# 第二章基于Web的实时通信的更多技术简介

## 1.本章将向您介绍新的基于Web的实时通信（WebRTC）标准背后的技术概念。

阅读本章后，您将清楚地了解以下主题：

1）如何设置点对点通信

2）信令选项

3）API如何相互关联

3）建立通信

虽然WebRTC通信的基础是点对点的，但建立此通信的初始步骤需要进行某种协调。 这最通常由网络服务器和/或信令服务器提供。这使得两个或多个具有WebRTC能力的设备或对等体能够找到彼此，交换联系人详细信息，协商定义它们将如何通信的会话，然后最终建立在它们之间流动的直接的对等流媒体。

## 2.一般流程

有各种各样的场景，从单个设备上运行的单个网页演示到具有媒体中继和归档服务组合的复杂分布式多方会议。 要开始使用，我们将专注于最常见的流程，其中包括使用WebRTC的两个Web浏览器在它们之间建立简单的视频通话。

以下是此流程的总结：

1）连接用户

2）启动信号

3)寻找候选人

4)协商媒体会话

5)启动RTCPeerConnection流

### 2.1连接用户

这个过程的第一步是让两个用户以某种方式进行连接。最简单的选择是两个用户访问同一个网站。然后，该页面可以识别每个浏览器，并使用类似WebSocket API的方式将它们连接到共享信令服务器。 这种类型的网页通常会分配一个唯一的令牌，可以用来链接这两个浏览器之间的通信。

您可以将此令牌视为房间或会话ID。在http://apprtc.appspot.com演示如前所述，第一个用户访问http://apprtc.appspot.com，然后提供一个唯一的URL，其中包含一个新的独特令牌。 然后，该第一个用户将这个唯一的URL发送给第二个用户，并且一旦它们都同时打开此页面，则第一步完成。

### 2.2启动信号

既然这两个用户都有一个共享令牌，他们现在可以交换信令消息来协商他们的WebRTC连接的设置。 在这种情况下，“信令消息”只是任何形式的通信，帮助这两个浏览器建立和控制他们的WebRTC通信。 WebRTC标准没有准确地确定如何完成这一工作。这一点很好，因为它使这一过程的一部分开放进行创新和演进。

这也是一个挑战，因为这种不确定性往往会使RTC通信的新开发者感到困惑。之前描述的apprtc演示使用XHR和Google AppEngine Channel API的组合。这可能就像XHR轮询，服务器发送事件，WebSockets或这些的任何组合，你感到舒服。

### 2.3寻找候选人

下一步是让两台浏览器交换有关其网络的信息，以及如何联系他们。这个过程通常被描述为“找到候选人”，最后每个浏览器都应映射到可直接访问的网络接口和端口。每个浏览器都可能位于路由器后面，该路由器可能正在使用网络地址转换（NAT）将本地网络连接到互联网。他们的路由器也可能会阻止某些端口和传入连接的防火墙限制。找到通过这些类型连接的方法,路由器通常被称为NAT穿越，对于建立WebRTC通信至关重要。实现此目的的一个常见方法是使用用于NAT（STUN）服务器的会话遍历实用程序，这简单地帮助您识别如何从公共网络联系然后以有用的形式返回此信息。有很多人提供公共STUN服务器。

如果STUN服务器无法从公共互联网中找到连接到浏览器的方法，那么您不必再选择使用中继媒体的解决方案，例如Traversal Using Relay NAT（TURN）服务器。这有效地使您回到非对等架构，但在某些情况下，您处于特别严格的专用网络中，这可能是您唯一的选择。

在WebRTC中，这个整个过程通常被绑定到单个交互式连接建立（ICE）框架中，该框架处理连接到STUN服务器，然后返回到TURN服务器需要。

### 2.4协商媒体会话

既然这两个浏览器都知道如何交流，他们还必须就它们将交换的媒体（例如音频和视频）的类型和格式达成一致，包括编解码，分辨率，比特率等。这通常是使用协商的基于会议描述协议（SDP）的基于提议/答案的模型。 这被定义为JavaScript会话建立协议（JSEP）。

### 2.5启动RTCPeerConnection流

一旦这一切完成，浏览器就可以直接通过他们的对等连接或通过他们已经回到使用的任何媒体中继网关来开始流媒体。

在这个阶段，浏览器可以继续使用相同的信令服务器解决方案来共享通信来控制此WebRTC通信。 他们也可以使用特定类型的WebRTC数据通道直接相互进行。

## 3.使用WebSockets

WebSocket API使Web开发人员可以轻松地在其Web应用程序中使用双向通信。您只需使用var connection = new WebSocket（url）创建一个新的连接; 构造函数，然后在收到消息和错误时创建自己的函数来处理。 发送消息和使用connection.send（message）一样简单; 方法。

这里的主要优点是消息传递是真正的双向，快速和轻量级的。这意味着，只要它想WebSocket的API服务器可以直接发送邮件到您的浏览器，你只要他们发生时接收它们。没有延迟或不断的网络流量，因为它在XHR轮询或长轮询模型中，这使得这种设置WebRTC通信所需的报价/回答信号舞蹈的理想选择。

然后，WebSocket API服务器可以使用先前描述的唯一房间或会话令牌来确定哪些WebSocket API客户端消息应该被转发到。以这种方式，单个WebSocket API服务器可以支持非常大量的客户端。 而且由于网络连接设置很少发生，消息本身往往很小，因此所需的服务器资源非常适中。

几乎所有主要的编程语言都有WebSocket API库，由于Node.js是基于JavaScript的，因此已经成为这种类型实现的流行选择。诸如socket.io的库提供了一个很好的例子，说明这种方法真的可以容易。

## 4.其他信令选项

允许浏览器通过服务器发送和接收消息的任何方法都可用于WebRTC信令。最简单的模式是使用XHR API来发送消息并定期轮询服务器来收集任何新消息。任何Web开发人员都可以轻松实现这一点，而无需任何其他工具。然而，它有许多缺点。它具有基于每个轮询周期的频率的内置延迟。这也是浪费带宽，因为即使没有消息准备好被发送或接收，轮询周期也被重复。但是，如果你正在寻找一个很好的老式解决方案，那就是一个。

基于轮询的稍微更精细的方法被称为长轮询。在这个模型中，如果服务器还没有任何新的消息，网络连接将保持活动，直到它使用HTTP 1.1保持活动机制。当服务器有一些新的信息时，它只是将其发送到电线以完成请求。在这种情况下，轮询的网络开销减少。但是，与诸如WebSockets这样的更现代化的解决方案相比，它仍然是一种过时和低效的方法。

服务器发送事件是另一个选项。您使用var source = new EventSource（url）建立与服务器的连接; 构造函数，然后将侦听器添加到该源对象以处理服务器发送的消息。 这样就可以让服务器直接发送消息，一旦发生就收到消息。 但是您仍然使用单独的频道（如XHR）将您的消息发送到服务器，这意味着您被迫对两个独立的频道进行管理和同步。这种组合确实提供了一个有用的解决方案，已经在多个 WebRTC演示应用程序，但它与真正的双向通道（如WebSockets）没有相同的优雅。

还有各种其他创意想法可以用来促进所需的信号。 但我们所涵盖的是您最常用的选项，您会发现它被使用。

## 5.MediaStream API

MediaStream API旨在允许您从本地输入设备（如摄像机和麦克风）访问媒体流。 它最初专注于getUserMedia API或gUM，但现在已经被形式化为更广泛的媒体捕获和流API或简称为MediaStream API。然而，getUserMedia（）方法仍然是启动对本地输入设备的访问的主要方式。

每个MediaStream对象可以包含多个不同的MediaStreamTrack对象，每个对象都代表不同的输入媒体，例如来自不同输入源的视频或音频。

每个MediaStreamTrack可以包含多个通道（例如，左，右音频通道）。这些通道是由MediaStream API定义的最小单位。

MediaStream对象可以通过两种关键方式输出。 首先，它们可以用于将输出渲染到MediaElement中，例如<video>或<audio>元素（尽管后者可能需要预处理）。 其次，它们可以用于发送到RTCPeerConnection，然后可以将该媒体流发送到远程对等体。

MediaStreamTrack可以由states（）方法返回的MediaSourceStates对象描述的多个状态进行表示。 每个MediaStreamTrack还可以提供一系列功能，可以通过capabilities（）方法访问。

在顶层，MediaStream对象可以触发一系列事件，如addtrack，removetrack或者结束。 此外，MediaStreamTrack可以启动一系列事件，如启动，静音，取消静音，超限制和结束。

## 6.RTCPeerConnection API

RTCPeerConnection API是启用WebRTC的浏览器或对等体之间的对等连接的核心。要创建RTCPeerConnection对象，请使用var peerconnection = RTCPeerConnection（configuration）; 构造函数。配置变量包含至少一个名为iceServers的密钥，它是一组URL对象，其中包含有关STUN信息的信息，也可能包含在查找候选阶段期间使用的TURN服务器。

然后，根据您是呼叫者还是被呼叫者，每个客户端以稍微不同的方式使用对等连接对象。来电者的流程，以下是创建对等连接对象后调用方流程的摘要：

1）注册onicecandidate处理程序。

2）注册onaddstream处理程序。

3）注册消息处理程序。

4）使用getUserMedia访问本地摄像机。

5）JSEP提供/回答过程。

### 6.1注册onicecandidate处理程序

首先，您注册一个专用处理程序，它将任何ICE候选项发送给另一个对等体，因为它们使用之前描述的信令通道之一接收。

### 6.2注册onaddstream处理程序

然后，您注册一个从远程对等体收到视频流的onaddstream处理程序。

### 6.2注册消息处理程序

您的信令通道还应该有一个注册的处理程序来响应从另一个对等体接收到的消息。如果消息包含RTCIceCandidate对象，则应使用addIceCandidate（）方法将其添加到对等连接对象。如果消息包含RTCSessionDescription对象，则应使用setRemoteDescription（）方法将它们添加到对等连接对象。

### 6.3使用getUserMedia访问本地摄像机

然后，您可以使用getUserMedia（）设置本地媒体流，并在本地页面上显示，并使用addStream（）方法将其添加到对等连接对象。

### 6.4JSEP提供/回答过程

现在,您可以使用createOffer()方法开始协商，并注册接收RTCSessionDescription对象的回调处理程序。 此回调处理程序应该使用setLocalDescription（）将此RTCSessionDescription添加到对等连接对象。 然后最后，它也应该通过您的信令通道将此RTCSessionDescription发送到远程对等体。

## 7．被叫方的流

以下是被调用者的流程的总结，这在很多方面与呼叫者的流程非常相似，除了它响应于报价与答案：

1)注册onicecandidate处理程序

2)注册onaddstream处理程序

3)注册消息处理程序

4)使用getUserMedia访问本地摄像机

5)JSEP提供/回答过程

### 7.1注册onicecandidate处理程序

就像呼叫者一样，您首先注册一个专用处理程序，该处理程序使用之前描述的信令通道之一，将任何ICE候选项发送到另一个对等体。

### 7.2注册onaddstream处理程序

然后，像呼叫者一样，您注册一个从远程对等体收到的显示视频流的onaddstream处理程序。

### 7.3注册消息处理程序

与呼叫者一样，您的信令通道也应该有一个注册的处理程序，以响应从另一个对等体接收到的消息。 如果消息包含RTCIceCandidate对象，则应使用addIceCandidate（）方法将其添加到对等连接对象。 如果消息包含RTCSessionDescription对象，则应使用setRemoteDescription（）方法将它们添加到对等连接对象。

### 7.4使用getUserMedia访问本地摄像机

然后，像调用者一样，您可以使用getUserMedia（）设置本地媒体流，并在本地页面上显示，并使用addStream（）方法将其添加到对等连接对象。

### 7.5JSEP提供/回答过程

在这里，您不同于呼叫者，您可以通过将remoteDescription传递给createAnswer（）方法并注册接收RTCSessionDescription对象的回调处理程序来协商。 此回调处理程序应该使用setLocalDescription（）将此RTCSessionDescription添加到对等连接对象。 最后，它也应该通过您的信令通道将此RTCSessionDescription发送到远程对等体。同样重要的是要注意，在从主叫方收到报价之后，所有的被叫方都是启动的。

## 8.RTCPeerConnection在哪里进行？

下图显示了www.WebRTC.org网站的整体WebRTC架构。 它显示了隐藏在RTCPeerConnection API下方的复杂程度。

图 1 总体架构图

...

The web

Web API (Edited by W3C WG)

**WebRTC**

WebRTC C C++ API (PeerConnection)

Session management / Abstract signaling (Session)

**Voice Engine**

iSAC / iLBC Codec

**Video Engine**

VP8 Codec

**Transport**

SRTP

Your browser

NetEQ for voice

Video jitter buffer

Multiplexing

Echo Canceler / Noise Reduction

Image enhancements

P2P

STUN +TURN+ ICE

**Audio Capture/Render**

**Video Capture**

**Network I/O**

API for web developers

API for browser makers

Overrideable by browser makers

## 9.RTCDataChannel API

除了使用WebRTC在对等体之间发送媒体流之外，还可以使用DataChannel API发送任意数据流。 虽然很多人通常将其称为RTCDataChannel API，但它更准确地定义为WebRTC DataChannel API，并通过使用var datachannel = peerconnection.createDataChannel（label）创建; 构造函数。 它是一个非常灵活和强大的解决方案，通过send（）方法和onmessage事件被专门设计为与WebSocket API相似。

在撰写本章时，该API仍然处于不断变化的状态，不同的浏览器实现仍然在努力与标准化。

## 小结

您现在应该清楚地了解组合使WebRTC工作的各种API和协议。

在本书的其余部分中，我们将在将这些概念应用于现实世界的示例中更详细地探索MediaStream，RTCPeerConnection和RTCDataChannel API。首先，我们将首先将简单的点对点视频通话场景整理为完全正常的应用程序。然后，我们将探讨如何将其简化为仅限音频通话或扩展与基于文本的聊天和文件共享。最后，我们将探讨基于电子学习和团队沟通的两个现实世界的应用场景。

# 第三章 创建实时视频通话

## 1.本章介绍如何使用MediaStream和RTCPeerConnection API

在两个人之间创建一个工作的点对点视频聊天应用程序。 阅读本章后，您将明白：

1）使用Web服务器连接两个用户

2）设置用于对等呼叫的信令服务器

3）呼叫者的浏览器如何创建连接

4）受访者的浏览器如何响应

5）预览本地视频流

6）建立和呈现远程视频流

7）流处理类型

8）将其扩展到Chatroulette应用程序

## 2.设置简单的WebRTC视频通话

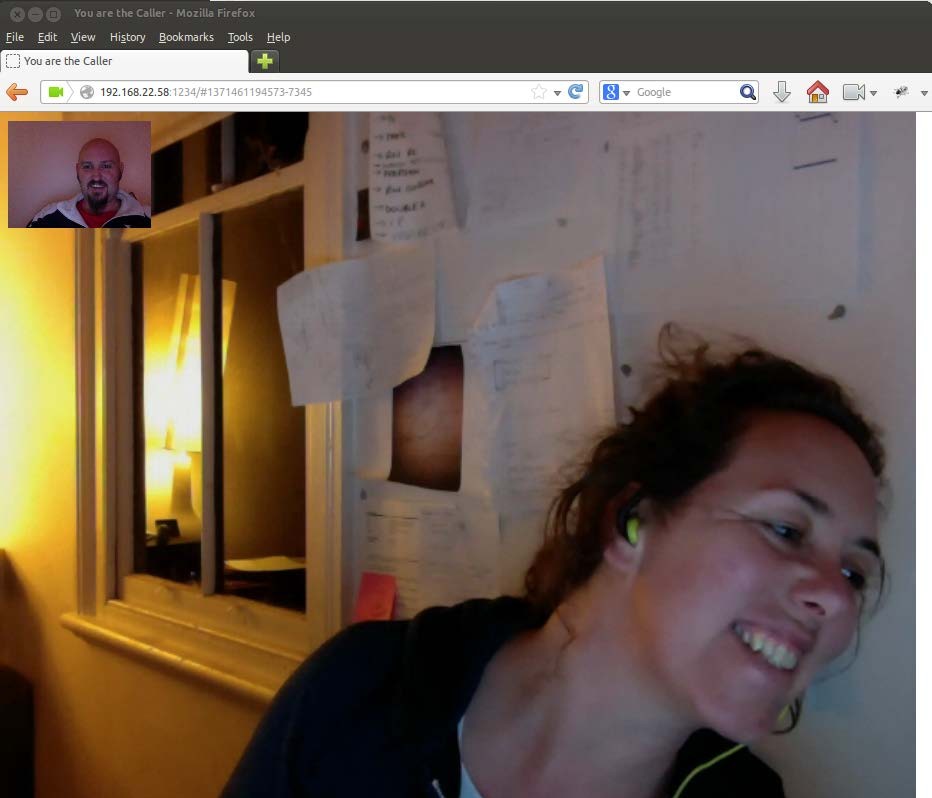
最常见的WebRTC示例应用程序涉及在两个单独的用户之间设置视频通话。在几秒钟之内，您可以轻松地查看和交流世界上任何拥有十亿个或更多WebRTC浏览器之一的任何人。我们来详细了解如何实现这一点，并在我们走的时候创建我们需要的代码。

在本书中，将使用一些简单的编码约定来协助沟通和可读性。W3C和其他标准定义组织标准化的JavaScript API将使用传统的骆驼案例格式。为本书定义的函数和变量将使用所有小写字符串，并用下划线替换单词或空格。

本示例应用程序中的Web和WebSocket服务器功能将使用JavaScript和Node.js实现。提供有关如何安装和配置Node.js的信息超出本书的范围，但您可以在找到所需的所有信息。

但是，本书确实为您提供了完整描述的工作Node.js示例代码，它提供了运行演示应用程序所需的所有功能。

图 2 使用WebRTC进行基本的点对点视频通



## 3.使用Web服务器连接两个用户

第一步只是使用Web连接两个不同的用户。 我们首先创建一个包含DOCTYPE定义，文档头和文档正文的标准HTML5网页：

<!DOCTYPE html>

<html>

<head>

…

</head>

<body>

…

</body>

</html>

然后，文档头中的第一个元素是webrtc\_polyfill.js脚本，包含一对<script>标签之间的内联。 webrtc\_polyfill.js代码正是它所说的，它旨在使JavaScript易于编写适用于WebRTC和MediaStream API的所有常见浏览器实现JavaScript。 这是一个如何工作的概述。

首先，我们为其主要功能设立了六个全球占位符：

var webrtc\_capable = true;

var rtc\_peer\_connection = null;

var rtc\_session\_description = null; var get\_user\_media = null;

var connect\_stream\_to\_src = null;

var stun\_server = "stun.l.google.com:19302";

然后根据检测到的浏览器功能的类型，填充这些全局占位符的最终值。

如果您在基于WebKit的浏览器（如Chrome）中使用早期WebRTC实现，则rtc\_peer\_connection是指向标准RTCPeerConnection，

mozRTCPeerConnection（如果使用早期的Firefox WebRTC实现）或webkitRTCPeerConnection的指针。rtc\_session\_description也是指向RTCSessionDescription构造函数的浏览器特定实现的指针。 为此，唯一真正的例外是早期的Firefox WebRTC实现。

get\_user\_media非常相似。 它是指向标准导航器的指针。getUserMedia，navigator.mozGetUserMedia如果您在Firefox中使用早期的MediaStream API实现，或者如果您在基于WebKit的浏览器（如Chrome）中使用早期的MediaStream API实现，则可以使用navigator.webkitUserMedia。

connect\_stream\_to\_src是接受对MediaStream对象的引用和对HTML5 <video>媒体元素的引用的函数。然后将流连接到<video>元素，以便可以在本地浏览器中显示。

最后，stun\_server变量包含指向Google公共STUN服务器的指针。目前，Firefox要求这是一个IP地址，但Chrome支持基于DNS的主机名和端口。

然后，浏览器检测的核心是在一组简单的if / else块中处理。

首先，它检查是否支持标准navigator.getUserMedia，否则它检查是否支持navigator.mozGetUserMedia（例如早期的Firefox MediaStream API），否则如果支持navigator.webkitGetUserMedia（例如，早期的WebKit浏览器MediaStream API ）。

最后的else块假设这是一个不支持getUserMedia的浏览器。此代码还假设如果以某种方式支持getUserMedia，则也隐式支持匹配的RTCPeerConnection API。

connect\_stream\_to\_src函数根据检测到的浏览器类型稍作调整。默认的标准版本将media\_stream直接分配给视频元素的.srcObject属性：

connect\_stream\_to\_src = function(media\_stream, media\_element)

{ media\_element.srcObject = media\_stream;

media\_element.play();

};

在早期的Firefox WebRTC实现中，<video>媒体元素使用mozSrcObject属性，可以将媒体流对象直接分配给它：

connect\_stream\_to\_src = function(media\_stream, media\_element) { media\_element.mozSrcObject = media\_stream; media\_element.play();

};

在早期基于WebKit的WebRTC实现中，webkitURL。createObjectURL函数传递媒体流对象，然后将来自该对象的响应直接分配给<video>元素的.src属性：

connect\_stream\_to\_src = function(media\_stream, media\_element) { media\_element.src = webkitURL.createObjectURL(media\_stream);

};

一旦webrtc\_polyfill.js设置了一切，我们需要创建独立于浏览器的WebRTC代码; 然后我们可以移动到这个视频通话应用程序的主体上。定义basic\_video\_call.js浏览器端逻辑的代码包含在另一对<script> </script>标签内。

首先，我们设置了我们将在其余代码中使用的一般变量。call\_token变量是将两个用户链接在一起的唯一ID。它用于确保通过信令服务器的任何信号仅在这两个特定用户之间交换。

var call\_token; // unique token for this call

signaling\_server是一个变量，表示与呼叫者和被调用方连接的信令服务器的WebSocket API连接：

var signaling\_server; // signaling server for this call

peer\_connection变量表示将在这两个用户之间建立的实际RTCPeerConnection：

var peer\_connection; // peerconnection object

接下来，我们设置了一个基本的start（）函数，

body.onload事件：

function start() {}

此功能基本上可以检测您是呼叫者还是被叫者，然后设置相关功能来匹配。 它还设置了呼叫者和被调用者使用的许多常见功能。

这里的第一步是使用由webrtc\_polyfill.js设置的rtc\_peer\_connection构造函数，使用真实的RTCPeerConnection对象填充peer\_connection变量。 我们将配置对象传递给定义要使用的STUN服务器的此函数。 在这个例子中，我们使用了由Google提供的公共STUN服务器; 但是，这只是为了演示的目的。如果你打算建立一个商业应用程序，你必须找到一个商业的STUN提供商。

// create the WebRTC peer connection object peer\_connection = new rtc\_peer\_connection({

"iceServers": [

{ "url": "stun:"+stun\_server }, // stun server info

]

});

接下来，我们为peer\_connection.onicecandidate事件处理程序设置自己的函数，如果ice\_event包含候选项，则我们将其序列化为JSON blob，并通过signaling\_server变量将其发送到另一个调用者的浏览器：

// generic handler that sends any ice candidates to the other peer peer\_connection.onicecandidate = function (ice\_event) {

…

};

然后,我们为peer\_connection.onaddstream处理程序设置自己的函数。这只是接收任何新的传入视频流并将其连接到本地<video>元素在本地浏览器中,所以您可以看到并听到呼叫的另一端的人。

// display remote video streams when they arrive peer\_connection.onaddstream = function (event) {

…

};

之后，我们使用WebSocket API设置与信令服务器的连接。 这是通用的，因为呼叫者和被叫方都使用相同类型的连接。在这个基本例子中，两者都必须连接到相同的信令服务器。

// setup generic connection to the signaling server using the WebSocket API

signaling\_server = new WebSocket("ws://localhost:1234");

现在，所有的通用功能都已经设置好了，我们可以根据您是来电者还是被呼叫者进行自定义。这通过检测浏览器是否加载了具有call\_token哈希片段的页面来完成。

如果您是来电者，那么您是第一个访问该页面的人，而您根本没有call\_token。在这种情况下，我们将为您创建一个，并设置location.hash，以便您可以在浏览器的位置栏（例如http://localhost：1234＃1370439693969-433）中看到这一点。重要的是要注意，本地主机应该被替换为您实际打算使用的主机名或IP地址，并且还必须能够访问有意加入该呼叫的其他人。

然后，您可以将此URL发送给另一个人（通过电子邮件，短信，载体鸽或任何最适合您的方法）。一旦他们加载了这个URL，我们就会检测到他们已经定义了一个call\_token哈希片段，然后将它们视为被叫方。

if (document.location.hash === ""

|| document.location.hash === undefined) { // you are the Caller

…

} else { // you have a hash fragment so you must be the Callee

...

}

}

在start（）函数之后，我们定义了一些其他通用处理函数的详细实现，这些函数由调用者和被调用者中的一个或两者使用。

首先，我们实现了处理JSEP提供/回答过程中设置的任何新描述的功能。这将在以下代码段中更详细地描述：

// handler to process new descriptions

function new\_description\_created(description) {

…

}

然后，我们从呼叫者的角度实现处理从信令服务器接收的所有信号的功能。这可以处理四个关键场景：

1.如果signal.type被callee\_arrived，那么我们启动JSEP提供/回答过程。 这在下面的代码片段中有更详细的描述。

2.如果signal.type是new\_ice\_candidate，那么这个候选项被添加到

peer\_connection。

3.如果signal.type是new\_description，那么我们调用peer\_connection。setRemoteDescription（）。

4.或者你可以用自己的定制信号来扩展它。这是处理这四种情况的功能：

// handle signals as a caller

function caller\_signal\_handler(event) { var signal = JSON.parse(event.data);

if (signal.type === "callee\_arrived") {

...

} else if (signal.type === "new\_ice\_candidate") {

...

} else if (signal.type === "new\_description") {

...

} else {

// extend with your own signal types here

}

}