从 0 开始打造万人直播系统

直播系统概述

直播系统历史

直播是一种媒介播放传播的方式，直播就是你实时能看到的电视或网络画面，我们最开始熟知的是各种电视直播。直播的特点是真实性、实时性、互动性，所谓眼见为实，每个画面也都是即时的，不能更改的，网络直播则互动性体现的更为淋漓尽致。

直播最开始是电视直播，后来基于网络的发达，开始了网络游戏直播，再到移动 4G 的发展，直播开始更多走向户外，短视频也随之兴起，网络直播开始疯狂成长，最后到全民皆可直播的盛况，不仅仅是视频网站，各种电商网站也都开启了直播卖货的模式。

从网络直播的发展来看，大致有这么几个阶段。

一、PC 端秀场直播时代

从 2005 年开始，各大视频网站展开流量大战。其中六间房、9158，以美女主播为卖点，发展起秀场直播聊天室，而 YY 从语音软件进军秀场直播领域。那个时代的末尾，尽管智能手机逐渐走向普及，4G 也在萌芽，但大部分用户的碎片化时间还集中在 PC 端。

二、游戏直播时代

2014 年，YY 剥离游戏直播业务成立虎牙直播，斗鱼直播从 Acfun 独立出来。2015 年，龙珠和熊猫直播通过抢占赛事资源、挖掘人气主播等方式快速抢占市场。那为何把这个阶段单独分成“游戏时代”呢？一是因为此阶段的电竞赛事频繁，二是相比起其他领域的直播来讲，游戏直播的用户粘性更高，而且游戏直播的时效性和观赏性更加优秀。不过这个阶段，随着手机直播的萌芽，PC 端的用户已经呈下降趋势，更多的用户流向了以手机为主的移动平台。

三、移动直播时代

2015 年末，六间房、斗鱼、映客、快手、抖音、花椒、易直播等手机平台的加入，使得网络视频直播系统的应用场景更加多元化。技术瓶颈的打破，不再使游戏直播“一家独大”，也让泛娱乐领域也加入了直播，直播内容也更综合，开始往户外、财经、旅游、金融、购物等各行各业发展。

而随着快手和抖音的发展，移动网络的发达，各种形式的短视频兴起，推动着直播内容的丰富性，很多素人也在这个时候因为视频内容而爆火，逐渐的拥有了直播的功能，观众可以选择的内容更多，形式也更丰富，素人网红也越来越多。

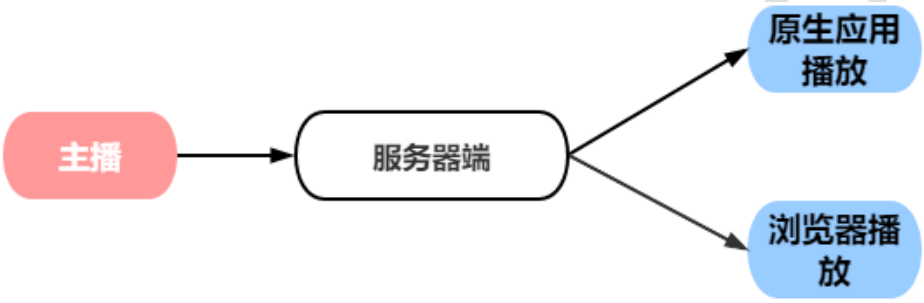
四、直播的继续发展

一对一直播开始兴起，为越来越多的直播用户提供了私密空间，也使主播的入驻门槛更低，还让更多的行业看到了额外的道路（一对一心理辅导、一对一财经直播间等）。

从技术上来看，VR 将扁平的音视频和图像转变为立体的沉浸式体验，大大提升了用户使用感受，但是在开发还是在使用上，技术成本比较高，而且 VR 技术带给部分用户的生理“眩晕感”，还需要解决。VR 直播可能是直播继续发展的方向，但是还需时间验证。

直播模型原理与架构

一个通用的直播模型一般包括三个模块:主播方、服务器端和播放端。



首先是主播方，它是产生视频流的源头，由一系列流程组成：

第一，通过一定的设备来采集数据；

第二，将采集的这些视频进行一系列的处理，比如水印、美颜和特效滤镜等处理；

第三，将处理后的结果视频编码压缩成可观看可传输的视频流；

第四，分发推流，即将压缩后的视频流通过网络通道传输出去。

其次是播放端，播放端功能有两个层面，第一个层面是关键性的需求；另一层面是业务层面的。

先看第一个层面，它涉及到一些非常关键的指标，比如秒开，在很多场景当中都有这样的要求，然后是对于一些重要内容的版权保护。为了达到更好的效果，我们还需要配合服务端做智能解析，这在某些场景下也是关键性需求。

第二个层面也即业务层面的功能，对于一个社交直播产品来说，在播放端，观众希望能够实时的看到主播端推过来的视频流，并且和主播以及其他观众产生一定的互动，因此它可能包含一些像点赞、聊天和弹幕这样的功能，以及礼物这样更高级的道具。

直播服务器端提供的最核心功能是收集主播端的视频推流，并将其放大后推送给所有观众端。除了这个核心功能，还有很多运营级别的诉求，比如鉴权认证，视频连线和实时转码，自动鉴黄，多屏合一，以及云端录制存储等功能。

另外，对于一个主播端推出的视频流，中间需要经过一些环节才能到达播放端，因此对中间环节的质量进行监控，以及根据这些监控来进行智能调度，也是非常重要的诉求。

实际上无论是主播端还是播放端，他们的诉求都不会仅仅是拍摄视频和播放视频这么简单。在这个核心诉求被满足之后，还有很多关键诉求需要被满足。比如，对于一个消费级的直播产品来说，除了这三大模块之外，还需要实现一个业务服务端来进行推流和播放控制，以及所有用户状态的维持。如此，就构成了一个消费级可用的直播产品。

很明显数据的流向应该是：录制->编码->网络传输->解码->播放。

直播系统在其中就要负责：采集、前处理、编码、传输、解码和渲染。在两端传输的过程中再加上一个服务端处理。

整个流程中就要解决：

怎样录制直播视频？

怎样实时上传直播视频？

怎样播放直播视频？

等等这一系列的问题，本课程中就会带着大家来一一解决这些问题，搭建属于自己的直播平台。

不过直播平台和音视频的开发不是 Java 语言擅长的领域，这部分主要是 C/C++为主，特别是音视频的开发，虽然比不上操作系统、游戏引擎的开发难度，但是在 C/C++领域也能算的上是难点之一。

不过不用担心，本课程作为主要面向 Java 程序员的课程，会尽量屏蔽底层的复杂，从 0 开始带大家领略直播系统的开发，需要的前置技术主要也就是基础的 JavaScript、Nginx 的操作与配置即可。

浏览器就能完成音视频直播?

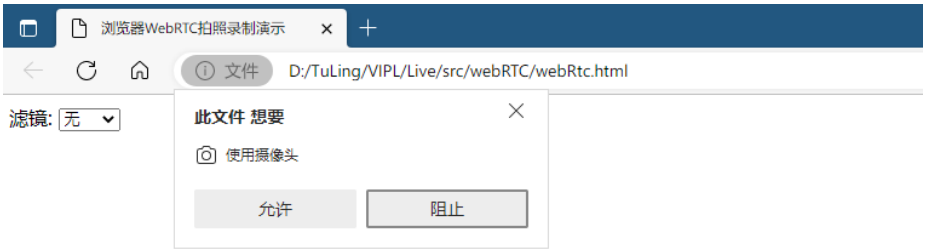
在我们一般的印象中，音视频的播放和录制需要使用原生的应用程序，不管是 PC 还是手机都是如此，不这样做似乎性能跟不上。但其实用浏览器就能做到，所以我们就用浏览器搭建我们最开始的音视频直播系统，方便快捷，易于上手，而且搭建的过程中还可以熟悉最基本的一些音视频的术语和原理。

如何用浏览器做到呢？“浏览器＋WebRTC”就是 Google 给出的答案。2011 年，Google 创立了 WebRTC 项目，其愿景就是可以在浏览器之间快速地实现音视频通信。

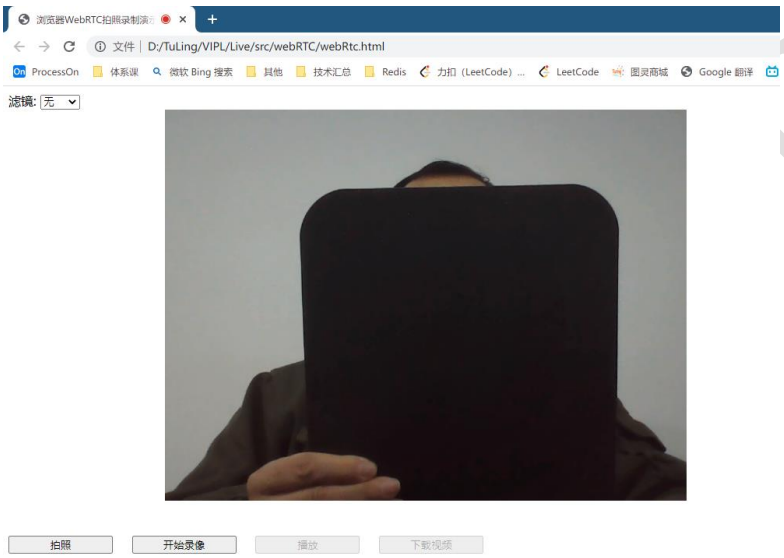
随着 WebRTC 1.0 规范的推出，现在主流浏览器 Chrome、Firefox、Safari 以及 Edge 都已经支持了 WebRTC 库。换句话说，在这些浏览器之间进行实时音视频通信已经很成熟了。

浏览器拍照录制演示

用浏览器＋WebRTC 可以做到很多有趣的事情，我们先来看看效果



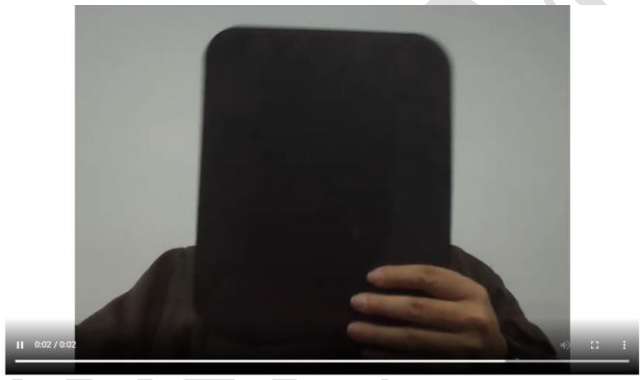
允许使用摄像头后



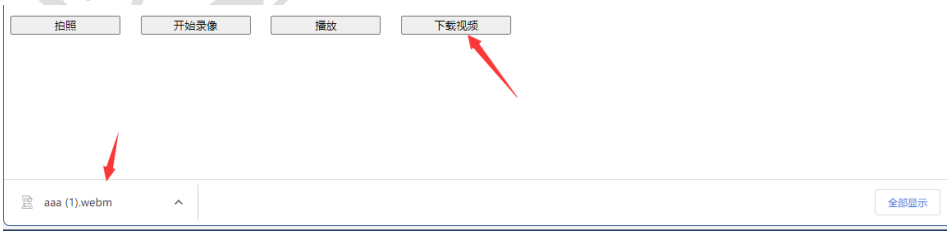
还可以使用各种滤镜

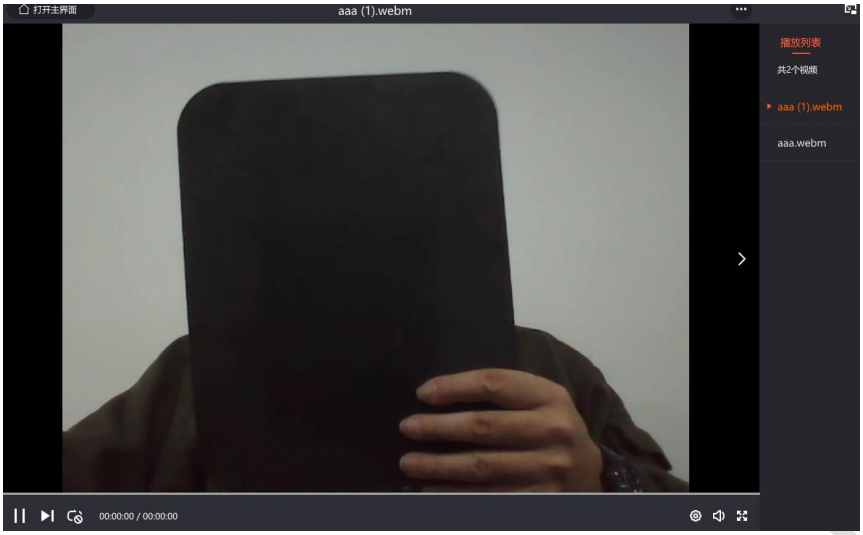
顺便拍个照：

同时也支持录像和录像后的播放：



还可以将录像后的视频文件下载，并用其他播放器打开





我们来看看如何实现，但是要播出音视频，首先肯定要采集音视频，还需要了解一些音视频基本概念。

音视频基本概念

摄像头：这个不用多说，用于捕捉（采集）图像和视频。

帧率：是以帧称为单位的位图图像连续出现在显示器上的频率（速率），以赫兹（Hz）表示，单位为秒。我们在中学物理中知道，人眼观看电影感觉流畅一般要求的帧率为 24Hz，一般的 FPS 游戏高的帧率可以得到更流畅、更逼真的动画。一般来说 30fps 就是可以接受的，但是将性能提升至 60fps 则可以明显提升交互感和逼真感，但是一般来说超过 75fps 一般就不容易察觉到有明显的流畅度提升了。

摄像头一秒钟采集图像的次数同样也称为帧率。现在的摄像头功能已非常强大，一般情况下，一秒钟可以采集 30 张以上的图像，一些好的摄像头甚至可以采集 100 张以上。我们把帧率越高，视频就越平滑流畅。然而，在直播系统中一般不会设置太高的帧率，因为帧率越高，占的网络带宽就越多。

分辨率：摄像头除了可以设置帧率之外，还可以调整分辨率。我们常见的分辨率有 2K、1080P、720P、420P 等。分辨率越高图像就越清晰，但同时也带来一个问题，即占用的带宽也就越多。所以，在直播系统中，分辨率的高低与网络带宽有紧密的联系。也就是说，分辨率会跟据你的网络带宽进行动态调整。

宽高比：分辨率一般分为两种宽高比，即 16:9 或 4:3。4:3 的宽高比是从黑白电视而来，而 16:9 的宽高比是从显示器而来。现在一般情况下都采用 16:9 的比例。

麦克风：这个不用多说，用于采集音频数据。它与视频一样，可以指定一秒内采样的次数，称为采样率。每个采样用几个 bit 表示，称为采样位深或采样大小。

轨（Track ）：WebRTC 中的“轨”借鉴了多媒体的概念。火车轨道的特性你应该非常清楚，两条轨永远不会相交。“轨”在多媒体中表达的就是每条轨数据都是独立的，不会与其他轨相交，如 MP4 中的音频轨、视频轨，它们在 MP4 文件中是被分别存储的。

流(Stream)：可以理解为容器。在 WebRTC 中，“流”可以分为媒体流(MediaStream)和数据流(DataStream)。其中，媒体流可以存放 0 个或多个音频轨或视频轨；数据流可以存 0 个或多个数据轨。

实现解析

上述的功能是如何做到呢？Html 页面本身是很简单的，Html 代码中包含了我们在前面看到的音视频的操控区域，图片显示区域和录像回放区域，其中关键 Html 标签我们在后面会进行讲解。

Html 代码中比较关键的是 JS 文件

```
<script src="https://webrtc.github.io/adapter/adapter-latest.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/2.0.3/socket.io.js"></script>
```

adapter-latest.js 就是我们 WebRTC 相关的 js 文件，而 socket.io.js 则是和音视频在网络上传输相关的 js 文件。

```
<script src="./js/photo-client.js"></script>
<script src="./js/record-client.js"></script>
```

photo-client.js 和 record-client.js 则是我们自己的 js 文件，很明显这两个就是我们使用 WebRTC 进行音视频的代码。photo-client.js 主要负责音视频设备的管理、拍照功能，record-client.js 主要负责录像功能。

来看看具体的实现，分为采集和播放两个部分来讲解。

WebRTC 设备管理

既然要进行音视频，肯定要知道机器上有哪些音视频相关的设备，先了解几个 WebRTC 关于设备的基本概念和接口以及函数。

MediaDevices，该接口提供了访问（连接到计算机上的)媒体设备(如摄像头、麦克风)以及截取屏幕的方法。实际上，它允许你访问任何硬件媒体设备。而咱们要获取可用的音视频设备列表，就是通过该接口中的方法来实现的。

MediaDeviceInfo，它表示的是每个输入/输出设备的信息。包含以下三个重要的属性：

deviceID，设备的唯一标识;

label，设备名称;

kind，设备种类，可用于识别出是音频设备还是视频设备，是输入设备还是输出设备。

检测和获取音视频设备

我们在浏览器上通过 WebRTC 可以获取音视频设备列表的接口：

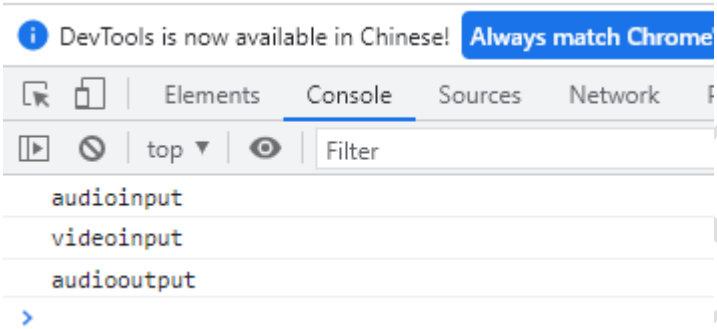
MediaDevices.enumerateDevices()

通过调用 MediaDevices 的 enumerateDevices()方法就可以获取到媒体输入和输出设备列表，例如：麦克风、相机、耳机等。

该函数返回的是一个 Promise 对象。我们只需要向它的 then 部分传入一个函数，就可以通过该函数获得所有的音视频设备信息了。

传入的函数有一个参数，它是一个 MediaDeviceInfo 类型的数组，用来存放 WebRTC 获取到的每一个音视频设备信息。比如：

```
navigator.mediaDevices.enumerateDevices()
.then(function(deviceInfos) {
deviceInfos.forEach(function(deviceInfo) {
  console.log(deviceInfo.kind);
});
})
.catch(function(err) {
  console.log(err.name + ": " + err.message);
});
```



通过上面的代码，浏览器检测到机器上有一个音频输入、一个音频输出、一个视频输入。

如何采集

getUserMedia 方法

在浏览器中访问音视频设备非常简单，只要调用 getUserMedia 这个 API 即可。该 API 的基本格式如下：

navigator.mediaDevices.getUserMedia(constraints)

它返回一个 JS 中的 Promise 对象。如果 getUserMedia 调用成功，则可以通过 Promise 获得 MediaStream 对象，也就是说现在我们已经从音视频设备中获取到音视频数据了。

如果调用失败，比如用户拒绝该 API 访问媒体设备（音频设备、视频设备），或者要访问的媒体设备不可用，则返回的 Promise 会得到 PermissionDeniedError 或 NotFoundError 等错误对象。

MediaStreamConstraints 参数

getUserMedia 方法有一个输入参数 constraints，其类型为 MediaStreamConstraints。它可以指定 MediaStream 中包含哪些类型的媒体轨(音频轨、视频轨），并且可为这些媒体轨设置一些限制。

按照 WebRTC 1.0 规范对 MediaStreamConstraints 的定义

dictionary MediastreamConstraints {

(boolean or MediaTrackConstraints) video = false,

(boolean or MediaTrackConstraints) audio = false

};

该结构可以指定采集音频还是视频，或是同时对两者进行采集，比如：

```
const mediaStreamContrains = {
  video: true,
  audio: true
};
```

并且通过该结构，还能做进一步的设定：

```
var constraints = {
    video : {
        width: 640,
        height: 480,
        frameRate:15,
        facingMode: 'enviroment'
    },
    audio : false
}
```

我们在 JS 脚本中，设置了视频的帧率 15 帧每秒；宽度是 640，高度是 480。音频未开启。

其实音视频的设定远不止这些，比如音频还可以设置为开启回音消除、降噪以及自动增益功能，视频还可以设置是否启用裁剪、所选摄像头等等。我们 JS 脚本中的 facingMode 表示所选摄像头，string 类型，可选的值：user（前置摄像头）、environment（后置摄像头）、left（前置左侧摄像头）、right（前置右置摄像头），在手机中可以通过设置该参数来切换摄像头。因为我们目前是在 PC 上，所以这个选项其实没什么用。

更详细的设置大家可以参考 WebRTC 规范：
https://www.w3.org/TR/2020/PR-webrtc-20201215/

显示视频和图片

不管是图片还是视频，不管是采集的还是从远程服务器获得，我们希望能在浏览器上显示。怎么做呢？

视频播放

首先要在 Html 页面中使用 HTML5 的视频标签<video>：

```
<video autoplay playsinline id="player"></video>
```

使用这个标签不仅可以播放多媒体文件，还可以用于播放采集到的数据。其参数含义如下：

autoplay，表示当页面加载时可以自动播放视频；

playsinline，表示在 HTML5 页面内播放视频，而不是使用系统播放器播放视频。

而在我们的 JS 文件（photo-client.js）中，我们首先利用上面所说的 getUserMedia()方法在用户允许访问，且设备可用后访问摄像头后，在 then 方法中调用 gotMediaStream 方法。

```
navigator.mediaDevices.getUserMedia(constraints)
    .then(gotMediaStream)
    .catch(handleError);
```

gotMediaStream 方法输入参数为 MediaStream 对象，该对象中存放着 getUserMedia 方法采集到的音视频轨。我们将它作为视频源赋值给 HTML5 的 video 标签的 srcObject 属性。这样在 HTML 页面加载之后，就可以在该页面中看到摄像头采集到的视频数据了。

```
var videoplay = document.querySelector('video#player');

function gotMediaStream(stream){
    var videoTrack = stream.getVideoTracks()[0];

    window.stream = stream;
    videoplay.srcObject = stream;
}
```

至于视频显示时的滤镜功能，则是通过 CSS 来控制的，就定义在 html 文件中，大家自行查看即可。

拍照

显示图片似乎很容易，我们常用 image 标签。但是这个标签要求图片有确定的 url 链接，可能来自固定的图片，也可能来自服务器的图片流。但是我们这里需要从视频流中获取正在显示的视频帧或图片，这个标签不是太合适。

实际上，浏览器提供了一个非常强大的对象，称为 Canvas。可以把它想像成一块画布，既可以在这块画布上画上点、线或各种图形，也可以将一幅画直接绘制到上面。

```
<div>
    <canvas id="picture"></canvas>
</div>
```

在我们拍照按钮的 onclick 事件中：

```
snapshot.onclick = function() {
    picture.className = filtersSelect.value;
    picture.getContext('2d').drawImage(videoplay, 0, 0, picture.width, picture.height);
}
```

获得 HTML 中的 Canvas 标签，命名为 picture，并设置了 Canvas 的宽高；然后调用 Canvas 上下文的 drawImage 方法，这样就可以从视频流中抓取当时正在显示的图片。

可以看到 drawImage 方法有四个参数：

image：可以是一幅图片，也可以是一个 Video 元素。

dx, dy：图片起点的 x、y 坐标。

dWidth：图片的宽度。

dHeight：图片的高度

视频录屏重播与下载

在音视频会议、在线教育等系统中，录制是一个特别重要的功能。尤其是在线教育系统，基本上每一节课都要录制下来，以便学生可以随时回顾之前学习的内容。

实现录制功能有很多方式，一般分为服务端录制和客户端录制。

服务端录制：优点是不用担心客户因自身电脑问题造成录制失败（如磁盘空间不足），也不会因录制时抢占资源（CPU 占用率过高）而导致其他应用出现问题等；缺点是实现的复杂度很高。

客户端录制：优点是方便录制方操控，并且所录制的视频清晰度高，实现相对简单。本地高清的视频在上传服务端时由于网络带宽不足，视频的分辨率很有可能会被自动缩小到了 640x360，这就导致用户回看时视频特别模糊，用户体验差。不过客户端录制也有很明显的缺点，其中最主要的缺点就是录制失败率高。因为客户端在进行录制时会开启第二路编码器，这样会特别耗 CPU。而 CPU 占用过高后，就很容易造成应用程序卡死。除此之外，它对内存、硬盘的要求也高。

我们现在学习有关浏览器+WebRTC 的知识，所以我们只实现客户端录制音视频流。

既然是录制视频，必然需要存放音视频数据，在 JavaScript 中，有很多用于存储二进制数据的类型，这些类型包括:ArrayBuffer、ArrayBufferView 和 Blob。这三者的区别请自行查阅 JS 技术文档。

WebRTC 录制音视频流之后，最终是通过 Blob 对象将数据保存成多媒体文件的。Blob (Binary Large Object)是 JavaScript 的大型二进制对象类型，而它的底层是由 ArrayBuffer 对象的封装类实现的。

Blob 对象的格式如下:

var blob = new Blob(array, options);

其中，array 可以是 ArrayBuffer、ArrayBufferView、Blob、DOMString 等类型；option，用于指定存储成的媒体类型。

由谁来负责把音视频数据放入 Blob 对象呢？WebRTC 为我们提供了一个非常方便的类，即 MediaRecorder。创建 MediaRecorder 对象的格式如下:

var mediaRecorder = new MediaRecorder(stream[, options]);

参数含义如下:

stream，通过 getUserMedia 获取的本地视频流。

options，可选项，指定视频格式、编解码器、码率等相关信息，如 mimeType: 'video/webm; codecs=vp8 '。

MediaRecorder 对象还有一个特别重要的事件，即 ondataavailable 事件。当

MediaRecoder 捕获到数据时就会触发该事件。通过它，我们才能将音视频数据录制下来。

我们在 Html 中定义了三个 button

```
<button id="record">开始录像</button>
<button id="recplay" disabled>播放</button>
<button id="download" disabled>下载视频</button>
```

用于开启录制、用于播放录制内容和将录制的视频下载。三个 button 的 click 事件则定义在 record-client.js 中。

录制视频关键的函数就是 startRecord()

```
var buffer;
var mediaRecorder;

function handleDataAvailable(e) {
  if(e && e.data && e.data.size > 0) {
    buffer.push(e.data);
  }
}

function startRecord() {

  buffer = [];

  var options = {
    mimeType: 'video/webm;codecs=vp8'
  }

  if(!MediaRecorder.isTypeSupported(options.mimeType)) {
    console.error(`${options.mimeType} is not supported!`);
    return;
  }

  try{
    mediaRecorder = new MediaRecorder(window.stream, options);
  }catch(e) {
    console.error('Failed to create MediaRecorder:', e);
    return;
  }

  mediaRecorder.ondataavailable = handleDataAvailable;
  mediaRecorder.start(10);

}
```

可以看到，startRecord 中使用了我们上面说明的 MediaRecorder，并在 ondataavailable 事件中，将数据 push 进事先定义好的 buffer。

而播放和下载都是对这个 buffer 进行处理

```
btnPlay.onclick = ()=> {
  var blob = new Blob(buffer, {type: 'video/webm'});
  recvideo.src = window.URL.createObjectURL(blob);
  recvideo.srcObject = null;
  recvideo.controls = true;
  recvideo.play();
}

btnDownload.onclick = ()=> {
  var blob = new Blob(buffer, {type: 'video/webm'});
  var url = window.URL.createObjectURL(blob);
  var a = document.createElement('a');

  a.href = url;
  a.style.display = 'none';
  a.download = 'aaa.webm';
  a.click();
```

浏览器进行桌面共享

无论是做音视频会议，还是做远程教育，共享桌面都是一个必备功能。浏览器通过 WebRTC 也能共享桌面。

共享桌面的基本原理

传统共享桌面

共享桌面的基本原理其实非常简单：

对于共享者，每秒钟抓取多次屏幕（可以是 3 次、5 次等），每次抓取的屏幕都与上一次抓取的屏幕做比较，取它们的差值，然后对差值进行压缩;如果是第一次抓屏或切幕的情况，即本次抓取的屏幕与上一次抓取屏幕的变化率超过 80%时，就做全屏的帧内压缩，其过程与 JPEG 图像压缩类似（有兴趣的可以自行学习)。最后再将压缩后的数据通过传输模块传送到观看端；数据到达观看端后，再进行解码，这样即可还原出整幅图片并显示出来。

对于远程控制端，当用户通过鼠标点击共享桌面的某个位置时，会首先计算出鼠标实际点击的位置，然后将其作为参数，通过信令发送给共享端。共享端收到信令后，会模拟本地鼠标，即调用相关的 API，完成最终的操作。一般情况下，当操作完成后，共享端桌面也发生了一些变化。

通过上面的描述，可以总结出共享桌面的处理过程为：抓屏、压缩编码、传输、解码、显示、控制这几步。

对于共享桌面，很多人比较熟悉的可能是 RDP (Remote Desktop Protocal)协议，它是 Windows 系统下的共享桌面协议；还有一种更通用的远程桌面控制协议——VNC (Virtual Network Console)，它可以实现在不同的操作系统上共享远程桌面，像 TeamViewer、RealVNC 都是使用的该协议。

其实在 WebRTC 中也可以实现共享远程桌面的功能。但 WebRTC 的远程桌面不需要远程控制，所以其处理过程使用了视频的方式，而非传统意义上的 RDP/VNC 等远程桌面协议，仍然属于音视频采集的范畴，但是这次采集的不是音视频数据而是桌面。

WebRTC 共享桌面

WebRTC 在桌面数据处理的有好几个环节：

第一个环节，共享端桌面数据的采集。WebRTC 对于桌面的采集与 RDP/VNC 使用的技术是相同的，都是利用各平台所提供的相关 API 进行桌面的抓取。以 Windows 为例，可以使用下列 API 进行桌面的抓取。

BitBlt : XP 系统下经常使用，在 vista 之后，开启 DWM 模式后，速度极慢。

Hook :一种黑客技术，实现稍复杂。

DirectX:由于 DirectX 9/10/11 之间差别比较大，容易出现兼容问题。最新的 WebRTC 都是使用的这种方式

GetWindowDC:可以通过它来抓取窗口。

第二个环节，共享端桌面数据的编码。WebRTC 对桌面的编码使用的是视频编码技术，即 H264/VP8 等;但 RDP/VNC 则不一样，它们使用的是图像压缩技术。使用视频编码技术的好处是压缩率高，而坏处是在网络不好的情况下会有模糊等问题。

第三个环节，传输。编码后的桌面数据会通过流媒体传输协议发送到观看端。对于 WebRTC 来说，当网络有问题时，数据是可以丢失的。但对于 RDP/VNC 来说，桌面数据一定不能丢失。

第四个环节，观看端解码。WebRTC 对收到的桌面数据通过视频解码技术解码，而 RDP/VNC 使用的是图像解码技术（可对比第二个环节)。

第五个环节，观看端渲染。一般会通过 OpenGL/D3D 等 GPU 进行渲染，这个 WebRTC 与 RDP/VNC 都是类似的。

实现桌面共享

在实现上桌面共享和视频采集非常相似，视频采集中使用的是 getUserMedia 函数，抓取桌面则使用 getDisaplayMedia

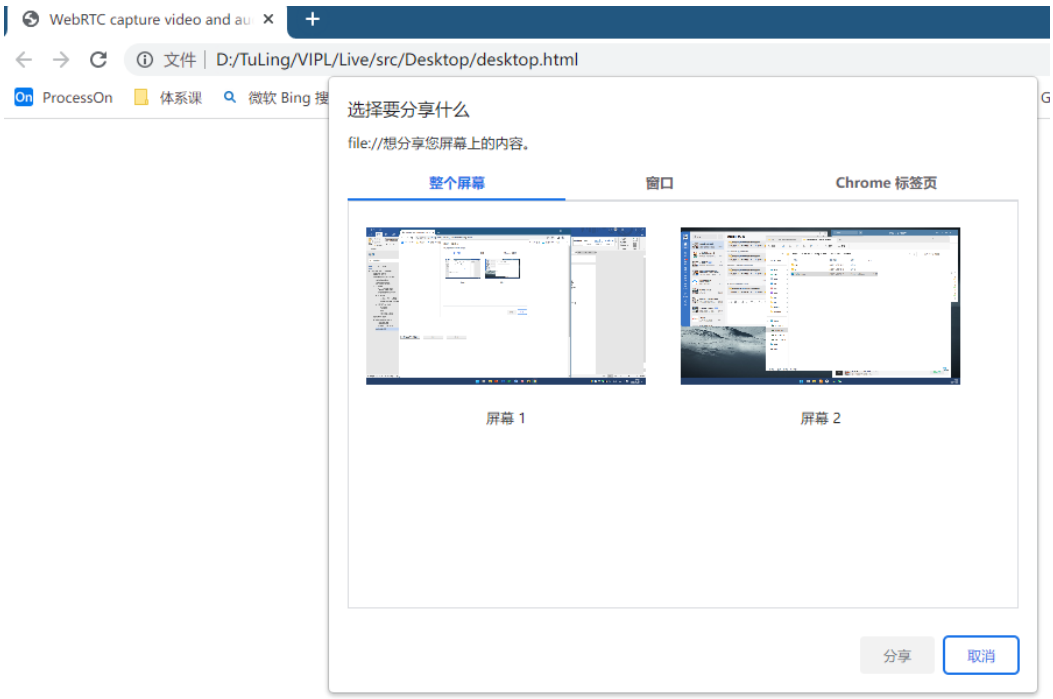
```
var constraints = {
  video : {
    width: 640,
    height: 480,
    frameRate:15
  },
  audio : false
}

navigator.mediaDevices.getDisplayMedia(constraints)
  .then(gotMediaStream)
  .catch(handleError);
```

这两个 API 都需要一个 constraints 参数来对采集的桌面/视频做一些限制。但需要注意的是，在采集视频时，参数 constraints 也是可以对音频做限制的，而在桌面采集的参数里却不能对音频进行限制了，也就是说，不能在采集桌面的同时采集音频。

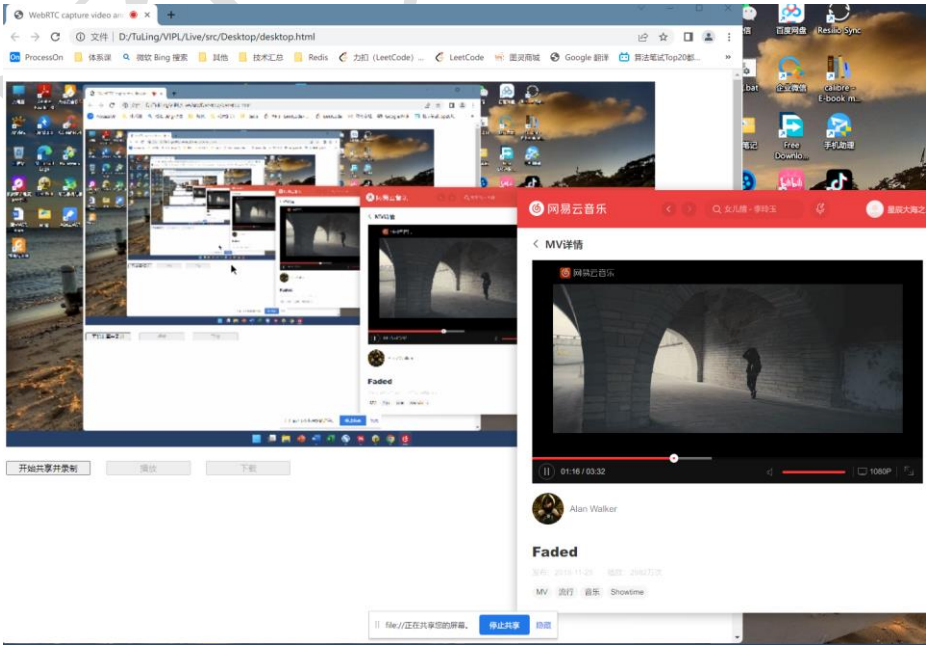
抓取到桌面后，自然还是可以用<video>标签显示，同样的通过 getDisplayMedia 方法获取到本地桌面数据后，然后将该流当作参数传给

MediaRecorder 对象，并实现 ondataavailable 事件，一样可以将音视频流录制下来。

具体效果如下：



从选型上可以看到，分享的时候可以选择屏幕，也可以单独选择窗口或者选择浏览器的标签页分享。



但是录制桌面的 API，目前很多浏览器支持得还不够好，只有 Chrome 浏览器相对比较完善，暂时还不能大面积利用到实际生产环境。

浏览器 1 对 1 音视频实时直播

前面我们用浏览器实现了音视频的捕捉、播放等等功能，都还是自娱自乐，我们的目的还是希望和外界交流。和外界交流自然就离不开网络通信，而 WebRTC 同样可以做到， WebRTC 不但有 AEC、AGC、ANC 等目前世界上最顶级处理音频的算法，同样也有优秀的网络通信部分，包括带宽的评估、平滑处理、各种网络协议的实现。

既然是网络通信，绕不开的就是通信双方各种协商的协议。

协议

网络层的协议常见的有 TCP 和 UDP，对于音视频通信应该选择哪种呢？当然是 UDP。因为 TCP 协议在实现上为了保证传输的可靠，使用了滑动窗口、重发等机制，这其实对音视频通信其实是不利的。

因为音视频通信要求时延小，反而对数据的正确性要求没有那么高，人类的生理已经决定了通信过程中丢失一两帧的数据对用户的观看和收听没有太大影响，而且通过音视频算法可以做到很大程度的弥补。

所以往往音视频通信都是基于 UDP 来实现的，不过不能直接把音视频数据流交给 UDP 传输，而是先给音视频数据加个 RTP 头，然后再交给 UDP 进行传输。

RTP 协议和 RTCP 协议

什么是 RTP 头？想一想为一个视频帧的数据量是非常大的，最少也要几十 K。而以太网的最大传输单元是 1500 字节，所以要传输一个帧需要拆成几十个包。并且这几十个包传到对端后，还要重新组装，这样才能进行解码还原出一幅幅的图像。要完成这样的过程，至少需要以下几个标识。

序号：用于标识传输包的序号，这样就可以知道这个包是第几个分片了。

起始标记：记录分帧的第一个 UDP 包。

结束标记：记录分帧的最后一个 UDP 包。

有了上面这几个标识字段，我们就可以在发送端进行拆包，在接收端将视频帧重新再组装起来了。

RTP 协议就是做这个事的，其中有好几个字段，比如：

sequence number：序号，用于记录包的顺序。

timestamp：时间戳，同一个帧的不同分片的时间戳是相同的。这样就省去了前面所讲的起始标记和结束标记。

PT：Payload Type，数据的负载类型。音频流的 PT 值与视频的 PT 值是不同的，通过它就可以知道这个包存放的是什么类型的数据。

如果从事音视频传输相关的工作，RTP 头中的每个字段的含义都必须全部弄清，这个大家可以自行查阅相关协议文档。

当然，UDP 作为一种不可靠的传输，会发生丢包等情况，WebRTC 对这些问题在底层都有相应的处理策略。除此之外，WebRTC 还使用了 RTCP 协议让通信的两端知道它们自己的网络质量。

RTCP 有两个最重要的报文：RR（Reciever Report）和 SR(Sender Report)。通过这两个报文的交换，各端就知道自己的网络质量到底如何了。比如 SR 报文并不仅是指发送方发了多少数据，它还报告了作为接收方，它接收到的数据的情况。当发送端收到对端的接收报告时，它就可以根据接收报告来评估它与对端之间的网络质量了，随后再根据网络质量做传输策略的调整。当然 RTCP 协议的具体详情也请查阅相关协议文档。

SDP 协议

在我们日常“看片”中有这样的情形：从网上下载了最新大片的资源，结果喜滋滋的双击打开，结果播放器不支持这种视频格式，这就很郁闷了。

在网络上进行音视频通话也是一样的，沟通的双方必须要确保两方对音视频的数据理解是一致的，才能高效的沟通，比如 A 的音频数据采样率是 48000，使用双声道，而 B 只能支持音频采样频率是 32000，使用单声道，很明显进行通信的时候两者的音频通信使用 B 这边的格式就是最好的选择。

WebRTC 使用了 SDP（Session DescriptionProtocal）描述的各端（PC 端、Mac 端、Android 端、iOS 端等）的能力。这里的能力指的是各端所支持的音频编解码器是什么，这些编解码器设定的参数是什么，使用的传输协议是什么，以及包括的音视频媒体是什么等等。

两个客户端 / 浏览器进行 1 对 1 通话时，首先要进行信令交互，而交互的一个重要信息就是 SDP 的交换。

交换 SDP 的目的是为了让对方知道彼此具有哪些能力，然后根据双方各自的能力进行协商，协商出大家认可的音视频编解码器、编解码器相关的参数（如音频通道数，采样率等）、传输协议等信息。

举个例子，A 与 B 进行通讯，它们先各自在 SDP 中记录自己支持的音频参数、视频参数、传输协议等信息，然后再将自己的 SDP 信息通过信令服务器发送给对方。当一方收到对端传来的 SDP 信息后，它会将接收到的 SDP 与自己的 SDP 进行比较，并取出它们之间的交集，这个交集就是它们协商的结果，也就是它们最终使用的音视频参数及传输协议了。

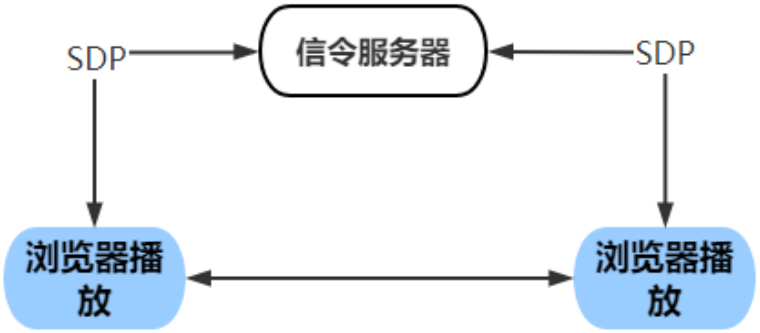
SDP 是文本格式的报文，WebRTC 对标准 SDP 规范做了一些调整，更具体的描述可以参考 https://www.ietf.org/archive/id/draft-nandakumar-rtcweb-sdp-08.txt

通过信令协商

知道了协议，但是还有一个基础的重要问题，通过浏览器进行音视频交流，双方怎么连接呢？在网上看视频听音乐，那是因为我们知道一个确切的网址，登录对应的网站即可下载资源进行欣赏。

对于通过浏览器进行 1 对 1 通话的人怎么办呢？难道双方每次都要查询自己的 IP 地址，然后通过某种手段通知对方？既然能够通知对方，又何必再多此一举再进行音视频通话呢？

所以整个流程中还需要个第三方介入，方便双方进行正式交流之前的协商，这个第三方被称为信令服务。



首先，通信双方将它们各自的媒体信息，如编解码器、媒体流参数、传输协议、IP 地址和端口等，按 SDP 格式整理好。

然后，通信双方通过信令服务器交换 SDP 信息，并待彼此拿到对方的 SDP 信息后，找出它们共同支持的媒体能力。

最后，双方按照协商好的媒体能力建立连接开始音视频通信。

搭建信令服务器完成通信

我们已经知道 WebRTC 信令服务器在 WebRTC 1 对 1 通信中所起的作用。实际上它的功能是不复杂，就是进行信令的交换，但作用却十分关键。在通信双方彼此连接、传输媒体数据之前，它们要通过信令服务器交换一些信息。

前面已经说过，假设 A 与 B 要进行音视频通信，那么 A 要知道 B 已经上线了，同样，B 也要知道 A 在等着与它通信呢。也就是说，只有双方都知道彼此存在，才能由一方向另一方发起音视频通信请求，并最终实现音视频通话。

那在 WebRTC 信令服务器上要实现哪些功能，才能实现上述结果呢？至少要实现下面两个功能:

1、房间管理。即每个用户都要加入到一个具体的房间里，比如两个用户 A 与 B 要进行通话，那么它们必须加入到同一个房间里。

2、信令的交换。即在同一个房间里的用户之间可以相互发送信令。

要实现信令服务器，可以使用 C/C++、Java 等语言一行一行从头开始编写代码，也可以以现有的、成熟的服务器为基础，做二次开发。

目前 Java 下没有成熟的信令服务器的实现，我们可以使用 Netty + Websocket 或 spring-boot-starter-websocket 来实现，大家可以自行查阅相关资料。

而 JS 下的 socket.io 本身是 WebSocket 超集，又内置了房间的概念，所以我们本次使用 Node.js + socket.io 来实现一个信令服务器，同时为了方便使用，我们还将 Node.js 作为 web 服务器，提供 html 页面访问。

在一台 CentOS 服务器上可以直接使用如下命令进行安装：

yum install nodejs

yum install npm

```
[root@VM-16-8-centos ~]# node -v
v16.17.0
[root@VM-16-8-centos ~]# npm -v
8.15.0
```

然后使用 npm 安装：

npm install node-static

npm install log4js

npm install express

npm install server-index

npm install socket.io@2.0.3

因为我们现在不是直接从本地打开网页进行访问而是从远程服务器进行访问，而 WebRTC 里网页要能打开本地媒体设备必须是 https，所以服务器上还需要有自签证书。于是接下来执行

openssl req -newkey rsa:2048 -new -nodes -x509 -days 3650 -keyout key.pem -out cert.pem

然后将获得的 cert.pem 和 key.pem 这两个文件放入 Nodejs 安装目录下的 cert 目录

```
[root@VM-16-8-centos ~]# ls
app.log  cert.pem  key.pem  node_modules  package.json  package-lock.json  server.js
[root@VM-16-8-centos ~]# mkdir cert
[root@VM-16-8-centos ~]# mv cert *.pem
```

```
[root@VM-16-8-centos ~]# ls
app.log  cert  node_modules  package.json  package-lock.json  server.js
[root@VM-16-8-centos ~]# cd cert
[root@VM-16-8-centos cert]# ls
cert.pem  key.pem
```

这样就完成了我们 Nodejs 服务器的搭建。

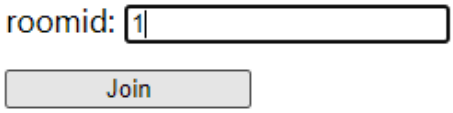
接下来实现我们进行音视频通话的 html 和相关的 css、js 文件，并放入 Nodejs 安装目录下，其中 server.js 是作为服务端处理逻辑，提供了 https 访问能力，日志输出，房间管理等一系列能力，public 中的文件则是客户端（也就是在浏览器）中执行的代码

```
[root@VM-16-8-centos ~]# ls
app.log  cert  coturn  node_modules  package.json  package-lock.json  public  server.js
[root@VM-16-8-centos ~]# cd public
[root@VM-16-8-centos public]# ls
css  index.html  js  room.html
```

然后执行 node server.js ，我们的信令服务器就运行起来了。

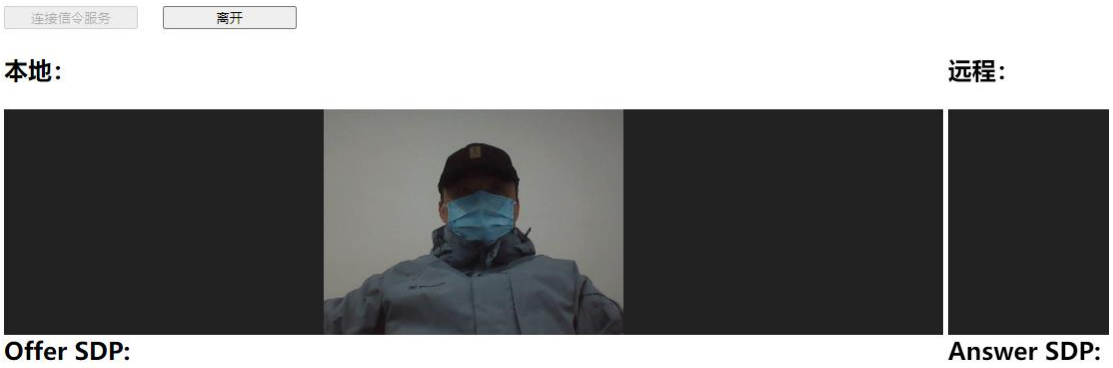
通过浏览器访问：

https://ip 地址/index.html



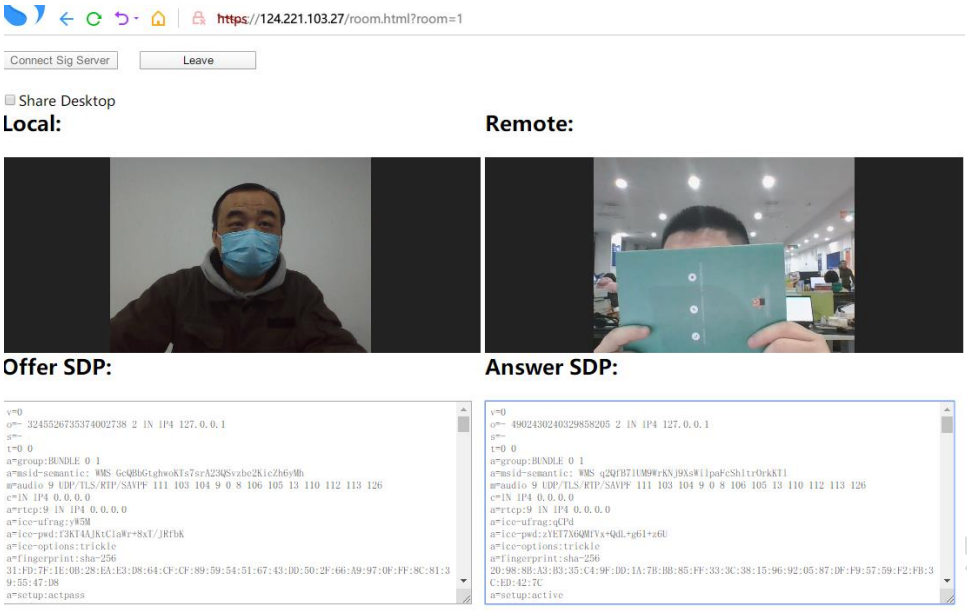
输入房间号 1（此时其实系统内还没有房间，这个操作其实可以视为新建一个编号为 1 的房间），并点击 Join

点击“连接信令服务”后，本地的媒体设备就准备好了。



因为本机的媒体设备已经被占据了，现在需要另外找一台 PC（**注意这台 PC 要求和本机必须在同一子网内**，为什么要在同一子网内，后面会详细解释）。

同样访问：https://ip 地址/index.html，并输入房间号 1，加入房间，也需要连接信令服务器，成功后双方就可以进行 1 对 1 音视频通话了。



用 RTCPeerConnection 协商

我们前面的 1V1 音视频通信的代码量比较大，而且比较复杂，需要对 JavaScript 等等有很好的的掌握，我们不做具体讲解。

但是目前还遗留了两个问题：

1、为什么两台 PC 要在同一子网内呢？

2、页面上的 Offer SDP、Answer SDP 分别是什么意思呢？

这个需要我们对 WebRTC 的底层有一定程度的理解。

WebRTC 中负责在端与端之间建立的连接的是 RTCPeerConnection 类，该类是整个 WebRTC 库中最关键的一个类。

在通讯双方都创建好 RTCPeerConnection 对象后，它们就可以开始进行媒体协商了。不过在进行媒体协商之前，有两个重要的概念，即 Offer 与 Answer 要弄清楚。

Offer 与 Answer 是什么呢？对于 1 对 1 通信的双方来说，我们称首先发送媒体协商消息的一方为呼叫方，而另一方则为被呼叫方。

Offer，在双方通讯时，呼叫方发送的 SDP 消息称为 Offer。

Answer，在双方通讯时，被呼叫方发送的 SDP 消息称为 Answer。

首先，呼叫方创建 Offer 类型的 SDP 消息。创建完成后，将该 Offer 保存到本地 Local 域，然后通过信令将 Offer 发送给被呼叫方。

被呼叫方收到 Offer 类型的 SDP 消息后，将 Offer 保存到它的 Remote 域。作为应答，被呼叫方创建 Answer 类型的 SDP 消息，然后将 Answer 类型的 SDP 消息保存到本地的 Local 域。最后，被呼叫方将 Answer 消息通过信令发送给呼叫方。至此，被呼叫方的工作就完部完成了。

接下来是呼叫方的收尾工作，呼叫方收到 Answer 类型的消息后，将 Answer 保存到它的 Remote 域。

至此，整个媒体协商过程处理完毕。紧接着在 WebRTC 底层会收集 Candidate（候选者），并进行连通性检测，最终在通话双方之间建立起一条链路来。

连通性检测

WebRTC 之间建立连接的过程是非常复杂的。之所以复杂，主要的原因在于它既要考虑传输的高效性，又要保证端与端之间的连通率。

换句话说，当同时存在多个有效连接时，它首先选择传输质量最好的线路，如能用内网连通就不用公网。另外，如果尝试了很多线路都连通不了，那么它还会使用服务端中继的方式让双方连通。

现在我们假设通信的双方为 A 和 B。

.场景一：双方处于同一网段内

A 与 B 进行通信，假设它们现在处于同一个办公区的同一个网段内。在这种情况下，A 与 B 有两种连通路径：

一种是双方通过内网直接进行连接；

另一种是通过公网，也就是通过公司的网关，从公网绕一圈后再进入公司实现双方的通信。

很显然第一种连接路径是最好的。A 与 B 在内网连接就好了，谁会舍近求远呢?

但现实却并非如此简单，要想让 A 与 B 直接在内网连接，首先要解决的问题是：A 与 B 如何才能知道它们是在同一个网段内呢?

这个问题还真不好回答，也正是由于这个问题不太好解决，所以，现在有很多通信类产品在双方通信时，无论是否在同一个内网，它们都统一走了公网。不过 WebRTC 不是这样处理的，后面我们可以看一下它是如何解决这个问题的。

场景二：双方处于不同点

A 与 B 进行通信，它们分别在不同的地点，比如一个在北京，一个在上海，此时 A 与 B 通信必须走公网。但走公网也有两条路径：

一是通过 P2P 的方式双方直接建立连接；

二是通过中继服务器进行中转，我们假设 C 为中继服务器，即 A 与 B 都先与 C 建立连接，当 A 向 B 发消息时，A 先将数据发给 C，然后 C 再转发给 B;同理，B 向 A 发消息时，B 先将消息发给 C，然后 C 再转给 A。

对于这两条路径你该如何选择呢？很明显 P2P 的方式更好，但 A 和 B 未必能够直连，所以 WebRTC 优先使用 P2P 方式；如果 P2P 方式不通，才会使用中继的方式。

既然建立连接有这么多路径，所以 WebRTC 中存在 ICE Candidate（ICE 候选者）的概念。一般由本地 IP 地址、本地端口号、候选者类型（包括 host、srflx 和 relay）、优先级、传输协议、访问服务的用户名等字段组成。

其中，候选者类型中

host 类型，即本机内网的 IP 和端口；

srflx 类型, 即本机 NAT 映射后的外网的 IP 和端口；

relay 类型，即中继服务器的 IP 和端口。

当 WebRTC 通信双方彼此要进行连接时，每一端都会提供许多候选者。

WebRTC 会按照上面描述的格式对候选者进行排序，然后按优先级从高到低的顺序进行连通性测试，当连通性测试成功后，通信的双方就建立起了连接。

在众多候选者中，host 类型的候选者优先级是最高的。在 WebRTC 中，首先对 host 类型的候选者进行连通性检测，如果它们之间可以互通，则直接建立连接。其实，host 类型之间的连通性检测就是内网之间的连通性检测。

同样的道理，如果 host 类型候选者之间无法建立连接，那么 WebRTC 则会尝试次优先级的候选者，即 srflx 类型的候选者。也就是尝试让通信双方直接通过 P2P 进行连接，如果连接成功就使用 P2P 传输数据；如果失败，就最后尝试使用 relay 方式建立连接。

由此，我们知道了为什么我们前面实战的时候声明要两台 PC 要在同一子网内才能进行一对一音视频通话了，同一子网内的机器连通性一般都是没什么问题的，而跨子网的两台 PC 在网络管理员没有配置的情况下一般是无法相互访问的，这个时候往往需要外部公网中转。如何进入外部公网中转？

我们知道，如果主机没有公网地址，是无论如何都无法访问公网上的资源的。例如你要通过百度搜索一些信息，如果你的主机没有公网地址的话，百度搜索到的结果怎么传给你呢？

而一般情况下，主机都只有内网 IP 和端口，那它是如何访问外网资源的呢？实际上，在内网的网关上都有 **NAT（Net Address Transport）** 功能，NAT 的作用就是进行内外网的地址转换。这样当你要访问公网上的资源时，NAT 首先会将该主机的内网地址转换成外网地址，然后才会将请求发送给要访问的服务器；服务器处理好后将结果返回给主机的公网地址和端口，再通过 NAT 最终中转给内网的主机。

知道了上面的原理，你要想让内网主机获得它的外网 IP 地址也就好办了，只需要在公网上架设一台服务器，并向这台服务器发个请求说：“Hi！伙计，你看我是谁？”对方回：“你不是那 xxxx 吗？”这样你就可以知道自己的公网 IP 了，是不是很简单？

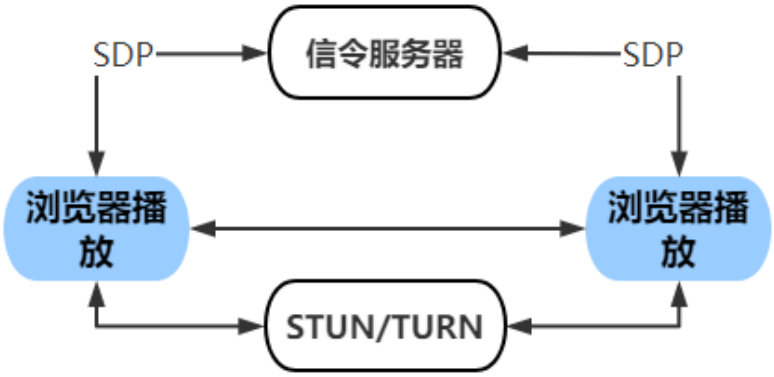
实际上，上面的描述已经被定义成了一套规范，即 RFC5389，也就是 STUN 协议，我们只要遵守这个协议就可以拿到自己的公网 IP。

Google 已经开源了一个 STUN 服务器的实现 coturn，访问地址 https://github.com/coturn/coturn。

但是通信的双方并不是拿到了 NAT 映射后的外网的 IP 和端口就一定能建立 P2P 的直接通信的，因为 NAT 穿透有好几种机制，比如完全锥形、受限制锥形等等，并不是每种机制都允许通信双方能够直接建立连接的，这个时候就需要通过中继服务器进行中转了。

那 relay 类型即中继服务器我们从哪儿获得呢？其实 coturn 也提供了 TURN 服务，对应的协议规范是 RFC5766 。也就是说 coturn 有两个作用，一是提供 STUN 服务，客户端可以通过 STUN 服务获取自己的外网地址；二是提供数据中继服务。

这样，当通信的双方不在同一子网，又无法通过 P2P 进行数据传输时，为了保证数据的连通性，就可以使用中继的方式让通信双方的数据可以互通，于是 1 对 1 音视频通话的流程图就演变为下面这个样子：



多人音视频实时通讯架构

1 对于 1 当然不是我们的终点，我们还希望知道多人音视频实时通讯是如何架构的？WebRTC 可以支持多人音视频实时通讯吗？要了解这个我们就需要知道跨越 20 多年的 RTC（实时通信技术）架构演化史。

RTC 架构演化史

RTC 从 1996 年开始到如今已经发展成为一个非常复杂的技术领域，其包含了网络传输、全局调度、媒体处理算法、媒体编解码、信令协议、输入输出设备、Web、操作系统等相关的技术，至今为止发展了 20 多年。这期间伴随互联网发展经历了多次技术迭代，从网络通信架构演化过程来看可把它分为三大阶段（其实最后的阶段还可再细分），每个阶段 RTC 从终端技术到通信架构都有大的技术变化。

第 1 代：MCU 阶段（1996~2004 年）

第 2 代：P2P 阶段（2001~2007 年）

第 3 代：SFU 阶段（2003~至今，又可细分为 2003~2012 单 SFU 阶段、2010~2017 年云 SFU 阶段、2016 至今全球实时云通信阶段）

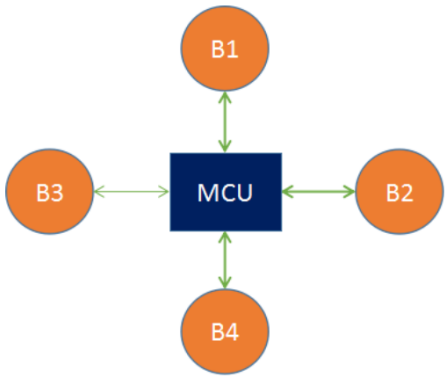
MCU 阶段

上个世纪 90 年代中期，随着 WWW 互联网的崛起，大量的终端用户通过电话线拨号涌入互联网，当时上网的终端网速只有 56kbps（俗称 56K 猫），看网页、下载图片有时候需要数十秒钟。网速慢且流量昂贵，但企业和个人都有通过 PC 联网的强烈需求。

在这种大背景下，ITU（国际电信联盟）基于传统 PSTN 架构在 1996 年率先推出 H.323/RTP 的 IP 网络多媒体标准，并在 H.323 协议中提出了 MCU 和网关的概念。

MCU 主要的处理逻辑是：接收每个共享端的音视频流，经过解码、与其他解码后的音视频进行混流、重新编码，之后再将混好的音视频流发送给房间里的所有人。

MCU 技术在视频会议领域出现得非常早，目前技术也非常成熟，主要用在硬件视频会议领域。MCU 方案的模型是一个星形结构。



MCU 的优势有哪些呢？大致可总结为如下几点：技术非常成熟，在硬件视频会议中应用非常广泛。作为音视频网关，通过解码、再编码可以屏蔽不同编解码设备的差异化，满足更多客户的集成需求，提升用户体验和产品竞争力。将多路视频混合成一路，所有参与人看到的是相同的画面，客户体验非常好。

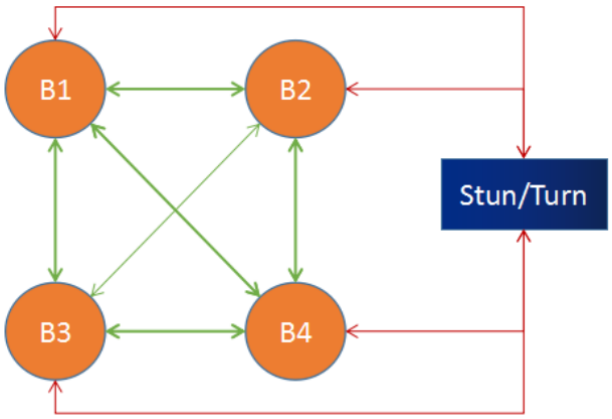
同样，MCU 也有一些不足，主要表现为：重新解码、编码、混流，需要大量的运算，对 CPU 资源的消耗很大。重新解码、编码、混流还会带来延迟。由于机器资源耗费很大，所以 MCU 所提供的容量有限，一般十几路视频就是上限了。

P2P 阶段

1999 年，ITU 批准了 ADSL 技术成为宽带标准，下载速度可达 8Mbps，上行可以到 1Mbps，ADSL 解决了终端带宽瓶颈问题，一时间 IM 社交（OICQ/MSN）和 P2P 下载服务（KaZaA/BT/eMule/迅雷）成为 PC 互联网的主宰。同时摩尔定律继续发挥效应，终端 PC 的计算能力进一步加强，已经出现双核 CPU。终端设备的升级使得个人用户的互联网通信成为主流并形成了新的实时通信需求，同时也给 RTC 通信带来新的变革和机会。

正是终端计算能力和带宽能力大幅提升，RTC 技术形成了黄金发展时期，出现了众多影响至今的技术和模型。比如 SIP/SDP 协议、P2P 实时通信（在 P2P 系统中发展出来了很多新的网络传输技术，例如：STUN/TURN 穿越技术、ICE 技术、RUDP 技术、数据切片组帧技术等等）、瑞典 GIPS 公司开发出了独特的 3A 声音算法算法（AEC/AGC/ANS）、网络抗抖算法（NetEQ）、高质量语音编码器（iSAC），GIPS 被 Google 收购，它与 GTalk 的 libjingle 合并成为早期 WebRTC 的一部分。

由于 PC 终端能力增强，RTC 的架构从 MCU 架构演化成了基于每个流的 P2P Mesh 架构，这种架构脱离中心服务的束缚，转变成去中心化的服务模型。最先把这个模型用到极致的是第一个全球实时通信网络——Skype。



当某个浏览器想要共享它的音视频流时，它会将共享的媒体流分别发送给其他 3 个浏览器，这样就实现了多人通信。这种结构的优势有如下:

后台服务轻量，架构扩展性强，开发编程简单。充分利用了客户端的带宽资源。节省了服务器资源和带宽。

当然，有优势自然也有不足之处，主要表现如下：共享端共享媒体流的时候，需要给每一个参与人都转发一份媒体流，这样对上行带宽的占用很大。参与人越多，占用的带宽就越大。除此之外，对 CPU、Memory 等资源也是极大的考验。一般来说，客户端的机器资源、带宽资源往往是有限的，资源占用和参与人数是线性相关的。这样导致多人通信的规模非常有限，通过实践来看，这种方案在超过 4 个人时，就会有非常大的问题。服务质量完全依赖于各个终端的网络环境和稳定性，连通性差，各端能力差异会引来通信差异。对于跨国跨运营商的通信无法保证通话 QoS，去中心化后对于敏感通话信息无法进行中心管理监控，有政治安全风险。

SFU 阶段

单 SFU 阶段

网络 MMO RPG 游戏的兴起使得游戏社区开始萌芽，游戏领域的实时沟通成了一个强需求，并进而诞生出直播聊天室，后续从中又演化出 9158、六间房等产品。2011 年底，YY 在其大频道加入单视频直播模式，直播方式在这个阶段成了主要流量。

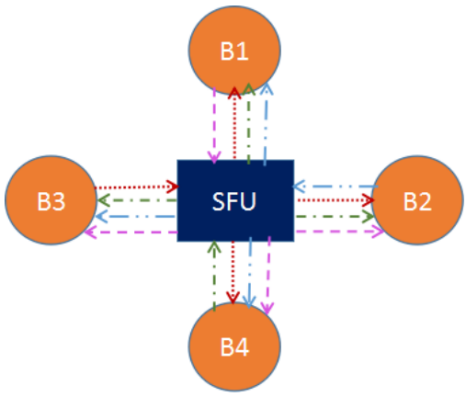
社区模式使得 RTC 重新回到了传输的 Client/Server 模式，Client 负责个人媒体数据收发和录制播放，Server 负责控制和传输转发，前后端侧重的技术点不一样，给社区场景带来了后面的技术转变。

为了满足频道多人互动需求，YY 在 2008 年左右开发出万人语音频道，很快 iSpeaker 和其他的游戏语音厂商也开始跟进。万人语音频道还是有很大的技术挑战，当时大部分厂商采用的是单物理机的 SFU 模式，相当于单个机器并发 UDP 包事件在 20 万/秒左右。在 2012 年的 YY 利用自己的分布式 UDP Overlay 架构成功支持了单频道 120 万人线上观看直播，这是 RTC 技术上的里程碑。。

2010 年，Google 将 WebRTC 库纳入 Chrome 体系并开源 2012 年它获得各大浏览器厂商支持并纳入 W3C 标准，自此基本形成能支持 Web 形式的实时音

视频通信。WebRTC 的开源是 RTC 技术领域的里程碑事件，大大降低了 RTC 开发的门槛，催生了后来移动互联网 RTC 应用的大时代。

这个阶段 RTC 出现了大规模直播表演类的应用，使得 SFU 架构迅速成型并大规模应用，SFU 架构很好地分离了客户端和服务端功能，SFU 专注于控制与传输转发。SFU 架构简洁，适合大规模和超大规模实时音视频互动场景。



SFU 在结构上显得简单很多，只是接收流然后转发给其他人。然而，这个简单结构也给音视频传输带来了很多便利。比如，SFU 可以根据终端下行网络状况做一些流控，可以根据当前带宽情况、网络延时情况，选择性地丢弃一些媒体数据，保证通信的连续性。

SFU 的优势有哪些呢?可总结为如下:

由于是数据包直接转发，不需要编码、解码，对 CPU 资源消耗很小。直接转发也极大地降低了延迟，提高了实时性。

带来了很大的灵活性，能够更好地适应不同的网络状况和终端类型。

同样，SFU 有优势，也有不足，主要表现为:

由于是数据包直接转发，参与人观看多路视频的时候可能会出现不同步;相同的视频流，不同的参与人看到的画面也可能不一致。

参与人同时观看多路视频，在多路视频窗口显示、渲染等会带来很多麻烦，尤其对多人实时通信进行录制，多路流也会带来很多回放的困难。总之，整体在通用性、一致性方面比较差。

云 SFU 阶段

智能手机的诞生让互联网从 PC 时代迈向了移动时代，这个巨大的改变使 RTC 应用场景发生了转变。这个阶段有三个特点：

1)终端计算能力削弱，网络接入向不稳定的无线（WiFi/3G/4G）接入迁移。

2)后端计算能力增强，网络消除了分区连通问题。

3)联网用户激增，RTC 原来的社交、游戏、娱乐直播等领域进一步渗透，RTC 扩展到教育、电商和金融。

正因如此，RTC 在 2014 年出现了对外提供 RTC 云计算 PaaS 服务商网易云信和声网，RTC PaaS 是部署在云 IaaS 上的 SFU，直接利用了 IaaS 基础设施能力，简化了 SFU 架构，这个架构简称为云 SFU。

在线教育开始萌芽，RTC 除了音视频通信以外还产生了富媒体的实时通信需求：课件同步与书写同步。富媒体同步需求让 RTC 产生了实时可靠传输数据能力，在此基础上演化出来实时互动白板。

娱乐直播领域视频分辨率整体提高，甚至出现 720P/1080P 的实时视频直播，此时 CDN 接管了音视频分发，RTC 主要应用在直播连麦和推流，在这个场景当中 RTC 产生了诸多新技术。

企业办公移动化，视频会议产品开始移动化，标志性产品 Zoom。

视频的分辨率和码率爆炸性的增加，而终端接入从原来的有线接入转变为无线接入方式，网络质量受周围环境的影响加大，而且时刻处于高度变化当中，那么在一个高度变化的网络上提供稳定的实时通信的需求就变得异常强烈。在这个阶段，各大 RTC PaaS 使出浑身解数，各展绝技，不断提高技术指标和效果，toC 产品的技术部门紧跟其后。

比如网络通讯和音视频算法方面的：拥塞控制、FEC、ARQ 与 NACK、Simulcast 与 SVC、语音带内冗余编码技术、QUIC 与 RUDP、媒体处理 AI 化。

还有服务能力方面的 SFU Fullmesh（单 SFU 架构有用户接入和可用性方面的问题，云 SFU 用 BGP 接入解决用户接入问题。将 SFU 做了平行扩展，同一个频道用户可以接入到多个对等的 SFU 进行实时音视频通信，很好地消除了服务异常造成不可用的缺陷，这种架构还能跨 IDC 部署，实现 7x24 小时的云服务）、旁路服务（由于直播、教育等业务都是需要审核监控和录制相关的需求，RTC 后端除了 SFU Fullmesh 变化外，还引入了旁路网关单元，旁路网关功能繁多，包括：实时录制、AI 内容审核、合流 CDN 推流、其他协议接入（小程序网关/PSTN））等等。

全球实时云通信阶段

教育大规模线上化和中国音视频产品出海预示着全球实时通信阶段的到来，2020 年开始全球疫情加速了 RTC 音视频通信在各行业的落地，一方面在线课堂、在线办公、企业协同、直播带货等出现了大量 RTC 应用场景，另外一方面 RTC 流量成本居高不下，场景流量暴增和成本居高不下之间的矛盾日益严峻，催生了新一代 RTC 架构迭代。这个阶段主要是两个问题：

1.如何将 RTC 流量成本降下来？

2.如何保证下沉用户和跨国长距离通信的质量？

为了解决上面的问题，声网 2016 年首先研发出基于 SDN 架构的 SD-RTN 技术来构建全球实时通信网络，利用廉价的边缘节点和独特的智能路由加速技术让全球端到端延迟控制在 400ms 以内。

RTN 的出现直接导致了终端到 Server 、Server 到 Server 之间通信的解耦，RTN 专注于 Server 之间的加速，终端与 Server 之间慢慢演变成了类 QUIC 的半可靠 RUDP 协议通信。正因为 RTN 架构能有效解决以上两方面的问题，又符合未来边缘计算趋势，所以各大 RTC 厂商都纷纷跟进研发自己的 RTN 组网技术。

其中关键技术包括边缘计算容器、智能调度、智能路由与多链路备份、虚拟组网技术、终端接入协议类 QUIC 统一等等。

目前 RTN 还没有统一的标准和做法，各家厂商都是根据自身的业务特点构建自己的 RTN 系统。

基于 WebRTC 的多对多实时通信

基于 WebRTC 的多对多实时通信能不能实现，当然也能实现，相关开源项目也有很多，具体的组网方案也没有脱离我们前面所说的 RTC 架构的演化史，基本上就是三种方案。

Mesh 方案，即多个终端之间两两进行连接，形成一个网状结构。比如 A、B、C 三个终端进行多对多通信，当 A 想要共享媒体（比如音频、视频）时，它需要分别向 B 和 C 发　送数据。同样的道理，B 想要共享媒体，就需要分别向 A、C 发送数据，依次类推。这种方案对各终端的带宽要求比较高。

MCU（Multipoint Conferencing Unit）方案，该方案由一个服务器和多个终端组成一个星形结构。各终端将自己要共享的音视频流发送给服务器，服务器端会将在同一个房间中的所有终端的音视频流进行混合，最终生成一个混合后的音视频流再发给各个终端，这样各终端就可以看到 / 听到其他终端的音视频了。实际上服务器端就是一个音视频混合器，这种方案服务器的压力会非常大。

SFU（Selective Forwarding Unit）方案，该方案也是由一个服务器和多个终端组成，但与 MCU 不同的是，SFU 不对音视频进行混流，收到某个终端共享的音视频流后，就直接将该音视频流转发给房间内的其他终端。它实际上就是一个音视频路由转发器。

自然目前基于 WebRTC 多方通信媒体服务器都是 SFU 架构，常见支持 WebRTC 的 SFU 流媒体服务器的开源项目有 Licode、Janus-gateway、MediaSoup、Medooze 等。Meooze、Mediasoup、Licode 这三个流媒体服务器的媒体通信部分都是由 C++ 实现的，而控制逻辑是通过 Node.js 实现，而 Janusgateway 是完全通过 C 语言实现的。

而国内目前做的比较好的，则是 SRS(Simple Realtime Server)，官方网站 http://www.ossrs.net/lts/zh-cn/docs/v4/doc/introduction，也是开源的，同样也支持 WebRTC，更支持我们后面将要学习到的 RTMP/HLS/HTTP-FLV 等一系列协议 。不过我们接下来的课程则不会使用 WebRTC ，因为目前的直播系统中它还不是主流技术，而是接下来我们要说的的目前直播中常见的协议了。

直播中的协议与格式

常见的流媒体直播协议有 RTSP、RTMP 和 HLS（流媒体协议，英文学名 Streaming Protocol，是一种用于通过 Web 传递多媒体的协议）。

从技术角度来讲，如果主要考虑传输的实时性，因此一般使用 UDP 作为底层传输协议；如果更多关注的是画面的质量等问题，一般采用 TCP 作为传输协议。

RTSP（Real Time Streaming Protocol，可以理解为是个协议族，包含了 RTSP、RTP、RTCP），支持 UDP/TCP，时延在几百毫秒内，是目前直播协议中最低的一种，但是技术实现复杂，通常应用于安防视频监控、IPTV 等场景，所以这个协议不会做详细的讲解。

基于 TCP 的的传输协议以 RTMP 和 HLS 为主。其中，RTMP 是由 Adobe 公司开发的，虽然不是国际标准，但也算是工业标准，在 PC 占有很大的市场;而

HLS 是由苹果公司开发的，主要用在它的 iOS 平台，不过 Android 3 以后的平台也是默认支持 HLS 协议的。

实际的直播系统大都支持 RTMP 和 HLS，较大的公司也会支持 RTSP，比如抖音就兼容 RTSP 的直播。

一般的商业级直播架构由直播客户端、信令服务器和 CDN 网络这三部分组成。直播客户端主要包括音视频数据的采集、编码、推流、拉流、解码与播放这几个功能。

主播和观众的需求是不一样的，直播客户端也就分为两类，一类是主播使用的客户端，包括音视频数据采集、编码和推流功能；另一类是观众使用的客户端，包括拉流、解码与渲染（播放）功能。

对于主播客户端来说，它可以从 PC 或移动端设备的摄像头、麦克风采集数据，然后对采集到的音视频数据进行编码，最后将编码后的音视频数据按 RTMP 协议推送给 CDN 源节点（RTMP 接入服务器）。

对于观众客户端来说，它首先从直播管理系统中获取到房间的流媒体地址，然后通过 RTMP 协议从边缘节点拉取音视频数据，并对获取到的音视频数据进行解码，最后进行视频的渲染与音频的播放。

通过上面的描述，看上去主播端与观众端的开发工作量差不多。但实际上，观众端的开发工作量要小得多。其原因是，观众端要实现的就是一个播放器功能，而目前开源界有两个比较有名而又成熟的开源项目 Ijkplayer 和 VLC，所以只要将这两个开源项目中的一个集成到你自己的项目中，基本上就完成了观众端的所有开发工作。

信令服务器，主要用于接收信令，并根据信令处理一些和业务相关的逻辑，如创建房间、加入房间、离开房间、送礼物、文字聊天等。

CDN 网络，主要用于媒体数据的分发。它内部的实现非常复杂，我们姑且先把它当作是一个黑盒子，然后只需要知道传给它的媒体数据可以很快传送给全世界每一个角落。换句话说，你在全世界各地，只要接入了 CDN 网络，你都可以快速看到你想看到的“节目”了。

介绍完直播客户端、信令服务器和 CDN 网络之后，我们再来来看看主播到底是如何将自己的音视频媒体流进行分享的。

主播客户端在分享自己的音视频媒体流之前，首先要向信令服务器发送“创建房间”的信令；信令服务器收到该信令后，给主播客户端返回一个推流地址（CDN 网络源站地址）；此时，主播客户端就可以通过音视频设备进行音视频数据的采集和编码，生成 RTMP 消息，最终将媒体流推送给 CDN 网络。

无论主播端使用的是 PC 机还是移动端，其推流步骤都是一样的。

当观众端想看某个房间里的节目时，首先也要向信令服务器发消息，不过发送的不是“创建房间” 消息了，而是 “加入房间”；服务端收到该信令后，会根据用户所在地区，分配一个与它最接近的“CDN 边缘节点”；观众客户端收到该地址后，就可以从该地址拉取媒体流了。

一般来说，直播系统中，一般推流都使用的 RTMP 协议，而拉流可以选择使用 RTMP 协议或者 HLS 协议。

RTMP 介绍

RTMP，全称 Real Time Messaging Protocol，即实时消息协议。但它实际上并不能做到真正的实时，一般情况下可以控制在 3 秒左右的延迟，底层是基于 TCP 协议的。

RTMP 的传输格式为 RTMP Chunk Format，媒体流数据的传输和 RTMP 控制消息的传输都是基于此格式的。

优势

RTMP 协议在苹果公司宣布其产品不支持 RTMP 协议，且推出 HLS 技术来替代 RTMP 协议的“打压”下，已停止更新。但协议停止更新后，这么多年仍然屹立不倒，说明该协议肯定有它独特的优势。那有哪些呢？

RTMP 协议底层依赖于 TCP 协议，不会出现丢包、乱序等问题，因此音视频业务质量有很好的保障。

使用简单，技术成熟。有现成的 RTMP 协议库实现，如 FFmpeg 项目中的 librtmp 库，用户使用起来非常方便。而且 RTMP 协议在直播领域应用多年，技术已经相当成熟。

市场占有率高。在日常的工作或生活中，我们或多或少都会用到 RTMP 协议。如常用的 FLV 文件，实际上就是在 RTMP 消息数据的最前面加了 FLV 文件头。

相较于 HLS 协议，它的实时性要高很多。

2. 劣势

RTMP 有优势，也有劣势。最为关键的是 Adobe 已经停止对 RTMP 协议的更新。

可以看出 RTMP 协议已经失去了未来，只是由于目前没有更好的协议可以直接代替它，所以它还能“苟延残喘”存活几年。在没有其他变数的情况下最终一定会消亡。

HLS 介绍

HLS，全称 HTTP Live Streaming，是苹果公司实现的基于 HTTP 的流媒体传输协议。它可以支持流媒体的直播和点播，主要应用在 iOS 系统和 HTML5 网页播放器中。

HLS 的基本原理非常简单，它是将多媒体文件或直接流进行切片，形成一堆的 ts 文件和 m3u8 索引文件并保存到磁盘，在我们后面的实战中将会看到。

当播放器获取 HLS 流时，它首先根据时间戳，通过 HTTP 服务，从 m3u8 索引文件获取最新的 ts 视频文件切片地址，然后再通过 HTTP 协议将它们下载并缓存起来。当播放器播放 HLS 流时，播放线程会从缓冲区中读出数据并进行播放。

通过上面的描述我们可以知道，HLS 协议的本质就是通过 HTTP 下载文件，然后将下载的切片缓存起来。由于切片文件都非常小，所以可以实现边下载边

播的效果。HLS 规范规定，播放器至少下载一个 ts 切片才能播放，所以 HLS 理论上至少会有一个切片的延迟。

1. 优势

HLS 是为了解决 RTMP 协议中存在的一些问题而设计的，所以，它自然有自己的优势。主要体现在以下几方面：

HLS 使用的是 HTTP 协议传输数据，在一些有访问限制的网络环境下，比如企业网防火墙，是没法访问外网的，因为企业内部一般只允许 80/443 端口可以访问外网， HLS 协议天然就解决了这个问题。

HLS 协议本身实现了码率自适应，不同带宽的设备可以自动切换到最适合自己码率的视频进行播放。

基于 HTTP 协议，就意味着浏览器天然支持 HLS 协议。

2. 不足

HLS 最主要的问题就是实时性差。由于 HLS 往往采用 10s 的切片，所以最小也要有 10s 的延迟，一般是 20～30s 的延迟，有时甚至更差。

HLS 之所以能达到 20～30s 的延迟，主要是由于 HLS 的实现机制造成的。HLS 使用的是 HTTP 短连接，且 HTTP 是基于 TCP 的，所以这就意味着 HLS 需要不断地与服务器建立连接。TCP 每次建立连接时都要进行三次握手，而断开连接时，也要进行四次挥手，基于以上这些复杂的原因，就造成了 HLS 延迟比较久的局面。

HTTP-FLV

FLV (Flash Video) 是 Adobe 公司推出的另一种视频格式，是一种在网络上传输的流媒体数据存储容器格式。其格式相对简单轻量，不需要很大的媒体头部信息。整个 FLV 由 The FLV Header, The FLV Body 以及其它 Tag 组成。因此加载速度极快。采用 FLV 格式封装的文件后缀为 .flv。其实 FLV 文件与 RTMP 之间是“近亲”关系，可以理解为 RTMP 数据之上加个 FLV 头就成了 FLV 文件。

FLV 文件是一个流式的文件格式，这个结构有一个天然的好处，可以将音视频数据随时添加到 FLV 文件的末尾，而不会破坏文件的整体结构。

在众多的媒体文件格式中，只有 FLV 具有这样的特点。像 MP4、MOV 等媒体文件格式都是结构化的，也就是说音频数据与视频数据是单独存放的。当服务端接收到音视频数据后，如果不通过 MP4 的文件头，你根本就找不到音频或视频数据存放的位置。由于 FLV 是流式的文件格式，所以它特别适合在音视频录制中使用。在实际生产过程中，很多的时候会生成好的 FLV 直接推送到服务器，服务端将 FLV 文件转成 HLS 切片后，供用户观看，自然 RTMP 与 FLV 相互转换时也很容易。

而我们所说的 HTTP-FLV 即将流媒体数据封装成 FLV 格式，然后通过 HTTP 协议传输给客户端。

HTTP-FLV 依靠 MIME 的特性，根据协议中的 Content-Type 来选择相应的程序去处理相应的内容，使得流媒体可以通过 HTTP 传输。相较于 RTMP 协议，HTTP-FLV 能够好的穿透防火墙，它是基于 HTTP/80 传输，有效避免被防火墙拦截。除此之外，它可以通过 HTTP 302 跳转灵活调度/负载均衡，支持使用 HTTPS 加密传输，也能够兼容支持 Android，iOS 的移动端。

对于网页播放端，本来还是需要 Flash 才能播放，但「flv.js」的出现又弥补了这个缺陷。flv.js 是由 bilibili 公司开源的项目。它可以解析 FLV 文件，从中取出音视频数据并转成 BMFF 片段（一种 MP4 格式），然后交给 HTML5 的<video>标签进行播放。通过这种方式，使得浏览器在不借助 Flash 的情况下也可以播放 FLV 文件了。

HTTP-FLV 的缺点在于由于它的传输特性，会让流媒体资源缓存在本地客户端，在保密性方面不够好，而且网络流量较大。

如何选择直播协议

流媒体接入，也就是推流，应该使用 RTMP 协议。

流媒体系统内部分发使用 RTMP 协议。因为内网系统网络状况好，使用 RTMP 更能发挥它的高效本领。

在 PC 上，尽量使用 RTMP 协议，因为 PC 基本都安装了 Flash 播放器，直播效果要好很多。

移动端的网页播放器最好使用 HLS 协议。

iOS 要使用 HLS 协议，因为不支持 RTMP 协议。

点播系统最好使用 HLS 协议。因为点播没有实时互动需求，延迟大一些是可以接受的，并且可以在浏览器上直接观看。

而综合 RTMP、HLS、HTTP-FLV 和 WebRTC 四者来看，平台适配性上，且实现效果差不多的情况下，RTMP、HLS 要比 HTTP-FLV 和 WebRTC 更优秀。其次从市场环境上来说，经过了很多年的发展和磨合，很多的 CDN 大厂已经非常完美的支持 RTMP 和 HLS 了，CDN 不会对稳定盈利的系统轻易做出变化。同样，越来越多的公司来用 RTMP 和 HLS，那么就造成 CDN 与 RTMP、CDN 与 HLS 之间的优化和兼容更强了。这是一个循环过程，一般 CDN 公司不会轻易去打破。所以目前直播还是以 RTMP 和 HLS 为主。

不过 RTMP 和 HLS 都是掌握在大企业手中的协议，而 WebRTC 已被纳入 W3C 标准；无需安装插件，支持的浏览器越来越多，它的未来也是可以期待的，完全可能成为未来的主流，但是目前来说还有待时日，这也是为什么我们接下的课程不继续使用 WebRTC 的原因。

所以接下来我们以 RTMP 和 HLS 为主来进行点播和直播实战。

点播和直播实战

要实现直播和点播当然离不开服务器支持（自然也包括了 WEB 访问），我们使用了 Nginx 作为我们的直播和点播服务器。

搭建 Nginx 点播直播服务器

当然我们也知道，Nginx 本身是不支持视频的，所以需要 Nginx 增加相应的 RTMP 模块进行支持。

这个时候对 Nginx 的安装，我们需要选择的是用源码编译方式进行安装，因为这种方式可以自定义安装指定的模块以及最新版本。

安装 Nginx 我们需要先安装依赖，然后下载相关的源码进行编译，最后执行编译安装。

安装依赖

对依赖的执行如下命令即可：

安装命令 install -y libpcre3 libpcre3-dev libssl-dev zlib1g-dev gcc wget unzip vim make curl

注意：不同的 Linux 操作系统安装依赖的命令不一样，有的是 apt-get ，有的是 yum，请根据自己的操作系统自行选择。

下载源码

有两个源码下载，一个是 Nginx 本身，另一个是 rtmp 模块。

我们首先将需要的模块下载下来，rtmp 模块有 nginx-rtmp-module，但是我们这里使用 nginx-http-flv-module 来替代。因为后者是基于前者开发的，前者拥有的功能后者都有，后者是国内的开发开发，有中文文档，所以就采用它了，首先将它下载下来并解压，执行的命令如下所示

wget https://github.com/winshining/nginx-http-flv-module/archive/master.zip

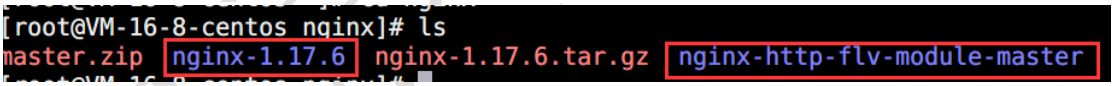
unzip master.zip

接着下载 nginx.本身的源码，下载 Nginx 源码并解压的命令如下所示

wget http://nginx.org/download/nginx-1.17.6.tar.gz

然后解压:

tar -zxvf nginx-1.17.6.tar.gz



编译安装

接着我们进入编译安装环节，首先进入刚才解压的 nginx 目录当中，如果只安装 Nginx，我们直接执行./configure 即可，但是我们需要将刚才下载的插件 nginx-http-flv-module 加入进来，

执行的命令如下所示

./configure --add-module=../nginx-http-flv-module-master

接下来执行

Make & make install

Nginx 以及 rtmp 模块就安装完成了，一般情况下，Nginx 被安装在/usr/local 目录中

配置 rtmp 服务

在完成 Nginx 的安装之后，还不能支持直播和点播，需要进行配置

编辑 Nginx 的配置文件 nginx.conf

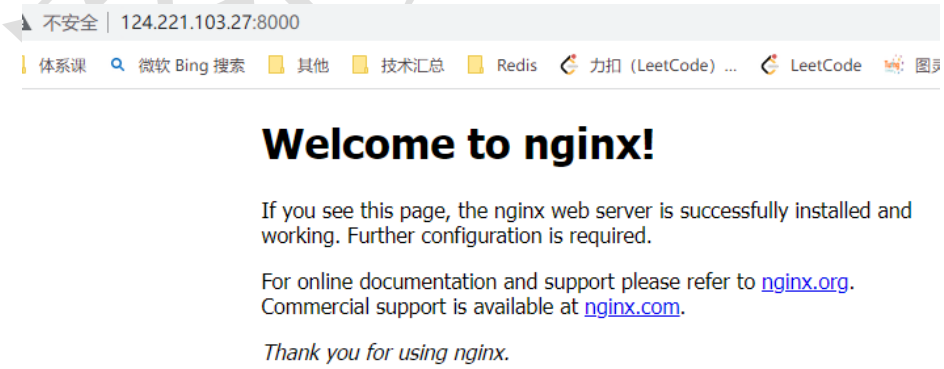
```
rtmp{
    server{
        listen 1935;
        chunk_size 4096;
        application live1{
                live on;
        }

        application hls1{
                live on;
                hls on;
                hls_path /usr/local/nginx/html/hls1;
        }

        application vod{
                play /usr/local/nginx/vod;
        }
    }
}
```

主要是增加了和点播直播相关的 rtmp 配置，设置了端口为 1935，点播相关的是 vod 条目，直播相关的是 live1 和 hls1 条目。在图上未显示的是 http 访问端口为 8000，因为同一台服务器上 Nodejs 占用了 80 端口。

可以通过./nginx -t 检查配置文件的正确性

```
[root@VM-16-8-centos sbin]# ./nginx -t
nginx: the configuration file /usr/local/nginx/conf/nginx.conf syntax is ok
nginx: configuration file /usr/local/nginx/conf/nginx.conf test is successful
```

配置完成后启动 Nginx，通过访问 http://xxxx:8000 来验证 Nginx 是否成功启动。



点播访问

点播的意思很明显，就是服务器已经存在视频，根据我们的需求来播放。于是我们在 vod 条目指定的目录下放置视频

```
[root@VM-16-8-centos vod]# pwd
/usr/local/nginx/vod
[root@VM-16-8-centos vod]# ls
01.mp4
```

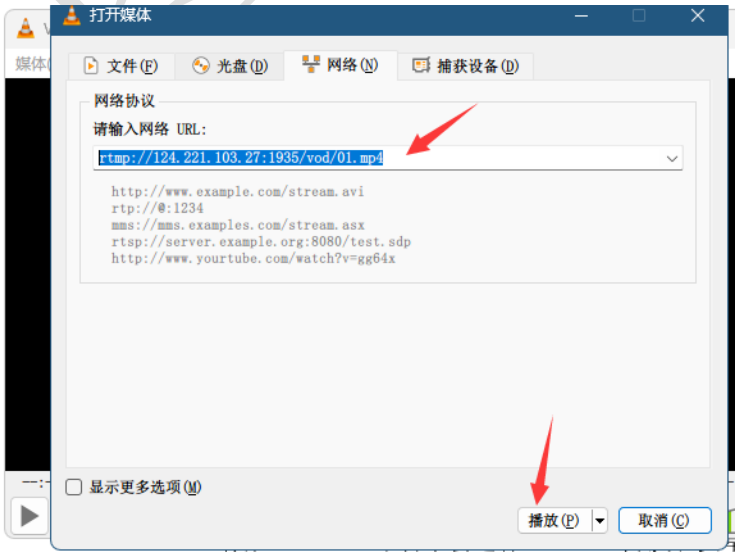
而且很明显，我们需要访问的不是普通 WEB 网页，自然需要单独的播放器来支持视频的播放。这里我们选择了 VLC 播放器。

VLC 播放器支持多种常见音视频格式，支持多种流媒体传输协议，也可当做本地流媒体服务器使用，功能十分强大。官网下载地址:
https://www.videolan.org/

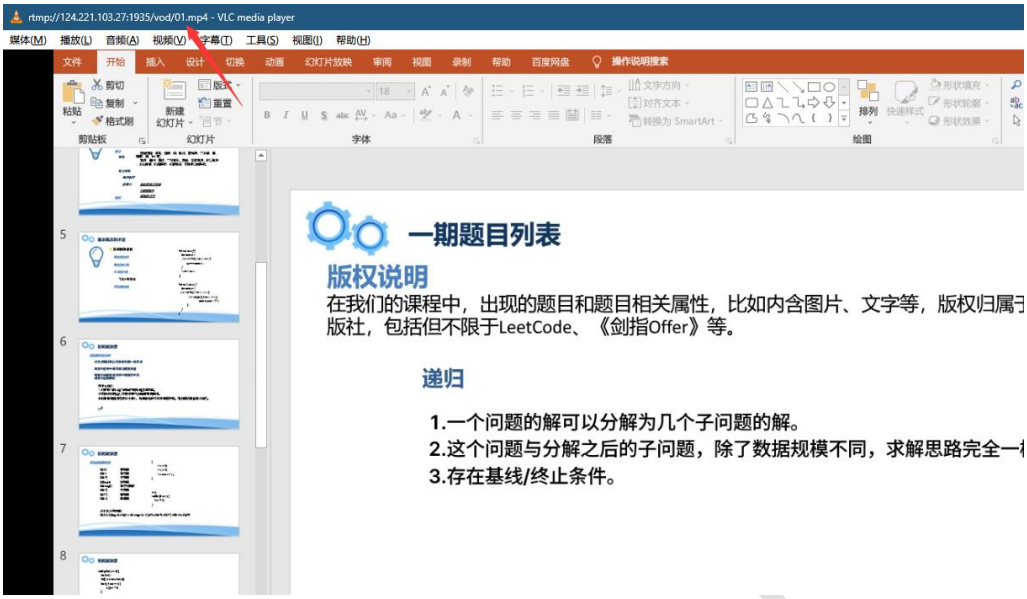
从官网上我们可以知道 VLC 多媒体播放器(最初命名为 VideoLAN 客户端)是 VideoLAN 计划的多媒体播放器。它支持众多音频与视频解码器及文件格式，并支持 DVD 影音光盘，VCD 影音光盘及各类流式协议。它也能作为 unicast 或 multicast 的流式服务器在 IPv4 或 IPv6 的高速网络连接下使用。它融合了 FFmpeg 的解码器与 libdvdcss 程序库使其有播放多媒体文件及加密 DVD 影碟的功能。

打开 VLC media player，在菜单上选择“媒体”-“打开网络串流”，在出现的窗口上输入网络 url

rtmp://124.221.103.27:1935/vod/01.mp4



根据服务器和带宽的情况，稍后 VLC media player 就开始播放相关的视频。



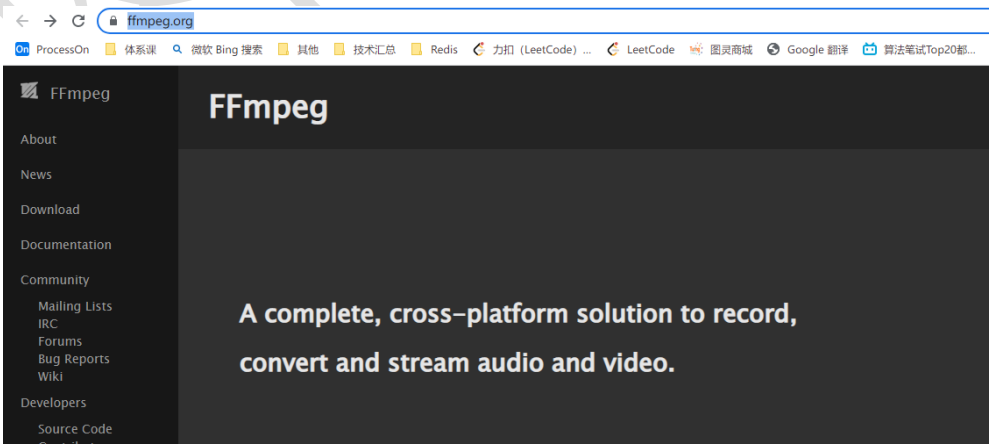
直播访问

点播访问的是服务器上已经存在的视频资源，但是直播很明显是没有这么一个已经存在的视频资源，需要我们实时推送视频，怎么才能做到？这就需要 Ffmpeg(读音读作“ef ef em peg”)了。

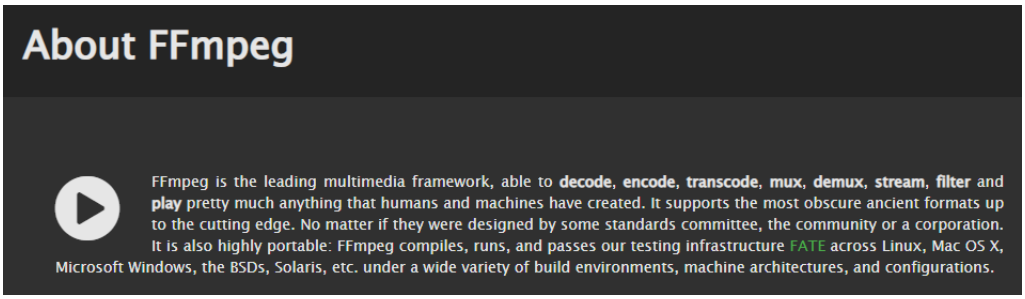
FFmpeg 简介

ffmpeg 是一个跨平台的音视频处理库，是一套可以用来记录、转换数字音频、视频，并能将其转化为流的开源计算机程序。采用 LGPL 或 GPL 许可证。它提供了录制、转换以及流化音视频的完整解决方案。现在我们所用的各种播放器底层基本上都是基于 ffmpeg 进行封装或二次开发的。

官网：https://ffmpeg.org/

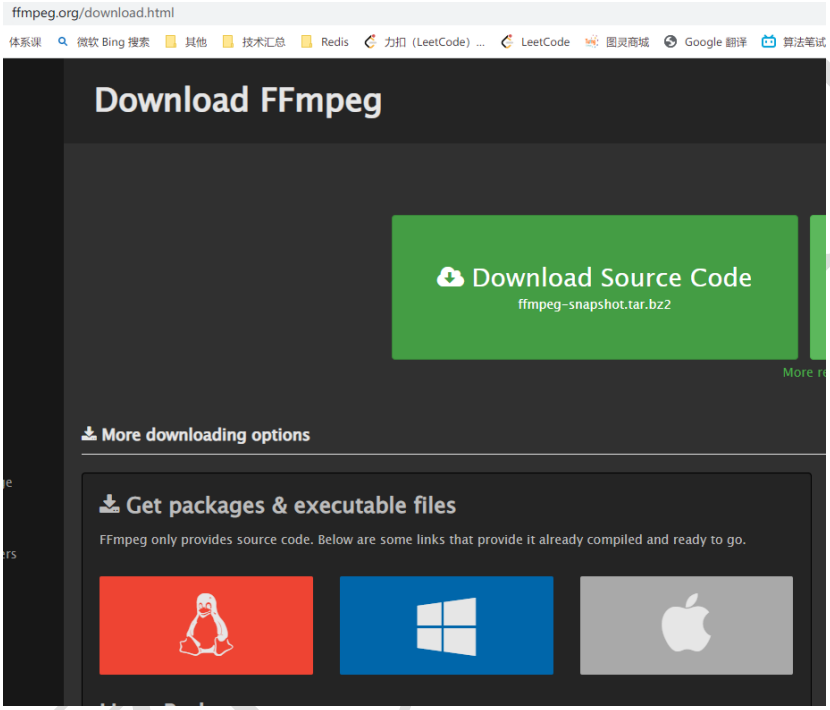


从官网本身的介绍来看



FMPEG 是一个多媒体框架，其包含了多个模块库:AVFormat，AVCodec，AVFilter，AVDivice，AVUtil 等，并且提供了基于这些库的三个命令行工具。

下载地址：https://ffmpeg.org/download.html



我们是基于本地推流的，自然就是下载 windows 版本的 ffmpeg，下载后解压缩，有个 bin 目录，里面有三个可执行程序

ffmpeg：该项目提供的一个工具，可用于格式转换、解码或电视卡即时编码等，也包括我们即将使用的推流；

ffplay：是一个简单的播放器，使用 ffmpeg 库解析和解码;

ffprobe：主要用来查看多媒体文件的信息。

当然，因为我们要推流，所以使用的主要是使用 ffmpeg，ffmpeg 的使用方式分为两种:

一种方式是直接使用 ffmpeg 来进行多媒体处理;另一种是使用 ffmpeg 封装的库进行二次开发，不管是哪种方式，对 Java 都不是太友好。

通过 FFmpeg 命令行我们能做到的事情有很多，包括：

1)列出支持的格式

2)剪切一段媒体文件

3)提取一个视频文件中的音频文件

4)从 MP4 文件中抽取视频流导出为裸的 H264 数据

5)视频静音，即只保留视频

6)使用 AAC 音频数据和 H264 视频生成 MP4 文件

7)音频格式转换

8)从 WAV 音频文件中导出 PCM 裸数据

9)将一个 MP4 的文件转换为一个 GIF 动图

10)使用一组图片生成 gif

11)淡入效果器使用

12)淡出效果器使用

13)将两路声音合并，比如加背景音乐

14)为视频添加水印效果

15)视频提亮效果器

16)视频旋转效果器的使用

17)视频裁剪效果器的使用

18)将一段视频推送到流媒体服务器上

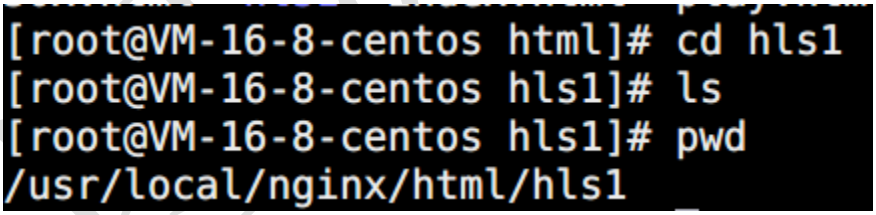
19)将流媒体服务器上的流 dump 到本地

20)将两个音频文件以两路流的形式封装到一个文件中

总的来说，对视频的各种操作都可以借助 Ffmpeg，为此 FFmpeg 提供了很多命令行参数，具体如何使用请自行查阅官方文档。

HLS 直播实战

首先我们看到在 Nginx 服务器上/usr/local/nginx/html/hls1 下没有任何文件



进入 ffmpeg 的目录，在命令提示符下执行

.\ffmpeg -re -i D:\Temp\03.mp4 -c copy -f flv rtmp://124.221.103.27:1935/hls1/test

-re 控制读取 AVpacket 的速度，按照帧率速度读取文件

-i 设定输入流

-c copy “-c”，是“codec，编解码器”的意思，加上 copy 表示使用与源文件相同的编码器

-f 指定输出文件格式

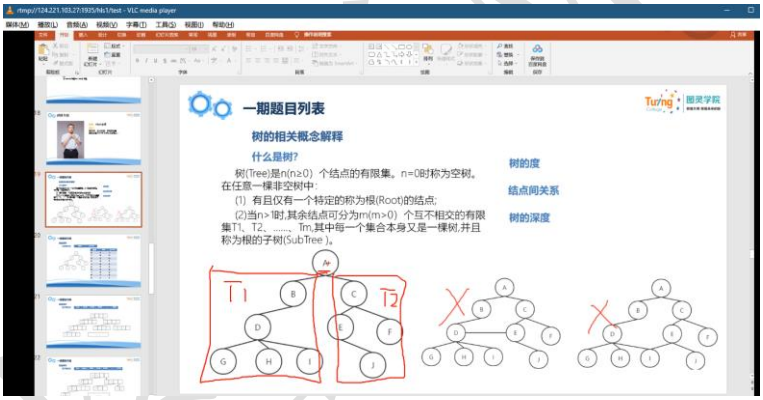
```
PS D:\TuLing\VIPL\Live\src\ffmpeg\bin> .\ffmpeg -re -i D:\Temp\03.mp4 -c copy -f flv rtmp://124.221.103.27:1935/hls1/test
ffmpeg version 2023-01-01-git-62da0b4a74-full_build-www.gyan.dev Copyright (c) 2000-2023 the FFmpeg developers
  built with gcc 12.1.0 (Rev2, Built by MSYS2 project)
  configuration: --enable-gpl --enable-version3 --enable-static --disable-w32threads --disable-autodetect --enable-fontconfig
--enable-iconv --enable-gnutls --enable-libxml2 --enable-gmp --enable-bzlib --enable-lzma --enable-libsnappy --enable-zlib --e
nable-librist --enable-libsrt --enable-libssh --enable-libzmq --enable-avisynth --enable-libbluray --enable-libcaca --enable-s
dl2 --enable-libaribb24 --enable-libdav1d --enable-libdavs2 --enable-libuavs3d --enable-libzvbi --enable-librav1e --enable-lib
svtav1 --enable-libwebp --enable-libx264 --enable-libx265 --enable-libxavs2 --enable-libxvid --enable-libaom --enable-libjxl -
-enable-libopenjpeg --enable-libvpx --enable-mediafoundation --enable-libass --enable-frei0r --enable-libfreetype --enable-lib
fribidi --enable-liblensfun --enable-libvidstab --enable-libvmaf --enable-libzimg --enable-amf --enable-cuda-llvm --enable-cuv
id --enable-ffnvcodec --enable-nvdec --enable-nvenc --enable-d3d11va --enable-dxva2 --enable-libvpl --enable-libshaderc --enab
```

```
  Stream #0:1(und): Audio: aac (LC) ([10][0][0][0] / 0x000A), 48000 Hz, stereo, fltp, 319 kb/s (default)
    Metadata:
      handler_name    : SoundHandler
      vendor_id       : [0][0][0][0]
Stream mapping:
  Stream #0:0 -> #0:0 (copy)
  Stream #0:1 -> #0:1 (copy)
Press [q] to stop, [?] for help
frame=  324 fps=9.8 q=-1.0 size=    2399kB time=00:00:33.08 bitrate= 594.0kbits/s speed=   1x
```

此时，ffmpeg 就开始将我们机器上的 03.mp4 推流到了服务器上，此时我们检查服务器上/usr/local/nginx/html/hls1 目录下，发现已经生成了视频的切片文件，而且随着时间的推移，切片文件也在不停的变化。

```
[root@VM-16-8-centos hls1]# ls
test-10.ts  test-11.ts  test-12.ts  test-13.ts  test-14.ts  test-15.ts  test-6.ts  test-7.ts  test-8.ts  test-9.ts  test.m3u8
[root@VM-16-8-centos hls1]# ls
test-10.ts  test-11.ts  test-12.ts  test-13.ts  test-14.ts  test-15.ts  test-6.ts  test-7.ts  test-8.ts  test-9.ts  test.m3u8
[root@VM-16-8-centos hls1]# ls
test-10.ts  test-12.ts  test-14.ts  test-16.ts  test-7.ts  test-9.ts
test-11.ts  test-13.ts  test-15.ts  test-6.ts   test-8.ts  test.m3u8
[root@VM-16-8-centos hls1]# ls
test-11.ts  test-12.ts  test-13.ts  test-14.ts  test-15.ts  test-16.ts  test-17.ts  test-18.ts  test.m3u8
```

同样通过 VLC 播放器播放网络流：
http://124.221.103.27:8000/hls1/test.m3u8



注意：因为服务器性能、带宽和 hls 本身的特性，显示视频往往需要一定的时间，而且有时候会出现只有声音没有图像的情况，此时可能需要进行检查，或者重新打开网络串流。

浏览器播放

看到这里，有同学会想到，既然支持 http，浏览器能不能播放呢？答案是可以，但是需要一定的步骤。

首先生成一个 Html 网页

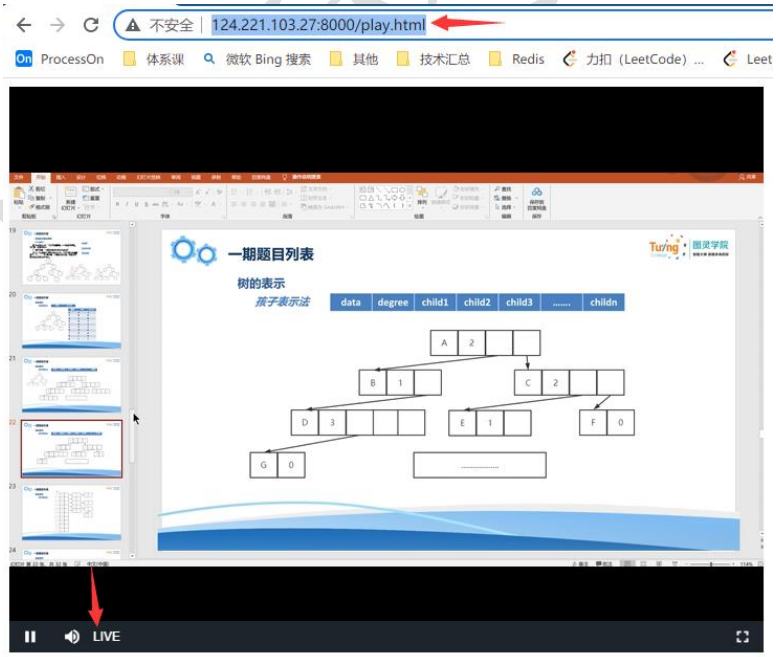
```
<!DOCTYPE html>
<html lang="zh-CN">

<head>
<meta charset="UTF-8">
<title>demo</title>
<link href="https://vjs.zencdn.net/7.0.3/video-js.css" rel="stylesheet">
</head>
<body>
<video id="myVideo" class="video-js vjs-default-skin vjs-big-play-centered" controls preload="auto" width="720" height="540" data-setup='{}'>
<source id="source" type="application/x-mpegURL" src="http://124.221.103.27:8000/hls1/test.m3u8">
</video>
</body>
<script src="https://vjs.zencdn.net/7.0.3/video.js"></script>
</html>
```

关键是要引入 video.js 和相关的 video-js.css。相对于 flv.js，flv.js 更聚焦在多媒体格式方面，其主要是将 FLV 格式转换为 MP4 格式，而对于播放器的音量控制、进度条、菜单等 UI 展示部分没有做特别的处理。而 video.js 对音量控制、进度条、菜单等 UI 相关逻辑做了统一处理，对媒体播放部分设计了一个插件框架，可以集成不同媒体格式的播放器进去。所以相比较而言，video.js 更像是一款完整的播放器。可以说 video.js 是目前在浏览器上最好用、最著名的开源流媒体播放器。它的功能非常强大，既可以处理多种多媒体格式，如 MP4、FLV 等；又可以支持多种传输协议，如 HLS、RTMP。因此，它是播放音视频直播媒体流必不可少的播放工具。

其次，这个网页建议不要在本地直接打开，通过访问 web 服务器后打开更好，所以我们将这个网页上传到了 Nginx 服务器上。

```
[root@VM-16-8-centos html]# ls
50x.html  hls1  index.html  play.html
[root@VM-16-8-centos html]# pwd
/usr/local/nginx/html
[root@VM-16-8-centos html]#
```

然后通过浏览器打开这个网页：http://124.221.103.27:8000/play.html



RTMP 直播实战

除了 HLS 格式的直播，我们同样也可以实现 rtmp 格式的直播，方法大同小异。

在本地执行 .\ffmpeg -re -i D:\Temp\03.mp4 -c copy -f flv rtmp://124.221.103.27:1935/live1/test

注意，这个时候服务器上接受推流的地址变为 live1，我们也知道在 Nginx 的配置中，live1 条目不像 hls1 条目，它是不存在视频切片保存目录的。

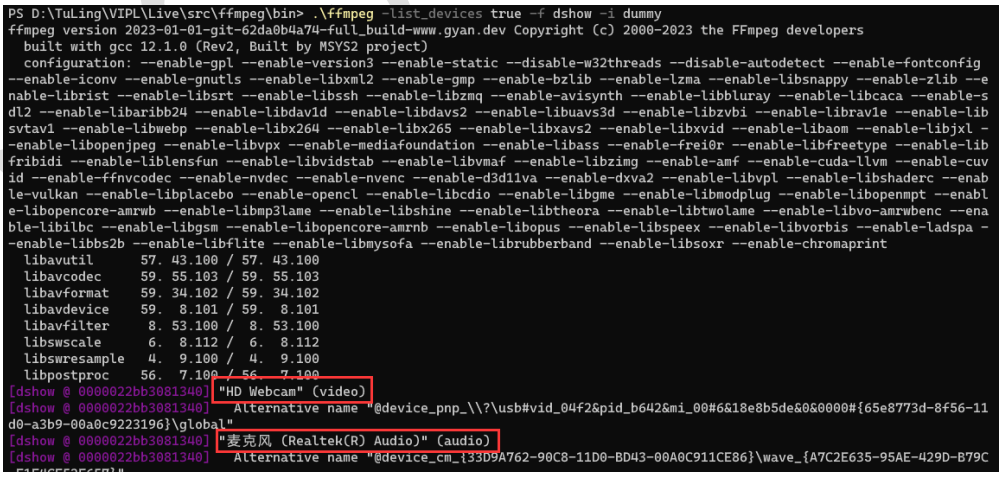
进行推流后，在 VLC 播放器播放网络流 rtmp://124.221.103.27:1935/live1/test

摄像头推流

对摄像头推流其实和我们上面的推流视频文件并没有本质区别，只不过音视频来源不同而已，可以在命令提示符下执行

.\ffmpeg -list_devices true -f dshow -i dummy

首先找到本地的设备

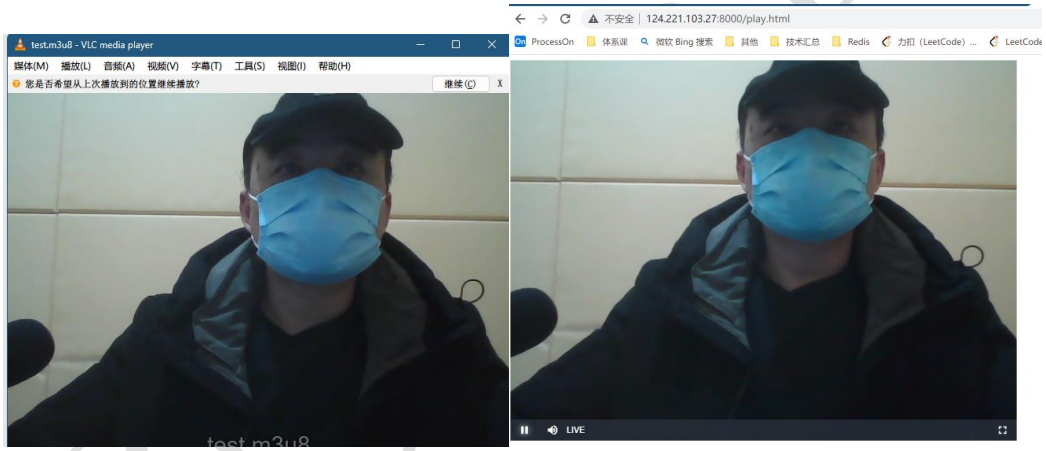


再执行

.\ffmpeg -f dshow -i video="HD Webcam":audio="麦克风 (Realtek(R) Audio)" -vcodec libx264 -acodec aac -f flv rtmp://124.221.103.27:1935/hls1/test

```
PS D:\TuLing\VIPL\Live\src\ffmpeg\bin> .\ffmpeg -f dshow -i video="HD Webcam":audio="麦克风 (Realtek(R) Audio)" -vcodec libx26
4 -acodec aac -f flv rtmp://124.221.103.27:1935/hls1/test
ffmpeg version 2023-01-01-git-62da0b4a74-full_build-www.gyan.dev Copyright (c) 2000-2023 the FFmpeg developers
  built with gcc 12.1.0 (Rev2, Built by MSYS2 project)
  configuration: --enable-gpl --enable-version3 --enable-static --disable-w32threads --disable-autodetect --enable-fontconfig
--enable-iconv --enable-gnutls --enable-libxml2 --enable-gmp --enable-bzlib --enable-lzma --enable-libsnappy --enable-zlib --e
nable-librist --enable-libsrt --enable-libssh --enable-libzmq --enable-avisynth --enable-libbluray --enable-libcaca --enable-s
dl2 --enable-libaribb24 --enable-libdav1d --enable-libdavs2 --enable-libuavs3d --enable-libzvbi --enable-librav1e --enable-lib
svtav1 --enable-libwebp --enable-libx264 --enable-libx265 --enable-libxavs2 --enable-libxvid --enable-libaom --enable-libjxl -
-enable-libopenjpeg --enable-libvpx --enable-mediafoundation --enable-libass --enable-frei0r --enable-libfreetype --enable-lib
fribidi --enable-liblensfun --enable-libvidstab --enable-libvmaf --enable-libzimg --enable-amf --enable-cuda-llvm --enable-cuv
id --enable-ffnvcodec --enable-nvdec --enable-nvenc --enable-d3d11va --enable-dxva2 --enable-libvpl --enable-libshaderc --enab
le-vulkan --enable-libplacebo --enable-opencl --enable-libcdio --enable-libgme --enable-libmodplug --enable-libopenmpt --enabl
e-libopencore-amrwb --enable-libmp3lame --enable-libshine --enable-libtheora --enable-libtwolame --enable-libvo-amrwbenc --ena
ble-libilbc --enable-libgsm --enable-libopencore-amrnb --enable-libopus --enable-libspeex --enable-libvorbis --enable-ladspa -
-enable-libbs2b --enable-libflite --enable-libmysofa --enable-librubberband --enable-libsoxr --enable-chromaprint
  libavutil      57. 43.100 / 57. 43.100
  libavcodec     59. 55.103 / 59. 55.103
  libavformat    59. 34.102 / 59. 34.102
  libavdevice    59.  8.101 / 59.  8.101
  libavfilter     8. 53.100 /  8. 53.100
  libswscale      6.  8.112 /  6.  8.112
  libswresample   4.  9.100 /  4.  9.100
  libpostproc    56.  7.100 / 56.  7.100
Guessed Channel Layout for Input Stream #0.1 : stereo
Input #0, dshow, from 'video=HD Webcam:audio=麦克风 (Realtek(R) Audio)':
  Duration: N/A, start: 327257.378000, bitrate: 1411 kb/s
  Stream #0:0: Video: mjpeg (Baseline) (MJPG / 0x47504A4D), yuvj422p(pc, bt470bg/unknown/unknown), 640x480, 30 fps, 30 tbr, 10
000k tbn
  Stream #0:1: Audio: pcm_s16le, 44100 Hz, 2 channels, s16, 1411 kb/s
Stream mapping:
  Stream #0:0 -> #0:0 (mjpeg (native) -> h264 (libx264))
  Stream #0:1 -> #0:1 (pcm_s16le (native) -> aac (native))
Press [q] to stop, [?] for help
[libx264 @ 0000027a21434700] using cpu capabilities: MMX2 SSE2Fast SSSE3 SSE4.2 AVX FMA3 BMI2 AVX2
```

就可完成基于摄像头的推流，同样的也可以在浏览器和 VLC 播放器中看到相应的画面。



用 Java 实现直播推流

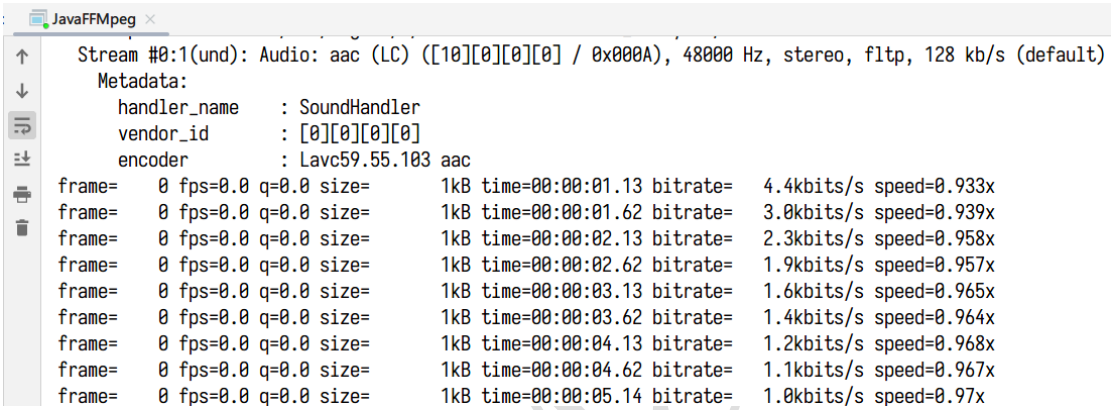
作为 Java 程序员自然关心能不能使用 Java 来实现推流？如何实现？答案是能，但不是很方便，所以真正开发播放器或者直播项目往往还是 C/C++/C#为主。

前面我们说过，对 ffmpeg 一种方式是直接使用 ffmpeg 提供的三个命令行工具来进行多媒体处理；另一种是使用 ffmpeg 封装的这些库进行二次开发。对于 Java 来说，第二种方式很麻烦，所以一般用第一种方式。说到底就是在 Java 中执行 ffmpeg 的可执行程序。

Java 中执行第三方可执行程序，当然是使用 Runtime.getRuntime().exec()方法，具体代码如下：

```
public class JavaFFMpeg extends Thread{
    5 usages
    private String ffmpegCmd = "";
    public String getTheFmpgCmdLine () { return ffmpegCmd; }
    1 usage
    public void setTheFmpgCmdLine (String theFmpgCmdLine) { this.ffmpegCmd = theF

    public void run(){
            openFFmpegExe();
            System.out.println("fmpg thread over");
    }
    1 usage
    private void openFFmpegExe () {
        if (ffmpegCmd == null || ffmpegCmd.equals("")) {
            return;
        }
        //开启一个进程
        Runtime rn = Runtime.getRuntime();
        Process p = null;
        try {
            //传递ffmpeg推流命令行
            p = rn.exec(ffmpegCmd);
            //ffmpeg输出的都是"错误流": stdin,stdout,stderm,
            BufferedInputStream in = new BufferedInputStream(p.getErrorStream());
            BufferedReader inBr = new BufferedReader(new InputStreamReader(in));
            String lineStr;
            System.out.println("开始...");
            while ((lineStr = inBr.readLine()) != null) {
                //获得命令执行后在控制台的输出信息
                System.out.println(lineStr);//打印输出信息
            }
            inBr.close();
            in.close();
        } catch (Exception e) {
            System.out.println("Error exec: " + e.getMessage());
        }
    }

    public static void main(String[] args) throws Exception {
        System.out.println ( " ---开始推流---") ;
        JavaFFMpeg javaFFMpeg = new JavaFFMpeg();
        String ffmpegCmd = "D:\\TuLing\\VIPL\\Live\\src\\ffmpeg\\bin\\ffmpeg -re
        javaFFMpeg.setTheFmpgCmdLine (ffmpegCmd);
        javaFFMpeg.start ();
        javaFFMpeg.join() ;
        System .out.println ( "---结束推流---" );
    }
}
```

我们在一个线程中执行具体的 exec()方法，方法需要的参数则由外部传入，这个参数其实就是我们前面在命令行直接执行的 ffmpeg 相关命令，而对 ffmpeg 在执行过程中输出信息，我们使用 Java 中的输入输出流获得后显示在控制台，可以看到，执行后输出的信息和我们在命令行直接执行的 ffmpeg 输出的信息几乎一致。



能推流到 Nginx 上，自然就能播放，我们这里就不再显示在 VLC 中的播放内容了。

支持万人同时在线的直播系统实战推演

前面的课程我们已经知道了如何自行实现推流和播放，但是支持万人同时在线的大型直播系统却不仅仅只有这些。

从技术构成角度来说，我们在分布式课程中学习到的缓存机制、异步读写、读写分离，数据分片、批量写、读写的热点分散这些应对高并发的读写技术在大型直播系统中依然是适用的，只是在具体的实现会根据直播系统的特点进行调整。

从系统构成来说，一个商用生产级的大型直播系统其实应该包括两个部分，一个是我们前面一直学习的音视频的推流与播放相关的音视频系统，另一个则是直播系统中 IM 消息系统。因为我们对推流和播放已经有了很多的了解，现在来看看直播系统中 IM 消息系统的架构实践。

万人在线直播间实时聊天消息架构设计

在目前直播系统中，IM 实时消息技术则是实现观看直播的所有用户和主播实现互动的关键技术点。因为通过直播系统中的 IM 消息模块，可以完成公屏互动、彩色弹幕、全网送礼广播、私信、PK 等核心秀场直播的功能开发。“IM 消息”是用户和用户、用户和主播之间“沟通”的信息桥梁。

直播消息的技术特征

在直播业务中，有几个关于消息模型的核心概念，我们先简单地总结一下，方便大家对直播相关的消息模型有一个整体上的理解。

直播消息的实体

直播系统消息模块对应的实体就是主播和观众。

主播和观众：对于 IM 系统来说，都是普通用户，都会有一个唯一用户标识（用户 ID），它也是 IM 分发到点对点消息的重要标识。

主播和房间号：一个主播对应一个房间号（RoomId），主播在开播之前，进行身份信息验证之后，就会绑定唯一的房间号，房间号是 IM 系统进行直播间消息分发的重要标识。

直播消息类型划分

按照直播业务特性，IM 消息划分的方式有很多方式，例如：

1）按照接收方维度进行划分；

2）按照直播间消息业务类型进行划分；

3）按照消息的优先级进行划分；

4）按照消息的存储方式进行划分等等。

按照接收方维度，可以这样划分：

1）点对点消息（单聊消息）；

2）直播间消息（群聊消息）；

3）广播消息（系统消息）。

按照具体的业务场景，可以这样进行划分：

1）礼物消息；

2）公屏消息；

3）PK 消息；

4）业务通知类消息。

直播消息优先级

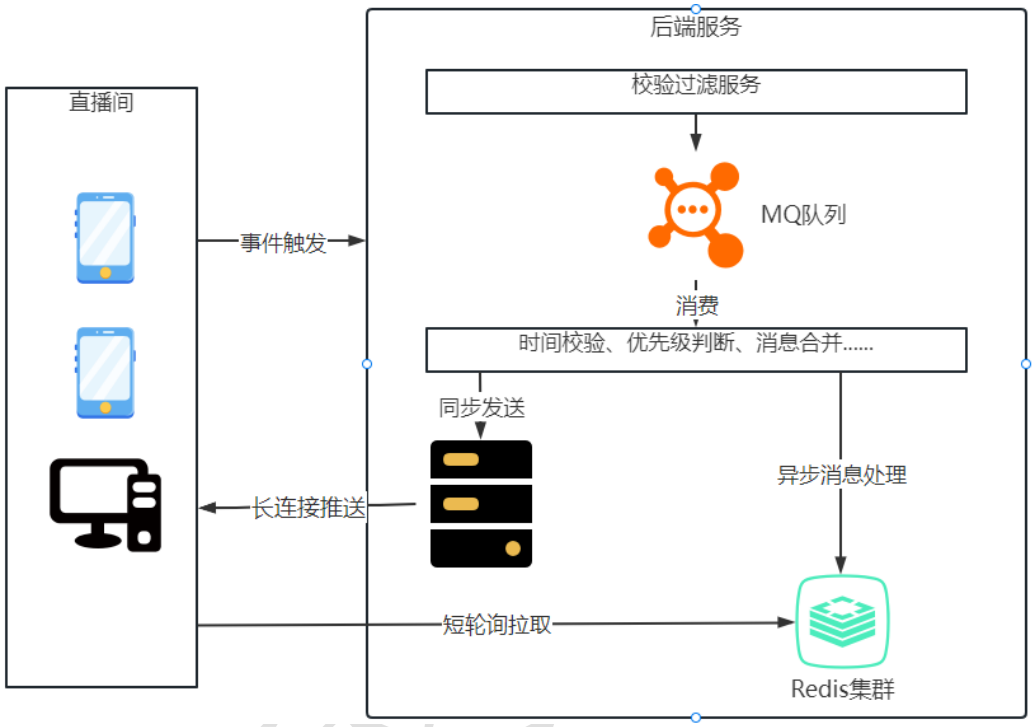
直播系统中的 IM 消息是有优先级的，这一点是很重要的，与微信、QQ 等标准社交聊天 IM 产品不一样的地方是：直播间消息是分优先级的。

微信等标准社交 IM 产品，不管是私聊还是群聊，每个人发送消息的优先级基本上是一样的，不存在谁的消息优先级高，谁的消息优先级低，都需要将消息准确实时地分发到各个业务终端。但是直播因为业务场景的不同，消息分发的优先级也是不一样的。

为什么要做消息的优先级划分呢？如果一个直播间每秒只能渲染 15~20 个消息，一个热点直播间一秒钟产生的消息量大于 20 条或者更多，如果不做消息优先级的控制，直接实时分发消息，那么导致的结果就是直播间公屏客户端渲染卡顿，礼物弹框渲染过快，用户观看体验大幅下降。所以我们要针对不同业务类型的消息，给出不同的消息优先级。

一般来说，礼物消息大于公屏消息，同等业务类型的消息，大额礼物的消息优先级又大于小额礼物的消息，高等级用户的公屏消息优先级高于低等级用户或者匿名用户的公屏消息，在做业务消息分发的时候，需要根据实际的消息优先级，选择性地进行消息准确地分发。

直播系统的消息模块架构模型



从上图我们可以看到，消息模块中消息的交互方式就是一般是推拉结合的，重要的消息一般是同步发送，并用长连接进行消息的分发和推送。一般的消息基本上是异步处理，而且采用短轮询的方式进行消息的拉取。

短轮询

短轮询实现

短轮询很好理解，就是客户端定时从服务器去主动拉取消息，在具体实现上，一般设定一个轮询间隔，每次轮询附带房间 ID 和时间戳；服务器根据房间 ID 和房间 ID 查询该房间在时间戳后产生的消息事件，返回限定条数的消息例如（例如返回 10~15 条，当然在这个 timestamp 之后产生的消息数远远大于 15 条，不过因为客户端渲染能力有限和过多的消息展示，会影响用户体验，所以限制返回的条数），并且同时返回这些消息中最后一条消息产生的时间戳，作为客户端下次请求服务器的基准请求时间戳；以此反复，这样就可以以轮询间隔按照各个终端要求，更新每个直播间的最新消息了。

短轮询的存储模型

适合于短轮询的消息，它的存储需要：

1）消息插入时间复杂度要相对比较低；

2）消息查询的复杂度要相对比较低；

3）消息的存储的结构体要相对比较小，不能占用太大的内存空间或者磁盘空间；

4）历史消息能够按照业务需要做磁盘持久化存储。

结合以上 4 点 Redis 的 SortedSet 数据结构是一个很好的选择。

前面我们说过，按照直播间产品业务类型，业务消息可以划分为如下四大类型：礼物、公屏、PK、通知。

一个直播间的四种业务类型消息就可以使用四个 Redis 的 SortedSet 数据结构进行存储。

比如以 房间 id + 业务类型 为键，score 是消息真实产生的时间戳，value 就是序列化好的 json 字符串。

为何选用 Redis 的 SortedSet 而不是 List 呢？主要原因如下：虽然 list 的 LPUSH 或者 RPUSH 命令以及获取列表长度命令的时间复杂度 o(1)，但是需要删除过期数据时，比较难以维护；但是 SortedSet 虽然使用 ZADD 命令增加新数据时时间复杂度是 O(log(N))，但因为按自然时间作为 score 进行顺序插入其实真实时间复杂度接近 O(1)，而删除元素时使用 zrangeWithScores 命令和

zremrangeByScore 命令可以较快进行删除。而最重要的查询，list 的 lrange 命令时间复杂度是 O(N)，而 SortedSet 的只有 O(log(N))+k)，其中 k 是要获取的成员个数，N 是当前有序集合成员个数，在 N>>k 的情况下，自然 SortedSet 更合适。

短轮询的时间控制

短轮询的时间控制及其重要，我们需要在直播观众观看体验和服务器压力之间找到一个很好的平衡点。

轮询的间隔时间长：用户体验就会下降很多，直播观看体验就会变差，会有"一顿一顿"的感觉。

短轮询的频率过高：会导致服务器的压力过大，也会出现很多次"空轮询"，所谓的"空轮询"就是无效轮询，也就是在上一秒有效轮询返回有效消息之后，间隔期直播间没有产生新的消息，就会出现无效的轮询。

而且在，直播高峰期的时候，查询次数可能达到上亿次，对业务服务器和 Redis 都会施加很大的压力。这块需要根据机器的和 Redis 的实时负载的压力，做服务器的水平扩容和 Redis Cluster 的节点扩容，甚至让一些超高热度值的直播间负载到指定的 Redis Cluster 集群上，做到物理隔离，确保各个直播间的消息相互不影响。

直播人数不一样、查询压力不一样的的直播间，轮询的时间可能是完全不同的，比如：

例如人数比较少的直播间，可以设置比较高频的轮询频率；中型直播间可以较长一些；万人直播间就需要设置的更长一些。

所以这些配置应该都可以通过配置中心实时下发，客户端能够实时更新轮询的时间，调整的频率可以根据实际直播间用户体验的效果，并且结合服务器的负载，找到一个轮询间隔的相对最佳值。

短轮询的注意点

1）服务端需要校验客户端传递过来的时间戳：

这一点非常重要，试想一下，如果观众在观看直播的时候，将直播退出后台，客户端轮询进程暂停，当用户恢复直播观看画面进程的时候，客户端传递过来的时间就会是非常老旧甚至过期的时间，这个时间会导致服务器查询 Redis 时出现慢查。

2）客户端需要校验重复消息：

在极端情况下，客户端有可能收到重复的消息，产生的原因可能如下，在某一个时刻客户端发出的请求，因为网络不稳定或者服务器 GC 的原因，导致该请求处理比较慢，耗时超过 2s，但是因为轮询时间到了，客户端又发出了的请求，服务器返回部分重复的数据，就会出现客户端重复渲染相同的消息进行展示。这样也会影响用户体验，所以每一个客户端有必要对重复消息进行校验。

3）海量数据无法实时返回渲染的问题：

设想一下，如果一个热度极大的直播间，每秒钟产生的消息量是数千或者上万的时候，因为我们每次因为各个因素的限制，每次只返回 10~20 条消息，那么我们需要很长的时间才能把这热度很多的一秒钟的数据全部返回，这样就会造成最新的消息无法快速优先返回。所以轮询返回的消息需要按照消息优先级进行选择性丢弃。

4）短轮询比较适合可能存在消息扩散的消息。什么是消息扩散？来举个例子，在一个有 10000 人同时在线的房间里，如果其中一个用户发送了文字消息，那么服务端收到该消息之后就要给 10000 人转发。如果主播说“请能听到我声音的人回复 1”，那这时 10000 人同时发消息，服务端要转发多少条呢？要转发 10000 * 10000=1 亿条消息。

而从我们上面对短轮询的设计就知道，短轮询机制天生就没有消息扩散的问题。

长连接

基本实现

客户端首先通过 http 请求长连接服务器，获取 TCP 长连接的 IP 地址，长连接服务器根据路由和负载策略，返回最优的可连接的 IP 列表；一般来说集群按照地域进行业务划分，不同地域的终端机器按需接入；

客户端根据长连接服务器返回的 IP 列表，进行长连接的客户端的连接请求接入，长连接服务器收到连接请求，进而建立连接；

客户端发送鉴权信息，进行通信信息的鉴权和身份信息确认，最后长连接建立完成，长连服务器需要对连接进行管理，心跳监测，断线重连等操作。

不过要注意两点，一是当大量用户在线的时候，维护这些连接、保持会话，需要用到大量内存和 CPU 资源，所以一般接入层尽量保持功能简洁，尽量不要承载业务逻辑，业务逻辑应该下沉到后面的业务服务中进行处理，为了防止发布的时候，重启进程会导致大量的外网设备重新建立连接，影响用户体验。

对于长连接来说，需要判断客户端假在线的情况，特别是手机端因为移动设备，断线属于非常常见的情况，所以长连接需要进行维护，让连接的两端能够快速得到通知，然后通过重连来建立新的可用连接，从而让我们这个长连接一直保持高可用状态，同时也可以让服务器快速剔除断线的客户端，节约服务器资源，一般的实现手段是心跳机制。

直播间 IM 消息的分发策略

消息的分发一般遵循的原则是：

单聊、群聊、广播消息调用 IM 长连接分发；直播间用户行为产生的消息通过长连接分发还是短轮询分发，都是由直播业务服务器控制，由当时的系统情况和直播间人数等一系列条件来决定。

消息优先级

直播间中有进出场消息、文本消息、礼物消息和公屏消息等多种多样消息。消息的重要程度不一样，需要为每个消息设定相应的优先级。

不同优先级的消息放在不同的消息队列中，高优先级的消息优先发送给客户端，消息堆积超过限制时，丢弃最早、低优先级的消息。

直播间消息发送时：根据直播间成员分片通知对应的消息发送服务，再把消息分别下发给分片中对应的每一个用户。为了实时、高效地把直播间消息下发给用户，当用户有多条未接收消息时，下发服务采用批量下发的方式将多条消息发送给用户。

消息压缩

分发直播间消息的时候，需要注意消息体的大小。

如果某一个时刻，分发消息的数量比较大，或者同一个消息在做群播场景的时候，群播的用户比较多，IM 连接层的机房的出口带宽就会成为消息分发的瓶颈。

一般的优化手段从两方面着手，一是使用序列化后体积较小的框架，比如 protobuf，另外相同类型的消息进行可以合并发送。

批量消息

所谓批量消息，也就是多个消息进行合并发送。举个例子每秒分发 10~20 个消息，如果每秒中，业务服务器积累到的消息大于 10~20 个，那就按照消息的优先级进行丢弃。如果这 10~20 个消息的优先级都比较高，例如都是礼物类型的消息，则将消息放在后一个消息块进行发送。

这样做的好处如下：

1）减少传输消息头：合并消息，可以减少传输多余的消息头，多个消息一起发送，在自定义的 TCP 传输协议中，可以共用消息头，进一步减少消息字节数大小；

2）防止消息风暴：直播业务服务器可以很方便的控制消息分发的速度，不会无限制的分发消息到直播客户端，客户端无法处理如此多的消息；

3）提升用户体验：直播间的消息因为流速正常，渲染的节奏比较均匀，会带来很好的用户直播体验，整个直播效果会很流畅。

消息丢弃

例如：在游戏直播中，有出现比较精彩瞬间的时候，评论公屏数会瞬间增加，同时送低价值的礼物的消息也会瞬间增加很多，用来表示对自己选手精彩操作的支持，那么服务器通过 IM 长连接或者 http 短轮询每秒分发的消息数就会数千或者上万。

一瞬间的消息突增，会导致客户端出现如下几个问题：

1）客户端通过长连接获取的消息突增，下行带宽压力突增，其他业务可能会受到影响；

2）客户端无法快速处理渲染如此多的礼物和公屏消息，CPU 压力突增，音视频处理也会受到影响；

3）因消息存在积压，导致会展示过期已久消息的可能，用户体验会下降。

所以：因为这些原因，消息是存在丢弃的必要的。

举一个简单的例子：礼物的优先级一定是高于公屏消息的，PK 进度条的消息一定是高于全网广播类消息的，高价值礼物的消息又高于低价值礼物的消息。

所以在开发实践中，可以做如下的控制：

1）选择性丢弃低优先级消息、选择性丢弃“老”消息；

2）消息补偿：在业务开发中，消息的设计中，尽量地让后续到达的消息能够包含前续到达的消息。

举个例子：9 点 10 的消息，主播 A 和主播 B 的 PK 值是 20 比 10，到 9 点 11 分时，主播 A 又赢下 2 分，这个时候消息的分发有两种选择；直接分发增量消息 2:0，由客户端做累加（20+2 ： 10+0），另一种选择是分发 PK 消息就是 22 比 10。考虑到存在消息因为网络颤抖或者前置消息丢弃，导致消息丢失，所以选择分发 22 比 10 的消息会更合适。

万人在线大型直播音视频架构

前面说过大型直播音视频架构由直播客户端、信令服务器、支撑业务系统和 CDN 网络这几部分组成。

直播客户端主要包括音视频数据的采集、编码、推流、拉流、解码与播放这几个功能。而且要分开主播使用的客户端，包括音视频数据采集、编码和推流功能和观众使用的客户端，包括拉流、解码与渲染（播放）功能。

支撑业务系统需要完成转码录制转推，这包括整个 RTC 实时画面合成和转码的工作，CDN 转推以及云端录制保存。还包括监控、计费、接入以及增值业务四个模块。监控部分是针对线上服务运营情况的监测，方便我们与用户定位问题并优化产品。接入服务主要负责用户的就近接入以及负载均衡。

总的来说，支撑业务系统和信令服务器和我们一般的网站架构差别不大，要面对的问题和解决的方案也大同小异。

CDN 网络，主要用于媒体数据的分发。不可能所有的用户都直接访问直播服务器，一定会借助 CDN，而且是多家 CDN。在使用它们时，按照一定的比例将“节目”分配到不同的 CDN 网络上。

一般情况下，它先在各运营商内构建云服务，然后再将不同运营商的云服务通过光纤连接起来，从而实现跨运营商的全网 CDN 云服务。

CDN 网络从源节点接收到媒体数据后，会主动向各个主干结点传送流媒体数据，这样主干结点就将媒体数据缓存起来了。当然这个缓冲区的大小是有限的，随着时间流逝，缓冲区中的数据也在不断更替中。

当有观众想看某个主播的节目时，会从直播系统的信令服务器获取离自己最近的 CDN 边缘节点，然后到这个边缘节点去拉流。由于他是第一个在该节点拉流的用户，因此该 CDN 边缘节点，用于接收用户推送的媒体流。

所以总的来说，主干结点，起到媒体数据快速传递的作用，比如与其他运营商传送媒体流。

过缘节点，用于用户来主动接流。一般边缘节点的数量众多，但机子的性能比较低，它会被布署到各地级市，主要解决网络最后一公里的问题。

如果边缘节点还没有用户想要的媒体流，怎么办呢？那就向主干结点发送请求。主干结点收到请求后，从自己的缓冲区中取出数据流源源不断地发给边缘节点，这时边缘节点再将媒体数据发给观众。

当第二个观众再次到该 CDN 边缘节点接流时，该节点发现该流已经在自己的缓存里了，就不再向主干结点请求，直接将媒体流下发下去了。因此，观众在使用 CDN 网络时会发现，第一个观众在接流时需要花很长时间才能将流拉下来，可是后来的用户很快就将流拉下来进行播放了。

在万人直播架构里，CDN 网络是非常重要的组成部分，用户观看的体验好坏很大程度就是由 CDN 网络质量来决定。

万人直播音视频架构的性能优化

万人直播性能痛点

万人直播互动的难点有很多，经常遇到或普遍关注的主要问题：大流量、高并发以及用户分布。高并发的解决手段和大型网站没有太大差别。

大流量：实时互动本身就是一种大流量的数据交互活动，而万人直播则是在小规模直播互动形式的基础上进行了万级别的末端放大或中间链路的放大，因此其数据流量是非常庞大的，尤其是在拉流端。

用户分布：用户分布是多人实时活动中比较常见的一个问题。用户分布于全国各地，面对不同的运营商、不同的网络质量，就近接入该如何为每个用户提供更优质的观看体验。

比如负载均衡用来将流量分散，其次是就近接入，为用户选择最优质的节点进行调度。流量隔离以及分区保护也是应对大流量常用的手段。流量隔离是指服务器或者用户的业务可能会受到攻击和流量暴增，此时我们需要对存在问题的区域进行流量隔离，以降低风险。分区保护是指可能存在部分用户对分区

有一定需求（如活动产生的流量暴增），此时可能造成系统的不稳定，就需要对这一特定区域进行隔离，单独进行容量扩容，同时保证不会对其他用户服务产生影响。

推流接入

推流部分的流量一般不会很大，但是推流端影响的则是所有观众的体验。因此在推流端一定要保证接入质量的稳定，以及链路容灾的备份。推流端接入可以通过 DNS 进行分区后，分配多个接入地址，然后进行动态的 Ping 速，选择最快的地址为主接入，其余的为备份。

多路径接入：推流的边缘节点由多个接入点作为备份，如果一个点发生故障，推流端可以通过另外的节点继续进行推流，中断切换影响最多仅有秒级别。

拉流端

从推流端到拉流端一般会使用智能路由的路由转发系统，由多个路由节点进行各个区域路由转发的操作，同时也可进行几个节点间的灾备。路由节点的链路依旧要进行适当的冗余，链路一般会选择最优路径以及次优路径作为传输路径。延时、丢包是其中比较关键的衡量指标，跳数与成本两项指标也会占有一定权重，综合计算得出一条最优路径。所有的数据包经过路由转发后都会通过最优路径优先到达另外的边缘。次优路径则会作为灾备或最优路径变差情况下暂时性的链路传输。

万人直播系统中拉流端一般优化最多的地方。其中常见的问题有以下几种：

第一，用户的网络是不对等的，要解决不对等网络下用户的体验。

第二，万人直播互动中，端上的解码性能可能会遇到瓶颈。在同时接收和解码多路码流时，由于移动端解码能力有限，可能会导致用户体验骤降。

第三，在系统达到极限的情况下，应该通过柔性的降级使得用户的实时交互正常进行。

用户的位置、网络运营商、网络（4G/WIFI）千差万别，而远端推流来源相同。针对这种情况，在 Web 端常用的解决方式为多播，压力主要集中在推流端上传，需要编码多路流上传（常见的是一路高质量流和一路低质量流）。用户端只需要根据自己的实际情况选择相应质量流完成下发，以此达到不同用户可以享受不同的用户级别。在 Native 端更多采用的是分层的编码。

在客户端解码播放上，在互联网中用户大都使用的是分辨率较低的场景，因此解码性能的应用比较少。但在传统领域例如教育、金融，对于清晰度的要求则会相对较高。对于使用较稳定的有线网络以及性能较好的 PC 设备的用户来说没有什么问题。但对于 4G 网络，低功耗移动端设备用户来说 高分辨率论是带宽还是解码性能都会可能成为瓶颈。此时多播依然是比较好的选择，我们可以通过转发低质量的码率供移动端用户享用，高质量的码流供 PC 端、有线网络用户享用。

同时音视频算法也是优化的一大重点，比如阿里直播的窄带高清技术、自研实时高性能视频编码 Ali S265