

12 | RTCPeerConnection : 音视频实时通讯的核心

2019-08-10 李超

从0打造音视频直播系统

[进入课程 >](#)



讲述：李超

时长 17:19 大小 15.86M



RTCPeerConnection 类是在浏览器下使用 WebRTC 实现 1 对 1 实时互动音视频系统最核心的类。你可以认为它是一个总的接口类或者称它为聚合类，而该类中实现的很多功能都是由其他类具体实现的。

像我前面讲的很多文章，都是 RTCPeerConnection 类的一部分功能，如：

[《06 | WebRTC 中的 RTP 及 RTCP 详解》](#)讲的是底层网络数据传输协议与其控制协议。

[《07 | 你竟然不知道 SDP ? 它可是 WebRTC 的驱动核心 ! 》](#)和 [《08 | 有话好商量，论媒体协商》](#)讲的是 SDP 协议及使用 SDP 协议进行媒体协商的过程等。

[《09 | 让我们揭开 WebRTC 建立连接的神秘面纱》](#)讲的是 WebRTC 底层是如何建立连接的。

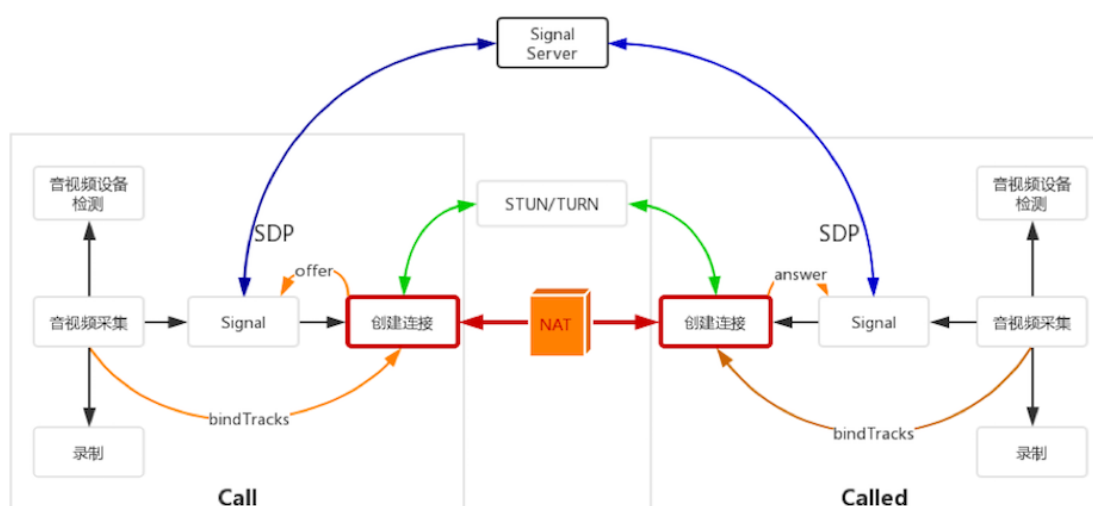
《10 | WebRTC NAT 穿越原理》介绍了在 WebRTC 底层进行 NAT 穿越的过程。

以上这些内容都是 RTCPeerConnection 类的功能。除了上述讲的这些内容之外，RTCPeerConnection 类还有许多其他的功能，我在后面的文章中还会向你逐一介绍。

RTCPeerConnection 这个知识点是你掌握 WebRTC 开发的重中之重，抓住它你就抓住了学习 WebRTC 的钥匙（这里你一定要清楚，**SDP 是掌握 WebRTC 运行机制的钥匙，而 RTCPeerConnection 是使用 WebRTC 的钥匙**），这样可以让你很快学会 WebRTC 的使用。

在 WebRTC 处理过程中的位置

还是老规矩，我们先来看一下本文在整个 WebRTC 处理过程中的位置。



WebRTC 处理过程图

通过上面这张图，你可以看到本文所要讲述的内容就是两个端点之间是如何通过 RTCPeerConnection 创建连接的。

传输要做哪些事儿

可以想像一下，如果你自己要实现一套 1 对 1 的通话系统，你会怎么做呢？如果你有一些 socket 开发经验的话，我想你首先会想到在每一端创建一个 socket，然后通过该 socket 与对端相连。

当 socket 连接成功之后，你就可以通过 socket 向对端发送数据或者接收对端的数据了。这个过程是不是看起来还是蛮简单的呢？实际上，**RTCPeerConnection 类的工作原理与 socket 基本是一样的**，不过它的功能更强大，实现也更为复杂。因为它有很多细节需要处理，这里我们从“提问题”的角度出发，反向分析，你就知道 RTCPeerConnection 要处理哪些细节了。

端与端之间要建立连接，但它们是如何知道彼此的外网地址呢？

如果两台主机都是在 NAT 之后，它们又是如何穿越 NAT 进行连接的呢？

如果 NAT 穿越不成功，又该如何保证双方之间的连通性呢？

好不容易双方连通了，如果突然丢包了，该怎么办？

如果传输过程中，传输的数据量过大，超过了网络带宽能够承受的负载，又该如何保障音视频的服务质量呢？

传输的音视频要时刻保持同步，这又该如何做到呢？

数据在传输之前要进行音视频编码，而在接收之后又要做音视频解码，但 WebRTC 支持那么多编解码器，如 H264、H265、VP8、VP9 等，它是如何选择呢？

.....

如果不是篇幅有限，我可以一直问下去哈，从中你也可以看出 RTCPeerConnection 要处理多少事情了。通过这些描述，我想你也清楚，这就是原理与真正的实际工作之间还差着十万八千里的距离呢。不过原理的好处是可以帮你简化问题，方便发现问题的本质。

什么是 RTCPeerConnection

了解了传输都要做哪些事之后，你再理解什么是 RTCPeerConnection 就比较容易了。实际上，RTCPeerConnection 就与普通的 socket 一样，在通话的每一端都至少有一个 RTCPeerConnection 对象。在 WebRTC 中它负责与各端建立连接，接收、发送音视频数据，并保障音视频的服务质量。

在操作时，你完全可以把它当作一个 socket 来用，而且还是一个具有超强能力的“**SOCKET**”。至于它是如何保障端与端之间的连通性，如何保证音视频的服务质量，又如何确定使用的是哪个编解码器等问题，作为应用者的你可以不必关心，因为所有的这些问题都已经在 RTCPeerConnection 对象的底层实现好了。

因此，如果有人问你什么是 `RTCPeerConnection`？你可以简要地回答说：“**它就是一个功能超强的 socket！**”这一下就点出了 `RTCPeerConnection` 的本质。

实现通话

今天我们要实现的例子是在同一个页面中，使两个 `RTCPeerConnection` 对象之间建立连接。它没有什么实际价值，但却能很好地证明 `RTCPeerConnection` 是如何工作的。

这里需要特别强调一点，在音视频通话中，每一方只需要有一个 `RTCPeerConnection` 对象，用它来接收或发送音视频数据。然而在真实的场景中，为了实现端与端之间的通话，还需要利用信令服务器交换一些信息，比如交换双方的 IP 和 port 地址，这样通信的双方才能彼此建立连接（信令服务器的实现可以参考[上一篇文章](#)）。

而在本文的例子中，为了最大化地减少额外的工作量，所以我们选择在同一个页面中进行音视频的互通，这样就不需要开发、安装信令服务器了。不过这样也增加了一些理解的难度，所以在阅读下面的内容时，你一定要在脑子中想象：**每一个 `RTCPeerConnection` 就是一个客户端**，这样就比较容易理解后面的内容了。

接下来我们就实操起来，一步一步实现通话吧！

1. 添加视频元素和控制按钮

我们首先开发一个简单的显示界面，在该页面中有两个 `<video>` 标签，一个用于显示本地捕获的视频，另一个用于显示“远端”的视频。


除此之外，在该页面上还有三个 `<button>` 按钮：

start 按钮，用于打开本地视频；

call 按钮，用于与对方建立连接；

hangup 按钮，用于断开与对方的连接。

具体代码如下：

 复制代码

```
1 <video id="localVideo" autoplay playsinline></video>
2 <video id="remoteVideo" autoplay playsinline></video>
```

```
3
4 <div>
5   <button id="startButton">start</button>
6   <button id="callButton">call</button>
7   <button id="hangupButton">hang up</button>
8 </div>
```

2. 适配各种浏览器


一般情况下，我都还会在显示页面中添加一个叫做 **adapter.js** 的脚本，它的作用是为各种浏览器都提供统一的、最新的 WebRTC API 接口。

在 WebRTC 1.0 规范没有发布之前，虽然各大浏览器厂商都在各自的浏览器上移植了 WebRTC，但你会发现它们最终实现的接口各不相同。这一问题直到 WebRTC 规范正式推出之后才有所改善，但很多用户依然使用老版本的浏览器，这就为使用 WebRTC 开发音视频应用增添了不少麻烦。

由于浏览器版本众多，而且用户基数大，可以预见各浏览器访问 WebRTC API 接口不统一的问题，在未来很长一段时间内会一直存在。但幸运的是，Google 很早之前就已经注意到了这个问题，因此开发了 adapter.js 这个适配器脚本，以弥补各浏览器 API 不统一的问题。


当然，你也可以自己做这件事儿，只不过适配各种浏览器版本真的是一件令人头疼的事儿，这会是一个不小的挑战。而 adapter.js 正好解决了这个痛点。随着 adapter.js 的发展，它为了解决各种各样复杂的问题，也从一小段很简单的 JavaScript 代码逐渐变得越来越庞大、复杂。但对于 adapter.js 这种即稳定、又成熟，且性能不错的库我一向是奉行“拿来主义”哈！

在页面中引入 adapter.js 的方法如下：

 复制代码

```
1 ...
2 <script src="https://webrtc.github.io/adapter/adapter-latest.js"></script>
3 ...
```

修改后的 index.html 代码如下：

 复制代码

```
1 <!DOCTYPE html>
2 <html>
3
4 <head>
5   <title>Realtime communication with WebRTC</title>
6   <link rel="stylesheet" href="css/main.css" />
7 </head>
8
9 <body>
10  <h1>Realtime communication with WebRTC</h1>
11
12  <video id="localVideo" autoplay playsinline></video>
13  <video id="remoteVideo" autoplay playsinline></video>
14
15  <div>
16    <button id="startButton">Start</button>
17    <button id="callButton">Call</button>
18    <button id="hangupButton">Hang Up</button>
19  </div>
20
21  <script src="https://webrtc.github.io/adapter/adapter-latest.js"></script>
22  <script src="js/client.js"></script>
23 </body>
24 </html>
```

通过上面的操作，界面显示及各浏览器之间适配的问题就全部完成了。接下来我们来看一下 `RTCPeerConnection` 是如何工作的。

3. `RTCPeerConnection` 如何工作

为了讲清楚 `RTCPeerConnection` 是如何工作的，我们还是看一个具体的例子吧。

假设 A 与 B 进行通信，那么对于每个端都要创建一个 `RTCPeerConnection` 对象，这样双方才可以通信，这个应该很好理解。但由于我们的例子中，通信双方是在同一个页面中（也就是说一个页面同时扮演 A 和 B 两个角色），所以在我们的 JavaScript 代码中会同时存在两个 `RTCPeerConnection` 对象，我们称它们为 `pc1` 和 `pc2` 好啦！这里你一定要注意，虽然 `pc1` 和 `pc2` 是在同一个页面中，但你一定要把 `pc1` 和 `pc2` 想像成两个端的连接对象，这样才便于对后面代码的理解。

在 WebRTC 端与端之间建立连接，包括三个任务：

为连接的每个端创建一个 `RTCPeerConnection` 对象，并且给 `RTCPeerConnection` 对象添加一个本地流，该流是从 `getUserMedia()` 获取的；


获取本地媒体描述信息，即 SDP 信息，并与对端进行交换；

获得网络信息，即 Candidate（IP 地址和端口），并与远端进行交换。

下面我们就来详细看看代码是如何实现的。

（1）获取本地音视频流

你需要调用 `getUserMedia()` 获取到本地流，然后将它添加到对应的 `RTCPeerConnection` 对象中，代码如下：

 复制代码


```
1 ...
2 // 创建 RTCPeerConnection 对象
3 let localPeerConnection = new RTCPeerConnection(servers);
4 ...
5
6 // 调用 getUserMedia API 获取音视频流
7 navigator.mediaDevices.getUserMedia(mediaStreamConstraints).
8   then(gotLocalMediaStream).
9   catch(handleLocalMediaStreamError);
10
11 // 如果 getUserMedia 获得流，则会回调该函数
12 // 在该函数中一方面要将获取的音视频流展示出来
13 // 另一方面是保存到 localStream
14 function gotLocalMediaStream(mediaStream) {
15   ...
16   localVideo.srcObject = mediaStream;
17   localStream = mediaStream;
18   ...
19
20 }
21
22 ...
23
24 // 将音视频流添加到 RTCPeerConnection 对象中
25 localPeerConnection.addStream(localStream);
26
27 ...
```

(2) 交换媒体描述信息

当 `RTCPeerConnection` 对象获得音视频流后，就可以开始与对端进行媒体协商了。整个媒体协商的过程我已经在[《08 | 有话好商量，论媒体协商》](#)一文中做了详细介绍，若记不清了，可以回看下，我们下面的实践中要用到。

并且前面我们也说了，在真实的应用场景中，各端获取的 SDP 信息都要通过信令服务器进行交换，但在我们这个例子中为了减少代码的复杂度，直接在一个页面中实现了两个端，所以也就不需通过信令服务器交换信息了，只需要直接将一端获取的 offer 设置到另一端就好了，具体步骤可以大致描述为如下。

我们首先创建 offer 类型的 SDP 信息。A 调用 `RTCPeerConnection` 的 `createOffer()` 方法，得到 A 的本地会话描述，即 offer 类型的 SDP 信息：

 复制代码

```
1 ...
2 localPeerConnection.createOffer(offerOptions)
3   .then(createdOffer).catch(setSessionDescriptionError);
4 ...
```


如果 `createOffer` 函数调用成功，会回调 `createdOffer` 方法，并在 `createdOffer` 方法中做以下几件事儿。

A 使用 `setLocalDescription()` 设置本地描述，然后将此会话描述发送给 B。B 使用 `setRemoteDescription()` 设置 A 给它的描述作为远端描述。

之后，B 调用 `RTCPeerConnection` 的 `createAnswer()` 方法获得它本地的媒体描述。然后，再调用 `setLocalDescription` 方法设置本地描述并将该媒体信息描述发给 A。

A 得到 B 的应答描述后，就调用 `setRemoteDescription()` 设置远程描述。

整个媒体信息交换和协商至此就完成了。具体代码如下：

 复制代码

```
1 // 当创建 offer 成功后，会调用该函数
```



```

2 function createdOffer(description) {
3   ...
4   // 将 offer 保存到本地
5   localPeerConnection.setLocalDescription(description)
6     .then(() => {
7       setLocalDescriptionSuccess(localPeerConnection);
8     }).catch(setSessionDescriptionError);
9
10  ...
11  // 远端 pc 将 offer 保存起来
12  remotePeerConnection.setRemoteDescription(description)
13    .then(() => {
14      setRemoteDescriptionSuccess(remotePeerConnection);
15    }).catch(setSessionDescriptionError);
16
17  ...
18  // 远端 pc 创建 answer
19  remotePeerConnection.createAnswer()
20    .then(createdAnswer)
21    .catch(setSessionDescriptionError);
22 }
23
24 // 当 answer 创建成功后，会回调该函数
25 function createdAnswer(description) {
26   ...
27   // 远端保存 answer
28   remotePeerConnection.setLocalDescription(description)
29     .then(() => {
30       setLocalDescriptionSuccess(remotePeerConnection);
31     }).catch(setSessionDescriptionError);
32
33   // 本端 pc 保存 answer
34   localPeerConnection.setRemoteDescription(description)
35     .then(() => {
36       setRemoteDescriptionSuccess(localPeerConnection);
37     }).catch(setSessionDescriptionError);
38 }

```


(3) 端与端建立连接

在本地，当 A 调用 setLocalDescription 函数成功后，就开始收到网络信息了，即开始收集 ICE Candidate。

当 Candidate 被收集上来后，会触发 pc 的 **icecandidate** 事件，所以在代码中我们需要编写 icecandidate 事件的处理函数，即 onicecandidate，以便对收集到的 Candidate 进

行处理。

为 `RTCPeerConnection` 对象添加 `icecandidate` 事件的方法如下：

 复制代码

```
1 ...
2 localPeerConnection.onicecandidate= handleConnection(event);
3 ...
```

上面这段代码为 `localPeerConnection` 对象的 `icecandidate` 事件添加了一个处理函数，即 `handleConnection`。

当 `Candidate` 变为有效时，`handleConnection` 函数将被调用，具体代码如下：

 复制代码

```
1 ...
2 function handleConnection(event) {
3
4     // 获取到触发 icecandidate 事件的 RTCPeerConnection 对象
5     // 获取到具体的 Candidate
6     const peerConnection = event.target;
7     const iceCandidate = event.candidate;
8
9     if (iceCandidate) {
10         // 创建 RTCIceCandidate 对象
11         const newIceCandidate = new RTCIceCandidate(iceCandidate);
12         // 得到对端的 RTCPeerConnection
13         const otherPeer = getOtherPeer(peerConnection);
14
15         // 将本地获到的 Candidate 添加到远端的 RTCPeerConnection 对象中
16         otherPeer.addIceCandidate(newIceCandidate)
17             .then(() => {
18                 handleConnectionSuccess(peerConnection);
19             }).catch((error) => {
20                 handleConnectionFailure(peerConnection, error);
21             });
22
23         ...
24     }
25
26 }
27 ...
```


每次 `handleConnection` 函数被调用时，就说明 WebRTC 又收集到了一个新的 Candidate。在真实的场景中，每当获得一个新的 Candidate 后，就会通过信令服务器交换给对端，对端再调用 `RTCPeerConnection` 对象的 `addIceCandidate()` 方法将收到的 Candidate 保存起来，然后按照 Candidate 的优先级进行连通性检测。

如果 Candidate 连通性检测完成，那么端与端之间就建立了物理连接，这时媒体数据就可能通过这个物理连接源源不断地传输了。

显示远端媒体流

通过 `RTCPeerConnection` 对象 A 与 B 双方建立连接后，本地的多媒体数据就被源源不断地传送到了远端。不过，远端虽然接收到了媒体数据，但音视频并不会显示或播放出来。以视频为例，不显示视频的原因是 `<video>` 标签还没有与 `RTCPeerConnection` 对象进行绑定，也就是说数据虽然到了，但播放器还没有拿到它。

下面我们就来看一下如何让 `RTCPeerConnection` 对象获得的媒体数据与 H5 的 `<video>` 标签绑定到一起。具体代码如下所示：

 复制代码

```
1 ...
2
3 localPeerConnection.onaddstream = handleRemoteStreamAdded;
4 ...
5
6 function handleRemoteStreamAdded(event) {
7     console.log('Remote stream added.');
```

```
8     remoteStream = event.stream;
9     remoteVideo.srcObject = remoteStream;
10 }
11
12 ...
```

上面代码的关键点是 **addstream** 事件。在创建好 `RTCPeerConnection` 对象后，我们需要给 `RTCPeerConnection` 的 `addstream` 事件添加回调处理函数，即 `onaddstream` 函数。也就是说，当有数据流到来的时候，浏览器会回调它，在我们的代码中设置的回调处理函数就是 **handleRemoteStreamAdded**。

当远端有数据到达时，WebRTC 底层就会调用 addstream 事件的回调函数，即 handleRemoteStreamAdded。在 handleRemoteStreamAdded 函数的输入参数 event 中，包括了远端的音视频流，即 MediaStream 对象，此时将该对象赋值给 video 标签的 srcObject 字段，这样 video 就与 RTCPeerConnection 进行了绑定。

至此，video 就能从 RTCPeerConnection 获取到视频数据，并最终将其显示出来了。

小结

在文中我向你详细介绍了 RTCPeerConnection 类，当你从不同的角度去观察它时，你会对它有不同的认知：如果你从使用的角度看，会觉得 RTCPeerConnection 是一个接口类；如果你从功能的角度看，它又是一个功能聚合类。这就是真实的 RTCPeerConnection 类。

在使用 RTCPeerConnection 时，你可以把它当作一个功能超强的 socket 使用。在它的底层，它做了很多很细致的工作，而在应用层，你不必关心这些细节，只要学会如何使用它，就可以在浏览器上轻松实现你对音视频处理的想法。

在本文的后半段，我还通过一个具体的例子向你讲解了如何使用 RTCPeerConnection 对象。通过这个例子你就知道如何在一个页面内实现音视频流的发送与接收了。

思考时间

在[上一篇文章](#)中我向你讲解了如何通过 Node.js 搭建一套信令服务器，今天我又向你讲解了如何使用 RTCPeerConnection 对象，那你是否可以将二者结合起来实现一个简单的 1 对 1 系统了呢？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

[所做 Demo 的 GitHub 链接（有需要可以点这里）](#)

从 0 打造音视频直播系统

手把手教你打造实时互动音视频直播系统

李超

新东方音视频直播技术专家
前沪江音视频架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 11 | 如何通过Node.js实现一套最简单的信令系统？

精选留言 (4)

写留言



forever

2019-08-10

老师您好，我想请教一下如何解决webrtc 兼容性的问题，特别是在ios端使用腾讯x5内核的时候

作者回复: iOS端只能用 safari，其它的浏览器都不能用 webrtc，之所以这样是因为苹果不允许其它浏览器访问底层 API，只能通过 webview实现浏览器。因此你会发现iOS上只有 safari才能访问专栏中讲解的那些API



1



Ray-J

2019-08-12

老师,Candidate的作用是用来做什么的呢?翻译过来 是候选人, 很难理解这个词,希望能得到

老师的帮助加深理解

展开 ∨



许童童

2019-08-11

读完这一讲，记住一句话就可以了。
RTCPeerConnection就是一个功能超强的 socket !



Keep-Moving

2019-08-10

老师，本节完整的代码github有吗？

展开 ∨

作者回复: 有，今天晚上会上传上去

