# 开源直播服务srs详细介绍(wiki整理)

目录

[开源直播服务srs详细介绍(wiki整理) 0](#_Toc26097915)

[1. 简介 2](#_Toc26097916)

[1.1. SRS功能 2](#_Toc26097917)

[1.2. 架构 2](#_Toc26097918)

[2. 流接入方式 3](#_Toc26097919)

[2.1. 推送RTMP到SRS 4](#_Toc26097920)

[2.2. 分发RTMP流 4](#_Toc26097921)

[2.2.1. 应用场景 4](#_Toc26097922)

[2.2.2. WIKI 5](#_Toc26097923)

[2.2.3. 补充:SRS不支持点播 5](#_Toc26097924)

[2.2.4. 补充:常用的第三方的推流与播放工具 6](#_Toc26097925)

[2.3. 推送RTSP/UDP/FLV 到SRS 6](#_Toc26097926)

[2.3.1. Push MPEG-TS over UDP 7](#_Toc26097927)

[2.3.2. Push RTSP to SRS 7](#_Toc26097928)

[2.3.3. Push HTTP FLV to SRS 8](#_Toc26097929)

[2.3.4. WIKI 8](#_Toc26097930)

[2.4. 拉取流到SRS 9](#_Toc26097931)

[2.4.1. 补充:RTSP开源项目 10](#_Toc26097932)

[2.4.2. WIKI 11](#_Toc26097933)

[2.5. RTMP流的低延时配置 11](#_Toc26097934)

[2.5.1. 低延时直播应用 11](#_Toc26097935)

[2.5.2. 应用场景 11](#_Toc26097936)

[2.5.3. RTMP和延时 12](#_Toc26097937)

[2.5.4. 累积延迟 13](#_Toc26097938)

[2.5.5. 低延时配置 14](#_Toc26097939)

[2.5.6. WIKI 15](#_Toc26097940)

[3. 流变换 15](#_Toc26097941)

[3.1. 将RTMP流转码 15](#_Toc26097942)

[3.1.1. 应用场景 15](#_Toc26097943)

[3.1.2. 工作流程 16](#_Toc26097944)

[3.1.3. 配置 17](#_Toc26097945)

[3.1.4. 转码规则 17](#_Toc26097946)

[3.1.5. Transcode on ARM cpu 19](#_Toc26097947)

[3.1.6. FFMPEG Transcode the Stream by Flash encoder 20](#_Toc26097948)

[3.1.7. 补充:测试效果图 20](#_Toc26097949)

[3.1.8. WIKI 20](#_Toc26097950)

[流截图 20](#_Toc26097951)

[WIKI 21](#_Toc26097952)

[补充:测试效果图 21](#_Toc26097953)

[转发给其他服务器(Forward) 21](#_Toc26097954)

[转封装成HTTP直播流 22](#_Toc26097955)

[转封装成HLS 25](#_Toc26097956)

[转封装成HDS 28](#_Toc26097957)

[录制成FLV 28](#_Toc26097958)

[分发方式比较 29](#_Toc26097959)

[集群与CDN相关功能 32](#_Toc26097960)

[RTMP多级集群 32](#_Toc26097961)

[VHOST虚拟服务器 37](#_Toc26097962)

[无中断服务Reload 46](#_Toc26097963)

[HTTP-FLV集群 47](#_Toc26097964)

[Kafka对接 48](#_Toc26097965)

[应用接口 48](#_Toc26097966)

[HTTP回调 48](#_Toc26097967)

[安全策略Security 54](#_Toc26097968)

[HTTP API接口 57](#_Toc26097969)

[RTMP测速 71](#_Toc26097970)

[其他功能 74](#_Toc26097971)

[Send Minimal Interval 74](#_Toc26097972)

[Reduce Sequence Header 75](#_Toc26097973)

[Publish 1st Packet Timeout 75](#_Toc26097974)

[Publish Normal Timeout 75](#_Toc26097975)

[Debug SRS Upnode 75](#_Toc26097976)

[UTC Time 76](#_Toc26097977)

[HLS TS Floor 76](#_Toc26097978)

[HLS Wait Keyframe 76](#_Toc26097979)

[HttpHooks On HLS Notify 76](#_Toc26097980)

[TCP NoDelay 77](#_Toc26097981)

[ATC Auto 77](#_Toc26097982)

[NGINX RTMP EXEC 77](#_Toc26097983)

[SRS与其他流媒体比较 78](#_Toc26097984)

[流发分 Stream Delivery 78](#_Toc26097985)

[集群 Cluster 79](#_Toc26097986)

[流处理服务 Stream Service 79](#_Toc26097987)

[Efficiency 79](#_Toc26097988)

[Stream Caster 79](#_Toc26097989)

[Debug System 80](#_Toc26097990)

[Docs 80](#_Toc26097991)

[Others 80](#_Toc26097992)

[市面主要的流媒体服务器对比 80](#_Toc26097993)

[SRS商业版（BMS）简介 93](#_Toc26097994)

[常用的直播平台网站 96](#_Toc26097995)

# 简介

该文档介绍了开源直播服务器SRS架构、功能以及应用、SRS与其他直播服务器的比较、SRS商业版的功能简介。内容主要来自于SRS WIKI整理，以及网上有关博客内容,有些地方加了补充内容。出掉WIKI上一些讲解配置使用的内容，需要了解这些内容可点击文档中相关wiki链接。

整理这些内容的目的是为了方便查阅，阅读后能对SRS比较全面的了解，特别是对刚接解SRS的使用者。

## SRS功能

SRS提供了丰富的接入方案将RTMP流接入SRS，包括推送RTMP到SRS、推送RTSP/UDP/FLV到SRS、拉取流到SRS。SRS还支持将接入的RTMP流进行各种变换，譬如将RTMP流转码、流截图、转发给其他服务器、转封装成HTTP-FLV流、转封装成HLS、转封装成HDS、录制成FLV。SRS包含支大规模集群如CDN业务的关键特性，譬如RTMP多级集群、VHOST虚拟服务器、无中断服务Reload、HTTP-FLV集群、Kafka对接。此外，SRS还提供丰富的应用接口，包括HTTP回调、安全策略Security、HTTP API接口、RTMP测速。

## 架构

+---------+ +----------+

| Publish | | Deliver |

+---|-----+ +----|-----+

+----------------------+-------------------------+----------------+

| Input | SRS(Simple RTMP Server) | Output |

+----------------------+-------------------------+----------------+

| Encoder(1) | +-> RTMP/HDS --------+-> Flash player |

| (FMLE,FFMPEG, -rtmp-+->-+-> HLS/HTTP ---------+-> M3u8 player |

| Flash,XSPLIT, | +-> FLV/MP3/Aac/Ts ---+-> HTTP player |

| ......) | +-> Fowarder ---------+-> RTMP server |

| | +-> Transcoder -------+-> RTMP server |

| | +-> DVR --------------+-> Flv file |

| | +-> BandwidthTest ----+-> flash |

+----------------------+ | |

| MediaSource(2) | | |

| (RTSP,FILE, | | |

| HTTP,HLS, --pull-+->-- Ingester(3) -(rtmp)-+-> SRS |

| Device, | | |

| ......) | | |

+----------------------+ | |

| MediaSource(2) | | |

| (RTSP,FILE, | | |

| HTTP,HLS, --push-+->-- Streamer(4) -(rtmp)-+-> SRS |

| Device, | | |

| ......) | | |

+----------------------+-------------------------+----------------+

# 流接入方式

可以通过以下几种方式将音视频流接入到SRS:

1.客户端使用编码器将音视频编码成h264 ,aac,mp3使用rtmp协议推送到SRS;

2.使用SRS采取器(Ingester),将各种源(流，文件，设备等)拉过来后，推送给自己;

3.SRS作为服务器侦听并接收其他协议的流（譬如RTSP，MPEG-TS over UDP等等），将协议的流转换成RTMP推送给自己;

## 推送RTMP到SRS

这是SRS的核心功能之一，通过客户端使用rtmp协议将音视频流推送srs。然后使用rtmp播放器连接srs后观看， Srs目前支持视频H264, 音频AAC、MP3。

## 分发RTMP流

SRS（Simple RTMP Server）分发RTMP也是核心功能之一，srs的主要定位就是分发RTMP低延时流媒体，同时支持分发HLS流。

### 应用场景

RTMP是PC-flash支持最完善的流分发方式，主要的应用场景包括：

* 无插件流媒体应用：十年前各种浏览器插件大行其道，最后adobe的flash一统天下，现在如何观看视频还需要用户装插件，已经是非常罕见的事情。打开浏览器就能用，不用装插件，这是RTMP的最基本的应用方式。
* 适配广泛的播放器：如果没有专业的flash开发人员，那么RTMP会是个很好的选择，只要3行代码就能完成一个播放器，和html5的video标签一样方便。HDS/HLS在PC上，都需要库支持，N行代码很麻烦。
* 苛刻的稳定性支持：RTMP服务器能365x24提供服务，当然http服务器也可以。客户端的稳定性呢？RTMP在flash中连续播放10天 没有问题，flash如果播放HTTP流就真的很难讲。如果在PC上需要客户端长时间播放，稳定播放，选择RTMP会是最佳选择。
* 稳定的较小延迟：RTMP延迟在0.8-3秒，能应用于交互式直播，视频会议，互动式直播等等。如果对延时有一定要求，就不要选择HLS，RTMP会是最佳选择。
* 通用接入标准：RTMP是编码器到服务器的实际标准协议，所有编码器都支持RTMP推送流。选择RTMP作为直播接入协议，能适配多种编码器，不至于绑定到一种编码器。如果服务器只能接入HTTP FLV流，像某些公司做的私有协议，那么对接通用编码器就有问题。何必闭门造车？！绑定用户的方式在于良好的客户关系和优秀的软件质量，而不是上了贼船就 下不了船了。

SRS直播将RTMP作为基本协议，以各种方式转码为RTMP后输入到SRS，输出为RTMP和HLS，支持广泛的客户端和各种应用场景。

### WIKI

<https://github.com/ossrs/srs/wiki/v1_CN_SampleRTMP>

### 补充:SRS不支持点播

SRS不支持点播，一般点播不会使用RTMP作为点播协议，目前点播以HTTP协议为主。Nginx rtmp可以使用rtmp点播.不过在测试过程中发现，托放支持不是很好,国内很少使用rtmp点播，现在点播大都使用http。

### 补充:常用的第三方的推流与播放工具

常用的第三方的推流工具有OBS, XSplit, FMLE, video\_broadcast++(Android)，Broadcast\_Me(iPhone),除上面这些工具外，还可以使用ffmpeg

常用播放器: flash player, ffplay, vlc等。

## 推送RTSP/UDP/FLV 到SRS

Streamer（流服务）是SRS作为服务器侦听并接收其他协议的流（譬如RTSP，MPEG-TS over UDP等等），将这些协议的流转换成RTMP推送给自己，以使用RTMP/HLS/HTTP分发流。

常见的应用场景包括：

Push MPEG-TS over UDP to SRS：通过UDP协议，将MPEG-TS推送到SRS，分发为RTMP/HLS/HTTP流。

Push RTSP to SRS：通过RTSP协议，将流推送到SRS，分发为RTMP/HLS/HTTP流。

POST FLV over HTTP to SRS: 通过HTTP协议，将FLV流POST到SRS，分发为RTMP/HLS/HTTP流。

备注：Streamer将其他支持的协议推送RTMP给SRS后，所有SRS的功能都能支持。譬如，推RTSP流给 Streamer，Streamer转成RTMP推送给SRS，若vhost是edge，SRS将RTMP流转发给源站。或者将RTMP流转码，或者直接转发。另外，所有分发方法都是可用的，譬如推RTSP流给Streamer，Streamer转成RTMP推给SRS，以RTMP/HLS/HTTP分发。

### Push MPEG-TS over UDP

SRS可以侦听一个udp端口，编码器将流推送到这个udp端口（SPTS）后，SRS会转成一路RTMP流。后面RTMP流能支持的功能都支持。

配置如下，参考conf/push.mpegts.over.udp.conf：

# the streamer cast stream from other protocol to SRS over RTMP.

# @see https://github.com/ossrs/srs/tree/develop#stream-architecture

stream\_caster {

enabled on;

caster mpegts\_over\_udp;

output rtmp://127.0.0.1/live/livestream;

listen 1935;

}

参考：[https://github.com/ossrs/srs/issues/250#issuecomment-72321769](https://github.com/ossrs/srs/issues/250" \l "issuecomment-72321769)

### Push RTSP to SRS

SRS可以侦听一个tcp端口，编码器将流推送到这个tcp端口（RTSP）后，SRS会转成一路RTMP流。后面RTMP流能支持的功能都支持。

配置如下，参考conf/push.rtsp.conf：

# the streamer cast stream from other protocol to SRS over RTMP.

# @see https://github.com/ossrs/srs/tree/develop#stream-architecture

stream\_caster {

enabled on;

caster rtsp;

output rtmp://127.0.0.1/[app]/[stream];

listen 554;

rtp\_port\_min 57200;

rtp\_port\_max 57300;

}

参考：[https://github.com/ossrs/srs/issues/133#issuecomment-75531884](https://github.com/ossrs/srs/issues/133" \l "issuecomment-75531884)

### Push HTTP FLV to SRS

SRS可以侦听一个HTTP端口，编码器将流推送到这个http端口后，SRS会转成一路RTMP流。所有RTMP流的功能都能支持。

配置如下，参考conf/push.flv.conf：

# the streamer cast stream from other protocol to SRS over RTMP.

# @see https://github.com/ossrs/srs/tree/develop#stream-architecture

stream\_caster {

enabled on;

caster flv;

output rtmp://127.0.0.1/[app]/[stream];

listen 8936;

}

这个配置时，客户端推流的地址，例如：http://127.0.0.1:8936/live/sea.flv  
播放RTMP流地址是：rtmp://127.0.0.1/live/sea  
播放HLS流地址是：http://127.0.0.1:8080/live/sea.m3u8

注意：需要配置HTTP服务器和HLS，参考conf/push.flv.conf

### WIKI

<https://github.com/ossrs/srs/wiki/v2_CN_Streamer>

## 拉取流到SRS

采集(Ingest)指的是将文件（flv，mp4，mkv，avi，rmvb等等），流（RTMP，RTMPT，RTMPS，RTSP，HTTP，HLS等等），设备等的数据，转封装为RTMP流（若编码不是h264/aac则需要转码），推送到SRS。采集基本上就是使用FFMPEG作为编码器，或者转封装器，将外部流主动抓取到SRS。

采集的主要应用场景包括：

1.虚拟直播：将文件编码为直播流。可以指定多个文件后，SRS会循环播放。

2.RTSP摄像头对接：以前安防摄像头都支持访问RTSP地址，RTSP 无法在互联网播放。可以将RTSP采集后，以RTMP推送到SRS，后面的东西就不用讲了。

3.直接采集设备：SRS采集功能可以作为编码器采集设备上的未压缩图像数据，譬如video4linux和alsa设备，编码为h264/aac后输出RTMP到SRS。

4.将HTTP流采集为RTMP：有些老的设备，能输出HTTP的ts或FLV流，可以采集后转封装为RTMP，支持HLS输出。

总之，采集的应用场景主要是“SRS拉流”；能拉任意的流，只要ffmpeg支持；不是h264/aac都没有关系，ffmpeg能转码。SRS默认是支持“推流”，即等待编码器推流上来，可以是专门的编码设备，FMLE，ffmpeg，xsplit，flash, obs等等。

如此，SRS的接入方式可以是“推流到SRS”和“SRS主动拉流”，基本上作为源站的功能就完善了。

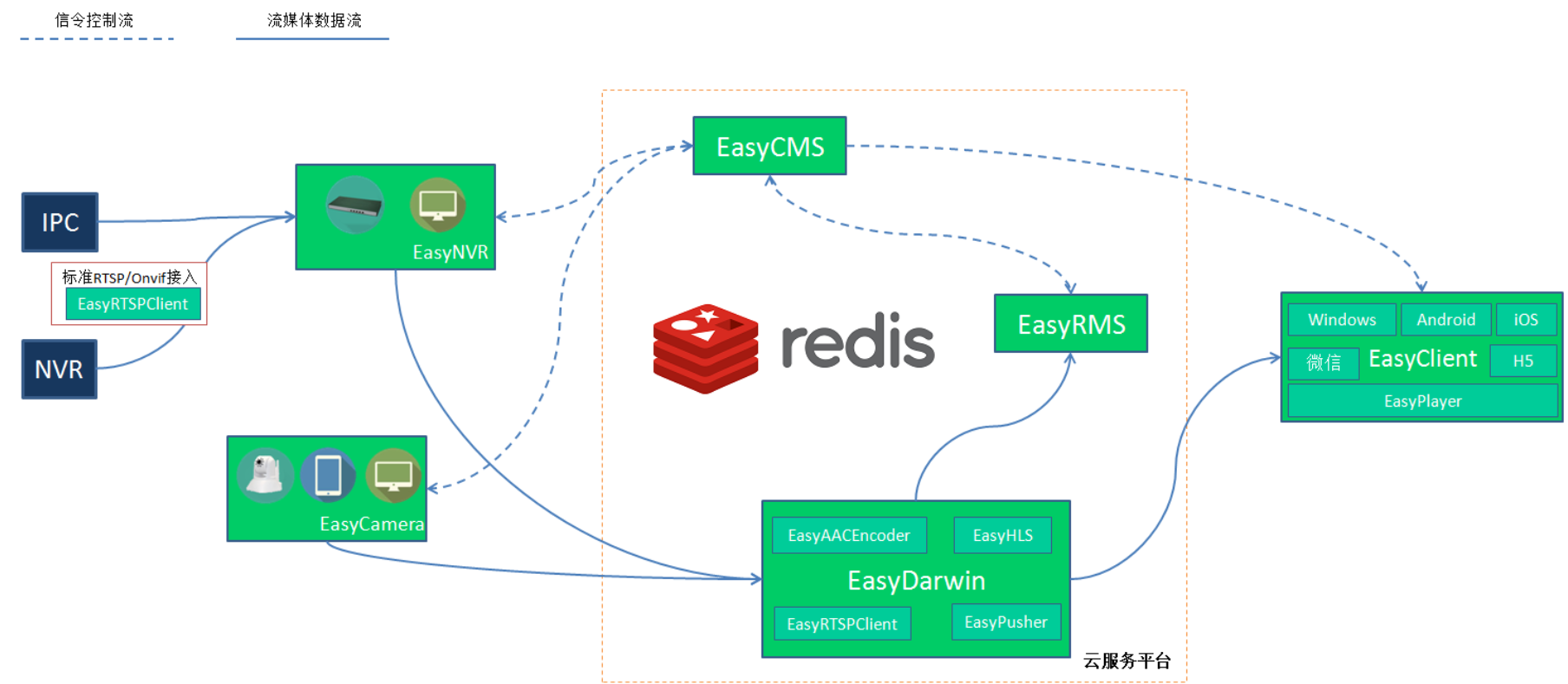
### 补充:RTSP开源项目

“推送RTSP/UDP/FLV 到SRS”功能目前版本SRS中，还是实验性功能，不是特别稳定，测试rtsp过程中, 声音正常，视频有花屏，SRS商业版应该这些功能已经稳定，因为SRS原作者2016春节后更新很慢，现在主要做商业版。

RTSP协议主要在安防监控，iptv项目使用特别多。现在市场上网络摄象机（IPC）中rtsp是标配。而现在互联网播放端通用的基本是rtmp,hls，移动端通用的主要是hls。未来IPC走互联网越来越多。所以对于通用型直播平台来说，多协议转换是非常重要的功能。

常用的rtsp开源流媒体服务器有live555（http://www.live555.com）,Easydarwin（http://www.easydarwin.org）等。

目前EasyDarwin流媒体云平台整套解决方案包括有：EasyCMS(中心管理服务、跨平台、支持分布式部署)，EasyDarwin(流媒体服务、跨平台、支持分布式部署)，EasyRMS(云录像服务、跨平台、支持分布式部署)，EasyCamera(开源流媒体云摄像机方案、支持ARM、Android)，EasyNVR(将标准RTSP/Onvif摄像机接入到云平台)，EasyPlayer（流媒体播放器），EasyClient（云平台客户端）， 以及周边众多工具库(EasyHLS / EasyRTSPClient / EasyPusher / EasyAACEncoder)，后续也将继续扩展的录像、回放等多种服务和工具集，各个功能单元既可以独立使用于项目，又可以整体使用，形成一个完整、简单、易用、高效的流媒体解决方案。



### WIKI

<https://github.com/ossrs/srs/wiki/v1_CN_Ingest>

## RTMP流的低延时配置

RTMP流的延时一般在1-3秒，比HLS的延时小，hls延时10-30秒左右。

### 低延时直播应用

直播应用中，RTMP和HLS基本上可以覆盖所有客户端观看（参考：[DeliveryHLS](https://github.com/ossrs/srs/wiki/v1_CN_DeliveryHLS)），HLS主要是延时比较大，RTMP主要优势在于延时低。

### 应用场景

低延时应用场景包括：

* 互动式直播：譬如2013年大行其道的美女主播，游戏直播等等各种主播，流媒体分发给用户观看。用户可以文字聊天和主播互动。
* 视频会议：SRS的DEMO就有视频会议应用，我们要是有同事出差在外地，就用这个视频会议开内部会议。其实会议1秒延时无所谓，因为人家讲完话后，其他人需要思考，思考的延时也会在1秒左右。当然如果用视频会议吵架就不行。
* 其他：监控，直播也有些地方需要对延迟有要求，互联网上RTMP协议的延迟基本上能够满足要求。

### RTMP和延时

RTMP的特点如下：

* Adobe支持得很好：RTMP实际上是现在编码器输出的工业标准协议，基本上所有的编码器（摄像头之类）都支持RTMP输出。原因在于PC市场巨大，PC主要是Windows，Windows的浏览器基本上都支持flash，Flash又支持RTMP支持得灰常好。
* 适合长时间播放：因为RTMP支持的很完善，所以能做到flash播放RTMP流长时间不断流，当时测试是100万秒，即10天多可以连续播放。 对于商用流媒体应用，客户端的稳定性当然也是必须的，否则最终用户看不了还怎么玩？我就知道有个教育客户，最初使用播放器播放http流，需要播放不同的 文件，结果就总出问题，如果换成服务器端将不同的文件转换成RTMP流，客户端就可以一直播放；该客户走RTMP方案后，经过CDN分发，没听说客户端出 问题了。
* 延迟较低：比起YY的那种UDP私有协议，RTMP算延迟大的（延迟在1-3秒），比起HTTP流的延时（一般在10秒以上）RTMP算低延时。 一般的直播应用，只要不是电话类对话的那种要求，RTMP延迟是可以接受的。在一般的视频会议（参考SRS的视频会议延时）应用中，RTMP延时也能接 受，原因是别人在说话的时候我们一般在听，实际上1秒延时没有关系，我们也要思考（话说有些人的CPU处理速度还没有这么快）。
* 有累积延迟：技术一定要知道弱点，RTMP有个弱点就是累积误差，原因是RTMP基于TCP不会丢包。所以当网络状态差时，服务器会将包缓存起 来，导致累积的延迟；待网络状况好了，就一起发给客户端。这个的对策就是，当客户端的缓冲区很大，就断开重连。当然SRS也提供配置。

### 累积延迟

除了GOP-Cache，还有一个有关系，就是累积延迟。SRS可以配置直播队列的长度，服务器会将数据放在直播队列中，如果超过这个长度就清空到最后一个I帧：

vhost your\_vhost {

# the max live queue length in seconds.

# if the messages in the queue exceed the max length,

# drop the old whole gop.

# default: 30

queue\_length 10;

}

当然这个不能配置太小，譬如GOP是1秒，queue\_length是1秒，这样会导致有1秒数据就清空，会导致跳跃。

有更好的方法？有的。延迟基本上就等于客户端的缓冲区长度，因为延迟大多由于网络带宽低，服务器缓存后一起发给客户端，现象就是客户端的缓冲区变大了，譬如NetStream.BufferLength=5秒，那么说明缓冲区中至少有5秒数据。

处理累积延迟的最好方法，是客户端检测到缓冲区有很多数据了，如果可以的话，就重连服务器。当然如果网络一直不好，那就没有办法了。

### 低延时配置

考虑GOP-Cache和累积延迟，推荐的低延时配置如下（参考min.delay.com）：

# the listen ports, split by space.

listen 1935;

vhost \_\_defaultVhost\_\_ {

# whether cache the last gop.

# if on, cache the last gop and dispatch to client,

# to enable fast startup for client, client play immediately.

# if off, send the latest media data to client,

# client need to wait for the next Iframe to decode and show the video.

# set to off if requires min delay;

# set to on if requires client fast startup.

# default: on

gop\_cache off;

# the max live queue length in seconds.

# if the messages in the queue exceed the max length,

# drop the old whole gop.

# default: 30

queue\_length 10;

}

当然，服务器的性能也要考虑，不可以让一个SRS进程跑太高带宽，一般CPU在80%以下不会影响延迟，连接数参考[性能](https://github.com/ossrs/srs/wiki/v1_CN_Performance)。

### WIKI

<https://github.com/ossrs/srs/wiki/v3_CN_SampleRealtime>

# 流变换

SRS还支持将接入的RTMP流进行各种变换，譬如将RTMP流转码、流截图、转发给其他服务器、转封装成HTTP-FLV流、转封装成HLS、转封装成HDS、录制成FLV等。

## 将RTMP流转码

SRS通过FFMPEG对RTMP直播流转码，SRS在收到编码器推送的直播流后，可以对直播流进行转码，输出RTMP流到服务器（也可以到SRS自己）。如使用该功能将直播流转换多码率流，或者给流添加水印等。

### 应用场景

转码的重要应用场景包括：

1. 推送一路高码率，转多路输出。譬如：游戏直播中，推送一路1080p流到SRS，SRS可以转码输出1080p/720p/576p多路，低码率可以给移动设备观看。这样节省了推流带宽（一般源站为BGP带宽，很贵），也减轻了客户端压力（譬如客户端边玩游戏边直播）。
2. 支持多屏输出。譬如：网页推流（主播）编码为vp6/mp3或speex，推流到SRS后无法支持HLS（要求h264+aac），可以转码成h264+aac后切片成HLS或者推送到其他服务器再分发。
3. 加水印。适用于需要对流进行加水印的情况，譬如打上自己的logo。SRS支持文字水印和图片水印，也可以支持视频作为水印，或者将两路流叠加（参考ffmpeg的用法）。
4. 截图。
5. 其他滤镜：SRS支持所有ffmpeg的滤镜。

### 工作流程

SRS转码的主要流程包括：

1. 编码器推送RTMP流到SRS的vhost。
2. SRS的vhost若配置了转码，则进行转码。
3. 转码后，按照配置，推送到SRS本身或者其他RTMP服务器。

### 配置

SRS可以对vhost的所有的流转码，或者对某些app的流转码，或者对某些流转码。

对app或流转码时，只要在transcode后面加app和stream就可以。譬如：

listen 1935;

vhost \_\_defaultVhost\_\_ {

# 对app为live的所有流转码

transcode live{

}

}

以及对指定的流转码：

listen 1935;

vhost \_\_defaultVhost\_\_ {

# 对app为live并且流名称为livestream的流转码

transcode live/livestream{

}

}

### 转码规则

SRS的转码参数全是FFMPEG的参数，有些参数SRS做了自定义，见下表。

| **SRS参数** | **FFMPEG参数** | **实例** | **说明** |
| --- | --- | --- | --- |
| vcodec | vcodec | ffmpeg ... -vcodec libx264 ... | 指定视频编码器 |
| vbitrate | b:v | ffmpeg ... -b:v 500000 ... | 输出的视频码率 |
| vfps | r | ffmpeg ... -r 25 ... | 输出的视频帧率 |
| vwidth/vheight | s | ffmpeg ... -s 400x300 -aspect 400:300 ... | 输出的视频宽度x高度，以及宽高比 |
| vthreads | threads | ffmpeg ... -threads 8 ... | 编码线程数 |
| vprofile | profile:v | ffmpeg ... -profile:v high ... | 编码x264的profile |
| vpreset | preset | ffmpeg ... -preset medium ... | 编码x264的preset |
| acodec | acodec | ffmpeg ... -acodec libfdk\_aac ... | 音频编码器 |
| abitrate | b:a | ffmpeg ... -b:a 70000 ... | 音频输出码率。libaacplus：16-72k。libfdk\_aac没有限制。 |
| asample\_rate | ar | ffmpeg ... -ar 44100 ... | 音频采样率 |
| achannels | ac | ffmpeg ... -ac 2 ... | 音频声道 |

另外，还有三个是可以加其他ffmpeg参数：

* vfilter：添加在vcodec之前的滤镜参数。
* vparams：添加在vcodec之后，acodec之前的视频编码参数。
* aparams：添加在acodec之后，-y之前的音频编码参数。

SRS的ffmpeg转码器可以使用ffmpeg所有功能如:

1 不转码只复制流：可以配置vcodec/acodec copy，实现不转码。譬如，视频为h264编码，但是音频是mp3/speex，需要转码音频为aac，然后切片为HLS输出。

2 禁用视频或者音频：可以禁用视频或者音频，只输出音频或视频。譬如，电台可以丢弃视频，对音频转码为aac后输出HLS。该配置只输出纯音频，编码为aac。

conf/full.conf中有很多FFMPEG转码配置的实例，也可以参考ffmpeg的命令行。

* mirror.transcode.srs.com 将视频流上半截，翻转到下半截，看起来像个镜子。
* drawtext.transcode.srs.com 加文字水印。
* crop.transcode.srs.com 剪裁视频。
* logo.transcode.srs.com 添加图片logo。
* audio.transcode.srs.com 只对音频转码。
* copy.transcode.srs.com 不转码只转封装，类似于SRS的Forward。
* all.transcode.srs.com 转码参数的详细说明。
* ffempty.transcode.srs.com 一个ffmpeg的mock，不转码只打印参数。
* app.transcode.srs.com 对指定的app的流转码。
* stream.transcode.srs.com 对指定的流转码。
* vn.transcode.srs.com 只输出音频，禁止视频输出。

### Transcode on ARM cpu

SRS可以在ARM下调用系统的ffmpeg转码，参考：[Raspberry pi 转码](https://github.com/ossrs/srs/wiki/v1_CN_ARMTranscode)

注意：使用自己的工具时，需要禁用ffmpeg，但是打开transcode选项：--with-transcode --without-ffmpeg，这样就不会编译ffmpeg，但是编译了直播转码功能。参考

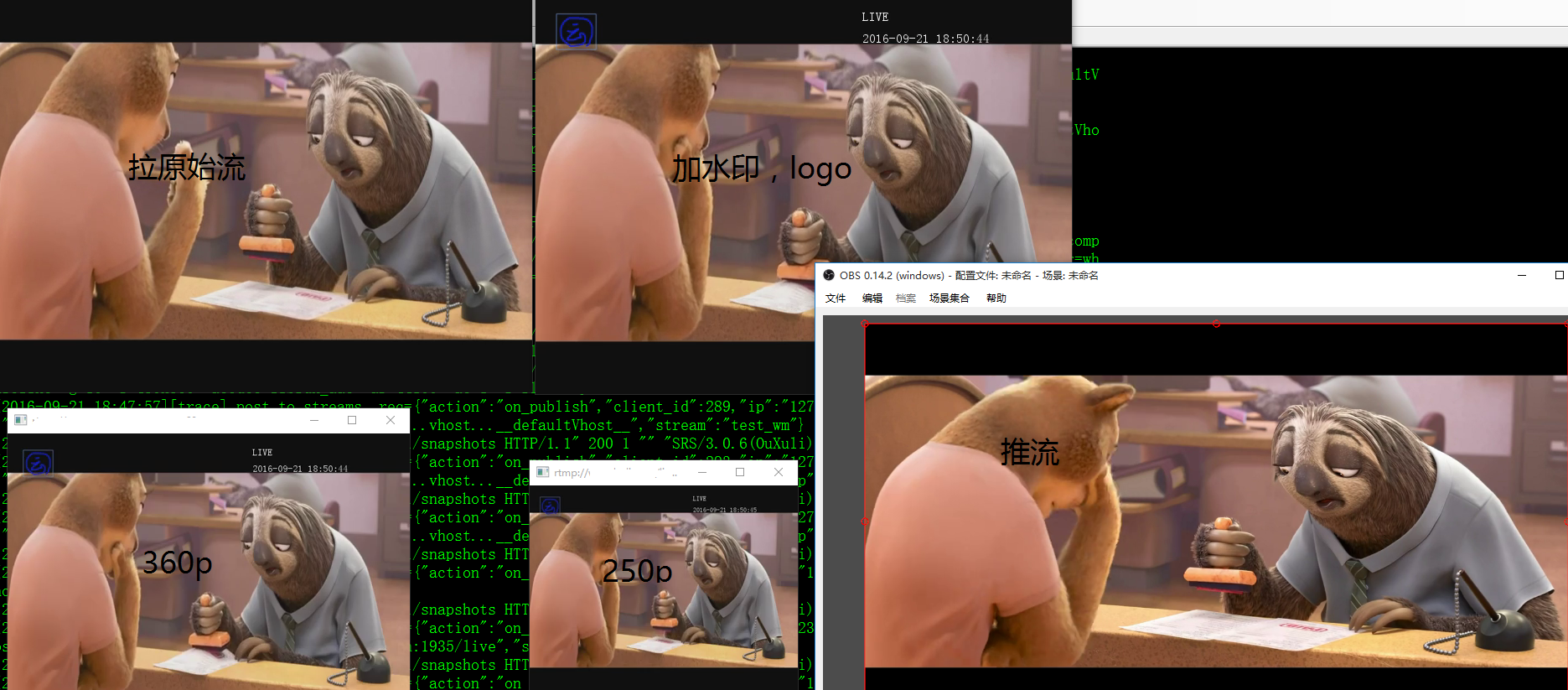
### FFMPEG Transcode the Stream by Flash encoder

flash可以当作编码器推流，参考演示中的编码器或者视频会议。flash只支持speex/nellymoser/pcma/pcmu，但flash会有一个特性，没有声音时就没有音频包。FFMPEG会依赖于这些音频包，如果没有会认为没有音频。

所以FFMPEG用来转码flash推上来的RTMP流时，可能会有一个问题：ffmpeg认为没有音频。

另外，FFMPEG取flash的流的时间会很长，也可能是在等待这些音频包。

### 补充:测试效果图



### WIKI

<https://github.com/ossrs/srs/wiki/v1_CN_SampleFFMPEG>

## 流截图

使用SRS实现截图有以下几种方式可以实现：

1. HttpCallback：使用HTTP回调，收到on\_publish事件后开启ffmpeg进程截图，收到on\_unpublish事件后停止ffmpeg进程。
2. Transcoder：转码可以配置为截图，实际上是通过ffmpeg命令将直播流按一帧数据转换为图片。

### WIKI

<https://github.com/ossrs/srs/wiki/v3_CN_Snapshot>

### 补充:测试效果图



### 转发给其他服务器(Forward)

SRS可以将送到SRS的流转发给其他RTMP服务器，实现简单集群/热备功能，也可以实现一路流热备（譬如编码器由于带宽限制，只能送一路流到RTMP服务器，要求RTMP服务器能将这路流也转发给其他RTMP备用服务器，实现主备容错集群）。

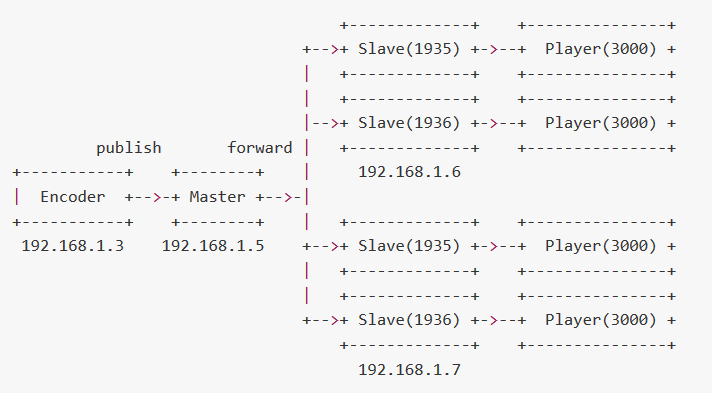
Forward就是SRS将流拷贝输出给其他的RTMP服务器，以SRS转发给SRS为例：

主SRS：Master, 编码器推流到主SRS，主SRS将流处理的同时，将流转发到备SRS

备SRS：Slave, 主SRS转发流到备SRS，就像编码器推送流到备用SRS一样。

该功能会在集群与CDN相关功能章节做详细的介绍

forward也可以用作搭建小型集群。架构图如下：



#### WIKI

<https://github.com/ossrs/srs/wiki/v3_CN_SampleForward>

### 转封装成HTTP直播流

SRS支持将RTMP流转封装为HTTP流，HTTP流格式可以flv, ts,mp3,aac等。

#### HTTP TS Live Stream

SRS支持将RTMP流转封装为HTTP ts流，即在publish发布RTMP流时，在SRS的http模块中挂载一个对应的http地址（根据配置），用户在访问这个http ts文件时，从rtmp流转封装为ts分发给用户。

#### HTTP Mp3 Live Stream

SRS支持将rtmp流中的视频丢弃，将音频流转封装为mp3格式，在SRS的http模块中挂载对应的http地址（根据配置），用户在访问这个http mp3文件时，从rtmp转封装为mp3分发给用户。

#### HTTP Aac Live Stream

SRS支持将rtmp流中的视频丢弃，将音频流转封装为aac格式，在SRS的http模块中挂载对应的http地址（根据配置），用户在访问这个http aac文件时，从rtmp转封装为aac分发给用户。

#### HTTP FLV Live Stream

SRS支持将RTMP流转封装为HTTP flv流，即在publish发布RTMP流时，在SRS的http模块中挂载一个对应的http地址（根据配置），用户在访问这个http flv文件时，从rtmp流转封装为flv分发给用户

##### What is HTTP FLV

所有的HTTP FLV流都是一个HTTP FLV地址，譬如：http://ossrs.net:8081/live/livestream.flv，但是，流的形式却至少有三种：

1. FLV文件，渐进式HTTP流。放一个文件到nginx目录，可以访问下载在播放器播放，这是HTTP FLV文件，也就是渐进式下载流。所谓渐进式下载，也就是用户观看时无法从未下载的地方开始看。
2. FLV伪流。一般说的HTTP FLV，比上面的渐进式流高级一点，譬如，一个120分钟的电影，作为渐进式流播放时，用户需要从60分钟开始看，如何支持呢？因为nginx是当做文件 下载的，无法直接跳转到第60分钟（nginx也不知道60分钟对应的字节偏移是多少呀）。后来有人就支持这种跳着播放，通过指定时间服务器从指定的位置 开始给流，这种支持flv?start=，就是http flv的伪流，本质上还是点播流。
3. FLV直播流。SRS所指的HTTP FLV流，是严格意义上的直播流，有RTMP的所有特征，譬如集群、低延迟、热备、GOP cache，而且有HTTP的优势，譬如302、穿墙、通用。由于SRS内部实现了HTTP服务器，所以SRS是在边缘将RTMP流转换成HTTP 流，SRS集群内部还是使用RTMP分发。当前唯一将RTMP和HTTP协议都解析的服务器，目前只有SRS和nginx-rtmp，可惜nginx- rtmp没有实现这个流。

用一句话概括，SRS的HTTP FLV就是增强的RTMP，真正的实时流媒体分发。

##### Confuse HTTP FLV

SRS的HTTP FLV容易和下面的几种分发方式混淆：

1. RTMPT：这个实际上是最接近SRS的HTTP FLV的概念的。但是从本质上来讲，rtmpt是基于HTTP的RTMP，所以还是RTMP而不是FLV。
2. HDL/HFL：国内一些厂家的HXX流，就是FLV流，主要和SRS的区别在于服务器集群内部SRS还是走RTMP，所以延迟可能会有很大差异。SRS的HTTP FLV和RTMP延迟一样，0.8-3秒。
3. HDS：这个差的太远了，不是一个东西。HDS和HLS像，但是HTTP FLV和他们两个都完全不像。

##### Why HTTP FLV

为何要整个HTTP FLV出来呢？当下HTTP FLV流正大行其道。主要的优势在于：

1. 互联网流媒体实时领域，还是RTMP。HTTP-FLV和RTMP的延迟一样，因此可以满足延迟的要求。
2. 穿墙：很多防火墙会墙掉RTMP，但是不会墙HTTP，因此HTTP FLV出现奇怪问题的概率很小。
3. 调度：RTMP也有个302，可惜是播放器as中支持的，HTTP FLV流就支持302方便CDN纠正DNS的错误。
4. 容错：SRS的HTTP FLV回源时可以回多个，和RTMP一样，可以支持多级热备。
5. 通用：Flash可以播RTMP，也可以播HTTP FLV。自己做的APP，也都能支持。主流播放器也都支持http flv的播放。
6. 简单：FLV是最简单的流媒体封装，HTTP是最广泛的协议，这两个到一起维护性很高，比RTMP简单多了。

Srs除了支持http-flv直播流外，现可以支持HTTP TS Live Stream，HTTP Mp3 Live Stream，HTTP Aac Live Stream

备注：若需要同时分发不同的http live stream，可以使用forward到其他vhost，不同的vhost配置不同的http live stream。

HSTRS(http stream trigger rtmp source)由HTTP流触发的RTMP回源，该功能可以用于构建HTTP-FLV集群，即HTTP-FLV流的合并回源，以及HTTP-FLV在没有流时的等待standby。

HSTRS需要开启配置项http\_remux的hstrs，默认是开启的

#### 点播FLV流

##### HTTP VOD

推荐以下的方式：

* 点播建议用http分发，http服务器一大堆。 SRS能将直播流录制为flv文件，并且提供了一些工具来支持flv点播流， 但是应该使用其他的HTTP服务器分发flv文件。
* 总之，srs不支持点播，只支持直播。这是官方回答。

点播FLV流的主要流程是：

* 服务器录制直播为FLV文件，或者上传FLV点播文件资源，到SRS的HTTP根目录：objs/nginx/html
* HTTP服务器必须要支持flv的start=offset，譬如nginx的flv模块，或者SRS的实验性HTTP服务器。
* 使用research/librtmp/objs/srs\_flv\_injecter将FLV的时间和对于的offset（文件偏移量）写入FLV的metadata。
* 播放器请求FLV文件，譬如：http://192.168.1.170:8080/sample.flv
* 用户点击进度条进行SEEK，譬如SEEK到300秒。
* 播放器根据inject的时间和offset对应关系找出准确的关键帧的offset。譬如：300秒偏移是6638860
* 根据offset发起新请求：http://192.168.1.170:8080/sample.flv?start=6638860

备注：SRS还不支持限速，会以最快的速度将文件发给客户端。 备注：SRS还提供了查看FLV文件内容的工具research/librtmp/objs/srs\_flv\_parser，可以看到metadata和每个tag信息。

#### SRS Embeded HTTP server

SRS支持http-api，因此也能解析HTTP协议（目前是部分支持），所以也实现了一个简单的HTTP服务器。

SRS的HTTP服务器已经重写，稳定可以商用。

对于一些嵌入式设备，并发也不高时，可以考虑使用SRS的HTTP服务器分发HLS，这样比较简单。

#### Wiki

<https://github.com/ossrs/srs/wiki/v2_CN_DeliveryHttpStream>

### 转封装成HLS

SRS支持HLS/RTMP两种成熟而且广泛应用的流媒体分发方式。

RTMP指Adobe的RTMP(Realtime Message Protocol)，广泛应用于低延时直播，也是编码器和服务器对接的实际标准协议，在PC（Flash）上有最佳观看体验和最佳稳定性。

HLS指Apple的HLS(Http Live Streaming)，本身就是Live（直播）的，不过Vod（点播）也能支持。HLS是Apple平台的标准流媒体协议，和RTMP在PC上一样支持得天衣无缝。

HLS主要的应用场景包括：

* 跨平台：PC主要的直播方案是RTMP，也有一些库能播放HLS，譬如jwplayer，基于osmf的hls插件也一大堆。所以实际上如果选一种协议能跨PC/Android/IOS，那就是HLS。
* IOS上苛刻的稳定性要求：IOS上最稳定的当然是HLS，稳定性不差于RTMP在PC-flash上的表现。
* 友好的CDN分发方式：目前CDN对于RTMP也是基本协议，但是HLS分发的基础是HTTP，所以CDN的接入和分发会比RTMP更加完善。能在各种CDN之间切换，RTMP也能，只是可能需要对接测试。
* 简单：HLS作为流媒体协议非常简单，apple支持得也很完善。Android对HLS的支持也会越来越完善。至于DASH/HDS，好像没有什么特别的理由，就像linux已经大行其道而且开放，其他的系统很难再广泛应用。

总之，SRS支持HLS主要是作为输出的分发协议，直播以RTMP+HLS分发，满总各种应用场景。点播以HLS为主。

#### 各种分发流协议介绍

| **分发** | **平台** | **协议** | **公司** | **说明** |
| --- | --- | --- | --- | --- |
| RTMP | Windows Flash | RTMP | Adobe | 主流的低延时分发方式， Adobe对RTMP是Flash原生支持方式， FMS（Adobe Media Server前身）， 就是Flash Media Server的简写，可见Flash播放RTMP是多么“原生”， 就像浏览器打开http网页一样“原生”， 经测试，Flash播放RTMP流可以10天以上不间断播放。 |
| HLS | Apple/ Android | HTTP | Apple/ Google | 延时一个切片以上（一般10秒以上）， Apple平台上HLS的效果比PC的RTMP还要好， 而且Apple所有设备都支持， Android最初不支持HLS，后来也支持了， 但测试发现支持得还不如Apple， 不过观看是没有问题，稳定性稍差， 所以有些公司专门做Android上的流媒体播放器。 |
| HDS | - | HTTP | Adobe | Adobe自己的HLS， 协议方面做得是复杂而且没有什么好处， 国内没有什么应用，传说国外有， SRS2.0以后已经支持。 |
| [DASH](http://en.wikipedia.org/wiki/Dynamic_Adaptive_Streaming_over_HTTP) | - | HTTP | - | Dynamic Adaptive Streaming over HTTP (DASH)， 一些公司提出的HLS， 国内还没有应用，国外据说有用了， nginx-rtmp好像已经支持了， 明显这个还不成熟。 |

#### HLS Introduction

HLS是提供一个m3u8地址，Apple的Safari浏览器直接就能打开m3u8地址，譬如：

http://demo.srs.com/live/livestream.m3u8

Android不能直接打开，需要使用html5的video标签，然后在浏览器中打开这个页面即可，譬如：

<!-- livestream.html -->

<video width="640" height="360"

autoplay controls autobuffer

src="http://demo.srs.com/live/livestream.m3u8"

type="application/vnd.apple.mpegurl">

</video>

HLS的[m3u8](https://github.com/ossrs/srs/blob/master/trunk/doc/hls-m3u8-draft-pantos-http-live-streaming-12.txt)，是一个ts的列表，也就是告诉浏览器可以播放这些ts文件，譬如：

#EXTM3U

#EXT-X-VERSION:3

#EXT-X-MEDIA-SEQUENCE:64

#EXT-X-TARGETDURATION:12

#EXTINF:11.550

livestream-64.ts

#EXTINF:5.250

livestream-65.ts

#EXTINF:7.700

livestream-66.ts

#EXTINF:6.850

livestream-67.ts

有几个关键的参数，这些参数在SRS的配置文件中都有配置项：

* EXT-X-TARGETDURATION：所有切片的最大时长。有些Apple设备这个参数不正确会无法播放。SRS会自动计算出ts文件的最大时长，然后更新m3u8时会自动更新这个值。用户不必自己配置。
* EXTINF：ts切片的实际时长，SRS提供配置项hls\_fragment，但实际上的ts时长还受gop影响，详见下面配置HLS的说明。
* ts文件的数目：SRS可配置hls\_window，指定m3u8中保存多少个切片，SRS会自动清理旧的切片。
* livestream-67.ts：SRS会自动维护ts切片的文件名，在编码器重推之后，这个编号会继续增长，保证流的连续性。直到SRS重启，这个编号才重置为0。

譬如，每个ts切片为10秒，窗口为60秒，那么m3u8中会保存6个ts切片。

#### HLS Workflow

HLS的主要流程是：

1. FFMPEG或FMLE或编码器，推送RTMP流到SRS，编码为H264/AAC（其他编码需要SRS转码）
2. SRS将RTMP切片成TS，并生成M3U8。若流非H264和AAC，则停止输出HLS（可使用SRS转码到SRS其他vhost或流，然后再切HLS）。
3. 访问m3u8，srs内置的http服务器（或者通用http服务器）提供HTTP服务。

注意：SRS只需要在Vhost上配置HLS，会自动根据流的app创建目录，但是配置的hls\_path必须自己创建

#### Wiki

<https://github.com/ossrs/srs/wiki/v1_CN_DeliveryHLS>

#### 补充:直播时移

使用hls可以实现时移回看功能。时移回看功能实际m3u8文件还是按直播的方式，只不过里面的ts片段播放列表，不是从直播实时流生成，而是从历史ts文件中来生成播放列表。直播生成的ts片段，需要按时间目录存放，比如2016071209目录存放9点内生成所有ts片段，ts文件需要按格式:“时间戳\_Ts片长.ts”来存放。时移回看时根据指定时间搜索ts文件片段，生成直播m3u8文件。SRS可以使用on\_hls事件来对直播流生成ts文件按时间目录下转存，也可以通进脚本解析srs生成的直播m3u8文件来转存ts片段。后一种方法也可以适用于nginx rtmp 的hls中。

通过直播时移，用户可以随时回到当前时间点之前的任意时间点开始回看。回看的时间可以根据自己需求来定，可以一天，七天，一个月等。

### 转封装成HDS

HDS指Adobe的Http Dynamic Stream，和Apple的[HLS](https://github.com/ossrs/srs/wiki/v2_CN_DeliveryHLS)类似。

#### Wiki

<https://github.com/ossrs/srs/wiki/v2_CN_DeliveryHDS>

### 录制成FLV

SRS可以将RTMP流录制成flv文件。

DVR的计划即决定什么时候关闭flv文件，打开新的flv文件，主要的录制计划包括：

**session：**按照session来关闭flv文件，即编码器停止推流时关闭flv，整个session录制为一个flv。

**segment：**按照时间分段录制，flv文件时长配置为dvr\_duration和dvr\_wait\_keyframe。注意：若不按关键帧切flv（即dvr\_wait\_keyframe配置为off），所以会导致后面的flv启动时会花屏。

**time\_jitter:** 时间戳抖动算法。full使用完全的时间戳矫正；zero只是保证从0开始；off不矫正时间戳。

**dvr\_path:** 录制的路径，规则如下：

按年月日以及流信息生成子目录。便于做软链，或者避免一个目录的文件太多（貌似超过几万linux会支持不了）。

1.按日期和时间以及流信息生成文件名。便于搜索。

2.提供日期和时间，以及流信息的变量，以中括号代表变量。

3.保留目前的方式，按照时间戳生成文件名，保存在一个文件夹。若没有指定文件名（只指定了目录），则默认使用[stream].[timestamp].flv作为文件名，和目前保持一致。

DVR的apply决定了是否对某个流开启dvr，默认的all是对所有开启。 这个功能是SRS实现nginx提供的control module的一个基础，而且更丰富。 也就是可以支持用户调用[http raw api](https://github.com/ossrs/srs/wiki/v3_CN_HTTPApi)控制是否以及何时DVR。

#### Wiki

<https://github.com/ossrs/srs/wiki/v3_CN_DVR>

### 分发方式比较

互联网上的两种主要的分发方式：[HLS](https://github.com/ossrs/srs/wiki/v1_CN_DeliveryHLS)和[RTMP](https://github.com/ossrs/srs/wiki/v1_CN_DeliveryRTMP)，什么时候用谁，完全决定于应用场景。还有其他的分发方式，这些分发方式不属于互联网常见和通用的方式，不予以比较：

* UDP：譬如YY的实时应用，视频会议等等，或者RTSP之类。这类应用的特点就是实时性要求特别高，以毫秒计算。TCP家族协议根本就满足不了要求，所以HTTP/TCP都不靠谱。这类应用没有通用的方案，必须自己实现分发（服务端）和播放（客户端）。
* P2P：譬如RTMFP或者各家自己的协议。这类应用的特点是节省带宽。目前PC/flash上的RTMFP比较成熟，Android上的P2P属于起步群雄纷争标准不一，IOS上P2P应该没有听说过。
* RTSP：这种不是互联网上的主要应用，在其他领域譬如安防等有广泛应用。

另外，HTTP的也分为几种：

* HTTP progressive：早期流媒体服务器分发http文件时，以普通的http文件分发，这种叫做渐进式下载，意思就是如果文件很大譬如1小时时长 1GB大小，想从中间开始播放是不行的。但这种方式已经是作古了，很多http服务器支持http文件的seek，就是从中间开始播放。
* HTTP stream：支持seek的HTTP流，譬如各家视频网站的点播分发方式。或者稍微复杂点的，譬如把一个大文件切几段之后分发。目前在pc/flash上点播国内的主流分发是这种方式。
* HLS：这种是现在适配方式最广（除了flash, 需要额外的as库支持），在PC上有vlc，Android/IOS原生播放器就支持播放HLS，HTML5里面的url可以写HLS地址。总之，在移动端是以HLS为主。
* HDS：adobe自己的HLS。
* DASH：各家提出的HLS，目前还没有广泛应用。

对比以下互联网上用的流媒体分发方式：

* HLS：apple的HLS，支持点播和直播。
* HTTP：即HTTP stream，各家自己定义的http流，应用于国内点播视频网站。
* RTMP：直播应用，对实时性有一定要求，以PC为主。

#### RTMP

RTMP本质上是流协议，主要的优势是：

* 实时性高：RTMP的实时性在3秒之内，经过多层CDN节点分发后，实时性也在3秒左右。在一些实时性有要求的应用中以RTMP为主。
* 支持加密：RTMPE和RTMPS为加密协议。虽然HLS也有加密，但在PC平台上flash对RTMPE/RTMPS支持应该比较不错。
* 稳定性高：在PC平台上flash播放的最稳定方式是RTMP，如果做CDN或者大中型集群分发，选择稳定性高的协议一定是必要的。HTTP也很稳定，但HTTP是在协议上稳定；稳定性不只是服务端的事情，在集群分发，服务器管理，主备切换，客户端的支持上，RTMP在PC分发这种方式上还是很有优势。
* 编码器接入：编码器输出到互联网（还可以输出为udp组播之类广电应用），主要是RTMP。譬如专业编码器，或者flash网页编码器，或者FMLE，或者ffmpeg，都支持RTMP输出。若需要接入多种设备，譬如提供云服务；或者希望网页直接采集摄像头；或者能在不同编码器之间切换，那么RTMP作为服务器的输入协议会是最好的选择。
* 系统容错：容错有很多种级别，RTMP的集群实现时可以指定N上层，在错误时切换不会影响到下层或者客户端，另外RTMP的流没有标识，切到其他的服务器的流也可以继续播放。HLS的流热备切换没有这么容易。若对于直播的容错要求高，譬如降低出问题的概率，选择RTMP会是很好的选择。
* 可监控：在监控系统或者运维系统的角度看，流协议应该比较合适监控。HTTP的流监控感觉没有那么完善。这个不算绝对优势，但比较有利。

RTMP的劣势是：

* 协议复杂：RTMP协议比起HTTP复杂很多，导致性能低下。测试发现两台服务器直连100Gbps网络中，HTTP能跑到60Gbps，但是 RTMP只能跑到10Gbps，CPU占用率RTMP要高很多。复杂协议导致在研发，扩展，维护软件系统时都没有HTTP那么方便，所以HTTP服务器现在大行其道，apache/nginx/tomcat，N多HTTP服务器；而RTMP协议虽然早就公开，但是真正在大规模中分发表现良好的没有，adobe自己的FMS在CDN中都经常出问题。
* Cache麻烦：流协议做缓存不方便。譬如点播，若做RTMP流协议，边缘缓存RTMP会很麻烦。如果是HTTP，缓存其实也很麻烦，但是HTTP服务器的缓存已经做了很久，所以只需要使用就好。这是为何点播都走HTTP的原因。

#### HTTP

HTTP说的是HTTP流，譬如各大视频网站的点播流。

HTTP本质上还是文件分发，主要的优势是：

* 性能很高：HTTP的性能没得说，协议简单，各种HTTP高性能服务器也完善。如果分发的量特别大，譬如点播视频网站，没有直播的实时性要求，HTTP协议是最好选择。
* 没有碎片：HTTP比HLS没有碎片，HTTP分发大文件会比小文件分发方便很多。特别是存储，小文件的性能超低，是个硬伤。
* 穿墙：互联网不可能不开放HTTP协议，否则就不叫互联网。所以任何端口封掉，也不会导致HTTP流看不了。（不过RTMP也能穿墙，用RTMPT协议）。

HTTP的劣势是：

* 实时性差：基本上没有实时性这个说法。
* 原生支持不好：就PC上flash对于HTTP流支持还可以，Android/IOS上似乎只能mp4，总之移动端对于HTTP的支持不是很完善。

#### HLS

HLS是Apple的开放标准，在Android3?以上也原生支持.

HLS的主要优势是：

* 性能高：和HTTP一样。
* 穿墙：和HTTP一样。
* 原生支持很好：IOS上支持完美。Android上支持差些。PC/flash上现在也有各种as插件支持HLS。

HLS的主要劣势是：

* 实时性差：基本上HLS的延迟在10秒以上。
* 文件碎片：若分发HLS，码流低，切片较小时，小文件分发不是很友好。特别是一些对存储比较敏感的情况，譬如源站的存储，嵌入式的SD卡。

#### 应用方式

推荐的方式是：

* 编码器输出RTMP协议。
* 流媒体系统接入使用RTMP协议。
* 流媒体系统内部直播分发使用RTMP。
* PC+直播+实时性要求高：使用flash播放RTMP。
* PC+直播+没有实时性要求：使用RTMP或者HLS均可。
* PC+点播：使用HTTP或者HLS。
* Apple IOS/OSX：都使用HLS（实时性要求高得自己解析RTMP，或者使用外部库，譬如<https://www.vitamio.org>, 开源播放器ijkplayer : <https://github.com/Bilibili/ijkplayer>）

Andorid：和IOS一样，不过可以确定的是可以自己开发支持RTMP。

## 集群与CDN相关功能

SRS包含支大规模集群如CDN业务的关键特性，譬如RTMP多级集群、VHOST虚拟服务器、无中断服务Reload、HTTP-FLV集群、Kafka对接。

### RTMP多级集群

SRS可以使用edge搭建大规模集群，使用forward搭建小规模集群。

#### Edge边缘服务器

SRS的Edge提供访问时回源机制，在CDN/VDN等流众多的应用场景中有重大意义， forward/ingest方案会造成大量带宽浪费。同时，SRS的Edge能对接所有的RTMP源站服务器， 不像FMS的Edge只能对接FMS源站（有私有协议）；另外，SRS的Edge支持SRS源站的所有逻辑 （譬如转码，转发，HLS，DVR等等），也就是说可以选择在源站切片HLS，也可以直接在边缘切片HLS。

备注：Edge一般负载高，SRS支持的并发足够跑满千兆网带宽了。

Edge的主要应用场景：

* CDN/VDN大规模集群，客户众多流众多需要按需回源。
* 小规模集群，但是流比较多，需要按需回源。
* 骨干带宽低，边缘服务器强悍，可以使用多层edge，降低上层BGP带宽。

注意：edge可以从源站拉流，也可以将流转发给源站。也就是说，播放edge上的流时，edge会回源拉流；推流到edge上时，edge会直接将流转发给源站。

注意：若只需要中转流给源站，不必用forward，直接使用edge模式即可。可以直接支持推流 和拉流的中转，简单快捷。Forward应用于目标服务器是多个，譬如将一路流主动送给多路服务器；edge虽然配置了多台服务器，但是只用了一台，有故障时才切换。

注意：优先使用edge，除非知道必须用forward，才使用forward。

##### 概念

所谓边缘edge服务器，就是边缘直播缓存服务器，配置时指定为remote模式和origin（指定一个或多个源站IP），这个边缘edge服务器就是源站的缓存了。

当用户推流到边缘服务器时，边缘直接将流转发给源站。譬如源站在北京BGP机房，湖南有个电信ADSL用户要推流发布自己的直播流，要是直接推流到北京BGP可能效果不是很好，可以在湖南电信机房部署一个边缘，用户推流到湖南边缘，边缘转发给北京源站BGP。

当用户播放边缘服务器的流时，边缘服务器看有没有缓存，若缓存了就直接将流发给客户端。 若没有缓存，则发起一路回源链接，从源站取数据源源不断放到自己的缓存队列。也就是说， 多个客户端连接到边缘时，只有一路回源。这种结构在CDN是最典型的部署结构。譬如北京源站，在全国32个省每个省都部署了10台服务器，一共就有320台边缘，假设每个省1台边缘服务器都有 2000用户观看，那么就有64万用户，每秒钟集群发送640Gbps数据；而回源链接只有320个， 实现了大规模分发。

边缘edge服务器，实际上是解决大并发问题产生的分布式集群结构。SRS的边缘可以指定多个源站， 在源站出现故障时会自动切换到下一个源站，不影响用户观看，具有最佳的容错性，用户完全不会觉察。

##### HLS边缘

Edge指的是RTMP边缘，也就是说，配置为Edge后，流推送到源站（Origin）时，Edge不会切片生成HLS。

HLS切片配置在源站，只有源站会在推流上来就产生HLS切片。边缘只有在访问时才会回源（这个时候 也会生成HLS，但单独访问边缘的HLS是不行的）。

也就是说，HLS的边缘需要使用WEB服务器缓存，譬如nginx反向代理，squid，或者traffic server等。

补充：<https://github.com/ossrs/srs/issues/466> srs作者考虑hls回源功能也就是HLS+

##### 下行边缘结构设计

下行边缘指的是下行加速边缘，即客户端播放边缘服务器的流，边缘服务器从上层或源站取流。

SRS下行边缘是非常重要的功能，需要考虑以下因素：

* 以后支持多进程时结构变动最小。
* 和目前所有功能的对接良好。
* 支持平滑切换，源站和边缘两种角色。

权衡后，SRS下行边缘的结构设计如下：

* 客户端连接到SRS
* 开始播放SRS的流
* 若流存在则直接播放。
* 若流不存在，则从源站开始取流。
* 其他流的功能，譬如转码/转发/采集等等。

核心原则是：

* 边缘服务器在没有流时，向源站拉取流。
* 当流建立起来后，边缘完全变成源站服务器，对流的处理逻辑保持一致。
* 支持回多个源站，错误时切换。这样可以支持上层服务器热备。

备注：RTMP多进程（计划中）的核心原则是用多进程作为完全镜像代理，连接到本地的服务器 （源站或边缘），完全不考虑其他业务因素，透明代理。这样可以简单，而且利用多CPU能力。 HTTP多进程是不考虑支持的，用NGINX是最好选择，SRS的HTTP服务器只是用在嵌入式设备中,没有性能要求的场合。

##### 上行边缘结构设计

上行边缘指的是上行推流加速，客户端推流到边缘服务器，边缘将流转发给源站服务器。

考虑到下行和上行可能同时发生在一台边缘服务器，所以上行边缘只能用最简单的代理方式， 完全将流代理到上层或源站服务器。也就是说，只有在下行边缘时，边缘服务器才会启用其他的功能，譬如HLS转发等等。

上行边缘主要流程是：

* 客户端连接到SRS
* 开始推流到SRS。
* 开始转发到源站服务器。

##### 边缘的难点

RTMP边缘对于SRS来讲问题不大，主要是混合了reload和HLS功能的边缘服务器，会是一个难点。

譬如，用户在访问边缘上的HLS流时，是使用nginx反向代理回源，还是使用RTMP回源后在边缘切片？ 对于前者，需要部署srs作为RTMP边缘，nginx作为HLS边缘，管理两个服务器自然是比一个要费劲。 若使用后者，即RTMP回源后边缘切片，能节省骨干带宽，只有一路回源，难点在于访问HLS时要发起 RTMP回源连接。

正因为业务逻辑会是边缘服务器的难点，所以SRS对于上行边缘，采取直接代理方式，并没有采取边缘缓存方式。所谓边缘缓存方式，即推流到边缘时边缘也会当作源站直接缓存（作为源站）， 然后转发给源站。边缘缓存方式看起来先进，这个边缘节点不必回源，实际上加大了集群的逻辑难度， 不如直接作为代理方式简单。

#### Forward小规模集群

srs定位为直播服务器，其中一项重要的功能是forward，即将服务器的流转发到其他服务器。

forward本身是用做热备，即用户推一路流上来，可以被SRS转发（或者转码后转发）到多个slave源站，CDN边缘可以回多个slave源，实现故障热备的功能，构建强容错系统。

注意：edge可以从源站拉流，也可以将流转发给源站。也就是说，播放edge上的流时，edge会回源拉流；推流到edge上时，edge会直接将流转发给源站。

若只需要中转流给源站，不必用forward，直接使用edge模式即可。可以直接支持推流和拉流的中转，简单快捷。Forward应用于目标服务器是多个，譬如将一路流主动送给多路服务器；edge虽然配置了多台服务器，但是只用了一台，有故障时才切换。优先使用edge，除非知道必须用forward，才使用forward。

##### Keywords

为了和edge方式区分，forward定义一次词汇如下：

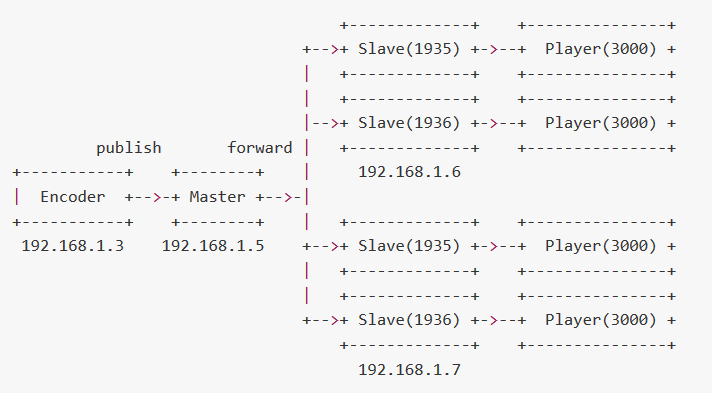
* master：主服务器，编码器推流到这个服务器，或者用ingest流到服务器。总之，master就是主服务器，负责转发流给其他服务器。
* slave：从服务器，主服务器转发流到这个服务器。

如果结合edge集群方式，一般而言master和slave都是origin（源站服务器），edge边缘服务器可以从master或者slave回源取流。

实际上master和slave也可以是edge，但是不推荐，这种组合方式太多了，测试没有办法覆盖到。因此，强烈建议简化服务器的结构，只有origin（源站服务器）才配置转发，edge（边缘服务器）只做边缘。

##### For Small Cluster

forward也可以用作搭建小型集群。架构图如下：



##### Forward VS Edge

Forward架构和CDN架构的最大区别在于，CDN属于大规模集群，边缘节点会有成千上万台，源站2台（做热备），还需要有中间层。CDN的客户很多，流也会有很多。所以假若源站将每个流都转发给边缘，会造成巨大的浪费（有很多流只有少数节点需要）。

可见，forward只适用于所有边缘节点都需要所有的流。CDN是某些边缘节点需要某些流。

forward的瓶颈在于流的数目，假设每个SRS只侦听一个端口：

系统中流的数目 = 编码器的流数目 × 节点数目 × 端口数目

考虑5个节点，每个节点起4个端口，即有20个SRS边缘。编码器出5路流，则有20 \* 5 = 100路流。

同样的架构，对于CDN的边缘节点来讲，系统的流数为用户访问边缘节点的流，假设没有用户访问，系统中就没有流量。某个区域的用户访问某个节点上的流，系统中只有一路流，而不是forward广播式的多路流。

另外，forward需要播放器随机访问多个端口，实现负载均衡，或者播放器访问api服务器，api服务器实现负载均衡，对于CDN来讲也不合适（需要客户改播放器）。

总之，forward适用于小型规模的集群，不适用于CDN大规模集群应用。

##### 其他使用场景

forward还可以结合hls和transcoder功能使用，即在源站将流转码，然后forward到Slave节点，Slave节点支持rtmp同时切HLS。

因为用户推上来的流，或者编码器（譬如FMLE）可能不是h264+aac，需要先转码为h264+aac（可以只转码音频）后才能切片为hls。

需要结合vhost，先将流transcode送到另外一个vhost，这个vhost将流转发到Slave。这样可以只转发转码的流。

参考vhost，hls和transcoder相关wiki。

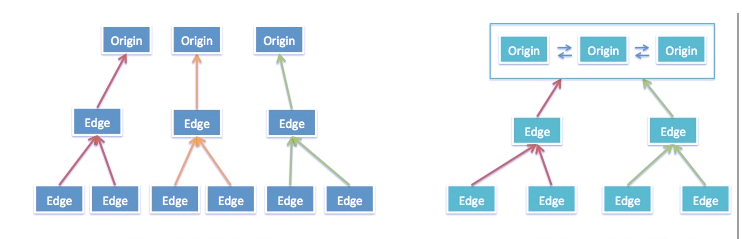
#### WIKI

<https://github.com/ossrs/srs/wiki/v3_CN_SampleRTMPCluster>

#### 补充:源站集群问题

目前srs多个边缘回多个源站时，某一时刻只能选择一个源站。  
因此，如果流推到N(N>=3)个源站中的两个，譬如做热备时，一定有一个源站是没有流的；这个时候边缘如果回源到这个源站，就会导致没有流；而边缘还得等待，因为边缘是无法知道源站没有这个流。  
 从热备和负载均衡的角度看，应该支持多个源站，这些源站之间需要通信和同步状态，这样边缘在连接到没有流的源站时，源站可以告知边缘正确的源站是谁，也就是源站集群。

源站集群实现可通过RTMP重定向和实时动态反馈调度算法让多个彼此独立的源站服务器建立通信，将它们组织成一个松耦合的虚拟“超级源站”，与之的交互，就像与一个超性能、高可用的单台源站交互一样。



传统源站结构图 源站集群结构图

### VHOST虚拟服务器

#### RTMP的URL/Vhost规则

RTMP的url其实很简单，vhost其实也没有什么新的概念，但是对于没有使用过的同学来讲，还是很容易混淆。几乎每个新人都必问的问题：RTMP那个URL推流时应该填什么，什么是vhost，什么是app？

#### 应用场景

Vhost的主要应用场景包括：

* 一个分发网络支持多个客户：譬如CDN，一个分发网络中，有N个客户公用一套流媒体系统，如何区分用户，计费，监控等等？通过app么？大家可能都叫做live之类。最好是通过各自的域名。
* 不同的应用配置：譬如FMLE推上来的流是h264+mp3，可以将音频转码后放到其他的vhost分发hls，这样接入h264+mp3的vhost就不用切hls。

总之，vhost作为应用配置的单元，能隔离客户，应用不同的配置。

#### 标准RTMP URL

标准RTMP URL指的是最大兼容的RTMP URL，基本上所有的服务器和播放器都能识别的URL，和HTTP URL其实很相似，例如：

| **HTTP** | **Schema** | **Host** | **Port** | **App** | **Stream** |
| --- | --- | --- | --- | --- | --- |
| [http://192.168.1.10:80/players/srs\_player.html](http://192.168.1.10/players/srs_player.html) | http | 192.168.1.10 | 80 | players | srs\_player.html |
| rtmp://192.168.1.10:1935/live/livestream | rtmp | 192.168.1.10 | 1935 | live | livestream |

其中：

* Schema：协议头，HTTP为HTTP或HTTPS，RTMP为RTMP/RTMPS/RTMPE/RTMPT等众多协议，还有新出的RTMFP。
* Host：主机，表示要连接的主机，可以为主机DNS名称或者IP地址。商用时，一般不会用IP地址，而是DNS名称，这样可以用CDN分发内容（CDN一般使用DNS调度，即不同网络和地理位置的用户，通过DNS解析到的IP不一样，实现用户的就近访问）。
* Port：端口，HTTP默认为80，RTMP默认为1935。当端口没有指定时，使用默认端口。
* Path：路径，HTTP访问的文件路径。
* App：RTMP的Application（应用）名称，可以类比为文件夹。以文件夹来分类不同的流，没有特殊约定，可以任意划分。
* Stream：RTMP的Stream（流）名称，可以类比为文件。

#### NoVhost

其实，vhost大多数用户都用不到，而且不推荐用，有点复杂。一般的用户用app就可以了。因为vhost/app/stream，只是一个分类方法而已；vhost需要在配置文件中说明，app/stream都不需要配置。

什么时候用vhost？如果你是提供服务，譬如你有100个客户，都要用一套平台，走同样的流媒体服务器分发。那可以每个客户一个vhost，这样他们的app和stream可以相同都可以。

一般的用法，举个例子，有个视频网站，自己搭建服务器，所以只有他自己一个客户，就不要用vhost了。假设视频网站提供聊天服务，聊天有不同的话题类型，譬如：军事栏目，读书栏目，历史栏目三个分类，每个分类下面有很多聊天室。只要这么配置就好：

listen 1935;

vhost \_\_defaultVhost\_\_ {

}

生成网页时，譬如军事栏目的网页，都用app名称为military，某个聊天室叫做火箭，这个页面的流可以用：rtmp://yourdomain.com/military/rock，编码器也推这个流，所有观看这个军事栏目/火箭聊天室的页面的人，都播放这个流。

军事栏目另外的网页，都用同样的app名称military，但是流不一样，譬如某个聊天室叫做雷达，这个页面的流可以用：rtmp://yourdomain.com/military/radar，推流和观看一样。

如此类推，军事栏目页面生成时，不用更改srs的任何配置。也就是说，新增聊天室，不用改服务器配置；新增分类，譬如加个公开课的聊天室，也不用改服务器配置。足够简单！

另外，读书栏目可以用app名称为reader，栏目下的某个聊天室叫红楼梦，这个页面的流可以用：rtmp://yourdomain.com/reader/red\_mansion，所有在这个聊天室的人都是播放这个流。

#### Vhost的应用

RTMP的Vhost和HTTP的Vhost概念是一样的：虚拟主机。详见下表（假设域名demo.srs.com被解析到IP为192.168.1.10的服务器）：

| **HTTP** | **Host** | **Port** | **Vhost** |
| --- | --- | --- | --- |
| <http://demo.srs.com:80/players/srs_player.html> | 192.168.1.10 | 80 | demo.srs.com |
| rtmp://demo.srs.com:1935/live/livestream | 192.168.1.10 | 1935 | demo.srs.com |

Vhost主要的作用是：

* 支持多用户：当一台服务器需要服务多个客户，譬如CDN有cctv（央视）和wasu（华数传媒）两个客户时，如何隔离他们两个的资源？相当于不同的用户共用一台计算机，他们可以在自己的文件系统建立同样的文件目录结构，但是彼此不会冲突。
* 域名调度：CDN分发内容时，需要让用户访问离自己最近的边缘节点，边缘节点再从源站或上层节点获取数据，达到加速访问的效果。一般的做法就是Host是DNS域名，这样可以根据用户的信息解析到不同的节点。
* 支持多配置：有时候需要使用不同的配置，考虑一个支持多终端（PC/Apple/Android）的应用，PC上RTMP分发，Apple和 Android是HLS分发，如何让PC延迟最低，同时HLS也能支持，而且终端播放时尽量地址一致（降低终端开发难度）？可以使用两个Vhost，PC 和HLS；PC配置为最低延迟的RTMP，并且将流转发给HLS的Vhost，可以对音频转码（可能不是H264/AAC）后切片为HLS。PC和HLS 这两个Vhost的配置肯定是不一样的，播放时，流名称是一样，只需要使用不同的Host就可以。

#### Vhost支持多用户

假设cctv和wasu都运行在一台边缘节点(192.168.1.10)上，用户访问这两个媒体的流时，Vhost的作用见下表：

| **RTMP** | **Host** | **Port** | **Vhost** | **App** | **Stream** |
| --- | --- | --- | --- | --- | --- |
| rtmp://show.cctv.cn/live/livestream | 192.168.1.10 | 1935 | show.cctv.cn | live | livestream |
| rtmp://show.wasu.cn/live/livestream | 192.168.1.10 | 1935 | show.wasu.cn | live | livestream |

在边缘节点（192.168.1.10）上的SRS，需要配置Vhost，例如：

listen 1935;

vhost show.cctv.cn {

}

vhost show.wasu.cn {

}

#### 补充:Vhost域名调度

每个客户都有自己的域名。使用这个域名推流或拉流,如rtmp://pull.test.com/live/stream，然后通过下面这些流程，向边缘服务器推流或拉流。

* 1. 首先pull.test.com通过本地域名服务器解析到对应运营商的授权服务器，
  2. 授权服务器通过CNAME记录，将请求转向全局负载均衡GLBS，GLBS使用智能调度算法，将请求解析到离用户最近本地负载均衡，获取虚拟服务器地址，
  3. 用户使用虚拟服务器地址，通过本地负载均衡调试算法，选择一台负载最轻的srs边缘服务器。用户最终向使用这台服务器推流或拉流。

大概流程如下图所示



#### Vhost支持多配置

以上面举的例子，若cctv需要延迟最低（意味着启动时只有声音，画面是黑屏），而wasu需要快速启动（打开就能看到视频，服务器cache了最后一个gop，延迟会较大）。

只需要对这两个Vhost进行不同的配置，例如：

listen 1935;

vhost show.cctv.cn {

chunk\_size 128;

gop\_cache off;

}

vhost show.wasu.cn {

chunk\_size 4906;

gop\_cache on;

}

总之，这两个Vhost的配置完全没有关系，不会相互影响。

#### \_\_defaultVhost\_\_

FMS的\_\_defaultVhost\_\_是默认的vhost，当用户请求的vhost没有匹配成功时，若配置了defaultVhost，则使用它来提供服务。若匹配失败，也没有defaultVhost，则返回错误。

譬如，服务器192.168.1.10上的SRS配置如下：

listen 1935;

vhost demo.srs.com {

enabled on;

}

那么，当用户访问以下vhost时：

* rtmp://demo.srs.com/live/livestream：成功，匹配vhost为demo.srs.com
* rtmp://192.168.1.10/live/livestream：失败，没有找到vhost，也没有defaultVhost。

defaultVhost和其他vhost的规则一样，只是用来匹配那些没有匹配成功的vhost的请求的。

#### 访问指定的Vhost

如何访问某台服务器上的Vhost？有两个方法：

* 配置hosts：因为Vhost实际上就是DNS解析，所以可以配置客户端的hosts，将域名（Vhost）解析到指定的服务器，就可以访问这台服务器上的指定的vhost。
* 使用app的参数：需要服务器支持。在app后面带参数指定要访问的Vhost。SRS支持?vhost=VHOST和...vhost...VHOST这两种方式，后面的方式是避免一些播放器不识别？和=等特殊字符。

普通用户不用这么麻烦，直接访问RTMP地址就好了，有时候运维需要看某台机器上的Vhost的流是否有问题，就需要这种特殊的访问方式。考虑下面的例子：

RTMP URL: rtmp://demo.srs.com/live/livestream

边缘节点数目：50台

边缘节点IP：192.168.1.100 至 192.168.1.150

边缘节点SRS配置：

listen 1935;

vhost demo.srs.com {

mode remote;

origin: xxxxxxx;

}

各种访问方式见下表：

| **用户** | **RTMP URL** | **hosts设置** | **目标** |
| --- | --- | --- | --- |
| 普通用户 | rtmp://demo.srs.com/live/livestream | 无 | 由DNS 解析到指定边缘 |
| 运维 | rtmp://demo.srs.com/live/livestream | 192.168.1.100 demo.srs.com | 查看192.168.1.100上的流 |
| 运维 | rtmp://192.168.1.100/live? vhost=demo.srs.com/livestream | 无 | 查看192.168.1.100上的流 |
| 运维 | rtmp://192.168.1.100/live ...vhost...demo.srs.com/livestream | 无 | 查看192.168.1.100上的流 |

访问其他服务器的流也类似。

#### FMLE的奇怪URL方式

FMLE推流时，URL那个地方，有三个可以输入的框，参考[Adobe FMLE](http://help.adobe.com/en_US/FlashMediaLiveEncoder/3.0/Using/WS5b3ccc516d4fbf351e63e3d11c104ba878-7ff7.html)：

* FMS URL: 需要输入rtmp://host:port/app，例如：rtmp://demo.srs.com/live
* Backup URL: 备份的服务器，格式同FMS URL。若指定了备份服务器，FMLE会同时推送给这两个服务器。
* Stream: 流名称，例如：livestream

实际上是将RTMP URL分成了两部分，stream前面那部分和stream。为何要这么搞？我猜想有以下原因：

* 支持多级app和Stream：我们目前举的例子都是一级app和一级stream，实际上RTMP支持多级app和stream，就像子文件夹，实际上很少用得到。所以SRS的URL都是一个地址，默认最后一个/后面就是stream，前面是app。
* 支持流名称带参数：Adobe的鬼HLS/HDS非常之麻烦，那个地址是个恶心的完全不一致。参考[FMS livepkgr](http://help.adobe.com/en_US/flashmediaserver/devguide/WSd391de4d9c7bd609-52e437a812a3725dfa0-8000.html" \l "WSd391de4d9c7bd609-52e437a812a3725dfa0-7ff5)，例如发布一个rtmp，并切片成HLS：

FMLE:

FMS URL: rtmp://demo.srs.com/livepkgr

Stream: livestream?adbe-live-event=liveevent

Client:

RTMP: rtmp://demo.srs.com/livepkgr/livestream

HLS: http://demo.srs.com/hls-live/livepkgr/\_definst\_/liveevent/livestream.m3u8

HDS: http://demo.srs.com/hds-live/livepkgr/\_definst\_/liveevent/livestream.f4m

没有比这个更恶心的东西了。比较SRS的简洁方案：

FMLE:

FMS URL: rtmp://demo.srs.com/livepkgr

Stream: livestream

Client:

RTMP: rtmp://demo.srs.com/livepkgr/livestream

HLS: http://demo.srs.com/livepkgr/livestream.m3u8

HDS: not support yet.

既然谈到了RTMP URL中的参数，下一章就说说这个。

#### RTMP URL参数

RTMP URL一般是不带参数，类似于http的query，有时候为了特殊的要求，会在RTMP URL中带参数，譬如：

* Vhost：前面讲过，在app后面加参数，可以访问指定服务器的指定Vhost。这个SRS的特殊约定，方便排错。
* FMLE的Stream后面的参数，指定event之类的。SRS不需要这么麻烦，HLS是内置支持，无需这种复杂的配置。Callback也是http的，FMS为了支持服务器端脚本，需要很复杂的配置和复杂的参数，实在是很麻烦的设计。
* token认证：SRS还未实现。在连接服务器时，在app后面指定token（方式和vhost一样），例如rtmp://server/live?vhost=xxx&token=xxx/livestream，服务器可以取出token，进行验证，若验证失败则断开连接，这种是比Refer更高级的防盗链。

app和stream后面带参数，这两者有何区别，为何SRS把参数放在app后面？客户端播放流的as3代码大约是：

// how to play url: rtmp://demo.srs.com/live/livestream

conn = new NetConnection();

conn.connect("rtmp://demo.srs.com/live");

stream = new NetStream(conn);

stream.play("livestream");

从RTMP协议的角度来看：

* NetConnection.connect(vhost+app)：这一步会完成握手，connect到vhost，切换到app。类似于登录到vhost后，cd到app这个目录。也就是vhost的验证，都可以在这一步做，也就是指定vhost也是在一步了，所以app后面跟的参数都是和vhost/app相关的。
* NetStream.play(stream)：这一步是播放指定的直播流。所以和stream相关的事件，都可以传递参数，譬如Adobe的event。SRS是没有这些事件的，流启动时，若配置了HLS会自动开始切片。

#### SRS的URL规则

SRS只做简化的事情，绝对不把简单的事情搞复杂。

SRS的RTMP URL使用标准的RTMP URL，一般不需要对app和stream加参数，或者更改他们的意义。除了两个地方：

* vhost支持参数访问：为了方便运维访问某台服务器的vhost，不需要设置hosts。不影响普通用户。
* 支持token验证：为了支持token验证，在app后面带参数，这个是token验证必须的方式。

另外，SRS建议用户使用一级app和一级stream，不使用多级app和多级stream。譬如：

// 不推荐使用的多级app或stream

rtmp://demo.srs.com/show/live/livestream

rtmp://demo.srs.com/show/live/livestream/2013

srs播放器(srs\_player)和srs编码器(srs\_publisher)不支持多级app和stream，他们认为最后一个斜杠（/）后面的就是stream，前面的是app。即：

// srs\_player和srs\_publisher的解析方式：

// play or publish the following rtmp URL:

rtmp://demo.srs.com/show/live/livestream/2013

schema: rtmp

host/vhost: demo.srs.com

app: show/live/livestream

stream: 2013

做此简化的好处是，srs播放器和编码器，只需要指定一个url，而且两者的url是一样的。

SRS常见的三种RTMP URL，详细见下表：

| **URL** | **说明** |
| --- | --- |
| rtmp://demo.srs.com/live/livestream | 普通用户的标准访问方式，观看直播流 |
| rtmp://192.168.1.10/live?vhost=demo.srs.com/livestream | 运维对特定服务器排错 |
| rtmp://demo.srs.com/live?key=ER892ID839KD9D0A1D87D/livestream | token验证用户，或者带宽测试的key验证 |

#### WIKI

<https://github.com/ossrs/srs/wiki/v1_CN_RtmpUrlVhost>

### 无中断服务Reload

SRS配置完全支持Reload，即在不中断服务时应用配置的修改。

#### NotSupportedFeatures

不支持reload的功能包括：

* deamon，是否后台启动。
* mode，vhost的模式。

daemon选项当然是不支持reload的。

mode选项，即决定vhost是源站还是边缘，不支持reload。若修改mode之后reload会导致server异常退出，由看门狗重启。原因在于：

* 源站和边缘角色切换过于复杂。
* 一般源站会建立设备组，全部做源站，不会突然变成边缘
* 上层和源站重启后，对最终用户没有影响，只是表现会切换上层的卡顿（客户端缓冲区设为3秒以上时，卡顿都不会出现）。

一个修改vhost的mode属性的workaround：

* 删除vhost并reload。
* 确认vhost已经删除了。
* 添加vhost，使用新的mode，并reload。

#### 应用场景

Reload主要应用场景：

* 配置快速生效：不用重启服务，修改配置后，只需要killall -1 srs即可生效配置。
* 不中断服务：商用服务器往往时时刻刻都在服务用户，如何将一个转码流的码率调低？如何禁用某些频道的HLS？如何添加和删除频道？而且还中断现有用户的服务？使用Reload。

#### 使用方法

Reload的方法为：killall -1 srs

或者指定发送的SRS进程：kill -1 7635

使用启动脚本：/etc/init.d/srs reload

#### WIKI

<https://github.com/ossrs/srs/wiki/v1_CN_Reload>

### HTTP-FLV集群

http-flv集群实际使用edge边缘集群模式，在边缘服务器上启动http-flv功能，当用户通过输入地址如:http://demo.srs.com/live/test.flv使用http-flv访问时，会在边缘转换成rtmp, 然后rtmp回源。结构如下: 用户< -------http-flv------> 边缘< -------rtmp-------- > 源。http-ts http-aac, http-aac集群原理也一样。按原理来说Hls集群也可以采用这种模式来实现，不过目前srs开源没有实现hls集群，只能通过反回代理方式,如squid, nginx，SRS商业版实现了hls集群，也就是他们说的hls+。请参考：<https://github.com/ossrs/srs/issues/466>

http-flv详细内容请参考“[转封装成HTTP直播流](https://github.com/ossrs/srs/wiki/v2_CN_SampleHttpFlv)”章节

SRS的HTTP FLV边缘只能使用单进程，如何做到多进程呢？可以使用HTTP反向代理，SRS提供了go-sharp，支持根据SRS边缘的负载均衡以及心跳检测。参考：[go-sharp](https://github.com/simple-rtmp-server/go-sharp)

#### WIKI

<https://github.com/ossrs/srs/wiki/v3_CN_SampleHttpFlvCluster>

### Kafka对接

SRS可以将信息以JSON形式发送到KAFKA集群，Spark等大数据系统从Kafka获取数据并处理。

目前SRS会将如下事件写入Kafka集群：

1. accept: 当收到客户端连接时。
2. close: 当关闭客户端连接时。

#### WIKI

<https://github.com/ossrs/srs/wiki/v3_CN_Kafka>

## 应用接口

SRS还提供丰富的应用接口，包括HTTP回调、安全策略Security、HTTP API接口、RTMP测速。

### HTTP回调

SRS不支持服务器脚本，服务器端定制有一个重要的替代功能，就是HTTP回调。譬如当客户端连接到SRS时，回调指定的http地址，这样可以实现验证功能。

#### 服务器脚本

SRS不支持服务器端脚本，所谓服务器端脚本，指的是服务器可以加载外部脚本文件，解释并执行。

支持服务器脚本的服务器有FMS，语言是actionscript1.0；nginx支持的是lua。

SRS不支持服务器脚本的原因有：

* 不Simple：违反了SRS(Simple RTMP Server)的第一个S，支持扩展脚本，出错的几率也扩展了。
* 实际用处很小：我在国内知名的CDN公司工作时，所在部门就是用FMS，当然FMS不提供源码，所以只能支持服务器脚本来定制。结果商用起来很费劲，基本上每天出问题，而且还没法查原因。所以实际的用处很小。
* SRS支持HTTP调用：调用外部http，实际上也是一种扩展方式，SRS支持这种较好的方式。譬如当用户连接上SRS时，会调用HTTP接口，可以做验证。
* SRS开源：为何要定制脚本？重要的一个原因就是闭源，SRS开源，可以修改源码。
* SRS代码定制简单：SRS整个服务器实现代码才2万行，nginx-rtmp是3万行+nginx的14万行，定制SRS要简单很多。而且SRS是“同步”处理的，逻辑很少。

综上所述，SRS暂时不考虑支持扩展脚本，这个东西没啥用。

#### HTTP callback events

SRS的回调事件包括：

| **事件** | **数据** | **说明** |
| --- | --- | --- |
| on\_connect | { "action": "on\_connect", "client\_id": 1985, "ip": "192.168.1.10",  "vhost": "video.test.com",  "app": "live", "tcUrl": "rtmp://x/x?key=xxx", "pageUrl": "http://x/x.html" } | 当客户端连接到指定的vhost和app时 |
| on\_close | { "action": "on\_close", "client\_id": 1985, "ip": "192.168.1.10",  "vhost": "video.test.com",  "app": "live",  "send\_bytes": 10240,  "recv\_bytes": 10240  } | 当客户端关闭连接，或者SRS主动关闭连接时 |
| on\_publish | { "action": "on\_publish", "client\_id": 1985, "ip": "192.168.1.10",  "vhost": "video.test.com",  "app": "live", "stream": "livestream" } | 当客户端发布流时，譬如flash/FMLE方式推流到服务器 |
| on\_unpublish | { "action": "on\_unpublish", "client\_id": 1985, "ip": "192.168.1.10",  "vhost": "video.test.com",  "app": "live", "stream": "livestream" } | 当客户端停止发布流时 |
| on\_play | { "action": "on\_play", "client\_id": 1985, "ip": "192.168.1.10",  "vhost": "video.test.com",  "app": "live", "stream": "livestream", "pageUrl": "<http://www.test.com/live.html>" } | 当客户端开始播放流时 |
| on\_stop | { "action": "on\_stop", "client\_id": 1985, "ip": "192.168.1.10",  "vhost": "video.test.com",  "app": "live", "stream": "livestream" } | 当客户端停止播放时。备注：停止播放可能不会关闭连接，还能再继续播放。 |
| on\_dvr | { "action": "on\_dvr", "client\_id": 1985, "ip": "192.168.1.10",  "vhost": "video.test.com",  "app": "live", "stream": "livestream", "cwd": "/opt", "file": "./l.xxx.flv" } | 当DVR录制关闭一个flv文件时 |

其中，

* 事件：发生该事件时，即回调指定的HTTP地址。
* HTTP地址：可以支持多个，以空格分隔，SRS会依次回调这些接口。
* 数据：SRS将数据POST到HTTP接口。
* 返回值：SRS要求HTTP服务器返回HTTP200并且response内容为整数错误码（0表示成功），其他错误码会断开客户端连接。

#### 基于http回调的认证

##### DRM

DRM重要的功能就是防盗链，只有允许的用户，才能访问服务器的流。有多种DRM的方式：

* refer防盗链：检查用户从哪个网站过来的。譬如不是从公司的页面过来的人都不让看。
* token防盗链：用户在播放时，必须先申请token，SRS会回调http检查这个token合法性。
* FMS token tranverse：边缘RTMP服务器收到每个连接，都去上行节点验证，即token穿越认证。
* Access服务器：专门的access服务器负责DRM。譬如adobe的access服务器。
* 推流认证：adobe的RTMP推流时，支持几种认证方式，这个也可以归于防盗链概念。

##### Refer Authentication

SRS支持refer防盗链，adobe的flash在播放RTMP流时，会把页面的http url放在请求中， as客户端代码不可以更改。当然如果用自己的客户端，不用flash播放流，就可以随意伪造了； 尽管如此，refer防盗链还是能防住相当一部分盗链。

配置Refer防盗链，在vhost中开启refer即可，可以指定publish和play的refer：

# the vhost for antisuck.

vhost refer.anti\_suck.com {

# refer hotlink-denial.

refer {

# whether enable the refer hotlink-denial.

# default: off.

enabled on;

# the common refer for play and publish.

# if the page url of client not in the refer, access denied.

# if not specified this field, allow all.

# default: not specified.

all github.com github.io;

# refer for publish clients specified.

# the common refer is not overrided by this.

# if not specified this field, allow all.

# default: not specified.

publish github.com github.io;

# refer for play clients specified.

# the common refer is not overrided by this.

# if not specified this field, allow all.

# default: not specified.

play github.com github.io;

}

}

备注：SRS1/2的Refer配置方法和SRS3不一致，SRS3兼容SRS1/2的配置方法。

##### Token Authentication

token类似于refer，不过是放在RTMP url中，或者在connect的请求参数中：

* token在RTMP url，譬如：rtmp://vhost/app?token=xxxx/stream，这样服务器在on\_connect回调接口中， 就会把url带过去验证。
* token在connect的参数中：as函数NetConnection.connect(url, token)，服务器也可以拿到这个token。注意：SRS目前不支持。

token比refer更强悍，可以指定超时时间，可以变更token之类。可惜就是需要服务器端做定制，做验证。 SRS提供http回调来做验证，已经有人用这种方式做了，比较简单靠谱。

举个常用的token认证的例子：

1. 用户在web页面登录，服务器可以生成一个token，譬如token=md5(time+id+私钥+有效期)=88195f8943e5c944066725df2b1706f8
2. 服务器返回给用户一个地址，带token，譬如：rtmp://192.168.1.10/live?time=1402307089& expire=3600&token=88195f8943e5c944066725df2b1706f8/livestream
3. 配置srs的http回调，on\_connect http://127.0.0.1:8085/api/v1/clients;
4. 用户播放时，srs会回调那个地址，解析请求的内容，里面的tcUrl就有那些认证信息。 按同样的算法验证，如果md5变了就返回错误，srs就会拒绝连接。如果返回0就会接受连接。

##### TokenTraverse

Token防盗链的穿越，指的是在origin-edge集群中，客户播放edge边缘服务器的流时， 边缘将认证的token发送给源站进行验证，即token穿越。

FMS的edge和FMS的origin使用私有协议，使用一个连接回源取数据，一个连接回源传输控制命令， 譬如token穿越就是在这个连接做的。

token认证建议使用http方式，也就是说客户端连接到边缘时，边缘使用http回调方式验证token。 像fms那种token穿越，是需要走RTMP协议，其他开源服务器一般都不支持这种方式（中国特色）。

SRS可以支持类似fms的token穿越，不过实现方式稍微有区别，不是采用fms edge的私有协议， 而是每次新开一个连接回源验证，验证通过后边缘才提供服务。也就是边缘先做一个完全的代理。

SRS这种方式的特点是：

* 在token认证上，能和fms源站对接，fms源站感觉不到什么区别。
* 每次边缘都会新开连接去验证，开销会大一些；而且只限于connect事件验证，马上验证过后就会收到disconnect事件。
* 会导致源站的短连接过多（连接验证token，断开），不过可以加一层fms edge解决，这样比所有都是fms edge要好。

对于源站短连接过多的问题，可以加一层fms边缘缓解，假设1000个客户端连接到边缘：

* srs => 客户fms 这种方案，会有1000个连接去回源验证，然后断开。
* srs => cdn-fms => 客户fms 这种方案，会有1000个连接去cdn的fms去验证，只有1个连接去客户那边验证。

SRS的token穿越(traverse)的配置，参考edge.token.traverse.conf：

listen 1935;

vhost \_\_defaultVhost\_\_ {

cluster {

mode remote;

origin 127.0.0.1:19350;

token\_traverse on;

}

}

##### Access服务器

SRS暂时不支持。

##### 推流认证

SRS暂时不支持，是RTMP特殊的握手协议。

#### 默认的HTTP服务器

SRS自带了一个默认的处理HTTP Callback的服务器，启动时需要指定端口，譬如8085端口。

启动方法：python research/api-server/server.py 8085

#### Wiki

<https://github.com/ossrs/srs/wiki/v3_CN_HTTPCallback>

### 安全策略Security

#### Security

SRS提供了禁用或允许客户端的简单安全策略。

#### Config

Vhost中安全策略的配置：

vhost your\_vhost {

# security for host to allow or deny clients.

# @see https://github.com/ossrs/srs/issues/211

security {

# whether enable the security for vhost.

# default: off

enabled on;

# the security list, each item format as:

# allow|deny publish|play all|<ip>

# for example:

# allow publish all;

# deny publish all;

# allow publish 127.0.0.1;

# deny publish 127.0.0.1;

# allow play all;

# deny play all;

# allow play 127.0.0.1;

# deny play 127.0.0.1;

# SRS apply the following simple strategies one by one:

# 1. allow all if security disabled.

# 2. default to deny all when security enabled.

# 3. allow if matches allow strategy.

# 4. deny if matches deny strategy.

allow play all;

allow publish all;

}

}

SRS应用安全策略的方式是:

* 若securty没有开启，则允许所有。
* 若security开启了，默认禁止所有。
* 允许客户端，若找到了匹配的允许策略。
* 禁用客户端，若找到了匹配的禁用策略。

参考配置文件conf/security.deny.publish.conf.

#### Kickoff Client

可以踢掉连接的用户，SRS提供了HTTP RESTful接口：

DELETE /api/v1/clients/{id}

可以先查询到需要踢掉的Client的ID：

GET /api/v1/clients

若需要踢掉推流的Client，可以从streams接口中查询推流client的id：

GET /api/v1/streams

or GET /api/v1/streams/6745

流信息中的stream.publish.cid就是推流的客户端id：

1. GET http://192.168.1.170:1985/api/v1/streams/6745

2. Response stream.publish.cid:

stream: {

publish: {

active: true,

cid: 107

}

}

3. DELETE http://192.168.1.170:1985/api/v1/clients/107

备注：HTTP请求可以使用[HTTP REST Tool](http://ossrs.net/srs.release/http-rest/index.html)

备注：HTTP请求还可以使用Linux的工具curl，常见的请求如下：

curl -v -X GET http://192.168.1.170:1985/api/v1/clients/426 && echo ""

curl -v -X DELETE http://192.168.1.170:1985/api/v1/clients/426 && echo ""

#### Reload

当Reload改变security配置后，只影响新连接的客户端，已经连接的客户端不受影响。

#### Wiki

<https://github.com/ossrs/srs/wiki/v2_CN_Security>

### HTTP API接口

SRS提供HTTP接口，供外部程序管理服务器，并支持跨域（js可以直接控制和获取服务器的各种信息）。

#### 设计原则

SRS的HTTP接口遵循最简单原则，主要包括：

* 只提供json数据格式接口，要求请求和响应的数据全都是json。
* 不提供html数据，譬如运行SRS后，浏览器打开HTTP接口或HTTP服务地址，看到的是json，不是html。
* 发生错误时，支持HTTP错误码，或者json中的code错误码。

#### Config

配置文件需要开启http-api：

listen 1935;

# system statistics section.

# the main cycle will retrieve the system stat,

# for example, the cpu/mem/network/disk-io data,

# the http api, for instance, /api/v1/summaries will show these data.

# @remark the heartbeat depends on the network,

# for example, the eth0 maybe the device which index is 0.

stats {

# the index of device ip.

# we may retrieve more than one network device.

# default: 0

network 0;

# the device name to stat the disk iops.

# ignore the device of /proc/diskstats if not configed.

disk sda sdb xvda xvdb;

}

# api of srs.

# the http api config, export for external program to manage srs.

# user can access http api of srs in browser directly, for instance, to access by:

# curl http://192.168.1.170:1985/api/v1/reload

# which will reload srs, like cmd killall -1 srs, but the js can also invoke the http api,

# where the cli can only be used in shell/terminate.

http\_api {

# whether http api is enabled.

# default: off

enabled on;

# the http api listen entry is <[ip:]port>

# for example, 192.168.1.100:1985

# where the ip is optional, default to 0.0.0.0, that is 1985 equals to 0.0.0.0:1985

# default: 1985

listen 1985;

# whether enable crossdomain request.

# default: on

crossdomain on;

# the HTTP RAW API is more powerful api to change srs state and reload.

raw\_api {

# whether enable the HTTP RAW API.

# default: off

enabled off;

# whether enable rpc reload.

# default: off

allow\_reload off;

# whether enable rpc query.

# default: off

allow\_query off;

# whether enable rpc update.

# default: off

allow\_update off;

}

}

vhost \_\_defaultVhost\_\_ {

}

其中，http\_api开启了HTTP API，stats配置了SRS后台统计的信息，包括：

* network: 这个配置了heartbeat使用的网卡ip，即SRS主动汇报的网卡信息。参考[Heartbeat](https://github.com/ossrs/srs/wiki/v1_CN_Heartbeat)
* disk: 这个配置了需要统计的磁盘的IOPS，可以通过cat /proc/diskstats命令获得名称，譬如阿里云的磁盘名称叫xvda.

#### 访问api

直接在浏览器中就可以访问，或者用curl发起http请求。

SRS提供了api的面包屑，可以从根目录开始导航，不需要任何记忆。一般的字段包括：

* code表示错误码，按照linux惯例，0表示成功。
* urls表示是面包屑导航，该api下面的子api（链接）。
* data表示最后一级提供服务的api，返回的数据。

另外，提供服务的api按照HTTP RESTful规则是复数，譬如versions/authors，表示资源。HTTP的各种方法表示操作，譬如GET查询，PUT更新，DELETE删除。参考：[Redmine HTTP Rest api](http://www.redmine.org/projects/redmine/wiki/Rest_api)

根目录：

# curl http://192.168.1.170:1985/

"urls": {

"api": "the api root"

}

返回的urls表示子链接可以访问。接着访问：

# curl http://192.168.1.170:1985/api/

"urls": {

"v1": "the api version 1.0"

}

继续：

# curl http://192.168.1.170:1985/api/v1/

"urls": {

"versions": "the version of SRS",

"authors": "the primary authors and contributors"

}

继续：

# curl http://192.168.1.170:1985/api/v1/versions

"major": 0,

"minor": 9,

"revision": 43,

"version": "0.9.43"

或者：

# curl http://192.168.1.170:1985/api/v1/authors

"primary\_authors": "winlin,wenjie.zhao",

"contributors\_link": "https://github.com/ossrs/srs/blob/master/AUTHORS.txt",

"contributors": "winlin<winlin@vip.126.com> wenjie.zhao<740936897@qq.com> xiangcheng.liu<liuxc0116@foxmail.com> naijia.liu<youngcow@youngcow.net> alcoholyi<alcoholyi@qq.com> "

SRS的API属于“自解释型，HTTP RESTful API”

#### Error Code

SRS可能返回HTTP错误，即Status不等于200；或者在HTTP Status为200时，响应的json的code不为0.

譬如，返回HTTP错误：

winlin:~ winlin$ curl -v http://127.0.0.1:1985 && echo ""

< HTTP/1.1 404 Not Found

< Connection: Keep-Alive

< Content-Length: 9

< Content-Type: text/plain; charset=utf-8

< Server: SRS/2.0.184

<

Not Found

譬如，HTTP200时内容中code不等于0：

winlin:~ winlin$ curl -v http://127.0.0.1:1985/api/v1/tests/errors && echo ""

< HTTP/1.1 200 OK

< Connection: Keep-Alive

< Content-Length: 12

< Content-Type: application/json

< Server: SRS/2.0.184

<

{"code":100}

用户应该处理这两种错误。

#### Crossdomain

SRS HTTP API支持跨域，js可以直接调用srs的http api。

SRS支持两种跨域方式：

* OPTIONS: jquery可以直接跨域请求API，浏览器会发送一个OPTIONS跨域请求，SRS允许跨域后，浏览器再次发起API请求。
* JSONP: jquery/angularjs可以发起JSONP跨域请求，服务器会将响应作为js文件，内容是调用一个函数，函数名由QueryString中的callback指定。
* JSONP-DELETE: JSONP只能GET，因此DELETE方法是由QueryString的method指定的。

JSONP实例，例如：

GET http://localhost:1985/api/v1/vhosts/?callback=JSON\_CALLBACK

JSON\_CALLBACK({"code":0,"server":13449})

GET http://localhost:1985/api/v1/vhosts/100?callback=JSON\_CALLBACK&method=DELETE

JSON\_CALLBACK({"code":0})

#### Server ID

SRS返回的api中都会带有server的信息，即Server的ID，用来标识服务器。客户端在获取信息时，必须检查ServerID是否改变，改变时就是服务器重启，之前所有的数据都应该作废了。

#### API及描述

SRS提供了API的导航，即所有支持的API及描述。

地址是：http://192.168.1.170:1985/api/v1，主要包含的子api有：

| **API** | **Example** | **Description** |
| --- | --- | --- |
| server | 4481 | 服务器标识 |
| versions | /api/v1/versions | 获取服务器版本信息 |
| summaries | /api/v1/summaries | 获取服务器的摘要信息 |
| rusages | /api/v1/rusages | 获取服务器资源使用信息 |
| self\_proc\_stats | /api/v1/self\_proc\_stats | 获取服务器进程信息 |
| system\_proc\_stats | /api/v1/system\_proc\_stats | 获取服务器所有进程情况 |
| meminfos | /api/v1/meminfos | 获取服务器内存使用情况 |
| authors | /api/v1/authors | 获取作者、版权和License信息 |
| features | /api/v1/features | 获取系统支持的功能列表 |
| requests | /api/v1/requests | 获取请求的信息，即当前发起的请求的详细信息 |
| vhosts | /api/v1/vhosts | 获取服务器上的vhosts信息 |
| streams | /api/v1/streams | 获取服务器的streams信息 |
| clients | /api/v1/clients | 获取服务器的clients信息，默认获取前10个 |
| configs | /api/v1/configs | CUID配置，RAW API |

#### Summaries

SRS提供系统的摘要信息接口，譬如当前的内存、CPU、网络、负载使用率。

地址为：http://192.168.1.170:1985/api/v1/summaries

#### Vhosts

SRS提供获取所有vhost的接口，vhost中的server为srs的id，用来标识是否服务器重启了。

地址为：http://192.168.1.170:1985/api/v1/vhosts

还可以继续处理某个vhost的信息，譬如http://192.168.1.170:1985/api/v1/vhosts/3756

#### Streams

SRS提供获取所有stream的接口，stream中的server为srs的id，用来标识是否服务器重启了。vhost为stream所属的vhost的id。

地址为：http://192.168.1.170:1985/api/v1/streams

还可以继续处理某个stream的信息，譬如http://192.168.1.170:1985/api/v1/streams/3756

#### Clients

SRS提供查询客户端信息的接口，和Vhosts或Streams不一样的是，Clients查询时需要指定start和count(默认start为0，count为10，即查询头10个clients)。

地址为：http://192.168.1.170:1985/api/v1/clients

还可以继续处理某个client的信息，譬如http://192.168.1.170:1985/api/v1/clients/3756

#### Kickoff Client

可以踢掉连接的用户，SRS提供了HTTP RESTful接口：

DELETE /api/v1/clients/{id}

可以先查询到需要踢掉的Client的ID：

GET /api/v1/clients

若需要踢掉推流的Client，可以从streams接口中查询推流client的id：

GET /api/v1/streams

or GET /api/v1/streams/6745

流信息中的stream.publish.cid就是推流的客户端id：

1. GET http://192.168.1.170:1985/api/v1/streams/6745

2. Response stream.publish.cid:

stream: {

publish: {

active: true,

cid: 107

}

}

3. DELETE http://192.168.1.170:1985/api/v1/clients/107

备注：HTTP请求可以使用[HTTP REST Tool](http://ossrs.net/srs.release/http-rest/index.html)

备注：HTTP请求还可以使用Linux的工具curl，常见的请求如下：

curl -v -X GET http://192.168.1.170:1985/api/v1/clients/426 && echo ""

curl -v -X DELETE http://192.168.1.170:1985/api/v1/clients/426 && echo ""

#### Persistence Config

可以发送信号SIGUSR1给SRS，用来将当前系统的配置写入配置文件。主要用于：

1. 去掉注释，没有用的配置，格式化配置。
2. 支持HTTP API写入配置，通过HTTP API改变配置后写入然后Reload，参考：[HTTP RAW API](https://github.com/ossrs/srs/issues/319).

命令实例：

killall -s SIGUSR1 srs

killall -30 srs

写出来的配置文件可能是：

listen 1935;

max\_connections 1000;

daemon off;

srs\_log\_tank console;

pithy\_print\_ms 1000;

http\_api {

enabled on;

listen 1985;

}

http\_server {

enabled on;

listen 8080;

}

stream\_caster {

enabled off;

caster flv;

output rtmp://127.0.0.1/[app]/[stream];

listen 8936;

}

vhost \_\_defaultVhost\_\_ {

ingest livestream {

enabled on;

input {

type file;

url doc/source.200kbps.768x320.flv;

}

ffmpeg ./objs/ffmpeg/bin/ffmpeg;

engine {

enabled off;

output rtmp://127.0.0.1:[port]/live?vhost=[vhost]/livestream;

}

}

http\_remux {

enabled off;

mount [vhost]/[app]/[stream].flv;

hstrs on;

}

}

#### HTTP RAW API

SRS支持RAW API，一般的服务器只能提供读(Read)形式的API，譬如获取系统状态之类，但是SRS提供写(Write)形式的API，譬如Reload和修改系统配置等所有改写系统的行为。

**注意:** 必须在http\_api配置中，开启http\_api.raw\_api.enabled才能允许HTTP RAW API，否则会返回错误代码是1061。

http\_api {

enabled on;

listen 1985;

raw\_api {

enabled on;

allow\_reload on;

allow\_query on;

allow\_update on;

}

}

SRS支持的HTTP RAW API包括：

* [Raw](https://github.com/ossrs/srs/wiki/v3_CN_HTTPApi#raw): 查看HTTP RAW API的配置。
* [Reload](https://github.com/ossrs/srs/wiki/v3_CN_HTTPApi#reload): 支持reload配置。
* [Query](https://github.com/ossrs/srs/wiki/v3_CN_HTTPApi#raw-query): 查询全局和Vhost配置。
* [Update](https://github.com/ossrs/srs/wiki/v3_CN_HTTPApi#raw-update): 更新全局和Vhost配置。
* [Vhost](https://github.com/ossrs/srs/wiki/v3_CN_HTTPApi#raw-vhost): Vhost操作是Update的子集。
* [DVR](https://github.com/ossrs/srs/wiki/v3_CN_HTTPApi#raw-dvr): DVR操作是Update的子集。

##### Raw

| **Key** | **DESC** |
| --- | --- |
| feature | 查询服务器端HTTP RAW API的配置 |
| url | /api/v1/raw?rpc=raw |
| curl | curl http://127.0.0.1:1985/api/v1/raw?rpc=raw |
| config | 不需要 |
| params | 无参数 |

##### RAW Reload

| **Key** | **DESC** |
| --- | --- |
| feature | 可以重新加载配置，和killall -1 srs的效果是一样的 |
| url | /api/v1/raw?rpc=reload |
| curl | curl http://127.0.0.1:1985/api/v1/raw?rpc=reload |
| config | allow\_reload on; |
| params | 无参数 |

##### RAW Query

| **Key** | **DESC** |
| --- | --- |
| feature | 查询服务器全局配置 |
| url | /api/v1/raw?rpc=query&scope=global |
| curl | curl 'http://127.0.0.1:1985/api/v1/raw?rpc=query&scope=global' |
| config | allow\_query on; |
| params | scope=global，查询服务器的全局配置 |
|  |  |

| **Key** | **DESC** |
| --- | --- |
| feature | 查询服务器最小全局配置 |
| url | /api/v1/raw?rpc=query&scope=minimal |
| curl | curl 'http://127.0.0.1:1985/api/v1/raw?rpc=query&scope=minimal' |
| config | allow\_query on; |
| params | scope=minimal，查询服务器的最小全局配置 |

| **Key** | **DESC** |
| --- | --- |
| feature | 查询服务器指定的Vhost配置 |
| url | /api/v1/raw?rpc=query&scope=vhost&vhost=\_\_defaultVhost\_\_ |
| curl | curl 'http://127.0.0.1:1985/api/v1/raw?rpc=query&scope=vhost&vhost=\_\_defaultVhost\_\_' |
| config | allow\_query on; |
| params | scope=vhost&vhost=xxx，查询服务器的指定的Vhost的配置 |

##### RAW Update

| **Key** | **DESC** |
| --- | --- |
| feature | 更新服务器侦听端口 |
| url | /api/v1/raw?rpc=update&scope=listen&value=1935,1936 |
| curl | curl 'http://127.0.0.1:1985/api/v1/raw?rpc=update&scope=listen&value=1935,1936' |
| config | allow\_update on; |
| params | scope=listen&value=1935,1936，指定侦听的端口列表 |
| require | 参数必须是整数端口列表，多个端口时以逗号分割，譬如：1935,1936,1937 |

| **Key** | **DESC** |
| --- | --- |
| feature | 更新服务器PID文件 |
| url | /api/v1/raw?rpc=update&scope=pid&value=./objs/srs.pid |
| curl | curl 'http://127.0.0.1:1985/api/v1/raw?rpc=update&scope=pid&value=./objs/srs.pid' |
| config | allow\_update on; |
| params | scope=pid&value=./objs/srs.pid，指定新的PID文件 |
| require | 文件路径必须以./，/tmp/或/var/开头，并且扩展名必须是.pid，譬如：/var/srs.pid |

| **Key** | **DESC** |
| --- | --- |
| feature | 设置RTMP全局chunk\_size |
| url | /api/v1/raw?rpc=update&scope=chunk\_size&value=60000 |
| curl | curl 'http://127.0.0.1:1985/api/v1/raw?rpc=update&scope=chunk\_size&value=60000' |
| config | allow\_update on; |
| params | scope=chunk\_size&value=60000，指定新的全局chunk\_size |
| require | chunk\_size必须是数字，并且在[128, 65535]中，譬如：60000 |

| **Key** | **DESC** |
| --- | --- |
| feature | 设置ffmpeg的全局日志路径 |
| url | /api/v1/raw?rpc=update&scope=ff\_log\_dir&value=./objs |
| curl | curl 'http://127.0.0.1:1985/api/v1/raw?rpc=update&scope=ff\_log\_dir&value=./objs' |
| config | allow\_update on; |
| params | scope=ff\_log\_dir&value=./objs，指定新的全局ff\_log\_dir |
| require | ff\_log\_dir必须以./, /tmp/或/var/开头。譬如：./objs |

| **Key** | **DESC** |
| --- | --- |
| feature | 设置SRS的日志容器 |
| url | /api/v1/raw?rpc=update&scope=srs\_log\_tank&value=file |
| curl | curl 'http://127.0.0.1:1985/api/v1/raw?rpc=update&scope=srs\_log\_tank&value=file' |
| config | allow\_update on; |
| params | scope=srs\_log\_tank&value=file，设置新的日志容器 |
| require | srs\_log\_tank必须是file或console。譬如：file |

| **Key** | **DESC** |
| --- | --- |
| feature | 设置SRS的日志级别 |
| url | /api/v1/raw?rpc=update&scope=srs\_log\_level&value=trace |
| curl | curl 'http://127.0.0.1:1985/api/v1/raw?rpc=update&scope=srs\_log\_level&value=trace' |
| config | allow\_update on; |
| params | scope=srs\_log\_level&value=trace，设置新的日志级别 |
| require | srs\_log\_level必须是verbose,info,trace,warn,error。譬如：trace |

| **Key** | **DESC** |
| --- | --- |
| feature | 设置SRS的日志文件路径 |
| url | /api/v1/raw?rpc=update&scope=srs\_log\_file&value=./objs/srs.log |
| curl | curl 'http://127.0.0.1:1985/api/v1/raw?rpc=update&scope=srs\_log\_file&value=./objs/srs.log' |
| config | allow\_update on; |
| params | scope=srs\_log\_file&value=./objs/srs.log，设置新的日志文件路径 |
| require | srs\_log\_file必须是.log类型，并且在./, /var/, /tmp/开头。譬如：./objs/srs.log |

| **Key** | **DESC** |
| --- | --- |
| feature | 设置SRS能服务的最大连接数，包含RTMP和HTTP连接 |
| url | /api/v1/raw?rpc=update&scope=max\_connections&value=1000 |
| curl | curl 'http://127.0.0.1:1985/api/v1/raw?rpc=update&scope=max\_connections&value=1000' |
| config | allow\_update on; |
| params | scope=max\_connections&value=1000，设置新的最大连接数 |
| require | max\_connections必须是整型，并且在[10, 65535]范围内。譬如：1000 |
|  |  |

| **Key** | **DESC** |
| --- | --- |
| feature | 是否开启utc时间，影响日志和包含时间的路径，譬如DVR和HLS |
| url | /api/v1/raw?rpc=update&scope=utc\_time&value=false |
| curl | curl 'http://127.0.0.1:1985/api/v1/raw?rpc=update&scope=utc\_time&value=false' |
| config | allow\_update on; |
| params | scope=utc\_time&value=false，设置是否开启utc时间 |
| require | utc\_time是bool，必须是true或false。譬如：false |

| **Key** | **DESC** |
| --- | --- |
| feature | 设置全局的pithy打印时间间隔 |
| url | /api/v1/raw?rpc=update&scope=pithy\_print\_ms&value=10000 |
| curl | curl 'http://127.0.0.1:1985/api/v1/raw?rpc=update&scope=pithy\_print\_ms&value=10000' |
| config | allow\_update on; |
| params | scope=pithy\_print\_ms&value=10000，设置新的pithy打印时间间隔 |
| require | pithy\_print\_ms单位是毫秒，在[100,300000]范围内。譬如：10000 |

##### RAW Vhost

Vhost操作是Update的一个子集。

| **Key** | **DESC** |
| --- | --- |
| feature | 新增vhost，指定vhost名称 |
| url | /api/v1/raw?rpc=update&scope=vhost&value=ossrs.net&param=create |
| curl | curl 'http://127.0.0.1:1985/api/v1/raw?rpc=update&scope=vhost&value=ossrs.net&param=create' |
| config | allow\_update on; |
| params | scope=vhost&value=ossrs.net&param=create，创建新的vhost |
| require | 新创建的vhost必须是不存在的。注意禁用的vhost也算是存在的vhost。 |

| **Key** | **DESC** |
| --- | --- |
| feature | 修改禁用的vhost名称 |
| url | /api/v1/raw?rpc=update&scope=vhost&value=ossrs.net&param=update&data=new.ossrs.net |
| curl | curl 'http://127.0.0.1:1985/api/v1/raw?rpc=update&scope=vhost&value=ossrs.net&param=update&data=new.ossrs.net' |
| config | allow\_update on; |
| params | scope=vhost&value=ossrs.net&param=update&data=new.ossrs.net，修改vhost名称为新名称 |
| require | 需要修改的vhost必须存在，并且是禁用状态。 |

| **Key** | **DESC** |
| --- | --- |
| feature | 禁用vhost |
| url | /api/v1/raw?rpc=update&scope=vhost&value=ossrs.net&param=disable |
| curl | curl 'http://127.0.0.1:1985/api/v1/raw?rpc=update&scope=vhost&value=ossrs.net&param=disable' |
| config | allow\_update on; |
| params | scope=vhost&value=ossrs.net&param=disable，禁用vhost |
| require | 要禁用的vhost必须存在并且是启用状态。 |

| **Key** | **DESC** |
| --- | --- |
| feature | 启用vhost |
| url | /api/v1/raw?rpc=update&scope=vhost&value=ossrs.net&param=enable |
| curl | curl 'http://127.0.0.1:1985/api/v1/raw?rpc=update&scope=vhost&value=ossrs.net&param=enable' |
| config | allow\_update on; |
| params | scope=vhost&value=ossrs.net&param=enable，启用vhost |
| require | 要启用的vhost必须存在并且是禁用状态。 |

##### RAW DVR

DVR操作是Update的一个子集。

| **Key** | **DESC** |
| --- | --- |
| feature | 开启Vhost的某个流的DVR |
| url | /api/v1/raw?rpc=update&scope=dvr&value=ossrs.net&param=enable&data=live/livestream |
| curl | curl 'http://127.0.0.1:1985/api/v1/raw?rpc=update&scope=dvr&value=ossrs.net&param=enable&data=live/livestream' |
| config | allow\_update on; |
| params | scope=dvr&value=ossrs.net&param=enable&data=live/livestream，对Vhost的Stream开启DVR |
| require | 必须Vhost的DVR是开启状态。 |

| **Key** | **DESC** |
| --- | --- |
| feature | 关闭Vhost的某个流的DVR |
| url | /api/v1/raw?rpc=update&scope=dvr&value=ossrs.net&param=disable&data=live/livestream |
| curl | curl 'http://127.0.0.1:1985/api/v1/raw?rpc=update&scope=dvr&value=ossrs.net&param=disable&data=live/livestream' |
| config | allow\_update on; |
| params | scope=dvr&value=ossrs.net&param=disable&data=live/livestream，对Vhost的Stream关闭DVR |
| require | 必须Vhost的DVR是开启状态。 |

#### WIKI

<https://github.com/ossrs/srs/wiki/v3_CN_HTTPApi>

### RTMP测速

视频很卡，播放不了，缓冲区突然很大，推流上不来，都有可能是带宽过低，SRS支持测试客户端到服务器的带宽。

#### SRS配置

SRS配置一般是单独加一个vhost支持测速。SRS的配置conf/bandwidth.conf。譬如：

listen 1935;

vhost \_\_defaultVhost\_\_ {

}

vhost bandcheck.srs.com {

enabled on;

chunk\_size 65000;

bandcheck {

enabled on;

key "35c9b402c12a7246868752e2878f7e0e";

interval 30;

limit\_kbps 4000;

}

}

其中：

* key：服务器的key，若客户端给出的key和配置的不一致，断开连接。
* interval：测速的间隔，单位为秒，可为小数。若连续发起测速，时间间隔小于interval，服务器拒绝连接。
* limit\_kbps：测速的最大带宽，即可以测出来的最大带宽，防止服务器收到攻击。

**假设服务器的IP是：192.168.1.170**

#### Flash测速工具

启动后用带宽测试客户端就可以查看：<http://winlinvip.github.io/srs.release/trunk/research/players/srs_bwt.html?server=192.168.1.170>

备注：请将所有实例的IP地址192.168.1.170都换成部署的服务器IP地址。

检测完毕后会提示带宽，譬如：

检测结束: 192.168.1.170 上行: 1965 kbps 下行: 3631 kbps 测试时间: 6.0 秒

server:SRS 0.9.156 (github.com/winlinvip/simple-rtmp-server),

authors:winlin,wenjie.zhao, srs\_id:123, srs\_pid:32057, ip:192.168.1.170

#### 测速库

我提供了AS和JS的库，可以直接调用来和服务器测速。

AS的库，直接拷贝文件SrsBandwidth.as到工程，调用即可（参考注释说明）：

* AS库对象：[SrsBandwidth.as](https://github.com/ossrs/srs/blob/master/trunk/research/players/srs_bwt/src/SrsBandwidth.as)
* AS调用对象（主对象）：[srs\_bwt.as](https://github.com/ossrs/srs/blob/master/trunk/research/players/srs_bwt/src/srs_bwt.as)，如何调用SrsBandwidth.as的实例。

JS的库，需要拷贝srs\_bwt.swf和srs.bandwidth.js，调用方法参考js说明：

* JS库对象：[srs.bandwidth.js](https://github.com/ossrs/srs/blob/master/trunk/research/players/srs_bwt/src/srs.bandwidth.js)
* JS调用对象（页面）：[srs\_bwt.html](https://github.com/ossrs/srs/blob/master/trunk/research/players/srs_bwt.html)，如何调用srs.bandwidth.js的实例。

备注：JS需要调用swf导出的js函数，由Flash发送RTMP包测速，因此js库依赖于as。可以导入Flex工程自己编译，或者使用已经编译好的[srs\_bwt.swf](https://github.com/ossrs/srs/blob/master/trunk/research/players/srs_bwt/release/srs_bwt.swf)

#### Linux工具测速

另外，SRS还提供了带宽检测命令行工具：

[winlin@dev6 srs]$ cd objs/research/librtmp/

[winlin@dev6 librtmp]$ ./srs\_bandwidth\_check

RTMP bandwidth check/test with server.

Usage: ./srs\_bandwidth\_check <rtmp\_url>

rtmp\_url RTMP bandwidth url to check. format: rtmp://server:port/app?key=xxx,vhost=xxx

For example:

./srs\_bandwidth\_check rtmp://127.0.0.1:1935/app?key=35c9b402c12a7246868752e2878f7e0e,vhost=bandcheck.srs.com

./srs\_bandwidth\_check rtmp://127.0.0.1:1935/app?key=35c9b402c12a7246868752e2878f7e0e,vhost=bandcheck.srs.com>/dev/null

@remark, output text to stdout, while json to stderr.

直接执行将打印文本和json信息：

[winlin@dev6 librtmp]$ ./srs\_bandwidth\_check rtmp://127.0.0.1:1935/app?key=35c9b402c12a7246868752e2878f7e0e,vhost=bandcheck.srs.com

RTMP bandwidth check/test with server.

srs(simple-rtmp-server) client librtmp library.

version: 0.9.158

bandwidth check/test url: rtmp://127.0.0.1:1935/app?key=35c9b402c12a7246868752e2878f7e0e,vhost=bandcheck.srs.com

simple handshake success

connect vhost/app success

bandwidth check/test success

SRS 0.9.158 (github.com/winlinvip/simple-rtmp-server), winlin,wenjie.zhao

127.0.0.1, 0.9.158, srs\_pid=15264, srs\_id=107

duration: 6395ms(3165+3148)

play: 3578kbps

publish: 4035kbps

terminate with ret=0

{"code":0,

"srs\_server":"SRS 0.9.158 (github.com/winlinvip/simple-rtmp-server)",

"srs\_primary\_authors":"winlin,wenjie.zhao",

"srs\_server\_ip":"127.0.0.1",

"srs\_version":"0.9.158",

"srs\_pid":15264,

"srs\_id":107,

"duration":6395,

"play\_duration":3165,

"play\_kbps":3148,

"publish\_kbps":3578}

可以只打印json信息，将stdout定向到/dev/null：

[winlin@dev6 librtmp]$ ./srs\_bandwidth\_check rtmp://127.0.0.1:1935/app?key=35c9b402c12a7246868752e2878f7e0e,vhost=bandcheck.srs.com>/dev/null

{"code":0,

"srs\_server":"SRS 0.9.158 (github.com/winlinvip/simple-rtmp-server)",

"srs\_primary\_authors":"winlin,wenjie.zhao",

"srs\_server\_ip":"127.0.0.1",

"srs\_version":"0.9.158",

"srs\_pid":15264,

"srs\_id":109,

"duration":6354,

"play\_duration":3092,

"play\_kbps":3177,

"publish\_kbps":3662}

#### WIKI

<https://github.com/ossrs/srs/wiki/v1_CN_BandwidthTestTool>

## 其他功能

SRS提供了一些特殊的配置，主要用来和各种系统对接的设置。

### Send Minimal Interval

vhost \_\_defaultVhost\_\_ {

# for play client, both RTMP and other stream clients,

# for instance, the HTTP FLV stream clients.

play {

# the minimal packets send interval in ms,

# used to control the ndiff of stream by srs\_rtmp\_dump,

# for example, some device can only accept some stream which

# delivery packets in constant interval(not cbr).

# @remark 0 to disable the minimal interval.

# @remark >0 to make the srs to send message one by one.

# @remark user can get the right packets interval in ms by srs\_rtmp\_dump.

# default: 0

send\_min\_interval 10.0;

}

}

### Reduce Sequence Header

vhost \_\_defaultVhost\_\_ {

# for play client, both RTMP and other stream clients,

# for instance, the HTTP FLV stream clients.

play {

# whether reduce the sequence header,

# for some client which cannot got duplicated sequence header,

# while the sequence header is not changed yet.

# default: off

reduce\_sequence\_header on;

}

}

### Publish 1st Packet Timeout

vhost \_\_defaultVhost\_\_ {

# the config for FMLE/Flash publisher, which push RTMP to SRS.

publish {

# the 1st packet timeout in ms for encoder.

# default: 20000

firstpkt\_timeout 20000;

}

}

### Publish Normal Timeout

vhost \_\_defaultVhost\_\_ {

# the config for FMLE/Flash publisher, which push RTMP to SRS.

publish {

# the normal packet timeout in ms for encoder.

# default: 5000

normal\_timeout 7000;

}

}

### Debug SRS Upnode

vhost \_\_defaultVhost\_\_ {

# when upnode(forward to, edge push to, edge pull from) is srs,

# it's strongly recommend to open the debug\_srs\_upnode,

# when connect to upnode, it will take the debug info,

# for example, the id, source id, pid.

# please see: https://github.com/ossrs/srs/wiki/v1\_CN\_SrsLog

# default: on

debug\_srs\_upnode on;

}

### UTC Time

# whether use utc\_time to generate the time struct,

# if off, use localtime() to generate it,

# if on, use gmtime() instead, which use UTC time.

# default: off

utc\_time off;

### HLS TS Floor

vhost \_\_defaultVhost\_\_ {

hls {

# whether use floor for the hls\_ts\_file path generation.

# if on, use floor(timestamp/hls\_fragment) as the variable [timestamp],

# and use enahanced algorithm to calc deviation for segment.

# @remark when floor on, recommend the hls\_segment>=2\*gop.

# default: off

hls\_ts\_floor off;

}

}

### HLS Wait Keyframe

vhost \_\_defaultVhost\_\_ {

hls {

# whether wait keyframe to reap segment,

# if off, reap segment when duration exceed the fragment,

# if on, reap segment when duration exceed and got keyframe.

# default: on

hls\_wait\_keyframe on;

}

}

### HttpHooks On HLS Notify

vhost \_\_defaultVhost\_\_ {

http\_hooks {

# when srs reap a ts file of hls, call this hook,

# used to push file to cdn network, by get the ts file from cdn network.

# so we use HTTP GET and use the variable following:

# [app], replace with the app.

# [stream], replace with the stream.

# [ts\_url], replace with the ts url.

# ignore any return data of server.

# @remark random select a url to report, not report all.

on\_hls\_notify http://127.0.0.1:8085/api/v1/hls/[app]/[stream][ts\_url];

}

}

### TCP NoDelay

vhost \_\_defaultVhost\_\_ {

# whether enable the TCP\_NODELAY

# if on, set the nodelay of fd by setsockopt

# default: off

tcp\_nodelay on;

}

### ATC Auto

vhost \_\_defaultVhost\_\_ {

# for play client, both RTMP and other stream clients,

# for instance, the HTTP FLV stream clients.

play {

# whether enable the auto atc,

# if enabled, detect the bravo\_atc="true" in onMetaData packet,

# set atc to on if matched.

# always ignore the onMetaData if atc\_auto is off.

# default: off

atc\_auto off;

}

}

### NGINX RTMP EXEC

NGINX-RTMP支持的EXEC方式，参考[nginx exec](https://github.com/arut/nginx-rtmp-module/wiki/Directives" \l "exec)，SRS只支持常用的几种。下面是exec的支持情况：

1. exec/exec\_publish: 当发布流时调用，支持。
2. exec\_pull: 不支持。
3. exec\_play: 不支持。
4. exec\_record\_done: 不支持。

#### Config

SRS EXEC的配置参考conf/exec.conf，如下：

vhost \_\_defaultVhost\_\_ {

# the exec used to fork process when got some event.

exec {

# whether enable the exec.

# default: off.

enabled off;

# when publish stream, exec the process with variables:

# [vhost] the input stream vhost.

# [port] the intput stream port.

# [app] the input stream app.

# [stream] the input stream name.

# [engine] the tanscode engine name.

# other variables for exec only:

# [url] the rtmp url which trigger the publish.

# [tcUrl] the client request tcUrl.

# [swfUrl] the client request swfUrl.

# [pageUrl] the client request pageUrl.

# @remark empty to ignore this exec.

publish ./objs/ffmpeg/bin/ffmpeg -f flv -i [url] -c copy -y ./[stream].flv;

}

}

## SRS与其他流媒体比较

### 流发分 Stream Delivery

| **Feature** | **SRS** | **NGINX** | **CRTMPD** | **FMS** | **WOWZA** |
| --- | --- | --- | --- | --- | --- |
| RTMP | Stable | Stable | Stable | Stable | Stable |
| HLS | Stable | Stable | X | Stable | Stable |
| HDS | Experiment | X | X | Stable | Stable |
| HTTP FLV | Stable | X | X | X | X |
| HLS(aonly) | Stable | X | X | Stable | Stable |
| HTTP Server | Stable | Stable | X | X | Stable |

### 集群 Cluster

| **Feature** | **SRS** | **NGINX** | **CRTMPD** | **FMS** | **WOWZA** |
| --- | --- | --- | --- | --- | --- |
| RTMP Edge | Stable | X | X | Stable | X |
| RTMP Backup | Stable | X | X | X | X |
| VHOST | Stable | X | X | Stable | Stable |
| Reload | Stable | X | X | X | X |
| Forward | Stable | X | X | X | X |
| ATC | Stable | X | X | X | X |

### 流处理服务 Stream Service

| **Feature** | **SRS** | **NGINX** | **CRTMPD** | **FMS** | **WOWZA** |
| --- | --- | --- | --- | --- | --- |
| DVR | Stable | Stable | X | X | Stable |
| Transcode | Stable | X | X | X | Stable |
| HTTP API | Stable | Stable | X | X | Stable |
| HTTP hooks | Stable | X | X | X | X |
| GopCache | Stable | X | X | Stable | X |
| Security | Stable | Stable | X | X | Stable |
| Token Traverse | Stable | X | X | Stable | X |

### Efficiency

| **Feature** | **SRS** | **NGINX** | **CRTMPD** | **FMS** | **WOWZA** |
| --- | --- | --- | --- | --- | --- |
| Concurrency | 7.5k | 3k | 2k | 2k | 3k |
| MultipleProcess | Experiment | Stable | X | X | X |
| RTMP Latency | 0.1s | 3s | 3s | 3s | 3s |
| HLS Latency | 10s | 30s | X | 30s | 30s |

### Stream Caster

| **Feature** | **SRS** | **NGINX** | **CRTMPD** | **FMS** | **WOWZA** |
| --- | --- | --- | --- | --- | --- |
| Ingest | Stable | X | X | X | X |
| Push MPEGTS | Experiment | X | X | X | Stable |
| Push RTSP | Experiment | X | Stable | X | Stable |
| Push HTTP FLV | Experiment | X | X | X | X |

### Debug System

| **Feature** | **SRS** | **NGINX** | **CRTMPD** | **FMS** | **WOWZA** |
| --- | --- | --- | --- | --- | --- |
| BW check | Stable | X | X | X | X |
| Tracable Log | Stable | X | X | X | X |

### Docs

| **Feature** | **SRS** | **NGINX** | **CRTMPD** | **FMS** | **WOWZA** |
| --- | --- | --- | --- | --- | --- |
| Demos | Stable | X | X | X | X |
| WIKI(EN+CN) | Stable | EN only | X | X | Stable |

### Others

| **Feature** | **SRS** | **NGINX** | **CRTMPD** | **FMS** | **WOWZA** |
| --- | --- | --- | --- | --- | --- |
| ARM/MIPS | Stable | Stable | X | X | X |
| Client Library | Stable | X | X | X | X |

### 市面主要的流媒体服务器对比

目前市面上主流的流媒体服务器，有以Adobe FMS、Real Helix、Wowza为代表的第一代产品，它们的特点是单进程多线程。基于Linux2.7 epoll技术，出现了以多进程单线程为特点的第二代流媒体服务器，NginxRTMP、Crtmpd为其优秀的代表，另外还有基于JAVA的流媒体祖先Red5等。

开源流媒体服务器SRS（Simple RTMP Server），凭借其功能强大、轻量易用、特别适合互动直播等诸多特点备受海内外视频从业者的青睐。蓝汛Chiancache曾用SRS承载其直播边缘分发业务，高升CDN基于SRS搭建其流媒体基础平台，其它还有赛维安讯、VeryCDN、VeryCloud、云博视等也将SRS应用到了自身的业务当中。各家视频云、云计算平台在源站的对接上也非常注重对SRS的支持。SRS作为纯国产的开源Server，在中国流媒体业界实属难能可贵。



BMS（Bravo Media Server）是SRS的商业版，BMS在SRS基础上增强了11项大功能，新增了9个大功能： **注意:表中比较是srs 1.0 release版本，有些功能在srs2.0,srs3.0中以及实现，只是没有release版本， BMS是结合srs2.0 与srs3.0是在这个基础出发出来的。**

BMS和SRS的主要区别如下：

| **Feature** | **SRS** | **BMS** | **Remark** |
| --- | --- | --- | --- |
| 级别 | 工业级 | 工业级 | 发行版本都是工业级集群标准 |
| 开源 | 是 | 否 | BMS为闭源商业软件，提供售前咨询和售后服务， 以及定制化开发，系统对接等 |
| 发行版 | 1.0 | 3.0 | SRS目前发行版为1.0，SRS2为alpha测试版。 BMS合并了SRS2和SRS3的功能，为发行版 |
| 周期 | 1-2年 | 3-6个月 | SRS的版本发行周期为1到2年 |
| 代码量 | 5.95万行 | 13.3万行 | 包含服务器的注释和单元测试 BMS是SRS代码量的2.23倍。 |
| 自动测试 | 无 | 支持 | BMS包含自动化测试系统，每次提交自动回归测试 |

BMS除了有SRS的10个基础功能，还增强了13项大功能，新增了22个大功能。详细对比如下（注意对比的是发行版，即SRS1和BMS3）。

以下是BMS和SRS都有的功能：

| **Feature** | **SRS** | **BMS** | **Remark** |
| --- | --- | --- | --- |
| HTTP回调 | 支持 | 支持 | 和外部业务系统对接 |
| 测速 | 支持 | 支持 | 支持服务器上行和下行速度测试 |
| TS矫正 | 支持 | 支持 | 支持时间戳矫正，避免重推和跳变引起播放器卡死 |
| Gop合并 | 支持 | 支持 | HLS按照GOP输出切片 |
| ATC | 支持 | 支持 | 支持绝对时间戳 |
| 边缘 | 支持 | 支持 | 源站和边缘组成流媒体分发集群 |
| 日志 | 支持 | 支持 | 提供可追溯的排错日志 |
| 采集 | 支持 | 支持 | 将外部流采集到服务器 |
| 转发 | 支持 | 支持 | 将流转发给其他服务器 |
| HDS | 实验性 | 实验性 | Adobe HDS分发 |

以下为BMS增强的功能：

| **Feature** | **SRS** | **BMS** | **Remark** |
| --- | --- | --- | --- |
| 输入 | RTMP | RTMP, ***FLV, RTSP, UDP*** | 推流到服务器的输入方式 |
| 输出 | RTMP, HLS | RTMP, HLS, ***FLV, TS, MP3, HLS+, FLS*** | 服务器分发给客户端的方式 |
| 边缘回源 | RTMP | RTMP, ***FLV, HLS, HLS CUP*** | 边缘支持以RTMP或HTTP FLV/HLS方式回源 |
| DVR | FLV文件 | ***对接观止收录系统*** | 支持录制RTMP到FLV文件 |
| 低延迟 | RTMP(3s+) | **RTMP/*FLV(1s+)*** | 低延迟模式 |
| 转码 | FFMPEG | ***对接观止转码云*** | 转码消耗非常多的系统资源 |
| HTTP API | 简版 | ***完善的API支持*** | API为服务器提供给外部的接口 |
| 下行并发 | 2.7K | ***7K*** | 下行RTMP/FLV的并发 |
| 上行并发 | 1.2K | ***4K*** | 上行推流RTMP的并发 |
| 热备 | RTMP | RTMP, ***HLS*** | 边缘在上层服务器故障时，切换到备用服务器 |
| HTTP | 实验性 | ***商用服务器*** | 内置HTTP服务器，实现HTTP流、API、内存HLS的分发 |
| Gop Cache | 支持 | ***多GOP*** | 缓存最近的Gop，让播放器快速启动(<0.1s)，BMS支持cache多个gop，更快启动 |
| 回源切换 | 错误时切换 | 错误时切换， ***API无缝切换*** | SRS/BMS在错误时切换源站，BMS支持API调用切换源站 |

以下为BMS新增的功能：

| **Feature** | **SRS** | **BMS** | **Remark** |
| --- | --- | --- | --- |
| 源站集群 | 无 | ***支持*** | 基于Redis的源站集群，边缘自动负载均衡和容错 |
| HLS+ | 无 | ***支持*** | 支持流模式分发HLS切片，边缘转封装，同一套流媒体分发（不用走HTTP集群） |
| 时移 | 无 | ***对接观止时移系统*** | 在时移的基础上可以做高级收录和P2P |
| CDN预推 | 无 | ***支持*** | 将HLS预推到CDN节点 |
| 计费 | 无 | ***支持对接计费系统*** | 计费系统做定制和对接 |
| HLS纯音频 | 不支持 | ***支持MP3和AAC*** | HLS纯音频即没有视频流 |
| Vhost转换 | 不支持 | ***支持*** | 边缘回源，以及复杂业务系统需要转换Vhost |
| Kafka | 不支持 | ***支持*** | 对接到Kafka大数据集群，参考BIG技术 |
| 动态配置 | 不支持 | ***支持*** | 通过HTTP API从业务系统动态加载配置 |
| 日志切割 | 不支持 | ***支持*** | nginx风格的日志切割LogRotate |
| HLS CUP | 不支持 | ***支持*** | 支持HLS多源站并发回源 |
| 毫秒开 | 不支持 | ***支持*** | 支持FastStartup毫秒级灌满播放器缓冲区，比秒开更快 |
| 内存转储 | 不支持 | ***支持*** | 支持分析当前服务器的内存消耗MemoryApi，方便运维 |
| 软矩阵 | 不支持 | ***支持*** | 支持M组每组N路输入一路输出，即M\*N输入M输出 |
| 延播 | 不支持 | ***支持*** | 支持延迟播出(流审核)和时移，支持内存和磁盘 |
| FLS | 不支持 | ***支持*** | 支持FLV Live Streaming将HLS的TS换成FLV切片，去掉M3u8索引 |
| Security | 不支持 | ***支持*** | 支持Referer和Token防盗链，踢流和禁播等 |
| BAR | 不支持 | ***已规划*** | 支持应用层回源智能路由 |
| 截图 | 不支持 | ***已规划*** | 支持快速截图Snapshot，供鉴黄和预览等 |
| 音频转码 | 不支持 | ***已规划*** | 支持AT音频转码，speex转aac，拓展flash页面推流方式 |
| 混音 | 不支持 | ***已规划*** | 支持AM多路音频混合，支持连麦和会议等业务 |
| UDP | 不支持 | ***已规划*** | 支持UDP传输，更低级别的系统延迟 |

流媒体Server的话说来也不短，上述列举的目前市面上主流流媒体服务器中，有名副其实的先烈RED5，有生不逢时的CRTMPD，都未大规模商用就不过于讨论了。其中应用最为广泛莫属nginx-rtmp，以下是nginx-rtmp几个盛行于世的重要因素：

* 2012年CDN业务开始极增长，随之直播需求也多了起来，彼时业界都还没有一套公认的特别满意的流媒体服务器；
* Nginx是HTTP领域绝对的霸主，大家（尤其是CDN运维）对Nginx熟悉程度很高，便于上手维护；
* 基于Nginx，直播点播使用一套服务器，这也极具诱惑力，一套管理起来总比多套要简单；
* CDN是靠运维的行当，运维的信心都是长年运出来的，Nginx在图文上那么优秀，Nginx RTMP也差不了。

nginx-rtmp确实生来就自带光环外，性能也的确是高，比Crtmpd还要高。然而，时过境迁，随着互动直播、移动直播的强势兴起的大直播时代，选择nginx-rtmp到底是福还是祸？

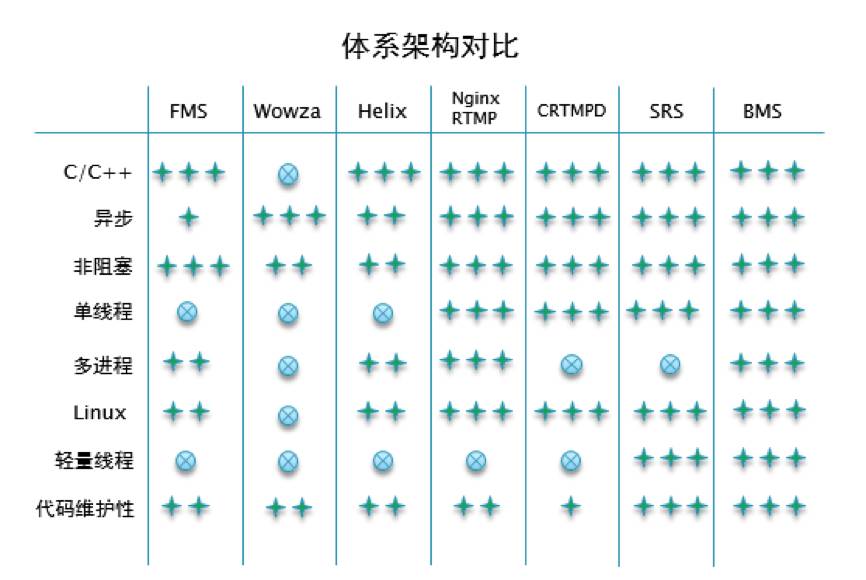
下面小编将从**协议支持、体系架构、核心功能支持、配置运维、性能、服务器日志、数据**这七大维度将目前市面主流的流媒体Server做一个横向对比，供视频从业者根据自身业务场景特性择优选用。

#### 网络协议对比



BMS支持HDS、DASH、RTMPE/S/T等协议的分发，这将支持更多业务应用场景，FLASH P2P的支持能够显著降低网络带宽成本。

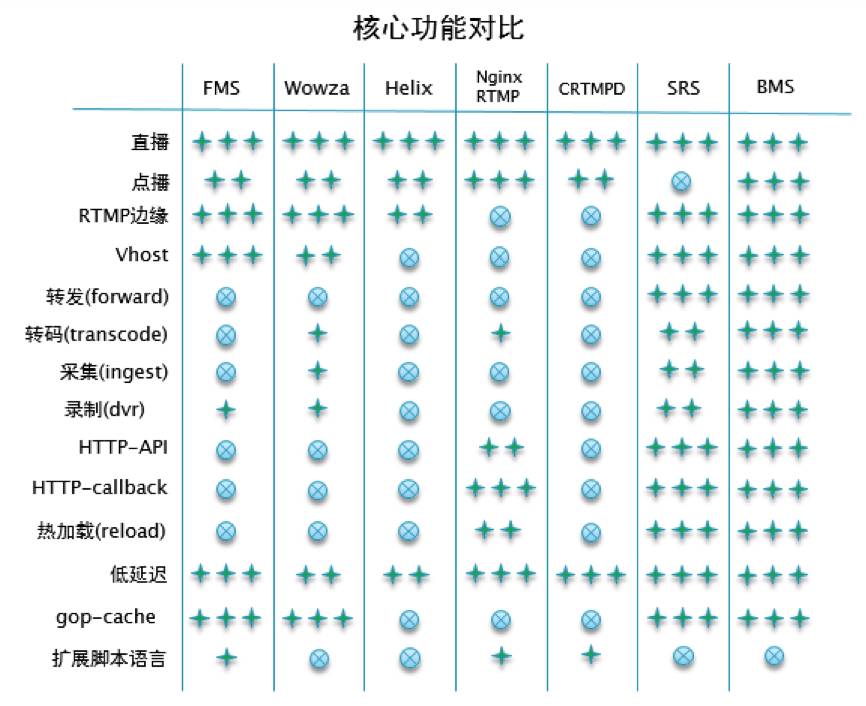
#### 体系架构对比



架构方面，较之于nginx-rtmp的16万行代码，**SRS仅用了6.5万行代码就实现了比nginx-rtmp 多了230%的功能**，nginx-rtmp注释率为3%，而SRS是23.7%。由此可见SRS在体系架构上的轻，Simple。

BMS在SRS的基础上新增了**多进程支持、源站集群、动态配置、可追溯日志**等方面能力。源站集群子系统打通了跨网跨地区的源站分布式部署难题；动态配置子系统从业务系统读取配置，依据更新机制动态更新配置，保证直播业务配置变化时依然不中断；端到端的可追溯日志及监控排错子系统将直播故障定位时间缩短到了分钟级别。

#### 核心功能对比



核心功能方面，BMS支持了**当期互动直播、移动直播急需的大规模直播流实时转码、大规模录制、秒级低延迟、HLS+、并发回源**等其它所有流媒体系统不具备的功能。HLS+基于每个播放请求实现了流媒体的“虚拟连接 ”（UUID标识），在减小回源量、排错、防盗链、移动Web端低延迟等方面具有诸多优势。并发回源能够解决回源网络状况差、跨国传输丢包严重等方面能够显著提升回源质量。

#### 配置运维对比

以下仅是流媒体众多配置之中几个常用例子，运维日常工作中，需要操作的配置数量更多。

##### （1）vhost配置

**FMS**

拷贝默认vhost目录：sudo cp -r conf/\_defaultRoot\_/\_defaultVHost\_ conf/\_defaultRoot\_/bravo.sina.com

**nginx-rtmp**

不支持

**SRS/BMS**

动态获取配置文件：vhost bravo.sina.com { }

结论：BMS动态获取配置最简单

##### （2）app配置

**FMS**

拷贝默认app目录：cp applications/live applications/mylive -r

**nginx-rtmp**

修改配置文件，增加如下内容：application live {  live on; }

**SRS/BMS**

无需配置

结论：BMS无需配置，最简单

##### （3）http配置

在输出为hls、http-flv等基于http协议的直播流时，需要配置http服务

**FMS**

配置FMS内置的Apache服务器文件：Apache2.2/conf/httpd.conf

再修改如下字段：

<Location /hds-live>

    HttpStreamingEnabled true

    HttpStreamingLiveEventPath "../applications"

    HttpStreamingContentPath "../applications"

    HttpStreamingF4MMaxAge 2

    HttpStreamingBootstrapMaxAge 2

    HttpStreamingFragMaxAge -1

    Options -Indexes FollowSymLinks

</Location

**nginx-rtmp**

nginx本身就是一个http服务器，

修改其配置文件：

conf/nginx.conf

设置端口和根目录：

http {

    include       mime.types;

    default\_type  application/octet-stream;

    sendfile        on;

    keepalive\_timeout  65;

    server {

        listen       80;

        server\_name  localhost;

        location /dash {

            root /tmp;

            add\_header Cache-Control no-cache;

        }

    }

}

**SRS/BMS**

修改其配置文件：

conf/http.hls.conf

设置端口和根目录：

http\_stream {

    enabled         on;

    listen          8080;

    dir             ./objs/nginx/html;

}

**结论：nginx-rtmp需指定与app对应的ts文件存放目录，SRS/BMS会自动生成，更简单。**

##### （4）推流、播放URL配置

RTMP直播时，各大服务器推流、播流URL均为：

rtmp://server\_ip\_or\_dns/app/stream

用作HLS直播时，

**FMS**

推流域名：

rtmp://fms-ip-or-dns/app/stream?adbe-live-event=liveevent

播流域名：

http://fms-ip-or-dns/hds-live/app/\_definst\_/liveevent/stream.f4m

**nginx-rtmp**

推流域名：

rtmp://server\_ip\_or\_dns/app/stream

播流域名：

http://server\_ip\_or\_dns/app/stream.m3u8

**SRS/BMS**

同nginx-rtmp

**结论：nginx-rtmp、SRS/BMS均简单，FMS较复杂。**

#### 性能

**先说结论：**

**SRS单进程能支持9000并发，nginx-rtmp单进程最多支持3000个，单进程的性能SRS是nginx-rtmp的三倍。单进程性能SRS > nginx-rtmp > crtmpd > wowza > fms > RED5**

 再例举SRS性能如此高的几个原因：

1. st-load，这个是SRS能做到高性能的最重要的原因，一个st-load可以模拟2000+的客户端，如果没有st-load，如何知道系统的性能瓶颈在哪里？总不能打开3000个flash页面播放rtmp流吧？开启3000个ffmpeg来抓流？高性能不是想象和猜测出来的，而是反复测试、调试和改进出来的。

2. gperf/gprof性能，编译SRS时，就可以打开gcp或者gprof的性能分析选项，非常方便的拿到数据。缩短了改进和优化开发周期。

3. 引用计数的msgs避免内存拷贝。

4. 使用writev发送chunked包，避免消息到chunked包的内存拷贝。

5. mw(merged-write)技术，即一次发送多个消息。

6. 减少timeout recv，每个连接都是一个st-thread在服务。

7. fast buffer和cache。

8. vector还是list？vector！vector比list高10%性能。

#### 服务器日志

日志是定位故障的唯一途径，定位故障才能快速排错。可以这么说，对于直播，10分钟的排错，谁都会觉得长。然而，当前的视频云或CDN，谁又能做到10分钟呢？

来看看日志吧。

FMS的日志是这样的，恕我愚钝，你能看得出什么信息么？

2015-03-24 12:23:58 3409 (s)2641173 Accepted a connection from IP:192.168.1.141, referrer:http://www.ossrs.net/players/srs\_player/release/srs\_player.swf?\_versi>

702111234525315439     3130         3448         normal      livestream         -        -         rtmp://192.168.1.185:1935/live/livestream     rtmp://192.168.1.185:1935/live/livestream        -        flv     -        -        0       -        0       0         -        -    http://www.ossrs.net/players/srs\_player.html?vhost=dev&stream=livestream&server=dev&port=1935    -1      -1.000000

crtmpd的日志详细，但我又愚钝，若是上千人在线，你又能看出什么有用的东西么？

/home/winlin/tools/crtmpserver.20130514.794/sources/thelib/src/netio/epoll/iohandlermanager.cpp:120Handlers count changed: 15->16 IOHT\_TCP\_CARRIER

/home/winlin/tools/crtmpserver.20130514.794/sources/thelib/src/netio/epoll/tcpacceptor.cpp:185Client connected: 192.168.1.141:54823 -> 192.168.1.173:1935

/home/winlin/tools/crtmpserver.20130514.794/sources/applications/appselector/src/rtmpap

# 常用的直播平台网站

<https://github.com/ossrs/srs> SRS官网

<https://github.com/arut/nginx-rtmp-module> nginx rtmp官网

<https://www.qcloud.com/product/LVB.html> 腾讯云直播

<https://www.aliyun.com/product/live?spm=5176.7960203.237031.164.Ve6N2d> 阿里云直播

<https://bce.baidu.com/product/lss.html> 百度云直播

<http://www.lecloud.com/zh-cn/product/live.html> 乐视云直播

<https://www.upyun.com/products/video.html> 又拍云直播

<http://www.qiniu.com/products/live> 七牛云直播

<http://xycdn.com/site/solution> 星域云(讯雷旗下的公司)

<http://www.easydarwin.org> RTSP开源平台