

第1章

Chapter 1

WebRTC 概述

随着网络基础设施日趋完善以及终端计算能力不断提升，实时通信技术已经渗透到各行各业，支撑着人们的日常生活。在 WebRTC 诞生之前，实时通信技术非常复杂，想获得核心的音视频编码及传输技术需要支付昂贵的专利授权费用。此外，将实时通信技术与业务结合也非常困难，并且很耗时，通常只有较大规模的公司才有能力实现。

WebRTC 的出现使实时通信技术得以广泛应用。WebRTC 制定、实现了一套统一且完整的实时通信标准，并将这套标准开源。这套标准包含了实时通信技术涉及的所有内容，使用这套标准，开发人员无须关注音视频编解码、网络连接、传输等底层技术细节，可以专注于构建业务逻辑，且这些底层技术是完全免费的。

WebRTC 统一了各平台的实时通信技术，大部分操作系统及浏览器都支持 WebRTC，无须安装任何插件，就可以在浏览器端发起实时视频通话。

WebRTC 技术最初为 Web 打造，随着 WebRTC 自身的演进，目前已经可以将其应用于各种应用程序。

随着 4G 的普及和 5G 技术的应用，实时音视频技术正在蓬勃发展。在互联网领域，花椒、映客等直播平台吸引了大量的用户；在教育领域，通过实时直播技术搭建的“空中课堂”惠及全球数亿学生；在医疗行业，随着电子处方单纳入医保，互联网看病、复诊正在兴起，地域之间医疗资源不均衡的问题被实时直播技术逐步消除。

WebRTC 1.0 规范发布以来，以 Chrome、Firefox 为代表的浏览器对 WebRTC 提供了全方面的支持，Safari 11 也开始对 WebRTC 提供支持。

1.1 WebRTC 的历史

WebRTC（Web Real-Time Communication）是一个谷歌开源项目，它提供了一套标准

2 ◆ WebRTC 技术详解：从 0 到 1 构建多人视频会议系统

API，使 Web 应用可以直接提供实时音视频通信功能，不再需要借助任何插件。原生通信过程采用 P2P 协议，数据直接在浏览器之间交互，理论上不需要服务器端的参与。

“为浏览器、移动平台、物联网设备提供一套用于开发功能丰富、高质量的实时音视频应用的通用协议”是 WebRTC 的使命。

WebRTC 的发展历史如下。

- 2010 年 5 月，谷歌收购视频会议软件公司 GIPS，该公司在 RTC 编码方面有深厚的技术积累。
- 2011 年 5 月，谷歌开源 WebRTC 项目。
- 2011 年 10 月，W3C 发布第一个 WebRTC 规范草案。
- 2014 年 7 月，谷歌发布视频会议产品 Hangouts，该产品使用了 WebRTC 技术。
- 2017 年 11 月，WebRTC 进入候选推荐标准（Candidate Recommendation, CR）阶段。

1.2 WebRTC 的技术架构

从技术实现的角度讲，在浏览器之间进行实时通信需要使用很多技术，如音视频编解码、网络连接管理、媒体数据实时传输等，还需要提供一组易用的 API 给开发者使用。这些技术组合在一起，就是 WebRTC 技术架构，如图 1-1 所示。

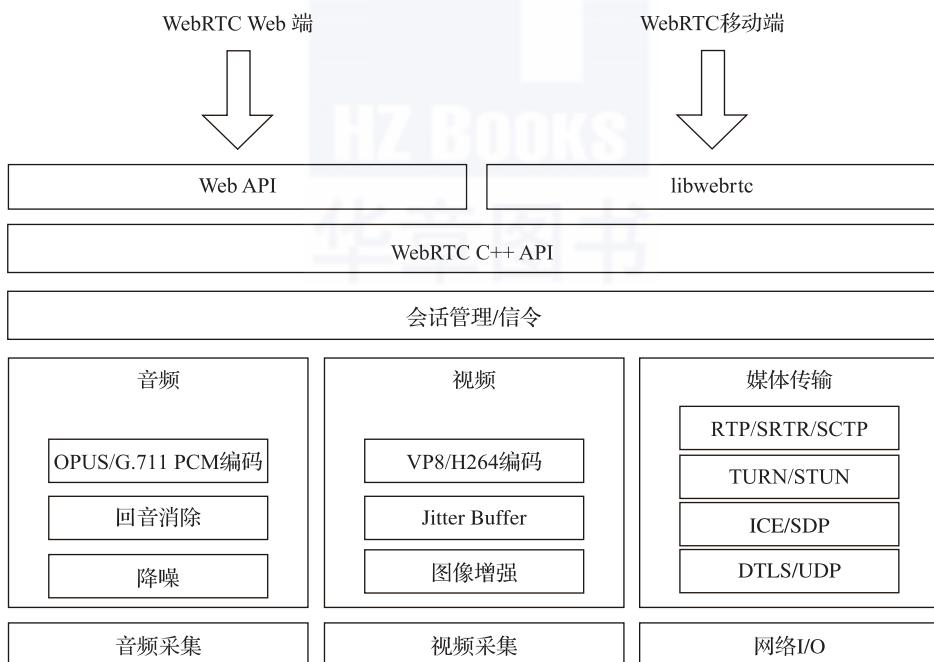


图 1-1 WebRTC 技术架构

WebRTC 技术架构的顶层分为两个部分。一部分是 Web API，一组 JavaScript 接口，由

W3C 维护，开发人员可以使用这些 API 在浏览器中创建实时通信应用程序。另一部分是适用于移动端及桌面开发的 libwebrtc，即使用 WebRTC C++ 源码在 Windows、Android、iOS 等平台编译后的开发包，开发人员可以使用这个开发包打造原生的 WebRTC 应用程序。

第二层是 WebRTC C++ API，它是 Web API 和 libwebrtc 的底层实现。该层包含了连接管理、连接设置、会话状态和数据传输的 API。基于这些 API，浏览器厂商可以方便地加入对 WebRTC 的支持。

WebRTC 规范里没有包含信令协议，这部分需要研发人员依据业务特点自行实现。

WebRTC 支持的音频编码格式有 OPUS 和 G.711，同时还在音频处理层实现了回音消除及降噪功能。WebRTC 支持的视频编码格式主要有 VP8 和 H264（还有部分浏览器支持 VP9 及 H265 格式），WebRTC 还实现了 Jitter Buffer 防抖动及图像增强等高级功能。

在媒体传输层，WebRTC 在 UDP 之上增加了 3 个协议。

- 数据包传输层安全性协议（DTLS）用于加密媒体数据和应用程序数据。
- 安全实时传输协议（SRTP）用于传输音频和视频流。
- 流控制传输协议（SCTP）用于传输应用程序数据。

WebRTC 借助 ICE 技术在端与端之间建立 P2P 连接，它提供了一系列 API，用于管理连接。WebRTC 还提供了摄像头、话筒、桌面等媒体采集 API，使用这些 API 可以定制媒体流。

我们将在后面的章节详细讨论 WebRTC 架构的主要技术（不包含 C++ 部分），并结合实例展示这些技术的应用。

1.3 WebRTC 的网络拓扑

WebRTC 规范主要介绍了使用 ICE 技术建立 P2P 的网络连接，即 Mesh 网络结构。在 WebRTC 技术的实际应用中，衍生出了媒体服务器的用法。

使用媒体服务器的场景，通常是因为 P2P 连接不可控，而使用媒体服务器可以对媒体流进行修改、分析、记录等 P2P 无法完成的操作。实际上，如果我们把媒体服务器看作 WebRTC 连接的另外一端，就很容易理解媒体服务器的工作原理了。媒体服务器是 WebRTC 在服务器端的实现，起到了桥梁的作用，用于连接多个 WebRTC 客户端，并增加了额外的媒体处理功能。通常根据提供的功能，将媒体服务器区分为 MCU 和 SFU。

1. Mesh 网络结构

Mesh 是 WebRTC 多方会话最简单的网络结构。在这种结构中，每个参与者都向其他所有参与者发送媒体流，同时接收其他所有参与者发送的媒体流。说这是最简单的网络结构，是因为它是 WebRTC 原生支持的，无须媒体服务器的参与。Mesh 网络结构如图 1-2 所示。

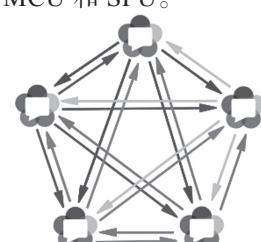


图 1-2 Mesh 网络结构

4 ◆ WebRTC 技术详解：从 0 到 1 构建多人视频会议系统

在 Mesh 网络结构中，每个参与者都以 P2P 的方式相互连接，数据交换基本不经过中央服务器（部分无法使用 P2P 的场景，会经过 TURN 服务器）。由于每个参与者都要为其他参与者提供独立的媒体流，因此需要 $N-1$ 个上行链路和 $N-1$ 个下行链路。众多上行和下行链路限制了参与人数，参与人过多会导致明显卡顿，通常只能支持 6 人以下的实时互动场景。

由于没有媒体服务器的参与，Mesh 网络结构难以对视频做额外的处理，不支持视频录制、视频转码、视频合流等操作。

2. MCU 网络结构

MCU (Multipoint Control Unit) 是一种传统的中心化网络结构，参与者仅与中心的 MCU 媒体服务器连接。MCU 媒体服务器合并所有参与者的视频流，生成一个包含所有参与者画面的视频流，参与者只需要拉取合流画面，MCU 网络结构如图 1-3 所示。

这种场景下，每个参与者只需要 1 个上行链路和 1 个下行链路。与 Mesh 网络结构相比，参与者所在的终端压力要小很多，可以支持更多人同时在线进行音视频通信，比较适合多人实时互动场景。但是 MCU 服务器负责所有视频编码、转码、解码、合流等复杂操作，服务器端压力较大，需要较高的配置。同时由于合流画面固定，界面布局也不够灵活。



图 1-3 MCU 网络结构

3. SFU 网络结构

在 SFU (Selective Forwarding Unit) 网络结构中，仍然有中心节点媒体服务器，但是中心节点只负责转发，不做合流、转码等资源开销较大的媒体处理工作，所以服务器的压力会小很多，服务器配置也不像 MCU 的要求那么高。每个参与者需要 1 个上行链路和 $N-1$ 个下行链路，带宽消耗低于 Mesh，但是高于 MCU。

我们可以将 SFU 服务器视为一个 WebRTC 参与方，它与其他所有参与方进行 1 对 1 的建立连接，并在其中起到桥梁的作用，同时转发各个参与者的媒体数据。SFU 服务器具备复制媒体数据的能力，能够将一个参与者的数据转发给多个参与者。SFU 服务器与 TURN 服务器不同，TURN 服务器仅仅是为 WebRTC 客户端提供的一种辅助数据转发通道，在无法使用 P2P 的情况下进行透明的数据转发，TURN 服务器不具备复制、转发媒体数据的能力。

SFU 对参与实时互动的人数也有一定的限制，适用于在线教学、大型会议等场景，其网络结构如图 1-4 所示。



图 1-4 SFU 网络结构

1.4 Simulcast 联播

在进行 WebRTC 多方视频会话时，参与人数较多，硬件设施、网络环境均有差异，这

种情况下如何确保会话质量呢？使用 MCU 时，这个问题相对简单一些。MCU 可以根据参与者的网络质量和设备能力，提供不同的清晰度和码率。但是随之而来的问题是服务器资源压力较大，难以支撑大规模并发，同时也显著增加了使用成本。

多人会话场景选择 SFU 网络结构是目前通用的做法。早期的 SFU 只是将媒体流从发送端转发给接收端，无法独立为不同参与者调整视频码率，其结果是发送者需要自行调整码率，以适应接收条件最差的参与者。而那些网络环境较好的参与者只能接收相同质量的媒体流，别无选择。

Simulcast 技术对 SFU 进行了优化，发送端可以同时发送多个不同质量的媒体流给接收端。SFU 能够依据参与者的网络质量，决定转发给参与者哪种质量的媒体流。

因为发送者需要发送多个不同质量的媒体流，所以会显著增加发送设备的载荷，同时占用发送者上行带宽资源。

1.5 可伸缩视频编码

可伸缩视频编码（Scalable Video Coding，SVC）是 Simulcast 的改进技术。它使用分层编码技术，发送端只需要发送一个独立的视频流给 SFU，SFU 根据不同的层，解码出不同质量的视频流，并发送给不同接收条件的参与者。

SVC 中多个层次的媒体流相互依赖，较高质量的媒体数据需要较低质量的媒体数据解码。SFU 接收到 SVC 编码的内容后，根据客户端的接收条件选择不同的编码层次，从而获得不同质量的媒体流。

如果媒体流包括多个不同分辨率的层，则称该编码具有空间可伸缩性；如果媒体流包含多个不同帧率的层，则称该编码具有时间可伸缩性；如果媒体流包含多个不同码率的层，则称该编码具有质量可伸缩性。

在编码空间、时间、质量均可伸缩的情况下，SFU 可以生成不同的视频流，以适应不同客户端的接收条件。

1.6 WebRTC 的兼容性

据 caniuse.com 统计，大部分浏览器都实现了对 WebRTC 的支持，各浏览器支持情况如下。

- Firefox 版本 22+
- Chrome 版本 23+
- Safari 版本 11+
- iOS Safari 版本 11+
- Edge 版本 15+

6 ◆ WebRTC 技术详解：从 0 到 1 构建多人视频会议系统

- Opera 版本 18+
- Android Browser 版本 81+
- Chrome for Android 版本 84+
- Firefox for Android 版本 68+
- IE 不支持

Android 和 iOS 原生应用都支持 WebRTC，可以使用原生 SDK 开发跨平台的 WebRTC 应用。

Android WebView 自 36 版本之后，提供了对 WebRTC 的支持，这意味着可以使用 WebRTC API 开发 Android 混合 App。注意，一些手机厂商对部分 Android 版本里的 WebView 进行了裁剪，导致不能使用 WebRTC，这时候下载并安装最新的 WebView 即可。

iOS WebView 目前还不支持 WebRTC，但是可以使用 cordova 的插件 cordova-plugin-iosrtc 在混合 App 中使用 WebRTC。

WebRTC 目前处于活跃开发阶段，各个浏览器的实现程度不一样。为了解决兼容性的问题，谷歌提供了 adapter.js 库。

在 GitHub 上可以下载最新版本的 adapter.js 库，地址如下所示。

<https://github.com/webrtc/adapter/tree/master/release>

将下载的文件放到 Web 服务器根目录，在 Web 应用中引用。

```
<script src="adapter.js"></script>
```

1.7 其他直播技术

在 WebRTC 流行之前，低延迟的直播技术就已经普及了。这些技术一般包括用于互联网直播的 RTMP 协议、用于监控领域的 RTSP 协议，还有一些较新的协议，如 SRT 和 QUIC。WebRTC 由多个传输协议构成，实际上是一套实时通信技术的解决方案，而其他直播技术则大多以单独协议的形式存在。

1. 实时消息传输协议

实时消息传输协议（Real Time Messaging Protocol，RTMP）基于 TCP，最初由 Macromedia 公司开发，并于 2005 年被 Adobe 收购。它包括 RTMP 基本协议及 RTMPT、RTMPS、RTMPE 等多个变种。RTMP 是一种实时数据通信网络协议，主要用来在 Flash/AIR 平台和支持 RTMP 协议的流媒体 / 服务器之间进行音视频和数据通信。RTMP 与 HTTP 一样，都属于 TCP/IP 四层模型的应用层。

RTMP 协议的应用非常广泛，至今仍是最常用的直播传输协议之一，大多数流媒体平台和软件都支持 RTMP 协议。

传统的 RTMP 延迟较高，通常延迟 5~20s，经过优化，延迟可以降低到 2~3s。如

果想再进一步降低延迟时间，则需要改造 RTMP 协议，将其底层 TCP 协议改为 UDP，如微信小程序使用了基于 UDP 协议的 RTMP，其延迟可以降低到毫秒级。

RTMP 协议的优点如下。

- 主流的 CDN 厂商都支持 RTMP 协议。
- 协议简单，在各平台均可实现。

这些优势使得基于 RTMP 协议的应用程序可以获得良好的基础设施支撑，而且开发及使用成本可控。

RTMP 协议的缺点如下。

- 基于 TCP 导致传输延迟高，在弱网环境下问题尤为明显。
- 由于 Flash 技术即将被浏览器淘汰，主流浏览器都不支持推送 RTMP 协议。

2. RTSP 实时流协议

实时流协议（Real Time Streaming Protocol，RTSP）是 TCP/IP 协议体系中的一个应用层协议。该协议定义了一对多应用程序该如何有效地通过 IP 网络传送多媒体数据。RTSP 的协议层次位于 RTP 和 RTCP 之上，支持使用 TCP 或 UDP 传输数据。

RTSP 中所有的操作都是通过服务器和客户端的消息应答机制完成的，其中消息包括请求和应答两种。RTSP 是对称的协议，客户端和服务器都可以发送和回应请求。RTSP 是一个基于文本的协议，它使用 UTF -8 编码（RFC2279）和 ISO10646 字符序列，采用 RFC882 定义通用消息的格式，每行语句以 CRLF 结束。

RTSP 建立并控制一个或多个与时间同步的连续流媒体，负责定义具体的媒体控制信息、操作方法、状态码以及描述与 RTP 之间的交互操作，如播放、录制、暂停等。

RTSP 不负责数据传输，这部分工作由 RTP/RTCP 协议完成。从这个层面来看，RTSP 和 WebRTC 是类似的，都使用了 RTP/RTCP 协议完成媒体数据传输，但是 WebRTC 的功能更为丰富，技术栈也更为完善。

3. 安全可靠传输

安全可靠传输（Secure Reliable Transport，SRT）是流媒体前沿的后起之秀，支持在各种网络环境下进行低延迟、高质量的音视频传输。SRT 协议可以为媒体数据提供高达 256 位的高级加密标准（AES）加密。

为了促进 SRT 技术的发展，SRT 开放了源代码，开源社区成立了 SRT 联盟，包括众多行业领导者及开发人员。目前已经集成了 SRT 技术的流行软件有 OBS Studio、GStreamer 和 VLC。

SRT 被称为“卫星替代技术”，其低成本和实时通信能力为直播公司提供了一种替代卫星技术的方案。

SRT 的优势如下。

- 可传输低延迟、高质量的视频和音频。
- 可在 SRT 源（编码器）和 SRT 目标（解码器）之间轻松穿越防火墙。

8 ◆ WebRTC 技术详解：从 0 到 1 构建多人视频会议系统

□ 控制延迟以适应不断变化的网络状况。

□ 使用多达 256 位 AES 加密的安全直播。

SRT 基于 UDP 协议，在 SRT 源（编码器）和 SRT 目标（解码器）之间建立用于控制和恢复数据包的专用通信链路，目标可以是服务器、CDN 或其他 SRT 设备。SRT 使用自己的拥塞控制算法，该算法可以自动适应网络环境，并随着网络波动进行实时调整。

SRT 和 WebRTC 都依赖于增强的 UDP 协议，能够提供实时通信的能力。但是 WebRTC 的优势在于它是一种基于浏览器的协议，可以在任何主流浏览器中使用，无须借助插件或硬件。

4. QUIC 协议

快速 UDP 互联网连接（Quick UDP Internet Connection, QUIC）协议是谷歌制定的一种基于 UDP 的低时延互联网传输层协议。我们知道，TCP/IP 协议簇是互联网的基础协议，其中传输层协议包括 TCP 和 UDP。与 TCP 协议相比，UDP 更为轻量，错误校验要少得多。这意味着虽然 UDP 的传输效率更高，但是传输可靠性不如 TCP。通常游戏、流媒体等应用采用 UDP，而网页、邮件、远程登录等大部分应用采用 TCP。

QUIC 同时具备 TCP 和 UDP 的优势，并弥补了它们的短板，很好地解决了网络连接安全性、可靠性和低延迟的问题。QUIC 基于 UDP 传输，当客户端第一次连接服务器时，QUIC 只需要 1 个往返时间（Round-Trip Time, RTT）的延迟就可以建立安全可靠的连接，相比于 TCP+TLS 建立连接需要 1~3 个 RTT，QUIC 要更加快捷。之后客户端可以在本地缓存加密的认证信息，再次与服务器建立连接时可以实现 0~1 个 RTT 的连接建立延迟。QUIC 借用了 HTTP/2 协议的多路复用功能（Multiplexing），但由于 QUIC 基于 UDP，所以避免了 HTTP/2 线头阻塞（Head-of-Line Blocking）的问题。因为 QUIC 运行在用户域而不是系统内核，所以 QUIC 协议可以快速更新和部署到生产环境中，从而解决了 TCP 协议部署及更新较为困难的问题。

2016 年 11 月，国际互联网工程任务组召开了第一次 QUIC 工作组会议，这意味着 QUIC 开始了它的标准化过程，成为新一代传输层协议。

由于 QUIC 工作在传输层，与 WebRTC 没有竞争关系，所以实际上 WebRTC 也可以使用 QUIC 作为底层传输协议，在新的 WebRTC 规范草案中已经提供了对 QUIC 的支持。

1.8 统一计划与 Plan B

统一计划（Unified Plan）是用于在会话描述协议（SDP）中发送多个媒体源的 IETF 草案。谷歌在 2013 年于 Chrome 浏览器中实施了 Plan B。Plan B 实际上是 Unified Plan 的一个变种。谷歌后续又在 Chrome 浏览器中提供了对 Unified Plan 的支持。作为过渡方案，目前 Chrome 浏览器同时支持 Plan B 和 Unified Plan，将来 Chrome 可能会取消对 Plan B 的支持。

在 Plan B 中，SDP 协议为同一类的媒体使用一个“m =”字段。如果同一类媒体包括

多个不同的媒体轨道，比如同时包含摄像头和屏幕共享的媒体轨道，则在“m=”字段中列出多个“a=ssrc”信息，以示区分。

而使用 Unified Plan 时，每个媒体轨道都分配单独的“m=”字段。如果使用多个媒体轨道，则会创建多个“m=”字段。

由于处理多个媒体轨道的方式不同，如果使用同一媒体类型的多个媒体轨道，则 Unified Plan 和 Plan B 是不兼容的，对于同一媒体类型只有一个轨道的情况，则会保持兼容。

如果 Unified Plan 客户端收到 Plan B 客户端生成的提案（offer），则 Unified Plan 客户端在调用 setRemoteDescription() 时报错。同样，如果 Plan B 客户端收到 Unified Plan 客户端生成的提案，则它只能在第一个媒体轨道触发 ontrack 事件，并丢弃其他相同类型的媒体轨道。

Chrome M69 版本开始支持 Unified Plan，但是默认支持的仍然是 Plan B。Chrome 在 WebRTC 的媒体管理接口 RTCPeerConnection 中添加了一项新的枚举类型 SdpSemantics，用于在两种计划之间进行切换。

```
enum SdpSemantics { "plan-b", "unified-plan" };
partial dictionary RTCConfiguration { SdpSemantics sdpSemantics; }
```

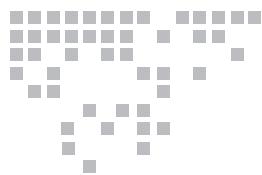
在创建对等连接时，使用如下命令启用 unified-plan。

```
let peer = new RTCPeerConnection ({ sdpSemantics : "unified-plan" });
```

Chrome 从 M72 版本开始改为默认支持 Unified Plan。在上述代码中，将 unified-plan 改为 plan-b 则可以切换回对 Plan B 的支持。

1.9 本章小结

本章对 WebRTC 的背景及技术进行了简单的介绍，希望通过这些内容，读者能够对 WebRTC 有个初步的认识。WebRTC 是独特的，同时也是非常复杂的。可以说，它是目前构建实时通信系统的最佳技术。从第 2 章开始，我们来全面认识 WebRTC 技术，开始一场实时通信技术之旅。



Chapter 2

第2章

本地媒体

一个实时音视频通话过程，通常包括媒体采集、编码、传输、解码、播放等环节，媒体采集是控制会话质量的第一步，决定了媒体源的内容和形式。本章将围绕媒体采集，介绍相关的知识点，为读者展现一个完整的 Web 本地媒体管理流程。

WebRTC 作为 Web 技术的一种，其应用过程离不开与其他 Web 技术的结合，如 WebRTC 结合 canvas 技术能够改变视频源内容、实现虚拟背景的效果；结合媒体录制 API，能够实现实时录制与回放等。本章也会对这些技术进行详细介绍，让视频直播应用呈现更加丰富的内容。

2.1 媒体流

在 WebRTC 的众多技术中，我们首先介绍媒体流（MediaStream），因为媒体流应用在 WebRTC 技术的各个方面，理解了媒体流的相关概念和使用方法，才能更好地展开介绍其他技术。

媒体流是信息的载体，代表了一个媒体设备的内容流。媒体流可以被采集、传输和播放，通常一个媒体流包含多个媒体轨道，如音频轨道、视频轨道。

媒体流使用 MediaStream 接口来管理，通常获取媒体流的方式有如下几种。

- ❑ 从摄像头或者话筒获取流对象。
- ❑ 从屏幕共享获取流对象。
- ❑ 从 canvas (HTMLCanvasElement) 内容中获取流对象。
- ❑ 从媒体元素 (HTMLMediaElement) 获取流对象。

上述方法获取的媒体流都可以通过 WebRTC 进行传输，并在多个对等端之间共享。

MediaStream 的定义如代码清单 2-1 所示。

代码清单2-1 MediaStream的定义

```
interface MediaStream : EventTarget {  
    constructor();  
    constructor(MediaStream stream);  
    constructor(sequence<MediaStreamTrack> tracks);  
    readonly attribute DOMString id;  
    sequence<MediaStreamTrack> getAudioTracks();  
    sequence<MediaStreamTrack> getVideoTracks();  
    sequence<MediaStreamTrack> getTracks();  
    MediaStreamTrack? getTrackById(DOMString trackId);  
    void addTrack(MediaStreamTrack track);  
    void removeTrack(MediaStreamTrack track);  
    MediaStream clone();  
    readonly attribute boolean active;  
    attribute EventHandler onaddtrack;  
    attribute EventHandler onremovetrack;  
};
```

我们将在本节详细讨论媒体流的构造函数、属性、方法和事件。

2.1.1 构造媒体流

构造函数 `MediaStream()` 可以创建并返回一个新的 `MediaStream` 对象，可以创建一个空的媒体流或者复制现有媒体流，也可以创建包含多个指定轨道的媒体流，命令如下。

```
// 创建一个空媒体流  
newStream = new MediaStream();  
// 从stream中复制媒体流  
newStream = new MediaStream(stream);  
// 创建包含多个指定轨道的媒体流  
newStream = new MediaStream(tracks[]);
```

2.1.2 MediaStream 属性

1. active 只读

返回 `MediaStream` 的状态，类型为布尔，`true` 表示处于活跃状态，`false` 表示处于不活跃状态。

2. id 只读

返回 `MediaStream` 的 UUID，类型为字符串，长度为 36 个字符。

2.1.3 MediaStream 方法

1. addTrack() 方法

该方法向媒体流中加入新的媒体轨道。

12 ◆ WebRTC 技术详解：从 0 到 1 构建多人视频会议系统

```
stream.addTrack(track);
```

- 参数：Track，媒体轨道，类型为 MediaStreamTrack。
- 返回值：无。

2. clone() 方法

返回当前媒体流的副本，副本具有不同且唯一的标识。

```
const newstream = stream.clone();
// sameId为false
const sameId = newstream.id === stream.id? true : false
```

- 参数：无。
- 返回值：一个新的媒体流对象。

3. getAudioTracks() 方法

返回媒体种类为 audio 的媒体轨道对象数组，数组成员类型为 MediaStreamTrack。

注意，数组的顺序是不确定的，每次调用都可能不同。

```
const mediaStreamTracks = mediaStream.getAudioTracks()
```

- 参数：无。
- 返回值：mediaStreamTracks，媒体轨道对象数组，如果当前媒体流没有音频轨道，则返回数组为空。

代码清单 2-2 使用 getUserMedia() 方法获取包含视频及音频轨道的媒体流，如果调用成功，则将媒体流附加到 <video> 元素，然后设置计时器，5s 后调用 getAudioTracks() 方法获取所有音频轨道，最后停止播放第一个音频轨道。

代码清单 2-2 getAudioTracks()方法示例

```
navigator.mediaDevices.getUserMedia({audio: true, video: true})
.then(mediaStream => {
  document.querySelector('video').srcObject = mediaStream;
  // 5s后，停止播放第一个音频轨道
  setTimeout(() => {
    const tracks = mediaStream.getAudioTracks()
    tracks[0].stop()
  }, 5000)
})
```

4. getVideoTracks() 方法

返回 kind 属性值为 video 的媒体轨道对象数组，媒体轨道对象类型为 MediaStream Track。

注意，对象在数组中的顺序是不确定的，每次调用都可能不同。

```
const mediaStreamTracks = mediaStream.getVideoTracks()
```

- 参数：无。

- 返回值：mediaStreamTracks 是媒体轨道对象数组。如果当前媒体流没有视频轨道，则返回数组为空。

代码清单 2-3 调用 getUserMedia() 方法获取视频流，如果调用成功，则将媒体流附加到 <video> 元素，之后获取第一个视频轨道并从视频轨道截取图片。

代码清单2-3 getVideoTracks()方法示例

```
navigator.mediaDevices.getUserMedia({video: true})
.then(mediaStream => {
  document.querySelector('video').srcObject = mediaStream;
  const track = mediaStream.getVideoTracks()[0];
  // 截取图片
  const imageCapture = new ImageCapture(track);
  return imageCapture;
})
```

5. getTrackById() 方法

返回指定 ID 的轨道对象。如果未提供参数，或者未匹配 ID 值，则返回 null；如果存在多个相同 ID 的轨道，该方法返回匹配到的第一个轨道。

```
const track = MediaStream.getTrackById(id);
```

- 参数：id，类型为字符串。
- 返回值：如果输入参数 id 与 MediaStreamTrack.id 匹配，则返回相应的 MediaStreamTrack 对象，否则返回 null。

代码清单 2-4 获取指定 ID 的媒体轨道并应用约束，将音量调整到 0.5。

代码清单2-4 getTrackById()方法示例

```
stream.getTrackById("primary-audio-track").applyConstraints({ volume: 0.5 });
```

6. getTracks() 方法

返回所有媒体轨道对象数组，包括所有视频及音频轨道。数组中对象的顺序不确定，每次调用都有可能不同。

```
const mediaStreamTracks = mediaStream.getTracks()
```

- 参数：无。
- 返回值：媒体轨道对象数组。

代码清单 2-5 使用 getUserMedia() 方法获取包含视频轨道的流，如果调用成功，则将流附加到 <video> 元素，然后设置计时器，5s 后获取所有媒体轨道，并停止播放第一个媒体轨道（即视频轨道）。

代码清单2-5 getTracks()方法示例

```
navigator.mediaDevices.getUserMedia({audio: false, video: true})
```

```
.then(mediaStream => {
  document.querySelector('video').srcObject = mediaStream;
  // 5s后，停止播放第一个媒体轨道
  setTimeout(() => {
    const tracks = mediaStream.getTracks()
    tracks[0].stop()
  }, 5000)
})
```

2.1.4 MediaStream 事件

1. addtrack 事件

当有新的媒体轨道（MediaStreamTrack）加入时触发该事件，对应事件句柄 `onaddtrack`。

注意，只有在如下情况下，才会触发该事件，主动调用 `MediaStream.addTrack()` 方法则不会触发。

- `RTCPeerConnection` 重新协商。
- `HTMLMediaElement.captureStream()` 返回新的媒体轨道。

如代码清单 2-6 所示，当有新的媒体轨道添加到媒体流时，显示新增媒体轨道的种类和标签。

代码清单 2-6 `onaddtrack`示例

```
// event 类型为MediaStreamTrackEvent
// event.track 类型为MediaStreamTrack
stream.onaddtrack = (event) => {
  let trackList = document.getElementById("tracks");
  let label = document.createElement("li");

  label.innerHTML = event.track.kind + "：" + event.track.label;
  trackList.appendChild(label);
};
```

此外，也可以使用 `addEventListener()` 方法监听事件 `addtrack`。

2. removetrack 事件

当有媒体轨道被移除时触发该事件，对应事件句柄 `onremovetrack`。

注意，只有在如下情况下才会触发该事件，主动调用 `MediaStream.removeTrack()` 方法则不会触发。

- `RTCPeerConnection` 重新协商。
- `HTMLMediaElement.captureStream()` 返回新的媒体轨道。

如代码清单 2-7 所示，当从媒体流中删除媒体轨道时，记录该媒体轨道信息。

代码清单 2-7 `onremovetrack`示例

```
// event 类型为MediaStreamTrackEvent
```

```
// event.track类型为MediaStreamTrack
stream.onremovetrack = (event) => {
  let trackList = document.getElementById("tracks");
  let label = document.createElement("li");

  label.innerHTML = "Removed: " + event.track.kind + ": " + event.track.label;
  trackList.appendChild(label);
};
```

此外，也可以使用 addEventListener() 方法监听事件 removetrack。

2.2 媒体轨道

我们已经在多个方法中接触到了媒体轨道，媒体流由媒体轨道构成，而媒体轨道则代表着一个能够提供媒体服务的媒体，如音频、视频等。

媒体轨道使用 MediaStreamTrack 接口管理，MediaStreamTrack 的定义如代码清单 2-8 所示。

代码清单2-8 MediaStreamTrack的定义

```
interface MediaStreamTrack : EventTarget {
  readonly attribute DOMString kind;
  readonly attribute DOMString id;
  readonly attribute DOMString label;
  attribute boolean enabled;
  readonly attribute boolean muted;
  attribute EventHandler onmute;
  attribute EventHandler onunmute;
  readonly attribute MediaStreamTrackState readyState;
  attribute EventHandler onended;
  MediaStreamTrack clone();
  void stop();
  MediaTrackCapabilities getCapabilities();
  MediaTrackConstraints getConstraints();
  MediaTrackSettings getSettings();
  Promise<void> applyConstraints(optional MediaTrackConstraints
    constraints = {});
};
```

2.2.1 MediaStreamTrack 属性

1. enabled

返回 MediaStreamTrack 的有效状态，类型为布尔值，值为 true 表示轨道有效，可以被渲染；值为 false 表示轨道失效，被渲染时将会出现静音或黑屏。如果媒体轨道连接中断，enabled 值仍然可以被改变，但不会生效。

设置 enabled 值为 false 可以实现静音效果，与 mute 方法相同。

代码清单 2-9 实现了按钮的单击事件，在事件处理函数中控制媒体轨道的暂停和播放。

代码清单2-9 enabled示例

```
pauseButton.onclick = function(evt) {
    const newState = !myAudioTrack.enabled;

    pauseButton.innerHTML = newState ? "Pause" : "Play";
    myAudioTrack.enabled = newState;
}
```

2. id 只读

返回 MediaStreamTrack 的 UUID 值，类型为字符串。

```
const id = track.id
```

3. kind 只读

返回 MediaStreamTrack 的内容种类，类型为字符串，返回 audio 表示轨道内容种类是音频，返回 video 表示轨道内容种类是视频。

代码清单 2-10 对 kind 的种类进行判断并使用 console.log 打印。

代码清单2-10 kind示例

```
const type = track.kind;
if ( type === 'video' ) {
    console.log('video track');
} else if ( type === 'audio' ) {
    console.log('audio track');
}
```

4. label 只读

返回 MediaStreamTrack 的标签，类型为字符串，表示媒体轨道的来源，比如 internal microphone。

label 的值可以为空，并且在没有媒体源与媒体轨道连接的情况下会一直为空。当轨道与它的源分离时，label 的值不会改变。

```
const label = track.label
```

5. muted 只读

返回 MediaStreamTrack 是否处于静音状态，类型为布尔，值为 true 表示轨道静音，值为 false 表示轨道未静音。处于静音状态的媒体轨道不能提供媒体数据，当视频轨道处于静音状态时，则会表现为黑屏。

代码清单 2-11 对媒体轨道数组进行遍历，并统计处于静音状态的媒体轨道数目。

代码清单2-11 muted示例

```
let mutedCount = 0;
```

```
trackList.forEach((track) => {
  if (track.muted) {
    mutedCount += 1;
  }
});
```

6. readyState 只读

返回 MediaStreamTrack 的就绪状态，可能的取值如下。

- live：表示输入媒体源已经连接，可以正常提供媒体数据。
- ended：表示输入媒体源处于结束状态，不能再提供新的媒体数据。

```
const state = track.readyState
```

2.2.2 MediaStreamTrack 方法

1. applyConstraints() 方法

该方法为媒体轨道指定约束条件，如可以指定帧率、分辨率、回音消除等。

可以根据需要使用约束定制媒体流，比如在采集清晰度高的视频时，可以将帧率降低一些，这样就不至于占用太大的网络带宽。

```
const appliedPromise = track.applyConstraints([constraints])
```

- 参数：constraints，可选参数，一个类型为 MediaTrackConstraints 的对象，包含了准备应用到媒体轨道里的所有约束需求，未在需求中指定的约束将使用默认值。如果 constraints 为空，则会清除当前轨道所有的自定义效果，全部使用默认值。
- 返回值：appliedPromise 是一个 Promise 值，决议失败时返回 MediaStreamError。当指定的约束太严格时，可能会导致该方法调用失败。

如代码清单 2-12 所示，使用 getUserMedia() 方法获取视频流，并针对第一个视频轨道应用约束条件，约束指定了视频的宽、高和宽高比。

代码清单 2-12 applyConstraints()方法示例

```
const constraints = {
  width: {min: 640, ideal: 1280},
  height: {min: 480, ideal: 720},
  aspectRatio: {ideal: 1.777777777778}
};

navigator.mediaDevices.getUserMedia({ video: true })
.then(mediaStream => {
  const track = mediaStream.getVideoTracks()[0];
  track.applyConstraints(constraints)
  .then(() => {
    // 成功应用了约束条件
    console.log('succesed!');
  });
});
```

```
})
.catch(e => {
  // 不能满足约束条件
  console.log('applyConstraints error, error name: ' + e.name);
});
});
```

2. clone() 方法

获取媒体轨道的副本，新的轨道具有不同的 ID 值。

```
const newTrack = track.clone()
```

3. getCapabilities() 方法

获取媒体轨道的能力集。

```
const capabilities = track.getCapabilities()
```

□ 参数：无。

□ 返回值：MediaTrackCapabilities 对象，描述了媒体轨道的能力信息。

MediaTrackCapabilities 的定义如代码清单 2-13 所示。

代码清单 2-13 MediaTrackCapabilities 的定义

```
dictionary MediaTrackCapabilities {
  ULONGRange width;
  ULONGRange height;
  DoubleRange aspectRatio;
  DoubleRange frameRate;
  sequence<DOMString> facingMode;
  sequence<DOMString> resizeMode;
  ULONGRange sampleRate;
  ULONGRange sampleSize;
  sequence<boolean> echoCancellation;
  sequence<boolean> autoGainControl;
  sequence<boolean> noiseSuppression;
  DoubleRange latency;
  ULONGRange channelCount;
  DOMString deviceId;
  DOMString groupId;
};
```

代码清单 2-14 获取媒体轨道的能力集，并将媒体能力输出到日志。

代码清单 2-14 getCapabilities() 方法示例

```
const capabilities = track.getCapabilities()
// 遍历capabilities，将媒体能力输出到日志
Object.keys(capabilities).forEach(value => {
  console.log(value + '=>' + capabilities[value]);
})
```

媒体能力、约束、设定值的概念相近，有区别，也有联系，我们将在 2.3 节详细讨论。

4. getConstraints() 方法

获取最近一次调用 applyConstraints() 传入的自定义约束。

```
const constraints = track.getConstraints()
```

- 参数：无。
- 返回值：MediaTrackConstraints 对象，包含了媒体轨道的约束集。

代码清单 2-15 实现了摄像头的切换。

代码清单2-15 getConstraints()方法示例

```
function switchCameras(track, camera) {  
    const constraints = track.getConstraints();  
    constraints.facingMode = camera;  
    track.applyConstraints(constraints);  
}
```

5. getSettings() 方法

获取媒体轨道约束的当前值。

```
const settings = track.getSettings()
```

- 输入：无。
- 返回值：MediaTrackSettings 对象，包含媒体轨道约束的当前值。
MediaTrackSettings 的定义如代码清单 2-16 所示。

代码清单2-16 MediaTrackSettings的定义

```
dictionary MediaTrackSettings {  
    long width;  
    long height;  
    double aspectRatio;  
    double frameRate;  
    DOMString facingMode;  
    DOMString resizeMode;  
    long sampleRate;  
    long sampleSize;  
    boolean echoCancellation;  
    boolean autoGainControl;  
    boolean noiseSuppression;  
    double latency;  
    long channelCount;  
    DOMString deviceId;  
    DOMString groupId;  
};
```

获取当前使用的摄像头，如代码清单 2-17 所示。

代码清单2-17 getSettings()方法示例

```
function whichCamera(track) {  
    return track.getSettings().facingMode;  
}
```

6. stop() 方法

用于停止播放当前媒体轨道。如果多个媒体轨道与同一个源相连，停止某个轨道不会导致源终止，只有当所有相连媒体轨道都停止时，媒体源才会停止。调用该方法后，属性 readyState 将被设置为 ended。

```
track.stop()
```

- 输入：无。
- 返回值：无。

代码清单 2-18 从 video 元素的 srcObject 属性获取媒体流，然后调用 getTracks() 方法获取所有媒体轨道，遍历媒体轨道数组并调用 stop() 方法逐一停止播放。

代码清单2-18 stop()方法示例

```
function stopStreamedVideo(videoElem) {  
    const stream = videoElem.srcObject;  
    const tracks = stream.getTracks();  
    tracks.forEach(track => {  
        track.stop();  
    });  
    videoElem.srcObject = null;  
}
```

2.2.3 MediaStreamTrack 事件

1. ended 事件

当媒体轨道结束时，触发该事件，此时属性 readyState 取值变为 ended，对应事件句柄 onended。

- 以下情况会触发该事件。
 - 媒体源没有更多数据。
 - 用户注销了相关媒体设备的访问权限。
 - 媒体源设备被移除。
 - 当媒体源来自对等端时，意味着对等端永久性终止了数据发送。

代码清单 2-19 使用 addEventListener 监听 ended 事件，并在触发该事件时改变对应的图标。

代码清单2-19 ended事件示例

```
track.addEventListener('ended', () => {
```

```
let statusElem = document.getElementById("status-icon");
statusElem.src = "/images/stopped-icon.png";
})
```

为 onended 事件句柄设置处理函数也可以达到同样的目的，如代码清单 2-20 所示。

代码清单2-20 onended事件句柄示例

```
track.onended = () => {
  let statusElem = document.getElementById("status-icon");
  statusElem.src = "/images/stopped-icon.png";
}
```

2. mute 事件

当属性 mute 被设置为 true 时触发该事件，表明媒体轨道暂时不能提供数据，对应事件句柄 onmute。

代码清单 2-21 使用 addEventListener 监听 mute 事件，在事件触发时改变指定 ID 元素的背景色。

代码清单2-21 mute事件示例

```
musicTrack.addEventListener("mute", event => {
  document.getElementById("timeline-widget").style.backgroundColor = "#aaa";
}, false);
```

为 onmute 事件句柄设置处理函数也可以达到同样的目的，如代码清单 2-22 所示。

代码清单2-22 onmute事件句柄示例

```
musicTrack.onmute = (event) => {
  document.getElementById("timeline-widget").style.backgroundColor = "#aaa";
}
```

3. unmute 事件

当取消静音状态时触发该事件，表明媒体轨道可以正常提供数据，对应事件句柄 onunmute。

代码清单 2-23 使用 addEventListener 监听 unmute 事件，在事件触发时改变指定 ID 元素的背景色。

代码清单2-23 unmute事件示例

```
musicTrack.addEventListener("unmute", event => {
  document.getElementById("timeline-widget").style.backgroundColor = "#fff";
}, false);
```

为 onunmute 事件句柄设置处理函数也可以达到同样的目的，如代码清单 2-24 所示。

代码清单2-24 onunmute事件句柄示例

```
musicTrack.onunmute = (event) => {
```

```
document.getElementById("timeline-widget").style.backgroundColor = "#fff";
}
```

2.3 媒体约束

应用媒体约束是较为复杂且灵活的一部分，我们在本节进行详细讨论。

媒体约束、媒体能力、媒体约束设定值理解起来容易混淆。媒体约束是指媒体某一项技术特性，如分辨率、帧率等；媒体能力是当前设备能够支持的某个约束的量化指标，如帧率最高 30；媒体约束设定值是指包含了浏览器默认设定值的所有媒体约束。

通常，我们使用如下方法处理媒体约束、媒体能力和媒体约束设定值。

- `MediaDevices.getSupportedConstraints()`：获取当前浏览器支持的约束数组。
- `MediaStreamTrack.getCapabilities()`：有了支持的约束数组，使用该方法获取这些约束的取值范围。
- `MediaStreamTrack.applyConstraints()`：根据应用程序的需要，调用该方法为约束指定自定义的值。
- `MediaStreamTrack.getConstraints()`：获取上述 `applyConstraints()` 方法传入的值。
- `MediaStreamTrack.getSettings()`：获取当前轨道上所有约束的实际值。

由于浏览器对约束的支持情况不同，下文介绍的约束并不一定是所有浏览器都支持。所以在使用约束前，需要先使用方法 `MediaDevices.getSupportedConstraints()` 进行检查。

代码清单 2-25 获取当前浏览器支持的所有约束对象，并检查是否支持 iso 约束。

代码清单 2-25 检查iso约束示例

```
let supportedConstraints = navigator.mediaDevices.getSupportedConstraints();
if ( supportedConstraints && supportedConstraints.iso ) {
    // 存在名为iso的约束
}
```

2.3.1 约束类型

媒体约束包括媒体流约束（`MediaStreamConstraints`）和媒体轨道约束（`MediaTrackConstraints`）。

`MediaStreamConstraints` 的定义如代码清单 2-26 所示。

代码清单 2-26 `MediaStreamConstraints` 的定义

```
dictionary MediaStreamConstraints {
    (boolean or MediaTrackConstraints) video = false;
    (boolean or MediaTrackConstraints) audio = false;
};
```

`MediaStreamConstraints` 属性说明如表 2-1 所示。

表 2-1 MediaStreamConstraints 属性说明

约 束	类 型	描 述	说 明
audio	Boolean MediaTrackConstraints	类型为布尔值 (true/false) 时, 表示是否请求音频; 类型为 MediaTrackConstraints 对象时, 表示具体的约束参数	如: {audio: false}
video	Boolean MediaTrackConstraints	类型为布尔值 (true/false) 时, 表示是否请求视频; 类型为 MediaTrackConstraints 对象时, 表示具体的约束参数	如: {video: true}

MediaTrackConstraints 的定义如代码清单 2-27 所示。

代码清单 2-27 MediaTrackConstraints 的定义

```
dictionary MediaTrackConstraints : MediaTrackConstraintSet {  
    sequence<MediaTrackConstraintSet> advanced;  
};  
  
dictionary MediaTrackConstraintSet {  
    ConstrainULong width;  
    ConstrainULong height;  
    ConstrainDouble aspectRatio;  
    ConstrainDouble frameRate;  
    ConstrainDOMString facingMode;  
    ConstrainDOMString resizeMode;  
    ConstrainULong sampleRate;  
    ConstrainULong sampleSize;  
    ConstrainBoolean echoCancellation;  
    ConstrainBoolean autoGainControl;  
    ConstrainBoolean noiseSuppression;  
    ConstrainDouble latency;  
    ConstrainULong channelCount;  
    ConstrainDOMString deviceId;  
    ConstrainDOMString groupId;  
};
```

2.3.2 数据类型与用法

通常为约束指定值时, 既可以指定具体值也可以指定一个对象, 在对象中包含 exact 或 ideal 属性, 用来告诉浏览器该约束的确切值和理想值; 有的约束还支持在对象中指定最小值 (min) 或最大值 (max)。

以视频 width 为例, 其数据类型是 ConstrainULong, 代码清单 2-28 中展示了类型为 ConstrainULong 时, 为 width 指定约束条件的 3 种方法。

代码清单 2-28 为 width 指定约束条件的 3 种方法

```
// 方法1: 直接指定值  
const constraints = {  
    width: 1280,
```

```
height: 720,  
aspectRatio: 3/2  
};  
// 方法2：指定最小值和理想值  
const constraints = {  
    frameRate: {min: 20},  
    width: {min: 640, ideal: 1280},  
    height: {min: 480, ideal: 720},  
    aspectRatio: 3/2  
};  
// 方法3：指定最小值、理想值和最大值  
const constraints = {  
    width: {min: 320, ideal: 1280, max: 1920},  
    height: {min: 240, ideal: 720, max: 1080},  
};
```

可以看到，约束的使用非常灵活，每种用法表达的含义不同，而用法与其数据类型又有直接的关系，为了更好地理解约束并掌握约束的用法，我们先介绍约束使用的数据类型，如表 2-2 所示。

表 2-2 约束相关的数据类型

类 型	说 明	示 例
DOMString	UTF-16 编码的字符串，同 String	{cursor: 'always'}
ULongRange	用于描述具有范围的整型值，可指定最大和最小值，包含属性如下：1) max, 32 位整型值，指定属性的最大值；2) min, 32 位整型值，指定属性的最小值	height: {min: 240, ideal: 720, max: 1080}
DoubleRange	用法与 ULongRange 基本相同，但取值类型为双精度浮点值	
ConstrainDOMString	用于类型为字符串的约束，可取值如下：1) DOMString；2) DOMString 对象数组；3) 包含 exact、ideal 属性的对象，exact 指定了一个确切值，如果浏览器不能精确匹配，则返回错误，而 ideal 指定了理想值，如果浏览器不能精确匹配，将使用最接近的值	facingMode: {exact: 'user'} aspectRatio: { ideal: 1.7777777778 }
ConstrainBoolean	用于类型为布尔值的约束。可取值如下：1) true 或者 false；2) 包含 exact、ideal 属性的对象 exact/ideal 的说明同上	
ConstrainULong	用于类型为整型值的约束。继承自 ULongRange，取值既可以是一个整型值也可以是一个对象。当取值为对象时，支持 exact/ideal 语法	width: {min: 640, ideal: 1280, max: 1920}
ConstrainDouble	用于类型为双精度浮点值的约束。继承自 DoubleRange，取值既可以是一个浮点值也可以是一个对象。当取值为对象时，支持 exact/ideal 语法	

请注意 MediaTrackConstraints 定义中属性的类型，不同的类型意味着不同的使用方法，将数据类型与表 2-2 进行对照，其用法便清晰了。

2.3.3 通用约束

通用约束是能够用于所有媒体轨道的约束，如表 2-3 所示。

表 2-3 通用约束

约 束	类 型	描 述	说 明
deviceId	ConstrainDOMString	设备 ID 或者多个设备 ID 的数组	RTCPeerConnection 关联的流不包含该约束
groupId	ConstrainDOMString	组 ID 或者多个组 ID 的数组	

2.3.4 视频约束

视频约束是仅用于视频轨道的约束，如表 2-4 所示。

表 2-4 视频约束

约 束	类 型	描 述	说明 / 示例
aspectRatio	ConstrainDouble	视频宽高比	{ aspectRatio: 16/9 }
facingMode	ConstrainDOMString	摄像头可取值如下：1) user，前置摄像头；2) environment，后置摄像头；3) left，左侧摄像头；4) right，右侧摄像头	RTCPeerConnection 关联的流不包含该约束，如：{ facingMode: 'user' }
frameRate	ConstrainDouble	帧率	{ frameRate: { ideal: 10, max: 15 } }
height	ConstrainULong	视频高度	{ height: 720 }
width	ConstrainULong	视频宽度	{ width: 1280 }
resizeMode	ConstrainDOMString	调整视频尺寸，可取值如下： 1) none，使用摄像头的原生分辨率；2) crop-and-scale，允许对视频进行裁剪	{ resizeMode: 'none' }

2.3.5 音频约束

音频约束是仅用于音频轨道的约束，如表 2-5 所示。

表 2-5 音频约束

约 束	类 型	描 述	说明 / 示例
autoGainControl	ConstrainBoolean	自动增益控制，值为 true 则开启；值为 false 则关闭	{ autoGainControl: true }
channelCount	ConstrainULong	音轨数量	值为 1 表示单声道，值为 2 表示立体声，如：{channelCount: 2}
echoCancellation	ConstrainBoolean	回音消除，值为 true 表示开启，值为 false 表示关闭	RTCPeerConnection 关联的流不包含该约束，如：{echoCancellation: true}
latency	ConstrainDouble	延时，单位为秒。一般来讲，延时越低越好	RTCPeerConnection 关联的流不包含该约束，如：{latency: 1}

(续)

约 束	类 型	描 述	说明 / 示例
noiseSuppression	ConstrainBoolean	降噪，值为 true 表示开启，值为 false 表示关闭	{ noiseSuppression: true }
sampleRate	ConstrainULong	采样率	{ sampleRate: 44100 }
sampleSize	ConstrainULong	采样大小	{ sampleSize: 16 }

2.3.6 屏幕共享约束

屏幕共享约束是仅用于屏幕共享的约束，如表 2-6 所示。

表 2-6 屏幕共享约束

约 束	类 型	描 述	说 明
cursor	ConstrainDOMString	在流中如何显示鼠标光标，可取值如下：1) always，一直显示光标；2) motion，当移动鼠标时显示光标，静止时不显示；3) never，不显示光标	{cursor: 'always'}
displaySurface	ConstrainDOMString	指定用户可以选择的屏幕内容，可取值如下：1) application，应用程序；2) browser，浏览器标签页；3) monitor，显示器；4) window，某个应用程序窗口	{displaySurface: 'application'}
logicalSurface	ConstrainBoolean	是否开启逻辑显示面	{logicalSurface: true}

2.3.7 图像约束

图像约束是仅用于图像采集的约束，如表 2-7 所示。

表 2-7 图像约束

约 束	类 型	描 述	说明 / 示例
whiteBalanceMode	ConstrainDOMString	白平衡模式，可取值如下： 1) none，关闭聚焦/曝光/白平衡模式；2) manual，开启手动控制； 3) single-shot，开启一次自动聚焦/曝光/白平衡；4) continuous， 开启连续自动聚焦/曝光/白平衡	{whiteBalanceMode: 'none'}
exposureMode	ConstrainDOMString	曝光模式，可能的取值同 whiteBalanceMode	{exposureMode: 'none'}
focusMode	ConstrainDOMString	聚焦模式，可能的取值同 whiteBalanceMode	{focusMode: 'manual'}
pointsOfInterest	ConstrainPoint2D	兴趣点，与上述 3 种模式结合使用	
exposureCompensation	ConstrainDouble	曝光补偿	

(续)

约 束	类 型	描 述	说明 / 示例
colorTemperature	ConstrainDouble	色温	
iso	ConstrainDouble	感光度	
brightness	ConstrainDouble	亮度	
contrast	ConstrainDouble	对比度	
saturation	ConstrainDouble	饱和度	
sharpness	ConstrainDouble	锐度	
focusDistance	ConstrainDouble	聚焦距离	
pan	ConstrainDouble boolean	控制摄像头的 pan 值	
tilt	ConstrainDouble boolean	控制摄像头的 tilt 值	
zoom	ConstrainDouble boolean	缩放	
torch	ConstrainBoolean	是否支持 torch 模式：值为 true 表示支持，值为 false 表示不支持	

2.3.8 约束的 advanced 属性

在 MediaTrackConstraints 的定义里，我们可以看到 MediaTrackConstraints 继承自 MediaTrackConstraintSet，增加了 advanced 属性。

advanced 属性用来指定更加高级的约束需求，通常与其他基础约束一起使用。当浏览器满足了基础约束需求后，再尝试进一步满足 advanced 的约束需求。

advanced 和 ideal 都能表示进一步的约束需求，但是它们是有区别的，advanced 的优先级高于 ideal。为了读者能够更好地理解 advanced 的用法及其与 ideal 的区别，下面结合示例展示浏览器满足约束需求的流程。

代码清单 2-29 展示了一个基础的约束需求，如果浏览器只能同时满足部分约束，比如能够满足 width 和 height，但是不能满足 aspectRatio，此时浏览器会为不能满足的约束需求分配一个合理值。

代码清单 2-29 基础约束需求

```
const constraints = {
  width: 1280,
  height: 720,
  aspectRatio: 3/2
};
```

代码清单 2-30 增加了一些复杂度，引入了 min 和 ideal 值，min 指定了强制性的最小值，ideal 指定了期望的理想值。

在 width 的约束值中，指定了 min 为 640，ideal 为 1280，表示希望采集的视频最小宽

度为 640 像素，理想宽度为 1280 像素。

代码清单2-30 引入min/ideal的约束需求

```
const constraints = {  
    frameRate: {min: 20},  
    width: {min: 640, ideal: 1280},  
    height: {min: 480, ideal: 720},  
    aspectRatio: 3/2  
};
```

浏览器在处理该约束需求时，返回的视频宽度不能低于 min 的值，如果不支持采集宽度大于或等于 640 像素的视频，则返回失败。

如果浏览器支持采集宽度为 1280 像素的视频，则 width 值使用 1280，返回成功；如果浏览器不支持，则默认为 width 指定一个大于 640 的值，仍然返回成功。至于这个默认值是多少，就由浏览器来决定了。

代码清单 2-31 引入 advanced 属性，继续增加复杂度，浏览器的行为与上面的例子基本相同，但是在尝试满足 ideal 之前，浏览器会先去处理 advanced 列表。

代码清单2-31 包含advanced属性的约束需求

```
const constraints = {  
    width: {min: 640, ideal: 1280},  
    height: {min: 480, ideal: 720},  
    advanced: [  
        {width: 1920, height: 1280},  
        {aspectRatio: 4/3}  
    ]  
};
```

advanced 列表包含了两个约束集，第一个指定了 width 和 height，第二个指定了 aspectRatio。它表达的含义是“视频分辨率应该至少为 640 像素 × 480 像素，能够达到 1920 像素 × 1280 像素最好，如果达不到，就满足 4/3 的宽高比，如果还不能满足，就使用一个最接近 1280 像素 × 720 像素的分辨率。”

2.4 媒体设备

WebRTC 使用 navigator.mediaDevices 接口访问与浏览器相连的媒体设备，该接口提供了访问摄像头、话筒以及屏幕共享的入口 API。

2.4.1 WebRTC 隐私和安全

为了保护用户的隐私，必须在安全的内容中使用 WebRTC，所谓安全内容指如下两点。

- 使用 HTTPS/TLS 加载的页面内容。

- 从本地 localhost/127.0.0.1 加载的页面内容。

如果在不安全的内容中使用 WebRTC，navigator.mediaDevices 值为 undefined，此时访问 getUserMedia() 将会报错。

在第一次打开媒体设备时，getUserMedia 会弹出请求授权的提示框，如果用户通过了授权，浏览器会记录授权结果，同一域名不重复请求授权。

浏览器必须明确显示媒体设备的使用状态和授权状态，当摄像头处于使用状态时，硬件指示灯必须亮起。另外，浏览器通常会在 URL 地址栏中显示媒体设备的状态。

在 iframe 中使用 WebRTC 时，需要明确为该 frame 请求权限，如代码清单 2-32，使用 allow 为 iframe 请求摄像头和话筒权限。

代码清单 2-32 为 iframe 请求权限

```
<iframe src="https://mycode.example.net/etc" allow="camera;microphone">
</iframe>
```

2.4.2 获取摄像头与话筒

WebRTC 使用 getUserMedia() 方法获取摄像头与话筒对应的媒体流。

```
const promise = navigator.mediaDevices.getUserMedia(constraints);
```

- 参数：constraints，这是一个 MediaStreamConstraints 对象，指定了获取媒体流的约束需求，MediaStreamConstraints 对象的使用详见 2.2 节。
- 返回值：promise，如果方法调用成功则得到一个 MediaStream 对象。如果调用失败，则抛出 DOMException 异常，异常对象的 name 属性可取值如表 2-8 所示。

表 2-8 getUserMedia() 异常说明

错误	说明
NotAllowedError	请求的媒体源不能使用，以下情况会返回该错误：1) 当前页面内容不安全，没有使用 HTTPS；2) 没有通过用户授权
NotFoundError	没有找到指定的媒体轨道
NotReadableError	尽管已经通过了用户授权，但是在访问媒体硬件时出现了错误
OverconstrainedError	不能满足指定的媒体约束。错误对象中包含 constraint 属性，用于指明不能满足的属性名称
SecurityError	Document 对象禁用了媒体支持
AbortError	用户已授权，但是因为其他原因导致访问硬件失败

代码清单 2-33 调用 getUserMedia() 方法请求音频和视频，如果调用成功则将 stream 关联到 <video> 元素，并在加载完 meta 数据后播放视频。如果调用失败，则打印错误对象的 name 和 message。

代码清单2-33 getUserMedia()方法示例

```
const constraints = {  
    audio: true,  
    video: { width: 1280, height: 720 }  
};  
  
navigator.mediaDevices.getUserMedia(constraints)  
.then((stream) => {  
    const video = document.querySelector('video');  
    video.srcObject = stream;  
    video.onloadedmetadata = (e) => {  
        video.play();  
    };  
})  
.catch((err) => {  
    console.log(err.name + " : " + err.message);  
});
```

2.4.3 共享屏幕

调用 MediaDevices.getDisplayMedia() 方法获取共享屏幕流，该方法弹出提示框，提示用户授权并选择屏幕内容。

```
const promise = navigator.mediaDevices.getDisplayMedia(constraints);
```

- 参数：constraints，可选参数，MediaStreamConstraints 约束对象，用于指定共享屏幕的约束需求。
- 返回值：promise，如果调用成功则得到媒体流；如果调用失败，则返回一个 DOMException 对象，异常说明如表 2-9 所示。

表 2-9 getDisplayMedia() 异常说明

错 误	说 明
InvalidStateError	非用户动作触发
NotAllowedError	未通过授权
NotFoundError	未找到捕获源
NotReadableError	捕获源不可读
OverconstrainedError	不能满足约束需求
TypeError	指定了不支持的约束需求
AbortError	非上述原因导致的其他失败情况

屏幕共享有可能泄露用户的隐私，出于安全考虑，WebRTC 规定：

- 1) 每次调用 getDisplayMedia() 方法都要弹出授权提示框，如果通过了授权，则不保存

授权状态；

2) `getDisplayMedia()` 方法必须由用户触发，且当前的 `document` 上下文处于激活状态。

代码清单 2-34 使用了 `async/await` 语法获取共享屏幕流，`displayMediaOptions` 是一个 `MediaStreamConstraints` 对象，指定了约束需求，如果调用成功则返回 `captureStream`；如果调用失败则打印错误信息。

代码清单2-34 `getDisplayMedia()`方法示例

```
async function startCapture(displayMediaOptions) {
    let captureStream = null;

    try {
        captureStream = await navigator.mediaDevices.getDisplayMedia(displayMediaOptions);
    } catch(err) {
        console.error("Error: " + err);
    }
    return captureStream;
}
```

2.4.4 查询媒体设备

为了让 Web 应用提供更好的使用体验，我们通常调用 `enumerateDevices()` 方法列出所有可用的媒体设备供用户选择。

```
const enumeratorPromise = navigator.mediaDevices.enumerateDevices();
```

- 参数：无。
- 返回值：`enumeratorPromise`，如果调用成功则得到一个包含成员 `MediaDeviceInfo` 的数组，该数组列出了所有可用的媒体设备；如果调用失败，则得到空值。

`MediaDeviceInfo` 的定义如代码清单 2-35 所示。

代码清单2-35 `MediaDeviceInfo`的定义

```
interface MediaDeviceInfo {
    readonly attribute DOMString deviceId;
    readonly attribute MediaDeviceKind kind;
    readonly attribute DOMString label;
    readonly attribute DOMString groupId;
    [Default] object toJSON();
};

enum MediaDeviceKind {
    "audioinput",
    "audiooutput",
    "videoinput"
};
```

表 2-10 对 MediaDeviceInfo 的属性进行说明。

表 2-10 MediaDeviceInfo 属性说明

属性	类型	说明
deviceId	DOMString	设备 ID
groupId	DOMString	组 ID，如耳机音频输入和输出设备的 groupId 相同
label	DOMString	设备的描述信息，如“External USB Webcam”
kind	MediaDeviceKind	设备类型，取值为 audioinput、audiooutput、videoinput 三者之一

代码清单 2-36 枚举所有媒体设备，并打印 MediaDeviceInfo 的属性 kind、label 和 deviceId。

代码清单 2-36 enumerateDevices()方法示例

```
navigator.mediaDevices.enumerateDevices()
.then((devices) => {
  devices.forEach((device) => {
    console.log(device.kind + " : " + device.label + " id = " + device.deviceId);
  });
})
.catch((err) => {
  console.error(err.name + " : " + err.message);
});
```

在 Chrome 浏览器的开发者工具中运行代码清单 2-36，输出如代码清单 2-37 所示。

代码清单 2-37 enumerateDevices()输出

```
audioinput: 默认 - Internal Microphone (Built-in) id = default
audioinput: Internal Microphone (Built-in) id = 77ae20211ff909ae81072ce848530071
           61d3a4d9e19838946df3fe532b8ca5a3
videoinput: FaceTime 高清相机 (内建) (05ac:8510) id = 1d510919cb6ddb949d6a7611b638
           7a83378c4246757cecf37719969eed064c7f
audiooutput: 默认 - Internal Speakers (Built-in) id = default
audiooutput: Internal Speakers (Built-in) id = a29e903d3c1e53b1a1842f21d0254ca70
           433f06e55abe50950229cbff959c8d8
```

代码清单 2-38 找出所有 kind 属性为 videoinput 的设备，即摄像头，如果找到则打印 Cameras found 信息。

代码清单 2-38 找出所有摄像头

```
function getConnectedDevices(type, callback) {
  navigator.mediaDevices.enumerateDevices()
  .then(devices => {
    const filtered = devices.filter(device => device.kind === type);
    callback(filtered);
  });
}
getConnectedDevices('videoinput', cameras => console.log('Cameras found', cameras));
```

2.4.5 监听媒体设备变化

大部分计算机都支持设备运行时热插拔，比如随时插拔 USB 摄像头、蓝牙耳机或者外接音箱。

为了在应用程序中监测媒体设备的变化，WebRTC 提供了 devicechange 事件和 ondevicechange 事件句柄，与 navigator.mediaDevices 结合即可实时监控媒体设备的热插拔。

代码清单 2-39 展示了 devicechange 事件的两种处理方法。

代码清单2-39 devicechange事件用法示例

```
// 方法1：使用addEventListener监听事件
navigator.mediaDevices.addEventListener('devicechange', (event) => {
  updateDeviceList();
});

// 方法2：使用ondevicechange事件句柄
navigator.mediaDevices.ondevicechange = (event) => {
  updateDeviceList();
}
```

代码清单 2-40 结合使用 devicechange 和 navigator.mediaDevices，当摄像头设备发生变化时，重新监测摄像头设备并更新 HTML 的下拉列表。

代码清单2-40 监测摄像头

```
// 更新select元素
function updateCameraList(cameras) {
  const listElement = document.querySelector('select#availableCameras');
  listElement.innerHTML = '';
  cameras.map(camera => {
    const cameraOption = document.createElement('option');
    cameraOption.label = camera.label;
    cameraOption.value = camera.deviceId;
  }).forEach(cameraOption => listElement.add(cameraOption));
}

// 根据指定的设备类型，获取设备数组
async function getConnectedDevices(type) {
  const devices = await navigator.mediaDevices.enumerateDevices();
  return devices.filter(device => device.kind === type)
}

// 获取初始状态的摄像头
const videoCameras = getConnectedDevices('videoinput');
updateCameraList(videoCameras);
// 监听事件并更新设备数组
navigator.mediaDevices.addEventListener('devicechange', event => {
  const newCameraList = getConnectedDevices('videoinput');
  updateCameraList(newCameraList);
});
```

34 ◆ WebRTC 技术详解：从 0 到 1 构建多人视频会议系统

2.5 从 canvas 获取媒体流

调用 `HTMLCanvasElement.captureStream()` 方法可以从 `canvas` 实时获取视频流。

```
MediaStream = canvas.captureStream(frameRate);
```

- 参数：`frameRate`，可选参数，表示视频帧率，类型为双精浮点值。如果未指定参数，则每次画布变化时都会捕获一个新帧；如果取值为 0，则不会自动捕获，而是在调用 `requestFrame()` 方法时触发捕获。
- 返回值：返回 `MediaStream` 媒体流对象，该对象包含类型为 `CanvasCaptureMediaStreamTrack` 的单一媒体轨道。

`CanvasCaptureMediaStreamTrack` 的定义如代码清单 2-41 所示。

代码清单 2-41 `CanvasCaptureMediaStreamTrack` 的定义

```
interface CanvasCaptureMediaStreamTrack : MediaStreamTrack {  
    readonly attribute HTMLCanvasElement canvas;  
    void requestFrame();  
};
```

`CanvasCaptureMediaStreamTrack` 继承自 `MediaStreamTrack`，增加了 `canvas` 属性和 `requestFrame()` 方法。

代码清单 2-42 从 `canvas` 元素捕获视频流，将视频流发送给对等端。

代码清单 2-42 `HTMLCanvasElement.captureStream()` 方法示例

```
// 获取canvas元素  
const canvasElt = document.querySelector('canvas');  
// 获取媒体流，帧率为25  
const stream = canvasElt.captureStream(25);  
// 使用RTCPeerConnection将媒体流发送给对等端  
pc.addStream(stream);
```

2.6 从媒体元素获取媒体流

调用 `HTMLMediaElement.captureStream()` 方法可以获取任意媒体元素的媒体流。

视频元素 `HTMLVideoElement` 和音频元素 `HTMLAudioElement` 都继承自 `HTMLMediaElement`，所以都支持 `captureStream()` 方法。

```
const mediaStream = mediaElement.captureStream()
```

- 参数：无。
- 返回值：返回获取到的媒体流，包含的媒体轨道与媒体源相同。

代码清单 2-43 从视频元素获取视频流，将视频流发送给对等端。

代码清单2-43 HTMLMediaElement.captureStream()方法示例

```
const playbackElement = document.getElementById("playback");
const captureStream = playbackElement.captureStream();
playbackElement.play();
pc.addStream(captureStream);
```

2.7 播放媒体流

我们使用 API 成功获取媒体流后，通常希望将该媒体流播放出来。将 MediaStream 对象指定给 HTML 的 video（或 audio）元素即可进行本地播放。

代码清单 2-44 使用 getUserMedia() 方法获取包含音视频轨道的媒体流，将流对象赋值给视频元素的 srcObject 属性，从而实现本地播放音视频。

代码清单2-44 本地播放媒体流示例

```
<html>
  <head>
    <title>Local video playback</title>
  </head>
  <body>
    <video id="localVideo" autoplay playsinline controls />
  </body>
</html>

async function playVideoFromCamera() {
  try {
    const constraints = {'video': true, 'audio': true};
    const stream = await navigator.mediaDevices.getUserMedia(constraints);
    const videoElement = document.querySelector('video#localVideo');
    videoElement.srcObject = stream;
  } catch(error) {
    console.error('Error opening video camera.', error);
  }
}
```

HTML 的 video 元素通常需要指定 autoplay、controls 和 playsinline 属性，autoplay 允许自动播放视频，controls 显示播放器控制按钮，playsinline 允许在 Safari 环境中进行非全屏播放。

iOS Safari 的限制

iOS 10 之前的版本，Safari 不支持自动播放视频，也不支持内联播放。也就是说，视频只能由用户主动操作才能播放，并且是全屏播放的。

iOS 10 版本引入了新的播放政策，通过设置 playsinline 属性可以让 Safari 浏览器窗口播放视频；如果不设置 playsinline 属性，Safari 仍会默认全屏播放视频。

iOS 10 版本还允许自动播放无音轨或者静音的视频，但是对于有声音的视频，仍然需要用户进行如下主动操作。

- 1) 用户点击播放按钮。
- 2) 若用户触发了 click/doubleclick/keydown 等事件，则在事情处理函数中调用 video.play() 方法。

2.8 录制媒体流

WebRTC 的应用经常会用到媒体流录制，下面进行详细介绍。MediaRecorder 接口提供了录制相关的 API，其定义如代码清单 2-45 所示。

代码清单 2-45 MediaRecorder 接口定义

```
interface MediaRecorder : EventTarget {  
    readonly attribute MediaStream stream;  
    readonly attribute DOMString mimeType;  
    readonly attribute RecordingState state;  
    attribute EventHandler onstart;  
    attribute EventHandler onstop;  
    attribute EventHandler ondataavailable;  
    attribute EventHandler onpause;  
    attribute EventHandler onresume;  
    attribute EventHandler onerror;  
    readonly attribute unsigned long videoBitsPerSecond;  
    readonly attribute unsigned long audioBitsPerSecond;  
    readonly attribute BitrateMode audioBitrateMode;  
    void start(optional unsigned long timeslice);  
    void stop();  
    void pause();  
    void resume();  
    void requestData();  
    static boolean isTypeSupported(DOMString type);  
};
```

2.8.1 构造 MediaRecorder

构造 MediaRecorder 对象的语法如下所示。

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

- 参数：stream，MediaStrem 对象，录制源；options，类型为 MediaRecorderOptions 的可选参数，MediaRecorderOptions 的定义如代码清单 2-46 所示。

代码清单 2-46 MediaRecorderOptions 的定义

```
dictionary MediaRecorderOptions {  
    DOMString mimeType = "";  
    unsigned long audioBitsPerSecond;
```

```
unsigned long videoBitsPerSecond;
unsigned long bitsPerSecond;
BitrateMode audioBitrateMode = "vbr";
};
```

MediaRecorderOptions 属性如表 2-11 所示。

表 2-11 MediaRecorderOptions 属性说明

属性	说 明
mimeType	指定录制流的编码格式 调用 MediaRecorder.isTypeSupported() 方法检查当前浏览器是否支持指定的编码格式。 如果当前浏览器不支持指定的编码格式，则该构造函数抛出异常 NotSupportedError
audioBitsPerSecond	指定录制流的音频码率
videoBitsPerSecond	指定录制流的视频码率
bitsPerSecond	指定录制流中音视频的码率，用于替代 audioBitsPerSecond 和 videoBitsPerSecond 属性， 如果这两个属性只指定了一个，则 bitsPerSecond 将替代另外一个
audioBitrateMode	指定音频码率模式，取值为 cbr 或 vbr。cbr 指以固定码率进行编码，vbr 指以可变码率 进行编码

如果没有指定录制流的码率，则默认视频码率为 2.5Mbps，音频码率取决于采样率和通道数。

如代码清单 2-47 所示，创建录制流，指定的视频编码格式是 mp4，如果创建成功则返回 MediaRecorder 对象，创建失败则打印错误信息并返回 null。

代码清单2-47 MediaRecorder构造函数示例

```
function getRecorder(stream) {
  const options = {
    audioBitsPerSecond : 128000,
    videoBitsPerSecond : 2500000,
    mimeType : 'video/mp4'
  };

  let mediaRecorder = null;
  try {
    mediaRecorder = new MediaRecorder(stream,options);
  } catch(e) {
    console.error('Exception while creating MediaRecorder: ' + e);
  }
  return mediaRecorder;
}
```

2.8.2 MediaRecorder 属性

1. mimeType 只读

返回构造 MediaRecorder 对象时指定的 MIME 编码格式，如果在构造时未指定，则返

回浏览器默认使用的编码格式，类型为字符串。

代码清单 2-48 调用 getUserMedia 方法获取音视频流，并指定 mp4 编码格式进行录制。

代码清单2-48 mimeType示例

```
if (navigator.mediaDevices) {
    console.log('getUserMedia supported.');

    const constraints = { audio: true, video: true };
    const chunks = [];

    navigator.mediaDevices.getUserMedia(constraints)
        .then(stream => {
            const options = {
                audioBitsPerSecond: 128000,
                videoBitsPerSecond: 2500000,
                mimeType: 'video/mp4'
            }
            const mediaRecorder = new MediaRecorder(stream,options);
            console.log(mediaRecorder.mimeType);
        }).catch(error => {
            console.log(error.message);
        });
}
```

2. state 只读

返回 MediaRecorder 对象的当前状态，类型为 RecordingState。RecordingState 的定义如代码清单 2-49 所示。

代码清单2-49 RecordingState的定义

```
enum RecordingState {
    "inactive",
    "recording",
    "paused"
};
```

表 2-12 对 RecordingState 的属性进行了说明。

表 2-12 RecordingState 属性说明

属性	说明
inactive	没有进行录制，原因可能是录制没有开始或者已经停止
recording	录制正在进行
paused	录制已开始，当前处于暂停状态

代码清单 2-50 在 onclick 事件的处理函数中启动录制并打印录制的状态。

代码清单2-50 state示例

```
record.onclick = () => {
```

```
    mediaRecorder.start();
    console.log(mediaRecorder.state);
}
```

3. stream 只读

返回构造 MediaRecorder 对象时指定的媒体流对象，类型为 MediaStream。

4. videoBitsPerSecond 只读

返回当前的视频码率，可能与构造时指定的码率不同，类型为数值。

5. audioBitsPerSecond 只读

返回当前的音频码率，可能与构造时指定的码率不同，类型为数值。

6. audioBitrateMode 只读

返回音频轨道的码率模式，类型为 BitrateMode。BitrateMode 的定义如代码清单 2-51 所示。

代码清单2-51 BitrateMode的定义

```
enum BitrateMode {
  "cbr",
  "vbr"
};
```

其中，cbr 指以固定码率进行编码，vbr 指以可变码率进行编码。

2.8.3 MediaRecorder 方法

1. isTypeSupported() 静态方法

检查当前浏览器是否支持指定的 MIME 格式。

```
const canRecord = MediaRecorder.isTypeSupported(mimeType)
```

□ 参数：mimeType，MIME 媒体格式。

□ 返回值：类型为 Boolean，如果支持该 mimeType 则返回 true，否则返回 false。

代码清单 2-52 检测 types 数组中的 mimeType，如果当前浏览器支持此 mimeType，则打印 YES，如果不支持则打印 NO。

代码清单2-52 isTypeSupported示例

```
const types = ["video/webm",
  "audio/webm",
  "video/webm\;codecs=vp8",
  "video/webm\;codecs=daala",
  "video/webm\;codecs=h264",
  "audio/webm\;codecs=opus",
  "video/mpeg"];
```

```
for (let i in types) {
    console.log("Is " + types[i] + " supported? " +
        (MediaRecorder.isTypeSupported(types[i]) ? "YES" : "NO"));
}
```

2. requestData() 方法

该方法触发 dataavailable 事件，事件包含 Blob 格式的录制数据。该方法通常需要周期性调用。

```
mediaRecorder.requestData()
```

□ 参数：无。

□ 返回值：无。如果 MediaRecorder.state 不是 recording，将抛出异常 InvalidState。

如代码清单 2-53 所示，每秒调用一次 requestData() 方法，并在 dataavailable 事件处理函数中获取录制数据。

代码清单 2-53 requestData()示例

```
this.mediaRecorder.ondataavailable = (event) => {
    if (event.data.size > 0) {
        this.recordedChunks.push(event.data);
    }
};
this.recorderIntervalHandler = setInterval(() => {
    this.mediaRecorder.requestData();
}, 1000);
```

3. start(timeslice) 方法

启动录制，将录制数据写入 Blob 对象。

```
mediaRecorder.start(timeslice)
```

□ 参数：timeslice，可选参数，用于设置录制缓存区时长，单位为毫秒（ms）。如果指定了 timeslice，当 Blob 缓存区写满后，触发 dataavailable 事件，并重新创建一个 Blob 对象。如果未指定 timeslice，则录制数据会始终写入同一个 Blob 对象，直到调用 requestData() 方法才会重新创建新的 Blob 对象。

□ 返回值：无。如果调用出错，会抛出异常，如表 2-13 所示。

表 2-13 start 异常说明

异常	说 明
InvalidModificationError	录制源的媒体轨道发生了变化，录制时不能添加或删除媒体轨道
InvalidStateError	MediaRecorder 当前状态不是 inactive
NotSupportedError	媒体源处于 inactive 状态，或者媒体轨道不可录制
SecurityError	媒体流不允许录制
UnknownError	其他未知错误

代码清单 2-54 启动录制，并将 Blob 缓存区设置为 100ms，缓存区满后触发 dataavailable 事件。

代码清单2-54 start示例

```
recorder.ondataavailable = (event) => {
    console.log(' Recorded chunk of size ' + event.data.size + "B");
    recordedChunks.push(event.data);
};

recorder.start(100);
```

4. MediaRecorder.pause() 方法

暂停录制。当调用该方法时，浏览器将产生如下行为。

- 如果 MediaRecorder.state 的状态为 inactive，则抛出异常 InvalidStateError，不再执行下面的步骤。
- 将 MediaRecorder.state 设置为 paused。
- 停止向 Blob 追加数据，等待录制恢复。
- 触发 pause 事件。

```
MediaRecorder.pause()
```

- 参数：无。
- 返回值：无。

5. MediaRecorder.resume() 方法

恢复录制。当调用该方法时，浏览器会产生如下行为。

- 如果 MediaRecorder.state 的状态为 inactive，则抛出异常 InvalidStateError，不再执行下面的步骤。
- 将 MediaRecorder.state 设置为 recording。
- 继续向 Blob 追加数据。
- 触发 resume 事件。

```
MediaRecorder.resume()
```

- 参数：无。
- 返回值：无。

代码清单 2-55 展示了暂停 / 恢复状态的切换。

代码清单2-55 resume()示例

```
pause.onclick = () => {
    if(MediaRecorder.state === "recording") {
        //暂停录制
        mediaRecorder.pause();
```

```
    } else if(MediaRecorder.state === "paused") {
        //恢复录制
        mediaRecorder.resume();
    }
}
```

2.8.4 MediaRecorder 事件

1. start 事件

当调用 MediaRecorder.start() 方法时触发该事件。此时启动录制，录制数据开始写入 Blob，对应事件句柄 onstart。

以下两种语法都可以为 start 事件设置处理函数。

```
MediaRecorder.onstart = (event) => { ... }
MediaRecorder.addEventListener('start', (event) => { ... })
```

代码清单 2-56 启动录制，并在 onstart 事件句柄中处理录制数据。

代码清单 2-56 start 事件示例

```
record.onclick = () => {
    mediaRecorder.start();
    console.log("recorder started");
}

mediaRecorder.onstart = () => {
    // start 事件处理流程
}
```

2. pause 事件

当调用 MediaRecorder.pause() 方法时触发该事件。此时暂停录制数据，对应事件句柄 onpause。

以下两种语法都可以为 pause 事件设置处理函数。

```
MediaRecorder.onpause = (event) => { ... }
MediaRecorder.addEventListener('pause', (event) => { ... })
```

代码清单 2-57 在 onclick 事件中切换录制状态并在相应的事件句柄中输出日志。

代码清单 2-57 pause 事件示例

```
pause.onclick = () => {
    if(mediaRecorder.state === "recording") {
        mediaRecorder.pause();
    } else if(mediaRecorder.state === "paused") {
        mediaRecorder.resume();
    }
}
```

```
mediaRecorder.onpause = () => {
    console.log("mediaRecorder paused!");
}

mediaRecorder.onResume = () => {
    console.log("mediaRecorder resumed!");
}
```

3. resume 事件

当调用 MediaRecorder.resume() 方法时触发该事件。此时由暂停恢复录制，对应事件句柄 onresume。

以下两种语法都可以为 resume 事件设置处理函数。

```
MediaRecorder.onResume = (event) => { ... }
MediaRecorder.addEventListener('resume', (event) => { ... })
```

4. stop 事件

当调用 MediaRecorder.stop() 方法或媒体流中止时触发该事件。此时停止录制数据，对应事件句柄 onstop。

以下两种语法都可以为 stop 事件设置处理函数。

```
MediaRecorder.onstop = (event) => { ... }
MediaRecorder.addEventListener('stop', (event) => { ... })
```

代码清单 2-58 在 ondataavailable 事件句柄中将录制的数据保存到 chunks 数组，当录制停止时，使用 chunks 生成音频地址，回放录制的数据。

代码清单 2-58 stop 事件示例

```
mediaRecorder.onstop = (e) => {
    console.log("data available after MediaRecorder.stop() called.");
    let audio = document.createElement('audio');
    audio.controls = true;
    const blob = new Blob(chunks, { 'type' : 'audio/ogg; codecs=opus' });
    const audioURL = window.URL.createObjectURL(blob);
    audio.srcObject = audioURL;
    console.log("recorder stopped");
}

mediaRecorder.ondataavailable = (e) => {
    chunks.push(e.data);
}
```

5. dataavailable 事件

该事件用于处理录制数据，对应事件句柄 ondataavailable，以下情况会触发该事件。

- 媒体流终止，导致获取不到媒体数据。

- 调用了 MediaRecorder.stop() 方法，将所有未处理的录制数据写入 Blob，停止录制。
- 调用了 MediaRecorderrequestData() 方法，将所有未处理的录制数据写入 Blob，继续录制。
- 如果在调用 MediaRecorder.start() 方法时传入了参数 timeslice，则每隔 timeslice（单位毫秒）触发一次该事件。

以下两种语法都可以为 dataavailable 事件设置处理函数。

```
MediaRecorder.ondataavailable = (event) => { ... }
MediaRecorder.addEventListener('dataavailable', (event) => { ... })
```

6. error 事件

在创建录制对象或录制过程中出现错误时触发该事件，事件类型为 MediaRecorderErrorEvent，对应事件句柄 onerror。

以下两种语法都可以为 error 事件设置处理函数。

```
MediaRecorder.onerror = (event) => { ... }
MediaRecorder.addEventListener(error, (event) => { ... })
```

表 2-14 列出了该事件触发时的错误名，错误名可以通过 MediaRecorderErrorEvent.error.name 获取。

表 2-14 MediaRecorder 错误名

错误名	说 明
InvalidStateError	在活跃状态调用了 start() 方法、resume() 方法以及在不活跃状态调用了 stop() 方法和 pause() 方法都会导致该错误
SecurityError	因为安全问题，该媒体流不允许被录制。比如使用 getUserMedia() 获取媒体流时，用户未通过授权
NotSupportedError	不支持传入 MIME 格式
UnknownError	其他未知错误

代码清单 2-59 实现了录制流函数 recordStream，在该函数中启动录制，保存录制数据，并在出错时打印错误信息。

代码清单2-59 error事件示例

```
function recordStream(stream) {
  let bufferList = [];
  let recorder = new MediaRecorder(stream);
  recorder.ondataavailable = (event) => {
    bufferList.push(event.data);
  };
  recorder.onerror = (event) => {
    let error = event.error;
```

```
switch(error.name) {  
    case InvalidStateError:  
        console.log("You can't record the video right now. Try again later.");  
        break;  
    case SecurityError:  
        console.log("Recording the specified source is not allowed due to security  
        restrictions.");  
        break;  
    default:  
        console.log("A problem occurred while trying to record the video.");  
        break;  
}  
};  
recorder.start(100);  
return recorder;  
}
```

2.9 示例

我们通过一个示例展示 WebRTC 如何与 canvas 结合，实现虚拟场景，本例从摄像头实时采集视频画面，并将视频中的白色背景替换成指定的图片，最后生成一个媒体流并展示出来，该媒体流同样可以通过 WebRTC 传输给对等端。

本例的 GitHub 地址为 <https://github.com/wistingcn/dove-into-webrtc/tree/master/chroma-keying>。

运行效果如图 2-1 所示。

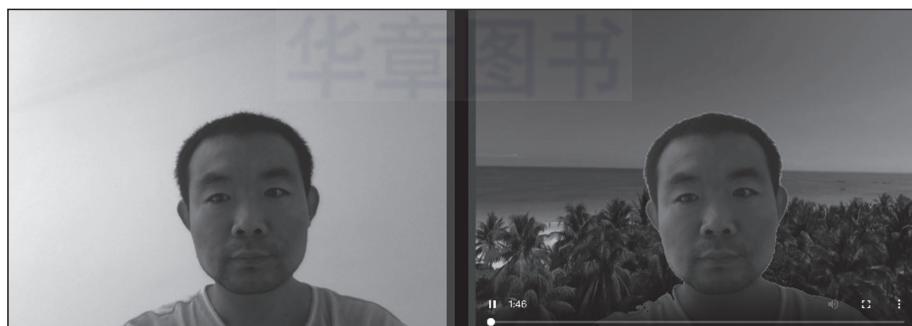


图 2-1 运行效果

2.9.1 代码结构

本例包含 3 个文件：index.html、processor.js 和 beach.jpg。

- index.html 是页面文件，定义页面样式和页面元素。

- processor.js 是 JavaScript 文件，使用 ES6 语法实现了一个 ChromaKey 类。
- beach.jpg 是背景图片，用于实时替换白色背景。

在 index.html 文件里，我们定义了两个 video 元素，ID 为 camera 的元素用于展示摄像头拍摄到的视频；ID 为 camera-chroma 的元素用于展示替换背景后的视频。index.html 的主要内容如代码清单 2-60 所示。

代码清单2-60 index.html文件代码

```
<body>
  <div>
    <video id="camera" autoplay playsinline controls/>
  </div>
  <div>
    <video id="camera-chroma" autoplay playsinline controls/>
  </div>
<script type="text/javascript" src="processor.js"></script>
</body>
```

processor.js 文件的主要流程如下。

- 使用 getUserMedia() 方法获取摄像头媒体流，我们在约束需求里指明了只获取视频流，不获取音频流。
- 在 ID 为 camera 的 video 元素里播放视频流。
- 获取图片 beach.jpg 的 RGB 像素数据并保存到 imageFrame 中，我们后续将使用这些数据替换白色像素。
- 将 ID 为 camera 的视频画面渲染到名为 c1 的 canvas 中，并从 c1 获取 RGB 像素数据。对该像素数据的 RGB 值进行判断，如果 RGB 值接近白色，则将该像素替换为对应的 imageFrame 值。
- 替换后的像素重新渲染到名为 c2 的 canvas 中。
- 调用 canvas.captureStream() 方法从 c2 中捕获视频流，在 ID 为 camera-chroma 的视频元素中播放该视频流。

请注意上文对 canvas 的使用，在整个流程中我们分别在以下 3 个地方用到了 canvas。

- 提取图片的 RGB 数据。
- 提取摄像头的 RGB 数据。
- 渲染修改后的 RGB 数据。

因为我们使用 canvas 处理图像数据，所以没有在 index.html 页面中包含 canvas 元素，而是在 Javascript 代码中进行动态创建。

2.9.2 获取图片像素数据

代码清单 2-61 展示了获取图片像素数据的方法。首先创建一个 Image 对象，把背景图

片作为 Image 对象的源，然后在 Image 对象的 onload 事件句柄中创建 canvas，并将 Image 对象绘制在 canvas 上，最后使用 canvas 方法 getImageData() 获取图片 RGBA 像素数据。

代码清单2-61 获取图片像素数据

```
getImageFrame() {
    const backgroundImg = new Image();
    backgroundImg.src = 'media/beach.jpg';
    backgroundImg.onload = () => {
        const imageCanvas = document.createElement('canvas');
        imageCanvas.width = this.width;
        imageCanvas.height = this.height;
        const ctx = imageCanvas.getContext('2d');
        ctx.drawImage(backgroundImg, 0, 0, this.width, this.height);
        this.imageFrame = ctx.getImageData(0, 0, this.width, this.height);
        this.timerCallback();
    }
}
```

getImageData() 方法获取的像素数据会保存在 this.imageFrame.data 中，类型为 Uint8ClampedArray，每个像素由 4 个 Uint8 整数组成，分别表示 R（红）、G（绿）、B（蓝）、A（透明度）数据，整数取值范围为 0~255。

2.9.3 替换视频背景

代码清单 2-62 将视频内容绘制在 canvas 上，使用 getImageData() 方法获取视频的像素数据，然后遍历所有像素。如果像素 RGB 大于 (150,150,150)，说明像素偏白色，则使用背景图片对应的像素进行替换，最后将替换后的数据重新绘制在 canvas 上。

代码清单2-62 替换视频背景

```
computeFrame() {
    this.ctx1.drawImage(this.video, 0, 0, this.width, this.height);
    let frame = this.ctx1.getImageData(0, 0, this.width, this.height);
    let l = frame.data.length / 4;

    for (let i = 0; i < l; i++) {
        let r = frame.data[i * 4 + 0];
        let g = frame.data[i * 4 + 1];
        let b = frame.data[i * 4 + 2];
        let a = frame.data[i * 4 + 3];

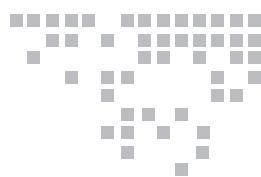
        if (r > 150 && g > 150 && b > 150) {
            frame.data[i * 4 + 0] = this.imageFrame.data[i * 4 + 0];
            frame.data[i * 4 + 1] = this.imageFrame.data[i * 4 + 1];
            frame.data[i * 4 + 2] = this.imageFrame.data[i * 4 + 2];
        }
    }
    this.ctx2.putImageData(frame, 0, 0);
}
```

2.10 本章小结

本章介绍了 WebRTC 本地媒体相关的内容，包括媒体流、媒体轨道、媒体约束、媒体设备等，我们还介绍了获取及录制媒体流的方法，最后我们将媒体流与 canvas 结合，实现了视频背景的替换。

通过本章的学习，读者应该对本地媒体有了较为全面的认识，希望大家可以运用本章介绍的知识按需操作媒体流。下一步就是将本地媒体流进行压缩编码并传输到对等端，我们将从第 3 章开始重点介绍这部分内容。





第3章

Chapter 3

传输技术

我们在第2章讨论了本地媒体的相关内容，从本章开始，我们将讨论如何将媒体流传输到对等端，其中涉及媒体信息协商、网络建连协商、网络传输等技术。这些技术不仅用于WebRTC底层，也广泛用于其他流媒体领域，比如RTP/RTCP广泛用于传统直播、监控等领域，理解这些技术的原理才能更好地使用WebRTC技术。

WebRTC基础传输技术架构如图3-1所示。

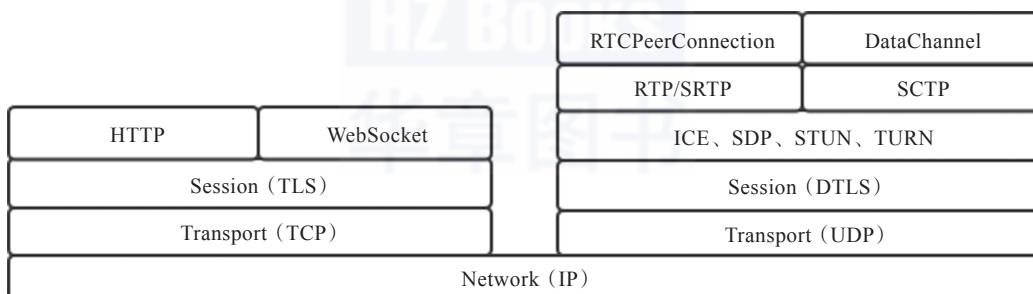


图3-1 WebRTC基础传输技术架构

在这些技术中，RTP/SRTP、SCTP用于传输媒体流，提供拥塞和流控制功能；SDP、ICE、STUN、TURN用于建立和维护网络连接；DTLS则用于保护传输数据的安全。

3.1 RTP

RTP（Real-time Transport Protocol，实时传输协议）通过IP网络实时传输音频和视频。RTP常用于流媒体服务的通信系统，例如网络电话、WebRTC视频电话会议、电视服务等。

RTP 是由 Internet 工程任务组（IETF）的音视频传输工作组开发的，其标准规范 RFC 1889 于 1996 年发布，2003 年更新为 RFC 3550。

RTP 是专为流媒体的端到端实时传输设计的，更关注信息的实时性，可以避免出现因网络传输丢失数据造成通话质量下降的情况。例如，音频应用程序中丢失数据包可能会导致音频数据丢失几分之一秒，而使用合适的纠错算法就可以实现用户无感知。

大多数 RTP 应用都是基于 UDP 构建的，并额外提供抖动补偿、包丢失检测和无序传递检测的功能。RTP 也支持 TCP，但是因为 TCP 更注重可靠性而不是实时性，所以在 RTP 应用中很少使用。

RTP 的主要特点如下。

- 具有较低的延时。
- 数据包在网络传输的过程中可能会丢失，到达对等端的顺序也可能发生变化。对等端收到 RTP 数据包后，需要根据数据包的序列号和时间戳进行重新组合。
- 支持多播（multicast），尽管目前 WebRTC 还没有使用这个特性，但是在海量用户通话场景，这个特性就变得很重要。
- 可用于音视频通话之外的场景，如实时数据流、状态实时更新、控制信息传输等连续数据传输场景。

尽管 RTP 的定位是低延时场景数据传输，但它本身并没有提供服务质量保障功能（Quality of Service，QoS），所以在 WebRTC 中，RTP 需要和 RTCP 结合使用。

RTP 会为每个媒体流建立一个会话，即音频和视频流使用单独的 RTP 会话，这样接收端就能选择性地接收媒体流。RTP 使用的端口号为偶数，每个关联的 RTCP 端口为下一个较高的奇数，端口号范围为 1024~65535。

1. RTP 配置文件与载荷

RTP 在设计之初就考虑到了在不修改标准的情况下携带多种媒体格式并允许使用新格式，所以，RTP 标头数据中不包含媒体格式信息，而是在单独的 RTP 配置文件（profile）和有效载荷（payload）格式中提供，这种方式提供了更好的可扩展性。RTP 对每类应用（如音频或视频）都定义了一个配置文件和至少一个有效载荷格式。表 3-1 列出了几种特定应用的 RTP 载荷。

表 3-1 特定应用的 RTP 载荷

载荷类型	名称	类型	通道数	频率 (Hz)	默认包 (ms)	参考规范
0	PCMU	audio	1	8 000	20	RFC 3551
8	PCMA	audio	1	8 000	20	RFC 3551
9	G722	audio	1	8 000	20	RFC 3551
26	JPEG	video		90 000		RFC 2435
32	MPV	video		90 000		RFC 2250

(续)

载荷类型	名称	类型	通道数	频率 (Hz)	默认包 (ms)	参考规范
34	H263	video		90 000		RFC 3551
dynamic	H264 AVC	video		90 000		RFC 6184
dynamic	H264 SVC	video		90 000		RFC 6190
dynamic	H265	video		90 000		RFC 7798
dynamic	opus	audio	2	48 000	20	RFC 7587
dynamic	mpeg4	audio/video		90 000		RFC 3640
dynamic	VP8	video		90 000		RFC 7741
dynamic	VP9	video		90 000		draft-ietf-payload-vp9
dynamic	jpeg2000	video		90 000		RFC 5371

载荷类型字段中定义了编解码的格式。每个配置文件都附带几种有效载荷格式规范，每个规范描述特定编码数据的传输。音频有效载荷格式包括 G.711、G.723、G.726、G.729、GSM、opus、MP3 和 DTMF 等，视频有效载荷格式包括 H.263、H.264、H.265、VP8 和 VP9 等。

RTP 配置文件包括以下 3 种。

- 音频和视频会议的 RTP 配置文件 (RFC 3551)。该配置文件定义了一组静态有效载荷类型分配以及使用会话描述协议 (SDP) 在有效载荷格式和 PT 值之间进行映射的动态机制。
- SRTP (RFC 3711) 定义了一个 RTP 配置文件，该配置文件提供用于传输有效载荷数据的加密服务，WebRTC 使用的就是 SRTP。
- 用于机器对机器通信的 RTP (RTP / CDP) 实验性控制数据配置文件。

2. RTP 数据包标头

在应用层创建 RTP 数据包并传递到传输层进行传输。应用层创建的 RTP 媒体数据的每个单元都以 RTP 数据包标头开始，标头结构如表 3-2 所示。

表 3-2 RTP 数据包标头域

偏移量	字节	0	1	2	3
字节	位	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
0	0	版本号 P X CC M	PT		序列号
4	32			时间戳	
8	64			SSRC 同步源标识	
12	96			CSRC 贡献源标识	
12+4 × CC	96+32 × CC	扩展头 ID		扩展头长度	
16+4 × CC	128+32 × CC		扩展头		
			...		

RTP 标头最小为 12 个字节。标头后面有可选的扩展头，随后是 RTP 有效载荷，其格式由特定应用类别确定。标头中各字段解释如下所示。

- 版本号（2 位）：表示协议的版本，当前版本是 2。
- P（Padding，1 位）：表示 RTP 数据包末尾是否有额外的 Padding 字节。Padding 字节用于填充一定大小的数据块，如加密算法，其最后一个字节表示 Padding 的字节数（包括自身）。
- X（扩展名，1 位）：表示在标头和有效载荷数据之间是否存在扩展标头。
- CC（CSRC 计数，4 位）：表示包含 CSRC 标识符的数量。
- M（标记，1 位）：表示在应用程序级别使用的信令。对于视频，标记一帧的结束；对于音频，标记会话的开始。
- PT（有效载荷类型，7 位）：表示有效载荷的格式，从而确定应用程序对其的解释。值是特定于配置文件的，可以动态分配。
- 序列号（16 位）：RTP 数据包的序列号。每发送一个 RTP 数据包，序列号都会递增，接收方将使用该序列号检测包丢失并适应无序交付。为了提升 RTP 的安全性，序列号的初始值应随机分配。
- 时间戳（32 位）：RTP 数据包的时间标识。接收端以此在适当的时间播放接收到的样本。当存在几个媒体流时，每个流的时间戳都是独立的。时序的粒度特定于应用程序，如音频应用程序的常见采样率是每 $125\mu\text{s}$ 对数据进行一次采样，换算成时钟频率为 8kHz，而视频应用程序通常使用 90 kHz 时钟频率。时间戳反映了发送者采样 RTP 报文的时刻，接收者使用时间戳计算延迟和延迟抖动，并进行同步控制。
- SSRC（32 位）：表示 RTP 数据包的同步源，用于标识媒体源。同一 RTP 会话中的同步源是唯一的。
- CSRC（32 位）：表示 RTP 数据包的贡献源，同一 RTP 会话可以包含多个贡献源。
- 标头扩展名：可选项，由扩展名字段决定是否存在。第一个 32 位字包含一个特定于配置文件的标识符（16 位）和一个长度说明符（16 位）。

3.2 RTCP

RTCP（RTP Control Protocol）是实时传输协议（RTP）的姊妹协议，其基本功能和数据包结构在 RFC 3550 中定义。RTCP 为 RTP 会话提供带外统计信息和控制信息，与 RTP 协作提供多媒体数据的传输和打包功能，其本身不传输任何媒体数据。

RTCP 的主要功能是定期发送数据包计数、数据包丢失、数据包延迟变化以及往返延迟时间等统计信息，向媒体参与者提供媒体分发中的服务质量保障。应用程序在接收到这些信息后，可以通过限制流量或更换编解码格式的方式提升服务质量。RTCP 流量的带宽很小，

通常约为总占用带宽的 5%。

RTP 通常在偶数 UDP 端口上发送，而 RTCP 消息将在下一个更高的奇数端口发送。

RTCP 本身不提供任何流加密或身份验证方法。如果对安全性有更高的要求，可以使用 RFC 3711 中定义的 SRTP 实现此类机制，WebRTC 便采用了 SRTP。

RTCP 提供以下功能。

- 在会话期间收集媒体分发质量方面的统计信息，并将这些数据传输给会话媒体源和其他会话参与方。源可以将此类信息用于自适应媒体编码（编解码器）和传输故障检测。如果会话是多播网络承载的，则允许进行非侵入式会话质量监视。
- RTCP 为所有会话参与者提供规范的端点标识符（CNAME）。CNAME 是跨应用程序示例端点的唯一标识。尽管 RTP 流的 SSRC 也是唯一的，但在会话期间，SSRC 与端点的绑定关系仍可能改变。
- 提供会话控制功能。RTCP 是联系所有会话参与者的便捷方式。

在数以万计的接收者参与的直播会话中，所有参与者都发送 RTCP 报告，网络流量与参与者的数量成正比。为了避免网络拥塞，RTCP 必须支持会话带宽管理功能。RTCP 通过动态报告传输的频率来实现这一功能；RTCP 带宽使用率通常不应超过会话总带宽的 5%，应始终将 RTCP 带宽的 25% 预留给媒体源，以便在大型会议中，新的参与者可以接收发送者的 CNAME 标识符而不会产生过多延迟。

RTCP 报告间隔是随机的，最小报告间隔为 5 秒，通常发送 RTCP 报告的频率不应低于 5 秒一次。

RTCP 数据包标头结构如表 3-3 所示。

表 3-3 RTCP 数据包标头

偏移	字节	0							1							2							3									
		位	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
字节	0	版本号	P	RC					PT								长度															
4	32	SSRC																														

RTCP 标头长度为 4 个字节，标头中各字段解释如下。

- 版本号（2 位）：表示 RTCP 的版本号。
- P（1 位）：表示 RTP 数据包末尾是否有额外的 Padding 字节。Padding 字节用于填充一定大小的块，最后一个字节表示 Padding 的字节数（包括自身）。
- RC（5 位）：表示此数据包中接收报告块的数量，可以为 0。
- PT（8 位）：包含一个常数，用于表示 RTCP 数据包类型。
- 长度（16 位）：表示此 RTCP 数据包的长度。
- SSRC（32 位）：同步源标识符，用于唯一标识媒体源。

RTCP 支持以下几种类型的数据包。

1. 发送者报告 (SR)

活跃发送者在会议中定期发送报告，报告该时间间隔内发送的所有 RTP 数据包的发送和接收统计信息。发送者报告包含绝对时间戳，表示自 1900 年 1 月 1 日零点以来经过的秒数。绝对时间戳帮助接收方同步 RTP 消息，对于同时传输音频和视频的场景尤为重要，因为音频和视频的 RTP 流独立使用相对时间戳，必须使用 RTCP 绝对时间戳进行同步。

2. 接收者报告 (RR)

接收者报告适用于不发送 RTP 数据包的被动参与者，用于通知发送者和其他接收者服务质量。

3. 源描述 (SDES)

源描述可以用于将 CNAME 项发送给会话参与者，也可以用于提供其他信息，例如名称、电子邮件地址、电话号码以及源所有者或控制者的地址。

4. 关闭流 (BYE)

源发送 BYE 消息以关闭流，允许端点 (endpoint) 宣布即将离开会议。

5. 特定于应用程序的消息 (APP)

APP 提供了一种机制，用于扩展 RTCP。

3.3 SRTP/SRTCP

对于未加密的实时通信应用，可能会遇到多种形式的安全风险。在浏览器和浏览器之间，或者浏览器和服务器通信之间传输未加密的数据时，都有可能被第三方拦截并窃取。

基础的 RTP 没有内置任何安全机制，因此不能保证传输数据的安全性，只能依靠外部机制进行加密。实际上，WebRTC 规范明确禁止使用未加密的 RTP。出于增强安全性的考虑，WebRTC 使用的是 SRTP。

SRTP 是 RTP 的一个配置文件，旨在为单播和多播应用程序中的 RTP 数据提供加密、消息身份验证和完整性以及重放攻击保护等安全功能。SRTP 有一个姊妹协议：安全 RTCP (SRTCP)，它提供了与 RTCP 相同的功能，并增强了安全性。

使用 SRTP 或 SRTCP 时必须启用消息身份验证功能，其他功能（如加密和身份验证）则都是可选的。

SRTP 和 SRTCP 默认的加密算法是 AES，攻击者虽然无法解密数据，但可以伪造或重放以前发送的数据。因此，SRTP 标准还提供了确保数据完整性和安全性的方法。

为了验证消息并保护其完整性，SRTP 使用 HMAC-SHA1 算法计算数据内容的摘要，并将摘要数据附加到每个数据包的身份验证标签。接收者收到数据后也同样计算摘要数据，

如果摘要数据相同，表示内容完整；如果摘要数据不同，表示内容不完整或者被篡改了。



注意 SRTP 仅加密 RTP 数据包的有效载荷，不对标头进行加密。但是，标头可能包含需要保密的各种信息。RTP 标头中包含的此类信息之一就是媒体数据的音频级别。实际上，任何看到 SRTP 数据包的人都可以判断出用户是否在讲话。尽管媒体数据是加密的，但这仍有可能泄露重要的隐私。

3.4 TLS/DTLS

安全套接层（Secure Socket Layer, SSL）是为网络通信提供安全保证及数据完整性的一种安全协议。SSL 由网景公司（Netscape）研发，用于确保互联网连接安全，保护两个系统之间发送的敏感数据，防止网络犯罪分子读取和篡改传输信息。IETF 对 SSL 3.0 进行了标准化，并添加了一些机制，经过标准化的 SSL 更名为 TLS（Transport Layer Security，安全传输层）协议。所以，可以将 TLS 理解为 SSL 标准化后的产物，SSL 3.0 对应着 TLS 1.0 版本。

TLS 的最新版本是 1.3，在 RFC 8446 中定义，于 2018 年 8 月发布。

由于 TLS 是基于 TCP，不能保证 UDP 上传输的数据的安全性，因此在 TLS 协议架构上进行了扩展，提出 DTLS（Datagram Transport Layer Security，数据包传输层安全性）协议，使之支持 UDP，DTLS 即成为 TLS 的一个支持数据包传输的版本。DTLS 使用非对称加密方法，对数据身份验证和消息身份验证提供完全加密。

在 WebRTC 规范中，加密是强制要求的，并在包括信令机制在内的所有组件上强制执行。其结果是，通过 WebRTC 发送的所有媒体流都通过标准化的加密协议进行安全加密。在选择加密协议时，具体取决于通道类型；通过 RTCDatagramChannel 发送的数据流使用 DTLS 协议加密；通过 RTP 传输的音视频媒体流则使用 SRTP 加密。

DTLS 协议内置在 WebRTC 的标准化协议中，并且是在 Web 浏览器、电子邮件和 VoIP 平台中始终使用的协议，这意味着基于 Web 的应用程序无须提前设置。

3.5 SDP

SDP（Session Description Protocol）是用于描述媒体信息的协议，以文本格式描述终端功能和首选项。SDP 只包含终端的媒体元数据，不包含媒体数据内容。建立连接的双方通过交换 SDP 获取彼此的分辨率、编码格式、加密算法等媒体信息。SDP 广泛用于会话启动协议（SIP）、RTP 和实时流协议（RSP）。

SDP 通常包含如下内容。

□ 会话属性。

- 会话活动的时间。
- 会话包含的媒体信息。
- 媒体编 / 解码器。
- 媒体地址和端口信息。
- 网络带宽的信息。

WebRTC 使用 SDP 交换双方的网络和媒体元数据，当遇到连接失败、黑流等问题时，分析 SDP 往往是查找问题最为有效的方法。

1. SDP 字段的含义及格式

表 3-4 所示是 SDP 各个字段的含义及格式。

表 3-4 SDP 字段含义及格式

字 段	含 义	格 式
v=	SDP 版本	v=0
o=	会话发起人和标识信息	o=<username> <session id> <version> <network type> <address type> <address>
s=	会话名称	s=<session name>
i=*	会话信息	i=<session description>
u=*	描述的 URI	u=<URI>
e=*	email 地址	e=<email address>
p=*	电话号码	p=<phone number>
c=*	连接信息	c=<network type> <address type> <connection address>
b=*	带宽信息	b=<modifier>:<bandwidth-value>
z=*	时区校正	z=<adjustment time> <offset>
k=*	密钥	k=<method>:<encryption key>
a=*	会话属性	a=<attribute>:<value>
t=	会话有效时间	t=<start time> <stop time>
r=*	重复次数	r=<repeat interval> <active duration> <list of offsets from start-time>
m=	媒体名称和传输地址	m=<media> <port> <transport> <fmt list>
j=	标题	j=<media or session title>

2. SDP 示例

我们看一段典型的 SDP 示例，如代码清单 3-1 所示。

代码清单 3-1 SDP示例

```
v=0
o=jdoe 2890844526 2890842807 IN IP4 10.47.16.5
s=SDP Seminar
```

```
i=A Seminar on the session description protocol
u=http://www.example.com/seminars/sdp.pdf
e=j.doe@example.com (Jane Doe)
c=IN IP4 224.2.17.12/127
t=2873397496 2873404696
a=recvonly
m=audio 49170 RTP/AVP 0
m=video 51372 RTP/AVP 99
a=rtpmap:99 h263-1998/90000
```

该会话由用户 jdoe 创建, email 地址是 j.doe@example.com, 发起会话的源地址为 10.47.16.5, 会话名称是 SDP Seminar, i 和 u 字段描述了会话的扩展信息。

t 字段指明了会话在 2 个小时内有效, c 字段指明了目标的 IP 地址为 224.2.17.12, 地址的 TTL 是 127, a 字段表明只接收数据。

两个 m 字段指明都使用 RTP 音视频配置: 第一个音频媒体流使用端口 49170, 载荷类型是 0; 第二个视频媒体流使用端口 51372, 载荷类型是 99。

最后, a 字段指明了类型 99 使用的编码格式是 h263-1998, 编码时钟频率是 90kHz。

我们再来看一段 WebRTC 使用 H.264 编码时的 SDP 信息片段, 如代码清单 3-2 所示。

代码清单3-2 H.264编码SDP片段

```
m=video 49170 RTP/AVP 98
a=rtpmap:98 H264/90000
a=fmtp:98 profile-level-id=42A01E;packetization-mode=1;
```

代码清单 3-2 表达的含义是本会话包含的是视频内容, 使用 H.264 进行编码, 编码时钟频率是 90kHz, profile-level-id 和 packetization-mode 是传给 H.264 的参数。

3.6 ICE

ICE (Interactive Connectivity Establishment, 交互式连接建立协议) 是用于提案 / 应答模式的 NAT (网络地址转换) 传输协议, 主要用于在 UDP 协议下建立多媒体会话。对于采用 Mesh 结构的 WebRTC 应用程序, 当通信双方尝试建立 P2P 连接时, 如果有一方 (或者双方) 位于 NAT 网络内部, 则直接建立 P2P 连接会失败, 这时候必须有一种能够突破 NAT 限制的技术, 这个技术就是 ICE 协议。

由于 IPv4 地址资源较为有限, 而且目前仍然在大量使用, 因此大多数接入互联网的设备都部署于 NAT 网络内部, 不是真正拥有一个公网 IPv4 地址。NAT 网关将出站请求地址动态映射为公网地址, 相应地, 将入站请求转换为内网地址, 以确保内部网络上的路由正确。由于 NAT 的限制, 在使用内网 IP 地址建立 P2P 连接时经常会出现连接失败的情况。ICE 技术可以克服 NAT 的限制, 是建立 P2P 网络连接的最佳路径。

尝试建立连接的双方都有可能位于 NAT 网络之中，也就是说它们都不能直接使用 IP 地址建立网络连接。

使用 ICE 技术建立网络连接的步骤如下所示。

第一次尝试：ICE 首先尝试使用本地网卡地址与对等方建立 P2P 连接，此地址通常为内网地址。如果连接双方位于同一个内网，则成功建立连接。

第二次尝试：如果第一次尝试失败（这对于双方都位于 NAT 网络内部的情况是不可避免的，由于网络的复杂性，也可能会由其他原因导致连接失败），则 ICE 将使用 STUN 服务器获取 NAT 设备的公网 IP 地址及映射端口，并尝试使用该 IP 地址建立连接。对于只有一方位于 NAT 网络内部，或者双方都位于非对称 NAT 网络内的情况，连接通常会成功建立。

第三次尝试：如果第二次尝试失败，意味着双方无法直接建立 P2P 连接，这时需要通过一个中介进行数据中转，这个中介即 TURN 服务器。也就是说，第三次尝试是直接与 TURN 服务器建立连接，而随后的媒体数据流将通过 TURN 中继服务器进行转发。

TURN 服务器通常会架设在数据中心，并指定公网 IP 地址，只要 TURN 服务器正常，则在网络通畅的情况下，通信双方与 TURN 服务器建立连接一定会成功。

在上述情况下，ICE 尝试建立连接所使用的地址称为候选地址，这是因为这些地址能否成功建立连接是不确定的，需要尝试后才能确定。候选地址以文本的格式呈现，多个候选地址按如下顺序排序。

- ❑ 使用内网 IP 地址。
- ❑ 使用 STUN 发现公网 IP 地址。
- ❑ 使用 TURN 作为网络中继。

ICE 是在所有的候选地址中，选择开销最小的路由。

1. NAT

NAT 是一种实现内网主机与互联网通信的方法。使用这种方法时需要在内网出口设备上安装 NAT 软件，而这种装有 NAT 软件的设备叫作 NAT 路由器，且需要至少有一个有效的公网 IP 地址。这样，所有使用内网地址的主机在和外界通信时，都要在 NAT 路由器上将内网地址转换成公网 IP 地址，才能和互联网连接。NAT 将自动修改 IP 报文的源 IP 地址和目的 IP 地址，IP 地址校验则在 NAT 处理过程中自动完成。

NAT 的应用极为广泛，当我们接入某个局域网，或者连接 Wi-Fi 时，我们实际上已经处于 NAT 网络之中了。NAT 具有以下优点。

- ❑ 共享上网：NAT 技术通过地址和端口映射，使用少量公网 IP 即可实现大量内网 IP 地址共享上网。
- ❑ 提高网络安全：不同的内网 IP 地址映射到少量公网 IP 地址，对外隐藏了内网网络结构，从而防止外部攻击内网服务器，降低了网络风险。

□ 方便网络管理：通过改变映射关系即可实现内网服务器的迁移和变更，便于对网络进行管理。

□ 节省成本：使用了少量公网 IP 地址，节省了 IP 地址的注册及使用费用。

按照地址转换方法进行划分，NAT 分为如下 4 类。

(1) 全锥形 NAT (Full cone NAT)

□ 一旦一个内网地址 (ip1:port1) 映射到公网地址 (ip2:port2)，所有发自 ip1:port1 的包都经由 ip2:port2 向外发送。任意外部主机都能通过向 ip2:port2 发包到达 ip1:port1。

(2) 地址受限锥形 NAT (Address-Restricted cone NAT)

□ 只接收曾经发送到对端 IP 地址的数据包。一旦有一个内网地址 (ip1:port1) 映射到公网地址 (ip2:port2)，所有发自 ip1:port1 的包都经由 ip2:port2 向外发送。任意外部主机 (hostAddr:any) 都能通过向 ip2:port2 发包到达 ip1:port1，但前提是 ip1:port1 之前有向 hostAddr:any 发送过包，any 表示端口不受限制。

(3) 端口受限锥形 NAT (Port-Restricted cone NAT)

□ 类似地址受限锥形 NAT，但是端口也受限制。一旦有一个内网地址 (ip1:port1) 映射到外网地址 (ip2:port2)，所有发自 ip1:port1 的包都经由 ip2:port2 向外发送。一个外部主机 (hostAddr:port3) 能够发包到达 ip1:port1 的前提是 ip1:port1 之前有向 hostAddr:port3 发送过包。

(4) 对称 NAT (Symmetric NAT)

□ 映射的外网地址端口号不固定，会随着目的地址的变化而变化。

锥形 NAT 与对称 NAT 的区别在于，在 NAT 已分配端口号 port2 给客户端的情况下，如果 Client 继续用 port1 端口与另一外网服务器通信，锥型 NAT 还会继续用原来的 port2 端口，即所分配的端口号不变。而对于对称 NAT，NAT 将会分配另一端口号（如 port3）给 Client 的 port1 端口。也就是说，同一内网主机、同一端口号，对于锥形 NAT，无论与哪一个外网主机通信，都不改变所分配的端口号；而对于对称 NAT，同一内网主机、同一端口号，每一次与不同的外网主机通信，就重新分配一个端口号。

对称 NAT 的这个特性使得位于该网络下的 WebRTC 用户无法使用 STUN 协议建立 P2P 连接。

2. STUN 与 TURN

位于 NAT 网络内的设备能够访问互联网，但并不知道 NAT 网络的公网 IP 地址，这时候就需要通过 STUN 协议实时发现公网 IP。

STUN (Session Traversal Utilities for NAT) 是一种公网地址及端口的发现协议，客户端向 STUN 服务发送请求，STUN 服务返回客户端的公网地址及 NAT 网络信息。

对于建立连接的双方都位于对称 NAT 网络的情况，使用 STUN 发现网络地址后，仍然

无法成功建立连接。这种情况就需要借助 TURN 协议提供的服务进行流量中转。

TURN (Traversal Using Relays around NAT) 通过数据转发的方式穿透 NAT，解决了防火墙和对称 NAT 的问题。TURN 支持 UDP 和 TCP 协议。

通信双方借助 STUN 协议能够在不使用 TURN 的情况下成功建立 P2P 连接。如有特殊情况，无法建立 P2P 连接，则仍需要使用 TURN 进行数据转发。

图 3-2 展示了单独使用 STUN 与结合使用 STUN 和 TURN 的对比。

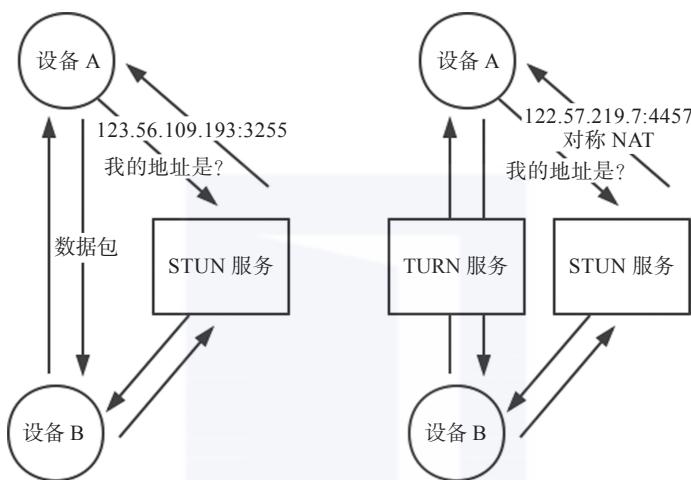


图 3-2 STUN/TURN 示意图



使用 STUN 建立的是 P2P 的网络模型，网络连接直接建立在通信两端，没有中间服务器介入；而使用 TURN 建立的是流量中继的网络模型，用户两端都与 TURN 服务建立连接，用户的网络数据包通过 TURN 服务进行转发。

3. ICE 候选者

ICE 候选者描述了用于建立网络连接的网络信息，包含网络协议、IP 地址、端口等。如果设备上有多个 IP 地址，那么每个 IP 地址都会对应一个候选。例如设备 A 上有内网 IP 地址 IP-1，还有公网 IP 地址 IP-2，A 通过 IP-1 可以直接与 B 进行通信，但是 WebRTC 不会判断优先使用哪个 IP 地址，而是同样从两个 IP 地址收集候选，并将候选放在 SDP 中，作为提案发送给 B。

设备 A 和 B 很有可能位于 NAT 网络环境中，这时就涉及另外两种类型的候选：NAT 映射候选和 TURN 中继候选。当使用 TURN 服务时，两种类型的候选都从 TURN 获取；如果只使用了 STUN 服务，则只需要获取 NAT 映射候选。

图 3-3 使用了 TURN 服务来发现这两种类型的候选，X:x 指的是 IP 地址 X 和 UDP 端口 x。

在图 3-3 中，设备 A 向 TURN 服务发起了地址分配请求，由于 A 位于 NAT 网络环境下，NAT 将创建一个映射地址 X1:x1，为设备 A 收发网络数据包，如果 A 位于多个 NAT 设备下，那么每个 NAT 都会创建一个映射地址，但是只有最外层的映射地址能够被 TURN 服务发现。

TURN 服务收到请求后，会在自己的地址 Y 上分配一个端口 y，将 Y:y 作为中继候选发送给设备 A。设备 A 完成 3 种类型候选的收集后，将它们按照优先级从高到低排序，以会话属性的形式加入 SDP，然后通过信令服务器发送给设备 B。设备 B 收到设备 A 的候选信息后，也以同样的方式完成自己的候选信息收集，并回传给设备 A。这时候 A 和 B 都有了自己和对方的候选信息，WebRTC 会将这些候选信息进行配对，再进行连通性检查，使用通过检查的候选对建立网络连接。

4. ICE 候选者在 SDP 中的语法

ICE 候选者在 SDP 中的语法格式如代码清单 3-3 所示。

代码清单3-3 SDP中的ICE候选者

```
a=candidate:<foundation> <component-id> <transport> <priority> <connection-
address> <port> <cand-type>
// 举例
a=candidate:4234997325 1 udp 2043278322 192.168.0.56 44323 host
```

代码清单 3-3 中的字段说明如下。

- ❑ foundation：创建标识。
- ❑ component-id：值为 1 表示 RTP 协议，值为 2 表示 RTCP 协议。
- ❑ transport：传输协议，可以使用 UDP 和 TCP，由于 UDP 性能好，故障恢复快，推荐使用 UDP。
- ❑ priority：优先级，综合考虑延时、带宽开销、丢包等因素，候选类型的优先级一般是 host>srvflx>prflx>relay。
- ❑ connection-address：IP 地址。
- ❑ port：端口。
- ❑ cand-type：候选类型，UDP 候选类型取值有 host(本机候选)、srflx(映射候选)、relay(中继候选) 和 prflx(来自对称 NAT 的映射候选)。

5. ICE 配对

将本地 ICE 候选项和对等端 ICE 候选项进行一一对应，每一组称为一个 ICE 候选对。

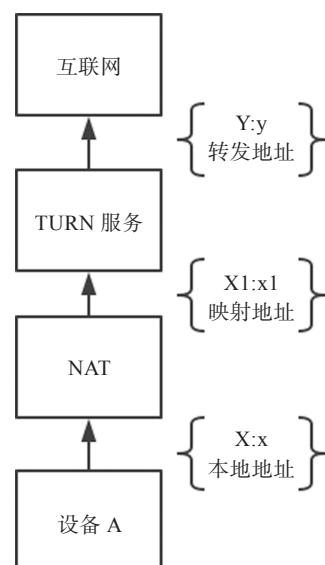


图 3-3 ICE 候选者发现步骤
(使用 TURN 服务)

在进行 ICE 建连协商时，ICE 层将从两端选择一个作为控制代理（controlling agent），另外一端作为受控代理（controlled agent）。

控制代理负责决定建立连接使用的 ICE 候选对，并将最终结果发送给受控代理，受控代理只需要等待结果。

控制代理通常会为同一个 ICE 会话选择多个候选对，每选择一个候选对都会通知受控代理，两端同时使用新的候选对尝试建立连接。

当控制代理发送完所有的候选项时，需要通知对等端，做法是将 RTCIceCandidate.candidate 属性设置为空字符串，调用 addIceCandidate() 方法将 RTCIceCandidate 加入 ICE 连接。

当 ICE 协商完成后，当前正在使用的候选对即为最终配对结果。如果出现 ICE 重新协商，则重新开始配对。需要注意的是，由于网络连接配置可能发生变化（如切换网络），每次最终配对的结果可能不同。

RTCIceCandidate.transport.role 属性指出了当前 ICE 的代理角色，代理角色的控制由 WebRTC 自动完成，通常不需要我们关注它。

6. ICE 重启

使用 WebRTC 的应用程序时，网络环境经常会发生变化，比如用户可能从 Wi-Fi 切换到移动网络，或者从移动网络切换到 Wi-Fi，网络故障导致的闪断现象也时有发生。

当出现网络切换或者网络中断的情况时，需要重新协商网络连接，协商过程与最初建立连接相同，这个过程称为 ICE 重启。ICE 重启能够确保媒体流的传输不会中断，实现平滑的网络切换。

关于 ICE 重启的使用示例，参见第 4 章。

7. ICE Trickle

在实际使用过程中，ICE 技术存在的一个问题是呼叫建连很慢，原因是 ICE 协商过程耗费了过多时间。客户端在发起呼叫时先与 STUN 服务器通信，从 STUN 服务器获取映射候选地址和中继候选地址，加上本地候选地址，构造三类 ICE 候选。之后把这三类候选放到 SDP 属性中（`a=*`），完成这个动作后才实际发起 SDP 提案（offer）请求。接收者经过同样的过程，待两边都收到对方完整的 SDP 信息后才开始进行实际的 P2P 建连。建连要发生在所有候选都获取完后，造成大量时间浪费，所以为了加快通话建立的速度，建议把连通性检测提前，即 ICE Trickle 方案。该方案的思想是客户端一边收集候选一边发送给对方，比如本地候选不需要通过 STUN 服务获取，直接就可以发起，这样节省了连通性检测的时间。

在 WebRTC 中使用 ICE Trickle，需要在对象 RTCPeerConnection 监听事件 `icecandidate`。WebRTC 完成本地 ICE 候选者的搜集后，会触发该事件，该事件对象中包含 `candidate` 属性，然后使用信令服务器将 `candidate` 传送给对等端。

WebRTC 使用 ICE Trickle 的示例如代码清单 3-4 所示。

代码清单3-4 WebRTC使用ICE Trickle

```
// 在RTCPeerConnection对象中监听icecandidate
peerConnection.addEventListener('icecandidate', event => {
  if (event.candidate) {
    signalingChannel.send({ 'new-ice-candidate': event.candidate });
  }
});
// 在信令服务器上监听对等端的ICE信息，并将ICE信息加入本地RTCPeerConnection
signalingChannel.addEventListener('message', async message => {
  if (message.iceCandidate) {
    try {
      await peerConnection.addIceCandidate(message.iceCandidate);
    } catch (e) {
      console.error('Error adding received ice candidate', e);
    }
  }
});
```

3.7 搭建 STUN/TURN 服务器

WebRTC 开源社区提供了一个较为成熟的项目 coturn 来实现 STUN/TURN 服务。coturn 项目的开源地址如下。

<https://github.com/coturn/coturn.git>

在 coturn 项目主页里，可以下载最新源代码，并对源代码进行编译，如代码清单 3-5 所示。

代码清单3-5 源码编译turnserver

```
$ tar xvfz turnserver-<...>.tar.gz
$ ./configure
$ make
$ make install
```

在编译过程中，如果当前服务器缺少编译所必须的依赖库，编译可能会失败。可以使用系统自带的包管理器进行安装，包管理器会自动处理包依赖关系。

在 ubuntu 服务器上，使用 apt-get 安装 coturn 服务，如代码清单 3-6 所示。

代码清单3-6 使用apt-get安装coturn

```
~# apt update
~# apt search coturn
Sorting... Done
Full Text Search... Done
coturn/bionic-updates,bionic-security,now 4.5.0.7-1ubuntu2.18.04.1 amd64
  TURN and STUN server for VoIP
~# apt install coturn
```

使用 coturn 时，需要注意以下事项。

- coturn 支持多种数据存储方式，默认采用 sqlite，数据库文件地址为 /var/lib/turn/turndb。
- coturn 的配置文件默认位于 /etc/turnserver.conf 下，可以通过命令行 -c 参数指定配置文件。
- 由于 WebRTC 使用 long-term 的认证机制，所以启动 coturn 时必须指定 -a 选项（或者 --lt-cred-mech）。
- WebRTC 要求不能使用匿名访问模式，必须通过 turnadmin 工具创建用户名及密码。
- 指定 -r 选项，设置默认域 (realm)。
- 指定 -f 选项。
- 使用 -v 选项可以方便地查看当前连接的客户端信息。
- 如果你的服务器位于 NAT 网络中，则需要提供外部 IP 地址，可以用命令行 -X 选项指定，也可以在配置文件里指定。大部分云主机都位于 NAT 网络中，需要指定外部 IP。
- 需要提供 HTTPS 证书。

代码清单 3-7 是启动 coturn 服务的代码。

代码清单3-7 启动coturn服务

```
// 以后台方式运行
~# coturn -afo
// 添加用户
~# turnadmin -a -u liwei -r liweix.com -p password123
// 使用telnet命令查看turnserver的运行状态
~# telnet 127.0.0.1 5766
// 查看客户端连接信息
~# turnserver -v
```

turnserver.conf 文件示例如代码清单 3-8 所示。

代码清单3-8 turnserver.conf文件

```
external-ip=[外部IP地址]
realm=liweix.com
cert=/usr/local/ssl/liweix_com.pem
pkey=/usr/local/ssl/liweix_com.key
cli-password=qwerty
```

至此，STUN/TURN 服务器搭建完毕。

STUN 协议规范定义了一些错误码，当 ICE 协商失败时，可以使用这些错误码诊断失败原因，如表 3-5 所示。

表 3-5 STUN 错误码

错误码	说 明
300	将本次请求重定向到另外一个可替代的服务

(续)

错误码	说 明
400	错误请求
401	未授权
403	禁止访问
420	未知属性
437	服务器端已经收到请求，但是配额不匹配
438	NONCE 失效
441	credentials 错误
442	传输协议不支持
486	用户配额达到上限
500	服务器错误
508	服务器达到了性能瓶颈

3.8 本章小结

WebRTC 作为一套应用层的实时通信框架，对众多底层传输技术做了整合，这些技术诞生于 WebRTC 之前，并且已经在其他应用场景中得到了验证。

为了帮助读者更好地理解和使用 WebRTC 的 API，本章对这些底层传输技术的主要内容进行了介绍。每一项技术如果展开介绍，都会涉及众多内容，而本章并不想脱离 WebRTC 的主题，所以如果读者有兴趣深入了解某项技术，请查阅相关的 IETF 规范文档。

我们将在下一章介绍 WebRTC 如何使用这些传输技术进行连接管理。