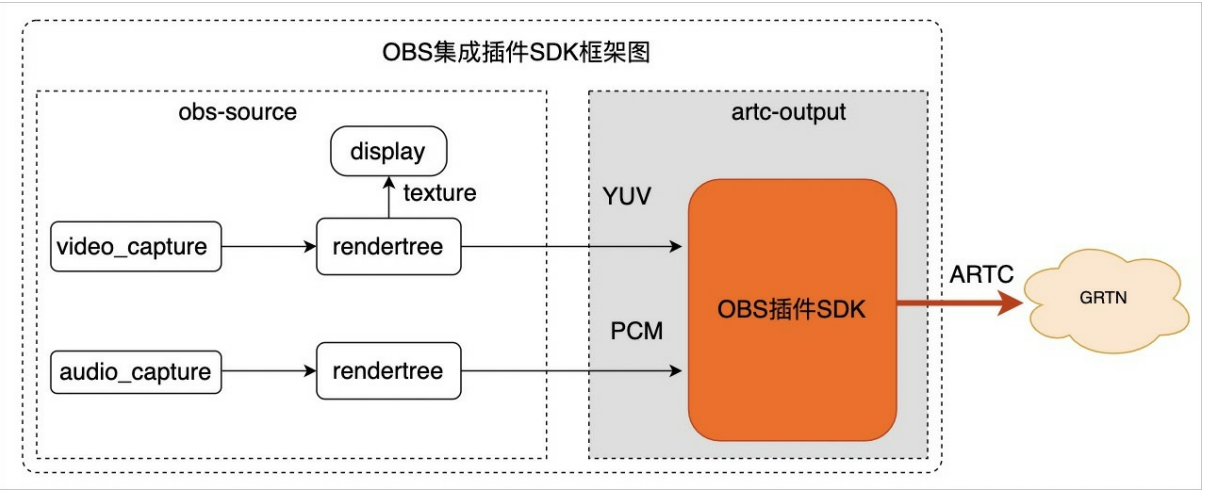
9.1. OBS插件SDK简介

通过阅读本文，您可以了解OBS插件SDK的基本信息。

产品特性

OBS插件SDK是为OBS量身打造，无需改动OBS原生框架，接入之后即可实现OBS OutPut功能，可以将OBS采集的音视频画面进行编码并实时推送RTS流至阿里云GRTN网络。

架构方案



Demo体验

您可以通过下载OBS插件Demo快速体验低延时推流。下载地址，请参见OBS插件SDK。

9.2. OBS插件SDK集成与实现

通过阅读本文，您可以了解OBS插件SDK的集成与实现方法。

前提条件

推流域名已开启低延时推流功能。具体操作，请参见通过控制台开启低延时推流功能。

操作步骤

> **说明** 本文中的实现方法仅供参考，您可以根据实际业务需求进行开发，详细示例请参见*ali_rts_sdk/obs-output-example/artc-stream.c*。

1. 下载并解压OBS插件SDK。下载地址，请参见SDK下载。
2. 工程中导入头文件*ali_rts_sdk.h*。

```
#include "ali_rts_sdk.h"
```

3. 创建实例。

```
/** 1.事件回调设置 **/
alirts_event_handler_t event_handler = {0};
// user_data
event_handler.user_data = user_data;
// 日志回调
event_handler.on_log_data = on_log_data;
// 错误回调
event_handler.on_error = on_error;
// 推流开始回调
event_handler.on_push_start = on_push_start;
// 推流结束回调
event_handler.on_push_stop  = on_push_stop;
// 网络断开回调
event_handler.on_connection_lost = on_connection_lost;
// 重连状态回调
event_handler.on_reconnect_state = on_reconnect_state;
/** 2.创建alirts实例,获得句柄 **/
alirts_handler_t* alirts = alirts_create(&event_handler);
```

4. 推流。

```
/** 1.推流配置设置 **/
alirts_config_t config = {0};
// 视频相关配置 1280x720@25fps
config.width = 1280;
config.height = 720;
config.gop_size = 2;
config.max_fps = 25;
// 视频码率设置为2.5M
config.video_bitrate_bps = 2500000;
// 视频编码器配置
config.video_codec = VIDEO_CODEC_H264;
config.video_preset = VIDEO_PRESET_FAST;
config.video_profile = VIDEO_PROFILE_BASELINE;
// 音频相关配置
// 音频仅支持44.1k和48k
config.sample_rate = SAMPLE_RATE_48000;
// 音频仅支持16bit
config.sample_format = FORMAT_16BIT;
// 音频通道设置
config.channel = 2;
// 当前音频码率设置为64k
config.audio_bitrate_bps = 64000;
// 重连配置
//建议开启自动重连
config.enable_auto_reconnect = true;
// 设置重连超时时间为10s
config.reconnect_timeout = 10;
// 推流地址
config.url= "artc://xxxx/xx/xxxxx";
/** 2.开始推流 **/
int ret = alirts_start(alirts, &config);
if (ret != 0) {
  // 开始推流失败
}
```

> **说明** `alirts_start` 调用成功之后，会触发 `on_push_start` 回调。

5. 发送数据。

```
// 以发送视频数据为例，音频发送类似
/** 1.设置视频帧 **/
// 视频数据设置之前必须要清零
alirts_frame_t video_frame = {0};
// 当前视频仅支持MEDIA_TYPE_VIDEO_I420输入
video_frame.type = MEDIA_TYPE_VIDEO_I420;
video_frame.width = frame->width;
video_frame.height = frame->height;
video_frame.timestamp = frame->timestamp;
video_frame.data[0] = frame->data[0];
video_frame.data[1] = frame->data[1];
video_frame.data[2] = frame->data[2];
video_frame.linesize[0] = frame->linesize[0];
video_frame.linesize[1] = frame->linesize[1];
video_frame.linesize[2] = frame->linesize[2];
/** 2.发送视频帧 **/
int ret = alirts_send_frame(alirts, &video_frame);
if (ret != 0) {
    // 视频发送失败
}
```

> 说明 建议在 `on_push_start` 回调之后，调用 `alirts_send_frame` 发送数据。

6. 结束推流。

```
alirts_stop(alirts);
```

> 说明 `alirts_stop` 调用成功之后，会触发 `on_push_stop` 回调。

7. 释放实例。

```
alirts_destroy(alirts);
```

9.3. 实现OBS OutPut插件功能

OBS工具配合OBS插件SDK可实现OutPut插件功能。通过阅读本文，你可以了解实现OBS OutPut插件的方法。

使用限制

仅支持在Windows和Mac平台编译OBS源码。

操作步骤

1. 下载并解压OBS源码。下载地址，请参见OBS-Studio。
2. 下载并解压OBS插件SDK。下载地址，请参见SDK下载。
3. 在*obs-studio/plugins/obs-outputs*目录（OBS源码目录）中创建*ali_rts_sdk*文件夹，并复制*ali_rts_sdk*目录（OBS插件SDK解压之后的目录）中的*include*和*lib*文件夹至*obs-studio/plugins/obs-outputs/ali_rts_sdk*目录中。
4. 复制*ali_rts_sdk/obs-output-example*目录（OBS插件SDK解压之后的目录）中的*artc-stream.h*和*artc-s*

*tream.c*文件至*obs-studio/plugins/obs-outputs*目录（OBS源码目录）中。

5. 修改编译脚本*obs-studio/plugins/obs-outputs/CMakeLists.txt*。

~/coderoom/obs/plugins/obs-outputs — vim CMakeLists.txt

```
        set_source_files_properties(${obs-outputs_librtmp_SOURCES} PROPERTIES
                COMPILE_FLAGS "-fvisibility=hidden")
endif()

set(obs-artc_HEADERS
  ${CMAKE_CURRENT_SOURCE_DIR}/ali_rts_sdk/include/ali_rts_sdk.h
        artc-stream.h)
set(obs-artc_SOURCES
        artc-stream.c)

if(WIN32)
        set(obs-artc_LIBS
            ${CMAKE_CURRENT_SOURCE_DIR}/ali_rts_sdk/lib/win64/ali_rts_sdk.lib
      ${CMAKE_CURRENT_SOURCE_DIR}/ali_rts_sdk/lib/win64/RtsSDK.lib)
else()
        set(obs-artc_LIBS
                ${CMAKE_CURRENT_SOURCE_DIR}/ali_rts_sdk/lib/mac/ali_rts_sdk.dylib
      ${CMAKE_CURRENT_SOURCE_DIR}/ali_rts_sdk/lib/mac/libRtsSDK.dylib)
endif()

set(obs-outputs_HEADERS
        "${CMAKE_BINARY_DIR}/plugins/obs-outputs/config/obs-outputs-config.h"
        obs-output-ver.h
        rtmp-helpers.h
        rtmp-stream.h
        net-if.h
        flv-mux.h
        ${obs-artc_HEADERS})
set(obs-outputs_SOURCES
        obs-outputs.c
        null-output.c
        rtmp-stream.c
        rtmp-windows.c
        flv-output.c
        flv-mux.c
        net-if.c
        ${obs-artc_SOURCES})

if(WIN32)
        set(MODULE_DESCRIPTION "OBS output module")
        if(NOT BUILD_LIVE_ENGINE_SHARED)
                add_definitions(-DENGINE_STATIC)
        endif()
        configure_file(${CMAKE_SOURCE_DIR}/cmake/winrc/obs-module.rc.in obs-outputs.rc)
        list(APPEND obs-outputs_SOURCES
                obs-outputs.rc)
endif()

add_library(obs-outputs MODULE
        ${ftl_SOURCES}
        ${ftl_HEADERS}
        ${obs-outputs_SOURCES}
        ${obs-outputs_HEADERS}
        ${obs-outputs_librtmp_SOURCES}
        ${obs-outputs_librtmp_HEADERS})

target_link_libraries(obs-outputs ${obs-artc_LIBS})

if(FTL_FOUND)
        target_link_libraries(obs-outputs ${FTL_LIBRARIES})
        target_include_directories(obs-outputs PUBLIC ${FTL_INCLUDE_DIRS})
endif()
```

```
set(obs-artc_HEADERS
  ${CMAKE_CURRENT_SOURCE_DIR}/ali_rts_sdk/include/ali_rts_sdk.h
        artc-stream.h)
set(obs-artc_SOURCES
        artc-stream.c)
if(WIN32)
        set(obs-artc_LIBS
            ${CMAKE_CURRENT_SOURCE_DIR}/ali_rts_sdk/lib/win64/ali_rts_sdk.lib
      ${CMAKE_CURRENT_SOURCE_DIR}/ali_rts_sdk/lib/win64/RtsSDK.lib)
else()
        set(obs-artc_LIBS
                  ${CMAKE_CURRENT_SOURCE_DIR}/ali_rts_sdk/lib/mac/ali_rts_sdk.dylib
      ${CMAKE_CURRENT_SOURCE_DIR}/ali_rts_sdk/lib/mac/libRtsSDK.dylib)
endif()
set(obs-outputs_HEADERS
        "${CMAKE_BINARY_DIR}/plugins/obs-outputs/config/obs-outputs-config.h"
        obs-output-ver.h
        rtmp-helpers.h
        rtmp-stream.h
        net-if.h
        flv-mux.h
        ${obs-artc_HEADERS})
set(obs-outputs_SOURCES
        obs-outputs.c
        null-output.c
        rtmp-stream.c
        rtmp-windows.c
        flv-output.c
        flv-mux.c
        net-if.c
        ${obs-artc_SOURCES})
if(WIN32)
        set(MODULE_DESCRIPTION "OBS output module")
        if(NOT BUILD_LIVE_ENGINE_SHARED)
                add_definitions(-DENGINE_STATIC)
        endif()
        configure_file(${CMAKE_SOURCE_DIR}/cmake/winrc/obs-module.rc.in obs-outputs.rc)
        list(APPEND obs-outputs_SOURCES
                obs-outputs.rc)
endif()
add_library(obs-outputs MODULE
        ${ftl_SOURCES}
        ${ftl_HEADERS}
        ${obs-outputs_SOURCES}
        ${obs-outputs_HEADERS}
        ${obs-outputs_librtmp_SOURCES}
        ${obs-outputs_librtmp_HEADERS})
target_link_libraries(obs-outputs ${obs-artc_LIBS})
```

6. 修改*obs-studio/plugins/obs-outputs/obs-outputs.c*文件注册artc_output插件。

```
diff --git a/plugins/obs-outputs/obs-outputs.c b/plugins/obs-outputs/obs-outputs.c
index 31534cbd3..2db55f1af 100644
--- a/plugins/obs-outputs/obs-outputs.c
+++ b/plugins/obs-outputs/obs-outputs.c
@@ -16,6 +16,7 @@ MODULE_EXPORT const char *obs_module_description(void)
 }

 extern struct obs_output_info rtmp_output_info;
+extern struct obs_output_info artc_output_info;
 extern struct obs_output_info null_output_info;
 extern struct obs_output_info flv_output_info;
 #if COMPILE_FTL
@@ -66,6 +67,7 @@ bool obs_module_load(void)
 #endif

        obs_register_output(&rtmp_output_info);
+       obs_register_output(&artc_output_info);
        obs_register_output(&null_output_info);
        obs_register_output(&flv_output_info);
 #if COMPILE_FTL
```

```
extern struct obs_output_info artc_output_info;
```

```
obs_register_output(&artc_output_info);
```

7. 修改*obs-studio/UI/window-basic-main-outputs.cpp*文件配置启动artc_output插件。

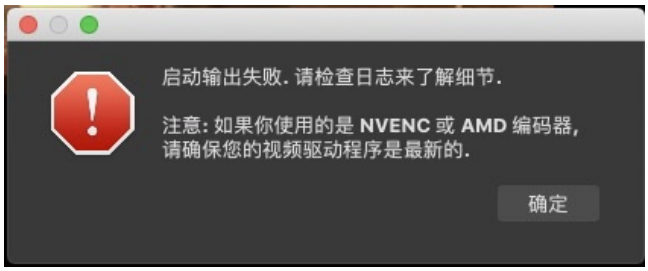
```
diff --git a/UI/window-basic-main-outputs.cpp b/UI/window-basic-main-outputs.cpp
index b66fdaef7..08fe5d2a6 100644
--- a/UI/window-basic-main-outputs.cpp
+++ b/UI/window-basic-main-outputs.cpp
@@ -18,6 +18,7 @@ volatile bool virtualcam_active = false;

 #define FTL_PROTOCOL "ftl"
 #define RTMP_PROTOCOL "rtmp"
+#define ARTC_PROTOCOL "artc"

 static void OBSStreamStarting(void *data, calldata_t *params)
 {
@@ -766,6 +767,9 @@ bool SimpleOutput::SetupStreaming(obs_service_t *service)
                if (url != NULL &&
                    strncmp(url, FTL_PROTOCOL, strlen(FTL_PROTOCOL)) == 0) {
                        type = "ftl_output";
+               } else if (url != NULL &&
+                   strncmp(url, ARTC_PROTOCOL, strlen(ARTC_PROTOCOL)) == 0) {
+                       type = "artc_output";
                } else if (url != NULL && strncmp(url, RTMP_PROTOCOL,
                                                  strlen(RTMP_PROTOCOL)) != 0) {
                        type = "ffmpeg_mpegts_muxer";
@@ -1690,6 +1694,9 @@ bool AdvancedOutput::SetupStreaming(obs_service_t *service)
                if (url != NULL &&
                    strncmp(url, FTL_PROTOCOL, strlen(FTL_PROTOCOL)) == 0) {
                        type = "ftl_output";
+               } else if (url != NULL &&
+                   strncmp(url, ARTC_PROTOCOL, strlen(ARTC_PROTOCOL)) == 0) {
+                       type = "artc_output";
                } else if (url != NULL && strncmp(url, RTMP_PROTOCOL,
                                                  strlen(RTMP_PROTOCOL)) != 0) {
                        type = "ffmpeg_mpegts_muxer";
```

```
const char *type = obs_service_get_output_type(service);
if (!type) {
    type = "rtmp_output";
    const char *url = obs_service_get_url(service);
    if (url != NULL &&
        strncmp(url, FTL_PROTOCOL, strlen(FTL_PROTOCOL)) == 0) {
        type = "ftl_output";
    } else if (url != NULL &&
        strncmp(url, ARTC_PROTOCOL, strlen(ARTC_PROTOCOL)) == 0) {
        type = "artc_output";
    } else if (url != NULL && strncmp(url, RTMP_PROTOCOL,
                    strlen(RTMP_PROTOCOL)) != 0) {
        type = "ffmpeg_mpegts_muxer";
    }
}
```

8. 编译并运行。具体操作，请参见安装说明。

   编译完成之后，运行之前需要将OBS插件SDK动态库文件复制到OBS运行目录下。

   以Mac端Debug环境为例（其他端环境与Mac端类似，本文不再赘述），复制*ali_rts_sdk/lib/mac*目录（OBS插件SDK动态库目录）中的*libRtsSDK.dylib*和*ali_rts_sdk.dylib*文件至*obs-studio/build/rundir/Debug/bin*目录（OBS Debug环境运行目录）中。如果没有复制动态库文件，可能会提示以下错误：



后续步骤

执行以下步骤测试artc_output插件功能（RTS推流）。

1. 打开OBS推流工具。
2. 在左侧导航栏选择**推流**，**服务器**输入已生成的完整推流地址`artc://<推流地址>`，进行RTS推流。
3. 在播放器中输入播流地址，例如：`artc://<播流地址>`，播放RTS直播流。

9.4. API

9.4.1. 数据类型

通过阅读本文，你可以了解OBS插件SDK所涉及的数据类型。

目录

数据类型	描述
alirts_handler_t	SDK句柄定义，用于SDK实例管理。
alirts_config_t	推流配置。

数据类型	描述
alirts_sei_info_t	用户自定义的SEI信息。
alirts_frame_t	发送的音视频帧数据。
alirts_push_info_t	推流统计信息。
alirts_event_handler_t	SDK推流的回调事件通知。

详情

- alirts_handler_t：SDK句柄定义，用于SDK实例管理。

```
typedef void alirts_handler_t;
```

- alirts_config_t：推流配置。

数据成员	类型	描述
width	int	推流视频的宽，与alirts_frame_t视频帧的宽一致。
height	int	推流视频的高，与alirts_frame_t视频帧的高一致。
gop_size	int	编码器GOP大小，单位：秒，取值范围：[0,30]，建议取值：[2,5]。
max_fps	int	视频帧率。
video_bitrate_bps	int	视频目标码率。
video_codec	alirts_video_codec_e	视频编码器类型。
video_preset	alirts_video_encoder_preset_e	视频编码器预设。
video_profile	alirts_video_encoder_profile_e	视频编码器配置文件。
enable_video_enhance	int	开启视频增强，图像更清晰。
video_enhance_strength	float	视频增强强度，取值范围：[1.0,2.0]。
sample_rate	alirts_audio_sample_rate_e	音频采样率，当前仅支持44100 Hz和48000 Hz。
sample_format	alirts_audio_format_e	音频格式，当前仅支持FORMAT_16BIT。
channel	int	音频通道数。
audio_bitrate_bps	int	音频目标码率。
enable_auto_reconnect	bool	是否开启自动重连，不传会取随机值，建议开启。

数据成员	类型	描述
reconnect_timeout	int	重连超时时间，单位：秒，建议设置10秒左右。
url	char *	推流地址，必选，否则无法推流。

- alirts_sei_info_t：用户自定义的SEI信息。

数据成员	类型	描述
uuid[16]	char	自定义SEI的UUID标识。
payload	char *	SEI负载信息。
payload_size	int	SEI负载信息（payload）的长度。

- alirts_frame_t：发送的音视频帧数据。

数据成员	类型	描述
type	alirts_media_type_e	帧类型，目前仅支持MEDIA_TYPE_VIDEO_I420和MEDIA_TYPE_AUDIO_PCM两种格式输入。
data[4]	uint8_t *	帧data buffer，音频只需data[0]。
linesize[4]	int	帧linesize，音频只需linesize[0]。
width	int	视频帧宽。
height	int	视频帧高。
audio_nb_samples	int	音频一帧包含的sample数。
audio_sample_rate	int	音频采样率，当前仅支持44100 Hz和48000 Hz。
audio_channels	int	音频通道数。
timestamp	int64_t	时间戳。
sei	alirts_sei_info_t*	SEI信息。

- alirts_push_info_t：推流统计信息。

数据成员	类型	描述
input_frame_rate	int	视频输入帧率。
encode_frame_rate	int	视频编码帧率。
send_frame_rate	int	视频发送帧率。
current_video_send_pts	int	当前视频发送时间戳。

数据成员	类型	描述
current_audio_send_pts	int	当前音频发送时间戳。
available_send_bandwidth_bps	int	根据带宽估算的当前可用码率，单位：bit/s。
target_enc_bitrate_bps	int	设置给视频编码器的码率，单位：bit/s。
actual_enc_bitrate_bps	int	视频编码器实际编码码率，单位：bit/s。
current_send_bitrate_bps	int	当前音视频发送码率（current_video_send_bitrate_bps和current_audio_send_bitrate_bps之和），单位：bit/s。
current_video_send_bitrate_bps	int	当前视频发送码率，单位：bit/s。
current_video_resend_bitrate_bps	int	当前视频重传码率，单位：bit/s。
current_audio_send_bitrate_bps	int	当前音频发送码率，单位：bit/s。
current_audio_resend_bitrate_bps	int	当前音频重传码率，单位：bit/s。
current_video_enc_psnr	int	当前视频编码平均峰值信噪比。
total_send_bytes	int	从推流开始发送的总字节数。
video_current_sent_lost	int	当前的视频丢包率。
audio_current_sent_lost	int	当前的音频丢包率。
rtt_ms	int	往返延时，单位：ms。
server_ip	char *	连接的服务端IP。
trace_id	char *	用于服务端问题定位的ID。

- alirts_event_handler_t：SDK推流的回调事件通知。

数据成员	类型	描述
user_data	void *	用户输入的自定义信息，之后的回调参数中会携带该user_data。
on_log_data		

数据成员	类型	描述
on_error	回调接口	回调接口的详细信息，请参见回调说明。
on_push_start		
on_push_stop		
on_connection_lost		
on_reconnect_state		

9.4.2. 接口说明

通过阅读本文，你可以了解OBS插件SDK所涉及的接口详情。

目录

API函数	描述
alirts_get_version	获取SDK当前版本号。
alirts_set_log	设置SDK日志输出级别及路径。
alirts_create	创建SDK实例。
alirts_start	启动RTS推流，根据输入配置启动音视频编码器，与CDN服务端建立连接。
alirts_send_frame	发送音视频媒体数据。
alirts_get_statistics_info	获取推流统计信息。
alirts_stop	停止推流，释放编码器与CDN断开连接。
alirts_destroy	释放SDK实例。

详情

- alirts_get_version：获取SDK当前版本号。

```
const char *alirts_get_version();
```

返回说明

返回SDK版本号。

- alirts_set_log：设置SDK日志输出级别及路径。

```
int alirts_set_log(alirts_log_level_e level, const char * dir);
```

参数说明

参数	类型	描述
level	alirts_log_level_e	日志等级。详情请参见SDK文件ali_rts_sdk.h中的alirts_log_level_e定义。
dir	const char *	日志文件存储路径，参数为NULL表示不写本地日志。 **说明** 如果已设置on_log_data日志回调，则SDK不写本地日志，仅回调日志。

返回说明

返回0表示调用成功，小于0表示调用失败。

- alirts_create：创建SDK实例。

```
alirts_handler_t *alirts_create(const alirts_event_handler_t *event_handler);
```

参数说明

参数	类型	描述
event_handler	const alirts_event_handler_t *	事件回调句柄，SDK通过该句柄回调事件。

返回说明

返回SDK句柄表示调用成功，NULL表示调用失败。

- alirts_start：启动RTS推流，根据输入配置启动音视频编码器，与CDN服务端建立连接。

```
int alirts_start(alirts_handler_t *handler, const alirts_config_t *config);
```

参数说明

参数	类型	描述
handler	alirts_handler_t *	SDK实例句柄。
config	const alirts_config_t *	推流配置。

返回说明

返回0表示调用成功，小于0表示调用失败。

说明 此接口调用成功后会触发on_push_start事件回调，之后可以调用alirts_send_frame发送音视频数据。

- alirts_send_frame：发送音视频媒体数据。

```
int alirts_send_frame(const alirts_handler_t *handler, const alirts_frame_t *frame);
```

参数说明

参数	类型	描述
handler	const alirts_handler_t *	SDK实例句柄。
frame	const alirts_frame_t *	音视频帧数据。

返回说明

返回0表示调用成功，小于0表示调用失败。

> 说明 建议在on_push_start事件回调之后调用此接口发送音视频数据。

- alirts_get_statistics_info：获取推流统计信息。

```
int alirts_get_statistics_info(const alirts_handler_t *handler, alirts_push_info_t *info)
;
```

参数说明

参数	类型	描述
handler	const alirts_handler_t *	SDK实例句柄。
info	alirts_push_info_t *	当前推流统计信息。

返回说明

返回0表示调用成功，小于0表示调用失败。

> 注意 请勿频繁调用此接口，建议调用间隔不小于1秒。

- alirts_stop：停止推流，释放编码器与CDN断开连接。

```
void alirts_stop(alirts_handler_t *handler);
```

参数说明

参数	类型	描述
handler	alirts_handler_t *	SDK实例句柄。

> 说明 此接口调用成功后会触发on_push_stop事件回调。

- alirts_destroy：释放SDK实例。

```
void alirts_destroy(alirts_handler_t *handler);
```

参数说明

参数	类型	描述
handler	alirts_handler_t *	SDK实例句柄。

9.4.3. 回调说明

通过阅读本文，你可以了解OBS插件SDK所涉及的回调函数详情。

目录

API	描述
on_log_data	SDK层的日志信息回调。
on_error	SDK错误回调。
on_push_start	推流成功回调。
on_push_stop	推流结束回调。
on_connection_lost	网络连接断开回调。
on_reconnect_state	网络重连状态回调。

详情

- on_log_data：SDK层的日志信息回调。

```
void (*on_log_data)(int level, const char *log, int len)
```

参数	类型	描述
level	int	日志等级。详情请参见SDK文件ali_rts_sdk.h中的alirts_log_level_e定义。
log	const char *	日志字符串。
len	int	日志字符串长度。

> **说明** 如果设置on_log_data，则不写本地日志（仅回调日志）；如果不需要SDK回调日志信息，可将该回调置NULL。

- on_error：SDK错误回调。

```
void (*on_error)(void * user_data, int code, const char *msg)
```

参数	类型	描述
user_data	void *	用户自定义信息。

参数	类型	描述
code	int	错误码。详情请参见SDK文件ali_rts_sdk.h中的alirts_err_code_e定义。
msg	const char *	错误描述信息。

- on_push_start：推流成功回调。

```
void (*on_push_start)(void * user_data)
```

参数	类型	描述
user_data	void *	用户自定义信息。

- on_push_stop：推流结束回调。

```
void (*on_push_stop)(void * user_data)
```

参数	类型	描述
user_data	void *	用户自定义信息。

- on_connection_lost：网络连接断开回调。

```
void (*on_connection_lost)(void * user_data)
```

参数	类型	描述
user_data	void *	用户自定义信息。

- on_reconnect_state：网络重连状态回调。

```
void (*on_reconnect_state)(void * user_data, bool success)
```

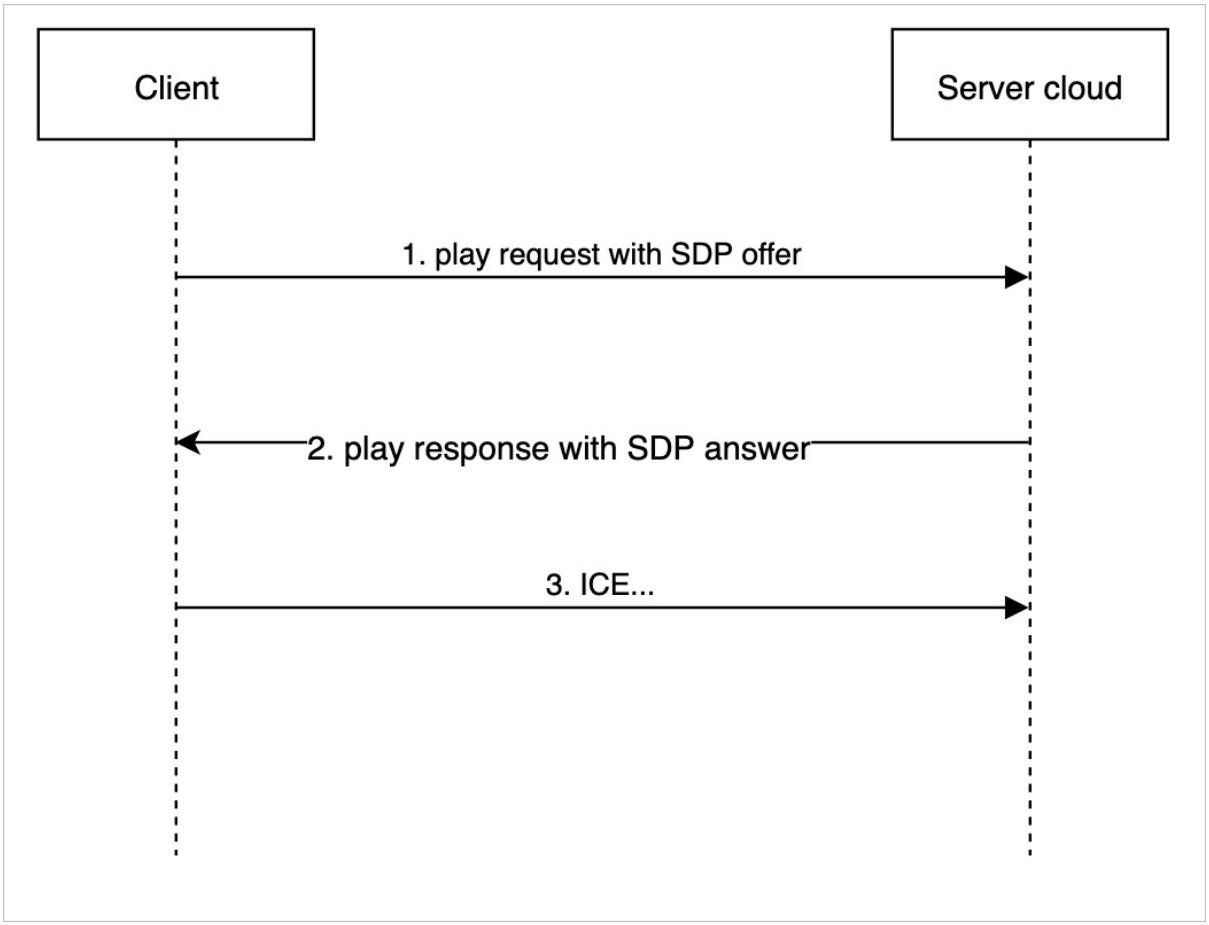
参数	类型	描述
user_data	void *	用户自定义信息。
success	bool	重连是否成功，取值： ○ true：重连成功。 ○ false：重连超时。重连超时配置请参见alirts_config_t。

10.RTS信令协议规范

RTS低延时直播是以WebRTC信令交互方式为基础，借助阿里云直播节点的全球覆盖及优秀调度算法能力，实现低延时直播功能。通过阅读本文，您可以了解RTS信令协议的详细信息。本文适合具有WebRTC基础的开发人员阅读。

信令交互流程

信令交互图如下所示：



信令交互流程

1. 客户端发送offer请求。
   i. 客户端本地创建RTCPeerConnection，设置接收音视频属性Stream Direction，并创建offer SDP。

   ```
   // 开启音视频，recvonly or sendonly
   { offerToReceiveVideo: true, offerToReceiveAudio: true }
   ```

   ii. 客户端向直播服务发送拉流请求，通过HTTPS POST协议方式将JSON格式的请求信息发送至直播服务。协议格式请参见信令协议定义。

   > 说明
   > - `version` 字段为协议版本，当前版本固定为2。
   > - `sdk_version` 字段为SDK版本，您可以自定义该字段。

iii. 生成协议内容后，通过POST方式将信令地址给直播服务完成信令交互，在请求的JSON包体中包含要拉取的流地址。

```
POST /app/streamname?auth=xxx HTTP/1.1
Host: domain
Connection: keep-alive
Content-Length: 2205
Content-Type: application/json
```

> **说明** 信令地址与流地址基本一致，协议头不同，如下所示：
>
> - 信令地址：`https://domain/app/streamname?auth=xxx`。
> - 流地址：`artc://domain/app/streamname?auth=xxx`。

2. 服务端SDP Answer响应。

   直播服务端校验安全性后生成SDP Answer, 节点信息封装在响应包体中返回给客户端。协议格式请参见信令协议定义。

3. 客户端ICE建联。

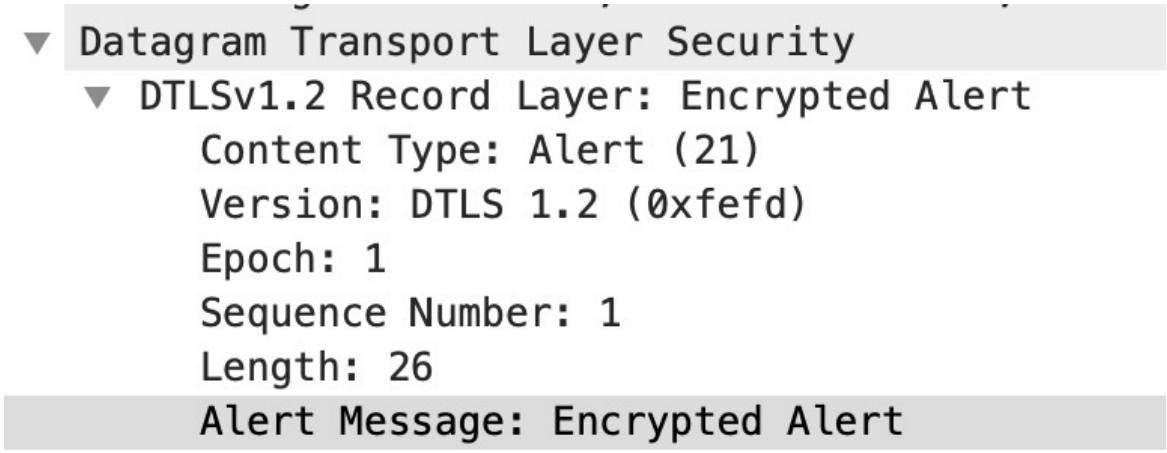
   i. 客户端收到SDP Answer响应后，设置到RTCPeerConnection中。

   ```
   peerConnection.setRemoteDescription(new RTCSessionDescription(answer.jsep));
   ```

   ii. RTCPeerConnection启动ICE建连流程以及后续的DTLS流程，媒体通道建立成功后可以获取到直播服务输出的媒体流，实现WebRTC标准接入拉流播放。

4. 断开连接。

   客户端需要断开连接停止推流或播放，使用DTLS的Alert消息表达。



H5 Demo示例

```
// Create peer connection and local offer sdp.
peerConnection = new RTCPeerConnection();
peerConnection.onicecandidate = iceCandidateCallback;
peerConnection.ontrack = remoteStreamCallback;
peerConnection.createOffer({ offerToReceiveVideo: true, offerToReceiveAudio: true })
      .then(signaling_pull).catch(errorHandler);
// CDN live post pull stream request.
function signaling_pull(offer_sdp) {
  console.log('local offer sdp', offer_sdp);
  peerConnection.setLocalDescription(offer_sdp).then(function() {
    // Get pull stream url.
    var stream_url = $("#stream_url").val();
    console.log("stream url:" , stream_url);
    // Add sdk and protocol versions.
    var protocol_version = 2;
    var sdk_version = "0.0.1";
    $.ajax({url: stream_url, data: JSON.stringify({
          mode: "live",
          version: protocol_version,
          sdk_version: sdk_version,
          jsep:description,
      }),
      type: "post",
      success:function(result){
          var signal = JSON.parse(result);
          peerConnection.setRemoteDescription(new RTCSessionDescription(signal.jsep)).then(
function() {
              console.log("get remote answer sdp: ", signal.jsep.sdp);
          }).catch(errorHandler);
      }});
  }).catch(errorHandler);
}
```

信令协议定义

RTS信令协议通道为HTTPS（短链接），格式为JSON。协议详情如下所示：

请求示例

```
Request:
{
    "version":2,
    "sdk_version":"0.0.1",
    "mode":"live",
    "pull_streams":[
        {
            "url":"artc://demo.aliyundoc.com/liveApp****/liveStream****",
            "amsid":[
                "rts audio"
            ],
            "vmsid":[
                "rts video"
            ]
        }
    ],
    "jsep":{
        "type":"offer",
        "sdp":"v=0\n\ro=- 6839248142876176651 2 IN IP4 127.0.0.1\n\rs=-\n\r以下省略"
    }
}
```

参数	类型	是否必选	描述
mode	string	是	模式，取值：live。
version	int	是	协议版本号，取值：2。
push_stream	string	否	推流URL。
pull_streams	[]object	否	拉流对象，支持多个拉流。详情请参见下表。
sdk_version	string	否	SDK版本号。
jsep.type	string	是	SDP类型，取值：offer。
jsep.sdp	string	是	SDP描述。

pull_stream参数

字段	类型	是否必填	描述
url	string	是	拉流URL，例如：`artc://<拉流地址>`。
amsid	[]string	是	拉流音频msid，直播场景取值`rts audio`。
vmsid	[]string	是	拉流视频msid，直播场景取值`rts video`。

相应示例

```
Response:
{
    "trace_id":"2_1591173296_101.227.0.169_702080732320_dec327eb6eed0e0b07b349c8a565****",
    "code":200,
    "jsep":{
        "type":"answer",
        "sdp":"v=0\r\no=- 1591173291 2 IN IP4 127.0.0.1\n\r 以下省略"
    }
}
```

字段	类型	是否必需	描述
code	int	是	返回码。正确返回200，错误码请参见下表。
trace_id	string	是	全局唯一请求标识ID。由CDN生成，请尽量做好保存，用于对请求进行定位和问题排查。
jsep.type	string	是	SDP类型，取值：answer。
jsep.sdp	string	是	直播CDN回源拉流生成SDP。

错误码描述

错误码	描述
403	鉴权失败。
404	流不存在。
611	需要客户端强制降级到TCP播放。
302	需要客户端向新的服务地址发起信令请求。

增强的SDP

在信令交互中，SDP用来描述媒体信息。通用的SDP协商是基于RFC 4566展开的，阿里云RTS在其基础上扩展出了更多丰富的语义，兼容直播行业的特点，支持更多的音视频封装和通信协议，突破了WebRTC仅支持音频OPUS、不支持视频B帧等的窘境，满足了如今流媒体业界日益扩大的协议族。

支持AAC音频

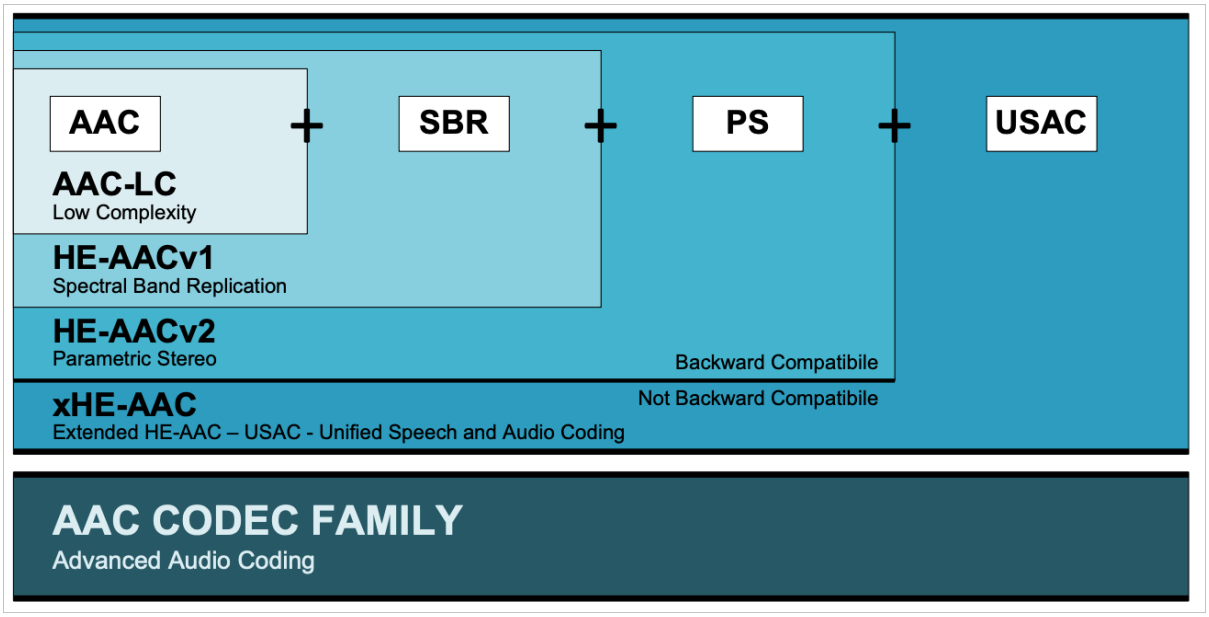
RTS支持传输任何能在RTMP中传输的AAC格式的音频，包括AAC-LC、HE-AACv1和HE-AACv2。AAC格式详情请参见ISO IEC 14496-3。

RTS支持以LATM格式进行AAC音频传输。LATM可根据传输中的音频是否自带音频编码信息来决定是带内（每个音频都发）或带外（只发一次）音频编码信息，即由Audio Mux Element中的muxconfigPresent决定AudioSpecificConfig是带内或带外传输。因此，LATM相对ADTS更灵活，其AudioSpecificConfig如果保持不变，则可以通过SDP会话先传输StreamMuxConfig（AudioSpecificConfig）信息。LATM详情请参见Application-Bulletin_AAC-Transport-Format。

在信令交互阶段，RTS会解析推流侧音频编码信息，并在协商响应中返回对应的信息。如下所示：

offer SDP	answer SDP		
	AAC-LC	HE-AACv1	HE-AACv2
	AudioSpecificConfig = 0x1210	AudioSpecificConfig = 2b920800	AudioSpecificConfig = eb8a0800
m=audio 9 UDP/RTP/AVPF 120 96 a=rtpmap:120 MP4A-LATM/44100/2	a=rtpmap:120 MP4A-LATM/44100/2 a=fmtp:120 cpresent=0;profile-level-id=1;object=2;config=400024203fc0	a=rtpmap:120 MP4A-LATM/44100/2 a=fmtp:120 cpresent=0;profile-level-id=1;object=2;config=4000572410003fc0;SBR-enabled=1	a=rtpmap:120 MP4A-LATM/44100/2 a=fmtp:120 cpresent=0;object=2;profile-level-id=1;config=4001d71410003fc0;PS-enabled=1;SBR-enabled=1

此处，通过在answer SDP音频MP4A-LATM的fmtp中添加 `SBR-enabled=1` 表示AAC-HE，添加 `SBR-enabled=1` 和 `PS-enabled=1` 表示HE-AACv2。由于从AAC-LC到HE-AACv2有技术递进关系，因此在表示fmtp时，递进地增加SBR、PS的标识来表示不同的AAC格式。此外，在fmtp中添加 `config=StreamMuxConfig` ，其由推流AudioSpecificConfig信息组装而来，具体地描述了音频编码的各种参数信息，客户端可以各取所需。



更多信息，请参见AAC-LC / HE-AACv1 / HE-AACv2 Encoder Parameters。

支持H.265视频

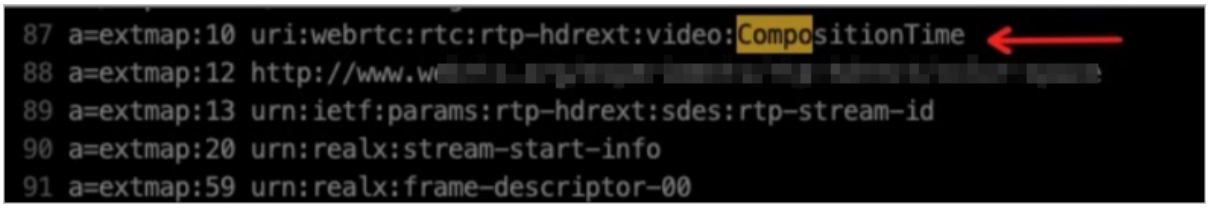
RTS会解析推流侧视频编码信息（H.264、H.265），在answer SDP中根据推流侧实际的编码方式，返回 `H.264` 或 `H.265` 。

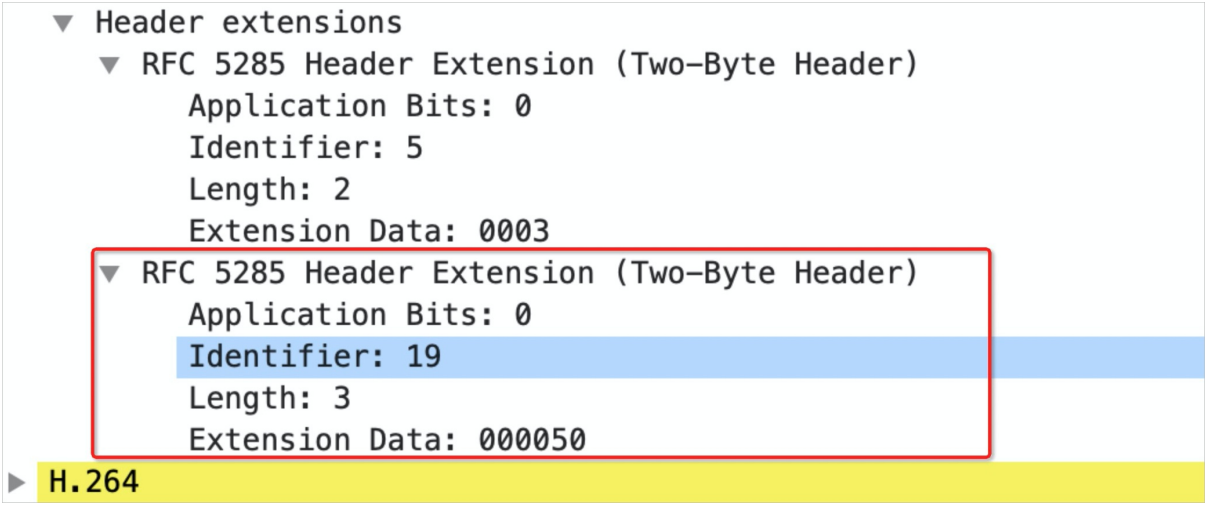
编码类型	offer SDP	answer SDP
H.265	`a=rtpmap:102 H265/90000`	`a=rtpmap:122 H265/90000` `a=fmtp:122`

支持B帧视频

在信令交互阶段，客户端可以在offer SDP中增加字段标识其是否支持解析B帧视频。例如：在视频fmtp中增加 `BFrame-enabled = 1` 标识以表示支持B帧，此时， `RTP timestamp = PTS` ，sequence number递增，正常情况下客户端按照sequence number解码即可。如果不支持B帧，RTS可以对视频源流进行转码，去掉B帧。

此外，RTS还支持额外返回CTS扩展头以支持需要精确计算DTS的客户端（PTS=DTS+CTS），如果offer SDP中带有 `a=extmap:{$id} uri:webrtc:rtc:rtp-hdrext:video:CompositionTime` ，RTS会在每一帧视频的第一个RTP包上增加 `extension identifier = {$id}` 的CTS扩展头， `id` 由offer SDP来决定。offer SDP片段及拉流抓包示例如下所示：





RTS给予客户端充分的表达能力，将决定权下放给用户，让其决定是否接受视频带B帧，以及是否需要额外的CTS信息，构建更通用的通信能力。

MSID相关概念

关于MSID详细信息，请参见The Msid Mechanism。